ADVANCED SERIAL SCAN DESIGN FOR TESTABILITY

Rajesh Gupta

CEng Technical Report 91-10

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
(Computer Engineering)

(Copyright May 1991)

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA. 90089-0781
(213)740-4579

March 7, 1991

Acknowledgements

I am grateful to Prof. Melvin Breuer for the inspiration and guidance I received from him during my dissertation work. In addition I would like to thank him for providing access to excellent hardware and software resources at all times. During my years at USC I benefited greatly from interacting with many colleagues and friends. In particular I would like to mention Dr. Rajiv Gupta, Kuen-Jong Lee, Jung-Cheun Lien, Rajagopalan Srinivasan, Amit Majumdar, and Dr. Charles Njinda. Sridhar Narayana helped me obtain some of the experimental results. I also wish to thank Prof. Alice Parker who gave me some useful feedback and Prof. Ming-Deh Huang who served on my dissertation committee.

I would like to acknowledge the financial support provided by the Defense Advanced Research Projects Agency through Contract No. N00014-87-K-0861 (monitored by the Office of Naval Research) and by the Semiconductor Research Corporation through Contract No. 88-DP-075.

Contents

Dedication ii

Acknowledgements iii

Abstract xiv

1 Introduction 1

1.1 Design for Testability 1

1.2 Serial Scan Design 2

1.2.1 Circuit Modification 3

1.2.2 Test Generation and Application 5

1.3 Scan Design Costs 5

1.4 Reducing Scan Design Costs 7

2 Background 11

2.1 Partitioning Approaches 11

2.2 Partial Scan Approaches 13

2.3 Multiple Scan Chain Approaches 17

3 Partial Scan Design with Balanced Structures 19

3.1 Introduction 19

3.2 Basic Circuit Model 22

3.3 B-Structures and their Properties 23

3.4 Scan Design Using B-Structures 24
3.5 Proof of Correctness 28
3.5.1 Single-Pattern Testability 28
3.5.2 Generating Single-Pattern Tests 36
3.5.3 Observations on Testing B-Structures 37
3.6 Algorithm for Scan Register Selection 39
3.6.1 Removal of Feedback Arcs 40
3.6.2 Balancing Acyclic Sequential Structures 42
3.6.2.1 Verification Procedure 42
3.6.2.2 Balancing Procedure 43
3.7 Implementation of Scan Path 46
3.7.1 Example 47
3.7.2 Construction of Scan Path 47
3.7.3 Test Application 51
3.7.4 Circuit Modifications 52
3.8 Testing Register Functional Modes 52
3.9 Case Study 55
3.10 Summary 56

4 Partial Scan Design with Unbalanced Structures **58**

4.1 Introduction 58
4.2 Optimal Test Scheduling 63
4.2.1 The Compaction Principle 63
4.2.2 Modeling Schedule Constraints 64
4.2.3 Picking a Schedule 68
4.3 Test Generation Model 71
4.3.1 Condensing the Test Generation Model 74
4.3.2 Test Pattern Generation 77
4.4 Summary 79

5 Partial Scan Design of Circuits Containing Switches **80**

5.1 Switches 80

5.2 Circuit Model 82

5.2.1 Atomic Combinational Logic Units 82

5.2.2 Generalized Topology Graph 85

5.3 Switched Balanced Structures 85

5.3.1 The Class of SB-Structures 86

5.3.2 Testability Properties of SB-Structures 88

5.4 Algorithm for Scan Register Selection 91

5.4.1 Removal of Feedback Registers 93

5.4.2 Balancing Acyclic Sequential Structures 95

5.4.2.1 Verification Procedure 95

5.4.2.2 Balancing Procedure 97

5.5 Partial Scan Testing Using I-Paths 101

5.5.1 I-Paths 102

5.5.2 Kernels with I-Paths 103

5.5.3 Unsatisfiable Kernels 106

5.6 Finding a Satisfiable Kernel 109

5.6.1 Expansion Procedure 110

5.6.2 Dealing with No-Match Conflicts 111

5.6.3 Dealing with Data Conflicts 113

5.6.4 Dealing with Control Conflicts 114

5.6.5 Satisfiability Procedure 115

5.6.6 Example 118

5.7 Summary 122

6 Partitioned Partial Scan Testing **123**

6.1 Introduction 123

6.2 Output-Based Partitioning 125

6.3 Switch-Based Partitioning 129

6.4 Size-Based Partitioning 134

6.5 Global Partitioning Strategy 137

6.6 Summary 138

7 Test Scheduling **140**

7.1 Introduction 140

7.2 The Test Scheduling Problem 141

7.2.1 Test Application with Multiple Scan Chains 141

7.2.2 Kernel Relationships 144

7.2.2.1 Incompatible Kernels 145

7.2.2.2 Dependent Kernels 145

7.2.3 Modeling Test Relationships 146

7.2.4 The No-Dependence Scheduling Problem 150

7.3 General Test Scheduling Algorithm 151

7.3.1 Terminology 152

7.3.2 Incremental Scheduling 153

7.3.3 Optimal Scheduling 159

7.3.4 Discussion 163

7.4 Summary 164

8 Scan Path Chaining **165**

8.1 The Chaining Problem 165

8.2 Test Application in Fully Compatible Designs 167

8.2.1 Combined Test 167

8.2.2 Separate Test 169

8.2.3 Overlapped Test 170

8.2.4 Comparison 171

8.3 Single Chain in Fully Compatible Design 173

8.3.1 Flip-Flop Position Ranges 174

8.3.2 Single Chain Algorithm 181

8.3.3 Non-Ideal Solutions 183
8.3.4 A Simplifying Transformation 186
8.3.5 Case Study 186
8.4 Multiple Chains for Two Compatible Kernels 189
8.4.1 Modeling the Problem 190
8.4.2 Problem Characteristics 192
8.4.3 Constructing Optimal Chains 198
8.4.3.1 Nonlinear Problem Formulation 198
8.4.3.2 Linearizing the Problem 200
8.4.4 Discussion 201
8.4.5 Experimental Results 202
8.5 The Rest of the Iceberg 205

9 Conclusion **207**
9.1 Partial Scan Design 210
9.2 Partitioning 213
9.3 Test Scheduling 215
9.4 Scan Path Chaining 216

Reference List **218**

List Of Tables

3.1 Comparison of full scan and BALLAST partial scan. 55

4.1 Relationships among inputs and outputs. 66

8.1 Results of multiple scan path chaining for circuit with $l_1 = 20$, $l_2 = 80$, $n_1 = 15$, and $(n_2 + n_{12}) = 10$. 203

8.2 Average saving in test time for different circuit examples over various numbers of scan chains. 204

List Of Figures

1.1 General synchronous circuit. (a) Huffman model, (b) circuit with scan path. 3

1.2 Storage element designs for various scan techniques. 4

1.3 Example of use of switching elements for partial scan. 9

3.1 Illustration of BALLAST methodology. (a) Synchronous circuit, (b) partial scan design of (a). 21

3.2 Example of topology graph. 23

3.3 (a) Kernel of Figure 3.1(b); (b) combinational equivalent of (a). 25

3.4 Additional illustrations of the methodology. (a) Example with HOLD mode; (b) kernel of (a); (c) example with unbalanced paths; (d) kernel of (c). 27

3.5 (a) Test for e stuck-at-1, (b) transformed test with no HOLD operations. 34

3.6 Illustration of balance procedure. (a) Original topology graph, (b) balanced topology graph. (Bold arcs represent HOLD registers.) . . . 45

3.7 Various scan design solutions for the same circuit. 46

3.8 Example of scan path implementation. 49

3.9 Organization of scan path into groups of scan path registers. 50

3.10 Modeling FF functional modes. (a) FF connecting two clouds, (b) model of RESET operation, (c) model of PRESET operation, (d) model of HOLD operation. 54

3.11 Combinational equivalent for detecting HOLD faults. 55

4.1 Example of ACYST. (a) Partial scan design, (b) acyclic kernel, (c) test generation model. 61

4.2 Example of acyclic kernel. 65

4.3 Construction of schedule constraint graph. (a) Constraints for (H, D) only; (b) constraints for (H, D) and (A, D). 68

4.4 Basic test generation model. (a) Acyclic structure; (b) basic TGM. . 73

4.5 TGM for balanced structure. (a) Balanced structure, (b) combinational equivalent. 74

4.6 Condensation of test generation model. (a) Step 1, (b) step 2. 76

5.1 General form of a switch. (a) MUX, (b) bus. 81

5.2 Generalized topology graph. (a) Circuit showing labels, (b) GTG with ACLUs as nodes. 84

5.3 SB-structure example. (a) SB-structure, (b) combinational equivalent. 87

5.4 Transformation of the GTG for feedback register analysis. (a) Circuit, (b) GTG G, (c) transformed GTG G_F. 94

5.5 Illustration of **checkSW** procedure. 96

5.6 Illustration of **balanceSW** where no finite mincut exists. 100

5.7 Illustration of kernels with I-paths. (a) Circuit with two kernels, (b) test plan for K1, (c) test plan for K2. 103

5.8 Use of I-paths in reducing partial scan overheads. (a) Circuit, (b) test plan. 105

5.9 Relationship between maximal (K_{MAX}) and minimal (K_{MIN}) kernels. 106

5.10 No-match situation. (a) Kernels K1, K2; (b) test plan for K2. 108

5.11 Unsatisfiable kernel due to data conflict. (a) Circuit, (b) test plan showing conflict. 108

5.12 Relationship between maximal (K_{MAX}), minimal (K_{MIN}) and minimal satisfiable (K_{SAT}) kernels. 110

5.13 Resolving a no-match conflict for K1 in Figure 5.10. 112

5.14 Schematic illustration of primary data conflicts. (a) Input side of kernel, (b) output side of kernel. 114

5.15 Kernel minimization: I-paths and minimal kernel. 119

5.16 Kernel minimization: I-paths and minimal satisfiable kernel. 121

6.1 Subdividing a kernel by output-based partitioning. 126

6.2 Overlapping kernels. (a) Low overlap, (b) high overlap. 128

6.3 Subdividing a kernel by switch-based partitioning. 130

6.4 Illustration of switch-based partitioning procedure. (a) Original kernel K, (b) state space, (c) kernels generated, (d) combinational equivalents. 132

6.5 Subdividing a kernel by size-based partitioning. 135

7.1 Example of partitioned testing. (a) Partitioned kernels, (b) test relationship graph. 142

7.2 TRG example to illustrate dependence groups. 148

7.3 Illustration of incremental scheduling: (a) partial schedule S', (b) incremented schedule S'' without interruptions, (c) incremented schedule S'' with interruption. 155

7.4 Implicit enumeration search space for generating a schedule. 160

7.5 Schedules generated by the search algorithm. (a) Schedule for (A, B, C) as well as (B, A, C); (b) schedule for (C, B, A). 161

8.1 Circuit example for illustrating different test schemes. 168

8.2 Example of single scan path chaining problem. 174

8.3 Ranges for placement of scan FFs. (a) For session TS_3, (b) for session TS_2, (c) for session TS_1, (d) combined ranges. 177

8.4 Illustration of computation of minimum session cycle. (a) Case 1, (b) case 2, (c) case 3. 179

8.5 Example of single scan chain ordering. 183

8.6 Example of single scan path chaining with empty ideal range for a register. 184

8.7 Example of single scan path chaining with no perfect matching. . . . 185

8.8 Case study for single scan chain. (a) Schematic description of circuit, (b) optimal scan chain ordering. 188

8.9 Circuit model for two-kernel multiple scan chain study. 191

8.10 Optimal chaining solutions. (a) Original configuration, (b) in region A, (c) in region B, (d) in intersection region. 194

8.11 Venn diagram of search space for multiple chain design problem. . . . 195

8.12 Decrease in overall test time with multiple scan chains. 204

9.1 SIESTA 1.0 system architecture. 208
9.2 SIESTA user interface. 209
9.3 The partial scan design space. 212

Abstract

Serial scan design is an approach to design for testability that can greatly reduce the cost of test generation for sequential circuits. It represents a class of techniques in which the storage elements are connected together into a continuous shift register chain in test mode, with the ends of the chain connected to I/O pins. This allows the storage elements to be fully controlled and fully observed, simplifying the test problem and reducing it to that of testing a combinational structure. Despite its benefits, serial scan design is often unattractive to circuit designers because of overheads in chip area, performance, pin count, and/or test time. Further, the traditional form of serial scan design is fairly rigid and does not provide flexibility in meeting specific design goals. This thesis presents an integrated scan design methodology called SIESTA that uses a range of strategies to reduce scan design costs. The concept of *partial scan*, in which only a subset of the circuit's storage elements are included in the scan path, is used to reduce area overhead and performance costs. Techniques are presented for selecting scan storage elements such that the overheads can be minimized while ensuring that combinational test generation is sufficient for the resulting design. *Partitioning* for test is employed to help reduce test generation and test application costs. Algorithms are presented for efficiently *scheduling tests* for various circuit partitions so as to minimize the overall circuit test time. A study of the *scan path chaining* problem, for both single and multiple chain designs, is also presented. The results demonstrate that an optimal configuration of multiple scan chains for minimum test time may actually have scan chains of unequal lengths. Given a circuit under design, the SIESTA methodology is able to apply different strategies to generate a range of testable design solutions that trade off different design costs against each other. Thus it can help meet the specific goals and constraints on the circuit.

Chapter 1

Introduction

"If you would know what nobody knows, read what everybody reads, just one year afterwards."

—Ralph W. Emerson

The objective of the research presented here is to develop an integrated scan design system. Various scan design schemes have been developed previously but implementing them optimally has remained an open problem. In the following sections we discuss the need for design for testability in general, study the well-known serial scan DFT techniques, and explain why an integrated scan system is required.

1.1 Design for Testability

Due to the rapid increase in the density of digital ICs, the amount of logic within a single chip has become extremely high. This makes the chips hard to test since there is little access to the internal circuit elements except through the I/O pins. Automatic test pattern generation (ATPG) is a computation-intensive problem even for combinational circuits. For general sequential circuits the test generation problem is almost intractable due to the difficulty of bringing the circuit to an arbitrary state.

Design-for-testability (DFT) techniques [1] are intended to reduce the difficulty of test generation by providing increased access to the internal elements of a

circuit, particularly the storage elements. **Ad hoc DFT techniques**, as the name implies, provide guidelines for circuit designers on how to increase controllability and observability, which are the two key factors related to test generation. Some of these techniques dictate that special test points be added to parts of a circuit that are difficult to access. On the other hand, **structured DFT techniques** provide well-defined design rules and are usually associated with a specific test methodology such as deterministic testing, exhaustive testing or pseudorandom testing. Examples of structured DFT techniques are Built-In Logic Block Observation (BILBO) [2] and Level-Sensitive Scan Design (LSSD) [3].

BILBO is an example of a *fully built-in* DFT technique in which test data is generated and analyzed on the chip, at the cost of high overhead. LSSD is a type of *serial scan* technique in which test data is stored off the chip, which means that a large fraction of the test time is taken up in moving data on and off the chip. Both techniques have their advantages and disadvantages which will not be discussed here. The Testable Design Expert System (TDES) [4] was developed to handle the tradeoffs among various techniques by using a common model for describing such techniques. In this research, however, we focus on serial scan design techniques and their particular implementation issues.

1.2 Serial Scan Design

Serial scan design represents a range of DFT techniques having the distinguishing characteristic that test data is stored outside the circuit and is shifted in and out of it serially. This is normally achieved by connecting some or all of the internal storage elements (latches or flip-flops) into a shift register when in the test mode, and providing access to the end points of the shift register through I/O pins. This shift register is known as a **scan path**. Instead of a single scan path, multiple scan paths may be used, with their end points connected to separate pins or multiplexed to share the same pins.

A generic scan architecture is illustrated in Figure 1.1. Figure 1.1(a) shows the Huffman model representation of a general synchronous sequential circuit. It

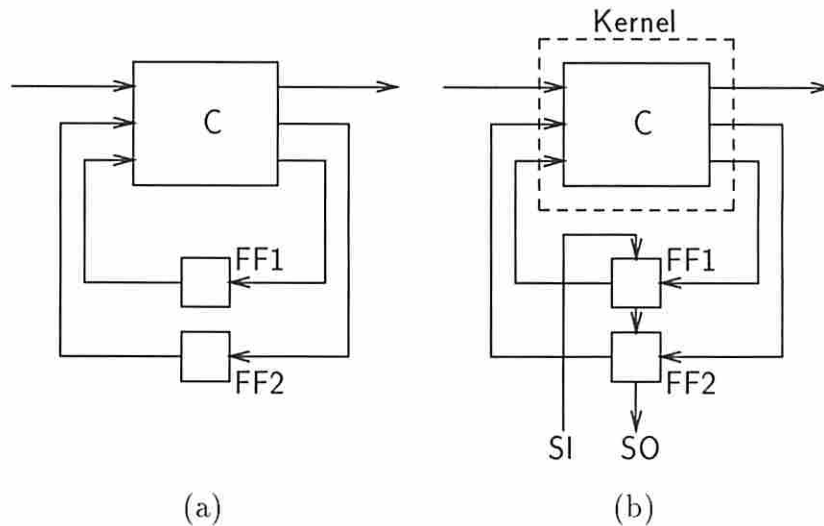
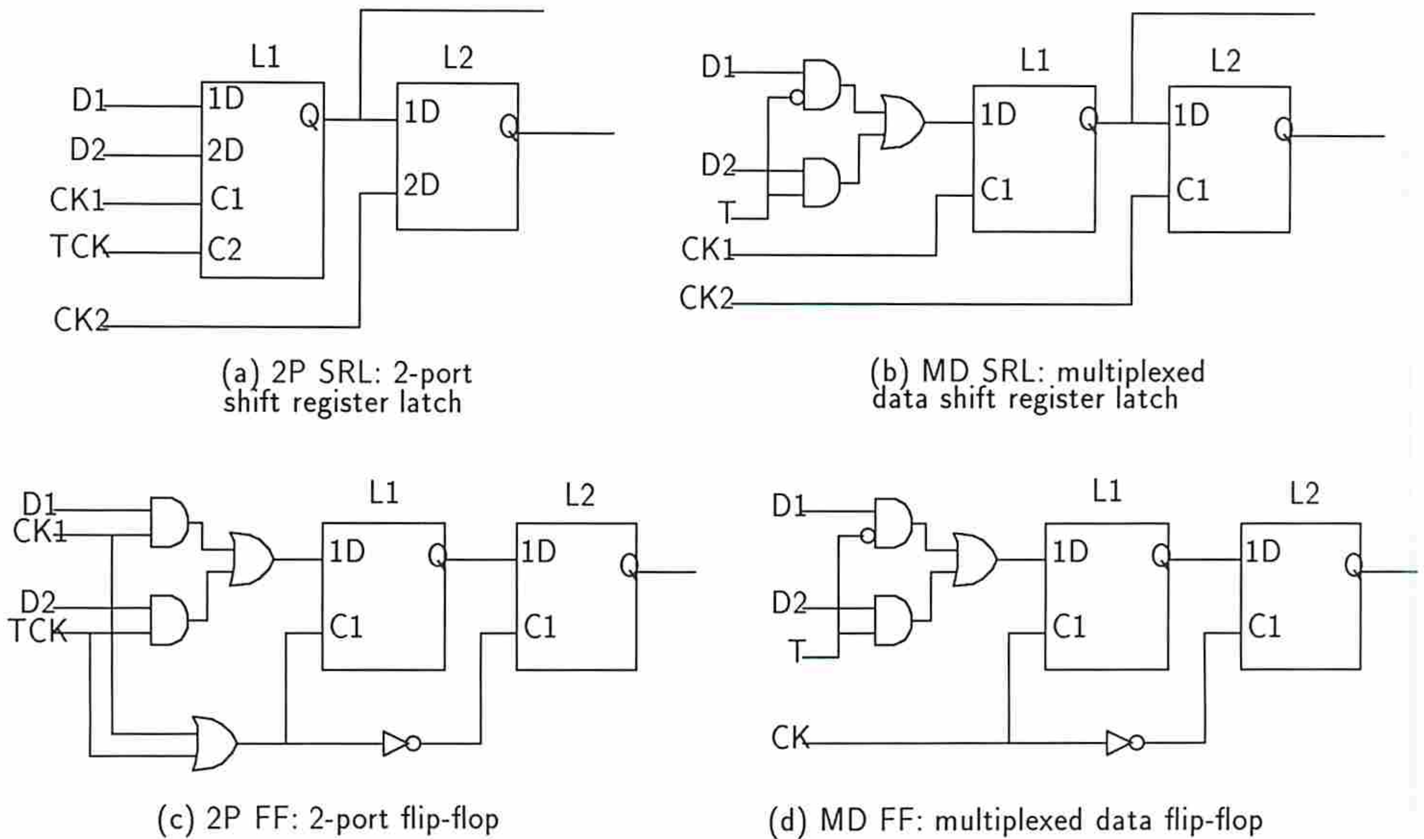


Figure 1.1: General synchronous circuit. (a) Huffman model, (b) circuit with scan path.

consists of a block of combinational logic (which may or may not be fully connected) along with a set of storage elements connected in feedback loops. Figure 1.1(b) shows a modified circuit in which all the storage elements can be configured into a scan path in test mode. This architecture is known as *full scan design*, since all storage elements in the circuit are included in the scan path. The scan path can be accessed via *scan-in* and *scan-out* pins. At least one additional *test control* signal, which is not shown, is required for distinguishing between the test mode and the normal operation mode. Thus three additional I/O pins are required, irrespective of the type of scan design used. It is possible to multiplex the scan-in and scan-out pins (but not the test control pin) with existing pins used for normal system operation, at the cost of some additional multiplexing logic.

1.2.1 Circuit Modification

Each storage element in the scan path needs to be modified so that in the test mode it can shift data serially. A large number of storage element designs have been proposed [3, 5, 6, 7]. The various designs can usually be characterized based on the type of storage elements they use. There are three basic types [8]: **single latches, double latches** (master-slave latches), and **flip-flops**. For each type, one or more of



(a) 2P SRL: 2-port shift register latch

(b) MD SRL: multiplexed data shift register latch

(c) 2P FF: 2-port flip-flop

(d) MD FF: multiplexed data flip-flop

Figure 1.2: Storage element designs for various scan techniques.

the storage element types in Figure 1.2 may be applicable. The 2PSRL and MDSRL are applicable to latch-based designs; the 2PFF and MDFF are applicable to FF-based designs. In each storage element design, L1 and L2 are latches; D1 and D2 are the system and test data inputs respectively; CK, CK1, CK2 are system clocks. In Figures 1.2(b) and 1.2(d), T is a test mode control signal which determines which of the data inputs is read in. In Figures 1.2(a) and 1.2(c), TCK is a test clock signal which enables the test data to be loaded in when a pulse is applied on it. All designs may have an optional LOAD ENABLE control signal LE (not shown).

All circuits constructed using one of the above scan path storage element (SPSE) designs have two basic modes of operation. In the *normal mode* the circuit carries out its normal system function. In the *test mode* the test mode control signal or the test clock signal (whichever is applicable) is activated so that the SPSEs behave collectively as a serial shift register.

1.2.2 Test Generation and Application

A set of test patterns for a scan-testable circuit can be obtained by removing the SPSEs from it and carrying out ATPG on the remainder of the circuit. Assuming that all storage elements are in the scan path, the remainder of the circuit is fully combinational. Thus the complexity of ATPG is significantly reduced compared to that for the original sequential circuit. Note that all the inputs and outputs of the combinational portion of the circuit are accessible either as primary I/O or through SPSEs. Given the set of test patterns for the combinational logic, they can be applied to the circuit using the following test plan. Assume that the scan path consists of k SPSEs and that N test vectors need to be applied.

1. Keep the system in the test mode for k clock cycles and shift the appropriate portion of the first test pattern into the scan path.
2. Repeat N times:
   (a) Keep the system in the normal mode for 1 clock cycle while the appropriate portion of the current test pattern is applied at the circuit primary inputs. At the end of the clock cycle, sample the values at the circuit primary outputs.
   (b) Keep the system in the test mode for k clock cycles and shift out the test result in the scan path; simultaneously, shift in the appropriate portion of the next test pattern. □

1.3 Scan Design Costs

The scan technique described above requires that the circuit storage elements be modified into SPSEs such that they can be connected into a scan path in the test mode. The result of these modifications affect the quality of the design in various ways. Some of these effects are described below.

Test generation effort Clearly the problem of testing the circuit is reduced to that of testing a combinational circuit, since all inputs and outputs of the combinational portion of a scan-testable circuit are fully accessible via the scan path. Despite recent advances in test generation techniques for sequential circuits, the problem of ATPG for a combinational circuit is still orders of magnitude simpler than that for a sequential circuit of comparable size. This implies that the test generation effort required for the scan-testable version is orders of magnitude lower than that for the original sequential circuit. The reduction in test generation cost is the major achievement of the serial scan design methodology, which makes it actually feasible to obtain comprehensive test sets for complex circuits, and achieve a high degree of confidence in their correct operation. However, depending on the manner in which the serial scan approach is used, the benefit is achieved at the cost of certain overheads described below.

Area overhead The modification of the storage elements to make them scannable causes an increase in the circuit area. The amount of the increase depends on the type of scan design used and the number of storage elements to be connected into the scan path. The area overhead often makes serial scan design unattractive to designers because it reduces the space available for functional logic on a chip. If on the other hand the total size of a chip is allowed to increase, the yield (i.e., the percentage of manufactured chips that are fault-free) of the circuit may decrease.

Performance degradation The modification of a storage element to make it scannable introduces an additional delay at its input of approximately two gate delays, even in normal system operation. In a traditional full scan design, every storage element is modified; hence the system clock cycle may need to be extended by this amount, causing a degradation in the circuit's performance. In the competitive integrated circuit market, this could be a serious liability and, along with the area overhead cost, sometimes discourages designers from using the scan design approach. Clearly, if storage elements that lie in critical timing paths could be excluded from the scan path, the overall performance of the circuit could be made more acceptable; this fact will be exploited in the work presented here.

Test time In a full scan circuit each test applied to the circuit consists of a single pattern fed to the scan path rather than a complex initialization sequence. However, the scan path can only be accessed serially. This makes the time to apply a given pattern proportional to the number of storage elements in the scan path. Since a typical full scan design may have hundreds of storage elements in the scan path, this may lead to hundreds of of clock cycles to apply a test for a given target fault. A high overall test time for a design can be expensive in a production environment, where automatic test equipment is usually costly and in high demand, and in maintenance testing, where a system may need to be shut down for the duration of a test.

I/O pin count The scan designs described previously require three I/O pins to be added to the circuit for scan-in, scan-out, and the test mode control signal or test clock (whichever is applicable). The first two can actually be multiplexed with existing system I/O, thus trading off I/O pins with multiplexer area.

In summary, scan design leads to lower test generation effort, at the cost of higher area, extra I/O pins, degradation in system performance, and a high test time to apply each test. Below we discuss different ways in which these costs can be reduced.

1.4 Reducing Scan Design Costs

Due to the modern thrust for high density of transistors on ICs and high operating speed, the overheads due to scan design can be expensive and make scan design unattractive to designers. However, the overheads can be reduced by making use of a range of design options which are described below.

Partial Scan Scan design techniques in which all circuit storage elements are included in a scan path are known as *full scan* techniques. In *partial scan*, only a subset of the storage elements are made scannable. This leads to reduced area overhead and possibly reduced test time. Performance degradation can also be avoided by keeping registers in critical paths out of the scan path. Most partial scan

techniques achieve these benefits at the expense of lower fault coverage or higher test generation effort [9, 10, 11]. However, this research has yielded techniques to achieve the benefits of partial scan without incurring these expenses; these techniques will be described in later chapters.

Partitioning In traditional scan techniques, test patterns are generated for the circuit as a whole, hence each pattern needs to be shifted into the whole scan path. If the circuit is partitioned suitably and individual partitions are tested separately, only a part of the scan path may need to be accessed for each pattern. In some cases this may lead to a reduction in the overall test time. Another potential benefit is that the overall test generation cost may be reduced since it increases rapidly with the size of the circuit under test.

Use of Switching Elements Circuit elements such as multiplexers (MUXes) and demultiplexers (DEMUXes), which are used as "switches," have the ability to transfer data unchanged from input to output under certain control inputs. In traditional testing such elements are treated as random combinational logic elements. By partitioning the circuit appropriately, it may be possible to make use of the switching elements in such a manner that the same scan path storage elements are used to test different partitions at different times. Thus partial scan can be used without loss in fault coverage within any partition. This type of test is illustrated in Figure 1.3, where R1 can be used to provide test patterns to either K1 or K2 by appropriately controlling MUX. Separate test sessions are required to test K1 and K2 since R1 cannot supply the required patterns to both simultaneously. Note that some registers and switching elements may need to be tested separately; for example, R2 and MUX in Figure 1.3 do not get fully exercised in all their operation modes during the test of the two partitions.

Multiple Scan Path Chains The bulk of the test time for a scan testable circuit is consumed in shifting test data serially in and out of the scan path. This time can be reduced by using several scan path chains and scanning them in parallel. Thus the test time is reduced at the expense of two I/O pins per scan chain.

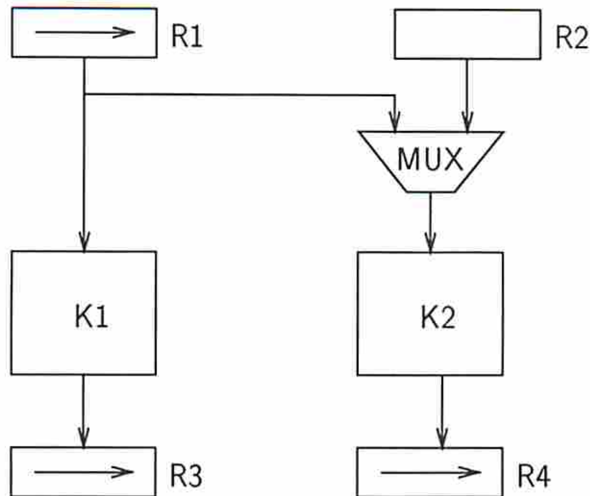


Figure 1.3: Example of use of switching elements for partial scan.

Given a circuit to be made testable, a multi-dimensional space of scan design solutions can be generated by using different combinations of the various scan design options described above with different parameters. Each design in the space is characterized by its test generation effort, area overhead, test time, test storage, I/O pin overhead and performance degradation. The problem of scan design can thus be stated as follows: given a set of constraints on the various design costs, use a combination of the various design options described above to obtain a design in the solution space that satisfies the constraints.

In the next chapter we look briefly at previous attempts to use one or more of the scan design options described above in an automated design system. In Chapter 3 a new approach to partial scan, which reduces the cost due to full scan but retains the benefit of combinational test generation, is presented. This approach, which is implemented in a subsystem called BALLAST, is based on defining a class of easily testable sequential structures having a "balanced" property. In Chapters 4 and 5 this approach is extended in two different ways. In the former, some of the restrictions of BALLAST are relaxed, leading to a further decrease in the scan design overheads at the expense of some increase in the test generation complexity. In the latter, any multiplexers and/or buses present in the circuit are used during test so as to improve the controllability and observability of the circuit. Chapter 6 studies different ways of partitioning the circuit in order to reduce overall test costs, and a

procedure for scheduling tests for the various partitions is developed in Chapter 7. Chapter 8 discusses the problem of constructing single as well as multiple scan path chains with the objective of minimizing the overall test time. Chapter 9 lists some of the achievements of this dissertation and discusses some open problems for future work.

Chapter 2

Background

"The merit belongs to the beginner should his successor do even better."
—Egyptian proverb

In this chapter we study previous approaches to the problem of managing costs and tradeoffs in scan-based design. Each approach will be classified based on the type of option it uses primarily—partial scan, partitioning, or multiple scan chains.

2.1 Partitioning Approaches

Four approaches to partitioning for test are described here. The first identifies independent connected regions of combinational logic. The second uses multiplexers to gain access to various partitions. The third uses the functionality of the logic elements to set up high-level sensitized paths to and from internal partitions. The fourth uses shift registers to gain access to the partitions and to isolate them from each other.

Cloud-Based Partitioning [12] In a full scan design, the combinational logic under test in the circuit can be subdivided into distinct regions of connected combinational logic called *clouds*. Each cloud is a maximal region of connected combinational logic that has either primary inputs or scan path storage elements at its inputs, and either primary outputs or scan path storage elements at its outputs. The CRETE system [12] takes in a hierarchical circuit description, and reorganizes the hierarchy to identify the clouds. ATPG can then be carried out on each cloud separately and the test sets can be merged later. This helps to reduce the maximum size of the circuit processed by ATPG and also leads to reduced overall test time. If any cloud is itself very large, the other types of partitioning listed below can be applied to subdivide it further.

Multiplexer Partitioning [13] This technique was actually proposed for the case where a combinational circuit was to be tested exhaustively. Multiplexers are inserted in the circuit and connected such that during the test of a partition, all its embedded inputs and outputs are accessible via primary inputs and outputs of the circuit. To apply this technique in serial scan design, the outputs and inputs of the scan path storage elements can be treated as primary inputs and primary outputs respectively, since they are fully accessible using the scan path. In general this technique can be used to reduce the size of the individual partitions under test and to achieve controllability and observability of internal wires. However, the multiplexers added for test can significantly increase the area overhead and may affect the speed of operation of the circuit.

I-Mode–Based Partitioning [4, 13, 14] In some circuits, partitioning can also be achieved by appropriately setting the required control inputs of certain partitions (called *data transporters* in [14]), giving them a transparent behavior (called an *I-mode* in [4]) so that they can trasmit test data to and from other partitions that are under test. For example, most ALUs can be made transparent to data at one input using some combination at the control inputs. A data transfer path set up in a circuit by a series of such partitions with I-modes enabled is called an *I-path*

[4]. I-paths can be used to obtain high controllability and observability of internal circuit elements with a low hardware cost.

Functional Partitioning [15] In this technique the circuit is decomposed into subunits on a functional basis so that functional test vectors can be used. The storage elements associated with each subunit are connected into a separate scan path. To provide access to signals that connect different subunits, either multiplexer partitioning is used (as described above) or a special *feedback shift register* (FSR) is used for each unit. The FSR is inserted in the interconnection lines so that in test mode the subunit under test can be provided with test patterns by scanning data into the FSR. Only one subunit is tested at a time, and all the scan paths share the same scan in and scan out pins through multiplexers and demultiplexers. An additional *partitioning control shift register* (PCSR) with its own scan in and scan out pins is used to configure the circuit during test so that the desired subunit can be accessed. This approach has a high overhead due to the large number of shift registers required for isolating the subunits; however, it may be feasible if good functional tests are available for the subunits.

2.2 Partial Scan Approaches

Three scan design approaches based on partial scan are presented here. The first uses heuristics based on testability estimates to select storage elements for the scan path. The second attempts to cover as many faults as possible out of a target fault set using combinational TPG. The third uses an analysis of the state transitions to generate tests and to help construct the scan path.

Partial Scan based on Testability Evaluation [9] In this approach only hard-to-test storage elements are included in the scan path. The testability of each circuit node is evaluated using the Sandia Controllability and Observability Analysis Program (SCOAP). Each node is assigned six values that represent the difficulty of controlling and observing it. Higher values indicate nodes that are harder to test.

Using the SCOAP values, hard-to-test storage elements are selected and connected into a scan path. Since these nodes can be easily initialized and observed, the overall ATPG cost for the circuit decreases. The number of storage elements in the scan path can be based on the permissible area overhead or on the desired levels of the SCOAP values in the circuit. According to experimental results the greatest incremental cost reduction can be achieved by including 15% to 30% of the hardest-to-test storage elements in the scan path.

This technique is simple to implement and can be used to trade off testability improvement versus overhead. However, it depends on SCOAP for identifying hard-to-control circuit nodes, and SCOAP estimates may not be very accurate for some circuits. In this approach the portion of the circuit consisting of all circuit logic excluding the scan path FFs, called the **kernel**, is sequential; hence a sequential ATPG program is required.

Partial Scan based on Target Fault Set [10] This approach requires initial functional testing to be carried out with high fault coverage (typically at least 70%). The partial scan analysis is carried out with only the set of remaining untested faults being targeted. Two alternative algorithms are proposed. In the first, *all* possible test vectors are initially generated for each target fault using a modified version of the PODEM program; for each test vector the set of FFs that are required to be in the scan path is determined. A minimal subset of FFs is then selected such that all (or the required number of) target faults can be detected. In the second algorithm only one test vector is generated for each target fault, but the distance heuristic of PODEM is used to minimize the number of additional FFs that need to be scanned. Each time a test vector is generated the distance heuristics are updated so that the FFs already in the scan path can be used for the remaining target faults at no cost. According to experimental results using this approach, 95% fault coverage can be achieved with less than 65% of the FFs in the scan path.

One advantage of this approach compared to the previous one is that the kernel is combinational, hence it requires only combinational ATPG. However, there is a high processing overhead, especially in the first of the two alternative algorithms,

which requires *all* test patterns to be enumerated for each fault. Another disadvantage is that prior functional testing is a prerequisite for this approach to be useful in getting high fault coverage.

Partial Scan based on State Transitions [11] This technique determines a minimal subset of storage elements to be made scannable such that all faults in the circuit are easily testable using multiple-pattern test sequences. A test sequence for a given fault consists of two parts: a justification sequence, which must begin with the circuit in a fixed reset state, and a propagation sequence, which must have a limited length. In case of justification or propagation failure, the algorithm uses the state transition graph of the circuit to determine all minimal subsets of storage elements which, if made scannable, allow the fault to be detected. These minimal subsets are determined for all faults under consideration. Heuristics are then used to select the set of storage elements in the scan path such that at least one feasible test sequence exists for each fault.

This technique requires a state transition table for the circuit to be available. It basically searches for a near-optimal solution at the cost of processing a large amount of information, hence it has a high computation complexity and may be impractical for large circuits.

Partial Scan based on Circuit Structure [16, 17] These techniques analyze the interconnection of combinational logic with FFs, and select scan FFs such that the resulting kernel is acyclic or close to acyclic.

Cheng et al. [16] make the assumption that the difficulty of test generation increases with the *length* of the cycles in the circuit, where the length of a cycle is the number of FFs it passes through. They use a heuristic to select a minimal number of scan FFs so as to break all cycles except *self-loops*, i.e., cycles consisting of combinational logic and a single FF. The argument is that self-loops do not contribute substantially to the sequential behavior of a circuit. For example, test generation for a subcircuit containing a single self-loop can be carried out in at most two time frames. However, this argument does not necessarily extend to the circuit as a whole, and the authors do not provide any conclusive evidence that a collection

of interacting self-loops is easier to test than a single long loop with the same number of FFs.

The technique of Kunzmann [17] is based on ideas that are somewhat similar to those seen in Chapter 3 of this work but were developed independently. The approach is to select scan FFs so as to result in a kernel that is "equivalent" (or "balanced", according to our terminology). Such a kernel is acyclic, and further, if any pair of circuit nodes is connected by more than one path, all such paths must have equal numbers of FFs. During test mode all non-scan FFs are placed in a bypass mode so that the kernel behaves as a combinational structure. By selecting FFs in the manner described above, only combinational ATPG is required to obtain complete stuck-at tests for the resulting circuit. Kunzmann's technique has the disadvantage that it requires all storage elements, including non-scan FFs, to be modified in some way, which diminishes the advantage of using partial scan.

In general, structure-based partial scan approaches have the advantage that a high-level model of the circuit is sufficient, provided it captures the data dependencies among the FFs and the combinational logic. This helps to simplify the analysis for selecting scan FFs. Our analytical study in Chapters 3–5 will bring out the strong dependency of the ATPG cost on structural features such as cycles, unbalanced paths, and the location of any buses/multiplexers ("switches") present in the circuit. The results of this study will be utilized in a range of scan path methodologies that attempt to satisfy the limitations of the earlier approaches.

Recently an optimization-based approach to partial scan design has been proposed by Chickermane et al. [18]. They pose the problem of selecting scan FFs as an optimization problem with different objective functions. Given a cost function associated with converting each FF to a scan FF, and an upper bound on the cost of the design, the goal is to select a satisfactory set of scan FFs so as to maximize certain testability measures. The testability measures used include the sequential depth of the resulting kernel, the number of cycles, the estimated test sequence length, and SCOAP-based controllability/observability values. This study shows that provided a good estimator of the cost of scanning each FF is available, an optimization-based approach is feasible for partial scan design. The authors have

presented an estimation function based on a standard cell design style; however, estimating this cost in the general case is a complex problem.

2.3 Multiple Scan Chain Approaches

Two approaches to multiple scan path chaining are described in this section. The first attempts to construct scan paths of equal length while the second forms scan path chains that can run independent of each other.

Multiplexed Access Scan Testable (MAST) Design [19] In this technique the scan path is broken up into several equal-length scan path chain segments, and the scan in/out ports of each are multiplexed with existing system I/O pins. Thus the possible number of scan chains is bounded by the number of system I/O pins available, and all chains can be operated in parallel in test mode. With this setup the time required to access all the storage elements depends on the length of the longest scan chain. The MAST technique distributes the storage elements evenly among the scan chains; this minimizes the length of the longest scan chain. The equal-length scan chain approach is effective when the circuit is tested as a unit in a single test session. However, in Chapter 8 it will be shown that when the test for the circuit is carried out in a partitioned manner, relaxing the equal-length chain requirement can lead to more optimal results; i.e., the lowest overall test time may actually be obtained by using an unequal-length scan chain configuration.

DFTEXPERT Scan Path Design [14] Unlike the MAST approach, the DFTEXPERT approach assigns storage elements to scan path chains based on the interconnection patterns among the registers and data processors (DPs), the latter being blocks of random logic. The design process [14] is based on every DP having exactly one input port and one output port, hence it may not be easily extensible to general circuits. In DFTEXPERT, the circuit is partitioned into independent networks, each of which is characterized as either a *single-DP net*, a *chain net* or a *graph net*. A single-DP net consists of a DP with its driver and receiver registers

(i.e., registers that supply test patterns and load in the test results, respectively). A chain net consists of a series of alternating registers and DPs, in which the first register is a driver, the last register is a receiver, and the intermediate registers serve as both drivers and receivers. A graph net is a general net that does not belong to either of the first two types. The formation of scan chains in DFTEXPERT is guided by rules. The registers in single-DP and chain nets may be included in the same scan chain or in different scan chains operating in parallel. The registers in a graph net are always connected in the same scan chain. Nets of small size may be combined into groups based on an *affinity* measure between nets. The scan path chaining rules attempt to use a range of strategies in order to meet the constraints imposed on circuit overheads and on test time.

In DFTEXPERT, the ordering of registers in each scan chain is carried out by heuristics based on the number of DPs that make use of each register in the chain to provide or receive test data. However, the *test lengths* of the DPs (i.e., the number of test patterns required for each DP) are not taken into account. In Chapter 8 we will show how the relative test lengths of various DPs, when taken into account, can help in configuring multiple scan chains such that the overall test time is minimized.

Chapter 3

Partial Scan Design with Balanced Structures

"The greatest truths are the simplest. ..."
—Julius Charles Hare and Augustus William Hare, 'Guesses at Truth' (1827)

"I have made this letter longer than usual because I lack the time to make it shorter."
—Pascal

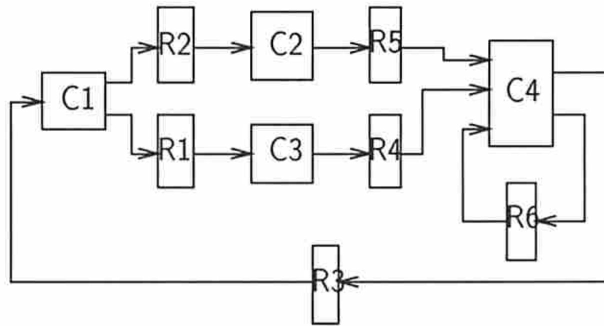
3.1 Introduction

Automatic test pattern generation (ATPG) for sequential circuits is generally considered to be an intractable problem that requires large amounts of computation even for circuits of moderate size. Full scan design techniques attempt to alleviate this problem by connecting all flip-flops (FFs) or latches into a scan path during test mode so that all these elements become easily controllable and observable. Thus in a circuit designed using full scan the portion of the circuit consisting of all circuit logic excluding the scan path FFs, which we shall refer to as the **kernel** of the circuit, is fully combinational. Due to the overhead in modifying the FFs, partial scan techniques [9] have been proposed in which only a subset of the FFs are included in the scan path. This implies that the kernel is itself sequential. Hence these techniques either require sequential ATPG [11, 16] or use combinational ATPG techniques without covering all faults in the kernel [10].

