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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa • University of Ottawa

GENERALIZED MERGEABILITY IN SPACE COMPRESSION USING NONEXHAUSTIVE TEST PATTERNS FOR BUILT-IN SELF-TESTING OF VLSI CIRCUITS: MATHEMATICAL ANALYSIS AND SIMULATION RESULTS

By

Toni Barakat, B.A.Sc., B.Sc.

A dissertation submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

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

Faculty of Engineering
University of Ottawa

December 1997

©1997, Toni Barakat, Ottawa, Canada

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

0-612-32525-3

Canadä

ABSTRACT

With continued growth in electronics industry with more and more emphasis on complex systems and increasing levels in the densities of integration, the need for better and more effective testing strategies ensuring reliable operations of chips are constantly being felt. A variety of techniques have been developed over the years to address this complicated problem of testing VLSI chips. Built-in self-testing (BIST) that includes additional circuitry built into the chip itself to have it tested whenever any failure occurs has proved to be quite attractive to the circuit designers and very popular as well.

This thesis deals with response compaction techniques of BIST of VLSI circuits which translates into a process of reducing the test response of the circuit under test (CUT) to a signature. Instead of comparing bit by bit the fault free responses to the observed outputs of the CUT as in conventional testing, here the observed signature is compared to the correct one, thereby reducing the storage requirements for the correct CUT responses. The thesis specifically deals with designing efficient space compaction techniques for BIST of VLSI circuits using nonexhaustive test sets. The developed techniques utilize the concepts of Hamming distance, sequence weights (generalized), and failure probabilities with multiplicities of errors at the CUT output in selecting specific gates for merger of a number of outputs streams from the CUT using a generalized mergeability criteria. The criteria are developed under conditions of both stochastic independence and stochastic dependence of line errors.

Design algorithms are proposed in the thesis and the methods of implementation were demonstrated with many examples. Specifically, extensive simulation runs on ISCAS 85 benchmark circuits with FSIM, ATALANTA and COMPACTEST confirms the effectiveness of the suggested approaches under conditions of stochastic independence and dependence of multiple line errors. The techniques guarantee simple designs with reasonably high fault coverage for single stuck-line errors with low CPU simulation time and acceptable area overhead. A performance comparison of the designed space compactors with conventional linear parity tree compactors are also given.

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation and gratitude to my thesis advisor, Dr. Sunil R. Das, Professor in the Department of Electrical and Computer Engineering at the University of Ottawa, for his guidance and support throughout this research.

I would also like to thank Dr. Emil Petriu Director of the Department of Electrical and Computer Engineering, University of Ottawa, and Dr. Amiya R. Nayak, School of Computer Science, Carleton University, Ottawa. Thanks are also due to Dr. Dong S. Ha of the Department of Electrical Engineering at the Virginia Polytechnic Institute and State University, Virginia, U.S.A. for kindly providing me with ATALANTA and FSIM fault simulators, as well as Dr. S. M. Reddy and Dr. I. Pomeranz, Faculty of Engineering, University of Iowa, Iowa, U.S.A. for providing me with COMPACTEST which enabled me to conduct this research.

I am also grateful to my co-workers at **Lucent Technologies (Bell Labs Innovations)** for their help and support.

The financial support from the Natural Sciences and Engineering Research Council of Canada under Grant A 4750 is also gratefully acknowledged.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGMENTS	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	ix
LIST OF TABLES	xi
CHAPTER 1 Introduction	1
1.1 Evolution Perspective	2
1.2 Basic Testing Concepts	2
1.3 Problems of Testing in VLSI Circuits	4
1.4 Fault Models	5
1.5 Testability of a Digital System	6
1.5.1 Quality of Testing and Fault Simulation	6
1.6 Test Pattern Generation for Combinational Circuits	8
1.6.1 Manual Pattern Generation	8
1.6.2 Algorithmic Pattern Generation	9
1.6.2.1 Single Path Sensitization	9
1.6.2.2 D-algorithm	9
1.6.2.3 PODEM, FAN and CONT	10
1.6.2.4 Boolean Difference	11
1.6.3 Pseudorandom Pattern Generation	11
1.7 Design for Testability and BIST	12
1.8 Overall Testing Scheme	13

1.8.1 Response Analysis 14
1.9 Thesis Outline 16

CHAPTER 2 Overview of Test Compaction Techniques 17
2.1 Input Compaction 17
2.2 Output Compaction 18
2.3 Time Compaction 18
2.3.1 Ones Count Compaction 18
2.3.2 Syndrome Testing 19
2.3.3 Transition Count Compaction 20
2.3.4 Double and Multiple Transition Counting 20
2.3.5 Parity Check Compaction 22
2.3.6 Cyclic Code Compaction 22
2.3.7 Signature Analysis 23
2.3.8 Compaction Using Walsh Spectra 25
2.3.9 Parallel Compaction Analysis 26
2.4 Space Compaction Techniques 27
2.4.1 Parity Tree Compaction 27
2.4.2 Hybrid Space Compression 28
2.4.3 Dynamic Space Compression 29
2.4.4 Modified Dynamic Space Compression 30
2.4.5 Modified Hybrid Space Compaction 31
2.4.6 Quadratic Functions Compaction 32
2.4.7 Programmable Space Compaction 33
2.4.8 Multiplexed Parity Trees 34

CHAPTER 3 Designing of Space Compaction Trees for Multi-Output Circuits in BIST Based on Stochastic Independence of Multiple Line Errors 35
3.1 First Order Weights 35

3.1.1 First-Order 1-Weight 35
3.1.2 First-Order 0-Weight 36
3.2 Nth Order 1-Weight 36
3.3 Nth Order 0-Weight 37
3.4 Derived Sequences 38
3.4.1 Second-Order Derived Sequences 38
3.4.2 Nth-Order Derived Sequences 40
3.5 Implementation 50
3.6 The First Algorithm 51
3.7 Explanation of the First Algorithm 54
3.8 Implementation of the Selection of the Best Group Sub-Algorithm 59

CHAPTER 4 Designing of Space Compaction Trees for Multi-Output Circuits in BIST Based on Stochastic Dependence of Multiple Line Errors 63
4.1 Mergeability Criteria 63
4.2 Derivation of the Nth-Order Detectable Error Probability Estimate for Individual Gates 64
4.3 Generalized Mergeability Criteria 68
4.4 Implementation 69
4.5 The Second Algorithm 69

CHAPTER 5 Experimental Results and Discussions 87
5.1 Simulation Method 87
5.2 Classification of the Simulated Results 88
5.3 Simulation Results 88
5.3.1 Simulation Results Without Using Compactors 88
5.3.2 Simulation Results by Assuming Stochastic Independence 90
5.3.3 Simulation Results by Assuming Stochastic Dependence 92
5.3.4 Simulation Results by Using Parity Tree as a Space Compactor 94

5.4 Hardware Overhead 96

5.5 Compaction Circuits for C432 98

5.6 Graphical Representation of the Simulation Results 99

5.6.1 Comparison of the Fault Coverage of All Circuits in the Cases of Parity Tree, Stochastic Independence and Dependence of Line Errors 100

5.6.2 Graphical Representation of All Simulated Circuits Assuming Stochastic Independence of Line Errors 101

5.6.3 Graphical Representation of All Simulated Circuits Assuming Stochastic Dependence of Line Errors 103

CHAPTER 6 Concluding Remarks 105

PUBLICATIONS

Papers Published by the Author of this Thesis 107

BIBLIOGRAPHY 108

APPENDICES

APPENDIX A C Program to Construct the Space Compaction Trees for Combinational Logic Circuits Assuming Stochastic Independence of Line Errors 115

APPENDIX B C Program to Construct the Space Compaction Trees for Combinational Logic Circuits Assuming Stochastic Independence of Line Errors 139

LIST OF FIGURES

Figure 1.1	Testing of a simple combinational circuit	4
Figure 1.2	The fault simulation process	7
Figure 1.3	Fault detection in a digital circuit	14
Figure 1.4	Test response analyzer in BIST environment	15
Figure 2.1	Implementation of ones counting	19
Figure 2.2	Syndrome test structure	19
Figure 2.3 (a)	DTC implementation for testing single-output circuits	21
Figure 2.3 (b)	MTC implementation for testing multi-output circuits	21
Figure 2.4	Parity check compressor	22
Figure 2.5	Signature analysis using LFSRs	24
Figure 2.6	MISRs hardware implementation	25
Figure 2.7	The Walsh spectral testing scheme	26
Figure 2.8	Parity tree scheme	28
Figure 2.9	Hybrid space compression scheme	29
Figure 2.10	Estimation of S_1 and S_2 based on the CUT structure	30
Figure 2.11	Syndrome counter used as a time compressor	31
Figure 2.12	Modified hybrid space compaction	32
Figure 2.13	The QFC implementation diagram	33

Figure 2.14	The MPT's compaction scheme	34
Figure 5.1	Space compactor circuit for C432 assuming stochastic independence of line errors	98
Figure 5.2	Space compactor circuit for C432 assuming stochastic dependence of line errors	99
Figure 5.3	Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by assuming stochastic independence of multiple line errors	100
Figure 5.4	Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by assuming stochastic dependence of multiple line errors	100
Figure 5.5	Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by considering parity tree as space compactor	101
Figure 5.6	Simulation results for all benchmark circuits using FSIM and by assuming stochastic independence of line errors	102
Figure 5.7	Simulation results for all benchmark circuits using ATALANTA and by assuming stochastic independence of line errors	102
Figure 5.8	Simulation results for all benchmark circuits using COMPACTEST and by assuming stochastic independence of line errors	102
Figure 5.9	Simulation results for all benchmark circuits using FSIM and by assuming stochastic dependence of line errors	103
Figure 5.10	Simulation results for all benchmark circuits using ATALANTA and by assuming stochastic dependence of line errors	103
Figure 5.11	Simulation results for all benchmark circuits using COMPACTEST and by assuming stochastic dependence of line errors	104

LIST OF TABLES

Table 5.1	Simulation results of the ISCAS 85 benchmark circuits using FSIM without compactors	89
Table 5.2	Simulation results of the ISCAS 85 benchmark circuits using ATALANTA without compactors	89
Table 5.3	Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST without compactors	90
Table 5.4	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic independence of multiple line errors	91
Table 5.5	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using ATALANTA and assuming stochastic independence of multiple line errors	91
Table 5.6	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic independence of multiple line errors	92
Table 5.7	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic dependence of multiple line errors	93
Table 5.8	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using ATALANTA and assuming stochastic dependence of multiple line errors	93

Table 5.9	Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic dependence of multiple line errors	94
Table 5.10	Simulation results of the ISCAS 85 benchmark circuits using FSIM with parity tree	95
Table 5.11	Simulation results of the ISCAS 85 benchmark circuits using ATALANTA with parity tree	95
Table 5.12	Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST with parity tree	96
Table 5.13	Estimates of the hardware overhead for ATALANTA or FSIM	97
Table 5.14	Estimates of the hardware overhead for COMPACTEST	97

Chapter 1

Introduction

The unprecedented growth in electronic systems and more specifically in computing technology, has led to a rapid increase in the complexity of VLSI circuits. It has become possible to mount more and more transistors on a single integrated circuit.

The trend towards extremely high levels of integration density has created a substantial testing difficulty of VLSI circuits. In order to overcome this problem and to produce testable VLSI circuits, designers are increasingly including additional testing guidelines and techniques as well as taking into consideration testing as an integral part of the design process. A variety of techniques for alleviating the testing problem have been developed. This thesis deals with one increasingly popular technique, built-in self-test (BIST). BIST seeks to include additional circuitry that is needed to test the circuit directly on the chip, in order to ensure that the circuit is capable of testing itself. The addition of circuitry to the chip can also have the side effects of rendering the chip larger in area, therefore higher in cost and slower in speed.

The main thrust of this thesis is to design a space compaction technique for BIST of VLSI circuits with the primary objective of minimizing the storage required for the circuit under test (CUT) while maintaining a high percentage of fault coverage. In order to achieve the design, a generalized mergeability criteria have been theoretically developed and established as well as experimentally demonstrated for merging an arbitrary number of output sequences of the CUT under both conditions of stochastic independence and dependence of multiple line errors.

It will be readily concluded that the proposed space compressor requires low area overhead which is suitable for on-chip self testing of VLSI circuits, and guarantees a very high fault coverage as well as a low CPU simulation time.

1.1 Evolution Perspective

During the last four decades, the technique of integration density on a semiconductor wafer has been one of continuing and rapid evolution based largely on planar techniques. Since 1962 small-scale integrated (SSI) circuit packages with up to 10 logic gates fully interconnected per package were becoming available on the market. In the late 1960s, the SSI technology was followed by medium-scale integrated (MSI) circuit packages with integration densities between 10 to 100 logic gates per wafer. The early 1970s saw the birth of large-scale integrated (LSI) circuit packages with integration densities of about 100-1000 logic gates on a single chip. In the late 1970s, LSI technology led to the introduction of very-large scale integrated (VLSI) circuit technology with possible inclusion capability of tens of thousands of logic gates. Today, VLSI technology is virtually the basis for most of the modern electronic and computer systems.

Since the demand for these systems is expected to keep on increasing, the evolution of the integration density in the semiconductor technology will likely continue as long as specific testing guidelines and rules are applied in the process of producing a test-friendly circuit design.

1.2 Basic Testing Concepts

The testing of VLSI circuits which perform complex logic functions is costly and requires substantial efforts. Therefore, designing testable VLSI circuits is both motivated and justified by economic factors.

An integrated circuit is fabricated according to a design document that specifies the function, structure and performance of the end product. Testing is being performed to ensure that any or all of the above characteristics of the final product were not changed or affected during the fabrication process or the life of the product.

A computer system can be characterized according to its function, structure and performance [56]. Function means the scope of the system or its specified task. Structure denotes how the components are interconnected to realize the desired function. Performance refers to how fast and accurately the structure accomplishes the specified function.

Functional testing is performed to verify that digital systems accomplish their intended task. For example, an AND gate would be tested to find out whether it could perform a bitwise AND operation as an arithmetic operator. Functional testing has provided a good measure of proper operation, but in the presence of faults, it was a very difficult task to isolate the fault [7]. In addition to this problem, exhaustive functional testing (i.e. all possible combinations of values) is obviously impractical. Thus, only a sample test set is used to reveal existing faults. However, it is often difficult to determine the size of the sample and which sample is essential. The lack of an effective measure of functional testing was the motivation to create a structural testing approach.

Structural testing is performed to ensure that the specified structure (composed of gates and connections) is correctly implemented according to the design specification and that the final product is free of faults. The procedure implemented to execute structural testing is described in [7] as follows:

"Each of the logic elements and connections are faulted in turn, and a test is generated for that fault. A practical technique to reduce the search time consumed by test generation algorithms is to fault-simulate the test pattern generated for one fault against all other faults in the system. Those additional faults detected by the pattern are marked as tested. It is worth noting that built-in testing can be either functional or structural in nature. Pseudorandom testing is a type of built-in test that performs a structural test of the system involved."

Let us consider the testing of a combinational circuit (a circuit which does not contain flip-flops or latches). The question that comes to mind is how to determine the test set required to test the circuit exhaustively?

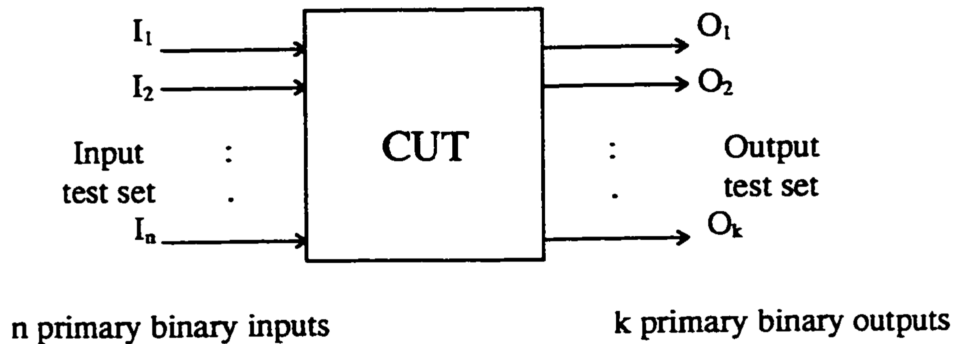


Figure 1.1 Testing of a simple combinational circuit.

In order to answer the above question, let us consider Figure 1.1 which represents the testing of a simple combinational circuit. The circuit has n binary inputs. This leads us to say that 2^n is the number of input test vectors (combinations of values of n binary variables) required to execute an exhaustive input test set. The output bit sequence of each of the 2^n input vectors will be compared to the known fault-free response. This operation undoubtedly requires an enormous amount of time, i.e. exponential in the number of binary inputs.

Performance testing is concerned with ensuring that a fabricated structure achieves the specified performance norms, in terms of stress, time and accuracy.

1.3 Problems of Testing in VLSI Circuits

The major problem in testing VLSI circuits is the inability of the existing tools to handle the increasingly growing number of logic gates in digital systems which were designed without taking into consideration the design complexity impact on testing.
The tools such as current test pattern generators and fault simulators are limited by factors such as: number of gates in the CUT, the speed of the test generator, the capability and efficiency of producing a reduced test set, the number of faults assumed and the fault coverage. Moreover, two fundamental problems can be introduced when testing VLSI circuits. They are as follows:

a) excessive time and expensive cost whenever an exhaustive test is executed,

b) very limited access to the CUT since we are able to only access those pins which belong to the primary inputs and outputs.

1.4 Fault Models

In order to determine some minimum test set of input vectors to test a digital circuit, we must speculate on the kind of faults that are likely to exist in the circuit. The fault model proposed in [39] is as follows: "The effects of failures are hence modeled in terms of the incorrect signal values which they produce. A fault model is the representation of the effect of a failure by means of the change that is produced in the system signals."

On the other hand, fault modeling is described in [31] as follows: upon the application of a particular input test vector, the observable output of the CUT has a certain logic value, for example, 0 when the circuit is free of faults, but it will change to 1 whenever the fault is available somewhere in the CUT. In other words, fault modeling can be used to determine a minimum set of test patterns which will detect the presence or absence of a given type of fault on every internal node of the circuit.

Three models are well known. They are single stuck-at faults, multiple stuck-at faults, and bridging faults models. The most commonly used one is the single stuck-at fault model. This model indicates that only the value of one of the signal lines in the CUT is permanently fixed to either a logical 0 (stuck-at-0) or logical 1 (stuck-at-1), rather than having its value determined by either a gate output or circuit input, due to any physical defect in the circuit.

In the case of the multiple stuck-at fault model, one or more signal lines in a circuit can have values that are permanently fixed at a specific binary value, and they are independent of the other signal lines in the circuit.

The bridging fault model [40] is, as its name indicates, an electrical connection

(which acts as a bridge) between two or more signal lines in a circuit which are supposed to be independent. The main disadvantage of the bridging fault model is the fact that it can convert a combinational circuit into a sequential circuit, which renders test generation a very difficult task.

In testing digital circuits, single stuck-at fault is the most acceptable model to use. In addition to being the simplest fault model, it has been shown both theoretically and empirically to result in test that detects very many multiple stuck-at faults and bridging faults without the necessity to explicitly consider the more complex faults such as multiple stuck-at and bridging faults.

1.5 Testability of a Digital System

A digital system is testable if the exercised effort to test it is relatively insubstantial [38]. In order to enhance the testability of a system, project managers must attempt to lower its testing costs. In general, testing costs can be determined by two main factors:

1. the cost of test pattern generation which depends on the required computing time to run the test pattern generation program or on the cost of developing a set of customized tests.
2. the cost of test application which depends on the cost of test equipments in addition to the cost of the time required to apply the test.

In fact, the testability of a system increases in response to any decrease in terms of test cost.

1.5.1 Quality of Testing and Fault Simulation

The quality of testing a digital system is evaluated in terms of the type of faults detected and the fault coverage. In general, the single stuck-at fault (SSF) model is adopted during the test generation conception. In addition, it has been proved that many

other types of faults will be detected when the SSF model is used [39]. The fault coverage has been an acceptable measure of a given test set. It is defined as the ratio of modeled faults detected among all possible detectable faults in the system. Moreover, the test efficiency gets improved in response to any percentage increase of the fault coverage.

In this thesis, the fault simulation process is used to determine the percentage of fault coverage. This process consists of applying test patterns to the CUT, while injecting specified faults into the CUT, and at the same time observing the responses at the outputs of the CUT by comparing the actual test responses with the previously stored correct ones. The difference between both bit streams allows us to identify the detected faults by the applied test set. Therefore the percentage of the ratio of the detected faults to the total number of the injected faults can be easily determined.

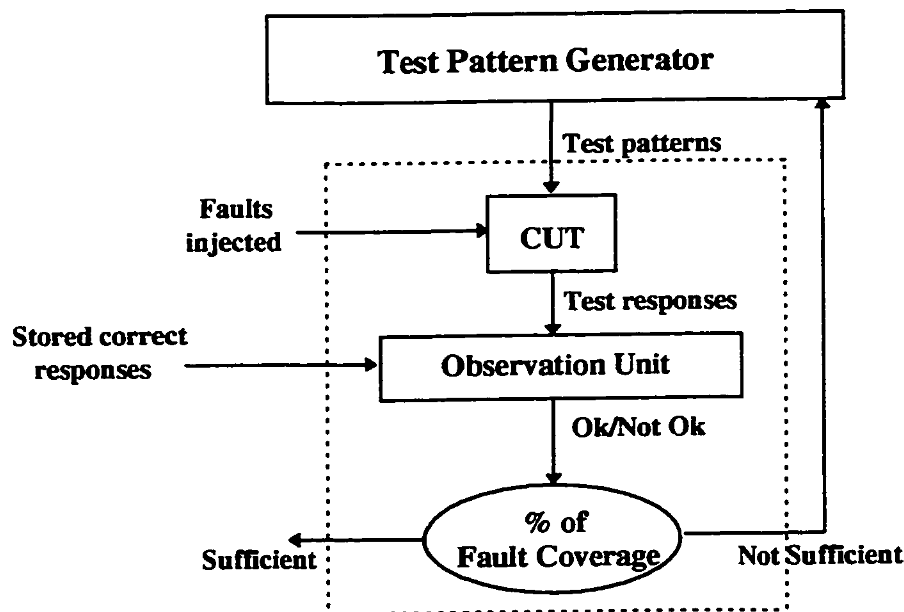


Figure 1.2 The fault simulation process.

1.6 Test Pattern Generation for Combinational Circuits

Since we are only dealing with combinational circuits in this thesis, we will focus our study of the test pattern generation to these circuits only. As their name indicates, the output is directly related to the combination of the values of the input vector at every instant of time.

For a given fault in a circuit, a test is a set of input vectors which render the fault effect observable at a primary output. In combinational circuits, a specific stuck-fault requires a single input vector to be applied.

Test pattern generation is the design process of creating test vectors. Reduced test set is usually sought because the use of exhaustive testing (2^n input patterns for n-input circuit) is impractical, specifically in case of large and complex circuits with numerous primary inputs. Three methods are well-known in the generation of test patterns. They are as follows:

a) manual generation,

b) algorithmic generation,

c) pseudorandom generation.

1.6.1 Manual Pattern Generation

Manual test generation is usually achieved by listing all the input vectors which cause every gate in the digital circuit to change from one logical state to the other. This kind of test set generation becomes very time-consuming when more than one thousand gates are present in a circuit. Thus, for larger and more complex circuits, algorithmic or pseudo-random test-pattern generation is more appropriate.

1.6.2 Algorithmic Pattern Generation

Algorithmic or automatic test-pattern generation (ATPG) can be characterized as either structural or mathematical [48]. Structural techniques are usually derived by a topological gate representation of the circuit, in which all gate inputs and outputs are identified. In addition, a fault model is adopted. Numerous ATPG techniques have been proposed for combinational circuits. Among them, we will discuss the following methods: single path sensitization, D-algorithm, PODEM, FAN, CONT and Boolean Difference.

1.6.2.1 Single Path Sensitization

All structural test pattern generator methods make use of the path sensitization concept as their basis in one way or another. This method traces the path from the location of the fault to an observable output, where the logic state at any gate output along the path is dependent on the logic value at the location of the fault. Thus, the path is sensitive (changes) according to the fault status.

The problem with this method is that sometimes only a single path may not be sensitizable from the location of the fault to an observable output in the circuit; there might be a necessity of multiple path sensitization. Moreover, this method may not generate a test for a fault in a circuit which contains a reconvergent fanout structure. These problems are overcome in the following described method, the D-algorithm.

1.6.2.2 D-algorithm

This algorithm was first introduced by Roth [47] in 1966. In this method, all possible paths are sensitized and systematically processed, until a specific path is traced from the location of the fault to an observable output.

The D-algorithm is a recursive search procedure, which advances one gate at a time and repeats itself until the fault is detected [3].

The main advantage of this method is that it guarantees to find a test for any fault modeled in a combinational circuit, if such a test exists.

The main disadvantage found in this algorithm is the degradation in performance (time-consumption and/or memory utilization) when reconvergent fanouts or Exclusive-OR gates are used, due to the excessive amount of backtracking operations which take place before a test is actually derived. In an attempt to reduce the effect of the backtracking problem of the D-algorithm, many algorithms were born. The best known three algorithms will be studied in the following.

1.6.2.3 PODEM, FAN and CONT

Goel [25] developed a new algorithm called Path Oriented DEcision Making (PODEM) to deal with the backtracking problem of the D-algorithm. For a given fault, PODEM directly searches the set of possible input patterns to create a corresponding test. In other words, it assigns values only to the primary inputs which are in turn propagated towards internal lines by forward implications [29,18,57]. Thus, backtracking is substantially reduced when compared to the D-algorithm because it occurs only at the primary inputs. It has been proven that PODEM implementations can run an order of magnitude faster than the D-algorithm [3].

FANout-Oriented (FAN) algorithm was later proposed by Fujiwara and Shimono [24] to further reduce the effect of the backtracking problem. This algorithm executes special processing of fanout points in the circuit. Backtracking in FAN can stop at internal lines, rather than stopping at primary inputs as in the case of PODEM. FAN uses a multiple backtrack procedure which attempts to satisfy simultaneously a set of objectives. Moreover, it has been shown experimentally that FAN is faster and more efficient than the PODEM algorithm.

CONcurrent Test generation algorithm CONT [1] minimizes the backtracking by a different approach. For a given fault, CONT acts exactly as PODEM. However, for

every gate, a list of active faults is generated. The fault list of a gate contains the faults whose effects appear at the output of that gate in presence of the current primary input values. Once a vector is found for the given fault, all faults in the fault lists associated with primary outputs are also detected by this vector. In addition, when a conflict occurs, instead of changing the primary input values, the target is switched to another fault that is activated by the current vector and is most likely to be propagated to a primary output. In this case, the fault lists of gates are useful in selecting the new target fault [18]. The concept of switching the target fault substantially minimizes the backtracking.

The well-known mathematical technique is the Boolean Difference method [53], which derives test patterns from the logic equations realized by the combinational circuits.

1.6.2.4 Boolean Difference

The Boolean difference method [53] is a mathematical approach to generate test sets for combinational circuits. Even though this approach does not have the popularity enjoyed by structural methods, it captures the basic concepts of path sensitization in elegant algebraic terms and it provides good insights into the process of test generation.

The Boolean Difference is defined [19] as being the Exclusive-OR operation between two Boolean functions, one representing the fault-free circuit while the other represents the faulty circuit. Thus, whenever the Boolean Difference happens to be a 1, a fault in the circuit is indicated.

The disadvantage of this method is the difficulty in manipulating the algebraic equations to derive a test for a given fault.

1.6.3 Pseudorandom Pattern Generation

A bit sequence is called a pseudorandom sequence when the sequence appears to be random in the local scale because both digits 1 and 0 have the same probability to

exist in a given bit position. On the other hand, the whole sequence is reproducible at later time instant, and therefore, the sequence is not fully random in terms of time scale. Hence, the name pseudorandom.

As opposed to a random test, a pseudorandom test is fully repeatable. This represents a great advantage in terms of test applications, since reapplying the same test sequence is required to ensure repeatable signatures for comparison purposes.

One of the primary interests of pseudorandom tests is their extensive use as stimuli for Built-In Self-Test (BIST) which is a particularly powerful form of Design For Testability (DFT).

1.7 Design For Testability and BIST

The advent of VLSI and the rapid increase in gate densities have imposed a real challenge to test engineers. Experiencing new types of failure modes and increasing complexities, as well as deteriorating of gate-to-pin ratios, have hampered our ability to verify the fault-free operation of a given VLSI circuit. In order to solve the VLSI testing problem, approaches have been grouped into two categories: DFT and BIST, though BIST is also a form of DFT. These two categories can be differentiated as follows:

- DFT requires that testability must become an integral part of the design process. Consequently, it simplifies testing since it reduces the cost and the time required to perform the test. The main backdraw of DFT is the additional substantial development time and effort required to include some methods which should be adopted at the design stage. In addition, an increase of I/O pins will be required as a penalty to be paid for providing accessibility to the VLSI circuit.
- BIST seeks to include additional circuitry directly on the chip, in order to ensure that the circuit is capable of testing itself. In BIST, test generation, test application and response verification procedures are accomplished through a built-in hardware [21]. In sum, BIST has the following three most desirable advantages:

1. Reduces the costs of test pattern generation and fault simulation.

2. Eliminates the need to use very expensive external test equipments to apply test patterns and the experience tester to monitor the results.
3. Having all BIST circuitry on the chip itself, tests can be performed at normal operating clock rate. This indicates that BIST techniques have the capability of detecting timing related problems which may be easily missed by an external test.

On the other hand, BIST suffers from the following disadvantages:

1. The additional chip overhead area which is required for the test circuitry. Moreover, this may also cause some degradation in system performance.
2. It has been proven that, to achieve a specific percentage of fault coverage, the number of test patterns generated using BIST is greater than the number of deterministic test patterns required to obtain the same percentage of fault coverage.
3. Since BIST use pseudorandom (non exhaustive) test pattern generator, there is always a probability that a faulty circuit may be considered as free of fault [31]. However, exhaustive (deterministic) testing for complex VLSI circuits is no more attempted, since 100% fault coverage can never be afforded.

1.8 Overall Testing Scheme

In this thesis the following testing scheme will be followed. A Test Pattern Generator (TPG) sends its output bit streams (O_L), where L is the length of the sequence, to the inputs of the CUT. The CUT reacts to this stimulus by generating at its output a bit stream sequence which called test response (R_L), as shown in Figure 1.3. This test response is fed into a "test response analyzer".

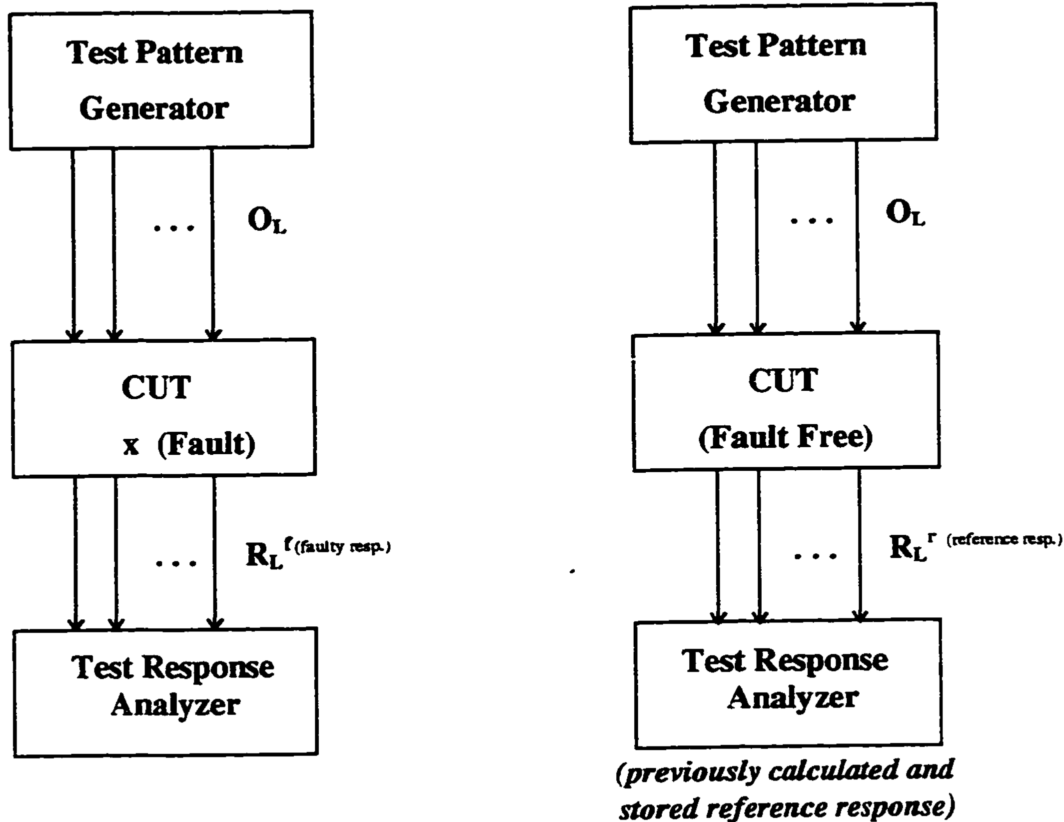


Figure 1.3 Fault detection in a digital circuit.

