Merging Concurrent Checking and Off-line BIST

by

Xiaoling Sun

B.Eng., Shenyang Industrial University, 1982
M.Sc., Concordia University, 1988

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. M. Serra, Supervisor (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. R. N. Horspool, Departmental Member (Department of Computer Science)

Dr. P. Guibaly, Outside Member (Department of Elec. & Comp. Eng.)

Dr. H. Kwok, Outside Member (Department of Elec. & Comp. Eng.)

Dr. Y. Zorian, External Examiner (AT&T Bell Laboratories, Princeton, USA)

Supervisor: Dr. Micaela Serra

Abstract

This dissertation encompasses primarily design for testability (DFT) problems of concurrent checking and structural off-line Built-In Self-Test. We present a new DFT method, which employs cyclic code checking as a medium to combine the concurrent checking and signature analysis in a built-in fashion. It uses bit-sliced linear feedback shift registers (LFSRs) or linear cellular automata registers (LCARs) as the implementation mechanism. A circuit under test designed in this method supports both on-line and off-line testability with shared hardware resources. It has comparable on-line error-detecting ability to the conventional error-detecting codes and without affecting the high fault coverage of off-line signature analysis. This testing scheme complies with the IEEE boundary-scan standard and is applicable to general circuitry.

Evaluations of the proposed scheme are carried out with respect to the area overhead, performance and testing time, design complexity, pin count, and fault coverage.

The concatenation properties of LCARs are introduced and recent developments in related issues are reviewed. Finally, a new area estimation method for circuit design is presented to ease silicon cost measurement.

Examiners:

Dr. M. Serra, Supervisor (Department of Computer Science)

Dr. D M. Miller, Departmental Member (Department of Computer Science)

Dr. R. N. Horspool, Departmental Member (Department of Computer Science)

Dr. F. El Guibaly, Outside Member (Department of Elec. & Comp. Eng.)

Dr. H. Kwok, Outside Member (Department of Elec. & Comp. Eng.)

Dr. Y. Zorian, External Examiner (AT&T Bell Laboratories, Princeton, USA)

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Micaela Serra, for her guidance, supervision and help throughout my Ph.D program. I would like to acknowledge my external examiner, Dr. Yavent Zorian, for his criticism and suggestions on the final version of this dissertation.

I would like to thank Drs. Jon. C. Muzio and D. Michael Miller for their encouragement, advise and help during the years of my doctoral studies.

My thanks go to all the present and past members of the VLSI design & test group in the last four years for their assistance and friendship. My special thanks go to Mr. Michael Whitney for proof reading of my dissertation, to Mr. R. Byrne, Mr. P. Walsh and Mr. K. Cattle for their helpful discussions in the early stage of my Ph.D research, to Mr. D. Wessels, Mr. S. Zhang and Mr. R. Byrne for their work on the fault simulators, and to Mr. J. Walkowicz, and Mr. W. Kastelic for their technical expertise, cooperation and help.

I would like to acknowledge Shenyany Institute of Computing Technology, Academia Sinica, for providing me with scholarships in the early years of my graduate study, to University of Victoria, for offering me the University of Victoria Fellowships, and to the Division of Computer Engineering, the Department of Electrical Engineering and the Faculty of Engineering at University of Alberta, for their support at the finishing stage of my dissertation.

Finally, I would like to thank my parents and my sister for their love, understanding, support, and sacrifices through all these years.

Chapter 1

Introduction

The reliability of digital systems depends on testing – that is, the determination of whether a circuit is manufactured properly and behaves correctly. Digital testing encompasses logic and parametric tests. *Logic testing* concerns the logical correctness of a *circuit under test* (CUT), while *parametric testing* examines the circuit parameters such as current, voltage, time delay and power consumption. This dissertation is devoted to logic testing issues. The term *testing* is used to refer to logic testing.

As circuit density rapidly increases, testing has become more difficult because of increased system complexity and decreased circuit accessibility. In a testing process, only a "pass/fail" signal is required to indicate if a circuit is good or not, as opposed to *diagnostic* methods, where the location of defects is required.

It is important to present new research in its appropriate context. Figure 1.1 illustrates a taxonomy of logic testing techniques. As shown, logic testing can be classified into two main categories: *external test* and *Built-In Self-Test (BIST)* [46, page 131]. External test uses a tester, external to the system, to stimulate a circuit. Such general purpose testers are very expensive and not necessarily available to all designers. Moreover, a large volume of data needs to be handled by the testers,

resulting in long testing times and high testing cost.

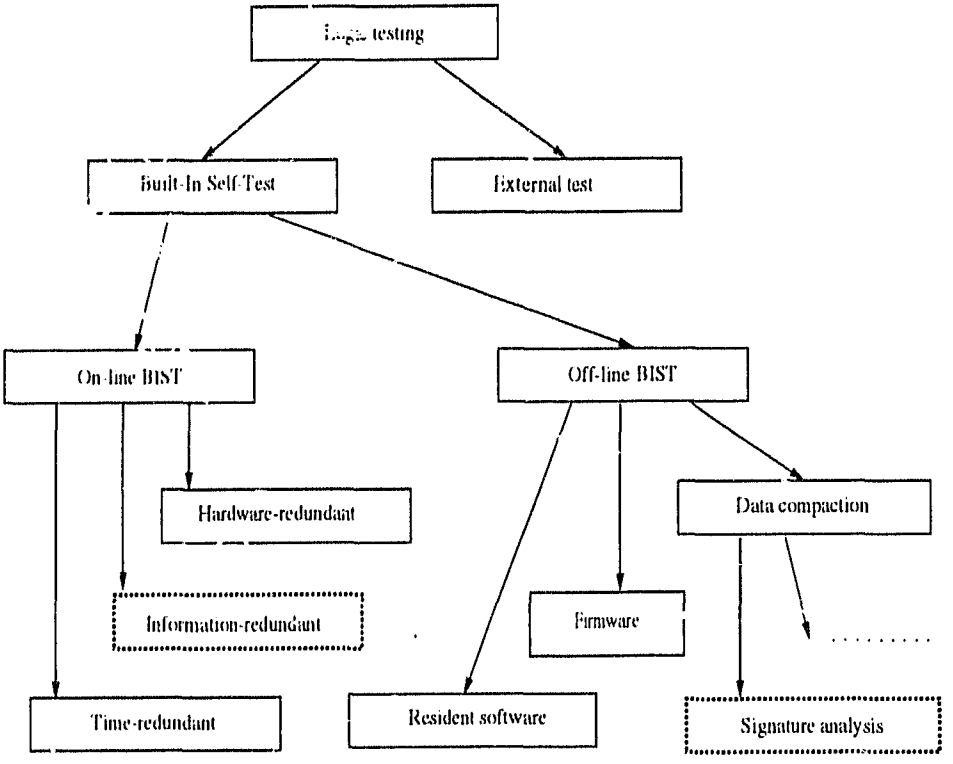


Figure 1.1: Taxonomy of testing

A viable alternative to external test is BIST, which in general refers to the inclusion of on-chip circuitry to facilitate testing. BIST techniques often imply using some *design for testability* (DFT) methods to reduce test cost and increase circuit testability. DFT techniques include *ad hoc* approaches such as partitioning [2, pages 314-315] and inserting test points [2, page 347], and *structural* approaches such as scan techniques [2, pages 358-407]. A BIST structure consists of two key components: a mechanism to provide input stimuli to the CUT, and a mechanism to evaluate the circuit responses. BIST techniques can be further divided into on-line and off-line methods. *On-line BIST* allows the circuit to be tested as it is performing its intended functions, while *off-line BIST* typically requires the CUT to suspend normal operation and enter a separate test mode.

As shown in Figure 1.1, on-line BIST techniques can be further characterized into *hardware-redundant*, *time-redundant* and *information-redundant* categories. Hardware-redundant techniques vary from self-checking circuits at the gate level through duplications at the module level, up to replicated computers at the system level [46]. Time-redundancy uses additional time to repeat computations in order to achieve fault detection. Information-redundant approaches employ error detecting codes such as Berger, residue and cyclic redundancy checks as part of the function specifications. In this dissertation, the term *concurrent checking* is used to refer to the information-redundant methods. One focus of the new research falls in this category.

Off-line BIST techniques have been widely accepted by design engineers in practical applications. These techniques can be grouped into *resident software*, *firmware* and *data compaction* methods. Various data compaction techniques are available [2, pages 421-452]. The best known is *signature analysis* [2, pages 432-448]. It is based on the concept of cyclic redundancy checking (CRC), and realized in hardware using linear feedback shift registers (LFSRs). A comprehensive exposition of these background subjects can be found in [2, 4, 24, 46]. The new research in this dissertation encompasses data compaction techniques using signature analysis and, in fact, presents methods for merging such off-line BIST techniques with concurrent checking.

Scan-based test is a DFT technique that provides the means to access internal parts of a circuit. It converts a sequential circuit test problem into a combinational one by connecting storage elements into a chain of shift registers, such that test data can be shifted in for testing and circuit responses can be shifted out for examination. There are many variations of scan design and the details can be found in [2, 358-395]. In this dissertation, the term *off-line BIST* refers only to testing methods that combine signature analysis and scan-based DFT techniques. The IEEE Standard

1149.1 Standard Test Access Port and Boundary Scan Architecture [1], also known as the boundary-scan standard, is based on scan techniques. It declares the arrival of the era of design for testability, mainly enhancing off-line testability. The new testing scheme presented in this dissertation can be adapted to support this standard, with enhanced on-line testability.

The main goal of this research is to combine concurrent checking and off-line BIST techniques, so that the hardware resources involved in the two types of testing can be shared, and the overall silicon cost can be minimized. To achieve this merging, it is essential to find a means to bridge the two different test techniques. There are two basic related issues: an appropriate error-detecting code for the concurrent checking and a suitable implementation mechanism for the off-line BIST. The new *design for test* method proposed in this research results in a new testing architecture, where the hardware resources are shared in both on-line and off-line tests. It supports the IEEE boundary-scan standard and is applicable to general circuitry. In the course of presenting the new merging techniques, we also discuss some other new research ideas for the concatenation properties of linear cellular automata, for the estimation of area overhead and for the evaluation of error coverage for some concurrent checking techniques.

Some initial approaches to this problem utilizing effectively the built-in hardware resources and perform on-line testing are reported in [30, 48, 51]. Testing schemes and DFT methods that merge signature analysis and scan-based test, or boundary scan techniques, can be found in [2, pages 496-513] and [17].

The remainder of this dissertation is organized as follows. The general background is provided in Chapter 2. Additional details are introduced as necessary in later chapters. Chapter 3 reviews the previous work in merging concurrent checking and off-line BIST, and discusses the main obstacles. Chapter 4 explores the concatenation properties of linear cellular automata registers (LCARs), which play a

key role in our solution to the merging.

Chapter 5 presents the new testing scheme. Three modified structures for improving the system performance are introduced. A design template of a benchmark circuit is provided to demonstrate the applicability of this study.

Chapter 6 is dedicated to evaluation measures. A new area cost estimation method for general circuitry is suggested. The cost evaluation of the proposed schemes in terms of silicon cost, testing time, design complexity, and pin count are presented.

Chapter 7 is devoted to the error coverage capabilities of LFSR/LCAR based cyclic codes, as compared to conventional error-detecting codes such as Berger and residue codes. Concluding remarks and topics of future research are presented in Chapter 8.

There are two appendices. Appendix 1 contains the tables of LCAR self concatenation for length 2 to 16 LCARs, concatenating up to length 64. Appendix 2 is the manual page of the *Concurrent Checking code generator and MINImizer* (ccmini) program, developed by this author to automate the encoding process of concurrent checking.

Chapter 2

Background

This chapter provides the necessary background material. We first present general fault models used in testing. Then, we consider the problems that Built-In Self-Test addresses, and explore two widely used testing techniques: pseudorandom test generation and data compaction. Finally, we introduce two extensively used circuit structures for test generation and data compaction: linear feedback shift registers (LFSRs) and linear cellular automata registers (LCARs).

2.1 Faults and Fault Models

The elements of digital systems are subject to physical defects that can cause them to malfunction. The defects can be manufacturing defects at chip, board or system level, or service-related defects such as burnout due to overloading. To avoid dealing directly with the large number of physical defects, the logical behaviour of the defects are modeled as *faults*, such that their effects can be examined.

A fault model represents a range of physical defects. Testing is always defined with respect to a set of faults. Faults can be modeled at different levels of the design hierarchy from layout geometry, transistor, logic gate, up to functional and system

levels. Faults can be classified according to their nature of appearance: a *permanent fault* is always present and does not disappear, or change its nature, during testing; a *transient fault* is present in some intervals of time and absent in others. There are two main types of faults with respect to their effects on circuit behaviours: *sequential faults* and *combinational faults.* A sequential fault is caused by some defects, e.g. the bridging of two or more signal lines in a circuit, such that a feedback path is created to form a new state in the network. Combinational faults include the faults which do not have such feedback paths.

Many fault models have been proposed, including the stuck-at model, the stuck open model, the delay model and the current test model. The most commonly used fault model is the *stuck-at* fault model [2, page 94] at logic gate level. This model has been generalized to apply to any fault condition that causes a logic gate to behave as though one of its inputs or outputs is stuck at logical 1 or 0. A fault in this model is detected by applying a test vector that should set the line to the opposite of the value it is stuck at, then propagating the value of the line to the circuit output.

The stuck-open fault model is created to deal with the memory effects caused by breaks in a CMOS circuit [46, pages 8-10]. It models a transistor that is stuck-open (i.e. non-conducting). A test to detect a particular stuck-open transistor consists of two vectors. The first initializes the value of the line that the transistor in question drives, and the second creates a situation where the transistor must change the value of the line and also must propagate the new value of the line to an output.

The delay fault model is used to detect timing faults in the circuit. It models a fault that causes a signal line (or path of signal lines) to be slow to rise or slow to fall. To detect a delay fault, the appropriate transition on the faulty line must be generated, and the transition must be visible at an output.

Faults in *the current test* model are bridges between two nodes. To test for a

bridging fault between two nodes, a test vector is applied which drives the nodes to opposite values.

Faults can be single or multiple. The single fault model is used in most testing research due to the simplicity of analysis and the limited computational power to simulate multiple faults [2, page 94].

In this dissertation, unless otherwise noted, we consider both permanent and transient combinational faults, which can be single and multiple. Other fault models are also used, and they are introduced when needed.

2.2 Pseudorandom Test Pattern Generation

Test generation refers to techniques for generating an appropriate set of input stimuli for a circuit under test. Test generation is a complex problem. The most important measures of a test generation technique are the cost of test generation, the quality of the generated test, and the cost of applying the test. Traditional testing methods require generation of test vectors, which is an expensive process, and a large amount of memory may be required to store the set.

Exhaustive testing uses all possible input combinations as test patterns [4, page 10]. Exhaustive testing guarantees detection of all detectable combinational faults. However, the exponential growth of the required number of vectors limits the practical applicability of this method to circuits with less than twenty or so inputs.

Random testing involves generation of random test vectors. However, to achieve a high-quality test, a large set of random vectors is needed, requiring a long time to apply the test [4, page 177]. In *pseudorandom testing*, test vectors are generated deterministically. The vectors can be regenerated, and therefore do not need to be stored. Moreover, the generator itself can be implemented by simple and economical circuit structures (e.g. LFSRs). Hybrid approaches combining the above test

generation methods have also been explored in order to obtain the best features of each technique [4].

2.3 Data Compaction

To test a circuit, a large volume of output data, generated by applying a large number of input patterns to the CUT, has to be stored in memory and compared with the expected good response. A possible alternative is given by *data compaction* testing, where the output vector from a CUT is compacted into a much shorter compressed form, called a *signature*, with some loss of information. An invalid signature indicates the presence of errors in the output stream. Figure 2.1 shows the concept of data compaction techniques. A fault is detected if the signature $S(R')$ obtained from the CUT differs from the precomputed signature $S(R_0)$ of a fault-free circuit.

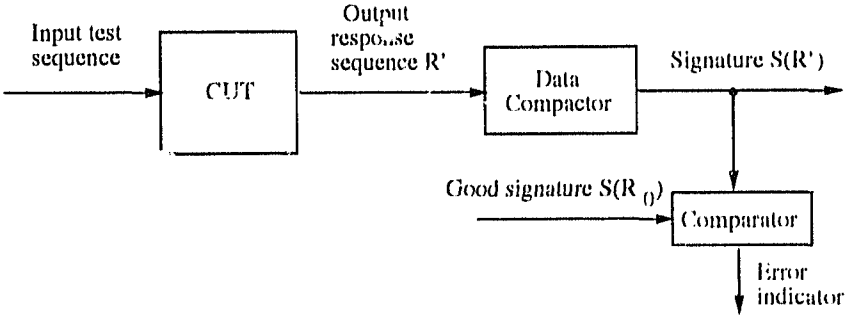


Figure 2.1: Data compaction techniques

Many methods can be used to obtain a signature: one's counting (the signature is the sum of the number of 1's in the circuit output), parity check (the signature is one bit signifying odd or even parity of the output), transition counting (the signature is the number of 1 to 0 and 0 to 1 transitions appearing on the information bits), syndrome testing (the signature is the normalized number of 1's in the circuit output

under exhaustive application of all possible input patterns), and Walsh spectra compression (the signature is all the Walsh coefficients or a carefully chosen subset of them). Each data compaction method has its own advantages and disadvantages. A detailed discussion of these techniques can be found in [2, 4].

The most commonly used data compaction technique in digital circuit testing is *signature analysis* [4, pages 109-144]. The actual implementation of signature analysis relies on linear feedback shift registers (LFSRs) as compactors. A LFSR performs a polynomial division over a binary field, where the input stream is the dividend polynomial and the LFSR itself implements the divisor polynomial [4, page 112]. At the end of the division, the last stage of the LFSR describes the remainder polynomial, which is used as the signature. The output of the LFSR, representing the coefficients of the quotient polynomial [4, page 115]. As an alternative, linear cellular automata registers (LCARs) can be used to obtain the signature of a CUT in a similar way. LCARs have been found having the same behaviour as LFSRs when used as data compactors [56].

Data compaction solves the test set storage problem and simplifies the comparison process as only the signature needs to be examined. One drawback of this process is that error patterns from a faulty circuit might be compacted into the same signature as the good circuit. This phenomenon is called *masking* or *aliasing* [4, page 114]. It is unavoidable since the compaction process introduces some loss of information. In the next two sections we provide some background on the operations of LFSRs and LCARs.

2.4 Linear Feedback Shift Registers (LFSRs)

2.4.1 Binary Sequence and Polynomial Transformations

A polynomial of degree [1] $m-1$,

$$P(x) = r_{m-1}x^{m-1} + r_{m-2}x^{m-2} + \ldots + r_1 x + r_0,$$

can be expressed by an m-bit binary sequence, $B = r_{m-1}r_{m-2}\ldots r_1 r_0$, by replacing all the nonzero coefficients of the polynomial with ones and inserting zeros for the corresponding zero coefficients. For example, the degree 4 polynomial, $P(x) = x^4 + x^3 + 1$, can be represented by the binary sequence, $B = 11001$. The reverse representation, i.e., a binary sequence to a polynomial transformation, can also be performed. The binary sequence $r_{m-1}r_{m-2}\ldots r_1 r_0$ corresponds to the coefficients of the appropriate powers of x, assuming that the left-most bit of B is the most significant bit. The transformations provide convenient ways to represent data in different domains.

2.4.2 LFSR Implementation of Polynomials

A *linear* binary network is one constructed from the following basic components: unit delays, modulo-2 adders and modulo-2 scalar multipliers. A network or sequential machine constructed of linear elements has the property that its response to a linear combination of inputs will preserve the principle of superposition[2]. As a result, linearity is an important consideration in the analysis of shift-register sequences. A shift register with a linear feedback network is called a linear feedback shift register

[1] The *degree* of a polynomial is the largest power of x with nonzero coefficient.

[2] *The Principle of Superposition: the response of a linear network to linear combination of* stimuli is the linear combination of the responses of the network to the individual stimuli (the linear network is initialized such that all storage elements (unit delay) are in the 0 state in each case).

(LFSR) [4, page 64]. It is composed of memory elements (unit delay) and XOR gates (Modulo 2 adders), controlled by a synchronous clock.

A LFSR represents in the structure of its feedback loops a particular polynomial which acts as the divisor in the operation of the linear finite state machine. There are three types of LFSR structures, type-1, type-2 [56] and hybrid LFSRs [63]. For example, Figure 2.2 shows the type-1 LFSR of the polynomial $P(x) = x^4 + x + 1$. Here, a non-zero coefficient, excluding the highest order bit, of a power of x is mapped into an XOR gate with a feedback connection between the output of the highest stage (corresponding to the highest power of x in the polynomial) and the XOR gate; a zero coefficient is corresponding to a direct connection from stage i to stage $i + 1$. In type-2 LFSRs, all feedbacks are connected to external XOR gates as shown in Figure 2.3. For a given polynomial, its type-1 and type-2 LFSRs have isomorphic behaviour (see section 4.1.1) [56]. A hybrid LFSR is the combination of the two types [63]. To simplify our discussion through this dissertation, we assume that type-1 LFSRs are used.

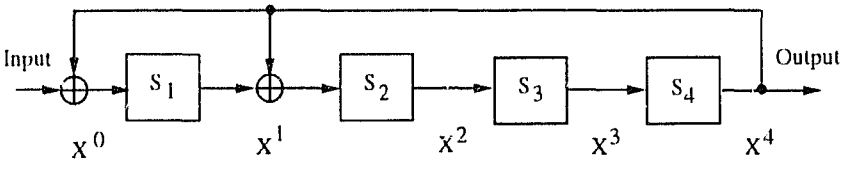


Figure 2.2: An example of a type-1 LFSR

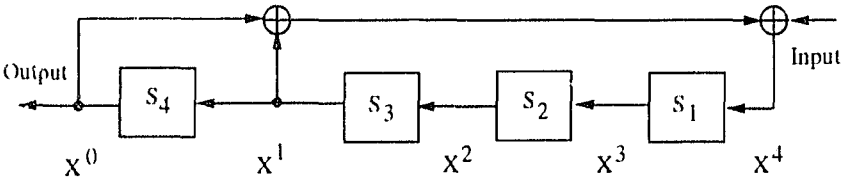


Figure 2.3: An example of a type-2 LFSR

The state equations which describe the transitions between states can be directly

obtained from a LFSR structure. A length-r LFSR consists of r stages of memory cells, and each of them can have a binary value "0" or "1". The state of the finite state machine at time t is represented by the contents of the r-bit memory cells. Let S_i^t and S_i^{t+1} denote the values of stage i of the LFSR at time t and $t+1$, respectively. For the type-1 LFSR shown in Figure 2.2, for example, we have

$$\begin{aligned} S_1^{t+1} &= Input^t \oplus S_4^t \\ S_2^{t+1} &= S_1^t \oplus S_1^t \\ S_3^{t+1} &= S_2^t \\ S_4^{t+1} &= S_3^t. \end{aligned}$$

A LFSR, as shown in Figure 2.2, performs polynomial division over a binary field [59]. The LFSR itself represents the divisor polynomial, $P(x)$; the serial input sequence represents the dividend polynomial, $M(x)$; the serial output is the quotient polynomial, $Q(x)$. At the end of m cycles, where m is the degree of $M(x)$, the last state of the LFSR describes the remainder polynomial, $R(x)$. Thus, this operation can be described as follows:

$$M(x) = P(x)Q(x) + R(x).$$

Table 2.1 shows the state table corresponding to the above state equations, for $M(x) = x^8 + x + 1 = 100000011$. At the end of the polynomial division, we have

$$\begin{aligned} Q(x) &= x^4 + x + 1 = 000010011 \\ R(x) &= x^2 + x = 0110. \end{aligned}$$

2.5 Linear Cellular Automata Registers (LCARs)

A *cellular automata* (CA) is defined as a uniform array of identical cells in an n-dimensional space [21, 56, 66]. Each cell is capable of having a particular state,

Clock	Input(M(x))	S_1^{t+1}	S_2^{t+1}	S_3^{t+1}	S_4^{t+1}	Output(Q(x))
		0	0	0	0	
1	1	1	0	0	0	0 (x^8)
2	0	0	1	0	0	0 (x^7)
3	0	0	0	1	0	0 (x^6)
4	0	0	0	0	1	0 (x^5)
5	0	1	1	0	0	1 (x^4)
6	0	0	1	1	0	0 (x^3)
7	0	0	0	1	1	0 (x^2)
8	1	0	1	0	1	1 (x^1)
9	1	0	1	1	0	1 (x^0)
	R(x)	(x^0)	(x^1)	(x^2)	(x^3)	

Table 2.1: An example of polynomial division

in our case a binary state. Further, each cell is restricted to local neighbourhood interaction only and has no global communication, i.e., the neighbourhood of a cell is typically taken to be the cell itself and all immediately adjacent cells. A *one-dimensional linear cellular automata* is a linear array of identical cells with left and right neighbours and linear operations. The algorithms the cells use to compute their successor states, based on the information received from their nearest neighbours, are referred as the *computation rules* [66]. For a binary one-dimensional linear CA, there exist a total of $2^{2^3} = 256$ distinct binary rules. However, a class of linear CA using only linear *rules 90* and *150* is extensively considered in the literature. In rule 90, the next state of a cell is the modulo-2 sum of the nearest neighbours' present state, while rule 150 derives the next state from the modulo-2 sum of the present states of the cell and both its left and right neighbours. These rules can be more formally given as

$$\text{Rule 90:} \quad S_i^{t+1} = S_{i-1}^t \oplus S_{i+1}^t$$

$$\text{Rule 150:} \quad S_i^{t+1} = S_{i-1}^t \oplus S_i^t \oplus S_{i+1}^t,$$

where S_i^t and S_i^{t+1} denote the state of stage i at time t and $t+1$, respectively.

A linear cellular automata register (LCAR) is a finite-state machine implementing a linear cellular automata. In this dissertation, we use LCARs with rules 90 and 150 only. Figure 2.4 depicts an example of a LCAR. The corresponding computation rules are shown for each cell. The LCAR is said to have *null boundary* conditions since the first and the last cells receive constant 0's, as shown.

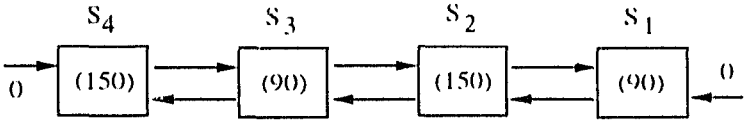


Figure 2.4: A LCAR

If we use "1" to denote a cell with rule 150 and "0" for a cell with rule 90, the LCAR structure can be represented by a binary sequence. The example LCAR in Figure 2.4 can be represented by "1010".

If there is no external input to a LFSR or a LCAR, it is said to have a null boundary condition. A LFSR or a LCAR with null boundary condition behaves as an autonomous machine that cycles through sequences of states[3]. A LFSR/LCAR without an external input can be used as a *pseudorandom test pattern generator* (PRPG) – the states of the machine are used as test patterns to stimulate a CUT in a pseudorandom order [4, pages 61-80].

2.6 Summary

Digital system testing requires an application of a suitable set of test patterns as input stimuli and the comparison of actual circuit responses with the correct responses. Various test pattern generation and data compaction techniques have been

[3]An autonomous machine is a finite state machine with no external input.

developed to meet this requirement. In Built-In Self-Test, the circuitry for test pattern generation and data compaction is built onto the chip itself. LFSRs are widely used as test pattern generators and data compactors in BIST applications [4]. Recently, there has been interest in using LCARs as a source of test stimuli [18, 21].

In this chapter, the fault types and models in digital circuit testing are briefly defined. We reviewed the two important testing techniques, pseudorandom test pattern generation and data compaction. Moreover, we described the operations of two well-known structures for pseudorandom test pattern generation and data compaction: linear feedback shift registers (LFSRs) and linear cellular automata registers (LCARs).

Chapter 3

Concurrent Checking and Off-line BIST

3.1 Introduction

Concurrent checking and off-line testing are complementary techniques in testing: concurrent checking is capable of detecting errors in real time while off-line testing provides testing during test mode; concurrent checking is able to catch *transient* faults which may not be detected by off-line testing. Off-line BIST has been widely used by design engineers in commercial chips, while concurrent checking is still considered to be expensive in silicon. However, it is possible, with proper coordination of design techniques, to expedite off-line testing by making use of on-line testing facilities during off-line testing [20, 51].

In the next three sections, we summarize three important testing and DFT techniques: concurrent checking, signature analysis, and scan-based test. A review of the literature combining the three techniques can be found in section 5. Finally, we present the problem statement for the merging of the techniques and state the justification and motivation of this research in section 6.

3.2 Concurrent Checking

Since off-line BIST techniques are conducted only at preset times and not continuously, a transient fault may go undetected if it does not happen to appear during a testing periods. Moreover, off-line BIST requires the circuit under test suspend normal operation and switch to a separate test mode. In safety critical applications, both high reliability and continuous operation are considered to be very important.

Concurrent checking includes error detection circuitry, which detects errors that occur during normal system operation [46]. Extra checking bits are added to the original circuit function so that the operations of the integrated function (including the original and the augmented checking circuitry) can be monitored when the original circuit is functioning. This is achieved through the use of error detecting codes. Various error-detecting codes well known in data communication can be used in concurrent checking. They are: Berger codes (the code bits are the binary representation of the number of 0's or 1's in the original data), residue codes (the checking bits are the residue of information bits), *m-out-of-n* codes (m out of n information bits are required to be "1"), and parity checking (an odd or even parity bit is added to the information bits). Where a parity code can be used for general circuitry, a residue code is more suitable to arithmetic units, and Berger and *m-out-of-n* codes are better for error-detection of circuit functions with unidirectional errors (1-to-0 or 0-to-1 errors only, but not both).

Error-detecting codes can be grouped into separable codes and non-separable codes. Separable codes are those in which the data can be extracted directly from the circuit outputs without the need for decoding, i.e. code and data outputs are disjoint, and the data outputs are unaltered by the existence of the added code bits. For non-separable codes, decoding of all outputs (the data bits plus the code bits) is necessary to extract the data information after the encoding process. One useful property of separable codes is that the data can be used by the system independent

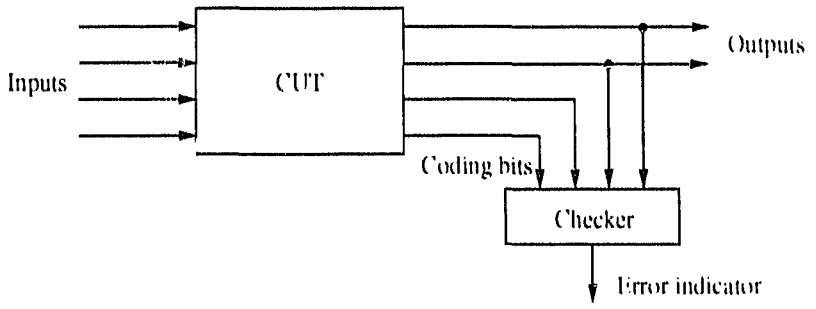


Figure 3.1: A concurrent checking organization

of the checking process. In contrast, with non-separable codes, the data is not available until a decoding process is completed. This may imply a time delay. The decoding circuitry required by the non-separable codes can also represent a significant hardware cost in comparison to the separable schemes.

Figure 3.1 shows a concurrent checking organization using a separable code. During normal operation, the checker collects both the normal outputs and the coding bits from the CUT for every input combination. An expected signature is computed from the outputs and it is compared with the carried coding bits. The checker signals the detection of a fault. The system continues operating for the next input vector.

One advantage of concurrent checking is that the data patterns used in normal operation also serve as test patterns, thus eliminating some explicit testing expenses. Another advantage is the ability to detect transient faults, with less silicon area overhead than that of hardware duplication. Conversely, there are several problems. The application patterns may not exercise all the storage elements or all the internal connection lines, such that defects, which exist in unexercised places, are not detected. Additional hardware circuitry is required for the augmented circuit and to implement the checkers. A circuit function with added checking bits usually results in an augmented circuit that is larger in size than the original one.

Complexities of the specially designed checkers vary in the different coding schemes. At least two (sometimes three or four) extra pins are required as error indicators, which can be considered expensive in a design because of the limited number of pins available. The degree of fault coverage provided by concurrent checking can also be less than that of off-line testing. These problems have limited the use of concurrent checking in the VLSI testing environment.

3.3 Signature Analysis

Signature analysis is an extensively used data compaction technique for off-line BIST, based on the concept of cyclic redundancy checking (CRC) and realized in hardware using linear feedback shift registers (LFSRs) as explained in section 2.4. Figure 3.2 shows a signature analysis organization of a single output network, where the LFSRs serve two purposes: data compaction and test pattern generation. The LFSR-A is used as a pseudorandom test pattern generator, and the states of the LFSR are used as test patterns to stimulate the CUT. The LFSR-B works as a data compactor.

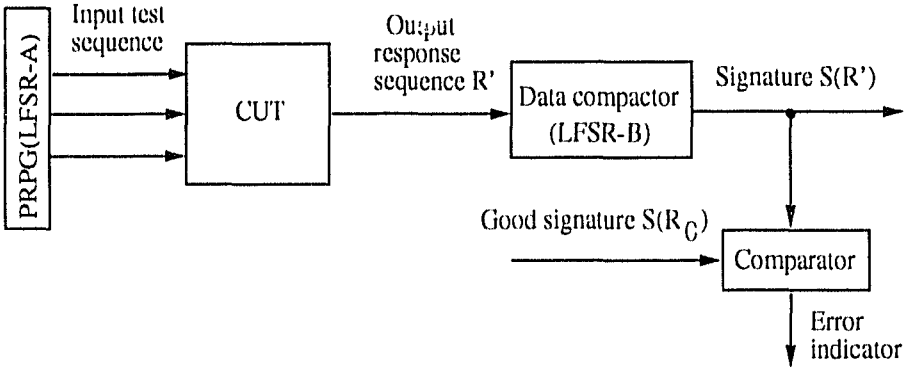


Figure 3.2: A signature analysis organization

For a multiple-output circuit, the overhead of a single-input signature analyzer on

every output would be high. Normally, a *multiple-input signature analyzer* (MISR) is used. The outputs of a CUT are connected to the MISR through XOR gates added to the shift lines between stages. A considerable amount of work has been done on using MISRs in BIST, both in theory and applications. A detailed description can be found in [2, 4].

For both LFSRs and MISRs, aliasing occurs when signatures for faulty and fault-free responses are the same. This results in a failure of detecting faults in a CUT. The probability of aliasing is primarily a function of the LFSR/MISR machine length, r, and it is asymptotic to 2^{-r} [4, page 116]. Aliasing can be minimized by choosing a long LFSR/MISR (e.g. $r = 16$), making an appropriate selection of the divisor polynomial [4], and applying a sufficient number of test vectors.

As for LFSRs, LCARs can be used as data pattern generators and data compactors. A LCAR with null boundary conditions is capable of generating pseudorandom test patterns. It performs data compaction if an input stream is fed into the machine from one of the boundary cells. For multiple-output circuits, a *multiple-input cellular automata* (MICA) can be used as a data compactor [45]. It has been shown that LCARs have better pseudorandomness than LFSRs [21] and have better fault coverage for certain types of faults [33].

3.4 Scan-based Test

An extremely large number of input test vectors may be necessary for an exhaustive test. For an n-input circuit containing s latches, the exhaustive input test set is $2^n \times 2^s = 2^{n+s}$ input vectors. For example, a circuit with 50 inputs requires a total of 2^{50} input vectors for an exhaustive test.

Scan design is a structural technique used to increase the controllability and observability of sequential circuits, thus making them easier to test. This is done by

modifying the circuit's memory elements and connecting them together as a large shift register. The contents of the memory elements are controlled and observed by shifting or scanning values into and out of the shift register. The circuit is tested as a strictly combinational circuit, with the memory elements being treated as primary inputs and outputs. The complexity of testing the sequential circuit directly is avoided. There are many variations of scan-based testing techniques but the main principle is the same. Figure 3.3 shows an example of such a system.