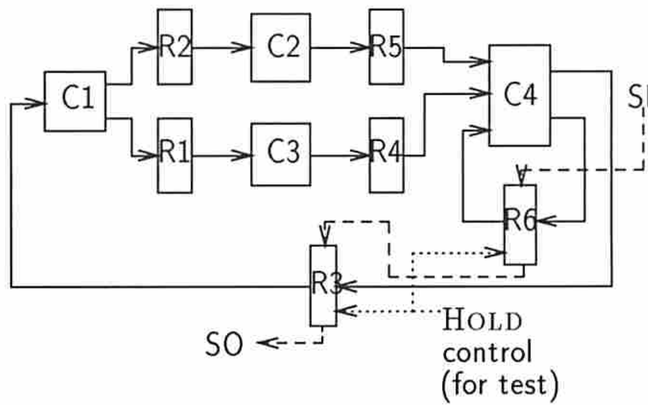
In this chapter we present an alternative partial scan methodology, BALLAST (BALAnced structure Scan Test), that requires only combinational ATPG and leads to complete coverage of all detectable single stuck-at (SSA) faults. Scan path FFs are selected in such a way that the resulting kernel belongs to a certain class of easily testable structures called *B-structures*. Test patterns for the kernel are obtained by treating it as being combinational, with FFs within it simply replaced by delayless wires. During test application, each pattern is *held constant* in the scan path for a fixed number of clock cycles before the output pattern of the kernel is sampled and shifted out.

The BALLAST methodology is illustrated using the circuit of Figure 3.1 consisting of six registers and four combinational logic blocks. The only registers in the scan path are R3 and R6. In this example the kernel has a pipelined structure. For the time being let us assume that the scan path registers are provided with the ability to hold data constant across consecutive clock cycles (later this requirement will be relaxed). Test patterns can be obtained by using combinatinal ATPG in the manner described above. To test the kernel, each test pattern is shifted into the scan path and held in it for two clock cycles before the kernel output is loaded back into the scan path and shifted out. We will prove that using this approach we can detect all faults in the kernel that can cause errors in the logical operation of the circuit. We shall also study the implications of "unbalanced" paths in the kernel with unequal delays, and of the presence of HOLD modes in non-scan registers for normal system operation.

A partial scan technique with some similarities to BALLAST was developed independently by Kunzmann [17]. This technique has several limitations which are addressed by BALLAST. First, in addition to modifying the storage elements in the scan path, even the non-scan storage elements need to be modified to make them transparent in test mode. This diminishes the advantage of using partial scan. Second, non-scan storage elements are passive during test (since they are bypassed) and do not get exercised. Third, the technique is not applicable if any storage elements have LOAD ENABLE control signals.



(a)



(b)

Figure 3.1: Illustration of BALLAST methodology. (a) Synchronous circuit, (b) partial scan design of (a).

3.2 Basic Circuit Model

In this work we consider only synchronous sequential circuits in which every cyclic path contains at least one clocked storage element. The storage elements are assumed to be flip-flops (FFs). The circuit may have any number of clocks. However, the FF clocks must all be controlled by primary input signals, and no clock signal may feed the data input of any FF either directly or through combinational logic. The FFs may have LOAD ENABLE control signals, with similar restrictions as for the clock signals. The FFs may also have RESET/PRESET controls provided they are present in all FFs and all are controlled by the same primary RESET/PRESET control signal.

In general a synchronous sequential circuit S consists of blocks of combinational logic connected with each other either directly or through registers. A **register** is defined as a collection of one or more FFs with all FFs driven by the same clock signal and controlled by the same mode control signal (if any). Any subset of the FFs in a register forms a valid register.

The set of registers in the circuit can be partitioned into two subsets based on the presence or absence of explicit LOAD ENABLE controls on the FFs comprising them. We define the **load set** L as the set of registers in the circuit whose FFs have no explicit load enable control; these registers always operate in the LOAD mode (in which data is read from the data input during every clock cycle). Similarly, we define the **hold set** H as the set of registers whose FFs have an explicit LOAD ENABLE control signal; these registers have two modes of operation: a HOLD mode (in which they retain their value across consecutive clock cycles) as well as a LOAD mode.

The combinational logic in S can be partitioned into **clouds**, where each cloud is a maximal region of connected combinational logic such that its inputs are either primary inputs or outputs of FFs and its outputs are either primary outputs or inputs to FFs. In Figure 3.1(a) each block of combinational logic C1, C2, C3, C4 represents a cloud. A **vacuous cloud** is a special type of cloud consisting of simple wires with no logic; a vacuous cloud is present wherever a primary input directly feeds an FF, or an FF directly feeds a primary output, or an FF directly feeds another FF.

Figure 3.2: Example of topology graph.

From the way in which clouds are defined it follows that no two clouds can be directly connected together; they must be separated by one or more registers. Further, each FF in a register must receive data from exactly one cloud and must feed exactly one cloud. We can therefore constrain the grouping of FFs into registers (by splitting registers where necessary) such that each register receives data from exactly one cloud and feeds exactly one cloud. Under this constraint the topology of the circuit can be modeled by a directed **topology graph** $G = (V, A, H, w)$ in which nodes in V represent clouds, each arc in A represents a connection between two clouds through a register, arcs in $H \subset A$ represent HOLD registers, and $w : A \rightarrow \mathcal{Z}^+$ (positive integers) defines the number of FFs in each register. We will use $w(a)$ to represent the cost of converting the register a into a scan path register. Figure 3.2 shows a typical topology graph; bold arcs represent HOLD registers and others represent LOAD registers.

3.3 B-Structures and their Properties

Let S be an arbitrary synchronous sequential circuit with topology graph $G = (V, A, H, w)$.

Definition: S is said to be a **balanced sequential structure (B-structure)** if:

1. G is acyclic;
2. $\forall v_1, v_2 \in V$, all directed paths (if any) from v_1 to v_2 are of equal length (this actually includes condition 1); and
3. $\forall h \in H$, if h is removed from G, the resulting graph is disconnected. □

The example of Figure 3.2 satisfies these conditions and is therefore a B-structure.

Given a balanced sequential structure S^B, we define its **combinational equivalent,** C^B as the combinational circuit formed by replacing each FF in every register in S^B by a wire (if the output of the register uses the Q output of the FF) or an inverter (if it uses the $\overline{Q}$ output of the FF) or both. Define the **depth**, d, of S^B as the longest directed path in its topology graph. Given an input pattern I applied to S^B, define the **single-pattern output** of S^B for I as the steady-state output of S^B when I is held constant at the inputs to S^B and all its registers are operated in LOAD mode for at least d clock cycles. Given some fault f in S^B, if the single-pattern outputs for I of the good and the faulty circuits are different, then I is a **single-pattern test** for f.

B-structures have two interesting properties which allow them to be used as kernels in a BALLAST partial scan design: (1) every detectable fault is single-pattern testable, and (2) a complete single-pattern test set for all detectable SSA faults can be derived using combinational test generation techniques. Both properties will be elaborated on and proved in Section 3.5. Next we present an overview of the proposed BALLAST methodology.

3.4 Scan Design Using B-Structures

When any register in a sequential circuit is included in a scan path it serves as a control and observation point for the rest of the circuit. In effect it becomes a primary output of the cloud feeding it and a primary input of the cloud it drives. Thus in our circuit model, the inclusion of a register in the circuit scan path corresponds to its removal from the topology graph of the circuit; the reduced topology graph represents the kernel, i.e., the portion of the circuit to be tested using the scan path.

The following is an outline of the BALLAST methodology.

1. Construct the topology graph G of the circuit as defined in Section 3.3.

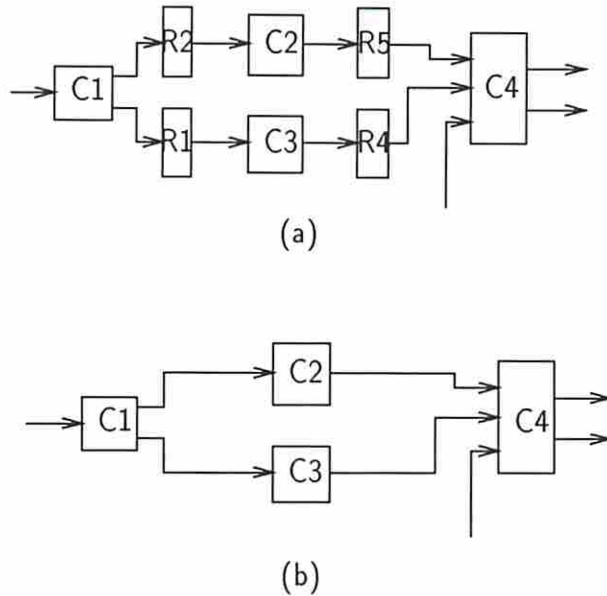


Figure 3.3: (a) Kernel of Figure 3.1(b); (b) combinational equivalent of (a).

2. Select a minimal cost set of arcs, R, to be removed from G such that the remaining topology graph is balanced. Arcs in R represent the registers that must be included in the scan path. Let S^B be the B-structure corresponding to the resulting topology graph which represents the kernel of the circuit. For the example of Figure 3.1 the kernel is the B-structure shown in Figure 3.3(a). Algorithms for implementing this step are presented in Section 3.6.

3. Determine the combinational equivalent C^B of the kernel S^B. The combinational equivalent of the kernel in Figure 3.3(a) is shown in Figure 3.3(b). Using traditional combinational ATPG, determine a complete test set T for C^B. Since S^B is balanced, T will constitute a complete single-pattern test set for all detectable faults in the kernel S^B. (See Section 3.5 for proof of correctness of this statement.)

4. Construct a scan path by appropriately ordering the registers in R and connecting them so that they are capable of (i) shifting test patterns in/out, (ii) holding a pattern constant at the kernel inputs for d clock cycles (where d is the depth of the B-structure comprising the kernel), and (iii) loading in the test results from the kernel. Requirement (ii) can be achieved by providing all scan registers with a HOLD mode as in Figure 3.1(b). In Section 3.7 we will

show how the need for all scan registers to have a HOLD mode can be partially removed.

Figure 3.4 shows two illustrations of the above methodology which are variations on the example of Figure 3.1. In Figure 3.4(a) the register R1 has a HOLD mode provided for normal system operation. Figure 3.4(b) shows the kernel when R1, R3 and R6 are made scannable; clearly it is a B-structure and hence a valid kernel. Figure 3.4(c) shows a second variation in which the two paths between C1 and C4 have unequal delays. By scanning R1, R3 and R6, one of the paths is broken and the resulting kernel in Figure 3.4(d) is again a B-structure.

Given a circuit designed in the manner described above, the test plan for applying a sequence of single-pattern tests to the circuit is as follows. N is the number of test patterns to be applied and l is the total number of FFs in the scan path.

1. *Operate all scan registers in the* SHIFT *mode for l clock cycles. (Scan in the first test pattern.)*
2. *Repeat N times:*
   (a) *Place all scan registers in the* HOLD *mode and all non-scan registers in the* LOAD *mode for d clock cycles. (This allows test data to propagate through the kernel.)*
   (b) *Operate all scan registers in the* LOAD *mode for 1 clock cycle. (Load the test result into the scan registers.)*
   (c) *Operate all scan registers in the* SHIFT *mode for l clock cycles. (Scan out the test result and scan in the next test pattern.)* □

The partial scan technique described in this section is applicable to general sequential circuits. It is particularly well suited to pipelined circuits with limited feedback and feedforward connections. On the other hand, it is not very effective for circuits in which there are few clouds or in which every storage element is connected in a feedback loop through a single cloud; in the latter case the technique reduces to full scan.

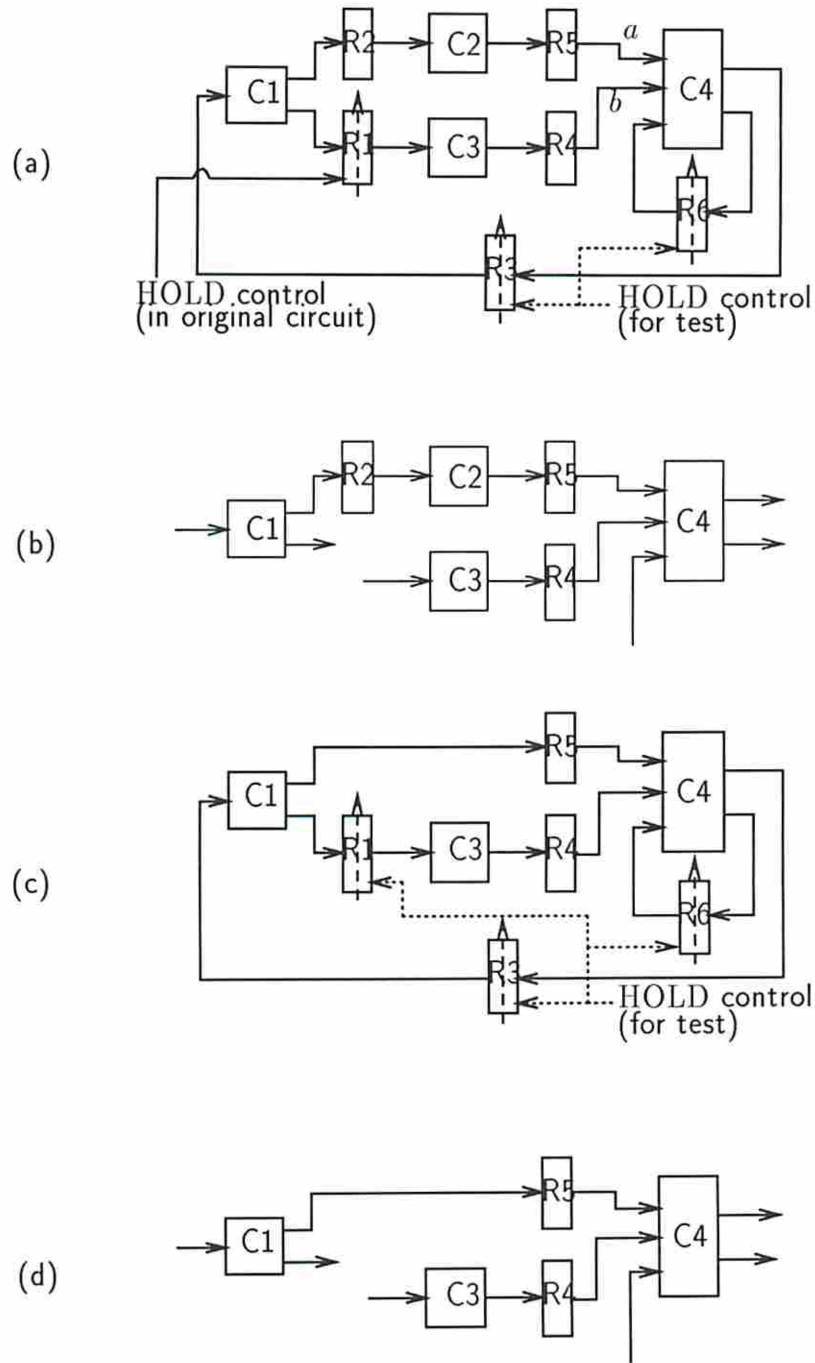


Figure 3.4: Additional illustrations of the methodology. (a) Example with HOLD mode; (b) kernel of (a); (c) example with unbalanced paths; (d) kernel of (c).

3.5 Proof of Correctness

In this section the testability properties of B-structures are studied with the objective of proving that any B-structure is a valid kernel in the BALLAST test methodology. We shall focus on the kernel of the circuit under consideration in isolation from the scan path. We shall treat all connections of the kernel from/to scan path registers as primary I/O lines of the kernel. The BALLAST methodology ensures that the kernel is a B-structure; this has the following implications. Despite the kernel being sequential, the fact that it is acyclic greatly simplifies the process of obtaining a test sequence [20, Section 3.5]. But this fact is of limited practical importance to us since arbitrary multiple-pattern test sequences cannot be applied to the kernel using a serial scan path. However, by restricting the kernel to being a B-structure, we shall demonstrate that single-pattern tests exist for all detectable faults in the kernel and that a complete test set of this type can be obtained using ordinary combinational ATPG on the combinational equivalent of the kernel.

3.5.1 Single-Pattern Testability

Let S^B be an arbitrary B-structure and C^B its combinational equivalent. The set of faults of a B-structure refers to the union of the sets of single stuck-at faults of the individual clouds. A stuck-at fault in a register can be considered to be equivalent to a stuck-at fault in one of the clouds adjacent to it. A *detectable* fault is one that can be detected by some sequence of test patterns.

By definition of the combinational equivalent, S^B can be viewed as the combinational circuit C^B with delays introduced appropriately within it. Since delays do not affect the steady-state output function of combinational logic, we have the following lemma.

Lemma 1 *For any input pattern I, the output of C^B and the single-pattern output of S^B are identical.*

Proof: By induction on the depth of S^B.

Let G be the topology graph of S^B and let its depth be d.

$\underline{d = 0:}$ If d is 0, S^B is combinational, and the statement is trivially true.

$\underline{d > 0:}$ Assume that the statement is true for $0 \leq d \leq n-1$, and consider $d = n$. Remove from G all "first-level" nodes, i.e., nodes that have no incoming arcs, and all corresponding first-level arcs. There must be at least one first-level node because G is acyclic. Let the resulting graph be G_1, representing a corresponding balanced structure S_1^B of depth $n-1$ with combinational equivalent C_1^B. When the input vector I is applied to S^B, all clouds corresponding to first-level nodes must settle at constant values within the first clock cycle; hence the values loaded in by the first-level registers must in fact be their final steady-state values. Thus after the first clock cycle, S_1^B receives a constant input pattern. Since the depth of S_1^B is $n-1$, and both C_1^B and S_1^B receive the same input pattern after the first clock cycle, the output of C_1^B and the single-pattern output of S_1^B must be identical, by induction. Thus the lemma is true for $d = n$. □

Note that the only property of B-structures used in the above lemma is the fact that the topology graph is acyclic, hence it is true of all acyclic structures, balanced or unbalanced.

Lemma 2 *Let f_S be a fault in S^B and let f_C be the corresponding fault in C^B. Then any test pattern t for f_C in C^B is a single-pattern test for f_S in S^B.*

Proof: Let C_f^B and S_f^B be the faulty circuits produced by f_C and f_S respectively. Since t detects f_C, the outputs of C^B and C_f^B must differ for input t. Due to Lemma 1, the single-pattern outputs of S^B and S_f^B must differ for input t. Thus t is a single-pattern test for f_S in S^B. □

Note that the above lemma does not prove that there is a single-pattern test for *every* detectable fault in S^B.

Theorem 1 *Every B-structure is fully testable for all* detectable *SSA faults using single-pattern tests.*

Proof Outline Let the B-structure S^B have depth d and let its combinational equivalent be C^B. Let f be a detectable SSA fault in S^B and let T be a test for f. T is in general a sequence of patterns to be applied to all data as well as control inputs of S^B. It is sufficient to prove that T can be transformed into a single-pattern test for f.

Let T cause the fault to be first detected at a particular output z of S^B. Consider each data input x_i of S^B (i.e., any input other than a control input for a register). Since S^B is balanced, all paths from x_i to z must have equal delays. (Note that since S^B is a B-structure, if there is a HOLD register in any of these paths then all the paths must pass through it, thus preserving equal delays.) This implies that the output value at z when the fault is detected depends on the value at x_i during at most one clock cycle, say τ_{x_i}. Hence if we transform T to T' such that for each input x_i, the value required at clock cycle τ_{x_i} in T is applied at every clock cycle in the sequence T', then T' must be a valid test for f. We can now transform T' into T'' by operating all HOLD registers in S^B in the LOAD mode throughout the test. Note that the final output values at z for T' and T'' respectively must be identical, since both can be simulated by applying the corresponding input pattern to C^B. It follows that the resulting sequence T'' must also detect f by producing an erroneous output at z. Hence any test sequence can be transformed into a single-pattern test. $\square$

In the above proof outline, the crucial step is the transformation of an arbitrary test sequence into a single-pattern test. Below a more complete proof is presented in which the complete transformation procedure is listed.

Proof It is sufficient to prove that given any detectable fault f in the balanced structure S^B, there exists a single-pattern test vector for f. We first present a formal proof and then illustrate the proof using an example.

Let $G = (V, A, H, w)$ be the topology graph of S^B. During any clock cycle t, let the *state* of S^B be defined by the tuple $G^t = (G, I^t, h^t, x^t)$, where $I^t(v), v \in V$ represents the input pattern applied at the circuit primary inputs (if any) of the cloud v during clock cycle t; $h^t(a), a \in H$ represents the mode signal of the HOLD register

a during clock cycle t; and $x^t(a), a \in A$ represents the logic value of the register a during clock cyle t. A 5-valued logic system is used. The possible logic values are $\{0, 1, D, \overline{D}, \times\}$, where D and $\overline{D}$ represent erroneous states due to a fault, and $\times$ represents an unspecified or *don't-care* value. Assume (without loss of generality) that for a HOLD register a, $h^t(a) = 0$ represents the LOAD mode and $h^t(a) = 1$ represents the HOLD mode.

Since f is detectable, some test sequence T for f must exist. Let T consist of a sequence of patterns applied to the circuit primary inputs feeding the clouds and to the control signals feeding the HOLD registers. Let the first pattern be applied at clock cycle 1 and let the fault be first observed at a circuit primary output at clock cycle m. The application of T to S^B can be fully described by the sequence of states $(G^1, G^2, \ldots, G^m)$ that the circuit experiences.

We shall now show that based on T it is possible to derive a single-pattern test for f. The following procedure transforms T into a single-pattern test T'.

procedure transform $(T = (G^1, \ldots, G^m))$:

1. Pick a node $v_0 \in V$ such that the corresponding logic in S^B has one or more outputs that are circuit primary outputs and at least one primary output has value D or $\overline{D}$ in state G^m.
2. Define a relation called *activation time*, $\alpha : V \rightarrow \{1, 2, \ldots, m\}$ where $\alpha(v)$ represents the clock cycle during which the node v is is actively involved in sensitizing or propagating the effect of the fault f; i.e., there is a functional dependency between the output of node v during clock cycle $\alpha(v)$ and the output of v_0 during clock cycle m. Note that $\alpha(v)$ need not be defined for all v.
   (It will be shown that the activation time of every node, if defined, must have a unique value.)
   Set $\alpha(v_0) \leftarrow m$.
3. Construct a set $\mathcal{F}$, representing a frontier of nodes being processed, and set $\mathcal{F} \leftarrow \{v_0\}$.
4. Repeat the following until $\mathcal{F} = \phi$:
   (a) Pick some $v \in \mathcal{F}$ having the highest activation time $\alpha(v)$, and remove it from $\mathcal{F}$.
   $k \leftarrow \alpha(v)$.

(b) For all incoming arcs a of v, do the following:

   i. $u \leftarrow$ source node of a;
      Add u to $\mathcal{F}$, with $\alpha(u) \leftarrow k-1$.

   ii. If $a \in H$ and $h^k(a) = 1$ then

      A. $t \leftarrow$ min. j such that $h^{j+1}(a) = h^{j+2}(a) = \ldots = h^k(a) = 1$;
         If there is no such j, skip steps B and C.
         (Note that in this case the value present in register a at time k is undefined, hence it cannot have an influence on the outcome of the test.)
         t represents the clock cycle during which node u is active, and register a loads the active data, in the original test plan.

      B. $\tau \leftarrow k - t =$ number of HOLD cycles of a in the current sequence.
         The next step eliminates the HOLD cycles while keeping the test valid.

      C. The removal of a from G must disconnect G into separate components, by definition of balanced structures. Let G_u be the component that contains u. Modify the test by delaying all electrical activity within G_u by τ clock cycles as follows.
         $\forall v \in V$ in G_u, $I^j(v) \leftarrow I^{j-\tau}(v)$, $\tau < j \leq k$;
         $\forall h \in H$ in G_u, $h^j(h) \leftarrow h^{j-\tau}(h)$, $\tau < j \leq k$;
         $\forall a \in A$ in G_u, $x^j(a) \leftarrow x^{j-\tau}(a)$, $\tau < j \leq k$;
         $h^k(a) \leftarrow 0$.
         It can easily be seen that the sequence of modified states $(G^1, \ldots, G^m)$ is still a valid test for f.

The preceding portion of the procedure transforms the test sequence such that in the final state sequence every register is in the LOAD mode at the time when the node driving it becomes active, and adjacent nodes are active during adjacent clock cycles. Note that the activation time $\alpha(v)$ of each node v depends only on its distance from the output node v_0. Since G is balanced, this implies that *for each node, there is a unique clock cycle during which it is active.*

5. Let $t_0 \leftarrow \min_v [\alpha(v)] =$ earliest activation time among nodes that have been assigned an activation time.
   Then $(G^{t_0}, \ldots, G^m)$ is a valid test for f.

6. Any circuit primary input feeding a node v must have the value determined by $I^{\alpha(v)}(v)$ during the activation time $\alpha(v)$ of v; at other times its values do not affect the test. Hence we can transform the input patterns as follows to obtain a valid test:

$\forall v \in V,\ I^t(v) \leftarrow I^{\alpha(v)}(v),\ t_0 \leq t \leq m.$

In other words, the input pattern consisting of the pattern $I^{\alpha(v)}(v)$ applied to each node v is a single-pattern test for f in S^B.

7. Return $I^{t_0}(v)$, $\forall v \in V$; this represents the single-pattern test vector T'.

The above procedure demonstrates that every fault in S^B has a single-pattern test; this proves the theorem. □

Example

Figure 3.5(a) shows a typical B-structure with three clouds and three registers. For simplicity in this example, each register consists of a single FF. R3 is a HOLD register while R1 and R2 are LOAD registers. The lines a, b, c, d are circuit primary inputs to the various clouds and g is a circuit primary output. h controls the HOLD mode of R3 such that if $h = 0$ during clock cycle t, then R3 loads new data between clock cycles t and $t + 1$, and if $h = 1$, it holds the data present during clock cycle t for an additional cycle.

Consider the fault f which makes the line e stuck at 1. f is detectable, and Figure 3.5(a) shows a test sequence T, consisting of three test patterns, that detects f. The patterns shown at the primary inputs and at h must be applied in order from left to right over consecutive clock cycles. The states of the internal signals during this time are also shown. Note that the fault is first detected at primary output g in clock cycle 3.

We shall now show that based on T it is possible to derive a single-pattern test for f. We first transform the test so that all HOLD registers (i.e., R3) operate only in the LOAD mode, and then further transform it so that the value applied to each primary input is constant throughout the test.

Given the test sequence T, we say that a cloud C is *active* during a clock cycle k if it is actively involved in sensitizing or propagating the effect of the fault during clock cycle k. In other words, there is a functional dependency between the output of cloud C3, at whose output the fault is first detected at clock cycle 3, and the output of C at clock cycle k. The cycles during which the various clouds in

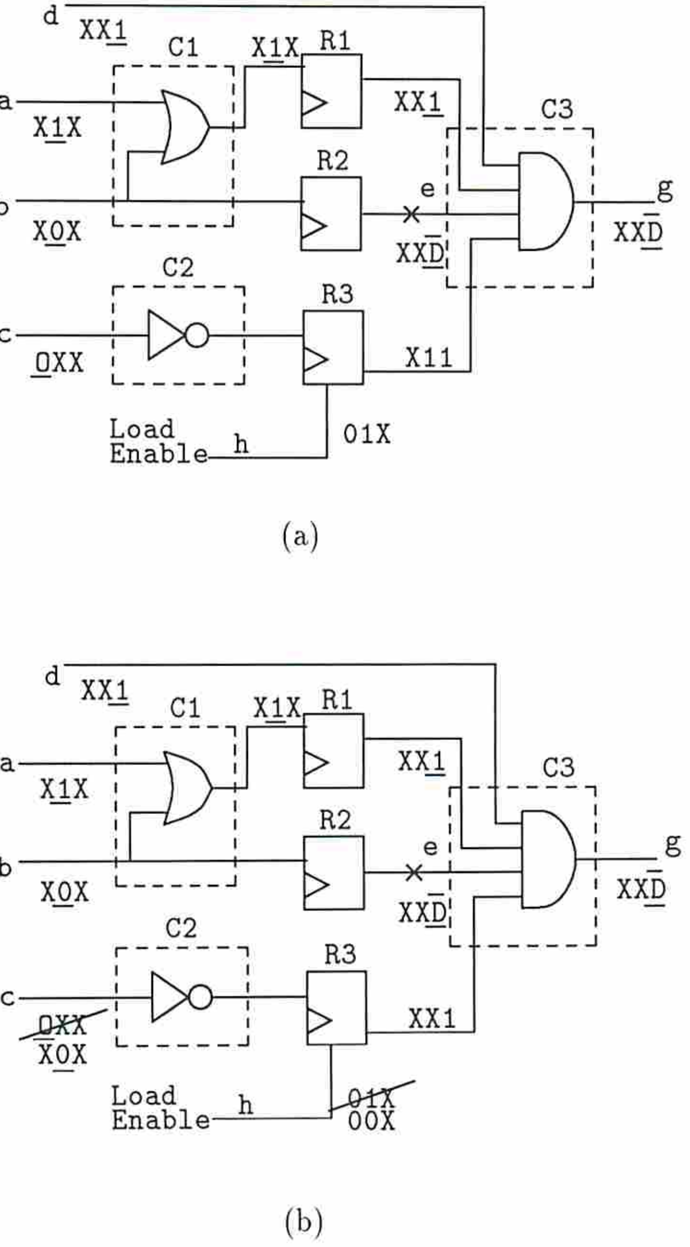


Figure 3.5: (a) Test for e stuck-at-1, (b) transformed test with no HOLD operations.

Figure 3.5 are active are indicated by underlining the corresponding logic values in the state sequences. Clearly, C3 is active at clock cycle 3. C1 must be active at clock cycle 2, since it feeds C3 through the LOAD register R1 which has a constant delay of 1 clock cycle. Note that every path between C1 and C3 must have exactly the same number of LOAD registers, because this structure is balanced; hence C1 must be active at clock cycle 2 and at no other time. The other cloud feeding C3 is C2, and in this case the connection is via a HOLD register, which introduces a variable delay. The control sequence applied to R3 in test sequence T is '01×', which means that R3 is actually in the HOLD mode at the time (clock cycle 2) just before C3 becomes active. The most recent clock cycle at which new data was loaded into R3 is 1; hence the erroneous output of C3 actually depends on the output of C2 in clock cycle 1. Thus C2 is active in clock cycle 1.

Figure 3.5(b) shows a transformed test in which C2 is active in clock cycle 2 instead of 1, and the HOLD operation of R3 is eliminated. In effect, all logical activity in C2 and in all clouds feeding C2 (directly or indirectly) is delayed by one clock cycle, so that the HOLD cycle of R3 can be eliminated. Note that by delaying the activity in the portion of the circuit feeding C2, the activity in the rest of the circuit is unaffected. This follows from the fact that all paths between this portion of the circuit and the rest of the circuit must pass through the HOLD register R3. Thus the sequence of states shown in Figure 3.5(b) is a valid test for f in which the error is first observed at clock cycle 3.

The transformation above ensures that every cloud feeding C3 (which is active at clock cycle 3) is active at clock cycle 2. The transformation process must be repeated for all clouds feeding C1 and C2, respectively (none in this case), until (i) the clock cycles during which the various clouds are active have been determined, and (ii) every HOLD register operates in the LOAD mode in the transformed test. From the preceding arguments it follows that every cloud is active during some unique clock cycle; further, no cloud can be active earlier than clock cycle 2, since adjacent clouds are active during consecutive clock cycles and C3 is active at clock cycle 3 and the depth of the B-structure is 1. Hence the test can be further transformed such that (a) it consists of only two vectors, applied at clock cycles 2 and 3; and (b) the input pattern required at the primary inputs of each cloud C during its *active*

cycle are actually applied during *both* cycles 2 and 3. The resulting single-pattern test vector T' for f is $a = 1$, $b = 0$, $c = 0$ and $d = 1$.

The above example illustrates the procedure for transforming an arbitrary test into a single-pattern test, and demonstrates that every detectable fault is single-pattern testable.

Corollary to Theorem 1 *The maximum required duration of any single-pattern test is d clock cycles, where d is the depth of the B-structure.*

Proof: This follows from the fact that in the transformed test, all registers operate in the LOAD mode, hence adjacent clouds are active during adjacent clock cycles.□

In Theorem 1 we have shown that every B-structure is single-pattern testable. Note that the single-pattern testable property could be characterized a type of "delay-independent" behavior in the sense that for every input-output pair, any value at the output depends on the value at the input at not more than one clock cycle. Based on the information available in our circuit model, i.e., the interconnection of the clouds and registers, B-structures represent a complete class of structures that follow this behavior and are single-pattern testable. Since B-structures can be easily identified (using algorithms which will be presented in Section 3.6), they lead to an efficient partial scan methodology. It should be noted, however, that other single-pattern testable structures could theoretically exist that are not B-structures.[1]

3.5.2 Generating Single-Pattern Tests

We now proceed to show that a complete single-pattern test set for a B-structure can be obtained using ordinary combinational ATPG. Lemma 2 showed that any fault that is detectable in C^B is single-pattern testable in S^B. It remains to be shown that *every* detectable fault in S^B is detectable in C^B.

[1]In fact, in Chapter 5 we will make use of additional information about circuits, viz. the presence of multiplexers and buses, to identify a larger class of structures, called *SB-structures*, that have the single-pattern testable property.

Lemma 3 *Let f_S be a fault in S^B and let f_C be the corresponding fault in C^B. If f_S is detectable in S^B then f_C is detectable in C^B.*

Proof: Let C_f^B and S_f^B be the faulty circuits. Since f_S is detectable in S^B, by Theorem 1 there must be a single-pattern test t for this fault. Hence S^B and S_f^B must have different single-pattern outputs for t. This implies, by Lemma 1, that C^B and C_f^B must have different outputs for t. Thus t is a test for f_C in C^B. □

Theorem 2 *Given a balanced structure S^B, any complete test set for all detectable faults in its combinational equivalent C^B is a complete single-pattern test set for all detectable faults in S^B.*

Proof: Let T be a test set for all detectable faults in C^B. We need to show that T is a single-pattern test set for all detectable faults in S^B.

Let f_S be a detectable fault in S^B and f_C the corresponding fault in C^B. Since f_S is detectable, by Lemma 3 f_C must also be detectable. Since T detects all detectable faults in C^B, it must contain some pattern t that detects f_C. Hence, by Lemma 2, t must detect f_S. Thus T detects all detectable faults in S^B. □

The two theorems reduce the problem of test generation for B-structures to the simpler problem of combinational ATPG, from which a complete single-pattern test set can be obtained. The single-pattern nature of the test sequences makes it possible, in a partial scan circuit, to easily test a kernel that is a B-structure. Each single-pattern test can be shifted into the scan path and held constant for the required number of clock cycles before the test results are loaded into the scan path and shifted out. The implementation issues related to the scan path will be discussed in Section 3.7. Note that there is no loss in single stuck-at fault coverage when this form of partial scan is used in place of full scan.

3.5.3 Observations on Testing B-Structures

Although combinational ATPG is used in both the partial scan and full scan cases, the *test generation effort* may not be the same in both cases. To explain this,

let us denote the combinational equivalent of the B-structure kernel in the partial scan design by PSKCE, and the kernel in the full scan design (which is obviously combinational) by FSK. FSK is simply a collection of all the clouds in the circuit. In PSKCE, the various clouds are cascaded with each other into a single connected structure (according to the connection of registers in the original B-structure). Thus given a particular SSA fault present in both PSKCE and FSK, the fault site is in general farther from the inputs and outputs of the circuit in PSKCE than it is in FSK. This implies a higher amount of analysis to derive a test for the given target fault in PSKCE than in FSK. However, note that the test pattern for this fault may detect more additional faults in PSKCE as a side effect than in FSK, since there are more "critical" faults that get simultaneously activated and sensitized in PSKCE. In general the relative effort in ATPG for the partial and full scan designs is not easily predictable and depends on the nature of the ATPG algorithm.

Regarding *test length*, however, a somewhat stronger statement can be made. In a full scan design, the test length for the circuit is equal to the maximum test length among its individual clouds, since test patterns can be generated separately and applied simultaneously to different clouds. The following lemma relates this test length to that for a B-structure kernel in a partial scan design.

Lemma 4 *Treating only detectable SSA faults in a B-structure as target faults, the test length for the combinational equivalent of the B-structure must be at least as high as the test length for any individual cloud within it.*

Proof by Example Consider the B-structure in Figure 3.3(a). Let the combinational equivalent of this B-structure have a test set T of length L that covers all its detectable SSA faults. Now consider any cloud, say C2. Let the set of faults in C2 covered by T be $F2$. Let $T2$ denote any set of test patterns for C2 that detects all the faults in $F2$, and let $L2$ denote the length of $T2$. We will show that irrespective of the nature of T, there must exist a test set $T2$ such that $L \geq L2$. Assume the contrary, i.e., $L < L2$. Then for each test pattern in T, determine by simulation the corresponding pattern present in R2 during the single-pattern test application. The resulting set of patterns at R2, if applied to C2, must cause all faults in $F2$ to

be detected at the output of C2. By fault simulation on this set of patterns, and dropping those that do not detect any new target fault from $F2$, a test set for C2 with length $L2 \leq L$ is obtained. □

Corollary *Given a partial scan design in which the kernel is a B-structure, and given a single-pattern test set of length L that covers a* given *set of faults, there exists a test set of length at most L for the corresponding full scan design that covers the* same *set of faults.*

Lemma 4 and its corollary imply that the number of test patterns for a partial scan design must generally be higher than the number of patterns required to detect the *same set of faults* in the corresponding full scan design. However, it is possible that the full scan design may have more detectable faults within the combinational logic than the partial scan design. For example, there may be faults in the cloud C2 that can be detected by some pattern applied directly at the inputs of C2 but cannot be detected by any single-pattern test applied to the B-structure. Such faults are said to be *sequentially redundant* since they are detectable using full scan but are not detectable during normal operation of the original sequential circuit.

3.6 Algorithm for Scan Register Selection

Given an arbitrary sequential circuit to which partial scan is to be applied, BALLAST selects a set of registers to be made scannable such that the kernel is a B-structure and the cost of modifying the circuit is minimized. It assumes that the modification cost is the same for registers having equal width, except in the case of a tie between a LOAD register and a HOLD register. In this case the latter is chosen to be in the scan path because converting a HOLD register into a scan register may require a lower area. The selection is carried out using the algorithms presented in this section.

Let $G = (V, A, H, w)$ be the topology graph of the circuit. Formally, we need to determine a set of registers $R \subseteq A$ such that the topology graph of the kernel, $(V, A - R, H - R, w)$, is balanced and $\sum_{a \in R} w(a)$ is minimized.

A near-optimal solution can be obtained using the following steps.

Step 1. Transform $G = (V, A, H, w)$ into an acyclic topology graph G_A by removing a set of "feedback" arcs R_A such that $\sum_{a \in R_A} w(a)$ is minimized.

Step 2. Transform G_A into a balanced topology graph G_B by removing a set of arcs R_B such that $\sum_{a \in R_B} w(a)$ is minimized.

Step 3. $R = R_A \cup R_B$ is the desired set of arcs, and the resulting topology graph $G_B = (V,\ A - R,\ H - R,\ w)$ represents the kernel.

Both steps 1 and 2 above are intractable, as will be explained later. Note that because the problem is partitioned in this way, optimal solutions for the individual steps do not imply an optimal overall solution.

3.6.1 Removal of Feedback Arcs

The problem of carrying out Step 1 above, which requires a minimum-weight feedback arc set R_A to be determined, is known to be NP-complete [21]. Our implementation of BALLAST uses the branch-and-bound algorithm outlined below. We will refer to this algorithm as **breakCycles**. This algorithm is adapted from the one presented in [22], to which the reader is referred for details.

Let $I = \{1, 2, \ldots, |V|\}$, and let $P : V \to I$ represent some total ordering of the nodes in V. Then P uniquely determines an arc set $R_A(P)$ of "feedback" arcs, based on the ordering P of nodes, as follows:

$$R_A(P) = \{(i,j) \in A \mid P(j) < P(i)\}.$$

Essentially, $R_A(P)$ contains all arcs from a higher numbered node to a lower numbered node if the nodes were enumerated according to P. Deletion of all the arcs in $R_A(P)$ would eliminate all cycles. Thus the problem is to find an ordering P such that the total weight of the arcs in $R_A(P)$ is minimum; then $R_A = R_A(P)$.

To characterize our branch-and-bound algorithm we need to define a state in the search space, a branching rule, and a bounding rule. A *state* is defined as a tuple

(W, σ) where $W \subseteq V$ and σ is a total ordering on W. A state with $W = V$ is called a *leaf state* and represents a potential solution; any non-leaf state represents only a partial solution. Given a current state (W, σ) with $W \neq V$, branching is carried out by generating a new state of the form (W', σ') where $W' = W \cup \{w'\}$, $w' \in V - W$; $\forall w \in W$, $\sigma'(w) = \sigma(w)$; and $\sigma'(w') = |W| + 1$.

A partial solution state (W, σ) can be *bounded,* or removed from further consideration along with all states derived from it, if the lower bound on the weight of any leaf state that can be generated from it is higher than the weight of the best potential solution generated so far. Our algorithm computes the lower bound as follows. Let F_L represent the set of local feedback arcs whose source and destination nodes are both in W and let F_G represent the set of global feedback arcs with source node in $V - W$ and destination node in W; i.e.,

$$\begin{aligned} F_L &= \{(i,j) \in A \,|\, i,j \in W \text{ and } \sigma(j) < \sigma(i)\}; \\ F_G &= \{(i,j) \in A \,|\, i \in V - W \text{ and } j \in W\}. \end{aligned}$$

Using F to denote $F_L \cup F_G$, a lower bound on the cost of eliminating all feedbacks is given by the expression

$$\sum_{a \in F} w(a).$$

The **breakCycles** algorithm begins with the state $(W = \phi, \sigma = \phi)$ and the process of branching and bounding continues until all possible states in the state space have been either visited or eliminated by bounding. Note that since all possible states are implicitly enumerated, the solution obtained is optimal.

The number of states that can be eliminated by bounding depends not only on the computation of the lower bounds but also on the *branching rule* used for selecting the next state to branch to. With a good branching rule, one or more good solutions can be found early in the search process, which help to eliminate partial solutions later. Our algorithm uses the following guideline. Each time a branch is generated at a given state, all the possible child states are clearly similar except for the newly added node. We select that state whose new node has the largest number of other nodes in G reachable from it as the next state to visit. The

underlying goal is to ensure that as many arcs as possible lie in the forward direction according to the ordering generated, and thus to try to find good solutions quickly. The information on the number of reachable nodes is determined beforehand by carrying out a transitive closure operation on the topology graph.

3.6.2 Balancing Acyclic Sequential Structures

A simplified form of Step 2 has also been shown to be computationally intractable [17]. In this section we present a heuristic procedure, **balance**, for solving this problem. **balance** uses a verification procedure, **check**, to verify that a given structure is balanced.

3.6.2.1 Verification Procedure

Given the topology graph G of a sequential structure S, the following procedure checks whether S is balanced. First the procedure checks whether the removal of each arc in the hold set would disconnect the topology graph. Then, starting at each root node (i.e., node with no incoming arcs) in turn, it levelizes the portion of the graph reachable from it. If every node is found to be at a unique distance from each root the graph is pronounced balanced and the procedure returns SUCCESS. If an imbalance is detected at any time the procedure exits and returns FAILURE.

function check $(G = (V, A, H, w))$**:** Returns SUCCESS if G is balanced, FAILURE otherwise.

1. Construct the graph $G' = (V, A - H, \phi, w)$, by removing all HOLD registers from G. Determine the connected components $\{C_1, C_2, \ldots, C_K\}$ of G'.
2. Construct the graph G'' consisting of the original topology graph G with each connected component C_i of G' replaced by a single node. Note that the arc set of G'' is H.
3. Determine whether G'' is a tree. If it is, condition 3 of the definition of B-structures is satisfied; proceed to the next step. Otherwise return FAILURE.
4. Repeat for each C_i:

(a) Determine the set $ROOTS$ of root nodes of C_i, where *root nodes* are defined as nodes with no incoming arcs. If there are one or more root nodes, proceed to the next step; otherwise return FAILURE since condition 1 in the definition must be violated.

(b) Pick a root node v_1 in $ROOTS$. Starting at v_1, carry out a breadth-first traversal of all nodes in C_i reachable from v_1 by a directed path, and assign each node visited a *level number* equal to its distance from v_1. If at any time a node v_2 needs to be assigned a level number when it has been previously assigned a different level number with respect to v_1, stop the search and return FAILURE, since C_i must violate either condition 1 or condition 2. Continue the traversal until no more nodes can be visited.

(c) Repeat step 4b for all root nodes in $ROOTS$.