In order to detect a fault in the CUT, the following steps are necessary:

1. The fault has to be made visible at the output of the CUT. This is achieved, when the test response sequence R_L^f is different from the reference (or nominal) R_L^r which a fault free circuit would have produced.
2. Having R_L^f different than R_L^r has to be translated into a fault message by the test response analyzer.

1.8.1 Response Analysis

Analyzing the test response by comparing bit-by-bit between R_L (test response sequence which might be faulty or not) and R_L^r (fault-free reference response sequence) as in most conventional testing methods, is not practical and unsuitable for complex

VLSI implementation, because it requires a large amount of memory to store the reference responses.

In order to overcome the storage problem, several data compression techniques have been proposed and some of them are in actual use. In all of these techniques the Test Response Analyzer, shown in Figure 1.4, must consist of a Data Compaction Unit, a Storage area for the reference responses of the CUT, and a Comparator which has the ability to compare signatures and the corresponding reference responses.

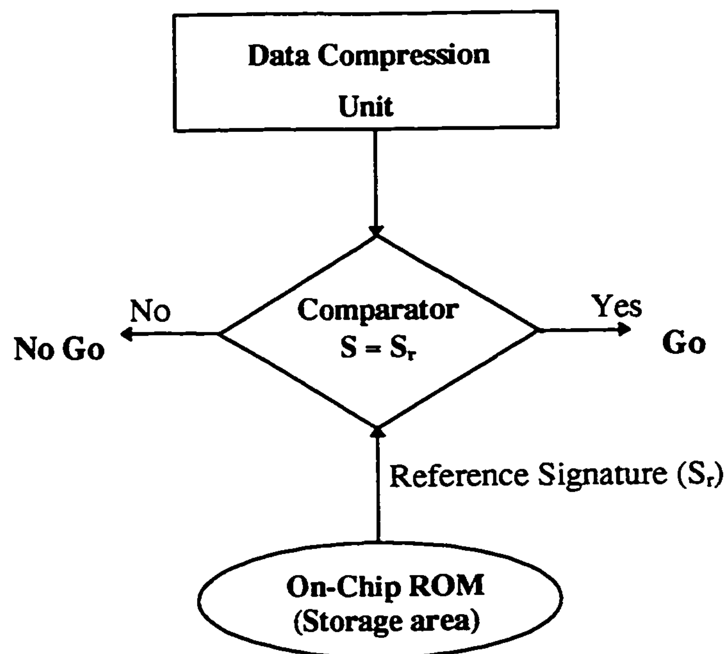


Figure 1.4 Test response analyzer in BIST environment.

The data compaction unit compresses the CUT responses which are therefore called signatures (binary sequences) "S". These signatures are compared with their corresponding precomputed and prestored reference signatures "S_r" in the on-chip ROM. These comparison take place in the comparator unit. As a result, faults are detected whenever a match does not occur.

Compression techniques can be grouped into two fundamental categories: space compression and time compression. The differences between, and the details of each category will be discussed at length in the next chapter.

Built-in self-test techniques usually combine both a built-in stimulus source and response data compression. This approach eliminates the task of test pattern generation and minimizes the storage of test patterns and response data [39].

This thesis deals with designing and analyzing an efficient space compressor technique for built-in self-test of VLSI circuits. Consequently, it uses the compressed CUT response technique of BIST "signature". The observed signature is compared with the already stored "reference signature" of the fault-free circuit. The use of this method results in reduction in the memory area needed to store the correct circuit responses.

1.9 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 presents various existing space and time compaction techniques.

Chapter 3 designs space compaction trees for multi-output combinational circuits assuming stochastic independence of multiple line errors. A design algorithm is proposed in order to establish the generalized mergeability criteria for merging an arbitrary number of output sequences under conditions of stochastic independence of multiple line errors.

Chapter 4 designs space compaction trees for multi-output combinational circuits assuming stochastic dependence of multiple line errors. A design algorithm is proposed in order to establish the generalized mergeability criteria for merging an arbitrary number of output sequences under conditions of stochastic dependence of multiple line errors.

Chapter 5 provides and discusses the experimental results of the simulation of the ISCAS 85 benchmark circuits which are illustrated in a series of tables and graphs.

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

Chapter 2

Overview of Test Compaction Techniques

Compaction can be executed either at the input or at the output of the CUT. Since this thesis deals with output compaction, this topic will be fully elaborated in a devoted section. However, input compaction will be briefly described in the following section.

2.1 Input Compaction

When the number of binary inputs is large, typical combinatorial outputs often depend on a reduced set of those inputs. Thus, for a system of n inputs, where n is large, it requires 2^n input vectors for a global exhaustive testing, but individual outputs may be dependent only on i, j, ... inputs, where i, j, ... < n. Obviously, (2^i, 2^j, ...) input vectors are required for local exhaustive test purposes [30].

In mid 1980's, [50] and [36] provided the foundation approach to the generation of reduced input test sets for exhaustive local testing. This foundation was continued and improved by Akers [4] in 1985.

If we consider the case of three primary inputs (V_1, V_2, V_3, 3-bit test-vector generator) a total of seven bit streams may be generated by using linear (EXOR) relationships [31]. The seven possible input vectors available are: V_1, V_2, V_3, $V_1 \oplus V_2$, $V_1 \oplus V_3$, $V_2 \oplus V_3$, $V_1 \oplus V_2 \oplus V_3$, from which 28 exhaustive 3-bit data streams can be selected. The total number of bit streams possible from a j-bit test vector generator is (2^j-1).

2.2 Output Compaction

During testing, the actual output responses are compared with the prestored reference values; when both are identical the circuit is considered to be fault-free; otherwise, the circuit is faulty.

Data compression techniques have been proposed as new approaches to reduce the amount of data to be stored.

Most of the data compression techniques are oriented towards creating a single-output digital circuit and they all tend to lose some information with an acceptable range of fault coverage. The loss of information is usually called **"masking"** [50] and is measured by the probability that a faulty circuit will produce the same compressed signature as the fault-free circuit. Relative masking probabilities of various compression techniques can be used as a measure of their efficiency. The smallest the masking probability, the better is the compression technique.

2.3 Time Compaction

Many time compaction methods have been suggested in the literature and some are in actual use. These compaction methods are: ones count, syndrome testing, transition count, double and multiple transition counting, parity check, cyclic code, signature analysis, compaction using Walsh spectra, and parallel compaction analysis.

2.3.1 Ones Count Compaction

The signature in ones count [7] is the sum of the number of 1's existing in the output bit stream during the execution of the test.

Masking probability [50] is low when the ones count of the signature is near either the minimum or the maximum of its length range, but it increases rapidly and it

reaches a maximum at the midrange of the count. The hardware implementation of ones counting is depicted in Figure 2.1.

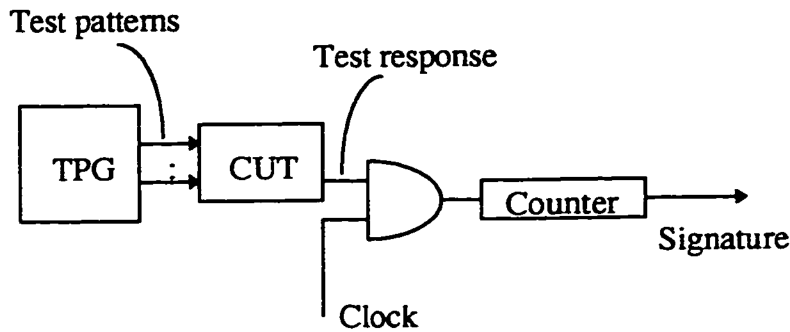


Figure 2.1 Implementation of ones counting.

2.3.2 Syndrome Testing

Syndrome testing [46] is a variation of the ones counting technique. It requires that all 2^n patterns be applied to the n-input combinational circuit; it also works by counting the number of 1's in the output bit stream.

Faults are detected by comparing the number of 1's of the output syndrome (signature) to the number of 1's of the previously stored fault-free syndrome. The general test setup is shown in Figure 2.2.

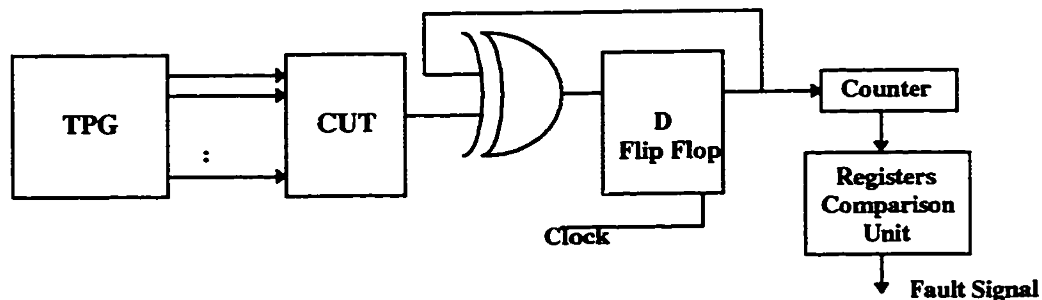


Figure 2.2 Syndrome test structure.

The great advantage of syndrome testing over ones counting is the concept of design for syndrome testing, where a circuit can be designed in such a way that no single fault can cause the circuit to have the same syndrome as the fault-free circuit. On the other hand, not all circuits are syndrome testable [49].

2.3.3 Transition Count Compaction

Transition count compaction, as the name implies, counts the number of times the output bit stream changes from 1 to 0 and vice versa. The actual bit values are ignored.

The similarity [51] between syndrome testing and transition count lies in that both normally consider a fully exhaustive input test set. However, they differ in that, as opposed to syndrome count, transition count is dependent on the order of application of the input test vectors. Thus, exhaustive input set may be ordered such that the number of transitions in the fault-free output response is minimized, which in the extreme is a single 0-to-1 (or 1-to-0) transition [22,27,28,39,42].

Masking [50] in transition count depends on the number of faulty circuits that have the same transition count as the fault-free circuit. Masking probability in transition count is a function of the actual count value. Intermediate counts are more susceptible to masking than low and high counts. Transition count does not guarantee the detection of single-bit errors.

2.3.4 Double and Multiple Transition Counting

A new compaction technique has been recently proposed termed Double/Multiple transition counting (DTC/MTC) [7] which does not result in any loss of information. Single-output circuits can be tested using DTC testing, while MTC testing is used for multi-output circuits. DTC/MTC detects any faults that can be detected by conventional testing methods.

Unlike Transition counting, DTC/MTC techniques do not need repeated input test patterns, and they do not use a counter. Test circuitry consists of an inverter, a switch, an OR logic gate, and a D flip-flop. Figure 2.3 (a) and (b) outline hardware implementations of DTC/MTC testing techniques.

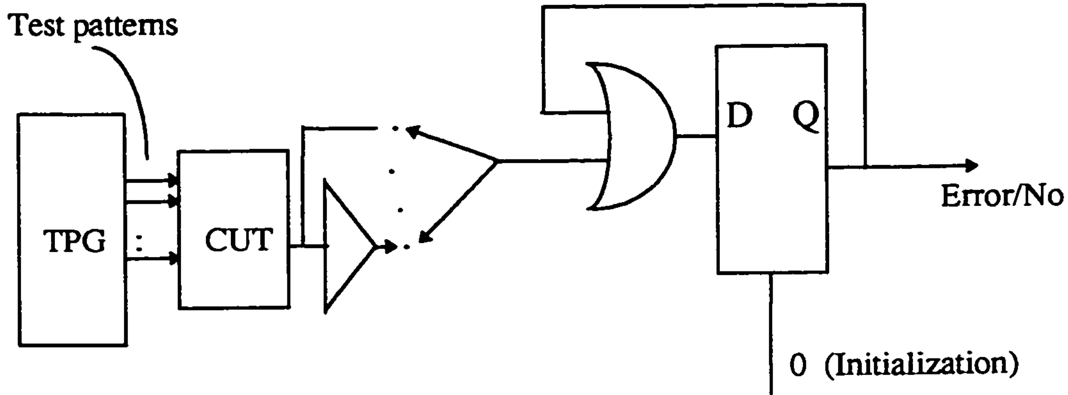


Figure 2.3 (a) DTC implementation for testing single-output circuits.

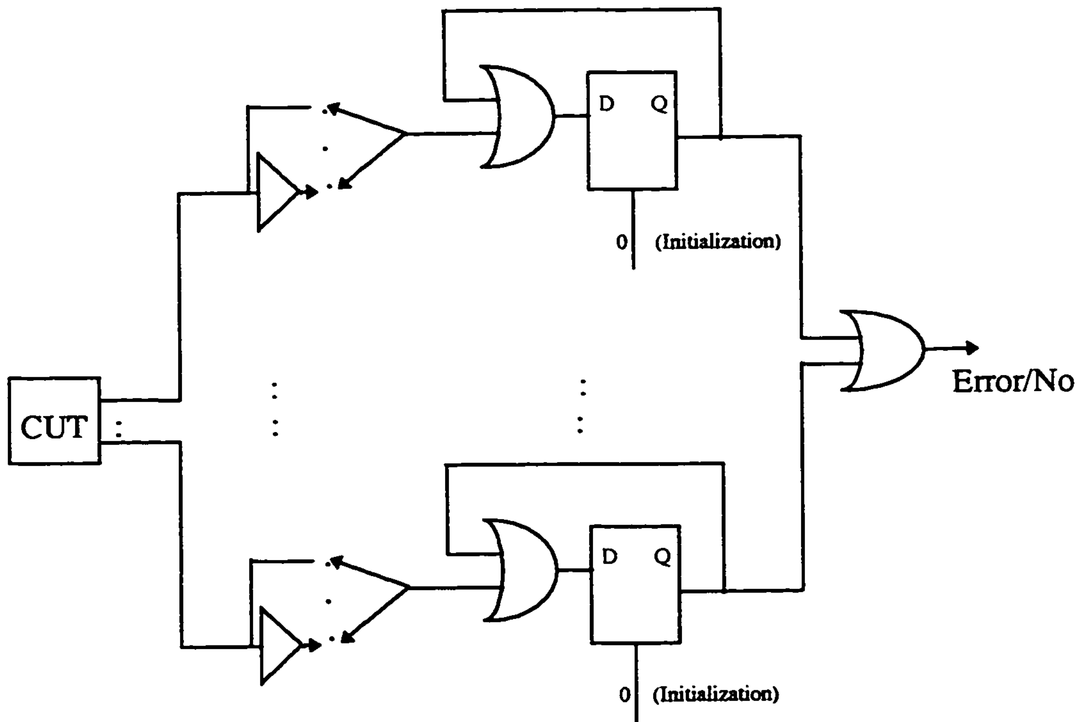


Figure 2.3 (b) MTC implementation for testing multi-output circuits.

2.3.5 Parity Check Compaction

Checking parity [7] on the output data stream is realized by compression of the original n-bit stream to one bit. The value of this bit is 1 if the parity of the test response sequence is odd and 0 if the parity is even. Parity checking detects only errors that result in an odd number of error bits, while it fails to detect errors in the case where an even number of errors occur on the circuit output. This fact renders the parity checking a relatively ineffective compression technique. The simplicity of the compressor for the parity checking is considered as a valuable advantage. It consists of a 1-bit SRL (Shift-Register Latch) with feedback through an Exclusive-OR circuit. The compressor is shown in Figure 2.4.

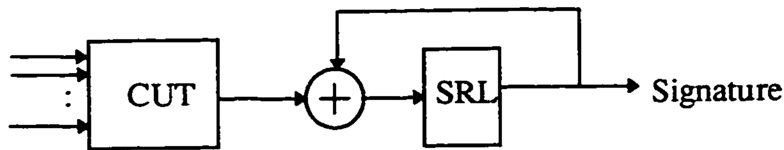


Figure 2.4 Parity check compressor.

2.3.6 Cyclic Code Compaction

An alternative to transition count and more popular signature type is cyclic redundancy check or CRC signature. The CRC [7] is a very powerful and easily implemented technique to detect errors in data communications. In addition to the ease of implementation the CRC requires little overhead and it has extreme error detection capabilities. These advantages combined enable the CRC technique to become one of the most popular coding schemes.

CRC coding has been used very successfully for the compression of test response data [39]. The residue left in the feedback shift register after the responses from the CUT has been compressed is the signature. This process is called signature analysis [7].

2.3.7 Signature Analysis

Signature analysis [29] is currently the most popular compaction technique. It uses linear feedback shift registers (LFSRs) consisting of flip-flops and XOR gates. Signature analysis technique is based on the concept of cyclic redundancy checking (CRC).

LFSRs are used for generating pseudorandom input test patterns, and for response compaction as well. The nature of the generated sequence patterns is determined by the LFSR's characteristic polynomial defined by its interconnection structure. A sequence test $P(x)$ is fed into the signature analyzer, and then divided by the characteristic polynomial $G(x)$ of the signature analyzer's LFSR.

The remainder $R(x)$ obtained by dividing $P(x)$ by $G(x)$ over a Galois field such that $P(x)=G(x).Q(x)+R(x)$ represents the state of the LFSR. In other words, $R(x)$ represents the observed signature.

Signature analysis involves comparing the observed signature $R(x)$ to a known fault-free signature $R_0(x)$. An error is detected if these two signatures differ. Suppose that $P(x)$ is the correct response and $P'(x)=P(x)+E(x)$ is the faulty one, where $E(x)$ is an error polynomial; it can be shown that aliasing occurs whenever $E(x)$ is a multiple of $G(x)$ or $E(x) = h\,G(x)$, where h is a constant.

Different methods for computing and reducing the aliasing probability have been proposed, such as the signature analysis model proposed by Williams *et al.* in [58]. This method uses Markov chains and derives 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. Another approach to the computation of aliasing probability is presented in [44].

An error pattern in signature analysis causes aliasing, if and only if, it is a codeword in the cyclic code generated by the LFSR's characteristic polynomial.

Unlike other methods, the fault coverage in signature analysis may be improved without changing the test. This can be done by changing the length of the LFSR or by using different characteristic polynomial G(x). As demonstrated in [52], for short test length, signature analysis detects all single-bit errors. However, there is no known theory that characterizes fault detection in signature analysis. An outline of the signature analysis scheme is shown in Figure 2.5.

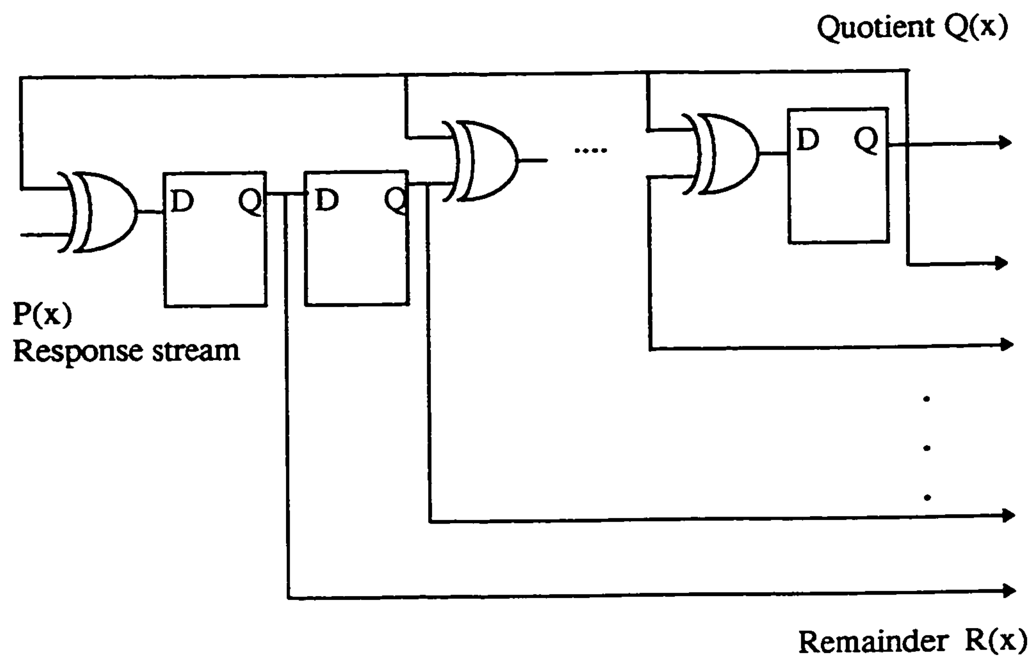


Figure 2.5 Signature analysis using LFSRs.

Multi-output circuits can be tested using multiple-input signature registers (MISRs) as shown in Figure 2.6. The use of this method of testing eliminates the need for a space compactor. Unfortunately, MISRs increase aliasing and require extra hardware which make it far from being practical.

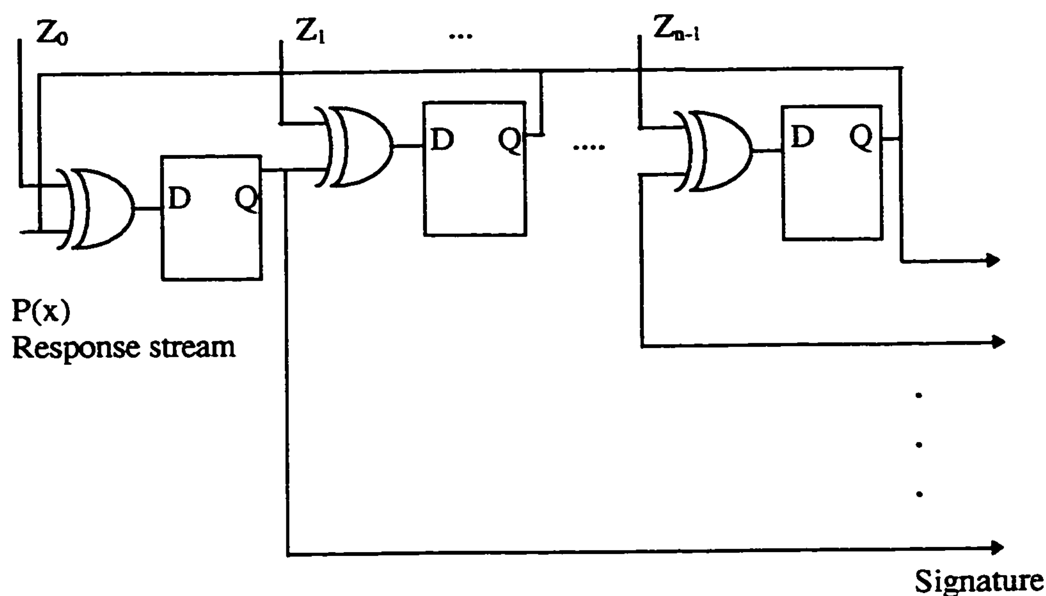


Figure 2.6 MISRs hardware implementation.

Zorian and Agrawal [77] have proposed a method based on modification of the test response for reducing aliasing probability. This method suffers from two problems: it involves high hardware overhead and increases testing time without ensuring zero aliasing.

2.3.8 Compaction Using Walsh Spectra

A technique which is similar to syndrome analysis in that it requires that all possible input patterns be applied to the combinational network is testing by verifying Walsh coefficients [29]. A test procedure that checks for the correctness of Walsh coefficients requires both an exhaustive test and verification of all (2^k, where k is the number of input variables) Walsh coefficients. Obviously, this procedure guarantees the correct behavior of the tested circuit, but it has no advantage over simply checking the response of the circuit to each of the exhaustive input patterns. However a partial

verification of the behavior by checking the validity of a small subset of the Walsh coefficients can provide a very good fault coverage if the subset is so chosen that any modeled fault changes one or more of the coefficients in the subset. The process of collecting and comparing a subset of the complete Walsh functions is described as a mechanism for data compaction.

Although the use of spectral coefficients provides a very good fault coverage, the generation of these coefficients requires a larger area overhead, this can be considered a disadvantage [49]. The hardware implementation of the Walsh spectral testing scheme is depicted in Figure 2.7.

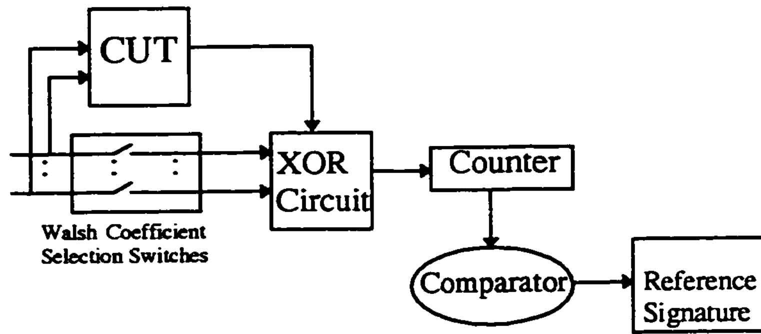


Figure 2.7 The Walsh spectral testing scheme.

2.3.9 Parallel Compaction Analysis

Testing using two different compaction schemes in parallel has been extensively investigated. The combination of signature analysis and transition counting is analyzed in [51]. Analysis shows that using simultaneously both techniques leads to a very small overlap in their error masking.

As a result of using two different compaction schemes in parallel, the fault coverage is improved while the fault signature size and the hardware overhead are highly increased.

2.4 Space Compaction Techniques

Several discrete system outputs may also be compacted into fewer test outputs by the use of EXOR relationships. This compaction may be termed "space compaction", since it does not modify the time for the exhaustive testing of any individual output.

Space compaction may be useful and effective when the number of nodes monitored to derive the response of the fault-free circuit is large. In general, the space compactor is used to reduce the width of the test data, before a time compressor is used to derive the signature.

The objective of the space compactor is to compress the (p) test responses coming from the CUT into (q) test outputs lines (where $q \ll p$), while retaining the fault detection properties of the test set.

The main advantages of the space compactor are the minimization of the memory space required for the reference signatures and the reduction in the number of pins monitored by the tester. However, its serious disadvantage is that compaction causes a loss of useful information which leads to a reduction in fault coverage.

In this section, several space compaction techniques are studied. These techniques are: parity tree compaction, hybrid space compression, dynamic space compression, modified dynamic space compression, modified hybrid space compression, quadratic functions compaction, programmable space compaction, and multiplexed parity tree.

2.4.1 Parity Tree Compaction

The parity tree circuits [46] are composed of only Exclusive-OR gates. An Exclusive-OR gate has a very good error propagation property which is useful for space compactors. The function realized by parity tree compactors is of the form:

$Z = Z_1 \oplus Z_2 \oplus \dots \oplus Z_k$, where n is the number of input streams and $k \ll n$.

The main advantages of the parity tree are that it propagates all errors that appear on an odd number of its inputs and it guarantees a high fault coverage percentage. On the other hand, its most serious disadvantage is that errors that appear on an even number of circuit inputs are masked. The parity tree hardware diagram is shown in Figure 2.8.

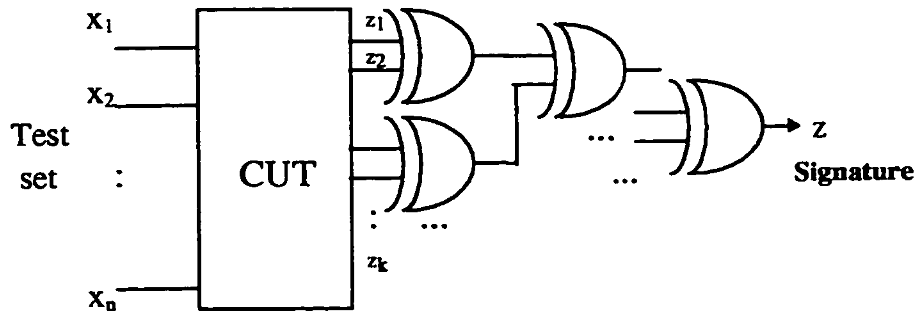


Figure 2.8 Parity tree scheme.

2.4.2 Hybrid Space Compression

Hybrid space compression (HSC) was originally proposed by Li and Robinson [34]. As its name indicates, it contains a hybrid of logic gates. In other words, instead of using only Exclusive-OR (EXOR) gates as an output compaction tool, this technique uses various logic gates such as: AND, OR and EXOR gates. These gates constitute an output compression tree to compress the multiple outputs of the CUT into a single line, as depicted in Figure 2.9. Assuming stochastic dependence of errors, the compaction tree is constructed by using the following detectable error probability estimate (E) for a two input logic function, given two input sequences of length L_s:

$$E = S_1 \frac{R_1}{2\ L_s} + S_2\ \frac{R_2}{L_s}, \text{ where}$$

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

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

R_1: number of single line errors at the output of gate g, if gate g is used, and
R_2: number of double line errors at the output of gate g, if gate g is used.

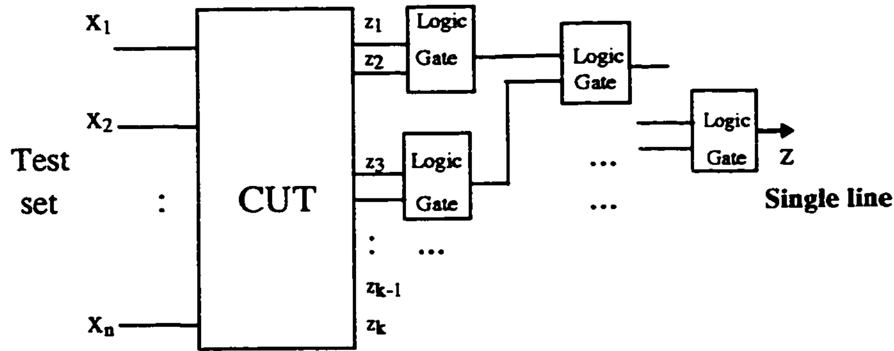


Figure 2.9 Hybrid space compression scheme.

2.4.3 Dynamic Space Compression

Dynamic space compression (DSC) technique was subsequently proposed by Jone and Das [32]. This technique is considered to be a modified version of the HSC method. Instead of assigning a static value for the probabilities of single errors (S_1) and double errors (S_2), this method dynamically estimates those values based on the CUT structure during the computation process. The values of S_1 and S_2 were determined based on the number of single and shared lines connected to an output as shown in Figure 2.10. In this figure, X and Y which are connected to O_i and O_j respectively, represent two single lines. Z which is connected to both outputs, represents a shared line.

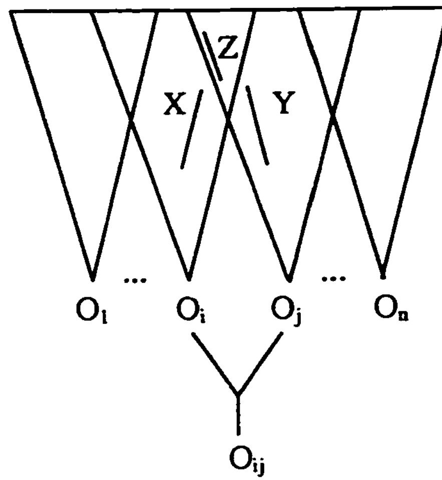


Figure 2.10 Estimation of S_1 and S_2 based on the CUT structure.