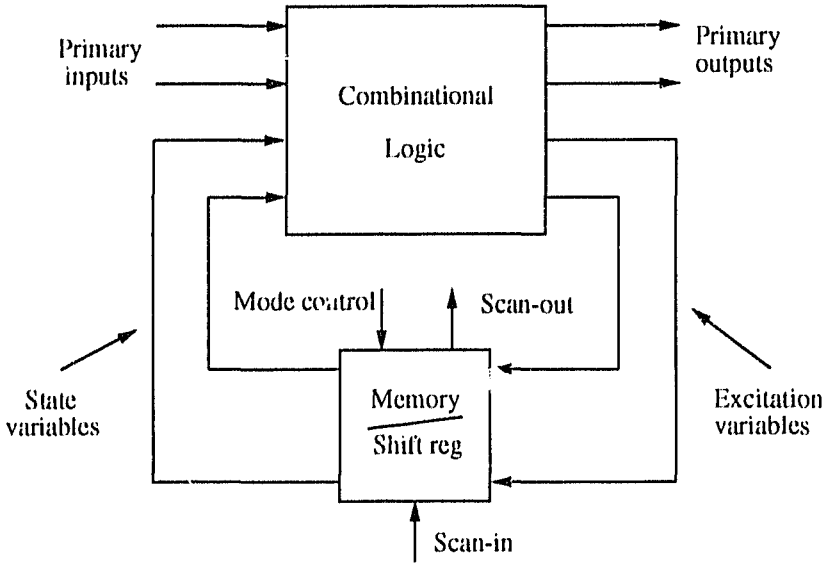


Figure 3.3: Scan-based test

IEEE Standard 1149.1 has identified the boundary-scan technique as means of addressing the problems of future highly complex, miniaturized board designs. It provides a framework for structured design for testability. The boundary-scan technique involves the inclusion of a shift-register latch (contained in a boundary-scan cell) adjacent to each functional component pin. This allows the signals at component boundaries to be controlled and observed using scan-based testing principles. Boundary-scan also provides an excellent basis for the use of BIST features during board testing. Figure 3.4 depicts a board of two circuits with boundary scan cells

[1].

Each device on the board has a 4 or 5-wire interface called the *Test Access Port (TAP)* that includes two control pins, TMS (test mode select) and TCK (test clock), two scan pins, TDI (test data in) and TDO (test data out), and optionally, the TRST (test reset input) pin. The test logic in boundary scan also includes a TAP controller and various registers [1]. The standard supports the following test modes:

(a) *Bypass:* in this mode a minimum length serial path is set up to bypass a chip. This allows access to the chip of interest in the minimum possible time, increasing test throughput.

(b) *Idcode/Usercode test*: this allows a binary data pattern to be read from the chip that identifies the manufacture, the part number, the variant, and (where appropriate) the programmed state; the identification register is used for this purpose. This information might be used, for example, to verify that correct IC has been mounted in each board location.

(c) *Sample/Normal test:* this allows a snapshot of normal operation of the component to be taken and examined. Data is latched in both the input and output registers. During this test only the boundary scan register is connected between TDI and TDO.

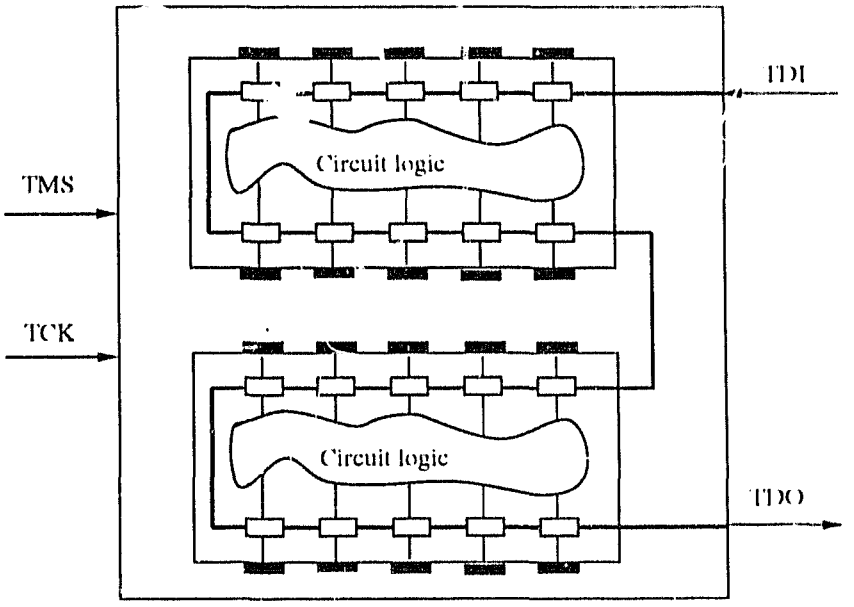
(d) *External test:* this mode is used to test the interconnections of the printed circuit board. The boundary scan register is the one and only register that is to be connected between TDI and TDO for data scanning purposes. Boundary scan register cells at output pins are used to apply test stimuli to the board, while those at input pins capture the test results flowing from another chip via the board. Capture results

are scanned out of the serially linked boundary scan registers of a board while the next set of test input values is scanned in.

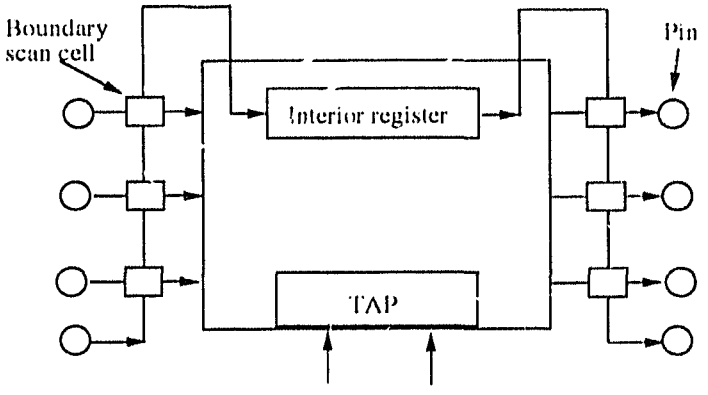
(e) *Internal test:* this is one of the two optional test modes defined by the standard. It provides the means to test the internal logic of the design. During internal test, test stimuli are shifted in through the boundary scan input register, one at a time, and applied to the on-chip system logic. The corresponding responses are captured in the IC's boundary scan output register and are examined by subsequent shifting.

(f) *Built-In self-test:* this is the second optional test mode. However, the Standard recommends its implementation wherever possible. The use of BIST mode allows a component user to determine the health of a component without the need to load complex data patterns. While the BIST mode is selected, the boundary scan register may act as a pattern generator and/or signature compactor. The resulting signature is shifted out and is checked to ensure proper circuit operation. When BIST is the current test mode, the test data register into which the results of the self-tests will be loaded, must be connected for the serial access between TDI and TDO (this register can be the boundary scan register).

When the circuit is not in one of the test modes, it performs its normal functions. The standard mainly addresses the problems of off-line testing, as it is evident from five of the six test modes above (except the sample test). The sample mode provides a degree of on-line monitoring (i.e. a snapshot of the circuit can be obtained in time), but does not support continuous on-line checking (i.e. concurrent checking).



(a) A board level view of boundary scan



(b) A chip level view of boundary scan

Figure 3.4: A view of boundary scan

3.5 Previous Work in Merging

Most DFT techniques involve either resynthesis of an existing design or the addition of extra hardware to the design. Most approaches require circuit modifications and affect such factors as area, performance and I/O pin count. One trade-off is to merge some test techniques to obtain the combined testability features. Meanwhile, it is also possible to reduce the overall cost through the merging. In this dissertation, we are interested in two types of merging: the merging of BIST and boundary scan techniques, and the merging of concurrent checking and BIST.

3.5.1 Merging BIST and Boundary Scan

Both BIST and boundary scan employ register based structures in implementation. Various architectures for merging the two techniques have been proposed [2, pages,483-501]. In general, these merging methods allow a CUT to be tested using BIST, and scan paths are used to obtain additional access to the internal circuitry. LFSRs are used as the implementation structures of the BIST circuitry. The *built-in evaluation and self-test* (BEST) [2] and the *circular self-test path* (CSTP) [27] are applicable to both combinational and sequential circuits. The *LLSD on-chip self-test* (LOCST) [13, 28], the *concurrent BIST* (CBIST) [49], *Centralized and embedded BIST* (CEBS) [2, pages 490-492], and *random test data* (RTD) [4] are for combinational CUTs only. In these approaches, some testing circuitry is separated from the CUTs and other circuitry is embedded in the design. Figure 3.5 shows an example of these approaches, the CEBS. The first r-bit section of the input boundary scan registers acts as a PRPG and single-output autonomous LFSR (SRPG), the last s-bits of the output boundary scan registers act as both a MISR and a single-input signature register (SISR). The testing proceeds as follows. A test mode signal is set and the scan registers are seeded. Then the PRPG loads the scan path with

pseudorandom test data. A system clock is issued and the scan-path registers are loaded in parallel with system data, except for the signature register, which operates in the MISR mode. The scan path is again loaded with pseudorandom data while the signature register operates in the SISR mode, compressing data that were in the scan path.

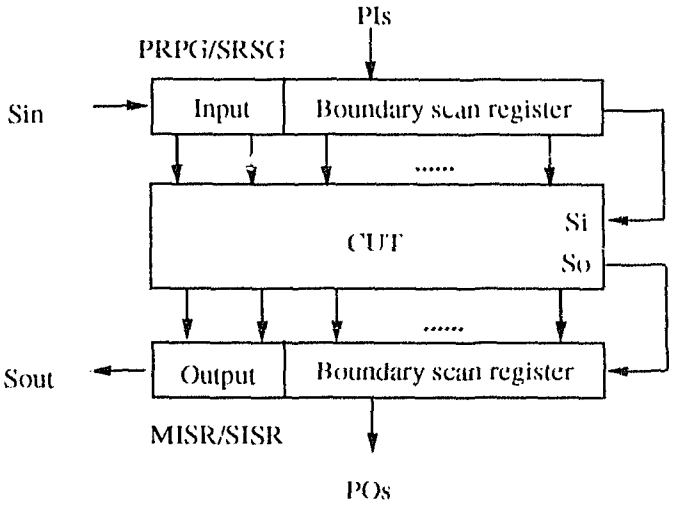


Figure 3.5: The CEBS architecture

Gloster and Brglez have demonstrated that LCARs are a feasible alternative to LFSRs in testing schemes with the combined BIST and boundary scan [18]. A detailed discussion of this scheme can be found in section 4.5. Moreover, IEEE Std. 1149.1 further promotes the merging of BIST and boundary scan techniques by defining BIST as one of the recommended test modes [1]. Merging BIST and boundary scan is an active area of applied research.

3.5.2 Merging Concurrent Checking and BIST

In some critical applications, concurrent checking and off-line BIST have different requirements. In a system with off-line BIST circuitry in place, one may not be

satisfied with the periodical off-line test provided, and concurrent monitoring may be desirable to improve the reliability and the fault detection latency of the system. The fault detection latency is the time delay to report a fault after a testing vector is applied. Conversely, in a system with concurrent checking, the application patterns of the CUT may not exercise all the internal elements, and the defects in the unexercised places may go undetected. To avoid having these defects appear at a critical computation time, the CUT may need to be exhaustively tested when it is off duty. Off-line testing, may, thus, still be required for better fault coverage. Hence, sharing of resources between the two test modes would seem a natural solution to these problems.

The idea of merging on-line and off-line BIST was first suggested by Sedmak in [51] as an alternative to dealing with testability problems in VLSI-based designs. There, the general design philosophy and some guidelines for possible implementation of the approach are presented. Specific techniques applicable to this approach are discussed in [52], including partitioning the logic, placement and design of the test circuitry. The checking circuitry recommended in that paper involves duplication of the functional circuitry and comparison of the outputs of the two implementations. This technique avoids the aliasing problem and consequent loss of effective fault coverage of the signature analyzer. Figure 3.6 shows a suggested BIST implementation using this scheme, where the duplicated circuitry is actually realized in complementary form to reduce design and common mode faults. The test patterns are shifted serially into the input registers from the input stimuli generator which is added for test pattern generation. The generator cycles through all input patterns thus generating exhaustive test patterns. The contents of the two input registers are checked by the compare 1 circuit. The output response of the combinational circuitry is checked by the compare 2 circuit.

Lu and McCluskey [30] describe an efficient method for explicit testing of a

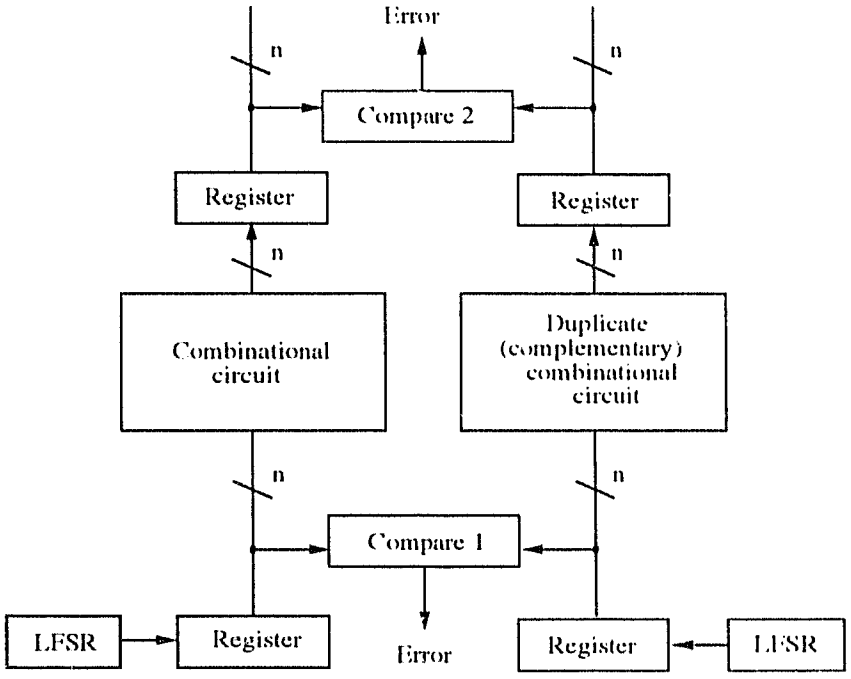


Figure 3.6: BIST using concurrent checking to compare circuit response

RALU (register arithmetic logic unit) that incorporates parity prediction for implicit testing.

A notable result in the merging is due to Saluja *et al.* [48], illustrated in Figure 3.7. In this scheme, the off-line testing resources are modified such that during system operation they can also observe the normal inputs and outputs of a combinational CUT. During off-line test mode, the multiplexor is set such that the tests generated by the Test Generator are applied to the CUT, and the responses are compressed using the Response Verifier. After the application of all tests, the contents of the Response Verifier are used to determine the status of the CUT. In the normal mode, the testing of the CUT proceeds concurrently with the normal operation of the system. The Response Verifier is enabled only if a vector from a particular set occurs. The Test Generator produces, at its outputs, vectors from the test set. During the normal operation, outputs of the Test Generator are con

stantly compared, bit-by-bit, with the normal inputs to the CUT by the equality comparator. A signal HIT is generated if, and only if, the normal inputs to the CUT are the same as an active test vector of the Test Generator. When a HIT is generated, the Response Verifier is enabled to compress the normal outputs of the circuit, and the Test Generator is advanced to the next state to produce the next vector. Unfortunately, the concurrent checking test latency can be 2^n, where n is the number of inputs of the circuit. A number of approaches were suggested to reduce the test latency by paying the price in silicon, time, and performance.

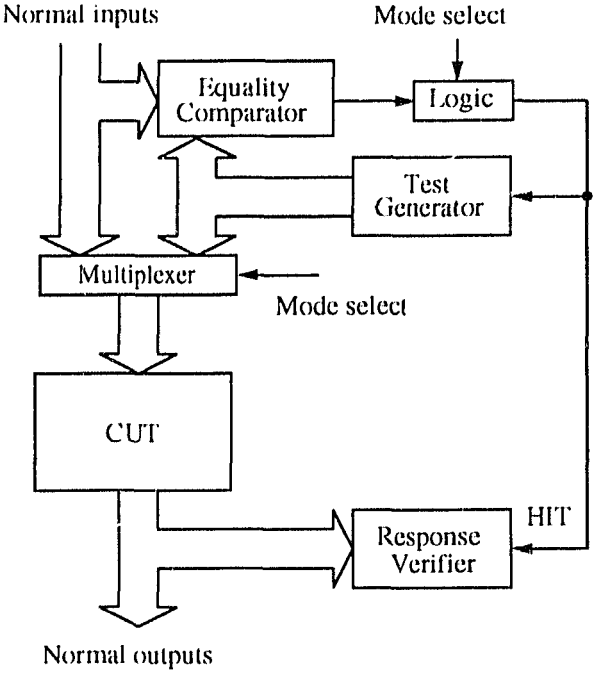


Figure 3.7: The concurrent comparative testing organization

Nicolaidis [37, 38] suggests a unified BIST scheme (UBIST), which integrates off-line test (design validation test, go/no go test of dies, test of boards,and maintenance test) and on-line error detection at system level, and provides high fault coverage for single faults. UBIST is based upon the concept of *self-exercising checkers*, shown in Figure 3.8. In addition to conventional double-rail checker, pass transistors are

used to connect the checker inputs and the test pattern generator, and a logic block (CNCI) is introduced to indicate whether a code word is generated (S = 1). Under fault free condition, f_0 and f_1 are double-rail encoded. The code will not retain if a fault presents.

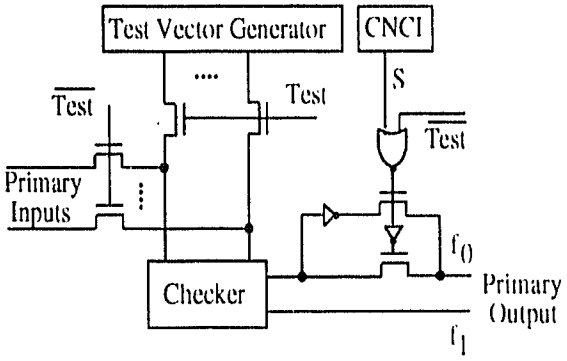


Figure 3.8: Self-exercising checker

For the UBIST method, all functional blocks are designed to be *strongly fault secure (SFS)* [36, 39, 58]. Figure 3.9 shows the unified BIST scheme. UBIST modules are composed of an LFSR part and a coding part. The LFSR part may be realized as the BILBO[25], and the coding part (CNCI block) is as proposed in the schemes of self-exercising checkers. The checker modules transpose code word inputs to correct indication and non-code word to error indication. During the TEST1 phase, test vector generated by odd UBILBOs are applied on odd functional blocks and on even checker modules. The responds of odd functional blocks are verified by odd checker modules, and are also compressed by even UBILBOs to give signatures. At the end of the TEST1 phase the signatures in even UBILBOs are shifted out for verification. Similarly, during TEST2 phase we test even functional blocks and odd checker modules. The double-rail checker is used for reduction of error indication signals and is tested during the TEST3 phase.

Recently, Gupta and Pradham [20] demonstrate that the information generated

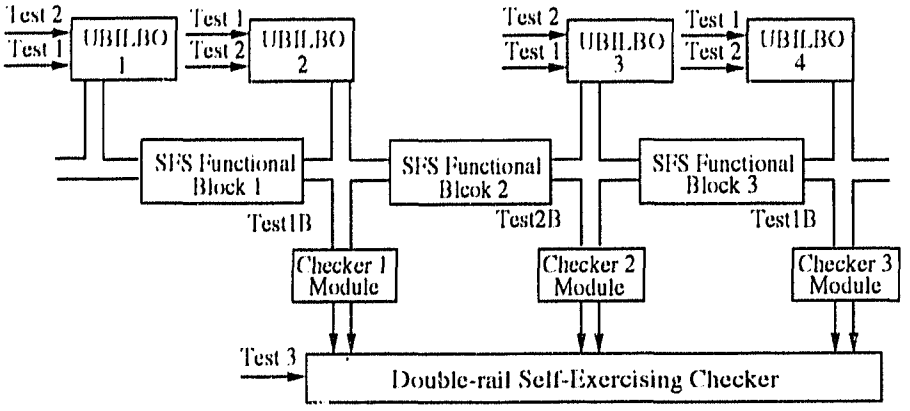


Figure 3.9: Unified BIST scheme

in concurrent checking (even parity checking) can be utilized to reduce off-line testing time and fault escape probability. The modification to the conventional signature analysis structure is that the output of a CUT is collected by both the signature analyzer and the parity checker. If any of the test responses is found erroneous by the checker, then the CUT is determined to be faulty immediately, with no further testing necessary. In the case, on completion of testing, the content of the signature analyzer is analyzed to determine whether it corresponds to the good signature.

It should also be noted that the *Sample/Normal* mode of the IEEE Std. 1149.1 permits a snapshot of normal system operation and allows the output response to be shifted out for examination, without suspending the computation. This test provides a degree of on-line monitoring and it is a step towards the merging.

3.6 Problem Statement

Although some design and test engineers may still be reluctant to incorporate BIST and/or boundary scan in their designs, an average of 30% or lower area overhead has been widely accepted as *reasonable*. The main concern of this dissertation is the merging of concurrent checking and off-line BIST, since, even for VLSI technology

available today, concurrent checking is still considered expensive.

Historically, on-line and off-line testability studies have been the concern of two separate technical communities. Consequently, two sets of test algorithms, which are rarely common in design and implementation, have been developed. On the one hand, off-line BIST using data compaction schemes usually requires (a) a mechanism for supplying test patterns to the CUT, and (b) a means for compacting the CUT responses into a signature. On the other hand, concurrent checking employs error detecting codes, such as parity, Berger, and residue codes, to encode a given circuit function at design stage. Modification of the original circuit function usually results in an augmented circuit realization that is bigger than the original one. It has been shown that for an augmented circuit, a 15%-200% hardware increase can be expected [54, 64]. The increase depends on the functionality and size of the CUT, and the coding scheme chosen. Moreover, since data patterns also serve as test patterns in concurrent checking, there is no reference to the fault free signature of the good circuit response. Thus, checking circuitry is required to be self checking in order to guarantee a high fault coverage. Designs of *totally self-checking (TSC)* and strongly code disjoint (SCD) checkers can be found in [2, pages 578 588] and [40], respectively. Specially designed checkers are required for distinct coding schemes. For example, a parity checker is a $\log\ n$-level binary tree of XOR gates, where n is the number of information bits being checked, while a residue checker is an arithmetic unit performing modulo operations.

The main technical obstacle of the merging is the use of different test methodologies in the two test modes. It leads to separation of hardware and introduces unacceptable silicon cost. As discussed above, concurrent checking and BIST usually employ distinct testing techniques. These techniques require different implementation structures, thus making the merging of on-line and off line techniques difficult. In this dissertation, we propose to use a common test technique for on line and

off-line test modes. To achieve this, it is necessary to find a link between the two types of testing methodologies, as well as their implementation mechanism. The objective is to support both on-line and off-line testability with maximum sharing of hardware resources and low impact on system performance.

Chapter 4

Concatenation and Partitioning of LFSRs and LCARs

At the circuit, board or system level it is desirable to have a cost effective implementation of bit-sliced registers, reconfigurable for different lengths as needed. LFSRs and LCARs are the most commonly used register devices. In chapter 2, we provided some basic background on LFSRs and LCARs, and their conventional applications as pseudo-random pattern generators and signature analyzers, in BIST. This chapter examines theoretical and practical issues of LFSR and LCAR concatenation and partitioning, giving reconfigurable register lengths. The results of this study add flexibility to digital circuit design and BIST in general, and is one of the bases of the new testing scheme proposed in Chapter 5.

Early work on the properties of autonomous LFSRs' concatenation can be found in Elspas [14]. Later, Bhavsar [5] introduced the principle of polynomial concatenation, and the concept of polynomial splitting (polynomial partitioning) and bit sliced design of LFSRs. Consequently, a family of general purpose bit sliced LFSR register chips is proposed to facilitate self-test. Sun and Serra [60] lay the foundation of the principle of LCAR concatenation and partitioning. Recently, Kontopidi [26]

established a more formal definition of polynomial partitioning, and explored the partitioning behaviour of LFSRs and LCARs from some new perspectives.

In this chapter, we first give some theoretical background and definitions, then summarize the work on polynomial concatenation and partitioning. Subsequently we introduce the corresponding issues for LCARs, discuss various applications of LFSR and LCAR concatenation and partitioning, and present some results of a comparative study of the two types of linear machines.

4.1 Background

In this section we give some theoretical background of linear finite state machines and their algebraic context, and show their different representations.

4.1.1 Linear Finite State Machines and Isomorphism

A basic result in linear algebra is that a linear transformation in a vector space can be represented by a matrix and that, in turn, such a transition matrix represents a linear transformation.

Definition 4.1 [59, page 297] A function f: $V \rightarrow V'$ is a *linear* function from a vector space V into a vector space V' over the same scalar field K as V if, for all c_1 and c_2 in K and all v_1 and v_2 in V

$$f(c_1 v_1 + c_2 v_2) = c_1 f(v_1) + c_2 f(v_2).$$

For our purposes, the field is a binary field and the + operation is Modulo-2 (XOR) addition. The linear transformation describes the behaviour of a corresponding linear finite state machine, whose definition follows.

Definition 4.2 [59, page 297] A machine M is a *linear finite state machine* if

(1) the state space S_M of M, the input space I_M, and the output space Y_M are each vector spaces over the appropriate finite field (here a binary field), and

(2) let the vector s_i denote the state of the machine, the vector u_i denote the inputs to the machine and the vector y_i denote the outputs of the machine. The next state s_i^+ of M is defined by

$$s_i^+ = Rs_i + Pu_i$$

and the output is defined by

$$y_i = Ts_i + Qu_i,$$

where R, P, T, and Q are transformation matrices of the appropriate size over the finite field. In the case of an autonomous machine (with no external input u_i), the second term is omitted from each equation.

For our purposes, we can summarize by the fact that, in the operation of a linear finite state machine, the next state is always computed from the previous state using only linear rules, i.e. rules which translate into XOR implementations.

Definition 4.3 [59, page 306] If two matrices A and B are square matrices for which there exists an invertible matrix P such that $B = P^{-1}AP$, then A is said to be *similar* to B.

Theorem 1 [29, page 155] Two matrices A and B represent the same linear operator T if and only if they are similar to each other.

Definition 4.4 [59, page 310] The *characteristic polynomial* of a square matrix A is defined as $\Delta_A(\lambda) = det(\lambda I - A)$.

Definition 4.5 [44, page 148] A polynomial $p(X)$ of degree n which is not divisible by any polynomial of degree k, where $0 < k < n$, is called *irreducible*.

Definition 4.6 [44, page 161] An irreducible polynomial of degree n over a binary field is *primitive* and if, and only if, it divides $X^m - 1$ for no m less than $2^k - 1$.

The important consequence of the primitivity of a characteristic polynomial is that the linear finite state machine in its autonomous operation traverses all possible $2^n - 1$ non-zero states before returning to its initial configuration.

A LCAR is a finite state machine, which can be represented by a transition matrix constructed from the next state equations of the computational rules used. Similarly, a transition matrix can be found for an LFSR. For each of these matrices, a characteristic polynomial can be computed. The relationship between LFSRs and LCARs is stated as follows:

Theorem 2 [56] A one-dimensional linear LCAR and an LFSR with the same irreducible (or primitive) characteristic polynomial are isomorphic, and the corresponding transition matrices are similar.

The consequence is that a LCAR and a LFSR, which are based on the same irreducible or primitive polynomial, have the same behaviour as linear finite state machines up to permutation of the order in which the states appear. However, *the cycle structure* of the states [59, page 307] is identical, that is, the states may be permuted in order, but they must appear in the same cycle configuration.

Example 1 Figure 4.1 shows a LCAR and LFSR with their corresponding transition matrix and their characteristic polynomial. The cycle structure is also shown in the state transition diagrams. Since this polynomial is irreducible, but not primitive, the states form four separate cycles, where state 0 always goes back to itself.

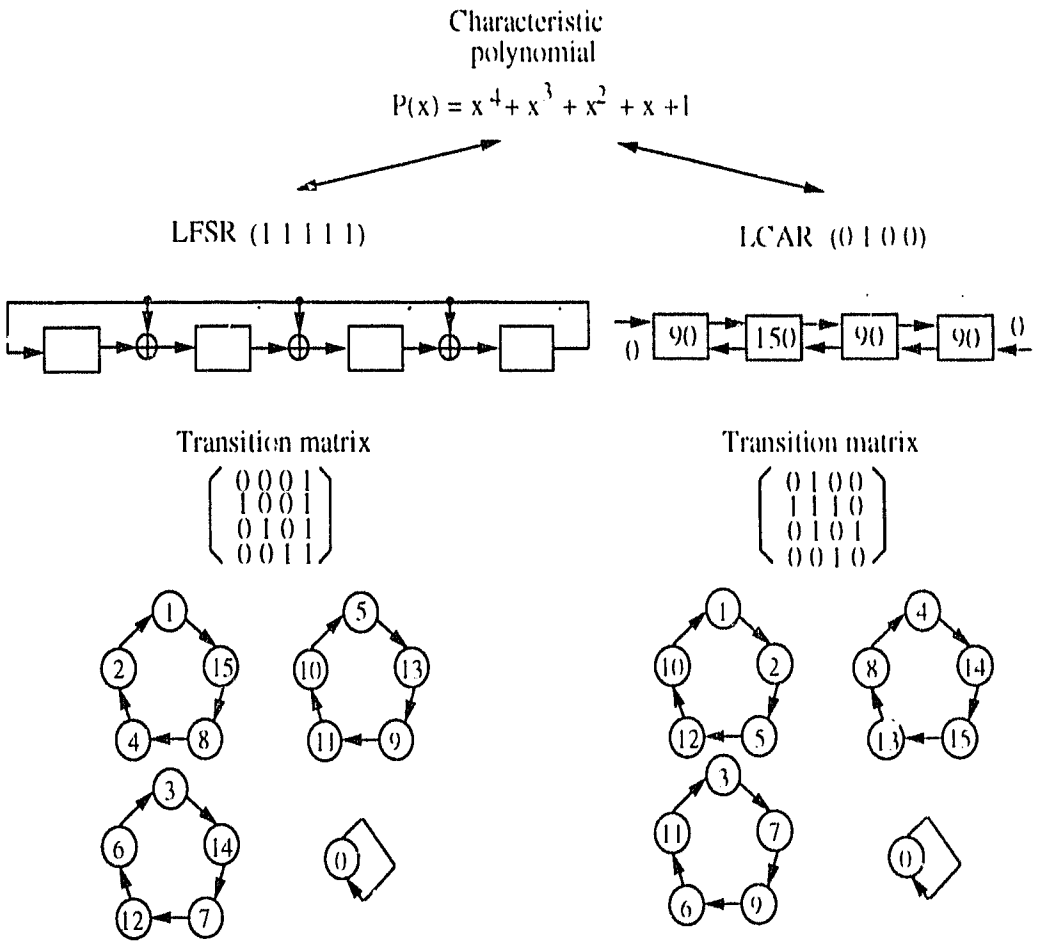


Figure 4.1: Transition matrices of a LFSR and a LCAR

4.1.2 Different Representations

There are three different representations which are used interchangeably in polynomials and their LFSR implementations: polynomials in a binary field, binary string representations, and the LFSR implementation of polynomials. Each representation provides a convenient expression in a corresponding domain, and can be easily transformed to either of the other two. A polynomial $P(x)$ can be directly mapped into a LFSR implementation, where the zero and non-zero coefficients correspond to feedback taps of the LFSR; and it can also be mapped to a binary string, where the

non-zero and zero coefficients correspond to 1's and 0's, respectively. The reverse transformations hold as well. In the previous example, these transformations were shown.

The situation is slightly more complex for LCARs. Given a characteristic polynomial, three algorithms have been developed and implemented to find its corresponding LCAR:

(1) the first algorithm in [56] is based on a pruned search;

(2) a better algorithm, in [55], uses the Lanczos tridiagonalization method but requires exponential time;

(3) the best known algorithm reported recently [8] applies Euclid's greatest common divisor algorithm to compute the LCAR. It has a polynomial running time, which is sufficiently fast to generate LCARs for polynomials of very large degree.

Conversely, given a LCAR, it is easy to calculate the characteristic polynomial of its transition matrix [56]. The mapping between a LCAR implementation and its binary form is also simple: rule-90 and rule-150 cells correspond to 0's and 1's respectively and form the pattern of the main diagonal of the transition matrix. This is also shown in Figure 4.1. Figure 4.2 summarizes the transformations above, and visualizes the relations among the different representations.

4.2 Polynomial Concatenation and Partitioning

Aiming at building off-the-shelf chips with different sizes of LFSR for board and system level self-test, Bhavsar [5] considered a subset of polynomials, which preserve certain properties such as primitivity, and certain types of concatenations which benefit BIST. One of the main interests in polynomials concerns *primitive* polynomials

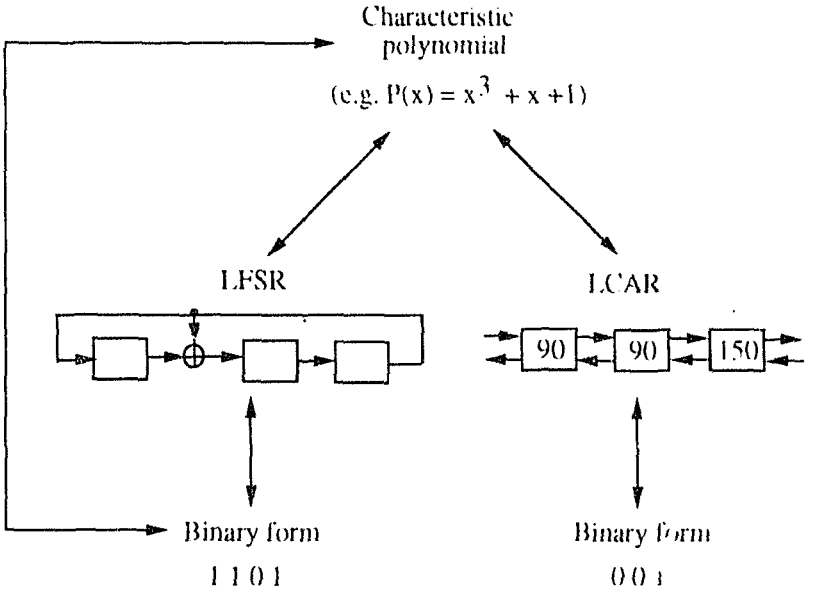


Figure 4.2: Transformations of LFSR and LCAR

[4]. The important property is that linear finite state machines based on primitive polynomials cycle through a maximum length sequence when in autonomous mode (i.e. without external inputs). This is particularly useful when they are used as pseudorandom pattern generators, as they can stimulate a circuit exhaustively. With regards to concatenation, the interest lies in finding structures which can be concatenated and still maintain the attribute of maximum length cycles, that is, finding primitive polynomials which remain primitive after concatenation.

Since a partition is the opposite operation of a concatenation, they are the same in terms of implementation. Kontopidi [26] explored statistically the general partitioning behaviour of LFSRs, applicable to polynomials. The whole space of irreducible and primitive LFSRs from length 2 to 16 are examined in [26]. In the following subsections, we first give the basic problem statements for concatenation and partitioning, then present the results of the two issues separately.

4.2.1 Definitions

The definitions for polynomial concatenation and partitioning are collected here. Most of them are reworded from [5, 26] and [60].

Definition 4.7 [5] Let $A(X)$ and $B(X)$ be two polynomials of degree r and s respectively. Then A and B are *concatenated* to realize two distinct polynomials, C'_{AB} and C'_{BA}, of degree $r+s$ as follows:

$$\begin{aligned} C'_{AB} &= X^s(A+1)+B \\ C'_{BA} &= X^r(B+1)+A. \end{aligned}$$

The two types of concatenation are referred to as AB and BA concatenations respectively.

Definition 4.8 [5] If the polynomial C' formed by either of the above operations is primitive, then the operation is called *primitive concatenation.*

Definition 4.9 [5] The concatenation of a polynomial of degree s with itself n times ($n > 1$) to form a polynomial of degree $n \times s$ is called *self-concatenation.* It is called *non-self-concatenation* if the polynomial concatenates with one or more other polynomial(s).

