AN EFFICIENT TEST-PATTERN RELAXATION TECHNIQUE FOR SYNCHRONOUS SEQUENTIAL CIRCUITS

by

KHALED ABDUL-AZIZ AL-UTAIBI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements
for the Degree

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY
OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

NOVEMBER 2002

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by KHALED ABDUL-AZIZ AL-UTAIBI under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.

Thesis Committee

____
Dr. Aiman El – Maleh (Chairman)

____
Dr. Alaaeldin Amin (Member)

____
Dr. Atef Al – Najjar (Member)

____
Dr. Sadiq M. Sait
(Department Chairman)

____
Dr. Osama A. Jannadi
(Dean of Graduate Studies)

____
Date

Dedicated

to

the memory of my father

and

to my mother.

Acknowledgments

Acknowledgement due to King Fahd University of Petroleum and Minerals for supporting this research.

I would like to express my appreciation to my thesis committee chairman Dr. Aiman El-Maleh for his guidance, patience, and sincere advice throughout this thesis. I acknowledge him for his valuable time, constructive criticism, and stimulating discussions. Thanks are also due to my thesis committee members, Dr. Alaaeldin Amin and Dr. Atef Al-Najjar for their comments and critical review of the thesis.

I am very thankful to the department chairman Dr. Sadiq Sait, for his cooperation and support.

I am also thankful to my fellow graduate students and all my friends especially Mr. Ali Al-Suwaiyan.

Contents

Acknowledgments ii

List of Tables v

List of Figures vii

List of Algorithms viii

Abstract (English) ix

Abstract (Arabic) x

1 Introduction 1
1.1 Testing Systems-on-a-Chip (SOC) . . . 1
1.2 Motivation . . . 3
1.3 Problem Definition . . . 5
1.4 Thesis Organization . . . 6

2 Literature Review 8
2.1 Compression/Decompression Techniques . . . 9
2.1.1 Line-Feed-Shift-Register-Reseeding (LFSR-Reseeding) . . . 10
2.1.2 Run-length Coding . . . 16
2.1.3 Statistical Coding . . . 26
2.2 Compaction Techniques . . . 29
2.2.1 Compaction by Overlapping Self-Initializing Test Sequences . . . 29
2.2.2 Compaction Based on Insertion, Omission, and Selection . . . 33
2.2.3 Compaction Based on Vector Restoration . . . 37
2.2.4 Compaction Based on Inert Subsequence Removal . . . 40
2.3 Test-Pattern Relaxation Techniques for Combinational Circuits . . . 45

3 Proposed Test-Relaxation Technique 52
3.1 Illustrative Examples . . . 53

3.1.1 Single Value Justification 54
3.1.2 Limitations of Single Value Justification 57
3.1.3 Two Values Justification 59
3.2 Formal Description 62
3.2.1 Single-Value Justification 64
3.2.2 Two-Values Justification 70
3.3 Selection Criteria 79
3.4 Worst Case Analysis 86
3.4.1 Space Complexity 86
3.4.2 Time Complexity 88

4 Experimental Results 90
4.1 Comparison with Bitwise-Relaxation 91
4.2 Experiments on Cost Functions 94
4.3 Examining Different Aspects of the Two-Values Justification Technique 95

Conclusion 100

BIBLIOGRAPHY 103

List of Tables

2.1 Transition table for run-length coding. 19
2.2 Variable to fixed length coding. 20
2.3 Golomb codes ($m = 4$). 23
2.4 FDR code. 26
2.5 Extended FDR code. 27
2.6 The test sequence of Example 2.1. 34
2.7 The fault detection of the test sequence in Example 2.1. 34
2.8 The test sequence of Example 2.1 after insertion. 34
2.9 The fault detection of Example 2.1 after insertion. 35
2.10 Fault detection of Example 2.2. 38
2.11 Test subsequences of Example 2.2. 38
2.12 Fault detection of Example 2.3. 39

4.1 Benchmark circuits. 91
4.2 Test relaxation comparison between the two-values justification (TVJ) technique and the bitwise-relaxation method. 93
4.3 Cost function effect on the extracted percentage of X's. 94
4.4 Percentage of X's obtained by SVJ using different weights. 97
4.5 Test relaxation comparison between TVJ and SVJ. 98
4.6 Percentage of X's obtained by GVCF using different weights. 98
4.7 Test relaxation comparison between TVJ and GVCF. 98
4.8 Percentage of X's obtained by UACF using different weights. 99

List of Figures

1.1 An example of test-sequence relaxation in sequential circuits. 5

2.1 General structure of the single-polynomial LFSR. 12
2.2 Decompression using a single-polynomial LFSR. 13
2.3 An example of seed-computation in a single-polynomial LFSR. 14
2.4 General structure of the multiple-polynomial LFSR. 15
2.5 An example of a Burrows-Wheeler transformation. 17
2.6 Preparation of the matrix to be encoded. 18
2.7 An example of the modified run-length coding. 20
2.8 Structure of the cyclical scan chain. 22
2.9 An example of a Golomb coding. 23
2.10 Structure of the decoder used in the Golomb coding technique. 25
2.11 Construction of a Huffman tree. 28
2.12 Merging of two test sequences. 31
2.13 Examples of merging two test sequences. 32
2.14 Subsequence removal (Criterion 1). 41
2.15 Subsequence removal (Criterion 2). 42
2.16 Subsequence removal (Criterion 3). 43
2.17 Circuit of Example 2.4. 45
2.18 Circuit of Example 2.5. 47
2.19 Circuit of Example 2.6. 49
2.20 Limitation of the extended implication/justification. 49
2.21 Circuit of Example 2.7. 51

3.1 Iterative array model for asynchronous sequential circuits. 54
3.2 An example of the single-value relaxation. 56
3.3 Limitations of the single-value relaxation. 59
3.4 An example of the two-value relaxation. 60
3.5 Circuit of Example 3.13. 81
3.6 Circuit of Example 3.14. 82
3.7 Illustration of the effect of reconverging fanouts on the regular cost. . 84
3.8 Circuit of Example 3.16. 85

4.1 Effect of PO's selection on test vector relaxation. 93

List of Algorithms

3.1 Main Algorithm 65
3.2 MarkReachableLines(f,t) 67
3.3 JustifyFault(f, t) 69
3.4 Main Algorithm 71
3.5 ComputeFaultyValues(f,t) 72
3.6 JustifyFault(f,po,t) 73
3.7 JustifyGate(g,f,t) 74
3.7 (Cont.) JustifyGate(g,f,t) 75
3.8 GetCorrespondingValue(g,h,v_1,v_2) 78

THESIS ABSTRACT

Name: KHALED ABDUL-AZIZ AL-UTAIBI

Title: AN EFFICIENT TEST-PATTERN RELAXATION TECHNIQUE FOR SYNCHRONOUS SEQUENTIAL CIRCUITS

Major Field: COMPUTER ENGINEERING

Date of Degree: NOVEMBER 2002

Testing systems-on-a-chip (SOC) involves applying huge amounts of test data, which is stored in the tester memory and then transferred to the circuit under test (CUT) during test application. Therefore, practical techniques, such as compression and compaction, are required to reduce the amount of test data in order to reduce both the total testing time and the memory requirements for the tester. Some of the existing compression/compaction techniques require the test data to be partially specified, while others can benefit from partially specified test sets either directly or by specifying the don't care values in these test sets in a way that improves their efficiency. One obvious way to extract the don't care values in the test sets is to test for the possibility of changing every bit in the test set to an X based on fault simulation. This is called bitwise relaxation. In this thesis, we propose a novel and efficient test relaxation technique for synchronous sequential circuits. The proposed technique is faster than the bitwise relaxation method by several orders of magnitude.

MASTER OF SCIENCE DEGREE

King Fhad University of Petroleum and Minerals, Dhahran.

NOVEMBER 2002

ملخص الرسالة

الاســـــــــــم: خالد بن عبد العزيز العتيبي

عنوان الرســـــالة: استخلاص القيم غير المحددة في بيانات الاختبار للدوائر المتسلسلة

التخصـــــــــص: هندسة الحاسب الآلي

تاريخ التخـــــرج: رمضان ١٤٢٣ هـ

إن اختبار الأنظمة المدمجة على رقاقة واحدة يتطلب تطبيق أعداد ضخمة من بيانات الاختبار التي يتم تخزينها في ذاكرة أجهزة الاختبار قبل تطبيقها على الدوائر المراد فحصها، ولذلك فان التعامل مع هذه الكميات الضخمة من بيانات الاختبار يتطلب استخدام تقنيات فعالة، كضغط البيانات وتقليصها، وذلك لتقليص وقت الاختبار وكمية الذاكرة المطلوبة لفحص الدوائر، والملاحظ أن بعض تقنيات ضغط/تقليص البيانات تتطلب أن تكون بيانات الاختبار محددة جزئيا، بينما يمكن للبعض الآخر من هذه التقنيات الاستفادة من بيانات الاختبار المحددة جزئيا من خلال تحديد هذه البيانات بما يزيد من فاعلية عملية الضغط أو التقليص.

تقتضي الطرق التقليدية لاستخلاص بيانات الاختبار المحددة جزئيا؛ تحويل قيمة كل وحدة من بيانات الاختبار إلى قيمة غير محددة، ومن ثم اختبار الدائرة لمعرفة تأثير الوحدة المعدلة على عدد الأخطاء المكتشفة، وبناء على نتيجة الاختبار تحتفظ الوحدات التي لم تؤثر على عدد الأخطاء المكتشفة بالقيم غير المحددة، بينما تستعيد الوحدات التي أثرت على عدد الأخطاء المكتشفة قيمها الأصلية.

في هذه الأطروحة نقدم تقنية جديدة لاستخلاص بيانات الاختبار المحددة جزئيا للدوائر المتسلسلة. التقنية المقترحة تتفوق على الطرق التقليدية في عامل الوقت الذي تتطلبه عملية استخلاص البيانات المحددة جزئيا.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران، المملكة العربية السعودية

رمضان ١٤٢٣ هـ

Chapter 1

Introduction

1.1 Testing Systems-on-a-Chip (SOC)

Rapid advancement in VLSI technology has lead to a new paradigm in designing integrated circuits where a *system-on-a-chip* (SOC) is constructed based on pre-designed and pre-verified cores such as CPUs, digital signal processors, and RAMs.

Testing SOC is a major challenge due to limited access to the input and output lines of each core during test application. The test vectors for each core must be applied to the core inputs and the test responses must be observed at the core's outputs. A straightforward solution is to have full access to the input and output lines of all cores by multiplexing these lines to the chip pins. However, the complexity of this solution could be enormous. A more efficient way for providing test access to the cores is to use scan chains. The number and organization of these scan chains

determine the test data bandwidth (i.e., the rate of scanning in and scanning out test vectors and test responses respectively).

Another challenging problem in testing SOC is to deal with a large amount of test data that must be loaded from the tester memory, transferred to the SOC, and applied to the individual cores. The amount of time required to test a chip depends on the size of the test data that needs to be transferred and the speed of this transfer operation (i.e., test data bandwidth). The cost of automatic test equipments (ATE's) increases significantly with the increase in their speed, channel capacity, and memory size. Thus reducing the amount of test data and test time is a major concern in testing SOC [1].

One solution to this problem is to use *built-in self-test* (BIST) where on-chip hardware is added to enable the chip to test itself. The scheme has many advantages, especially the ability of self-testing at normal clocking rates (i.e., test at-speed), the ability for testing systems on-line, and reducing or eliminating the need for the expensive external ATE's. However, BIST has several drawbacks such as the complexity of designing test tools and degradation of the system performance that may occur due to the added hardware. In addition to that, BIST tools, which depend only on pseudo random generators, can't achieve high fault coverage because some faults are hard-to-detect using random test vectors.

Another alternative is to reduce the amount of test data using compression and compaction techniques. The objective of test set compression is to reduce the num-

ber of bits needed to represent the test set. Several test compression techniques have been proposed [2, 3, 4, 5, 6, 7, 8, 9]. In test compaction, the number of test vectors is reduced into a smaller number that achieves the same fault coverage. Test compaction techniques can be classified into two categories: *dynamic compaction* and *static compaction*. Dynamic compaction schemes such as [10, 11, 12, 13] try to reduce the number of test vectors during test vector generation. Static compaction schemes, on the other hand, perform compaction on test sequences after they are generated. Several static test compaction techniques have been proposed for synchronous sequential circuits. The techniques proposed in [14] use overlapping of self-initializing test sequences. Four compaction techniques based on insertion, omission, selection and restoration have been proposed by Pomeranz and Reddy [15, 16]. The technique in [17] compacts test sequences by removing inert subsequences under certain conditions.

1.2 Motivation

Compression techniques can achieve better results if the test set is composed of test cubes (i.e., if the test set is partially specified). In fact, some compression techniques such as, Line-Feed-Shift-Register-Reseeding (LFSR-Reseeding) [2, 3], require the test vectors to be partially specified. Even those techniques which require fully specified test data can benefit from the unspecified bits in the test set. For example,

variable-to-fixed-length coding [4] and variable-to-variable-length coding [5, 6] are known to perform better for long runs of 0's. Hence, assigning 0's to the don't care values in the test set will improves the efficiency of these techniques. Similarly, run-length coding techniques [7] can specify the don't care values in a way that will reduce test vector activity (i.e., the number of transitions from 0 to 1 and vice versa), which in tern improves the compression efficiency. On the other hand, the amount of compression that can be achieved with statistical coding techniques depends on degree of variation in the occurrences of unique test patterns (i.e., code words). If all test patterns occur with equal frequency, then no compression is achieved at all [18]. Thus, using partially specified test vectors adds more flexibility to statistical coding techniques in a sense that test patterns containing don't care values can be encoded with various possibilities.

Compaction techniques can also benefit from partially specified test sets. For example, when merging two test sequences using the overlapping compaction techniques described in [14], a don't care value, 'X', can be merged with any one of the values: '0', '1', and 'X'. Therefore, increasing the number of X's in a test set will reduce the number of conflicts that may occur when merging two test sequences, and hence, improves the efficiency of compaction.

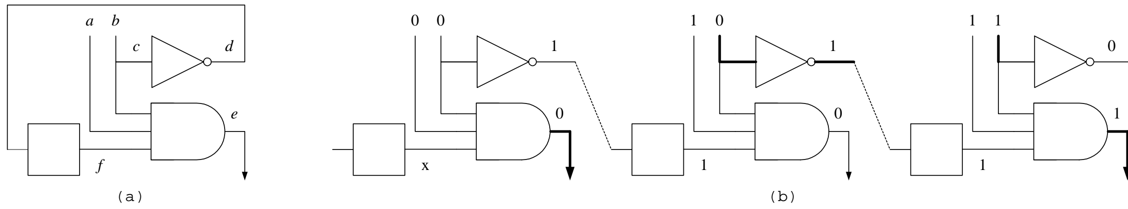


Figure 1.1: An example of test-sequence relaxation in sequential circuits.

1.3 Problem Definition

The problem of test set relaxation, i.e., extracting a partially specified test set from a fully-specified one, has not been solved effectively in the literature. This problem, which is targeted in this thesis, can be defined as follows. *Given a synchronous sequential circuit and a fully specified test set, generate a partially specified test set that maintains the same fault coverage as the fully specified one while maximizing the number of unspecified bits.* As an example, consider the circuit shown in Figure 1.1 (a) and the test set ab={00, 10, 11}. Under this test set 5 faults are detected: $e/1$, $e/0$, $d/0$, $b/1$ and $b/0$. These faults are shown as bold lines in Figure 1.1 (b). Notice that it is enough to set either $a = 0$ or $b = 0$ in the first test vector in order to detect the fault $e/1$. Hence, either one of these two bits can be set to 'X' and the fault is still detected. Also, the assignment $a = 1$ in the second test vector does not affect the detection of any one of the 5 faults. Therefore, this bit can be set to 'X' as well. As a result, the five faults can be detected under the relaxed test set ab={X0, X0, 11}.

One obvious way to solve the problem of test set relaxation is to use the bitwise-

relaxation method, where we test for every bit of the test set whether changing it to 'X' reduces the fault coverage or not. Obviously, this technique is $O(nm)$ fault simulation runs, where n is the width of one test vector, m is the number of test vectors. Obviously, this technique is impractical for large circuits.

In this thesis, we propose an efficient test relaxation technique for synchronous sequential circuits that maximizes the number of unspecified bits while maintaining the same fault coverage as the original test set. Our technique uses fault simulation to collect information about faults detected in every time frame (i.e., test vector) and faults propagating from one time frame to another. This information is used during a back-tracing phase starting from the last time frame all the way to the first time frame. The purpose of this phase is to mark all lines whose values are necessary to detect all the faults detected during the fault simulation phase. Obviously, any primary input that is not marked during the back-tracing phase is not required for fault detection, and hence can be relaxed. As compared to the bitwise-relaxation method, out technique is faster by several orders of magnitude.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, literature survey is presented. This chapter reviews several compression and compaction techniques proposed for synchronous sequential circuits. In addition to that, this chapter dis-

cusses some of the techniques proposed in the literature for solving the problem of test pattern relaxation for combinational circuits.

Chapter 3 covers implementation details of the proposed test-pattern relaxation algorithm for synchronous sequential circuits. This chapter starts with illustrative examples that explain the general behavior of the proposed algorithm. Then, it discusses the implementation details on different phases of the relaxation algorithm.

In Chapter 4 experimental results for the proposed technique are compared to the bitwise-relaxation method. The thesis ends with conclusion and future work.

Chapter 2

Literature Review

One of the challenges in testing SOC is to deal with a large amount of test data that must be loaded from the tester memory and transferred to the CUT during test application time. Reducing this amount of test data will significantly reduce the total test time, which in turn reduces the time-to-market. Test compression and compaction are known to be practical solutions for this problem. The efficiency of the test compression and compaction schemes depends on the test data itself. For most of the schemes that will be shown in this chapter, it is more efficient to work with a partially specified test set rather than a fully-specified one. Therefore, an efficient relaxation scheme is required to improve the efficiency of test compression and compaction techniques.

In this chapter, we review several test compression and compaction techniques proposed for synchronous sequential circuits, as well as some of the existing test-

pattern relaxation techniques for combinational circuits.

2.1 Compression/Decompression Techniques

The objective of test data compression is to compress (encode) a given test set T_D to a much smaller test set T_E that is stored in the tester memory. During test application, T_E is loaded from the tester memory and decompressed (decoded) using some decompression mechanism to obtain the original test set T_D before applying it to the required core. In order to guarantee the correctness of all applied test vectors and achieve a reduction in the overall testing time, a test compression/decompression scheme should meet two characteristics: lossless and simple decompression. The first characteristic must be met to guarantee the correctness of all applied test vectors. The second characteristic is important to guarantee a reduction in the overall testing time. Moreover, the decompression circuitry, residing at the core side, must be small so that it doesn't add significant area overhead. In addition to that, test compression/decompression schemes, designed for sequential circuits, must preserve the order of test vectors in every individual test set.

Compression/decompression schemes for deterministic test vectors can be classified, based on the type of test data they require, into three categories.

1. Schemes that require test data to be in the form of test cubes. Examples include LFSR reseeding [2, 3].

2. Schemes that require fully specified test vectors such as variable-to-fixed-length codes [4], variable-to-variable-length codes, and Extended frequency-directed run-length codes (EFDR) [5, 6, 19].

3. Schemes that have no specific requirements about the type of the test data. They compress test data regardless of their type. Run-length coding [7] and Huffman coding [8] are among the examples of this category.

Next, we will review some of the compression/decompression schemes targeting deterministic test vectors generated by ATPGs (schemes based on pseudo-random test vectors are out of the scope of this work).

2.1.1 Line-Feed-Shift-Register-Reseeding (LFSR-Reseeding)

Pseudo-random techniques (i.e., LFSRs) used in BIST designs for test generation don't need storage memories, and can be implemented with a small amount of hardware. However, they require long test sets to achieve high fault coverage because some faults are hard-to-detect using random vectors. Thus, LFSRs may generate several millions of test patterns before detecting these faults [20].

Several techniques have been proposed to address this problem. One approach involves the insertion of test-points in the CUT to enhance its random testability [14]. The disadvantage of this technique is that it requires modification of the CUT with possible performance degradation. Mixed-mode test pattern generation

is an alternative solution to this problem. The key idea in this approach is to use some mechanism to generate deterministic vectors that are known to cover hard-to-detect faults, while covering the remaining faults using pseudo-random test vectors. The additional cost of this approach depends on the logic required to include the deterministic test vectors, and more importantly, the storage capacity of these test vectors.