In [32], the authors developed a general theory to predict the performance of the space compression techniques. Experimental results showed that the information loss, compared with separate syndrome counting technique, is between 0% and 12.7%. DSC was later improved in [20], in which some circuit-specific information is used to calculate the probabilities.

Both methods, HSC and DSC do not provide an adequate measure of fault coverage because they both rely on estimates of error detection probabilities.

2.4.4 Modified Dynamic Space Compression

Modified dynamic space (MDS) compression [20] is a refinement of the DSC technique. For a circuit under test, a compaction tree is generated based on its structure. The detectable error probability is calculated by using the Boolean Difference Method. The output data modification was employed to minimize the number of faulty output data patterns which have the same compressed form as the fault free patterns. The compressed

outputs were then fed into a syndrome counter to derive the signature for the circuit as shown in Figure 2.11.

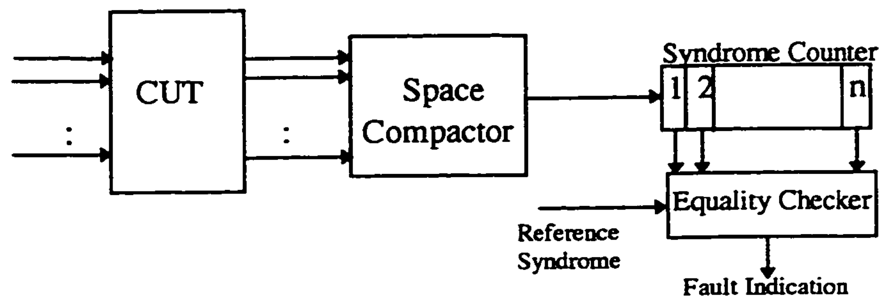


Figure 2.11 Syndrome counter used as a time compressor.

One of the limitations of using the Boolean Difference method to derive the number of detectable faults in a circuit is that the method requires a very high storage capacity. Furthermore, for circuits with a very high density of gates, this method becomes unsuitable because of the computational complexity involved. Experimental results showed that the loss of information was in the range of 0% to 10%.

2.4.5 Modified Hybrid Space Compaction

Assaf [6] has recently proposed a space compression technique in which pairs of output sequences are merged together with appropriately chosen logic gates to form a compaction tree that minimizes the storage for the CUT by using the concepts of Hamming distance and sequence weights. This is shown in Figure 2.12.

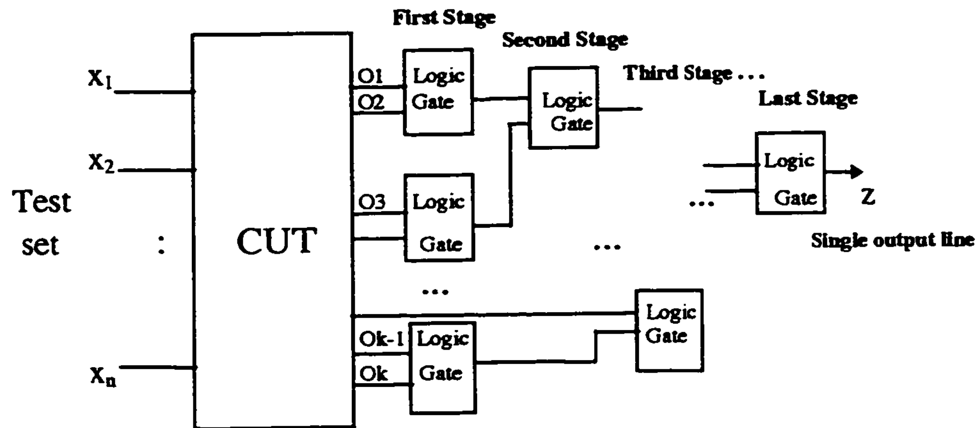


Figure 2.12 Modified hybrid space compaction.

Among the advantages of this approach is the simplicity of the design and low area overhead. However, the main disadvantage is that only pairs are merged together, which leads to the existence of too many intermediate stages and loss of useful information.

2.4.6 Quadratic Functions Compaction

Quadratic functions compaction (QFC) [33] uses quadratic functions to construct the space compaction circuit and has been shown to reduce aliasing. In QFC, the observed output responses $Z_1,\ldots,Z_k$ of the CUT are processed and compressed in a serial fashion based on a function of type

$Z_{i0} Z_{i1} \oplus Z_{i2} Z_{i3} \oplus \ldots \oplus Z_{i(2k-2)} Z_{i(2k-1)}$ where Z_{it} and $Z_{i(t+1)}$ are blocks of length k, for t = 0, 2, ..., 2k-2. The hardware implementation of a quadratic compressor consists of a finite-field multiplier and XOR gates as shown in Figure 2.13.

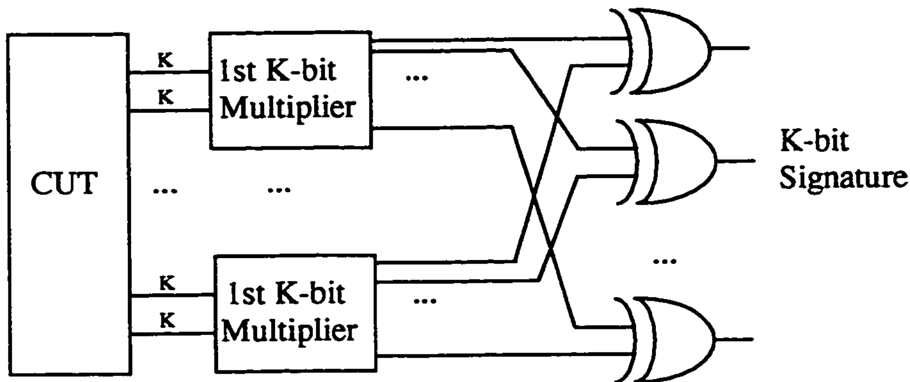


Figure 2.13 The QFC implementation diagram.

As it is readily noticed, the quadratic compressor scheme requires substantial hardware overhead. Unlike compression techniques by LFSRs, the error-masking probability of QFC is shown to be constant. Moreover, QFC does not guarantee zero aliasing.

2.4.7 Programmable Space Compaction

A new approach termed Programmable space compaction (PSC) has recently been proposed for designing low-cost space compactors that provide high fault coverage [76]. In PSC, a circuit-specific space compactor is designed to increase the likelihood of error propagation. However, PSC does not guarantee zero aliasing. A compaction circuit that minimizes aliasing and has the lowest cost can only be found by exhaustively enumerating all $(2^k)^k$ k-input Boolean functions, where k represents the number of primary outputs of the CUT.

PSC can be practically implemented [76] using search techniques based on genetic algorithms.

2.4.8 Multiplexed Parity Trees

A new class of space compactors, based on parity tree circuits, was recently proposed by Chakrabarty and Hayes in [13,16]. This method is termed Multiplexed parity trees (MPTs), and introduces no aliasing. Multiplexed parity trees perform space compaction of test responses by combining the error propagation properties of multiplexers and parity trees through multiple time-steps. The authors show that the associated hardware overhead is moderate and that very high fault coverage is obtained for faults in the compactor. The MPT scheme is shown in Figure 2.14.

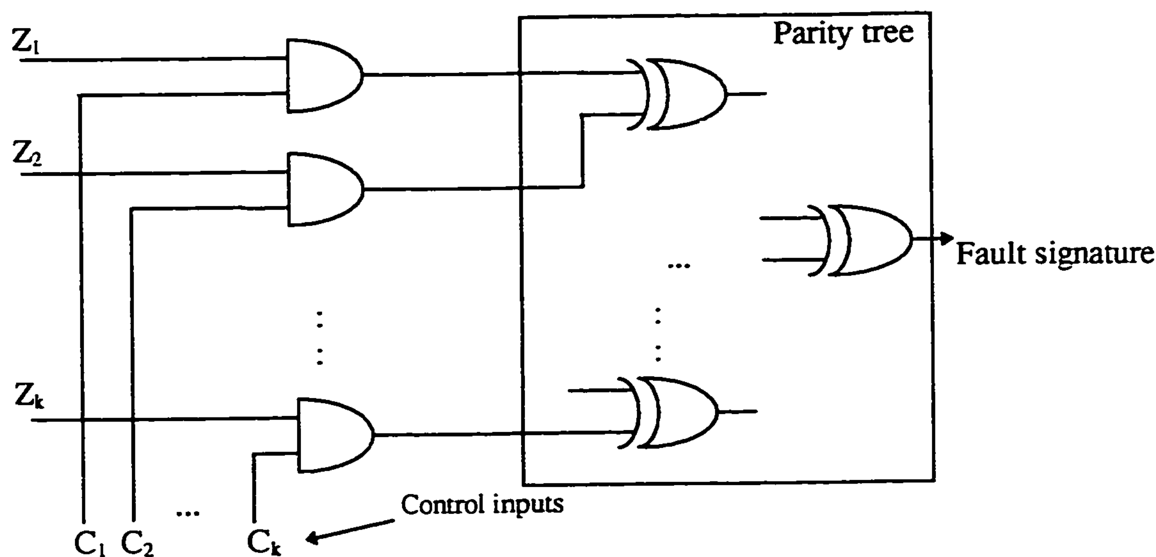


Figure 2.14 The MPT's compaction scheme.

A different approach to achieving zero aliasing was also proposed by the authors. This technique is based on constructing multiple-output space compactors. They show that a two-output circuit implementing two parity functions is adequate for aliasing-free compaction.

Chapter 3

Designing of Space Compaction Trees for Multi-Output Circuits in BIST Based on Stochastic Independence of Multiple Line Errors

This chapter is divided in two major parts: theory and implementation. The theory part defines some basic notions of line errors and proves several important theorems which form the mathematical basis of the proposed approach. The implementation part discusses the algorithm in which the generalized mergeability (gate selection) criterion was used to design compaction trees for ISCAS 85 benchmark circuits.

3.1 First-Order Weights

3.1.1 First-Order 1-Weight:

The number of 1's in a bit stream sequence Ψ_1 of length L_s is called its 1-weight (first-order).

Example:

Let Ψ_1 = 1100 0101 0010,

Here, $L_s = 12$ and the 1-weight (first-order) is $W_{11} = 5$.

3.1.2 First-Order 0-Weight:

The number of 0's in a bit stream sequence Ψ_1 of length L_s is called its 0-weight (first-order).

Example:

In the Ψ_1 above, 0-weight (first-order) is $W_{10} = 7$.

Theorem 1

For a sequence Ψ_1 of length L_s with 1-weight (first-order) W_{11} and 0-weight (first-order) W_{10}, respectively,

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

Proof

Follows by definitions.

Example

We know that $W_{11} = 5$, $W_{10} = 7$ and $L_s = 12$ then $L_s = W_{11} + W_{10}$.

This result can be extended to N sequences: $\Psi_1, \Psi_2, \ldots, \Psi_N$, as follows.

3.2 Nth Order 1-Weight:

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ each of length L_s, let the sequences have 1's in m <u>identical</u> bit positions (each of the sequences might have 1's in other positions as well as besides these m positions). Then, the Nth order 1-weight of the sequences in the group

is given by the number of positions in which all the N sequences have 1's which is m in this case.

Example:

Consider the following three sequences of length $L_s = 12$:

$\Psi_1 = 1111 \quad 0010 \quad 1010$

$\Psi_2 = 1111 \quad 0111 \quad 0110$

$\Psi_3 = 1111 \quad 0101 \quad 1010$

Here, the third-order 1 weight of sequences Ψ_1, Ψ_2, Ψ_3 is: $W_{31} = 5$, since the sequences match in bit positions 2, 9, 10, 11, 12 by having 1's (counting from right to left).

Note 1: Each of the sequences Ψ_1, Ψ_2, and Ψ_3 have 1's in other bit positions besides second, ninth, tenth, eleventh and twelfth positions.

Note 2: Nth-order 1-weight and 0-weight are denoted, respectively, by W_{N1} and W_{N0}, where the first subscript corresponds to the sequence number or order and the second subscript gives the binary digit corresponding to the weight.

3.3 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 K of bit positions in which all the N sequences have 0's.

Example:

Considering the example above for a group of 3 sequences, the third-order 0-weight is: $W_{30} = 2$, since all the three sequences have 0's in the first and eight bit positions from the right to the left counting order.

Definition

If N sequences are grouped together having certain Nth order 1-weight or 0-weight, 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 that we are working under bundling constraint.

3.4 Derived Sequences

3.4.1 Second-Order Derived Sequences:

Consider two sequences of length L_s. If the Hamming distance is H_s (the number of bit positions in which they differ), the derived sequences (second-order) are obtained by deleting these bit positions and putting a dash (-) in these positions.

Example:

Consider the sequences Ψ_1 and Ψ_2 of the earlier example. The derived sequences are denoted by Ψ'_1 and Ψ'_2, respectively, and given as:

$\Psi'_1 =$ 1111 0-1- --10

$\Psi'_2 =$ 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 2

For a pair of sequences Ψ_1 and Ψ_2 of length L_s, with Hamming distance H_s, let the derived sequences be Ψ'_1 and Ψ'_2 and the 1-weight and 0-weight W'_{21} and W'_{20}, respectively. Then

$$L_s = H_s + W'_{21} + W'_{20}.$$

Proof: The proof is provided by the following example.

Example:

Consider the following two sequences, Ψ_1 and Ψ_2.

$\Psi_1 = 1111 \quad 0011 \quad 1010$

$\Psi_2 = 1110 \quad 0000 \quad 1010$

The length of both streams is 12 ($L_s = 12$). The hamming distance between Ψ_1 and Ψ_2 is 3 ($H_s = 3$).

The derived sequences are denoted by Ψ'_1 and Ψ'_2, respectively, and given as:

$\Psi'_1 = 111\text{-} \quad 00\text{--} \quad 1010$

$\Psi'_2 = 111\text{-} \quad 00\text{--} \quad 1010$

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 $W'_{21} = 5$ and $W'_{20} = 4$.

Therefore, it is obvious that $L_s = 3+5+4 = 12$.

3.4.2 Nth-Order Derived Sequences

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ of length L_s, the Nth order derived sequences $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$ 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 by replacing the bit positions with a dash (-).

Example:

Consider the following three sequences of length $L_s = 12$:

$\Psi_1 = 1111 \quad 0010 \quad 1010$

$\Psi_2 = 1111 \quad 0111 \quad 0110$

$\Psi_3 = 1111 \quad 0101 \quad 1010$

For the above three sequences, the corresponding derived sequences are as follows:

$\Psi'_1 = 1111 \quad 0--- \quad --10$

$\Psi'_2 = 1111 \quad 0--- \quad --10$

$\Psi'_3 = 1111 \quad 0--- \quad --10$

<u>Theorem 3</u>

If the Nth order 1-weight is W'_{N1} and the Nth order 0-weight is W'_{N0} of the N derived sequences in the bundle, then

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

where R is the residue (Hamming distance of multiple sequences) which is the number of bit positions where at least two of the sequences of $\Psi_1, \Psi_2, \ldots, \Psi_N$ in the bundle have different entries and the corresponding set of derived sequences have (-) dash entries.

Proof:

The theorem follows by definitions.

Theorem 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 provided by the following example. Consider two bundled sets of three sequences as follows:

$\Psi_1 = 1101 \quad 0101$

$\Psi_2 = 1110 \quad 0000$

$\Psi_3 = 1111 \quad 0100$

and

$\Phi_1 = 0110 \quad 1001$

$\Phi_2 = 0111 \quad 0010$

$\Phi_3 = 0110 \quad 1011$

Their corresponding derived sequences are as follows:

$\Psi_1' = 11\text{--} \quad 0\text{-}0\text{-}$

$\Psi_2' = 11\text{--} \quad 0\text{-}0\text{-}$

$\Psi_3' = 11\text{--} \quad 0\text{-}0\text{-}$

and

$\Phi_1' = 011- \quad -0--$

$\Phi_2' = 011- \quad -0--$

$\Phi_3' = 011- \quad -0--$

Both the derived sets have the same W'_{31} , W'_{30} and R the residue. Surely, they come from two different sets of sequences and also represent entirely distinct sets of derived sequences.

Definition: Error Multiplicity

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

Theorem 5

On the assumption of stochastic independence of errors for a bundled set of N output sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ each of length L_s at the output of a CUT, the maximum number of possible errors with all possible multiplicities of errors ν_i, is:

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

Proof:

For N sequences of length L_s, every bit position can have either only one error (error multiplicity ν_1), can have two simultaneous errors with (error multiplicity ν_2), and so on up to N possible simultaneous errors with error multiplicity ν_N. The total sum of all possible errors in one bit position is then equal to: $2^N - 1$, 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 sequences of length L_s, the total maximum value of all errors is:

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

<u>**Theorem 6**</u>

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$, 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 $\Psi_1, \Psi_2, \ldots, \Psi_N$ are merged together by using an N-input AND/NAND gate is:

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

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

Proof:

Since the derived sequences have 1-weight W'_{N1}, having all 1's in these positions, errors of multiplicities 1, 2, ..., N ($\nu_1, \nu_2, \ldots, \nu_N$) are detected by an N-input AND gate in each position. The sum of all these errors is $[W'_{N1}]^{(2N-1)}$. 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 0's in a bit position, only one error of appropriate multiplicity is detected (if the sequence has r

0's, then only error of multiplicity $\nu = r$ making the position 1 is detected). The total number of all these errors is R. Hence, the AND gate detects:

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

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

<u>**Theorem 7**</u>

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$, have Nth order 1-weight and 0-weight, respectively, W'_{N1} and W'_{N0} and let R be the residue. The total number of faults detected when the N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ are merged by using an N-input OR/NOR gate is:

$$T\ (OR/NOR) = [\ (W'_{N0})\ \nu_1, \nu_2, \ldots, \nu_N\ ]^{(2N-1)} + (W'_{N1})\ \nu_N + \sum_{\nu_j} (R_i)\nu_j$$

where $(W'_{N0})\ \nu_1, \nu_2, \ldots, \nu_N$ represents the number of error multiplicities of 1, 2, ..., N and $(R_i)\ \nu_j$ represents the number of errors corresponding to the ith order residue position of ν_j 0's.

Proof:

The OR gate obviously detects $[W'_{N0}]^{(2N-1)}$ errors corresponding to W'_{N0} bit positions having all 0's and W'_{N1} errors corresponding to W'_{N1} bit positions having all 1's and $R = L_s - W'_{N1} - W'_{N0}$ in all the R residual positions. Hence, the total numbers of the errors detected is R. Hence, the OR gate detects:

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

Theorem 8

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$, respectively, have Nth order 1-weight and 0-weight, respectively, W'_{N1} and W'_{N0} and let R be the residue. The total fault detected when the N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ are merged together by using an N-input EXOR/EXNOR gate is:

$$T\,(\text{EXOR/EXNOR}) = [(W'_{N1})\, v_1, v_3, \ldots, v_{N-1}]^{(2N/2)} + [(W'_{N0})\, v_2, v_4, \ldots, v_N]^{(2N/2)} + \sum_{v_j} (R_i) v_j = [L_s]^{(2N/2)} = [L_s]^{(2N-1)}.$$

where $(W'_{N1})\, v_1, v_3, \ldots, v_{N-1}$ represents the number of (1) error multiplicities of 1, 3, ... , N-1 and $(W'_{N0})\, v_2, v_4, \ldots, v_N$ represents the number of (0) error multiplicities of 2, 4, ... , N. In addition, $(R_i)v_j$ represents the number of errors corresponding to the ith order residue position of v_j 0's, as well as N is even.

Proof:

If the number of sequences N is even, then corresponding to each of W'_{N1} positions having all 1's, all errors of odd multiplicity are detected at the output of CUT. Similarly, all errors of even multiplicities are detected at the output of the CUT corresponding to each of the W'_{N0} positions having all 0's. For the remaining R residual positions, depending on the number of 0's, 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 EXOR/EXNOR gate detects a total of:

$$T(EXOR/EXNOR) = [L_s]^{(2N-1)} \text{ faults.}$$

Theorem 9

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences be $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$, respectively. Let the Nth order 1-weight be W'_{N1} and Nth order 0-weight be W'_{N0}. For merger of the N sequences, an N-input AND gate will be preferable to an N-input OR gate for maximizing the error detection if:

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

Proof:

The total errors detected by the N-input AND gate and N-input OR gate are respectively:

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

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

where R is the residual bit positions where at least two of the sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ have differing entries.

AND/NAND is preferable to OR/NOR if:

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

or

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

or

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

Corollary 9.1

For the merger of N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input OR gate is preferable to an N-input AND gate if:

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

Theorem 10

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences be: $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$.
In addition, let the Nth order 1-weight be W'_{N1} and the Nth order 0-weight be W'_{N0}. For merger of N sequences, an N-input AND will be preferable to an N-input EXOR gate for maximizing the error detection if:

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

Proof:

The total errors detected by the N-input AND gate and N-input EXOR gate are, respectively, as follows:

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

$$T\,(\text{EXOR/EXNOR}) = L_s^{(2N-1)}.$$

where R is the residual bit positions where at least two of the sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ have differing entries.

AND/NAND is preferable to EXOR/EXNOR if:

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

or

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

or

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

or

$$2W'_{N1} \;>\; (W'_{N1} + W'_{N0} + R) = L_s$$

or

$$W'_{N1} \;>\; (W'_{N0} + R)$$

or

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

Corollary 10.1

For merger of N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input EXOR gate is preferable to an N-input AND gate if:

$$W'_{N1} \;<\; (W'_{N0} + R)$$

or

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

Theorem 11

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences be: $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$.
In addition, let the Nth order 1-weight be W'_{N1} and the Nth order 0-weight be W'_{N0}. For merger of N sequences, an N-input OR will be preferable to an N-input EXOR 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 gate and N-input EXOR gate are, respectively, as follows:

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

$$T\,(EXOR/EXNOR) = L_s^{(2N-1)}.$$

where R is the residual bit positions where at least two of the sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ have differing entries.

OR/NOR is preferable to EXOR/EXNOR if:

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

or

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

or

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

or

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

or

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

or

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

Corollary 11.1

For merger of N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ of length L_s, 1-weight W'_{N1} and 0-weight W'_{N0}, an N-input EXOR gate is preferable to an N-input OR gate if:

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

or

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

Theorem 12

For N sequences $\Psi_1, \Psi_2, \ldots, \Psi_N$ at the output of a CUT of length L_s, let the derived sequences be: $\Psi'_1, \Psi'_2, \ldots, \Psi'_N$.

In addition, 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 N-input AND, OR and EXOR gates are respectively, as follows:

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

and

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

Proof

Follows by definitions.

3.5 Implementation

The implementation part can be realized by establishing the first algorithm. This algorithm uses the generalized mergeability criterion and it is coded in the C language, in order to obtain the space compactors for all the ISCAS 85 benchmark circuits. In the implementation, the first proposed algorithm is built and explained in details and it is followed by two detailed examples for clarification purposes.

3.6 The First Algorithm

A) Separation Criteria of Sequences for AND and OR Gates

A.1) All original sequences are separated in two lists: ANDlist, ORlist as follows:

- All sequences that have number of 1's > $L_s/2$ are sent to the ANDlist (where L_s is the length of the sequence).
- All sequences that have number of 1's < $L_s/2$ are sent to the ORlist.

A.2) Sequences (x_i) where i = {0,1,2,...} that have number of 1's = $L_s/2$ are sent to the appropriate list where there is a potential grouping. This is done according to the following algorithm:

Let us consider the case where i = 1,

- Execute the operation: [x_1] bitwise AND [all sequences of ANDlist] then retain and count the outputs that have number of 1's = $L_s/2$, let "andMatch" be that number.
- Execute [x_1 inverted] bitwise AND [all sequences of Orlist inverted] then retain and count the outputs that have number of 1's = $L_s/2$, let "orMatch" be that number.
- Now, compare "andMatch" and "orMatch" numbers, then send x_1 to the largest list among them, where a potential grouping exists.

<u>Notes</u>

- Inversion here means one's complement. For example, if x_1 = 011001111 ==> [x_1 inverted] = 100110000.

- L_s: is the length of a sequence.

B) Stage 1

B.1) Obtaining the list of sequences having Best 1's or 0's Grouping

- From the ANDlist, group sequences according to an Algorithm which takes into considerations (W_{N1}, N_1). i.e. in case of AND gate:
  W_{N1}: number of matching 1's in the same bit positions of all candidate sequences,
  N_1: Number of sequences that have W_{N1} matching of 1's bits in common.

- Detailed explanation of this Algorithm is given as follows:

B.1.1) W_{N1} overrides N_1. In other words, the group that has maximum value of W_{N1} is the best grouping as long as $N_1 \geq 2$.

B.1.2) The best grouping among two groups that have the same value of W_{N1} is the one that has the bigger N_1 value.

B.2) The same Algorithm can be applied to ORlist but we should replace 1 by 0.

B.3) Mergeability Criteria

Let us assume the following definitions:
W_{N1}: number of matching 1's in the same bit positions of all candidate sequences,
W_{N0}: number of matching 0's in the same bit positions of all candidate sequences.

B.3.1) For $\mathbf{W_{N1} > 0.6L_s}$: the best grouping of sequences obtained from B.1) are merged together with an AND gate only. The resulting output is sent back to the AND sublist.

B.3.2) For $\mathbf{0.5L_s \leq W_{N1} \leq 0.6L_s}$: the best grouping of sequences obtained from B.1) are merged together with an AND and XOR gates. The resulting output is sent either to

AND, OR, or XOR (when $L_s/2$ case exists) sublists. All resulting trees of gates are found by using a recursive algorithm.

B.3.3) For $\mathbf{W_{N0} > 0.6L_s}$: the best grouping of sequences obtained from B.2) are merged together with an OR gate only. The resulting output is sent back to the OR sublist.

B.3.4) For $\mathbf{0.5L_s \leq W_{N0} \leq 0.6L_s}$: the best grouping of sequences obtained from B.1) are merged together with an OR and XOR gates. The resulting output is sent either to AND, OR, or XOR (when $L_s/2$ case exists) sublists. All resulting trees of gates are found by using a recursive algorithm.

B.4) Sorting the output sequence after merging with an XOR gate

If the output sequence has:

- Number of 1's > $L_s/2$: the sequence is sent to AND sublist of candidates,
- Number of 1's < $L_s/2$: the sequence is sent to OR sublist of candidates,
- Number of 1's = $L_s/2$: the sequence is sent either to AND or OR sublists according to the algorithm described in A.2).

C) Stage Two and Intermediate Stages

- Repeat the same logic as in Stage A.
- Intermediate stages exist as long as there are still sequences which can be grouped by either AND, AND/XOR, OR, OR/XOR gates.

D) Last Stage (XOR Processing)

The sequences obtained without being able to be grouped in AND as well as in OR gates are merged together in XOR gate. At this point, we receive the last output sequence of the whole list of sequences.

3.7 Explanation of the First Algorithm

The following rules with an example are important for the clarification of the first algorithm.

Rule

Let us assume we have two 1's grouping of sequences with (W_{N1}, N_1) and (W'_{N1}, N_1'). The following rule is applied to obtain the optimal solution (best grouping out of both of them):

a) Weight "W" takes advantage over (overrides) number of sequences "N". In other words, the group that has the maximum value of W is the best solution,
b) If $W_{N1} = W'_{N1}$, our optimal solution is the group that has the maximum value of N,
c) If $W_{N1} = W'_{N1}$ and $N_1 = N_1'$, our optimal solution is either one.

General Rule

The previous rule can be generalized for several groups of sequences i.e. N sequences. Let us consider the following sequences: 0, 1, 2, 3, 4, ..., N. G_k is the group of sequences which has K sequences ($k < N$) that have matching of 1's bits.

G_i is the group of sequences which has i sequences ($i < N$) that have matching of 1's bits.

Three cases can be considered as follows:

a) if $W_{k1} > W_{i1}$ ==> group G_k is better than group G_i,

b) if $W_{k1} < W_{i1}$ ==> group G_i is better than group G_k,

c) if $W_{k1} = W_{i1}$ and $N_k \geq N_i$ ==> group G_K is better than group G_i.

<u>Note:</u> The same procedure can be followed for matching 0's of bits.

<u>Example 1</u>

Find the FIRST best gate selection for the following bit streams.

1 1 1 1 1 1 0 0 1 ==> seq 0
1 1 1 1 1 1 0 1 0 ==> seq 1
1 1 1 1 1 1 0 1 0 ==> seq 2
1 1 1 1 1 1 0 1 1 ==> seq 3
1 1 0 0 0 0 0 1 0 ==> seq 4
1 1 0 0 0 0 1 0 1 ==> seq 5
1 1 0 1 0 1 0 1 0 ==> seq 6

In the above example, we have 3 groups of 1's and 5 groups of 0's to be considered.
$L_s = 9$ and ($0.6L_s = 5.4$).

All Possibilities of 1's Grouping are:

First 1's group: $W_{71} = 2 > 0.6L_s$, $N_1 = 7$ (seq 0, ..., seq 6)
Second 1's group: $W_{41} = 6 < 0.6L_s$, $N_2 = 4$ (seq 0, ... , seq 3)
Third 1's group: $W_{31} = 7 > 0.6L_s$, $N_3 = 3$ (seq 1, ..., seq 3)

All Possibilities of 0's Grouping are:

First 0's group: $W_{40} = 1 < 0.6L_s$, $N_1 = 4$ (seq 0, ..., seq 3)
Second 0's group: $W_{20} = 4 < 0.6L_s$, $N_2 = 2$ (seq 4, seq 5)
Third 0's group: $W_{20} = 2 < 0.6L_s$, $N_3 = 2$ (seq 1, seq 2)
Fourth 0's group: $W_{20} = 4 < 0.6L_s$, $N_4 = 2$ (seq 4, seq 6)
Fifth 0's group: $W_{30} = 2 < 0.6L_s$, $N_5 = 3$ (seq 4, seq 5, seq 6)

From all the above possibilities, according to the highest weight, only two groups are to be considered:

- Second 1's group: $W_{41} = 6 > 0.6L_s$, $N_1 = 4$ (seq 1, seq 2, seq 3)

and

- Third 1's group: $W_{31} = 7 > 0.6L_s$, $N_3 = 3$ (seq 0, ..., seq 6).

Solution:

By applying the general rule and since (W_{31} = 7) > (W_{41} = 6), we select the third 1's group (seq 1, seq 2, seq 3) as the BEST GROUP to be merged into an **AND** gate.

Example 2 Circuit C432

In this example, the output sequences of the CUT for circuit C432 are grouped in logical gates according to the first proposed algorithm using the FSIM/ATALANTA simulators. The FSIM/ATALANTA space compactor logical circuit is illustrated in Chapter 5.

The output sequences of the CUT for circuit C432 are as follows:

Ψ_0=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1
Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0
Ψ_3=1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0
Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0

Let us consider the following variables:
The left threshold = 0.5, the right threshold = 0.6, the length of each of the above sequences is L_s = 46, $0.5xL_s$ = 23, and $0.6xL_s$ = 27.60.

The sequences that have the total number of 1's greater than $L_s/2$ are Ψ_0, Ψ_1, Ψ_2, and Ψ_3. The number of 1's in: Ψ_0 is 32, Ψ_1 is 26, Ψ_2 is 27, Ψ_3 is 38. According to the proposed algorithm, all the sequences that have number of 1's > $L_s/2$ are going to be sent to an AND candidates list, as follows:

Ψ_0=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1
Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0

Ψ_3=1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1

In order to obtain the best group among Ψ_0, Ψ_1, Ψ_2, and Ψ_3 (sequences that have the highest number of matching 1's in the same bit positions), we consider all the grouping possibilities of the various combinations of sequences Ψ_0, Ψ_1, Ψ_2, and Ψ_3. Sequences Ψ_0 and Ψ_3 constitute the "best group" because their second order 1-weight W_{21} is equal to 27. Obviously, W_{21} satisfies the following double inequality equation: $0.5L_s \leq W_{21} \leq 0.6L_s$. According to the first algorithm, Ψ_0 and Ψ_3 which constitute the sequences of the best group are going to be merged with an XOR and AND gates. Their resulting output of each gate is going to be sent either to AND, OR or XOR candidate lists, depending on the number of 1's in the output sequence.
The following two cases are to be considered.

