INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

RESPONSE DATA COMPACTION IN BIST UNDER GENERALIZED MERGEABILITY BASED ON SWITCHING THEORY FORMULATION AND UTILIZING A NEW MEASURE OF FAILURE PROBABILITY

By

Jing Yi Liang, B.A.Sc.

A Thesis Submitted to the School of Graduate Studies and Research
in Partial Fulfillment of the Requirements for the Degree of

Master of Applied Sciences
in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
(Electrical and Computer Engineering)

Faculty of Engineering
University of Ottawa

September, 2000

To my mother, my father and my brother

ACKNOWLEDGEMENTS

I would like to express my sincerest appreciation and gratitude to my thesis advisor, Dr. Sunil R. Das, Professor, School of Information Technology and Engineering, University of Ottawa for his guidance and support throughout this research.

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

Thanks are also due to Dr. Dong S. Ha of the Department of Electrical Engineering at the Virginia Polytechnic Institute and State University, Blacksburg, VA, U.S.A. for kindly providing us with ATALANTA and FSIM fault simulators and to Dr. S. M. Reddy and Dr. I. Pomeranz, Faculty of Engineering, University of Iowa, Iowa City, IA, U.S.A. for providing us with COMPACTEST which greatly helped us in our research.

Finally, I owe special thanks to my parents and my brother for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Built-in self-testing (BIST) is a design approach that provides the capability of solving many of the problems otherwise encountered in testing today's highly sophisticated digital circuits and systems. A typical BIST environment utilizes a test pattern generator that sends its outputs to a circuit under test (CUT) and the output streams from the CUT are fed into a test data analyzer. A fault is detected if the circuit response is different from that of the fault-free circuit. The test data analyzer is composed of a response compaction unit together with storage for the fault-free responses from the CUT and a comparator. In general, s responses coming out of the CUT are first fed into a space compressor, providing in general, only one sequence at the output which eventually is fed into a time compaction unit to extract the CUT signatures. The present thesis deals with the general problem of designing and analyzing efficient space compression techniques for built-in self-testing of VLSI circuits using compact test sets. The techniques are based on identifying certain inherent properties of the test data responses of the CUT along with the knowledge of nonoccurrence of failure probabilities. To that effect, generalized mergeability criteria are developed in the thesis that utilize the well known switching theory concepts of Hamming distance, cover table, and frequency ordering of literals in conjunction with those of sequence weights (first-order and Nth-order) and derived sequences. The thesis also explores the effect on sequence mergeability under constraints of stochastic independence of multiple line errors and its outcome on the fault coverage. Extensive simulation experiments on ISCAS 85 combinational benchmark circuits with FSIM, ATALANTA, and COMPACTEST programs indicate that the proposed techniques, both under stochastic independence and dependence of line errors, achieve a relatively high fault coverage for single stuck-line faults with low CPU simulation time and acceptable area overhead for the designed compactors. The subject thesis also rates the performance of the designed compactors with that of the conventional linear parity tree space compactors.

Chapter 1

Introduction

With increasing complexity in systems design with increased levels of integration densities in digital design, better and more effective methods of testing to ensure reliable operations of chips, mainstay of today's many sophisticated systems, are required [5, 9 20]. The concept of testing has a broad applicability, and finding highly efficient testing techniques that ensure correct system performance has assumed significant importance. The conventional testing technique of digital circuits requires application of test patterns generated by a test pattern generator (TPG) to the circuit under test (CUT) and comparing the responses with known correct responses. However, for large circuits, because of higher storage requirements for the fault-free responses, the test procedure becomes rather expensive and thus alternative approaches are sought to minimize the amount of needed storage. Built-in self-testing (BIST) is a design approach that provides the capability of solving many of the problems otherwise encountered in testing digital systems. It combines concepts of both built-in test (BIT) and self-test (ST) in one termed built-in self-test (BIST). In BIST, test generation, test application and response verification are all accomplished through built-in hardware, which allows different parts of a chip to be tested in parallel, reducing the required testing time besides eliminating the need for external test equipment. A typical BIST environment uses a test pattern generator (TPG) that sends its outputs to a circuit under test (CUT) and output streams from the CUT are fed into a test data analyzer. A fault is detected if the circuit response is different from that of the fault-free circuit. The test data analyzer is comprised of a response compaction unit (RCU), a storage for the fault-free response of the CUT and a comparator.

In order to reduce the amount of data represented by the fault-free and the faulty CUT responses, data compression is used to create signatures from the CUT and its corresponding fault-free circuit. BIST techniques use pseudoexhaustive or even exhaustive test patterns or sometimes on-chip storing of reduced test sets. The response compaction unit can be divided into: a space compression unit and a time compression

unit. In general, s responses coming out of a CUT are first fed into a space compressor, providing t output streams such that $t << s$. Most often, test responses are compressed into only one sequence ($t = 1$). Space compression brings a solution to the problem of achieving high quality built-in self-testing of complex circuits without the necessity of monitoring a large number of internal test points, thereby reducing both testing time and area overhead by merging test sequences coming from these internal test points into a single bit stream. This single bit stream of length r is next fed into a time compressor and eventually a shorter sequence of length q, $q < r$ is obtained at the output.

The extra logic representing the compression circuit must be as simple as possible, to be easily embedded within the CUT, and should not introduce signal delays to affect either the test execution time or the normal functionality of the CUT. Besides, 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. In addition, signatures derived from faulty output responses and their corresponding fault-free signatures should not be the same, which unfortunately is not always the case. A fundamental problem with compression techniques is hence error masking or aliasing [10], which occurs when the signatures from faulty output responses map into fault-free signatures. Aliasing causes loss of information, which in turn affects the test quality of BIST and reduces the fault coverage. Several methods have been proposed in the literature for computing the aliasing probability of which the exact computation is known to be NP-hard.

This thesis considers the general problem of designing and analyzing efficient space compaction techniques for built-in self-testing of VLSI circuits using compact test sets. The techniques are based on identifying certain inherent properties of the test output responses of the CUT and the knowledge of failure probabilities. The mergeability criteria of output sequences are developed utilizing the concepts of Hamming distance and sequence weights for a pair of outputs as well as an arbitrary number of outputs (generalized mergeability) and the effect of failure probabilities on the mergeability criteria is analyzed as well. The techniques proposed achieve a very high fault coverage for single stuck-line faults, with low CPU simulation time, and acceptable hardware overhead, as evident from extensive simulation results on ISCAS 85 combinational

benchmark circuits, under condition of both stochastic independence and dependence of single and multiple line errors.

1.1 Testing Concepts

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

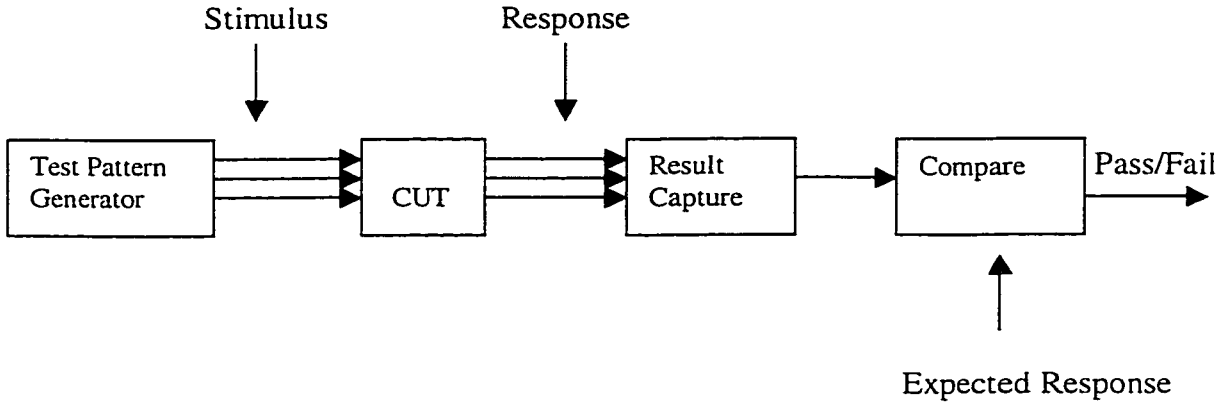


Figure 1.1 Digital logic circuit testing setup.

Testing can be carried out at different stages of the manufacturing process. The sooner a defect is detected, the lower is the cost. For instance, the approximate cost to a company for detecting a fault at the various levels may be summarized as follows [58]:

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

In general, testing could be of two basic types: functional testing and manufacturing testing [57]. Functional tests are used to validate the correct operation of a system with respect to its functional specifications. They are usually implemented early in the design cycle to verify the functionality of the CUT. Functional tests are said to be *exhaustive* if they could detect almost any fault whatsoever. However, exhaustive testing be better used in small circuits because of the length of the resulting tests. *Pseudoexhaustive* tests could be significantly shorter than exhaustive tests and could be used if some prior information about the structure of the circuit is available, thereby narrowing the universe of search space.

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

1.1.1 Logical fault models

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

In order to determine the existence of the aforesaid faults in the circuits, a precise fault model is required. A fault model basically is the representation of the effect of a failure by means of the change that is produced in the system signals [33, 40]. The fault models help generate tests and evaluate test quality defined in terms of coverage of the modeled faults. The most popular model is the single stuck-at-fault model (i.e. stuck-at-one and stuck-at-zero). Other fault models include stuck-open or close, bridging-fault or multiple-fault models. In most cases, the single stuck-at-fault model is extensively used because of its simplicity and accurate representation of a large class of logic faults.

1.1.2 Observability and controllability

When a designer or tester desires to measure the output of a gate within a large circuit to verify if it works correctly, and to assess the degree of difficulty of testing a particular signal within a circuit, observability and controllability become significantly important. The concepts of controllability and observability are defined in [55] as:

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

1.1.3 Test evaluation

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

1.1.4 Test generation for combinational circuits

Test generation is a process to determine a test for a given fault in a given circuit. Test generation could be done off-line or concurrently. In the former case, the test vectors can be generated off-line either manually or by a test pattern generation program and then stored in an on-chip ROM. However, it is not widely utilized in practice, since it needs a very large ROM. In this thesis we will mainly be concerned with concurrent test pattern generation, where the test vectors can be generated while they are being applied.

Logic circuits can be tested by using exhaustive testing, which applies all the 2^n input combinations to an n-input combinational circuit and detecting all detectable faults that do not convert the circuit to a sequential circuit. This method is straightforward and can obtain a high fault coverage with proper design, and do not need fault modeling. However, it is impractical for systems with a large number of inputs (i.e. n greater than

about 25) unless the system is partitioned into subsystems with fewer than 25 inputs, a procedure usually called pseudoexhaustive testing [41].

Random testing [7] is another kind of test generation method where the test patterns are generated randomly or pseudorandomly. The cost of random testing is usually less than that of deterministic testing; however, random testing may require a long testing time and evaluation of fault coverage by fault simulation [3]. In order to achieve a high fault coverage, in general, it is necessary to use high random test length. The IBM RISC System/6000 processor uses random test patterns [45].

Exhaustive testing, pseudoexhaustive testing and pseudorandom testing are circuit-independent test generation methods. We will discuss about circuit-dependent test generation methods later. The circuit-dependent test generation methods are of two categories: fault-oriented and fault-independent (or critical-path). Fault-oriented methods are to generate a test for a specified target fault, as done in D-algorithm [47], PODEM [28] and FAN [26]. Fault-independent methods derive a set of tests in order to detect a large set of single stuck-at faults without targeting individual faults.

The D-algorithm uses the propagation, justification, and implication procedures to find a test pattern for a fault. Propagation is the procedure to move the fault effect forward from the fault site to a primary output by creating a sensitized path. Justification is to find a primary input pattern to obtain a specified signal value at an internal node. Implication refers to the process of computing all values of signal lines that can be uniquely determined from the assigned signal lines in the circuit.

D-algorithm uses a five-valued logic consisting of the states 1, 0, D, $\bar{D}$ and X. The entities 0 and 1 represent logical zero and logical one, respectively, while X represents the unknown or DO NOT CARE state. D represents a logic 1 in a good circuit and a logic 0 in a faulty circuit while $\bar{D}$ represents a logic 0 in a good one and a logic 1 in a fault one. In basis, D-algorithm starts by propagating the D value on an internal node to a primary output. Once the D value is observable at a primary output, then the primary input values that are required to enable the fault to be observed and tested are determined. Decisions are made by the propagation and justification procedures. The D-algorithm returns to an earlier decision point and makes an alternative decision, a process called

backtracking, when decisions cannot be made, at some point, without invalidating a value already assigned to a line in the circuit.

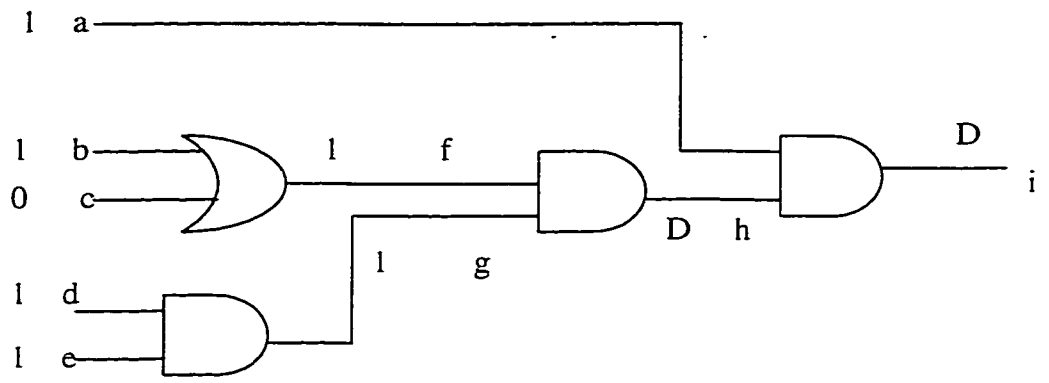


Figure 1.2 The D-algorithm-sensitization step.

Considering the circuit shown in Figure 1.2, where a stuck-at-0 fault is to be detected at node h, we can examine the use of this D-algorithm. As previously described, we first propagate D on node h to one or more primary outputs, then set node h to state D via a set of primary inputs. From node h we backtrack to the primary inputs (a, b, c, d, e) to find the necessary input vector required to set node h to 1. Thus the resultant test vector is {a, b, c, d, e} = {1,1,0,1,1} or {1,0,1,1,1}.

In an attempt to reduce the backtracking of the D-algorithm, Goel [28] developed PODEM an acronym for path-oriented decision making. PODEM test generation algorithm is characterized by a direct search process, in which decisions consist only of primary input assignments. Compared to D-algorithm, PODEM does not require justification; the number of backtracks is reduced since the decision only depends on the value of primary input; also the execution speed of PODEM is faster than that of D-algorithm.

In order to accelerate the algorithm by reducing the number of backtracks and to shorten the processing time between backtracks, the FAN (fan-out-oriented test generation algorithm) was developed by Fujiwara and Shimono [26]. FAN introduces two major extensions to the backtracking concept of PODEM [28, 26, 12]. The

backtracing in FAN may stop at internal lines rather than stopping at primary inputs. In order to simultaneously satisfy a set of objectives, FAN uses a multiple-backtrace procedure rather than trying to satisfy one objective. By decreasing the number of backtracks, FAN thus could be faster and more efficient than PODEM.

CONTEST (concurrent test generation) [1] method considers a group of faults simultaneously at the early stage of test generation when a large number of faults have not been tested. This can generate tests for multiple faults at one time to quickly reduce the number of faults to be considered.

Boolean difference method [54] is a mathematical approach to generate test sets for combinational circuits. It captures the basic concepts of path sensitization in elegant algebraic terms and provides good insights into the process of test generation.

The Boolean difference [15] is defined as being the Exclusive-OR operation between two Boolean functions, one representing the fault-free circuit while the other representing the faulty circuit. If the Boolean difference happens to be a 1, a fault in the circuit is detected. However, this method is computationally little involved because of the algebraic equations that need to be solved in order to derive a test for a give fault.

Fault-independent TG works without targeting individual faults, such as the critical-path testing algorithm. The critical-path testing algorithm is composed of two steps, i.e. assigning a selected primary output a critical 0-value or 1-value (the value of a primary output is always critical) and justifying the primary input value recursively to justify any critical value on a gate output by the critical values on the gate inputs.

Unlike a fault-oriented algorithm, a fault-independent algorithm fails to identify undetectable faults. However, this method has the following advantages compared to fault-oriented test generation:

- Without using a separate fault simulation step required in a fault-oriented algorithm. the single stuck faults detected by the generated tests are immediately known.
- A new test is produced by modifying the critical paths obtained in the previous test to avoid much duplicated efforts inherent in a fault-oriented algorithm.

1.1.5 Automatic test equipment (ATE)

Testing usually is executed by using automatic test equipment (ATE, Figure 1.3) that applies test vectors stored in a memory or generated automatically by special circuits to the CUT. Responses of the CUT to applied test vectors are evaluated by capturing and comparing them to expected values. This is done in most ATE systems by retrieving the expected values from a memory as each test vector is applied, and then comparing to the corresponding circuit output. In most cases, a list of expected values is produced using logic simulation.

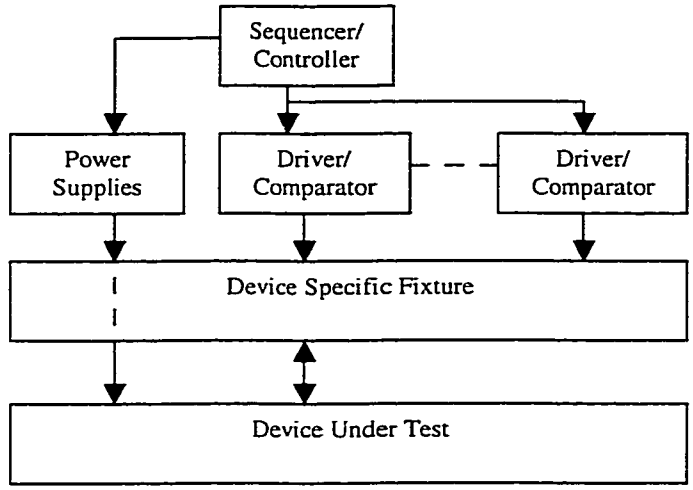


Figure 1.3. Simple model of automatic test equipment (ATE).

In ATE, each D/C unit can supply circuit stimulus, receive expected data and may have its own memory. Moreover, all D/C units work in parallel at high speeds.

However, there are problems in using ATE to test complex VLSI circuits, viz.,

- Commercial testers are very expensive.
- When the number of transistors in an IC increases with the fixed number of pins, it is difficult to activate a fault by applying appropriate stimulus at the input pins and observe an error by probing the output pins which results in reducing the quality of testing.
- Compared with the circuit's normal operating speed, the ATE usually applies test patterns at a slower speed so that the testing quality is affected again since some faults can only be detected at the system's operating speed.

- The number of test stimuli and the corresponding fault-free responses increase together with the increase in the circuit size and might become too huge to be handled efficiently by the memories in ATE.

1.1.6 Design for testability (DFT)

For the sake of reducing the cost of testing and improving the quality of test, design for testability (DFT) is often recommended in recent years. DFT techniques are design efforts specifically employed to ensure that a device is testable, and it makes the design to be closely integrated with test. Although DFT techniques solve the controllability and observability problems, at same point, the values of chip area, the number of I/O pins and circuit delay usually increase to result in increased power consumption and decreased yield.

Scan design [22] is one of the most popular structured DFT techniques used for external testing, in which the memory elements are separated from combinational circuits during testing. Usually flip-flops are chained together into a shift register during the selected testing mode, and the circuit is partitioned into combinational blocks so that the test generation can be executed easily. The tests can be scanned-in and the test responses scanned-out for response verification. Level sensitive scan design (LSSD) introduced by IBM [22, 23, 19] is a popular approach, which is based on two tenets. First, that the circuit is level sensitive. A logic system is level-sensitive if and only if, the steady state response to any allowed input state change is independent of the circuit and wire delays within the system. Also, if an input state change involves the changing of more than one input signal, then the response must be independent of the order in which they change. Steady state response is the final value of all logic gate outputs after all change activity has terminated [22,23,19]. The second principle of LSSD is that each register might be converted to a serial register. Figure 1.4 shows a typical LSSD scan system.

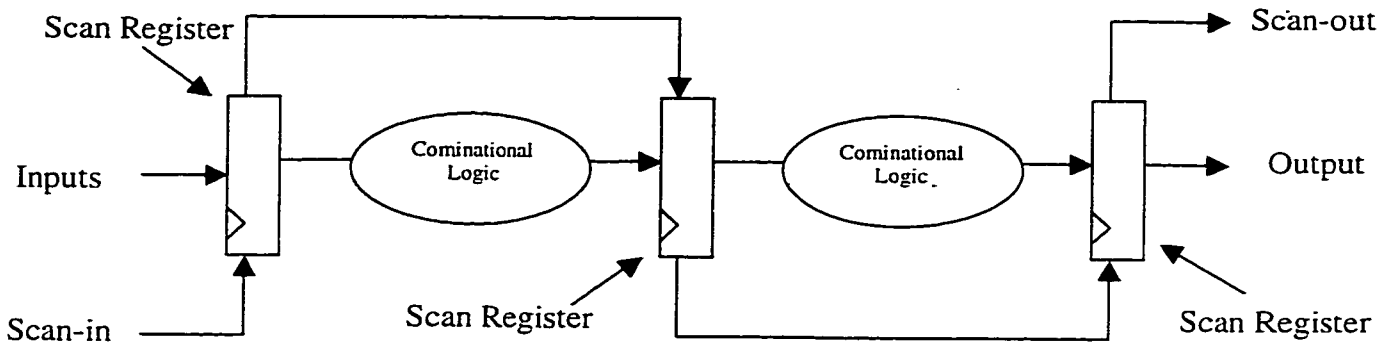


Figure 1.4 Basic architecture of an LSSD scan chain.

There are several attributes associated with the use of scan designs, such as:

- Scan designs are expensive using board or silicon area because flip-flops and latches are more complex.
- Require one or more additional I/O.
- Both the test time per pattern and the total test time for a circuit are increased.
- To avoid degradation in performance caused by the extra circuit delay, a slower clock rate is required.
- Test generation costs are reduced and fault coverage is improved.

Based on aforesaid discussions, it is obvious that DFTs which are usually used for external testing have some advantages and disadvantages as well. In the next part we will introduce the concept of BIST, a design technique which can offer superior and effective solutions for all design level (including circuit, chip, board or system level) to solve the problems encountered in testing: such as the complexity issue, the quality issue, the test generation problem as well as the test application problem.

1.2 Built-In Self-Test

Built-in self-test (BIST) is a test technique in which a circuit (chip, board, or system) can test itself; in other words, the testing (test generation, test application and response analysis) could be accomplished through built-in hardware [3]. Actually, BIST is a combination of two concepts, i.e. built-in test (BIT) and self-test (ST). This technique

is intended to solve the some of major testing problems, like time and volume [39], the test cost and diagnosis.

The automatic test generation (ATG) [48] produces the test vectors for application to the CUT and the response data from an entire test sequence are compressed into a single value called a *signature*, which is then compared to the signature of a fault-free circuit, as shown in Figure 1.5. A fault is detected if the test signature is different from the good signature. A typical BIST scheme is composed of an automatic test pattern generator, the circuit under test and a response compaction unit.

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

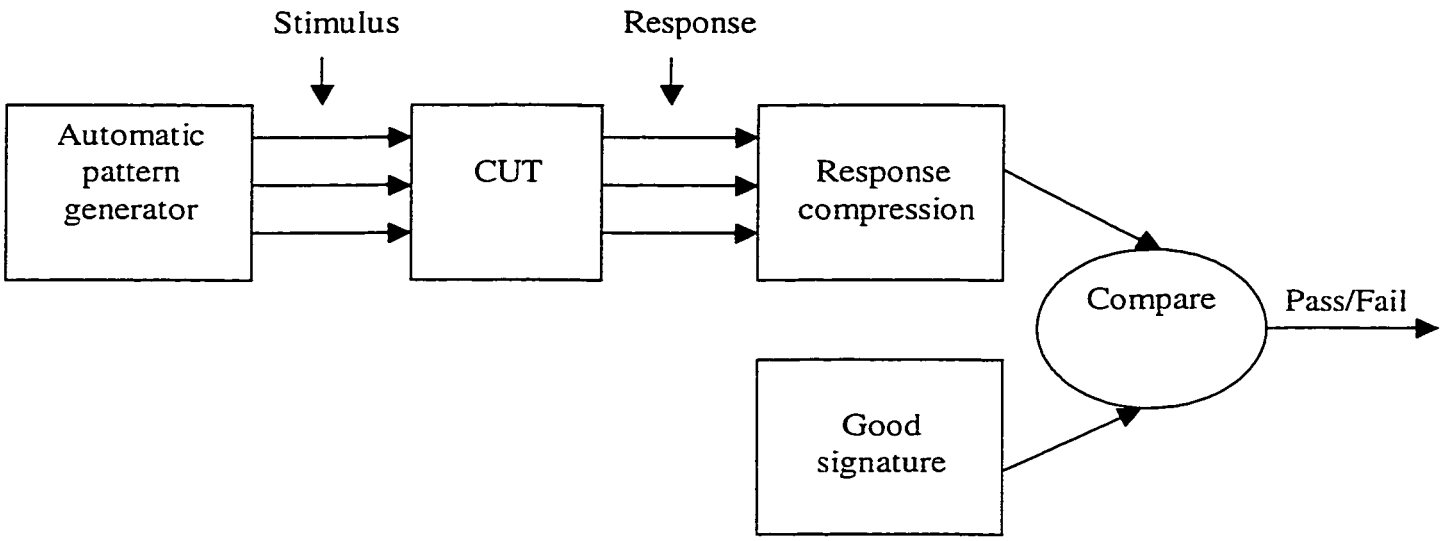


Figure 1.5 The typical BIST environment.

BIST techniques use exhaustive, pseudoexhaustive or pseudorandom test patterns because of the ease to generate on–chip. Many practical circuits need few test patterns for full coverage of single stuck faults [29, 43]. Using algorithms (such as PODEM), the reduced test sets can be generated on–chip at a low hardware cost with high fault coverage.

The autonomous linear feedback shift register (LFSR) can be used to generate pseudorandom test vectors. An ALFSR is a serial connection of D flip-flops with no external inputs and Exclusive-OR (XOR) gates providing the feedback. A four-stage ALFSR is shown in Figure 1.6.

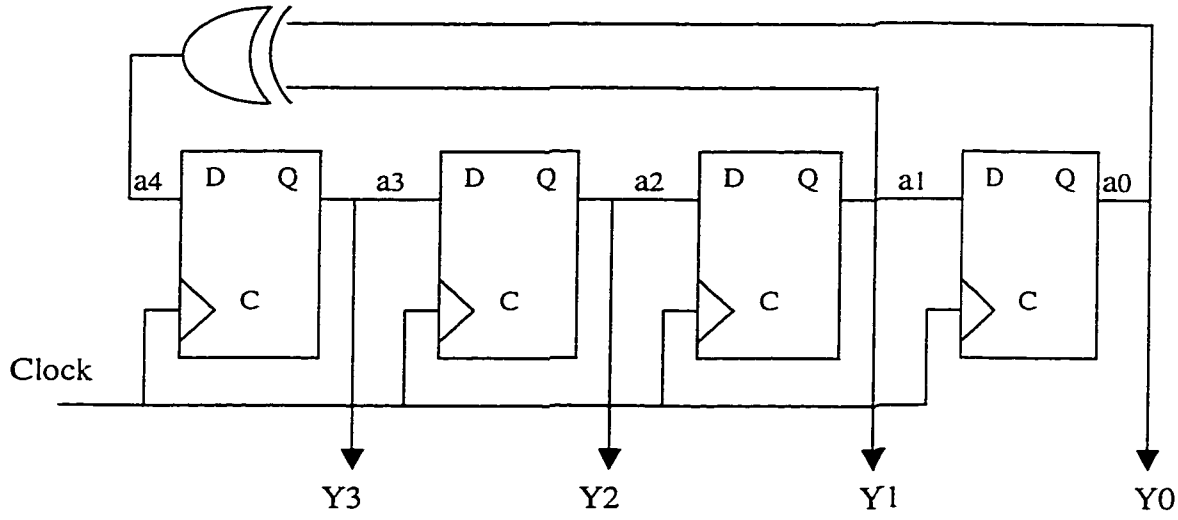


Figure 1.6 ALFSR structure with $p(x) = x^4 + x + 1$.

Comparing bit-by-bit the fault-free responses to the observed responses of the CUT is done in conventional testing methods making them unpractical and unsuitable for complex VLSI implementation, because a large amount of memory for storing the reference responses are required. To reduce the silicon area in BIST, the test responses and the corresponding responses of a fault-free CUT are compressed which are regenerated during self-test. The compacted test outcome is called the *signature*, and test verification is carried out by comparing the fault-free signature to the observed signature. The signature analysis [24, 42] is popular in BIST techniques. The response compaction is a process to reduce the test response to a signature. A typical response compaction scheme is illustrated in Figure 1.7. The compaction circuit consists of a space compactor and a time compactor. The space compactor reduces the width of the test response from P to Q (in most case Q=1). Time compaction refers to the compression of a longer bit

stream to a shorter one. In the next chapter, the details of these two types of compactions will be introduced.

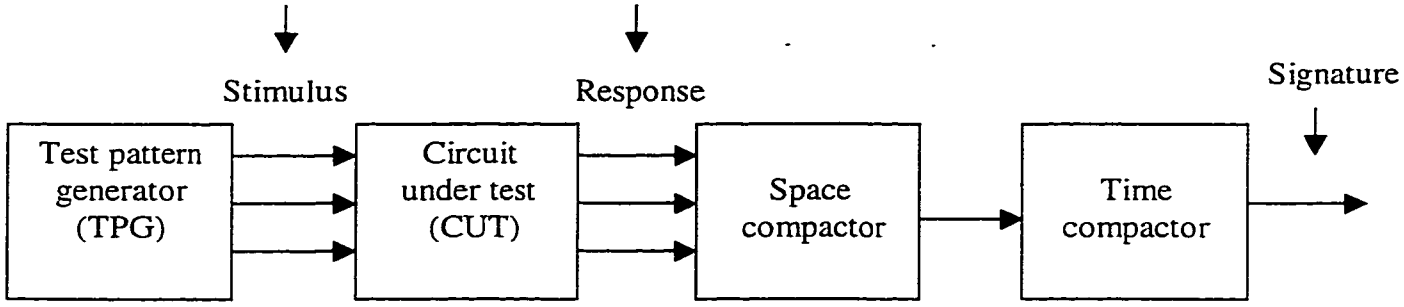


Figure 1.7 A typical test response compaction diagram.

BIST has the following advantages, in general:

- The costs of test pattern generation and fault simulation are decreased.
- The expensive external test equipment for applying test patterns and the process of monitoring the results are eliminated.
- Having all BIST circuitry on the chip itself, tests can be performed at normal operating clock rate, and so BIST techniques can detect timing related problems which may be easily missed by an external test.

Recently several companies have included BIST in their products. For example, the Motorala 68020 microprocessor [14] is designed using BIST. The Intel 80386 microprocessor [27] uses BIST for the microcode ROM. Similarly, AT&T has incorporated BIST into more than 200 of their chips [4].

The present thesis deals with design and analysis of efficient space compression techniques for built-in self-testing of VLSI circuits. The space compaction tree is designed based on some of the inherent properties of the output sequences of the circuit under test and on the knowledge of failure probabilities. The major objective has been to develop a method for space compaction which is simple, incurs low hardware overhead with minimal information loss. The mergeability criteria of output sequences are

developed and the effects of missed error probabilities on the selection and merger of sequences are investigated.

1.3 Organization of the Thesis

The thesis is organized as follows:

- Chapter 1 gives an overview of the conventional testing methods for digital combinational circuits, test generation methods and built-in self-testing.

- Chapter 2 introduces several existing space and time compaction techniques.

- Chapter 3 considers design of space compaction trees for multi-output combinational circuits assuming stochastic independence of multiple line errors based on switching theory formulation. A design algorithm is proposed and a specific example is provided for verification.

- Chapter 4 designs space compaction trees for multi-output combinational circuits assuming stochastic dependence of multiple line errors. A design algorithm is proposed here and also a specific example for verification is discussed.

- Chapter 5 provides a discussion of the experimental results on the simulation of the ISCAS 85 combinational benchmark circuits using fault simulation programs ATALANTA, FSIM and COMPACTEST.

- Chapter 6 provides a conclusion and summarizes the results presented in this thesis.

Chapter 2

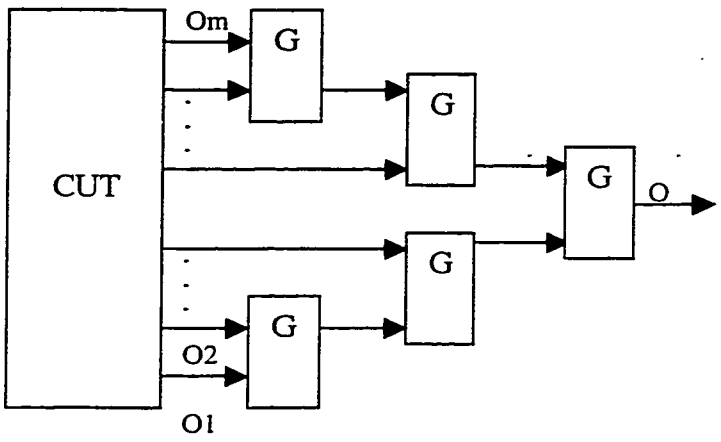
Response Compaction – An Overview

In this chapter, we review response data compaction techniques for BIST. As evident, a reliable and practical compression technique should be easily implemented by a simple circuit included in the CUT as the part of its BIST logic and should not introduce signal delays affecting either the normal behavior or the test execution time of the CUT. Moreover, the length of the test signature should be short to reduce the amount of storage required for fault-free signature. The faulty and fault-free signatures should be different; otherwise, error masking or *aliasing* will result introducing information loss and affecting test quality of BIST.

Compression techniques can be formally divided into two categories: space compression and time compression.

2.1 Space Compaction Techniques

The basic idea of space compression is to produce a limited number of signatures (usually a single signature) for multiple output circuits instead of producing an individual sequence for each output (see Figure 2.1). This results in achieving high-quality BIST of complex chips without the necessity of monitoring a large number of internal test points. It also reduces testing time and area overhead by grouping test sequences coming from a CUT into a single stream of bits. Sequence coming out of the space compressor is stored in a time compressor as a signature (Figure 1.7).



G: Logic function AND/NAND, OR/NOR, and EXOR/NEXOR

Figure 2.1 Space compaction using logic reduction network.

Space compression thus reduces the number of pins monitored by the tester as well as the memory space required for reference signature, with loss of some useful information generally leading to a reduction in fault coverage.

A brief discussion of various space compaction techniques as proposed in the literature or used in industries like parity tree compaction, hybrid space compression, dynamic space compression, modified dynamic space compression, modified hybrid space compression, quadratic functions compaction, programmable space compaction, multiplexed parity tree compaction in the contest of BIST is provided next in the following sections.

2.1.1 Parity tree compaction

The parity tree space compaction circuits [46] consist of Exclusive-OR (XOR) or Exclusive-NOR (NXOR) gates only. The parity tree circuits realize function of the form $z = z_1 \oplus z_2 \oplus \ldots \oplus z_k$. The parity tree space compactor propagates all errors that appear on an odd number of its inputs z_1, z_2, z_k; however, aliasing occurs if error appears on an even number of its inputs. If a parity tree is used for space compaction, pseudorandom or

compact test patterns can be used to detect most of the single stuck-at faults [10, 25]. A parity tree compaction scheme is shown in Figure 2.2.

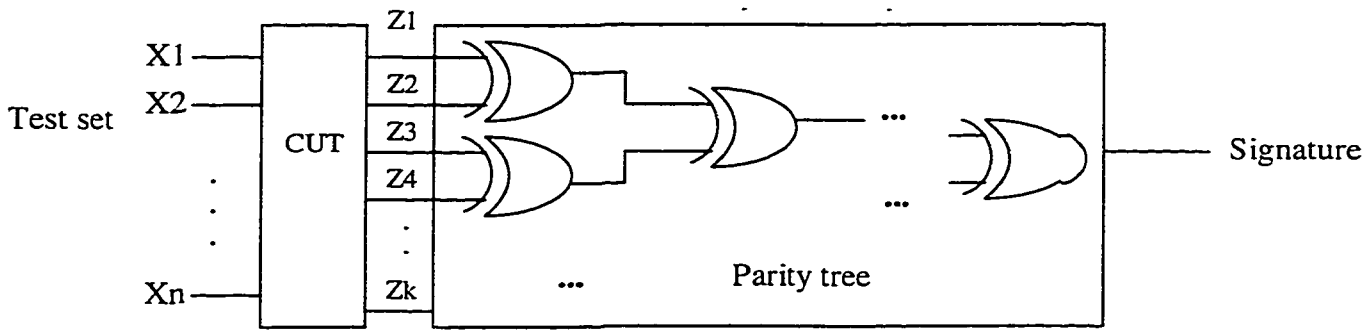


Figure 2.2 A parity tree.

2.1.2 Hybrid space compactors (HSC)

Li and Robison have proposed a space compression method called hybrid space compression (HSC) [38] which uses two-input AND, OR and XOR gates to construct a compression tree to compress the multiple outputs of the CUT into a single line, rather than using XOR/XNOR gates as an output compression tool [49]. The compaction tree is constructed based on the detectable error probability estimate $\mathcal{E}$ defined as:

$$\mathcal{E} = S_1 \frac{R_1}{2L} + S_2 \frac{R_2}{L}.$$

where

S_1 is the probability of single error felt at the output of the CUT,

S_2 is the probability of double error felt at the output of the CUT,

R_1 is the number of single line errors detected at the output of that logic gate,

R_2 is the number of double line errors detected at the output of that logic gate, and

L is the input sequence length.

The circuit realization of HSC is shown in Figure 2.3. Space compression in HSC is achieved by deriving the $\mathcal{E}$s for AND, OR and XOR gates, the gate with the maximum

$\mathcal{E}$ being selected to merge the candidate outputs. The process is repeated stage by stage until the outputs have been merged into a single line.

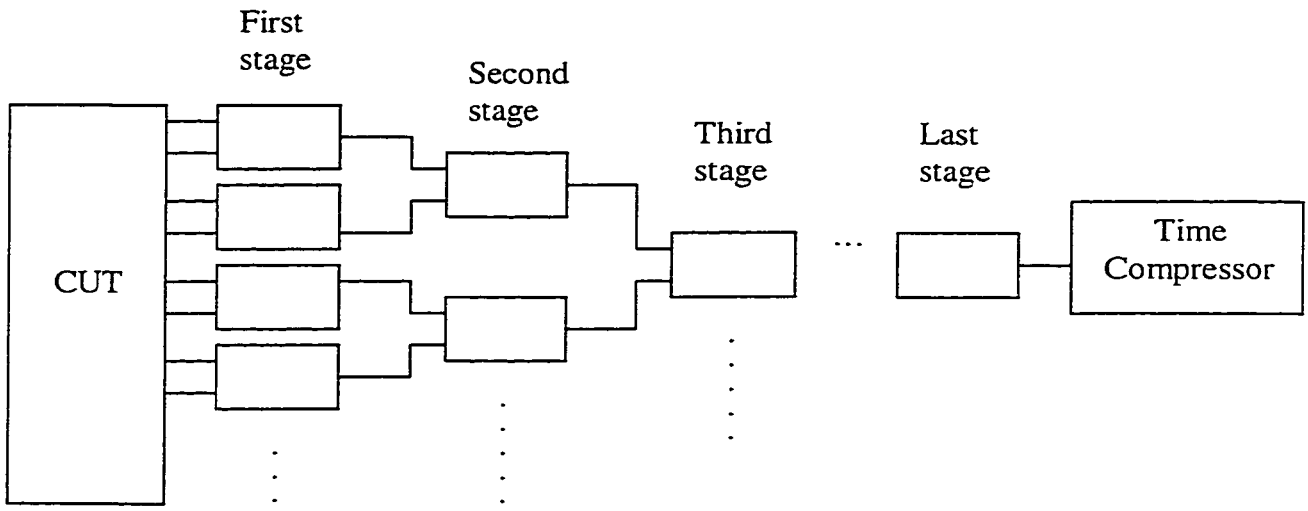


Figure 2.3 The circuit realization of HSC.

2.1.3 Dynamic space compression

Jone and Das subsequently proposed a modified method called dynamic space compression (DSC) [34], where the probabilities S_1 and S_2 are dynamically estimated based on the structure of the CUT during the entire derivation process of the space compressor, instead of fixing the values of S_1 and S_2 as in HSC. Experimental results show that the information loss in general is between 0% and 12.7%. A theory to predict the performance degradation of general space compression methods was also developed by Jone and Das in [34].

2.1.4 Modified dynamic space compression

A refinement of the DSC technique called modified dynamic space compression (MDSC) was proposed in [17]. For a CUT, a compaction tree is constructed based on its structure. In this technique, the detectable error probability estimates are calculated by

using the Boolean difference method. The output data modification [60] is used to minimize the number of faulty output data patterns, which have the same compressed form as the fault-free patterns. The compressed outputs are fed into a syndrome counter [2] to derive the signature for the circuit as shown in Figure 2.4. Experimental results indicate that the loss of information is in the range of 0% to 10%. However, the Boolean difference method used to derive the number of detectable faults in a circuit may require high storage and thus the method may not be suitable for circuits with a very high density of gates because of involved computations.

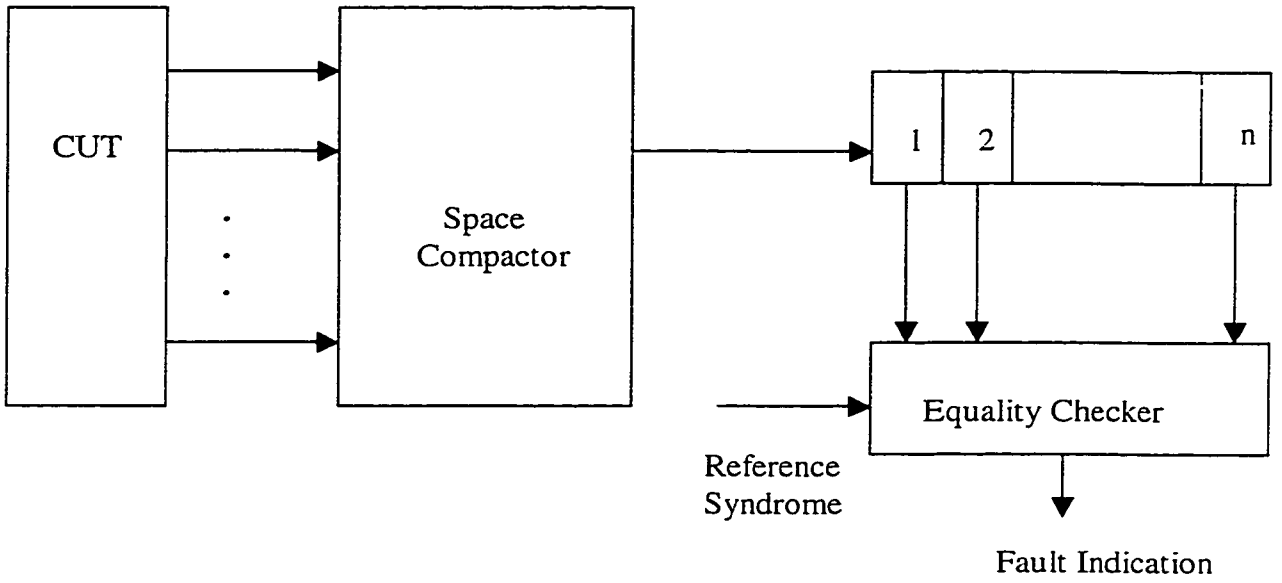


Figure 2.4 Syndrome counter used as a time compressor.

