National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

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

by

Huong T. Ho

A thesis

submitted to the School of Graduate Studies and Research in partial fulfillment of the requirements for the Degree of Master of Applied Sciences

OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING

University of Ottawa

1994

ISBN 0-612-07895-7

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

In recent years, many test output data compression techniques have been introduced, which reduce the storage requirements of reference signatures for the circuit under test. A major problem, however, is that the compression always results in loss of error coverage.

This work proposes a space compression technique for digital circuits with the objective of minimizing the storage for the circuits under test while maintaining the fault coverage information. The technique introduced is called a Modified Dynamic Space Compression method.

For a circuit under test, a compaction tree is generated based on its structure. The detectable error probability was calculated by using the Boolean Difference Method. The output data modification was employed to minimize the number of faulty output data patterns which have the same compressed form as the fault free patterns. The compressed outputs were then fed into a syndrome counter to derive the signature for the circuit.

A design program is written in C language and executed on PC which combines the space compression, output data modification, and faults testing. Simulations were performed on known combinational circuits and the results indicate that the loss in fault coverage caused by compression is rather small.

ACKNOWLEDGMENTS

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

Furthermore, I would also like to thank **Dr. Valek Szwarc**, my supervisor at work for his kind understanding and encouragement during the course of this work.

Thanks must also go to my colleagues at **C.R.C** for many favors rendered during the period the thesis was written.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

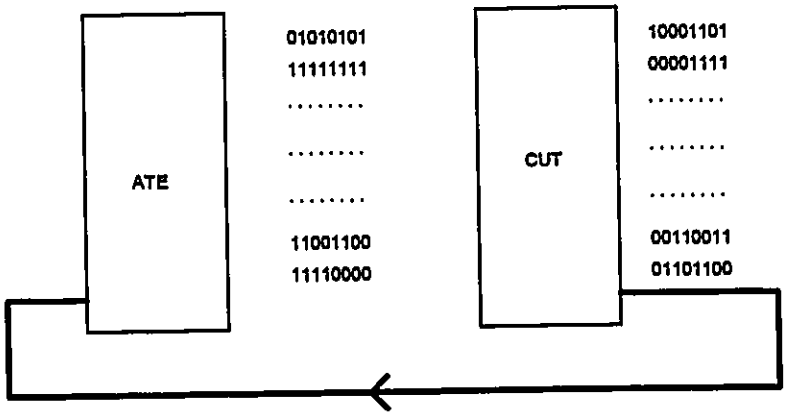
INTRODUCTION

Testing of a system or a device is an experiment in which the system or device is exercised and its resulting response is analyzed to ascertain whether it behaved correctly.

Testing of digital circuits is a major portion of the effort in their design, production and use. The following are some facts about the cost of testing:

- More than 20 % of manufacturing dollar are spent on testing.
- Largest direct and indirect labor cost is involved.
- Largest product manufacturing cost opportunity is in TEST.

Figure 1 shows how testing is done on a circuit under test (CUT).



ATE : Automatic Test Equipment

CUT : Circuit Under Test

Fig.1 Testing of a digital circuit.

Obviously a greater design complexity implies much greater test complexity. Furthermore, low quality chips produce lower quality boards and even lower quality systems.

As the electronics industry evolved from discrete components, through the early integrated circuits that contained a single gate or flip-flop per package, to increasingly high levels of integration, the demands on testing methods and their effectiveness have caused test technology to evolve to a high degree of sophistication. As the number of circuits that can be integrated into one piece of silicon approaches millions, it would seem that some small portion of those circuits could be devoted to the testing or assurance of the function the remainder are intended to implement. This concept is called "built-in self-test". Building the test into the design consumes added circuit and I/O overhead, but at the same time results in visible reductions to the costs of testing when compared with an external test using automatic test equipment. Buit-in testing achieves these savings by

1. Eliminating (or reducing) the costs of test pattern generation and fault simulation.
2. Shortening the time duration of tests (by running tests at circuit speeds).
3. Simplifying the external test equipment.

4. Easily adopting to engineer changes.

Testing is a major concern in integrated circuits. Built-in self-test techniques usually combine a built-in stimulus source with response data compression. This approach eliminates the task of test pattern generation and minimizes the storage of test patterns and response data.

For a given set of test vectors applied in a particular order, we can obtain the expected responses and their order from a CUT or by simulating the CUT. Similar to stored-pattern BIST, we can also store responses in-chip ROM, but such a scheme may require too much silicon to be of practical value. Alternatively, methods that compress the test pattern and the corresponding responses of a fault-free CUT and regenerate them during self-test are also of limited value for general VLSI circuits.

An alternative to response compression is compaction of responses into relatively short binary sequences called a signatures.

This thesis deals with the test compression of combinational circuits. This work consists of generating test compaction tree for a digital circuit under test.

The material of this thesis is organized as follows:

Chapter 1 provides an overview of economic effects in testing as well as a brief introduction of the work discussed in this thesis.

Chapter 2 provides several existing test compression techniques.

Chapter 3 presents a novel technique for test compression of combinational circuits called Modified Dynamic Compression method.

Chapter 4 provides experimental results achieved by applying MDSC on several known combinational circuits.

Chapter 5 gives a summary and conclusions of the work done.

CHAPTER 2

TEST COMPRESSION TECHNIQUES

It is obviously unsatisfactory to build into the circuit a bit-by-bit comparison of its test responses to the expected reference responses because of the large reference storage capacity that would be needed. Instead, it is usual to perform some form of data compression on the responses before making the reference comparison [Sal'83]. The compressed response is referred to as the signature of the circuit under test, and comparison is made to the precomputed and stored reference signature. A block diagram of testing procedure on a digital circuit with output data compression is shown in Figure 2

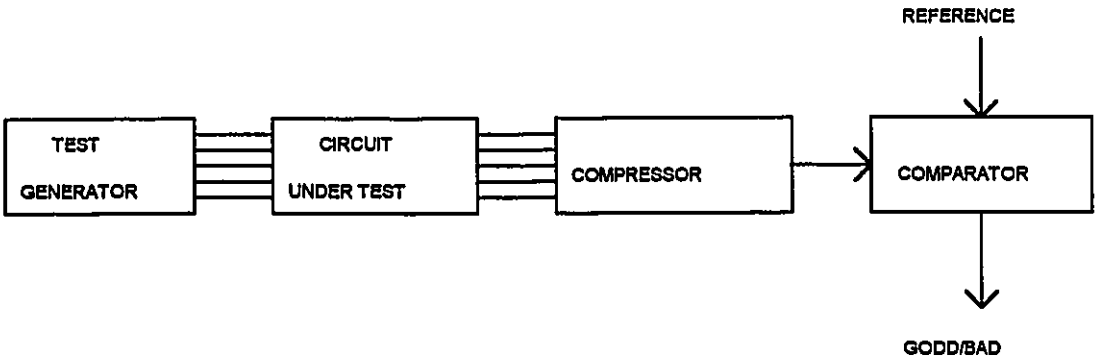


Fig.2 Block diagram of testing procedure with output data compression.

The compression procedure should not introduce signal delays that affect either the normal behavior or the test execution time of the CUT. In addition, the length of the test signature should be a logarithmic factor of the length of the output response data so that the amount of storage is significantly reduced [Yar'90]. Also, for many faults and

associated output responses containing one or more errors, the signatures of the faulty and fault-free cases should not be the same. This is required to ensure that the compression method does not lose information.

Many compression methods have been suggested in the literature and some are in actual use [Bar'87]:

1. Parity checking.
2. Transition counting.
3. Syndrome generation or ones counting.
4. Signature analysis.
5. Walsh spectra.

Parity checking of a response bit stream from a circuit under test reduces a multitude of output data to a signature of length 1 bit. Parity checking is relatively ineffective since a large number of the possible response bit streams from a faulty circuit will result in the same parity as the correct stream. Transition counting counts the number of times the output bit stream changes from 1 to 0 and vice versa. Low counts and high counts are less susceptible to masking than are intermediate counts.

Syndrome or ones counting uses as its signature the number of ones in the binary circuit response stream. Syndrome testing differs from ordinary ones counting in that a circuit can be designed for zero masking under syndrome test.

Signature analysis uses a shift register with various stages tapped and fed back to an EXCLUSIVE OR gate that in turn feeds the register input. The circuit output bit stream is applied to an additional EXCLUSIVE OR input and perturbs the feedback. The signature is the state of the register following completion of the test.

Walsh spectral analysis verifies, in a sense, the truth table of the function. Collecting and comparing a subset of the complete Walsh functions has been described as a mechanism for data compression.

Linear feedback shift register, multiple input linear feedback shift register, transition count and syndrome counting are typical time compressors. In general, compression techniques can be divided into two categories, compression in time or in space. They essentially have the same structure as Figure 2. When a number of nodes of the CUT, monitored in deriving the response of the fault-free circuit, is large it may be cost effective to introduce a space compressor, to reduce the width of the test data, before a time compressor is used to derive the signature.

The objective of the space compressor is to compress the m test inputs of the CUT into r test outputs ($r << m$) while retaining the fault detection properties of the test set.

The principal advantages of space compression are the reduction in the number of pins monitored by the tester and the minimization of the memory space required for the reference signatures. However, compression will reduce useful information and hence fault

coverage. Therefore, any space compression technique which maintains more error coverage information would have to be considered for investigation.

In this work, we will be presenting a novel space compression technique which is employed in the compression of the test data output of digital circuits. We call this technique a Modified Dynamic Space Compression or MDSC method in brief.

The space compression technique initially discussed in this work was first proposed by Li and Robinson [Li'87]. Instead of using only EXCLUSIVE OR (EXOR) gates as an output compression tool, this technique uses AND, OR and EXOR gates, which organize an output compression tree to compress the multiple outputs of the CUT to a single line. This technique is called HSC (Hybrid Space Compressor).

A technique subsequently proposed by Jone and Das [Jon'91] was a modified version of the HSC method. Instead of assigning a static value for the probabilities of single errors and double errors S1 and S2, this method dynamically estimated those values based on the CUT structure during the computation process. This method is called DSC (Dynamic Space Compression) method. The values of S1 and S2 were determined based on the number of single lines and shared lines connected to an output.

The Modified Dynamic Space Compression or MDSC method as proposed herein is a refinement over that proposed by Jone and Das, as will be evident from subsequent discussions.

CHAPTER 3

THE MODIFIED DYNAMIC SPACE COMPRESSION METHOD

The principal idea of the MDSC method is to compress m functional test outputs of the CUT into one simple test output line without sacrificing too much information. Figure 3 illustrates this technique.

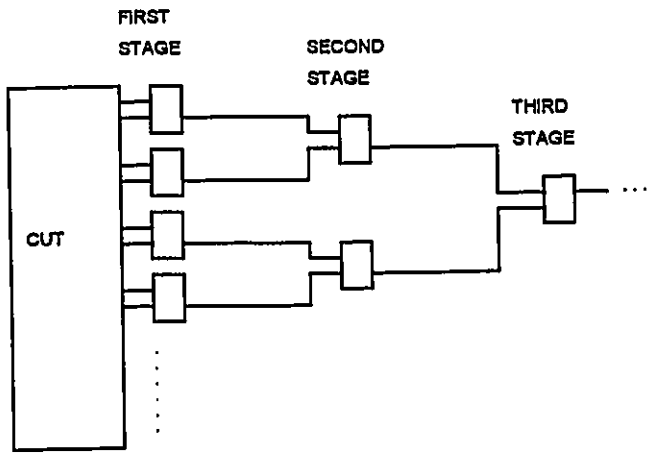


Fig.3 Modified Dynamic Space Compression technique.

Generally, Space Compression has been accomplished using EXOR gates in cascade or in a tree structure. In MDSC, we will assume a tree structure for our framework with AND, OR and EXOR operators. The logic function to be selected to build the tree will be determined by the characteristics of the sequence which are inputs to the gate as well as the structure of the CUT.

We assume a syndrome counter [Aga'83] at the output of the two-input gates of the compaction tree as shown in Figure 4.

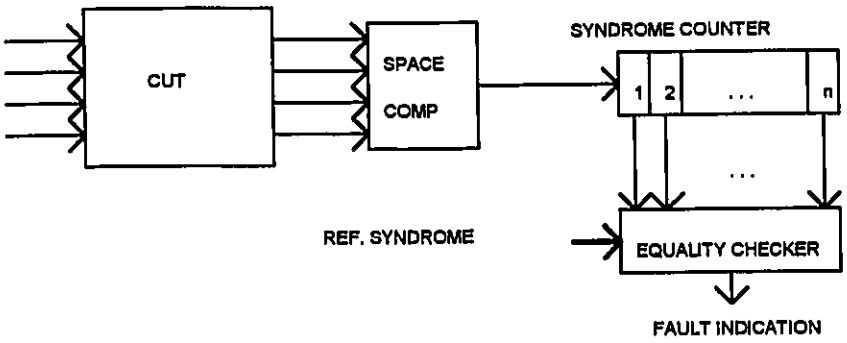


Fig.4 Syndrome counter used as time compressor in test data output compression.

The time compressor is used to derive the signature for the CUT. The syndrome counter counts the number of ones appearing at the output of the gate (or the CUT). The equality checker checks the counter contents with the expected syndrome. If the syndrome from the CUT is different from syndrome expected from the fault-free circuit, the CUT is defined 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 CUT into faulty syndrome has to be examined. Consequently, the output data modification in [Zor'84] is applied to increase the error coverage in single output CUT.

The basic idea of the MDSC is to select a suitable gate to merge two candidate output lines of the CUT. The selection is based on a detectable error probability estimate [Li'87].

The detectable error probability estimate E for a two input logic function, given two input sequences of length L is defined as follows:

$$E = S1\frac{R1}{2L} + S2\frac{R2}{L} \qquad (3.1)$$

where

S1 : probability of single error effect felt at the output of the CUT.

S2 : probability of double error effect felt at the output of the CUT.

R1 : number of single line errors detected at the output of gate i, if logic gate i is used.

R2 : number of double line errors detected at the output of gate i, if logic gate i is used.

L : input sequence length.

The probability of single error S1 and the double error S2 is dynamically calculated based on the CUT structure. For each output pair selected to be merged, the expected number of faults α which yield single errors and the expected number of faults β which yield double errors is determined by the Boolean Difference Method (BDM) [Lee'76].

3.1 Estimated Probabilities of Single Error S1 and Double Error S2

3.1.1 The Boolean Difference Method

The Boolean Difference is defined as being the exclusive OR operation between two Boolean functions, one representing the normal circuit and the other representing the faulty circuit. Thus, if the Boolean Difference is a 1, a fault is indicated. This indication

denotes that either a single stuck-at fault or a multiple stuck-at fault is present in the circuit.