Definition 4.10 [26] A polynomial C' formed by the concatenation of two polynomials, A and B, can be also *partitioned* into two parts, A and B.

Definition 4.11 [26] The partition of a polynomial C, of degree t, into two irreducible polynomials A and B, of length r and s respectively, such that $r+s=t$, is called an *irreducible partitioning.* Otherwise, it is called a *reducible partitioning.*

Definition 4.12 [26] The partition of a polynomial C, of degree t, into two primitive polynomials A and B, of degree r and s respectively, such that $r+s=t$, is called a *primitive partitioning.* Otherwise, it is called a *nonprimitive partitioning.*

It should be noted that primitive concatenation respects primitivity of the newly formed polynomial, while primitive partitioning insists both polynomials obtained from the partitioning are primitive. The logic behind these seemingly inconsistent definitions is that in both cases, concatenation and partitioning, the attribute of primitivity is only related to the result of the operations (primitivity of the resulting C in concatenation, and the resulting A and B in partitioning), while the inputs (the polynomials A and B before concatenation, and polynomial C before partitioning) are not necessarily primitive.

Definition 4.13 [26] If a finite state machine of length l is partitioned into two bit-slices of length r and s respectively, such that $r + s = l$, $r = 1$ or $s = 1$, then it is called a *degenerate partitioning*.

Definition 4.14 [26] If a finite state machine of length l is partitioned into two bit-slices of length r and s respectively, such that $r + s = l$, $1 < r, s < l - 1$, then it is called a *proper partitioning*.

Example 2 The degenerate partitions of length 6 finite state machines are:

□ ‖ □ □ □ □ □

□ □ □ □ □ ‖ □

where "‖" indicates the partition point. The proper partitionings are:

□ □ ‖ □ □ □ □

□ □ □ ‖ □ □ □

□ □ □ □ ‖ □ □.

Hence, a finite state machine of length n has $(n-1)$ possible partitionings, two degenerate partitionings, and $(n-3)$ proper partitionings.

4.2.2 Concatenation of Polynomials and LFSRs

Table 4.1 shows *the primitive self-concatenations* of primitive polynomials of degree 3 to degree 8, forming polynomials up to degree 64. The second column of this table lists all the primitive polynomials of these degrees, which are used as the initial polynomial for concatenation. The third column represents the number of primitive self-concatenations of the corresponding initial polynomials to construct primitive polynomials. Both the initial and the resulting polynomials are primitive. As an example, the degree 8 primitive polynomial represented by 101100101 self-concatenates in groups of 2, 4 and 7 to form degree 16, degree 32 and degree 56 primitive polynomials respectively.

The original table of degree 3 to 10 primitive polynomial self-concatenation can be found in [5]. Table 4.1 is with our modifications (in italics). It should be noted that Table 4.1 does not include reciprocal polynomials[1]of the polynomials. It is known that reciprocal polynomials of primitive polynomials are also primitive [44, page 163], and this property is preserved in primitive self-concatenation. For example, the degree 3 polynomial, $x^3 + x + 1 = 1011$, self-concatenates 2, 3 and 4 times forming degree 6, 9 and 27 primitive polynomial respectively, while its reciprocal, $x^3 + x^2 + 1 = 1101$, can also construct degree 6, 9 and 27 primitive polynomials by self-concatenation.

Primitive self-concatenation is interesting as it allows reconfiguration of *identical* short registers into much longer ones, which preserve primitivity.

In [5], the principle of general concatenation of LFSRs is also addressed in terms of testing applications. In general, concatenation of two polynomials of degree s and r results in a polynomial of degree $s + r$. The initial polynomials and the resulting one may or may not be primitive.

[1]The reciprocal polynomial is defined as $P(1/x)$ and its binary representation is the reverse image of the original $P(x)$.

Degree	Initial Polynomial	Number of Primitive Self Concatenations of Initial Poly.
3	1011	2, 3, 9
4	none	none
5	100101	3, 12
	110111	5
6	1000011	3, 5
	1100111	2
7	10001001	9
	10001111	3, 9
	10011101	3, 9
	11001011	3, 9
	11010101	3
	11100101	2, 9
8	100011101	6
	101100101	2, 4, 7

Table 4.1: Self-concatenation of degree 3 to 8 polynomials.

4.2.3 Partitioning of Polynomials and LFSRs

Bhavsar first introduced the concept of polynomial *splitting*, and presented examples. In [26], polynomial partitioning is formally defined and the concepts are expanded. We summarize here the results of LFSR partitioning from [26].

The partitioning behaviour of LFSRs, for all degree 2 to 16 primitive and irreducible polynomials, are examined by means of simulation. The exhaustive search is applied to primitive and irreducible polynomials with primitive partitioning and/or irreducible partitioning respectively. It implies that both the original and the resulting machines are also primitive and irreducible. Moreover, the study is restricted to proper partitionings, since degenerate partitionings result in a machine of length one, which seems to be of no direct use [26].

The partitioning behaviour of primitive and irreducible polynomials of *degree* n is measured using two parameters:

(1) *ATLOP:* the number of degree n polynomials which have AT Least One Partitioning with the desired property.

(2) *PEPP:* the PErcentage of Partitionings which have the desired property.

For example, for primitive polynomials of degree n, a partition with the desired property is a primitive partitioning, and hence PEPP is defined as

$$\frac{P_{prim} \times 100}{(n-3) \times prim(n)}$$

where P_{prim} is the total number of (proper) primitive partitionings, and $prim(n)$ is the total number of primitive polynomials of degree n. The factor $(n-3)$ is used to subtract degenerate cases.

Table 4.2 shows the partitioning behaviour of polynomials implementing primitive polynomials of degree 4 to 16; the corresponding table for irreducible polynomials can be found in [26]. Note that there are no length four LFSRs that have primitive partitionings. As a result the values of *PEPP* and *ATLOP* are both zero.

4.3 LCAR Concatenation and Partitioning

Recently, Linear Celular Automata Registers (LCARs) have been proposed as an alternative to LFSRs for pseudorandom data pattern generation and data compaction [21]. This section introduces the principle of LCAR concatenation which is the original work of the author, and also presents the behaviour of LCAR partitioning from [26].

4.3.1 Definitions

A linear finite state machine is said to have a *maximal length cycle* [56] if all possible states lie on a single connected cycle in the state transition graph (except for the all 0's state) in its autonomous behaviour (no external inputs). The characteristic

Degree	PEPP	Number of Primitive Polynomials	ATLOP
4	0	2	0
5	33.33	6	1
6	22.22	6	1
7	16.66	18	10
8	12.50	16	10
9	9.72	48	22
10	8.57	60	26
11	6.81	176	80
12	6.32	144	70
13	5.42	630	276
14	4.93	756	336
15	3.94	1800	704
16	4.15	2048	904

Table 4.2: The partitioning behaviour of primitive LFSRs

polynomial is primitive in this case. In this research we denote by *primitive LCAR* as one which has a maximal length cycle.

LCAR concatenation can be examined from three different perspectives: the method of concatenation, the attributes of the resulting LCARs, and the attributes of the participating LCARs.

4.3.1.1 Method of Concatenation

Definition 4.15 Let A and B be two LCARs of length r and s, respectively. Then a LCAR C of length $r+s$ can be obtained by equations as:

$$C_{AB} = A \ll s + B$$

$$C_{BA} = B \ll r + A$$

where "$A \ll s$" denotes the shifting of s-bits of A to the left and "$+$" denotes the logic OR operation. The two equations are referred to as *AB concatenation* and *BA*

concatenation, respectively.

Definition 4.16 The concatenation of a LCAR of length s with itself n times ($n > 1$) to form a LCAR of degree $n \times s$ is called *self-concatenation*. It is termed *non-self-concatenation* if the LCAR concatenates with one or more other LCAR(s).

4.3.1.2 Attributes of Resulting LCARs

Definition 4.17 If a LCAR obtained from either of the concatenations in the equations above is primitive, then the concatenation is called a *primitive concatenation*. Otherwise, it is called a *non-primitive concatenation*.

4.3.1.3 Attributes of Participating LCARs

There are four possible cases for the attributes of the participating LCARs which are considered:

(1) Primitive-Primitive LCAR concatenation (PP),

(2) Primitive-Nonprimitive LCAR concatenation (PN),

(3) Nonprimitive-Primitive LCAR concatenation (NP), and

(4) Nonprimitive-Nonprimitive LCAR concatenation (NN).

The concatenation operations given above can be used repeatedly to combine any number of LCARs. Obviously, there are many choices. In the case of self-concatenation, the two LCARs obtained from AB and BA concatenations are identical. Figure 4.3 shows the taxonomy of LCAR concatenation. We assume that the root is level 1, then levels 2, 3 and 4 in turn represent the three perspectives of the classification: method of concatenation, attributes of resulting machine, and attributes of participating machines, respectively. Note that this classification is applicable to LFSRs as well.

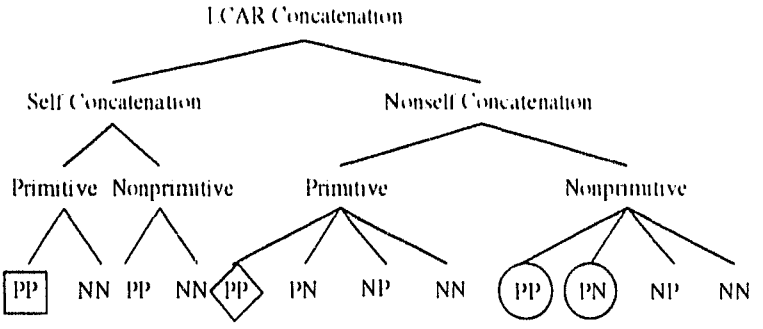


Figure 4.3: The taxonomy of LCAR concatenation

Similarly, we define LCAR partitioning as follows:

Definition 4.18 If a LCAR, C, is constructed by concatenation of two LCARs, A and B, then C can also be said to be partitionable into two LCARs, A and B.

Definition 4.19 A *primitive partitioning* occurs when a LCAR, C, of length t can be partitioned into two LCARs, A and B of length r and s, such that $r + s = t$ and both A and B are primitive.

The definitions of *degenerate* and *proper* partitioning for polynomials are also applicable to LCARs. Hence, a LCAR of length n has $(n - 1)$ possible partitionings, two degenerate partitionings and $(n - 3)$ proper partitionings.

4.3.2 LCAR Concatenation

There are two distinct approaches to studying the concatenation properties of linear finite state machines: find the mathematical basis of LCAR concatenation or use a simulation method. An attempt was made to find the mathematical basis of the concatenation. The concatenation of polynomials, both for LFSRs and LCARs, implies a composition of their respective transition matrices. The primitivity and irreducibility of the characteristic polynomial of the resulting matrix can not be

deduced by the attributes of the characteristic polynomial of the submatrices. Thus, it is investigated by simulation.

It was shown in the previous section that there are many possible concatenations. Our study in LCAR concatenation emphasizes the self-concatenation of primitive LCARs due to the following reasons:

(1) a LCAR-based pattern generator with a primitive characteristic polynomial produces the maximum pseudo-random set of input stimuli, which is important because maximal runs are needed for testing;

(2) Williams [65] showed that primitive LFSRs, and thus LCARs, are more suitable as data compactors under certain assumptions;

(3) self-concatenation of LCARs reflects the modularity and regularity natures of VLSI design and implementation, and provides flexibility for dynamic reconfiguration, while primitive self-concatenation ensures that machines with maximal length cycle are available in different lengths.

Tables of LCAR concatenation are presented in Appendix 1. The tables are summarized below and laid out to correspond to the concatenation hierarchy shown in Figure 4.3.

(1) Self primitive concatenation.

Tables under this category are produced for self primitive concatenation only and are shown in Section 1 of Appendix 1.

One table is shown for each length from 2 to 16. We list all primitive LCARs forming longer primitive LCARs by self-concatenation up to length 64. The numbers in the brackets are the number of self-concatenations to form a primitive LCAR from a given LCAR. As an example, in the table of length 4, the primitive LCAR 1010 self-concatenates

3, 7, 13 and 15 times to form primitive LCARs of length 12, 28, 52 and 60 respectively. An interesting observation here is that the length 8 primitive LCARs, 11101110 and 11110111, self-concatenate twice and four times respectively to give length 16 and 32 primitives. 16 and 32 are the word lengths of most computer systems. This type of concatenation corresponds to the one in the square bounding box of Figure 4.3 on page 49.

(2) Non-self primitive concatenation.

Due to the very large number of possible concatenations of each length, we only include some examples here.

In Section 2 of Appendix 1, we list all length 5 primitive LCARs in the first row of the table. For each of the LCARs, we concatenate the remaining LCARs to it and check the primitivity. The length 10 primitive LCARs on the second row are obtained from the non-self primitive concatenation of the LCARs in row 1. Similarly, the primitive LCARs of rows 3 are formed by non-self primitive concatenation of the length 5 LCARs in row 1 and the 10 LCARs in row 2. The diamond box in Figure 4.3, page 49, represents the concatenations in this category.

(3) Non-self primitive/non-primitive concatenation.

The table in Section 3 of Appendix 1 lists a set of LCARs of length 16 constructed by both PP and PN concatenation from length 8 LCARs. The newly formed LCARs with * are primitive. The circles in Figure 4.3 denotes concatenation of this kind.

Degree	PEPP	Number of Primitive LCARs	ATLOP
4	50.00	2	1
5	16.66	6	2
6	27.77	6	4
7	18.05	18	11
8	12.50	16	8
9	12.15	48	27
10	11.42	60	40
11	8.59	176	91
12	7.40	144	78
13	6.38	630	311
14	5.73	756	378
15	5.53	1800	910
16	4.81	2048	1002

Table 4.3: The partitioning behaviour of primitive LCARs

4.3.3 LCAR Partitioning

Using the same parameters, *ATLOP* and *PEPP* defined in section 1.2.3 of page 45, the primitive partitioning behaviour of all LCARs implementing degree 4 to 16 primitive polynomials is given in Table 4.3. *PEPP* reaches the highest values, 50%, for $n = 4$. The percentage of primitive partitionings decreases gradually until it reaches the lowest value, 4.81, at $n = 16$. The corresponding table for irreducible LCARs can be found in [26].

The partitioning behaviour of LCARs presented above is obtained experimentally. However, as the length of the linear machines becomes greater than 16, the simulation process reaches the memory limit of our computers. A probabilistic treatment of a LCAR corresponding to randomly chosen irreducible and primitive polynomials, is presented in [26] and provides a method to estimate the partitioning behaviour of longer LCARs. A comparative study of partitioning behaviour of LFSR

and LCARs and their application issues is given in sections 1.4 and 1.5 respectively.

4.3.4 Summary of Look-up Tables

Tables providing a fast means for finding a certain type of polynomial, or a LCAR with a given characteristic polynomial, have been produced. Of particular interest are both LCARs and LFSRs (or polynomials) with the minimal number of non zero terms, thus leading to a more cost effective implementation. These are usually denoted as *minimum weight*. They are summarized as follows:

(1) Peterson [44] contains a complete table of all irreducible polynomials (including both primitive and non-primitive ones) of degree 2 to 16.

(2) In [4], at least one primitive polynomial with the fewest number of nonzero terms of each degree up to degree 300 is listed.

(3) Tables of isomorphic LCARs of all the irreducible polynomials from [44], and a large portion of the primitive minimal weight polynomials from [4], can be found in [57].

(4) Isomorphic LCARs of all primitive minimal weight polynomials from [4] are listed in [9]. Note that the isomorphic LCARs of the primitive minimal weight polynomials are usually not the primitive minimal weight LCARs of each length.

(5) The primitive minimal weight LCARs from degree 2 to 16 and degree 1 to 150, one for each length, are produced and reported in [61] and [70], respectively.

(6) Self primitive concatenation of degree 3 to 10 polynomials from [14] are given in [5].

(7) Self primitive concatenation of length 2 to 16 LCARs from [57] to form LCARs up to length 64 are presented in [61].

(8) Examples of non-self primitive/non-primitive concatenation of LCARs can be found in [61].

4.4 Comparison of LFSRs and LCARs for Concatenation and Partitioning

In the previous sections of this chapter, we presented the results of LFSR and LCAR concatenation and partitioning. In this section, we compare the two types of linear machines in the context of concatenation and partitioning.

4.4.1 Choice of Primitive Concatenation and Partitioning

There are many primitive polynomials for each degree and there are many possible self concatenations of the primitive polynomials. However, from degree 2 to 8, there are only 13 primitive polynomials that have primitive self-concatenations to form primitive LCARs up to degree 64. Table 4.1 of page 44 lists the 13 initial primitive polynomials and their 23 possible primitive self-concatenations. Adding the corresponding reciprocal polynomials, we double the two numbers above, and have 26 primitive polynomials with primitive self-concatenations and 46 primitive self concatenations from the 26 initial ones.

On the other hand, in Section 1 of Appendix 1, there exist 58 initial primitive LCARs with primitive self-concatenation, from length 2 to 8, and 110 possible primitive self-concatenations from the initial LCARs. The mirror image (corresponding to the reciprocal of a polynomial) of a LCAR is considered as a distinct LCAR. In spite of the fact that there are equal number of initial primitive polynomials and

primitive LCARs at any degree, LCARs provide a richer set of self concatenable machines that maintain primitivity. It is evident that the number of primitive machines with primitive self-concatenation is 52% more than the number of polynomials from degree 2 to 8, thus, LFSRs, and the number of primitive self concatenations from the initial machines is 39% more than that for polynomials.

An interesting observation is that there is no primitive self concatenation and proper primitive partitioning for degree 4 primitive polynomials, while LCARs implementing degree 4 primitive polynomials have many choices in concatenation and partitioning.

In the case of partitioning, the difference between the behaviour of the two types of machines is not substantial. LCARs appear to have slightly better performance in primitive and irreducible partitionings than do LFSRs. However, the number of primitive and irreducible partitionings is quite small in comparison with the number of primitive and irreducible polynomials in each degree. For example, for LCARs corresponding to degree 16 polynomials, the total number of proper primitive partitionings over the total number of proper partitionings, i.e. $PEPP$, is only 4.81%. This led to an investigation to improve the partitioning behaviour. It is demonstrated that a significant improvement of the partitioning behaviour can be obtained by allowing *minimum modification*, i.e. introduction or elimination of a non-zero term in a LFSR's characteristic polynomial, or one reconfiguration of a rule 90 cell to a rule 150 cell or vice versa, to form a primitive LFSR and LCAR with both primitive and irreducible partitionings. The hardware cost of this modification is very small [26].

To conclude, in both concatenation and partitioning, LCARs provide more choices in the selection of primitive machines and consequently provide more flexibility in applications.

4.4.2 Aliasing and Pseudorandomness

It has been shown [56] that isomorphic LFSRs and LCARs exhibit the same aliasing behaviour when they are used as single-input data compactors. However, the aliasing behaviour differs in the case of multiple-input data compactors [35].

In [21, 34], it has been shown that LCARs are slightly better than LFSRs as pseudorandom test pattern generators. The result is in better fault coverage for certain types of faults by LCARs. It is also reported in [17] that 99.7% of transition faults can be covered when a LCAR is used as the pattern source, as compared with 93.4% transition fault coverage in LFSR implementation.

4.4.3 Structure of Concatenation and Partitioning

The comparison of LCAR and LFSR concatenation (partitioning) in terms of their structures can be made at two levels of description: logic and circuit levels.

4.4.3.1 Logic Level

Since communication in a LCAR implementation is restricted to nearest neighbours, two LCARs can be concatenated by simply linking the output of the one to the input of the other. Figure 4.4 shows how two 2-bit LCARs are concatenated to form a LCAR of length 4. All LCARs involved are primitive. Obviously, LCAR concatenation does not introduce any delays.

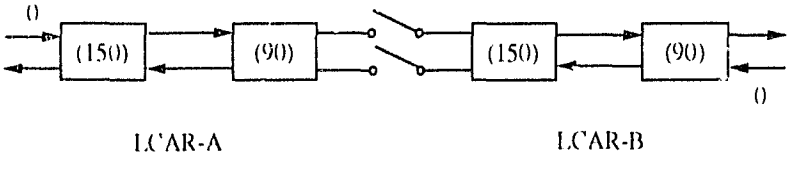


Figure 4.4: LCAR concatenation

Conversely, global broadcasting forms an essential part of LFSRs. The feedback

connection of a LFSR runs across its length and connects at least the output of the last cell to the input of the first. There are two ways to concatenate LFSRs:

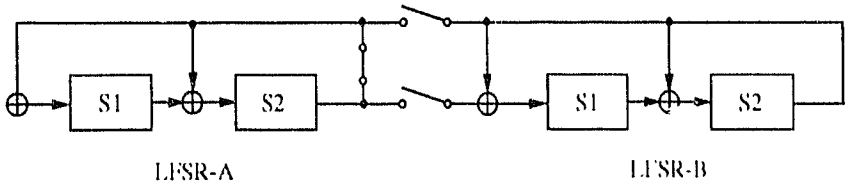


Figure 4.5: LFSR concatenation

(1) to break the feedback loops of each participating LFSRs except the right most one, and connect the last cell of the concatenated LFSR to the appropriate non-zero coefficients of the newly constructed LFSR, as shown in Figure 4.5.

(2) to make side by side direct connection of two LFSRs without the reconfiguration of each feedback tap in the feedback loop, as done in Elspas [14].

The first approach can maintain primitivity of the resulting machine if participating LFSRs have a primitive concatenation. However, breaking the feedback loops is not a very easy task in practice. The second method is easy in terms of reconfiguration, but it always leads to a longer LFSR implementing a reducible polynomial and, thus, without a maximal length cycle [14]. Hence, direct connections, maintaining primitivity of linear machines, are much simpler for LCARs.

4.4.3.2 Circuit Level

Figures 4.6 and 4.7 [26] demonstrate examples of circuit level implementation of LFSR and LCAR with concatenation (partitioning). When signal *PART* is high,

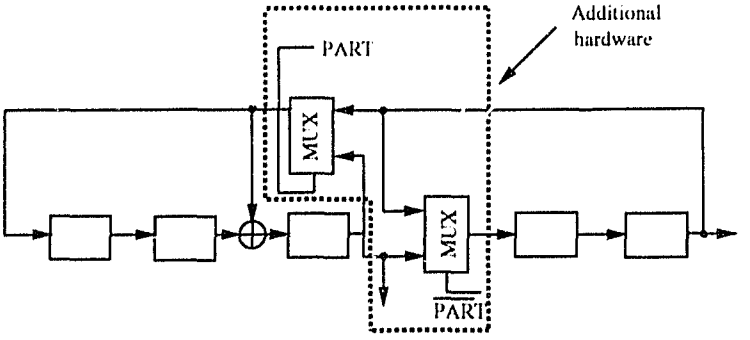


Figure 4.6: Dynamic reconfiguration of a LFSR

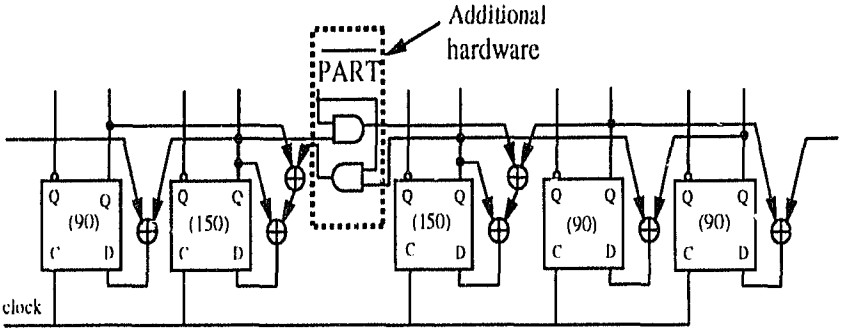


Figure 4.7: Dynamic reconfiguration of a LCAR

the two LFSRs or LCARs are disconnected and work independently; when *PART* is low, i.e. for concatenation, the 4-bit machines are constructed.

In practice, the concatenation of LFSRs is even more complex. For example, if we concatenate two LFSRs of 16 bits each, the driving capacity of the last stage cell in the newly constructed LFSR could be crucial to the system performance. Solutions to this problem can be either to design a special driver for it, or to allow the system to run at lower speed. LCARs do not share this problem.

degree	Min-cost Prim. Poly.	Min-cost Prim. LCAR
2	111	10
3	1011	001
4	10011	1010
5	100011	10000
6	1000011	100000
7	10000011	0010000
8	101100011	01100000
9	1000010001	100000000
10	10000001001	0100001000
11	100000000101	000010000000
12	10000010001101	001000100000
13	100000000011011	000010000000
14	101100000000011	100000000000000
15	1000000000003011	000000100000000
16	10000000000001101	000010100000000

Table 4.4: Minimum cost LFSRs and LCARs

4.4.4 Cost of Bit-sliced Implementation

Bit-sliced implementation of LCARs in general requires more Exclusive OR gates than LFSRs, since they have to be placed between every two stages. Some saving in area can be achieved by careful choice of LCARs. Primitive LCARs with a minimum number of "1" coefficients, called *minimum cost/weight* LCARs, can be used to reduce the cost since rule 150 (corresponding to a "1") is more expensive in implementation. Table 4.4 lists a selection of primitive polynomials and primitive LCARs with the minimum number of "1" coefficients from degree 2 to 16, where column 2 is reprinted from [4] and column 3 is produced by the author [60] and [70].

The following observations can be made from Table 4.4:

(1) most primitive minimum cost polynomials are trinomials (having three terms), and there are no primitive trinomials of degree 8 or degree $8n$ for any n [19];

(2) the primitive LCARs all have fewer than three non-zero terms, and most of them have only one non-zero term.

It has been reported [26] that length 16 LCARs are at least 40% more expensive than LFSRs of the same length, using actual layouts of the machines as implemented by OASIS[2]. The same margin of hardware increase is also reported on CALBO (cellular automata logic block observation) and BILBO (built-in logic block observation) designs [22]. The cost of the BIST resources can be reduced by using low cost XOR [43], and careful design layout. Overall, the hardware cost of LFSR and LCAR is still very low, compared with other BIST circuitry in implementation. One has to evaluate tradeoffs of better performance (as PRPG and delay fault detection) and area overhead. If used in the context of boundary scan, the difference in area between LFSRs and LCARs may not significant since the boundary scan cells are placed in the unused area next to I/O pads.

4.5 Applications of LFSR and LCAR Concatenation and Partitioning

Given the principle of LCAR concatenation and its advantages over an LFSR implementation, we outline some of its direct applications to DFT and VLSI testing.

(1) The principle of LCAR concatenation allows efficient use of BIST resources, such as PRPG, signature analyzer and CALBO at function, chip, printed circuit board or system level. It includes dynamic reconfiguration of the devices into different lengths or for different functionalities.

[2]Open Architecture Silicon Implementation Software, by the Microelectronic Center of North Carolina.

(2) In a boundary scan environment, a chip may consist of several function blocks with different numbers of inputs and outputs. LCAR concatenation can be used to reconfigure the boundary scan cells into BIST circuitry to facilitate on-chip test of the functions.

(3) The principle of LCAR concatenation can also be used to design special-purpose LCAR-based chips, which have multiple operation modes such as pattern generation and data compaction, and can be configured into standard off-the-shelf circuitry of variable sizes for the board and system designers and test engineers.

(4) Gloster and Brglez [17] proposed a new way to merge boundary scan with Built-In Self-Test of printed circuit boards (PCB). This approach makes use of the hardware resource for boundary scan as part of BIST circuitry, such that the total cost in boundary scan and BIST is reduced. The boundary scan registers used are based on LCAR implementation, and have similar functions as the registers of BILBO [25], with less area overhead. In the BIST mode, the boundary scan register, instead of being idle, serves as a source of a n-bit pseudorandom pattern generator. When an $(n+m)$-bit test pattern is needed for testing the circuit, where m is the length of an interior register, one may to clock the source register m times and shift the serial output of the source register to the interior register. Then, the $(n+m)$-bit pattern (stored in both the source and the interior registers) can be applied to the CUT in a single clock cycle, as shown in Figure 4.8.

This scheme has two main drawbacks:

(a) the price paid for sharing the boundary scan resource is to introduce delays when each test pattern is generated. If the

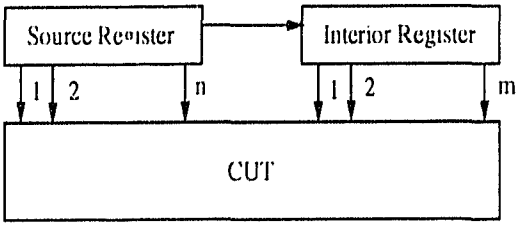


Figure 4.8: The test pattern generation scheme

average number of test patterns applied to a CUT with a reasonable fault coverage is, for example, ten thousands, then the delay of m cycles for every test pattern is significantly large;

(b) although an $(n+m)$-bit test pattern can be generated in this way, the number of distinct patterns can be very small in comparison with the total number of possible test patterns that can be produced by a PRPG of the same length. Faults may remain undetected due to the low test pattern coverage.

Using the principle of LCAR concatenation, an $(n+m)$-bit test pattern can be obtained by means of dynamic reconfiguration of the source and interior registers in one clock cycle, as opposed to the m-bit delay in the original scheme. Meanwhile, exhaustive sets of test patterns can be guaranteed for the n-bit, m-bit and $(n+m)$-bit PRPGs. To achieve this, we choose two primitive LCARs of length n and m, which can be concatenated to form a primitive LCAR of length $(n+m)$, so that the structure of the source and interior registers are defined, or partly defined by these LCARs. In addition, the interior register can be designed in such a way that it works with the source register as the $(n+m)$-bit PRPG in the BIST mode, and functions as a shift register otherwise. Thus, a notable improvement in the testing delay (especially for large

m) and fault coverage can be achieved. However, hardware overhead will be introduced when converting the regular interior register into a register/PRPG. For a large m, the overhead may contribute a significant portion of the overall cost. Tradeoff has to be made as a particular application is concerned.

(5) The latest application is the topic of some new research which has already suggested some promising results [61]. Details of this work are presented in Chapter 5.

As can be seen, the modularity and regularity of LCARs naturally fit the hierarchical nature of VLSI design and automatic design process. The high flexibility of LCAR concatenation enhances the modular approaches of such, and allows new designs and testing schemes to be explored.

4.6 Future Work on LFSR and LCAR Concatenation and Partitioning

There are many interesting topics which can be further investigated in LFSR and LCAR concatenation and partitioning.

(1) Theoretical issues:

(a) The mathematical basis for predicting the primitivity of both LFSR and LCAR concatenations is a major open question. Different approaches rather than the direct sum of square matrices could be required in solving this problem.

(b) Primitivity of LFSR and LCAR concatenations could be predicted by means of probability analysis. An investigation could

be conducted in terms of behaviour of (i) irreducible, (ii) primitive, (iii) minimum weight and (iv) reducible machines, as an alternative to an analytical approach to solve this problem.

(c) A mathematical basis for predicting if a given degree n polynomial can be partitioned into $(n - k)$ length primitive or irreducible bit slices is desirable.

(2) Application issues:

(a) For practical use, more comprehensive tables of the concatenation, especially for LFSRs and LCARs of large degrees, need to be produced.

(b) The cost of LFSR and LCAR with concatenation (partitioning) would be examined when low cost XOR gates are used. Performance of the two types of machines can be investigated in terms of their impact on a system with BIST, and with both boundary scan and BIST.

Chapter 5

The New Testing Scheme

In this chapter, we present a new testing scheme that merges concurrent checking, off-line data compaction and scan-based test with shared hardware resources. It is applicable to general circuitry and can be adapted to comply with IEEE Standard 1149.1.

Off-line data compaction is one of the most widely used BIST techniques in digital system design to replace or reduce the use of expensive external testers. The term *off-line BIST* usually refers to signature analysis. A general organization of the BIST scheme, shown in Figure 2.1 of chapter 2, is depicted in Figure 5.1 (a). A system designed with BIST permits two basic operations: BIST and NORMAL. During NORMAL operation, the test pattern generator and the signature analyzer used in BIST stay idle, as shown in Figure 5.1 (b) in the dotted lines. They are the potential resources to be used in concurrent checking in our new testing scheme. If the circuit is designed to comply with the boundary scan standard, it supports a SCAN mode which encompasses By-pass, Idcode/Usercode, Internal and External tests. The standard also makes an attempt to introduce some form of on-line testing through the "Sample" test available in the normal operation [1].

Initially, we assume that the circuit under test is combinational logic, and full

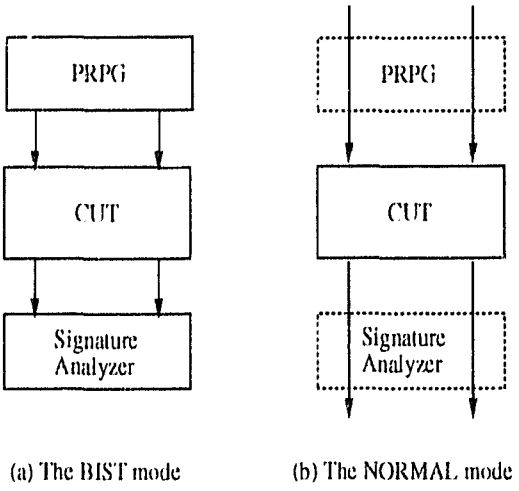


Figure 5.1: A BIST organization

scan design is used. These restrictions are lifted in the later sections of this chapter. We incorporate full concurrent checking in the new testing scheme, in addition to all testing modes above, and we use the hardware resources already in place. This is accomplished by using a unified method of test signatures, based on cyclic codes (as signature analysis with LFSRs or LCARs already is). Conversely, one may start with a circuit already designed with concurrent checking and add off-line signature analysis. We propose here that if concurrent checking is designed based on a cyclic code, resources can be used during off-line testing.

This chapter is organized as follows: we first introduce the new coding scheme, which plays a key role in making use of the idle BIST and boundary scan resources for concurrent checking. Then we present the new test scheme in section 2. We discuss some modified structures to improve the system performance and enhance the testability in section 3. A design template of a benchmark circuit is given in section 4. Finally, in section 5, we show that the scheme is applicable to sequential circuit testing and discuss some alternatives to full scan design in section 5.

5.1 The New Error-Detecting Code

The choice of error-detecting codes is crucial in a design with concurrent checking. The choice affects the following design factors:

- the types of hardware used for the encoding/decoding,
- the area overhead introduced, and
- the fault coverage.

Commonly used error-detecting codes in concurrent checking are Berger and residue codes, where Berger codes are used to detect unidirectional errors (mainly in PLAs) [46, page 338] and residue codes are used for error detection in arithmetic units [46, page 339]. Historically, PLAs have played an important role in digital system design, when circuit complexity increased dramatically and design automation tools for integrated circuit were still in their infancy. The regular nature of PLAs is suitable for implementing large combinational logic functions and is easily incorporated in a design automation process. It has been shown that, in a PLA, all internal single faults (single stuck-at faults, shorts between adjacent lines, or contact faults) cause only unidirectional errors at its outputs [31]. Consequently, codes that detect unidirectional errors, such as Berger and m out of n codes can be used to detect the faults [46, page 292]. With many powerful CAD tools available today, the new trend in implementing combinational logic is back to conventional gate realization, which is normally cheaper in silicon. Therefore, two new problems are raised in concurrent checking:

(1) general error-detecting codes are needed since unidirectional errors are no longer the main concern in standard gate realization;

(2) new models describing the relationship between faults and error patterns are required for error detection of gate realizations of general circuits.

Inputs Vectors	Output Vectors + LCAR code
001011	000001110000 00
001001	000001010000 10
101110	100001000000 01
110011	101000100000 00
010-00	000100000000 10
10-010	010010000000 00

Table 5.1: An example function encoded with LCAR codes

In this section, we propose a new error-detecting code, called the LFSR/LCAR code, to replace Berger and residue codes in concurrent checking. The LFSR/LCAR code is a separable cyclic code, used in many applications, but not previously in concurrent checking.

