INFORMATION TO USERS

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

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

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

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

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

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

A TWO-DIMENSIONAL GEOMETRIC-SHAPES-BASED COMPRESSION SCHEME FOR DETERMINISTIC TESTING OF SYSTEMS-ON-A-CHIP

BY

ESAM ALI HASAN KHAN

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

JUNE 2001

UMI Number: 1406109

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHARAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

ESAM ALI HASAN KAHN

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
(Thesis Advisor)

7 Jun 01
Department Chairman

Dr. Mostafa Abd-El-Barr
(Member)

Dean of Graduate Studies

Dr. Alaaeldin Amin
(Member)

9/6/2001
Date

DEDICATION

In the Name of Allah, the Most Gracious, the Most Merciful.

To

My parents, who opened the way for me to success

And to

My wife, who was patient and supported me a lot

ACKNOWLEDGEMENT

All praise be to Allah, Subhanahu-wa-Ta'ala, for his limitless blessing and guidance. May Allah bestow peace on his prophet, Muhammad (Peace and blessing of Allah be upon him) and his family.

All my appreciation and thanks to my thesis advisor, Dr. Aiman H. El-Maleh, for his guidance and help all the way till the achievement of this thesis. Thanks are also due to Dr. Saif Al-Zahir, who started the work with us and had good ideas and suggestions.
I would like also to thank my thesis committee members, Prof. Mostafa I. Abd-El-Barr and Dr. Alaaeldin Amin for their cooperation and constructive comments.

All my thanks to the Computer Engineering Department, and especially its chairman, Prof. Sadiq M. Sait, for all support and motivation I got. I also thank the dean of the College of Computer Sciences and Engineering, Dr. Jarallah Al-Ghamdi. My deepest acknowledgement is due to King Fahd University of Petroleum and Minerals (KFUPM) for all support and facilities.

Last, but not least, thanks to all my colleagues and friends, who encouraged me a lot in my way to the achievement of this work.

CONTENTS

DEDICATION iii

ACKNOWLEDGEMENT iv

CONTENTS v

LIST OF TABLES ix

LIST OF FIGURES x

THESIS ABSTRACT xii

خلاصة الرسالة xiii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. BACKGROUND MATERIAL 8

CHAPTER 3. LITERATURE REVIEW 14

3.1. COMPACTION TECHNIQUES 14

3.1.1. Compaction Techniques for Combinational Circuits 15

3.1.1.1. Static Compaction 15

3.1.1.2. Dynamic Compaction 18
3.1.2. Compaction Techniques for Sequential Circuits 23
3.1.2.1. Static Compaction 23
3.1.2.2. Dynamic Compaction 27
3.2. COMPRESSION TECHNIQUES 30
3.2.1. Basic Compression Schemes 31
3.2.2. PRG- and BIST-Based Compression Techniques 33
3.2.2.1. Test Width Compression 34
3.2.2.2. Variable-Length Reseeding 38
3.2.2.3. Design For High Test Compression (DFHTC) 41
3.2.3. Deterministic Compression Techniques 43
3.2.3.1. Run-Length Coding 44
3.2.3.2. Statistical Coding 57
3.2.3.3. Compression by Replacement Words 59
3.3. OVERALL COMPARISON 61
3.4. CONCLUDING REMARKS 63
CHAPTER 4. PROPOSED COMPRESSION SCHEME 65
4.1. MOTIVATION 65
4.2. METHODOLOGY 67
4.3. SORTING TEST VECTORS 71

4.4. ENCODING PROCESS 76
4.4.1. Test Set Partitioning 76
4.4.2. Encoding Blocks 77
4.4.3. Time Complexity 81
4.5. AN ILLUSTRATIVE EXAMPLE 82
4.6. CONCLUDING REMARKS 87
CHAPTER 5. DECOMPRESSION PROCESS 89
5.1. SOFTWARE DECODER 90
5.2. HARDWARE DECODER 93
5.2.1. Data Path Implementation 93
5.2.2. Implementation of the FSM 102
5.3. DECODER INTERFACE 106
5.3.1. Interface of the software decoder 107
5.3.2. Interface of the hardware decoder 108
5.4. CONCLUDING REMARKS 109
CHAPTER 6. EXPERIMENTAL RESULTS 111
6.1. EFFECT OF DIFFERENT FACTORS ON COMPRESSION RATIO 112
6.2. TIMING PERFORMANCE 120
6.3. STATISTICS ON BLOCK ENCODING 122
6.4. COMPARISON WITH OTHER TECHNIQUES 125

6.5. CONCLUDING REMARKS 127

CHAPTER 7. CONCLUSION AND FUTURE WORK 129

APPENDEX A. VHDL CODE FOR THE HARDWARE DECODER 134

REFERENCES 162

VITA 168

LIST OF TABLES

TABLE 3.1. TRANSITION TABLE FOR RUN-LENGTH CODING 47

TABLE 3.2. GOLOMB CODE (M = 4) 54

TABLE 3.3 . AN EXAMPLE OF FDR CODE 56

TABLE 4.1. PRIMITIVE SHAPES USED IN THE PROPOSED SCHEME 71

TABLE 4.2. WEIGHTS FOR THE 0-DISTANCE BETWEEN TWO TEST VECTORS 72

TABLE 4.3. WEIGHTS FOR THE 1-DISTANCE BETWEEN TWO TEST VECTORS. 72

TABLE 4.4. WEIGHTS FOR THE 0/1-DISTANCE BETWEEN TWO TEST VECTORS 73

TABLE 6.1. EFFECT OF X-WEIGHT 114

TABLE 6.2. EFFECT OF TYPE OF SORTING 115

TABLE 6.3. EFFECT OF BLOCK SIZE 117

TABLE 6.4. EFFECT OF TEST SET SIZE 119

TABLE 6.5. TIMING OF THE ENCODER 120

TABLE 6.6. TIMING OF THE HARDWARE DECODER 121

TABLE 6.7. STATISTICS ON BLOCK ENCODING (8X8 BLOCKS) 124

TABLE 6.8. STATISTICS ON BLOCK ENCODING (16X16 BLOCKS) 124

TABLE 6.9. STATISTICS ON BLOCK ENCODING (32X32 BLOCKS) 125

TABLE 6.10. COMPARISON WITH OTHER TECHNIQUES 126

LIST OF FIGURES

FIGURE 1.1. TEST DATA TRANSFER BETWEEN THE TESTER AND THE CIRCUIT UNDER TEST. 3

FIGURE 3.1. MERGING OF TWO TEST SEQUENCES 24

FIGURE 3.2. EXAMPLE OF VECTOR RESTORATION 26

FIGURE 3.3. RUN-LENGTH CODING 31

FIGURE 3.4. HUFFMAN CODING FOR THE EXAMPLE OF FIGURE 3.3 32

FIGURE 3.5. EXAMPLE OF A TEST SET TD (A) AND ITS TGC (B) 36

FIGURE 3.6. DECOMPRESSION USING VARIABLE-LENGTH RESEEDING 39

FIGURE 3.7. ARCHITECTURE OF A DFHTC CORE 42

FIGURE 3.8. EXAMPLE OF WEIGHTED TEST GENERATION 43

FIGURE 3.9. BW TRANSFORMATION 45

FIGURE 3.10. MODIFIED RUN-LENGTH CODING 47

FIGURE 3.11 . PREPARATION OF THE MATRIX TO BE ENCODED 49

FIGURE 3.12. NEXT SYMBOL 49

FIGURE 3.13. VARIABLE-TO-BLOCK RUN-LENGTH CODING 51

FIGURE 3.14. CYCLICAL SCAN CHAIN ARCHITECTURE 52

FIGURE 3.15. EXAMPLE OF MODIFIED STATISTICAL CODING 58

FIGURE 3.16. REPLACEMENT WORD 60

FIGURE 4.1. AN EXAMPLE CIRCUIT (S420.BENCH) AND A SUBSET OF ITS TEST VECTORS .. 69

FIGURE 4.2. TEST VECTORS ENCODING ALGORITHM 78
FIGURE 4.3. AN EXAMPLE OF A SUBSET OF A TEST SET 83
FIGURE 4.4. THE TEST SET AFTER 0-SORTING 83
FIGURE 4.5. THE TEST SET AFTER 1-SORTING 84
FIGURE 4.6. THE TEST SET AFTER 0/1-SORTING 84
FIGURE 5.1. TEST VECTOR DECODING ALGORITHM 92
FIGURE 5.2. DATA PATH OF THE DECODER 94
FIGURE 5.3. THE FSM OF THE DECODER 103
FIGURE 5.4. INTERFACE OF THE SOFTWARE DECODER 107
FIGURE 5.5. INTERFACE OF THE HARDWARE DECODER 108
FIGURE 6.1. EFFECT OF X-WEIGHT 114
FIGURE 6.2. EFFECT OF TYPE OF SORTING 116
FIGURE 6.3. EFFECT OF BLOCK SIZE 117
FIGURE 6.4. GREEDY VS. NEAR-OPTIMAL 118
FIGURE 6.5. COMPARISON WITH OTHER TECHNIQUES 127

THESIS ABSTRACT

Name: **Esam Ali Hasan Khan**

Title: **A Two-Dimensional Geometric-Shapes-Based Compression Scheme for Deterministic Testing of Systems-on-a-Chip**

Major Field: **Computer Engineering**

Date of Degree: **June 2001**

The increasing complexity of systems-on-a-chip with the accompanied increase in their test data size has made the need for test data reduction imperative. In this thesis, we introduce a novel and very efficient lossless compression technique for testing systems-on-a-chip based on two-dimensional geometric shapes. The technique is based on reordering test vectors to minimize the number of shapes needed to encode the test data. Then, the test set is partitioned into equal size blocks and each block is encoded independently. To test a chip, the encoded data is transferred from the tester to the chip-under-test and decoded there. The decoder can be implemented in software or in hardware. For software decoder, there must be an embedded processor where the decoding algorithm is executed. If this processor is not available, an additional hardware may be added to perform the decoding process. Both solutions need some amount of temporary memory to store a segment of decoded blocks. In this thesis, we have implemented the decoder in both software and hardware. The experimental results on ISCAS85 & ISCAS89 benchmark circuits showed the effectiveness of the proposed scheme in achieving very high compression ratio for most of the circuits. The achieved compression ratio is significantly higher than those obtained by most recently proposed schemes in the literature.

MASTER OF SCIENCE DEGREE
King Fahd University of Petroleum & Minerals
Dhahran – Saudi Arabia
June 2001

خلاصة الرسالة

الاسم: **حسام بن علي بن حسن خان**

عنوان الرسالة: **ضغط بيانات الاختبارات المفصلة للشرائح الحاوية لأنظمة متكاملة باستخدام الأشكال الهندسية ثنائية الأبعاد**

التخصص: **هندسة الحاسب الآلي**

تاريخ التخرج: **ربيع الأول 1422 هـ**

زيادة التعقيد في الشرائح الحاوية لأنظمة متكاملة والذي يتطلب زيادة حجم بيانات الاختبارات الخاصة بهذه الشرائح يجعل الحاجة إلى تقليل حجم هذه البيانات أمراً مهماً. في هذه الرسالة، نقدم طريقة جديدة وفعالة لضغط بيانات اختبارات الشرائح الحاوية لأنظمة متكاملة بدون فقد أي معلومات باستخدام الأشكال الهندسية الأساسية ثنائية الأبعاد. هذه الطريقة تعتمد على إعادة ترتيب متجهات الاختبار بهدف التقليل من الأشكال اللازمة لترميز البيانات. وبعد ترتيب المتجهات، تقسم البيانات إلى قوالب متساوية الحجم ويطبق خوارزم الترميز على القوالب كلاً على حدة. ولاختبار شريحة ما، تنقل البيانات مرمّزةً من جهاز الاختبار إلى داخل الشريحة حيث يتم فك رموز البيانات. يمكن فك رموز البيانات باستخدام برنامج أو جهاز خاص. لاستخدام برنامج لفك الرموز، يجب توفر معالج داخل الشريحة ليقوم بتنفيذ البرنامج. وإذا لم يكن هذا المعالج متوفراً، فيمكن تصميم جهاز خاص لفك الرموز. وفي كلتا الحالتين، يحتاج فك الرموز إلى كمية من الذاكرة المؤقتة لتخزين قطعة من القوالب التي تم فك رموزها. في هذه الرسالة، تم تنفيذ كلتا الطريقتين السابقتين لفك رموز البيانات. نتائج التجارب على الدوائر القياسية (ISCAS85 & ISCAS89) أظهرت فعالية الطريقة المقترحة في الحصول على نسبة ضغط عالية جداً. وتعتبر نسبة الضغط الناتجة عن الطريقة المقترحة هي الأفضل بين أحدث الطرق المقترحة سابقاً.

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران- المملكة العربية السعودية
ربيع الأول 1422 هـ

CHAPTER 1

INTRODUCTION

One of the primary tasks of any design process is the verification of system functionality against the desired specifications. Functional tests are often used at the various stages of the design process to verify correct system behavior. Once the system is manufactured, manufacturing tests are used to verify that the circuit implements the desired functionality. Manufacturing tests are based on fault models that can detect certain specific faults in the structure of the implemented circuit. The most widely used fault model is the single stuck-at fault model where each line in the circuit can be tested for being stuck at 0 or 1. This simple fault model detects a large number of the defects in the manufactured circuit. In this thesis, the test vectors generated are based on the single stuck-at fault model.

Testing of a chip requires generating test vectors that can detect all the faults in the structure of the circuit, if possible, according to the used fault model. Such test vectors are often generated by automatic test pattern generation (ATPG) programs,

which can successfully generate test vectors with a high fault coverage (i.e. can detect a large number of the modeled faults). Then, the test vectors are applied to the circuit in sequence and the circuit responses are observed. If the observed responses are not equal to the expected correct responses of the circuit, then the circuit is marked as defective; otherwise it is said that the circuit passes the test without failure. For defective circuits, another type of test set can then be applied to diagnose the circuit and try to identify the locations of the defects. This can help in improving the manufacturing process to increase the yield, which is the ratio between the non-defective chips to the total number of manufactured chips. In this thesis, we focus on detecting test sets rather than diagnostic ones.

To test a certain chip, the entire set of test vectors, for all the cores and components inside the chip, has to be stored in the tester memory. Then, during testing the test data must be transferred to the chip under test and test responses collected from the chip to the tester (as illustrated in Figure 1.1).

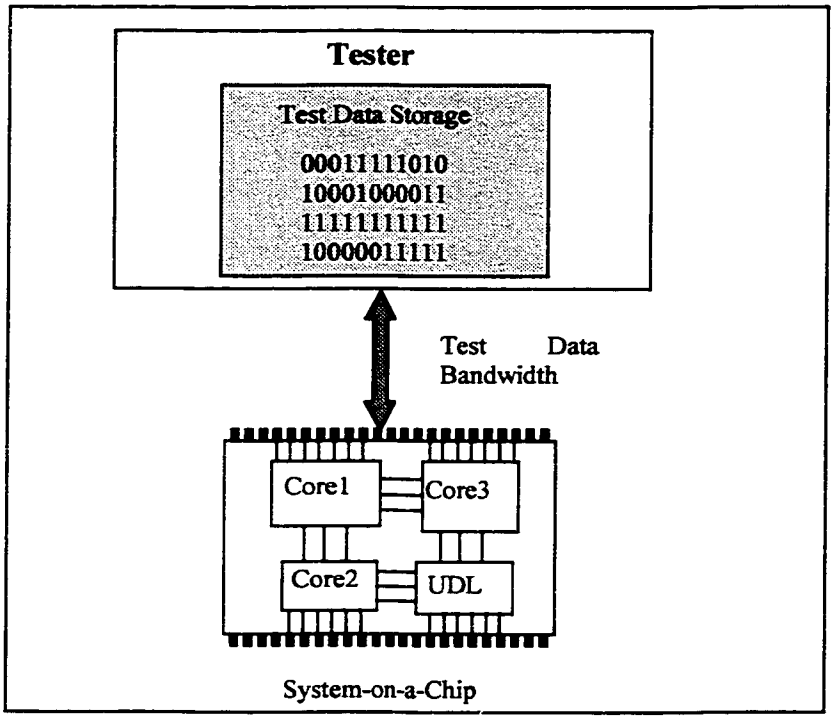


Figure 1.1. Test data transfer between the tester and the circuit under test

With today's technology, it is possible to build complete systems containing millions of transistors on a single chip. Systems-on-a-chip (SOC) are comprised of a collection of pre-designed and pre-verified cores and user defined logic (UDL). As the complexity of systems-on-a-chip continues to increase, the difficulty and cost of testing such chips is increasing rapidly [10], [39].

One of the challenges in testing SOC is dealing with the large size of test data that must be stored in the tester and transferred between the tester and the chip. The amount of time required to test a chip depends on the size of test data that has to be transferred from the tester to the chip and the channel capacity. The cost of automatic test equipment (ATE) increases significantly with the increase in their speed, channel capacity, and memory. As testers have limited speed, channel bandwidth, and memory, the need for test data reduction becomes imperative.

Test data reduction has many advantages. The most important one is the reduction of testing time, which in turn reduces the time-to-market. Testing time includes time to transfer test data from the Automatic Test Equipment (ATE) to the Circuit Under Test (CUT) and test application time. In addition, reducing test data volume saves memory requirement, which may be very expensive, especially if the test is to be stored inside a chip [9].

The problem of test data reduction has been addressed in different solutions. These solutions can be classified into two main categories, **test set compaction** and **test vector compression/decompression**. In test set compaction, the goal is to minimize the number of test vectors in the test set while maintaining the same fault coverage;

i.e. detecting the same number faults. There are two approaches to compact test sets. The first is *static* compaction where the compaction is performed after generating the test set. The second is *dynamic* compaction in which test set is minimized while generating the vectors [17]. In test vector compression, test vectors are encoded in a different format and transferred to the Automatic Test Equipment (ATE) and then to the chip under test in this compressed format. Then, a decoding technique on the chip is used to reconstruct the original test data. This approach usually exploits the fact that test vectors are highly correlated to each other [38]. To achieve better test data reduction, more than one of the above techniques can be applied. For example, it is possible to generate a test set using a dynamic compaction technique. Then, a static compaction technique is used to further reduce the test vectors. After that, a compression technique is applied to encode the test data.

Most of the compression techniques proposed in the literature for deterministic testing take advantage of the high correlation between test vectors. However, most of these techniques use a one-dimensional approach, where vectors are encoded serially. In addition, most of these techniques are based on variations of run-length coding and statistical coding. In this Thesis, we propose a novel technique to

compress/decompress deterministic test data based on two-dimensional geometric shapes.

The proposed technique is based on reordering the test vectors to take advantage of the high correlation between them. The goal of this reordering is to generate minimal number of primitive geometric shapes needed to encode the test data. Then, the test data is partitioned into blocks and then each block is encoded individually. There are three possibilities to encode a block. The first is to encode the block as filled by either 0's or 1's. The second is to encode the block using geometric shapes that cover the 0's or the 1's depending on which gives minimum cost. The third is to store the actual test data. This is done when the cost of encoding the block is more than the cost of the real data.

The encoder is implemented in software because it is executed offline. The decoder can be implemented in software if the SOC has an embedded processor to execute it. However, if there is no such processor to run the software decoder on the chip, then a hardware implementation of the decoder is needed. In this thesis, we have implemented the encoder and the software decoder in C++ code. The hardware decoder has been designed and implemented using VHDL. The encoder and both

implementations of the decoder are very fast. Compared to the results of the most recent compression techniques published in the literature, we achieved very high compression ratio. The only limitation of our proposed technique is the need of the decoder for some amount of memory to store a segment of decoded blocks before applying its vectors to the circuit under test. However, the required memory is often available in systems-on-a-chip, which can be exploited.

The organization of this thesis is as follows. In Chapter 2, some definitions and preliminaries are given. A literature review of the solutions proposed for reducing test data is presented in Chapter 3. In Chapter 4, details of our proposed encoding scheme are described. The decoding process is explained in Chapter 5, including the software and the hardware decoders. In Chapter 6, experimental results are discussed to show the effect of some factors of the sorting and partitioning steps on the compression ratio and to show the effectiveness of our proposed technique compared to the most recent techniques published in the literature. Finally, we conclude and indicate the future directions in Chapter 7.

CHAPTER 2

BACKGROUND MATERIAL

In this chapter, we give some definitions and preliminaries required to understand the following chapters.

- A **test vector** is a string of *n* logical values ("0", "1", and don't care "x") that are applied to the *n* corresponding Primary Inputs (PIs) of a circuit at the same time frame to detect one or more faults [32].

- A **test sequence** is a series of test vectors that are applied in order to detect faults in sequential circuits [32].

- A **test cube** is a **partially specified** test vector, where only the necessary PIs for detecting the targeted fault(s) are assigned binary values (0 or 1). The other PIs are left as x's. If all PIs are assigned binary values, then the vector is **fully**

specified [20]. Two test cubes are **compatible** if each PI is assigned the same value (0 or 1) in both of them or it has an x in at least one of the two test cubes [32].

- A **test set** is a collection of test vectors that are applied to the Circuit Under Test (CUT) to achieve a certain **fault coverage**, which is defined as ratio between the number of detected faults to the total number of faults. For sequential circuits, the order of test vectors must be preserved. However, this is not necessary for combinational circuits. A **test** can be used for a test set, a test vector or a test sequence [32].

- An **essential fault** of a test vector t in a test set T is the fault that is detected only by t in T. In this case, t is an **essential vector** in T. A **redundant vector** is a vector in a test set that does not detect any essential fault. In other words, it is the vector whose all faults can be detected by other vectors in the set [24], [15].

- Two faults are **compatible** if they can be detected by a single vector. In contrast, two faults are **incompatible** (or **independent**) if they cannot be detected by a single vector. An **independent fault set** is a set of faults in which all faults are pairwise incompatible. A **Maximal Independent Fault Set (MIFS)** is an independent fault set with maximal cardinality.

- A test is generated using an **Automatic Test Pattern Generator (ATPG)**. If the test is generated such that each vector targets certain fault(s), the test is **deterministic**. If the vectors are generated randomly, then the test is **random**. **Pseudo-random** tests are generated by deterministic algorithms but have statistical properties as random sets [1]. **Fault simulation** is the application of a test to the CUT and determining faults detected by that test. An **ATE (Automatic Test Equipment)** is a tester that stores the test data and applies it to the CUT.

- The reduction of the number of test vectors in a test set is called **test compaction**. There are two main categories of compaction, **static compaction** and **dynamic compaction**. In static compaction, the number of test vectors is

reduced after they have been generated. Examples of static compaction algorithms include *reverse order faults simulation* [35], *forced pair merging* [11], *N_by_M* [24] and *redundant vector elimination (RVE)* [15]. In dynamic compaction, number of vectors is minimized during the ATPG process. Examples of dynamic compaction algorithms include *COMACTEST* [30], *dynamic compaction using genetic optimization* [29], and *bottleneck removal* [7].

- In **test data compression**, a test vector is encoded in such a way that reduces the number of bits needed to store it. For test data compression, it is essential that the compression is **lossless**. That is, no data is lost when a vector is decompressed. In **lossy** compression, some of the original data can be lost, which is not allowed in testing.

- **Compression ratio** is the percentage of data reduced after compression. There are a number of ways to compute it. In this thesis, the compression ratio is

computed according to the following equation: {*compression ratio* = *(Original data – reduced data) / Original data* x *100*}.

- To simplify testing sequential circuits, some of the memory elements – or flip-flops (FFs) – inside the circuit are modified to be **controllable** (i.e. their value can be controlled by some PIs) and/or **observable** (i.e. their value can be observed through some Primary Outputs (POs)). This is called **scan design**. A **scan register** or a **scan chain** is a set of such modified flip flops that have a **serial in** pin that is used to shift the desired values into the register (**scan in**) and a **serial out** pin to shift the content of the register out (**scan out**). In **full scan** design, all flip-flops in the circuit are included in the scan chain(s). In **partial scan** design, only a partial set of flip-flops is included. In **non-scan** design, no flip-flop is made scannable and this is the original sequential circuit [32], [1].

- A **system-on-a-chip (SOC)** is an integrated circuit (IC) constructed based on pre-designed and pre-verified **cores** and user defined logic (UDL). Each core has a specific function in the SOC. Examples of cores are CPUs, DSPs, MPEG,

and JBEG cores. For **Intellectual property** cores, no information is given about the internal implementation of the core by the vendor. Only the test set is provided [20].

- **Built-in Self-Test (BIST)** is a type of Design For Testability (DFT), where the circuit is designed to have a mechanism for testing itself. One of the most popular means for designing BIST is the **Linear Feedback Shift Register (LFSR)**, which is a combination of flip-flops and XOR gates that act as a special pseudo-random ATPG.

CHAPTER 3

LITERATURE REVIEW

3.1. Compaction Techniques

Test set compaction is the process of reducing the number of test vectors in a test set while maintaining the same fault coverage. Finding the smallest set is proven to be an NP-hard problem [17]. Therefore, some heuristics can be used in order to find a reasonable solution.

There are two main categories of test compaction techniques: static compaction and dynamic compaction. In static compaction, the test set is reduced after it has been generated. On the other hand, it is reduced during the generation process in dynamic compaction. In this section, we discuss some of the techniques proposed for each class. We start by discussing those techniques proposed for combinational circuits. Then, we discuss the techniques proposed for sequential circuits.

3.1.1. Compaction Techniques for Combinational Circuits

3.1.1.1. Static Compaction

One of the simplest techniques for static compaction is called **reverse order fault simulation** [35]. In this technique, the order of the vectors is reversed and the reversed vectors are fault simulated. It has been shown experimentally that usually the desired fault coverage is achieved with a subset of the original set of test vectors. The reason for this is that the vectors generated in the last stages usually target hard-to-detect faults. Therefore, they can also detect some of the easy-to detect faults that are addressed individually by other vectors. This leads to the reduction of test vectors.

Another approach is called **forced-pair-merging** [11], in which test vectors are relaxed to have more don't care (x's) while maintaining the same fault coverage. This may lead to more compatible test vectors that can be merged and hence the set is reduced. However, finding the inputs that can be don't care is time consuming. Furthermore, the best merging solution is by itself an NP-hard problem.

In [17], the merging process has been solved as a *Set Cover problem*. In a set cover problem, the goal is to find the smallest collection of sets that cover a given set of elements. Here, the elements are the faults to be covered and the set to be minimized is the number of vectors. This problem has been formulated in [17] as an integer programming problem and solved using *Linear programming (LP) Relaxation*. In LP relaxation, the requirement that the variable must be integers is relaxed (or removed). So, if the variables are binary (0 or 1) in the integer problem, they are converted to real numbers between 0 and 1. This LP relaxation has the property that its solution is a lower bound on the value of the optimal integer solution. The main objective of using LP relaxation for merging test vectors is to find the *maximum merging*, i.e. maximize the number of merged vectors.

Another approach for static compaction is the **two_by_one** scheme [24]. In this approach, two vectors in a test set T are replaced by a new vector. The approach depends on the following concepts:

- Finding the essential faults of each vector in T. The definition of essential faults is extended here to include essential faults of two vectors t_1 and t_2. These are the faults that are detected by either t_1 or t_2 but cannot be detected by any other vector in T.

- Finding a maximal independent fault set F. This is used to make sure that for two vectors to be replaced by one vector, their essential faults are not independent. If some of these faults are independent, it is not possible to find a replacement vector. Finding F will save search time.

The algorithm finds two vectors such that their essential faults are not independent and generates a new vector to replace them. After all vectors are tried, the new vectors are added to the set and fault simulation will remove the redundant vectors. The algorithm can be extended to be an **N_by_M** in which M new vectors will replace N vectors of T. However, this seems to be more complex and time consuming.

In [15], a new algorithm called **Essential Fault Reduction (EFR)** is proposed to enhance the two_by_one and the N_by_M algorithms. It adds some new techniques that can achieve better results in less computation time (especially than the N_by_M).

3.1.1.2. Dynamic Compaction

One of the earliest and most popular dynamic compaction algorithms for combinational circuits is **COMPACTEST** proposed in [30]. It is based on finding maximal independent fault sets for *fanout-free regions* (FFRs) of the CUT. The procedure can be summarized as follows:

1) For each FFR, find a maximal independent fault set (MIFS).
2) Build an ordered fault list for the CUT. The faults of the largest MIFS are placed at the top of the list, followed by the second largest, and so on. Faults that are not included in any MIFS are listed then.
3) Select a fault from the top of the ordered fault list as a *primary target fault f*. Find a test vector to test this fault.
4) *Maximally compact* this vector by finding the maximum number of PIs that can be unspecified (i.e. having x value). This is done as follows:
   (i) Complement a specified PI, *p*.
   (ii) Fault simulate the vector.
   (iii) If *f* is still detected, mark *p*.
   (iv) Complement *p* again.
   (v) Repeat the steps (i) to (iv) until all PIs are tried.
   (vi) Unspecify the marked PIs.