5. S is a B-structure; return SUCCESS. □

Figure 3.2 indicates the level number assigned by the procedure to each node. In the above procedure, the computation complexity is $O(n+m)$ where $n = |V|$ and $m = |A|$.

3.6.2.2 Balancing Procedure

An acyclic topology graph $G = (V, A, H, w)$ is balanced if and only if all paths between any given pair of nodes are of equal length and the removal of any arc in H disconnects G. A heuristic procedure, **balance**, for balancing G is presented below. It returns a set of arcs $R \subseteq A$ such that the derived topology graph $(V, A - R, H - R, w)$ is balanced. The procedure assumes that G is fully connected; if G is disconnected, the procedure must be invoked on each connected component to balance it separately.

The procedure works in a recursive manner by partitioning the topology graph into two smaller topology graphs, balancing them independently, and then merging the solutions. The partition is obtained by determining a minimum cost cutset (mincut) CS of the topology graph. The intuitive idea behind using a mincut is to minimize the sensitivity of the overall solution to the degree of optimality of the merging process (which is based on a greedy heuristic). The mincut can be determined by applying the maximum flow algorithm [23] to the topology graph,

treating primary inputs as sources and primary outputs as sinks, with capacity of arc $a \in A$ defined by $w(a)$.

During merging, a greedy algorithm is first used to determine a maximal set of LOAD registers taken from the mincut CS that can join the two balanced substructures while keeping the merged topology graph balanced. From the definition of B-structures it follows that if any HOLD register is used to connect the two balanced sub-structures, no other register should connect them. Thus an alternative to using the set of LOAD arcs derived above is to use the maximum-weight HOLD arc in the mincut. The costs of these two solutions are compared to determine the set of arcs to be actually used for merging.

function balance ($G = (V, A, H, w)$: acyclic topology graph)**:** Returns $R \subseteq A$ such that $(V, A - R, H - R, w)$ is balanced.

1. If (**check**(G) = SUCCESS) then return ($R \leftarrow \phi$), else proceed.
2. $CS \leftarrow$ minimal cost cutset of G; let G_s, G_d be the subgraphs of G induced by CS.
3. Balance G_s and G_d separately; $R \leftarrow$ **balance**(G_s) $\cup$ **balance**(G_d) $\cup\ CS$.
4. Let $CS_H \leftarrow CS \cap H,\ CS_L \leftarrow CS \cap (A - H)$ be a partition of CS into its HOLD and LOAD registers. Sort the arcs in CS_L in order of decreasing cost.
5. $CS'_L \leftarrow \phi$, the set of LOAD arcs retained in the topology graph when merging G_s and G_d.
6. For all arcs a in CS_L, in order of decreasing cost:
   Check whether the inclusion of a makes the merged graph unbalanced:
   If [**check**(V, $(A - R) \cup CS'_L \cup \{a\}$, $H - CS$, w) = SUCCESS]
   then $CS'_L \leftarrow CS'_L \cup \{a\}$.
7. Let $a_H \leftarrow$ highest-cost arc in CS_H.
8. If $\sum_{a_L \in CS'_L} w(a_L) > w(a_H)$ then $R \leftarrow R - CS'_L$ else $R \leftarrow R - \{a_H\}$.
9. Return R. □

The run time of the above algorithm consists of two parts: (a) the computation of a minimum cutset at each recursive step, which may be repeated $O(m)$ times

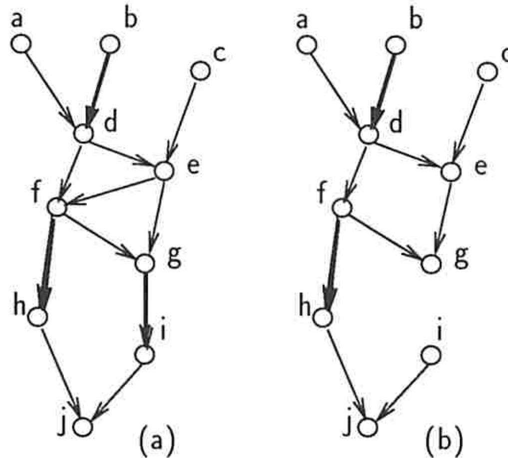


Figure 3.6: Illustration of balance procedure. (a) Original topology graph, (b) balanced topology graph. (Bold arcs represent HOLD registers.)

where $m = |A|$; and (b) the invocation of the **check** procedure, which is carried out at most once for every arc in a minimum cutset, i.e., $O(m)$ times. Since step (a) dominates the time required by the **check** procedure, the overall complexity is that of computing $O(m)$ minimum cutsets.

In this section we have described a procedure for determining a near-minimal cost set of registers to be made scannable. Since a branch-and-bound algorithm is used to solve a part of the problem, the time complexity of the procedure grows exponentially with the number of clouds in the worst case. However, the computation time is not excessive since the procedure deals with each cloud of connected combinational logic as a single node irrespective of the number of gates/transistors within the cloud.

Figure 3.6(a) shows a topology graph that is not balanced. (HOLD registers are indicated by bold arcs.) Figure 3.6(b) shows the result of applying **balance** to it; the arcs removed are (e, f) and (g, i), and the registers in the circuit that correspond to these arcs must be included in the scan path.

Note that although the heuristic **balance** produces a single scan FF selection, the heuristic could easily be extended (for example, by using different cuts other than the mincut to partition the topology graph) to produce a range of solutions. Thus for the circuit of Figure 3.1, a range of alternative scan designs shown in Figure 3.7 could be derived. (Shaded blocks represent scan registers.) Note that every scan register

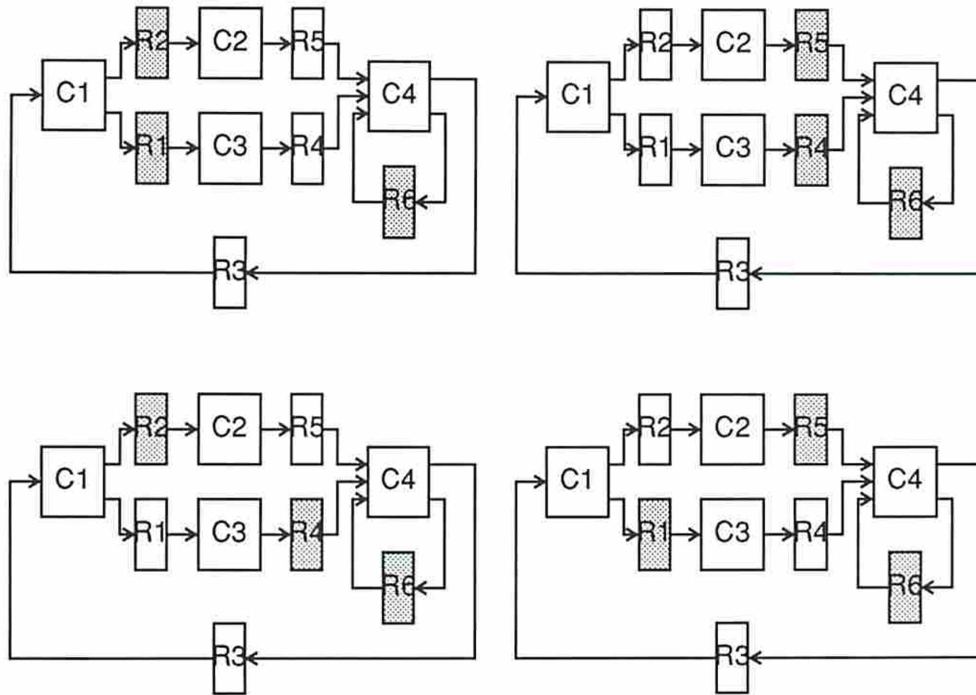


Figure 3.7: Various scan design solutions for the same circuit.

implies an additional delay for signals passing through it. Hence these designs may have varying levels of performance degradation depending on what scan registers, if any, lie in critical timing paths. By obtaining the combinational equivalent of each kernel, the test length and associated test application time can also be determined for each design. Thus among these designs, the solution that best meets the user's goals on area overhead, performance degradation, test time, etc. could be selected.

3.7 Implementation of Scan Path

The BALLAST test application procedure, in its simplest form, consists of scanning a pattern into the selected registers and holding it constant for d clock cycles, where d is the depth of the B-structure comprising the kernel. If the overhead of providing all selected SPRs with HOLD modes is acceptable, no extra work needs to be done. However, not all patterns need to be held at the kernel inputs during all d clock

cycles. In this section we show how by introducing dummy bits the need for some SPRs to have a HOLD mode can be eliminated.

3.7.1 Example

Consider the BALLAST partial scan design shown in Figure 3.1(b). The scan path contains the registers R6 and R3, which are ordered in the scan path as shown. Each test pattern consists of two parts, t_3 and t_6, corresponding to the SPRs R3 and R6 respectively. If R3 and R6 are both provided with HOLD modes, one way to apply the test is to scan in the combined pattern $t_3 t_6$ and hold it in the scan path for 2 additional clock cycles before loading the output of C4 into R6 and R3 after the 3rd clock cycle.

The test result loaded back into the scan path depends on the value in R3 during the 1st clock cycle and on the value in R6 during the 3rd clock cycle. Hence if we can arrange to have t_3 in R3 in the 1st clock cycle and t_6 in R6 in the 3rd clock cycle, the HOLD modes in these registers can be eliminated without affecting the test result. This can be done by inserting two dummy (*"don't care"*) bits in the test pattern between t_3 and t_6. When t_3 becomes available in R3, t_6 is still displaced by 2 bits from its final destination in R6. After two more shifts, during the 3rd cycle R6 contains the correct input pattern. Hence the desired test result is loaded into R3 and R6 after the 3rd clock cycle. Thus the modified test pattern with two dummy bits eliminates the need for HOLD modes in both R3 and R6.

The example presented above is a simple illustration of the concept used to minimize the overhead due to additional HOLD modes. In general, however, it may not be possible to eliminate the HOLD modes of all the SPRs. We now describe how the number of additional HOLD modes can be minimized in an arbitrary circuit.

3.7.2 Construction of Scan Path

In general a SPR can play two roles: it can serve as a **driver,** i.e., a SPR that feeds test inputs to a cloud, or as a **receiver,** i.e., a SPR that is fed test results by a cloud.

A single SPR can simultaneously play both roles; it can drive test patterns into one cloud and receive test results from another (or the same) cloud. An example of a pure driver or pure receiver would be a boundary scan element in a chip employing the boundary scan methodology.

Let the **distance** between a driver and a receiver, given that at least one path exists between them, denote the number of registers in any path between the driver and the receiver (excluding themselves) through the kernel. Note that since the kernel is a B-structure, all such paths must have the same number of registers. If the distances from a given driver to all receivers to which it has a path (including itself if it is also a receiver) have the same value, define the **latency** of the driver as this value; otherwise let the latency be undefined. A latency of x for a driver implies that the test result loaded into the scan path during clock cycle k depends only on the value present in the driver during clock cycle $k - 1 - x$.

The above concepts are illustrated in Figure 3.8. The registers R2, R3, R6, R7 and R8 are included in the scan path and ensure that the kernel is a valid B-structure. Note that R7 and R8 are not strictly required to be scannable to ensure that the kernel is a B-structure; however, they may be included in the scan path if the circuit employs boundary scan. In this circuit R7 is a pure driver; R8 is a pure receiver; and R2, R3 and R6 play both roles. In the original circuit all registers are LOAD registers except R2, which has a HOLD mode. The latency of each driver in Figure 3.8 is indicated alongside it as $[x]$. Undefined latencies are denoted by $[U]$.

We now need to determine the following: (1) which SPRs (if any) should be provided with a new HOLD mode; (2) an ordering of all the SPRs in the scan path; and (3) a pattern of dummy bits to be inserted into each test pattern. We consider these issues in turn in the following rules for constructing the scan path.

Given a driver with *undefined* latency, the test result due to each test pattern depends on a single pattern being present in the driver during more than one clock cycle. This follows from the definition of latency. For example, consider the driver R3 in Figure 3.8. If the scan path is to be loaded with the test result during clock cycle k, the result to be loaded into R2 depends on the pattern in R3 during $k - 1$, and the results to be loaded into R8, R6 and R3 depend on the pattern in R3 during

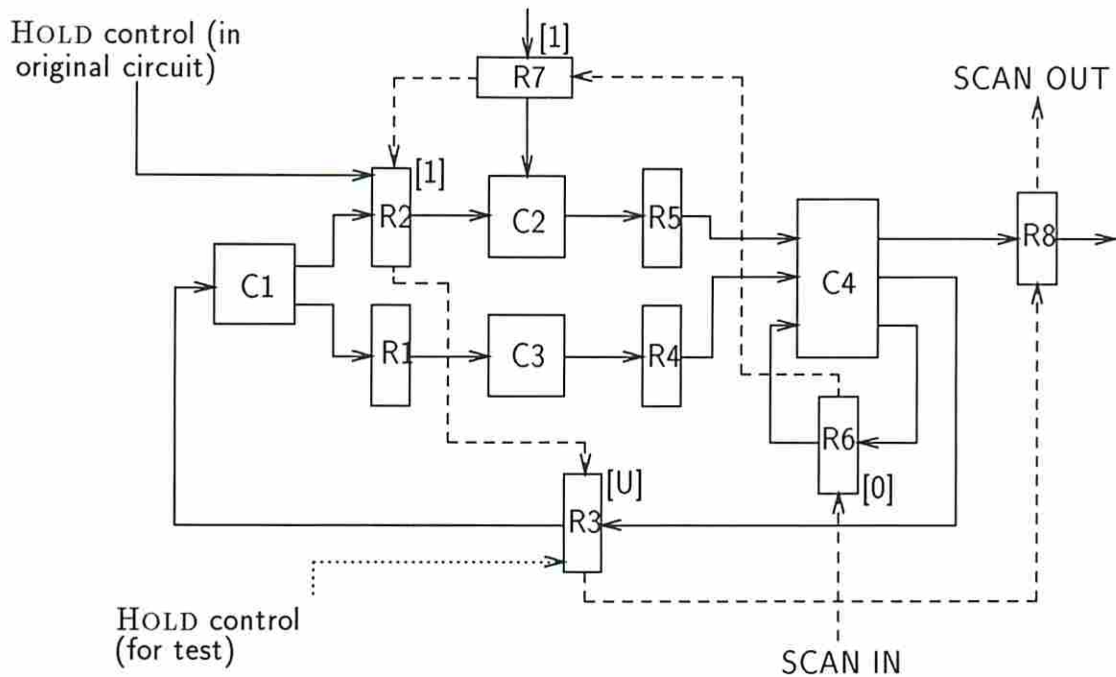


Figure 3.8: Example of scan path implementation.

$k-3$. For single-pattern testability it is essential that the same desired value be present in R3 during $k-1$ as well as $k-3$. Hence R3 must be provided with a HOLD mode. The general rule is stated below.

Rule 1 *All drivers whose latencies are undefined must have* HOLD *modes.*

The scan path is constructed by forming $d+2$ groups of SPRs, where d is the depth of the B-structure comprising the kernel. Figure 3.9 shows the sequence of groups in the scan path. The group closest to the scan-in pin is Group 0 and the group closest to the scan-out pin is Group $d+1$. Within each group the ordering of SPRs may be based on routing considerations. SPRs are assigned to groups according to the three rules presented below.

Since pure receivers never need to be supplied with test input data, all such registers can be placed at the scan-out end of the scan path so as to minimize the number of clock cycles needed to shift a new pattern into the scan path. In the example of Figure 3.8 (which has $d = 2$), R8 is placed in Group 3 of the scan path. This principle is stated in the following rule.

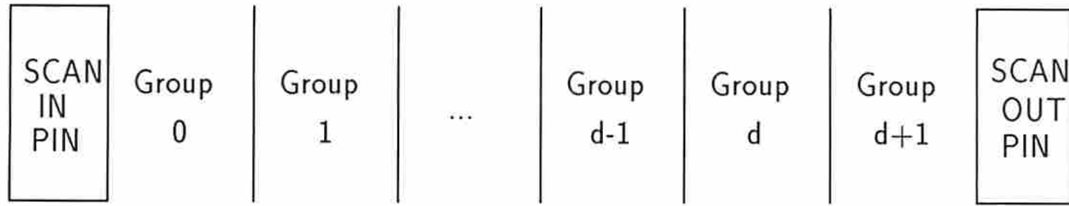


Figure 3.9: Organization of scan path into groups of scan path registers.

Rule 2 *All pure receivers are placed in Group* $d + 1$.

Drivers are placed in the scan path according to their latency values. As we have seen, a driver with a well-defined latency need not have a HOLD mode, provided the arrival time of the desired test data in this driver is synchronized with the appropriate arrival times of test data in the other drivers. For example, in Figure 3.8, R6 and R7 are not provided with HOLD modes. For the test to be applied correctly to the kernel, R6 must receive its correct pattern exactly one clock cycle after R7, since the difference in their latency values is 1.

One way to achieve this synchronization is as follows. Drivers with high latency are placed further along the scan path. The test patterns for all drivers are concatenated into a single pattern and scanned in; however, dummy bits are placed at selected positions in the test pattern so that test data destined for drivers with lower latencies lag behind those for drivers with higher latencies. The following two rules deal with the ordering of registers so that the test patterns can be applied in this manner, and the actual formatting of pattern is described in a final rule.

Rule 3 *All drivers not having a* HOLD *mode and having latency* l *must be placed in Group* l.

Note that all drivers not having a HOLD mode must have well-defined latency values. This rule ensures that drivers having higher latency are placed further along the scan path than those with lower latencies. Thus drivers in Group d receive the correct pattern first; at this instant the testing of the kernel effectively begins even though the patterns destined for drivers in Group 0 still lag by d clock cycles.

So far we have not considered the placement of drivers having HOLD modes. The following rule places such drivers in the same group as drivers having latency d.

This ensures that they are among the first drivers to receive their correct pattern, which they then hold for the duration of the test (i.e., for the next d clock cycles).

Rule 4 *All drivers provided with* HOLD *modes for test and all drivers with* HOLD *modes already present are placed in Group d.*

Thus both Rules 3 and 4 may assign drivers to Group d. According to these rules, R2 and R3 are placed in Group 2, R7 is placed in Group 1, and R6 is placed in Group 0. The resulting scan path is shown in Figure 3.8.

3.7.3 Test Application

Once the drivers have been ordered according to the grouping determined above, we simply need to ensure that the pattern for each group of drivers lags behind the pattern for its neighboring group by one clock cycle. This is done by the following rule.

Rule 5 *Each single-pattern test is modified by introducing one dummy bit between each pair of consecutive groups* i *and* $i+1$, $0 \leq i \leq d-1$.

Thus a total of d dummy bits is added. Two or more adjacent dummy bits may be present in a sequence when there are empty groups in the scan path. In the example of Figure 3.8, if t_i represents an input pattern for register Ri and $\times$ represents a dummy bit, the modified test pattern to be scanned in (ordered from left to right) is $t_3 t_2 \times t_7 \times t_6$. No input pattern is required for R8 since it is a pure receiver.

When the scan path is implemented as described in this section, the formal test procedure given in Section 3.4 needs to be modified by generalizing step 2(a) as follows:

2. (a) *Place all scan registers having a* HOLD *mode in the* HOLD *mode, all other scan registers in the* SHIFT *mode, and all non-scan registers in the* LOAD *mode for d clock cycles. (This allows test data to propagate through the kernel.)*

3.7.4 Circuit Modifications

The registers in the original circuit may be classified into four types depending on (1) whether or not they have a HOLD capability, and (2) whether or not they are to be included in the scan path. The modifications required for each type of register so that they can perform the appropriate functions according to the test plan are listed below.

Non-Scan Registers Without HOLD Mode: These require no modification.

Non-Scan Registers With HOLD Mode: The individual HOLD controls of all such registers should be externally controllable so that they can be operated in LOAD mode while the kernel is being tested.

Scan Registers With HOLD Mode: These registers need to be converted into scan path registers by the addition of SHIFT modes. Their HOLD mode signals must be externally controllable so that they can be made to hold test patterns for the required number of clock cycles. No HOLD control signal for a scan register may serve as a control signal for a non-scan register.

Scan Registers Without Hold Mode: In addition to being augmented with SHIFT modes, some of these registers need to be provided with HOLD modes as well. The HOLD control signals for all such registers may be controlled by a single external pin.

These modifications may result in a small logic overhead and possibly additional I/O pins for ensuring that the HOLD signals are appropriately configured.

3.8 Testing Register Functional Modes

As we have mentioned previously, a complete set of single-pattern test vectors can be obtained for covering all detectable faults in the combinational logic. While

exercising the combinational logic, every FF must operate in the LOAD mode; hence other built-in modes of operation of the FFs may not get exercised. In this section we deal with the problem of testing the RESET (or CLEAR), PRESET and HOLD modes of operation, if present in any FFs in the circuit. Further, we focus our attention on faults in non-scan FFs, i.e., FFs within the kernel. FFs in the scan path are either tested for free during tests for the kernel or can be easily tested using special patterns shifted in and out of the scan path.

The three functional modes listed above give rise to six possible fault modes: stuck-at-RESET, cannot-RESET, stuck-at-PRESET, cannot-PRESET, stuck-at-HOLD and cannot-HOLD. We shall map faults in the functional operation of FFs on to structural faults on the control lines for the FFs. Given the B-structure S^B under test, we generate the combinational equivalent C^B using the functional combinational equivalent of each FF, depending on its built-in functions, rather than simple wires and/or inverters.

Figure 3.10(a) shows a FF connecting two clouds C_1 and C_2, and Figures 3.10(b) and (c) show how the combinational equivalents of FFs having RESET and PRESET modes, respectively, are used. Each fault in these functional modes is functionally equivalent to a stuck-at-0/1 fault on the corresponding control line. Thus it is sufficient to detect both stuck-at-0 and stuck-at-1 faults on the control lines using the combinational equivalent. Note that some of the faults may be tested for free while testing the clouds of the kernel. The control lines may be treated as primary inputs for the purpose of test pattern generation. If a control line fans out to more than one FF, as shown in Figure 3.10, faults on all the lines marked $\times$ (i.e., the fanout stem as well as the fanout branches) must be tested. This ensures that all appropriate fault modes are covered.

Figure 3.10(d) shows the combinational equivalent of a HOLD FF. Unlike the faults considered earlier, the manifestation of HOLD faults depends on the previous state of the circuit. Two single-pattern test vectors are required to detect the cannot-HOLD fault. While the first vector is applied all FFs operate in the LOAD mode, and while the second vector is applied the FF under test (and possibly other FFs) operate in the HOLD mode. Two copies of the combinational equivalent of the B-structure, viz. C_1^B and C_2^B, are required for generating a test. This is illustrated

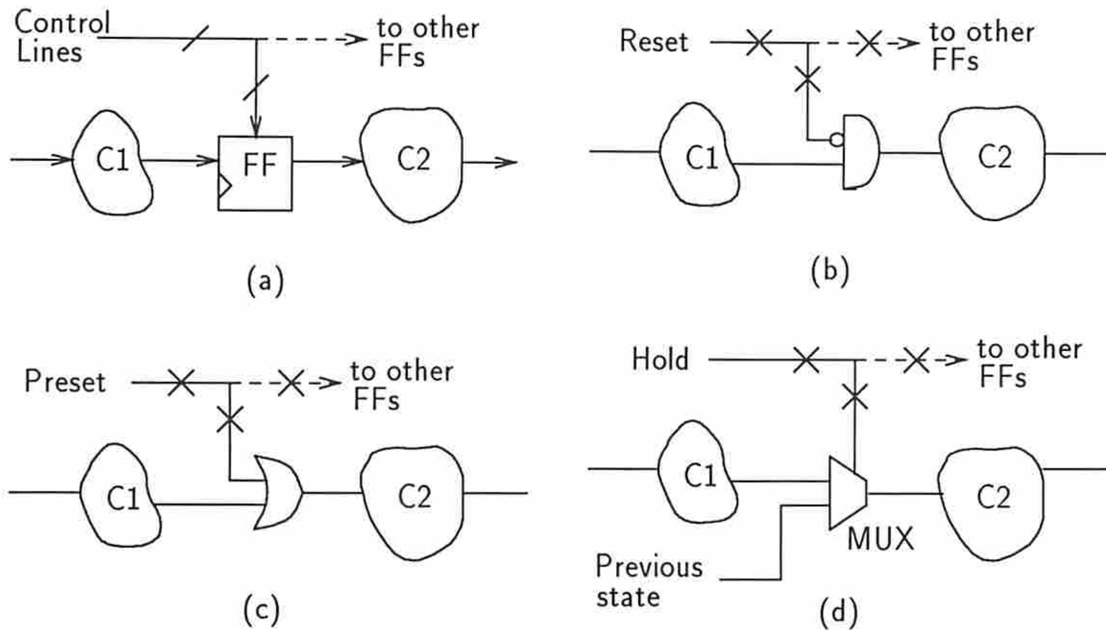


Figure 3.10: Modeling FF functional modes. (a) FF connecting two clouds, (b) model of RESET operation, (c) model of PRESET operation, (d) model of HOLD operation.

in Figure 3.11. Note that multiplexers are not required to model the HOLD FFs in C_1^B since all FFs operate in the LOAD mode while the first vector is applied. Test pattern generation on the combined combinational structure yields two patterns t_1 and t_2 corresponding to C_1^B and C_2^B, respectively. During test, first t_1 is scanned in and applied to the kernel in the normal way with all FFs in the LOAD mode. After the kernel outputs have stabilized the control signal under test is switched to the HOLD mode. t_2 is now scanned in and applied to the kernel in the normal way except for the control signal under test being in the HOLD mode. The output of the kernel for the second pattern is used to detect the fault.

The approach described above is sufficient if the HOLD control line for the FF under consideration is independent of all other control lines. If there are dependencies among the control lines, however, the issue of testing them is more complex and some form of functional testing may be required.

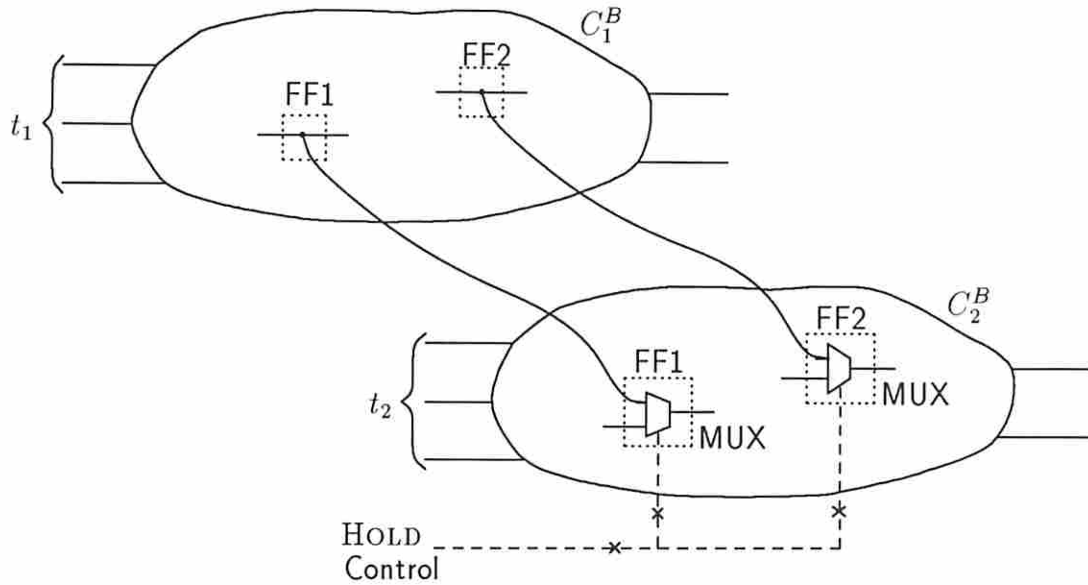


Figure 3.11: Combinational equivalent for detecting HOLD faults.

3.9 Case Study

Both full scan and the BALLAST technique were applied to a Viterbi decoder designed by the Jet Propulsion Laboratory. The basic building block was a chip containing 16 butterfly circuits. Each chip contained 448 FFs and the total gate count was 7,280. In our overhead analysis we have ignored the area due to 16 shift registers present on the chip since they do not cause an area overhead for either full or partial scan. In both versions a global test mode signal was used to control the operation of the scan path FFs. The various characteristics of both designs are shown in Table 3.1. The logic overhead figures are based on the increase in the gate count.

Type of scan	**Full scan**	**BALLAST**
No. of scan path FFs	448	256
Logic overhead	24.6%	14.1%
Pin overhead	3	4
No. of test patterns	34	41
Clock cycles to apply each pattern	449	262
Overall test time	15,714	10,998

Table 3.1: Comparison of full scan and BALLAST partial scan.

BALLAST required only 256 of the 448 FFs to be made scannable, of which only 96 were required to hold data during test application. This was achieved in the following manner. Instead of one global clock signal controlling all FFs in the circuit, two separate clock signals CK1 and CK2 were used. CK2 was connected to the 96 scan FFs that needed to hold data during test; CK1 was connected to all other scan and non-scan FFs. Both clocks were operated simultaneously at all times *except* during the part of the test plan when the 96 scan FFs connected to CK2 were required to hold their data. During these clock cycles only CK1 was operated while CK2 remained inactive. This implementation required one additional I/O pin and there would be an additional routing overhead due to the separate clock signals.

The resulting kernel had depth 5. The number of single-pattern tests generated by combinational ATPG on the combinational equivalent circuit was higher than in the full scan case. However, there was a 30% reduction in the overall test time for the BALLAST circuit due to the shorter scan path, even though dummy bits were added to each test pattern.

3.10 Summary

In this chapter a methodology for partial scan design has been described. The foundation of this approach is based on defining a new class of synchronous sequential circuits called *balanced structures*. They have the following properties which make them useful as kernels in a partial scan circuit: (i) they are single-pattern testable for all detectable faults in the combinational logic and nearly all faults in the storage elements; (ii) the FFs internal to these networks need not be made scannable; and (iii) they can be treated as combinational circuits for the purpose of test generation.

The concept of balanced structures can be used to reduce various overheads in partial scan design for arbitrary sequential circuits. By identifying the balanced sub-structure of the circuit that has the largest number of FFs, it is possible to minimize the test overhead by configuring the balanced sub-structure as a kernel to be tested using a partial scan path without any loss in fault coverage. The computation time for this analysis is in general orders of magnitude lower than the

time required for sequential ATPG on the original circuit. Some FFs in the scan path may need to have a HOLD mode so that single-pattern tests can be applied to the kernel. By ordering the scan path registers appropriately the number of such HOLD modes required can be reduced to a minimum, provided the test patterns are modified by inserting dummy bits where necessary. Case studies indicate that the logic overhead for the scan path can be reduced significantly using this partial scan methodology, particularly in pipelined circuits such as those which often occur in digital signal processing chips. By eliminating scan registers in the critical path the performance of the circuit can also be enhanced.

Chapter 4

Partial Scan Design with Unbalanced Structures

"Less is more."—Robert Browning

4.1 Introduction

Automatic test pattern generation (ATPG) for acyclic sequential structures is known to require substantially lower computation effort than for arbitrary sequential structures [20]. This fact is made use of in the BALLAST approach presented in Chapter 3 as well as in other techniques [16]. Essentially they select flip-flops (FFs) to be included in the scan path such that the portion of the circuit effectively under test, the **kernel**, is either acyclic or close to acyclic. For example, BALLAST makes the kernel acyclic and balanced. BALLAST ensures that a single-pattern test is sufficient for testing all faults. For a general unbalanced kernel, however, a sequence consisting of one or more patterns may be required to detect any fault.

In this chapter we assume that a set of FFs has already been selected such that the resulting kernel is acyclic. This selection could be carried out using the algorithm of Section 3.6.1. Any sequential ATPG program can then be used to obtain test sequences for the faults in the kernel. To apply a sequence of test patterns to the kernel, the circuit must be designed such that while in the test mode the clock signals for scan FFs should be controllable independently of the clock signals for the

non-scan FFs. A test sequence can then be applied to the kernel using the following two steps alternately:

1. Serially shift a test pattern into the scan path while disabling the clock signal feeding the non-scan FFs (this effectively puts the non-scan FFs in a HOLD mode);

2. While disabling the clock signal feeding the scan FFs (putting them in a HOLD mode), activate the clock signal for the non-scan FFs for one clock cycle (this enables test data to propagate through one level in the kernel).

Formally, a *test sequence* for a fault in a sequential circuit consists of a set of consecutive time frames, in each of which patterns containing both specified and don't-care values may need to be applied at the various inputs. The *length* of a sequence is the total number of time frames in it. For an acyclic circuit structure the length is related to the **depth** or the highest number of FFs in any path in the structure. If d is the depth of a structure, the test sequence length is bounded by $d+1$. However, a given test sequence may contain unassigned or don't-care input values such that not all primary inputs need to be provided with new data at each of the $d+1$ clock cycles. If the inputs and outputs of the structure under test are directly accessible, the time for applying the sequence is $d+1$ clock cycles and is not affected by the presence of don't-care inputs. However, in a partial scan design many of the inputs and outputs of the structure are accessed by shifting data serially. Hence the presence of don't-care input values could potentially lead to a great saving in test time. In such circuits, where the length of the scan path is usually much higher than d, the test time is dominated by the time to shift new patterns into and out of the scan path.

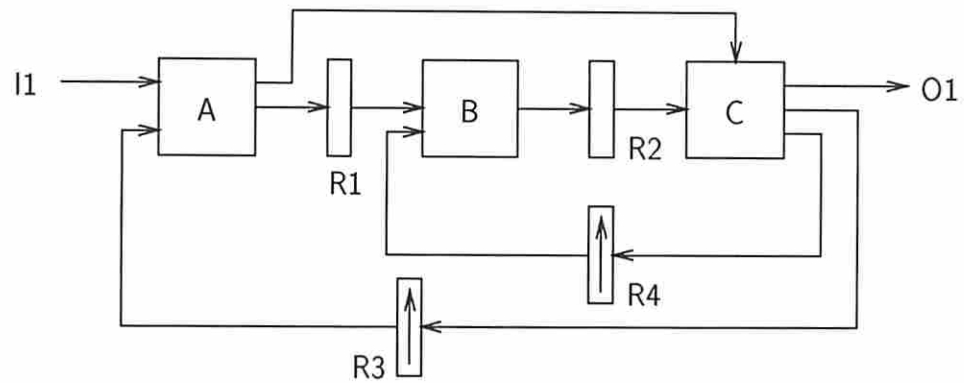
An example of acyclic structures which need less than $d+1$ input patterns are kernels in the BALLAST methodology. In BALLAST, scan path FFs are selected so as to make the kernel not only acyclic but also balanced, i.e., every path between any two points in the kernel has the same number of FFs. Such kernels (known as *B-structures*; see Chapter 3) require only a single-pattern test for any fault. Each test pattern in general needs to be held in the scan path for $d+1$ clock cycles. The number of FFs to be included in the scan path in the BALLAST approach, however,

is higher than that required to just make the kernel acyclic, hence the overhead cost is higher.

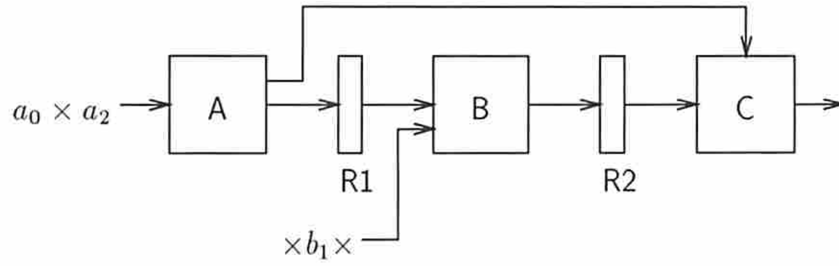
In this chapter we present an alternative partial scan approach, ACYST (ACYclic structured Scan Test). We study the implications of using a kernel that has an acyclic structure but may be unbalanced. Clearly the area overhead for this case is lower than for a balanced kernel. However, there are two drawbacks. First, a simple combinational equivalent cannot be used for ATPG as in BALLAST. Second, any given fault may require a sequence of up to $d+1$ patterns to detect it; this can lead to a high test time to apply a single test sequence, since the individual patterns need to be shifted separately into the scan path. The ACYST methodology deals with these two drawbacks introduced by the unbalanced nature of the kernel. Although it cannot eliminate them, it minimizes their impact, making the use of an unbalanced kernel more acceptable.

For example, consider the circuit shown in Figure 4.1(a) consisting of combinational logic blocks A, B, C interconnected with registers. Each connection shown may consist of any number of wires. Registers R3 and R4 are selected to be made scannable since the resulting kernel, shown in Figure 4.1(b), is acyclic. Note that the kernel is obtained from the original circuit simply by removing the scan registers and replacing them by pseudo-inputs/outputs. In the kernel, inputs/outputs connected to the same combinational logic block are merged together; thus the kernel effectively has one output at C which feeds scan path registers and the primary output O1, and inputs at both A and B are fed by scan registers and/or the primary input I1. The depth of this kernel is 2.

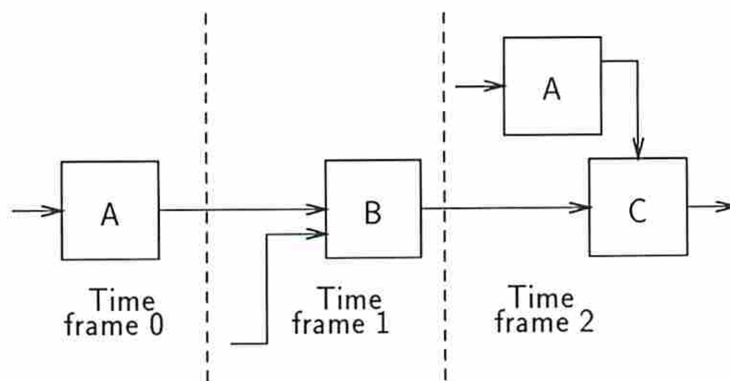
Typical sequential ATPG programs would construct an *iterative array* consisting of up to 3 copies of the kernel, each representing one time frame, and attempt to find a test sequence for a given fault (if one exists) within these time frames. Since the kernel has only one primary output at C, any test sequence for a fault must propagate an error to this output. Assume that the error is visible at the output at time frame 2. Because of the topology of the circuit this output value must depend only on the primary input values at A at time steps 0 and 2 and on the primary input value at B at time frame 1. All other input values are essentially "don't-cares" in all possible test sequences and are indicated by '×' in the input sequences in Figure



(a)



(b)



(c)

Figure 4.1: Example of ACYST. (a) Partial scan design, (b) acyclic kernel, (c) test generation model.

4.1(b) (to be applied in the order from left to right). Note that each '$\times$' represents a vector of don't-care values. An ATPG program would normally assign random logic values to these inputs. This means that approximately half the test time in this case could be taken up in shifting random data into the scan path.

The effectiveness of this test process can be improved in two ways. First, the test pattern generator can be enhanced to fill in the unassigned input values with deterministic patterns that detect one or more additional faults, rather than with random data. In general there is no guarantee that any additional faults can be detected in this way, especially if most of the circuit faults have already been covered. Second, the test sequence can be compacted so that the shifting time is reduced without reducing its effectiveness. The latter approach is the subject of this chapter. In the simple example of Figure 4.1, A has its input value specified at time frame 0 but not at 1, while B has its input value specified at 1 but not at 0. Hence we can combine the first two patterns by shifting a_0 and b_1 simultaneously into the scan path, holding them there for two clock cycles instead of one, and then shifting in a_2. This is a modification of the basic test procedure described earlier in this section. Thus the number of *shift cycles* of the scan path (i.e., the number of times a new pattern is shifted in) is reduced from three to two without losing any of the deterministic part of the test sequence. Note that this applies irrespective of which fault is under test. Intuitively, the fact that there are two "unbalanced" paths from A to D with unequal delays indicates that in general two distinct input patterns at A will be required to guarantee detection of an arbitrary fault.

In Section 4.2 we present a more formal and general discussion of how to test unbalanced acyclic structures. We use a formal model to compute a lower bound on the number of shift cycles required to test for an arbitrary fault, and present a test compaction algorithm that achieves the lower bound. In Section 4.3 we study the problem of simplifying the iterative array model used for ATPG by making use of the fact that the kernel is acyclic. For example, Figure 4.1(c) shows a test generation model (TGM) consisting of a reduced iterative array that is sufficient for any fault in the kernel. The TGM can be further condensed based on the compacted schedule that will be used for test application, as described in Section 4.3.

4.2 Optimal Test Scheduling

A given test sequence for a partial scan design whose kernel is an acyclic structure of depth d may consist of up to $d+1$ time frames. Our objective is to find a way of compacting the patterns in a test sequence so as to minimize the number of time frames at which new data needs to be applied. This ensures that when the test patterns are applied using the scan path, the shifting time (which usually dominates the test time) is minimized. Assume that the time frames are numbered from 0 (the earliest) to d (the latest, at which time the fault gets detected). We define the **schedule** as the list of time frames in the test sequence that require new data to be shifted in, in ascending order. Thus a schedule $(0, 1, 2, \ldots, d)$ means that new data is shifted in at every time frame, while (0) represents a single-pattern test. In the example of Figure 4.1 presented earlier time frames 0 and 1 are combined together, hence the schedule is $(0, 2)$.

We shall refer to each element in the schedule as a **shift step**, since in the corresponding time frame a new pattern needs to be shifted into the scan path, and each element not in the schedule as a **hold step**, since it requires the contents of the scan path to be held for an additional clock cycle. Note that the test pattern scanned in during a given shift step i in the schedule $(\ldots, i, j, \ldots)$ is actually the result of compacting the test patterns for time frames i, $i+1$, $i+2$, $\ldots$, $j-1$.

4.2.1 The Compaction Principle

In our earlier example using Figure 4.1, we combined the test patterns for time frames 0 and 1 because neither of the inputs at A or B need to have a value specified in *both* frames. This compaction applies to all test sequences in this example. Before studying more complex cases we state the following principle that governs our compaction problem. The term *minimal test sequence* refers to a test sequence in which all unspecified input values are left as don't-care values.

Compaction Principle: *A set of consecutive time frames in a minimal test sequence may be compacted together into a single shift step in a schedule only if no input is required to be assigned values in more than one of these time frames.*

We will apply this compaction principle before test pattern generation is actually carried out. Note that the principle does not make use of the actual values of the test patterns; it uses only the information, derived from the circuit structure, about which input values can be specified and which must be don't-cares in various time frames. In a single-output acyclic structure, such as our previous example, the required information about the input values can be derived using the following rule: An input can be assigned a value in time frame x if there exists a path from that input to the output that passes through $d - x$ FFs (assuming the error is first observed at the output in time frame d). Later in this section we describe how to determine an optimally compacted schedule that satisfies the compression principle based on this information.

4.2.2 Modeling Schedule Constraints

Let us now consider the multi-output structure in Figure 4.2 in which blocks A, B, C, etc. are combinational and unlabeled blocks are registers. Its depth d is 4 and it has three inputs and four outputs, all of arbitrary width. As before the inputs and outputs are accessible only through a scan path which is not shown. Given an arbitrary fault in the circuit, a test sequence may propagate the fault to any of the outputs. At some outputs it may be possible to detect a fault at a time frame earlier than d. However, note that the same test sequence displaced in time can be used to detect a fault at different time frames. Hence we shall assume without loss of generality that a fault is to be detected at time frame d. This is justified since any fault that is propagated into the scan path at time frame d will be observed during the first shift step for the subsequent test sequence. This assumption will lead to a simplified test generation model discussed in Section 4.3.