3.1.1.1 Single error faults determination based on BDM

Assume that there is a switching function that has one output F and n inputs x_1, x_2, x_3,...,x_n; so F(X) = F(x_1, x_2, x_3,..., x_n). If one of the inputs to the switching function was in error, input x_i for example, then the output would be F(x_1, x_2,..,$\overline{x_i}$,....x_n). The following function was defined:

$$\frac{dF(X)}{dx_i} = F(x_1,\ldots,x_i,\ldots,x_n) \oplus F(x_1,\ldots,\overline{x_i},\ldots,x_n) \qquad (3.2)$$

The function $\frac{dF(X)}{dx_i}$ is called the Boolean Difference of F(X) with respect to x_i. The function F(X) is independent of x_i if

$$\frac{dF(X)}{dx_i} = 0 \qquad (3.3)$$

If we consider F(X) as an output function of a combinational circuit, then we can see that F(X) is independent of variable x_i if for any values of the other variables, the output F(X) is independent of the value of x_i. This implies that a fault in x_i will not affect the final output F(X).

By using this property, we assume that if the function F(x) is dependent of xi then for each s-a fault in the xi, the fault effects can appear at the output F(x) or not appear at the output F(x) with equal probability when test patterns applied.

Similarly, we consider now F(X) and G(x) as output functions of a combinational circuit. If the functions F(x) and G(x) are dependent simultaneously on xi then for each s-a fault in the xi, the fault effects can appear at F(x), or G(x), or neither, or both with equal probability.

For each output pair O1 and O2 selected to be merged, we define:

L1 = Number of lines in the CUT dependent only on O1

L2 = Number of lines in the CUT dependent only on O2

L12 = Number of lines in the CUT dependent simultaneously on O1 and O2

And the expected number of single stuck-at faults, α, which yield single errors, equals

$$\alpha = 2(\frac{1}{2}L1 + \frac{1}{2}L2 + \frac{1}{4}2L12)$$

or

$$\alpha = L1 + L2 + L12 \qquad (3.4)$$

The Boolean Difference of the function F(x) with respect to xi can be determined by using this property:

$$\frac{dF}{dx_i} = \frac{dF}{dx_j}\frac{dx_j}{dx_i} \tag{3.5}$$

Here j is an internal line and i is either an internal line or a primary input line of the CUT such that every path from i to the primary output line passes through j. Note that xi and xj are the variables representing the logic values on line i and j, respectively.

3.1.1.2 Double error faults determination based on BDM

Using the above assumption, we can derive the number of faults, β, which yield double errors at the outputs as

$$\beta = 2(L12\frac{1}{4})$$

or

$$\beta = \frac{1}{2}L12 \tag{3.6}$$

For every output pair of the circuit under test chosen to be merged, the expected number of single errors faults α and double errors faults β are dynamically determined based on the Boolean Difference method.

3.1.2 Calculation of S1 and S2

Assuming that the sum of all the probabilities is equal to one, the probabilities of single error S1 and double error S2 are defined as:

$$S1 = \frac{\alpha}{\alpha + \beta} \tag{3.7}$$

and

$$S2 = \frac{\beta}{\alpha + \beta} \tag{3.8}$$

Example

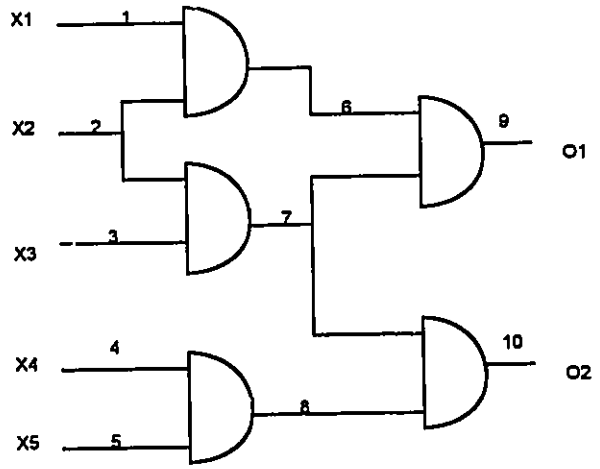


Fig.5 Architecture for estimation of α and β.

Let us take an arbitrary circuit as shown in Figure 5. The Boolean differences of O_1 and O_2 with respect to every line in the circuit (which is not a primary output) are:

$$\frac{dO_1}{dX_1} = X_3X_2 \qquad \frac{dO_2}{dX_1} = 0$$

$$\frac{dO_1}{dX_2} = X_2X_1X_3 \qquad \frac{dO_2}{dX_2} = X_4X_3X_5$$

$$\frac{dO_1}{dX_3} = X_1X_2 \qquad \frac{dO_2}{dX_3} = X_4X_2X_5$$

$$\frac{dO_1}{dX_4} = 0 \qquad \frac{dO_2}{dX_4} = X_2X_3X_5$$

$$\frac{dO_1}{dX_5} = 0 \qquad \frac{dO_2}{dX_5} = X_2X_3X_4$$

$$\frac{dO_1}{dX_6} = X_2X_3 \qquad \frac{dO_2}{dX_6} = 0$$

$$\frac{dO_1}{dX_7} = X_1X_2 \qquad \frac{dO_2}{dX_7} = X_4X_5$$

$$\frac{dO_1}{dX_8} = 0 \qquad \frac{dO_2}{dX_8} = X_2X_3$$

We can easily determine the number of lines dependent only on the primary output O1

L1 = 2

and the number of lines dependent only on the primary output O2

L2 = 3

Finally, the number of lines dependent simultaneously on both of the primary outputs O1 and O2

L12 = 3

Hence, the expected number of single stuck-at faults α which yield single errors and the expected number of faults β, which yield double errors for the circuit in Figure 5 are calculated as follows:

$$\alpha = 8 \qquad \text{and} \qquad \beta = 3/2$$

where

$$L1 = 2; \quad L2 = 3; \quad L12 = 3$$

And the probabilities of single error S1 and double error S2 are:

$$S1 = \frac{16}{19} \qquad S2 = \frac{3}{19}$$

3.1.3 The Boolean Difference algorithm

The algorithm for generating the Boolean Difference equations to determine the number of single error faults and double error faults at the outputs of the CUT is described as follows:

STEP 1

Number all the lines of the logic circuit under consideration as follows:

1- Number all the lines using positive integers in ascending order starting with primary inputs from top to bottom.

2- Draw the circuit diagram in such a way that all the gates are lined up into levels of gates. The NOT gates, if any, are drawn into separate columns from those of other types. But a NOT gate will be treated the same as other types if its input is one of the fan-out branches.

3- Label the lines from left to right. The last ones to be numbered are the primary output lines.

STEP 2

For each output pair selected to be merged O1 and O2, find the Boolean differences $\frac{dO_1}{dx_j}$ and $\frac{dO_2}{dx_j}$ where j starts from the largest number to the smallest number of lines (not a primary output).

STEP 3

Determine the number of lines dependent only on the output O1(L1), the number of lines dependent only on the output O2(L2) and the number of lines dependent on both of the outputs O1 and O2(L12) based on the Boolean Differences equations obtained above.

STEP 4

Determine the expected number of single errors faults α and double errors faults β with the values of L1, L2 and L12 obtained in step 3. Calculate the estimated probabilities of single errors S1 and double errors S2 based on these α and β.values.

Example: Consider the circuit shown in Figure 6. The Boolean Difference Algorithm is now applied to this circuit, in which, each line is labeled using the rules described in Step 1.

According to Step 2, all the $\frac{dO_1}{dX_j}$, j = 8,7,...,1 are listed:

$$\frac{dO_1}{dX_8} = 0 \qquad \frac{dO_1}{dX_7} = \overline{X_1} + \overline{X_2}$$

$$\frac{dO_1}{dX_6} = X_2 + X_3 \qquad \frac{dO_1}{dX_5} = 0$$

$$\frac{dO_1}{dX_4} = 0 \qquad \frac{dO_1}{dX_3} = \overline{X_2}$$

$$\frac{dO_1}{dX_2} = \overline{X_2}\,\overline{X_3} + \overline{X_1}\,\overline{X_3} \qquad \frac{dO_1}{dX_1} = X_2$$

and all the $\frac{dO_2}{dX_j}$, j = 8,7,...,1 are

$$\frac{dO_2}{dX_8} = \overline{X_2\,X_3} \qquad \frac{dO_2}{dX_7} = X_4 X_5$$

$$\frac{dO_2}{dX_6} = 0 \qquad \frac{dO_2}{dX_5} = \overline{X_2}\,\overline{X_3} X_4$$

$$\frac{dO_2}{dX_4} = \overline{X_2}\,\overline{X_3} X_5 \qquad \frac{dO_2}{dX_3} = 0$$

$$\frac{dO_2}{dX_2} = 0 \qquad \frac{dO_2}{dX_1} = 0$$

According to step 3, the total number of lines dependent only on the output O1 are

$$L1 = 4$$

and the total number of lines dependent only on the output O2 are

$$L2 = 3$$

The total number of lines dependent on both of the outputs O1 and O2

$$L12 = 1$$

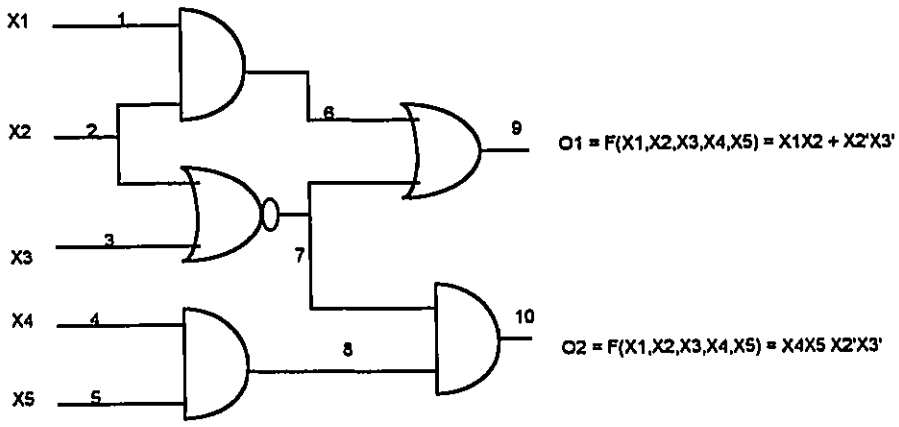


Fig.6 Circuit to illustrate the algorithm for the calculation of S1 and S2 based on Boolean Difference Method.

Based on the above calculated values, we can determine the number of expected single errors faults α and the number of expected double errors faults β for the primary outputs of the circuit in Figure 6 as

$$\alpha = 8 \qquad \text{and} \qquad \beta = 1/2$$

Thus, the probability of single errors S1 and double errors S2 are :

$$S1 = \frac{16}{17} \qquad \text{and} \qquad S2 = \frac{1}{17}$$

Table 1 summarizes the Boolean differences of six most commonly used gates.

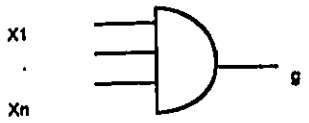
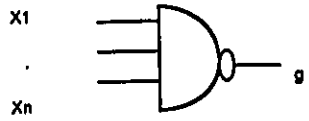
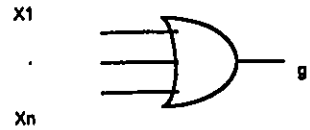
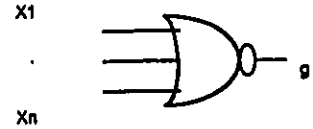
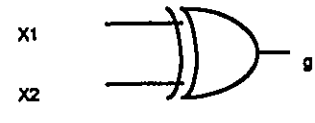
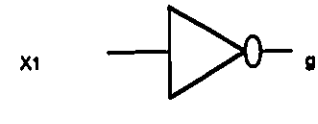
 AND gate  NAND gate	$\frac{dg}{dXk} = X1X2...Xk\text{-}1Xk\text{+}1...Xn$
 OR gate  NOR gate	$\frac{dg}{dXk} = X1'X2'...Xk\text{-}1'Xk\text{+}1'...Xn$
 EXOR gate	$\frac{dg}{dx1} = \frac{dg}{dx2} = 1$
 INVERTER	$\frac{dg}{dx1} = 1$

Table 1: Boolean differences of six most commonly used gates.

3.2 Determination of Errors Detected R1 and R2

The number of single line errors detected at the output of gate i, R1 and the number of double line errors detected at the output of gate i, R2 will be determined by examining the output patterns at the output gate i (if logic gate i is used).

The values of R1 and R2 have to be calculated separately for the case of AND, OR and EXOR gates. The logic behavior of the gate under investigation used to merge two candidate outputs will determine the number of single line errors and double line errors detected at the output of that gate.

Consider the Ten-line decimal-to-8421 BCD conve;ter in Figure 7 The circuit has 9 inputs: X1, X2, X3, X4, X5, X6, X7,X8 and X9, and four outputs: O1, O2, O3 and O4. For the ten exhaustive input patterns as shown in the following table, the output line values are shown below.

X1	X2	X3	X4	X5	X6	X7	X8	X9	O1	O2	O3	O4
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	0	1	0	0
0	0	1	0	0	0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	1	0

0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	0	1	1	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
									--	--	--	--
									155	cc	3c	03
									5	4	4	2

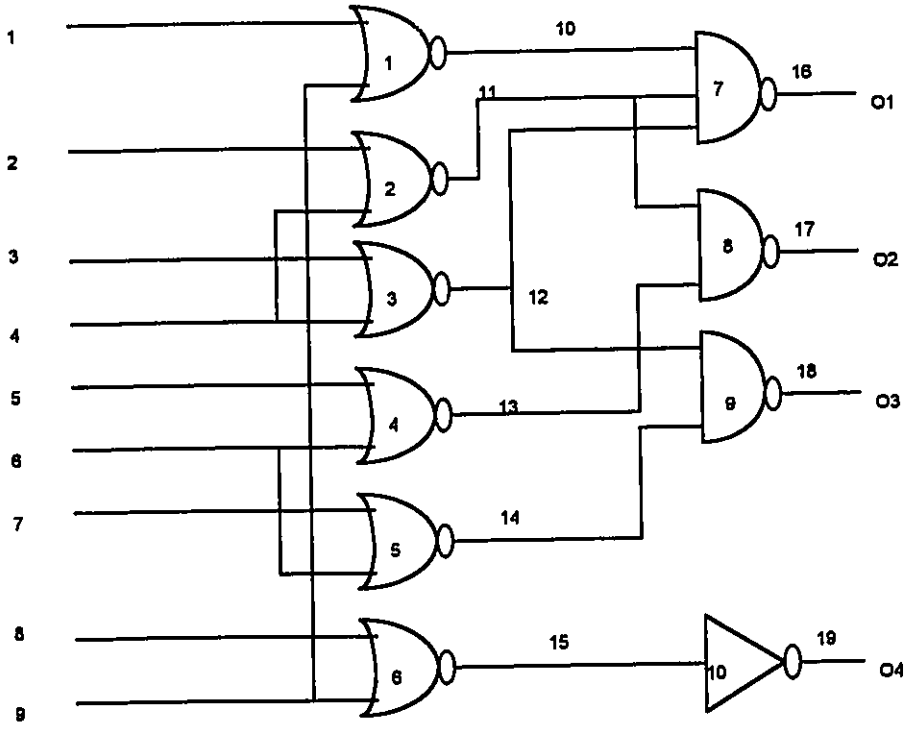
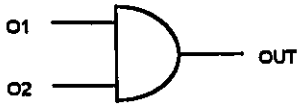


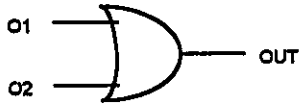
Fig.7 Ten line Decimal to 8421 BCD converter.