This step is done to maximize the possibilities of detecting other faults by the same vector. It should be noted that this new vector might not detect f if some of the unspecified PIs are assigned certain values. However, this is a rare case as shown experimentally.

5) Select another fault f_s as a *secondary target fault* and try to detect it by the same vector. Do not change any specified PIs. However, some of the unspecified PIs can be now specified. If f_s cannot be tested by this vector, unspecify all PIs specified for f_s.
6) Maximally compact the part of the vector specified for f_s.
7) Repeat steps (5) and (6) until all PIs are specified or all faults in the fault list are tried as secondary target faults.
8) If there exist some unspecified PIs, specify them randomly.
9) Fault simulate the vector and remove all detected faults from the fault list.
10) Repeat steps (3) to (9) until all faults are detected.

Modifications to COMPACTEST have been proposed in [24]. The first modification is in the ordering of the fault list. Here, after each generation of a vector, all detected faults are removed from the fault list. Then, the fault list is reordered based on the same criterion (i.e. the largest MIFS after the removal of detected faults). This reordering allows the selection of target faults from the independent fault sets that

have the largest number of undetected faults. The second modification is called **double detection**, which is based on the generation of redundant test vectors. After generating a test vector, the unspecified PIs are used to detect some of the already detected faults. In this way, some of the previously generated vectors may become redundant. Then, fault simulation is performed and redundant vectors are identified. These vectors are removed from the test set.

Another algorithm that performs the same function as the double detection algorithm is the **Redundant Vector Elimination (RVE)** algorithm proposed in [15]. This algorithm keeps track of the faults detected by each vector, the number of times each fault is detected and the number of essential faults of each vector. After generating each vector, the algorithm fault simulates all the faults in the fault list and updates the three parameters listed above. A test vector may become redundant if its number of essential faults becomes zero. Then, these redundant vectors are removed. This algorithm, along with the EFR algorithm (described in Section 3.1.1.1) has been incorporated in an ATPG called **MinTest**, which is a dynamic compaction ATPG for stuck-at faults. In [14], this ATPG is extended to include other fault models such as transition and stuck-open fault models that require two-pattern test sets.

Another dynamic compaction algorithm that is based on building MIFSs is proposed in [36]. The algorithm first finds MIFSs for all faults, and based on these sets, other sets are constructed that include *compatible* faults. The goal is to find the minimum number of compatible sets because faults in each set will be detected by one test vector. However, finding the best solution for MIFSs and compatible sets is NP-hard. So, some heuristics must be used. Another contribution in [36] is that in finding a test vector, multiple target faults are chosen instead of only one.

Since finding a minimal test set is an NP-hard problem, iterative algorithms can be used. One of the most popular iterative heuristics is **Genetic Algorithm** [18]. It has been applied to the problem of finding a minimal test set in [26]. This technique achieved complete fault coverage for the ISCAS85 benchmark circuits. In addition, it generated complete *n-detection* test sets, which are the test sets in which each target fault is detected n times. These *n-detection* test sets have desirable properties in detecting unmodeled faults. However, this technique generates larger test sets than those generated by deterministic compaction techniques.

In [29], another procedure was proposed to reduce the size of the test sets generated by the technique proposed in [26]. In each iteration of the proposed scheme, one test

vector is added to the compacted test set *COMP_T* from the best set generated in any previous iteration. This set is called *BEST_T*. The procedure can be summarized as follows:

1) Set $COMP\_T = \phi$. Set $BEST\_T = \phi$. F is the target faults.
2) For N iterations:
   (i) Let $T = COMP\_T$.
   (ii) Use the procedure of [26] to generate a test set T' for F and add it to T.
   (iii) If T is better than *BEST_T*, let $BEST\_T = T$.
3) If $COMP\_T = BEST\_T$, stop.
4) Select the best vector t in *BEST_T* that is not in *COMP_T* and add it to *COMP_T*.
5) Drop all faults in F detected by t.
6) Go to step (2).

There are two measures in the above procedure:

(1) A test set is better than another. This is measured by the first satisfied of the following three criteria:
   (a) Number of faults detected by the set that are not yet detected.
   (b) Set size (number of test vectors)
   (c) If there is a vector that detects more yet-undetected faults than any vector in the other set.

(2) The best test vector in a test set. This vector is the vector that detects the maximum number of yet-undetected faults.

3.1.2. Compaction Techniques for Sequential Circuits

The main criterion that has to be met in testing sequential circuits is to preserve the order in which a test sequence is applied to the CUT [32]. This complicates the issue of compaction (and compression) of test data for sequential circuits than for combinational circuits because any technique based on reordering of test vectors cannot be applied here. In this section, we review briefly some compaction techniques proposed to solve this problem. We start with static compaction and then discuss dynamic compaction.

3.1.2.1. Static Compaction

One important observation in testing sequential circuits is that overlapping of test sequences is allowed as long as the order of each sequence is preserved. This fact can be exploited to compact test sequences if they are *self-initializing*. This means that each sequence is independent of the previous applied sequences. **TESEUS** is an

example of ATPGs that generate such sequences [16]. Such sequences can be merged if they are **compatible**. Compatibility of test sequences can be defined in two ways.

(i) The first, which is simple and straightforward, is that two test sequences S_1 and S_2 are compatible if the i^{th} vector of S_1 is compatible with the i^{th} vector of S_2, for $i = 1$ to smaller length of S_1 and S_2.

(ii) The second, which is more accurate, is that two sequences may be compatible if the start of one of them is *skewed* from the start of the other one.

Figure 3.1 shows the possibilities of skews of two sequences S_1 and S_2 of length l_1 and l_2, respectively. Figure 3.1(a) shows the two sequences with no merging. Figure 3.1 (b) shows the first definition. Figure 3.1 (c), (d) and (e) show different start and end points.

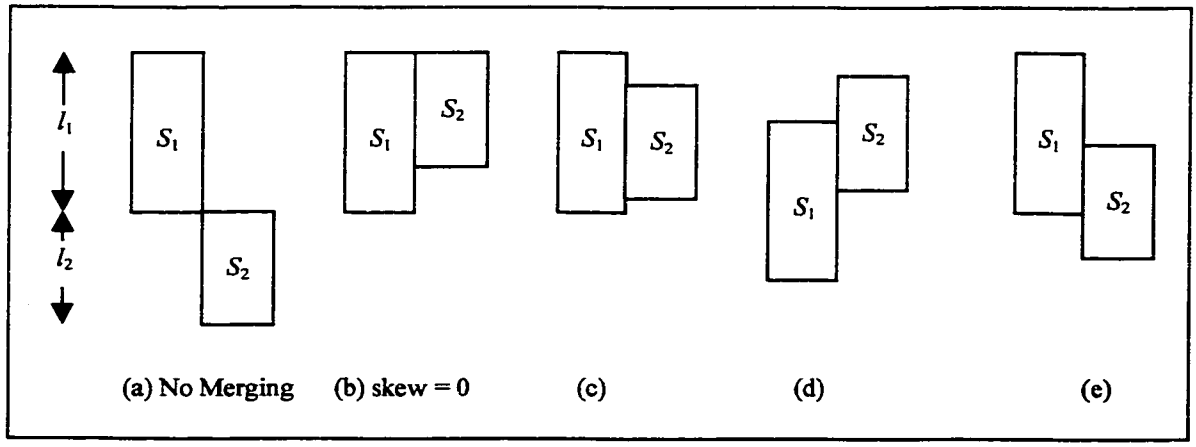


Figure 3.1. Merging of two test sequences.

These compatibility definitions can be used to merge test sequences and hence reduce the test set size. In [32], algorithms are given for merging compatible sequences with and without skews. Another idea proposed in [32] that enhances the compatibility of sequences is **stretching** of sequences. A sequence is stretched if some of its vectors are repeated (one or more times) as long as each vector is repeated in its order. For example, a sequence 1100 1000 1110 can be stretched as 1100 1000 1000 1110. It was observed that, in general, stretching a test sequence will not affect the response of the CUT to the sequence as it does not change the state of the circuit.

Another approach for static compaction is **vector restoration** [27], [4]. In this technique, vectors are restored for each fault starting from the hardest-to-detect fault to the easiest. In this way, some faults can be detected by sequences of other faults. This technique is similar in concept to the reverse order fault simulation explained in Section 3.1.1.1. We summarize the technique in the following with the illustration by an example of Figure 3.2.

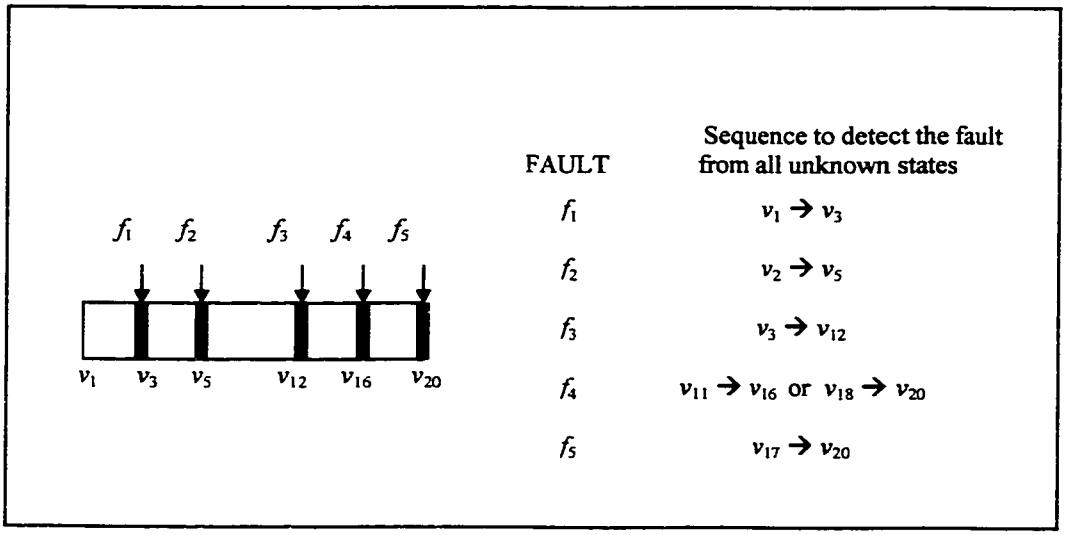


Figure 3.2. Example of vector restoration

The first step is fault simulation to determine at which vector each fault is detected (this is called *detection time*). Starting from the last detected fault (f_5 in the figure), vectors are restored one at a time until the fault is detected by a sequence of vectors starting from all unknown states. For instance, v_{20} is simulated first and since it does not detect f_5, v_{19} is added to the sequence and now the sequence (v_{19}, v_{20}) is simulated, and so on until the sequence (v_{17}, ... , v_{20}) is restored. Any fault that is detected by this sequence is removed from the current fault list. For example, f_4 is removed from the list since it is detected by (v_{18}, ... , v_{20}).

If all remaining faults have detection time less than the start of the last restored sequence, the detected faults by this sequence form a segment. In our example, f_4 and

f_5 form a segment since all remaining faults have less detection time than 17 (3, 5 and 12 for f_1, f_2 and f_3, respectively). Then, start from the last yet-undetected fault and repeat the process until another segment is formed. For the example given, when vectors are restored for f_3, it is detected by (v_3, ..., v_{12}) but f_2 has detection time $5 > 3$; so the restoration continues for f_2. And this continues for f_1 as well. Therefore, the second segment contains f_1, f_2, and f_3. By applying this method to the given example, a reduction of 4 vectors was achieved.

3.1.2.2. Dynamic Compaction

In this section, we discuss two dynamic compaction approaches for sequential circuits. The first approach is based on **omission and insertion** of test sequences [28]. This technique uses static compaction algorithms to implement dynamic compaction. These algorithms are omission and insertion of test sequences. In the omission operation, some test vectors are omitted from a test sequence T in order to introduce a new subsequence T' in T. For example, if $T = (0,1,0,0,1,1,0,0)$, we can introduce $T' = (1,1,1)$ by omitting the (0,0) in the third and forth positions of T and we get $T = (0,1,1,1,0,0)$. In the insertion operation, a new subsequence T' is inserted in T. This can be done in two ways: either by finding a prefix and a suffix of T' in T and inserting the missing vectors of T' between them, or by finding a suffix only and

inserting the missing vectors before it. As an example of the first way, if T = (00,10,11,00,00,01,10) and T' = (11,00,11,00), the prefix is (11,00), which is in the 3rd and 4th positions of T, and the suffix is (00), which is the 5th position. The missing part of T' is (11), so we can insert after the 4th position to get T = (00,10,11,00,11,00,01,10) and T' is underlined. For the second way, consider T = (0,1,1,0,1,0) and T' = (0, 0, 0, 1, 0). A suffix of T' is (1, 0), which exists in the 3rd and 4th positions of T. We can introduce T' in T by inserting (0,0,0) before the 4th position and T becomes (0,1,0,0,0,1,0,1,0).

Both the omission and insertion operations modify the test sequences and they have two effects: (i) some faults that were detected may no longer be detected, and (ii) some undetected faults may now be detected. Experiments show that a large number of omission and insertion operations does not reduce fault coverage but results in reduced test set sizes.

The second approach we discuss here is based on using genetic algorithms (GA) in the test generation process [33], [34]. The procedure can be summarized as follows:

(1) Use some ATPG to generate a test sequence for a set of target faults.

(2) Fault simulate this sequence and remove any unnecessary vector.

(3) Invoke the GA algorithm to evolve new generations of the obtained sequence.

(4) Fault simulate the best sequence obtained in any generation and remove the faults it detects.

(5) Go to step (1) for new set of target faults.

Two algorithms have been proposed using this technique, **squeeze** [34] and **GA-COMPACT** [33]. The major difference between the two is that in GA-COMPACT, the specified bits of the sequence generated by the ATPG must be preserved, while they can be changed in squeeze. Experiments show that squeeze achieves better results.

3.2. Compression Techniques

In test data compression, the goal is to minimize the number of bits needed to represent the test data. The advantage of this technique is the reduction of the time required to transfer data from an external workstation to the ATE and from the ATE to the chip under test. Due to the limited speed, channel capacity, and memory of ATE, the reduction of test data may save hours [37], [21].

Many techniques have been proposed to achieve minimal test data. One of the earliest is BIST, where the circuit is designed to have the capability of testing itself. The BIST-based compression techniques require pseudo-random test generation to generate the test vectors. Another approach is the deterministic-test-set approach, where a compression technique is applied to the test data regardless of the internal architecture of the circuit. In this section, we review some of the techniques proposed under these two approaches.

In the following section, the basic compression techniques (or algorithms) that have been used either in testing or other fields are discussed briefly. Those used in some of the techniques addressed in this thesis are discussed in more detail. In Section 3.2.2, compression techniques based on pseudo-random generators are discussed

briefly. Then, we discuss the techniques proposed for compressing deterministic test data in Section 3.2.3.

3.2.1. Basic Compression Schemes

One of the simplest compression schemes is **run length coding.** A sequence of symbols can be encoded using two elements for each *run*, which is a consecutive sequence of equal symbols. The two elements are the repeating symbol and the number of times it appears in the run. Figure 3.3 shows an example of a sequence of letters with the corresponding run-length code [37].

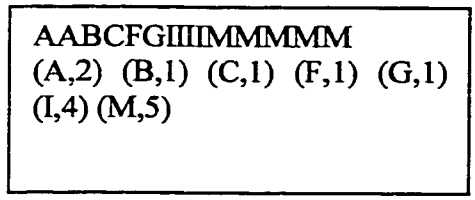


Figure 3.3. Run-Length coding

Another more sophisticated and more efficient scheme is **Huffman coding** which builds a binary tree based on the probability of the occurrence of symbols. Leaves in the tree correspond to the symbols. So, a symbol can be encoded by traversing the

tree from the root to the corresponding node by encoding any left branch with "0" and any right branch with "1". Figure 3.4 shows the Huffman tree and the Huffman code of the example of Figure 3.3. First, the two symbols with the smallest number of occurrences are picked to form the first two leaves. In the example, these can be any two of the four symbols B, C, F and G. The root of these two leaves has a value equals the sum of their occurrences. This root is considered as a symbol in the next phase. Then, another two symbols are selected and the procedure continues until all symbols are selected. An important feature of Huffman coding is that it is *prefix-free*; that is, no codeword is a prefix of another codeword.

Huffman coding belongs to a class of coding schemes called **statistical coding** where codewords of variable length are used to encode fixed-length blocks of data [37], [22].

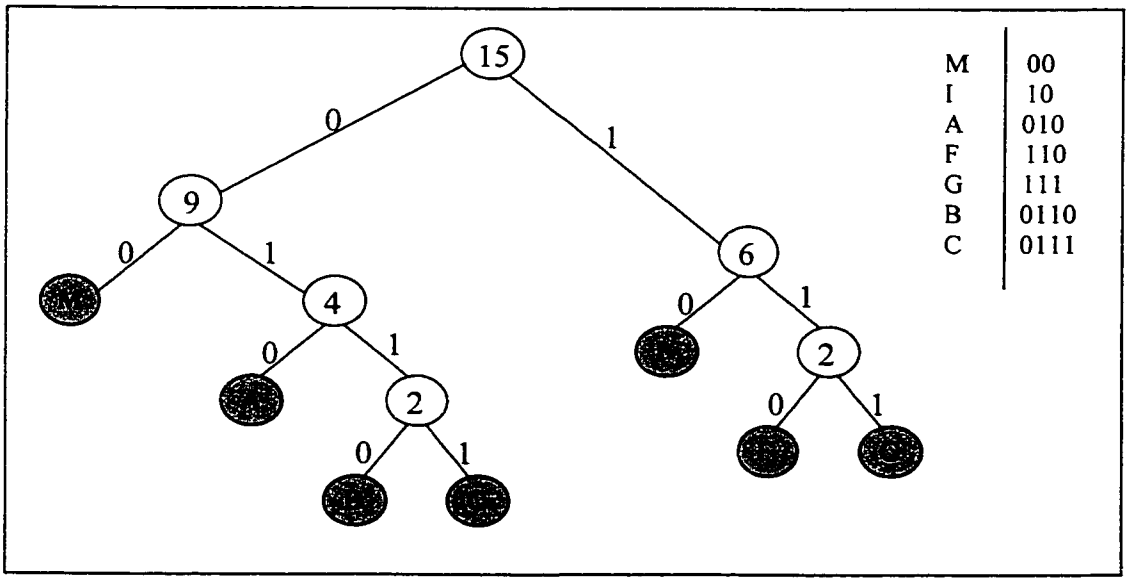


Figure 3.4. Huffman coding for the example of Figure 3.3

Arithmetic coding is another scheme that assigns a unique identifier to each block of symbols. This identifier is used to restore the original data [13].

Lempel-Ziv (LZ) and its variations (**Lempel-Ziv-Welsh (LZW) and Lempel-Ziv-Storer-Szymanski (LZSS)**) are used to encode symbols by constructing a dictionary. They differ in minor aspects [13].

Some of these techniques can be used in test data compression. However, any scheme used in test data compression must have two characteristics: lossless compression and simplicity in decompression. The first feature must be met to avoid losing any bit, which in turn may reduce the fault coverage. The second characteristic is important to reduce test application time and hardware overhead [37].

3.2.2. PRG- and BIST-Based Compression Techniques

BIST is a Design For Testability (DFT) technique. It has some characteristics such as (i) the ability for a system to test itself in-speed, which reduces the test application time, (ii) cheaper ATE, and (iii) the ability for testing systems on-line [6]. However, it has many limitations. We can summarize them in the following:

(a) Difficulty to achieve high fault coverage because it depends only on pseudo-random generators (PRG). Some faults are hard-to-detect using random vectors. This can be solved by adding some test points to the circuit under test (CUT) but this usually degrades performance [6], [23].

(b) Long test lengths are required. This may add to the time that the chip sits in the tester socket.

(c) The complexity of designing a BIST testing tool, especially when other cores on the chip are tested externally [22], [23].

Some techniques have been developed to overcome the first problem by combining deterministic testing with BIST or other DFT methods that exploit PRG. In the following subsections, some of these techniques are discussed briefly. These techniques are somehow outside the scope of this survey, but we address them because they have some kind of determinism for test data compression.

3.2.2.1. Test Width Compression

In test width compression, a Test Generation Circuit (TGC) produces a compressed vector with width w that is less than the original test width N. Then, a decoder circuit

is responsible for restoring the original set. This technique can be used for both combinational and sequential circuits.

The deterministic test set T_D is generated first using conventional ATPG. Then a TGC is used to compress it in order to reduce both timing and storage requirements. Different TGC implementations have been proposed. One uses a counter that generates the addresses of the ROM in which T_D is stored. Another uses an FSM to generate T_D without the need to store it. The most effective and widely used method is to combine LFSR with ROM. T_D is stored in a compressed form then the LFSR is used to decode it.

Width compression can be achieved in different ways. Here, we explain the method proposed in [6]. The technique is based on the following assumptions:

1) T_D is a precomputed, partially specified set and stored as a matrix of mxN size, where m is the number of vectors and N is the vector width.
2) Full scan is employed for sequential circuits.
3) LFSR (or another counter-like circuit) is available for decoding.

The following definitions are important to understand the technique:

- In a test matrix T_D, two columns a and b are **compatible** if for every row i, $a_i = b_i$ or one of them is don't care (x). They are **inversely compatible** if for every row i, $a_i \neq b_i$ or one of them is x. For example, columns 3, 4 and 5 in Figure 3.5 (a) are compatible. Columns 1 and 2 are inversely compatible.
- Two columns a and b are **d-compatible** if there is no row in which both of them equal 1. In Figure 3.5 (a), columns 1 and 2 are d-compatible and also 1 and 4 are d-compatible.
- A **maximal d-compatible (MDC)** class is the set of all columns in T_D that are pairwise d-compatible. In an MDC, there is at most one 1 in each row. Therefore, it is possible to encode the row by specifying the position of that 1 (1 to n, where n is the number of columns in the MDC or 0 if no 1 exists). So, for each row, the number of bits needed for encoding = $\lceil \log_2(n+1) \rceil$.

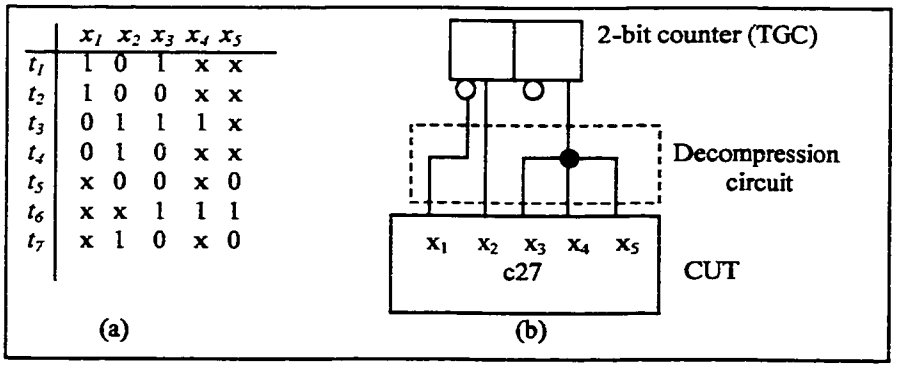


Figure 3.5. Example of a test set TD (a) and its TGC (b)

To encode a test set T_D, a set M of k MDCs $\{C_1, C_2, \ldots, C_k\}$ is to be found such that:

(i) each column in T_D must appear in at least one MDC, and

(ii) the width of all compressed vectors w is minimized, where

$$w = \sum_{i=1}^{k} \lceil \log_2(n_i + 1) \rceil .$$

The set M is called *an optimal MDC cover*. However, finding the best M is an NP-complete problem.

The procedure of encoding T_D can be summarized as follows:

1) Reduce T_D by merging compatible and inversely compatible columns. This is done in hardware by assigning compatible columns to the same output and inversely compatible columns to an inverter of the same output of the TGC.
2) Apply column complementation to reduce the number of 1's. If a column is complemented, the corresponding output of the TGC must be inverted.
3) Apply row complementation to reduce the number of 1's. Here, a redundant column is added to indicate whether a row is complemented or not. By XORing this column with the rows, the original set can be restored. However, careful computation must be done to compromise between compression and added overhead.
4) Compute a near-optimal MDC cover using some heuristic.
5) Encode rows of each MDC.

3.2.2.2. Variable-Length Reseeding

This technique is used with deterministic test cubes (partially specified test vectors). A test cube is constructed in a Multiple Polynomial LFSR (MP-LFSR) using a seed computed based on the specified bits of the test cube. The seed can be of variable length since specified bits vary from a test cube to another. The technique has the following characteristics:

1) On average, a test cube with s specified bits can be encoded with a seed of s-bit length. The seed size is usually less than the LFSR size.
2) The decompression hardware can be implemented using scan flip-flops and/or RPG flip-flops because it is loaded for each test pattern; so, there is no need to preserve the content of the MP-LFSR. Therefore, additional flip-flops are not needed.

Variable-length reseeding uses a k-bit LFSR to generate the test patterns. The following steps summarize the procedure:

(i) Reset the LFSR.

(ii) Switch to the shift mode and load the seed to the LFSR.

(iii) Apply enough clock cycles to shift the seed into the scan register, which will hold the desired pattern.

(iv) Switch to the functional mode and apply the pattern to the CUT.

(v) Shift out the response to the test response analyzer.

Figure 3.6 shows an example of decompressing a 9-bit test pattern using a 2-bit seed. The decompression hardware consists of a 5-bit LFSR formed using an existing 3-bit LFSR and 2 flip-flops of the scan chain.

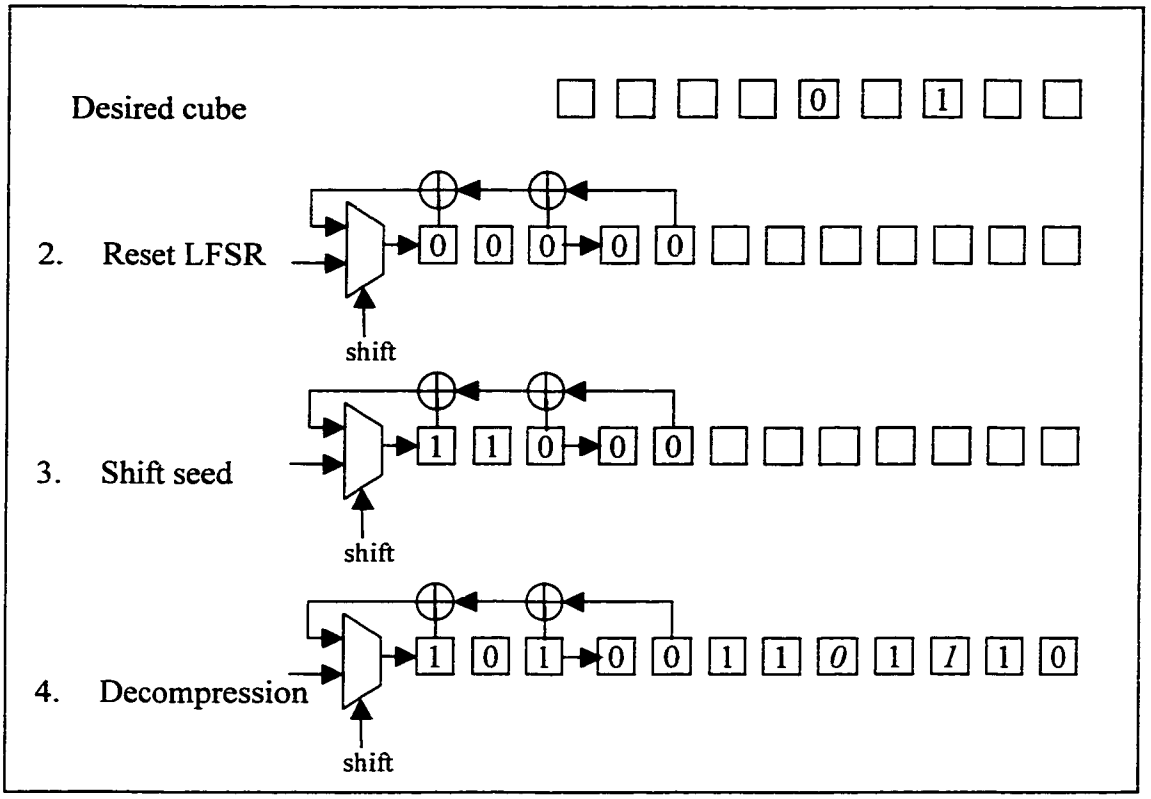


Figure 3.6. Decompression using variable-length reseeding

Since seeds are of variable lengths, each one is assigned a bit to indicate when the current length is to be increased. The increase is done with a constant *d*, with the addition of extra 0's whenever the current length + *d* is greater than the new length. The first seed to be stored is the shortest one.