2.1.5 Modified hybrid space compaction

Using the concepts of Hamming distance and sequence weights, a new space compression technique was proposed in [5] by Das and Assaf in which two of the output sequences are merged by logic gates AND/NAND, OR/NOR, XOR/XNOR to construct a compaction tree to minimize the storage for the CUT. Figure 2.5 illustrates an outline of the proposed scheme. Some of the advantages inherent in this method are its simplicity and low hardware overhead. Besides, the method uses nonexhaustive test sets for the

compaction tree design. Later, Das and Barakat [6] modified this scheme introducing generalized mergeability to merge an arbitrary number of output streams from CUT. The proposed method uses compact test sets and achieves reasonably high fault coverage with low overhead.

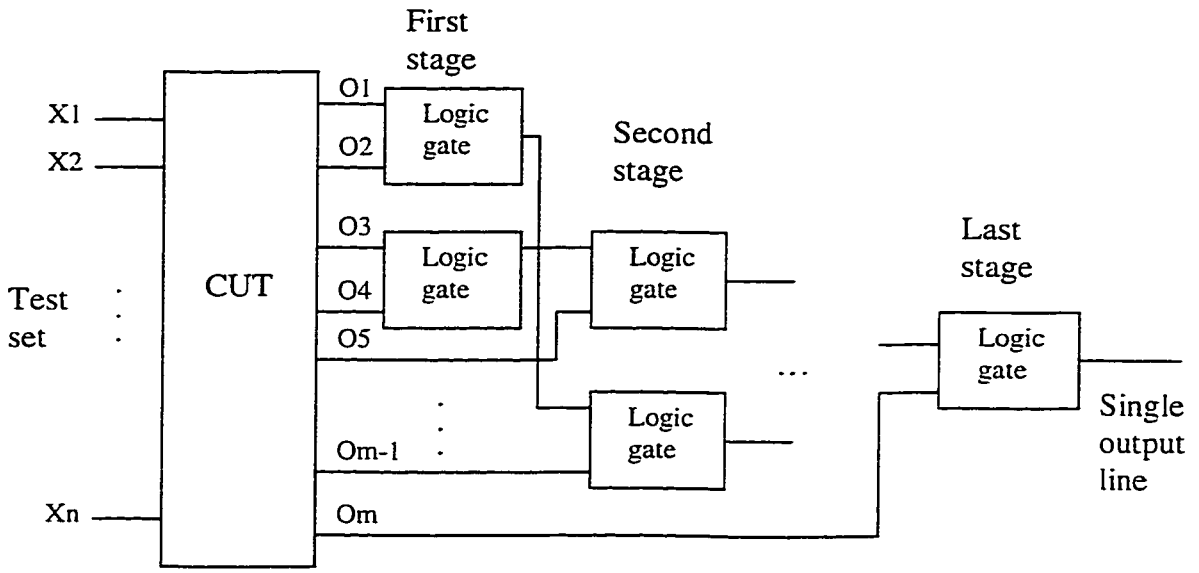


Figure 2.5 Modified hybrid space compaction.

2.1.6 Quadratic function compaction (QFC)

The quadratic function compaction is yet another space compaction approach [46, 49] that employs the concept of quadratic function and in which the k outputs $z_1, z_2, \ldots, z_k$ of the circuit under test are compressed and processed using a function of the type $z_{i0}z_{i1} \oplus z_{i2}z_{i3} \oplus \ldots \oplus z_{i(2k-2)}z_{i(2k-1)}$ where $z_{it}z_{i(t+1)}$ are blocks of length k, for t=0, 2, ..., 2k-2. The compressor consisting of a finite-field multiplier and XOR gates is shown in Figure 2.6.

QFC [35] requires more hardware overhead for data compression than any linear compaction scheme and also does not guarantee zero aliasing or no fault loss.

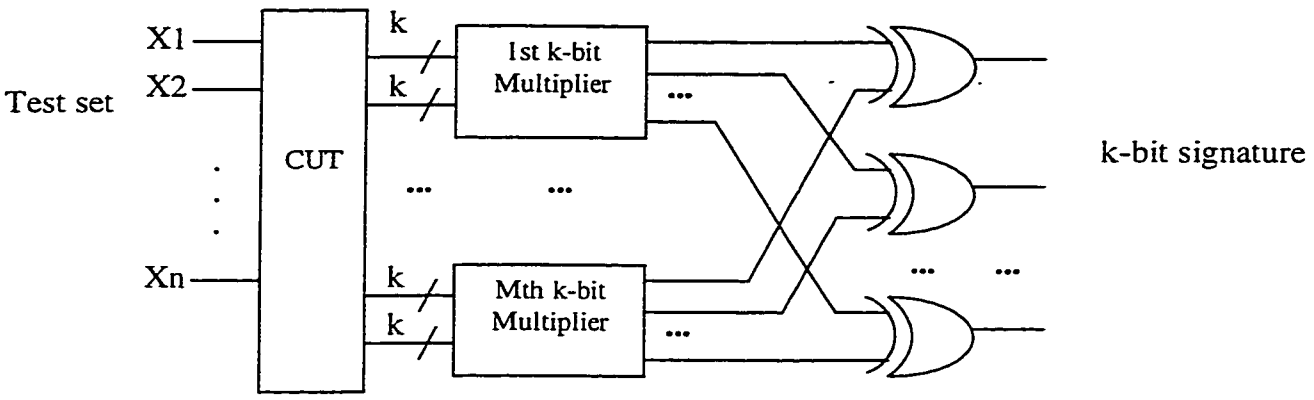


Figure 2.6 Space compression by quadratic compression.

2.1.7 Programmable space compaction

Programmable space compaction (PSC) is a technique for designing low-cost space compactors with high fault coverage [61]. In PSC, a circuit specific space compactor is designed to increase the likelihood of error propagation. However, PSC as well does not guarantee zero aliasing. An optimal compaction circuit with minimized aliasing and lower cost can be found by exhaustively enumerating all the 2^{2^k} k-input Boolean functions for a k-output circuit under test. PSC can be practically implemented using search techniques based on genetic algorithms.

2.1.8 Multiplexed parity tree

A new approach termed multiplexed parity tree (MPT) based on parity tree circuits has been proposed in [10, 11] to achieve high fault coverage. MPTs (shown in Figure 2.7) perform zero aliasing space compaction of test responses by combining the error propagation properties of multiplexers and parity trees. Experimental results indicate that the associated hardware overhead is moderate and a very high fault coverage is obtained for faults in the CUT including the compaction circuit. A different approach to achieve zero aliasing was also proposed by the authors. This technique is based on

constructing multiple-output space compactors. It is shown that a two –output circuit implementing two parity functions is adequate for aliasing-free compaction.

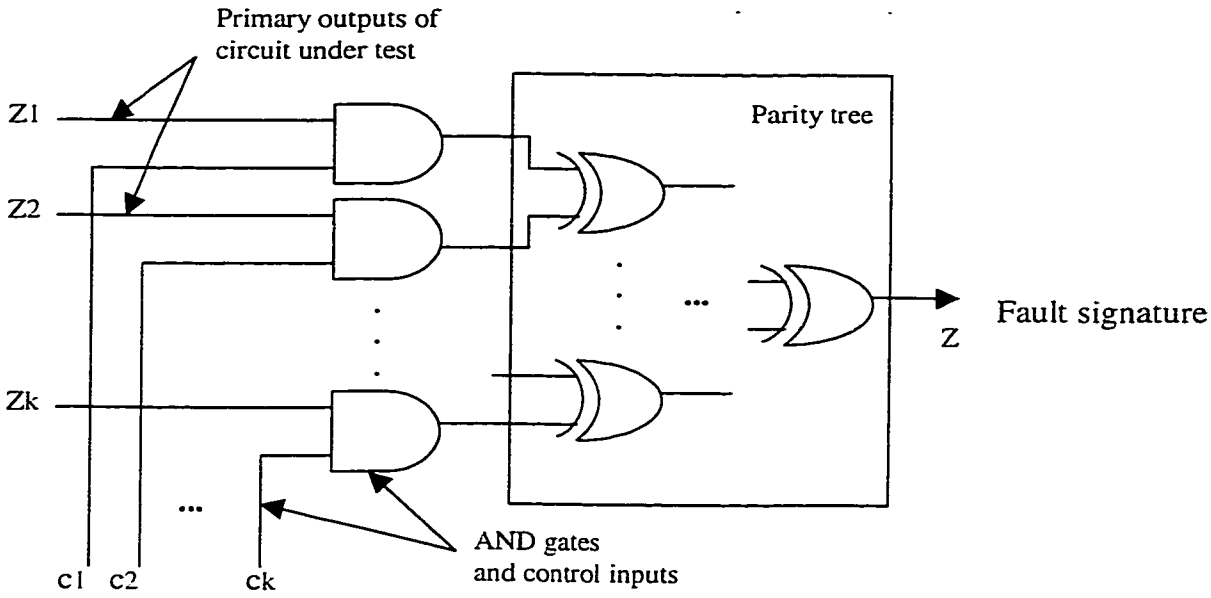


Figure 2.7 The MPT compaction scheme.

2.2 Time Compaction

The space compressors reduce the width of the test responses, while a time compressor is used to derive the signature in order to minimize the amount of storage needed to store information of the fault-free signatures.

In what follows is a brief discussion on some of the time compaction methods, such as ones-counting, syndrome testing, transition counting, signature analysis and so on.

2.2.1 Ones –count compression

In ones counting [30], the response signature is the number of 1s appearing in the test response. The compressor is a simple counter (Figure 2.8 (a)) and independent of the CUT. The length of the signature is a logarithmic factor of the length of the test response (the signature is of length $[\log_2 m]$ for an m-bit test response). The signature values do

not depend on the order of the applied test patterns. The statistical properties of ones-counting technique will not change if the output is inverted. The probability of aliasing increases rapidly, as the signature length is close to m/2 where m is the length of the output stream. When the signature of the fault-free response equals zero or m, no aliasing can occur. A fault is always detected if it creates an odd number of errors in the output response; if the number of errors is even, the fault may or may not be detected.

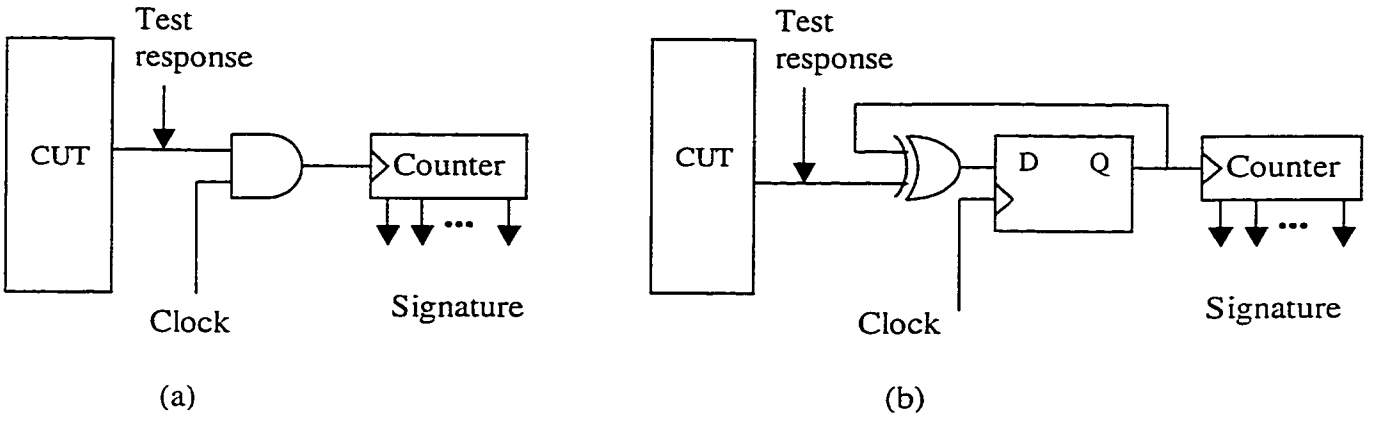


Figure 2.8 The one's counting and transition counting techniques for time compaction.

2.2.2 Transition-count compression

In transition-count testing [31], the signature is the number of 0-to-1 and 1-to-0 transitions in the output data stream. The hardware implementation of transition counting consists of a D flip-flops and an XOR logic gate that serve as a transition detector as well as a counter (Figure 2.8 (b)). For an m-bit test response, the signature length is less than $[\log_2 m]$. Like one's counting, the probability of masking [51] is low when the signature is far from the midpoint, i.e., m/2, but the probability is high at the midpoint. However, unlike one's counting, transition counting depends on the order of the test response. This technique also does not guarantee detecting all single-bit errors.

Subsequently, a new transition count method called double/multiple transition counting (DTC/MTC) has been proposed [21]. Single-output circuits can be tested using DTC testing, while MTC testing is used for multi-output circuits. DTC/MTC techniques

detect any faults that can be detected by conventional testing methods, and thus DTC/MTC technique does not result in any more information loss than the conventional testing. The size of a DTC/MTC test equals the size of the equivalent conventional test since no test vectors need to be repeated. The hardware implementations of DTC/MTC testing techniques are simple, as shown in Figure 9 (a) and (b), comprised of one flip-flop, one OR gate, one inverter, and one switch per output.

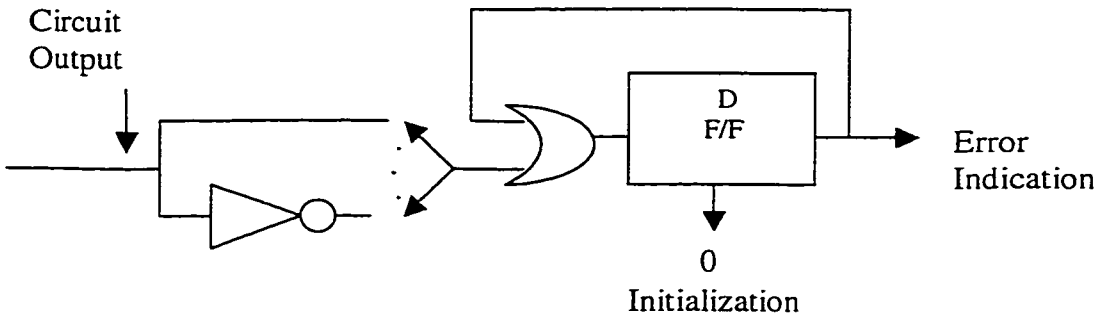


Figure 2.9 (a) A DTC tester for the single-output case.

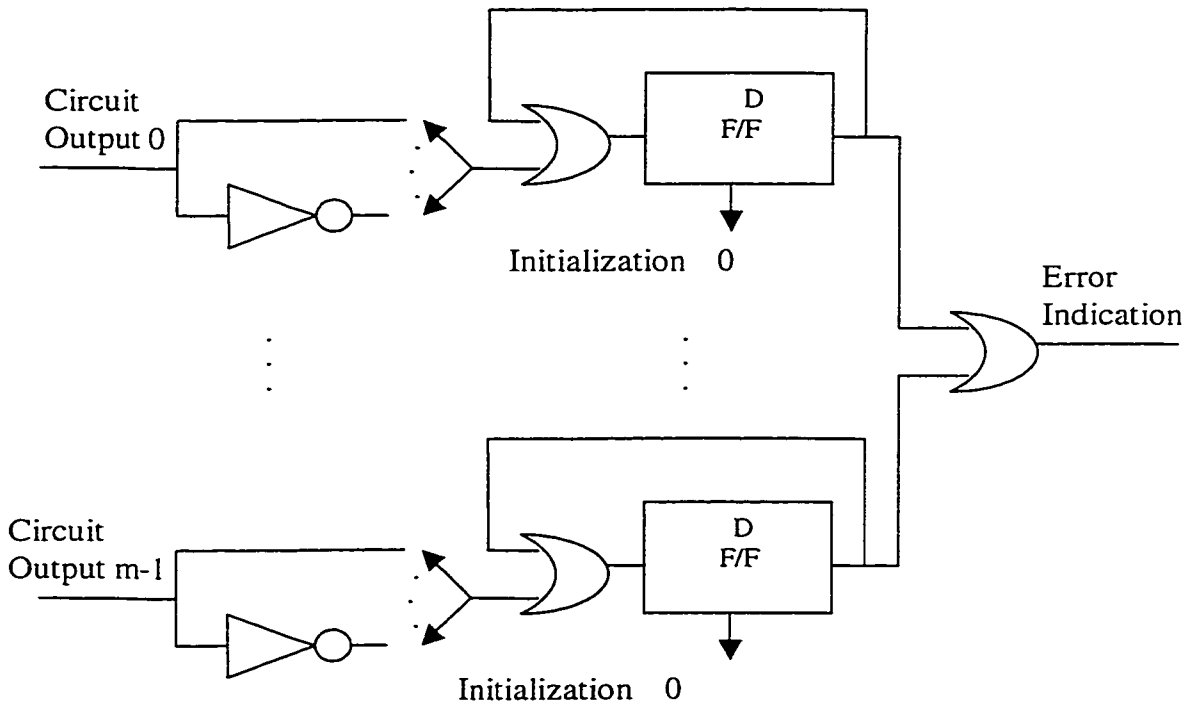


Figure 2.9 (b) An MTC tester for an m-output case.

2.2.3 Syndrome testing

One special case of one's counting is syndrome testing [50], which relies on exhaustive testing, i.e., all the 2^n test vectors are applied to an n-input combinational circuit. The syndrome S (or signature) is the normalized number of 1s in the test response, i.e., $S=K/2^n$, where K is the number of minterms of the function implemented by the single-output CUT. By adding extra inputs, circuits implementing logic functions can be transformed to circuits in which every single stuck-at fault is syndrome-testable. The signature in syndrome testing is test order independent as well.

2.2.4 Parity check compression

In parity check compression [7], the signature S is the parity of the circuit response. It detects all single-bit errors and all multiple-bit errors consisting of an odd number of error bits in the response sequence, while faults that create an even number of errors are not detected. Though this technique is not so effective, its simplicity is an advantage. Figure 2.10 shows a parity check compression circuit consisting of a one-bit shift-register latch (SRL) with feed back through an XOR circuit.

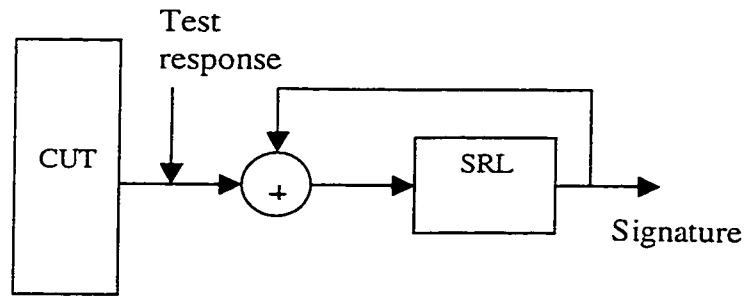


Figure 2.10 Parity check compression.

2.2.5 Signature analysis

Signature analysis [24] is the most common time compaction technique, which is realized in hardware using linear feedback shift registers (LFSRs), and is based on the concept of cyclic redundancy checking (CRC).

As described earlier, one of the applications for LFSR is to create pseudorandom test patterns; the other application is to carry out response compression, i.e., signature analysis.

The feature of the generated sequence patterns is determined by the LFSR's characteristic polynomial, which is defined by its interconnection structure. The theory for signature analysis is based on the concept of polynomial division, where the "remainder" left in the register after completion of the test process corresponds to the final signature. Usually a signature analyzer divides the data sequence polynomial Z(x) by the generator polynomial of the LFSR, P(x), producing a quotient Q(x) and remainder R(x), as:

$$Z(x) = Q(x)\,P(x) + R(x)$$

Polynomial division is performed serially by the LFSR as the response data sequence arrives. As the operation is performed, the quotient Q(x) is shifted out of the LFSR, leaving the remainder R(x) forming the signature.

In signature analysis, if the signature obtained from the CUT, i.e., $R^*(x)$, is different from the fault-free signature R(x), the circuit is recognized as faulty. Let Z(x) be the correct response and $Z^*(x)=Z(x) \oplus e(x)$ be the fault response, where polynomial e(x) represents an error sequence. In some cases, some information is lost during compression of the test responses. It is possible for certain faults to go undetected, which means that the signatures of some faulty circuits may be the same as the signatures of the fault-free circuit resulting in fault masking.

The aliasing problem being significantly important in signature analysis, various methods have been proposed to compute and reduce the aliasing probability, defined as the probability that a faulty circuit's signature maps onto the fault-free signature. Williams *et al.* model signature analysis by using Markov chains and derive an upper bound on the aliasing probability in terms of the test length and the probability of an error occurring at the output of the CUT [59]. In [44], another approach to computation of the aliasing probability is presented. Unlike other methods, the fault coverage in signature analysis may be improved without changing the test. It can be achieved by increasing the length of the LFSR or by being repeated using a different feedback polynomial. It is

demonstrated in [53] that for short test length, signature analysis detects all single-bit errors; however, there is no known theory that characterizes fault detection in signature analysis. Figure 2.11 shows an LFSR with a single input called a serial input signature register (SISR), with generating function $P(x)=x^4+x+1$.

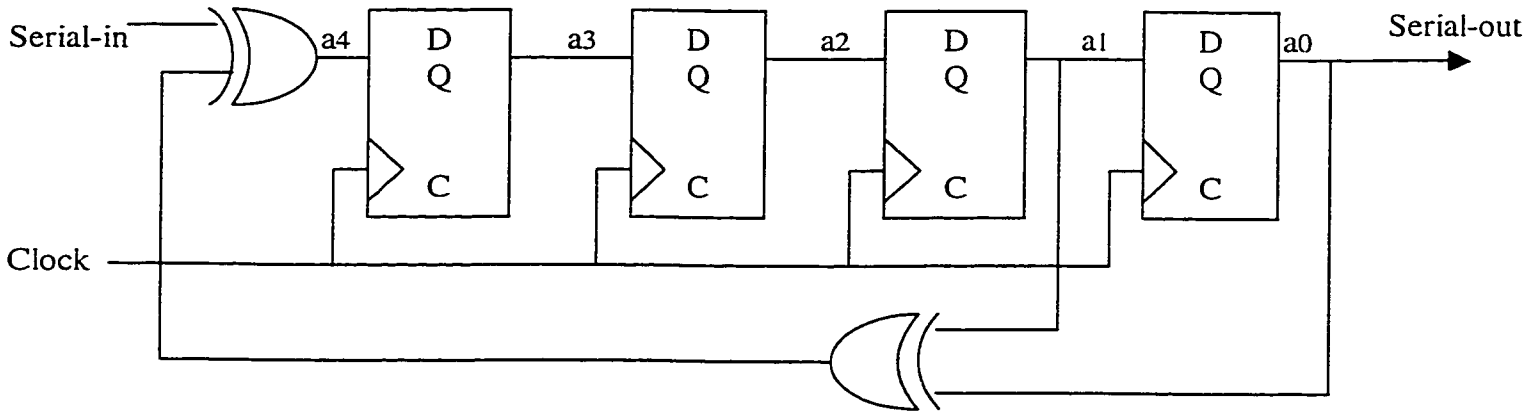


Figure 2.11 SISR logic diagram.

When a logic circuit has m parallel outputs, $z_1 \ldots z_m$, an efficient method is to use a multiple-input signature register (MISR), which effectively computes the signatures of the parallel input sequences concurrently with a single LFSR circuit and eliminate the need for a space compactor. On the other hand, MISR increases aliasing and requires extra hardware. A 4-bit MISR is shown in Figure 2.12, of which the generator function is also $P(x)=x^4+x+1$.

Zorian and Agarwal [61] proposed a method based on modification of the test response to reduce fault masking in signature analysis but the high hardware overhead and increased testing time make it unsuitable.

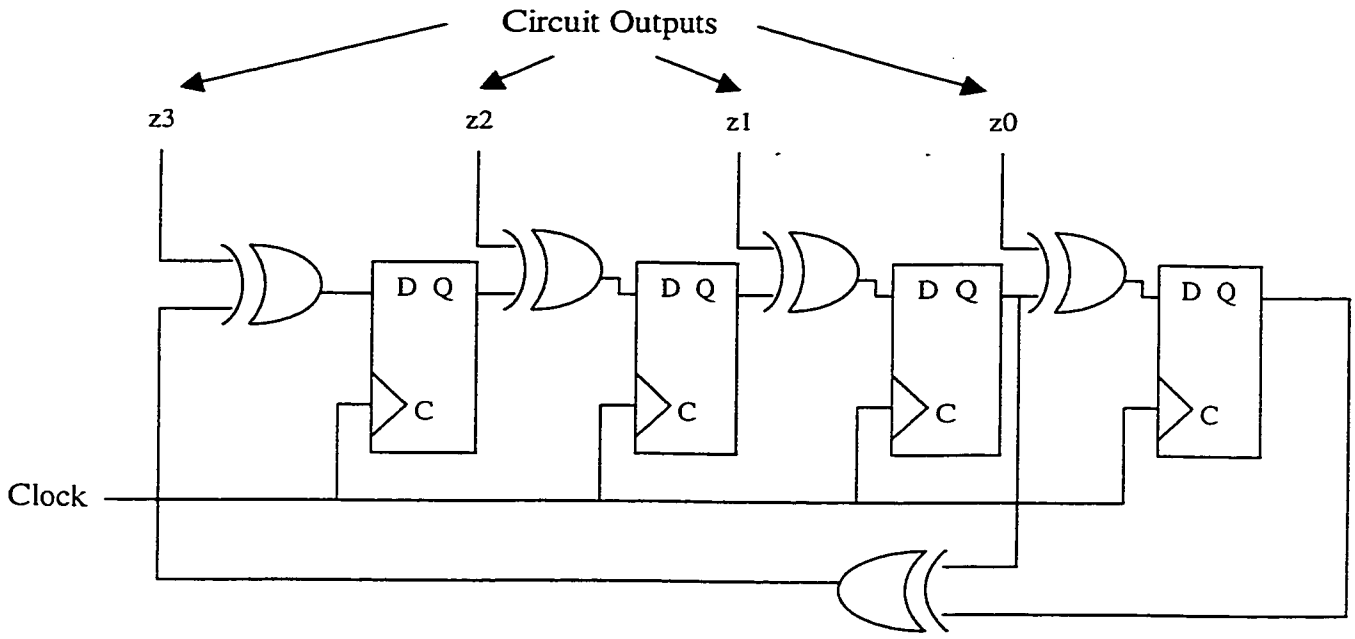


Figure 2.12 Multiple-input signature register (MISR).

2.2.6 Parallel compaction analysis

Using two different compaction techniques in parallel for testing was investigated in [52]. The combination of transition counting and signature analysis results in small overlap in their error masking, though the use of two different compaction techniques in parallel can improve fault coverage. However, the hardware overhead and the size of the fault signature are increased and as such it is seldom used in practice.

2.2.7 Walsh spectral analysis

In Walsh spectral analysis [32] use of the spectral coefficients representing the switching functions is done. The spectral coefficients are compared to well-known correct coefficient values. In other words, the truth table of the switching function is verified. The process of collecting and comparing a subset of the complete Walsh functions is described as a mechanism for data compaction. Use of the spectral coefficients improves the fault coverage, but it requires high area overhead [56] for

generating them. Figure 2.13 shows a hardware implementation of Walsh spectral testing scheme.

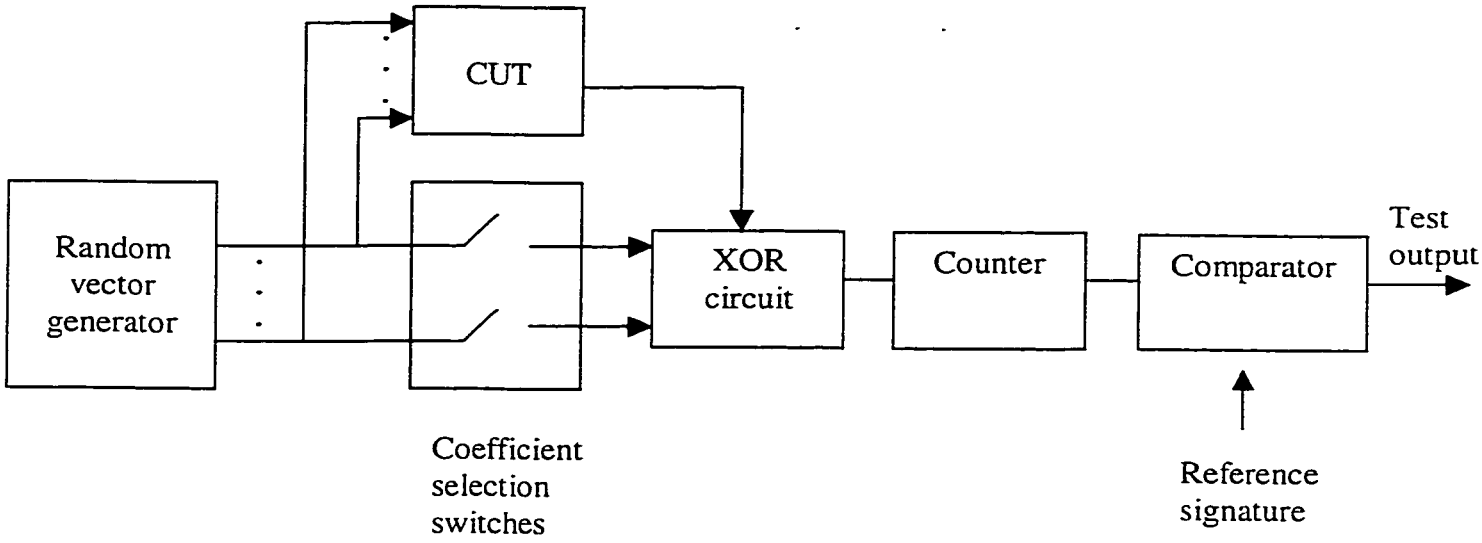


Figure 2.13 Implementing the RW compact testing scheme.

Chapter 3

Space Compaction Based on Sequence Characterization and Stochastic Independence of Line Errors

Introduction

The principle idea underlying space compression is to compress m functional outputs of the CUT possibly into one single output without substantial information loss. Mostly space compression has been accomplished using XOR gates in cascade or in a tree structure. In this thesis, we will adopt a combination of both cascade and tree structures (cascade-tree) for the design space compactors with AND/NAND, OR/NOR and XOR/XNOR logic gates. The basic approach to select a suitable gate to merge multi-candidate output lines of the CUT here will be based on certain mergeability criteria developed using switching theory formulation.

In particular, the concepts of Hamming distance, cover table [13], sequence weights, derived sequences and frequency ordering of literals [18] will be utilized to develop a general framework of sequence merger as will be evident from the following sections.

3.1. First-Order 1-Weight

The number of 1s in a sequence ψ_1 of length L_s is called its first-order 1-weight. The first-order 1-weight is denoted by W_{11}.

Example 3.1 Let ψ_1 = 1100 0101 0010.

Here, L_s=12 and the first-order 1-weight is

W_{11}=5.

3.2 First-Order 0-Weight

The number of 0s in a sequence ψ_1 of length L_s is called its first-order 0-weight. The first-order 0-weight is denoted by W_{10}.

Example 3.2 In ψ_1 above, the first- order 0-weight is

$W_{10}=7$.

In the case of first-order sequence weights, W_{1k}, the first subscript refers to the order of the sequence while the second subscript k is 1 or 0 depending on if the weight is a 1-weight or a 0-weight.

Theorem 3.1 For a sequence ψ_1 of length L_s with first-order 1-weight W_{11} and first order 0-weight W_{10}, respectively,

$$L_s=W_{11}+W_{10}$$

Proof: Follows obviously.

The concept can be extended to N sequences as follows.

3.3 Nth-Order 1-Weight

For N sequences $\psi_1, \psi_2, ..., \psi_N$, each of length L_s, let the sequences have 1s in m identical bit positions (the sequences each might have 1s in other positions as well besides these m positions). Then the Nth-order 1-weight of the sequences in the group is given by the number of bit positions in which all the N sequences have 1s which is m in this case.

Example 3.3 Consider 3 sequences of length $L_s=12$ as below.

$\psi_1=$ 1111 0010 1010

$\psi_2=$ 1111 0111 0110

$\psi_3=$ 1111 0101 1010

Here the third-order 1-weight of the sequences ψ_1, ψ_2, ψ_3 is

$$W_{31}=5.$$

i.e., the sequences agree in bit positions 2, 9, 10, 11, 12 having 1s (counting from the right).

3.4 Nth-Order 0-Weight

For N sequences ψ_1, ψ_2,....,ψ_N, the Nth-order 0-weight is defined in the same way as the Nth-order 1-weight and corresponds to the number of bit positions in which all the N sequences have 0s.

Here an Nth-order 1-weight and 0-weight are denoted, respectively, by W_{N1} and W_{N0}, where the first subscript as before corresponds to the sequence order while the second subscript gives the binary digit corresponding to the weight.

Example 3.4 Considering the example above, for a group of 3 sequences, the third-order 0-weight is

$$W_{30}=2.$$

i.e., all the three sequences have 0s in the first and eighth bit positions from the right.

3.5 Bundling of Sequences

If N sequences having certain Nth-order 1-weight (or 0-weight as defined above) are grouped together, then the process will be termed *bundling* and the grouped sequences will be termed bundled sequences.

Whenever there will be bundling of a number of sequences, we will say, we are working under bundling constraint.

3.6 Second-Order Derived Sequences

For two sequences of length L_s each with a Hamming distance H_s (the number of bit positions in which they differ), the second-order derived sequences are obtained by deleting those bit positions in which they do not agree and putting a dash (-) in those positions.

Example 3.5 Consider the sequences ψ_1 and ψ_2 of the previous example. The derived sequences are denoted by ψ_1' and ψ_2' respectively, and given as:

ψ_1' = 1111 0-1- --10

ψ_2' = 1111 0-1- --10

The second-order 1-weight and 0-weight of the derived sequences ψ_1' and ψ_2' are denoted respectively by W_{21}' and W_{20}', where W_{21}'=6 and W_{20}'=2.

Theorem 3.2 For a pair of sequences ψ_1 and ψ_2 of length L_s each with Hamming distance H_s, let the derived sequences be ψ_1' and ψ_2', and let the second-order 1-weight and 0-weight be W_{21}' and W_{20}', respectively. Then

$$L_s = H_s + W_{21}' + W_{20}'.$$

Proof: Follows obviously.

It may be noted that the second-order 1-weight and 0-weight for the original sequence pair ψ_1 and ψ_2, W_{21} and W_{20}, respectively, are the same as the weights W_{21}' and W_{20}'.

3.7 Nth-Order Derived Sequences

For N sequences ψ_1, ψ_2, ...,ψ_N of length L_s each, the Nth-order derived sequences ψ_1', ψ_2' ,....,ψ_N' are obtained exactly as in the case of two sequences, by deleting the bit positions in which at least two of the sequences differ and replacing the bit positions with a dash(-).

Example 3.6 For the 3 sequences ψ_1, ψ_2, ψ_3 of length L_s=12 each, as shown above, we have the derived sequences

$\dot{\psi_1}$= 1111 0--- --10

$\dot{\psi_2}$= 1111 0--- --10

$\dot{\psi_3}$= 1111 0--- --10

Theorem 3.3 If the Nth-order 1-weight is $\dot{W}_{N1}$ and the Nth-order 0-weight is $\dot{W}_{N0}$ of the N derived sequences in a bundle, then

$$L_s = \dot{W}_{N1} + \dot{W}_{N0} + R$$

where R called the *residue* is the number of bit positions in which at least two of the sequences of ψ_1, ψ_2,...,ψ_N in the bundle have different entries and the corresponding set of derived sequences have dash (-) entries.

Proof: Follows obviously.

Theorem 3.4 Two Nth-order derived sequences may have the same Nth-order 1-weight, 0-weight and residue R, but they may represent two entirely different sets of bundled sequences.

Proof: The proof is given by the following example. Consider two bundled sets of 3 sequences as:

ψ_1 = 1101 0101

ψ_2 = 1110 0000

ψ_3 = 1111 0100

and

ϕ_1 = 0110 1001

ϕ_2 = 0111 0010

ϕ_3 = 0110 1011

Their derived sequences are:

$\psi_1^{\cdot}$ = 11-- 0-0-

$\psi_2^{\cdot}$ = 11-- 0-0-

$\psi_3^{\cdot}$ = 11-- 0-0-

and

$\phi_1^{\cdot}$ = 011- -0--

$\phi_2^{\cdot}$ = 011- -0--

$\phi_3^{\cdot}$ = 011- -0--

Both the derived sets have the same $W_{31}^{\cdot}$, $W_{30}^{\cdot}$ and the residue R, but they come from two different sets of sequences and also represent entirely distinct sets of derived sequences.

3.8 Error Multiplicity

The error multiplicity denoted by ν is the number of output lines which can be faulty simultaneously.

Theorem 3.5 On the assumption of stochastic independence of line errors for a bundled set of N output sequences ψ_1, ψ_2, ..., ψ_N, each of length L_s at the output of a CUT, the maximum possible errors with all possible multiplicities of errors ν_i is

$$E_{max} = L_s[2^N - 1].$$

Proof: For N sequences of length L_s each, every bit position can have either only one error (error multiplicity ν_1), can have two simultaneous errors (error multiplicity ν_2) and so on up to N possible simultaneous errors with error multiplicity ν_N. The sum total of all possible errors in one bit position is then $2^N - 1$, where 2^N is the total number of binary combinations in the bit position and 1 corresponds to the one error-free pattern. Thus, for N sequences of length L_s, the total number of all errors is

$$E_{max} = L_s[2^N - 1].$$

Theorem 3.6 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s each, let the derived sequences ψ_1', ψ_2', ..., ψ_N', respectively, have Nth-order 1-weight and 0-weight, W'_{N1} and W'_{N0}, respectively, and let R be the residue. The total number of faults detected when the N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input AND/NAND gate is

$$T(\text{AND/NAND}) = (W'_{N1})_{\nu_1,\nu_2,\ldots,\nu_N} \times (2^N - 1) + (W'_{N0})_{\nu_N} + \sum_{\nu_j} R_{i_{\nu_j}}$$

where $(W'_{N1})_{\nu_1,\nu_2,\ldots,\nu_N}$ represents error multiplicities of 1, 2, ..., N and $R_{i_{\nu_j}}$ represents the ith residue position of ν_j 0s.

Proof: Since the derived sequences have 1-weight W'_{N1}, having all 1s in these positions, errors of multiplicities 1, 2, ..., N (ν_1, ν_2, ..., ν_N) are detected by an N-input AND/NAND gate in each position. The sum of all these errors is W'_{N1} $(2^N - 1)$. Similarly, since the 0-weight is W'_{N0}, only one fault of multiplicity N(ν_N) is detected in each bit position and the total number of errors detected is W'_{N0}. In each of the remaining R residual positions, depending on the number of 0s in a bit position, only one error of appropriate multiplicity is detected (if the sequence has r 0s, then only error of multiplicity ν = r making the position 1 is detected). The totality of all these errors is R. Hence the AND/NAND gate detects

$$T(\text{AND/NAND}) = W'_{N1}(2^N - 1) + W'_{N0} + R$$

where $R = L_s - (W'_{N1} + W'_{N0})$.

Theorem 3.7 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s each, let the derived sequences ψ_1', ψ_2', ..., ψ_N', respectively, have Nth-order 1-weight and 0-weight, W'_{N1} and W'_{N0}, respectively, and let R be the residue. The total number of

faults detected when the N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input OR/NOR gate is

$$T(\text{OR/NOR}) = (\dot{W}_{N0})_{v_1.v_2.....v_N} \times (2^N\text{-}1) + (\dot{W}_{N1})_{v_N} + \sum_{v_j} R_{i_{v_j}}$$

where $(\dot{W}_{N0})_{v_1.v_2.....v_N}$ represents the number of error multiplicities of 1, 2, ..., N and $R_{i_{v_j}}$ represents the number of errors corresponding to the ith residue position of v_j 0s.

Proof: The OR/NOR gate detects obviously $\dot{W}_{N0}(2^N\text{-}1)$ errors corresponding to $\dot{W}_{N0}$ bit positions having all 0s, and $\dot{W}_{N1}$ errors corresponding $\dot{W}_{N1}$ bit positions having all 1s, and $R = L_s - \dot{W}_{N1} - \dot{W}_{N0}$ errors in all the R residual positions. Hence the totality of the errors detected is

$$T(\text{OR/NOR}) = \dot{W}_{N0}(2^N\text{-}1) + \dot{W}_{N1} + R.$$

Theorem 3.8 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s each, let the derived sequences ψ_1', ψ_2', ..., ψ_N', respectively, have Nth-order 1-weight and 0-weight, $\dot{W}_{N1}$ and $\dot{W}_{N0}$, respectively, and let R be the residue. The total number of faults detected when the N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input XOR/XNOR gate is

$$T(\text{XOR/XNOR}) = (\dot{W}_{N1})_{v_1.v_3.....v_{N-1}}(2^N/2) + (\dot{W}_{N0})_{v_2.v_4.....v_N}(2^N/2) + \sum_{v_j} R_{i_{v_j}}$$

$$= L_s(2^N/2) = L_s(2^{N-1})$$

where N is even.

Proof: If the number of sequences N is even, then corresponding to each of the positions of $\dot{W}_{N1}$ having all 1s, all errors of odd multiplicities are detected at the output of the CUT. Similarly, all errors of even multiplicities are detected at the output of the CUT corresponding to each of the positions of $\dot{W}_{N0}$ having all 0s. For the remaining R residual bit positions, depending on the number of 0s, either errors of all odd or even

multiplicities are detected. In every bit position, the total number of all errors of either odd or even multiplicity is $2^N/2 = 2^{N-1}$. Hence, an XOR/XNOR gate detects a total of

$$T(\text{XOR/XNOR}) = L_s(2^{N-1}) \text{ faults.}$$

Theorem 3.9 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s each, let the derived sequences be ψ_1', ψ_2', ..., ψ_N', respectively. Let the Nth-order 1-weight be W'_{N1} and the Nth-order 0-weight be W'_{N0}. For merger of the N sequences, an N-input AND/NAND gate will be preferable to an N-input OR/NOR gate for maximizing error detection if

$$W'_{N1} > W'_{N0}.$$

Proof: The total errors detected by the N-input AND/NAND gate and N-input OR/NOR gate are, respectively,

$$T(\text{AND/NAND}) = W'_{N1}(2^N - 1) + W'_{N0} + R$$

$$T(\text{OR/NOR}) = W'_{N0}(2^N - 1) + W'_{N1} + R$$

where R is the residue, that is, bit positions where at least two of the sequences ψ_1, ψ_2, ..., ψ_N have differing entries.