From the above truth table, we can see that O1 and O2 have the highest and second highest weights; so they are chosen to be merged. We are going to examine all cases.

For the AND gate, the number of single line errors detected at the output of the AND gate is 9. This is obtained by examining from 10 bit test output patterns of the two lines O1 and O2. For every test pattern applied to the input lines O1 or O2, we count the number of faults appearing at the output of the AND gate. Similarly, when the two lines O1 and O2 change their status simultaneously, the number of the double line errors detected at the output of the AND gate is 5.



For the OR gate, either O1 or O2 change its status, and the single line error detected at the output of the gate is 11. The same estimation for the number of double line errors detected at the output of the OR gate gives 5.



For the EXOR gate, any single line changing its status will cause the output to change its value. Thus, the number of single line errors detectable is always 10. The double line errors detectable is 0. This value is derived from the fact that the output of the EXOR gate is unchanged if the two lines change their status at the same time.

3.3 Input Sequence Length L

The input sequence length L is the length of the test input patterns applied to the CUT. Apparently, this variable plays a very important part in the determination of the compaction tree for the circuit under test. The elements of the compaction tree will change simultaneously with the modification of test inputs sequence length. Furthermore, the test inputs sequence length has a significant effect on faults coverage for the circuit. This will be demonstrated later in the simulation section.

3.4 The Data Output Modification

Compression techniques are divided into two classes: time compression and space compression [Sal'83]. The space compressor produces a single signature for multiple output circuits rather than an individual signature for each output. Sequences coming out of the space compressor are stored in a time compressor as a signature.

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 for our compression model is one that uses the Syndrome Counter as a time compressor. The Syndrome Counter is used to reduce test output data by counting the number of ones. Hence, this is independent of the order of the test patterns.

The syndrome of a circuit is defined as the number of times the function has logic value one for the test sequence [Sav'80]. 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 [Sal'83].

Moreover, to increase the error coverage, the concept of modification of the output data before compression [Aga'83] is used in the design. The basic idea of output data modification is to reduce the deception volume of output data after compression (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.

Let the test output sequence be Q and the serial string Q' be the sequence coming out of the modification stage. If the number of 1's in Q is W and Q' is W', then [Zor'84] the improvement in deception volume is given by the ratio:

$$\frac{P'}{P} = \frac{\binom{n}{W'}}{\binom{n}{W}} \qquad (3.9)$$

where P' (P) is the error masking probability obtained by compressing W'(W).

Example: As illustration, let the weight W of the test output sequence be 15 with test sequence length n = 32 and the weight W' of the output string of the syndrome counter be 11 as given below:

W = 1101 0011 1000 0101 1001 0110 1100 0001

W' = 0010 1001 1000 0000 0110 1100 1100 0100

Then the improvement in deception volume is calculated as

$$\frac{P'}{P} = \frac{\binom{n}{W'}}{\binom{n}{W}}$$

$$\frac{P'}{P} = 0.228$$

Figure 8 shows that the reduction in deception volume is realized by modifying the output sequence Q, by using a Modifier sequence S, so that the resulting sequence Q' has a deception volume which is much smaller than that of Q.

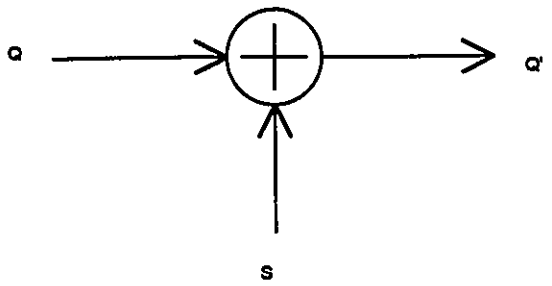


Fig. 8 Output data modification.

In the framework of this thesis, we use the Syndrome counter as a time compressor; hence, the compression is done by counting the number of ones in the output sequence. This one's counting scheme, have a nonuniform deception volume, as shown in Figure 9, where the deception volume is a function of weight. For instance, the resulting signature is the weight of its n-bit output string. Since this weight is generally close to $\frac{n}{2}$, its deception volume, as seen in Fig. 9, will be $2^{(n-1)}$. However, if the weight happens to be, either close to zero or 1, then the deception volume will be much smaller.

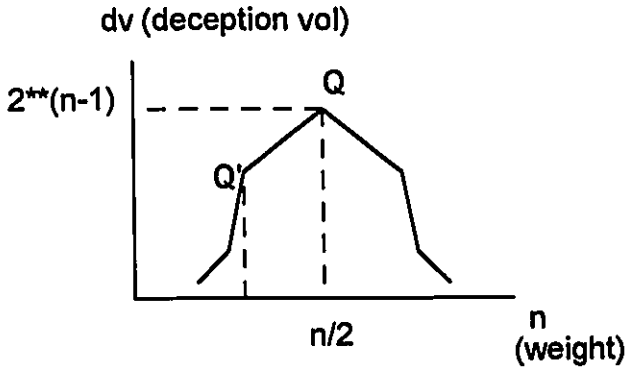


Fig. 9 Nonuniform deception volume

The purpose of the modification concept is to take advantage of this non-uniform distribution, so as to reduce the deception volume of the output sequence.

According to [Zor'84], if the number of ones in the output sequence of the CUT (after the compaction tree) is w and the weight of the output string of the counter is $\frac{n}{2}$, then the improvement in deception volume is given by the ratio:

$$S0 = \frac{\binom{n}{W}}{\binom{n}{\frac{n}{2}}} \qquad (3.10)$$

where n is the sequence length.

This modification concept will be applied to the output sequence of the last stage of the compression procedure before feeding it to the time compressor to derive the signature of the CUT. The procedure is not really complex but it leads to a reduction in the number of output data patterns which have the same compressed form as the fault-free patterns.

3.5 Algorithm to Determine the Compaction Tree

The algorithm used for the space compression is the following [Li'87]:

Do n = 1, m - 1; except the last stage

Do N = 1,p,2; within the same stage

a)Find „(N) and „(N+1)

b)Derive S1 and S2 for the output pair corresponding to „(N) and „(N+1), respectively.

c)Compare the detectable error probability for „(N) and „(N+1) with AND, OR and EXOR gates.

d)Take the logic function for „(N) and „(N+1) with the highest value of detectable error probability to provide another sequence for next stage.

Enddo;{N}

Enddo;{n}

For the last stage, derive S1 and S2 for the two outputs, and choose a gate with the highest value of detectable error probability. If two or three gates have the same value, choose the one with smaller S0 (modification factor)

where:

$_n$(N): The number of ones in the sequence in the nth stage beginning the Nth rank.

$_n$(N+1): The number of ones in the sequence in the nth stage beginning the (N+1)th rank.

The modification factor S0 is

$$S0 = \frac{\binom{n}{W}}{\binom{n}{\frac{n}{2}}}$$

where

$$\binom{n}{W} = \frac{n!}{W(n-W)!} \tag{3.11}$$

and

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\frac{n}{2}\left(\frac{n}{2}\right)!} \tag{3.12}$$

with W being the number of ones in the output sequence of the CUT (last stage)

Example 1: Let us examine the Ten-line Decimal-to-8421 BCD converter in Figure 7. In the first stage, since the weight of O1 and O2 are the highest and the second highest, these two lines are selected to be merged. By applying BDM, we derive the values of S1 and S2 as 0.91 and 0.09, respectively. The detectable error probability estimate E of the logic gate AND, OR and EXOR used to merge two lines O1 and O2 are calculated as:

$$\text{AND} \quad E = 0.91\left(\frac{9}{20}\right) + 0.09\left(\frac{5}{10}\right) = 0.45$$

$$\text{OR} \quad E = 0.91\left(\frac{11}{20}\right) + 0.09\left(\frac{5}{10}\right) = 0.54$$

$$\text{EXOR} \quad E = 0.91\frac{20}{20} = 0.91$$

We can see from the above calculations that the detectable error probability estimate E of the EXOR gate is the highest; therefore, EXOR gate is selected to merge two lines O1 and O2. The output of this EXOR gate is assigned name O5.

The lines O3 and O4 have the second highest weight; hence, the two output lines O3 and O4 will be merged next. The values of S1 and S2 were calculated to be 0.95 and 0.05. And the detectable error probability estimates E are:

$$\text{AND} \quad E = 0.95\left(\frac{6}{20}\right) + 0.05\left(\frac{4}{10}\right) = 0.3$$

$$\text{OR} \quad E = 0.95\left(\frac{14}{20}\right) + 0.05\left(\frac{4}{10}\right) = 0.68$$

$$\text{XOR} \quad E = 0.95\left(\frac{20}{20}\right) = 0.95$$

The detectable error probability estimate E of the EXOR gate is the highest; therefore, EXOR gate is selected to merge two lines O3 and O4. The output of this gate is now named O6.

In the second stage, we have:

O5 0110011001 --> weight = 5

O6 0000111111 --> weight = 6

The process is repeated until a single output line is reached. The compression tree obtained is shown in Figure 10

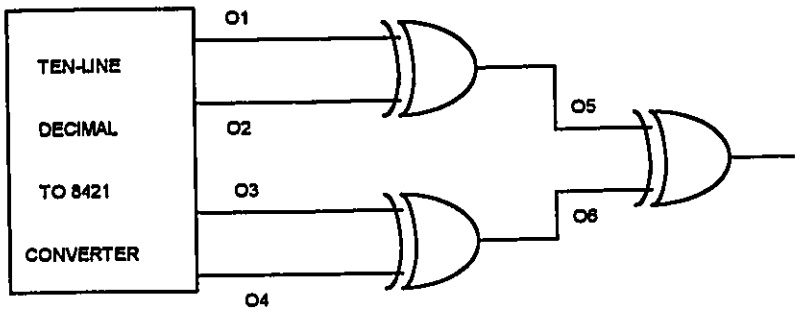


Fig. 10 The compaction tree of the Ten-line Decimal to 8421 BCD converter generated using MDSC method.

Example 2: In Figure 11, a 4-output demultiplexer is shown with four different inputs (grouped input test pattern) X(1), X(2), X(3) and X(4). The fault-free output patterns are:

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
				=	=	=	=
			weight	1	1	1	1

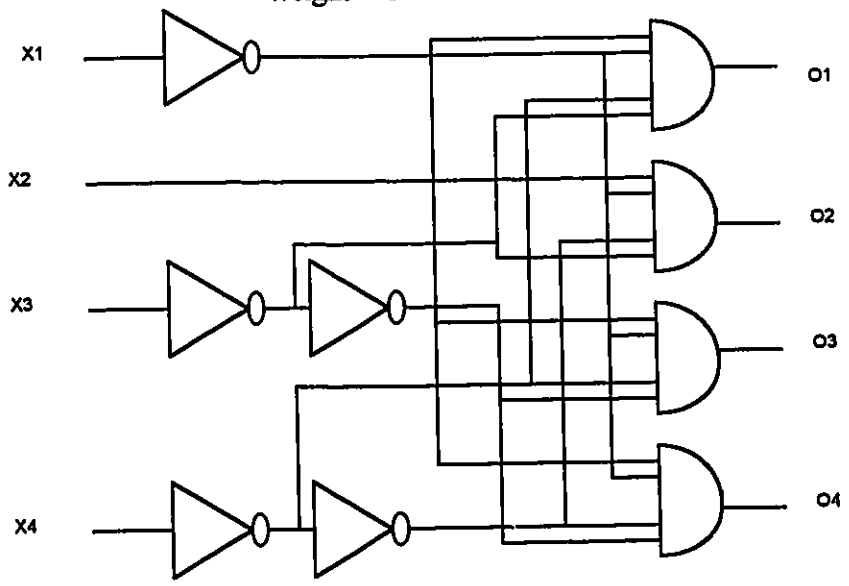


Fig.11 4-output demultiplexer.

Since all the outputs have the same weights, we can select any two of them to be merged. Let us take O1 and O2 to be compressed. By using BDM, we can derive the expected single errors faults α and expected double errors faults β to be 8 and 3.5. So S1 and S2 are calculated to be 0.7 and 0.3. The detectable error probabilities are as follows:

$$\text{AND} \quad E = 0.7(\frac{2}{32}) + 0.3(\frac{14}{16}) = 0.3$$

$$\text{OR} \quad E = 0.7(\frac{30}{32}) + 0.3(\frac{14}{16}) = 0.91$$

$$\text{EXOR} \quad E = 0.7(\frac{32}{32}) = 0.7$$

Hence, the OR logic gate is selected to compress O1 and O2. Then O3 and O4 are selected to be compressed. Once again, by using BDM, the values of α and β are calculated to be 14 and 42. The values of S1 and S2 are evaluated to be 0.7 and 0.3. The detectable error probabilities are:

$$\text{AND} \quad E = 0.7(\frac{2}{32}) + 0.3(\frac{14}{16}) = 0.3$$

$$\text{OR} \quad E = 0.7(\frac{30}{32}) + 0.3(\frac{14}{16}) = 0.91$$

$$\text{EXOR} \quad E = 0.7(\frac{32}{32}) = 0.7$$

The OR logic gate is selected to compress O3 and O4. Finally, O12 and O34 are merged by an OR gate (the values of α and β were calculated to be 13 and 4)to a single output line. The circuit with the compression tree in shown in Figure 12.

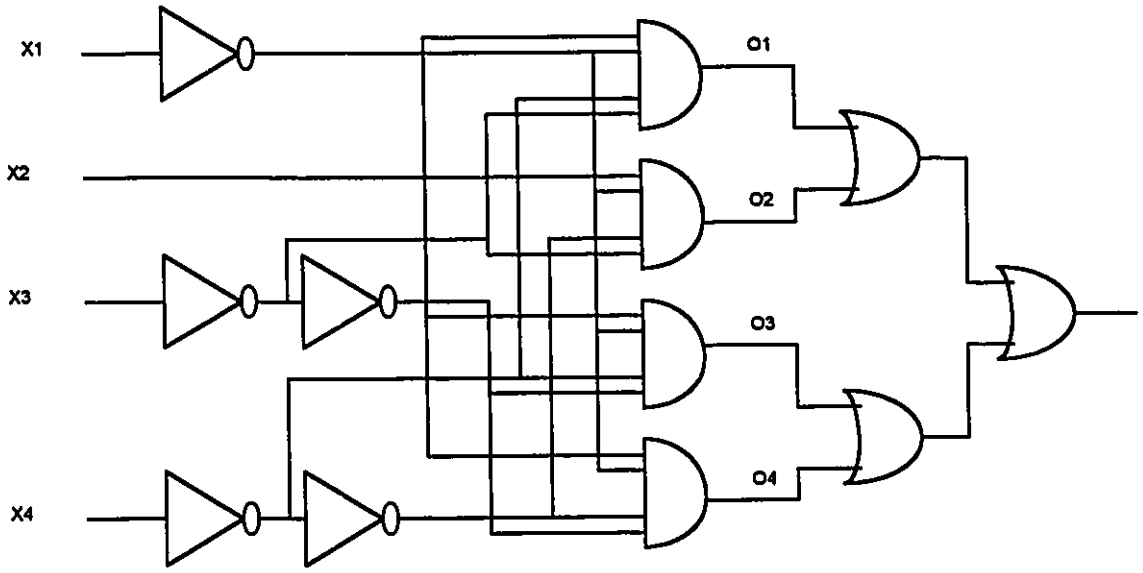


Fig. 12 4-output demultiplexer with compression tree.

