A KNOWLEDGE BASED SYSTEM

FOR DESIGNING

TESTABLE VLSI CIRCUITS

by

Magdy S. Abadir

Technical Report CRI-86-11

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

December 1985

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90089

This dissertation, written by

MAGDI S. ABADIR

under the direction of h.is...... Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

Dean of Graduate Studies

Date February 13, 1986

DISSERTATION COMMITTEE

Chairperson

DEDICATION

To
Samia and Adam

ACKNOWLEDGEMENTS

I would like to express my deepest sense of appreciation to Professor Melvin A. Breuer for his encouragement, guidance, and support during the course of this thesis. I consider it my privilege to have worked with him. His scholarship has been a constant source of inspiration, and his invaluable constructive criticisms have greatly enhanced the quality and presentation of this thesis.

I would also like to thank Professor Alice Parker and Professor Walt Scacchi for serving on my dissertation committee.

I take this opportunity to thank my parents for their endless love, care and support throughout my life.

My heartfelt thanks go to all my friends and colleagues in the Department of Electrical Engineering for their warm friendship and continuous help throughout my graduate study.

I gratefully acknowledge the financial support received from the National Science Foundation (Grant MCS-8203485), and from the Defense Advanced Research Projects Agency (Contract No. N00014-84-K-0649.)

Last but not least, my sincere gratitude goes to my best friend, companion, and wife Samia for her endless love and support. She was always there when I needed her. It was her undying faith in me that enabled me to complete my doctorate.

ABSTRACT

The complexity of VLSI circuits has significantly increased the need for design for testability (DFT). Numerous techniques for designing more easily testable circuits have evolved over the years, with particular emphasis on built-in testing approaches. What has not evolved is a design methodology for evaluating and making choices among the numerous existing approaches. This thesis describes a knowledge based system which can be used for designing testable VLSI circuits.

First, a model for capturing circuit designs which incorporates all the relevant aspects of a design from the testing point of view is introduced.

Next, a methodology which incorporates structural, behavioral, qualitative and quantitative aspects of known DFT techniques is presented. A frame model for representing testable design methodologies is then introduced. This methodology framework provides a designer with a systematic design for testability synthesis approach.

The process of partitioning a design into subcircuits which can be processed individually is examined next. A three-phase partitioning scheme is presented. The new concept of an I-path, which is used to transfer data from one place in the circuit to another, is introduced. This simple, yet powerful concept represents an important departure from previous efforts in the DFT field.

The mechanism and rules that govern the process of applying testable design methodologies to circuit partitions are presented. Measures for evaluating and comparing different testable designs are also developed.

The concept of a test plan which describes how a test methodology is to

execute in an actual circuit is introduced. A theory of test plan execution overlap is presented, and used as the basis for constructing test schedules with optimal execution times.

A global strategy for controlling the operation of the system is presented. This strategy employs a heuristic for ordering the processing tasks in order to enhance the effectiveness and efficiency of the system.

A prototype of the system described in the thesis has been implemented in LISP. Experimental results dealing with several case studies were very encouraging. Sample results of one case study are presented.

Table of Contents

1. Introduction 1
1.1. Background 1
1.2. Problem Characteristics and Motivations 3
1.2.1. Previous Work 6
1.3. Proposed Tasks 8
1.4. Thesis Outline 9
2. A Model of the Design 12
2.1. introduction 12
2.2. A View of the CUC 12
2.3. Modeling the CUC 15
2.3.1. Model Requirements 15
2.3.2. The Graph Model 16
2.3.3. Modeling Bit-sliced Array Structures 19
2.4. Summary 25
3. Testability Knowledge Engineering 26
3.1. Introduction 26
3.2. Testable Design Methodology 26
3.2.1. The Structure to be Tested 27
3.2.2. Internal BIT Components 27
3.2.3. External Components 28
3.2.4. The Operational Aspect of a TDM 28
3.2.5. TDM Evaluation Measures 28
3.3. A Frame Model for TDMs 30
3.3.1. The TDM Structural Template 31
3.3.2. The TDM Test Schema 34
3.3.3. The TDM Measures 36
3.4. Summary 42
4. Partitioning the CUC 43
4.1. Motivation 43
4.2. Statement of the Problem 44
4.3. Kernel Identification 45
4.3.1. Top-down Partitioning 47
4.3.2. Flat Design Partitioning 47
4.3.3. Bottom-up Clustering 48
4.3.3.1. Combinational Clusters 49

4.3.3.2. Sequential Clusters 51
4.4. Forming the Kernel Subgraph 53
4.4.1. Data Transfer Paths 53
4.4.2. Constructing SG_k 59
4.5. Summary 61
5. TDM Selection and Embedding **63**
5.1. Introduction 63
5.2. Embedding TDM Frames into Kernel Subgraphs 63
5.2.1. TDM Embedding for a Single Input Port/Single Output Port Kernel 64
5.2.1.1. Formal Definition of A Feasible Embedding 67
5.2.1.2. Generating a Test Plan for an Embedding 71
5.2.1.3. Creating New Structures in a Circuit 72
5.2.2. TDM Embedding for Kernels with Multiple Input and Output Ports 78
5.2.2.1. Generating a Test Plan for a Multiple Port Kernel 79
5.3. Embedding Measures 89
5.3.1. Circuit-related Measures 91
5.3.1.1. Local Circuit Measures 91
5.3.1.2. Global Circuit Measures 95
5.3.2. TDM-related Measures 99
5.3.2.1. Local TDM Measures 100
5.3.2.2. Global TDM Measures 102
5.4. Evaluating Embeddings 103
5.4.1. Scoring Functions 104
6. Test Plan Scheduling **105**
6.1. Introduction 105
6.2. Exploring Parallelism in Test Plans 106
6.2.1. Cyclic Behavior of the Execution Schedules of Test Plans 108
6.2.2. The Conflict Graph 109
6.2.2.1. A Process for Constructing the CG of a Test Plan 110
6.2.3. Finding Feasible Values for D 110
6.3. Lower Bounds on D 111
6.4. Inserting No-Op Steps 113
6.5. Conflicts Caused By No-Ops 117
6.5.1. Modifying The Conflict Graph 119
6.6. Test Plans for Scan-Type TDMs 120
6.7. Scheduling Test Plans of Multiple Kernels 122
6.7.1. Combining Test Schedules of Different Kernels 123
6.7.2. Combining Test Plans of Multiple Kernels 123
6.8. Remarks About Pipeline Optimization 124
6.9. Summary 126

7. A Global Strategy for TDES **127**
7.1. Introduction 127
7.2. Global Control Schema 128
7.3. Ordering the Kernels 131
7.4. Kernels Tested for Free 133
7.4.1. Testing Multiplexers and Busses 133
7.4.2. Testing Registers 136
8. A Prototype System **137**
8.1. Introduction 137
8.2. Circuit Description Environment 137
8.3. Testability Knowledge Capturing Environment 138
8.4. The Testability Synthesis Environment 140
8.4.1. The Top Level Menu 141
8.4.2. The CUC Menu 142
8.4.3. The TDM Menu 143
8.4.4. The Parameters Menu 143
8.4.5. The Embedding Menu 147
8.5. An Example 150
9. Conclusions and Future Research **159**
9.1. Summary 159
9.2. Future Research 161
9.2.1. Developmental Tasks 161
9.2.2. Conceptual Problems 162
Appendix A. Examples of TDM Frames **166**
A.1. The BILBO TDM Frame 166
A.2. Serial/Serial Random TDM Frame 168
A.3. Serial/Parallel Random TDM Frame 171
A.4. The Exhaustive TDM Frame 174
A.5. The Scan Path TDM Frame 176
A.6. The SPLASH TDM Frame 178
A.7. The Scan TDM Frame for Multiplexers 180
A.8. The Scan TDM Frame for Shift Registers 183
A.9. A Counter TDM Frame 185
References **187**

List of Figures

Figure 2-1: A circuit modeling example (a) The circuit (b) Its graph model (c) The graph of node X. 17
Figure 2-2: Examples of node labels. 20
Figure 2-3: An n bit-sliced adder. 21
Figure 2-4: Modeling a 16-bit-sliced adder (a) pictorial view of node representation of 16-bit sliced adder (b) labels of a 16-bit sliced adder node (c) pictorial view of a slice (d) node representation of a single slice (e) the arcs of the FA-graph. 23
Figure 3-1: The structural template of the BILBO TDM frame. 32
Figure 3-2: The test schema of the BILBO TDM frame. 35
Figure 3-3: Some of the measures associated with the BILBO TDM frame. 37
Figure 3-4: An estimator function for determining the number of random patterns required to achieve a certain fault coverage. 41
Figure 4-1: Various ways of forming sequential clusters. 52
Figure 4-2: Examples of structures with various I-modes 55
Figure 4-3: α/β I-path P(S1:X → S2:Y) (a) the trivial case (b) the recursive case. 58
Figure 4-4: (a) A P/S I-path P(C:X → R3:Q) (b) its activation plan. 60
Figure 4-5: Illustrating the subgraph concept. 62
Figure 5-1: A circuit example for illustrating the embedding process. 69
Figure 5-2: A feasible BILBO embedding (a) matching data (b) the test plan. 70
Figure 5-3: Adding structures to a circuit (a) kernel subgraph (b) inserting a register (c) inserting a multiplexer. 74
Figure 5-4: Different schemes for creating an I-path between two structures. 75
Figure 5-5: A register feedback circuit case. 78
Figure 5-6: A loop formed by the frontier steps of q I-paths. 84
Figure 5-7: A circuit example with a multiple port kernel. 85
Figure 5-8: The CPCG of the activation plans of the input I-paths. 86
Figure 5-9: The CPCG of the activation plans of the output I- 88

paths.
Figure 5-10: Embedding evaluation measures. 90
Figure 5-11: The PD measure versus the introduced delay t as a function of the freedom of the modified structure τ_s. 96
Figure 6-1: Various execution schedules for a generic 5 step test plan. 107
Figure 6-2: An execution schedule with 2-phase schedule cycle. 108
Figure 6-3: The CG for Example 6.2. 111
Figure 6-4: An optimal execution schedule (D=2) obtained using No-Op steps. 113
Figure 6-5: The CG of Example 6.6 (plus node 6') 117
Figure 6-6: An example with a potential for conflicting No-Ops. 118
Figure 6-7: The Extended CG (ECG) of Example 6.8. 120
Figure 6-8: The structural template of the Scan Path TDM. 121
Figure 6-9: Inserting a delay 'd' in a reservation table. 125
Figure 7-1: A flowchart of a global control schema for TDES. 129
Figure 8-1: The top level menu of TDES1. 141
Figure 8-2: The CUC menu. 142
Figure 8-3: The TDM menu. 144
Figure 8-4: The parameters menu. 145
Figure 8-5: The embedding menu. 147
Figure 8-6: (a) The original CUC (b) BILBO embedding for the PLA (c) exhaustive embedding for the ROM (d) scan path embedding for the ROM (e) the modified circuit. 158
Figure 8-7: Feasible embeddings for the PLA as produced by TDES1. 158
Figure 8-8: Detailed data of the best embedding for the PLA. 158
Figure 8-9: Detailed data of the best embedding for the ROM. 158
Figure 8-10: Detailed data of the best scan path embedding for C. 158
Figure A-1: The BILBO TDM frame (a) the structural template (b) the test schema (c) BILBO measures. 166
Figure A-2: The serial/serial random TDM frame (a) the structural template (b) the test schema (c) some measures. 169
Figure A-3: The serial/parallel random TDM frame (a) the structural template (b) the test schema (c) some measures. 172
Figure A-4: The Exhaustive TDM frame (a) the structural template (b) the test schema (c) some measures. 174
Figure A-5: The scan path TDM frame (a) the structural template (b) the test schema (c) measures. 176
Figure A-6: The SPLASH TDM frame (a) the test schema (b)SPLASH measures. 179

Figure A-7: The frame of the scan TDM for MUXs (a) the structural template (b) the test schema (c) measures. 181

Figure A-8: The scan TDM frame for shift registers (a) the structural template (b) the test schema (c) measures. 183

Figure A-9: A counter TDM frame (a) the structural template (b) the test schema(c) some measures. 185

Table

Table 5-1: Different mapping combinations 70

Chapter 1
Introduction

1.1. Background

VLSI offers a host of opportunities for significant advancements in future electronic systems. However, along with the benefits comes a major bottleneck to the manufacturing of VLSI and to the products that use them: the time and cost required for testing. The key problem is that the increasing complexity and density of VLSI circuits lead to ever-worsening accessibility of internal logic points and to greater complexity and time associated with the generation and grading of test patterns. Such a dilemma exists in part because progress in integrated circuit technology is occurring at a faster rate than advancements in test technology. One very promising discipline that deals not only with the limited accessibility problem, but also the costly test generation and test problems, is **design for testability** (DFT).

Since the early 1970's a variety of **ad hoc** guidelines for designing easily testable circuits have been proposed, mostly dealing with SSI/MSI circuits. Examples of these guidelines are: the use of flip-flops with resetting capability, prohibiting the use of clock signals that cannot be controlled externally, and the placement of test points to increase accessibility [Hayes 73, Hayes 74, Buehler 82].

Extensions to these early ad hoc methods have led to more **structured** techniques, such as LSSD [Eichelberger 77], Scan Path [Funatsu 75], Scan/Set [Stewart 77], and Random Access Scan [Ando 80]. The basic concept

employed by these techniques is the incorporation of a second mode of operation, namely a test mode, in which flip-flops are chained together to form a shift register so that the state of the circuit can be easily set or interrogated. This eliminates the necessity of testing complex sequential networks as well as making it possibly to segment large combinational networks. During the test mode, test data is usually furnished by external test circuitry, and response data analyzed by external test circuitry, such as automatic test equipment (ATE).

With the increase in chip density, the concept of circuits with **built-in self-test** (BIST) has emerged. BIST is defined as the capability of a circuit (chip, multichip assembly, or system) to test itself, with input stimulation or output evaluation, or both, being integral to the circuit and not requiring external test equipment. Many BIST techniques have been proposed, such as BILBO [Konemann 79], self verification [Sedmak 79, Sedmak 80], syndrome testing [Savir 80], autonomous testing [McCluskey 81], and store and generate [Agrawal 81]. The basic strategy in all BIST techniques is to add special **built-in test** (BIT) structures to the original circuit so that during the test mode the circuit can carry out various go/no-go testing tasks. Examples of BIT structures include counters, pseudo random number generators, multiplexers(MUXs), ROMs, and signature analyzers. BIT structures are added in such a way that they do not affect the circuits original behavior, except for a possible degradation in performance.

Another strategy for approaching the VLSI testing problem suggests designing with easily testable components. This strategy attacks the problem at the early stages of the design process by enforcing the use of certain implementation styles to synthesize the design. Examples of these easily testable components are EX-OR trees [Seth 77], Canonic Reed-Muller circuits [Saluja 75], easily testable PLAs [Ostapko 78, Dong 81, Fujiwara 81], and easily testable iterative logic arrays [Friedman 73, Sridhar 81].

1.2. Problem Characteristics and Motivations

Although DFT techniques offer the potential to alleviate the significant problems associated with testing VLSI circuits, its benefits do not come without cost. Applying a DFT technique to a circuit may take its toll in one or more of the following ways.

- Additional pin and silicon area overhead.
- Decreased reliability due to the increased silicon area.
- A performance impact due to additional circuitry.
- Additional design time and cost.

A sound determination of whether to employ DFT concepts for a particular circuit can only be made in the context of a total life cycle assessment cost of both the positive and negative cost factors. To minimize the penalties discussed above it is necessary to incorporate the DFT discipline with the functional design processes. Hence, DFT becomes a designer's responsibility and not the responsibility of test engineers.

A designer whose goal is to design an "easily" testable VLSI chip is faced with a serious dilemma. First, he may not be aware of all the DFT techniques which exist, and secondly, he may not know all the ramifications of these techniques when used in his particular circuit. It is important to note that most DFT techniques are only applicable to particular types of logic. For instance, there are more than 20 DFT techniques dealing with the design of easily testable PLAs [Ostapko 78, Dong 81, Fujiwara 81, Breuer 85a]. These techniques take advantage of the regular structure of PLAs and their unique failure mechanisms. However, these techniques cannot be applied to other forms of combinational circuits. Clearly, the task of just designing a testable PLA is not trivial and may require expert assistance [Breuer 85a].

A VLSI chip usually consists of various types of logic structures, such as PLAs, RAMs, and ROMs, each having different fault modes and detection mechanisms. Thus different DFT techniques may have to be used for different parts of the design. This in turn leads to questions regarding the compatibility of various techniques and how the chip's architecture affects the selection process.

Each DFT technique can be associated with a set of measures reflecting the various costs and gains associated with that technique, such as area overhead, extra number of I/O pins, cost of ATE required, test time, and degree of fault coverage. Unfortunately, the value of most of these measures are not easy to estimate, and often they are functions of the circuit to be made testable.

Another aspect which adds to the complexity of the problem is the fact that, in general, there are several ways of applying a particular DFT technique to one part of the circuit, each way leading to different values of the measures of interest. For example, to test a ROM using a BIST check-sum approach, one of the requirements is a way of stepping through the entire address space of the ROM. Solutions include using a counter, a linear feedback shift register (LFSR) implementing a primitive polynomial [Peterson 72], or an adder. For the LFSR option, one can either add it to the design with a multiplexer feeding the ROM, or one can modify one of the circuit registers which has access to the ROM input to behave as an LFSR during the testing mode. Clearly, there might be several candidate registers, each leading to a different solution with different area overhead, test time and performance degradation.

Realizing the complexity and richness of the VLSI DFT problem, a key question arises: can this wealth of knowledge be embodied into a CAD tool which can aid a designer in creating easily testable VLSI chips? The problem has the following characteristics:

1. There is a large body of knowledge that has to be considered. This includes knowledge about circuit objects, knowledge about testing methodologies, and hard-to-represent design knowledge about goals, constraints, trade-offs and actions.

2. Constraints imposed on the design come from many sources. Unfortunately, there is no comprehensive theory that integrates constraints with design choices. Thus it is not possible to immediately assess the consequences of design decisions. Hence, there is a need for exploring tentative design possibilities.

3. Partitioning is essential to cope with problem complexity. The boundaries and the interactions between subproblems have to be identified . Provisions should be made in order to escape from points in the solution space that are only locally optimal.

4. Most DFT techniques were developed for particular kinds of circuits. Very little is known about how to apply these techniques to complex circuits, and about the ramifications of combining various DFT techniques.

5. It is not feasible to exhaustively explore all possible ways making a VLSI chip testable. First, there are often many DFT techniques that can be applied to a given part of a chip. Secondly, for each technique there are usually many ways to implement it. Hence, there is a need for an intelligent decision making process to guide the system through the large search space.

Clearly, the task of designing an easily testable chip can be characterized as an expert task. Attempting to emulate the knowledge and behavior of a testability expert by a CAD tool is very challenging, since it is very hard to formally describe the methodology he or she employs when designing a testable chip.

1.2.1. Previous Work

The fields of testing and design for testability are very rich and we will not try to survey the state of the art in these fields. The interested reader can refer to surveys in [Breuer 76, Muehldorf 81, Williams 79, Williams 82] and the latest proceedings of the Fault Tolerant Computing Symposiums and the International Test Conferences.

Recently, the design and use of so called **expert systems** has received a great deal of attention. An expert system is a problem-solving computer program that can reach a level of performance comparable to that of a human expert in some specialized problem domain. What distinguishes an expert system from ordinary application programs is the fact that the problem-solving methodology is explicitly modeled as a separate entity or **knowledge base** rather than appearing implicitly as part of the program code. Expert systems has been developed for a number of different problem domains such as bacterial infection, oil exploration, and computer configuration. In addition languages are being developed to provide tools for creating expert systems. A good survey of the state of the art of expert systems can be found in [Nau 83].

Several researchers have suggested the use of knowledge-based expert systems to aid in problems related to testing and design for testability.

Bellon, et al. [Bellon 83] discuss an intelligent assistant program for automatic test pattern generation (ATPG), called "Supercat". The program employs a data base of various ATPG algorithms with information such as their fault coverage, complexity, and applicable type of circuits. The user can choose one of two models to represent the circuit under test, a data flow model and a control flow model. Their work is directed toward test generation rather than DFT.

Hortsmann [Hortsman 83] discusses a system written in PROLOG that checks a design for LSSD rules violation. The design structure is represented as a set of logical clauses. The main emphasis is to identify the latches and their interconnections. The latches interconnections are then passed through a number of tests to check for any LSSD rules violation. Clearly, the scope of this work is limited to the LSSD methodology.

Mangir [Mangir 83] has suggested the use of a data base and an expert system written in PROLOG for the design of testable VLSI circuits. The data base would contain information about various built-in-self-test techniques reflecting their costs of implementation and their applicability at various levels of testing, such as system, board, and chip. Her ideas are primarily a proposal. The use of logic programming to represent knowledge suffers from various problems. Most important, the lack of an explicit scheme to index into relevant knowledge, the awkwardness of handling changing or incomplete knowledge, and the perceived limitations of deductive inference.

Several MIT researchers [Davis 82, Davis 83, Hamscher 83a, Hamscher 83b] and others [Hartley 84], are developing expert systems for fault diagnosis. Given that a circuit is faulty, the problem is to identify which part of it is faulty. The MIT system uses structural and behavioral description of a circuit, modeled as two graphs, to derive a diagnostic for the circuit. The methodology used is an extension of the concept of path sensitization [Breuer 76]. Clearly, their work is not directly related to ours.

Agrawal and Jain [Agrawal 84] reported a CAD tool called TITUS (testability implementation and test generation using scan). TITUS automatically enforces the scan path design rules into a circuit, and generates the required test patterns. There are limitations to the size of circuits which TITUS can handle and clearly it is limited to the scan path methodology.

Finally, researchers at GTE Labs [Fung 85] proposed a system called TEXPERT (testability expert). TEXPERT is to be used in a silicon compilation environment. Their proposed methodology will combine the use of a test bus, BILBO and incomplete scan path to produce testable designs. Even though their goals are very similar to ours, their research is still in the preliminarily stages. The two major subblocks of TEXPERT the testability expert and the testability evaluator, have not yet been designed.

1.3. Proposed Tasks

The main objective of this thesis is to develop a knowledge based expert system, called TDES (Testable Design Expert System), to aid a designer in creating easily testable VLSI chips [Breuer 84, Abadir 85a, Abadir 85b, Abadir 85c, Breuer 85a]. TDES is a part of the Advanced Design AutoMation system (ADAM) currently under development at USC [Granacki 85]. Before discussing the details of our research, the following scenario illustrates the general role of TDES and how it fits into the ADAM system.

Assume one desires to build a VLSI chip which implements a well defined algorithm or function. In addition, assume that the chip has to be easily testable. An early step in the design process is to translate the behavioral specifications into a design consisting of the interconnection of structures, such as PLAs, registers, busses, RAMs, and ROMs. This RTL design description is then passed to TDES, together with design goals and constraints. The knowledge base of TDES contains information about DFT techniques and methodologies. The knowledge base also contains rules that describe how to apply these test methodologies to the circuit under consideration (CUC), as well as mechanisms for evaluating trade-offs between the various solutions. Using its knowledge base, TDES will carry out the process of modifying the design such that the resulting circuit can be easily tested and satisfies specified goals and constraints. TDES will automatically insert the required hardware

structures that are needed to enhance the testability of the design without changing its normal behavior. At this point the final logical synthesis of the CUC can be carried out, and the resulting design can be laid out and fabricated.

In accordance with the broad goals of TDES, we will examine the following problems.

- Construct a model for VLSI circuits which accommodates aspects of a design data which are important from the testing point of view.
- Develop a framework for a methodology which can be used for representing all the relevant aspects of known DFT techniques. This framework should serve as the basic building block of the testability knowledge base of TDES.
- Develop a scheme for partitioning a design into components that can be tested independently.
- Develop a mechanism for exploring all the various ways of applying a DFT methodology to a design component and for evaluating the various solutions which may exist.
- Develop a global strategy for the selection of DFT methodologies for various design components.

1.4. Thesis Outline

Chapter 2 identifies the design level at which we are approaching the problem. The key aspects of a design data which are important from the test point of view are identified and the terminology used to describe design information is presented. A hierarchical graph model of the design data of the CUC is then described.

In Chapter 3 we develop the concept of a testable design methodology (TDM) [Breuer 84] which incorporates structural, behavioral, quantitative, and

qualitative aspects of known DFT techniques, and which can provide a designer with a systematic design for testability synthesis approach. A frame model for representing TDMs is then presented. TDM frames are used as the main source of testability knowledge in the knowledge base of TDES.

In Chapter 4 we present a three-phase process for partitioning the CUC into components which can then be processed individually. These components are referred to as kernels. During the first phase, the CUC model is traversed top-down to explore potential kernels. In the second phase a flat description of the CUC is used to identify the kernels. Finally, in the third phase a bottom-up clustering process is carried out to cluster circuit structures together to produce complex kernels. The new concept of I-modes and I-paths is presented and used to identify circuit components that can potentially be useful in testing a given kernel.

In Chapter 5 the mechanism and rules that govern the process of applying TDMs to a circuit kernel is presented. Techniques for exploring all the feasible ways of making a kernel testable using one of the available TDMs are illustrated. Various evaluation measures which are used to compare the various solutions are developed, and a scheme for selecting a solution for a particular kernel based on these measures and, which takes into consideration the global side effects is then presented.

In Chapter 6 we introduce the concept of a test schedule which incorporates pipelining concepts in the execution of a testing process associated with a particular realization of a TDM in a real circuit. Theoretical lower bounds on the test time associated with such TDM realization are presented. Techniques are then developed to construct test schedules which optimizes test time.

In Chapter 7 we describe a global strategy for managing the operation of TDES. Various heuristic provisions for enhancing the quality of the global solutions produced by TDES are discussed.

In Chapter 8 we briefly describe a prototype implementation of TDES. The user interface with that system and a case study is presented.

A thesis summary is provided in Chapter 9, and topics for future research are discussed.

Chapter 2
A Model of the Design

2.1. introduction

The purpose of this chapter is twofold. First we outline our view of the circuit under consideration (CUC). This includes identifying the building blocks that are commonly used to build a chip and the level of abstraction at which we want to deal with the design. We also identify aspects of the design data which are important from the viewpoint of test.

Secondly, we present a model for representing the design of the CUC. The model is very general and accommodates the relevant attributes of a design need for test.

2.2. A View of the CUC

Assume one desires to build a VLSI chip that implements a well defined algorithm or function. In addition assume that the chip has to be easily testable. One step in the design process is to translate the functional specifications of the algorithm into a design consisting of structures like PLAs, registers, busses, RAMs, and ROMs (RTL level design). At this point we can assume that a floor plan for the chip exists, and that we have a good estimate of the area occupied by every structure and routing channel in the design. Some but not all of the internal logic of these structures may be known. At this stage, the design is passed to TDES to transform it into an easily testable design. Before describing the design model, we will first identify the important aspects of a design at this level of abstraction.

There are three fundamental units of logic used to implement digital systems at the RTL level of abstraction, namely combinational logic, registers, and random access memories (RAMs).[1]

Definition 2.1: A circuit component that can be classified as either combinational, register, or RAM defines a **basic structure** [Breuer 84].

Definition 2.2: A **structure** is the interconnection of one or more basic structures.

Definition 2.3: A **maximal basic structure** is a basic structure not contained within a larger basic structure.

For example, an adder can be classified as combinational hence it is a basic structure. Similarly, a shift register is a basic structure of type "register." On the other hand, a microprocessor structure cannot be classified under any basic structure category. Note that all basic structures are structures while the converse is not true.

There are various ways of implementing a basic structure. These variations deal primarily with the style of constructing and interconnecting transistors and the technology being used. These variations are referred to as **design styles** [Breuer 84]. For example, there are four common design styles for implementing combinational logic, namely PLAs, ROMs, combinational gate networks (CGN), and random combinational logic. The latter style indicates an implementation that cannot be classified under one of the first three styles.

The existence of a basic structure can often be associated with a design

---

[1]In [Breuer 84] busses were recognized as the fourth fundamental unit of logic. In our view a bus with the control circuitry that decides which structure places its output on the bus lines are treated as a combinational logic structure that functions like a multiplexer.

style. Structures with different design styles have different failure mechanisms and hence they are often tested differently. For example, techniques for testing PLAs often require adding rows and columns to the PLA to test for extra or missing crosspoints [Breuer 85a]. Such techniques are not applicable to a ROM or a NAND network implementing the same functions of the PLA. Hence, it is important to identify the design styles of basic structures. Also, certain **architectural** features which structures in the CUC might possess, like bit-slicing, should be identified as they may be applicable to unique test methodologies.

The function of a basic structure can play a role in testing that structure or even testing other structures. For example, a multiplexer (MUX) can be easily tested functionally, and can also be useful in gaining accessibility to other structures. However, this is not true in all cases. For example, the particular function realized by a combinational structure is normally ignored when using an exhaustive or a random testing approach. Though the behavior of every basic structure is assumed to be known, the behavior of arbitrary complex structures consisting of the interconnection of basic structures is usually not assumed to be explicitly available.

Register structures play a very important role in the design of VLSI chips. Their contents define the state of the CUC at any given time, hence it is important to have access to these registers. Some registers have very limited capabilities and they can only be used to hold data. Others, like shift registers and counters, have additional functional capabilities. For example, a BILBO register has four modes of operation: latch parallel inputs, shift right, clear, and generate pseudorandom (or exhaustive) patterns [Konemann 79].

One approach to model these register structures is to view them as simple latches connected to some combinational logic and treat the two structures

separately. However, in doing so we ignore functionality, which makes testing these two structures and the surrounding ones much more difficult. Hence, we view these multi-mode registers as basic structures, with high emphasis on their functional capabilities.

Another factor that motivated our approach with registers is the fact that most DFT techniques utilize multiple mode registers. These registers can either be added to the CUC, or existing registers can be modified to accommodate the additional capabilities needed. Clearly, the way we view registers makes this modification process a very easy task.

In summary, we see that different basic structures with different design styles have unique characteristics and are thus amenable to unique testing approaches. The functions realized by basic structures may also imply special testing treatment. Register structures and their functional capabilities play a key role in testing the CUC.

2.3. Modeling the CUC

2.3.1. Model Requirements

To guide the process of selecting a good representation model for the CUC, we will first identify the key attributes that the model should possess.

- The model should reflect a good structural view of the circuit. Thus, it should identify the structures that form the circuit and how they are interconnected. It should also identify the basic structures that form more complex structures.

- The design styles of basic structures have to be identified. Also, certain architectural styles, like bit-slicing, should be identified.

- The model should reflect certain functional attributes associated with structures.

- The model should be suitable for automation. In other words it should be convenient to store and process the model on a digital computer.

- Finally, the model should be flexible and easy to modify to reflect the changes that might be made in the design in order to make it testable.

2.3.2. The Graph Model

In view of the above requirements, a **directed graph model** is used to represent the CUC [Abadir 85a]. **Nodes** are used to represent structures, and **arcs** represent interconnections between the structures. Every node has a number of input, control and output **ports** from which it can receive or send signals. The model is **hierarchical** in the sense that a node that represents a complex structure (as opposed to a basic structure) is itself represented by another graph that identifies its substructure. Thus, at the top level of the hierarchy, the CUC can be viewed as a single node with ports representing the circuit primary inputs and outputs (the chip I/O pins). As we go down the hierarchy nodes represent simpler structures until we reach the level at which every node represents a basic structure.

For example, consider the circuit of Figure 2-1a and its graph model shown in Figure 2-1b. Control signals and primary inputs and outputs are not shown in these figures. All the nodes in the graph model correspond to basic structures, except for node X, which is described by its own graph model in Figure 2-1c.

Various kinds of **labels** are attached to nodes in order to describe certain **attributes** of the corresponding structures, such as the design style, number of inputs and outputs, chip area, and functional capabilities. Labels are attribute-value pairs, and new labels can be attached to elements as required. For example, Figure 2-2 shows some of the labels attached to four of the nodes of

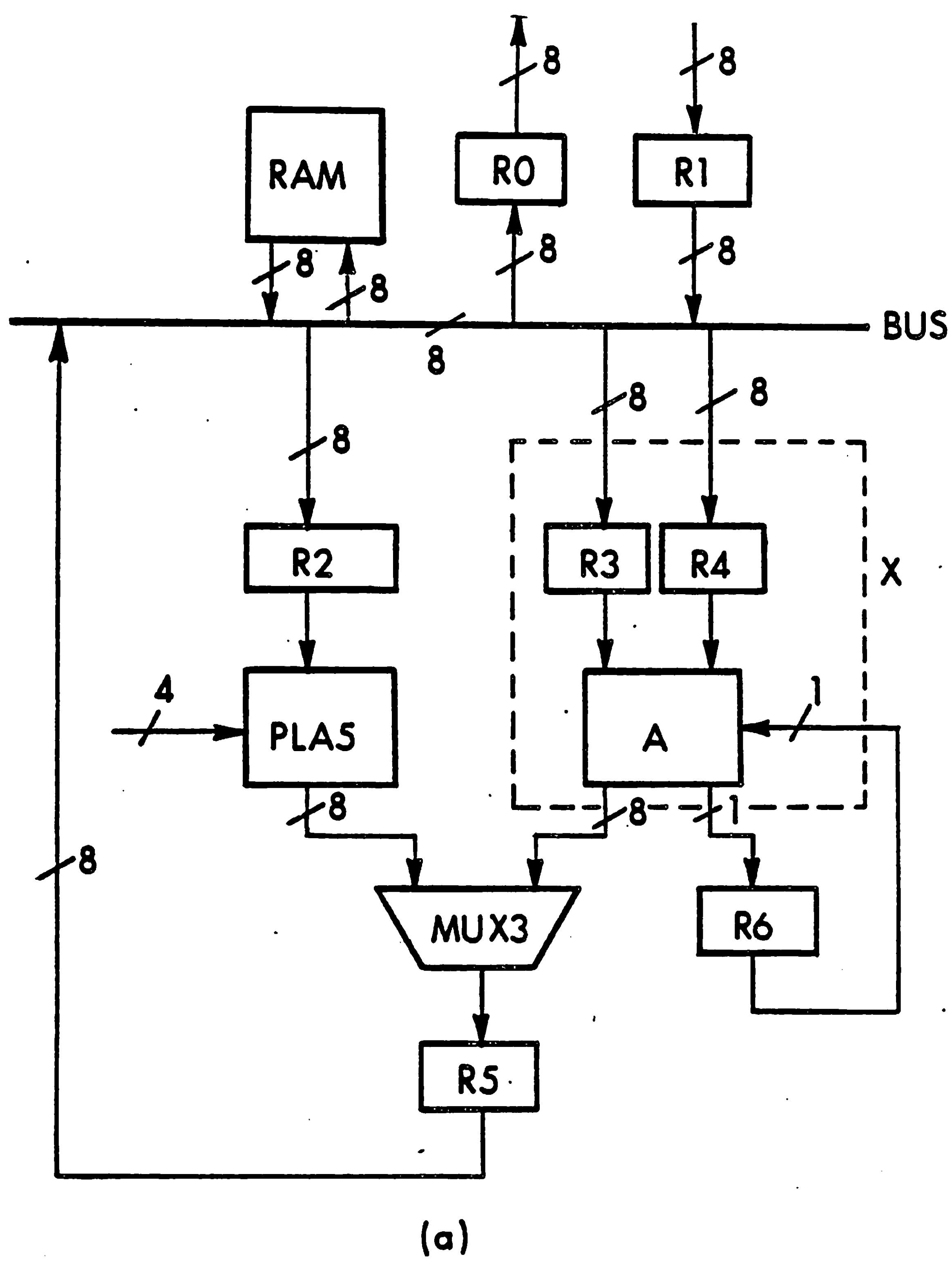


Figure 2-1: A circuit modeling example (a) The circuit (b) Its graph model (c) The graph of node X.

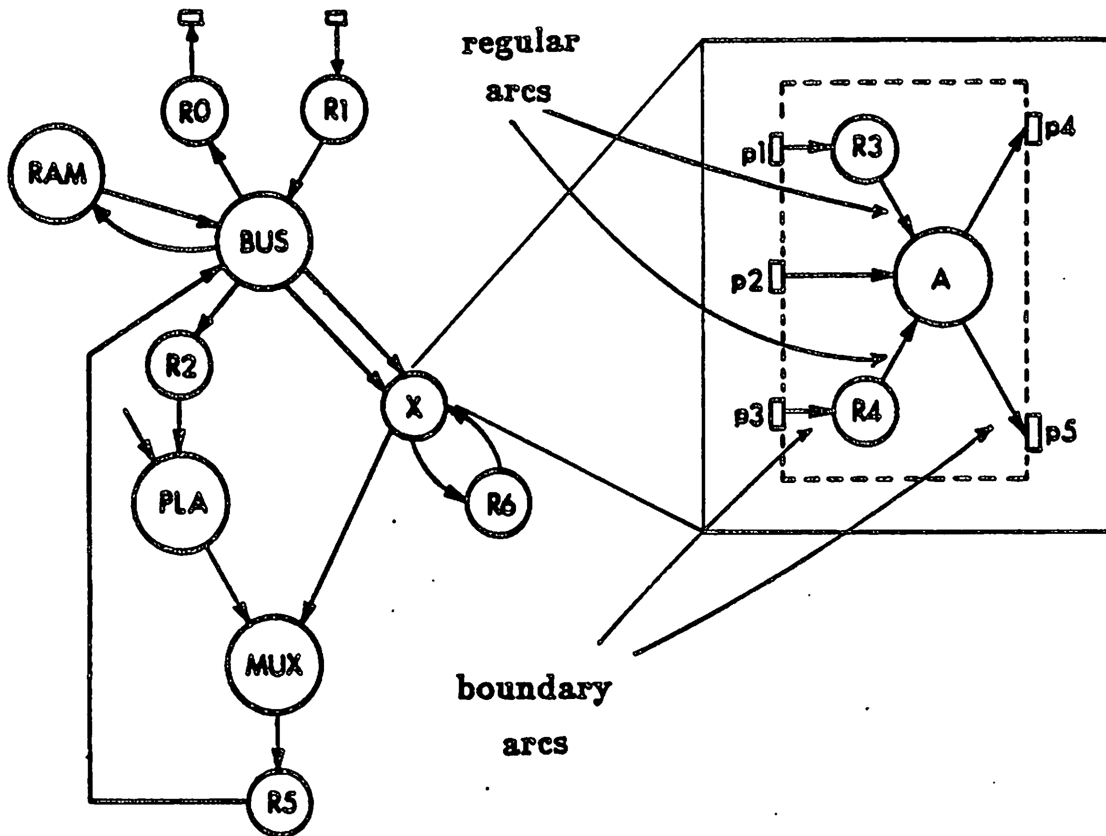


Figure 2-1: (continued).

the graph of Figure 2-1b. Note that different kinds of attributes are attached to different types of nodes. More important, because of the hierarchy, certain attributes of a node can be inferred from the attributes of its parent node. For example, one can deduce that one structure is combinational from the fact that its parent node is combinational. It is interesting to note the similarities between our hierarchical graph model and the semantic network model and IS-A hierarchies [Woods 75, Brachman 83].

An arc between two ports of two nodes indicates a physical interconnection between the associated structures. Labels are also attached to the arcs between nodes to identify attributes such as the source, destination, width and type of interconnection. The width of an arc represent the number of physical wires that constitute the interconnection.

In addition to the **regular** type of arcs which represent interconnections between nodes of one graph at some level of the hierarchy, there are other types of arcs which are supported by this model. For example, **boundary** arcs are used to describe how the ports of node X are internally connected to the ports of its substructures as shown in Figure 2-1c.

2.3.3. Modeling Bit-sliced Array Structures

Modeling bit-sliced array structures in a concise way requires special handling. For example, consider an n-bit adder constructed using n 1-bit full adders as shown in Figure 2-3. To model such an array structure, it is enough to describe one slice together with all the relevant interconnections.

For a general bit-sliced one dimensional array structure consisting of n identical slices, there are 4 different types of interconnections that need to be modeled.

Definition 2.4: An interconnection between an array boundary port of

```
Node = R1
 Type : Register
 Size : 8
 Modes: Clear, Hold, Parallel
        latch, Count-Up.
        ...........

Node = PLA5
 Type : Combinational
 Design Style: PLA
 Input-Ports: p1= 8, p2=4
 Output-Ports: p3=8
 # of product terms: 150
 Function: personality matrix # 5
 .....................

Node = MUX3
 Type : Combinational
 Design Style: Random Logic
 Function: multiplexer
 Input-Ports: p1=8, p2=8
 Output-Ports: p3=8
 ................

Node = X
 Type: Complex node
 Contents: (R3, R4, A)
 Input-Ports: p1=8 p2=1 p3=8
 Output-Ports: p4=8 p5=1
 Function : Complex
 Bit-Sliced: No
 ..............
```

Figure 2-2: Examples of node labels.