AND/NAND is preferable to OR/NOR if

$$W'_{N1}(2^N - 1) + W'_{N0} + R > W'_{N0}(2^N - 1) + W'_{N1} + R$$

or

$$W'_{N1}(2^N - 2) + W'_{N1} + W'_{N0} + R > W'_{N0}(2^N - 2) + W'_{N1} + W'_{N0} + R$$

or

$$W'_{N1} > W'_{N0}.$$

Theorem 3.10 For merger of N sequences ψ_1, ψ_2, ..., ψ_N of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input OR/NOR gate is preferable to an N-input AND/NAND gate if

$$W'_{N0} > W'_{N1}.$$

Theorem 3.11 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s, let the derived sequences be ψ'_1, ψ'_2, ..., ψ'_N, respectively. Let the Nth-order 1-weight be W'_{N1} and the Nth-order 0-weight be W'_{N0}. For merger of the N sequences, an N-input AND/NAND gate will be preferable to an N-input XOR/XNOR gate for maximizing error detection if

$$W'_{N1} > W'_{N0} + R = L_s/2.$$

Proof: The total errors detected by the N-input AND/NAND gate and N-input XOR/XNOR gate are, respectively,

$$T(\text{AND/NAND}) = W'_{N1}(2^N - 1) + W'_{N0} + R$$

$$T(\text{XOR/XNOR}) = L_s(2^{N-1})$$

where R is the residue, that is, bit positions where at least two of the sequences ψ_1, ψ_2, ..., ψ_N have differing entries.

An AND/NAND gate is preferable to an XOR/XNOR gate if

$$W'_{N1}(2^N - 1) + W'_{N0} + R > L_s(2^{N-1})$$

or

$$W'_{N1}(2^N - 2) + W'_{N1} + W'_{N0} + R > (W'_{N1} + W'_{N0} + R)(2^{N-1})$$

or

$$2W'_{N1}(2^{N-1} - 1) > (W'_{N1} + W'_{N0} + R)(2^{N-1} - 1)$$

or

$$2W'_{N1} > W'_{N1} + W'_{N0} + R = L_s$$

or

$$W'_{N1} > W'_{N0} + R$$

or

$$W'_{N1} > L_s/2.$$

Theorem 3.12 For merger of N sequences ψ_1, ψ_2, ..., ψ_N of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input XOR/XNOR gate is preferable to an N-input AND/NAND gate if

$$W'_{N1} < W'_{N0} + R$$

or

$$W'_{N1} < L_s/2.$$

Theorem 3.13 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s, let the derived sequences be ψ'_1, ψ'_2, ..., ψ'_N, respectively. Let the Nth-order 1-weight be W'_{N1} and the Nth-order 0-weight be W'_{N0}. For merger of the N sequences, an N-input OR/NOR gate will be preferable to an N-input XOR/XNOR gate for maximizing the error detection if

$$W'_{N0} > W'_{N1} + R$$

or

$$W'_{N0} > L_s/2.$$

Proof: The total errors detected by the N-input OR/NOR gate and N-input XOR/XNOR gate are, respectively,

$$T(OR/NOR) = W'_{N0}(2^N - 1) + W'_{N1} + R$$

$$T(XOR/XNOR) = L_s(2^{N-1})$$

where R is the residual bit positions where at least two of the sequences ψ_1, ψ_2, ..., ψ_N have differing entries.

An OR/NOR gate is preferable to an XOR/XNOR gate if

$$W'_{N0}(2^N - 1) + W'_{N1} + R > L_s(2^{N-1})$$

or

$$W'_{N0}(2^N - 2) + W'_{N1} + W'_{N0} + R > (W'_{N1} + W'_{N0} + R)(2^{N-1})$$

or

$$2W'_{N0}(2^{N-1}-1) > (W'_{N1} + W'_{N0} + R)(2^{N-1}-1)$$

or

$$2W'_{N0} > W'_{N1} + W'_{N0} + R$$

or

$$W'_{N0} > W'_{N1} + R$$

or

$$W'_{N0} > L_s/2.$$

Theorem 3.14 For merger of N sequences ψ_1, ψ_2, ..., ψ_N of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input XOR/XNOR gate is preferable to an N-input OR/NOR gate if

$$W'_{N0} < W'_{N1} + R$$

or

$$W'_{N0} < L_s/2.$$

Theorem 3.15 For N sequences ψ_1, ψ_2, ..., ψ_N at the output of a CUT of length L_s, let as usual the derived sequences be ψ'_1, ψ'_2, ..., ψ'_N, respectively. Let the Nth order 1-weight be W'_{N1} and the Nth order 0-weight be W'_{N0}. In the extreme case when

$$R = L_s \text{ (and } W'_{N1} = W'_{N0} = 0)$$

the total faults detected by the N-input AND/NAND, OR/NOR and XOR/XNOR gates are respectively,

$$T(\text{AND/NAND}) = T(\text{OR/NOR}) = L_s$$

and

$$T(\text{XOR/XNOR}) = L_s(2^{N-1}).$$

Proof: The proof follows obviously.

3.9 Implementation

In this section, we will develop our first algorithm. This algorithm uses the generalized mergeability criteria and is implemented using the C language to obtain the space compactors for all the ISCAS 85 combinational benchmark circuits. An example will be explained in detail for clarification later.

3.9.1 Definition

(1) 1-cover table (0-cover table)

A 1-cover table (or 0-cover table) is defined as follows: Write all the bit positions of a sequence in a row, and below each of the bit positions (1 for 1-cover table, 0 for 0-cover table) make an entry of the sequence including it in a columnar form. Each of the columns headed by a specific bit position will then give a collection of the sequences having bit value 1 (or 0) in that particular bit position. Such a table will be called a 1-cover table (or 0-cover table) [13].

Example 3.7

Consider the following 8 sequences:

x_1= 1101 1100

x_2= 1101 0110

x_3= 1111 1101

x_4= 0111 1001

x_5= 1111 1111

x_6= 0000 0001

x_7= 1011 1111

x_8= 1111 1101

According to the above definition, we can obtain the 1-cover table and 0-cover table as follows (P1, P2, ..., P8 denote the bit-positions):

P1	P2	P3	P4	P5	P6	P7	P8
x_1	x_1	x_3	x_1	x_1	x_1	x_2	x_3
x_2	x_2	x_4	x_2	x_3	x_2	x_5	x_4
x_3	x_3	x_5	x_3	x_4	x_3	x_7	x_5
x_5	x_4	x_7	x_4	x_5	x_5		x_6
x_7	x_5	x_8	x_5	x_7	x_7		x_7
x_8	x_8		x_7	x_8	x_8		x_8
			x_8				

Table 3.1 The 1-cover table for Example 3.7.

P1	P2	P3	P4	P5	P6	P7	P8
x_4	x_6	x_1	x_6	x_2	x_4	x_1	x_1
x_6	x_7	x_2		x_6	x_6	x_3	x_2
		x_6				x_4	
						x_6	
						x_8	

Table 3.2 The 0-cover table for Example 3.7.

(2) Frequency ordering of the literals

For any two distinct literals x_i and x_j appearing in a cover table, if the literal x_i occurs in more columns in the cover table than the literal x_j, an ordering of the literals can be made as $x_i \geq x_j$. If both the literals x_i and x_j appear in the same number of columns in the cover table, the ordering can be made either $x_i \geq x_j$ or $x_j \geq x_i$. This kind of ordering ($\geq$) that can be established amongst different literals in a cover table is called the frequency ordering of the literals [18].

Example 3.8 Consider the 1-cover table in Example 3.7. We find the one frequency-ordering of the literals x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 as follows:

$$x_5 > x_3 = x_7 = x_8 > x_1 = x_2 = x_4 > x_6.$$

Consider next the 0-cover table in Example 3.7. We find the zero frequency-ordering of the literals x_1, x_2, x_3, x_4, x_6, x_7, x_8 as given below:

$$x_6 > x_1 = x_2 = x_4 > x_3 = x_7 = x_8.$$

(3) Other related definitions

We repeat the definitions for the following terms, though they have been described in the previous section.

W_{N1}: Number of matching 1s in the same bit positions of all candidate sequences.

W_{N0}: Number of matching 0s in the same bit positions of all candidate sequences.

R: The residue (Hamming distance of multiple sequences).

L_s: The length of the sequence.

3.9.2 Mergeability criteria

The main idea for merging a bundled set of sequences by a logic gate (AND/NAND or OR/NOR or XOR/NXOR) is to find if the sequences satisfy one of the following conditions.

- If $W_{N1} \geq L_s/2$, the obtained sequences are merged together with an AND/NAND gate only.
- If $W_{N0} > L_s/2$, the obtained sequences are merged together with an OR/NOR gate only.
- If $W_{N1} < L_s/2$ and $W_{N0} < L_s/2$, the sequences which cannot be merged by either AND/NAND or OR/NOR gates are merged together with an XOR/XNOR gate only at this level.

3.9.3 Algorithm 1

3.9.3.1 To find the candidate sequences in the original sequences for merger by using AND/NAND gates:

1) From the input sequence list (excluding the discarded sequences), set up the 1-cover table.
2) Find the corresponding frequency ordering in the 1-cover table.
3) Try to find the one_maxgroup sequences in the frequency ordering,
   i.e., when $W_{N1} \geq L_s/2$, the number of candidate sequences(S1) be as many as possible. If we can find this kind of one_maxgroup sequences in the frequency ordering, we go to step 4); otherwise, go to step 3.9.3.2.
   (For finding one_maxgroup of sequences, go to the sub-algorithm).
4) Based on the mergeability criteria, merge the candidate sequences in the one_maxgroup from step 3) by AND/NAND gate (Henceforth, we use only AND gate).
5) Store the resulting output sequence from AND operation and discard the merged candidate sequences.
6) Repeat steps 1), 2), 3), 4) and 5). Find all other one_maxgroup sequences that can be merged by AND/NAND gate at the same level until none can be found. Go to step 3.9.3.2.

3.9.3.2 To find the candidate sequences in the original sequences for merger by using OR/NOR gates:

1) From the all original sequences (excluding the discarded sequences), set up the 0-cover table.
2) Find the corresponding frequency ordering in the 0-cover table.
3) Try to find the zero_maxgroup sequences in the frequency ordering, i.e., when $W_{N0} > L_s/2$, the number of candidate sequences(S0) be as many as possible. If we can find this kind of zero_maxgroup sequences in the frequency ordering, we go to step 4), otherwise, go to step 3.9.3.3.
   (For finding zero_maxgroup sequences, go to the sub-algorithm).
4) Based on the mergeability criteria, merge the candidate sequences in the zero_maxgroup from step 3) by OR/NOR gate (Henceforth, we use only OR gate).

5) Store the output sequence of OR operation and discard the candidate sequences.
6) Repeat steps 1), 2), 3), 4) and 5). Find all other zero_maxgroup sequences that can be merged by gate OR/NOR at the same level until none can be found. Go to step 3.9.3.3.

3.9.3.3 Merger of the remaining sequences using XOR/XNOR gates:

In this step, the remaining sequences, which cannot be merged by either AND/NAND or OR/NOR gates at the same level will be merged by XOR/XNOR gate (Henceforth, we use only XOR gate). Store the output sequence of XOR operation, and discard the remaining sequences.

3.9.3.4 Set up the new original input sequence list composed of the output sequences obtained from the previous level. Repeat steps 3.9.3.1 ~ 3.9.3.3 until a single output sequence is obtained.

3.9.4 Explanation of the Algorithm 1 by an example

In this section, an example will be used to explain in detail the Algorithm 1.

Example 3.9 c432 Benchmark Circuit

Consider the following sequences from the output of the circuit c432 which form the inputs to our space compactor. Using Algorithm 1 together with ATALANTA/FSIM simulator, we gradually merge the sequences until a single output sequence results.

The output sequences for the circuit c432 are:

x_0 = 1111011101010110111110111001110111111101110111111
x_1 = 0101011111111010110001011110101110010101001011101
x_2 = 1001101110011010111111011110010011111100000100100
x_3 = 1111110101111111011110111111101111111111001111111111
x_4 = 0110100001010110011001100110010110011000010111101
x_5 = 1010111101011100111000110010001010110001010011100
x_6 = 1111110101001100000100000100010000010010111011000
The length of the sequences is: L_s=49.

LEVEL ONE:

A. Find the candidate sequences among the input sequences to be merged by AND/NAND gates.

A.1) Set up the 1-cover table for the original sequences. The 1-cover table is shown in Table 3.3.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_0	x_0	x_0	x_0	x_2	x_0	x_0	x_0	x_1	x_0	x_1	x_0	x_1	x_0	x_0	
x_2	x_1	x_3	x_1	x_3	x_1	x_1	x_1	x_2	x_1	x_3	x_1	x_2	x_3	x_1	
x_3	x_3	x_4	x_2	x_4	x_3	x_2	x_2		x_3		x_2	x_3	x_4	x_2	
x_5	x_4	x_5	x_3	x_5	x_5	x_5	x_3		x_4		x_3	x_5	x_5	x_3	
x_6	x_6	x_6	x_6	x_6	x_6		x_5		x_5		x_4	x_6	x_6	x_4	
							x_6		x_6		x_5				

Table 3.3 The 1-cover table of Example 3.9 (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_0	x_0	x_0	x_0	x_0	x_1	x_0	x_0	x_0	x_1	x_1	x_0	x_0	x_0	x_1	x_0
x_1	x_1	x_2	x_2	x_2	x_2	x_3	x_1	x_1	x_2	x_2		x_1	x_2	x_3	x_1
x_2	x_2	x_3	x_3		x_3	x_4	x_2	x_2	x_3	x_3		x_3	x_3	x_5	x_3
x_3	x_3	x_4	x_6		x_4	x_5	x_3	x_3	x_4	x_4			x_4		x_4
x_5	x_4	x_5					x_5		x_6	x_5			x_6		
	x_5														

Table 3.3 The 1-cover table of Example 3.9 (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_0	x_0	x_0	x_0	x_0	x_0	x_6	x_0	x_0	x_0	x_1	x_0	x_0	x_0	x_0	x_0	x_0
x_1	x_2	x_2	x_1	x_2	x_1		x_1	x_3	x_3	x_3	x_2	x_1	x_1	x_1	x_3	x_1
x_2	x_3	x_3	x_2	x_3	x_2		x_3	x_6	x_4	x_6	x_3	x_3	x_3	x_2		x_3
x_3		x_5	x_3	x_4			x_5		x_5		x_4	x_4	x_4	x_3		x_4
x_4			x_4						x_6			x_5	x_5	x_4		
x_5			x_5									x_6	x_6	x_5		
			x_6													

Table 3.3 The 1-cover table of Example 3.9.

A.2) From the 1-cover table (Table 3.3), we obtain the frequency ordering of the literals x_0, x_1, x_2, x_3, x_4, x_5, and x_6. The number of columns in which these literals x_0, x_1, x_2, x_3, x_4, x_5, x_6 appear in the 1-cover table are 38, 30, 28, 42, 24, 25, 20. respectively. So we get the frequency ordering of literals x_0, x_1, x_2, x_3, x_4, x_5, and x_6 as:

$$x_3 > x_0 > x_1 > x_2 > x_5 > x_4 > x_6 \qquad (1)$$

A.3) Finding the one_maxgroup in the frequency ordering (1):

a) Select the candidate group from the frequency ordering having 1's number greater than $L_s/2$. In this example, the 1's number of the sequences x_0, x_1, x_2, x_3, x_4, x_5, and x_6 are 38, 30, 28, 42, 24, 25, and 20, respectively, and so the sequences constituting the candidate group are x_3, x_0, x_1, x_2, and x_5.

b) From the candidate group composed of sequences x_3, x_0, x_1, x_2, and x_5, we try to get a one_maxgroup of sequences, i.e., for which $W_{N1} \geq L_s/2$.

c) According to the Sub-algorithm, sequence x_3 and sequence x_0 constitute one_maxgroup, because $W_{21}=34 > L_s/2$.

A.4) Merge sequence x_3 and sequence x_0 by using gate AND (NAND is omitted for discussion) based on the mergeability criteria. Next x_3 and x_0 are replaced by their output sequence y_0, which is:

y_0= 1111010101010110111100111000110111111100111010111111.

Discard sequences x_3 and x_0 from the original input list, and store their output sequence y_0.

A.5) Now we have the following remaining original input sequences:

x_1 = 0101011111111101011000101111010111001010100101111 01
x_2 = 1001101110011010111111011110010011111100000100100
x_4 = 0110100001010110011001100110010110011000010111101
x_5 = 1010111101011100111000110010001010110001010011100
x_6 = 1111110101001100000100000100010000010010111011000

A.6) Form the new 1-cover table (Table 3.4) for the remaining input sequences comprised of sequences x_1, x_2, x_4, x_5, and x_6.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_2	x_1	x_4	x_1	x_2	x_1	x_1	x_1	x_1	x_1	x_1	x_1	x_1	x_4	x_1	
x_5	x_4	x_5	x_2	x_4	x_5	x_2	x_2	x_2	x_4		x_2	x_2	x_5	x_2	
x_6	x_6	x_6	x_6	x_5	x_6	x_5	x_5		x_5		x_4	x_5	x_6	x_4	
				x_6			x_6		x_6		x_5	x_6			

Table 3.4 New 1-cover table (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_1	x_1	x_2	x_2	x_2	x_1	x_4	x_1	x_1	x_1	x_1		x_1	x_2	x_1	x_1
x_2	x_2	x_4	x_6		x_2	x_5	x_2	x_2	x_2	x_2			x_4	x_5	x_4
x_5	x_4	x_5			x_4		x_5		x_4	x_4			x_6		
	x_5								x_6	x_5					

Table 3.4 New 1-cover-table (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_1	x_2	x_2	x_1	x_2	x_1	x_6	x_1	x_6	x_4	x_1	x_2	x_1	x_1	x_1		x_1
x_2		x_5	x_2	x_4	x_2		x_5		x_5	x_6	x_4	x_4	x_4	x_2		x_4
x_4			x_4						x_6			x_5	x_5	x_4		
x_5			x_5									x_6	x_6	x_5		
			x_6													

Table 3.4 New 1-cover table.

From the new 1-cover table, the number of occurrences of the sequence x_1, x_2, x_4, x_5, x_6 are 30, 28, 24, 25, 20, respectively, and so the corresponding frequency ordering is

$$x_1 > x_2 > x_5 > x_4 > x_6 \qquad (2)$$

According to the Sub-algorithm, we cannot find a one_maxgroup of sequences for the frequency ordering (2). That means, in these remaining sequences, there is no sequences satisfying the mergeability criteria (i.e., $W_{N1} \geq L_s/2$) for merger by using AND gate.

Now, at the current level (LEVEL ONE), we have the following input sequences which cannot be merged by AND gate:

x_1 = 0101011111111101011000101111010111001010100101110 1
x_2 = 1001101110011010111111011110010011111100000100100
x_4 = 0110100001010110011001100110010110011000010111101
x_5 = 1010111101011100111000110010001010110001010011100
x_6 = 1111110101001100000100000100010000010010111011000

So we consider grouping them by OR/NOR gate, and go to step B.

B. Finding the candidate sequences for merger by OR/NOR gates.

B.1) Set up the 0-cover table for the remaining input sequences. The 0-cover table is shown in Table 3.5.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_1	x_2	x_1	x_4	x_1	x_2	x_4	x_4	x_4	x_2	x_2	x_6	x_4	x_1	x_5	x_1
x_4	x_5	x_2	x_5		x_4	x_6		x_5		x_4			x_2	x_6	x_2
								x_6		x_5					x_4
										x_6					x_5
															x_6

Table 3.5 0-cover table of Example 3.9 (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_4	x_6	x_1	x_1	x_1	x_5	x_1	x_4	x_4	x_5	x_6	x_1	x_2	x_1	x_2	x_2
x_6		x_6	x_4	x_4	x_6	x_2	x_6	x_5			x_2	x_4	x_5	x_4	x_5
			x_5	x_5		x_6		x_6			x_4	x_5		x_6	x_6
				x_6							x_5	x_6			
											x_6				

Table 3.5 0-cover table of Example 3.9 (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_6	x_1	x_1		x_1	x_4	x_1	x_2	x_1	x_1	x_2	x_1	x_2	x_2	x_6	x_1	x_2
	x_4	x_4		x_5	x_5	x_2	x_4	x_2	x_2	x_4	x_5				x_2	x_5
	x_5	x_6		x_6	x_6	x_4	x_6	x_4		x_5	x_6				x_4	x_6
	x_6					x_5		x_5							x_5	
															x_6	

Table 3.5 0-cover table of Example 3.9.

B.2) From the above 0-cover table, we obtain the frequency ordering of the literals x_1, x_2, x_4, x_5, and x_6. The numbers of columns in which the literals x_1, x_2, x_4, x_5, x_6 appear in the 0-cover table are 19, 21, 25, 24, 29, respectively. So we get the frequency ordering of literals x_1, x_2, x_4, x_5, and x_6 as:

$$x_6 > x_4 > x_5 > x_2 > x_1 \qquad (3)$$

B.3) Find the zero_maxgroup in the frequency ordering (3):

a) Select the candidate group from the frequency ordering (3), such that the number of sequences in the group is greater than $L_s/2$. The candidate group is found to be comprised of sequences x_6 and x_4.

b) From all the candidate groups, we wish to get a zero_maxgroup, i.e., a group for whcih $W_{N0} > L_s/2$, but at the same time the number of sequences in the group is the largest.

However, we cannot find this kind of zero_maxgroup of sequences at this level, which means there does not exist sequences for merger by OR/NOR gates at Level One. Go to step C.

C. Merging the remaining sequences by XOR/XNOR gates at the same level.

Here, we have these remaining sequences:

x_1 = 0101011111111101011000101111010111001010100101111 01

x_2 = 1001101110011010111111011110010011111100000100100

x_4 = 0110100001010110011001100110010110011000010111101

x_5 = 1010111101011100111000110010001010110001010011100

x_6 = 1111110101001100000100000100010000010010111011000

According to Algorithm 1, for these remaining sequences which cannot be merged either by AND/NAND or OR/NOR gates at LEVEL ONE, will now be grouped by XOR/XNOR gates. We merge the remaining sequences x_1, x_2, x_4, x_5, and x_6 by XOR gate, and the output sequence is:

y_1 = 1111011000100110101011010000110001010010110000000.

Discard x_1, x_2, x_4, x_5, x_6 from the original input sequence list.

D. Select the output sequences y_0 and y_1 obtained from previous level to form the new input sequences.

The new input sequences are now:

x_0 = 1111010101010110111100111000110111111100111010111111

x_1 = 1111011000100110101011010000011000101001011 0000000

LEVEL TWO:

Repeat steps A, B, C, and D; however, we still cannot find a one_maxgroup (or zero_maxgroup) of sequences to be merged by either AND/NAND or OR/NOR gates at LEVEL TWO; so we select gate XOR to merge them, and get the final output sequence as:

y_0 = 0000001101111000001011110100000011010101100011111 1.

This bit stream is the single output stream of the space compactor.

The space compactor is designed as follows:

433 = AND (421, 223)

434 = XOR (329, 370, 430, 431, 432)

435 = XOR (433, 434)

The logic diagram of the space compactor will be illustrated in Chapter 5.

3.9.5 Subalgorithm

Finding the max_group of sequences from the frequency ordering

3.9.5.1 Definition

a) N1: Number of matching 1s in the same bit positions of all candidate sequences in one_maxgroup.

b) N0: Number of matching 0s in the same bit positions of all candidate sequences in zero_maxgroup.

c) S1: Number of the candidate sequences, when N1> $L_s/2$.

d) S0: Number of the candidate sequences, when N0> $L_s/2$.

e) $x1_m$: Number of 1s in the sequence x_m.

f) $x0_m$: Number of 0s in the sequence x_m.

g) Good group:

A group of candidate sequences, where N1 (N0) > $L_s/2$ and S1 (S0) ≥ 2.

h) Better group:

It has the same conditions as in a good group. In addition, it is the best one in the good groups in a stage.

i) one_maxgroup/zero_maxgroup:

A group of candidate sequences, where N1(N0) $\geq L_s/2$, S1(S0) is as many as possible.

3.9.5.2 Finding one_maxgroup sequences from the frequency ordering obtained from 1-cover table

Assume a frequency ordering obtained from the 1-cover table as:

$$x_0 \geq x_1 \geq x_2 \geq x_3 \ldots \geq x_n \qquad (1)$$

1. Obtain the candidate group in which 1's number of each sequence should be greater than $L_s/2$; otherwise, the sequences (1's<$L_s/2$) will not be included in the candidate group.

Example 3.10 From above frequency ordering (1), we can get the candidate group (x_0, x_1, x_2, x_3, ..., x_m), where, $m \le n$, $xl_m \ge L_s/2$ and $xl_{m+1} < L_s/2$.

2. Select the candidate sequences.

   2.1. We start from x_0, check sequences x_0 and x_1.

      2.1.1. if $xl_0 != xl_1$, check sequences x_1 and x_2.

      if $xl_1 = xl_2$, go to sequences x_1 and x_3, and so on, until find $xl_1 != xl_i$, break;

      Then the candidate sequence set is (x_0, x_1, x_2, x_3,..., x_{i-1}).

      2.1.2. If $xl_0 = xl_1$, check x_0 and x_2 and so on, until find $xl_0 != xl_d$.

      Then the candidate sequence set is (x_0, x_1, x_2, x_3,..., x_{d-1}).

Example 3.11 Considering the following frequency ordering:

$x_0 = x_1 > x_2 > x_3 > ... > x_i > x_{i+1} > ... > x_m$,

we get the candidate sequences: (x_0, x_1).

Example 3.12 Considering the following frequency ordering:

$x_0 > x_1 = x_2 = x_3 > x_4 ... > x_i > x_{i+1} > ... > x_m$,

we get the candidate sequences: (x_0, x_1, x_2, x_3).

3. From the candidate sequences (x_0, x_1, ...x_e), get a better group.

   3.1. Start from sequence x_0. In order to obtain the 1's number in each bit position, we do bitwise AND operation.

   R1 = (x_0 AND x_1);

   N1= 1's number in R1;

   We compare N1 with $L_s/2$:

   3.1.1. if $N1 \ge L_s/2$, S1=2, go to step 3.2.

3.1.2. if $N1 < L_s/2$, we exclude x_1 from the group at the same stag and restart from x_0 and x_2 till we find $R1 = (x_0 \text{ AND } x_a)$, where $N1 \geq L_s/2$, S1=2. Stop and then go to step 3.2.

3.2. We execute $R2 = (R1 \text{ AND } x_2)$ (if from step 3.1.2, then $R2 = R1 \text{ AND } x_{a+1}$), count the 1's number in R2, i.e. N1'.

3.2.1. if $N1' \geq L_s/2$, then N1=N1'and S1=3, go to step 3.3.

3.2.2. if $N1' < L_s/2$, we exclude x_2 (or x_{a+1}) from the group under the same stag and restart from R1 and x_3 (or x_{a+2}) till we find $R2 = (R1 \text{ AND } x_b)$, where $N1' \geq L_s/2$. Stop. Then N1=N1' and S1=3. Go to step 3.3.

3.3. We execute $R3 = (R2 \text{ AND } x_3)$(if from step 3.2.2, then $R3 = R2 \text{ AND } x_{b+1}$), count the 1's number in R2, i.e. N1'.

3.3.1. if $N1' \geq L_s/2$, then N1=N1'and S1=4, go to step 3.4.

3.3.2. if $N1' < L_s/2$, we restart for R2 and x_4 (or x_{b+2}), till we find $R3 = (R1 \text{ AND } x_c)$, where $N1' \geq L_s/2$. Stop. Then N1=N1'and S1=4. Go to step 3.4.

3.4. Repeat the above steps until we exhaust all candidate sequences starting with x_0, and whenever we find the good group in the first stag (step 3.1, 3.2, 3.3).

3.5. Execute the same algorithm, starting with x_1 and get other good groups.

3.6. Repeat until we end with the sequence x_e.

3.7. From above steps, we get a group of "good groups". Select a better group from these good groups, where S1 is the maximum one.

4. From step 3, we get a better group with which N1 ($\geq L_s/2$), S1, and R (the result of bitwise AND operation of all S1 sequences) among the candidate sequences (x_0, x_1, ...x_e). Now select the new candidate sequences between R and (x_{e+1}, ..., x_m) using step 2.1.1., and get the new candidate sequences (R, x_{e+1}, ..., x_l), $l \leq m$.

5. Find a better group among candidate sequences (R, x_{e+1}, ..., x_l), $l \leq m$.

5.1. Execute R1= (R AND x_{e+1}), get N1'(1s' number in R1).

5.1.1. if N1' $\geq L_s/2$, N1=N1', S1++, go to step 5.2.

5.1.2. if N1' $< L_s/2$, we exclude x_{e+1} from the group under the same stag and restart from x_{e+2}, till we find R1=R AND x_j ($j \leq l$), where N1' $\geq L_s/2$. Stop. Then N1=N1'and S1++. Go to 5.2.

5.2. We execute R2=R1 AND x_{e+2} (if from 5.1.2, then R2=R1 AND x_{j+1}), count the 1's number in R2, i.e.N1';

5.2.1. if N1' $\geq L_s/2$, then N1=N1'and S1++; go to 5.3.

5.2.2. if N1'$< L_s/2$; we restart for R1 and x_{e+3} (or x_{j+2}), till we find R2=R1 AND x_k, where N1' $\geq L_s/2$. Stop. Then N1=N1'and S1++. Go to 5.3.

5.3. Repeat the above steps (steps 5.1-5.2) until we exhaust all candidate sequences starting with x_{e+1}, and get resulting better group, with N1 ($\geq L_s/2$), S1, and R (the result of bitwise AND of all S1 sequences).

6. Select the new candidate sequences among R and the remaining sequences in the candidate group, as in step 4. Repeat step5.

7. Repeat step 6, until we go through all the sequences in the candidate group and find a better group. In the end, the better group we get is a one_maxgroup in which N1 ($\geq L_s/2$), with S1 as large as possible.

3.9.5.3 Finding the zero_maxgroup sequences from the frequency ordering obtained from 0-cover table

To get a zero_maxgroup from the frequency ordering as obtained from the 0-cover table, use bitwise OR, and replaced 1by 0. We will get the zero_maxgroup with N0>$L_s/2$ and S0 as large as possible.

3.10 Conclusions

In this chapter we proposed a technique for designing space compaction trees for multi-output circuits in BIST based on sequence characterization and under stochastic independence of multiple line errors using concepts of cover table and frequency ordering of literals. As a specific example, C432 benchmark circuit is used as illustration.

Chapter 4

Designing Space Compactors Based on Stochastic Dependence of Multiple Line Errors

In the previous chapter, we proposed the new space compaction technique assuming stochastic independence of multiple line errors. In this chapter, we will discuss space compaction technique in which the line errors are considered stochastically dependent on one another, indicating the importance of the probability in selecting the gates for merger.

4.1 Mathematical Basis

By assuming stochastic dependence of line errors, Li and Robinson [38] defined the detectable error probability estimate $\mathcal{E}$ for a two-input logic function, given two input sequences of length L_s, as follows:

$$\mathcal{E} = S_1 \frac{R_1}{2L_s} + S_2 \frac{R_2}{L_s}$$

where,

R_1 is the number of single bit errors detected at the input sequences for that logic gate;

R_2 is the number of double bit errors detected at the input sequences for that logic gate;

S_1 is the probability of single error effect felt at the output of the CUT;

S_2 is the probability of double error effect felt at the output of the CUT.

According to the above detectable error probability estimate ($\mathcal{E}$) proposed by Li an Robinson, the following results can be deduced in the two-sequence case and the multiple-sequence case, which are discussed in detail in [5, 6]. All the related theorems are stated here as follows without proof.

For N sequences ψ_1, ψ_2, ..., ψ_N of length L_s at the output of a CUT, let the derived sequences be ψ_1', ψ_2', ..., ψ_N' with Nth-order 1-weight and 0-weight W_{N1}' and W_{N0}', respectively. Let S_i (i =1, 2, ..., N) be the probability of i-line errors. Realistically, we assume $S_1 > S_2 > \ldots > S_N$ and $S_1 + S_2 + \ldots + S_N = 1$. H_s' represents the Hamming distance between the two sequences, i.e., the number of bit positions in which these two sequences differ. Let R be the residue, that is, the number of bit positions in which two sequences in the set have different entries. Let A_i be the number of columns in the bundle set with vertical 1-weight equal to i. The binomial coefficient $\binom{N}{i}$ is simply denote by the symbol C^i or C_N^i.

Theorem 4.1 For an output sequence pair $\{\psi_1, \psi_2\}$ of length L_s and Hamming distance H_s, the detectable error probability estimate when ψ_1 and ψ_2 are merged by AND/NAND gate is:

$$\mathcal{E}(\text{AND/NAND}) = S_1 \frac{H_s + 2W_{21}'}{2L_s} + S_2 \frac{W_{20}' + W_{21}'}{L_s}.$$

Theorem 4.2 For an output sequence pair $\{\psi_1, \psi_2\}$ of length L_s and Hamming distance H_s, the detectable error probability estimate when ψ_1 and ψ_2 are merged by OR/NOR gate is:

$$\mathcal{E}(\text{OR/NOR}) = S_1 \frac{H_s + 2W_{20}'}{2L_s} + S_2 \frac{W_{20}' + W_{21}'}{L_s}.$$

Theorem 4.3 For an output sequence pair $\{\psi_1, \psi_2\}$ of length L_s and Hamming distance H_s, the detectable error probability estimate when ψ_1 and ψ_2 are merged by XOR/XNOR gate is:

$$\mathcal{E}\ (\text{XOR/XNOR}) = S_1.$$

Theorem 4.4 The Nth-order detectable error probability estimate when N sequences ψ_1, ψ_2,..., ψ_N are merged by using an N-input AND(NAND) gate is:

$$\varepsilon_N \text{ (AND/NAND)} = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times [\sum_{i=1}^{N-1} \frac{S_i}{C_i} A_{N-i} + S_N W_{N0}].$$

Theorem 4.5 The Nth-order detectable error probability estimate when N sequences ψ_1, ψ_2,..., ψ_N are merged by using an N-input OR(NOR) gate is:

$$\varepsilon_N \text{ (OR/NOR)} = \frac{W_{N0}}{L_s} + \frac{1}{L_s} \times [\sum_{i=1}^{N-1} \frac{S_i}{C_i} A_i + S_N W_{N1}].$$

Theorem 4.6 The Nth-order detectable error probability estimate when N sequences ψ_1, ψ_2,..., ψ_N are merged by using an N-input XOR(XNOR) gate is:

$$\varepsilon_N \text{ (XOR/XNOR)} = S_1 + S_3 + S_5 \ldots + S_N \text{ (assuming N odd).}$$

4.2 Derivation of the Nth-order Missed Error Probability Estimate for Individual Gates

In this section, we will introduce a new probability measure called missed error probability estimate δ (Nth-order) for AND/NAND, OR/NOR, and XOR/XNOR gates.

Let the N sequences ψ_1, ψ_2, ..., ψ_N, at the output of a CUT be of length L_s, having Nth-order 1-weight and 0-weight, W_{N1} and W_{N0}, respectively.

Definition 4.1

W_{N1}: Number of matching 1s in the same bit positions of all sequences in the bundle;

W_{N0}: Number of matching 0s in the same bit positions of all sequences in the bundle;

R: The residue (Hamming distance of multiple sequences), the number of bit positions in which at least two of the sequences in the bundle have different entries;

t_i: The probability of missing i-line errors, i = 1, 2, ...N, called missed error

probability estimate;

A_i: The number of columns in the bundle set with vertical 1-weight equal to i;

C_N^i: The binomial coefficient $\binom{N}{i}$.

Obviously, $L_s = W_{N1} + W_{N0} + R$.

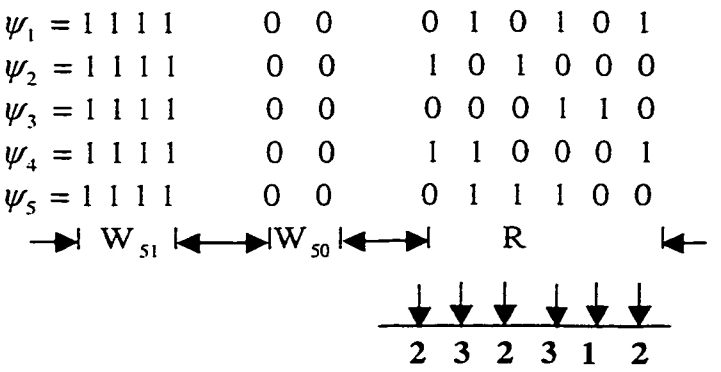
Realistically, we can assume that

$t_1 < t_2 < \ldots < t_{N-1} < t_N$

and

$t_1 + t_2 + \ldots + t_{N-1} + t_N = 1$.

Example 4.1 Consider the following 5 sequences of length $L_s = 12$.



Here the 5th-order 1-weight is $W_{51}=4$, the 5th-order 0-weight is $W_{50}=2$, and the residue R= 6, since there are 6 bit positions in which at least two of the sequences differ. The vertical 1-weights are $A_1=1$, $A_2=3$, $A_3=2$, $A_4=0$, and $A_5=4$.

Theorem 4.7 The Nth-order missed error probability estimate when N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input AND/NAND gate is:

$$\delta_N(\text{AND/NAND}) = \frac{1}{L_s}\sum_{i=0}^{N-1} A_i - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_{N-i}\right).$$

Proof:

1) Number of single line missed errors

$$(C_N^1 - 1)A_{N-1} + C_N^1 A_{N-2} + C_N^1 A_{N-3} + \ldots + C_N^1 A_{N-(N-1)} + C_N^1 A_{N-N}$$

$$= C_N^1 (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_{N-1}.$$

2) Number of double line missed errors

$$C_N^2 A_{N-1} + (C_N^2 - 1)A_{N-2} + C_N^2 A_{N-3} + \ldots + C_N^2 A_{N-(N-1)} + C_N^2 A_{N-N}$$

$$= C_N^2 (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_{N-2}.$$

3) Number of (N-1) line missed errors

$$C_N^{N-1} A_{N-1} + C_N^{N-1} A_{N-2} + C_N^{N-1} A_{N-3} + \ldots + (C_N^{N-1} - 1)A_{N-(N-1)} + C_N^{N-1} A_{N-N}$$

$$= C_N^{N-1} (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_1.$$

4) Number of N line missed errors

$$A_{N-1} + A_{N-2} + A_{N-3} + \ldots + A_{N-(N-1)}$$

$$= A_{N-1} + A_{N-2} + A_{N-3} + \ldots + A_1 + A_0 - A_0$$

$$= C_N^N (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_0.$$

δ_N (AND/NAND)

$$= \frac{t_1}{C_N^1 L_s} [C_N^1 (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_{N-1}]$$

$$+ \frac{t_2}{C_N^2 L_s} [C_N^2 (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_{N-2}]$$

$$+ \ldots$$

$$+ \frac{t_{N-1}}{C_N^{N-1} L_s} [C_N^{N-1} (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_1]$$

$$+ \frac{t_N}{C_N^N L_s} [C_N^N (A_{N-1} + A_{N-2} + \ldots + A_1 + A_0) - A_0]$$

$$=(A_0+A_1+\ \ldots\ +A_{N-2}+A_{N-1})(\frac{t_1}{C_N^1 L_s}C_N^1+\frac{t_2}{C_N^2 L_s}C_N^2+\ldots+\frac{t_{N-1}}{C_N^{N-1} L_s}C_N^{N-1}+\frac{t_N}{C_N^N L_s}C_N^N)$$

$$-\ (\frac{t_1}{C_N^1 L_s}A_{N-1}+\frac{t_2}{C_N^2 L_s}A_{N-2}+\frac{t_3}{C_N^3 L_s}A_{N-3}+\ldots+\frac{t_{N-1}}{C_N^{N-1} L_s}A_1+\frac{t_N}{C_N^N L_s}A_0)$$

$$=\frac{1}{L_s}(A_0+A_1+\ \ldots\ +A_{N-2}+A_{N-1})(t_1+t_2+\ldots+t_{N-1}+t_N)-\frac{1}{L_s}\sum_{i=1}^{N}(\frac{t_i}{C_N^i}A_{N-i})$$

$$=\frac{1}{L_s}\sum_{i=0}^{N-1}A_i\ -\frac{1}{L_s}\sum_{i=1}^{N}(\frac{t_i}{C_N^i}A_{N-i}).$$

Theorem 4.8 The Nth-order missed error probability estimate when N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input OR/NOR gate is:

$$\delta_N(\text{OR/NOR})=\frac{1}{L_s}\sum_{i=1}^{N}A_i\ -\frac{1}{L_s}\sum_{i=1}^{N}(\frac{t_i}{C_N^i}A_i).$$

Proof:

1) Number of single line missed errors

$$C_N^1 A_N+C_N^1\ A_{N-1}+C_N^1\ A_{N-2}+\ldots+C_N^1 A_2+(C_N^1-1)A_1$$

$$=C_N^1(A_1+A_2+\ldots+A_{N-1}+A_N)-A_1.$$

2) Number of double line missed errors

$$C_N^2 A_N+C_N^2 A_{N-1}+\ \ldots\ +(C_N^2-1)A_2+C_N^2\ A_1$$

$$=C_N^2\ (A_1+A_2+\ldots+A_{N-1}+A_N)-A_2.$$

3) Number of (N-1) line missed errors

$$C_N^{N-1}A_N+(C_N^{N-1}-1)A_{N-1}+\ldots+C_N^{N-1}A_2+C_N^{N-1}A_1$$

$$=C_N^{N-1}\ (A_1+A_2+\ldots+A_{N-1}+A_N)-A_{N-1}.$$

4) Number of N line missed errors

$$A_1+A_2+A_3+\ldots+A_{N-1}$$

$$= A_1 + A_2 + A_3 + \ldots + A_{N-1} + A_N - A_N$$

$$= C_N^N \ (A_1 + A_2 + A_3 + \ldots + A_{N-1} + A_N) - A_N .$$

δ_N (OR/NOR)

$$= \frac{t_1}{C_N^1 L_s} [C_N^1 (A_1 + A_2 + \ldots + A_{N-1} + A_N) - A_1]$$

$$+ \frac{t_2}{C_N^2 L_s} [C_N^2 \ (A_1 + A_2 + \ldots + A_{N-1} + A_N) - A_2]$$

$$+ \ldots$$

$$+ \frac{t_{N-1}}{C_N^{N-1} L_s} [C_N^{N-1} \ (A_1 + A_2 + \ldots + A_{N-1} + A_N) - A_{N-1}]$$

$$+ \frac{t_N}{C_N^N L_s} [C_N^N \ (A_1 + A_2 + A_3 + \ldots + A_{N-1} + A_N) - A_N]$$

$$= (A_1 + A_2 + \ldots + A_{N-1} + A_N)\left(\frac{t_1}{C_N^1 L_s} C_N^1 + \frac{t_2}{C_N^2 L_s} C_N^2 + \ldots + \frac{t_{N-1}}{C_N^{N-1} L_s} C_N^{N-1} + \frac{t_N}{C_N^N L_s} C_N^N\right)$$

$$- \left(\frac{t_1}{C_N^1 L_s} A_1 + \frac{t_2}{C_N^2 L_s} A_2 + \frac{t_3}{C_N^3 L_s} A_3 + \ldots + \frac{t_{N-1}}{C_N^{N-1} L_s} A_{N-1} + \frac{t_N}{C_N^N L_s} A_N\right)$$

$$= \frac{1}{L_s} (A_1 + A_2 + \ldots + A_{N-1} + A_N)\ (t_1 + t_2 + \ldots + t_{N-1} + t_N) - \frac{1}{L_s} \sum_{i=1}^{N} \left(\frac{t_i}{C_N^i} A_i\right)$$

$$= \frac{1}{L_s} \sum_{i=1}^{N} A_i \ - \frac{1}{L_s} \sum_{i=1}^{N} \left(\frac{t_i}{C_N^i} A_i\right).$$

Theorem 4.9 The Nth-order missed error probability estimate when N sequences ψ_1, ψ_2, ..., ψ_N are merged by using an N-input XOR/XNOR gate is:

1. when N is odd is :

   δ_N (XOR/XNOR)

   $$= t_2 + t_4 + t_6 + \ldots + t_{N-1} .$$

Proof:

1) Number of single line missed errors is 0.
2) Number of double line missed errors

$$C_N^2 (W_{N1} + W_{N0} + R).$$

3) Number of (N-1) line missed errors

$$C_N^{N-1} (W_{N1} + W_{N0} + R).$$

4) Number of N line missed errors is 0.

$$\delta_N (\text{XOR/XNOR}) = \frac{t_1}{C_N^1 L_s} [0] + \frac{t_2}{C_N^2 L_s} [C_N^2 (W_{N1} + W_{N0} + R)]$$

$$+ \ldots + \frac{t_{N-1}}{C_N^{N-1} L_s} [C_N^2 (W_{N1} + W_{N0} + R)] + \frac{t_N}{C_N^N L_s} [0]$$

$$= t_2 + t_4 + t_6 + \ldots + t_{N-1}.$$

2. when N is even is:

$$\delta_N (\text{XOR/XNOR})$$

$$= t_2 + t_4 + t_6 + \ldots + t_N.$$

Proof:

1) Number of single line missed errors is 0.
2) Number of double line missed errors

$$C_N^2 (W_{N1} + W_{N0} + R).$$

3) Number of (N-1) line missed errors is 0.
4) Number of N line missed errors

$$C_N^N (W_{N1} + W_{N0} + R).$$

$$\delta_N (\text{XOR/XNOR})$$

$$= \frac{t_1}{C_N^1 L_s} [0] + \frac{t_2}{C_N^2 L_s} [C_N^2 (W_{N1} + W_{N0} + R)]$$

$$+ \ldots + \frac{t_{N-1}}{C_N^{N-1} L_s}[0] + \frac{t_N}{C_N^N L_s}[C_N^2 (W_{N1} + W_{N0} + R)]$$

$$= t_2 + t_4 + t_6 + \ldots + t_N .$$

4.3 Mergeability Criteria and Gate Selection

In the previous section, we have deduced the Nth-order missed error probability estimates for individual gates. Now we can establish the generalized mergeability criteria for merging an arbitrary number of output sequences under conditions of stochastic dependence of line errors.