A LFSR/LCAR code, applied to concurrent checking, is generated in the usual manner as for other such error-detecting codes. The information bits are sent into an encoder, an LFSR/LCAR; this in turn produces the checking bits which are attached. In circuit design, an output word is fed into the encoder, one bit at a time. At the end of the computation, we use the last state of the encoder provide the checking bits and attach them to the output word to form the entire code word. As explained in Chapter 3, the circuit is now implemented with the augmented code words as output

Table 5.1 shows a segment of a truth table describing a function of 6 inputs and 12 outputs, encoded with a 2-bit LCAR code, where the structure of the LCAR is defined by 10, corresponding to a characteristic polynomial of $x^2 + x + 1$, and a machine with a rule-150 and a rule-90 cells. For example, the original output vector 000001010000, on the second row, corresponding to the input stimulus 001001 becomes 000001010000 *10* after the encoding, where the last two bits are the LCAR (cyclic) code.

For a circuit description given in cubes[1], the encoding process can be automated by a computer program performing the following functions:

(1) call a minimizer (e.g. *espresso* [6]) to minimize the given function;

(2) obtain the fully specified truth table if "don't cares" exist in the given truth table representing the circuit function;

(3) compute a LFSR/LCAR code for every output word in the fully specified circuit description, and append it to the corresponding output word;

(4) call the minimizer again to minimize the augmented function (increased in the number of output bits), to be implemented as the main circuit.

A Concurrent Checking code generator and MINImizer (*ccmini*) was designed and implemented to automate the procedures above, using *espresso* as the minimizer. *Ccmini* is capable of adding Berger, residue, parity, and LFSR/LCAR codes to a given circuit function. Note that *espresso* can be appropriately substituted by other minimizer . The *ccmini* manual page can be found in Appendix 2.

For a circuit designed and implemented with LFSR/LCAR error detecting codes, concurrent checking can be enabled during normal system operation. Given an input combination, the CUT generates the normal outputs and the coding bits. The normal outputs serve two purposes: they are utilized by other system functions, and also serve as the information bits to be checked against the coding bits. This checking is performed by recomputing the LFSR/LCAR code from the normal circuit outputs in real time, using a LFSR/LCAR with the same structure as the logical LFSR/LCAR employed in the encoding process. The newly computed LFSR/LCAR code is then compared with the checking bits, which form a part of the CUT outputs. A fault is signalled if the two codes differ.

[1] A truth table form plus some information on number of inputs, outputs and product terms [6].

In determining the overhead of a concurrent checking scheme, there are two concerns: the augmented CUT and the concurrent checking test logic. The area overhead of the augmented CUTs depends on the error-detecting codes used as well as the circuit functions. We show later that the introduction of an LFSR/LCAR cyclic code proves to be significant because: (1) there is no loss of concurrent error detection from other codes; (2) it permits shared implementation with signature analysis and scan-based test.

The cost of the concurrent checking logic depends heavily on the error-detecting codes employed. In the case of the new code, an LFSR/LCAR of length-r is used as the code generator, where r is the number of checking bits. We recommend LCARs over LFSRs in our proposed scheme due to the ease of concatenation and partitioning of LCARs, as discussed in the previous chapter, not withstanding the higher area overhead of LCARs. However, LFSRs can certainly be used with the same effectiveness of fault coverage. Detailed discussion on the area overheads and other cost measurements can be found in Chapter 6.

5.2 The Primary Scheme

Our proposed test scheme complies with the IEEE Std. 1149.1, with the three main modes of operation: NORMAL, SCAN and BIST. We modify the NORMAL mode by introducing concurrent checking, and keep the other two modes unchanged.

5.2.1 The NORMAL Mode

In the previous section, we show how to encode a given circuit function with LCAR codes at the design stage. As shown in Figure 5.2, the CUT takes data from the input bus and produces code words (outputs) of $(m + r)$ bits. The normal (m-bit) outputs proceed in the data path to other modules in the system. The concurrent

checking facility is called the *checker*. It consists of two functional units, the *code generator* and the *comparator*. The code generator is capable of recomputing the coding bits from the normal outputs in real time, and the comparator compares the coding bits carried by the CUT to the newly computed one. In the code generator, the normal outputs are loaded into the m-bit shift register, and the coding bits are held by the r-bit buffer. The shift register provides bit-serial inputs to the compactor (with LCARs), and the compactor computes the expected coding bits in real time. A comparison can be made between the expected coding bits and the ones stored in the buffer after m clock cycles. The presence of a fault is signalled if the two are different.

In this initial approach, the concurrent checking is conducted only for output words $i \times m$, where $i = 0, 1, 2, \ldots$, due to the latency introduced by the serial LFSR/LCAR code computation. Some modified structures for improving the system performance are found in the next section.

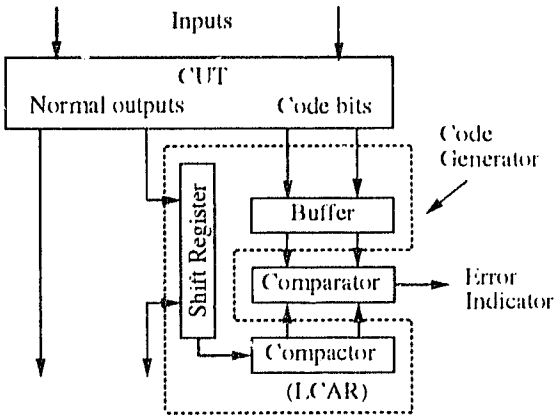


Figure 5.2: The concurrent checking organization

We introduce here three definitions to help evaluate a concurrent checking scheme.

Definition 5.1 The delay introduced by the serial code computation is the *concurrent checking latency*. It is measured by the number of clock cycles.

Definition 5.2 If one out of every t consecutive output code words, $t \geq 1$, is examined in concurrent checking, then $1/t$ is the *concurrent checking frequency*, t is the *concurrent checking interval.*

In the proposed NORMAL/concurrent checking mode, the concurrent checking latency is m clock cycles, and the concurrent checking frequency and interval are $1/m$ and m, respectively.

The concurrent checking scheme requires the following hardware resources:

(a) an m-bit shift register;

(b) an r-bit buffer;

(c) an r-bit signature analyzer with LCAR implementation;

(d) an r-bit comparator.

5.2.2 The SCAN Mode

Prior to a scan test, the concurrent checking test logic, the shift register, the buffer and the compactor, or the BIST test circuitry, to be described in section 5.2.3, are reconfigured to form a shift register chain, as shown in Figure 5.3 (a). The testing of the circuit is performed in two phases. The first phase consists of a functional testing of the shift register. The test is performed by shifting three sequences of test vectors in and out of the register. A sequence of ones (zeros) is followed by a sequence of zeros (ones), and by a sequence of ones (zeros). For example, if a sequence of zeros is shifted into the register, and the sequence shifted out contains some ones, then stuck-at-1 faults in the register are detected. Such sequences are used to check if every storage element of the register can flip from 1 to 0 and from 0 to 1 properly.

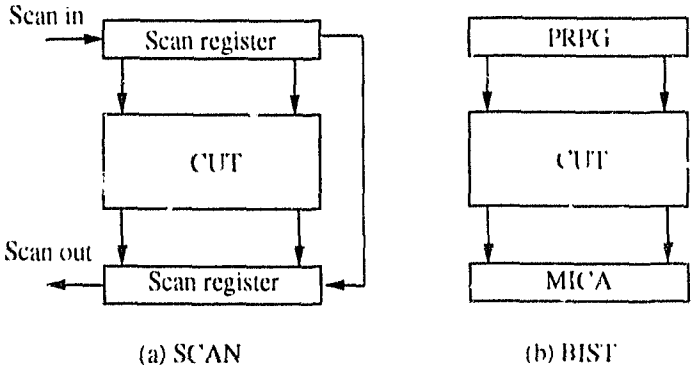


Figure 5.3: The SCAN and BIST organizations

The second phase of testing involves the testing of the combinational part of the circuit. The test is accomplished by (a) selecting the SCAN mode and scanning in a test vector, (b) selecting the NORMAL mode, applying the stimulus to the CUT, clocking the circuit once and collecting the circuit response in the output portion of the shift register; (c) switching to the SCAN mode again and shifting the circuit responses out for evaluation. Ideally, the second phase is repeated many times by applying different testing vectors in order to achieve high fault coverage.

Two scan registers are needed in the SCAN mode:

(a) an n-bit scan register, where n is the number of inputs of the CUT;

(b) an $(m + r)$-bit scan register.

5.2.3 The BIST Mode

In this mode, the testing hardware in the NORMAL or the SCAN mode needs to be configured into a PRPG and a MICA (Multiple Input Cellular Automata), as shown in Figure 5.3 (b). The PRPG generates input vectors and the MICA compacts the outputs into a signature in parallel, as in conventional signature analysis. Then, this signature can be shifted out and compared with the expected one.

The hardware resources for the BIST mode are as follows:

(a) an n-bit PRPG;

(b) an $(m + r)$-bit signature analyzer with MICA.

5.2.4 Sharing of Hardware Resources

It has been shown that all the hardware resources (except the r-bit comparator for concurrent checking) in the three modes of operation are register-based units. Applying the concatenation and partitioning properties of LCARs, dynamic reconfiguration can be achieved, so that the hardware resources can be shared in the different modes of operation. For example, the items (a), (b) and (c) of the NORMAL mode can be shared with the items (a) and (b) of the SCAN mode, as well as with the items (a) and (b) of the BIST mode. A design template for a benchmark circuit is given in a later section of this chapter, where detailed considerations of the sharing are more evident.

5.2.5 Summary

We have shown the main concepts of our testing scheme. The key idea is to employ LCAR based cyclic codes to replace conventional error-detecting codes for concurrent checking, since, after all, commonly used signature analysis is based on cyclic codes. The LCAR-based code generator is cost effective in comparison to those for Berger and residue codes, and can be used with the off-line test resources, i.e. the scan register in the SCAN mode, and the PRPG and the MICA in the BIST mode, through simple dynamic reconfigurations.

In the next two sections, we first present some modified structures to overcome some of the problems in the primary scheme, thereby improving the system performance. We then present a design template of the proposed scheme.

5.3 The Modified Structures

The testing scheme presented in the previous section demonstrates a new DFT and BIST structure. One drawback of this scheme is the concurrent checking latency due to the time delay of serial data compaction. In this section we suggest three modified structures: *output partitioning*, *output multiplexing* and *output partitioning & multiplexing*, to increase the concurrent checking frequency. In all cases, the principle of the SCAN test and the BIST remains the same.

5.3.1 Output Partitioning

In the primary testing scheme introduced in the preceding section, a code word (outputs) of a CUT consists of m-bit regular outputs and an r-bit LCAR code. We partition the m-bit output vector into k subvectors and the r bit coding bits into k segments, where $1 < k < m$ and $m/k > r$, depending on the speed requirements of the circuit and the hardware overhead which can be tolerated. The CUT is designed in such a way that each subvector of the output is encoded by an (r/k) bit LCAR code. In concurrent checking, the k subvectors are fed into their corresponding LCARs simultaneously, and m/k clock cycles are needed to compute the LCAR codes. Then, the comparator compares the r-bit recomputed LCAR codes collected from the k LCARs with the r-bit coding bits stored in the k buffers. Hence, with the partitioning, the concurrent checking latency is reduced from m (for the primary scheme) to m/k. Moreover, the cost in silicon is the same for both schemes since the same number of LCAR code bits (r) is used to encode the circuit outputs, and the same numbers of shift registers, buffers LCARs are required for concurrent checking. The aliasing probability of the LCAR code remains the same.

Figure 5.4 depicts an example of the partitioning for $k = 2$ and $r = 4$. The normal outputs and the coding bits of the CUT are logically divided into two parts.

The two output portions are denoted by O_1 and O_2 and the two coding segments are named as C_1 and C_2, respectively. The two LCARs, LCAR-1 and LCAR-2, work in parallel to compute the corresponding LCAR codes for O_1 and O_2. The comparison of the recomputed LCAR codes and the coding bits can then be made in a single clock cycle.

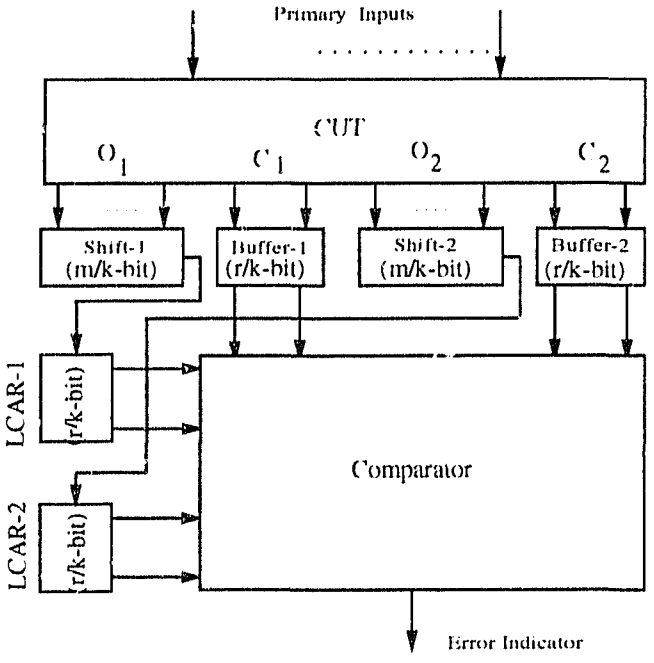


Figure 5.4: The output partitioning organization

5.3.2 Output Multiplexing

If concurrent checking is required for every output word, we can duplicate the *code generator* of the primary scheme, shown in Figure 5.2, m times, such that an output code word is sent (multiplexed) to a code generator, as depicted in Figure 5.5. For example, code generator 1 can be enabled when the first code word is available, and set 2 is assigned to the second code word, and so on. Each of the data compaction sets takes m clock cycles to compute a LCAR code. At time m, the LCAR code

of the first code word is ready in set 1, and comparison between the LCAR code and the coding bits held in the buffers can be made by the comparator. At time $(m+1)$, data compactor set 1 starts processing output code word $(m+1)$, and the LCAR code of the second code word is ready in code generator 2. It takes m cycles to fill up the m code generators, then all the sets are working simultaneously. The comparator receives the first pair of valid data at clock m and is busy the rest of the time. Thus, every output word is checked with m unit concurrent checking latency, and an additional $(m-1)$ code generators are needed in silicon.

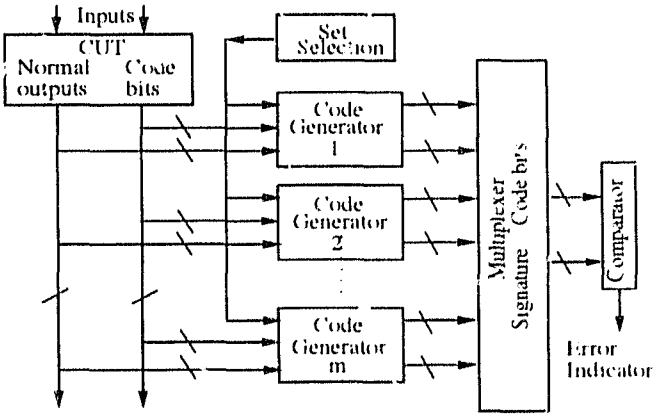


Figure 5.5: The output multiplexing organization

5.3.3 Output Partitioning & Multiplexing

This alternative structure is a combination of the partitioning and the multiplexing schemes. The CUT is first designed according to the techniques described in the partitioning. Next, the concurrent checking hardware (excluding the comparator), as shown in Figure 5.4, is duplicated (m/k) times to form a system which is the same as that of the multiplexing, except that the number of code generators is reduced to m/k.

The newly constructed system has the advantages of the two combined struc

tures, with (m/k) concurrent checking latency, the highest checking frequency (every output word is examined) and the silicon cost of $(m/k-1)$ data compaction sets.

5.4 A Design Example

In this section, we present a design template for the output partitioning test architecture introduced in section 5.3.2. A Berkeley[2] benchmark PLA circuit, *apex4*, is used to show a complete design procedure. The system implementing *apex4* has three modes of operation: NORMAL, SCAN and BIST.

Encoding of the circuit function: *Apex4* has 9 inputs, 19 outputs and 438 product terms, given in the input format of *espresso*. The outputs are partitioned into two parts, $O_1 = o_{18}o_{17}\ldots o_9$ and $O_2 = o_8o_7\ldots o_0$, and each of the segments is encoded by a 2-bit LCAR code, i.e. $k = 2$ and $r = 4$. The LCAR codes are computed following the encoding procedures given in section 4.1 for each partition. The augmented function thus has 23 outputs, where 19 of them are the normal outputs and 4 of them are the LCAR coding bits. For example, the output word "00010000000010010000", corresponding to the input stimuli "01101i011", becomes "0001000000 *00*" "010010000 *10*", where the coding bits are in italics.

Choices of LCARs: LCARs should be chosen carefully when designing the circuit for the NORMAL and the BIST modes. In the SCAN test, all storage cells are configured into a shift register, where the structure of LCARs does not matter. The quality of test pattern generators and data compactors depends on the structure of LCARs or LFSRs[21]. Primitive LCARs/LFSRs have been used almost exclusively in test applications. However, these linear machines have some associated cost, although they are cheaper in silicon than other approaches. Tradeoffs need to be made between the quality and the cost of the machines. We have suggested in

[2]With the Berkeley VLSI design tools.

section 4.4 that minimum-weight primitive LCARs are the most suitable choices. We show below how to choose LCARs by going through the design process of *apex1*.

A Pseudorandom Pattern Generator (PRPG) and a Multiple Input Cellular Automata (MICA) are required in the BIST operation. LCAR structures implementing the two mechanisms are determined by the following considerations. For the PRPG, we choose a length 9 (corresponding to the 9 inputs of the circuit) minimum cost primitive LCAR, "1000000000", which has the maximal length cycle and is the cheapest in hardware implementation. Where the MICA is concerned, other issues should be taken into consideration as well.

- First of all, the MICA should be primitive. If a standard length MICA, e.g. 16 bit, is required and the number of outputs is greater than the standard length, a multiplexor can be used to reduce the outputs to the required length. In this design template, we simply use a MICA that has the same length as the outputs, i.e. 23 bits long. Another version of this design was carried out with a 16-bit register and a multiplexor, with similar results.

- Secondly, the 23-bit MICA should be partitionable into 2 bit primitive LCARs, in order to compute LCAR codes during the concurrent checking, one for each output partition.

- Thirdly, all the primitive LCARs, namely the 2-bit short LCARs and the 23 bit long LCAR should contain the minimum number of 1's in order to reduce the hardware cost. Since there exist only two primitive LCARs of length 2, namely "01" and "10", the 23-bit LCAR can consist of any combination of the 2 bit LCARs. This implies that the LCAR should have at least two 1's. It turns out that the LCAR, "01000010010000000000000", is one of the best choices meeting all the requirements above. We show later how this LCAR structure can be used to obtain the maximum sharing between different operation modes

and the best machine performance.

Logic design – the NORMAL mode: Figure 5.6 shows the logic diagram of the concurrent checking organization, where the Shift-1 and Shift-2 registers hold the output segments O_1 (the outputs 0 to 8) and O_2 (the outputs 9 to 18) and the Buffer-1 and Buffer-2 registers contain the coding bits C_1 (the coding bits 0 to ') and C_2 (the coding bits 2 to 3). In real time, registers Shift-1 and LCAR-1, and registers Shift-2 and LCAR-2, form two sets of serial data compaction sets, working in parallel. The concurrent checking latency in this case is 10 clock cycles, which is the length of the longest output partition.

Logic design – the BIST mode: Figure 5.7 shows the block diagram of the BIST architecture. The PRPG and the MICA are 9- and 23-bit, respectively.

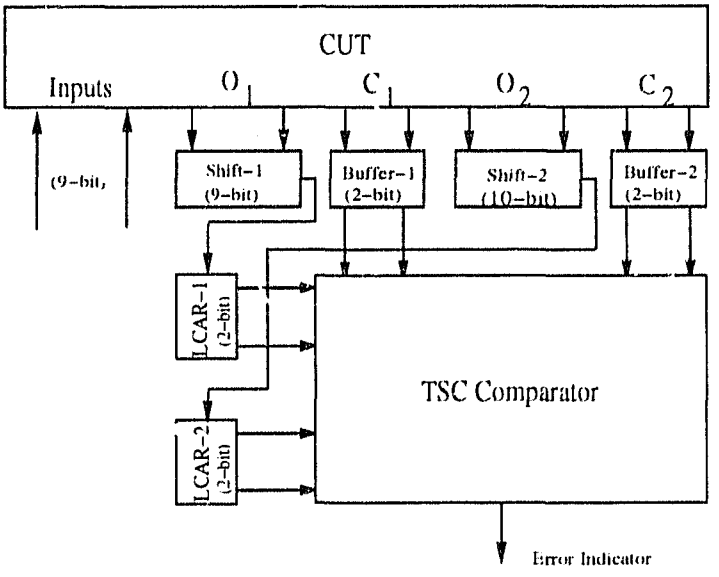


Figure 5.6: Apex4 concurrent checking organization

Figure 5.8 shows the sharing of MICA and the concurrent checking resources. The binary string at the top represents the structure of MICA (implementing LCAR "01000010010000000000000"). The LCAR-1 and Buffer-1 registers, each 2 bits long,

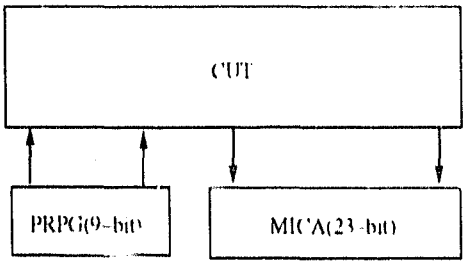


Figure 5.7: Apex4 BIST organization

are shared with the first 4 bits of the MICA, where the first two bits implement the primitive LCAR, "01". The Buffer 2 and LCAR-2 regiters, each 2 bits long, are shared with the second 4 bits of the MICA, where the last two bits implement the primitive LCAR "10". The Shift-1 register, 9 bits long, has the same length as the PRPG and perfect sharing can be made between the two. Finally, the Shift 2, 10 bits long, can be utilized by the next 10 bits of the MICA.

The MICA consists of five types of bit-sliced cells: MICA90(150)/LCAR90(150), MICA90/Buffer, MICA90(150)/Shift, MICA90/- and Connector, where the left portions of the "/" denote the function of the cells in the BIST mode, and the right portions represent their functionalities in the NORMAL and concurrent checking. For example, cell MICA90/LCAR90 works as a rule 90 cell of the MICA in the BIST mode and a rule-90 cell of the LCAR-1(LCAR 2) in normal operation. It is assumed that in the SCAN mode every cell is connected in the shift chain. Thus the scan function is not explicitly stated in the cell names. The *Connector* are simple data switches to concatenate and partition the MICA block into different operation modes.

Similarly, Figure 5.9 shows the block diagram of the PRPG in the BIST mode and the Shift-1 in the concurrent checking. The binary string again denotes the structure of the PRPG, "100000000". The cells used are PRPG90(150) Shift

The logic designs of the bit-sliced cells mentioned above are depicted in Fig

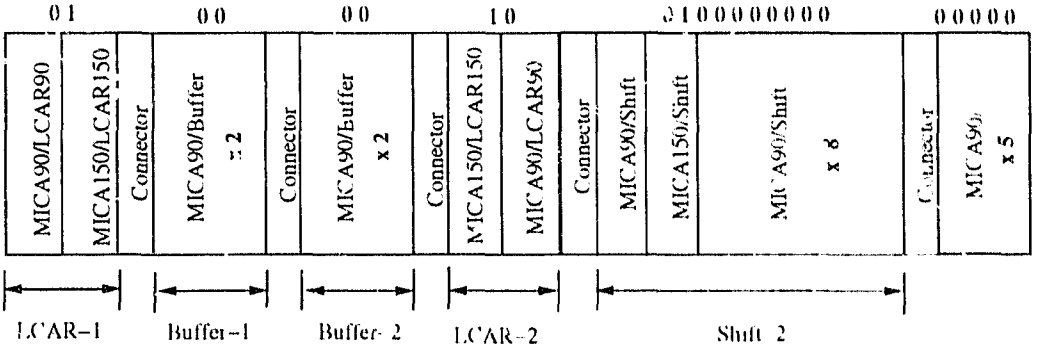


Figure 5.8: The block diagram of the MICA

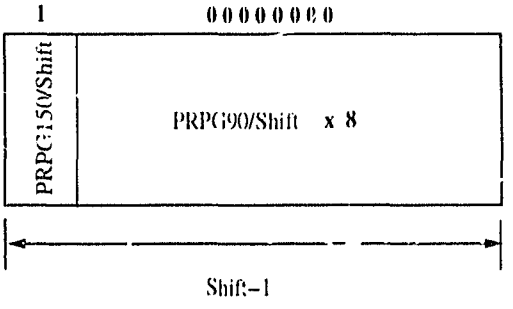


Figure 5.9: The block diagram of PRPG

ures 5.10, 5.11 and 5.12. The cells have three basic functions: the *load* signal loads data into the flip-flops, the *shift* signal shifts the content of neighbour $i-1$ to cell i, and the *offline* signal enables the BIST mechanism such as the PRPG and the MICA. We only show the design of rule-90 cells in the figures. The corresponding rule-150 implementations are similar to the rule 90 cells, except one more EXOR gate is needed to add the current state of cell i to the sum of the current states of cells $(i-1)$ and $(i+1)$.

Logic design – the SCAN mode. All the storage elements of the test logic (for both the NORMAL and BIST modes) are connected to form a shift register chain, assuming that full scan design is used. Alternative approaches to the full scan design are discussed in section 5.5.2.

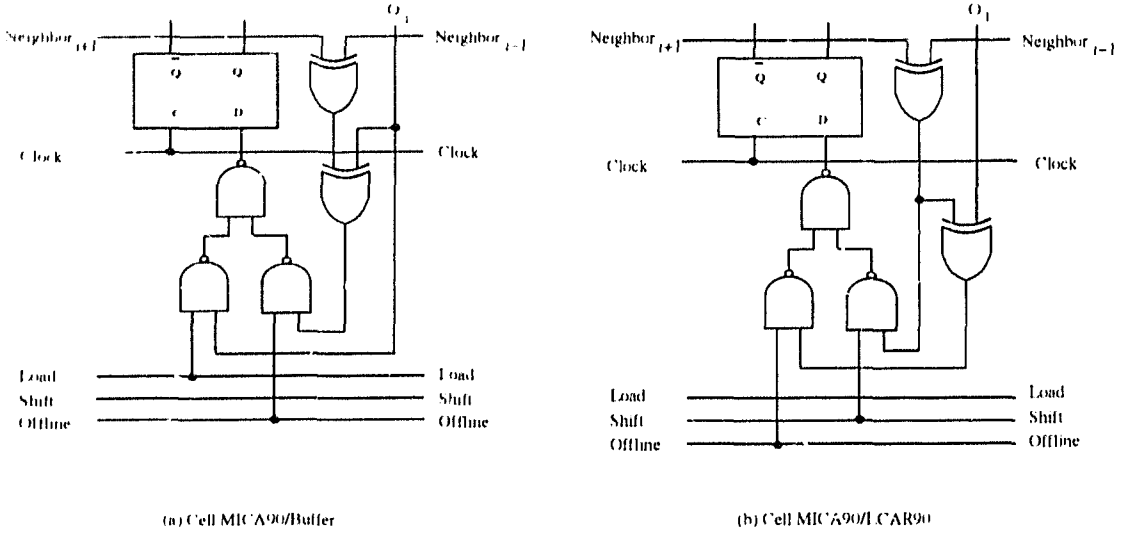
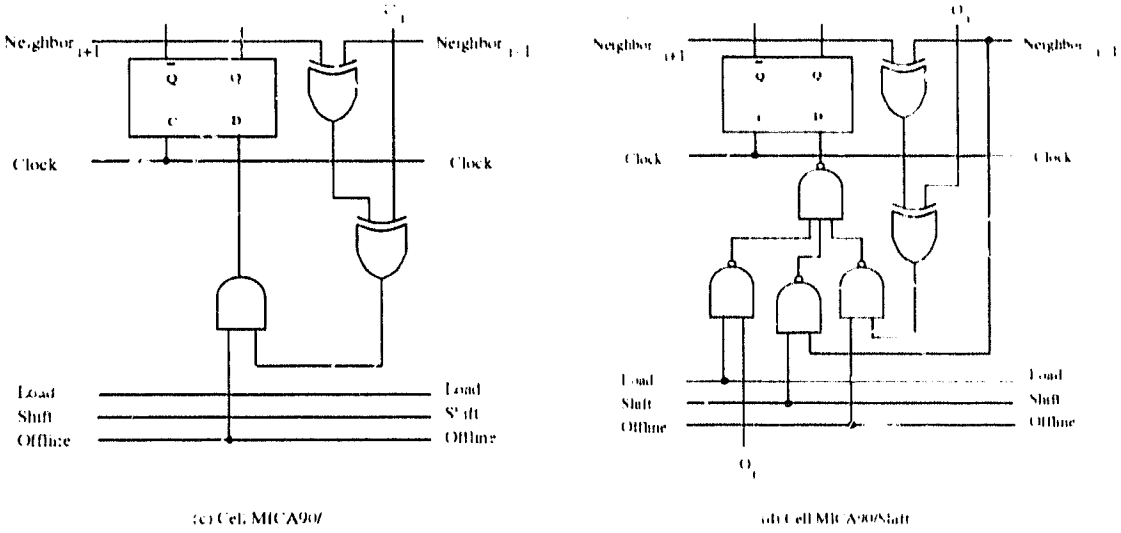


Figure 5.10: Logic design of bit-sliced cells

Logic design – the Comparator. Finally, a comparator, which compares the coding bits and the newly computed LCAR code, is needed in concurrent checking. It is shown in [32] that a tree realization of an r-bit comparator is Totally Self-checking (TSC) [2, pages 581-585] if all 2^r combinations of the checking bits are available. The r-bit comparator is formed as an interconnection of comparator modules [32], with two input pairs (know as 2-tail checker [2, page 585]). Figure 5.13 (a) and (b) show the comparator module and the tree realization constructed from the comparator modules, where (a) compares two input pairs, a_1a_2 and b_1b_2 and detects all single faults during operation, and (b) shows the block diagram of a 4-bit TSC comparator.

In summary, in the BIST modes, there are a total of (9 + 23) bit-sliced cells needed to form the PRPG and the MICA, while concurrent checking requires a total of (23 + 4) bit-sliced cells, which is a subset of the BIST resources. Thus, the only hardware introduced by concurrent checking is the comparator, which is inexpensive for a small number of checking bits, plus the silicon for the encoded circuit function.



(c) Cell MICA90/ (d) Cell MICA90Shift

Figure 5.11: Logic design of bit-sliced cells (continued)

A complete examination of area overhead (including this case study), and other associated costs, can be found in Chapter 6.

5.5 Testing of Sequential Circuitry

A sequential machine can be defined as a 6-tuple $< I, O, S, \delta, \lambda, s_0 >$, where I is the input alphabet, O is the output alphabet, S is the set of state, δ is the next state function, λ is the output function, and s_0 is the initial state. The *espresso* format of a sequential machine can be used [6]. The left block of Table 5.2 shows the *espresso* description of a benchmark circuit, dk27, which has one input, two outputs and 7 states. Columns 1, 2, 3 and 4 correspond to the input, the current state, the next state and the outputs, respectively. In the right columns of the table, we show a possible binary state assignment. Thus, a sequential circuit can be defined by a state transition table describing the input and output relations. The inputs include two parts: the primary inputs and the current states of the machine, while the outputs are the primary outputs and the next states.

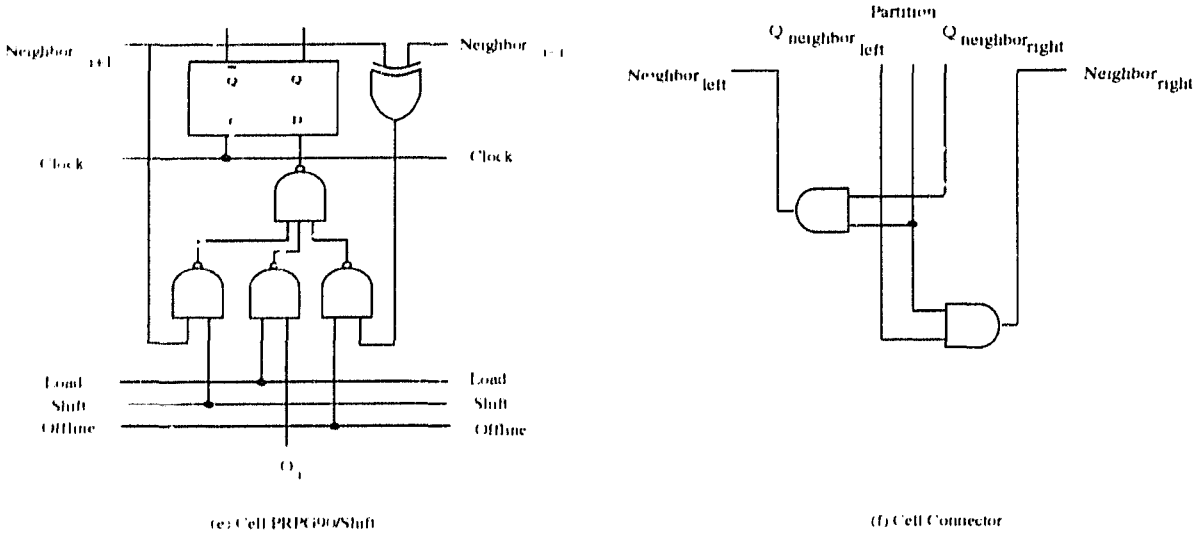
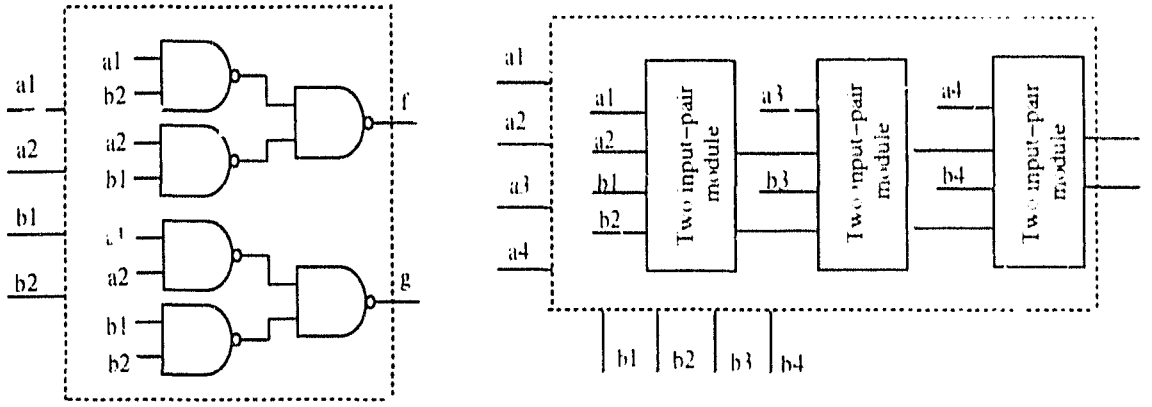


Figure 5.12: Logic design of bit-sliced cells (continued)

A sequential circuit can be designed in our proposed scheme in a similar manner as for combinational logic. At the design stage, we encode the circuit outputs (primary outputs and the next state) by LCAR codes following the procedures given in section 5.1. The system supports three modes of operation: NORMAL, BIST, and SCAN test, where the BIST and the SCAN modes work similarly to the case of combinational networks. Our main concern is concurrent checking. We briefly describe the BIST and the SCAN modes of operation below, then devote the rest of this section to concurrent checking.

The BIST mode. The state register of a sequential machine is configured as part of the MICA, and the feedback from the outputs of the state register to the inputs of the circuit are disconnected, as shown in Figure 5.14. The length of the PRPG is $(n + s)$, where n and s are the numbers of the primary inputs and the length of the state register, respectively. Thus, the combinational subnetwork of the machine can be tested in the same manner as in conventional parallel signature analysis.