3.6 Estimation of Faults Loss by MDSC

Compression of two or more output data always results in loss of information. To measure the loss in fault coverage by using the MDSC scheme, exhaustive tests are performed on the circuit under test. These tests have to be done on the circuit before and after compression. The same number of faults is injected in to the circuits and the fault-free signatures are compared to the faulty signatures. Faults loss by MDSC will be determined by the difference in the number of faulty signatures detected in the two architectures.

3.6.1 Test input patterns generation

Faults can occur singly or in multiples. However, in this work we assume that only single stuck-at faults are present in the circuit under test.

To determine the number of faults missed before and after compaction, exhaustive test vectors are generated. The test vectors equal 2^n, where n is number of input lines of the CUT. A set of single stuck-at faults is introduced into the CUT; the number of faults equal 2*m, where m is the number of lines in the CUT. The faults are to be distributed all over the circuit excluding the compaction tree.

3.6.2 Fault coverage loss by MDSC

The testing procedure is as follows:

-For every test vector applied to the circuit, the correct signature is derived and is stored in memory. Faults (first s-a-0 and then s-a-1) are introduced in the circuit and are distributed all over the m lines, one at a time. Fault signatures are compared to the references to determine if they are detectable. Faulty signatures must not be identical to the correct ones; otherwise, these faults are undetectable.

-The total number of faults missed is known by accumulating the undetectable faults obtained from the test.

-This procedure is applied to the circuit before compaction and after compaction to get the number of faults missed in these two cases.

Consequently, faults missed by MDSC are derived based on the following equation:

$$\% \text{ of faults coverage missed by MDSC} = \frac{N_a - N_b}{N_{Tot}} \qquad (3.13)$$

where

N_a: Number of faults missed after compaction.

N_b: Number of faults missed before compaction.

N_{Tot}: Total number of faults injected.

CHAPTER 4

EXPERIMENTAL RESULTS

To verify the proposed space compression method, simulations were performed on several known logic circuits. These circuits employed for simulation were: Carry Look Ahead generator, Ten line Decimal to 8421 BCD converter, and 2 to 4 decoder.

Table 2 shows the number of single stuck-at faults injected in each circuit. The number of faults missed in the CUT before space compression and the number of faults missed after compression are also given.

The fault coverage loss by MDSC for each circuits is shown in Table 3.

CUT	Number of faults injected	Faults missed before compression	Faults missed after compression
Ten line Decimal to 8421 BCD converter	58	6	7
Carry Look Ahead generator	188	81	83
2 to 4 decoder	56	0	4

Table 2: Number of faults missed before and after compaction on different logic circuits.

CUT	loss of error coverage by MDSC
Ten line Decimal to 8421 BCD converter	1.72%
Carry Look Ahead generator	1.06%
2 to 4 decoder	7.14%

Table 3: Loss of fault coverage caused by MDSC.

From the above results, we can see that the faults missed by MDSC are in the range of 0% to 10%. Comparing to [Jon'91] where the faults missed by compression were in the range of 20%, it is obvious that the estimated probabilities S1 and S2 have a very important role in the generation of the compaction tree.

In MDSC, the values of S1 and S2 are computed for each different candidate output pair and based on the structure of the CUT so that the information loss by the space compressor can be well controlled.

The test input patterns are yet another factor influencing the estimation of the detectable error probability E [Yar'90]. A major parameter of any testing system for digital circuits is the time consumed by the testing procedure, which is defined uniquely by a test sequence length. For deterministic test design methods, the test length is determined by the required coverage for all possible faults in the circuit and does not exceed 1000 patterns on an average. Obviously, the fault coverage increases with the number of test patterns. Generally, by careful selection of test patterns, the loss caused by compression will be minimized.

Table 4 shows the number of faults missed on a Ten line Decimal to 8421 BDC converter by MDSC based on different test sequences. We can see that for longer test sequence lengths, the faults missed are smaller.

Test sequence length	% of fault coverage loss by compaction
2	6.9%
4	3.45%
64	1.72%

Table 4: Faults missed on Ten line Decimal to 8421 BCD converter corresponding to different test sequence lengths.

Another factor which would influence the fault coverage performance of a circuit designed with tests outputs compression based on MDSC technique is the fan out lines in the circuit under test. For circuits with a large number of lines shared by different outputs, the probability of faults masking will be very high. This faults masking problem is apparently caused by the Syndrome Counter.

By using the Syndrome Counter as a time compressor to derive signatures for the tests outputs of the CUT, the storage requirement for the tests references is reduced. Besides, the test procedure of the CUT will be simplified because the only characteristic of the fault-free circuit would be required is the Syndrome.

However, the penalty paid for using Syndrome Counter in the testing procedure is the increase in the number of faults masking due to the fan-out lines in the CUT.

According to [Sav'80], every fan-out free, irredundant combinational circuit, composed of AND, OR, NAND, NOR and NOT gates is Syndrome testable. It is clear that the candidate lines for Syndrome untestability are the fan-out lines. In other words, the number of faults masked by the time compressor increases with the number of fan-out lines in the CUT.

As an example of syndrome calculation, consider the digital circuit of Figure 13, which realizes the switching function

$$F = x_1\overline{x_2 x_3} + \overline{x_1}x_2x_3$$

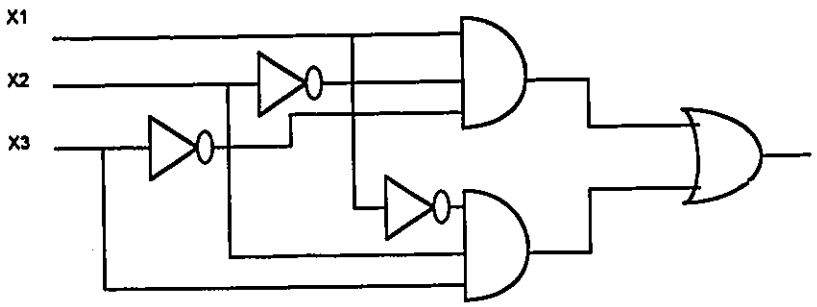


Fig. 13 A digital circuit that realizes the switching function $F = x_1\overline{x_2 x_3} + \overline{x_1}x_2x_3$

The syndrome of the above function is defined by

$$S(F) = S(x_1\overline{x_2 x_3} + \overline{x_1}x_2x_3)$$
$$S(F) = S(x_1\overline{x_2 x_3}) + S(\overline{x_1}x_2x_3) - S(x_1\overline{x_2 x_3}\,\overline{x_1}x_2x_3)$$
$$S(F) = \frac{1}{2^3} + \frac{1}{2^3} - 0$$

$$S(F)=\frac{1}{4}$$

At the same time, the syndrome of the function that describes the behavior of the circuit of Fig. 13 having fault stuck-at 1 on line x_1 is

$$S(F)=S(\overline{x_2}\,\overline{x_3})$$
$$S(F)=\frac{1}{4}$$

This implies that the stuck-at 1 fault at the input x_1 of the circuit in Fig. 13 is undetectable by comparing the reference syndrome with the actual syndrome.

Consider now the digital circuit of Figure 14, which realizes the switching function

$$F=x_1\overline{x_2}\,\overline{x_3}+\overline{x_4}x_5x_6$$

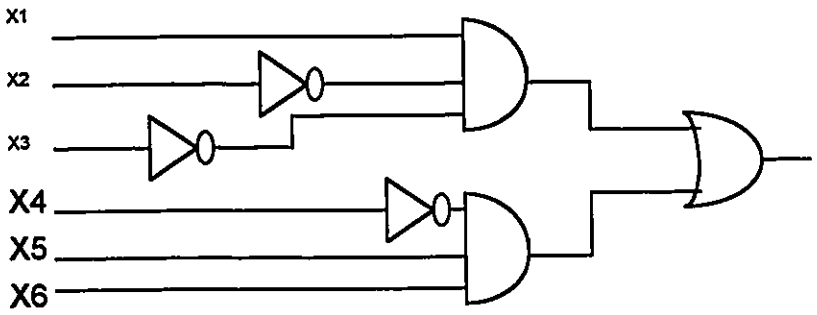


Fig. 14 A digital circuit that realizes the switching function $F=x_1\overline{x_2}\,\overline{x_3}+\overline{x_4}x_5x_6$

The syndrome of this function is defined as

$$S(F)=S(x_1\overline{x_2}\,\overline{x_3}+\overline{x_4}x_5x_6)$$
$$S(F)=S(x_1\overline{x_2}\,\overline{x_3})+S(\overline{x_4}x_5x_6)-S(x_1\overline{x_2}\,\overline{x_3}x_4x_5x_6)$$

$$S(F)=\frac{1}{2^3}+\frac{1}{2^3}-\frac{1}{2^6}$$
$$S(F)=\frac{15}{64}$$

And the syndrome of the circuit that describes the behavior of the circuit of Fig. 14 having fault x_1=1 is

$$S(F)=S(\overline{x_2}\,\overline{x_3}+\overline{x_4}x_5x_6)$$
$$S(F)=S(\overline{x_2}\,\overline{x_3})+S(\overline{x_4}x_5x_6)-S(\overline{x_2}\,\overline{x_3}\,\overline{x_4}x_5x_6)$$
$$S(F)=\frac{1}{2^2}+\frac{1}{2^3}-\frac{1}{2^5}$$
$$S(F)=\frac{11}{32}$$

This indicates that the stuck-at 1 fault on line x_1 of the circuit in Fig. 14 is syndrome testable due to the difference of the actual syndrome and the reference syndrome.

As a matter of fact, the circuit in Fig. 13 is syndrome untestable in all variables while the circuit in Fig. 14 is implemented to be syndrome testable. This confirms the effect of fan-out lines on faults coverage in a CUT designed with tests output compaction using Syndrome counter as a time compressor.

The programs to generate the compaction tree for the circuit under test and the faults simulator were developed in C and were executed on PC. Due to the large storage requirements of the data used in the simulations, the programs cannot handle big circuits. Furthermore, the CPU time spent in the overall simulation on an average size circuit such

as the Carry Look Ahead generator is rather large, and therefore, another platform such as workstations would be preferable for the simulations than conventional microcomputers.

CHAPTER 5

CONCLUSIONS

A test compaction technique for VLSI BIST has been presented in this thesis. The compaction tree is generated by dynamically calculating the detectable error probability estimates of the circuit under test. The error probability estimates were determined based on the Boolean Difference Method. The signature analysis was done by a Syndrome counter. An output data modification technique was applied in the study. This technique is called Modified Dynamic Space Compression method.

The Boolean Difference is a very simple and straightforward method to generate tests, and in our application, determined the number of detectable faults for the circuit under test. One of the limitations of using the Boolean Difference technique to derive the number of detectable faults in a circuit is that the method required a very high storage capacity. Furthermore, for a very complex circuit with a high density of gates, this method becomes unsuitable because of the computational complexity involved.

Faults simulation on the circuits under test was based on an exhaustive input patterns. Only single stuck-at faults were considered for the simulations. Faults were injected in every node of the circuit under test, excluding the compaction tree. The loss of information was found to be in the range of 0 to 10 percent.

For a given circuit under test, if a long sequence of test patterns is applied and the number of shared lines in the circuit under test is small, then the information loss induced by the space compressor is very small.

REFERENCES

[Aga'83] V. K. Agarwal, "Increasing effectiveness of built-in-testing by output data modification," Proc. of FTCS-13, June 1983, pp. 227-234

[Ale'70] I. Aleksander, "Introduction to Logic Circuit Theory," George G. Harrap & Co. Ltd. 1970

[Ale'61] Howard W. Alexander, "Elements of Mathematical Statistics," John Wiley & sons Inc. 1961

[Bar'87] Paul H. Bardell et al. , "Built-In Test for VLSI: Pseudorandom Techniques," John Willey & Sons, Inc. 1987

[Bha'89] Bhargab B. Bhattacharya and Sharad C. Seth, "Design of parity testable combinational circuits," IEEE Trans. on Computers, Vol. 38, No. 11, November 1989, pp. 1580-1584

[Jon'91] Wen-Ben Jone and Sunil R. Das, "Space compression method for built-in self-testing of VLSI circuits," Intl. Journal of Comp. Aided VLSI Design, Vol.3, 1991, pp.309-322

[Hay'76] John P. Hayes, "Transition count testing of combinational logic circuits," IEEE Trans. on Computers, Vol. C-25, No. 6, June 1976, pp. 613-620

[Hog'88] Robert V. Hogg and Elliot A. Tanis , "Probability and Statistical Inference," Macmillan Publishing Company, 1988

[Kar'87] M. Karpovsky and P. Nagvajara, "Optimal time and space compression of test responses for VLSI devices," IEEE International Test Conference, 1987, pp. 523-529

[Lee'76] Samuel C. Lee," Digital Circuits and Logic Design," Prentice Hall Inc., 1976

[Li'87] Yiu Kei Li and John P. Robinson, "Space compression methods with output data modification," IEEE Trans. on Computer Aided Design, Vol. CAD-6, No. 2, March 1987, pp. 290-294

[McC'85] E. J. McCluskey, "Built-in self-test techniques," IEEE Design & Test of Computers, Vol. 2, April 1985, pp. 21-28

[Red'88] Sudhakar M. Reddy, Kewal K. Saluja and Mark G. Karpovsky, "A data compression technique for built-in self-test," IEEE Trans. on Computers, Vol. 37, No. 9, September 1988

[Rob'87] John P. Robinson and Nirmal R. Saxena, "A unified view of test compression methods," IEEE Trans. on Computers, Vol. C-36, No. 1, January 1987, pp. 94-99

[Roth'79] Charles H. Roth Jr., "Fundamentals of Logic Design," West Publishing Co. 1979

[Sal'83] Kewal K. Saluja and M. Karpovsky, "Testing computer harware through data compression in space and time," Intl. Test Conference, 1983, pp. 83-88

[Sav'80] Jacob Savir, "Syndrome testable design of combinational circuits," IEEE Trans. on Computers, Vol.C-29, No. 6, June 1980, pp.442-451

[Sav'85] J. Savir and W. H. McAnney, "On the masking probability with one's count and transition count," Proc. of International Conference on Computer-Aided Design, 1985, pp. 111-113

[Sus'83] Alfred K. Susskind, "Testing by verifying Walsh coefficients," IEEE Trans. on Computers, Vol. C-32, No. 2, February 1983, pp. 198-201

[Zor'84] Y. Zorian and V. K. Agarwal, "Higher certainty of error coverage by output data modification," Intl. Test Conference, 1984, pp.140-147

[Yar'90] V. N. Yarmolik, "Fault Diagnosis of Digital Circuit," John Willey & sons, Inc. 1990