Below we review two approaches to reduce the storage requirements for the mixed-mode designs by encoding test cubes of the hard-to-detect faults using fixed-length seeds.

Single-Polynomial-LFSR Reseeding (SP-LFSR)

Koenemann [2] has proposed a compression/decompression technique for mixed-mode designs based on an intelligent reseeding of a single-polynomial LFSR. The technique uses short-length seeds to encode the test cubes. The resulting seeds are, then, stored in a ROM or a finite-state-machine (FSM). During test application time, the same LFSR is used to generate pseudo-random test vectors as well as deterministic test vectors.

The general structure of this technique is shown in Figure 2.1, where a deterministic test vector is generated by, first, resetting the LFSR. Then, the corresponding seed is loaded into the LFSR (reseeding). Finally, enough clocks are applied to shift the seed into the scan register, which will hold the desired pattern.

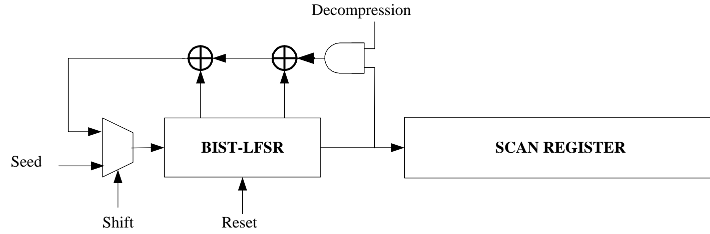


Figure 2.1: General structure of the single-polynomial LFSR.

Figure 2.2 illustrates an example of generating a test pattern “X0X11X” from the seed “110”. The decompression hardware, for this example, consists of a 4-bit LFSR formed using an existing 3-bit LFSR and one flip-flop from the scan register.

The seeds are computed based on the specified bits of the corresponding test cubes. As an example consider the LFSR represented by the polynomial $h(x) = x^3 + 1$. If the LFSR is to generate a test cube “X0X10X”, then a corresponding seed can be determined, as shown in Figure 2.3 (a), by solving the following system of equations:

$$a_0 + a_2 = 0$$

$$a_2 = 1$$

$$a_1 = 0$$

The resulting seed is $(a_2, a_1, a_0) = (101)$, which will subsequently produce the test pattern “001101” as shown in Figure 2.3 (b).

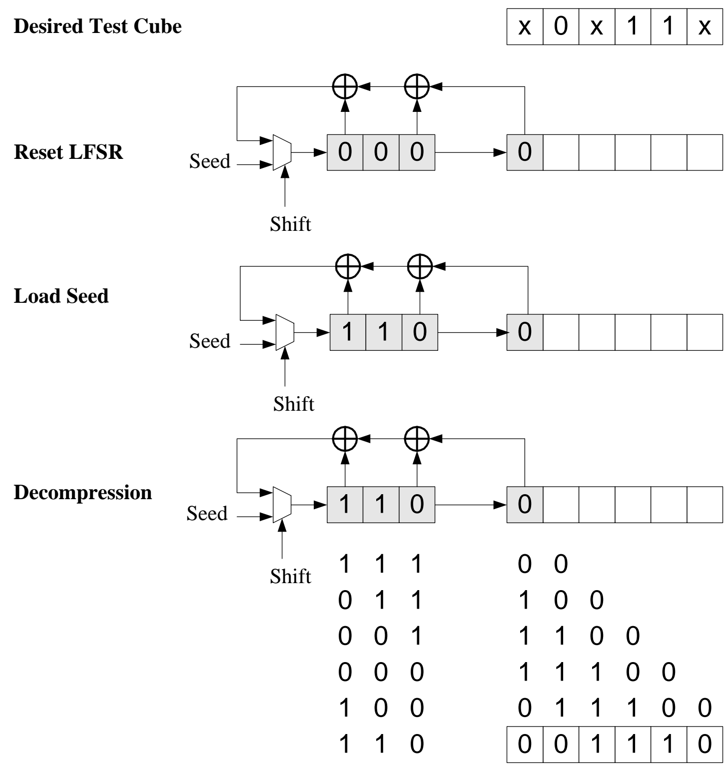
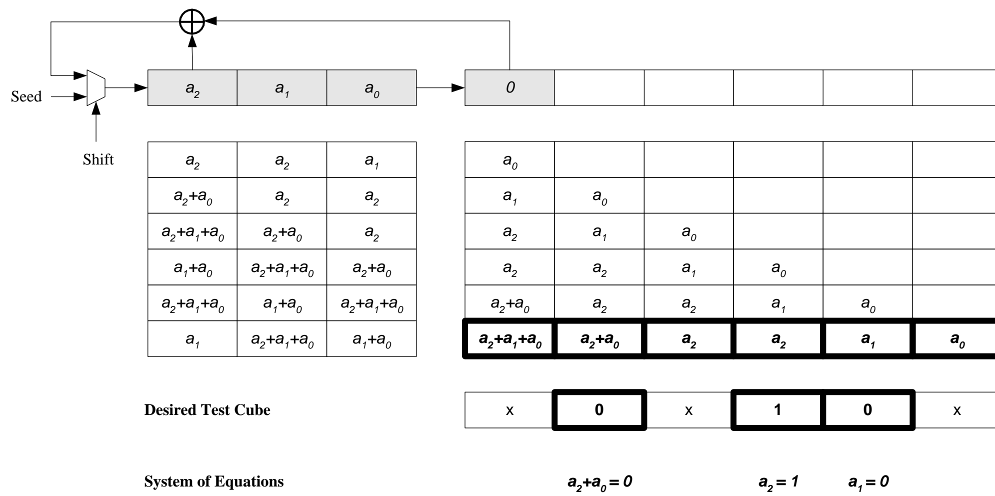
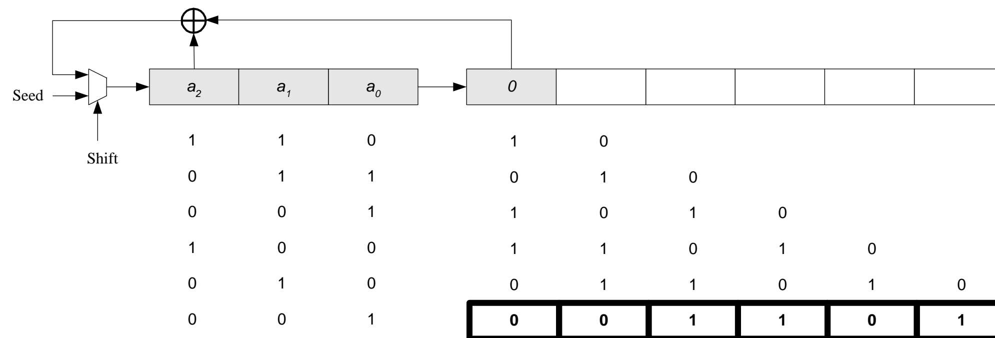


Figure 2.2: Decompression using a single-polynomial LFSR.



(a)



(b)

Figure 2.3: An example of seed-computation in a single-polynomial LFSR.

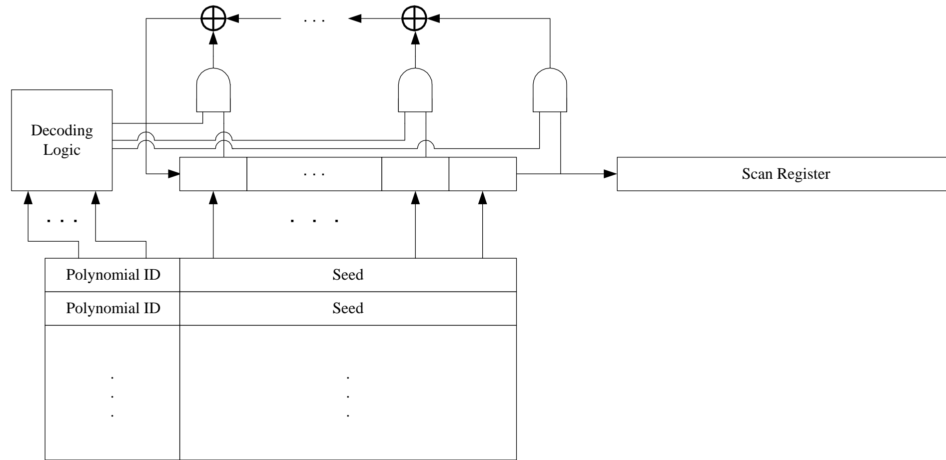


Figure 2.4: General structure of the multiple-polynomial LFSR.

Multiple-Polynomial-LFSR Reseeding

The reseeding approach described in [3] uses multiple primitive polynomials instead of a single polynomial to encode the test cubes as shown in Figure 2.4. The LFSR can operate according to one out of many primitive polynomials. Each test cube is encoded as a polynomial identifier and an initial seed. This process involves solving systems of equations similar to the one shown in Figure 2.2 (a). For each test cube, the process tries to find a suitable seed by checking all the available polynomials one after the other, and stops when finding the first encoding.

It is clear that the SP-LFSR approach requires less computational effort than the MP-LFSR. However, it requires an LFSR with $(s + 20)$ bits in order to reduce the probability of not finding a seed for a test cube with s specified bits to less than 10^{-6}, while a MP-LFSR with 16 polynomials can achieve the same probability using only $(s + 4)$ bits.

2.1.2 Run-length Coding

Many compression techniques are based on a well-known compression technique referred to as *run-length* coding. The basic idea of this technique is to encode a sequence of equal symbols (*run*) with a certain *codeword* depending on the length of that run. In this section, we will discuss three compression techniques which are based on run-length coding.

Run-length Coding with Burrows-Wheeler Transformation (BWT)

The original run-length coding technique compresses data by representing each run into two elements, the repeating-symbol and the run-length. For example, the string "*aaabbbbd*" is encoded as: (a, 3), (b, 4), and (d, 1). This technique is simple in compression and decompression. However, its efficiency depends on a feature of the encoded string called "*activity*". The activity of a string is the number of transitions in the string from one symbol to another. For example, the string "*aaabbbbd*" has an activity of two. Thus, run-length encoding is more efficient for strings with low activity.

In [7], Burrows-Wheeler transformation (BWT) was used to improve the efficiency of run-length coding. BWT transforms a string S into another string S' by rearranging its symbols in a way that may reduce the string activity. The first step in the BWT is to form an $n \times n$ matrix, where the nth row is obtained by rotating the original string $(n-1)$ times to the left. Then, the rows of the matrix are sorted

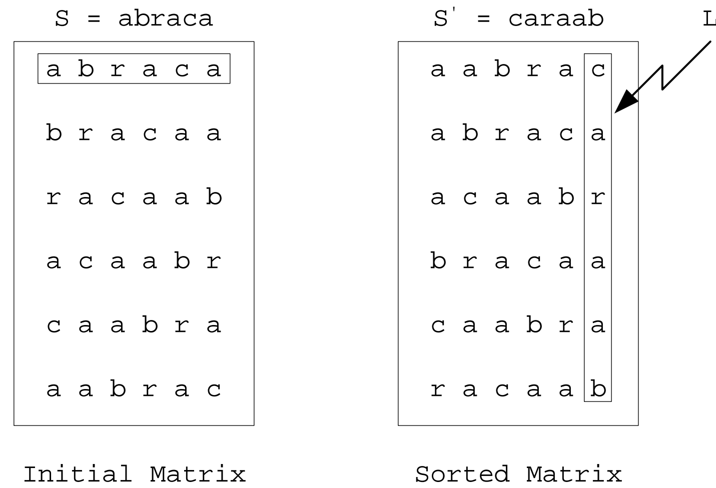


Figure 2.5: An example of a Burrows-Wheeler transformation.

lexicographically. Finally, the symbols in the last column L are grouped together to form the transformed string. Figure 2.5 shows an example of the BWT.

The BWT, as shown above, is simple and usually results in a better compression using run-length coding. Also, retrieving the original string from the resulting transformation is a simple operation that does not require sorting.

The procedure proposed by Yamaguchi, Tilgner and Ishida [7] works as follows. Given a test set D represented as a matrix $P \times Q$, where P is the number of test patterns and Q is the number of PI's, partition D into several equal size submatrices D_i each of size $M \times Q$. This is to reduce the processing time. The BWT is, then, applied on individual columns of every test set D_i to get an intermediate test set D_i'. Next, the activity, $\alpha'(k)$ of every column k in intermediate test set D_i', is compared with some threshold activity α_t and with the activity, $\alpha(k)$, of the

D_i: original test set

	1	2	3	4	5	6	7
	1	x	1	x	0	1	1
	x	1	x	1	x	0	0
	0	x	x	1	x	x	1
	x	1	1	1	x	1	0
	1	x	1	1	x	0	1
	0	0	0	x	x	x	0
a(*k*)	5	5	3	2	1	5	5

D'_i: intermediate test set

1	2	3	4	5	6	7
1	x	1	x	x	1	1
x	x	1	1	x	1	1
x	x	x	1	x	x	1
0	1	0	1	x	x	0
1	1	x	x	x	0	0
0	0	1	1	0	0	0
4	2	4	3	1	2	1

E_i: transformed test set

1	2	3	4	5	6	7
1	**x**	1	x	0	**1**	**1**
x	**x**	x	1	x	**1**	**1**
0	**x**	x	1	x	**x**	**1**
x	**1**	1	1	x	**x**	**0**
1	**1**	1	1	x	**0**	**0**
0	**0**	0	x	x	**0**	**0**
5	**2**	3	2	1	**2**	**1**

Figure 2.6: Preparation of the matrix to be encoded.

corresponding column in the original test set. If the activity of that column is less than both α_t and $\alpha(k)$, the column is copied in the final test set E_i. Otherwise, the corresponding column from original test set is copied into E_i. This will save the reverse transformation time for those columns whose BWT will not improve the run-length compression much. Finally, run-length encoding is applied to E_i.

Figure 2.6 shows an example for constructing an E_i for some test set D_i. The last row of each table in the figure indicates the activity of the columns. After applying BWT on D_i, the activity of the columns: 1, 2, 6, and 7, has decreased. However the activity of column 1, which is 4, is still above the threshold value ($\alpha = 3$). Hence, column 1 is copied from the original test set. Note that copying column 3 from D_i is trivial since its activity has increased after the BWT.

The transformed test set can be encoded using a straightforward run-length coding that encodes a run using two tuples, (s, L), where s is the repeating symbol and L is the length of the run. However, a more efficient run-length coding scheme was used in [7]. The main idea of this scheme is to combine the run lengths of two

Table 2.1: Transition table for run-length coding.

Transition($s \rightarrow t$)		**Symbol t**		
		0	**1**	**x**
Symbol s	**0**	–	L	$L+M$
	1	$L+M$	–	L
	x	L	$L+M$	–

consecutive runs into a single integer using some predefined transition table. As an example, consider the string "0000XX". This string has two runs: s="0000" and t="XX". One possible transition table for combining the lengths of two consecutive runs with three logic values ('0', '1', 'X') is shown in Table 2.1, where M is the length of the whole string (which is 6 in this example) and L is the length of the first run (4 in this case). Thus, the lengths of the two runs can be combined into a single integer ($L + M = 4 + 6 = 10$).

In the above scheme, a string is encoded by giving the first symbol, the activity of the string, and the integers corresponding to the combined run lengths using the transition table. Figure 2.7 (a) shows a complete example for encoding a test set E_i.

The decompression procedure is done as follows. Start with the symbol given in the encoded data. The integer i, corresponding to the current run, is decoded by computing ($i\%M$) to obtain the length of the current run. The symbol of the next run is the jth character ($j = \lceil i/M \rceil$) following the current symbol in the circular fashion shown in Figure 2.7 (b).

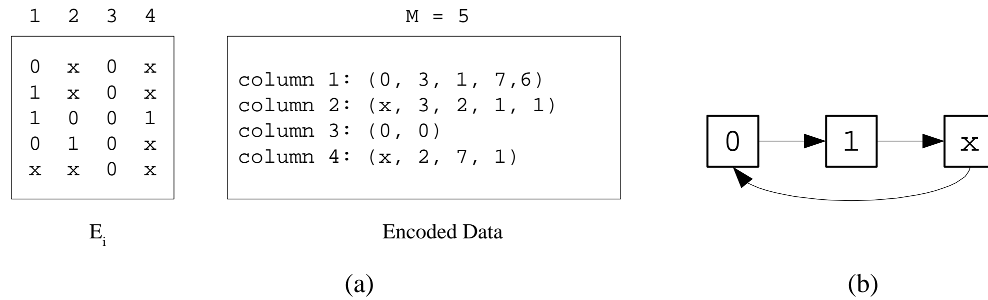


Figure 2.7: An example of the modified run-length coding.

Table 2.2: Variable to fixed length coding.

Data Block	Codeword
1	000
01	001
001	010
0001	011
00001	100
000001	101
0000001	110
0000000	111

Variable-to-Fixed-Length Codes

In this scheme, fully specified test data is compressed using *variable-to-fixed-length* codes, where a *fixed-length* codeword is used to encode a block of data based on the number of 0's in that block. Table 2.2 shows a variable-to-fixed-length coding with 3-bits codewords. Using this encoding scheme, the vector "0000010000001100001" can be encoded as follows. The block "000001" is encoded as 101, "0000001" is encoded as 110, "1" is encoded as 000, and "00001" is encoded as 100. Hence, the resulting vector is "101110000100".

The decompression for this scheme can be implemented simply using a counter

that counts down to 0 and outputs a "0" each time it decrements, and outputs a "1" (except for 7 0's) at the end of the count.

This scheme was used in [4] to compress the difference vector, T_{diff}, which is given by $(t_1, t_1 \oplus t_2, t_2 \oplus t_3, , t_{n-1} \oplus t_n)$, instead of compressing the test sequence T_D itself. This is motivated by the fact that successive test vectors in a test sequence often differ in a small number of bits. Hence, compressing the difference vector is more efficient since it contains more 0's than the original test vectors. In [4], the compressed difference vectors are used as an input to a 3-bit run-length decoder followed by a cyclical scan chain with a feedback to an XOR gate as shown in Figure 2.8 (a). The run-length decoder decompresses the encoded difference vectors, while the cyclical scan chain retrieves the original test vectors. These vectors are passed to a second cyclical scan chain that applies them to the CUT as illustrated in Figure 2.8 (b).

Golomb Codes

A drawback of the compression scheme described in [4] is that it uses *variable-to-fixed-length* codes, which are less efficient than the more general *variable-to-variable-length* codes. In [5] Golomb codes are used to map *variable-length* runs of 0's in a difference vector to *variable-length* codewords. This scheme works for both full-scan and non-scan (sequential) circuits. For full-scan circuits, the test vectors in a given test set can be reordered to maximize the number of 0's. On the other hand, the

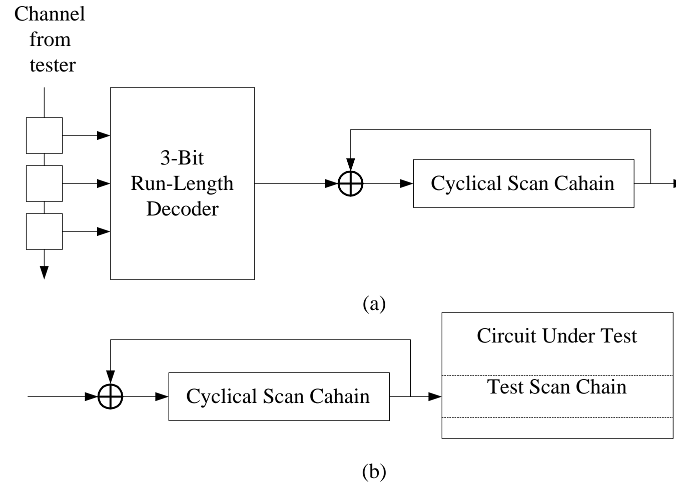


Figure 2.8: Structure of the cyclical scan chain.

order of test vector is preserved for non-scan circuits.

This scheme divides the codewords into g groups each of size m. Each one of these groups, as shown in Table 2.3, has a unique prefix, and m members each having $\log_2 m$ bit sequence (tail) that distinguishes it from other members of the group. The prefix of the kth group consists of $(k-1)$ 1's followed by a 0.