In [38] and [31], three techniques are proposed for the implementation of the decompressor. The first one uses a two-dimensional hardware decompressor for multiple-scan chain designs. It exploits the existing scan flip-flops and the flip-flops of the PRGs with the addition of few extra logic (XOR and AND gates). The goal of this implementation is to allow the decompression of large number of specified bits while minimizing the area overhead.

The second implementation uses the embedded processor available in some core-designs to load and execute the decompression algorithm. In this way, no additional hardware is needed and hence no area overhead. Here, a program (or a microcode) is executed in the embedded processor to read data from external memory to the local register file and decompress the data to the scan chains of the CUT. To reduce test application time, the number of instructions needed to decompress the data has to be minimized.

The third alternative is used with designs that include boundary scan chains. This is useful when CUTs are mounted in a board during testing.

3.2.2.3. Design For High Test Compression (DFHTC)

The basic idea of this technique proposed in [23] is to design a core that is identical to conventional cores but can be tested with a much smaller number of test vectors. It combines PRG with deterministic testing. However, it is different from BIST in three main concepts:

(a) It is compatible to all other cores that use scan chains. So, there is no need for additional hardware for each core.

(b) BIST needs hardware for scheduling tests for different cores.

(c) BIST suffers from large power dissipation because it runs simultaneously for all cores on a chip except one.

A DFHTC consists of the following components:

(i) A collection of N PRGs, each one generates a b-bit block, where Nb is the width of a test vector for the CUT.

(ii) A test controller, which acts as an interface to the ATE and gets from it two signals: *SDI* (scan data in) and *SE* (scan enable). It contains a b-bit *serial-in parallel-out* shift register.

(iii) A *multiple input signature register* (MISR) that has a "scan data out" bit (*SDO*).

These components are shown in Figure 3.7.

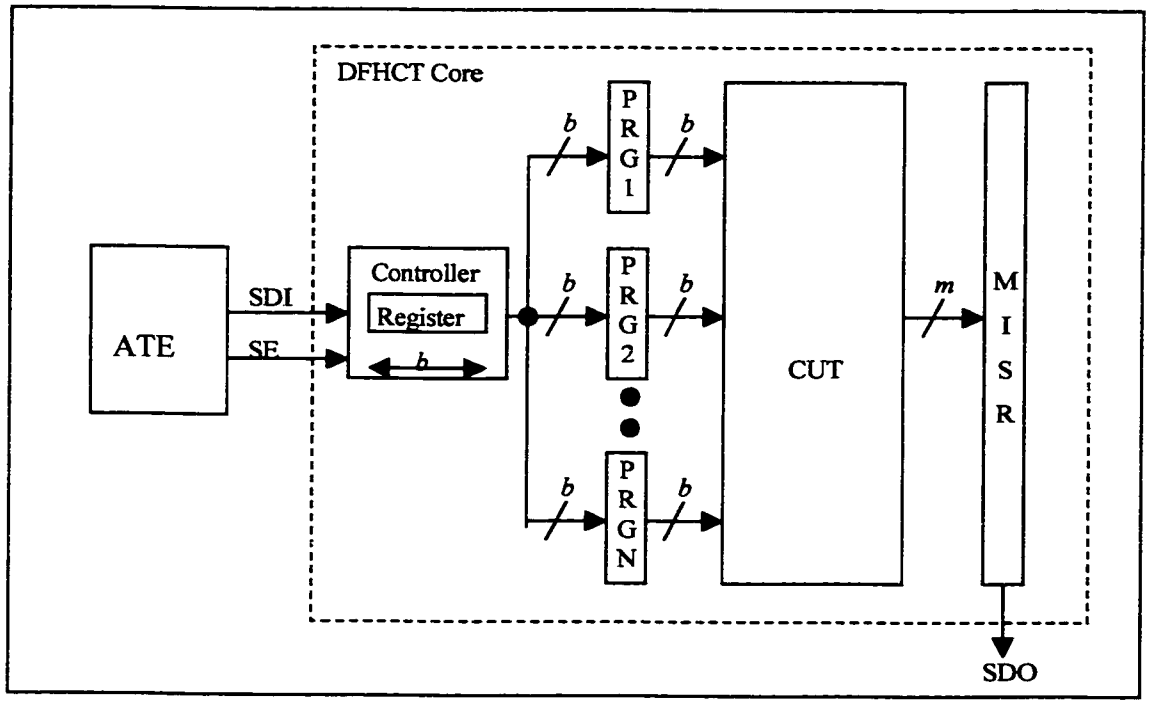


Figure 3.7. Architecture of a DFHTC core

The procedure for testing a DFHTC core can be summarized as follows:

1) During the first *b* clock cycles, the first *b* bits of a deterministic test vector are shifted in to the controller register. During this period, PRGs generate *b* random test vectors to the CUT.
2) When the *b* bits are received completely, the controller loads them to the first PRG and locks it. That is, that PRG will no longer generate random blocks. Instead, the loaded bits are generated. This acts as a "weighting" technique for the pseudo-random vectors.

3) In the following b cycles, the second b bits are shifted in and another b "weighted" random vectors are generated.

4) The second PRG is loaded with the second b bits of the test vector and locked.

5) This procedure is continued until all N blocks of the test vectors are loaded.

Figure 3.8 shows an example of generating a test vector with 12 bits. Its pattern is (100010101110). In this figure, t_i is the test vector generated from PRGs during the i^{th} cycle.

t_1	1111 1000 0011
t_2	0111 0100 0001
t_3	0011 0010 1000
t_4	0001 1100 0010
t_5	**1000** 1100 0010
t_6	**1000** 0110 1001
t_7	**1000** 1011 1100
t_8	**1000** 0101 0110
t_9	**1000 1010** 1011
t_{10}	**1000 1010** 0101
t_{11}	**1000 1010** 1010
t_{12}	**1000 1010** 1101
t_{13}	**1000 1010 1110**

Figure 3.8. Example of weighted test generation

3.2.3. Deterministic Compression Techniques

In this section, we discuss the techniques that are used to compress deterministic test data, regardless of the internal implementation of the cores (or SOC). Some of these

techniques modify some known compression techniques (explained in Section 3.2.1) to gain higher compression ratio; others come up with new ideas to compress test data. All these techniques take advantage of the high correlation between test vectors. We discuss some of these techniques based on the basic compression scheme utilized in each.

3.2.3.1. Run-Length Coding

Many proposed schemes are based on the well-known compression technique Run-Length coding. However, each proposal has some modification to the basic idea in order to get higher compression ratio. We discuss here four schemes proposed in the literature using run-length coding.

(1) Run-Length coding with Burrows-Wheeler (BW) Transformation

This scheme was proposed in [37] and used a modified version of run-length coding to encode columns of test data after performing a BW transformation on each column. The technique is based on some observations on test data. These observations are:

- Many test vectors differ only in a subset of inputs. Other inputs are kept constant. Therefore, if a test set is viewed as a matrix, some columns change their values more frequently than others. This introduces a feature of test columns called "activity". The **activity** of a string of symbols S ($\alpha(S)$) is the number of transitions on S. For instance, the string (aaabaaabcc) has an activity of 4.
- These *active* columns usually form cycles, where a **cycle** is a sequence of symbols that repeats more than once in a string. For example, the above string has a cycle (aaab) that repeats two times before it breaks.

To exploit these two characteristics of test data, Burrows-Wheeler (BW) transformation and run-length coding are used.

BW transformation is performed on a string S of length n as follows:

- Form a matrix of size nxn, the first row of the matrix is S, and the following rows are formed by rotating-left the previous row.
- Sort the matrix lexicographically.

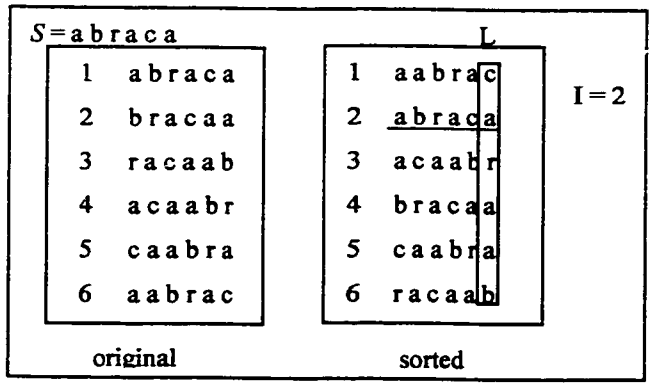


Figure 3.9. BW transformation

- To restore S from the obtained matrix, we need to know the last column L and the index I of the original string S. Details of how to restore the original string S are discussed in [5] and [25]. Figure 3.9 illustrates the BW transformation.

The advantage of BW transformation is that it usually results in less number of runs than the original set. However, this is not always the case. Sometimes, BW transformation results in more runs. This happens when the activity of the original string is greater than some threshold value α_t. Another advantage is that the inverse operation is simple since it does not involve sorting. This is good for simple decompression.

Another concept used in this technique is a modified version of run-length coding that needs fewer bits to encode a string. It is based on the idea of activity (number of transitions) of the string. The idea works as follows:

(i) Let s be a repeating symbol, L be the length of its run, t be the following symbol, and M be the length of the string.

(ii) Build a "transition table" that gives the equations to encode the next symbol after each run. This table for the three-valued logic {0,1,x} is shown in Table 3.1.

(iii) A string is encoded by giving the first symbol, then the number of transitions, and then the integers corresponding to the following

transitions using the transition table. An example is given in Figure 3.10. Since the length of all strings is the same, it is given only once. For instance, column 1 starts with 0 and there are 3 transitions. The first run is a "0" with L= 1; the following symbol is 1; so using Table 3.1, it is encoded as L, which is 1. The next run is "11", with L = 2 and the transition is to 0. From the table, it is encoded as L+M, *which* is 7. Then 0 with L = 1 and the transition to x. It is encoded as L+M which is 6. The last run does not have transitions, so no need to encode it.

Transition $s \rightarrow t$		Symbol t		
		0	1	x
Symbol s	0	_	L	L+M
	1	L+M	_	L
	x	L	L+M	_

Table 3.1. Transition table for run-length coding

Test data

1	2	3	4
0	X	0	X
1	X	0	X
1	0	0	1
0	1	0	X
X	X	0	X

Encoded data

M = 5
Column 1: (0, 3, 1, 7, 6)
Column 2: (x, 3, 2, 1, 1)
Column 3: (0, 0)
Column 4: (x, 2, 7, 1)

Figure 3.10. Modified run-length coding

The compression procedure can be summarized in the following steps:

1) Partition the test set into equal size matrices D_i of size MxQ, where M = the number of rows and Q = vector length (the last matrix may have smaller size).
2) Apply BW transformation on each individual column and compute its activity before and after the transformation.
3) Build a new matrix E_i such that each column k of E_i is the BW transformation of the corresponding column in D_i if its activity is less than the original column and less than some threshold α_t; otherwise, the original column is copied to E_i because run-length coding does not gain any compression for columns whose activities exceed α_t.
4) Use the modified run-length coding to encode each column of E_i. An example is shown in Figure 3.11 for $\alpha_t = 3$. The bolded columns are those on which BW transformation is applied. Although column 1 has less activity after BW transformation ($4 < 5$), the original column is copied to E_i because $4 > \alpha_t$.

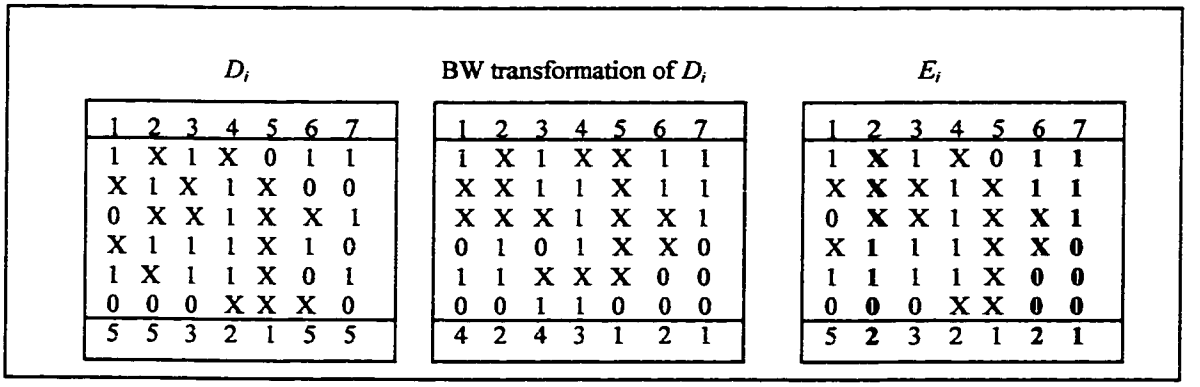


Figure 3.11 . Preparation of the matrix to be encoded

The decompression procedure is done as follows:

a) Start with the symbol given in the encoded data.

b) The length of the current run = (i%*M*) where i is the corresponding integer given in the code for the run.

c) Let $j = \left\lceil \frac{i}{M} \right\rceil$, then the following symbol in the string is the j^{th} symbol from the current symbol as shown in Figure 3.12.

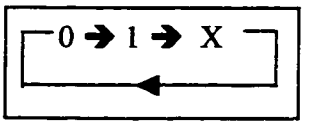


Figure 3.12. Next symbol

In [19], the authors enhanced this technique by including another compression technique called "GZIP" to encode the resultant matrix E_i. This hybrid technique is called **COMPACT**. It exploits the fact that GZIP is effective for encoding strings with high activity, while run-length coding is efficient with low-active strings. So, columns with high activity are encoded using GZIP and those with low activity are encoded with run-length coding. For example, run-length coding is performed on columns 2,4,5,6 and 7 of E_i in Figure 3.11. The rest of the columns (1 and 3) are encoded by GZIP. Two flags are needed here; one to indicate whether BW transformation was performed on the column or not, and the other to indicate which compression technique was used.

(2) Variable-to-Block Run-Length Coding

This technique was proposed in [20] for compressing fully specified test data that feeds a *Cyclical Scan Chain.* A cyclical scan chain is used to decompress this data and transfer it to the "test scan chain". Two requirements are necessary for a cyclical scan chain:

1) It must have the same number of scan elements as the test scan chain.
2) Its content must be protected against overwriting during the application of tests to the CUT. This is important because the content of the cyclical scan

Data Block (1)	Data Block (2)	Codeword
1	10	000
01	11	001
001	01	010
0001	001	011
00001	0001	100
000001	00001	101
0000001	000001	110
0000000	000000	111

Figure 3.13. variable-to-block run-length coding

chain is used to build the next test vector from the compressed data. This will be descried below.

The second requirement can be achieved by implementing the cyclical scan chain using the chip boundary scan, the boundary scan around a core, or a scan chain in a different system that has another clock.

Now let us look at the compression scheme. Test data will be compressed using the *variable-to-block* run-length code as shown in Figure 3.13. Block (1) has the feature that a three-bit codeword is used to encode a block of data based on the number of zeros in that block. The decompression for this is simply a counter that counts down to 0 and outputs a 0 each time it decrements, then outputs a 1 (except for 7 0's). Block (2) is a modification of block (1). It was obtained to have two advantages over (1); the first is that it needs less bits for runs of 1's; the second is that it needs at most six clocks to decode a block of data, which is important for the efficiency of the

decoder. The decoder for this case can be implemented by a small finite state machine (FSM).

A cyclical scan chain is implemented with a feedback to an XOR gate as shown in Figure 3.14. Therefore, if test vector *i* is currently in the cyclical scan chain, the next vector *j* is stored by scanning in the *difference vector* $i \oplus j$. Since test vectors have high correlation, it is possible to maximize the number of zeros in the difference vectors by careful sorting of the test vectors. Maximizing number of zeros will minimize the number of bits needed for encoding them using run-length coding of Figure 3.13.

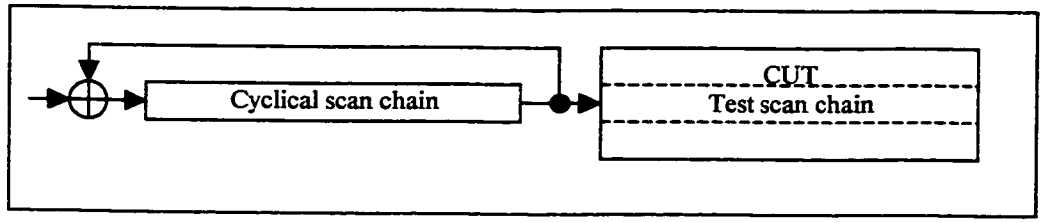


Figure 3.14. Cyclical Scan chain architecture

The compression procedure is summarized as follows:

(i) For a test set, form a cyclical scan chain with stages equal to the vector length.

(ii) Sort the vectors so as to minimize the run-length code of the difference vectors.

(iii) Initialize the cyclical scan chain to 0.

(iv) Scan in the first vector.

(v) For the following vectors, decompress the difference vectors and scan in the decompressed form.

(3) Golomb Coding

Golomb code is a *variable-to-variable* run-length code. It was used in [9] to enhance the scheme proposed in [20] (described above). It is based on the same concept of compressing test vectors by encoding the difference vectors and decompressing them in a cyclical scan chain. The authors applied it to both full scan circuits, where ordering of vectors is allowed, and non-scan (sequential) circuits, where the order of vectors must be preserved.

Golomb code divides the runs into groups, each is of size m. The number of groups is determined by the length of the longest run, and the group size m is dependent on the distribution of test data. However, optimality can only be achieved through actual experimentation.

Each group has a unique prefix. If m is selected as a power of 2 (2^N), each group will have 2^N members. Each member will have a tail that distinguishes it from other members of the group. Concatenating the prefix of the group with the tail of the member constructs the codeword of the corresponding run. This is illustrated in Table 3.2 for $m = 4$. The run-length is for 0's followed by a 1. So, if a string is 00001, for example, then its corresponding codeword is 1000 (member 1 of A_2).

Group	Run-length	Prefix	Tail	Codeword
A_1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
A_2	4	10	00	1000
	5		01	1001
	6		10	1010
	7		11	1011
A_3	8	110	00	11000
	9		01	11001
	10		10	11010
	11		11	11011
.	.	.	.	.

Table 3.2. Golomb Code (m = 4)

The decompression is done exactly as in the previous work of [20]. It needs an i-bit counter (where $i = log_2 m$) and an FSM. The FSM reads the codewords one bit at a time. For each codeword, the FSM allows the decoder to output m 0's for each bit of the prefix whose value is 1 until it reads a 0. Then, the FSM reads the tail and outputs

a number of 0's equal to the tail followed by 1. The size of the FSM depends on m. For example, if $m = 4$, then the FSM has 8 states.

(4) FDR coding

Another enhancement to the works done in [20] and [9] was proposed in [8]. It uses frequency-directed run-length (FDR) codes, which is another variable-to-variable coding technique. The following properties differentiate the FDR code from Golomb code:

1. The prefix and the tail of any codeword are of equal size.
2. In any group A_i, the prefix is of size i bits.
3. The prefix of a group is the binary representation of the run length of the first member of that group.
4. When moving from group A_i to group A_{i+1}, the length of the codewords increases by two bits, one for the prefix and one for the tail.

Table 3.3 shows the first three groups of an FDR code.

It has been shown by experimentation on ISCAS89 benchmark circuits that the run-lengths are always within groups A_1 to A_{10}.

Group	Run-length	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
.	.	.	.	.

Table 3.3 . An example of FDR code

FDR outperforms Golomb code based on the observation that the frequency of runs decreases with the increase in their lengths. Hence, assigning smaller codewords to runs with small lengths and larger codewords to those with larger lengths will decrease the overall cost.

Similar to the decoders in [9] and [20], the decoder of the FDR code is implemented using counters and an FSM. A k-bit counter (where k is the number of the last group) is used to decode the prefix and the tail of each codeword. Another counter of size $log_2\ k$ is used to identify the group number. The FSM, which consists of 9 states, controls the operations of these decoders.

Golomb codes and FDR codes have been applied to the original test sets and to the difference test sets T_D, which requires a cyclical scan chain to restore the original vectors. It has been concluded in [8] that, in general, applying these two techniques to the original test sets achieves better results than applying them to the difference test sets.

3.2.3.2. Statistical Coding

In [22], statistical coding is used for encoding deterministic test data. The technique uses a modified version of Huffman coding as to minimize the bits needed for codewords. The idea can be summarized as follows:

1) Divide the test vectors into equal size blocks, each of size b. If the size of test vectors is not divisible by b, additional x's can be added to the beginning of the vectors (since shifting them to the scan chain will not affect the test as long as the final content is the same as the original vectors).
2) Compute the frequency (number of occurrences) of each block.
3) Divide the blocks into two sets: one includes n most frequent blocks and the other includes the rest.
4) Build a Huffman tree for the n blocks (the first set).

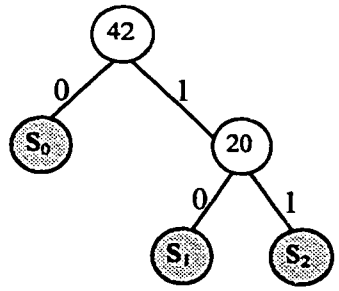
5) Encode the blocks as follows: (i) if the block belongs to the first set, its codeword is obtained from the Huffman tree with a 1 preceding it. (ii) Otherwise, the block is not encoded. Instead, it is preceded by a 0 to indicate this situation.

Figure 3.15 shows an example of the proposed technique for $b = 4$ and $n = 3$. The second column gives the frequency of each block in the test data. The last 3 blocks are listed for completeness of all possible 4-bit blocks. The bolded bit in column 4 is the added bit.

The important parameters are b and n. By careful choosing of these parameters, both compression ratio and simplicity of the decoder are improved. The decoder can be implemented by an FSM of $n + b$ states.

Symbol	Freq.	block	code
S_0	22	0010	10
S_1	13	0100	110
S_2	7	0110	111
S_3	5	0111	**0**0111
S_4	3	0000	**0**0000
S_5	2	1000	**0**1000
S_6	2	0101	**0**0101
S_7	1	1011	**0**1011
S_8	1	1100	**0**1100
S_9	1	0001	**0**0001
S_{10}	1	1101	**0**1101
S_{11}	1	1111	**0**1111
S_{12}	1	0011	**0**0011
S_{13}	0	1110	-
S_{14}	0	1010	-
S_{15}	0	1001	-

(a) 4-bit blocks and their codewords



(b) Huffman tree

Figure 3.15. Example of modified statistical coding

The benefit of this technique over the normal Huffman coding is in the simplicity of the decoding hardware, which reduces the area overhead. However, Huffman coding gives better compression ratio.

3.2.3.3. Compression by Replacement Words

This technique was proposed in [21] to compress test data that is decompressed using an embedded processor, which usually exists in a SOC. The technique is based on storing the different bits between two test vectors. It divides each test vector into blocks and stores those blocks that are different from the preceding vector. The block size is dependent on the word size of the processor. To minimize the number of different blocks, careful sorting of test vectors is required.

The compression procedure can be summarized as follows:

1) Divide test vectors into N blocks, each of which has the size of the word of the processor.
2) Sort the vectors as to minimize the different blocks between consecutive vectors.
3) Store the first block.

4) For each vector, find the different blocks. Store each block in a *replacement word.*

A replacement word has three fields as shown in Figure 3.16: (i) the last block flag which is set for the last block of a vector, (ii) the block number, which indicates which block in the vector is to be replaced by this block, and (iii) the block pattern which represents the test data that has to be replaced in the indicated block.

Figure 3.16. Replacement word

The decompression is done by first loading a program to the embedded processor. Then, the stored data is transferred into the memory on the chip. The processor runs the program and stores the blocks of a vector in a portion of the memory. When the last block is read, the processor loads the scan chain(s) with the resultant vector.

3.3. Overall Comparison

In this section, we discuss the main advantages and disadvantages of the techniques reviewed in the previous sections for reducing the size of test data.

Test compaction techniques aim at reducing the number of vectors in a test set while maintaining the same fault coverage. So, one advantage of compaction techniques is the reduction in test application time. Compaction techniques are classified into dynamic and static. The main advantage of dynamic compaction techniques is that they generate the test vectors and minimize them at the same time. However, this may increase the ATPG time. In addition, dynamic compaction techniques cannot be applied to already-generated test sets. On the other hand, static compaction techniques can be applied to any generated test set because they compact test sets after the ATPG process. The disadvantage of these techniques is that they are limited by the generated test vectors. Sometimes, other vectors that do not exist in the test set may be better in fault detection than some of the existing vectors. There is no exact criterion to tell which compaction techniques (dynamic or static) will result in less number of vectors. Although compaction techniques reduce the size of test data, it still needs to be reduced more.

Compression techniques are used to further reduce the size of test data. There are two main categories of test compression techniques. The first is based on BIST and PRGs. The problem with this class is the hardware overhead required for self-testing capability. Not all designs can afford this overhead.

The other category of test compression techniques is based on deterministic testing. This can be applied to any design because it is independent of the internal architecture. Several techniques under this category are based on run length coding. The first one uses Burrows Wheeler transformation to reduce the number of runs and then applies a modified version of run length coding. Although it is based on partitioning the test set into two-dimensional blocks, the encoding is done in one-dimensional basis. In addition, the scope of this technique is in decoding the test data in the ATE based on software algorithm. Therefore, it may not be applicable for encoding test data that is transferred to chips. The other techniques that are based on run length coding use variable-to-block or variable-to-variable run length coding. The technique based on FDR codes enhanced the one based on Golomb codes, which enhanced the variable-to-block approach. The problem of these techniques is that they encode runs of 0's followed by a 1. Hence, when the test set contains many runs of 1's, their performance in compression degrades. Another technique is based on statistical coding. The main advantage of this technique is the simplicity of the decoder. However, the decoding scheme depends on the test set. In addition, the

compression ratio achieved by this technique is not as much as the one achieved by the original statistical coding. The last technique discussed in this category encodes the difference bits in a replacement word. It achieves a good compression ratio. However, the decoding scheme depends on the existence of an embedded processor on the chip under test. If this processor is not available, this technique cannot be applied.

3.4. Concluding Remarks

In this chapter, we have reviewed some of the techniques proposed in the literature to reduce the amount of test data. These techniques are classified into two main categories, test set compaction and test data compression.

In test set compaction, the number of vectors in the test set is reduced. This is done either *dynamically* during the test generation process or *statically* after generating the test set.

In test data compression, some encoding algorithm is applied to the test data to represent it in a different format with smaller cost than the original data. This

encoded format is then decoded on chip using a decoding algorithm. Lossless compression is required for compressing test data. There are two main approaches for test data compression, one is based on BIST and PRGs and the other is based on deterministic testing. The former requires the internal architecture of the chip under test to be capable of self testing. However, the latter can be used for compression regardless of the internal architecture.

In the last section of the chapter, we have outlined the main advantages and disadvantages of the above discussed techniques.

CHAPTER 4

PROPOSED COMPRESSION SCHEME

4.1. Motivation

In the previous chapter, we have reviewed some of the techniques proposed in the literature to reduce the amount of test data. These techniques are classified into two main categories, test set compaction and test data compression.

There are two main approaches for test data compression, one is based on BIST and PRGs and the other is based on deterministic testing. The former requires the internal architecture of the chip under test to be capable of self testing. However, the latter can be used for compression regardless of the internal architecture. In this thesis, ee focus on this category.

Most of the techniques proposed for deterministic testing take advantage of the high correlation between test vectors. However, most of these techniques consider each

vector separately, i.e. they work in one-dimensional basis. So, the advantage of this correlation is not exploited completely. Although the technique that is based on BW transformation partitions the test set into two-dimensional blocks, each column is encoded separately, which means that it is again a one-dimensional approach. Another observation about these techniques is that most of them are based on variations of some of the basic compression schemes such as run length coding and statistical coding.

In this thesis, we propose a novel idea for deterministic testing. The main contributions of our proposed scheme are the following:

- A new idea for reordering the test vectors in order to maximize the correlation between consecutive test vectors.
- The partitioning of the test set into two-dimensional blocks. Each block is encoded separately and independently.
- Using geometric shapes for encoding the blocks, which exploits the high correlation between test vectors.
- The choice to encode either the 0's or the 1's in a block.
- The possibility of encoding a block as filled with 0's or 1's.

- The choice of not to encode a block and to store the real data when the cost of encoding the block is more than the cost of the original test data in that block.