APPENDIX A

C Program to generate the test compaction tree for combinational digital circuit

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <math.h>
#define ESC  27
#define RETURN 13
#define AND  1
#define NAND 2
#define OR   3
#define NOR  4
#define XOR  5
#define XNOR 6
#define INVERTER 7
#define TRUE  1
#define FALSE 0
#define MAX_GATE_INPUT 8
#define MAX_PRIM_OUT 10
#define MAX_PRIM_IN 10
#define MAX_NUM_OF_GATE 50
#define MAX_NUM_OF_LINE 50
#define MAX_TEST_VECTOR 600

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

/*********************************************************************
        This structure is assigned to the variable which is used to
        memorize the inputs from the user.
*********************************************************************/
struct gate
   { short gate_number;
     short gate_type;
     short total_of_input;
     short inputname[MAX_GATE_INPUT];
     short outputname;
   };
struct gate *list;
/*********************************************************************
        This structure is defined to memorize Boolean Difference
        of every primary output against every line of the CUT.
*********************************************************************/

struct number
    {
        short element[MAX_NUM_OF_LINE];
        short num_of_element;
    };
```

```
struct dx
  {
   struct number index[MAX_NUM_OF_LINE];
  };
struct dx single[MAX_PRIM_OUT];

/*********************************************************************
        This structure is defined to memorize the informations
        of the output sequences of every primary output.
*********************************************************************/

struct seq
  {
   short flag; /* to indicate that the calculation of compaction is already performed */
   short value;
   short out_number;
   short weight;
  };
struct seq *out_seq;

/*********************************************************************
        BEGIN OF GLOBAL VARIABLES DEFINITION
*********************************************************************/

short totalnumberofgate; /* the total of gate in the circuit */
short gate_output[MAX_NUM_OF_GATE];     /* Temporary variable to check the gate enterring by the
user not to be duplicated */
short prim_in_number;   /*number of primary inputs in the CUT */
short prim_out_number;  /*number of primary outputs in the CUT */
short num_of_line;      /* total number of lines in the circuit, this variable will be updated after
performing every compact calculation */
short num_of_stage;     /*number of stage in the compaction tree*/
float seq_len;          /* the sequence bit wide*/
float S1;               /* probability estimated of single line errors */
float S2;               /* probability estimated of double line errors */
int *prim_out_array;  /* array of primitive outputs of  CUT, this array has the same index order as the
single array */
int *signature;         /* signature of the CUT on differents test vectors (exhausive test) */
short first_line;     /* number of line of the CUT before compaction */
short first_prim_out; /* number of primary outputs in the CUT before compaction */
FILE *in,*out;

/*********************************************************************
        END OF GLOBAL VARIABLES DEFINITION
*********************************************************************/
/* ==========================================================
  PROCEDURE : header
  Display  header of the mdsc program
```

```
═══════════════════════════════════════════════════════════════*/
int header ( void)
{ int x,y,choice=FALSE;
 char key;

clrscr();
window(0,0,70,20);
textcolor(WHITE);
textbackground(BLACK);
gotoxy(24,2);
cprintf("OTTAWA-CARLETON UNIVERSITY INSTITUTION");
textcolor(YELLOW);
gotoxy(25,10);
cprintf("TEST COMPACTION TECHNIQUE");
gotoxy(36,11);
cprintf("FOR");
gotoxy(29,12);
cprintf("BUILT-IN SELF-TEST");
gotoxy(37,13);
cprintf("IN");
gotoxy(31,14);
cprintf("VLSI CIRCUITS");
gotoxy(30,20);
cprintf(" Written by    : Huong T. Ho");
gotoxy(30,21);
cprintf(" Supervised by : Dr. S. R. Das");
gotoxy(30,22);
cprintf("        Date  : Nov 12th, 1993");
gotoxy(20,24);
textcolor(WHITE);
cprintf(" Hit escape key to continue... ");
if (ESC == (key = getche()))
  {
    choice = TRUE;
  }
return(choice);

}
/* ═══════════════════════════════════════════════════════════════
  FUNCTION : menu
  Display  menu
  ═══════════════════════════════════════════════════════════════*/
int menu( void)
{ int x,y,choice;

window(0,0,70,20);
textcolor(WHITE);
textbackground(BLACK);
clrscr();
```

```
gotoxy(5,3);
cprintf(" Program Name : DSC.CPP");
gotoxy(5,4);
cprintf(" Language    : 'C'");
gotoxy(5,5);
cprintf(" Written by  : Huong Thanh Ho");
gotoxy(31,9);
cprintf("  MENU");

window(10,10,70,50);
textbackground(BLUE);
gotoxy(22,12);
textcolor(YELLOW);
cprintf(" 0 :  Enter CUT                ");
gotoxy(22,13);
cprintf(" 1 :  Read CUT from file       ");
gotoxy(22,14);
cprintf(" 2 :  Read sequences from the CUT");
gotoxy(22,15);
cprintf(" 3 :  DSC data                 ");
gotoxy(22,16);
cprintf(" 4 :  Terminate Program        ");
textcolor(WHITE);

gotoxy(18,22);

cprintf(" Please select your choice....");
scanf("%d",&choice);
clrscr();
return(choice);
}

/* ═══════════════════════════════════════════════════════════════════
  FUNCTION : Init
  Intialization all of the global variables
  ═══════════════════════════════════════════════════════════════════ */
void init ( void)
{
 t_number = 0;
 totalnumberofgate = 0;
 seq_len = 0;
 prim_in_number = 0;
 prim_out_number = 0;
 num_of_line = 0;
 num_of_stage = 0;
 S1 = 0;
 S2 = 0;
 first_line = 0;
 first_prim_out = 0;
```

```
 list = (struct gate *) malloc(MAX_NUM_OF_GATE * sizeof(struct gate));
 if (!list)
   exit(1);
 out_seq = (struct seq *) malloc(MAX_PRIM_OUT * sizeof(struct seq));
 if (!out_seq)
   exit(1);
 prim_out_array = (int *) malloc (MAX_PRIM_OUT * sizeof(int));  /* array of primitive outputs of
CUT, this array has the same index order as the single array */
 if (!prim_out_array)
    exit(1);
 }


/* ================================================================
   FUNCTION : check
   Check the output of the gate of the CUT enterring by the user
   if it match with the gates of the CUT in the database. If it matched
   then that gate is already enterred so the user is asking to
   try again with different gate.
   ================================================================ */
int check ( int outputname)
 { int i,found = FALSE;
   for (i = 0; i< (totalnumberofgate); i++)
     { if (outputname == gate_output[i])
           found = TRUE;
      }
   return ( found);
 }

/* ================================================================
   FUNCTION : ReadCUT
   Read CUT from user.
   ================================================================ */
void ReadCUT ( void )
{int gatenumber,totalofinput,length,gatetype,output;
 int i,value[8],rt,secondtime= FALSE;
 char ch;
 in = fopen("cut.dat","w");
 if(in)
    {
      while (ESC != (ch = getche()))
            {
              printf(" Enter Gate number, Total of input, Gate type, Name output \n");
              scanf("%d%d%d%d",&gatenumber,&totalofinput,&gatetype,&output);
              if(secondtime)
                    { rt = check(output);
                      if(rt)
                        { do{
                              printf(" Your output's name is conflicted. Try again!!!\n");
```

```
                    printf(" Enter Gate number, Total of input, Gate type, Name output \n");
                    scanf("%d%d%d%d",&gatenumber,&totalofinput,&gatetype,&output);
                    rt = check(output);
                     }while( rt);
                  }
                } /* end of if statement */
              gate_output[totalnumberofgate ++] = output;
              fprintf(in," %d %d %d %d ",gatenumber,totalofinput,gatetype,output);
              for(i=0;i<totalofinput;i++)
                    {
                      printf(" Enter %d input name\n",i+1);
                      scanf("%d",&value[i]);
                      fprintf(in," %d",value[i]);
                    }
              fprintf(in," \n");
              secondtime = TRUE;
              printf(" Hit escape key to terminate program or hit any key to continue\n");
            } /* End of While statement */

  } /* End of If statement */
else
  printf(" Error opening for write\n");
fclose(in);
}
/* ═══════════════════════════════════════════════════════════════
  FUNCTION : WriteCUT
  write the CUT input by the user to the file CUT.dat.
  ═══════════════════════════════════════════════════════════════ */
void WriteCUT ( void )
{
 int i,j;
 in = fopen("cut.dat","rt");
 if(in)
   {
     fscanf(in,"%d",&totalnumberofgate);
     i = 0;
     while(!feof(in))
         {
           fscanf(in,"%d%d",&list[i].gate_number,&list[i].total_of_input);
           fscanf(in,"%d%d",&list[i].gate_type,&list[i].outputname);
           for (j = 0; j <list[i].total_of_input;j++)
             {
                  fscanf(in,"%d",&list[i].inputname[j]);
             }
           i++;
           if ( i == totalnumberofgate) break;
         }
   } /* End of If statement */
 else
```

```
  printf(" Error opening for write\n");
fclose(in);
}

/* ============================================================
  FUNCTION : Read CUT from file "CUT.file"
  ============================================================ */
void CUT_fr_file ( void )
{
int i,j;
in = fopen("cut.fil","rt");
if(in)
  {
    fscanf(in,"%d",&totalnumberofgate);
    i = 0;
    while(!feof(in))
        {
          fscanf(in,"%d%d",&list[i].gate_number,&list[i].total_of_input);
          fscanf(in,"%d%d",&list[i].gate_type,&list[i].outputname);
          for (j = 0; j <list[i].total_of_input;j++)
            {
                fscanf(in,"%d",&list[i].inputname[j]);
            }
          i++;
          if ( i == totalnumberofgate) break;
        }
  } /* End of If statement */
else
  printf(" Error opening for write\n");
fclose(in);
}


/* ============================================================
  FUNCTION : read_sequences
  Read sequences of every primary output of the CUT from user and
  save them in out_seq database.
  ============================================================ */
void read_sequences ( void )
{
int i,j,index,out_number,seq_value,flag,old_index;
char ch;
float wide;

for ( i = 0; i < prim_out_number ; i++)
  {
    out_seq[i].flag = FALSE;
    out_seq[i].value = 0;
```

```
    out_seq[i].out_number = 0;
    out_seq[i].weight = 0;
  }
index = 0;
old_index = 0;
if (totalnumberofgate != 0)
 {
   while (ESC != (ch = getche()))
        {
         printf(" Enter output number, sequences values in HEX, sequences bit wide \n");
         scanf("%d%x%f",&out_number,&seq_value,&wide);
         flag = FALSE;
         for ( i = 0; i < prim_out_number; i++)
           {
                if (prim_out_array[i] == out_number)
                  {
                    flag = TRUE;
                    break;
                  }
           }
         if (flag == FALSE)
          {
            printf("This is not a valid output, try again!");
          }
         else
          {
            for (j = 0; j < index ; j++)
                  {
                    if (out_seq[j].out_number == out_number)
                          {
                            old_index = index - 1;
                            index = j;
                          }
                    else
                          old_index = index;
                  }
            out_seq[index].value = seq_value;
            out_seq[index].out_number = out_number;
            index = old_index+1;
            if (index >= prim_out_number)
                  {
                    index = 0;
                  }
            seq_len = wide;
            printf(" Hit escape key to terminate program or hit any key to continue\n");
          }
        } /* End of While statement */
  }
 else
```

```
  printf("you must enter the CUT first or select the option terminate to quit the program\n");

}

/* ================================================================
  FUNCTION : primary_IO
  Determine the primary inputs and primary outputs of the circuit
  under test (CUT) and save them in prim_input table and prim_output
  table.

  ================================================================*/
void primary_IO ( void)
{ int i,j,k,l,output,flag,smallest;

  smallest = list[0].outputname;
  for (i = 1; i <totalnumberofgate;i++)
    {
        if (list[i].outputname < smallest)
         {
           smallest = list[i].outputname;
         }
    }
  prim_in_number = smallest - 1;
  prim_out_number = 0;
  l = 0;
  for (i=0; i<totalnumberofgate; i++)
    {
        flag = FALSE;
        output = list[i].outputname;
        for ( j=0; j<totalnumberofgate;j++)
          { if (j != i)
              { for ( k=0; k<list[j].total_of_input; k++)
                 {
                  if ((list[j].inputname[k]) == output)
                      {
                        flag = TRUE;
                        break;
                       }
                 }
              }
           if (flag)
              {
                break;
              }
          }
         if ( flag == FALSE )
           {
             prim_out_array[l++] = output;
             prim_out_number++;
```

```
            }
    }
  output = prim_out_array[0];
  for (i = 1; i < prim_out_number; i++)
    {
          if (output > prim_out_array[i])
            {
              output = prim_out_array[i];
            }
    }
  num_of_line = output - 1;
  first_line = num_of_line;
  first_prim_out = prim_out_number;

}

/* ================================================================
  FUNCTION : first_single_diff
  Determine the boolean difference (for single faults)
  of every primary outputs against every lines which
  are not a primary outputs of the CUT and save them
  in single table. This calcule is done only once when
  the CUT first be entered in.

  ================================================================*/
void first_single_diff ( void)
{ int i,j,k,l,m,n,p,q,temp,counter;

 /* Initializing the number of element in every index of single list */

 /****************start of initialization**************************/
 for (i =0 ; i < 10 ; i++)
   {
     for (j = 0; j < 20 ; j++)
          {
            single[i].index[j].num_of_element = 0;
          }
   }
 /****************end of initialization*****************************/

 for ( i=0; i<prim_out_number; i++)
   {
     for ( j=0; j<totalnumberofgate; j++)
          {
            if (list[j].outputname == prim_out_array[i])
              {
                    for ( k=0; k<list[j].total_of_input; k++)
                      {
                            temp = list[j].inputname[k];
```

```
                        m = 0;
                        for ( l=0; l<list[j].total_of_input; l++)
                          {
                            if ( l != k)
                                  {
                                    single[i].index[temp].element[m++] = list[j].inputname[l];
                                    single[i].index[temp].num_of_element++;
                                  }
                          }
                    }
                  break;
              }
        }
  for ( j = num_of_line; j > prim_in_number; j--)
        {
         if (single[i].index[j].num_of_element != 0)
           {
                for ( l=0; l<totalnumberofgate; l++)
                 {
                  if (list[l].outputname == j)
                        {
                         for ( m=0; m<list[l].total_of_input; m++)
                          {
                            temp = list[l].inputname[m];
                            p = 0;
                            for (q = 0; q<list[l].total_of_input; q++)
                                  {
                                   if (q != m)
                                     {
                                      single[i].index[temp].element[p++] = list[l].inputname[q];
                                     }
                                  }
                            for (k=0; k< single[i].index[j].num_of_element; k++)
                                  {
                                   single[i].index[temp].element[p++] = single[i].index[j].element[k];
                                  }
                            single[i].index[temp].num_of_element = p;
                          }
                         break;
                        }
                 }
           }
        }
  }
/* reset to zero all of num_of_elements of the primary output in the
 single array where the index are larger than number of lines in the
 circuit up  to primary output itselfs. */

 for (i = 0; i < prim_out_number; i++)
```

```
    {
        for (j =prim_out_array[i]; j > num_of_line; j--)
          {
            single[i].index[j].num_of_element = 0;
          }
    }

}

/* ================================================================
   FUNCTION : single_diff
   Count the number of the boolean difference (for single faults)
   which is different zero in the single Boolean array. This value
   will be used in the calculation of S1. These are the real single
   faults detectable of the circuit.
   ================================================================*/

float single_diff ( short  index_1, short index_2)
{ int i,counter;


  counter = 0;
  for (i = 1; i < prim_out_array[index_1] ; i++)
    {
        if (single[index_1].index[i].num_of_element != 0)
          {
           counter++;
          }
        else
          if (single[index_2].index[i].num_of_element != 0)
            {
                counter++;
            }
    }

 return (counter);

 }


 /* ================================================================
   FUNCTION : double_diff
   Count the number of the boolean difference (for double faults)
   which is different zero in the single Boolean array. This value
   is served in the calculation of S2. These values are the real double
   faults detectable of the circuit.
   ================================================================*/
```

```
float double_diff ( short  index_1, short index_2)
{ int i,counter;
float value;

 counter = 0;
 for (i = 1; i < prim_out_array[index_1] ; i++)
   {
        if ((single[index_1].index[i].num_of_element != 0)&&(single[index_2].index[i].num_of_element
!= 0))
          {
            counter++;
          }
   }
 value = counter/2 ;

return (value);
}


/* ═══════════════════════════════════════════════════════════════
  FUNCTION : calcul_tree_state
  Calculate the number of stage of the compaction tree (m).