<u>Case 1</u> Merging in XOR gate

In this case, Ψ_0 and Ψ_3 are going to be merged in XOR gate. In other words, Ψ_0 and Ψ_3 are going to be replaced by their output sequence, which has the following bit stream:
Ψ_7=0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 1 0 1 0 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0
Since the number of 0's in Ψ_7 sequence is equal to 30 which is greater than 0.5 Ls, Ψ_7 should be sent to the OR candidate list. The remaining AND and OR candidate lists are as follows:

AND candidates list:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0

In this list, $W_{21} = 15 < 0.5L_s$. This inequality leads us to state that the AND gate will not be considered for merger in this case.

OR candidates list:

Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_7=0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 1 0 1 0 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0

If we consider all grouping combinations of 2, 3 or 4 sequences of this list, we find that none of them is suitable for merger because they all have W_{N0} smaller than $0.5L_s$, where N can be 2, 3, or 4. This leads us to merge all the sequences that were not grouped in either AND or OR gates into an XOR gate. Their output sequence is going to be:

Ψ_8=0 1 1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 0 1 1 0 0 1 1 1 1 0 0 0 1 0 1 1 0 1 0 1 1 1

Ψ_8 bit stream is also going to be the single output of the FSIM/ATALANTA space compactor. The space compactor (tree) is given as follows:

433 = XOR(421, 223)

434 = XOR(329, 370, 430, 431, 432, 433).

The above tree is composed of two XOR gates. The line numbers 421 and 223 which are the outputs of the CUT are going to constitute the inputs of the first XOR gate. The output of 421 and 223 after merging them in XOR gate is line number 433.

Case 2 Merging in AND gate

In this case, Ψ_0 and Ψ_3 are going to be merged in AND gate. In other words, Ψ_0 and Ψ_3 are going to be replaced by their output sequence, which has the following bit stream:

Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1

Since the number of 1's in Ψ_7 sequence is equal to 27 which is greater than 0.5 Ls, Ψ_7 should be sent to the AND candidate list. The remaining AND and OR candidate lists are as follows:

OR candidates list:

Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0
Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0

AND candidates list:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0
Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1

If we consider all grouping combinations of 2 or 3 sequences of both OR and AND lists of candidates, we find that none of them is suitable for merger because they all have W_{N0} and W_{N1} smaller than $0.5L$, respectively, where N can be 2 or 3. This leads us to merge all the sequences that were not grouped in either AND or OR gates into an XOR gate. Their output sequence is going to be:

Ψ_8=1 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0 0 0

Ψ_8 bit stream is also going to be the single output of the FSIM/ATALANTA space compactor. The space compactor (tree 2) is given as follows:

433 = AND(421, 223)

434 = XOR(329, 370, 430, 431, 432, 433).

The above tree is composed of two logical gates AND and XOR. The line numbers 421 and 223 which are the outputs of the CUT are going to constitute the inputs of the AND gate. The output of 421 and 223 after merging them in AND gate is line number 433.

In order to choose the "best tree" between tree 1 and tree 2, we run the simulation and we select the tree that gives the highest fault coverage; if the both compactors have identical fault coverage, we select the one that has the smallest CPU time. In this case, tree 2 is the best tree and it is illustrated in Chapter 5.

3.8 Implementation of the Selection of the Best Group Sub-Algorithm

In order to understand the process to obtain the "Best Group", the following definitions, categories of grouping and stages of processing are provided.

Definition and initialization of parameters

- maxW: a variable which denotes the maximum number of matching bits (can be 0 or 1 according to the kind of grouping taken into consideration) in the same bit positions obtained so far in the process of getting the BEST GROUP,
- maxN: a variable which denotes the maximum number of sequences that have maxW matching of (1's or 0's according to the kind of grouping taken into consideration) bits in common obtained so far in the process of getting the BEST GROUP.

- As a first step, we initialize both: maxW and maxN to zero.

Stages of processing

At the first stage, the "Selecting the Best Group" sub-algorithm (explained in details later in this chapter) is applied by starting with seq0 going through each of: seq1, seq2,..., seqN.

At the second stage, the "Selecting the Best Group" sub-algorithm is applied by starting with seq1 going through each of the following sequences: seq2, seq3,..., seqN. At the ith stage, the "Selecting the Best Group" algorithm is applied by starting with seq(i-1) going through each of the following sequences: seqi, seq(i+1),..., seqN.

The last stage of processing during the execution of the algorithm starts at seq(N-1) going through seq(N).

Categories of groupings

Various kind of grouping are defined and classified in the following categories:

Cat[1] Bad Grouping:

If resultingW < maxW. It means that we must exclude this sequence from the grouping under the same stage.

Cat[2] Good Grouping:

If resultingW > maxW or [resultingW = maxW and resultingN ≥ maxN]. It means that we continue processing in the same stage.

Cat[3] Better Grouping:

It has the same conditions as in Good Grouping. In addition, it is the last good grouping found in a stage. The grouping is considered as a candidate for the "best grouping".

Cat[4] Best Grouping:

It has the maximum of W and N parameters found after processing through all the stages. It is the best group found after processing all the stages.

Sub-Algorithm: How to select the "BEST GROUP"?

In stage 1, we start processing the operation with seq0.

1) To obtain the matching of 1's in each bit position we do: (seq0 AND seq1) = R_1 which has (W_{N1},N_1). Now, we compare W_{N1} (resultingW) with maxW and if needed (when W_{N1} = maxW) we compare N_1 (resultingN) with maxN, we record the highest of both, i.e. MAX[W_{N1},maxW], we put it in maxW, and we put MAX[N_1,maxN] in maxN.

- If resultingW < maxW then (seq0 , seq1) is a bad grouping according to Cat[1], and we restart 1) for seq0 and seq2.

- if (seq0,seq1) is a good grouping according to Cat[2], we can say that (seq0,seq1) is the best group so far. Then, we go to 2).

2) Now we execute: (R_1 AND seq2) = R_2 which has (W_2, N_2). Then, we compare W_2 (resultingW) with maxW and if needed (when W_{N1} = maxW) we compare N_2 (resultingN) with maxN, we record the highest of both, i.e. MAX[W_2,maxW], we put it in maxW and we put MAX[N_2,maxN] in maxN.

- If resultingW < maxW then (seq0,seq1,seq2) is a bad grouping according to Cat[1] and we restart 2) for R_1 and seq3.
- If (R_1,seq2) is a good grouping according to Cat[2], we can say that (R_1,seq2) is the best group so far. Then, we go to 3).

3) Now we execute: (R_2 AND seq3) = R_3 which has (W_3, N_3). Then, we compare W_3 (resultingW) with maxW (obtained in the step 2) and if needed (when W_3 = maxW) we compare N_3 (resultingN) with maxN (obtained in the step 2), we record the highest of both, i.e. MAX[W_2,maxW], we put it in maxW and we put MAX[N_2,maxN] in maxN.

- If resultingW < maxW (seq0,seq1,seq2,seq3) is a bad grouping according to Cat[1], and we restart 3) for R_2 and seq4.
- if (R_2,seq3) is a good grouping according to Cat[2], we can say that (R_2,seq3) is the best group so far. Then, we go to the next step.

4) We repeat the above until we exhaust all sequences starting with sequence 0. Whenever we find the "better grouping" according to Cat[3] in the first stage (steps: 1, 2, 3), we compare its resulting parameters to the first grouping of the second stage which starts with sequence 1.

5) We repeat step 4) for all the existing stages. At the last stage, we obtain the optimal grouping which is the "best grouping for the whole list of sequences" according to Cat[4] (This step is achieved using a recursive method).

Chapter 4

Designing of Space Compaction Trees for Multi-Output Circuits in BIST Based on Stochastic Dependence of Multiple Line Errors

4.1 Mergeability Criteria

By assuming stochastic dependence of errors, Li and Robinson [36] have determined the detectable error probability estimate E for a two input logic function, given two input sequences of length L_s, as follows: $E = S_1 \frac{R_1}{2\ L_s} + S_2\ \frac{R_2}{L_s}$,

where

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

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

R_1: number of single line error at the output of gate g, if gate g is used, and

R_2: number of double line error at the output of gate g, if gate g is used.

Based on the detectable error probability estimate (E) of Li and Robinson as given above, the following results can be deduced in the two-sequence case and they are well explained in [6]. Hence all theorems related to the two-sequence case are stated without proof.

Theorem 1

The detectable error probability estimate *(2nd-order)* when two sequences Ψ_1 and Ψ_2 of length Ls are merged by using an AND (NAND) gate is:

$$E_2\,(AND\,/\,NAND) = S_1 \frac{H_s + 2W'_{21}}{2L_s} + S_2 \frac{W'_{20} + W'_{21}}{L_s}$$

Where H_s is the hamming distance, i.e. the number of bit positions the two sequence differ.

Theorem 2

The detectable error probability estimate *(2nd-order)* when two sequences Ψ_1 and Ψ_2 of length L_s are merged by using an OR (NOR) gate is:

$$E_2\,(OR\,/\,NOR) = S_1 \frac{H_s + 2W'_{20}}{2L_s} + S_2 \frac{W'_{20} + W'_{21}}{L_s}$$

Theorem 3

The detectable error probability estimate *(2nd-order)* when two sequences Ψ_1 and Ψ_2 of length Ls are merged by using an XOR (XNOR) gate is:

$$E_2\,(XOR\,/\,XNOR) = S_1$$

4.2 Derivation of the Nth-Order Detectable Error Probability Estimate for Individual Gates

This study extends the mergeability criteria for merging a pair of output sequences which was established by Li and Robinson. In this section, we will determine the detectable error probability estimate E (Nth-order) for AND/NAND, OR/NOR, and XOR/XNOR gates. They are derived for the case when N is odd. The expressions for E's will be very much similar when N is even.

Let $\Psi_1, \Psi_2, ..., \Psi_N$, each of length L_s, be the N output sequences at the output of a CUT. Let the corresponding Nth-order derived sequences be $\Psi'_1, \Psi'_2, ..., \Psi'_N$, having 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, one can assume that $S_1 > S_2 > ... > S_N$ and $S_1 + S_2 + ... + S_N = 1$. Let R be the residue, that is the number of bit positions in which at least two of the sequences in the bundle have different entries. Let A_i be the number of columns in the bundle set with vertical 1-weight equal to i. Let us denote the binomial coefficient $\binom{N}{i}$ simply by the symbol C_i or C_i^N.

Theorem 4

The Nth-order detectable error probability estimate when N sequences $\Psi_1, \Psi_2, ...,$ Ψ_N are merged by using an N-input AND (NAND) gate is

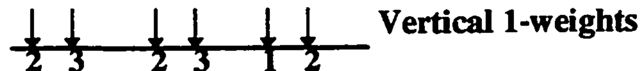
$$E_N(AND\ /\ NAND) = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N\, W_{NO} \right]$$

Example

Consider the following 5 sequences of length L_s = 12:

Ψ_1 =	1	1	1	1	0	0	0	1	0	1	0	1
Ψ_2 =	1	1	1	1	0	0	1	0	1	0	0	0
Ψ_3 =	1	1	1	1	0	0	0	0	0	1	1	0
Ψ_4 =	1	1	1	1	0	0	1	1	0	0	0	1
Ψ_5 =	1	1	1	1	0	0	0	1	1	1	0	0

→| $W_{S1} = 4$ |← →| $W_{S0} = 2$ |← →| R |←



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

Proof

$$E_N\ (AND/NAND) = \frac{S_1}{C_1^N \times L_s}\left[C_1^N \times W_{N1} + A_{N-1}\right] + \frac{S_2}{C_2^N \times L_s}\left[C_2^N \times W_{N1} + A_{N-2}\right] + \cdots + \frac{S_{N-1}}{C_{N-1}^N \times L_s}\left[C_{N-1}^N \times W_{N1} + A_{N-(N-1)}\right] + \frac{S_N}{C_N^N \times L_s}\left[C_N^N \times W_{N1} + C_N^N \times W_{N0}\right]$$

$$E_N\ (AND/NAND) = \frac{S_1}{C_1^N \times L_s}\left[C_1^N \times W_{N1}\right] + \frac{S_2}{C_2^N \times L_s}\left[C_2^N \times W_{N1}\right] + \cdots + \frac{S_{N-1}}{C_{N-1}^N \times L_s}\left[C_{N-1}^N \times W_{N1}\right] + \frac{S_N}{C_N^N \times L_s}\left[C_N^N \times W_{N1}\right] + \frac{S_1}{C_1^N \times L_s}\left[A_{N-1}\right] + \frac{S_2}{C_2^N \times L_s}\left[A_{N-2}\right] + \cdots + \frac{S_{N-1}}{C_{N-1}^N \times L_s}\left[A_1\right] + \frac{S_1}{C_N^N \times L_s}\left[C_N^N \times W_{N0}\right] =$$

$$E_N\ (AND/NAND) = \frac{W_{N1}}{L_s}\left(S_1 + S_2 + \cdots + S_{N-1} + S_N\right) + \frac{1}{L_s} \times \left[\frac{S_1}{C_1^N}\left[A_{N-1}\right] + \frac{S_2}{C_2^N}\left[A_{N-2}\right] + \cdots + \frac{S_{N-1}}{C_{N-1}^N}\left[A_1\right] + \frac{S_N}{C_N^N}\left[C_N^N \times W_{N0}\right]\right]$$

$$E_N\ (AND/NAND) = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times \Big[\frac{S_1}{C_1^N}[A_{N-1}] + \frac{S_2}{C_2^N}[A_{N-2}] + \cdots + \frac{S_{N-1}}{C_{N-1}^N}[A_1] + S_N[W_{N0}] \Big] = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times \Big[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N\ W_{N0} \Big]$$

$$E_N\ (AND/NAND) = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \Big[\frac{1}{C_1} (\ S_1\ A_{N-1} + S_{N-1}\ A_1\ ) + \frac{1}{C_2} (\ S_2\ A_{N-2} + S_{N-2}\ A_2\ ) + \cdots + S_N\ W_{N0} \Big].$$

Theorem 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:

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

Proof

$$E_N(O/N) = \frac{S_1}{C_1^N \times L_s}[C_1^N \times W_{N0} + A_1] + \frac{S_2}{C_2^N \times L_s}[C_2^N \times W_{N0} + A_2] + \cdots + \frac{S_{N-1}}{C_{N-1}^N \times L_s}[C_{N-1}^N \times W_{N0} + A_{N-1}] + \frac{S_N}{C_N^N \times L_s}[C_N^N \times W_{N0} + C_N^N \times W_{N1}]$$

$$E_N(O/N) = \frac{W_{N0}}{L_s}(S_1 + S_2 + \cdots + S_{N-1} + S_N) +$$
$$\frac{1}{L_s} \times \left[\frac{S_1}{C_1^N}[A_1] + \frac{S_2}{C_2^N}[A_2] + \cdots + \right.$$
$$\left. \frac{S_{N-1}}{C_{N-1}^N}[A_{N-1}] + \frac{S_N}{C_N^N}[C_N^N \times W_{N1}] \right]$$

$$E_N(O/N) = \frac{W_{N0}}{L_s} + \frac{1}{L_s} \times \left[\frac{S_1}{C_1^N}[A_1] + \frac{S_2}{C_2^N}[A_2] + \cdots + \frac{S_{N-1}}{C_{N-1}^N}[A_{N-1}] + S_N[W_{N1}] \right]$$
$$= \frac{W_{N0}}{L_s} + \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N W_{N1} \right]$$

$$E_N(O/N) = \frac{W_{N0}}{L_s} + \frac{1}{L_s} \left[\frac{1}{C_1}(S_1 A_1 + S_{N-1} A_{N-1}) + \right.$$
$$\left. \frac{1}{C_2}(S_2 A_2 + S_{N-2} A_{N-2}) + \cdots + S_N W_{N1} \right].$$

Theorem 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:

$$E_N(XOR/XNOR) = S_1 + S_3 + S_5 + \cdots + S_N.$$

4.3 Generalized Mergeability Criteria

Having derived the Nth-order detectable error probability estimate for individual gates, we now can establish the generalized mergeability criteria for merging an arbitrary number of output sequences under conditions of stochastic dependence of line errors.

Theorem 7

In general, for two N-input gates G_1 and G_2, G_1 is preferable to G_2 if and only if :
$E_N(G_1) > E_N(G_2)$.

4.4 Implementation

A heuristic approach has been adopted and implemented in the following algorithm in order to get results within an acceptable CPU time. The heuristic approach is very useful in this case since a closed-form expression of the Nth-order detectable error probability estimates can be computationally intensive.

4.5 The Second Algorithm

A.1) From all the output sequences, select a group of N_0 sequences having largest Nth-order 0-weight. Then, select another group of N_1 sequences having the largest Nth-order 1-weight.

A.2) Choice between the selected 1-weight and 0-weight groups

In order to make a choice between the largest group based on 1's grouping or 0's grouping let us consider the following example.

Suppose that we obtained from step A.1) a 0's grouping and a 1's grouping which have as weights and number of sequences (W_{N0}, N_0) and (W_{N1}, N_1) respectively. The selection of the best group between both groups is done according to the following rule:

a. W (Weight) overrides N (Number of sequences) which means the group that has the maximum value of W is the best solution,

b. If $W_{N0} = W_{N1}$, the best group is the group that has the largest value of N,

c. If $W_{N0} = W_{N1}$ and $N_0 = N_1$, the best group is either one. In the implementation, the Nth-order 1-weight is chosen.

A.3) Choice between (AND,OR) gates

A.3.1) If 1-Weight is chosen

- Compute W_{N1} and R/N:
  - If $W_{N1} > R/N$ then AND gate is chosen over OR, but it still has to be compared with XOR,
  - If $W_{N1} \leq R/N$ then compute both formulas:

$$E_N(AND/NAND) = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N W_{N0} \right]$$

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

- Then compare them and:

  *If E_N[AND/NAND] > E_N[OR/NOR] then AND gate is chosen over OR, but it still has to be compared with XOR,

  *If E_N[AND/NAND] < E_N[OR/NOR] then OR gate is chosen over AND, but it still has to be compared with XOR,

  *If E_N[AND/NAND] = E_N[OR/NOR] then either AND or OR can be chosen, but the chosen gate still has to be compared with XOR gate. In the implementation, AND gate was chosen in this situation.

A.3.2) If 0-Weight is chosen

- Compute W_{N0} and R/N:
  - If W_{N0} > R/N then OR gate is chosen over AND, but it still has to be compared with XOR,
  - If $W_{N0} \leq$ R/N then compute both formulas:

$$E_N(AND / NAND) = \frac{W_{N1}}{L_s} + \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N\, W_{N0} \right]$$

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

- Then compare them and:

  *If E_N[AND/NAND] > E_N[OR/NOR] then AND gate is chosen over OR, but it still has to be compared with XOR,

  *If E_N[AND/NAND] < E_N[OR/NOR] then OR gate is chosen over AND, but it still has to be compared with XOR,

  *If E_N[AND/NAND] = E_N[OR/NOR] then either AND or OR can be chosen, but the chosen gate still has to be compared with XOR gate. In the implementation (C program), AND gate was chosen in this situation.

A.4) Choice between (Above-chosen-gate, XOR) gates

A.4.1) Case 1 (AND, XOR):

a) compute: $A = W_{N1}/L_s - (S_1 + S_3 + \dots + S_N)$ for N odd,

$A = W_{N1}/L_s - (S_1 + S_3 + \ldots + S_{N-1})$ for N even.

b) if $A \geq 0$ then use AND gate (nonexhaustive approach is fulfilled).

c) if $A < 0$ then

$$\text{compute } B = \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N W_{N0} \right]$$

then compute $(A+B) = C$,

d) if $C > 0$ then use AND gate,

e) if $C < 0$ then use XOR gate,

f) if $C = 0$ then use either AND or XOR. In the implementation XOR gate was chosen in this situation.

A.4.2) Case 2 (OR, XOR):

a) compute $A = W_{N0}/L_s - (S_1 + S_3 + \ldots + S_N)$ for N odd,

$A = W_{N0}/L_s - (S_1 + S_3 + \ldots + S_{N-1})$ for N even.

b) if $A \geq 0$ then use OR gate (nonexhaustive approach is fulfilled).

c) if $A < 0$ then

$$\text{compute } B = \frac{1}{L_s} \times \left[\sum_{i=1}^{N-1} \frac{S_i}{C_i}[A_i] + S_N W_{N1} \right]$$

then compute $(A+B) = C$,

d) if $C > 0$ then use OR gate,

e) if $C \leq 0$ then use XOR gate,

f) if $C = 0$ then use either OR or XOR. In the implementation XOR gate was chosen in this situation.

A.5) Apply the above procedure to the merging sequence and all remaining sequences until we end up with only one sequence at the output.

Example 1

Consider the following six sequences:

$\Psi_1 = 0000 \quad 0011 \quad 11$

$\Psi_2 = 0000 \quad 0011 \quad 11$

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

$\Psi_5 = 0000 \quad 0011 \quad 01$

$\Psi_6 = 0000 \quad 0000 \quad 00$

By following the first proposed algorithm, we are going to keep on merging the above sequences (which constitute the outputs of a CUT) until we obtain a single output of the space compactor. This can be achieved as follows:

First gate merger

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

Here, the number of matching 1's is 6, and the number of sequences is 2.

OR group:

$\Psi_5 = 0000 \quad 0011 \quad 01$

$\Psi_6 = 0000 \quad 0000 \quad 00$

Here, the number of matching 0's is 7, and the number of sequences is 2.

Definition of variables:

A_i: the number of columns with vertical 1-weight equals to i.

S_i: the probability of the ith-line error occurrence is equal to $2/3[1/3^{(i-1)}]$ for $1 \leq i \leq (N-1)$, and it is equal to $1/3[1/3^{(i-2)}]$ for $i = N$.

According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is OR group, where: $W_{20} \equiv A[0] = 7$, $A[1] = 3$, $W_{21} \equiv A[2] = 0$, $R = 3$, $N = 2$, $L_s = 10$. In addition, grouping can be determined as follows: Since $W_{N0} - (R/N) = 5.5 > 0$, OR gate is chosen over AND gate. OR still needs to be compared with XOR gate.

Choice between (OR, XOR) gates for merger:

Since N is even (N=2) then, $A[0]/L_s - (S_1 + ... + S_{N-1}) = 0.033 > 0$. Thus, use OR gate for merger of the group in question. This resulting sequence after merging both sequences Ψ_5 and Ψ_6 in OR gate is $\Psi_7 = 0000001101$. Sequences Ψ_5 and Ψ_6 are going to be deleted from the pool of sequences and their output Ψ_7 will be added to the rest of the sequences.

Second gate merger

Ψ_7 represents the newly added sequence. The remaining sequences are as follows:

$\Psi_1 = 0000 \quad 0011 \quad 11$

$\Psi_2 = 0000 \quad 0011 \quad 11$

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

$\Psi_7 = 0000 \quad 0011 \quad 01$

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

Here, the number of matching 1's is 6, and the number of sequences is 2.

OR group:

$\Psi_7 = 0000 \quad 0011 \quad 01$

$\Psi_1 = 0000 \quad 0011 \quad 11$

$\Psi_2 = 0000 \quad 0011 \quad 11$

Here, the number of matching 0's is 6, and the number of sequences is 3.

According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is OR group, where: $W_{30} \equiv A[0] = 6$, $A[1] = 1$, $A[2] = 1$, $W_{31} \equiv A[3] = 3$, $R = 1$, $N = 3$, $L_s = 10$.

Since W_{N0} - $R/N = 5.66 > 0$, OR gate is chosen over AND gate. OR still needs to be compared with XOR gate.

Choice between (OR, XOR) gates for merger:

Since N is odd (N=3), $A = W_{N0}/L_s - (S_1+...+S_N) = -0.178 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B \equiv Bor = 0.041$, $C = -0.137 < 0$. Merge sequences (Ψ_7, Ψ_1, Ψ_2) in XOR gate. The resulting sequence of the XOR operation is $\Psi_8 = 0000001101$. Sequences Ψ_7, Ψ_1 and Ψ_2 are going to be replaced by their output Ψ_8.

Third gate merger

Ψ_8 represents the newly added sequence. The remaining sequences are as follows:

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

$\Psi_8 = 0000 \quad 0011 \quad 01$

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0- weight. These groups are:

AND group:

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

Here, the number of matching 1's is 6, and the number of sequences is 2.

OR group:

$\Psi_3 = 1111 \quad 1100 \quad 00$

$\Psi_4 = 1111 \quad 1100 \quad 00$

Here, the number of matching 0's is 4, and the number of sequences is 2.

According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is AND group, where: $W_{20} \equiv A[0] = 4$, $A[1] = 0$, $W_{21} \equiv A[2] = 6$, , $R = 0$, $N = 2$, $L_s = 10$.

Since $W_{N1} - R/N = 6.0 > 0$, AND gate is chosen over OR gate. AND still needs to be compared with XOR gate.

Choice between (AND, XOR) gates for merger:

Since N is even (N=2), $A = W_{N1}/L_s - (S_1 + \ldots + S_{N-1}) = -0.067 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B \equiv Band = 0.133$, $C = 0.067 > 0$. Merge sequences (Ψ_3, Ψ_4) in AND gate. The resulting sequence of the AND operation is $\Psi_9 = 1111110000$. Sequences Ψ_3 and Ψ_4 are going to be replaced by their output Ψ_9.

<u>Fourth gate merger</u>

Ψ_9 represents the newly added sequence. The remaining sequences are as follows:

$\Psi_9 = 1111 \quad 1100 \quad 00$

$\Psi_8 = 0000 \quad 0011 \quad 01$

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

$\Psi_9 = 1111 \quad 1100 \quad 00$

$\Psi_8 = 0000 \quad 0011 \quad 01$

Here, the number of matching 1's is 0, and the number of sequences is 2.

OR group:

$\Psi_9 = 1111 \quad 1100 \quad 00$

$\Psi_8 = 0000 \quad 0011 \quad 01$

Here, the number of matching 0's is 1, and the number of sequences is 2.

According to rule which allows us to choose between 1-weight and 0-weight groups, the best group is OR group, where: $W_{20} \equiv A[0] = 1$, $A[1] = 9$, $W_{21} \equiv A[2] = 0$, $R = 9$, $N = 2$, $L_s = 10$.

Since $W_{N0} - R/N = 1 - (9/2) = -3.5 < 0$, compute and compare both formulas of E[AND] and E[OR], as follows:

E[AND] = 0.300 + 0.033 = 0.333,

E[OR] = 0.367 + 0.033 = 0.400

Since E[OR] > E[AND], OR gate is chosen over AND. OR still needs to be compared with XOR gate to decide on the final merger.

Choice between (OR, XOR) gates for merger:

Since N is even (N=2), $A = W_{N0}/L_s - (S_1 + \ldots + S_{N-1}) = -0.567 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B = 0.30$, $C = -0.267 < 0$. Merge sequences (Ψ_8, Ψ_9) in XOR gate. The resulting sequence of the XOR operation is $\Psi_{10} = 1111111101$, which is the last and single output obtained after the series of sequence groupings.

<u>Example 2</u> Circuit C432

The output sequences of the CUT for circuit C432 are as follows:

Ψ_0=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1

Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0

Ψ_3=1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1

Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0

Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0

By following the second proposed algorithm, we are going to keep on merging the above sequences until we obtain a single output of the FSIM/ATALANTA space compactor. This can be achieved as follows:

<u>First gate merger</u>

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_0=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1

Ψ_3=1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1

Here, the number of matching 1's is 27, and the number of sequences is 2.

OR group:

Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0

Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0

Here, the number of matching 0's is 16, and the number of sequences is 2.

According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is AND group, where: $W_{20} \equiv A[0] = 3$,

A[1] = 16, W_{21}≡A[2] = 27, R = 46-3-27 = 16, N = 2, L_s = 46. This result can also be obtained, as follows:

Since $W_{N1} - (R/N) = 19.00 > 0$, AND gate is chosen over OR gate. AND still needs to be compared with XOR gate.

Choice between (AND, XOR) gates for merger:

Since N is even (N=2), $A = W_{N1}/L_s - (S_1+...+S_{N-1}) = -0.079 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

B = 0.137, C = 0.058 > 0. Merge sequences (Ψ_0, Ψ_3) in AND gate. The resulting sequence of the AND operation is Ψ_7. Sequences Ψ_3 and Ψ_4 are going to be replaced by their output Ψ_7. Sequence Ψ_7 is as follows:

Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1

<u>**Second gate merger**</u>

Ψ_7 represents the newly added sequence. The remaining sequences are as follows:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0
Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0
Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0

Here, the number of matching 1's is 15, and the number of sequences is 2.

OR group:

Ψ_4=0 1 1 1 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0
Ψ_5=0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 0 1 0

Here, the number of matching 0's is 16, and the number of sequences is 2.

According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is OR group, where: $W_{30} \equiv A[0] = 16$, $A[1] = 20$, $A[2] = 10$, $R = 20$, $N = 2$, $L_s = 46$. This result can be obtained as follows:

Since W_{N0} - $R/N = 6.00 > 0$, OR gate is chosen over AND gate. OR still needs to be compared with XOR gate.

Choice between (OR, XOR) gates for merger:

Since N is even (N=2), $A = W_{N0}/L_s - (S_1+...+S_{N-1}) = -0.319 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B \equiv B_{or} = 0.217$, $C = -0.101 < 0$. Merge sequences (Ψ_4, Ψ_5) in XOR gate. The resulting sequence of the XOR operation is Ψ_8. Sequences Ψ_4 and Ψ_5 are going to be replaced by their output Ψ_8 which is as follows:

Ψ_8=0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

<u>**Third gate merger**</u>

The remaining sequences are as follows:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0
Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0
Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1
Ψ_8=0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_1=1 0 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1
Ψ_2=1 0 1 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0

Here, the number of matching 1's is 15, and the number of sequences is 2.

OR group:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Here, the number of matching 0's is 15, and the number of sequences is 2.

In this case, $W_{20} \equiv A[0] = 8$, $A[1] = 23$, $W_{21} \equiv A[2] = 15$, $R = 23$, $N = 2$, $L_s = 46$. According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is the AND group. AND still needs to be compared with XOR gate.

Choice between (AND, XOR) gates for merger:

Since N is even (N=2), $A = W_{N1}/L_s - (S_1+...+S_{N-1}) = -0.34 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B = 0.224$, $C = -0.116 < 0$. Merge sequences (Ψ_1, Ψ_2) in XOR gate. The resulting sequence of the AND operation is Ψ_9. Sequences Ψ_1 and Ψ_2 are going to be replaced by their output Ψ_9 which has the following bit stream:

Ψ_9=0 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 1 1 1 1

<u>Fourth gate merger</u>

The remaining sequences are as follows:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0
Ψ_9=0 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 1 1 1 1

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_7=1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1 1
Ψ_9=0 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 1 1 1 1

Here, the number of matching 1's is 15, and the number of sequences is 2.

OR group:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Here, the number of matching 0's is 15, and the number of sequences is 2.

In this case, $W_{20} \equiv A[0] = 11$, $A[1] = 20$, $W_{21} \equiv A[2] = 15$, $R = 20$, $N = 2$, $L_s = 46$. According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is the AND group. AND still needs to be compared with XOR gate.