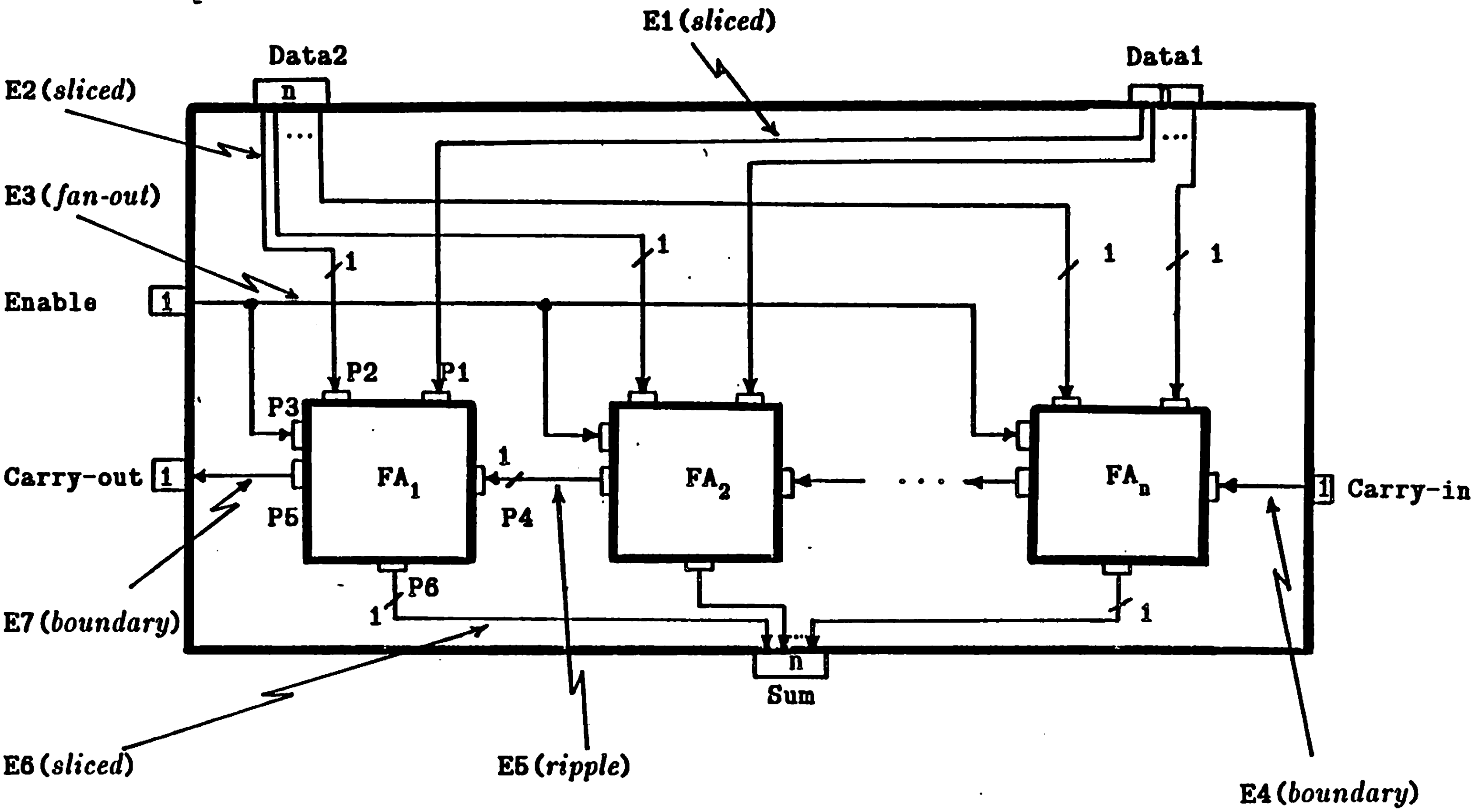


Figure 2-3: An n bit-sliced adder.

size n × w and n internal ports each of size w is defined as a **sliced** interconnection. Hence, a sliced interconnection directs (takes) different bits of data to (from) different slices. Examples are the interconnections labeled E1, E2, and E6 in Figure 2-3.

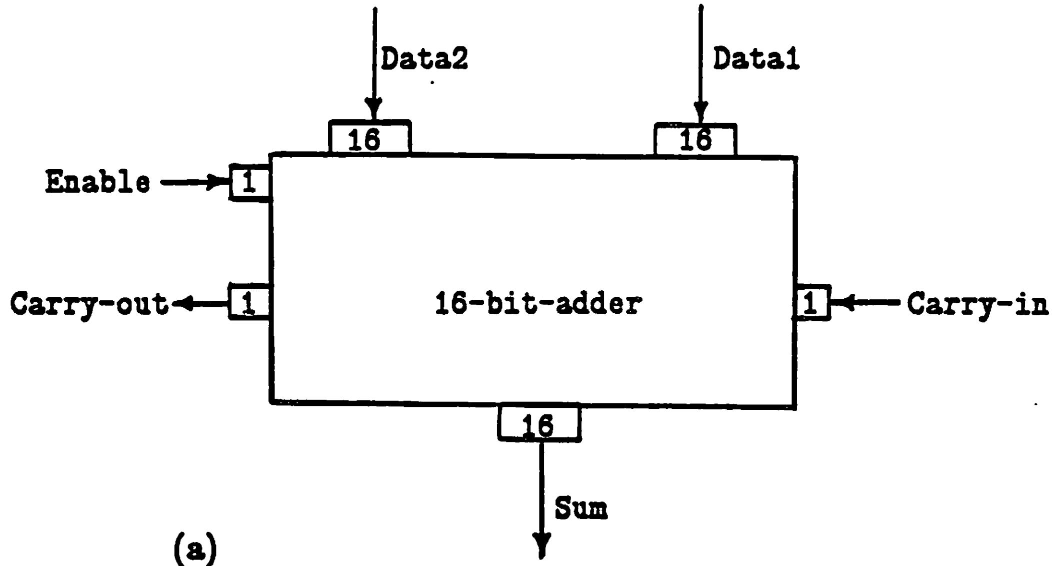
Definition 2.5: A **Fan-out** interconnection directs the same b bits of data from an array boundary port to one of the input ports of every slice. The enable signal interconnection, E3, is an example of a fan-out interconnection.

Definition 2.6: An interconnection between an output port of one slice and an input port of the adjacent slice is called a **ripple** interconnection. An example is the rippling carry signal interconnection E5 in Figure 2-3.

Definition 2.7: An interconnection between a boundary port and a port of either the first or the last slice is called a **boundary** interconnection. E4 and E7 in Figure 2-3 are two examples of this kind of interconnections.

Figure 2-4 describes how to model a 16-bit adder consisting of 16 1-bit slices. The node illustrated in Figures 2-4a and b represents the bit sliced array structure. The substructure of that node is described by a graph called FA-graph. The FA-graph consists of one node representing a single full adder slice, as shown in Figures 2-4c and d, together with 7 arcs as shown Figure 2-4e. Note that the full adder node inherits both its type and design style from its parent node.

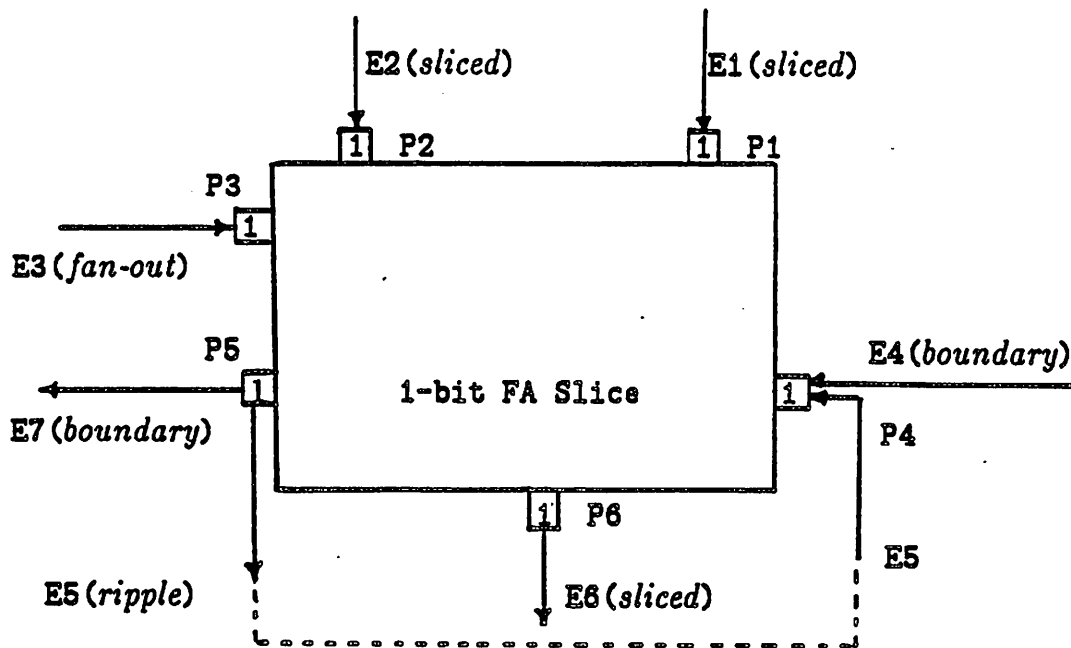
One can easily see the flexibility of our model. For example, assume one desires to model the 16-bit adder as four 4-bit adders where each 4-bit adder is constructed using four 1-bit slices. The hierarchy of the new model has three-levels as opposed to two levels. At the highest level a single node similar to the one in Figure 2-4b is used. However the number of slices is 4 instead of 16. At the third level of the hierarchy the FA-graph can be used without any changes. At the second level of the hierarchy, a graph similar to the FA-graph can be




```
Type: Combinational
Design Style: CGN
Input-ports:  Data1=16,
              Data2=16,
              Carry-in=1,
              Enable=1
Output-ports: Sum=16,
              Carry-out=1
Architectural Style: Bit-sliced
# of Slices: 16
Substructure graph: FA-graph
Function: 16-bit adder
      ...........
```

(b)

Figure 2-4: Modeling a 16-bit-sliced adder (a) pictorial view of node representation of 16-bit sliced adder (b) labels of a 16-bit sliced adder node (c) pictorial view of a slice (d) node representation of a single slice (e) the arcs of the FA-graph.



(c)

```
    Input-ports:  P1=1, P2=1
                  P3=1, P4=1
   Output-ports:  P5=1, P6=1
       Function:  full-adder
                  .............
```

(d)

Arc ID	Arc Type	From	To	Width
E1	Sliced	Data1	P1	1
E2	Sliced	Data2	P2	1
E3	Fan-out	Enable	P3	1
E4	Boundary	Carry-in	P4	1
E5	Ripple	P5	P4	1
E6	Sliced	P6	Sum	1
E7	Boundary	P5	Carry-out	1

(e)

Figure 2-4: (continued) Modeling a 16-bit-sliced adder.

used. The only difference is that the size of the input and output ports are different as well as the width of the arcs. Note that it is easy to automate the process of adding a new level of abstraction to an existing bit-slice model, which might be useful for testing purposes. For example, testing the 16-bit adder as one unit might be hard, while testing every slice individually might be too costly. An acceptable solution might be to partition the adder into two 8-bit adders and test each one separately.

2.4. Summary

In this chapter we have presented the basic model used for describing designs to be processed by TDES. The model is very general and accommodates aspects of design data which are important from the viewpoint of TDES. Additional attributes can be added as required so that the task of testability synthesis can be carried out.

Chapter 3
Testability Knowledge Engineering

3.1. Introduction

As noted in Chapter 1, numerous techniques for designing more easily testable circuits have evolved over the years, with particular emphasis on built-in testing approaches. What has not evolved is a design methodology for evaluating and making choices among the numerous existing approaches.

In this chapter we describe our efforts in constructing a knowledge base which incorporates structural, behavioral, qualitative and quantitative aspects of known DFT techniques, and which supports a systematic design for testability synthesis process. First we identify the basic elements involved in the process of designing an easily testable circuit and testing it. We also discuss various measures and criteria that are commonly used to evaluate various DFT techniques. Next, we present a unified framework within which testability techniques may be represented.

3.2. Testable Design Methodology

A **testable design methodology** (TDM) [Breuer 84, Abadir 85a, Abadir 85b, Abadir 85c] deals with the complete process of (1) designing an easily testable structure, (2) developing the test programs, where appropriate, and (3) testing the structure using external and/or built-in test hardware. Examples of well known TDMs are scan-path, LSSD, scan/set, BILBO, syndrome testing, and autonomous testing [Williams 82]. Over 20 TDMs exist for PLAs alone [Breuer 85a]. Following is a list of the major components that constitute a TDM.

3.2.1. The Structure to be Tested

We refer to the structure to be tested by a TDM as a **kernel**. Obviously, every TDM can only be applied to certain types of kernels. For example, the BILBO TDM can be applied to a combinational kernel of style PLA, ROM, CGN, or random, but it is not applicable to RAMs.

Often the kernels of certain TDMs should satisfy a number of constraints or possess special features. For example, some TDMs associated with bit sliced iterative logic arrays [Sridhar 81] can only be applied to kernels which possess special testability features, namely C-testability and I-testability. Another example is the syndrome TDM [Savir 80] which is effective only when applied to combinational blocks that are specially designed for syndrome testing.

3.2.2. Internal BIT Components

Various BIT structures are employed by TDMs to carry out one or more of the following tasks:

1. Generation of the test stimuli.
   Examples are counters, ROMs, and BILBO registers.

2. Processing the test responses.
   Examples are comparators, transition counters, and BILBO registers.

3. Gaining access to the internal structures of the circuit.
   This is by far the most important task of BIT structures. The goal here is to be able to control the inputs (observe the outputs) of the structure under test from the test generation site (response evaluation site). Examples are scan/set registers, LSSD registers, and MUXs.

4. Controlling the testing process.
   Test strategies that avoid the use of external ATE require built-in test controllers to supervise the testing process and to supply the appropriate control signals that will drive the other BIT structures.

5. Transforming the logic of a structure to make it easily testable. Examples are adding rows and columns to a PLA to make it universally testable [Fujiwara 81], and modifying an iterative logic array (ILA) to be CI-testable [Sridhar 81].

It should be noted that some BIT structures can perform more than one of the above tasks. For example a BILBO register can be used for tasks 1, 2, and 3.

3.2.3. External Components

1. Software components necessary to create the test data off-line such as test generators, fault simulators, and ATE program generators.

2. Automatic testing equipment (ATE) needed to supply the test data and the required control signals to the CUC at testing time.

3.2.4. The Operational Aspect of a TDM

The three previous TDM components that we have discussed dealt primarily with the structural aspect of a TDM. Another important aspect deals with the operation of a TDM and how the test will be carried out. Most TDMs assume that testing will be done while the CUC is operating off-line in an explicit testing mode. Others employ a concurrent testing approach in which testing is done concurrently while the circuit is carrying out its normal tasks. Clearly, for every TDM belonging to the former class of TDMs, there must be some form of a procedure that specifies the type of actions to be carried out during the test mode of a kernel in order to execute the test.

3.2.5. TDM Evaluation Measures

One can associate with every TDM a set of measures that collectively reflects the various implementation costs of the TDM as well as the testability gains achieved. Below we list the important measures that are often used to characterize a TDM:

1. Initial test creation cost.

This measure reflects the cost of acquiring and executing software modules to carry out tasks which are needed to create the test data. Examples of these tasks are test generation, fault simulation, signature evaluation, and ATE program generation. Clearly, this cost measure is incurred only once during the lifetime of a circuit.

2. ATE cost.
   This measure reflects the size and complexity of the tester needed by a TDM.

3. Area overhead.
   This measure indicates the increase in chip area due to the added BIT structures or due to modifications made to circuit kernels. The increase in chip area due to BIT structures can have negative effects on the yield, which is the percentage of good chips that are produced in a certain chip processing environment.

4. Degradation in performance due to of the added hardware.
   This measure indicates the effect, if any, of the additions made to the design on the speed of the circuit during normal operation. Clearly, introducing new levels of logic in the design results in additional delays that might reduce the circuit speed.

5. I/O signals demand.
   This measure indicates the number of I/O signals that have to be added to the existing ones to furnish additional test signals as required by the TDM.

6. Testing application time.
   This is the time it takes to execute the test. It is directly related to the number of test patterns used, and also to the rate by which we can apply test vectors to the kernel. Such a measure is very critical to manufacturer testing of high volume chips. It is also very important in maintenance testing of systems with expensive down-time (i.e., time when the system is not available for normal use).

7. Degree of fault coverage with respect to a particular fault model.
   This measure is normally the key indicator of how successful the testing process operates.

The measures listed above are highly interrelated, and often trade-offs

are made between costs and gains or between different kinds of costs. For example, the degree of fault coverage achieved using LSSD strategy is a function of the number of test patterns used. Hence there is a trade-off between fault coverage and test time. Another classical trade-off is the one between initial test creation and ATE costs associated with TDMs, such as LSSD and scan-path, and high area overhead often associated with BIST methodologies, such as exhaustive and BILBO. Evaluating these trade-offs and deciding on the best approach clearly depends on the circuit itself and the goals and constraints of the design.

3.3. A Frame Model for TDMs

In constructing a knowledge based system, such as TDES, a good solution often depends on a good representation of the knowledge involved [Nilsson 71, Woods 75]. Knowledge representation has been recognized by AI researchers as a very challenging problem. The most important current approaches to represent knowledge are semantic networks and IS-A hierarchies [Woods 75, Brachman 83], first order logic [Nilsson 71], frames [Minsky 75], and production or expert systems [Nau 83].

Our philosophy in organizing and representing the testability knowledge lends itself to the use of **frames** [Minsky 75, Winston 77], which are data structures for representing stereotyped situations or concepts. When one of these standard concepts is recognized, slots (frame variables) inside the appropriate frame can be filled in with tokens representing the actual actors or actions. After this step, precompiled knowledge can be gleaned directly from the frame.

Every TDM is modeled using a frame [Abadir 85a]. The information slots in a TDM frame can be classified into three major groups. The first deals with the structural aspect of the methodology and how the structures involved

are interconnected. The second deals with the operational aspect. The last group of slots contain the values of the various measures associated with the methodology. The components of a TDM frame are described in detail in the next sections. Appendix A contains several examples of TDM frames.

3.3.1. The TDM Structural Template

The **template** of a TDM describes its structural architecture. It conveys information about the type, style, and size of the kernel to which the TDM is applicable. It also describes the BIT structures needed by the methodology and the connection paths that must connect them to the kernel.

A graph model similar to the one used for modeling the CUC is used to describe the design template part of a TDM frame. The structural template graph of a TDM consists of a kernel node plus a number of nodes representing the BIT structures required. Figure 3-1 shows the template for the BILBO TDM [Konemann 79]. The template indicates that the BILBO TDM employs two linear feedback shift registers B1 and B2. The first plays the role of a **driver** of the kernel inputs which functions as a pseudo random number generator. The second register plays the role of a **receiver** of the kernel response which can function as a signature analyzer.

The signature evaluator module can be realized in several ways. It can either be on-chip or off-chip. In the former case, it can be a comparator which is a part of an on-chip controller, or it can be implemented as a decoder which is a part of the signature analyzer. In the off-chip case, an ATE is required and the signature can either be transferred in parallel or in serial from B2 to the external output pins of the circuit.

Labels are attached to template nodes describing their different attributes, and arcs are used to model interconnections which must exist

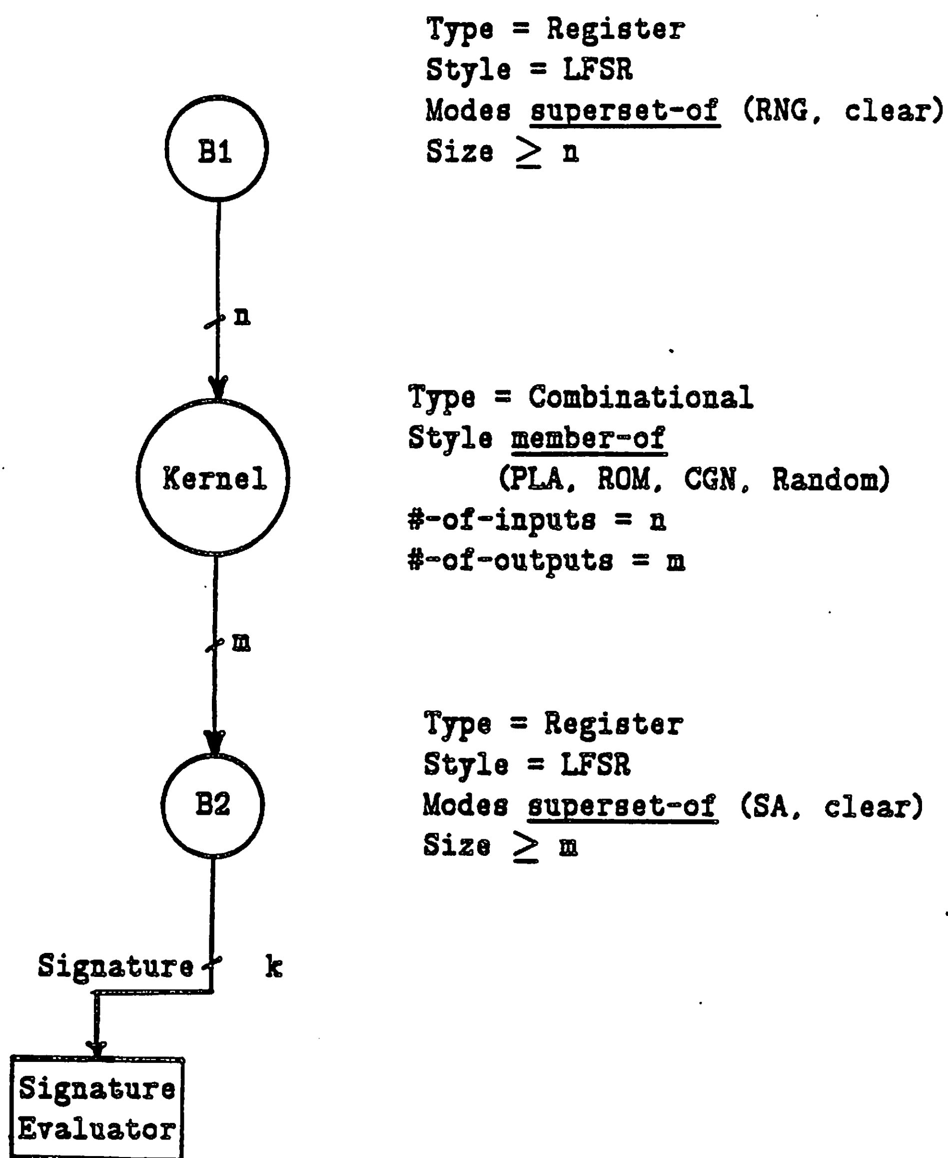


Figure 3-1: The structural template of the BILBO TDM frame.

between the various nodes. Even though a TDM template graph looks very similar to circuit graphs, there are two key differences. First the labels attached to a template node are more general than their counterparts of circuit graphs. As we described earlier in Chapter 2, labels associated with a circuit node are attribute value pairs. However, labels attached to a template node are triplets consisting of (1) an attribute name, (2) a **qualification-function**, and (3) a value or a list of values. For example, the kernel node of the BILBO TDM has two labels. The first, (type equal combinational), indicates that the BILBO TDM requires a kernel of type combinational. The second label, (design-style member-of (PLA, CGN, ROM, random)), indicates that the BILBO TDM is applicable to structures whose design style is either a PLA, CGN, ROM, or random.

Various kinds of labels can be attached to a template node to specify certain functional or structural features which the node should have. For example, attaching the label (#-of-inputs less-than 30) to the kernel node of the exhaustive TDM template indicates that this TDM is only applicable to kernels with less than 30 inputs. Another example, is attaching the label (modes superset-of (clear, signature-analysis)) to the receiver of the BILBO template. This indicates that the receiver node of the BILBO TDM should have the two specified modes of operation among its capabilities.

We will refer to labels attached to template nodes as **qualification labels**. It is interesting to note that the labels attached to circuit nodes can be regarded as a special case of qualification labels where the qualification function has always the value "equal."

The second key difference between template graphs and circuit graphs is in the use of arcs. An arc in the graph of a TDM template has a broader meaning than just a connection consisting of one or more wires. It represents a

data transfer path between two nodes. Such a path can be as simple as a wire, or it can be a complex path through a number of busses, registers and MUXs. This simple yet powerful concept represents an important departure from previous descriptions of TDMs. This concept is discussed in detail in the next chapter. Several examples of TDM templates are illustrated in Appendix A.

3.3.2. The TDM Test Schema

A **test schema** describes how a test methodology is to execute. In general, a test schema consists of three sections: a head, a body, and a tail. The body of a test schema describes the on-chip actions that constitutes the life cycle of a single test vector from the time it is generated until its effects on the kernel have been captured and processed. The main aspects specified in the body of a test schema are: (1) the generation of test vectors; (2) the transfer of the test vectors to the kernel; (3) the propagation of the test vectors through the kernel; (4) the transfer of response data to some response analysis circuit; and (5) the processing of the response data. In addition to the actions in the body of a schema which must be executed once for every test vector, a test schema often has a head (tail) section in which initialization (closing) actions are specified.

There are two kinds of actions that constitute a test schema: data transfer actions and data processing action.

Definition 3.1: A **data transfer action** calls for transferring n-bits of data (without modifying it) between two structures. The format of a data transfer action is as follows:

Transfer (source $\xrightarrow{n}$ destination)

Definition 3.2: A **data processing action** calls for propagating data through a structure. If that structure has different modes of operation, one of these modes is specified in the action. The format of a data processing action

is as follows:

Structure (mode of operation)

As an example consider the BILBO test schema shown in Figure 3-2. The body of the schema has to be executed T times, where T is the number of test vectors to be used. First the input BILBO register, B1, is clocked while in the random number generation (RNG) mode to produce a new test vector. Next the output of B1 is transferred to the kernel inputs, the test vector then propagates through the kernel logic, and finally the response is transferred to the second BILBO register, B2, which is clocked while in the signature analysis (SA) mode. The head and tail sections are both self explanatory.

```
Head
   B1 ( clear )
   B2 ( clear )

Body
   Execute T times:
       B1 ( Random Number Generation )
       Transfer ( B1 output -n-> Kernel input )
       Kernel ( - )
       Transfer ( Kernel output -n-> B2 input )
       B2 ( Signature Analysis )

Tail
   Transfer ( B2 output -k-> Signature Evaluator)
```

Figure 3-2: The test schema of the BILBO TDM frame.

The actions in the body of a test schema has to be performed in the specified sequence. However, it is often feasible to initiate one iteration before the end of previous iterations. This issue will be discussed in detail in Chapter 6. Examples of TDM test schemas are provided in Appendix A.

3.3.3. The TDM Measures

The last group of slots in a TDM frame contains values for the measures which reflect the various costs of implementing the methodology as well as its merits. Examples of these measures were given earlier in Section 3.2.5.

There are two primary reasons why specifying a priori the values of many of these measures is a difficult task. First, quantifying many of the attributes, like the ATE or test creation software cost, is not easy. Second, and more importantly, many of the measures are functions of the kernel and of the CUC. For example, predicting the test application time or the fault coverage when using the BILBO TDM is not easy without carrying out fault simulation. Moreover, the circuitry surrounding the kernel that will be used to control (observe) its inputs (outputs), will have an effect on the test application time. Also the area overhead for employing BILBO depends a great deal on the type and location of the registers in the CUC, and hence on the architecture of the CUC.

The previous example uncovers a very important issue, namely that one can apply the same TDM to a circuit kernel in more than one way resulting in different testable designs with different measures. This issue will be explored in detail in Chapter 5. The actual values of all TDM measures can only be calculated after all the slots in the TDM template and test schema have been filled by actual actors as a result of applying the methodology to a real circuit.

However, rather than leaving most of the measure slots in a TDM frame empty, functions or typical estimated values are used instead. The estimates are used as bounds to make early global design decisions between alternative classes of TDMs. In general, the values of the measures will take one of three forms: (1) numeric values, (2) functions (estimators), and (3) comments.

Next, we list the names of some of the measures in a TDM frame and the kind of values expected for each measure. Figure 3-3 illustrates the value of some of the measures associated with the BILBO TDM. Appendix A contains descriptions of the measures associated with several other TDM frames.

```
Test creation
  software     : (fault-simulation, signature-evaluation)
ATE            : 2
Area overhead  : ( Area(n-bit LFSR) - Area(n-bit register))
                 + ( Area(k-bit LFSR) - Area(k-bit  register))
                 ≈ (n+k) × (3000-1200)λ² [Newkirk 83].
Performance degradation:
                 set-up delay(LFSR) - set-up delay(register)
Extra I/O signals: 3; can be less if control is shared.
Test length    : random-test-estimator-function
Test time      : ≈ test length × D, where D is a measure
                   of the speed of processing the test schema.
Fault coverage:  (single-stuck-at 100×(1 - 2^-k) )
Storage requirement: k bits
    ...............
    ...............
```

Figure 3-3: Some of the measures associated with the BILBO TDM frame.

Test Creation Software

A list of the main software packages employed by a TDM is stored in the test creation software slot. Keywords such as, test-generation, fault-simulation, and signature-evaluation are typical entries. For example, Figure 3-3 indicates that fault simulation is required by the BILBO TDM to grade the quality of the random patterns used. Also signature evaluation is required to calculate the value of the correct signature. Note, however, that it is quite possible to eliminate the use of either software programs. For example, it is possible to use an estimator function to predict the fault coverage instead of fault simulation [Savir 83, Jain 85]. In addition, fault coverage may be ignored and only the probability of detecting an error considered [Smith 80]. Also the correct signature can be determined via hardware measurements of actual chips.

ATE

The ATE field in a TDM frame indicates the relative size and complexity of the ATE required by that TDM to supply, observe and/or control the test. An integer between 0 and 10 is required as a value for this measure, where 10 represents the most expensive and complex ATE while 0 represents no ATE requirement. For example, the BILBO TDM usually requires a relatively simple ATE to supply the necessary control signals required to execute the test process. In addition, the ATE usually employed by BILBO is responsible for checking the correctness of the final signature. However, one should note that it is possible to eliminate the need for an ATE completely in the BILBO TDM by using an on-chip controller to supply the necessary control signals and to perform the necessary signature checking.

Area Overhead

As we already mentioned, the area overhead associated with a particular TDM is a function of the CUC. Later in Chapter 5 we will discuss in detail the various factors involved in determining the extra area required to apply a TDM to a particular circuit. Instead of leaving the area overhead field unspecified in a TDM frame, a rough estimate of the extra area typically associated with a TDM is used. For example, as shown in Figure 3-3, a typical value of the BILBO area overhead measure corresponds to modifying two regular registers (ones which can latch and hold data) to become LFSRs.

Performance Degradation

The extra delay in normal operation resulting from hardware modifications required to apply a TDM in a particular circuit is used as a measure of the performance degradation associated with that TDM. Again it is clear that such a measure is very dependent on the circuit and its timing characteristics. Hence, a typical value is used in a TDM frame rather than an exact value. For example consider the BILBO TDM frame. The extra set-up delay typically introduced along the normal input path of a register as a result

of modifying it to become an LFSR is used as the performance degradation factor, as shown in Figure 3-3.

Extra I/O Signals

The extra I/O signals measure indicates the number of extra signal lines required to employ a TDM in a particular circuit. Clearly, there is an obvious dependence on the circuit and on the kind of hardware modifications which need to be carried out. A typical value for this measure for the BILBO TDM is three: two for the extra control signals used for the LFSRs; and one for scanning out the signature. If several BILBO TDMs are used on a chip, this overhead cost can be shared among them. This subject will be discussed in more detail in Chapter 5.

Test Length

The test length slot of a TDM frame contains the name of an estimator function that when invoked returns an estimate of the size of the test set that should be employed to achieve a certain degree of fault coverage. A test length estimator function often employs parameters such as

1. the number of kernel inputs and outputs,
2. the number of gates in the kernel or some measure of the kernel size and complexity,
3. the type and design style of the kernel, and
4. the degree of fault coverage required.

Figure 3-4 illustrates a function that can be used to estimate the number of random test patterns required in the BILBO TDM to achieve a certain degree of fault coverage. T_{100} indicates the number of test patterns required to get 100% single-stuck-at fault coverage. Note that the function assumes that at $0.1 \times T_{100}$, 80% fault coverage is attained. As shown in Figure 3-4 fault coverage is assumed to grow linearly between the origin and the 80% point, and also between the 80% and 100% points. Given a fault coverage value, one

can use this function to obtain an estimate of the test length required. The value of T_{100} is a function of a parameter called the random-susceptibility-factor which reflects how susceptible a kernel is to random testing [Savir 83, Jain 85, Williams 85, Abadir 85d]. This parameter can either be supplied by the user and included as one of the kernel attributes, or it can be deduced from special tables using data about the kernel type, design style, size, and complexity [Abadir 85e].

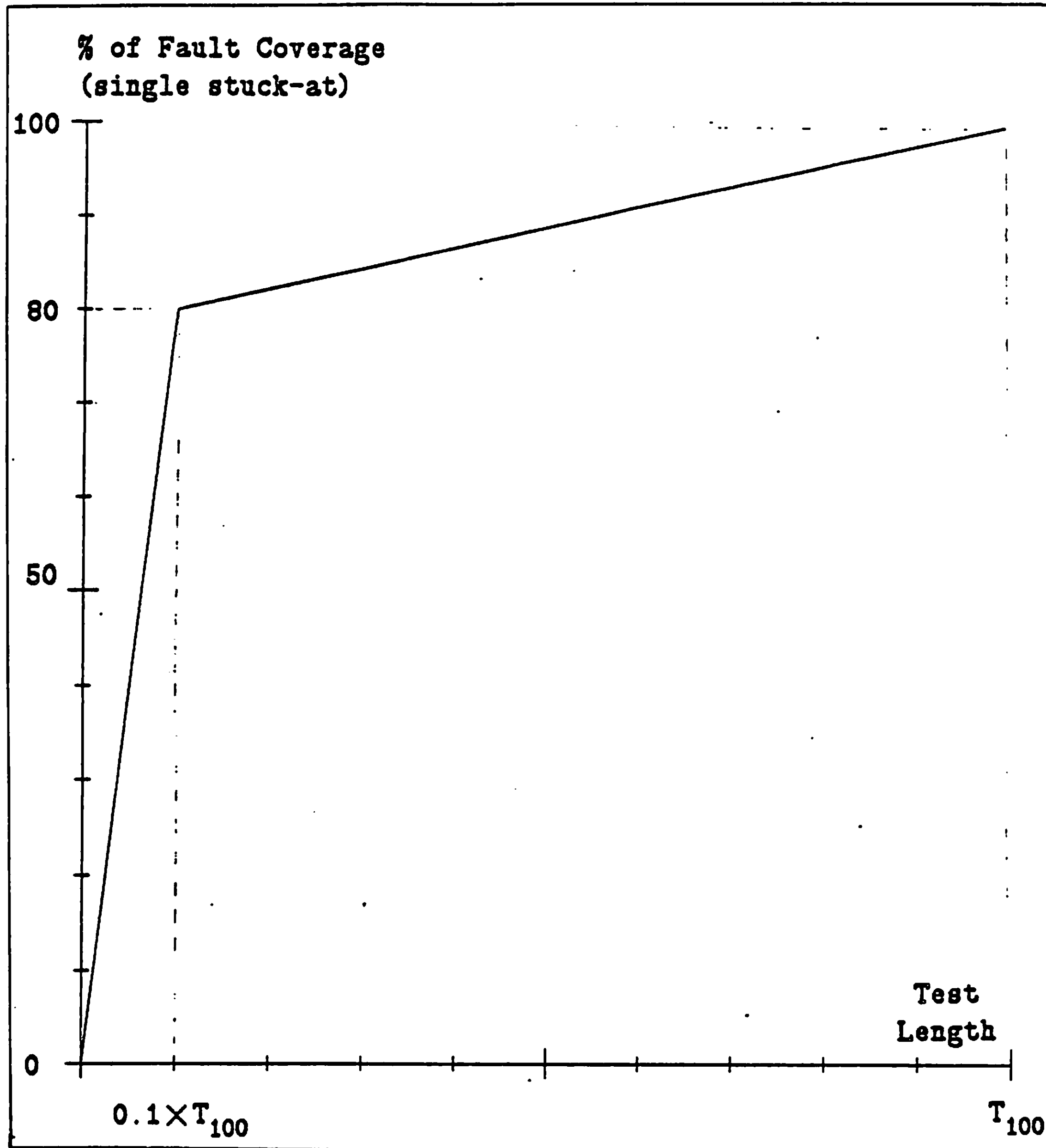
The estimator function that we have just described is for illustrative purposes only. By no means should this function be considered as the best available estimator function. Many others exist [Savir 83, Jain 85, Williams 85, Abadir 85d], and in addition new ones can be developed and used. One should note that it is always possible to compute the exact value of this measure by carrying out the necessary steps required to generate a test satisfying the fault coverage requirement. For some TDMs, such as the exhaustive TDM or the universally testable PLA TDM [Fujiwara 81], the test length function returns an exact value rather than just an estimate.

Test Time

The test time associated with a TDM is usually a function of the test length, parameters related to the kernel size such as the number of inputs and outputs, and the speed of processing the test schema. For example, as shown in Figure 3-3, the BILBO test time is the product of the test length and the speed of processing the test schema. In Chapter 6 we will discuss in detail how to calculate the latter parameter.

Fault Coverage

A list of pairs is used as the value of this slot. Each pair consists of a keyword representing a particular fault model and an approximate <u>upper bound</u> on the degree of fault coverage which could be achieved by the TDM with respect to that fault model. For example, using BILBO with a k-bits



$$T_{100} = \frac{\textit{number-of-inputs} \times \textit{number-of-gates} \times \textit{random-susceptibility-factor}}{\textit{number-of-outputs}}$$

Figure 3-4: An estimator function for determining the number of random patterns required to achieve a certain fault coverage.

signature analyzer, a probabilistic upper bound on the stuck-at degree of fault coverage which could be achieved is given by $100 \times (1 - 2^{-k})\%$.

Storage Requirement

The storage requirement measure of a TDM indicates the amount of storage required to store both the test set and the fault-free response. Clearly, such a measure is a function of the test length and other kernel parameters. For example, the storage requirement associated with the scan path TDM equals the test length multiplied by the sum of the number of kernel inputs and outputs. For the BILBO TDM, only k bits are required to store the correct signature of a kernel, where k is the size of the signature analyzer used.

3.4. Summary

In this chapter we identified the basic elements involved in the process of designing an easily testable circuit and testing it. We then presented a unified framework for modeling structural, behavioral, qualitative and quantitative aspects of TDMs. A template graph model for describing structural aspects of a TDM was presented. We introduced the concept of a test schema which is used to describe the behavioral aspects of a TDM. Measures and criteria that are commonly used to evaluate TDMs were also discussed and represented in the frame model. TDMs are the major source of knowledge for TDES. The TDM frame model supports a systematic design for testability synthesis process, and it is the basic building block of the knowledge base of TDES.

Chapter 4
Partitioning the CUC

4.1. Motivation

In general, VLSI circuits can be partitioned into functional blocks, such as control, data path, input/output, and memory. For testing purposes, a VLSI circuit can also be partitioned into "testable" subcircuits, each subcircuit being tested in its own unique ways. These subcircuits may, but not necessarily correspond to functional blocks. By definition, they are circuit structures. The first step in designing a testable VLSI circuit is to partition it into kernels. The kernels in turn define circuit structures whose fault characteristics are well defined and for which one or more TDMs are known.

The need to partition the CUC into kernels is motivated by the following:

1. The size and complexity of VLSI circuits makes it extremely hard to deal with the CUC as one kernel.

2. Different TDMs exist for structures with different types or different design styles. Hence these structures should be dealt with separately.

3. The goals and constraints of the design may impose limitations on the size of the kernels. For example, to meet a testing speed requirement of 1 second with a clock cycle of 100 nanoseconds, a simple calculation will show that to use an exhaustive TDM every combinational kernel should have 23 inputs or less.

4. Some of the structures in the CUC may have unique architectural styles, such as bit-sliced arrays, that can be exploited by the use of special kinds of TDMs. Similarly, complex structures with well

defined functions, such as shifters, counters, and FIFO queues, can often be tested using methodologies that exploit their functionality. Obviously, such situations have to be realized while partitioning the CUC.

Often, structures in the neighborhood of a kernel can be used - possibly after simple modifications - to aid in the testing of the kernel. For example, a register connected to the kernel inputs can be used to gain access to the kernel, or it can be modified to become a BILBO register which can be used to supply test patterns to the kernel. The same statement still applies if the register is connected to the kernel input via a multiplexer or a bus. Rather than dealing with the whole CUC graph when considering a particular kernel, we will extract a subgraph that consists of the kernel node itself plus all the structures in the kernel neighborhood that can potentially be used in testing the kernel. Obviously, subgraphs of different kernels need not be disjoint: in fact they often have built-in test structures in common.

4.2. Statement of the Problem

In view of the above, the partitioning problem can be formulated as follows:

1. Identify a set of structures, called the kernels, such that (1) the type, the style, the function, and the size of every kernel qualify it as a candidate for one or more known TDMs, and (2) every basic structure in the CUC is either a kernel or a subcomponent of one of the kernels.

2. For every kernel form a subgraph consisting of the kernel itself plus all its neighboring structures that can potentially be used in testing the kernel.

In the next two sections we will consider the above two subproblems individually.

4.3. Kernel Identification

There is a simple logical rule that governs this complicated process. A structure qualifies as a kernel if there is at least one known TDM which can be applied to that structure. Clearly, we have to devise a scheme for examining whether a circuit structure satisfies the above simple rule or not.

One approach is to add to the knowledge base of TDES **template** nodes or subgraphs describing all the acceptable forms of kernels. The process of checking whether a circuit structure qualifies as a kernel or not becomes equivalent to that of trying to find a match between the structure and one of the template kernels. Recall from Chapter 3, the structural template of every TDM frame contains as a part of it a node (or a subgraph) representing the kernel to which the TDM is applicable. The union of all such template nodes or graphs defines all the acceptable forms of kernels. Hence, there is no need to represent knowledge about kernel templates as a separate entity in the knowledge base.

As described in Chapter 3, the attributes (labels) of a kernel template node represent the qualification conditions that have to be met by a matching circuit structure. For example, as shown in Figure 3-1, the labels attached to the template of the BILBO kernel indicates that a structure can be tested using BILBO if its type is combinational and if its design style is either a PLA, CGN, ROM, or random. In addition, one can also attach labels that places upper bounds on the number of inputs or the number of gates which are allowed for a BILBO kernel. A structure S which satisfies all the qualification conditions associated with a template kernel is said to **match** that template kernel, and hence S **qualifies** as a kernel.