The first step in encoding a test set T_D is to generate its difference-vector test set T_{diff}. The next step is to determine the number of groups, g, and the group size m. The number of groups is determined by the length of the longest run, and the size m is dependent on the distribution of the test data. However, an optimal group size can be determined through actual experiments. Once g and m are determined, the runs of 0's in T_{diff} are mapped to the g groups, according to their lengths, as follows. The set of run-lengths $\{0, 1, 2, \ldots, m-1\}$ belongs to group 1; the set $\{m,$

Table 2.3: Golomb codes ($m = 4$).

Group	**Run-Length**	**Group Prefix**	**Tail**	**Codeword**
1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
2	4	10	00	10000
	5		01	10001
	6		10	10010
	7		11	10011
3	8	110	00	110000
	9		01	110001
	10		10	110010
	11		11	110011

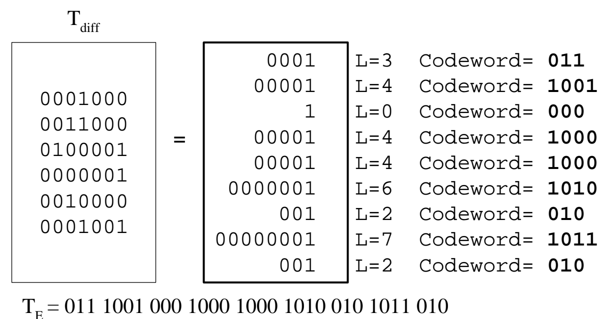


Figure 2.9: An example of a Golomb coding.

$m+1, \ldots, 2m-1\}$ belongs to group 2, and so on. The final codeword for a run of length L that belongs to group k consists of the prefix of group k followed by the tail corresponding to the jth ($j = L - (k-1)m + 1$) member of group k. Figure 2.9 illustrates the encoding of a difference vector test set consisting of 6 vectors using the Golomb codes of Table 2.3. Notice that only the first two groups of this table are required since the length of longest run is 7.

The decompression operation can be implemented using a $\log_2 m$ bit counter that

counts up to m and a finite-state-machine as shown in Figure 2.10. The *en* signal synchronizes the input of the decoder. When *en* is high the next bit of the current codeword is shifted into the FSM through *bit_in*. The *inc* signal is used to increment the counter, and the *rs* signal indicates that the counter has finished counting. The v signal synchronizes the decoder output. When v is high one decoded bit is shifted out through *bit_out*.

The decoder starts by decoding the prefix of the codeword. For every 1 in the prefix, the counter counts up to m (the *en* signal is low while the counter is busy counting). During this operation, the decoder outputs m 0's. When the last 1 in the prefix gets decoded (i.e., a 0 is shifted in), the FSM starts decoding the tail sequence of the codeword. The number of 0's generated during this operation is equivalent to value of the tail sequence. Finally, the decoder outputs a 1 at the end of the decoding operation.

Frequency-Directed Run-Length Code

Frequency-directed run-length (FDR) code [5] is another variable-to-variable coding technique based on encoding runs of 0's. In FDR code, the prefix and tail of any codeword are of equal size. In any group A_i, the prefix is of size i bits. The prefix of a group is the binary representation of the first member of that group. When moving from group A_i to group A_{i+1}, the length of the code words increases by two bits, one for the prefix and the other for the tail. Runs of length i are mapped to

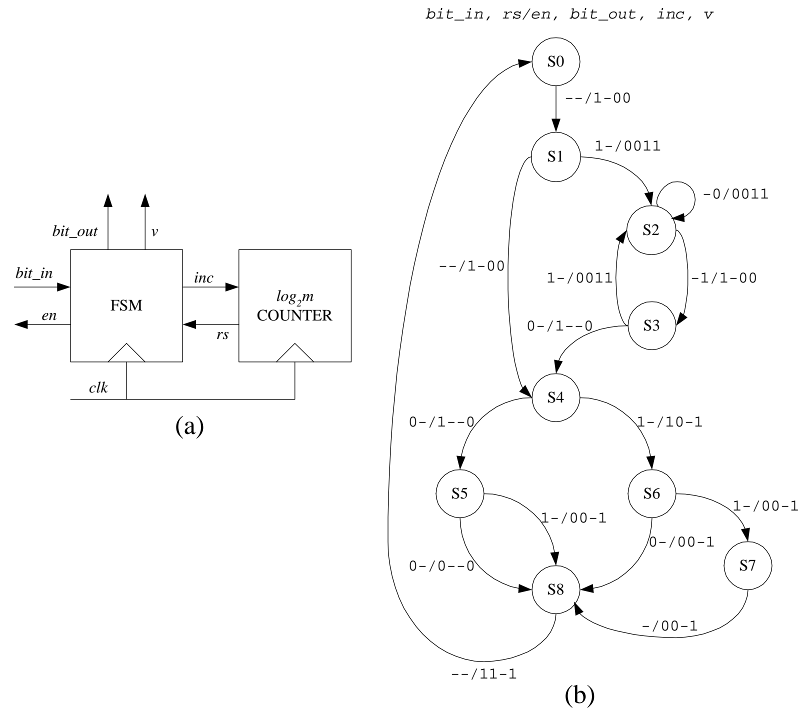


Figure 2.10: Structure of the decoder used in the Golomb coding technique.

group A_j, where $j = \lceil \log_2(i+3) \rceil - 1$. The size of the ith group is equal to 2^i. The FDR code for the first three groups is shown in Table 2.4.

Since the FDR technique considers only runs of 0's, it is not efficient for test sets that contain large numbers of runs of 1's. As an example, consider the test $T = \{0110001111111000000001\}$, which contains 10 runs of 0's. Encoding this test using FDR code results in the encoded test $T_{FDR} = \{01\ 00\ 1001\ 00\ 00\ 00\ 00\ 00\ 00\ 110010\}$. This is larger than the original test.

The FDR technique was extended in [19] to encode both runs of 0's and 1's by adding an extra bit to the begging of a codeword to indicate the type of run. If the bit is 0, this indicates that the codeword is encoding a run of 0's, otherwise it

Table 2.4: FDR code.

Group	Run-Length	Group Prefix	Tail	Codeword
A_1	0	0	0	00
	1		1	01
A_2	2	10	00	1000
	3		01	1001
	4		10	1010
	5		11	1011
A_3	6	110	000	110000
	7		001	110001
	8		010	110010
	9		011	110011
	10		100	110100
	11		101	110101
	12		110	110110
	13		111	110111

encodes a run of 1's. This code, called Extended FDR (FDR), is shown in Table 2.5. As with the FDR code, in this code when moving from group A_i to group A_{i+1}, the length of codewords increases by two bits, one for the prefix and the other for the tail. Runs of length i are mapped to group A_j, where $j = \lceil \log_2(i+2) \rceil - 1$.

If we apply the EFDR on the test $T = \{0110001111111000000001\}$, we find that it contains 5 runs of 0's and 5 runs of 1's. This can be encode as $T_{EFDR} = \{000\ 100\ 001\ 11011\ 0110000\}$, which contains 21 bits. Obviously, for this example EFDR is performing better than the FDR code.

2.1.3 Statistical Coding

Statistical coding methods minimize the average length of a codeword by assigning short codewords to frequently occurring patterns, while assigning longer codewords

Table 2.5: Extended FDR code.

Group	**Run Length**	**Group Prefix**	**Tail**	**Codeword Runs of 0's**	**Codeword Runs of 1's**
A_1	1	0	0	000	100
	2		1	001	101
A_2	3	10	00	01000	11000
	4		01	01001	11001
	5		10	01010	11010
	6		11	01011	11011
A_3	7	110	000	0110000	1110000
	8		001	0110001	1110001
	9		010	0110010	1110010
	10		011	0110011	1110011
	11		100	0110100	1110100
	12		101	0110101	1110101
	13		110	0110110	1110110
	14		111	0110111	1110111

to less frequently occurring patterns. Examples of statistical encoding methods include Huffman coding, Shannon-Fano coding, and Lempel-Ziv coding [21, 22, 23].

The scheme proposed in [8] uses Huffman coding to compress the test patterns in a test set T_D. It starts by computing the frequency (number of occurrences) of every unique test pattern in T_D. Then, it builds a Huffman tree out of these test patterns. The edges of the Huffman tree are labeled either with 0 or 1, while every leaf in the tree represents one unique pattern. The codeword of every unique pattern is obtained by traversing the path from the root to the corresponding leaf noting the sequence of 0's and 1's along the edges of this path.

Figure 2.11 illustrates the procedure for constructing the Huffman tree for a test set with five unique patterns x_1, x_2, x_3, x_4, and x_5 having frequencies of occurrence

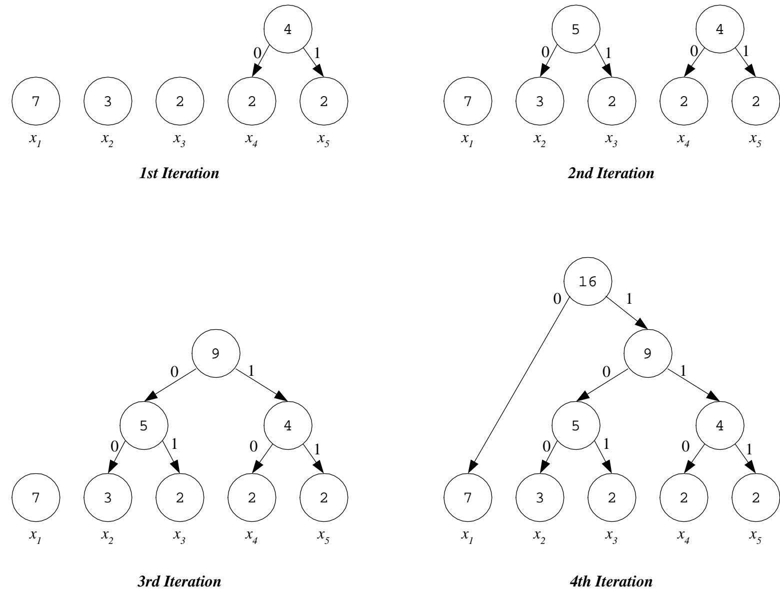


Figure 2.11: Construction of a Huffman tree.

$f(x_1) = 7$, $f(x_2) = 3$, $f(x_3) = 2$, $f(x_4) = 2$, and $f(x_5) = 2$. The procedure iteratively selects two nods v_1 and v_2 with the lowest frequencies, and generates a parent node v for v_1 and v_2. The edges (v, v_1) and (v, v_2) are labelled 0 and 1 respectively, and the node v is assigned the frequency $f(v) = f(v_1) + f(v_2)$. The codewords are obtained, as explained previously, by traversing the tree from the root to every leaf. Thus, the codewords for x_1, x_2, x_3, x_4, and x_5 are 0, 100, 101, 110, and 111 respectively.

The encoded test set T_E is stored in the tester's memory, and is read out one bit at a time during test application. T_E can be decoded using a simple finite-state-machine (FSM) with $n - 1$ states, where n is the number of leaves in the corresponding Huffman tree.

2.2 Compaction Techniques

Test set compaction is the process of reducing the length of a test set while achieving the desired fault coverage. Test compaction procedures can be classified into two categories: *dynamic* compaction and *static* compaction. Dynamic compaction procedures try to reduce the length of the test set during the generation of test vectors. Static compaction, on the other hand, performs compaction after the generation process. Hence, it does not require any modification to the test generation procedures. In addition to that, static compaction can be used after dynamic compaction, to further reduce the length of the test sequence [24].

In testing sequential circuits the order in which a test sequence is applied to the CUT is critical and can't be altered. This complicates the issue of compacting test data for sequential circuits than that for combinational circuits because any technique based on reordering of test vectors can't be applied here.

In this section we will review some of the static compaction procedures proposed for sequential circuits (dynamic compaction is out of the scope of our work).

2.2.1 Compaction by Overlapping Self-Initializing Test Sequences

In a combinational circuit, two test vectors are compatible if they do not specify opposite values for any primary input (PI). Two compatible test vectors t_i and t_j

can be combined into one test $t_{ij} = t_i \bigcap t_j$ using the intersection operation defined below.

$\bigcap$	0	1	x
0	0	ϕ	0
1	ϕ	1	1
x	0	1	x

Since the order of applying test vectors in combinational circuits is not important, test vectors can be compacted in any order. However, this is not the case with sequential circuits where the memory elements must be set to a specific state in order to excite, propagate or detect certain fault(s). A test sequence (with a strict order) is used to set the memory elements in the required state. Hence, two patterns can't be compacted arbitrary.

Self-initializing test sequence: In a test set containing self-initializing sequences, any test sequence does not depend on the state at which the sequential circuit arrives due to the application of previous sequences. Thus, the set of test sequences may be applied in any order without affecting the fault coverage. Such test sequences can be overlapped as long as they are compatible with each other, and the order of test vectors in each test sequence is preserved.

Compatibility of a test sequences: Consider two test sequences T_1 and T_2 of lengths l_1 and l_2 respectively, and $l_1 \geq l_2$. Originally the total test length due to T_1 and T_2 is $l_1 + l_2$ as shown in Figure 2.12 (a). There are four ways in which T_2 can be skewed from the start of T_1 as shown in Figure 2.12 (b)-(e). In the first

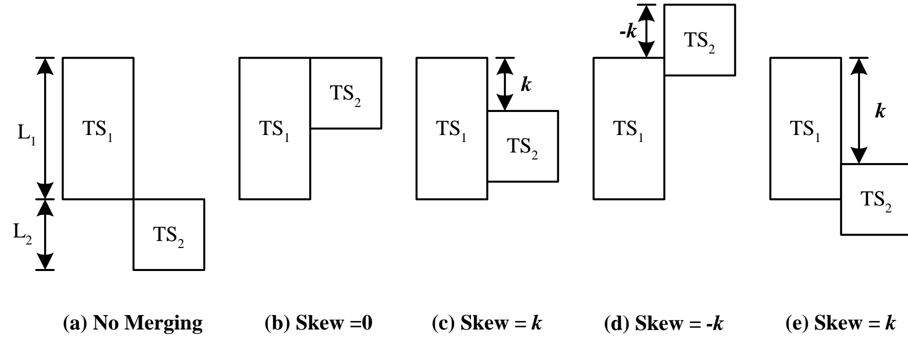


Figure 2.12: Merging of two test sequences.

group, the two test sequences are aligned with each other such that they start at the same time (skew = 0). These two sequences will be compatible if the ith vector in T_1 is compatible with the ith vector in T_2 for (i = 1, 2, ..., l_2). In the second group, the two sequences will be compatible if every vector in T_2 is compatible with its corresponding vector in T_1 (note that the 1st vector in T_2 corresponds to the kth vector T_1). The reduction in total test length in this case is l_2. The compatibility of the test sequences in the third and fourth groups can be handled in a similar manner, where the reductions in the total test length due to compaction are $(l_2 - k)$ and $(l_1 - k)$ respectively.

In [14], three algorithms were given to merge self-initializing test sequences. The first algorithm merges aligned test sequences as shown in Figure 2.13 (a). If aligning two sequences will result in a conflict between one or more vectors, a second algorithm is used to merge two sequences with a skew as shown in Figure 2.13 (b). The third algorithm improves the compatibility of test sequences using stretching. A sequence is stretched if some of its vectors are repeated several times without

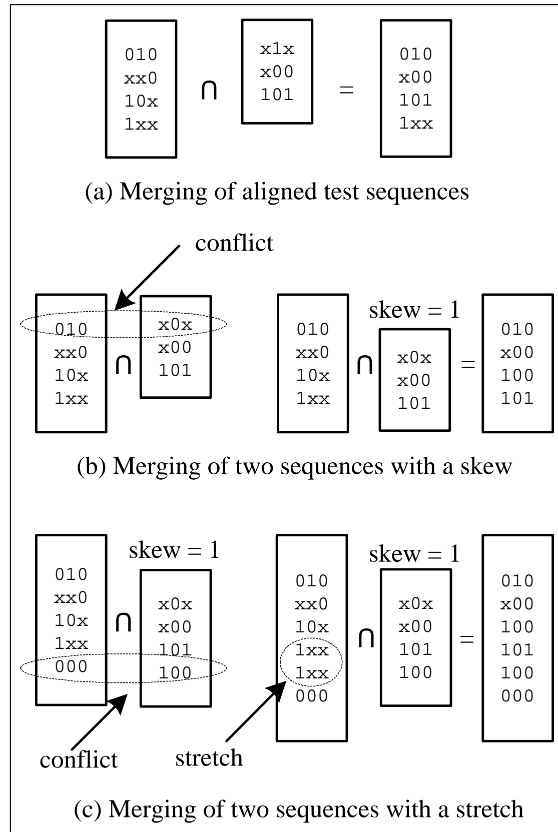


Figure 2.13: Examples of merging two test sequences.

changing their order. For example the sequence (101x, 1x01, 111x) can be replaced by (101x, 1x01, 1x01, 111x). This will add one more degree of freedom to the process of compaction as shown in Figure 2.13 (c). Merging two test sequences using the last two algorithms may affect the fault coverage. Therefore, a fault simulation step is performed after the merging process.

2.2.2 Compaction Based on Insertion, Omission, and Selection

Three compaction algorithms were proposed in [15]. These algorithms are insertion, omission and selection. The following definitions and notations will be used to describe these algorithms.

- $T[i]$ is the ith test vector in the test sequence T.
- $T[u_i, u_k]$ is a subsequence of T between the two time units u_i and u_k.
- S_i is the state of the fault-free circuit at time u_i.
- S_i^f is the state of the faulty circuit at time unit u_i for some fault f.
- The effective test length of T, L_{eff} , is the minimum length of a subsequence of T that starts at time 0 and include the detection time of every detected fault.

Compaction Based on Insertion

Consider a fault f detected at time unit u_d by the test vector $T[d]$. If there exist two time units u_j and u_k such that $u_j < u_k \leq u_d$ and $S_j/S_j^f = S_k/S_k^f$, then f can be detected earlier by duplicating and inserting the subsequence $T[u_k, u_d]$ at u_j. The insertion operation increases the total length of the test sequence, however, it reduces the effective length L_{eff} by reducing the highest detection time. The insertion

Table 2.6: The test sequence of Example 2.1.

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T[i]	0011	1101	0001	0011	1110	1011	0001	0010	0000	0110

Table 2.7: The fault detection of the test sequence in Example 2.1.

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Detected Faults	ϕ	f_1, f_2	ϕ	f_3	f_4, f_5, f_6	f_7	ϕ	f_8, f_9, f_{10}	ϕ	f_{11}

operation is applied iteratively, until no additional improvements in effective test length or in the fault coverage can be obtained. The following example demonstrates the insertion operation.

Example 2.1: Consider the test sequence shown in Table 2.6. The corresponding fault detection of this test sequence is shown in Table 2.7. Assume that the combined fault-free/faulty states are identical at times u_7 and u_8. Then, f_{11} can be detected at earlier time by duplicating and inserting $T[8, 9]$=(0000, 0110) at u_7 as shown in Table 2.8. After inserting the subsequence, fault simulation is performed to see the effect of the insertion operation on the faults detected in time units following u_7. Table 2.9 shows a possible fault detection due to the insertion operation, where the detection time of f_{11} has been shifted from u_9 to u_8.

Table 2.8: The test sequence of Example 2.1 after insertion.

i	**0**	**1**	**2**	**3**	**4**	**5**
T[i]	0011	1101	0001	0011	1110	1011
i	**6**	**7**	**8**	**9**	**10**	**11**
T[i]	0001	0000	0110	0010	0000	0110

Table 2.9: The fault detection of Example 2.1 after insertion.

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Detected Faults	ϕ	f_1, f_2	ϕ	f_3	f_4, f_5, f_6	f_7	ϕ	f_8, f_9	f_{10}, f_{11}	ϕ

Compaction Based on Omission

This algorithm is based on the omission of redundant test vectors from the test sequence. The omission of a test vector t_i at time unit u_i may affect faults detected in the time units following u_i. Also, it may result in detecting new faults. Hence, a fault simulation is performed after every omission operation to check the fault coverage. Test vectors are considered for omission in the order in which they appear in the test sequence. After omitting a test vector t_i, a fault simulation is performed for all faults detected in time units following u_i and for all undetected faults. If the fault coverage after omission is not lower than the fault coverage before omission, the change is accepted, otherwise, t_i is restored.