Choice between (AND, XOR) gates for merger:

Since N is even (N=2), $A = W_{N1}/L_s - (S_1+...+S_{N-1}) = -0.34 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B = 0.224$, $C = -0.116 < 0$. Merge sequences (Ψ_7, Ψ_9) in XOR gate. The resulting sequence of the XOR operation is Ψ_{10}. Sequences Ψ_7 and Ψ_9 are going to be replaced by their output Ψ_{10} which has the following bit stream:

Ψ_{10}=1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0

<u>Fifth gate merger</u>

The remaining sequences are as follows:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Ψ_{10}=1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Here, the number of matching 1's is 9, and the number of sequences is 2.

OR group:

Ψ_6=0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0
Ψ_8=0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0

Here, the number of matching 0's is 15, and the number of sequences is 2.

In this case, $W_{20} \equiv A[0] = 15$, $A[1] = 22$, $W_{21} \equiv A[2] = 9$, $R = 22$, $N = 2$, $L_s = 46$. According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is the OR group. OR still needs to be compared with XOR gate.

Choice between (OR, XOR) gates for merger:

Since N is even (N=2), $A = W_{N0}/L_s - (S_1+...+S_{N-1}) = -0.34 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B = 0.224$, $C = -0.116 < 0$. Merge sequences (Ψ_6, Ψ_8) in XOR gate. The resulting sequence of the XOR operation is Ψ_{11}. Sequences Ψ_6 and Ψ_8 are going to be replaced by their output Ψ_{11} which has the following bit stream:

Ψ_{11}=0 1 1 0 1 1 0 0 1 1 1 0 0 1 0 1 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0

<u>**Sixth and last gate merger**</u>

The remaining sequences are as follows:

Ψ_{10}=1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0
Ψ_{11}=0 1 1 0 1 1 0 0 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0

Choice between (AND, OR) gates:

We start by selecting N_1 sequences having the largest Nth-order 1-weight. Then we select N_0 sequences having the largest Nth-order 0-weight. These groups are:

AND group:

Ψ_{10}=1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0
Ψ_{11}=0 1 1 0 1 1 0 0 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0

Here, the number of matching 1's is 12, and the number of sequences is 2.

OR group:

Ψ_{10}=1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0
Ψ_{11}=0 1 1 0 1 1 0 0 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0

Here, the number of matching 0's is 16, and the number of sequences is 2.

In this case, $W_{20} \equiv A[0] = 16$, $A[1] = 18$, $W_{21} \equiv A[2] = 12$, $R = 18$, $N = 2$, $L_s = 46$. According to the best grouping rule which allows us to choose between 1-weight and 0-weight groups, the best group here is the OR group. OR still needs to be compared with XOR gate.

Choice between (OR, XOR) gates for merger:

Since N is even (N=2), $A = W_{N0}/L_s - (S_1+...+S_{N-1}) = -0.318 < 0$. According to the algorithm, we should compute B followed by computation of $(A+B) \equiv C$,

$B = 0.217$, $C = -0.101 < 0$. Merge sequences (Ψ_{10}, Ψ_{11}) in XOR gate. The resulting sequence of the XOR operation is Ψ_{12}. Sequences Ψ_{10} and Ψ_{11} are going to be replaced by their output Ψ_{12} which is the singled output of the space compactor. Ψ_{12} has the following bit stream:

Ψ_{12}=1 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0 0 0

The space compactor for circuit C432 is composed of the following logical gates:

433 = AND(421, 223)

434 = XOR(431, 430)

435 = XOR(370, 329)

436 = XOR(435, 433)

437 = XOR(434, 432)

438 = XOR(437, 436)

The above tree is composed of an AND gate and five XOR gates. This space compactor is illustrated in Chapter 5.

Chapter 5

Experimental Results and Discussions

5.1 Simulation Method

The generalized mergeability criteria were used in order to design space compressors for all the ISCAS 85 combinational circuits. The compressor circuits were composed of a series of AND, OR and XOR gates which were determined appropriately by using the gate selection criteria previously explained.
In order to demonstrate the feasibility of our approach, independent simulations were performed on all the ISCAS 85 combinational circuits using ATALANTA, FSIM and COMPACTEST. We have used ATALANTA [35] (a logic and fault simulation program developed at Virginia Polytechnic Institute & State University) to generate the fault-free output sequences needed to construct our space compactor and to test the benchmark circuits using reduced test sets accompanied with a random test session. In addition, we have used FSIM [34] as a fault simulation program to generate pseudorandom test sets, and COMPACTEST [42] program to generate reduced test sets that detect most detectable single stuck-at faults for all benchmark circuits.

For comparison purposes, we used a parity tree compactor, composed of XOR gates, that propagates all errors that appear on an odd number of inputs and is , therefore, considered ideal for space compaction. Having parity tree space compactor as a reference, we simulated some combinational benchmark circuits to demonstrate the feasibility of the proposed scheme of constructing compaction trees using the concept of Nth-order of sequence weights.

As a result of the conducted simulation and for some combinational circuits, we obtained several space compressors (trees). In such situation, we selected the "best tree", which is the tree that gave the highest fault coverage; when two or more trees had identical fault coverages, we selected the one that gave the smallest CPU time.
For each ISCAS 85 combinational circuit, several variables were determined and included in tabular format. These variables are: the number of test vectors used to construct the best compaction tree, the CPU time taken to construct the best compaction tree, the number of applied test vectors corresponding to the best tree obtained, the simulation CPU time and the percentage of fault coverage by running ATALANTA, FSIM and COMPACTEST programs on a Sparc 5 SUN workstation.

5.2 Classification of the Simulated Results

All the simulation results obtained from FSIM, ATALANTA and COMPACTEST are classified in tables and presented in the next paragraph under the following four categories: simulations without using compactors, simulations by assuming stochastic independence of line errors, simulations by considering stochastic dependence of line errors, and simulations by using parity tree as a space compactor. In addition, we estimated the hardware overhead for all the circuits in two tables. The first table determines the percentage of hardware overhead for FSIM and ATALANTA while the second determines the same percentage for COMPACTEST.

5.3 Simulation Results

5.3.1 Simulation Results Without Using Compactors

Using FSIM, ATALANTA and COMPACTEST as fault simulators, we determined the fault coverage and the CPU simulation time required for all ISCAS 85

benchmark circuits, without using compactors. The results are shown in Tables 5.1, 5.2 and 5.3 respectively.

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	32	0.26	100.00
c432	224	0.31	98.60
c499	224	0.41	96.00
c880	224	0.43	95.50
c1355	224	0.50	91.99
c1908	224	0.68	84.03
c2670	224	0.76	82.27
c3450	224	1.35	87.22
c5315	224	1.15	96.71
c6288	224	3.00	99.56
c7552	224	1.73	90.79

Table 5.1 Simulation results of the ISCAS 85 benchmark circuits using **FSIM without compactors.**

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	8	0.03	100.00
c432	70	0.45	99.20
c499	73	0.30	98.94
c880	110	0.95	100.00
c1355	104	0.86	99.49
c1908	177	2.78	99.52
c2670	190	7.31	95.74
c3450	240	11.78	96.00
c5315	210	5.45	98.89
c6288	52	19.00	99.56
c7552	358	35.25	98.25

Table 5.2 Simulation results of the ISCAS 85 benchmark circuits using **ATALANTA without compactors.**

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	4	0.00	100.00
c432	44	5.17	99.23
c499	65	12.07	98.94
c880	30	2.93	100.00
c1355	96	15.75	99.49
c1908	137	16.37	99.52
c2670	68	116.09	95.74
c3450	110	266.70	95.91
c5315	55	36.54	98.89
c6288	16	73.51	99.56
c7552	85	151.04	98.26

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

From the above tables, we can readily conclude that ATALANTA and COMPACTEST provide almost similar fault coverage simulation results. These results are much higher than what is provided by the FSIM simulations for all ISCAS 85 circuits.

Since all the simulations were conducted on Sparc 5 SUN workstation therefore, it is perfectly legitimate to compare the CPU time of all the simulators. Having done that, we notice that FSIM provides the smallest (best) CPU simulation time for almost all circuits, while COMPACTEST provides by far the highest (worst) CPU time than ATALANTA except for circuit c17.

5.3.2 Simulation Results by Assuming Stochastic Independence

Tables 5.4, 5.5 and 5.6 show the simulation results for all ISCAS 85 benchmark circuits by assuming stochastic independence of multiple line errors using FSIM, ATALANTA and COMPACTEST respectively. The number of space compressor obtained for each circuit is indicated in the last column of each table. In the case where several space compactors are obtained, we selected the space compactor (tree) that gave

the highest fault coverage; when two or more compactors had identical fault coverages we selected the one that has the smallest CPU time.

It is worth mentioning that the CPU time taken to construct the best compaction tree is obtained by dividing the total CPU time computed for all trees which is obtained from the C program simulation by the total number of trees obtained for a specific ISCAS 85 benchmark circuit.

Circuit name	No of test vectors used to construct the best compaction tree	CPU time taken to construct the best compaction tree [secs]	No of applied test vectors corresponding to best tree in the simulation results	CPU simulation time [secs]	Fault coverage [%]	Total # of obtained trees
c17	7	0.12	32	0.17	100.00	1
c432	46	0.15	224	0.23	90.30	2
c499	54	0.75	224	0.25	93.16	1
c880	56	0.25	224	0.35	93.91	24
c1355	86	1.50	224	0.38	88.89	1
c1908	115	1.63	224	0.60	82.08	1
c2670	108	31.69	224	0.73	80.18	128
c3450	144	3.63	224	1.47	83.97	1
c5315	117	70.00	224	1.23	89.84	24
c6288	31	0.25	224	4.68	99.56	8
c7552	217	67.38	224	1.97	89.33	214

Table 5.4 Simulation results and values used to construct the best compaction tree 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 best compaction tree	CPU time taken to construct the best compaction tree [secs]	No of applied test vectors corresponding to best tree in the simulation results	CPU simulation time [secs]	Fault coverage [%]	Total # of obtained trees
c17	7	0.12	6	0.03	100.00	1
c432	46	0.15	47	0.88	93.75	2
c499	54	0.75	54	0.40	100.00	1
c880	56	0.25	57	1.05	98.00	24
c1355	86	1.50	83	0.98	99.62	1
c1908	115	1.63	113	3.38	98.88	1
c2670	108	31.69	64	42.12	87.15	128
c3450	144	3.63	169	12.70	95.83	1
c5315	117	70.00	104	25.17	94.26	24
c6288	31	0.25	41	19.95	99.56	8
c7552	217	67.38	141	68.93	95.71	214

Table 5.5 Simulation results and values used to construct the best compaction tree 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 best compaction tree	CPU time taken to construct the best compaction tree [secs]	No of applied test vectors corresponding to best tree in the simulation results	CPU simulation time [secs]	Fault coverage [%]	Total # of obtained trees
c17	4	0.17	6	0.02	100.00	1
c432	68	0.13	41	15.04	94.85	2
c499	63	0.63	22	95.82	84.17	20
c880	30	0.13	40	32.11	95.22	10
c1355	96	0.73	45	260.71	86.91	320
c1908	137	2.52	31	760.63	81.21	2
c2670	68	16.79	56	962.31	83.11	101
c3450	110	1.68	115	807.28	90.40	4
c5315	55	9.14	24	2043.18	76.49	48
c6288	16	0.04	75	993.34	99.43	246
c7552	85	9.56	116	726.08	96.69	92

Table 5.6 Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using **COMPACTEST and assuming stochastic independence of multiple line errors.**

In the case of stochastic independence, obviously ATALANTA provides the best fault coverage results among all the simulators. FSIM provides the best CPU simulation time followed by ATALANTA.

5.3.3 Simulation Results by Assuming Stochastic Dependence

By assuming stochastic dependence of line errors, we obtained the following tables which represent the simulation results obtained from the C program as well as the simulation results obtained from FSIM, ATALANTA and COMPACTEST.

Circuit name	No of test vectors used to construct the compactor (c prog.)	CPU time taken to construct the compactor from (c prog.) [secs]	No of applied test vectors simulation results	CPU time simulation time [secs]	Fault coverage [%]
c17	7	0.12	32	0.15	100.00
c432	46	0.25	224	0.25	94.38
c499	54	10.10	224	0.23	94.51
c880	56	7.53	224	0.30	94.62
c1355	86	25.73	224	0.45	89.98
c1908	115	21.00	224	0.70	84.02
c2670	108	2503.87	224	0.87	79.03
c3450	144	22.65	224	1.78	84.99
c5315	117	3790.17	224	1.50	84.96
c6288	31	4.03	224	5.00	99.47
c7552	217	4560.80	224	2.58	87.91

Table 5.7 Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using **FSIM and assuming stochastic dependence of multiple line errors.**

Circuit name	No of test vectors used to construct the compactor (c prog.)	CPU time taken to construct the compactor from (c prog.) [secs]	No of applied test vectors simulation results	CPU time simulation time [secs]	Fault coverage [%]
c17	7	0.12	6	0.03	100.00
c432	46	0.25	67	0.78	98.50
c499	54	10.10	53	1.27	96.10
c880	56	7.53	70	1.65	99.07
c1355	86	25.73	85	2.33	98.04
c1908	115	21.00	145	4.58	98.91
c2670	108	2503.87	122	42.63	87.80
c3450	144	22.65	169	19.08	95.12
c5315	117	3790.17	148	26.43	89.91
c6288	31	4.03	67	36.35	99.56
c7552	217	4560.80	161	127.82	94.66

Table 5.8 Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using **ATALANTA and assuming stochastic dependence of multiple line errors.**

Circuit name	No of test vectors used to construct the compactor (c prog.)	CPU time taken to construct the compactor from (c prog.) [secs]	No of applied test vectors simulation results	CPU time simulation time [secs]	Fault coverage [%]
c17	4	0.10	6	0.01	100.00
c432	44	0.25	65	9.86	98.28
c499	63	3108.00	87	32.45	98.81
c880	30	3.43	57	7.40	99.05
c1355	96	29.75	109	37.23	99.24
c1908	137	29.35	168	134.66	98.83
c2670	68	1016.03	117	354.129	89.36
c3450	110	12.80	162	624.97	94.49
c5315	55	9626.33	139	321.87	96.51
c6288	16	2.12	70	112.62	99.56
c7552	85	789.53	129	1313.04	93.09

Table 5.9 Simulation results and values used to construct the best compaction tree of the ISCAS 85 benchmark circuits using **COMPACTEST and assuming stochastic dependence of multiple line errors.**

It is worth noting here that for each grouping, S_i which is the probability of the ith-line error occurrence is equal to $2/3[1/3^{(i-1)}]$ for $1 \leq i \leq (N-1)$, and it is equal to $1/3[1/3^{(i-2)}]$ for $i = N$. This operation is computed for each grouping encountered during the execution of the algorithm described in Chapter 4.

As far as fault coverage concerns, ATALANTA provides the best results, while FSIM provides the best results in terms of CPU simulation time.

5.3.4 Simulation Results by Using Parity Tree as a Space Compactor

We also simulated all ISCAS 85 benchmark circuits with a parity tree space compactor using FSIM, ATALANTA and COMPACTEST. The obtained simulation results are framed in Tables 5.10, 5.11 and 5.12, respectively.

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	32	0.26	100.00
c432	224	0.51	95.14
c499	224	0.38	94.60
c880	224	0.55	93.10
c1355	224	0.50	90.80
c1908	224	0.86	81.88
c2670	224	0.83	67.90
c3450	224	1.71	85.70
c5315	224	1.45	95.06
c6288	224	4.91	99.46
c7552	224	2.41	88.00

Table 5.10 Simulation results of the ISCAS 85 benchmark circuits using **FSIM with parity tree.**

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	9	0.03	100.00
c432	104	0.88	99.20
c499	63	1.13	96.09
c880	107	1.38	99.29
c1355	102	2.50	98.04
c1908	225	4.90	98.91
c2670	120	76.10	68.95
c3450	285	19.00	94.98
c5315	271	8.78	98.57
c6288	87	30.50	99.52
c7552	332	114.00	95.15

Table 5.11 Simulation results of the ISCAS 85 benchmark circuits using **ATALANTA with parity tree.**

Circuit name	No of applied test vectors	CPU simulation time [secs]	Fault coverage [%]
c17	6	0.02	100.00
c432	63	5.73	99.24
c499	61	56.73	95.78
c880	47	7.75	99.26
c1355	96	64.27	97.97
c1908	157	89.45	98.19
c2670	61	819.14	78.81
c3450	139	639.02	94.46
c5315	82	203.56	98.71
c6288	73	1722.42	99.41
c7552	13	10728.89	62.05

Table 5.12 Simulation results of the ISCAS 85 benchmark circuits using **COMPACTEST with parity tree.**

By comparing the fault coverage of all the simulators, ATALANTA in general provides the best results followed by COMPACTEST. In terms of the CPU time results, FSIM provides the smallest (best) results followed by ATALANTA.

From the simulation experiments, it is obvious that in all cases, our space compactor is comparable in all respects with the parity tree space compactor. For some circuits, we obtained better fault coverage with reduction in CPU time using our space compactor than what we have when a parity tree compactor is used.

5.4 Hardware Overhead

Tables 5.13 and 5.14 show the hardware overhead estimates for all ISCAS 85 benchmark circuits corresponding to ATALANTA/FSIM and COMPACTEST respectively.

In order to estimate the hardware overhead, we used the ratio of the weighted gate count metric, that is, average fanins multiplied by the number of gates, of the compactor and that of the total circuit comprised of the CUT and the space compactor.

Circuit name	Total No of fanins in the best compactor	Total No of gates in the best compactor	Average fanins in the best compactor	Total No of gates in CUT	Average fanin in CUT	H/W overhead [%]
c17	2	1	2	6	2.00	14.29
c432	8	2	4	160	2.10	1.62
c499	32	1	32	202	2.02	0.93
c880	39	14	2.79	383	1.90	4.19
c1355	32	1	32	546	1.95	0.34
c1908	25	1	25	880	1.70	0.21
c2670	162	23	7.04	1269	1.64	2.89
c3450	25	4	6.25	1669	1.76	0.37
c5315	169	47	3.59	2307	1.90	2.61
c6288	35	4	8.75	2416	1.99	0.27
c7552	127	20	6.35	3513	1.75	0.89

Table 5.13 Estimates of the **hardware overhead for ATALANTA or FSIM.**

Circuit name	Total No of fanins in the best compactor	Total No of gates in the best compactor	Average fanins in the best compactor	Total No of gates in CUT	Average fanin in CUT	H/W overhead [%]
c17	2	1	2	6	2.00	14.29
c432	8	2	4	160	2.10	1.62
c499	57	26	2.19	202	2.02	11.8
c880	37	12	3.08	383	1.90	3.75
c1355	50	19	2.63	546	1.95	3.86
c1908	26	2	13	880	1.70	0.40
c2670	256	122	2.10	1269	1.64	0.98
c3450	28	7	4	1669	1.76	0.01
c5315	180	58	3.10	2307	1.90	0.03
c6288	51	21	2.43	2416	1.99	0.01
c7552	208	101	2.06	3513	1.75	0.03

Table 5.14 Estimates of the **hardware overhead for COMPACTEST.**

As readily can be seen from the above tables, the hardware overhead of the best compactor for all the ISCAS 85 benchmark circuits is as small as (0.2-4.2)% in the case of ATALANTA and FSIM, and equals to 14.3% for circuit c17 (the space compactor of this circuit is composed of one gate only). For the case of COMPACTEST simulator, the hardware overhead is even smaller and it ranges from 0.01 to 3.9% and equals to 14.3% for circuit c17.

5.5 Compaction Circuits for C432

To give an idea about how our space compactor looks like, we provide the compaction circuits for the benchmark circuit C432 with 160 gates, 36 primary inputs, and 7 primary outputs, corresponding to stochastic independence and dependence (according to the formula provided in Chapter 4) of line errors.

Figure 5.1 illustrates the FSIM/ATALANTA space compactor logical circuit assuming stochastic independence of multiple line errors.

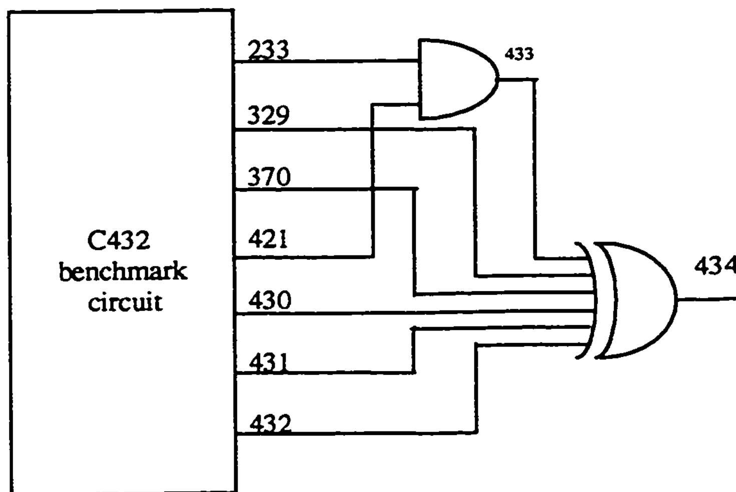


Figure 5.1 Space compactor circuit for C432 assuming stochastic independence of line errors.

Figure 5.2 illustrates the FSIM/ATALANTA space compactor circuit for C432 assuming stochastic dependence of multiple line errors. It is worth mentioning here that, by using the algorithm given in Chapter 4 to conduct the simulation, we obtain a grouping of two sequences for all the small ISCAS 85 circuits. For the large circuits such as C2670, C7552, we obtained groupings of two, three, and four sequences. In the case of stochastic dependence of errors, the space compactor logical circuit depends on the values assigned to S_i. Therefore, changing S_i could change the gate composition of the corresponding logical circuit.

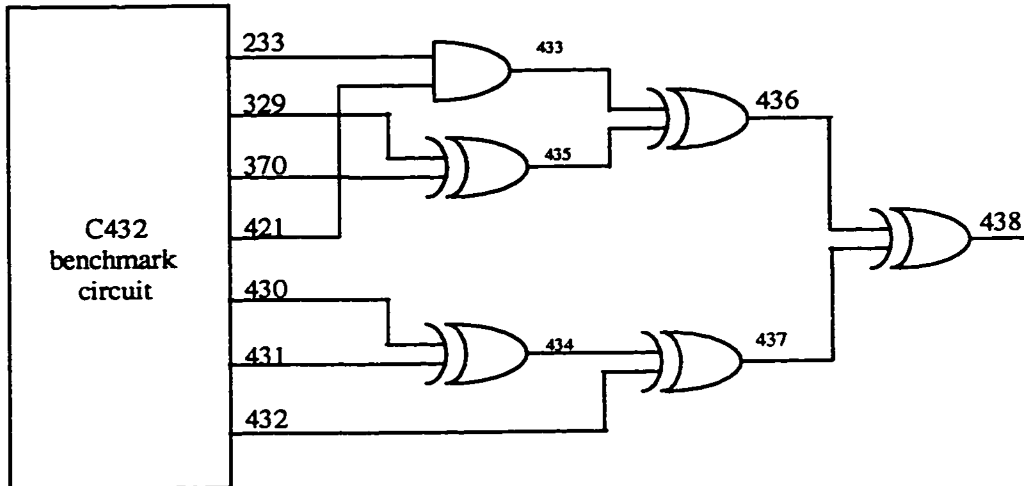


Figure 5.2 Space compactor circuit for C432 assuming stochastic dependence of line errors.

It is worth mentioning here that if we used COMPACTEST we would end up with the same logical gates as in both Figures 5.1 and 5.2. The only difference is that the input lines for each gate would be different than the ones used in the case of FSIM/ATALANTA logical circuits.

The hardware overhead for the space compactor shown in Figure 5.1 (which is composed of only two gates), measured by the weighted gate count metric (gate count x average fanin), is 1.62%. It is almost 35% less than what was recently published by Assaf [6] where he obtained 3.44% as a hardware overhead.

5.6 Graphical Representation of the Simulation Results

This section is divided into the following three subsections.

5.6.3 Comparison of the Fault Coverage of All Circuits in the Cases of Parity Tree, Stochastic Independence and Dependence of Line Errors

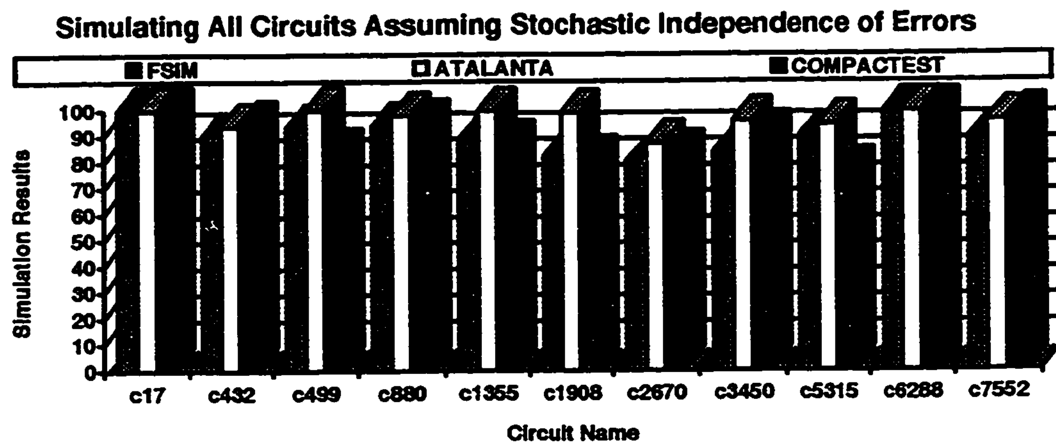


Figure 5.3 Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by assuming stochastic independence of multiple line errors.

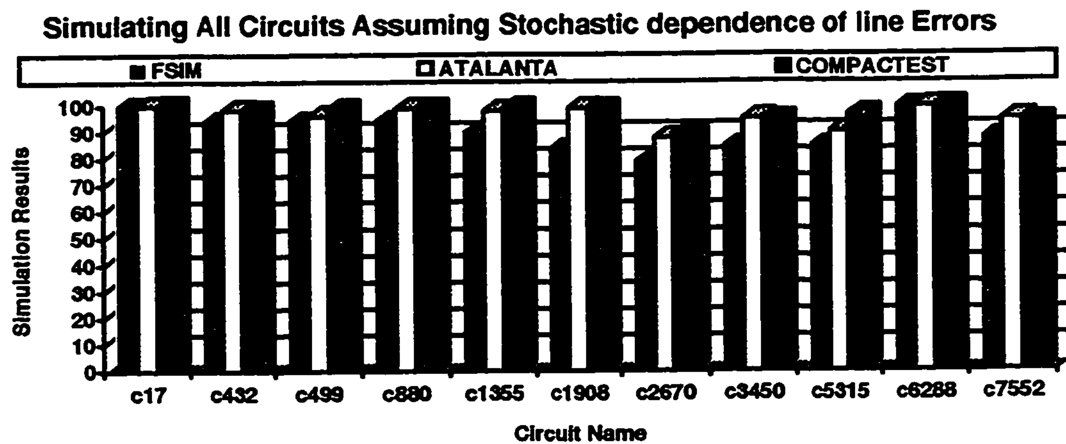


Figure 5.4 Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by assuming stochastic dependence of multiple line errors.

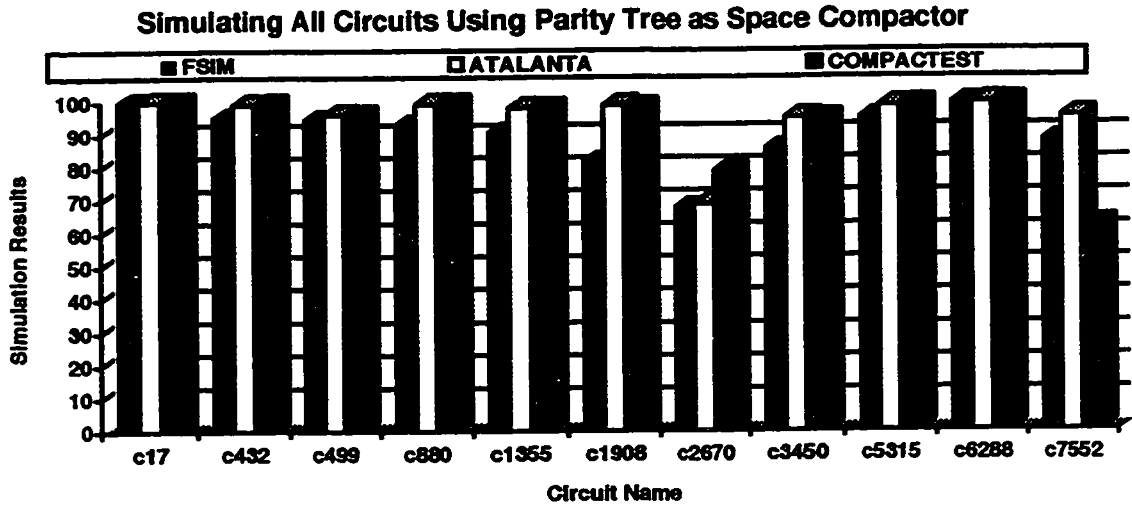


Figure 5.5 Comparison of all fault coverage obtained using FSIM, ATALANTA, and COMPACTEST and by considering parity tree as space compactor.

5.6.3 Graphical Representation of All Simulated Circuits Assuming Stochastic Independence of Line Errors

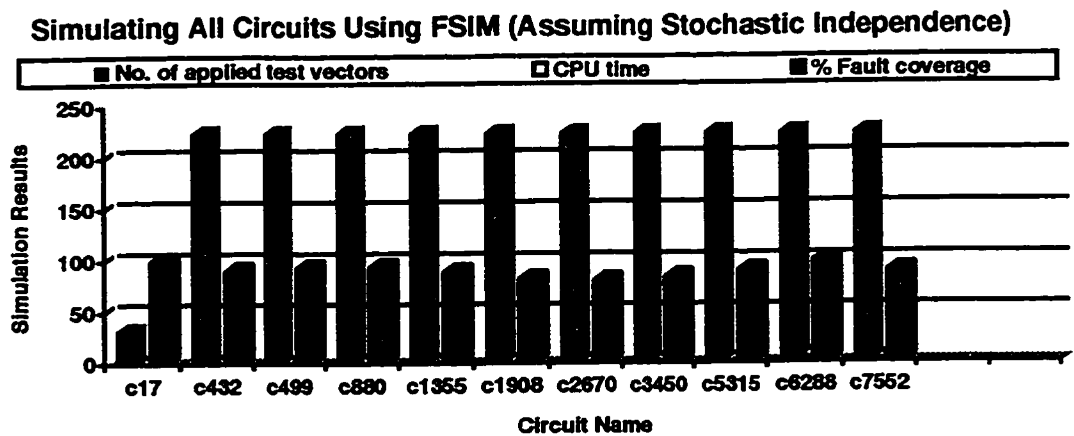


Figure 5.6 Simulation results for all benchmark circuits using FSIM and by assuming stochastic independence of line errors.

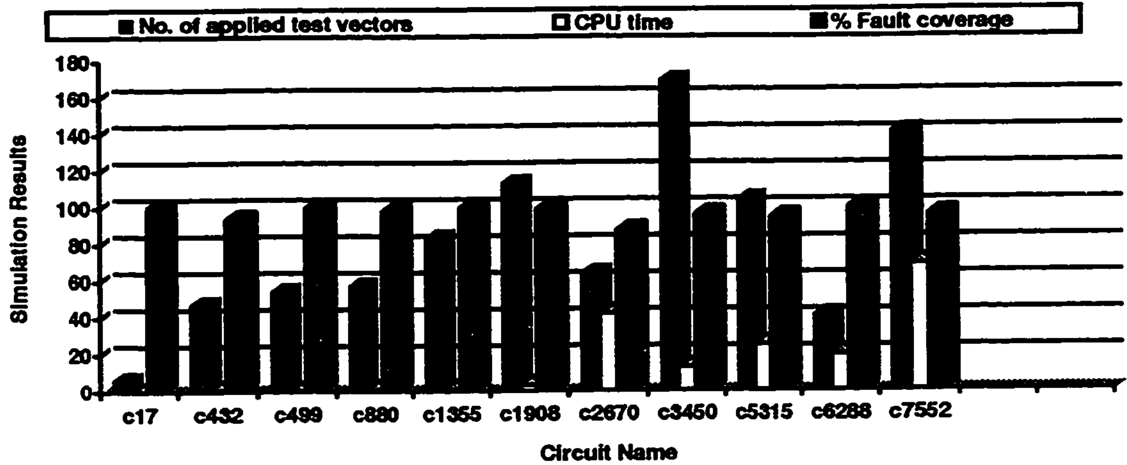


Figure 5.7 Simulation results for all benchmark circuits using ATALANTA and by assuming stochastic independence of line errors.