The patterns shown at each input in Figure 4.2 indicate the time frames at which an input may possibly need to be specified in order to detect a fault at one of the outputs at time frame d. This information is determined in the same way as

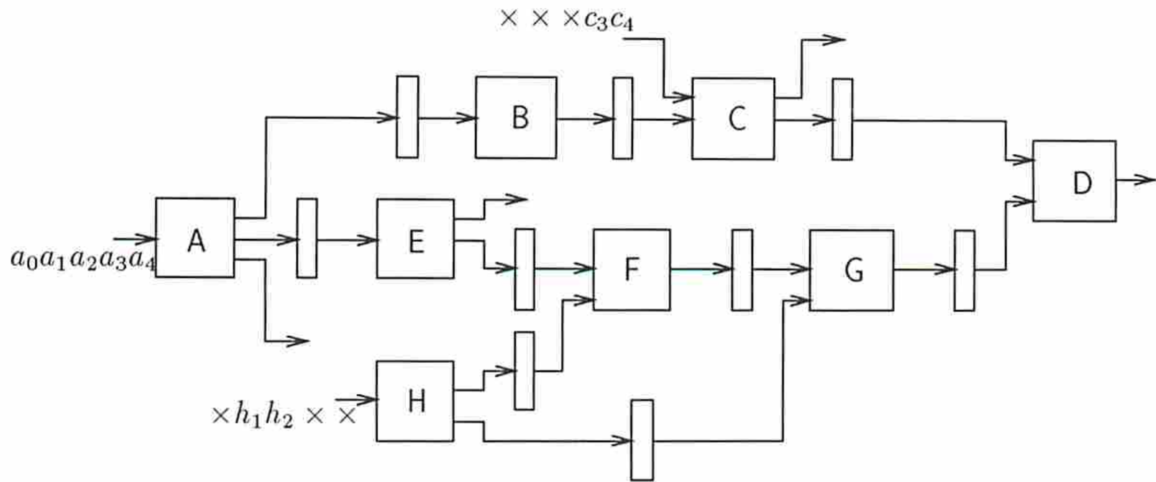


Figure 4.2: Example of acyclic kernel.

in the single-output case, except that for a given input, paths to all outputs have to be taken into account. Thus for example the input to C has values at time frames 3 and 4 but is always unspecified at all other times. The input to A may need specified values at any time frame, because corresponding to each time frame there is some path with the appropriate number of FFs ending in time frame d at one of the outputs. Hence it appears at first glance that the compaction principle will not allow a reduced test schedule.

However, to detect any fault it is sufficient to propagate it to just one of the outputs. If some test sequence for a fault propagates an error to more than one output, this implies that there may exist a reduced form of the same sequence that propagates it to only one output. Table 4.1 shows, for each output, what input values need to be specified such that the fault is observed at that output at the end of the 4th time frame. The table shows that no test sequence would require all 5 values at A to be specified. Also it is clear that all 5 frames cannot be compacted together, since the output at D (which we shall refer to as D for short) requires two different input values at A and also two different input values at H. Under these constraints it seems intuitively clear that a bare minimum of two shift steps will be needed in the schedule for an arbitrary test sequence in order to satisfy the compaction principle stated earlier. A model for representing the schedule constraints is described below.

Output	Input values on which the output depends		
	Sequence at A	Sequence at C	Sequence at H
A	$\times \times \times \times a_4$	—	—
C	$\times \times a_2 \times \times$	$\times \times \times \times c_4$	—
D	$a_0\, a_1 \times \times\times$	$\times \times \times c_3 \times$	$\times h_1\, h_2 \times \times$
E	$\times \times \times a_3 \times$	—	—

Table 4.1: Relationships among inputs and outputs.

Given an input x and an output y, let $\sigma(x, y)$ be defined as the ordered list of time frames at which the input sequence at x for output y can have specified values. Thus for example Table 4.1 shows that $\sigma(A, D) = (0, 1)$ and $\sigma(H, C) = (\,)$. $|\sigma(x, y)|$ denotes the number of elements in $\sigma(x, y)$. We shall attempt to find a minimal schedule by constructing a **schedule constraint graph** (SCG). We define an SCG as a directed graph $G = (V, A)$ where $V = (0, 1, 2, \ldots, d)$ represents the set of time frames and an arc (f_1, f_2) in A implies that frame f_1 must occur strictly before frame f_2 in any compacted test sequence. An SCG is constructed using the following procedure, which takes as input the values $\sigma(x, y)$ for all inputs x and all outputs y.

procedure constructSCG (σ): Returns schedule constraint graph, $G = (V, A)$.

{
  $V \leftarrow \{0, 1, 2, \ldots, d\}$, where d = depth of the circuit;
  $A \leftarrow \{\,\}$;
  For all input-output pairs (x, y) of the circuit such that $|\sigma(x, y)| \geq 2$ do:
  /* Add constraints corresponding to this input-output pair */
  {
    $L \leftarrow \sigma(x, y)$;
    While $|L| \geq 2$ do:
    {
      $i \leftarrow$ first element of L;
      $j \leftarrow$ second element of L;
      Remove i from L;

```
      /* Time frames i and j cannot be compacted together */
      For each k, 0 ≤ k ≤ i, do:
        A ← A ∪ {(k, j)};
      For each k, j < k ≤ d, do:
        A ← A ∪ {(i, k)};
    }
  }
}
```
□

For the circuit of Figure 4.2 the construction of the SCG is illustrated in Figure 4.3. We begin with the set of nodes $V = \{0, 1, 2, 3, 4\}$ and no arcs in A. Referring to Table 4.1, there are two input-output pairs that may contribute to arcs in the SCG: $\sigma(H, D) = (1, 2)$ and $\sigma(A, D) = (0, 1)$. The fact that $\sigma(H, D) = (1, 2)$ implies that there must be a shift step separating time frames 1 and 2 since distinct test patterns may be required at input H. In terms of constraints on the schedule, this implies that:

1. all time frames up to and including time frame 1 must occur before time frame 2 in the schedule; and
2. all time frames including 2 and beyond must occur after time frame 1 in the schedule.

The first item above contributes arcs (0, 2) and (1, 2), while the second contributes arcs (1, 3) and (1, 4). Thus the constraints due to the input-output pair (H, D) translate into the arcs shown in Figure 4.3(a), which is the result of the first iteration of the outer 'for' loop in the procedure. In the second iteration the constraints due to the pair (A, D) are added, resulting in the completed SCG shown in Figure 4.3(b). Note that in the above example it is not sufficient to have only the arcs (0, 1) and (1, 2) in the SCG. By adding the other arcs we are explicitly encoding the fact that although some time frames represented by nodes in V may be compacted with others, they can never be scheduled in reverse order.[1]

[1] With these constraints encoded explicitly, our problem is actually a special restriction of the equal execution time job scheduling problem [24][p. 402] with the number of processors not less than the number of jobs.

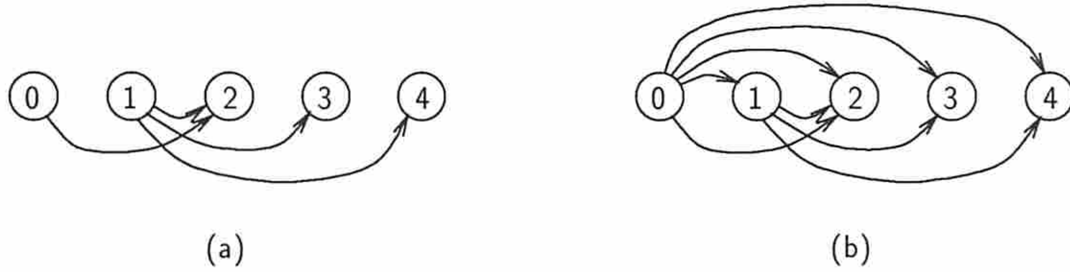


Figure 4.3: Construction of schedule constraint graph. (a) Constraints for (H, D) only; (b) constraints for (H, D) and (A, D).

In the procedure **constructSCG**, the outer 'for' loop may be repeated for all N_I inputs and all N_O outputs. Within the loop the time complexity is $O(d^2)$, hence the overall time complexity is $O(N_I N_O d^2)$.

4.2.3 Picking a Schedule

The SCG is essentially a representation of information on which time frames may be compacted together and which may not. Based on the SCG we are in a position to make the following statements about the schedules resulting from compaction.

Lemma 5 *Given a compacted schedule* $S \equiv (f_1, f_2, \ldots, f_n)$ *where* $0 = f_1 < f_2 < \cdots < f_n \leq d$, *$S$ satisfies the compaction principle if for every arc* (a, b) *in the SCG there is some* f_i *in* S *such that* $a < f_i \leq b$.

Proof Assume that for all arcs (a, b) in the SCG, there is some f_i in S such that $a < f_i \leq b$. Assume for the purpose of contradiction that the compaction principle is violated by S. Then there must be some input of the circuit that needs distinct values in some time frames a and b that are compacted into the same shift step in S. This implies that the SCG has an arc (a, b). But the fact that a and b are in the same shift step also implies that there is no f_i in S such that $a < f_i \leq b$, which is a contradiction. □

The above lemma essentially means that a given schedule S is valid, i.e., does not violate the compaction principle, if no two time frames that have an arc between them in the SCG are merged within the same shift step.

Lemma 6 *The number of nodes in the longest directed path in the schedule constraint graph is a lower bound on the number of steps in any schedule that satisfies the compaction principle.*

Proof Let P be a longest path in the SCG and let it consist of the δ nodes f_1, f_2, ..., f_δ in sequence. From the construction of the SCG, every arc (f_i, f_{i+1}) implies that if the frames f_i and f_{i+1} were compacted into the same shift step, the compaction principle would be violated. Hence there cannot be less than δ shift steps in any valid schedule. □

In Figure 4.3(b) the path consisting of nodes 0, 1, 2 is the longest, hence at least three shift steps are required in any compacted schedule.

Our problem is now to find a schedule $(f_1, f_2, \ldots, f_\delta)$ of minimum length that satisfies the following condition: given any arc (a, b) in A, the nodes a and b must not be compacted into the same shift step in the schedule, i.e., there must be an f_i in the schedule such that $a < f_i \leq b$. We present below a greedy algorithm that achieves the lower bound of Lemma 6. It essentially places the nodes of the SCG in levels such that the nodes in the longest path lie in consecutive levels, and all arcs that begin at a particular level end at some higher-numbered level. Then all nodes (time frames) at the same level can be compacted into the same shift step in the schedule.

algorithm schedule ($G = (V, A)$: schedule constraint graph): Returns S, a schedule of minimum length satisfying G.

```
{
  l ← 0;
  While |V| > 0 do:
  {
    l ← l + 1;
    R_l ← nodes in V having no incoming arcs;
    /* R_l consists of consecutively numbered time frames starting with the
         lowest-numbered time frame in V; see proof of correctness */
    Remove the nodes in R_l, along with adjacent arcs, from G;
```

}
/* Final value of l represents number of steps in schedule */
Return schedule $S = (m_1, m_2, \ldots, m_l)$ where
m_i = lowest-numbered time frame in R_i, $1 \leq i \leq l$.
} □

The sets R_l determined by this algorithm for the SCG of Figure 4.3 are {0}, {1} and {2, 3, 4}, hence the schedule is (0, 1, 2). The computation involved in computing R_l in each iteration is of order $O(d^2)$ assuming that an adjacency matrix is used to represent the SCG. Since the number of iterations is bounded by d, the overall complexity is $O(d^3)$. Below we demonstrate that the algorithm **schedule** works correctly in all cases.

Proof of Correctness We need to prove two assertions: first, that S is a schedule satisfying the compaction principle; second, that the resulting schedule is optimal.

Consider the first iteration of the 'while' loop. By construction, the lowest-numbered node in G (i.e., 0 for $l = 1$) cannot have incoming arcs, hence it must be included in R_l. Let this node be r_1. Let the highest-numbered node in R_l be r_2. We will now show that all nodes r such that $r_1 < r < r_2$ must be in R_l. Assume that there is in fact a node v, $r_1 < v < r_2$, that is not in R_l. Then there must be a node $u < v$ with an arc (u, v) in G. Then by construction of G, u must have outgoing arcs to all nodes $v' \geq v$. Hence there must be an arc (u, r_2) in G, which is a contradiction since r_2 is in R_l. Thus R_l represents a group of consecutively numbered time frames starting with the lowest-numbered one currently in V.

After the nodes in R_l are removed from G, the resulting graph is similar in form to G since only a consecutive set of lowest-numbered nodes has been removed. Hence the arguments above can be applied recursively to the resulting graph for the subsequent iterations. Thus every set R_l consists of consecutive time frames. Note also that for any arc in G, the two adjacent nodes cannot be in the same R_l. From Lemma 5 it follows that S is a valid schedule satisfying the compaction principle.

Since all nodes with no incoming arcs are removed in each iteration of the 'while' loop, the length of the longest path must decrease by 1 each time. Thus the

number of iterations is equal to the number of nodes in the longest path. According to Lemma 6, this is in fact a lower bound on the number of steps in any valid schedule. Hence the schedule S returned by the algorithm is optimal. □

In this section we have shown how to determine an optimally compacted schedule based on the structure of the acyclic circuit under test. This schedule can be utilized in two ways. First, it can be used in conjunction with a traditional sequential ATPG program to compact each test sequence produced before random data is used to fill in unspecified input values. In the sequences produced by ATPG, the time frame at which the fault is detected may be treated as frame d, and the sequence can be compacted according to the schedule. Some test sequences produced by ATPG may propagate a fault effect to more than one output. Such sequences should be preprocessed by selecting any one of those outputs and then forcing any input value to don't-cares if the value in that time frame does not influence the selected output at the time of detection.

The second and more efficient way to utilize the schedule is to use it as a guide for test generation itself. In the following section we will show how to construct a restricted test generation model to replace the traditional iterative array used in sequential ATPG. Test generation on this model will directly result in compacted test sequences for the desired schedule.

4.3 Test Generation Model

We now turn to the problem of test pattern generation for an acyclic structure. Given an optimized schedule with the smallest number of shift steps, we shall use it to influence the test generation process and simplify it if possible.

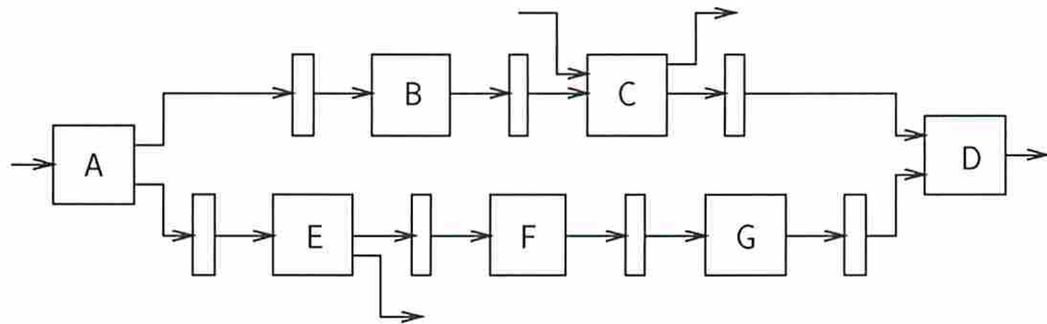
In test generation for general cyclic circuits, sequential ATPG programs typically construct an iterative array containing repeated copies of the circuit in order to represent the behavior of the circuit in different time frames [1]. With cyclic circuits the size of the iterative array required to detect an arbitrary fault may grow exponentially with the number of FFs in the circuit. However, in an acyclic circuit every irredundant fault must be detectable within $d + 1$ clock cycles, where d

is the depth of the structure, and the complexity of the test generation process is comparable to that for combinational circuits [20]. In fact a simple combinational **test generation model** (TGM) can be derived from the circuit structure, and any combinational ATPG program capable of dealing with multiple faults can be used. Not only is a sequential ATPG program unnecessary, this also avoids any execution overhead caused by the need to maintain iterative arrays of various lengths.

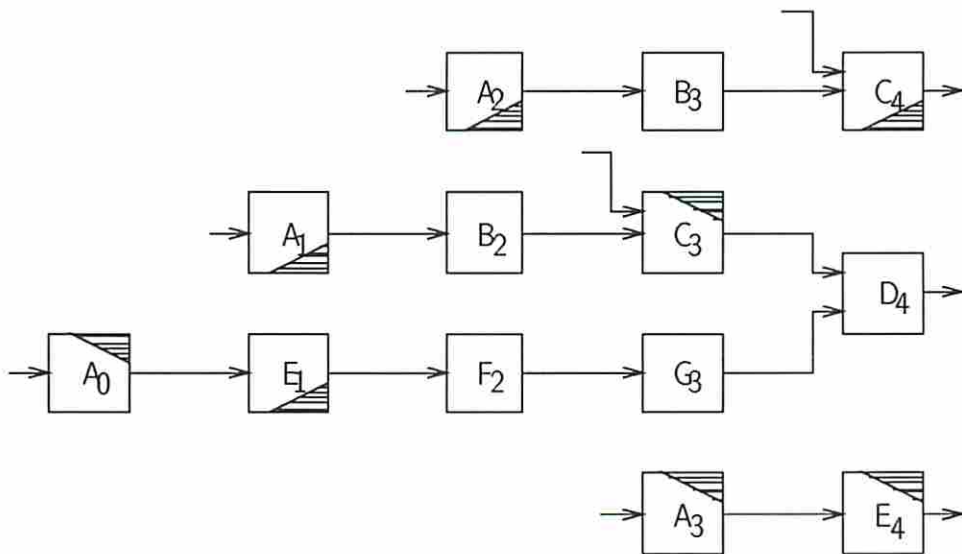
The concept of combinational TGMs is illustrated in Figure 4.4. Figure 4.4(a) shows a simplified version of the structure in Figure 4.2. It has three outputs at C, D and E respectively. We assume that any fault under test will be detected at one of the outputs at time frame 4. Figure 4.4(b) shows all outputs placed in time frame 4, and the portion of the circuit that feeds each output is laid out in a levelized fashion corresponding to the time frames. Blocks that are required to be in more than one time frame are replicated; thus for example A occurs at several different time frames in the expanded structure since the output values may depend on the behavior of A in various time frames.

Each copy of a repeated block has been pruned to remove any logic that will not be used for test generation; this is indicated by shaded regions but will not be explicitly shown from now on. The subscripts on A_0, E_1, etc. refer to the time frames in which the corresponding instances of the logic blocks exist; the highest subscript is clearly the depth $d = 4$. All registers in the expanded structure have been replaced by wires and the resulting TGM is combinational. This is the general form of the TGM before any compaction; we shall refer to it as the **basic TGM** and it represents the schedule $(0, 1, 2, \ldots, d)$.

To generate a test for a fault in the original sequential circuit, the fault must first be mapped to the set of corresponding fault instances in the combinational TGM. (This is analogous to the modeling of faults in iterative arrays.) Ordinary combinational ATPG can now be carried out on the TGM. The test pattern obtained can be transformed into a test sequence for the sequential circuit using the following rule: all input patterns at logic blocks with subscript i must be used as the ith pattern in the test sequence. This of course applies if the schedule $(0, 1, 2, \ldots, d)$ is used with no compaction.



(a)



(b)

Figure 4.4: Basic test generation model. (a) Acyclic structure; (b) basic TGM.

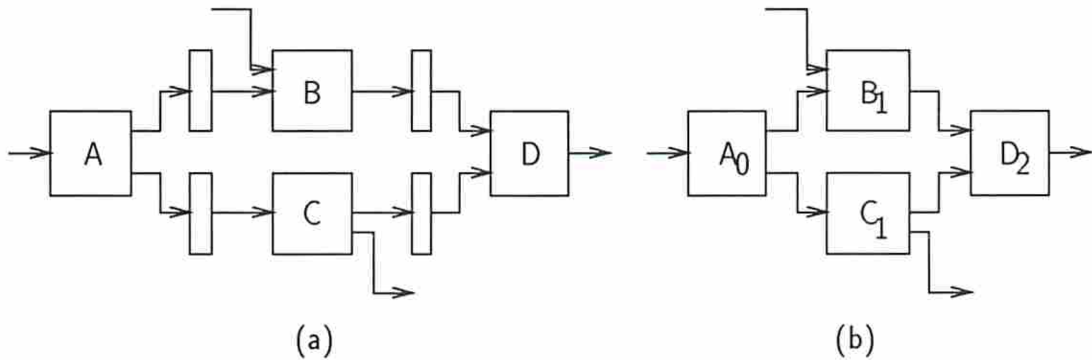


Figure 4.5: TGM for balanced structure. (a) Balanced structure, (b) combinational equivalent.

4.3.1 Condensing the Test Generation Model

When the test schedule is compacted as described in Section 4.2, not only is the test time per test sequence minimized, but we can also take advantage of the compacted schedule to condense the TGM. For the special case of *balanced structures* [25] such as the one shown in Figure 4.5(a) it has been shown that a single pattern is always sufficient for detecting any fault, i.e., the optimal schedule is always (0). The TGM for this class of structures is simply the combinational equivalent of the structure formed by replacing all FFs by wires as shown in Figure 4.5(b). Thus each logic block appears only once in the TGM, and only single faults need to be considered during combinational ATPG. However, this is not the case with general unbalanced structures. Given the schedule to be used, we shall show how a maximally condensed TGM can be derived. We shall prove that provided the schedule satisfies the compaction principle, the condensed TGM is sufficient for complete test pattern generation.

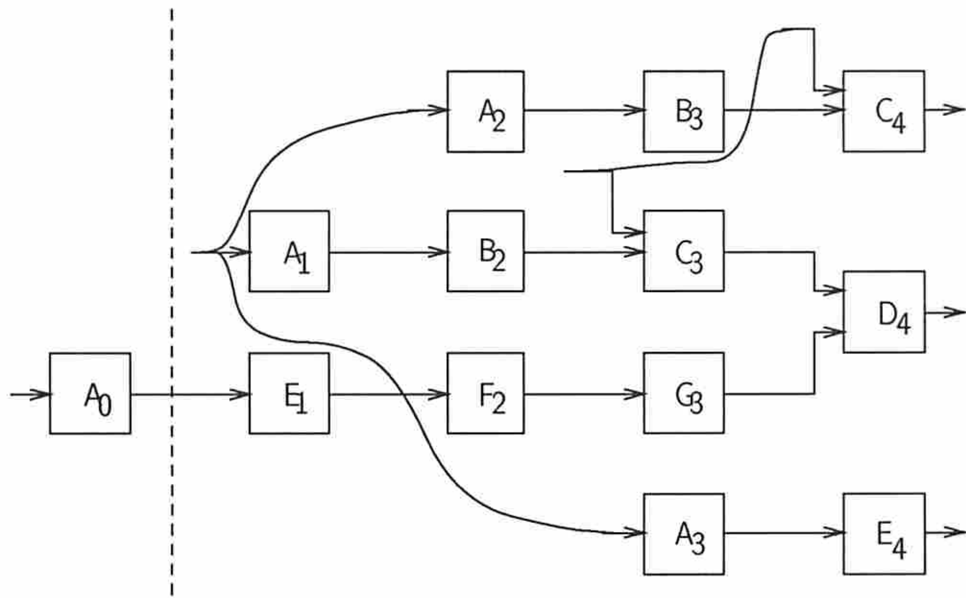
In condensing the TGM we begin with the basic TGM for the schedule $(0, 1, 2, \ldots, d)$ and modify it based on the schedule provided. We essentially utilize the fact that each input pattern applied to the kernel at a shift step in the schedule is also applied during the subsequent hold steps. The condensation process can be carried out by repeating the following two steps which are illustrated in Figure 4.6 for the circuit of Figure 4.4. The term **schedule interval** refers to a shift step and its subsequent hold steps.

Step 1: Repeat the following for each schedule interval. If any input signal occurs at more than one time frame in the same interval, connect the different copies together by fanning out the earliest copy of the signal (i.e., the one occurring in the lowest-numbered time frame) to the other copies so that only one copy of the input signal remains within the interval.

For example, consider the circuit of Figure 4.4(a) for which $(0, 1)$ is an optimal schedule. Figure 4.4(b) shows the basic TGM which is to be condensed. The schedule $(0, 1)$ has only one interval containing more than one step, namely the one consisting of time frames 1, 2, 3 and 4. In this interval the input feeding logic block A occurs three times, hence these inputs are connected together as shown in Figure 4.6(a). Similarly the inputs to C in time frames 3 and 4 are connected together. Note that A_1, A_2 and A_3 now receive identical inputs and in fact they represent exactly the same behavior extended over three clock cycles. The following operation will replace them with one merged copy in A_1.

Step 2: Repeat the following operation until no further changes can be made in the TGM. Let β be a logic block in the circuit and let $\beta_{i_1}, \beta_{i_2}, \ldots, \beta_{i_n}$ be different copies of β such that $i_1 < i_2 \cdots < i_n$ and every signal feeding an input of β_{i_1} also fans out to the corresponding input of each of $\beta_{i_2}, \ldots, \beta_{i_n}$. Then remove $\beta_{i_2}, \ldots, \beta_{i_n}$ from the TGM and fan out each output of β_{i_1} to all the signals originally fed by the corresponding outputs of $\beta_{i_2}, \ldots, \beta_{i_n}$.

At first this step can be applied to remove A_2 and A_3 and fan out the output of A_1 to B_3 and E_4 as well. Because block A must have exactly the same behavior in time frames 1, 2 and 3, we have simply combined the three copies for the purpose of ATPG, and fanned out the outputs appropriately. Note that this does not alter the execution of the test in any way; it only incorporates some information present in the test schedule into the test generation process, reducing the amount of analysis carried out during ATPG. In the resulting structure both B_2 and B_3 are fed by A_1, hence they can be merged into B_2. Finally in a similar way C_4 can be merged into



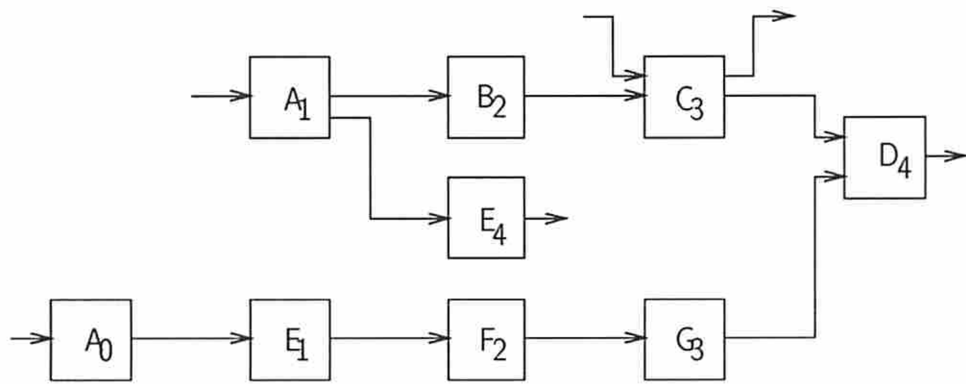


Figure 4.6: Condensation of test generation model. (a) Step 1, (b) step 2.

C_3. No further merging is possible and the final condensed TGM for the schedule (0, 1) is shown in Figure 4.6(b).

The condensation steps can be applied to any basic TGM, given a schedule, to yield a condensed TGM. Thus for the balanced acyclic structure of Figure 4.5(a), with optimal schedule (0), using steps 1 and 2 we obtain the simple combinational equivalent of Figure 4.5(b).

The computation complexity of the condensation steps depends on the implementation and on the level of description of the circuit. The computation in Step 2 can be minimized by forming maximally connected clusters of combinational logic blocks, and carrying out the circuit manipulations on this form of the circuit. Later the lower-level logic descriptions can be filled in for each high-level block, pruned where necessary as described earlier.

4.3.2 Test Pattern Generation

Given an acyclic structure C, a schedule $S = (\phi_1, \phi_2, \ldots, \phi_n)$, and a condensed TGM T_C for C based on the schedule S, the following procedure can be used to generate a test for an arbitrary fault f in C. Let f_C be the corresponding fault (possibly a multiple fault since some logic may be replicated) in T_C. Let us assume that f_C is detectable in T_C; then ordinary combinational ATPG can be used to derive a test pattern for f_C in T_C. Note that due to the nature of the condensation process, any given input signal to C can occur in T_C only at time frames $\phi_1, \phi_2, \ldots, \phi_n$, etc. in S. For each ϕ_i, let p_i be an input pattern containing the values of all inputs that occur in time frame ϕ_i in T_C, and containing don't-care values for all other inputs. Then the sequence of patterns $(p_1, p_2, \ldots, p_n)$, if applied according to the schedule S, will detect f in C.

To justify the use of the condensed TGM we need to validate the assumption that if f is detectable in C then f_C is detectable in T_C. This is done by the following theorem.

Theorem 3 *Given a fault f detectable in an acyclic circuit C, and given a schedule S that satisfies the compaction principle, the corresponding fault f_C (possibly multiple) is detectable in the condensed TGM T_C.*

Proof Let T_B be the basic TGM of C and let f_B be the fault (possibly multiple) corresponding to f in T_B. Since f is detectable in C, f_B must be detectable in T_B using some test pattern τ_B. Suppose the error is propagated to output Ω_d in T_B. Then the cone of logic feeding Ω_d in T_B has certain input values in τ_B that constitute a sufficient test pattern τ'_B for f_B, irrespective of the other input values. Note that in τ'_B, no input signal takes on more than one distinct value within the same schedule interval, otherwise the compaction principle would be violated by the schedule S. Hence for every input signal in the condensed TGM T_C there is a unique value that can be applied to it in every schedule interval in order to simulate the behavior of T_B. Let the input pattern formed by these values be τ_C; note that it is a condensed form of τ'_B. Since τ'_B detects f_B, it must cause different output values at Ω_d in the good and faulty versions of T_B. Hence τ_C must cause different output values at Ω_d in the good and vaulty versions of T_C. Thus f_C is detectable in T_C. □

The above theorem proves that for any detectable fault in C, a test sequence that follows the schedule S can be generated using combinational ATPG on T_C. This leads to the following corollary.

Corollary *Given an acyclic circuit C and a schedule S that satisfies the compaction principle, a complete test pattern set for the condensed TGM T_C results in a complete test sequence set for C using the schedule S.* □

We have thus shown that the condensed TGM derived in this section is a sufficient and complete model for test generation. The size of this TGM is lower than the expanded iterative array used by traditional sequential ATPG programs. Hence some redundant computations during the test generation process are eliminated. If the schedule is minimal, the model guarantees that an arbitrary fault can be detected using the smallest possible number of shift steps.

It should be noted, however, that the condensed TGM may contain more than one copy of some logic blocks. Hence the memory required to store the TGM may be higher than that for storing a single copy of the original circuit. Since sequential ATPG programs typically use a single stored copy of the circuit to represent different time frames, the memory required for carrying out combinational ATPG on the condensed TGM may be higher than that required by sequential ATPG using the original sequential circuit. Each logic block could appear up to d times in the condensed TGM, where d is the depth of the circuit structure; thus in the worst case the TGM could be up to d times larger than the circuit.

4.4 Summary

In this chapter we have studied the problem of testing acyclic structures in partial scan designs. We have presented a new approach to test sequence compaction in which the objective function is the number of distinct patterns to be shifted into the scan path per test sequence. In our approach, each test sequence is compacted into the smallest number of patterns needed to be shifted into the scan path. This leads to the lowest test time to detect an arbitrary fault. An algorithm for determining the optimal schedule, based on the structure of the circuit, was presented.

We have also presented a specialized test generation model (TGM) for acyclic structures. Like the iterative array model, this model reduces the ATPG problem to that of combinational ATPG with multiple faults. A special feature of this model is that it uses the optimal schedule determined separately in order to derive a condensed TGM that is smaller than the iterative array used in conventional sequential ATPG. This leads to fewer redundant computations during ATPG. The optimal scheduling algorithm and the test generation model can be used to potentially reduce the testing costs in partial scan designs, especially for signal processing and pipelined data path circuits.

Chapter 5

Partial Scan Design of Circuits Containing Switches

"It's them as take advantage that get advantage i' this world."
—*George Eliot,* Adam Bede

5.1 Switches

Circuits that are designed using a top-down hierarchical approach are usually constructed such that they consist of functional blocks and registers that are connected to each other via MUXes and buses. This is especially true of data path and signal processing circuits. For example, a bus-based architecture is used in the PIRAMID silicon compiler developed by Philips [26]. Every PIRAMID design consists of several execution units, each implementing a specific function, and a number of buses through which they interact. The arbitration of the bus is carried out through control input lines, which are accessible through a logically separate control unit.

We shall refer to MUXes and buses whose control input lines are accessible during test as **switches**. The general form of a switch is shown in Figure 5.1. The condition on control lines is true of ordinary data path architectures with separate controllers, as for example the PIRAMID design style mentioned above, provided either the control lines are directly accessible or a *control scan path* is provided to

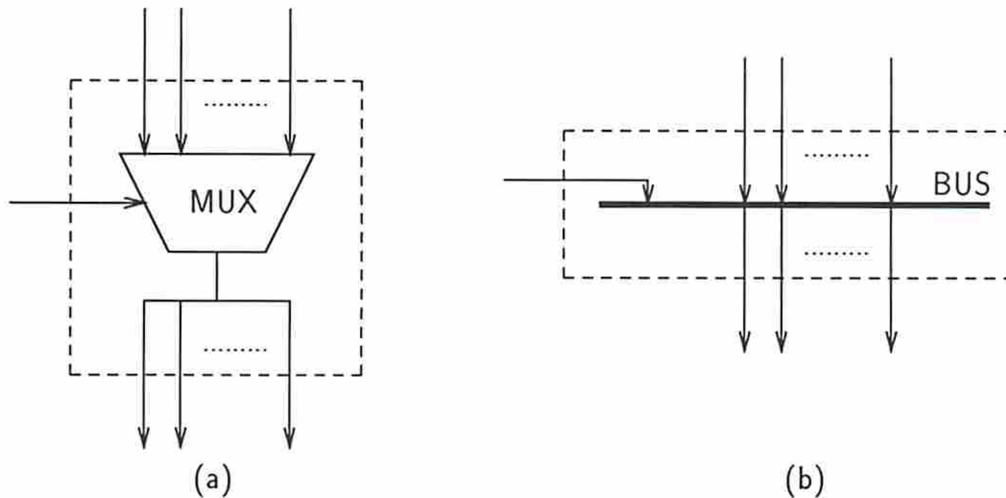


Figure 5.1: General form of a switch. (a) MUX, (b) bus.

access them. (In most designs a control scan path is also invaluable for testing the control unit, which is typically a finite state machine implementation and therefore hard to test functionally.) Switches can be used to improve the testability of the circuit in a variety of different ways. Some of these ways are listed below.

- The class of structures that have the nice testability properties of B-structures can be expanded if switches are present; this information can help to reduce design-for-test overheads.

- Switches can be used to set up data transfer paths through a circuit, increasing the controllability and observability of internal elements and thus reducing the complexity of ATPG.

- By setting every switch to a fixed configuration in each separate test session, the circuit under test can be implicitly partitioned for the purpose of ATPG, potentially reducing both ATPG complexity and overall test time.

Clearly, whenever switches are present in a circuit for functional purposes, they provide an opportunity to reduce the costs associated with testing the circuit. In this chapter the various benefits of switches listed above will be illustrated and studied in detail.

5.2 Circuit Model

In Section 3.2 we introduced a circuit model based on a *topology graph* (TG). The clouds of a circuit form the nodes of the TG and registers form the arcs. By merging all combinational logic, including MUXes and buses, into clouds, the computation complexity of the partial scan design algorithms of Chapters 3 and 4 is minimized. However, to take advantage of switches we need to modify the circuit model so that the information about these structures is not lost. The new model presented below allows switches to be represented as individual nodes in the graph.

5.2.1 Atomic Combinational Logic Units

In the topology graph (TG) circuit model of Section 3.2, all connected combinational logic regions are combined into a single *cloud*, which is the smallest circuit unit that is considered in the analysis. Each cloud becomes a node in the TG. In general a cloud may contain switches buried inside it along with random combinational logic. We need to modify the concept of a node so that the functional properties of switch logic are no longer ignored.

In the new model, a circuit consists of three types of elements: registers, switches, and random combinational logic. We need to construct a compact graph model that can be used for efficient circuit analysis. A new graph model derived from the TG model will be presented shortly; in this model, nodes represent switches as well as random combinational logic, and arcs represent delayless wires as well as registers. The important issue is identifying the set of nodes, from which the set of arcs will follow naturally.

We assume that the circuit under consideration is provided in the form of a register-transfer (RT) level description. In particular, (1) switches (MUXes and buses) should be clearly identified as such, and (2) wires, FFs, or switches that form an array at the RT level (e.g., a 16-bit bus that carries a binary value or a 32-bit register that stores a vector of status bits) must be appropriately identified. The remainder of the circuit may consist of arbitrary circuit elements, either combinational

logic (switches, gates, or blocks) or storage elements (FFs or registers). A typical circuit is shown in Figure 5.2(a). Each combinational block A, B, C has an associated gate-level structure that is not shown. To help construct a graph model, all non-switch combinational logic is partitioned into *atomic combinational logic units* (ACLUs) as follows. Each ACLU will form a node in the circuit graph.

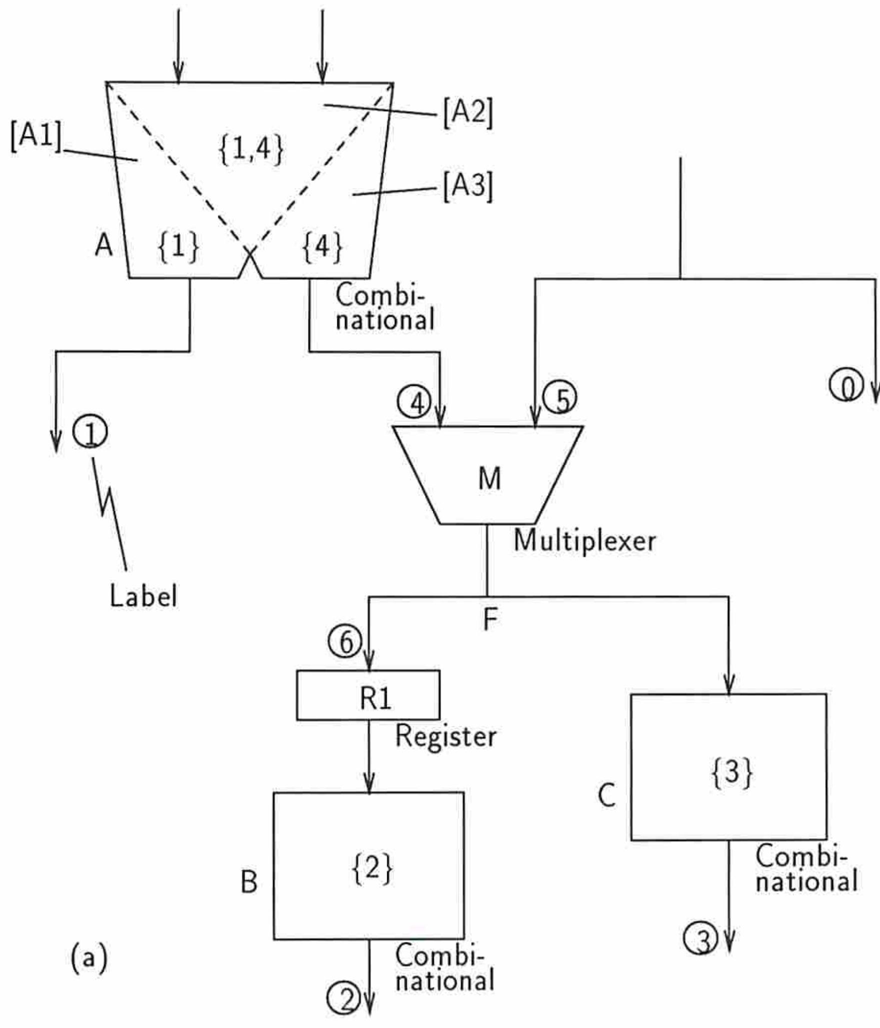
1. Assign labels to every circuit primary output, every register input and every switch input. Wires that belong to the same vector must have the same label. Wires that do not must have distinct labels. The labels in the circuit of Figure 5.2(a) are shown as circled numbers.

2. For every non-switch combinational element c in the circuit, construct a set of labels $label(c)$ such that a label l is present in $label(c)$ if and only if there is a path that starts at c, ends at a wire having label l, and passes through only non-switch combinational logic. Thus for example all gates in B are assigned the label set $\{2\}$. A has two outputs with different labels; hence some gates in A have label set $\{1\}$, some have label set $\{4\}$, and others have label set $\{1,4\}$.

3. Cluster the combinational elements into ACLUs such that every element in a given ACLU has exactly the same set of labels; i.e., c_1 and c_2 are in the same ACLU if and only if $label(c_1) \equiv label(c_2)$. Thus all gates in B form one ACLU, while A is subdivided into three different ACLUs A1, A2, A3 as shown.

In addition to the set of ACLUs derived above, two more types of ACLUs are defined. First, every switch (or array of switches) with externally accessible control input lines, such as M in Figure 5.2(a), is considered to be an ACLU. Second, if any wire in the circuit fans out to feed more than one ACLU, then a special type of ACLU called a **fanout node**, having one input and the appropriate number of outputs, may be constructed at the fanout point. In particular, if any switch or register has its output fanning out to more than one destination, for example the switvh M, there *must* be an explicit fanout node fed by the switch.

Any ACLU that has a primary input is referred to as a **primary input (PI) node**. Similarly, any ACLU that has a primary output is referred to as a **primary output (PO) node**.



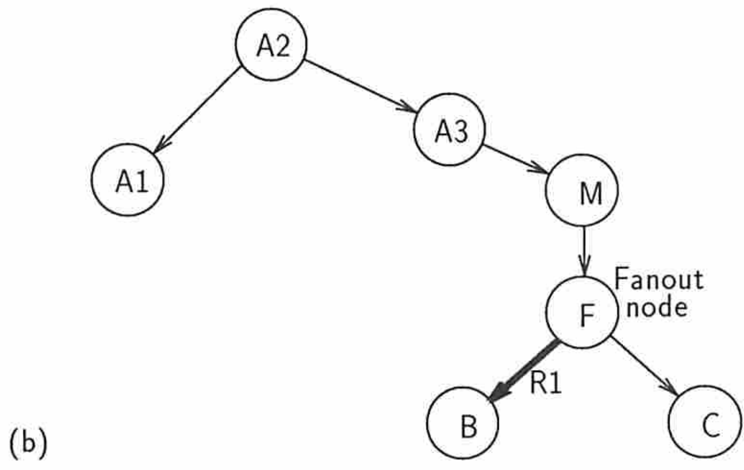


Figure 5.2: Generalized topology graph. (a) Circuit showing labels, (b) GTG with ACLUs as nodes.

5.2.2 Generalized Topology Graph

Definition A **connection** between two ACLUs is *either* a group of wires from an output port of one ACLU to an input port of the other ACLU *or* a register that is fed by one ACLU and feeds the other ACLU.

Thus each connection transmits data, either within a single clock cycle or with a delay of one or more clock cycles. We define the **type** of a connection as one of the elements $\{0, 1, v\}$. Simple wired connections are of type 0, LOAD registers are of type 1 (since data is transmitted with a delay of 1 clock cycle), and HOLD registers are of type v (since data is transmitted with a variable delay).

Definition A **generalized topology graph** (GTG) is a directed graph $G = (V, A, c, w)$. V is the set of ACLUs that includes all the combinational logic in the circuit. $A = \{a_i\}$ represents the set of *connections* between ACLUs, with each connection $a_i \in V \times V$ being either a group of simple wires or a register. $c : A \to \{0, 1, v\}$ defines the types of the connections in A. $w : A \to \mathcal{Z}^+$ (positive integers) defines the bit widths of the connections.

Thus the set of registers is $\{r \in A \mid c(r) \in \{1, v\}\ \}$. As before, we will use $w(r)$ to represent the cost of converting a register r into a scan path register. Note that between a given pair of nodes $u, v \in V$ there could be multiple arcs, representing different types of connections. Figure 5.2(b) shows the GTG of the circuit in Figure 5.2(a). The register R1 is represented by the arc from F to B.

Definition The **length** of an arc a in a GTG is 1 if $c(a) = v$, and $c(a)$ otherwise. The length of a path is the sum of the lengths of the arcs in it.

5.3 Switched Balanced Structures

In Chapter 3 we defined a class of easily testable structures called *balanced structures* (B-structures) and showed that they are single-pattern testable, and require only combinational ATPG. In that analysis the functional behavior of the combinational

logic was ignored. In this section we shall study how switches can implicitly partition a circuit such that a structure that is balanced in parts, but is unbalanced on the whole, can actually behave as a balanced structure for the purpose of ATPG. In the following discussions the set $SW \subseteq V$ will represent the set of switch nodes.

5.3.1 The Class of SB-Structures

Consider the structure shown in Figure 5.3(a). It consists of three B-structures B_1, B_2 and B_3 connected together via a switch as shown. Because B_1 and B_2 have different depths, the overall structure is not a B-structure. Hence the theorems of Chapter 3 do not apply to it. Now consider the operation of the circuit if the control line c were to be set permanently such that the switch always received its input data from B_1. In this configuration the circuit output would be independent of B_2. In fact B_2 would be logically removed from the circuit, and the circuit would behave as if it is balanced. Conversely, if the control line c were to be set to the opposite value, B_1 would be logically removed, and again the remaining structure would behave as a balanced structure. In the following analysis it will be shown that even when the control line is allowed to change freely, the entire structure is single-pattern testable using the combinational equivalent shown in Figure 5.3(b). Thus although this structure is not a B-structure, the presence of the switch gives it the properties of B-structures; we shall refer to it as a **switched balanced structure** or **SB-structure**.