In the following section, we discuss the methodology upon which our proposed scheme is based. Then we talk about the sorting algorithm in Section 4.3. In Section 4.4, we discuss the encoding process, which consists of partitioning the test set into blocks and encoding each block independently, and analyze its time complexity. Finally, we give an illustrative example in Section 4.5.

4.2. Methodology

In this work, we assume that a full scan test methodology is used for deterministic testing of SOC. The scan-based methodology is the dominantly used design-for-testability (DFT) technique as it reduces the complexity of automatic test pattern generation from that of a sequential ATPG to that of a combinational one. It also improves the testability of the design. In this methodology, sequential elements are converted into a scan type where they are connected in a scan chain that works like a shift register. During scan test mode, the desired test pattern on the sequential

elements is shifted in through the scan-in (SI) pin. Then, the vectors are applied to the primary inputs and the circuit is put in functional mode. This allows capturing the output generated by the circuit, which can be observed either through the primary outputs of the circuit or through scanning out the captured pattern through the scan-out (SO) pin. For example, consider the circuit shown in Figure 4.1. This circuit, s420.bench, is one of the ISCAS89 benchmark circuits, which has 18 primary inputs, 1 primary output, and 16 sequential elements. To convert this circuit into a full-scan circuit, the 16 FFs are converted into a shift register during test mode, and two pins are added; namely, scan-in (SI) for serially shifting the desired pattern into the scan chain, and scan-out (SO) for serially shifting out the captured output from the scan chain. An additional control pin is added, called scan enable (SE), to control the selection of data in the scan chain.

Test data was generated for the full-scanned version of the circuit and 167 vectors were needed to detect all single stuck-at faults in the circuit. Portions of the generated test vectors required for the scan chain are shown in Figure 4.1. Examining the test vectors for this circuit, and other circuits, has led to the following important observations:

- Most of the test vectors have a large number of unspecified bits.
- Similar vectors that differ by few bits can be grouped into sets.

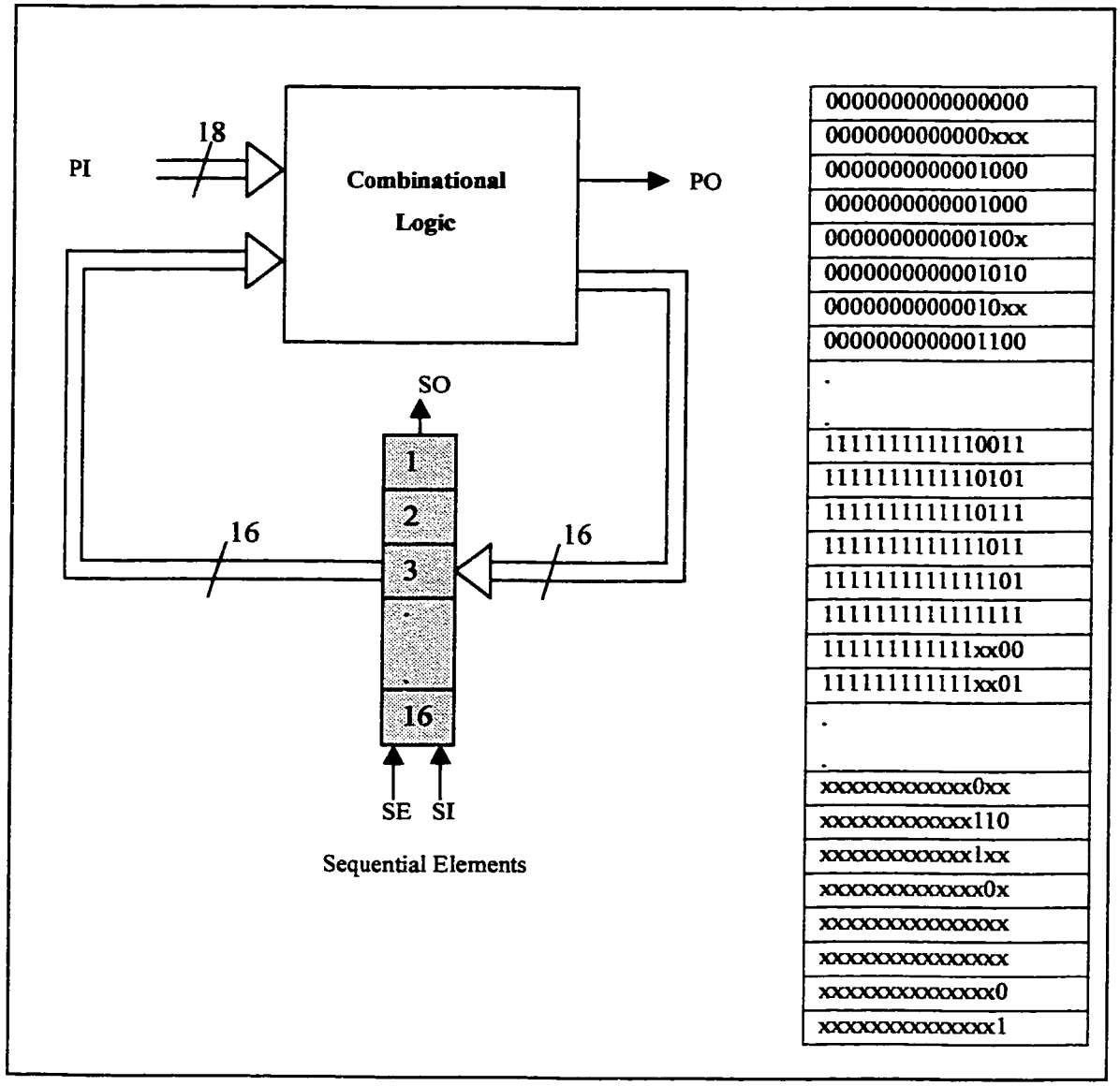


Figure 4.1. An example circuit (s420.bench) and a subset of its test vectors

These two observations clearly indicate that the test sets generated can be highly compressed. As we can see from the test set for the circuit s420, test vectors can be

repeated or they can be a subset of other test vectors. This is expected in case of testing for faults that have the same requirements on the sequential elements with different requirements on the primary inputs. In this thesis, we propose a scheme that takes advantage of the similarities between a group of test vectors resulting in a compressed test data.

Having a large number of unspecified bits also helps in the compression process as these bits can be assigned any desired value, i.e. either 0 or 1. The proposed encoding algorithm is based on encoding the 0's or the 1's in a test set by primitive geometric shapes. In this work, we limit those primitive shapes to the basic four, namely: point, line, triangle, and rectangle as shown in Table 4.1. These shapes are the most frequently encountered shapes in any test set. First, we need to order the vectors in such a way that minimizes the number of primitive geometric shapes needed to encode test data. It should be noted here that this reordering does not affect the detection of faults and the fault coverage remains the same. This step will be explained in the following section. Then, the encoding process takes place which is explained in Section 4.4.

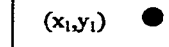
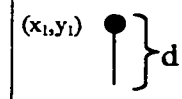
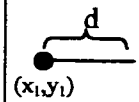
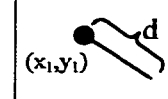
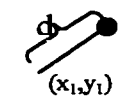
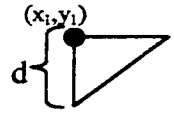
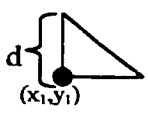
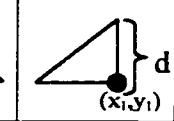
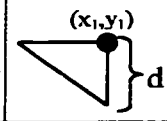
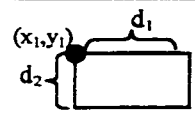
	Type 1	Type 2	Type 3	Type 4
Point		X	X	X
Lines				
Triangles				
Rectangle		X	X	X

Table 4.1. Primitive shapes used in the proposed scheme

4.3. Sorting Test Vectors

Sorting the vectors in a test set is crucial and has a significant impact on the compression ratio. In this step, we aim at generating clusters of either 0's or 1's in such a way that minimizes the number of shapes, shown in Table 4.1, needed to fit these clusters. Several sorting scenarios may be considered. In this work, we use a simple correlation-based sorting technique in which a distance D between two vectors A and B that maximizes the clusters of 0's and 1's is found. The distance D

may be computed with respect to 0's (*0-distance*), to 1's (*1-distance*) or to 0's and 1's (*0/1*-distance) as follows:

$$D = \sum_{i=0}^{k-1} W(A_i, B_{i-1}) + W(A_i, B_i) + W(A_i, B_{i+1})$$

where k is the test vector length and $W(A_i, B_i)$ is the weight between bits A_i and B_i. Table 4.2, Table 4.3 and Table 4.4 specify the weights used in computing the 0-distance, the 1-distance, and the 0/1-distance between two vectors, respectively. The assignment of these weights will be discussed later. Note that for i = 0, $W(A_i, B_{i-1}) = 0$ and for i = k-1, $W(A_i, B_{i+1}) = 0$. Finding the 0-distance results in *0-sorting*, finding the 1-distance results in *1-sorting* and finding the 0/1-distance results in *0/1-sorting*.

	0	1	x
0	1.0	0.0	0.25
1	0.0	0.0	0.0
x	0.25	0.0	0.25

Table 4.2. Weights for the 0-distance between two test vectors.

	0	1	x
0	0.0	0.0	0.0
1	0.0	1.0	0.25
x	0.0	0.25	0.25

Table 4.3. Weights for the 1-distance between two test vectors.

	0	1	x
0	1.0	0.0	0.25
1	0.0	1.0	0.25
x	0.25	0.25	0.25

Table 4.4. Weights for the 0/1-distance between two test vectors

The sorting algorithm starts by selecting a vector to be the first vector in the sorted test set. Then, for each test vector added to the sorted set, the distance D to all remaining vectors is computed. The vector that generates the maximum distance is then selected and added to the test set. This process continues until all vectors are included.

The following factors may affect the sorting and hence may affect the compression ratio:

(1) Selecting the first vector:

The first vector selected in the new test set is important because it is the base upon which all other vectors are selected. Different approaches in selecting the first vector can be considered. We have experimented with two of them. The first one is to select the first vector in the original test set to be the first vector in the sorted test set. The second approach is to select the vector with

the maximum number of 1's for 1-sorting and the maximum number of 0's for 0-sorting. This approach is better because it puts this vector in the first line of the test set which may arrange the shapes in a better way than if it is put in the middle of the test set. For 0/1-sorting, it is possible to select the vector with the maximum number of 1's or with the maximum number of 0's. However, we always select the one with maximum number of 0's because we have found by experiments that 0-sorting gives better results than 1-sorting most of the time as will be shown in Chapter 6. A third possible approach is to select the first vector randomly. However, this is similar to the first approach and does not have any advantage.

(2) The weight of the 'x' bit:

Assigning a weight to the x bit whenever its neighbor bit is x or the bit used for sorting (0 for 0-sorting, 1 for 1-sorting and 0 and 1 for 0/1-sorting) is done due to the following reasons. First, this weight may help in completing, integrating, or generating additional geometric shapes that can lead to a better solution. Second, this can help in generating blocks filled with 'x's which can be minimally encoded. It is possible to treat the x bit as the bit used for sorting or treat it differently. In the first case, we assign a weight of 1.0 to the x bit. In the second case, the x bit is assigned a weight that is less than 1.0.

The advantage of the second method is that it favors the vectors with more bits of the type used in the sorting than those with x's, which may minimize the shapes needed for encoding the bits; taking into consideration that each block is encoded separately. As shown in Table 4.2, Table 4.3 and Table 4.4, the weight of the x bit is assigned 0.25 as the second method suggests. We have also experimented with other weights including the weight of 0.5 and 1.0 and found that a weight of 0.25 gives better results in general. In the experimental results that will be shown in Chapter 6, we only show the results of weights 0.25, 0.5 and 1.0 to give an insight of the effect of the x-weight.

(3) The 'next' vector:

As explained above, the sorting algorithm selects each time the closest vector to the last vector selected. We call this approach the "greedy" approach. One can think of optimizing the solution by finding a Hamiltonian cycle between all vectors, which gives a near-optimal solution [12]. We have tried this near-optimal sorting but we have found that greedy sorting is better. We have tried another near-optimal sorting in which the Hamiltonian cycle is built for the whole set and then a subset of these vectors are selected. Then, the Hamiltonian cycle for the remaining vectors is built and another subset is

selected and so on. The size of the subset could be 8, 16 or 32 (depending on the block size as will be explained in the next section). However, greedy sorting also gave better results than this algorithm. A possible reason for this will be discussed in Chapter 6.

4.4. Encoding Process

After sorting the test set, the encoding process takes place. Since this process is performed offline, it does not matter how much complicated it is. However, the decoding process must be simple enough to be run online. The decompression process will be explained in Chapter 5. Figure 4.2 shows an outline of the encoding algorithm. In the following subsections, we explain each step in details.

4.4.1. Test Set Partitioning

A set of sorted test vectors, M, is represented in a matrix form, TxV, where T is the number of test vectors and V is the length of each test vector. The test set is

segmented into LxK blocks each of which is NxN bits, where L is equal to $\left\lceil \frac{T}{N} \right\rceil$ and K is equal to $\left\lceil \frac{V}{N} \right\rceil$. A *segment* consists of K blocks. In other words, the test set is segmented into L segments each contains K blocks. For test vectors whose columns and/or rows are not divisible by the predetermined block dimension N, a partial block will be produced at the right end columns and/or the bottom rows of the test data. These partial blocks are useful because the number of bits used to encode the coordinates of the geometric shapes can be less than $log_2 N$. The decoder is provided by a test header consisting of five parameters that help in decoding the test set. These parameters are the block size (N), the number of segments (L), the number of blocks per segment (K), the remainder (R) of dividing T by N, and the remainder (C) of dividing V by N.

4.4.2. Encoding Blocks

For each block of the test set, the procedure *Extract_Shapes(b)* finds the best group of shapes that cover the bits that are of type b as shown in the algorithm in Figure 4.2. It goes through all bits in the block line by line from left to right. For each bit of

```
Encoder (N)
    Partition_Test_Set (N);
    For i = 1 to # of segments
      For j = 1 to # of blocks in i
        Extract_Shapes (1, j);
        α1 = Encode_Shapes ();
        Extract_Shapes (0, j);
        α0 = Encode_Shapes ();
        B = # of bits in j + 2;
        E = min (α0, α1,B);
        Store_Encoded_Bits ();
        E_total += E;
End Encoder;

Extract_Shapes(b, j)
  For each bit x in block j {
    If x = b Then {
      Find the largest line of each type starting at x
      Find the largest triangle of each type such that x is the vertex of the right
          angle
      Find the largest rectangle such that x is its up-left corner
    }
  }
   Solve a covering problem to find the best group of shapes covering all bits b in
     block j.
End Extract_Shapes;
```

Figure 4.2. Test vectors encoding algorithm

type b, the procedure finds the largest shape of each type covering this bit as stated in the figure. Then, after finding all possible shapes for all bits, a covering problem is solved to select the best group of shapes that cover all bits with minimum cost.

The procedure *Encode_Shapes* determines the number of bits, α, needed to encode the group of shapes found by *Extract_Shapes*. There are two cases that may occur:

(a) The block contains only 0's and x's or only 1's and x's. In this case, the block can be encoded as a rectangle. However, instead of this, it is encoded as "01" (indicating that the block can be filled by 0's or 1's) followed by the bit that fills the block. Hence, the number of bits to encode the block $\alpha = 3$. We call such blocks *filled blocks*.

(b) The block needs to be encoded by a number of shapes. We call such a block *encoded block*. In this case, we need the following:

- 2 bits to indicate the existence of shapes and the type of bit encoded. If the encoded bit is 0, then the code is 10, otherwise it is 11.
- $P = (2*\text{Log}_2 N - 3)$ bits to encode the number of shapes, S. If the number of shapes exceeds 2^P, then the number of bits needed to encode the shapes is certainly greater than the total number of bits in the block. In this case, the block is not encoded and the real data is stored. Therefore, selecting $N = 4$ or less is not effective in our technique because the maximum possible number of shapes in this case $= 2^P = 2^{2*2-3} = 2^1 = 2$ shapes. Hence, we have experimented with 8x8, 16x16, and 32x32 block sizes.
- $\sum_{i=1}^{S} L_i$ bits; where L_i is computed as follows (refer to Table 4.1)

- If shape i is a point, $L_i = 2 + 2*\log_2 N$ (shape type + 2 coordinates).
- If shape i is a line or a triangle, $L_i = 2 + 2 + 3*\log_2 N$ (shape type + type of line or triangle + 2 coordinates + distance)
- If shape i is a rectangle, $L_i = 2 + 4*\log_2 N$ (shape type + 2 coordinates + 2 distances)

Therefore, $\alpha = 2 + P + \sum_{i=1}^{S} L_i$

For partial blocks, the encoder will output the needed bits and the decoder will take care of that. This will be explained in more detail in Chapter 5.

If α_0 (number of bits needed to encode shapes with 0) and α_1 (number of bits needed to encode shapes with 1) are greater than B which equals (N*N+2), then it is better not to encode the block. Instead, the real data is stored after a 2-bit code (00). We call such blocks *real-data blocks*. The procedure *Store_Encoded_Bits* will decide which case is the best (encoding 0's, encoding 1's, or storing the real data) based on E, which is the minimum of α_0, α_1, and B.

4.4.3. Time Complexity

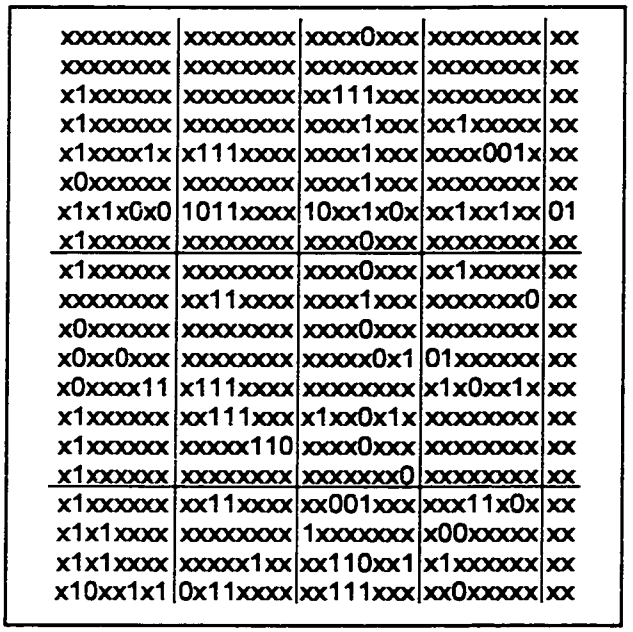
Now let us analyze the time complexity of the encoding process (see [2] for more information about time analysis). This process consists of two algorithms, the sorting and the encoding. The sorting algorithm compares every bit of a vector with three bits of every other vector. This requires $O(VT^2)$ time; where T is the number of vectors in the test set and V is the vector length. This algorithm is executed once before the encoding algorithm.

For the encoding algorithm, there are three main steps, test set partitioning, extracting shapes in each block, and solving a covering problem to select the best group of shapes. The partitioning step requires constant time; i.e. it runs in O(1) time. For each block, all possible shapes that cover every bit of the block are extracted. There are N^2 bits in a block; where N is the dimension of the block. Extracting each type of shapes requires $O(N^2)$ time at most (for example, the rectangle). Since we have constant number of shapes, the time complexity of extracting shapes for each block is $O(N^4)$. Then, a covering step is performed to select the best group of shapes. The maximum number of shapes for any block (before selecting) is $10*N^2$; where 10 is the number of shape types. Therefore, this step requires $O(N^2)$. Hence, the encoding algorithm for each block requires $O(N^4)$ time. There are L*K blocks; where

$L = \left\lceil \frac{T}{N} \right\rceil$ and $K = \left\lceil \frac{V}{N} \right\rceil$. Therefore, the total time complexity of the encoding algorithm is $O(LKN^4) = O(TVN^2)$. Since the maximum value of N in our algorithm is 32, then $N^2 = 1024$ at most, which means that N^2 is constant. Hence, the time complexity of the algorithm is O(TV), which means that the algorithm runs in linear time with respect to the size of the test set. The (N^2) term gives an indication that the time needed by the encoding algorithm increases with the increase in the block size.

4.5. An Illustrative Example

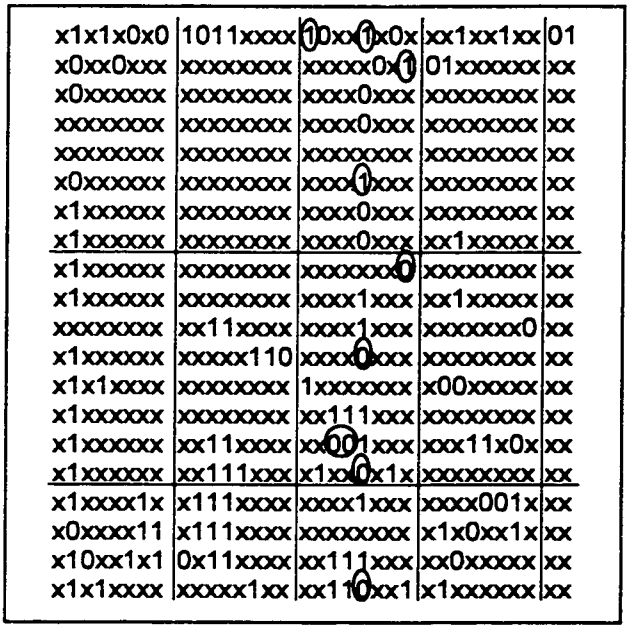
In this section, we give an example that illustrates the sorting and encoding processes. We illustrate this example on a portion of a test set shown in Figure 4.3. This subset consists of 20 vectors of length 34 each. The horizontal lines divide the subset into segments of 8 vectors each except for the last segment, which consists of 4 vectors (the remainder vectors, or row remainder). The vertical lines divide each segment into blocks of size 8x8. Notice that the last block of each segment consists of vectors of two bits each (column remainder). Therefore, the subset has been divided into 15 blocks, 8 of them are of size 8x8, 4 are of size 4x8, 2 are of size 8x2, and one is of size 4x2 (which is the last block of the last segment).



xxxxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
xxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xx
x1xxxxx	xxxxxxxx	xx111xxx	xxxxxxxx	xx
x1xxxxx	xxxxxxxx	xxxx1xxx	xx1xxxxx	xx
x1xxxx1x	x111xxxx	xxxx1xxx	xxxx001x	xx
x0xxxxx	xxxxxxxx	xxxx1xxx	xxxxxxxx	xx
x1x1x0x0	1011xxxx	10xx1x0x	xx1xx1xx	01
x1xxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
x1xxxxx	xxxxxxxx	xxxx0xxx	xx1xxxxx	xx
xxxxxxx	xx11xxxx	xxxx1xxx	xxxxxxx0	xx
x0xxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
x0xx0xxx	xxxxxxxx	xxxxx0x1	01xxxxxx	xx
x0xxxx11	x111xxxx	xxxxxxxx	x1x0xx1x	xx
x1xxxxx	xx111xxx	x1xx0x1x	xxxxxxxx	xx
x1xxxxx	xxxxx110	xxxx0xxx	xxxxxxxx	xx
x1xxxxx	xxxxxxxx	xxxxxxx0	xxxxxxxx	xx
x1xxxxx	xx11xxxx	xx001xxx	xxx11x0x	xx
x1x1xxxx	xxxxxxxx	1xxxxxxx	x00xxxxx	xx
x1x1xxxx	xxxxx1xx	xx110xx1	x1xxxxxx	xx
x10xx1x1	0x11xxxx	xx111xxx	xx0xxxxx	xx

Figure 4.3. An example of a subset of a test set

Figure 4.4, Figure 4.5 and Figure 4.6 show the subset after sorting it using the 0-distance, the 1-distance and the 0/1-distance, respectively.



x1x1x0x0	1011xxxx	10xx1x0x	xx1xx1xx	01
x0xx0xxx	xxxxxxxx	xxxxx0x1	01xxxxxx	xx
x0xxxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
xxxxxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xx
x0xxxxxx	xxxxxxxx	xxxx1xxx	xxxxxxxx	xx
x1xxxxxx	xxxxxxxx	xxxx0xxx	xxxxxxxx	xx
x1xxxxxx	xxxxxxxx	xxxx0xxx	xx1xxxxx	xx
x1xxxxxx	xxxxxxxx	xxxxxxx0	xxxxxxxx	xx
x1xxxxxx	xxxxxxxx	xxxx1xxx	xx1xxxxx	xx
xxxxxxxx	xx11xxxx	xxxx1xxx	xxxxxxx0	xx
x1xxxxxx	xxxxx110	xxxx0xxx	xxxxxxxx	xx
x1x1xxxx	xxxxxxxx	1xxxxxxx	x00xxxxx	xx
x1xxxxxx	xxxxxxxx	xx111xxx	xxxxxxxx	xx
x1xxxxxx	xx11xxxx	xx001xxx	xxx11x0x	xx
x1xxxxxx	xx111xxx	x1xx0x1x	xxxxxxxx	xx
x1xxxx1x	x111xxxx	xxxx1xxx	xxxx001x	xx
x0xxxx11	x111xxxx	xxxxxxxx	x1x0xx1x	xx
x10xx1x1	0x11xxxx	xx111xxx	xx0xxxxx	xx
x1x1xxxx	xxxxx1xx	xx110xx1	x1xxxxxx	xx

Figure 4.4. The test set after 0-sorting

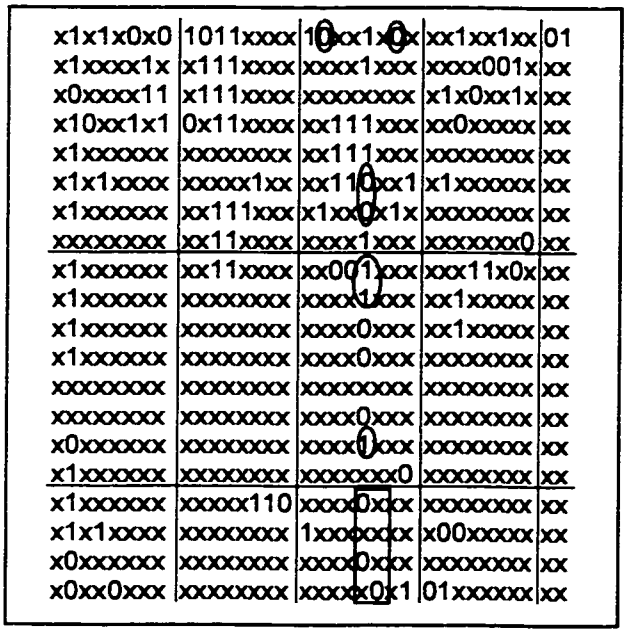


Figure 4.5. The test set after 1-sorting

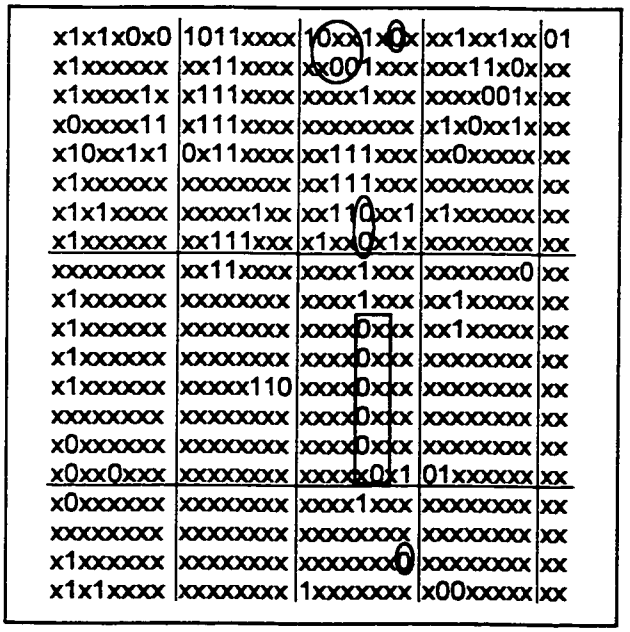


Figure 4.6. The test set after 0/1-sorting

The first vector in Figure 4.4 and in Figure 4.6 is the one with the maximum number of 0's, while it is the one with the maximum number of 1's in Figure 4.5. As an example, consider the third block of each segment. The shapes needed to be encoded are circled. Let us analyze each block individually. In this analysis, we consider only the cost of the shapes since the cost of the header information is the same for all blocks. Remember that, for 8x8 blocks, the cost of a point = 2+3+3 = 8, the cost of a line = 2+2+3+3+3 = 13, the cost of a triangle = 2+2+3+3+3 = 13, and the cost of a rectangle = 2+3+3+3+3 = 14,

1) The first block: In Figure 4.4, the bit used for encoding shapes is 1 and there are 4 points. So, the cost of this block = 4*8 = 32. In Figure 4.5, the bit is 0 and there are 3 shapes, 2 points and 1 line. So, the cost of this block = 13 + 2*8 = 29. In Figure 4.6, the bit is also 0 and there are 3 shapes again, 1 point, 1 line and 1 rectangle. So the cost of this block = 8+13+14 = 35. Notice the benefit of the x's in encoding the rectangle. Although the number of shapes is equal in Figure 4.5 and Figure 4.6, the cost is different because the cost of a rectangle is more than that of a point.