  ═══════════════════════════════════════════════════════════════*/
void calcul_tree_stage ( void)
{  int i,j,k;


  i = prim_out_number;
  if (prim_out_number >= 2)
    {
        j = 0;
        while ( i > 2)
          {
            k = i;
            i =  k / 2;
            i = i + (k % 2);
            j++;
          }
        num_of_stage = j+1;
    }
  else
    {  num_of_stage = 0;
        printf("number of primary output must be greater than 2 \n");
        printf("enter the key read data to input the new circuit \n");
    }

}
```

```
/* ==========================================================
  FUNCTION : calculate_wn
  Calculate the number one in the nth sequence for every output
  sequences and save them in out_seq array.

  ==========================================================*/
void calculate_wn ( void)
{  int i,j,temp,mask,count;


  for ( j =0; j < prim_out_number; j++)
    {
        count = 0;
        mask = 1;
        for ( i=0; i <seq_len; i++)
          {
            temp = out_seq[j].value;
            temp = temp & mask;
            if (temp != 0)
                 {
                  count++;
                 }
            mask = mask * 2;
          }
        out_seq[j].weight = count;

    }

}

/* ==========================================================
  FUNCTION : factorial
  Calculate the factorial of a given number and return the result.
  This calculation is using to determine the modification factor
  S0.
              x! = x (x-1) (x-2)....2.1
  and
              0! = 1
  ==========================================================*/
float factorial (float input)

{

float result,scale;

if (input > 0)
  {
    result = 1;
    scale = 2;
```

```
    while(input-- > 1)
          {
            result = result * scale;
            scale++;
          }
  }
else
  {
    result = 1;
  }

return(result);
}

/* ==========================================================================
  FUNCTION : modif_for_gate
  Calculate the modification factor S0 for the logic gate AND, OR
  and XOR. The modification factor is calculated by the following
  equation:
                              S0 = ((n/2)! (n/2))! / (w! (n - w')!)
  compare the modification factor of the three logic gates and
  return the logic gate with the smallest modification factor S0.
  ========================================================================*/
int modif_for_gate (int val_l1, int val_l2)

{
int i,temp,gate,mask;
float S0_AND,S0_OR,S0_XOR,n_factor,count,w_AND,w_OR,
    w_XOR,wp_AND,wp_OR,wp_XOR;

/*********************************************************************
AND the two output values of line1 and line2 to get the output
value of the space compressor for the calculation of w'.
*********************************************************************/

gate = val_l1 & val_l2 ;
mask = 1;
count = 0;
for (i =0; i < seq_len; i++)
  {
    temp = gate & mask;
    mask = mask * 2;
    if ( temp != 0)
          {
            count++;
          }
  }
w_AND = factorial(count);
count = seq_len - count;
```

```
wp_AND = factorial(count);

/*********************************************************************
OR the two output values of line1 and line2 to get the output
value of the space compressor for the calculation of w'.
*********************************************************************/

gate = val_l1 | val_l2 ;
mask = 1;
count = 0;
for (i =0; i < seq_len; i++)
  {
    temp = gate & mask;
    mask = mask * 2;
    if ( temp != 0)
          {
            count++;
          }
  }
w_OR = factorial(count);
count = seq_len - count;
wp_OR = factorial (count);
/*********************************************************************
XOR the two output values of line1 and line2 to get the output
value of the space compressor for the calculation of w'.
*********************************************************************/

gate = val_l1 ^ val_l2 ;
mask = 1;
count = 0;
for (i =0; i < seq_len; i++)
  {
    temp = gate & mask;
    mask = mask * 2;
    if ( temp != 0)
          {
            count++;
          }
  }
w_XOR = factorial(count);
count = seq_len - count;
wp_XOR = factorial(count);

/*********************************************************************
calculation of modification factor for the three gates : AND, OR
and XOR.
*********************************************************************/

n_factor = (seq_len / 2);
```

```
n_factor = factorial(n_factor);

S0_AND = (n_factor * n_factor)/((w_AND) * (wp_AND));
S0_OR  = (n_factor * n_factor)/((w_OR) * (wp_OR));
S0_XOR = (n_factor * n_factor)/((w_XOR) * (wp_XOR));

if ((S0_AND <  S0_OR) && (S0_AND < S0_XOR))
  {
    gate = AND;
  }
else
  {
    if ((S0_OR < S0_AND) && (S0_OR < S0_XOR))
          {
            gate = OR;
          }
     else
          {
            gate = XOR;
          }
  }

return(gate);
}


/* ================================================================
  FUNCTION : determine_gate
  If it is not the last stage (flag = FALSE) then we just have to
  pick the logic gate with the highest detectable error probability
  estimayed E. If it is the last stage then we have to determine
  the modification factor (S0) and pick the logic gate with the
  smallest modification factor.
 ================================================================*/
int determine_gate ( float AND_gate,float OR_gate,float XOR_gate,int flag,int val_l1, int val_l2)


{
int i,temp;

if (flag == FALSE)
  {
    if ((AND_gat > OR_gate) && (AND_gate > XOR_gate))
          {
            temp = AND;
          }
     else
          {
            if ((OR_gate > AND_gate) && (OR_gate > XOR_gate))
```

```
            {
                temp = OR;
            }
          else
            temp = XOR;
        }
  }
else
  {
    if (( AND_gate > OR_gate) && (AND_gate > XOR_gate))
          {
           temp = AND;
          }
    else
          if ((OR_gate > AND_gate) && (OR_gate > XOR_gate))
            {
              temp = OR;
            }
          else
            if ((XOR_gate > AND_gate) && (XOR_gate > OR_gate))
                  {
                    temp = XOR;
                  }
            else
                  {
                    temp = modif_for_gate(val_l1,val_l2);
                  }
  }

return(temp);

}

/* ================================================================
  FUNCTION : calculate_Rn
  Calculate the number of single (double) line errors detected
  at the output of gate i and j (correspond to the two gates l1
  and l2). These calculation repeat for the three logic gates AND,
  OR and XOR. Calculate the detectable error probability estimated E
  for each of the three logic gates and return the gate which has
  the highest detectable error probability estimated E.
================================================================*/
int calculate_Rn ( int l1,int l2,int flag,float S1_val, float S2_val)

{
int i,line1,line2,count,temp,mask,index;
float R1,R2,AND_gate,OR_gate,XOR_gate;

index = t_number * 8;
```

```
for (i =0; i < prim_out_number; i++)
  {
    if (out_seq[i].out_number == 11)
          {
            line1 = out_seq[i].value;
            break;
          }
  }

for (i =0; i < prim_out_number; i++)
  {
    if (out_seq[i].out_number == 12)
          {
            line2 = out_seq[i].value;
            break;
          }
  }
/*****************************************************
Calculate for OR gate by counting  the total number
of zeros in 11 and 12. This number equal R1.
*****************************************************/

R1 = 0;
mask = 1;
for ( i=0; i <seq_len; i++)
  {
    temp = line1;
    temp = temp & mask;
    if (temp == 0)
          {
           R1++;
          }
    mask = mask * 2;
  }

mask = 1;
for ( i=0; i <seq_len; i++)
  {
    temp = line2;
    temp = temp & mask;
    if (temp == 0)
          {
           R1++;
          }
    mask = mask * 2;
  }

/*****************************************************
Calculate R2 for OR gate by XOR the lines line1 and
```

```
line2, the total number of zeros in the result will
determine the value of R2.

*****/

R2 = 0;
mask = 1;
for ( i=0; i <seq_len; i++)
  {
    temp = line1 ^ line2;
    temp = temp & mask;
    if (temp == 0)
          {
           R2++;
          }
    mask = mask * 2;
  }

OR_gate = (S1 * (R1/(2*seq_len))) + (S2 * (R2/seq_len));
t_array[index++] = R1;
t_array[index++] = R2;

/**********end of OR gate**********/
/*****************************************************
Calculation for AND gate. First, the lines line1
and line2 will be ANDed, the number of one's in the
result will be multiplied by 2 and stored in the
variable temp. Then the lines line1 and line2 will
be XOR and the total number of one's in this
calculation plus the variable temp determine the
value of R1.
*****************************************************/


R1 = 0;
mask = 1;
for ( i=0; i <seq_len; i++)
  {
    temp = line1 & line2;
    temp = temp & mask;
    if (temp != 0)
          {
           R1++;
          }
    mask = mask * 2;
  }
mask = 1;
for ( i=0; i <seq_len; i++)
  {
```

```
    temp = line1 ^ line2;
    temp = temp & mask;
    if (temp != 0)
          {
           R1++;
          }
    mask = mask * 2;
  }

/*****************************************************
To determine the value of R2 for AND gate, the lines
line1 and line2 must be XOR together. The total number
of zero's in the result will determine the value of R2.
*****************************************************/

R2 = 0;
mask = 1;
for ( i=0; i <seq_len; i++)
  {
    temp = line1 ^ line2;
    temp = temp & mask;
    if (temp == 0)
          {
           R2++;
          }
    mask = mask * 2;
  }

AND_gate = (S1 * (R1/(2*seq_len))) + (S2 * (R2/seq_len));

t_array[index++] = R1;
t_array[index++] = R2;

/***********************************************************
Calculations for XOR gate. The detectable error probability
estimate for a XOR gate is defined by S1, i. ·:, E(XOR) = S1.
***********************************************************/

XOR_gate = S1;
temp = determine_gate(AND_gate,OR_gate,XOR_gate,flag,line1,line2);

t_array[index++] = S1_val;
t_array[index++] = S2_val;
t_array[index++] = S1;
t_array[index] = S2;
t_number++;

return(temp);
```

```
}
/*********************************************************
        PROCEDURE update_list
        Update the list array by adding the new gate of
        the compact tree which is used to merge the two
        gates.

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

void update_list ( short gate, short out_1, short out_2, short largest)
{
list[totalnumberofgate].gate_type = gate;
list[totalnumberofgate].total_of_input = 2;
list[totalnumberofgate].inputname[0] = out_1 ;
list[totalnumberofgate].inputname[1] = out_2 ;
list[totalnumberofgate].outputname = largest + 1;
list[totalnumberofgate].gate_number = totalnumberofgate + 1;
totalnumberofgate++;

}

/*********************************************************
        PROCEDURE update_single
        Update the single array. The two array of out_1
        and out_2 will be merged and save their result
        in the single array.
*********************************************************/

void update_single (short out_1, short out_2, short largest)
{
int count,small,large,i,index,index_1,index_2,array_index;
struct dx *single_temp;
single_temp =  (struct dx *) malloc(MAX_PRIM_OUT * sizeof(struct dx));
 if (!out_seq)
 exit(1);

for ( i = 0; i < prim_out_number; i++)
 {
   if (prim_out_array[i] == out_1)
        {
          index_1 = i;
        }
   else
        {
          if (prim_out_array[i] == out_2)
            {
                index_2 = i;
            }
        }
```

```
 }

if (index_1 > index_2)
  {
   small = index_2;
   large = index_1;
  }
else
  {
    small = index_1;
    large = index_2;
  }


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

        Initialize the single_temp array.
*********************************************************/

for ( i=0; i < largest ; i++)
  {
    single_temp[0].index[i].num_of_element = 0;
  }

single_temp[0] = single[large];
array_index = prim_out_array[small];

for (i =0; i < array_index; i++)
  {
    if (single_temp[0].index[i].num_of_element ==0)
          {
            if (single[small].index[i].num_of_element != 0)
              {
                    single_temp[0].index[i] = single[small].index[i];

              }
          }

  }
single_temp[0].index[out_1].num_of_element = 1;
single_temp[0].index[out_1].element[0] = out_2;

single_temp[0].index[out_2].num_of_element = 1;
single_temp[0].index[out_2].element[0] = out_1;

single[prim_out_number] = single_temp[0];

/* reset to zero all of num_of_element in this array
   where the index is larger than the largest between
```

```
  out_1 and out_2 upto the value of largest + 1 */

if (out_1 > out_2)
  {
    index = out_1;
  }
else
   index = out_2;
for ( i= largest + 1; i > index; i--)
   {
     single[prim_out_number].index[i].num_of_element = 0;
   }


prim_out_array[prim_out_number] = largest + 1 ;
num_of_line = num_of_line + 2;
free(single_temp);

}
/*************************************************************
          PROCEDURE update_outseq
          Update the output sequences array by merging the
          sequences of the two gates out_1 and out_2 into
          the new gate.
*************************************************************/

void update_outseq ( short gate, short out_1, short out_2, short largest)
{
int temp,out_number,value,weight,i,index_1,index_2,mask;

for (i =0; i < prim_out_number; i++)
   {
     if (out_seq[i].out_number == out_1)
          {
            index_1 = i;
          }
     else
          {
            if (out_seq[i].out_number == out_2)
              {
                    index_2 = i;
              }
          }
   }
 out_number = largest + 1 ;
 if (gate == AND)
   {
     value = out_seq[index_1].value & out_seq[index_2].value;
   }
```

```
else
  {
    if (gate == OR)
          {
            value = out_seq[index_1].value | out_seq[index_2].value;
          }
    else /* XOR */
          {
            value = out_seq[index_1].value ^ out_seq[index_2].value;
          }

  }
 mask = 1;
 weight = 0;
 for (i=0; i <seq_len ; i++)
   {
     temp = value & mask;
     if (temp != 0)
           {
             weight++;
           }
      mask = mask * 2;
   }


 out_seq[prim_out_number].value = value;
 out_seq[prim_out_number].out_number = out_number;
 out_seq[prim_out_number].weight = weight;
 out_seq[prim_out_number].flag = FALSE;
 prim_out_number++;

 }

 /*************************************************************
          PROCEDURE last_stage
          Calculate the logic function for the last stage of
          the compression circuit. The logic function with
          the highest value of detectable error probability
          will be considered. But if two or three of them
          have the same value then the logic function with
          the smallest modification factor (S0) will be taken.

