SPACE COMPACTOR DESIGN FOR BUILT-IN SELF-TESTING OF VLSI CIRCUITS FROM COMPACT TEST SETS USING SEQUENCE CHARACTERIZATION AND FAILURE PROBABILITIES

By

Mansour H. Assaf, B.A.Sc.

A dissertation submitted to the School of Graduate Studies and Research of the University of Ottawa in partial fulfilment of the requirements for the Degree of Master of Applied Sciences

Ottawa-Carleton Institute for Electrical Engineering
Department of Electrical Engineering
Faculties of Engineering
University of Ottawa

August 1996

ISBN 0-612-19927-4

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

A new space compaction technique for built-in self-testing (BIST) of VLSI circuits using compact test sets with the primary objective of minimizing the storage required for the circuit under test (CUT) while maintaining fault coverage information is presented in this dissertation. The compaction technique utilizes the concepts of the Hamming distance and sequence weight, together with the use of failure probabilities of the errors in the selection of specific gates for merger of a pair of output streams from the CUT. The outputs coming out of the space compactor may eventually be fed into a time compactor (syndrome counter) to derive the signature for the circuit. The proposed technique guarantees a simple design with a very high fault coverage for single stuck-line faults, with low CPU simulation time, and acceptable area overhead.

Design algorithms are proposed and the simplicity and ease of implementation are demonstrated with numerous examples. Specifically, extensive simulation runs on ISCAS 85 combinational circuits with FSIM, ATALANTA, and COMPACTEST confirm the efficiency of the suggested approach under conditions of stochastic independence as well as dependence of single line and double line output errors.

A performance comparison of the designed space compactor with a conventional linear parity tree compactor is also presented.

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation and gratitude to **Dr. Sunil R. Das**, Professor, Department of Electrical Engineering at the University of Ottawa, Ottawa, Ontario, Canada for his guidance and immense support throughout the progress of this research.

I would also like to thank **Dr. Amiya R. Nayak**, School of Computer Science, Carleton University, Ottawa, **Dr. Krishnendu Chakrabarty**, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan, U.S.A., and **Dr. Wen-Ben Jone**, Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, Republic of China for their comments and many help during the progress of this work. 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 us ATALANTA and FSIM fault simulators for our research.

I finally would like to thank my family members for their support and help during the past two years I was occupied with my academic pursuit.

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1. INTRODUCTION

As the electronics industry continues to grow, complex systems and high level of integration continue to increase, better and more effective testing methods that ensure reliable operations of chips, integrated in complex digital systems, are always needed. The concepts of testing have broad applicability. Consider for example medical tests and diagnoses instruments, aeroplane controllers and other safety-critical systems that have to be tested before use (off-line testing) and during use (on-line testing). Similarly, an application where failure can have severe economic consequences is real-time transaction processing.

The testing process must be fast and effective to guarantee that such systems operate correctly. Finding highly sophisticated testing techniques that ensure correct system performance has become increasingly important for two main reasons [20,7,27]:

- Technology-based reason. The cost of testing integrated circuits (ICs) is expensive; it ranges from 35% to 55% of its total manufacturing cost. Besides, testing a chip is time consuming which may take up to one-half of the design cycle time [71].

- Market-based reason. The amount of time available for manufacturing, testing, and marketing a product continues to decrease. Moreover, as a result of global competition, customers demand lower cost and better quality products. Therefore, to achieve this higher quality at lower cost, testing techniques have to be improved.

1.1 The Concept of Testing

The concept of testing systems consists of exercising a certain system (say, testing a PC card or debugging a computer program) and observing its resulting response to ascertain whether it behaves correctly.

Digital system denotes a combination of complex digital circuits, whereas a digital circuit may represent a certain number of integrated circuits. Testing of digital systems can be done at different levels of abstraction describing their operations (Table 1.1) [17]. The type of information processed by the circuit characterizes the level of abstraction. At the system level (the higher level of abstraction), the data processed by the circuit are of type: messages. Testing is often performed by software; while an effective software test may suffice at this level, it may require too much time and may be expensive to develop. At the transistor level (the lowest level of abstraction), the processed information are voltage and current of analog types. Testing at this level deals with the detection of malfunctions in the operation of an integrated circuit (IC) which can be due to fabrication defects or component wear-out. Faults, at this level, are of physical type such as discontinuities in polysilicon gate connections, metal-to-metal shorts, or incorrect transistor threshold voltages. This type of faults is difficult to analyze directly and it is more convenient to represent them by their effects on the logical operation of the system.

LEVEL OF ABSTRACTION	CONTROL	DATA
1. System level	Messages	Messages
2. Processor level	Programs	Data structures
3. Instruction set level	Instructions	Words
4. Register level	Logic values	Words
5. Logic level	Sequences of logic values	Sequences of logic values
6. Transistor level	Voltage and current analog values	Voltage and current analog values

Table 1.1 Levels of abstraction describing the complexity of digital circuits.

At the logic level, the information processed is represented by discrete logic values: binary values (0 and 1), ternary values or any multiple valued logic.

The concept of error has different meanings at different levels. At the logic level an error usually means an incorrect binary value and represents the effect of physical faults on the logical behavior of the system. The standard fault model is the single stuck-at fault model (SSF) and is the most widely used logical fault model for the following reasons:

- It represents many different physical faults;
- The number of SSFs in a circuit is small compared to other fault models;
- SSFs can be used to model other type of faults as well.

A signal line which is stuck at a fixed logic value v is symbolically represented by $s\text{-}a\text{-}v$ where v can be either the logic value 0 or 1.

The conventional testing process of digital circuits consists of applying test patterns generated by a test pattern generator (TPG) to the circuit under test (CUT) and comparing the responses with known correct responses. Therefore, testing digital circuits requires knowledge of the fault-free response of the circuit to the test set. In practice, for large circuits, the large storage requirement for the fault-free responses makes the test procedure very expensive, and so different methods have been proposed for minimizing the amount of memory needed [46].

The quality of a test and its effectiveness (fault coverage) is determined by calculating the ratio of the number of faults detected to the total number of faults injected into the circuit. To determine the percentage of fault coverage, a circuit is simulated by applying test patterns, injecting faults into the circuit under test, and observing the responses at the outputs of the circuit to identify all faults detected by the applied test set. The above process is called fault simulation and is shown in Figure 1.1.

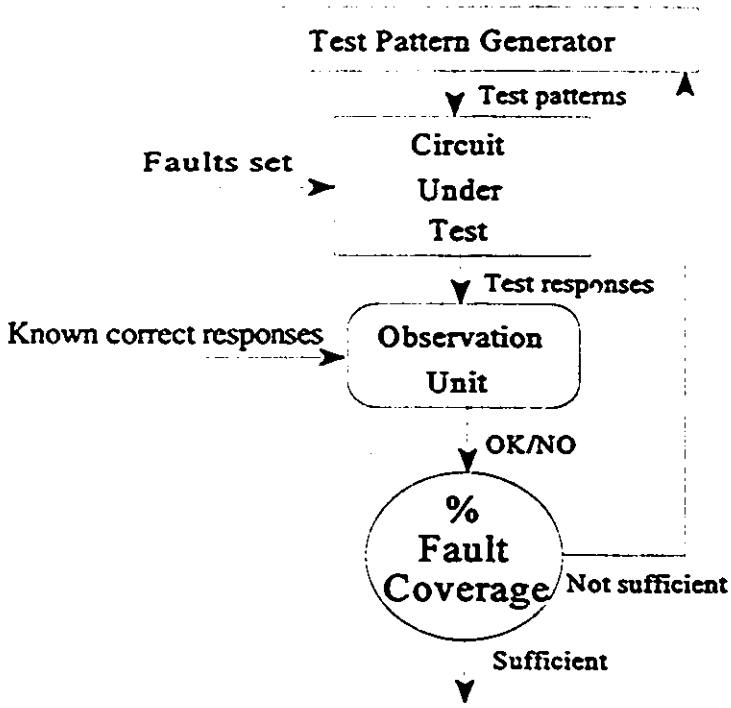


Figure 1.1 The fault simulation process.

An input pattern $x_1 x_2 \ldots x_n$ where n represents the number of primary input lines is called a test for a certain fault y (s-a-0 or s-a-1) if it causes the effect of y to appear as an error at a primary output z_i of the circuit. This is done by placing an appropriate value on the faulty line and creating a path (sensitized path) along which an error due to a fault can propagate to an observable output. Consider the circuit shown in Figure 1.2. An input pattern $x_1 x_2 x_3 x_4 = X110$ (where X represents an input "Don't Care" which can be either 0 or 1) can be used as a test to simulate the circuit, both without and with the fault y (s-a-0) present. The results of the simulations are represented by v/v_f where v and v_f are the corresponding signal values in the fault-free and faulty circuits. The fault y (s-a-0) is detected since the output value $v = 1$ is different than $v_f = 0$ shown at the primary output z_2. The created sensitized path from y to z_2 is shown in the figure below.

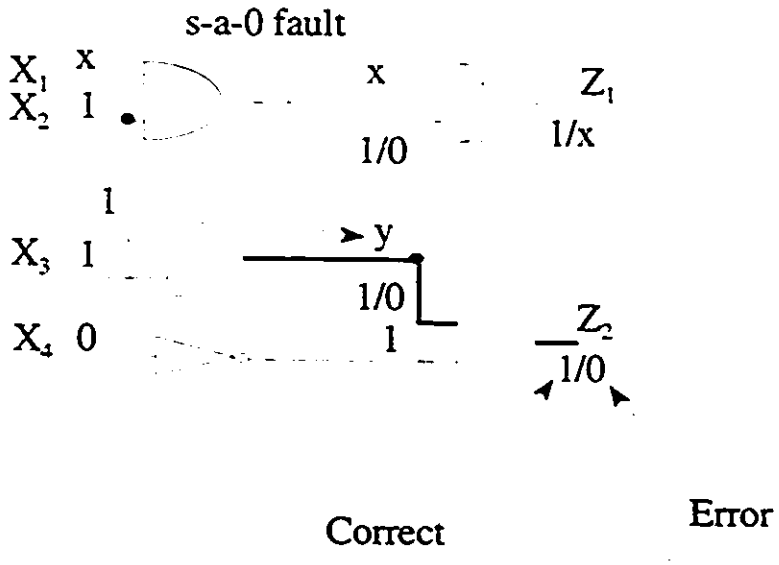


Figure 1.2 Fault propagation and sensitized path.

Test generation depends on the cost of generating a set of test patterns, on the quality of the generated tests, and on the cost of applying these test patterns to test digital circuits.

Logic circuits can be tested exhaustively by applying all possible input combinations to combinational logic circuits and by fully exercising the circuit's state table in the case of sequential logic circuits. Figure 1.3 illustrates the concepts of exhaustive testing for the C17 circuit in the ISCAS 85 benchmark series [9]. It can be exhaustively tested by applying all $2^5 = 32$ possible input patterns.

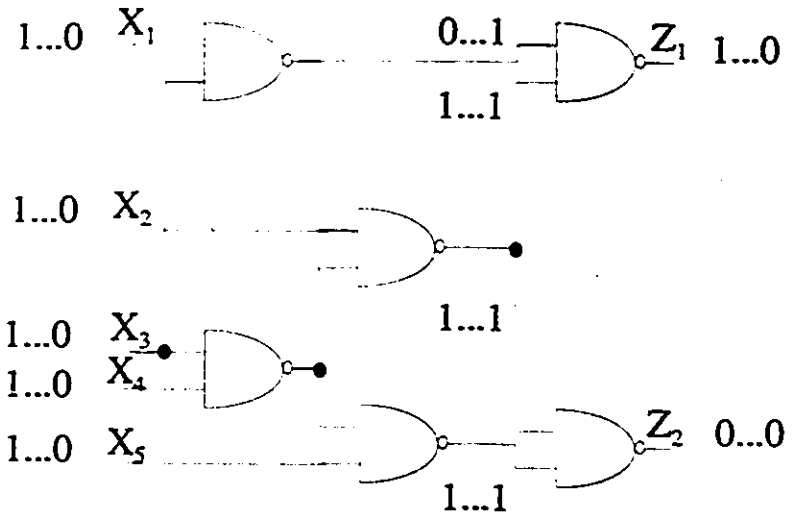


Figure 1.3 The concept of exhaustive testing.

The cost of exhaustive testing is high because computational effort to generate tests grows exponentially with the circuit size (number of primary inputs).

The amount of time needed for exhaustive testing can be reduced by partitioning the circuit under test (CUT) into subcircuits with smaller number (less than 20) of primary inputs; each subcircuit is tested exhaustively then by applying up to 2^{20} input patterns and the whole process is termed pseudoexhaustive testing [49].

Random testing is another test generation method where input test patterns are generated randomly [7], or pseudorandomly since no true random test exists.

Random testing is less expensive than deterministic (exhaustive) testing. However, it is a time consuming process; to achieve a high fault coverage, a high random test length is usually needed. In practice, low test length with high fault coverage is desirable for testing. As a practical industrial example, the RISC System/6000 processor by IBM is tested randomly. Exhaustive testing, pseudoexhaustive and pseudorandom testing are circuit-independent test generation methods.

Test generation methods can be circuit-dependent which in turn can be divided into two categories:

- Fault-oriented
- Fault-independent

The goal of fault-oriented method is to generate a test for a specified target fault. The fundamental steps in generating a set of tests for a circuit are first to determine the initial set of faults of interest, next to select a target fault, and then to maintain the set of remaining undetected faults. The D-algorithm [60], PODEM [32], and FAN [30] are typical circuit-dependent/fault-oriented techniques.

The D-algorithm uses the propagation, justification, and implication procedures to find a test pattern for a selected fault. Briefly, the propagation procedure is used to propagate an error from a faulty line to an observable primary output, the justification procedure determines consistent values for unassigned inputs of a module whose output is assigned, and the implication procedure refers to the process of computing all values of signal lines that can be uniquely determined from the assigned signal lines in the circuit.

The propagation and justification procedures involve making decisions. If decisions cannot be made, at some point, without invalidating a value already assigned to a line in the circuit, the D-algorithm returns to an earlier decision point and makes an alternative decision; this process is termed backtracking.

A characteristic feature of the D-algorithm is its ability to propagate errors on several reconvergent paths (multiple-path sensitization). This feature is required to detect certain faults that would not be detected by sensitizing only a single path.

PODEM (Path-Oriented Decision Making) is a test generation algorithm characterized by a direct search, in which decisions consist only of primary input assignments. PODEM differs from the D-algorithm in that it does not require the justification procedure. By only assigning values to the primary inputs (PI), PODEM reduces the number of backtracks needed to find a test.

The FAN (Fanout-Oriented) test generation algorithm [32,30,20] differs from PODEM in two major extensions it introduces to the backtracing concept.

- Backtracing in FAN may stop at internal lines, rather than stopping at primary inputs (PIs) as in PODEM.
- FAN uses a multiple backtrace procedure that attempts to satisfy simultaneously a set of objectives.

In PODEM, the fault-activation and the error-propagation problems, mapped into a set of line-justification problems, become two different objectives that are individually backtracked. Both objectives might be satisfied with an appropriate PI assignment, and to minimize the number of backtracks required to find the best assignment, FAN uses the multiple backtrace procedure; this procedure processes every current objective until all objectives that belong to a specific set are traced. Critical path test generation is an example of the circuit-dependent/fault-independent method. Its goal is to derive a set of tests that detect a large set of single stuck faults (SSFs) without targeting individual faults. It generates tests that produce long critical paths.

Two basic procedures are involved in the critical-path test generation algorithm. First, the selection procedure, in which a primary output (PO) is selected and a critical 0-value or 1-value is assigned to it. The second one is the justification procedure, in which the primary output value is justified recursively, trying to justify any critical value on a gate output by critical values on the gate inputs.

Compared to fault-oriented test generation, critical-path test generation offers the following advantages:

- The SSFs detected by the generated tests are immediately known without using a separate fault simulation step, as is required with a fault-oriented algorithm.
- To avoid the duplicated effort inherent in a fault-oriented algorithm, the critical-path algorithm generates a new test by modifying the critical paths obtained in the previous test.

Unlike a fault-oriented algorithm, a fault-independent algorithm cannot identify undetectable faults.

Traditionally, testing is done using Automatic Test Equipments (ATEs) that apply test patterns, generated automatically using a computer model of the device under test or manually by the design engineer, to the input pins of the VLSI chip or Unit Under Test (UUT) and observe the responses at the output pins. Input test patterns and known fault-free responses are stored in a memory and are to be used to compare the responses at the outputs of the UUT to its correct known responses, as shown in Figure 1.4.

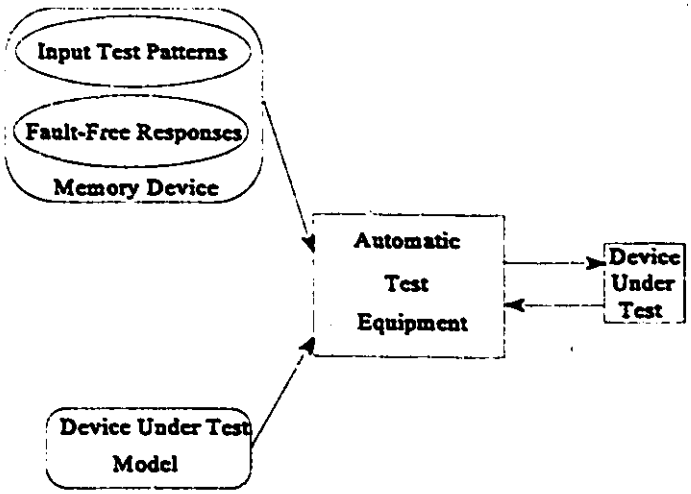


Figure 1.4 Testing chips using Automatic Test Equipment (ATE).

Several problems affecting the manufacturing cost and the testing quality are associated with the use of ATEs to test complex VLSI chips [7.20], such as:

- Commercial ATEs are very expensive.
- As the number of transistors in a single chip continues to grow with respect to the number of input and output pins, activating faults by applying appropriate test patterns at the input pins and observing errors at the output pins is no longer easy to do; this adversely affects the quality of testing.
- As the circuit size increases, the number of test stimuli and its corresponding fault-free responses increase and become too large to be handled efficiently by the ATE's memory.
- ATE usually applies test patterns at a slower rate than the circuit's normal operating speed and since some faults can only be detected at the system's operating speed, the testing quality is hence once again affected.

To simplify test generation and to improve the testing quality, design engineers usually use design for testability (DFT) techniques that make circuits much easier to test. Design and test become closely related issues and the ease with which complex digital circuits can be tested is a major issue in the design process. Design for testability techniques reduce the cost of testing by introducing testability criteria early in the design stage. In fact, DFT techniques simplify testing and solve the controllability and observability problems but at the same time require additional logic and I/O pins (higher chip area) which can cause loss of information.

The most popular and widely used DFT technique for external testing is scan design [28], in which the memory elements are separated from combinational circuits during the testing process. There are several forms of scan designs; they differ primarily in how the scan cells are designed. Usually, flip-flops are chained all together into a shift register when selecting a test mode, and the circuit is partitioned into combinational blocks. Test vectors set for these combinational blocks can be generated easily. A shift register is used to scan-in all input data (tests set) to each combinational block and then results can be scanned-out for response verification. The concept of scan design is illustrated in Figure 1.5.

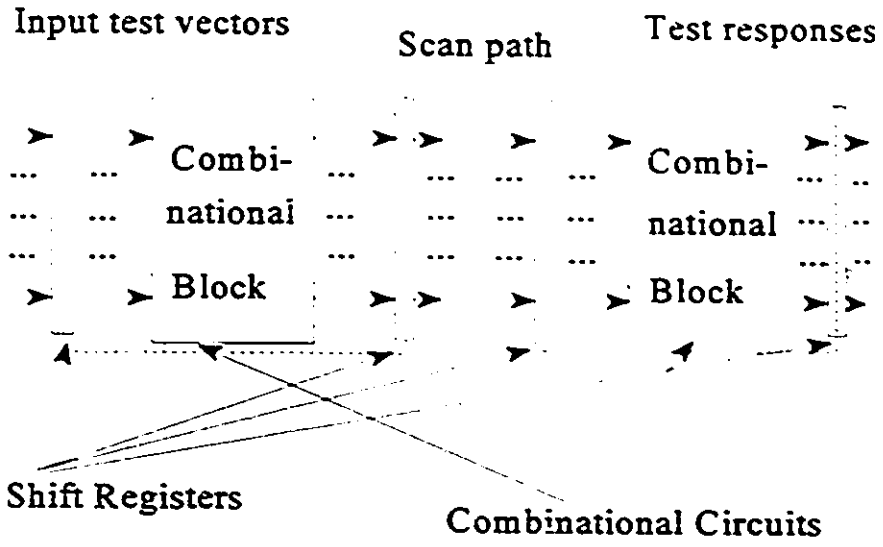


Figure 1.5 The scan-based design for testability techniques.

Several attributes are associated with the use of scan design such as:

- Scan designs are expensive in terms of silicon area.
- Additional input/output pins are required.
- Test time per pattern is increased, since the test vectors set needs to be shifted into the scan path.
- The percentage of fault coverage can be higher and the cost of test generation is much lower than other design techniques costs.
- Flip-flops and latches are more complex and introduce extra delay which may require a slower clock rate, resulting in loss of information and a degradation in performance.

Algorithms for test generation (TG) and design for testability techniques are usually used when external testing is employed. As we have seen, these different methods have some advantages and disadvantages over each other. In the next section, we discuss built-in self testing (BIST), a design technique in which parts of a circuit are used to test the circuit itself, providing an efficient test methodology that can be used at the circuit, chip, board, or system level of design.

1.2 Built-In Self-Test (BIST)

Built-in self-testing (BIST) is a design methodology that has the capability of solving most of the problems encountered in testing digital systems. It combines both concepts of built-in test (BIT) and self-test (ST) in one termed built-in self-test (BIST).

In BIST, test generation, test application, and response verification procedures are accomplished through built-in hardware [3]. It allows different parts of a chip to be tested in parallel, reducing the required testing time. It also simplifies the need for external test equipments. As the cost of testing is becoming the major component of the manufacturing cost of a new product, BIST reduces manufacturing, test, and maintenance costs, and improves diagnosis. Several companies like Motorola, AT&T, and IBM have included BIST in their products. For example, the Motorola 68020 microprocessor [23] is tested using BIST techniques. The microcode ROM is self-tested in the Intel 80386 microprocessor [17]. Similarly, AT&T has incorporated BIST into more than 200 of their chips [4].

On the whole, a test pattern generator (TPG) sends its output to a circuit under test (CUT) as shown in Figure 1.6, and output streams are fed into a test data analyzer. A fault is detected if the test sequence is different from the response of the fault-free circuit. The test data analyzer consists of a response compaction unit (RCU), a storage for the fault-free responses of the CUT, and a comparator.

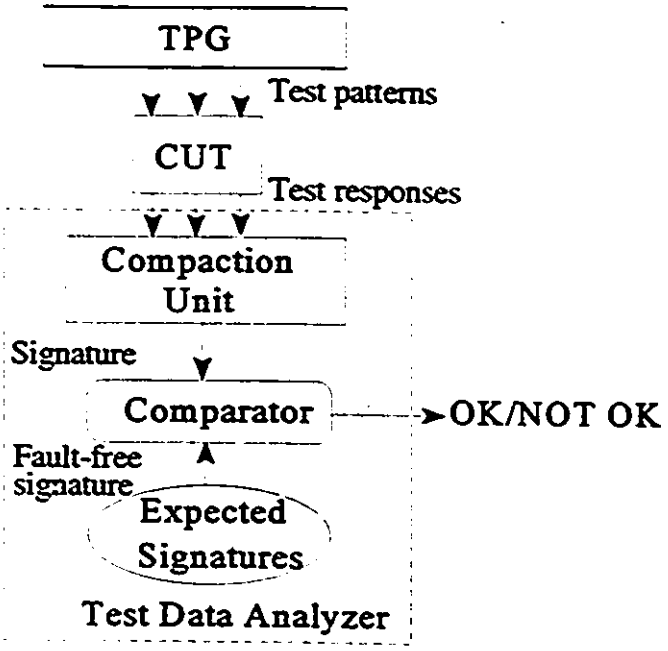


Figure 1.6 The BIST environment.

To reduce the amount of data represented by the fault-free and the faulty CUT responses, data compaction is used to create signatures (short binary sequences) from the CUT and its corresponding fault-free circuit. Signatures are compared and faults are detected if a match does not occur.

BIST techniques may be used during normal functional operating conditions of the unit under test (on-line testing), as well when a system is not carrying out its normal functions (off-line testing). In the case where detecting real-time errors is not that important, systems, boards, and chips can be tested in off-line BIST mode.

BIST techniques use pseudorandom, pseudoexhaustive test pattern generators or on-chip storing of reduced test sets. Today, testing logic circuits exhaustively is no more used, since only few test patterns are needed to ensure full fault coverage for single stuck type faults [35,53]. Reduced pattern test sets can be generated using algorithms such as FAN, and others. Built-in test generators can often generate such reduced test sets at low cost, making BIST techniques suitable for on-chip self-testing.

This dissertation focuses on the response compaction process of built-in self-testing techniques which translates into a process of reducing the test response to a signature; instead of comparing bit-by-bit the fault-free responses to the observed outputs of the CUT as in conventional testing methods, the observed signature is compared to the correct one, thereby reducing the storage needed for the correct circuit responses.

A block diagram of the output data compaction scheme is shown in Figure 1.7. The test data analyzer consists of a compaction unit, a comparator, and a storage (memory device).
The compaction unit can be divided into a space compaction unit and a time compaction unit as shown below.

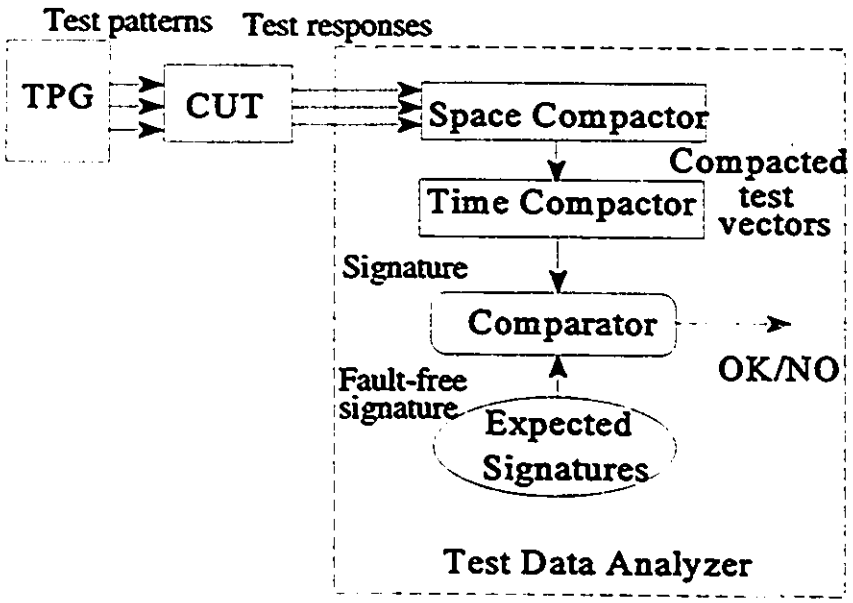


Figure 1.7 The test response compaction scheme.

In general, s input sequences coming from a CUT are fed into a space compactor, providing t output streams of bits such that t << s; most often, test responses are compressed into one sequence (t=1).

Space compaction brings a solution for the problem of achieving high-quality built-in self-testing of complex chips without monitoring a large number of internal test points; it reduces testing time and area overhead by merging test sequences coming from these internal test points into a single stream of bits. This single bit stream of length r is fed into a time compactor and a shorter one of length q ($q < r$) is obtained at the output.The extra logic representing the compaction circuit must be as simple as possible, to be easily embedded within the circuit under test and should not introduce signal delays that affect either the test execution time or the normal functionality of the circuit being tested. In addition, the length of the signature must be as short as it can be in order to minimize the amount of memory needed to store the fault-free signatures. Also, signatures obtained from faulty output responses and their corresponding fault-free signatures should not be the same, which unfortunately is not always the case.

A fundamental problem with compaction techniques is error masking (aliasing), which occurs when the signatures of a faulty output response maps to the fault-free signature, usually calculated by identifying a good circuit, applying test patterns to it, and then have the compaction unit generate the fault-free reference. Aliasing causes loss of information, which affects the testing quality of BIST and reduces the fault coverage (the number of faults detected, after compaction, over the total number of injected faults). Several methods have been suggested for computing the aliasing probability. The exact computation of this aliasing probability is known to be an NP-hard problem. In practice, high fault coverage is required (over 99%). Therefore, any space compression technique which maintains more percentage error coverage information would have to be considered for investigation.