Compaction Based on Selection

This technique works as follows. For every detected fault f, it first collects all subsequences that detect the fault f, such that each subsequence starts from unspecified states (i.e., all memory elements set to X's). Next, it uses a covering procedure to select a minimal subset of the collected subsequences that detect all the faults detected by the original test set. During this selection phase, a newly selected subsequence is combined with the previously selected subsequences in the order by which they

appear in the original test set. Then the new sequence is simulated to identify the faults that still need to be detected. This can expressed by means of the following example.

Example 2.2: Assume that we are given a circuit with 8 test vectors (t_1 to t_8) which detect 10 faults (f_1 to f_{10}). First, the circuit is fault simulated starting from every test vector with all states set to X's. Thus, the circuit is simulated starting from t_1 up to t_8, then from t_2 up to t_8, and so on. Suppose that the fault detections shown in Table 2.10 represent the result of the above fault simulations. From this table we can obtain the subsequences that detect a given fault. For example, the fault f_1 is detected in t_3 of the first fault simulation. Thus, it can be detected by any one of the subsequences (1,3), (1,4), (1,5), (1,6), (1,7) and (1,8). Table 2.11 shows all the subsequences that detect every one of the 10 faults.

Next, a minimal subset of these subsequences is selected to detect all the faults detected by the original test set as follows. Starting from f_1, we find that this fault can be detected by any one of the subsequences (1,3), (1,4), (1,5), (1,6), (1,7) and (1,8). However, the subsequence (1,3) is selected because it is the shortest among the 6 subsequences. Notice that this sequence detects the faults f_2, f_3, and f_5 as well. Therefore, no need to consider these faults in the next selection. The next fault to be consider is f_4 which is detected by 7 subsequences. The shortest sequence among these subsequences is (7,8). This subsequence is combined with the subsequence (1,3) to form the sequence $T' = \{t_1, t_2, t_3, t_7, t_8\}$. Notice the new

sequence may detect additional faults other than those detected by the individual subsequences.Therefore, we should fault simulate the circuit under the new test sequence before selecting a new subsequence.

Suppose that after simulating the circuit under the test sequence T' we found that the faults f_1, f_2, f_3, f_5, f_9 and f_{10} are detected (i.e., f_4 is missing). In this case, f_4 is considered again and the next choice is the subsequence (6,8). This subsequence is combined with T' to form the test sequence $T'' = \{t_1, t_2, t_3, t_6, t_7, t_8\}$. Then, the circuit is fault simulated under the new test sequence. Suppose that T'' detects all the 10 faults (i.e., the fault f_6 is accidently detected). In this case, the selection process stops and the compacted test set will be in T''.

2.2.3 Compaction Based on Vector Restoration

For many test sequences considered in [15], the test length after compaction is less than half of the original test length. This suggests that it may be faster to decide which test vectors must be restored in the test sequence in order to maintain the fault coverage, instead of deciding on the test vectors that may be omitted. The technique in [16] restores test vectors for each fault starting from the hardest-to-detect to the easiest. In this way, some faults can be detected by sequences of other faults. The technique is illustrated by the following example.

Example 2.3: Consider the fault detection shown in Table 2.12. The technique starts by omitting (almost) all test vectors in the test sequence (test vectors that

Table 2.10: Fault detection of Example 2.2.

Starting From	TV(s)	Detected Faults
t_1	t_1, t_2	ϕ
	t_3	f_1, f_2, f_3, f_5
	t_4, t_5, t_6	ϕ
	t_7	f_6
	t_8	f_4, f_7, f_8, f_9, f_{10}
t_2	t_2, t_3, ..., t_7	ϕ
	t_8	f_4, f_6, f_7, f_8, f_9, f_{10}
t_3	t_3, t_4, ..., t_7	ϕ
	t_8	f_4, f_6, f_7, f_8, f_9, f_{10}
t_4	t_4, t_5, ..., t_7	ϕ
	t_8	f_4, f_6, f_7, f_8, f_9, f_{10}
t_5	t_5, t_6, t_7	ϕ
	t_8	f_4, f_6, f_7, f_8, f_9, f_{10}
t_6	t_6, t_7	ϕ
	t_8	f_4, f_7, f_8, f_9, f_{10}
t_7	t_7	ϕ
	t_8	f_4, f_9, f_{10}
t_8	t_8	ϕ

Table 2.11: Test subsequences of Example 2.2.

Fault	Subsequences
f_1	(1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
f_2	(1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
f_3	(1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
f_4	(1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8)
f_5	(1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
f_6	(1,7) (1,8) (2,8) (3,8) (4,8) (5,8)
f_7	(1,8) (2,8) (3,8) (4,8) (5,8) (6,8)
f_8	(1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8)
f_9	(1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8)
f_{10}	(1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (7,8)

Table 2.12: Fault detection of Example 2.3.

Fault	**Subsequence to detect the fault**
f_1	(t_1, t_3)
f_2	(t_2, t_5)
f_3	(t_3, t_{12})
f_4	(t_{11}, t_{16}) or (t_{18}, t_{20})
f_5	(t_{17}, t_{20})

synchronize the circuit by taking it from an unspecified state to a fully specified one are not omitted). Starting from the last detected fault (i.e., f_5), the algorithm restores test vectors one at a time until the required fault is detected. For instance, t_{20} is simulated first and since it does not detect f_5, t_{19} is added to the sequence. Now, the sequence (t_{19}, t_{20}) is simulated, and so on, until the sequence (t_{17}, t_{20}) is restored. Notice that any fault that is detected by this sequence is dropped from the current fault list. For example, f_4 is removed from the list since it is detected by (t_{18}, t_{20}). After restoring f_5, the algorithm starts from the last yet-undetected fault and repeats the process until all detected faults are covered.

We can speed-up the process of vector restoration in the above technique by considering several faults in parallel during the restoration process. In [25], faults are grouped together according to their detection times by the original test sequence. Specifically, all the faults detected by the original test sequence at time unit u_i are considered together during the restoration process.

The restoration techniques we have seen so far restore the test vectors in the same order in which they appear in the original test sequence. For example, if the

compacted test sequence is $T_c = \{t_1,\ t_2,\ t_9\}$, then the compacted test sequence after restoring t_8 becomes $T_c = \{t_1,\ t_2,\ t_8,\ t_9\}$. Notice that in this example, the faults detected after restoring t_9 may not be detected after restoring t_8. Thus, all faults detected at a certain test vector need to be resimulated when a new vector is restored into the compacted test sequence. The technique proposed in [26] avoids this problem by reversing the order of the vectors during restoration. In this technique, every time a subsequence of the original test sequence is restored to detect a subset of faults, the subsequence is placed at the end of the compacted test sequence (T_c). In this way, a fault detected by T_c is guaranteed to remain detected by at the end of the compaction process.

2.2.4 Compaction Based on Inert Subsequence Removal

The technique proposed in [17] is based on the observation that test sequences traverse through a small set of states and some states are frequently revisited throughout the application of a test set. Subsequences that start and end on the same states may be removed if sufficient conditions are met. Before going over these conditions, the following two definitions are given to help in explaining the inert subsequence removal technique.

Definition 2.1 *A state-recurrence subsequence* T_{rec} *is a subsequence of test vectors such that the fault-free states reached at the beginning of* T_{rec} *and those reached at*

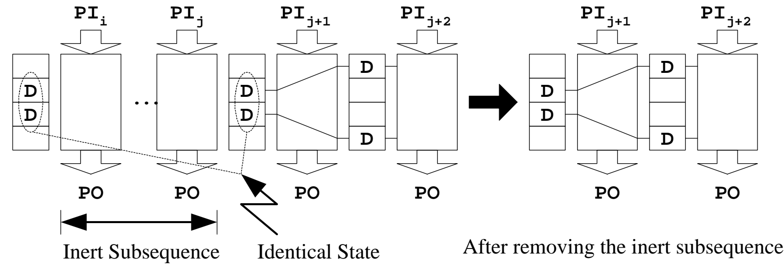


Figure 2.14: Subsequence removal (Criterion 1).

the end of it are identical.

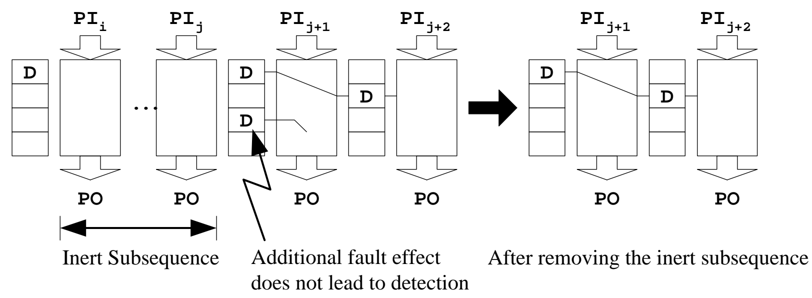
Definition 2.2 *An inert recurrence subsequence T_{inert} is a state-recurrence subsequence such that no additional faults are detected within this subsequence.*

Inert subsequences can be removed, in order to compact a test sequence, given that certain criteria are satisfied. These criteria are discussed below.

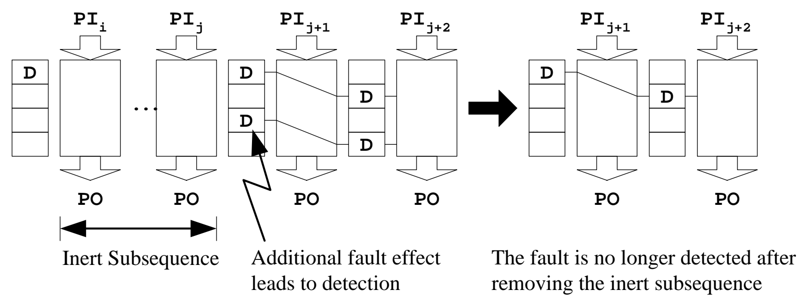
Criterion 1: Consider the situation in Figure 2.14. If the faulty state before and after the inert subsequence T_{inert} are identical for every undetected fault f that is activated by this subsequence, then T_{inert} can be removed without affecting the fault coverage.

Criterion 2: For an inert subsequence T_{inert}, if the faulty state after the subsequence covers the faulty state before the subsequence for every activated fault, and any additional fault effects propagated at the end of the subsequence do not lead to detection, then T_{inert} can be removed. This criterion is illustrated in Figure 2.15.

Criterion 3: For an inert subsequence T_{inert}, if the faulty state before the subsequence covers the faulty state after the subsequence and the additional fault effects



(a)



(b)

Figure 2.15: Subsequence removal (Criterion 2).

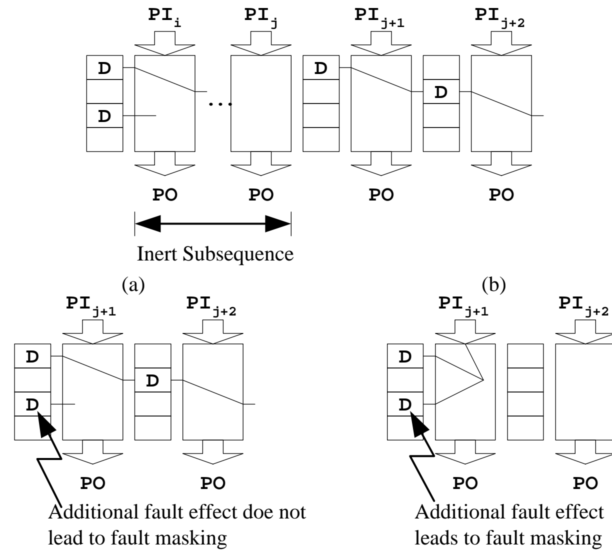


Figure 2.16: Subsequence removal (Criterion 3).

propagated before the subsequence do not cause fault-masking in time frames following the subsequence, then T_{inert} can be removed. This situation is shown in Figure 2.16.

Criterion 4: For an inert subsequence T_{inert}, if neither of the faulty states before and after the subsequence cover the other, then conditions imposed on activated faults in both Criterion 2 and Criterion 3 must be satisfied in order to remove T_{inert}.

The above criteria can identify subsequences that may be removed in order to compact a given test sequence. However, this technique will not be able to compact test sequences that do not contain state-recurrence subsequences. In such cases, state relaxation can be used to identify state-recurrence subsequences that may be removed. For example, assume that the test set T transfer the circuit through the states $S_1 = 10110$, $S_2 = 00100$, $S_3 = 01110$, $S_4 = 00110$ and $S_5 = 11001$ without

repeating any state. It is possible that not all specified bits in S_4 are necessary to reach S_5. Therefore, if S_4 can be relaxed to "X0110", then there exists a state-recurrence subsequence between S_1 and S_4. The technique proposed in [27] extends the subsequence removal technique using state relaxation to identify more cycles in a test set. In state relaxation, logic simulation is performed for all test vectors in the test set. For every test vector, the state of the circuit (i.e., values of the memory-elements) is analyzed to identify those bits that have no effect on the next state or primary outputs, and relax them. This process is illustrated in the following example.

Example 2.4: Consider the circuit shown in Figure 2.17. This circuit has two primary inputs (A, B), one primary output ($G5$), and three memory-elements ($G7$, $G8$, $G9$). If the initial state of the circuit is ($G7$, $G8$, $G9$)=(0, 0, 1) and the input is (A, B)=(1, 0), then the output circuit becomes 0 and the circuit is transferred to the state ($G7$, $G8$, $G9$)=(0, 1, 1). Notice that a fault effect on $G7$ can't be propagated to the primary output or the memory-elements because of the controlling value $A = 1$. Therefore, the value on $G7$ can be relaxed to 'X'. Similarly, $G8$ can be relaxed without affecting the primary output or the memory-elements because of the the controlling value $B = 0$. Thus the initial state can be relaxed to ($G7$, $G8$, $G9$)=(X, X, 1).

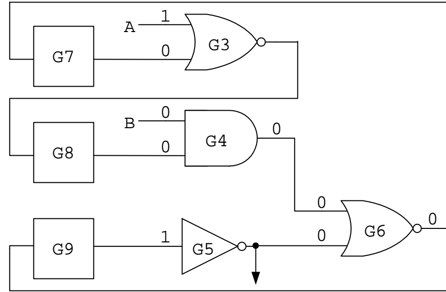


Figure 2.17: Circuit of Example 2.4.

2.3 Test-Pattern Relaxation Techniques for Combinational Circuits

In this section, we discuss two test-pattern relaxation techniques proposed for combinational circuits [28, 29]. The main idea of both techniques is to determine logic values in the fully-specified test set that are necessary to cover (i.e., detect) all faults which are detectable by this test set. Unnecessary logic values are set to X's. Before we go into details of the two techniques, we define some of the terminologies used in each technique. Given a test set T, if any fault detected by test vector t in T is detected by at least one test vector in $T - \{t\}$, t is called *redundant* test vector. If a fault, f, is detected by t in T, but not detected by any test vector in $T - \{t\}$, f is called an *essential* fault of t. To indicate that a line l is stuck-at value v, the notation l/v is used. The notation $l = v/\bar{v}$ indicates that the fault-free value of line l is v, and the faulty value of line l is $\bar{v}$.

Relaxation Using Implication/Justification Procedures

The technique proposed in [28] uses fault simulation and implication/justification procedures similar to those used by ATPG algorithms. The technique consists of three phases. The first phase starts by fault-simulating the circuit under the original test set to determine essential faults of every test vector. Then, implication/justification procedures are used to specify logic values necessary to detect these essential faults. The resulting relaxed test set is still an intermediate test set. In the second phase, the circuit is fault-simulated under the intermediate test set. Then, implication/justification are used again to specify logic values necessary to detect any fault that is detected by the original test set but not detected by the intermediate test set. After all, if the second phase fails to detect any of the detectable faults, a more restricted implication/justification (call *extended* implication/justification) is used in third phase to specify logic values necessary to detect all faults missed by the previous phases. The general behavior of this technique is illustrated by the following example.

Example 2.5: Consider the combinational circuit shown in Figure 2.18 under the test set $abcdef$ ={010100, 010001}. The first test vector, t_1, can detect three faults: $a/1$, $c/1$ and $j/1$. The second test vector, t_2, can detect the fault $e/1$ in addition to the faults $a/1$ and $j/1$. Therefore, $c/1$ is considered as an essential fault for t_1, while $e/1$ is an essential fault for t_2.

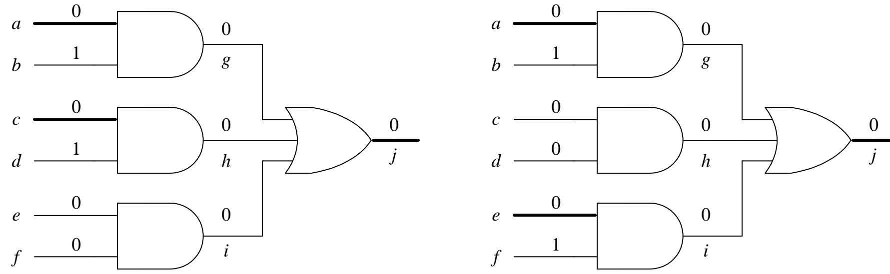


Figure 2.18: Circuit of Example 2.5.

In the first phase, the algorithm proposed in [28] will specify logic values necessary to detect the essential faults in each test vector using implication/justification as follows. In t_1, the assignments $c = 0$, $d = 1$, $g = 0$ and $i = 0$ are necessary to excite and propagate the fault $c/1$ to the primary output h. In another word, excitation and propagation of $c/1$ implies these four assignments. Next, the algorithm justifies the assignments $g = 0$ and $i = 0$. The assignment $g = 0$ must be satisfied by the assignment $a = 0$ (i.e., $g = 0$ implies $a = 0$), while $i = 0$ can be satisfied (i.e., justified) by either $e = 0$ or $f = 0$. Since the technique does not specify any mechanism to select a specific assignment when there are more than one choice, the first assignment (i.e., $e = 0$) is selected. Notice that unnecessary assignments, i.e., $b = 0$ and $f = 0$, are set to don't care. The second test vector, t_2, is handled in a similar manner. The resulting intermediate test set is $abcdef$ ={0X010X, 0X0X01}.

In the second phase, the algorithm fault simulates the circuit under the intermediate test set to determine undetected faults. Since $a/1$ is not detected under the intermediate test set, the algorithm will use implication/justification in order

to specify logic values that are necessary to detect the missing fault. In t_1, the assignments $a = 0$ and $b = 1$ are necessary to excite and propagate the fault to the line g. The assignments $h = 0$, and i are necessary to propagate the fault to the primary output j. These two assignments are already justified in the previous phase. Notice that any bit that is specified as either '0' or '1' in one phase will maintain its value in the next phase. (i.e., can't be set back to 'X'). Since no further faults need to be detected, the second test vector will not be processed by the second phase and the third phase is not needed. Thus, the relaxed test set is $abcdef$ ={01010X, 0X0X01}.

The implication/justification used in the first two phases are based on 3-valued logic ('0', '1', 'X'). Hence, it takes only fault-free values into account, and ignores faulty values. Due to this, the obtained test set may miss some detectable faults, even though the faults have been treated explicitly. This situation is shown in the following example.

Example 2.6: Consider the combinational circuit shown in Figure 2.19. The test vector $abc = 000$ detects the fault $b/1$. The assignments $b = 0$, $c = 0$ and $d = 0$ are required to detect this fault. The assignment $d = 0$ can be satisfied by $b = 0$ and $a = 0$ is not needed. However, as shown in Figure 2.19 (b), if we set $a = x$ the detection of $b/1$ is not guaranteed, i.e., it is not detected when $a = 1$.

In order to a void this problem, a more restricted (*extended*) implication/justification procedures are used in the third phase to detect any fault that is missed by the second

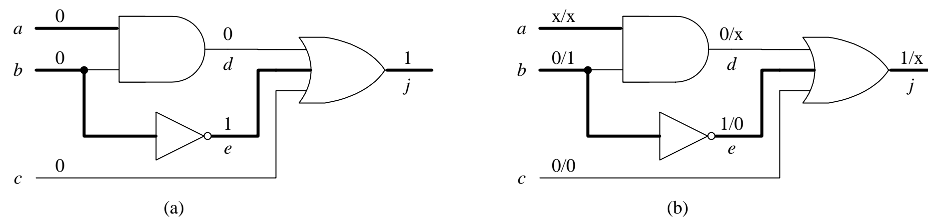


Figure 2.19: Circuit of Example 2.6.