2) The second block: In Figure 4.4, the bit used for encoding shapes is 0 and there are 4 shapes, 3 points and one line. So, the cost of this block = 3*8 + 13 = 37. In Figure 4.5, the bit is 0 and there are 2 shapes, 1 point and 1 line. So, the cost of

this block = 13 + 8 = 21. In Figure 4.6, the bit is 0 and there is 1 shape which is a rectangle. So the cost of this block = 14.

3) The third block: This block is a partial block where the y-dimension is less than 8. Therefore, we can represent the y-dimension by 2 bits ($\log_2 4$) instead of 3 bits. In Figure 4.4, there is 1 point. So, the cost of this block = 2+3+2 = 7. In Figure 4.5, there is 1 rectangle. So, the cost of this block = 2+3+2+3+2 = 10. In Figure 4.6, there is 1 point. So, the cost of this block = 2+3+2 = 7.

After encoding the three blocks, we find that

- The cost of the 0-sorting (Figure 4.4)= 32+37+7= 76.
- The cost of the 1-sorting (Figure 4.5)= 29+21+10= 60.
- The cost of the 0/1-sorting (Figure 4.6)= 35+14+7= 56.

This shows that, for this part of the subset, the 0/1-sorting gives the best compression.

When we applied the encoding algorithm to the three subsets, we got the following compression ratios:

- 57.5% for the 0-sorting (Figure 4.4).
- 58.38% for the 1-sorting (Figure 4.5).
- 59.7% for the 0/1-sorting (Figure 4.6).

4.6. Concluding Remarks

In this chapter, we have discussed our proposed compression scheme. First, we have talked about the motivation of our work by looking at the disadvantages of the previous proposed techniques and the need for improvement in the compression schemes. Then, we have outlined the methodology of our technique, which is based on encoding the test data using two-dimensional geometric shapes. The geometric shapes used in our proposed technique are point, four types of a line, for types of a triangle, and rectangle.

The proposed compression technique consists of three main steps. First, the test vectors are sorted in order to minimize the number of shapes needed to encode the test data. Second, the test set is partitioned into equal size blocks. Third, each block is encoded separately. There are three possibilities for encoding a block. The first is to fill the whole block with 0's or 1's. The second is to encode the 0 bits or the 1 bits in the block using geometric shapes. The third is to store the real data.

The time complexity of the sorting algorithm is $O(VT^2)$, where T is the number of vectors and V is the vector length. The encoding algorithm requires O(VT) time, which is linear with respect to the size of the test set.

CHAPTER 5

DECOMPRESSION PROCESS

One of the main issues when designing a compression scheme for testing data is the implementation of the decompressor (or the decoder). The decoder of any compression scheme must be simple enough to achieve two requirements, minimizing the time needed for decompression and minimizing hardware overhead. We can classify the decoders of the compression schemes described in Chapter 3 into three main categories:

1. The scan chains available in the SOC are exploited to implement the decoder with possibly some additional logic.
2. An FSM is used to decompress the testing data. Sometimes, additional hardware is needed.
3. If there exists an embedded processor in the SOC, a microcode is loaded to this processor and used to decode the compressed data.

Since our proposed compression scheme does not require any specific internal architecture, the first solution is not applicable here. We have implemented the

decoder for our scheme using the other two choices. We call the third choice the *software decoder* and this is explained in Section 5.1. The second solution is called the *hardware decoder* and is explained in Section 5.2. In Section 5.3, we outline the interface between these decoders and the whole system.

5.1. Software Decoder

Most of the SOC's have embedded processors and some amount of memory inside the chip. In this case, the decoder can be implemented as a microcode executed by the processor to output the original test vectors. In our scheme, some amount of temporary memory is needed to store the blocks one after the other until a whole segment is decoded. Then, the test vectors of that segment are applied to the scan chains in order.

Figure 5.1 shows the pseudo-code of the decoding algorithm. It first reads the arguments given by the encoder. In order to reconstruct the vectors, each segment has to be stored before sending its vectors to the circuit under test. For each segment,

its blocks are decoded one at a time. The first two bits indicate the status of the block as follows:

- 00: the block is not encoded and the following N*N bits are the real data.
- 01: fill the whole block with 0's or 1's depending on the following bit.
- 10: There are shapes that are filled with 0's.
- 11: There are shapes that are filled with 1's.

For those blocks that have shapes, the procedure *Decode_Shapes* is responsible for decoding these shapes. It reads the number of shapes in the block and then for each shape it reads its type and based on this it reads its parameters and fills it accordingly.

Based on the arguments read first, the decoder can determine the number of bits needed for each variable (e.g. the coordinates and the distances). These are used for the partial blocks when only one block of each segment remains and when the last segment is being decoded.

Similar to the complexity analysis shown in Chapter 4 for the encoding algorithm, we can conclude that the time required by the software decoder is O(VT). This means that it runs in linear time with respect to the test set size. It should be noted

here that this algorithm is much simpler than the encoding algorithm because it does not require extracting shapes; i.e. the (N^2) term found in the analysis of the encoding algorithm does not exist here.

```
Decoder ()
  Read (N, # of segments (L), # of blocks per
segment (K), row remainder (R), column
remainder (C) );
   For i = 1 to # of segments {
    For j = 1 to # of blocks in i {
      b1b0 = Read_Bits (2);
      Case b1b0
         00 : Read_Bits (N* N);
         01 : b_type = Read_Bits (1);
              Fill_Block (j, b_type);
         10 : Decode_Shapes (0);
         11 : Decode_Shapes (1);
      End Case;
    }
    Output_Segment ();
  }
End Decoder;

Decode_Shapes (b)
  Num_Shapes = Read_Bits (2log2 N -3);
  For j = 1 to Num_Shapes
    Shape_type = Read_Bits (2);
    Case Shape_type
        00 : c = Get_Coordinate ();
             Fill_Point (b,c);
        01 : t = Get_Type ();
             c = Get_Coordinate ();
             d = Get_Distance ();
             Fill_Line(b,t,c,d);
        10 : t = Get_Type ();
             c = Get_Coordinate ();
             d = Get_Distance ();
             Fill_Triangle(b, t, c,d);
        11 : c = Get_Coordinate ();
             d1 = Get_Distance ();
             d2 = Get_Distance ();
              Fill_Rectangle (b,c1,d1,d2);
End Decode_Shapes;
```

Figure 5.1. Test vector decoding algorithm.

5.2. Hardware Decoder

The hardware decoder is implemented using an FSM controlling the data path which consists of some counters, registers and some basic gates. The data path is shown in Figure 5.2 and is explained in Section 5.2.1. Section 5.2.2 discusses the FSM implementation, which is shown in Figure 5.3. The hardware decoder has been designed and then modeled and verified using VHDL [3]. The basic components of the data path have been modeled functionally while the data path has been modeled structurally. The FSM has been modeled algorithmically. The VHDL code of the hardware decoder is shown in Appendix A.

5.2.1. Data Path Implementation

The data path consists mainly of some registers and counters. The registers are:

(1) A shift register *I* is used to hold the input data before loading it to the corresponding register or counter. So, the size of this register is the maximum size of all registers and counters, which is 12 bits (as shown in Figure 5.2).

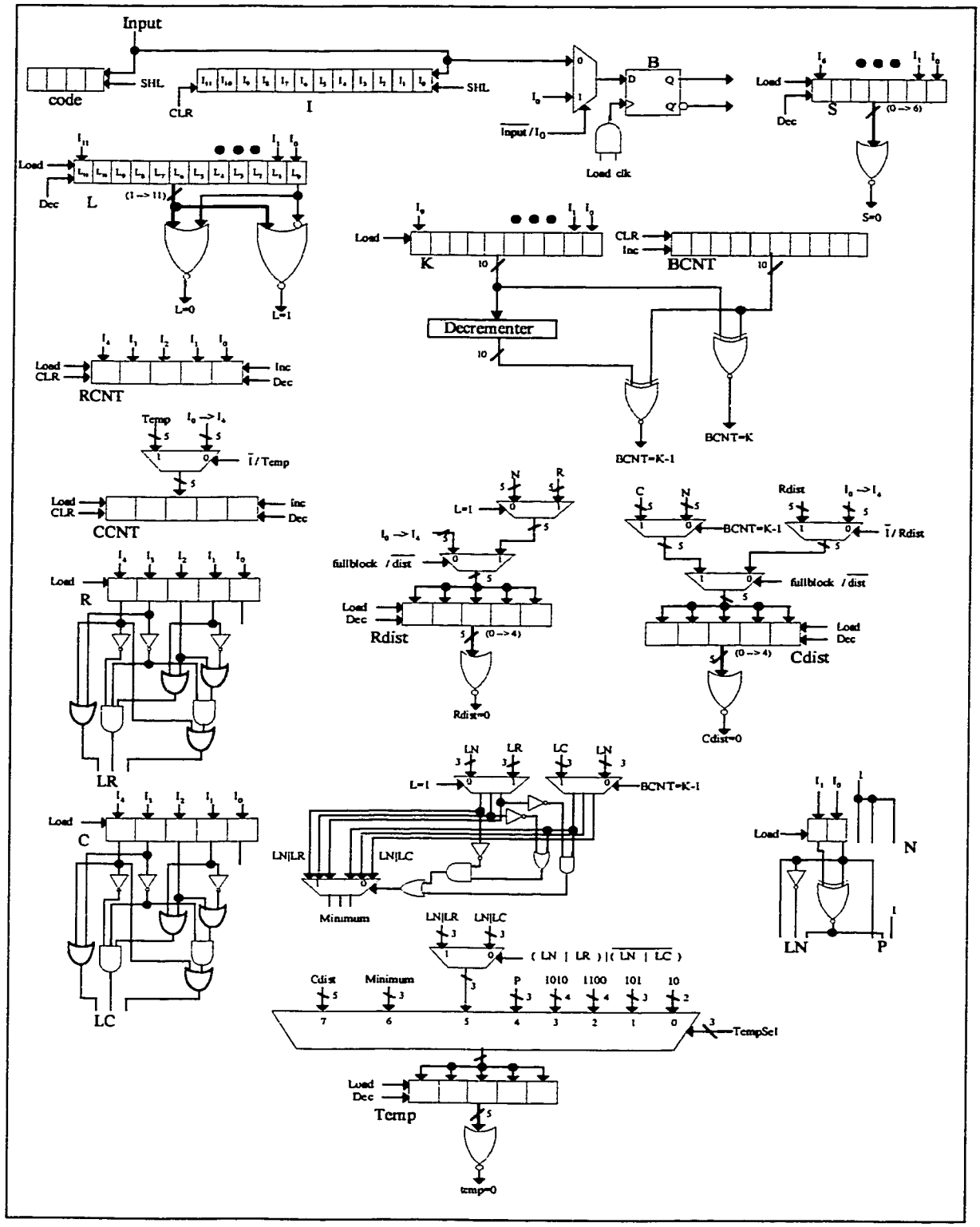


Figure 5.2. Data path of the decoder

A shift-left signal (*SHL I*) is used to shift a bit from the input data to the LSB of *I* and a clear signal (*CLR I*) is used to reset the register.

(2) Another shift register (*code*) is used to save the type of the shape that is currently decoded (point, line...etc) and the type of that shape if it is line or triangle. The size of this register is 4 bits. Only one signal is needed to control this register which is (*SHL code*) that shifts a bit from the input data to the LSB of *code*.

(3) A 1-bit register (B) to save the bit with which the current block is filled. This FF can be loaded from either the input data or from I_0. So, a MUX is needed to select between these two inputs. The signals needed here are *Load B* and the select signal.

(4) A 2-bit register (*N*) is used to save the block dimension (8, 16 or 32). We need to get the actual size *N* from two bits given by the encoder as follows:

- 00 ➔ $N = 8 = 00111$ (we start counting from 0).
- 01 ➔ $N = 16 = 01111$.
- 11 ➔ $N = 32 = 11111$.

Let the two bits given by the encoder be I_1 and I_0 and the needed 5 bits be $N_4N_3N_2N_1N_0$. Then, we find that $N_4 = I_1$, $N_3 = I_0$, and $N_2 = N_1 = N_0 = 1$. The last three bits can be stored as wires connected to VDD. Therefore, the only

hardware added here is a two-bit register connected directly to the least significant two bits of the input shift register I as shown in Figure 5.2. For this register, only a *Load* signal is required.

(5) Another two 5-bit registers are used to save the row remainder R and the column remainder C. These two registers will be loaded directly from the input register I. For these two registers, only a *Load* signal is required.

(6) In order to know how many bits are to be read for each dimension (for the coordinates and the distances), $\log_2$ of the current dimension (N, R, or C) is required. In addition, we need to know how many bits are needed to store the number of shapes. We showed in Section 4.4 that this number (P) depends on the dimension of the block N such that $P = 2^{*}\log_2 N - 3$. All these can be obtained from the N, R and C registers using some combinational logic. We illustrate this as follows:

First, we want to get $\log_2 N$ (LN) as follows:

- $N = 00111$ ➔ $LN = 011$ ($\log_2 8 = 3$).
- $N = 01111$ ➔ $LN = 100$ ($\log_2 16 = 4$).
- $N = 11111$ ➔ $LN = 101$ ($\log_2 32 = 5$).

We can notice that $LN_2 = N_3$, $LN_1 = \overline{N_3}$ and $LN_0 = N_4$ XNOR N_3.

We can get the value of P from N as follows:

- 00111 (N = 8) ➔ 011 ($2*\log_2 8 - 3 = 3$).
- 01111 (N = 16) ➔ 101 ($2*\log_2 16 - 3 = 5$).
- 11111 (N = 32) ➔ 111 ($2*\log_2 32 - 3 = 7$).

Notice that $P_2 = N_3$, $P_1 = N_4$ XNOR N_3 and $P_0 = 1$.

For the partial blocks, the dimensions range between 1 and 31. We need to get $\log_2$ of these dimensions to know how many bits need to be read for the coordinates and distances in these blocks. Let the input (the dimension given by the encoder) be $A_4A_3A_2A_1A_0$ and the output (the bits needed) be $B_2B_1B_0$, then the following truth table is obtained:

A_4 A_3 A_2 A_1 A_0	$B_2B_1B_0$	
1 x x x x	1 0 1	5 bits needed for 17 to 31
0 1 x x x	1 0 0	4 bits needed for 9 to 15
0 0 1 x x	0 1 1	3 bits needed for 5 to 7
0 0 0 1 x	0 1 0	2 bits needed for 2 and 3
0 0 0 0 1	0 0 1	1 bit needed for 1

Using K-map technique, we get the following equations for B_2, B_1, and B_0:

- $B_2 = A_4 + A_3$.
- $B_1 = \overline{A_4} . \overline{A_3} . (A_2 + A_1)$.

- $B_0 = A_4 + \overline{A_3} \cdot (A_2 + \overline{A_1})$.

(7) The last register needed in the data path is *K* which holds the number of blocks in a segment. In our implementation, we assume that the maximum vector length is 8K. Therefore, the maximum number of blocks in a segment is 1K blocks (when the dimension of a block = 8) and hence *K* is a 10-bit register.

Now, let us discuss the counters used in the data path:

(1) The first counter needed is *L* which is initially loaded with the number of segments in the test set. The size of this counter = $\lceil \log_2 L \rceil$. We assume that the maximum number of vectors in a test set is 32K vectors. Therefore, the maximum number of segments = 32K / 8 = 4K. So, the size of *L* = 12 bits. Whenever a segment is decoded, it is sent to the scan chains and *L* is decremented. When *L* = 0, the process is terminated. This condition can be checked by NORing all bits of *L*. Another condition that has to be checked is when *L* = 1 whereby the last segment is to be decoded. This can be checked also by NORing all bits of *L* with inverting L_0. The signals that we need here are *Load* and *Dec* (Decrement).

(2) Another counter (*BCNT*) is required to keep track of the block number within the current segment. The size of this counter equals the size of register *K* which is 10 bits. This counter must start counting from 0 because it is used for addressing the memory (as will be explained shortly). Therefore, we did not use this counter as in the case of the number of segments *L*. Instead, we added some comparators to check for the last block and to check if all blocks in a segment have been decoded. For each segment, *BCNT* is cleared first and then incremented for every block decoded until it equals *K*, which means that all blocks in the current segment have been decoded. This condition can be checked by XNORing every bit of *BCNT* with the corresponding bit of *K* then ANDing the results. To know when the last block of the current segment is to be decoded, *BCNT* is compared with *K*-1, which is obtained by decrementing the content of *K* and XNORing the result with *BCNT*. The signals needed to control *BCNT* are *CLR* (Clear) and *Inc* (Increment).

(3) In each block, there may be some shapes encoded. To know how many shapes are in the block, a counter *S* is used. The size of this counter = 7 bits ($2*\log_2 32 - 3$) which is the maximum possible for all block sizes as explained in Section 4.4. For each block that has shapes, *S* is loaded with the

number of shapes. Whenever a shape is decoded, *S* is decremented until it reaches 0. Also here we need to check for 0 (similar to *L*). The signals needed are *Load* and *Dec.*

(4) Four 5-bit counters are used for decoding shapes and writing them to memory. These are *RCNT, CCNT, Rdist* and *Cdist. RCNT* and *CCNT* are used to address the bit to be written within the current block in the form (row, column), respectively. They are loaded with the coordinate of a shape and then incremented or decremented according to the direction of writing. *Rdist* and *Cdist* are used for the length of writing in each direction. They are loaded with the distance and then decremented until they reach 0. Hence, a check for 0 is needed for each. The loading can be from *N, R,* or *I* for *Rdist* and from *N, C, I* or *Rdist* for *Cdist.* This depends on the block number, on whether a full block is to be filled or only a portion of it and on the type of the shape (line, triangle, or rectangle). For *RCNT* and *CCNT*, the signals needed are *Load, CLR, Inc* and *Dec.* For *Rdist* and *Cdist*, the signals are *Load, Dec* and the select signals.

(5) The last counter is a temporary counter (*temp*) that is used mainly to decide the number of bits to be read from input data. Since there are many cases, the value that is loaded to *temp* must be selected depending on the parameter

to be read. The values *LN|LR, LN|LC,* and *Minimum* are used to select between full blocks and partial blocks. Each value is selected in a certain case as shown in Figure 5.3. For reusing resources, *temp* is used as a temporary register in the case of decoding a triangle. In this case, it is loaded from *Cdist.* In all cases, we need to know when *temp* = 0 to stop reading data. So, a check for 0 is required. The signals required to control *temp* are *Load, Dec* and the select signals.

For hardware implementation, as well as for software implementation, some amount of memory is required to store a segment before applying its vectors to the CUT. The size of this memory is equal to the size of the scan chain times the number of vectors per segment, which is in our case equal to 32 as maximum. For the hardware decoder to be simple and fast, we need to address this memory bit-wise. This can be achieved by dividing the address into three fields:

1) Block #: this specifies the block to be decoded among the blocks of the current segment. The size of this field = $\lceil \log_2 K \rceil$; where K = the number of blocks per segment.
2) Row #: this indicates the row of the current block. The size of this field = $\log_2 N = 5$ as maximum (when N=32).

3) Column #: this indicates the column of the current block. The size of this field = $\log_2 N$ = 5 as maximum (when N=32).

The three counters *BCNT, RCNT*, and *CCNT* are used to decide the address of the bit to be written.

As we mentioned before, the maximum vector length is 8K. Therefore, the maximum memory size required = 32*8K = 256 Kbit. This needs an address of 18 bits. Since we have 5+5+10 = 20 bits in the three counters, we need to select the bits to represent the address in each case (*N*=8, 16, or 32) using multiplexers. The outputs of the multiplexers are connected directly to the memory address bus.

5.2.2. Implementation of the FSM

The FSM controlling the decoding process is shown in Figure 5.3. It consists of 62 states, which means that 6 FFs plus some combinational logic are enough to implement it. This FSM is designed to decode the whole test set, not only one segment or one block. The FSM can be summarized in the following:

(1) The decoding process is activated at state S_0 when a starting signal *start* = 1.

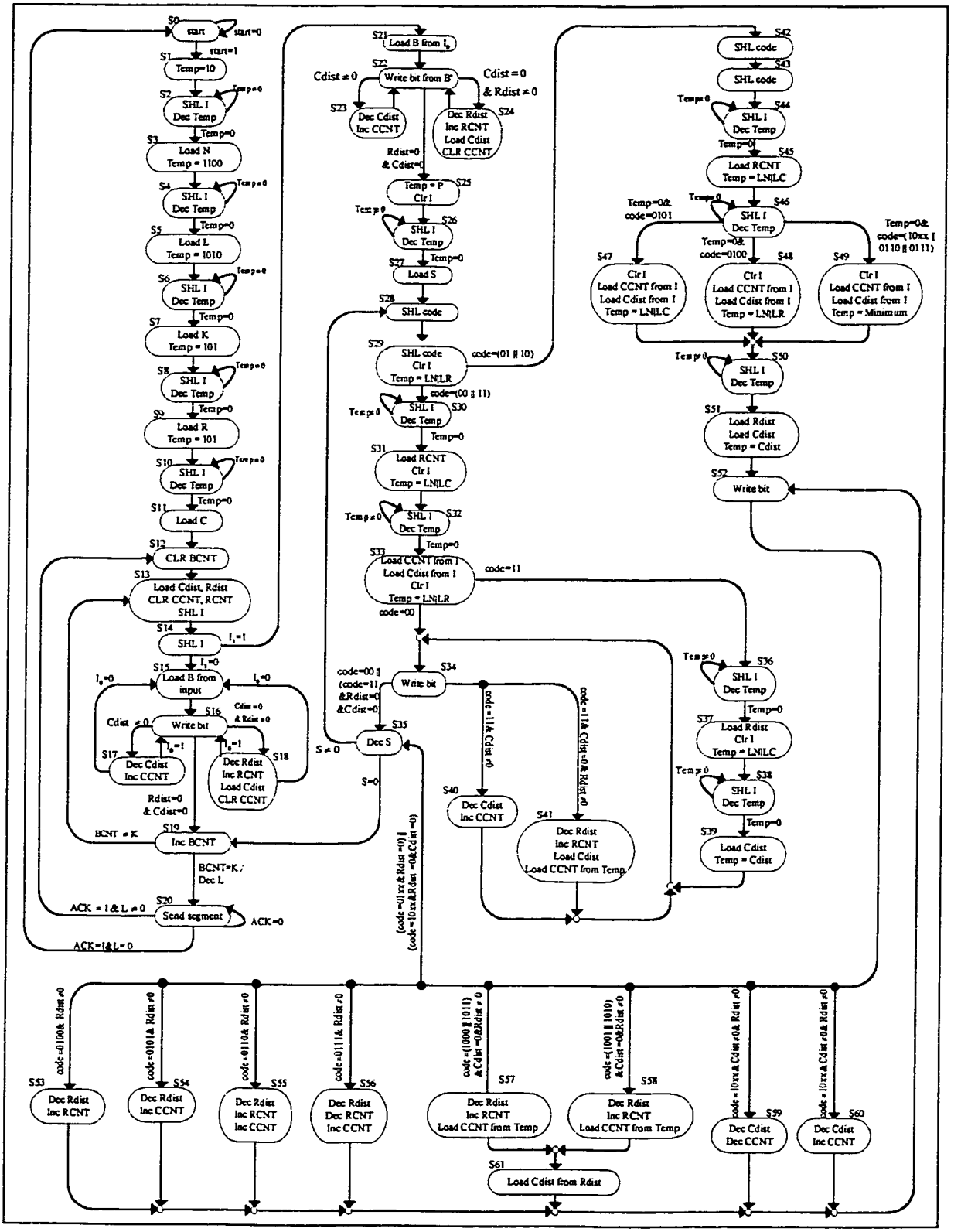


Figure 5.3. The FSM of the decoder

(2) From S_1 to S_{11}, the five parameters (block dimension (N), # of segments (L), # of blocks per segment (K), row remainder (R), and column remainder (C)) are read and loaded to the appropriate counters and registers. Then, the counters used for addressing the memory are initialized in states S_{12} and S_{13} and this is done for each block.

(3) In S_{14}, there are two possibilities:

(i) There are no shapes to be decoded: in this case the whole block will be filled with either 1 bit (0 or 1) or filled with real data. In the former case, S_{15} is visited only once to initialize the bit with which the block is filled and then the process goes through states S_{16}, S_{17} and S_{18}. In the latter case, S_{15} is visited for each bit read.

(ii) There are shapes to be decoded: in this case the process goes to state S_{21}.

(4) States S_{21} to S_{24} initialize the block with the complement of the bit with which all shapes are encoded. This is important to make sure that all bits in the block have the correct values. When the shapes are decoded, the corresponding bits will be overwritten.

(5) The number of shapes is read in states S_{25} to S_{27}.

(6) For each shape, S_{28} is visited to read the first bit in that shape.

(7) The type of the shape is determined in S_{29}. If it is a point or a rectangle, the process continues in S_{30}. Otherwise, it goes to S_{42}.

(8) States S_{32} and S_{33} initialize the first bit of the point and the rectangle shapes. If the shape is a point, only this bit is written in S_{34}. If, on the other hand, it is a rectangle, the process goes to states S_{36} to S_{39} to initialize the counters and then goes through S_{34}, S_{40} and S_{41} until the whole rectangle is written.

(9) If the shape is a line or a triangle, the process goes through states S_{42} to S_{51} to initialize the counters and determine the type of the shape. Then according to the type of the shape and the status of the counters, the process goes to one of the states S_{53} to S_{60}. Then, the process repeats until the shape is written.

(10) After every shape is decoded, the number of shapes is decremented in S_{35}. If there are other shapes, the process goes back to S_{28}. Otherwise, it goes to S_{19} in which the number of blocks (*BCNT*) is incremented. If there are still other blocks, the process goes to S_{13}; otherwise it goes to S_{20} if all blocks of the current segment have been decoded.

(11) In S_{20}, the segment just decoded is sent to the scan chains and the process waits for an acknowledgment to proceed. If there are other segments, the process goes to S_{12}. Otherwise, the process is terminated and goes back to the

initial state. This is the only case where a mealy output is required. Therefore, we can say that our FSM is almost Moore.

Because applying test vectors to scan chains needs a special kind of control, which is out of the scope of our work, we left that to another controller. The FSM will send a signal indicating that a segment is ready to be applied and then holds on until an acknowledgment reaches and then resumes.

5.3. Decoder Interface

In this section, we outline the interface between the decoder and the tester and between the decoder and the scan chains. First, we discuss the interface of the software decoder and then we discuss the interface of the hardware decoder.

5.3.1. Interface of the software decoder

Figure 5.4 shows the interface between the software decoder and the tester. The decoding program is stored in a ROM on chip. When the tester starts sending the encoded data to the processor, the processor reads the instructions from the ROM and executes them in order to decode the test data. Then, it writes the decoded data to the memory. After a whole segment is decoded, the processor will send a signal to the controller to start applying the test vectors to the scan chains. It should be stated here that there must be some synchronization mechanism between the processor and the tester in order to avoid overflow.