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


 void last_stage (void)

 {
 int i,j,largest,index_1,index_2,wn_1,wn_2,outgate,flag;
```

```
float S1_val,S2_val;

for ( i = 0; i < prim_out_number; i++)
  {
    if (out_seq[i].flag == FALSE)
          {
            out_seq[i].flag = TRUE;
            wn_1 = out_seq[i].out_number;
            index_1 = i;
            break;
          }
  }
for ( i = 0; i < prim_out_number; i++)
  {
    if (out_seq[i].flag == FALSE)
          {
            wn_2 = out_seq[i].out_number;
            index_2 = i;
            break;
          }
  }

flag = TRUE;
S1_val = single_diff(index_1,index_2);
S2_val = double_diff(index_1,index_2);
S1 = S1_val / (S1_val + S2_val);
S2 = S2_val / (S1_val + S2_val);
outgate = calculate_Rn(wn_1,wn_2,flag,S1_val,S2_val);
largest = prim_out_array[0];
for (j = 1; j < prim_out_number; j++)
  {
    if (prim_out_array[j] > largest)
          {
            largest = prim_out_array[j];
          }
  }
update_list(outgate,wn_1,wn_2,largest);
printf("%d %d %d",wn_1,wn_2,outgate);
num_of_line = num_of_line + 2;
}
/* ============================================================
  FUNCTION : dsc
  Main body of DSC.
============================================================*/

void dsc ( void)
{
int i,j,k,step,N,index_1,index_2,wn_1,wn_2,largest,second,outgate,flag,
   index,temp[MAX_PRIM_OUT*2],index_temp,counter,mem_prim_out;
```

```
float S1_val,S2_val;

mem_prim_out = prim_out_number;
if (num_of_stage > 0)
  {
    if (num_of_stage > 1)
        {
          for (i = 0; i <(num_of_stage - 1) ; i++)
            {
                step = mem_prim_out;
                N = 0;
                index_temp = 0;
                counter = 0;
                while (N < (step-1))
                  {
                    index_1 = 0;
                    largest = 0;
                    for (j = 0; j < prim_out_number; j++)
                        {
                          if ((out_seq[j].weight > largest) && (out_seq[j].flag == FALSE))
                            {
                                largest = out_seq[j].weight;
                                index_1 = j;
                            }
                        }
                    wn_1 = out_seq[index_1].out_number;
                    out_seq[index_1].flag = TRUE;
                    second = 0;
                    index_2 = 0;
                    for (j=0; j < prim_out_number; j++)
                        {
                          if ((out_seq[j].weight > second) && (j != index_1) && (out_seq[j].flag ==
FALSE))
                            {
                                second = out_seq[j].weight;
                                index_2 = j;
                            }
                        }
                    wn_2 = out_seq[index_2].out_number;
                    out_seq[index_2].flag = TRUE;

/*****************************************************************
  assumming that the probability of single stuck at 0(1) accurring at
  gate wn_1 equal to the probability of single stuck at 0(1) occuring
  at gate wn_2. Therefore, select wn_q or wn_2 for calculation is not
  improtant. Here, we choose wn_1 for the calculation.
*****************************************************************/

                    for ( j = 0; j < prim_out_number; j++)
```

```
        {
         if (prim_out_array[j] == wn_1)
          {
               break;
          }
        }
   index_1 = j;
   for ( j = 0; j < prim_out_number; j++)
        {
         if (prim_out_array[j] == wn_2)
          {
               break;
          }
        }
   index_2 = j;
   largest = prim_out_array[0];
   for (j = 1; j < prim_out_number; j++)
        {
         if (prim_out_array[j] > largest)
          {
               largest = prim_out_array[j];
          }
        }
   S1_val = single_diff(index_1,index_2);
   S2_val = double_diff(index_1,index_2);

   S1 = S1_val / (S1_val + S2_val);
   S2 = S2_val / (S1_val + S2_val);
   flag = FALSE;
   outgate = calculate_Rn(wn_1,wn_2,flag,S1_val,S2_val);

   temp[index_temp++] = wn_1;
   temp[index_temp++] = wn_2;
   temp[index_temp++] = outgate;
   temp[index_temp++] = largest + counter;
   counter++;
   mem_prim_out--;
   N = N + 2;
  }
k = 0;
for ( j = 0; j < counter; j++)
  {
    wn_1   = temp[k++];
    wn_2   = temp[k++];
    outgate = temp[k++];
    largest = temp[k++];
    update_list(outgate,wn_1,wn_2,largest);
    update_single(wn_1,wn_2,largest);
    update_outseq(outgate,wn_1,wn_2,largest);
```

```
                }
            }

            last_stage();

        }
    else
        {
            index_1 = 0;
            index_2 = 1;
            S1_val = single_diff(index_1,index_2);
            S2_val = double_diff(index_1,index_2);

            S1 = S1_val / (S1_val + S2_val);
            S2 = S2_val / (S1_val + S2_val);
            flag = TRUE;
            outgate = calculate_Rn(prim_out_array[0],prim_out_array[1],flag,S1_val,S2_val);
            if (prim_out_array[0] > prim_out_array[1])
              {
                    largest = prim_out_array[0];
              }
            else
              {
                    largest = prim_out_array[1];
              }
            update_list(outgate,prim_out_array[0],prim_out_array[1],largest);
            num_of_line = num_of_line + 2;
        }
  }
else
  {
    printf("can not process data with the number of output < 1 !");
  }

}

/* ============================================================
  FUNCTION : Printdata
  Display data on the screen.
  ============================================================*/

void PrintData ( void)
{  int i=0,j,k;
  out = fopen("circuit","w");
  if (out)
    {
          while(i < totalnumberofgate)
            {
```

```
            fprintf(out,"\ngate number%2d number of
input%2d",list[i].gate_number,list[i].total_of_input);
            fprintf(out," gate type%2d output name%2d",list[i].gate_type,list[i].outputname);
            for (j = 0; j <list[i].total_of_input;j++)
                  {
                    fprintf(out," input%2d",list[i].inputname[j]);
                  }
            i++;
           }
         i = 0;
         while(i<t_number)
           {
            j = i * 8;
            fprintf(out,"\n");
            for (k = 0; k < 8; k++)
                  {
                    fprintf(out,"%2d ",t_array[j++]);
                  }
            i++;
           }

   }
  else
   {
        printf ("error opening for write \n");
   }
 fclose(out);
}

/* ============================================================
                        MAIN PROGRAM
   ============================================================ */

void main ( void)
{ int rt,flag=FALSE;
  char ch='y';
  printf("\n\n\n\n\n");
  init();
  while( ch != 'q')
  {
  if (!flag)
    {
          rt = header();
          if (!rt)
            {
             do{
                    rt = header();
               }while (!rt);
            }
          flag = TRUE;
```

```
  }
rt = menu();
switch (rt)
  {
        case 0 :
                ReadCUT();
                WriteCUT();
                primary_IO();
                break;
        case 1 :
                CUT_fr_file();/*read CUT from file CUT.file*/
                primary_IO();
                break;
        case 2 :
                first_single_diff();
                calcul_tree_stage();  /**/
                read_sequences();
                calculate_wn();   /* these functions are called here
                                     for the CUT entered by user
                                     and will be called in DSC() to
                                     calculate for the compaction tree*/
                break;
        case 3 :
                dsc();
                PrintData();
                break;
        default: printf(" Not selection. Enter 'q' to terminate program\n");
                ch = getche();
                break;
   }
  }
free (list);
free (out_seq);
free (prim_out_array);
}
```

APPENDIX B

C Program for caculation of faults missed in the circuit under test before and after compaction

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <math.h>
#define AND  1
#define NAND 2
#define OR   3
#define NOR  4
#define XOR  5
#define XNOR 6
#define INVERTER 7
#define SPECIAL 8
#define TRUE  1
#define FALSE 0
#define MAX_GATE_INPUT 8
#define MAX_PRIM_OUT 10
#define MAX_PRIM_IN 10
#define MAX_NUM_OF_GATE 80
#define MAX_NUM_OF_LINE 120
#define MAX_TEST_VECTOR 600

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

/*********************************************************************
        This structure is assigned to the variable which is used to
        memorize the inputs from the user.
*********************************************************************/
struct gate
   { short gate_number;
     short gate_type;
     short total_of_input;
     short inputname[MAX_GATE_INPUT];
     short outputname;
   };
struct gate *list;

/*********************************************************************
        This structure is defined to memorize the data at the output
        of every gates of the CUT corresponding to the applied test
        vector.  The number of test vectors applies to the CUT is in
        order of 2**n (exhausive test).
*********************************************************************/

struct test
   {
    short test_input[MAX_GATE_INPUT];
```

```
    short test_output;
  };
struct test test_array[MAX_NUM_OF_GATE];

/*****************************************************************
        BEGIN OF GLOBAL VARIABLES DEFINITION
*****************************************************************/

short totalnumberofgate; /* the total of gate in the circuit */
short prim_in_number;   /*number of primary inputs in the CUT */
short prim_out_number;  /*number of primary outputs in the CUT */
short num_of_line;      /* total number of lines in the circuit, this variable will be updated after
performing every compact calculation */
int *prim_out_array;  /* array of primitive outputs of CUT, this array has the same index order as the
single array */
int *signature;       /* signature of the CUT on differents test vectors (exhausive test) */
int *fault_sig;       /* signature of the CUT simulating with faults on differents test vectors (exhausive
test) */
int *test_a;          /* temp array to store test results*/
short num_of_test_vector;/*total number of test vector to be applied to the CUT */
int dsc_fault_mis;    /*faults missed by DSC */
int undetect_f;     /* undetectable faults in CUT before compaction */
FILE *in,*out;

/* ================================================================
  FUNCTION : Init
  Intialization all of the global variables
  ================================================================ */
void init ( void)
{
 prim_in_number = 0;
 prim_out_number = 0;
 num_of_line = 0;
 dsc_fault_mis = 0;
 undetect_f = 0;
 list =  (struct gate *) malloc(MAX_NUM_OF_GATE * sizeof(struct gate));
 if (!list)
  exit(1);
 prim_out_array = (int *) malloc (MAX_PRIM_OUT * sizeof(int));  /* array of primitive outputs of
CUT, this array has the same index order as the single array */
 if (!prim_out_array)
   exit(1);
 signature = (int *) malloc (MAX_TEST_VECTOR * sizeof(int));
 if (!signature)
   exit(1);
 fault_sig = (int *) malloc (MAX_TEST_VECTOR * sizeof(int));
 if (!fault_sig)
   exit(1);
 test_a = (int *) malloc ((2*MAX_NUM_OF_LINE) * sizeof(int));
```

```
 if (!test_a)
 exit(1);
}


/* ==========================================================
  FUNCTION : Read CUT from file "CUT.fil"
  ========================================================== */
void CUT_fr_file ( void )
{
 int i,j;
 in = fopen("cut.fil","rt");
 if(in)
   {
     fscanf(in,"%d",&totalnumberofgate);
     i = 0;
     while(i < totalnumberofgate)
           {
             fscanf(in,"%d%d",&list[i].gate_number,&list[i].total_of_input);
             fscanf(in,"%d%d",&list[i].gate_type,&list[i].outputname);
             for (j = 0; j <list[i].total_of_input;j++)
                   {
                     fscanf(in,"%d",&list[i].inputname[j]);
                    }
             i++;
           }
     fscanf(in,"%d",&prim_in_number);
     fscanf(in,"%d",&num_of_line);
     fscanf(in,"%d",&prim_out_number);
     for(i=0; i<prim_out_number; i++)
           {
             fscanf(in,"%d",&prim_out_array[i]);
           }
   } /* End of If statement */
 else
   printf(" Error opening for write\n");
 fclose(in);
}
/* ==========================================================
  FUNCTION : Read CUT from file "compact.fil"
  ========================================================== */
 void compact_fr_file ( void )
 {
  int i,j;
  in = fopen("compact.fil","rt");
  if(in)
    {
      fscanf(in,"%d",&totalnumberofgate);
      i = 0;
```

```
    while(i < totalnumberofgate)
          {
            fscanf(in,"%d%d",&list[i].gate_number,&list[i].total_of_input);
            fscanf(in,"%d%d",&list[i].gate_type,&list[i].outputname);
            for (j = 0; j <list[i].total_of_input;j++)
                  {
                    fscanf(in,"%d",&list[i].inputname[j]);
                  }
            i++;
          }
    fscanf(in,"%d",&prim_in_number);
    fscanf(in,"%d",&num_of_line);
    fscanf(in,"%d",&prim_out_number);
  } /* End of If statement */
else
  printf(" Error opening for write\n");
fclose(in);
}

/* ================================================================
  FUNCTION : Init_test_value
  Initialisation test inputs and test outputs of gates in the CUT.
  ================================================================*/

void init_test_value ( void)
{ int i,j;

  for(i=0; i<totalnumberofgate;i++)
    {
          for(j=0; j<list[i].total_of_input;j++)
            {
              test_array[i].test_input[j] = 0xF;
            }
          test_array[i].test_output = 0xF;
    }

}

/* ================================================================
  FUNCTION : update_test_array_special
  writing the test output of gate i to the test input of the gate
  whose one of the input is the output of gate i.

  ================================================================*/

void update_test_array_special ( int index)
{ int i,j,k;

for (i=0; i<totalnumberofgate; i++)
```

```
  {
    for (j=0; j<list[i].total_of_input; j++)
          {
            for (k=0; k<list[index].total_of_input; k++)
              {
                    if (list[i].inputname[j] == list[index].inputname[k])
                      {
                            test_array[i].test_input[j] = test_array[index].test_input[k];
                      }
              }
          }
  }

}
/* ==========================================================
  FUNCTION : update_test_array_special_f
  writing the test output of gate i to the test input of the gate
  whose one of the input is the output of gate i.

  ==========================================================*/

void update_test_array_special_f ( int index)
{  int i,j,k;

for (i=0; i<totalnumberofgate; i++)
  {
    for (j=0; j<list[i].total_of_input; j++)
          {
            for (k=0; k<list[index].total_of_input; k++)
              {
                    if (list[i].inputname[j] == list[index].inputname[k])
                      {
                            if (test_array[i].test_input[j] == 0xF)
                              {
                                test_array[i].test_input[j] = test_array[index].test_input[k];
                              }
                      }
              }
          }
  }

}


/* ==========================================================
  FUNCTION : update_next

  ==========================================================*/
```

```
void update_next ( int index)
{  int j,k,l,done,yes,flag;

  flag = FALSE;
  while (!flag)
   {
        yes = FALSE;
        done = FALSE;
        for (j=0; j< totalnumberofgate; j++)
          {
            for (k=0; k< list[index].total_of_input; k++)
                  {
                    if (list[j].outputname == list[index].inputname[k])
                      {
                            yes = TRUE;
                            test_array[j].test_output = test_array[index].test_input[k];
                            for (l=0; l < list[j].total_of_input; l++)
                              {
                                    test_array[j].test_input[l] = test_array[index].test_input[k];
                              }
                            index = j;
                            update_test_array_special(index);
                            done = TRUE;
                            break;
                      }
                  }
              if(done){
                    break;
              }
            }
          if (!yes)
            {
              flag = TRUE;
            }
    }

}

/* ==========================================================
  FUNCTION : update_next_f

  ==========================================================*/

void update_next_f ( int index)
{  int j,k,l,done,yes,flag;

  flag = FALSE;
```

```
while (!flag)
 {
       yes = FALSE;
       done = FALSE;
       for (j=0; j< totalnumberofgate; j++)
         {
           for (k=0; k< list[index].total_of_input; k++)
                 {
                   if (list[j].outputname == list[index].inputname[k])
                     {
                           yes = TRUE;
                           if (test_array[j].test_output == 0xF)
                             {
                                   test_array[j].test_output = test_array[index].test_input[k];
                             }
                           for (l=0; l < list[j].total_of_input; l++)
                             {
                                   if (test_array[j].test_input[l] == 0xF)
                                     {
                                       test_array[j].test_input[l] = test_array[index].test_input[k];
                                     }
                             }
                           index = j;
                           update_test_array_special_f(index);
                           done = TRUE;
                           break;
                     }
                 }
            if(done){
                  break;
            }
           }
        if (!yes)
          {
           flag = TRUE;
          }
   }

}

/* ============================================================
  FUNCTION : update_test_array
  writing the test output of gate i to the test input of the gate
  whose one of the input is the output of gate i.

  ============================================================*/

void update_test_array ( int index)
{  int i,j,k,l,flag,yes,done;
```

```
 i = 0;
 while(i <totalnumberofgate)
   {
     if(list[i].outputname == list[index].outputname)
       {
         for(j=0; j<list[i].total_of_input; j++)
                  {
                    test_array[i].test_input[j] = test_array[index].test_output;
                  }
            test_array[i].test_output =  test_array[index].test_output;
            update_test_array_special(i);
            update_next(i);
          }
         else{
           for(j=0; j<list[i].total_of_input;j++)
             {
               if (list[i].inputname[j] == list[index].outputname)
                     {
                          test_array[i].test_input[j] = test_array[index].test_output;
                       break;
                     }
             }
           }
         i++;
   }
}
/* ============================================================
  FUNCTION : update_test_array_f

  writing the test output of gate i to the test input of the gate
  whose one of the input is the output of gate i.

  ============================================================*/

void update_test_array_f ( int index)
{  int i,j,k,l,flag,yes,done;

  i = 0;
  while(i <totalnumberofgate)
    {
      if(list[i].outputname == list[index].outputname)
        {
          for(j=0; j<list[i].total_of_input; j++)
                  {
                    if (test_array[i].test_input[j] == 0xF)
                          {
                            test_array[i].test_input[j] = test_array[index].test_output;
                          }
```

```
                }
            if (test_array[i].test_output == 0xF)
                {
                  test_array[i].test_output =  test_array[index].test_output;
                }
            update_test_array_special_f(i);
            update_next_f(i);
          }
        else{
          for(j=0; j<list[i].total_of_input;j++)
            {
              if (list[i].inputname[j] == list[index].outputname)
                  {
                      if (test_array[i].test_input[j] == 0xF)
                        {
                          test_array[i].test_input[j] = test_array[index].test_output;
                        }
                      break;
                  }
            }
          }
        i++;
    }
}


/* ================================================================
  FUNCTION : compare_test
  compare signature to fault_signature and determine number of
  faults missed.

  ================================================================*/

int compare_test ( void)
{  int i,fault_mis,flag;

  flag = FALSE;
  for (i=0; i< num_of_test_vector; i++)
    {
        if (signature[i] != fault_sig[i])
          {
            flag = TRUE;
            break;
          }
    }
  if (flag)
    {
        fault_mis = 0;
    }
```

```
  else
    fault_mis = 1;
return(fault_mis);
}

/* ═══════════════════════════════════════════════════════════
  FUNCTION : output_gate
  Determine the output data for the gate corresponding to the given
  input data.
  ═══════════════════════════════════════════════════════════*/

void output_gate ( int index)
{  int i,flag,value,mask;

  switch (list[index].gate_type)
    {
        case AND :

                flag = TRUE;
                for (i=0; i<list[index].total_of_input; i++)
                  {
                    if(test_array[index].test_input[i] == 0)
                          {
                            test_array[index].test_output = 0;
                            flag = FALSE;
                            break;
                          }
                  }
                if(flag)
                  {
                    test_array[index].test_output = 1;
                  }
                break;
        case OR :
                flag = TRUE;
                for (i=0; i<list[index].total_of_input; i++)
                  {
                    if(test_array[index].test_input[i] == 1)
                          {
                            test_array[index].test_output = 1;
                            flag = FALSE;
                            break;
                          }
                  }
                if(flag)
                  {
                    test_array[index].test_output = 0;
                  }
                break;
```

```
case NAND :
        flag = TRUE;
        for (i=0; i<list[index].total_of_input; i++)
          {
            if(test_array[index].test_input[i] == 0)
                  {
                    test_array[index].test_output = 1;
                    flag = FALSE;
                    break;
                  }
          }
        if(flag)
          {
            test_array[index].test_output = 0;
          }
        break;
case NOR :
        flag = TRUE;
        for (i=0; i<list[index].total_of_input; i++)
          {
            if(test_array[index].test_input[i] == 1)
                  {
                    test_array[index].test_output = 0;
                    flag = FALSE;
                    break;
                  }
          }
        if(flag)
          {
            test_array[index].test_output = 1;
          }
        break;
case XOR :
        value = 0;
        for (i=0; i<list[index].total_of_input; i++)
          {
            if(test_array[index].test_input[i]  != 0)
                  {
                    value = value + 1;
                  }
          }

        if (value == 1)
          {
            test_array[index].test_output = 1;
          }
        else
          {
            test_array[index].test_output = 0;
```

```
                }

                break;
        case XNOR :
                mask = 1;
                value = 0;
                for (i=0; i<list[index].total_of_input; i++)
                  {
                    if((test_array[index].test_input[i] & mask) != 0)
                          {
                            value++;
                          }
                  }
                if ((value != 0) && ((value & mask)!=0))
                  {
                    test_array[index].test_output = 0;
                  }
                else
                  {
                    test_array[index].test_output = 1;
                  }
                break;
        case INVERTER :
                if (test_array[index].test_input[0] == 0)
                  {
                    test_array[index].test_output = 1;
                  }
                else
                  {
                    test_array[index].test_output = 0;
                  }
                break;
        default:
                break;
   }
}

/* ============================================================
  FUNCTION : output_circuit
  Determine the data at the output of the circuit corresponding to
  a given test vector.
  ============================================================*/

int output_circuit (int largest, int f_flag)
{  int i,j,complete,ok,value;

  complete = FALSE;
  while (!complete)
```

```
  {
      for (i=0; i<totalnumberofgate; i++)
        {
          ok = TRUE;
          for (j=0; j<list[i].total_of_input; j++)
                {
                  if(test_array[i].test_input[j] == 0xF)
                        {
                          ok = FALSE;
                          break;
                        }
                }
          if ((ok == TRUE) && (test_array[i].test_output == 0xF))
                {
                  output_gate(i);
                  if (! f_flag)
                        {
                          update_test_array(i);
                        }
                  else
                        {
                          update_test_array_f(i);
                        }
                }
          if(list[i].outputname == largest)
                {
                  if (test_array[i].test_output != 0xF)
                        {
                          value = test_array[i].test_output;
                          complete = TRUE;
                          break;
                        }
                }
        }
    }
  return(value);
}
/* ================================================================
  FUNCTION : get_sig
  get signature of the CUT (after compaction) on diffferents test
  vectors (2**n).
  ================================================================*/

int get_sig ( int test_vector, int largest)
{  int i,j,mask,bit,value,index,k,l,f_flag;

  for (i=0; i<totalnumberofgate; i++)
    {
        if (list[i].outputname <= prim_in_number)
```

```
            {
              mask = pow(2,(list[i].outputname-1));
              bit = test_vector & mask;
              if (bit != 0)
                    {
                      bit = 1;
                    }
              test_array[i].test_output = bit;
              for (j = 0; j < list[i].total_of_input; j++)
                    {
                      test_array[i].test_input[j] = bit;
                    }
              update_test_array_special(i);
              index = i;
              update_next(index);
            }
          else{
            for (j=0; j<list[i].total_of_input; j++)
              {
                    if (list[i].inputname[j] <= prim_in_number)
                      {

                            mask = pow(2,(list[i].inputname[j]-1));
                            bit = test_vector & mask;
                            if (bit != 0)
                              {
                                test_array[i].test_input[j] = 1;
                              }
                            else
                              {

                                test_array[i].test_input[j] = 0;
                              }
                      }
              }
            }
      }
    f_flag = FALSE;
    value = output_circuit(largest,f_flag);
    return(value);
}

/* ================================================================
    FUNCTION : get_fault_sig
    get signature of the CUT (after compaction) on diffferents test
    vectors (2**n) simulating with s-a-1(s-a-0) fault.
    ================================================================*/

int get_fault_sig ( int test_vector, int index, int s_a_1, int largest)
```

```
{  int i,j,k,l,mask,value,bit,f_flag;

  for (i=0; i<totalnumberofgate; i++)
    {
        if (list[i].outputname <= prim_in_number)
          {
            mask = pow(2,(list[i].outputname-1));
            bit = test_vector & mask;
            if (bit != 0)
                  {
                    bit = 1;
                  }
            test_array[i].test_output = bit;
            for (j = 0; j < list[i].total_of_input; j++)
                  {
                    test_array[i].test_input[j] = bit;
                  }
            update_test_array_special(i);
            update_next(i);
          }
        else{
          for (j=0; j<list[i].total_of_input; j++)
            {
                  if (list[i].inputname[j] <= prim_in_number)
                    {
                          mask = pow(2,(list[i].inputname[j]-1));
                          bit = test_vector & mask;
                          if (bit != 0)
                            {
                              test_array[i].test_input[j] = 1;
                            }
                          else
                            {

                              test_array[i].test_input[j] = 0;
                            }
                    }
            }
          }
    }
  for (i=0; i< totalnumberofgate; i++)
    {
     if (list[i].outputname == (index+1))
          {
            if ( list[i].gate_type != SPECIAL)
                  {
                    test_array[i].test_output = s_a_1;
                  }
            else{
```

```
                    test_array[i].test_output = s_a_1;
                    for (j=0; j<list[i].total_of_input; j++)
                      {
                            test_array[i].test_input[j] = s_a_1;
                      }
                    update_test_array_special(i);
                    update_next(i);
                   }
          }
     else{
             for (j=0; j<list[i].total_of_input; j++)
                    {
                     if (list[i].inputname[j] == (index+1))
                       {
                            test_array[i].test_input[j] = s_a_1;
                       }
                    }
           }
    }
  f_flag = TRUE;
  value = output_circuit(largest,f_flag);
  return(value);
}

/* ================================================================
  FUNCTION : Testing
  Performing an exhausive test on the circuits:
  1- On the circuit with the compaction tree.
  ================================================================*/

void testing ( void)
{
 int test_vector,index,s_a_1,done;

 num_of_test_vector = 2; /*pow(2,prim_in_number);*/
 dsc_fault_mis = 0;
 for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
   {
    init_test_value();
    signature[test_vector] = get_sig(test_vector,prim_out_number);
   }
 s_a_1 = FALSE;
 for(index = 0; index < num_of_line; index++)
    {
     done = FALSE;
     while (!done)
          {
           if (s_a_1 == FALSE)
                {
```

```
                    for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
                     {
                          init_test_value();
                          fault_sig[test_vector] =
get_fault_sig(test_vector,index,s_a_1,prim_out_number);
                     }
                    dsc_fault_mis = dsc_fault_mis + compare_test() ;
                    s_a_1 = TRUE;
                   }
              else
                   {
                    for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
                      {
                          init_test_value();
                          fault_sig[test_vector] =
get_fault_sig(test_vector,index,s_a_1,prim_out_number);
                      }
                    dsc_fault_mis = dsc_fault_mis + compare_test() ;
                     s_a_1 = FALSE;
                     done = TRUE;
                   }
          }
   }
}
/* ================================================================
  FUNCTION : first_test
  Performing an exhausive test on the circuits:
  1- On the original circuit (before compaction).
  ================================================================*/

void first_test ( void)
{  int i,j,test_vector,done,s_a_1,index,count,port_out;

  if (totainumberofgate != 0)
    {
        for (i=0; i<(num_of_line*2) ; i++)
          {
            test_a[i] = 0;
          }
        num_of_test_vector = 2; /*pow(2,prim_in_number);*/
        undetect_f = 0;
        for (i=0; i < prim_out_number; i++)
          {
            port_out = prim_out_array[i];
            for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
                  {
                    init_test_value();
                    signature[test_vector] = get_sig(test_vector,port_out);
                  }
```

```
            s_a_1 = FALSE;
            for(index = 0; index < num_of_line; index++)
                  {
                    done = FALSE;
                    while (!done)
                          {
                            if (s_a_1 == FALSE)
                              {
                                    for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
                                      {
                                        init_test_value();
                                        fault_sig[test_vector] =
get_fault_sig(test_vector,index,s_a_1,port_out);
                                      }
                                    if (compare_test() == 0)
                                      {
                                        test_a[index] = 1;
                                      }
                                    s_a_1 = TRUE;
                              }
                            else
                              {
                                    for(test_vector=0; test_vector < num_of_test_vector; test_vector++)
                                      {
                                        init_test_value();
                                        fault_sig[test_vector] =
get_fault_sig(test_vector,index,s_a_1,port_out);
                                      }
                                    if (compare_test() == 0)
                                      {
                                        test_a[index+num_of_line] = 1;
                                      }
                                    s_a_1 = FALSE;
                                    done = TRUE;
                              }
                          }
                  }
            }
          for (i =0; i < (num_of_line * 2) ; i++)
            {
              if (test_a[i] == 0)
                    {
                      undetect_f++;
                    }
            }
      }
    else
      {
          printf(" the number of gate in CUT must be > 1!!!");
```

```
    }
}

/* ==========================================================
  FUNCTION : print_signature
  write signature in to signature file.
  ==========================================================*/

void print_signature ( void)
{  int i,j;

 out = fopen("signature","w");
 if (out)
   {
        fprintf(out," \n");
        fprintf(out,"number of faults injected  %2d\n",(num_of_line*2));
        fprintf(out,"faults missed before compaction  %2d\n",undetect_f);
        fprintf(out,"faults missed after compaction  %2d\n",dsc_fault_mis);
        fprintf(out,"number of test vector %2d\n",num_of_test_vector);
        fprintf(out,"number of line before compaction %2d\n",num_of_line);
        fprintf(out,"number of line after compaction %2d\n",num_of_line);
        fprintf(out,"number of output line before compaction %3d\n",prim_out_number);
   }
 else
   {
        printf ("error opening for write \n");
   }
 fclose(out);
}

/* ==========================================================
                    MAIN PROGRAM
  ==========================================================*/
 void main ( void)
 {

  init();
  CUT_fr_file();/*read CUT from file CUT.file*/
  first_test(); /*test CUT before compaction */
  compact_fr_file(); /*read CUT from file compact.file */
  testing();
  print_signature();

  free (list);
  free (prim_out_array);
  free (signature);
  free (fault_sig);
  free (test_a)
  }
```