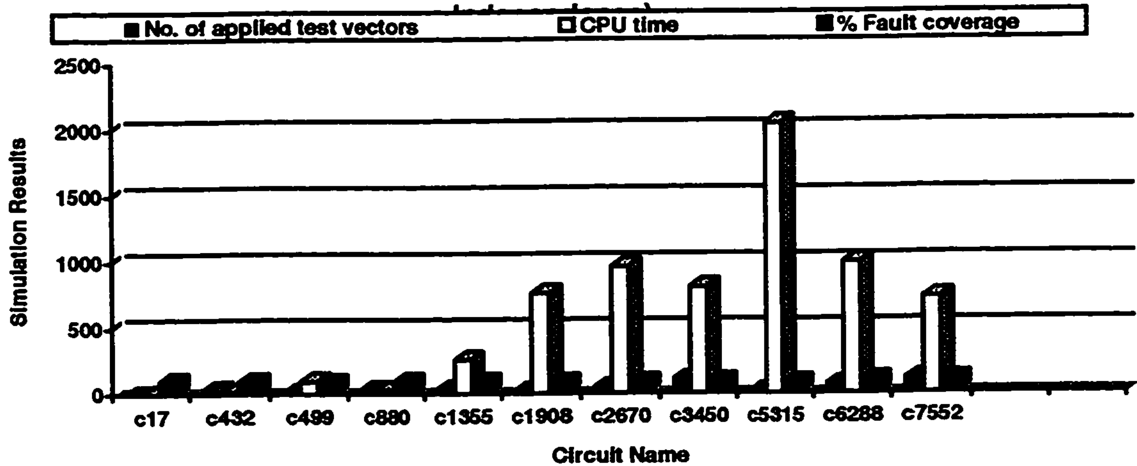


Figure 5.8 Simulation results for all benchmark circuits using COMPACTEST and by assuming stochastic independence of line errors.

5.6.3 Graphical Representation of All Simulated Circuits Assuming Stochastic Dependence of Line Errors

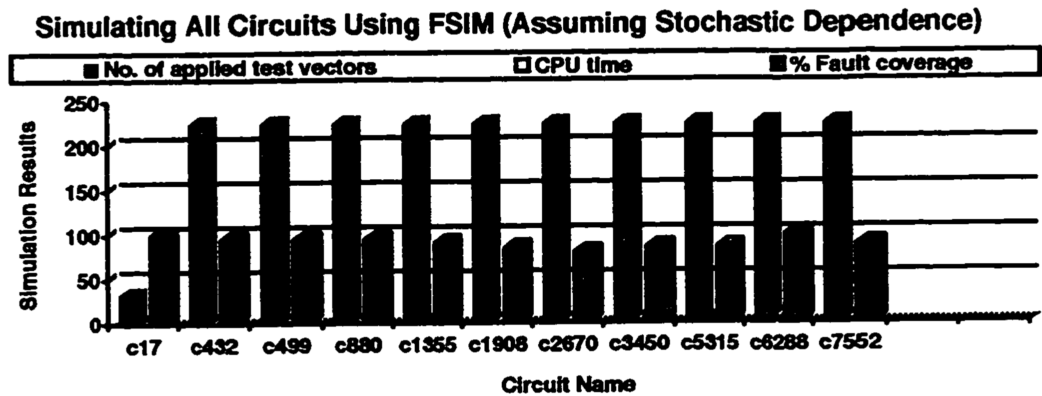


Figure 5.9 Simulation results for all benchmark circuits using FSIM and by assuming stochastic dependence of line errors.

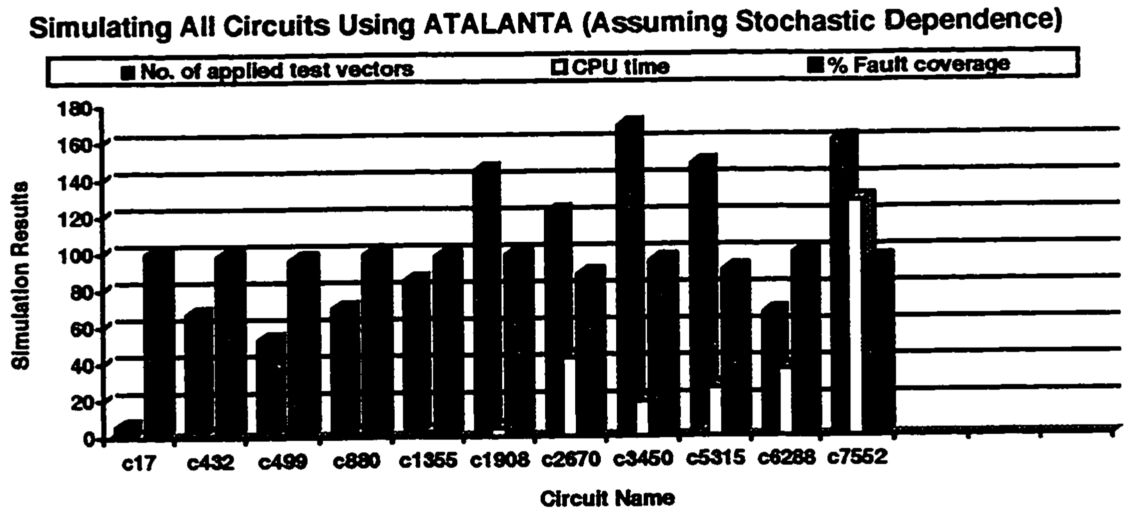


Figure 5.10 Simulation results for all benchmark circuits using ATALANTA and by assuming stochastic dependence of line errors.

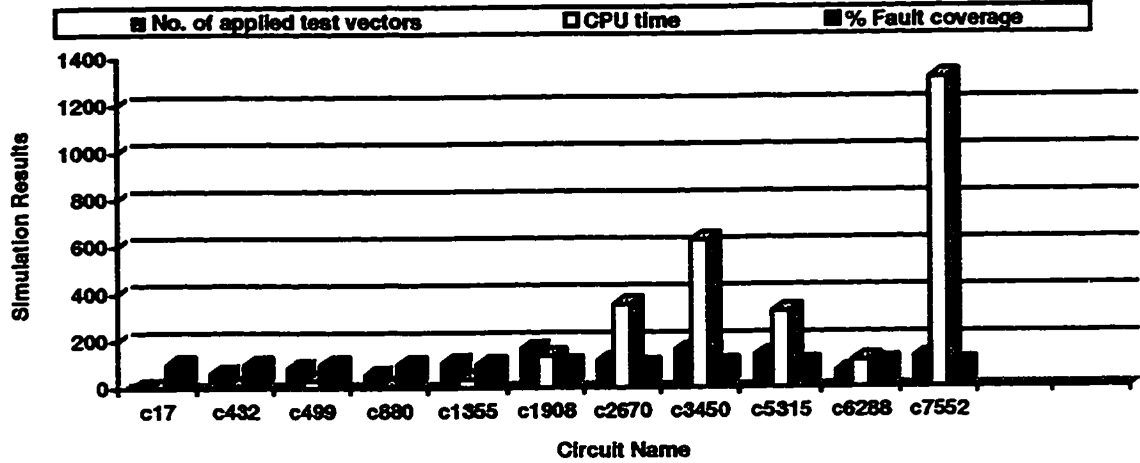


Figure 5.11 Simulation results for all benchmark circuits using COMPACTEST and by assuming stochastic dependence of line errors.

Chapter 6

Concluding Remarks

This thesis discusses several space compaction techniques of test responses of VLSI circuits for built-in self-testing (BIST). The compaction tree is comprised of gates: AND (NAND), OR (NOR), and XOR (XNOR) and is constructed using concepts of Hamming distance, generalized sequence weights to select an appropriate gate for merger of a number of output streams from CUT under conditions of both stochastic independence and stochastic dependence of multiple line errors, based on a generalized mergeability criteria developed in the thesis.

In the present thesis, no endeavor has been made to design aliasing-free space compactors. Rather, many of the concepts used in the design of space compactors through merger of only two output sequences of CUT as commonly used and discussed in the literature are extended in order to include the optimal mergeability conditions of any number of output streams of the circuit under generalized error occurrence conditions, considering both stochastic independence and dependence of output errors from the circuit. Loss of information in space as well as in time compaction is unavoidable, in general, though this could be minimized in many designs. In the thesis, reduced test sets provided by FSIM, ATALANTA and COMPACTEST to simulate all ISCAS 85 benchmark circuits are used, though in many instances, these reduced test inputs do not constitute minimal test sets to ensure 100% fault coverage for single stuck-line faults. However, experimental results do demonstrate that the designed space compactors with these reduced test sets perform very well compared to the parity tree space compactor which only propagates errors on an odd number of inputs and considered suitable for ad hoc design.

The techniques proposed in the thesis are simple as design tools and their resulting low hardware overhead make them suitable for built-in self-testing in VLSI design, even though they do not guarantee 100% fault coverage for single stuck-line errors.

Publications

Papers Published by the Author of this Thesis

1) Generalized mergeability in space compressor design in built-in self-test of VLSI circuits. Presented at the *IEEE Instrumentation and Measurement Technology Conference*, Ottawa, Ontario, Canada, May 19-21, 1997 (Conference Proceedings, Vol. 2, pp. 1448-1453).
(Jointly with S. R. Das, A. R. Nayak and M. H. Assaf).

2) Generalibility detectability error probability estimate and output data compression in integrated circuit design under different multiplicities of error. Presented at the *IASTED International Conference on Modeling, Simulation and Optimization*, Singapore, August 11-13, 1997 (Conference Proceedings, pp. 195-198).
(Jointly with S. R. Das, E. M. Petriu, M. H. Assaf and A. R. Nayak).

3) Generalized error probability estimate in output data compression under multiplicities of error - mathematical analysis - *Third World Conference on Integrated Design and Process Technology, Berlin, Germany*, July 1998 (accepted for presentation).
(Jointly with S. R. Das, E. M. Petriu, M. H. Assaf and A. R. Nayak).

4) Space compaction under generalized mergeability. Presented at the *IEEE Instrumentation and Measurement Technology Conference*, St. Paul, MN, U.S.A., May 18-21, 1998 (Conference Proceedings, Vol. 1, pp. 519-524).
(Jointly with S. R. Das, E. M. Petriu, M. H. Assaf and A. R. Nayak).

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

3. V. D. Agrawal and S. C. Seth, *Test Generation for VLSI Chips - Tutorial*, IEEE Computer Society Press, Washington D.C., 1988.

4. S. B. Akers, "A parity bit signature for exhaustive testing," *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 333-338, March 1988.

5. S. B. Akers, "In the use of linear sums in exhaustive testing," *Procedure IEEE Design Automation Conf.*, pp. 148-153, 1985.

6. 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.

7. P. H. Bardell, W. H. McAnney and J. Savir, *Built-in Test for VLSI: Pseudorandom Techniques*, John Wiley & Sons, Inc., New York, 1987.

8. B. B. Bhattacharya and S. C. Seth, "Design of parity testable combinational circuits," *IEEE Transactions on Computers*, Vol. C-38, pp. 1580-1584, November 1989.

9. K. Chakrabarty, "Balanced Boolean Functions: Theory and Applications," M.S. Thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, September 1992.

10. K. Chakrabarty and J. P. Hayes, "Aliasing-free error detection," *Digest of Papers 1993 IEEE VLSI Test Symp.*, pp. 260-266, 1993.

11. K. Chakrabarty and J. P. Hayes, "Balance testing of logic circuits," *Proc. 1993 Int. Symp. on Fault-Tolerant Computing*, pp. 350-359, 1993.

12. K. Chakrabarty and J. P. Hayes, "DFBT: A design for testability method based on balance testing," *Proc. 1994 Design Automation Conference*, pp. 351-357, 1994.

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

14. K. Chakrabarty and J. P. Hayes, "Cumulative balance testing of logic circuits," *IEEE Transactions on VLSI Systems*, Vol. 3, pp. 72-83, March 1995.

15. K. Chakrabarty B. T. Murray, and J. P. Hayes, "Optimal space compaction of test responses," *Proc. 1995 Int. Test Conference*, pp. 615-622, 1995.

16. K. Chakrabarty, "Test response compaction for built-in self-testing," Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, U.S.A., 1995.

17. C.-I. H. Chen and J. T. Yuen, "Automated synthesis of pseudo-exhaustive test generator in VLSI BIST design," *IEEE Transactions on VLSI Systems*, Vol. 3, pp. 273-291, September 1994.

18. K.-T. Cheng, V. D. Agrawal, *Unified Methods for VLSI Simulation and Test Generation*, Kluwer Academic Publishers, Massachusetts, 1989.

19. S. R. Das, *Boolean Difference Methods*, Notes for graduate course ELG 5195, University of Ottawa, 1995.

20. 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.

21. S. R. Das, T. Barakat, A. Nayak, M. H. Assaf "Generalized mergeability in space compressor design in BIST of VLSI circuits," *Proc. IEEE Instrumentation and Measurement Technology Conference*, Vol. 2, pp. 1448-1453, May 1997.

22. R. A. Frohwerk, "Signature analysis: A new digital field service method," *Hewlett-Packard Journal*, Vol. 28, pp. 2-8, September 1977.

23. H. Fujiwara and K. Kimoshita, "Testing logic circuits with compressed data," *Proc. IEEE 8th Int. Symp. on Fault Tolerant Computing*, pp. 108-113, 1978.

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

25. 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.

26. S. K. Gupta, D. K. Pradhan, and S. M. Reddy, "Zero aliasing compression" *Proc. 1990 Int. Symp. on Fault Tolerant Computing*, pp. 254-263, 1990.

27. 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.

28. J. P. Hayes, "Generation of optimal transition count tests," *IEEE Transactions on Computers*, Vol. C-27, pp. 36-41, January 1978.

29. J. P. Hayes, "Transition count testing of combinational logic circuits," *IEEE Transactions on Computers*, Vol. C-25, pp. 613-620, June 1976.

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

31. S. L. Hurst, *Custom VLSI Microelectronics*, Prentice-Hall International, UK, 1992.

32. 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.

33. 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.

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

35. 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.

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

37. W. H. McAnney and J. Savir, "Built-in checking of the correct self-test signature," *IEEE Trans. on Computers*, Vol. 37, pp. 1142-1145, September 1988.

38. E. J. McCluskey, "Verification testing - A pseudo-exhaustive test technique," *IEEE Trans. on Computers*, Vol. C-33, pp. 541-546, June 1984.

39. E. J. McCluskey, *Logic Design Principles, with Emphasis on Testable Semicostum Circuits*, Prentice-Hall, New Jersey, 1986.

40. E. J. McCluskey, "Built-in self-test structures," *IEEE Design and Test of Computers*, Vol. 2, pp. 29-36, January 1985.

41. D. M. Miller, *Developments Integrated Circuit Testing*, Academic Press, London, 1987.

42. 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.

43. I. Pomeranz, S. M. Reddy and R. Tangirala, "On achieving zero aliasing for modeled faults," *Proc. 1992 European Design Automation Conference*, pp. 291-299, March 1992.

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

45. S. M. Reddy, "A note on logic circuit testing by transition counting," *Trans. IEEE Computers*, Vol. C-26, pp. 313-314, 1977.

46. S. M. Reddy, K. K. Saluja and M. G. Karpovsky, "A data compression technique for built-in self-test," *IEEE Trans. on Computers*, Vol. C-37, pp. 1151-1156, September 1988.

47. J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop.*, pp. 278-291, 1966.

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

49. J. Savir, "Syndrome-testable design of combinational circuits," *IEEE Trans. on Computers*, Vol. C-29, pp. 442-451, June 1980.

50. 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.

51. 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.

52. 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.

53. F. F. Sellers, M. Y. Hsiao and L. W. Beamson, "Analyzing errors with Boolean Difference," *IEEE Trans. on Computers*, Vol. C-17, pp. 676-683, July 1968.

54. A. K. Susskind, "Testing by verifying Walsh coefficients," *Proc. 1981 Int. Symp. on Fault-Tolerant Computing*, pp. 206-208, 1981.

55. D. T. Tang and C. L. Cheng, "Logic test patterns using linear codes," *Proc. IEEE Design Automation Conference*, Vol. C-33, pp. 845-850, 1984.

56. C. C. Timoc, "Selected reprints on logic design for testability," *IEEE Computer Society*, 1988.

57. C. Vivier, "Trends in test generation and fault simulation," *Proceedings VLSI and Computers*, pp. 367-372, 1987.

58. T. W. Williams, W. Deahn, 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.

59. T. W. Williams and K. P. Parker, "Design for testability - A survey," *Proc. of the IEEE*, Vol. 71, pp. 98-112, January 1983.

60. V. N. Yarmolik, *Fault Diagnosis of Digital Circuits*, John Wiley & Sons, Inc., New York, 1990.

61. Y. Zorian and V. K. Agrawal, "A general scheme to optimize error masking in built-in self-testing," *Proc. Int. Symp. on Fault-Tolerant Computing*, pp. 410-415, 1986.