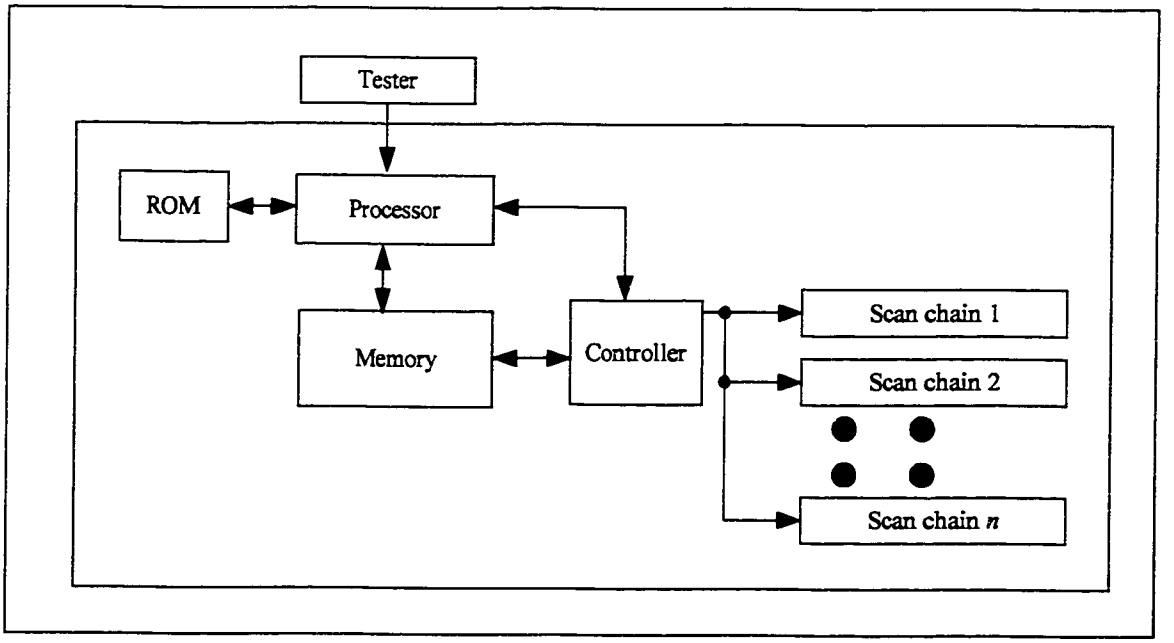


Figure 5.4. Interface of the software decoder

5.3.2. Interface of the hardware decoder

Similar to the interface explained above for the software decoder, the hardware decoder can be interfaced to the tester in place of the processor. This is shown in Figure 5.5. Here, the hardware decoder reads the encoded data from the tester and writes the decoded data to the memory. After decoding a complete segment, the decoder sends a signal to the controller to apply the test vectors and waits for the acknowledgement to start decoding another segment. Also here, we need some synchronization mechanism between the tester and the decoder.

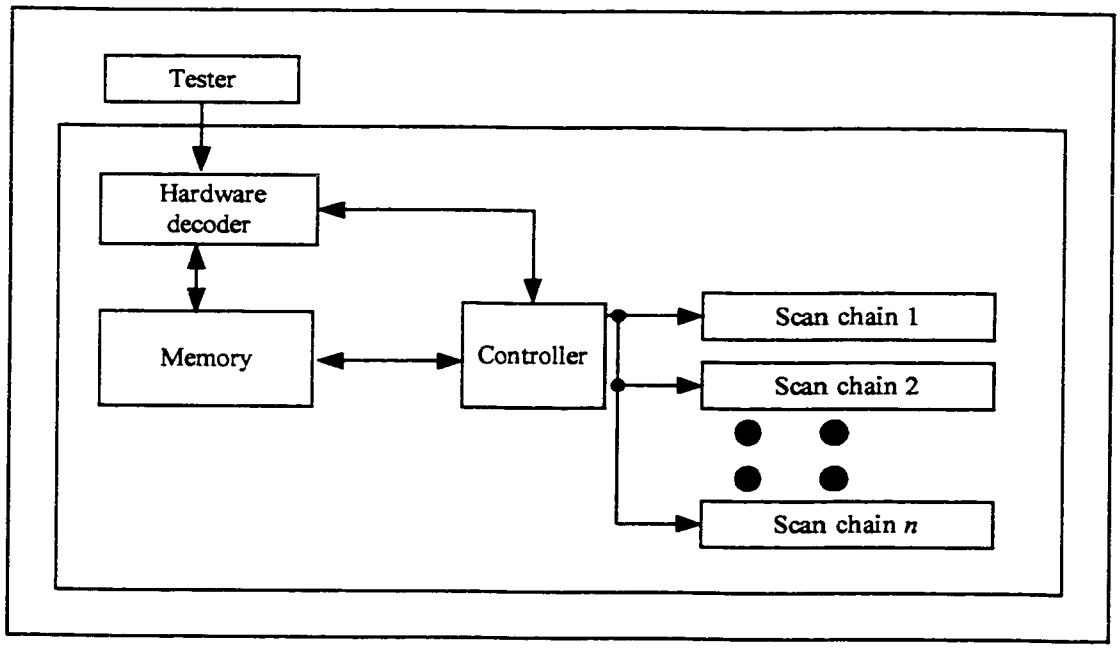


Figure 5.5. Interface of the hardware decoder

In both implementations of the decoder, the test application process and the decoding process can be performed in parallel if there is enough memory. In this case, the controller will read the decoded vectors from the memory and apply them to the scan chains. At the same time, the decoder will start decoding another segment and write it in another portion of the memory. This process requires more memory but it can speed up the testing time.

5.4. Concluding Remarks

In this chapter, we have discussed the implementation of the decompression process of our proposed technique. The decoder can be implemented in software or in hardware. The software decoder is executed using an embedded processor, which is available in most of the SOC's. This decoder runs in linear time with respect to the size of the test set.

If the embedded processor is not available on chip, then a hardware decoder is required. We have designed this decoder using an FSM of 62 states and a data path. The design has been modeled and verified using VHDL.

Both implementations of the decoder need some amount of temporary memory to store a segment of decoded blocks before applying its vectors to the circuit under test. This memory requirement and the complexity of the hardware decoder represent the overhead of our technique.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter, we show the results of our proposed compression scheme. In Section 6.1, we show the effect of different factors on the compression ratio. These factors are the x-weight, the type of sorting, the block size, the greedy sorting in comparison with the near-optimal sorting, and the size of the test sets. In Section 6.2, the timing performance of the encoder and the decoder of our technique is discussed. In Section 6.3, we give some statistics on block encoding that show the advantages of partitioning the test set into blocks and also show the possibility of improvement in future work. Finally, we compare the achieved results of our technique with some of the best results achieved in the literature in Section 6.4.

We ran our experiments on a number of the largest ISCAS85 and full-scanned versions of ISCAS89 benchmark circuits. The experiments have been run on a Pentium II processor with a speed of 350 MHz and a 32 Mbyte RAM. We have used

two test sets generated by MinTest [15], one is based on dynamic compaction and the other is based on static compaction. These test sets are highly compacted test sets that achieve 100% fault coverage of the detectable faults in each circuit. For the test sets generated by static compaction, test cubes were generated as this has the advantage of keeping unnecessary assignments as x's, which enables higher compression. To make sure that all results are correct, we have fault simulated the decoded test sets obtained from both the software and the hardware decoders.

6.1. Effect of Different Factors on Compression Ratio

There are a number of factors that affect the compression ratio resulting from our proposed scheme. These factors have been outlined in Chapter 4. In this section, we show the effect of these factors on the compression ratio. We assume a default value for each of these factors. This value is the one that gives the best results most of the time. These factors are:

1. The weight of the x bit in a test vector. The default value is x=0.25.
2. The type of sorting (0-sorting, 1-sorting or 0/1-sorting). The default value is the 0/1-sorting.

3. The block size (8x8, 16x16, or 32x32). The default size is 8x8.
4. The greedy sorting versus the near-optimal sorting. The default is greedy sorting.
5. The size of the test set. For this factor, we compared the same benchmark circuits with different sizes (different number of test vectors). The first set is generated using static compaction and the other is generated using dynamic compaction. The default is the set generated using dynamic compaction, which is larger in size.

In each experiment, we changed only one factor and fixed the other factors to the default. Doing so, we can illustrate the effect of that factor on the compression ratio. The compression ratio is calculated using the following formula:

$$Comp.\ Ratio = \frac{\#Original\ Bits - \#Compressed\ Bits}{\#Original\ Bits} \quad X100.$$

(1) The x-Weight

Table 6.1 and Figure 6.1 show the compression ratio for the three values of x-weight. It is clear from the figure that the best results are achieved when x=0.25 in most of the cases. The only exception in this set of circuits is for circuit s35932f in which the best result is for x=1.0. A possible reason for this exception is that most of

the shapes in this test set are of type rectangle or horizontal line. Therefore, if we treat the x's as the other bits, rectangles with maximum sizes may be formed. For all the cases, the compression ratio is moderate when x=0.5.

Circuit	x=0.25	x=0.5	x=1.0
s13207f	85.561	85.372	85.098
s15850f	70.188	69.239	66.971
s35932f	62.231	63.812	64.4
s38417f	62.226	60.229	61.72
s38584f	65.594	63.474	64.868
s5378f	57.94	52.936	53.006
s9234f	57.22	55.598	52.942
average	65.851429	64.38	64.143571

Table 6.1. Effect of x-weight

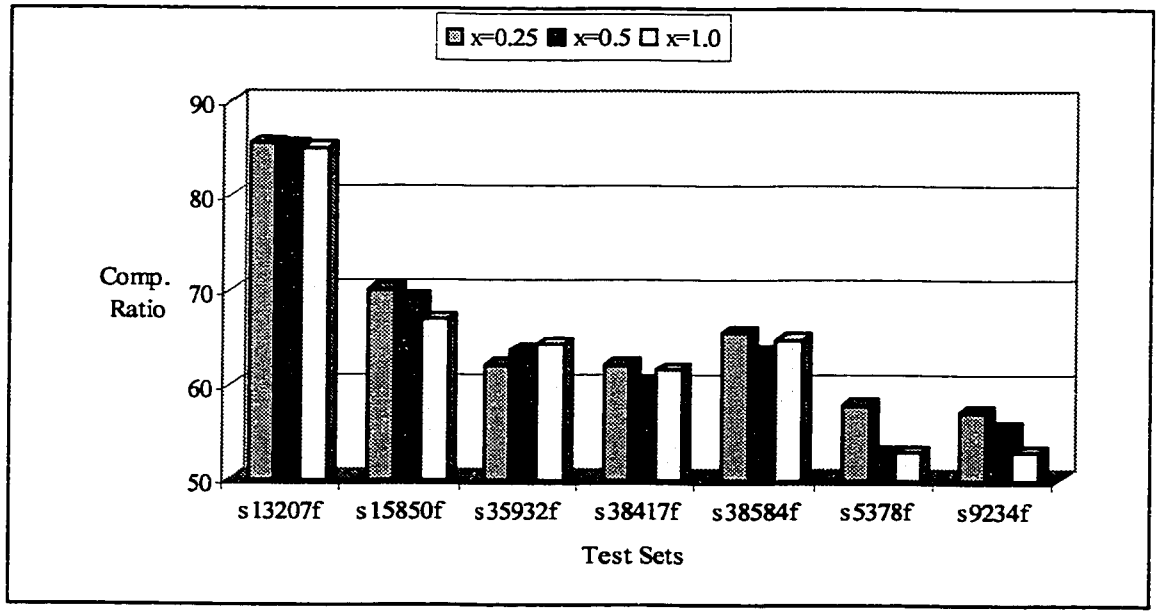


Figure 6.1. Effect of x-weight

(2) Type of Sorting

The 0/1-sorting gives the best results in most of the cases as shown in Table 6.2 and Figure 6.2. This is because our technique partitions the test set into blocks and has the option of encoding either the 0's or the 1's for each block separately. Therefore, it is better to have clusters of both the 0's and the 1's rather than having only clusters of 0's or clusters of 1's. Also in this case, the only exception is circuit s35932f. In this case, the 0-sorting achieves the best. It is interesting to notice that the average compression ratio for the three sorting criteria is almost the same.

Circuit	1-sorting	0-sorting	0/1-sorting
s13207f	84.952	84.724	85.561
s15850f	69.646	69.782	70.188
s35932f	65.177	65.889	62.231
s38417f	61.84	61.677	62.226
s38584f	65.203	65.186	65.594
s5378f	55.805	55.658	57.94
s9234f	54.989	55.921	57.22
average	65.373143	65.548143	65.851429

Table 6.2. Effect of type of sorting

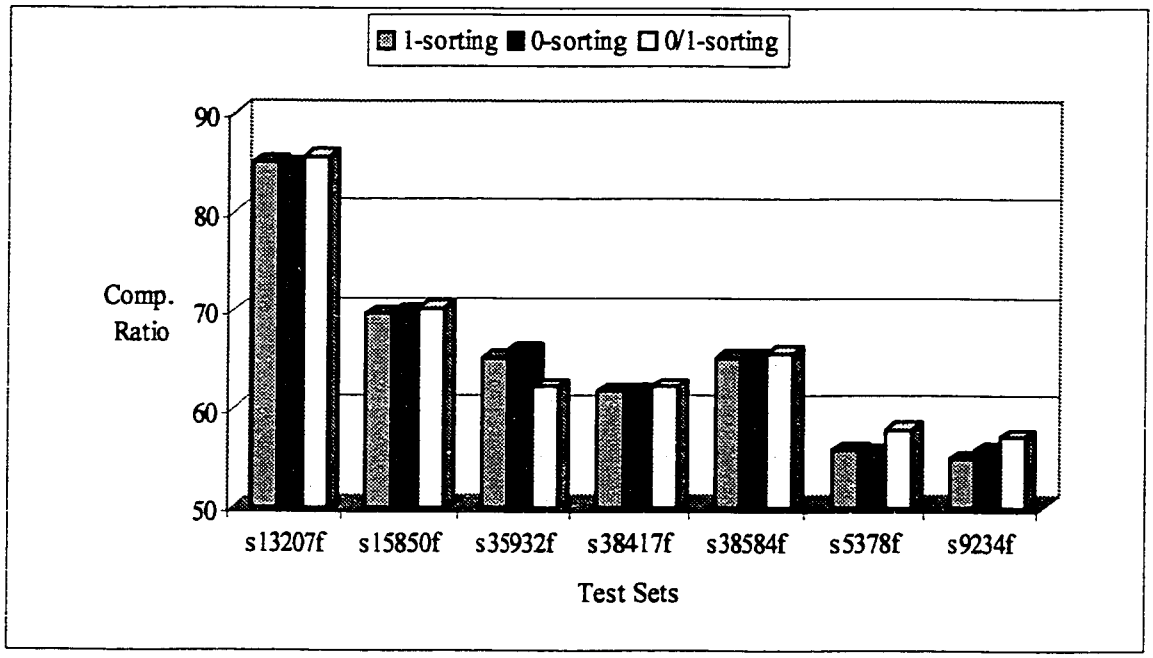


Figure 6.2. Effect of type of sorting

(3) Block Size

The block size has a high impact on the compression ratio. This is shown in Table 6.3 and Figure 6.3. In five of the seven cases, the 8x8 block size gives the best result. For circuit s35932f, the 32x32 block size is the best. The reason for this case is that the number of test vectors in this test set is 16, which means that all the blocks are partial blocks. In addition, most of the shapes in this test set are of type rectangle or horizontal line. This means that when the horizontal dimension increases, the cost of encoding these shapes decreases.

Circuit	8x8	16x16	32x32
s13207f	85.561	86.628	85.316
s15850f	70.188	69.253	65.776
s35932f	62.231	74.688	78.123
s38417f	62.226	59.304	54.245
s38584f	65.594	65.085	61.13
s5378f	57.94	52.854	48.657
s9234f	57.22	55.789	52.148
average	65.851429	66.228714	63.627857

Table 6.3. Effect of block size

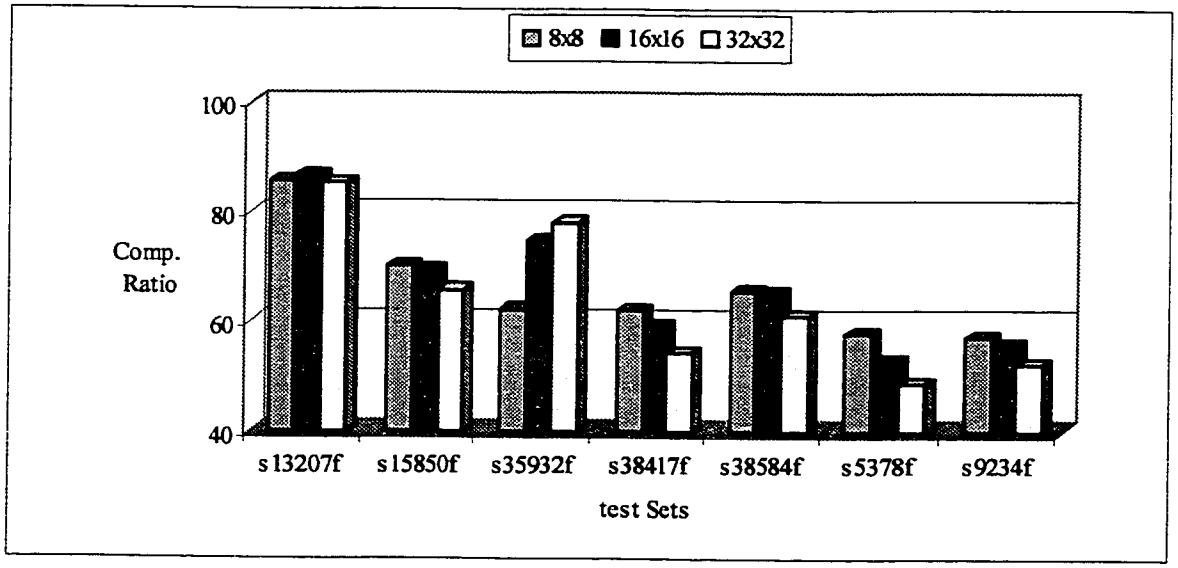


Figure 6.3. Effect of block size

(4) Greedy versus Near-Optimal

As explained in Chapter 4, in the greedy sorting, each time we select the closest vector to the last vector selected. We have tried to optimize the solution using the Hamiltonian cycle. However, we have found that the greedy solution is always better than the near-optimal solution. This is illustrated in Figure 6.4. The problem of this near-optimal solution is that it is based on finding a spanning tree and then finding a walk through the vertices of the tree (see [12] for details). It is possible by this solution that the walk groups vectors that are not highly correlated together which may take away some vectors that may have more correlation.

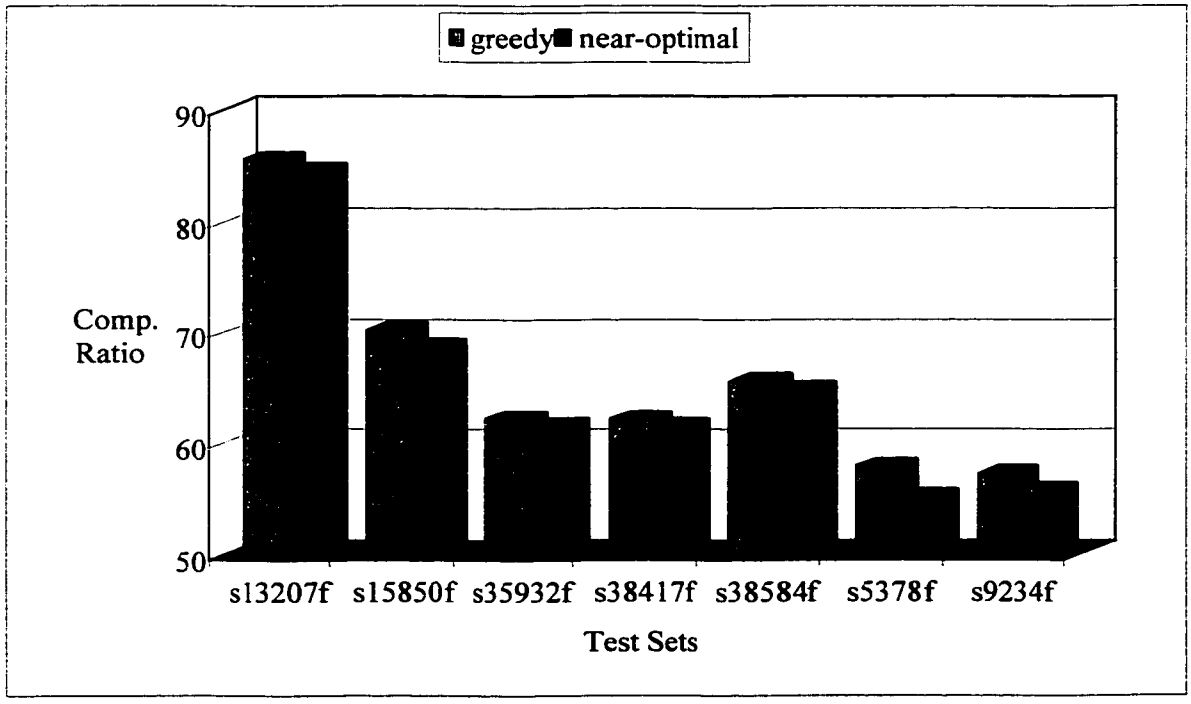


Figure 6.4. Greedy vs. near-optimal

(5) Size of the Test Set

The two test sets shown in Table 6.4 have different sizes. The first set is generated using a dynamic compaction technique based on MinTest [15] while the second is generated using a static compaction technique also based on MinTest [15]. The first set is larger in size than the second. The effect of the test set size cannot be shown if we consider only the compression ratio. However, if we look at the number of bits after compression, we can see that some of the circuits ended up with smaller number of bits although the size of the original test set is larger. These circuits are shaded in Table 6.4. From this observation, we can conclude that adding some redundancy to the test set may help in achieving higher compression.

	Test set 1			Test set 2		
Circuit	Original Bits	Comp. Ratio	Comp. Bits	Original Bits	Comp. Ratio	Comp. Bits
s5378	**23754**	**57.94**	**9991**	**20758**	**51.551**	**10057**
s9234	39273	57.22	16801	25935	43.451	14666
s13207	**165200**	**86.628**	**22091**	**163100**	**85.012**	**24445**
s15850	76986	70.188	22952	57434	60.32	22790
s35932	**28208**	**78.123**	**6171**	**21156**	**25.78**	**15702**
s38417	164736	62.226	62228	113152	46.497	60540
s38584	199104	65.594	68504	161040	65.944	54844

Table 6.4. Effect of test set size

6.2. Timing Performance

In this section, we show the timing performance of our technique. As we stated earlier, we performed our experiments on a Pentium II processor with a 32 Mbyte RAM.

(1) Performance of the Encoder

The encoder is implemented in software and it is run offline. Therefore, it can be more complicated than the decoder. Table 6.5 shows the time (in seconds) needed to encode each circuit for different block sizes. The last column gives the total time if all three sizes are tried to choose the best among them. This timing includes the sorting step. Since the block size has a high impact on the compression ratio and because the encoder is fast, all three sizes can be used for encoding and the one that gives the best results is selected.

Circuit	8x8	16x16	32x32	Total
s13207f	6	8	15	29
s15850f	2	2	6	10
s35932f	2	2	1	5
s38417f	8	11	35	54
s38584f	8	9	16	33
s5378f	1	1	2	4
s9234f	2	2	10	14
average	4.1428571	5	12.142857	21.285714

Table 6.5. Timing of the encoder

(2) Performance of the Decoder

The decoder can be implemented in software or hardware. In our experiments, we have found that the software decoder of our technique is very fast and the time needed for it to finish decoding a whole test set is very small and can be neglected.

For the hardware decoder, we have counted the clock cycles needed to complete decoding each circuit using the VHDL model of the hardware decoder described in Chapter 5. If we assume a certain clock rate, then we can find the time required by the decoder by dividing the number of clock cycles by the clock rate. Table 6.6 shows the results for a clock rate of 500 MHz and for different block sizes. The time given in the table is in μ seconds. We should indicate here that this timing is for the decoding process only and does not include the test application time.

circuit						
s13207f	366506	733.012	373039	746.078	406249	812.498
s15850f	191193	382.386	200540	401.08	220739	441.478
s35932f	83120	166.24	80880	161.76	77967	155.934
s38417f	438157	876.314	464892	929.784	516920	1033.84
s38584f	509046	1018.092	542300	1084.6	562884	1125.768
s5378f	61748	123.496	67458	134.916	72380	144.76
s9234f	107482	214.964	113818	227.636	118152	236.304

Table 6.6. Timing of the hardware decoder

If we look at Table 6.6 and Table 6.3, we can notice that the number of clock cycles needed to decode a test set increases with the decrease in the compression ratio. The only exception in this trend is in the case of circuit s13207f, where the highest compression ratio is for the 16x16 block size while the smallest number of clock cycles is for the 8x8 block size. The reason for this is that the compression ratios for the two block sizes are very close to each other while the percentage of real-data blocks is higher for the case of 16x16 block size (as will be shown in the following section). Since the real-data blocks need more time for decoding (because they require more reading cycles), the number of clock cycles increases with the increase in the percentage of real-data blocks.

6.3. Statistics on Block Encoding

As explained in Chapter 4, there are three possibilities for encoding a block. The first is to encode the block as filled by either 0's or 1's. The second is to encode the block using geometric shapes. The third is to store the real data if the number of bits needed to encode the block is greater than the actual number of bits in that block. We call the first type of blocks *filled blocks*, the second type of blocks *encoded*

blocks and the last type of blocks *real-data blocks*. The cost of each filled block is only 3 bits, while the cost of each real-data block is the size of the block + 2. The cost of an encoded block depends on the shapes in that block. In this section, we show the percentage of each type of blocks to the total number of blocks for the benchmark circuits used.

Table 6.7, Table 6.8, and Table 6.9 show the percentage of these types of blocks for the benchmark circuits used for block size 8x8, 16,16, and 32x32, respectively. From these tables, we can notice the following:

1) The percentage of filled blocks decreases with the increase in block size while the percentage of real-data blocks does change much. This shows why the 8x8 block size gives the best results most of the time followed by the 16x16 block size. From this point, we can notice the advantage of partitioning the test set into blocks.
2) Some of the circuits have high percentage of real-data blocks. This shows that there is a room for improvement if these blocks are encoded using another compression scheme.

	8x8						
circuit	total # of blocks	# of real blocks	%	# of filled blocks	%	# of encoded blocks	%
s13207f	2640	68	2.5758	2041	77.31061	531	20.113636
s15850f	1232	82	6.6558	614	49.83766	536	43.506494
s35932f	442	2	0.4525	56	12.66968	384	86.877828
s38417f	2704	189	6.9896	1068	39.49704	1447	53.513314
s38584f	3111	347	11.154	1180	37.92993	1584	50.916104
s5378f	378	78	20.635	143	37.83069	157	41.534392
s9234f	620	60	9.6774	150	24.19355	410	66.129032
average			8.3057		39.89559		51.798686

Table 6.7. Statistics on block encoding (8x8 blocks)

	16x16						
circuit	total # of blocks	# of real blocks	%	# of filled blocks	%	# of encoded blocks	%
s13207f	660	22	3.3333	356	53.93939	282	42.727273
s15850f	312	27	8.6538	75	24.03846	210	67.307692
s35932f	111	1	0.9009	0	0	110	99.099099
s38417f	728	41	5.6319	162	22.25275	525	72.115385
s38584f	828	98	11.836	145	17.51208	585	70.652174
s5378f	98	17	17.347	13	13.26531	68	69.387755
s9234f	160	17	10.625	7	4.375	136	85
average			8.3325		19.34043		72.327054

Table 6.8. Statistics on block encoding (16x16 blocks)

	32x32						
circuit	total # of blocks	# of real blocks	%	# of filled blocks	%	# of encoded blocks	%
s13207f	176	0	0	44	25	132	75
s15850f	80	6	7.5	3	3.75	71	88.75
s35932f	56	1	1.7857	0	0	55	98.214286
s38417f	208	2	0.9615	31	14.90385	175	84.134615
s38584f	230	44	19.13	21	9.130435	165	71.73913
s5378f	28	5	17.857	2	7.142857	21	75
s9234f	40	7	17.5	0	0	33	82.5
average			9.2478		8.56102		82.191147

Table 6.9. Statistics on block encoding (32x32 blocks)

6.4. Comparison with Other Techniques

In this section, we compare the results of our technique with some of the best results published in the literature. These are the techniques proposed in [8] and [9], which are the most recent techniques proposed in the literature. The first technique uses the variable-to-variable run length coding using the FDR codes while the second uses the variable-to-variable run length coding using Golomb codes. Table 6.10 and Figure 6.5 show the results of each of the three techniques. The first column (Geometric) shows our results using the default factors (explained in Section 6.1 above) except

for the block size where the best among the three block sizes (8x8, 16x16, or 32x32) is selected. The other two columns show the compression ratio achieved by FDR code and Golomb code when applied to the original test sets. It should be stated here that the test sets used for comparison are the same, which are the ones obtained by dynamic compaction by MinTest.