It is clear that this novel scheme for qualifying circuit structures as potential kernels is very flexible. The ability to attach different kinds of

qualification attributes to different kernel templates allows for the recognition of different classes of kernels that can be tested using extremely different testing methodologies. For example, functional attributes can be used to recognize kernels which are candidates for functional testing methodologies. On the other hand, structures with special architectural styles, such as bit sliced arrays, or structures with unique design styles such as PLAs or EX-OR trees, can also be recognized and identified as candidates for specialized testing methodologies. As described earlier, labels can be used to restrict the formation of certain types of kernels. Thus, for example, sequential kernels can be prohibited. Also upper bounds on the number of inputs of a combinational kernel can be placed.

Note that the qualification labels are a part of the TDM frames which means that they can be easily modified without changing the system code. This is a good example of how knowledge should be separated from code when designing a knowledge-based system such as TDES.

Recall from Chapter 2, a subcircuit consisting of a number of structures and their interconnections defines a structure and hence is a candidate kernel. Obviously, the number of all possible subcircuits is enormous, which rules out any exhaustive approach to the partitioning problem. A three-phase strategy for identifying potential kernels is used in TDES. First, **top-down partitioning** is carried out, followed by **flat design partitioning**, and finally **bottom-up clustering** is performed. In the next three subsections, the three partitioning phases are described in detail.

4.3.1. Top-down Partitioning

As described in Chapter 2, the CUC is represented as a hierarchical graph. In top-down partitioning, the circuit hierarchy is traversed top-down. At the top most level, every node is a candidate kernel which has to be checked against all the kernel templates. If one of the nodes does not match any of the templates, then it is partitioned into its subcomponents by going one level down the hierarchy. The new exposed nodes are candidate kernels that have to be examined and so on. Note also, that even though a node may qualify as a kernel at a certain level of the hierarchy, one can still partition to a lower level to explore other, and possibly better, matches.

4.3.2. Flat Design Partitioning

By traversing the design hierarchy top-down, a level is reached where every node is a basic structure. The design description at this level will be referred to as a **flat design**. Let $B = \{b_1, b_2, ..., b_n\}$ denote the set of basic structure nodes in the flat design of a CUC.

Every node in a flat design can be considered as a candidate kernel, and usually all of them qualify as kernels. However, if a node v in a flat design does not match any kernel template, it might be possible to combine v with other structures to form a structure which qualifies as a kernel. For example, assume node v is basic with 30 inputs and cannot be partitioned further. Let the upper bound on the number of inputs to any kernel in the library of TDMs be 25. Obviously, v does not qualify as a kernel. However, if 20 of the inputs of v come from another node u which has only 10 inputs, then by combining u and v a structure with only 20 inputs is formed which qualifies as a kernel.

The previous example raises a very important issue. It is sometimes beneficial to combine nodes together to form large kernels. Top-down

partitioning can be helpful in identifying many such complex kernels. However, top-down partitioning is very sensitive to the way the design was initially described by the user. For example, assume a node u is a subcomponent of a larger node X, and assume that there is another node v, external to X , which if combined with u results in a potentially easy-to-test kernel. However, using top-down partitioning, such a kernel cannot be identified. Hence, a third phase of partitioning is required.

4.3.3. Bottom-up Clustering

In bottom-up clustering we start from a flat design and explore ways for combining nodes together to form large structures that might qualify as kernels. Clearly, attempting to try all possible ways of clustering nodes of a flat design is computationally infeasible for large circuits (the number of possible clusters equals 2^n, where n is the number of nodes of a flat design). Besides computational difficulty, it is easy to see that one does not need to form all possible clusters, as many of them can be easily ruled out. For example, combining a RAM with a combinational structure is not useful.

There are two general kinds of clusters which are analyzed as potential kernels by TDES.

- Combinational clusters consisting of combinational basic structures.
- Sequential clusters consisting of registers plus combinational basic structures.

Next we formally define the two schemes used by TDES to identify potentially useful clusters.

4.3.3.1. Combinational Clusters

Definition 4.1: The set of **primitive combinational nodes** V is defined as the subset of B which is formed by eliminating from B all RAM nodes and register nodes.

Now that the focus of attention has been reduced to the set of primitive combinational nodes in V, an efficient scheme for identifying combinational clusters is formally presented as follows.

Combinational clustering scheme

1. Form the **primitive nodes connectivity graph PG(V, E)**, where for all u, v $\in$ V, an edge (u,v) $\in$ E exists if and only if either
   a. there is a direct connection between u and v, or
   b. the two structures have a common input source (e.g., both structures are fed from the same bus or from an output port of a third structure).
2. A set of nodes V' $\subseteq$ V can be clustered together to form a **candidate kernel** if and only if the subgraph G'(V', E') $\subseteq$ PG(V, E) is connected, where E' = { (u,v) | (u,v) $\in$ E and u, v $\in$ V' }.

It is easy to explain the logic behind the above scheme. Two primitive nodes which are not interconnected and without any common input source, are completely independent from each other. Hence there is no point in trying to consider them together as one kernel. Similarly, assume we have two clusters of nodes that have no edge in common in the PG. Clearly, these two clusters are independent from each other and one should not cluster the two groups together.

Clustering two combinational nodes together will always result in a combinational node with a "random" design style, except in one case. Clustering combinational nodes with design style "CGN" produces a combinational structure with the same design style.

There are several situations where the above clustering scheme produces beneficial kernels.

1. The number of inputs of a cluster is less than the number of inputs of one of its components. An example was given earlier.

2. By clustering nodes together, one eliminates the need to gain controllability and observability on lines which are completely contained inside a cluster, hence reducing the area overhead required to make a design testable. To clarify this point, consider two nodes u and v in V with a connection line l from an output port of u to an input port of v. Moreover assume that l does not fan out. If u and v are considered as two independent kernels, then it is necessary to (1) gain control over l to be able to drive test data into v, and (2) observe the signal values on l while testing u. As will be discussed in later chapters, to accomplish these tasks, multiplexers have to be added to the design together with the necessary interconnections. Now compare this to the solution obtained by clustering u and v together and considering them as one kernel. Clearly, line l is internal to the kernel in this case and requires no hardware modification. However, in the latter case the size and complexity of the kernel is larger and hence the number of test patterns required might be larger. Note that there is an underlying assumption that the cluster generated still qualifies as a kernel and that generating a test for it is a feasible task.

3. The clustering scheme is independent of the design hierarchical description supplied by the user in contrast to top-down partitioning. Hence, subcomponents of different complex structures can be clustered together if they are strongly connected. Moreover, by clustering the subcomponents of a complex structure in a

different way, we can effectively introduce new levels in the hierarchy which might lead to better partitioning results. For example, assume a 16-bit adder was originally described as a structure consisting of 16 1-bit adders. Using top-down partitioning, there is a choice between either considering the 16-bit adder as one kernel or considering every 1-bit adder as a kernel. However, it might be beneficial to cluster the 1-bit adders into groups of 4 or 8 and consider each group as one kernel.

4.3.3.2. Sequential Clusters

The combinational clustering scheme does not allow the formation of sequential kernels by clustering combinational nodes with register nodes. Ideally, one should avoid sequential kernels, although, in some cases it might be beneficial to deal with small sequential kernels.

Figure 4-1 illustrates three different ways for forming a sequential circuit by combining a combinational structure c with a register. Only the case shown in Figure 4-1(a) is considered in TDES as it produces kernels with less inputs and outputs than c. Hence, the area overhead associated with testing c may be reduced, with the expense of using a more complicated testing technique that employs a longer sequence of test patterns. On the other hand, sequential clusters of the type shown in Figure 4-1(b) and (c) are not beneficial because any TDM which is applicable to the sequential cluster can be applied as well to the combinational structure c, hence reducing the complexity of the testing problem. Note that the register in Figures 4-1b and 4-1c can still be used to either supply test patterns to c or receive the response of c.

A clustering scheme which allows the formation of sequential clusters is formally described next. Let R denote the set of register nodes in B.

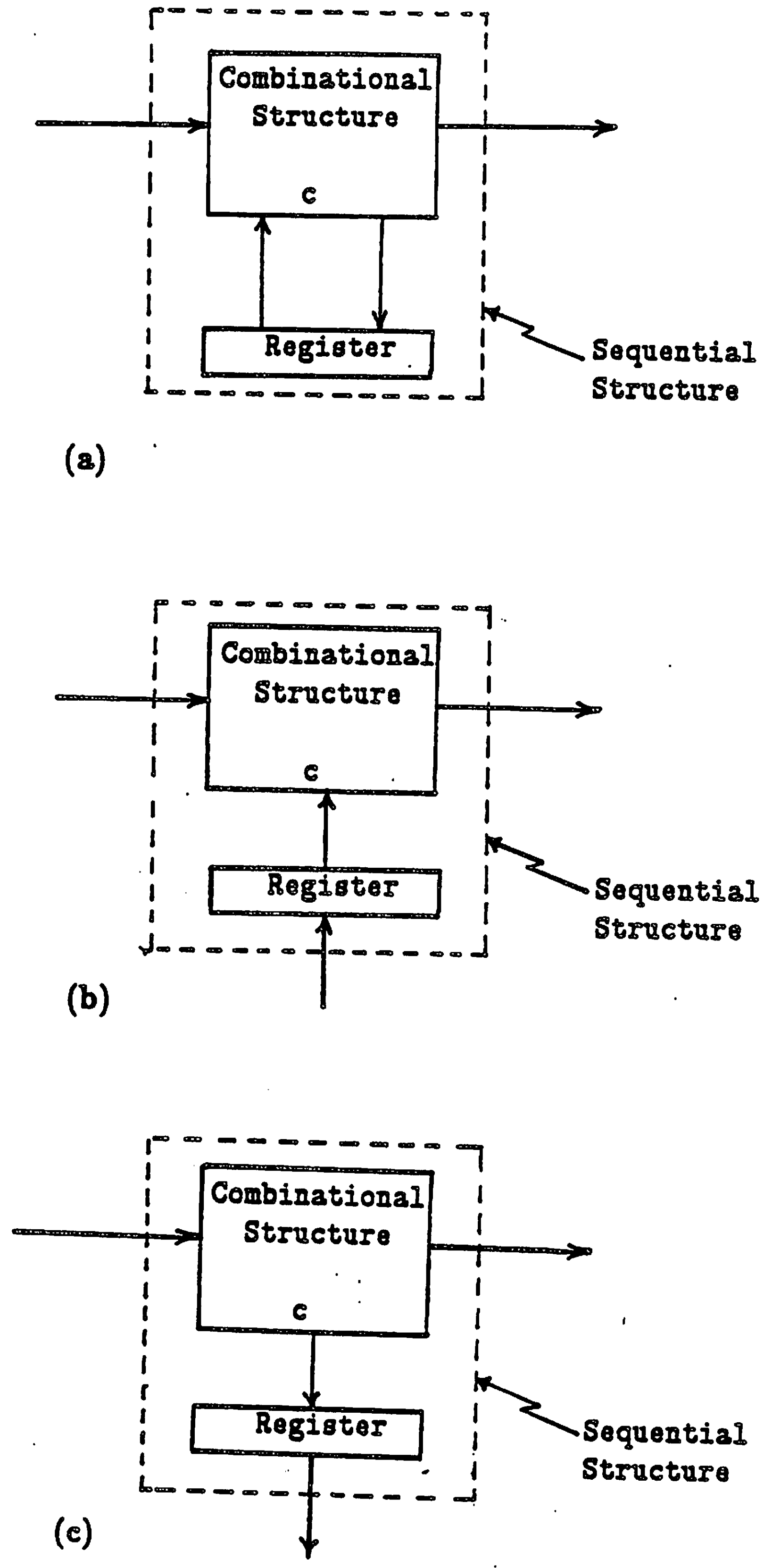


Figure 4-1: Various ways of forming sequential clusters

<u>Sequential Clustering Scheme</u>

1. Append to the PG graph the nodes in R to form the extended PG graph, EPG($V \cup R$, $E \cup F$), where $(r,v) \in F$ if and only if $r \in R$, $v \in V$, and there is a direct connection from r to v and vice versa.

2. A set of nodes $V' \subseteq V \cup R$ can be clustered together to form a **candidate kernel** if and only if the subgraph $G'(V', E') \subseteq EPG$ is connected, where $E' = \{ (u,v) \mid (u,v) \in E \cup F \text{ and } u, v \in V' \}$.

The above scheme generates sequential clusters as well as combinational clusters. A cluster is sequential if at least one of its members is an element of R, otherwise it is combinational.

4.4. Forming the Kernel Subgraph

Often, structures in the neighborhood of a kernel can be used - possibly after simple modifications - to aid in the testing of the kernel. For example, a register connected to the kernel inputs can be used to gain access to the kernel, or it can be modified to become a BILBO register which can be used to supply test patterns to the kernel. The same statement still applies if the register is connected to the kernel input through a multiplexer. This concept of passing data unchanged through circuit structures is a very important one and plays a major role in the process of designing testable circuits. In the next subsection this concept is formally defined.

4.4.1. Data Transfer Paths

Definition 4.2: A structure S with an input port X and an output port Y is said to have an **identity mode (I-mode)**, denoted by $M(S: X \rightarrow Y)$, if the data on port X can be transferred, possibly after clocking one or more times, unchanged to port Y.

A time tag t and an activation plan p are associated with every I-mode, where t is the time, in clock cycles and gate delays, for the data to be transferred from X to Y, and p indicates the sequence of data processing

actions that have to be carried out by structure S in order to activate the mode. The format for these data processing actions is the same as the one used in Section 3.3.2 for describing test schemas.

There are 4 different types of I-modes depending on the width of the ports involved and the scheme used for transferring the data. Let w denote the width of the data being transferred, measured in units of bits, and x and y denote the width of ports X and Y, respectively.

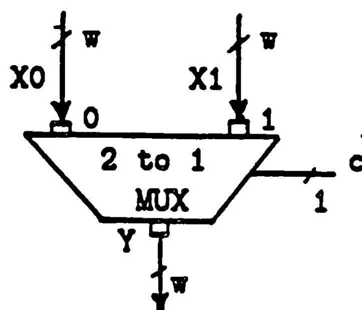
- *Parallel to Parallel I-mode* (P/P I-mode): In this mode x = y = w, and the data at X is transferred in parallel to Y. An example of a structure with parallel I-modes is a 2-to-1 multiplexer as shown in Figure 4-2(a).

- *Serial to Serial I-mode* (S/S I-mode): In this mode x = y = 1, and the w data bits are transferred serially from X to Y. The time tag t associated with an S/S I-mode indicate the time for one of the data bits to travel from X to Y. A shift register with an S/S I-mode is shown in Figure 4-2(b).

- *Parallel to Serial I-mode* (P/S I-mode): In this mode x = w and y = 1. Only structures with internal memory elements, such as registers, can have such a mode. The data is first entered in parallel via X and latched, then it is transferred serially to Y. The time tag of a P/S I-mode indicate the time for the last data bit to appear at Y. A register with a P/S I-mode is shown in Figure 4-2(c). The arrows (→) in the activation plan of the I-mode indicates the appearance of the data bits on the serial output port.

- *Serial to Parallel I-mode* (S/P I-mode): In this mode x = 1 and y = w. Again only structures with internal memory elements, such as registers, can have such a mode. The data is entered serially via X and held within the structure, then it is made available in parallel on port Y. A register with an S/P I-mode is shown in Figure 4-2(d).

The I-modes associated with a circuit structure are either explicitly specified as a part of the circuit description in the form of node labels, or in most cases, they can be inferred from the functional attributes of a structure.

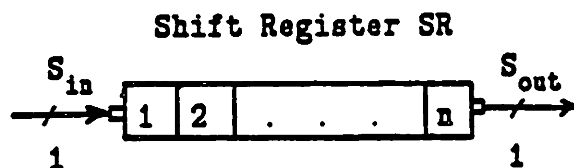


P/P I-mode M(MUX: X0 → Y) Activation plan: MUX(select 0)

P/P I-mode M(MUX: X1 → Y) Activation plan: MUX(select 1)

(a)

Shift Register SR

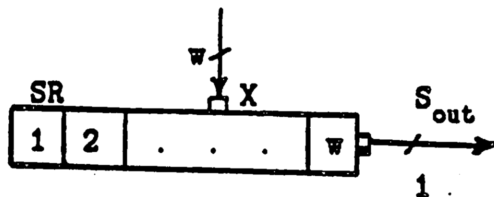


S/S I-mode M(SR: $S_{in} \rightarrow S_{out}$)
Activation plan : SR(Shift), ..., SR(Shift)

n times

(b)

Figure 4-2: Examples of structures with various I-modes

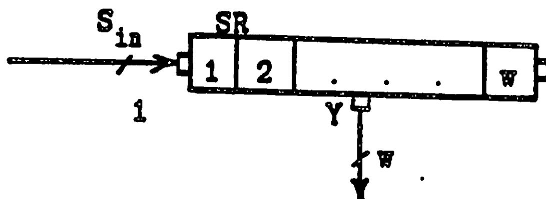


P/S I-mode M(SR: X → S_{out})

Activation plan : SR(Latch) →, SR(Shift) →, ..., SR(Shift) →

w-1 times

(c)



S/P I-mode M(SR: S_{in} → Y)

Activation plan : SR(Shift), ..., SR(Shift)

w times

(d)

Figure 4-2: (continued) Examples of structures with I-modes.

For example, by specifying that a structure functions as a multiplexer or as a shift register, the I-modes associated with that structure are implicitly specified and they can be easily inferred by TDES.

Definition 4.3: There exist an **identity transfer path (I-path)** from output port X of structure S1 to input port Y of structure S2, denoted by P(S1:X → S2:Y) if the data at port X can be transferred unchanged to port Y.

Every I-path has a time tag and an activation plan. The time tag specifies the time, in clock cycles, for the data to be transferred from X to Y, while the activation plan indicates the sequence of actions which must take place to establish the I-path. Again the format used for describing activation plans is the same as the one used for describing data processing actions of TDM test schemas.

Similar to the way I-modes were classified, there are also 4 common types of I-paths depending on the width of the ports involved and the scheme of transferring the data. Let w denote the width of the data being transferred, and let x and y denote the width of ports X and Y, respectively.

- *Parallel to Parallel I-path* (P/P I-path): $x = y = w$.
- *Serial to Serial I-path* (S/S I-path): $x = y = 1$.
- *Parallel to Serial I-path* (P/S I-path): $x = w$ and $y = 1$.
- *Serial to Parallel I-path* (S/P I-path): $x = 1$ and $y = w$.

Let W(S1:X → S2:Y) denote a physical connection between output port X of structure S1 and input port Y of structure S2. Clearly, X and Y should have the same width. An α/β I-path, where α and $\beta \in \{ P, S \}$, is formally defined as follows.

Definition 4.4: There exist an α/β I-path P(S1:X $\rightarrow$ S2:Y), where α and $\beta \in \{$ P, S $\}$ if either

1. $\alpha = \beta$ and W(S1:X $\rightarrow$ S2:Y), or

2. W(S1:X $\rightarrow$ S3:Z), α/γ I-mode M(S3: Z $\rightarrow$ Q) where $\gamma \in \{$ P, S $\}$, and γ/β I-path P(S3:Q $\rightarrow$ S2:Y).

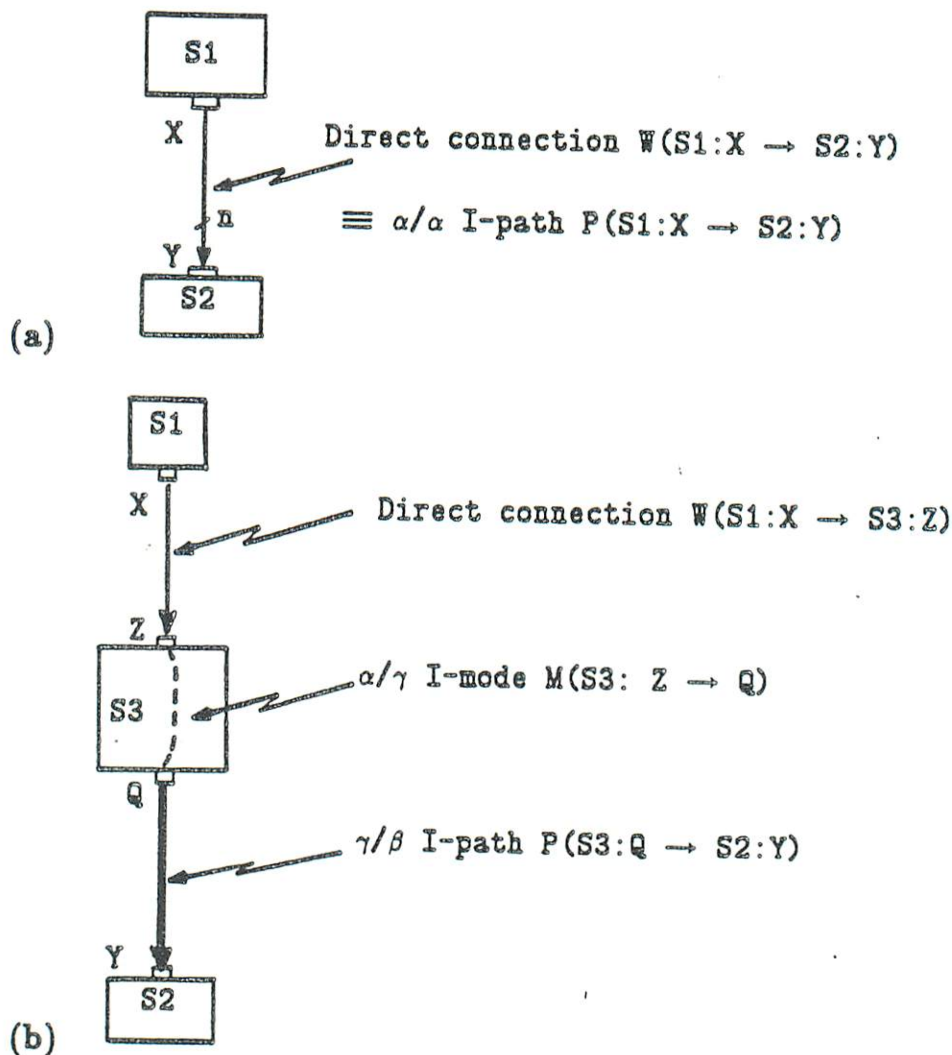


Figure 4-3: α/β I-path P(S1:X $\rightarrow$ S2:Y)
(a) the trivial case (b) the recursive case.

Figure 4-3 illustrates the above definition. The time delay associated with an I-path equals the sum of the time tags associated with all the I-modes encountered along the path. The activation plan of an I-path can be formed

by concatenating the activation plans of all the I-modes encountered along the path.[2]

As an example of I-paths, consider the circuit of Figure 4-4(a). There is a P/P I-path between port X of C and port Y of R2. There is also an S/S I-path P(R2:Z → R3:Q). Linking these two I-paths with the P/S I-mode of R2 produces a P/S I-path P(C:X → R3:Q). The time delay of the latter I-path is six clock cycles and its activation plan is shown in Figure 4-4(b). The symbol →R3:Q indicates the arrival of the data bits at the input serial port of R3. R3 might be a signature analyzer or a transition counter that is processing the test responses of C.

4.4.2. Constructing SG_k

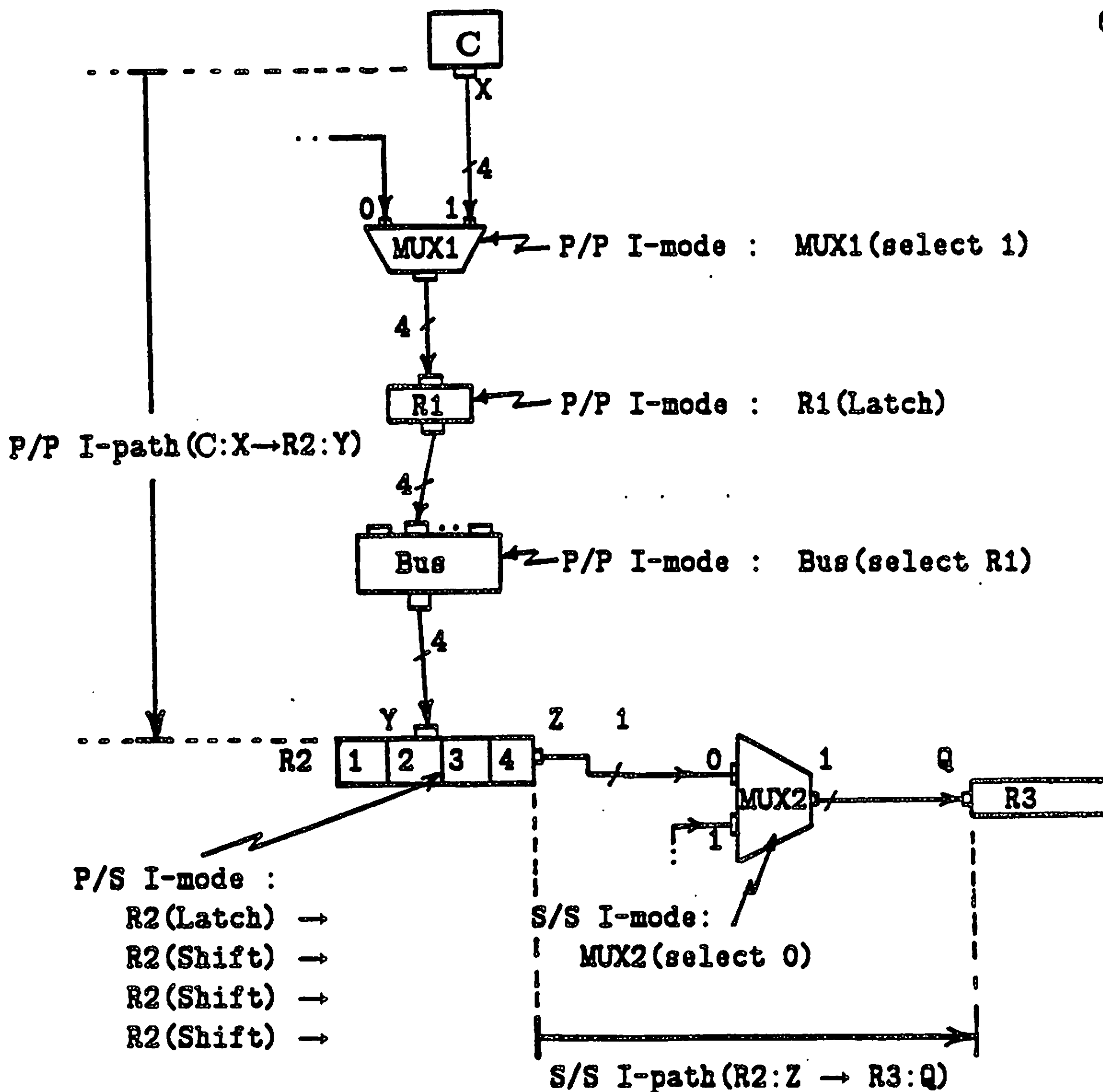
Given a kernel k, it is of interest to identify all the circuit structures that can play a role in testing k. These structures together with the kernel form what we call the subgraph of k. Thus, while analyzing the problem of testing k, instead of considering every structure in the CUC, only those structures in the subgraph of k are considered.

Recall that B is the set of basic structure nodes that constitutes a flat design of the CUC. Let A be the set of interconnections between nodes in B. Using the concept of I-paths, a kernel subgraph is formally defined as follows.

Definition 4.5: Given a kernel k, its **subgraph** is denoted by $SG_k(N_k, E_k)$, where $N_k = \{ x \mid x \in B$ and either (1) $x=k$, (2) there is an I-path from an output port of x to an input port of k, or (3) there is an I-path from an output port of k to an input port of x$\}$, and $E_k = \{(x, y) \mid (x, y) \in A$ and $x, y \in N_k\}$.

---

[2] We assume that an I-path exists only if its activation plan is feasible, that is the plan does not call for actions in the same clock cycle that require conflicting control signals.



(a)

<u>Activation Plan</u> of P(C:X → R3:Q):

MUX1(select 1), R1(Latch).
Bus(select R1), R2(Latch).
R2(Shift), MUX2(select 0), →R3:Q.
R2(Shift), MUX2(select 0), →R3:Q.
R2(Shift), MUX2(select 0), →R3:Q.
MUX2(select 0), →R3:Q.

(b)

Figure 4-4: (a) A P/S I-path P(C:X → R3:Q) (b) Its activation plan.

Nodes in N_k satisfying condition 2 (3) are called driving (receiving) nodes.

As an example, consider the circuit shown in Figure 4-5. The structures labeled c1, c2, ..., c6 are combinational structures while the ones labeled R1, R2, ..., R6 are registers. The structure labeled MUX is a 3 to 1 multiplexer. The subgraphs of c1, c3 and c6 are surrounded by dotted lines in Figure 4-5. For example, the subgraph of c3, SG_{c3}, consists of c1, c2, MUX, R3, c3, R4 and c4. The driving nodes of c3 are c1, c2, MUX, R3 and R4, while the receiving nodes are R4, c4, MUX and R3. Note that R3, R4 and MUX are both driving and receiving nodes in SG_{c3}.

4.5. Summary

In this chapter the problem of partitioning a circuit into a number of components, called kernels, which can be tested independently was examined. A three phase partitioning scheme incorporating (1) top-down partitioning, (2) flat-design partitioning, and (3) bottom-up clustering was presented. The new concepts of I-modes and I-paths were introduced. Using these concepts, a process for forming a subgraph for any circuit kernel was described. This subgraph will play a key role when TDMs are embedded in the CUC.

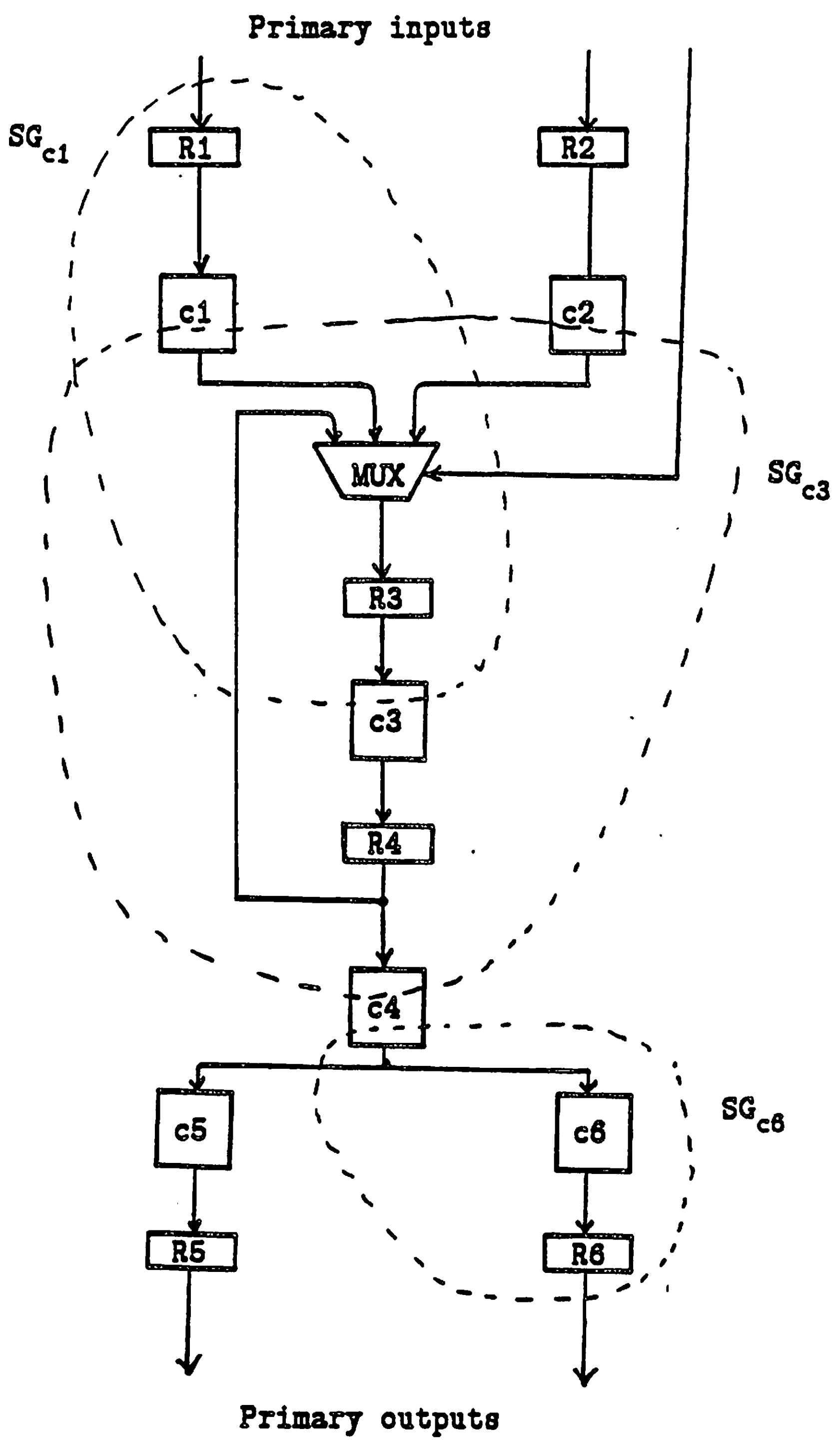


Figure 4-5: Illustrating the subgraph concept.

Chapter 5
TDM Selection and Embedding

5.1. Introduction

In the previous chapter we described how to partition the circuit into a number of kernels. Hence the problem of designing an easily testable chip consists of a number subproblems each dealing with one of the circuit kernels.

In this chapter we reduce our focus of attention to a single kernel. We examine the problem of how a kernel can be made testable by applying the knowledge stored in one of the TDM frames. Recall that both TDM templates and a kernel with its surrounding environment are represented by graphs. Hence the problem becomes equivalent to that of **embedding** a TDM template graph into the kernel subgraph.

In the next sections we first examine the unique characteristics of the TDM embedding process, explore the various cases that might be encountered and the various solutions that might exist. Later we present various measures that are used to evaluate and compare different embedding solutions. These are used to identify the best TDM embedding for a given kernel.

5.2. Embedding TDM Frames into Kernel Subgraphs

The process of embedding a TDM frame into a kernel subgraph is nothing but a **unification** process [Nilsson 80] in which we try to associate the various elements of a TDM frame with actual actors from the CUC. Hence, an embedding solution can be regarded as an instantiation of a TDM frame. The

structural slots of the instantiated frame points to real circuit objects, the behavioral slots specify real circuit actions, and the measure slots contains the actual values of the corresponding measures.

To help understand the unique characteristics of the TDM embedding process, we first examine the case of a kernel with a single input port and a single output port. Later we remove that restriction and examine the general case.

5.2.1. TDM Embedding for a Single Input Port/Single Output Port Kernel

Let k denote a single input port, single output port kernel under consideration. The subgraph of k is denoted by $SG_k(N_k, E_k)$. Assume that we are interested in embedding a particular TDM, say TDM_i, into SG_k. Let the template graph of TDM_i be denoted by $TG_i(TN, TA)$, where TN is the set of nodes in the template graph and TA is the set of arcs. Recall from Chapter 3, various qualification labels are attached to template nodes in TN.

Definition 5.1: A circuit node n **matches** a template node t if and only if for every qualification label attached to t of the form
($attribute_i$ $qualification\text{-}function_i$ $required\text{-}value_i$) either

1. n has a label of the form ($attribute_i$ value) such that the expression "value $qualification\text{-}function_i$ $required\text{-}value_i$" is logically true, or

2. n can be modified such that one of its labels becomes of the form ($attribute_i$ value) and such that the expression "value $qualification\text{-}function_i$ $required\text{-}value_i$" is logically true.

Using the above definition, we can informally define a **feasible embedding** solution as a solution in which (1) there is a matching circuit node for every template node and (2) for every arc in TA from node t_1 to node t_2, there exist an I-path from the circuit node matching t_1 to the circuit node matching t_2.

Finding a feasible TDM embedding often requires modifying existing circuit nodes or adding new ones in order to satisfy the matching requirements. For example, a simple register in the CUC has to be modified to become an LFSR in order to match the driver node of the BILBO TDM. The modified register should have both the RNG and clear modes of operation. In addition there should be a parallel I-path from that register to the inputs of the kernel under consideration.

Note that the modifications and additions that are made to the circuit to obtain a feasible embedding represent a way of transforming an existing design into an easily testable one. These transformations should not change the designated behavior of the circuit, except possibly for a small degradation in performance.

In general, there might be several embeddings for a particular TDM into a circuit, each employing a different matching combination. In embeddings of certain TDMs, a circuit node can be used to match more than one template node. For example, in a scan path embedding, one shift register can be used as a driver of the kernel inputs and also as a receiver from the kernel outputs. Hence, both the driver and the receiver shift register nodes in the scan-path template are matched by the same shift register in the circuit. However, for a BILBO embedding, the register used to generate the random patterns must be different from that used for computing a signature. To illustrate the above point formally we first present two definitions.

Definition 5.2: A TDM is said to have a **retentive driver** if the on-chip actions required to generate and apply a test vector to a kernel is a function of the previous test vectors. TDMs which do not satisfy this property are said to have a **memoryless driver**.

For example, BILBO has a retentive driver as it employs a linear

feedback shift register (LFSR) to generate the test patterns. On the other hand, the traditional scan-path TDM [Funatsu 75] has a memoryless driver because the test vector used in one iteration is independent of the test vectors used in the previous iterations. Another example of a TDM with a retentive driver is a new TDM called SPLASH [Abadir 85d], which incorporates the traditional scan-path structural template with a new test schema that requires a retentive driver.

Definition 5.3: A TDM is said to have a **retentive receiver** if the on-chip actions required to capture and process a kernel output response is a function of the previous responses. TDMs which do not satisfy this property are said to have a **memoryless receiver**.

BILBO has a retentive receiver because it employs a response compaction technique, namely signature analysis. On the other hand, both the traditional scan-path TDM [Funatsu 75] and SPLASH [Abadir 85d] have memoryless receivers. The retentive status of a TDM driver (receiver) is explicitly specified in the TDM frame in the form of a label attached to the driver (receiver) node of the TDM structural template.

The following theorem provides an important condition which must be satisfied in any feasible embedding.

Theorem 5.4: In any feasible embedding of a TDM with a retentive driver (receiver), the circuit register used to match the template test generator (response evaluator) node cannot be used either (1) to match a different TDM node, or (2) as a part of an I-path which is used to match one of the template arcs in the embedding.

Proof: By contradiction. Consider an embedding of a TDM with a retentive driver (receiver). Assume register R is used to match the test generator (response evaluator) template node. Also assume that R is used

either to match another template node or as a part of an I-path in the embedding. By definition, the TDM requires that the contents of the test generator (response evaluator) should be retained unchanged between consecutive iterations. However, since R has dual roles in the embedding its contents will change, at least once, in the period between consecutive iterations. Hence a contradiction.

□

In view of the above theorem, the following two observations should be emphasized.

1. In an embedding of a TDM with a memoryless driver and receiver, a circuit node can be used to match more than one template node.

2. In any TDM embedding, it is feasible to use a circuit node more than once as a part of several I-paths. This is true because a node used as a part of an I-path need not retain its state after establishing its role in the I-path.

5.2.1.1. Formal Definition of A Feasible Embedding

A feasible embedding of the template graph TG(TN, TA) of TDM_i into $SG_k(N_k, E_k)$ can formally be defined as a mapping $\mathcal{M}$ from the set of template nodes TN into the set of circuit nodes N_k, denoted by $\mathcal{M}:TN \Rightarrow N_k$, such that

1. $\mathcal{M}:t \Rightarrow n$ where n matches t.

2. For every arc from node u to node v in TA, there exist an I-path(x → y), where $\mathcal{M}:u \Rightarrow x$ and $\mathcal{M}:v \Rightarrow y$.

3. If TDM_i has a retentive driver (receiver) and $\mathcal{M}:r \Rightarrow n$, where r is the driver (receiver) node, then

   a. For all template node $t \neq \dot{r}$, $\mathcal{M}:t \Rightarrow a \neq n$, and

   b. n is not a part of any I-path used to match one of arcs in TA.

In the rest of our discussion we will refer to a feasible embedding as just an embedding. To clarify the characteristics of the embedding process consider the following example.

Example 5.1: Assume we want to embed the BILBO TDM, whose template graph is shown in Figure 3-1, into the subgraph SG_k of kernel k, shown in Figure 5-1. The labels associated with the nodes in both figures indicate their key attributes. The first step is to make sure that the BILBO methodology is applicable to k. The qualification labels of the template kernel indicate that a BILBO kernel should be combinational with a design style of either PLA, ROM, CGN, or random logic. Clearly, k matches the BILBO kernel.

The next step is to find two matching nodes for B1 and B2 among the circuit nodes. There are 3 candidate registers in SG_k, namely R1, R3, and R4. All three registers meet the size requirement and have I-paths to the input port of k. R1 has all the functional modes of B1. However, in order for either R3 or R4 to match B1 they have to be modified by adding a clear mode and a pseudo random generation capability to their logic. Three registers R2, R3 and R4 are candidates for matching B2. Both R3 and R4 lack the SA and clear modes, while R2 is only missing the SA mode. Out of the 9 possible mapping combinations, and assuming that the required functional features are added, only 4 lead to feasible embeddings as illustrated in Table 5-1. The 5 other mappings are rejected because they do not satisfy the condition of Theorem 5.4. Recall that the BILBO TDM has a retentive driver and receiver.

Let us consider the embedding obtained by mapping B1 and B2 to R1 and R2, respectively. The matching results are shown in Figure 5-2a. Note how the arcs of the template graph are mapped into I-paths in SG_k. The activation plans for the I-paths are also shown in Figure 5-2a.