Theorem 4.10 For two N-input gates G_1 and G_2, if and only if, the Nth-order missed error probability estimate $\delta(G_1) < \delta(G_2)$, then gate G_1 is preferable to gate G_2.

4.4 Implementation

Here, we will introduce two approaches: one is an exhaustive algorithmic approach based on exact computation and the other is a heuristic approach that does not involve detailed computation for the solution of sequence mergeability and gate selection problems.

4.4.1 Exhaustive algorithmic approach

We select a particular N-input gate for merging the N sequences based on Theorem 4.10, that is, the gate G_1 is preferable to gate G_2 when the missed error probability estimate $\delta(G_1) < \delta(G_2)$. In this approach, we calculate δ(AND/NAND), δ(OR/NOR) for the N candidate sequences first, compare these results, and then select the logic gate corresponding the least value of the probability estimates. Once we decide to choose gate AND/NAND over gate OR/NOR or vice versa, then have to compare with δ(XOR/XNOR), and select the logic gate again with the least value of the probability

estimate. Unfortunately, this approach is not computationally simple and may take large CPU time in general. To avoid this drawback, we propose a simpler heuristic approach in the next section.

4.4.2 Heuristic approach

A heuristic approach has been proposed and implemented to get results within an acceptable CPU time. It might be sometimes critical to use this heuristic approach particularly because a closed-form solution of the Nth-order missed error probability estimates can be computationally intensive. The derivation below is based on N odd. The expression will be very much similar when N is even.

4.4.2.1 Definition and assumption

W_{N1}: Number of matching 1s in the same bit positions of all sequences in the bundle;

W_{N0}: Number of matching 0s in the same bit positions of all sequences in the bundle;

R: The residue (Hamming distance of multiple sequences), the number of bit positions in which at least two of the sequences in the bundle have different entries;

A_i: Number of columns in the bundle set with vertical 1-weight equal to i;

t_i: The probability of missing i-line errors;

C_N^i: The binomial coefficient $\binom{N}{i}$ or C^i.

Let:

$W_{N1} = A_0$,

$W_{N0} = A_N$,

$A_1 + A_2 + ... + A_{N-2} + A_{N-1} = R$.

(1) Case: N odd

$C_N^0 = C_N^N = 1$;

$C_N^1 = C_N^{N-1} = N;$

$C_N^2 = C_N^{N-2} = \frac{N(N-2)}{2!} = aC_N^1;$

...

$C_N^r = C_N^{N-r} = \frac{N(N-1)(N-2)...(N-r+1)}{r!} = bC_N^1;$

...

$C_N^{\frac{N-1}{2}} = C_N^{\frac{N+1}{2}} = \frac{N(N-1)(N-2)...(N-\frac{N-1}{2}+1)}{(\frac{N-1}{2})!} = qC_N^1;$

where, a, b, ..., q each is greater than 1.

Note when N is odd, there does not exist a single middle term in the array of binomial coefficients $C_N^0, C_N^1, ..., C_N^N$.

(2) Case: N even

$C_N^0 = C_N^N = 1;$

$C_N^1 = C_N^{N-1} = N;$

$C_N^2 = C_N^{N-2} = \frac{N(N-2)}{2!} = aC_N^1;$

...

$C_N^r = C_N^{N-r} = \frac{N(N-1)(N-2)...(N-r+1)}{r!} = bC_N^1;$

...

$C_N^{\frac{N}{2}-1} = C_N^{\frac{N}{2}+1} = \frac{N(N-1)(N-2)...[N-(\frac{N}{2}-1)+1]}{(\frac{N}{2}-1)!} = uC_N^1;$

$C_N^{\frac{N}{2}} = \frac{N(N-1)(N-2)...(N-\frac{N}{2}+1)}{(\frac{N}{2})!} = vC_N^1$ (middle item);

where, a, b, ..., u, v each is greater than 1.

(3) Assuming N is odd,

$t_1 < t_2 < ... < t_{N-1} < t_N$,

and

$t_1 + t_2 + ... + t_{N-1} + t_N = 1$.

Let:

$r_1 = t_{N-1} - t_1$; $r_2 = t_{N-2} - t_2$; ... ; $r_{\frac{N-1}{2}} = t_{\frac{N+1}{2}} - t_{\frac{N-1}{2}}$;

then we get: $r_1 > r_2 > ... > r_{\frac{N-1}{2}}$.

Because $t_1 + t_2 + ... + t_{N-1} + t_N = 1$,

$1 - t_N = t_1 + t_2 + ... + t_{N-1}$

$= (t_{N-1} - t_1) + (2t_1 + t_2 + ... + t_{N-2})$.

$t_{N-1} - t_1 = (1 - t_N) - (2t_1 + t_2 + ... + t_{N-2})$

$\leq 1 - t_N$,

since $2t_1 + t_2 + ... + t_{N-2} \geq 0$;

$$\frac{t_{N-1} - t_1}{1 - t_n} = \frac{r_1}{1 - t_n} \leq 1.$$

Similarly,

$$\frac{t_{N-2} - t_2}{1 - t_n} = \frac{r_2}{1 - t_n} \leq 1;$$

$$\frac{t_{N-3} - t_3}{1 - t_n} = \frac{r_3}{1 - t_n} \leq 1;$$

...

$$\frac{t_{\frac{N+1}{2}} - t_{\frac{N-1}{2}}}{1 - t_n} = \frac{r_{\frac{N-1}{2}}}{1 - t_n} \leq 1.$$

Because a, b, ..., q is greater than 1, we get :

$$\frac{r_2}{a(1-t_n)} \le 1; \quad \frac{r_3}{b(1-t_n)} \le 1; \ \ldots; \ \frac{r_{\frac{N-1}{2}}}{q(1-t_n)} \le 1.$$

Theorem 4.11 For N sequences $\psi_1, \psi_2, \ldots, \psi_N$, at the output of a CUT of length L_s, let the Nth-order 1-weight be W_{N1} and the Nth-order 0-weight be W_{N0}. For merger of the N sequences, an N-input AND/NAND gate will be preferable to an N-input OR/NOR gate for minimizing the missed error probability estimate if

$$W_{N1} - W_{N0} \ge \frac{R}{N},$$

or, $$W_{N0} - W_{N1} \le \frac{-R}{N}.$$

Proof:

We have proved that

$$\delta_N(\text{AND/NAND}) = \frac{1}{L_s}\sum_{i=0}^{N-1} A_i - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_{N-i}\right);$$

$$\delta_N(\text{OR/NOR}) = \frac{1}{L_s}\sum_{i=1}^{N} A_i - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_i\right).$$

Target:

If $\delta_N(\text{AND/NAND}) - \delta_N(\text{OR/NOR}) < 0$, the gate AND/NAND is preferable to gate OR/NOR.

$\delta_N(\text{AND/NAND}) - \delta_N(\text{OR/NOR})$

$$= \frac{1}{L_s}\sum_{i=0}^{N-1} A_i - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_{N-i}\right) - \frac{1}{L_s}\sum_{i=1}^{N} A_i + \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_i\right)$$

$$= \frac{1}{L_s}[A_0 - A_N] - \frac{1}{L_s}\left[\sum_{i=1}^{N}\frac{t_i}{C_N^i}(A_{N-i} - A_i)\right]$$

$$= \frac{1}{L_s}[A_0 - A_N] - \frac{1}{L_s}[\frac{t_1}{C_N^1}(A_{N-1} - A_1) + \frac{t_2}{C_N^2}(A_{N-2} - A_2)$$

$$+ \ldots + \frac{t_{N-1}}{C_N^{N-1}}(A_1 - A_{N-1}) + \frac{t_N}{C_N^N}(A_0 - A_N)]$$

$$= \frac{1}{L_s}[A_0 - A_N] - \frac{1}{L_s}[t_N\ (A_0 - A_N) + \frac{1}{C_N^1}(t_{N-1} - t_1)(A_1 - A_{N-1}) + \frac{1}{C_N^2}(t_{N-2} - t_2)(A_2 - A_{N-2})$$

$$+ \ldots + \frac{1}{C_N^{\frac{N-1}{2}}}(t_{\frac{N+1}{2}} - t_{\frac{N-1}{2}})(A_{\frac{N-1}{2}} - A_{\frac{N+1}{2}})]$$

$$= \frac{1}{L_s}(1 - t_N)(W_{N0} - W_{N1}) - \frac{1}{L_s}[\frac{1}{C_N^1}(t_{N-1} - t_1)(A_1 - A_{N-1}) + \frac{1}{C_N^2}(t_{N-2} - t_2)(A_2 - A_{N-2})$$

$$+ \ldots + \frac{1}{C_N^{\frac{N-1}{2}}}(t_{\frac{N+1}{2}} - t_{\frac{N-1}{2}})(A_{\frac{N-1}{2}} - A_{\frac{N+1}{2}})]$$

$$= \frac{1}{L_s}(1 - t_N)(W_{N0} - W_{N1}) - \frac{1}{L_s}Q$$

$$= \frac{1}{L_s}[(1 - t_N)(W_{N0} - W_{N1}) - Q],$$

where,

$$Q = \frac{1}{C_N^1}(t_{N-1} - t_1)(A_1 - A_{N-1}) + \frac{1}{C_N^2}(t_{N-2} - t_2)(A_2 - A_{N-2})$$

$$+ \ldots + \frac{1}{C_N^{\frac{N-1}{2}}}(t_{\frac{N+1}{2}} - t_{\frac{N-1}{2}})(A_{\frac{N-1}{2}} - A_{\frac{N+1}{2}})$$

$$= \frac{r_1}{C_N^1}(A_1 - A_{N-1}) + \frac{r_2}{aC_N^1}(A_2 - A_{N-2}) + \ldots + \frac{r_{\frac{N-1}{2}}}{qC_N^1}(A_{\frac{N-1}{2}} - A_{\frac{N+1}{2}})$$

$$= \frac{1}{C_N^1}[(r_1 A_1 + \frac{r_2}{a}A_2 + \ldots + \frac{r_{\frac{N-1}{2}}}{q}A_{\frac{N-1}{2}}) - (r_1 A_{N-1} + \frac{r_2}{a}A_{N-2} + \ldots + \frac{r_{\frac{N-1}{2}}}{q}A_{\frac{N+1}{2}})].$$

Let:

$$E_1 = r_1 A_1 + \frac{r_2}{a} A_2 + \ldots + \frac{r_{\frac{N-1}{2}}}{q} A_{\frac{N-1}{2}};$$

$$E_2 = r_1 A_{N-1} + \frac{r_2}{a} A_{N-2} + \ldots + \frac{r_{\frac{N-1}{2}}}{q} A_{\frac{N+1}{2}}.$$

$E_1 \geq 0, E_2 \geq 0.$

Then, $Q = \frac{1}{C_N^1}(E_1 - E_2);$

$(1-t_N)(W_{N0} - W_{N1}) \leq Q;$

or, $(1-t_N)(W_{N0} - W_{N1}) \leq \frac{1}{C_N^1}(E_1 - E_2),$

or, $(1-t_N)(W_{N0} - W_{N1}) \leq \frac{1}{N}(E_1 - E_2),$

or, $(W_{N0} - W_{N1}) \leq \frac{1}{N(1-t_N)}(E_1 - E_2) = F_1.$

F_1 has a minimum value N_1, when $E_1 = 0.$

That means,

$$E_1 = r_1 A_1 + \frac{r_2}{a} A_2 + \ldots + \frac{r_{\frac{N-1}{2}}}{q} A_{\frac{N-1}{2}} = 0.$$

Since $r_1, r_2, \ldots, r_{\frac{N-1}{2}} \neq 0$, then $A_1 = A_2 = \ldots = A_{\frac{N-1}{2}} = 0;$

and we get $A_{N-1} + A_{N-2} + \ldots + A_{\frac{N+1}{2}} = R.$

Now,

$$N_1 = \frac{1}{N(1-t_N)}(-E_2)$$

$$= \frac{-1}{N(1-t_N)}\left[r_1 A_{N-1} + \frac{r_2}{a} A_{N-2} + \ldots + \frac{r_{\frac{N-1}{2}}}{q} A_{\frac{N+1}{2}}\right]$$

$$=\frac{-1}{N}[\frac{r_1}{1-t_N}A_{N-1}+\frac{r_2}{a(1-t_N)}A_{N-2}+\ldots+\frac{r_{\frac{N-1}{2}}}{q(1-t_N)}A_{\frac{N+1}{2}}].$$

Previously, we have proved that $\frac{r_1}{1-t_N}$, $\frac{r_2}{a(1-t_N)}$, $\frac{r_{\frac{N-1}{2}}}{q(1-t_N)}$ have the maximum value "1".

Then N_1 has the minimum value,

$$N_{1\min}=\frac{-1}{N}[A_{N-1}+A_{N-2}+\ldots+A_{\frac{N+1}{2}}]=\frac{-R}{N},$$

i.e., $W_{N0}-W_{N1}\le\frac{-R}{N}$,

or, $W_{N1}-W_{N0}\ge\frac{R}{N}$.

Conclusion:

If $W_{N1}-W_{N0}\ge\frac{R}{N}$ or $W_{N0}-W_{N1}\le\frac{-R}{N}$, we can select gate AND/NAND over gate OR/NOR. Similarly, when N is even we can get the same result.

Theorem 4.12 For N sequences ψ_1, $\psi_2,\ldots,\psi_N$, at the output of a CUT of length L_s, let the Nth-order 1-weight be W_{N1} and the Nth-order 0-weight be W_{N0}. For merger of the N sequences, an N-input OR/NOR gate will be preferable to an N-input AND/NAND gate for minimizing the missed error probability if

$W_{N0}-W_{N1}\ge\frac{R}{N}$,

or, $W_{N1}-W_{N0}\le\frac{-R}{N}$.

The heuristic algorithm will now be described in the following sections.

4.5 Algorithm 2

Definition 4.2

Max_group:

A group of candidate sequences with the maximum value of W_{N1} (or W_{N0}) and N (≥2) parameters as found from the frequency ordering.

W_{N1}: Number of matching 1s in the same bit positions of max_group sequences;

W_{N0}: Number of matching 0s in the same bit positions of max_group sequences;

R: The residue (Hamming distance of multiple sequences);

N: Number of candidate sequences in the max_group;

t_i The probability of missing i-line errors, where $t_i = 2/3[(1/3)^{N-i}]$, for $2 \leq i \leq N$, and $1/3[(1/3)^{N-2}]$, for $i = 1$.

A) Using AND/NAND (or XOR/XNOR) gate to merge the selected candidate sequences from the original input sequences

1) Using the original input sequence list (excluding the discarded sequences), set up the 1-cover table.
2) Find the corresponding frequency ordering from the 1-cover table.
3) Try to find the max_group sequences in the frequency ordering with the maximum W_{N1} with N greater than two.
4) If $W_{N1} - W_{N0} \geq \frac{R}{N}$ or $W_{N0} - W_{N1} \leq \frac{-R}{N}$, AND/NAND gate is chosen over OR/NOR gate, but it still has to compared with XOR/XNOR gate. Go to next step 5).
   Otherwise, go to step B), that means, there are no max_group sequences to be merged by AND/NAND gate over OR/NOR gate.
5) Calculate:

$$\delta_N(\text{AND/NAND}) = \frac{1}{L_s}\sum_{i=0}^{N-1} A_i \; - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_{N-i}\right)$$

$$\delta_N(\text{XOR/XNOR}) = t_2 + t_4 + t_6 + \ldots$$

6) Compare the results from the previous step (step-5).

- If $\delta_N(\text{AND/NAND}) \le \delta_N(\text{XOR/XNOR})$, the candidate sequences will be merged by AND/NAND gate.
- If $\delta_N(\text{AND/NAND}) > \delta_N(\text{XOR/XNOR})$, the candidate sequences will be merged by XOR/XNOR gate.

7) Store the output from the above step, discarding the merged candidate sequences.

8) Repeat steps 1) ~ 7), try to find other satisfied max_group sequences which can be merged by AND/NAND (or XOR/XNOR) gates at the same level until no such sequences can be found. Go to step B).

B) Using OR/NOR (or XOR/XNOR) gate to merge the selected candidate sequences from the original input sequences

1) Using the original input sequence (excluding the discarded sequences), set up the 0-cover table.

2) Find the corresponding frequency ordering in the 0-cover table.

3) Try to find the max_group sequences from the frequency ordering.
i.e., Try to find the max_group sequences in the frequency ordering with the maximum W_{N0}, with N as large as possible.

4) If $W_{N0} - W_{N1} \ge \frac{R}{N}$ or $W_{N1} - W_{N0} \le \frac{-R}{N}$, OR/NOR gate is chosen over AND/NAND gate, but it still has to compared with XOR/XNOR gate. Go to next step 5).
Otherwise, go to step C), that means, there are no sequences that can be merged by OR/NOR gate.

5) Calculate:

$$\delta_N(\text{OR/NOR}) = \frac{1}{L_s}\sum_{i=1}^{N} A_i \; - \frac{1}{L_s}\sum_{i=1}^{N}\left(\frac{t_i}{C_N^i} A_i\right)$$

$$\delta_N(\text{XOR/XNOR}) = t_2 + t_4 + t_6 + \ldots$$

6) Compare the results as obtained from step 5).

- If δ_N (OR/NOR) $\leq \delta_N$ (XOR/XNOR), the candidate sequences will be merged by OR/NOR gate.
- If δ_N (OR/NOR) $> \delta_N$ (XOR/XNOR), the candidate sequences will be merged by XOR/XNOR gate.

7) Store the outputs from the above step, discarding the candidate sequences.

8) Repeat steps 1) ~ 7), try to find other satisfied max_group sequences which can be merged by OR/NOR (or XOR/XNOR) gates at the same level until no such sequences can be found. Go to step C).

C) Merging remaining sequences by AND/NAND, OR/NOR or XOR/XNOR gates

1) Compute δ_N (AND/NAND), δ_N (OR/NOR) and δ_N (XOR/XNOR), then compare the results:

- If δ_N (AND/NAND) $\leq \delta_N$ (OR/NOR) and δ_N (AND/NAND) $\leq \delta_N$ (XOR/XNOR), the candidate sequences will be merged by AND/NAND gate.
- If δ_N (OR/NOR) $\leq \delta_N$ (AND/NAND) and δ_N (OR/NOR) $\leq \delta_N$ (XOR/XNOR), the candidate sequences will be merged by OR/NOR gate.
- Otherwise, the candidate sequences will be merged by XOR/XNOR gate.

2) Store the resulting output sequence, discarding grouped sequences.

D) Setting up the new original input sequence list comprised of the sequences obtained at the previous level, repeating steps A) ~ C), a single output sequence is obtained.

4.6 Explanation of the Algorithm 2 by an Example

Consider the following sequences at the output of circuit c432, they will be grouped according to Algorithm 2 until a single output is obtained.

Example 4.2 c432 Benchmark Circuit

The output sequences for the circuit c432 are as follows:

x_0 = 1111011101010110111110111001110111111110111011111 (Line 223)
x_1 = 0101011111111101011000101111010111001010100101101 (Line 329)
x_2 = 1001101110011010111111011110010011111100000100100 (Line 370)
x_3 = 1111110101111110111101111110111111111001111111111 (Line 421)
x_4 = 0110100001010110011001100110010110011000010111101 (Line 430)
x_5 = 1010111101011100111000110010001010110001010011100 (Line 431)
x_6 = 1111110101001100000100000100010000010010111011000 (Line 432)

The length of sequence is 49.

LEVEL ONE:

A) Using AND/NAND (or XOR/XNOR) gate to merge the selected candidate sequences from the input sequence list

A.1) Set up the 1-cover table using the original input sequence. The 1-cover table is shown in Table 4.1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_0	x_0	x_0	x_0	x_2	x_0	x_0	x_0	x_1	x_0	x_1	x_0	x_1	x_0	x_0	
x_2	x_1	x_3	x_1	x_3	x_1	x_1	x_1	x_2	x_1	x_3	x_1	x_2	x_3	x_1	
x_3	x_3	x_4	x_2	x_4	x_3	x_2	x_2		x_3		x_2	x_3	x_4	x_2	
x_5	x_4	x_5	x_3	x_5	x_5	x_5	x_3		x_4		x_3	x_5	x_5	x_3	
x_6	x_6	x_6	x_6	x_6	x_6		x_5		x_5		x_4	x_6	x_6	x_4	
							x_6		x_6		x_5				

Table 4.1 The 1-cover table of Example 4.2 (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_0	x_0	x_0	x_0	x_0	x_1	x_0	x_0	x_0	x_1	x_1	x_0	x_0	x_0	x_1	x_0
x_1	x_1	x_2	x_2	x_2	x_2	x_3	x_1	x_1	x_2	x_2		x_1	x_2	x_3	x_1
x_2	x_2	x_3	x_3		x_3	x_4	x_2	x_2	x_3	x_3		x_3	x_3	x_5	x_3
x_3	x_3	x_4	x_6		x_4	x_5	x_3	x_3	x_4	x_4			x_4		x_4
x_5	x_4	x_5					x_5		x_6	x_5			x_6		
	x_5														

Table 4.1 The 1-cover table of Example 4.2 (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_0	x_0	x_0	x_0	x_0	x_0	x_6	x_0	x_0	x_0	x_1	x_0	x_0	x_0	x_0	x_0	x_0
x_1	x_2	x_2	x_1	x_2	x_1		x_1	x_3	x_3	x_3	x_2	x_1	x_1	x_1	x_3	x_1
x_2	x_3	x_3	x_2	x_3	x_2		x_3	x_6	x_4	x_6	x_3	x_3	x_3	x_2		x_2
x_3		x_5	x_3	x_4			x_5		x_5		x_4	x_4	x_4	x_3		x_4
x_4			x_4						x_6			x_5	x_5	x_4		
x_5			x_5									x_6	x_6	x_5		
			x_6													

Table 4.1 The 1-cover table of Example 4.2.

A.2) From the obtained 1-cover –table (Table 4.1), we can get the corresponding frequency ordering of literals x_0, x_1, x_2, x_3, x_4, x_5, and x_6 as follows:

$$x_3 > x_0 > x_1 > x_2 > x_5 > x_4 > x_6 \qquad (1)$$

A.3) To get the max_group sequences from the frequency ordering (1):

Sequences x_3 and x_0 constitute the max_group, where its maximum $W_{21}=34$,

A.4) Computing:

A_0=3; A_1=12; A_2=34; W_{21}=34; W_{20}=3; R=12; N=2,

we get W_{21}-W_{20} = 31 > R/N = 6;

Hence AND/NAND gate is chosen over OR/NOR gate, but it still has to compared with XOR/XNOR gate.

A.5) Calculate:

δ_2(AND/NAND)= 0.224490 < δ_2(XOR/XNOR)= 0.666667

So the candidate sequences x_3 and x_0 are merged by AND gate and the output sequence of the AND operation is y_0. The sequences x_3 and x_0 are discarded from the input sequence list and replaced by their output y_0 which is:

y_0 = 1111010101010110111100111000110111111001110111111

The remaining input sequences are as follows:

x_1 = 0101011111111010110001011110101110010101001011101

x_2 = 1001101110011010111111011110010011111100000100100

x_4 = 0110100001010110011001100110010110011000010111101

x_5 = 1010111101011100111000110010001010110001010011100

x_6 = 1111110101001100000100000100010000010010111011000

A.6) Form the new 1-cover table using the remaining input sequences (i.e., x_1, x_2, x_4, x_5 and x_6) as shown in Table 4.2.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_2	x_1	x_4	x_1	x_2	x_1	x_1	x_1	x_1	x_1	x_1	x_1	x_1	x_4	x_1	
x_5	x_4	x_5	x_2	x_4	x_5	x_2	x_2	x_2	x_4		x_2	x_2	x_5	x_2	
x_6	x_6	x_6	x_6	x_5	x_6	x_5	x_5		x_5		x_4	x_5	x_6	x_4	
				x_6			x_6		x_6		x_5	x_6			

Table 4.2 New 1-cover table (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_1	x_1	x_2	x_2	x_2	x_1	x_4	x_1	x_1	x_1	x_1		x_1	x_2	x_1	x_1
x_2	x_2	x_4	x_6		x_2	x_5	x_2	x_2	x_2	x_2			x_4	x_5	x_4
x_5	x_4	x_5			x_4		x_5		x_4	x_4			x_6		
	x_5								x_6	x_5					

Table 4.2 New 1-cover table (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_1	x_2	x_2	x_1	x_2	x_1	x_6	x_1	x_6	x_4	x_1	x_2	x_1	x_1	x_1		x_1
x_2		x_5	x_2	x_4	x_2		x_5		x_5	x_6	x_4	x_4	x_4	x_2		x_4
x_4			x_4						x_6			x_5	x_5	x_4		
x_5			x_5									x_6	x_6	x_5		
			x_6													

Table 4. 2 New 1-cover table.

A.7) From the new 1-cover-table (Table 4.2), get the corresponding frequency ordering as:

$$x_1 > x_2 > x_5 > x_4 > x_6 \qquad (2)$$

A.8) To get the max_group sequences from the frequency ordering (2):

Sequences x_1 and x_2 constitute the max_group, since their $W_{21}=18$ has the largest value.

A.9) Computing:

$A_0=9$; $A_1=22$; $A_2=18$; $W_{21}=18$; $W_{20}=9$; R=22; N=2,

$W_{21}-W_{20} = 9 < R/N = 11$;

As a result, sequences x_1 and x_2 cannot be merged by AND/NAND gate at this level.

The remaining sequences are as follows:

$x_1 = 0101011111111010110001011110101110010101001011101$
$x_2 = 1001101110011010111111011110010011111100000100100$
$x_4 = 0110100001010110011001100110010110011000010111101$
$x_5 = 1010111101011100111000110010001010110001010011100$
$x_6 = 1111110101001100000100000100010000010010111011000$

Until now, we cannot find a max_group of sequences for merger by AND/NAND gate at LEVEL ONE, i.e., the condition (W_{N1} -W_{N0} $\geq \frac{R}{N}$ or W_{N0} -W_{N1} $\leq \frac{-R}{N}$) is not satisfied. So we go to step B) to find the sequences which can be merged by OR/NOR gate.

B) Using OR/NOR (or XOR/XNOR) gate to merge the selected candidate sequences from the original input sequences

B.1) Set up the 0-cover table using the remaining original input sequences. The 0-cover table is shown in Table 4.3.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_1	x_2	x_1	x_4	x_1	x_2	x_4	x_4	x_4	x_2	x_2	x_6	x_4	x_1	x_5	x_1
x_4	x_5	x_2	x_5		x_4	x_6		x_5		x_4			x_2	x_6	x_2
								x_6		x_5					x_4
										x_6					x_5
															x_6

Table 4.3 0-cover table of Example 4.2 (cont.).

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
x_4	x_6	x_1	x_1	x_1	x_5	x_1	x_4	x_4	x_5	x_6	x_1	x_2	x_1	x_2	x_2
x_6		x_6	x_4	x_4	x_6	x_2	x_6	x_5			x_2	x_4	x_5	x_4	x_5
			x_5	x_5		x_6		x_6			x_4	x_5		x_6	x_6
				x_6							x_5	x_6			
											x_6				

Table 4.3 0-cover table of Example 4.2 (cont.).

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
x_6	x_1	x_1		x_1	x_4	x_1	x_2	x_1	x_1	x_2	x_1	x_2	x_2	x_6	x_1	x_2
	x_4	x_4		x_5	x_5	x_2	x_4	x_2	x_2	x_4	x_5				x_2	x_5
	x_5	x_6		x_6	x_6	x_4	x_6	x_4		x_5	x_6				x_4	x_6
	x_6					x_5		x_5							x_5	
															x_6	

Table 4.3 0-cover table of Example 4.2.

B.2) From the above 0-cover–table (Table 4.3), we get the corresponding frequency ordering of literals x_1, x_2, x_4, x_5, and x_6 as follows:

$$x_6 > x_4 > x_5 > x_2 > x_1 \qquad (3)$$

B.3) To find the max_group in the frequency ordering (3)

The sequences x_6 and x_4 constitute of the max_group whose value of W_{20} is 16.

Computing:

A_0=16; A_1=22; A_2=11; W_{21}=11; W_{20}=16; R=22; N=2,

$W_{20} - W_{21} = 5 < R/N = 11$;

Hence, the sequences x_6 and x_4 cannot be merged by gate OR/NOR at this level.

The remaining 5 sequences are as follows:

$x_1 = 0101011111111010110001011110101110010101001011101$

$x_2 = 1001101110011010111111011110010011111100000100100$

$x_4 = 0110100001010110011001100110010110011000010111101$

$x_5 = 1010111101011100111000110010001010110001010011100$

$x_6 = 1111110101001100000100000100010000010010111011000$

So far, we cannot find a max_group of sequences to be merged by OR/NOR gate at LEVEL ONE, i.e., the condition ($W_{N0} - W_{N1} \geq \frac{R}{N}$ or $W_{N1} - W_{N0} \leq \frac{-R}{N}$) is not satisfied. So we go to step C) and try to find the sequences which can be merged by AND/OR/XOR gate.

C) Merging the remaining sequences by AND/NAND, or OR/NOR, or XOR/XNOR gate

C.1) Compute:

δ_5(AND/NAND) = 0.930058

δ_5(OR/NOR) = 0.911212

δ_5(XOR/XNOR) = 0.246914

C.2) Compare the results:

δ_5(XOR/XNOR)< δ_5(OR/NOR)< δ_5(AND/NAND);

Hence, the sequences x_1, x_2, x_4, x_5 and x_6 can be merged by XOR/XNOR gate, and the output sequence is:

$y_1 = 1111011000100110101011010000110001010010110000000$

D) The sequences y_0 and y_1 obtained from previous level to form the new input sequences

The new input sequences are as follows:

x_0 = 1111010101010110111100111000110111111001110111111

x_1 = 1111011000100110101011010000110001010010110000000

LEVEL TWO:

Repeat steps A and B. We get A_0=10; A_1=23; A_2=16; W_{21}=16; W_{20}=10; R=25; N=2.

However, it does not satisfy any of the following two conditions

- $W_{N1} - W_{N0} \geq \frac{R}{N}$ or $W_{N0} - W_{N1} \leq \frac{-R}{N}$
- $W_{N0} - W_{N1} \geq \frac{R}{N}$ or $W_{N1} - W_{N0} \leq \frac{-R}{N}$

We, therefore, cannot simply decide to merge the above two sequences by one of the three gates (AND/OR/XOR). The decision is dependent on the value of the missed error probability estimate of the individual gates.

They are as follows:

δ_2(AND/NAND) = 0.459184

δ_2(OR/NOR) = 0.500000

δ_2(XOR/XNOR) = 0.666667

Comparing the results:

δ_2(AND/NAND)< δ_2(OR/NOR)< δ_2(XOR/XNOR);

So the sequences x_0 and x_1 can be merged by AND gate, and the output sequence is going to be:

y_0 = 1111010000000110101000010000110001010000110000000

This bit stream is then the single output of the space compactor.

The space compactor is comprised of:

433 = AND (421, 223)

434 = XOR (329, 370, 430, 431, 432)

435 = AND (433, 434)

This space compactor tree is composed of two AND gates and one XOR gate and is shown in Chapter 5.

4.5 Summary

In this chapter, we developed algorithms to design space compaction trees for multi-output circuits under conditions of stochastic dependence of multiple line errors. A specific example based on the benchmark circuit c432 is explained in detail to explain our algorithms.

Chapter 5

Experimental Results and Discussions

5.1 Simulation Methods

In the previous two chapters, we proposed the generalized mergeability criteria for two cases: assuming stochastic independence of multiple line errors and stochastic dependence of multiple line errors. Based on these generalized mergeablility criteria, the space compressors were designed for the ISCAS 85 combinational benchmark circuits [8]. According to the gate selection criteria as previously explained, the space compressors were composed of a series of selected AND/NAND, OR/NOR and XOR/XNOR logic gates.

In order to investigate the feasibility of the proposed space compression methods, independent sets of simulations were executed for the ISCAS 85 combinational benchmark circuits using ATALANTA, FSIM and COMPACTEST programs. ATALANTA [36] (a logic and fault simulation program developed at the Virginia Polytechnic Institute and State University) and COMPACTEST [43] programs generate reduced test sets that detect most of the detectable SSL faults. FSIM fault simulation program [37] is also used to generate pseudorandom test sets for the circuits.

For comparison purposes, we also used the parity tree space compactor composed of XOR gates only, considered ideal for space compaction, since it propagates all errors appearing on an odd number of inputs. Having the parity tree space compactor as a reference, we simulated the ISCAS 85 combinational benchmark circuits, and also indicated the feasibility of the proposed approaches of constructing space compression trees in terms of the concepts developed in the thesis.

For each ISCAS 85 combinational circuit, several variables were determined and listed in tabular format. These include the number of test vectors to be used to construct the compaction tree, the CPU time required to construct the compaction tree, the number of applied test vector corresponding the obtained tree, the simulation CPU time and fault coverage by running ATALANTA and FSIM programs on the Sparc 5 SUN workstation

and COMPACTEST on IBM AIX machine respectively. In addition, we also estimated the hardware overhead for the space compactors and gave the space compaction circuits for c432 benchmark circuit.

5.2 Simulation Results

5.2.1 Simulation results without space compactors

To get a realistic estimate of the fault coverage, we first carried out the simulation without compactor.

In Table 5.1, 5.2 and 5.3, we provide the fault coverage and the CPU simulation time for the ISCAS 85 benchmark circuits without the space compactors running ATALANTA, FSIM and COMPACTEST fault simulators, respectively.

Circuit name	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	8	0.033	100.000
c432	74	0.150	99.237
c499	65	0.133	98.945
c880	102	0.283	100.000
c1355	105	0.317	99.492
c1908	182	0.900	99.521
c2670	192	2.933	95.741
c3450	253	3.933	96.004
c5315	223	2.533	98.897
c6288	58	7.950	99.561
c7552	354	18.517	98.265

Table 5.1 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA without the space compactors.

Circuit name	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	32	0.017	100.000
c432	224	0.050	98.092
c499	224	0.050	96.966
c880	224	0.067	94.798
c1355	224	0.100	92.376
c1908	224	0.183	85.311
c2670	224	0.233	82.490
c3450	224	0.467	86.844
c5315	224	0.417	96.430
c6288	224	1.050	99.561
c7552	224	0.717	89.894

Table 5.2 Simulation results of the ISCAS 85 benchmark circuits using FSIM without the space compactors.

Circuit name	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.00	100.000
c432	44	4.70	99.237
c499	63	10.40	98.945
c880	30	0.80	100.000
c1355	96	13.76	99.492
c1908	137	14.43	99.521
c2670	68	102.399	95.741
c3450	110	240.65	95.916
c5315	55	32.109	98.897
c6288	16	64.919	99.561
c7552	85	136.26	98.265

Table 5.3 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST without the space compactors.

From the above tables, we see that the fault coverage result provides by the ATALANTA is almost the same as that provided by the COMPACTEST. The FSIM provided the least fault coverage result among the three simulators, though it provides the best CPU simulation time followed by ATALANTA.

5.2.2 Simulation results by assuming stochastic independence of multiple line errors

The simulation results assuming stochastic independence of multiple line errors are shown in Table 5.4, 5.5 and 5.6.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.1000	9	0.017	100.000
c432	49	0.2833	92	0.250	97.917
c499	54	0.5000	68	0.150	100.000
c880	50	0.7167	130	0.433	98.087
c1355	85	0.5333	104	0.333	100.000
c1908	118	0.7333	195	1.200	98.352
c2670	108	12.1667	181	19.900	87.368
c3450	149	1.3000	313	5.833	93.878
c5315	122	10.7167	290	7.267	97.222
c6288	28	0.4833	89	12.000	99.561
c7552	208	11.2667	374	48.367	94.738

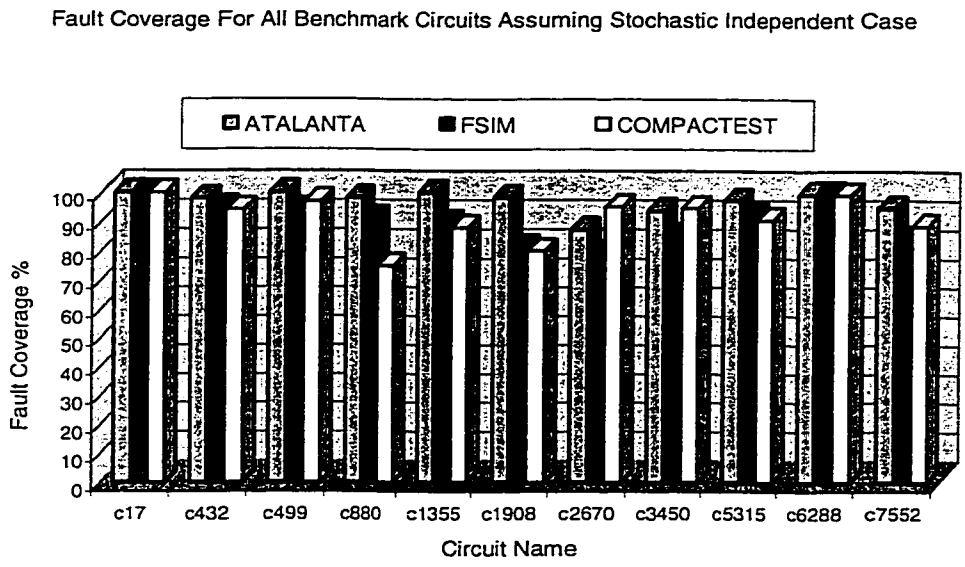
Table 5.4 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming stochastic independence of multiple line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.1000	32	0.017	100.00
c432	49	0.2833	224	0.050	94.697
c499	54	0.5000	224	0.067	94.211
c880	50	0.7167	224	0.100	91.180
c1355	85	0.5333	224	0.117	89.721
c1908	118	0.7333	224	0.200	81.127
c2670	108	12.1667	224	0.267	79.189
c3450	149	1.3000	224	0.483	85.044
c5315	122	10.7167	224	0.467	92.378
c6288	28	0.4833	224	1.850	99.561
c7552	208	11.2667	224	0.883	84.292

Table 5.5 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic independence of multiple line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.1167	6	0.010	100.000
c432	44	0.2500	74	17.679	93.962
c499	63	1.0833	71	132.300	96.883
c880	30	0.650	17	79.679	74.842
c1355	96	3.0167	45	231.199	87.890
c1908	137	0.7833	31	709.870	80.011
c2670	68	6.6500	126	142.93	95.049
c3450	110	1.0167	225	445.320	94.645
c5315	55	5.0833	86	1070.159	90.561
c6288	16	0.9500	87	904.289	99.239
c7552	85	7.3000	354	5704.410	88.537

Table 5.6 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic independence of multiple line errors.



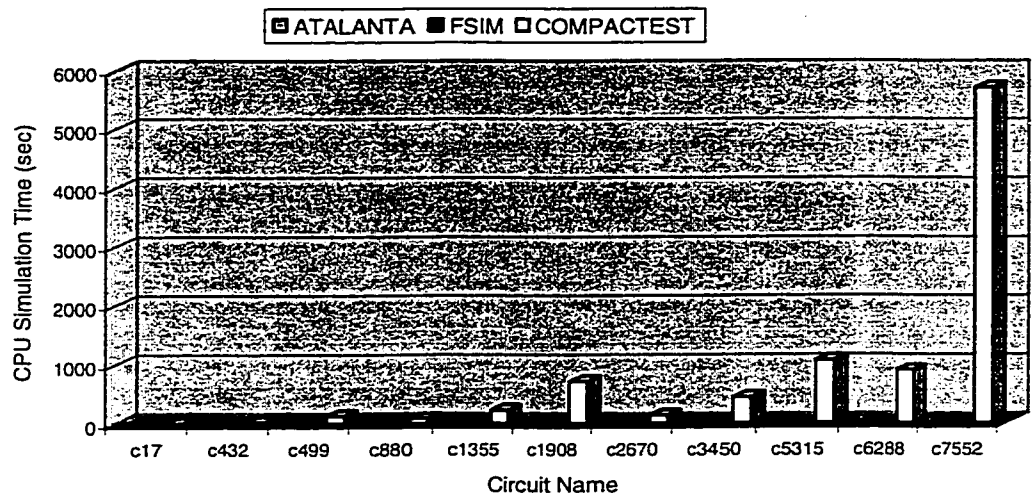


Figure 5.1 Simulation results for all benchmark circuits using ATALANTA, FSIM and COMPACTEST programs and assuming stochastic independence of multiple line errors.

Compared with the simulation results without space compactors, the simulation results with space compactors show that all the values (including CPU simulation time, fault coverage) are changed to varying degrees due to inclusion of the compressors. From Table 5.4 to Table 5.6 as well as from Figure 5.1, we see that the fault coverage results of all benchmark circuits are decreased. Under stochastic independence of multiple line errors, ATALANTA provides the highest fault coverage among all three simulators ranging from 87.368% to 100.000%. The CPU simulation time provided by FSIM is the best, being less than 1 sec.

5.2.3 Simulation results by assuming stochastic dependence of multiple line errors

5.2.3.1 Simulation result by exhaustive approach

For comparison purpose, we also developed the exhaustive approach as mentioned in the previous chapters and simulated all the ISCAS 85 benchmark circuits using ATALANTA, FSIM and COMPACTEST. The results are shown in Table 5.7, Table 5.8 and Table 5.9.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree (sec)	No of applied test vectors	CPU simulation times (sec)	Fault coverage (%)
c17	5	0.267	10	0.000	100.000
c432	49	2.483	109	0.367	92.748
c499	54	32.667	171	0.350	98.681
c880	50	22.483	229	0.850	99.257
c1355	85	34.533	191	0.967	98.602
c1908	118	22.533	346	2.850	99.308
c2670	108	779.717	297	8.050	90.532
c3540	149	19.017	476	8.433	95.858
c5315	122	591.333	522	10.700	95.475
c6288	28	32.017	221	35.483	99.561
c7552	208	415.250	735	43.600	95.865

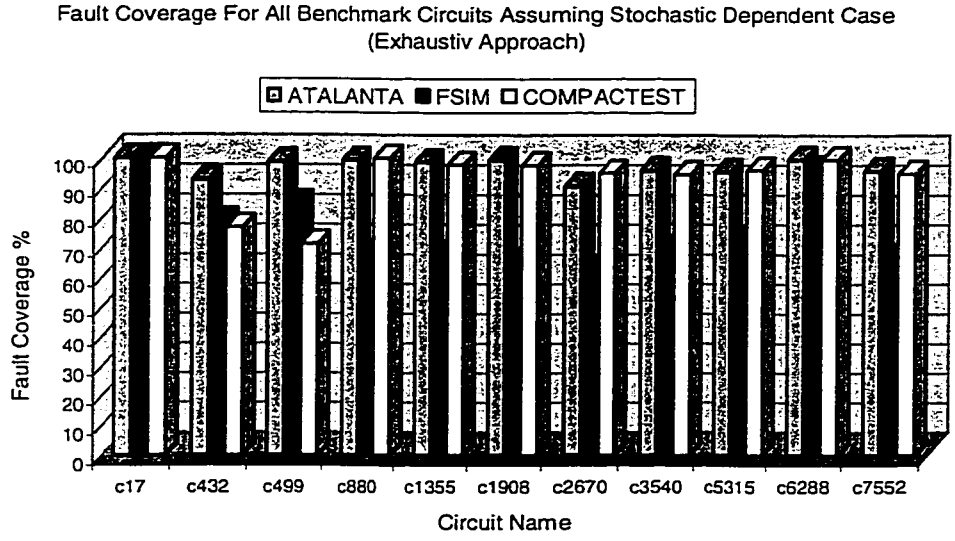
Table 5.7 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming stochastic dependence of multiple line errors (Using exhaustive approach).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree (sec)	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.267	160	0.017	100.000
c432	49	2.483	224	0.083	78.817
c499	54	32.667	224	0.083	84.828
c880	50	22.483	224	0.117	67.622
c1355	85	34.533	224	0.167	67.789
c1908	118	22.533	224	0.267	65.194
c2670	108	779.717	224	0.333	62.819
c3540	149	19.017	224	0.733	69.166
c5315	122	591.333	224	0.633	72.569
c6288	28	32.017	224	2.800	96.320
c7552	208	415.250	224	1.167	66.711

Table 5.8 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic dependence of multiple line errors (Using exhaustive approach).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree (sec)	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.200	6	0.010	100.000
c432	44	2.650	47	80.420	76.679
c499	63	34.950	12	305.250	71.098
c880	30	20.983	106	3.610	100.000
c1355	96	32.633	126	75.880	97.738
c1908	137	22.733	208	102.440	97.714
c2670	68	807.083	204	128.650	94.941
c3540	110	18.300	233	476.740	94.697
c5315	55	547.133	300	259.800	96.038
c6288	16	32.150	72	92.570	99.564
c7552	85	333.250	501	1166.040	94.812

Table 5.9 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic dependence of multiple line errors (Using exhaustive approach).



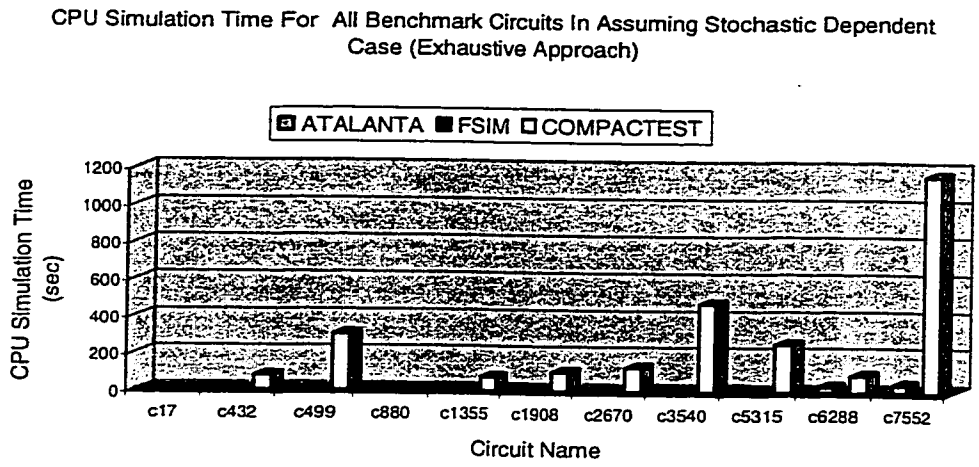


Figure 5.2 Simulation results for all benchmark circuits using ATALANTA, FSIM and COMPACTEST programs and assuming stochastic dependence for multiple line errors (By Exhaustive Approach).

It is worth noting that for each max_group, t_i is the probability of missing i-line errors, which equals to $2/3[(1/3)^{N-i}]$, for $2 \le i \le N$, and $1/3[(1/3)^{N-2}]$, for $i = 1$.

In the case of stochastic dependence for multiple line errors, we see that ATALANTA provides the best fault coverage results, and FSIM provides the shortest CPU simulation time while using exhaustive approach.

5.2.3.2 Simulation results by heuristic approach

We proposed the heuristic approach to reduce the CPU time for constructing the space compactors assuming stochastic dependence of multiple line errors. The simulation results of ISCAS 85 benchmark circuits using ATALANTA, FSIM and COMPACTEST are shown in Table 5.10, 5.11 and 5.12 respectively.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.450	10	0.000	100.000
c432	49	1.867	120	0.583	77.376
c499	54	10.417	75	0.200	100.00
c880	50	18.417	183	0.667	96.308
c1355	85	11.050	128	0.567	98.160
c1908	118	8.233	248	1.917	95.587
c2670	108	493.517	308	9.117	90.000
c3450	149	10.300	328	7.550	94.315
c5315	122	524.017	478	14.567	94.002
c6288	28	21.517	133	28.867	97.999
c7552	208	294.083	329	59.317	92.846

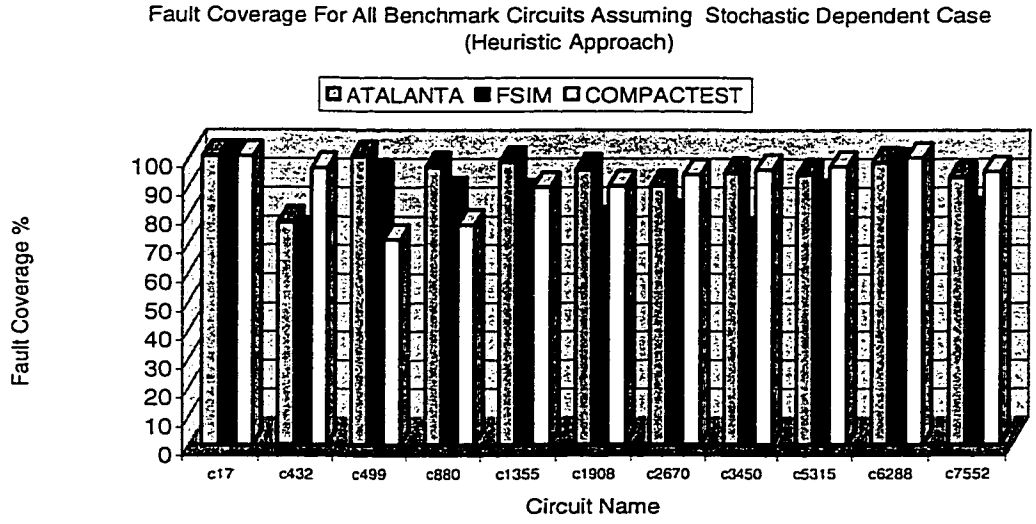
Table 5.10 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming stochastic dependence of multiple line errors (Using Heuristic approach).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.450	64	0.033	100.000
c432	49	1.867	224	0.067	73.004
c499	54	10.417	224	0.050	92.763
c880	50	18.417	224	0.083	87.447
c1355	85	11.050	224	0.117	86.675
c1908	118	8.233	224	0.183	77.193
c2670	108	493.517	224	0.250	79.636
c3450	149	10.300	224	0.650	73.703
c5315	122	524.017	224	0.517	86.622
c6288	28	21.517	224	2.283	97.586
c7552	208	294.083	224	0.967	80.578

Table 5.11 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic dependence of multiple line errors (Using Heuristic approach).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.250	6	0.000	100.000
c432	44	2.133	53	17.170	96.030
c499	63	34.717	12	306.100	71.098
c880	30	15.683	17	118.810	76.294
c1355	96	29.950	56	282.870	89.370
c1908	137	12.417	76	473.790	89.677
c2670	68	544.100	236	134.910	93.686
c3540	110	12.033	157	351.700	95.531
c5315	55	486.800	159	230.310	96.480
c6288	16	23.333	41	129.540	99.563
c7552	85	307.250	351	1655.800	94.767

Table 5.12 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic dependence of multiple line errors (Using Heuristic approach).



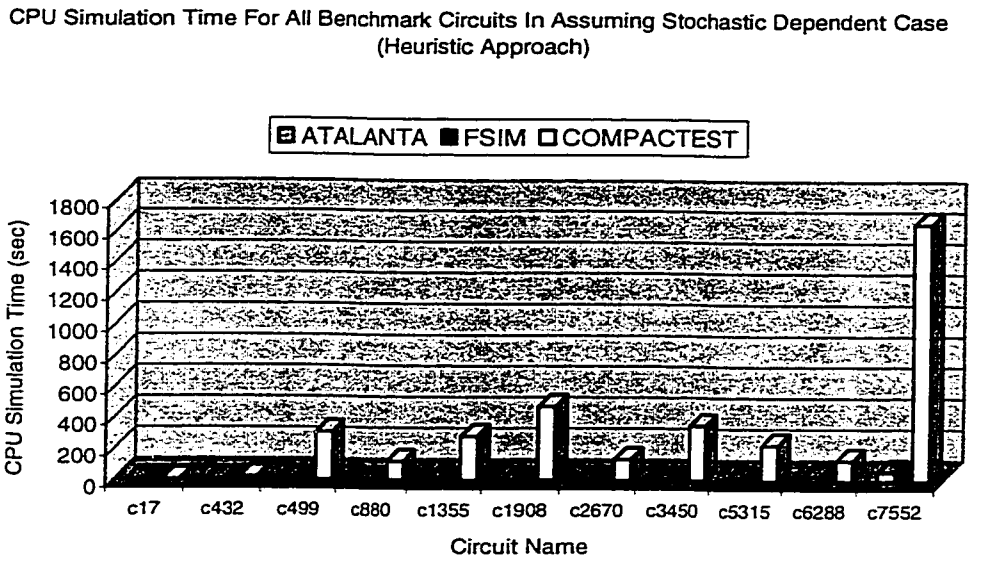


Figure 5.3 Simulation results for all benchmark circuits using ATALANTA, FSIM and COMPACTEST programs and assuming stochastic dependence for multiple line errors (Heuristic Approach).

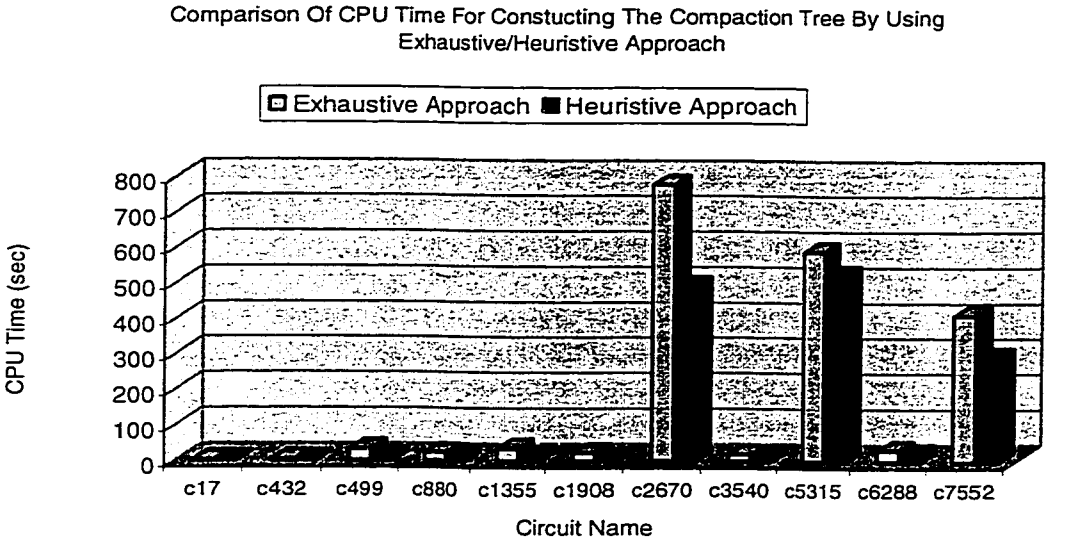


Figure 5.4 Comparison of the CPU time to construct the compaction tree using ATALANTA/FSIM with exhaustive approach and heuristic approach.

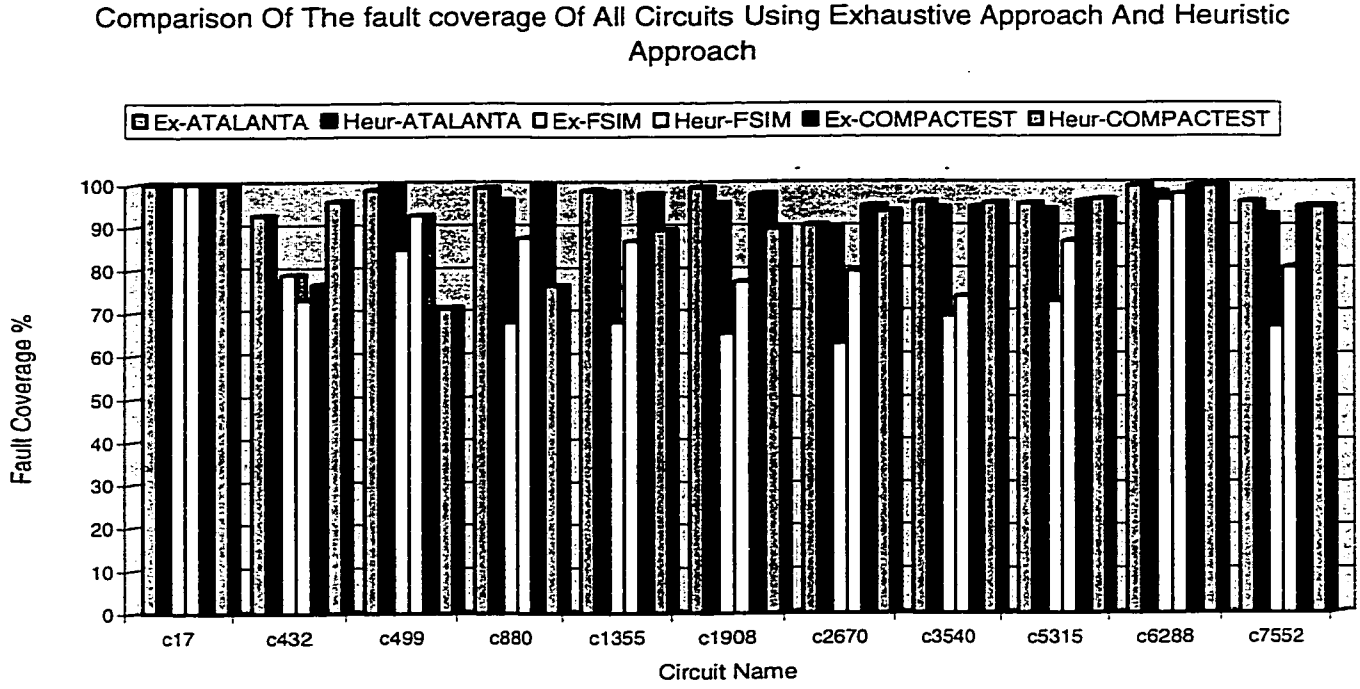


Figure 5.5. Comparison of the fault coverage for ISCAS 85 benchmark circuits in the cases of exhaustive approach and heuristic approach using ATALANTA, FSIM and COMPACTEST simulators.

Again, it is worth noting that for each max_group, t_i is the probability of missing i-line errors, which equals to $2/3[(1/3)^{N-i}]$, for $2 \leq i \leq N$, and $1/3[(1/3)^{N-2}]$, for $i = 1$.

As expected, the CPU time for constructing the space compactor is reduced by heuristic approach as compared to the exhaustive approach (see Figure 5.4), specially for circuits with a great number of primary outputs. For example, to construct the space compactor tree for c2670 benchmark circuit, the consumed CPU time using the heuristic approach is about 37% less (ATALANT/FSIM) or 33% (COMPACTEST) than those using the exhaustive approach. The results on fault coverage (using ATALATNA/COMPACTEST) by heuristic approach show some reduction, though most are still above 80%. We also found that the fault coverage (using FSIM) by heuristic approach, on the contrary, are higher than those by exhaustive approach. Those simulation results demonstrate that the proposed heuristic approach not only reduces the CPU time to construct the space compactors, but also ensures the relative high fault

coverage. Moreover, the ATALANTA provides the best fault coverage results, while FSIM provides the best results in terms of CPU simulation time.

5.2.3.3 Simulation results at different missed error probability estimates

In section 5.2.3.2, we defined the missed error probability estimate of N sequences (i.e., t_N) to be 2/3, and the probability is decreased while the number of sequences is reduced. That means, $t_N > t_{N-1} > \ldots > t_2 > t_1$ and $t_1 + t_2 + \ldots + t_N = 1$. In this section, we study the simulation results with various values of t_N, like $t_N = 0.5$, 0.6, 0.7, 0.8, 0.9 and 1.0 respectively. We still assume $t_N > t_{N-1} > \ldots > t_2 > t_1$ and $t_1 + t_2 + \ldots + t_N = 1$. The simulation results are shown in Table 5.13, 5.14 and 5.15.

Circuit Name	No. of test vector s used to construct the compaction tree	Fault coverage(%) (t_n=0.5)	Fault coverage (%) (t_n=0.6 or 0.7 or 0.8 or 0.9)	Fault coverage (%) (t_n=2/3)	Fault coverage (%) (t_n=1.0)
c17	5	100.000	100.000	100.000	100.00
c432	49	78.137	77.567	77.376	78.327
c499	54	100.000	100.000	100.000	98.082
c880	50	98.404	96.730	96.308	97.766
c1355	85	97.208	96.447	98.160	97.736
c1908	118	96.917	96.438	95.587	96.863
c2670	108	90.000	90.000	90.000	89.927
c3540	149	94.606	94.140	94.315	82.484
c5315	122	94.413	94.357	94.002	67.984
c6288	28	98.012	98.567	97.999	96.841
c7552	208	93.311	92.965	92.846	50.027

Table 5.13 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming the missed error probability estimate values (t_n=0.5, 0.6, 0.7, 0.8, 0.9, 2/3 and 1.0 respectively) for n line errors.

Circuit Name	No. of test vector s used to construct the compaction tree	Fault coverage (t_n=0.5)	Fault coverage (t_n=0.6 or 0.7 or 0.8 or 0.9)	Fault coverage (t_n=2/3)	Fault coverage (t_n=1.0)
c17	5	100.000	100.000	100.000	100.000
c432	49	68.251	69.582	73.004	69.392
c499	54	95.669	95.789	92.763	40.959
c880	50	81.809	87.131	87.447	76.596
c1355	85	85.089	85.660	86.675	34.864
c1908	118	78.522	77.725	77.193	82.616
c2670	108	80.000	80.255	79.636	78.945
c3540	149	75.743	71.778	73.703	26.714
c5315	122	87.201	87.033	86.622	27.164
c6288	28	97.638	97.264	97.586	66.326
c7552	208	85.060	75.609	80.578	40.436

Table 5.14 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming the missed error probability estimate values (t_n=0.5, 0.6, 0.7, 0.8, 0.9, 2/3 and 1.0 respectively) for n line errors.

Circuit Name	No. of test vector s used to construct the compaction tree	Fault coverage % (t_n=0.5)	Fault coverage % (t_n=0.6 or 0.7 or 0.8 or 0.9)	Fault coverage % (t_n=2/3)	Fault coverage % (t_n=1.0)
c17	4	100.000	100.000	100.000	100.000
c432	44	96.053	96.053	96.030	96.053
c499	63	71.098	71.098	71.098	97.567
c880	30	97.723	80.062	76.294	80.062
c1355	96	89.370	89.370	89.370	91.892
c1908	137	89.646	89.646	89.677	96.989
c2670	68	94.850	94.850	93.686	94.850
c3540	110	94.693	94.693	95.531	92.737
c5315	55	93.960	93.960	96.480	95.811
c6288	16	99.408	99.125	99.563	99.100
c7552	85	88.772	88.772	94.767	88.608

Table 5.15 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming the missed error probability estimate values (t_n=0.5, 0.6, 0.7, 0.8, 0.9, 2/3 and 1.0 respectively) for n line errors.

In Figure 5.6, we compare the fault coverage results for c432 benchmark circuit assuming different missed error probability estimate values for N sequences (t_n) under heuristic approach.

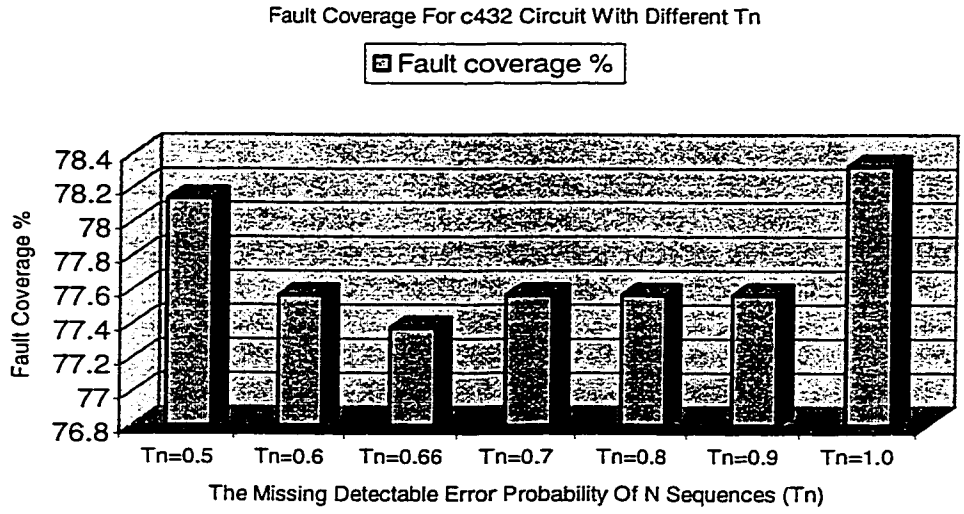


Figure 5.6 The fault coverage for the c432 benchmark circuit at various values of the missed error probability estimate for N sequences (t_n) using heuristic approach.

From the above simulation results, we can find that the changes in fault coverage results take place while t_N changes from 0.5 to 0.9, and it demonstrates that our assumption is reasonable and in accordance with the practical situations. We also find that the fault coverage results become quite low while using FSIM, when t_N equals 1.0, some being under 50%.

5.3 Hardware Overhead

To estimate the hardware overhead, we use the ratio of the weighted gate count metric, i.e. average fanins × number of gates, of the compressor and that of the total circuit comprised of the CUT and the space compactor.

5.3.1 Hardware overhead under stochastic independence case

Tables 5.16 and Table 5.17 show the hardware overhead estimates for all of the ISCAS 85 benchmark circuits corresponding to ATALANTA/FSIM and COMPACTEST, respectively, assuming stochastic independence of multiple line errors.

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	9	3	3.00	160	2.10	2.61
c499	32	1	32.00	202	2.02	7.27
c880	30	5	6.00	383	1.90	3.96
c1355	32	1	32.00	546	1.95	2.92
c1908	25	1	25.00	880	1.70	1.64
c2670	148	9	16.44	1269	1.64	6.64
c3540	25	4	6.25	1669	1.76	0.84
c5315	131	9	14.56	2307	1.90	2.90
c6288	34	3	11.33	2416	1.99	0.70
c7552	112	5	22.40	3515	1.75	1.79

Table 5.16 Estimates of the hardware overhead for ATALANTA/ FSIM assuming stochastic independence of multiple line errors.

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	9	3	3.00	160	2.10	2.61
c499	37	6	6.17	202	2.02	8.31
c880	30	5	6.00	383	1.90	3.96
c1355	45	14	3.21	546	1.95	4.06
c1908	25	1	25.00	880	1.70	1.64
c2670	149	10	14.90	1269	1.64	6.68
c3540	25	4	6.25	1669	1.76	0.84
c5315	132	10	13.2	2307	1.90	2.92
c6288	39	8	4.75	2416	1.99	0.80
c7552	118	11	10.73	3515	1.75	1.88

Table 5.17 Estimates of the hardware overhead for COMPACTEST assuming stochastic independence of multiple line errors.

It can be seen that the hardware overhead range of the compressors for all the benchmark circuits is from 0.70% to 7.27% in the most cases of ATALANTA/FSIM and is 14.29% for circuit C17.

We can also find that the hardware overhead range in case of COMPACTEST is from 0.80% to 8.31% for most of the benchmark circuits, and is 14.29% for C17.

5.3.2 Hardware overhead under stochastic dependence case ($t_N = 2/3$)

Tables 5.18 and 5.19 show the hardware overhead estimates for all ISCAS 85 benchmark circuits corresponding to ATALANTA/FSIM using the exhaustive approach and heuristic approach, respectively.

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	12	6	2.00	160	2.10	3.45
c499	62	31	2.00	202	2.02	13.19
c880	50	25	2.00	383	1.90	6.43
c1355	62	31	2.00	546	1.95	5.50
c1908	48	24	2.00	880	1.70	3.11
c2670	224	85	2.64	1269	1.64	9.72
c3540	42	21	2.00	1669	1.76	1.41
c5315	205	83	2.47	2307	1.90	4.47
c6288	62	31	2.00	2416	1.99	1.27
c7552	162	55	2.95	3515	1.75	2.57

Table 5.18 Estimates of the hardware overhead for ATALANTA/FSIM assuming stochastic dependence of line error (exhaustive approach).

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	9	3	3.00	160	2.10	2.61
c499	34	3	11.33	202	2.02	7.69
c880	40	15	2.67	383	1.90	5.21
c1355	34	3	11.33	546	1.95	3.09
c1908	27	3	9.00	880	1.70	1.77
c2670	166	27	6.15	1269	1.64	7.39
c3540	28	7	4.00	1669	1.76	0.94
c5315	168	46	3.65	2307	1.90	3.69
c6288	42	11	3.82	2416	1.99	0.87
c7552	129	22	5.864	3515	1.75	2.05

Table 5.19 Estimates of the hardware overhead for ATALANTA/ FSIM assuming stochastic dependence of line error (heuristic approach).

Tables 5.20 and 5.21 show the hardware overhead estimates for all ISCAS 85 benchmark circuits corresponding to COMPACTEST by exhaustive approach and heuristic approach, respectively

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	12	6	2.00	160	2.10	3.45
c499	62	31	2.00	202	2.02	13.19
c880	50	25	2.00	383	1.90	6.43
c1355	62	31	2.00	546	1.95	5.50
c1908	47	23	2.04	880	1.70	3.05
c2670	239	100	2.39	1269	1.64	10.30
c3540	42	21	2.00	1669	1.76	1.41
c5315	211	89	2.37	2307	1.90	4.59
c6288	32	31	2.00	2416	1.99	1.27
c7552	158	51	3.10	3515	1.75	2.50

Table 5.20 Estimates of the hardware overhead for COMPACTEST assuming stochastic dependence of line error (exhaustive approach).

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	10	4	2.50	160	2.10	2.89
c499	62	31	2.00	202	2.02	13.19
c880	37	12	3.08	383	1.90	4.84
c1355	53	22	2.41	546	1.95	4.74
c1908	30	6	5.00	880	1.70	1.97
c2670	178	39	4.56	1269	1.64	7.88
c3540	30	9	3.33	1669	1.76	1.01
c5315	174	52	3.35	2307	1.90	3.82
c6288	45	14	3.21	2416	1.99	0.93
c7552	139	32	4.34	3515	1.75	2.21

Table 5.21 Estimates of the hardware overhead for COMPACTEST assuming stochastic dependence of line error (heuristic approach).

5.4 Simulation Results by Using Parity Tree as a Space Compactor

For comparison purposes, we also used the parity tree space compactor composed of XOR gates, which propagates all errors appearing on an odd number of inputs and is, therefore, considered ideal for space compaction. The simulation results of the parity tree space compactor using ATALANTA, FSIM and COMPACTEST simulators are given next.

We assume two situations: first, the space compactor consists of 2-input XOR gates and then consists of 10-input XOR gates.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.033	9	0.000	100.000
c432	49	0.033	116	0.300	99.254
c499	54	0.150	66	0.433	96.098
c880	50	0.067	125	0.533	99.294
c1355	85	0.233	102	1.017	98.044
c1908	118	0.267	233	1.550	98.910
c2670	108	0.633	117	30.867	69.124
c3540	149	0.183	288	7.750	95.101
c5315	122	0.450	259	5.867	98.659
c6288	28	0.067	85	12.150	99.526
c7552	208	0.617	338	64.933	94.977

Table 5.22 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA with parity tree as a space compactor (2 inputs of gate XOR).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.033	10	0.017	100.000
c432	49	0.050	85	0.250	97.529
c499	54	0.133	99	0.200	99.740
c880	50	0.100	180	0.583	97.158
c1355	85	0.100	171	0.767	96.465
c1908	118	0.233	406	4.067	86.222
c2670	108	1.017	210	22.650	85.976
c3540	149	0.233	366	8.050	94.616
c5315	122	0.967	421	13.433	96.934
c6288	28	0.100	102	16.450	99.445
c7552	208	1.417	480	54.017	95.539

Table 5.23 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA with parity tree as a space compactor (10 inputs of gate XOR).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.033	32	0.017	100.000
c432	49	0.033	224	0.050	94.403
c499	54	0.150	224	0.067	94.512
c880	50	0.067	224	0.083	93.952
c1355	85	0.233	224	0.117	89.731
c1908	118	0.267	224	0.217	80.228
c2670	108	0.633	224	0.283	67.603
c3540	149	0.183	224	0.567	85.764
c5315	122	0.450	224	0.517	95.138
c6288	28	0.067	224	1.750	99.449
c7552	208	0.617	224	0.967	87.816

Table 5.24 Simulation results of the ISCAS 85 benchmark circuits using FSIM with parity tree as a space compactor (2 inputs of gate XOR).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	5	0.033	32	0.017	100.000
c432	49	0.050	224	0.067	94.297
c499	54	0.133	224	0.067	91.927
c880	50	0.100	224	0.083	79.158
c1355	85	0.100	224	0.100	88.258
c1908	118	0.233	224	0.217	74.775
c2670	108	1.017	224	0.267	73.894
c3540	149	0.233	224	0.517	81.985
c5315	122	0.967	224	0.450	89.428
c6288	28	0.100	224	1.700	99.329
c7552	208	1.417	224	0.783	90.589

Table 5.25 Simulation results of the ISCAS 85 benchmark circuits using FSIM with parity tree as a space compactor (10 inputs of gate XOR).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.017	6	0.000	100.000
c432	44	0.033	67	5.690	99.257
c499	63	0.183	62	34.290	96.098
c880	30	0.117	49	5.890	99.297
c1355	96	0.283	96	42.230	98.044
c1908	137	0.300	168	81.209	98.810
c2670	68	0.817	809	2399.969	61.820
c3540	110	0.183	144	445.769	94.847
c5315	55	0.617	80	103.209	98.678
c6288	16	0.083	34	94.760	99.526
c7552	85	0.733	369	1516.990	95.224

Table 5.26 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST with parity tree as a space compactor (2 inputs of gate XOR).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No of applied test vectors	CPU simulation times (secs)	Fault coverage (%)
c17	4	0.033	6	0.020	100.000
c432	44	0.033	26	29.650	89.544
c499	63	0.150	24	69.590	86.589
c880	30	0.067	12	170.900	73.053
c1355	96	0.233	31	266.230	85.038
c1908	137	0.267	74	655.550	82.936
c2670	68	0.633	291	1039.799	83.783
c3540	110	0.183	104	1047.050	88.417
c5315	55	0.450	352	1467.510	95.305
c6288	16	0.067	135	585.090	99.032
c7552	85	0.617	396	5851.779	89.106

Table 5.27 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST with parity tree as a space compactor (10 inputs of gate XOR).

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	12	6	2.00	160	2.10	3.44
c499	62	31	2.00	202	2.02	13.19
c880	50	25	2.00	383	1.90	6.42
c1355	62	31	2.00	546	1.95	5.50
c1908	48	24	2.00	880	1.70	3.10
c2670	278	139	2.00	1269	1.64	11.78
c3540	42	21	2.00	1669	1.76	1.40
c5315	244	122	2.00	2307	1.90	5.27
c6288	62	31	2.00	2416	1.99	1.27
c7552	214	107	2.00	3515	1.75	3.36

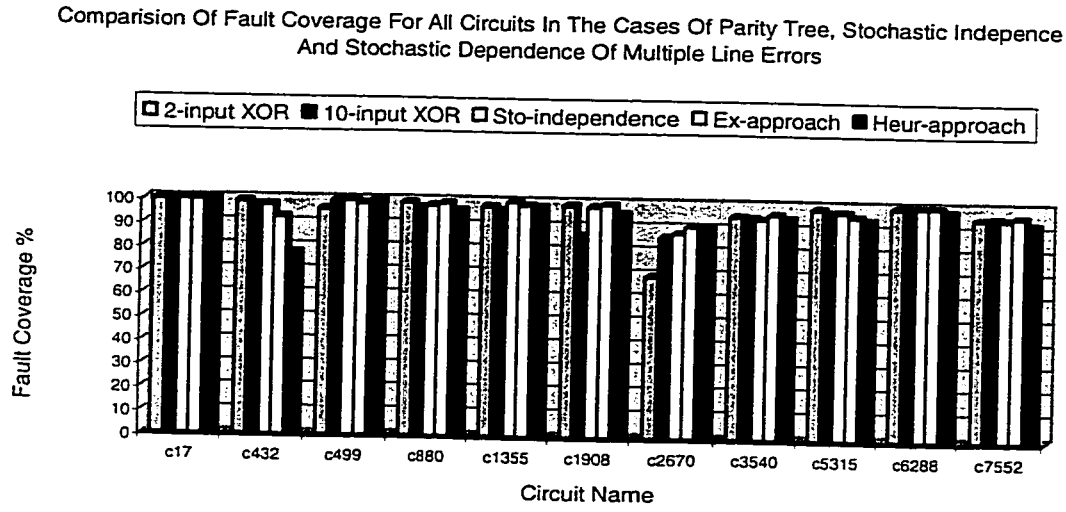
Table 5.28 Estimates of the hardware overhead with parity tree as a space compactor (2 inputs of gate XOR).

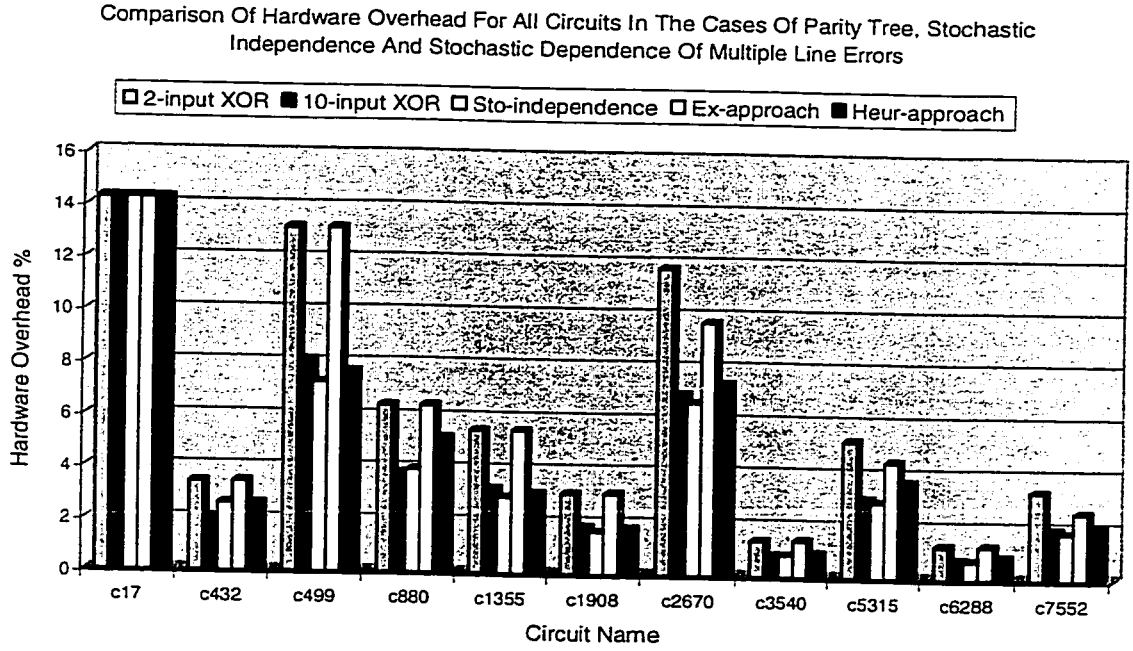
Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fainin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.29
c432	7	1	7.00	160	2.10	2.04
c499	36	5	7.20	202	2.02	8.11
c880	29	4	7.25	383	1.90	3.83
c1355	36	5	7.20	546	1.95	3.27
c1908	28	4	7.00	880	1.70	1.84
c2670	156	17	9.17	1269	1.64	6.97
c3540	25	4	6.25	1669	1.76	0.84
c5315	138	16	8.62	2307	1.90	3.05
c6288	36	5	7.20	2416	1.99	0.742
c7552	121	14	8.64	3515	1.75	1.93

Table 5.29 Estimates of the hardware overhead with parity tree as a space compactor (10 inputs of gate XOR).

To have an overall view, we compared the ATALANTA simulation results for ISCAS 85 benchmark circuits in all cases, including parity tree (using 2-input XOR gate

and 10-input XOR gate), stochastic independence and stochastic dependence of multiple line errors (exhaustive approach and heuristic approach).





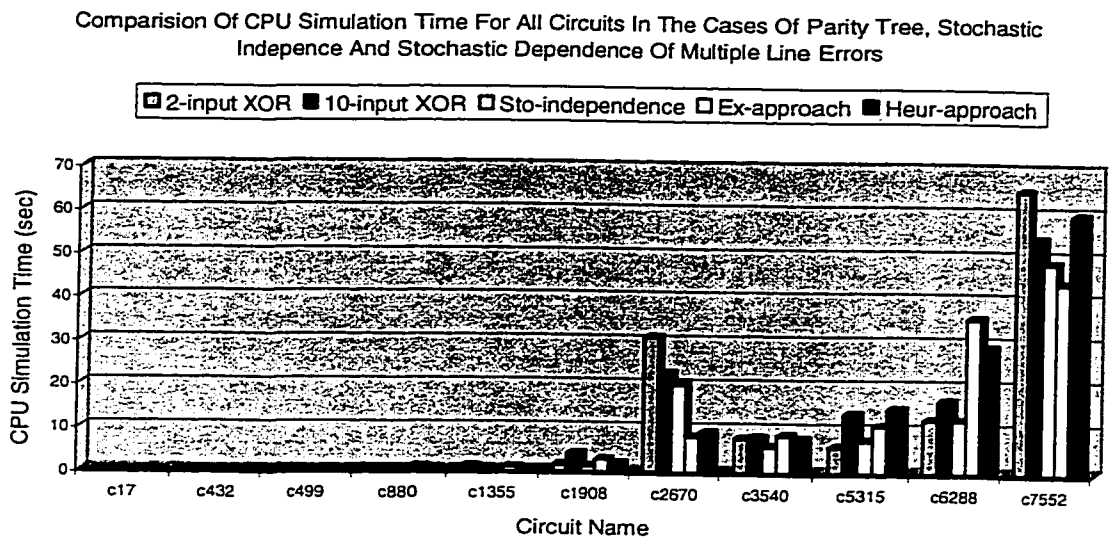


Figure 5.7 Comparison of the fault coverage, hardware overhead and CPU simulation time for ISCAS 85 benchmark circuits in the cases of parity tree (2-input XOR gate and 10-input XOR gate), stochastic independence of multiple line errors and stochastic dependence of multiple line errors (including exhaustive approach and heuristic approach) using ATALANTA.

From the above comparison results, we found that the fault coverage results as obtained assuming stochastic independence and stochastic dependence of line errors are almost the same as the results in parity tree case. The difference is rather small and acceptable. The XOR gate is conceptually neat and simple, but practically it has to be implemented by using other gates, so the hardware overhead of parity tree (both using 2-input XOR and 10-input XOR) is much higher.

5.5 Compaction Circuits for c432

We have shown the space compaction circuits for c432 benchmark circuit to give an idea about how our space compactor looks like, corresponding to stochastic independence and dependence of line errors (according to the formula in Chapter 4). There are 36 primary inputs, 160 gates and 7 primary outputs in c432 benchmark circuit.

Figure 5.8 illustrates the space compactor assuming stochastic independence of multiple line errors, using the ATALANTA /FSIM.

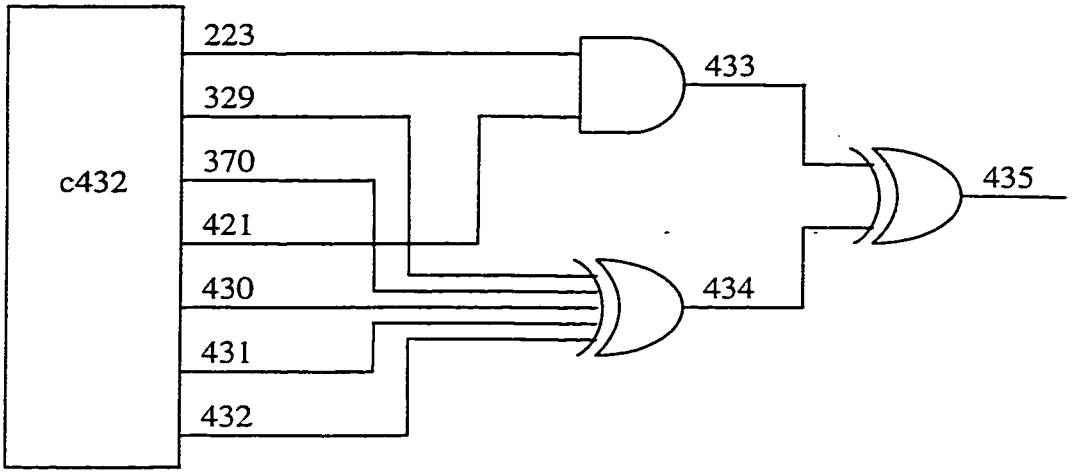


Figure 5.8 Space compactor circuit for c432 assuming stochastic independence of multiple line errors.

In Figure 5.9, we show the space compactor assuming stochastic dependence of multiple line errors while using the exhaustive approach.