Appendix A

C Program to Construct the Space Compaction Trees for Combinational Logic Circuits Assuming Stochastic Independence of Multiple Line Errors

```
/*********************************************************************
*                                                                    *
*                                                                    *
*     Copyright (C) 1997,                                            *
*     University of Ottawa                                           *
*                                                                    *
*     Module name: Algorithm 1                                       *
*                                                                    *
*     This C program is written by Toni (Tony) F. Barakat under      *
*     the supervision of Dr. Sunil Das, Professor at the University  *
*     of Ottawa, Department of Electrical and Computer Engineering.  *
*                                                                    *
*     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 permi-   *
*     ssion of the author and the supervisor.                        *
*                                                                    *
*     For detailed information, please contact:                      *
*                                                                    *
*     Dr. Sunil R. Das                                               *
*     Department of Electrical and Computer Engineering              *
*     University of Ottawa                                           *
*     Ottawa, Ontario                                                *
*     Canada K1N 6N5                                                 *
*     Telephone:    (613) 562-5800 ext. 6216                         *
*     Fax:          (613) 562-5175                                   *
*     E-mail:       s.r.das@ieee.org                                 *
*                                                                    *
*     or                                                             *
*                                                                    *
*     Mr. Toni (Tony) F. Barakat                                     *
*     Lucent Technologies                                            *
*     (Bell Labs Innovations)                                        *
*     1200 E. Warrenville Rd.                                        *
*     Room 1F-416                                                    *
*     P.O. Box 3045                                                  *
*     Naperville, IL 60566-7045                                      *
*     Telephone:    (630) 979-1671                                   *
*     Fax:          (613) 979-7506                                   *
*     E-mail:       tonib@lucent.com                                 *
*                                                                    *
*                                                                    *
*********************************************************************/
```

```
/***************************************************************
*                    alg1_all.c                                *
*        This program executes First Algorithm                 *
*             and prints the detailed results                  *
*                                                              *
***************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/times.h>
#include <time.h>
#include <malloc.h>
#include <memory.h>


/***************************************************************
*                                                              *
*       FUNCTION: gettime                                      *
*       Description: it gets the CPU time used                 *
*       by the program                                         *
*                                                              *
***************************************************************/

void gettime(usertime,systemtime,total)
float *usertime;
float *systemtime;
float *total;
{
  struct tms timesbuffer;
  time_t totaltime;
  time_t utime;
  time_t stime;

  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;
}


/*********************************************************************
*                                                                    *
*       Definitions of all variable types                            *
*       used in this program                                         *
*                                                                    *
*********************************************************************/


#define MAXLINES 140
#define MAXBITS 217
```

```
#define LEFT_THR 0.5
#define RIGHT_THR 0.6

#define MAXWORDS MAXBITS/sizeof(long) + 1 /*sizeof(long) is 32 bit of length*/

#define getSeqId(x) ((x)<0)?maxNumber - x - (seq_number - 1):SeqId[x]


typedef struct {
 unsigned long seq[MAXWORDS];
 int cnt;
} SEQ;
SEQ seq[MAXLINES];


typedef struct {
 SEQ seq[MAXLINES];
 int OrigInd[MAXLINES]; /*original index*/
 int seq_number; /*Actual number of sequences (usually smaller than MAXLINES*/
} CANDIDATE;
typedef struct {
 CANDIDATE* pOr;  /* pOr is a pointer to CANDIDATE structure */
 CANDIDATE* pAnd; /* pAnd is a pointer to CANDIDATE structure */
 CANDIDATE* pXor; /* pXor is a pointer to CANDIDATE structure */
 int OrFlag;
 int AndFlag;
 int XorFlag;
} CANDIDATE_SET;

CANDIDATE_SET set;

int count;
float leftThr = LEFT_THR;
float rightThr = RIGHT_THR;

int max_cnt = 0; /*Maximum number of matching 1's in the selected group*/
int SeqList[MAXLINES];
int SeqListL = 0; /* Length of sequence list*/
int Level = 0; /*global variable: it is the level of grouping = N*/

int input_seq[MAXLINES][MAXBITS];

int count_1bits( unsigned long* );
int bits_cnt;                        /*length of a sequence = Ls*/
int seq_number;                      /*number of primary inputs*/

int SeqId[MAXLINES];
int maxNumber;      /*Maximum number of the line numbering file */
int count_Added_Seq; /* It counts the added sequence after AND, OR and XOR*/
int counterTree = 0; /* It counts how many trees exists */

char Tree[2000];

void Delete(CANDIDATE* cand, int* list, int list_size);
void Add(CANDIDATE* cand, SEQ* new);
void SeqPrint(unsigned long *tmp);
```

```
void Copy(unsigned long *source, unsigned long *target);
int HasItOne(unsigned long *oper1);
void SetItUp(unsigned long *oper1);
void LeftShift(unsigned long *oper1, unsigned long* res);
void RightShift(unsigned long *oper1, unsigned long* res);
void Invert(unsigned long *oper1, unsigned long* res);
void SeqXor(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void SeqOr(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void SeqAnd(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void lOverTwo(SEQ* seq, CANDIDATE_SET* newSet, int i);
void Selection(SEQ* new_seq, CANDIDATE_SET* newSet);
int AllocateSet(CANDIDATE_SET* newSet);
void FreeSet(CANDIDATE_SET* newSet);
int Explore(CANDIDATE_SET* s, char* tree);

FILE *in, *out; /*pointers to input and output files */

float minutes,seconds,inittime,runtime,simtime;


/*****************************************************************
 *                                                               *
 *              Main program                                     *
 *                                                               *
 *****************************************************************/

main(int argc, char **argv)
{

 int i, j;
 int k;
 unsigned long tmp[MAXWORDS], mask[MAXWORDS],res[MAXWORDS];
 SEQ new_seq;
 CANDIDATE_SET s;
 if(argc > 3)
  leftThr = atof(argv[3]);
 if(argc > 4)
  rightThr = atof(argv[4]);
 printf("leftThr = %f rightThr = %f\n",leftThr,rightThr);
 in = fopen(argv[1],"r");
 if(!in) /*if input file cannot be opened do the following*/
  {
       printf("ERROR OPENING %s FILE\n",argv[1]);
       exit(1); /*in UNIX exit(0) is a success, others are failures*/
  }

 fscanf(in,"%d%d",&seq_number,&bits_cnt);
 if (seq_number < 2)
  {
       printf("Number of bit streams should be positive\n");
       printf("Check your data file\n");
       exit(1); /*in UNIX exit(0) is a success, others are failures*/
  }

 if (bits_cnt < 2)
  {
```

```
        printf("Sequence length should be positive\n");
        printf("Check your data file\n");
        exit(1); /*in UNIX exit(0) is a success, others are failures*/
  }

 while(!feof(in))  /*as long as it is not end of file do the following*/
  {
        for(j=0; j<bits_cnt; j++)
        {
                for(i=0; i<seq_number; i++)
                {
                        fscanf(in,"%d",&input_seq[i][j]);
                        if (input_seq[i][j] != 0 && input_seq[i][j] != 1)
                        {
                                printf("Bit stream values has to be binary either 0 or 1\n");
                                printf("Check your data file\n");
                                exit(1);
                        }       /*end of if test*/
                } /*end of inner for loop*/
        } /*end of outer for loop*/

  }     /*end of while loop*/
 fclose(in);


in = fopen(argv[2],"r");
 if(!in)  /*if input file cannot be opened do the following*/
  {
        printf("ERROR OPENING %s FILE\n",argv[2]);
        exit(1); /*in UNIX exit(0) is a success, others are failures*/
  }
for (i=0; i<seq_number; i++)
{
  if(fscanf(in, "%d", &SeqId[i]) == EOF)
        {
         printf("errors were found in the line numbering file");
         exit(1);
        }
}
fclose(in);


count= seq_number;

 printf(" seq_num = %d  bits_cnt = %d\n",seq_number,bits_cnt);

/****************************************************************
 *                                                              *
 *      Allocate memory for AND, OR, and  XOR Candidates        *
 *                                                              *
 ****************************************************************/
 if(!AllocateSet(&s)) {
  printf("can't allocate memory\n");
  exit(1);
 }
```

```
/****************************************************************
 *                                                              *
 *      Conversion of a byte sequence into a bit sequence       *
 *                                                              *
 ****************************************************************/

 for(i = 0; i < seq_number; i++) {
    SetItUp(seq[i].seq); /*storage of a sequence*/
    seq[i].cnt = 0; /* count of number of 1's in a sequence*/
    for( j = 0; j < bits_cnt; j++ ) {
          seq[i].cnt += input_seq[i][j];
          LeftShift(seq[i].seq,seq[i].seq);
          *(seq[i].seq) |= input_seq[i][j]; /*by now each sequence is transfered to a long*/

    }
/****************************************************************
 *                                                              *
 *                                                              *
 *      Seperation of AND and OR candidates:                    *
 *      All input sequences that have number of 1's > Ls/2 are  *
 *      sent to the AND list. All input sequences that have     *
 *      number of 1's < Ls/2 are sent to OR list                *
 *                                                              *
 *                                                              *
 ****************************************************************/

    if( seq[i].cnt > ((float)bits_cnt)/2.0 ) {
          s.pAnd->OrigInd[s.pAnd->seq_number] = i;
          s.pAnd->seq[s.pAnd->seq_number++] = seq[i];

    } else     if( seq[i].cnt < ((float)bits_cnt)/2.0 ) {
          s.pOr->OrigInd[s.pOr->seq_number] = i;
          s.pOr->seq[s.pOr->seq_number] = seq[i];
          Invert(s.pOr->seq[s.pOr->seq_number].seq, s.pOr->seq[s.pOr->seq_number].seq);
          s.pOr->seq_number++;
    }
    printf("%d\t%d\t%d\t",i,SeqId[i], seq[i].cnt );
    SeqPrint(seq[i].seq);

 maxNumber = SeqId[i];
 count_Added_Seq = (SeqId[i] + 1); /* to count the newly added sequence
 coming from a merging gate */

  }
 /****************************************************************
  *                                                              *
  *      Handling of sequences that have number of 1's=Ls/2 case *
  *                                                              *
  ****************************************************************/
  for(i = 0; i < seq_number; i++) {
     if( seq[i].cnt == ((float)bits_cnt)/2.0 )
          1OverTwo(seq + i,&s,i);
  }

 printf("Max Number is: %d", maxNumber);
```

```
 /*--------------------------------------------------------*/
 Tree[0] = 0;

/*****************************************************************
 *                                                               *
 *       Built all possible Trees recursively                    *
 *                                                               *
 *****************************************************************/
 Explore(&s, Tree);
 FreeSet(&s); /* Free candidates memory */

 /*--------------------------------------------------------*/

/*****************************************************************
 *                                                               *
 *       Printing: CPU and simulation time by using the          *
 *       gettime function                                        *
 *                                                               *
 *****************************************************************/
 gettime(&minutes,&seconds,&runtime);
 printf("  CPU time\n");
/* printf("  Initialization     :%.3f secs.\n",inittime);
 printf("  Simulation         :%.3f secs.\n",simtime);
*/ printf("  Total              :%.3f secs.\n\n",runtime);

} /************************end of main*************************/




/*****************************************************************
 *                                                               *
 *       Mergeability criteria for AND and OR candidates         *
 *               "Choice of Best Grouping "                      *
 *                                                               *
 *****************************************************************/


/*****************************************************************
 *                                                               *
 * FUNCTION: And.                                                *
 * Description: it generates indexces list of the "Best          *
 * Group" using recurssion                                       *
 *                                                               *
 *****************************************************************/
int And(CANDIDATE *candList,int i,unsigned long *Operand,int level)
{
 int j;
 unsigned long tmp[MAXWORDS];
 int bit1_cnt;
 int RecordYourself = 0;

 for( j = i; j < candList->seq_number; j++ ) {
```

```
   SeqAnd(candList->seq[j].seq , Operand, tmp); /* Binary ANDing the first two input sequences, then
put their result in tmp */
   bit1_cnt = count_1bits(tmp);  /*calculate number of 1's in the [result==tmp]*/

   if( bit1_cnt < max_cnt)
        continue; /* getting a Bad Group */


/***************************************************************
 *                                                             *
 *          Getting a Good Group                               *
 *                                                             *
 ***************************************************************/
   if( bit1_cnt > max_cnt ) {
        SeqListL = 0;  /* erase the previous list */
        max_cnt = bit1_cnt;
        Level = level;  /* pass the value of local variable(level) to the global one (Level) */
        SeqList[SeqListL++] = j;
        RecordYourself = 1;
   } /* End of getting a Good Group */

/***************************************************************
 *                                                             *
 *    When  "bit1_cnt = max_cnt" (Bk = Bi) then                *
 *       we check for N; if Nk >= Ni then group Gk             *
 *       is better than group Gi.                              *
 *                                                             *
 ***************************************************************/
   else
   {
/* obtaining a better group than previous one */
        if(level > Level) {
          SeqListL = 0;
          Level = level;
          RecordYourself = 1;
          SeqList[SeqListL++] = j;
        } /* End of obtaining a better group than previous one */
   }
  if((j + 1 < candList->seq_number) && And(candList,j+1, tmp, level + 1)) { /* A recursive call is issued
here */
          RecordYourself = 1;
          SeqList[SeqListL++] = j;
   }
  }
  return(RecordYourself);
 }

 /***************************************************************
  *                                                             *
  *  FUNCTION: count_1bits                                      *
  *  Description: it counts the number of 1's in a              *
  *  bit streamsequence                                         *
  *                                                             *
  ***************************************************************/

 int
```

```
count_1bits( unsigned long* Operand)
{
        int j, k=0;
        unsigned long tmp[MAXWORDS];
        Copy(Operand,tmp);
        for (j=0; j<bits_cnt; j++)
        {
                k += (*tmp&0x1);
                RightShift(tmp,tmp);
        }
        return (k);
}


/*****************************************************************
 *                                                               *
 * FUNCTION: Delete                                              *
 * Description: it deletes sequences which are merged together.  *
 * these sequences are removed from the memory by using the      *
 * command "memmove".                                            *
 *                                                               *
 *****************************************************************/

void Delete(CANDIDATE* cand, int* list, int list_size)
{
  int i,j;
  int *pStart, *pEnd; SEQ *pSeqStart, *pSeqEnd;

  for (i=0; i<list_size; i++)
    cand->OrigInd[list[i]] = -99999;

  for (i=cand->seq_number; i>0; i--){
    if (cand->OrigInd[i-1] == -99999) {
      pStart = &(cand->OrigInd[i-1]);
      pEnd = &(cand->OrigInd[i]);
      memmove(pStart, pEnd, (cand->seq_number -i)*sizeof(int));
      pSeqStart = &(cand->seq[i-1]);
      pSeqEnd = &(cand->seq[i]);
      memmove(pSeqStart, pSeqEnd, (cand->seq_number -i)*sizeof(SEQ));
      cand->seq_number--;
    }
  }
}


/*****************************************************************
 *                                                               *
 * FUNCTION: Add                                                 *
 * Description: It adds a new sequence resulting from the        *
 * merge of certain candidate list.                              *
 *                                                               *
 *****************************************************************/


void Add(CANDIDATE *cand, SEQ *new)
```

```
{
        cand->seq[cand->seq_number] = *new;
        cand->OrigInd[cand->seq_number] = -count;
/*      cand->OrigInd[cand->seq_number] = count_Added_Seq; */
        cand->seq_number ++;
        count ++;
        count_Added_Seq ++;
}

/****************************************************************
 *                                                              *
 * FUNCTION: SeqAnd (&)                                         *
 * Description: It executes the binary AND operation of         *
 * two or more bit sequences. No value is returned.             *
 * In other words, this is binary AND for arrays of longs.      *
 *                                                              *
 ****************************************************************/


void SeqAnd(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  res[word]=oper1[word]&oper2[word];
 }
}

/****************************************************************
 *                                                              *
 * FUNCTION: SeqOr (|)                                          *
 * Description: It executes the binary OR operation of          *
 * two or more bit sequences.                                   *
 * It is simply:  [oper1 | oper2 = res]                         *
 *                                                              *
 ****************************************************************/

void SeqOr(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  res[word]=oper1[word]|oper2[word];
 }
}

/****************************************************************
 *                                                              *
 * FUNCTION: SeqXor (^)                                         *
 * Description: It executes the binary XOR operation of         *
 * two or more bit stream sequences                             *
 * It is simply:  [oper1 ^ oper2 = res]                         *
 *                                                              *
 ****************************************************************/
void SeqXor(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
 int word;
```

```
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  res[word]=oper1[word]^oper2[word];
 }
}

/****************************************************************
 *                                                              *
 * FUNCTION: Invert                                             *
 * Description: It executes the inversion or one's complement   *
 * operation on a bit stream sequence.                          *
 *                                                              *
 * Def. of a bit stream sequence: is a sequence which is        *
 * stored in an array of longs.                                 *
 *                                                              *
 ****************************************************************/
void Invert(unsigned long *oper1, unsigned long* res)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  res[word] = ~oper1[word];
 }
}
/****************************************************************
 *                                                              *
 * FUNCTION: RightShift ( >> )                                  *
 * Description: It executes the rigt shift                      *
 * operation on a bit stream sequence                           *
 *                                                              *
 *                                                              *
 ****************************************************************/
#define MASK 0x80000000L
void RightShift(unsigned long *oper1, unsigned long* res)
{
 unsigned long tmp;
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  tmp = oper1[word+1] & 1;
  res[word] = (oper1[word] >> 1);
  if(tmp)
   res[word] |= MASK;
 }
}

/****************************************************************
 *                                                              *
 * FUNCTION: LeftShift  ( << )                                  *
 * Description: It executes the left shift                      *
 * operation on a bit stream sequence                           *
 *                                                              *
 *                                                              *
 ****************************************************************/

void LeftShift(unsigned long *oper1, unsigned long* res)
{
 unsigned long tmp=0,new_tmp;
 int word;
```

```
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
   new_tmp = oper1[word] & MASK;
   res[word] = oper1[word] << 1;
   if(tmp)
     res[word] |= 1;
   tmp = new_tmp;
 }

}

/****************************************************************
 *                                                              *
 * FUNCTION: SetItUp                                            *
 * Description: it zeroes out a bit stream sequence.            *
 * In other words, it sets all bits of a bit stream             *
 * sequence to zeros.                                           *
 *                                                              *
 ****************************************************************/

void SetItUp(unsigned long *oper1)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
   oper1[word] = 0;
 }

}

/****************************************************************
 *                                                              *
 * FUNCTION: HasItOne                                           *
 * Description: It finds out whether a bit stream               *
 * sequence has at least a "1" in any of the bits.              *
 * in other words, finds out whether a bit stream               *
 * sequence is NOT a zero.                                      *
 *                                                              *
 ****************************************************************/

int HasItOne(unsigned long *oper1)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
   if(oper1[word])
     return(1);
 }
 return(0);
}

/****************************************************************
 *                                                              *
 * FUNCTION: Copy                                               *
 * Description: It executes the copy operation on               *
 * a bit stream sequence                                        *
 *                                                              *
 ****************************************************************/
```

```
void Copy(unsigned long *source, unsigned long *target)
{
 int word;
 for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
  target[word] = source[word];
 }
}
/*****************************************************************
 *                                                               *
 * FUNCTION: SeqPrint                                            *
 * Description: It prints a bit stream sequence on the screen    *
 *                                                               *
 *****************************************************************/
void SeqPrint(unsigned long *tmp)
{
 int j;
 unsigned long mask[MAXWORDS],res[MAXWORDS];
 SetItUp(mask);
 *mask = 1L;
 for (j=0; j<bits_cnt; j++){
  LeftShift(mask,mask);
 }
 for (j=0; j<bits_cnt; j++){
  RightShift(mask,mask);
  SeqAnd(tmp , mask, res);
  if (HasItOne(res))
   printf("1");
  else
   printf("0");

 }
 printf("\n");

}
/*****************************************************************
 *                                                               *
 * FUNCTION: lOverTwo                                            *
 * Description: Sequences that have number of ones = Ls/2        *
 * are sent either an AND or OR list of candidate, depending     *
 * where a better potential grouping exists                      *
 *                                                               *
 *                                                               *
 *                                                               *
 *****************************************************************/
void lOverTwo(SEQ* seq,CANDIDATE_SET* new_set, int j)
{
 int i,m,n;
 int andMatch = 0; /* Number of outputs resulting from executing
AND operation between the sequence which has number of 1's = Ls/2
with the candidates of the AND list. */
 int orMatch = 0; /* Number of outputs resulting from executing
OR operation between the sequence which has number of 1's = Ls/2
with the candidates of the OR list. */

 SEQ res; /* Resulting sequence */
 SEQ inverted; /* inverted sequence */
```

```
m = count_1bits(seq->seq);

for(i = 0; i < new_set->pAnd->seq_number; i++) {
  SeqAnd(seq->seq,new_set->pAnd->seq[i].seq,res.seq);
  if( m == count_1bits(res.seq))
    andMatch++;
}
Invert(seq->seq,inverted.seq); /* used to find out if there is a matching with OR list */
inverted.cnt = count_1bits(inverted.seq);
for(i = 0; i < new_set->pOr->seq_number; i++) {
  SeqAnd(inverted.seq,new_set->pOr->seq[i].seq,res.seq);
  if( m == count_1bits(res.seq))
    orMatch++;
}
if(!andMatch && !orMatch) {
  if(j<0)
    Add(new_set->pXor,seq);
  else {
        new_set->pXor->OrigInd[new_set->pXor->seq_number] = j;
        new_set->pXor->seq[new_set->pXor->seq_number++] = *seq;
  }
  return;
}
if(andMatch > orMatch) {  /* if andMatch > orMatch then send sequence "with Ls/2" to
OR list*/
  if(j<0)
    Add(new_set->pAnd,seq);
  else {
        new_set->pAnd->OrigInd[new_set->pAnd->seq_number] = j;
        new_set->pAnd->seq[new_set->pAnd->seq_number++] = *seq;
  }
  new_set->AndFlag = 1;
  return;
}
if(j<0)
  Add(new_set->pOr,&inverted);
else {
  new_set->pOr->OrigInd[new_set->pOr->seq_number] = j;
  new_set->pOr->seq[new_set->pOr->seq_number++] = inverted;
}

new_set->OrFlag = 1;
}

/*****************************************************************
 *              SELECTION OF CANDIDATES                          *
 *                                                               *
 * FUNCTION: Selection. (it has no return value).                *
 * Description: if in a bit stream sequence the number           *
 * of 1's > Ls/2 we send this sequence to the AND list,          *
 * if the number of 1's < Ls/2  we send this sequence            *
 * to the OR list.                                               *
 *                                                               *
 *****************************************************************/
```

```
void Selection(SEQ* new_seq, CANDIDATE_SET* new_set) {
    if (new_seq->cnt > ((float)bits_cnt)/2.0) {
        Add(new_set->pAnd, new_seq);
        new_set->AndFlag = 1;
        return;
    }
    if (new_seq->cnt < ((float)bits_cnt)/2.0) {
        Invert(new_seq->seq,new_seq->seq);
        Add(new_set->pOr, new_seq);
        new_set->OrFlag = 1;
        return;
    }
    lOverTwo(new_seq, new_set, -1);
}

/*****************************************************************
 *                                                               *
 * FUNCTION: SetCopy                                             *
 * Description: it copies candidates sets.                       *
 *                                                               *
 *                                                               *
 *****************************************************************/


void SetCopy(CANDIDATE_SET* source, CANDIDATE_SET* target)
{
 *(target->pOr) = *(source->pOr);
 *(target->pAnd) = *(source->pAnd);
 *(target->pXor) = *(source->pXor);
}

/*****************************************************************
 *                                                               *
 * FUNCTION: AllocateSet                                         *
 * Description: Allocates memory for candidates set.             *
 *                                                               *
 *****************************************************************/
int AllocateSet(CANDIDATE_SET* newSet)
{
 if(!(newSet->pOr = malloc(sizeof(CANDIDATE))))
   return(0);
 if(!(newSet->pAnd = malloc(sizeof(CANDIDATE))))
   return(0);
 if(!(newSet->pXor = malloc(sizeof(CANDIDATE))))
   return(0);
 newSet->pOr->seq_number = 0; /* number of sequences which are candidates for OR gate */
 newSet->pAnd->seq_number = 0;
 newSet->pXor->seq_number = 0;

 newSet->AndFlag = 1;
 newSet->OrFlag = 1;
 return(1);
}
```

```
/****************************************************************
 *                                                              *
 * FUNCTION: FreeSet                                            *
 * Description: Free up the candidates set memory               *
 * (it frees up the memory which was allocated)                 *
 *                                                              *
 ****************************************************************/


void FreeSet(CANDIDATE_SET* newSet)
{
 free(newSet->pOr);
 free(newSet->pAnd);
 free(newSet->pXor);
}

/****************************************************************
 *                                                              *
 * FUNCTION: Explore                                            *
 * Description: it builds all possible compaction               *
 * trees recursively                                            *
 *                                                              *
 *                                                              *
 ****************************************************************/


 unsigned long _tmp[MAXWORDS], _res[MAXWORDS];
int Explore(CANDIDATE_SET* s, char* tree)
{

 int i, j;
 int k;
 int oldCount_Added_Seq,oldCount;
 SEQ new_seq;
 CANDIDATE_SET new_set;
 char newTree[2000];
 int oldSeqList[MAXLINES];
 int oldSeqListL; /* Length of sequence list*/
 int oldLevel; /*global variable: it is the level of grouping = N*/

 while( s->OrFlag || s->AndFlag ) {

  /****************************************************************
   *                                                              *
   *          Selection of AND candidates                         *
   *                                                              *
   ****************************************************************/
  SeqListL = 0;
  max_cnt = 0;
  Level = 0;
  printf("\nand candidates\n");

  for(i = 0; i < s->pAnd->seq_number; i++) {
   printf("%d\t%d\t%d\t",s->pAnd->OrigInd[i],getSeqId(s->pAnd->OrigInd[i]), s->pAnd->seq[i].cnt );
   SeqPrint(s->pAnd->seq[i].seq);
  }
```

```
  for(i = 0; i < s->pAnd->seq_number - 1; i++) {
   if(And(s->pAnd,i + 1, s->pAnd->seq[i].seq, 1)){
        SeqList[SeqListL++] = i;
   }
  }
  if( (SeqListL > 1) && (max_cnt >= leftThr * ((float)bits_cnt))) {
   s->AndFlag = 1;
   if( max_cnt <= rightThr * ((float)bits_cnt)){
        for(i = 0; i < SeqListL; i++) {
         oldSeqList[i] = SeqList[i];
        }
        oldSeqListL = SeqListL;
        oldLevel = Level;
        printf("\nSequences (");
        for(i = 0; i < SeqListL; i++) {
         printf("(%d)/%d ",s->pAnd->OrigInd[SeqList[i]],getSeqId(s->pAnd->OrigInd[SeqList[i]]));
        }
        printf(") are merged together with XOR gate into ");

        printf("(%d) or ", count);
        printf("%d.\n", count_Added_Seq);

        printf("SEQ_LIST_L = %d\n",SeqListL);

        Copy(s->pAnd->seq[SeqList[0]].seq,_tmp);
        printf("Index\t%d\t%d\tAND -->",s->pAnd->OrigInd[SeqList[0]],getSeqId(s->pAnd-
>OrigInd[SeqList[0]]));
        SeqPrint(s->pAnd->seq[SeqList[0]].seq);
        for(i = 1; i < SeqListL; i++) {
         SeqXor(_tmp, s->pAnd->seq[SeqList[i]].seq, _tmp);
         printf("Index\t%d\t%d\tAND -->",s->pAnd->OrigInd[SeqList[i]],getSeqId(s->pAnd-
>OrigInd[SeqList[i]]));
         SeqPrint(s->pAnd->seq[SeqList[i]].seq);
        }
        SeqPrint(_tmp);

        Copy(_tmp,new_seq.seq);
        new_seq.cnt = count_1bits(_tmp);
        strcpy(newTree,tree);
        strcat(newTree," XOR");

/*
/*****************************************************************
 *                                                               *
 *  Printing FSIM, ATALANTA Format for XOR/AND gates             *
 *                                                               *
 *                                                               *
 *                                                               *
 *****************************************************************/


/*
   sprintf(newTree+strlen(newTree),"%d = XOR(",count_Added_Seq);
    for(i = 0; i < SeqListL;i++) {
     if(i != 0)
        sprintf(newTree+strlen(newTree),",");
```

```
    sprintf(newTree+strlen(newTree),"%d",getSeqId(s->pAnd->OrigInd[SeqList[i]]));
    }
  sprintf(newTree+strlen(newTree),")\n");
*/

        oldCount = count;
        oldCount_Added_Seq = count_Added_Seq;

        if(!AllocateSet(&new_set)) {
          printf("can't allocate memory\n");
          exit(1);
        }
        SetCopy(s,&new_set);
        Delete(new_set.pAnd, SeqList, SeqListL);
        Selection(&new_seq,&new_set);
        Explore(&new_set,newTree);
        count_Added_Seq = oldCount_Added_Seq;
        count = oldCount;
        FreeSet(&new_set);
        for(i = 0; i < oldSeqListL; i++) {
          SeqList[i] = oldSeqList[i];
        }
        SeqListL = oldSeqListL;
        Level = oldLevel;

    }

    printf("\nSequences (");
    for(i = 0; i < SeqListL; i++) {
         printf("(%d)/%d ",s->pAnd->OrigInd[SeqList[i]],getSeqId(s->pAnd->OrigInd[SeqList[i]]));
    }
    printf(") are merged together with AND gate into ");


    strcat(tree," AND");

/*
/****************************************************************
 *                                                              *
 *  Printing FSIM, ATALANTA Format for AND gate only            *
 *                                                              *
 *                                                              *
 *                                                              *
 ****************************************************************/

/*
   sprintf(tree+strlen(tree),"%d = AND(",count_Added_Seq);
    for(i = 0; i < SeqListL;i++) {
     if(i != 0)
          sprintf(tree+strlen(tree),",");
    sprintf(tree+strlen(tree),"%d",getSeqId(s->pAnd->OrigInd[SeqList[i]]));
    }
   sprintf(tree+strlen(tree),")\n");

 */
```

```
    printf("(%d) or ", count);
    printf("%d.\n", count_Added_Seq);
    printf("SEQ_LIST_L = %d\n",SeqListL);

    Copy(s->pAnd->seq[SeqList[0]].seq,_tmp);
    printf("Index\t%d\t%d\tAND -->",s->pAnd->OrigInd[SeqList[0]],getSeqId(s->pAnd-
>OrigInd[SeqList[0]]));
    SeqPrint(s->pAnd->seq[SeqList[0]].seq);
    for(i = 1; i < SeqListL; i++) {
        SeqAnd(_tmp, s->pAnd->seq[SeqList[i]].seq, _tmp);
        printf("Index\t%d\t%d\tAND -->",s->pAnd->OrigInd[SeqList[i]],getSeqId(s->pAnd-
>OrigInd[SeqList[i]]));
        SeqPrint(s->pAnd->seq[SeqList[i]].seq);
    }
    printf("\n_tmp = ");SeqPrint(_tmp);
    Delete(s->pAnd, SeqList, SeqListL);
    Copy(_tmp,new_seq.seq);
    new_seq.cnt = count_1bits(_tmp);
    Selection(&new_seq,s);
   } else
    s->AndFlag = 0;

  /***************************************************************
   *                                                             *
   *        Selection of OR candidates                           *
   *                                                             *
   ***************************************************************/

  SeqListL = 0;
  max_cnt = 0;
  Level = 0;
  printf("\nor candidates\n");

  for(i = 0; i < s->pOr->seq_number; i++) {
   Invert(s->pOr->seq[i].seq,_res);
   printf("%d\t%d\t%d\t",s->pOr->OrigInd[i], getSeqId(s->pOr->OrigInd[i]), s->pOr->seq[i].cnt
);SeqPrint(_res);
  }
  for(i = 0; i < s->pOr->seq_number - 1; i++) {
   if(And(s->pOr,i + 1, s->pOr->seq[i].seq, 1)){
        SeqList[SeqListL++] = i;
   }
  }
  if( (SeqListL > 1) && (max_cnt >= leftThr * ((float)bits_cnt))) {
   s->OrFlag = 1;
   if( max_cnt <= rightThr * ((float)bits_cnt)){
        for(i = 0; i < SeqListL; i++) {
         oldSeqList[i] = SeqList[i];
        }
        oldSeqListL = SeqListL;
        oldLevel = Level;
        printf("\nSequences (");
        for(i = 0; i < SeqListL; i++) {
         printf("(%d)/%d ",s->pOr->OrigInd[SeqList[i]],getSeqId(s->pOr->OrigInd[SeqList[i]]));
        }
        printf(") are merged together with XOR gate into ");
```

```
        printf("(%d) or ", count);
        printf("%d.\n", count_Added_Seq);

        printf("SEQ_LIST_L = %d\n",SeqListL);
        Invert(s->pOr->seq[SeqList[0]].seq,_tmp);
        printf("Index\t%d\t%d\tOR -->",s->pOr->OrigInd[SeqList[0]],getSeqId(s->pOr-
>OrigInd[SeqList[0]]));
        SeqPrint(_tmp);
        for(i = 1; i < SeqListL; i++) {
          Invert(s->pOr->seq[SeqList[i]].seq,_res);
          SeqXor(_tmp,_res,_tmp);
          printf("Index\t%d\t%d\tOR -->",s->pOr->OrigInd[SeqList[i]],getSeqId(s->pOr-
>OrigInd[SeqList[i]]));
          Invert(s->pOr->seq[SeqList[i]].seq,_res);
          SeqPrint(_res);
        }

        Copy(_tmp,new_seq.seq);
        new_seq.cnt = count_1bits(_tmp);
        strcpy(newTree,tree);


        strcat(newTree," XOR");


/*

/**************************************************************
 *                                                            *
 *  Printing FSIM, ATALANTA Format for OR/XOR gates           *
 *                                                            *
 *                                                            *
 *                                                            *
 **************************************************************/

/*
  sprintf(newTree+strlen(newTree),"%d = XOR(",count_Added_Seq);

    for(i = 0; i < SeqListL;i++) {
      if(i != 0)
          sprintf(newTree+strlen(newTree),",");
    sprintf(newTree+strlen(newTree),"%d",getSeqId(s->pOr->OrigInd[SeqList[i]]));
     }
   sprintf(newTree+strlen(newTree),")\n");

 */


          oldCount = count;
          oldCount_Added_Seq = count_Added_Seq;

          if(!AllocateSet(&new_set)) {
            printf("can't allocate memory\n");
            exit(1);
          }
```

```
        SetCopy(s,&new_set);
        Delete(new_set.pOr, SeqList, SeqListL);
        Selection(&new_seq,&new_set);
        Explore(&new_set,newTree);
        count_Added_Seq = oldCount_Added_Seq;
        count = oldCount;

        FreeSet(&new_set);
        for(i = 0; i < oldSeqListL; i++) {
         SeqList[i] = oldSeqList[i];
        }
        SeqListL = oldSeqListL;
        Level = oldLevel;

    }

    printf("\nSequences (");
    for(i = 0; i < SeqListL; i++) {
         printf("(%d)/%d ",s->pOr->OrigInd[SeqList[i]],getSeqId(s->pOr->OrigInd[SeqList[i]]));
    }
    printf(") are merged together with OR gate into ");



    strcat(tree," OR");

/*
/*****************************************************************
 *                                                               *
 *  Printing FSIM, ATALANTA Format for OR gate only              *
 *                                                               *
 *                                                               *
 *                                                               *
 *****************************************************************/
/*
 Printing FSIM, ATALANTA Format for OR gate only
    sprintf(tree+strlen(tree),"%d = OR(",count_Added_Seq);
    for(i = 0; i < SeqListL;i++) {
      if(i != 0)
         sprintf(tree+strlen(tree),",");
    sprintf(tree+strlen(tree),"%d",getSeqId(s->pOr->OrigInd[SeqList[i]]));
    }
   sprintf(tree+strlen(tree),")\n");

*/

    printf("(%d) or ", count);
    printf("%d\n", count_Added_Seq);
    printf("SEQ_LIST_L = %d\n",SeqListL);
    Invert(s->pOr->seq[SeqList[0]].seq,_tmp);
    printf("Index\t%d\t%d\tOR -->",s->pOr->OrigInd[SeqList[0]],getSeqId(s->pOr-
>OrigInd[SeqList[0]]));
    SeqPrint(_tmp);
    for(i = 1; i < SeqListL; i++) {
         Invert(s->pOr->seq[SeqList[i]].seq,_res);
         SeqOr(_tmp,_res,_tmp);
```

```
        printf("Index\t%d\t%d\tOR -->",s->pOr->OrigInd[SeqList[i]],getSeqId(s->pOr-
>OrigInd[SeqList[i]]));
        Invert(s->pOr->seq[SeqList[i]].seq,_res);
        SeqPrint(_res);
    }
    printf("\n_tmp = ");SeqPrint(_tmp);
    Delete(s->pOr, SeqList, SeqListL);
    Copy(_tmp,new_seq.seq);
    new_seq.cnt = count_1bits(_tmp);
    Selection(&new_seq,s);
  } else
    s->OrFlag = 0;
}


 /***************************************************************
  *                                                             *
  *         Selection of XOR candidates                         *
  *                                                             *
  ***************************************************************/

/*XOR proccesing*/
/*  s->pXor->seq_number = 0;*/
printf("\nxor candidates\n");
for(i = 0; i < s->pAnd->seq_number; i++) {
  s->pXor->OrigInd[s->pXor->seq_number] = s->pAnd->OrigInd[i];
  s->pXor->seq[s->pXor->seq_number++] = s->pAnd->seq[i];
}
s->pAnd->seq_number = 0;
for(i = 0; i < s->pOr->seq_number; i++) {
  s->pXor->OrigInd[s->pXor->seq_number] = s->pOr->OrigInd[i];
  Invert(s->pOr->seq[i].seq, s->pXor->seq[s->pXor->seq_number++].seq);
}
s->pOr->seq_number = 0;
if(s->pXor->seq_number > 0) {
  Copy(s->pXor->seq[0].seq,_tmp);
  printf("Index\t%d\t%d\tXOR -->",s->pXor->OrigInd[0],
          getSeqId(s->pXor->OrigInd[0]));
  SeqPrint( s->pXor->seq[0].seq);
  for(i = 1; i < s->pXor->seq_number; i++) {
    SeqXor(_tmp, s->pXor->seq[i].seq,_tmp);
    printf("Index\t%d\t%d\tXOR -->",s->pXor->OrigInd[i],
            getSeqId(s->pXor->OrigInd[i]));
    SeqPrint(s->pXor->seq[i].seq);
  }


  s->pOr->seq_number = 0;
  if(s->pXor->seq_number > 1) {
        printf("\nSequences (");
        printf("(%d)/%d ",s->pXor->OrigInd[0],getSeqId(s->pXor->OrigInd[0]));
        for(i = 1; i < s->pXor->seq_number; i++) {
            printf("(%d)/%d ",s->pXor->OrigInd[i],getSeqId(s->pXor->OrigInd[i]));
        }
        printf(") are merged together with XOR gate into ");
```

```
   strcat(tree," XOR");


/*
/******************************************************************
 *                                                                *
 *  Printing FSIM, ATALANTA Format for XOR gate only              *
 *                                                                *
 *                                                                *
 *                                                                *
 ******************************************************************/
/*
 Printing FSIM, ATALANTA Format for XOR gate only
        sprintf(tree+strlen(tree),"%d = XOR(",count_Added_Seq);
        sprintf(tree+strlen(tree),"%d",getSeqId(s->pXor->OrigInd[0]));
        for(i = 1; i < s->pXor->seq_number; i++) {
                if(i != 0)
                        sprintf(tree+strlen(tree),",");
                sprintf(tree+strlen(tree),"%d",getSeqId(s->pXor->OrigInd[i]));
        }
        sprintf(tree+strlen(tree),")\n");
*/

        printf("(%d) or ",count);
        printf("%d.\n",count_Added_Seq);
/*      count ++; */
/*      count_Added_Seq ++;  */
   }

  }
  printf("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
counterTree ++;
  printf("Tree: %d %s\t",counterTree, tree);
  SeqPrint(_tmp);
  printf("\n$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");

  printf("\n\n");

 }
```