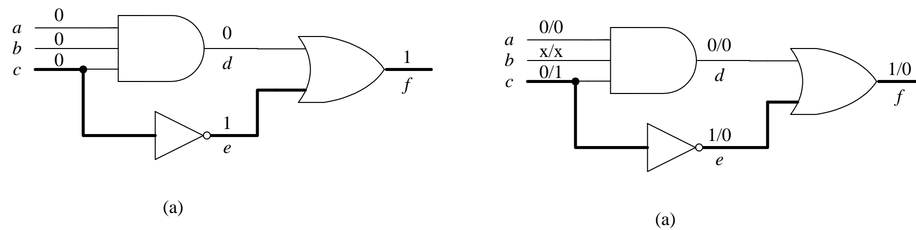


Figure 2.20: Limitation of the extended implication/justification.

phase. In the extended implication/justification, when a fault effect is propagated to at least one fan-in line of a gate, values of all fan-in lines of the gate are specified. In Figure 2.19, a is set to 0 since the fault $b/1$ is propagated to the second fan-in of the AND gate.

Relaxation by Marking Required/Nonrequired Lines

The main drawback of the previous technique is the way it handles fault masking using extended implication/justification. This technique may specify unnecessary values as shown in Figure 2.20. Since the fault-effect of $c/1$ reaches one of the fan-in lines of the AND gate, the extended implication/justification will specify all input lines of this gate (i.e., a, b, and c). However, as shown in Figure 2.20 (b), it is enough to specify one input line in addition to c in order detect the fault $c/1$.

A more clever way is used in [29] to avoid the problem of fault masking. Unlike the previous technique which requires 2 to 3 fault simulations per test vector, the algorithm proposed in [29] requires only 1 fault simulation per test vector. In brief words, the algorithm does the following for every test vector t of the test set. First, it fault simulates the circuit under the test vector t and generates a list of newly detected faults. Then, for every newly detected fault f, it marks all the lines whose values are required for f to be detected (i.e., excited and propagated to a primary output). Obviously, the unmarked input lines are not required for fault detection, and thus they are relaxed. These steps are illustrated in the following example.

Example 2.7: Consider the circuit shown in Figure 2.21 and the test vector $abcde = 00000$. Under this test vector, the faults $m/0$, $l/0$, $k/1$, $i/0$, $g/1$, and $b/1$ are detected. Assume that the newly detected fault is only $b/1$, i.e., other faults are either previously detected by an earlier test vector, or not part of the fault list. The assignment $b = 0$ is required to excite the fault, while the assignments $j = 0$ and $k = 0$ are required propagate it to the primary output. The assignment $j = 0$ can be satisfied by either one of the two assignments $c = 0$, or $de = 00$. If we choose to satisfy $j = 0$ by the assignment $c = 0$, then the assignment $de = 00$ is no longer necessary, and this implies that we can relax cde to "0XX". Similarly, if we choose to satisfy $j = 0$ by the assignment $de = 00$, then cde can be relaxed to "X00". This shows that a test vector can be relaxed in more than one way, and some ways might have more relaxed bits than others. In [29], some sort of cost functions are used to

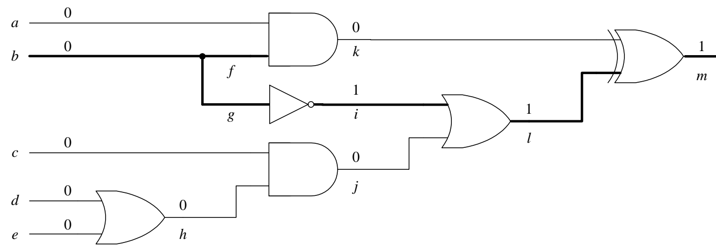


Figure 2.21: Circuit of Example 2.7.

guide the selection when there are more than one choice.

The other requirement for fault propagation, which is $k = 0$, can't be satisfied by the assignment $b = 0$ because this will result in an incorrect relaxation of the input a. To show this, assume that stem b is faulty, i.e., $b = 0/1$. In this case, if line a is relaxed, the fault on the stem will not propagate to the output. It will be masked by the 'X' value on line a, producing the value $1/x$ on the output m. The problem occurs because we justified the requirement on line k from line f, which is reachable from the fault on the stem b. Justifying a required value from a reachable line guarantees that the required value is satisfied in the fault-free machine but not in the faulty machine. This problem can be avoided by justifying the required value from an unreachable line. This guarantees that the value will be satisfied for both the fault-free and the faulty machines. For this example, the required value on line k has to be satisfied by marking line a as required, resulting in the test vector $abcde$ =100xx, or $abcde$ =10x00. Therefore, lines that are reachable from faulty stems should be identified before justifying the required values. In Figure 2.21, reachable lines from stem b are f, g, i, l, and m.

Chapter 3

Proposed Test-Relaxation Technique

In this chapter, we discuss our proposed test relaxation technique for synchronous sequential circuits that maximizes the number of unspecified bits in a given test set while maintaining the same fault coverage as the original test set. This technique uses fault simulation to collect information about faults detected in every time frame (i.e., test vector) and faults propagating from one time frame to another. This information is used during a back-tracing phase starting from the last time frame all the way to the first time frame. The purpose of this phase is to mark all the lines whose values are necessary to detect all the faults detected during the fault simulation phase. Obviously, any primary input that is not marked during the back-tracing phase is not required for fault detection, and hence can be relaxed.

During our work, we have implemented two versions of test-pattern relaxation: *single-value justification* and *two-values justification*. The first version uses a technique similar to the one used in [29]. In this technique, faults are justified based on the fault-free values of the circuit. It also uses some rules based on fault-reachabillity analysis to avoid fault masking. However, these rules, as will be shown in the next section, have some limitations. Therefore, we have extended the justification process in the second version to handle both the fault-free and faulty values of the circuit.

3.1 Illustrative Examples

In this section, we demonstrate the two versions of our proposed technique by a number of examples. The following conventions are assumed. A synchronous sequential circuit can be represented as a linear iterative array of combinational cells as shown in Figure 3.1. Each cell represents one time frame in which the current states of the flip-flops become pseudo-inputs (y_i), and the next states become pseudo-outputs (Y_i). A fault in this model can be represented by multiple identical faults (one in every cell). To indicate that a line l is stuck at value v, we use the notation l/v. The notation $l = v/\bar{v}$ is used to indicate that the fault-free (good) value of line l is v, and the faulty value of line l is $\bar{v}$. When we say that a line l is required, we mean that the value on l is required.

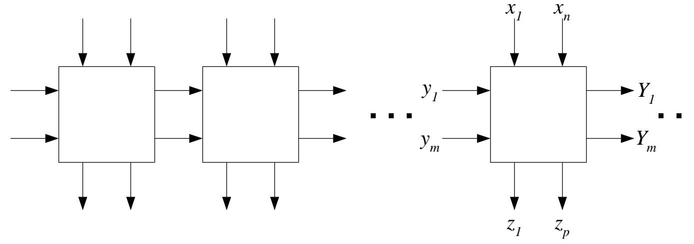


Figure 3.1: Iterative array model for asynchronous sequential circuits.

3.1.1 Single Value Justification

The general behavior of this technique can be explained as follows. At any time frame t, all logic values which are necessary to excite a newly detected fault and propagate it to some primary output p are marked as required. Next, these logic values are justified backwards starting from p towards primary inputs and/or memory-elements. At the end, unmarked primary inputs are not required and can be relaxed. On the other hand, required values on the memory-elements are justified when the next time frame, $t-1$, is processed.

The justification process in this technique is based on logic values only (single-value justification), which may result in masking some of the detected faults as will be shown int the next example. Therefore, the technique uses some rules based on fault-reachability analysis to avoid fault masking. These rules can be summarized as follows. Assume that a value on a line l is to be justified. If l is a primary input (PI), then it is required. If l is a single-input, XOR or XNOR gate, then all the values on l's inputs are required. Similarly, if l is an AND, OR, NAND or NOR gate with a non-controlling value, then all the values on its inputs are required. However,

if l has a controlling value, then we need to check if it has an unreachable input with a controlling value. In this case, it is sufficient to justify the value using that unreachable input. Otherwise, we check whether l is reachable or not. If it is not reachable, then we justify only the reachable lines. Otherwise, all the values on the inputs will be justified.

Example 3.8: Consider the iterative-array-model shown in Figure 3.2 (a). This model represents two time frames of a synchronous sequential circuit under two test vectors: $t_1 = 110$ and $t_2 = 000$. In the first time frame, the fault $G5/0$ is excited and propagated to the memory element (i.e., D-flip-flop), but no fault is detected yet. In the second time frame, three faults are excited: $G7/0$, $G7b/0$ and $G6/0$. These faults are propagated together with the fault $G5/0$ to the primary output $G6$ where they get detected. So, in order to relax t_1 and t_2 in a correct manner, we should take into account not to relax any test bit that is necessary to excite/propagate any one of the four faults: $G5/0$, $G7/0$, $G7b/0$ and $G6/0$. Starting from the second time frame, we find that the assignment $G7 = 1$ is required to excite the faults $G7/0$, $G7b/0$ and $G6/0$. The assignment $G4 = 0$ is required to propagate the faults $G5/0$, $G7/0$ and $G7b/0$ to the primary output. Notice that the fault $G6/0$ is excited at the primary output, and hence does not require any propagation. Next, we need to justify the two assignments $G7 = 1$ and $G4 = 0$. Since $G7$ is a memory element, its value can't be justified in the current time frame. The assignment $G4 = 0$, on the other hand, can be satisfied only by one assignment which is $G2 = 0$. Notice

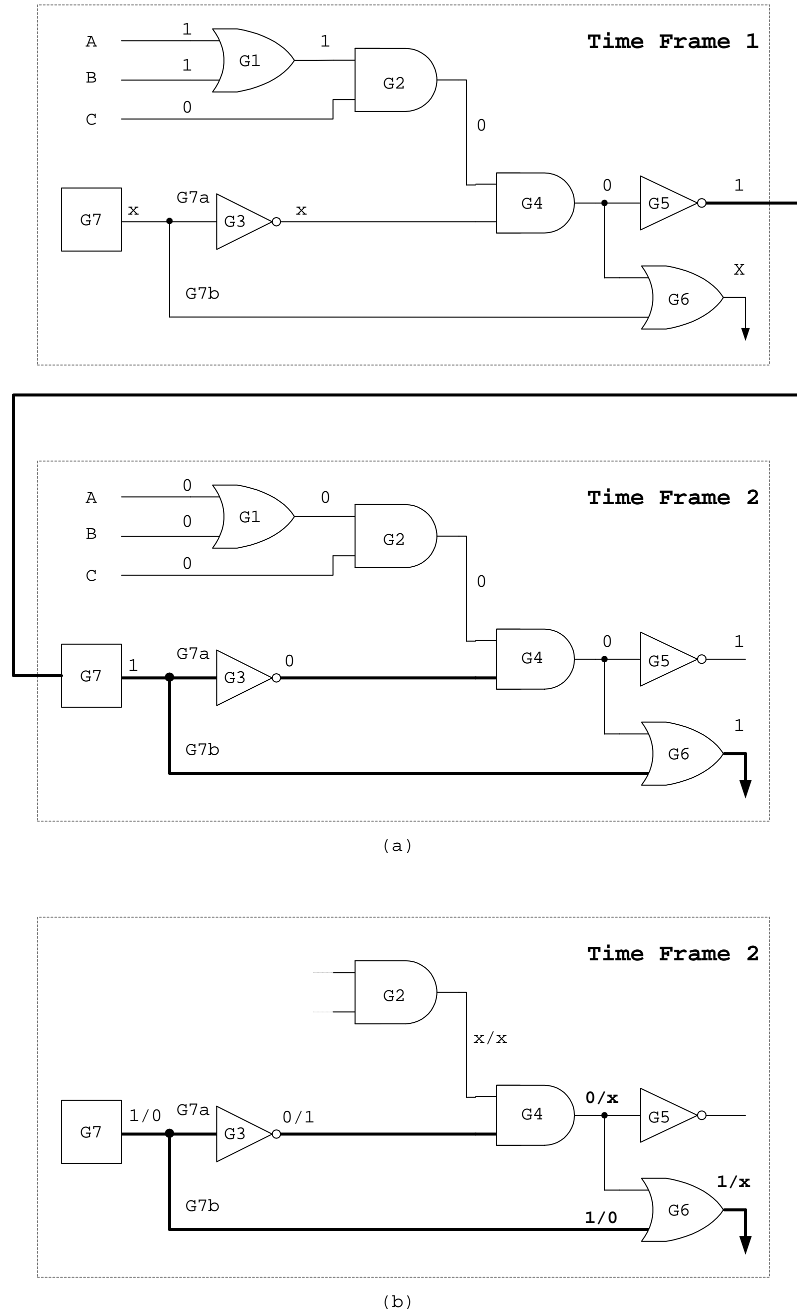


Figure 3.2: An example of the single-value relaxation.

that the assignment $G3 = 0$ is reachable from the faults $G5/0$ and $G7/0$. Therefore, $G3 = 0$ can't be chosen to justify $G4 = 0$ otherwise it will result in masking the faults $G5/0$ and $G7/0$ as shown in Figure 3.2 (b). In the presence of the fault $G7/0$ (or $G5/0$), the value of $G3$ is 0 in the fault-free machine and 1 in the faulty machine. Therefore, if we relax the value at $G2$ (i.e., $G2$ =X), then a value of 'X' will propagate through $G4$ to the first input of $G6$ causing the faulty value at the second input to be masked.

The assignment $G2 = 0$ can be satisfied by either one of the two assignments $AB = 00$ or $C = 0$. The first assignment will result in relaxing one input which is C, while the second assignment will result in relaxing the two inputs A and B.

In the first time frame, the assignment $G4 = 0$ is required to excite the fault $G5/0$ as well as to justify the assignment on the memory element (i.e., $G7 = 1$) that has not been justified in the second time frame. Next, $G4 = 0$ is satisfied by the assignment $G2 = 0$, which is satisfied in turn by the assignment $C = 0$. The two inputs A and B are not required in this time frame, and hence can be relaxed. The resulting relaxed test set is {XX0, XX0}.

3.1.2 Limitations of Single Value Justification

The technique shown in the previous section depends on justifying logical values of the good machine that are necessary to excite/propagate those faults detected during the fault-simulation phase. During the justification process, it avoids justifying a

required value from a reachable line to account for fault masking. However, there are situations, as will be shown in the next example, where it is possible to justify the required value from a reachable line without masking any of the detected faults.

Example 3.9: Consider the model shown in Figure 3.3 (a) which represents one time frame of a synchronous sequential circuit. Assume that the fault $G4/1$ is newly detected in this time frame. The two assignments $G4 = 0$ and $G2 = 0$ are necessary to excite the newly detected fault and propagate it to the primary output $G3$. According to the single-value justification, $G2 = 0$ can be satisfied only from A since $G1$ is reachable from the fault $G4/1$. However, if we choose $G1 = 0$ to justify the assignment $G2 = 0$ and relax A, as shown in Figure 3.3 (b), then the fault $G4/1$ is still detected. This is because the faulty value at $G4$ is considered as a controlling value for the gate $G3$, and hence can't be masked by an 'X' propagating through $G2$.

Another limitation of the single-value justification technique is that it doesn't take advantage of some situations in which the fault can be propagated by justifying only its faulty value. This limitation is discussed in the following example.

Example 3.10: Consider the model shown in Figure 3.3 (c). Assume that the only newly detected fault is $G4/0$. The assignment $G4 = 1$ is required to excite the newly detected fault. It seems that the assignment $G2 = 0$ is required to propagate the fault $G4/0$ to the primary output, and hence $A = 0$ is required to satisfy this assignment. However, if we relax A, as shown in Figure 3.3 (d), the fault $G4/0$ still

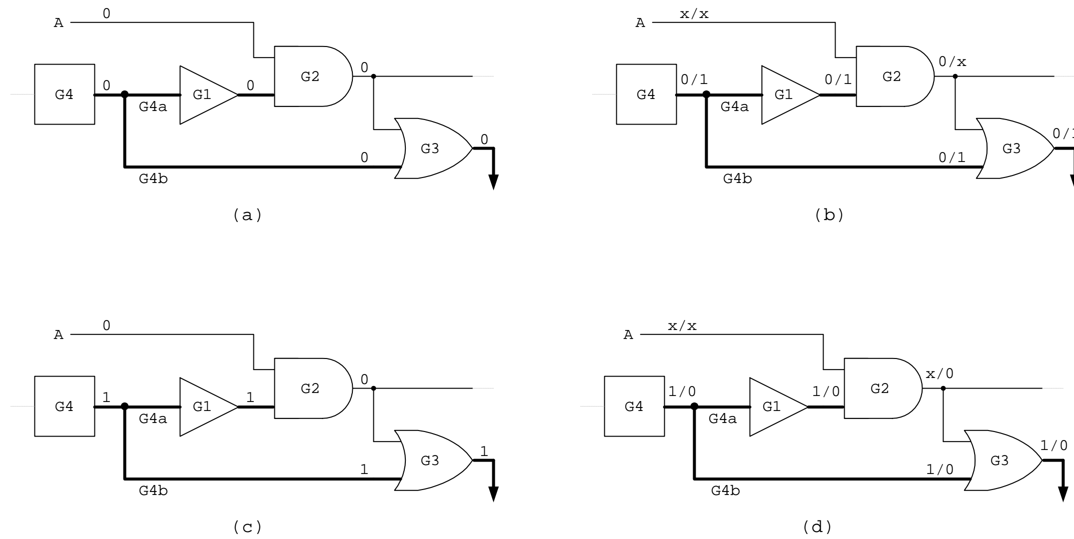


Figure 3.3: Limitations of the single-value relaxation.

can propagate to $G3$. This can be explained as follows. The value of $G4$ in the good machine (i.e., 1) can propagate to the output of $G3$ regardless of the value at $G2$. In the faulty machine, the value of $G4$ is 0 which requires another 0 at $G2$ in order to propagate to the output of $G3$. This requirement can be satisfied by $G1$ since its value in the faulty machine is 0.

3.1.3 Two Values Justification

The limitations discussed in the previous section can be avoided if we justify both good and faulty values of the circuit that are necessary to excite/propagate every newly detected fault. This is illustrated by the following examples.

Example 3.11: Consider the model shown in Figure 3.4 (a). Assume that the fault $G4/1$ is newly detected. Under this fault, the circuit lines have the following

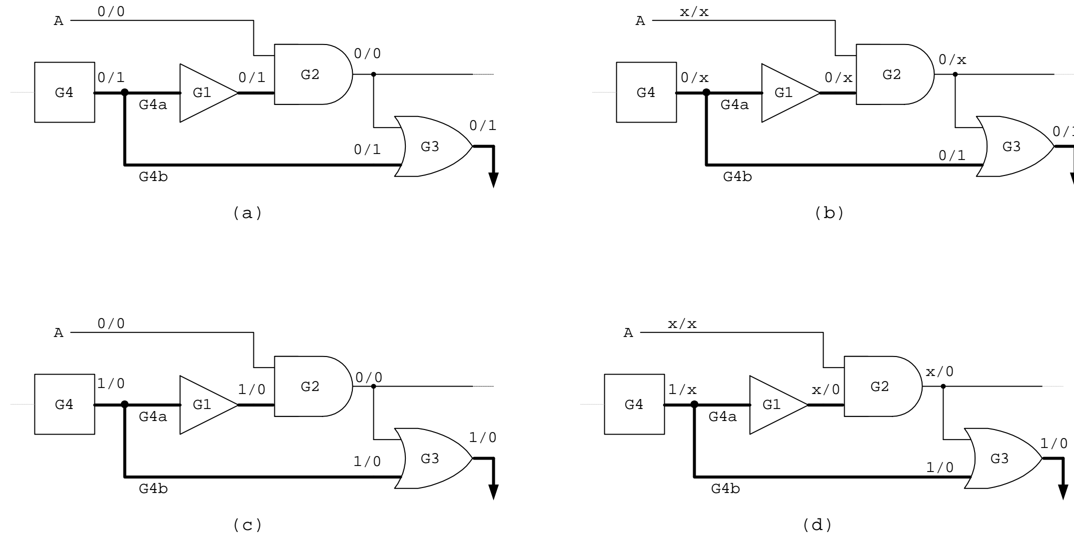


Figure 3.4: An example of the two-value relaxation.

combinations of good/faulty values: $A = 0/0$, $G4 = 0/1$, $G1 = 0/1$, $G2 = 0/0$, and $G3 = 0/1$. In order to detect the fault $G4/1$, as shown in Figure 3.4 (b), it is enough to justify the good/faulty values on the primary output $G3$. The good value of $G3$, which is 0, requires the good values on both $G2$ and $G4$ to be 0's. On the other hand, the faulty value on $G4$, which is 1, is enough to justify the faulty value on $G3$. Since only the good value of $G2$ is required, it can be justified by either the good value of A or the good value of $G1$. However, choosing the second assignment allows us to relax the input A. Notice that, we could justify the required value on $G2$ through $G1$ although it is reachable from the fault $G4/1$. This is because the faulty value on $G2$ has no effect on the fault detection. Now, we have two requirements on the stem $G4$: $0/X$ from $G1$ and $0/1$ from $G3$. Therefore, $G4$ need to justify a good value of 0 and faulty value of 1. Since the fault that need to be justified is excited

at $G4$, only the good value of $G4$ is required.

Example 3.12: Consider the model shown in Figurie 3.4 (c) under the fault $G4/0$. To justify the good/faulty values on the primary output $G3$, we need a 1 in the fault-free machine and 0 in the faulty machine. These requirements can be satisfied by the two assignments $G4 = 1/0$ and $G2 =$X/0 as shown in Figure 3.4 (d). Notice that the good value of $G2$ is not required to propagate the fault $G4/0$ to the primary output. This is because the value of $G4$ in the fault-free machine is 1 which can propagate to the output of $G3$ regardless of the value on the second input. The assignment $G2 =$X/0 can be satisfied by either one of the two assignments $A =$X/0 or $G1 =$X/0. However, choosing the second assignment allows us to relax the primary input A. Now, the line $G4$ needs to justify a value of 1 in the fault-free machine and a value of 0 in the faulty machine. Since the fault that needs to be justified is excited at $G4$, only the good value of $G4$ is required.

Unlike the single value justification technique, the two-values justification technique needs to justify the newly detected faults one by one because the second value (i.e., faulty value) is fault-dependent. However, the advantages gained by this approach is worth paying the price of fault-by-fault justification. In the next section, we describe our technique in a formal way.

3.2 Formal Description

Before describing the proposed techniques, we give the following definitions and lemmas.

Definition 3.1 *The value of an input is said to be controlling if it determines the value of the gate output regardless of the values of the other inputs.*

Definition 3.2 *The good value of a gate g, denoted by* **goodvalue(g)***, is the value of the gate under the fault-free machine.*

Definition 3.3 *The faulty value of a gate g, denoted by* **faultyvalue(g)***, is the value of the gate under the faulty machine.*

Definition 3.4 *The justify value of a gate g, denoted by* **justifyvalue(g)***, is the fault-free/faulty assignment that needs to be justified by g.*

Lemma 3.1 *If the fault-free/faulty values of one or more memory-elements are required to justify a fault f during some time frame t, then f can't be completely justified during that time frame. Hence, the justification of f has to continue during the time frame t-1.*

Lemma 3.2 *A gate g is said to satisfy the value v in the fault-free machine iff v='X' or* **goodvalue(g)***=v. Similarly, a gate g is said to satisfy the value v in the faulty machine iff v='X' or* **faultyvalue(g)***=v.*

Lemma 3.3 *If a line l is stuck-at some value v, then the faulty value of this line is not required, and it is enough to justify its good value.*

Due to the nature of sequential circuits (i.e., feedback from memory-elements), a fault excited in one time frame may propagate through several time farms before it gets detected. Hence, several time frames need to be traced back to justify such faults. Therefore, we need to store enough information about fault propagation, detection and justification in order to perform the justification process frame by frame. Four lists are used to store the the required information: *POJustificationList*, *FFJustificationList*, *FaultJustificationList*, and *FaultPropagationList*. In addition to these four lists, a fifth list, *RelaxedTestSet*, is used to store the relaxed test set. The purpose of each list is explained bellow.

- Faults which are newly detected in time frame t are stored in **POJustification-List**[t] in order to justify them starting from that time frame backwards.

- If a fault f can't be completely justified during time frame t, it is added to the **FFJustificationList**[$t-1$] in order to continue its justification process from $t-1$ backwards.

- All memory elements whose good/faulty values are required to justify the fault f, which has not been completely justified, are added to the **Fault-JustificationList(**f**)**.

- During fault simulation, if a fault f propagates to one or more memory-elements, then these memory elements are added to the **FaultPropagation-List**(f).

- The **RelaxedTestSet** represents the test set after relaxation. Initially, all the bits in this set are X's. However, more bits will be specified throughout the relaxation process in order to justify the detected faults.

3.2.1 Single-Value Justification

Algorithm 3.1 shows an outline of the single-value justification technique which consists of three phases. The first phase initializes the lists: *POJustificationList*, *FFJustificationList*, *FaultPropagationList*, *FaultJustificationList* and *RelaxedTest-Set*.

Fault simulation is performed in the second phase to identify newly detected faults in every time frame. Every newly detected fault in time frame t is stored in *POJustificationList*[t]. During fault simulation of test vector t, whenever a fault f propagates to a memory-element d, the tuple ($t+1$, d, *faultyvalue*(d)) is added to the *FaultPropagationList*[f]. The information in this list will be used to mark reachable lines of the circuit during the justification phase. It is important to point out here that during this phase, all logic values of the memory-elements are stored in a separate list. This will enable the third phase to perform logic simulation in a

Algorithm 3.1 Main Algorithm

```
(*Initialization phase*)
for every fault, f, in the fault list of the given circuit do
    Let FaultPropagationList[f] ← φ
    Let FaultJustificationList[f] ← φ
end for
for every test vector t do
    Let POJustificationList[t] ← φ
    Let FFJustificationList[t] ← φ
    for every primary input i do
        Let RelaxedTestSet[t][i] ←'X'
    end for
end for

(*Fault simulation phase*)
for t ← 1 to n do
    Fault simulate the circuit under test vector t
    for every fault, f, newly detected at primary output po do
        Add (f, po) to POJustificationList[t]
    end for
    for every fault f propagating to flip-flop d do
        Add (t + 1, d, faultyvalue(d)) to FaultPropagationList[f]
    end for
end for

(*Fault justification phase*)
for t ← n downto 1 do
    Logic simulate the circuit under the test vector t
    while FFJustificationList[t] ≠ φ do
        Remove f from FFJustificationList[t]
        MarkReachableLines(f, t)
        while FaultJustificationList[f]≠ φ do
            Remove FlipFlop from FaultJustificationList[f]
            Let i be the input of FlipFlop
            Add i to the EventList[level(i)]
        end while
        Justify(f, t)
    end while
    while POJustificationList[t] ≠ φ do
        Remove (f, po) from POJustificationList[t]
        MarkReachableLines(f, t)
        Add po to the EventList[level(po)]
        Justify(f, t)
    end while
end for
```

given time frame independent of the other time frames.

The third phase starts from the last time frame down to the first one. In every time frame, t, the algorithm performs the following. First, it logic simulates the circuit under the test vector t to determine the good value of every gate. Then, it checks the *FFJustificationList*[t] for any fault that has not been completely justified in time frame $t+1$. Such a fault is justified by justifying all memory-elements whose values are required to detect this fault in time frame $t+1$. These memory-elements and their corresponding values are stored in *FaultJustificationList*[f]. Next, it checks the *POJustificationList*[t] for newly detected faults and justify them. Justifying a fault, f, involves two operations: *marking reachable lines* and *backward justification*, which are described in Algorithm 3.2 and Algorithm 3.3 respectively.

Algorithm 3.2 marks all the gates which are reachable from given fault f. It starts by injecting the fault f at its corresponding line in the circuit, and adds the gate of that faulty line to an event list. Then, it sets the faulty values of the memory-elements in the circuit according to the faulty values propagating from the previous time frame $t-1$. These values are stored in the *FaultPropagationList*[f] in a descending order of the test vectors (i.e time frames). The outputs of every memory-element that receives a propagation of the fault f is added to the event list. Next, the event list is processed level by level starting from the minimal one. The faulty values of the gates in the current level are evaluated based on the faulty values of their inputs. If the faulty value of a gate g is found to be different from

Algorithm 3.2 MarkReachableLines(f,t)

```
(*Inject the fault*)
Inject the fault f at its corresponding line a
Add g (i.e., the gate of the faulty line a) to the EventList[level(g)]

(* Set the faulty values of the memory-elements according    *)
(* to the faulty values stored in the FaultPropagationList[f] *)
while FaultPropagationList[f] ≠ φ do
    Get(TestVector, FlipFlop, FaultyValue) from FaultPropagationList[f]
    if TestVector < t then
        exit while loop
    else
        Remove (TestVector, FlipFlop, FaultyValue)
        from FaultPropagationList[f]
        if TestVector = t then
            Let faultyvalue(FlipFlop) ← FaultyValue
            for every output i of FlipFlop do
                Add i to the EventList[level(i)]
            end for
        end if
    end if
end while

(*Process the event list level by level starting from the minimal one*)
for every level l of the circuit do
    while EventList[l] ≠ φ do
        Remove gate g from EventList[l]
        Fault evaluate g
        if goodvalue(g) ≠ faultyvalue(g) then
            Set g as reachable
            for every output i of g do
                Add i to the EventList[level(i)]
            end for
        end if
    end while
end for
```

its good value, then g is marked as reachable and all its outputs are added to the event list. Notice that the gates are added to the event list according to their levels in the circuit, and hence a gate will not be processed before its inputs.

Algorithm 3.3 gives the justification process of a given fault, f, at time frame t. In this algorithm, the event list is processed level by level starting from the maximum one. In each level, all the logical values of the stored gates are justified as follows. If g is a primary input (*PI*), then the logical value of g is required to detect the fault f. Therefore, the corresponding bit in the *RelaxedTestSet* is set to the logical value of g. If g is a memory-element (*DFF*), then the logical value of g can not be justified in the current time frame. Therefore, the fault f is added to the justification list of time frame $t-1$ (*FFJustificationList*[$t-1$]), and the memory-element is added to the *FaultJustificationList*[f]. If g is a single-input, XOR, or an XOR gate, then all its inputs need to be justified. Hence, all the inputs of g are added to the event list according to their levels in the circuit. If g is an AND, OR, NAND or NOR gate with a non-controlling value, then we need to justify all the inputs of g. However, if g has a controlling value, then we need to check if it has an unreachable input with a controlling value. If it has, then it is sufficient to justify that input. Otherwise, we check whether g is reachable or not. If it is not reachable, then we need to justify only the reachable inputs of g. Otherwise, all the inputs of g need to be justified.

Algorithm 3.3 JustifyFault(f, t)

```
(* Process the event list level by level starting from the maximum one. *)
for every level, l, of the circuit do
    while EventList[l] ≠ φ do
        Remove gate g from the EventList[l]
        case g is
            (1) PI:
                Let RelaxedTestSet[t][g] ← goodvalue(g)
            (2) DFF:
                Add g to the FaultJustificationList[f]
                Add f to the FFJustificationList[t − 1]
            (3) BUF, NOT, XOR, XNOR:
                for every input, i, of g do
                    Add i to the the EventList[i]
                end for
            (4) AND, OR, NAND, NOR:
                if g has a non-controlling value then
                    for every input, i, of g do
                        Add i to the the EventList[i]
                    end for
                else if there is an unreachable input, i, of g with controlling
                value then
                    Add i to the the EventList[i]
                else if g is unreachable then
                    for every reachable input, i, of g do
                        Add i to the the EventList[i]
                    end for
                else
                    for every input, i, of g do
                        Add i to the the EventList[i]
                    end for
                end if
        end case
    end while
end for
```

3.2.2 Two-Values Justification

Algorithm 3.4 shows an outline of the two-values justification technique which is almost the same as Algorithm 3.1. The two algorithms differ slightly in the third phase. Instead of marking reachable lines, Algorithm 3.4 computes only the faulty values of the circuits lines under the current fault. Then, it justifies good and faulty values that are necessary to detect the current fault. Faulty value computations and two-values justification are described in Algorithm 3.5 and Algorithm 3.6 respectively.

Algorithm 3.5 computes the faulty value of every gate in the circuit under a given fault f. It is exactly the same as Algorithm 3.2 except that it does not mark reachable gates.

Algorithm 3.6 gives the justification process of a fault f in time frame t. There are two possible scenarios to justify the fault f. First, f is not completely justified in the time frame $t{+}1$. In this case the justification of the fault will continue throughout the current time frame by satisfying the required values of all memory-elements stored in the *FaultJustificationList*[f]. Notice that, if the fault-free and/or the faulty values of one or more memory-elements are required to justify a fault f during the time frame $t+1$, then, the required memory-elements are added to the *FaultJustificationList*[f]. In the second scenario, the fault f is newly detected through primary output *po*. In this case it is justified by satisfying the fault-free/faulty values of *po*. In either

Algorithm 3.4 Main Algorithm

```
(*Initialization phase*)
for every fault, f, in the fault list of the given circuit do
    Let FaultPropagationList[f] ← φ
    Let FaultJustificationList[f] ← φ
end for
for every test vector t do
    Let POJustificationList[t] ← φ
    Let FFJustificationList[t] ← φ
    for every primary input i do
        Let RelaxedTestSet[t][i] ←'X'
    end for
end for

(*Fault simulation phase*)
for t ← 1 to n do
    Fault simulate the circuit under test vector t
    for every fault, f, newly detected at primary output po do
        Add (f, po) to POJustificationList[t]
    end for
    for every fault f propagating to flip-flop d do
        Add (t + 1, d, faultyvalue(d)) to FaultPropagationList[f]
    end for
end for

(*Fault justification phase*)
for t ← n downto 1 do
    Logic simulate the circuit under the test vector t
    while FFJustificationList[t] ≠ φ do
        Remove (f, po) from FFJustificationList[t]
        ComputeFaultyValues(f, t)
        Justify(f, po, t)
    end while
    while POJustificationList[t] ≠ φ do
        Remove (f, po) from POJustificationList[t]
        ComputeFaultyValues(f, t)
        Justify(f, po, t)
    end while
end for
```

Algorithm 3.5 ComputeFaultyValues(f,t)

```
(*Inject the fault*)
Inject the fault f at its corresponding line a
Add g (i.e., the gate of the faulty line a) to the EventList[level(g)]

(* Set the faulty values of the memory-elements according    *)
(* to the faulty values stored in the FaultPropagationList[f] *)
while FaultPropagationList[f]≠ φ do
    Get(TestVector, FlipFlop, FaultyValue) from FaultPropagationList[f]
    if TestVector< t then
        exit while loop
    else
        Remove (TestVector, FlipFlop, FaultyValue)
        from FaultPropagationList[f]
        if TestVector = t then
            Let faultyvalue(FlipFlop)←FaultyValue
            for every output i of FlipFlop do
                Add i to the EventList[level(i)]
            end for
        end if
    end if
end while

(*Process the event list level by level starting from the minimal one*)
for every level l of the circuit do
    while EventList[l]≠ φ do
        Remove gate g from EventList[l]
        Fault evaluate g
        if goodvalue(g) ≠ faultyvalue(g) then
            for every output i of g do
                Add i to the EventList[level(i)]
            end for
        end if
    end while
end for
```

Algorithm 3.6 JustifyFault(f,*po*,t)

```
if po=φ then
    (* f has not been completely justified in time frame t+1. In this *)
    (* case, inputs of the memory-elements stored in                  *)
    (* FaultJustificationList[f] are scheduled for justification.     *)
    while FaultJustificationList[f] ≠ φ do
        Remove (FlipFlop,JustifyValue) from FaultJustificationList[f]
        Let i be the input of FlipFlop
        Let justifyvalue(i) ← JustifyValue
        Add i to the EventList[level(i)]
    end while
else
    (* f is newly detected in this time frame. In this case, the primary      *)
    (* output at which the fault get detected is scheduled for justification. *)
    justifyvalue(po) ← (goodvalue(po), faultyvalue(po))
    Add po to the EventList[level(g)]
end if

(* Start justifying the scheduled gates.                                   *)
(* Process the event list level by level starting from the maximum one. *)
for every level, l, of the circuit do
    while EventList[l] ≠ φ do
        Remove gate g from the EventList[l]
        if justifyvalue(g) ≠ ('X','X') then
            JustiyGate(g,f,t)
        end if
    end while
end for
```

Algorithm 3.7 JustifyGate(g,f,t)

```
case g is
    (1) PI:
        GetCorrespondingValue(g,φ,v1,v2)
        if v1 ≠'X' then
            Let RelaxedTestSet[t][g]← v1
        else if v2 ≠'X' and g is not stuck-at v2 then
            Let RelaxedTestSet[t][g]← v2
        end if
    (2) DFF:
        GetCorrespondingValue(g,φ,v1,v2)
        Add (g, (v1,v2)) to FaultJustificationList[f]
        Add (f, φ) to the FFJustificationList[t − 1]
    (3) BUF|NOT:
        GetCorrespondingValue(g,φ,v1,v2)
        Let i be the input of g
        Let justifyvalue(i)←(v1,v2)
        Add i to the EventList[level(i)]
    (4) XOR|XNOR:
        for every input, i, of g do
            GetCorrespondingValue(g,i,v1,v2)
            Let justifyvalue(i)←(v1,v2)
            Add i to the EventList[level(i)]
        end for
```

case, the required gate(s) are added to an event list. Then, the list is processed level by level starting from the maximum one. The gates in every level are justified according to Algorithm 3.7.

In Algorithm 3.7, the required values on a gate g (i.e., *justifyvalue(g)*) are satisfied according to the following procedure. First, the algorithm determines the corresponding values (v_1/v_2) on the input(s) of the gate g. For example, if the required values on the output of an *inverter* are 0/1, then the corresponding requirements on

Algorithm 3.7 (Cont.) JustifyGate(g,f,t)

```
(5) AND|OR|NAND|NOR:
    GetCorrespondingValue(g,φ,v1,v2)
    if both v1 and v2 are controlling values of g then
        Find an input, i, of g that satisfy v1
        Find an input, j, of g that satisfy v2
        if i=j then
            Let justifyvalue(i)←(v1,v2)
        else
            Let justifyvalue(i)←(v1,'X')
            Let justifyvalue(j)←('X',v2)
        end if
        Add i to the EventList[level(i)]
        Add j to the EventList[level(j)]
    else if v1 is a controlling value of g then
        Find an input, i, of g that satisfy v1
        Let justifyvalue(i)←(v1,v2)
        Add i to the EventList[level(i)]
        for every input, j, of g such that j ≠ i do
            Let justifyvalue(j)←('X',v2)
            Add j to the EventList[level(j)]
        end for
    else if v2 is a controlling value of g then
        Find an input, i, of g that satisfy v2
        Let justifyvalue(i)←(v1,v2)
        Add i to the EventList[level(i)]
        for every input, j, of g such that j ≠ i do
            Let justifyvalue(j)←(v1,'X')
            Add j to the EventList[level(j)]
        end for
    else
        for every input, i, of g do
            Let justifyvalue(i)←(v1,v2)
            Add i to the EventList[level(i)]
        end for
    end if
end case
```

the input of this gate are 1/0. These values are determined using the procedure in Algorithm 3.8. The next step is to justify v_1/v_2 through the input(s) of g as follows. If g is a primary-input (*PI*), then we should specify its value whenever the good value is required (i.e not 'X'), or when the faulty value is required and there is no stuck-at fault on g. A requirement on a memory-element (*DFF*) can't be justified in the current time frame t. Therefore, the fault f is added to the justification list of time frame $t-1$, *FFJustificationList*$[t-1]$, and the DFF together with the required values v_1/v_2 are added to the *FaultJustificationList*$[f]$. If g is an *INVERTER* or a *BUFFER*, then its input is required to justify v_1/v_2. Hence, the input of g is added to the proper level in the event list of Algorithm 3.6. If the fault-free and/or the faulty value of an *XOR* or an *XNOR* gate is required, then the corresponding values (v_1/v_2) on every input of the gate are required as well. If g is an *AND*, *OR*, *NAND* or *NOR* gate, then we have four different possibilities. First, both v_1 and v_2 are controlling values of g. In this case, the algorithm searches for an input that satisfies both values and adds it to the event list. If v_1/v_2 can't be satisfied by a single input, then it will be justified through two different inputs. In case only v_1 is a controlling value of g, the algorithm will find an input i with a fault-free value that satisfies v_1. Since v_2 is a non-controlling value (or an 'X'), then all inputs of g are required to justify this value. Therefore, input i is added to the event list to justify the value v_1/v_2, while other inputs are added to the event list to justify the value X$/v_2$. Notice that if v_2 ='X', then the justify value of all inputs, except input

i, is X/X. Hence, these inputs will not be processed when they are removed from the event list in Algorithm 3.6. In the third case, only v_2 is controlling value of g. This can be handled exactly as done in the previous case except that v_2 is justified through one input, while v_1 is justified through all the inputs of g. Finally, if neither v_1 nor v_2 is a controlling value of g, then all the inputs of g are required to justify the value v_1/v_2. Hence, all inputs of g are added to the event list.

Algorithm 3.8 determines the input values corresponding to a required fault-free/faulty value on the output of a given gate g. If g is an *AND*, *OR*, *DFF* or *BUFFER*, then v_1/v_2 is the same as the required fault-free/faulty value unless the output of g is faulty. In this case, only v_1 is required since it is enough to satisfy the fault-free value of g and excite the faulty value on its output line. In the same way, the algorithm determines v_1/v_2 for *NOT*, *NAND* and NOR gates, except that v_1/v_2 are the complements of the fault-free/faulty values. For *XOR* and *XNOR* gates, the values v_1/v_2 may differ from one input to the other depending on the good/faulty values of that input. For input i of g, v_1 equals to the fault-free value of i unless the required fault-free value of g is 'X'. In this case, v_1 becomes 'X'. Similarly, v_2 equals to the faulty value of i unless the required faulty-value of g is 'X' or the output line of g is faulty, where v_2 becomes 'X' as well.

Algorithm 3.8 GetCorrespondingValue(g,h,v_1,v_2)

```
case g is
(1) PI:
Let (v1,v2)←justifyvalue(g)
(2) DFF|BUF|AND|OR:
Let (v1,v2)←justifyvalue(g)
if g is stuck-at v2 then
    Let v2 ←'X'
end if
(3) NOT|NAND|NOR:
Let (v1,v2)←justifyvalue(g)
if v1 ≠'X' then
    Let v1 ← 1 − v1
end if
if g is stuck-at v2 then
    Let v2 ←'X'
else if v2 ≠'X' then
    Let v2 ← 1 − v2
end if
(4) XOR|XNOR:
if v1 ≠'X' then
    Let v1 ←goodvalue(h)
end if
if g is stuck-at v2 then
    Let v2 ←'X'
else if v2 ≠'X' then
    Let v2 ←faultyvalue(h)
end if
end case
```

3.3 Selection Criteria

When justifying a controlling value through the inputs of a given gate, there could be more than one choice. In this case, the priority is given to the input that is already selected to justify other gates. Otherwise, cost functions are used to guide the selection. The cost functions give a relative measure on the number of primary inputs required to justify a given value. Hence, they can guide the relaxation procedure to justify the required values with the smallest number of assignments on the primary inputs.

The cost functions proposed in [29] combine the *regular* recursive controllability cost functions [30] with new cost functions called *fanout-based* cost functions. The regular cost functions are computed as follows. For every gate g, we compute two cost functions $C_{reg0}(g)$ and $C_{reg1}(g)$. For example, if g is an AND gate with i inputs, then the cost functions are computed as:

$$C_{reg0}(g) = \min_{i} C_{reg0}(i) \tag{3.1}$$

$$C_{reg1}(g) = \sum_{i} C_{reg1}(i) \tag{3.2}$$

These costs functions are computed for other gates in a similar manner. The fanout-based cost functions can be computed for an AND gate as follows. Let g be an AND gate with i inputs. Let $F(g)$ denotes the number of fanout branches of g.

Then, the fanout-based cost functions are computed as:

$$C_{fan0}(g) = \frac{\min_i C_{fan0}(i)}{F(g)} \tag{3.3}$$

$$C_{fan1}(g) = \frac{\sum_i C_{fan1}(i)}{F(g)} \tag{3.4}$$

The regular cost functions are accurate for fanout-free circuits. However, when fanouts exist, regular cost functions do not take advantage of the fact that a stem can justify several required values. This is illustrated in the following example.

Example 3.13: Consider the circuit shown in Figure 3.5. In order to justify the assignment on $G3$, the two assignments $G1 = 0$ and $G2 = 0$ are required. The first assignment, $G1 = 0$, can be justified by either one of the two assignments $A = 0$ or $B = 0$. Similarly, the assignment $G2 = 0$ can be justified by $B = 0$ or $C = 0$. If the regular cost functions are used, then $C_{reg0}(A) = C_{reg0}(B) = C_{reg0}(C) = 1$. According to these values any one of the following four assignments is possible: $\{A = 0,\ B = 0\}$, $\{A = 0,\ C = 0\}$, $\{B = 0,\ C = 0\}$, and $\{B = 0\}$. However, the last possibility, $\{B = 0\}$, requires only one assignment on the primary inputs, while other possibilities require two assignments. Now, if the fanout-based cost functions are used, then $C_{reg0}(A) = 1$, $C_{reg0}(B) = 1/2$, and $C_{reg0}(C) = 1$. It is clear that $B = 0$ is the proper choice to justify both $G1 = 0$ and $G2 = 0$ since its cost is less than the costs of the other two inputs.

In general, the fanout-based cost functions provide better selection criterion than

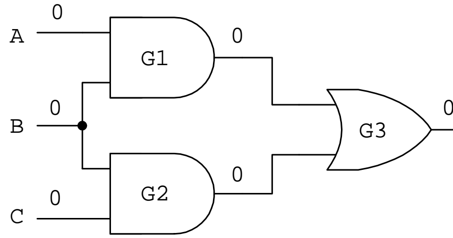


Figure 3.5: Circuit of Example 3.13.

the regular fanout cost functions. However, there are some cases where the regular fanout cost functions can perform better than the fanout-based cost functions as shown in the following example.

Example 3.14: Consider the circuit shown in Figure 3.6. To justify the required value on $G8$, we could either select $G7 = 0$ or $G = 0$ because their fanout-based cost functions are equal ($C_{fan0}(G7) = 1$ and $C_{fan0}(G) = 1$). However, if $G7 = 0$ is selected, then two primary input assignments are required, namely $B = 0$ and $E = 0$. On the other hand, using the regular cost functions will result in selecting $G = 0$ because its cost, i.e., $C_{reg0}(G) = 1$, is less than that of $G7 = 0$ which is $C_{reg0}(G7) = 2$. Thus, in this example using the regular cost functions has led to a better relaxation.

To take advantage of both cost functions, a weighted sum cost function of the two cost functions was proposed in [29]. The combined cost functions are defined as follows, where A is the weight of the regular cost function and B is the weight of

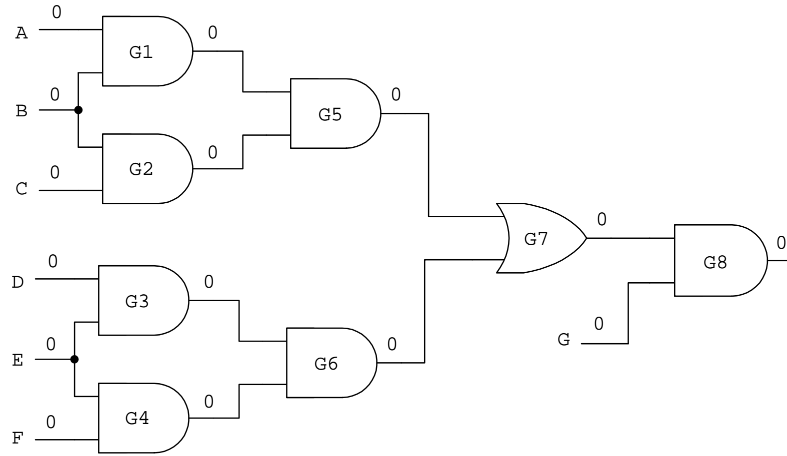


Figure 3.6: Circuit of Example 3.14.

the fanout-based cost function:

$$C_0(g) = A \cdot C_{reg0}(g) + B \cdot C_{fan0}(g) \tag{3.5}$$

$$C_1(g) = A \cdot C_{reg1}(g) + B \cdot C_{fan1}(g) \tag{3.6}$$

In synchronous sequential circuits, the controllability values of the circuit in one time frame depend on the controllability values computed in the current frame as well as the values computed in the previous frames. Therefore, the controllability values should be computed in an iterative manner starting from the first time frame. However, the iterative computation of the controllability values may cause the regular cost function to grow much faster than the fanout-based cost function such that the effect of the second cost function in the weighted sum becomes negligible. This is illustrated in the following example.

Example 3.15: Consider the iterative model shown in Figure 3.7. The controllability values of each gate are shown as a tuple of two values. The first value represents the regular cost, while the second value represents the fanout-based cost. Let the regular and fanout-based costs of all primary inputs equal to 1. Assume that the regular and fanout-based costs of the flip-flop in the first time frame equal to 1 and 1/2 respectively. Then, in the first time frame, the regular and fanout-based costs of $G3 = 1$ are 4 and $\frac{3}{2}$ respectively. After 10 time frames, the regular cost of $G3 = 1$ becomes 3070, while the fanout-based cost becomes $\frac{2047}{1024} \approx 2$.

The huge difference between the two costs in the previous example is due to the reconverging fanout branches of the flip-flop $G5$. Therefore, the regular cost of a flip-flop with reconverging fanout branches should be adjusted to reduce the difference between the two costs. This can be done as follows. Let g be a flip-flop with n fanout branches. Assume that m out of the n fanout branches reconverge at some gate in the circuit, then the regular cost of every one of these branches equals to the regular cost of g divided by m. In Figure 3.7, both branches of the flip-flop $G5$ reconverge at the gate $G3$. Therefore, the regular cost of each branch is computed as the regular cost of the flip-flop divided by 2. After adjusting the regular costs on the fanout branches of $G5$, the regular cost of $G3 = 1$ becomes 3 in the first time frame and 21 in the 10th time frame.

The cost functions described so far assume equal probability of a gate having a value of 0 or 1. Computing the controllability with this assumption is less accu-

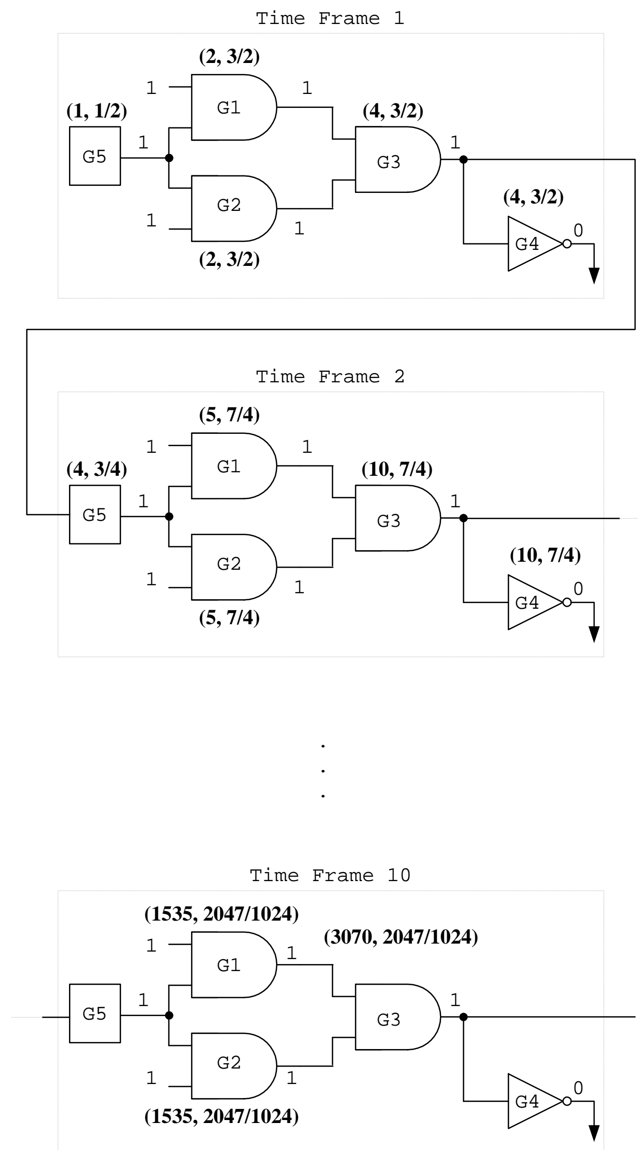


Figure 3.7: Illustration of the effect of reconverging fanouts on the regular cost.

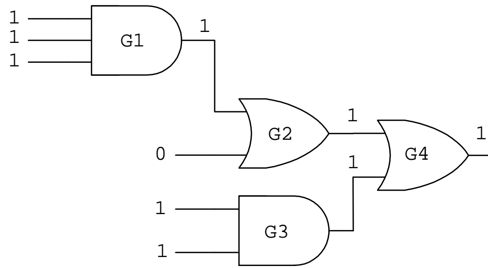


Figure 3.8: Circuit of Example 3.16.

rate than computing the controllability based on the actual logical values. This is illustrated in the following example.

Example 3.16: Consider the circuit shown in Figure 3.8. If we compute the regular costs assuming equal probability of a line being 0 or 1, then we will get the following costs: $C_{reg1}(G1) = 3$, $C_{reg1}(G2) = 1$, $C_{reg1}(G3) = 2$, and $C_{reg1}(G4) = 1$. These costs suggest to justify the assignment $G4 = 1$ through $G2$ which results in three assignments on the primary inputs. However, if the regular costs were computed based on the actual logical values, then we have the following costs: $C_{reg1}(G1) = 3$, $C_{reg1}(G2) = 3$, $C_{reg1}(G3) = 2$, and $C_{reg1}(G4) = 2$. In this case, $G3 = 1$ will be selected to justify the assignment $G4 = 1$. This assignment requires only two assignments on the primary inputs.

Based on the results of Example 3.15 and Example 3.16, we decided to use a weighted sum cost function of the adjusted regular cost function and the fanout-based cost function such that both cost functions are computed based on the actual logical values.

3.4 Worst Case Analysis

This section gives a worst case analysis for both space and time complexities of the proposed technique. In this analysis, we assume that the test set, fault list, and circuit structure are given as inputs. Thus, their memory and time requirements are not considered in the analysis.

Before giving the analysis, lets start with the following notations. The total number of test vectors, gates, memory elements, faults for a given circuit are given by n, G, D, and F respectively. The symbol NDF represents the number of faults detected by the given set out of the total number of faults F. The number of undetected faults up to time frame i is denoted by UDF_i. Finally, the number of unjustified faults in a time frame i is denoted by UJF_i.

3.4.1 Space Complexity

During fault simulation, the following lists are constructed: *PPInputList*, *FaultPropagationList*, and *POJustificaitonList*. The first list stores the logical values of the memory-elements in every time frame. The second list stores faults propagating from one time frame to the other. The third list stores the faults newly detected in every time frame.

The memory space needed for the first list is proportional to the number of memory-elements by the number of test vectors ($n \times D$). The memory space required

by the *FaultPropagationList* is proportional to the number of undetected faults in every time frame. For example, if we consider one time frame i, then the space required is at most UDF_i. Thus, the memory space needed for the whole list is proportional to $(UDF_1 + UDF_2 + \cdots + UDF_n)$. Since $F \geq UDF_1 \geq UDF_2 \geq \cdots \geq UDF_n$, we can say that the memory space required by the *FaultPropagationList* is proportional to the number of faults (F). The memory space needed for the *POJustificationList* is proportional to the number of detected faults (NDF). Since $F \geq NDF$, we can say that the memory space required by this list is proportional to the number of faults F.

During fault justification, the following lists are constructed. *RelaxedTestSet*, *FFJustificationList* and *FaultJustificationList*. The first list is used to store the relaxed test set. The second list is used to store the faults that could not be justified during any time frame. The third list is used to store memory elements whose values are required by the unjustified faults.

The memory space required by the *RelaxedTestSet* is proportional to the number to test vectors (n). The memory space required by the *FFJustificationList* at any time frame i is proportional to the number of faults that could not be justified in that time frame (UJF_i). Since the number of unjustified faults in any time frame can't exceed the total number of faults (F), we can say that the memory space required by the *FFJustificationList* is proportional to F. Similarly, the memory space required by the *FaultJustificationList* is proportional to the number of memory-elements by

the number of faults ($D \times F$). This is because the number of unjustified faults during any time frame is less than the number of faults (F), and the number of memory-element whose values are required by the unjustified faults is at most D.

Based on the above analysis, we observe that the memory space required by the proposed technique is proportional to $\max(nD, DF)$.

3.4.2 Time Complexity

As can be seen from Algorithm 3.4, the proposed technique consists of three main phases. In the first phase, the two lists *FaultPropagationList* and *FaultJustificationList* are processed for every fault, while the lists *POJustificationList*, *FFJustificationList* and *RelaxedTestSet* are processed for every test vector. Thus, the time complexity of this phase is proportional to the $\max(n, F)$.

In the second phase, the circuit is fault simulated for every test vector i in the test set. The time required to fault simulate the circuit in time frame i is proportional to $UDF_i \times G_i$, where UDF_i is the number of undetected faults up to time frame i, and G_i is the maximum number of gates processed by the fault simulated for any one of the undetected faults. Notice that the undetected faults at any time frame is at most F, and the maximum number of gates is G. Thus, the time complexity of the fault simulation phase is proportional to $(n \times F \times G)$.

The justification phase consists of three main parts. For every test vector, starting from the last one, the circuit is first simulated logically. Next, all faults that

could not be justified during the previous time frames are processed during the current time frame. Then, all newly detected faults are justified. In logic simulation, at most G gates are processed. Thus, the time complexity of the logic simulation at any time frame is proportional to G.

The main operations in the second and third parts are *ComputeFaultyValues* and *Justify*. The time complexity of each one of these operations is proportional to the maximum number of gates processed for a given fault. Therefore, the time complexity of these operations is also proportional to G. Notice that the number of unjustified faults at any time frame in the second part is at most F. Similarly, the number of newly detected faults at any time frame in the third part is bounded by F as well. Therefore, the time complexity of the fault justification phase is proportional to $(n \times F \times G)$.

In conclusion, the time complexity of the proposed technique is proportional to $(n \times F \times G)$, where n is the number of test vectors, F is the number of detectable faults, and G is the number of gates in the given circuit.

Chapter 4

Experimental Results

In order to demonstrate the effectiveness of the two-values justification technique, we have preformed some experiments on a number of the ISCAS89 benchmark circuits shown in Table 4.1. The first column gives the name of the benchmark circuit. Columns 2 to 4 give the number of primary inputs, number of primary outputs, number of D flip-flops, and the total number of gates respectively.

The experiments were run on a SUN Ultra60 (UltraSparc II 450MHz) with a RAM of 512MB. We have used test sets generated by HITEC[31]. In addition to that, we have used the fault simulator HOPE[32] for fault simulation purposes.

This chapter is organized as follows. Next, we compare the two-values justification technique with the bitwise-relaxation technique. Then, we investigate the effect of the cost functions used in the two-values relaxation technique on the percentage of X's. After that, we examine different aspects of the two-values justification tech-

Table 4.1: Benchmark circuits.

Circuit Name	**No. Inputs**	**No. Outputs**	**No. Flip-Flops**	**No. Gates**
s1423	17	5	7	490
s1488	8	19	6	550
s1494	8	19	6	558
s3271	26	14	116	1035
s3330	40	73	132	815
s3384	43	26	183	1070
s4863	49	16	104	1600
s5378	35	49	179	1004

nique such as single-value verses two-values justification, computation of the cost functions using actual and general values, and the effect of using the modified regular cost function on the consistency of the solutions.

4.1 Comparison with Bitwise-Relaxation

In Table 4.2, we compare the two-values justification technique with the bitwise-relaxation method. The two techniques are compared in terms of the percentage of X's extracted, and the CPU time taken for relaxation. It is important to point out here that in order to have a fair comparison between our technique and the bitwise-relaxation method, we have implemented the bitwise-relaxation method such that all faults detected at a particular time frame remain detected in the same time frame after relaxation. The percentage of X's are shown in Table 4.2 for both constrained and unconstrained bitwise-relaxation respectively.

It is clear that, for all the circuits, the CPU time taken by our technique is less

than that of the bitwise-relaxation method by several orders of magnitude. The bitwise-relaxation method requires enormous CPU times, and hence is impractical for large circuits.