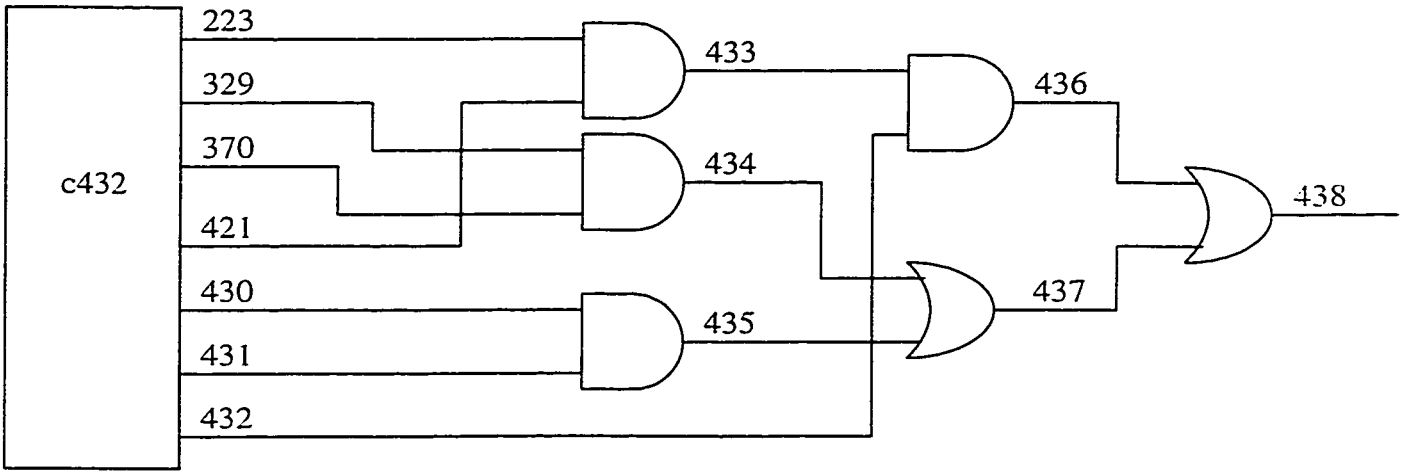


Figure 5.9 Space compactor circuit for c432 assuming stochastic dependence of multiple line errors (Exhaustive Approach).

Figure 5.10 shows the space compactor assuming stochastic dependence of multiple line errors (when t_n =2/3) using the heuristic approach.

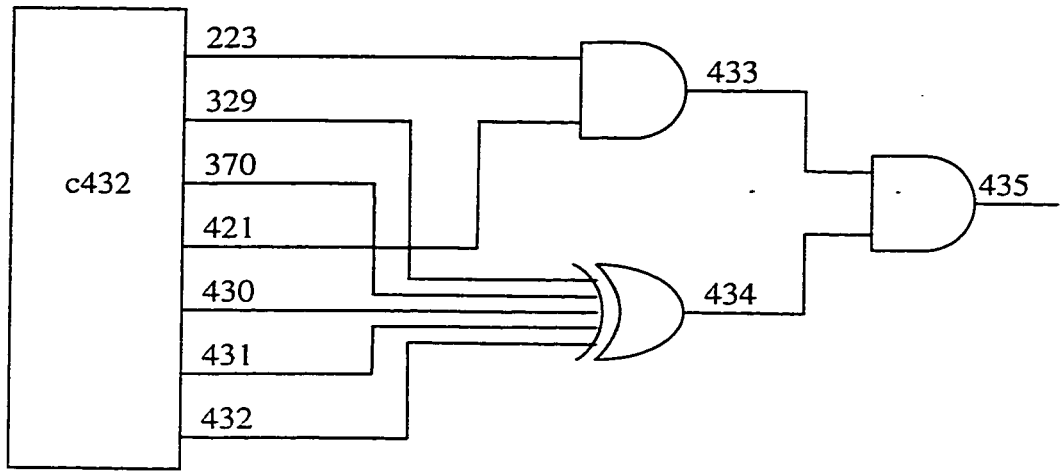


Figure 5.10 Space compactor circuit for c432 assuming stochastic dependence of multiple line errors (Heuristic Approach, t_n =2/3).

Figure 5.11 illustrates the parity tree as a space compactor consisted of 2-input XOR gates.

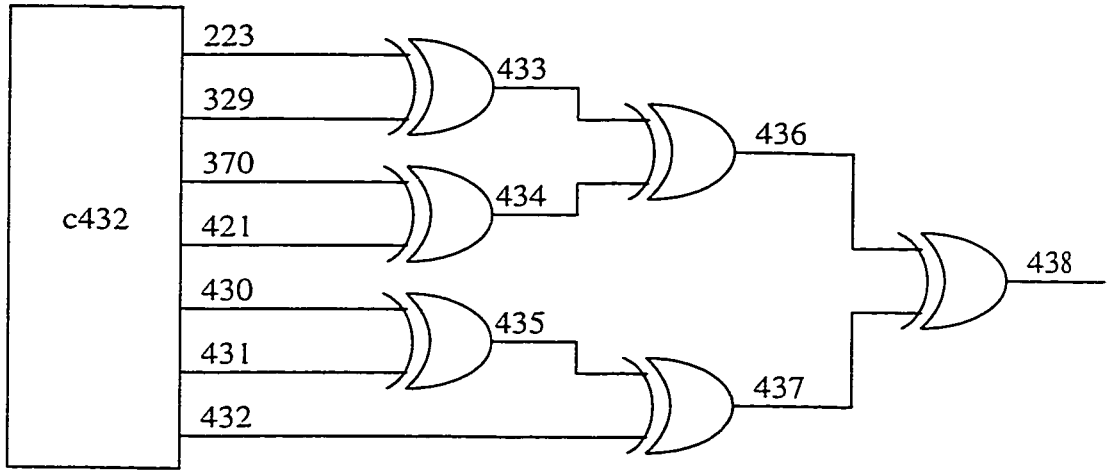


Figure 5.11 Parity tree as a space compactor for c432 (using 2-input XOR gate).

5.6 Summary

In this chapter we provided simulations on ISCAS 85 benchmark circuits using ATALANTA, FSIM, and COMPACTEST simulators under stochastic independence and stochastic dependence of multiple line errors.

From the simulation experiments, it is obvious that our space compactors, in all cases, are comparable with the parity tree space compactor. For most of the benchmark circuits, we obtained better fault coverage with reduced CPU simulation time and hardware overhead using our space compactor than what we have when using a parity tree space compactor.

Chapter 6

Conclusions

This thesis considers the general problem of designing and analyzing efficient space compression techniques for built-in self-testing of VLSI circuits using compact test sets. The techniques are based on identifying certain inherent properties of the test output responses of the CUT and the knowledge of nonoccurrence of failure probabilities. The generalized mergeability criteria of output sequences are developed utilizing the concepts of Hamming distance, sequence weights, cover table, and frequency ordering of literals for an arbitrary number of outputs and the effect of failure probabilities on the mergeability criteria is analyzed as well. The techniques proposed achieve rather high fault coverage for single stuck-line faults, with low CPU simulation time, and acceptable hardware overhead, as evident from extensive simulation results on ISCAS 85 combinational benchmark circuits, under conditions of both stochastic independence and dependence of single and multiple line errors. No simulation did on ISCAS 89 circuits.

No attempt has been made in the present thesis to design aliasing-free space compactors. Obviously, the best design should strive to achieve the objective that the compression network should not involve any information loss. This is though somewhat tricky, attempts have been made by other investigators with some success. However, in the present case, since we are using only compact sets of tests (which in general are not guaranteed to be complete test sets for all single detectable faults of the given circuits), it *not* expected that a 100% fault coverage could be obtained. However, research in this direction is surely worthwhile and desirable.

BIBLIOGRAPHY

[1] M. Abramovici, J. J. Kulikowski, P. R. menon and D. T. Miller, "SMART and FAST: test generation for VLSI scan-design circuits", *IEEE Design and Test of Computers*, Vol. 3, pp. 43-54, August 1986.

[2] V.K. Agarwal, "Increasing effectiveness of built-in testing by output data modification", *Proc. FTCS-13*, June 1983, pp. 227-234.

[3] V. D. Agrawal, C. R. Kime and K. K. Saluja, "A tutorial on Built-In Self-Test (Part 1)", *IEEE Design and Test of Computers*, pp. 73-82, March 1993.

[4] V. D. Agarwal, et al., "Built-in self-test for digital circuits", *AT&T Technical Journal*, Vol. 73, pp. 30-39, March/April 1994.

[5] M. H. Assaf, *Space Compactor Design for BIST of VLSI Circuits from Compact Test Sets Using Sequence Characterization and Failure Probabilities*, M.S. Thesis, Department of Electrical Engineering, University of Ottawa, Ottawa, August 1996.

[6] T. Barakat, *Generalized Mergeability in Space Compression Using Nonexhaustive Test Patterns for Built-In Self-Testing of VLSI Circuits: Mathematical Analysis and Simulation Results*, M.S. Thesis, Department of Electrical Engineering, University of Ottawa, Ottawa, December 1997.

[7] P. H. Bardell, W. H. McAnney, and J. Savir, *Built in test for VLSI: Pseudorandom techniques*, New York: Wiley-Interscience, 1987.

[8] F. Brglez and H. Fujiwara, " A neutral netlist of 10 combinational benchmark circuits and a target simulation in Fortran", *Proc. 1985 Int. Symp. on Circuits and Systems*. pp. 695-698. 1985.

[9] R. L. Campbell and A. A. Tarbox, "Testing goes critical path", *AT&T Technical Journal*, vol. 73, pp. 4-9, March/April 1994.

[10] K. Chakrabarty, *Test Response Compaction for Built-In Self-Testing*. Ph. D. Dissertation, University of Michigan, MI, 1995.

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

[12] K. Cheng and V. D. Agrawal, *Unified methods for VLSI simulation and test generation*, Kluwer Academic Publishers, MA, 1989.

[13] A. K. Choudhury and S. R. Das, "Some studies on connected cover term matrices of switching functions", *International Journal of Control* Vol.2, pp. 441-501, 1965.

[14] 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.

[15] S. R. Das, *Boolean Difference Methods*, Notes for graduate course ELG 5195, University of Ottawa.

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

[17] S. R. Das, H. T. Ho, W. B. Jone and A. R. Nayak, "An improved output compaction modification technique for built-in self-test in VLSI circuits", *Proc. 1994 Int Conf. VLSI Design*, pp. 403-407, 1994.

[18] S. R. Das and N. S. Khabra, "Clause-column table approach for generating all the prime implicants of switching function", *IEEE Transactions on Computer*, Vol. C-21, No.11, pp. 1239-1246, November 1972.

[19] S. DasGupta, E. B. Eichelberger and T. W. Williams, "LSI chip design for testability", *Digest of Technical Papers, IEEE International Solid State Circuits Conference*, San Francisco, pp.216 –217, February 1978.

[20] S.Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill Series on Computer Engineering, NY, 1994.

[21] K. I. Diamantaras and N. K. Jha, "A new transition count method for testing of logic", *IEEE Transaction on Computers*, Vol. C-10, pp. 407-410, March 1991.

[22] E. B. Eichelberger and T. W. Williams, "A logic design structure for LSI testability", *Proc. 1977 Design Automation Conference*, pp. 462 – 468, 1977.

[23] E. B. Eichelberger and T. W. Williams, "A logic design structure for LSI testing", *Journal of Design Automation and Fault Tolerant Computing*, Vol. 2, no. 2, pp. 165 – 178 May 1978.

[24] R. A. Frowerk, "Signature analysis – a new digital field service method", *Hewlett Packard Journal*, pp. 2 – 8, May 1977.

[25] H. Fujiwara and A. Yamamoto, "Parity-scan design to reduce the cost of test application", *IEEE Transactions on Computer-Aided Design*, Vol. 12, pp. 1604-1611, October 1993.

[26] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithm", *IEEE Transactions on Computers*, Vol. C-32; pp. 1137-1144, December 1983.

[27] P. Gelsinger, "Design and Test of the 80386", *IEEE Design & Test of Computers*, Vol. 4, No. 3, pp. 42-45, June 1987.

[28] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits", *IEEE Transactions on Computers*, Vol. C-30, pp. 215-222, March 1981.

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

[30] J. P. Hayes, "Check sum methods for test data compression ", *Journal of Design Automation and Fault-Tolerant Computing*, Vol. 1, pp. 3-7, January 1976.

[31] J. P. Hays, "Transition count testing of combinational logic circuits", *IEEE Transaction on Computers*, Vol. C-25, pp. 613-620, June 1976.

[32] T. C. Hsiao and S. C. Seth, "An analysis of the use of Rademacher-Walsh spectrum in compact testing", *IEEE Transactions on Computer*, Vol. 33, pp. 934-937, October 1984.

[33] S. L. Hurst, *Custom VLSI Microelectronics*, Prentice-Hall International, UK, 1992.

[34] W. B. Jone and S. R. Das, "Space compression method for built-in self-testing of VLSI circuits". *Int. Journal of Computer-Aided Design*, Vol. 3, pp. 309-322, September 1991.

[35] M. Karpovsky and P. Nagvajara, "Optimal time and space compression of test responses for VLSI devices", *Proc. 1987 Int. Test Conference*, pp. 523-529. 1987.

[36] H. K. Lee and D. S. Ha, "On the generation of test patterns for combinational circuits", *Technical Report No. 12-93*, Dept. of Electrical Eng., Virginia Polytechnic Institute and State University.

[37] H. K. Lee and D. S. Ha, "An efficient forward fault simulation algorithm based on the parallel pattern single fault propagation", *Proc. 1991 Int. Conference*, pp. 946-955, 1991.

[38] Y. K. Li and J. P. Robinson, "Space compression methods with output data modification", *IEEE Transactions on Computer-Aided Design*, Vol. 6, pp. 290-294, March 1987.

[39] E. J. McCluskey, "Built-In Self-Test Techniques", *IEEE Design and Test of Computers*. Vol. 2, pp. 29-36, April 1985.

[40] E. J. McCluskey, *Logic Design Principles, with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, New Jersey, 1986.

[41] E. J. McCluskey, "Verification testing a pseudoexhaustive test technique", *IEEE Transactions on Computers,* Vol. C-33 , pp. 541-546, June 1984.

[42] H. J. Nadig, "Signature analysis – concepts, examples and guidelines", *Hewlett Packard Journal*, pp. 15 – 21, May 1977.

[43] I. Pomeranz, L. N. Reddy and S. M. Reddy, "COMPACTEST: a method to generate compact test sets for combinational circuits", *Proc. 1991 Int. Test Conference*, pp. 194-203, 1991.

[44] D. K. Pradhan and S. K. Gupta, "A new framework for designing and analyzing BIST techniques and zero aliasing compression", *IEEE Transaction on Computers*, Vol. C-10, pp. 407-410, March 1991.

[45] I. M. Ratiu and H. B. Bakoglu, "Pseudorandom built-in self-test methodology and implementation for the IBM RISC System/6000 processor", *IBM Journal of Research and Development*, Vol. 34, pp. 78-87, January 1990.

[46] 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 1988.

[47] J. P. Roth, "Diagnosis of automata failures: a calculus and a method", *IBM Journal of Research and Development*, Vol. 10, pp. 278-291, July 1966.

[48] G. Russell and I. L. Sayers, *Advanced Simulation and Test Methodologies for VLSI Design*, Van Nostrand Reinhold (International), 1989.

[49] K. K, Saluja and M.G. Karpovsky, "Testing computer hardware through data compression in space and time", *Digest of International Test Conf.*, pp.83-88, 1983.

[50] J. Savir, "Syndrome-testable design of combinational circuits", *IEEE Transaction on Computers*, Vol. C-29, pp. 442-451, June 1980.

[51] J. Savir and W. H. McAnney, "On the masking probability with one's count and transition count", *Proc. Int. Conference on Computer-Aided Design*, pp. 111-113, 1985.

[52] N. R. Saxena and J. P. Robinson, "Syndrome and transition count are uncorrelated", *IEEE Transactions on Information Theory*, Vol. 34, pp. 64-69, January 1988.

[53] N. R. Saxena and J. P. Robinson, "A unified view of test response compression methods", *IEEE Transactions on Computers*, Vol. C-36, pp. 94-99, January 1987.

[54] F. F. Sellers, M. Y. Hsiao and L. W. Bearnson, "Analyzing errors with Boolean Difference", *IEEE Transactions on Computers*, Vol. C-17, pp. 676-683, July 1968.

[55] J. E. Stephenson and J. Grason, "A testability measure for register transfer level digital circuits", *Dig. 6th Int. Symp. Fault-Tolerant Comput.* (FTCS-6), Pittsburgh, pp. 101-107, June 21-23, 1976.

[56] A. K. Susskind, "Testing by verifying Walsh coefficients", *Proc. 1981 Int. Symp. Fault-Tolerant Computing*, pp. 206-208, 1981.

[57] Neil H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, 1994.

[58] T. W. Williams, "Design for testability", in *Computer Design Aids for VLSI Circuits* (P. Antogneti, D. O. Pederson, and H de Man, eds.), NATO AIS Series, The Netherlands: Martinus Nijhoff Publishers, pp. 359-416, 1986.

[59] T. W. Williams, W. Daehn, M. Gruetzner and C. W. Starke, "Aliasing errors in signature analysis registers", *IEEE Design and Test of Computers*, Vol. 4, pp. 39-45, April 1987.

[60] Y. Zorian and V.K. Agarwal, "Higher certainty of error coverage by output data modification", *Proc. Int. Test Conference*, pp. 140-147, 1984.

[61] Y. Zorian and Ivanov, "Programmable space compaction for BIST", *Proc. 1993 Int. Symp. on Fault-Tolerant Computing*, pp. 340-349, 1993.

LIST OF PAPERS BY THE CANDIDATE

1. Jingyi Liang, "Increasing fault coverage using a new probability measure in output compaction based on switching theory formulation", Presented at *the IASTED International Conference on Modeling, Identification and Control*, Innsbruck, Austria, February 14-17, 2000.
   (With Sunil R. Das, Made Sudarma, Emil M. Petriu, Wen B. Jone and Krishnendu Chakrabarty)

2. Jingyi Liang, "Data compression in space under generalized mergeability based on concepts of cover table and frequency ordering", *IEEE Instrumentation and Measurement Technology Conference*, Baltimore, MD, U.S.A., May 1-4, 2000.
   (With Sunil R. Das, Emil M. Petriu, Wen B. Jone and Krishnendu Chakrabarty)

3. Jingyi Liang, "Parity bit signature in response data compaction and built-in self-testing of VLSI circuits with compact test sets", *43rd Midwest Symposium on Circuits and Systems*, East Lansing, MI, U.S.A., August 8-11, 2000 (accepted).
   (With S.R. Das, E.M. Petriu, Made Sudarma, M.H. Assaf, and W.B. Jone)

4. Jingyi Liang, "Multiple-output parity bit signature", *IASTED International Conference on Applied Simulation and Modeling*, Banff, Alberta, July 24-26, 2000 (accepted).
   (With S.R. Das, E.M. Petriu, Made Sudarma, and W.B. Jone)

5. Jingyi Liang, "Output compaction in time for nonexhaustive testing", *International Forum cum Conference on Information Technology and Communication at the Dawn of the Millennium*, Bangkok, Thailand, August 1-4, 2000.
   (With S.R. Das, E.M. Petriu, Made Sudarma, and W.B. Jone)

ACRONYMS

ATE: automatic test equipment.

ATG: automatic test generation.

BIST: built-in self-testing.

BIT: built-in test.

CONTEST: concurrent test.

CRC: cyclic redundancy checking.

CUT: circuit under test.

DFT: design for testability.

DSC: dynamic space compression.

DTC/MTC: double/multiple transition counting.

FAN: fanout-oriented TG.

HSC: hybrid space compactors.

LFSR: linear feedback shift register.

LSSD: level sensitive scan design.

MDSC: modified dynamic space compression.

MISR: multiple-input signature register.

MPT: multiplexed parity tree.

PODEM: path-oriented decision making.

PSC: programmable space compaction.

QFC: quadratic function compaction.

RCU: response compaction unit.

ST: self-test.

TPG: test patter generator.

APPENDIX

C Programs to Construct the Space Compaction Trees for Combinational Logic Circuits

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada  K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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

/*************************** HISTORY ***********************************

        JTesting_I: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        defin.h
        JTesting_I performs output compression for combinational circuits
        based on sequence characterization and stochastic independence
        of line errors. Output sequences can be read from a user supplied
        file.

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


/*****************************************************************
                        Defin.h
            Define Global Variables Used in Jtesting_I
*****************************************************************/

#define MAX_OUT 300                             /*maximum number of primary outputs*/
#define MAX_BITS 250                            /*maximum Length of a sequence*/
#define PTREE_NO 10                             /*input number of xor gate consist of  parity tree*/
int stage;
int new_line;                                   /*line number created after compression*/
int length;                                     /*length of the sequence*/
int discard_number;                             /*it will reduce 1 when one is merged*/
int merge_no;
int merge_result[MAX_OUT][MAX_BITS];            /* put the merge result sequence */
int merge_output_line[MAX_OUT];                 /*keep the merged output line number*/
FILE *in, *out;                                 /*input/output File Pointers*/

float minutes, seconds, inittime, simtime, runtime;
```

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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


/***************************** HISTORY ***************************************

        JTesting_I: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        stoch_indep.c
        JTesting_I performs output compression for combinational circuits
        based on sequence characterization and stochastic independence
        of line errors. Output sequences can be read from a user supplied
        file.

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


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
#include <string.h>
#include "defin.h"
#include<malloc.h>

#include<memory.h>

struct OUTDATA                                  /*structure to store the output data*/
{
 int out_number;                                /*number of primary outputs*/
 int length;                                    /*length of sequence*/
 int out_seq[MAX_OUT][MAX_BITS];                /*Storage for output sequences*/
 int init_values[MAX_OUT];                      /*Initial Line number*/
 int discard[MAX_OUT];
} ;

struct OUTDATA data;

typedef struct
{
 int cover_table[MAX_OUT][MAX_BITS];            /*cover_table*/
 int seq_no;                                    /*rower numbers in  cover table*/
```

```
}COVER_TABLE;

typedef struct
{
 int seq[MAX_BITS];                    /*result of bitwise operation*/
 int line_number[MAX_OUT];             /*output line number in best group*/
 int max_w;                            /*maxnumber(0/1) matching bits in the best group*/
 int max_n;                            /* maxnumber sequences in best group*/
}BESTGROUP;

typedef struct
{
 int posi_number[MAX_OUT];             /*count 1/0 numbers of bit position in 1/0 covertable*/
 int sort_line[MAX_OUT];               /*after sorting, the name of line*/
 int orig_position[MAX_OUT];           /*store the original position in the original input*/
}FREQ_ORDER;                           /*keep some information about frequency ordering*/


/*****************************************************************
                        gettime()
    FUCNTION FOR GETTING CPU TIME USED IN THE PROGRAM
*****************************************************************/

void gettime(float *usertime, float *systemtime, float *total)
{
 struct tms timesbuffer;
 time_t totaltime;
 time_t utime;
 time_t stime;

 printf("FUNCTION [gettime] BEGIN\n");

 times(&timesbuffer);
 utime=timesbuffer.tms_utime;
 stime=timesbuffer.tms_stime;
 totaltime=timesbuffer.tms_utime+timesbuffer.tms_stime; /*In 60th seconds*/
 *usertime=(float)utime/60.0;
 *systemtime=(float)stime/60.0;
 *total=(float)totaltime/60.0;

 printf("inittime is %.3f\n", *usertime);
 printf("simtime is %.3f\n", *systemtime);
 printf("total is %.3f\n", *total);

 printf("FUNCTION [gettime] END\n");

}          /*function gettime() end */


/*****************************************************************
                        copy()
        FUCNTION FOR GETTING TWO SAME SEQUENCES
*****************************************************************/

copy(int *target, int *source)
{
 int m;

 printf("FUNCTION [copy] BEGIN\n");

 for (m=0; m<length; m++)
 target[m]=source[m];

 printf("FUNCTION [copy] END\n");

}/*function copy() end*/
```

```
/*********************************************************************
                              count_number()
     FUCNTION FOR GETTING 1'S/0'S NUMBER IN A SEQUENCE
*********************************************************************/

int count_number(int flag,int *candi_list)
{
 int i, result;
 int count=0;

 printf("FUNCTION [count_number] BEGIN\n");

 if (flag==1)
 {
  for (i=0; i<length; i++)
  { count+=candi_list[i]; }        /*for end*/
  result=count;
 }          /* if end*/

 else if (flag==0)
 {
  for (i=0; i<length; i++)
  {
   if (candi_list[i]==0)
   count++;
   else
   count=count;
  }         /*for end*/
  result=count;
 }/*else if end*/

 printf("FUNCTION [count_number] END\n");

 return(result);

}/*function count_number() end*/



/*********************************************************************
                              intcmp()
                        FUCNTION FOR COMPARING
*********************************************************************/

int intcmp(const void *v1, const void *v2)
{   return(*(int*)v2-*(int*)v1);   }/*function intcmp() end*/



/*********************************************************************
                         frequency_ordering()
          FUCNTION FOR GETTING FREQUENCY ORDERING
              FROM THE ONE/ZERO COVER TABLE
*********************************************************************/

FREQ_ORDER frequency_ordering(int flag, COVER_TABLE ct)
{
 int i, j,m;
 int h=0;
 int temper[MAX_OUT];
 int line[MAX_OUT];
 int position[MAX_OUT];                    /*store the original position in the original  input*/
 FREQ_ORDER fo;

 printf("FUNCTION [frequency_ordering] BEGIN\n");

 for(i=0; i<MAX_OUT; i++)                  /*initial the value*/
 {
  fo.posi_number[i]=0;
```

```
    fo.sort_line[i]=0;
    fo.orig_position[i]=0;
    temper[i]=0;
    line[i]=0;
    position[i]=0;
   }

  /******** count the bit position ********/

  for(m=0;m<data.out_number; m++)
  {
   for(j=0; j<length; j++)
   {
    for(i=0; i<ct.seq_no; i++)
    {
     if(ct.cover_table[i][j]==data.init_values[m])
     { fo.posi_number[h]++; }
    }/*for end*/
   }/*for end*/
   if(fo.posi_number[h]!=0)
   {
    line[h]=data.init_values[m];
    position[h]=m;
    h++;
   }
  }          /*for end*/
  for (i=0; i<data.out_number; i++)
  { temper[i]=fo.posi_number[i]; }

 /******** sort-frequency ordering ********/

 qsort(fo.posi_number, data.out_number, sizeof(fo.posi_number[0]), intcmp);

 /******** get the relative original line number ********/
 for (i=0; i<data.out_number; i++)
 {
  for(j=0; j<data.out_number; j++)
  {
   if((fo.posi_number[i]==temper[j])&&(temper[j]!=999))
   {
    fo.sort_line[i]=line[j];
    fo.orig_position[i]=position[j];
    temper[j]=999;
    break;
   }/*if end*/
  }/*foe end*/
 }/*for end*/

 printf("FUNCTION [frequency_ordering] END\n");
  return(fo);

}/*function frequency_ordering() end*/



/*****************************************************************
                          final_merge()
  FUCNTION FOR GROUPING REMAINING SEQUENCS BY GATE XOR
*****************************************************************/

int final_merge()
{
 int i, j;
 int seq_result[MAX_BITS];                    /* the result of merging sequences */
 int line_merge[MAX_OUT];                     /* keep the left line to merge*/
 int p=0;

 printf("FUNCTION [final_merge] BEGIN\n");

 for (i=0; i<MAX_BITS; i++)
```

```
  seq_result[i]=0;

  for (i=0; i<MAX_OUT; i++)
  line_merge[i]=0;
  for (i=0; i<data.out_number; i++)
  {
   if(data.discard[i]!=1)
   {
    line_merge[p]=data.init_values[i];
    p++;
    data.discard[i]=1;
    for(j=0; j<length; j++)
    {
      seq_result[j]=seq_result[j]^data.out_seq[i][j];
      data.out_seq[i][j]=9;
    }/*for 2 end*/
    discard_number--;
   }/*if end*/
  }/*for 1 end*/

  new_line++;
  merge_output_line[merge_no]=new_line;

  for (j=0; j<length; j++)
  { merge_result[merge_no][j]=seq_result[j]; }
  in=fopen("merge_result.dat", "a");
  if(in)
  {
   printf(" STAGE %d\n", stage);
   printf("*****************************************\n\n");
   fprintf(in,"%d = XOR(", new_line);
   for (i=0; i<p; i++)
   {
    printf("#%d  ", line_merge[i]);
    fprintf(in, "%d, ", line_merge[i]);
   }
   fprintf(in,")\n");
   printf("\nare merged with an XOR gate into #%d.\n", new_line);
  }                    /*if(in) end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);
   merge_no++;
  stage++;

  printf("FUNCTION [final_merge] END\n");

  return(0);

}/*function final_merge() end*/



/*****************************************************************
                          merge_seq()
          FUCNTION FOR MERGING SELECTED SEQUENCES
                     BY AND/OR LOGIC GATES
*****************************************************************/

merge_seq(int select, BESTGROUP maxgroup)
{
  int i, j, m, W, R;
  int counter, result;

  printf("FUNCTION [merge_seq] BEGIN\n");

  switch (select)
  {
   case 0:                                          /*merge by gate OR*/
    {
```

```
  if( (maxgroup.max_n>=2)&&(maxgroup.max_w>=length/2))/*if 1 begin*/
  {
   for(i=0; i<maxgroup.max_n; i++)
   {
    for (j=0; j<data.out_number;j++)
    {
     if (maxgroup.line_number[i]==data.init_values[j])
     {
      data.discard[j]=1;
      for(m=0; m<length; m++)
      { data.out_seq[j][m]=9; }
     }/*if end*/
    }/*for 2 end*/
    discard_number--;
   }      /*for 1 end*/
   new_line++;

   merge_output_line[merge_no]=new_line;

   for (j=0; j<length; j++)
   { merge_result[merge_no][j]=maxgroup.seq[j]; }

   in=fopen("merge_result.dat", "a");
   if(in)
   {
    printf(" STAGE %d\n", stage);
    printf("*****************************************\n\n");
    fprintf(in, "%d = OR(", new_line);

    for (i=0; i<maxgroup.max_n; i++)
    {
     printf("#%d  ", maxgroup.line_number[i]);
     fprintf(in, "%d, ", maxgroup.line_number[i]);
    }
    fprintf(in, ")\n");

    printf("\nare merged with an OR gate into #%d.\n", new_line);

   }/*if(in) end*/

   else
   printf("Error opening result_compress.dat file\n");
   fclose(in);

   merge_no++;
   stage++;
   result=1;
   break;

  }/* if 1 end*/
 }/*case 0 end*/

case 1:                                                    /*merge by gate AND*/
{
 if( (maxgroup.max_n>=2)&&(maxgroup.max_w>=length/2))  /*if 1 begin*/
 {
  for(i=0; i<maxgroup.max_n; i++)
  {
   for (j=0; j<data.out_number;j++)
   {
    if (maxgroup.line_number[i]==data.init_values[j])
    {
     data.discard[j]=1;
     for(m=0; m<length; m++)
     { data.out_seq[j][m]=9; }
     }/*if end*/
   }/*for 2 end*/
   discard_number--;
  }/*for 1 end*/
  new_line++;
```

```
      merge_output_line[merge_no]=new_line;
       for (j=0; j<length; j++)
      {merge_result[merge_no][j]=maxgroup.seq[j];}
       in=fopen("merge_result.dat", "a");
      if(in)
      {
       printf(" STAGE %d\n", stage);
       printf("*****************************************\n\n");
       fprintf(in, "%d = AND(", new_line);

       for (i=0; i<maxgroup.max_n; i++)
       {
        printf("#%d  ", maxgroup.line_number[i]);
        fprintf(in, "%d, ", maxgroup.line_number[i]);
       }
       fprintf(in, ")\n");
       printf("\nare merged with an AND gate into #%d.\n", new_line);
      }/*if(in) end*/
      else
      printf("Error opening result_compress.dat file\n");
      fclose(in);

      merge_no++;
      stage++;
      result=1;
      break;
     }/* if 1 end*/
    }/*case 1 end*/
   }/*switch end*/

   printf("\n********  get new merger line # and result  ********\n");
   for(i=0; i<merge_no; i++)
   {
    printf("%d\n", merge_output_line[i]);
    for (j=0; j<length; j++)
    { printf("%d", merge_result[i][j]); }          /*for end*/
    printf("\n");
   }/*for end*/
   printf("\n\n");

   printf("FUNCTION [merge_seq] END\n");

}/*function merge_seq() end*/



/*****************************************************************
                          get_maxgroup()
    FUCNTION FOR GETTING MAXGROUP FROM THE COVER TABLE
*****************************************************************/

BESTGROUP get_maxgroup(int flag, COVER_TABLE ct)
{
 int k, i, j, jj, m, u, v, p1, W1, N1,N;
 int h=1;
 int t;
 int cs=0;
 int temper[MAX_BITS], operater[MAX_BITS];
 int candidate_group[MAX_OUT];                    /*the candidate sequences line  name for selecting can_seq*/
 int candidate_posi[MAX_OUT];
 int candidate_number=0;                          /*number of sequences in the candidate group*/
 int candidate_orig_position[MAX_OUT];
 int can_orig_position[MAX_OUT];
 int can_seq[MAX_OUT];                            /*the sequences inside to select max_group*/
 BESTGROUP  goodgroup[MAX_OUT], bettergroup;
 FREQ_ORDER fo;
 bettergroup.max_w=0;
 bettergroup.max_n=0;

 printf("FUNCTION [get_maxgroup] BEGIN\n");
```

```
for(i=0; i<length; i++)            /*initial*/
bettergroup.seq[i]=0;

for(i=0; i<MAX_OUT; i++)     /*initial*/
{
 candidate_group[i]=0;
 candidate_posi[i]=0;
 candidate_orig_position[i]=0;
 can_orig_position[i]=0;
 can_seq[i]=0;
}/*for end*/

fo=frequency_ordering(flag, ct);                    /*get frequency ordering*/

/******** get candidate group ********/

for (i=0; i<data.out_number; i++)
{
  if (fo.posi_number[i]>=length/2)
  {
   candidate_group[i]=fo.sort_line[i];
   candidate_posi[i]=fo.posi_number[i];
   candidate_orig_position[i]=fo.orig_position[i];
   candidate_number++;
  }

  if (fo.posi_number[i]<length/2)
  break;
}/*for end*/

printf("the sequences in candidate group are\n");
for (i=0; i<candidate_number; i++)
printf("candidate_group[%d]=%d\n", i, candidate_group[i]);
printf("candidate_number=%d\n", candidate_number);

/******** select candidate squences ********/

if (candidate_number>=2)
{
 can_seq[0]=candidate_group[0];
 can_seq[1]=candidate_group[1];
 can_orig_position[0]=candidate_orig_position[0];
 can_orig_position[1]=candidate_orig_position[1];
 cs=2;/* the number of sequences in the candidate lists*/

 for (i=2; i<candidate_number; i++)
 {
  if (candidate_posi[i]==candidate_posi[1])
  {
   can_seq[cs]=candidate_group[i];
   can_orig_position[cs]=candidate_orig_position[i];
   cs++;
  }/*if end*/
  else if (candidate_posi[i]!=candidate_posi[1])
  break;

 } /*for end*/

 /******** get good groups starting x0 ********/

 for (i=0; i<cs-1; i++)                    /*for 1 begin*/
 {
  t=0;
  goodgroup[i].max_n=0;
  goodgroup[i].max_w=0;
  goodgroup[i].line_number[t]=can_seq[i];

  t++;
  goodgroup[i].max_n++;
```

```
    for(m=0; m<length; m++)
    operater[m]=data.out_seq[can_orig_position[i]][m];

    for(j=i+1; j<cs; j++)            /*for 2 begin*/
    {
     for(m=0; m<length; m++)   /*for 3 begin*/
     {
      if (flag==1)
      { temper[m]=operater[m]&data.out_seq[can_orig_position[j]][m]; }

      else if (flag==0)
      { temper[m]=operater[m]|data.out_seq[can_orig_position[j]][m]; }
     }/*for 3 end*/

     N=count_number(flag, temper);

     if (N>=(float)length/2)
     {
      copy(operater, temper);
      goodgroup[i].line_number[t]=can_seq[j];
      t++;
      goodgroup[i].max_n++;
      goodgroup[i].max_w=N;
     }/*if end*/

     else if (N<(float)length/2)
     { copy(operater, operater); }
    }/*for2 end*/

   copy(goodgroup[i].seq, operater);
  }         /*for 1 end*/
}/*if end*/

/******** get better group from good groups ********/

for (i=0; i<cs-1; i++)
{
 if((goodgroup[i].max_n>bettergroup.max_n)&&(goodgroup[i].max_n>=2)&&(goodgroup[i].max_w>=(float)length/2))
 {
  bettergroup.max_n=goodgroup[i].max_n;
  bettergroup.max_w=goodgroup[i].max_w;

  for (j=0; j<goodgroup[i].max_n; j++)
  { bettergroup.line_number[j]=goodgroup[i].line_number[j]; }   /*for 2 end*/

  copy(bettergroup.seq, goodgroup[i].seq);

 }         /*if end*/
}/*for 1 end*/

/******** select new candidate sequences ********/

while((cs<candidate_number)&&(candidate_number>=2))
{
 v=cs;
 can_seq[v]=candidate_group[v];
 can_orig_position[v]=candidate_orig_position[v];
 cs++;

 for (i=v+1; i<candidate_number; i++)
 {
  if (candidate_posi[i]==candidate_posi[v])
  {
   can_seq[cs]=candidate_group[i];
   can_orig_position[cs]=candidate_orig_position[i];
   cs++;
  }/*if end*/

  else if (candidate_posi[i]!=candidate_posi[1])
```

```
      break;

    } /*for end*/

  /******** get better group from new candidate sequences ********/

    copy(operater, bettergroup.seq);

    for(j=v; j<cs; j++)  /*for 2 begin*/
    {
     for(m=0; m<length; m++)     /*for 3 begin*/
     {
      if (flag==1)
      { temper[m]=operater[m]&data.out_seq[can_orig_position[j]][m]; }

      else if (flag==0)
      { temper[m]=operater[m]|data.out_seq[can_orig_position[j]][m]; }
     }/*for 3 end*/

     N=count_number(flag, temper);

     if (N>=length/2)
     {
      copy(operater, temper);
      copy(bettergroup.seq, operater);
      bettergroup.line_number[bettergroup.max_n]=can_seq[j];

      bettergroup.max_n++;
      bettergroup.max_w=N;
     }/*if end*/
    } /*for2 end*/
  }/* do end*/

  printf("FUNCTION [get_maxgroup] END\n");

  return(bettergroup);

}/*function get_maxgroup() end*/



/*****************************************************************
                        form_covertable()
              FUCNTION FOR FORMING 1/0 COVER TABLE
*****************************************************************/

COVER_TABLE form_covertable(int flag)
{
 int i, j, m, u;
 int v=0;
 int temp[MAX_OUT];

 COVER_TABLE ct;
 ct.seq_no=0;

 printf("FUNCTION [form_covertable] BEGIN\n");

 /******** initial the 1/0 cover table ********/

 for (i=0; i<MAX_OUT; i++)
 {
  temp[i]=0;
  for (j=0; j<length; j++)
  { ct.cover_table[i][j]=0;  }/* for 2 end*/
 }/*for 1 end*/

 /******** form the 1 covertable ********/

 if(flag==1)
 {
```

```
   for (i=0; i<data.out_number; i++)/*for 1 begin*/
   {
    if(data.discard[i]!=1)
    {
     for (j=0; j<length; j++)
     {
      if(data.out_seq[i][j]==1)
      {
       for (u=0; u<data.out_number; u++)
       {
        if(ct.cover_table[u][j]==0)
        {
         ct.cover_table[u][j]=data.init_values[i];
         break;
        }
       }/*for 3 end*/
      }/*if end*/
     }/*for 2 end*/
    }/*if end*/

    else
    {printf(" this sequence is discarded!!\n\n");}
   }/*for 1 end*/
 } /*if end*/

 /******** form the 0 covertable ********/

 if(flag==0)
 {
  for (i=0; i<data.out_number; i++)/*for 1 begin*/
  {
   if(data.discard[i]!=1)
   {
    for (j=0; j<length; j++)
    {
     if(data.out_seq[i][j]==0)
     {
      for (u=0; u<data.out_number; u++)
      {
       if(ct.cover_table[u][j]==0)
       {
        ct.cover_table[u][j]=data.init_values[i];
        break;
       }
      }/*for 3 end*/
     }/*if end*/
    }/*for 2 end*/
   }/*if end*/

   else
   {printf(" this sequence is discarded!!\n\n");}
  }/*for 1 end*/
 }/*if end*/

 /******** find the size of covertable ********/
 for (j=0; j<length; j++)
 {
  for (i=0; i<data.out_number+1; i++)
  {
   if(ct.cover_table[i][j]==0)
   {
    temp[v]=i;
    break;
   }
  }
  v++;
 }/*for end*/

 for(i=0; i<length; i++)
 {
```

```
  if (ct.seq_no<temp[i])
  ct.seq_no=temp[i];
 }/*for end*/

 printf("FUNCTION [form_covertable] END\n");

 return(ct);

}/*function form_covertable() end*/



/*****************************************************************
                    multiline_construct_tree()
          FUCNTION FOR MERGERING MULTILINE OUTPUTS
*****************************************************************/

multiline_construct_tree()
{
 int d, i, j, N1, N2;
 int old_outnumber;
 COVER_TABLE one_ct, zero_ct;
 BESTGROUP one_maxgroup, zero_maxgroup;
 new_line=data.init_values[data.out_number-1];

 printf("FUNCTION [multiline_construct_tree] BEGIN\n");

 do
 {                                          /*first do begin*/

  /******** level one ********/

  discard_number=data.out_number; /*when one line merged this number*/

  /******** select the sequences merged by gate AND ********/

  do
  {
   one_ct=form_covertable(1);  /*get 1 cover table*/
   one_maxgroup=get_maxgroup(1,one_ct);

   if((one_maxgroup.max_n>=2)&&(one_maxgroup.max_w>=length/2))
   {  merge_seq(1, one_maxgroup);  } /*1 means merge by AND*/
      }while((one_maxgroup.max_n>=2)&&(one_maxgroup.max_w>=length/2)&&(discard_number>1));

  /******** select the sequences merged by gate OR  ********/

  do
  {
   zero_ct=form_covertable(0);  /*get 0 cover table*/
   zero_maxgroup=get_maxgroup(0, zero_ct);

   if((zero_maxgroup.max_n>=2)&&(zero_maxgroup.max_w>=length/2))
   {  merge_seq(0, zero_maxgroup);  }  /* 0 means merge by OR*/

}while((zero_maxgroup.max_n>=2)&&(zero_maxgroup.max_w>=length/2)&&(discard_number>1));

  /******** merge the remaining sequences by XOR  ********/

  if (discard_number>=2)  /*if left more than 2 line to merge by XOR*/
  {  final_merge();  }  /*if left more than 2 line to merge by XOR end*/


  /******** setup new input by the left line and merger output  ********/

  d=0;                                                  /*d is the new out put numbers*/
  for(i=0; i<data.out_number; i++)                      /*install the remaining sequence in
                                  the new data list*/
  {
   if(data.discard[i]!=1)
```

```
   {
    data.init_values[d]=data.init_values[i];
    data.discard[d]=0;
    for(j=0; j<length;j++)
    { data.out_seq[d][j]=data.out_seq[i][j]; }/*for end*/
    d++;
   }/*if end*/
  }/*for end*/

  for(i=0; i<merge_no; i++)                    /*instal new outputs of gates in the
                                                 new data list*/
  {
   data.init_values[d]=merge_output_line[i];
   data.discard[d]=0;
   for(j=0; j<length;j++)
   {
    data.out_seq[d][j]=merge_result[i][j];
   }/*for end*/
   d++;
  }/*for end*/

  old_outnumber=data.out_number;
  data.out_number=d;

  for(i=data.out_number; i<old_outnumber; i++)
  {
   data.init_values[i]=0;
   data.discard[i]=0;
   for(j=0; j<length;j++)
   {
    data.out_seq[i][j]=0;
   }/*for end*/
  }/*for end*/

  for (i=0; i<merge_no; i++)  /*reset */
  {
   merge_output_line[i]=0;
   for (j=0; j<length; j++)
   {   merge_result[i][j]=0;   }/*for end*/
  }          /*for end*/

  merge_no=0;

  in=fopen("result_compress.dat", "a");
  if(in)
  {
   fprintf(in, "------new output is-----\n\n");
   fprintf(in, "data.out_number is %d\n", data.out_number);
   fprintf(in, "data.length is %d\n", data.length);
   fprintf(in, "The input sequences are :\n");
   for (i=0; i<data.out_number; i++)
   {
    fprintf(in, "\n");
    for (j=0; j<data.length; j++)
    { fprintf(in, "%d", data.out_seq[i][j]);} /*for loop end*/
   }         /*for loop end*/
   fprintf(in, "\n");
   fprintf(in, "outline number are: \n");
   for(i=0; i<data.out_number; i++)
   {
    fprintf(in, "outline number is %d\t",data.init_values[i]);
    fprintf(in, "discard[%d]=%d\n", i, data.discard[i]);
   }         /*for end*/
  }          /*if (in) end*/

  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

 }while(data.out_number>1);     /*first do end*/
```

```
 printf("FUNCTION [multiline_construct_tree] END\n");

}/*function multiline_construct_tree() end*/



/*****************************************************************
                        read_init_values()
      FUCNTION FOR GETTING THE INITIAL OUTPUT VALUES
    DEFINED IN THE ISCAS CIRCUIT DESCTRIPTION FROM A FILE
*****************************************************************/

 read_init_values(int init_values[MAX_OUT], int outnumber)
 {
  int out_number;
  register i;
  out_number=outnumber;

  printf("FUNCTION [read_init_values] BEGIN\n");

  in=fopen("outline_num.dat", "r");
  if(in)
  {
   while(!feof(in))
   {
    for(i=0; i<out_number; i++)
    {
     fscanf(in, "%d\n", &init_values[i]);
     if (init_values[i]>=0)
     {  init_values[i]=init_values[i];  }     /*if end*/
     else
     {
      printf("The initial line number should be positive\n");
      printf("Check your data file\n");
      exit(1);
     }
    }/* for end*/
   }/*while end*/
  }/*if end*/
  else
  printf("Error opening out_line_numb.dat file\n");
  fclose(in);

  printf("FUNCTION [read_init_values] END\n");

} /*function read_init_values() end*/



/*****************************************************************
                        getdata_file()
          FUCNTION FOR GETTING DATA FROM A FILE
*****************************************************************/

getdata_file()
{
 int outnumber;
 int init_values[MAX_OUT];

 register i, j;

 printf("FUNCTION [getdata_file] BEGIN\n");

 in=fopen("result_compress.dat","a");
 if(in)
 {
  fprintf(in, "\n\n\n    The data read from file as following:\n");
  fprintf(in, "************************************************\n\n\n");
 }
```

```
 fclose(in);

 in = fopen("out_seq.dat", "r");
 while(!feof(in))
 {
  fscanf(in, "%d\n", &data.out_number);
  fscanf(in, "%d\n", &data.length);

  length=data.length;

  if ((data.out_number>0)&&(data.length>0))
  {
   for (j=0; j<data.length; j++)
     {
       for (i=0; i<data.out_number; i++)
       {
        fscanf(in, "%d", &data.out_seq[i][j]);
        if((data.out_seq[i][j]!=0)&&(data.out_seq[i][j]!=1))
        {
         printf("Value of sequence should be either 1 or 0\n");
         printf("Pls. check the data file\n");
         exit(1);
        }/*end if */
       }/*end of for loop*/
     } /*end of for loop*/
  }/*end if*/
  else
  {
   printf("Numberof bit strams and sequence length should be positive\n");
   printf("Pls. check the data file\n");
   exit(1);
  }
}/*end of while loop*/

fclose(in);

in=fopen("result_compress.dat", "a");
if(in)
{
 length=data.length;
 printf("data.out_number is %d\n", data.out_number);
 printf("data.length is %d\n", data.length);
 printf("The input sequences are :\n");
 fprintf(in, "The input sequences are:\n");
 for (i=0; i<data.out_number; i++)
 {
  printf("\n");
  fprintf(in, "\n");
  for (j=0; j<data.length; j++)
  {
   printf("%d", data.out_seq[i][j]);
   fprintf(in, "%d", data.out_seq[i][j]);
  }/*for loop end*/
 }/*for loop end*/
 printf("\n");
 fprintf(in, "\n\n");
}/* if end*/
else
printf("Error opening result_compress.dat file\n");
fclose(in);

outnumber=data.out_number;
read_init_values(init_values, outnumber);

in=fopen("result_compress.dat", "a");
if(in)
{
 fprintf(in, "outline number are: \n");
 for(i=0; i<data.out_number; i++)
 {
```

```
    data.init_values[i]=init_values[i];
    printf("outline number is %d\n",data.init_values[i]);
    fprintf(in, "%d\n",data.init_values[i]);
   }

   fprintf(in, "\ndata.out_number is %d\n", data.out_number);
   fprintf(in, "\ndata.length is %d\n\n", data.length);
   fprintf(in, "*****************************************\n\n\n");
  }          /*if(in) end*/

  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

  printf("FUNCTION [getdata_file] END\n");

}          /*function getdata_file() end*/



/*********************************************************************
                              main()
                          MAIN FUNCTION
*********************************************************************/

main()
{
 int i, j, rt, number;

 printf("FUNCTION [main]  BEGIN\n");

 do{
 printf("\n\n---------MENU--------\n");
 printf("---------------------\n\n");
 printf("Please enter your choice.\n\n");
 printf("1: Read sequences from a file.\n\n");
 printf("2: Creating the compaction tree\n");
 printf("   with stochastically independent probability values\n");
 printf("   of multiple line errors\n\n");
 printf("0: Exit the program.\n\n");
 scanf("%d",&number);

 switch(number)
 {
  case 1:
  {
   getdata_file();
   break;
  }/*case 1 end*/

  case 2:
  {
   multiline_construct_tree();
   in=fopen("merge_result.dat", "a");
   if(in)
   {
    gettime(&inittime, &simtime, &runtime);
    fprintf(in, "\n\nCPU time\n");
    fprintf(in, "Initialization: %.4f secs.\n", inittime);
    fprintf(in, "Simulation    : %.4f secs.\n", simtime);
    fprintf(in, "Total         : %.4f secs.\n", runtime);
    fprintf(in, "*****************************************\n\n\n");

   }/* if end*/
   else
   printf("Error opening result_compress.dat file\n");
   fclose(in);

   break;
  }/*case 2 end*/
```

```
   case 0:
   {
    printf("Exit the program.\n\n");
    break;
   }/*case 0 end*/

   default:
   { printf("You enter the wrong number, please star againg.\n"); }
  }/*switch end*/
 }while(number!=0);

 printf("FUNCTION [main] END\n");

}/*function main() end*/
```

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada  K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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

/***************************** HISTORY ***************************************

        JTesting_II: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        defin.h
        JTesting_II performs output compression for combinational circuits
        based on sequence characterization and stochastic dependence
        of line errors. Output sequences can be read from a user supplied
        file.

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



/*****************************************************************
                        Defin.h
        Define All Global Variables Used in Jtesting_II
*****************************************************************/

#define MAX_OUT 300                             /*maximum number of primary outputs*/
#define MAX_BITS 250                            /*maximum Length of a sequence*/
#define PTREE_NO 3                              /*input number of xor consist of parity tree*/
int stage;
int new_line;                                   /*Line Number Created After compression*/
int length;                                     /*length of the sequence*/
int discard_number;                             /*it will reduce 1 when one is merged*/
int merge_no;
int merge_result[MAX_OUT][MAX_BITS];            /* put the merge result sequence */
int merge_output_line[MAX_OUT];                 /*keep the merged output line number*/
FILE *in, *out;                                 /*input/output File Pointers*/
float minutes, seconds, inittime, simtime, runtime;
```

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used : for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada  K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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

/***************************** HISTORY ***************************************

        JTesting_II: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        exhaustive.c
        JTesting_II performs output compression for combinational circuits
        based on sequence characterization and stochastic dependence
        of line errors. Output sequences can be read from a user supplied
        file.

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


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
#include <string.h>
#include "defin.h"
#include<malloc.h>
#include<memory.h>

struct OUTDATA                                  /*struct to store the output data*/
{
 int out_number;                                /*Number of Primary Outputs*/
 int length;                                    /*Length of Sequence*/
 int out_seq[MAX_OUT][MAX_BITS];                /*Storage for Output Sequences*/
 int init_values[MAX_OUT];                      /*Initial Line Number*/
 int discard[MAX_OUT];
} ;

struct OUTDATA data;

typedef struct
{
 int cover_table[MAX_OUT][MAX_BITS];            /*cover_table*/
 int seq_no;                                    /*rower numbers in  cover table*/
}COVER_TABLE;

typedef struct
{
```

```
int line_number[MAX_OUT];                    /*output line number in the best group*/
int max_w;                                   /*maxnumber(0/1) matching bits in the best group*/
 int max_n;                                  /* maxnumber sequences in best group*/
}BESTGROUP;

typedef struct
{
    int posi_number[MAX_OUT];                /*count 1/0 numbers of bit position in 1/0 covertable*/
 int sort_line[MAX_OUT];                     /*after sorting, the name of line*/
 int orig_position[MAX_OUT];                 /* store the original position in the original input*/
    int freorder_no;                         /* sequence number on the frequency ordering*/
}FREQ_ORDER;                                 /*keep some information about frequency ordering*/

int candidate_seq[MAX_OUT][MAX_BITS];        /*the candidate sequences from original list*/
int A[MAX_OUT];                              /* keep the vertical weights*/


/*********************************************************************
                          gettime()
    FUCNTION FOR GETTING CPU TIME USED IN THE PROGRAM
*********************************************************************/

void gettime(float *usertime, float *systemtime, float *total)
{
  struct tms timesbuffer;
  time_t totaltime;
  time_t utime;
  time_t stime;

  printf("FUNCTION [gettime] BEGIN\n");

  times(&timesbuffer);
  utime=timesbuffer.tms_utime;
  stime=timesbuffer.tms_stime;
  totaltime=timesbuffer.tms_utime+timesbuffer.tms_stime; /*In 60th seconds*/
  *usertime=(float)utime/60.0;
  *systemtime=(float)stime/60.0;
  *total=(float)totaltime/60.0;
  printf("inittime is %.3f\n", *usertime);
  printf("simtime is %.3f\n", *systemtime);
  printf("total is %.3f\n", *total);

  printf("FUNCTION [gettime] END\n");

}/*function gettime() end */


/*********************************************************************
                          copy()
        FUCNTION FOR MAKING THE TWO SEQUENCES SAME
*********************************************************************/

copy(int *target, int *source)
{
  int m;

  printf("FUNCTION [copy] BEGIN\n");
  for (m=0; m<length; m++)
  target[m]=source[m];
  printf("FUNCTION [copy] END\n");

}/*function copy() end*/
```

```
/*****************************************************************
                        count_number()
   FUCNTION FOR GETTING 1'S OR 0'S NUMBER IN A SEQUENCE
*****************************************************************/

int count_number(int flag,int *candi_list)
{
 int i, result;
 int count=0;

 printf("FUNCTION [count_number] BEGIN\n");

 if (flag==1)
 {
  for (i=0; i<length; i++)
  { count+=candi_list[i]; }/*for end*/
  result=count;
 }/* if end*/
 else if (flag==0)
 {
  for (i=0; i<length; i++)
  {
   if (candi_list[i]==0)
   count++;
   else
   count=count;
  }/*for end*/
  result=count;
 }/*else if end*/

 printf("FUNCTION [count_number] END\n");
 return(result);

}/*function count_number() end*/



/*****************************************************************
                        merge_seq();
                FUCNTION FOR MERGERING SEQUENCES
          WITHIN SELECTED MAXGROUPBY BY SELECTED GATE
*****************************************************************/

merge_seq(int flag_gate, BESTGROUP maxgroup)
{

 int i, j, m, u,v,w,W, R;
 int result_seq[MAX_BITS];
 int counter, result;

 printf("FUNCTION [merge_seq] BEGIN\n");

 for (m=0; m<MAX_BITS; m++)                          /*reset*/
 {
  if(flag_gate==1)
  result_seq[m]=1;
  else if ((flag_gate==2)||(flag_gate==3))
  result_seq[m]=0;
 }
 switch (flag_gate)
 {
  case 1:                                            /*merge by gate AND*/
  {
   for (u=0; u<maxgroup.max_n; u++)/*for 1 begin*/
    {
     for (v=0; v<data.out_number; v++)/*for 2 begin*/
     {
      if ((data.discard[v]!=1)&&(maxgroup.line_number[u]==data.init_values[v]))
      {
       data.discard[v]=1;
```

```
        for( w=0; w<data.length; w++)
        {
         result_seq[w]=result_seq[w]&data.out_seq[v][w];
         data.out_seq[v][w]=9;
        }/*for end*/
       }/*if end*/
      }/*for 2 end*/
     discard_number--;
    }/*for 1 end*/
    new_line++;
    merge_output_line[merge_no]=new_line;
    for (j=0; j<length; j++)
    { merge_result[merge_no][j]=result_seq[j]; }
    in=fopen("merge_result.dat", "a");
    if(in)
    {
     fprintf(in, "%d = AND(", new_line);
     for (i=0; i<maxgroup.max_n; i++)
     { fprintf(in, "%d, ", maxgroup.line_number[i]);}
     fprintf(in, ")\n");
    }/*if(in) end*/
    else
    printf("Error opening merge_result.dat file\n");
    fclose(in);
    merge_no++;
    stage++;
    result=1;
    break;
   }/*case 1 end*/
   case 2:                                          /*merge by gate OR*/
   {
    for (u=0; u<maxgroup.max_n; u++)/*for 1 begin*/
    {
     for (v=0; v<data.out_number; v++)/*for 2 begin*/
     {
      if ((data.discard[v]!=1)&&(maxgroup.line_number[u]==data.init_values[v]))
      {
       data.discard[v]=1;
       for( w=0; w<data.length; w++)
       {
        result_seq[w]=result_seq[w]|data.out_seq[v][w];
        data.out_seq[v][w]=9;
       } /*for end*/
      }/*if end*/
     }/*for 2 end*/
    discard_number--;
   }/*for 1 end*/
   new_line++;
   merge_output_line[merge_no]=new_line;
   for (j=0; j<length; j++)
   { merge_result[merge_no][j]=result_seq[j]; }
   in=fopen("merge_result.dat", "a");
   if(in)
   {
    fprintf(in, "%d = OR(", new_line);
    for (i=0; i<maxgroup.max_n; i++)
    { fprintf(in, "%d, ", maxgroup.line_number[i]); }
    fprintf(in, ")\n");
   }/*if(in) end*/
   else
   printf("Error opening merge_result.dat file\n");
   fclose(in);
   merge_no++;
   stage++;
   result=1;
   break;
  }/*case 2 end*/
  case 3:                                           /*merge by gate XOR*/
  {
   for (u=0; u<maxgroup.max_n; u++)/*for 1 begin*/
```

```
    {
     for (v=0; v<data.out_number; v++)/*for 2 begin*/
     {
      if ((data.discard[v]!=1)&&(maxgroup.line_number[u]==data.init_values[v]))
      {
       data.discard[v]=1;
       for( w=0; w<data.length; w++)
       {
        result_seq[w]=result_seq[w]^data.out_seq[v][w];
        data.out_seq[v][w]=9;
       } /*for end*/
      }/*if end*/
     }/*for 2 end*/

     discard_number--;
    }/*for 1 end*/
    new_line++;
    merge_output_line[merge_no]=new_line;
    for (j=0; j<length; j++)
    {  merge_result[merge_no][j]=result_seq[j];  }
    in=fopen("merge_result.dat", "a");
    if(in)
    {
     fprintf(in, "%d = XOR(", new_line);
     for (i=0; i<maxgroup.max_n; i++)
     {  fprintf(in, "%d, ", maxgroup.line_number[i]); }
     fprintf(in, ")\n");
    }                    /*if(in) end*/
    else
    printf("Error opening merge_result.dat file\n");
    fclose(in);
    merge_no++;
    stage++;
    result=1;
    break;
   }/*case 3 end*/
 }/*switch end*/

 printf("FUNCTION [merge_seq] END\n");

}/*function merge_seq() end*/



/*****************************************************************
                        factorial()
          RECURSIVE FUCNTION FOR CALCULATING
          THE FACTORIAL OF A POSITIVE INTEGER
*****************************************************************/

double factorial(int n)
{
 double result;

 printf("FUNCTION [factorial]  BEGIN\n");
 if(n==0)
 result=1;
 else
 result= n*factorial(n-1);
 printf("result=%f\n", result);
 printf("FUNCTION [factorial]  END\n");
 return(result);

} /* function factorial() end*/
```

```
/*****************************************************************
                        get_Ti()
          FUCNTION FOR GETTING Ti MISSED ERROR
               PROBABLITY ESTIMATE OF Ith LINE
*****************************************************************/

double get_Ti(int N, int I)
{
 double u=1.0/3.0;
 double v;
 double w;

 printf("FUNCTION [get_ti]  BEGIN\n");

 if(I>=2&&I<=N)
 {
  v=N-I;
  w=((2.0/3.0)*(pow(u,v)));
  printf("w=%f\n", w);
 }          /* if end*/
 else if (I==1)
 {
  v=N-2;
  w=((1.0/3.0)*(pow(u,v)));
  printf("w=%f\n", w);
 }

 printf("FUNCTION [get_ti]  END\n");
 return(w);

}/* function get_Ti() end*/



/*****************************************************************
                        get_Dand()
                FUCNTION FOR THE MISSED ERROR
              PROBABLITY ESTIMATE FOR GATE AND
 *****************************************************************/

double get_Dand(int max_n)
{
 int i, j,m;
 double D_and;                          /* probability*/
 double C, T;
 double sum_A=0.0;                      /* sum of the vertical weights A0 to A(n-1)*/
 double sum_B=0.0;                      /* sum of Ti*A[N-i]/C*/

 printf("FUNCTION [get_Dand] BEGIN\n");

 for (i=0; i<max_n; i++)
 sum_A+=A[i];
  for (i=1 ; i<=max_n; i++)
 {
  j=max_n-i;
  C=factorial(max_n)/(factorial(max_n-i)*factorial(i));
  T=get_Ti(max_n, i);
  sum_B+=T*A[j]/C;
 }/*for end*/
 D_and=(1.0*sum_A/data.length)-(sum_B/data.length);

 printf("FUNCTION [get_Dand] END\n");
 return(D_and);
}/*function get_Dand() end*/
```

```
/*****************************************************************
                        get_Dor()
                FUCNTION FOR THE MISSED ERROR
               PROBABLITY ESTIMATE FOR GATE OR
*****************************************************************/

double get_Dor(int max_n)
{
 int i, j,m;
 double D_or;                                   /* probability*/
 double C, T;
 double sum_A=0.0;                              /* sum of the vertical weights A1 to A(n)*/
 double sum_B=0.0;                              /* sum of Ti*A[i]/C*/

 printf("FUNCTION [get_Dor] BEGIN\n");

 for (i=1; i<=max_n; i++)
 sum_A+=A[i];
 for (i=1; i<=max_n; i++)
 {
  j=i;
  C=factorial(max_n)/(factorial(max_n-i)*factorial(i));
  T=get_Ti(max_n, i);
  sum_B+=T*A[j]/C;
}/*for end*/
D_or=(1.0*sum_A/data.length)-(sum_B/data.length);

printf("FUNCTION [get_Dor] END\n");
 return(D_or);
}/*function get_Dor() end*/



/*****************************************************************
                        get_Dxor()
                FUCNTION FOR THE MISSED ERROR
               PROBABLITY ESTIMATE FOR GATE XOR
*****************************************************************/

double get_Dxor(int max_n)
{
 int i, j,m;
 double D_xor=0.0;                              /* probability*/

 printf("FUNCTION [get_Dxor] BEGIN\n");
 m=max_n%2;
 if (m==1)
 {
  for (i=2; i<=max_n-1; i=i+2)
  D_xor+=get_Ti(max_n, i);
 }
 else if (m==0)
 {
  for (i=2; i<=max_n; i=i+2)
  D_xor+=get_Ti(max_n, i);
 }
 printf("FUNCTION [get_Dxor] END\n");
 return(D_xor);
}/*function get_Dxor() end*/



/*****************************************************************
                        get_candi_seq():
          FUCNTION FOR GETTING THE CANDIDATE SEQUENCES
                    FROM THE ORIGINAL LIST
*****************************************************************/

get_candi_seq(int max_n, int *line_number)
{
```

```
 int i, j, m, n, u, v, w;
 int h=0;

 printf("FUNCTION [get_candi_seq] BEGIN\n");

 for (m=0; m < MAX_OUT; m++)                    /*reset*/
 {
  for(n=0; n<MAX_BITS; n++)
  { candidate_seq[m][n]=0; }
 }/*for end*/

 for (u=0; u<max_n; u++)/*for 1 begin*/
 {
  for (v=0; v<data.out_number; v++)/*for 2 begin*/
  {
   if (line_number[u]==data.init_values[v])
   {
    for( w=0; w<data.length; w++)
    { candidate_seq[h][w]=data.out_seq[v][w]; } /*for end*/
    h++;
   }/*if end*/
  }/*for 2 end*/
 }/*for 1 end*/

 printf("FUNCTION [get_candi_seq] END\n");

}/*function get_candi_seq() end*/


/*********************************************************************
                        vertical_weight();
              FUCNTION FOR GETTING VERTICAL WEITHTS
*********************************************************************/

vertical_weight(int max_n)
{
 int i, j,m, n, u, v;
 int count;

 printf("FUNCTION [vertical_weight] BEGIN\n");

 for(u=0; u<MAX_OUT; u++)
 { A[u]=0;}
 for (i=0; i<data.length; i++) /*for 1 begin*/
 {
  count=0;
  for (j=0; j<max_n; j++)
  { count+=candidate_seq[j][i]; }
  for(v=0; v<=max_n; v++)
  {
   if (count==v)
   {
    A[v]++;
    break;
   }/*if end*/
  }/*for end*/
 } /*for 1 end*/

 printf("FUNCTION [vertical_weight] END\n");

}/*function vertical_weight() end*/


/*********************************************************************
                        intcmp();
                  FUCNTION FOR COMPARING
*********************************************************************/
```

```
int intcmp(const void *v1, const void *v2)
{   return(*(int*)v2-*(int*)v1);  }/*function intcmp() end*/


/*****************************************************************
                        frequency_ordering();
          FUCNTION FOR GETTING FREQUENCY ORDERING
               WITHIN THE ONE/ZERO COVER TABLE
*****************************************************************/

FREQ_ORDER frequency_ordering(int flag, COVER_TABLE ct)
{
 int i, j,m;
 int h=0;
 int temper[MAX_OUT];
 int line[MAX_OUT];
 int position[MAX_OUT];                          /*stor original position of original input*/
 FREQ_ORDER fo;

 printf("FUNCTION [frequency_ordering] BEGIN\n");

 for(i=0; i<MAX_OUT; i++)     /*initial the value*/
 {
  fo.posi_number[i]=0;
  fo.sort_line[i]=0;
  fo.orig_position[i]=0;
  fo.freorder_no=0;
  temper[i]=0;
  line[i]=0;
  position[i]=0;
 }

/**********count the bit position**********/
 for(m=0;m<data.out_number; m++)
 {
  for(j=0; j<length; j++)
  {
   for(i=0; i<ct.seq_no; i++)
   {
    if(ct.cover_table[i][j]==data.init_values[m])
    {  fo.posi_number[h]++;  }
   }/*for end*/
  }/*for end*/
  if(fo.posi_number[h]!=0)
  {
   line[h]=data.init_values[m];
   position[h]=m;
   h++;
  }
 }/*for end*/
  fo.freorder_no=h;
  for (i=0; i<data.out_number; i++)
  { temper[i]=fo.posi_number[i];}

/**********sort-frequency ordering**********/

qsort(fo.posi_number, data.out_number, sizeof(fo.posi_number[0]), intcmp);

/**********et the relative original line number**********/
for (i=0; i<data.out_number; i++)
{
 for(j=0; j<data.out_number; j++)
 {
  if((fo.posi_number[i]==temper[j])&&(temper[j]!=999))
  {
   fo.sort_line[i]=line[j];
   fo.orig_position[i]=position[j];
   temper[j]=999;
   break;
```

```
    }/*if end*/
   }/*foe end*/
}/*for end*/

printf("FUNCTION [frequency_ordering] END\n");
return(fo);

}/*function frequency_ordering() end*/



/*****************************************************************
                              get_maxgroup();
                        FUCNTION FOR GETTING MAXGROUP
                        WITHIN THE ONE/ZERO COVER TABLE
*****************************************************************/

BESTGROUP get_maxgroup(int flag, COVER_TABLE ct)
{
 int h, k, i, j, jj, m, u, v, p1, W1, N1,N;
 int t=0;
 int cs=0;
 int count, max_bit, max_seq;
 int temp[MAX_BITS], oper[MAX_BITS];
 FREQ_ORDER fo;
 BESTGROUP  better_group, best_group;

 /**********initial the values**********/
 best_group.max_w=0;
 best_group.max_n=0;
  for(i=0; i<MAX_OUT; i++)
 {
  better_group.line_number[i]=0;
  best_group.line_number[i]=0;
  for (j=0; j<MAX_BITS; j++)
  {
   temp[j]=0;
   oper[j]=0;
  }
 }

 printf("FUNCTION [get_maxgroup] BEGIN\n");

 fo=frequency_ordering(flag, ct);                    /* get frequency ordering*/

 for( i=0; i<fo.freorder_no; i++) /* for 1 begin*/
 {
  max_bit=0;
  max_seq=1;
  h=0;
  better_group.max_w=0;
  better_group.max_n=0;
   for(u=0; u<MAX_OUT; u++)
  {
   better_group.line_number[u]=0;
   for (v=0; v<MAX_BITS; v++)
   {
    temp[v]=0;
    oper[v]=0;
   }/*for end*/
  }/*for end*/
  better_group.line_number[h]=data.init_values[fo.orig_position[i]];
  for(m=0; m<length; m++)
  { oper[m]=data.out_seq[fo.orig_position[i]][m];  }    /*for end*/
  for(j=i+1; j<fo.freorder_no; j++)
  {
   copy(temp, oper);
   if (flag==1)
   {
    for(m=0; m<length; m++)
```

```
      {temp[m]=temp[m]&data.out_seq[fo.orig_position[j]][m];}
     }/*if end*/
     else if (flag==0)
     {
      for(m=0; m<length;m++)
      {temp[m]=temp[m]|data.out_seq[fo.orig_position[j]][m];}
     }/*end else for*/
    count=count_number(flag, temp);
    if (count>=max_bit)
     {
      h++;
      copy(oper, temp);
      better_group.line_number[h]=data.init_values[fo.orig_position[j]];
      max_seq++;
      max_bit=count;
     }/* if end*/
   }/* for j end*/
   better_group.max_n=max_seq;
   better_group.max_w=max_bit;

   if
(((better_group.max_w>best_group.max_w)&&(better_group.max_n>=2))||((better_group.max_w==best_group.max_w)&&(better_group.max_n>b
est_group.max_n)&&(better_group.max_n>=2)))
    {
     best_group.max_w=better_group.max_w;
     best_group.max_n=better_group.max_n;
     for (m=0; m<best_group.max_n; m++)
     { best_group.line_number[m]=better_group.line_number[m]; } /*for end*/

     for(m=best_group.max_n; m<MAX_OUT; m++)
     best_group.line_number[m]=0;
    }/* if end*/
  }/* for l end*/

 for( i=0; i<fo.freorder_no; i++)
 {
  max_bit=0;
  max_seq=1;
  h=0;
  better_group.max_w=0;
  better_group.max_n=0;
  for(u=0; u<MAX_OUT; u++)
  {
   better_group.line_number[u]=0;
   for (v=0; v<MAX_BITS; v++)
   {
    temp[v]=0;
    oper[v]=0;
   }/*for end*/
  }/*for end*/
  better_group.line_number[h]=data.init_values[fo.orig_position[i]];
  for(m=0; m<length; m++)
  { oper[m]=data.out_seq[fo.orig_position[i]][m];  }/*for end*/
  for(j=0; j<fo.freorder_no; j++)
  {
   if (j!=i)
   {
    copy(temp, oper);
    if (flag==1)
    {
     for(m=0; m<length; m++)
     {temp[m]=temp[m]&data.out_seq[fo.orig_position[j]][m];}
    }      /*if end*/
    else if (flag==0)
    {
     for(m=0; m<length;m++)
     {temp[m]=temp[m]|data.out_seq[fo.orig_position[j]][m];}
    }      /*end else for*/
    count=count_number(flag, temp);
    if (count>=max_bit)
```

```
      {
       h++;
       copy(oper, temp);
       better_group.line_number[h]=data.init_values[fo.orig_position[j]];
       max_seq++;
       max_bit=count;
      }/* if end*/
     } /* if i!=j end*/
    }/* for j end*/
    better_group.max_n=max_seq;
    better_group.max_w=max_bit;

  if
(((better_group.max_w>best_group.max_w)&&(better_group.max_n>=2))||((better_group.max_w==best_group.max_w)&&(better_group.max_n>b
est_group.max_n)&&(better_group.max_n>=2)))
    {
      best_group.max_w=better_group.max_w;
      best_group.max_n=better_group.max_n;
      for (m=0; m<best_group.max_n; m++)
      {  best_group.line_number[m]=better_group.line_number[m]; } /*for end*/
      for(m=best_group.max_n; m<MAX_OUT; m++)
      best_group.line_number[m]=0;
    }/* if end*/
   }/* for 2 end*/

  printf(" FUNCTION [get_maxgroup] END\n\n");
  return (best_group);

}/*function get_maxgroup() end*/



/*****************************************************************
                              form_covertable()
     FUCNTION FOR FORMING 1 COVER TABLE AND 0 COVER TABLE
 *****************************************************************/

COVER_TABLE form_covertable(int flag)
{
 int i, j, m, u;
 int v=0;
 int temp[MAX_OUT];
 COVER_TABLE ct;
 ct.seq_no=0;

 printf("FUNCTION [form_covertable] BEGIN\n");

 /**********initial the 1/0 cover table**********/

 for (i=0; i<MAX_OUT; i++)
 {
  temp[i]=0;
  for (j=0; j<length; j++)
  { ct.cover_table[i][j]=0;  }/* for 2 end*/
 }/*for 1 end*/

 /**********form the 1covertable**********/
 if(flag==1)
 {
  for (i=0; i<data.out_number; i++)          /*for 1 begin*/
  {
   if(data.discard[i]!=1)
   {
    for (j=0; j<length; j++)
    {
     if(data.out_seq[i][j]==1)
     {
      for (u=0; u<data.out_number; u++)
      {
       if(ct.cover_table[u][j]==0)
```

```
          { ct.cover_table[u][j]=data.init_values[i];
            break;
           }
         }/*for 3 end*/
        }/*if end*/
       }/*for 2 end*/
     }/*if end*/
     else
     {printf(" this sequence is discarded!!\n\n");}
   }/*for 1 end*/
 }/*if end*/

 /**********form the 0 covertable**********/

 if(flag==0)
 {
  for (i=0; i<data.out_number; i++)/*for 1 begin*/
  {
   if(data.discard[i]!=1)
   {
    for (j=0; j<length; j++)
    {
     if(data.out_seq[i][j]==0)
     {
      for (u=0; u<data.out_number; u++)
      {
       if(ct.cover_table[u][j]==0)
       {
        ct.cover_table[u][j]=data.init_values[i];
        break;
       }
      }/*for 3 end*/
     }/*if end*/
    }/*for 2 end*/
   }/*if end*/
   else
   {printf(" this sequence is discarded!!\n\n");}
  }/*for 1 end*/
 } /*if end*/

 /**********find the size of cover-table**********/
 for (j=0; j<length; j++)
 {
  for (i=0; i<data.out_number+1; i++)
  {
   if(ct.cover_table[i][j]==0)
   {
    temp[v]=i;
    break;
   }
  }
  v++;
 }/*for end*/

 for(i=0; i<length; i++)
 {
  if (ct.seq_no<temp[i])
  ct.seq_no=temp[i];
 }/*for end*/

 printf("FUNCTION [form_covertable] END\n");

 return(ct);

}/*function form_covertable() end*/
```

```
/*****************************************************************
                        construct_tree()
          FUCNTION FOR MERGERING MULTILINE OUTPUTS
     BY 1/0 - COVER TABLE ASSUMING STOCHATIC DEPENDENCE
                    OF MULTIPLE LINE ERRORS
*****************************************************************/

construct_tree()
{
 int d, i, j, m, n, u, v, w, N1, N2;
 int flag_gate;
 double D_and, D_or, D_xor;                          /*missing detect probablity*/
 int old_outnumber;
 COVER_TABLE one_ct, zero_ct;
 BESTGROUP one_maxgroup, zero_maxgroup;
 BESTGROUP remain;                                   /*after merged by gate AND&OR, the remaining sequences*/
 new_line=data.init_values[data.out_number-1];

 printf("FUNCTION [construct_tree] BEGIN\n");
 do
 {                                                   /*first do begin*/

  /**********level one**********/

  discard_number=data.out_number;                    /*when one line merged this number*/
  for (u=0; u<MAX_OUT; u++)                          /* reset*/
  remain.line_number[u]=0;
  remain.max_w=0;
  remain.max_n=0;

  /*****select the sequences to merge by AND until not find them*****/
  do
  {
   flag_gate=0;                                      /*reset flag_gate*/
   one_ct=form_covertable(1);                        /*get 1 cover table*/
   one_maxgroup=get_maxgroup(1,one_ct);
   get_candi_seq(one_maxgroup.max_n,  one_maxgroup.line_number);
   vertical_weight(one_maxgroup.max_n);
   D_and=get_Dand(one_maxgroup.max_n);
   D_or=get_Dor(one_maxgroup.max_n);
   D_xor=get_Dxor(one_maxgroup.max_n);

   if (D_and<=D_or)
   {
    if (D_and<=D_xor)
    {
     flag_gate=1;
     merge_seq(flag_gate, one_maxgroup);
    }
    else if (D_and>D_xor)
    {
     flag_gate=3;
     merge_seq(flag_gate, one_maxgroup);
    }
   }
  }while((one_maxgroup.max_n>=2)&&(flag_gate==1)&&(discard_number>1));

/*****select the sequences to merge by OR until not find them*****/
do
{
 if (discard_number>=2)  /* if left more than 2 line to merge by OR*/
 {
  flag_gate=0;                                       /*reset flag_gate*/
  zero_ct=form_covertable(0);                        /*get 0 cover table*/
  zero_maxgroup=get_maxgroup(0, zero_ct);
  get_candi_seq(zero_maxgroup.max_n,  zero_maxgroup.line_number);
  vertical_weight(zero_maxgroup.max_n);
  D_and=get_Dand(zero_maxgroup.max_n);
  D_or=get_Dor(zero_maxgroup.max_n);
  D_xor=get_Dxor(zero_maxgroup.max_n);
```

```
    if (D_or<=D_and)
    {
     if (D_or<=D_xor)
     {
      flag_gate=2;
      merge_seq(flag_gate, zero_maxgroup);
     }
     else if (D_or>D_xor)
     {
      flag_gate=3;
      merge_seq(flag_gate, zero_maxgroup);
     }
    }
   }/*if end*/
 }while((zero_maxgroup.max_n>=2)&&(flag_gate==2)&&(discard_number>1));

/*****merge the remaining squences by AND/OR/EXOR*****/

 if (discard_number>=2)  /* if left more than 2 line to merge by EXOR*/
 {
   v=0;
   in=fopen("result_compress.dat", "a");
   if(in)
   {
    remain.max_n=discard_number;
    fprintf(in, "remain.max_n=%d\n", remain.max_n);
    for(u=0; u<data.out_number; u++)
    {
     if(data.discard[u]!=1)
     {
      remain.line_number[v]=data.init_values[u];
      fprintf(in, "remain.line_number[%d]=%d\n", v, remain.line_number[v]);
      v++;
     }     /*if end*/
     fprintf(in, "v=%d\n", v);
    }       /*for end*/
   }/*if(in) end*/
   else
   printf("Error opening result_compress.dat file\n");
   fclose(in);

   get_candi_seq(remain.max_n,  remain.line_number);
   vertical_weight(remain.max_n);
   remain.max_w=A[remain.max_n];
   D_and=get_Dand(remain.max_n);
   D_or=get_Dor(remain.max_n);
   D_xor=get_Dxor(remain.max_n);
    if ((D_and<=D_or)&&(D_and<=D_xor))
   {
    flag_gate=1;
    merge_seq(flag_gate, remain);
   }/*if end*/

   if ((D_or<=D_and)&&(D_or<=D_xor))
   {
    flag_gate=2;
    merge_seq(flag_gate, remain);
   }/*if end*/
   if ((D_xor<=D_and)&&(D_xor<=D_or))
   {
    flag_gate=3;
    merge_seq(flag_gate, remain);
   }/*if end*/
 }/*if left more than 2 line to merge by AND/OR/EXOR end*/

/*****setup new input by the left line and merger output*****/

 d=0;                                          /*d is the new out put numbers*/
 for(i=0; i<data.out_number; i++)              /* instal the remaining sequence in the new data list*/
 {
```

```
      if(data.discard[i]!=1)
      {
       data.init_values[d]=data.init_values[i];
       data.discard[d]=0;
       for(j=0; j<length;j++)
       { data.out_seq[d][j]=data.out_seq[i][j]; }
       d++;
      }/*if end*/
     }/*for end*/
     for(i=0; i<merge_no; i++)                    /* instal new outputs of gates in the new data list*/
     {
      data.init_values[d]=merge_output_line[i];
      data.discard[d]=0;
      for(j=0; j<length;j++)
      { data.out_seq[d][j]=merge_result[i][j]; }
      d++;
     }/*for end*/
     old_outnumber=data.out_number;
     data.out_number=d;
     for(i=data.out_number; i<old_outnumber; i++)     /*set the left of the data list as zero*/
     {
      data.init_values[i]=0;
      data.discard[i]=0;
      for(j=0; j<length;j++)
      { data.out_seq[i][j]=0; }
     }/*for end*/
     for (i=0; i<merge_no; i++) /*reset */
     {
      merge_output_line[i]=0;
      for (j=0; j<length; j++)
      { merge_result[i][j]=0; }/*for end*/
     }/*for end*/
     merge_no=0;
    }while(data.out_number>1);

    printf("FUNCTION [construct_tree] END\n");

    }/*function construct_tree() end*/



    /*****************************************************************
                                read_init_values()
       FUCNTION FOR GETTING THE INITIAL OUTPUT VALUES
       DEFINED IN THE ISCAS CIRCUIT DESCTRIPTIONFROM A FILE
    *****************************************************************/

    read_init_values(int init_values[MAX_OUT], int outnumber)
    {
     int out_number;
     register i;
     out_number=outnumber;

     printf("FUNCTION [read_init_values] BEGIN\n");

     in=fopen("outline_num.dat", "r");
     if(in)
     {
      while(!feof(in))
      {
       for(i=0; i<out_number; i++)
       {
        fscanf(in, "%d\n", &init_values[i]);
        if (init_values[i]>=0)
        { init_values[i]=init_values[i]; }        /*if end*/
        else
        {
         printf("The initial line number should be positive\n");
         printf("Check your data file\n");
         exit(1);
```

```
        }
      }       /* for end*/
    }         /*while end*/
  }           /*if end*/

  else
  printf("Error opening out_line_numb.dat file\n");
  fclose(in);

  printf("FUNCTION [read_init_values] END\n");

} /*function read_init_values() end*/



/*********************************************************************
                              getdata_file()
          FUCNTION FOR GETTING DATA FROM A FILE
*********************************************************************/

getdata_file()
{
 int outnumber;
 int init_values[MAX_OUT];
 register i, j;

 printf("FUNCTION [getdata_file] BEGIN\n");
 in=fopen("result_compress.dat","a");
 if(in)
 {
  fprintf(in, "\n\n\n   The data read from file as following:\n");
  fprintf(in, "*************************************************\n\n\n");
 }
 fclose(in);

 in = fopen("out_seq.dat", "r");
 while(!feof(in))
 {
  fscanf(in, "%d\n", &data.out_number);
  fscanf(in, "%d\n", &data.length);
  length=data.length;
  if ((data.out_number>0)&&(data.length>0))
  {
   for (j=0; j<data.length; j++)
    {
      for (i=0; i<data.out_number; i++)
      {
       fscanf(in, "%d", &data.out_seq[i][j]);
       if((data.out_seq[i][j]!=0)&&(data.out_seq[i][j]!=1))
       {
        printf("Value of sequence should be either 1 or 0\n");
        printf("Pls. check the data file\n");
        exit(1);
       }/*end if */
      }/*end of for loop*/
    } /*end of for loop*/
  } /*end if*/
  else
  {
   printf("Numberof bit strams and sequence length should be positive\n");
   printf("Pls. check the data file\n");
   exit(1);
  }
 }/*end of while loop*/
 fclose(in);

in=fopen("result_compress.dat", "a");
if(in)
{
 length=data.length;
```

```
    printf("data.out_number is %d\n", data.out_number);
    printf("data.length is %d\n", data.length);
    printf("The input sequences are :\n");
    fprintf(in, "The input sequences are:\n");
    for (i=0; i<data.out_number; i++)
    {
     printf("\n");
     fprintf(in, "\n");
     for (j=0; j<data.length; j++)
     {
      printf("%d", data.out_seq[i][j]);
      fprintf(in, "%d", data.out_seq[i][j]);
     }/*for loop end*/
    }/*for loop end*/
    printf("\n");
    fprintf(in, "\n\n");
  }          /* if end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);
  outnumber=data.out_number;
  read_init_values(init_values, outnumber);
  in=fopen("result_compress.dat", "a");
  if(in)
  {
   fprintf(in, "outline number are: \n");
   for(i=0; i<data.out_number; i++)
   {
    data.init_values[i]=init_values[i];
    printf("outline number is %d\n",data.init_values[i]);
    fprintf(in, "%d\n",data.init_values[i]);
   }
   fprintf(in, "\ndata.out_number is %d\n", data.out_number);
   fprintf(in, "\ndata.length is %d\n\n", data.length);
   fprintf(in, "**************************************\n\n\n");
  }/*if(in) end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

  printf("FUNCTION [getdata_file] END\n");

}/*function getdata_file() end*/



/*****************************************************************
                              main()
                          MAIN FUNCTION
*****************************************************************/

main()
{
 int i, j, rt, number;

 printf("FUNCTION [main]  BEGIN\n");

 do{
 printf("\n\n----------MENU----------\n");
 printf("------------------------\n\n");
 printf("Please enter your choice.\n\n");
 printf("1: Read sequences from a file.\n\n");
 printf("2: Creating the compaction tree\n");
 printf("   with stochastically dependent probability values\n");
 printf("   of multiple line errors using exhaustive approach\n\n");
 printf("0: Exit the program.\n\n");
 scanf("%d",&number);
 switch(number)
 {
  case 1:
```

```
  {
   getdata_file();
   break;
  }         /*case 1 end*/
  case 2:
  {
   in=fopen("merge_result.dat", "a");
   if(in)
   {
    fprintf(in, "\n\n**********Exhaustive Approach**********\n\n");
   }/* if end*/
   else
   printf("Error opening result_compress.dat file\n");
   fclose(in);
   construct_tree();
   in=fopen("merge_result.dat", "a");
   if(in)
   {
    gettime(&inittime, &simtime, &runtime);
    fprintf(in, "\nCPU time\n");
    fprintf(in, "Initialization: %.4f secs.\n", inittime);
    fprintf(in, "Simulation    : %.4f secs.\n", simtime);
    fprintf(in, "Total         : %.4f secs.\n", runtime);
    fprintf(in, "****************************************\n\n\n");
   }        /* if end*/
   else
   printf("Error opening result_compress.dat file\n");
   fclose(in);
   break;
  }/*case 2 end*/
  case 0:
  {
   printf("Exit the program.\n\n");
   break;
  }/*case 0 end*/
  default:
  { printf("You enter the wrong number, please star againg.\n"); }
 }/*switch end*/
}while(number!=0);

printf("FUNCTION [main] END\n");

}/*function main() end*/
```

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada  K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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

/***************************** HISTORY ***********************************

        JTesting_III: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        defin.h
        JTesting_III performs output compression for combinational circuits
        based on sequence characterization and stochastic dependence
        of line errors. Output sequences can be read from a user supplied
        file.

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


/*****************************************************************
                        Defin.h
          Define All Global Variables Used in Jtesting_III
*****************************************************************/

#define MAX_OUT 300                          /*maximum number of primary outputs*/
#define MAX_BITS 250                         /*maximum Length of a sequence*/
#define PTREE_NO 3                           /*input number of xor consist of parity tree*/
int stage;
int new_line;                                /*Line Number Created After compression*/
int length;                                  /*length of the sequence*/
int discard_number;                          /*it will reduce 1 when one is merged*/
int merge_no;
int merge_result[MAX_OUT][MAX_BITS];         /* put the merge result sequence */
int merge_output_line[MAX_OUT];              /*keep the merged output line number*/

FILE *in, *out;                              /*input/output File Pointers*/

float minutes, seconds, inittime, simtime, runtime;
```

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

        This program was written by Jingyi Liang under the supervision of
        Dr. Sunil R. Das, School of Information Technology and Engineering,
        University of Ottawa, Ontario, Canada in 2000.

        This program is released for research use only. This program, or
        any derivative thereof, may not be reproduced or used for any
        commercial product or purpose without the written permission of
        Dr. Das or Jingyi Liang.

        For detailed information, please contact:
        Dr. Sunil R. Das
        School of Information Technology and Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada  K1N 6N5
        Phone No.: (613) 562-5800 ext. 6216
        Fax:       (613) 562-5175
        E-Mail:    das@site.uotttawa.ca

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

/************************** HISTORY ***************************************

        JTesting_III: Version 1.0
        Original: Jingyi Liang, 23/04/2000

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

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

        heuristic.c
        JTesting_III performs output compression for combinational circuits
        based on sequence characterization and stochastic dependence
        of line errors. Output sequences can be read from a user supplied
        file.

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


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
#include <string.h>
#include "defin.h"
#include<malloc.h>
#include<memory.h>


struct OUTDATA                                  /*struct to store the output data*/
{
 int out_number;                                /*Number of primary outputs*/
 int length;                                    /*Length of sequence*/
 int out_seq[MAX_OUT][MAX_BITS];                /*Storage for Output sequences*/
 int init_values[MAX_OUT];                      /*Initial line umber*/
 int discard[MAX_OUT];
};

struct OUTDATA data;

typedef struct
{
 int cover_table[MAX_OUT][MAX_BITS];/*cover_table*/
 int seq_no;                                    /*rower numbers in  cover table*/
}COVER_TABLE;

typedef struct
```

```
{
 int line_number[MAX_OUT];                    /*output line number in best_group*/
 int max_w;                                   /*maxnumber(0/1) matching bits in the best group*/
 int max_n;                                   /*maxnumber sequences in best group*/
}BESTGROUP;

typedef struct
{
 int posi_number[MAX_OUT];                    /*count 1/0 numbers of bit position in 1/0 covertable*/
 int sort_line[MAX_OUT];                      /*after sorting, the name of line*/
 int orig_position[MAX_OUT];                  /*store the original position in the rigmal input*/
 int freorder_no;                             /*sequence number on the frequency ordering*/
}FREQ_ORDER;                                  /*keep some information about  frequency ordering*/

int candidate_seq[MAX_OUT][MAX_BITS];         /*the candidate sequences from  original list*/
int candidate_line[MAX_OUT];                  /*the candidate line name from original list*/
int candidate_max_n;
int A[MAX_OUT];                               /*keep the vertical weights*/


/*****************************************************************
                        gettime()
     FUCNTION FOR GETTING CPU TIME USED IN THE PROGRAM
*****************************************************************/

 void gettime(float *usertime, float *systemtime, float *total)
 {
  struct tms timesbuffer;
  time_t totaltime;
  time_t utime;
  time_t stime;

  printf("FUNCTION [gettime] BEGIN\n");

  times(&timesbuffer);
  utime=timesbuffer.tms_utime;
  stime=timesbuffer.tms_stime;
  totaltime=timesbuffer.tms_utime+timesbuffer.tms_stime; /*In 60th seconds*/
  *usertime=(float)utime/60.0;
  *systemtime=(float)stime/60.0;
  *total=(float)totaltime/60.0;
  printf("inittime is %.3f\n", *usertime);
  printf("simtime is %.3f\n", *systemtime);
  printf("total is %.3f\n", *total);

  printf("FUNCTION [gettime] END\n");

 }/*function gettime() end */


/*****************************************************************
                        copy()
          FUCNTION FOR GETTING TWO SAME SEQUENCES
*****************************************************************/

 copy(int *target, int *source)
 {
  int m;

  printf("FUNCTION [copy] BEGIN\n");

  for (m=0; m<length; m++)
  target[m]=source[m];

  printf("FUNCTION [copy] END\n");

 }/*function copy() end*/
```

```
/*********************************************************************
                        count_number()
          FUCNTION FOR GETTING 1'S/0'S NUMBER IN A SEQUENCE
*********************************************************************/

int count_number(int flag,int *candi_list)
{
 int i, result;
 int count=0;

 printf("FUNCTION [count_number] BEGIN\n");

 if (flag==1)
 {
  for (i=0; i<length; i++)
  { count+=candi_list[i]; }/*for end*/
  result=count;
 }          /* if end*/
 else if (flag==0)
 {
  for (i=0; i<length; i++)
  {
   if (candi_list[i]==0)
   count++;
   else
   count=count;
  }/*for end*/
  result=count;
 }/*else if end*/

 printf("FUNCTION [count_number] END\n");

 return(result);

}/*function count_number() end*/



/*********************************************************************
                          merge_seq()
        FUCNTION FOR MERGERING SELECTED SEQUENCES
                  BY SELECTED LOGIC GATES
*********************************************************************/

merge_seq(int flag_gate)
{
 int i, j, m, u,v,w,W, R;
 int result_seq[MAX_BITS];
 int counter, result;

 printf("FUNCTION [merge_seq] BEGIN\n");

 for (m=0; m<MAX_BITS; m++) /*reset*/
 {
  if(flag_gate==1)
  result_seq[m]=1;
  else if ((flag_gate==2)||(flag_gate==3))
  result_seq[m]=0;
 }

 switch (flag_gate)
 {
  case 1:                                          /*merge by gate AND*/
  {
   for (u=0; u<candidate_max_n; u++)               /*for 1 begin*/
   {
    for (v=0; v<data.out_number; v++)              /*for 2 begin*/
    {
     if ((data.discard[v]!=1)&&(candidate_line[u]==data.init_values[v]))
```

```
    {
     data.discard[v]=1;
     for( w=0; w<data.length; w++)
     {
      result_seq[w]=result_seq[w]&data.out_seq[v][w];
      data.out_seq[v][w]=9;
     } /*for end*/
    }     /*if end*/
   }      /*for 2 end*/
   discard_number--;
  }       /*for 1 end*/

  new_line++;
  merge_output_line[merge_no]=new_line;
  for (j=0; j<length; j++)
  {  merge_result[merge_no][j]=result_seq[j]; }

  in=fopen("merge_result.dat", "a");
  if(in)
  {
   printf(" STAGE %d\n", stage);
   printf("*****************************************\n\n");
   fprintf(in, "%d = AND(", new_line);

   for (i=0; i<candidate_max_n; i++)
   {
    printf("#%d  ", candidate_line[i]);
    fprintf(in, "%d, ", candidate_line[i]);
   }
   fprintf(in, ")\n");
   printf("\nare merged with an AND gate into #%d.\n",  new_line);
  }/*if(in) end*/
  else
  printf("Error opening merge_result.dat file\n");
  fclose(in);
  merge_no++;
  stage++;
  result=1;
  break;
 }/*case 1 end*/

 case 2:                                                    /*merge by gate OR*/
 {
  for (u=0; u<candidate_max_n; u++)                         /*for 1 begin*/
  {
   for (v=0; v<data.out_number; v++)                        /*for 2 begin*/
   {
    if ((data.discard[v]!=1)&&(candidate_line[u]==data.init_values[v]))
    {
     data.discard[v]=1;
     for( w=0; w<data.length; w++)
     {
      result_seq[w]=result_seq[w]|data.out_seq[v][w];
      data.out_seq[v][w]=9;
     } /*for end*/
    }     /*if end*/
   }/*for 2 end*/
   discard_number--;
  }/*for 1 end*/

  new_line++;
  merge_output_line[merge_no]=new_line;
  for (j=0; j<length; j++)
  { merge_result[merge_no][j]=result_seq[j]; }
  in=fopen("merge_result.dat", "a");
  if(in)
  {
   printf(" STAGE %d\n", stage);
   printf("*****************************************\n\n");
   fprintf(in, "%d = OR(", new_line);
```

```
   for (i=0; i<candidate_max_n; i++)
   {
    printf("#%d  ", candidate_line[i]);
    fprintf(in, "%d, ", candidate_line[i]);
   }
   fprintf(in, ")\n");
   printf("\nare merged with an OR gate into #%d.\n", new_line);
  }/*if(in) end*/
  else
  printf("Error opening merge_result.dat file\n");
  fclose(in);

  merge_no++;
  stage++;
  result=1;
  break;
 }/*case 2 end*/

 case 3:                                                    /*merge by gate XOR*/
 {
  in=fopen("result_compress.dat", "a");
  if(in)
  {
   fprintf(in, "***candidate_max_n=%d\n", candidate_max_n);
   fprintf(in, "data.out_number=%d\n",data.out_number);
   for (m=0; m<data.out_number; m++)
   fprintf(in, "candidate_line[%d]=%d\n", m, candidate_line[m]);
  }/*if(in) end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);
 for (u=0; u<candidate_max_n; u++)                          /*for 1 begin*/
  {
   for (v=0; v<data.out_number; v++)                        /*for 2 begin*/
   {
    if ((data.discard[v]!=1)&&(candidate_line[u]==data.init_values[v]))
    {
     data.discard[v]=1;
     for( w=0; w<data.length; w++)
     {
      result_seq[w]=result_seq[w]^data.out_seq[v][w];
      data.out_seq[v][w]=9;
     } /*for end*/
    }     /*if end*/
   }      /*for 2 end*/
   discard_number--;
  }/*for 1 end*/
 new_line++;
 merge_output_line[merge_no]=new_line;
 for (j=0; j<length; j++)
 { merge_result[merge_no][j]=result_seq[j];}

 in=fopen("merge_result.dat", "a");
 if(in)
 {
  printf(" STAGE %d\n", stage);
  printf("****************************************\n\n");
  fprintf(in, "%d = XOR(", new_line);

  for (i=0; i<candidate_max_n; i++)
  {
   printf("#%d  ", candidate_line[i]);
   fprintf(in, "%d, ", candidate_line[i]);
  }
  fprintf(in, ")\n");
  printf("\nare merged with an XOR gate into #%d.\n", new_line);
 }/*if(in) end*/
 else
 printf("Error opening merge_result.dat file\n");
 fclose(in);
```

```
      merge_no++;
      stage++;
      result=1;
      break;
     }/*case 3 end*/
    }/*switch end*/

    in=fopen("result_compress.dat", "a");
    if(in)
    {
     fprintf(in, "\n******** get new merger line # and result ********\n");
     for(i=0; i<merge_no; i++)
     {
      fprintf(in, "%d\n", merge_output_line[i]);
      for (j=0; j<length; j++)
      { fprintf(in, "%d", merge_result[i][j]); }/*for end*/
      fprintf(in, "\n");
     }/*for end*/
     fprintf(in, "\n\n");
    }/*if in end*/
    else
    printf("Error opening result_compress.dat file\n");
    fclose(in);

    printf("FUNCTION [merge_seq] END\n");

   }/*function merge_seq() end*/



/*********************************************************************
                              factorial()
            RECURSIVE FUCNTION FOR CALCULATING
              THE FACTORIAL OF A POSITIVE INTEGER
*********************************************************************/

double factorial(int n)
 {
  double result;
  printf("FUNCTION [factorial]  BEGIN\n");
  if(n==0)
  result=1;
  else
  result= n*factorial(n-1);
  printf("result=%f\n", result);
  printf("FUNCTION [factorial]  END\n");
   return(result);
 }/* function factorial() end*/



/*********************************************************************
                              get_Ti()
                FUCNTION FOR GETTING Ti MISSED ERROR
                  PROBABLITY ESTIMATE OF ith LINE
*********************************************************************/

double get_Ti(int N, int I)
{
  double u=1.0/3.0;
  double v;
  double w;

  printf("FUNCTION [get_ti]  BEGIN\n");

  if(I>=2&&I<=N)
  {
   v=N-I;
   w=((2.0/3.0)*(pow(u,v)));
   printf("w=%f\n", w);
```

```
  }/* if end*/
  else if (I==1)
  {
   v=N-2;
   w=((1.0/3.0)*(pow(u,v)));
   printf("w=%f\n", w);
  }

  printf("FUNCTION [get_ti]  END\n");

  return(w);

 }/* function get_Ti() end*/



/*********************************************************************
                          get_Dand()
                FUCNTION FOR GETTING THE MISSED ERROR
                  PROBABLITY ESTIMATE USING GATE AND
*********************************************************************/

double get_Dand(int max_n)
{
 int i, j,m;
 double D_and;                          /* probability*/
 double C, T;
 double sum_A=0.0;                      /* sum of the vertical weights A0 to A(n-1)*/
 double sum_B=0.0;                      /* sum of Ti*A[N-i]/C*/

 printf("FUNCTION [get_Dand] BEGIN\n");

 for (i=0; i<max_n; i++)
 sum_A+=A[i];

 for (i=1; i<=max_n; i++)
 {
  j=max_n-i;
  C=factorial(max_n)/(factorial(max_n-i)*factorial(i));
  T=get_Ti(max_n, i);
  sum_B+=T*A[j]/C;
 }/*for end*/
 D_and=(1.0*sum_A/data.length)-(sum_B/data.length);

 printf("FUNCTION [get_Dand] END\n");

 return(D_and);

}/*function get_Dand() end*/



/*********************************************************************
                          get_Dor()
                FUCNTION FOR GETTING THE MISSED ERROR
                  PROBABLITY ESTIMATE USING GATE OR
*********************************************************************/

double get_Dor(int max_n)
{
 int i, j,m;
 double D_or;                           /* probability*/
 double C, T;
 double sum_A=0.0;                      /* sum of the vertical weights A1 to A(n)*/
 double sum_B=0.0;                      /* sum of Ti*A[i]/C*/

 printf("FUNCTION [get_Dor] BEGIN\n");

 for (i=1; i<=max_n; i++)
 sum_A+=A[i];
```

```
  for (i=1; i<=max_n; i++)
  {
   j=i;
   C=factorial(max_n)/(factorial(max_n-i)*factorial(i));
    T=get_Ti(max_n, i);
    sum_B+=T*A[j]/C;
  }/*for end*/
  D_or=(1.0*sum_A/data.length)-(sum_B/data.length);

  printf("FUNCTION [get_Dor] END\n");

  return(D_or);

}/*function get_Dor() end*/



/*********************************************************************
                          get_Dxor()
                  FUCNTION FOR GETTING MISSED ERROR
                  PROBABLITY ESTIMATE USING GATE XOR
*********************************************************************/

double get_Dxor(int max_n)
{
 int i, j,m;
 double D_xor=0.0;                                    /* probability*/

 printf("FUNCTION [get_Dxor] BEGIN\n");

 m=max_n%2;
 if (m==1)
 {
  for (i=2; i<=max_n-1; i=i+2)
  D_xor+=get_Ti(max_n, i);
 }
 else if (m==0)
 {
  for (i=2; i<=max_n; i=i+2)
  D_xor+=get_Ti(max_n, i);
 }

 printf("FUNCTION [get_Dxor] END\n");

 return(D_xor);

}/*function get_Dxor() end*/



/*********************************************************************
                          get_candi_seq()
          FUCNTION FOR  GETTING THE CANDIDATE SEQUENCES
                    FROM THE ORIGINAL LIST
*********************************************************************/

get_candi_seq(int max_n, int *line_number)
{
 int i, j, m, n, u, v, w;
 int h=0;

 printf("FUNCTION [get_candi_seq] BEGIN\n");

 for (m=0; m < MAX_OUT; m++)   /*reset*/
 {
  for(n=0; n<MAX_BITS; n++)
  { candidate_seq[m][n]=0; }
 }          /*for end*/

 for (u=0; u<max_n; u++)        /*for 1 begin*/
```

```
 {
  for (v=0; v<data.out_number; v++)          /*for 2 begin*/
  {
   if (line_number[u]==data.init_values[v])
   {
    for( w=0; w<data.length; w++)
    { candidate_seq[h][w]=data.out_seq[v][w]; } /*for end*/
    h++;
   }/*if end*/
  }/*for 2 end*/
 }/*for 1 end*/

 printf("FUNCTION [get_candi_seq] END\n");

}/*function get_candi_seq() end*/



/*********************************************************************
                        vertical_weight()
          FUCNTION FOR GETTING VERTICAL WEITHTS
*********************************************************************/

vertical_weight(int max_n)
{
 int i, j,m, n, u, v;
 int count;

 printf("FUNCTION [vertical_weight] BEGIN\n");

 for(u=0; u<MAX_OUT; u++)
 {A[u]=0;}
  for (i=0; i<data.length; i++) /*for 1 begin*/
 {
  count=0;
  for (j=0; j<max_n; j++)
  { count+=candidate_seq[j][i]; }
  for(v=0; v<=max_n; v++)
  {
   if (count==v)
   {
    A[v]++;
    break;
   }/*if end*/
  }/*for end*/
 } /*for 1 end*/
 in=fopen("result_compress.dat", "a");
 if(in)
 {
  for(m=0; m<=max_n; m++)
  fprintf(in, "A[%d]=%d\n", m, A[m]);
 }/*if(in) end*/
 else
 printf("Error opening result_compress.dat file\n");
 fclose(in);
 printf("FUNCTION [vertical_weight] END\n");

}/*function vertical_weight() end*/



/*********************************************************************
                            intcmp()
                    FUCNTION FOR COMPARING
*********************************************************************/

int intcmp(const void *v1, const void *v2)
{   return(*(int*)v2-*(int*)v1);  }/*function intcmp() end*/
```

```
/*****************************************************************
                        frequency_ordering()
          FUCNTION FOR GETTING FREQUENCY ORDERING
              WITHIN THE ONE/ZERO COVER TABLE
*****************************************************************/

FREQ_ORDER frequency_ordering(int flag, COVER_TABLE ct)
{
 int i, j,m;
 int h=0;
 int temper[MAX_OUT];
 int line[MAX_OUT];
 int position[MAX_OUT];                          /*stor the original position in the original input*/

 FREQ_ORDER fo;

 printf("FUNCTION [frequency_ordering] BEGIN\n");

 for(i=0; i<MAX_OUT; i++)     /*initial the value*/
 {
  fo.posi_number[i]=0;
  fo.sort_line[i]=0;
  fo.orig_position[i]=0;
  fo.freorder_no=0;
  temper[i]=0;
  line[i]=0;
  position[i]=0;
 }

/******** count the bit position ********/

 for(m=0;m<data.out_number; m++)
 {
  for(j=0; j<length; j++)
  {
   for(i=0; i<ct.seq_no; i++)
   {
    if(ct.cover_table[i][j]==data.init_values[m])
    { fo.posi_number[h]++; }
   }/*for end*/
  }/*for end*/

  if(fo.posi_number[h]!=0)
  {
   line[h]=data.init_values[m];
   position[h]=m;
   h++;
  }
 }/*for end*/
 fo.freorder_no=h;

 for (i=0; i<data.out_number; i++)
 { temper[i]=fo.posi_number[i]; }

/******** sort-frequency ordering ********/

qsort(fo.posi_number, data.out_number, sizeof(fo.posi_number[0]), intcmp);

 /******** get the relative original line number ********/

for (i=0; i<data.out_number; i++)
{
 for(j=0; j<data.out_number; j++)
 {
  if((fo.posi_number[i]==temper[j])&&(temper[j] !=999))
  {
   fo.sort_line[i]=line[j];
   fo.orig_position[i]=position[j];
   temper[j]=999;
```

```
    break;
   }        /*if end*/
  }         /*foe end*/
 }/*for end*/

 printf("FUNCTION [frequency_ordering] END\n");

 return(fo);

}/*function frequency_ordering() end*/



/*****************************************************************
                        get_maxgroup()
                  FUCNTION FOR GETTING MAXGROUP
                  WITHIN THE ONE/ZERO COVER TABLE
*****************************************************************/

BESTGROUP get_maxgroup(int flag, COVER_TABLE ct)
{
 int h, k, i, j, jj, m, u, v, p1, W1, N1,N;
 int t=0;
 int cs=0;
 int count, max_bit, max_seq;
 int temp[MAX_BITS], oper[MAX_BITS];

 FREQ_ORDER fo;
 BESTGROUP  better_group, best_group;


 /******** initial the values ********/

 best_group.max_w=0;
 best_group.max_n=0;
 for(i=0; i<MAX_OUT; i++)
 {
  better_group.line_number[i]=0;
  best_group.line_number[i]=0;
  for (j=0; j<MAX_BITS; j++)
  {
   temp[j]=0;
   oper[j]=0;
  }
 }

 printf("FUNCTION [get_maxgroup] BEGIN\n");

 fo=frequency_ordering(flag, ct);                    /*get frequency ordering*/

 for( i=0; i<fo.freorder_no; i++) /* for 1 begin*/
 {
  max_bit=0;
  max_seq=1;
  h=0;
  better_group.max_w=0;
  better_group.max_n=0;

  for(u=0; u<MAX_OUT; u++)
  {
   better_group.line_number[u]=0;
   for (v=0; v<MAX_BITS; v++)
   {
    temp[v]=0;
    oper[v]=0;
   }/*for end*/
  }         /*for end*/
  better_group.line_number[h]=data.init_values[fo.orig_position[i]];

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

```
  { oper[m]=data.out_seq[fo.orig_position[i]][m]; }/*for end*/

  for(j=i+1; j<fo.freorder_no; j++)
  {
   copy(temp, oper);
   if (flag==1)
   {
    for(m=0; m<length; m++)
    {temp[m]=temp[m]&data.out_seq[fo.orig_position[j]][m];}
   }/*if end*/

   else if (flag==0)
   {
    for(m=0; m<length;m++)
    {temp[m]=temp[m]|data.out_seq[fo.orig_position[j]][m];}
   }/*end else for*/

   count=count_number(flag, temp);

   if (count>=max_bit)
   {
    h++;
    copy(oper, temp);
    better_group.line_number[h]=data.init_values[fo.orig_position[j]];
    max_seq++;
    max_bit=count;
   }/* if end*/
  }/* for j end*/

  better_group.max_n=max_seq;
  better_group.max_w=max_bit;

  if
(((better_group.max_w>best_group.max_w)&&(better_group.max_n>=2))||((better_group.max_w==best_group.max_w)&&(better_group.max_n>b
est_group.max_n)&&(better_group.max_n>=2)))
  {
    best_group.max_w=better_group.max_w;
    best_group.max_n=better_group.max_n;
    for (m=0; m<best_group.max_n; m++)
    { best_group.line_number[m]=better_group.line_number[m]; }/*for end*/

    for(m=best_group.max_n; m<MAX_OUT; m++)
    best_group.line_number[m]=0;
   }/* if end*/
  }/* for 1 end*/

 for( i=0; i<fo.freorder_no; i++)   /* for 2 begin*/
 {
  max_bit=0;
  max_seq=1;
  h=0;
  better_group.max_w=0;
  better_group.max_n=0;
  for(u=0; u<MAX_OUT; u++)
  {
   better_group.line_number[u]=0;
   for (v=0; v<MAX_BITS; v++)
   {
    temp[v]=0;
    oper[v]=0;
   }/*for end*/
  }/*for end*/
  better_group.line_number[h]=data.init_values[fo.orig_position[i]];

  for(m=0; m<length; m++)
  { oper[m]=data.out_seq[fo.orig_position[i]][m]; }/*for end*/

  for(j=0; j<fo.freorder_no; j++)
  {
   if (j!=i)
```

```
    {
     copy(temp, oper);
     if (flag==1)
     {
       for(m=0; m<length; m++)
       {temp[m]=temp[m]&data.out_seq[fo.orig_position[j]][m];}
     }       /*if end*/

     else if (flag==0)
     {
       for(m=0; m<length;m++)
       {temp[m]=temp[m]|data.out_seq[fo.orig_position[j]][m];}
     }       /*end else for*/

     count=count_number(flag, temp);

     if (count>=max_bit)
     {
       h++;
       copy(oper, temp);
       better_group.line_number[h]=data.init_values[fo.orig_position[j]];
       max_seq++;
       max_bit=count;
     }/* if end*/
    }/* if i!=j end*/
   }/* for j end*/

   better_group.max_n=max_seq;
   better_group.max_w=max_bit;

   if
(((better_group.max_w>best_group.max_w)&&(better_group.max_n>=2))||((better_group.max_w==best_group.max_w)&&(better_group.max_n>b
est_group.max_n)&&(better_group.max_n>=2)))
   {
     best_group.max_w=better_group.max_w;
     best_group.max_n=better_group.max_n;
     for (m=0; m<best_group.max_n; m++)
     { best_group.line_number[m]=better_group.line_number[m]; }/*for end*/

     for(m=best_group.max_n; m<MAX_OUT; m++)
     best_group.line_number[m]=0;
    }/* if end*/
  }/* for 2 end*/

 printf(" FUNCTION [get_maxgroup] END\n\n");

 return (best_group);

}/*function get_maxgroup() end*/



/*****************************************************************
                        form_covertable()
    FUCNTION FOR FORMING 1 COVER TABLE AND 0 COVER TABLE
*****************************************************************/

COVER_TABLE form_covertable(int flag)
{
 int i, j, m, u;
 int v=0;
 int temp[MAX_OUT];

 COVER_TABLE ct;

 ct.seq_no=0;

 printf("FUNCTION [form_covertable] BEGIN\n");
```

```
/******** initial the 1/0 cover table ********/

for (i=0; i<MAX_OUT; i++)
{
 temp[i]=0;
 for (j=0; j<length; j++)
 { ct.cover_table[i][j]=0; }/* for 2 end*/
}/*for 1 end*/


/******** form the 1 covertable ********/

if(flag==1)
{
 for (i=0; i<data.out_number; i++)/*for 1 begin*/
 {
  if(data.discard[i]!=1)
  {
   for (j=0; j<length; j++)
   {
    if(data.out_seq[i][j]==1)
    {
     for (u=0; u<data.out_number; u++)
     {
      if(ct.cover_table[u][j]==0)
      {
       ct.cover_table[u][j]=data.init_values[i];
       break;
      }
     }/*for 3 end*/
    }/*if end*/
   }/*for 2 end*/
  }/*if end*/

  else
  {printf(" this sequence is discarded!!\n\n");}
 }/*for 1 end*/
}/*if end*/


/******** form the 0 covertable ********/

if(flag==0)
{
 for (i=0; i<data.out_number; i++)/*for 1 begin*/
 {
  if(data.discard[i]!=1)
  {
   for (j=0; j<length; j++)
   {
    if(data.out_seq[i][j]==0)
    {
     for (u=0; u<data.out_number; u++)
     {
      if(ct.cover_table[u][j]==0)
      {
       ct.cover_table[u][j]=data.init_values[i];
       break;
      }
     }/*for 3 end*/
    }/*if end*/
   }/*for 2 end*/
  }/*if end*/
  else
  {printf(" this sequence is discarded!!\n\n");}
 }/*for 1 end*/
} /*if end*/

/******** find the size of cover-table ********/
```

```
 for (j=0; j<length; j++)
 {
  for (i=0; i<data.out_number+1; i++)
  {
   if(ct.cover_table[i][j]==0)
   {
    temp[v]=i;
    break;
   }
  }
  v++;
 }          /*for end*/

 for(i=0; i<length; i++)
 {
  if (ct.seq_no<temp[i])
  ct.seq_no=temp[i];
 }/*for end*/


 printf("FUNCTION [form_covertable] END\n");

 return(ct);

}/*function form_covertable() end*/



/*****************************************************************
                        construct_tree()
        FUCNTION FOR MERGERING MULTILINE OUTPUTS
    BY 1/0 - COVER TABLE ASSUMING STOCHATIC DEPENDENCE
                    OF MULTIPLE LINE ERRORS
*****************************************************************/

construct_tree()
{
 int d, i, j, m, n, u, v, w;
 int W_n1, W_n0;
 int temper_number;
 int flag_gate;
 float R, N, RN, reduce_gain;
 double D_and, D_or, D_xor;                    /* missing detect probablity*/
 int old_outnumber;
 COVER_TABLE one_ct, zero_ct;
 BESTGROUP one_maxgroup, zero_maxgroup;
 BESTGROUP remain;                             /* after merged by gate AND&OR, the  remaining sequences*/

 new_line=data.init_values[data.out_number-1];

 printf("FUNCTION [construct_tree] BEGIN\n");

 do
 {                                             /*first do begin*/

  /******** level one ********/
  discard_number=data.out_number;              /* when one line merged this number*/

  for (u=0; u<MAX_OUT; u++)                    /* reset*/
  candidate_line[u]=0;
  candidate_max_n=0;


  /******** select the sequences merged by AND ********/

  do  /*do and begin*/
  {
   flag_gate=0;  /*reset flag_gate*/
   one_ct=form_covertable(1);                  /* get 1 cover table*/
   one_maxgroup=get_maxgroup(1,one_ct);
```

```
  in=fopen("result_compress.dat", "a");
  if(in)
  {
   candidate_max_n=one_maxgroup.max_n;
   for (u=0; u<candidate_max_n; u++)
   {
    candidate_line[u]=one_maxgroup.line_number[u];
          fprintf(in, "candidate_line[%d] is %d\n", u,candidate_line[u]);
   }/*for end*/
  }/* if(in) end*/

  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

  get_candi_seq(candidate_max_n,  candidate_line);
  vertical_weight(candidate_max_n);
  W_n1=A[candidate_max_n];
  W_n0=A[0];
  N=candidate_max_n;
  R=data.length-W_n1-W_n0;
  RN=R/N;

 if (((W_n1-W_n0)>=(RN))||((W_n0-W_n1)<=(-RN)))          /*if 1 begin*/
 {
  D_and=get_Dand(candidate_max_n);
  D_xor=get_Dxor(candidate_max_n);

  if (D_and<=D_xor)          /*if 2 begin*/
  {
   flag_gate=1;
   merge_seq(flag_gate);
  }       /*if 2 end*/

  else if (D_and>D_xor)      /*else if 2 begin*/
  {
   flag_gate=3;
   merge_seq(flag_gate);
  }/*else if 2 end*/
 }/*if 1 end*/

}while((one_maxgroup.max_n>=2)&&(flag_gate!=0)&&(discard_number>1));
/*do and end*/

/******** select the sequences merged by OR ********/

 do  /*do or begin*/
 {
  if (discard_number>=2) /*if left more than 2 line to merge by OR*/
  {
   flag_gate=0;  /*reset flag_gate*/
   zero_ct=form_covertable(0);                          /*get 0 cover table*/
   zero_maxgroup=get_maxgroup(0,zero_ct);

   in=fopen("result_compress.dat", "a");
   if(in)
   {
    candidate_max_n=zero_maxgroup.max_n;

    for (u=0; u<candidate_max_n; u++)
    {

     candidate_line[u]=zero_maxgroup.line_number[u];
     fprintf(in, "candidate_line[%d] is %d\n", u,candidate_line[u]);
    }/*for end*/
   }/* if(in) end*/

   else
   printf("Error opening result_compress.dat file\n");
```

```
    fclose(in);

    get_candi_seq(candidate_max_n,  candidate_line);
    vertical_weight(candidate_max_n);
    W_n1=A[candidate_max_n];
    W_n0=A[0];
    N=candidate_max_n;
    R=data.length-W_n1-W_n0;
    RN=R/N;

    if (((W_n0-W_n1)>=(RN))||((W_n1-W_n0)<=(-RN)))/*if 1 begin*/
    {
     D_or=get_Dor(candidate_max_n);
     D_xor=get_Dxor(candidate_max_n);

     if (D_or<=D_xor)/*if 2 begin*/
     {
      flag_gate=2;
      merge_seq(flag_gate);
     }/*if 2 end*/

     else if (D_or>D_xor)/*else if 2 begin*/
     {
      flag_gate=3;
      merge_seq(flag_gate);
     }/*else if 2 end*/
    }/*if 1 end*/
  } /* if left more than 2 line to merge by OR end*/
}while((one_maxgroup.max_n>=2)&&(flag_gate!=0)&&(discard_number>1));
/*do and end*/

/********  merge the remaining sequences by AND/OR/EXOR  ********/

if (discard_number>=2)  /* if left more than 2 line to merge by EXOR*/
{
  v=0;

  in=fopen("result_compress.dat", "a");
  if(in)
  {
   candidate_max_n=discard_number;
   fprintf(in, "remain.max_n=%d\n", remain.max_n);

   for(u=0; u<data.out_number; u++)
   {
    if(data.discard[u]!=1)
    {
     candidate_line[v]=data.init_values[u];
     fprintf(in, "candidate_line[%d]=%d\n", v, candidate_line[v]);
     v++;
    } /*if end*/

    fprintf(in, "v=%d\n", v);
   } /*for end*/

   for (m=0; m<data.out_number; m++)
   fprintf(in, "candidate_line[%d]=%d\n", m, candidate_line[m]);

  }/*if(in) end*/

  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

  get_candi_seq(candidate_max_n, candidate_line);
  vertical_weight(candidate_max_n);

  D_and=get_Dand(candidate_max_n);
  D_or=get_Dor(candidate_max_n);
  D_xor=get_Dxor(candidate_max_n);
```

```
   if ((D_and<=D_or)&&(D_and<=D_xor))
   {
    flag_gate=1;
    merge_seq(flag_gate);
   }/*if end*/

   if ((D_or<=D_and)&&(D_or<=D_xor))
   {
    flag_gate=2;
    merge_seq(flag_gate);
   }/*if end*/

   if ((D_xor<=D_and)&&(D_xor<=D_or))
   {
    flag_gate=3;
    merge_seq(flag_gate);
   }/*if end*/
 }/*if left more than 2 line to merge by AND/OR/EXOR end*/


/******** setup new input by the left line and merger output ********/

d=0;                                          /* d is the new out put numbers*/
for(i=0; i<data.out_number; i++)              /* install the remaining sequence in  the new data list*/
{
 if(data.discard[i]!=1)
 {
  data.init_values[d]=data.init_values[i];
  data.discard[d]=0;
  for(j=0; j<length;j++)
  {
   data.out_seq[d][j]=data.out_seq[i][j];
  }       /*for end*/
  d++;
 }/*if end*/
}/*for end*/

for(i=0; i<merge_no; i++)                     /*install new outputs of gates in the new data list*/
{
 data.init_values[d]=merge_output_line[i];
 data.discard[d]=0;
 for(j=0; j<length;j++)
 {
  data.out_seq[d][j]=merge_result[i][j];
 }/*for end*/
 d++;
}/*for end*/

old_outnumber=data.out_number;
data.out_number=d;
for(i=data.out_number; i<old_outnumber; i++)
/*set the left of the data list as zero*/
{
 data.init_values[i]=0;
 data.discard[i]=0;
 for(j=0; j<length;j++)
 { data.out_seq[i][j]=0; }/*for end*/
}          /*for end*/

for (i=0; i<merge_no; i++)  /*reset */
{
 merge_output_line[i]=0;
 for (j=0; j<length; j++)
 { merge_result[i][j]=0; }
}/*for end*/

merge_no=0;

in=fopen("result_compress.dat", "a");
```

```
  if(in)
  {
   fprintf(in, "------new output is------\n\n");
   fprintf(in, "data_out_number is %d\n", data_out_number);
   fprintf(in, "data_length is %d\n", data_length);
   fprintf(in, "The input sequences are :\n");
   for (i=0; i<data_out_number; i++)
   {
    fprintf(in, "\n");
    for (j=0; j<data_length; j++)
    {
     fprintf(in, "%d", data_out_seq[i][j]);
    }          /*for loop end*/
   }          /*for loop end*/
   fprintf(in, "\n");
   fprintf(in, "outline number are: \n");
   for(i=0; i<data_out_number; i++)
   {
    fprintf(in, "outline number is %d\t",data_init_values[i]);
    fprintf(in, "discard[%d]=%d\n", i, data_discard[i]);
   }          /*for end*/
  }          /*if (in) end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);
 }while(data_out_number>1);                                    /*first do end*/

 printf("FUNCTION [construct_tree] END\n");

}/*function construct_tree() end*/



/*****************************************************************
                        read_init_values()
     FUCNTION FOR GETTING THE INITIAL OUTPUT VALUES
    DEFINED IN THE ISCAS CIRCUIT DESCTRIPTION FROM A FILE
*****************************************************************/

 read_init_values(int init_values[MAX_OUT], int outnumber)
 {
  int out_number;
  register i;
  out_number=outnumber;

  printf("FUNCTION [read_init_values] BEGIN\n");

  in=fopen("outline_num.dat", "r");
  if(in)
  {
   while(!feof(in))
   {
    for(i=0; i<out_number; i++)
    {
     fscanf(in, "%d\n", &init_values[i]);
     if (init_values[i]>=0)
     { init_values[i]=init_values[i]; }
     else
     {
      printf("The initial line number should be positive\n");
      printf("Check your data file\n");
      exit(1);
     }
    }/* for end*/
   }/*while end*/
  } /*if end*/
  else
  printf("Error opening out_line_numb.dat file\n");
  fclose(in);
```

```
  printf("FUNCTION [read_init_values] END\n");

} /*function read_init_values() end*/


/*********************************************************************
                              getdata_file()
          FUCNTION FOR GETTING DATA FROM A FILE
*********************************************************************/

getdata_file()
{
 int outnumber;
 int init_values[MAX_OUT];
 register i, j;

 printf("FUNCTION [getdata_file] BEGIN\n");

 in=fopen("result_compress.dat","a");
 if(in)
 {
  fprintf(in, "\n\n\n    The data read from file as following:\n");
  fprintf(in, "*************************************************\n\n\n");
 }
 fclose(in);
 in = fopen("out_seq.dat", "r");
 while(!feof(in))
 {
  fscanf(in, "%d\n", &data.out_number);
  fscanf(in, "%d\n", &data.length);
  length=data.length;
  if ((data.out_number>0)&&(data.length>0))
  {
   for (j=0; j<data.length; j++)
    {
     for (i=0; i<data.out_number; i++)
     {
      fscanf(in, "%d", &data.out_seq[i][j]);
      if((data.out_seq[i][j]!=0)&&(data.out_seq[i][j]!=1))
      {
       printf("Value of sequence should be either 1 or 0\n");
       printf("Pls. check the data file\n");
       exit(1);
      }/*end if */
     }/*end of for loop*/
    } /*end of for loop*/
  }/*end if*/
  else
  {
   printf("Number of bit streams and sequence length should be positive\n");
   printf("Pls. check the data file\n");
   exit(1);
  }
}          /*end of while loop*/
fclose(in);

in=fopen("result_compress.dat", "a");
if(in)
{
 length=data.length;
 fprintf(in, "The input sequences are:\n");
 for (i=0; i<data.out_number; i++)
 {
  fprintf(in, "\n");
  for (j=0; j<data.length; j++)
  { fprintf(in, "%d", data.out_seq[i][j]); }/*for loop end*/
 }         /*for loop end*/
 fprintf(in, "\n\n");
}/* if end*/
```

```
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);
  outnumber=data.out_number;
  read_init_values(init_values, outnumber);
   in=fopen("result_compress.dat", "a");
  if(in)
  {
    fprintf(in, "outline number are: \n");
    for(i=0; i<data.out_number; i++)
    {
      data.init_values[i]=init_values[i];
      fprintf(in, "%d\n",data.init_values[i]);
    }
    fprintf(in, "\ndata.out_number is %d\n", data.out_number);
    fprintf(in, "\ndata.length is %d\n\n", data.length);
    fprintf(in, "*****************************************\n\n\n");
  }          /*if(in) end*/
  else
  printf("Error opening result_compress.dat file\n");
  fclose(in);

  printf("FUNCTION [getdata_file] END\n");

}/*function getdata_file() end*/



/*********************************************************************
                                    main ()
                                MAIN FUNCTION
*********************************************************************/

main()
{
 int i, j, rt, number;

 printf("FUNCTION [main]  BEGIN\n");

 do{
 printf("\n\n--------MENU-------\n");
 printf("-------------------\n\n");
 printf("Please enter your choice.\n\n");
 printf("1: Read sequences from a file.\n\n");
 printf("2: Creating the compaction tree\n");
 printf("   with stochastically dependent probability values\n");
 printf("   of multiple line errors using heuristic approach\n\n");
 printf("0: Exit the program.\n\n");
 scanf("%d",&number);

 switch(number)
 {
  case 1:
  {
    getdata_file();
    break;
  }/*case 1 end*/

  case 2:
  {
    in=fopen("merge_result.dat", "a");
    if(in)
    {  fprintf(in, "\n\n**********Heuristic Approach**********\n\n");  }/* if end*/
    else
    printf("Error opening result_compress.dat file\n");
    fclose(in);

    construct_tree();
    in=fopen("merge_result.dat", "a");
    if(in)
```

```
    {
     gettime(&inittime, &simtime, &runtime);
     fprintf(in, "\nCPU time\n");
     fprintf(in, "Initialization: %.4f secs.\n", inittime);
     fprintf(in, "Simulation   : %.4f secs.\n", simtime);
     fprintf(in, "Total        : %.4f secs.\n", runtime);
     fprintf(in, "*************************************************\n\n\n");
    }/* if end*/
    else
    printf("Error opening result_compress.dat file\n");
    fclose(in);
    break;
   }/*case 2 end*/

   case 0:
   {
    printf("Exit the program.\n\n");
    break;
   }/*case 0 end*/

   default:
   {   printf("You enter the wrong number, please star againg.\n");   }/*default end*/
  }/*switch end*/
}while(number!=0);

printf("FUNCTION [main] END\n");

}/*function main() end*/
```