(a) The two input–pair module (b) The 4–bit comparator

Figure 5.13: Basic modules of the TSC checker

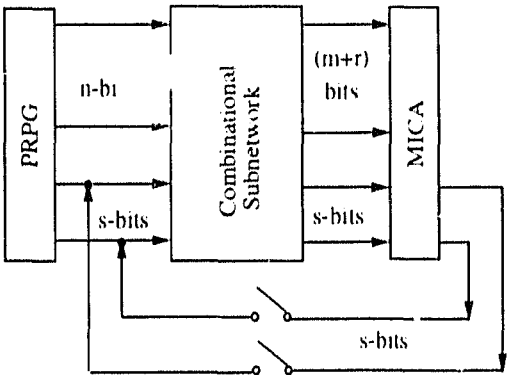


Figure 5.14: The BIST organization of a sequential circuit

The SCAN mode. The state register is reconfigured as part of the scan chain at the outputs, as shown in Figure 5.15. The scan register and the combinational subnetwork can be tested by shifting in test vectors, and then connecting and shifting out the circuit response from the scan register.

The NORMAL mode. In order to conduct concurrent checking, circuit outputs are used as the information bits encoded, in this case, by a LCAR code at design time. For a sequential circuit, the circuit outputs to be encoded are considered to be the concatenation of the primary outputs and the current state.

.type fr				.ype fr			
.start state1				.s'art state1			
.mv 4 1 -7 -7 2				.mv 4 1 -7 -7 2			
.ki s				.kiss			
.p 14				.p 14			
0	state1	state6	00	0	001	110	00
0	state2	state5	00	0	010	101	00
0	state3	state5	00	0	011	101	00
0	state4	state6	00	0	100	110	00
0	state5	state1	10	0	101	001	10
0	state6	state1	01	0	110	001	01
0	state7	state5	00	0	111	101	00
1	state1	sta.e4	00	1	001	100	00
1	s.ate2	state3	00	1	010	011	00
1	state3	state7	00	1	011	111	00
1	state4	state6	10	1	100	110	10
1	state5	state2	10	1	101	010	10
1	state6	state2	01	1	110	010	01
1	state7	state6	10	1	111	110	10
.end				.end			

Table 5.2: Espresso expression of dk27

Figure 5.16 shows the conceptual diagram of the NORMAL/concurrent checking operation. We demonstrate later that the error coverage of this system can be further improved, and the structure can be simplified, by using some existing BIST resources.

We assume that the State Register (S.R.), the Outputs Buffer (O.B.), the Code bits Buffer (C.B.) and the Next State buffer (N.S.) are working under the same clock. Thus, at time t_0, the S.R. holds the current state of the sequential machine. When the primary inputs are available, the combinational subnetwork produces the primary outputs and the next states. At time t_1, the current primary outputs, coding bits, and the next state variables are loaded into the O.B., the C.B. and the N.S., respectively. Note that the next state of the circuit becomes the current state at t_1, and is also held in the S.R. We compute the LCAR code of the primary outputs and the next states for the inputs and the current states at time t_0 by feeding the contents of the N.S. and the O.B. into the data compactor (LCAR). This takes $(s + m)$ clock cycles. The comparator then compares the contents of the

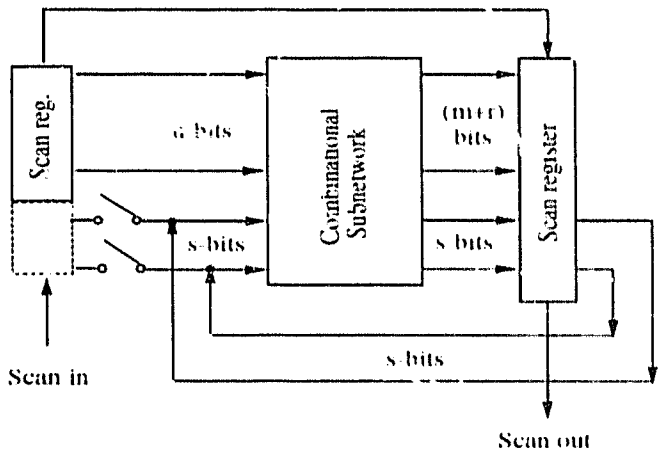


Figure 5.15: The SCAN organization of a sequential circuit

LCAR and the C.B. to determine if a fault is present.

By using the structure shown in Figure 5.16, the following faults can be detected besides for aliasing:

- detectable faults in the combinational subnetwork,
- faults in all the storage devices (except the S.R.), and on the signal lines between the combinational subnetwork and the buffers, and
- the faults on the connections between the combinational network and the S.R. to the N.S.

These faults can be divided into three types: the output bits are faulty, the next states are faulty, and both the outputs and the next states are faulty. However, faults in the S.R. or on the outputs of the S.R. cannot be detected by this scheme, since a faulty current state is directly connected to the inputs of the circuit and leads to another entry of the state transition table to be exercised. Thus, these faults are equivalent to the faults on primary inputs of a circuit. Below, we propose two alternative structures to detect faults on primary inputs for both combinational and sequential networks.

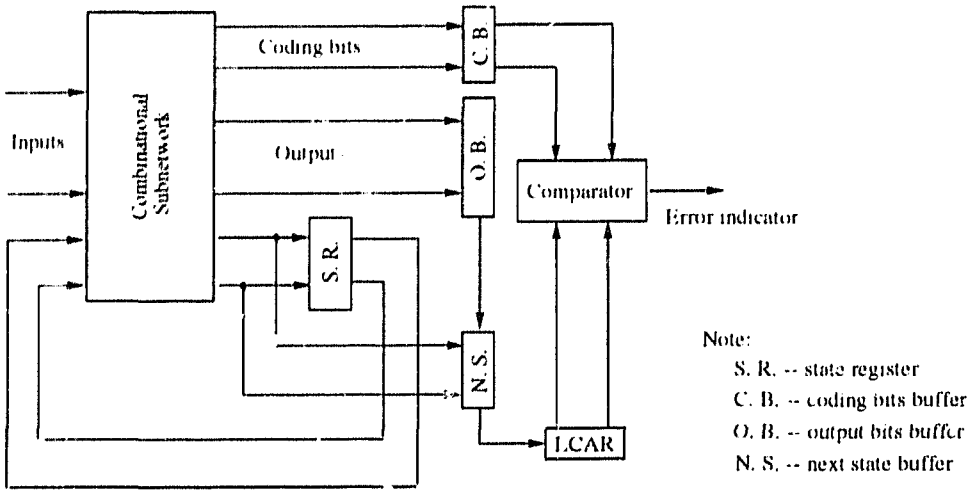


Figure 5.16: The concurrent checking organization of a sequential circuit

5.5.1 Alternatives to Full Scan Design

Although full scan techniques increase the controllability and observability of sequential circuits, and offer good test coverage, they have some drawbacks: (a) for long scan chains, the testing time becomes very long; (b) scannable flip-flops have longer propagation delays than non-scannable flip-flops, thus, they cause scan designs to have lower performance.

Partial or *incomplete scan path* is an alternative to full scan design [3, 10, 11]. It is based on the assumption that the difficulty in testing sequential circuits resides mostly in some localized parts of the circuit. In partial scan, only a subset of the flip-flops are configured in a shift register. Hence the problem becomes how to choose a set of flip-flops which gives the best improvement in testability.

For the testing schemes, for both combinational and sequential circuits, proposed in the previous sections, if full scan path is too expensive in silicon or test time, partial scan techniques can be used to reduce the overhead. However, if the BIST is included in a design in the first place, full scan can usually be incorporated without paying extra cost in silicon. In the case when test time is the main concern, partial

scan provides considerable flexibility of design choices.

It has been recommended that scan test can be used in conjunction with boundary scan at the chip level. Since boundary scan cells make use of the silicon areas next to i/O pads, savings are obtained by sharing the resources of the scan cells with that of the BIST hardware. Moreover, if there is more than one function block in a chip, they can be tested one at time using the same boundary scan/BIST hardware resources. These same savings can be inherited and included when merging BIST and concurrent checking as proposed here.

5.5.2 Conflict with the Boundary Scan Standard

The boundary scan standard mainly addresses the problems of off line testing. It provides a basis for establishing a framework to combine off line BIST and SCAN testing to facilitate the problems of chip, board and system level testing. In this section, we explore the feasibility of adapting our new testing scheme to support the standard. Since the BIST and SCAN operations are directly supported by the standard, the problem becomes how to incorporate concurrent checking under this standard.

In Chapter 3 we showed that the boundary scan standard defines six modes of testing: Bypass, Idcode/Usercode, Sample/Normal, Internal, External tests and Built-In Self-Test [1]. We classified the six test modes into three categories: NORMAL (including Sample/Normal), SCAN (consisting of Bypass, Idcode/Usercode, Internal and External tests) and BIST (including Built-In Self-Test), where all the tests under the SCAN category employ scan-based techniques. Thus, a system designed to support the three operations should be adaptable to the standard.

The Sample/Normal mode of the standard allows a snapshot of the normal system operation (see also Chapter 3). The NORMAL/concurrent checking test of our scheme works in real time as well. There is a conflict between the operations defined

by the Sample/Normal mode of standard and the NORMAL/concurrent checking of our testing scheme. Two possible solutions are given as follows:

(1) In the Sample/Normal mode, circuit responses are captured in the scan registers, whose contents are then shifted out for examination. To achieve this, it is required that the scan registers operate under a separate testing clock. This also implies that the scan registers cannot be used for other purposes in this mode. Moreover, since Sample/Normal is one of the mandatory test modes defined by the standard, we suggest that a design incorporating our proposed scheme add an user defined test mode, which can be called "concurrent checking", such that Sample/Normal operation can be performed as defined in the standard, and Normal/concurrent checking can also be supported for closer on-line monitoring. A modification of our test scheme is required: the data compaction set(s) for the NORMAL/concurrent checking must be independent of the SCAN and the BIST resources. Since the mechanism to implement LCAR codes is cheap, the overhead due to the separation of concurrent checking and off-line resources would not be significant to the overall cost. Thus, our testing scheme can be adapted to accommodate the boundary scan standard.

(2) It is also possible that the restrictions on using separate clock and scan registers in the Sample/Normal mode will be lifted in future standards, if more efficient on-line testing architectures can be found to replace the Sample/Normal operation.

5.6 Summary

In this chapter we presented the new testing scheme which involves the merging, in a built-in fashion, of concurrent checking and off-line BIST. This method is applicable to general circuitry. Concurrent checking is achieved using LCAR-based cyclic codes,

replacing other conventional codes. The choice of error detecting code is crucial. It allows the sharing of hardware resources in the on line and off line testing modes by means of LCAR concatenation, because the LCARs used in concurrent checking are the same ones used as signature analyzers.

Three modified structures for improving the concurrent checking latency and interval are presented. The best performance with our schemes is 100% concurrent checking with (m/k) checking latency, where m is the number of output bits of a circuit under test and k is the number of partitions over the outputs. Alternative implementation structures are discussed and a design template is provided.

Chapter 6

Cost Characteristics

This chapter is devoted to cost measures that are suitable for evaluating the testing scheme of Chapter 5. The concerns are area overhead, testing time, design complexity, and pin count. In section 1, we discuss the problems of existing area overhead estimation methods and present a new method of estimation for general circuits. In section 2, we evaluate the proposed scheme and show some evaluation results on benchmark circuits. Fault coverage, which is another crucial measure of a testing scheme, is treated separately in Chapter 7.

6.1 Area Overhead Estimation Methods

Area overhead is one of the main concerns of a BIST scheme. Comparisons are very difficult among designs presented in different technologies, CAD tools and benchmarks. There are two basic requirements for reasonable comparisons: a set of standard benchmarks, and a universal cost estimation method. Researchers and design engineers have been using benchmarks such as Berkeley[1] and ISCAS89[2] benchmarks

[1] With Berkeley tools.

[2] 1989 International Symposium on Circuit Systems.

for testing scheme evaluation. However, little effort has been made on area estimation methodologies

Historically, research in on line and research in off line testing techniques have been developed separately. Distinct area estimation methodologies are employed Area cost of off-line testing schemes is usually measured directly on circuit geometry layouts, as the added testing circuitry is usually quite distinct from the original circuits. However, one may not be interested in a particular design. Thus, going through a complete design to make such a measurement becomes uneconomical. An alternative is to count the number of transistors involved in designs [22]. The main drawback of this method is that it is only suitable for standard gate implementation of designs. For example, the number of transistors in a PLA does not reflect the cost of silicon.

In concurrent checking, a given circuit function is encoded by error detecting codes. The augmented function usually results in a circuit that is bigger in silicon than the original one. In addition, a checker is needed to monitor the operation of the augmented circuit in real time. Theoretically, both the circuit and the checker can be implemented by a PLA (two-level construct) or a standard gate realization (multi-level construct). The area overhead of concurrent checking schemes reported in the literature is mostly for the augmented circuit, and in particular for PLA implementations [2, pages 600-608]. The cost of the checker alone is frequently unstated. The cost of checkers can be bigger than the original circuits themselves, depending on the functionality, size and implementation. A common estimation technique used for an augmented circuit with PLA implementation is to use a cost function, which is a function of the number of inputs, outputs and product terms [53, 61, 64, 68].

In Chapter 5, we showed that a checker consists of two parts, the code generator and the comparator, where the code generator regenerates the coding bits in real

time and the comparator compares the coding bits and the check bits carried by the circuit function. Checkers are usually implemented by standard gates. In this case, it is more difficult to compare the cost of a checker in a standard gate implementation to that of the original circuit function realized in a PLA, because of the different implementations. In [64], it has been shown that PLA implementations can be used for the augmented circuit and the checker. The similarity of implementation for the two parts eases the area estimation of the design, but often introduces higher silicon overhead to the checker. Recently, some CAD tools, such as OASIS[3], have provided area cost information for designs, thereby reducing the amount of work involved in cost measurements. However, to obtain such information, a complete design using the CAD tools is required, and the estimation is still dependent on the availability of the tool.

We propose a new method below to estimate the area cost of a general design and overcome the problems discussed above. We present an example of estimation using one of our proposed schemes given in the previous chapter.

6.1.1 Transistor Pair Layout Estimation

Transistor pair layout (TPL) estimation is a silicon cost estimation method for general circuitry, using the area of a transistor pair (a pair of N and P transistors, assuming that CMOS technology is used in the designs) as the measurement. In other words, we use a grid line system, where the width of the transistor pair is used as the vertical grid separation, and the height for the horizontal ones, to measure a design. In this dessertation, we limite our discussion to the fully complemented CMOS technology, but the TPL can be generalized for the others.

We classify circuit implementations into three categories: standard cell realization, modular implementation, and miscellaneous structures. The standard cell re-

[3]Open Architecture Silicon Implementation Software by MCNC.

alization encompasses circuit functions implemented in standard cells; the modular implementation includes structural designs, such as PLAs and ROMs etc. the miscellaneous structures can be anything which is excluded in the first two categories, such as data bus, routing channels, I/O pins and so on.

6.1.1.1 Area estimation for standard cell realization

The area cost of any design at gate level can be easily estimated by counting the number of transistors in it. The general formula can be given as follows.

$$\begin{aligned} A_{gate} &= NO_{gatetype1} \times TP_{gatetype1} + NO_{gatetype2} \times TP_{gatetype2} \\ &\quad + \ldots + NO_{gatetypeg} \times TP_{gatetypeg}, \end{aligned} \tag{6.1}$$

where A_{gate} is the total area cost of a design using a standard gate implementation, $NO_{gatetypei}$ and $TP_{gatetypei}$ are the number of gates and the number of transistor pairs in *gatetypei*, respectively. For example, the standard CMOS implementation of a NAND gate consists of two pairs of transistors. Then, we say that the area cost of the NAND is 2 TPL.

6.1.1.2 Area estimation for modular structures

Let the height and the width of a transistor pair be H_{pair} and W_{pair} respectively. The area cost of a modular structure can be estimated in terms of the area of the transistor pair. If the height and width of the module are H and W, respectively, then the area of the unit is computed by

$$A_{module} = (H/H_{pair}) \times (W/W_{pair}) = (H \times W)/(H_{pair} \times W_{pair}). \tag{6.2}$$

To clarify the equation we show how the area cost of a generic PLA is estimated in the TPL method. This method can be easily adapted to other modular structures.

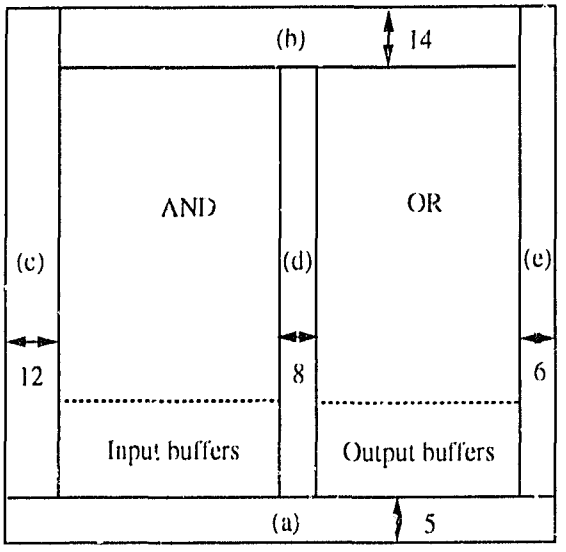


Figure 6.1: A generic PLA layout schematic

Figure 6.1 depicts a generic structure of a pseudo-nMOS PLA layout in CADENCE[4]. The areas (a) and (b) are for the substrate extension and output lines' pull-up transistors, respectively; the areas (c), (d) and (e) are for the product lines' pull-up transistors, the space separating the AND and the OR arrays, and the areas of substrate extension respectively. Each input variable corresponds to two bit lines (vertically across the input buffer of the AND array). Each output variable is represented by one output line (running vertically across the OR array). The product lines run horizontally, crossing both AND and OR arrays. Thus, the heights of (a) and (b), named $H_{boundary}$, add to the height of the PLA, H_{pla}. The widths of (c), (d) and (e), called $W_{boundary}$, contribute to the module width, W_{pla}. We assume that: (1) the width of the area for a pair of bit lines and their cross points are determined by the width of the input buffers; (2) the height of the AND array depends on the number of product terms in the given function; and (3) the width of the area for output lines and their cross point transistors is the same as the distance between

[4]A VLSI design tool by Cadence Design System, Inc.

two input or output buffers. Let us use the height and the width of a transister pair in an input buffer of the PLA as the H_{pair} and W_{pair}, respectively. Let $H_{product}$ be the height of a product line including the spacing between, and I, O, P be the number of inputs, outputs and product terms, respectively. Using equation 6.2, the area cost of the PLA in term of the transistor pairs is as follows:

$$A_{pla} = H_{pla} \times W_{pla} \tag{6.3}$$

where

$$W_{pla} = \frac{W}{W_{pair}} = \frac{W_{boundary} + W_{pair} \times (I + O)}{W_{pair}} \tag{6.4}$$

$$= \frac{W_{boundary}}{W_{pair}} + I + O; \tag{6.5}$$

$$H_{pla} = \frac{H}{H_{pair}} = \frac{2 \times H_{pair} + H_{boundary} + H_{product} \times P}{H_{pair}} \tag{6.6}$$

$$= \frac{2 \times H_{pair} + H_{boundary}}{H_{pair}} + \frac{H_{product}}{H_{pair}} \times P. \tag{6.7}$$

Equations 6.3, 6.5 and 6.7 can be simplified when a particular CAD tool and fabrication technology are chosen. For an example, we use CADENCE and a CMOS technology (cmos3dlm[5]), a pair of N and P transistors consumes 10 (width) $\times$ 19 (height) square units of silicon, i.e. $W_{pair} = 10$ and $H_{pair} = 19$. The height of a product line and the corresponding spacing, i.e. $H_{product}$, is 6 units. The heights of areas (a) and (b) in the generic PLA shown in Figure 6.1 are 5 and 14 units, respectively. The widths of the areas (c), (d) and (e) are 12, 8, and 6. Thus, the $H_{boundary} = 19$ and $W_{boundary} = 26$. Replacing the parameters in equations 6.5 and 6.7, we have

$$W_{pla} = \frac{26}{10} + I + O$$

[5]A 3-micron technology supplied by the Canadian Microelectronics Corporation

$$= 2.6 + I + O \tag{6.8}$$

$$H_{pla} = \frac{2 \times 19 + 19}{19} + \frac{6}{19} \times P$$

$$= 3 + 0.32P \tag{6.9}$$

For example, if a PLA has 2 inputs, 2 outputs and 4 product terms, then, we have $W_{pla} = 6.6$ and $H_{pla} = 4.28$. Thus, the estimated silicon cost is approximately 28 TPL.

6.1.1.3 Area estimation for miscellaneous structures

The TPL method can be applied easily to estimate the cost of any structures excluded from the first two categories. For example, if a routing channel occupies 5×16 square units of silicon, then its area cost is estimated as $(5 \times 16)/(H_{pair} \times W_{pair})$ TPL.

6.1.1.4 Summary

In summary, the TPL method provides a uniform way to estimate area cost of general designs. Although the results of TPL estimation are not as accurate as exact layout measurement, we believe it is more appropriate than a simple transistor count for the following reasons.

- The TPL method is based on logic level design, thus layout design, which is needed in the exact layout measures, is not required for the estimation;
- This method is applicable to general circuit structures unlike the conventional transistor count method, which is restricted to standard gate implementation;
- For a given implementation structure, estimation equations can always be found by using the area of a transistor pair as a measurement unit in a CAD tool. These equations then can be used by others who may not share the same

design tool. Thus, relatively fair comparison can be made among designs with the same measurement;

- This method can be easily adopted to support other technologies, such as NMOS technology, where the measurement unit is the area of a single NMOS transistor.

6.1.2 An Area Estimation Example

We present the area overhead estimation of the design example given in section 5.4 by using the TPL method introduced in the previous subsection. The example circuit, *apex4*, has 9 inputs, 19 outputs and 436 product terms after *espresso* minimization .

(1) We first calculate the area cost of the original *apex4* PLA. Using equations 6.8 and 6.9, we have $H_{pla} = 142.58$, $W_{pla} = 30.6$, and the silicon cost of the original PLA is about 4361 transistor pairs.

(2) We count the area overhead of the PLA with the output partitioning scheme. The cost function is divided into two parts: the augmented circuit with LCAR encoding and the rest of the testing circuitry in standard cell implementation.

(a) **Augmented circuit.** It is assumed that *apex4* is designed using the output partitioning with $k = 2$ (partitioning outputs into two segments) and $r = 2 \times k = 4$, i.e. each segment is encoded by a 2-bit LCAR codes. The augmented PLA has 9 inputs, 23 outputs and 450 product terms after minimization. Using equations 6.8 and 6.9, we have $W_{pla} = 34.6$ and $H_{pla} = 147$. The area cost of the encoded PLA is 5086 transistor pairs. Hence, the area overhead with the LCAR encoding is (5086 − 4361)/4361 = 16.6%.

(b) **Other circuitry.** The basic elements used to construct the bit sliced

cells (shown in Figures 5.10, 5.11 and Figures 5.12) of *apex4* in section 5.4 are NANDs, EXORs, AND and D flip-flops. Table 6.1 lists the cost of each of the elements in number of transistor pairs. It is assumed that the standard 2 pair transistor implementation is used for the NAND gates. The circuit diagrams of the EXOR [43] and the D flip-flop are shown in Figure 6.2 (b) and Figure 6.2 (a) respectively. The AND gates consist of 3 pairs of transistors (2 for the NAND function and 1 for the inverter).

Column 1 of table 6.2 is a list of bit-sliced cells used in the *apex4* design. Columns 2, 3, 4 and 5 list the number of basic elements required by bit sliced cells, and the right-most column is the corresponding silicon cost.

Element Name	No. of Transistor Pairs
NAND	2
EXOR	2
AND	3
D flip-flop	10

Table 6.1: Cost of the basic elements

Figures 5.8 5.9 and 5.13 of Chapter 5 depict the functional modules used in *apex4*, the MICA, PRPG and comparator, respectively. The area costs of these modules can be found in Table 6.3. Columns 2, 3 and 4 present the number of bit-sliced cells required by the modules. Column 5 is the number of connectors needed, and column 6 shows the total number of transistor pairs used in each type of cell. The total cost of each module is given in column 7. The bottom-right entry is the total cost.

Using the information provided in Tables 6.1, 6.2 and 6.3, Table 6.4 summarizes

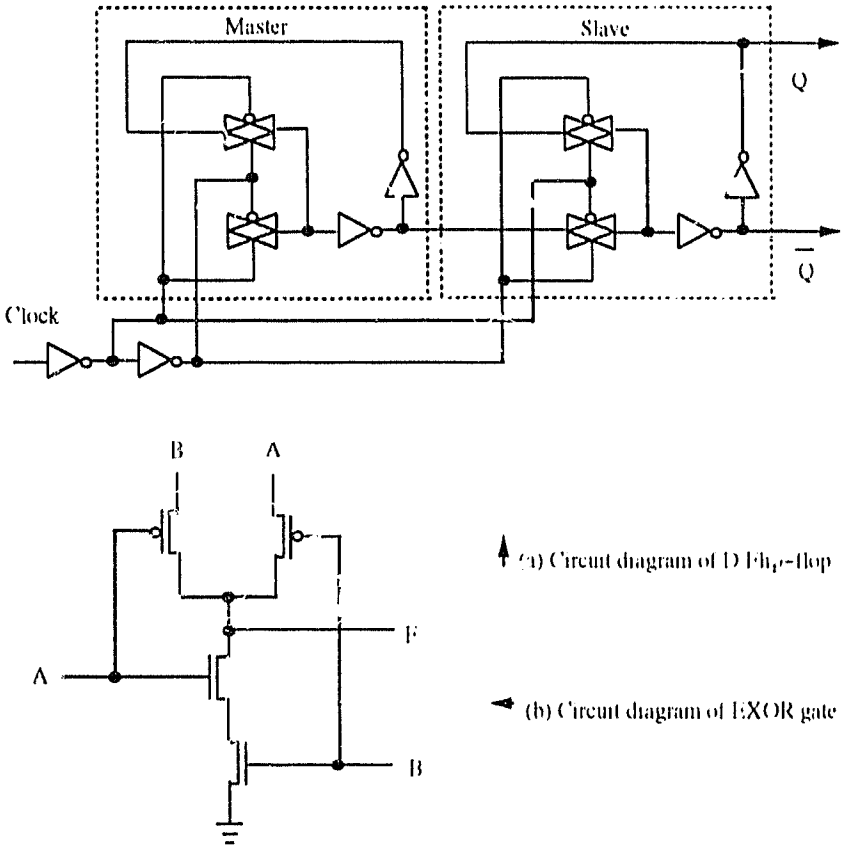


Figure 6.2: Circuit diagrams of basic bit sliced cells

the costs of *apex4* with different testability designs. Data on the rows are explained as follows:

(a) Row 1 shows the cost of the original circuit, 4361 transistor pairs;

(b) The cost of the original *apex4* circuit with concurrent checking (LCAR code) alone is given in row 2. The area overhead in comparison to the original circuit is the sum of:

- the number of transistor pairs for the coding part of circuit, (5086 - 4361),
- the cost of the connectors and the comparator, (18), and

1 Bit-sliced cell	2 No. of NANDs	3 No. of EXORs	4 No. ANDs	5 No. of Ds	6 No. Tran. Pairs
MICA90/LCAR90	3	2	-	1	20
MICA150/LCAR150	3	3	-	1	22
MICA90/Buffer	3	2	-	1	21
MICA90/Shift	4	2	-	1	22
MICA150/Shift	4	3	-	1	24
MICA90/	2	2	-	1	18
PRPG90/Shift	4	1	-	1	20
PRPG150/Shift	4	2	-	1	22
Average cell cost	3.4	-	2.1	1	21
Connector	-	-	2	-	6
2-input Module (comparator)	6	-		-	6

Table 6.2: Cost of the basic cells

- the cost in bit-sliced cells for concurrent checking is (21-4) × (23+4), where 21 is the average cost of bit-sliced cells (see Table 6.2), (21-4) is the cost of a bit-sliced cell without supporting the BIST and SCAN operations, and (23+4) is the total number of bit-sliced cells used. Thus, the area overhead with respect to the original *apex4* is 1202/4361=27.6%.

(c) Row 3 shows the cost of the original circuit with the BIST and SCAN test. It costs 4963 transistor pairs, which is the sum of the original cost, 4361, plus the cost of the other resources for the MICA and PRPG, (476+180), and minus the cost of the NAND gates for concurrent checking in bit-sliced cells, (2 × (23+4)). The area overhead over the original *apex4* is 13.8%.

(d) The cost of our proposed scheme (i.e. with concurrent checking plus the BIST and SCAN tests), shown in row 4, is 5790 TPL, which is the sum

1 Bit-sliced cell	2 MICA	3 PRPG	4 Comparator	5 Connectors	6 No. of Tran. Pairs	7 Tran. Pairs/module
MICA90/LCAR90	2	-	-	-	40	
MICA150/LCAR150	2	-	-	-	44	
MICA90/Buffer	4	-	-	-	80	176
MICA90/Shift	9	-	-	-	198	
MICA150/Shift	1	-	-	-	24	
MICA90/	5	-	-	-	90	
PRPG90/Shift	-	8	-		160	184
PRPG150/Shift	-	1	-	-	20	
Connector	-	-	-	5	30	30
2-input module (comparator)	-	-	3	-	18	18
Total Tran. Pairs	-		-	-		704

Table 6.3: Cost of the modules

of the cost of the original circuit with coding, 5086, plus the cost of the rest of the testing circuitry, 704 (see Table 6.3). The total area overhead is 32.8% over the original.

It can be seen from Table 6.4 that:

(a) With concurrent checking alone, 27.6% area overhead is required for *apex4*, while using our merging scheme, only an additional 3% increase is needed to provide both on-line and off-line testability (see row 4). Note also that we used cyclic codes for concurrent checking, and thus a small checker, instead of Berger coding and a corresponding larger checker. More discussion on this can be found in Chapter7.

(b) With BIST and SCAN testing alone, 13.8% area overhead is induced by adding BIST and SCAN circuitry. To use our scheme, an extra 19%

	Desgins	No. of Transistor Pairs	Area Overhead (%)
1	Original	4361	0
2	Concurrent checking	4361 + 1202	27.6
3	BIST+SCAN	4963	13.8
4	Concurrent checking +BIST +SCAN	5086 + 704	32.8

Table 6.4: Comparison of different schemes

(32.8 13.8) silicon is needed to add concurrent checking to it, as opposed to the 27.6% required by having concurrent checking alone.

In summary, we have shown through a design example how the TPL method works. It provides an efficient way to estimate silicon cost of designs. It is based on gate level logic designs but also permits lower level descriptions such as transistors. This method can be used at different levels of a design hierarchy, and comparisons among designs can be made without depending on the availability of VLSI design tools.

6.2 Costs on Benchmarks

In the previous section, we introduce a set of formulas, which can be used to estimate the silicon cost of general circuitry and provide a case study example of area overhead estimation. This section presents various cost measures of our proposed testing scheme on Berkeley benchmark circuits. The cost measures discussed in the following subsections are area overhead, and testing time. Fault coverage and error-detecting issues are presented separately in Chapter 7. In this exposition, we use a number of benchmarks to illustrate the *feasibility* of the scheme, but we do not rely on them as proof of its effectiveness in all cases. When dealing with concurrent checking techniques, one must be aware that the encoding is very dependent on the function and one can always find counter examples for which a particular scheme

does not work as well as others in area overhead.

6.2.1 Area Overhead

In our experiments on area overhead, we look only at designs implemented either with a PLA (two-level construct) or a standard gate realization (multi level construct). In the area overhead estimation discussed in this section, we assume that a PLA implementation is used for the main part of a circuit, and a standard gate realization is employed for the rest of the test circuitry (i.e. the BIST, SCAN resources and the checker for concurrent checking). The justifications behind this assumption are twofold: first of all, most of the work reported in the literature in concurrent checking is for PLA implementations. Secondly, the circuit description of a PLA contains only the functional information of the circuit, such as the number of inputs, outputs and product terms, while the corresponding implementation structure is implied and is the same for all when a layout format is chosen. In contrast, a gate realization may greatly vary from one designer or CAD package to another, and the area cost usually depends on many other factors such as cell libraries and routing strategies, etc.

The discussion of area overhead is divided into two parts: the main circuit and the rest of the test circuitry.

6.2.1.1 Area overhead of the main circuits

Designing the augmented circuit of a given function involves obtaining the fully specified (i.e. no "don't-cares") form, adding code bits to each output vector, minimizing the encoded function and implementing it as the augmented circuit, as detailed in section 5.1. Note that this process is the same for PLA and gate realizations. The area overhead is investigated from four different perspectives.

(1) In the previous chapter, we show that length-r cyclic codes can be used to replace other error-detecting codes for concurrent checking, and we choose to implement them with LCARs. However, for a given length-r code, there are many choices of LCARs. Table 6.5 shows the area overhead of the benchmark circuits encoded by LCAR codes generated by all length-2 and -3 primitive LCARs. The exact silicon cost and area overhead is examined by using CADENCE's PLA generator since either the TPL or the product term counting methods may not be able to reflect the difference among different LCARs of the same length. Column 3, Amin, is the exact area cost of the original benchmarks after *espresso* minimization. Columns 4 and 5 present the area overhead of LCAR codes generated by length-2 LCARs, "10" and "01", respectively. Similarly, columns 6, 7 and 8 are the area overhead of LCAR codes from length-3 primitive LCARs, "100", "001" and "110".

Observations:

(a) When the number of coding bits increases, the overhead usually increases.

(b) Area overhead of cyclic codes generated by different LCAR machines of the same length may vary significantly. For example, length-3 LCAR, "001", introduces only 23.8% overhead for circuit *cm1*, while LCARs, "100" and "110" cause 80.5% area increase.

(c) The area overhead induced by encoding depends heavily on the nature of an implemented function. The best LCAR, introducing the minimum area overhead to a circuit, can be found by simulation.

(2) Table 6.6 shows the area overhead estimated in product term counting method [61, 62] on the same set of benchmarks. Comparisons are made among 2-bit LCAR codes, residue codes of the same length, and Berger codes, which usually require more checking bits. The area overhead is defined as the percentage of the number of product terms increased after encoding, over the number of product terms of the original circuit. Table 6.6 provides the results of the experiments. Columns 3, 4 and 5 list the number of inputs, outputs and product terms of the circuits respectively. Column 6 is the number of product terms after the initial minimization. Columns 7, 9, and 11 give the number of product terms after adding a 2-bit cyclic code based on LCAR=01, Mod3 and Berger codes respectively, and minimizing. Columns 8, 10 and 12 are the percentage increase comparing with column 6.

(3) Table 6.7 presents the area overhead estimation in TPL with LCAR, residue and Berger codes. The area overhead of an augmented PLA in the TPL method is defined as the number of transistor pairs increased after adding the checking bits. The table shows the results of the estimation. Columns 3 and 4 list the number of inputs and outputs of the circuits respectively. Column 5 is the number of transistor pairs after the minimization. Columns 6, 8, and 10 give the number of transistor pairs after adding a 2-bit cyclic code based on LCAR=01, Mod3 and Berger codes respectively, and minimizing. Columns 7, 9 and 11 are the percentage increase comparing with column 5.

Observations from (2) and (3):

(a) It can be seen from Tables 6.6 and 6.7 that the area overhead of an augmented CUT is usually better with LCAR codes than

with residue codes, and always better than Berger encoding (if the number of circuit outputs is greater than one).

(b) The area overhead in Table 6.7 are higher than those in Table 6.6 in most cases. Table 6.7 not only reflects the changes in the number of product terms after encoding, but also shows the difference in the number of checking bits introduced. For example, *dk15* of Table 6.6 has the same number of product terms (17) after adding the LCAR, mod3 and Berger codes. Thus, the three error-detecting codes have the same area overhead. It is obvious that the area overhead for Berger code is underestimated since the 5-output circuit requires a 3-bit Berger code but only 2-bit LCAR and mod3 codes. The data shown in Table 6.7 also shows the effects of the increased number of product terms or checking bits on the circuits of different sizes. Thus, the TPL estimation is superior to the product term counting approach for area estimation.