We first define the class of SB-structures and then study its properties. Let S be an arbitrary synchronous sequential circuit, and let $G = (V, A, c, w)$ be a GTG of S in which every switch that has externally accessible controls is represented by a separate node in V. Let SW be the set of switch nodes. A **Hold register** refers to an arc $h \in A$ with $c(h) = v$. The following definition is derived from the definition of a B-structure in Chapter 3.

Definition S is an **SB-structure** if:

1. G is acyclic;

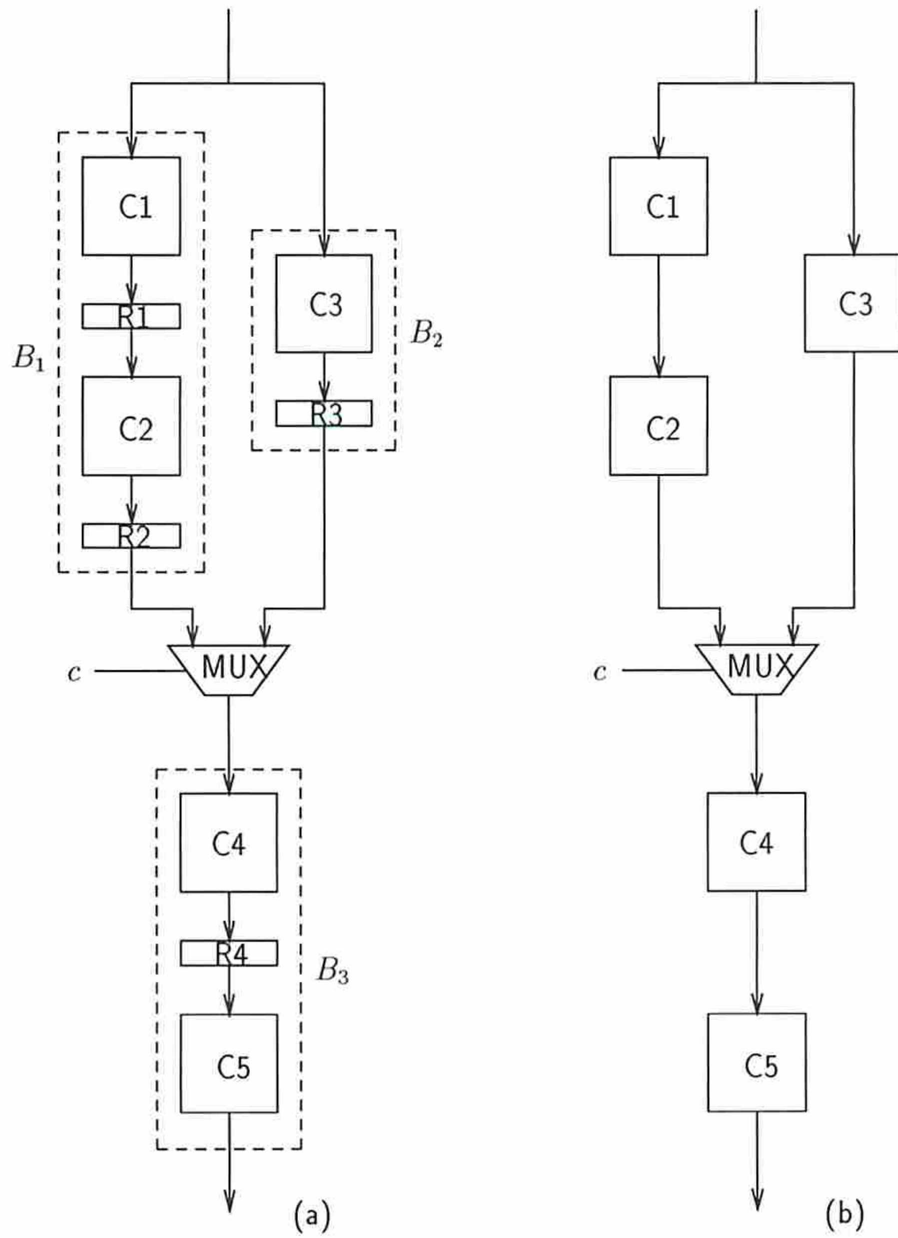


Figure 5.3: SB-structure example. (a) SB-structure, (b) combinational equivalent.

2. $\forall v_1, v_2 \in V$, all directed paths from v_1 to v_2 (if any) are of equal length *or* pass through the same switch node $v_s \in SW, v_s \neq v_1$;

3. $\forall v_1, v_2 \in V$, if any directed path from v_1 to v_2 passes through a HOLD register $h \in A$, then all such paths pass through h *or* all pass through the same switch node $v_s \in SW, v_s \neq v_1$. □

The SB-structure of Figure 5.3(a) satisfies this definition.

The **combinational equivalent** of an SB-structure is constructed in exactly the same way as for a B-structure in Section 3.3. Figure 5.3(b) shows the combinational equivalent of the SB-structure in Figure 5.3(a). Other terms associated with B-structures are also defined analogously for SB-structures.

5.3.2 Testability Properties of SB-Structures

SB-structures have exactly the same testability properties as those of B-structures established in Chapter 3; i.e., they are single-pattern testable (Theorem 1), and combinational ATPG is sufficient (Theorem 2). The proofs of the theorems, however, have to be modified.

Lemmas 1 and 2 apply to all acyclic structures including SB-structures. The proof of Theorem 1, however, is specific to B-structures. Below we re-state the theorem and indicate a modified proof. S^B denotes an arbitrary SB-structure and C^B denotes its combinational equivalent. Let the depth of S^B be d.

Theorem 4 *Every SB-structure is fully testable for all* detectable *faults using single-pattern tests.*

Proof We shall modify the procedure **transform** in the original proof into a procedure **transform′** which applies to SB-structures. Let $G = (V, A, c, w)$ be the GTG of S^B, and $SW \subset V$ the set of switches. Define H as the set of HOLD registers, i.e., $\{h \in A \mid c(h) = v\}$. Define the *state* G^t of S^B exactly as before.

The modified procedure **transform′** uses the same steps 1, 2, 3, 5, 6 and 7. The modified step 4 is listed below.

4. Repeat the following until $\mathcal{F} = \phi$:

   (a) Pick some $v \in \mathcal{F}$ having the highest activation time $\alpha(v)$, and remove it from $\mathcal{F}$.
   $k \leftarrow \alpha(v)$.

   (b) Construct a set $\mathcal{A}$ of arcs to be processed, as follows.
   If $v \in SW$, i.e., v is a switch, then use the control input value to the switch at clock cycle $\alpha(v)$ to determine which input connection a_{sel} is selected by the switch at this clock cycle, and set $\mathcal{A} \leftarrow \{a_{sel}\}$;
   else $\mathcal{A} \leftarrow$ {all arcs incident onto v}.
   For all arcs $a \in \mathcal{A}$ do the following.

      i. $u \leftarrow$ source node of a.
      If $c(u) = 0$ (i.e., u is not a register) $\alpha' \leftarrow k$
      else $\alpha' \leftarrow k - 1$.
      If u has been visited earlier (i.e., $\alpha(u)$ has been assigned a value) then $\alpha(u)$ must be equal to α'—see Claim 40(b)i below.
      Otherwise add u to $\mathcal{F}$, with $\alpha(u) \leftarrow \alpha'$.

      ii. If $a \in H$ and $h^k(a) = 1$ then

         A. $t \leftarrow$ min. j such that $h^{j+1}(a) = h^{j+2}(a) = \ldots = h^k(a) = 1$;
         If there is no such j, skip steps B and C.
         t represents the clock cycle during which node u is active, and register a loads the active data, in the original test plan.

         B. $\tau \leftarrow k - t =$ number of HOLD cycles of a in the current sequence.
         The next step eliminates the HOLD cycles while keeping the test valid.

         C. Let G_u be the subgraph of G consisting of all nodes from which u is reachable. (I.e., G_u is the *cone of influence* of node u.)
         Then any change in the input data and/or the state of the substructure G_u will not affect the outcome of the test, provided the values at u at clock cycle $\alpha(u)$ are unchanged—see Claim 40(b)iiC below.
         Hence we can modify the test by delaying all electrical activity within G_u by τ clock cycles as follows.
         $\forall v \in V$ in G_u, $I^j(v) \leftarrow I^{j-\tau}(v)$, $\tau < j \leq k$;
         $\forall h \in H$ in G_u, $h^j(h) \leftarrow h^{j-\tau}(h)$, $\tau < j \leq k$;
         $\forall a \in A$ in G_u, $x^j(a) \leftarrow x^{j-\tau}(a)$, $\tau < j \leq k$;
         $h^k(a) \leftarrow 0$.

It can easily be seen that the sequence of modified states $(G^1, \ldots, G^m)$ is still a valid test for f.

The preceding portion of the procedure transforms the test sequence such that in the final state sequence every register is in the LOAD mode at the time when the node driving it becomes active. Note also that every node v that is active during the test, there is a unique clock cycle $\alpha(v)$ during which it is active. (This is proved in Claim 40(b)i.) □

The correctness of the modified procedure depends on the two claims proved below.

Claim 40(b)i *In Step 40(b)i of procedure* **transform′**, *if u has been visited before (i.e., $\alpha(u)$ has been assigned a value) then $\alpha(u) = \alpha'$.*

Proof Assume, on the contrary, that u has been visited before and assigned $\alpha(u) \neq \alpha'$. Then there must be two paths of different lengths from u to z, both of which must pass through the same switch $w \in SW$ (by definition of SB-structures). Also, w must already have been visited according to the order of traversing nodes, and must have a well-defined $\alpha(w)$ value. Thus both paths from u to w, ending at different data inputs of w, must have been traversed. However, note that in the procedure **transform′**, only one incoming arc to a switch node is ever placed in the frontier $\mathcal{F}$ during the traversal. This is a contradiction. Hence the statement of the claim must be true. □

Claim 40(b)iiC *In Step 40(b)iiC of procedure* **transform′**, *any change in the input data or the state of the substructure G_u will not affect the outcome of the test, provided the values at u at clock cycle $\alpha(u)$ are unchanged.*

Proof Consider an arbitrary node u' in G_u. Since there are directed paths $u' \longrightarrow u$ and $u \longrightarrow z$, and the latter passes through the HOLD register h, there must be a path $u' \longrightarrow z$ passing through h. Assume that some changes in the values at node u' do not affect the values at node u at clock cycle $\alpha(u)$, but affect the outcome of the test. This implies that there must be another path $u' \longrightarrow z$ not passing through

h. Hence there must be a switch node $w \in SW$ through which both paths pass (by definition of SB-structures), and w must already have been visited according to the order of traversing nodes, and it must have a well-defined $\alpha(w)$ value. Thus both the distinct paths, $u \longrightarrow w$ and $u' \longrightarrow w$ ending at different data inputs of w, must have been traversed. This leads to a contradiction since w is a switch, and only one of its incoming arcs could ever have been placed in the frontier $\mathcal{F}$ during the traversal. □

The two claims above validate the procedure **transform**′, which completes the proof of Theorem 4 along the same lines as that of Theorem 1 in Chapter 3. □

Lemma 3 and Theorem 2 can be proved to hold for SB-structures by making use of Theorem 4 instead of Theorem 1 in their proofs.

Thus by proving that both the theorems on B-structures in Chapter 3 are also applicable to SB-structures, we have established that SB-structures have the same testability properties. The concept of SB-structures leads to two benefits. First, the overhead for partial scan design can be reduced; because of the switches, it is sufficient for the structure under test to be balanced in parts even if the overall structure is not a B-structure. Second, an SB-structure can be partitioned into smaller regions of logic that can be tested as independent kernels, thus lowering the overall ATPG cost. This aspect will be studied in Chapter 6.

5.4 Algorithm for Scan Register Selection

In Chapter 3 the BALLAST procedure for selecting scan path registers was presented. The selection was carried out in two steps using the topology graph (TG) circuit model. The first was to determine a minimal weight set of scan registers to make the kernel acyclic; the second was to select a minimal weight set of additional scan registers to make the resulting kernel balanced. Both steps are NP-complete. In this section we study the analogous problem of selecting a minimal set of scan registers using the generalized topology graph (GTG) model. The problem is stated as follows.

Let $G = (V, A, c, w)$ be the GTG of the circuit. We need to determine a set of scan registers $R \subseteq A$ such that the resulting kernel is an SB-structure and $\sum_{a \in R} w(a)$ is minimized. As before, we solve the problem in three steps.

Step 1. Remove a set of "feedback" arcs R_A from G such that $\sum_{a \in R_A} w(a)$ is minimized and the resulting kernel G_A is acyclic.

Step 2. Remove an additional set of arcs R_B from G_A such that $\sum_{a \in R_B} w(a)$ is minimized and the resulting kernel G_B is an SB-structure.

Step 3. $R = R_A \cup R_B$ is the desired set of arcs, and the resulting topology graph $G_B = (V,\ A - R,\ c,\ w)$ represents the kernel.

The steps above correspond to the steps in the original procedure, which applied to the TG model. Since the TG is a specialization of the GTG, both the subproblems are NP-complete in the GTG model also. We use the same basic solution approach and modify it for the new problem. The algorithms used in steps 1 and 2 for balancing the TG need to be extended in two ways so that they can be used for balancing the GTG.

1. In the TG model, every arc represents a register and hence a potential scan register, with the modification weight proportional to its width. In the GTG model, an arc could be a register or a simple wire. Only the former are candidates for being converted into scan registers. Hence the algorithms need to be modified to distinguish between registers and wires.

2. In the TG model, every node is a cloud of combinational logic, with no information about its functionality. In the GTG model, some of the nodes may be switches, and the concept of balanced structures is extended to the class of switched balanced structures. Hence the algorithms need to be extended also.

In the following discussion on the modified algorithms, the influence of both the issues above will be described.

5.4.1 Removal of Feedback Registers

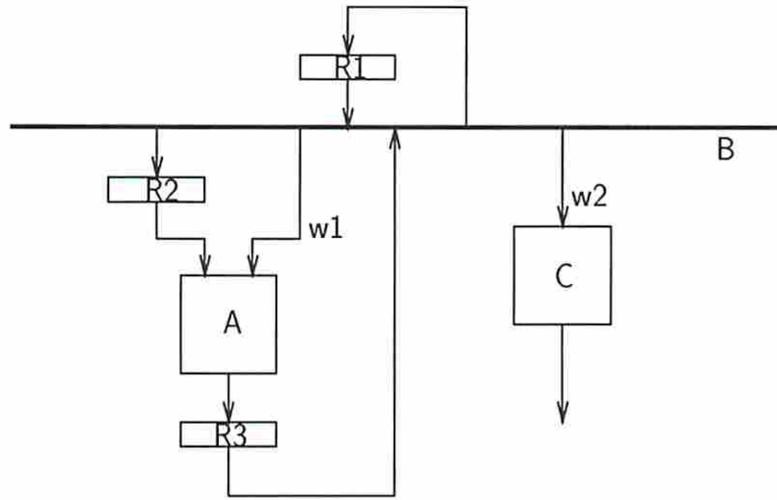
In Step 1 of the balancing procedure, we need to make a minimal-weight set of registers scannable such that the resulting kernel is acyclic. Since the arcs in the GTG $G = (V, A, c, w)$ consist of both registers and simple wires, it is not sufficient to find the minimum feedback arc set (MFAS) of G. Instead we transform G into another graph G_F such that the MFAS of G_F is indeed the required set of registers. In constructing G_F we use the following observation.

Lemma 7 *If $u, v \in V$, and $a \equiv (u, v)$ is a wire, i.e., $c(a) = 0$, then any register $a' \equiv (u, v)$ cannot be in a minimal-weight set of scan registers that makes the kernel acyclic.*

Proof Let R_1 be a register corresponding to arc $a' \equiv (u, v)$. Assume that R_1 is one of the registers in a minimal set of scan registers that results in an acyclic kernel. By adding R_1 to the kernel it must still be acyclic, since there is already a wire from node u to node v in the kernel. Hence the set of scan registers could not have been minimal, which contradicts the assumptions. $\square$

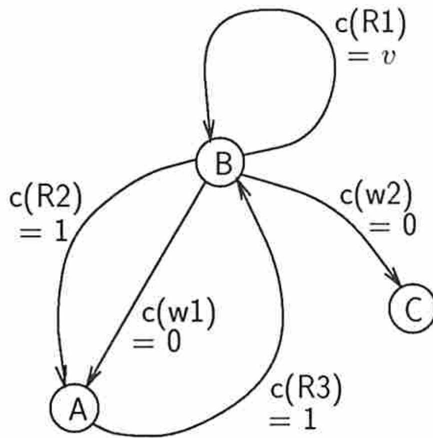
We can use this observation to construct the graph $G_F = (V_F, A_F, c_F, w_F)$ as follows. Let $V_F = V$ be the set of nodes. Construct two sets of arcs A_{F1} and A_{F2}. Let $A_{F1} = \{a \equiv (u, v) \in A \mid c(a) \in \{1, v\}$ and there is no wire between u and $v\}$. For all arcs (registers) a in A_{F1} set $c_F(a) = c(a)$ and $w_F(a) = w(a)$. Let $A_{F2} = \{a \equiv (u, v) \in a \mid c(a) = 0\}$. For all arcs (wires) a in A_{F2} set $c_F(a) = 0$ and $w_F(a) = \infty$. Then the arc set $A_F = A_{F1} \cup A_{F2}$ is the set of arcs of G_F. For example, Figure 5.4(a) shows a simple circuit in which B is a bus (i.e., a switch); Figure 5.4(b) shows its GTG G and Figure 5.4(c) shows the transformed graph G_F. Note that in G_F, all registers that cannot be in the set of scan registers due to Lemma 7 have been removed.

In G_F, all wires are represented as arcs having infinite weight. Thus any finite-weight set of feedback arcs represents a set of registers, and our problem is to find a minimum-weight set. Below we show that such a set must exist.

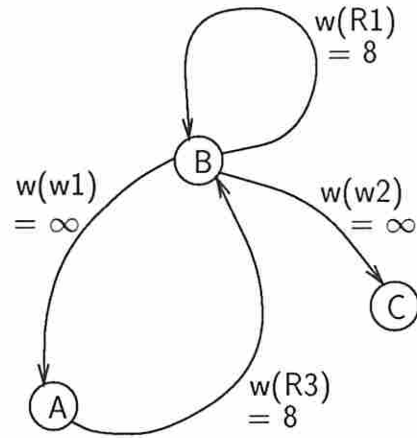


All connections have width = 8

(a)



(b)



(c)

Figure 5.4: Transformation of the GTG for feedback register analysis. (a) Circuit, (b) GTG G, (c) transformed GTG G_F.

Lemma 8 *For a* synchronous *circuit with GTG G and the transformed GTG G_F constructed as described above, G_F must have a finite-weight set of feedback arcs.*

Proof Assume for the purpose of contradiction that G_F has no finite-weight set of feedback arcs. Then there must be a cycle in G_F consisting entirely of infinite-weight arcs. Since each infinite-weight arc is derived from a wire connection, the original circuit must have a continuous cyclic path consisting of wires and combinational elements with no storage elements, i.e., an asynchronous loop. This is a contradiction since by assumption the circuit is synchronous. □

Thus we have transformed the GTG G into a new graph G_F to which the branch-and-bound algorithm of Section 3.6 can be applied. The MFAS obtained from this algorithm, denoted by R_A, corresponds to the registers that must be made scannable in order to eliminate all cycles in the kernel.

5.4.2 Balancing Acyclic Sequential Structures

The result of Step 1 described above is a set of arcs R_A representing scan registers. The corresponding kernel is $G_A \equiv (V_A, A_A, c_A, w_A)$ where $V_A = V$, $A_A = A - R_A$, and c_A, w_A are equivalent to c, w respectively restricted to the domain A_A. We use a heuristic **balanceSW**, derived from **balance** of Chapter 3, to carry out Step 2. As before we use a verification procedure **checkSW**, derived from **check**, which returns SUCCESS only if G_A is an SB-structure.

5.4.2.1 Verification Procedure

Like **check** in Chapter 3, **checkSW** attempts to levelize the GTG starting from various root nodes. Because arcs may represent different types of connections, the levelizing procedure is different. The level number of a node x is denoted by $l(x)$. An arc a leaving node x is also assumed to have level number $l(a)$ identical to $l(x)$. Every level number is either a nonnegative integer, if all paths from the current root to the current node/arc are of that length, or the symbol v, if there is no unique path length or if one of the paths includes a connection of type v (i.e., a HOLD register).

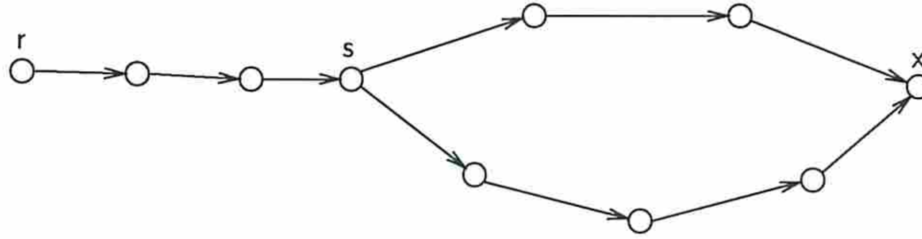


Figure 5.5: Illustration of **checkSW** procedure.

To deal with the presence of switches in SB-structures, **checkSW** has some enhancements. Unlike in B-structures, a node is allowed to have a variable level number, caused by unequal-length converging paths, provided all the paths pass through the same switch. For every node x, the procedure computes a set $SWP(x)$ of switches in SW such that *all* paths from the current root node to x pass through every switch in $SWP(x)$. These sets are useful in checking whether a given set of unbalanced paths is acceptable. (Note that the root node itself must never be included in $SWP(x)$ even if it is a switch; this follows from the definition of SB-structures.)

Another enhancement is in the set of starting points for the levelizing process. In **check** it is sufficient to levelize the input graph starting at each root node in turn. In **checkSW** it is also necessary to levelize starting at each switch. The need for this is shown in Figure 5.5, in which each arc represents a register. There are paths of unequal length between the switch node s and the node x. When the levelizing process starts at the root node r, the set $SWP(x)$ is computed as $\{s\}$, hence no violation is apparent at x. However, when the levelizing process is started at s, the set $SWP(x)$ is now computed as $\{\}$, and the unbalanced paths ending at x are detected as a violation. The set $LEVEL0$ is used to store the set of root nodes as well as switches that will be used as starting points for levelization.

In the listing below, for node x, $src(x)$ represents the set of source nodes for all incoming arcs to x, and $dst(x)$ represents the set of destination nodes for all outgoing arcs from x.

function checkSW $(G = (V, A, c, w))$: Returns SUCCESS if G is an SB-structure, FAILURE otherwise.

1. $LEVEL0 \leftarrow \{\text{root nodes of } G\} \cup SW$, where *root nodes* are defined as nodes with no incoming arcs.

   If $LEVEL0 \neq \phi$, proceed to the next step; otherwise return FAILURE since G has no root nodes and cannot be acyclic.

2. Pick a node r in $LEVEL0$; remove r from $LEVEL0$.

   $l(r) \leftarrow 0$;
   $SWP(r) \leftarrow \phi$;
   $L \leftarrow \{r\}$ (nodes currently levelized).

3. While $L \neq V$ (i.e., $L \subset V$) do the following.

   (a) Pick $x \in V - L$ such that $src(x) \subseteq L$.

   (b) $\forall a \equiv (u, x) \in A$:
   If $c(a) = v$ or $l(u) = v$ then $l(a) \leftarrow v$
   else $l(a) \leftarrow l(u) + c(a)$.

   (c) $SWP(x) \leftarrow \bigcap_{u \in src(x)} SWP(u)$;
   If $x \in SW$, $SWP(x) \leftarrow SWP(x) \cup \{x\}$.

   (d) If the $l(a)$ values computed above are all equal to some numeric value la (i.e., $la \neq v$), then $l(x) \leftarrow la$; else (imbalance is present) do the following:
   If $SWP(x) \neq \phi$ then
   (the unbalanced paths all pass through some switch) $l(x) \leftarrow v$,
   else return FAILURE.

   (e) Add x to L as a node that has been assigned a level.

4. If $LEVEL0 \neq \phi$ go to 2.

5. S is an SB-structure; return SUCCESS. □

Because of the manner in which level numbers are computed, the initial steps in **check** which analyze the connections of HOLD registers are not needed in **checkSW**.

5.4.2.2 Balancing Procedure

Like **balance**, the heuristic procedure **balanceSW** uses a mincut approach to recursively subdivide the GTG, balance the parts separately, and merge the results.

In order to modify **balance** to **balanceSW**, we need to consider two issues: the presence of both wires and registers as arcs in the graph model, and the extension of the class of B-structures to that of SB-structures.

In the earlier discussion on breaking feedback loops it was shown that by transforming the GTG containing wires and registers, that problem could be reduced to the one solved in Chapter 3. The same transformation is not appropriate for the problem of removing unbalanced feedforward paths. We use a similar transformation in which all wires are modified to have a width (i.e., cost) of infinity but all registers are retained with no change in their width. This ensures that every mincut of the GTG contains only registers. However, in this case it is not guaranteed that a finite-weight mincut, i.e., a cut consisting of only registers, exists after this transformation.

Case 1 In the case when a finite-weight mincut CS exists, this cut is used to partition the GTG exactly as in **balance**. The two induced partitions G_s and G_d are balanced separately (recursively). The resulting balanced sub-GTGs are then merged by reintroducing a subset of the mincut register arcs $CS' \subseteq CS$ into the result GTG such that it is an SB-structure, and the combined weight of the arcs in CS' is maximal. Unlike in **balance**, in **balanceSW** the set of registers reintroduced during merging can contain more than one HOLD register (this follows from the definition of SB-structures); hence the merging procedure does not distinguish between LOAD and HOLD registers.

Case 2 If no finite-weight mincut exists, i.e., the mincut CS contains one or more wires (infinite-weight arcs), clearly a path containing only wires from the inputs of the structure to its outputs must exist. Depending on the nature of the cut CS, one of two strategies is used.

> **Case 2(a)** If CS contains at least one register, the procedure first shrinks the cutset CS so that it contains only registers; i.e., $CS_R \leftarrow \{a \in CS \mid c(a) \neq 0\}$. The resulting set CS_R is no longer a cutset since at least one arc has been removed. Let G' be the graph formed by removing the arcs in CS_R from the original GTG. G' is balanced recursively; then, in

the resulting GTG, a maximal-weight subset $CS' \subseteq CS_R$ is reintroduced such that the GTG remains balanced.

Case 2(b) If CS consists entirely of wires, the procedure first removes *all* registers from the original GTG, resulting in a purely combinational structure. These registers are placed in the set CS_R. (Here the title CS_R is actually a misnomer since it has no relation to any cutset.) The procedure then determines a maximal-weight set of register arcs $CS' \subseteq CS_R$ that can be reintroduced into the GTG without causing an imbalance.

To illustrate the strategies used in Cases 2(a,b), consider the circuit in Figure 5.6. There is no cut consisting only of registers; the mincut consists of R1 and the wire from B to D, which has infinite weight. Using Case 2(a), R1 is tentatively removed from the structure and placed in CS_R. The balancing procedure is then invoked recursively on the remaining circuit. Within the recursive call, the mincut consists of the wire from B to D, which has infinite weight. (Note that in our analysis, the number *infinity* is treated as *a large finite number*.) Using Case 2(b), R2 is removed since its presence causes an imbalance. Returning from the recursive call, R1 is placed back from CS_R into the structure since it does not cause an imbalance. Thus the resulting solution requires R2 to be removed from the structure (i.e., made scannable) but not R1.

The overall organization of **balanceSW**, which is listed below, is similar to that of **balance** in Chapter 3. The sets CS and CS_R described in the above discussions are both represented by the same variable CS in the procedure listing, since they are never required at the same time and both need to be manipulated in the same way after being constructed.

function balanceSW ($G = (V, A, c, w)$: acyclic GTG): Returns $R \subseteq A$ such that $(V, A - R, c, w)$ is balanced.

1. If (**checkSW**(G) = SUCCESS) then return ($R \leftarrow \phi$), else proceed.

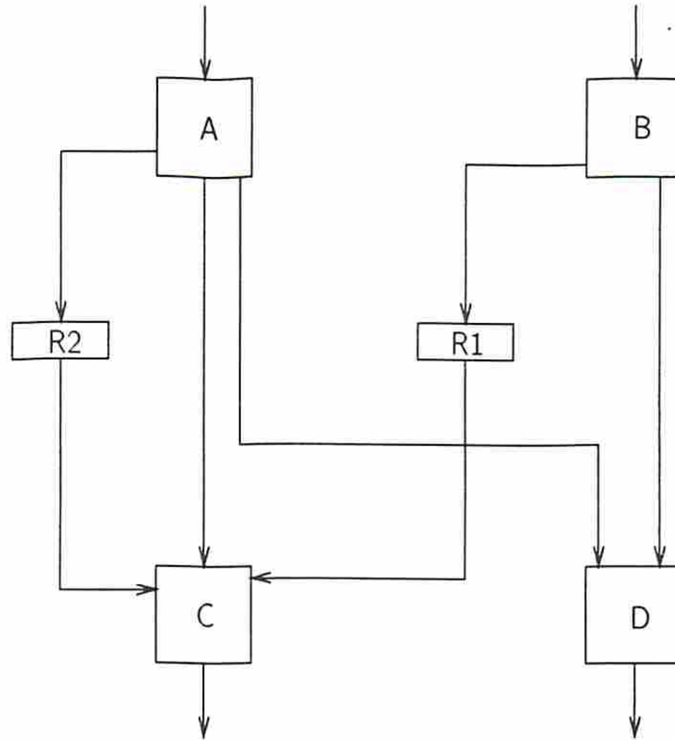


Figure 5.6: Illustration of **balanceSW** where no finite mincut exists.

2. Transform $G \equiv (V, A, c, w)$ to $G_u \equiv (V_u, A_u, c_u, w_u)$, in which all wire arcs have "infinite" weight, as follows. We use ∞ to represent some large finite number, $\infty > \max_{a \in A} w(a)$.

   $G_u \leftarrow G$;
   $\forall a \in A_u$, if $c_u(a) = 0$ then $w_u(a) \leftarrow \infty$.

3. $CS \leftarrow$ minimal cost cutset of G_u.

   If CS contains only registers (finite-weight arcs), do the following:

   (a) Determine G_s, G_d, the subgraphs of G_u induced by CS.

   (b) Balance G_s and G_d separately;
   $R \leftarrow$ **balanceSW**$(G_s) \cup$ **balanceSW**$(G_d) \cup CS$.

   Else if CS contains wires and at least one register, do the following:

   (a) $CS \leftarrow \{a \in CS \mid c(a) \neq 0\}$.

   (b) $G' \leftarrow G$ with registers in CS removed.

   (c) $R \leftarrow$ **balanceSW**$(G') \cup CS$.

   Else (CS contains only wires) do the following:

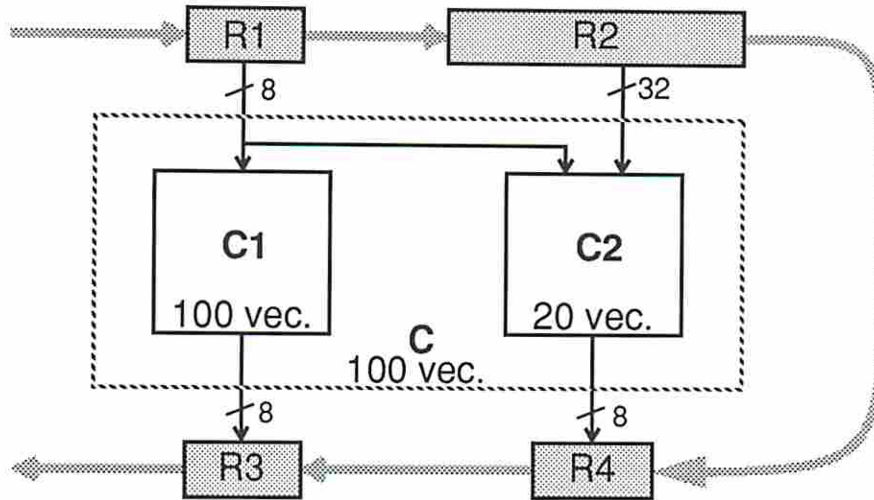


Figure 6.1: Subdividing a kernel by output-based partitioning.

Note that to test C1, test patterns need to be scanned only into R1 and test results need to be scanned only out of R3. Depending on the relative number of test patterns for C1 and C2 and the sizes of the various scan registers, this could imply a saving in test time in certain cases *provided* the ordering of registers in the scan path is appropriate for this purpose. The problem of ordering the scan registers will be dealt with in Chapter 8. However, for the current example, assume that the scan path contains the registers connected in the order shown in Figure 6.1. Let the test length of C be 100 (patterns) and let the test lengths of the subkernels C1 and C2 be 100 and 20, respectively. Assume also that C1 and C2 are combinational. The test time for C is $(41 \times 100) + 16 = 4116$ clock cycles. This test time can be reduced to 1744 clock cycles simply by testing C1 and C2 as separate kernels. During the test for C1, each test vector is shifted into R1 and each test result is shifted out of R3. Thus only a portion of the scan path needs to be accessed, and only 908 clock cycles are required to test C1. During the test for C2 the entire scan path needs to be accessed, requiring a total of 836 clock cycles. Thus the overall test time for C is reduced significantly in this example.

The process of output-based partitioning can be described in formal terms as follows. Let the GTG of the original circuit be $G \equiv (V, A, c, w)$, and let the kernel resulting from the partial scan analysis on G be $K \equiv (V_k, A_k, c, w)$. Let $KO \subseteq V_K$

(a) $CS \leftarrow \{a \in A_u \mid c(a) \neq 0\}$.

(b) $R \leftarrow CS$.

4. Sort the arcs in CS in order of decreasing weight.

5. $CS' \leftarrow \phi$, the set of arcs retained in the GTG initially.

6. For all arcs a in CS, in order of decreasing cost:
   Check whether the inclusion of a makes the merged graph unbalanced:
   If [**checkSW**(V_u, $(A_u - R) \cup CS' \cup \{a\}$, c_u, w_u) = SUCCESS]
   then $CS' \leftarrow CS' \cup \{a\}$.

7. $R \leftarrow R - CS'$; return R. □

Thus the algorithms of Chapter 3, which are applied to the TG circuit model containing clouds and registers, have been modified so that they can be applied to circuits containing switches and can take advantage of the expanded class of balanced structures. Since the basic outline of the algorithms is unchanged, the computation complexities as a function of the graph sizes are similar. For a given circuit, however, the TG is smaller than the GTG, since every node in the former represents a whole cloud of combinational logic; hence in general the amount of computation for the modified algorithms is higher. The actual complexity of the modified algorithms depends on how many switches are present, and on a related factor, how much logic is contained in each ACLU. In summary, the explicit consideration of switches may lead to a lower number of required scan registers, but the amount of computation to achieve this may be higher.

5.5 Partial Scan Testing Using I-Paths

In the preceding sections, the properties of switches have been used to expand the class of balanced structures based on embedded switches. We now consider another aspect of switches: the fact that they can often be used to transfer test data unchanged between scan registers and other parts of the circuit.

5.5.1 I-Paths

When switches are present in a circuit for functional purposes, they can also be used during test to transport input patterns and output results. This can lead to a higher degree of controllability and observability of inner parts of a circuit. Essentially switches help to set up paths along which data can be transmitted unchanged. Such paths are referred to as identity-paths or **I-paths**. The concept of I-paths has been defined and developed extensively in [4]. Here we derive a restricted definition of an I-path that is suited to its role within the partial scan design problem. All I-paths according to this definition satisfy the definition of I-paths in [4], but the converse does not hold.

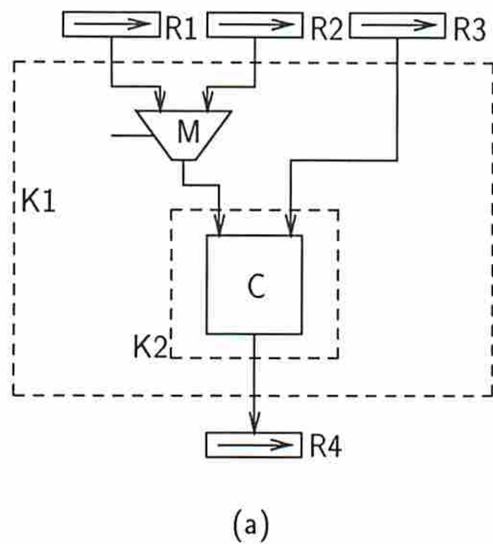
As before, the GTG $G \equiv (V, A, c, w)$ will represent a circuit under consideration; $SW \subseteq V$ represents the set of switches; and in addition, $FN \subseteq V$ represents the set of fanout nodes.

Definition An **I-path** is a loop-free path in the GTG in which every node is either a switch or a fanout node.

In the circuit of Figure 5.2, M1 is a switch and F1 is a fanout node. The individual nodes M1 and F1 constitute two separate I-paths of length 0 each; the path from M1 to F1 is another I-path of length 1. In order to store information about useful I-paths, we introduce two sets, IL and OL.

$IL \subseteq A$ is a list of **inlets**, or connections that are fully controllable. Thus $IL = \{a \in A \mid \exists$ an I-path starting at some PI node and ending at the source node of $a\}$. In the circuit of Figure 5.2, $IL = \{$(M, F), (F, B), (F, C)$\}$.

$OL \subseteq A$ is a set of **outlets**, or connections that are fully observable. Thus $OL = \{a \in A \mid \exists$ an I-path starting at the destination node of a and ending at some PO node$\}$. The circuit of Figure 5.2 has no outlets.



PLAN (K1, t):
(t, R4, REC)
(t-1, R1, DRI)
(t-1, R2, DRI)
(t-1, R3, DRI)

(b)

PLAN (K2, t):
(t, R4, REC)
(t-1, M, R1)
(t-1, R1, DRI)
(t-1, R3, DRI)

(c)

Figure 5.7: Illustration of kernels with I-paths. (a) Circuit with two kernels, (b) test plan for K1, (c) test plan for K2.

5.5.2 Kernels with I-Paths

In the absence of I-paths the kernel in a partial scan design is simply the circuit with scan registers removed and replaced with primary input/output pairs. In effect the kernel is that portion of the circuit to which arbitrary test patterns can be applied and on which ATPG can be carried out. However, if there are any I-paths adjacent to any scan register or circuit primary input/output, it may be possible to isolate a smaller portion of the circuit to which test patterns can be applied using I-paths. This is illustrated in Figure 5.7. Assuming all the registers R1–R4 are scan registers, the structure K1 could be considered as the kernel. However, the switch M provides an I-path from either register R1 or R2 to the inputs of C. This makes the output of M an inlet along with the output of R3; the input of R4 is of course an outlet. Consider the subcircuit K2, consisting of block C, which has inlets at its inputs and an outlet at its output. Clearly we can consider K2 as a kernel that can be tested as a separate unit. Given a test pattern set for K2, each pattern can be applied by scanning it into (say) R1 and R3, setting M to read data from R1 for one clock cycle, and scanning out the test result in R4.

Associated with every potential kernel K (such as K1 or K2) is a **test plan**, $PLAN(K, t)$, which describes what I-paths (if any) are to be set up and how test

patterns are to be applied. The test plans for K1 and K2 are shown in Figures 5.7(b) and 5.7(c), respectively. Each test plan consists of an unordered collection of 3-tuples of the form $(f(t), X, mode)$. $f(t)$ is some expression that is a function of time; X is a circuit object, a scan register or a switch, that actively participates in the test at time unit $f(t)$; and *mode* is the specific mode of operation of X at time $f(t)$ during the test. In our example it is assumed that test results are loaded into the scan path registers at clock cycle t, and all other time instants are expressed in relation to this parameter.

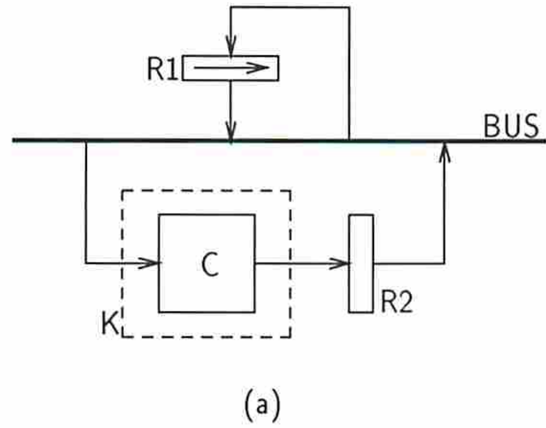
The description of the *mode of operation* depends on the type of circuit structure. For a switch, the mode of operation refers to the name of the structure whose output is read by the switch at the corresponding time. For a scan register, the mode of operation is either one of its functional modes (LOAD, HOLD, etc.) or the keyword DRI or the keyword REC. DRI indicates that the scan register must have the appropriate test input data present in it at the corresponding time, and it is implicit that it must have been held there since it actually arrived. REC indicates that the scan register loads in a test result, and it is implicit that this result must be held for as many clock cycles as required until it can be shifted out.

Returning to Figure 5.7, using K2 as the kernel rather than K1 has certain advantages.

- Since K2 has a lower depth (in terms of the amount of combinational logic from inputs to outputs) than K1, and in general is smaller than K1, the ATPG cost for K2 can be expected to be lower.
- Since the test for K2 uses only two input scan registers, R1 and R3, rather than three in the case of K1, a lower amount of time is required for shifting in test patterns, provided the scan path registers are chained appropriately.

Note that the switch M is exercised in only one of its two modes of operation in the test plan of Figure 5.7(c), and some simple functional testing is required for the other mode.

Another benefit of using I-paths is illustrated in the hypothetical example of Figure 5.8(a). In order to break all cycles and make the structure balanced, both



PLAN (K, t):
(t, R1, REC)
(t-1, BUS, R2)
(t-1, R2, LOAD)
(t-2, BUS, R1)
(t-2, R1, DRI)

(b)

Figure 5.8: Use of I-paths in reducing partial scan overheads. (a) Circuit, (b) test plan.

R1 and R2 need to be made scannable. However, consider the effect of making R1 alone scannable, represented in the figure by the arrow inside R1. Due to the I-paths associated with the bus, the input to C becomes an inlet and its output becomes an outlet. Any test pattern can be applied to the kernel formed by C by using R1 as both a driver and a receiver of test data. The corresponding test plan is shown in Figure 5.8(b). In this example the overhead due to partial scan has been reduced by making use of I-paths to isolate a balanced kernel.

Thus three potential benefits of using I-paths have been presented above: reduced scan overhead, reduced ATPG effort, and reduced test time. In the following analysis we assume that for the circuit under consideration, a set of scan registers has already been selected, using the algorithms described earlier in this chapter which ignore I-paths, such that the resulting kernel is balanced. Note that the algorithms for selecting scan registers could be modified so as to give preference to those registers that are close to switches; this heuristic would help to ensure that I-paths present in the circuit are best utilized for enhancing internal controllability and observability. Given the scan registers, we now study the problem of identifying a minimal kernel, in the presence of I-paths, that can be fully tested while achieving the various benefits described above.

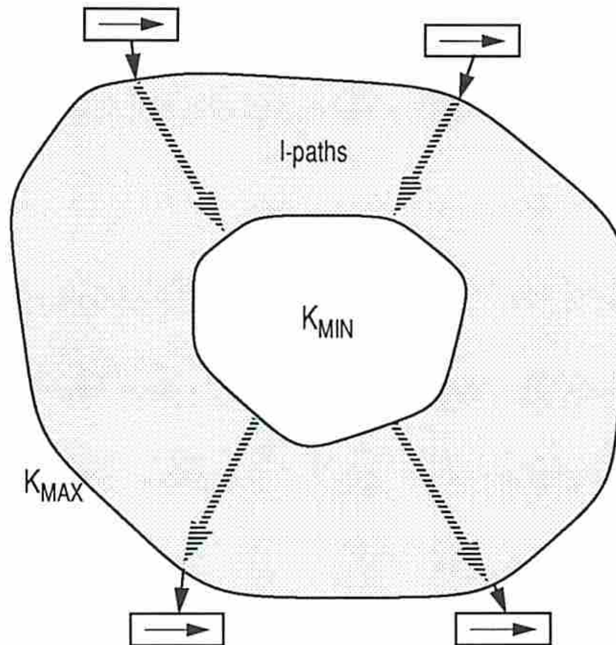


Figure 5.9: Relationship between maximal (K_{MAX}) and minimal (K_{MIN}) kernels.