In all cases, we got significantly higher compression ratio than the other techniques. The average percentage of compression is shown in the last row of Table 6.10. One of the interesting circuits is s35932f, where Golomb codes did not achieve any compression and the FDR codes only got 19.37% compression; while our technique achieved 78.12% compression. This is because these techniques depend on encoding runs of 0's followed by a 1. The cost of encoding runs of 1's is more than the cost of the runs themselves because each 1 is encoded as a run of 0's of length 0 followed by a 1. Since circuit s35932f has many runs of 1's (which are sometimes very long), these techniques do not perform well for encoding this circuit.

circuit	Geometric	FDR	Golomb
s5378f	57.94	48.02	37.11
s9234f	57.22	43.59	45.25
s13207f	86.628	81.3	79.74
s15850f	70.188	66.23	62.82
s35932f	78.123	19.37	0
s38417f	62.226	43.26	28.37
s38584f	65.594	60.91	57.17
average	68.27414	51.81143	44.35143

Table 6.10. Comparison with other techniques

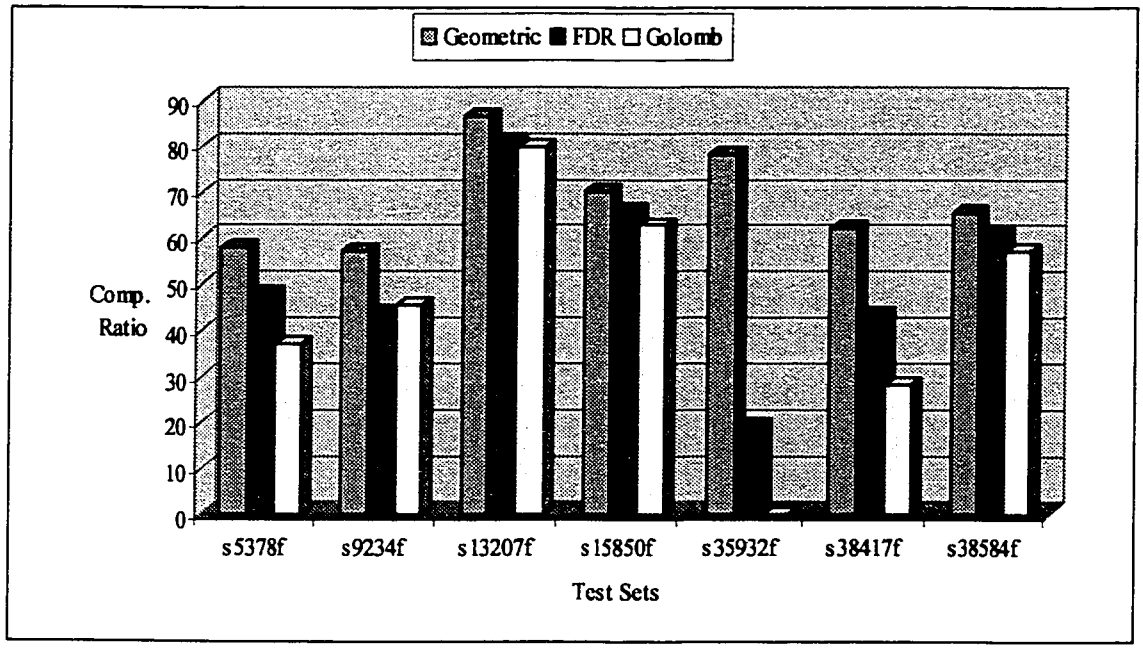


Figure 6.5. Comparison with other techniques

6.5. Concluding Remarks

In this chapter, we have presented the experimental results of our proposed technique. First we have shown the effect of different factors on the compression ratio. These factors are the x-weight, the type of sorting, the block size, the greedy sorting, and the size of the test set.

Then, we have discussed some statistics on block encoding. These statistics show the advantage of the partitioning step and the impact of the block size. In addition, they show that there is a room for improvement on our technique if the real-data blocks (those blocks that are not encoded) are exploited.

After that, we have discussed the timing performance of the encoder and the decoder. It has been shown that the encoder is fast and hence it is possible to try all block sizes and select the one that gives the best compression ratio. The software decoder runs in negligible time. For hardware decoder, we used the VHDL model to compute the number of clock cycles required to decode the test data. This also shows that the hardware decoder is fast.

To show the effectiveness of our technique, we have compared the achieved compression ratio with those of the most recent techniques proposed in the literature. These are the techniques that use Golomb codes and FDR codes for encoding the test data. To be fair in comparison, we used the same test sets used by these techniques. It has been shown by this comparison that our technique outperforms the others in all the cases.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Systems-on-a-chip (SOC) design is popular nowadays and it is evolving very rapidly. Testing SOC requires a huge amount of test data, which increases testing time. It is, therefore, very desirable to reduce this time by reducing the amount of test data that must be transferred to the chip under test.

Two solutions are possible to reduce test data. The first is *test set compaction* in which the number of test vectors is reduced while the fault coverage is maintained. There are two kinds of compaction, static and dynamic. In static compaction, the test set is reduced after it has been generated. In dynamic compaction, the reduction is performed during the test generation process.

The other solution is *test data compression*. In this case, test data is represented in a different format that requires less number of bits. This is called *encoding* the test data. It is then transferred to the chip under test and decoded there to restore the original test data. The compression must be lossless. Some of the compression

techniques are based on BIST and PRGs, where the internal architecture of the chip under test is designed to have self-testing capability. Other techniques are based on deterministic testing, regardless of the internal architecture. The scope of this thesis is in the latter case.

In Chapter 3, a review of the compaction and compression techniques proposed in the literature has been discussed. Most of the compression techniques are based on one-dimensional approaches, where vectors are encoded serially. Furthermore, most of these techniques are based on variations of some of the basic compression schemes such as run-length and statistical coding for encoding test data.

In this thesis, we have proposed a novel technique for encoding test data. This technique is a two-dimensional approach in which the test set is partitioned into equal-size blocks, each of which is encoded separately and independently. The encoding of test data in each block is based on geometric shapes, which exploits the high correlation between test vectors.

The encoding process can be summarized as follows. First, the test set is sorted in order to group the test data into the minimum number of shapes. The sorting procedure we used in this thesis is greedy, where a test vector is selected each time based on a *distance* function. The sorting can be done with respect to the 0 bits (0-

sorting), the 1 bits (1-sorting,) or to both 0's and 1's (0/1-sorting). Second, the test set is partitioned into equal-size blocks. The block size has very high impact on the compression ratio. We have experimented with three block sizes, 8x8, 16x16, and 32x32. After partitioning the test set, each block is encoded individually. There are three possibilities for encoding a block. The first one is to fill the whole block by either 0's or 1's. This case is the least costly because it requires only three bits. The second possibility is to encode the block by geometric shapes that cover either the 0 bits or the 1 bits. The third possibility is to store the real test data if its cost is less than the cost of encoding the block. It has been shown that the sorting algorithm runs in $O(VT^2)$ time, where T is the number of vectors and V is the vector length. The encoding algorithm requires O(VT) time, which is linear with respect to the size of the test set. The discussion of the sorting and the encoding processes has been presented in Chapter 4.

An important step of any compression technique is the design of the decoder. The decoder should be simple enough because it is run on-chip. There are two possibilities for implementing the decoder, either in software or in hardware. In software decoder, a simple procedure is run using an embedded processor on-chip, which is available in most of the SOCs nowadays. In hardware decoder, some additional hardware is added to the chip to perform the decoding process. We have implemented the software decoder using C++ code. The design for the hardware

decoder has been modeled and verified using VHDL. The implementation of the decoding process has been discussed in Chapter 5. One limitation of our decoder is the requirement of some amount of memory to store a segment of decoded blocks before the test vectors of that segment are applied to the circuit under test.

In Chapter 6, we have discussed the experimental results of our technique. First, we have shown the effect of different factors on the compression. These factors are the x-weight, the type of sorting, the size of the block, the greedy sorting and the size of the test set. Each factor has a default value that gives the best results most of the time. Then, we have discussed some statistics on the encoding process that show the advantage of the partitioning step and the possibilities for improvement. After that, the timing of the encoder and the decoders have been discussed. It has been shown that this time is very small for both the encoder and the two implementations of the decoder. Finally, we have compared our results to those of the most recent techniques proposed in the literature. This comparison shows the effectiveness of our proposed technique since we achieved significantly higher compression ratio than the others in all the cases.

Future Work

The future direction can be summarized in the following:

- Since sorting of test vectors highly affects the compression ratio, it is desirable to find better sorting criteria or approaches that may increase the compression ratio.
- We have shown that there are some blocks that are not encoded because the cost of their encoding is more than the cost of the real data. Hence, it is possible to combine our technique with some other techniques to produce a *hybrid* compression scheme, which may exploit these real-data blocks. This is possible because each block in our technique is encoded independently. A possible candidate is the FDR coding scheme proposed in [8].