This present dissertation deals with designing and analyzing an efficient space compaction technique for built-in self-testing of VLSI circuits. It consists of constructing a compaction tree for combinational circuits based on the inherent properties of the output sequences of the circuit under test and the knowledge of failure probabilities. The major objective is to develop a method for space or space-time compaction that is simple, suitable for on-chip self-testing, requires low area overhead, and has little adverse impact on circuit performance. To that effect, mergeability criteria of output sequences were developed and the effect of error probabilities on these criteria was analyzed. The fault coverage on the assumption of single stuck-at fault model was also experimentally evaluated.

1.3 Outline of the Dissertation

The material in this dissertation is organized as follows:

Chapter 1 provides an overview of different conventional testing methods for digital circuits, test generation methods, built-in self-testing, and a brief introduction of what is going to be discussed in the present dissertation.

Chapter 2 discusses several existing space and time compaction techniques.

Chapter 3 introduces certain important properties of circuit responses to develop mergeability criteria for the design of space compactor on the assumption of stochastic independence of single and double line errors.

Chapter 4 designs compaction tree for multi-output circuits assuming stochastic dependence of single and double line errors.

Chapter 5 provides experimental results on extensive simulations of the ISCAS 85 combinational benchmark circuits.

Chapter 6 provides discussions on the results.

Chapter 2

2. OVERVIEW OF TEST COMPACTION TECHNIQUES

In this chapter, we review various compaction techniques for BIST that have been proposed in the literature. We briefly describe them, concentrating on some of their properties, like the area overhead, fault coverage, signature length, and the error masking problem.

2.1 Time Compaction

In this section, we examine some time compaction methods like Ones counting, Syndrome testing, Transition counting, Signature analysis, and others.

2.1.1 Ones counting

Ones counting [36] uses as its signature the number of ones in the binary circuit response stream. The hardware that represents the compaction unit consists of a simple counter (Figure 2.1 (a)), and is independent of the CUT; it only depends on the nature of the test response. Signatures' values do not depend on the order in which the input test patterns are applied to the CUT. The length of the signature is a logarithmic factor of the length of the output response data; therefore, if the circuit response is of length r bits, then its signature will be of length $\lceil \log_2 r \rceil$.

The probability distribution function of aliasing is approximately Gaussian with a mean value at $r/2$ where r is the length of the output stream; therefore, it takes a maximum value whenever the signature length approaches $r/2$ as shown in Figure 2.1 (b).

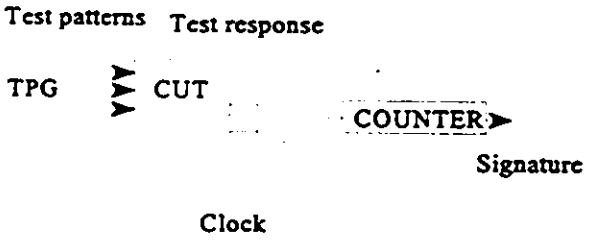


Figure 2.1 (a) The Ones counting hardware implementation.

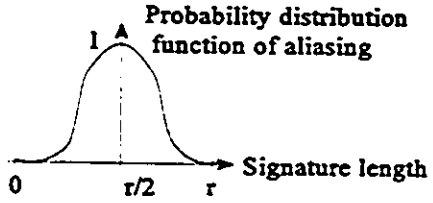


Figure 2.1 (b) The probability of aliasing.

2.1.2 Syndrome testing

Syndrome testing [62] differs from ordinary Ones counting in that a circuit can be designed for zero masking. The syndrome S equals K over 2^n , where K is the number of minterms of the function implemented by the single-output CUT, while 2^n represents all input test patterns applied to the n-input combinational circuit. Like Ones counting, the signature in Syndrome testing is test order independent.

Circuits implementing logic functions can be transformed, by adding extra inputs, to circuits in which every single stuck-at line fault is syndrome-testable.

2.1.3 Transition counting

Transition counting [37] counts the number of times the output bit stream changes from 1 to 0 and vice versa. In Transition counting, the signature length is less than or equal to $\lceil \log_2 r \rceil$, where r is the length of a response stream.

Since what is counted is the number of 0-to-1 or 1-to-0 transitions, it is obvious that the signature depends on the order in which input test patterns are applied to the CUT. The masking

probability takes high values when the signature value is close to *r/2* and low values when it is close to 0 or *r*. Transition count testing does not guarantee the detection of all single-bit errors. The hardware implementation of Transition counting consists of an XOR logic gate, D flip-flop, and a counter as it is sketched in Figure 2.2.

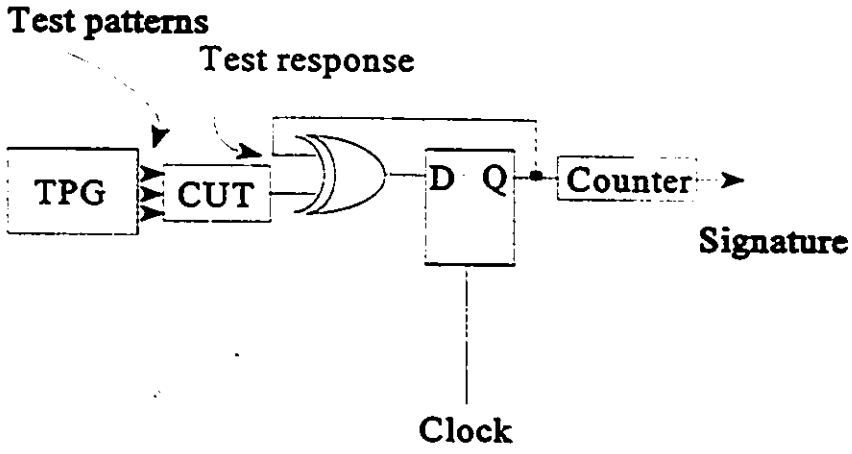


Figure 2.2 The Transition counting implementation.

2.1.4 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 detect 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.

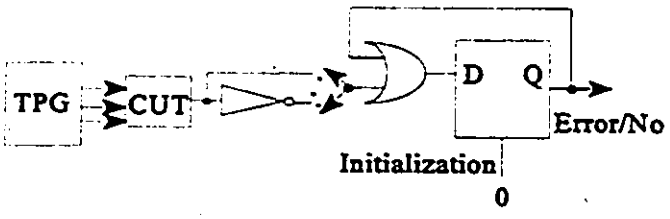


Figure 2.3 (a) DTC implementation for testing single-output circuits.

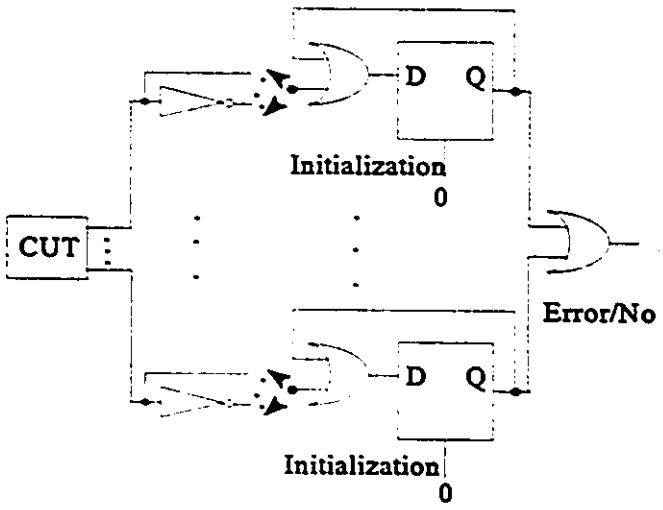


Figure2.3 (b) MTC implementation for testing multi-output circuits.

2.1.5 Walsh spectral analysis

In Walsh spectral analysis [38], switching functions are represented by their spectral coefficients which are compared to known correct coefficient values. In a sense, the truth table of the switching function is verified. The process of collecting and comparing a subset of the complete Walsh functions is described as a mechanism for data compaction.

The use of spectral coefficients promises higher percentage of error coverage, but also requires higher area overhead for generating them which is seen as a disadvantage [68]. Figure 2.4 shows a hardware implementation of Walsh spectral testing scheme.

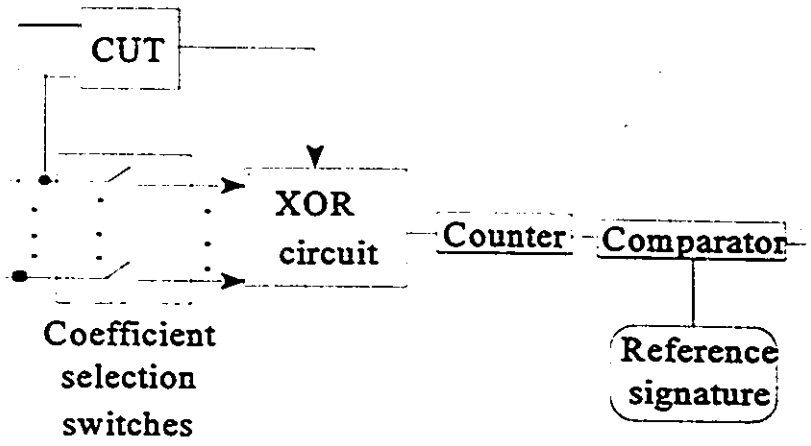


Figure 2.4 The Walsh spectral testing scheme.

2.1.6 Parity checking

In Parity checking [7] the response bit stream coming from a circuit under test reduces a multitude of output data to a signature of length 1 bit. The signature is a single bit whose value equals the parity of the test response sequence. Parity checking detects all errors involving an odd number of bits, while faults that give rise to an even number of error bits are not detected. This method is relatively ineffective since a large number of possible response bit streams from a faulty circuit will result in the same parity as the correct stream. All single stuck-at line faults in fanout-free circuits are detected by Parity check technique. The hardware implementation consists of a flip-flop and an XOR gate.

2.1.7 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).

Different methods for computing and reducing the aliasing probability have been proposed, such as the signature analysis model proposed by Williams *et al.* which uses Markov chains and derive an upper bound on the aliasing probability in terms of the test length and the probability of an error occurring at the output of the CUT [73]. Another approach to the computation of aliasing probability is presented in [55]. 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 playing with the length of the LFSR or by using different characteristic polynomial G(x). As demonstrated in [65], 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.

Multi-output circuits can be tested using multiple-input signature registers (MISRs) (Figure 2.6), eliminating the need for a space compactor. Unfortunately, MISRs increase aliasing and require extra hardware which make it far from being practical.

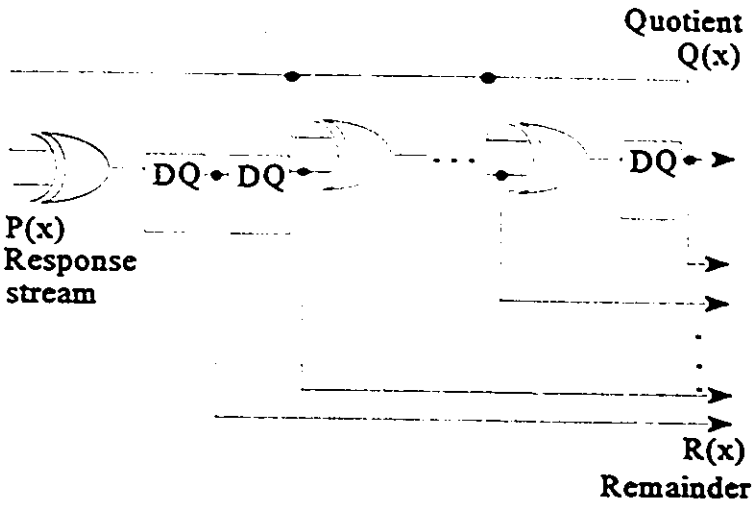


Figure 2.5 Signature analysis using LFSRs.

Zorian and Agarwal [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.

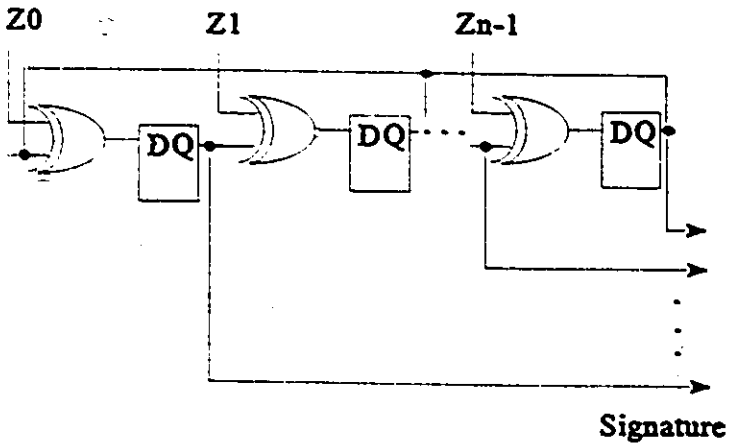


Figure 2.6 MISRs hardware implementation.

2.1.8 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 [64]. 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.

2.2 Space Compaction

In this section we examine several space compaction techniques that have been proposed in the literature or are in actual industrial use. Some of the common ones are: Parity tree space compactor, Hybrid space compression, Dynamic space compression, Quadratic functions compactor, Programmable space compactor, and Cumulative balance testing.

2.2.1 Parity tree analysis

The parity tree circuits [58] are composed of only XOR gates. An XOR gate has very good signal/error propagation properties which are desirable properties of space compactors. Functions realized by parity tree compactors are of the form $Z = Z_1 \oplus Z_2 \oplus \ldots \oplus Z_k$.
The parity tree space compactor propagates all errors that appear on an odd number of its inputs. Thereby, errors that appear on an even number of circuit inputs are masked. As experimentally demonstrated, most single stuck-at line faults are detected using pseudorandom input test pattern generators and reduced test sets. The parity tree compaction scheme is shown in Figure 2.7.

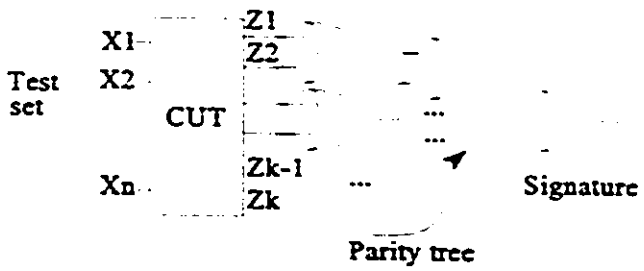


Figure 2.7 The parity tree diagram.

2.2.2 Hybrid space compression

Hybrid space compression (HSC) technique, originally proposed by Li and Robison [47], uses AND, OR, and XOR logic gates as output compression tools to compress the multiple outputs of the CUT into a single line (Figure 2.8). The compaction tree is constructed based on the detectable error probability estimate $\varepsilon = S1 \times R1/(2 \times L_s) + S2 \times R2/L_s$.

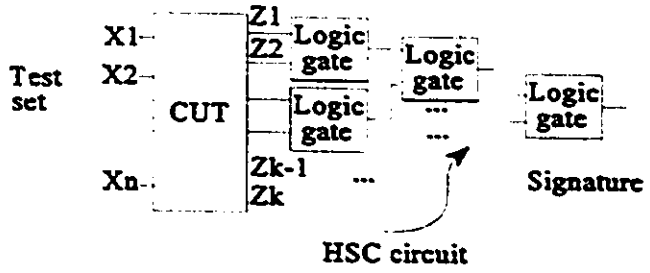


Figure 2.8 Hybrid space compression scheme.

2.2.3 Dynamic space compression

A modified version of the HSC method was proposed by Jone and Das [40]. Instead of assigning a static value for the probabilities of single errors S1 and double errors S2, Dynamic space compression (DSC) method dynamically estimates those values based on the CUT structure during the computation process. The values of S1 and S2 are determined based on the number of single lines and shared lines connected to an output as shown in Figure 2.9 where X and Y, connected to O_i and O_j, respectively, represent two single lines, and Z , connected to both outputs, represents a shared line.

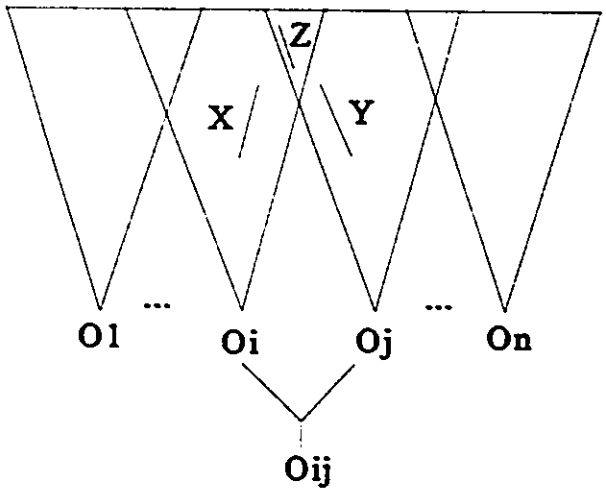


Figure 2.9 Estimation of S1 and S2 based on the CUT structure.

A general theory to predict the performance of space compression techniques was also developed in [40]. Experimental results show that the information loss, combined with separate syndrome counting technique, is between 0% and 12.7 %. DSC was later improved in [24], in which some circuit-specific information is used to calculate the probabilities. However, either HSC or DSC does not provide an adequate measure of fault coverage because they both rely on estimates of error detection probabilities.

2.2.4 Quadratic functions compaction

Quadratic functions compaction (QFC) [42] 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 $Z_0Z_1 \oplus Z_2Z_3 \oplus \ldots \oplus Z_{(2k-2)}Z_{(2k-1)}$ where Z_t and $Z_{(t+1)}$ 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 2.10.

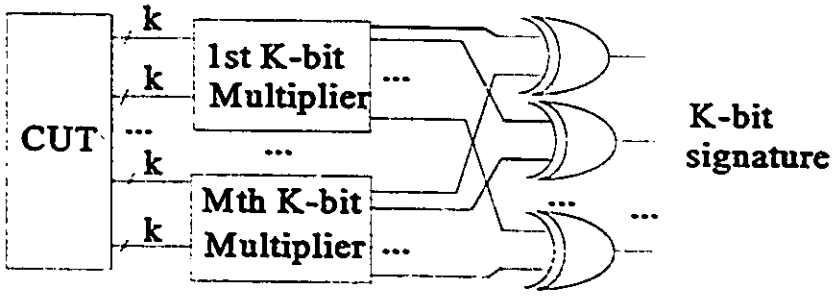


Figure 2.10 The QFC implementation diagram.

As one notices, the quadratic compressor scheme requires higher 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.

2.2.5 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 $(2^h)^k$ k-input Boolean functions, where k represents the number of primary outputs of the CUT.

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

2.2.6 Multiplexed parity trees

A new class of space compactors, based on parity tree circuits, was recently proposed by Chakrabarty and Hayes in [17]. 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 2.11 .

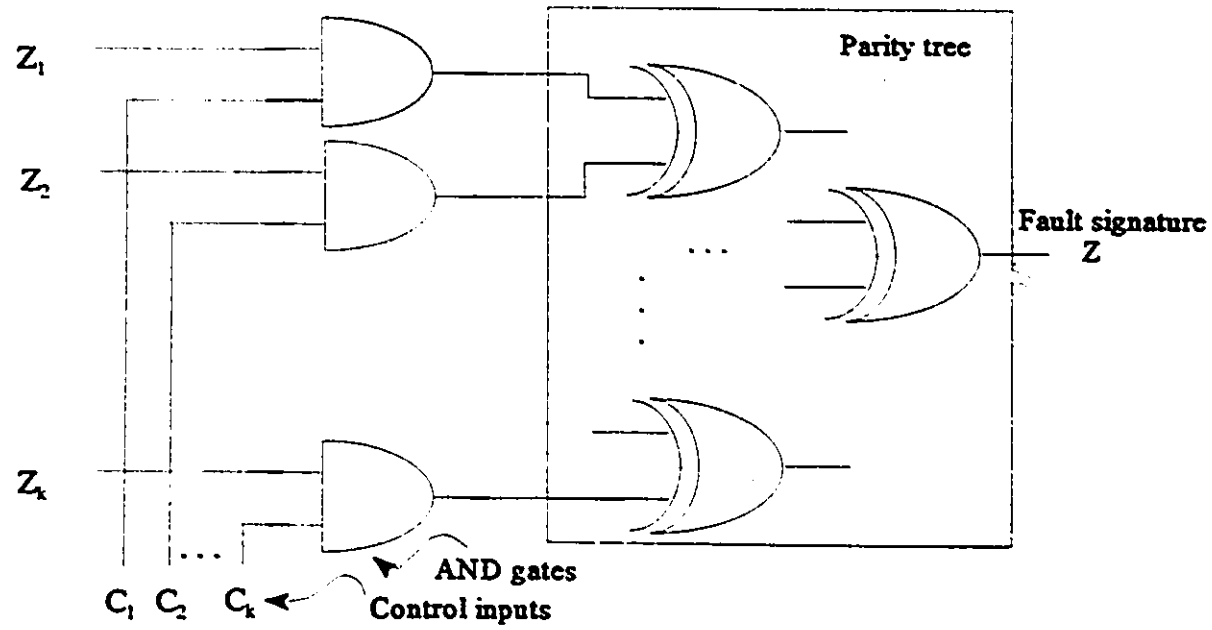


Figure 2.11 The MPT compaction scheme.

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.

2.3 Commercial VLSI Products Using BIST

Several companies such as AT&T, IBM, Motorola, and Intel have incorporated BIST in their products. BIST is widely used to test embedded regular structures that exhibit a high degree of periodicity such as memory arrays (SRAMs, ROMs, FIFOs, and Registers). This type of circuits does not require complex extra hardware for test generation and response compaction. Also including BIST in these circuits can guarantee high fault coverage with zero aliasing. Unlike regular circuits, random-logic circuits cannot be adequately tested with only BIST techniques, since generating adequate on-chip test sets using simple hardware is a difficult task to be accomplished. Moreover, since test responses generated by random-logic circuits seldom exhibit regularity, it is extremely difficult to ensure zero-aliasing compaction. Therefore, random-logic circuits are usually tested using a combination of BIST, scan design techniques, and external test equipments. For example, three large PLAs and the microcode ROM in the Intel 80386 microprocessor were built-in self tested [17].

The Alpha AXP21164, a general-purpose microprocessor chip, was tested using BIST [17]. The on-chip caches consist of SRAM arrays that contain 8 million transistors, pattern generators, and comparison logic were embedded to self-test the chip's instruction cache. However, the other caches on the chip were tested by loading self-test program into the instruction caches. The fault coverage using deterministic BIST is 100% and the area overhead is less than 15%.

Software tools that automatically insert BIST in VLSI designs have been developed at AT&T [4]. These tools were used to self-test a video processing chip. The fault coverage using pseudorandom BIST is over 96% at a 14% hardware area overhead.

Chapter 3

3. DESIGNING COMPACTION TREE FOR MULTI-OUTPUT CIRCUITS IN BIST BASED ON SEQUENCE CHARACTERIZATION AND STOCHASTIC INDEPENDENCE OF ERRORS

The principle idea of the proposed approach is to compress *m* functional test outputs of the CUT into one single test output line without sacrificing too much information. Figure 3.1 illustrates this approach.

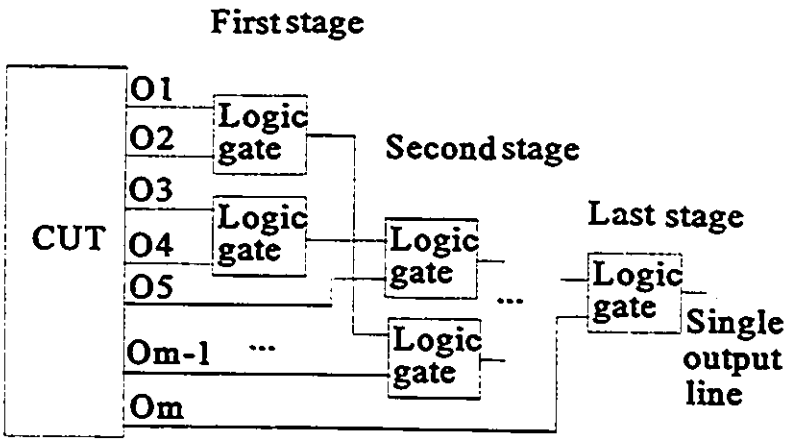


Figure 3.1 Sequence characterization compaction scheme.

Generally, space compression has been accomplished using XOR gates in cascade or in a tree structure. We will adopt a combination of both cascade and tree structures (cascade-tree) for our framework with AND (NAND), OR (NOR), and XOR (XNOR) operators.

The logic function to be selected to build the compaction tree will be determined by the characteristics of the sequences which are inputs to the gates based on some mergeability criteria. We also assume a syndrome counter [5] at the output of the two-input gates of the compaction tree as shown in Figure 3.2.

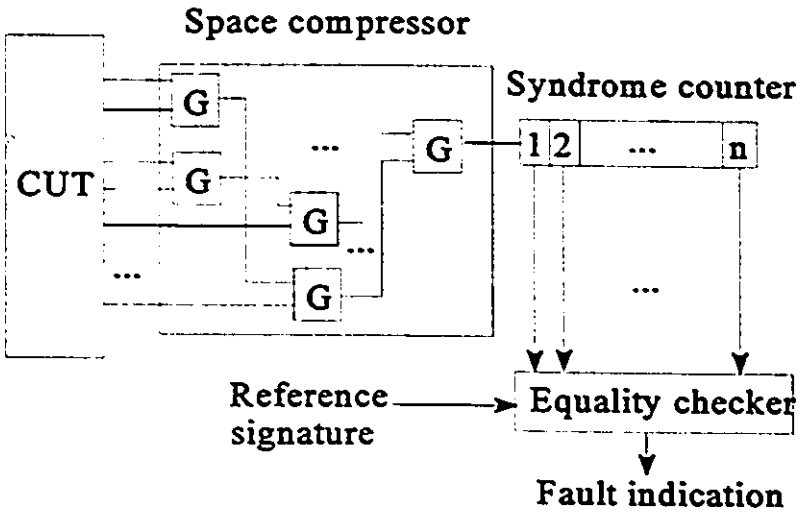


Figure 3.2 Syndrome counter is used as time compressor in test data output compaction.

The time compressor (i.e. Syndrome counter) derives the signature for the CUT, by counting the number of ones appearing at the output of the space compactor and the equality checker checks the counter contents with the expected correct signature. If the syndrome from the CUT, going through the space compactor, is different from the expected syndrome from the fault-free circuit, the CUT is said to be faulty. However, if the two syndromes are the same, the CUT may or may not be faulty. Therefore, an output data modification technique with the objective to turn any single stuck-at fault in the CUT into a faulty syndrome has to be examined. Consequently, the output data modification in [75] may be corporated to increase the error coverage of the single output CUT.

The basic theme of the proposed approach is to select a suitable gate to merge two candidate output lines of the CUT. The selection is essentially based on certain mergeability criteria that use the properties of Hamming distance and sequence weight [25].

3.1 Mathematical Basis for the Proposed Approach

3.1.1 Derived sequences

Let $\{\Psi_i, \Psi_j\}$ represent a pair of output sequences of length L_s, where the length is the number of bit positions in Ψ_i and Ψ_j. Let $H_s(\Psi_i, \Psi_j)$ represents the Hamming distance between Ψ_i and Ψ_j, i.e., the number of bit positions in which Ψ_i and Ψ_j differ.

Definition 1 The 1-weight, denoted by $W_1(\Psi_i)$, of a sequence Ψ_i, is the number of 1s in the sequence. Similarly, the 0-weight, denoted by $W_0(\Psi_i)$, of a sequence Ψ_i, is the number of 0s in the sequence.

Example 3.1 Consider an output sequence pair $\{\Psi_1, \Psi_2\}$ where :

$\Psi_1 = (00101010)$;

and

$\Psi_2 = (00001110)$.

The length of both output streams is 8 ($L_s = 8$). The Hamming distance between Ψ_1 and Ψ_2 is 2 ($H_s(\Psi_1, \Psi_2) = 2$).

The 1- and 0- weights of Ψ_1 and Ψ_2 are $W_1(\Psi_1) = 3$, $W_1(\Psi_2) = 3$, $W_0(\Psi_1) = 5$, and $W_0(\Psi_2) = 5$, respectively. Figure 3.3 illustrates how to calculate the above terms.

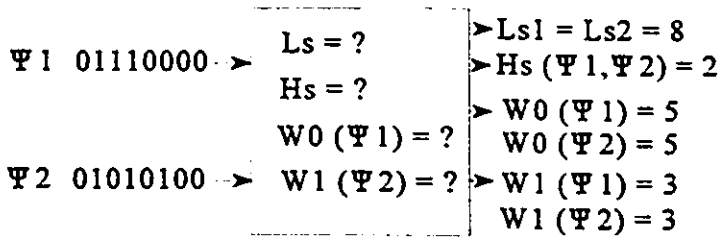


Figure 3.3 Calculation of the sequence characteristics.

Property 1 For any sequence Ψ_i, it can be shown that $L_s = W_0(\Psi_i) + W_1(\Psi_i)$.

For the output sequence Ψ_1 = (00101010) given in Example 3.1, we have found that $W_0(\Psi_1) = 5$ and $W_1(\Psi_1) = 3$. Therefore, it is obvious that $L_s = 5 + 3 = 8$.

Definition 2 Consider an output sequence pair $\{\Psi_i, \Psi_j\}$ of equal length L_s. Then the sequence pair derived by discarding the bit positions in which the two sequences differ (indicated by a dash -) is called its derived sequence pair, $\{\Psi'_i, \Psi'_j\}$.

In the rest of the dissertation, we will denote the derived sequence of a sequence Ψ_i by Ψ'_i, its 0-weight by $W'_0(\Psi'_i)$, and its 1-weight by $W'_1(\Psi'_i)$.

Example 3.2 For $\{\Psi_1, \Psi_2\}$ given in Example 3.1, $\{\Psi'_1, \Psi'_2\}$ = (00-01-10 , 00-01-10}. The 1- weight and 0-weight of the derived pair $\{\Psi'_1, \Psi'_2\}$ are, respectively: $W'_1(\Psi'_1) = 2$, $W'_1(\Psi'_2) = 2$, $W'_0(\Psi'_1) = 4$, and $W'_0(\Psi'_2) = 4$, as shown in Figure 3.4.

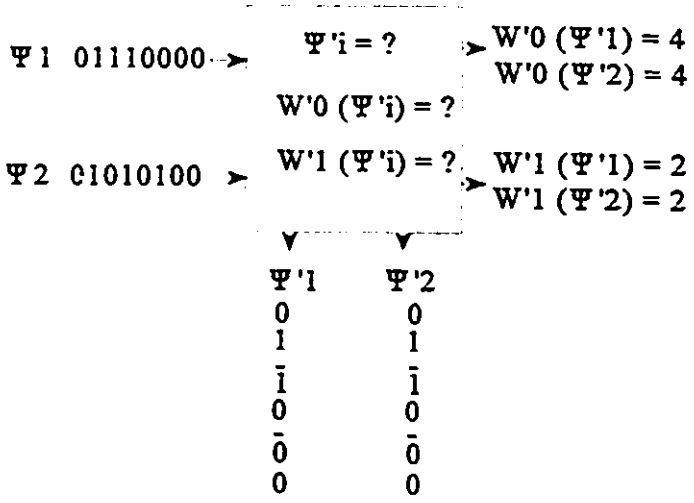


Figure 3.4 The derived sequence pair.

Now some key properties of the derived sequences will be described.

Property 2 For any derived sequence pair $\{\Psi'_i, \Psi'_j\}$, we have $W'_1(\Psi'_i) = W'_1(\Psi'_j)$ and $W'_0(\Psi'_i) = W'_0(\Psi'_j)$.

i.e. as shown in Example 3.2, for the same output sequence pair $\{\Psi_i, \Psi_j\}$ given in Example 3.1, we have shown that $W'_1(\Psi'_1) = W'_1(\Psi'_2) = 2$ and $W'_0(\Psi'_1) = W'_0(\Psi'_1) = 4$.

By the above property, when no ambiguity arises, we will denote 0- and 1- weight for the derived sequence pair by simply W'_0 and W'_1, respectively. The length of the derived sequence pair will be denoted by L'_s, where $L'_s = L_s - H_s(\Psi_i, \Psi_j)$. Also, since $H_s(\Psi'_i, \Psi'_j)$ is always zero, we will simply use H_s to denote $H_s(\Psi'_i, \Psi'_j)$.

That is, for $\{\Psi_i, \Psi_j\}$ in Example 3.1, $\Psi'_1 = (00\text{-}01\text{-}10)$, $\Psi'_2 = (00\text{-}01\text{-}10)$, $L_s = 8$ and $H_s(\Psi_1, \Psi_2) = 2 = H_s$. Therefore, $L'_s = 8 - 2 = 6$ and $H_s(\Psi'_1, \Psi'_2) = 0$.

Property 3 For every distinct pair of output sequences $\{\Psi_i, \Psi_j\}$ at the output of a CUT, the corresponding derived pair $\{\Psi'_i, \Psi'_j\}$ of equal length L'_s is distinct.

Two derived sequence pairs may have the same length L'_s but they are still distinct and not identical.

Property 4 Two derived sequence pairs $\{\Psi'_i, \Psi'_j\}$ and $\{\Phi'_i, \Phi'_j\}$ of original output streams pairs $\{\Psi_i, \Psi_j\}$ and $\{\Phi_i, \Phi_j\}$, respectively, having the same length L'_s are identical. Thereby, $\{\Psi'_i, \Psi'_j\} = \{\Phi'_i, \Phi'_j\}$, if and only if, $\{\Psi_i, \Psi_j\} = \{\Phi_i, \Phi_j\}$.
Consider $\Psi_1 = (\ 00001010\ )$, $\Psi_2 = (\ 00101110\ )$, $\Phi_1 = (\ 00001111\ )$, and $\Phi_2 = (\ 00000011\ )$ as four output sequence streams of a CUT. Let $\{\Psi_1, \Psi_2\}$ and $\{\Phi_1, \Phi_2\}$ be two distinct output pairs, and $\{\Psi'_1, \Psi'_2\}$, $\{\Phi'_1, \Phi'_2\}$ their corresponding derived sequence pairs, respectively, such that $\{\Psi'_1, \Psi'_2\}$ = {00-01-10, 00-01-10} and $\{\Phi'_1, \Phi'_2\}$ = {0000--11, 0000--11}.
Both derived sequence pairs have the same length $L'_s = 6$ but they are not identical.

However, in general, it is not expected that any two distinct pairs of sequences at the output of the CUT will be identical and hence the possibility of the corresponding derived pairs being identical is also remote.

3.2 Mergeability Criteria and Gate Selection

Definition 3 The maximum expectation of error (E_E) for two-input logic functions, given two input sequences of length L_s is defined as the sum of all single line errors ($2L_s$) and all double line errors (L_s).
For the output sequence pair $\{\Psi_1, \Psi_2\}$ given in Example 3.1, the maximum expectation of error (E_E) for two-input logic functions is $E_E = 2 \times 8 + 8 = 24$.

This definition assumes stochastic independence of single line and double line errors and thus includes all of the possibilities of error occurrence. We will define later the mergeability criteria assuming distinct probabilities for single and double line errors. Now let $E_s(g)$ be the number of single line errors and $E_d(g)$ be the number of double line errors detected at the output of a gate g.

Definition 4 The number of single and double line errors detected at the output of a two-input logic gate are defined in Table 3.1 as follows.

Gate type	Input 1	Input 2	Single line errors detection	Double line errors detection
AND/NAND	0	0	0	1
AND/NAND	0	1	1	0
AND/NAND	1	0	1	0
AND/NAND	1	1	2	1
Gate type	Input 1	Input 2	Single line errors detection	Double line errors detection
OR/NOR	0	0	2	1
OR/NOR	0	1	1	0
OR/NOR	1	0	1	0
OR/NOR	1	1	0	1
Gate type	Input 1	Input 2	Single line errors detection	Double line errors detection
XOR/XNOR	0	0	2	0
XOR/XNOR	0	1	2	0
XOR/XNOR	1	0	2	0
XOR/XNOR	1	1	2	0

Table 3.1 Error detection using different two-input logic gates.

These values are separately calculated for the case of AND/NAND, OR/NOR, and XOR/XNOR gates.

The logic behavior of the gate under investigation used to merge two candidate outputs determines the number of single and double line errors detected at the output of that gate. Two-input XOR/XNOR gates detect all single line errors while both suffer from not being able to detect any double line error.

Definition 5 The maximum detectable error ($E_{max}(g)$) for two-input logic functions, given an input sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s when the gate type is g, is defined as $E_{max}(g) = E_s(g) + E_d(g)$, as given in Table 3.2.

Gate type (g)	Maximum detectable error	Detectable error
AND/NAND	$E_{max}(AND/NAND) = 2L_s + L_s$	$E(AND/NAND) <= 2L_s + L_s$
OR/NOR	$E_{max}(OR/NOR) = 2L_s + L_s$	$E(OR/NOR) <= 2L_s + L_s$
XOR/XNOR	$E_{max}(XOR/XNOR) = 2L_s + 0$	$E(XOR/XNOR) = 2L_s + 0$

Table 3.2 The detectable and maximum detectable error using different types of gates.

$E_{max}(XOR/XNOR) = E_s(XOR/XNOR) + E_d(XOR/XNOR) = 2L_s + 0 = E(XOR/XNOR)$ where $E_s(XOR/XNOR)$ and $E_d(XOR/XNOR)$ are, respectively, the single and double line detectable errors when using XOR or XNOR gates.

Theorem 1 For any derived output sequence pair $\{\Psi'_1, \Psi'_2\}$ of length L'_s, if an AND (NAND) gate is used to merge the derived sequence pair, the total number of detected errors will be $3W'_1$, $2W'_1$ being single line errors and W'_1 double line errors (assuming $W'_0 = 0$).

For $\{\Psi_1, \Psi_2\}$ in Example 3.1, $W'_1(\Psi'_1) = 2$ and $W'_1(\Psi'_2) = 2$. Therefore, the total number of errors detected at the output of an AND (NAND) gate is 6 (all errors are detected). ■

Theorem 2 For any derived output sequence pair $\{\Psi'_i, \Psi'_j\}$ of length L'_s, if an OR (NOR) gate is used to merge the derived sequence pair, the total number of detected errors will be $3W'_0$, $2W'_0$ being single line errors and W'_0 double line errors (assuming $W'_1 = 0$). For $\{\Psi_1, \Psi_2\}$ in Example 3.1, $W'_0(\Psi'_1) = 4$ and $W'_0(\Psi'_2) = 4$. Therefore, the total number of errors detected at the output of an OR (NOR) gate is 12 (all errors are detected). ■

Theorem 3 If an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s is merged with an AND (NAND) gate, the maximum errors detected $E_{max}(AND/NAND)$ will be as follows:

$E_{max}(AND/NAND) = H_s$ *single line errors* $+ 3W'_1$ *single and double line errors* $+ W'_0$ *double line errors.*

Proof: Since the Hamming distance is H_s, if one sequence has a 0, the other has a 1 in the corresponding position for all the H_s bits. If an AND (NAND) gate is used, only single fault that makes both sequences 1 will be detectable, and thus H_s single line faults are detected.

Similarly, $3W'_1$ single line and double line faults are detected if W'_1 is the 1-weight of the derived sequence pair. Likewise, if W'_0 is the 0-weight of the derived sequence pair, only W'_0 double line faults making both sequences 1 in the corresponding bit positions will be detected. ■

Theorem 4 If an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s is merged using an OR (NOR) gate, the maximum errors detected $E_{max}(OR/NOR)$ will be as follows:

$E_{max}(OR/NOR) = H_s$ *single line errors* $+ 3W'_0$ *single and double line errors* $+ W'_1$ *double line errors.*

Proof: As before, since the Hamming distance is H_s, if one sequence has a 0 the other has a 1 in the corresponding position for all the H_s bits. If an OR (NOR) gate is used, only single fault that makes both sequences 0 will be detectable, and hence H_s single line faults are detected. Similarly, $3W_0$ single line and double line faults are detected if W_0 is the 0-weight of the derived sequence pair. Likewise, if W_1 is the 1-weight of the derived sequence pair, only W_1 double line faults making both sequences 0 in the corresponding bit positions will be detected.

■

Theorem 5 An output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s is XOR (XNOR) gate mergeable if the maximum number of errors detected is $2L_s$, which is greater than or equal to the maximum number of errors detected by AND (NAND) gates E_{max}*(AND/NAND)* or OR (NOR) gates E_{max}*(OR/NOR)*, when used for merger.

Proof: The theorem is a direct outcome of Theorems 4 and 5.

■

Theorem 6 For any output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s, an OR (NOR) gate or an AND (NAND) gate may be used for merger if the maximum number of errors detected is E_{max} where $2L_s \leq E_{max} \leq 3L_s$.

Proof: For sequences of length L_s, the maximum expectation of error E_E is $3L_s$ ($2L_s$ single line and L_s double line errors). An XOR (XNOR) gate can only detect $2L_s$ single line errors. Hence, if the error detection count is between $2L_s$ and $3L_s$, we need to use OR (NOR) or AND (NAND) gates based on whether E_{max}(AND/NAND) or E_{max}(OR/NOR) has a greater value.

■

Theorem 7 An AND (NAND) gate may be selected for merging an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s with Hamming distance H_s, if and only if, $W'_1 > W'_0$ and

$E_{max}(AND/NAND) = H_s$ *single line errors* $+ 3W'_1$ *single and double line errors*

$+ W'_0$ *double line errors*

$\geq 2L_s$.

Proof: Since $E_{max}(AND/NAND)$ is $H_s + 3W'_1 + W'_0$ and $E_{max}(OR/NOR)$ is $H_s + 3W'_0 + W'_1$, if AND/NAND is preferable to OR/NOR, then obviously

$E_{max}(AND/NAND) - E_{max}(OR/NOR) > 0$,

or $H_s + 3W'_1 + W'_0 - H_s + 3W'_0 + W'_1 > 0$,

or $W'_1 > W'_0$. Further, if $E_{max}(AND/NAND) \geq 2L_s$, total faults detected by XOR/XNOR gate, then evidently AND/NAND gate is selected for merger.

■

Theorem 8 An OR (NOR) gate may be selected for merger of an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s with Hamming distance H_s, if and only if, $W'_0 > W'_1$ and

$E_{max}(OR/NOR) = H_s$ *single line errors* $+ 3W'_0$ *single and double line errors*

$+ W'_1$ *double line errors*

$\geq 2L_s$.

Proof: Follows as in Theorem 7.

Corollary 8.1 However, if $W'_0 = W'_1$, either OR (NOR) or AND (NAND) gate may be used for merger.

Theorem 9 An output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s needs to be merged with an AND (NAND) gate , if and only if, $E_{max}(AND/NAND)$ is maximized over $m(m-1)/2$ sequence pairs for an m-output CUT.

Proof: Since $E_{max}(AND/NAND)$, considering all output sequence pairs for an m-output CUT, $(m(m-1)/2)$, has the greatest value, obviously AND/NAND gate must be selected.

■

Theorem 10 An output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s needs to be merged with an OR (NOR) gate if and only if $E_{max}(OR/NOR)$ is maximized over $m(m-1)/2$ sequence pairs for an m-output CUT.

Proof: If $E_{max}(OR/NOR)$ is maximized over all $m(m-1)/2$ output sequence pairs for an m-output CUT, then evidently OR/NOR gate should be selected since OR/NOR detects the largest number of faults.

■

Theorem 11 An output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s is AND (NAND) gate mergeable, if and only if, $W'_1 > L_s/2$.

Proof: $E_{max}(AND/NAND) = H_s + 3W'_1 + W'_0$ whereas $E_{max}(XOR/XNOR) = 2L_s$.

If AND/NAND is selected over XOR/XNOR, then

$E_{max}(AND/NAND) - E_{max}(XOR/XNOR) > 0$,

or $H_s + 3W'_1 + W'_0 - 2L_s > 0$,

or $(H_s + W'_1 + W'_0) + 2W'_1 - 2L_s > 0$,

or $L_s + 2W'_1 - 2L_s > 0$,

or $W'_1 > L_s / 2$.

Besides, if $W'_1 > L_s/2$, W'_1 is also greater than W'_0 and hence the selection of AND/NAND gate is unique.

■

Theorem 12 An output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s is OR (NOR) gate mergeable, if and only if, $W'_0 > L_s/2$.

Proof: The theorem follows in the same way as Theorem 11.

■

3.3 Algorithm 1

The algorithm for implementation of the proposed method is given below in a pseudocode format.

stage = total_number_of_outputs
while (current_stage < stage-1)
{
for (i = 0; i < total_number_of_outputs; i++)
for (j = 1; j < total_number_of_outputs, j ≠ i; j++)
{
Determine the derived sequence pair $\{\Psi'_i, \Psi'_j\}$ that corresponds to the sequence pair $\{\Psi_i, \Psi_j\}$.

Compute its corresponding Hamming distance H_s, 0-weight W'_0, and 1-weight W'_1.

Select a gate for merger based on the mergeability criteria discussed earlier (AND/NAND gate if $W'_1 > (L_s / 2)$; OR/NOR gate if $W'_0 > (L_s / 2)$; XOR/XNOR gate otherwise) and compute E_{max} value for the selected gate.
}
Select the sequence pair $\{\Psi_i, \Psi_j\}$ for merger that has the maximum E_{max} value, choosing one arbitrarily when there is a tie.

Compute the corresponding new output sequence using the chosen gate.

Discard sequence pair $\{\Psi_i, \Psi_j\}$ which will no longer be used as the result of this selection.
stage++
}

Example 3.3 In Figure 3.5 (a), a 4-output demultiplexer is shown with four different inputs (grouped input test patterns) X_1, X_2, X_3, and X_4. The fault-free output patterns O_1, O_2, O_3, and O_4 are given in Table 3.3 .

There are only six output sequence pairs $\{O_i, O_j\}$ $i = 1,...,6; j = 1,...,6; i \neq j$ to consider in the first stage. The Hamming distances of all sequence pairs are computed and given below.

$H_s(O_1, O_2) = H_s(O_1, O_3) = H_s(O_1, O_4) = H_s(O_2, O_3) = H_s(O_2, O_4) = H_s(O_3, O_4) = 2$. Furthermore, W'_0, the 0-weight of the derived sequences corresponding to every sequence pair is 14 which is greater than $L_s / 2$ ($L_s = 16$). Therefore, every pair is OR (NOR) gate mergeable.

X_1	X_2	X_3	X_4	O_1	O_2	O_3	O_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

Table 3.3 Fault-free output patterns for the 4-output demultiplexer.

An OR gate is selected to merge O_1 and O_2 chosen arbitrarily. The remaining sequences O_3 and O_4 are also merged with an OR gate. The two output sequences in the subsequent stage turn out to be also OR (NOR) mergeable: so they are merged with an OR gate to complete the compression tree. The circuit with the compression tree is shown in Figure 3.5 (b).

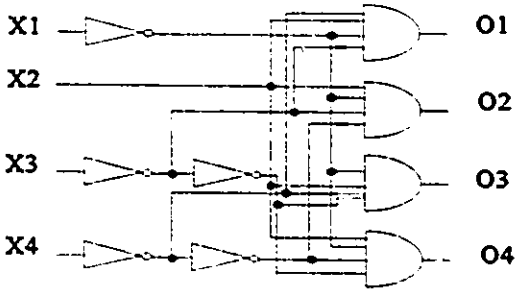


Figure 3.5 (a) The 4-output demultiplexer circuit.

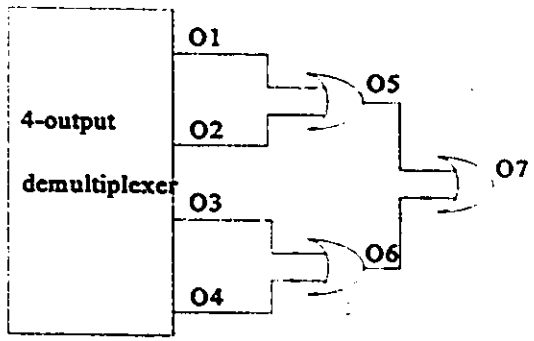


Figure 3.5 (b) The circuit with its corresponding compaction tree.

Example 3.4 Consider the Ten-line decimal-to-8421 BCD converter shown in Figure 3.6 (a). The circuit has got 9 primary inputs X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, and X_9, and four primary outputs O_1, O_2, O_3, and O_4. For the ten grouped input test patterns, the fault-free output patterns are shown below in Table 3.4 .

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	O_1	O_2	O_3	O_4
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	0	0	1	1	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	1	1	0	0	1

Table 3.4 Fault-free output patterns for the Ten-line decimal-to-8421 BCD converter.

There are only six output sequence pairs $\{O_i, O_j\}$ $i = 1,...,6; j = 1,...,6; i \neq j$ to consider in the first stage. The Hamming distances of all sequence pairs are computed and given below.

$H_s(O_1, O_2) = H_s(O_1, O_3) = H_s(O_1, O_4) = 5;$

$H_s(O_2, O_3) = 4;$

$H_s(O_2, O_4) = H_s(O_3, O_4) = 6.$

Furthermore, W'_0 and W'_1, the 0-weight and 1-weight, respectively, of the derived sequences corresponding to every sequence pair are as follows:

$W'_0(O_1, O_2) = W'_0(O_1, O_3) = 3;\ W'_1(O_1, O_2) = W'_1(O_1, O_3) = W''_1(O_2, O_3) = 2;$

$W''_0(O_1, O_4) = W'_0(O_2, O_3) = W'_0(O_2, O_4) = W'_0(O_3, O_4) = 4;\ W'_1(O_1, O_4) = 1;$

$W'_1(O_2, O_4) = W'_1(O_3, O_4) = 0.$

Since $E_{max}(XOR/XNOR) = 2L_s = 20$ ($L_s = 10$) is greater than $E_{max}(AND/NAND)$ and $E_{max}(OR/NOR)$ for every pair of sequences, every pair is XOR (XNOR) gate mergeable.

An XOR gate is selected to merge O_1 and O_2 chosen arbitrarily. The remaining sequences O_3 and O_4 are also merged with an XOR gate.

The two output sequences in the subsequent stage turn out to be also XOR (XNOR) gate mergeable: so they are merged with an XOR gate to complete the compression tree. The circuit with the compression tree is shown in Figure 3.6 (b).

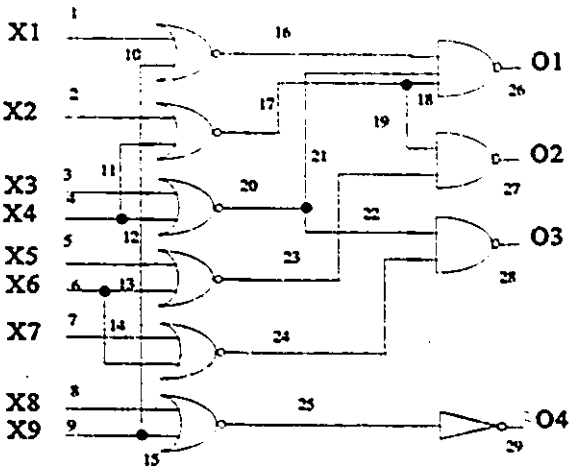


Figure 3.6 (a) The Ten-line decimal-to-8421 BCD converter.

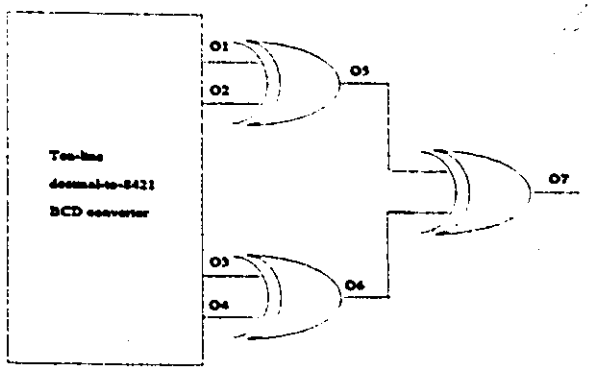


Figure 3.6 (b) The circuit with its corresponding compaction tree.

The Ten-line decimal-to-8421 BCD converter has got 58 single stuck-line faults, 2 undetectable faults at ${}_{11}{}^{0}O_1$ and ${}_{12}{}^{0}O_1$ (the effect of lines 11 and 12 being stuck-at-0 is not visible at output O_1), 34 equivalent faults, and 22 single stuck-line faults in its collapsed fault set. All detectable single stuck-at faults are injected into the circuit. The fault effect for this circuit at the outputs with the test set of Table 3.4 is shown in Table 3.5.

The fault-free space compactor response for the 10 input test vectors is the following single 10-bit stream $O_7 = (0\,1\,0\,0\,1\,1\,0\,1\,1\,0)$.

If our space compactor is followed by a time compactor (i.e. Syndrome Counter), the resulting fault-free signature for the output stream O_7 is simply the number of ones which will be equal to 5 in this case. This fault-free signature is saved in memory to be compared to the faulty signatures in order to determine the percentage of loss.

Fault effect	${}_1{}^0O_1$	${}_1{}^1O_1$	${}_2{}^0O_1$	${}_2{}^0O_2$	${}_2{}^1O_2$	${}_3{}^0O_1$	${}_3{}^0O_3$	${}_3{}^1O_3$	${}_4{}^0O_1$	${}_4{}^0O_2$	${}_4{}^0O_3$	${}_5{}^0O_2$	${}_6{}^0O_2$	${}_6{}^0O_3$	${}_7{}^0O_3$	${}_8{}^0O_4$	${}_8{}^1O_4$	${}_9{}^0O_1$
For the	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
same	0	1	1	0	1	1	0	1	1	0	0	0	0	0	0	0	1	1
grouped	1	1	0	0	1	1	0	1	1	1	0	1	1	0	0	0	1	1
input test	1	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1
patterns	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	1	1
used to	0	1	0	1	1	0	0	1	0	1	0	0	1	0	0	0	1	0
calculate	0	1	0	1	1	0	1	1	0	1	1	1	0	0	1	0	1	0
the	0	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	1	0
fault-free	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
responses	1	1	1	0	1	1	0	1	1	0	0	0	0	0	0	1	1	0

Fault effect	${}_9{}^0O_4$	${}_{16}{}^1O_1$	${}_{17}{}^1O_1$	${}_{17}{}^1O_2$	${}_{20}{}^1O_1$	${}_{20}{}^1O_3$	${}_{23}{}^1O_2$	${}_{24}{}^1O_3$	${}_{25}{}^1O_4$	${}_{26}{}^0O_1$	${}_{27}{}^0O_2$	${}_{28}{}^0O_3$
For the	0	0	0	0	0	0	0	0	0	0	0	0
same	0	0	1	0	1	0	0	0	0	0	0	0
grouped	0	1	0	0	1	0	1	0	0	0	0	0
input test	0	1	1	0	0	0	0	1	0	0	0	0
patterns	0	1	0	0	0	0	1	1	0	0	0	0
used to	0	0	0	1	0	0	0	0	0	0	0	0
calculate	0	0	0	1	0	1	0	0	0	0	0	0
the	0	0	0	0	0	1	0	0	0	0	0	0
fault-free	1	0	0	0	0	0	0	0	0	0	0	0
responses	0	0	1	0	1	0	0	0	0	0	0	0

Table 3.5 The fault effect at the outputs of the Ten-line decimal-to-8421 BCD converter.

The faulty signatures corresponding to the single line faults ${}_1{}^0O_1$, ${}_1{}^1O_1$, ... , ${}_{27}{}^0O_2$, ${}_{28}{}^0O_3$ are, respectively, the following: 4, 10, ... , 0,0. Comparing all the faulty signatures to the expected signature, we notice that two of the faulty signatures match the fault-free signature. Therefore, 56 detectable faults are injected into the circuit and the percentage of loss before compaction is 0.00 %. Furthermore, we calculate percentage of fault coverage at the output of the space compactor by comparing the fault-free stream bits to the faulty stream bits at the output of the compactor. The number of undetected faults turns out to be 5 at the output of the space compactor. Hence the percentage of fault coverage at the output of the compressor equals to 91.07 % which is equivalent to 8.92 % fault loss.

Chapter 4

4. DESIGNING COMPACTION TREE FOR MULTI-OUTPUT CIRCUITS ASSUMING STOCHASTIC DEPENDENCE OF SINGLE AND DOUBLE LINE ERRORS

In the previous chapter, we presented the new compaction technique assuming stochastic independence of single line and double line errors. In the absence of stochastic independence, we observe that the probability plays an important role in the selection of gates for merger. In the following we deal with the construction of the compaction tree when single and double line errors are considered stochastically dependent.

4.1 Effect of Error Probability in Selection Criteria

Li and Robinson [47] determine the detectable error probability estimate ε for a two-input logic function, given two input sequences of length L_s as follows:

$$\varepsilon = S_1\left(\frac{R_1}{2L_s}\right) + S_2\left(\frac{R_2}{L_s}\right)$$

where

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

S_2: probability of double error effect felt at the output of the CUT;

R_1: number of single line errors at the output of gate i, if gate i is used;

R_2: number of double line errors at the output of gate i, if gate i is used.

Based on the detectable error probability estimate ε of Li and Robinson as given above, we deduce the following results that profoundly influence the selection of gates for merger.

Theorem 13 For an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s, an AND/NAND gate is preferable to an XOR(XNOR) gate if

$$SI<\frac{[2W'_1+2W'_0]}{[H_s+4W'_0+2W'_1]} \quad .$$

Proof: The detectable error probability estimate when an AND/NAND gate is used is:

$$\varepsilon(A/N)=S_1\frac{[H_s+2W'_1]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad ,$$

while the corresponding one when an XOR/XNOR gate is used is:

$$\varepsilon(X/X)=S_1\frac{[2L_s]}{2L_s}=S_1 \quad .$$

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

$\varepsilon(A/N) > \varepsilon(X/X)$, or,

$\varepsilon(A/N) - \varepsilon(X/X) > 0$, or,

$$S_1\frac{[H_s+2W'_1]}{2L_s}+\left[S_2\frac{[W'_0+W'_1]}{L_s}-S_1\right]>0 \quad , \text{or,}$$

$$S_1\frac{[H_s+2W'_1]}{2L_s}+\left[\frac{[1-S_1][W'_0+W'_1]}{L_s}-S_1\right]>0 \quad , \text{or,}$$

$$S_1\frac{[H_s+2W'_1-2L_s]}{2L_s}+\left[\frac{[1-S_1][W'_0+W'_1]}{L_s}\right]>0 \quad \text{, or,}$$

$$S_1\frac{[H_s+2W'_1-2L_s-2W'_0-2W'_1]}{2L_s}+\left[\frac{[W'_0+W'_1]}{L_s}\right]>0 \quad \text{, or,}$$

$$S_1\frac{[-H_s-4W'_0-2W'_1]}{2L_s}>\frac{-[W'_0+W'_1]}{L_s} \quad \text{, or,}$$

$$-S_1[H_s+4W'_0+2W'_1]>-2[W'_0+W'_1] \quad \text{, or,}$$

$$S_1<\frac{2[W'_0+W'_1]}{[H_s+4W'_0+2W'_1]} \quad .$$

Corollary 13.1 On the other hand, an XOR/XNOR gate is selected if

$$SI>\frac{[2W'_1+2W'_0]}{[H_s+4W'_0+2W'_1]} \quad .$$

Theorem 14 For an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s, an OR (NOR) gate is preferable to an XOR/XNOR gate if

$$SI<\frac{[2W'_1+2W'_0]}{[H_s+4W'_1+2W'_0]} \quad .$$

Proof: The detectable error probability estimate when an OR/NOR gate is used is:

$$\varepsilon(O/N)=S_1\frac{[H_s+2W'_0]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad .$$

while the corresponding one when an XOR/XNOR gate is used is:

$$\varepsilon(X/X)=S_1\frac{[2L_s]}{2L_s}=S_1 \quad .$$

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

$\varepsilon(O/N) > \varepsilon(X/X)$, or,

$\varepsilon(O/N) - \varepsilon(X/X) > 0$, or,

$$S_1\frac{[H_s+2W'_0]}{2L_s}+\left[S_2\frac{[W'_0+W'_1]}{L_s}-S_1\right]>0 \quad , \text{or,}$$

$$S_1\frac{[H_s+2W'_0]}{2L_s}+\left[\frac{[1-S_1][W'_0+W'_1]}{L_s}-S_1\right]>0 \quad , \text{or,}$$

$$S_1\frac{[H_s+2W'_0-2L_s]}{2L_s}+\left[\frac{[1-S_1][W'_0+W'_1]}{L_s}\right]>0 \quad \text{. or.}$$

$$S_1\frac{[H_s+2W'_0-2L_s-2W'_0-2W'_1]}{2L_s}+\left[\frac{[W'_0+W'_1]}{L_s}\right]>0 \quad \text{. or.}$$

$$S_1\frac{[-H_s-4W'_1-2W'_0]}{2L_s}>\frac{-[W'_0+W'_1]}{L_s} \quad \text{. or.}$$

$$-S_1[H_s+4W'_1+2W'_0]>-2[W'_0+W'_1] \quad \text{. or.}$$

$$S_1<\frac{2[W'_0+W'_1]}{[H_s+4W'_1+2W'_0]} \quad .$$

■

Corollary 14.1 On the other hand, an XOR/XNOR gate is selected if

$$S_1>\frac{2[W'_0+W'_1]}{[H_s+4W'_1+2W'_0]} \quad .$$

However, the probability does not play any role in the selection between AND (NAND) and OR (NOR) gates if we use the empirical formula of Li and Robinson. The undernoted theorem states the condition that determines the selection criteria between AND/NAND and OR/NOR gates.

Theorem 15 For an output sequence pair $\{\Psi_i, \Psi_j\}$ of length L_s and Hamming distance H_s, an AND (NAND) gate is preferable to an OR (NOR) gate if $W'_1 > W'_0$.

Proof: The detectable error probability estimate when an AND/NAND gate is used is:

$$\varepsilon(A/N)=S_1\frac{[H_s+2W'_1]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad .$$

and the corresponding one when an OR/NOR gate is used is:

$$\varepsilon(O/N)=S_1\frac{[H_s+2W'_0]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad .$$

An AND/NAND gate is surely preferable to an OR/NOR gate if:

ε(AND/NAND) - ε(OR/NOR) > 0 , or,

$$S_1\frac{[H_s+2W'_1-H_s-2W'_0]}{2L_s}+\frac{S_2[W'_0+W'_1-W'_0-W'_1]}{L_s}>0 \quad , \text{or,}$$

$$S_1\frac{[W'_1-W'_0]}{L_s}>0 \quad , \text{or,}$$

$$W'_1>W'_0 \quad .$$

■

Corollary 15.1 An OR/NOR gate is selected if, on the other hand,

$$W'_1<W'_0 \quad .$$

Theorem 16 Sequence merger following Li and Robinson criterion may result in a compression network that is different from the one that will result based on the selection criteria given above even if the same probability estimate of Li and Robinson is used as the guide to selection.

Proof: The proof is provided by the following example.

Consider the following three sequences Ψ_1 = (0000001000000000), Ψ_2 = (0000000100000000), and Ψ_3 = (0000110000000000). By Li and Robinson criterion, we will merge Ψ_1 and Ψ_3, while according to our selection criteria, Ψ_1 and Ψ_2 will be merged with an OR (NOR) gate because E_{max} *(OR/NOR)* for $\{\Psi_2, \Psi_3\}$ equals 0.81, E_{max} *(OR/NOR)* for $\{\Psi_1, \Psi_3\}$ also equals 0.81, whereas E_{max} *(OR/NOR)* for $\{\Psi_1, \Psi_2\}$ equals 0.87 which is obviously the greatest probability estimate, assuming $S_1 = 0.0$ and $S_2 = 1.0$.

■

The next theorem establishes that the gate selection based on the aforementioned results provide a stricter guideline as opposed to the criteria used by Li and Robinson.

Theorem 17 The use of the detectable error probability estimate ε as suggested by Li and Robinson and discussed earlier gives a weaker condition in gate selection for output merger in the construction of the compaction network compared to the selection criteria proposed by Theorems 13 through 16.

The proof once again is provided by the following counterexample.

Proof: Consider the following four output sequences:

W: 0000100000000000

X: 0000010000000000

Y: 0000001000000000

Z: 0000000100000000

All sequence pairs {W , X}, {W , Y}, {W , Z}, {X , Y}, {X , Z}, and {Y , Z} have the same characteristics: $H_s = 2$, $W'_0 = 14$, and $W'_1 = 0$

For $S_1 = 0.0$ and $S_2 = 1.0$, the corresponding probability estimates of all sequence pairs are the following:

ε(AND/NAND) = 0.87

ε(OR/NOR) = 0.87

ε(XOR/XNOR) = 0.0

Since sequence pairs have the same probability estimate, any one can be selected for merger (i.e. {W , X}).

Now, if we use Li and Robinson formula to merge the selected output sequences, then we will have to decide between choosing AND or OR gate (both have got the same probability estimate). But, according to our mergeability criteria, an OR gate should be used for merger because $W'_0 (W , X) = 14$ is greater than $W'_1 (W , X) = 0$. This shows that our condition in gate selection is stronger than Li and Robinson selection criteria for output merger in the construction of the compaction network.

■

We now prove a very important theorem concerning gate selection for merger with certain particular but generally chosen values of probabilities S_1 and S_2 for single and double line errors, respectively.

Theorem 18 The gate selection criteria established earlier in Chapter 3 under condition of stochastic independence of single line and double line errors at the output of a CUT remains unchanged even under condition of stochastic dependence of single line and double line errors based on computation of detectable error probability estimate ε, the empirical formula of Li and Robinson, if the values of the probabilities S_1 and S_2 are chosen as $S_1 = 2/3$ and $S_2 = 1/3$.

Proof: The detectable error probability estimates, as used earlier, for AND/NAND, OR/NOR and XOR/XNOR gates are, respectively,

$$\varepsilon(A/N)=S_1\frac{[H_s+2W'_1]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad ;$$

$$\varepsilon(O/N)=S_1\frac{[H_s+2W'_0]}{2L_s}+S_2\frac{[W'_0+W'_1]}{L_s} \quad ;$$

$$\varepsilon(X/X)=S_1\frac{[2L_s]}{2L_s}=S_1 \quad ;$$

for output sequences of length L_s, Hamming distance H_s, 1-weight W'_1 and 0-weight W'_0, with probability of single line and double line errors being, respectively, S_1 and S_2 as usual.

Evidently, an AND/NAND gate is preferable to XOR/XNOR gate, if and only if, again

$\varepsilon(A/N) - \varepsilon(X/X) > 0$,

and similarly, an OR/NOR gate is preferable to an XOR/XNOR, if

$\varepsilon(O/N) - \varepsilon(X/X) > 0$.

With values of S_1 and S_2 being, respectively, 2/3 and 1/3, we have:

$$\varepsilon(A/N)=\frac{[H_s+3W'_1+W'_0]}{3L_s} \quad ,$$

$$\varepsilon(O/N)=\frac{[H_s+3W'_0+W'_1]}{3L_s} \quad , \text{while}$$

$$\varepsilon(X/X)=\frac{2}{3} \quad .$$

Then

$$\varepsilon(A/N)-\varepsilon(X/X)=\frac{[H_s+3W'_1+W'_0]}{3L_s}-\frac{2}{3}>0 \quad , \text{or,}$$

$$[H_s+3W'_1+W'_0]-2L_s>0 \quad , \text{or,}$$

$$[H_s+3W'_1+W'_0-2H_s-2W'_1-2W'_0]>0 \quad , \text{or,}$$

$$[W'_1-W'_0-H_s]>0 \quad , \text{or,}$$

$$[W'_1-W'_0-L_s+W'_1+W'_0]>0 \quad , \text{or,}$$

$$2W'_1 > L_s \quad , \text{ or,}$$

$$W'_1 > \frac{L_s}{2} \quad .$$

This is exactly the same as the one established previously for AND/NAND gate selection under condition of stochastic independence of errors. It can be shown likewise that an OR/NOR gate is preferable to XOR/XNOR gate, if

$$W'_0 > \frac{L_s}{2}$$

and AND/NAND gate is preferable to OR/NOR, if

$$W'_1 > W'_0 \quad .$$

■

We thus see that our gate selection criteria need not to be changed even under condition of stochastic dependence of errors when their probabilities are $S_1 = 2/3$ and $S_1 = 1/3$, which considerably simplifies the computational problem involved with detectable error probability estimate ε.

4.2 Other Aspects of the Space Compactor

The space compactor produces a single signature for multiple output circuits rather than an individual signature for each output. As in [24-40], sequences coming out of the space compactor are stored in a time compactor as a signature. The compression is done by counting the number of ones in the output sequence. This one's counting scheme has a nonuniform deception volume where the deception volume is a function of weight. Generally, when two or more output functions are compressed, error masking can occur more easily. Furthermore, it would be desirable to have the storage problem associated with a large number of test patterns and the circuit size to be eliminated. An approach that seems to fit our compression model is one that uses the syndrome counter as a time compactor. The syndrome counter is used to reduce test output data by only counting the number of ones. Besides, this is independent of the order of test patterns.

The syndrome of a circuit is defined as the number of times the output has logic value one for the test sequence [62] . For exhaustive testing, the syndrome is the function weight or the number of minterms. Error patterns which are missed have the same number of ones changed to zeros as zeros changed to ones. Thus, only special patterns with an even number of bit errors are missed. Faults which only expand or only contract the function are always detected by syndrome compression [61].

Moreover, to increase the error coverage, the concept of modification of the output data before compaction [5] can be used in the design. The basic idea of output data modification is to reduce the deception volume of output data after compression, or to decrease the probability of error masking in single output CUT. This modification, indeed, modifies the output sequence before compressing it into the number of ones.

4.3 Algorithm 2

The pseudocode description of the algorithm that constructs the compaction tree assuming different probability values for single and double line errors is given below:

Define the probability of single error (S_1) and double error (S_2) effect felt at the output of the CUT.
stage = total_number_of_outputs
while (current_stage < stage-1)
{
for (i = 0; i < total_number_of_outputs; i++)
for (j = 1; j < total_number_of_outputs, j ≠ i; j++)
{
Determine the derived sequence pair $\{\Psi'_i, \Psi'_j\}$ that corresponds to the sequence pair $\{\Psi_i, \Psi_j\}$.
Compute its corresponding Hamming distance H_s, 0-weight W'_0, and 1-weight W'_1.
Compute its corresponding number of single line (R_1) and double line (R_2) errors at the output of the three basic type of gates (AND/NAND, OR/NOR,and XOR/XNOR).
Select the sequence pair $\{\Psi_i, \Psi_j\}$ for merger based on the detectable error probability estimate formula $\varepsilon = S_1(R_1/2L_s) + S_2(R_2/L_s)$ discussed earlier, choosing one arbitrarily when there is a tie.
}
Select a gate to merge the chosen sequence pair $\{\Psi_i, \Psi_j\}$ based on the mergeability criteria: an XOR/XNOR gate is preferable to an AND/NAND gate if $S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_0 + 2W'_1]$,
an XOR/XNOR gate is preferable to an OR/NOR gate if $S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_1 + 2W'_0]$, while
an AND/NAND gate is preferable to an OR/NOR gate if $W'_1 > W'_0$.
Compute the corresponding new output sequence using the chosen gate.
Discard sequence pairs $\{\Psi_i, \Psi_j\}$ which will no longer be used as a result of this selection.
stage++
}

4.4 Input Sequence Length

Apparently, the length of the test input patterns L applied to the CUT plays an important role in the construction of the circuit's compaction tree. The nature of its elements (line selection and type of logic gates used) changes simultaneously with the modification of the input test length L. Furthermore, as will be shown in the results section, the input test length has a significant effect on the percentage of faults coverage for the circuit.

Example 4.1 Consider the following three sequences: Ψ_1 = (111111000), Ψ_2 = (010101000), and Ψ_3 = (000100010). All sequences are of length L = 9.
We have shown that according to our selection criteria, Ψ_2 and Ψ_3 will be used for merger. If we now consider the same sequences, each increased by 6 bits of 1 (L = 15), we have the new sequences as follows: Ψ_1 = (111111000111111), Ψ_2 = (010101000111111), and Ψ_3 = (000100010111111); and now Ψ_1 and Ψ_2 will be selected for merger (assuming S_1 = 0.666 and S_2 = 0.333).

Next, we will apply the proposed scheme on the design of compaction trees for the 4-output demultiplexer and the Decimal-to-8421 BCD converter.

Example 4.2 Consider again the 4-output demultiplexer shown in Figure 4.1 (a).
Six output stream pairs are to be analyzed $\{O_i, O_j\}$ $i = 1,....,6; j = 1,....,6; i \neq j$ in the first stage. The Hamming distance is the same and equals to 2 for each pair: $H_s(O_1, O_2) = H_s(O_1, O_3) = H_s(O_1, O_4) = H_s(O_2, O_3) = H_s(O_2, O_4) = H_s(O_3, O_4) = 2$. The 0-weight is also the same for all pairs: $W'_0 = 14$, while the 1-weight $W'_1 = 0$.

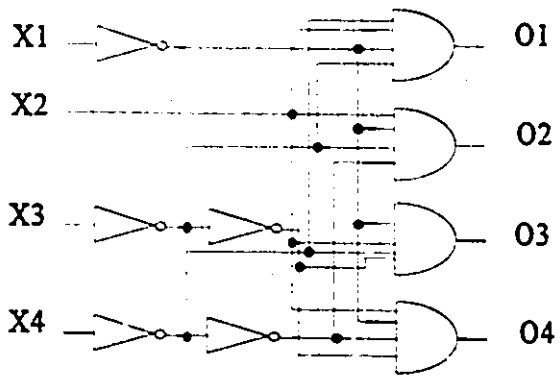


Figure 4.1 (a) The 4-output demultiplexer.

Since all sequence pairs have the same H_s, W'_0, and W'_1, therefore we have the same values for R_1 (AND/NAND) = 2, R_2 (AND/NAND) = 14, R_1 (OR/NOR) = 30, R_2 (OR/NOR) = 14, R_1 (XOR/XNOR) = 32, and R_2 (XOR/XNOR) = 0 which give the same probability estimates ε (AND/NAND) = 0.87, ε (OR/NOR) = 0.87, and ε (XOR/XNOR) = 0.0 . We assume here that S_1 = 0.0 and S_2= 1.0 which means we are considering the case where only the effect of double error is felt at the output of the CUT. Any sequence pair can be selected for merger in the first stage. O_1 and O_2 are arbitrarily chosen for merger with an OR gate since both mergeability equations

$S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_1 + 2W'_0]$

and

$S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_0 + 2W'_1]$

are not satisfied, and $(W'_1 = 0) < (W'_0 = 14)$.

The two output sequences O_3 and O_4 in the subsequent stage turn out to be also OR/NOR gate mergeable, and merged with an OR gate. Also in the last stage, O_5 and O_6 are merged with an OR gate to complete the compression tree.

The circuit with the compression tree is shown in Figure 4.1 (b).

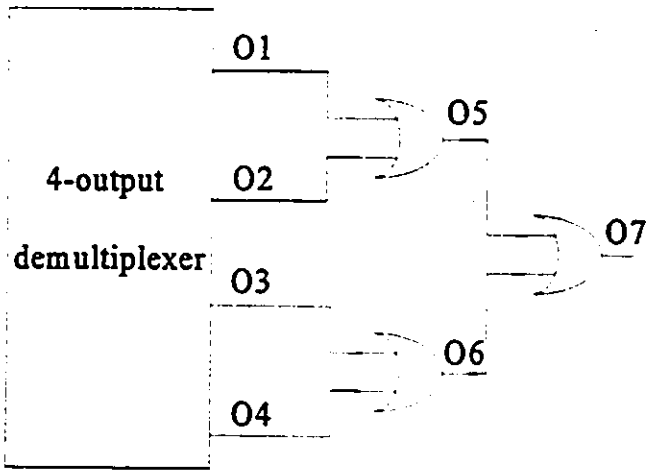


Figure 4.1 (b) The circuit with its corresponding compaction tree.

Now, assuming $S_1 = 1.0$ and $S_2 = 0.0$, we are considering the opposite case where only the effect of single error is felt at the output of the CUT. A different compaction tree is constructed. The output sequence selection turns out to be the same as in the previous case, but all pairs are XOR/XNOR gate mergeable.

Table 4.1 summarizes the results on the construction of compaction trees for different values of the probabilities of single and double error effects felt at the output of the CUT.

S_1	S_2	COMPACTION TREE
0.0	1.0	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = OR(O_5 , O_6)$
1.0	0.0	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.66	0.33	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = OR(O_5 , O_6)$
0.33	0.66	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = OR(O_5 , O_6)$
0.1	0.9	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = OR(O_5 , O_6)$
0.9	0.1	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.95	0.05	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.05	0.95	$O_5 = OR(O_1 , O_2)$ $O_6 = OR(O_3 , O_4)$ $O_7 = OR(O_5 , O_6)$

Table 4.1 The 4-output demultiplexer's compaction trees for different values of S_1 and S_2 .

Example 4.4 Once again, consider the Ten-line decimal-to-8421 BCD converter shown in Figure 4.2 (a). The circuit has got six sequence pairs $\{O_i, O_j\}$, $i = 1,....,6$; $j = 1,....,6$; $i \neq j$ to be considered in the first stage. The Hamming distances of all sequence pairs are calculated as shown below.

$H_s(O_1, O_2) = H_s(O_1, O_3) = H_s(O_1, O_4) = 5;$

$H_s(O_2, O_3) = 4;$

$H_s(O_2, O_4) = H_s(O_3, O_4) = 6.$

The 0-weight and 1-weight of the derived sequence pairs are:

$W'_0(O_1, O_2) = W'_0(O_1, O_3) = 3;\ W'_1(O_1, O_2) = W'_1(O_1, O_3) = W'_1(O_2, O_3) = 2;$

$W'_0(O_1, O_4) = W'_0(O_2, O_3) = W'_0(O_2, O_4) = W'_0(O_3, O_4) = 4;$

$W'_1(O_1, O_4) = 1;\ W'_1(O_2, O_4) = W'_1(O_3, O_4) = 0.$

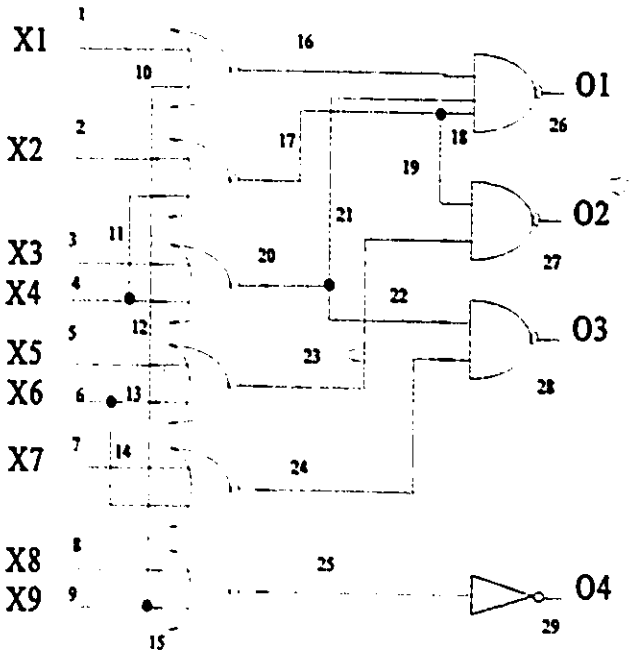


Figure 4.2 (a) The Ten-line decimal-to-8421 BCD converter.

Each pair of sequences has got different number of single line (R_1) and double line (R_2) errors at the output of gate i, if gate i is used for merger. The pair $\{O_2, O_3\}$ has got R_1 (AND/NAND) = 8, R_2 (AND/NAND) = 6, R_1 (OR/NOR) = 12, R_2 (OR/NOR) = 6, R_1 (XOR/XNOR) = 20, and R_2 (XOR/XNOR) = 0.

The corresponding probability estimate ε (OR/NOR) = 0.6 has the highest value (assuming $S_1 = 0.05$ and $S_2 = 0.95$). Therefore, O_2 and O_3 are selected for merger in the first stage. They are merged with an OR gate because both mergeability equations

$S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_1 + 2W'_0]$

and

$S_1 > [2W'_1 + 2W'_0] / [H_s + 4W'_1 + 2W'_0]$

are not satisfied, and $(W'_1 = 2) < (W'_0 = 4)$.

The two output sequences O_1 and O_4 in the subsequent stage turn out to be also OR/NOR gate mergeable (and merged with an OR gate), while in the last stage, O_5 and O_6 are merged with an AND gate, completing the compaction tree.

The circuit with the compression tree is shown in Figure 4.2 (b).

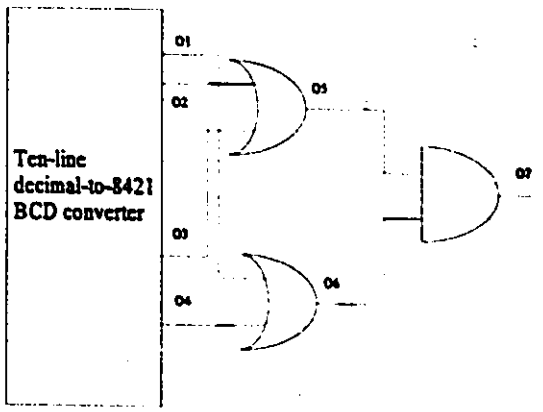


Figure 4.2 (b) The circuit with its corresponding compaction tree.

Table 4.2 summarizes the results on the construction of compaction trees for different values of the probabilities of single and double error effects felt at the output of the CUT.

S_1	S_2	COMPACTION TREE
0.0	1.0	$O_5 = OR(O_2 , O_3)$ $O_6 = OR(O_1 , O_4)$ $O_7 = AND(O_5 , O_6)$
1.0	0.0	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.66	0.33	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.33	0.66	$O_5 = OR(O_2 , O_3)$ $O_6 = OR(O_1 , O_4)$ $O_7 = AND(O_5 , O_6)$
0.1	0.9	$O_5 = OR(O_2 , O_3)$ $O_6 = OR(O_1 , O_4)$ $O_7 = AND(O_5 , O_6)$
0.9	0.1	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.95	0.05	$O_5 = XOR(O_1 , O_2)$ $O_6 = XOR(O_3 , O_4)$ $O_7 = XOR(O_5 , O_6)$
0.05	0.95	$O_5 = OR(O_2 , O_3)$ $O_6 = OR(O_1 , O_4)$ $O_7 = AND(O_5 , O_6)$

Table 4.2 The Ten-line decimal-to-8421 BCD converter's compaction trees for different values of S_1 and S_2.

All 56 detectable single stuck-at line faults are injected into the circuit and their effect at the outputs of the circuit is shown in Table 4.3 below.

Fault effect	$_1{}^0O_1$	$_1{}^1O_1$	$_2{}^0O_1$	$_2{}^0O_2$	$_2{}^1O_2$	$_3{}^0O_1$	$_3{}^0O_3$	$_3{}^1O_3$	$_4{}^0O_1$	$_4{}^0O_2$	$_4{}^0O_3$	$_5{}^0O_2$	$_6{}^0O_2$	$_6{}^0O_3$	$_7{}^0O_3$	$_8{}^0O_4$	$_8{}^1O_4$	$_9{}^0O_1$
For the	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
same	0	1	1	0	1	1	0	1	1	0	0	0	0	0	0	0	1	1
grouped	1	1	0	0	1	1	0	1	1	1	0	1	1	0	0	0	1	1
input test	1	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1
patterns	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	1	1
used to	0	1	0	1	1	0	0	1	0	1	0	0	1	0	0	0	1	0
calculate	0	1	0	1	1	0	1	1	0	1	1	1	0	0	1	0	1	0
the	0	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	1	0
fault-free	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0
responses	1	1	1	0	1	1	0	1	1	0	0	0	0	0	0	1	1	0

Fault effect	$_9{}^0O_4$	$_{16}{}^1O_1$	$_{17}{}^1O_1$	$_{17}{}^1O_2$	$_{20}{}^1O_1$	$_{20}{}^1O_3$	$_{23}{}^1O_2$	$_{24}{}^1O_3$	$_{25}{}^1O_4$	$_{26}{}^0O_1$	$_{27}{}^0O_2$	$_{28}{}^0O_3$
For the	0	0	0	0	0	0	0	0	0	0	0	0
same	0	0	1	0	1	0	0	0	0	0	0	0
grouped	0	1	0	0	1	0	1	0	0	0	0	0
input test	0	1	1	0	0	0	0	1	0	0	0	0
patterns	0	1	0	0	0	0	1	1	0	0	0	0
used to	0	0	0	1	0	0	0	0	0	0	0	0
calculate	0	0	0	1	0	1	0	0	0	0	0	0
the	0	0	0	0	0	1	0	0	0	0	0	0
fault-free	1	0	0	0	0	0	0	0	0	0	0	0
responses	0	0	1	0	1	0	0	0	0	0	0	0

Table 4.3 The fault effect at the outputs of the Ten-line decimal-to-8421 BCD converter with the same test set as used to calculate the fault-free responses.

Depending on the choice of S_1 and S_2, we get different lines selection and different types of gates to use when constructing the corresponding space compactor.

The fault-free space compactor response for the 10 input test vectors and for (S_1 , S_2) equaling (0.0 , 1.0), (0.33 , 0.66), (0.1 , 0.9), and (0.05 , 0.95) is the following single 10-bit stream O_7 = (0 0 1 1 1 0 0 0 0 0), while it is the bit-stream O_7 = (0 1 0 0 1 1 0 1 1 0) when (S_1 , S_2) equals (1.0 , 0.0), (0.66 , 0.33), (0.9 , 0.1), and (0.95 , 0.05), respectively.

The fault-free signatures using Syndrome counter as time compressor are 3 and 5, respectively.

For (S_1, S_2) with the following values (0.0 , 1.0), (0.33 , 0.66), (0.1 , 0.9), and (0.05 , 0.95), by comparing the fault-free streams at the circuit's outputs to the faulty bit streams when injecting all 56 detectable faults, we find that the percentage of fault loss is 0.00 % before compaction. Moreover, 17 out of 56 stuck-at line faults turn out to be undetectable after space compaction and thus, we get about 69.64 % fault coverage or 30.35 % fault loss.

For the probability values (1.0 , 0.0), (0.66 , 0.33), (0.9 , 0.1), and (0.95 , 0.05), the compressor is simply a parity tree circuit which is a much better compressor for this circuit (91.07 % fault coverage) as seen previously in the case when stochastic independence of single and double line errors was assumed.

Chapter 5

5. EXPERIMENTAL RESULTS

To demonstrate the feasibility of the proposed new space compaction scheme, independent simulations were performed on various ISCAS 85 combinational benchmark circuits. We have used ATALANTA [46] (fault simulation program developed at Virginia Polytechnic Institute and State University) to generate the fault-free output sequences needed to construct our space compactor circuits and to test the benchmark circuits using reduced test sets accompanied with a random testing session by FSIM fault simulation program [45] to generate pseudorandom test sets, and COMPACTEST [53] program to generate reduced test sets that detect most detectable single stuck-at faults for all benchmark circuits. For each circuit, we determined the number of test vectors used to construct the compaction tree, the CPU time taken to construct the compactor, the number of applied test vectors, the simulation CPU time, and the percentage of fault coverage by running ATALANTA and FSIM programs on a SPARC 5 SUN workstation, and COMPACTEST on an IBM AIX machine. The results are listed in the tables that follow.

The CPU times needed to construct the compactors for all different ISCAS 85 benchmark circuits on a SUN SPARC 5 workstation were in the range of 2.0-102.0 seconds.

For comparison purposes, we used a parity tree space 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. With the parity tree space compactor as reference, we have simulated some combinational benchmark circuits to demonstrate the feasibility of the proposed scheme of constructing a compaction tree using the concept of Hamming distance and sequence weight.

To give an idea about how our space compactor looks like, we have drawn the space compaction circuits for the C432 benchmark circuit with 160 gates, 36 primary inputs and 7 primary outputs, corresponding to different probability values for single and double line errors. Figure 5.1 shows the space compactor assuming stochastic independence of single and double line errors.

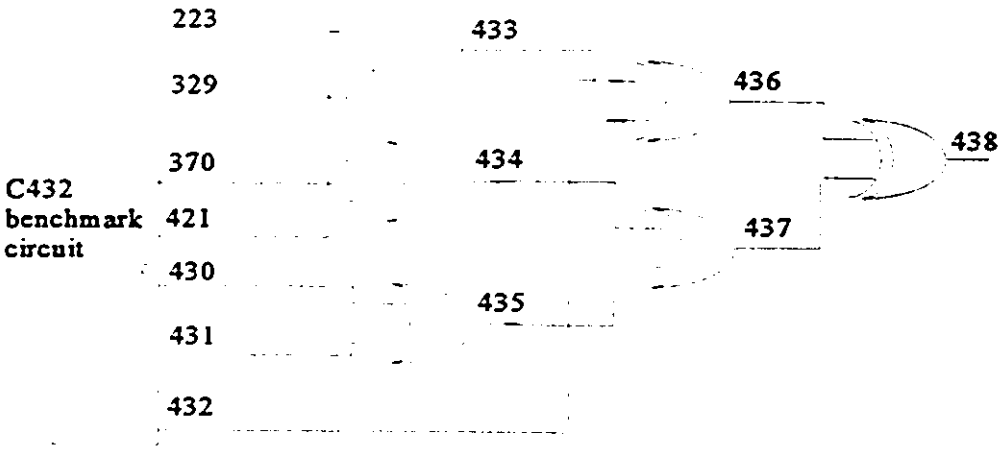


Figure 5.1 Space compactor circuit for C432 assuming stochastic independence of single and double line errors.

Figures 5.2-5.5 illustrate the compaction circuits when (S_1 , S_2) equals (0.1 , 0.9), (0.05 , 0.95), (0.33 , 0.66), (0.66 , 0.33), (0.9 , 0.1), (0.95 , 0.05), (1.0 , 0.0), and (0.0 , 1.0), respectively.

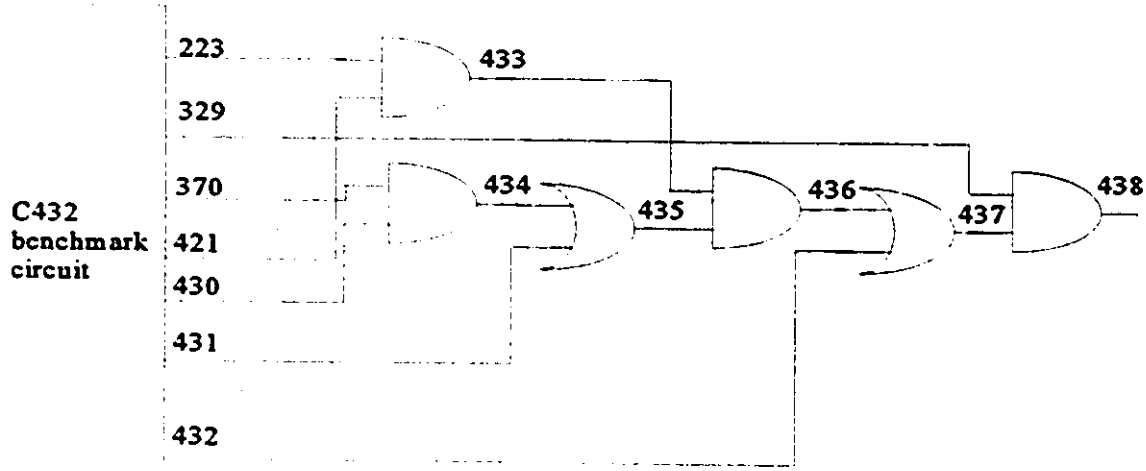


Figure 5.2 Space compactor circuit for C432 assuming different probability values for single and double line errors (i.e. when (S_1 , S_2) equals (0.1 , 0.9), (0.05 , 0.95), and (0.33 , 0.66), respectively).

The hardware overhead for the space compactors shown in Figures 5.1-5.5, measured by the weighted gate count metric (gate count x average fanin), is 3.44 %. It is about 10% less area than what was recently measured by a Multiplexed Parity Tree compactor [17] which provides zero-aliasing.

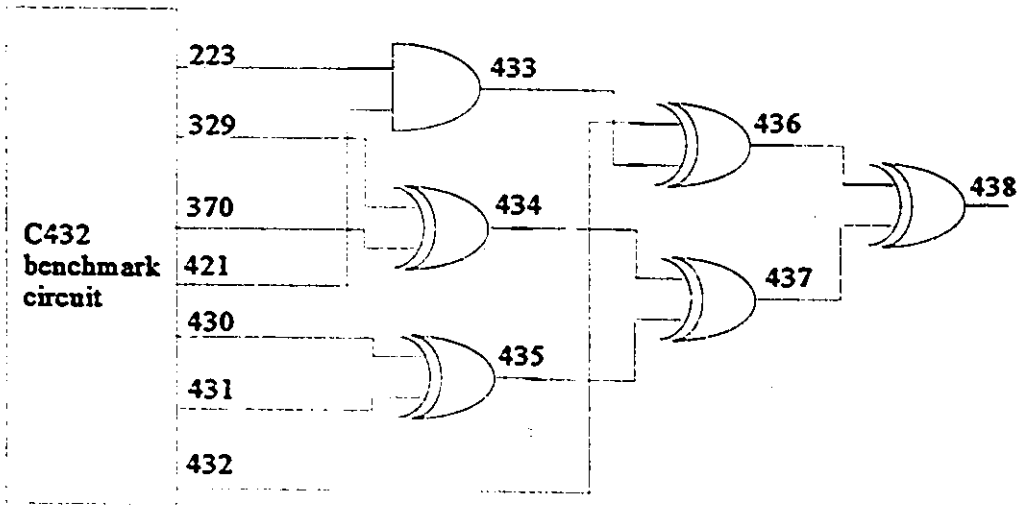


Figure 5.3 Space compactor circuit for C432 assuming different probability values for single and double line errors (i.e. when (S_1 , S_2) equals (0.66 , 0.33)).

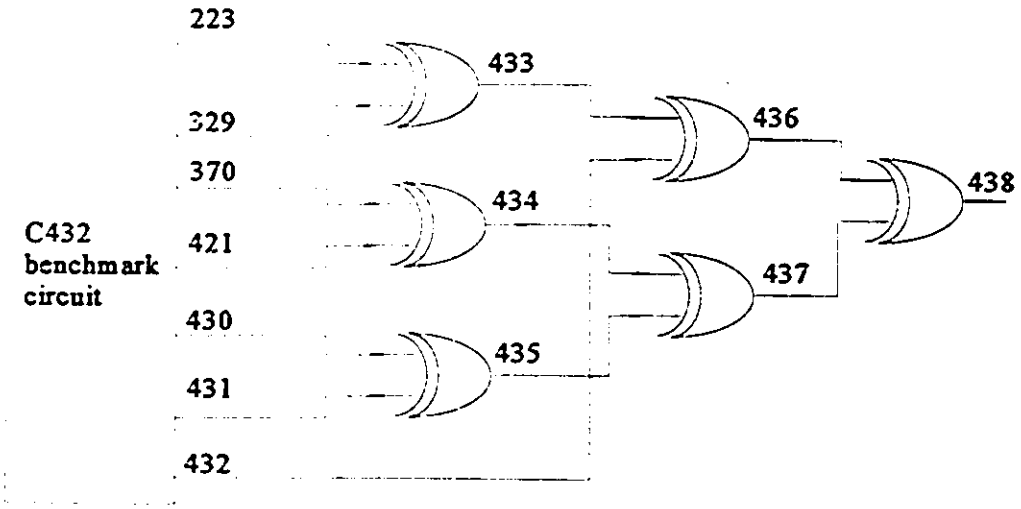


Figure 5.4 Space compactor circuit for C432 assuming different probability values for single and double line errors (i.e. when (S_1 , S_2) equals (0.9, 0.1), (0.95, 0.05), and (1.0, 0.0), respectively).

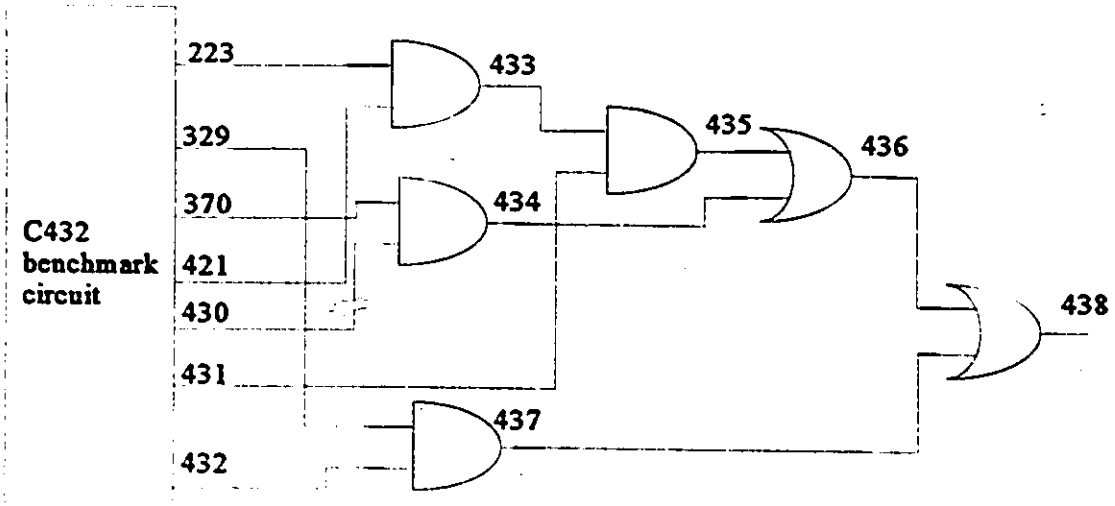


Figure 5.5 Space compactor circuit for C432 assuming different probability values for single and double line errors (i.e. when (S_1 , S_2) equals (0.0 , 1.0)).

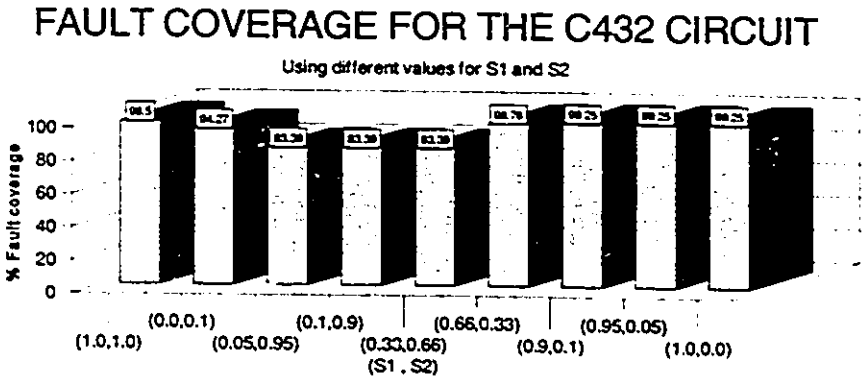


Figure 5.6 The percentage fault coverage for the C432 benchmark circuit.

The above Figure 5.6 illustrates the percentage fault coverage for the C432 circuit, using ATALANTA, and various values for the probabilities of single and double line errors.

Tables 5.1 and 5.2 show the fault coverage and the simulation CPU time for all ISCAS 85 benchmark circuits without compactors using FSIM and ATALANTA, 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
c3540	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.

Circuit name	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	8	0.033	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
c3540	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.

ATALANTA provides 100.00 % fault coverage for the C17 and C880 benchmark circuits and within the range of 95-99 % for other circuits. The fault coverage is much higher than what is provided by FSIM in almost all cases.

Tables 5.3, 5.4, and 5.5 show the simulation results for all benchmark circuits with their compactors when stochastic independence of single and double line errors is assumed, using FSIM and ATALANTA, respectively.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.0	64	0.26	100.00
c432	46	4.2	224	0.43	95.30
c499	54	15.3	224	0.36	91.44
c880	56	12.5	224	0.50	93.70
c1355	86	15.5	224	0.56	89.40
c1908	115	12.5	224	0.76	81.57
c2670	108	117.5	224	0.86	78.75
c3540	144	11.1	224	1.7	83.28
c5315	117	95.8	224	1.46	85.76
c6288	31	15.2	224	4.65	99.56
c7552	217	99.3	224	2.56	87.67

Table 5.3 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming stochastic independence of single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.0	10	0.033	100.00
c432	46	4.2	75	0.91	98.50
c499	54	15.3	86	1.03	96.08
c880	56	12.5	137	1.13	99.00
c1355	86	15.5	111	1.88	98.04
c1908	115	12.5	246	4.60	98.91
c2670	108	117.5	278	17.00	94.45
c3540	144	11.1	270	15.78	94.91
c5315	117	95.8	255	16.50	89.53
c6288	31	15.2	110	28.50	99.56
c7552	217	99.3	333	84.40	95.24

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

Comparing the results given in Tables 5.4 and 5.5, where the program ATALANTA is used, we notice that the fault coverage is almost the same while the CPU time is higher in the case where faults were not considered in compressors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.0	6	0.03	100.00
c432	46	4.2	69	0.70	98.47
c499	54	15.3	54	2.08	95.77
c880	56	12.5	68	2.00	99.04
c1355	86	15.5	87	3.21	97.96
c1908	115	12.5	144	5.80	98.82
c2670	108	117.5	164	24.15	94.10
c3540	144	11.1	166	23.16	94.98
c5315	117	95.8	149	20.85	89.75
c6288	31	15.2	53	28.58	99.56
c7552	217	99.3	194	160.70	95.21

Table 5.5 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA, assuming stochastic independence of single and double line errors, and when faults were not considered in compressors.

Tables 5.6-5.21 show the number of applied test vectors, the simulation CPU time, and the percentage of fault coverage for all benchmark circuits using FSIM and ATALANTA, respectively, for the following probability values of single and double line errors: ((S_1 , S_2) equals (0.33 , 0.66), (0.66 , 0.33), (0.1 , 0.9), (0.9 , 0.1), (0.95 , 0.05), (0.05 , 0.95), (0.0 , 1.0), and (1.0 , 0.0), respectively).

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.5	0.33	0.66	224	0.28	90.90
c432	46	4.3	0.33	0.66	224	0.43	80.15
c499	54	15.5	0.33	0.66	224	0.51	53.82
c880	56	13.8	0.33	0.66	224	0.58	64.86
c1355	86	15.5	0.33	0.66	224	0.68	64.00
c1908	115	12.5	0.33	0.66	224	0.91	63.06
c2670	108	116.5	0.33	0.66	224	1.46	6.66
c3540	144	11.1	0.33	0.66	224	2.30	59.07
c5315	117	95.1	0.33	0.66	224	2.15	30.29
c6288	31	15.4	0.33	0.66	224	7.95	96.56
c7552	217	103.0	0.33	0.66	224	3.86	17.94

Table 5.6 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.5	0.33	0.66	10	0.033	90.90
c432	46	4.3	0.33	0.66	86	1.26	83.39
c499	54	15.5	0.33	0.66	90	1.11	98.15
c880	56	13.8	0.33	0.66	225	2.30	100.00
c1355	86	15.5	0.33	0.66	182	4.20	97.45
c1908	115	12.5	0.33	0.66	317	10.95	95.47
c2670	108	116.5	0.33	0.66	482	39.93	94.53
c3540	144	11.1	0.33	0.66	466	21.93	95.04
c5315	117	95.1	0.33	0.66	772	60.70	87.42
c6288	31	15.4	0.33	0.66	269	85.26	99.51
c7552	217	103.0	0.33	0.66	784	265.36	88.95

Table 5.7 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.66	0.33	64	0.26	100.00
c432	46	4.28	0.66	0.33	224	0.40	95.25
c499	54	15.4	0.66	0.33	224	0.36	91.30
c880	56	14.1	0.66	0.33	224	0.46	93.50
c1355	86	15.5	0.66	0.33	224	0.46	89.39
c1908	115	12.4	0.66	0.33	224	0.76	81.52
c2670	108	117.0	0.66	0.33	224	0.90	78.71
c3540	144	11.0	0.66	0.33	224	1.78	83.35
c5315	117	95.2	0.66	0.33	224	1.45	85.78
c6288	31	15.8	0.66	0.33	224	4.73	99.52
c7552	217	99.5	0.66	0.33	224	2.35	87.82

Table 5.8 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.66	0.33	10	0.033	100.00
c432	46	4.28	0.66	0.33	74	0.80	98.76
c499	54	15.4	0.66	0.33	88	1.20	96.22
c880	56	14.1	0.66	0.33	135	1.41	99.02
c1355	86	15.5	0.66	0.33	113	2.50	98.08
c1908	115	12.4	0.66	0.33	239	4.43	98.91
c2670	108	117.0	0.66	0.33	275	19.47	94.58
c3540	144	11.0	0.66	0.33	273	18.31	94.97
c5315	117	95.2	0.66	0.33	252	11.63	89.51
c6288	31	15.8	0.66	0.33	111	27.75	99.52
c7552	217	99.5	0.66	0.33	336	96.35	95.84

Table 5.9 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.1	0.9	224	0.26	90.90
c432	46	4.3	0.1	0.9	224	0.45	75.95
c499	54	15.2	0.1	0.9	224	0.40	80.21
c880	56	13.3	0.1	0.9	224	0.51	63.48
c1355	86	15.5	0.1	0.9	224	0.56	75.54
c1908	115	12.4	0.1	0.9	224	1.03	49.33
c2670	108	118.6	0.1	0.9	224	1.20	32.98
c3540	144	11.1	0.1	0.9	224	2.40	57.84
c5315	117	95.0	0.1	0.9	224	2.18	32.63
c6288	31	15.2	0.1	0.9	224	8.80	90.99
c7552	217	101.0	0.1	0.9	224	3.10	43.65

Table 5.10 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.1	0.9	7	0.033	90.90
c432	46	4.3	0.1	0.9	99	1.23	83.39
c499	54	15.2	0.1	0.9	161	0.86	98.81
c880	56	13.3	0.1	0.9	224	2.18	100.00
c1355	86	15.5	0.1	0.9	235	3.06	98.92
c1908	115	12.4	0.1	0.9	342	11.71	98.13
c2670	108	118.6	0.1	0.9	425	22.55	94.50
c3540	144	11.1	0.1	0.9	582	35.56	95.04
c5315	117	95.0	0.1	0.9	786	56.40	87.64
c6288	31	15.2	0.1	0.9	292	54.06	99.45
c7552	217	101.0	0.1	0.9	865	120.91	93.70

Table 5.11 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.1	0.9	0.1	64	0.33	100.00
c432	46	4.2	0.9	0.1	224	0.38	95.89
c499	54	15.3	0.9	0.1	224	0.43	94.02
c880	56	13.1	0.9	0.1	224	0.48	92.71
c1355	86	15.4	0.9	0.1	224	0.51	90.89
c1908	115	12.4	0.9	0.1	224	0.83	83.23
c2670	108	116.5	0.9	0.1	224	0.91	67.37
c3540	144	11.1	0.9	0.1	224	1.71	86.51
c5315	117	96.7	0.9	0.1	224	1.48	94.72
c6288	31	15.1	0.9	0.1	224	4.73	99.46
c7552	217	100.4	0.9	0.1	224	2.43	87.81

Table 5.12 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.1	0.9	0.1	10	0.033	100.00
c432	46	4.2	0.9	0.1	108	1.35	99.25
c499	54	15.3	0.9	0.1	67	1.13	96.09
c880	56	13.1	0.9	0.1	124	1.48	99.59
c1355	86	15.4	0.9	0.1	109	2.46	98.04
c1908	115	12.4	0.9	0.1	240	4.60	98.91
c2670	108	116.5	0.9	0.1	116	77.13	69.15
c3540	144	11.1	0.9	0.1	294	17.96	95.13
c5315	117	96.7	0.9	0.1	254	10.41	98.67
c6288	31	15.1	0.9	0.1	83	39.45	99.52
c7552	217	100.4	0.9	0.1	330	104.83	95.09

Table 5.13 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.95	0.05	64	0.30	100.00
c432	46	4.3	0.95	0.05	224	0.45	94.21
c499	54	16.7	0.95	0.05	224	0.40	94.14
c880	56	13.0	0.95	0.05	224	0.43	93.54
c1355	86	15.5	0.95	0.05	224	0.55	92.48
c1908	115	12.4	0.95	0.05	224	0.78	84.17
c2670	108	116.1	0.95	0.05	224	0.91	69.91
c3540	144	11.1	0.95	0.05	224	1.75	84.35
c5315	117	96.5	0.95	0.05	224	1.43	95.37
c6288	31	15.6	0.95	0.05	224	4.61	99.52
c7552	217	100.3	0.95	0.05	224	2.41	87.64

Table 5.14 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.95	0.05	9	0.033	100.00
c432	46	4.3	0.95	0.05	108	1.65	99.25
c499	54	16.7	0.95	0.05	61	1.13	96.09
c880	56	13.0	0.95	0.05	127	1.25	99.29
c1355	86	15.5	0.95	0.05	103	2.63	98.04
c1908	115	12.4	0.95	0.05	231	4.66	98.80
c2670	108	116.1	0.95	0.05	114	82.21	79.26
c3540	144	11.1	0.95	0.05	315	19.53	95.13
c5315	117	96.5	0.95	0.05	230	9.41	98.65
c6288	31	15.6	0.95	0.05	78	28.63	99.52
c7552	217	100.3	0.95	0.05	339	106.70	95.15

Table 5.15 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.05	0.95	224	0.25	90.90
c432	46	4.35	0.05	0.95	224	0.41	78.43
c499	54	15.8	0.05	0.95	224	0.41	64.38
c880	56	12.6	0.05	0.95	224	0.55	64.22
c1355	86	15.5	0.05	0.95	224	0.66	38.62
c1908	115	12.4	0.05	0.95	224	0.98	54.55
c2670	108	116.3	0.05	0.95	224	1.16	40.59
c3540	144	11.2	0.05	0.95	224	2.30	66.54
c5315	117	98.2	0.05	0.95	224	2.06	38.56
c6288	31	15.5	0.05	0.95	224	9.30	92.91
c7552	217	98.6	0.05	0.95	224	3.00	49.08

Table 5.16 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.2	0.05	0.95	10	0.033	90.90
c432	46	4.35	0.05	0.95	92	1.26	83.39
c499	54	15.8	0.05	0.95	171	1.05	98.68
c880	56	12.6	0.05	0.95	227	2.00	100.00
c1355	86	15.5	0.05	0.95	218	3.60	97.71
c1908	115	12.4	0.05	0.95	377	8.98	99.30
c2670	108	116.3	0.05	0.95	420	21.66	94.53
c3540	144	11.2	0.05	0.95	479	20.91	95.50
c5315	117	98.2	0.05	0.95	751	45.50	87.66
c6288	31	15.5	0.05	0.95	282	56.33	99.41
c7552	217	98.6	0.05	0.95	833	159.60	92.88

Table 5.17 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.15	0.0	1.0	224	0.28	90.90
c432	46	4.28	0.0	1.0	224	0.40	84.73
c499	54	15.5	0.0	1.0	224	0.40	80.60
c880	56	12.9	0.0	1.0	224	0.56	70.91
c1355	86	15.8	0.0	1.0	224	0.63	51.58
c1908	115	12.7	0.0	1.0	224	0.90	71.10
c2670	108	123.0	0.0	1.0	224	1.21	42.55
c3540	144	12.5	0.0	1.0	224	2.33	67.91
c5315	117	101.2	0.0	1.0	224	2.31	33.79
c6288	31	16.0	0.0	1.0	224	6.83	96.38
c7552	217	102.1	0.0	1.0	224	2.90	44.94

Table 5.18 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.15	0.0	1.0	8	0.033	90.90
c432	46	4.28	0.0	1.0	113	1.20	94.27
c499	54	15.5	0.0	1.0	158	0.85	98.02
c880	56	12.9	0.0	1.0	243	2.76	100.00
c1355	86	15.8	0.0	1.0	212	2.88	97.84
c1908	115	12.7	0.0	1.0	343	6.96	99.36
c2670	108	123.0	0.0	1.0	417	19.53	94.53
c3540	144	12.5	0.0	1.0	499	26.23	95.44
c5315	117	101.2	0.0	1.0	804	47.91	87.90
c6288	31	16.0	0.0	1.0	247	63.91	99.56
c7552	217	102.1	0.0	1.0	867	123.86	94.41

Table 5.19 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.1	1.0	0.0	64	0.30	100.00
c432	46	4.3	1.0	0.0	224	0.40	95.33
c499	54	15.6	1.0	0.0	224	0.36	93.41
c880	56	12.7	1.0	0.0	224	0.46	94.15
c1355	86	15.8	1.0	0.0	224	0.48	90.83
c1908	115	12.6	1.0	0.0	224	0.83	81.00
c2670	108	119.9	1.0	0.0	224	0.91	68.00
c3540	144	11.3	1.0	0.0	224	1.75	84.12
c5315	117	96.8	1.0	0.0	224	1.45	94.92
c6288	31	15.2	1.0	0.0	224	4.76	99.48
c7552	217	100.7	1.0	0.0	224	2.38	87.95

Table 5.20 Simulation results of the ISCAS 85 benchmark circuits using FSIM and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.1	1.0	0.0	9	0.033	100.00
c432	46	4.3	1.0	0.0	119	1.40	99.25
c499	54	15.6	1.0	0.0	73	1.15	96.09
c880	56	12.7	1.0	0.0	103	1.30	99.29
c1355	86	15.8	1.0	0.0	112	2.40	98.04
c1908	115	12.6	1.0	0.0	230	4.25	98.91
c2670	108	119.9	1.0	0.0	118	66.13	69.05
c3540	144	11.3	1.0	0.0	297	16.95	95.10
c5315	117	96.8	1.0	0.0	252	9.31	98.69
c6288	31	15.2	1.0	0.0	91	23.53	99.52
c7552	217	100.7	1.0	0.0	333	103.80	95.31

Table 5.21 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA and assuming different probability values for single and double line errors.

Furthermore, we used COMPACTEST program to simulate the circuits. Table 5.22 shows the simulation results before compaction. It shows higher fault coverage than what we have got with

FSIM and ATALANTA but higher CPU simulation time as well. Table 5.23 shows all the results after compaction and when stochastic independence of single and double line errors is assumed.

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
c3540	110	266.70	95.91
c5315	55	36.54	98.89
c6288	16	73.51	99.56
c7552	85	151.04	98.26

Table 5.22 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.12	6	0.00	100.00
c432	68	4.30	68	10.19	97.71
c499	63	15.90	57	44.50	95.77
c880	30	12.51	60	4.74	99.04
c1355	96	16.01	97	44.55	97.96
c1908	137	12.70	167	85.54	98.88
c2670	68	99.22	112	226.54	92.57
c3540	110	11.25	146	443.86	94.86
c5315	55	75.50	90	440.85	89.73
c6288	16	15.32	39	59.78	99.56
c7552	85	69.54	141	715.54	96.42

Table 5.23 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming stochastic independence of single and double line errors.

The simulation results given in Tables 5.24 and 5.31 are for different values of the probabilities of single line (S_1) and double line (S_2) errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.10	0.66	0.33	6	0.00	100.00
c432	44	4.37	0.66	0.33	67	9.76	97.69
c499	63	15.80	0.66	0.33	56	45.56	95.78
c880	30	12.53	0.66	0.33	58	4.54	99.00
c1355	96	15.81	0.66	0.33	96	42.46	97.96
c1908	137	12.66	0.66	0.33	166	83.25	98.72
c2670	68	98.66	0.66	0.33	110	236.54	92.43
c3540	110	11.20	0.66	0.33	147	456.65	94.96
c5315	55	75.48	0.66	0.33	92	444.83	89.97
c6288	16	15.37	0.66	0.33	39	61.00	99.50
c7552	85	69.83	0.66	0.33	140	704.52	96.17

Table 5.24 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.13	0.33	0.66	6	0.00	100.00
c432	44	4.26	0.33	0.66	65	27.30	91.98
c499	63	15.88	0.33	0.66	88	42.48	95.64
c880	30	12.50	0.33	0.66	119	4.61	99.25
c1355	96	16.13	0.33	0.66	92	175.01	92.56
c1908	137	12.68	0.33	0.66	217	81.81	98.08
c2670	68	98.73	0.33	0.66	240	462.98	94.72
c3540	110	11.18	0.33	0.66	349	428.30	94.48
c5315	55	75.18	0.33	0.66	396	346.48	95.81
c6288	16	15.36	0.33	0.66	104	274.16	99.50
c7552	85	69.41	0.33	0.66	438	885.28	94.47

Table 5.25 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.14	0.10	0.90	6	0.01	100.00
c432	44	4.31	0.10	0.90	65	27.78	91.98
c499	63	15.91	0.10	0.90	88	44.11	95.64
c880	30	12.55	0.10	0.90	118	5.29	98.93
c1355	96	16.03	0.10	0.90	111	79.81	97.33
c1908	137	12.95	0.10	0.90	213	91.06	97.71
c2670	68	98.91	0.10	0.90	238	261.07	94.24
c3540	110	11.10	0.10	0.90	248	352.83	89.30
c5315	55	75.45	0.10	0.90	342	685.50	99.50
c6288	16	15.33	0.10	0.90	104	275.23	95.35
c7552	85	69.53	0.10	0.90	429	911.24	95.35

Table 5.26 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.11	0.90	0.10	6	0.00	100.00
c432	44	4.21	0.90	0.10	63	4.09	99.23
c499	63	15.76	0.90	0.10	62	34.83	95.77
c880	30	12.53	0.90	0.10	50	5.94	99.25
c1355	96	16.10	0.90	0.10	96	41.24	97.96
c1908	137	12.71	0.90	0.10	167	80.77	98.88
c2670	68	98.33	0.90	0.10	59	615.17	66.14
c3540	110	11.03	0.90	0.10	143	615.17	94.63
c5315	55	75.33	0.90	0.10	94	456.06	98.43
c6288	16	15.15	0.90	0.10	34	208.12	99.52
c7552	85	70.38	0.90	0.10	101	692.71	96.14

Table 5.27 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.13	0.05	0.95	6	0.01	100.00
c432	44	4.28	0.05	0.95	65	27.93	91.98
c499	63	15.70	0.05	0.95	88	41.60	95.64
c880	30	12.52	0.05	0.95	118	5.51	98.93
c1355	96	16.01	0.05	0.95	111	80.77	97.33
c1908	137	12.78	0.05	0.95	211	91.88	97.81
c2670	68	98.60	0.05	0.95	227	230.09	94.68
c3540	110	11.20	0.05	0.95	248	354.90	95.09
c5315	55	75.68	0.05	0.95	339	624.30	89.30
c6288	16	15.16	0.05	0.95	104	267.11	99.50
c7552	85	69.85	0.05	0.95	429	860.39	95.35

Table 5.28 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.13	0.95	0.05	6	0.01	100.00
c432	44	4.28	0.95	0.05	63	4.04	99.23
c499	63	15.75	0.95	0.05	63	34.10	96.30
c880	30	12.60	0.95	0.05	55	6.89	99.15
c1355	96	16.03	0.95	0.05	96	41.45	97.96
c1908	137	12.68	0.95	0.05	167	80.33	98.88
c2670	68	98.36	0.95	0.05	82	537.46	77.68
c3540	110	11.18	0.95	0.05	143	469.97	94.63
c5315	55	75.33	0.95	0.05	92	241.79	98.15
c6288	16	15.11	0.95	0.05	34	85.72	99.52
c7552	85	69.26	0.95	0.05	111	683.23	96.18

Table 5.29 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.12	1.00	0.00	6	0.01	100.00
c432	44	4.35	1.00	0.00	63	4.05	99.23
c499	63	15.70	1.00	0.00	62	33.98	95.77
c880	30	12.51	1.00	0.00	111	3.06	99.25
c1355	96	16.01	1.00	0.00	96	41.45	97.96
c1908	137	12.85	1.00	0.00	167	80.43	98.88
c2670	68	99.21	1.00	0.00	62	686.01	66.18
c3540	110	11.20	1.00	0.00	143	451.50	94.63
c5315	55	76.03	1.00	0.00	86	97.87	98.71
c6288	16	15.15	1.00	0.00	34	83.01	99.52
c7552	85	69.83	1.00	0.00	109	684.00	96.19

Table 5.30 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	S_1	S_2	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	4	2.12	0.00	1.00	6	0.00	100.00
c432	44	4.28	0.00	1.00	68	25.42	93.70
c499	63	15.81	0.00	1.00	83	42.94	95.77
c880	30	12.78	0.00	1.00	55	6.94	99.15
c1355	96	16.21	0.00	1.00	105	176.61	94.47
c1908	137	12.75	0.00	1.00	211	91.70	97.81
c2670	68	99.36	0.00	1.00	230	340.97	94.53
c3540	110	11.11	0.00	1.00	249	406.91	95.18
c5315	55	75.00	0.00	1.00	375	645.01	89.04
c6288	16	15.45	0.00	1.00	134	295.36	99.10
c7552	85	69.66	0.00	1.00	453	842.42	95.57

Table 5.31 Simulation results of the ISCAS 85 benchmark circuits using COMPACTEST and assuming different probability values for single and double line errors.

Figure 5.7 below shows the No. of test vectors, CPU time, and the % fault coverage for all circuits using FSIM, ATALANTA, and COMPACTEST, respectively, when stochastic independence for single and double line errors is assumed.

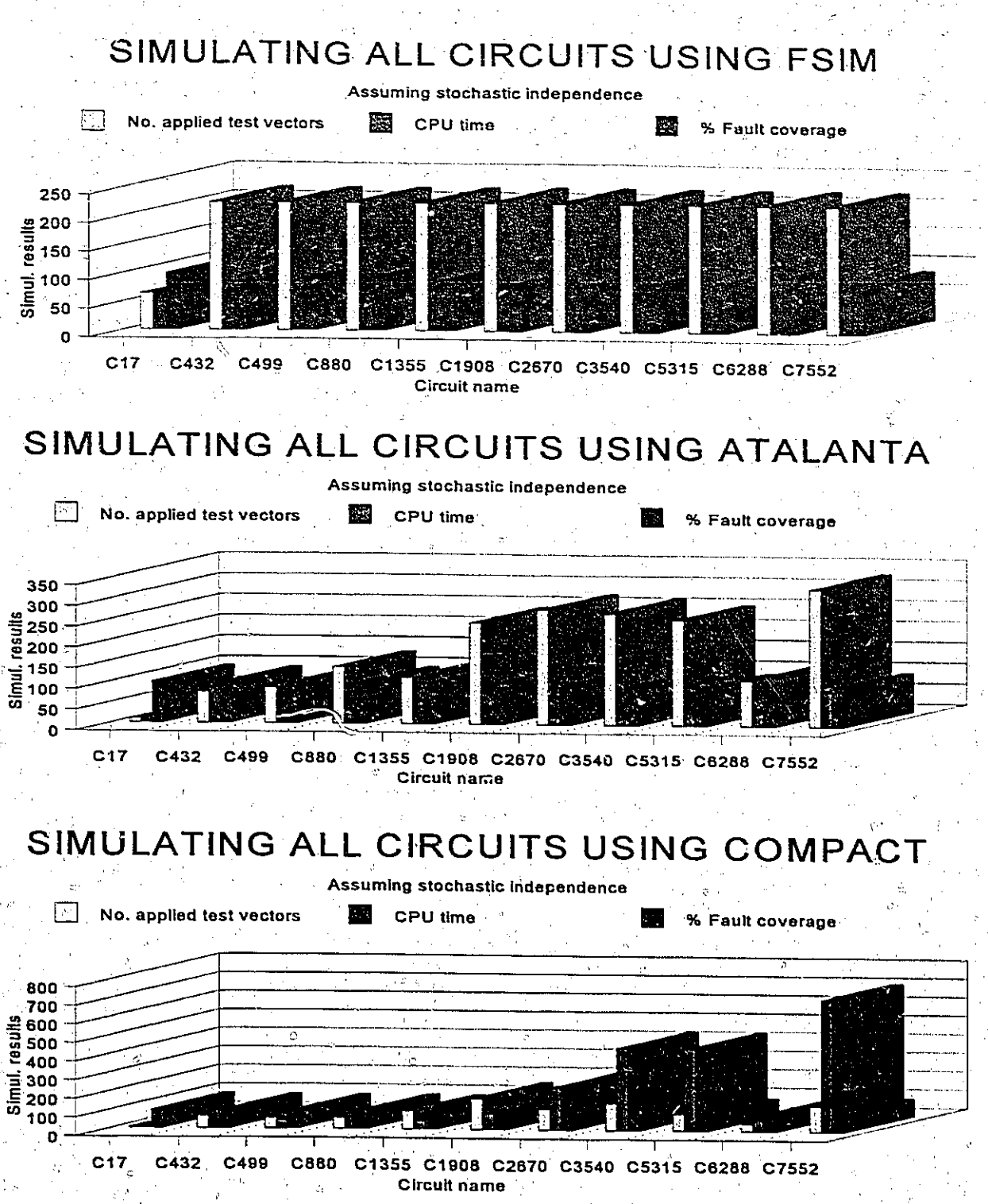


Figure 5.7 Simulation results for all benchmark circuits using FSIM, ATALANTA, and COMPACTEST programs and assuming stochastic independence for single and double line errors.

We also simulated all benchmark circuits using pseudorandom testing (FSIM) with the same test sets in both cases when stochastic independence and stochastic dependence of single and double line errors are assumed. Table 5.32 shows the simulation results for all circuits before compaction.

Circuit name	No. of identical applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	5	0.30	100.00
c432	53	0.35	99.23
c499	54	0.33	98.94
c880	49	0.40	100.00
c1355	85	0.45	99.49
c1908	117	0.61	99.52
c2670	104	0.73	95.74
c3540	145	1.26	96.00
c5315	116	1.13	98.89
c6288	37	2.81	99.56
c7552	208	1.73	98.26

Table 5.32 Simulation results of the ISCAS 85 benchmark circuits using FSIM with identical test sets.

Table 5.33 shows the simulation results for all circuits after compaction and assuming stochastic independence of single and double line errors.

Circuit name	No. of test vectors used to construct the compaction tree	CPU time taken to construct the compaction tree	No. of identical applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	2.5	5	0.30	95.83
c432	46	4.3	53	0.38	88.95
c499	54	15.5	54	0.36	90.44
c880	56	13.8	49	0.45	93.78
c1355	86	15.5	85	0.50	94.79
c1908	115	12.5	117	0.75	96.10
c2670	108	116.5	104	0.78	87.19
c3540	144	11.1	145	1.63	92.69
c5315	117	95.1	116	1.35	85.32
c6288	31	15.4	37	4.60	97.93
c7552	217	103.0	208	2.66	94.58

Table 5.33 Simulation results of the ISCAS 85 benchmark circuits using FSIM with identical test sets and assuming stochastic independence of single and double line errors.

Moreover, we simulated all ISCAS 85 benchmark circuits with a parity tree space compactor using FSIM and ATALANTA, respectively. Tables 5.34 and 5.35 show the simulation results when circuits are simulated after being compacted.

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
c3540	224	1.71	85.70
c5315	224	1.45	95.06
c6288	224	4.91	99.46
c7552	224	2.41	88.00

Table 5.34 Simulation results of the ISCAS 85 benchmark circuits using FSIM with parity tree as a space compressor .

Circuit name	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	9	0.033	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
c3540	285	19.00	94.98
c5315	271	8.78	98.57
c6288	87	30.50	99.52
c7552	332	114.00	95.15

Table 5.35 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA with parity tree as a space compressor.

Table 5.36 shows the simulation results using ATALANTA for all benchmark circuits after compaction and when faults were not considered in compressors.

Circuit name	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	7	0.03	100.00
c432	67	0.78	99.04
c499	52	2.76	95.77
c880	66	1.25	99.25
c1355	84	5.00	97.96
c1908	146	6.15	98.88
c2670	73	197.80	66.18
c3540	166	26.00	94.77
c5315	129	11.78	98.71
c6288	47	27.38	99.52
c7552	203	228.43	94.88

Table 5.36 Simulation results of the ISCAS 85 benchmark circuits using ATALANTA with parity tree as a space compressor and when faults were not considered in compressors.

Table 5.37 shows the simulation results using pseudorandom testing when the same test sets are applied as in the previous case.

Circuit name	No. of applied test vectors	CPU simulation time (secs)	Fault coverage (%)
c17	5	0.26	95.83
c432	53	0.36	88.43
c499	54	0.41	94.87
c880	49	0.48	96.77
c1355	85	0.48	97.43
c1908	117	0.75	96.10
c2670	104	0.88	68.82
c3540	145	1.66	92.65
c5315	116	1.31	96.03
c6288	37	4.70	98.20
c7552	208	2.33	95.29

Table 5.37 Simulation results of the ISCAS 85 benchmark circuits using FSIM with parity tree as a space compressor and identical test sets.

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 a *reduction in the CPU time* using our space compactor than what we have when a parity tree compactor is used.

We also estimated the hardware overhead of the compressor for different benchmark circuits and found it to be as small as 1-5 % in most cases and within 15% for three benchmark circuits. Table 5.38 shows the hardware overhead estimates for all ISCAS 85 benchmark circuits. To estimate the hardware overhead, we used the ratio of the weighted gate count metric (i.e. average fanins x number of gates) of the compressor and that of the total circuit comprised of the CUT and the space compactor.

Compactor circuit for	Total no. of fanin in the compactor	Total no. of gates in the compactor	Average fanin in the compactor	Total no. of gates in the CUT	Average fanin in the CUT	Hardware overhead (%)
c17	2	1	2.00	6	2.00	14.28
c432	12	6	2.00	160	2.10	3.44
c499	62	31	2.00	202	2.02	13.19
c880	50	25	2.00	383	1.90	6.42
c1355	62	31	2.00	546	1.95	5.50
c1908	48	24	2.00	880	1.70	3.10
c2670	278	139	2.00	1269	1.64	11.78
c3540	42	21	2.00	1669	1.76	1.40
c5315	244	122	2.00	2307	1.90	5.27
c6288	62	31	2.00	2416	1.99	1.27
c7552	214	107	2.00	3513	1.75	3.36

Table 5.38 Estimates of the hardware overhead.

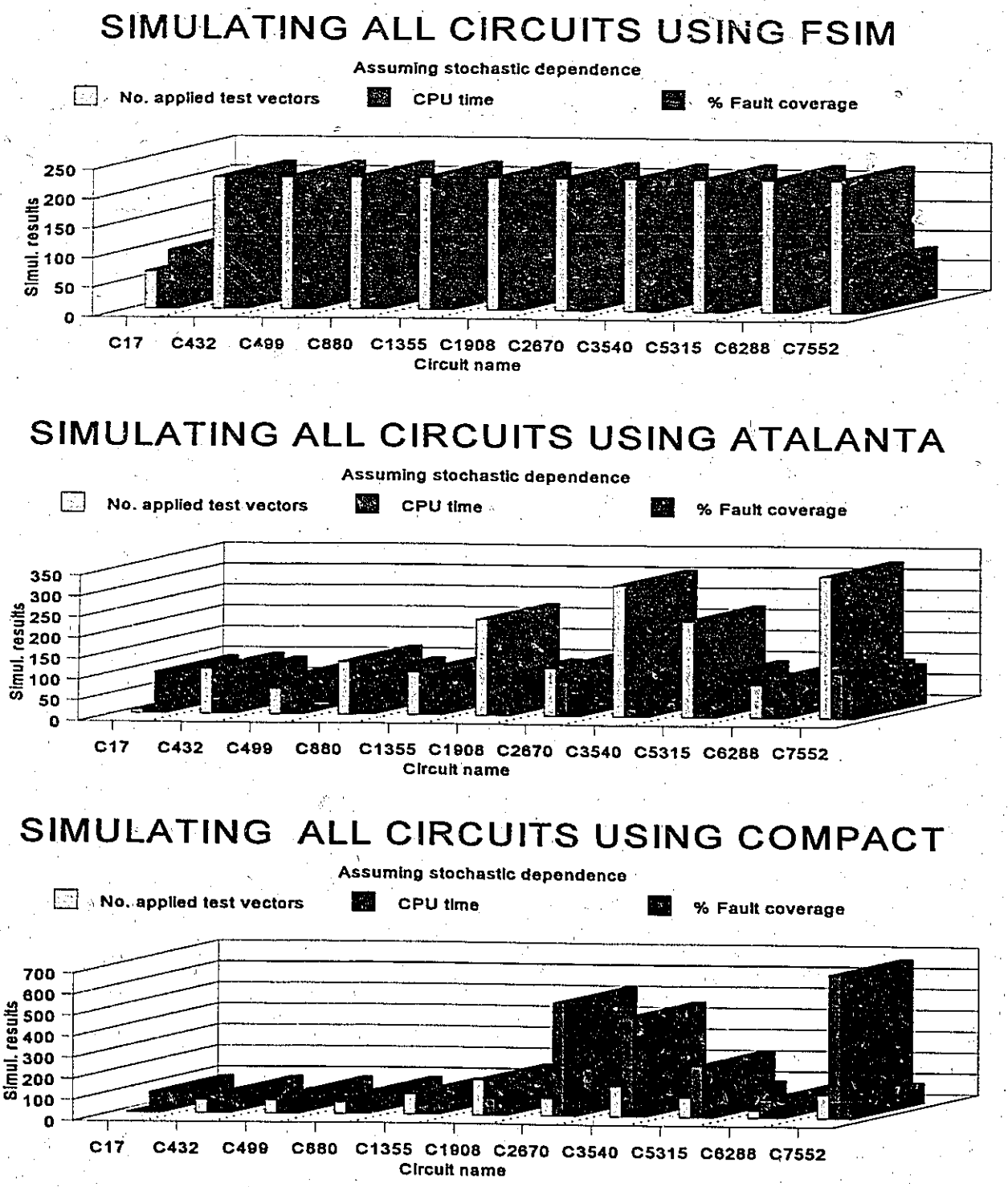


Figure 5.8 Simulation results for all benchmark circuits using FSIM, ATALANTA, and COMPACTEST programs and assuming stochastic dependence for single and double line errors ((S_1 , S_2) equals (0.95 , 0.05)).

Figure 5.8 shows the No. of applied test vectors, the CPU time, and the % fault coverage for all benchmark circuits using FSIM, ATALANTA, and COMPACTEST, respectively, assuming stochastic dependence for single line (S_1) and double line (S_2) errors. To illustrate the different measures we got with all simulation programs, we chose (S_1 , S_2) as (0.95 , 0.05). The percentage of fault coverage measured using COMPACTEST program was higher (close to 99 % in most cases) than what was measured by ATALANTA and FSIM, but at the same time COMPACTEST requires more CPU time than the other two fault simulation programs.

CHAPTER 6

6. CONCLUSIONS

In the present dissertation we described a space compaction technique of test responses for digital combinational circuits. This technique uses AND (NAND), OR (NOR), and XOR (XNOR) gates appropriately to construct an output compaction tree that compress the m outputs of the CUT to a single line. The compaction tree is generated by using the concepts of Hamming distance and sequence weight. The logic function selected to build the tree is determined by the characteristics of the sequences which are inputs to the gates. The mergeability criteria were obtained on the assumption of stochastic independence or dependence of single line and double line errors.

When we do not assume stochastic independence for single line and double line errors, the probability plays a role in the selection of gates in certain cases. In this latter case (stochastic dependence), output selection is based on calculating the detectable error probability estimates.

In our work, we do not emphasize designing an aliasing-free compressor but rather endeavour to show how the input test sets and their length play a role in the construction of the compaction tree. Loss of information may not be avoided when the size of all output responses is reduced. Therefore, depending on the amount of information loss, the corresponding space compactor design will be affected. In our design experiments we used the reduced test sets provided by ATALANTA and COMPACTEST to simulate all the ISCAS 85 benchmark circuits.

Even though these reduced input test sets are not the minimal test sets needed to ensure a 100 % fault coverage, experimental results indicate the designed space compactors are comparable in all respects with the parity tree space compactor which propagates all errors that appear on an odd number of inputs and which is usually considered ideal for *ad hoc* space compaction.

Some of the advantages inherent in the proposed method are its simplicity and the resulting low overhead for the designed compactor which might make it suitable for built-in self-testing in VLSI design even though it does not guarantee 100 % fault coverage.

The technique was illustrated through designing space compactors for all benchmark circuits with elaborate detail.

Finally, Htesting can be combined with efficient input test patterns to synthesize BIST circuits that provide more than 99 % fault coverage with a small CPU time and low area overhead.

BIBLIOGRAPHY

[1] D. R. Aadsen, H. N. Scholz and Y. Zorian, "Automated BIST for regular structures embedded in ASIC devices", *AT&T Technical Journal*, vol. 69, pp. 97-109, May/June 1990.

[2] P. Agrawal and V. D. Agrawal, "Probabilistic analysis of random test generation method for irredundant combinational networks", *IEEE Transactions on Computers*, vol. C-24, pp. 691-694, July 1975.

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

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

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

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

[7] P. H. Bardell, W. H. McAnney, and J. Savir. *Built-in Test for VLSI: Pseudorandom Techniques*. John Wiley, NY, 1987.

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

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

[10] K. Chakrabarty. *Balanced Boolean Functions: Theory and Applications*. M.S. thesis, Department of Electrical Engineering and Computer Science, University of Michigan, MI, 1992; also published as *Technical Report CSE-TR-140-92*.

[11] K. Chakrabarty and J. P. Hayes, "Aliasing-free error detection (ALFRED)", *Digest of Papers 1993 IEEE VLSI Test Symp*, pp. 260-266, 1993.

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

[13] 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.

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

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

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

[18] A. Chatterjee and J. A. Abraham, "Test generation, design-for-testability and built-in self-test for arithmetic units based on graph labelling", *Journal of Electronic Testing: Theory and Applications*, vol. 2, pp. 351-372, December 1991.

[19] 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.

[20] K.-T. Cheng and V. D. Agrawal. *Unified Methods for VLSI Simulation and Test Generation*. Kluwer Academic Publishers, MA, 1989.

[21] A. Crouch, M. Pressly and J. Circello, "Testability features of the MC68060 microprocessor", *Proc. 1994 Int. Test Conference*, pp. 60-69, 1994.

[22] W. Daehn and J. Mucha, "Hardware test pattern generation for built-in testing", *Proc. 1981 Int. Test Conference*, pp. 110-113, 1981.

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

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

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

[26] R. David, "Comments on 'Signature analysis for multiple output circuits'", *IEEE Transactions on Computers*, vol. 39, pp. 287-288, February 1990.

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

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

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

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

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

[32] 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.

[33] S. W. Golomb. *Shift Register Sequences*. Holden-Day, CA, 1967.

[34] 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.

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

[36] 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.

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

[38] 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.

[39] A. Ivanov and V. K. Agarwal, "An iterative technique for calculating aliasing probability of linear feedback shift register", *Proc. 1988 Int. Symp. on Fault-Tolerant Computing*, pp. 70-75, 1988.

[40] 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.

[41] W.-B. Jone and C.-J. Wu, "Multiple fault detection in parity checkers", *IEEE Transactions on Computers*, vol. 43, pp. 1096-1099, September 1994.

[42] 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.

[43] W. H. Kautz, "Testing for faults in combinational logic arrays", *Proc. 8th Annual Switching and Automata Theory Symposium*, pp. 161-174, 1967.

[44] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, NY, 1978.

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

[46] 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.

[47] 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.

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

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

[50] E. J. McCluskey, "Built-in self-test techniques", *IEEE Design and Test of Computers*. vol. 2, pp. 21-28, January 1985.

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

[52] E. J. McCluskey and S. Bozorgui-Nesbat, "Design for autonomous test", *IEEE Transactions on Computers*. vol. C-30, pp. 866-875, November 1981.

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

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

[55] 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.

[56] R. Raina and P. N. Marinos, "Signature analysis with modified linear feedback shift registers", *In Proc. 1991 Int. Symp. on Fault-Tolerant Computing*, pp. 88-95, 1991.

[57] J. Rajski and J. Tyser, "Accumulator-based compaction of test responses", *IEEE Transactions on Computers*, vol. 42, pp. 643-650, June 1993.

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

[59] J. P. Robinson and N. R. Saxena, "Simultaneous signature ans syndrome compression", *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 584-589, May 1988.

[60] J. P. Roth, "Diagnosis of automata failures: a calculus and a method", *IBM Journal of Res. and Dev.*, vol. 10, pp. 278-291, July 1966.

[61] K. K. Saluja and M. Karpovsky, "Testing computer hardware through data compression in space and time", *Proc. Int. Test. Conference*, pp. 83-88, 1983.

[62] J. Savir, "Syndrome-testable design of combinational circuits", *IEEE Transactions on Computers*, vol. C-29, pp. 442-451, June 1980.

[63] J. Savir and W. H. McAnney, "On the masking probability with one's count and transition count", *Proc. Int. Conference on Computer-Added Design*, pp.111-113, 1985.

[64] 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.

[65] 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.

[66] N. R. Saxena, P. Franco, and E. J. McCluskey, "Bounds on signature analysis for random testing", *Proc. 1991 Int. Symp. Fault-Tolerant Computing*, pp. 104-111, 1991.

[67] J. E. Smith, "Measures of effectiveness of fault signature analysis", *IEEE Transactions on Computers*, vol. C-29, pp. 510-514, June 1980.

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

[69] Texas Instruments. *The TTL Data Book*. vol. 2. TX, 1988.

[70] T. W. Williams, "VLSI testing", *Computer*, vol. 17, pp. 126-136, October 1984.

[71] T. W. Williams, K. P. Parker, "Testing logic networks and design for testability", *Computer*, vol. 21, pp. 9-21, October 1979.

[72] T. W. Williams and K. P. Parker, "Design for testability-a survey", *Proc. of the IEEE*, vol. 71, pp. 98-112, January 1983.

[73] T. W. Williams, W. Daehn, M. Gruetzner, and C. W. Starke, "Aliasing errors in signature analysis registers", *IEEE Design and Test of Computers*, vol. 4, pp. 39-45, April 1987.

[74] V. N. Yarmolik. *Fault Diagnosis of Digital Circuits*. John Willey, NY, 1990.

[75] Y. Zorian and V. K. Agarwal, "Higher certainty of error coverage by output data modification", *Proc. Int. Test Conference*, pp. 140-147, 1984.

[76] Y. Zorian and V. K. Agarwal, "A general scheme to optimize error masking in built-in self-testing", *Proc. 1986 Int. Symp. On Fault-Tolerant Computing*, pp. 410-415, 1986.

[77] Y. Zorian and Ivanov, "Programmable space compaction for BIST", *Proc. 1993 Int. Symp. on Fault-Tolerant Computing*, pp. 340-349, 1993.

List of Papers by the Candidate

1) M. H. Assaf, "Selecting outputs for merger in space compression using concepts of hamming distance and sequence weights", Presented at the *IASTED International Conference on Modelling, Simulation and Optimization*, Gold Coast, Australia, May 6-9, 1996 (Conference Proceedings, pp. 1-9).
(Jointly with S. R. Das and A. R. Nayak)

2) M. H. Assaf, "Realizing Ultimate Compression with Acceptable Fault Coverage Degradation to Reduce MISR Size in BIST Applications by Nonexhaustive Test Patterns", *International Conference on Circuits and Systems* , Hong Kong, June 9-12, 1997 (Submitted).
(Jointly with S. R. Das, A. R. Nayak, and W. B. Jone)

3) M. H. Assaf, "On the design of space compressor for VLSI circuits in BIST using nonexhaustive test sets", *Systems Design and Software Engineering Symposium (Second World Conference on Integrated Design and Process Technology)*, Austin, TX, December 1-4, 1996 (Accepted).
(Jointly with S. R. Das and A. R. Nayak)

APPENDIX

SOFTWARE USED TO CONSTRUCT SPACE COMPACTION TREES FOR COMBINATIONAL LOGIC CIRCUITS

```
/*****************************************************************************

        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written
        permission of Dr. Das or Mr. Assaf.

        For detailed information, please contact :

        Dr. Sunil R. Das
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*****************************************************************************/

/************************** HISTORY ***********************************

        HTesting: Version 1.0

        Original: M. H. Assaf, 15/04/1996

*****************************************************************************/
/*---------------------------------------------------------------------------
        Hamming
        HTesting constructs the compaction tree for combinational circuits
        based on the mergeability criteria.
        Output sequences can be read from a user supplied file or
        generated by a logic simulator.

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

*****************************************************************************

REFERENCE:

   Sunil R. Das, Mansour H. Assaf and Amiya R. Nayak, "Selecting
   Outputs for Merger in Space Compression using Concepts of Hamming
   Distance and Sequence Weights" Presented at the IASTED Int. Conf. on
   Modelling, Simulation, and Optimization, Gold Coast, Australia, May 6-9,
   1996, (Conference Proceedings, pp. 1-9).

*****************************************************************************

------------------ Installation Procedure ----------------------------

    To install fsim, follow the procedures described below.
```

1. To install fsim, make a bin directory under your home
   directory. Suppose that the home directory is ~/ and
   the compressed file HTesting.tar.Z is under the directory
   ~/HTesting.

2. Go to the directory HTesting by typing
   cd ~/HTesting.

3. To uncompress HTesting.tar.Z, first type
   uncompress HTesting.tar.Z, as a result a
   tar file (HTesting.tar) will be created;
   then type tar -xvf HTesting.tar; it will
   give the source code of all the compressed files.

4. To compile HTesting, in the directory
   HTesting, type "make";
   as a result, an executable file "Hamming" will
   be created.

5. Copy "Hamming" to the directory ~/bin by
   typing cp Hamming ~/bin.

---------- User's Guide for HTesting ----------

NAME: HTesting --------> Program that constructs the compaction tree
for combinational circuits.

SYNOPSIS: Hamming

INPUTS: Sequence values can be read from a user supplied file
"out_seq.dat" or be entered manually.

OUTPUTS: The summary of the compaction tree program is by default
reported to the standard output and to an output file
"compr_resl.dat". Results reported to the screen are
more detailled.

EXAMPLES: To run the program just type "Hamming" at the prompt.

Suppose the line numberings are read from a file
"out_line_numb.dat".

```
The first primary output line has got the number      0
The second primary output line has got the number     1
The third primary output line has got the number      2
The fourth primary output line has got the number     3
```

Moreover, suppose the sequence values are read from an input
file "out_seq.dat".

```
4 5
0 1 1 0
0 0 1 0
1 1 1 0
```

```
1 1 1 0
0 0 1 0
```

where 4 represents the number of output sequences and 5 is
the number of bits in each output sequence.
Input file in the following format would mean:

```
Output sequence #0 has got the following bit-values: 0 0 1 1 0
Output sequence #1 has got the following bit-values: 1 0 1 1 0
Output sequence #2 has got the following bit-values: 1 1 1 1 1
Output sequence #3 has got the following bit-values: 0 0 0 0 0
```

The main menu will look like:

```
Program Name : Hamming Testing

Language     : 'C'


MENU

0 : Read sequences manually.

1 : Read sequences from a file.

2 : Creating the compaction tree assuming equal probability
    but stochastically independent values for single and
    double line errors.

3 : Executing the modified algorithm assuming equal
    probability but stochastically independent values for
    single and double line errors.

4 : Constructing the (stochastically dependent) compaction
    tree assuming different probability values for single
    double line errors.

5 : Help file.

6 : Terminate Program.

        Please select your preferences....
```

You select your preferences by choosing first either 0 or 1 to read
the input data and then you choose 2 to create the compaction
tree assuming error occurrences to be independent or 3 to execute the
modified compaction tree, 4 to construct
the compaction tree when the error occurrences are assumed
stochastically dependent, 5 to call a help file or you
might terminate your program by selecting 6.

The results of the program will be reported to the screen and

```
to an output file "compr_resl.dat".

********************* Important Note **********************
*                                                         *
*    The output file "compr_resl.dat" should be erased    *
*    each time you run the program!                       *
*                                                         *
***********************************************************


The resulting output of the program reported to the screen would
look like:

       OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING

       TEST COMPACTION TECHNIQUE FOR BUILT-IN SELF-TEST
                IN VLSI CIRCUITS

        Written by     : Mansour H. Assaf

         Supervised by : Dr. S. R. Das

           Date  : April 25, 1996


The output sequences are as follows:

00110
10110
11111
00000
*****************************************************
STAGE:#0
*****************************************************
Output sequences #0 and #3 are merged with an OR gate into #4.
The new output sequences are:

10110
11111
00110
*****************************************************
STAGE:#1
*****************************************************
Output sequences #1 and #2 are merged with an AND gate into #5.
The new output sequences are:

00110
10110
*****************************************************
STAGE:#2
*****************************************************
Output sequences #4 and #5 are merged with an EXOR gate into #6.
The new output sequences are:

10000
*****************************************************
   CPU time
   Initialization                          : 0.0001 secs.
```

```
Simulation                                : 0.0002 secs.
Total                                     : 0.0003 secs.
```

The resulting output of the program reported to the output file "compr_resl.dat" would look like:

```
Output sequences #0 and #3 are merged with an OR gate into #4.
Output sequences #1 and #2 are merged with an AND gate into #5.
Output sequences #4 and #5 are merged with an EXOR gate into #6.
```

```
# makefile for the compaction tree program
# hamming.c print.c read.c help.c are source files for processing
# They use definitions in the header file defin.h

CFLAGS=-O
GDATA=defin.h
SRCS=hamming.c print.c read.c help.c
OBJS=hamming.o print.o read.o help.o

Hamming: $(OBJS) $(GDATA)
                gcc $(CFLAGS)$(OBJS) -o Hamming

hamming.o: hamming.c $(GDATA)
                gcc $(CFLAGS) -c hamming.c

print.o: print.c $(GDATA)
                gcc $(CFLAGS) -c print.c
read.o: read.c $(GDATA)
                gcc $(CFLAGS) -c read.c
help.o: help.c $(GDATA)
                gcc $(CFLAGS) -c help.c

lint:
        lint $(SRCS)

clear:
        rm $(OBJS)
```

```
/*************************************************************************


        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written permission
        of Dr. Das or Mr. Assaf.

        For detailed information, please contact:

        Dr. Sunil R. Das
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*************************************************************************/
/***************************** HISTORY*********************************

        HTesting: Version 1.0

        Original: M. H. Assaf, 15/04/1996

*************************************************************************/
/*-----------------------------------------------------------------------
       defin.h
       HTesting performs output compression for combinational circuits.
       Output sequences can be read from a user supplied file or
       generated by a logic simulator.

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

/*****************************************************************
*                                                                *
*              Defin.h                                           *
*              Defines global data structure for Htesting.       *
*                                                                *
*****************************************************************/


#define MAX 500                 /*maximum number of primary outputs*/
#define TRUE 1
#define FALSE 0

/*******GLOBAL VARIABLES*********/

int Length;                     /*sequence length*/
int weight1[MAX];               /*number of 1s in a sequence*/
int weight0[MAX];               /*number of 0s in a sequence*/
```

```
int stage;                      /*stage at which compression is performed*/
int out_number;                 /*number of primary outputs*/
int stage_number;               /*number of stages*/
int WEIGHT;                     /*temporary flag holding the highest weight valu
e*/
int new_line;                   /*line number created after compression*/
int out_seq[MAX][MAX];          /*storage for output sequences*/
int compress_out[MAX][MAX];     /*storage for compressed sequences*/
int hamming_dis;                /*hamming distance value*/
int weightp1;                   /*weight of 1s value*/
int weightp0;                   /*weight of 0s value*/
int discard[MAX];               /*sequence number to be discarded*/
int Weightp1[MAX][MAX];         /*the evaluation of the number of 1s*/
int Weightp0[MAX][MAX];         /*the evaluation of the number of 0s*/
int Hamming_dis[MAX][MAX];      /*the number of bits that differ*/
int init_values[MAX];           /*initial line number*/
int R1and;                      /*prob. of single line error*/
int R2and;                      /*prob. of double line error*/
int R1or;                       /*prob. of single line error*/
int R2or;                       /*prob. of double line error*/
int R1xor;                      /*prob. of single line error*/
int R2xor;                      /*prob. of double line error*/
int count;                      /*folows lines' numbering*/

float E_AND[MAX][MAX];          /*error estimate using AND gate*/
float E_OR[MAX][MAX];           /*error estimate using OR gate*/
float E_EXOR[MAX][MAX];         /*error estimate using XOR gate*/
float E_MAX;                    /*maximum error estimate*/
float EPSILON_AND[MAX][MAX];    /*detectable error probability for AND*/
float EPSILON_OR[MAX][MAX];     /*detectable error probability for OR*/
float EPSILON_XOR[MAX][MAX];    /*detectable error probability for XOR*/
float EPSILON_MAX;              /*maximum detectable error probability*/
float S1;                       /*prob. of single error effect*/
float S2;                       /*prob. of double eeror effect*/

char G;                         /*chosen gate for merger*/
char g;                         /*selected logic gate*/
char G_check;                   /*checking to selected type of gate*/

FILE *in, *out;                 /*input/output file pointers*/
```

```
/*************************************************************************



        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written permission
        of Dr. Das or Mr. Assaf.

        For detailed information, please contact:

        Dr. Sunil R. Das
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*************************************************************************/

/***************************** HISTORY ***********************************

        HTesting: Version 1.0

        Original: M. H. Assaf, 15/04/1996

*************************************************************************/
/*-----------------------------------------------------------------------
        hamming.c
        HTesting performs output compression for combinational circuits.
        Output sequences can be read from a user supplied file or
        generated by a logic simulator.

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

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
#include <string.h>
#include "defin.h"


extern void header ();
extern int menu();
extern void read_data_file();
extern void read_data_man();
extern void help();
extern void read_S();
```

```
float minutes,seconds,inittime,runtime,simtime;

/*********************************************************************
 *                                                                   *
 *              FUNCTION THAT INITIALIZES ALL VARIABLES              *
 *                                                                   *
 *********************************************************************/

init()
{
int I,J;
for(I=0; I<MAX; I++)
        {
        weight1[I] = 0;
        weight0[I] = 0;
        }       /*end of the for loop*/
for (I=0; I<MAX; I++)
        for(J=0; J<MAX; J++)
        {
        E_AND[I][J] = 0.0;
        E_OR[I][J] = 0.0;
        E_EXOR[I][J] = 0.0;
        EPSILON_AND[I][J] = 0.0;
        EPSILON_OR[I][J] = 0.0;
        EPSILON_XOR[I][J] = 0.0;
        Weightp1[I][J] = 0;
        Weightp0[I][J] = 0;
        Hamming_dis[I][J] = 0;
        }       /*end of the for loop*/

E_MAX = 0.0;
weightp1 = 0;
weightp0 = 0;
hamming_dis = 0;
WEIGHT = 0;
EPSILON_MAX = 0.0;
R1and = 0;
R2and = 0;
R1or = 0;
R2or = 0;
R1xor = 0;
R2xor = 0;
}

/*****************************************************************************
 *                                                                           *
 *      FUNCTION THAT CONSTRUCTS THE MODIFIED COMPACTION TREE                *
 *                                                                           *
 *****************************************************************************/

mod_compact ()
{
int i,j;

stage_number=out_number;

          header ();
```

```
for(i=0; i<MAX; i++)
{
discard[i] = 0;
}       /*end of the for loop*/
stage=0;
new_line = init_values[out_number-1];
while(stage < stage_number-1)
        {
                init();
                combinations();
                gate_selection();

        stage++;
         }      /*end of the while loop*/
}


/*****************************************************************
 *                                                               *
 *              FUNCTION THAT GETS 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;
}

/*****************************************************************************
 *                                                                           *
 *              FUNCTIONS THAT SELECTS GATE ACCORDING TO THE                 *
 *                      MERGEABILITY CRITERIA                                *
 *                                                                           *
 *****************************************************************************/

gate_selection()
{
int I,J,K,P;
int M,N,O,L,tmp1 = 0,tmp2 = 0;
int last_line = out_number-1;
M=0,N=0,g='a';

for(L=0; L<out_number-1; L++)
```

```
        {
        O = L+1;
        while(O<out_number+1)
                {
                if(E_MAX<E_AND[L][O])
                        {
                        E_MAX=E_AND[L][O];
                        M=L;
                        N=O;
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        M=M;
                        N=N;
                        }
                if(E_MAX<E_OR[L][O])
                        {
                        E_MAX=E_OR[L][O];
                        M=L;
                        N=O;
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        M=M;
                        N=N;
                        }
                if(E_MAX<E_EXOR[L][O])
                        {
                        E_MAX=E_EXOR[L][O];
                        M=L;
                        N=O;
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        M=M;
                        N=N;
                        }
                O++;
                }       /*end of the while loop*/
        }       /*end of the for loop*/

if ((1.0*Weightp1[M][N]) > ((Length*1.0)/2.0))
        {
        M=M;
        N=N;
        g='0';
        }
else if ((1.0*Weightp0[M][N]) > ((Length*1.0)/2.0))
        {
        M=M;
        N=N;
        g='1';
        }
else
        {
        M=M;
        N=N;
```

```
        g='2';
        }

        switch(g)
                {
                case'0': {
                         for(J=0; J<Length; J++)
                                 {
                        out_seq[out_number][J]=out_seq[M][J] && out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
                        last_line++;
                        init_values[last_line] = new_line;
in = fopen ("compr_resl.dat","a");
        if(in)
        {
for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
        else
        M = M;
        }
for (P=0; P<MAX; P++)
        {
if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
fprintf(in,"and 0 2 %d %d\n",tmp1,tmp2);        /*end of if test*/
        else
fprintf(in,"and 1 2 %d %d\n",tmp1,tmp2);
/*fprintf(in,"%d = AND(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
   fclose(in);
                        out_number++;
                         }
                         break;
                case'1': {
                         for(J=0; J<Length; J++)
                                 {
                        out_seq[out_number][J]=out_seq[M][J]||out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
```

```
                        last_line++;
                        init_values[last_line] = new_line;
 in = fopen ("compr_resl.dat","a");
        if(in)
        {
 for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
        else
        M = M;
        }
 for (P=0; P<MAX; P++)
        {
 if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
 fprintf(in,"or 0 2 %d %d\n",tmp1,tmp2);          /*end of if test*/
        else
 fprintf(in,"or 1 2 %d %d\n",tmp1,tmp2);
 /*fprintf(in,"%d = OR(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
    fclose(in);
                        out_number++;
                         }
                         break;
                case'2': {
                         for(J=0; J<Length; J++)
                                 {
                        out_seq[out_number][J]=out_seq[M][J]^out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
                        last_line++;
                        init_values[last_line] = new_line;
in = fopen ("compr_resl.dat","a");
        if(in)
        {
for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
```

```
        else
        M = M;
        }
for (P=0; P<MAX; P++)
        {
if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
fprintf(in,"xor 0 2 %d %d\n",tmp1,tmp2);        /*end of if test*/
        else
fprintf(in,"xor 1 2 %d %d\n",tmp1,tmp2);
/*fprintf(in,"%d = XOR(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_res1.dat FILE\n");
   fclose(in);
                        out_number++;
                         }
                         break;

                default: printf("NO CHOSEN GATE\n");
                     }          /*end of the switch test*/

}



/*************************************************************************
 *                                                                       *
 *             Main loop for the logic circuit compressor                *
 *             1. Read primary output sequences                          *
 *             2. Find Hs, W'1, W'0 for each pair of sequences           *
 *             3. Find the best pair of sequences to be merged           *
 *             4. Update output sequences and discard the used pair      *
 *                                                                       *
 *************************************************************************/

main()
{
int i,j,rt;
char ch='y';

init();         /*initialize all variables*/
count = 0;
S1 = 0.0;
S2 = 0.0;
for (i=0; i<MAX; i++)
{
init_values[i] = 0;
}
```

```
while( ch != 'q')
   {
  rt = menu();
  switch (rt)
        {
        case 0 :
                read_data_man();        /*read output sequences from a user*/
                break;

        case 1:
                read_data_file();       /*read output sequences from a file*/
                break;

        case 2:
                compact();              /*performs space compression*/
                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",runtime);
                break;

        case 3:
                mod_compact();
                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",runtime);
                break;

        case 4:
                read_S();
                prob_compact();
                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",runtime);
                break;

        case 5:
                help();
                break;

        case 6:
                printf(" No selection/Program aborted.\n");
                ch = 'q';

        }       /*end of the switch test*/
```

```
    }    /*end of the while loop*/

}

/**********************END OF MAIN***********************************/

/*****************************************************************************
 *                                                                           *
 *              FONCTION THAT CONSTRUCTS THE COMPACTION TREE                 *
 *                                                                           *
 *****************************************************************************/

compact()
{
int i,j;

stage_number=out_number;


           header ();

printf("The output sequences are as follows:\n");
for(i=0; i<out_number; i++)
        {
        printf("\n");
        for (j=0; j<Length; j++)
                {
                printf("%d",out_seq[i][j]);
                }       /*end of the for loop*/
        }       /*end of the for loop*/
        printf("\n");
for(i=0; i<MAX; i++)
{
discard[i] = 0;
}       /*end of the for loop*/
stage=0;
new_line=out_number-1;
while(stage < stage_number-1)
        {
/*out = fopen ("compr_resl.dat","w");
        if(out)
        {
        while(!feof(out))
          {
        */printf("*********************************************************\n");
        printf("STAGE:#%d\n",stage);
        printf("*********************************************************\n");
                init();
                combinations();
                choose_gate();
                printf("The new output sequences are:\n");
                        for(i=0; i<out_number; i++)
                           if(out_seq[i][0] != 9)
                           {
                                {
                                printf("\n");
                                        for(j=0; j<Length; j++)
                                                {
                                                printf("%d",out_seq[i][j]);
                                                }       /*end of the for loop*/
```

```
                                }
                        }       /*end of the conditional if*/
                printf("\n");
        stage++;
         /*   }
        }
        else
                printf("ERROR OPENING out_seq.dat FILE\n");
   fclose(out);
        */}     /*end of the while loop*/
        printf("************************************************************\n");

}

/*****************************************************************************
 *                                                                           *
 *      FUNCTION THAT ANALYZES DIFFERENT OUTPUT SEQUENCES                    *
 *                                                                           *
 *****************************************************************************/

combinations()
{
int i,j,k,l,r,p,m,n,o;

i=0;
k=0;
while (i != out_number-1)
        {
        if((discard[i] != 1))
        {
        k=i+1;
                while(k!=out_number)
                        {
                        if((discard[k] != 1))
                        {
                        for (j=0; j<Length; j++)
                                {
                                compress_out[0][j]=out_seq[i][j];
                                }       /*end of the for loop*/
                                for(j=0; j<Length; j++)
                                        {
                                        compress_out[1][j]=out_seq[k][j];
                                        }       /*end of the for loop*/
                                hamming_dis=0;
                                weightp1=0;
                                weightp0=0;
                                derived(i,k);
                         }      /*end of the conditional if*/
                         else
                                discard[k] = discard[k];
                                k++;
                        }       /*end of the while loop*/
        }       /*end of the conditional if*/
        else
                discard[i] = discard[i];
                i++;
        }       /*end of the while loop*/
}
```

```
/*************************************************************************
 *                                                                       *
 *              FUNCTION THAT FINDS THE DERIVED SEQUENCE                 *
 *              AND CALCULATES ERROR ESTIMATES                           *
 *                                                                       *
 *************************************************************************/

derived(int r, int p)
{
int m,n;

for(n=0; n<Length; n++)
        {
        if(compress_out[0][n]!=compress_out[1][n])
                {
                hamming_dis++;
                compress_out[2][n]=2;
                }
        else
                {
                hamming_dis=hamming_dis;
                compress_out[2][n]=compress_out[1][n];
                }
        }       /*end of the for loop*/

for(n=0; n<Length; n++)
        {
        if(compress_out[2][n]==1)
                weightp1++;
        else if(compress_out[2][n]==0)
                weightp0++;
        else
                {
                weightp1=weightp1;
                weightp0=weightp0;
                }
        }       /*end of the for loop*/

                E_EXOR[r][p]=2*Length*1.0;
                E_OR[r][p]=hamming_dis+(3*weightp0)+(weightp1*1.0);
                E_AND[r][p]=hamming_dis+(3*weightp1)+(weightp0*1.0);

                Weightp1[r][p]=weightp1;
                Weightp0[r][p]=weightp0;
}

/*************************************************************************
 *                                                                       *
 *              FUNCTION THAT CHOOSES THE BEST GATE FOR MERGING          *
 *                                                                       *
 *************************************************************************/

choose_gate()
{
int m,n,l,o,i,j;

m=0;
n=0;
```

```
G='a';

for(l=0; l<out_number-1; l++)
        {
        o = l+1;
        while(o<out_number+1)
                {
                if(E_MAX<E_AND[l][o])
                        {
                        E_MAX=E_AND[l][o];
                        m=l; n=o; G='0';
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        m=m; n=n; G=G;
                        }
                if(E_MAX<E_OR[l][o])
                        {
                        E_MAX=E_OR[l][o];
                        m=l; n=o; G='1';
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        m=m; n=n; G=G;
                        }
                if(E_MAX<E_EXOR[l][o])
                        {
                        E_MAX=E_EXOR[l][o];
                        m=l; n=o; G='2';
                        }       /*end of the if test*/
                else
                        {
                        E_MAX=E_MAX;
                        m=m; n=n; G=G;
                        }
                o++;
                }       /*end of the while loop*/
        }       /*end of the for loop*/


        switch(G)
                {
                case'0': {
                         for(j=0; j<Length; j++)
                                {
                        out_seq[out_number][j]=out_seq[m][j]&&out_seq[n][j];
                                out_seq[m][j]=9;
                                out_seq[n][j]=9;
                                discard[m] = 1;
                                discard[n] = 1;
                                }       /*end of the for loop*/
                        new_line++;
printf("Output sequences #%d and #%d are merged with an AND gate into #%d.\n",m,
n,new_line);
out = fopen ("compr_res1.dat","a");
        if(out)
        {
fprintf(out,"Output sequences #%d and #%d are merged with an AND gate into #%d.\
```

```
n",m,n,new_line);
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
   fclose(out);
                        out_number++;
                         }
                         break;
                case'1': {
                         for(j=0; j<Length; j++)
                                 {
                        out_seq[out_number][j]=out_seq[m][j]||out_seq[n][j];
                                out_seq[n][j]=9;
                                out_seq[m][j]=9;
                                discard[m] = 1;
                                discard[n] = 1;
                                }       /*end of the for loop*/
                        new_line++;
printf("Output sequences #%d and #%d are merged with an OR gate into #%d.\n",m,n
,new_line);
out = fopen ("compr_resl.dat","a");
        if(out)
        {
fprintf(out,"Output sequences #%d and #%d are merged with an OR gate into #%d.\n
",m,n,new_line);
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
   fclose(out);
                        out_number++;
                         }
                         break;
                case'2': {
                         for(j=0; j<Length; j++)
                                 {
                        out_seq[out_number][j]=out_seq[m][j]^out_seq[n][j];
                                out_seq[m][j]=9;
                                out_seq[n][j]=9;
                                discard[m] = 1;
                                discard[n] = 1;
                                }       /*end of the for loop*/
                        new_line++;
printf("Output sequences #%d and #%d are merged with an EXOR gate into #%d.\n",m
,n,new_line);
out = fopen ("compr_resl.dat","a");
        if(out)
        {
fprintf(out,"Output sequences #%d and #%d are merged with an EXOR gate into #%d.
\n",m,n,new_line);
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
   fclose(out);
                        out_number++;
                         }
                         break;
                default: printf("NO CHOSEN GATE\n");
                     }  /*end of the while loop*/

}
```

```
/*************************************************************************
 *                                                                       *
 *      FUNCTION THAT CONSTRUCTS THE PROBABILISTIC COMPACTION TREE       *
 *                                                                       *
 *************************************************************************/

prob_compact ()
{
int i,j;

stage_number=out_number;

           header ();

for(i=0; i<MAX; i++)
{
discard[i] = 0;
}       /*end of the for loop*/
stage = 0;
new_line = init_values[out_number-1];
while(stage < stage_number-1)
        {
                init();
                prob_combinations();
                prob_gate_selection();

        stage++;
         }      /*end of the while loop*/
}

/*************************************************************************
 *                                                                       *
 *      FUNCTION THAT ANALYZES DIFFERENT OUTPUT SEQUENCES                *
 *                                                                       *
 *************************************************************************/

prob_combinations()
{
int i,j,k,l,r,p,m,n,o;

i=0;
k=0;
while (i != out_number-1)
        {
        if((discard[i] != 1))
        {
        k=i+1;
                while(k!=out_number)
                        {
                        if((discard[k] != 1))
                        {
                        for (j=0; j<Length; j++)
                                {
                                compress_out[0][j]=out_seq[i][j];
                                }       /*end of the for loop*/
                                for(j=0; j<Length; j++)
                                        {
```

```
                                        compress_out[1][j]=out_seq[k][j];
                                        }       /*end of the for loop*/
                                hamming_dis=0;
                                weightp1=0;
                                weightp0=0;
                                prob_derived(i,k);
                        }       /*end of the conditional if*/
                        else
                                discard[k] = discard[k];
                                k++;
                        }       /*end of the while loop*/
        }       /*end of the conditional if*/
        else
                discard[i] = discard[i];
                i++;
        }       /*end of the while loop*/
}

/*************************************************************************
 *                                                                       *
 *              FUNCTION THAT FINDS THE DERIVED SEQUENCE                 *
 *              AND CALCULATES ERROR ESTIMATES                           *
 *                                                                       *
 *************************************************************************/

prob_derived(int r, int p)
{
int m,n;

for(n=0; n<Length; n++)
        {
        if(compress_out[0][n]!=compress_out[1][n])
                {
                hamming_dis++;
                compress_out[2][n]=2;
                }
        else
                {
                hamming_dis=hamming_dis;
                compress_out[2][n]=compress_out[1][n];
                }
        }       /*end of the for loop*/

for(n=0; n<Length; n++)
        {
        if(compress_out[2][n]==1)
                weightp1++;
        else if(compress_out[2][n]==0)
                weightp0++;
        else
                {
                weightp1=weightp1;
                weightp0=weightp0;
                }
        }       /*end of the for loop*/

                R1and = (2*weightp1)+(0*weightp0)+(1*hamming_dis);
                R2and = (1*weightp1)+(1*weightp0)+(0*hamming_dis);
                R1or  = (0*weightp1)+(2*weightp0)+(1*hamming_dis);
                R2or  = (1*weightp1)+(1*weightp0)+(0*hamming_dis);
```

```
                R1xor = (2*weightp1)+(2*weightp0)+(2*hamming_dis);
                R2xor = (0*weightp1)+(0*weightp0)+(0*hamming_dis);
/*printf("R1and=%d R2and=%d \n",R1and,R2and);
printf("R1or=%d R2or=%d \n",R1or,R2or);
printf("R1xor=%d R2xor=%d \n",R1xor,R2xor);
printf("S1=%f S2=%f Length=%d\n",S1,S2,Length);*/
EPSILON_AND[r][p] = ((S1*((1.0*R1and)/(2.0*Length)))+(S2*((1.0*R2and)/(1.0*Lengt
h))));
EPSILON_OR[r][p]  = ((S1*((1.0*R1or) /(2.0*Length)))+(S2*((1.0*R2or) /(1.0*Lengt
h))));
EPSILON_XOR[r][p] = ((S1*((1.0*R1xor)/(2.0*Length)))+(S2*((1.0*R2xor)/(1.0*Lengt
h))));
/*printf("EPSILON_AND[%d][%d]=%f\n",r,p,EPSILON_AND[r][p]);
printf("EPSILON_OR [%d][%d]=%f\n",r,p,EPSILON_OR [r][p]);
printf("EPSILON_XOR[%d][%d]=%f\n",r,p,EPSILON_XOR[r][p]);*/
                Weightp1[r][p] = weightp1;
                Weightp0[r][p] = weightp0;
                Hamming_dis[r][p] = hamming_dis;
/*printf("Weightp1[%d][%d] = %d\n",r,p,Weightp1[r][p]);
printf("Weightp0[%d][%d] = %d\n",r,p,Weightp0[r][p]);
printf("Hamming_dis[%d][%d] = %d\n",r,p,Hamming_dis[r][p]);*/

}


/*************************************************************************
 *                                                                       *
 *              FUNCTIONS THAT SELECTS GATE ACCORDING TO THE             *
 *                      MERGEABILITY CRITERIA                            *
 *                                                                       *
 *************************************************************************/

prob_gate_selection()
{
int I,J,K,P,tmp1 = 0,tmp2 = 0;
int M,N,O,L;
int last_line = out_number-1;
M=0,N=0,g='a';
for(L=0; L<out_number-1; L++)
        {
        O = L+1;
        while(O<out_number+1)
                {
                if(EPSILON_MAX<EPSILON_AND[L][O])
                        {
                        EPSILON_MAX=EPSILON_AND[L][O];
                        M=L;
                        N=O;
                        }       /*end of the if test*/
                else
                        {
                        EPSILON_MAX=EPSILON_MAX;
                        M=M;
                        N=N;
                        }
                if(EPSILON_MAX<EPSILON_OR[L][O])
                        {
                        EPSILON_MAX=EPSILON_OR[L][O];
                        M=L;
                        N=O;
```

```
                        }       /*end of the if test*/
                else
                        {
                        EPSILON_MAX=EPSILON_MAX;
                        M=M;
                        N=N;
                        }
                if(EPSILON_MAX<EPSILON_XOR[L][O])
                        {
                        EPSILON_MAX=EPSILON_XOR[L][O];
                        M=L;
                        N=O;
                        }       /*end of the if test*/
                else
                        {
                        EPSILON_MAX=EPSILON_MAX;
                        M=M;
                        N=N;
                        }
                O++;
                }       /*end of the while loop*/
        }       /*end of the for loop*/


if ((1.0*S1)>((2.0*(Weightp0[M][N]+Weightp1[M][N]))/(Hamming_dis[M][N]+(4.0*Weig
htp0[M][N])+(2.0*Weightp1[M][N]))))
   {
if ((1.0*S1)>((2.0*(Weightp0[M][N]+Weightp1[M][N]))/(Hamming_dis[M][N]+(4.0*Weig
htp1[M][N])+(2.0*Weightp0[M][N]))))
        {
        M=M;
        N=N;
        g='2';
        }
else
        {
        M=M;
        N=N;
        g='1';
        }
   }
else if ((1.0*S1)<((2.0*(Weightp0[M][N]+Weightp1[M][N]))/(Hamming_dis[M][N]+(4.0
*Weightp0[M][N])+(2.0*Weightp1[M][N]))))
   {
if ((1.0*S1)>((2.0*(Weightp0[M][N]+Weightp1[M][N]))/(Hamming_dis[M][N]+(4.0*Weig
htp1[M][N])+(2.0*Weightp0[M][N]))))
        {
       M=M;
        N=N;
        g='0';
        }
else
   {
        if (Weightp0[M][N]>Weightp1[M][N])
        {
        M=M;
        N=N;
        g='1';
        }
      else
```

```
        {
        M=M;
        N=N;
        g='0';
        }
    }
  }
else
        {
        printf("Bad selection of gates; check your files\n");
        }
                /*end of the if test*/

/*printf("M=%d N=%d\n",M,N);*/

        switch(g)
                {
                case'0': {
                         for(J=0; J<Length; J++)
                                {
                        out_seq[out_number][J]=out_seq[M][J] && out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
                        last_line++;
                        init_values[last_line] = new_line;
in = fopen ("compr_res1.dat","a");
        if(in)
        {
for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
        else
        M = M;
        }
for (P=0; P<MAX; P++)
        {
if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
fprintf(in,"and 0 2 %d %d\n",tmp1,tmp2);        /*end of if test*/
        else
fprintf(in,"and 1 2 %d %d\n",tmp1,tmp2);
/*fprintf(in,"%d = AND(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_res1.dat FILE\n");
```

```
   fclose(in);
                        out_number++;
                         }
                         break;
                case'1': {
                         for(J=0; J<Length; J++)
                                {
                        out_seq[out_number][J]=out_seq[M][J]||out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
                        last_line++;
                        init_values[last_line] = new_line;
in = fopen ("compr_resl.dat","a");
        if(in)
        {
for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
        else
        M = M;
        }
for (P=0; P<MAX; P++)
        {
if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
fprintf(in,"or 0 2 %d %d\n",tmp1,tmp2);         /*end of if test*/
        else
fprintf(in,"or 1 2 %d %d\n",tmp1,tmp2);
/*fprintf(in,"%d = OR(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_resl.dat FILE\n");
   fclose(in);
                        out_number++;
                         }
                         break;
                case'2': {
                         for(J=0; J<Length; J++)
                                {
                        out_seq[out_number][J]=out_seq[M][J]^out_seq[N][J];
                                discard[M] = 1;
                                discard[N] = 1;
                                }       /*end of the for loop*/
                        new_line++;
                        last_line++;
```

```
                        init_values[last_line] = new_line;
in = fopen ("compr_res1.dat","a");
        if(in)
        {
for (K=0; K<MAX; K++)
        {
        if (M == K)
        {
        /*printf("M = %d\n",M);*/
        tmp1 = init_values[K];
        /*printf("tmp1 = %d\n",tmp1);*/
        }
        else
        M = M;
        }
for (P=0; P<MAX; P++)
        {
if (N == P)
        {
        /*printf("N = %d\n",N);*/
        tmp2 = init_values[P];
        /*printf("tmp2 = %d\n",tmp2);*/
        }
        else
        N = N;
        }
        if (stage == stage_number-2)
fprintf(in,"xor 0 2 %d %d\n",tmp1,tmp2);        /*end of if test*/
        else
fprintf(in,"xor 1 2 %d %d\n",tmp1,tmp2);
/*fprintf(in,"%d = XOR(%d,%d)\n",new_line,tmp1,tmp2);*/
        }       /*end of the if test*/
        else
                printf("ERROR OPENING compr_res1.dat FILE\n");
  fclose(in);
                        out_number++;
                        }
                        break;

                default: printf("NO CHOSEN GATE\n");
                        }           /*end of the switch test*/
        /*for(P=0; P<MAX; P++)
        {
        printf("init_values[%d]=%d\n",P,init_values[P]);
        }*/
}
```

```
/*************************************************************************



        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written permission
        of Dr. Das or Mr. Assaf.

        For detailed information, please contact:

        Dr. Sunil R. Das
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*************************************************************************/

/***************************** HISTORY ***********************************

        HTesting: Version 1.0

        Original: M. H. Assaf  15/04/1996

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

        help.c
        Provides an on-line help manual.

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


#include <stdio.h>
#include <stdlib.h>
#include "defin.h"


/*****************************************************************
 *                                                               *
 *              FUNCTION THAT READS A HELP FILE                  *
 *                                                               *
 *****************************************************************/

help()
{
char ch;
```

```
printf("------------------ User's Guide for HTesting -------------------------\n
");
printf("\n");
printf("\n");
printf("NAME:     HTesting -------> program that constructs the compaction tree
\n");
printf("                              for combinational circuits.  \n");
printf("\n");
printf("\n");
printf("SYNOPSIS: Hamming \n");
printf("\n");
printf("\n");
printf("INPUTS:   Sequence values can be read from a user supplied file\n");
printf("          'out_seq.dat' or be entered manually.  \n");
printf("\n");
printf("\n");
printf("OUTPUTS:  The summary of the compaction tree program is by default \n");
printf("          reported to the standard output and to an output file \n");
printf("          'compr_resl.dat'. Results reported to the screen are \n");
printf("          more detailled.\n");
printf("\n");
printf("\n");
printf("EXAMPLES: To run the program just type 'Hamming' at the prompt. \n");
printf("\n");
printf("          Suppose that lines numbering are read from a file \n");
printf("          'out_line_numb.dat'.\n");
printf("\n");
printf("          The first primary output line has got the number      0.\n");
printf("          The second primary output line has got the number     1.\n");
printf("          The third primary output line has got the number      2.\n");
printf("          The fourth primary output line has got the number     3.\n");
printf("\n");
printf("          moreover, suppose that sequence values are read from an input
file\n");
printf("          'out_seq.dat'.\n");
printf("                                4 5\n");
printf("                                0 1 1 0\n");
printf("                                0 0 1 0\n");
printf("                                1 1 1 0\n");
printf("                                1 1 1 0\n");
printf("                                0 0 1 0\n");
printf("\n");
printf("         where 4 represents the number of output sequences and 5 is\n");
printf("          the number of bits of each output sequence.\n");
printf("          Input file in the following format would mean:\n");
printf("\n");
printf("Output sequence #0 has got the following bit-values: 0 0 1 1 0\n");
printf("  Output sequence #1 has got the following bit-values: 1 0 1 1 0\n");
printf("  Output sequence #2 has got the following bit-values: 1 1 1 1 1\n");
printf("  Output sequence #3 has got the following bit-values: 0 0 0 0 0\n");
printf("\n");
printf(" PRESS ANY KEY TO CONTINUE");
ch = getchar();
ch = getchar();
printf("\n");
printf("\n");
printf("          The main menu will look like:\n");
printf("\n");
printf("\n");
printf("                          Program Name : Hamming Testing\n");
```

```
printf("\n");
printf("                          Language      : 'C'\n");
printf("\n");
printf("\n");
printf("\n");
printf("\n");
printf("                          MENU\n");
printf("\n");
printf("\n");
printf("                                  0 :  Read sequences manually. \n");
printf("\n");
printf("                                  1 :  Read sequences from a file. \n");
printf("\n");
printf("                                  2 :  Creating the compaction tree assum
ing equal but stochastically independent probability \n");
printf("                                       values for single and double line
errors. \n");
printf("\n");
printf("                                  3 :  Executing the modified compaction
algorithm assuming equal \n");
printf("                                         but stochastically independent p
robability values for single and double line errors. \n");
printf("\n");
printf("                                  4 :  Constructing the (stochastically d
ependent) compaction tree assuming different \n");
printf("                                       probability values for single and
double line errors. \n");
printf("\n");
printf("                                  5 :  Help file. \n");
printf("\n");
printf("                                  6 :  Terminate Program. \n");
printf("\n");
printf("                                    Please select your preferences....\n"
);
printf("\n");
printf("\n");
printf("          You select your preferences by choosing first either 0 or 1 to
read\n");
printf("          the input data and then you choose 2 to create the compaction \
n");
printf("          tree assuming error occurrences to be stochastically independen
t or 3 to execute the modified compaction tree, 4 to construct\n");
printf("          the probabilistic compaction tree when the error occurrences ar
e assumed stochastically dependent,5 to call a help\n");
printf("          file or you might terminate your program by selecting 6.\n");
printf("\n");
printf("          The results of the program will be reported to the screen and\n
");
printf("          to an output file 'compr_resl.dat'.\n");
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
printf("\n");
printf("          ********************* Important Note **********************\n"
);
printf("          *                                                           *\n"
);
printf("          *    The output file 'compr_resl.dat' should be erased      *\n"
);
printf("          *    each time you run the program!                         *\n"
```

```
);
 printf("          *                                                    *\n"
 );
 printf("          ************************************************************\n"
 );
 printf("\n");
 printf("\n");
 printf("          The resulting output of the program reported to the screen woul
 d\n");
 printf("          look like: \n");
 printf("\n");
 printf("                OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING\n")
 ;
 printf("\n");
 printf("           TEST COMPACTION TECHNIQUE FOR BUILT-IN SELF-TEST\n");
 printf("                          OF VLSI CIRCUITS\n");
 printf("\n");
 printf("                 Written by     : Mansour H. Assaf\n");
 printf("\n");
 printf("                  Supervised by : Dr. S. R. Das\n");
 printf("\n");
 printf("                    Date  : April 25, 1996\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("          The output sequences are as follows:\n");
 printf("\n");
 printf("          00110\n");
 printf("          10110\n");
 printf("          11111\n");
 printf("          00000\n");
 printf("          ********************************************************\n");
 printf("          STAGE:#0\n");
 printf("          ********************************************************\n");
 printf("          Output sequences #0 and #3 are merged with an OR gate into #4.\
 n");
 printf("          The new output sequences are:\n");
 printf("\n");
 printf("PRESS ANY KEY TO CONTINUE");
 ch = getchar();
 printf("\n");
 printf("          10110\n");
 printf("          11111\n");
 printf("          00110\n");
 printf("          ********************************************************\n");
 printf("          STAGE:#1\n");
 printf("          ********************************************************\n");
 printf("          Output sequences #1 and #2 are merged with an AND gate into #5.
 \n");
 printf("          The new output sequences are:\n");
 printf("\n");
 printf("          00110\n");
 printf("          10110\n");
 printf("          ********************************************************\n");
 printf("          STAGE:#2\n");
 printf("          ********************************************************\n");
 printf("          Output sequences #4 and #5 are merged with an EXOR gate into #6
 .\n");
 printf("          The new output sequences are:\n");
 printf("\n");
```

```
printf("          10000\n");
printf("          ************************************************************\n");
printf("             CPU time\n");
printf("             Initialization                                : 0.0001 secs.\n")
;
printf("             Simulation                                    : 0.0002 secs.\n")
;
printf("             Total                                         : 0.0003 secs.\n")
;
printf("\n");
printf("\n");
printf("\n");
printf("\n");
printf("          The resulting outputs of the program reported to the output fil
e\n");
printf("          'compr_resl.dat' would look like:\n");
printf("\n");
printf("          Output sequences #0 and #3 are merged with an OR gate into #4.\
n");
printf("          Output sequences #1 and #2 are merged with an AND gate into #5.
\n");
printf("          Output sequences #4 and #5 are merged with an EXOR gate into #6
.\n");
printf("\n");
printf("         You could get the resulting outputs  reported to the output file
\n");
printf("         'compr_resl.dat' in FSIM and ATALANTA readble fromat:\n");
printf("\n");
printf("         4 = OR(0,3)\n");
printf("         5 = AND(1,2)\n");
printf("         6 = XOR(4,5)\n");
printf("\n");
printf("         You also could get the resulting outputs  reported to the output
\n");
printf("         file compr_resl.dat' in COMPACTEST readble fromat:\n");
printf("\n");
printf("         or 1 2 0 3\n");
printf("         and 1 2 1 2\n");
printf("         xor 0 2 4 5\n");
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
printf("\n");
}


/*****************************************************************
 *                                                               *
 *      FUNCTION THAT PROVIDES HELP FOR READING DATA             *
 *      FROM A FILE                                              *
 *                                                               *
 *****************************************************************/

help_read_data_file()
{
int answer;
char ch;
printf("Do you need help instructions? Answer 1 for 'yes' and 0 for 'no': ");
scanf("%d",&answer);
if (answer == 1)
```

```
{
printf("          Line number read from the file 'out_line_numb.dat'\n");
printf("          should have the following format:\n");
printf("                                    0\n");
printf("                                    1\n");
printf("                                    2\n");
printf("                                    3\n");
printf("\n");
printf("          Sequence values read from the input file\n");
printf("          'out_seq.dat' should have the following format:\n");
printf("                                    4 5\n");
printf("                                    0 1 1 0\n");
printf("                                    0 0 1 0\n");
printf("                                    1 1 1 0\n");
printf("                                    1 1 1 0\n");
printf("                                    0 0 1 0\n");
printf("\n");
printf("          where 4 represents the number of output sequences and 5 is\n")
;
printf("          the number of bits of each output sequence.\n");
printf("          Input file in the following format would mean:\n");
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
getchar();
printf("\n");
}       /*end of if test*/
else
{
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
getchar();
printf("\n");
}
}

/*help_data_format()
{
char ch;
printf("Bit stream values has to be binary either 0 or 1\n");
while ((out_seq[i][j] != 0) || (out_seq[i][j] != 1))
        {
printf("Bit value is not binary\n");
fscanf(in,"%d",&out_seq[i][j]);
        }
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
getchar();
printf("\n");
}
*/
```

```
/*************************************************************************



        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written permission
        of Dr. Das or Mr. Assaf.

        For detailed information, please contact:

        Dr. Sunil R. Das
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext No. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*************************************************************************/

/***************************** HISTORY ***********************************

        HTesting: Version 1.0

        Original: M. H. Assaf, 15/04/1996

*************************************************************************/
/*-----------------------------------------------------------------------
        print.c
        program that prints the space compression results
        into a file.
-----------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "defin.h"

void header()
{
puts ("\33[H\33[2J");
printf("OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING\n\n");
printf("TEST COMPACTION TECHNIQUE OF BUILT-IN SELF-TEST\n");
printf("            IN VLSI CIRCUITS\n\n");
printf(" Written by     : Mansour H. Assaf\n\n");
printf(" Supervised by  : Dr. S. R. Das\n\n");
printf("          Date  : April 25, 1996\n\n\n\n");
}
```

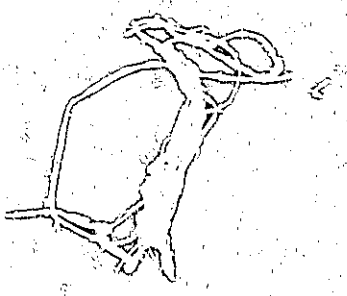
```
/*************************************************************************



        This program was written by Mansour H. Assaf under the
        supervision of Dr. Sunil R. Das, in the University
        of Ottawa, Department of Electrical Engineering, in 1996.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced or used
        for any commercial product or purpose without the written permission
        of Dr. Das or Mr. Assaf.

        For detailed information, please contact:

        Dr. Sunil R. Das.
        Department of Electrical Engineering
        University of Ottawa
        Ottawa, Ontario
        Canada K1N 6N5
        Phone No. (613) 562-5800 ext. 6216
        Fax:      (613) 562-5175
        E-Mail:   S.R.DAS@IEEE.ORG
*************************************************************************/

/***************************** HISTORY ***********************************

        HTesting: Version 1.0

        Original: M. H. Assaf, 15/04/1996

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

        read.c
        Reads sequence values supplied by a user or
        from an input file.

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


#include <stdio.h>
#include <stdlib.h>
#include "defin.h"

extern help_read_data_file();


/* ====================================================================
   FUNCTION : menu
   Display  menu
   ====================================================================*/

int menu( void)
```

```
{

int x,y,choice;
puts ("\33[H\33[2J");

printf(" Program Name : Hamming Testing\n\n");
printf(" Language     : 'C'\n\n");
printf(" \n\n\nMENU\n\n\n");


printf(" 0 :  Read sequences manually.   \n\n");
printf(" 1 :  Read sequences from a file.\n\n");
printf(" 2 :  Creating the compaction tree assuming equal\n");
printf("      but stochastically independent probability values\n");
printf("      for single and double line errors.\n\n");
printf(" 3 :  Executing the modified algorithm assuming equal\n");
printf("      but stochastically independent probability values\n");
printf("      for single and double line errors.\n\n");
printf(" 4 :  Constructing the (stochastically dependent) compaction\n");
printf("      tree assuming different probability values for single\n");
printf("      and double line errors.\n\n");
printf(" 5 :  Help file.\n\n");
printf(" 6 :  Terminate Program. \n\n");
printf(" Please select your preferences....\n\n");
scanf("%d",&choice);

return(choice);

}


/*********************************************************************
 *                                                                   *
 *              FUNCTION THAT READS DATA FROM A FILE                 *
 *                                                                   *
 *********************************************************************/

void read_data_file()
{
register i,j;
help_read_data_file();
in = fopen("out_seq.dat","r");
        if(in)
        {
        while(!feof(in))
                {
                /*fscanf(in,"%d\n",&count);
                if (count > 0)
                {*/
                fscanf(in,"%d\n",&out_number);
                if (out_number > 0)
                {
                fscanf(in,"%d\n",&Length);
                if (Length > 0)
                {
                for(j=0; j<Length; j++)
                        for(i=0; i<out_number; i++)
                                {
                                fscanf(in,"%d",&out_seq[i][j]);
```

```
                        if (out_seq[i][j] != 0)
                        if (out_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 for loop*/
                }       /*end of if test*/
                else
                {
                printf("Number of bit streams should be positive\n");
                printf("Check your data file\n");
                exit(1);
                }
                }       /*end of if test*/
                else
                {
                printf("Sequence length should be positive\n");
                printf("Check your data file\n");
                exit(1);
                }
                /*}*/   /*end of if test*/
                /*else
                {
                printf("The initial output test sequence should be positive\n");
                printf("Check your data file\n");
                exit(1);
                }*/
                }       /*end of while loop*/
        }       /*end of if test*/
        else
                printf("ERROR OPENING out_seq.dat FILE\n");
fclose(in);
read_init_values();
}

/*********************************************************************
 *                                                                   *
 *              FUNCTION THAT READS DATA MANUALLY                    *
 *                                                                   *
 *********************************************************************/

void read_data_man()
{
register i,j;
int answer,try;
char ch;
printf("Do you need help instructions? Answer 1 for 'yes' and 0 for 'no': ");
scanf("%d",&answer);
if (answer == 1)
{
printf("\n");
printf("          Sequence values entered manually by a user\n");
printf("          should have the following format\n");
printf("                                4 5\n");
printf("                                0 1 1 0\n");
printf("                                0 0 1 0\n");
printf("                                1 1 1 0\n");
printf("                                1 1 1 0\n");
```

```
printf("                                    0 0 1 0\n");
printf("\n");
printf("          where 4 represents the number of output sequences and 5 is\n")
;
printf("          the number of bits of each output sequence.\n");
printf("          Input file in the following format would mean:\n");
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
getchar();
printf("\n");
}       /*end of if test*/
else
{
printf("\n");
printf("PRESS ANY KEY TO CONTINUE");
ch = getchar();
getchar();
printf("\n");
}
                printf("Enter number of output sequences:  ");
                scanf("%d",&out_number);
                while (out_number <= 0)
                {
                printf("Number of bit streams should be positive\n");
                printf("Enter number of output sequuences:  ");
                scanf("%d",&out_number);
                }       /*end of while loop*/
                printf("\n");
                printf("Enter length of a sequence:  ");
                scanf("%d",&Length);
                while (Length <= 0)
                {
                printf("Sequence length should be positive\n");
                printf("Enter length of a sequence:  ");
                scanf("%d",&Length);
                }       /*end of while loop*/
                printf("\n");
                for(j=0; j<Length; j++)
                        for(i=0; i<out_number; i++)
                                {
                printf("Output sequence #%d / bit #%d:  ",i,j);
                                scanf("%d",&out_seq[i][j]);
                if (out_seq[i][j] != 0)
                if (out_seq[i][j] != 1)
                {
                printf("Bit stream values has to be binary either 0 or 1\n");
                printf("Output sequence #%d / bit #%d:  ",i,j);
                scanf("%d",&out_seq[i][j]);
                }       /*end of if test*/
                                }       /*end of for loop*/
read_init_values_manually();
}


/*****************************************************************
 *                                                               *
 *      FUNCTION THAT READS THE PROBABILITY OF SINGLE            *
 *      AND DOUBLE ERRORS WITH EFFECT LEFT AT THE OUTPUT         *
 *      OF THE CUT                                               *
```

```
 *                                                                    *
 **********************************************************************/

read_S()
        {
        printf("Enter a probability value for S1:       ");
        scanf("%f",&S1);
        while (S1<0.0)
                {
        printf("The probability of single error should be positive\n");
        printf("Enter a probability value for S1:       ");
        scanf("%f",&S1);
                }       /*end of while loop*/
        printf("\n");
        printf("Enter a probability value for S2:       ");
        scanf("%f",&S2);
        while (S2<0.0)
                {
        printf("The probability of double error should be positive\n");
        printf("Enter a probability value for S2:       ");
        scanf("%f",&S2);
                }       /*end of while loop*/
        printf("\n");
        }

/**********************************************************************
 *                                                                    *
 *      FUNCTION THAT READS MANUALLY THE INITIAL OUTPUT               *
 *      VALUES DEFINED IN THE ISCAS CIRCUIT DESCRIPTIONS              *
 *                                                                    *
 **********************************************************************/

read_init_values_manually()
        {
        int i;
        /*printf("Enter the last line number in a circuit:  );
        scanf("%d",&last_line);
        while(last_line < 0)
        {
        printf("The last line number should be positive\n);
        printf("Enter the last line number in a circuit:  );
        scanf("%d",&last_line);
        }*/
        for (i=0; i<out_number; i++)
                {
                printf("Enter the initial line number for output sequence #%d:
",i);
                scanf("%d",&init_values[i]);
                while(init_values[i] < -1)
                {
                printf("The initial line number should be positive\n");
                printf("Enter the initial line number for output sequence #%d:
",i);
                scanf("%d",&init_values[i]);
                }       /*end of while loop*/
                printf("\n");
                }       /*end of for loop*/
        }

/**********************************************************************
```



```
 *                                                                      *
 *      FUNCTION THAT READS FROM A FILE THE INITIAL OUTPUT              *
 *      VALUES DEFINED IN THE ISCAS CIRCUIT DESCRIPTIONS                *
 *                                                                      *
 ************************************************************************/

read_init_values()
{
register i;
in = fopen("out_line_numb.dat","r");
        if(in)
        {
        while(!feof(in))
                {
                for (i=0; i<out_number; i++)
                        {
                        fscanf(in,"%d\n",&init_values[i]);
                        if (init_values[i] >= 0)
                                {
                                init_values[i] = init_values[i];
                                }       /*end of if test*/
                        else
                                {
                        printf("The initial line number should be positive\n");
                        printf("Check your data file\n");
                        exit(1);
                                }
                        }       /*end of for loop*/
                }       /*end od while loop*/
        }       /*end of if test*/
        else
        printf("ERROR OPENING out_line_numb.dat FILE\n");
fclose(in);
}
```

```
# 4 output sequence streams of 5 bits each read from an input
# file "out_seq.dat".

4 5
0 1 1 0
0 0 1 0
1 1 1 0
1 1 1 0
0 0 1 0
```

```
# The initial line numbers read from a file "out_numb.dat".

0
1
2
3
```

```
# The Compaction Tree assuming stochastically independent
# values for single and double line errors as written
# on the output data file "compr_res1.dat".

Output sequences #0 and #3 are merged with an OR gate into #4.
Output sequences #1 and #2 are merged with an AND gate into #5.
Output sequences #4 and #5 are merged with an XOR gate into #6.
```

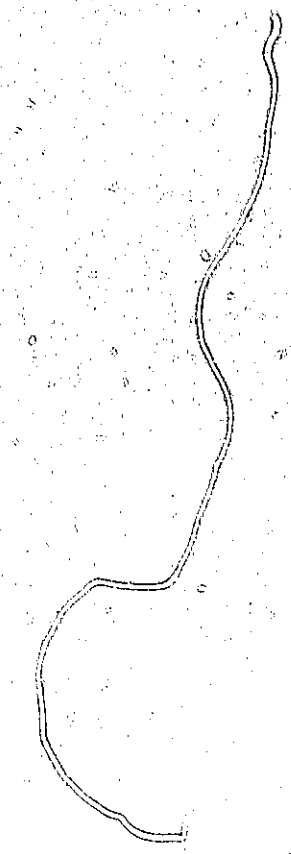
```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.33 , S2=0.66).

Output sequences #0 and #1 are merged with an AND gate into #4.
Output sequences #3 and #4 are merged with an OR gate into #5.
Output sequences #2 and #5 are merged with an AND gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.0 , S2=1.0).

Output sequences #0 and #1 are merged with an AND gate into #4.
Output sequences #3 and #4 are merged with an OR gate into #5.
Output sequences #2 and #5 are merged with an AND gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.05 , S2=0.95).

Output sequences #0 and #1 are merged with an AND gate into #4.
Output sequences #3 and #4 are merged with an OR gate into #5.
Output sequences #2 and #5 are merged with an AND gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.1 , S2=0.9).

Output sequences #0 and #1 are merged with an AND gate into #4.
Output sequences #3 and #4 are merged with an OR gate into #5.
Output sequences #2 and #5 are merged with an AND gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=1.0 , S2=0.0).

Output sequences #0 and #1 are merged with an XOR gate into #4.
Output sequences #2 and #3 are merged with an XOR gate into #5.
Output sequences #4 and #5 are merged with an XOR gate into #6.
```



```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.66 , S2=0.33).

Output sequences #0 and #3 are merged with an OR gate into #4.
Output sequences #1 and #2 are merged with an AND gate into #5.
Output sequences #4 and #5 are merged with an XOR gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.9 , S2=0.1).

Output sequences #0 and #1 are merged with an XOR gate into #4.
Output sequences #2 and #3 are merged with an XOR gate into #5.
Output sequences #4 and #5 are merged with an XOR gate into #6.
```

```
# The Compaction Tree assuming different probability (stochastically
# dependent) values for single and double line errors (S1=0.95 , S2=0.05).

Output sequences #0 and #1 are merged with an XOR gate into #4.
Output sequences #2 and #3 are merged with an XOR gate into #5.
Output sequences #4 and #5 are merged with an XOR gate into #6.
```