(4) To make comparisons with the area overhead obtained from the two level (PLA) realization above, experiments using multiple level gate implementations were also pursued. The area cost is evaluated by means of OASIS, in particular, using the program *decaf*. *Decaf* takes an *espresso* form of circuit description, and produces the corresponding multiple level gate realization automatically. It provides two sets of information regarding silicon cost: the types of gates and number of gates used, and the exact area cost in a desig...

Table 6.8 shows the area cost of some benchmark circuits. Columns 2, 3, 4 and 5 are the number of inputs, outputs, total number of gates, and silicon cost of the original circuit, respectively. Columns 6 and 8 list

the number of gates and area cost after adding 2-bit LCAR codes, and columns 7 and 9 are the percentage increase of the augmented circuits over the original ones. Similarly, columns 10 to 13 are for Mod-3 codes.

Observations and discussions from (4):

The percentage increase of area overhead in multiple level implementation is bigger than those of PLA realization. Mod-3 codes give lower area overhead than LCARs in terms of number of gates used, for this set of benchmark circuits. This is not necessarily true in general, and needs to be further examined. The important point is that the numbers are quite close. When exact silicon cost is concerned, the circuits with LCAR encoding show some better results. The reason behind this is that OASIS cell library contains gates with various sizes. For example, both a 2-input and 6-input gate are counted as one gate, but the size of their geometry layout differ significantly. It should also be noted that the data in Table 6.8 are design-tool-dependent. They rely on the design of the cell libraries, the placement and routing algorithms used, etc.

For this set of benchmarks, Mod-3 codes seem to introduce lower area overhead than LCARs. However, when both on-line and off-line testabilities are required, LCAR codes are more cost effective because of the merging of hardware resources in the two test modes.

6.2.1.2 Area overhead of the rest of test circuitry

In the previous section, we have shown how the TPL method can be used to estimate area cost of designs. In the *apex4* design, our scheme provides both on-line and off-line testability with significant savings in silicon when compared with incorporating concurrent checking, and BIST and SCAN separately. For the given example, only an extra 3% and 19% increase is required if the concurrent checking and off-line

testing are associated with a design in the first place, respectively (see Table 6.4), as opposed to 27.6% and 13.6%. If Berger or other conventional error-detecting codes are used, the cost of the code generator (which is part of the checker) would add significant overhead to the cost. It may make the total silicon cost exceed that of the original. The main saving of our scheme is from the LCAR-based code generator, which is inexpensive in implementation and can be shared with the BIST and SCAN resources.

Table 6.9 summarizes the area overhead of the testing circuitry (excluding the augmented circuit) for our proposed schemes in TPL estimation. We assume that every input and output of a design corresponds to a bit-sliced BIST cell, i.e. either connecting to a PRPG and a data compactor or a scan register. This is the most expensive case when hardware overhead is concerned, as opposed to partial scan or partitioning techniques for the CUT. Column 2 shows the number of bit-sliced cells required for each scheme. Column 3 is the corresponding number of transistor pairs, where factor 21 is the average cost of a bit-sliced cell from Table 6.2, and the $P(c)$ is the cost of the primary scheme defined as follows:

$$P(c) = \quad m + 2r \quad if\ n + (m + r) \leq m + 2r \tag{6.10}$$

$$P(c) = \quad n + m + r \quad if\ n + (m + r) > m + 2r. \tag{6.11}$$

where the $(m + 2r)$ in the *if* statements is the number of bit-sliced cells required in a design ($(m + r)$-bit for the output buffer and r-bit for the LCAR compactor). There are two cases here: (1) there is an equal or smaller number of bit-sliced cells in the BIST resources than the $(m + 2r)$ cells for the concurrent checking; (2) there are more bit-sliced cells in the BIST resource. For (1) as shown in equation 6.10, there are $(m + n + r)$ bit-sliced BIST cells, where m, n and r are the number of input, output and coding bits of the design. These cells can be modified for concurrent checking. For (2), since $(m+n+r) < (m+2r)$, an additional $(m+2r)-(m+n+r)$-bit sliced cells are needed when concurrent checking is introduced.

Figure 6.3 shows the logic diagram of a $4-to-1$ multiplexor. In general, a multiplexor of s control inputs and 2^s data inputs forms a 2^s-to-1 multiplexor. Assume that an i-input NAND gate requires i transistor pairs. Since there are 2^s NANDs at the first level and each of them has $(s+1)$ inputs, the number of transistor pairs at the first level is $2^s \times (s+1)$. Similarly, 2^s transistor pairs are needed for the second level NAND.

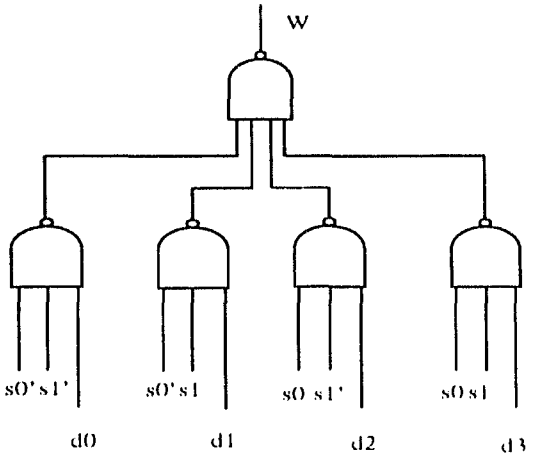


Figure 6.3: Logic diagram of a $4-to-1$ multiplexor

Moreover, each of the s controls needs two inverters (2 transistor pairs) to generate the true and complementary signals internally. Thus, the cost of a 2^s to 1 multiplexor is

$$2^s \times (s+1) + 2^s + 2s = 2^s \times (s+2) + 2s.$$

In the area overhead estimation above, we are aware of ignoring the routing cost, which can be easily incorporated into of the TPL method when a particular design is considered (see Section 6.1.1).

6.2.2 Performance and Testing Time

The testing time of the proposed scheme varies in the three modes of operation.

NORMAL/concurrent checking. In chapter 5, we defined two measurements for the performance of concurrent checking: *concurrent checking latency* and *concurrent checking interval.* The former is defined as the time delay to signal a fault in real time (defined differently from [48]); while the latter is the number of consecutive output words between any two checked output words, plus one.
In Table 6.10, m is the number of outputs of a circuit under test, and k is the number of partitions over the output bits. It can be seen that the primary scheme has the largest concurrent checking latency and interval. The partitioning scheme improves the two parameters by a factor of $1/k$. Multiplexing achieves 100% concurrent checking (i.e. checking every output word) with m clock latency. Finally, partitioning and multiplexing imposes the smallest checking latency, $\lceil \frac{m}{k} \rceil$, and the interval, 1. The best known merging scheme is by Saluja *et al* [48]. The concurrent checking interval of its primary scheme is 1 and the checking latency can be 2^n, where n is the number of CUT inputs.

BIST. The signature analysis, used in this case, can be exhaustive or non-exhaustive. In the case of exhaustive testing, the PRPG goes through all $2^n - 1$ possible patterns and a signature is obtained at the end of the test, where n is the number of inputs of the CUT. The testing time is thus $2^n - 1 + t = 2^n$, where t is the number of clock cycles required to initialize the BIST mode. If a system is designed to comply with the IEEE Std. 1149.1, then $t = l + 1$ clocks are needed, where l is the number of clock cycles to shift the BIST mode instruction into the instruction register of the boundary scan controller, and 1 is the number of clock cycles required to switch the system to the BIST mode. The performance of off-line BIST is unaffected by the new merged scheme.

SCAN. A simple scan operation requires t clock cycles to set up the testing mode, $2 \times L$ cycles to shift-in and out the scan data, and 1 cycle to exercise the CUT for a combinational logic, where L is the length of the scan path. A sequential

circuit requires the same set-up time, but $1 \times L$ cycles to shift in and out data, and 2 clock cycles to exercise the circuit, one for the initialization and one for evaluation. Again, this performance is unaffected by the new merged scheme.

6.2.3 Design Complexity

One solution to the design complexity of introducing testability is to use structured approaches and to automate the design process. Some VLSI CAD tools have included various testability features and greatly reduced the design complexity. The proposed testing schemes in this dissertation are all structured approaches. The encoding process adds complexity to the design, but the procedure is automated by using our program *ccmini* (the manual page can be found in Appendix 2).

6.2.4 Pin Count

We claim that our proposed scheme supports the IEEE Std. 1149.1. The minimum number of pins required by the standard is four. They are: *the test clock input(TCK)*, *the test mode select input(TMS)*, *the test data input(TDI)*, and *the test data output(TDO)*. With our scheme, an extra pin can be introduced to signal a fault in concurrent checking.

6.3 Summary

In this chapter, we present a new area estimation method, Transistor Pair Layout (TPL). The main goal of the TPL method is to provide a uniform method of area estimation for general circuit implementation structures, without going down to layout level designs, and allowing design comparisons to be made in design houses that do not have access to the same CAD tools. The TPL method employs the area of a transistor pair as a measurement unit and relies on designs mainly at the logic

gate level. The estimation accuracy lies between exact layout and conventional transistor counting. An area estimation example using the TPL method for our proposed scheme is provided.

We present various cost measures of our schem on Berkeley benchmark circuits. The area overhead is examined for the augmented circuit functions and the rest of the testing circuitry. We show the exact area cost of the cyclic codes generated by different finite state machines of the same length. The results are function-dependent. Overall, the area overhead of cyclic codes in concurrent checking is comparable to residue codes of the same length, and more cost effective than Berger codes. For the rest of the testing circuitry, significant savings can be obtained by merging the on-line and off-line resources.

The *testing time* is evaluated for concurrent checking and off-line test separately. In the former, our proposed primary scheme achieves m clock concurrent checking latency, as opposed to the 2^n in [48], where m and n are the number of input and output of the CUT, respectively. For the latter, the performance is not affected by merging. Better performance can be obtained by using our modified structures. The best achievement is having every output word checked with (m/k) checking latency.

Fault coverage of cyclic codes is another main concern of this research. In Chapter 7, we formally introduce the fault models, error coverage analysis and present fault simulation results on benchmark circuits.

1	2	3	4	5	6	7	8
Type	Name	Amin (μm^2)	LCAR=10 %	LCAR=01 %	LCAR=100 %	LCAR=001 %	LCAR=110 %
C	rd84	5,221,192	7	7	11.2	11.2	11.2
O	9sym	1,770,732	8	8	8	8	8
M	con1	270,000	65.5	13	80.5	23.8	80.5
B	misex1	464,400	11	11	12.2	12.2	12.2
I	misex3	260,020,842	-	-	-	-	-
N	bw	1,118,305	5.3	5.3	10.2	10.2	10.2
A	f2	222,040	18.9	18.9	32.8	27.4	26.7
T	rd53	582,627	10.8	10.8	12.5	12.5	12.5
I	rd73	2,273,658	13.1	16	12.8	12.8	12.8
O	sao2	1,446,280	10.4	10.4	18.5	18.5	16.4
N	5xp1	213,160	7.5	7.5	17.5	17.5	17.5
S	bbtas	306,033	2.5	0	0	0	0
E	dk14	520,359	5.7	10.1	6	6	6
Q	dk15	306,520	0	0	0	0	0
U	dk16	2,057,580	0	0	0	0	0
E	dk17	404,096	0	0	0	0	0
N	dk27	237,969	0	0	0	0	0
T	dk512	594,216	0	0	0	0	0
I	donfile	937,480	0	0	0	0	0
A	lion9	300,105	0	0	0	0	0
L	modulo12	541,200	0	0	0	0	0
	shiftreg	210,740	0	0	5.3	5.3	5.3

Table 6.5: Comparison of area overhead using different LCAR codes of the same length

1	2	3	4	5	6	7	8	9	10	11	12
		No.	No.	No.	No.	LCAR = 01		Mod3		Berger	
Type	Name	input	output	product	pmin	product	%	product	%	product	%
C	rd84	8	4	256	255	255	0	255	0	256	0.4
O	9sym	9	1	87	85	84	-1.2	84	-1.2	156	79.3
M	con1	7	2	9	9	9	0	17	88.9	22	144.4
B	misex1	8	7	32	12	13	8.3	13	8.3	18	50
I	misex3	14	14	1848	690	1044	51.3	1331	92.9	1937	180.7
N	bw	5	28	87	22	22	0	22	0	24	9.1
A	f2	4	4	12	8	9	12.5	10	25	15	87.5
T	rd53	5	3	32	31	31	0	31	0	32	3.2
I	rd73	7	3	141	127	129	1.6	127	0	128	0.8
O	sao2	10	4	58	58	58	0	58	0	60	3.4
N	5xp1	7	10	75	64	97	51.2	127	98.4	115	79.9
S	bbtas	2	2	24	16	16	0	17	6.3	17	6.3
E	dk14	3	5	56	25	28	12	28	12	27	8
Q	dk15	3	5	32	17	17	0	17	0	17	0
U	dk16	2	3	108	55	55	0	55	0	58	5.5
E	dk17	2	3	32	20	20	0	20	0	21	5
N	dk27	1	2	14	10	10	0	10	0	10	0
T	dk512	1	3	30	21	21	0	21	0	22	4.8
I	donfile	2	1	96	24	24	0	24	0	24	0
A	lion9	2	1	25	10	10	0	10	0	10	0
L	modulo12	1	1	24	24	24	0	24	0	24	0
	shiftreg	1	1	16	9	9	0	9	0	10	11.1

Table 6.6: Area overhead in terms of number of product terms

1	2	3	4	5	6	7	8	9	10	11
		No.	No.	No.	LCAR = 01		Mod3		Berger	
Type	Name	input	output	TP(min)	No. T.P.	%	No. T.P.	%	No. T.P	%
C	rd84	8	4	1235	1404	13.7	1404	13.7	1495	21.1
O	9sym	9	1	381	436	14.4	436	14.4	720	89
M	con1	7	2	68	80	17.6	115	69.1	137	101.5
B	misex1	8	7	120	140	16.7	140	16.7	180	50
I	misex3	14	14	6848	10989	60.4	13983	104	21550	214.7
N	bw	5	28	357	378	5.9	378	5.9	433	21.3
A	f2	4	4	59	74	25.4	78	32.2	106	79.7
T	rd53	5	3	137	162	18.2	163	19	137	21.9
I	rd73	7	3	550	646	17.4	637	15.8	641	16.5
O	sao2	10	4	357	401	12.3	401	12.3	435	21.8
N	5xp1	7	10	460	735	60	943	105	939	104
S	bbtas	2	2	54	70	29.6	73	35.1	73	35.1
E	dk14	3	5	117	151	29.1	151	29.1	158	35
Q	dk15	3	5	89	106	19.1	106	19.1	114	28.1
U	dk16	2	3	157	197	25.5	198	26.1	206	31.2
E	dk17	2	3	71	90	26.8	90	29.8	93	31
N	dk27	1	2	35	47	34.3	47	34.3	47	34.3
T	dk512	1	3	64	84	31.2	84	31.2	86	34.3
I	donfile	2	1	60	81	35	81	35	70	16.7
A	lion9	2	1	35	47	34.3	47	34.3	40	14.3
L	modulo12	1	1	49	70	42.9	70	42.9	59	20.4
	shiftreg	1	1	26	26	0	26	0	28	7.7

Table 6.7: Area overhead in TPL estimation

1	2	3	4	5	6	7	8	9	10	11	12	13
	No.	No.	The Original		2-bit LCAR(01)				Mod3			
Name	input	output	No.Gate	Area	No.Gate	%	Area	%	Gate	%	Area	%
con1	7	2	15	21808	22	46.7	29696	36.2	30	100	46400	112.8
bw	5	28	138	206480	159	15.2	234320	13.5	140	1.4	205088	-0.7
f2	4	4	20	25984	21	5	29696	14.3	20	0	33408	28.6
rd53	5	3	40	56144	60	50	86768	54.5	42	5	64032	14
rd73	7	3	94	141520	158	68.1	236176	66.9	133	41.5	194880	37.7
rd84	8	4	178	270976	278	56.2	408784	50.9	239	34.4	363776	34.2
sao2	10	4	123	182352	185	50.4	272832	49.6	151	22.8	232464	27.8
5xp1	7	10	105	154048	178	69.5	262624	70.5	156	47.2	236176	53.3

Table 6.8: Area overhead for standard gate realization

Schemes	No. of bit-sliced cells	No. of T.P.
Primary	$P(c)$	$21 \times P(c)$
Partitioning	$P(c)$	$21 \times P(c)$
Mutiplexing	$P(c) \times m$ + m to 1 Mux.	$21 \times P(c) \times m$ + m to 1 Mux.
Partitioning & Multiplexing	$P(c) \times \lceil \frac{m}{k} \rceil$ + $\lceil \frac{m}{k} \rceil$-to-1 Mux.	$21 \times P(c) \times \lceil \frac{m}{k} \rceil$ + $\lceil \frac{m}{k} \rceil$-to-1 Mux.

Table 6.9: Area cost formulas for different testing schemes

Schemes	Concurrent Checking Latency	Concurrent Checking Interval
Primary	m	m
Partitioning	$\lceil \frac{m}{k} \rceil$	$\lceil \frac{m}{k} \rceil$
Mutiplexing	m	1
Partitioning & Multiplexing	$\lceil \frac{m}{k} \rceil$	1

Table 6.10: Impact on concurrent checking

Chapter 7

Error Characteristics

In Chapter 5 we presented a new testing scheme, which employs LFSR/LCAR based cyclic code checking as a medium to combine the concurrent checking and signature analysis in a built-in fashion, and uses bit-sliced LFSRs/LCARs as the implementation mechanism. This chapter is devoted to the error-detecting capability of LFSR/LCAR codes. The investigation is carried out on both PLAs and general circuitry, by means of simulation. In section 1, we give the preliminaries. In section 2, general error coverage of several error-detecting codes is given. In section 3, we present some simulation results on fault coverage of error-detecting codes.

7.1 Preliminaries

7.1.1 Definitions

An *error* is a result of noise in a communication system or fault effects in a defective circuit. In on-line BIST, concurrent checking techniques employ error-detecting codes as part of a function's specification. It is achieved by encoding a given *data word* with an error-detecting code, and concatenating the generated *checking bits* to the data word to form the *code word*. During normal operation, the recomputed

checking bits, called the *signature*, are obtained from the output data word and compared with the attached checking bits. If the signature differs from the checking bits, an error is detected.

An *error pattern* is an erroneous code word, which differs from the correct code word. Error patterns can be classified into the following two categories:

1. *Unidirectional error pattern*: an erroneous code word that has only $1 \rightarrow 0$ or $0 \rightarrow 1$ errors in it;

2. *Bidirectional error pattern*: an erroneous code word that contains both $1 \rightarrow 0$ and $0 \rightarrow 1$ errors.

Definition 7.1 *The behaviour* of error-detecting codes is described in terms of their *error coverage*. We define

1. *Unidirectional error coverage* is the total number of unidirectional error patterns detected by an error-detecting code over the total number of unidirectional error patterns;

2. *Bidirectional error coverage* is the total number of bidirectional error patterns detected by an error-control code over the total number of bidirectional error patterns;

3. *Total error coverage* is the total number of error patterns detected (both unidirectional and bidirectional) over the total number of error patterns.

7.1.2 Error-detecting Codes: Berger, Residue, LFSR/LCAR

In Chapter 5, we introduced LFSR/LCAR codes as an alternative to conventional error-detecting codes in concurrent checking. LFSR/LCAR codes are general error-detecting codes that can be used for both PLA-based and conventional logic applications. Three error-detecting codes, Berger [46, page 338], residue [46, page 339],

and LCAR codes, are examined in this chapter. The reasons we chose Berger and residue codes for comparison are as follows:

1. Berger codes are traditionally used in concurrent error detection for PLAs. A considerable work has been reported in the literature. Berger codes are known to have high error coverage for unidirectional errors, but the checker is very expensive[12, 31];

2. Residue codes are especially suited for error detection in arithmetic operational units. In particular, Mod 3 and Mod 7 codes have the advantage of low cost both in checking bits and encoding/decoding process [47, page 128]. Residue codes are also used in non-arithmetic applications [50].

7.1.2.1 Berger Codes

Let $(d_m d_{m-1} \cdots d_1)$ be a given data word, where $d_i \in$ *a binary field* for $(i = 1, 2, \cdots, m)$. The checking bits are formed by counting the number of 0's in the data word, and converting such number into the corresponding binary representation of r bits, where $r = \lceil \log_2(m+1) \rceil$. The entire code word is $(m + r)$ bit long. For example, if the given data word is 00111, then the corresponding checking bits are 010, since there are two 0's in the data word. Berger codes for m bit data words have $(m + 1)$ valid checking patterns. For example, data words of length 5 have a maximum of five zeros, and thus have six valid checking patterns, 000, 001 $\cdots$, 101.

7.1.2.2 Residue Codes

Let D be a positive integer represented in binary form. The residue R of D for divisor b is denoted by

$$R = D \bmod b.$$

The concatenation of D and R, (DR), is called a *code word* of the residue code with check base b. *Low-cost residue codes* have a modulus of $b = 2^r - 1$, where r is some integer greater than or equal to 2. We use term *residue codes* to refer to low-cost residue codes in this dissertation. Residue codes of r bits have $2^r - 1$ valid checking patterns.

7.1.2.3 LFSR/LCAR Cyclic Codes

LFSR/LCAR codes are separable cyclic codes based on a linear cellular automata register implementation. LFSR/LCAR codes are obtained by taking a data word as the input stream of the LFSR/LCAR. The data stream is fed into the LFSR/LCAR one bit at each clock cycle. At the end of the operation, the last state of the LFSR/LCAR is used as the checking bits. LFSR/LCAR codes of r bits have 2^r valid patterns.

7.2 Error Coverage – General Estimation

The primary emphasis of error-detecting codes has been the design of reliable communication systems. In concurrent checking, checking circuitry is part of the function specification, and is often implemented along the main circuit with shared circuitry. Thus, faults may occur and cause errors in both data word and check bits. However, in the literature, it is usually assumed that checking bits either contain unidirectional errors only (for Berger codes) or are error-free (for other error-detecting codes). In practice, bidirectional errors may also occur in PLA- and ROM-based systems, where unidirectional errors are considered most likely to occur [15, 31, 67]. For example, it is known that multiple faults in PLA can cause bidirectional errors.

Although the *equally likely* error model may be considered inadequate [23], it can be used to provide some general estimation. Under the *equally likely* error model,

Data-word length (m)	Check-bits length $r = log_2{}^m$	Berger $\frac{1}{m+1}$	Residue $\frac{1}{2^r-1}$	LCAR $\frac{1}{2^r}$
2	2	33.3	33.3	25.0
4	3	20.0	14.3	12.5
8	4	11.1	6.67	6.25
16	5	5.88	3.22	3.12
32	6	3.03	1.59	1.56
64	7	1.54	0.79	0.78
128	8	0.78	0.39	0.39
256	9	0.38	0.20	0.20

Table 7.1: Total aliasing of the three codes

the aliasing probability of an error-detecting code can be computed by $1/k$, where k is the number of valid checking patterns of the code word. Table 7.1 shows the predicated aliasing probability of Berger, residue and LCAR codes, for data word lengths $2^r, r = 1, 2, \cdots 8$, when the same number of checking bits are used.

It is shown that the fault detection ability of LFSR/LCAR codes is comparable to residue and Berger codes because:

1. a residue code of r-bits has a general aliasing probability of $1/(2^r - 1)$, while that of a LFSR/LCAR code of the same length is asymptotic to $1/2^r$. For small values of r, (as assumed in our proposed testing scheme), and with r independent of m, residue and LFSR/LCAR cyclic codes are comparable;

2. the total aliasing of LFSR/LCAR codes is around one half that of Berger codes when data word lengths are equal to or greater than 4, all patterns are considered.

In both cases, LFSR/LCAR codes preserve lower aliasing probability. The general error coverage of the codes can be obtained by $1 - p_a$, where p_a is the aliasing probability of the error-detecting codes.

7.3 Fault Coverage – Simulation Results

In our experiments on fault coverage of the three error-detecting codes, both two level (PLA) and multiple level (standard gate) circuit implementations are considered. In this section, we first introduce the simulation environment of this experiment, including the simulators, benchmark circuits, and simulation processes. Then, we present some simulation results for the two circuit structures separately.

7.3.1 Simulation Environment

7.3.1.1 Fault Simulators

Three fault simulators are involved in this investigation. They are *faultsimulator* [64], *plasim* [69] and *LogicIII(uvic)sim* [7], where the first two are PLA simulators and the last one is for multiple level gate realization.

1. *Faultsimulator* is a deductive PLA fault simulator[42]. It takes as input the *espresso*-output style of PLA description, and simulates all single faults of the stuck-at, bridging (all adjacent-pair faults which do not cause the formation of a sequential circuit), and crosspoint faults [42]. The simulation is performed based upon a gate level implementation of PLAs. For a PLA with n inputs, m outputs, and p product terms, the expected running time is $O(2^n(n+m)^2p^2)$. Details on the design and implementation of the fault simulator can be found in Appendix B of [64].

2. *Plasim* is a transistor-level pseudo-NMOS PLA simulator. It accepts an *espresso* format description of a circuit, and simulates all single faults of the stuck-at, bridging and crosspoint faults in a PLA, and multiple stuck-at faults on the primary inputs and outputs. The total number of faults simulated for a PLA with n inputs, m outputs and p product terms, is $(9n+6m+3p+4pn+5pm-5)$.

Details on the design and implementation of the fault simulator are in [69].

3. *LogicIII(uvic)sim* is a gate level simulator. It takes a LogicIII[1] description of a circuit's function, and simulates all single stuck-at faults in the multiple level gate implementation. It is assumed that the gates are fault-free, and all stuck-at faults appear on signal lines.

   The motivation for designing the *LogicIII(uvic)sim* simulator is due to the complicated translation process between LogicIII language and the LogicIII simulator. A LogicIII circuit description is first translated into *RNL* format[2], then the *RNL* file is converted into *ISCAS* circuit description[3]. The LogicIII simulator takes ISCAS format as the input, simulates all single stuck at faults, and generates some statistics. However, to trace a fault simulated by the simulator, the reverse translations, i.e. from the simulator to LogicIII description, are required. The backtracking complicates the fault simulation environment. Detailed discussion on these issues can be found in [7].

   It should also be noted that *decaf*[4] accepts *espresso* forms of circuit description, performs multiple level circuit minimization, and produces a LogicIII circuit description. This feature of OASIS benefits our experiment on fault simulation, and allows the same set of PLA benchmarks to be used in multiple level gate realization.

7.3.1.2 Simulation Processes

All three fault simulators follow the same simulation processes:

---

[1]The hardware description language of OASIS.

[2]A netlist description of logic circuits.

[3]From 1985 and 1989 ISCAS (International Symposium on Circuits and Systems) benchmark circuits.

[4]The multiple level logic minimizer of OASIS.

1. for each fault enumerated by the simulators, exhaustive input combinations are applied to the circuit, the output responses are collected, and compared with the fault-free output;

2. the output responses are classified into fault-free, unidirectional and bidirectional categories;

3. the fault coverage of the circuit is obtained by computing the mean fault coverage for all the faults in the fault set.

7.3.1.3 Benchmark Circuits

The benchmark circuits used in this experiment are a subset of the Berkeley benchmarks[5], that we used for the area cost estimation. However, there are two main restrictions on circuit sizes when fault simulation is concerned: numbers of inputs and outputs. In *faultsimulator* (the first one used), although only single faults are taken into account, the total number of faults simulated is exponential in the inputs and outputs. Moreover, for each fault, all input combinations are used to exercise the circuit. That is exponential in the number of inputs as well. These factors pose severe restrictions on the sizes of the circuits we can simulate. In addition, extra coding bits are added to the original circuit function in order to achieve concurrent checking, increasing the problems because of the additional outputs. With all the restrictions outlined above, the benchmark circuits we use in practice are limited to 4 to 5 inputs, and 3 to 12 outputs.

7.3.2 Fault Coverage for PLAs

In Chapter 5 we show that concurrent checking is achieved by employing error-detecting codes to encode a given circuit function at design stage. Checking bits are

[5]With Berkeley tools.

added to each output vector, and the augmented circuit function is then minimized and implemented by certain circuit structures. The fault coverage of LCAR codes is investigated from two aspects.

1. For a given number of checking bits, there are different LCAR machines which can produce the cyclic codes. Tables 7.2 and 7.3 show the unidirectional and bidirectional fault coverage of LCAR codes generated by all length 2 and 3 LCARs, where the data in the two tables are obtained from *faultsimulator* and *plasim*, respectively.

   We apply statistical tests, t tests, [41, page 205] to the data collected in Table 7.2, and examine how the fault coverage from the simulation matches the expected value of aliasing, $1/2^k \times 100\%$, for a k-bit LCAR code. The test results are as follows.

   (a) For the unidirectional fault coverage of 2-bit LCAR codes, the expected fault coverage is 75%. The average unidirectional fault coverage of the 18 samples is 80.36%. The standard deviation is 4.51. The t is 5.05 and $t(.005)$ is 2.9. These results indicate that with 99% confidence the unidirectional fault coverage of 2-bit LCAR codes is higher than 75% in a two-tail test [41, page 197].

   (b) For the unidirectional fault coverage of 3-bit LCAR codes, the expected fault coverage is 87.5%. The average fault coverage of the 27 samples is 86.18%. The standard deviation is 4.52. The t is -1.10 and $t(.005)$ is 2.78. Since the t falls in the rejection region of the test [41, page 194], we have insufficient evidence to reject the expected fault coverage, 87.5%.

   (c) For the bidirectional fault coverage of 2-bit LCAR codes, the expected fault coverage is the same as (a), 75%. The average fault coverage of the 18 samples is 60.55%. The standard deviation is 17.73. The t is -3.46 and $t(.005)$

is 2.9. It appears that the bidirectional fault coverage of 2-bit LCAR codes is lower than 75%.

(d) For the bidirectional fault coverage of 3-bit LCAR codes, the expected fault coverage is 87.5%. The average fault coverage of the 27 samples is 69.54%. The standard deviation is 22.45. The t is -4.16 and $t(.005)$ is 2.78. It appears that the bidirectional fault coverage of 3-bit LCAR codes is lower than 87.5%.

(e) For the total fault coverage, both unidirectional and bidirectional ones, of 2-bit LCAR codes, the expected fault coverage is again 75%. The average fault coverage of the 36 samples is 70.47%. The standard deviation is 16.24. The t is -1.68 and $t(.005)$ is about 2.7. Thus, we cannot reject the value of 75%.

(f) For the total fault coverage of 3-bit LCAR codes, the average fault coverage of the 54 samples is 77.86%. The standard deviation is 18.35. The t is -3.85 and $t(.005)$ is around 2.66. It appears that the bidirectional fault coverage of 2-bit LCAR codes is lower than 87.5%, the expected fault coverage.

From the above investigation, it is not evident that any LCAR machine is superior to the others in terms of the fault coverage. As with many other logic problems, it is function-dependent.

The all-zero bidirectional fault coverage from *plasim* is due to the facts that it simulates a pseudo-NMOS PLA, where unidirectional errors dominate as only single faults are injected. As a consequence, the bidirectional faults simulated by *plasim* are the ones on primary outputs of circuits. However, due to the limited computational power of our machines, only small circuits with 3 and 12 outputs are chosen. Thus, a very small number of bidirectional errors occur during the simulation. Miscatching of a bidirectional error may results in a 0% bidirectional error coverage.

	LCAR=01		LCAR=10		LCAR=001		LCAR=100		LCAR=110	
	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.
rd53	79.83	0.00	83.40	60.00	80.19	0.00	87.92	60.00	80.88	0.00
9by2	75.75	53.37	75.82	63.81	74.44	66.67	83.96	76.19	76.62	45.24
f2	73.13	81.25	72.29	77.08	74.99	86.67	81.60	92.86	75.20	85.40
2by3	74.83	52.86	76.27	52.22	80.90	65.49	85.05	62.67	81.68	63.92
dc1	81.78	54.92	79.92	49.00	89.76	67.22	88.25	71.05	87.11	69.55
wim	82.27	69.24	83.16	61.29	91.89	74.57	91.34	76.83	94.10	82.51
Dk27	85.42	69.02	85.62	63.00	93.43	78.44	92.72	78.66	89.45	80.40
Apla	84.92	67.62	82.65	66.97	91.29	80.25	89.63	81.47	92.13	84.86
Sqr6	84.76	74.66	84.57	74.15	88.79	85.09	91.16	81.40	92.36	80.09

Table 7.2: Fault coverage of LCAR codes (*faultsimulator*)

2. Tables 7.4 and 7.5 compare the fault coverage of Berger, Mod3, and 2 and 3-bit LCAR codes, from *faultsimulator* and *plasim*, respectively. The fault coverage for LCAR codes listed is the best one for the given lengths. It can be seen that (a) Berger codes have very high unidirectional fault coverage, from 94.41% to 100.00%, in *faultsimulator*, and 100% in *plasim*; (b) 2 bit LCAR codes have better unidirectional fault coverage than Mod3 from both *faultsimulator* and *plasim*; (c) Berger codes give better fault coverage than Mod3 and LCARs. The reason behind this is mainly because more checking bits are used for Berger encoding. For example, a 4-bit output circuit requires 3-bit Berger codes, as opposed to the 2 bits for Mod3 and 2 bit LCAR codes; the zero bidirectional fault coverage by *plasim* is caused by the same reasons discussed earlier in 1.

	LCAR=01		LCAR=10		LCAR=001		LCAR=100		LCAR=110	
	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.
rd53	75.14	0.00	80.51	0.00	75.00	0.00	83.39	0.00	73.39	0.00
2by2	63.42	0.00	65.43	0.00	64.05	0.00	73.46	0.00	67.42	0.00
f2	60.93	0.00	60.93	0.00	61.33	0.00	69.48	0.00	61.33	0.00
2by3	62.75	0.00	62.19	0.00	64.54	0.00	68.99	0.00	64.96	0.00
de1	72.74	0.00	69.20	0.00	74.94	0.00	72.43	0.00	72.37	0.00
wim	77.44	0.00	78.46	0.00	85.02	0.00	86.38	0.00	91.64	0.00
Dk27	88.05	0.00	86.49	0.00	94.45	0.00	92.44	0.00	89.65	0.00
Apla	84.09	0.00	80.44	0.00	87.26	0.00	84.20	0.00	88.73	0.00
Sqr6	86.22	0.00	89.62	0.00	81.52	0.00	84.19	0.00	87.33	0.00

Table 7.3: Fault coverage of LCAR codes (*plasim*)

7.3.3 Fault Coverage for Multiple Level Gate Implementations

The second type of circuit structure in our experiments is a multiple level gate realization. The data is listed in the *LogicIII(uvic)sim* columns of Table 7.6, where the *Total* is the mean of the total erroneous output patterns detected by 2-bit LCAR codes, from the generator LCAR=10, divided by the mean number of the total erroneous output patterns for all input combinations.

In theory, for 2-bit LCAR codes, we expect 75% error coverage or 25% aliasing (see Table 7.1). The *Total* column of LCAR codes in Table 7.6 shows that about 50% of the benchmark circuits reach this expectation, and the remainder is 5.7% to 8.75% lower than the expected value. The reason is mainly because of the bidirectional fault coverage, which we are not able to simulate or cannot simulate adequate number of multiple faults in the circuits.

The *faultsimulator* and *plasim* columns of Table 7.6 depict the unidirectional, bidirectional and the total fault coverage of LCAR codes respectively, from machine LCAR=10. The fault coverage for multiple level gate realization falls between the

	#i.	#o.	#p	Berger		Mod3		LCAR			
								LCAR=10		LCAR=100	
				Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.
rd53	5	3	31	100.00	0.00	78.00	0.00	83.40	60.00	87.92	60.00
2by2	4	4	7	97.04	76.19	75.40	88.81	75.82	63.81	83.96	76.19
f2	4	4	12	94.41	65.53	72.03	72.95	73.13	81.25	81.60	92.86
2by3	5	5	12	97.95	71.73	73.15	73.15	76.27	52.22	85.05	62.67
dc1	4	7	9	96.96	89.28	75.33	67.69	81.78	54.92	88.25	71.05 9
wim	4	7	9	93.78	80.01	80.71	74.64	82.27	69.24	94.10	82.51
Dk27	5	9	10	98.36	89.09	85.52	56.86	85.42	69.02	93.43	78.44
Apla	4	12	26	X[6]	X	83.67	73.85	84.92	67.62	92.13	84.86
Sqr6	4	12	50	X	X	80.03	68.04	84.76	74.66	88.79	85.09

Table 7.4: Fault coverage of three codes (*faultsimulator*)

data from the two PLA simulators. It is interesting to see that the results from *LogicIII(uvic)sim* and *plasim* are very close, in spite of the different types of faults simulated for the two distinct circuit structures.