The percentage of X's obtained by our technique is also close to the percentage of X's obtained by the bitwise-relaxation method for most of the circuits. The difference in the percentage of X's ranges between 1% and 7% (3% and 11% when compared with the unconstrained bitwise-relaxation technique), while the average difference is about 3% (6% when compared with the unconstrained bitwise-relaxation technique).

It should be observed that when justifying a fault that is detected through more than one output, the two-values justification technique will select one of these primary output to justify the detected fault without taking into consideration that some primary outputs can lead to more relaxation than others. As an example consider the circuit shown in Figure 4.1 under the fault $e/1$. This fault is detected through the primary outputs g and h. If we justify the fault through g, then only the assignment $ab = 11$ is required. Hence, the test vector can be relaxed to $abcd$ =11XX. However, if we choose h to justify the fault, then we need at lease three assignments on the primary inputs (i.e., $abcd$ =110X or $abcd$ =11X0). The bitwise-relaxation method, on the other hand, implicitly chooses the output for detecting a fault that maximizes the number of X's according to the order used. For the circuit in Figure 4.1, the bitwise-relaxation method starts by setting a =X, then it fault simulates

Table 4.2: Test relaxation comparison between the two-values justification (TVJ) technique and the bitwise-relaxation method.

	Percentage of X's			CPU Time (seconds)	
Circuit	**Bitwise-Relaxation**	**TVJ Technique**	**Diff.**	**Bitwise-Relaxation**	**Proposed Technique**
s1423	69.922/74.392	63.020	6.902/11.37	943	1.750
s1488	76.154/81.090	72.244	3.910/8.846	12553	2.417
s1494	76.295/82.962	72.741	3.554/10.22	13146	3.100
s3271	83.894/85.527	81.908	1.986/3.619	87726	8.033
s3330	87.738/90.082	85.506	2.232/4.576	115585	5.633
s3384	78.579/81.655	77.755	0.824/3.900	16549	2.533
s4863	84.832/87.542	81.735	3.097/5.807	162894	7.800
s5378	87.738/88.969	86.056	1.682/2.913	218137	20.35
Avg.	80.644/84.027	77.621	3.023/6.406		

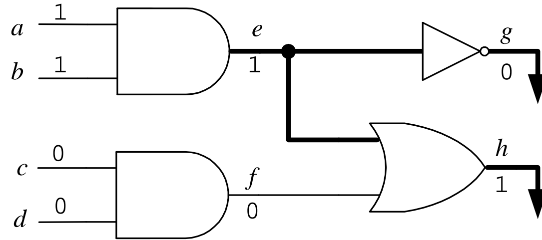


Figure 4.1: Effect of PO's selection on test vector relaxation.

the circuit under the test vector $abcd$ =X100. Since the fault $e/1$ is not detected under this test vector, a is set back to 1. Next, it tries to relax b, and fails. Then, it sets c =X and fault simulates the circuit under the test vector $abcd$ =11X0. Since the fault $e/1$ is detected at g, c remains as 'X'. The same thing happens when we set d =X. Thus, the resulting test vector is $abcd$ =11XX.

Table 4.3: Cost function effect on the extracted percentage of X's.

Circuit	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=10	A=1 B=30	A=1 B=50	A=1 B=70	A=1 B=90
s1423	37.882	50.863	57.059	62.431	63.686	63.961	**64.093**	63.020
s1488	44.448	**72.457**	56.624	66.218	69.968	71.767	71.571	72.244
s1494	43.515	72.661	57.410	66.687	70.502	71.767	72.098	**72.741**
s3271	57.361	78.860	**82.060**	82.017	82.033	81.979	81.892	81.908
s3330	66.548	85.251	84.805	85.446	85.407	85.484	**85.506**	**85.506**
s3384	69.247	71.703	77.755	**77.799**	77.784	77.755	77.755	77.755
s4863	72.114	78.934	**83.406**	82.846	82.582	82.393	82.038	81.735
s5378	77.788	85.692	82.130	84.110	85.053	85.085	85.094	**86.056**
Avg.	58.613	74.553	72.656	75.944	77.127	77.459	77.499	**77.621**

4.2 Experiments on Cost Functions

Table 4.3 shows the effect of varying the weights of the regular and fanout-based cost functions on the percentage of X's. The weight A is for the adjusted regular cost function and the weight B is for the fanout-based cost function. As can be seen from the table, the use of cost functions results in higher percentage of X's. Notice that the second column in the table shows the percentage of X's obtained when no selection criteria (i.e., no cost functions) are used at all. Also, it is worth mentioning here that neither the adjusted regular cost function nor the fanout-based cost function consistently performs better for all the circuits. However, when both cost functions are combined, better results are obtained. The table, also, shows that a weight of 1 for the adjusted regular cost function and a weight of 90 for the fanout-based cost function seems to be a good heuristic as it gives the highest percentage of X's on average.

4.3 Examining Different Aspects of the Two-Values Justification Technique

In this section, we will investigate the the following aspects. First, we will investigate the effect of two-values justification on the percentage of X's as compared to single-value justification. Single-value justification will be denoted by SVJ. Then, we will show the effect of using cost functions based on actual logic values on the percentage of X's. For this purpose, we will compare the percentage of X's obtained by the two-values justification technique with those obtained by a modified version that uses general values when computing the cost functions. The modified version will be denoted by $GVCF$. Finally, we will show the effect of using the adjusted regular cost functions on the percentage of X's. Again we will use a modified version of the two-values justification technique, denoted by $UACF$, which uses unadjusted regular cost functions without modification.

Each one of the above versions were applied on the circuits of Table 4.1 using the following weights of A and B respectively: $\{$(0, 1), (1, 0), (1, 5), (1,1 0), $\cdots$, (1, 100)$\}$. Among these 22 combinations of A and B, we will show only 8 snap-shots for each version. These snap-shots represent the best results of each version in terms of the average percentage of X's.

Table 4.4 shows the percentage of X's obtained by the SVJ. As can be seen from the table, the highest percentage of X's on average is obtained using the weights

$\{A = 1,\ B = 95\}$. Table 4.5 gives a comparison between the results in this column and those obtained by the two-values justification technique (see Table 4.2) using the weights $\{A = 1,\ B = 90\}$. It is clear that the percentage of X's obtained by the poposed technique, which uses two-values justification, is higher than the percentage of X's obtained by single-value justification for all the 8 circuits. The difference in terms of the percentage of X's varies between 0% and 7%, and the average difference is about 2%.

Table 4.6 shows the percentage of X's obtained by the GVCF. By looking to the average percentage of X's at each column, we find that the GVCF achieves the highest percentage of X's on average under the weights $\{A = 1,\ B = 0\}$. Table 4.7 compares the percentage of X's obtained by the two-values justification technique using the weights $\{A = 1,\ B = 90\}$ with those obtained by the GVCF using the weights $\{A = 1,\ B = 0\}$. As can be seen from this table, the two-values justification technique performs better than the GVCF for most of the circuits, especially the first circuit where the difference in percentage of X's is more than 17%. The average difference between the two technique is about 3%.

Table 4.8 shows the percentage of X's obtained by the UACF. Consider the results in the second and fifth columns with the weights $\{A = 0,\ B = 1\}$ and $\{A = 1,\ A = 50\}$ respectively. These two columns give inconsistent results for the second and third circuits, namely **s1488** and **s1494**. Although the weights $\{A = 1,\ A = 50\}$ improve most of the results in the second column, they cause an

Table 4.4: Percentage of X's obtained by SVJ using different weights.

Circuit	**A=1 B=65**	**A=1 B=70**	**A=1 B=75**	**A=1 B=80**	**A=1 B=85**	**A=1 B=90**	**A=1 B=95**	**A=1 B=100**
s1423	56.275	56.275	56.275	56.314	56.275	56.196	56.275	55.882
s1488	69.765	69.765	69.957	70.000	70.064	70.182	70.310	70.374
s1494	69.950	70.030	70.161	70.171	70.211	70.311	70.412	70.512
s3271	78.160	78.111	78.051	78.030	78.008	78.003	77.997	77.883
s3330	85.441	85.450	85.506	85.519	85.536	85.506	85.436	85.541
s3384	77.235	77.192	77.192	77.178	77.178	77.178	77.178	77.178
s4863	82.141	82.125	82.042	81.928	81.806	81.719	81.672	81.637
s5378	84.480	84.449	84.539	84.583	84.615	84.624	84.586	84.612
Avg.	75.431	75.425	75.465	75.465	75.462	75.465	**75.483**	75.452

enormous drop in the percentage of X's obtained for the circuits s1488 and s1494. To investigate this problem, we compared the values of the regular and fanout-based cost functions on the memory-elements of the two circuits, and we found that the values of the fanout-based cost functions are negligible compared to those of the regular cost functions. This indicates that the memory-elements of the two circuits have lots of reconverging fanouts, such that the regular cost function grows much faster the fanout-based cost function. This problem can be solved by modifying the regular cost function to account for reconverging fanout-branches as explained in Chapter 3. Columns 3 and 9 in Table 4.2 show how the results remain consistent with different weights (i.e., 0 and 1) for the modified-regular cost function.

Table 4.5: Test relaxation comparison between TVJ and SVJ.

Circuit	**TVJ**	**SVJ**	**Diff.**
s1423	63.020	56.275	**6.745**
s1488	72.244	70.310	1.934
s1494	72.741	70.412	2.329
s3271	81.908	77.997	3.911
s3330	85.506	85.436	0.070
s3384	77.755	77.178	0.577
s4863	81.735	81.672	0.063
s5378	86.056	84.586	1.470
Avg.	77.621	75.483	2.137

Table 4.6: Percentage of X's obtained by GVCF using different weights.

Circuit	**A=1 B=0**	**A=1 B=5**	**A=1 B=10**	**A=1 B=15**	**A=1 B=20**	**A=1 B=25**	**A=1 B=30**	**A=1 B=35**
s1423	45.569	49.412	49.569	49.686	49.725	49.255	48.863	48.667
s1488	70.150	68.846	68.365	68.355	68.355	68.355	68.301	68.301
s1494	72.339	70.542	70.633	70.633	70.633	70.582	70.592	70.592
s3271	82.174	80.921	80.975	81.089	80.281	80.200	78.024	77.970
s3330	84.619	85.143	84.931	84.771	84.585	83.045	83.382	83.400
s3384	77.842	77.784	77.784	76.383	74.014	74.014	74.014	74.014
s4863	83.102	80.754	79.009	78.174	76.960	76.842	76.759	76.637
s5378	82.303	84.179	84.207	84.737	85.066	85.063	85.069	85.078
Avg.	**74.762**	74.698	74.434	74.228	73.702	73.419	73.126	73.082

Table 4.7: Test relaxation comparison between TVJ and GVCF.

Circuit	**TVJ**	**GVCF**	**Diff.**
s1423	63.020	45.569	**17.451**
s1488	72.244	70.150	2.094
s1494	72.741	72.339	0.402
s3271	81.908	82.174	-0.266
s3330	85.506	84.619	0.887
s3384	77.755	77.842	-0.087
s4863	81.735	83.102	-1.367
s5378	86.056	82.303	3.753
Avg.	77.621	74.762	2.858

Table 4.8: Percentage of X's obtained by UACF using different weights.

Circuit	A=0 B=1	A=1 B=40	A=1 B=45	A=1 B=50	A=1 B=55	A=1 B=60	A=1 B=65	A=1 B=70
s1423	50.863	66.549	66.667	66.784	66.745	66.863	66.863	66.902
s1488	**72.521**	48.921	48.921	**48.942**	48.900	48.771	48.750	48.622
s1494	**72.671**	51.396	51.396	**51.396**	51.355	51.235	51.255	51.084
s3271	81.062	82.462	82.462	82.478	82.489	82.489	82.494	82.494
s3330	85.251	85.467	85.458	85.476	85.493	85.519	85.541	85.536
s3384	71.790	77.799	77.770	77.755	77.755	77.755	77.755	77.755
s4863	77.630	83.153	83.165	83.169	83.169	83.153	83.130	83.126
s5378	85.692	86.350	86.344	86.347	86.347	86.357	86.303	86.269
Avg.	74.685	72.762	72.773	72.793	72.782	72.768	72.761	72.724

Conclusion

Testing systems-on-a-chip (SOC) involves applying huge amounts of test data, which must be stored in the tester memory and transferred during test application to the circuit under test (CUT). Therefore, practical techniques, such as compression and compaction, are used to reduce the amount of test data in order to reduce both the total testing time and the memory requirements for the tester. Some of the existing compression/compaction techniques require the test data to be partially specified, while others can benefit from partially specified test sets either directly or by specifying the don't care values in these test sets in a way that improves their efficiency.

In this thesis, we have proposed a new technique for relaxing test-patterns in synchronous sequential circuits. The proposed technique is faster than the bitwise-relaxation method by several order of magnitude. The percentage of X's obtained by our technique is also close to the percentage of X's obtained by bitwise-relaxation for most of the circuits. The difference in the percentage of X's ranges between 1% and 7%, and the average difference is about 3%. It should be observed that the

bitwise-relaxation method, as explained in Chapter 4, implicitly chooses the output for detecting a fault that maximizes the umber of X's according the order used. However, our technique does not do any optimization in selecting the best output for detecting a fault. This can be investigated in future work.

Having an efficient test relaxation technique is crucial for improving the efficiency of compression and compaction. To see this, consider the compression and compaction techniques described in Chapter 2. The need for test relaxation for LFSR-reseeding [2, 3] is obvious since these techniques require the test vectors to be partially specified. However, compression techniques which require fully specified test data can benefit from relaxed test sets by specifying the don't care values in a way that improves their efficiency. For example, variable-to-fixed-length coding [4] and variable-to-variable-length coding [5, 6] are known to perform better for long runs of 0's. Hence, assigning 0's to the don't care values in the test set will improve the efficiency of these techniques. Compaction techniques can also benefit from partially specified test sets. For example, increasing the number of X's in a test set will reduce the number of conflicts that may occur when we overlap two test sequences. This is because a don't care value, 'X', can be merged with any one of the values: '0', '1', and 'X'. More importantly, the proposed test relaxation technique allows us to extract self-synchronizing test sequences which start from all unspecified states (i.e., all memory-elements have don't care values). Since self-synchronizing test sequences can be reordered without affecting their fault coverage, they can be used to improve

the efficiency of test sequence compaction by reverse-order fault simulation. The self-synchronizing test sequences can be extracted by our technique as follows. All memory-elements which have no effect on detected faults will not be selected during the justification process. Thus, we can relax the values of all unselected memory-elements. Then, any time frame with all memory-elements set to X's is considered as the start of a new self-synchronizing test sequence. Application of test relaxation in achieving more effective test compression and compaction will be investigated in future work.

Bibliography

[1] Y. Zorian, E. J. Marinissen and S. Dey. Testing Embedded-Core Based System Chips. In *Proc. International Test Conference*, pages 130–143.

[2] B. Koenemann. LFSR-Coded Test Patterns for Scan Designs. In *Proc. European Test Conference*, pages 237–242, 1991.

[3] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois. Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Feedback Shift Registers. In *IEEE International Test Conference*, pages 120–129, Sep. 1992.

[4] A. Jas and N. Touba. Test Vector Decompression via Cyclical Scan Chains and Its Application to Testing Core-Based Designs. In *Proc. International Test Conference*, pages 458–464, 1998.

[5] A. Chandra and K. Chakrabarty. Test Data Compression for System-On-a-Chip using Golomb Codes. In *Proc. of IEEE VLSI Test Symposium*, 2000.

[6] A. Chandra and K. Chakrabarty. Frequency-directed run-length (FDR) codes with application to system-on-a-chip test data compression. In *19th IEEE Proceedings on. VTS*, pages 42–47, 2001.

[7] T. Yamaguchi, M. Tilgner, M. Ishida and D. S. Ha. An Efficient Method for Compressing Test Data. In *Proc. International Test Conference*, pages 79–88, Nov. 1997.

[8] V. Iyengar, K. Chakrabarty and B. Murray. Huffman Encoding of Test Sets for Sequential Circuits. *IEEE Trans. on Instrumentation and Measurement*, 47(1):21–25, Feb. 1998.

[9] A. El-Maleh, S. Zahir, and E. Khan. A Geometric-Primitive-Based Compression Scheme for Testing Systems-on-a-Chip. In *Proc. IEEE VLSI Test Symposium*, Apr. 2001.

[10] I. Pomeranz and S. M. Reddy. On Generating Compact Test Sequences for Synchronous Sequential Circuits. In *Proc. EURODAC*, pages 105–110, Sep. 1995.

[11] S. Chakradhar and A. Raghunathan. Bottleneck Removal Algorithm for Dynamic Compaction and Test Cycle Reduction. In *Proc. EURODAC*, pages 98–104, Sep. 1995.

[12] I. Pomeranz and S. M. Reddy. Dynamic Test Compaction for Synchronous Sequential Circuits using Static Compaction Techniques. In *Proc. 26th Fault-Tolerant Computing Symp.*, pages 53–61, June 1996.

[13] E. M. Rudnick and J. H. Patel. Simulation-based Techniques for Dynamic Test Sequence Compaction. In *Proc. Intl. Conf. on Computer Aided Design*, Nov. 1996.

[14] R. Roy, T. Niermann, J. Patel, J. Abraham, and R. Saleh. Compaction of ATPG-Generated Test Sequences for Sequential Circuits. pages 382–385, Nov. 1988.

[15] I. Pomeranz and S. M. Reddy. On static compaction of test sequences for synchronous sequential circuits. In *Proc. Design Automation Conf.*, pages 215–220, 1996.

[16] I. Pomeranz and S. M. Reddy. Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits. In *Proc. Int. Conf. on Computer Design*, pages 360–365, Oct. 1997.

[17] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Fast Static Compaction Algorithms for Sequential Circuit Test Vectors. *IEEE Trans. on Computers*, 48(3):311–322, March 1999.

[18] A. Jas, J. G. Dastidar and N. Touba. Scan Vector Compression/Decompression Using Statistical Coding. In *Proc. IEEE VLSI Test Symposium*, pages 114–120, 1999.

[19] A. El-Maleh and R. Al-Abaji. Extended Frequency-Directed Run-Length Code with Improved Application to System-on-a-Chip Test Data Compression. In *Proc. IEEE ICECS 2002*, Sep. 2002.

[20] K. Chakrabarty, B. T. Murray, J. Liu and M. Zhu. Testing Width Compression for Built-In-Self Tesing. In *Proc. IEEE International Test Conference*, pages 328–337, Nov. 1997.

[21] T. Cover and J. Thomas. *Elements of Information Theory.* New York: John Wiley, 1991.

[22] G. Held. *Data Compression: Techniques and Applications.* Chichester, U.K.: Willey, 1991.

[23] M. Jakobssen. Huffman Coding in Bit-vector Compression. *Inf. Process. Lett.*, 7:304–307, Oct. 1978.

[24] I. Pomeranz and S.M. Reddy. Procedures for Static Compaction of Test Sequences for Synchronous Sequential Circuits. *IEEE Transactions on Computers*, 49(6), June 2000.

[25] R. Guo, I. Pomeranz and S.M. Reddy. On Speeding-Up Vector Restoration Based Static Compaction for Sequential Circuits. pages 467–471, Nov. 1998.

[26] R. Guo, I. Pomeranz and S.M. Reddy. Procedures for Static Compaction of Test Sequences for Synchronous Sequential Circuits. pages 583–587, Feb. 1998.

[27] M. S. Hsiao and S. T. Chakradhar. State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits. In *Proc. Design Automation and Testing Europe Conf.*, pages 577–582, Feb. 1998.

[28] S. Kajihara and K. Miyase. On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits. In *Proc. IEEE ICCAD*, pages 364–369, Nov. 2001.

[29] A. El-Maleh and A. Al-Suwaiyan. An Efficient Test Relaxation Technique for Combinational & Full-Scan Sequential Circuits. In *Proc. IEEE VLSI Test Symposium*, 2002.

[30] M. Abramovici, M. Breuer and A. Friedman. *Digital System Testing and Testable Design.* IEEE Press, 1990.

[31] Thomas M. Niermann and Janak H. Patel. HITEC: A test generation package for sequential circuits. In *Proc. of the European Conference on Design Automation (EDAC)*, pages 214–218, 1991.

[32] H. K. Lee and D. S. Ha. HOPE: An Effecient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Trans. on Computer Aided Design*, 15(9):1048–1058, Sep. 1996.