5.5.3 Unsatisfiable Kernels

In the example of Figure 5.7 the kernel K2 was identified as *the smallest part of the circuit that included all the non-switch logic and had inlets/outlets surrounding it.* In any circuit, given the scan registers and the I-paths, such a kernel can be easily identified; we will refer to it as the **minimal kernel**. On the other hand, the kernel K1 in Figure 5.7 is called the **maximal kernel**, since it consists of *the entire circuit excluding all scan registers* (which are replaced by primary I/O). The relationship betwen the maximal and minimal kernels is illustrated schematically in Figure 5.9, where K_{MAX} and K_{MIN} represent the maximal and minimal kernels, respectively.

Clearly the nature of the minimal kernel guarantees that every input and output is individually accessible to some scan register(s) through I-paths. However, in order to test the minimal kernel in a given circuit it may be necessary to apply arbitrary distinct patterns to every input simultaneously, which may in fact not be possible using any conceivable test plan. Such a kernel is said to be **unsatisfiable**.

Note that the maximal kernel in any design must always be satisfiable since every input/output is connected to either a scan register or primary I/O. There are three types of conflict situations that can cause a kernel to be unsatisfiable. These three types are described below, and methods for resolving conflicts are discussed in Section 5.6.

No-match Conflict In the circuit of Figure 5.10(a), which has two driving scan registers R1 and R2, the minimal kernel consists of the structure K1, which has three inputs. Clearly, given an arbitrary single-pattern test to be applied to K1, it is not possible to scan the entire pattern into the scan path and apply it to K1, unless of course the circuit is further modified. This is evident from the fact that there is no one-to-one mapping of a subset of scan registers to the kernel inputs such that there is an I-path between each scan register and the corresponding kernel input. Clearly the kernel K2 must be satisfiable since it is the maximal kernel; its test plan is shown in Figure 5.10(b). Hence one solution to the no-match conflict for K1 is to simply use the maximal kernel K2, i.e., ignore I-paths in this structure. A more detailed analysis will be presented in Section 5.6.

Data Conflict Another case of unsatisfiability is shown in Figure 5.11. Here every input/output of the kernel K1 can be mapped to an apppropriate scan register. However, as the test plan of Figure 5.11(b) shows, the bus is required to read data from two different sources at the same time, which is an obvious data conflict. This conflict can be eliminated by including the fanout point at the output of the bus in the kernel, resulting in kernel K2. The kernel now has only one input and the bus does not have to carry out conflicting tasks.

Control Conflict In the preceding analyses we have assumed that the control lines feeding all switches are independent of each other. However, in practice there may be constraints on the control patterns that can be applied to the various switches and registers. For example, consider the circuit of Figure 5.8(a) and the test plan for kernel K shown in Figure 5.8(b). Suppose that R2 has a HOLD mode of operation, and that the control lines for the circuit are configured such that R2 cannot load

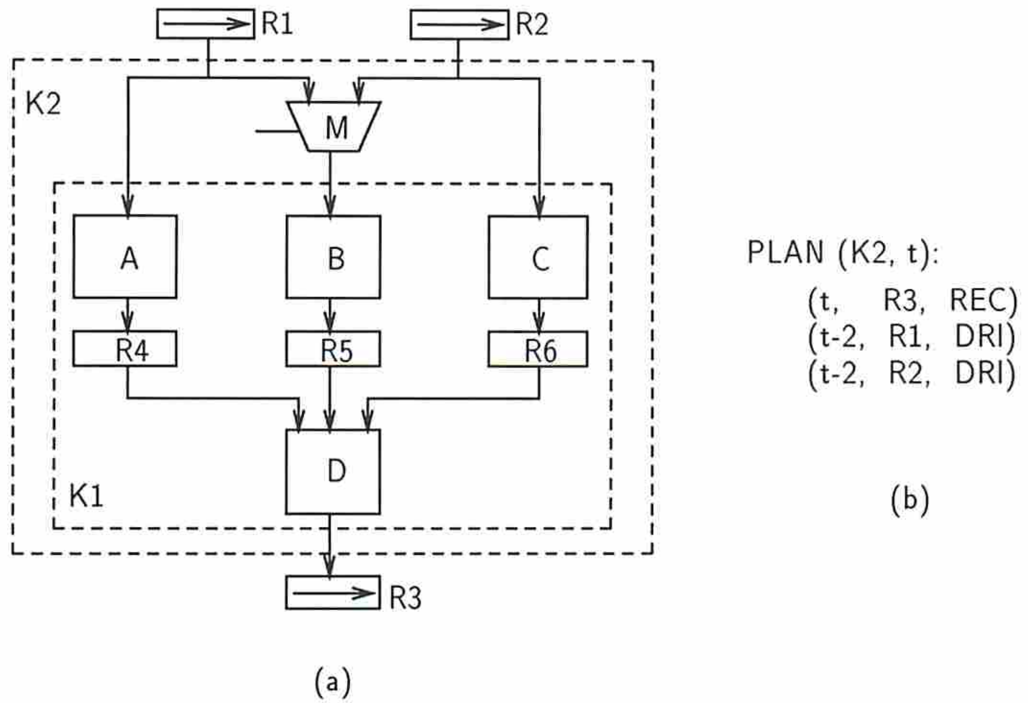


Figure 5.10: No-match situation. (a) Kernels K1, K2; (b) test plan for K2.

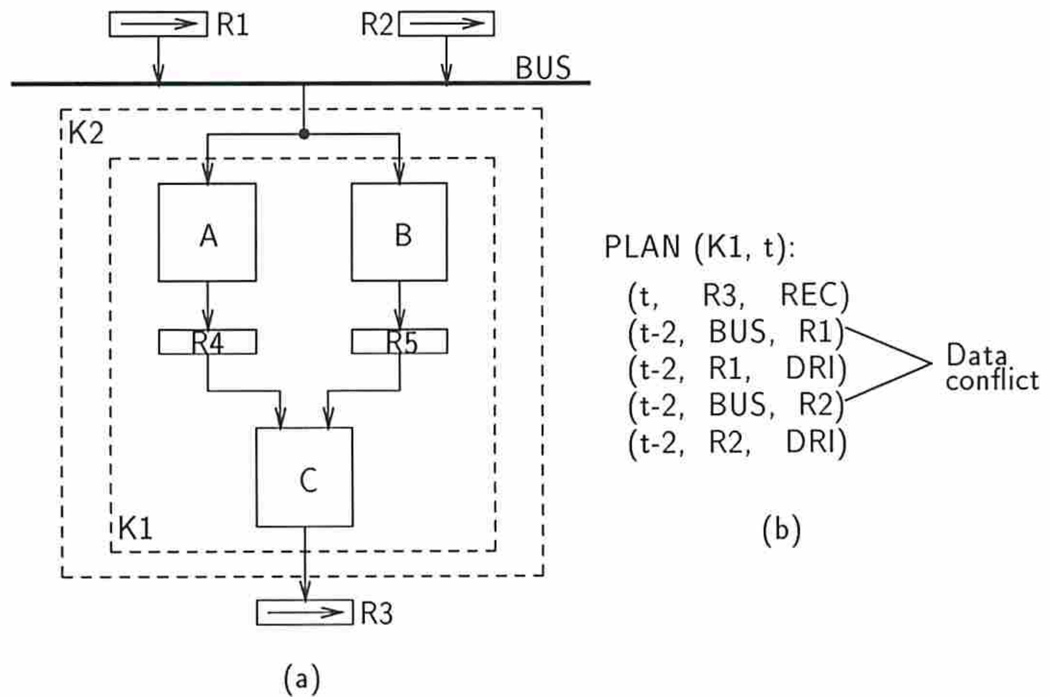


Figure 5.11: Unsatisfiable kernel due to data conflict. (a) Circuit, (b) test plan showing conflict.

data in the same clock cycle that the bus reads data from R2. In this case the operations required at time $t-1$ cannot be carried out, making K unsatisfiable.[1]

Thus given a kernel in a partial scan design, the following steps can be taken to check whether it is satisfiable. If possible, find a mapping of scan registers to kernel inputs (outputs) such that an I-path exists between every scan register and kernel input (output) pair. If both mappings exist, determine the test plan for the kernel based on these I-paths. If there are no data or control conflicts in the test plan, the kernel is satisfiable.

5.6 Finding a Satisfiable Kernel

Clearly the maximal kernel must be satisfiable, since it is the original kernel derived from the partial scan analysis and does not use I-paths. In the presence of I-paths, if the corresponding minimal kernel is found to be satisfiable, then it is sufficient to obtain a single-pattern test set for it and apply these tests according to its test plan. Note that if the maximal kernel is an SB-structure, the minimal kernel (which is a part of the maximal kernel) must also be an SB-structure, hence it should be possible to obtain a complete single-pattern test set using ordinary combinational ATPG on its combinational equivalent.

If the minimal kernel is found to be unsatisfiable, there must exist some satisfiable kernel that is a superset of the minimal kernel and a subset of the maximal kernel. One such kernel is of course the maximal kernel itself. Our problem is to find the smallest such kernel, so as to ensure the maximum benefit of using the I-paths available in the circuit. This is illustrated schematically in Figure 5.12, where K_{SAT} represents the minimal satisfiable kernel. The procedure **findKernel**, which will be listed shortly, carries out the task of determining the minimal satisfiable kernel.

[1]The circuit model used in this research requires the registers to be independently controllable through external control lines, which implies that control conflicts cannot occur. However, the control conflict is discussed here anyway to make the make the list of conflict types complete.

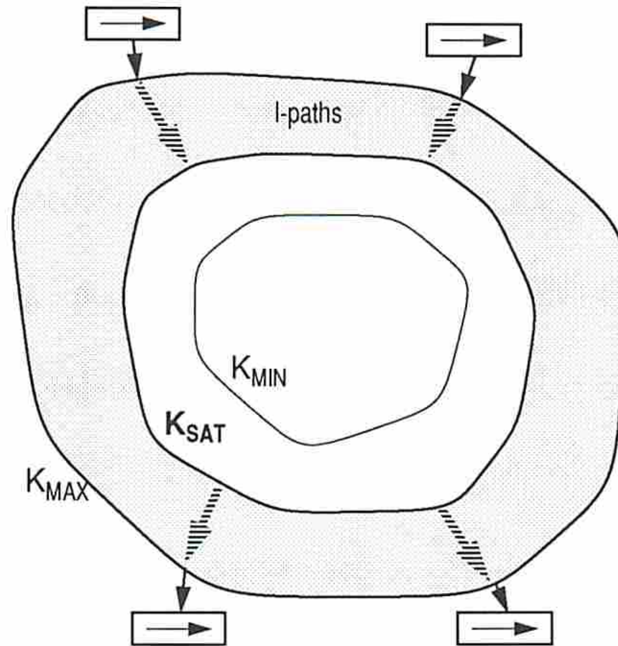


Figure 5.12: Relationship between maximal (K_{MAX}), minimal (K_{MIN}) and minimal satisfiable (K_{SAT}) kernels.

5.6.1 Expansion Procedure

The approach used in **findKernel** is to expand the unsatisfiable kernel in steps, eliminating conflicts one at a time. Each expansion step is based on a relatively simple procedure, described below, that adds one or more nodes to the kernel. Later the strategy for resolving conflicts as well as the complete **findKernel** procedure are presented.

The expansion process is carried out by the procedure **mergeNode**. It takes three inputs: the circuit under consideration, the current kernel within it, and some node outside the kernel (generally a switch or a fanout node) which is to be merged with the kernel.

function mergeNode ($G \equiv (V, A, c, w)$: GTG of circuit; $K \equiv (V_K, A_K, c, w)$: GTG of kernel; v: node that is to be merged into K): Returns K', GTG of resulting kernel.

1. $V'_K \leftarrow V_K \cup \{v\}$.

2. For all $u \in V$, if there is a path from v to any node in V_K passing through u, or there is a path from any node in V_K to v passing through u, then add u to V'_K.

3. $A'_K \leftarrow A \cap (V'_K \times V'_K)$.

4. Return $K' \equiv (V'_K, A'_K, c, w)$. □

The manner in which the merging procedure is actually invoked depends on what kind of conflict is currently being resolved.

5.6.2 Dealing with No-Match Conflicts

A no-match conflict could occur on either the input side or the output side of the kernel. The discussion below refers to a no-match conflict on the input side but applies to the output side also.

In the minimal kernel, as well as all kernels created by the expansion process, every kernel input must be accessible from some scan register through an I-path. Even if no *complete* one-to-one match of scan registers to kernel inputs exists, some *partial* one-to-one match must exist; a trivial example is a match between any one kernel input and one scan register that has an I-path feeding that input. For example, consider the no-match situation for kernel K1 in Figure 5.10(a). One possible partial match for the kernel inputs is R1 driving the input of A and R2 driving the input of B. Now for the input of C there is no possible matching scan register since R2 is already being used for driving another kernel input. In general, a no-match situation implies that for some input of the kernel under consideration, all possible driver scan registers have already been matched to some other kernel inputs.

In this situation, let KI_1 represent the kernel input for which no match can be found. Let R be any one of the scan registers that could possibly drive this input through an I-path, and let KI_2 be the kernel input to which R has already been matched. In our example, KI_1 is the input of C and KI_2 is the input of B. Let IP_1 and IP_2 be the I-paths connecting R to KI_1 and KI_2, respectively. Each I-path consists of registers, switches and fanout nodes; hence there must be a fanout node v at which the two I-paths IP_1, IP_2 fork out. In our example, v is the fanout

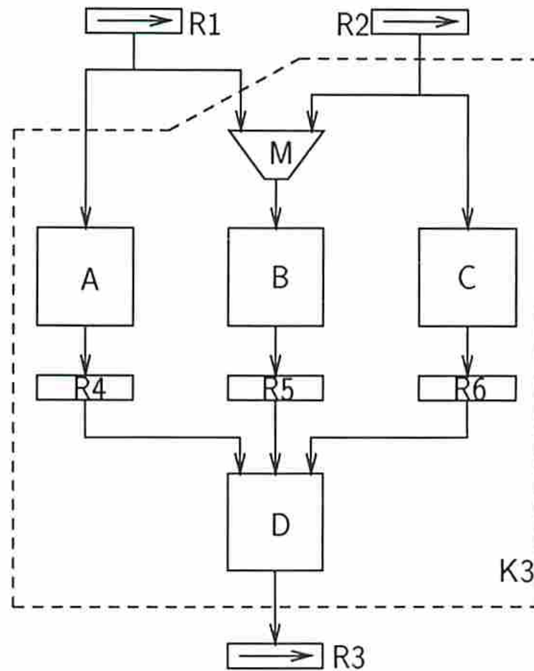


Figure 5.13: Resolving a no-match conflict for K1 in Figure 5.10.

point at the output of R2. We refer to v as the **conflict node** for the no-match conflict currently under consideration. The conflict can be eliminated by expanding the kernel K to include node v, using the procedure **mergeNode** presented above, resulting in a larger kernel K'. Thus in Figure 5.10(a), the switch M and the fanout point of R2 are added to the kernel, resulting in the kernel K3 shown in Figure 5.13.

Since the set of kernel I/O for the expanded kernel K' differs from that for the original kernel K, the search for a match must be carried out again on K'. The process of resolving any no-match conflict and obtaining a new match must be repeated, if necessary, until a complete match is obtained for all kernel I/O. In the worst case the resulting kernel could be the maximal kernel itself. In Figure 5.13, no complete match exists for the expanded kernel K3, and one more expansion step is required, resulting in kernel K2 of Figure 5.10(a) whose test plan is shown in Figure 5.10(b). The existence of a complete match does not guarantee that the associated test plan is conflict-free (unless the resulting kernel is the maximal kernel and uses no I-paths); we still need to check for data/control conflicts among the I-paths in order to ensure that the test plan is feasible.

5.6.3 Dealing with Data Conflicts

A data conflict is essentially a pair of steps in the test plan that cannot be executed simultaneously, making the test plan infeasible. Different conflicts may be related. For example, if a given I-path consisting of several switches is required to transmit two different streams of data simultaneously, the resulting test plan would have several conflicts, one at each switch. In such a case, the conflict that involves a node closest to the kernel is called the **primary conflict**, and the others are called its **secondary conflicts**. By resolving a primary conflict, the related secondary conflicts get simultaneously resolved.

Figure 5.14 illustrates two cases of related data conflicts occuring along an I-path segment. The I-path segment consists of a series of registers, switches, and/or fanout points. In Figure 5.14(a), the I-path segment along which the conflict occurs carries two streams of conflicting test input data diverging towards different inputs of the kernel. In this case there must be a fanout node $v \in FN$ at the point of divergence, and this is the site of the primary data conflict. Similarly in Figure 5.14(b), the two streams of conflicting test results must converge at a switch node $v \in SW$, which is the site of the primary data conflict. In both cases the node v is referred to as a **primary conflict node**.

In both cases (a) and (b) of Figure 5.14, the primary conflict can be eliminated by expanding the kernel K to include the node v, according to the procedure **mergeNode**, resulting in a larger kernel K'. In the merging process some additional switch and/or fanout nodes are included in K'. It should be noted that although the targeted primary and secondary conflicts get resolved, some new conflicts can be created by this procedure. For example, consider the switch v' in Figure 5.14(a) which gets merged into K'. The switch v' may have additional inputs which were previously ignored when K was the kernel, but now become inputs to the kernel K' itself. Thus it is quite possible that K' is unsatisfiable due to new conflicts related to the new input to be considered. However, note also that the process of resolving one data conflict increases the size of the kernel in a monotonic manner; repeated expansions must stop when the kernel becomes the maximal kernel, which is known to be satisfiable.

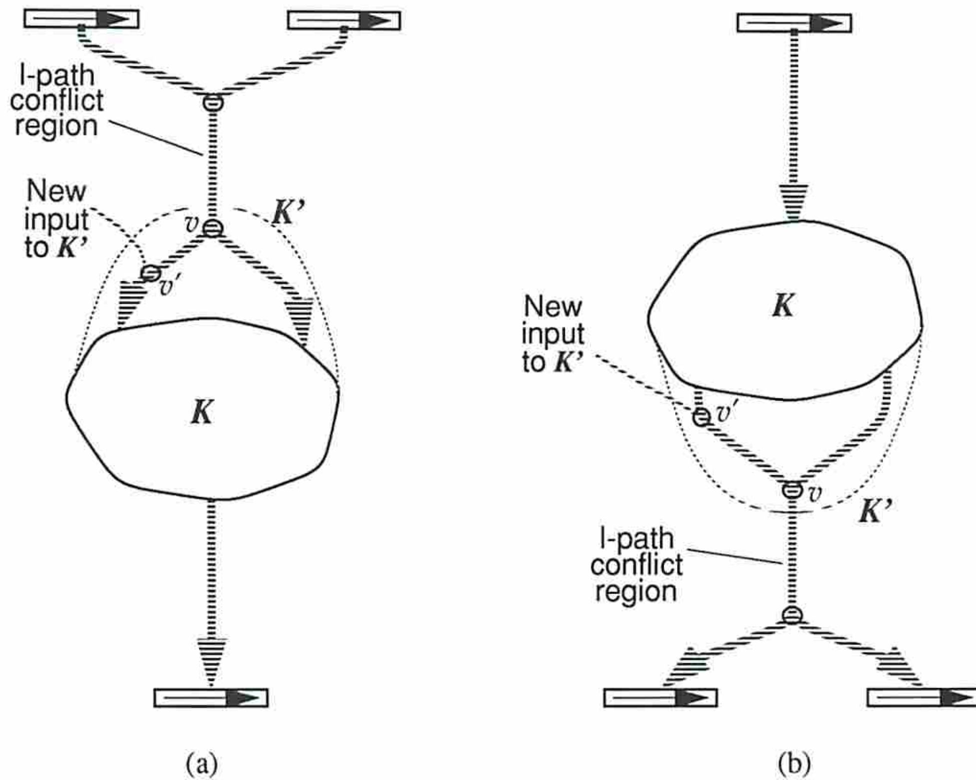


Figure 5.14: Schematic illustration of primary data conflicts. (a) Input side of kernel, (b) output side of kernel.

5.6.4 Dealing with Control Conflicts

A similar philosophy is used in resolving control conflicts. Let the conflicting test plan steps be $(t1, X1, M1)$ and $(t2, X2, M2)$. In general a control conflict may involve two test plan steps occurring at different times; i.e., $t1 \neq t2$. This could be caused by the nature of the state transitions in the controlling circuitry. In this analysis we will assume that information about control conflicts in the test plan is available to the software that eliminates conflicts; we will not deal either with how this information is determined or with its representation.

Whenever an I-path is used for transmitting test data, there is an implicit assumption that all elements (registers and switches) in the I-path, as well as across different I-paths, are independently controllable. If a control conflict is detected, it means that some constraint buried within the control regime has been violated. The

solution is to merge the conflicting I-path elements with the kernel and ensure that they are treated as ordinary functional logic during ATPG. The data propagation problem can then be tackled at a lower (say gate) level rather than the I-path level.

In terms of the graph model, the manipulations of the GTG to resolve a control conflict are identical to the situation where there are two separate data conflicts, one at each of X1 and X2. The two conflict nodes (each a switch or fanout node) are identified as before. The procedure **mergeNodes** must be invoked twice, once each with the two conflict nodes, to eliminate the control conflict.

5.6.5 Satisfiability Procedure

The **mergeNode** function is used by the function **findKernel**, listed below, to help in resolving data and control conflicts. **findKernel** takes as input the GTG of the circuit along with the set of scan registers, and returns the GTG of a minimal satisfiable kernel. It begins by determining the minimal kernel and checks whether it is satisfiable. (It is possible that in some cases the minimal kernel may consist of more than disjoint subcircuits. Although **findKernel** would treat them as a single kernel, Chapter 6 will show how a kernel may be broken into subkernels that can be tested separately.) If the minimal kernel is itself satisfiable, this kernel is returned. Otherwise, depending on the type of conflicts present, it expands the kernel in steps until the resulting kernel has a conflict-free test plan. Note that the result of this procedure is not unique since other solutions may also be possible.

function findKernel ($G \equiv (V, A, c, w)$: GTG of circuit; $R \subseteq A$: scan registers; $SW \subseteq V$: switches): Returns subgraph K of G such that K is a minimal satisfiable kernel, along with test plan $PLAN(K, t)$.

1. Determine K_{max} and K_{min}, the maximal and minimal kernels.
   $K \leftarrow K_{min}$.
2. For K:
   $KI \leftarrow$ ordered list of input arcs, $(ki_1, ki_2, \ldots, ki_n)$;
   $KO \leftarrow$ ordered list of output arcs, $(ko_1, ko_2, \ldots, ko_m)$.
3. For all $a \in KI$: $dri(a) \leftarrow \{r \in R \mid \exists$ I-path from r to $a\}$;
   For all $a \in KO$: $rec(a) \leftarrow \{r \in R \mid \exists$ I-path from a to $r\}$.

4. Find a match of scan registers to kernel I/O as follows.

   Construct two lists $RI \equiv (ri_1, \ldots, ri_n)$ and $RO \equiv (ro_1, \ldots, ro_m)$, if they exist, such that:

$$\begin{aligned} \forall x \in [1,n], \quad & ri_x && \in dri(ki_x); \\ \forall x \in [1,m], \quad & ro_x && \in rec(ko_x); \\ \text{all} \quad & ri_x && \text{distinct}; \\ \text{all} \quad & ro_x && \text{distinct}. \end{aligned}$$

   If no such lists exist [no-match conflict], do:

   (a) Determine a conflict node v associated with the current no-match conflict;

   (b) $K \leftarrow$ **mergeNode**(G, K, v);

   (c) Go to Step 2;

   Else proceed.

5. Determine test plan $PLAN(K, t)$. Check the plan for conflicts.
   If there are no data/control conflicts in $PLAN(K, t)$, or if $K = K_{max}$, return K along with $PLAN(K, t)$;
   Else proceed.

6. If there is a data conflict, identify a primary data conflict; let it be between the steps $(t1, X1, M1)$ and $(t1, X1, M2)$, with $M1 \neq M2$. Let v be the primary conflict node. Eliminate this primary conflict:

   $K \leftarrow$ **mergeNode**(G, K, v).

   Go to Step 2.

7. If there is a control conflict, let it be between the steps $(t1, X1, M2)$ and $(t2, X2, M2)$, with $X1 \neq X2$. Let $v1$ and $v2$ be the conflict nodes corresponding to $X1$ and $X2$, respectively. Eliminate both conflicts:

   $K \leftarrow$ **mergeNode**$(G,$ **mergeNode**$(G, K, v1), v2)$.

   Go to Step 2. □

Assuming that the original kernel K_{max} determined by the partial scan analysis is an SB-structure, the kernel K that returned by **findKernel** must also be an SB-structure since it is contained in K_{max}. The corresponding test plan guarantees

that any single-pattern test for K can be applied. Given the test properties of SB-structures, a complete single-pattern test set for K can be obtained using ordinary combinational ATPG on its combinational equivalent.

In some cases the test plan for K may show that not all scan registers in R are actually used in testing the kernel. This is because a given kernel I/O port may be accessible to more than one scan register, and some of the scan registers may not be used in the matching that is actually used in $PLAN(K,t)$. For example, in Figure 5.7, only R1, R3 and R4 are required for applying tests to K2, while R2 is not. The unused registers are still useful for carrying out some simple functional testing on the switching logic that is not exercised by the test plan. However, having identified these registers, one of two design options can be used.

The first option is to identify the scan registers that are not used for testing the kernel, and actually remove them from the set of scan registers as a postprocessing step. This may lead to some saving in area overhead, at the cost of making some of the switching logic harder to test. Thus, for example, in Figure 5.7, by making R2 a non-scan register, it is more difficult to test the switch M completely—especially if this structure is a subcircuit buried in a larger circuit. Either functional testing or sequential ATPG may be required to test it.

The second option is to retain such registers and use them for explicitly testing the switching logic in a separate test session. For the structure of Figure 5.7, the following approach could be used.

1. Based on the combinational equivalent of the minimal satisfiable kernel, which is K2, use combinational ATPG to obtain a test set for this kernel. Apply these tests using scan registers R1, R3 and R4. (This subset of registers can be placed at appropriate positions in the scan chain so as to minimize the time to apply each test; the chaining issue will be studied in Chapter 8.)

2. By fault simulation, determine the faults in the maximal kernel, which is K1, but outside the minimal kernel K2, that have not yet been detected. Based on the combinational equivalent of K1, use combinational ATPG to obtain a test set for these faults only. Apply these tests using all appropriate scan registers.

An interesting parallel can be drawn between the construction of kernels in [4] and that in this work. In [4], potential kernels are initially identified as arbitrary functional logic blocks at the register-transfer (RT) level, and various clusters of these blocks, without regard to the presence of I-paths. Each of these kernels is then analyzed to determine whether I-paths exist that can route test data to and from it. In the approach presented here, on the other hand, a kernel is initially identified as a cluster of circuit blocks that is guaranteed to have an inlet/outlet at every port. Conflicts among these I-paths are then identified and the information is used to reshape the kernel to ensure that tests can be applied effectively. It can be argued that the approach developed in this work is more efficient since no time is wasted in analyzing kernels that do not have inlets/outlets at all their ports.

5.6.6 Example

An example of kernel minimization using I-paths is shown in Figure 5.15. The structure shown is an SB-structure kernel in a partial scan design. It is actually a part of a larger data path circuit generated by the USC-ADAM synthesis system. For simplicity the scan registers are not shown; instead the inputs/outputs of the structure that are connected to scan registers are marked by small horizontal arrows. Since every I/O port of the structure is either a circuit primary I/O or connected to a scan register, the structure clearly represents the *maximal kernel* in this design. It consists of two functional units (*add-1* and *mul-2*), five MUXes, and one register. The depth of this SB-structure is 1. Interestingly, it is not a B-structure because of the two paths from *mul-2* to *Mux-10-1* having different delays.

The shaded portion of the figure indicates I-path logic that can be removed from the kernel to obtain the *minimal kernel*, namely the unshaded portion of the structure in the center. The test plan for this kernel is listed below.

PLAN (Kmin, t):

$(t,$	*reg-6,*	*REC)*
$(t-1,$	*Mux-10-1,*	*add-1)*
$(t-1,$	*Mux-10-1,*	*mul-2)*

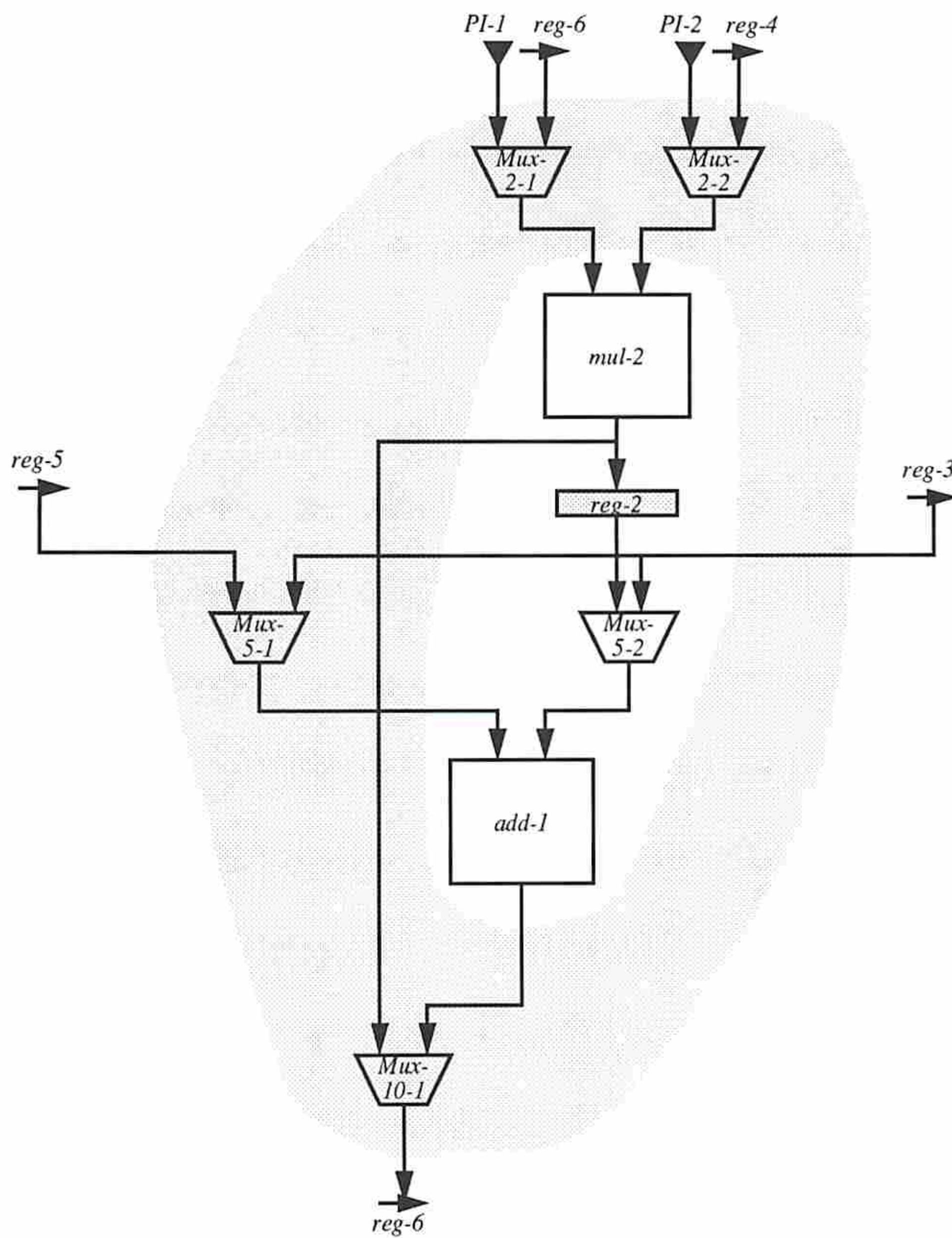


Figure 5.15: Kernel minimization: I-paths and minimal kernel.

($t-1$,	*Mux-5-1*,	*reg-5*)
($t-1$,	*reg-5*,	*DRI*)
($t-1$,	*reg-3*,	*DRI*)
($t-1$,	*Mux-2-1*,	*PI-1*)
($t-1$,	*Mux-2-2*,	*PI-2*)
($t-2$,	*Mux-2-1*,	*PI-1*)
($t-2$,	*Mux-2-2*,	*PI-2*)

There is a data conflict at time $(t-1)$ since *Mux-10-1* is required to execute two different actions. Hence the minimal kernel is not satisfiable. The conflict is eliminated by merging *Mux-10-1* with the minimal kernel, resulting in the kernel shown in Figure 5.16 (namely the unshaded region in the center). The test plan for this kernel is listed below.

PLAN (Ksat, t):

(t,	*reg-6*,	*REC*)
($t-1$,	*Mux-5-1*,	*reg-5*)
($t-1$,	*reg-5*,	*DRI*)
($t-1$,	*reg-3*,	*DRI*)
($t-1$,	*Mux-2-1*,	*PI-1*)
($t-1$,	*Mux-2-2*,	*PI-2*)
($t-2$,	*Mux-2-1*,	*PI-1*)
($t-2$,	*Mux-2-2*,	*PI-2*)

This test plan has no conflicts, hence the kernel is satisfiable. In order to obtain test patterns for this kernel, the register *reg-2* must be replaced by wires, and ordinary combinational ATPG can be carried out on the resulting structure. The process of obtaining the minimal satisfiable kernel guarantees that all irredundant faults in it can be detected by a test set obtained in this manner and applied using the test plan $PLAN(Ksat, t)$. Note that many of the faults in the I-path logic lying in the shaded region of Figure 5.16 are detected for free (and can be identified by fault simulation) during the test for $Ksat$. Any remaining undetected faults can be detected by carrying out ATPG separately on the maximal kernel.

kernel output. The second, *switch-based partitioning*, uses information about the switches present in the circuit to identify portions of the kernel that can be tested independent of others. The third, *size-based partitioning*, subdivides the kernel into smaller kernels by scanning additional registers. Using a suitable combination of these partitioning schemes, the single kernel resulting from the partial scan analysis can be subdivided into a set of smaller, more easily tested kernels.

The principles behind the three partitioning schemes are presented in the following sections, after which a global partitioning strategy is described in Section 6.5. To facilitate the discussion, we define a *cone* of logic as follows. $G \equiv \{V, A, c, w\}$ refers to the generalized topology graph (GTG) of the original circuit.

Definition Given an acyclic and/or balanced kernel $K \equiv \{V_K, A_K, c, w\}$ within G, and any node $v \in V_K$, the **cone of** v **in** K, denoted by **cone**(K, v) is defined as $K(v) \equiv \{V_K(v), A_K(v), c, w\}$ where $V_K(v) = \{u \in V_K \mid$ there is a path in K from u to $v\}$ and $A_K(v) = A \cap [V_K(v) \times V_K(v)]$.

6.2 Output-Based Partitioning

The first form of partitioning is illustrated in Figure 6.1. Assume that this structure is part of a larger circuit, and that all four registers R1–R4 have been selected to be in the scan path. The original unpartitioned kernel, C, consists of two substructures C1 and C2. C1 and C2 could be balanced or unbalanced structures, depending on the type of partial scan used. The outputs of C1 and C2 feed the scan registers R1 and R2, respectively. Starting at each of these outputs, and moving towards the inputs of C, we can identify the cones of logic feeding these two outputs. Each of these cones can be treated as a separate kernel. Thus the kernel associated with the output at R3 consists of C1 fed by the input at R1; the kernel for R4 consists of C2 fed by the inputs at both R1 and R2. ATPG can be carried out separately for C1 and C2. During test, the kernels C1 and C2 need to be tested in two separate sessions, because the same driver register R1 is required to supply different test data to C1 and C2.

denote the set of nodes in K at which there is an output to a scan register or to a primary output port of the circuit. Each such kernel output gives rise to a distinct subkernel. The set of all such output-based kernels, denoted by KS^{OP}, is given by the following expression:

$$KS^{OP} = \{\mathbf{cone}(K, v_o) \mid v_o \in KO\}.$$

Note that each kernel (cone) in KS^{OP} is in general an acyclic sequential structure (balanced or unbalanced, depending on the nature of the original kernel K).

Because the subkernels generated in this way may overlap, they do not strictly form a *partition* of the original kernel K. However, based on these kernels it is possible to form a strict partition of the *fault set* of K. For a given kernel $k_i \in KS^{OP}$, let $F(k_i)$ be the set of faults to be tested in it. When generating tests for the various kernels in KS^{OP}, faults that lie in the region of overlap among kernels $k_i, k_j \in KS^{OP}$ need to be tested only once. Thus after test generation is carried out on any kernel k_i, the faults already detected can be removed from the fault lists of all other kernels $k_j, j \neq i$ in which they occur. By doing so we can ensure that the fault sets $F(k_i)$ form a strict partition of the original set of faults $F(K)$. Note that any detectable fault in $F(K)$ must be detectable in at least one of the kernels $F(k_i)$.

Depending on the nature of the circuit under consideration, there may or may not be a high degree of overlap among the various cones generated in this way. Below we discuss the implications of various degrees of overlap among kernels.

Low Overlap Clearly, if two kernels k_i and k_j have very little logic in common and share few inputs, as illustrated in Figure 6.2(a), it is advantageous to treat them as separate kernels. This reduces the size of each circuit on which ATPG is invoked, and also has the potential for reduced test time as demonstrated earlier.

High Overlap Figure 6.2(b) shows a case where k_i and k_j have a large proportion of their circuit logic, as well as kernel inputs, in common with each other. In this case the reduction in the circuit size for ATPG is less significant, and there is low potential for reduced test time. In fact, there is actually a possibility of increased

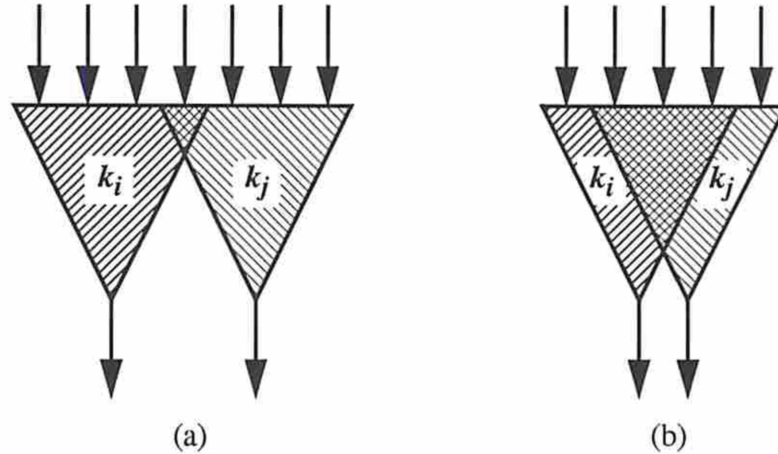


Figure 6.2: Overlapping kernels. (a) Low overlap, (b) high overlap.

test generation effort. Assume that ATPG is carried out first for all faults in k_i and then for the undetected faults in k_j. Suppose there is a fault f in the region of k_i common to k_j that can never be propagated to the output of k_i, but can only be propagated to the output of k_j. During ATPG on k_i, some backtracking effort may be wasted in trying to detect f. If however ATPG had been carried out on k_i and k_j combined, f might perhaps have been detected with less overall effort. The actual effort in each of the two cases depends on the characteristics of the ATPG algorithm and its implementation.

From the above observations it is evident that kernels with little or no overlap should be treated independently and kernels that overlap heavily should be merged. This guideline is used in the procedure presented below, **processKernels** to merge kernels that have a large amount of circuitry in common.

Merging Overlapping Kernels

The procedure **processKernels** compares all pairs of kernels k_i and k_j, which may have been derived using output-based partitioning or any other means. If the intersection of the two kernels (i.e., the overlapping region) has a size (number of combinational gates) that is more than a certain fraction σ (say, one third) of that

of their union (i.e., the combined size), it combines them into a single kernel. The procedure **mergeKernel** invoked by it, which is not listed here, simply returns a kernel formed by merging the two kernels passed as parameters. The kernels are considered in order of increasing size so that smaller kernels tend to get merged earlier before being compared with other kernels.

procedure processKernels (KS: set of kernels): Modifies KS by merging some kernels with others depending on overlaps, and returns modified set.

{
    $\forall k_i \in KS$, in order of increasing size $|k_i|$, do:
        $\forall k_j \in KS - \{k_i\}$, in order of increasing size $|k_j|$, do:
            if ($|k_i \cap k_j| \geq \sigma \cdot |k_i \cup k_j|$) /* σ is a parameter, $0 \leq \sigma \leq 1$ */
                {
                    $k_i \leftarrow$ **mergeKernel** (k_i, k_j);
                    Remove k_j from KS.
                }
    Return KS.
}

6.3 Switch-Based Partitioning

A form of partitioning using switches present in a circuit is shown in Figure 6.3. The circuit contains a kernel C of a circuit surrounded at the inputs and outputs by scan registers. The kernel consists of two sub-structures C1 and C2 connected to a multiplexer M. The original kernel can be subdivided into the smaller kernels C1 and C2, since any test for C1 or C2 can be applied by setting the multiplexer M to read data from C1 or C2, respectively. Hence it is sufficient to run ATPG on C1 and C2 separately. The tests for the various kernels need to be applied in separate sessions. However, as is the case for output-based partitioning, not all registers may need to be scanned in every session. Hence it is possible for the overall test time to be reduced due to partitioning in some cases, depending on the ordering of registers in the scan path.

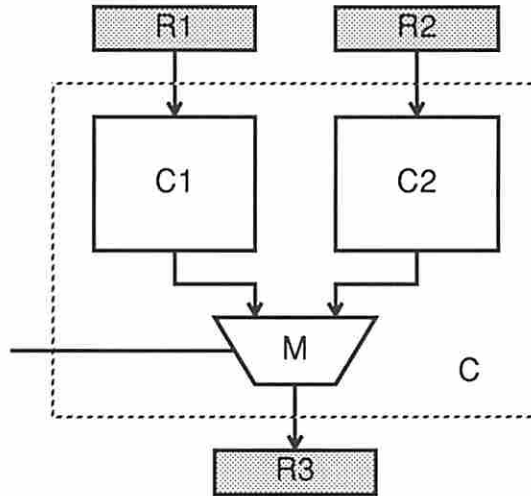


Figure 6.3: Subdividing a kernel by switch-based partitioning.

The example above illustrates an important fact, namely that switches provide a natural way to partition a circuit for ATPG. Given the set of switches contained in a kernel (if any), every combination of control settings for the various switches gives rise to one or more candidate kernels. For example, consider the circuit in Figure 6.4(a), which has two switches M1 and M2. Assume that during a given test session, the control input of switch M2 is set such that M2 reads data from A. During this entire session, M2 ignores the data at its input fed by B. Hence only faults in I1, I2 and A (i.e., in the cone of logic feeding the left input of M) along with M2 and C can be detected in this session. In effect, by setting M2 to read from A, we have isolated the kernel corresponding to the leftmost structure in Figure 6.4(c).

If M2 is configured instead to read data from B rather than A, then the resulting kernel contains I2, I3, M1, and B (i.e., the cone of logic feeding the right input of M2) along with M2 and C. This kernel can be further subdivided, by configuring M1 in its two possible modes, into two kernels to be tested in separate sessions. These two kernels correspond to the middle and rightmost structures in Figure 6.4(c). Thus we have shown how the original kernel of Figure 6.4(a) can be subdivided into three switch-based partitions by configuring the two switches in it in all possible ways. Note that the nodes C and M2, which feed the primary output

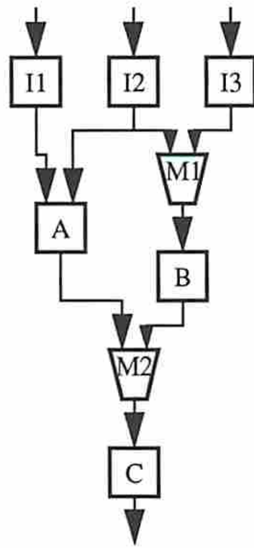
directly without passing through any switches, are always present in any subkernel irrespective of the switch configurations.

The procedure **switchPart** presented below can be used to construct the set of all switch-based partitions. The procedure generates a tree of states, each of which represents a partially constructed kernel for which some subset of the switches have already been configured to read data from one specific input port. Each leaf state corresponds to a complete kernel in which all applicable switches have been configured. The result of the search is a set of kernels KS^{SW}, each derived from a leaf state in the search tree. For the circuit of Figure 6.4(a), the states generated by the search process are shown in Figure 6.4(b).