APPENDEX A

VHDL CODE FOR THE HARDWARE DECODER

-- THESE ARE THE ENTITIES FOR THE COMPONENTS USED IN THE DATA

-- PATH (FUNCTINAL MODELING)

```
Entity xnor2 is
     generic(N:positive:=4;delay:time:=2 ns);
     port(A,B:in bit_vector(N-1 downto 0); O:out bit);
End xnor2;

Architecture behav of xnor2 is
Begin
     process(A,B)
          variable R:bit;
      begin
          R:='1';
          for i in 0 to N-1 loop
          R :=R and not(A(i) xor B(i));
         end loop;
         O <= R after delay;
      end process;
End behav;
-----------------------------------------------
Entity and2 is
     port(A,B:in bit; O:out bit);
End and2;

Architecture behav of and2 is
  Begin
     O <= A and B after 3 ns;
  End behav;
-----------------------------------------------
Entity and3 is
     port(A,B,C:in bit; O:out bit);
End and3;

Architecture behav of and3 is
  Begin
     O <= A and B and C after 3 ns;
  End behav;
-----------------------------------------------
```

```
Entity or2 is
     port(A,B:in bit; O:out bit);
End or2;

Architecture behav of or2 is
  Begin
     O <= A or B after 3 ns;
  End behav;
-----------------------------------------------
Entity xnor2_1 is
     port(A,B:in bit; O:out bit);
End xnor2_1;

Architecture behav of xnor2_1 is
  Begin
     O <= not(A xor B) after 4 ns;
  End behav;
-----------------------------------------------
Entity inv is
     port(A:in bit; O:out bit);
End inv;

Architecture behav of inv is
  Begin
     O <= not A after 1 ns;
  End behav;
-----------------------------------------------
Entity norN is
     generic(N:positive:=4;delay:time:=2 ns);
     port(A:in bit_vector(N-1 downto 0); O:out bit);
End norN;

Architecture behav of norN is
Begin
     process(A)
          variable R:bit;
      begin
          R:='0';
          for i in 0 to N-1 loop
               R :=R or A(i) ;
          end loop;
          O <= not R after delay;
```

```
      end process;
End behav;
-----------------------------------------------
Entity MUX2_1 is
     generic(delay:time:=4 ns);
     port(A,B:in bit; sel:in bit; O:out bit);
End MUX2_1;

Architecture behave of MUX2_1 is
     Begin
     with sel select
          O <= A after delay when '0',
               B after delay when '1';
   end behave;
-----------------------------------------------
Entity MUX2 is
     generic(N:positive:=4; delay:time:=4 ns);
   port(A,B:in bit_vector(N-1 downto 0); sel:in bit;
       O:out bit_vector(N-1 downto 0));
End MUX2;

Architecture behave of MUX2 is
     Begin
     with sel select
          O <= A after delay when '0',
               B after delay when '1';
   end behave;
-----------------------------------------------
Entity MUX4 is
     generic(N:positive:=4; delay:time:=4 ns);
     port(A,B,C,D:in bit_vector(N-1 downto 0); sel:in
          bit_vector(1 downto 0);
          O:out bit_vector(N-1 downto 0));
End MUX4;

Architecture behave of MUX4 is
     Begin
     with sel select
          O <= A after delay when "00",
               B after delay when "01",
               C after delay when "10",
               D after delay when "11";
```

```
   end behave;
-----------------------------------------------
Entity MUX8 is
     generic(N:positive:=4; delay:time:=4 ns);
     port(A,B,C,D,E,F,G,H:in bit_vector(N-1 downto 0);
          sel:in bit_vector(2 downto 0);
          O:out bit_vector(N-1 downto 0));
End MUX8;

Architecture behave of MUX8 is
     Begin
     with sel select
          O <= A after delay when "000",
               B after delay when "001",
               C after delay when "010",
               D after delay when "011",
               E after delay when "100",
               F after delay when "101",
               G after delay when "110",
               H after delay when "111";
   end behave;
-----------------------------------------------
Entity DFF is
     generic(delay:time:=2 ns);
     port(D,clk:in bit; Q,Q_bar:out bit);
End DFF;

Architecture behav of DFF is
Begin
     process(D,clk)
      begin
          if(clk='1' and clk'event) then
               Q <= D after delay;
               Q_bar <= not D after delay;
         end if;
      end process;
End behav;
-----------------------------------------------
Entity Reg is
     generic(N:positive:=4; delay:time:=4 ns);
     port(A:in bit_vector(N-1 downto 0); Load,clk:in bit;
          O:out bit_vector(N-1 downto 0));
```

```
End Reg;

Architecture behave of Reg is
     Begin
     process(clk,Load)
          begin
          if(clk='0' and clk'event) then
               if(Load='1') then
                    O <= A after delay;
               end if;
            end if;
         end process;
   end behave;
-----------------------------------------------
Entity Sh_Reg is
          generic(N:positive:=4; delay:time:=4 ns);
          port(A,SHL,clr,clk:in bit; O:out bit_vector(N-1
               downto 0));
End Sh_Reg;

Architecture behave of Sh_Reg is
     Begin
     process(clk,SHL,clr)
               variable R:bit_vector(N-1 downto 0);
          begin
          if(clk='0' and clk'event) then
               if(clr='1') then
               for i in N-1 downto 0 loop
                    R(i):= '0';
                end loop;
                    O <= R after delay;
               elsif(SHL='1') then
               for i in N-1 downto 1 loop
                    R(i):= R(i-1);
                end loop;
                  R(0):=A;
                  O <= R after delay;
               end if;
            end if;
         end process;
   end behave;
-----------------------------------------------
```

```
Entity counter is
     generic(N:positive:=4; delay:time:=4 ns);
     port(A:in bit_vector(N-1 downto 0);
          Load,clr,inc,dec,clk:in bit;
          O:out bit_vector(N-1 downto 0));
End counter;

Architecture behave of counter is
     Function int_to_bin(int:integer) return bit_vector is
     variable temp:integer;
      variable bin:bit_vector(N-1 downto 0);
      begin
          temp := int;
          for i in 0 to N-1 loop
          if(temp mod 2=1) then
               bin(i) := '1';
            else
               bin(i) := '0';
            end if;
            temp := temp/2;
         end loop;
          return(bin);
      end int_to_bin;

   Function bin_to_int(bin:bit_vector) return integer is
     variable int:integer:=0;
      begin
          for i in 0 to bin'length-1 loop
          if(bin(i)='1') then
               int := int+2**i;
            end if;
         end loop;
          return(int);
      end bin_to_int;

     Begin
     process(clk,Load,clr,dec,inc)
               variable R:integer;
          begin
          if(clk='0' and clk'event) then
               if(Load='1') then
                    R := bin_to_int(A);
```

```
                elsif(clr='1') then
                      R := 0;
                elsif(dec='1') then
                      R := R-1;
                elsif(inc='1') then
                      R := R+1;
                end if;
              end if;
              O <= int_to_bin(R) after delay;
          end process;
    end behave;
-----------------------------------------------
Entity Decrementer is
      generic(N:positive:=4; delay:time:=4 ns);
      port(A:in bit_vector(N-1 downto 0); O:out
           bit_vector(N-1 downto 0));
End Decrementer;

Architecture behave of Decrementer is
      Function int_to_bin(int:integer) return bit_vector is
      variable temp:integer;
       variable bin:bit_vector(N-1 downto 0);
       begin
           temp := int;
           for i in 0 to N-1 loop
           if(temp mod 2=1) then
                bin(i) := '1';
             else
                bin(i) := '0';
             end if;
             temp := temp/2;
          end loop;
           return(bin);
       end int_to_bin;

    Function bin_to_int(bin:bit_vector) return integer is
      variable int:integer:=0;
       begin
           for i in 0 to bin'length-1 loop
           if(bin(i)='1') then
                int := int+2**i;
             end if;
```

```
          end loop;
           return(int);
       end bin_to_int;

      Begin
      process(A)
                variable R:integer;
           begin
                R := bin_to_int(A);
                R := R-1;
                O <= int_to_bin(R) after delay;
       end process;
end behave;
-----------------------------------------------
Entity memory is
     generic(Nb:positive:=3;N:positive:=5; delay:time:=4
             ns; per:time:=40 ns);
     port(blc_size:in bit_vector(N-1 downto 0);
          Num_Blc,Blc:in bit_vector(Nb-1 downto 0);
          Row,Col:in bit_vector(N-1 downto 0);
          clk,R,W:in bit; B:in bit; ack:out bit);
End memory;

use STD.TEXTIO.all;

Architecture behave of memory is
   Function bin_to_int(bin:bit_vector) return integer is
     variable int:integer:=0;
      begin
          for i in 0 to bin'length-1 loop
           if(bin(i)='1') then
               int := int+2**i;
           end if;
         end loop;
           return(int);
      end bin_to_int;

     signal M1,v1: bit;

Begin
     process(clk,R,W,Num_Blc,blc_size)
```

```
        type mem is array(0 to 2**Nb-1,0 to 2**N-1,0 to
                          2**N-1)of bit;
    variable M:mem;
    variable vector: line;
    variable v:bit;
    file outfile:text is out "output.out";
    begin
        assert not(R='1' and W='1')
        report("ILLIGAL: READING AND WRITING AT THE
               SAME TIME")
       severity error;
       if(clk='0' and clk'event) then
        if(W='1') then
             M(bin_to_int(Blc),bin_to_int(Row),bin_to_in
             t(Col)) := B;
             M1 <=
                  M(bin_to_int(Blc),bin_to_int(Row),bin
                  _to_int(Col));
         elsif(R='1') then
             for i in 0 to bin_to_int(blc_size) loop
                  for j in 1 to bin_to_int(Num_Blc) loop
                     for k in 0 to bin_to_int(blc_size)
                       loop
                      v:=M(j-1,i,k);
                      v1 <= M(j-1,i,k);
                      write(vector,v);
                    end loop;
                end loop;
                writeline(outfile,vector);
             end loop;
             ack <= '1' after delay, '0' after per;
          end if;
       end if;
    end process;
End behave;
-----------------------------------------------
-----------------------------------------------
```

```
-- THIS IS THE DATA PATH ENTITY (STRUCTURAL MODELING)

Entity DataPath is
     port(input,clk,SHL_code,Clr_I,SHL_I,I0orINPUT,Load_B,
          BorB_bar,Load_S,Dec_S,
          Load_L,Dec_L,Load_K,Clr_BCNT,Inc_BCNT,Load_N,
          TemporI,Load_R,Load_C,Clr_CCNT,Load_CCNT,
          Inc_CCNT,Dec_CCNT,Clr_RCNT,Load_RCNT,Inc_RCNT,
          Dec_RCNT,FULLorDIST,RdistorI,Load_Cdist,
          Dec_Cdist,Load_Rdist,Dec_Rdist,LRorLC,Load_temp,
          Dec_temp,W,Rd:in bit;
          Dir,L_1,BCNT_K_1:inout bit;
          temp_sel:in bit_vector(2 downto 0);code:out
          bit_vector(3 downto 0);
          I1,I0,S_0,L_0,BCNT_K,Rdist_0,Cdist_0,temp_0,
          ack:out bit);
End DataPath;

Architecture struct of DataPath is

-- COMPONENET DECLARATION

   component xnor2
      generic(N:positive;delay:time);
      port(A,B:in bit_vector(N-1 downto 0); O:out bit);
   End component;

     component and2
          port(A,B:in bit; O:out bit);
   End component;

     component and3
          port(A,B,C:in bit; O:out bit);
   End component;

     component or2
          port(A,B:in bit; O:out bit);
   End component;

     component xnor2_1
          port(A,B:in bit; O:out bit);
```

```
End component;

  component inv
       port(A:in bit; O:out bit);
End component;

component norN
       generic(N:positive;delay:time);
       port(A:in bit_vector(N-1 downto 0); O:out bit);
  End component;

component MUX2_1
       generic(delay:time);
       port(A,B:in bit; sel:in bit; O:out bit);
  End component;

component MUX2
  generic(N:positive; delay:time);
  port(A,B:in bit_vector(N-1 downto 0); sel:in bit;
            O:out bit_vector(N-1 downto 0));
End component;

  component MUX4
       generic(N:positive; delay:time);
       port(A,B,C,D:in bit_vector(N-1 downto 0); sel:in
            bit_vector(1 downto 0);
            O:out bit_vector(N-1 downto 0));
  End component;

  component MUX8
       generic(N:positive; delay:time);
       port(A,B,C,D,E,F,G,H:in bit_vector(N-1 downto
            0); sel:in bit_vector(2 downto 0);
            O:out bit_vector(N-1 downto 0));
  End component;

  component DFF
       generic(delay:time);
       port(D,clk:in bit; Q,Q_bar:out bit);
  End component;

component Reg
```

```
        generic(N:positive; delay:time);
    port(A:in bit_vector(N-1 downto 0); Load,clk:in bit;
           O:out bit_vector(N-1 downto 0));
    End component;

    component Sh_Reg
        generic(N:positive; delay:time);
        port(A,SHL,clr,clk:in bit; O:out bit_vector(N-1
            downto 0));
    End component;

    component counter
        generic(N:positive; delay:time);
        port(A:in bit_vector(N-1 downto 0);
            Load,clr,inc,dec,clk:in bit;
            O:out bit_vector(N-1 downto 0));
    End component;

  component Decrementer
    generic(N:positive; delay:time);
    port(A:in bit_vector(N-1 downto 0); O:out
        bit_vector(N-1 downto 0));
  End component;

    component memory
        generic(Nb,N:positive; delay,per:time);
        port(blc_size:in bit_vector(N-1 downto 0);
            Num_Blc,Blc:in bit_vector(Nb-1 downto 0);
            Row,Col:in bit_vector(N-1 downto 0);
            clk,R,W:in bit; B:in bit; ack:out bit);
    End component;

-- COMPONENT INSTANTIATION
  For all:xnor2 use entity work.xnor2(behav);
  For all:and2 use entity work.and2(behav);
  For all:and3 use entity work.and3(behav);
  For all:or2 use entity work.or2(behav);
  For all:xnor2_1 use entity work.xnor2_1(behav);
  For all:inv use entity work.inv(behav);
  For all:norN use entity work.norN(behav);
  For all:MUX2_1 use entity work.MUX2_1(behave);
  For all:MUX2 use entity work.MUX2(behave);
```

```
For all:MUX4 use entity work.MUX4(behave);
For all:MUX8 use entity work.MUX8(behave);
For all:DFF use entity work.DFF(behav);
For all:Reg use entity work.Reg(behave);
For all:Sh_Reg use entity work.Sh_Reg(behave);
For all:counter use entity work.counter(behave);
For all:Decrementer use entity
       work.Decrementer(behave);
For all:memory use entity work.memory(behave);

signal Txnor,Tnor,T_ff:time:=4 ns;
signal Tmux,Tdec:time:=6 ns;
signal Treg,Tcount:time:=8 ns;
signal per:time:=200 ns;
signal Tmem:time:=10 ns;
signal one:positive:=1;
signal two:positive:=2;

signal three:positive:=3;
signal four:positive:=4;
signal five:positive:=5;
signal seven:positive:=7;
signal eight:positive:=8;
signal ten:positive:=10;
signal twelve:positive:=12;

signal clk_bar,DB,clk_B,B,B_bar,Dir_bar,L0_bar:bit;
signal N3_bar,R4_bar,R3_bar,R1_bar,R2orR1,R2orR1_b:bit;
signal C4_bar,C3_bar,C1_bar,C2orC1,C2orC1_b,Cand1:bit;
signal LNLR0_b,LNLR1_b,LNLR2_b,MINor1,MINor2:bit;
signal Rand1,MINand2,CCNT_Dec,B_Bbar,MINand1:bit;
signal high:bit:='1';
signal nul:bit:='0';
signal bin_2:bit_vector(4 downto 0):=B"00010";
signal bin_5:bit_vector(4 downto 0):=B"00101";
signal bin_10:bit_vector(4 downto 0):=B"01010";
signal bin_12:bit_vector(4 downto 0):=B"01100";
signal I,L,Lin:bit_vector(11 downto 0);
signal S:bit_vector(6 downto 0);
signal K,K1,BCNT:bit_vector(9 downto 0);
signal temp,N,R,C, NR,NC:bit_vector(4 downto 0);
signal CCNT,RCNT,Cdist,Rdist:bit_vector(4 downto 0);
```

```
    signal Nin,Nout:bit_vector(1 downto 0);
    signal RdistI,TempI,TempIn:bit_vector(4 downto 0);
    signal P5,LNLR5,Min5:bit_vector(4 downto 0);
    signal CdistIn,RdistIn:bit_vector(4 downto 0);
    signal P,LN,LR,LC,LN_LR,LN_LC:bit_vector(2 downto 0);
    signal LNLRorLNLC,Pin,LNin,Min:bit_vector(2 downto 0);

begin
-- THE TWO SHIFT REGISTERS (CODE & I)
    codeReg:Sh_Reg generic map(four,Treg)
                      port map(input,SHL_code,nul,clk,code);
    I_Reg:Sh_Reg generic map(twelve,Treg)
                port map(input,SHL_I,Clr_I,clk,I);

-- THE FLIP FLOP (B)
    Bmux1:MUX2_1 generic map(Tmux)
                port map(input,I(0),I0orINPUT,DB);
    invclk:inv port map(clk,clk_bar);
    and_B:and2 port map(Load_B,clk_bar,clk_B);
    BFF:DFF generic map(T_ff)
           port map(DB,clk_B,B,B_bar);
    Bmux2:MUX2_1 generic map(Tmux)
                  port map(B_bar,B,BorB_bar,B_Bbar);

-- S COUNTER (FOR # OF SHAPES)
    S_count:counter generic map(seven,Tcount)
                    port map(I(6 downto 0),
                    Load_S,nul,nul,Dec_S,clk,S);
    Snor:norN generic map(seven,Tnor)
                port map(S,S_0);

-- L COUNTER (FOR # OF SEGMENTS)
    L_count:counter generic map(twelve,Tcount)
                     port
map(I,Load_L,nul,nul,Dec_L,clk,L);
    Lnor0:norN generic map(twelve,Tnor)
                port map(L,L_0);
    invL0:inv port map(L(0),L0_bar);
    Lin<=L(11 downto 1)&L0_bar;
    Lnor1:norN generic map(twelve,Tnor)
               port map(Lin,L_1);
```

```
-- K REGISTER & BCNT (FOR # OF BLOCKS PER SEGMENT)
   K_Reg:Reg generic map(ten,Treg)
                port map(I(9 downto 0),Load_K,clk,K);
   BCNT_count:counter generic map(ten,Tcount)
          port map(K,nul,Clr_BCNT,Inc_BCNT,nul,clk,BCNT);
   Kxnor:xnor2 generic map(ten,Tnor)
                     port map(K,BCNT,BCNT_K);
   K_Dec:Decrementer generic map(ten,Tdec)
                 port map(K,K1);
   K1xnor:xnor2 generic map(ten,Tnor)
                port map(K1,BCNT,BCNT_K_1);

-- N REGISTER (FOR BLOCK SIZE)
   Nin<=I(1)&I(0);
   N_Reg:Reg generic map(two,Treg)
                port map(Nin,Load_N,clk,Nout);
   N<=Nout&high&high&high;

-- FINDING P & Log N
   Pxnor: xnor2_1 port map(N(4), N(3), P(1));
   P(2)<=N(3);
   P(0)<=high;
   invN3:inv port map(N(3),N3_bar);
   LN<=N(3)&N3_bar&P(1);

-- R & C REGISTERS (FOR ROW & COLUMN REMAINDERS) AND THEIR
-- Log
     R_Reg:Reg generic map(five,Treg)
                port map(I(4 downto 0),Load_R,clk,R);
     C_Reg:Reg generic map(five,Treg)
                port map(I(4 downto 0),Load_C,clk,C);
   invR4:inv port map(R(4),R4_bar);
   invR3:inv port map(R(3),R3_bar);
   invR1:inv port map(R(1),R1_bar);
   Ror1:or2 port map(R(2),R(1),R2orR1);
   Ror2:or2 port map(R(2),R1_bar,R2orR1_b);
   Rand:and2 port map(R3_bar,R2orR1_b,Rand1);
   LRor1:or2 port map(R(4),R(3),LR(2));
   LRand:and3 port map(R4_bar,R3_bar,R2orR1,LR(1));
   LRor2:or2 port map(R(4),Rand1,LR(0));
   invC4:inv port map(C(4),C4_bar);
   invC3:inv port map(C(3),C3_bar);
```

```
    invC1:inv port map(C(1),C1_bar);
    Cor1:or2 port map(C(2),C(1),C2orC1);
    Cor2:or2 port map(C(2),C1_bar,C2orC1_b);
    Cand:and2 port map(C3_bar,C2orC1_b,Cand1);
    LCor1:or2 port map(C(4),C(3),LC(2));
    LCand:and3 port map(C4_bar,C3_bar,C2orC1,LC(1));
    LCor2:or2 port map(C(4),Cand1,LC(0));

-- CCNT & RCNT (FOR ADDRESSING THE MEMORY)
    TempImux:MUX2 generic map(five,Tmux)
                 port map(I(4 downto 0),temp,TEMPorI,TempI);
    Ccount:counter generic map(five,Tcount)
                 port map(TempI, Load_CCNT, Clr_CCNT,
                 Inc_CCNT, Dec_CCNT,clk,CCNT);
    Rcount:counter generic map(five,Tcount)
                 port map(I(4 downto 0), Load_RCNT,
                 Clr_RCNT, Inc_RCNT,Dec_RCNT,clk,RCNT);
    NRmux:MUX2 generic map(five,Tmux)
                   port map(N,R,L_1,NR);
    NCmux:MUX2 generic map(five,Tmux)
                   port map(N,C,bcnt_K_1,NC);

-- Rdist & Cdist (FOR COVERING THE SHAPES)
    RdistImux:MUX2 generic map(five,Tmux)
                    port map(I(4 downto 0), Rdist,
                    RdistorI,RdistI);
    RdistMux:MUX2 generic map(five,Tmux)
                  port map(I(4 downto 0), NR,
                   FULLorDIST,RdistIn);
    CdistMux:MUX2 generic map(five,Tmux)
                  port map(RdistI,NC,FULLorDIST,CdistIn);
    RDcount:counter generic map(five,Tcount)
                     port map(RdistIn,Load_Rdist, nul,
                     nul,Dec_Rdist,clk,Rdist);
    CDcount:counter generic map(five,Tcount)
                    port map(CdistIn,Load_Cdist, nul,nul,
                    Dec_Cdist,clk,Cdist);
    RDnor:norN generic map(five,Tnor)
                  port map(Rdist,Rdist_0);
    CDnor:norN generic map(five,Tnor)
                  port map(Cdist,Cdist_0);
```

```
-- FINDING THE MINIMUM (FOR PARTIAL BLOCKS)
   LNLRmux:MUX2 generic map(three,Tmux)
                      port map(LN,LR,L_1,LN_LR);
   LNLCmux:MUX2 generic map(three,Tmux)
                      port map(LN,LC,BCNT_K_1,LN_LC);
   invLR0:inv port map(LN_LR(0),LNLR0_b);
   invLR1:inv port map(LN_LR(1),LNLR1_b);
   invLR2:inv port map(LN_LR(2),LNLR2_b);
   or1Min:or2 port map(LNLR1_b,LN_LC(2),MINor1);
   and1Min:and2 port map(LNLR0_b,LN_LC(2),MINand1);
   and2Min:and2 port map(LNLR2_b,MINor1,MINand2);
   or2Min:or2 port map(MINand1,MINand2,MINor2);
   MINmux:MUX2 generic map(three,Tmux)
                     port map(LN_LC,LN_LR,MINor2,Min);

-- TEMP (THE COUNTER FOR READING INPUTS)
   LRLCmux:MUX2 generic map(three,Tmux)
                port map(LN_LC,LN_LR,LRorLC,LNLRorLNLC);
   P5<=nul&nul&P;
   LNLR5<=nul&nul&LNLRorLNLC;
   Min5<=nul&nul&Min;
   TempMux:MUX8 generic map(five,Tmux)
                port
                map(bin_2,bin_5,bin_10,bin_12,P5,LNLR5,Min
                5,Cdist,temp_sel,TempIn);
   Temp_count:counter generic map(five,Tcount)
                      port map(TempIn,Load_temp,nul,nul,
                      Dec_temp,clk,temp);
   TempNor:norN generic map(five,Tnor)
                      port map(temp,temp_0);

-- THE MEMORY
   RAM:memory generic map(ten,five,Tmem,per)
              port map(N,K,BCNT,RCNT,CCNT,
              clk,Rd,W,B_Bbar,ack);
   I1<=I(1);
   I0<=I(0);
End struct;

-------------------------------------------
-------------------------------------------
```

```
-- THIS IS THE FSM ENTITY (ALGORITHMIC MODELING)

Entity FSM is
     port(clk,clock,start,L_1,BCNT_K_1,I1,I0,S_0,L_0,
          BCNT_K,Rdist_0,Cdist_0,temp_0,
          ack:in bit;code:in bit_vector(3 downto 0);
          input,BorB_bar,Load_S,Dec_S,Load_L,Dec_L,
          Load_K,Clr_BCNT,Inc_BCNT,Load_N,
          Load_R,Load_C,Clr_CCNT,
          Load_CCNT,Inc_CCNT,Dec_CCNT,TemporI,
          Clr_RCNT,Load_RCNT,Inc_RCNT,Dec_RCNT,
          FULLorDIST,RdistorI,Load_Cdist,Dec_Cdist,
          Load_Rdist,Dec_Rdist,LRorLC,
          Load_temp,Dec_temp,W,Rd,Clr_I:out bit;
          SHL_code,SHL_I,I0orINPUT,Load_B:inout bit;
          temp_sel:out bit_vector(2 downto 0));
End FSM;

use STD.TEXTIO.all;

Architecture Alg of FSM is

   signal PS,NS:integer:=0;
   signal NL:boolean:=true;
   signal i:integer;
   signal count:integer:=0;
     signal code10,code32,RCdist:bit_vector(1 downto 0);

begin
-- THIS PROCESS COUNTS THE NUMBER OF CLOCKS UNTIL THE
DECODING FINISHES
     clk_count:process(clk,NS)
     begin
     if(NS=0) then
          NULL;
          elsif(clk='1' and clk'event) then
          count<=count+1;
        end if;
     end process;
```

```
-- THIS PROCESS READS THE INPUTS BIT BY BIT
   read_bit:process
          variable vline,oline:line;
      variable v:character;
      file infile:text is "s15850f.minbin";
      file debug:text is out "debug.out";
      begin
          while not(endfile(infile)) loop
               readline(infile,vline);
            for j in 1 to vline'length loop
               wait until ((SHL_code='1' or SHL_I='1' or
                    (Load_B='1' and I0orINPUT='0'))and
                         (clock='1' and clock'event));
                  read(vline,v);
                  write(oline,v);
                  if(v='1') then
                    input<='1';
                  else
                    input<='0';
                  end if;
            end loop;
            writeline(debug,oline);
         end loop;
      end process;

-- THIS PROCESS UPDATES THE PRESENT STATES
     P_S:process(clk,NS)
     begin
          if(clk='1' and clk'event) then
          PS<=NS;
         end if;
      end process;

-- THIS PROCESS OUTPUTS THE Dec_L SIGNAL (A MEALY STATE)
   mealy:process
     begin
          wait until (PS=19 and BCNT_K='1'); --then
          Dec_L<='1';
            wait until clk'event;
            wait until clk'event;
          Dec_L<='0';
      end process;
```

```
-- THIS IS A BLOCK FOR DEFINING SOME SIGNALS
     define:block
          begin
          code10<=code(1)&code(0);
           code32<=code(3)&code(2);
               RCdist<=Rdist_0&Cdist_0;
      end block define;

-- THIS PROCESS DETERMINES THE NEXT STATE

N_S:process(PS,L_1,BCNT_K_1,I1,I0,S_0,L_0,BCNT_K,Rdist_0,
Cdist_0,temp_0,ack,code,code10,code32,RCdist,start)
     begin
          case PS is
            when 0 => if(start='1') then NS<=1; else
                       NS<=0; end if;
            when 1 => NS<=2;
            when 2 => if(temp_0='1') then NS<=3; else
                       NS<=2; end if;
            when 3 => NS<=4;
            when 4 => if(temp_0='1') then NS<=5; else
                       NS<=4; end if;
            when 5 => NS<=6;
            when 6 => if(temp_0='1') then NS<=7; else
                       NS<=6; end if;
            when 7 => NS<=8;
            when 8 => if(temp_0='1') then NS<=9; else
                       NS<=8; end if;
            when 9 => NS<=10;
            when 10 => if(temp_0='1') then NS<=11; else
                        NS<=10; end if;
            when 11 => NS<=12;
            when 12 => NS<=13;
            when 13 => NS<=14;
            when 14 => if(I1='1') then NS<=21; else
                       NS<=15; end if;
            when 15 => NS<=16;
            when 16 => if(Cdist_0='0') then NS<=17;
                       elsif(Cdist_0='1' and Rdist_0='0')
                       then NS<=18;
                      else NS<=19; end if;
```

```
when 17 => if(I0='0') then NS<=15; else
            NS<=16; end if;
when 18 => if(I0='0') then NS<=15; else
            NS<=16; end if;
when 19 => if(BCNT_K='0') then NS<=13;  else
            NS<=20; end if;
when 20 => if(ack='0') then NS<=20;
            elsif(ack='1' and L_0='0') then
            NS<=12;
           else NS<=0; end if;
when 21 => NS<=22;
when 22 => if(Cdist_0='0') then NS<=23;
            elsif(Cdist_0='1' and Rdist_0='0')
            then NS<=24;
           else NS<=25; end if;
when 23 => NS<=22;
when 24 => NS<=22;
when 25 => NS<=26;
when 26 => if(temp_0='1') then NS<=27; else
            NS<=26; end if;
when 27 => NS<=28;
when 28 => NS<=29;
when 29 => case code10 is
               when "00"|"11" => NS<=30;
               when "01"|"10" => NS<=43;
               end case;
when 30 => if(temp_0='1') then NS<=31; else
            NS<=30; end if;
when 31 => NS<=32;
when 32 => if(temp_0='1') then NS<=33; else
            NS<=32; end if;
when 33 => case code10 is
               when "00" => NS<=34;
               when "11" => NS<=36;
              when others => NULL;
               end case;
when 34 => case code10 is
              when "00" => NS<=35;
             when "11" => case RCdist is
                                 when
                                 "00"|"10" =>
                                 NS<=40;
```

```
                                        when "01" =>
                                        NS<=41;
                                      when "11" =>
                                      NS<=35;
                                    end case;
                  when others => NULL;
               end case;
when 35 => if(S_0='0') then NS<=28; else
               NS<=19; end if;
when 36 => if(temp_0='1') then NS<=37; else
               NS<=36; end if;
when 37 => NS<=38;
when 38 => if(temp_0='1') then NS<=39; else
               NS<=38; end if;
when 39|40|41 => NS<=34;
when 43 => NS<=44;
when 44 => NS<=45;
when 45 => if(temp_0='1') then NS<=46; else
               NS<=45; end if;
when 46 => NS<=47;
when 47 => if(temp_0='0') then NS<=47;
                    else
                       if(code="0101") then NS<=48;
                    elsif(code="0100") then NS<=49;
                    else NS<=50; end if;
               end if;
when 48|49|50 => NS<=51;
when 51 => if(temp_0='0') then NS<=51;
                    else NS<=52;
               end if;
when 52 => NS<=54;
when 54 => case code32 is
                          when "01" =>
                          if(Rdist_0='0') then
                               case code10 is
                                    when "00" =>
                                    NS<=55;
                                    when "01" =>
                                    NS<=56;
                                    when "10" =>
                                    NS<=57;
```

```
                                                    when "11" =>
                                                    NS<=58;
                                                  end case;
                                                else NS<=35;
                                                end if;
                                              when "10" =>
                                              if(Cdist_0='0') then
                                                   if(code(1)='1')
                                                   then NS<=61;
                                                        else NS<=62;
                                                        end if;
                                                else
                                                   if(Rdist_0='0')
                                                   then
                                                    case code10 is
                                                    when "00"|"11"
                                                          => NS<=59;
                                                    when "01"|"10"
                                                          => NS<=60;
                                                    end case;
                                                    else
                                                        NS<= 35;
                                                    end if;
                                                end if;
                                    when others => NULL;
                                    end case;
                  when 55|56|57|58|61|62|63 => NS<=54;
                  when 59|60 => NS<=63;
                  when others => NULL;
               end case;
            end process;

-- THIS PART ASSIGNS THE APPROPRIATE VALUES TO THE UTPUT
SIGNALS
SHL_code<='1' when (PS=28 or PS=29 or PS=43 or PS=44) else
'0';
SHL_I<='1' when(PS=2 or PS=4 or PS=6 or PS=8 or PS=10 or
PS=13 or PS=14 or PS=26 or PS=30 or PS=32 or PS=36 or
PS=38 or PS=45 or PS=47 or PS=51)  else '0';
Clr_I<='1' when (PS=25 or PS=29 or PS=31 or PS=33 or PS=37
or PS=46 or PS=48 or PS=49 or PS=50) else '0';
IOorINPUT<='0' when PS=15 else '1';
```

```
Load_B<='1' when (PS=15 or PS=21) else '0';
BorB_bar<='0' when PS=22 else '1';
Load_S<='1' when PS=27 else '0';
Dec_S<='1' when PS=35 else '0';
Load_L<='1' when PS=5 else '0';
Load_K<='1' when PS=7 else '0';
Clr_BCNT<='1' when PS=12 else '0';
Inc_BCNT<='1' when PS=19 else '0';
Load_N<='1' when PS=3 else '0';
Load_R<='1' when PS=9 else '0';
Load_C<='1' when PS=11 else '0';
Clr_CCNT<='1' when (PS=13 or PS=18 or PS=24) else '0';
Load_CCNT<='1' when (PS=33 or PS=41 or PS=48 or PS=49 or
PS=50 or PS=59 or PS=60) else '0';
Inc_CCNT<='1' when (PS=17 or PS=23 or PS=40 or PS=56 or
PS=57 or PS=58 or PS=62) else '0';
Dec_CCNT<='1' when (PS=42 or PS=61) else '0';
TemporI<='1' when (PS=41 or PS=59 or PS=60) else '0';
Clr_RCNT<='1' when PS=13 else '0';
Load_RCNT<='1' when (PS=31 or PS=46) else '0';
Inc_RCNT<='1' when (PS=18 or PS=24 or PS=41 or PS=55 or
PS=57 or PS=59) else '0';
Dec_RCNT<='1' when (PS=58 or PS=60) else '0';
FULLorDIST<='1' when (PS=13 or PS=18 or PS=24) else '0';
RdistorI<='1' when PS=63 else '0';
Load_Cdist<='1' when (PS=13 or PS=18 or PS=24 or PS=33 or
PS=39 or PS=41 or PS=48 or PS=49 or PS=50 or PS=52 or
PS=53 or PS=63)      else '0';
Dec_Cdist<='1' when (PS=17 or PS=23 or PS=39 or PS=40 or
PS=42 or PS=61 or PS=62) else '0';
Load_Rdist<='1' when (PS=13 or PS=37 or PS=52 or PS=53)
else '0';
Dec_Rdist<='1' when (PS=18 or PS=24 or PS=41 or PS=55 or
PS=56 or PS=57 or PS=58 or PS=59 or PS=60) else '0';
LRorLC<='1' when (PS=29 or PS=33 or PS=49) else '0';
Load_temp<='1' when (PS=1 or PS=3 or PS=5 or PS=7 or PS=9
or PS=25 or PS=29 or PS=31 or PS=33 or PS=37 or PS=39 or
PS=46 or PS=48 or PS=49 or PS=50 or PS=52 or PS=53) else
'0';
Dec_temp<='1' when(PS=2 or PS=4 or PS=6 or PS=8 or PS=10
or PS=26 or     PS=30 or PS=32 or PS=36 or PS=38 or PS=45
or PS=47 or PS=51)  else '0';
```

```
W<='1' when (PS=16 or PS=22 or PS=34 or PS=54) else '0';
Rd<='1' when PS=20 else '0';

temp_sel<= "000" when PS=1 else
                "001" when (PS=7 or PS=9) else
                "010" when PS=5 else
                "011" when PS=3 else
                "100" when PS=25 else
                "101" when (PS=29 or PS=31 or PS=33 or
                            PS=37 or PS=46 or PS=48 or
                            PS=49) else
                "110" when PS=50 else
                "111"; -- when (PS=52 or PS=53);
End Alg;

------------------------------------------------
------------------------------------------------


-- THIS IS THE DECODER ENTITY, WHICH ASSEMBLES THE FSM AND
THE DATA PATH

Entity Decoder is
     port(start,clk,clock:in bit);
end Decoder;

Architecture struct of Decoder is

     component DataPath
   port(input,clk,SHL_code,Clr_I,SHL_I,IOorINPUT,Load_B,
     BorB_bar,Load_S,Dec_S,Load_L,Dec_L,Load_K,Clr_BCNT,
     Inc_BCNT,Load_N,TemporI,Load_R,Load_C,Clr_CCNT,
     Load_CCNT,Inc_CCNT,Dec_CCNT,Clr_RCNT,Load_RCNT,
     Inc_RCNT,Dec_RCNT,FULLorDIST,RdistorI,Load_Cdist,
     Dec_Cdist,Load_Rdist,Dec_Rdist,LRorLC,Load_temp,
     Dec_temp,W,Rd:in bit;
     Dir,L_1,BCNT_K_1:inout bit;
     temp_sel:in bit_vector(2 downto 0);code:out
     bit_vector(3 downto 0);
     I1,I0,S_0,L_0,BCNT_K,Rdist_0,Cdist_0,temp_0,ack:out
     bit);
```

```
End component;


component FSM
   port(clk,clock,start,L_1,BCNT_K_1,I1,I0,S_0,L_0,BCNT_K,
        Rdist_0,Cdist_0,temp_0,ack:in bit;code:in
        bit_vector(3 downto 0);
        input,BorB_bar,Load_S,Dec_S,Load_L,Dec_L,Load_K,
        Clr_BCNT,Inc_BCNT,Load_N,Load_R,Load_C,Clr_CCNT,
        Load_CCNT,Inc_CCNT,Dec_CCNT,TemporI,Clr_RCNT,
        Load_RCNT,Inc_RCNT,Dec_RCNT,
        FULLorDIST,RdistorI,Load_Cdist,Dec_Cdist,
        Load_Rdist,Dec_Rdist,LRorLC,
        Load_temp,Dec_temp,W,Rd,Clr_I:out bit;
        SHL_code,SHL_I,I0orINPUT,Load_B:inout bit;
        temp_sel:out bit_vector(2 downto 0));
End component;

for F:FSM use entity work.FSM(Alg);
for D:DataPath use entity work.DataPath2(struct);

signal input,SHL_code,Clr_I,SHL_I,I0orINPUT,Load_B,
BorB_bar,Comp_Dir,set_Dir,reset_Dir,Load_S,Dec_S,Load_L,
Dec_L,Load_K,Clr_BCNT,Inc_BCNT,Load_N,Load_R,Load_C,
Clr_CCNT,Load_CCNT,Inc_CCNT,Dec_CCNT,Sel_Dir,Clr_RCNT,
Load_RCNT,Inc_RCNT,Dec_RCNT,FULLorDIST,TEMPorI,Load_Cdist,
Dec_Cdist,Load_Rdist,Dec_Rdist,LRorLC,Load_temp,Dec_temp,W
,Rd,RdistorI,Dir,L_1,BCNT_K_1,I1,I0,S_0,L_0,BCNT_K,Rdist_0
,
Cdist_0,temp_0,ack:bit;
signal temp_sel:bit_vector(2 downto 0);
signal code:bit_vector(3 downto 0);

begin
F:FSM port map(clk,clock,start,L_1,BCNT_K_1,I1,I0,S_0,
L_0,BCNT_K,Rdist_0,Cdist_0,temp_0,ack,code,input,BorB_bar,
Load_S,Dec_S,Load_L,Dec_L,
Load_K,Clr_BCNT,Inc_BCNT,Load_N,
Load_R,Load_C,Clr_CCNT,Load_CCNT,Inc_CCNT,Dec_CCNT,TemporI
,Clr_RCNT,Load_RCNT,Inc_RCNT,Dec_RCNT,FULLorDIST,RdistorI,
Load_Cdist,Dec_Cdist,Load_Rdist,Dec_Rdist,LRorLC,Load_temp
```

```
,Dec_temp,W,Rd,Clr_I,SHL_code,SHL_I,I0orINPUT,Load_B,
temp_sel);

D:DataPath port map(input,clk,SHL_code,Clr_I,SHL_I,
I0orINPUT,Load_B,BorB_bar,Load_S,Dec_S,Load_L,Dec_L,Load_K
,Clr_BCNT,Inc_BCNT,Load_N,TemporI,Load_R,Load_C,Clr_CCNT,
Load_CCNT,Inc_CCNT,Dec_CCNT,Clr_RCNT,Load_RCNT,Inc_RCNT,
Dec_RCNT,FULLorDIST,RdistorI,Load_Cdist,Dec_Cdist,
Load_Rdist,Dec_Rdist,LRorLC,Load_temp,Dec_temp,W,Rd,
Dir,L_1,BCNT_K_1,temp_sel,code,I1,I0,S_0,L_0,BCNT_K,
Rdist_0,Cdist_0,temp_0,ack);

end struct;

-------------------------------------------------------------
```

REFERENCES

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

[2] Alsuwaiyel, M. H. *Algorithms: Design Techniques and Analysis*, World Scientific, 1999.

[3] Baker, L., *VHDL Programming with Advanced Topics*, John Wiley & Sons, 1993.

[4] Bommu, S., Chakradhar, S. and Doreswamy, K., "Static Test Sequence Compaction Based on Segment Reordering and Accelerated Vector Restoration," *Proc. of International Test Conference*, pp. 954-961, Oct. 1998.

[5] Burrows, M. and Wheeler, D., "Block Storing Lossless Data Compression Algorithm," *System Research Center*, Research Report 124, digital system Research Center, Palo Alto, CA, May 1994.

[6] Chakrabatry, K., Murray, B., Liu, J. and Zhu, M., "Test Width Compression for Built-in Self Testing," *Proc. of International Test Conference*, pp. 328-337, 1997.

[7] Chakradhar, S. and Raghunathan, A., "Bottleneck Removal Algorithm for Dynamic Compaction in Sequencial Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 16, No. 10, pp. 1157-1172, Oct. 1997.

[8] Chandra, A. and Chakrabarty, K., "Frequency-Directed Run-Length (FDR) Codes with Application to Systems-on-a-Chip Test Data Compression," *Proc. of IEEE VLSI Test Symposium,* pp. 42-47, 2001.

[9] Chandra, A. and Chakrabarty, K., "Test Data Compression for System-On-a-Chip using Golomb Codes," *Proc. of IEEE VLSI Test Symposium,* pp. 113-120, 2000.

[10] Chandramouli, R. and Pateras, S., "Testing Systems on a Chip," *IEEE Spectrum,* pp. 42-47, Nov. 1996.

[11] Chang, J. and Lin, C., "Test Set Compaction for Combinational Circuits," *IEEE Trans. on Computer Aided Design*, pp. 1370-1378, Nov. 1995.

[12] Cormen, T.H., Leiserson, C.E. and Rivest, R.L., *Introduction to Algorithms*, McGraw Hill, 1989.

[13] Gibson, J., Berger, T., Lookabaugh, T., Lindbergh, D. and Baker, R., *Digital Compression for Multimedia*, Morgan Kaufman Publisher, Inc. 1998.

[14] Hamzaoglu, I. and Patel, J., "Compact Two-Pattern Test Set Generation for Combinational and Full Scan Circuit," *Proc. of IEEE International Test Conference*, pp. 944-953, Oct. 1998.

[15] Hamzaoglu, I. and Patel, J., "Test Set Compaction Algorithms for Combinational Circuits," *Proc. of the International Conference on Computer-Aided Design*, pp. 260-267, Nov. 1998.

[16] HHB Inc., *THESEUS User's Manual*, Malwah, New Jersey: HHB Inc., Oct. 1987.

[17] Hochbaum, D., "An Optimal Test Compression Procedure for Combinational Circuits," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 10, pp.1294-1299, Oct. 1996.

[18] Holland, J. H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[19] Ishida, M., Ha, D. S. and Yamaguchi, T., "COMPACT: A Hybrid Method for Compressing Test Data," *Proc. of VLSI Test Symposium*, pp. 62-69, 1998.

[20] Jas, A. and Touba, N., "Test Vector Decompression via Cyclical Scan Chains and its Application to Testing Core-Based Designs," *Proc. of International Test Conference*, pp. 458-464, 1998.

[21] Jas, A. and Touba, N., "Using an Embedded Processor for Efficient Deterministic Testing of System-on-a-chip," *Proc. of IEEE International Conference on Computer Design*, pp. 418-423, 1999.

[22] Jas, A., Dastidar, J. G. and Touba, N., "Scan Vector Compression/Decompression Using Statistical Coding," *Proc. of IEEE VLSI Test Symposium*, pp. 202-207, 1994.

[23] Jaz, A., Mohanram, K. and Touba, N., "An Embedded Core DFT Scheme to Obtain Highly Compressed Test Sets," *Proc. of IEEE Asian Test Symposium*, pp. 275-280, 1999.

[24] Kajihara, S., Pomeranz, I., Kinoshita, K., and Reddy, S., "Cost-Effective Generation of Minimal Test sets for Stuck-at Faults in Combinational Circuits," *IEEE Trans. on Computer Aided Design*, pp. 1496-1504, Dec. 1995.

[25] Nelson, M. R., "Data compression with the Burrows Wheeler Transformation," *Dr. Dobbs Journal*, pp. 46-50, Sept. 1996.

[26] Pomeranz, I. and Reddy, S. M., "On Improving Genetic Optimization based Test Generation," *Proc. of Europ. Design & Test Conf.*, pp. 506-511, March 1997.

[27] Pomeranz, I. and Reddy, S. M., "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits," *Proc. of the International Conference on Computer Design*, pp. 360-365, University of Iowa, Aug. 1997.

[28] Pomeranz, I. and Reddy, S., "Dynamic Test Compaction for Synchronous Sequential Circuits Using Static Compaction Techniques," *Proc. of International Symposium on Fault Tolerant Computing*, pp. 53-61, 1996.

[29] Pomeranz, I. and Reddy, S., "On the Compaction of Test Sets Produced by Genetic Optimization," *Proc. of IEEE Asian Test Symposium*, pp. 4-9, Nov. 1997.

[30] Pomeranz, I., Reddy, L., and Reddy, S., "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits," *Proc. of IEEE Intenational Test Conference*, pp. 194-203, 1991.

[31] Rajski J., Tyszer, J. and Zacharia, N., "Test Data Decompression for Multiple Scan Designs with Boundary Scan," *IEEE Tran. on Computers,* Vol. 47, No. 11, pp. 1188-1200, Nov. 1998.

[32] Roy, R., Niermann, T., Patel, J., Abraham, J., and Saleh, R., "Compaction of ATG-Generated Test Sequences for Sequential Circuits," *Proc. of IEEE International Conference on Computer-Aided Design*, pp. 382-385, Nov. 1998.

[33] Rudnick, E. and Patel, J., "Putting the squeeze on Test Sequences," *Proc. of IEEE International Test Conference*, pp. 723-732, 1997.

[34] Rudnick, E. and Patel, J., "Simulation-Based Technique for Dynamic Test Sequence Compaction," *Proc. of International Conference on Computer-Aided Design*, pp. 67-73, 1996.

[35] Schulz, M., Trischhler, E., and Sarfert, T., "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, pp. 126-137, Jan. 1988.

[36] Tromp, G., "Minimal Test Sets for Combinational Circuits," *Proc. of IEEE International Test Conference*, pp. 204-209, 1991.

[37] Yamaguchi, T., Tilgner, M., Ishida, M. and Ha, D. S., "An Efficient Method for Compressing Test Data," *Proc. of International Test Conference*, pp. 79-88, Nov. 1997.

[38] Zacharia, N., Rajski, J., Tyszer, J. and Waicukauski, A., "Two-Dimentional Test Data Decompressor for Multiple Scan Designs," *International Test Conference,* pp. 186-194, 1996.

[39] Zorian, Y., Marinissen, E.J. and Dey, S., "Testing Embedded-Core Based System Chips," *Proc. of Int. Test Conference,* pp. 130-143, 1998.

VITA

- Esam Ali Hasan Khan.
- Born in Makkah, Saudi Arabia.
- Received Bachelor of Science (B. S.) degree in Computer Engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia in June 1999.
- Joined Computer Engineering Department, KFUPM, as a graduate assistant in September 1999.
- Received Master of Science (M. S.) degree in Computer Engineering from KFUPM, Dhahran, Saudi Arabia, in June 2001.