Appendix B

C Program to Construct the Space Compaction Trees for Combinational Logic Circuits Assuming Stochastic Dependence of Multiple Line Errors

```
/*******************************************************************
*                                                                  *
*                                                                  *
*     Copyright (C) 1997,                                          *
*     University of Ottawa                                         *
*                                                                  *
*     Module name: Algorithm 2                                     *
*                                                                  *
*     This C program is written by Toni (Tony) F. Barakat under    *
*     the supervision of Dr. Sunil Das, Professor at the University *
*     of Ottawa, Department of Electrical and Computer Engineering. *
*                                                                  *
*     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 permi- *
*     ssion of the author and the supervisor.                      *
*                                                                  *
*     For detailed information, please contact:                    *
*                                                                  *
*     Dr. Sunil R. Das                                             *
*     Department of Electrical and Computer Engineering            *
*     University of Ottawa                                         *
*     Ottawa, Ontario                                              *
*     Canada K1N 6N5                                               *
*     Telephone:    (613) 562-5800 ext. 6216                       *
*     Fax:          (613) 562-5175                                 *
*     E-mail:       s.r.das@ieee.org                               *
*                                                                  *
*     or                                                           *
*                                                                  *
*     Mr. Toni (Tony) F. Barakat                                   *
*     Lucent Technologies                                          *
*     (Bell Labs Innovations)                                      *
*     1200 E. Warrenville Rd.                                      *
*     Room 1F-416                                                  *
*     P.O. Box 3045                                                *
*     Naperville, IL 60566-7045                                    *
*     Telephone:    (630) 979-1671                                 *
*     Fax:          (613) 979-7506                                 *
*     E-mail:       tonib@lucent.com                               *
*                                                                  *
*                                                                  *
*******************************************************************/
```

```
/*****************************************************************
 *                      alg2_all.c                               *
 *          This program execute Second ALG                      *
 *           and prints out all pertinent results                *
 *                                                               *
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/times.h>
#include <time.h>
#include <malloc.h>
#include <memory.h>

extern double Summation (int maxN, int bits_cnt, int *A, int code);
extern double rfact(int n);
extern double sifunc(int n, int m);
/*****************************************************************
 *                                                               *
 *      FUNCTION: gettime                                        *
 *      Description: it get the CPU                              *
 *      time used by the program                                 *
 *                                                               *
 *****************************************************************/

void gettime(usertime,systemtime,total)
float *usertime;
float *systemtime;
float *total;
{
        struct tms timesbuffer;
        time_t totaltime;
        time_t utime;
        time_t stime;

        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;
}

/*****************************************************************
 *                                                               *
 *      Definitions of all variable types used in the program    *
 *                                                               *
 *****************************************************************/


#define MAXLINES 150
```

```
#define MAXBITS 217

#define MAXWORDS MAXBITS/sizeof(long) + 1  /*sizeof(long) is 32 bit of length*/

#define getSeqId(x)  ((x)<0)?maxNumber - x - (seq_number - 1):SeqId[x]


typedef struct {
        unsigned long seq[MAXWORDS];
        int cnt;
} SEQ;
SEQ seq[MAXLINES];


typedef struct {
        SEQ seq[MAXLINES];
        int OrigInd[MAXLINES]; /*original index*/
        int seq_number; /*Actual number of sequences (usually smaller than MAXLINES*/
} CANDIDATE;
typedef struct {
        CANDIDATE* p;
        int Flag;
} CANDIDATE_SET;

CANDIDATE_SET set;

int count;


int max_cnt = 0;  /*Maximum number of matching 1's in the selected group*/
int SeqList[MAXLINES];
int SeqListL = 0;  /* Length of sequence list*/
int Level = 0;  /*global variable: it is the level of grouping = N*/


/********** Newly introduced variables ******************/
int maxW = 0;
int maxN = 0;
int residue;



int j, l, i, m, k, p, mask, group_seq_number;

double o,a;
double s;
double Ea, Eo, Ta, To, f1, Ba, Band, Bo, Bor;

int N;
int r;
static int A[100];

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

int  input_seq[MAXLINES][MAXBITS];
```

```
int count_1bits( unsigned long* );
int bits_cnt;                          /*length of a sequence*/
int seq_number;                        /*number of primary outputs*/

int SeqId[MAXLINES];
int maxNumber;      /*Maximum number of the line numbering file */
int count_Added_Seq;  /* It counts the added sequence after AND, OR and XOR*/

char Tree[2000];

void Delete(CANDIDATE* cand, int* list, int list_size);
void Add(CANDIDATE* cand, SEQ* new);
void SeqPrint(unsigned long *tmp);
void Copy(unsigned long *source, unsigned long *target);
int HasItOne(unsigned long *oper1);
void SetItUp(unsigned long *oper1);
void LeftShift(unsigned long *oper1, unsigned long* res);
void RightShift(unsigned long *oper1, unsigned long* res);
void Invert(unsigned long *oper1, unsigned long* res);
void SeqXor(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void SeqOr(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void SeqAnd(unsigned long *oper1, unsigned long *oper2, unsigned long* res);
void lOverTwo(SEQ* seq, CANDIDATE_SET* newSet, int i);
void Selection(SEQ* new_seq, CANDIDATE_SET* newSet);
int AllocateSet(CANDIDATE_SET* newSet);
void FreeSet(CANDIDATE_SET* newSet);
int Explore(CANDIDATE_SET* s, char* tree);

FILE *in, *out; /*pointers to input and output files */

float minutes,seconds,inittime,runtime,simtime;
/************************************************************
 *                                                          *
 *       FUNCTION: Si                                       *
 *                                                          *
 ************************************************************/

double sifunc(N,r)

int N;
int r;

  {

double x=3.0;
double y;
double z;
double s;
double pow(double x, double y);

  if (r>=1 && r<= (N-1))
        {
        y=(r-1);
/*
        printf("y=%f\n",y);
    printf("P=%f\n",1.0/pow(x,y));
```

```
    printf("(2/3)=%f\n",(2.0/3.0));
*/
        printf("Si= %f\n", ((2.0/3.0)*(1.0/(pow(x,y)))) );
        z=((2.0/3.0)*(1.0/(pow(x,y))));
        }
  if (r==N)
        {
        y=(r-2);
        printf("Si= %f\n", ((1.0/3.0)*(1.0/(pow(x,y)))) );
        z=((1.0/3.0)*(1.0/(pow(x,y))));
        }

return (z);

  }


/**********************************************************************
 *                                                                    *
 *      FUNCTION:     Recursive Factorial                             *
 *                                                                    *
 **********************************************************************/

double rfact(int n)     /* recursive function */
{

     double ans;

     if (n>0)
          ans=n * rfact(n-1);
     else
          ans = 1;
     return ans;
}


/**************************************************************
 *                                                            *
 *      Estimate and B functions                              *
 *                                                            *
 **************************************************************/

double Summation (int maxN, int bits_cnt, int *A, int code)
{

        double result = 0.0;
        double Ea, Eo, Ba, Bo, f1;
        int r;
        switch (code)
        {

        case 0:

                for(r=1; r<=maxN; r++)
                {
```

```
                f1 = rfact(maxN)/(rfact(maxN-r)*rfact(r));
                Eo=(sifunc(maxN,r)*(A[0] + (A[r]/f1))) / bits_cnt;

                printf("r=i= %d\n",r);
                printf("NC%d= %f\n", r, f1);
                printf("Ls= %d\n", bits_cnt);
                printf("A(%d) = %2d \n", r, A[r]);
                printf ("W[%d0] = %2d \n", maxN, A[0]);
                printf ("A/C= %f\n\n",  A[r] /f1);
                printf("E(OR)=  %f\n\n\n", Eo );

                result += Eo;
        }
        printf("Etotal(OR)=%f\n", result );
        return (result);



    case 1:

        for(r=1; r<=maxN; r++)
        {

                f1 = rfact(maxN)/(rfact(maxN-r)*rfact(r));
                Ea=(sifunc(maxN,r)*(A[maxN]+(A[maxN-r] /f1)))/bits_cnt;

                printf("r=i= %d\n",r);
                printf("NC%d= %f\n", r, f1);
                printf("Ls= %d\n", bits_cnt);
                printf("A(%d-%d) = %2d \n", maxN, r, A[maxN-r]);
                printf ("W[%d1] = %2d \n", maxN, A[maxN]);
                printf ("A/C= %f\n\n",  A[maxN-r] /f1);
                printf("E(AND)=  %f\n\n\n", Ea );

                result += Ea;
        }
        printf("Etotal(AND)=%f\n", result  );
        return (result);




    case 2:

      for(r=1; r<=maxN; r++)
        {
                f1 = rfact(maxN)/(rfact(maxN-r)*rfact(r));
        Ba= ( (sifunc(maxN,r) * A[maxN-r] /f1) )/bits_cnt;
        printf("Ba=  %f\n\n\n", Ba );
                result += Ba;
        }
/*
    printf("B(and)=%f\n", result );
*/
                return (result);
```

```
        case 3:

            for(r=1; r<=maxN; r++)
                {
                        f1 = rfact(maxN)/(rfact(maxN-r)*rfact(r));
                Bo= ( (sifunc(maxN,r) * A[r] /f1) )/bits_cnt;
                        printf("Bo=  %f\n\n\n", Bo );
                        result += Bo;
                }
/*
                printf("B(or)=%f\n", result );
*/
                return (result);

        }
/*
  return (result);
*/
}


/*********************************************************************
 *                                                                   *
 *      FUNCTION:     andGate                                        *
 *                                                                   *
 *********************************************************************/

void andGate (CANDIDATE_SET *s, int listL, int *list, unsigned long *tmp)
{

  int i;
        printf("(AND Gate)\n");
        printf("Index\t%d\t%d\tAND-->",s->p->OrigInd[SeqList[0]],
        getSeqId(s->p->OrigInd[list[0]]));
        SeqPrint(s->p->seq[list[0]].seq);
        for(i = 1; i < listL; i++)
                {
                        SeqAnd(tmp, s->p->seq[list[i]].seq, tmp);
                        printf("Index\t%d\t%d\tAND-->",s->p->OrigInd[list[i]],
                        getSeqId(s->p->OrigInd[list[i]]));
                        SeqPrint(s->p->seq[list[i]].seq);
                }
}

/*****************************************************************
 *                                                               *
 *      FUNCTION:     xorGate                                    *
 *                                                               *
 *****************************************************************/

void xorGate (CANDIDATE_SET *s, int listL, int *list, unsigned long *tmp)
{
```

```
 int i;
        printf("(XOR Gate)\n");
        printf("Index\t%d\t%d\tXOR-->",s->p->OrigInd[SeqList[0]],
        getSeqId(s->p->OrigInd[list[0]]));
        SeqPrint(s->p->seq[list[0]].seq);

        for(i = 1; i < listL; i++)
        {
                SeqXor(tmp, s->p->seq[list[i]].seq, tmp);
                printf("Index\t%d\t%d\tXOR-->", s->p->OrigInd[list[i]],
                getSeqId(s->p->OrigInd[list[i]]));
                SeqPrint(s->p->seq[list[i]].seq);
        }


}
/****************************************************************
 *                                                              *
 *      FUNCTION:    orGate                                     *
 *                                                              *
 ****************************************************************/

void orGate (CANDIDATE_SET *s, int listL, int *list, unsigned long *tmp)
{

  int i;
        printf("(OR Gate)\n");
        printf("Index\t%d\t%d\tOR-->",s->p->OrigInd[list[0]],
        getSeqId(s->p->OrigInd[list[0]]));
        SeqPrint(s->p->seq[list[0]].seq);

        Invert(tmp, tmp);
        for(i = 1; i < listL; i++)
        {
                Invert(s->p->seq[list[i]].seq,
                s->p->seq[list[i]].seq);
                SeqAnd(tmp, s->p->seq[SeqList[i]].seq, tmp);
                printf("Index\t%d\t%d\tOR-->", s->p->OrigInd[list[i]],
                getSeqId(s->p->OrigInd[list[i]]));
                Invert(s->p->seq[list[i]].seq,
                s->p->seq[list[i]].seq);
                SeqPrint(s->p->seq[list[i]].seq);
        }
        Invert(tmp, tmp);


}
/****************************************************************
 *                                                              *
 *      FUNCTION:    andXor                                     *
 *                                                              *
 ****************************************************************/


int andXor (int maxN, int bits_cnt, int *A)
```

```
{
        double Ba, f1;
        double totalS = 0.0;
        int i;

        for (i=1; i<=maxN; i=i+2)
                {
                        totalS=totalS + sifunc(maxN,i);
                        printf("A[maxN]/bits_cnt =%f\n", (float)A[maxN]/(float)bits_cnt);

                }
                printf(" Total sum of the S with odd indexces number = [totalS] = %f\n", totalS);
                printf("A[maxN]/bits_cnt - totalS  =%f\n", (((float)A[maxN]/(float)bits_cnt) -
(float)totalS) );
                if ( (((float)A[maxN]/(float)bits_cnt) >= (totalS)) )

                        {
                                return (1); /* And gate */
                        }
                        else
                        {
                                Summation(maxN, bits_cnt, &A[0], 2);
                                Ba = Summation(maxN, bits_cnt, &A[0], 2);
                                printf ("Band = %f\n\n", Ba);
                printf("(A+B)=C = %f\n", ((((float)A[maxN]/(float)bits_cnt) - (float)totalS))+Ba );

                                        if ( ((((float)A[maxN]/(float)bits_cnt)-(totalS))+Ba) > 0 )
                                        {
                                                return (1); /* And gate */

                                        }
                                        else
                                        {
                                                return (0); /* Xor gate */
                                        }

                        }

}


/*****************************************************************
 *                                                               *
 *      FUNCTION:       orXor                                    *
 *                                                               *
 *****************************************************************/

int orXor (int maxN, int bits_cnt, int *A)
{

        double Bo, f1;
        double totalS = 0.0;
        int i;

        for (i=1; i<=maxN; i=i+2)
                {
```

```
			totalS=totalS + sifunc(maxN,i);
			printf("A[0]/bits_cnt =%f\n", (float)A[0]/(float)bits_cnt);
			printf("maxN %d i %d totals %f \n",maxN,i, totalS);
		}
		printf(" Total sum of the S with odd indexces number = [totalS] = %f\n", totalS);
		printf("A[0]/bits_cnt - totalS  =%f\n", (((float)A[0]/(float)bits_cnt) - (totalS)) );

		if ( (((float)A[0]/(float)bits_cnt) >= (totalS))  )


			{
				return (1); /* OR gate */
			}
			else
			{
				Summation(maxN, bits_cnt, &A[0], 3);
				Bo = Summation(maxN, bits_cnt, &A[0], 3);
				printf ("Bor = %f\n\n", Bo);
				printf("(A+B)=C = %f\n", ((((float)A[0]/(float)bits_cnt) -
(float)totalS))+Bo );

					if ( ((((float)A[0]/(float)bits_cnt)-(totalS))+ Bo) > 0 )
					{
						return (1); /* OR gate */

					}
					else
					{
						return (0); /* XOR gate */
					}

			}

}




/****************************************************************
 *                                                              *
 *      Function that reads data from a file                    *
 *                                                              *
 ****************************************************************/

main(int argc, char **argv)
{

	int i, j;
	int k;
	unsigned long tmp[MAXWORDS], mask[MAXWORDS],res[MAXWORDS];
	SEQ new_seq;
	CANDIDATE_SET s;

	in = fopen(argv[1],"r");
```

```
if(!in) /*if input file cannot be opened do the following*/
        {
                printf("ERROR OPENING %s FILE\n",argv[1]);
                exit(1); /*in UNIX exit(0) is a success, others are failures*/
        }

fscanf(in,"%d%d",&seq_number,&bits_cnt);
if (seq_number < 2)
{
        printf("Number of bit streams should be positive\n");
        printf("Check your data file\n");
        exit(1); /*in UNIX exit(0) is a success, others are failures*/
}

if (bits_cnt < 2)
{
        printf("Sequence length should be positive\n");
        printf("Check your data file\n");
        exit(1); /*in UNIX exit(0) is a success, others are failures*/
}

while(!feof(in)) /*as long as it is not end of file do the following*/
{
        for(j=0; j<bits_cnt; j++)
        {
                for(i=0; i<seq_number; i++)
                {
                        fscanf(in,"%d",&input_seq[i][j]);
                        if (input_seq[i][j] != 0 && input_seq[i][j] != 1)
                        {
                                printf("Bit stream values has to be binary either 0 or 1\n");
                                printf("Check your data file\n");
                                exit(1);
                        }       /*end of if test*/
                } /*end of inner for loop*/
        } /*end of outer for loop*/

}       /*end of while loop*/
fclose(in);


in = fopen(argv[2],"r");
if(!in) /*if input file cannot be opened do the following*/
{
        printf("ERROR OPENING %s FILE\n",argv[2]);
        exit(1); /*in UNIX exit(0) is a success, others are failures*/
}
for (i=0; i<seq_number; i++)
{
        if(fscanf(in, "%d", &SeqId[i]) == EOF)
        {
                printf("errors were found in the line numbering file");
                exit(1);
        }
}
fclose(in);
```

```
count= seq_number;

printf(" seq_num = %d  bits_cnt = %d\n",seq_number,bits_cnt);

/******************************************************************
*                                                                 *
*       Print all input sequences on the screen                   *
*                                                                 *
******************************************************************/
if(!AllocateSet(&s)) {
        printf("can't allocate memory\n");
        exit(1);
}
for(i = 0; i < seq_number; i++) {
        SetItUp(seq[i].seq); /*storage of a sequence*/
        seq[i].cnt = 0; /* count of number of 1's in a sequence*/
        for( j = 0; j < bits_cnt; j++ ) {
                seq[i].cnt += input_seq[i][j];
                LeftShift(seq[i].seq,seq[i].seq);
                *(seq[i].seq) |= input_seq[i][j]; /*by now each sequence is transfered to a
long*/

        }
        /******************************************************************
         *                                                                *
         *          Seperation of AND and OR candidates                   *
         *                                                                *
         ******************************************************************/

        s.p->OrigInd[s.p->seq_number] = i;
        s.p->seq[s.p->seq_number++] = seq[i];

/*      printf("%d\t%d\t%d\t",i,SeqId[i], seq[i].cnt );
        SeqPrint(seq[i].seq);
*/
        maxNumber = SeqId[i];
        count_Added_Seq = (SeqId[i] + 1);

}
printf("Max Number is: %d\n", maxNumber);


/*----------------------------------------------*/
Tree[0] = 0;
Explore(&s, Tree);
FreeSet(&s);

/*------------------------------------------------------*/

/*CPU and simulation time*/

gettime(&minutes,&seconds,&runtime);
printf("   CPU time\n");
/* printf("   Initialization        :%.3f secs.\n",inittime);
```

```
                                                printf("  Simulation          :%.3f secs.\n",simtime);
                                               */ printf("  Total             :%.3f secs.\n\n",runtime);
} /*end of main*/


/*****************************************************************
 *                                                               *
 *    Mergeability criteria for AND and OR candidates            *
 *              "Choice of Best Grouping "                       *
 *                                                               *
 *****************************************************************/
int And(CANDIDATE *candList,int i,unsigned long *Operand,int level)
{
        int j;
        unsigned long tmp[MAXWORDS];
        int bit1_cnt;
        int RecordYourself = 0;

        for( j = i; j < candList->seq_number; j++ ) {
                SeqAnd(candList->seq[j].seq , Operand, tmp);
                bit1_cnt = count_1bits(tmp); /*calcuiate number of 1's in the result=tmp*/


                if( bit1_cnt < max_cnt)
                        continue;
                if(  bit1_cnt > max_cnt ) {
                        SeqListL = 0; /* erase the previous list */
                        max_cnt = bit1_cnt;
                        Level = level; /* pass the value of local variable(level) to the global one
(Level) */
                        SeqList[SeqListL++] = j;
                        RecordYourself = 1;
                }
                else
                {
                        if(level > Level) {
                                SeqListL = 0;
                                Level = level;
                                RecordYourself = 1;
                                SeqList[SeqListL++] = j;
                        }
                }
                if((j + 1 < candList->seq_number) && And(candList,j+1, tmp, level + 1)) {
                        RecordYourself = 1;
                        SeqList[SeqListL++] = j;
                }
        }
        return(RecordYourself);
}




/*****************************************************************
```

```
 *                                                                  *
 *      Function that counts the number of 1's in a sequence        *
 *                                                                  *
 ********************************************************************/

int count_1bits( unsigned long* Operand)
{
        int j, k=0;
        unsigned long tmp[MAXWORDS];
        Copy(Operand,tmp);
        for (j=0; j<bits_cnt; j++)
                {
                        k += (*tmp&0x1);
                        RightShift(tmp,tmp);
                }
        return (k);
}


/*******************************************************************
 *                                                                  *
 *      Function that deletes sequences which are merged together   *
 *                                                                  *
 ********************************************************************/

void Delete(CANDIDATE* cand, int* list, int list_size)
{
        int i,j;
        int *pStart, *pEnd;
        SEQ *pSeqStart, *pSeqEnd;

        for (i=0; i<list_size; i++)
                cand->OrigInd[list[i]] = -99999;

        for (i=cand->seq_number; i>0; i--){
                if (cand->OrigInd[i-1] == -99999) {
                        pStart = &(cand->OrigInd[i-1]);
                        pEnd = &(cand->OrigInd[i]);
                        memmove(pStart, pEnd, (cand->seq_number -i)*sizeof(int));
                        pSeqStart = &(cand->seq[i-1]);
                        pSeqEnd = &(cand->seq[i]);
                        memmove(pSeqStart, pSeqEnd, (cand->seq_number -i)*sizeof(SEQ));
                        cand->seq_number--;
                }
        }
}


/*******************************************************************
 *                                                                  *
 *      Function that adds a new sequence resulting from the merge  *
 *                                                                  *
 ********************************************************************/


void Add(CANDIDATE *cand, SEQ *new)
```

```
{
        cand->seq[cand->seq_number] = *new;
        cand->OrigInd[cand->seq_number] = -count;
        /*      cand->OrigInd[cand->seq_number] = count_Added_Seq; */
        cand->seq_number ++;
        count ++;
        count_Added_Seq ++;
}


void SeqAnd(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
        int word;
        for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
                res[word]=oper1[word]&oper2[word];
        }
}
void SeqOr(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
        int word;
        for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
                res[word]=oper1[word]|oper2[word];
        }
}
void SeqXor(unsigned long *oper1, unsigned long *oper2, unsigned long* res)
{
        int word;
        for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
                res[word]=oper1[word]^oper2[word];
        }
}
void Invert(unsigned long *oper1, unsigned long* res)
{
        int word;
        for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
                res[word] = ~oper1[word];
        }
}
#define MASK 0x80000000L
void RightShift(unsigned long *oper1, unsigned long* res)
{
        unsigned long tmp;
        int word;
        for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
                tmp = oper1[word+1] & 1;
                res[word] = (oper1[word] >> 1);
                if(tmp)
                        res[word] |= MASK;
        }
}
void LeftShift(unsigned long *oper1, unsigned long* res)
{
        unsigned long tmp=0,new_tmp;
        int word;
```

```
	for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
		new_tmp = oper1[word] & MASK;
		res[word] = oper1[word] << 1;
		if(tmp)
			res[word] |= 1;
		tmp = new_tmp;
	}

}
void SetItUp(unsigned long *oper1)
{
	int word;
	for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
		oper1[word] = 0;
	}

}
int HasItOne(unsigned long *oper1)
{
	int word;
	for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
		if(oper1[word])
			return(1);
	}
	return(0);
}
void Copy(unsigned long *source, unsigned long *target)
{
	int word;
	for(word = 0; word < bits_cnt/sizeof(long) + 1; word++ ) {
		target[word] = source[word];
	}
}

void SeqPrint(unsigned long *tmp)
{
	int j;
	unsigned long mask[MAXWORDS],res[MAXWORDS];
	SetItUp(mask);
	*mask = 1L;
	for (j=0; j<bits_cnt; j++){
		LeftShift(mask,mask);
	}
	for (j=0; j<bits_cnt; j++){
		RightShift(mask,mask);
		SeqAnd(tmp , mask, res);
		if (HasItOne(res))
			printf("1");
			else
			printf("0");

	}
	printf("\n");

}
```

```
void SetCopy(CANDIDATE_SET* source, CANDIDATE_SET* target)
{
	*(target->p) = *(source->p);
}

int AllocateSet(CANDIDATE_SET* newSet)
{
	if(!(newSet->p = malloc(sizeof(CANDIDATE))))
		return(0);
	newSet->p->seq_number = 0; /* number of sequences which are candidates for OR gate */
	newSet->Flag = 1;
	return(1);
}

void FreeSet(CANDIDATE_SET* newSet)
{
	free(newSet->p);
}


unsigned long _tmp[MAXWORDS], _res[MAXWORDS];
int Explore(CANDIDATE_SET* s, char* tree)
{

	int maxOnesCnt,onesSeqL;
	int maxNullsCnt,nullsSeqL;
	int onesSeqList[MAXLINES],nullsSeqList[MAXLINES];
	int i, j;
	int k;
	int oldCount_Added_Seq,oldCount;
	SEQ new_seq;
	CANDIDATE_SET new_set;
	char newTree[2000];
	int oldSeqList[MAXLINES];
	int oldSeqListL;  /* Length of sequence list*/
	int oldLevel; /*global variable: it is the level of grouping = N*/
	float S, totalS;
	unsigned long mask[MAXWORDS],res[MAXWORDS];
	int grouping;

	while( s->p->seq_number > 1 ) {
		/*****************************************************************
		 *                    Selection of AND candidates                  *
		 *****************************************************************/
		SeqListL = 0;
		max_cnt = 0;
		Level = 0;

		for(i = 0; i < s->p->seq_number; i++)
		{
			printf("%d\t%d\t%d\t",s->p->OrigInd[i],
			getSeqId(s->p->OrigInd[i]), s->p->seq[i].cnt );
			SeqPrint(s->p->seq[i].seq);
		}
		for(i = 0; i < s->p->seq_number - 1; i++)
```

```
{
        if(And(s->p , i + 1, s->p->seq[i].seq, 1))
        {
                SeqList[SeqListL++] = i;
        }
}
s->Flag = 1;
maxOnesCnt = max_cnt;
onesSeqL = SeqListL;
for ( i = 0; i < SeqListL; i++ )
        onesSeqList[i] = SeqList[i];
printf(" AND group Length %d seqNum %d \n", SeqListL, s->p->seq_number );
for ( i = 0; i < SeqListL; i++ )
        SeqPrint(s->p->seq[onesSeqList[i]].seq);

/*****************************************************************
 *                 Selection of OR candidates                     *
 *****************************************************************/

SeqListL = 0;
max_cnt = 0;
Level = 0;
for(i = 0; i < s->p->seq_number ; i++)
{
        Invert(s->p->seq[i].seq,s->p->seq[i].seq);
}

for(i = 0; i < s->p->seq_number - 1; i++)
{
        for(i = 0; i < s->p->seq_number - 1; i++)
        {
                if(And(s->p, i + 1, s->p->seq[i].seq, 1))
                {
                        SeqList[SeqListL++] = i;
                }
        }
}
for(i = 0; i < s->p->seq_number ; i++)
{
        Invert(s->p->seq[i].seq,s->p->seq[i].seq);
}

maxNullsCnt = max_cnt;
nullsSeqL = SeqListL;
for ( i = 0; i < SeqListL; i++ )
        nullsSeqList[i] = SeqList[i];
printf(" OR group Length %d\n", SeqListL );
for ( i = 0; i < SeqListL; i++ )
        SeqPrint(s->p->seq[nullsSeqList[i]].seq);
/*      for ( i = 0; i < onesSeqL; i++ )
        {
                SeqPrint(s->p->seq[onesSeqList[i]].seq);
        }
*/
        printf(" maxOnesCnt=%d maxNullsCnt=%d onesSeqL=%d nullsSeqL=%d\n",
        maxOnesCnt , maxNullsCnt, onesSeqL, nullsSeqL);
```

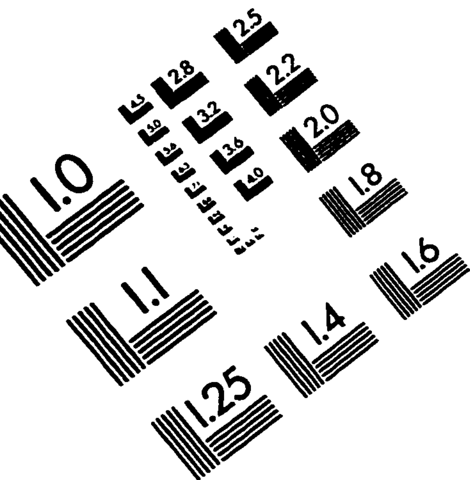
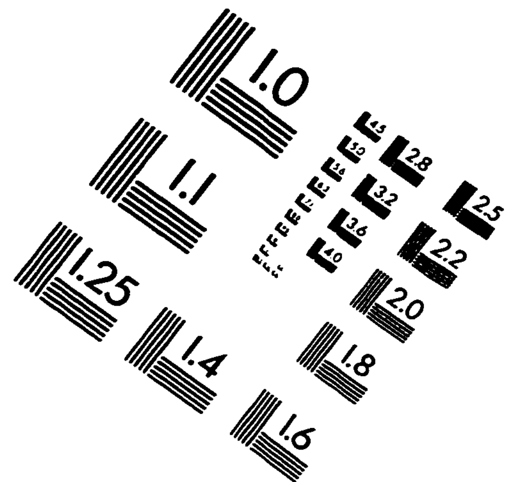
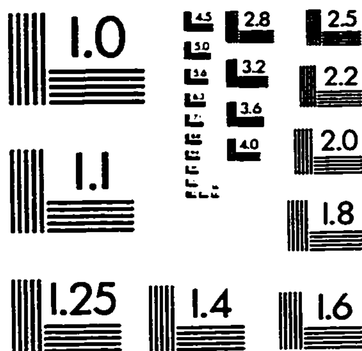
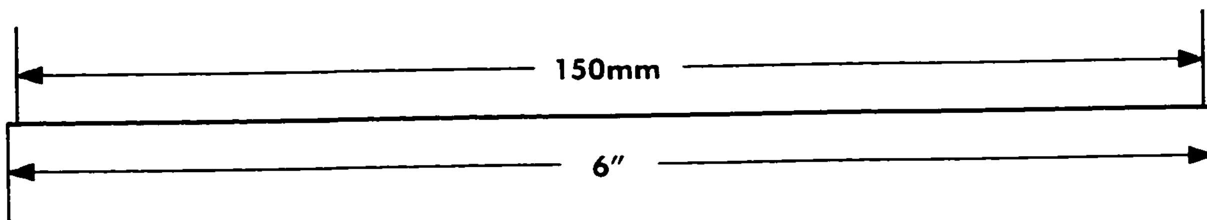
```
if ( maxOnesCnt > maxNullsCnt )
{
        grouping = 1;
        maxW = maxOnesCnt;
        maxN = onesSeqL;
        SeqListL = onesSeqL;
        for ( i = 0; i < onesSeqL; i++ )
                SeqList[i] = onesSeqList[i];

}
else if ( maxOnesCnt < maxNullsCnt )
{
        grouping = 0;
        maxW = maxNullsCnt;
        maxN = nullsSeqL;
        SeqListL = nullsSeqL;
        for ( i = 0; i < nullsSeqL; i++ )
                SeqList[i] = nullsSeqList[i];
}
else if( (maxOnesCnt = maxNullsCnt) && (onesSeqL >= nullsSeqL) )
{
        grouping = 1;
        maxW = maxOnesCnt; /*Max number of 1's matching bits */
        maxN = onesSeqL;
        SeqListL = onesSeqL;
        for ( i = 0; i < onesSeqL; i++ )
                SeqList[i] = onesSeqList[i];

}
else
{
        grouping = 0;
        maxW = maxNullsCnt; /* Max # of 0's matching bits */
        maxN = nullsSeqL; /* Max # of sequences */
        SeqListL = nullsSeqL;
        for ( i = 0; i < nullsSeqL; i++ )
                SeqList[i] = nullsSeqList[i];

}
printf ("\nBest Group is %d\n",grouping);
/***********************************************************
 *      Print the total number of 1's in each column       *
 ***********************************************************/

printf("\n");
for ( i = 0; i < SeqListL; i++ )
        SeqPrint(s->p->seq[SeqList[i]].seq);

SetItUp(mask);
*mask = 1L;
for(i = 0; i <= SeqListL; i++ )
        A[i] = 0;

for (j=0; j<bits_cnt; j++){
        k = 0;
```

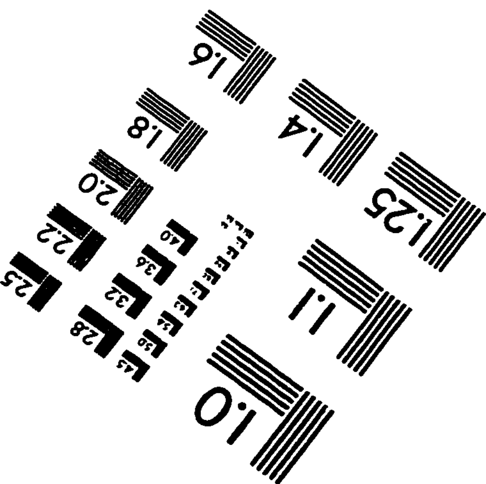
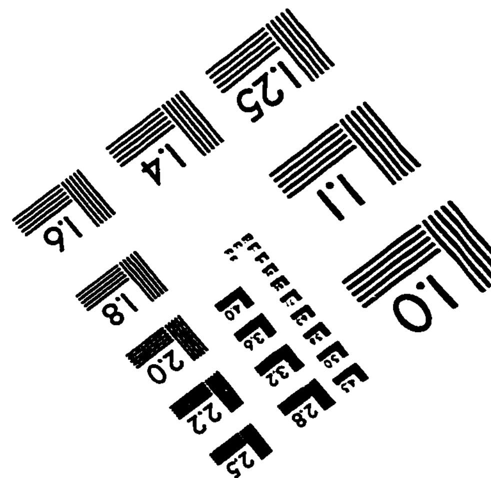
```
            for(i=0; i<maxN; i++)
            {
                    SeqAnd(s->p->seq[SeqList[i]].seq , mask, res);

                    k += HasItOne(res);
            }
            A[k]++;
            LeftShift(mask,mask);

    }

    for(i = 0; i <= SeqListL; i++ )
            printf("A[%d] = %d\n",i,A[i]);
    /****************************************************************
     *              Gate Selection                                  *
     ****************************************************************/
    /** If the "Best Group" is AND i.e. 1-weight is chosen, do: *****/


    printf(" maxN %d maxW %d\n", maxN,maxW);
    group_seq_number = maxN;
    printf ("\n\n\nW[%d] = %2d \n\n", 0, A[0]);
    printf ("W[%d] = %2d \n\n", group_seq_number ,
            A[group_seq_number ]);


    /******* Calculation of the RESIDUE *********/

    residue = bits_cnt - A[group_seq_number ] - A[0];
    printf ("\nResidue = %2d \n\n", residue);
    for(i = 0; i < SeqListL; i++)
    {
            printf("(%d)/%d ",s->p->OrigInd[SeqList[i]],
            getSeqId(s->p->OrigInd[SeqList[i]]));
    }
    printf(") are merged together into ");
    printf("(%d) or ", count);
    printf("%d.\n", count_Added_Seq);
    printf("SEQ_LIST_L = %d\n",SeqListL);
    Copy(s->p->seq[SeqList[0]].seq,_tmp);




    if(grouping)
    {
            printf("\nWN1 -  (residue/maxN)=%f\n", (float)A[group_seq_number]-
((float)residue/(float)maxN) );

            if ( (float)A[group_seq_number] > ((float)residue/(float)maxN))
            {
                    if( andXor(maxN, bits_cnt,A) )
                    {
                            andGate (s,SeqListL, SeqList, _tmp);
                    }
                    else
```

```
				{
					xorGate (s,SeqListL, SeqList, _tmp);
				}
			}
			else
			{
				if ( Summation(maxN, bits_cnt, &A[0], 1)
				>= Summation(maxN, bits_cnt, &A[0], 0))

				{
					if( andXor(maxN, bits_cnt,A) )
					{
						andGate (s,SeqListL, SeqList, _tmp);
					}
					else
					{
						xorGate (s,SeqListL, SeqList, _tmp);
					}
				}
				else
				{
					if( orXor(maxN, bits_cnt,A) )
					{
						orGate (s,SeqListL, SeqList, _tmp);
					}
					else
					{
						xorGate (s,SeqListL, SeqList, _tmp);
					}
				}
			}
		}
		else
		{
			printf("maxN %d\n",maxN );
			printf("\nWN0 -  (residue/maxN)=%f\n", (float)A[0]-
((float)residue/(float)maxN) );
			if ( (float)A[0] > ((float)residue/(float)maxN))
			{
				if( orXor(maxN, bits_cnt,A) )
				{
					orGate (s,SeqListL, SeqList, _tmp);
				}
				else
				{
					xorGate (s,SeqListL, SeqList, _tmp);
				}
			}
			else
			{
				if ( Summation(maxN, bits_cnt, &A[0], 1)
				>= Summation(maxN, bits_cnt, &A[0], 0))

				{
					if( andXor(maxN, bits_cnt,A) )
					{
```

```
                        andGate (s,SeqListL, SeqList, _tmp);
                    }
                    else
                    {
                        xorGate (s,SeqListL, SeqList, _tmp);
                    }

                }
                else
                {
                    if( orXor(maxN, bits_cnt,A) )
                    {
                        orGate (s,SeqListL, SeqList, _tmp);
                    }
                    else
                    {
                        xorGate (s,SeqListL, SeqList, _tmp);
                    }
                }
            }
        }
        SeqPrint(_tmp);
        Copy(_tmp,new_seq.seq);
        new_seq.cnt = count_1bits(_tmp);
        Add(s->p, &new_seq);
        Delete(s->p, SeqList, SeqListL);
    }
}
```

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved