

Copyright

by

Kedarnath Jayaraman Balakrishnan

2004

The Dissertation Committee for Kedarnath Jayaraman Balakrishnan certifies that this is the approved version of the following dissertation:

**New Approaches and Limits to Test Data Compression  
for Systems-on-chip**

Committee:

---

Nur A. Toubia, Supervisor

---

Tony Ambler

---

Gustavo de Veciana

---

Margarida Jacome

---

Abhijit Jas

**New Approaches and Limits to Test Data Compression  
for Systems-on-chip**

by

**Kedarnath Jayaraman Balakrishnan, B. Tech., M. S. E.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

August 2004

to  
my parents

# Acknowledgments

I would like to thank my advisor, Nur Touba for his guidance and encouragement throughout my Ph.D. He is unarguably the single biggest influence on this work. His valuable suggestions and inputs in our discussions have been extremely useful in my understanding of the subject. It would not be wrong to say that this thesis would not have been possible without his support and patience. I am deeply grateful to him for his personal warmth and caring advice at all times and especially during my job interview process. I admire his professionalism and high standards in integrity and honesty. They will be a goal for me all my life.

Thanks to Tony Ambler, Gustavo de Veciana, Margarida Jacome and Abhijit Jas for agreeing to be part of my dissertation committee. Gustavo has been very helpful in providing feedback and technical guidance on my research whenever I needed it.

I would like to acknowledge the contributions of Dinesh Kumar Sharma, Anil Kulkarni and H. Narayanan of IIT Bombay who sowed the seeds of research into my brain. My decision to pursue a Ph.D. was entirely due to their encouragement.

Thanks are due to Kartik and Ranga for innumerable discussions, insights and technical trivia during the initial stages of my Ph.D. I am very indebted to Kartik, who has been influential in most of the decisions I have made in my career so far.

Thanks to all my roommates, past and present, Karunesh, Aniket and Vishvjeet for bearing with me and providing me excellent food for the last four years. I would

also like to thank Madhu, Lasya and Devashree for providing me a home away from home where I could just unwind. And to all my friends Srujana, Baap, Subbuk and the ACES 6th floor gang esp. Amol, Arun and Ravi for the long hours of coffee and debates on Ph.D., life, universe and everything. Special thanks to Priya for just being there when I needed it and for bugging me to complete this dissertation.

Lastly, I would like to thank my parents and my brother Badri who have patiently endured me and supported me steadfastly in all my endeavors. I dedicate my doctorate to them.

KEDARNATH JAYARAMAN BALAKRISHNAN

*The University of Texas at Austin*

*August 2004*

# New Approaches and Limits to Test Data Compression for Systems-on-chip

Publication No. \_\_\_\_\_

Kedarnath Jayaraman Balakrishnan, Ph.D.  
The University of Texas at Austin, 2004

Supervisor: Nur A. Toubia

Recent advances in design technology have made it possible to build systems containing different types of components on the same chip. These complex systems-on-chip (SoC) contain components that cover a wide range of functions and technologies from processors and other digital circuits in CMOS to DRAM to analog circuits. As a result, the testing of such complex SoCs has become an important and difficult problem. This thesis investigates the use of test data compression methods to deal with the enormous amount of test data of complex SoCs. The first part of this thesis studies the use of an embedded processor present in the SoC to help in testing the other cores. Specifically, the focus is on the use of processor to perform test vector decompression in software. The next part of this thesis looks at methods for improving the compression of hardware based linear decompression techniques. The last part uses entropy theory to calculate the limits to test data compression.

# Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Abstract</b>	<b>vii</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xiii</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
<b>Chapter 2 Previous Work</b>	<b>9</b>
2.1 Test Data Compression . . . . .	9
2.2 Testing using an Embedded Processor . . . . .	12
<b>Chapter 3 Matrix-based Compression</b>	<b>14</b>
3.1 Compression Algorithm . . . . .	15
3.1.1 Single Size Decomposition (SSD) . . . . .	16
3.1.2 Multiple Size Decomposition (MSD) . . . . .	16
3.1.3 Multiple Vector Decomposition (MVD) . . . . .	17
3.2 Decompression Using an Embedded Processor . . . . .	18
3.3 Analysis . . . . .	20
3.4 Experimental Results . . . . .	23

3.4.1	Program Size . . . . .	23
3.4.2	Data Compression . . . . .	24
3.5	Summary . . . . .	28
<b>Chapter 4 Pseudo-Random Linear Expansion</b>		<b>30</b>
4.1	Overview of Proposed Scheme . . . . .	31
4.2	Details of Proposed Method . . . . .	34
4.3	Experimental Results . . . . .	38
4.4	Summary . . . . .	41
<b>Chapter 5 Diagnostic Response Compression for BIST</b>		<b>43</b>
5.1	Related Work . . . . .	43
5.2	Response Compression . . . . .	45
5.3	Diagnosis . . . . .	48
5.4	Experimental Results . . . . .	51
5.5	Summary . . . . .	54
<b>Chapter 6 Scan Inversion to Improve Compression</b>		<b>56</b>
6.1	Related Work . . . . .	58
6.2	Scan Inversion . . . . .	60
6.3	Selecting Set of Inverted Scan Cells . . . . .	61
6.4	Using Scan Inversion to Increase Encoding Efficiency . . . . .	66
6.4.1	Reducing Free-Variables per Test Cube . . . . .	67
6.4.2	Increasing Specified Bits per Test Cube . . . . .	68
6.5	Experimental Results . . . . .	68
6.6	Summary . . . . .	73
<b>Chapter 7 Entropy Limits to Compression</b>		<b>74</b>
7.1	Entropy Analysis for Test Data . . . . .	75

7.1.1	Partitioning Test Data into Symbols . . . . .	75
7.1.2	Specifying the Don't Cares . . . . .	77
7.2	Symbol Length Versus Compression Limits . . . . .	84
7.3	Analysis of Test Data Compression Schemes . . . . .	87
7.3.1	Fixed symbol length schemes . . . . .	88
7.3.2	Variable symbol length schemes . . . . .	89
7.4	Analysis of LFSR Reseeding Schemes . . . . .	89
7.5	Summary . . . . .	91
<b>Chapter 8 Conclusions and Future Work</b>		<b>92</b>
<b>Bibliography</b>		<b>94</b>
<b>Vita</b>		<b>102</b>

# List of Tables

3.1	Simulated values for the number of cycles . . . . .	21
3.2	Buffer sizes for different data transfer rates of the tester . . . . .	22
3.3	Program sizes of different schemes . . . . .	23
3.4	Compression obtained using single size decomposition (SSD) . . . . .	25
3.5	Compression obtained using multiple size decomposition (MSD) . . . . .	25
3.6	Compression obtained using multiple vector decomposition (MVD) . . . . .	26
3.7	Comparison of the three methods . . . . .	27
3.8	Comparison with other techniques . . . . .	28
4.1	Comparison of code size . . . . .	39
4.2	Compression results for proposed method . . . . .	40
4.3	Comparison of compressed test data . . . . .	40
4.4	Comparison of test data storage requirements . . . . .	41
5.1	Experimental results on ISCAS'89 benchmark circuits . . . . .	55
6.1	Results for combinational linear decompressor . . . . .	71
6.2	Results for sequential linear decompressor . . . . .	71
6.3	Comparison of test data for different encoding schemes . . . . .	72
7.1	Test set divided into 4-Bit blocks . . . . .	76

7.2	Probability table for symbol length of 4 . . . . .	77
7.3	Probability table for symbol length of 6 . . . . .	78
7.4	Probability table for runs of 0's . . . . .	79
7.5	Exact entropy compression limits using the greedy fill algorithm . .	83
7.6	Entropy compression limits using approximate fill algorithm . . . . .	84
7.7	Comparison of fixed symbol length schemes with entropy limit . . .	88
7.8	Comparison of variable symbol length schemes with entropy limit . .	89

# List of Figures

1.1	Test infrastructure for cores in an SoC . . . . .	2
1.2	Test data compression . . . . .	3
1.3	Scan-based BIST architecture . . . . .	6
3.1	Matrix operation $A \tilde{\oplus} B$ . . . . .	14
3.2	Set of linear equations for the decomposition . . . . .	15
3.3	Formation of matrix M with $n = 4$ . . . . .	16
3.4	Formation of matrix M using MVD . . . . .	17
3.5	Example of test architecture . . . . .	19
3.6	Pseudo-code for decompression algorithm . . . . .	29
4.1	Forming test set from compressed bits . . . . .	32
4.2	Software program for proposed method . . . . .	36
4.3	Random number generation . . . . .	37
4.4	Optimized code for the inner loop . . . . .	38
5.1	Architecture of a typical SoC . . . . .	46
5.2	Identification of incorrect bits . . . . .	49
5.3	Variation of suspect size and diagnostic accuracy with <i>numxors</i> . . . . .	51
5.4	Variation of suspect size and diagnostic accuracy with error probability . . . . .	52
5.5	Variation of suspect size and diagnostic accuracy with compaction ratio . . . . .	53

6.1	Normal scan chain . . . . .	60
6.2	Scan chain with 3rd scan cell inverted . . . . .	61
6.3	System of linear equations for test cube $t_1$ . . . . .	63
6.4	Gauss-Jordan reduction for test cube $t_1$ . . . . .	63
6.5	System of linear equations for test cube $t_2$ . . . . .	63
6.6	Gauss-Jordan reduction for test cube $t_2$ . . . . .	64
6.7	Constraint matrix for test set $\{t_1, t_2\}$ . . . . .	64
6.8	Combinational linear decompressor . . . . .	70
6.9	Continuous-flow sequential linear decompressor . . . . .	70
7.1	Example of specifying don't cares to minimize entropy . . . . .	80
7.2	Compression limits for different symbol lengths . . . . .	85
7.3	Compression limits for different symbol lengths . . . . .	86
7.4	Compression limits for different symbol lengths . . . . .	87
7.5	LFSR reseeding compression versus symbol length for <i>s13207</i> . . . . .	91

# Chapter 1

## Introduction

Recent advances in design technology have made it possible to build complete systems containing different types of components (also called cores) on the same chip. These complex systems-on-chips (SoCs) incorporate many different cores that cover a wide range of functions (processors, memory and other digital logic) and use an unprecedented range of technologies, from CMOS logic to DRAM to analog circuits. There is also an increasing trend of design reuse by utilizing Intellectual Property (IP) cores. IP cores are pre-designed and pre-verified by their vendors and the SoC designer has very limited knowledge of the structure of the core. The SoC designer has to treat the IP cores as a black box and just integrates them into the system. The use of these cores reduces the SoC design cycle and hence shortens the time-to-market, which greatly influences the cost and competitiveness of the SoC. But for such cores, the core provider develops the core test methodology - both the design-for-test structures and the corresponding patterns and delivers it with the core. Since the core provider has little or no knowledge about the system chip environment where the core will be used, the core test is developed independently for each core. Therefore, the same SoC might have some cores that require testing with deterministic patterns while some others with pseudo-random or scan-based built-in self-test (BIST). The system chip test developer has to take all these different test types into account when developing system level test and diagnosis strategies. The system level test is a composite test that consists of individual tests for each

core, and tests for the interconnect logic and wiring connecting the different cores together.

The test infrastructure of an embedded core in an SoC consists of three main elements as illustrated in Fig 1.1. The test pattern source generates the test stimuli for the core; the sink compares the test response(s) to the expected response(s) to give out pass/fail information. The test access mechanism transports test patterns from the test pattern source to the core under test. It also transports test responses from the core under test to a test pattern sink. The test pattern sources and sinks for cores can be implemented either off-chip using external automatic test equipment (ATE) as shown in Fig 1.1 (a), or on-chip, using BIST, or a combination of both as illustrated in Fig 1.1 (b).

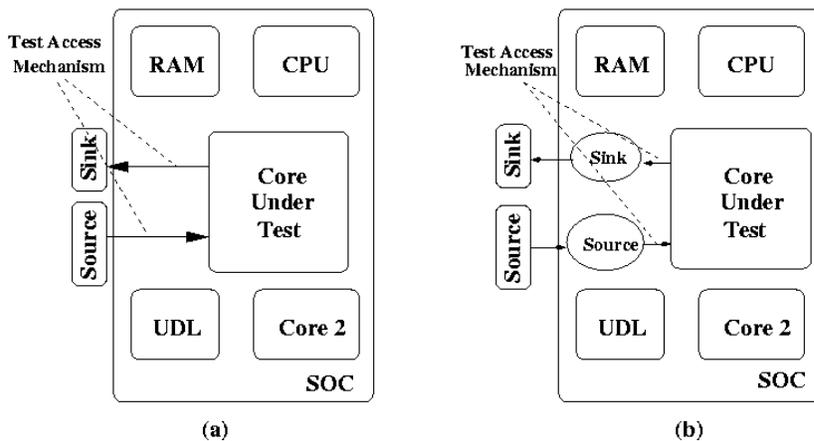


Figure 1.1: Test infrastructure for cores in an SoC

A major challenge of testing complex SoC designs is dealing with the enormous amount of test data volume [Zorian 98]. If there are several cores and each core comes with its own test patterns, the SoC on the whole ends up with a large test data volume. Large test data volume results in long test application time because the data needs to be transferred across the low test data bandwidth link between the

tester and the chip. Moreover, it requires a large amount of tester memory. Since ATE costs directly depend on the amount of tester memory, the costs of conventional testing of an SoC using ATE are scaling up rapidly. Even though BIST has been employed elsewhere to reduce dependency on expensive ATEs, for core based SoCs where the core under test doesn't come with its own BIST structure, it will be difficult to generate patterns on-chip in a cost-effective manner. The test patterns of such cores that come with functional tests or automatic test pattern generator (ATPG) generated tests are often irregular in structure and cannot be generated on-chip at acceptable area costs.

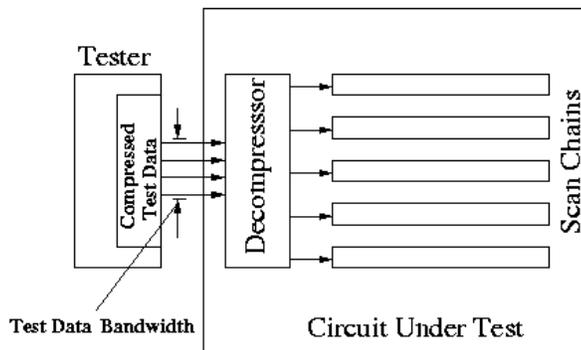


Figure 1.2: Test data compression

One approach for dealing with this problem is to use test data compression techniques to reduce the amount of data that is stored on the external tester. Both the test stimuli and the output response can be compressed. The output response is much easier to compress since lossy compression techniques can be used. Input test data compression is much more difficult because lossless compression techniques must be used to preserve the fault coverage. In test data compression, the test patterns are stored on the tester in a compressed form and then transferred to the chip using the test data bandwidth link as illustrated in Fig. 1.2. It is then

decompressed using special on-chip circuitry and sent to the core under test. A number of test data compression techniques have been proposed in the literature. They differ in terms of the following properties:

- Amount of compression of the test vectors
- Reduction in test time
- Hardware required for decompression
- Requirements on the ATPG software

Chapter 2 of this thesis discusses the different test data compression schemes that have been proposed in the literature. It should be noted that most of the previous work has focused on schemes where the on-chip decompression is done using a dedicated hardware circuit. It can also be done in software using an embedded processor. There are some advantages of doing on-chip test vector decompression in software as compared to hardware-based decompression, especially in the SoC context. First, an existing resource (embedded processor) is utilized thereby reducing the hardware overhead. Note that even if the decompression is done in software, some hardware might still be required for synchronizing the data from the processor to the core under test. Next, more complex compression algorithms can be used if the decompression is done in software. Note that there is a trade-off here between the complexity of the decompression program and the amount of compression obtained. For the same reason, popular data compression algorithms like *gzip* are not suitable for test vector compression even though they achieve very high compression since the space (memory) and time requirements may not be compatible. Finally, a major disadvantage of compression schemes that require hardware for decompression is the loss in coverage due to last minute design changes. If the decompression hardware depends on the deterministic test patterns, and has been designed for a given set of patterns, then any changes in the circuit would change the test patterns

and hence the decompression hardware. It may not be possible to redesign the decompression hardware in all cases and hence this may result in reduced compression ratio and/or loss of fault coverage.

As part of this thesis, we investigate the use of an embedded processor present in the SoC for testing the other cores of the SoC. Specifically, we focus on use of the processor for test vector decompression. Test vector decompression is done in software rather than having special decompression circuit. The basic idea is to transfer a decompression program along with compressed test data from a tester to on-chip memory, and then have the processor execute the program, which decompresses the test vectors and applies them to the core under test. A technique for decompression in software based on matrix operations is presented in chapter 3. Chapter 4 discusses another technique based on pseudo-random linear expansion of the compressed test vectors.

Output response is easy to compress since lossy compression techniques can be used. The idea is to minimize the *aliasing probability* - the likelihood of incorrectly identifying a faulty system as correct or vice-versa. It has been extensively studied in the past and well established techniques have been developed, especially in cases when only the test pass/fail information of the system is required. Unfortunately, even though most of these techniques usually achieve very high compression, the compressed response has little or no diagnostic information.

With the increasing popularity of BIST techniques for testing systems, extracting diagnostic information from test response is becoming even more difficult. The most common method of BIST is the combination of scan-design and BIST called scan-based BIST. In scan-based BIST, the patterns are generated on-chip using a pseudo-random pattern generator (PRPG) and shifted through the scan chains as illustrated in Fig 1.3. The output responses are then captured by the scan chains and compacted on-chip using a multiple input shift register (MISR) to generate a

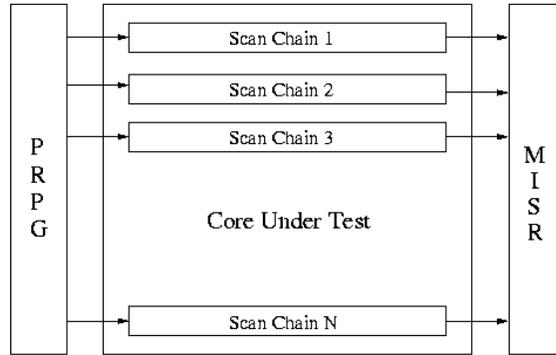


Figure 1.3: Scan-based BIST architecture

single signature. Since a large number of patterns are applied during BIST and the output response is compacted very highly into a signature, it only provides pass/fail information and as such contains very little diagnostic information.

Fault diagnosis is the process of observing the incorrect behavior of the circuit and locating faulty elements in the design that cause the incorrect behavior. Fault diagnosis to determine the location and cause of failures is very important during the initial manufacturing process. It helps in the identification of the manufacturing defects and is also used for yield learning to improve the production quality. Because of the decreasing feature sizes and corresponding increase in the integration densities, the requirements on the fault diagnostic tools are also increasing. Hence fault diagnosis in a BIST environment is an important problem for current technologies. We have developed a test response compression and diagnosis algorithm for scan-based BIST cores that utilizes the embedded processor present in the SoC. If the MISR detects a test fail, then the technique discussed in chapter 5 can be used to get diagnostic information.

An important class of hardware based test vector compression schemes involves using linear operations to decompress the test vectors. These are called linear de-

compressors and include techniques based on linear feedback shift register (LFSR) reseeding and combinational linear expansion circuits consisting of XOR gates. Linear decompression exploits the unspecified (don't care) bit positions in test cubes (i.e., deterministic test vectors where the unassigned bit positions are left as don't cares) to achieve large amounts of compression. The encoding efficiency of the linear decompressor is defined as the ratio of the number of specified bits in the test set to the number of bits stored on the tester. In chapter 6, we describe a technique that can significantly improve the encoding efficiency of a linear decompressor. It is applicable to any linear decompressor including both combinational and sequential.

Entropy is a measure of the disorder in a system. The entropy of a set of data is related to the amount of information that it contains and provides a theoretical bound on the amount of compression that can be achieved using any compression technique. While calculating entropy is well understood for fully specified data, it has not been studied for test data that can have unspecified bits. In chapter 7, we extend the concept of entropy to incompletely specified test data. Using entropy theory, we can derive theoretical limits on the compression that can be achieved by various types of coding techniques. It is also useful to identify the compression techniques that have a lot of room for improvement and offer scope for fruitful research.

This thesis is organized as follows. Chapter 2 gives an overview of the different test vector compression methods proposed in the literature and also the ways an embedded processor has been previously used in testing. Chapters 3 and 4 present test vector compression methods that can be implemented in software. In chapter 3, a matrix-based algorithm is described while in chapter 4, a method based on pseudo-random linear expansion is presented. The dual of the pseudo-random expansion method called pseudo random compaction is presented in chapter 5 as a test response compression and diagnosis algorithm. Chapter 6 presents a technique that can be

used to improve the compression obtained by any linear decompressor. In chapter 7, the entropy limits to test data compression are discussed in detail. Finally, conclusions and possible areas of future research are discussed in chapter 8.

## Chapter 2

# Previous Work

In this chapter, we first discuss the previous work found in literature on test data compression. Next, we look at the various ways a processor has been used to help in testing in the past.

### 2.1 Test Data Compression

Most of the test data compression/decompression schemes proposed in the literature involve designing a special hardware circuit that would do the required decompression on-chip. They can be classified into different categories depending on how the compression is achieved.

The first category of test data compression schemes is based on linear feedback shift register (LFSR) reseeding. These techniques make use of the many unspecified bits in deterministic test patterns. During test pattern generation, not all bits may be assigned to detect a particular fault or set of faults. Given a test cube (test vector with unassigned bits as don't cares), a seed for the LFSR can be computed such that using the seed as the starting state of the LFSR, the given test cube can be generated by running the LFSR in autonomous mode. Linear equations are formed based on the polynomial of the LFSR and then solved to find the seed. This concept was introduced in [Könemann 91] where it was also shown that this could be done with a very high probability given certain constraints on the size

of the LFSR (number of sequential elements) and the number of specified bits in the deterministic test cube. If the maximum number of specified bits in any test cube is  $s_{max}$ , then if the number of stages in the LFSR is  $s_{max} + 20$ , a solution to the system of linear equations can be found with a probability of 0.999999. Hence, instead of storing the full test vector in the tester, a much smaller LFSR seed can be stored thereby achieving compression. Several techniques for improving this basic LFSR reseeding have been proposed by using multiple polynomials [Venkataraman 93], variable length LFSRs [Zacharia 95], two dimensional reseeding [Zacharia 96], partial LFSR reseeding [Krishna 01], and seed compression [Krishna 02].

Another category of test data compression schemes use different coding techniques for compression. A number of schemes have been developed using a variety of codes. Codes can be classified into four categories depending on whether they encode a fixed or variable number of bits in the original data using either a fixed or variable number of bits in the encoded data. Each of the four categories are listed below:

Fixed-to-Fixed Codes: These codes encode fixed size blocks of data using smaller fixed size blocks of encoded data. Conventional LFSR reseeding [Könemann 91] falls into this category where each fixed size test vector is encoded as a smaller fixed size LFSR seed. Techniques that use combinational expanders with more outputs than inputs to fill more scan chains with fewer tester channels each clock cycle fall into this category. These techniques include using linear combinational expanders such as broadcast networks [Hamzaoglu 99] and XOR networks [Bayraktaroglu 01], as well as non-linear combinational expanders [Reddy 02], [Li 03]. If the size of the original blocks is  $n$  bits and the size of the encoded blocks is  $b$  bits, then there are  $2^n$  possible *symbols* (original block combinations) and  $2^b$  possible *codewords* (encoded block combinations). Since  $b$  is less than  $n$ , obviously not all possible symbols can be encoded using a fixed-to-fixed code. If  $S_{dictionary}$  is the set of symbols that can

be encoded (i.e., are in the “dictionary”) and  $S_{data}$  is the set of symbols that occur in the original data, then if  $S_{data} \subseteq S_{dictionary}$ , then it is a complete encoding, otherwise it is a partial encoding. For LFSR reseeding, if  $b$  is chosen to be 20 bits larger than the maximum number of specified bits in any  $n$ -bit block of the original data, then the probability of not having a complete encoding is less than  $10^{-6}$ . The technique in [Reddy 02] constructs the non-linear combinational expander so that it will implement a complete encoding. For techniques that do not have a complete encoding, there are two alternatives. One is to constrain the ATPG process so that it only generates test data that is contained in the dictionary (this is used in [Bayraktaroglu 01]), and the other is to *bypass* the dictionary for symbols that are not contained in it (this is used in [Li 03]). Bypassing the dictionary requires adding an extra bit to each codeword to indicate whether it is coded data or not.

Fixed-to-Variable Codes: These codes encode fixed size blocks of data using a variable number of bits in the encoded data. Huffman codes are in this category. The idea with a Huffman code is to encode symbols that occur more frequently with shorter codewords and symbols that occur less frequently with longer codewords. A method was shown in [Huffman 52] to construct the code in a way that minimizes the average length of a codeword. The problem with a Huffman code is that the decoder grows exponentially as the block size is increased. In [Jas 99, 03], the idea of a selective Huffman code was introduced where partial coding is used and the dictionary is selectively bypassed. This allows larger block sizes to be efficiently used.

Variable-to-Fixed Codes: These codes encode a variable number of bits using fixed size blocks of encoded data. Conventional run-length codes are in this category. A method for encoding variable length runs of 0’s using fixed size blocks of encoded data was proposed in [Jas 98]. The LZ77 based coding method in [Wolff 02] also falls in this category. Note that in [Wolff 02], it is a partial encoding that uses a

bypass mode.

Variable-to-Variable Codes: These codes encode a variable number of bits in the original data using a variable number of bits in the encoded data. Several techniques that use run-length codes with a variable number of bits per codeword have been proposed including using Golomb codes [Chandra 01a], frequency directed codes [Chandra 01b], and VIHC codes [Gonciari 02]. One of the difficulties with variable-to-variable codes is synchronizing the transfer of data from the tester. All of the techniques that have been proposed in this category require the use of a synchronizing signal going from the on-chip decoder back to the tester to tell the tester to stop sending data at certain times while the decoder is busy. Fixed-to-fixed codes do not have this issue because the data transfer rate from the tester to the decoder is constant.

## 2.2 Testing using an Embedded Processor

Previously, embedded processors have been used in testing to perform memory tests using the march algorithm [Saxena 98], [Rajsuman 99]. Simple programs for doing pseudo-random BIST by generating pseudo-random patterns and compacting test responses have been proposed in [Rajski 93], [Gupta 94], [Stroele 95, 96, 98], and [Dorsch 98]. [Huang 02] presents a self-test method for bus-based programmable SoCs, which supports both external testing and mixed-mode BIST. However, the previous research most related to this thesis is the work in using embedded processors for lossless test vector decompression. In [Yamaguchi 97] and [Ishida 98], techniques based on the Burrows-Wheeler transformation and run length encoding are proposed for compressing the test data transferred between a workstation and a tester. While these techniques could also be implemented on an embedded processor for on-chip decompression, they are not well suited for that as the decompression process is complex and time consuming. A deterministic test vector compression technique

based on geometric shapes is proposed in [Maleh 01]. This technique provides very good compression, but the software decompression process is complex and cannot be efficiently implemented for fast on-chip decompression. In [Jas 02], a very simple compression scheme is described which is suitable for fast on-chip decompression. Each test vector is divided into blocks, and only the blocks that are different from the preceding test vector are stored. The test vectors are then constructed on the fly by a program running on the embedded processor. In [Hwang 02b], a BIST technique composed of both random pattern testing and deterministic pattern testing using the embedded processor is proposed. The processor generates random patterns and selectively applies only those patterns that contribute to the fault coverage. The deterministic patterns are compressed by encoding the differences between them and similar random patterns. Though the number of random vectors that need to be applied is reduced, the total number of vectors is still much greater than the number of vectors in a fully deterministic test set.

## Chapter 3

# Matrix-based Compression

In this chapter, we describe a compression/decompression scheme for deterministic test patterns with unspecified bits based on matrix operations [Balakrishnan 02]. The compression scheme presented here is very simple yet produces effective results and can be efficiently implemented on a processor.

The proposed scheme is based on the decomposition of a matrix into two vectors based on a relation that we define below. The operation  $\mathbf{A} \tilde{\oplus} \mathbf{B}$  between two boolean vectors  $\mathbf{A} = [a_1, a_2, a_3 \dots a_n]$  and  $\mathbf{B} = [b_1, b_2, b_3 \dots b_n]$  where  $a_i, b_i \in \{0, 1\}$  is defined as shown in Fig. 3.1. Note that this is very similar to matrix multiplication except that the elements in the product matrix are defined differently ( $a_i \oplus b_i$  instead of  $a_i \bullet b_i$ ). This helps increase the chances of decomposition since the *XOR* operation puts less constraints on the inputs than *AND*.

$$\mathbf{A} \tilde{\oplus} \mathbf{B} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tilde{\oplus} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_1 \oplus b_1 & a_1 \oplus b_2 & a_1 \oplus b_3 \\ a_2 \oplus b_1 & a_2 \oplus b_2 & a_2 \oplus b_3 \\ a_3 \oplus b_1 & a_3 \oplus b_2 & a_3 \oplus b_3 \end{bmatrix}$$

Figure 3.1: Matrix operation  $\mathbf{A} \tilde{\oplus} \mathbf{B}$

In this way, any  $n \times n$  matrix can be represented with the two vectors  $\mathbf{A}$  and  $\mathbf{B}$  and the operation  $\mathbf{A} \tilde{\oplus} \mathbf{B}$ . This decomposition can be realized by solving a simul-

taneous set of equations in the variables  $a_i, b_i$ . The set of equations is represented in the matrix format ( $\mathbf{Ax}=\mathbf{b}$ ) as shown in Fig. 3.2. If and only if a solution of this set of equation exists, the given matrix can be decomposed.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \\ \mathbf{v}_5 \\ \mathbf{v}_6 \\ \mathbf{v}_7 \\ \mathbf{v}_8 \\ \mathbf{v}_9 \end{bmatrix}$$

Figure 3.2: Set of linear equations for the decomposition

### 3.1 Compression Algorithm

We propose a compression scheme for test vectors based on the above. This involves writing  $n^2$  bits of the test vector as an  $n \times n$  matrix  $\mathbf{M}$ . The matrix  $\mathbf{M}$  is then decomposed by solving the set of linear equations. Although the decomposition is not always possible, the unspecified bits in the test vectors increase the chances of decomposition. This is because only equations for the specified bits of the test vector need to be satisfied. The more unspecified bits there are, the fewer the number of equations and hence less constraints on the variables. Several different heuristics can be applied to form the matrix  $\mathbf{M}$  that needs to be decomposed from the given set of test vectors. These vary according to the complexity of the decompressing process. Three of these are discussed below.

### 3.1.1 Single Size Decomposition (SSD)

This is the simplest method to form the matrix  $M$  and also the easiest to decode. The first  $n^2$  bits of the test vector are written as an  $n \times n$  matrix with the first  $n$  bits being the first row of the matrix and the next  $n$  bits the next row and so on and so forth as illustrated in Fig. 3.3.

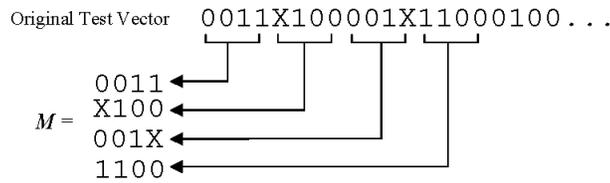


Figure 3.3: Formation of matrix  $M$  with  $n = 4$

The choice of the size of the matrix (i.e.,  $n$ ) would depend on the word size of the processor. If the matrix  $M$  thus formed cannot be decomposed for the next  $n^2$  bits of the test vector, we store the first  $n$  bits as they are (i.e. uncompressed) and then proceed with the algorithm for the next  $n^2$  bits after that. Hence we need one bit at the start of every set of bits to indicate whether the bits are compressed or not. In this method each test vector is compressed separately and hence the ordering of the test vectors will not make a difference to the compression obtained.

### 3.1.2 Multiple Size Decomposition (MSD)

A simple optimization of the previous method would be to try to form the matrix  $M$  with multiple sizes. The largest size is tried first since that gives the maximum compression. If the matrix for the largest size is not decomposable, then the next size is tried and so on. If none of the three sizes of the matrix are decomposable, then the next set of  $m$  bits are left uncompressed and the algorithm

is tried on the successive bits of the test vector. If the number of different size is  $k$ , then  $\lceil \log_2(k + 1) \rceil$  additional bits are needed to encode the  $k + 1$  cases that can occur indicating whether the succeeding bits have been compressed by any of the different sizes or left uncompressed.

### 3.1.3 Multiple Vector Decomposition (MVD)

The matrix  $M$  can also be formed from multiple test vectors as illustrated in Fig. 3.4. In this case, each row of the matrix would correspond to a different test vector. This is a better alternative than the earlier ones since there is usually a lot of similarity between test vectors due to the structural relationship among the faults that are detected by these test vectors. Furthermore, the test vectors can be ordered in an optimal way such that the chances of decomposition of the matrix are increased. A limiting factor of this method is that the decoding is more complex. The partial test vectors constructed after decoding each matrix cannot be directly applied to the core-under-test until a sufficient number of matrices have been decoded to get the complete test vector.

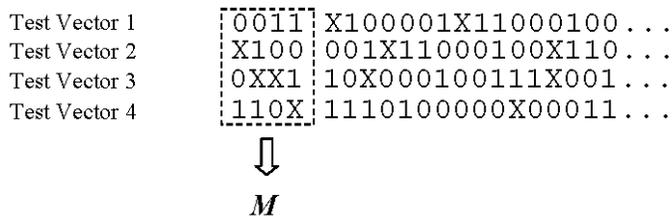


Figure 3.4: Formation of matrix M using MVD

The amount of compression obtained in this scheme depends on the ordering of the test vectors. Since it is impractical to examine all possible orderings (a set of  $n$  test vectors has  $n!$  different ways of ordering), we have used a simple greedy reordering heuristic referred to as the hill climbing approach in our experiments. In

the hill climbing approach, initially the given order of test vectors is used to calculate the compression. The positions of two vectors are then randomly exchanged and the new compression calculated. If the compression is better, the new order is saved; else the old order maintained. This process is continued until no better compression is obtained for a specific number of exchanges.

### 3.2 Decompression Using an Embedded Processor

An embedded processor present in the SOC can be used to efficiently decompress the compressed data and send it to the core-under-test (CUT). This is illustrated in Fig. 3.5. An external tester supplies the compressed data while a software program running on the embedded processor decodes the test data. The tester loads the test data into a specific set of addresses of the system memory through the memory I/O controller. The tester also writes to a given location to indicate the end of the current test vector. The processor reads the data from the corresponding locations in memory and decompresses it accordingly. Depending on the number of scan chains in the core, the processor either sends the data directly to the core or stores it back to memory so that it can apply the data to the core when it has a sufficient amount of decompressed test data. If the end of the test vector is reached, it sends an instruction to apply a capture cycle to capture the response into the scan chain. The response is shifted out into a multi-input shift register (MISR) for compaction as the next test vector is shifted into the core.

In general, the speed at which test data is transferred to the SOC by the tester will be much lower than the operating frequency of the embedded processor. Two potential problems could arise because of this discrepancy. If the processor is able to process the written data before the tester loads new data into the memory location, appropriate *NOPs* need to be inserted into the decompression program to make sure that next time the processor reads the memory location it has the new data. The

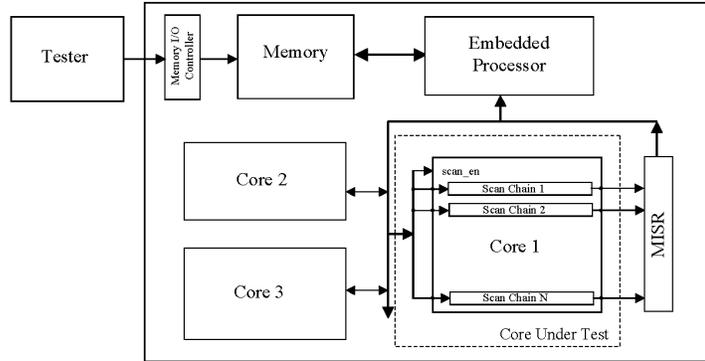


Figure 3.5: Example of test architecture

second problem arises if the tester overwrites new data to a memory location before the processor is able to process the old data in it. This can be taken care of by inserting *NOPs* at appropriate places into the tester program to slow it down.

Figure 3.6 shows the *pseudo-code* of the test procedure for the multiple size decomposition (MSD) heuristic when three different sizes ( $n_1, n_2, n_3$ ) are used. In this case, the first two bits denote which of the four cases apply to the next set of bits. Hence, we check for them and proceed accordingly. The macro *Apply* writes the test vector to the appropriate core. It will depend on the number of scan chains in the core and the width of the bus connecting the processor to the core. The macro will exit after receiving a signal that the decompressed vectors have been scanned into the core and the program proceeds to decompress the next set of bits. After each test vector is decompressed and sent to the scan chains, a capture signal is given to the core-under-test by the macro *Capture*.

The implementation of decompression algorithm can be made more efficient by appropriately choosing the matrix sizes according to the word size of the embedded processor. For single size decomposition (SSD), the matrix size,  $s$ , should be such that  $s + 1$  is equal to the word size of the processor. This is because we store

either  $2s + 1$  bits (if matrix can be decomposed) or  $s + 1$  bits (if matrix cannot be decomposed). In the case of multiple size decomposition (MSD), the three different sizes,  $s_1$ ,  $s_2$ ,  $s_3$  and the number of bits left uncompressed,  $m$ , should be such that  $2s_1 + 2$ ,  $2s_2 + 2$ ,  $2s_3 + 2$ , and  $m + 2$  are multiples of the word size of the processor. This would minimize the number of reads from the system memory (since each set of bits will be at word boundaries) and hence reduce the decompression time.

### 3.3 Analysis

The number of cycles that the decompression program runs,  $n$ , can be determined by studying the time complexity of the different schemes. For example, in the single size decomposition heuristic, this time will depend on the number of solvable and unsolvable matrix decompositions during compression. If the number of solvable and unsolvable decompositions are  $n_s$  and  $n_u$  respectively, then the number of cycles will be  $n = n_s c_1 + n_s s c_2 + n_u c_3 + c_4$  where  $s$  is the size of the matrix and  $c_1, c_2, c_3$ , and  $c_4$  are constants depending on the instruction set. These constants are essentially the number of cycles it takes for the processor to read from the memory, perform XOR operations and to write to the core-under-test. We performed simulations on the ISCAS'89 [Brglez 89] benchmark circuits to measure the number of cycles required for decompression. The decompression program for the single size decomposition with size  $s = 7$  was written in 'C' language and implemented on the ARM7 architecture. The SimpleScalar toolset was then used to run the program and estimate the number of processor clock cycles. Table 3.1 shows the number of solvable and unsolvable decompositions and the number of clock cycles obtained by simulation for the ISCAS'89 benchmark circuits. The values for the constants  $c_1, c_2, c_3$ , and  $c_4$  can be calculated for the ARM7 from the simulated results. A similar analysis can be done for the other two heuristics to get the time complexity of the proposed schemes.

Table 3.1: Simulated values for the number of cycles

Circuit	No. of Solvable	No. of Unsolvable	Number of cycles
s5378	437	769	24013
s9234	583	1204	29373
s13207	3232	1580	99430
s15850	1264	1475	47870
s38417	2350	4523	84569
s38584	3140	4185	104433

In general, the rate at which the tester transfers test data to the SoC will be different from the rate at which the embedded processor processes the compressed test data. The former depends on the tester clock cycle and the number of channels from the tester to SoC, while the latter depends on the operating frequency of the processor and its instruction set architecture. Two potential problems could arise because of this discrepancy. If the processor is able to process the written data before the tester loads new data into the memory location, it has to wait until the tester writes new data into the location. The second problem arises if the tester overwrites new data to a memory location before the processor is able to process the old data in it. These problems can be taken care of by inserting *NOPs* at appropriate places into the decompression program or the tester program to slow it, but this solution will result in an increase in the test time. Alternatively, by choosing the size of the buffer (set of memory locations where the tester writes the data in a cyclic fashion) and the start time for the processor appropriately, it may be possible to circumvent this problem. In this case, there is no need for synchronization between the tester and the processor – the tester keeps on writing to the buffer at its own speed while the processor keeps on decompressing data at its own rate. The analysis for calculating the buffer size for the single size decomposition is given below.

The number of cycles the processor takes to process each compressed and each uncompressed word can be determined through simulation. For the ARM7 processor, the number of cycles were 4 for each uncompressed word and 26 for each compressed set (two words) when the matrix size was equal to 7. This means if the current word is an uncompressed word, the processor will access the next word after 4 cycles or if the current word is a compressed word, the processor will access the word subsequent to the next word after 26 cycles (since it uses two words together if they are compressed). Using this and the trace of compressed/uncompressed words, the cycles in which the processor will access the buffer can be determined. If each word in buffer is guaranteed to be valid when the processor accesses it, then there is no need for explicit synchronization. Each word will be valid during the time interval between end of the previous loading of the tester and start of the next loading of the tester. This time interval will depend on the buffer size and the data transfer rate of tester. The minimum buffer size for which each accessed word is valid (with an appropriate start time for the processor) can be calculated for different tester data transfer rates.

Table 3.2: Buffer sizes for different data transfer rates of the tester

Circuit	Total Data	Tester data transfer rate (bits per processor clock cycle)				
		0.5	0.75	0.875	1.0	1.25
s5378	1643	743	298	237	340	834
s9234	2370	1124	485	367	390	1019
s13207	8044	2399	1440	2347	3505	7122
s15850	4033	1583	579	724	1112	2503
s38417	9223	4276	1718	1266	1489	4164
s38584	10465	4319	1766	1969	3069	6546

Table 3.2 shows the buffer sizes required for each of the ISCAS'89 benchmark circuits for different tester speeds with 7 as the size of the matrix for decomposition. The column labeled "total data" is the total number of bytes in the compressed set.

This is shown to compare the required buffer sizes with the total amount of memory required to store the entire compressed test set. The other columns show the size of the buffer (in bytes) required so that no synchronization is needed between tester and processor. The tester data transfer rate of 0.5 means the tester transfers 0.5 bits per processor clock cycle (i.e., 1 bit every 2 processor clock cycles). The tester transfer rate depends on the tester clock speed and the number of channels. Note from the results in Table 3.2 that the buffer size first decreases with an increase in tester rate and then increases again. If the tester is too slow, a larger buffer is required and the tester needs to start writing much earlier than the processor can start processing. If the tester is too fast, again a larger buffer is required since the tester must not overwrite memory before it is processed.

### 3.4 Experimental Results

The proposed compression schemes were implemented and experiments were performed on the larger ISCAS'89 and ITC'99 benchmark circuits. The following sections discuss the results in detail.

#### 3.4.1 Program Size

Table 3.3: Program sizes of different schemes

Program Size	SSD	MSD	MVD
instr.	100	294	475
bytes	400	1176	1900

In software based testing, the decompression program also needs to be transferred to the chip. Hence, the program size is required to calculate the total amount of data that needs to be transferred. In these experiments, the decompression pro-

gram was implemented in ‘C’ language and cross-compiled for the ARM7 architecture. Table 3.3 shows the program size for the three heuristics. As expected, the single size decomposition requires the least number of instructions to decode. The multiple vector decomposition heuristic requires the maximum number of instructions among the three heuristics.

### 3.4.2 Data Compression

Test cubes that provide 100% coverage of detectable faults were generated using the SynTest<sup>TM</sup> commercial ATPG tool for each circuit. The unspecified input assignments were left as X’s for better compression. The three heuristics described in Sec. 2 were used to compress the test set. The percentage compression is computed as:

$$\text{Percentage Data Compression} = \frac{\text{Original Bits} - \text{Compressed Bits}}{\text{Original Bits}} \times 100$$

Table 3.4 shows the compression obtained using the single size decomposition for three different matrix sizes. The first four columns have the details of the benchmark circuit including circuit name, number of scan elements, number of test vectors in the test set, and total number of bits in the test set. The remaining columns show the compressed test set size and the corresponding percentage compression for the three different sizes of the matrix. The matrix size under which maximum compression is obtained depends on the size of the circuit. As can be seen from the table, for the smaller circuits, a matrix size of  $n = 7$  gives maximum compression, while for some of the larger circuits,  $n = 10$  gives better compression. But as  $n$  increases further, the compression decreases since very few of the matrices are decomposable.

Table 3.5 compares the compression obtained using the multiple size decomposition heuristic with three sizes for three different configurations. The first configuration has the three matrix sizes  $n_1 = 15$ ,  $n_2 = 7$ ,  $n_3 = 3$  and the number of bits left uncompressed,  $m = 14$ . The next configuration has sizes  $n_1 = 31$ ,  $n_2 = 15$ ,  $n_3 = 7$  and

Table 3.4: Compression obtained using single size decomposition (SSD)

Circuit	Scan Size	Test Vect.	Orig. Bits	n = 7		n = 10		n = 13	
				Comp. Bits	% Comp.	Comp. Bits	% Comp.	Comp. Bits	% Comp.
s5378	214	119	25466	12707	<b>50.1</b>	16026	37.1	22869	10.2
s9234	247	147	36309	18377	<b>49.4</b>	24386	32.8	31925	12.1
s13207	700	239	167300	61120	63.5	48766	<b>70.9</b>	52596	68.6
s15850	611	120	73320	30760	<b>58.1</b>	35842	51.1	45997	37.3
s38417	1664	95	158080	71434	<b>54.8</b>	98904	37.4	141437	10.5
s38584	1464	131	191784	80580	<b>58.0</b>	94300	50.9	127191	33.7
b14s	281	110	30910	14000	<b>54.7</b>	17865	42.2	20977	32.1
b15s	489	240	117360	41041	65.0	36606	<b>68.8</b>	41986	64.2
b17s	1456	233	339248	120124	64.6	112454	<b>66.9</b>	135822	60.0
b20s	526	325	170950	70980	<b>58.5</b>	86720	49.3	108800	36.4
b21s	526	325	170950	80170	<b>53.1</b>	109764	35.8	153395	10.3
b22s	771	324	249804	99774	<b>60.1</b>	111829	55.2	138174	44.7

Table 3.5: Compression obtained using multiple size decomposition (MSD)

Circuit	Original Bits	$n_1 \equiv 15, n_2 \equiv 7, n_3 \equiv 3$ and $m \equiv 14$		$n_1 \equiv 31, n_2 \equiv 15, n_3 \equiv 7$ and $m \equiv 14$		$n_1 \equiv 23, n_2 \equiv 15, n_3 \equiv 7$ and $m \equiv 14$	
		Comp. Bits	Percent Comp.	Comp. Bits	Percent Comp.	Comp. Bits	Percent Comp.
s5378	25466	12504	<b>50.9</b>	13216	48.1	13264	47.9
s9234	36309	19480	<b>46.3</b>	20880	42.5	20976	42.2
s13207	167300	40512	75.8	29088	<b>82.6</b>	35504	78.8
s15850	73320	26680	<b>63.6</b>	27584	62.4	29200	60.2
s38417	158080	72864	<b>53.9</b>	82240	48.0	82640	47.7
s38584	191784	72048	<b>62.4</b>	74800	61.0	75360	60.7
b14s	30910	13560	<b>56.1</b>	13856	55.2	14352	53.6
b15s	117360	31592	73.1	24432	<b>79.2</b>	24976	78.7
b17s	339248	84880	75.0	75360	77.8	75152	<b>77.9</b>
b20s	170950	65744	61.5	63040	<b>63.1</b>	65488	61.7
b21s	170950	86440	<b>49.4</b>	90928	46.8	92016	46.2
b22s	249804	84312	66.2	82992	66.8	82576	<b>67.0</b>

$m = 14$ , while the last configuration has sizes  $n_1 = 23$ ,  $n_2 = 15$ ,  $n_3 = 7$  and  $m = 14$ . These sizes have been chosen such that the decompression program can fetch either one byte (8 bits) or multiples of bytes from system memory and operate on it.

Table 3.6: Compression obtained using multiple vector decomposition (MVD)

Circuit	Original Bits	$N = 4$		$N = 6$		$N = 8$	
		Comp. Bits	Percent Comp.	Comp. Bits	Percent Comp.	Comp. Bits	Percent Comp.
s5378	25466	11628	54.3	10374	59.3	9630	<b>62.2</b>
s9234	36309	17206	52.6	16858	53.6	15986	<b>56.0</b>
s13207	167300	59970	64.2	44790	73.2	37256	<b>77.7</b>
s15850	73320	30018	59.1	26292	64.1	24408	<b>66.7</b>
s38417	158080	73006	53.8	68496	<b>56.7</b>	68576	56.6
s38584	191784	76100	60.3	66092	<b>65.5</b>	66534	65.3
b14s	30910	13982	54.8	12680	59.0	12096	<b>60.9</b>
b15s	117360	44052	62.5	34673	70.5	29796	<b>74.6</b>
b17s	339248	132820	60.8	106988	68.5	97502	<b>71.3</b>
b20s	170950	72100	57.8	66274	61.2	64284	<b>62.4</b>
b21s	170950	83198	51.3	80722	<b>52.8</b>	81980	52.0
b22s	249804	100508	59.8	88818	64.4	88690	<b>64.5</b>

In Table 3.6, the compression obtained using multiple vector decomposition heuristic is presented. Three different values for the number of vectors compressed together,  $N$ , were tried. The hill climbing approach discussed in Section 3.1.3 was implemented. In general, the compression obtained increases with the number of vectors compressed together. Compressing eight vectors at a time ( $N = 8$ ) gives better compression for most of the circuits. However, the complexity of the decompression program will increase with  $N$ . The amount of on-chip memory required for decompression will also increase since  $N$  test vectors need to be stored at a time.

A comparison of the best compression obtained using the three different heuristics is given in Table 3.7. From the results, it can be seen that the multiple vector decomposition scheme has the best compression ratio for most of the circuits but it

Table 3.7: Comparison of the three methods

Circuit	Original Bits	Percentage Compression		
		SSD	MSD	MVD
s5378	25466	50.1	50.9	62.2
s9234	36309	49.4	46.3	56.0
s13207	167300	70.9	82.6	77.7
s15850	73320	58.1	63.6	66.7
s38417	158080	54.8	53.9	56.7
s38584	191784	58.0	62.4	65.5
b14s	30910	54.7	56.1	60.9
b15s	117360	68.8	79.2	74.6
b17s	339248	66.9	77.9	71.3
b20s	170950	58.5	63.1	62.4
b21s	170950	53.1	49.4	52.8
b22s	249804	60.1	67.0	64.5

is also the most complex to decode among the three methods discussed. The test application time will be the longest in this case since the test vectors need to be constructed completely and stored in the system memory before they can be applied to the core. The multiple size decomposition heuristic obtains a good amount of compression and is relatively simpler to decode. The choice of which method to apply will depend on the operating conditions of the test architecture. If test application time is critical, multiple size decomposition is a better alternative.

In Table 3.8, the compression results for the proposed scheme are compared with the geometric primitives based compression technique described in [Maleh 01] and the difference vectors based compression described in [Jas 02]. As seen from the table, the compression obtained by the proposed scheme is higher than both of the earlier methods for almost all of the circuits.

Table 3.8: Comparison with other techniques

Circuit	[Maleh 01]	[Jas 02]	Prop. Scheme
s5378	51.6	49.1	62.2
s9234	43.5	49.3	56.0
s13207	85.0	85.5	82.6
s15850	60.9	66.8	66.7
s38417	46.6	38.6	56.7
s38584	-	53.9	65.5

### 3.5 Summary

In this chapter, we presented a compression method based on matrix operations. The main advantages of this method vis-a-vis the previous works are the simplicity of the algorithm and the efficiency of implementation of the decompression program. Even though much more complex algorithms can be implemented in software, the space (amount of memory) and time requirements of the decompression algorithm are the limiting factors for practical implementation of such algorithms. The next chapter presents compression technique that is based on linear operations and is also simple to implement.

```

Data: numVectors := number of testvectors
      ScanSize := size of the scan chain
      n1, n2, n3 := the three matrix sizes
      m := number of uncompressed bits

Test_procedure() {
  while(vectorsProcessed < numVectors)
    while(bitsProcessed < scanSize)
      readNextMemoryLocation();
      /* check the first two bits */
      case 00:
        A[1.. n1] = next n1 bits
        B[1.. n1] = next n1 bits
        for (i = 1, n1){
          M[i][0.. n1] = A[i]⊕B[0.. n1];
          Apply(M[i][0.. n1]);
        }
      end
      case 01:
        A[1.. n2] = next n2 bits
        B[1.. n2] = next n2 bits
        for(i = 1, n2){
          M[i][0.. n2] = A[i]⊕B[0.. n2];
          Apply(M[i][0.. n2]);
        }
      end
      case 10:
        A[1.. n3] = next n3 bits
        B[1.. n3] = next n3 bits
        for(i = 1, n3){
          M[i][0.. n3] = A[i]⊕B[0.. n3];
          Apply(M[i][0.. n3]);
        }
      end
      case 11:
        M[0..m] = next m bits
        Apply(M[0..m]);
      end
    end while
    Capture();
    vectorsProcessed++;
  end while
}

```

Figure 3.6: Pseudo-code for decompression algorithm

## Chapter 4

# Pseudo-Random Linear Expansion

In this chapter, we present a compression/decompression scheme based on pseudo-random linear expansion of compressed test vectors [Balakrishnan 03a]. Before we introduce the proposed scheme, we take a look at two methods that are closely related where LFSR reseeding schemes are implemented in software for decompressing deterministic test cubes. An LFSR reseeding scheme using multiple polynomials was described in [Hellebrand 92]. This scheme improves the compression ratio by reducing the storage requirements for each test cube to  $s_{max} + 1$  ( $s_{max}$  is the maximum number of specified bits in any test cube). A software implementation of this scheme was described in [Hellebrand 96] for use on an embedded processor. While this approach is effective, it has a number of drawbacks:

- The storage requirement for each test cube is always  $s_{max} + 1$  regardless of the size of the test cube. This limits the encoding efficiency and hence compression.
- Static compaction during ATPG has to be constrained so that  $s_{max}$  doesn't become too large. This generally results in a significant increase in the size of the test set.
- The decompression program cannot be easily reused for different test sets (e.g., if different cores of the SOC are to be tested), since it requires multiple primitive polynomials of degree equal to  $s_{max}$ , and  $s_{max}$  will likely be different for each test set.

The method in [Zacharia 96] performs two-dimensional LFSR reseeding to generate a group of test cubes at a time. By grouping the vectors, the  $s_{max}$  for the groups can be made more balanced thereby improving the encoding efficiency. It uses multiple LFSRs with the same feedback polynomial so it can be processed in parallel using word-based operations.

The software-based compression scheme presented in this chapter, like LFSR reseeding, is also based on expanding compressed data using linear operations. However, it avoids many of the drawbacks of LFSR reseeding and can be implemented much more efficiently in software. The proposed approach performs pseudo-random word-based linear operations on compressed test data to decompress the test vectors. The compressed test data is obtained by solving linear equations for the specified bits in the test set. The advantages of the proposed technique include the following:

- The storage requirements are independent of  $s_{max}$ . They depend only on the total number of specified bits in the test set thereby providing a greater encoding efficiency. Moreover, no restrictions are placed on static compaction or the ATPG process as a whole. Both of these factors result in better compression.
- The proposed approach uses fewer operations for decompression and thus can be executed more efficiently on a processor with fewer instructions. This results in less test time.
- There is no need for primitive polynomials. This allows easy reuse of the same decompression program for multiple test sets.

## 4.1 Overview of Proposed Scheme

The basis for the proposed compression/decompression method is similar to that used in LFSR reseeding [Könemann 91]. In LFSR reseeding, an LFSR seed is expanded using linear operations to form a test vector. Each bit in the test vector

can be represented by a linear equation in terms of the bits of the LFSR seed. Note that many of the bits in the test vector are don't cares, and thus the equations only need to be solved for specified bits. If  $s$  is the number of specified bits and  $m$  is the size of the LFSR, then a system of linear equations,  $\mathbf{Ax}=\mathbf{b}$ , can be formed where  $\mathbf{A}$  is the coefficient matrix with  $s$  rows and  $m$  column,  $\mathbf{x}$  is a vector corresponding to the bits of the seed, and  $\mathbf{b}$  is a vector corresponding to the specified bits in the test vector. The coefficient matrix  $\mathbf{A}$  depends on the polynomial of the LFSR.

In the proposed method, the compressed data is expanded using word-based linear operations. A pseudo-random generator (e.g., the Mitchell and Moore additive generator [Knuth 97] which can very efficiently be implemented in software) is used select words from the compressed data that are XORed together to form each word of the test set (see example in Fig. 4.1). Each bit of the test set can be represented by a linear equation in terms of the bits in the compressed data. If  $s_{tot}$  is the total number of specified bits in the test set and  $m$  is the number of bits in the compressed data, then a system of linear equations,  $\mathbf{Ax}=\mathbf{b}$ , can be formed where  $\mathbf{A}$  is the coefficient matrix with  $s_{tot}$  rows and  $m$  column,  $\mathbf{x}$  is a vector corresponding to the bits of the compressed data, and  $\mathbf{b}$  is a vector corresponding to the specified bits in the test set. In this case, the coefficient matrix  $\mathbf{A}$  depends on the sequence of numbers generated by the pseudo-random number generator.

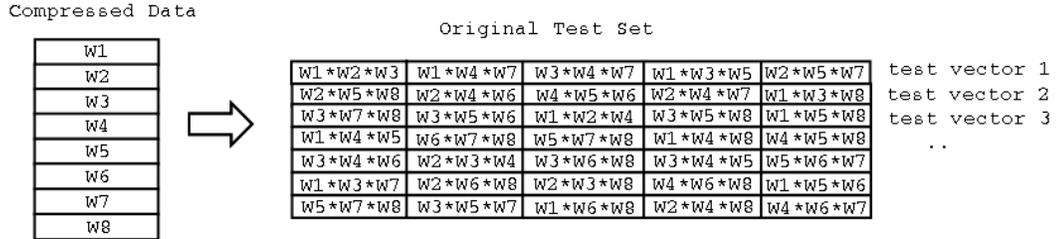


Figure 4.1: Forming test set from compressed bits

The nice features of the proposed method compared with LFSR reseeding are the following: there is no need for primitive polynomials, so it can be easily scaled

and adapted for any test set, the word-based linear operations can be efficiently executed on an embedded processor with a single XOR instruction, and the equations are solved across the entire test set so there is no dependence on the maximum number of specified bits,  $s_{max}$ , in any single test vector thereby allowing better compression.

In order to make the proposed compression method work, we need to be able to solve the system of linear equations for the care bits in the test set. If the rows of the coefficient matrix  $\mathbf{A}$  are linearly independent, then the set of equations will definitely have a solution. We will now discuss what needs to be done to ensure the rows are linearly independent. Suppose the coefficient matrix  $\mathbf{A}$  is generated randomly, then Theorem 4.1 below applies.

**Theorem 4.1:** The probability that an  $n \times m$  matrix ( $m > n$ ) with entries chosen randomly from  $\{0,1\}$  has all  $n$  rows independent is given by

$$\Pr [\text{rows are independent}] > 1 - 2^{-(m-n)}$$

*Proof:* Define the event  $A_i$  to be the bad event that the first  $i - 1$  rows are linearly independent but the  $i^{th}$  row is linearly dependent on the first  $i - 1$  rows.

Claim:  $\Pr [A_{i+1}] = 2^i / 2^m$ .

Assume  $i$  linearly independent rows. There are  $2^i$  possible linear combinations of these rows. Moreover, each of these  $2^i$  linear combinations produces distinct vectors. If not, then there exists a distinct set of coefficients  $\{a_j\}$  &  $\{b_j\}$  which when applied to the row vectors  $\{x_j\}$  yields:

$$\sum_{j=1}^i a_j x_j = \sum_{j=1}^i b_j x_j \Rightarrow \sum_{j=1}^i c_j x_j = 0, \quad c_j = a_j \oplus b_j$$

Since  $\{c_j\}$  is not all zero as  $\{a_j\}$  &  $\{b_j\}$  are distinct, this contradicts our assumption that these rows are linearly independent. Hence there are  $2^i$  distinct linear combinations. The  $(i + 1)^{th}$  row will be dependent on the first  $i$  rows if it is equal to any

of the linear combinations. Therefore,  $\Pr [A_{i+1}] = 2^i / 2^m$  when the rows are chosen uniformly at random from a set of  $2^m$  possibilities. Now,

$$\Pr \left[ \bigcup_{i=1}^n A_i \right] \leq \sum_{i=1}^n \Pr [A_i] = \sum_{i=0}^{n-1} \frac{2^i}{2^m} = \frac{(2^n - 1)}{2^m} < 2^{(n-m)}$$

Therefore,  $\Pr [\text{Rows are independent}] \geq 1 - \Pr \left[ \bigcup_{i=1}^n A_i \right] > 1 - 2^{-(m-n)} \quad \square$

For a randomly generated coefficient matrix  $\mathbf{A}$ , if we want to make the probability of a solution not existing negligible, say 1 in a million, then based on the above, the number of columns should be 20 more than the number of rows. So if the number of rows is  $s_{tot}$ , then the number of columns should be  $s_{tot} + 20$ .

In the proposed method, the coefficient matrix,  $\mathbf{A}$ , is not truly random because a limited number of words are XORed together to form each word in the test set. However, the proposed method approximates a random coefficient matrix, and thus the number of columns (which is the number of words of compressed data times the word size) will be roughly  $s_{tot} + 20$  though somewhat more depending on how many XOR operations are performed when forming each word. Our experimental results (presented in Sec. 4.3) validate this.

## 4.2 Details of Proposed Method

The previous section described the main ideas of the proposed method. This section discusses some of the details.

If there are  $M$  words in the compressed data and  $N$  words in the original test set, the decompression procedure is as follows:

1. Generate a pseudo-random number, *rotateby*, between 0 and the word size.
2. Generate  $k$  random numbers from 1 to  $M$ .
3. For each random number generated in Step 2, select the word of the compressed set that is indexed by it and rotate the selected word by *rotateby*.

4. XOR all the words in Step 3 together to get the next word of the original test set.
5. Continue doing steps 1 to 4 until all the words in the original test set have been generated.

Note that the procedure rotates each selected word by a random number and then XORs them. This is done to mix up the bits and remove the dependency on the bit position. To calculate the  $M$  words in the compressed set, a system of linear equations is formed and solved for the specified bits in the test set. The linear equations are formed by applying the decompression process using variables to represent the bits in the compressed words. If no solution can be found for the system of linear equations, then a larger value for  $M$  can be used, or a different value of  $k$  can be tried. If  $M$  is made sufficiently large, a solution can always be found.

Note that it may not be possible to process the entire test set altogether. There are two factors which may limit the number of test vectors that can be encoded with one set of compressed data. The first is that the amount of on-chip memory may be too small to hold the amount of compressed data required for the entire test set. The second is that the system of linear equations may become too large to solve in a reasonable amount of time. If these factors become an issue, then the test set can be simply partitioned and each partition processed one at a time. Partitioning the test set will reduce the overall compression slightly (the larger the partitions, the better the overall compression).

Using the proposed method to decompress the test vectors, as each word of the test vectors is decompressed; it can be directly applied to the scan chains of the CUT. One easy way for the processor to transfer the data to the scan chains is to make the scan chains a memory-mapped I/O port.

A software program for implementing the proposed method is shown in Fig.

4.2. The data required for the test program is the size of the scan vectors in the core-under-test (*scanSize*), the number of test vectors (*numVectors*), the number of words that need to be XORed to get one word (*numXors*), the seed of the random number generator (*randSeed*), and the number of words in the compressed set (*numWords*) and the compressed set itself (*compBits*).

---

```

Data: numXors   = Number of words that are XORed together
      numWords  = Total number of compressed words
      randSeed  = Seed of the random number generator
      scanSize  = Number of words in scan vector
      numVectors = Total number of test vectors
      compBits  = The compressed test vectors

Test_Procedure(){
/* Read numXors, numWords, randSeed, scanSize, numVectors */
InitiateRandomGenerator(randSeed,numWords);
  for i = 1 to numVectors do
    for j = 1 to scanSize do
      word = 0;
      rotateby = (GenerateRandomNumber())mod 32;
      for k = 1 to numXors do
        rand = GenerateRandomNumber();
        tempword = compBits[rand];
        tempword = tempword ROT rotateby;
        word = word XOR tempword;
      end
      Apply(word);
    end
    Capture();
  end
}

```

---

Figure 4.2: Software program for proposed method

The random number generator used is the Mitchell and Moore additive generator [Knuth 97]. The generator produces a random number in  $Z_m = \{0, 1, 2, \dots, m - 1\}$  given by the equation:

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \quad n \geq 55$$

This was shown to have good random characteristics [Knuth 97] and can be

implemented very fast on processors [Hwang 02a]. The random number generator is initialized with values given by the random seed in *InitiateRandomGenerator* as shown in Fig. 4.3. The test program generates one word of a test vector at a time. For every word,  $(1+numXors)$  random numbers need to be generated. The first random number is *rotateby*, the amount of rotation. The rest of the random numbers denote the index of the selected word in the compressed set. Each selected word is first rotated according to *rotateby* and then XORed to get one word of the test vector. This word is then sent to the appropriate core and shifted into the scan chains. After one full test vector is generated and shifted, it is applied to the core.

```

InitiateRandomGenerator(rand_seed,num_words){
  for i = 1 to 55 do
    randset[i] = ( rand_seed + i ) % num_words;
  end
}
GenerateRandomNumber() {
  randset[i] = (randset[i-55] + randset[i-24]) % num_words;
  return(randset[i]);
}

```

Figure 4.3: Random number generation

The total test time for the test program shown in Fig. 4.2 will depend on the circuit parameters, e.g., number of test vectors, number of scan bits, etc. If we optimize the innermost loop that generates one word, there will be a corresponding decrease in the total test time. Figure 4.4 below shows a piece of assembly code for the loop that is optimized for performance.

The generated word is stored in R0. R10 points to the starting address of the compressed bits while R6 contains the value of the random number generated. R3, R4, and R5 point to the  $n - 24$ ,  $n - 55$  and  $n$ -th elements respectively in a cyclic list. Even though the size of the cyclic list required is 55 for the Mitchell and Moore generator, it can be implemented on 64 to optimize performance [Hwang 02a]. The

---

```

LDI R0, #0
LDI R1, #numXors      R1=numXors
LDR R6, [R3], #4      R6=mem[R3], R3+=4
LDR R7, [R4], #4      R7=mem[R4], R4+=4
ADD R6, R6, R7        R6+=R7
ANDI R6, R6, #mod     R6&=mod
STR R6, [R5], #4      mem[R5]=R6
AND R3, R3, R2
AND R4, R4, R2
AND R5, R5, R2
AND R8, R6, #31       R8=R6&31

Loop:
LDR R6, [R3], #4
LDR R7, [R4], #4
ADD R6, R6, R7
ANDI R6, R6, #mod
STR R6, [R5], #4
AND R3, R3, R2
AND R4, R4, R2
AND R5, R5, R2
LDR R9, [R10 + R6]    R9= mem[R10+R6]
ROR R9, R8
XOR R0, R0, R9        R0=R0^R9
SUBI R1, #1           R1=R1-1
BNEZ R1, Loop

```

Figure 4.4: Optimized code for the inner loop

list is assumed to start at an address that has 0s in the 8 least significant bits. The three AND instructions with R2 are used to keep the addresses from going over the boundary of the cyclic list.

### 4.3 Experimental Results

The proposed compression/decompression method was used to compress test vectors for the larger ISCAS'89 benchmark circuits. The results of these experiments are detailed below.

The test program was implemented in 'C' language on a Pentium II processor. The program was compiled and then the executable disassembled to get the size of the program that needs to be transferred to the embedded processor. Even though a compact version of the program can be implemented in assembly language, the high-level compiled implementation should be a good indicator of the relative sizes.

Table 4.1 compares the program size obtained with the program size for the methods in [Hellebrand 96], [Zacharia 96] and [Hwang 02b]. Programs implementing the algorithm given in [Hellebrand 96] and [Zacharia 96] were written in ‘C’ and the program sizes calculated similar to our method. For [Hellebrand 96], the size of the LFSR was assumed to be 64 bits since most of the circuits in that paper required a 64-bit LFSR. The length of the two-dimensional decompressor and the number of parallel LFSR’s in [Zacharia 96] were taken as 52 and 32 respectively. The code size for the proposed scheme is much smaller than that of the other methods.

Table 4.1: Comparison of code size

Code Size	[Hellebrand 96]	[Zacharia 96]	[Hwang 02b]	Proposed Scheme
# instr.	151	152	253	87
# bytes	564	608	701	344

A commercial ATPG tool was used to generate test cubes with 100% fault coverage for each circuit. The unspecified input assignments were left as X’s to enable better compression. We assumed a single scan chain for each circuit in our experiments. Table 4.2 shows the compression results obtained using the proposed scheme with the number of XORs equal to 3. The number of test vectors and the total number of bits in the test data are shown for each circuit. The column “*Spec. Bits*” shows the number of specified bits in the test set. The column “*Comp. Bits*” has the number of bits in the compressed set. The percentage data compression was computed as:

$$\text{Percentage Data Compression} = \frac{\text{Original Bits} - \text{Compressed Bits}}{\text{Original Bits}} \times 100$$

Note that some additional bits are need for the code as shown in Table 4.1. These are not included in the equation above since the decompression program can be reused for multiple cores. The percentage compression would be slightly lower if

the code size is included. The encoding efficiency is the ratio of the specified bits to the compressed bits. The set of equations for the compressed set were solvable for all the circuits when the number of compressed words was taken to be 18 more than the number of specified bits.

Table 4.2: Compression results for proposed method

Circuit	Scan Size	Test Vect.	Orig. Bits	Spec. Bits	Comp. Bits	% Comp.	Enc. Eff.
s5378	214	143	30602	5102	5696	81.4	0.896
s9234	247	146	36062	8677	9280	74.3	0.935
s13207	700	255	178500	9335	9920	94.4	0.941
s15850	611	148	90428	10567	11168	87.7	0.946
s38417	1664	105	174720	29847	30432	82.6	0.981
s38584	1464	131	191784	29610	30208	84.3	0.980

Table 4.3 shows a comparison of the compressed test data for the proposed method with other software-based compression methods for fully deterministic test sets - geometric primitives method [Maleh 01], incremental test vector construction [Jas 02], and matrix based decompression [Balakrishnan 02]. The size of the compressed test data for the proposed method is much smaller than the other methods.

Table 4.3: Comparison of compressed test data

Circuit	[Maleh 01]	[Jas 02]	[Balakrishnan 02]	Proposed Scheme
s5378	10057	12950	10390	5696
s9234	14666	18423	16888	9280
s13207	24446	24284	33470	9920
s15850	22458	24335	23552	11168
s38417	60478	97024	69556	30432
s38584	-	88376	66838	30208

Since both of the previous methods most related to this method, [Hellebrand

96] and [Zacharia 96], are mixed mode techniques, we also ran experiments on mixed mode test sets for easy comparison. Like [Hellebrand 96] and [Zacharia 96], we first applied 10K random patterns to detect easy faults. Deterministic patterns were then generated for the remaining faults. In Table 4.4, we compare the compression obtained in the mixed mode version of our scheme with the results published in [Hellebrand 96] and [Zacharia 96]. The test data that needs to be transferred from the tester to the embedded processor is composed of both the test program and the compressed data. Hence for any compression scheme to be implemented using an embedded processor, the test data storage requirement should take into account both the code and data sizes. The column *data* represents the size of the compressed data while the column *total* indicates the total test data storage requirements. The test data storage requirements for all the three methods were obtained by adding the code sizes from Table 4.1 to the data sizes. The proposed scheme requires less test data storage requirement than both the previous methods.

Table 4.4: Comparison of test data storage requirements

Circuit	[Hellebrand 96]		[Zacharia 96]		Proposed Scheme	
	Data	Total	Data	Total	Data	Total
s5378	759	5271		-	544	3296
s9234	11766	16278	4576	9440	4448	7200
s13207	10796	15308	7488	12352	2880	5632
s15850	11826	16338	7392	12256	5152	7904
s38417	71491	76003	16640	21504	23136	25888
s38584	11529	16041	3584	8448	2912	5664

## 4.4 Summary

In this chapter, we presented a method pseudo-random linear expansion based test vector compression/decompression scheme. It provides a number of advantages

compared with software-based LFSR reseeding including better compression, faster decompression (since it is word-based), and easy reusability (since no primitive polynomials are needed). The amount of compressed data required for the proposed method is very close to the total number of specified bits in the test set. The decompression program size for the proposed method is very small. The amount of compressed data processed at a time can be easily scaled to fit into whatever amount of on-chip memory is available by partitioning the test set appropriately.

## Chapter 5

# Diagnostic Response Compression for BIST

Fault diagnosis is done to determine the location and cause of failures and is very important during the initial manufacturing process. It helps in the identification of the manufacturing defects and is also used for yield learning to improve the production quality. Diagnostic information in scan-based BIST can be classified into two categories: space information which is the set of scan cells that capture the faulty responses, and time information which is the set of test vectors that fail. In this chapter, we propose a test response compression technique for scan-based BIST using an embedded processor that retains diagnostic information [Balakrishnan 03b]. The rest of this chapter is organized as follows. In the section 5.1, we discuss the previous work done in BIST diagnosis. Section 5.2 presents the proposed response compression technique; the corresponding diagnosis algorithm is discussed in section 5.3. Experimental results are presented in section 5.4 and section 5.5 is a summary.

### 5.1 Related Work

Extracting space information during scan-based BIST is much easier than identifying the failing test vectors since the length of the scan chain is typically much

smaller than the number of test vectors applied during BIST. There has been a lot of work in the past for identifying the scan cells that capture faulty information. Most involve intelligently collecting signatures over multiple BIST sessions and analyzing them later. In [Wu 96], a programmable MISR is used to collect multiple signatures with the MISR programmed with a different polynomial each time. The scan cells that had faulty responses are then identified by solving a set of non-linear equations. Other techniques involve partitioning the scan cells into different groups and observing each partition separately. [Rajski 99] uses an LFSR to pseudo-randomly mask out a set of scan cells. This process is repeated over multiple sessions and in each session, a different set of scan cells is masked. This idea was improved upon by [Bayraktaroglu 00a] using the principle of superposition. Properties of high quality partitions were identified in [Bayraktaroglu 00b] for this process and instead of randomly partitioning the scan cells, a deterministic partitioning technique was proposed. In this case, the improvement in accuracy involved additional hardware overhead. One drawback of all these methods is that the number of signatures collected determines the maximum number of scan chains with faulty responses that can be identified.

Only the cone of logic where the fault exists can be located by the information provided by identifying the scan cells that capture faults. Identifying which test vectors fail can provide much more information and hence much faster and more precise diagnosis. Most techniques proposed earlier for getting time information require additional hardware for diagnosis and are limited either by multiplicity of errors that can be handled or the hardware overhead. Methods using LFSRs were proposed by [McAnney 87], [Savir 88], and [Stroud 95]. A single LFSR is used in [McAnney 87] that guarantees correct diagnosis of single error sequences while two LFSRs are used in [Savir 88] and [Stroud 95] to diagnose single and double error sequences. Methods based on error correcting codes were proposed in [Karpovsky

93] and [Damarla 95]. An approach that does not require intermediate signatures was presented in [Aitken 89]. [McAnney 87], [Savir 88] and [Stroud 95] are limited by the multiplicity of errors that can be handled while [Aitken 89], [Karpovsky 93] and [Damarla 95] require very high hardware overhead. [Ghosh-Dastidar 99] presented a method that combines cyclic registers with pruning techniques to identify the failing test vectors and provide better diagnostic resolution than [Savir 88]. [Liu 02] presented a method for identifying failing vectors based on the use of overlapping intervals of test vectors. Signatures are collected for each interval of consecutive test vectors and the signatures are analyzed using pruning techniques to identify the set of candidate failing vectors. This method can require the collection of a large number of signatures.

In this chapter, we investigate the use of an embedded processor to aid in diagnosis for scan-based BIST. In contrast to all the previous schemes, the proposed method is software based and requires very little additional hardware overhead. The proposed method can be implemented with a small number of instructions. The proposed method provides both time and space information and requires that the BIST session be run only once. Note that most previous methods require that the BIST session be run a large number of times to collect many signatures.

## 5.2 Response Compression

In this section, the proposed scheme for response compression of BIST vectors is described. The response compression is based on pseudo-randomly compacting the output response. It can be considered as the dual of the pseudo-random linear expansion scheme that was used to decompress deterministic test vectors in [Balakrishnan 03]. It can be efficiently implemented in software through linear operations. Figure 5.1 shows the architecture of a typical SoC. It consists of multiple cores with different types of DFT techniques for each. It also consists of an embedded proces-

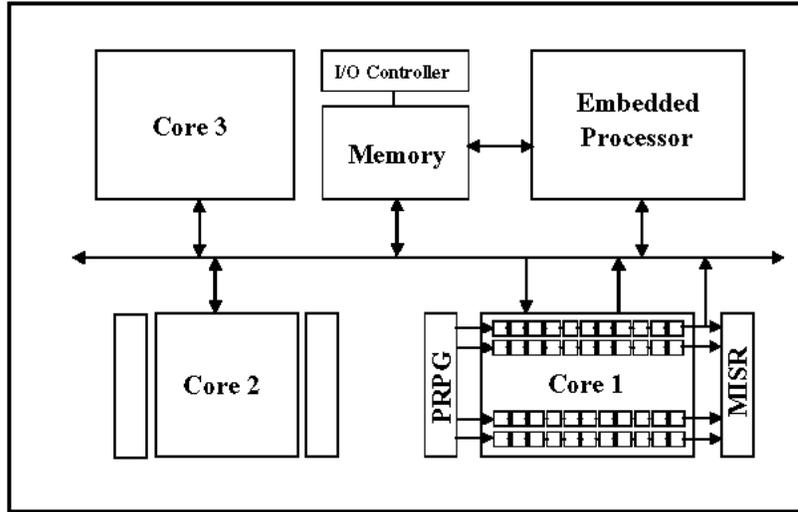


Figure 5.1: Architecture of a typical SoC

processor and embedded system memory connected to the different cores through a system bus. The embedded processor can be used to test the different cores in the SoC. On the input side, it can be used as an on-chip decompressor that decompresses the compressed deterministic test vectors coming from the tester. Otherwise, it can be used as a pseudo-random pattern generator for BIST. It can also be used for compacting the output response of the cores. For a BIST-ed core, the MISR performs very high compression with negligible aliasing. Since the MISR signature has very little diagnostic information, we propose to use the embedded processor for diagnosis in BIST.

Consider Core1 shown in Fig.5.1 as the circuit-under-test (CUT). During normal testing, the scan-BIST scheme is working. The pseudo-random pattern generator (PRPG) generates the patterns and sends it to the scan chains and the MISR compacts the output responses. If the MISR signature doesn't match with the fault-free signature, then an error is detected. At this point, the diagnosis process can be initiated. During diagnosis, the same test vectors are applied by the PRPG,

but this time the processor also reads the output responses. A software program running on the processor compacts the output responses and stores them on the system memory. The compacted output is then analyzed to get information about the failing vectors and the scan chains that capture faults. The software program stores a small number of uncompressed output responses at the beginning, and then compacts the remaining output responses. The reason for leaving some output responses uncompressed is to help in diagnosing faults that are detected by a large number of vectors. Such faults have a large number of errors in the output response and thus having just a small set of uncompressed output responses is sufficient for diagnosing them. However, for the faults that are detected by very few vectors, it is difficult to find the vectors for which they fail in the long BIST sequence, and hence it is for these faults that the proposed compression procedure is needed to aid the diagnosis process.

The response compression algorithm is based on pseudo-randomly XORing the output responses together to form a much smaller compacted set. The steps of the compression algorithm are:

1. Allocate a set of memory locations to store the compacted data and initialize them to zeros.
2. Read the output response word by word. Multiple scan chains (equal to word size of processor) can be read in parallel.
3. For each output response word, do the following multiple times
  - Randomly choose a location in allocated memory
  - Rotate the response word by a random number and
  - XOR the response word with the chosen memory location

The number of times each word is XORed together is represented by `numxors` and will determine the running time of the diagnosis program. The random number

generation is done using the Mitchell and Moore additive generator [Knuth 97] discussed in detail in Section 4.2.

### 5.3 Diagnosis

The proposed pseudo-random linear compression procedure can be represented by a binary matrix  $\mathbf{A}$  (each entry is 0 or 1), with one column for each bit in the output response and one row for each bit in the compacted response. The entry  $a_{ij}$  will be 1 if the output response bit corresponding to the column  $j$  is XORed into the compacted response bit corresponding to row  $i$ . For example, if the random number generator generates locations 3, 5 and 8 to XOR the first output response bit, then the entries  $a_{31}$ ,  $a_{51}$ , and  $a_{81}$  will be 1. All other entries in the first column will be 0. Since the number of non-zero entries in each column is equal to  $\text{numxors}$ , it is much smaller than the number of rows, thus the diagnosis matrix  $\mathbf{A}$  is a very sparse matrix. If the output response is represented by vector  $\mathbf{r}$ , the compacted response,  $\mathbf{c}_r$ , can be constructed as  $\mathbf{c}_r = \mathbf{A}\mathbf{r}$ .

During diagnosis, the faulty output response is compacted using the procedure discussed in Sec. 2 to get the compacted response vector  $\mathbf{c}_{faulty}$ . This is then compared with the compacted fault-free output response vector,  $\mathbf{c}_{ff}$ , to get the compacted difference or error vector,  $\mathbf{e} = \mathbf{c}_{faulty} \oplus \mathbf{c}_{ff}$ . The identification of the location of bits in error for diagnosis can then be represented in the form of a matrix equation  $\mathbf{A}\mathbf{x} = \mathbf{e}$ , where  $\mathbf{x}$  is the vector indicating the location of the bits of the output response that are in error. If an element of  $\mathbf{e}$  is 1 (the faulty and fault free compacted bits differ) then an odd number of non-zero columns corresponding to that row of  $\mathbf{A}$  are in error. Similarly, if an element of  $\mathbf{e}$  is 0, then either zero (no errors) or an even number of non-zero columns corresponding to that row of  $\mathbf{A}$  are in error.

This matrix equation is actually a set of simultaneous linear equations in the

$$\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & \mathbf{1} & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & \mathbf{1} & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & \mathbf{1} & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
x_8 \\
x_9 \\
x_{10} \\
x_{11} \\
x_{12}
\end{pmatrix}
=
\begin{pmatrix}
0 \\
\mathbf{1} \\
0 \\
0 \\
\mathbf{1} \\
\mathbf{1} \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{pmatrix}$$

Figure 5.2: Identification of incorrect bits

variables of  $\mathbf{x}$  that need to be solved to get the original output response. Since the number of equations is much smaller than the number of variables, the solution space for this set of equations will be enormous. The number of solutions depends on the compression ratio. The higher the compression ratio, the more the number of solutions. Since the solution space is enormous, a heuristic based approach is used to reduce it. The proposed heuristic is based on the fact that a lower number of errors is more probable than a higher number of errors. That is, if an erroneous compacted response bit (row) can be explained by errors in either 1 column or 3 columns, the 1 column error is a more probable solution and hence will be selected by our scheme. The reason why a lower number of errors is more probable than a higher number of errors is explained as follows. As was discussed in Sec. 5.2, a small set of output responses is initially left uncompressed. The faults that cause a large number of errors can be diagnosed using only these uncompressed responses. This leaves only the faults with a few number of errors for which diagnosis from the compacted responses is necessary. For these faults, the proposed heuristic is

applicable.

If the location of all non-zero bits in a column match the rows in error (in the vector  $e$ ), we add the column (and hence the corresponding bit in the output response) to the suspect set. Note that the suspect set contains all the bit locations in the output response that are suspected to be in error. Consider the example shown in Fig. 5.2 where 12 output response bits are compacted to 8 bits. Assume that the difference vector between the faulty and fault-free compacted responses,  $e$ , is as given in the figure. Since the position of 1's in the second column match with the position of 1's in the  $e$  vector (shown in bold in the figure), our method will add the output response bit corresponding to column 2 to the suspect set.

Note that if a row has an even number of errors, the errors will cancel out. So some columns that are in error will not match in the rows in error because the errors in one or more of the rows may have canceled out. Thus, some output response bits that are in error may not appear in the suspect set. Also, it is possible for a column that is not in error to match all the rows in error. This can happen if the column vector is equal to a linear combination of other column vectors that are in error. However, since the number of bits in error is very small compared to the number of bits not in error, the probability of this happening is low. Thus, the proposed heuristic is generally very effective at identifying a suspect set that accurately contains output response bits in error (which in turn identifies failing vectors and scan cells that capture errors). While not all output response bits that are in error will appear in the suspect set, having even a subset is very useful to guide the failure analysis process and greatly speed it up. Note also that the performance of the diagnosis procedure depends on the parameter *numxors* which is the number of XOR operations that are performed for each word of the output response. The parameter *numxors* determines the number of non-zero entries in each column. The higher the value of *numxors*, the greater the accuracy of the diagnosis since the

requirements for adding a column to the suspect set increase. However, the size of the suspect set goes down. Thus, the value of *numxors* can be used to trade-off a more inclusive suspect set versus a more accurate one. One approach to increase the effectiveness of the proposed diagnostic scheme would be to run the BIST session multiple times using different values of *numxors* and then compare the resulting suspect sets. Since the algorithm performs the check for each column of the matrix, the running time depends on the number of columns. In fact, the running time is  $O(\text{columns})$  since the checking is done a linear number of times for each column.

## 5.4 Experimental Results

We have performed several experiments to validate the proposed diagnostic scheme. The experiments can be classified into two categories: in the first set of experiments, we randomly inject errors into output response and simulate our diagnosis scheme, while the second set of experiments consists of injecting faults into the ISCAS'89 circuits and diagnosing the compacted output response.

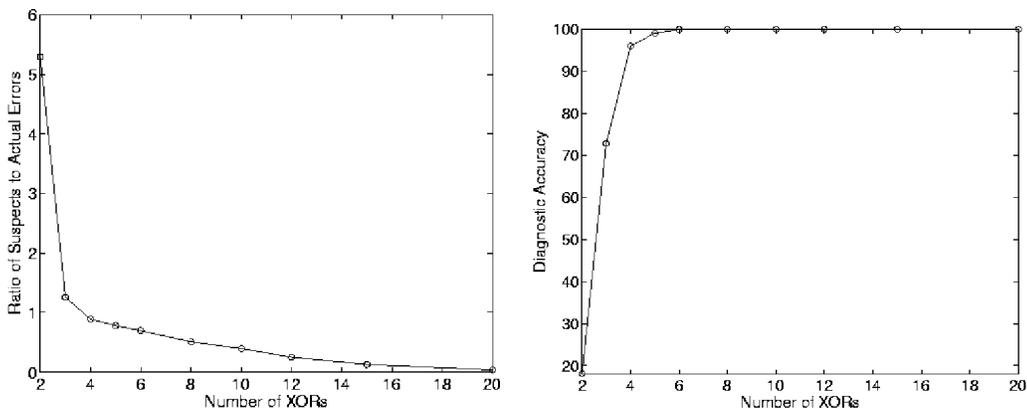


Figure 5.3: Variation of suspect size and diagnostic accuracy with *numxors*

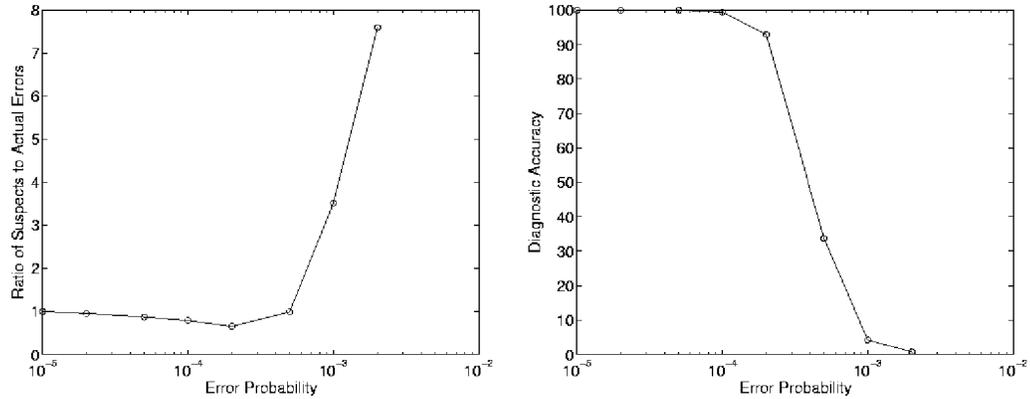


Figure 5.4: Variation of suspect size and diagnostic accuracy with error probability

The first set of experiments was done to study the impact of numxors, error probability (ratio of bits in error to total number of bits), and compression ratio (ratio of uncompressed response size to compacted response size) on the number of suspects, and diagnostic accuracy of our method. The results are shown in the following graphs. In Fig. 5.3, numxors is varied while the compression ratio was held constant at 100x and the error probability was held constant at 0.001. As numxors increases, the ratio of suspects to actual errors decreases while the diagnostic accuracy increases. The reason for this was explained in Sec. 3. In Fig. 5.4, error probability is varied while the compression ratio was held constant at 100x and numxors was held constant at 5. As the number of errors increases, the diagnostic accuracy initially remains relatively constant, but then drops off as the error probability exceeds 0.0002 in this case. The size of the suspect set decreases and reaches a minimum when the error probability is equal to 0.0002. By changing numxors, this inflection point can be moved. In Fig. 5.5, compression ratio is varied while numxors was held constant at 5 and the error probability was held constant at 0.001. As the compression ratio increases both the number of suspects

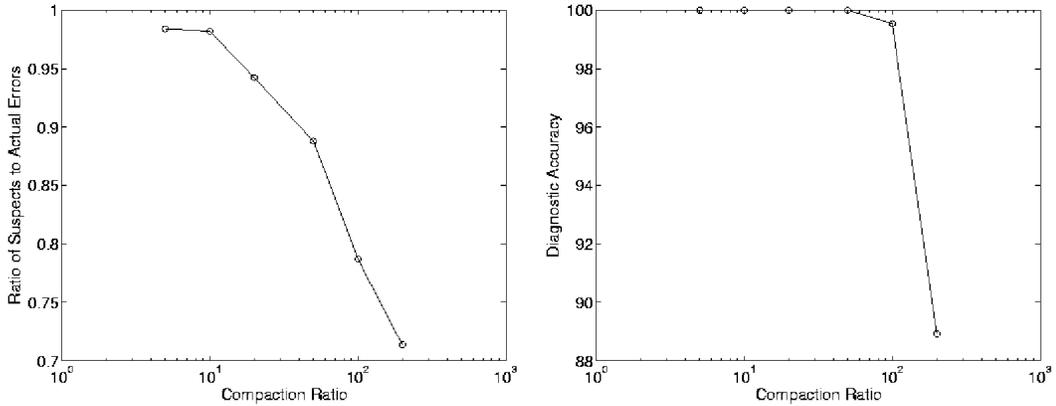


Figure 5.5: Variation of suspect size and diagnostic accuracy with compaction ratio

and diagnostic accuracy decrease.

A second set of experiments was done on the larger ISCAS'89 benchmark circuits. A single random stuck-at fault was injected into the CUT in each case and a BIST sequence of 10,000 patterns was simulated. The results are reported in Table 5.1. Even though we simulated our method for a large number of faults, we report results here for a small set of representative faults. We have incorporated faults that cause various numbers of errors. We implemented the cyclic registers method described in [Savir 88] and the pruning techniques for it described in [Ghosh-Dastidar 99]. Column 2 shows the number of failing test vectors out of the 10,000 random test vectors applied. Columns 3, 4 and 5 respectively give the number of suspect vectors  $|S|$ , the number of failing vectors correctly included in the suspect set and the number of non-failing vectors included in the suspect set for [Savir 88] for a particular fault. Columns 6-8 and columns 9-11 give the respective numbers for [Ghosh-Dastidar 99] and the proposed scheme for the same fault.

As can be seen in Table 5.1, the proposed scheme performs better than the other methods for all the circuits both in terms of the size of the suspect set and

the diagnostic accuracy. The number of non-failing vectors (last column of Table 5.1) in the suspect set for the proposed scheme is usually much smaller than the number of failing vectors. For all the faults (except the fourth row for circuit *s5378*), the number of non-failing vectors included in the suspect set is less than 8. This demonstrates the accuracy of our method for actual stuck-at faults. Note that in this comparison, the same amount of compacted response storage is used for all the techniques, so the improvement in the results is purely based on the algorithm and not simply due to using more storage.

## 5.5 Summary

In this chapter, a new response compression scheme for a scan-based BIST environment was presented which uses the power of an embedded processor to help in the diagnosis. The scheme requires that the BIST session be run only once and provides both space and time information for failure analysis. Experimental results validate the claim that the proposed scheme performs better than previously methods for identifying failing test vectors.

Table 5.1: Experimental results on ISCAS'89 benchmark circuits

Circuit	No. of Failing Vectors in 10,000	[Savir 88]			[Ghosh-Dastidar 99]			Proposed Scheme		
		No. of Suspect Vectors $ S $	Failing Vectors in $ S $	Non-Failing Vectors in $ S $	No. of Suspect Vectors $ S $	Failing Vectors in $ S $	Non-Failing Vectors in $ S $	No. of Suspect Vectors $ S $	Failing Vectors in $ S $	Non-Failing Vectors in $ S $
s5378	9	74	9	65	9	2	7	9	9	0
	50	1253	32	1221	0	0	0	46	46	0
	187	2104	44	1960	43	1	42	148	146	2
	404	2481	102	2379	46	3	43	435	245	190
s9234	2	4	2	2	2	2	0	2	2	0
	23	111	10	101	10	2	8	22	22	0
	74	45	7	38	7	1	6	67	67	0
	161	23	5	18	5	3	2	122	122	0
s13207	4	15	4	11	4	1	3	4	4	0
	44	1052	30	1022	30	0	30	44	44	0
	276	2217	80	2137	45	2	43	268	261	7
	453	581	43	538	23	1	22	367	364	3
s15850	5	22	5	17	5	0	5	5	5	0
	43	740	18	722	23	1	22	41	41	0
	103	1123	28	1095	33	1	32	101	101	0
	385	3116	119	2997	57	1	56	348	348	0

## Chapter 6

# Scan Inversion to Improve Compression

An important class of test vector compression schemes involves using a linear decompressor which uses only linear operations to decompress the test vectors. This chapter describes a new technique based on scan inversion that can significantly improve the encoding efficiency of any linear decompressor [Balakrishnan 04b].

If there are  $c$  scan cells, then the space of all possible scan vectors is  $2^c$ . The *output space* of a linear decompressor is the set of scan vectors that can be generated by the linear decompressor. Each bit stored on the tester can be thought of as a *free-variable* that can be assigned any value (0 or 1). Consider the case where the linear decompressor receives an input sequence from the tester consisting of  $n$  free-variables when generating a scan vector. Assuming the linear decompressor is always initialized to the same state before generating each scan vector (if it is a sequential circuit), then the size of the output space of the linear decompressor is less than or equal to  $2^n$  (since that is the number of possible unique input sequences that could be applied to it). The output space will be equal to  $2^n$  if every input sequence maps to a unique scan vector, and less than  $2^n$  if some input sequences map to the same scan vector. In the degenerate case where the linear decompressor is just a set of wires directly connecting each scan chain to a tester channel, then  $n = c$  and the content of every scan cell is equal to a unique free-variable such that the output space

of the linear decompressor contains all possible scan vectors. However, in order to get compression,  $n$  needs to be less than  $c$ , and thus in the general case, the output space of the linear decompressor will be a subset of all possible scan vectors. Linear decompressors have some nice properties compared with non-linear decompressors. They generally have a larger output space for the same  $n$  because of the use of XOR gates which minimize the number of input sequences that map to the same scan vector. Another very important property is that the output space of a linear decompressor is a linear subspace spanned by a Boolean matrix,  $\mathbf{A}_{c \times n}$ . The Boolean matrix  $\mathbf{A}$  can be obtained by symbolic simulation of the linear decompressor (see [Krishna 01] for details). Each row in  $\mathbf{A}$  corresponds to a scan cell and each column corresponds to a free-variable in the input sequence. The advantage of having the output space be defined by  $\mathbf{A}$  is that determining whether a particular test cube is contained in the output space and what the corresponding input sequence to generate it is can be quickly done by solving a set of linear equations.

In order to be able to compress a test set, the output space of the linear decompressor must contain all the test cubes in the test set. When using an LFSR as a linear decompressor, it has been shown that if the number of free-variables used to generate a test cube is 20 more than the number of specified bits in a test cube, then the probability of the test cube not being in the output space is less than  $10^{-6}$  [Könemann 91]. However, for a given test set, the number of free-variables can be reduced further provided the corresponding reduced output space still contains all the test cubes in the test set. Reducing the number of free-variables increases the encoding efficiency and hence compression. As the number of free-variables are reduced, the output space becomes smaller and smaller until a point is reached where one or more test cubes are no longer in the output space. This point terminates the reduction in free-variables and limits the encoding efficiency that can be achieved by the linear decompressor for a particular test set.

The proposed idea is to alter and reshape the output space of a linear decompressor using scan inversion to allow the number of free-variables to be reduced further while still keeping all the test cubes in the output space thereby increasing the encoding efficiency. The encoding efficiency can often be increased above 1 using the proposed approach. Any method for designing a linear decompressor can be used first to obtain the best linear decompressor. Using that linear decompressor as a starting point, the proposed method reduces the number of free-variables further to improve the encoding efficiency by incorporating scan inversion. The key property used by the proposed method is that scan inversion is a linear transformation of the output space and thus the output space remains a linear subspace spanned by a Boolean matrix. Using this property, a systematic procedure based on linear algebra is described for selecting the set of inverting scan cells to maximize encoding efficiency. A nice feature of the proposed method is that it can be implemented without any hardware overhead (this is described in detail in Sec. 6.2).

The rest of this chapter is organized as follows: Sec. 6.1 describes how this work relates to previous work. Sec. 6.2 discusses how scan inversion can be implemented in hardware. Section 6.3 presents a procedure for selecting the set of scan cells to invert based on linear algebra. Sec. 6.4 explains how the proposed method can be used to improve encoding efficiency for linear decompressors. Sec. 6.5 shows experimental results. Sec. 6.6 summarizes the chapter.

## 6.1 Related Work

A number of different techniques for designing linear decompressors have been proposed in the literature. The original idea of using an LFSR as a linear decompressor and solving for test cubes using linear algebra was described in [Könemann 91]. Several techniques for improving the encoding efficiency for linear decompressors based on LFSRs were described in [Venkataraman 93], [Hellebrand 95ab], [Zacharia

95, 96], [Rajski 98], and [Krishna 01, 02].

Linear decompressors that can receive data from the tester in a continuous-flow (i.e., "streaming" data) are especially useful for test data compression. Continuous-flow linear decompressors can be directly connected to the tester and operate very efficiently since they simply receive the data as fast as the tester can transfer it. From a tools integration standpoint, this is very nice since it mimics the standard behavior of normal scan chains. There is no need for any special scheduling or synchronization. A number of techniques for designing both combinational and sequential continuous-flow linear decompressors have been proposed. Combinational continuous-flow linear decompressors are described in [Hamzaoglu 99], [Hsu 01], [Bayraktaroglu 01, 03], [Mitra 03], and [Krishna 03]. Sequential continuous-flow linear decompressors are described in [Jas 00], [Könemann 01], [Rajski 02], [Volkerink 03], [Rao 03], and [Krishna 04]. Commercial tools for compressing test vectors including TestKompress from Mentor Graphics [Rajski 02], SmartBIST from IBM/Cadence [Könemann 01], and DBIST [Chandramouli 03] from Synopsys are based on linear decompressors.

The method described here can be used in conjunction with any linear decompressor including all of the ones referenced above to improve the encoding efficiency further. It transforms the output space of the linear decompressor in a way that allows a smaller number of free-variables from the tester to be used to encode the test set. The proposed idea has some relation to the idea of transforming the output space of a test pattern generator to encode test cubes which has been investigated in the past in the context of built-in self-test (BIST). Techniques for designing a non-linear circuit to transform the output space of a pseudo-random generator to encode test cubes were described in [Touba 95ab, 01], [Chatterjee 95], [Wunderlich 96], and [Kiefer 97, 98]. The proposed method differs significantly from these methods in that it uses linear transformations, is based on linear algebra, is targeted toward

lossless test vector compression, and can be implemented without any overhead.

## 6.2 Scan Inversion

The proposed method involves inverting a set of scan cells to transform the output space of the linear decompressor. Implementing inverted scan cells can be accomplished either by explicitly inserting inverters in the scan chains, or by simply using the  $Q'$  output instead of the  $Q$  output when connecting the output of one scan cell to the next scan cell. An example of inverting the contents of a scan cell is shown in Fig. 6.2. Figure 6.1 shows a normal scan chain without inversion, while Fig. 6.2 shows the scan chain with the third scan cell inverted. This is accomplished by inverting before and after the third scan cell. If the same scan vector was shifted into both the normal scan chain in Fig. 6.1 and the scan chain in Fig. 6.2, the contents of the third scan cell in Fig. 6.2 would have the opposite value from what it would be in the normal scan chain while the contents of all other scan cells would be the same. Also, when the output response is shifted out, the bit corresponding to the third scan cell will have the opposite value from what it normally would have.

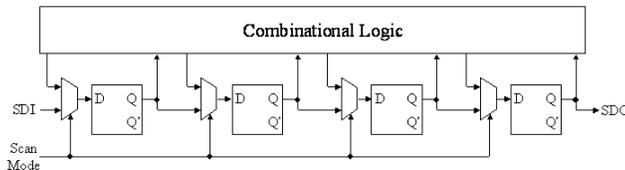


Figure 6.1: Normal scan chain

While the example in Fig. 6.2 only involves inverting one scan cell, any set of scan cells in the scan chain can be inverted by making appropriate connections from either  $Q$  or  $Q'$  to the next scan cell. To invert the first scan cell in a scan chain, either an inverter can be placed on the *scan\_data\_in* (*SDI*) input or the XOR

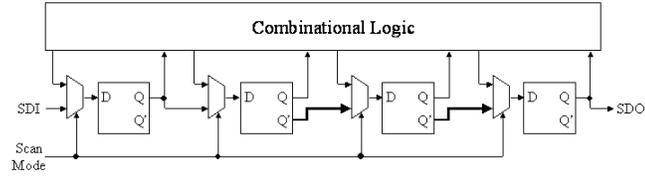


Figure 6.2: Scan chain with 3rd scan cell inverted

(XNOR) gate that is driving the SDI input can simply be changed to an XNOR (XOR) gate. If the inverted scan cells are implemented by simply changing some connections from  $Q$  to  $Q'$  and changing some XOR gates to XNOR gates, then there is effectively no overhead for using the proposed method. Even if explicit inverters are used, the overhead would still be small. The proposed method does not involve any special ordering of the scan cells. The scan chains can be ordered in any manner to optimize the layout.

### 6.3 Selecting Set of Inverted Scan Cells

The proposed method involves selecting a set of scan cells to be inverted. If there are  $c$  scan cells, then there are  $2^c$  different ways the scan cells can be inverted each of which results in a different transformation of the output space of the linear decompressor. Given a particular linear decompressor, the goal is to select the set of inverted scan chains so that the resulting transformed output space of the linear decompressor will contain all of the test cubes in the test set. This section describes a systematic procedure based on linear algebra for doing this.

The first step is to obtain the Boolean matrix  $\mathbf{A}$  that spans the output space of the linear decompressor. This can be obtained using symbolic simulation (see [Krishna 01] for details).

In order for a test cube to be in the output space of the linear decompressor,

there must exist a solution to the system of linear equations,  $\mathbf{Ax}=\mathbf{b}$  where  $\mathbf{x}$  is an assignment of values to the free-variables that are inputs to the decompressor when generating the test cube, and  $\mathbf{b}$  has the values of each bit in the test cube. Note that for the unspecified bits in the test cube, there is no need to solve the linear equations, so it is only the linear equations (rows) corresponding to the specified bits in  $\mathbf{b}$  that need to be considered. Gauss-Jordan Elimination [Cullen 97] can be used to perform rows operations that transform a set of columns into an identity matrix (these columns are called the *pivots*). An example of a system of linear equations for a test cube  $t_1$  is shown in Fig. 6.3, and the corresponding system after Gauss-Jordan Elimination is shown in Fig. 6.4. The vector  $\mathbf{b}$  after Gauss-Jordan Elimination will be referred to as  $\mathbf{z}$  to eliminate confusion. The rows after Gauss-Jordan Elimination can be classified as either *pivoted rows* or *linearly dependent rows*. The pivoted rows have pivots while the linearly dependent rows are all 0. In the example in Fig. 6.4, the first three rows are pivoted while the last two are linearly dependent. If all rows are pivoted, then a solution to the system of linear equations exists, and hence the test cube is in the output space of the linear decompressor. If some of the rows are linearly dependent, then a solution only exists if all of the corresponding values in  $\mathbf{z}$  (the vector  $\mathbf{b}$  after Gauss-Jordan Elimination) are equal to 0 for the linearly dependent rows. If there is a linearly dependent row whose corresponding value in  $\mathbf{z}$  is equal to 1, then a solution does not exist. In Fig. 6.4, the 4<sup>th</sup> row is linearly dependent, but the corresponding value in  $\mathbf{z}$  is 0 which is fine. However, the fifth row is also linearly dependent, but the corresponding value in  $\mathbf{z}$  is 1, and thus there is no solution. This is easy to see because in the original system of linear equations in Fig. 6.3, both rows 2 and 5 are identical in  $\mathbf{A}$ , however, the corresponding outputs for those two rows in  $\mathbf{b}$  have opposite values. Obviously there is no assignment to  $\mathbf{x}$  that will simultaneously solve the linear equations for both rows 2 and 5.

For the example in Fig. 6.3, let the specified values in  $\mathbf{b}$  correspond to scan

cells  $c_1$  through  $c_5$ . If either scan cell  $c_2$  or scan cell  $c_5$  were inverted, the system of linear equations would become solvable. For example, if scan cell 5 was inverted, then the last row in  $\mathbf{b}$  would be changed from 1 to 0. Now the fact that row 5 is the same as row 2 in  $\mathbf{A}$  is not a problem because the corresponding values in  $\mathbf{b}$  for those two rows are identical. This is an example of how scan inversion can be used to make a test cube solvable.

$$\begin{array}{c}
 \mathbf{A} \quad \mathbf{x} = \mathbf{b} \quad \mathbf{c}_1 \mathbf{c}_2 \mathbf{c}_3 \mathbf{c}_4 \mathbf{c}_5 \\
 \left( \begin{array}{cccccc}
 1 & 0 & 1 & 1 & 1 & 0 \\
 0 & 1 & 0 & 1 & 0 & 1 \\
 0 & 0 & 1 & 1 & 1 & 1 \\
 1 & 0 & 0 & 0 & 0 & 1 \\
 0 & 1 & 0 & 1 & 0 & 1
 \end{array} \right)
 \begin{array}{c}
 \mathbf{x}_1 \\
 \mathbf{x}_2 \\
 \mathbf{x}_3 \\
 \mathbf{x}_4 \\
 \mathbf{x}_5 \\
 \mathbf{x}_6
 \end{array}
 =
 \begin{array}{c}
 \left( \begin{array}{c}
 0 \\
 0 \\
 1 \\
 1 \\
 1
 \end{array} \right)
 \end{array}
 \left| \begin{array}{c}
 \left( \begin{array}{ccccc}
 1 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 1
 \end{array} \right)
 \end{array} \right.
 \end{array}$$

Figure 6.3: System of linear equations for test cube  $t_1$

$$\begin{array}{c}
 \text{Pivots} \quad \mathbf{z} \quad \text{Dependency} \\
 \left. \begin{array}{c}
 \text{Pivoted} \\
 \text{Rows}
 \end{array} \right\}
 \left( \begin{array}{c|c|c}
 \begin{array}{cccccc}
 1 & 0 & 0 & 0 & 0 & 1 \\
 0 & 1 & 0 & 1 & 0 & 1 \\
 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 &
 \begin{array}{c}
 1 \\
 0 \\
 1 \\
 0 \\
 1
 \end{array}
 &
 \begin{array}{ccccc}
 1 & 0 & 1 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 \\
 1 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 0 & 1
 \end{array}
 \end{array} \right)
 \begin{array}{c}
 \text{Constraints} \\
 i_1 \oplus i_3 \oplus i_4 = 0 \\
 i_2 \oplus i_5 = 1
 \end{array}
 \end{array}$$

Figure 6.4: Gauss-Jordan reduction for test cube  $t_1$

$$\begin{array}{c}
 \mathbf{A} \quad \mathbf{x} = \mathbf{b} \quad \mathbf{c}_1 \mathbf{c}_2 \mathbf{c}_6 \mathbf{c}_7 \\
 \left( \begin{array}{cccccc}
 1 & 0 & 1 & 1 & 1 & 0 \\
 0 & 1 & 0 & 1 & 0 & 1 \\
 1 & 1 & 1 & 0 & 1 & 1 \\
 0 & 1 & 0 & 0 & 0 & 1
 \end{array} \right)
 \begin{array}{c}
 \mathbf{x}_1 \\
 \mathbf{x}_2 \\
 \mathbf{x}_3 \\
 \mathbf{x}_4 \\
 \mathbf{x}_5 \\
 \mathbf{x}_6
 \end{array}
 =
 \begin{array}{c}
 \left( \begin{array}{c}
 0 \\
 0 \\
 1 \\
 1
 \end{array} \right)
 \left| \begin{array}{c}
 \left( \begin{array}{ccccc}
 1 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 1
 \end{array} \right)
 \end{array} \right.
 \end{array}$$

Figure 6.5: System of linear equations for test cube  $t_2$

$$\begin{array}{c}
\text{Linearly} \\
\text{Dependent}
\end{array}
\left\{ \begin{array}{c|c}
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{array}
&
\begin{array}{c}
\mathbf{z} \quad \text{Dependency} \\
\left. \begin{array}{c} 1 \\ 0 \\ 1 \\ 0 \end{array} \right\} \left. \begin{array}{c} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{array} \right\}
\end{array}
\right\} \text{Constraints } i_1 \oplus i_2 \oplus i_6 = 1$$

Figure 6.6: Gauss-Jordan reduction for test cube  $t_2$

$$\begin{array}{c}
\mathbf{C} \quad \mathbf{i} = \mathbf{z} \\
\text{Constraints for } t_1 \left\{ \begin{array}{c} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array} \right\} \\
\text{Constraints for } t_2 \left\{ \begin{array}{c} 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array} \right\}
\end{array}
\begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \\ i_6 \\ i_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 6.7: Constraint matrix for test set  $\{t_1, t_2\}$

Let  $\mathbf{i} = (i_1, i_2, \dots, i_c)$  be a vector in which if  $i_j=1$  then scan cell  $j$  is inverted and if  $i_j=0$  then scan cell  $j$  is not inverted. For a given test cube, a set of constraints on  $\mathbf{i}$  for which the test cube is solvable can be determined as follows. For the rows that are pivoted, there are no constraints on  $\mathbf{i}$  since it doesn't matter what the corresponding value of  $\mathbf{z}$  is for those rows as they are always solvable. For the linearly dependent rows, the corresponding value of  $\mathbf{z}$  must be 0, so whatever scan inversion takes place must ensure that the value for that row is 0 after Gauss-Jordan Elimination. This places constraints on  $\mathbf{i}$ . In the example in Fig. 6.4, the last two rows are linearly dependent, so constraints must be placed on  $\mathbf{i}$  to ensure that the corresponding values in  $\mathbf{z}$  for those rows are always 0. These constraints can be determined by augmenting the linear system with a dependency matrix which is a square matrix with the number of rows and columns equal to the number of rows in  $\mathbf{b}$ . The dependency matrix is initially an identity matrix since each value in  $\mathbf{b}$  depends only on itself. The row operations that are performed during Gauss-

Jordan Elimination are also applied to the dependency matrix. After Gauss-Jordan Elimination, the dependency matrix indicates which set of scan cells each value in  $\mathbf{z}$  depends on. For each linearly dependent row, the constraints on  $\mathbf{i}$  can be obtained by looking at the dependency for the value in  $\mathbf{z}$  for that row. The value in  $\mathbf{z}$  for each linearly dependent row must be 0, so a linear equation in terms of  $\mathbf{i}$  can be written to ensure that the value in  $\mathbf{z}$  will be 0. In the example in Fig. 6.4, the fourth row is linearly dependent and the corresponding value in  $\mathbf{z}$  depends on scan cells  $c_1$ ,  $c_3$ , and  $c_4$ . Since the corresponding value in  $\mathbf{z}$  in the normal scan chain is 0, the constraint on scan inversion is that  $i_1 \oplus i_3 \oplus i_4$  should be 0. In other words, either none of scan cells  $c_1$ ,  $c_3$ , and  $c_4$  should be inverted or two of them should be inverted (thus canceling each other out). If one or all three of them are inverted, then the value in  $\mathbf{z}$  for that row will become 1 and no solution for the test cube will exist. Similarly, a linear equation for the constraint due to the fifth row can be obtained. In this case, for the normal scan chain the value of  $\mathbf{z}$  for this row is 1, so it is necessary that one of the scan cells that it depends on be inverted, either  $c_2$  or  $c_5$  in order to get a solution for the test cube.

The procedure described above can be used to obtain a set of constraints on  $\mathbf{i}$  for each test cube to make it solvable. Consider test cube  $t_2$  whose system of linear equations is shown in Fig. 6.5. Note that this test cube has only 4 specified bits and in this case the specified bits are in scan cells  $c_1$ ,  $c_2$ ,  $c_6$ , and  $c_7$ . The system of linear equation after Gauss-Jordan Elimination is shown in Fig. 6.6. As can be seen, there is one linearly dependent row in this case. The corresponding set of constraints on  $\mathbf{i}$  can be obtained from the dependency matrix.

In order for the linear decompressor to be able to generate all of the test cubes in the test set, there needs to be a solution for all test cubes. Thus,  $\mathbf{i}$  must be selected to allow all test cubes to be simultaneously solved. A solution for  $\mathbf{i}$  can be obtained by forming a constraint matrix  $\mathbf{C}$  that incorporates all the constraints for

all of the test cubes. Each constraint for each test cube can be added as a row in  $\mathbf{C}$ . For example, suppose the test set consisted of test cube  $t_1$  and  $t_2$  whose constraints were obtained in Figs. 6.4 and 6.6. Then the first two rows in  $\mathbf{C}$  would correspond to the two linear constraint equations from Fig. 6.4, and the last row in  $\mathbf{C}$  would correspond to the linear constraint equation from Fig. 6.6. The resulting constraint matrix is shown in Fig. 6.7. Gauss-Jordan Elimination can then be used to find a solution to the system of linear equations  $\mathbf{C}\mathbf{i}=\mathbf{z}$ . The solution for  $\mathbf{i}$  gives the set of scan cells that need to be inverted so that the test set can be generated by the linear decompressor. If no solution exists, then it is not possible for the linear decompressor to generate the test set under any set of inverted scan cells.

The complexity of the Gauss-Jordan Elimination method for solving a set of  $n$  linear equations with  $m$  variables is of the order of  $O(nm^2)$ . In the proposed scheme, as with any other linear test vector compression scheme, a set of linear equations needs to be solved for each test cube. Hence, if  $s$  is the number of specified bits in a cube, and  $c$  is the number of compressed bits for that cube, then solving for that cube will require  $O(sc^2)$  time. The only additional task involved in the proposed scheme is to solve for the constraint matrix. The time complexity for this additional step will depend on how many equations there are in the constraint matrix and the number of scan cells that are included in the inversion constraints. Note that there is no need to include the scan cells that are not present in any inversion constraint for any test cube.

## 6.4 Using Scan Inversion to Increase Encoding Efficiency

Given a linear decompressor, the previous section described how to select a set of inverting scan cells so that all the test cubes in the test set will be in the output space (if such a set of inverting scan cells exists). This procedure can be used in two ways to increase encoding efficiency. One is to start with an initial decompressor

design and reduce the number of free-variables that it receives per test cube from the tester as much as possible while still keeping the test set in the output space through scan inversion. The other is to keep the number of free-variables that the decompressor receives per test cube constant, but use scan inversion to relax the constraints on ATPG (automatic test pattern generation) such that more specified bits per test cube can be generated by allowing more static and dynamic compaction while still being able to solve for the test cubes. Both of these applications will increase encoding efficiency and are discussed in detail in the following subsections.

#### 6.4.1 Reducing Free-Variables per Test Cube

Any method can be used to design the best linear decompressor for a normal scan chain. Then the number of free-variables that are input to the decompressor per test cube can be incrementally reduced and the procedure in Sec. 6.4 can be used to see if it is possible to still solve for all the test cubes using scan inversion. If so, then this process of incrementally reducing the number of free-variables and checking for a solution is repeated until a point is reached when no further reduction in the number of free-variables per test cube is possible while still being able to solve for all test cubes. For example, for a continuous-flow decompressor (either sequential or combinational), the number of free-variables per test cube can be reduced by simply holding one tester channel input to a constant 0 when doing symbolic simulation to obtain the matrix  $\mathbf{A}$  that spans the output space of the linear decompressor. This will reduce the rank of  $\mathbf{A}$  and hence reduce the output space. However, if the procedure in Sec. 6.4 can still solve for all test cubes using scan inversion, then the linear decompressor design can be simplified by eliminating that one tester channel input that was held to 0 since it is no longer needed. This can be repeated to try to eliminate additional tester channel inputs. The end result will be a linear decompressor that generates the exact same test set, but uses fewer tester channels

thereby reducing tester storage and bandwidth requirements.

### 6.4.2 Increasing Specified Bits per Test Cube

If the number of tester channels that are allocated for feeding the linear decompressor is fixed, then another way that scan inversion can be used to increase encoding efficiency is to allow more specified bits per test cube. Some test compression methodologies (e.g., [Bayraktaroglu 01, 03], [Rajski 02]) involve fixing the decompressor design and then constraining the ATPG so that the resulting test cubes will be in the output space of the decompressor. The constraints on the ATPG reduce the amount of static and dynamic compaction that are performed and therefore can result in more test cubes and hence more test time. The scan inversion method proposed here can be used to allow more specified bits per test cube while still being able to solve for the test cube. This can be used to relax the constraints on the ATPG and thereby allow more static and dynamic compaction. Each time a test cube is generated by the ATPG any additional constraints for solving the test cube via scan inversion can be added to the constraint matrix  $C$  described in Sec. 6.4. If the additional constraints cause the constraint matrix to no longer have a solution, then that test cube cannot be accepted and the ATPG must find a less specified test cube. If the constraint matrix still has a solution, then the test cube can be accepted. The end result of relaxing the constraints on the ATPG is that more static and dynamic compaction can be performed thereby reducing the total number of test cubes and hence reducing both test time and tester storage requirements.

## 6.5 Experimental Results

Two sets of experiments were performed to evaluate the effectiveness of the proposed method. The first set of experiments were done on randomly generated

test cubes for large industrial-size scan architectures. The second set of experiments were done on the largest ISCAS'89 benchmark circuits [Brglez 89]. For the randomly generated test cubes, experiments were performed using two different types of linear decompressors. The first was a combinational linear decompressor as shown in Fig. 6.8 using 512 scan chains and 1024 scan chains. The results are shown in Table 6.1. For the combinational decompressor with normal scan chains without inversion, the number of channels from the tester was 32. For each randomly generated test cube, the number of specified bits was incrementally increased until it could no longer be solved (i.e., it was no longer in the output space). The average percentage of specified bits per test cube that could be solved for is shown along with the corresponding encoding efficiency. This measures the best encoding efficiency that can be achieved for normal scan chains without scan inversion. Note that since each test cube is encoded independently, there is no dependence on the number of test cubes. Results are then shown for scan inversion using it in the two ways described in Sec. 6.5. The first is using scan inversion to reduce the number of tester channels while still encoding the same set of test cubes as before. The number of reduced tester channels is shown along with the resulting encoding efficiency that is achieved. The second way that scan inversion is used is to keep the tester channels at 32 and increase the percentage of specified bits per test cube as much as possible until it is no longer possible to solve for all the test cubes. The resulting encoding efficiency is shown for this case. Note that when using scan inversion, the effectiveness reduces as the number of test cubes increases since it is harder to keep all of the test cubes in the output space. Results are shown for several different number of test cubes.

A number of interesting observations can be made from Table 6.1. Scan inversion is remarkably effective at improving the encoding efficiency for combinational decompressors (it is more than doubled in most cases). Typically the encoding efficiency for combinational decompressors is fairly low because of the fact that they

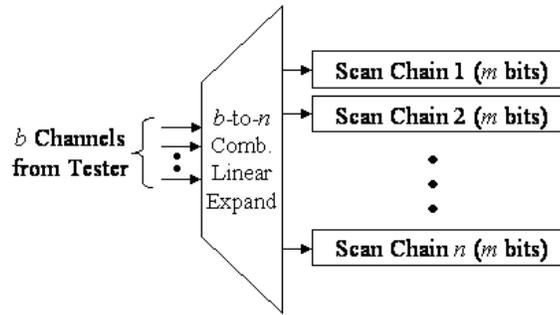


Figure 6.8: Combinational linear decompressor

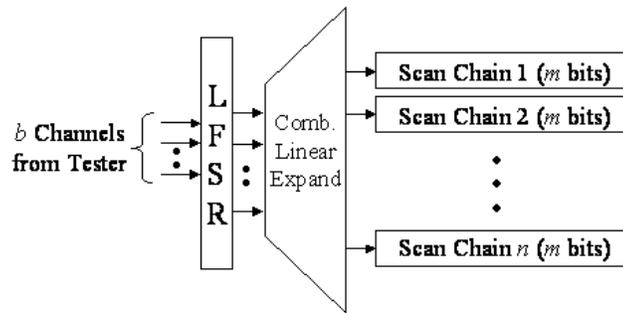


Figure 6.9: Continuous-flow sequential linear decompressor

must receive enough free-variables every clock cycle to be able to encode each bit-slice of the scan chains. The worst-case most heavily specified bit-slices typically limit the encoding efficiency. As can be seen in Table 6.1, the encoding efficiency is less than 0.5 without scan inversion. However, with scan inversion, the most heavily specified bit slices can be solved with fewer free-variables per clock cycle thereby allowing the number of tester channels to be reduced substantially. The fact that the encoding efficiency can be increased to around 0.9 with a combinational decompressor is a surprising result. This level of encoding efficiency is on the order of what is typically achieved with sequential decompressors, however, in this case no LFSR is needed thereby reducing the hardware overhead. Combinational decompressors are attractive because they are very simple requiring low hardware overhead. The

Table 6.1: Results for combinational linear decompressor

Scan Chains	Num. Test Cubes	Without Scan Inversion			With Scan Inversion			
		Tester Chan.	Perc. Spec.	Enc. Eff.	Red. Chan.	Enc. Eff.	Incr. % Spec.	Enc. Eff.
512	200	32	2.7%	0.43	11	1.26	6.2%	0.99
	400	32	2.7%	0.43	13	1.06	5.7%	0.91
	600	32	2.7%	0.43	15	0.92	5.5%	0.88
	1000	32	2.7%	0.43	16	0.86	5.2%	0.83
1024	200	32	1.3%	0.42	11	1.21	3.5%	1.12
	400	32	1.3%	0.42	11	1.21	3.1%	0.99
	600	32	1.3%	0.42	13	1.02	2.9%	0.93
	1000	32	1.3%	0.42	14	0.95	2.8%	0.90

Table 6.2: Results for sequential linear decompressor

Scan Chains	Num. Test Cubes	Without Scan Inversion			With Scan Inversion			
		Tester Chan.	Perc. Spec.	Enc. Eff.	Red. Chan.	Enc. Eff.	Incr. % Spec.	Enc. Eff.
512	200	16	3.0%	0.94	14	1.07	3.6%	1.13
	400	16	3.0%	0.94	15	1.00	3.3%	1.04
	600	16	3.0%	0.94	16	0.94	3.2%	1.01
	1000	16	3.0%	0.94	16	0.94	3.1%	0.98
1024	200	16	1.5%	0.94	11	1.37	2.1%	1.32
	400	16	1.5%	0.94	13	1.16	1.9%	1.20
	600	16	1.5%	0.94	14	1.07	1.8%	1.13
	1000	16	1.5%	0.94	15	1.00	1.7%	1.07

only drawback to using them has been the substantially reduced encoding efficiency compared with sequential decompressors, however, these results indicate that with scan inversion the gap in encoding efficiency between combinational and sequential decompressors can be significantly reduced. The results for keeping the number of channels at 32 and increasing the percentage of specified bits per test cube did not provide as high of encoding efficiency as reducing the tester channels did.

Same experiments were performed using a continuous-flow sequential linear decompressor as shown in Fig. 6.9. The results are shown in Table 6.2. In this case,

Table 6.3: Comparison of test data for different encoding schemes

Circuit Name	MinTest [Hamzaoglu 98]		Illinois Scan [Hamzaoglu 99]		FDR Codes [Chandra 01]		Mutation Enc. [Reda 02]	
	Num. Vect.	Total Bits	Num. Vect.	Total Bits	Num. Vect.	Total Bits	Num. Vect.	Total Bits
s13207	233	163,100	273	109,772	236	30,880	274	16,913
s15850	96	58,656	178	32,758	126	26,000	185	14,676
s38417	68	113,152	337	96,269	99	93,466	231	55,848
s38584	110	161,040	239	96,056	136	77,812	220	47,886

Circuit Name	Seed Overlapping [Rao 03]		Sequential Decompressor		Sequential with Scan Inversion	
	Num. Vect.	Total Bits	Num. Vect.	Total Bits	Num. Vect.	Total Bits
s13207	272	17,970	266	15,020	266	9,708
s15850	174	15,774	226	16,153	226	10,726
s38417	288	60,684	105	50,365	105	36,864
s38584	215	31,061	192	38,192	192	27,555

the number of channels from the tester for the normal scan chain without inversion was 16 and a 64 bit LFSR was used. Because the sequential linear decompressor is very efficient to begin with, the number of channels that can be reduced using scan inversion is not as spectacular as for combinational decompressors. Nonetheless, a significant increase in the encoding efficiency (above 1 in many cases) can be achieved especially for 1024 scan chains where the percentage of specified bits per test cube is less. Note also that the results for keeping the number of channels at 16 and increasing the percentage of specified bits per test cube were actually better than reducing the tester channels. This is the opposite of what happened for combinational decompressors where reducing channels achieved higher encoding efficiency.

Experiments were also performed for the largest ISCAS'89 benchmark circuits

to compare the results with previously published techniques. A scan architecture with 64 scan chains was assumed and a sequential decompressor like the one in Fig. 6.9 was used. Table 6.3 shows the results obtained by the proposed method along with other test vector compression schemes. The first column shows the circuit name and the next two columns are the number of test vectors and the total test size of the dynamically compacted test cubes generated by MINTEST [Hamzaoglu 98]. The next few columns show the number of test vectors and the compressed test size for the Illinois scan architecture [Hamzaoglu 99], frequency directed run length codes [Chandra 01], mutation encoding [Reda 02], and seed overlapping [Rao 03]. As can be seen from the results, scan inversion substantially reduces the tester storage requirements compared with using the sequential decompressor without scan inversion. The results indicate that the tester storage requirements are less than those in recently published test vector compression papers.

## 6.6 Summary

A method for improving the encoding efficiency of a linear decompressor using scan inversion was proposed. A systematic procedure based on linear algebra was described for selecting the set of inverted scan cells. Experimental results show that scan inversion can dramatically improve the encoding efficiency of combinational linear decompressors bringing it close to that of sequential decompressors. Scan inversion can also significantly improve the encoding efficiency of sequential linear decompressors. Scan inversion can be implemented with no hardware overhead.

## Chapter 7

# Entropy Limits to Compression

This chapter investigates the fundamental limits of test data compression, i.e., it looks to answer the question: *how much can we compress?* This is done by looking at the entropy of test sets and its relation to the compression limit [Balakrishnan 04a]. The idea of using entropy for calculating the test data compression limits was first studied in [Chandra 02]. However, in that work, the entropy calculations were limited to a special case related to FDR codes. All the don't cares were specified as 0's and only one set of symbols was considered. Different ways of specifying the don't cares and different symbol sets will lead to different entropies for the same test set. In this chapter, we investigate entropy including the additional degrees of freedom that were not explored in [Chandra 02]. A procedure for calculating the minimum entropy of a test set over all possible ways of specifying the don't cares is described. Also, the relationship between entropy and symbol partitioning is studied.

This chapter is organized as follows. In Sec. 7.1, the relationship of symbol partitioning and don't care filling to entropy is described. A greedy algorithm is described for specifying the don't cares in a way that minimizes entropy and it is proven to provide the optimum solution over all possible ways of filling the don't cares. In Sec. 7.2, the relationship between symbol length and maximum compression is discussed. In Sec. 7.3, different compression schemes that have been proposed in the literature are compared with their entropy limits. In Sec. 7.4, the

compression limits for LFSR reseeding are described. Section 7.5 is a summary.

## 7.1 Entropy Analysis for Test Data

Entropy is a measure of the disorder in a system. The entropy of a set of data is related to the amount of information that it contains which is directly related to the amount of compression that can be achieved. Entropy is equal to the minimum average number of bits needed to represent a codeword and hence presents a fundamental limit on the amount of data compression that can be achieved [Cover 91]. The entropy for a set of test data depends on how the test data is partitioned into symbols and how the don't cares are specified. These two degrees of freedom are investigated in this section.

### 7.1.1 Partitioning Test Data into Symbols

For block-to-block and block-to-variable codes, the test data is partitioned into fixed length symbols (i.e., each symbol has the same number of bits). The entropy for the test data will depend on the symbol length. Different symbol lengths will have different entropy. For a given symbol length, the entropy for the test data can be calculated. This will give a theoretical limit on the amount of compression that can be achieved by any block-to-block or block-to-variable code that uses that symbol length. This is illustrated by the following example. Consider the test set  $T$  in Table 7.1 which consists of four test vectors each partitioned into 4-bit blocks. The probability of occurrence of each unique 4-bit symbol is shown in Table 7.2. Note that the term “probability” here refers to the actual frequency of the symbol with respect to the total number of symbols in the data rather than the classical definition of probability as the chance of occurrence. The column “frequency” shows the number of times each symbol appears in the test set and the column “probability” is calculated by dividing the frequency with the total

number of blocks in the test set. The entropy of this test set is calculated from the probabilities of occurrence of unique symbols using the formula  $H = - \sum_{i=1}^n p_i \cdot \log_2 p_i$ , where  $p_i$  is the probability of occurrence of symbol  $x_i$  in the test set and  $n$  is the total number of unique symbols. This entropy is calculated and shown in the last row of Table 7.2. The entropy gives the minimum average number of bits required for each codeword. Thus, the maximum compression that can be achieved is given by  $(symbol\_length - entropy)/(symbol\_length)$  which in this case is equal to  $(4 - 2.64)/4 = 34\%$ . Now, if the test set is partitioned into 6-bit blocks instead of 4-bit blocks, then the corresponding probabilities of occurrence of unique symbols is shown in Table 7.3 and the entropy is shown in the last row. As can be seen, the entropy for 6-bit blocks is different than 4-bit blocks. The maximum compression in this case is equal to  $(6 - 3.25) / 6 = 46\%$ .

Table 7.1: Test set divided into 4-Bit blocks

test 1	1111	0001	0011	0000	1100	0111
test 2	0000	0011	0001	0111	0100	0000
test 3	0001	0111	0111	1100	1100	0000
test 4	0011	0000	0000	0100	1100	0111

For variable-to-block and variable-to-variable codes, the test data is partitioned into variable length symbols (i.e., the symbols can have different numbers of bits). One example is a run-length code where each symbol consists of a different size run-length. Given the set of variable length symbols, the entropy can be calculated the same way as for fixed length symbols. This is illustrated by partitioning the test set shown in Table 7.1 into symbols corresponding to different size runs of 0's. The probability of occurrence of each unique symbol is shown in Table 7.4 and the entropy is shown in the last row. Calculating the maximum compression for variable length symbols is different than for fixed length symbols. For variable length symbols, the

Table 7.2: Probability table for symbol length of 4

i	Symbol $x_i$	Freq.	Probability $p_i$
1	0000	6	0.2500
2	0111	5	0.2083
3	1100	4	0.1667
4	0011	3	0.1250
5	0001	3	0.1250
6	0100	2	0.0833
7	1111	1	0.0416
Entropy = 2.64			
Max. Compression = 34.0%			

maximum compression is equal to  $(avg\_symbol\_length - entropy) / (avg\_symbol\_length)$ . The average symbol length is computed as  $\sum_{i=1}^n p_i \cdot |x_i|$  where  $p_i$  is the probability of occurrence of symbol  $x_i$ ,  $|x_i|$  is the length of symbol  $x_i$ , and  $n$  is the total number of unique symbols. For this example, the average symbol length is 2.49, so the maximum compression is  $(2.53 - 2.07) / 2.53 = 18\%$ . This is the maximum compression that any code that encodes runs of 0's can achieve for the test data in Table 7.1.

### 7.1.2 Specifying the Don't Cares

While computing entropy for fully specified data is well understood, the fact that test data generally contains don't cares makes the problem of determining theoretical limits on test data compression more complex. Obviously, the entropy will depend on how the don't cares are filled with 1's and 0's since that affects the frequency distribution of the various symbols. To determine the maximum amount of compression that can be achieved, the don't cares should be filled in a way that minimizes the entropy.

While the number of different ways the don't cares can be filled is obviously exponential, for fixed length symbols, a greedy algorithm is presented here for filling

Table 7.3: Probability table for symbol length of 6

i	Symbol $x_i$	Freq.	Probability $p_i$
1	000000	4	0.2500
2	010011	2	0.1250
3	000111	2	0.1250
4	111100	1	0.0625
5	000011	1	0.0625
6	110001	1	0.0625
7	011101	1	0.0625
8	000101	1	0.0625
9	110111	1	0.0625
10	110011	1	0.0625
11	001100	1	0.0625
Entropy = 3.25			
Max. Compression = 45.8%			

the don't cares in an optimum way to minimize entropy. A proof is given that it minimizes entropy across all possible ways of filling the don't cares. However, for variable length symbols, the problem is more difficult and remains an open problem. In [Wurtenberger 03], this problem is discussed for run length codes, and a simulated annealing solution is adopted.

### Greedy fill algorithm

For fixed length symbols, we describe an algorithm called *greedy fill* below and prove that it is optimal, i.e., it results in minimum entropy among all possible ways of filling the don't cares given a specific partitioning of the test set into fixed length symbols.

The greedy fill algorithm is based on the idea that the don't cares should be specified in a way such that the frequency distribution of the patterns become as skewed as possible. This algorithm was first described in [Jas 03] where the goal was

Table 7.4: Probability table for runs of 0's

i	Symbol $x_i$	Freq.	Probability $P_i$
1	1	22	0.5789
2	01	4	0.1053
3	0001	4	0.1053
4	001	3	0.0789
5	000000001	2	0.0526
6	00001	1	0.0263
7	0000001	1	0.0263
8	000000001	1	0.0263
Entropy = 2.07			
Max. Compression = 18.2%			

to maximize the compression for Huffman coding, but it also applies for minimizing entropy. If the fixed symbol length is  $n$ , then the algorithm identifies the *minterm*, a fully specified symbol of length  $n$  (there are  $2^n$  such minterms), that is contained in as many of  $n$ -bit blocks of the unspecified test data as possible, i.e., having the highest frequency of occurrence. This minterm is then selected, and all the unspecified  $n$ -bit blocks that contain this minterm are specified to match the minterm. The algorithm then proceeds in the same manner by finding the next most frequently occurring minterm. This continues until all the don't cares have been specified.

To illustrate this algorithm, an example test set consisting of four test vectors with 12 bits each is shown in Fig. 7.1(a). The test vectors are partitioned into 4-bit blocks resulting in a total of twelve 4-bit blocks. Of the 16 ( $2^4 = 16$ ) possible minterms for a block, the most frequently occurring minterm is 1111 as seven of the unspecified blocks contain 1111. After the first iteration, only five blocks remain and the most frequently occurring minterm now is 1000. All the five blocks contain this minterm and hence the algorithm terminates. The resulting test set after filling up the unspecified bits in the above manner is shown in Fig. 7.1(b).

x0x0	10xx	1xxx	1000	1000	1111
xx1x	x1x1	111x	1111	1111	1111
xx00	1xx0	x111	1000	1000	1111
xxxx	x1xx	1000	1111	1111	1000

(a) Unspecified Test Set

(b) After Greedy Fill Algorithm

Figure 7.1: Example of specifying don't cares to minimize entropy

**Theorem 1:** The probability distribution obtained from the *greedy fill* algorithm described above results in minimum entropy.

**Proof:** Let there be  $m$  different (unique) symbols in the test set. Let  $\mathbf{p} = (p_1, p_2, p_3, \dots, p_m)$  be the probability distribution obtained using the greedy fill algorithm. Note that here  $p_i$  is actually the frequency of occurrence of the  $i$ -th term divided by the total number of blocks. By construction of the algorithm,

$$p_1 > p_2 > p_3 > \dots > p_m \tag{1}$$

The entropy of this distribution is given by

$$H(\mathbf{p}) = H(p_1, p_2, \dots, p_n) = \sum_{i=1}^m -p_i \log p_i \tag{2}$$

Let  $\mathbf{q} = (q_1, q_2, q_3, \dots, q_m)$  be any other feasible frequency distribution. Then we need to prove that  $H(\mathbf{q}) > H(\mathbf{p})$  for all such feasible  $\mathbf{q}$ . Before we proceed further, we introduce the concept of interchanges between terms and note how it affects the entropy. This is required since the unspecified bits in the test set make a single block compatible with many different symbols. Let us consider a block  $x_0$  with frequency  $p_0$  that is compatible with two different symbols  $x_1$  (frequency  $p_1$ ) and  $x_2$  (frequency  $p_2$ ). The following lemma summarizes the entropy results when  $x_0$  is moved between  $x_1$  and  $x_2$ .

*Lemma 1:* Transferring  $x_0$  from  $x_1$  to  $x_2$  will reduce the entropy if  $p_1 - p_0 < p_2$  and increase the entropy if  $p_1 - p_0 > p_2$ .

*Proof:* Let  $H = H(p_1, p_2, p_3, \dots, p_m)$  and  $H' = H(p_1 - p_0, p_2 + p_0, p_3, \dots, p_m)$ .

Consider

$$\begin{aligned} H' - H &= -(p_1 - p_0) \log(p_1 - p_0) - (p_2 + p_0) \log(p_2 + p_0) + p_1 \log p_1 + p_2 \log p_2 \\ &= \{p_1 \log p_1 - (p_1 - p_0) \log(p_1 - p_0)\} + \{p_2 \log p_2 - (p_2 + p_0) \log(p_2 + p_0)\} \end{aligned}$$

Let  $g(x) = x \log x$ , and dividing the above equation by  $p_0 > 0$ , we get

$$\frac{H' - H}{p_0} = \frac{g(p_1) - g(p_1 - p_0)}{p_0} - \frac{g(p_2 + p_0) - g(p_2)}{p_0} \quad (3)$$

Consider the second derivative of  $g(x)$ ,

$$\frac{d^2}{dx^2} g(x) = \frac{d}{dx} \left( \log_2 x + \frac{1}{\ln 2} \right) = \frac{1}{\ln 2} \cdot \frac{1}{x} \quad (4)$$

From (4) it is clear that the second derivative is strictly positive if  $x > 0$ , and hence the first derivative of  $g(x)$  is a monotonically increasing function when  $x > 0$ .

This means

$$\frac{g(x_1 + \Delta x) - g(x_1)}{\Delta x} < \frac{g(x_2 + \Delta x) - g(x_2)}{\Delta x} \text{ if } x_1 < x_2 \text{ and } \Delta x > 0$$

Using this in (3), by substituting  $x_1 = p_1 - p_0$  and  $x_2 = p_2$ , we get:

$$H' - H > 0 \quad \text{if} \quad p_1 - p_0 > p_2 \quad (5)$$

$$H' - H < 0 \quad \text{if} \quad p_1 - p_0 < p_2 \quad (6)$$

The proof for the main theorem is given below. We first prove that we can reach the greedy fill frequency distribution  $\mathbf{p}$  from any other feasible frequency distribution  $\mathbf{q}$  through a series of transfers, each of which will result in a decrease in the entropy. Consider  $q_1$ , the frequency corresponding to the first term. Note that either  $q_1 < p_1$  or  $q_1 = p_1$  ( $p_1$  is the maximum frequency of term  $x_1$  by construction). If  $q_1 = p_1$ , the following will apply to  $q_2$  or the next  $q_i$  which is less than the corresponding  $p_i$ . Given  $q_1 < p_1$ , we will transfer blocks from other terms to  $x_1$  so that  $q_1$  becomes equal to  $p_1$ . Consider other terms  $x_i$  ( $i \neq 1$ ) which have blocks that can be transferred to  $x_1$ . For all such terms whose frequency  $q_i$  is less than  $q_1$ ,

the transfer will result in decrease of entropy by equation (6) of Lemma 1. After all such transfers are done, let  $\mathbf{q}$  be again the frequency distribution for ease of notation. Now consider all other terms that can transfer blocks to  $x_1$  and whose frequencies  $q_i$  is greater than  $q_1$ . Let there be  $n$  such terms  $x_{i1}, x_{i2}, \dots, x_{in}$  with frequencies  $q_{i1}, q_{i2}, \dots, q_{in}$ . It can be proved by induction that these transfers will also result in a reduction of entropy. By proceeding similarly for each  $q_i$  that is less than  $p_i$ , we can reach the greedy fill frequency distribution from any feasible frequency distribution with the entropy reducing at every stage. Hence the greedy fill frequency distribution has the minimum entropy.  $\square$

### Experimental results for greedy fill algorithm

Experiments were performed using the greedy fill algorithm to calculate the theoretical limit on test data compression for the dynamically compacted test cubes generated by MINTEST [Hamzaoglu 98] for the largest ISCAS'89 [Brglez 89] benchmark circuits. These are the same test sets used for experiments in [Chandra 01ab], [Gonciari 03], and [Jas 03]. The compression values in Table 7.5 are calculated from the exact values of minimum entropy that were generated using the *greedy fill* algorithm. As can be seen from the table, the percentage compression that can be achieved increases with the symbol length. No fixed-to-fixed or fixed-to-variable code using these particular symbol lengths can achieve greater compression than the bounds shown in Table 7.5 for these particular test sets. Note, however, that these entropy bounds would be different for a different test set for these circuits.

The greedy fill algorithm is exponential in the symbol length because the number of minterms grows exponentially. Thus, it is not practical to calculate the entropy for larger symbol lengths using this algorithm. However, the next subsection presents an approximate algorithm that can scale to larger symbol lengths.

Table 7.5: Exact entropy compression limits using the greedy fill algorithm

Circuit	Original Test Data	Symbol Length			
		4	6	8	10
s5378	23754	52.0	56.3	59.2	63.8
s9234	39723	54.1	57.4	60.7	63.7
s13207	165200	83.6	85.0	85.6	87.1
s15850	76986	68.9	70.5	71.9	73.5
s38417	164736	59.8	63.2	65.3	67.7
s38584	199104	65.9	67.6	68.8	70.8

### Approximate fill algorithm

Since the *greedy fill* algorithm is exponential in the symbol length, an alternative algorithm is needed to calculate entropy for larger symbol lengths. An approximate fill algorithm is proposed for this. It does not guarantee to find the minimum entropy, but experimental data will be shown indicating that it comes very close. In the approximate algorithm, the 3-valued unspecified blocks are ordered from highest frequency of occurrence to lowest. One unspecified block is used as the starting point, and an attempt is made to merge it (similar to static compaction) with each of the other unspecified block going in order from high frequency to lowest. Two blocks can be merged if they do not conflict in any bit position (i.e., have opposite specified values in a bit position). After merging with as many blocks as possible, the resulting merged symbol will have a frequency of occurrence equal to the sum of the frequency of occurrence for each of the blocks it was merged with. Each block is used in turn as a starting point for this merging process, and the merged symbol that results in the highest frequency of occurrence is selected (if it has any don't cares after merging, they can be arbitrarily filled with no impact on entropy). The don't cares in each unspecified block that is compatible with the selected merged symbol are specified to match it. This process is then repeated until all the don't

cares in the test data are specified. This approximate algorithm is not exponential and thus can be used for large symbol lengths.

### Experimental results for approximate fill algorithm

Experiments were performed using the approximate fill algorithm to calculate the theoretical limit on test data compression for the dynamically compacted test cubes generated by MINTEST [Hamzaoglu 98] for the largest ISCAS'89 [Brglez 89] benchmark circuits. The results are shown in Table 7.6. In comparing the results in Table 7.5 and Table 7.6, it can be seen that the results for the approximate fill algorithm are very close to that of the exact greedy fill algorithm.

Table 7.6: Entropy compression limits using approximate fill algorithm

Circuit	Symbol Length									
	4	6	8	10	12	16	20	24	28	32
s5378	52.0	56.2	59.2	63.7	66.2	71.6	76.6	85.2	91.1	96.0
s9234	53.6	56.9	60.0	63.4	65.0	68.7	71.8	82.4	88.5	91.8
s13207	83.6	85.0	85.6	87.0	88.0	88.8	89.2	93.2	96.6	98.3
s15850	68.9	70.5	71.9	73.3	75.1	77.8	80.1	85.1	89.1	94.0
s38417	60.1	63.2	65.3	67.7	68.5	71.7	76.6	83.0	87.7	89.2
s38584	65.9	67.6	68.8	70.8	72.2	75.1	77.4	84.7	88.1	93.7

## 7.2 Symbol Length Versus Compression Limits

In this section, the relationship between symbol length, compression limits, and decoding complexity is investigated. Figure 7.2 shows a graph of the compression limits calculated from minimum entropy for the benchmark circuit *s9234* as the symbol length is varied. The percentage compression varies from 50% for a symbol length of 2 to 92 % for a symbol length of 32; this corresponds to a compression ratio range of 2 to 12. The reason why greater compression can be achieved for

larger symbol lengths is that more information is being stored in the decoder. This becomes very clear when the compression limit is graphed for all possible symbol lengths as shown in Fig. 7.3. As can be seen, the compression ratio goes toward infinity as the symbol length becomes equal to the number of bits in the entire test set. When the symbol length is equal to the number of bits in the entire test set, then the decoder is encoding the entire test set. This is equivalent to built-in self-test (BIST) where no data is stored on the tester.

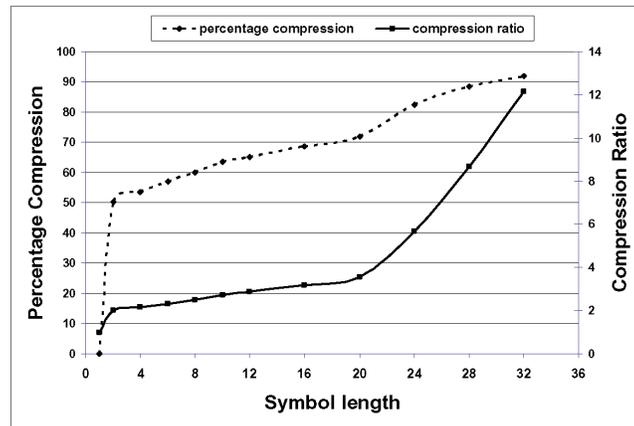


Figure 7.2: Compression limits for different symbol lengths

Consider the simple case where the information in the decoder is simply stored in a ROM without any encoding. The decoder ROM translates the codewords into the original symbols, so at a minimum, it needs to store the set of original symbols. For a symbol length of 2, there are  $2^2$  possible symbols, so the decoder ROM would have to store 4 symbols each requiring 2 bits, thus it requires 8 bits of storage. For a symbol length of 4, there are  $2^4$  possible symbols, so the decoder ROM would have to store 16 symbols each requiring 4 bits, thus it requires 64 bits of storage. Having a larger decoder ROM allows greater compression. For a symbol length of 32, there

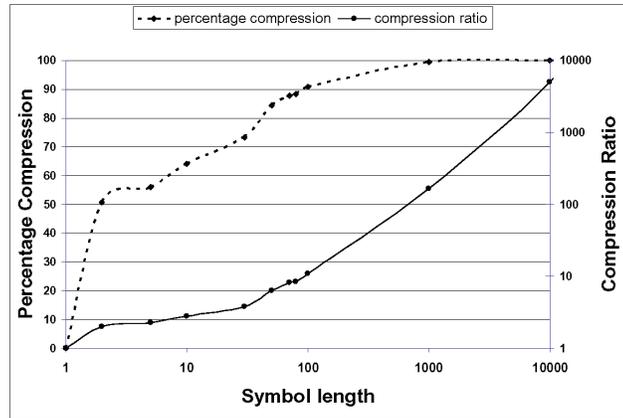


Figure 7.3: Compression limits for different symbol lengths

are  $2^{32}$  possible symbols, but it is not likely that the test data would contain all of them. The decoder would only need to store the symbols that actually occur in the test data. In Fig. 7.4, the size of a simple decoder ROM that is required for different symbols lengths is graphed for different symbol lengths. As can be seen, the decoder information goes up exponentially as the symbol length goes from 1 to 10 as in this range all possible symbols are occurring in the test data and the number of symbols is growing exponentially. After this, the decoder information goes up less than exponentially because not all possible symbols of that length occur in the test data. After the symbol length exceeds about 20, the decoder information is nearly equal to the entire test set as there is very little repetition of the symbols in the test data (i.e., almost all the symbols in the test data are unique at that point). One way to view the graph in Fig. 7.4 is that as the symbol length is increased, essentially more information is being stored in the on-chip decoder, and less information is being stored off-chip in the tester memory. A symbol length of 1 corresponds to conventional external testing with no compression, and a symbol length equal to

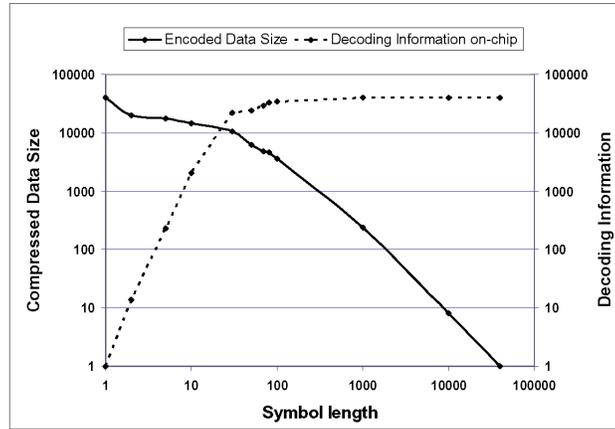


Figure 7.4: Compression limits for different symbol lengths

the size of the test set corresponds to conventional BIST where no data is stored on the tester.

The real challenge in test data compression is in the design of the decoder. The entropy of the test data places a fundamental limit on the amount of compression that can be achieved for a particular symbol length, but the real key is to achieve something close to the maximum compression using a small decoder. The next two sections evaluate existing test data compression schemes with respect to their compression limits based on entropy theory.

### 7.3 Analysis of Test Data Compression Schemes

In this section, different test data compression schemes proposed in the literature are compared with their entropy bounds. Both schemes that operate on fixed length symbols as well as those that work on variable length symbols are considered. One major constraint is that we are only able to report results for schemes for which

we have access to the exact test set that was encoded. For this reason, we have limited the results to only those schemes that reported results for the MINTEST test cubes [Hamzaoglu 98]. Discussion of LFSR reseeding schemes is deferred to Sec. 7.5.

### 7.3.1 Fixed symbol length schemes

Table 7.7 shows results for two compression schemes. One is a block-to-variable scheme based on Selective Huffman coding [Jas 99, 03], and the other is a block-to-block scheme that uses a selective dictionary with fixed length indices [Li 03]. The first two columns show the circuit name and the size of the original test set. Then results are shown for [Jas 03] comparing the actual compression achieved with the entropy limit for a symbol length of 8 and 12. The same is done for the method in [Li 03] for a block size of 16 and 32. The method in [Jas 03] is closer to the entropy limit because it uses variable length codewords whereas the method in [Li 03] uses fixed length codewords. However, the method in [Li 03] has the advantage of an easier interface to the tester since it is a block-to-block code.

Table 7.7: Comparison of fixed symbol length schemes with entropy limit

Circuit	Test Size	Selective Huffman [Jas 03]				Dictionary Based Compression [Li 03]			
		8		12		16		32	
		Entropy Limit	Actual Comp.	Entropy Limit	Actual Comp.	Entropy Limit	Actual Comp.	Entropy Limit	Actual Comp.
s5378	23754	59.2	50.1	66.2	55.1	71.6	45.3	96.0	67.2
s9234	39723	60.0	50.3	65.0	54.2	68.7	46.1	91.8	65.9
s13207	165200	85.6	69.0	88.0	77.0	88.8	49.2	98.3	73.4
s15850	76986	71.9	60.0	75.1	66.0	77.8	47.4	94.0	67.6
s38417	164736	65.3	55.1	68.5	59.0	71.7	44.0	89.2	49.0
s38584	199104	68.8	58.3	72.2	64.1	75.1	45.8	93.7	64.3

### 7.3.2 Variable symbol length schemes

Table 7.8 shows results for three compression schemes that are based on variable-to-variable codes. Each of these schemes are based on encoding runs of 0's. As was discussed in Sec. 7.2, filling the don't cares to minimize entropy for variable size symbols is an open problem. These three schemes all fill the don't cares by simply replacing them with specified 0's. For simplicity, the entropy limits in Table 7.8 were calculated by also filling the don't cares with specified 0's.

The results in Table 7.8 indicate that the VIHC method in [Gonciari 03] gets very close to the entropy limit. This is because this method is based on Huffman coding. One conclusion that can be drawn from these results is that there is not much room for improvement for research in variable-to-variable codes that encode runs of 0's. The only way to get better compression than the entropy limit that is shown here would be to use a different automatic test pattern generation (ATPG) procedure or fill the don't cares in a different way that changes the entropy limit.

Table 7.8: Comparison of variable symbol length schemes with entropy limit

Circuit	Test Size	Entropy Limit	Golomb [Chandra 01a]	FDR [Chandra 01b]	VIHC [Gonciari 03]
s5378	23754	52.4	40.7	48.0	51.8
s9234	39723	47.8	43.3	43.6	47.2
s13207	165200	83.7	74.8	81.3	83.5
s15850	76986	68.2	47.1	66.2	67.9
s38417	164736	54.5	44.1	43.3	53.4
s38584	199104	62.5	47.7	60.9	62.3

## 7.4 Analysis of LFSR Reseeding Schemes

Conventional LFSR reseeding [Könemann 91] is a special type of block-to-block code in which the set of symbols is the output space of the LFSR and the set of

codewords is the seeds. As was seen in the graphs in Sec. 7.3, larger symbol lengths are needed in order to achieve greater amounts of compression due to the entropy limits on compression. The difficulty with larger symbol lengths is that the decoder needs to be able to produce more symbols of greater length. The power of LFSR reseeding is that an LFSR is used for producing the symbols. An  $r$ -stage LFSR has a maximal output space as it produces  $2^r - 1$  different symbols as well as having a very compact structure resulting in low area. Thus, an LFSR is an excellent vehicle for facilitating larger symbol lengths with a small area decoder. The only issue is whether the set of symbols that occur in the test data,  $S_{data}$ , are a subset of the symbols produced by the LFSR,  $S_{LFSR}$ , (i.e.,  $S_{data} \subseteq S_{LFSR}$ ). Fortunately, the symbols produced by the LFSR have a pseudo-random property. If  $n$  is the symbol length, then as was shown in [Könemann 91], if the number of specified bits in an  $n$ -bit block of test data is 20 less than the size of the LFSR, then the probability of the  $n$ -bit block of test data not matching one of the symbols in  $S_{LFSR}$  is negligibly small (less than  $10^{-6}$ ). In [Hellebrand 95], a multiple-polynomial technique was presented which reduces the size of the seed information to just 1 bit more than the maximum number of specified bits in an  $n$ -bit block. Thus, the compression that is achieved with this approach for a symbol length of  $n$  is equal to the ratio of  $n$  to the maximum number of specified bits in an  $n$ -bit block plus 1. Figure 7.5 shows a graph of the compression for LFSR reseeding for all possible symbol lengths. As can be seen, no compression is achieved for short symbol lengths where the maximum number of specified bits is equal to the symbol length. Compression does not start to occur until the symbol length becomes larger than 20. The compression steadily improves as the symbol length is increased, but in general, it cannot exceed the total percentage of don't cares in the test data. For the test set in Fig. 7.5, the percentage of don't cares is 94%.

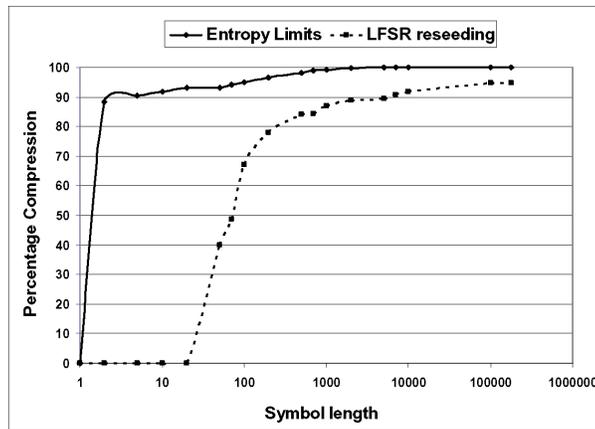


Figure 7.5: LFSR reseeding compression versus symbol length for *s13207*

## 7.5 Summary

In this chapter, we show how entropy theory can be used to calculate theoretical limits to the amount of test data compression that can be achieved with various coding techniques. These limits are useful as a measure of how much improvement is possible over existing test data compression schemes. They can be used to identify coding techniques where there is not much scope for improvement as well as identify coding techniques that hold promise for fruitful research. The relationship of symbol partitioning and don't care filling to entropy was discussed. An area for future research is to look at new coding techniques that can exploit this to achieve greater compression.

## Chapter 8

# Conclusions and Future Work

The continuing advances in design technology and the popularity of the system-on-a-chip design paradigm has fuelled even more increasing levels of integration that result in complex designs and SoCs. Increasing design reuse has led to the situation where test generation dominates the overall design time. Conventional testing solutions like using an external tester are becoming difficult as well as expensive. Test data compression solutions to the test volume and tester bandwidth problems are becoming increasingly popular which is reflected by the fact that all the major CAD tools vendors now offer some kind of test data compression solutions along with their test generation tools.

The ability to use an existing resource such an embedded processor to test the other cores in a SoC is a powerful tool that can not only be used to reduce dependence on expensive external testers but also to help overcome the inherent problems with logic BIST. Extending this concept further, there are ideas of having dedicated blocks on SoCs that provide on-chip support infrastructure. They can not only be used to enhance defect detection, test, diagnosis, yield optimization of very deep submicron technologies but also for measuring timing specifications and silicon debugging.

One possible future research area related to the above would be a methodology for processor self test. Functional tests utilize the instruction set of the processor and are a major part of the test strategy of a processor. The methodology should

incorporate functional test compression thereby reducing requirements on ATEs while the decompression and application of the functional tests should be done by the processor itself.

In chapter 6, a technique was proposed that modified the circuit itself to increase the compression obtained using decompressors. The advantage of inverting scan cells is that it doesn't require any additional hardware. But there might be a potential disadvantage because of last minute design changes that may need the test vectors to be regenerated. The next logical step would be techniques that will improve the compression by modifying the decompressor rather than the actual circuit. In other words, schemes where the decompressor is dynamically *reconfigurable* to enhance the compression.

The study of entropy of test sets in chapter 7 to calculate the fundamental limits to compression has opened up wide areas for future research. Entropy only gives a theoretical limit on the compression without any consideration for the complexity of the algorithm that is reflected as the area overhead. Since area is an important parameter of any decompressor design, a calculation of the limits taking area into consideration will be very useful to identify the efficiency of current techniques and the scope for improvement. The problem of finding an optimum way to minimize the entropy for variable length symbols is another interesting problem that might lead to further improvements in test data compression.

# Bibliography

- [Aitken 89] Aitken, R.C., and V.K. Agarwal, “A Diagnosis Method Using Pseudo-random Vectors without Intermediate Signatures”, *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 574-580, 1989.
- [Balakrishnan 02] Balakrishnan, K.J., and N.A. Touba, “Matrix-based Test Vector Decompression Using an Embedded Processor”, *Proc. of IEEE Symposium on Defect and Fault Tolerance*, pp. 159-165, 2002.
- [Balakrishnan 03a] Balakrishnan, K.J., and N.A. Touba, “Deterministic Test Vector Decompression in Software Using Linear Operations”, *Proc. of VLSI Test Symposium*, pp. 225-231, 2003.
- [Balakrishnan 03b] Balakrishnan, K.J., and N.A. Touba, “Scan-based BIST Diagnosis Using an Embedded Processor”, *Proc. of IEEE Symposium on Defect and Fault Tolerance*, pp. 209-216, 2003.
- [Balakrishnan 04a] Balakrishnan, K.J., and N.A. Touba, “Matrix-Based Software Test Data Decompression for Systems on a Chip”, *Journal of Systems Architecture (JSA)*, Vol. 50, Issue 5, pp. 247-256, Apr. 2004.
- [Balakrishnan 04b] Balakrishnan, K.J., and N.A. Touba, “Relating Entropy Theory to Test Data Compression”, *Proc. of IEEE European Test Symposium*, 2004.
- [Balakrishnan 04c] Balakrishnan, K.J., and N.A. Touba, “Improving Encoding Efficiency of Linear Decompressors Using Scan Inversion”, *Proc. of International Test Conference*, 2004.

- [Bayraktaroglu 00a] Bayraktaroglu, I., and A. Orailoglu, "Improved Fault Diagnosis in Scan-based BIST via Superposition", *Proc. of Design Automation Conference*, pp. 55-58, 2000.
- [Bayraktaroglu 00b] Bayraktaroglu, I., and A. Orailoglu, "Deterministic Partitioning Techniques for Fault Diagnosis in Scan-Based BIST", *Proc. of International Test Conference*, pp. 273-282, 2000.
- [Bayraktaroglu 01] Bayraktaroglu, I., and A. Orailoglu, "Test Volume and Application Time Reduction Through Scan Chain Concealment", *Proc. of Design Automation Conference*, pp. 151-155, 2001.
- [Bayraktaroglu 03] Bayraktaroglu, I., and A. Orailoglu, "Decompression Hardware Determination for Test Volume and Time Reduction through Unified Test Pattern Compaction and Compression", *Proc. of VLSI Test Symposium*, pp. 113-118, 2003.
- [Brglez 89] Brglez, F., D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits", *Proc. of International Symposium on Circuits and Systems*, pp. 1929-1934, 1989.
- [Chandra 01a] Chandra A., and K. Chakrabarty, "System-on-a-chip test data compression and decompression architectures based on Golomb Codes", *IEEE Transactions on Computer-Aided Design*, Vol. 20, No. 3, pp. 355-368, March 2001.
- [Chandra 01b] Chandra, A., and K. Chakrabarty, "Frequency-Directed Run Length (FDR) Codes with Application to System-on-a-Chip Test Data Compression", *Proc. of VLSI Test Symposium*, pp. 42-47, 2001.
- [Chandra 02] Chandra, A., K. Chakrabarty, and R.A. Medina, "How Effective are Compression Codes for Reducing Test Data Volume ?", *Proc. VLSI Test Symposium*, pp. 91-96, 2002.
- [Chandramouli 03] Chandramouli, M., "How to Implement Deterministic Logic Built-In Self-Test (BIST)", *Compiler: A Monthly Magazine for Technologists Worldwide*, Synopsys, Jan. 2003.

- [Chatterjee 95] Chatterjee, M., and D.K. Pradhan, "A New Pattern Biasing Technique for BIST", *Proc. of VLSI Test Symposium*, pp. 417-425, 1995.
- [Cover 91] Cover, T. M., and J.A. Thomas, *Elements of Information Theory*, John Wiley and Sons Inc., 2nd Edition.
- [Cullen 97] Cullen, C.G., *Linear Algebra with Applications*, Addison-Wesley, ISBN 0-673-99386-8, 1997.
- [Damarla 95] Damarla, T.R., C.E. Stroud, and A. Sathaye. "Multiple Error Detection and Identification via Signature Analysis", *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 7, pp. 193-207, 1995.
- [Ghosh-Dastidar 99] Ghosh-Dastidar, J., D. Das, and N.A. Touba, "Fault Diagnosis in Scan-Based BIST Using Both Time and Space Information", *Proc. of International Test Conference*, pp. 95-102, 1999.
- [Gonciari 02] Gonciari, P. T., B.M. Al-Hashimi, and N. Nicolici, "Improving compression ratio, area overhead, and test application time for systems-on-a-chip test data compression/ decompression", *Proc. Design Automation and Test in Europe*, pp. 604-611, 2002.
- [Gonciari 03] Gonciari, P. T., B.M. Al-Hashimi, and N. Nicolici, "Variable-Length Input Huffman Coding for System-on-a-chip Test", *IEEE Trans. on Computer-Aided Design*, Vol. 22, No 6, pp. 783-796, 2003.
- [Hamzaoglu 98] Hamzaoglu, I., and J.H. Patel, "Test Set Compaction Algorithms for Combinational Circuits", *Proc. of International Conf. on Computer-Aided Design (ICCAD)*, pp. 283-289, 1998.
- [Hamzaoglu 99] Hamzaoglu, I., and J.H. Patel, "Reducing Test Application Time for Full Scan Embedded Cores", *Proc. of International Symposium on Fault Tolerant Computing*, pp. 260-267, 1999.
- [Hellebrand 95a] Hellebrand, S., J. Rajski, S. Tarnick, S. Venkataraman and B. Courtois, "Built-In Test for Circuits with Scan Based on Reseeding of Multiple-

- Polynomial Linear Feedback Shift Registers”, *IEEE Trans. on Computers*, Vol. 44, No. 2, pp. 223-233, Feb. 1995.
- [Hellebrand 95b] Hellebrand, S., B. Reeb, S. Tarnick, and H.-J. Wunderlich, “Pattern Generation for a Deterministic BIST Scheme”, *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 88-94, 1995.
- [Hsu 01] su, F.F., K. M. Butler, J. H. Patel, “A Case Study on the Implementation of the Illinois Scan Architecture”, *Proc. of International Test Conference*, pp. 538-547, 2001.
- [Huffman 52] Huffman, D. A., “A method for the construction of minimum redundancy codes”, *Proc. IRE*, Vol. 40, pp. 1098-1101, 1952.
- [Ichihara 00] Ichihara, H., K. Kinoshita, I. Pomeranz, S.M. Reddy, “Test Transformation to Improve Compaction by Statistical Encoding”, *Proc. International Conference on VLSI Design*, pp. 294-299, 2000.
- [Jas 98] Jas, A., and N.A. Touba, “Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs”, *Proc. of International Test Conference*, pp. 458-464, 1998.
- [Jas 99] Jas, A., J.G. Dastidar, and N.A. Touba, “Scan Vector Compression/Decompression Using Statistical Coding,” *Proc. VLSI Test Symposium*, pp. 114-120, 1999.
- [Jas 00] Jas, A., B. Pouya, and N.A. Touba, “Virtual Scan Chains: A Means for Reducing Scan Length in Cores”, *Proc. of VLSI Test Symposium*, pp. 73-78, 2000.
- [Jas 03] Jas, A., J.G. Dastidar, M-E. Ng, N.A. Touba, “An Efficient Test Vector Compression Scheme Using Selective Huffman Coding”, *IEEE Trans. on Computer-Aided Design*, Vol. 22, No 6, pp. 797-806, 2003.
- [Karpovsky 93] Karpovsky, M.G., and S.M. Chaudhry, “Design of Self Diagnostic Boards by Multiple Signature Analysis”, *IEEE Trans. on Computers*, Vol. 42, No. 9, pp. 1035-1043, Sept. 1993.
- [Khoche 00] Khoche, A., and J. Rivoir, “I/O Bandwidth Bottleneck for Test: Is it Real ?”, *Proc. of International Workshop on Test Resource Partitioning*, 2000.

- [Kiefer 97] Kiefer, G., and H.-J. Wunderlich, "Using BIST Control for Pattern Generation", *Proc. of International Test Conference*, pp. 347-355, 1997.
- [Kiefer 98] Kiefer, G., and H.-J. Wunderlich, "Deterministic BIST with Multiple Scan Chains", *Proc. of International Test Conference*, pp. 1057-1064, 1998.
- [Knuth 97] Knuth, D.E., *The Art of Computer Programming*, Vol. 2. Addison-Wesley, Reading, MA, 3rd edition, ISBN: 0201896842, 1997.
- [Könemann 91] Könemann, B., "LFSR-Coded Test Patterns for Scan Designs", *Proc. of European Test Conference*, pp. 237-242, 1991.
- [Köemann 01] Könemann, B., "A SmartBIST Variant with Guaranteed Encoding", *Proc. of Asian Test Symposium*, pp. 325-330, 2001.
- [Krishna 01] Krishna, C.V., A. Jas, and N.A. Touba, "Test Vector Encoding Using Partial LFSR Reseeding", *Proc. of IEEE International Test Conference*, pp. 885-893, 2001.
- [Krishna 02] Krishna, C.V., and N.A. Touba, "Reducing Test Data Volume Using LFSR Reseeding with Seed Compression", *Proc. of International Test Conference*, pp. 321-330, 2001.
- [Krishna 03] Krishna, C.V., and N.A. Touba, "Adjustable Width Linear Combinational Scan Vector Decompression", *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 863-866, 2003.
- [Krishna 04] Krishna, C.V., and N.A. Touba, "3-Stage Variable Length Continuous-Flow Scan Vector Decompression Scheme", *Proc. of VLSI Test Symposium*, 2004.
- [Li 03] Li, L., and K. Chakrabarty, "Test data compression using dictionaries with fixed-length indices", *Proc. VLSI Test Symposium*, pp. 219-224, 2003.
- [Liu 02] Liu, C., K. Chakrabarty and M. Goessel, "An Interval-Based Diagnosis Scheme for Identifying Failing Vectors in a Scan-BIST Environment", *Proc. of Design, Automation and Test in Europe (DATE) Conference*, pp. 382-386, 2002.

- [McAnney 87] McAnney, W.H., and J. Savir., "There is Information in Faulty Signatures", *Proc. of International Test Conference*, pp. 630-636, 1987.
- [Mitra 03] Mitra, S., and K.S. Kim, "XMAX: X-tolerant architectures for Maximal Test Compression", *Proc. of International Conference on Computer Design*, pp. 326-330, 2003.
- [Rajski 98] Rajski, J., J. Tyszer, and N. Zacharia, "Test Data Decompression for Multiple Scan Designs with Boundary Scan", *IEEE Trans. on Computers*, Vol. 47, No. 11, pp. 1188-1200, Nov. 1998.
- [Rajski 99] Rajski, J., and J. Tyzer, "Diagnosis of Scan Cells in BIST Environment", *Proc. IEEE Transactions on Computers*, Vol. 48, No. 7, pp. 724-731, July 1999.
- [Rajski 02] Rajski, J., et al., "Embedded Deterministic Test for Low Cost Manufacturing Test", *Proc. of International Test Conference*, pp. 301-310, 2002.
- [Rao 03] Rao, W., I. Bayraktaroglu, and A. Orailoglu, "Test Application Time and Volume Compression through Seed Overlapping", *Proc. of Design Automation Conference*, pp. 732-737, 2003.
- [Reda 02] Reda, S., and A. Orailoglu, "Reducing Test Application Time Through Test Data Mutation Encoding", *Proc. of Design, Automation, and Test in Europe*, pp. 387-393, 2002.
- [Reddy 02] Reddy, S. M., K. Miyase, S. Kajihara, I. Pomeranz, "On test data volume reduction for multiple scan chain designs", *Proc. VLSI Test Symposium*, pp. 103-108, 2002.
- [Savir 88] Savir, J., and W. H. McAnney, "Identification of Failing Tests with Cyclic Registers", *Proc. of International Test Conference*, pp. 322-328, 1988.
- [Sinanoglu 02] Sinanoglu, O., I. Bayraktaroglu, and A. Orailoglu, "Test Power Reduction Through Minimization of Scan Chain Transistions", *Proc. of VLSI Test Symposium*, pp. 166-171, 2002.

- [Stroud 95] Stroud, C.E., and T.R. Damarla, "Improving the Efficiency of Error Identification via Signature Analysis", *Proc. of VLSI Test Symposium*, pp. 244-249, 1995.
- [Touba 95a] Touba, N.A., and E.J. McCluskey, "Transformed Pseudo-Random Patterns for BIST", *Proc. of IEEE VLSI Test Symposium*, pp. 410-416, 1995.
- [Touba 95b] Touba, N.A., and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST", *Proc. of IEEE International Test Conference*, pp. 674-682, 1995.
- [Touba 01] Touba, N.A., and E.J. McCluskey, "Bit-Fixing in Pseudo-Random Sequences for Scan BIST", *IEEE Transactions on Computer-Aided Design*, Vol. 20, No. 4, pp. 545-555, Apr. 2001.
- [Venkataraman 93] Venkataramann, S., J. Rajski, S. Hellebrand, and S. Tarnick, "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers", *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 572-577, 1993.
- [Volkerink 03] Volkerink, E.H., and S. Mitra, "Efficient Seed Utilization for Reseeding Based Compression", *Proc. of VLSI Test Symposium*, pp. 232-237, 2003.
- [Wolff 02] Wolff, FG, C. Papachristou, "Multiscan-based Test Compression and Hardware Decomposition Using LZ77", *Proc. International Test Conference*, pp. 331-338, 2002.
- [Wu 96] Wu, Y., and S. Adham, "BIST Fault Diagnosis in Scan-Based VLSI Environment", *Proc. of International Test Conference*, pp. 48-57, 1996.
- [Wunderlich 96] Wunderlich, H.-J., and G. Kiefer, "Bit-Flipping BIST", *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 337-343, 1996.
- [Wurtenberger 03] Wurtenberger, A., C. Tautermann, S. Hellebrand, "A Hybrid Coding Strategy for Optimized Data Compression", *Proc. International Test Conference*, pp. 451-459, 2003.
- [Zacharia 95] Zacharia, N., J. Rajski, and J. Tyszer, "Decompression of Test Data Using Variable-Length Seed LFSRs", *Proc. of VLSI Test Symposium*, pp. 426-433, 1995.

[Zacharia 96] Zacharia, N., J. Rajski, J. Tyszer, and J. Waicukauski, “Two Dimensional Test Data Decompressor for Multiple Scan Designs”, *Proc. of International Test Conference*, pp. 186-194, 1996.

[Zorian 98] Zorian, Y., E.J. Marinissen, S. Dey, “Testing embedded core based system chips”, *Proc. International Test Conference*, pp. 130-143, 1998.

# Vita

Kedarnath Jayaraman Balakrishnan was born in Madras, India, on April 25, 1979, the son of Umaa and Jayaraman. Less than two years later he had company in Badri, with whom he has spent most of his time fighting. His parents moved to New Delhi very soon where they have been living since then.

After completing his schooling at D. T. E. A. School, Lodi Estate, he joined the Indian Institute of Technology at Bombay in July 1996. Four years later, he received the degree of Bachelor of Technology in Electrical Engineering from I.I.T. Bombay. In August 2000, he joined the Department of Electrical and Computer Engineering at the University of Texas, Austin for graduate studies. He received a M.S.E. degree in May 2002 and is hoping to be awarded a Doctor of Philosophy degree in August 2004. He will start a new adventure as a research staff member at NEC laboratories, Princeton, New Jersey.

Permanent Address: 308, Naveen Kunj Apartments,  
Plot 22, Pocket 6, Nasirpur Road,  
Dwarka Ph-I, New Delhi, 110045  
India

This dissertation was typeset with  $\text{\LaTeX} 2_{\epsilon}$  by the author.