7.4 Summary

In this chapter, the error detection ability of LCAR codes was examined for a number of benchmark circuits. Comparisons are made among three error-detecting codes, LCAR, Berger and residue codes, by means of two PLA fault simulators and one multiple level gate fault simulator.

In spite of the restrictions on the computational power, the size of benchmark circuits, and the sets of faults simulated, we can conclude from this experiment that:

1. the standard deviation of LCAR code fault coverage generated by different LCARs of the same length varies up to 23%. However, it is not evident that

[6]The simulation failed due to the circuit exceeding the implementation limits.

	#i.	#o.	#p.	Berger		Mod3		LCAR			
								LCAR=10		LCAR=100	
				Uni.	Bi.	Uni.	Bi.	Uni.	Bi.	Uni.	Bi.
rd53	5	3	31	100.00	0.00	74.94	0.00	80.51	0.00	83.39	0.00
2by2	4	4	7	100.00	0.00	62.57	0.00	65.43	0.00	73.46	0.00
f2	4	4	12	100.00	0.00	60.90	0.00	60.93	0.00	69.48	0.00
2by3	5	5	12	100.00	0.00	61.51	0.00	62.75	0.00	68.99	0.00
dc1	4	7	9	100.00	0.00	68.10	0.00	72.74	0.00	74.96	0.00
wim	4	7	9	100.00	0.00	81.07	0.00	78.46	0.00	91.64	0.00
Dk27	5	9	10	100.00	0.00	84.34	0.00	88.05	0.00	94.45	0.00
Apla	4	12	26	100.00	0.00	80.25	0.00	81.09	0.00	88.73	0.00
Sqr6	4	12	50	100.00	0.00	81.14	0.00	89.62	0.00	87.33	0.00

Table 7.5: Fault coverage of three codes (*plasim*)

any LCAR is better than others, as a cyclic code generator. As with other logic problems, it is function-dependent. For a given circuit function and a number of checking bits, the best LCAR code generator can be found by simulation, if the circuit size and computational power permit;

2. the fault coverage of LCAR codes is comparable to residue and Berger codes of the same length. In most cases, the LCAR codes give better fault coverage than residue codes of the same length, in all unidirectional, bidirectional and total fault coverage categories.

In conjunction with the investigation in Chapters 5 and 6, it has been shown that LCAR codes can be used in concurrent checking as an alternative to the conventional error detecting codes in regard to its error detection ability. The high fault coverage of off-line BIST is not affected by introducing the proposed LFSR/LCAR-based concurrent checking. When the merging of on-line and off-line BIST is concerned, LFSR/LCAR-based concurrent checking allows the sharing of hardware resources with off-line signature analysis, and scan-based DFT techniques, thus, resulting in

	LogicIII(uvic)sim			*faultsimulator*			*plasim*		
	Uni.	Bi.	Total	Uni.	Bi.	Total	Uni.	Bi.	Total
rd53	74.80	55.70	78.19	83.40	60.00	82.44	80.51	0.00	65.73
2by2	69.17	13.89	69.30	75.82	63.81	73.11	65.43	0.00	56.70
f2	68.79	0.00	67.92	72.29	81.25	72.38	60.93	0.00	54.74
2by3	71.38	10.76	66.25	76.27	52.22	72.35	62.19	0.00	53.12
dc1	81.01	19.81	78.64	79.92	54.92	77.80	69.20	0.00	66.12
wim	81.36	16.18	74.40	83.16	69.24	79.90	78.46	0.00	64.40
s	-0.25	-7.16	-1.16	-0.32	-12.86	-1.44	1.6	-	6.14
t(.005)	4.03								

Table 7.6: Fault coverage comparisons

a significant reduction in the overall silicon cost.

Chapter 8

Conclusion

Technological advances have been dramatically changing one's traditional view of digital design and test. The IEEE boundary scan standard [1] futher encourages the testing community to accept built-in self-test (BIST) and incorporate testing circuitry in digital designs. Testing problems now can no longer be treated as post-manufacture problems and problems of testing engineers. Testing requirements need to be considered at every step of a design process from the module, chip, board up to system levels. IBM reported that testing costs exceeded 50% of the total chip cost [16]. To reduce the overall cost of production, testing costs must be minimized.

This dissertation primarily encompasses design for testability (DFT) problems of concurrent checking and structural off-line BIST. It differs from traditional circuit design methodologies, where design and testing problems are treated separately, and from the conventional strategy in solving testability problems, where on-line and off-line testing are considered independently. There has been an industry-wide effort to use boundary scan techniques to enhance mainly off-line testability since the release of the standard. In a system with the boundary scan circuitry in place, on-line BIST may be acceptable if it can be provided by sharing the existing resources. The goals are to introduce on-line BIST with a reasonable increase in silicon (in comparison

with having on-line and off-line testability separately), and with acceptable impact on the system performance. The objectives of this research are to pursue a feasibility study of the merging of on-line and off-line BIST, and to find a feasible solution. The main contributions of this dissertation are as follows.

(1) We reviewed previous work on the merging of on-line and off-line BIST, and examined the problems with the merging. This initial research led to a new DFT method and a new testing scheme.

(2) The main idea is to find commonality between the different testing solutions, such that on-line and off-line testing circuitry can be shared. The new DFT method employs LFSR[1] /LCAR[2]-based cyclic code checking as a medium to combine concurrent checking and signature analysis in a built-in fashion, and uses bit sliced LFSRs/LCARs as the implementation mechanism. The choice of LFSR/LCAR codes is crucial: cyclic codes are the basis of signature analysis, and the use of cyclic codes in both concurrent checking and off-line BIST implies the same implementation mechanism. Thus, the hardware resource can be shared between the two test modes. The design procedures for concurrent checking are automated in this research. The suggested DFT method is systematic and can be incorporated in design automation tools.

(3) The new BIST scheme supports both on-line and off-line testability with shared hardware resources and comparable on-line error-detecting ability to the conventional error-detecting codes, and without affecting the high fault coverage of off-line signature analysis. This scheme complies with the IEEE boundary scan standard and is applicable to general circuitry.

The proposed testing scheme is mainly evaluated from the following aspects:

(a) Area overhead is one of the main concerns of any BIST method. The

[1]Linear feedback shift register.

[2]Linear cellular automata register.

area overhead of circuits augmented with error-detecting codes is examined through the application of encoding methods on a set of Berkeley benchmark circuits. Both two level (PLA) and multiple level (standard gate) realizations of the circuits are considered. The error-detecting codes under examination are residue, Berger and LFSR/LCAR-based cyclic codes. Area overhead of LFSR/LCAR codes with the same code lengths but generated by different machine structures is also investigated. The results show that the area overhead of the augmented circuits is function-dependent, as predicted, and that the area overhead of the LFSR/LCAR-based cyclic codes is comparable to residue codes of the same length, and usually more cost effective than Berger codes. In the overall design, including checkers and controls, significant savings are gained by the use of LFSR/LCAR code generators and the merging of on-line and off-line testing resources. UBIST [37, 38] employs different test techniques in on-line and off-line tests, which usually results in a substantial increase in silicon, mainly due to the cost of code generators for concurrent checking. Moreover, considerable design effort is required for circuits designed with a number of error detecting codes.

(b) The error-detecting ability of LFSR/LCAR cyclic codes is investigated on both PLAs and general circuit structures, by means of simulation. It is shown that fault coverage of LCAR codes of the same length generated by different LCARs varies on the set of benchmark circuits. However, it appears that there is no evidence that any LCAR is better than others, as a cyclic code generator. Similarly to other logic problems, it is function-dependent. Statistic tests show that the fault coverage of LCAR codes is comparable to residue and Berger codes of the same length in all unidirectional, bidirectional and total fault coverage categories. Fur-

ther more, LFSR/LCAR-based cyclic codes are general error-detecting codes and applicable to general circuitry. When only unidirectional errors are concerned, however, Berger codes are superior to both residue and LFSR/LCAR codes. In comparison, UBIST [37, 38] employs distinct error detecting codes for different circuit functions. A system designed with UBIST is *strongly fault secure* (SFS)[58], and is superior to our proposed scheme in terms of fault coverage. However, the SFS feature is obtained by paying higher price in silicon.

(c) In conjunction with the investigation in the proposed testing scheme and the area cost estimation, it can be concluded that LFSR/LCAR-based cyclic codes is a viable alternative to the conventional error detecting codes in concurrent checking, regarding its error detection ability. The high fault coverage of off-line BIST is not affected by introducing the on-line testability. When the merging of on-line and off-line BIST is concerned, LFSR/LCAR-based concurrent checking allows the sharing of hardware resources with off-line signature analysis, and scan-based DFT techniques, thus, results in significant reduction on the overall silicon cost.

(d) The concurrent checking latency of the proposed primary scheme is m clock cycles, as opposed to 2^n reported in [48], where n and m are the number of inputs and outputs of the circuit, respectively. Better performance can be achieved by using the suggested modified structures. The high fault coverage of off-line BIST is retained in the merging. Design templates and simulation results on benchmark circuits are provided to demonstrate the feasibility and applicability of the DFT method and the new testing scheme.

The contributions of this dissertation are also made in some related issues in the course of pursuing the main research.

(4) There are two ways to vary lengths of LFSRs and LCARs: concatenation and partitioning. We clarified and summarized recent development in the theoretical and practical issues of LFSR and LCAR concatenation and partitioning. This study has led to a better understanding of the machine behaviour and testing applications in general. A direct application is dynamic reconfigurations of LFSRs/LCARs, which allow the registers to be used in both the on-line and off-line test modes.

(5) A new area estimation method for integrated circuit design, transistor pair layout (TPL), is presented. This method provides a uniform way of area estimation for general circuitry. The estimation is carried out at transistor and logic gate levels. To use the area of a transistor pair as a measurement unit, one can derive a set of estimation equations for standard library cells, different implementation structures, etc, using a CAD tool. Then, the equations can be used to make design comparisons by others in different design houses without requiring access to the CAD tool and going through layout designs. These equations also lift one restriction of the conventional transistor counting method [22], which is only applicable to standard gate realization. The estimation accuracy lies between exact layout and transistor counting.

Merging of on-line and off-line BIST is a promising area of research in testing. In the near future, one may expect more consideration to be given to on-line testability and to the merging of on-line and off-line BIST. This research should increase understanding of the design trade-offs in merging on-line and off-line BIST, and should therefore allow circuits to be tested more economically and effectively. The future research includes the investigation of other merging schemes, BIST structures and implementation mechanisms. In particular, different error-detecting codes and data compaction schemes will be examined for commonality and potential in merging.

Bibliography

[1] IEEE Standard 1149.1. *Standard Test Access Port and Boundary-Scan Architecture.* IEEE Standards Board, May 1990.

[2] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[3] V. D. Agrawal, K. T. Cheng, D. D. Johnson, and T. Lin A complete solution to the partial scan problem. *Proceedings of International Test Conference*, pages 44–51, October 1987.

[4] Paul H. Bardell, William H. McAnney, and Jacob Savir. *Built-In Test for VLSI: Pseudorandom Techniques.* John Wiley & Sons Inc., 1987.

[5] Dilip K. Bhavsar. Concatenable polydividers: bit-sliced LFSR chips for board self-test. *Proceedings of International Test Conference*, pages 88–93, October 1985.

[6] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Prentice Hall, 1985.

[7] R. Byrne. LogicII(uvic)sim. *Department of Computer Science, University of Victoria*, 1992.

[8] K. Cattell and J.C. Muzio. A linear cellular automata algorithm: theory. *Technical report, Department of Computer Science, University of Victoria, DCS-161-IR*, 1991.

[9] K. Cattell and J.C. Muzio. Tables of linear cellular automata for minimal weight primitive polynomials of degrees up to 300. *Technical report, Department of Computer Science, University of Victoria, DCS-163-IR*, 1991.

[10] K. T. Cheng and V. D. Agrawal. A partial scan method for sequential circuits with feedback. *IEEE Transactions on Computers*, pages 544–548, April 1990.

[11] Vivek Chickermane and Janak H. Patel. An optimization based approach to the partial scan design problem. *Proceedings of International Test Conference*, pages 377–386, October 1990.

[12] Hao Dong. Modified Berger codes for detection of unidirectional errors. *IEEE Transactions on Computers*, pages 572–575, June 1984.

[13] E. B. Eichelberger and E. Lindbloom. Random-pattern coverage enhancement and diagnosis for LSSD logic self-test. *IBM Journal of Research & Development*, pages 265–272, May 1983.

[14] Bernard Elspas. The theory of autonomous linear sequential networks. *IRE Transactions on Circuit Theory*, pages 45–60, March 1959.

[15] W. K. Fuchs and J. A. Abraham. A unified approach to PLAs with concurrent error detection in highly structured logic arrays. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 4–9, 1984.

[16] J. Giraldi and M.L. Bushnell. Search state equivalence for redundancy identification and test. *Proceedings of International Test Conference*, pages 184–193, October 1991.

[17] Clay S. Gloster and Franc Brglez. Boundary scan with built in self-test. *IEEE Design & Test of Computers*, pages 36–44, February 1989.

[18] Clay S. Gloster and Franc Brglez. Boundary scan with cellular-based built in self-test. *Proceedings of International Test Conference*, pages 138–145, October 1988.

[19] S. W. Golomb. *Shift Register Sequences*. Aegean Park Press, 1982.

[20] S.K. Gupta and D.K. Pradhan. Can concurrent checkers help BIST? *Proceedings of International Test Conference*, pages 140–150, September 1992.

[21] Peter D. Hortensius, Howard C. Card, Robert D. McLeod, and Werner Pries. Parallel random number generation for VLSI using cellular automata. *IEEE Transactions on Computers*, pages 769–774, June 1989.

[22] S. L. Hurst. A hardware consideration of CALBO testing. *The Third Technical Workshop: New Directions for IC Testing*, pages 129–146, October 1988.

[23] A. Ivanov and Y. Zorian. Count-based BIST compaction schemes and aliasing probability computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 768–777, June 1989.

[24] Barry W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley, 1989.

[25] Bernd Konemann, Joachim Mucha, and Gunther Zwiehoff. Built in logic block observation techniques. *Proceedings of 1979 Test Conference*, pages 37–41, October 1979.

[26] Evaggelia Kontopidi. The partitioning behavior of linear finite state machines and VLSI applications. Master's thesis, University of Victoria, 1992.

[27] A. Krasniewski and S. Pilarski. Cellular self-test path: A low-cost BIST technique for VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 46–55, January 1989.

[28] J. LeBlanc. LOCST: A built-in self-test technique. *IEEE Design & Computers*, pages 42–52, November 1984.

[29] S. Lipschutz. *Linear Algebra.* MacGraw-Hill, 1968.

[30] David Jun Lu and Edward J. McCluskey. Recurrent test patterns. *Proceedings of International Test Conference*, pages 76–82, October 1983.

[31] G. P. Mak, J. A. Abraham, and E. S. Davidson. The design of PLAs with concurrent error detection. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 303–310, 1982.

[32] M. A. Marouf and A. D. Friedman. Design of self-checking checkers for Berger codes. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 179–184, 1978.

[33] D. M. Miller, J. C. Muzio, M. Serra, X. Sun, S. Zhang, and R. D. McLeod. Cellular automata techniques for compaction based BIST. *Proceedings of International Symposium on Circuits and Systems*, pages 1893–1896, June 1991.

[34] D. M. Miller and S. Zhang. A study of the fault coverge of LFSR and CA pseudorandom test pattern generators. *New Direction for Testing: the 5th Technical Workshop*, August 1991.

[35] D. M. Miller and S. Zhang. Aliasing in multiple-input data compactors. *Proceedings of Canadian Conference on Electrical and Computer Engineering*, pages 347–351, September 1989.

[36] M. Nicolaidis. Evaluation of a self-checking version of the MC68000 microprocessor. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 350–356, 1985.

[37] M. Nicolaidis. Self-exercising checker for unified built-in self-test (UBIST). *IEEE Transactions on Computer-Aided Design*, pages 203–218, March 1989.

[38] M. Nicolaidis. Efficient UBIST implementation for microprocessor sequencing parts. *Proceedings of International Test Conference*, pages 316–326, October 1990.

[39] M. Nicolaidis and B. Courtois. Design of self-checking circuits using unidirectional error detecting codes. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 22–27, 1986.

[40] M. Nicolaidis, I. Jansch, and B. Courtois. Strongly code disjoint checkers. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 16–21, 1984.

[41] L. Ott and W. Mendenhall. *Understanding Statistics*. Duxbury Press, 1985.

[42] F. Ozguner. Deductive fault simulation of internal faults of inverter-free circuits and programmable logic arrays. *IEEE Transactions on Computers*, pages 70–73, January 1986.

[43] Bong-Hee Park and Prem R. Menon. Robustly scan-testable CMOS sequential circuits. *Proceedings of International Test Conference*, pages 263–272, October 1991.

[44] W. Wesley Peterson and E. J. Weldon Jr. *Error-Correcting Codes*. The M.I.T. Press, 1972.

[45] B. W. Podaima, P. D. Hortensius, R. D. Schneider, H. C. Card, and R. D. McLeod. CALBO - Cellular Automaton Logic Block Observation. *Proceedings of Canadian Conference on VLSI*, pages 171–176, October 1986.

[46] D. K. Pradhan. *Fault Tolerant Computing, Theory and Techniques*, volume I. Prentice Hall, 1986.

[47] T. R. N. Rao and E. Fujiwara. *Error-Control coding for Computer Systems.* Prentice Hall, 1989.

[48] Kewal K. Saluja, Rajiv Sharma, and C. R. Kime. Concurrent comparative testing using BIST resources. *Proceedings of International Conference on Computer-Aided Design*, pages 336–339, October 1987.

[49] Kewal K. Saluja, Rajiv Sharma, and Charles R. Kime. A concurrent testing technique for digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1250–1260, December 1988.

[50] I. L. Sayers and D. J. Kinniment. Low-cost residue codes and their application to self-checking VLSI systems. *IEE Proceedings, PartE*, pages 197–202, July 1985.

[51] R. M. Sedmak. Design for self-verification: an approach for dealing with testability problems in VLSI-based designs. *Proceedings of 1979 Test Conference*, pages 112–120, October 1979.

[52] R. M. Sedmak. Implementation techniques for self-verification. *Proceedings of 1980 Test Conference*, pages 267–278, October 1980.

[53] M. Serra. Fault detection using concurrent checking. *Proceedings of Canadian Conference on VLSI*, pages 74–79, October 1988.

[54] M. Serra. Some experiments on the overhead for concurrent checking. *3rd Workshop: New Directions for IC Testing*, pages 207–212, October 1988.

[55] M. Serra and T. Slater. A Lanczos algorithm in a finite field and its application. *Journal of Combinatorial Mathematics and Combinatorial Computing*, pages 21–32, April 1990.

[56] M. Serra, T. Slater, J. C. Muzio, and D. M. Miller. The analysis of one-dimensional linear cellular automata and their aliasing properties. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 767–778, July 1990.

[57] T. Slater and M. Serra. Tables of linear hybrid 90/150 cellular automata. *Technical report, Department of Computer Science, University of Victoria, DCS-105-IR*, 1989.

[58] Jame E. Smith and Gernot Metze. Strongly fault secure logic network. *IEEE Transactions on Computers*, pages 491–499, June 1978.

[59] H. S. Stone. *Discrete Mathematical Structures and Their Applications*. Science Research Associates, 1973.

[60] X. Sun and M. Serra. Cellular automata and their concatenation properties. *Technical report, Department of Computer Science, University of Victoria, DCS-157-IR*, February 1991.

[61] X. Sun and M. Serra. Concurrent checking and off-line data compaction testing with shared resources in PLAs. *Journal of Semicustom ICs, England*, pages 8–16, September 1990.

[62] X. Sun and M. Serra. Merging concurrent checking and off-line BIST. *Proceedings of International Test Conference*, pages 958–967, September 1992.

[63] L. T. Wang and E. J. McCluskey. A hybrid design of maximum length sequence generators. *Proceedings of International Test Conference*, pages 38–45, October 1986.

[64] David M. Wessels. The cost and fault coverage of unidirectional error detecting codes for concurrent checking. *Master's thesis, University of Victoria*, 1990.

[65] Thomas W. Williams, Wilfried Daehn, Matthias Gruetzner, and Cordt W. Starke. Bounds and analysis of aliasing errors in linear feedback shift registers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 75–83, January 1988.

[66] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, pages 601–644, July 1983.

[67] C. Y. Wong, W. K. Fuchs, J. A. Abraham, and E. S. Davidson. The design of a microprogram control unit with concurrent error detection. *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 476–483, June 1983.

[68] Yuejian Wu and Andre Ivanov. A multiple signature compaction scheme for BIST. *Proceedings of Canadian Conference on VLSI*, pages 2.2.1–2.2.7, 1991.

[69] S. Zhang. Pseudo-NMOS PLA fault simulation with parallel bit operation. *Department of Computer Science, University of Victoria*, 1991.

[70] S. Zhang, D. M. Miller, and J. C. Muzio. Determination of minimal cost one-dimensional linear hybrid cellular automata. *Electronics Letters*, pages 1625–1626, August 1991.

Appendix 1

Tables of LCAR Concatenation

1.1 Self-concatenation of Primitive LCARs

This section contains tables of primitive self-concatenation of primitive LCARs, one for each length from length 2 to 16, forming longer primitive LCARs up to length 64. The digits in the brackets are the number of self-concatenations to construct a primitive LCAR from a given LCAR. As an example, in the table of length 4, the primitive LCAR, 1010, self-concatenates 3, 7, 13 and 15 times to obtain primitive LCARs of length 12, 28, 52 and 60 respectively.

Length 2
Initial LCAR (Number of Self-Concatenations of Initial LCAR)
10 (2,3,5,6,9,11,14,23,26,29,30) 01 (2,3,5,6,9,11,14,23,26,29,30)

Length 3
Initial LCAR (Number of Self-Concatenations of Initial LCAR)
100 (4) 110 (16) 001 (4) 011(16)

Length 4
Initial LCAR (Number of Self-Concatenations of Initial LCAR)
1010 (3,7,13,15) 0101 (3,7,13,15)

Length 5		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
11000 (6,7)	01100 (3,9)	11100 (6,7)
00110 (3,9)	11001 (3,8,9)	00011 (6,7)
10011 (3,8,9)	00111 (6,7)	

Length 6		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
100000 (3)	101010 (2,3,10)	011010 (6,8)
010110 (6,8)	101110 (9)	000001 (3)
101001 (2,4,6)	100101 (2,4,6)	010101 (2,3,10)
011101 (9)		

Length 7		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
0010000 (6)	1001000 (3)	0000100 (6)
1010100 (3)	1110100 (4)	1000010 (8)
1101010 (3)	1110110 (3)	1011110 (4)
0100001 (8)	0001001 (3)	0010101 (3)
0111101 (4)	0101011 (3)	1111011 (6)
0010111 (4)	0110111 (3)	1101111 (6)

Length 8		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
11110000 (7)	10100010 (3,5)	11010010 (8)
11011010 (5)	10111010 (3)	11101110 (2 8)
01000101 (3,5)	01011101 (3)	01001011 (8)
01011011 (5)	01110111 (2,8)	11110111 (4,7)
00001111 (7)	11101111 (4,7)	

Length 9		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
011100000 (2)	101110000 (5,6)	001001000 (7)
111001000 (3,7)	111101000 (3)	111011000 (3,4,7)
110000100 (7)	000100100 (7)	001100100 (3)
001001100 (3)	010101100 (2,3)	110111100 (7)
111100010 (5,6)	001101010 (2,3)	111010110 (7)
000001110 (2)	101111110 (4)	111010001 (7)
110010101 (2)	000011101 (5,6)	011111101 (4)
001000011 (7)	101010011 (2)	110110011 (3,5)
110011011 (3,5)	111011011 (2,4,7)	001111011 (7)
000100111 (3,7)	100010111 (7)	011010111 (7)
000110111 (3,4,7)	110110111 (2,4,7)	010001111 (5,6)
000101111 (3)		

Length 10		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1111110000 (3)	0110101000 (3)	1111100010 (3)
1101001010 (3,5)	1010101010 (6)	0110100110 (5)
1000010110 (4)	0110010110 (5)	0001010110 (3)
1100000001 (2)	0110100001 (4)	1101110001 (3,4)
0101010101 (6)	1011101101 (2)	1011011101 (2)
1000000011 (2)	0101001011 (3,5)	1000111011 (3,4)
1111010111 (3,5)	1110101111 (3,5)	0100011111 (3)
0000111111 (3)		

Length 11		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
00110000000 (5)	00001000000 (3)	11100100000 (2)
00000010000 (3)	11101010000 (3)	10000001000 (3)
00100001000 (3)	11010001000 (3)	01101011000 (3)
11111011000 (2)	00010000100 (3)	01111000100 (2,4)
01010100100 (3)	01110010100 (5)	11101110100 (3)
00000001100 (5)	10001011100 (5)	01110100010 (5)
01001100010 (3,5)	11100010010 (3,4)	10000110010 (4)
01000110010 (3,5)	00100101010 (3)	10000010110 (5)
00011010110 (3)	01111110110 (2)	00101001110 (5)
00010101011110 (3)	01001010111110 (3)	10100111011110 (2)
01000101110 (5)	11000101110 (5)	00100011110 (2,4)
10010111110 (5)	01101111110 (2)	11101111110 (3)
00010000001 (3)	10010000001 (2)	01101000001 (5)
01001100001 (4)	11011100001 (2,4)	11010010001 (4)
00111010001 (5)	10111010001 (5)	11100110001 (4)
11010110001 (5)	11110110001 (4)	11101110001 (3)
10011110001 (4)	10000001001 (2)	01111101001 (5)
10001111001 (4)	10101100101 (3)	11011010101 (3)
10100110101 (3)	11110001101 (2)	10111001101 (5)
10113011101 (5)	10001011101 (5)	11001011101 (3)
01110100011 (5)	10111010011 (3)	11111110011 (5)
00010001011 (3)	10001001011 (4)	10001101011 (5)
10101011011 (3)	10000111011 (2,4)	01001000111 (3,4)
00000100111 (2)	10001100111 (4)	11101100111 (2)
00001010111 (3)	11100110111 (2)	10001110111 (3)
00101110111 (3)	01111110111 (3)	10110001111 (2)
10001101111 (4)	00011011111 (2)	11001111111 (5)

Length 12		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1011010100000 (2)	1100011100000 (2)	1000001010000 (4)
0011011010000 (2)	1001100111000 (2)	0011011111000 (3)
1101111000100 (3)	1010001101000 (3)	1101000011000 (2)
1000101011100 (2)	0001011011000 (2)	0101011011000 (4)
0001111011000 (3)	1101000111000 (4)	1101111111000 (3)
1000100100010 (3)	0111110010100 (4)	1010001010100 (2)
1010101010100 (5)	1001011010100 (2)	0011011010100 (4)
1101010111010 (3)	1101001110100 (3)	1111100101100 (3)
0101001111100 (4)	0001010000010 (4)	0100100100010 (3)
0011010100010 (2)	1001010010010 (2)	1001001010010 (2)
1010011010010 (2,3)	0101011010010 (2)	0001100110010 (2)
0101010001010 (2)	0010110001010 (3)	1001011001010 (2,3)
0101010101010 (5)	0000101011010 (2)	0000111000110 (2)
1111000100110 (3)	1110100100110 (2)	0011000010110 (2)
0011100010110 (4)	1101010010110 (2)	0101110010110 (3)
1101001010110 (2)	0101101010110 (3)	0010001110110 (3)
1110111110110 (2,4)	0011111110110 (3)	1100100101110 (2)
1101111101110 (2,4)	1100100011110 (3)	0110100111110 (3)

Length 13		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1011116000000 (2,4)	1101000100000 (3)	1100001100000 (2)
0001001100000 (2)	1100101100000 (3)	0110101100000 (3)
1101111100000 (4)	0011000010000 (2)	0110100010000 (2)
0011100010000 (3)	0101010010000 (2)	0000110010000 (4)
0100110010000 (3)	0010110010000 (3)	0110001010000 (4)
0011001010000 (3)	0001101010000 (4)	1111011010000 (2)
1000000110000 (3)	0000100110000 (4)	1100010110000 (2)
1001110110000 (2	1010101110000 (2,3)	0101101110000 (2)
1011101110000 (4)	0110111110000 (2)	1000010001000 (3)
1100110001000 (2)	1010001001000 (2)	1110101001000 (4)
0111101001000 (3)	0000011001000 (2)	0100100101000 (2)
1110100101000 (3)	1110110101000 (4)	1110101101000 (3)
1011111101000 (3)	1011000011000 (3)	0000101011000 (4)
1001011011000 (3)	1011010111000 (4)	1111100000100 (4)
1100101000100 (2)	1011100100100 (3)	1100110100100 (4)
0110000010100 (4)	0111001010100 (3)	1000101010100 (3)

Length 13 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
0010111010100 (2,3)	1010111010100 (3)	0011000110100 (4)
0000100110100 (3)	0010101110100 (2,3)	1111101110100 (3)
1000011110100 (2)	0100111110100 (4)	0110111110100 (3)
1011111110100 (4)	1000000001100 (3)	0000100001100 (2)
0011010001100 (2)	0010110001100 (4)	0000101001100 (3)
1101101001100 (4)	1110111001100 (2,3)	0011000101100 (2)
1100100101100 (3)	1111100101100 (3)	1010010101100 (2)
1101110101100 (2)	1011110101100 (3)	1001001101100 (3)
1100101101100 (3)	1010101101100 (3)	1011101101100 (3)
1000000011100 (4)	0100000011100 (2)	0000100011100 (3)
1100010011100 (2)	1100011011100 (2)	1111100111100 (2)
0111011111100 (2)	1101000000010 (4)	1110100000010 (3)
0011100000010 (2)	1110010000010 (2)	1001001000010 (2)
1100101000010 (3)	0101011000010 (4)	1100100100010 (3)
1101101100010 (3)	0110010010010 (3)	0001010010010 (2)
1110001010010 (4)	0111101010010 (3)	0000100110010 (3)
1000101110010 (2)	1110011110010 (3)	0010111110010 (4)
1101010001010 (3)	0110101001010 (2)	1101111001010 (4)
1111000101010 (2,3)	0000100101010 (2)	1100100101010 (3)
0100001101010 (4)	1100101011010 (2)	0000111011010 (2)
1110110111010 (2)	0110001110010 (3)	0010100000110 (4)
0111100000110 (3)	1111001000110 (2)	0000101000110 (4)
1010111000110 (4)	0101111000110 (3)	1000000100110 (3)
0100100100110 (3)	1111000010110 (4)	0000100010110 (2)
1110010010110 (3)	1001001010110 (4)	0101001010110 (2)
0000011010110 (3)	1001000110110 (3)	0111000110110 (3)
1011110110110 (2)	1001001110110 (3)	0000111110110 (2)
0010111110110 (3)	0110110001110 (3)	1000001001110 (3)
0010101001110 (3)	1011000101110 (2)	1101010101110 (3)
0011111101110 (2)	0110000011110 (3)	1101000011110 (2)
0001001011110 (3)	0100101011110 (3)	1010011011110 (2)
1000110111110 (3)	1100111111110 (3)	0011000000001 (3)
0011100000001 (4)	0110010000001 (3)	0000110000001 (3)
0111001000001 (3)	0001000100001 (3)	1011010100001 (3)
1110110100001 (3)	0010111100001 (2)	1001111100001 (3)
0010101010001 (3)	1110101010001 (3)	0100111010001 (2)
0111110110001 (3)	1001001110001 (3)	1101000001001 (3)
1111000001001 (2)	0110110001001 (3)	0100001001001 (2)
1011001001001 (2)	0110101001001 (4)	0011011001001 (3)
1000111001001 (3)	0110111001001 (3)	1111100101001 (3)

Length 13 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1101010101001 (2)	1010110101001 (2)	0001101101001 (3)
1111011101001 (2)	1010101011001 (2)	1011011011001 (3)
1010000111001 (3)	1111010111001 (4)	0000110111001 (2)
1000011111001 (3)	1001110000101 (3)	0001001000101 (2)
0011010100101 (2)	0111101100101 (2)	1011110010101 (4)
1001101010101 (2)	0011011010101 (3)	1111011010101 (2)
0000111010101 (2,3)	1001010110101 (2)	0110001110101 (4)
0010101110101 (3)	1101000001101 (4)	0001100001101 (3)
0111010001101 (2)	1001001001101 (2)	1111011001101 (3)
1000010101101 (3)	0001110101101 (4)	1001101101101 (3)
0010010011101 (3)	0011011011101 (3)	0000111011101 (4)
0000000111101 (2,4)	1101000111101 (2)	1010100111101 (4)
0011010111101 (3)	0110110111101 (2)	1110011111101 (3)
0001011111101 (3)	0010111111101 (4)	0000011000011 (2)
0011100100011 (2)	0000110100011 (2)	00 101100011 (2)
0001001011110 (3)	0100101011110 (3)	1010011011110 (2)
1111011100011 (3)	0100010010011 (3)	0101010010011 (3)
0011010010011 (3)	0100001010011 (3)	0010001010011 (2)
0101101010011 (2)	0000011010011 (3)	0011011010011 (3)
1100111010011 (2)	0001000110011 (2)	0010010110011 (4)
1111010110011 (3)	1100101110011 (2)	1111011110011 (2)
0111111110011 (3)	0100000001011 (4)	1001000001011 (3)
1011000001011 (4)	0111100001011 (2)	0000010001011 (3)
1101010001011 (2)	1011110001011 (2)	1111011001011 (3)
0101000101011 (3)	1101000101011 (2)	1001010101011 (2)
0111010101011 (3)	0011001011011 (4)	0100011011011 (3)
0011010111011 (2)	0101001111011 (4)	0000011111011 (4)
0100101000111 (4)	0101000100111 (2)	0110100100111 (3)
1111010100111 (4)	0100111100111 (3)	1011111100111 (3)
0100000010111 (3)	0001010010111 (3)	0001001010111 (4)
1000101010111 (3)	0001011010111 (3)	1000010110111 (3)
0001010110111 (4)	0101110110111 (2)	0011001110111 (2,3)
1001000001111 (2)	0110100001111 (4)	0101010001111 (2,3)
0110001001111 (2)	1111011001111 (4)	1110010 1111 (4)
1100110101111 (3)	1001110101111 (4)	1101001101111 (3)
1011001101111 (3)	1111001101111 (4)	0000101101111 (2)
1010101101111 (2)	1100011101111 (3)	1001011101111 (2)
1100111101111 (2)	0010000011111 (4)	1001010011111 (3)
0011010011111 (3)	0011110011111 (2)	0010111011111 (3)