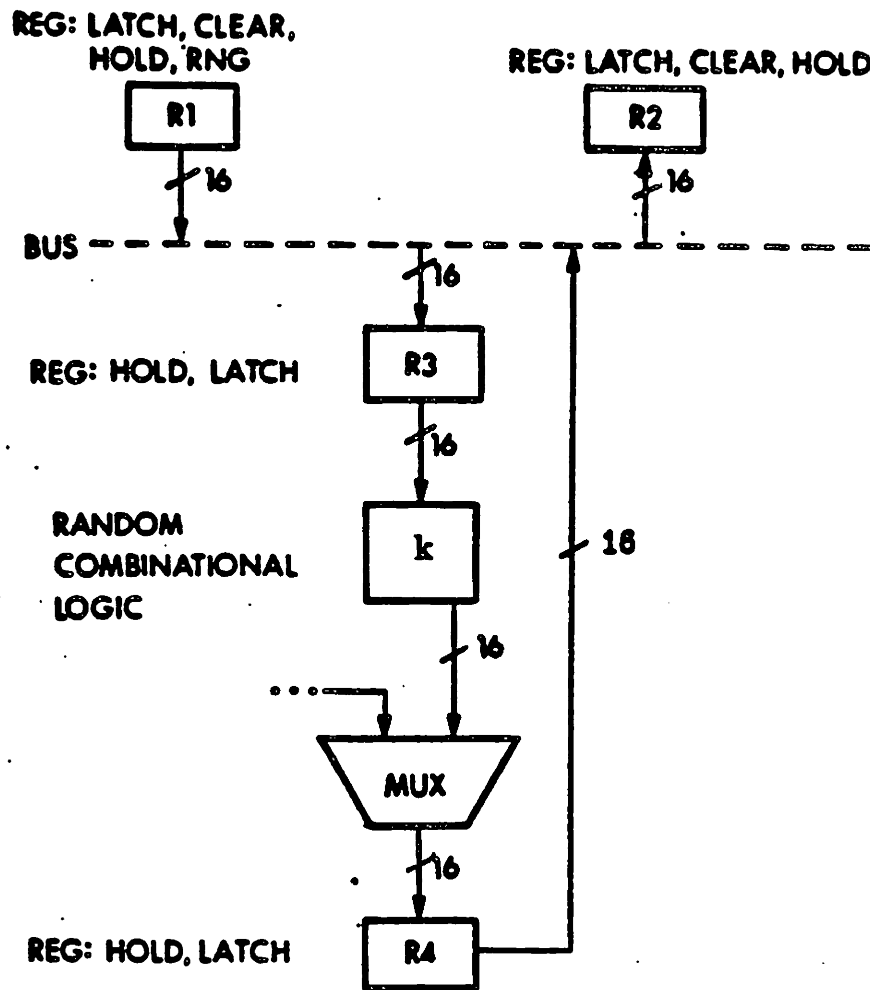


Figure 5-1: A circuit example for illustrating the embedding process.

Node matching B1	Node matching B2	Status of mapping	Reason of rejection
R1	R2	feasible	
R1	R3	not feasible	R3 is a part of P(R1 → k)
R1	R4	feasible	
R3	R2	feasible	
R3	R3	not feasible	mapping is not one-to-one
R3	R4	feasible	
R4	R2	not feasible	R4 is a part of P(k → R2)
R4	R3	not feasible	R3 is a part of P(R4 → k)
R4	R4	not feasible	mapping is not one-to-one

Table 5-1: Different mapping combinations

```
Kernel              ⇒ k
n                   ⇒ 16
m                   ⇒ 16
B1                  ⇒ R1
B2                  ⇒ R2 Modified (+ SA)
P(B1 → Kernel)      ⇒ P(R1→k) ≡ Bus(R1), R3(Latch).
P(Kernel → B2)      ⇒ P(k→R2) ≡ MUX(k) , R4(Latch).  Bus(R4),
```

(a)

```
Body
   Execute T times:
        1    R1(RNG).
        2    Bus(R1), R3(Latch).
        3    k, MUX(k), R4(Latch).
        4    Bus(R4), R2(SA).
```

(b)

Figure 5-2: A feasible BILBO embedding (a) matching data (b) the test plan.

The area overhead associated with this particular embedding corresponds to the cost of modifying R2 to accommodate the SA capability. Such a

modification may also have some adverse effects on the speed of the circuit if, during normal operation, R2 is used as a sink in a critical path.

The embedding solution considered above has a low area overhead. Contrast this solution with the embedding obtained by mapping B1 and B2 to R3 and R4, respectively. The area overhead of the second embedding is high because both R3 and R4 lack key capabilities. However, the test time or the performance degradation associated with the second embedding might be better than the ones associated with the first embedding. Clearly trade-offs exist. The various measures used for evaluating different embeddings will be discussed in detail in Section 5.3.

□

5.2.1.2. Generating a Test Plan for an Embedding

The test schema of the TDM can be customized to a given embedding producing a **test plan** for the kernel under consideration. The process of generating a test plan from the TDM test schema can be simply described as follows:

1. Replace the names of the TDM template structures in the schema with the corresponding circuit structures as defined in the embedding.

2. Replace every data transfer action in the schema with the activation plan of the corresponding I-path.

Clearly, a test plan has the same general form as a test schema, i.e., it has a head, a tail, and a body. However, the actions in a test plan are all data processing actions since the data transfer actions are replaced by activation plans which in turn consists of data processing actions (structures operating in their I-modes). The actions that form the body of a test plan can be organized, according to their order of execution, into a number of steps, such that the actions in one step can all be executed in one clock period. The steps are

numbered 1, 2, ..., S where S is the total number of steps. Normally a step ends when data is clocked into a register. We assume that we are dealing with synchronous circuits, and that registers are triggered by the trailing edge of every clock pulse. We also assume that all registers are in a hold state unless otherwise specified in the test plan. Example 5.2 illustrates the process of generating a test plan for a circuit kernel.

Example 5.2: Consider the circuit of Figure 5-1a and assume that the embedding of Figure 5-2a is selected for kernel k. By carrying out the necessary substitution in the BILBO test schema, a test plan with a 4-step body is generated as shown in Figure 5-2b. The period (.) between actions signals the end of one clock period.

□

Note that the body of a test plan has to be executed T times, where T is the number of test vectors to be used. Clearly, if the iterations are executed in a sequential manner, $S \times T$ clock cycles are needed to execute the test. However, it is often possible to introduce some amount of parallelism while executing the consecutive iterations of the test plan [Abadir 85b, Abadir 85c]. Hence, reducing the test time. Methods for optimizing the execution time of test plans are discussed in detail in the next chapter.

5.2.1.3. Creating New Structures in a Circuit

One situation that may be encountered in the embedding process is when none of the nodes in the subgraph of k can be modified to match one of the template nodes of a TDM. For example, consider the kernel subgraph shown in Figure 5-3a. Assume that we want to embed this kernel into a scan-path TDM. The scan-path TDM uses a shift register to scan-in test vectors and apply them to the kernel. Clearly, there is no registers in SG_k with an I-path to k that can be reconfigured to do this function. Therefore, the only option is to

add new nodes to the circuit. There are two possible solutions, as shown in Figures 5-3b and c. The first inserts a register, which can operate in either a shift mode or a parallel latch mode, in the input path to k. Obviously, this solution has very serious effects on the CUC normal performance (an extra clock delay), hence it will not be considered further.

The second solution inserts a 2 to 1 MUX in the input path to k, with a shift register driving one of the MUX inputs, and the normal input source of k driving the other MUX input. During normal operation the MUX is controlled such that k is derived from its normal input source. During the test mode, the shift register can be used to scan-in test vectors and apply them to k via the MUX. The shift register can either be added to the circuit, or in most cases, one of the already existing circuit registers is used.

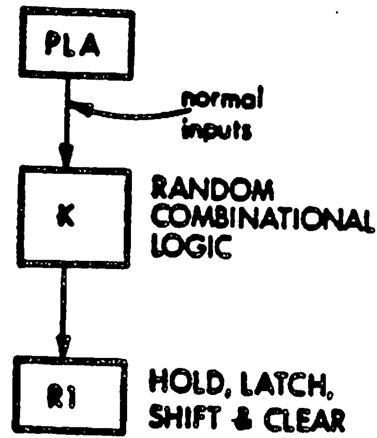
It is clear that by adding a MUX an I-path P(S1:X → S2:Y) can be created between two structures S1 and S2. Recall that S1:X denotes port X of structure S1. In general there are several ways to create an I-path P(S1:X → S2:Y) as illustrated in Figure 5-4.

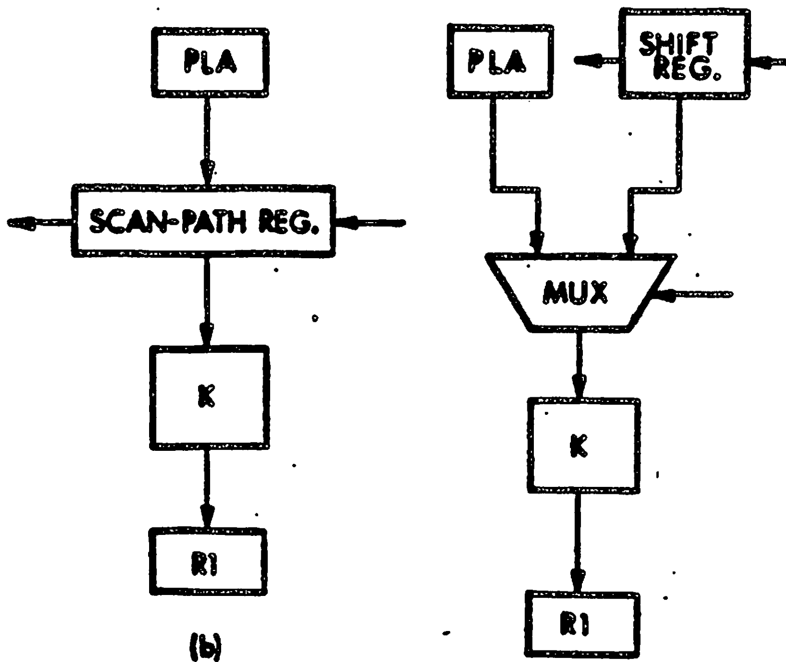
1. The trivial solution is shown in Figure 5-4a. The data of port X is multiplexed with the normal data source of Y.

2. Assume that there is a structure S3 in the circuit with an I-mode M(S3:Z→W), such that P(S1:X → S3:Z) exists. Thus, by multiplexing the data of port W with the normal data source of Y, an I-path P(S1:X → S2:Y) is created as shown in Figure 5-4b.

3. Assume that there is a structure S4 in the circuit with an I-mode M(S4:P→Q), such that P(S4:Q → S2:Y) exists. Thus, by multiplexing the data of port X with the normal data source of port P of S4, an I-path P(S1:X → S2:Y) is created as shown in Figure 5-4c.

Note that in cases 2 and 3 above, there might be several structures in the circuit that can play the intermediate role of S3 or that of S4. In all cases,





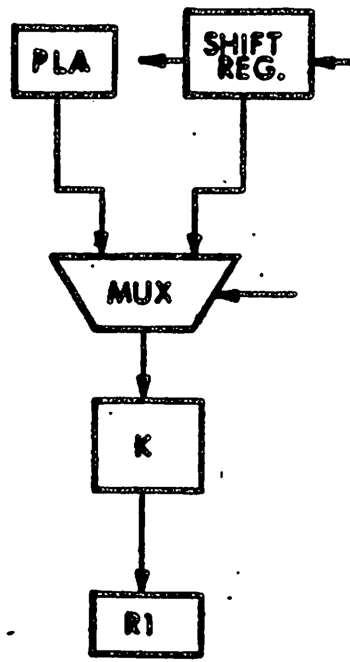
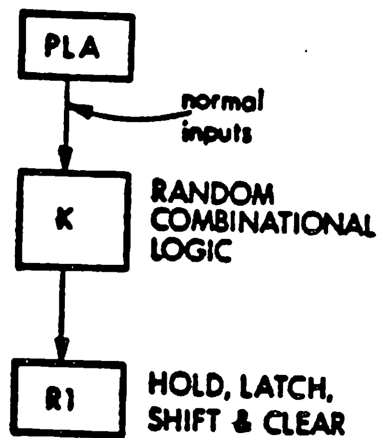
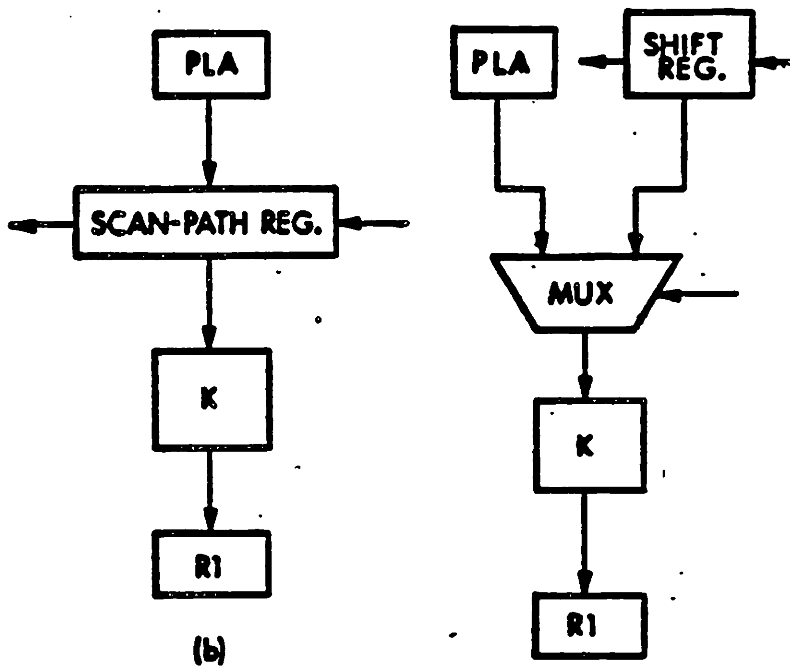


Figure 5-3: Adding structures to a circuit (a) kernel subgraph (b) inserting a register (c) inserting a multiplexer.





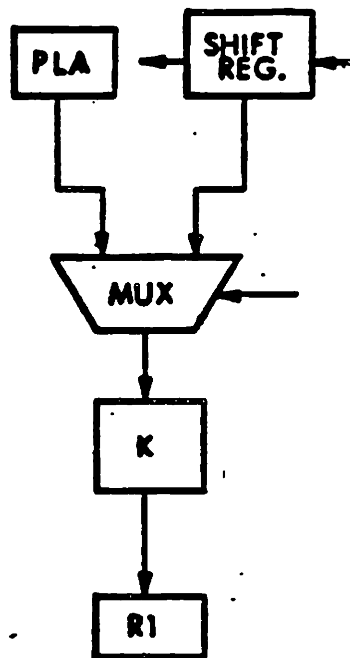


Figure 5-3: Adding structures to a circuit (a) kernel subgraph (b) inserting a register (c) inserting a multiplexer.

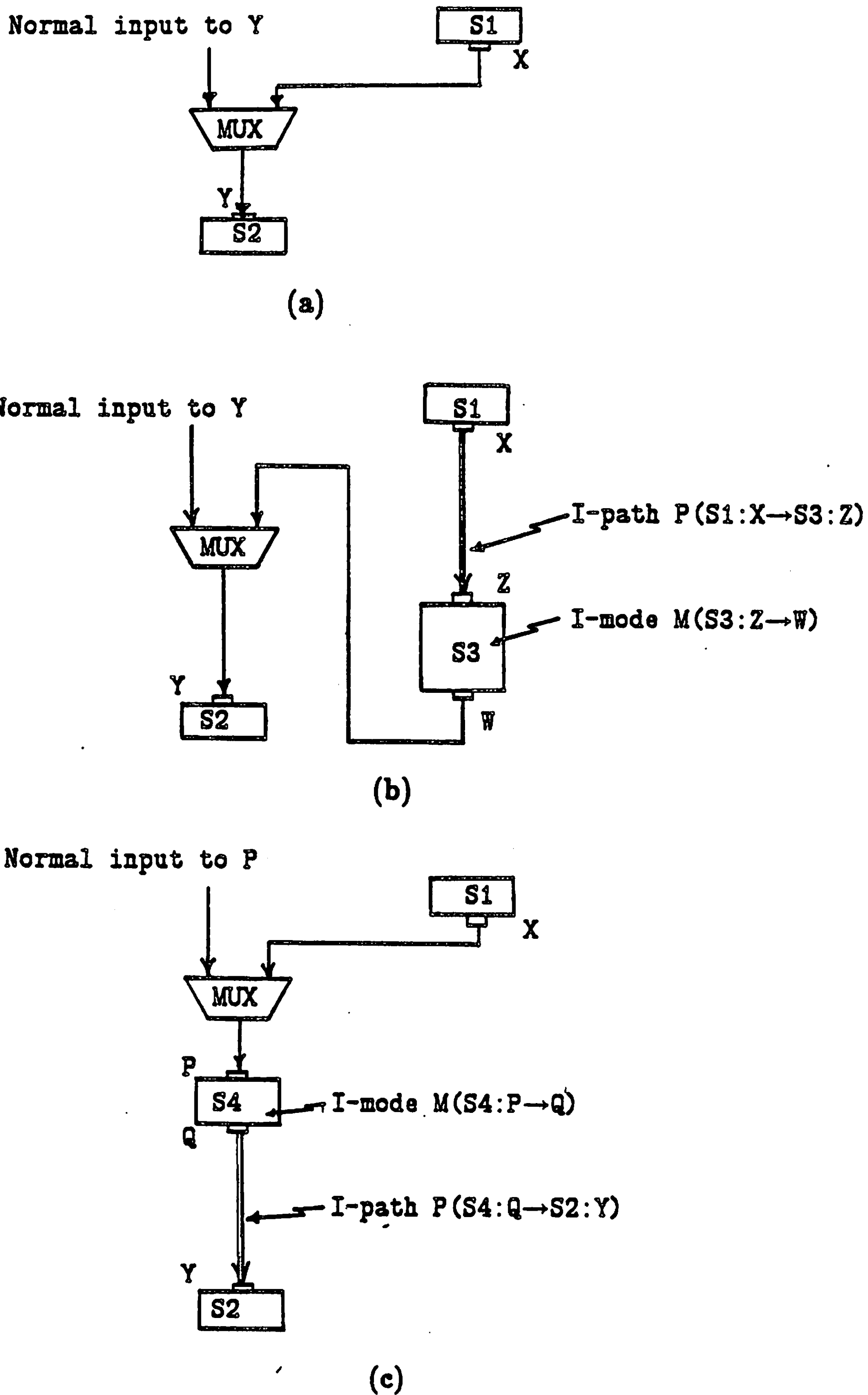


Figure 5-4: Different schemes for creating an I-path between two structures.

there is an area overhead penalty corresponding to the added MUX and the area used to route the new interconnections. The routing cost might be more expensive in some cases than others depending on the final layout of the circuit [Syed 81, Kurdahi 85]. Equally important, the effect of the modification on the circuit's normal operation depends on where the MUX is inserted. For example, Figure 5-4a an extra delay is added to all data paths feeding port Y during normal operation. On the other hand, in Figure 5-4c, a delay is added to all data paths feeding port P of structure S4. Depending on the delay characteristics of the circuit structures and the location of critical paths in the design, one can measure the impact of inserting a MUX on circuit performance [Park 85].

In the above discussion we focused on the process of creating an α/α I-path in a circuit, where $\alpha \in \{P, S\}$. Recall that an β/γ I-path, where $\beta \neq \gamma$, consists of a β/β I-path, followed by a β/γ I-mode, and finally a γ/γ I-path. Thus, if a TDM requires a β/γ I-path P(S1:X $\rightarrow$ S2:Y) which does not exist, then the task of establishing such an I-path can be divided into three subtasks.

1. Find a structure S3 in the circuit which either has a β/γ I-mode M(S3:Z$\rightarrow$W), or can be modified to have one.

2. Establish a β/β I-path P(S1:X $\rightarrow$ S3:Z).

3. Establish a γ/γ I-path P(S3:W $\rightarrow$ S2:Y).

Definition 5.5: A **MUX-embedding** is an embedding which creates an I-path in a circuit using a multiplexer.

Clearly, the number of MUX-embeddings in a circuit can be enormous. For example, consider the process of applying BILBO to a kernel k. Every register in the circuit can be used as a driver for k either via a natural I-path or via a created one. Note that there might be more than one way of creating such an I-path as described earlier. A similar argument applies to the receiver

register. Theoretically, one can explore MUX-embeddings even though feasible embeddings using original I-paths exist. However, this substantially increases the complexity of the problem. Note also that the area and performance penalties associated with MUX-embeddings are often much higher than other embeddings. Next, we identify two general situations where MUX-embeddings are essential and have to be considered. We will also identify another situation where MUX-embeddings might be superior to other embeddings.

No-match case

Consider a case where a particular template node cannot be mapped into a circuit node satisfying the I-path condition. An example of this situation was illustrated earlier in Figure 5-3. Clearly in this case the only way to obtain a feasible embedding is by creating I-paths via MUX-embeddings. However, to reduce the number of generated embeddings, one can restrict the number of choices associated with the mechanism for creating an I-path between two nodes. For example, one can prohibit the use of the scheme of Figure 5-4b, or that of Figure 5-4c, or both.

Register feedback case

Consider the circuit of Figure 5-5. Kernel k and register R form a feedback loop. Assume we want to embed the BILBO TDM into this circuit. The BILBO driver and receiver can both be mapped into R. However, since a BILBO embedding has to be one to one, a register other than R has to be used either as the receiver or as the driver. Thus, even though the circuit of Figure 5-5 does not qualify as a no-match case, MUX-embeddings are essential in this case as well. An I-path can be created either to drive k from a register other than R, or to direct the response of k into a register other than R.

It is interesting to note that for the scan-path TDM or any TDM with a memoryless driver and receiver, MUX-embeddings are not essential in the register feedback case. Clearly, register R can be used as the input shift

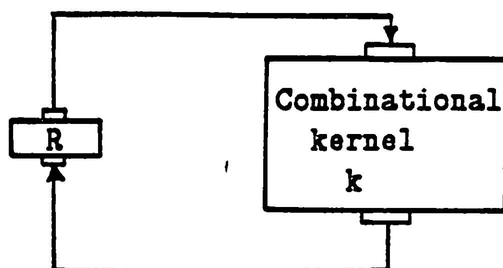


Figure 5-5: A register feedback circuit case.

register and also as the output shift register in a feasible scan-path embedding for kernel k.

Perfect or near perfect match case

Consider a situation where all feasible embeddings, which are not MUX-embeddings, have high area overhead penalty due to modifications required to match a template node t. Now assume that there is a structure S in the circuit that perfectly or near perfectly matches t, but cannot be used because of the lack of an I-path. Clearly, in such a case, it might be beneficial to consider MUX-embeddings that include S.

A useful guideline that can be followed in cases like this is to consider a MUX-embedding only if its area overhead penalty is less than or comparable to the area overhead penalty associated with all non MUX-embeddings.

5.2.2. TDM Embedding for Kernels with Multiple Input and Output Ports

In the previous sections we described how to generate TDM embeddings for a kernel with a single input and a single output port. In this section we consider the general case of multiple I/O ports. Consider a kernel with n input ports and m output ports. In such a case, a node in a TDM template graph with an arc A going into (coming from) the template kernel has to be mapped into n (m) disjoint nodes in the kernel subgraph, one for each port. Moreover, arc A has to be mapped into n (m) I-paths from the n matching nodes (m

kernel output ports) to the n kernel input ports (m matching nodes). Thus, in essence, the matching process is carried out once for every individual port.

Theorem 5.4 and its corollaries apply to the multiple ports kernel case as well. Hence, if a TDM has a memoryless driver and receiver, then in an embedding of that TDM a circuit node can be used to match a driver node and a receiver node at the same time. However, the n (m) nodes used to match one of the template nodes have to be disjoint.

One or more multiplexers can be used to create I-paths either to or from kernel ports, hence producing MUX-embeddings. Situations where MUX-embeddings are essential or beneficial are the same as those discussed earlier.

The only task associated with the multiple port kernel case which requires special treatment is the generation of a test plan. This task is discussed next.

5.2.2.1. Generating a Test Plan for a Multiple Port Kernel

In a TDM embedding for a multiple port kernel, there are n(m) I-paths that drive data into(from) the kernel. Each one of these I-paths is completely specified by an activation plan. Some of these I-paths may share resources. Our goal is to generate a test plan for the circuit that will exercise those I-paths in a correct fashion without creating conflicts. This task can be done in three steps as follows:

1. Combine the activation plans of the input I-paths.

2. Repeat for the output I-paths.

3. Concatenate the two plans generated above.

Next, we will describe a procedure that can be used to combine the activation plans of n I-paths into a single plan.

Problem Statement: Given the activation plans of n I-paths P_1, P_2, ..., P_n, generate a feasible plan to activate all the I-paths without causing any resource sharing conflicts. Let the number of steps in plan P_i be denoted by S_i. We will refer to step s of P_i by i§s. To activate the I-paths simultaneously and avoid conflicts, it might be necessary to delay the flow of data along one or more I-paths for one or more clock cycles at different points of time.

Definition 5.6: A **No-Op** step following step i§s, denoted by i§s', is a step that calls for holding the state of the register used to store the data in step i§s. In addition, the No-Op step that precedes the first step of P_i, denoted by i§0', calls for holding the state of the register(s) which contain the data to be transferred via P_i.

Definition 5.7: A **combined plan conflict graph** CPCG (N, E) reflects resource sharing conflicts between steps of different I-paths, where the nodes in N correspond to the steps of the plans and the edges in E correspond to the conflicts that exist between steps of different plans. Thus, N = { i§0', i§s, i§s', for s = 1, 2, ..., S_i and i = 1, 2, ..., n}, and (i§s1, j§s2) $\in$ E if and only if i§s1, j§s2 $\in$ N, $i \neq j$, and i§s1 and j§s2 call for actions which utilize the same resource.

Note that edges of the CPCG reflect conflicts which exist between steps of different plans only. Before we describe the procedure used to combine the activation plans of n I-paths we have to define two dynamic lists.

Definition 5.8: List **Frontier-Steps(t)** contains all the steps that are candidates for scheduling during time t. Every I-path is represented by two steps in this list, the one on the **frontier** of its activation plan (i.e., the step to be executed next), and a No-Op in case the frontier step cannot be scheduled due to conflicts. Thus,

Frontier-Steps(t): { i§f_i , i§f_i-1' | f_i is the frontier step of P_i at time t, for i = 1, 2, ..., n }.

Definition 5.9: List **Scheduled-Steps(t)** contains n steps, one for every I-path. These steps are scheduled during time slot t. The elements of Scheduled-Steps(t) are selected from Frontier-Steps(t) subject to the condition that they do not conflict with each other. Thus,

Scheduled-Steps(t): { i§x | i§x ∈ Frontier-Steps(t), and no two steps in this set conflict with each other, i = 1, 2, ..., n }.

Procedure: Combine-I-paths(P_1, P_2,..., P_n)

```
BEGIN
  t = 0.
  Frontier-Steps(1) = { i§1, i§0' for i  = 1,2, ..., n}.
  REPEAT UNTIL all steps are scheduled
       1) Increment t.
       2) Generate Scheduled-Steps(t) from Frontier-Steps(t).
       3) Generate Frontier-Steps(t+1).
       END REPEAT
END
```

All the operations performed by the above procedure are straightforward except step 2 which calls for selecting Scheduled-Steps(t) from Frontier-Steps(t) according to the definition given earlier. There are 2^n-1 possible selections[3], however, some of them may not be feasible due to conflicts. Also it is desirable to select a solution with a minimal number of No-Ops so as to minimize the total length of the final plan. Even though the problem complexity is exponential, the value of n is often small (in the order of 2-4). Hence, an exhaustive search is feasible.

Next we describe a simple exhaustive procedure which can be used to generate Scheduled-Steps(t). The procedure takes as an input Frontier-Steps(t)

[3]The combination which consists of all n No-Op steps is not acceptable.

and the CPCG, and returns a feasible Scheduled-Steps(t) list with a minimal number of No-Op steps.

Procedure Find-Schedule (F-List, CPCG)

```
BEGIN
  L = { x | x is a frontier step in F-List}

  IF for all x, y ∈ L, (x, y) is not in CPCG THEN RETURN(L).
  k = 1              /* k is the number of No-Ops allowed in L

  REPEAT WHILE k ≤ n

                     /* form all possible schedules with k No-Ops

     C = { L | (1) L is a subset of F-List of size n,
               (2) for all x, y in L, x and y do not belong to
                   the same I-path, and
               (3) there are exactly k No-Op steps in L}

                     /* reject a schedule if it causes a conflict

     FOR every L in C DO
       BEGIN
       IF for all x, y ∈ L, (x, y) is not in CPCG THEN RETURN(L)
       END

     Increment k
    END REPEAT WHILE
END
```

The above procedure is guaranteed to find a Scheduled-Steps(t) list with the smallest possible number of No-Op steps.

If procedure "Find-Schedule" fails to find a schedule for a given Frontier-Steps(t) list, this indicates a **scheduling deadlock** at time t. The following Theorem provides a necessary condition that must exist for deadlocks to occur.

Theorem 5.10: If procedure "Find-Schedule" encounters a scheduling deadlock at time t, then the flow of data as specified by the frontier steps of the I-paths must be forming one or more loops.

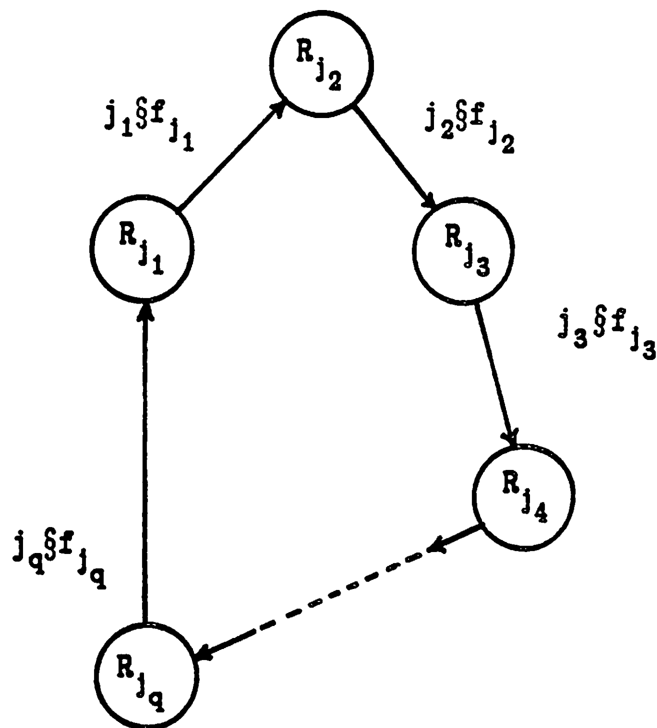
Proof: Assume that Frontier-Steps(t) is such that no Scheduled-Steps(t) list can be created. Hence a scheduling deadlock exist. Let us denote the registers that hold the data of P_i at time t-1 by R_i for i = 1, 2, ..., n. The deadlock implies the following. For all i = 1, 2, ..., n, Scheduled-Steps(t) = $\{i\S f_i, j\S f_j\text{-}1'$ for j = 1, 2, ..., n and $j \neq i\}$ is not feasible. In other words, for all i, the frontier step of P_i cannot be scheduled while holding the flow of data along all other I-paths. This implies that for all i = 1, ..., n, step $i{:}f_i$ moves data from R_i into one of the R_j for j = 1, 2, ..., n and $j \neq i$. But by definition $R_i \neq R_j$ for i, j = 1, 2, ..., n and $j \neq i$. Hence, the flow of data as specified by the frontier steps of the n I-paths forms one or more loops.

□

The situation described in the above proof is depicted in Figure 5-6, where the frontier steps of q I-paths form a loop. Only the registers used by the frontier steps are shown in Figure 5-6. Note, however, that in general a step of a test plan exercises I-modes of several combinational structures before finally reaching a register.

In practice, it appears that loops such as the one in Figure 5-6 do not exist. However, even if q I-paths form a loop, this is not a sufficient condition for a scheduling deadlock to occur. There are two reasons for this. First, it might be feasible to schedule the frontier steps that form the loop simultaneously without creating a conflict. In other words, it might be feasible to circulate the data among the registers of the loop, thus preventing a deadlock. Hence, in addition to the existence of a loop there must be at least one conflict between two of the frontier steps forming the loop for a deadlock to occur. Such a conflict prevent the circulation of the data.

The second reason why the existence of a loop is not a sufficient condition for a scheduling deadlock to occur is simply the fact that, it is quite



$j_k \in \{1,\ 2,\ \ldots,\ n\}$
$k = 1,\ 2,\ \ldots,\ q \leq n$

Figure 5-6: A loop formed by the frontier steps of q I-paths.

possible that steps which form a loop never belong to the same Scheduled-Steps(t) list. Clearly, steps which form a loop should arrive at the frontier of their respected activation plan at the same time for a deadlock to occur.

The next example illustrates the embedding process for a multiple port kernel and how to generate a test plan for it.

<u>Example</u> <u>5.3</u> Consider the circuit shown in Figure 5-7. The kernel has two input ports and two output ports. Assume that we are interested in embedding the BILBO TDM into this circuit. Consider a possible embedding where registers R1 and R2 assume the role of the BILBO driver node, while R9 and R10 assume the role of the BILBO receiver node.

There are two input I-paths to be combined. The activation plans of these two I-paths are given below.

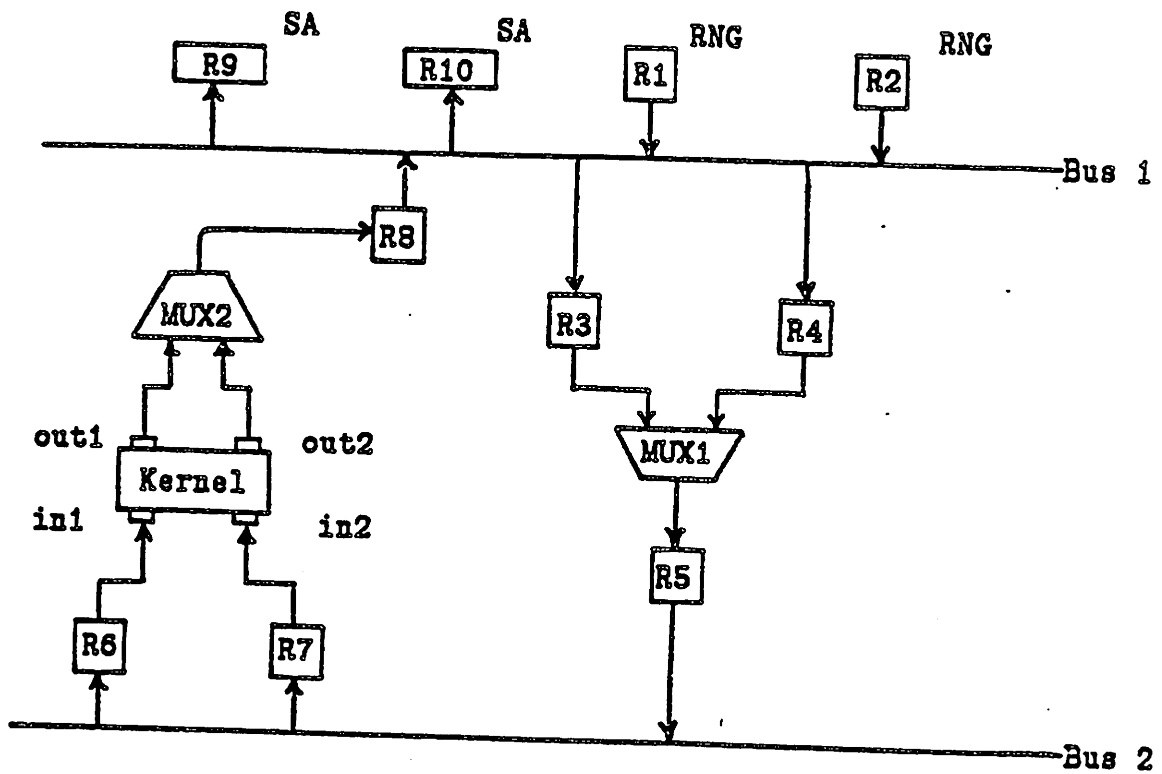


Figure 5-7: A circuit example with a multiple port kernel.

```
P1(R1:out → Kernel:in1) ≡
      1- Bus1(R1), R3(Latch).
      2- MUX1(R3), R5(Latch).
      3- Bus2(R5), R6(Latch).

P2(R2:out → Kernel:in2) ≡
      1- Bus1(R2), R4(Latch).
      2- MUX1(R4), R5(Latch).
      3- Bus2(R5), R7(Latch).
```

The CPCG of the activation plans of the input I-paths is shown in Figure 5-8. A trace of the lists generated by Procedure "Combine-I-Paths" while combining the two input I-paths is shown below:

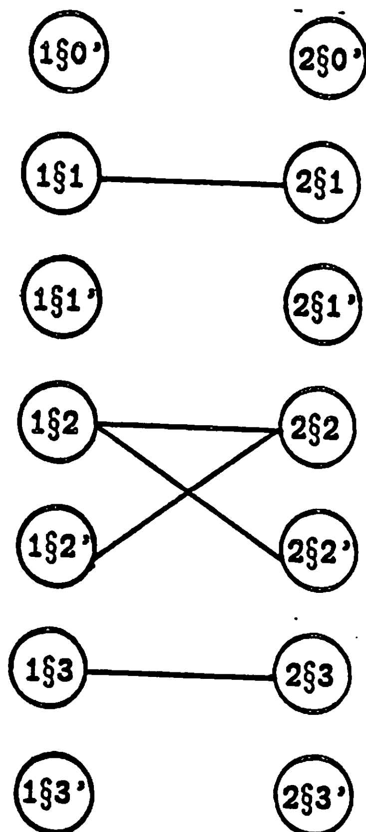


Figure 5-8: The CPCG of the activation plans of the input I-paths.

Time Slot	Frontier-Steps	Scheduled-Steps
1	1§1, 1§0', 2§1, 2§0'	1§1, 2§0'
2	1§2, 1§1', 2§1, 2§0'	1§2, 2§1
3	1§3, 1§2', 2§2, 2§1'	1§3, 2§2
4	1§3', 1§3', 2§3, 2§2'	1§3', 2§3

Note that, by definition, the No-Op step preceding P_2, 2§0', calls for holding the state of register R2 (the source of P_2). Thus, the combined activation plan of the input I-paths is given by

```
1- Bus1(R1), R3(Latch), R2(Hold).
2- MUX1(R3), R5(Latch), Bus1(R2), R4(Latch).
3- Bus2(R5), R6(Latch), MUX1(R4), R5(Latch).
4- R6(Hold),            Bus2(R5), R7(Latch).
```

Note that while combining the input I-paths, the test data has to be applied simultaneously to the kernel input ports. Hence, even though P_1 had been scheduled completely during time slot 3 above, a No-Op step, R6(Hold), is scheduled during time slot 4 to hold the data of P_1 until the rest of the test data arrives.

Similarly, there are two output I-paths to be combined. Their activation plans are given below.

```
P1(Kernel.out1, R9), R9(SA). ≡
        1- MUX2(Kernel.out1), R8(Latch).
        2- Bus1(R8), R9(SA).

P2(Kernel.out2, R10), R10(SA). ≡
        1- MUX2(Kernel.out2), R8(Latch).
        2- Bus1(R8), R10(SA).
```

The CPCG of the activation plans of the output I-paths is shown in Figure 5-9. A trace of the lists generated by Procedure "Combine-I-Paths" while combining the two output I-paths is shown below:

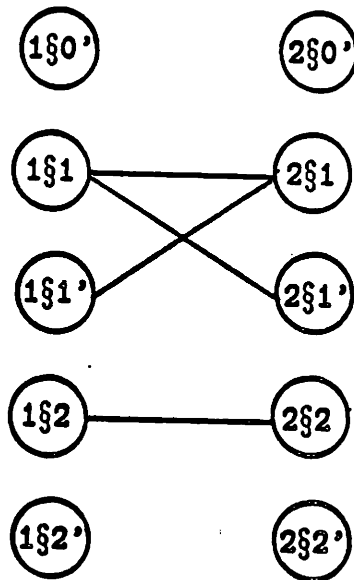


Figure 5-9: The CPCG of the activation plans of the output I-paths.

Time Slot	Frontier-Steps	Scheduled-Steps
1	1§1, 1§0', 2§1, 2§0'	1§1, 2§0'
2	1§2, 1§1', 2§1, 2§0'	1§2, 2§1
3	2§2, 2§1'	2§2

Note that according to Definition 5.6 the No-Op step 2§0' calls for holding the state of the register(s) holding the test data in the step preceding P_2. Thus 2§0' ≡ R6(Hold), R7(Hold). Note also that, during the last time slot in the above list, P_1 is not represented in Scheduled-Steps because all its steps had been scheduled already. No No-Op steps are appended to P_1 because signature analysis of the two output response of the kernel need not be done simultaneously.

By substituting the combined plans for the abstract I-paths of the BILBO TDM, the following test plan is generated.

```
1-  R1(RNG),                 R2(RNG).
2-  Bus1(R1), R3(Latch), R2(Hold).
3-  MUX1(R3), R5(Latch), Bus1(R2), R4(Latch).
4-  Bus2(R5), R6(Latch), MUX1(R4), R5(Latch).
5-  R6(Hold),                Bus2(R5), R7(Latch).
6-  Kernel, MUX2(Kernel.out1), R8(Latch), R6(Hold),R7(Hold).
7-  Bus1(R8), R9(SA), Kernel, MUX2(Kernel.out2), R8(Latch).
8-                           Bus1(R8), R10(SA).
```

□

5.3. Embedding Measures

From the previous discussion one can see the range of possibilities which can be encountered during the embedding process of a given kernel. On the one hand, there are usually several TDMs applicable to a kernel. On the other hand, for each of these TDMs, there is often many feasible embeddings. In order to evaluate and compare these embedding solutions, a number of measures are collected for each embedding. Figure 5-10 illustrates the key embedding measures. The measures can be partitioned into two categories, namely circuit-related and TDM-related.