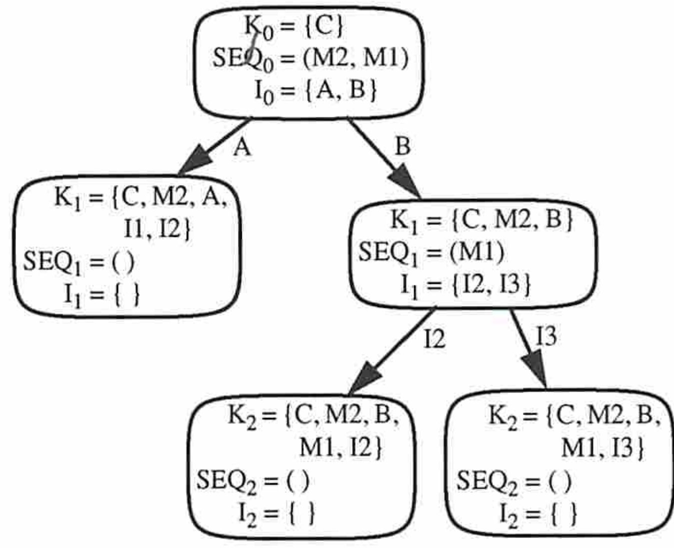
As before, $G \equiv (V, A, c, w)$ is the original circuit, $K \equiv (V_k, A_k, c, w)$ is the unpartitioned kernel, and $KO \subseteq V_k$ is the set of kernel output nodes. During the search the procedure maintains a list of states along the path from the start (root) state S_0 to the current state S_i. Each state S_i is represented by a 3-tuple (K_i, SEQ_i, I_i) where K_i is the current configuration of the kernel being generated; SEQ_i is a sequence of switches whose control settings have not yet been configured; and I_i is a subset of the input nodes of the first switch in SEQ_i.

The first operation in the procedure **switchPart** is to generate the start state $S_0 = (K_0, SEQ_0, I_0)$, as shown in Figure 6.4(b) for the current example. K_0 represents a part of the original kernel K that is present in all the kernels derived from switch-based partitioning. It is obtained by removing all switches from K and then merging the cones of all the output nodes KO in the resulting structure. SEQ_0 is a list of all switches in SW, sorted such that if $s_i, s_j \in SW$ and there is a path from s_i to s_j in K, then s_i appears *after* s_j in SEQ_0. Note that this ordering is not unique, and that such an ordering must exist since K is acyclic. Finally, I_0 is a set of nodes at the inputs of the first switch in SEQ_0. As the search proceeds, the elements in I_0 will actually be removed one by one until I_0 becomes empty. Note that the value of I_0 shown in Figure 6.4(b) is the initial value when the start state is created.

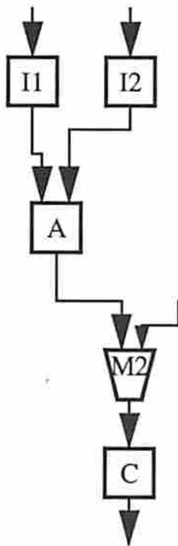
In general, from a non-leaf state S_i the procedure attempts to generate a new state S_{i+1} by deciding on a tentative control setting for the first switch in SEQ_i,

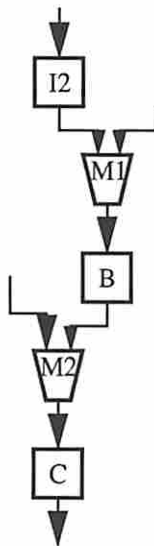


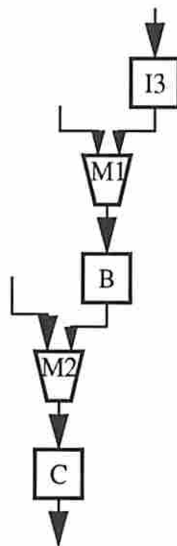
(a)



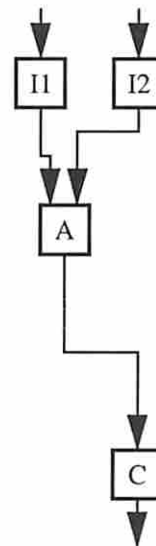
(b)

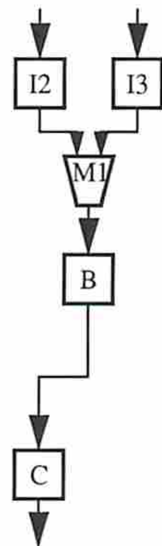






(c)





(d)

Figure 6.4: Illustration of switch-based partitioning procedure. (a) Original kernel K, (b) state space, (c) kernels generated, (d) combinational equivalents.

say M. In effect, the switch M gets added to the kernel along with the cone of logic feeding one of the inputs of M. If no further states can be generated, the current state is identified as a leaf state and the current kernel is added to KS^{SW}. The procedure then backs up from the current state to an earlier one and tries to generate alternative states from the new current state. The search ends when all states have been exhausted. The detailed procedure is listed below.

procedure switchPartition (circuit $G \equiv (V, A, c, w)$; original kernel $K \equiv (V_k, A_k, c, w)$; switches $SW \subseteq V_k$; kernel outputs $KO \subseteq V_k$): Returns KS^{SW}, set of kernels.

1. $K' \leftarrow K$ with all switches in SW removed.
2. $K'_O \leftarrow \bigcup_{v_O \in KO} \textbf{cone}(K', v_O)$.
3. Construct start state $S_0 \equiv (K_0, SEQ_0, I_0)$ as follows.
   (a) $K_0 \leftarrow K'_O$.
   (b) $SW' \leftarrow SW - \{\text{switches with no incoming arcs}\}$.
   (c) $SEQ_0 \leftarrow SW'$ sorted such that switch s_i in SEQ_0 does not have a path in K to any switch s_j in SEQ_0, $j > i$.
   (d) $I_0 \leftarrow \{\text{nodes in } K \text{ directly feeding the first element of } SEQ_0\}$.
4. Let current state be S_i.
   If SEQ_i is empty, this is a leaf state; do the following.
   (a) Add a copy of K_i to KS^{SW}.
   (b) Delete state S_i; find the most recent state S_j (i.e., the maximum j, $0 \leq j < i$) such that $I_j \neq \phi$.
   If there is no such state, the search is completed; **return** KS^{SW}.
   Else, make S_j the current state and denote it by S_i.
5. From current state S_i generate the next state $S_{i+1} \equiv (K_{i+1}, SEQ_{i+1}, I_{i+1})$ as follows. (Note that SEQ_i must be nonempty since S_i is not a leaf state).
   (a) $M \leftarrow$ first element of SEQ_i.
   (b) If I_i is empty do the following:
      i. If $i = 0$ **return** KS^{SW}.
      ii. $i \leftarrow i - 1$ (S_i is new current state).

iii. Go to 4.

Else do the following:

i. Let α be some arbitrary element of I_i; remove α from I_i.

ii. $K_{i+1} \leftarrow K_i \cup \{M\} \cup \textbf{cone}(K', \alpha)$.

(c) $SEQ_{i+1} \leftarrow SEQ_i - (M)$.

(d) While the first element of SEQ_{i+1} neither feeds any node in K_{i+1} nor feeds any primary output of K'_O:
remove the first element of SEQ_{i+1}.

(e) $I_{i+1} \leftarrow$ {nodes in K directly feeding the first element of SEQ_{i+1}}.

(f) $i \leftarrow i+1$ (S_i is new current state).

6. Go to 4. □

The subdivided kernels from switch-based partitioning of the kernel of Figure 6.4(a) are shown in Figure 6.4(c). As in the case of output-based partitioning, the set of subdivided kernels KS^{SW} may have overlaps among them; the procedure invocation **processKernels** (KS^{SW}) can be used to merge kernels that have a high degree of overlap, for example, the second and third kernels shown in Figure 6.4(c). The resulting combinational equivalents to be used for ATPG are shown in Figure 6.4(d). When a certain switch in a kernel has only one input present, this switch may be replaced by a simple wire in the combinational equivalent, as in the case of the first kernel. During test application this switch must be configured so that it selects the appropriate input. When two or more kernels are merged during postprocessing, the resulting kernel may have switches with more than one input present. In this case, for the purpose of ATPG, the switch should be modeled by a multiplexer having exactly as many inputs as are present in the kernel; the control inputs generated by ATPG should be transformed into the appropriate control patterns during test application.

6.4 Size-Based Partitioning

Both the output-based and the switch-based forms of partitioning can help to reduce the ATPG cost as well as potentially the overall test time. However, if any of the

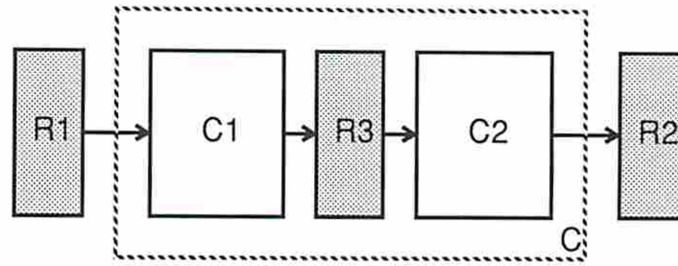


Figure 6.5: Subdividing a kernel by size-based partitioning.

resulting kernels is still too large, a third form of partitioning can be used to break it down at the cost of additional logic. This scheme, called size-based partitioning, involves introducing extra scan registers such that a kernel gets physically divided into two or more smaller kernels. An example is shown in Figure 6.5. The original kernel C is broken down into the subdivided kernels C1 and C2, each of which may be an acyclic/balanced structure, by converting R3 into a scan register.

Size-based partitioning differs from the output-based and switch-based forms of partitioning in two major characteristics.

- Size-based partitioning introduces extra scan registers, leading to additional overhead and possibly performance penalties.
- The resulting kernels have they advantage that they can be tested concurrently. This issue will be elaborated on in Section 7.2 under the discussion of test scheduling.

The "size" of a kernel refers to some characteristic that influences the ATPG cost. Typical size characteristics are the number of combinational gates, the number of inputs/outputs, or the largest number of gates in any input–output path. The exact notion of size depends on the actual design and ATPG environment, e.g., on what kind of computing power is available and the efficiency of the ATPG algorithm used. In the discussions here we assume that there is a function **size**(k) which, for any kernel k, returns either **0** if the kernel is acceptable or **1** if the kernel needs to be partitioned further.

Here we present an example of a simple greedy algorithm, **limitSize**, for size-based partitioning. It essentially determines a set of additional scan registers required to limit the *depth*, i.e., the largest number of non-scan registers in any path through the kernel. This procedure is an extension of that presented in [16], which has the limitation that it simply uses the number of registers in a path as an estimate of the test generation effort; other criteria are ignored. The **limitSize** procedure presented here allows the more general **size** function to be used as a partitioning criterion. The procedure builds up each partition by starting with a seed node and growing a cluster of circuit nodes until it reaches the size limit.

Whenever a node is added to a cluster, all nodes lying in the same *cloud* (as defined in Chapter 3) must be added to the cluster. In other words, a maximal set of nodes that are connected to each other via wires must be simultaneously added to the cluster. This ensures that at any stage in the formation of the cluster, all connections at the boundary between the cluster and the rest of the circuit are through registers. Hence the cluster can be made to form a valid partition, isolated from other partitions, by scanning these registers. In the procedure below, $cloud(v)$ denotes the set of nodes in the same cloud as the node v.

procedure limitSize (Circuit $G \equiv (V, A, c, w)$; original kernel $K \equiv (V_k, A_k, c, w)$): Returns R, set of additional scan registers.

1. $K' \leftarrow K$; $KS^{SZ} \leftarrow \phi$.
2. While K' is not empty do:
   (a) Pick a node k on the periphery of K'; i.e., $k \in K'$ such that k is a PI/PO node of K' *or* there is a node $v \in V$ adjacent to k but not in K'.
   (b) Construct kernel KP containing all nodes in $cloud(k)$.
   $K' \leftarrow K' - KP$.
   $TRIED \leftarrow \phi$.
   (c) If **size**$(KP) = 1$ then ($cloud(k)$ alone violates the size bound) go to step 2d; else expand subkernel KP in a greedy fashion as follows.
   Do forever:
      i. Pick a node $v \in K' - TRIED$ adjacent to a node in KP such that $|cloud(v)|$ is highest; if there is no such node, go to step 2d.
      ii. $KP_{temp} \leftarrow KP \cup cloud(v)$.

iii. If **size**(KP_{temp}) $= 1$ then add v to $TRIED$;
else add the nodes in $cloud(v)$ to KP and remove them from K'.

(d) KP is a maximal kernel satisfying the size limit; add it to KS^{SZ}.

3. $R \leftarrow$ all registers in A_k that connect two nodes in two different partitions in KS^{SZ}.

4. **Return** R. □

The procedure listed above results in a set of additional scan register required to partition the original kernel into subkernels. A side-effect of the greedy approach is that while most subkernels may be just under the size bound, a few residual kernels may be much smaller; this could be avoided if a more elaborate scheme were used.

6.5 Global Partitioning Strategy

Let K be the kernel resulting from the partial scan analysis of Chapter 5. The objective of the global partition strategy is to construct a set of kernels KS by using a combination of output-based, switch-based and size-based partitioning. A procedure **globalPartition** that carries out this task is listed below. First, it uses output-based and switch-based partitioning to break down the kernel as far as possible. If any of the resulting kernels violates the size criterion, size-based partitioning is used to introduce additional scan registers appropriately. In this case the entire procedure is repeated for the resulting kernel.

procedure globalPartition (Circuit $G \equiv (V, A, c, w)$; original kernel $K \equiv (V_k, A_k, c, w)$): Returns set of additional scan registers R and set of kernels KS.

1. $R \leftarrow \phi$.

2. *(Output-based partitioning)*
$KS^{OP} = \{\textbf{cone}(K, v_o) \mid v_o \in KO\}$.
$KTEMP \leftarrow \textbf{processKernels}(KS^{OP})$.

3. *(Switch-based partitioning)*
   $KS^{SW} \leftarrow \bigcup_{KP \in KTEMP}$ **switchPart**(G, KP).
   $KTEMP \leftarrow$ **processKernels**(KS^{SW}).

4. $KBIG \leftarrow \{KP \in KTEMP \mid \text{size}(KP) = 1\}$.
   If $KBIG$ is empty then **return** R and $KS \leftarrow KTEMP$.

5. *(Size-based partitioning is required)*
   For all $KP \in KBIG$ do:

   (a) $R^{SZ} \leftarrow$ **limitSize**(G, KP).

   (b) $R \leftarrow R \cup R^{SZ}$.

6. In original kernel K, remove all registers that are also present in R.
   Go to step 2. □

Each subkernel in KS obtained from the partitioning process can be treated as an independent kernel for the purpose of test generation and test application. Every subkernel has the same testability characteristics as the original maximal kernel; i.e., if the maximal kernel is an SB-structure, then every subkernel is an SB-structure. For each subkernel, the I-path analysis of Chapter 5 can be carried out to identify a *minimal satisfiable kernel*, along with the associated test plan.

6.6 Summary

This chapter has explored the problem of testing a circuit in a partitioned manner. The maximal kernel resulting from the partial scan analysis of the preceding chapters can be subdivided into smaller kernels by using a combination of three different forms of partitioning: output-based, switch-based, and size-based. Although the resulting kernels may have regions of shared circuit logic, the sets of faults targeted for detection in the various kernels form a proper partition of the set of target faults in the original kernel. By forming subdivided kernels, the overall test generation cost can be reduced, and in some cases the total test application time can also be reduced. The reader is asked to bear in mind that enhancing the automatic partitioning process, to maximize its benefits, would require more knowledge about

the relationships among partition features, ATPG cost, and test time; this is a subject for future research. The more important contribution of the partitioning study presented here is that it enables us to formulate the *test scheduling problem*, which is studied in Chapter 7.

Chapter 7

Test Scheduling

"The sum of the parts is lesser than the whole."
—Source unknown

7.1 Introduction

The partitioning procedure of Chapter 6 results in a set KS of independently testable subkernels. For each of these subkernels a combinational equivalent circuit can be derived in the same way as for the minimal satisfiable kernel of Chapter 5. ATPG can then be carried out separately on each combinational equivalent to obtain a test set for the corresponding kernel. We now have the problem of applying these tests in an efficient manner. The kernels could simply be tested one at a time in a sequence. However, as this chapter will show, the overall test time can be significantly reduced by exploiting the potential for testing different parts of the circuit in parallel.

Depending on the characteristics of the kernels in KS, some subsets of kernels may have mutual conflicts that require them to be tested in different test sessions; it may be possible to test other subsets concurrently with no conflicts. In this chapter the different types of kernel relationships that affect the scheduling of tests are studied, and subsequently a method for scheduling tests for the various partitions to achieve minimum overall test time is presented.

7.2 The Test Scheduling Problem

So far it has been assumed that all scan registers are to be connected together in a single scan chain. If instead they are distributed among multiple scan chains, each having its own scan in/scan out pins and if necessary its own test/normal mode control pin, the distribution has an influence on the scheduling of tests. The popular approach to multiple scan path chaining is to try to make all chains of equal length; however, it will be shown in Chapter 8 that the greatest reduction in test time can often be achieved by using unequal-length chains. In this section we assume that the scan chains have already been constructed; the problem of constructing the scan chains so as to minimize the overall test time will be studied in Chapter 8.

7.2.1 Test Application with Multiple Scan Chains

Consider the circuit example in Figure 7.1(a). A, B and C are kernels resulting from the partitioning process. Each may be a combinational logic block or an SB-structure; non-scan registers within the subkernels are not shown. Assume the test lengths of A, B and C are 50, 100 and 30 patterns, respectively. The scan registers, each having a width of 8 bits, are connected into two scan chains $chain1$ and $chain2$ as shown. For kernel K, let $T(K)$ denote the number of test patterns, and let $Ch(K)$ be the set of chains that are involved in testing K.

The number of clock cycles required to completely flush the scan chains in $Ch(K)$ is $max_{c \in Ch(K)}|c|$, where $|c|$ denotes the length of (i.e., the number of FFs in) scan chain c. However, depending on the ordering of the registers in the chains, and the positions of the registers that are actually used for testing K, it may not be necessary to flush the chains completely in order to apply tests to K. We define the **drive cycle** of K as the sufficient number of shifts required to scan a test pattern for K into the scan chain(s) in $Ch(K)$, and the **receive cycle** of K as the sufficient number of shifts required to scan a test result from K out of the scan chain(s) in $Ch(K)$. These terms are defined formally below.

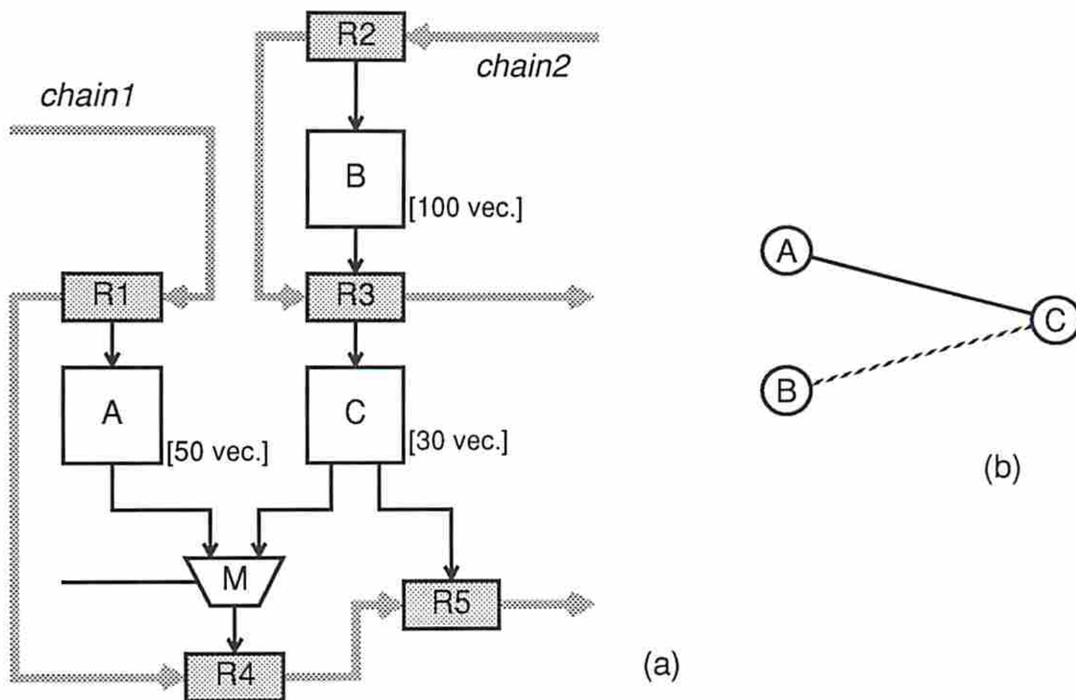


Figure 7.1: Example of partitioned testing. (a) Partitioned kernels, (b) test relationship graph.

Definition The **drive cycle** $DC(K)$ [**receive cycle** $RC(K)$] of K is the smallest integer n such that for all chains $c \in Ch(K)$, each FF in c that drives test patterns into [receives test results from] K is at most at the nth position away from the scan-in [scan-out] pin of c.

Thus $DC(A) = 8$, $DC(B) = 8$, and $DC(C) = 16$; $RC(A) = 16$, $RC(B) = 8$, and $RC(C) = 16$.

When applying tests to K, all chains in K must be operated with a cycle that is long enough to drive as well as receive test data for K.

Definition The **chain cycle** $CC(K)$ of K is $\max[DC(K), RC(K)]$.

For example since $DC(A) = 8$ and $RC(A) = 16$, we have $CC(A) = 16$. Note that to scan out the output of A present in $R4$ it is necessary to shift out the contents of both $R4$ and $R5$, thus requiring 16 clock cycles per test. When a series of tests are applied to A consecutively, it may be possible to get by with just 8 shifts per test by providing an individual HOLD control for $R5$ and using the following scheme. After applying a given test pattern, the test result must be shifted from $R4$ into $R5$ in 8 clock cycles, and then held there (to inhibit unwanted data being loaded into $R5$) while the next test pattern is applied. At this time a new test result is available in $R4$ and the old test result is available in $R5$, and the shifting operation is repeated over the next 8 clock cycles. The result of the previous test can now be observed. Although this "pipelined" scheme would reduce the effective chain cycle of A to 8, in general it requires some scan registers to have individual HOLD controls, which is a high overhead. Hence we will continue to assume that the scan chain(s) are completely flushed for each test vector application.

In cases where the assignment of registers to the various scan chains is known but the ordering of the registers in each chain is not known, the exact chain cycle cannot be computed. However, a worst-case assumption can be made by using $CC(K) = \max_{c \in Ch(K)} |c|$, i.e., the time required to flush all the chains.

For kernel B, $Ch(B) = \{chain2\}$ and $CC(B) = 8$. Thus every test requires 8 clock cycles for a test pattern to be scanned in while the previous test result is being scanned out, and 1 clock cycle for the test result to be loaded in to the scan

registers. Further, if B is a sequential structure with depth d, each test pattern needs to be applied to the kernel over a duration of d additional clock cycles. In the following discussions we assume that all three kernels have depth 0. Thus 9 clock cycles are required to apply a test pattern to B, and the total test time for 100 test patterns is 900 clock cycles, ignoring the time required to flush out the scan chains after the last test pattern is applied.

In the case of kernel A, $Ch(A) = \{chain1\}$, and $CC(A) = 16$. Since only 8 bits of actual test data are to be scanned into the driver register $R1$, the test patterns must be formatted and padded with dummy data so that $R1$ contains the appropriate pattern after 16 clock cycles of shifting. The total time to apply 50 test patterns to A is $17 \times 50 = 850$ clock cycles.

Kernel C is connected to more than one scan chain; $Ch(C) = \{chain1, chain2\}$ and $CC(C) = 16$. the total test time is $17 \times 30 = 510$ clock cycles.

Thus in general, for a kernel K, the time $TT(K)$ for testing it independent of all other kernels is given by the expression

$$TT(K) = T(K) \times [CC(K) + d + 1].$$

If all kernels are to be tested in separate test sessions, the overall test time TT for the circuit is given by

$$TT = \sum_{K \in KS} TT(K).$$

In the following section we investigate the conditions under which kernels can be tested concurrently, and how the various scan chains must be operated in order to achieve concurrent testing.

7.2.2 Kernel Relationships

Given a set KS of kernels to be tested and their *test plans* as described in Chapter 5, it is desirable to determine the relationships among them and accordingly schedule the tests so as to minimize the overall test time. There are two types of

relationships among kernels that can affect the scheduling of tests: *incompatibility* and *dependence*.

7.2.2.1 Incompatible Kernels

Incompatible kernels are those whose tests cannot be applied simultaneously. An incompatibility may either be a result of the partitioning process or be caused by conflicts among the I-paths in the test plans of the kernels. Let a kernel be represented by its GTG, and let $TP(K)$ denote the test plan for kernel K.

Definition Two kernels K_i, K_j are **incompatible** if either of the following holds:

1. there is a circuit node v present in both K_i and K_j;
2. there is a test step $(t1, X1, M1)$ in $TP(K_i)$ and a test step $(t1, X1, M2)$ in $TP(K_j)$ such that $M1 \neq M2$. □

In Figure 7.1(a), since the switch M transmits test results from both kernels A and C to the receiver register $R4$, it would be required to carry out conflicting actions if the kernels were to be tested concurrently. Thus A and C are incompatible due to the second condition in the definition above.

7.2.2.2 Dependent Kernels

In the example above, A and B are compatible and can be tested simultaneously. Further, $Ch(A) = \{chain1\}$ and $Ch(B) = \{chain2\}$, i.e., there is no chain that is used by both these kernels. This implies that A and B can be tested independently using *chain*1 and *chain*2, respectively, without the chains having to operate in sync.

For example, the 50 test patterns for A could be applied using the following sequence of operations: *Repeat* {*Scan in test pattern for A into chain*1 *(while scanning out previous test result) in 16 clock cycles; apply pattern to A; load test result into chain*1 *in 1 clock cycle*}. Simultaneously, the 100 test patterns for B could be applied using the following sequence of operations: *Repeat* {*Scan in test pattern*

for B into chain2 (while scanning out previous test result) in 8 clock cycles; apply pattern to B; load test result into chain2 in 1 clock cycle}. Of course this requires that the two chains be separately controllable; but the "asynchronous" operation helps to reduce the test time since *chain*2 does not have to waste time by operating at the longer chain cycle of *chain*1.

B and C are also compatible since they can be tested simultaneously. However, both use *chain*2 for applying tests, hence they are said to be **dependent.**

Definition Two *compatible* kernels K_i, K_j are **dependent** if both of the following hold:

1. $Ch(K_i) \cap Ch(K_j) \neq \phi$;
2. $CC(K_i) \neq CC(K_j)$.

The dependence between B and C implies that when testing C at the same time as B, the chains used for testing B (i.e., *chain*2) must be slowed down to be in sync with the other chains used for testing C (i.e., *chain*1). Thus *chain*2 would receive a new pattern for B every 16 clock cycles rather than every 8. Recall from the earlier analysis that the test time for C alone is $TT(C) = 510$ and that for B alone is $TT(B) = 900$. However, when testing them concurrently, the test time for B increases to more than 900 because of the slower operation of *chain*2 while C is also being tested. The reader will not be burdened at this point with the calculation of the new test time for B; this is deferred until the test scheduling model has matured. But it should be observed that in general a dependence between two kernels may tend to increase the test time, although the overall time for testing them in sync would still always be lower than the time for testing them separately as if they were incompatible.

7.2.3 Modeling Test Relationships

The incompatibility and dependence relationships among kernels are modeled using a **test relationship graph (TRG)**, which is defined as follows. A TRG is an

undirected graph (V, I, D) in which nodes in V represent the kernels resulting from partitioning; edges in $I \subseteq V \times V$ connect kernels that are incompatible; and edges in $D \subseteq (V \times V) - I$ connect *compatible* kernels that are dependent. Note that $I \cap D = \phi$. For the example of Figure 7.1(a), the TRG is shown in Figure 7.1(b). Solid lines represent incompatibility edges in I and broken lines represent dependence edges in D.

The TRG is an extension of the test incompatibility graph (TIG) defined by Craig et al. [28]. The TIG is used to model incompatibility relationships alone, and is similar to the TRG without any dependence edges. It is used for scheduling built-in self-test (BIST) circuits. In the TIG model, every kernel is assumed to have a fixed test time which does not depend on whether any other kernels are tested simultaneously. In contrast, in the case of scan design with multiple chains, the test time for a given kernel is not constant, since the chains may need to be slowed down depending on other kernels being tested at the same time. It is this feature of serial scan designs that cannot be modeled in the TIG, and is modeled in the TRG using the dependence relationship.

Definition A subset of kernels $V' \subseteq V$ is said to form a **compatibility group** if $I \cap (V' \times V') = \phi$, i.e., there are no incompatibility edges among them.

Thus in Figure 7.1(b), both $\{A, B\}$ and $\{B, C\}$ are compatibility groups. Each single kernel alone forms a compatibility group. Any set of kernels forming a compatibility group can be tested concurrently.

Definition A set of nodes V' forming a compatibility group is said to be a **dependence group** if the subgraph $(V',\ I' = \phi,\ D' = (V' \times V') \cap D)$ is connected.

Thus $\{B, C\}$ and $\{B\}$ are dependence groups but $\{A, B\}$ is not.

Definition Given a dependence group KD, the **chain group** of KD, $Ch(KD) = \cup_{K \in KD} Ch(K)$, i.e., the set of all chains used for applying tests to one or more kernels in KD.

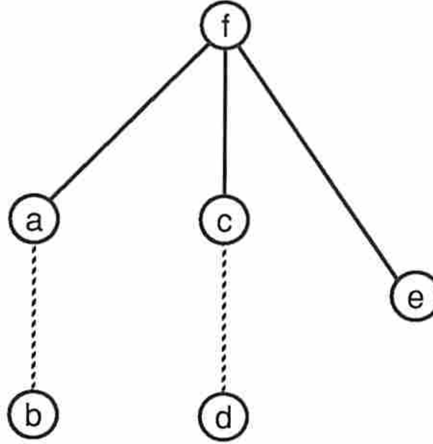


Figure 7.2: TRG example to illustrate dependence groups.

Analogous to the definition of the chain cycle of a kernel, we can define the chain cycle of a dependence group.

Definition The **chain cycle** of KD, denoted by $CC(KD)$, is $max_{K \in KD} CC(K)$, i.e., the maximum of the chain cycles among the kernels in KD.

For example, $Ch(B, C) = \{chain1, chain2\}$, and $CC(B, C) = 16$. The chain group is essentially the set of chains that must be used to test the kernels in the associated dependence group concurrently. All these chains must be run at the corresponding chain cycle, i.e., each new pattern must be scanned in over the corresponding number of clock cycles.

Every compatibility group $KC \subseteq V$ can be partitioned into a set of dependence groups by obtaining the connected components of the subgraph $(KC,\ \phi,\ (KC \times KC) \cap D)$. For example, Figure 7.2 shows a TRG with six kernels. The set $\{a, b, c, d, e\}$ is a compatibility group, and it can be partitioned into the three dependence groups $\{a, b\}$, $\{c, d\}$ and $\{e\}$.

The following property of dependence groups helps to construct a test schedule for a set of compatible kernels.

Lemma 9 *Given a compatibility group KC and two of its dependence groups KD_i, KD_j, if $CC(KD_i) \neq CC(KD_j)$ then $Ch(KD_i) \cap Ch(KD_j) = \phi$, i.e., the chain groups are mutually exclusive.*

Proof Assume for the purpose of contradiction that $CC(KD_i) \neq CC(KD_j)$ and $Ch(KD_i) \cap Ch(KD_j) \neq \phi$. Let $c \in Ch(KD_i) \cap Ch(KD_j)$. Then chain c is used for applying tests to some kernel $k_i \in KD_i$ as well as to some kernel $k_j \in KD_j$; hence there must be an edge $\{k_i, k_j\}$ in the TRG. This implies that k_i and k_j must be in the same connected component of the subgraph $(KC,\ \phi,\ (KC \times KC) \cap D)$ of the TRG. This contradicts the assumptions. □

Corollary *Given a compatibility group KC, and a chain c, let KD_i, $i = 1, 2, \ldots, m$ be different dependence groups of KC satisfying $c \in Ch(KD_i)$. Then $CC(KD_1) = CC(KD_2) = \cdots = CC(KD_m)$.*

Proof Consider any pair KD_i, KD_j, $1 \leq i, j \leq m$. Since c belongs to both KD_i and KD_j, the set $Ch(KD_i) \cap Ch(KD_j)$ is nonempty. Lemma 9 implies that $CC(KD_i) = CC(KD_j)$. Applying this argument to all pairs of dependence groups in $\{KD_1 \ldots KD_m\}$, we have $CC(KD_1) = CC(KD_2) = \cdots = CC(KD_m)$. □

The corollary above implies that given a compatibility group KC of kernels that are to be tested concurrently, every scan chain used for applying tests must be associated either with a unique dependence group KD_i within KC or with a subset of dependence groups having the same chain cycle. Hence the following rule can be used for concurrently applying tests to the set of kernels KC in a particular test session.

> **Test Session Execution Rule:** For each chain c, let KD_i be some dependence group, if it exists, such that $c \in Ch(KD_i)$. Then in the current test session c must be operated at the chain cycle $CC(KD_i)$. If no such dependence group exists, chain c is idle in this session.

Based on the model presented in this section, the test scheduling problem can be stated as follows: determine a schedule for applying tests to kernels such that (1) at all times, the kernels being tested form a compatibility group; and (2) the overall test time is minimized. Below we examine a special case of the problem in which there are no dependencies, and show that it is equivalent to the problem studied earlier by Craig et al. [28]. Since Craig's problem is intractable, the special case study also serves to show that the general scheduling problem in the presence of dependencies is intractable. Subsequently, in the following section, we study the general case of test scheduling with dependencies.

7.2.4 The No-Dependence Scheduling Problem

This problem is mostly of academic interest since the chance of an arbitrary circuit having no dependencies is low. It is, however, useful in the special case of a partial circuit design in which the following conditions hold:

1. either there is a single scan path or there are multiple scan paths with exactly the same length;
2. the assignment of registers to scan chains and/or the ordering of registers in each scan chain have not been determined yet or are not known.

In such a case, as we observed earlier, the worst case can be assumed in which every scan chain is flushed completely in order to drive/receive from any kernel; and the chain cycle is taken as N, the length of the single or multiple scan chain(s). Hence during the entire test, the scan chain(s) must operate at a chain cycle of N irrespective of what subset of the kernels is under test at any time.

Let K_{max} be the maximal kernel as defined in Chapter 5, and let its depth be d. Let $KS \equiv \{K_1, \ldots, K_n\}$ be the set of kernels in the circuit obtained by subdividing K_{max}. Note that every kernel $K_i \in KS$ must have depth $d(K_i)$ not higher than d. When applying tests to any kernel K_i, we will assume that each pattern is held in the scan path for $d - d(K_i)$ clock cycles before the test plan for K_i is initiated. This has the advantage that any group of compatible kernels can

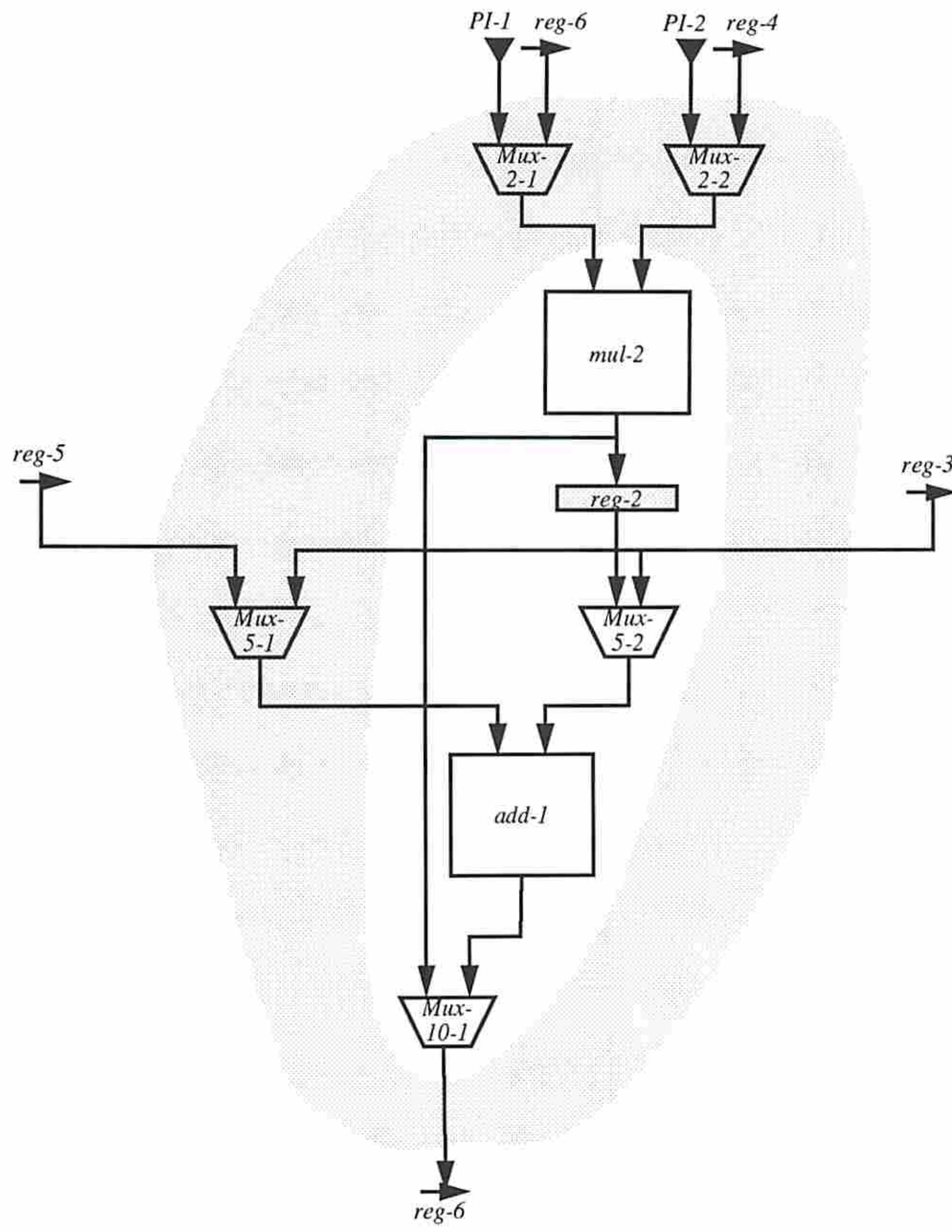


Figure 5.16: Kernel minimization: I-paths and minimal satisfiable kernel.

5.7 Summary

Switches, which occur in almost every circuit in the form of multiplexers and buses, are versatile circuit elements that can help reduce test costs in many ways. This chapter has explored many of these important benefits. First, the presence of switches can be used to expand the class of structures having the single-pattern testable property, and to use this information for more efficient partial scan design. Second, switches help to set up I-paths for transporting test data, which can lead to reduced test generation costs and potentially lower test time. Finally, although this approach preserves the testability of the non-switch functional logic, it provides the option to reduce the scan design overhead (in a postprocessing step) by trading it off against testability of the unexercised switching logic. In the following chapter one additional feature of switches will be encountered: their use in partitioning a circuit in a natural functional manner.

Chapter 6

Partitioned Partial Scan Testing

"United [they] stand, divided [they] fall."
—English proverb, with apologies

6.1 Introduction

In the preceding chapters, we have seen how scan registers can be selected for a sequential circuit such that some form of combinational ATPG can be used to obtain a complete test set for the circuit. Although the combinational ATPG problem is theoretically NP-complete, its complexity is significantly lower than that of sequential ATPG. Thus in general, given a combinational test generation model of moderate size for the circuit under test, tests can be obtained fairly quickly using procedures such as PODEM [27]. ATPG procedures use heuristics which have been shown to reduce the time complexity to approximately $O(n^2)$ or $O(n^3)$ in the average case. However, as the following paragraphs will explain, the complexity may become exponential in large circuits unless some form of partitioning is carried out.

The faults in a combinational circuit under test can be grouped into two categories: irredundant (or detectable) faults and redundant faults. *Irredundant faults* are those that can be detected by a static test consisting of a single test vector;

redundant faults are those that cannot. In other words, combinational ATPG algorithms such as the D-algorithm and PODEM will fail to find a test for a redundant fault. Redundant faults may be caused by additional logic that is provided in a circuit to reduce the existence of hazards and/or races. Unfortunately such faults, if present, may consume an exponential amount of ATPG computation, since the ATPG procedure may search fruitlessly for a test vector until all possibilities are exhausted. (In some cases a timeout is used to limit the backtracking.) Thus, since any redundant faults may contribute an exponential amount of computation, the test ATPG cost for a very large circuit could be dominated by such faults and may become excessively high.

The ATPG cost for a large circuit can be reduced by partitioning it into smaller subcircuits that can be tested individually. This partitioning process could be carried out manually, since human designers can easily identify functional modules or clusters of modules that can be treated as subkernels to be tested relatively independent of each other. It could also be carried out by software, provided the complexity of the partitioning process is relatively low compared to that of the ATPG process itself. Some ideas for automatically partitioning a circuit for test are suggested in this chapter.

Whether the partitioning is carried out manually or automatically, however, two interesting problems arise. (1) Given a test plan and test vectors for each subkernel, how should the various tests be scheduled? There may be interactions among the various subkernels that do now allow them to be tested simultaneously. For example, two kernels that use the same switch for propagating test results cannot be tested in the same test session. (2) Given the number of circuit I/O pins that are available for constructing separate scan path chains, how should the scan registers be configured into one or more scan chains so as to achieve the lowest test time? These two problems, namely *test scheduling* and *scan path chaining*, will be studied in Chapters 7 and 8, respectively.

In the the remainder of this chapter we study an approach for automatic partitioning of an acyclic and/or balanced kernel in a serial scan design. The approach consists of a combination of three different schemes. The first, *output-based partitioning*, breaks up the kernel by identifying all the circuit logic feeding each

be tested in a synchronized manner irrespective of their individual depths. Since in general $d \ll N$, the increase in test time (if any) due to this synchronization is likely to be negligible.

Let each kernel $K_i \in KS$ have test length $T(K_i)$. Then the test time for K_i is $TT(K_i) = (N + d + 1) \times T(K_i)$ clock cycles. This test time is fixed irrespective of what other compatible kernels are tested in parallel with it. In other words, there are no dependencies among the kernels. Hence we can construct a TRG in which there is an incompatibility edge for every pair of incompatible kernels, and there are no dependence edges. This graph is identical to the TIG defined by Craig et al. in [28], and the scheduling problem is identical to their *unequal execution time scheduling problem* with test time for node K_i equal to $TT(K_i)$.

This problem is studied in detail in [28]. It is shown to be intractable, and two different heuristics are presented. The first heuristic breaks down each kernel test into subtests such that all subtests for all kernels have equal duration. A suboptimal algorithm is then used to schedule the resulting set of independent equal-duration tests. The second heuristic is similar to the first, but it assumes that any test for a kernel, once initiated, must run to completion; i.e., all subtests for a kernel must be executed in a contiguous manner with no interruptions.

7.3 General Test Scheduling Algorithm

In the general case of multiple chains that are not all of the same length, the TIG of Craig et al. [28] is not sufficient to model the problem. This is because of the dependencies among kernels, based on shared scan chains, which makes the test time of a given kernel dependent on the rates at which the various scan chains are being operated. Given an arbitrary ordering of the kernels to be tested, a unique test schedule can be obtained by using a greedy scheduling approach to schedule the tests. Based on this idea, an algorithm for obtaining an optimal schedule is presented in this section. It implicitly enumerates all orderings of the kernels and determines the ordering that leads to the schedule with the lowest test time.

7.3.1 Terminology

We begin the discussion on the scheduling algorithm by presenting the terms used in describing schedules. A schedule is essentially a sequence of events that take place over a period of time; we define two types of events, those associated with kernels and those associated with scan chains.

Definition A **kernel event** is a tuple (ts, td, K, cc) where ts is a start time unit (representing a clock cycle) beginning at which tests are applied to kernel $K \in KS$; these tests are applied for a duration of td clock cycles during which all the chains in $Ch(K)$ operate at a chain cycle $cc \geq CC(K)$.

The test for a given kernel K may be distributed over one or more kernel events. Thus for any given kernel event, $td \leq T(K) \cdot (cc + d + 1)$.

Definition A **chain event** is a tuple (ts, td, c, cc) where ts is a start time unit (representing a clock cycle) beginning at which chain c operates at a chain cycle cc for a duration of td clock cycles.

Definition A **schedule** is a pair (KE, CE) where KE is a set of kernel events, CE is a set of chain events, and the earliest kernel/chain event has start time 0. The **length** of a schedule S, denoted by $|S|$, is

$$|S| = \max_{(ts,td,K,cc) \in KE} (ts + td).$$

The length of a schedule is the effective total test time in clock cycles.