Length 14		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
10000000000000 (4)	01101000000000 (3)	10101111000000 (3)
01010000100000 (3)	00001000100000 (2)	01101100100000 (3,4)
11110101100000 (2)	00101101100000 (2)	00110011100000 (2)
00000100010000 (2)	10110001010000 (3,4)	01011000110000 (3)
10010100001000 (4)	00100000101000 (4)	11001011101000 (2)
11100111101000 (4)	11110111101000 (2)	00100100011000 (3)
10111100011000 (2)	01010001011000 (4)	11110001011000 (3)
11011011011000 (3)	11010001111000 (2)	01010011111000 (3)
10110000000100 (2)	00111000000100 (3)	00010100000100 (4)
01010100000100 (3)	01110101000100 (4)	01110111000100 (2)
11011111000100 (4)	00011000100100 (3)	01011110100100 (4)
10101001100100 (3)	11001011010100 (2,3)	10011111010100 (2)
01110100110100 (4)	00000110110100 (2)	01101001110100 (3)
10110101110100 (4)	11011101110100 (3)	10011011110100 (3)
11010110001100 (3)	00110101001100 (4)	00000111001100 (2)
10111111001100 (4)	10000100101100 (4)	01100010101100 (3)
00110010101100 (4)	10011010101100 (3)	11010001101100 (3)
11011001101100 (4)	01101011101100 (3)	10001111101100 (3)
00100000011100 (3)	01001000011100 (2)	10111000011100 (3)
10010011011100 (2)	01100100111100 (3)	10001011111100 (3)
10110111111100 (2)	11011000000010 (4)	01111110100010 (2)
11000111100010 (3)	10101111100010 (4)	00111000010010 (2)
01111010010010 (3)	11101110010010 (3)	11111001010010 (3)
01101011010010 (2)	10010111010010 (2)	10111001110010 (2)
11110101110010 (3,4)	11011111110010 (2)	00000100001010 (3)
01101100001010 (2)	11111100001010 (3)	10100010001010 (2)
00011010001010 (4)	11010[illegible]1001010 (4)	01100011001010 (2)
10011011001010 (2)	00011111001010 (3)	10000000101010 (2)
00100000101010 (3)	10011001101010 (4)	11011001101010 (3)
10111011101010 (3,4)	00001100011010 (3)	10001100011010 (3,
00100101111010 (4)	11110101111010 (2,4)	11010100000110 (2)
10011110000110 (3)	00110101000110 (3)	01010011000110 (2)
11010000100110 (3)	10101100100110 (2)	00111100100110 (3)
10110010100110 (3)	11001010100110 (4)	10101001100110 (4)
10110111100110 (4)	00000000010110 (3)	10001000010110 (4)
10111010010110 (3)	00101110010110 (3)	01001011010110 (2)
00110111010110 (3)	01010000110110 (2)	00000100110110 (3,4)

Length 14 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
11110011110110 (4)	10011000001110 (2)	11110111001110 (4)
00101100101110 (4)	00100010101110 (4)	10011101101110 (2)
11011101101110 (4)	00100011101110 (2)	10111000011110 (3)
01001001011110 (3)	01000101111110 (2)	10110101111110 (2)
00000000000001 (4)	01010100000001 (2)	10111010000001 (2)
00110100100001 (4)	10110110100001 (3)	11001011100001 (3)
01101000010001 (4)	11011100010001 (2)	11000001010001 (4)
11011101010001 (4)	00111111010001 (3)	01011000110001 (3)
11001110110001 (3)	00110111110001 (3)	11000000001001 (3)
11000001001001 (2)	00111011001001 (2)	11111011001001 (3)
10101111001001 (2)	00010000101001 (4)	11001000101001 (3)
01001011101001 (2)	01110000011001 (2)	01010110011001 (4)
11010001011001 (4)	00110101011001 (3)	01010011011001 (2)
00101111011001 (3)	11011111011001 (3)	10101100111001 (2)
01110110111001 (2)	01100001111001 (3)	00101011111001 (2)
11111000000101 (3)	01010001000101 (2)	10110100100101 (2,3)
10110001100101 (3)	00100110010101 (3)	01100110010101 (4)
10111110010101 (2)	11011111010101 (3)	01100100110101 (2)
10011100110101 (2)	11100101110101 (4)	00000011110101 (3)
10010011110101 (2)	11111011110101 (3)	01000111110101 (4)
00100000001101 (2)	11010000001101 (3)	00001010001101 (3,4)
10100110001101 (3)	01100101001101 (3)	10100100101101 (2,3)
00101110101101 (4)	01111110101101 (2)	10000101101101 (3)
01100111101101 (4)	00111111101101 (2)	00111000011101 (3)
01111000011101 (3)	01001110011101 (2)	10111110011101 (4)
10000001011101 (2)	01101001011101 (3)	01010111011101 (3,4)
00011000111101 (2)	10101001111101 (2)	10111001111101 (4)
00110011111101 (4)	10010000000011 (3)	10010010000011 (2)
10001010000011 (4)	01000111100011 (3)	10010100010011 (3)
01100101010011 (4)	11001101010011 (4)	00101011010011 (2,3)
10000111010011 (3)	00010111010011 (2)	11111000110011 (2)
11001010110011 (4)	10001101110011 (3)	10110000001011 (3)
01100100001011 (3)	10011010001011 (4)	00110110001011 (3)
00011110001011 (2)	01100000101011 (2)	01010010101011 (4)
00110001101011 (3)	01000000011011 (4)	01010110011011 (3)
00110110011011 (4)	00011011011011 (3)	10001000111011 (2)
10001010111011 (4)	01110110111011 (4)	00101110111011 (3)
00100011111011 (4)	10101011111011 (3)	11101011111011 (4)
10011011111011 (3)	01001111111011 (2)	11110110000111 (2,3)
10101110100111 (4)	00010111100111 (4)	11111011010111 (4)

Length 14 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
11011111010111 (4)	01001001110111 (3)	00011010001111 (3)
01101111001111 (4)	00000110101111 (2)	01001110101111 (3,4)
01011110101111 (2,4)	11100001101111 (2,3)	01110011101111 (4)
00010111101111 (2)	10100000011111 (3)	11001100011111 (2)
01001010011111 (3)	10010011011111 (3)	11101011011111 (4)
10101113011111 (3)	01010000111111 (3)	

Length 15		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
100100100000000 (3)	001101010000000 (3)	111110110000000 (2)
010111110000000 (3)	111010001000000 (2)	001001001000000 (2)
010010011000000 (3)	111011011000000 (2)	000100111000000 (4)
011100000100000 (3)	110010000100000 (3)	001110000100000 (4)
111011000100000 (4)	111011100100000 (4)	110111100100000 (4)
110010010100000 (3)	111001010100000 (4)	100111010100000 (3)
100100110100000 (2)	011001001100000 (3)	001101101100000 (2)
111100111100000 (3)	001101111100000 (2,3)	110010000010000 (4)
110001100010000 (3)	101010010010000 (2)	111110110010000 (4)
111010101010000 (2)	001111100110000 (3)	0111101 0110000 (3)
100111001110000 (4)	101110111110000 (4)	010100110001000 (3)
011011110001000 (4)	101011001001000 (2)	110111001001000 (4)
111111001001000 (2)	011000101001000 (2)	111010101001000 (3)
000000111001000 (4)	001000111001000 (2)	001111111001000 (3)
110010000101000 (4)	001001100101000 (2)	101111100101000 (3)
010000001101000 (3)	011110001101000 (2)	011101101101000 (3)
011111011101000 (2)	100111111101000 (3)	100001100011000 (3)
111001001011000 (4)	001100101011000 (3)	001100011011000 (3)
100001011011000 (3)	001101000111000 (2)	110111010111000 (2)
101101001111000 (3)	100111001111000 (3)	011111001111000 (3)
010001011111000 (3)	010110111111000 (3,4)	100111010000100 (3)
001111001000100 (2)	111000101000100 (4)	001110101000100 (4)
010110011000100 (4)	100001011000100 (4)	000100111000100 (2)
100000000100100 (4)	000000100100100 (2)	010100010100100 (3)
101000001100100 (2)	001010001100100 (3)	101110001100100 (3)
000101001100100 (2)	100101001100100 (3)	100100011100100 (3)
100110100010100 (3)	001001100010100 (3)	001110110010100 (3)

Length 15 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
010010011010100 (4)	110001000110100 (3)	101111010110100 (3)
011110110110100 (4)	010001001110100 (3)	101111101110100 (3)
111001011110100 (3)	010100111110100 (2,3)	110011111110100 (2,3)
110110000001100 (2)	010111010001100 (3)	100000110001100 (3)
011000110001100 (3)	000110110001100 (3)	011010001001100 (3)
000110101001100 (3)	010101101001100 (3)	010101011001100 (2)
010010111001100 (2)	011110111001100 (3)	110001111001100 (3)
000111000101100 (2)	000000010101100 (3)	101000101101100 (3)
011110101101100 (2)	000001101101100 (2)	010011101101100 (3)
111010111101100 (4)	000001111101100 (2,3)	000001000011100 (4)
110011000011100 (3)	010111010011100 (3)	111101110011100 (3)
101001001011100 (4)	111001001011100 (3)	001000101011100 (4)
001010011011100 (3)	100001011011100 (2)	110100111011100 (3)
110000000111100 (4)	101100000111100 (2)	001000100111100 (2)
100110100111100 (3)	100010010111100 (4)	000011001111100 (3)
101111101111100 (4)	010110011111100 (3)	000100111111100 (3)
000101100000010 (3)	010011100000010 (3)	100100010000010 (2)
110100010000010 (3)	110100101000010 (2)	100101101000010 (4)
001011100100010 (3)	010111010100010 (3,4)	101111010100010 (2,3)
011110110100010 (3)	000111110100010 (3)	111001011100010 (3)
100011011100010 (3,4)	011101100010010 (3)	000000110010010 (3)
101100110010010 (3)	001010110010010 (4)	011011110010010 (3)
100101001010010 (3)	001100111010010 (2)	100101000110010 (3,4)
101101100110010 (3)	011111100110010 (3)	010111110110010 (4)
010000011110010 (3)	100101001110010 (3)	001101101110010 (3)
001001010001010 (3)	011101010001010 (2)	010111010001010 (4)
011010110001010 (3)	011001001001010 (3)	000100011001010 (3)
010101011001010 (3)	111011011001010 (2)	001011111001010 (2,3)
100101000101010 (2)	100101010101010 (4)	010100110101010 (3)
001100110101010 (2)	011000101101010 (4)	001100101101010 (3)
111100111101010 (2)	001000110011010 (4)	001111110011010 (3)
110001011010010 (4)	100001111011010 (3)	000111111011010 (3,4)
100111000111010 (3)	111011100111010 (4)	010100010111010 (4)
001100010111010 (3)	110010010111010 (3)	001110010111010 (3)
010001010111010 (3,4)	101110110111010 (3)	000000011111010 (3)
010011011111010 (4)	011011000000110 (3)	100010110000110 (2)
000100101000110 (2)	111110101000110 (3)	010101101000110 (4)

Length 15 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
110111101000110 (3)	111000011000110 (3)	001100011000110 (3)
011111000100110 (3)	010100100100110 (3)	000001100100110 (3)
110001100100110 (3)	011101100100110 (4)	110000110100110 (3)
110101110100110 (3)	111101111100110 (3)	001100100010110 (3)
110010001010110 (4)	111101001010110 (4)	101010101010110 (4)
010100011010110 (3)	110110011010110 (3)	101101011010110 (3)
101100111010110 (3)	011000000110110 (3)	111111110110110 (3)
100000011110110 (3)	000100011110110 (4)	010010011110110 (3)
111001111110110 (3)	000001000001110 (3)	111001100001110 (4)
011101110001110 (3)	101110001001110 (3)	100000101001110 (3)
010100010101110 (2)	010010001101110 (3)	011001001101110 (4)
110000101101110 (4)	101000101101110 (4)	000101101101110 (3)
011100011101110 (3)	101001000011110 (3,4)	000101100011110 (2)
111001110011110 (3)	110001001011110 (2)	111001001011110 (3)
001101101011110 (2)	010001011011110 (3)	000011011011110 (3)
001011011011110 (4)	001100111011110 (3)	011111111011110 (3)
011001000111110 (3)	100001100111110 (3)	110011100111110 (3)
000111100111110 (3)	100011010111110 (3)	000101110111110 (2)
100100001111110 (3)	010011001111110 (3)	011110111111110 (3)
100001000000001 (3)	001001000000001 (4)	111110010000001 (4)
100000110000001 (4)	101100110000001 (3)	011011110000001 (3)
111010001000001 (2)	011100101000001 (3)	100000011000001 (4)
001100011000001 (3)	100110111000001 (3)	100000000100001 (3)
110011000100001 (3)	101000100100001 (2)	110100100100001 (4)
111100100100001 (3)	110010010100001 (2)	001000110100001 (4)
000110110100001 (3)	001110110100001 (2)	000110001100001 (3)
011111001100001 (3)	111010011100001 (2)	010110111100001 (3)
100101000010001 (4)	100011100010001 (3)	111010010010001 (3)
100101010010001 (4)	001111010010001 (4)	100110110010001 (3)
101101110010001 (3)	101011101010001 (2)	011000011010001 (2)
111010010110001 (2)	011111010110001 (3)	010001110110001 (3,4)
100100001110001 (3)	101001001110001 (3)	111110111110001 (3)
101101100001001 (3)	011111100001001 (3)	010000010001001 (2)
100110110001001 (4)	001001110001001 (3)	000000001001001 (3)
000001011001001 (2)	100111111001001 (3)	100010000101001 (4)
010101000101001 (2)	010011000101001 (3,4)	010010100101001 (3)

Length 15 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
001001100101001 (3)	010011100101001 (3)	101011100101001 (3)
100010010101001 (4)	010101010101001 (4)	110110110101001 (2)
010000101101001 (4)	100011011101001 (3)	111101110011001 (4)
001010001011001 (3)	001111001011001 (3)	100100011011001 (4)
100010011011001 (3)	111110011011001 (3)	101011011011001 (3)
100000111011001 (3)	010111000111001 (3)	000011100111001 (4)
110011100111001 (3)	000111100111001 (3)	001000010111001 (3)
101010010111001 (3,4)	000001010111001 (3)	110010011111001 (2)
100100111111001 (3)	000101111111001 (3)	101100100000101 (4)
001001100000101 (2)	111111100000101 (3)	100001001000101 (2)
101110100010101 (3)	110001101000101 (3)	001101101000101 (3)
011101101000101 (4)	011110000100101 (3,4)	110011000100101 (2)
001110100100101 (4)	100011100100101 (3)	110000001100101 (3)
101100011100101 (3)	000010010010101 (2)	100111010010101 (3,4)
101110110010101 (3)	110011001010101 (2)	101011001010101 (3)
011010101010101 (4)	101110000110101 (3)	000100100110101 (2)
101010100110101 (3)	111011100110101 (3)	101100010110101 (3)
100110110110101 (3)	100101001110101 (3)	100010101110101 (2)
110010111110101 (3)	001111000001101 (2)	110010010001101 (3)
101011010001101 (3)	101001110001101 (3)	101111110001101 (3)
101000001001101 (4)	100000011001101 (3)	010010011001101 (3)
011010111001101 (3)	000111100101101 (3)	011010110101101 (3)
100100001101101 (3)	111110001101101 (3)	010011001101101 (3)
111111001101101 (3)	100010011101101 (3)	101011000011101 (3)
011100100011101 (3)	001001100011101 (3)	110110110011101 (2)
111111110011101 (4)	111000001011101 (3)	111100001011101 (2)
101000101011101 (3)	101010011011101 (3)	010111011011101 (3)
110110111011101 (3)	000011111011101 (4)	010001010111101 (2,3)
001011010111101 (3)	000101001111101 (3)	001011101111101 (3)
001111101111101 (4)	101100011111101 (3)	111001111111101 (2)
111011000000011 (3)	001111000000011 (4)	101001100000011 (3)
011101101000011 (4)	011001011000011 (3)	001011000100011 (3)
011110100100011 (2)	110101100100011 (3)	010110110100011 (4)
111001110100011 (3)	000010001100011 (3)	011001001100011 (3)
101000101100011 (3)	001100111100011 (3)	111110111100011 (2)
000010000010011 (4)	000001000010011 (3)	000101000010011 (4)
011010100010011 (4)	101100010010011 (3)	000001010010011 (3)

Length 15 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
100001010010011 (2)	010111010010011 (3)	100111110010011 (2)
111111101010011 (3)	111000111010011 (2)	101011111010011 (3)
001110000110011 (3)	100001000110011 (3)	101001000110011 (2)
101010100110011 (2)	111011100110011 (2)	100101110110011 (3)
111001001110011 (3)	100111001110011 (3)	011111001110011 (3)
001011111110011 (2,3)	010000010001011 (3)	100001001001011 (4)
010000101001011 (2)	001110111001011 (3)	110001001101011 (3)
111001101101011 (3)	110110011101011 (3)	011001011101011 (3)
001100000011011 (2)	111110100011011 (3)	111011010011011 (4)
011010110011011 (3)	110101110011011 (3)	101110011011011 (2)
100101011011011 (2)	101110111011011 (3)	111100000111011 (3)
000100100[illegible]1011 (4)	000111010111011 (2)	111101110111011 (3)
000001001111011 (4)	011000101111011 (3)	111100111111011 (2)
101110100000111 (3)	011000110000111 (3)	001000101000111 (4)
110010111000111 (2)	111110000100111 (2)	000110100100111 (4)
001110100100111 (3)	011110100100111 (3)	110011100100111 (3)
000001010100111 (4)	010001110100111 (3)	001011110100111 (3)
011100001100111 (4)	110101101100111 (3)	011110011100111 (3)
110001011100111 (3)	011011111100111 (3)	101111111100111 (2)
000000100010111 (2)	100000100010111 (2)	100010010010111 (3)
100011010010111 (2)	100001110010111 (2)	000100101010111 (3)
000010101010111 (2)	001101111010111 (4)	110000000110111 (3)
000001000110111 (4)	110110010110111 (4)	000000110110111 (2)
010100110110111 (2)	110011001110111 (2)	101011001110111 (3)
010111001110111 (4)	110111000001111 (3)	101110100001111 (2)
100001001001111 (3)	000001111001111 (3)	010101111001111 (2)
110111111001111 (2)	011010100101111 (4)	000001001101111 (4)
100110011101111 (4)	001110011101111 (3)	110111011101111 (3)
011001111101111 (3)	111001000011111 (2)	101101100011111 (3)
100000010011111 (4)	100110110011111 (3)	110110001011111 (3)
011000101011111 (3)	000000013011111 (2)	000010011011111 (4)
110001111011111 (2)	100011111011111 (3)	000100100111111 (2)
101101100111111 (3)	101000001111111 (3)	110010101111111 (3)
101110011111111 (4)	011011011111111 (3)	

Length 16		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
0011110000000000 (4)	0000101000000000 (2,4)	0100011000000000 (2)
1101100100000000 (2)	0111010010000000 (2)	1011000110000000 (4)
0110011110000000 (4)	1101110101000000 (4)	0111110011000000 (2)
1101101000100000 (3)	1001111100100000 (2)	1000001010100000 (3,4)
1110011010100000 (2)	0100010110100000 (4)	1111010001100000 (4)
0010001101100000 (3)	1010011101100000 (2)	0111001011100000 (2)
1110101011100000 (4)	0101011111100000 (2)	1010111111100000 (4)
1111101100010000 (3)	1001011100010000 (2)	1000011010010000 (3)
0000000001010000 (2,4)	0101100001010000 (4)	0011100000110000 (2)
1000000100110000 (3)	0010010001110000 (2)	0011110001110000 (3)
1001001011110000 (4)	1100101011110000 (3)	1111101000001000 (2)
1000100010001000 (4)	1000110010001000 (3)	1010110110001000 (4)
0111111110001000 (3)	0010100001001000 (3)	0001010001001000 (3)
1100011101001000 (4)	0011011101001000 (4)	0101111011001000 (2)
0010011111001000 (2)	0001001000101000 (3)	0101101100101000 (4)
1111100010101000 (4)	1111011001101000 (4)	1011000101101000 (2)
0110100101101000 (2)	0110010011101000 (2)	1010110011101000 (3)
0110101111101000 (3)	1101110000011000 (4)	1010101000011000 (4)
0111110110011000 (2)	1111101110011000 (4)	0001110101011000 (2,4)
1001110011011000 (3)	1110111011011000 (3)	0011000111011000 (2,3)
1100100111011000 (4)	1000111100111000 (4)	1000110010111000 (4)
0100110010111000 (4)	0001101010111000 (2,4)	1101000110111000 (2)
0011010110111000 (4)	1011001101111000 (4)	0110010011111000 (3)
1001000000000100 (2)	1000001100000100 (4)	0100110101000100 (3)
0100010011000100 (4)	0000011011000100 (3)	1110011111000100 (4)
0000111000100100 (2)	1101010100100100 (2)	1101110010100100 (2,4)
0111000101100100 (2)	0011101101100100 (3)	0100010011100100 (3)
0101010011100100 (3)	0100101111100100 (2)	0001001111100100 (2)
0100010000010100 (3)	0001001000010100 (3)	0110100100010100 (4)
1011100010010100 (3)	0100000110010100 (4)	1001100110010100 (4)
1001101110010100 (3)	1010110011010100 (3)	1110100111010100 (2)
0110100000110100 (3)	1101100000110100 (4)	1110111000110100 (2)
1111101010110100 (4)	0101000001110100 (3)	1110001001110100 (2)
1010100101110100 (2)	1000101101110100 (2)	1100100011110100 (3)
1101010011110100 (3)	1001100000001100 (4)	1010000010001100 (2)
1111100110001100 (4)	0001101110001100 (2,3)	0011110111001100 (4)
0110010110101100 (3)	0001110110101100 (4)	1110010001101100 (4)

Length 16 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
0001001011101100 (4)	0000110000011100 (2)	1101000100011100 (3)
0100100100011100 (4)	1001010100011100 (2)	1100000011011100 (2)
1000001011011100 (2)	0010011011011100 (3)	0111110111011100 (2,4)
0000000000111100 (4)	0101110000111100 (4)	1101110000111100 (3,4)
1110011000111100 (2)	0000111000111100 (3)	0111000010111100 (2)
1100001110111100 (2)	0011001110111100 (4)	0100010001111100 (2)
1100010011111100 (2)	0111100100000010 (4)	0010100110000010 (4)
1011001101000010 (3)	0111000111000010 (2)	1101010111000010 (3)
1010000000100010 (4)	0010100000100010 (3)	0011111000100010 (2)
1100001100100010 (4)	0010001100100010 (4)	0010011100100010 (3)
0110100010100010 (4)	0000010110100010 (4)	0010011110100010 (2)
0000000001100010 (2)	0101001101100010 (3)	10010101000100[illegible] (2)
0011100010010010 (4)	0100111010010010 (3)	1100000110010010 (4)
1001001110010010 (3)	1000100001010010 (3)	0111011101010010 (2,3)
0111000111010010 (3)	1110000100110010 (4)	0001110100110010 (4)
1010010 0110010 (2)	0010001010110010 (3)	0100100101110010 (3)
0100111101110010 (2)	1010000011110010 (3)	1000010011110010 (3)
0100111011110010 (2)	1101110111110010 (3)	0111011000001010 (3)
0010111000001010 (3)	1000010110001010 (3)	1110011110001010 (2)
1011101001001010 (4)	0111011001001010 (4)	1110111001001010 (4)
0100011011001010 (3)	1000010111001010 (2)	1010001000101010 (2)
0111000100101010 (4)	1101110100101010 (4)	0010011100101010 (3)
1101101010101010 (2)	1110011110101010 (4)	1001110011101010 (4)
1111101011101010 (2)	0000011111101010 (2)	0000101000011010 (4)
1100010001011010 (2)	1011000011011010 (4)	0001010011011010 (4)
1101001011011010 (4)	1110011111011010 (2)	0011110000111010 (4)
1010101110111010 (2)	1100100101111010 (3)	1111010101111010 (2)
0001001101111010 (2)	1000011000000110 (2,3,4)	1001100010000110 (3)
1110010011000110 (3)	0111111011000110 (2)	0111010000100110 (4)
0001011100100110 (2)	0001111100100110 (3)	0011010110100110 (3)
1000001110100110 (2)	1011100001100110 (3)	0111110001100110 (2)
1111001011100110 (3)	0000000111100110 (4)	1100111111100110 (2)
1100010000010110 (4)	0010110000010110 (3)	0100010100010110 (4)
0010100010010110 (4)	0001011010010110 (2)	0110110101010110 (3)
0110101101010110 (3)	1011011101010110 (2)	0110101011010110 (3)
0001011111010110 (3)	1100111100110110 (2)	1011111100110110 (4)

Length 16 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
0110101010110110 (3)	1110100110110110 (4)	1111011110110110 (3)
0111011001110110 (2)	1011000101110110 (2,4)	1000011011110110 (2)
1101111011110110 (4)	1001101111110110 (4)	1111000000001110 (2)
0011110100001110 (2)	0101010010001110 (4)	0010011010001110 (2)
0100001110001110 (2)	0100101110001110 (3)	0000011101001110 (2)
0110010000101110 (4)	0000000100101110 (2)	1101000.00101110 (1)
0101000001101110 (3)	0101001001101110 (4)	0110111001101110 (3)
0100101011101110 (2,3)	1110111011101110 (4)	1011111100011110 (2)
0100000010011110 (4)	1001000010011110 (2)	1110011010011110 (2)
1111001110011110 (2)	1010110001011110 (3)	1111100111011110 (4)
1011001111011110 (2)	1111001000111110 (3)	0110011000111110 (2)
1000000100111110 (4)	0000001100111110 (2)	1100010010111110 (2)
0001100110111110 (2)	0011101110111110 (2,4)	1110100101111110 (3)
0110001101111110 (2)	1101001011111110 (3)	0001000111111110 (3)
0000110010000001 (3)	0111110010000001 (4)	0000010101000001 (3,4)
0011101101000001 (2)	0010000011000001 (4)	1111101011000001 (2,4)
0110010111000001 (2)	1001001000100001 (3)	0100111100100001 (3)
0101000110100001 (3)	0101001110100001 (2)	0110000001100001 (2,3,4)
0000100101100001 (3)	0110111101100001 (2)	1011111101100001 (4)
1011000011100001 (2,3)	1011001111100001 (4)	0100101000010001 (3)
0001000100010001 (4)	1010010010010001 (4)	1101110010010001 (3)
1110011101010001 (3)	1010011011010001 (4)	0010111011010001 (2)
0001000100110001 (3)	1011000100110001 (2)	0001110100110001 (4)
1011101001110001 (3)	0001110011110001 (4)	1011001011110001 (3)
0010000000001001 (2)	0111100100001001 (2)	1111011100001001 (2)
1000010001001001 (3)	0000111101001001 (4)	0100100111001001 (3)
1001110111001001 (2,4)	1110011111001001 (3)	1110001000101001 (4)
0100100010101001 (2)	0011100010101001 (2)	0000100001101001 (2)
1111001011101001 (4)	0011000000011001 (4)	1101111000011001 (3)
0110000100011001 (3)	1111006100011001 (3)	1100110100011001 (4)
0010100110011001 (4)	1100101001011001 (3)	0010100111011001 (3)
1011100111011001 (3)	0110111111011001 (4)	1011000000111001 (2)
0001101100111001 (3)	0101011100111001 (4)	1001001110111001 (2,4)
1110001101111001 (2)	0000010011111001 (2)	1111110011111001 (2)
1101111011111001 (3)	0100010000000101 (4)	0011000100000101 (2)
0100111100000101 (3)	0101010001000101 (2)	1101100000100101 (2)

Length 16 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1000100100100101 (4)	0100110100100101 (2)	1110101100100101 (4)
1011100101100101 (2)	1111101011100101 (3)	1000101101100101 (4)
0000011011100101 (2)	0010111010010101 (2)	0001100001010101 (4)
0101110111010101 (2)	1110000000110101 (3)	1101000000110101 (4)
1011100000110101 (3)	0111101000110101 (3)	1100001100110101 (3)
0010[illegible]100110101 (3)	0001011100110101 (3)	0001060110110101 (4)
0000011111[illegible]10101 (4)	1101111111110101 (4)	1001110000001101 (2)
0101101100001101 (4)	1000011100001101 (2,3)	1111100010001101 (4)
1000110010001101 (2)	0001011010001101 (2)	0110111010001101 (2,4)
0000000110001101 (4)	1000111101001101 (3)	0100001011001101 (3)
0001111011001101 (4)	0111101111001101 (2)	1000011111001101 (4)
1101101010101101 (2)	0110101011101101 (2)	1010110000011101 (3)
0110011000011101 (3)	0010100100011101 (3)	1010011010011101 (2)
1001101110011101 (3)	1101100001011101 (2)	0101001001911101 (4)
1000111001011101 (3)	1101100611011101 (3)	1100100001111101 (2)
0111100011111101 (2)	0110110011111101 (4)	1000011011111101 (4)
0011101100000011 (2)	0100100110000011 (4)	1100110001000011 (4)
0100010011000011 (4)	1010110011000011 (3)	0011110111000011 (2)
1111111111000011 (4)	0110100000100011 (4)	1101110000100011 (2)
0101101000100011 (2)	1101100100100011 (3)	0111110100100011 (2)
0011111100100011 (2)	1110000101100011 (4)	1101110101100011 (3)
0001001011100011 (4)	1011111000010011 (2)	1110110100010011 (4)
0010111100010011 (2)	1110111010010011 (4)	0101111010010011 (3)
0001101110010011 (4)	1001101901010011 (3)	1101000101010011 (3)
0000111101010011 (3)	1100001000110011 (4)	1001100010110011 (4)
0110110011110011 (2)	0110011111110011 (2)	1010110000001011 (4)
0011100010001011 (3)	0111019010001011 (4)	1100101010001011 (3)
0001110110001011 (2)	0101101101001011 (4)	0111111101001011 (3)
0010111100101011 (3)	0010010010101011 (2)	0100001110101011 (3)
1110101101101011 (4)	1110010111101011 (2)	1010010000011011 (2)
0010110000011011 (4)	1011101000011011 (2)	1011101100011011 (3)
0000000010011011 (2)	1100010010011011 (3)	0000010001011011 (3)
0101010101011011 (2)	1011010101011011 (2)	1111000111011011 (2)
0001100000111011 (4)	1100010000111011 (2)	0011110000111011 (3,4)
1000100100111011 (3)	0010010100111011 (2,4)	0101010010111011 (4)
0000001010111011 (4)	1100011010111011 (3)	0100111110111011 (3)

Length 16 (continued)		
Initial LCAR (Number of Self-Concatenations of Initial LCAR)		
1001100001111011 (3)	0110111101111011 (4)	1001111101111011 (3)
1010111111111011 (4)	1010110000000111 (3)	0100110010000111 (4)
1100011010000111 (4)	1001010001000111 (4)	0010111001000111 (2)
1001111011000111 (2)	0011011000100111 (4)	0110001100100111 (3)
1101011110100111 (2)	0011110001100111 (2)	0111100101100111 (2)
0000010101100111 (2)	1900101011100111 (3)	0101000111100111 (2)
0101010111100111 (4)	0010001111100111 (4)	1001001111100111 (3)
0101101111100111 (2)	0111111010010111 (3)	0110110110010111 (4)
0010101113010111 (3)	0000011101010111 (4)	1010010011010111 (4)
1101011011010111 (4)	1100100010110111 (4)	1111101110110111 (3)
0010110001110111 (2)	0101001001110111 (4)	1100100101110111 (4)
0 0110110110111 (3)	0111011011110111 (4)	0111000000001111 (2)
1001100010001111 (3)	1101101110001111 (2)	0111110001001111 (3)
0110011101001111 (3)	1091011101001111 (4)	0111100111001111 (2)
0000011000101111 (4)	0101111010101111 (2)	0001011001101111 (4)
1001000011101111 (2)	1111110011101111 (4)	0110110111101111 (3)
1011000100011111 (4)	0001010100011111 (4)	0011000110011111 (4)
0111101110011111 (4)	0001000001011111 (2)	0010110101011111 (4)
1000001101011111 (2,4)	0101011101011111 (2)	0000100011011111 (3)
0001100111011111 (4)	1110110111011111 (3)	1111011100111111 (4)
1001111100111111 (2)	1010011010111111 (3)	1100001111111111 (4)

1.2 Non-self Primitive Concatenation of LCARs

Due to the very large number of possible concatenations of each length, we only include some examples here.

We list all length 5 primitive LCARs in the first row of the table. For each of the LCARs, we concatenate the rest of the LCARs to it and check the primitivity. The length 10 primitive LCARs on the second row are obtained from the non-self primitive concatenation of the LCARs in row 1. Similarly, the primitive LCARs of rows 3 are formed by the non-self primitive concatenation of the length 5 LCARs in row 1 and the 10 LCARs in rows 2.

engtli	Primitive LCARs
5	11110 10000 01100 11100 11000 11001
10	11110-01100 01100-11100 11100-01100 11000-11110 11000-11100 11000-11001 11001-11110
15	11110-01100-11100 10000-11100-01100 10000-11001-11110 01100-01100-11100 11000-11110-01100 11001-11110-01100 11001-11001-11110

1.3 Non-Self Primitive and Non-primitive Concatenation of LCARs

This lists a set of LCARs of length 16 constructed by both PP and PN concatenation from length 8 primitive LCARs. The LCARs with * are primitive.

Length 8 Primitive LCARs	Length 16 LCARs		
10100010	1010001000010001	1010001000010111	1010001000011100
	1010001000100010	1010001000101010*	1010001000110000
	1010001000111101	1010001001001110	1010001001011000
	1010001001100100	1010001001101001	1010001001111001
	1010001010000110	1010001010010100	1010001010101010
	1010001010101101	1010001011010100	1010001011010101*
	1010001011110001		
10101011	1010101100000001	1010101100110100	1010101101000101*
	1010101101001011*	1010101101001111	1010101101010000
	1010101101101110	1010101110110111*	1010101110011100*
	1010101110100101	1010101110101101	1010101110110000
	1010101110110101	1010101110111010*	1010101111000001
	1010101111000110	1010101111011010*	
11010010	1101001000101010*	1101001000101100	1101001001110100
	1101001001010000	1101001001110111*	1101001010000110
	1101001010010011*	1101001010100000	1101001010101010
	1101001010101101	1101001010110000	1101001011010101*
	1101001011011010*	1101001011011101	1101001011101111*
	1101001011110010	1101001011111001	1101001011111110
11010011	1101001100001101	1101001100010000	1101001100010110
	1101001100101000	1101001100110001	1101001100111110
	1101001101000000	1101001101000010	1101001101001010
	1101001101001101	1101001101110001	1101001101111100
	1101001101111111	1101001110101111	1101001110111011
	1101001111000100	1101001111001100	1101001111110010
	1101001111111001	1101001111111110	

Length 8 Primitive LCARs	Length 16 LCARs (continued)		
11010101	1101010100001100	1101010100100010	1101010100100100
	1101010100111011	1101010100111101	1101010101010101
	1101010101011101*	1101010101110010	1101010101110101
	1101010110000001	1101010110000110	1101010110001111
	1101010110010110	1101010110111001	1101010111000010
	1101010111111100		
11011010	1101101000011100	1101101000100000	1101101000101111
	1101101001010101	1101101001101001	1101101001110010
	1101101010000001	1101101010000110	1101101010001101
	1101101010101010	1101101010101101	1101101011111100
01000101	0100010100000001	0100010100010001	0100010100010110
	0100010100011000	0100010100110110*	0100010101001010
	0100010101001101	0100010101010100*	0100010101101000
	0100010110100000	0100010110110010	0100010111000001
	0100010111100100	0100010111101001	0100010111101111*
	0100010111110000*	0100010111111010*	
01001011	0100101100010100	0100101100100111	0100101100111011
	0100101101011010	0100101101100000*	0100101101100011
	0100101110001110	0100101110010001	0100101110101101
	0100101110110000	0100101111010010	0100101111010011*
	0100101111011101	0100101111101111*	0100101111111110

Appendix 2

CCMINI Manual Page

NAME

ccmini — a concurrent checking code generator and minimizor.

SYNOPSIS

ccmini [-h[elp]] [-b[erger]] [-p[arity]] [-m3] [-c[ombinational]]
[-s[equential]] [-r[pattern]] [-m7] [-l[pattern]] [<filename>]

DESCRIPTION

Ccmini is capable of adding berger code, Mod3/Mod7 code and LFSR/LCAR to output of benchmarks circuits (or other circuits), and parity code to input of the circuits. Minimization is automatically carried out after above codes are added. The format of input file is compatible to that of espresso. "Don't cares" can appear in input or output or the both. A statistic information about the given circuit can be found in a *.rpt file.

OPTIONS

-b[erger]	add berger code to output of the circuit.
-c[ombinational]	for combinational circuit.
-h[elp]	gets ccmini user manual.
-l[pattern]	add LFSR code whose divisor polynomial is gi :en by *pattern* in binary form.
-m3	add mod3 code to output of the circuit.
-m7	add mod7 code to output of the circuit.
p[arity]	add parity code to input of the circuit.
r[pattern]	add LCAR code whose LCAR structure is given by *pattern* in binary form.
-s[equential]	for sequential circuit.
<filename>	input file name.