Circuit-related measures are a function of circuit details related to an embedding, such as the particular circuit structures used for matching and the length and form of the I-paths used. On the other hand, TDM-related measures are functions of the characteristics of the TDM used and the kernel itself. Hence, by definition, the values of all TDM-related measures are constant for all embeddings of the same TDM and kernel.

Furthermore, the measures of each category can be partitioned into two groups. Measures in the first group, referred to as **local measures**, reflect various costs and gains related to how an embedding affects the *kernel* under consideration. Measures in the second group, referred to as **global measures**, deal with an embedding from a global point of view, hence projecting how the

Measure	Group	Category
Area overhead (AO)	Local	Circuit-related
Schedule initiation delay (D)	Local	Circuit-related
Performance degradation (PD)	Local	Circuit-related
Sharing potential (SP)	Global	Circuit-related
Modes exercised for free (MEF)	Global	Circuit-related
Test length (T)	Local	TDM-related
Test time	Local	TDM-related
Preprocessing cost	Local	TDM-related
ATE cost	Local	TDM-related
Storage requirement	Local	TDM-related
Homogeneity	Global	TDM-related
TDM bias	Global	TDM-related

Figure 5-10: Embedding evaluation measures.

embedding affects the entire circuit and how well it fits into the global test strategy.

In the next subsections we describe in detail the various key measures in each group and how they can be computed.

5.3.1. Circuit-related Measures

Circuit-related measures deal with detail circuit aspects of an embedding. The main purpose of these measures is to evaluate and compare various feasible embeddings of a given TDM into a circuit.

5.3.1.1. Local Circuit Measures

There are three key measures which belong to this group, namely, area overhead, test schedule initiation delay, and performance degradation.

Area Overhead

The area overhead measure associated with an embedding reflects the extra circuit area required for one or more of the following functions.

- Modify and/or add circuit structures to match TDM template nodes.
- Add multiplexers to create I-paths.

Recall that we are dealing with designs at a high level where the implementation detail of many of the structures may not be known. Moreover, without knowing the physical location of the structures in a circuit, calculating routing area becomes impractical As a first order approximation one can assume that chip area grows linearly with the increase in the number of gates or transistors [Kurdahi 85]. Hence, the extra number of gates or transistors added to a design can be used as a measure of the total area overhead.

In order to calculate the exact area overhead associated with a particular circuit modification, one has to carry out placement followed by channel routing [Syed 81] for both the original circuit and the modified one.

In practice, most of the circuit modifications required to obtain an embedding involve modifying registers to increase their functional capabilities. The penalty associated with modifying a register R to accommodate certain functional capabilities depends on the original functional capabilities of R. For example, the area required to modify a register which can latch and hold data to become an LFSR, is larger than the area required to modify a shift register to become an LFSR. A good scheme for handling register modifications is to precompute the area associated with every possible multiple mode register and store it as a part of the knowledge base of TDES. Thus, given a register R with modes m_1, m_2, ..., m_i, a measure of its estimated area can be gleaned directly from the knowledge base. Assume that it is desired to modify R to accommodate some additional modes a_1, a_2, ..., a_j. The area overhead associated with this modification equals the difference between the original area estimate of R and the area estimate of a register with modes m_1, m_2, ..., m_i, a_1, a_2, ..., a_j.

It is quite possible for a register to be modified more than once to satisfy embedding requirements of different kernels. Clearly, the cost associated with one modification depends on the order of execution of the modifications. However, the total area overhead is independent of such order.

Another kind of modification which is often encountered in TDM embeddings involve adding well defined structural features to kernels. Examples are adding extra rows and columns to a PLA [Breuer 85a], and increasing the width of a ROM or a RAM to incorporate parity bits. The area overhead associated with these kind of modifications can be calculated using

standard functions of various kernel parameters. For example, the area overhead for adding an extra product term in any PLA designed using the Mead and Conway VLSI design rules [Mead 80] is $8\times(16n + 8k + 37)\ \lambda^2$, where n and k are the number of PLA inputs and outputs, respectively.

Test Schedule Initiation Delay

The test schedule initiation delay measure affects the total time required to execute the test plan of an embedding. In the next chapter this measure is discussed in detail and techniques for calculating and minimizing this measure are presented. In order to clarify this concept, the test plan of Example 5.3 consists of 8 steps, and is repeated T times. If no optimization is carried out, executing this test plan would require $8 \times T$ clock cycles. In the next chapter we describe how the test time can be reduced to $4 \times T$. The "4" in the previous test time expression is the optimal test schedule initiation delay associated with the embedding of Example 5.3.

Performance Degradation

The performance degradation measure deals with the effect on a circuit's operating characteristics during its normal operation due to built-in test hardware. For example, modifying a simple latch register to become an LFSR increases the set-up delay associated with the latch mode of the register due to the additional control encoding required. Such a delay might be of little importance if the delay along the register input path is not critical with respect to the system clock rate. However, if that register is the sink in a critical path, then the extra delay may require a reduction in the system clock rate.

Clearly, to determine the performance degradation associated with an embedding, the timing characteristics of the CUC must be known. Typically, such information is usually produced by timing analysis routines and simulation runs which support design synthesis.

Instead of dealing with detailed timing information, we assume that every circuit structure s has a slack variable τ_s which indicates the amount of timing **freedom** associated with that structure. For example, $\tau_s = 0$ indicates a structure which is a part of a critical path and hence modifying it by adding extra logic levels forces the use of a slower system clock. On the other hand, $\tau_s > 0$ indicates a structure where the introduction of additional delay $\leq \tau_s$ can be absorbed without slowing the system speed. We assume that the freedom variables associated with circuit structures is specified as part of the design input description to TDES. Furthermore, we assume that if the freedom variable of a structure is set initially to zero, then this indicates that no modification which introduces a delay should be made to that structure.

The amount of delay associated with certain circuit modifications can be calculated using complicated timing analysis tools. However, similar to what we did in the area overhead case, a table can be constructed indicating the delay associated with common types of modifications. For example, an entry in that table might indicate that the delay associated with inserting a MUX in the input path of a structure is 20 nanoseconds. The extra set-up delay associated with the latch mode of a register due to the added modes of operation, can always be made equivalent to that resulting from the insertion of a MUX in the input path of that register.

The performance degradation , PD, measure associated with a particular circuit modification which introduces an additional delay t along an input path of a structure s is given by

$$PD = t / \tau_s.$$

The relationship between the PD measure versus t as a function of τ_s is illustrated in Figure 5-11. Note that the PD measure is normalized such that it

is equal to 1 at τ_s = t. The PD measure is undefined for τ_s=0, however, as we already mentioned we assume that structures with 0 freedom are not modifiable.

The PD measure associated with an embedding equals the sum of all the PD measures of the modifications required in that embedding.

Once a certain modification is carried out in the CUC, the freedom variable of all the affected structures are reduced by the appropriate amount. In case the amount t of additional delay introduced is larger than τ_s of an affected structure, then τ_s becomes negative indicating that a slower clock rate is needed.

It is interesting to note that, unlike area overhead, if a register is modified more than once to satisfy embedding requirements of different kernels, the additional delay introduced is independent of the number of extra modes added. Hence, the freedom variable of that register is reduced only once when the register is first modified.

5.3.1.2. Global Circuit Measures

The aim of this group of measures is to project the goodness of a local embedding from a global point of view. There are two key measures which belong to this group, namely, sharing potential and modes exercised for free.

Sharing Potential

The sharing potential (SP) measure indicates the potential degree of sharing of BIT structures (created for a local embedding) with other kernels in the CUC. For instance, an embedding utilizing registers which belong to the subgraphs of several kernels may lead to a significant amount of sharing of resources, and hence a lower total area overhead. Thus, even though a specific embedding may have a high area overhead, if several kernels can take

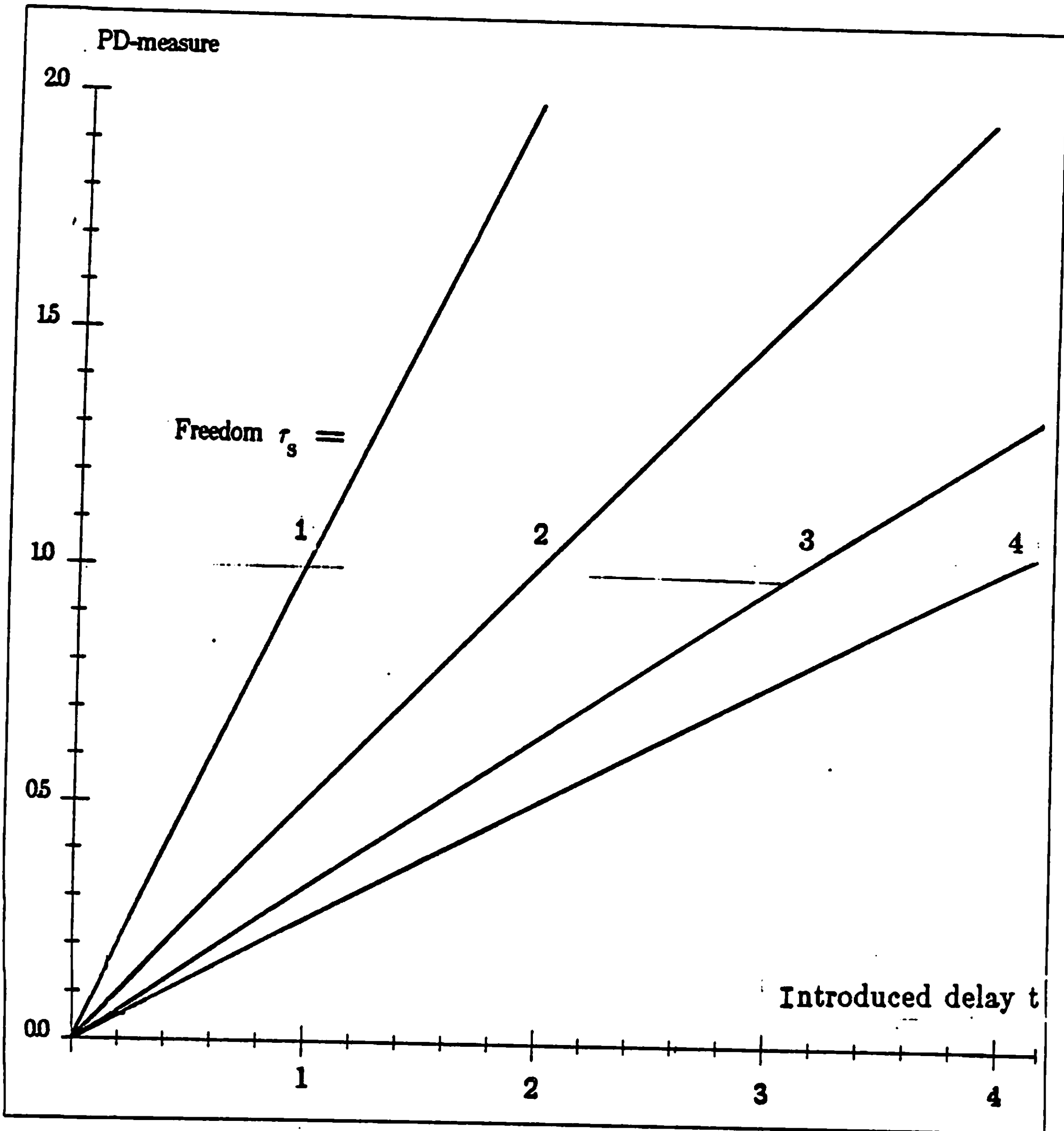


Figure 5-11: The PD measure versus the introduced delay t as a function of the freedom of the modified structure τ_s.

advantage of the added BIT structures, a globally testable design with a minimal area overhead may result.

Basically, the SP measure of an embedding acts as a counter balance to the area overhead and PD measures. The SP measure of an embedding E, denoted by SP(E), is computed as follows.

1. For every circuit structure s which gets modified as a part of E, compute its sharing potential measure SP(s) as follows.
   a. Let $s_d = \{ k \mid k$ is a kernel which has not been processed yet and there exist an I-path $P(s \rightarrow k)\}$.

   b. Similarly, let $s_r = \{ k \mid k$ is a kernel which has not been processed yet and there exist an I-path $P(k \rightarrow s)\}$.

   c. SP(s) is given by

$$SP(s) = \begin{cases} 2|s_d| + |s_r| & s \text{ is a driving node in } E \\ |s_d| + 2|s_r| & \text{otherwise} \end{cases}$$

   where $|x|$ denotes the size of set x.

2. $SP(E) = \sum_{\text{for all } s} SP(s)$.

Note that even though a register r modified as a part of E might not directly match embedding requirements of other kernels, the area overhead required to modify r for the second time to get a match is usually less than the area overhead required if r is to-be modified for the first time. Moreover, the PD measure associated with the second modification of r is usually zero. Note also that the SP(s) measure takes into account whether s is a driver node or a receiver node. Clearly, if s is used as a driver in E then the probability that s will closely match driver requirements for other kernels is usually bigger than the probability of s closely matching receiver requirements of other kernels.

A kernel which does not have an I-path either from or to a newly created BIT structure, will not count in the SP measure. This is despite the fact that MUXs can be used to create I-paths between any BIT structure and any kernel. However, adding MUXs involve an additional area overhead cost and possibly performance degradation, hence only existing I-paths are considered when calculating the SP measure.

Modes Exercised for Free

Another measure which belongs to the global circuit measures group is the degree to which parts of the circuit are tested while testing a kernel. For example, an embedding with a long test plan will exercise a lot of circuitry while testing a kernel. This may eliminate the need to consider testing this circuitry explicitly, hence reducing both the overall testing time and possibly the global area overhead. For example, consider a register r used as a part of an I-path driving one port of a kernel. While testing the kernel, r will be exercised in the latch mode of operation and possibly in the hold mode of operation as well. Thus, depending on how thorough the kernel is tested, we might conclude that there is no need to test the exercised functions of r individually. Note that if r has other modes of operation that were not exercised in the test plan, then it may be necessary to exercise such modes explicitly.

Exercising a mode of operation of a structure as a part of a test plan is not a guarantee that this mode of operation has been fully tested. To test whether a mode of operation is fully tested or not one has to examine the characteristics of the data transferred along the I-path. For example, a register used in the latch mode of operation to transfer exhaustive or random test data into a kernel is exercised thoroughly and hence the latch mode of operation of that register is tested for free. Such a conclusion might not be true for a multiplexer used to transfer the final signature as a part of the tail section of a test plan.

The following two lemmas illustrates the relationship between kernel faults and faults affecting structures along an I-path used to either drive or receive data from that kernel.

Lemma 5.11: Stuck-at or short faults affecting the exercised memory elements, input lines, or output lines of a register used as a part of an I-path to drive (receive) data into (from) a structure s are indistinguishable from faults affecting the input (output) lines of s.

Lemma 5.12: Stuck-at or short faults affecting the exercised input lines or and output lines of a MUX or a bus which is utilized as a part of an I-path to drive (receive) data into (from) a structure s are indistinguishable from faults affecting the input (output) lines of s.

The modes exercised for free (MEF) measure of an embedding is simply calculated as the number of modes of operation invoked by the test plan of that embedding. Thus even though exercising a mode in a test plan might not correctly indicate whether that mode is fully tested or not, it will count in the MEF measure of an embedding. This is justified because the purpose of the MEF measure is not to specify exactly what get tested for free, but as a probabilistic indicator of what might get tested for free in a certain embedding. In Chapter 7 the problem of determining whether a circuit structure is thoroughly tested or not is examined.

5.3.2. TDM-related Measures

TDM-related measures are functions of the TDM and the kernel itself. Hence, measures in this category are independent of the circuitry surrounding the kernel and the circuit embedding details. Measures that depend only on the TDM can be gleaned directly from the appropriate slots in the TDM frames.

TDM-related measures are mainly used to compare embeddings of different TDMs for the same kernel.

5.3.2.1. Local TDM Measures

There are many measures which belong to this group. Some of these measures are not independent in the sense that they can be represented as a function of other measures. Next we present five important measures which belong to the local TDM group.

Test Length

For most TDMs the number of test vectors used is a function of the kernel under consideration, the fault model used, and the degree of fault coverage required. Computing the exact value of such a measure is a very difficult problem in most cases involving the execution of test generation and fault simulation programs.

Clearly, since our goal is to compare different embeddings, one can rely on the use of easy to calculate estimated values rather than difficult to determine exact values. Recall that the test length slot in a TDM frame contains the name of an estimator function. This function takes as an input data about the kernel under test and a degree of fault coverage F, and returns an estimate of the test length required to achieve F. An example estimator function was given earlier in Chapter 3 for the BILBO TDM. We assume that F is specified by the user.

In general, test sets associated with TDMs can be classified into one of four categories.

- Algorithmically generated test sets.
- Random test sets.
- Exhaustive test sets.

- Precomputed or fixed test sets.

For example, BILBO employs a random test set while traditional scan path TDMs usually employ algorithmically generated test sets. The size of an algorithmic or random test set, in general, increases with the size of the kernel, the number of inputs, and the degree of fault coverage desired. The size of an exhaustive test set is a function of the kernel inputs and number of kernel states, if any.

Test Time

The test time associated with a particular TDM embedding can be computed using the function specified in the test time slot of the TDM frame. For most TDMs, test time equals the product of test length and the test schedule initiation delay. The values of these two factors are themselves considered as embedding measures. However, the test time associated with certain TDMs, such as SPLASH [Abadir 85d], depends on other parameters.

Preprocessing Cost

The preprocessing cost measure deals with the off-line process of test creation and the associated costs of acquiring and executing the required software. Typical preprocessing tasks include test generation, fault simulation, signature calculation, and ATE program preparation. A list of the software packages required by a particular TDM can be gleaned directly from the test creation software slot in the TDM frame. For every software package there are two kinds of costs. The first is the cost of acquiring a program if it is not available, and the second is the cost of executing it. Note that the cost of acquiring a program is a one-time cost. Hence to calculate the preprocessing cost of a particular embedding one first has to check the availability of the required software. Once a particular software package is acquired, it is marked in the knowledge base as being available. Initially, the user should specify the list of available software packages.

ATE Cost

The ATE cost measure associated with a TDM embedding can be directly obtained from the appropriate slot of the TDM frame. Recall that the value of this measure is an integer between 0 and 10 reflecting the size, speed, and complexity of automatic testing equipment (ATE) required by a TDM. The ATE cost measure is a one-time cost incurred only when a TDM is first used for a circuit kernel. Embeddings of the same TDM for consecutive kernels have 0 ATE cost.

Storage Requirement

The storage requirement measure of a TDM embedding is calculated using the function specified in the storage requirement slot of the TDM frame. Clearly, this measure is a function of how the testing process is carried out in a particular TDM, and on the number of test patterns used. For example, the storage requirement for the scan path TDM equal test length multiplied by the sum of the number of kernel inputs and outputs.

5.3.2.2. Global TDM Measures

There are two measures classified as global TDM measures. Their basic role is to direct the system towards better global solutions.

Homogeneity Measure

For practical reasons it is desirable to generate a homogeneous global solution, i.e., a solution which employs a small set of TDMs, rather than a solution employing a different TDM for every kernel. By so doing, many of the one-time costs are evenly distributed among the circuit kernels, and more important, it is often feasible to share BIT structures.

One way of applying such a guideline is to favor TDMs that have been used already in a circuit over other TDMs. In particular we attach a measure to each TDM frame indicating the number of circuit kernels which have

already used this methodology. Initially, the values of such measure are all set to zero. The higher the value of a TDM homogeneity measure, the more likely that it will be selected over other applicable TDMs with lower homogeneity measure values. This help leads to a homogeneous global solution.

TDM Bias Measure

The sole purpose of the TDM bias measure is to favor the selection of certain TDMs over others. In view of the design goals and constraints, it might be possible to make decisions such as which group of TDMs are more likely to lead to good solutions, or which group of TDMs should be avoided or only used if really needed. The TDM bias measure is used to reflect such global decisions.

5.4. Evaluating Embeddings

So far in this chapter we described how to generate all feasible TDM embeddings for a given kernel and the measures used to evaluate these embeddings. The next step is to select one of these embeddings and actually instantiate it by performing the necessary circuit modifications. Clearly, to be able to compare several embeddings, one needs a scheme for combining the values of different measures.

A two-phase scoring process is used by TDES. First, the circuit-related measures of all embeddings of the same TDM are combined into one score, and based on that score the best embedding(s) for each TDM is selected. Note that, by definition, the TDM-related measures for embeddings of the same TDM are constant, hence they need not be considered during this phase. At the end of this phase, we obtain a set of embeddings, one (or more) for each TDM applicable to the kernel. For each member of this set, we combine both the circuit-related and the TDM-related measures into a second score. Then based on that score the best TDM embedding is selected and instantiated.

Note that at both levels of evaluation, the local and the global measures are considered thereby directing the search toward a good global solution. Note also that by partitioning the evaluation process into two phases, we eliminate the need to compute TDM-related measures for most of the embeddings. This minimizes the time required to evaluate and select embeddings.

5.4.1. Scoring Functions

There are many ways for combining the values of different measures into a single score. The simplest form of a scoring function is as follows.

$$\text{Score} = \sum_i \text{Meas}_i \cdot \text{Norm}_i \cdot \text{Prior}_i$$

where Meas_i is the value of measure i, Norm_i is the normalized weight of measure i, and Prior_i is the priority weight assigned to measure i.

The normalization weights are constants used to map the units associated with the various measures to a standard unitless reference. The priority weights are application dependent and user specified, and are used to adjust contributions of the various measures in the embedding score. The values of these priority weights are initially set to a fixed constant, say 1. However, they can be adjusted to reflect the design constraints. In Chapter 7 we will describe how the priority weights can be used in adjusting the global strategy of TDES.

Chapter 6
Test Plan Scheduling

6.1. Introduction

In general a test plan deals with a repeated application of the following steps: (1) generate a test vector; (2) transmit it to the structure being tested; (3) process the test through the structure; (4) obtain the response from the structure; and (5) process the response. Because these steps must be repeated for each test vector, it is possible that steps used in processing one test vector can overlap those used in processing another vector. The manner of overlapping this testing process leads to the concept of a **test schedule**. The execution time of test schedules is used as one of the measures for evaluating embedding solutions. Clearly, it is of interest to construct test schedules that optimizes total test time.

This chapter deals with the task of creating a schedule for a given test plan which minimizes the total time required to test the structure. A model for describing test schedules and the conflicts that may arise due to the parallelism in these schedules is presented. Lower bounds on the time required to execute a test plan are derived, and algorithms for constructing optimal test schedules are then presented. The concept of optimizing test schedules is similar to that used in optimizing static pipelines [Kogge 81]. Yet, there are key differences between the two problems, and hence our approach and results are tailored to the problem in hand. Those differences are discussed at the end of the chapter.

6.2. Exploring Parallelism in Test Plans

To execute a test plan, the steps of its body have to be executed (iterated) T times, where T is the number of test vectors to be used. In addition the actions in the head and tail sections have to be executed once. These initialization and closing actions play a very minor role in determining the total execution time of a test plan, and hence they will be ignored. Unless explicitly stated otherwise, in the rest of our discussion we will use the term test plan to refer to the body of a test plan.

Consider a test plan that consists of S steps, and assume that the steps are numbered 1, 2, ..., S. Let T be the number of test patterns to be used. The simplest way to execute a test plan is in a sequential manner, hence requiring $S \times T$ clock cycles. On the other hand, it is often possible to overlap the execution of consecutive iterations, hence reducing the overall test time. In other words, it might be possible to start the execution of one iteration before the end of the previous one.

Definition 6.1: The **initiation delay** of a test schedule, denoted by D, is the number of clock periods between the initiation of two consecutive iterations.

$D=1$ indicates maximum overlapping, while $D=S$ implies sequential execution. Figure 6-1 displays all possible **execution schedules** for a **generic** five steps test plan. The numbers in the leftmost column represent the time in clock cycles, while the other numbers represent the steps. The third line of Figure 6-1a indicates that during the 3rd clock cycle, step 3 of the 1st iteration, step 2 of the 2nd iteration, and step 1 of the 3rd iteration will be executing in parallel.

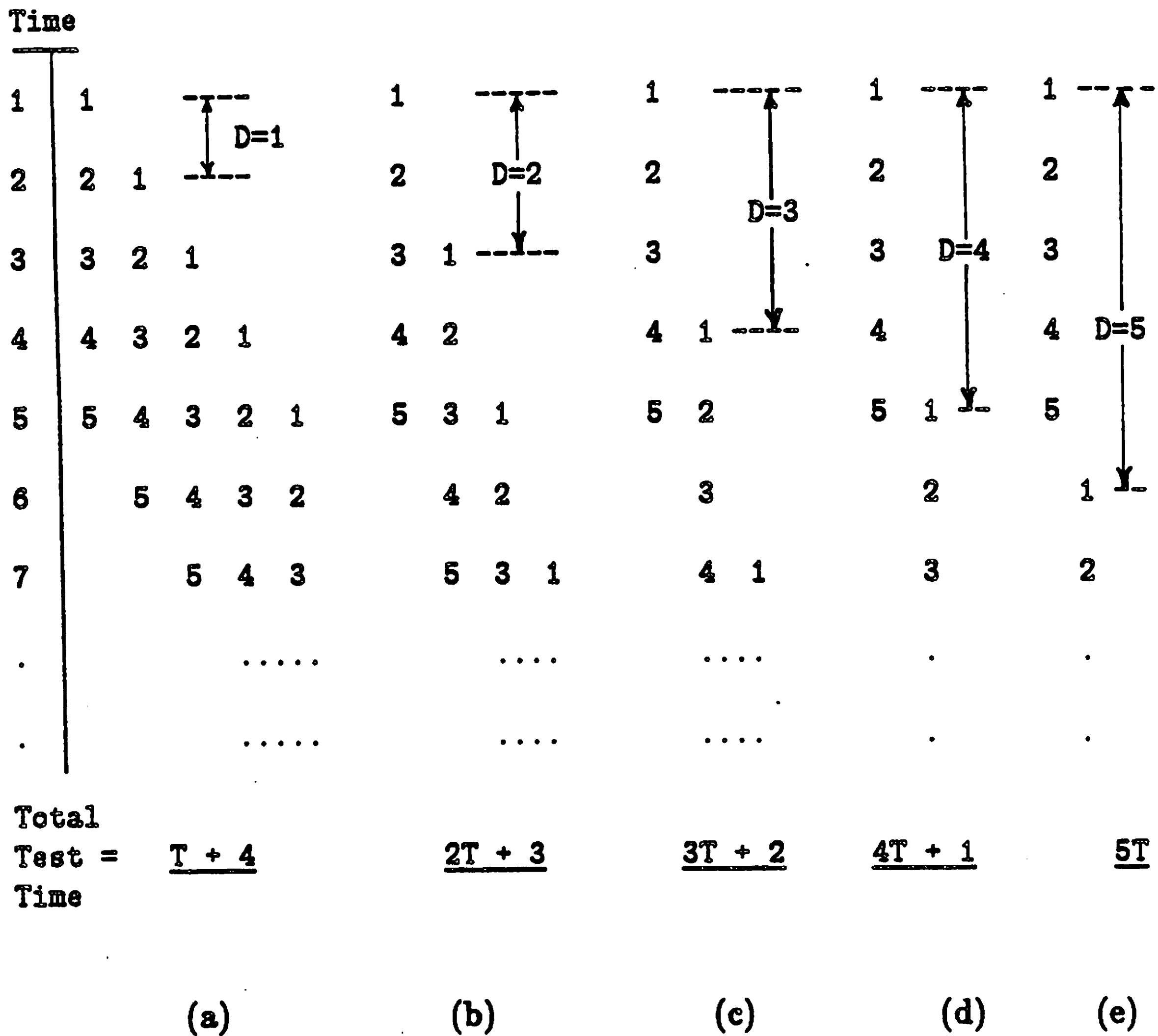


Figure 6-1: Various execution schedules for a generic 5 step test plan.

6.2.1. Cyclic Behavior of the Execution Schedules of Test Plans

By analyzing execution schedules of test plans with different values of D, one can observe the following cyclic behavior of test plans. Assuming T is greater than S, after the first $\lfloor S/D \rfloor \times D$ clock periods every schedule starts to **cycle** with a period equal to D. We will refer to the D time slots of one cycle as the D **phases** of the schedule. For example, consider the execution schedule shown in Figure 6-2. The schedule has a cycle with two phases.

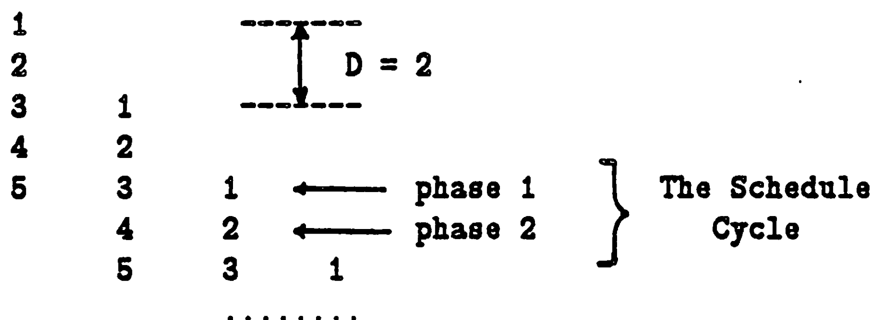


Figure 6-2: An execution schedule with 2-phase schedule cycle.

Note that every step of the plan is executed once every D clock periods. Thus, one can consider D as being the **effective length** of the test plan.

Let $t_i(r)$ indicate the execution time of step i of iteration r, for r=1, 2,..., T. Assume $t_i(1) = i$, for i = 1, 2, ..., S. Hence $t_i(r) = i + (r-1)D$, and the **total test time** is $t_S(T) = S + (T-1)D$.

Note that since T is usually large compared to S, the total test time is much more sensitive to D than to S.

6.2.2. The Conflict Graph

Clearly, it might not be possible to realize an execution schedule with an arbitrary value of D due to resource sharing conflicts. Thus, to explore the feasibility of these schedules, it is necessary to identify those steps in the test plan which cannot run in parallel either because they use the same hardware or they require conflicting control signals.

Example 6.1 Consider the 4 step test plan shown below which is reproduced from Figure 5.2b.

```
1   R1(RNG).
2   Bus(R1), R3(Latch).
3   k, MUX(k), R4(Latch).
4   Bus(R4), R2(SA).
```

Both steps 2 and 4 utilize the bus, hence they cannot be executed concurrently. This eliminates the applicability of the schedules with D=1 or D=2.

Clearly, knowledge about conflicts that exist between the steps of a test plan is essential in order to analyze the feasibility of different execution schedules.

Definition 6.2: Two steps of a test plan **conflict** if they utilize a common resource, or they call for actions that require conflicting control signals.

Definition 6.3: A **conflict graph** CG(N, C) of a test plan represents conflicts which exist between test plan steps. The nodes in N correspond to the test plan steps, while the edges in C represent the conflicts. Thus, $N = \{1, 2, ..., S\}$, and $(i, j) \in C$ if and only if step i and step j conflict.

For example, the CG of the test plan of Example 6.1 has one edge (2, 4) since steps 2 and 4 utilize the bus.

6.2.2.1. A Process for Constructing the CG of a Test Plan

If a structure in a circuit is used in q steps of a test plan, then the CG of that test plan has a clique of size q between the nodes corresponding to those q steps. A simple mechanism to generate the CG of a test plan is to form a clique for every structure used in at least two steps of the plan. Now by superimposing all these cliques we will get the CG. Note that if a structure is only used once in a test plan, its corresponding clique is of size 1, which is a singular node.

6.2.3. Finding Feasible Values for D

Clearly it is of interest to find the minimum value of D that leads to a feasible execution schedule for a given test plan. The following theorem provides a simple rule that can quickly be used to solve this problem.

Theorem 6.4: An execution schedule of a test plan with an initiation delay D is feasible if D does not divide (j - i) for all $(i, j) \in C$, where C is the set of edges in the CG and $j > i$.

Proof: By contradiction. Assume an execution schedule with an initiation delay D is feasible, such that D divides (j-i), where $(i, j) \in C$ and $j > i$. Hence, $(j-i)/D = p$ where p is an integer. Thus, $j = i + D\,p$. Recall that the execution time of step s of iteration r is $t_s(r) = s + (r-1)\,D$. Hence, $t_j(1) = t_i(p+1)$ which means that step j of iteration 1 is scheduled in parallel with step i of iteration p+1. Thus, the schedule is infeasible which leads to a contradiction.

□

Corollary 6.5: D=1 is void if any conflict exists since 1 divides all integers.

Corollary 6.6: D=S is always feasible as S doesn't divide (j - i) for all possible steps i and j ($1 \leq i < j \leq S$).

Example 6.2: Consider the CG shown in Figure 6-3. Both D=2 and D=3 are infeasible due to the conflicts (6,4) and (4,1), respectively. Also both D=5 and D=6 are infeasible due to (7,2) and (8,2), respectively. D=4, D=7 or D=8 are feasible.

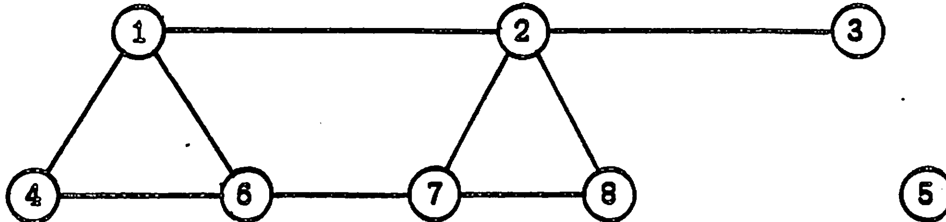


Figure 6-3: The CG for Example 6.2.

6.3. Lower Bounds on D

Theorem 6.7: The size of the largest clique in the CG of a test plan is a lower bound on any feasible value for D.

Proof: Let Q be the size of the largest clique in the CG, and let the steps that form that clique be denoted by $x_1, x_2, ..., x_Q$, where $x_1 < x_2 < ... < x_Q$. Assume that an execution schedule with an initiation delay $D \leq Q\text{-}1$ is feasible. Using the rule derived earlier for selecting feasible values for D, this implies that D does not divide $x_j - x_i$, for all j, i = 1, 2, ..., Q and j>i. Hence, $[x_j]_{\text{mod } D} \neq [x_i]_{\text{mod } D}$, for all j, i = 1, 2, ..., Q and j>i. But since there are only D possible values in a modulo D space, the above inequality cannot hold for Q different steps. Hence a contradiction.

□

Even though finding the largest clique in a graph is a known NP-complete problem [Bondy 76], the situation here is relatively simple. This is because of the one to one correspondence that exists between circuit structures and cliques in the CG. For example if register A is involved in actions in steps $x_1, x_2, .. , x_Q$ of a test plan, then there must be a clique in the CG between those steps. Thus, by identifying the structure used most often in a test plan,

a close estimate of the size of the largest clique in the CG is obtained. Clearly, this is a very fast way for obtaining a close lower bound on the optimal value of D. To find the largest clique, one has to check if the cliques that correspond to circuit structures combine producing larger cliques. Note that Q+1 cliques of size Q are needed to form a clique of size Q+1.

Theorem 6.8: The chromatic number of a CG is a lower bound on the value of D of any feasible test plan execution schedule.

Proof: By contradiction. Assume the chromatic number of the CG is X. Now assume that an execution schedule with D less than X is feasible. This implies the existence of schedule cycle with D phases, such that steps in the same phase are not connected in the CG. Hence the steps in each phase can be colored with one color. Thus, D colors are needed to color the CG, which leads to a contradiction.

□

The lower bound of Theorem 6.8 is stronger than the one of Theorem 6.7 [Bondy 76]. Coloring the CG with X-colors is equivalent to partitioning the steps of a plan such that the members of any partition do not conflict. The existence of such an X-way partition does not necessary imply the existence of a feasible execution schedule with D=X. For example, in order for a D=2 schedule to be feasible for any test plan, all the odd steps should be executed in one phase, and all the even steps in the other phase. Thus, even though there might be many feasible 2-way partitions, only one leads to a feasible schedule without changing the structure of the original test plan. In the following section we describe a technique for constructing test schedules which meet the lower bound of Theorem 6.8.

6.4. Inserting No-Op Steps

Theorem 6.4 provides a simple rule for checking the feasibility of different values of D. This, in general, leads to a minimal value of D which is far from the theoretical lower bounds. It will now be shown that the gap between the feasible value of D and the theoretical lower bounds can be eliminated by introducing No-Op steps in the plan body.

Definition 6.9: A **No-Op** step following step i of a test plan, denoted i', consists of actions that call for holding the state of all the register(s) used in step i for one clock period.

The next example illustrates this idea.

Example 6.3: Consider the test plan of Example 6.1. The smallest feasible value for D was 3, leading to a total test time of 3T+1. By introducing a No-Op step after step 2 that calls for holding the state of R3, the conflicting steps are now separated by 3 clock periods. Hence, D=2 is possible, as illustrated in Figure 6-4, leading to a total test time of 2T+3. Note that the new value of D equals both theoretical lower bounds, and hence it is optimal.

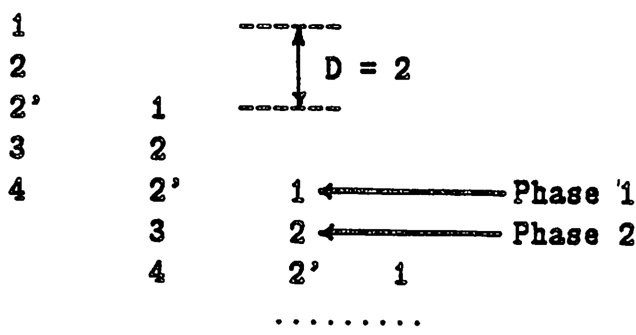


Figure 6-4: An optimal execution schedule (D=2) obtained using No-Op steps.

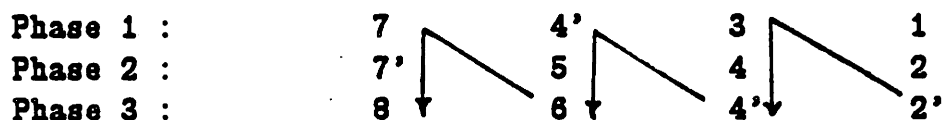
Before investigating the amount of improvement that can be reached using No-Op steps, it should be noted that a No-Op step may conflict with other steps of a test plan. Such a case is rare and is addressed later in Section 6.4.

Theorem 6.10: Assuming that No-Ops do not conflict with other steps of a test plan, by adding No-Op steps it is always possible to find a schedule with D equal to the chromatic number of the CG.

Proof: Let X be the chromatic number of the CG associated with a test plan. It follows that one can partition the steps of the plan into X groups, G_1, G_2, ..., G_X, such that the steps in one group do not conflict with each other. It is always possible to insert No-Op steps in the test plan such that if steps i and j belong to the same group, then the difference between the execution times of i and j in one iteration is X or a multiple of X. Now by using an execution schedule with D=X, a one to one correspondence between the G's and the schedule phases is created, i.e., the steps that belong to one group are all scheduled to execute in the same phase. This in turn implies that no conflicts will arise, and hence the execution schedule of the new test plan with D equal to X is feasible.

□

Example 6.4: Consider the 8-steps test plan of Example 6.2 with the CG shown in Figure 6-3. Without adding No-Ops, D=4 was the smallest possible initiation delay. However, the chromatic number of the CG is 3. Assume that the steps are partitioned as follows: G_1 = {1, 3, 7}, G_2 = {2, 4, 5}, and G_3 = {6, 8}. Following the argument used in the proof of Theorem 6.10, an optimal execution schedule with D=3 is constructed as shown below.



Four No-Op steps were added to the test plan. One can see the one-to-one correspondence between the phases of the schedule cycle and the partitions. Note that given the schedule cycle, one can find the test plan by traversing the columns of the cycle from right to left.

The number of No-Op steps added depends on the distribution of the steps in the partitions supplied as input. Clearly these partitions are not unique, and some will result in adding more No-Ops than others. For instance, in the previous example a test plan with only two No-Op steps inserted between steps 3 and 4 also supports a schedule with D=3. Every No-Op step added to a test plan increases the total test time by only one clock period. Such a small increase can usually be ignored when compared to the reduction due to the use of an optimal value for D.

The problem of coloring a graph (to partition the nodes of the CG) is in general an NP-complete problem [Bondy 76]. The following procedure avoids this expensive preprocessing step, and only assumes knowledge of a lower bound for D, denoted by B, such as the size of a large clique in the CG.

Procedure 1: (CG, B) This procedure will attempt to construct an execution schedule with B phases. The procedure has a loop that cycles through the B phases. In each iteration either a step is added to the current phase if it does not create conflicts, or a No-Op is added. Adding B No-Ops in a row implies failure, and the procedure restarts using a larger value for B. Procedure 1 is a fast heuristic which in most cases generates optimal schedules.[4]

---

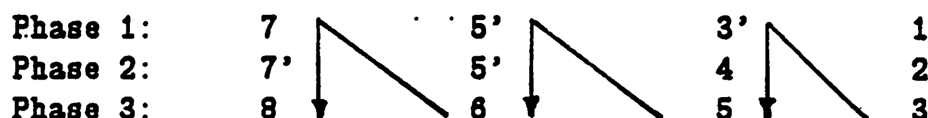
[4] The probability of not finding an optimal solution, when one exists, can be reduced significantly by employing a simple trick. Whenever the procedure fails to schedule the steps in B phases, it will try scheduling once more, but this time processing the steps in the reverse order. Only when that too fails, the number of phases is incremented.

```
s = 1, p = 0, No-Op-counter = 0, Test-Plan = empty list.

REPEAT WHILE s ≤ S
     p = p_mod B + 1
     IF No-Op-counter = B
      THEN increment B by 1 and RESTART the procedure.

     IF (s, x) exists in the CG, where x ∈ Phase p
      THEN append a No-Op to Test-Plan and to Phase p,
            increment No-Op-counter by 1.
      ELSE append step s to Test-Plan and to Phase p,
           s = s + 1,
           No-Op-counter = 0.
  END REPEAT
```

Example 6.5 Consider the 8-steps test plan of Example 6.4. Assume that procedure 1 starts with B = 3. The procedure succeeds in constructing an optimal schedule with 3 phases as shown below.



Example 6.6 Consider the 8 step test plan shown below which belongs to the multiple port kernel circuit illustrated in Figure 5.7 of the last chapter.

```
1-  R1(RNG),                R2(RNG).
2-  Bus1(R1), R3(Latch), R2(Hold).
3-  MUX1(R3), R5(Latch), Bus1(R2), R4(Latch).
4-  Bus2(R5), R6(Latch), MUX1(R4), R5(Latch).
5-  R6(Hold),               Bus2(R5), R7(Latch).
6-  Kernel, MUX2(Kernel.out1), R8(Latch), R6(Hold),R7(Hold).
7-  Bus1(R8), R9(SA), Kernel, MUX2(Kernel.out2), R8(Latch).
8-                          Bus1(R8), R10(SA).
```

The CG of this test plan is shown in Figure 6-5. Clearly, there is a clique of size 4 due to the sharing of Bus1 by steps 2, 3, 7, and 8. Using Procedure 1, an execution schedule with D=4 is found as shown below. Only one No-Op (after step 6) has been added to the test plan. Note that 6' does not conflict with step 3.

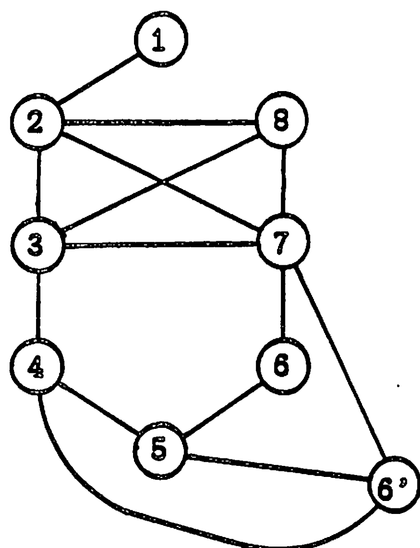
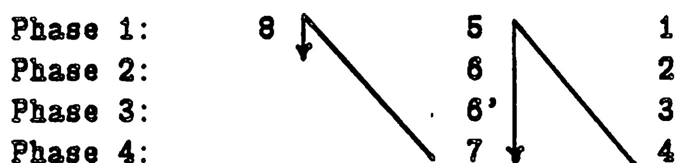


Figure 6-5: The CG of Example 6.6 (plus node 6')

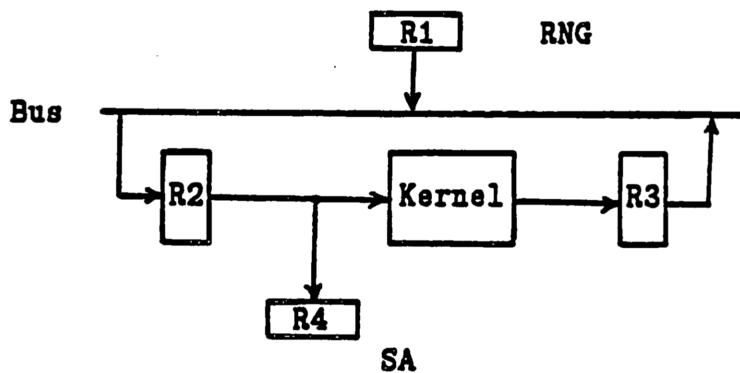


6.5. Conflicts Caused By No-Ops

A No-Op step inserted in a test plan calls for holding the state of one or more registers one clock period. Such a hold mode can create a conflict with a step that calls for changing the state of a register. To illustrate this concept, consider the following example.

Example 6.7: Consider the circuit shown in Figure 6-6a. Its BILBO test plan and CG are shown in Figure 6-6b and 6-6c, respectively.

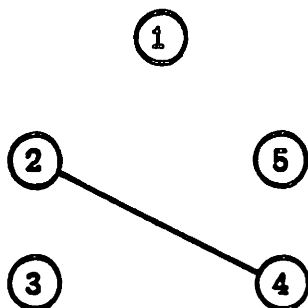
Without adding No-Op steps, D=2 is not feasible due to the conflict between steps 2 and 4. Clearly, both steps write data into R2. If a No-Op step is inserted after step 2, the new test plan cannot support D=2 because step 4 will be scheduled in parallel with 2' causing a conflict as shown below.



(a)

```
1- R1(RNG).
2- Bus(select R1), R2(Latch).
3- Kernel( - ), R3(Latch).
4- Bus(select R3), R2(Latch).
5- R4(SA).
```

(b)



(c)

Figure 6-6: An example with a potential for conflicting No-Ops.

```
Phase 1:     4     2'     1     <----------Conflict
Phase 2:     5     3      2
```

That is, step 4 will latch data in R2 which is supposed to hold its state as specified in 2'.

6.5.1. Modifying The Conflict Graph

Definition 6.11: A conflict involving two steps which write data into the same register is referred to as a **storage conflict.**

Clearly, the conflict graph has to be modified to reflect the extra conflicts caused by No-Ops when storage conflicts are present.

Definition 6.12: Given the conflict graph CG(N, C) of a test plan and assuming that there is at least one storage conflict in that CG, the **extended conflict graph** ECG($N \cup N'$, $C \cup C'$) can be formed, where
$N' = \{ i', j' \mid (i, j)$ is a storage conflict $\in C\}$, and
$C' = \{ (i, j'), (j, i') \mid (i, j)$ is a storage conflict $\in C\}$.

A modified version of Procedure 1 can be used to generate solutions for cases with conflicting No-Ops. Before adding a No-Op to one phase, the procedure refers to the ECG to check if that No-Op conflicts with any of the steps already scheduled in that phase. If it does, then the procedure backs up and restarts with more phases.

<u>Example</u> <u>6.8</u>: Again consider the test plan of the circuit of Example 6.7. The conflict (2, 4) is a storage conflict. The ECG is shown in Figure 6-7. Using the modified version of Procedure 1, a new test plan has been found supporting D=2 that involves adding a No-Op after step 3, as shown below. Note that since a No-Op after step 3 does not conflict with other steps, no node labeled 3' can be found in the ECG.

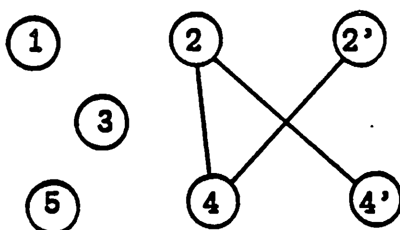


Figure 6-7: The Extended CG (ECG) of Example 6.8.

Theorem 6.10 does not hold for test plans with storage conflicts which arise when a register is used in an I-path that transfers tests to the kernel, and also in an I-path that transfers responses of the kernel. Such a case is unusual. Thus it is fair to claim that for all practical test plans one can always reach the lower bound given by Theorem 6.8.

6.6. Test Plans for Scan-Type TDMs

Test schemas for scan-type TDMs (like LSSD and scan path) have unique characteristics. The serial shifting of data in scan-type TDMs limits the amount of possible overlapping.

Consider the template of the scan-path TDM shown in Figure 6-8. An S/P I-path is required between the scan-in pin of the circuit and the input port of the kernel. This S/P I-path consists of an S/S I-path from the scan-in pin to a shift register that has an S/P I-mode, and then a P/P I-path from that shift register to the kernel input port. A P/S I-path is required between the kernel output port and the scan-out pin of the circuit as shown in Figure 6-8.

Assume that the kernel has n inputs and m outputs. Any scan-type test plan for that kernel will contain n steps in which the input shift register is serially loaded with a test vector. Hence, there will always be a clique of size n in the CG of that test plan. Similarly, there will also be another clique of size m corresponding to the activation plan of the P/S I-mode of the output shift register.

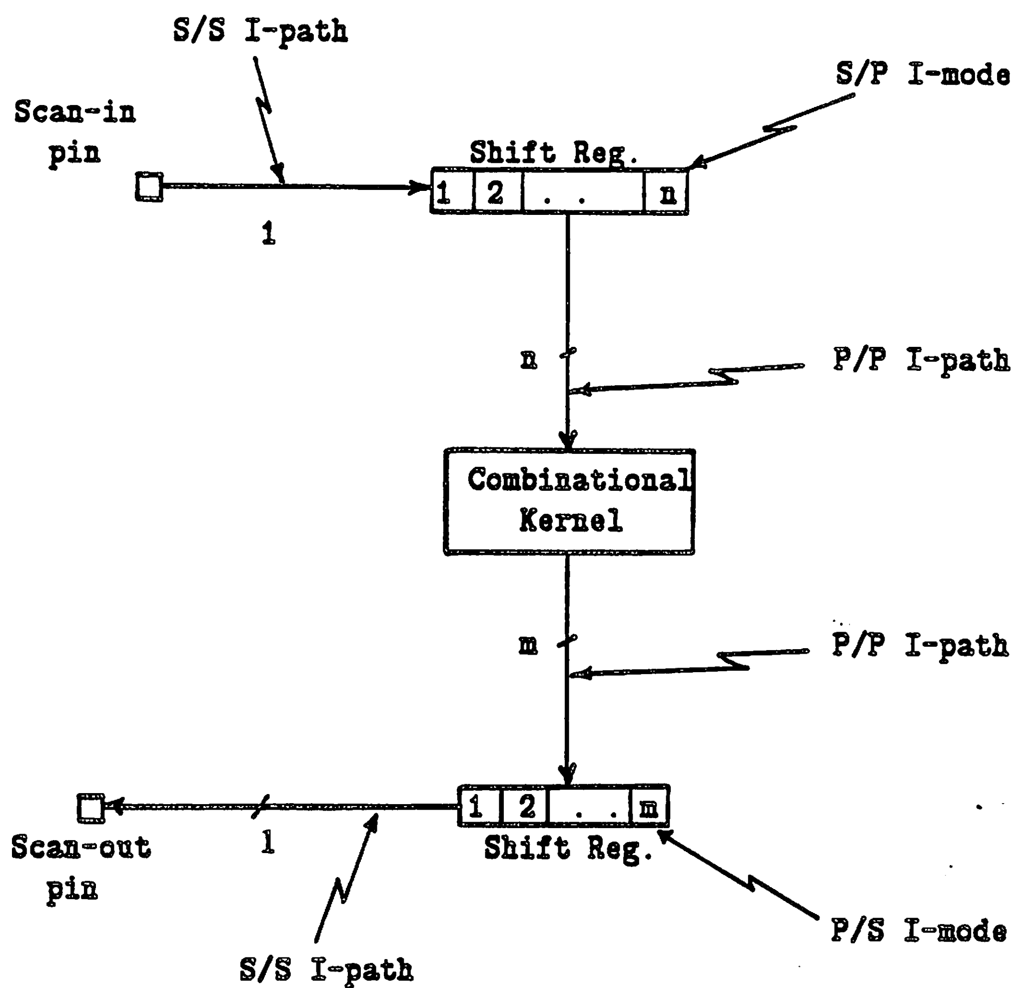


Figure 6-8: The structural template of the Scan Path TDM.

Theorem 6.7 implies that X = Max(n, m) is a lower bound on the value of D of any feasible execution schedule.

In most practical scan-type TDM embeddings the number of steps in the test plan corresponding to the parallel I-paths between the shift registers and the kernel is less than X. In all such cases it is easy to show that it is always possible to construct an execution schedule with D = X. Moreover, even if the extra steps due to the parallel I-paths is larger than X, but no cliques of size X or larger are generated by these steps (it is very unlikely otherwise), then again a schedule with D = X is always feasible. Hence, it is fair to claim that for scan-type test plans, test time is approximately T $\times$ X, where T is the number of test vectors in the test set.

6.7. Scheduling Test Plans of Multiple Kernels

In the previous sections we described how to construct an optimal test schedules for a single kernel. In this section, the problem of multiple kernels is considered. Kime and Saluja [Kime 82] considered the same problem. In their representation of the problem, every kernel is associated with a set of structures (called the resource set), which consists of the kernel itself and the structures used for test generation and for response evaluation. The problem then becomes that of scheduling the different kernel tests such that the overall test time is minimal. In such a schedule two tests can be scheduled in parallel only if their resource sets are disjoint. In the case where all tests have equal length, the problem is equivalent to the classical covering problem used in logic minimization. They also considered the case when tests have different lengths. No pipelining was considered in optimizing tests for single or multiple kernels.

Their work can be applied directly to tests generated using our techniques. Moreover, two techniques which can further minimize the total test time are described in the next two subsections.

6.7.1. Combining Test Schedules of Different Kernels

Consider two kernels K1 and K2, and assume that an optimal execution schedule was constructed for each kernel. Moreover assume that the initiation delay D of both schedules is the same. Let Px(j) denote phase j of the schedule of Kx. We can form a conflict graph to represent conflicts between phases of the two schedules.

Theorem 6.13: The test schedules of two kernels that have the same initiation delay D can be executed in parallel using a schedule with initiation delay D if there exist a positive integer $k < D$ such that P1(i) does not conflict with P2($(i+k)_{\text{mod } D} + 1$), for all i=1, 2, .. D.

Proof: A combined test schedule with D phases can be constructed as follows: Phase i of the new schedule consists of P1(i) and P2($(i+k)_{\text{mod } D} + 1$), for all i=1, 2, .. D. Clearly, the schedule is feasible and both kernels will get tested by the repeated execution of the D phases of the new schedule.

□

Note that even though the phases of two test schedules may have conflicts, it is quite possible to find a k that satisfies the above theorem, and hence execute the two schedules in parallel. Clearly, this was impossible to do using the problem formulation of [Kime 82].

6.7.2. Combining Test Plans of Multiple Kernels

The technique described in the previous subsection can only be applied to schedules with the same initiation delay. In cases where schedules have different delays, or when it is not possible to combine schedules because of conflicts, one solution is to combine the test plans of the different kernels. The same procedure used in Section 5.2.2.1 for combining activation plans of different I-paths can be used here as well. Once a combined test plan for the

two kernels is produced, scheduling optimization techniques which were described earlier can be used to generate an optimal test schedule.

Again notice that this approach can be used to schedule tests for kernels, with non-disjoint resource sets, in parallel.

Cleary, the two techniques described above for producing combined test schedules can be extended naturally to handle the n kernels case, for $n \geq 2$.

6.8. Remarks About Pipeline Optimization

As mentioned previously, the problem of generating optimal execution schedules for test plans is similar in concept to that of optimizing static pipelines. Using pipelines terminology [Kogge 81], a reservation table can be used to model the hardware utilization of a test plan. Every row in the reservation table corresponds to a circuit structures (called stages), while every column corresponds to a step in the plan. A mark in row i and column j indicates that structure i is used during step j. Davidson [Davidson 71] developed a procedure for constructing what is known as the modified state diagram of a reservation table. The diagram can then be traversed to find its cycles. Every cycle conveys a feasible value or a set of alternating values for D to assume. For example, a (2, 4, 5) cycle indicates that the delay between two consecutive iterations follows the sequence 2, 4, 5, 2, 4, 5, ..., etc.

Even though reservation tables are widely accepted for optimizing pipelines, they do not model conflicts explicitly as conflict graphs do. To determine if two steps i and j conflict using a reservation table, one has to check if columns i and j have marks in the same row. On the other hand, the explicit representation of conflicts in the CG model has led to the derivation of lower bounds on D which are stronger than the one given by Shar [Shar 72], namely, the maximum number of marks in one row of a reservation table.

Moreover, the theorem presented in Section 4.3 for finding feasible constant schedule cycles is much simpler to program and requires much less computation than the state diagram method of Davidson [Davidson 71]. However, our approach cannot generate alternating value cycles. Such a disadvantage is not very severe, because such schedules require complex controlling mechanisms which are not desirable in the context of built-in test.

Patel and Davidson [Patel 76] described a technique for inserting delays in pipelines to increase their throughput. The concept appears similar to the one of adding No-Op steps, yet there is a fundamental difference. To illustrate this, consider the reservation table shown in Figure 6-9a. A delay, denoted by d, is inserted in the second row of the table as shown in Figure 6-9b.

stage \ time	1	2	3	4
1		x	x	
2	x		x	
3	x	x		
4				x

(a)

stage \ time	1	2	3	4
1		x	x	
2	x		(d)	x
3	x	x		
4				x

(b)

Figure 6-9: Inserting a delay 'd' in a reservation table.

The new table supports D=2 while the first one does not. Note that the inserted delay has changed the structure of steps 3 and 4. Clearly, in the context of test plans such a transformation is not acceptable, without modifying the design, due to data dependencies. Recall that a No-Op step does not reorganize the actions performed in the steps of a test plan, but it introduces a step during which the data is held unchanged in some registers. Therefore, delay insertion techniques [Patel 76] are not generally applicable to test plans. Moreover, inserting delays in pipes often require the introduction of new registers. Such cost can not be justified when attempting to optimize test

plan schedules, and it may have some adverse effects on the circuit normal operation of the circuit. It is interesting to note that the use of special registers to implement delays alleviates the potential of creating conflicts as a result of inserting delays.

Thus, in summary, it is clear that even though one can apply classical pipeline optimization techniques to the problem of optimizing test plans, the approach presented here is more practical and effective and has led to many useful new results.

6.9. Summary

In this chapter a theory of test plan execution overlap was developed. First, the new concept of a test schedule was introduced, and a model was developed for describing the conflicts that may arise due to the parallelism in these schedules. Lower bounds on the time required to execute a test plan were derived, and algorithms for constructing optimal test schedules were presented. These algorithms employ schemes for inserting No-Op steps in a test plan to avoid scheduling conflicts. Cases involving No-Op steps which might create conflicts were analyzed and solutions for these cases were presented. The problem of constructing test schedules for multiple kernels was considered and two solutions were presented. Finally, we compared our techniques to classical pipeline optimization techniques and illustrated the superiority of our approach.

Chapter 7
A Global Strategy for TDES

7.1 Introduction

In Chapter 4 we described how to partition the CUC into a number of kernels. Given one of these kernels, we described in Chapter 5 how it can be made testable by embedding a TDM into the kernel's subgraph. Measures for evaluating various embedding solutions for a given kernel and a scheme for selecting the best solution based on these measures have also been described.

Based on this knowledge, the problem of designing a testable VLSI chip is formally defined as that of instantiating a TDM embedding for every kernel in the CUC. In practice the task of designing a testable VLSI chip (like any other design activity) is usually constrained by a number of constraints such as the available silicon area, an upper bound on test time, and the clock rate required. The same measures used to evaluate an embedding can also be used to evaluate a modified design. The values of these measures is computed from the measures of the instantiated embeddings. For example, the area overhead measure associated with a design is the sum of the area overhead of all the instantiated embeddings. A pessimistic test time measure equals the sum of the test time measure of all instantiated embeddings. However, it is quite possible to use more elaborate evaluation schemes to calculate more accurate values for the measures of interest. For example, by carrying out placement, routing, and timing analysis for both the initial design and the modified one, the exact area overhead, performance degradation, and extra I/O pins can be computed. One can also apply the techniques described in [Kime 82] or the

ones described in Section 6.7 to calculate a more accurate value of the total test time required. The values of these measures can then be compared to the design constraints to determine whether the design is acceptable or not.

The problem of designing a testable circuit can be viewed as that of searching a large space of embeddings to select an embedding for every kernel. Assume there are K kernels in the CUC, and assume that for each kernel there are, on the average, e feasible embeddings to choose from. The number of possible testable designs is e^K. Thus, it is clear that for any practical size circuit, trying to explore all possible designs is not feasible. Hence, a global scheme is required to control and direct the embedding process towards a testable design which satisfies all the constraints.

In the next section we describe a simple heuristic schema for globally controlling the operation of TDES.

7.2. Global Control Schema

A high-level flowchart illustrating a heuristic global control schema which can be used by TDES is shown in Figure 7-1. There are various provisions in this schema which aim at reducing the search space and in directing the system towards an acceptable testable design. It should be noted that this heuristic schema is not the only one possible. Many others can be devised either to execute faster or to enhance the quality of the final design generated. The following scenario illustrates the major steps in the proposed schema.

1. Partition the CUC and identify the kernels.

2. Since the kernels are to be processed individually, impose an ordering on the kernels to specify the order by which they will be processed. The ordered kernels are placed in a list called the k-list.

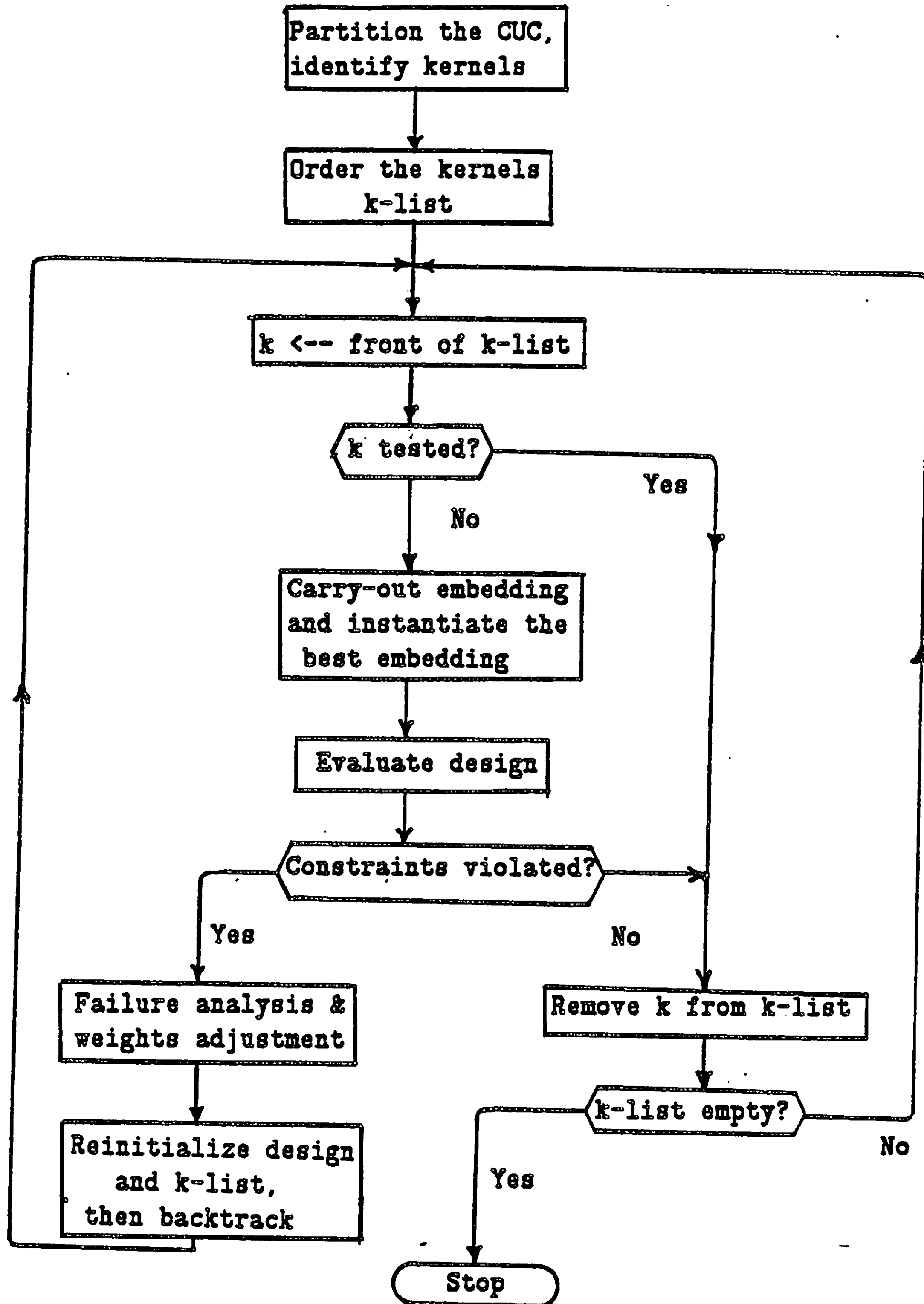


Figure 7-1: A flowchart of a global control schema for TDES.

3. The kernel in the front of the k-list, k, is selected for processing. If k is tested for free while testing kernels which have been processed already, then go to step 5. Otherwise, generate all the feasible embeddings for k using all applicable TDMs. Evaluate the embeddings and select the best one using the evaluation scheme described in Chapter 5. Instantiate the selected embedding by carrying out the necessary design modifications.

4. Evaluate the current state of the design with respect to the specified constraints. If none of the constraints are violated then go to the next step. Otherwise, if constraint i is violated by x%, then increase the weights of the embedding measures related to constraint i by x%. Erase all the modifications made to the design and restore the initial value of the k-list and restart the embedding selection process again by **backtracking** to step 3.

5. Remove k from the k-list. If the k-list is empty then this indicates that a "feasible" testable design has been found and we stop, otherwise go to step 3.

□

There are two interesting subtasks in the above schema.

1. Ordering the kernels.

2. Determining which kernels get tested for free.

These two subtasks are discussed in more detail in the next two sections.

7.3. Ordering the Kernels

An ordering is imposed on the kernels of the CUC indicating the order in which they will be processed. Several factors motivated the need for such ordering:

1. Some kernels, such as registers, MUXs, and busses may be used in testing other kernels, and hence as a side effect, may be tested repeatedly while other kernels are tested. Hence, it is more logical to place such kernels at the end of the processing list hoping that by the time they arrive at the front the processing list they are either fully or partially tested. In the next section we examine the task of determining whether a kernel is tested for free or not.

2. Some kernels may be harder to test than others, because of their type, size or location. These kernels should be processed early, because they are the ones which will contribute significantly to the final result. In doing so, TDES quickly discovers if one of the constraints is violated hence avoiding excessive backtracking.

3. Kernels with limited number of applicable TDMs should be processed before ones with many applicable TDMs. The rationale behind this is to reduce the probability of making a wrong decision early . Clearly, when the number of choices is small, the chance of making the right decision is fairly good. For example, assume there are 5 feasible embeddings for kernel k1 and 20 feasible embeddings for kernel k2. If k2 is processed first, TDES has to select an embedding from 20 possible embeddings and hence there is a reasonable chance that in despite of the global measures used in the evaluation process a non optimal decision is made. On the other hand, by processing k1 first, the chance of making a bad decision is

reduced. Moreover, knowing how k1 will be tested may be very helpful in deciding on which embedding to select for k2.

4. Kernels with no driving or receiving registers in their subgraph require MUX-embeddings. As was illustrated earlier in Chapter 5 there are usually many feasible MUX-embeddings. Deciding which one to select in the early stages of the embedding process is not an easy task. However, such a task becomes much simpler as soon as BIT structures are created in the design as a part of other kernels embeddings. Thus, it is better to process kernels which require MUX-embeddings after processing other kernels.

The above heuristic rules may not necessarily produce the best possible ordering. One can always devise situations in which one or more of these rules do not apply very well.

Based on the above intuitive rules, the next procedure can be used to order the kernels in a list, called the k-list.

1. Registers, busses and MUXs are placed at the end of the k-list in that order (MUXs are last).

2. The remaining kernels are partitioned into two groups: (a) kernels which do not necessarily require MUX-embeddings; and (b) kernels which require MUX-embeddings.

3. The kernels in group "a" are placed first in the k-list followed by those of group "b." The kernels in either group are then ordered according to the following three rules

    - in an increasing order according to the number of applicable TDMs (a kernel that can be tested by 3 TDMs is placed after a kernel that can be tested with only 2), and within that

    - in a decreasing order according to the kernel's size (silicon area), and within that

- in a random order.

7.4. Kernels Tested for Free

In Chapter 5 we described how a structure with an I-mode can be used to transfer input or output test data as a part of a test plan of a kernel. Lemmas 5.11 and 5.12 show that faults affecting data lines along an I-path are indistinguishable. However using a structure as a part of a kernel's test plan is not a guarantee that this structure is fully tested. Next we analyze the situation for MUXs , busses and registers.

7.4.1. Testing Multiplexers and Busses

Busses functionally operate exactly like MUXs, hence in this section we consider only MUXs, but all our arguments can be applied to busses as well. The function of a MUX is quite simple. In general a MUX has m input sources each n-bit wide, and one output port n-bit wide. In addition a MUX has a number of control signals. Depending on the values of the control signals only one of the input sources is allowed to pass its data to the output of the MUX. Multiplexers are often tested functionally by exercising their various modes of operation and verifying their behavior. One can partition the testing of multiplexers into two phases.

- First verify the integrity of the input sources as follows.

```
Repeat for i = 1, 2, ..., m
  Select input source i and apply enough test patterns
  to  input source i in order to verify the integrity
  of the input data lines.
```

- Secondly, verify the integrity of the control lines as follows.

```
Repeat for i = 1, 2, ..., m
  Select input source i and apply one or more test
  patterns to input source i while applying different
  patterns to all the other input sources, such that
```

```
the value of bit j of input source i is different
from the value of bit j of input source x during at
least one test pattern, for j = 1, ..., n and x ≠ i.
```

It should be noted that it is often possible to verify the integrity of the control lines while verifying the integrity of the data lines. However, we purposely separated the two phases to help identify what get tested for free when a MUX is used as a part of an I-path.

Verifying the integrity of n data lines depends on the fault model of interest. For example, if we limit ourselves to the stuck-at fault model then any two test patterns which are the complement of each other are adequate to cover all single and multiple stuck-at faults affecting the n data lines. However, if shorts between adjacent lines are also considered then the two test patterns 1010... and 0101... are adequate. If shorts can occur between any lines, then a marching 1 pattern or a marching 0 is adequate. In the worst case one can use all 2^n patterns.

In a TDM-embedding of a kernel k, it is fair to assume that the integrity of the kernel's input and output lines are verified with respect to the fault model under consideration. Hence, using Lemma 5.12, it follows that the integrity of the input source of a MUX used to transfer test data to k or to transfer k's responses is also verified. However, since no information is usually available from the test plan of k regarding the values of the other input sources of the MUX, one cannot claim, in general, that the MUX is selecting the proper input source. To illustrate this point, consider a 2 to 1 MUX which is used in a test plan to transfer data via its first input source. Assume that the same data also appears at the second input port of the MUX. Clearly, a faulty control signal which always selects the second input source of the MUX will not be detected while executing the test plan.

In most practical designs the above situation is very unlikely. However,

to guard against these unlikely situations, the unused input sources of a MUX should be held unchanged while using a MUX as a part of a test plan, or even simpler, it is enough to show that the data values on the unused input sources are different from the data being transferred via the MUX during at least one iteration of a test plan. Again, in most practical cases, at least one of the above two requirements is satisfied automatically, or can be satisfied by making minor changes to the test plan of k, such as holding the state of registers driving the unused input sources of the MUX.

Every time a MUX is used in an I-mode as a part of an instantiated embedding, that I-mode is recorded. If by the time a MUX arrives at the front of the k-list it is found that every one its I-modes has been exercised in one or more test plans, then the only remaining task is to make sure that the MUX is selecting input sources properly. As was illustrated earlier this may involve modifying the test plans of instantiated embeddings slightly to block any I-paths which might transfer test data to the wrong input source of the MUX.

On the other hand, if one of the MUX's I-modes was never exercised, then the MUX is treated like any other kernel by exploring feasible TDM embeddings. The only difference here is the fact that in addition to the TDMs which treats a MUX like any general combinational circuit, there are TDMs that test MUXs functionally, one I-mode at a time. An example of these TDMs is given in Appendix A. Note that the latter type of TDMs can take into consideration the fact that some of the MUX's I-modes have been tested already.

7.4.2. Testing Registers

Registers play a key role in all embeddings. Earlier we described how to modify registers to serve as BIT structures to test other kernels in the CUC. Clearly, there is no need to explicitly test a register in a mode of operation which has been added. Such a mode of operation, will be exercised for free as a part of the embedding in which it was created. For example, if a register is modified to become a signature analyzer as a part of an embedding for kernel k, then the signature analysis mode of operation is exercised for free while testing k.

Hence, only the original modes of operation of a register are explicitly considered by TDES. Consider a register R which is used in a mode of operation M to transfer test data A either to or from a kernel k. Typical values of M are "latch", "hold", and "shift." Consider the typical case where the volume of A is considerably larger than the size of R. Similar to the arguments which we used with MUXs and from Lemma 5.11, if kernel k is thoroughly tested then mode M of the register is tested for free.

Modes that are not exercised in any test plan have to be tested explicitly. Again specialized TDMs exist for testing the various modes of operation of registers. Two examples of these TDMs are in Appendix A, one for testing the shift mode and one for the count-up mode.

Chapter 8
A Prototype System

8.1. Introduction

A prototype of TDES has been implemented using a dialect of LISP called UCI-LISP [Meehan 79] and runs on a DEC-20. Since this prototype is the first version of TDES we will refer to it as TDES1. In this chapter we briefly describe the main features of TDES1. Documentation on TDES1 and a user's manual is provided elsewhere [Abadir 85e].

There are 3 different environments for TDES1.

1. Circuit description environment.

2. Testability knowledge capturing environment.

3. Testability synthesis environment.

The above environments will be discussed next.

8.2. Circuit Description Environment

TDES1 takes as an input a description of a flat graph model of the CUC. As described in Chapter 2, structures are modeled as nodes with various attribute value pairs attached as labels. Interconnections between structures are modeled as arcs. TDES1 requires two input files, one containing a description of the nodes, and another one for the arcs. As an example of how nodes are described, consider the following description of a register called RX.

```
(RX
     (TYPE REG)
     (FUNCTION (HOLD LATCH CLEAR))
     (SIZE 16)
     (IN-PORTS ((RX P1) 16) )
     (OUT-PORTS ((RX P2) 16) )
     (I-MODES (P1 P2))
     (FREEDOM 2)
     (I-DELAY 1)
     )
```

The meaning of most labels is self explanatory. A detailed description of the syntax of this input description language is in [Abadir 85e].

Arcs are described using a similar language. For example, an arc, A35, of size 16 bits from port P2 of node RX to port P4 of node C5 is described as follows.

```
(A35  (FROM (RX P2)) (TO (C5 P4)) (WIDTH 16) )
```

8.3. Testability Knowledge Capturing Environment

As described in Chapter 3, the major source of knowledge in TDES is the TDM frames. A simple description language was developed to describe TDM frames to TDES1. As an example, consider the following description of the BILBO TDM.

```
(BILBO
 (KERNEL (TYPE COMB) (STYLE (PLA ROM CGN RANDOM)) (SIZE N M))
 (DRIVER (TYPE REG) (FUNCTION (RNG CLEAR)) (SIZE >= N) )
         (RETENTIVE T) )
 (RECEIVER  (TYPE REG) (FUNCTION (SA CLEAR)) (SIZE >= M)
         (RETENTIVE T) )
 (TDM-SCHEMA (D-PLAN ( (DRIVER RNG :) ))
             (R-PLAN ( (RECEIVER SA :) )) )
 (MEASURES (SOFTWARE (SIGNATURE-EVALUATION FAULT-SIMULATION))
             (ATE 2)
             (ARE-OVERHEAD 100)
             (FAULT-COVERAGE 99.8)
             (PERF-DEGRADATION 2)
             (TEST-LENGTH RANDOM-ESTIMATOR-FUNCTION)
             (EXTRA-IO-SIGNALS 3)
             (TEST-TIME (TEST-LENGTH * D))
             )
 )
```

Note that the above description represents a simplified version of the BILBO TDM frame of Appendix A. TDM frames described in the above form are stored in a file and given to TDES1 as input. Currently, four TDM frames have been described in the above form, namely BILBO, scan path, exhaustive and check-sum. A testability expert who wishes to add more TDMs to the knowledge base of TDES1 has to describe the new TDM frames in the above form and then add it to the TDM file. When TDES1 is invoked, the TDM file will be read and the new TDMs will be used. Note that TDES1 does not have permanent storage capability. Thus, every time it is invoked, the TDM file is first loaded to construct the knowledge base. For a full description of the syntax of the input description language of TDM frames refer to [Abadir 85e].

Another form of testability knowledge which can be supplied by a testability expert are the parameters used to control the following tasks in TDES1, namely

1. kernels clustering,

2. computing area overhead estimates,
3. computing test length estimates, and
4. combining embedding measures into scores.

The values of these parameters greatly affect the operation of TDES1. A testability expert can easily update the values of these parameters via a user-friendly commands-menu as will be described later in this chapter.

8.4. The Testability Synthesis Environment

This is the operational environment of TDES1. In this environment TDES1 can be used to explore various ways of making a design easily testable. When this environment is invoked, TDES1 reads a description of the CUC from external files, and also reads a description of the TDM frames to be used during that run from another external file. TDES1 identifies the kernels in the CUC using first the flat-design partitioning scheme, and then using the bottom-up clustering scheme. The kernels are ordered in a decreasing order according to their size. Using the parallel I-path concept, a subgraph is formed for every kernel. Embedding TDMs into the CUC is done interactively with the user. A menu-driven user-friendly interface with TDES1 is provided. Using a number of display menus, the user can (1) examine different parts of the knowledge base, (2) examine and/or update the various parameters used by the partitioning and embedding processes, (3) explore all feasible embeddings for a kernel and examine the various evaluation measures associated with these embeddings, (4) instantiate an embedding into the CUC or erase the effects of a previously instantiated embedding, (5) examine the state of the design at any given time, and (6) save the state of a modified design for later processing.

In the next subsections we describe the various menus of TDES1 and the commands available in each menu.

8.4.1. The Top Level Menu

In the top level menu of TDES1 the user is given a choice between 4 different commands menus as shown in Figure 8-1. The top level menu also offers the user an option to save the state of the design and the knowledge base for later processing. From any secondary menu the user can always return back to the top level menu. The user can exits directly to the command level of LISP from any one of the menus.

```
TOP LEVEL MENU

1    THE CIRCUIT UNDER CONSIDERATION (CUC) MENU
2    THE TESTABLE DESIGN METHODOLOGIES (TDMs) MENU
3    THE SYSTEM PARAMETERS AND PRIORITIES MENU
4    THE EMBEDDINGS MENU
5    SAVE THE STATE OF THE KNOWLEDGE BASE IN THE DEFAULT FILE (DATA)
6    SAVE THE STATE OF THE KNOWLEDGE BASE IN A SPECIFIED FILE

ENTER YOUR SELECTION (E TO EXIT): *1
```

Figure 8-1: The top level menu of TDES1.

8.4.2. The CUC Menu

As shown in Figure 8-2, the CUC menu contains various commands for examining the circuit components and the results of the partitioning process. The following commands are available in the CUC menu.

```
THE CUC MENU

1    DISPLAY THE ATTRIBUTES AND CONNECTIONS OF A CIRCUIT NODE
2    DISPLAY THE LIST OF THE MAIN KERNELS TO BE TESTED IN THE CIRCUIT
3    DISPLAY THE ATTRIBUTES OF ONE OF THE KERNELS
4    DISPLAY THE SUBGRAPH OF A PARTICULAR KERNEL

0    GO BACK TO TOP LEVEL MENU

ENTER YOUR SELECTION (E TO EXIT): *
```

Figure 8-2: The CUC menu.

1. Display all the attributes and the arcs associated with one of the circuit nodes.

2. Display an ordered list of the main kernels in the CUC. Busses, registers, and MUXs are not considered main kernels. Kernels formed by clustering nodes are included in the kernel list.

3. Display all the attributes of a kernel. This command is very useful

```
THE TDMs MENU
1     DISPLAY THE NAMES OF ALL TDM FRAMES DEFINED
2     DISPLAY DETAILED INFORMATION ABOUT A PARTICULAR TDM

0     GO BACK TO TOP LEVEL MENU

ENTER YOUR SELECTION (E TO EXIT):*
```

Figure 8-3: The TDM menu.

b. the maximum number of nodes allowed in a cluster, and

c. the maximum number of inputs of various types of clusters.

Clustering is carried out initially based on a default value for these parameters. If the user modifies any of these parameters, clustering is carried out again and the new results are displayed. In the later case, a new kernel list is formed and displayed. Also the system forms the subgraphs for the newly created kernels.

2. Display the tabulated constants used to calculate the area overhead measure associated with register modifications.

3. Display the normalization weights of the various embedding measures. (See Section 5.4.1).

for kernels formed by clustering nodes. Thus, the user can examine the attributes of the newly formed cluster node and the various ports and arcs associated with it.

4. Display the subgraph of one of the kernels. For each input (output) port of a kernel, this command displays a list of the circuit nodes which have I-paths leading to (coming from) that port. The activation plans of the various I-paths are also displayed.

8.4.3. The TDM Menu

The TDM menu allows the user to examine the various TDMs in the knowledge base of TDES1. As shown in Figure 8-3, the following commands are offered in the TDM menu.

1. Display a list of the names of the TDM frames in the knowledge base.

2. Display detailed information about one of the TDM frames. This information covers the structural template of the TDM, the test schema, and the various TDM measures.

8.4.4. The Parameters Menu

In the parameters menu the user (or a testability expert) can examine and modify various system parameters. Some of these parameters are used to control the partitioning process, while the majority are used in controlling and evaluating the embeddings. As shown in Figure 8-4, the following commands are available in the parameters menu.

1. Display and/or update various parameters associated with the clustering process. These parameters specify
   a. whether sequential clusters are allowed or not,

```
THE PARAMETERS MENU

1   DISPLAY AND/OR UPDATE THE PARAMETERS ASSOCIATED WITH CLUSTERING

2   DISPLAY THE AREA ESTIMATE MEASURES USED IN CALCULATING THE COST OF
    MODIFYING REGISTERS

3   DISPLAY THE NORMALIZATION WEIGHTS OF THE EMBEDDING MEASURES

4   DISPLAY AND/OR UPDATE THE RELATIVE PRIORITIES ASSOCIATED WITH
    THE EMBEDDING MEASURES

5   DISPLAY AND/OR UPDATE THE FLAGS ASSOCIATED WITH THE CASES WHERE
    MUX-EMBEDDINGS ARE PERMITTED

6   DISPLAY AND/OR UPDATE THE TESTABILITY FACTORS ASSOCIATED WITH
    DIFFERENT TYPES OF KERNELS

7   DISPLAY AND/OR UPDATE THE DEGREE OF FAULT COVERAGE REQUIRED

ENTER YOUR SELECTION (0 FOR TOP LEVEL MENU) (E TO EXIT):*
```

Figure 8-4: The parameters menu.

4. Display and/or update the priority weights associated with the embedding measures. The default value for these weights is 1. The user can increase or reduce the weight of some measures according to the design goals and constraints. For example, if area constraints are very tight, the user can increase the weight associated with the area-overhead measure by a factor of 2 or more. In that case, the weight of the sharing potential measure should also be increased to increase the amount of sharing, and hence reduce the total area overhead. The weights of irrelevant measures can be set to zero to mask their effects during the evaluation process. The values of these weights can also be adjusted when design constraints are violated as a result of instantiating an embedding as was described in Chapter 7.

5. Display and/or update the parameters associated with MUX-embeddings. As described in chapter 5, there are three cases in which MUX-embeddings are potentially useful, namely (1) the no-match case, (2) the register feedback (sequential) case, and (3) the perfect match case. A flag is associated with each case to indicate its applicability. If the flag associated with case i is set to NIL, then TDES1 will not explore MUX-embeddings whenever case i is encountered. The default value of all three flags is T (not NIL).

6. Display and/or update the parameters associated with the test length estimator functions used by TDES1. There are two parameters associated with each of the following type of kernels: sequential, ROMs, PLAs, gate combinational logic, and random combinational logic. The first (second) parameter associated with kernels of type t is used to reflect the relative length of deterministically (randomly) generated test patterns for a kernel of that type. There is usually a factor of 5 or more between the random and deterministic parameters for any kernel. For example, the default values of these parameters are listed below.

Kernel type	Deterministic parameter	Random parameter
CGN	1	5
PLA	2	20
ROM	4	20
Random	3	15
Sequential	10	50

7. Display and/or update the overall degree of fault coverage desired. This degree is used in estimating the required test length associated with various embeddings. It also serves as a qualification condition for TDMs. If a TDM does not meet the required degree of fault coverage, then it is excluded from consideration during the embedding process.

8.4.5. The Embedding Menu

This is by far the most important menu in TDES1. As shown in Figure 8-5, there are 9 commands available in this menu.

```
THE EMBEDDING MENU

1   FIND ALL POSSIBLE  EMBEDDINGS FOR A KERNEL
2   DISPLAY THE IDs OF THE BEST EMBEDDINGS OF A KERNEL
3   DISPLAY ALL ATTRIBUTES OF AN EMBEDDING RECORD
4   DISPLAY ALL SOLUTIONS FOR MATCHING A TDM GRAPH TO A KERNEL
5   EXECUTE AN EMBEDDING
6   ERASE THE EFFECTS OF A KERNEL EMBEDDING
7   LIST THE CIRCUIT STRUCTURES THAT HAVE BEEN MADE TESTABLE
8   DISPLAY A KERNEL EMBEDDING THAT HAS BEEN EXECUTED
9   DISPLAY ALL THE MODIFICATIONS THAT HAVE BEEN MADE TO THE DESIGN
0   RETURN TO TOP LEVEL

ENTER YOUR SELECTION (E TO EXIT) :*
```

Figure 8-5: The embedding menu.

1. Generate all feasible TDM-embeddings for a particular kernel k. For each TDM applicable to k, TDES1 generates all feasible embeddings, constructs the test plan and the optimal execution schedule associated with each embedding, evaluates the various measures associated with each embedding, and calculates a score for each embedding based on the measures. TDES1 uses a two-phase scoring scheme as described in Chapter 5. In this scheme the best embedding(s) for each TDM is (are) first identified based on the CUC-related measures. The identified embeddings are then

evaluated using both the CUC and the TDM related measures to identify the best embedding for k. As described in Chapter 5, some of the embedding measures, such as test length and area overhead, are calculated using simple heuristic (or possibly ad-hoc) estimators. However, other measures, such as the test schedule initiation delay D, are calculated using more elaborate algorithmic techniques. The results obtained as a result of executing this command are displayed in a brief tabular form. Other commands are available to examine a particular embedding in detail.

2. Display the best TDM-embeddings associated with a particular kernel k. Clearly, this command can only be executed after the previous one. Its main purpose is to remind the user which embeddings were identified by TDES1 as being the best for kernel k.

3. Display in detail all the data associated with a particular embedding. This data include (1) the matching data of the embedding, (2) the circuit modifications required, (3) the test plan, (4) the optimal execution schedule, (5) the values of all the evaluation measures, and (6) the scores computed by TDES1 for this embedding.

4. Display the matching data associated with the embedding of a TDM into a kernel subgraph. This command displays in a concise manner all the possible ways of matching template components of a TDM using structures in a kernel subgraph. This matching data is generated by TDES1 in an intermediate step while generating all the embeddings for one TDM for a given kernel.

5. Instantiate one of the embeddings. TDES1 will carry out all the necessary modifications in the design, and will mark the circuit structures and the modes of operation that get tested by this embedding.

6. Erase a previously instantiated embedding. Using this command, the user can reverse a decision which he made earlier. TDES1 carries out a dependency analysis to determine if any subsequent instantiated embedding utilizes circuit features which will be removed. If this is the case, the user is given the option between either erasing all the dependent embeddings, or canceling the process.

7. Display a list of the circuit structures which have been made testable as a result of instantiating TDM-embeddings. In addition to the kernels of the instantiated embeddings, TDES1 also keeps track of the modes of operation of other structures which are tested for free. If all modes of operation of a structure are tested, then this structure is added to the list of covered structures. TDES1 also displays the tested modes of operation of partially covered structures.

8. Display detailed data of the instantiated embedding of a given kernel. The data displayed by this command are the same as those displayed in command 3 of this menu.

9. Display the modifications that has been made to the design. These modifications include register modifications, and the created multiplexers with their interconnections. The total area overhead measure associated with these modifications is also displayed.

Using the above commands the user can explore various ways of making a design testable. The user can leave the embedding menu at any point and go to another menu to examine the state of the knowledge base, or modify some parameters. Currently TDES1 does not attempt to evaluate a design with respect to design constraints. The user has to decide whether a testable design is acceptable or not with respect to the design constraints. If the design is not acceptable, the user can change the embedding parameters in a manner which reflects the design constraints better. Reevaluation of the TDM-embeddings can then be carried out by TDES1 in view of the new parameters resulting in new scores and ultimately leading to a new testable design. Once the user finds a satisfiable testable design, he can save the state of the design and the knowledge base using commands in the top-level menu.

8.5. An Example

We conducted several case studies on manually generated designs and real designs as well. Although TDES1 has a very small set of TDMs and lacks many of the features which the final version of TDES should have, the results obtained were very encouraging and illustrated the validity of our concepts and methodology.

A simple example using TDES1 is illustrated in Figure 8-6. The original circuit design is shown in Figure 8-6a. All the registers in the CUC can clear, hold and latch parallel data, except R4 which is also a shift register. TDES1 identified three major kernels in the CUC, namely the PLA, ROM and C. Default values of the system parameters were used [Abadir 85e]. The priority weights of the embedding measures were all equal. No feasible clusters were found in the CUC due to the upper bound (32) placed on the number of inputs of a combinational cluster. A 99% fault coverage requirement was specified.

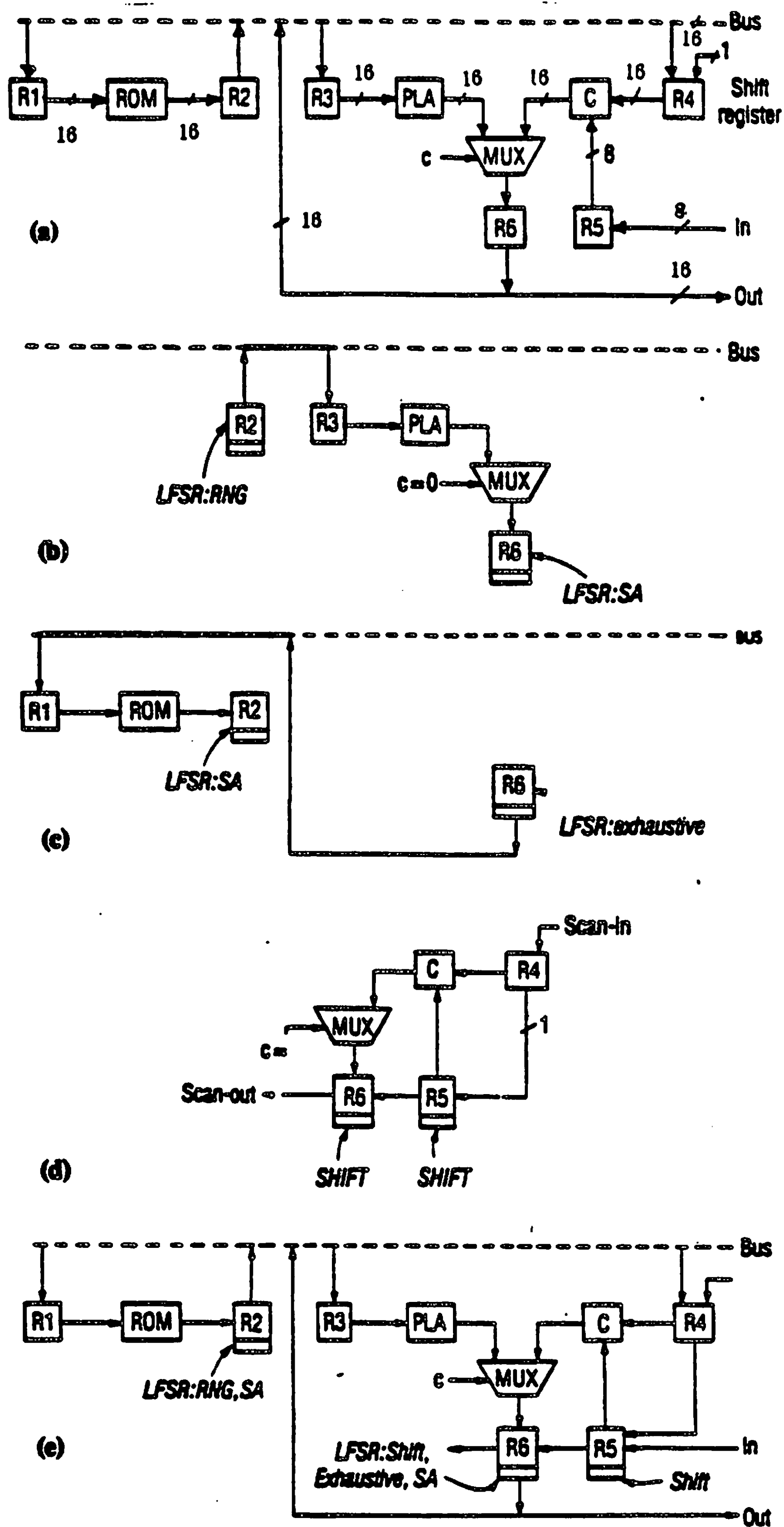


Figure 8-6: (a) The original CUC (b) BILBO embedding for the PLA (c) exhaustive embedding for the ROM (d) scan path embedding for the ROM (e) the modified circuit.

The PLA was considered first. As an example of the range of possibilities that are explored by TDES1, Figure 8-7 (as produced by TDES1) identifies the various embeddings for the PLA. There are three sets of embeddings, one for each applicable TDM. The scores of those embeddings are also shown with the best one(s) among each set marked with a "*." As shown in the last table of of Figure 8-7, two BILBO embeddings, I-RECORD9 and I-RECORD10, were identified as being the best overall embeddings for the PLA. I-RECORD10 was selected because it has a higher register-sharing measure. As shown in Figure 8-6b, to instantiate that embedding, register R2 was modified to become an LFSR with an RNG mode in order to generate pseudorandom test vectors, and R6 was modified to become an LFSR with an SA mode of operation to capture the PLA signature. Detailed information about the instantiated embedding as produced by TDES1 is shown in Figure 8-8. The test plan associated with I-RECORD10 is also shown in Figure 8-8. TDES1 reported that an optimal schedule with an initiation delay of 1 is feasible for that test

The ROM kernel was considered second. An exhaustive embedding, I-RECORD30, was selected by TDES1 as the best embedding for the ROM. A description of this embedding as produced by TDES1 is shown in Figure 8-9. As shown in Figure 8-6c, R6 is used as an exhaustive LFSR to address all the ROM contents, while R2 operates as a signature analyzer LFSR to compact the ROM responses into a unique signature. Note how the ROM embedding takes advantage of the modifications already made to R2 and R6 during the PLA embedding. As a result the ROM embedding has a very low area overhead. It is important to stress here that this was not an accident, since the global measures used by TDES1 to evaluate embeddings takes into consideration potential for sharing BIT structures, hence leading to good global solutions.

Finally, a scan path embedding, I-RECORD46, was instantiated for the combinational kernel C as shown in Figure 8-6d. A description of this

ALL EMBEDDINGS FOR KERNEL PLA USING THE EXHAUSTIVE TDM

EXTRA I/O SIGNALS 3
SOFTWARE REQUIREMENTS: (SIGNATURE-EVALUATION)
ATE COMPLEXITY 2

*** THE EMBEDDINGS WITH THE BEST SCORE ARE MARKED WITH AN *

I-REC#	DRIVERS ((PLA P1))	MOD'S	RECEIVERS ((PLA P2))	MOD'S	D, S	SCORE
I-RECORD1	(R3)	((LFSR))	(R6)	((SA))	1, 2	842
I-RECORD2	(R3)	((LFSR))	(R1)	((SA))	1, 3	813
I-RECORD3	(R3)	((LFSR))	(R4)	((SA))	1, 3	793*
I-RECORD4	(R2)	((LFSR))	(R6)	((SA))	1, 3	793*
I-RECORD5	(R2)	((LFSR))	(R1)	((SA))	2, 4	854
I-RECORD6	(R2)	((LFSR))	(R4)	((SA))	2, 4	834

ALL EMBEDDINGS FOR KERNEL PLA USING THE BILBO TDM

EXTRA I/O SIGNALS 3
SOFTWARE REQUIREMENTS: (SIGNATURE-EVALUATION FAULT-SIMULATION)
ATE COMPLEXITY: 2

*** THE EMBEDDINGS WITH THE BEST SCORE ARE MARKED WITH AN *

I-REC#	DRIVERS ((PLA P1))	MOD'S	RECEIVERS ((PLA P2))	MOD'S	D, S	SCORE
I-RECORD7	(R3)	((RNG))	(R6)	((SA))	1, 2	842
I-RECORD8	(R3)	((RNG))	(R1)	((SA))	1, 3	813
I-RECORD9	(R3)	((RNG))	(R4)	((SA))	1, 3	793*
I-RECORD10	(R2)	((RNG))	(R6)	((SA))	1, 3	793*
I-RECORD11	(R2)	((RNG))	(R1)	((SA))	2, 4	854
I-RECORD12	(R2)	((RNG))	(R4)	((SA))	2, 4	834

Figure 8-7: Feasible embeddings for the PLA as produced by TDES1.

ALL EMBEDDINGS FOR KERNEL PLA USING THE SCAN-PATH TDM

EXTRA I/O SIGNALS 4
SOFTWARE REQUIREMENTS: (TEST-GENERATION FAULT-SIMULATION RESPONSE-EVALUATION)
ATE COMPLEXITY 10

*** THE EMBEDDINGS WITH THE BEST SCORE ARE MARKED WITH AN *

I-REC#	DRIVERS ((PLA P1))	MOD'S	RECEIVERS ((PLA P2))	MOD'S	D, S	SCORE
I-RECORD13	(R3)	((SHIFT))	(R4)	(NIL)	17, 34	2204
I-RECORD14	(R3)	((SHIFT))	(R6)	((SHIFT))	17, 33	2343
I-RECORD15	(R3)	((SHIFT))	(R1)	((SHIFT))	17, 34	2314
I-RECORD16	(R3)	((SHIFT))	(R3)	(NIL)	17, 34	2224
I-RECORD17	(R2)	((SHIFT))	(R4)	(NIL)	17, 35	2155*
I-RECORD18	(R2)	((SHIFT))	(R6)	((SHIFT))	17, 34	2294
I-RECORD19	(R2)	((SHIFT))	(R1)	((SHIFT))	17, 35	2265
I-RECORD20	(R6)	((SHIFT))	(R6)	(NIL)	17, 34	2194

BEST EMBEDDINGS FOR KERNEL PLA

AREA COST	TEST TIME	PERF. DEGRD	FOR FREE	REG. SHAR	PINS	SOFT REQ.	ATE REQ.	SOL. HOMG.	TDM BIAS	SCORE
I-RECORD3		EXHAUSTIVE:								
55	64000	2	6	7	3	1	2	0	1	4750
I-RECORD4		EXHAUSTIVE:								
60	64000	2	6	9	3	1	2	0	1	4750
I-RECORD9		BILBO:								
55	620	2	6	7	3	2	2	0	1	808*
I-RECORD10		BILBO:								
60	620	2	6	9	3	2	2	0	1	808*
I-RECORD17		SCAN-PATH:								
15	850	1	9	10	4	3	10	0	1	1063

THE BEST EMBEDDING IS (ARE) (I-RECORD9 I-RECORD10)

Figure 8-7: (continued) Feasible embeddings for the PLA.

EMBEDDING RECORD I-RECORD10

KERNEL	PLA
TDM	BILBO
TEST LENGTH FOR 99% FAULT COVERAGE IS:	620.0
UPPER BOUND ON FAULT COVERAGE	100
EXTRA I/O PINS:	2

EMBEDDING DATA:

DRIVERS	(R2)
DRIVER-MODIFICATIONS	((RNG))
RECEIVERS	(R6)
RECEIVER-MODIFICATIONS	((SA))
THE MODIFICATIONS COST	60
POTENTIAL FOR SHARING MEASURE	9

TEST PLAN :
1: (R2 RNG :)
2: (BUS R2 +) (R3 LATCH :)
3: (PLA NIL +) (MUX PLA +) (R6 SA :)

SCHEDULE INITIATION DELAY D = 1 OPTIMAL

6 MODES ARE TESTED FOR FREE IN THIS EMBEDDING

BILBO EMBEDDING SCORE = 793
GLOBAL EMBEDDING SCORE = 808

MEASURES SUMMARY

AREA COST	TEST TIME	PERF. DEGRD	FOR FREE	REG. SHAR	I/O SIG.	SOFT REQ.	ATE REQ.	SOL. HOMG.	TDM BIAS	SCORE
60	620	2	6	9	3	2	2	0	1	808

INSTANTIATED?? T

Figure 8-8: Detailed data of the the best embedding for the PLA.

EMBEDDING RECORD I-RECORD30

KERNEL ROM
TDM EXHAUSTIVE

TEST LENGTH FOR 100% FAULT COVERAGE IS: 64000
UPPER BOUND ON FAULT COVERAGE 100
EXTRA I/O SIGNALS: 2

EMBEDDING DATA:

DRIVERS	(R6)
DRIVER-MODIFICATIONS	((LFSR))
RECEIVERS	(R2)
RECEIVER-MODIFICATIONS	((SA))
THE MODIFICATIONS COST =	30
POTENTIAL FOR SHARING MEASURE	4

TEST PLAN :
1: (R6 LFSR :)
2: (BUS R6 +) (R1 LATCH :)
3: (ROM NIL +) (R2 SA :)

SCHEDULE INITIATION DELAY D = 1 OPTIMAL

5 MODES ARE TESTED FOR FREE IN THIS EMBEDDING

EXHAUSTIVE EMBEDDING SCORE = 633
GLOBAL EMBEDDING SCORE = 4510

MEASURES SUMMARY

AREA COST	TEST TIME	PERF. DEGRD	FOR FREE	REG. SHAR	I/O SIG.	SOFT REQ.	ATE REQ.	SOL. HOMG.	TDM BIAS	SCORE
I-RECORD30	EXHAUSTIVE:									
30	64000	0	5	4	2	0	0	0	1	4510

EXECUTED?? T

Figure 8-9: Detailed data of the best embedding for the ROM.

embedding as produced by TDES1 is shown in Figure 8-10. Registers R4, R5 and R6 are linked together to form a shift register that can be used during the test mode to scan-in the test vectors and to scan-out the responses of C. In this example, three different TDMs were used for illustrating purposes.

The final version of the CUC is shown in Figure 8-6e with the modifications indicated. TDES1 reported that the bus, the MUX, and most of the modes of the registers are all tested for free while testing the three main kernels. For example the MUX is exercised in both I-modes while testing the PLA and C. Similarly, the two I-modes of the bus are exercised while testing the PLA and the ROM. However, registers R2, R4, and R5 are not exercised in the parallel latching mode, and hence require further attention. However, exercising these modes of operation is a simple task and does not require any additional hardware modification. TDES1 identifies the modes of operation of structures which are not covered by instantiated embeddings.

EMBEDDING RECORD I-RECORD46

KERNEL	C
TDM	SCAN-PATH

TEST LENGTH FOR 99% FAULT COVERAGE IS	120.0
UPPER BOUND ON FAULT COVERAGE	100
EXTRA I/O SIGNALS:	3

AN ATE IS NEEDED FOR THIS TDM WITH COMPLEXITY 10

EMBEDDING DATA:

DRIVERS	(R4 R5)
DRIVER-MODIFICATIONS	(NIL (SHIFT))
RECEIVERS	(R6)
RECEIVER-MODIFICATIONS	((SHIFT))
THE MODIFICATIONS COST =	20
POTENTIAL FOR SHARING MEASURE	0

```
TEST PLAN :
1:  (R4 SHIFT +)  (R5 SHIFT :)
2:  (R4 SHIFT +)  (R5 SHIFT :)
      . . . . . . . . . .
      . . . . . . . . . .
24:  (R4 SHIFT +)  (R5 SHIFT :)
25:  (C NIL +)  (MUX C +)  (R6 LATCH :)
26:  (R6 SHIFT :)
27:  (R6 SHIFT :)
       . . . .
       . . . .
41:  (R6 SHIFT :)
```

SCHEDULE INITIATION DELAY D = 24 OPTIMAL

6 MODES ARE TESTED FOR FREE IN THIS EMBEDDING

SCAN-PATH EMBEDDING SCORE = 30.11
GLOBAL EMBEDDING SCORE = 670

MEASURES SUMMARY

AREA COST	TEST TIME	PERF. DEGRD	FOR FREE	REG. SHAR	I/O SIG.	SOFT REQ.	ATE REQ.	SOL. HOMG.	TDM BIAS	SCORE
20	2880	1	6	0	3	3	10	0	1	670

Figure 8-10: Detailed data of the best scan path embedding for C.

Chapter 9
Conclusions and Future Research

This chapter summarizes the main contributions of this thesis, and suggests some further research topics.

9.1. Summary

The main goal of the research described in this thesis was to develop a knowledge based system which can aid a designer in creating easily testable VLSI circuits. The validity of the concepts presented in the dissertation have been illustrated by a prototype system.

In Chapter 2 the design level at which we approach the problem was identified, and a formal hierarchical graph model of relevant design data of VLSI circuits was described. In Chapter 3 the concept of a TDM which incorporates structural, behavioral, and qualitative and quantitative aspects of known DFT techniques was developed. This methodology provides a designer with a systematic design for testability synthesis approach. A frame model for representing TDMs and which acts as the basic building block of the knowledge base of TDES was presented.

In Chapter 4 the problem of partitioning a circuit into a number of kernels was examined. A three phase partitioning scheme was presented that incorporates (1) top-down partitioning, (2) flat-design partitioning, and (3) bottom-up clustering. The new concepts of I-modes and I-paths were also introduced and used to identify circuit components that can potentially be useful in testing a given kernel. These simple, yet powerful concepts represent an important departure from previous efforts in the DFT field.

In Chapter 5 the mechanism and rules that govern the process of embedding TDMs into a circuit were presented. Techniques for exploring all feasible ways of making a kernel testable using one of the available TDMs were illustrated. The new concept of a test plan was also introduced, and techniques for systematically generating the test plans associated with embeddings were described. Various measures which are used to evaluate embeddings were also developed, and a scheme for selecting an embedding for a particular kernel based on these measures and which takes into consideration the global side effects was then presented.

In Chapter 6 we introduced the concept of a test schedule which incorporates pipelining concepts in the execution of a test plan associated with a particular TDM embedding in a circuit. Theoretical lower bounds on the initiation delay associated with a test schedule were derived. Techniques were then presented to construct test schedules with optimal initiation delays, hence optimizing the test time required to execute a test plan.

In Chapter 7 a global strategy for controlling the operation of TDES was described. A heuristic for ordering the kernels' processing tasks which aims at enhancing the quality of the global solutions produced by TDES was presented. We also described how TDES takes advantage of structures which get tested for free while testing other kernels.

The first prototype implementation of TDES was the subject of Chapter 8. The user interface with that system and a case study were presented. The prototype system illustrated the validity of our concepts and the feasibility and usefulness of our methodology.

9.2. Future Research

In this thesis we developed the groundwork of TDES and illustrated the feasibility of our approach by means of a prototype system. There are many problems which require further investigation. Some of them involve implementation and developmental issues, while others are concerned with more fundamental conceptual issues.

9.2.1. Developmental Tasks

1. The model described in Chapter 2 assumes the availability of certain data about designs at a particular level of abstraction. In order to tie TDES to the ADAM system, and possibly to other CAD systems as well, such data has to be extracted from the often large spaces of design information. Translators are also required to bridge the gaps in the representation and the level of abstraction which may exist between TDES internal models and those in other CAD systems.

2. Another developmental project is to construct a large library of TDM frames which incorporates most of the useful DFT techniques currently available.

3. Upgrading TDES1 to incorporate missing features such as top-down partitioning, and P/S and S/P I-paths. More complicated TDM frames can also be added to the system to enhance its capabilities. As the state of TDES evolve, it is very important for the prototype system to evolve as well. Not only will this validate the results of the research, but also may identify problems which are not apparent from a researcher point of view.

4. The use of PROLOG and/or rule-based expert system languages for future implementations of TDES should also be investigated.

9.2.2. Conceptual Problems

1. Investigating the feasibility of extending the scope of TDES to cover fault-diagnosis. The scope of this thesis has been confined to fault detection. As was illustrated by Lemmas 5.14 and 5.15, using our methodology one cannot distinguish between faults affecting different points along an I-path. However, in certain environments, it is also of interest to identify the exact location of a failure. Even though the use of I-paths seem to be incompatable with diagnosability, they may prove to be instrumental in carrying out effect-cause analysis which is the basis of diagnosis.

2. Extending the scope of TDES to cover the rich field of fault-tolerance. A frame model for capturing Fault Tolerant Design Methodologies (FTDMs) should be constructed, and tools for measuring and evaluating fault-tolerant designs should be developed.

3. There are a number of possible research areas related to the partitioning problem.

   a. Extending the clustering scheme to incorporate new clustering rules.

   b. Developing a scheme for adjusting the clustering parameters according to the user goals and constraints. This scheme should take into consideration the key architectural features of the CUC.

c. Investigating functional partitioning of logic and its relationship to functional testing methodologies.

4. Developing better techniques for computing accurate values of the embedding measures. New measures can also be developed and more elaborate scoring schemes can be investigated.

5. Extending the I-mode concept to incorporate transformation modes (T-modes) in which there is a one-to-one correspondence between the inputs and the outputs of a structure. The simplest example of a structure with a T-mode is an inverter. Even though the output data of an inverter are not the same as its input data, the information content of the output is the same as that of the input. Hence, it might be feasible to have an inverter as a part of a path supplying test data or receiving test responses. For example, having an inverter in the path between a BILBO register and the kernel, does not prevent the applicability of the BILBO TDM. The same argument is also true for scan path embeddings. In this case a complemented version of the test vectors should be scanned in. In addition, it is also feasible to have an inverter as a part of an output I-path in a BILBO or a scan-path embedding. Note, however, that a structure with an arbitrary one-to-one transformation function can still be used as a part of a BILBO embedding, but not as a part of a scan path embedding. Clearly, new embedding rules which realize such T-modes should be developed. It is interesting to note that paths which incorporate T-modes can be regarded as a generalized form of sensitization paths.

6. The hard problem of selecting a good global strategy while designing a testable VLSI circuit requires further consideration. Some possible useful directions are listed below.

a. Partitioning the TDMs into a number of non disjoint groups, each representing a global TDM (GTDM) with well defined global characteristics. For example, one GTDM can represent the fully BIST TDMs, while another can represent TDMs which employ external testing. A scheme can then be devised to select one of these GTDMs depending on the design goals and constraints, and possibly on the key architectural features of the CUC. Large weights can then be assigned to the members of the selected GTDM, hence favoring their use during the embedding.

b. Developing procedures for finding optimal testable designs with respect to just one of the embedding measures. For example, finding a testable design which minimizes area overhead, or finding the design with the minimal test time. One can then use these extreme solutions as bounds to guide the search for a feasible solution which satisfies all design requirements.

c. More elaborate scheme are needed for initially setting the priority weights associated with the embedding measures and for dynamically updating them whenever design constraints are violated. The relationship between the various measures and their sensitivity to the design constraints should be examined. Situations in which a solution is not attainable with the given set of constraints should be realized. The conflict analysis ideas of the PLA-ESS system [Breuer 85a] could be useful here.

7. The problem of designing a test controller to support the testing

activities of should be studied. The test controller should be capable of supplying the correct control signals to execute the test schedules generated by TDES. Such controller could be designed as a part of the CUC or it can be implemented in a separate chip. There is also the option of using an ATE as an external test controller. The trade-offs should be carefully examined with respect to the design goals and constraints. Preliminary results on this subject are reported in [Breuer 85b]. Selecting a design style for the controller can be regarded as a subtask in the global strategy selection process of TDES.

Appendix A
Examples of TDM Frames

A.1. The BILBO TDM Frame

The BILBO TDM frame has been discussed in detail in Chapter 3. For completeness it is reproduced in Figure A-1. This frame can be referred to as a Parallel/Parallel Random TDM frame.

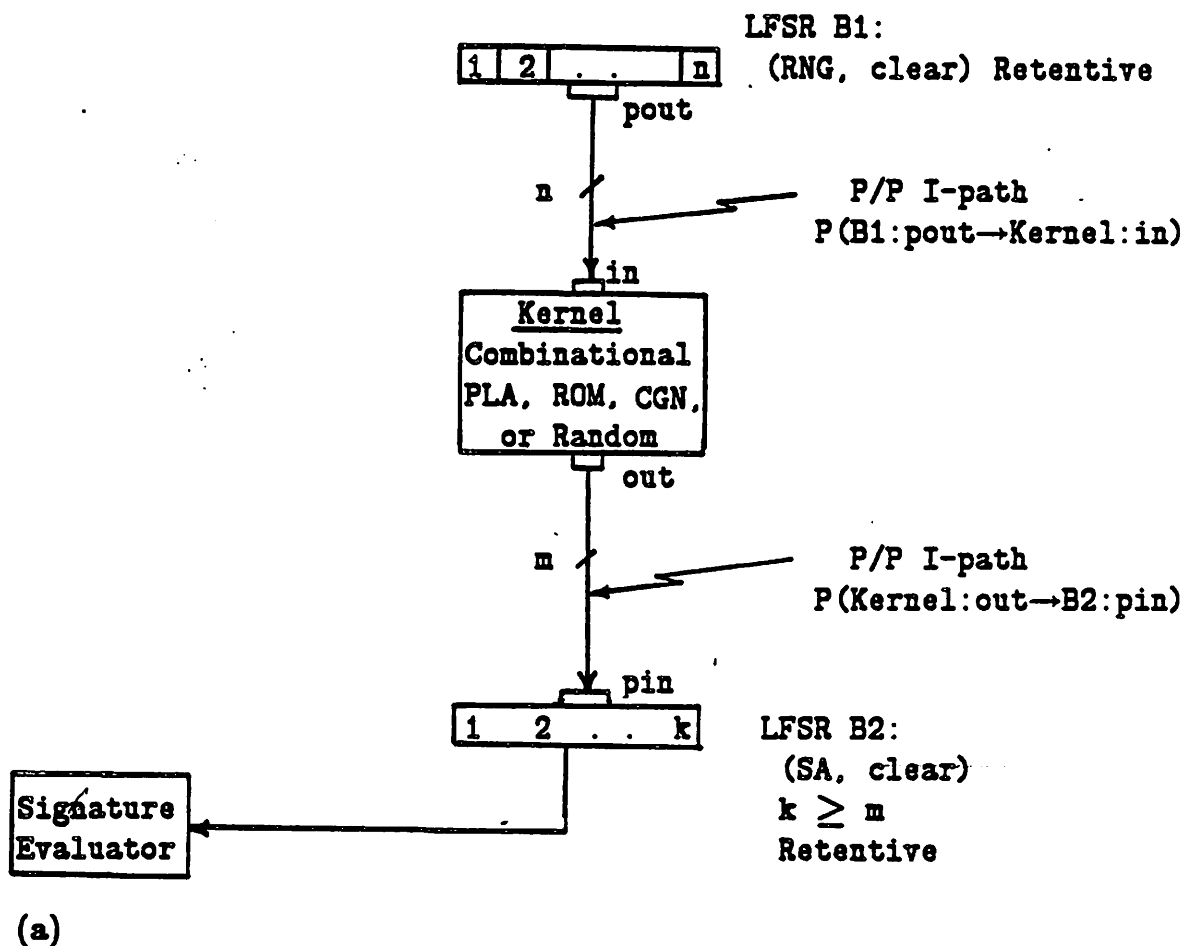


Figure A-1: The BILBO TDM frame (a) the structural template (b) the test schema (c) BILBO measures.

```
Head
   B1 ( clear )
   B2 ( clear )

Body
   Execute T times:
       B1 ( Random Number Generation )
       Transfer ( B1 output -n-> Kernel input )
       Kernel ( - )
       Transfer ( Kernel output -n-> B2 input )
       B2 ( Signature Analysis )

Tail
      Transfer (B2 output -k->Signature Evaluator)
```

(b)

```
Test creation
  software    : (fault-simulation, signature-evaluation)
ATE           : 2
Area overhead : ( Area(n-bit LFSR) - Area(n-bit  register))
                + ( Area(k-bit LFSR) - Area(k-bit  register))
                ≈ (n+k) × (3000-1200)λ² [Newkirk 83].
Performance degradation:
                Set-up delay(LFSR) - Set-up delay(register)
Extra I/O signals: 3
Test length   : random-test-estimator-function
Test time     : ≈ test length × D, where D is
                the test schedule initiation delay
Fault coverage:  (single-stuck-at 100×(1 - 2^-k) )
Storage requirement: k bits
```

(c)

Figure A-1: (continued) BILBO TDM frame
(b) the test schema (c) BILBO measures.

A.2. Serial/Serial Random TDM Frame

The serial/serial random TDM is a variation of the BILBO TDM. The basic difference is that the parallel I-paths between the kernel and the two LFSRs are replaced by a S/P I-path from the driver LFSR to the kernel, and a P/S I-path from the kernel to the receiver LFSR. As shown in Figure A-2a, two shift registers are needed to realize the two I-paths. One can view this TDM as a mixture of scan path and BILBO, where the input scan chain is fed by a random number generator and the output scan chain is fed into a signature analyzer. The advantage of this TDM is that it often leads to a small area overhead because the two LFSRs can be shared by different kernels, and at the same time avoids the high cost of test generation and ATE normally associated with scan path designs.

The test schema and some measures for the serial/serial random TDM are shown in Figures A-2b and c, respectively. Note that any test schedule for the serial/serial random TDM cannot have an initiation delay less than m due to the serial shifting of the kernel response out of S2.

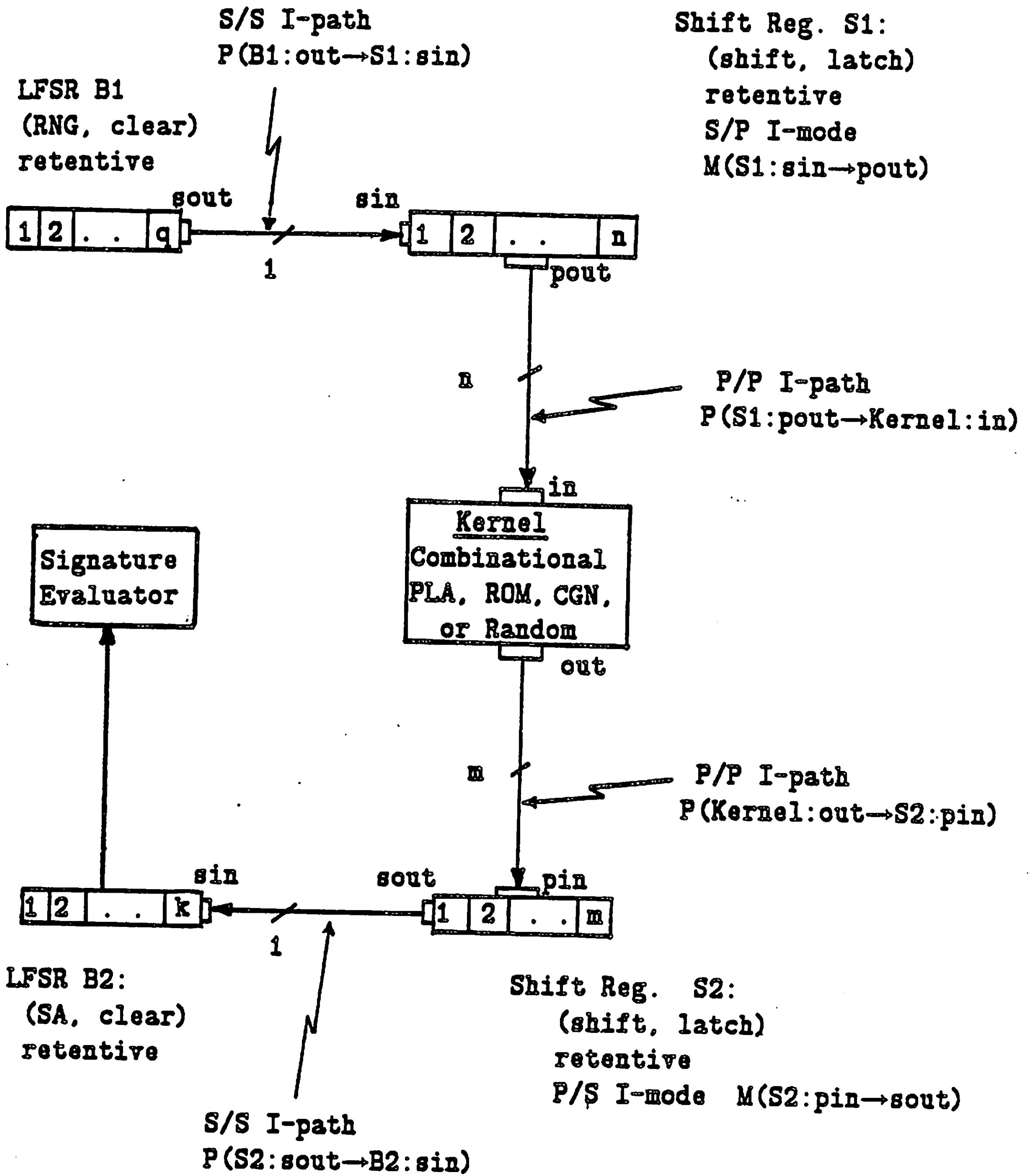


Figure A-2: The serial/serial random TDM frame
(a) the structural template (b) the test schema
(c) some measures.

```
Head
   B1 (clear)   B2 (clear)
   Execute n-1 times:
      B1(Random Number Generation)
      Transfer (B1 serial output -1-> S1 serial input )
      S1(shift)
Body
   Execute T times:
      B1(Random Number Generation)
      Transfer (B1 serial output -1-> S1 serial input)
      S1(shift)
      Transfer (S1 parallel output -n-> kernel input)
      Kernel ( - )
      Transfer ( Kernel output -m-> S2 parallel input)
      S2(Latch)
      Execute m-1 times:
         Transfer (S2 serial output -1-> B2 serial input)
         B2 (Signature Analysis)
         S2 (shift)
      Transfer (S2 serial output -1-> B2 serial input)
      B2 (Signature Analysis)

Tail
   Transfer (B2 output -k-> Signature Evaluator)
```

(b)

```
Test creation software: (fault-simulation, signature-evaluation)
ATE:               2
Area overhead: Area( structures matching S1, S2, B1, and B2) -
          Area(before modification) + Area(routing new  I-paths)
Performance degradation:
           Set-up delay(shift register) - Set-up delay(register)
Extra I/O signals: 4
Test length:        random-test-estimator-function
Test time:          ≈ test length × D
Fault coverage:     ((single-stuck-at 100×(1 - 2^-k) )
```

(c)

Figure A-2: (continued) The serial/serial random TDM frame (b) the test schema (c) some measures.

A.3. Serial/Parallel Random TDM Frame

The serial/parallel random TDM is another variation of the BILBO TDM. In this TDM, a S/P I-path connects the driver LFSR to the kernel. The kernel output is transferred via a P/P I-path to the receiver LFSR to generate a signature. The structural template of the serial/parallel random TDM is illustrated in Figure A-3a. Clearly, signature analysis is done in parallel here, hence avoiding the serial shifting of response data associated with the serial/serial random TDM.

The test schema and the measures of the serial/parallel random TDM are shown in Figures A-3b and c, respectively. Note that for the schema of the serial/parallel random TDM, it is quite possible to have an embedding which would support a test schedule with a minimal initiation delay of 1.

Another version of the basic BILBO TDM is the parallel/serial random TDM, where random vectors are transferred in parallel from the RNG to the kernel, and from the kernel to the SA register via a parallel to serial I-path.

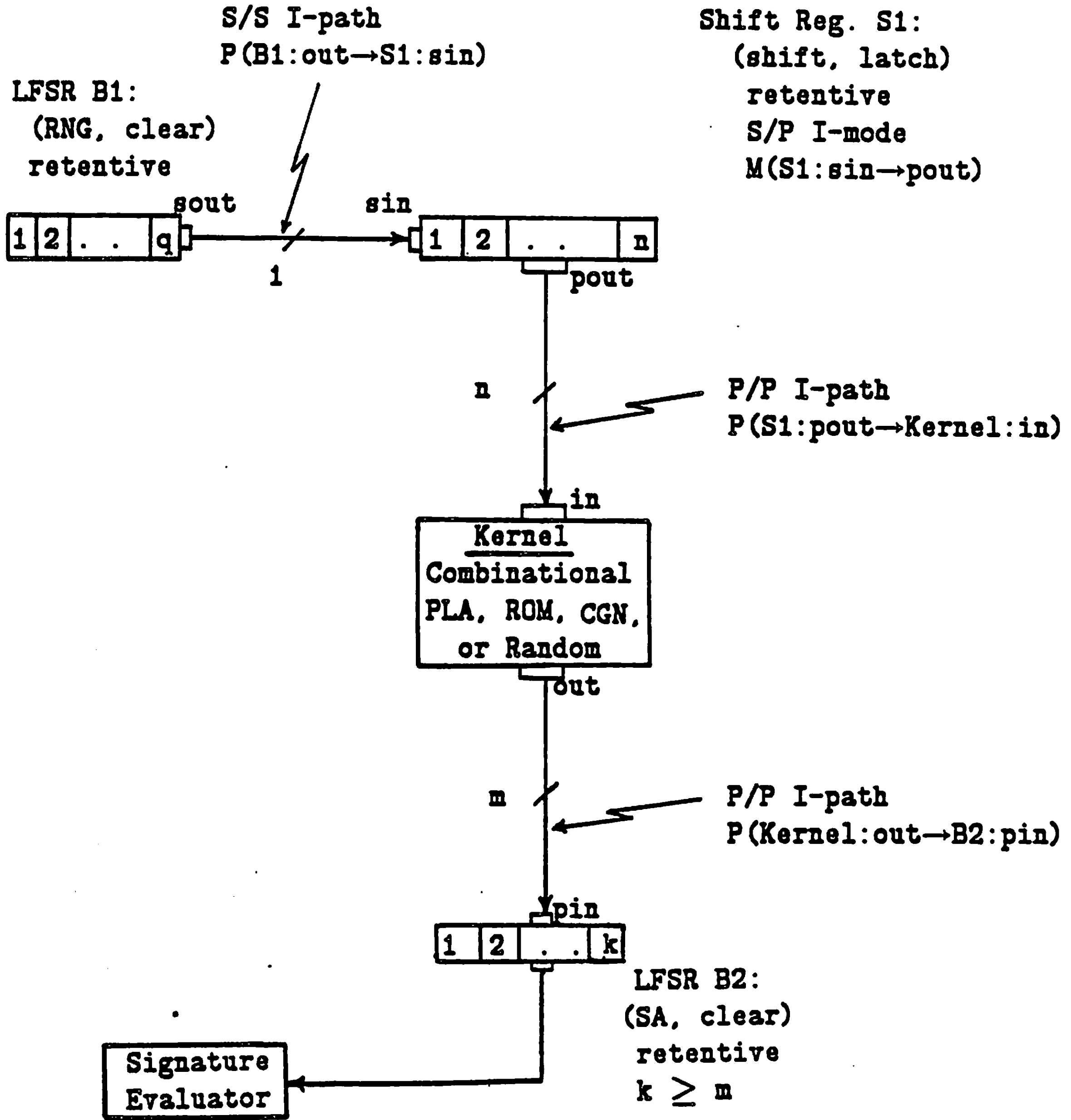


Figure A-3: The serial/parallel random TDM frame (a) the structural template (b) the test schema (c) some measures.

```
Head
   B1 (clear)
   B2 (clear)
   Execute n-1 times:
      B1(Random Number Generation)
      Transfer (B1 serial output -1-> S1 serial input)
      S1(shift)

Body
   Execute T times:
      B1(Random Number Generation)
      Transfer (B1 serial output -1-> S1 serial input)
      S1(shift)
      Transfer (S1 parallel output -n-> kernel input)
      Kernel ( - )
      Transfer (Kernel output -m-> B2 parallel input)
      B2 (Signature Analysis)

Tail
   Transfer (B2 output -k-> Signature Evaluator)

[(b)]

Test creation software: (fault-simulation, signature-evaluation)
ATE:                 2
Area overhead: Area( structures matching S1, B1, and B2) -
         Area(before modification) + Area(routing new  I-paths)
Performance degradation:
          Set-up delay(LFSR) - Set-up delay(register)
Extra I/O signals: 3
Test length:        random-test-estimator-function
Test time:          ≈ test length × D
Fault coverage:     ((single-stuck-at 100×(1 - 2^-k) )
(c)
```

Figure A-3: (continued) The serial/parallel random TDM frame (b) the test schema (c) some measures.

A.4. The Exhaustive TDM Frame

The Exhaustive TDM frame is very similar to the BILBO TDM. The key difference is that the test length of the exhaustive TDM is 2^n, where n is the number of kernel inputs. The template, test schema, and some of the measures of the exhaustive TDM are shown in Figure A-4. Similar to the BILBO TDM, the exhaustive TDM has three variations: a serial/serial version; a parallel/serial version; and a serial/parallel version.

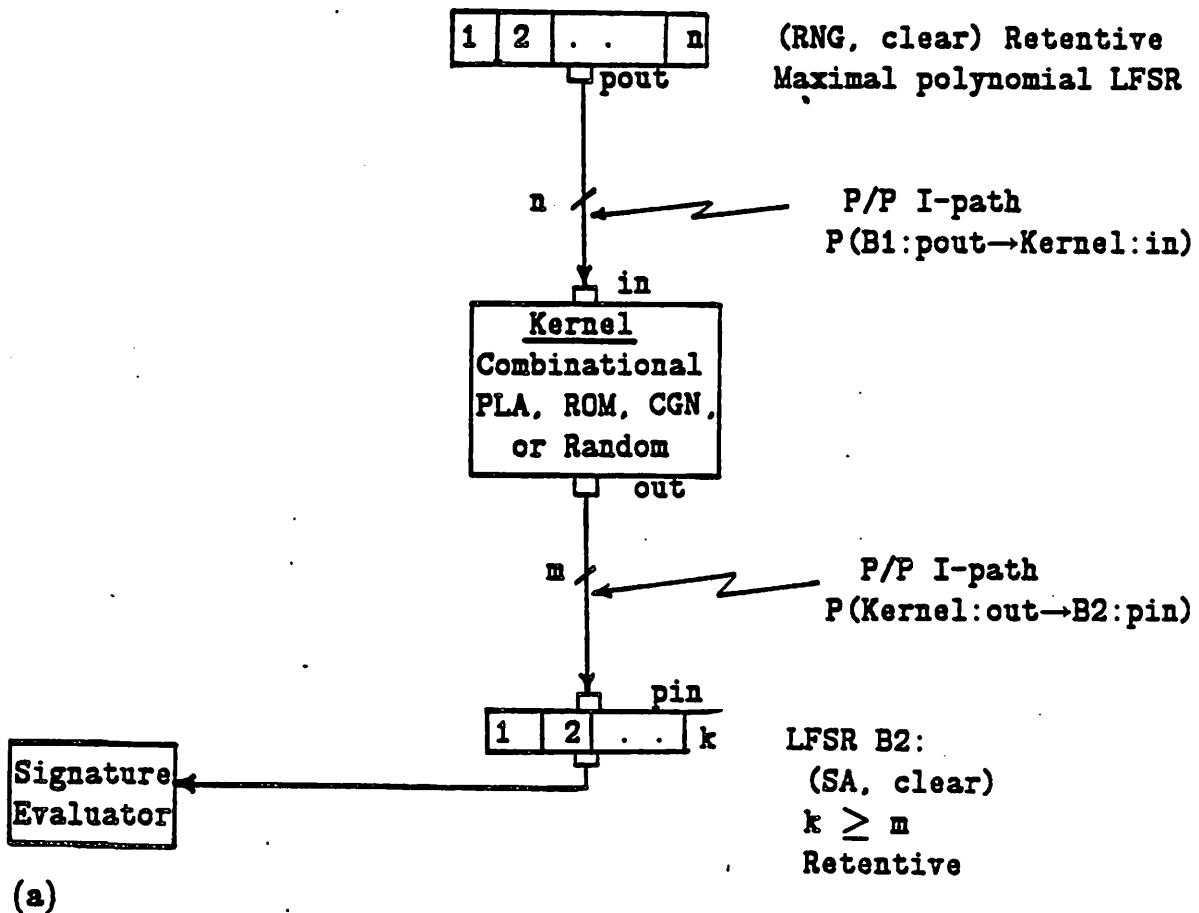


Figure A-4: The Exhaustive TDM frame (a) the structural template (b) the test schema (c) some measures.

```
Head
   B1 ( clear )
   B2 ( clear )

Body
   Execute 2^n times:
       B1 ( Random Number Generation )
       Transfer ( B1 output -n-> Kernel input )
       Kernel ( - )
       Transfer ( Kernel output -m-> B2 input )
       B2 ( Signature Analysis )

Tail
   Transfer ( B2 output -k-> Signature Evaluator)
```

(b)

Test creation software : (signature-evaluation)
ATE : 2
Area overhead : (Area(n-bit LFSR) - Area(n-bit register))
+ (Area(m-bit LFSR) - Area(m-bit register))
Performance degradation:
Set-up delay(LFSR) - Set-up delay(register)
Extra I/O signals: 3
Test length : 2^n
Test time : $\approx$ test length $\times$ D, where D is the test schedule initiation delay.
Fault coverage: ((single-stuck-at $100\times(1 - 2^{-k})$)
(functional $100\times(1 - 2^{-k})$))

(c)

Figure A-4: (continued) The exhaustive TDM frame
(b) the test schema (c) some measures.

A.5. The Scan Path TDM Frame

Figure A-5 illustrates the various components of the scan path TDM frame. Similar frames can also be constructed for the LSSD TDM and the Scan/Set TDM.

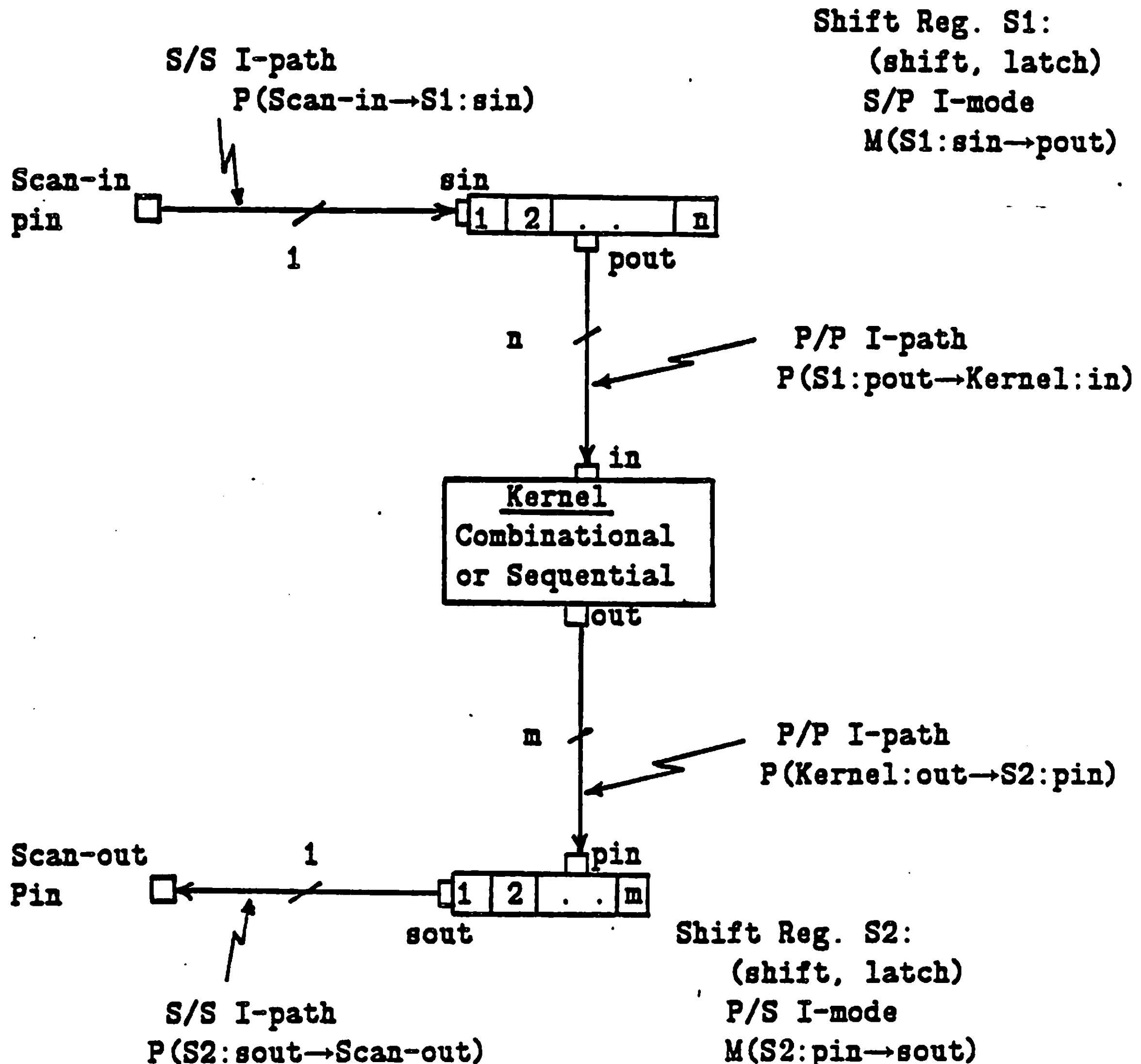


Figure A-5: The scan path TDM frame
(a) the structural template (b) the test schema (c) measures.

```
Body
   Execute T times:
      Execute n times
          Transfer (Scan-in pin -1-> S1 serial input)
          S1(shift)
      Transfer (S1 parallel output -n-> kernel input)
      Kernel ( - )
      Transfer (Kernel output -m-> S2 parallel input)
      S2 (Latch)
      Execute m-1 times
          Transfer (S2 serial output -1-> Scan-out pin)
          S2(shift)
      Transfer (S2 serial output -1-> Scan-out pin)
```

(b)

```
Test creation
  software    : (circuit-modeling, test-generation,
                 fault-simulation, ATE-program-generation)
ATE           : 10
Area overhead: Area( structures matching S1 and S2) -
        Area(before modification) + Area(routing created  I-paths)
Performance degradation:
               Set-up delay( latch mode of a shift register) -
               Set-up delay( latch mode of a latch register)
Extra I/O signals: 4
Test length   : deterministic-test-estimator-function
Test time     : ≈ test length × D, where D ≥ Max(n, m)
Fault coverage:  ( (single-stuck-at 100) (multiple-stuck-at 100) )
External storage
  requirement : ≈ test length × (n + m)
```

(c)

Figure A-5: (continued) The scan path TDM frame
(b) the test schema (c) some measures.

A.6. The SPLASH TDM Frame

The SPLASH TDM (stands for Scan Path with Look Ahead Shifting) [Abadir 85d] has the same structural template of the scan path frame, except that the driver shift register has to be retentive. The basic difference lies in the test schema of SPLASH. After applying a vector v_i to the kernel , rather than shifting-in the next test vector v_{i+1}, we shift-in only the first k bits of v_{i+1}. At this point S1 contains the first k bits of v_{i+1} and the last n-k bits of v_i. This vector is used as the next test vector, and is applied to the kernel inputs and the kernel responses are latched into S2 and shifted-out. This process is then repeated for the next k bits of v_{i+1}. After $r = \lfloor n/k \rfloor$ iterations, we require that S1 contain v_{i+1}. Ideally, k is selected such that it divides n. If this is not the case then one has to adjust the number of shifts every r iterations. For example, if n = 17 and k is selected to be 4, then the sequence of shifts is 4, 4, 4, 5. Note that k = n corresponds to the normal scan path schema. The schema of the SPLASH TDM frame is shown in Figure A-6a. Some measures of the SPLASH TDM are shown in Figure A-6b.

```
Body
  Execute T×r times:
    Execute k times
        Transfer (Scan-in pin -1-> S1 serial input)
        S1(shift)
    Transfer (S1 parallel output -n-> kernel input)
    Kernel ( - )
    Transfer (Kernel output -m-> S2 parallel input)
    S2 (Latch)
    Execute m-1 times
        Transfer (S2 serial output -1-> Scan-out pin)
        S2(shift)
    Transfer (S2 serial output -1-> Scan-out pin)
```

(a)

```
Test creation
  software     : (circuit-modeling, SPLASH-test-generation,
                  fault-simulation, ATE-program-generation)
ATE            : ?
Area overhead: Area( structures matching S1 and S2) -
        Area(before modification) + Area(routing created  I-paths)
Performance degradation:
               Set-up delay( latch mode of a shift register) -
               Set-up delay( latch mode of a latch register)
Extra I/O signals: 4
Test length:   SPLASH-estimator-function [Abadir 85d]
Test time: ≈ test length × D × r, where D ≥ max(m, k)
Fault coverage:  ( (single-stuck-at 100) (multiple-stuck-at 100) )
External storage
  requirement : ≈ test length × (n + m × r)
```

(b)

Figure A-6: The SPLASH TDM frame (a) the test schema (b)SPLASH measures.

A.7. The Scan TDM Frame for Multiplexers

Figure A-7 illustrates the various components of a TDM frame which can be used for MUXs (or busses). In this TDM one of the I-modes of a MUX, M(MUX:i→out), is fully tested functionally. Both the integrity of the data lines involved in that I-mode and the selection control required to activate that mode are verified. Test data are scanned in from the outside and the MUX output is also scanned out. To completely test a MUX functionally, one has to test every one of its I-modes. Note that port j of the MUX represents all the input sources of a MUX other than input source i. Hence, to embed this TDM into the subgraph of an x-to-1 MUX, port j of the structural template should be mapped into the x - 1 unselected input sources of the real MUX. Note also that the driver register of port j in the structural template has to be retentive in order to make sure that the data on port j are held unchanged while testing the I-mode of port i. Note that the parallel I-path P(R → Kernel:j) requirement is stringent. It could be relaxed to simply a path between a retentive register and port j. Hence, allowing structures without I-modes to exist along the path. Recall that our objective is to ensure that the data on port j are different from the data being transferred through the MUX.

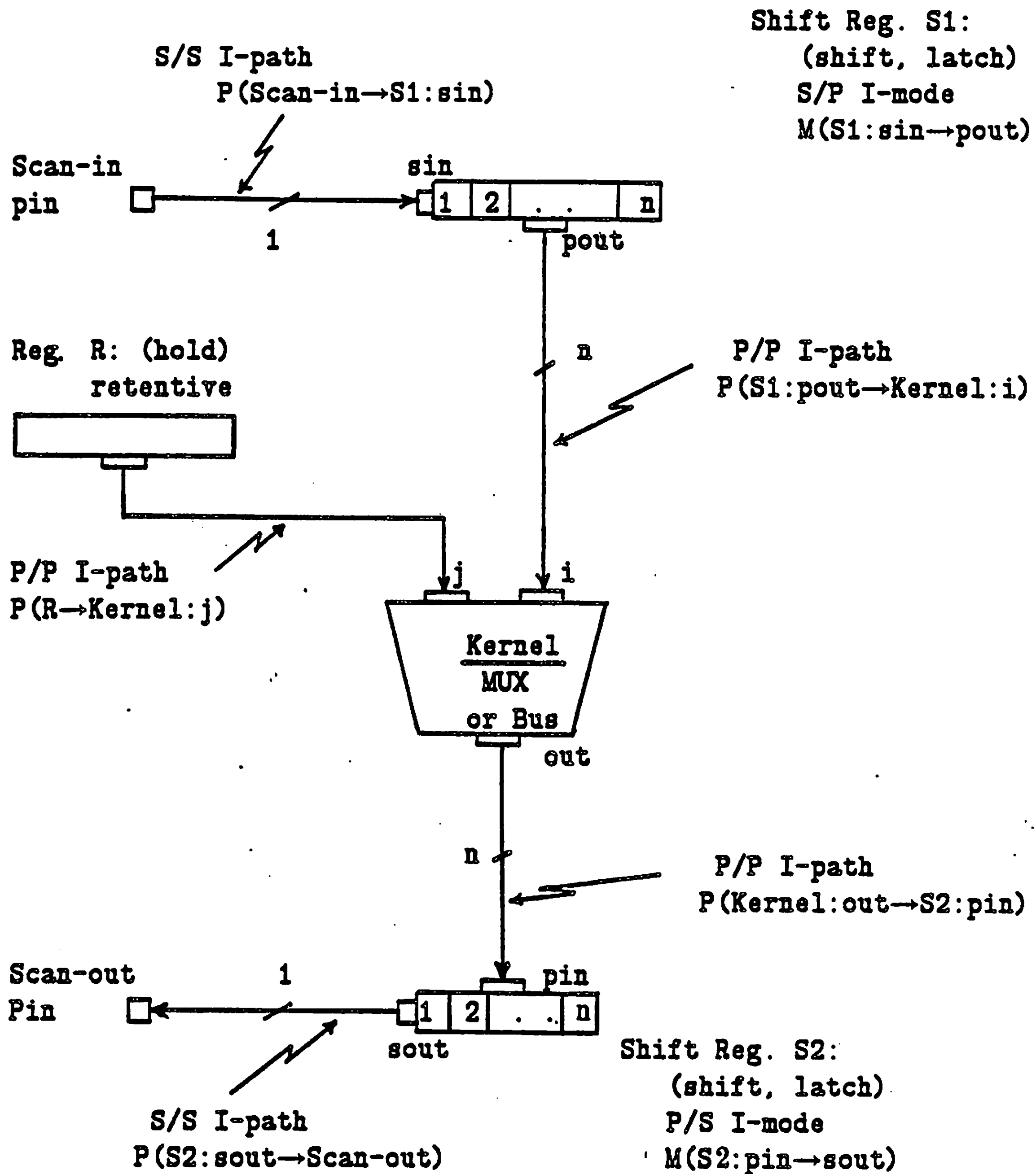


Figure A-7: The frame of the scan TDM for MUXs
(a) the structural template (b) the test schema (c) measures.

```
Body
   Execute T times:
      Execute n times
          Transfer (Scan-in pin -1-> S1 serial input)
          S1(shift)
      R(Hold)
      Transfer (S1 parallel output -n-> kernel input source i)
      Transfer (R parallel output  -n-> kernel input source j)
      kernel (select input source i)
      Transfer (Kernel output  -m-> S2 parallel input)
      S2 (Latch)
      Execute  m-1 times
          Transfer (S2 serial output -1-> Scan-out pin)
          S2(shift)
      Transfer (S2 serial output -1-> Scan-out pin)
```

(b)

```
Test creation
  software     : (MUX-test-generation, ATE-program-generation)
ATE            : 6
Area overhead: Area( structures matching S1 and S2) -
       Area(before modification) + Area(routing created  I-paths)
Performance degradation:
               Set-up delay( latch mode of a shift register) -
               Set-up delay( latch mode of a latch register)
Extra I/O signals: 3
Fault coverage: ( (single-stuck-at 100) (multiple-stuck-at 100)
                 (shorts 100) (functional 100))
Test length    : n
Test time      : ≈ test length × D, where D ≥ n
External storage requirement : n^2
```

(c)

Figure A-7: (continued).

A.8. The Scan TDM Frame for Shift Registers

The Scan TDM for shift registers is used to test the shift mode of operation of a register. Test data are scanned in from the scan-in pin and the register response are scanned-out to the scan out pin. A test pattern 101010... of length n-bits, where n is the length of the shift register, can be used. The various components of the frame are shown in Figure A-8.

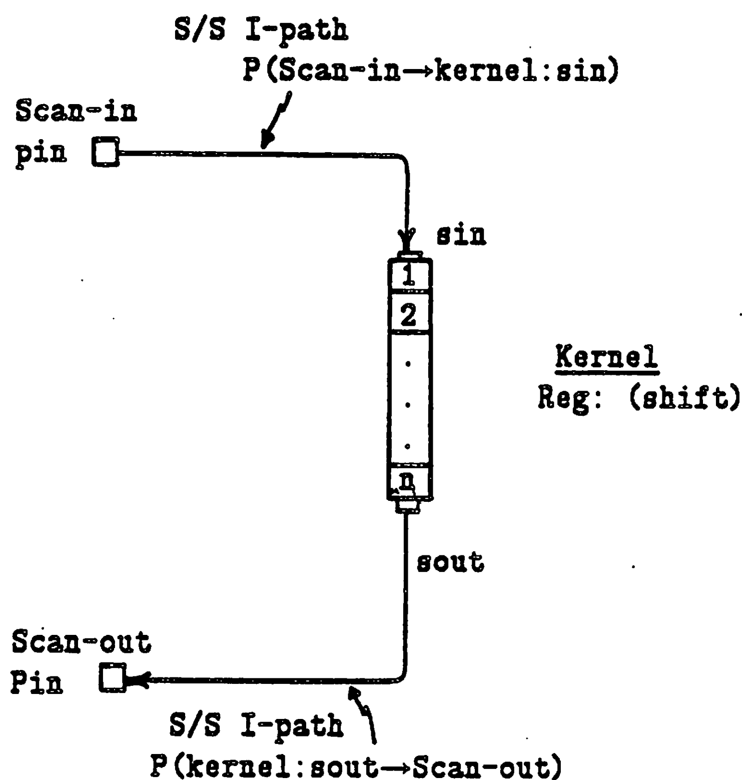


Figure A-8: The scan TDM frame for shift registers (a) the structural template (b) the test schema (c) measures.

```
Body
   Execute n times:
        Transfer (Scan-in pin 1→ kernel serial input)
        kernel(shift)
   Execute n-1 times:
        Transfer (kernel serial output 1→ Scan-out pin)
        kernel(shift)
   Transfer (kernel serial output 1→ Scan-out pin)
```

(b)

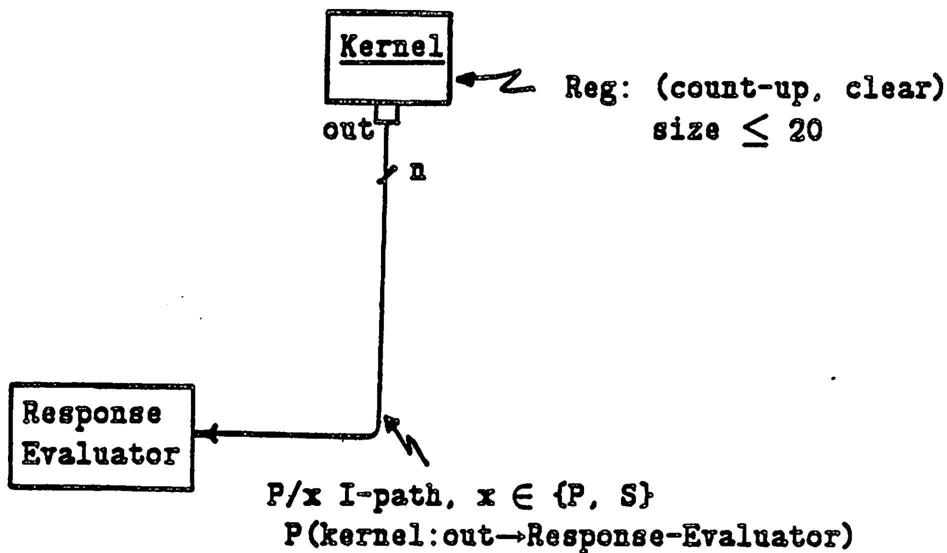
```
Test creation software : none
ATE            : 6
Area overhead : Area (routing created  I-paths)
Performance degradation:
               Set-up delay( latch mode of a shift register) -
               Set-up delay( latch mode of a latch register)
Extra I/O signals: 2
Fault coverage: ( (single-stuck-at 100) (multiple-stuck-at 100))
Test length    : n
Test time      : 2×n - 1 + length of both scan chains used.
External storage  requirement : n
```

(c)

Figure A-8: (continued).

A.9. A Counter TDM Frame

This Counter TDM frame is used to test the count-up mode of operation of a register. The counter output is transferred via a P/x I-path, where x ∈ {P, S}, to the response evaluator which can either be an on-chip test controller, or an off-chip ATE. Counters with more than 20 bits cannot be tested with this TDM (other bounds can be used instead of 20). The various components of the frame are shown in Figure A-9.



(a)

Figure A-9: A counter TDM frame
(a) the structural template (b) the test schema(c) some measures.

```
Head
    kernel(clear)

Body
   Execute 2^n times:
        kernel(count-up)
        Transfer (kernel output -n-> Response Evaluator)
```

(b)

```
Test creation software : none
ATE            : 4
Area overhead : Area (routing created  I-paths)
Performance degradation: Delay( latch mode of a shift register) -
                         Delay( latch mode of a latch register)
Extra I/O signals: 1
Fault coverage: ( (single-stuck-at 100) (multiple-stuck-at 100))
Test length    : 2^n
Test time      : ≈ test length × D
External storage  requirement : 0
```

(c)

Figure A-9: (continued)

References

[Abadir 85a] Abadir, M.S. and M.A. Breuer.
A knowledge based system for designing testable VLSI chips.
IEEE Design and Test of Computers :56-68, August, 1985.

[Abadir 85b] Abadir, M.S. and M.A. Breuer.
Constructing optimal test schedules for VLSI circuits having built-in test hardware.
In *Proc. 15th Int'l Fault-Tolerant Computing Conference*, pages 165-170. June, 1985.

[Abadir 85c] Abadir, M.S. and M.A. Breuer.
Test schedules for VLSI circuits having built-in test hardware.
Submitted to IEEE Transactions on Computers , July, 1985.

[Abadir 85d] Abadir, M.S. and M.A. Breuer.
Scan path with look ahead shifting (SPLASH).
Technical Report 7220-85-322, Hughes Aircraft Co. Technical Internal Correspondence, October, 1985.

[Abadir 85e] Abadir, M.S. and M.A. Breuer.
TDES1 user and maintenance manual.
USC technical report , November, 1985.

[Agrawal 81] Agrawal, V.K. and E. Cerny.
Store and generate built-in testing approach.
In *Digest of Papers 11th Int'l Symp. on Fault-Tolerant Computing*, pages 35-40. June, 1981.

[Agrawal 84] Agrawal, V.D., S.K. Jain and D.M. Singer.
Automation in Design for testability.
In *Proceddings of the IEEE Custom Integrated Circuits Conference*, pages 159-163. May, 1984.

[Ando 80] Ando, H.
Testing VLSI with random access scan.
In *Digest of Papers COMPCON 80*, pages 50-52. February, 1980.

[Bellon 83] Bellon, C., C. Robach and G. Saucier.
An intelligent assistant for test program generation: The Supercat system.
In *IEEE Int'l Conf. on Computer Aided Design*, pages 32-33. September, 1983.

[Bondy 76] Bondy, J. A. and Murty, U. S. R.
Graph Theory with Applications.
North Holland, New York, N.Y., 1976.

[Brachman 83] Brachman, R.J.
What IS-A is and isn't: An analysis of taxonomic links in semantic networks.
Computer 16:30-36, October, 1983.

[Breuer 76] Breuer, M.A. and A.D. Friedman.
Diagnosis and Reliable Design of Digital Systems.
Computer Science Press, Rochville, MD, 1976.

[Breuer 84] Breuer, M.A.
A methodology for the design of testable VLSI chips.
submitted to IEEE Design and Test of Computers, , 1984.
A modified version of this paper is available as a report SD-TR-85-33 "A methodology for the design of testable custom large scale integrated circuits", The Aerospace Corp., El Segundo Ca 90245, January 1985.

[Breuer 85a] Breuer, M.A. and X. Zhu.
A knowledge based system for selecting a test methodology for a PLA.
In *Proc. 22nd Design Automation Conference*, pages 259-265. June, 1985.

[Breuer 85b] Breuer, M.A.
The design of a test controller.
In *Submitted to Proc. 16th Int'l Fault-Tolerant Computing Conference.* 1985.
Also available as a Hughes Aircraft Co. internal report.

[Buehler 82] Buehler, M.A. and M.W. Sievers.
Off-line, built-in test techniques for VLSI circuits.
Computer 15:69-82, June, 1982.

[Davidson 71] Davidson, E.S.
The design and control of pipelined function generators.
In *Proc. Int. IEEE Conf. on Systems, Networks, and Computers*, pages 19-21. 1971.

[Davis 82] Davis, R. et al.
Diagnosis based on description of structure and function.
In *Proc. AAAI*, pages 137-142. 1982.

[Davis 83] Davis, R. and H. Schrobe.
Representing structure and behavior of digital hardware.
Computer :75-82, October, 1983.

[Dong 81] Dong H. and E.J. McCluskey.
Design of fully testable programmable logic arrays.
CRC 81-20, Stanford University, December, 1981.

[Eichelberger 77] Eichelberger, E.B. and T.W. Williams.
A logic design structure for LSI testing.
In *Proc. 14th Design Automation Conference*, pages 462-468. June, 1977.

[Friedman 73] Friedman, A.D.
Easily testable iterative systems.
IEEE Trans. on Computers C-22:1061-1064, December, 1973.

[Fujiwara 81] Fujiwara, H. and K. Kinoshita.
A design of programmable logic arrays with universal tests.
IEEE Trans. on Computers C-30:823-828, November, 1981.

[Funatsu 75] Funatsu S., N. Wakatsuki and T. Arima.
Test generation systems in Japan.
In *Proc. 12th Design Automation Conf.*, pages 114-122. June, 1975.

[Fung 85] Fung, H.S., S. Hirschhorn, and R. Kulkarni.
Design for testability in a silicon compilation environment.
In *Proc. 22nd Design Automation Conference*, pages 190-196. June, 1985.

[Granacki 85] Granacki J., D. Knapp and A. Parker.
The ADAM Advanced Design AutoMation system: overview, planner, and natural language interface.
In *Proc. of the 22nd Design Automation Conference*, pages 727-730. 1985.

[Hamscher 83a] Hamscher, W.
Using structural and functional information in diagnostic design.
In *Proc. National Conference on AI*, pages 152-156. Washington D.C., August, 1983.

[Hamscher 83b] Hamscher, W. and R. Davis.
Using structural and functional information in diagnostic.
AI memo 107, MIT, June, 1983.

[Hartley 84] Hartley, R.T.
CRIB: Computer fault finding through knowledge engineering.
Computer :76-83, March, 1984.

[Hayes 73] Hayes, J.P. and A.D. Friedman.
Test point placement to simplify fault detection.
In *Digest of Papers 10th Int'l Symp. Fault Tolerant Computing*, pages 73-78. June, 1973.

[Hayes 74] Hayes, J.P.
On modifying logic networks to improve their diagnosability.
IEEE Trans. on Computers C-23:56-62, January, 1974.

[Hortsman 83] Hortsmann, P.W.
Design for testability using logic programming.
In *Digest of Papers International Test Conference.* 1983.

[Jain 85] Jain, S.J., and V.D. Agrawal.
Statistical fault analysis.
IEEE Design and Test of Computers :38-45, February, 1985.

[Kime 82] Kime, C.R. and K. Saluja.
Test scheduling in testable VLSI circuits.
In *Proc. 12th Int'l. Sym. Fault-Tolerant Computing* , pages 406-412. June, 1982.

[Kogge 81] Kogge, P. M.
The Architecture of Pipelined Computers.
McGraw-Hill Advanced Computer Science Series, 1981.

[Konemann 79] Konemann, B. et al.
Built-in logic block observation techniques.
In *Proc. 1979 Test Conf.*, pages 37-41. October, 1979.

[Kurdahi 85] Kurdahi, F. and A. C. Parker.
Area estimation of VLSI integrated circuits.
Technical Report CRI 85-05, Dept. of EE-Systems, University of Southern California, 1985.

[Mangir 83] Mangir, T.E.,
Design for testability an integrated approach to VLSI testing.
In *IEEE Int'l Conf. on Computer Aided Design*, pages 68-70. September, 1983.

[McCluskey 81] McCluskey, E.J. and S. Bozorgui-Nesbat.
Design for autonomous test.
IEEE Trans. on Computers C-30:866-875, November, 1981.

[Mead 80] Mead, C. and L. Conway.
Introduction to VLSI systems.
Addison Wesley, 1980.

[Meehan 79] Meehan, J.R.
The new UCI LISP manual.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1979.

[Minsky 75] Minsky, M.
A framework for representing knowledge.
In *The Psychology of Computer Vision, edited by Patrick H. Winston.* McGraw-Hill, New York, 1975.

[Muehldorf 81] Muehldorf, E.I. and A.D. Savkar.
LSI logic testing - An overview.
IEEE Trans. on Computers C-30:1-17, January, 1981.

[Nau 83] Nau, D.S.
Expert computer systems.
Computer 16:63-85, February, 1983.

[Newkirk 83] Newkirk, J. and R. Mathews.
The VLSI designer's library.
Addison-Wesley, The VLSI Systems Series, 1983.

[Nilsson 71] Nilsson, N.
Problem Solving Methods in Artificial Intelligence.
McGraw-Hill, New York, 1971.

[Nilsson 80] Nilsson, N.J.
Principles of Artificial Intelligence.
Tioga, Palo Alto, CA, 1980.

[Ostapko 78] Ostapko, D.L. and S.J. Hong.
Fault analysis and test generation for PLAs.
In *Digest of Papers 8th Int'l Symp. on Fault Tolerant Computing*, pages 83-89. June, 1978.

[Park 85] Park, N.
Synthesis of optimal clocking schemes in high speed integrated circuits.
PhD thesis, Dept. of Electrical Engineering, University of Southern California, October, 1985.

[Patel 76] Patel, J.H. and E.S. Davidson.
Improving the throughput of a pipeline by insertion of delays.
In *Third Annual Symp. on Computer Architecture*, pages 163-169. 1976.

[Peterson 72] Peterson W.W. and E.J. Weldon.
2nd Edition: Error-correcting Codes.
MIT Press, Cambridge, MA, 1972.

[Saluja 75] Saluja, K.K. and R.M. Reddy.
Fault detecting test sets for Reed-Muller canonic networks.
IEEE Trans. on Computers C-24:995-998, October, 1975.

[Savir 80] Savir, J.
Syndrome-testable design of combinational circuits.
IEEE Trans. on Computers C-29:442-451, June, 1980.
(corrections: Nov 1980).

[Savir 83] Savir, J., G. Ditlow and P.H. Bardell.
Random pattern testability.
In *Proc. 13th Int'l Fault-Tolerant Computing Conference*, pages 80-89. June, 1983.

[Sedmak 79] Sedmak, R.M.
Design for self-verification: An approach for testability problems in VLSI-based designs.
In *Proc. IEEE Test Conf.*, pages 112-120. 1979.

[Sedmak 80] Sedmak, R.M.
Implementation techniques for self-verification.
In *Proc. IEEE Test Conf.*, pages 267-278. 1980.

[Seth 77] Seth, S.C. and K.L. Kondandapani.
Diagnosis of faults in linear tree networks.
IEEE Trans. on Computers C-26:29-33, January , 1977.

[Shar 72] Shar, L.E.
Design and scheduling of statically configured pipelines.
In *Digital System Lab Report SU-SEL-72-042, Stanford.* September, 1972.

[Smith 80] Smith, J. E.
Measures of the effectiveness of fault signature analysis.
IEEE Trans. on Computers C-29:510-514, June , 1980.

[Sridhar 81] Sridhar, T. and J.P. Hayes.
A functional approach to testing bit-sliced microprocessors.
IEEE Trans. on Computers C-30:563-571, August, 1981.

[Stewart 77] Stewart, J.H.
Future testing of large LSI circuit cards.
In *Digest of Papers IEEE semiconductor Test Symp.*, pages 6-17. October, 1977.

[Syed 81] Syed, Z. A.
On routing for custom integrated circuits.
PhD thesis, Dept. of Electrical Engineering, University of Southern California, July, 1981.

[Williams 79] Williams, T.W. and K.P. Parker.
Testing logic networks and design for testability.
Computer :9-21, October, 1979.

[Williams 82] Williams, T.W. and K.P. Parker.
Design for testability - A survey.
IEEE Trans. on Computers C-31:2-15, January, 1982.

[Williams 85] Williams, T.W.
Test length in a self-testing environment.
IEEE Design and Test :59-63, April, 1985.

[Winston 77] Winston, P.
Artificial Intelligence.
Addison Wesley, 1977.

[Woods 75] Woods, W.A.
What's in a link? Foundation for semantic networks.
In *Representation and Understanding: Studies in Cognitive Science, D.G. Bobrow and A. Collins, eds..* Academic Press, N.Y., 1975.