A typical schedule for the circuit of Figure 7.1(a) is shown in Figure 7.3(a). Each horizontal bar alongside the name of a kernel represents a kernel event, and the number adjacent to it is the duration of the event. Each range alongside the name of a chain represents a chain event, and the number in parentheses adjacent to it is the cycle at which the chain is operated during this event. In this example there are two kernel events, $(0, 850, A, 16)$ and $(850, 1360, C, 16)$, and two chain events, $(0, 1360, chain1, 16)$ and $(850, 510, chain2, 16)$. This is not a complete schedule for

the circuit since there is no test for B. Note that there are no inconsistencies in this schedule which could invalidate it, e.g., two incompatible kernels being scheduled to be tested simultaneously or the chains in $Ch(K)$ being operated at different cycles or at a cycle lower than $CC(K)$ while the kernel K is being tested.

Thus the scheduling problem is to construct a schedule such that the appropriate number of test patterns is applied to each kernel and the length of the schedule is minimal. We first present a greedy solution approach in which the schedule is grown in an incremental manner, adding tests for one kernel at a time, until all kernels are tested. Clearly the resulting schedule depends on the order in which the kernels are considered. An implicit enumeration algorithm is subsequently presented for exploring all the possible useful orderings of the kernels under consideration.

7.3.2 Incremental Scheduling

Let S' be a partial schedule for testing a subset KS' of the set of kernels KS. S' may or may not be an optimal schedule for KS'. Given a kernel $K \notin KS'$, we wish to expand the schedule S', without disturbing any of the tests already scheduled, so that the resulting schedule S'' contains the test for K and $|S''|$ is minimal. The problem is illustrated for the circuit of Figure 7.1(a) in Figure 7.3. Figure 7.3(a) shows a partial schedule $S' = (KE', CE')$ for the subset of kernels $KS' = \{A, C\}$. KE' contains the kernel events $(0, 850, A, 16)$ and $(850, 510, C, 16)$; CE' contains the chain events $(0, 1360, chain1, 16)$ and $(850, 510, chain1, 16)$. The schedule is to be expanded by including a test for kernel B.

In incrementing the schedule, one of two possible scheduling disciplines can be used. These are analogous to the two scheduling disciplines described in [28] for the BIST scheduling problem. The first discipline, *scheduling without interruptions*, is analogous to *scheduling with run to completion* [28]. It is presented here for completeness; however, it should be noted that running tests to completion is a stronger concern in BIST testing, where seeds and signatures need to be saved temporarily, than in scan testing, where a test can be interrupted and continued after any interval without a large effect on the test control regime. Hence the second

discipline, *scheduling with interruptions*, is more relevant in the context of scan testing.

Scheduling Without Interruptions This discipline requires that every kernel be tested over a contiguous time period. For example, consider the inclusion of the test for B in the schedule S' shown in Figure 7.3(a). B is compatible with A, hence the test for B could begin at time 0, in parallel with the test for A, by operating $chain2$ at a chain cycle of 8. At this chain cycle the test would last for $100 \times (8 + 1) = 900$ clock cycles. However, note that according to CE' $chain2$ has already been scheduled to operate at a cycle of 16 starting at time 850. Since the test would have to be interrupted in order for the two chains to be synchronized, this schedule must be rejected. Instead, to avoid the interruption, $chain2$ must begin operating at a cycle of 16 beginning at time 0. This would ensure that the two chains are always perfectly synchronized. However, since B is now to be tested at a chain cycle of 16 throughout, the test time for B increases to $100 \times (16 + 1) = 1700$, which in fact dominates the entire schedule as shown in Figure 7.3(b).

Scheduling With Interruptions If instead we permit the test for B to be interrupted, the first 94 test patterns for B could actually be applied at a chain cycle of 8 (using $chain2$) at the same time that all 50 test patterns for A are applied at a chain cycle of 16 (using $chain1$). This is shown in Figure 7.3(c). The 94th test for B ends at time 846, while the 50th test for A ends at time 850. Hence $chain2$ must be idle for 4 clock cycles before the two chains can synchronize and operate at the same chain cycle of 16. Once they are synchronized, the remaining 6 test patterns for B can be applied over the next 102 clock cycles, in parallel with the complete test for C. The resulting schedule S'' is shown in Figure 7.3(c). Clearly, by allowing the interruption in the schedule, the test time has been reduced.

The details of the analysis described above are listed in the procedure **firstfit** below, which adds a test for a single kernel K to an existing partial schedule S', resulting in an augmented schedule S''. It is capable of scheduling both with and without interruptions. The procedure essentially searches for an interval in S' during which the test set for K (or some portion of it, if interruptions are allowed) could

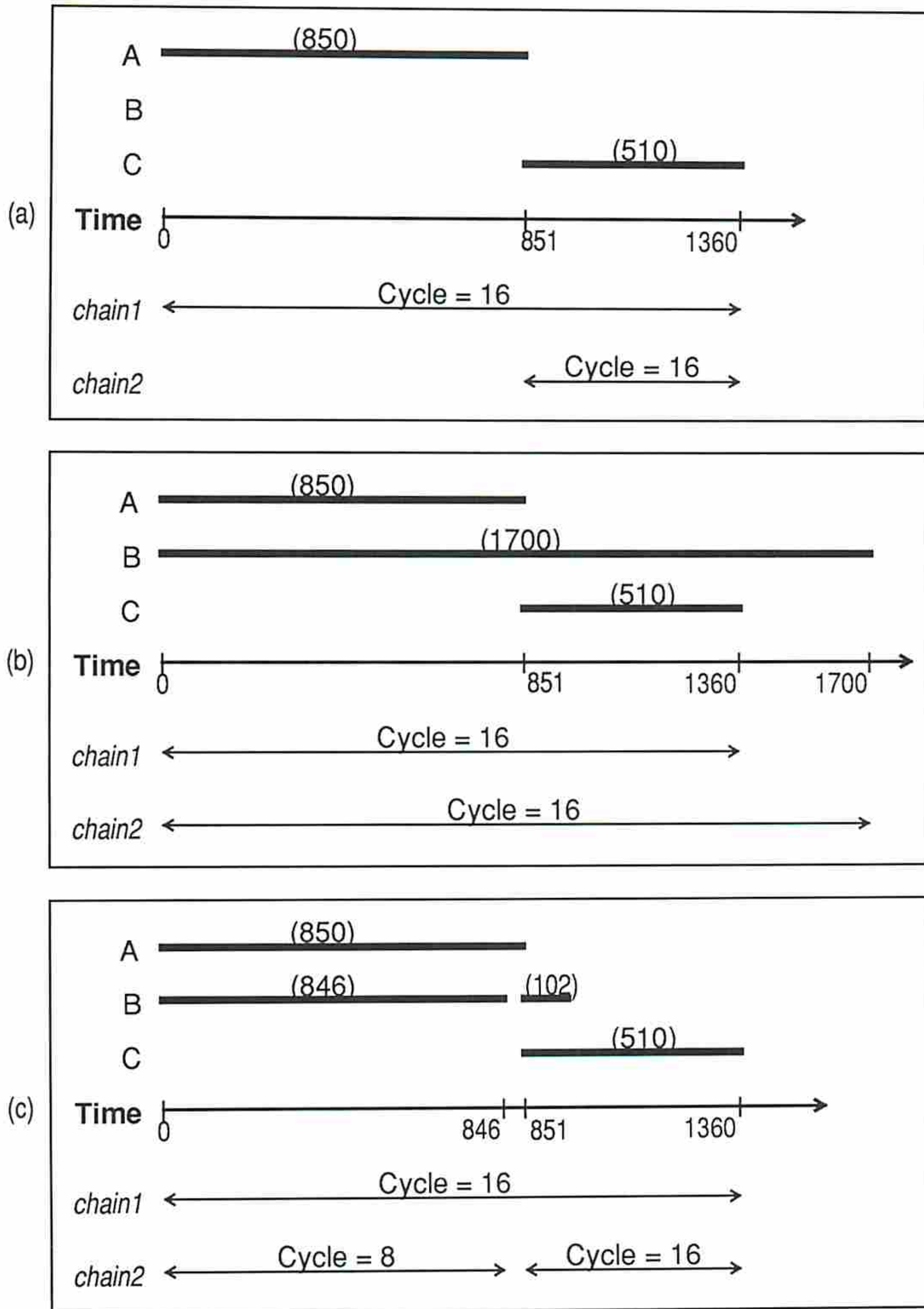


Figure 7.3: Illustration of incremental scheduling: (a) partial schedule S', (b) incremented schedule S'' without interruptions, (c) incremented schedule S'' with interruption.

be applied without clashing with existing tests. A "clash" refers either to another kernel that is incompatible with K and hence prevents K being tested at the same time, or to a set of chains that are already being operated at certain cycles which do not allow K to be tested at the same time.

procedure firstfit (Kernel K, partial schedule $S' \equiv (KE', CE')$): Returns schedule $S'' \equiv (KE'', CE'')$.

1. $TK \leftarrow T(K)$; (remaining number of patterns for K)
   $KE'' \leftarrow KE'$; $CE'' \leftarrow CE'$; (initial schedule)
   $t1 \leftarrow 0$; (tentative start time for test of K)
2. If $TK = 0$ **return** $S'' = (KE'', CE'')$; else continue.
3. (Check for kernel clashes at $t1$.)

   If $\exists(ts', td', K', cc') \in KE'$ such that:

   (i) K and K' are incompatible, *and*

   (ii) $(ts' \leq t1 < ts' + td')$

   then

   $t1 \leftarrow ts' + td'$; go back to step 3;

   else

   continue.
4. (Check for chain clashes at $t1$.)

   Construct RCE, set of chain events that involve chains associated with K and overlap with the test for K:
   $RCE \leftarrow \{(ts', td', c', cc') \in CE' \mid c' \in Ch(K) \text{ and } (ts' \leq t1 < ts' + td')\}$.

   If RCE is empty, then

   $cc \leftarrow CC(K)$; go to step 5.

   Else if there is a positive integer $X \geq CC(K)$ such that $\forall(ts', td', c', cc') \in RCE$, $cc' = X$, then

   $cc \leftarrow X$; go to step 5.

   Else

   $t1 \leftarrow \min_{(ts', td', c', cc') \in RCE}(ts' + td')$; go back to step 3.
5. (No clash exists at $t1$.)

   Determine the number of patterns that can be applied to K starting at $t1$ as follows.

(a) Tentative end-point of the current kernel event:
$t2 \leftarrow t1 + TK \times (cc + d + 1)$, where d is the depth of the original kernel resulting from the partial scan analysis.

(b) (Check all kernel events.)
$\forall (ts', td', K', cc') \in KE'$:

i. If [K and K' are incompatible] and $t1 \leq ts' < t2$ then
$t2 \leftarrow ts'$.

ii. If [interruptions are allowed] and [K and K' are dependent] and $ts' \leq t1 < td' < t2$ then (it might be possible to speed up the chain cycle after K' test is completed at td')
$t2 \leftarrow td'$.

(c) (Check all chain events.)
$\forall (ts', td', c', cc') \in CE'$, if $t1 \leq ts' < t2$ and $c' \in Ch(K)$ and $cc' \neq cc$ then $t2 \leftarrow ts'$.

(d) $t1$, $t2$ are now the end-points of the current interval in which tests for K can be applied. Determine how many patterns can be applied:
$TL \leftarrow \lfloor \frac{t2-t1}{cc+d+1} \rfloor$.

(e) If [$TL = 0$] *or* [no interruptions are allowed *and* $TL < TK$] then
$t1 \leftarrow t1 + \min(1, t2) + 1$ (i.e., move tentative start time for test forward); go to step 3.

6. Compute actual duration of kernel event: $td \leftarrow TL \times (cc + d + 1)$.

   Add kernel and chain events for K to S'':

   $KE'' \leftarrow KE'' \cup \{(t1, td, K, cc)\}$;
   $\forall c \in Ch(K)$, $CE'' \leftarrow CE'' \cup \{(t1, td, c, cc)\}$.

7. Number of test patterns remaining:
   $TK \leftarrow TK - TL$.
   $t1 \leftarrow t1 + td$ (tentative start time for rest of test).
   Go to step 2. □

At the end of Step 4 a chain cycle cc has been decided for the kernel K depending on what chain events, if any, are in progress at the start time $t1$. If there are none, cc simply defaults to $CC(K)$. In Step 5, this chain cycle value is taken as fixed; thus if any chain required for testing K has already been scheduled to run at a different chain cycle, say X, during a later period, the test for K cannot overlap with this period. However, a variation on the procedure listed above could allow the chain cycle selected for K to be increased to X so as to allow it to overlap with the

subsequent chain event with cycle X. Note that this would only be possible when no chain event was in progress at the start time $t1$ and hence the default chain cycle of $CC(K)$ was selected in Step 4. The schedule example of Figure 7.3(b) is actually based on this variation.

Given a fixed ordering of kernels, the procedure **firstfit** above can be used to construct a schedule in incremental steps based on the ordering. This is done by the procedure **scheduleOrder** listed below. The issue of determining a good ordering is dealt with in the next section.

procedure scheduleOrder (Sequence of kernels, $KSEQ$): Returns complete schedule $S \equiv (KE, CE)$.

1. $S \leftarrow (\phi, \phi)$.
2. $\forall K \in KSEQ$, in order:
   $S \leftarrow$ **firstfit**(K, S).
3. **Return** S. □

Thus with both scheduling disciplines (i.e., with or without interruptions), given an ordering of the kernels, a schedule can be constructed in an incremental greedy fashion by scheduling one kernel at a time. Clearly the order of considering the kernels affects the overall test time. For example, in the example of Figure 7.3(b), the fact that C was scheduled before B made it impossible to fit in the test for B before the test for C without a change in the operating cycle of $chain2$. If instead B had been scheduled first, the test for B could initially have been scheduled to run uninterrupted until time 900, with $chain2$ operating at a cycle of 8; the test for C could then have been applied over the next 510 clock cycles, resulting in a shorter schedule with length 1410. Since the schedule is clearly sensitive to the ordering of the kernels, we present below an algorithm that implicitly enumerates all possible orderings while searching for the one that leads to the minimal-length schedule.

7.3.3 Optimal Scheduling

In this section an algorithm **schedule** for generating a schedule for a set of kernels KS is presented. The algorithm implicitly enumerates all possible orderings of the kernels in KS, based on some default ordering $KSEQ$, and determines the ordering that results in the shortest schedule. Note that it is the ordering that is actually optimal, and the associated schedule is optimal under the constraint that the scheduling process is carried out incrementally, one kernel at a time.

Before listing the algorithm **schedule**, we will illustrate the scheduling process using the example of Figure 7.1(a). For this circuit let the default ordering $KSEQ$ be (A, B, C). Assume also that the scheduling discipline for this circuit allows interruptions in tests for kernels.

The first action is to generate a schedule for the kernels in the default order. This can be seen in the search space diagram of Figure 7.4. Starting at the root node, the leftmost path traces the incremental evolution of the schedule as the kernels are considered in the order A, B, C. At each node the sequence of kernels scheduled so far is listed along with the total test time for the partial schedule for these kernels. The schedule generated for the complete sequence (A, B, C) is shown in Figure 7.5(a), and the total test time at this state is 1410 clock cycles. This schedule will serve as the current "best schedule" until a better schedule (if any) is found.

The algorithm now backtracks to generate the new states (A, C) and (A, C, B); the schedules generated at these states are shown in Figure 7.3(a) and (c), respectively. The latter schedule contains an interruption in the test for B; however, the resulting test time is 1360, hence it replaces the current best schedule.

The generation of new states, or *branching*, continues in the depth-first search order of the state space shown in Figure 7.4. At each state the current schedule is generated and its length is compared with that of the current best schedule. The schedule at the leaf state (B, A, C) is identical to the one for (A, B, C) shown in Figure 7.5(a), with length 1410. At the next visited state (B, C) the resulting schedule has length 1410; this is greater than the current best schedule length 1360.

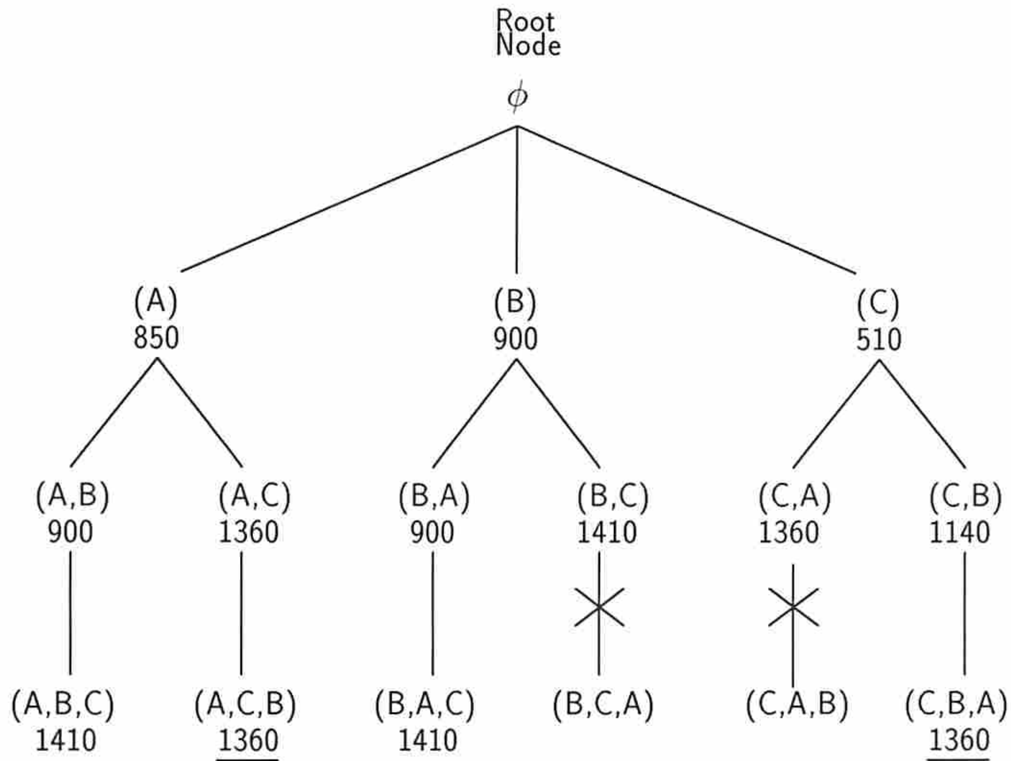


Figure 7.4: Implicit enumeration search space for generating a schedule.

Hence this state is removed from further consideration, called *pruning*; in other words, no further branching is carried out from this state. Instead, the algorithm backtracks to the preceding states in order to continue branching. Pruning also takes place at the state (C, A), at which the length of the partial schedule is 1360, and clearly any states generated by branching from this state must have a length at least equal to that of the current best schedule.

The schedule generated for the final state (C, B, A) is shown in Figure 7.5(b). The test time is 1360, which is equal to the length of the current best schedule. However, the new schedule has the advantage that there is no interruption in the test for any kernel, hence it may be preferred as the best schedule. Thus the result of the scheduling algorithm is the schedule of Figure 7.5(b) with test time 1360 clock cycles.

The details of the scheduling algorithm are presented in the procedure **schedule** listed below. The procedure maintains a series of states $ST_0, \ldots, ST_i$, where ST_0 is the root state, ST_i is the current state, and others represent the intermediate states

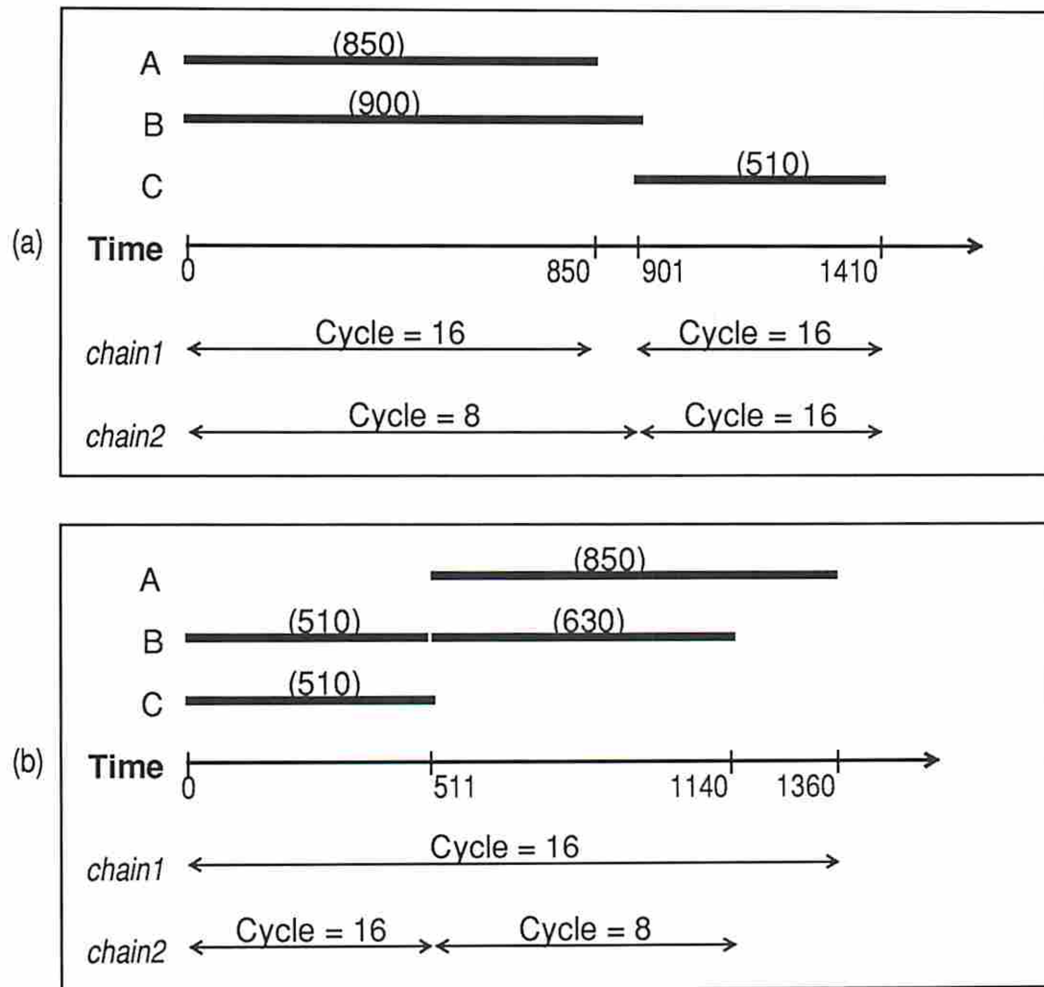


Figure 7.5: Schedules generated by the search algorithm. (a) Schedule for (A, B, C) as well as (B, A, C); (b) schedule for (C, B, A).

in the path from ST_0 to ST_i in the search space. Each state ST_j consists of a tuple $(KSEQ_j, SCH_j, NK_j)$ where $KSEQ_j$ is the sequence of kernels currently being scheduled, SCH_j is the corresponding partial schedule, and NK_j is a sequence of kernels whose tests need to be added to the current schedule by considering them in some order so as to make the schedule complete. Each element of NK_j is used for branching out to a new subtree of states. As the search proceeds, once an element has been used to create a new branch, it is removed from NK_j.

procedure schedule (Sequence of kernels $KSEQ$): Returns SCH, schedule for the kernels in $KSEQ$.

1. (Generate initial root state ST_0)
   $KSEQ_0 \leftarrow ()$; $SCH_0 = (\phi, \phi)$; $NK_0 \leftarrow KSEQ$.

   Initial value of state counter: $i \leftarrow 0$.
   Initial "best" test time: $TT \leftarrow \infty$.

2. (Check if "best schedule" needs to be updated)
   If $i = |KSEQ|$, i.e., all kernels are tested in the current schedule, then if $(|SCH_i| < TT)$ or $(|SCH_i| = TT$ and SCH_i has fewer interruptions than $SCH)$, then $SCH \leftarrow SCH_i$ and $TT \leftarrow |SCH_i|$.

3. (Backtrack if necessary)

   If $NK_i = ()$, i.e., no more branching is possible from this state, then determine the highest value of j if one exists, $0 \leq j < i$, such that $NK_j \neq ()$.

   If there is no such j (search is completed) **return** SCH;

   Else (make ST_j the current state) $i \leftarrow j$.

4. (Branch out from state ST_i)

   $K1 \leftarrow$ first element of NK_i; remove $K1$ from NK_i.
   (Generate ST_{i+1})
   $KSEQ_{i+1} \leftarrow KSEQ_i$ with $K1$ appended;
   $NK_{i+1} \leftarrow$ sequence of kernels that are in $KSEQ$ but not in $KSEQ_{i+1}$, in order of occurrence in $KSEQ$.

   (Generate new schedule by adding the new kernel to the old schedule)
   $SCH_{i+1} \leftarrow$ **firstfit**$(K1, SCH_i)$.

   (Prune if possible)
   If $|SCH_{i+1}| \geq TT$ then $NK_{i+1} \leftarrow ()$.

5. (Move to the newly-created state)
   $i \leftarrow i + 1$;
   Go to step 2. □

The search tree generated for the circuit of Figure 7.1(a) based on a scheduling discipline that allows interruptions is shown in Figure 7.4. At each node, the procedure **schedule** simply invokes **firstfit** to build a new schedule in an incremental manner. Thus the scheduling discipline in **schedule** is the same as that enforced in **firstfit**.

7.3.4 Discussion

For the circuit of Figure 7.1, if the three kernels were tested one after another without overlapping, the overall test time would be the sum of the independent test times for A ($50 \times 17 = 850$), B ($100 \times 9 = 900$), and C ($30 \times 17 = 510$), which is 2260 clock cycles. One of the optimal schedules generated by the procedure **schedule** is shown in Figure 7.5(b); the total test time is 1360 clock cycles. This result is influenced by certain characteristics of the circuit design which are discussed below.

The configuration of the scan chains in this example ensures that kernels A and B are *independent*, i.e., can be tested "out of sync" with each other. This feature allows B to be tested relatively quickly by using the shorter scan chain $chain2$, without having to be in sync with the longer scan chain $chain1$ used for testing A and C. Clearly a different configuration of scan chains, say with half the width of $R1$ moved out of $chain1$ into $chain2$ so as to equalize the chain lengths, might degrade the solution. In this situation the test for B would dominate the test schedule, and the chains would both have length 20; the resulting test time would be $100 \times (20 + 1) = 2100$ clock cycles, ignoring the flushing time for the scan chain at the end of the test.

Another feature that influences the test schedule is the manner in which the logic under test in the circuit is clustered into kernels. Different ways of forming the subkernels by partitioning would lead to different schedules. Consider for example the unpartitioned kernel K consisting of A, B, C along with M, and assume that

K is to be treated as a single kernel for the purposes of test generation and test application. K would involve a somewhat higher effort in test generation than the subdivided kernels. The resulting test set for K would probably have at least 100 patterns. With two chains of equal test length, the test time would be at least $100 \times (20 + 1) = 2100$ clock cycles. Thus in this example subdividing the kernel is clearly beneficial with respect to both test generation and test application. In other cases, however, it is possible that subdividing the kernel may lead to a higher test time, implying a tradeoff between the costs of test generation and test application. In general it is difficult to predict *a priori*, during kernel formation, the influence of different kernel partitioning strategies on these two measures. It is hoped that the ideas presented in this scheduling study may be useful in future research dealing with the development of estimators to guide the partitioning process itself.

7.4 Summary

Chapter 6 showed how the maximal kernel can be partitioned into subkernels for the purpose of test. The subdivision of the kernel gives rise to the problem of scheduling the tests for the various kernels so as to minimize the total test time. In this chapter we have assumed that the assignment of scan registers to scan chains and the ordering of registers in the scan chains have already been decided. Based on this assumption, the various relationships among kernels with respect to test scheduling have been studied, and an implicit enumeration algorithm for obtaining an efficient schedule has been presented. It is clear that there is a strong interrelationship between the problem of constructing the single/multiple scan chain(s) and the test scheduling problem studied above. The scan path chaining problem is tackled in Chapter 8.

Chapter 8

Scan Path Chaining

"When no exact solution to a difficult problem can be found, look for either an approximate solution to the complete problem, or an exact solution to a simplified form of the problem."

—Source unknown

8.1 The Chaining Problem

The scan design approach can greatly reduce the cost of test pattern generation for general sequential circuits. However, it constrains test data (both input patterns and output results) to be shifted in and out of the circuit in a manner, typically through a single serial scan path. Usually a large proportion of the test time is taken up in this shifting process while only a small proportion is used for actually propagating test data through the circuitry under test. As circuit sizes increase, the amount of test data passing through the scan-in and scan-out pins increases correspondingly, leading to test application times that may be unacceptably high.

The high test time caused by the need for serial shifting of test data can be reduced in two ways. The first is to distribute the scan registers among a set of multiple scan chains that operate in parallel with their own scan-in and scan-out pins. Note that during the process of scanning test data, all pins that do not serve as scan-in or scan-out pins are idle. Hence by using multiplexers, some or all of the

system I/O pins can be made to serve as scan-in or scan-out pins while test data is being scanned. Additional unused pins available on a chip package can also be used for this purpose. In this way the rate of test data entering/leaving the circuit can be maximized. The second approach to reducing test time is to order the registers in a scan chain such that registers that need to be accessed more frequently are closer to the scan-in/scan-out pins. This reduces the average time for applying tests to the circuit.

In general the scan path chaining problem can be stated as follows: *given a circuit, the test length (number of test vectors) for each kernel, the set of scan registers (each having one or more FFs), and the number of scan chains that can be used, (1) determine an assignment of scan registers to chains, and (2) determine the ordering of the registers in each chain, such that the test time for the resulting design is minimized, and constraints on scan path routing area and/or test control complexity (if any) are satisfied.* In this chapter some studies of the scan path chaining problem are carried out with a view to understanding the nature of the problem and its interaction with the test scheduling problem studied in Chapter 7.

Recognizing the difficulty of the general chaining problem, in this chapter we examine two special restricted cases for which accurate solutions can be found more easily. Both involve designs in which all kernels are compatible with each other, which we will call **fully compatible designs**. In Section 8.2 we describe three canonical test application schemes for such designs. In the following two sections we consider two cases of the chaining problem for fully compatible designs; Section 8.3 deals with the single chain problem, and Section 8.4 deals with a specified number of multiple chains for a restricted class of circuits, namely those with two kernels. The latter study brings out an important fact, namely that an optimal configuration of multiple chains may actually require the chains to have unequal lengths. In both problems we assume that the objective is to minimize test time without regard to routing area or control complexity.

This chapter essentially serves to open up a Pandora's box of scan path chaining problems. By providing a glimpse of a couple of the problems, and showing how they relate to the test scheduling problem, it will hopefully serve as a precursor to a much more comprehensive analysis of this difficult problem.

8.2 Test Application in Fully Compatible Designs

In a fully compatible design, any arbitrary subset of kernels can be tested simultaneously without conflicts. An example of a fully compatible design is a full scan design in which every *cloud* of connected combinational logic (as defined in Chapter 3) forms a separate kernel. Another example is a partial scan design in which the only form of partitioning used to decompose the kernel is size-based partitioning. Note that in a partial scan design with no partitioning, there is only one kernel and the design is obviously fully compatible.

As we have already mentioned, the main difficulty in the general chaining problem is its close relationship with the general test scheduling problem, which was dealt with in Chapter 7. However, in the case of fully compatible designs the optimal scheduling of tests is much more straightforward than in the general scheduling problem. This makes the chaining problem for fully compatible designs somewhat manageable, and is the motivation for the case studies in this chapter.

We begin by examining the simplified test scheduling problem for fully compatible designs given scan chains that have already been constructed. There are three ways in which tests can be applied to the kernels in such a circuit. These are called the *combined, separate* and *overlapped* schemes, respectively. These schemes vary in control complexity and in relative test application time. Each results in a distinct schedule for testing the various kernels. In discussing the schemes we will use the following notation, mostly borrowed from Chapter 7.

Let KS be the set of kernels to be tested, $\{K_1, \ldots, K_n\}$. Ch is the set of chains; $Ch(K_i)$ is the set of chains involved in testing K_i. Given a set of kernels $KC \subseteq KS$, we define the **drive cycle** $DC(KC)$, the **receive cycle** $RC(KC)$, and the chain cycle $CC(KC)$ of a set of kernels exactly as in Chapter 7.

8.2.1 Combined Test

The combined test scheme is the traditional scheme used in applying tests to scan-based circuits, in which tests are applied to all the kernels simultaneously. Note that

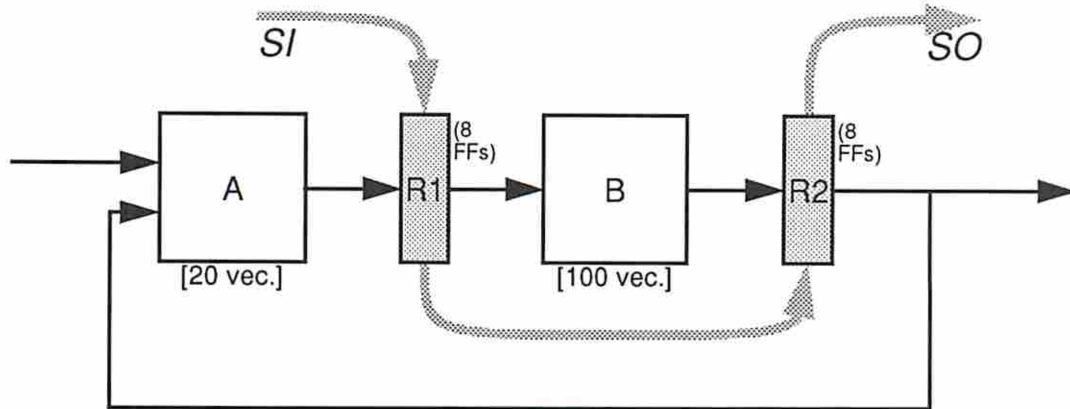


Figure 8.1: Circuit example for illustrating different test schemes.

no scan register can drive more than one kernel and no scan register can receive from more than one kernel; otherwise there would be incompatibilities among the kernels. Thus it is possible to merge the test sets for the kernels into a single test pattern set of length $T = \max_{K_i \in KS} T(K_i)$. To apply each test, the scan chain is clocked in the SHIFT mode L times, where L is the maximum length (i.e., maximum number of FFs) among the scan chains in Ch. Thus the total test time in clock cycles is

$$TT_{COM} = T \times (L + d + 1) + L,$$

where d is the maximal kernel depth as defined in Chapter 7, and the last L clock cycles serve to flush out the scan chain(s) after the last pattern has been applied.

Consider the circuit in Figure 8.1, in which A and B are the two kernels, both combinational. Both scan registers $R1$ and $R2$ are 8 bits wide and are connected into a single scan chain. A has 20 test vectors and B has 100. Using the combined test scheme, in effect A and B are treated as a single kernel with 100 test patterns. Each pattern is applied by shifting it into the entire scan chain over 16 clock cycles and then loading the test result into the scan chain in the next clock cycle. The total test time is $100 \times 17 + 16 =$ **1716** clock cycles.

In the combined test scheme, the control and clocking schemes are relatively simple since the same sequence of operations on the scan path (i.e., shift L times, hold d times, load once) is repeated throughout the entire test. Another feature

is that the test time is independent of the ordering of registers in the scan chains. Thus in the case of a single scan chain, the routing of the scan path can be carried out early in the physical design stage even if estimates of the test lengths of the kernels are not yet available. However, the practice of flushing all chains to apply each test vector can be wasteful since not all scan registers may need to be accessed every time. The following two schemes address this issue at the expense of more complicated control and clocking.

8.2.2 Separate Test

In the separate test scheme, every kernel K_i is tested in its own private test session TS_i during which no other kernels are tested. The motivation for this scheme is that if the test lengths of the kernels have wide differences, as for example in the circuit of Figure 8.1, then it is wasteful to combine the tests for the kernels and to flush the entire scan chain for each test. Instead, the various kernels can be tested independently, and in each session TS_i a minimal amount of shifting, i.e., $CC(K_i)$ clock cycles per test, can be carried out to apply tests to the current kernel K_i. Thus the test time per session is $TT_{SEP}(K_i) = T(K_i) \times (CC(K_i) + d + 1)$, and the overall test time is

$$TT_{SEP} = \sum_{K_i \in KS} TT_{SEP}(K_i) \;+\; \max_{K_i \in KS} CC(K_i).$$

The second term in the expression above represents the extra clock cycles used up in flushing out portions of the scan chains between sessions. If the kernels are tested in order of increasing test length, all this flushing takes place at the end of the last session. It is clear from the discussion above that the separate scheme requires more complex control than the combined scheme, since there are multiple test sessions with different chain cycles.

In the circuit of Figure 8.1, A can be tested in the first test session and B in the second. The chain cycle $CC(A)$ is 16, hence the test time for A is $20 \times 17 = 340$. In the second session, due to the ordering of the scan chain shown, the chain cycle $CC(B)$ is 8, hence the test time is $100 \times 9 = 900$. The overall test time is $340 + 900 + 16 = \mathbf{1256}$. In this example TT_{SEP} is lower than TT_{COM}. Note that the

ordering of the scan chain in this case is favorable to separate testing. In some cases, however, the test time for separate testing may be higher than that for combined testing even with the most favorable scan chain ordering.

8.2.3 Overlapped Test

In the overlapped test scheme, the kernels in KS are ordered according to nondecreasing test length. Let $KSEQ \equiv (K_1, K_2, \ldots, K_n)$ be the sequence of kernels in KS ordered in this manner; i.e., $T(K_1) \leq T(K_2) \leq \cdots \leq T(K_n)$. Consider the circuit of Figure 8.1, for which $KSEQ = (A, B)$. The test is carried out in two sessions TS_1, TS_2. In TS_1, tests are applied to both A and B simultaneously until the 20 test vectors for A are exhausted. During this session test patterns must be scanned into/out of both $R1$ and $R2$, and the scan chain is operated at a cycle $CC(A, B) = 16$. Hence the test time in TS_1 is $20 \times 17 = 340$. In TS_2, the remaining 80 test patterns for B are applied in a separate manner. Now the chain can be operated at a cycle of $CC(B) = 8$, and the session test time is $80 \times 9 = 720$. The overall test time is $340 + 720 + 16 = \mathbf{1076}$, including the flushing time for the scan chain.

In general, given the sequence of n kernels $KSEQ$, the following procedure can be used to apply the tests using the overlapped scheme:

(Session TS_1)
Apply $T(K_1)$ tests to all kernels $K_1, \ldots, K_n$.
For $i = 2$ to n do:
{
(Session TS_i)
Apply $T(K_i) - T(K_{i-1})$ test patterns to kernels $K_i, \ldots, K_n$.
}

Let $TT_{OVL}(TS_i)$ denote the test time for test session TS_i. Thus for the first session, $TT_{OVL}(TS_1) = T(K_1) \times (CC(K_1, \ldots, K_n) + d + 1)$. For all other sessions,

$TT_{OVL}(TS_i) = (T(K_i) - T(K_{i-1})) \times (CC(K_i, \ldots, K_n) + d + 1)$. The overall test time is

$$TT_{OVL} = \sum_{i=1}^{n} TT_{OVL}(TS_i) \; + \; CC(K_1, \ldots, K_n),$$

where the second term represents the total flushing time.

8.2.4 Comparison

The *overlapped* test scheme is the most efficient of the three schemes, i.e., for all circuits, $TT_{OVL} \leq TT_{COM}$ and $TT_{OVL} \leq TT_{SEP}$. This is proved formally in the two theorems below.

Theorem 5 $TT_{OVL} \leq TT_{COM}$.

Proof

$$\begin{aligned} TT_{OVL} &= \sum_{i=1}^{n} TT_{OVL}(TS_i) \; + \; CC(K_1, \ldots, K_n) \\ &= T(K_1) \cdot (CC(K_1, \ldots, K_n) + d + 1) \\ &\quad + \sum_{i=2}^{n} (T(K_i) - T(K_{i-1})) \cdot (CC(K_i, \ldots, K_n) + d + 1) \\ &\quad + CC(K_1, \ldots, K_n). \end{aligned}$$

For all i, $(CC(K_i, \ldots, K_n) + d + 1) \leq (CC(K_1, \ldots, K_n) + d + 1)$; hence

$$\begin{aligned} TT_{OVL} &\leq (CC(K_1, \ldots, K_n) + d + 1) \cdot \left[T(K_1) + \sum_{i=2}^{n} (T(K_i) - T(K_{i-1})) \right] \\ &= (CC(K_1, \ldots, K_n) + d + 1) \cdot T(K_n). \end{aligned}$$

Thus $TT_{OVL} \leq TT_{COM}$. □

Theorem 6 $TT_{OVL} \leq TT_{SEP}$.

Proof

$$\begin{aligned} TT_{OVL} &= \sum_{i=1}^{n} TT_{OVL}(TS_i) + CC(K_1,\ldots,K_n) \\ &= T(K_1)\cdot(CC(K_1,\ldots,K_n)+d+1) \\ &\quad + \sum_{i=2}^{n}(T(K_i)-T(K_{i-1}))\cdot(CC(K_i,\ldots,K_n)+d+1) \\ &\quad + CC(K_1,\ldots,K_n). \end{aligned}$$

Using the definition of $CC(K_i,\ldots,K_n)$ from Section 7.2.3, we have

$$\begin{aligned} CC(K_i,\ldots,K_n)+d+1 &= \max_{i\le j\le n}[CC(K_j)]+d+1 \\ &= \max_{i\le j\le n}[CC(K_j)+d+1] \\ &\le [CC(K_i)+d+1]+[CC(K_{i+1})+d+1]+ \\ &\quad \cdots+[CC(K_n)+d+1]. \end{aligned}$$

$$\begin{aligned} &\Rightarrow TT_{OVL} \le \\ &\quad T(K_1)\cdot([CC(K_1)+d+1]+\cdots+[CC(K_n)+d+1]) \\ &\quad + \sum_{i=2}^{n}(T(K_I)-T(K_{I-1}))\cdot([CC(K_i)+d+1]+\cdots+[CC(K_n)+d+1]) \\ &\quad + CC(K_1,\ldots,K_n). \end{aligned}$$

On the right hand side, collect terms containing $[CC(K_j)+d+1]$ for each j. This results in

$$\begin{aligned} TT_{OVL} &\le T(K_1)\cdot[CC(K_1)+d+1]+T(K_2)\cdot[CC(K_2)+d+1]+ \\ &\quad \cdots+TT(K_n)\cdot[CC(K_n)+d+1]+CC(K_1,\ldots,K_n) \\ &= \sum_{i=1}^{n} T(K_i)\cdot(CC(K_i)+d+1)+CC(K_1,\ldots,K_n) \\ &= \sum_{i=1}^{n} TT_{SEP}(K_i)+CC(K_1,\ldots,K_n). \end{aligned}$$

Thus $TT_{OVL} \le TT_{SEP}$. □

Thus we have shown that the overlapped test scheme always leads to a lower test time than the other two schemes in all circuits, irrespective of the scan chain organization. However, the *combined* scheme has two advantages over the overlapped and separate schemes: (1) test control and clocking are simpler, and (2) the ordering of registers in the scan chain is not important, which implies that the scan path routing can be carried out purely on geometric considerations and without referring to the test lengths of various kernels. Note that there is no situation in which the *separate* scheme is clearly superior; it has approximately the same control complexity as the overlapped scheme, yet it is not as efficient. Although we have described all three test schemes for completeness, in the remaining discussions on the chaining problem for fully compatible circuits, our attention will be focussed mostly on the combined and overlapped schemes.

Two case studies of the chaining problem for fully compatible circuits are presented in the following sections. In each we consider a constrained version of the problem. The first deals with constructing a single scan chain for a fully compatible design. For this problem we assume the overlapped test scheme, and present a heuristic whose output includes a *confidence level* indicating the degree of optimality of the solution. The second study deals with constructing multiple scan chains for fully compatible designs. Because the problem is highly complex, we consider only the two-kernel case and develop an optimal solution method.

8.3 Single Chain in Fully Compatible Design

In this section we consider the problem of constructing a single scan chain for a fully compatible circuit such that the test time, based on the *overlapped test scheme*, is minimal. We begin by defining some terminology.

Let the circuit have N flip-flops (FFs) $FS \equiv \{F_1, F_2, \ldots, F_N\}$. Note that registers must be broken down into individual FFs; this is because an optimal ordering of the scan chain may require the FFs in a scan register to be in different parts of the chain. The scan chain to be constructed can be represented by a sequence of N **slots** numbered $1, 2, \ldots, N$ to which scan FFs are to be assigned. The first

Figure 8.2: Example of single scan path chaining problem.

slot represents the FF closest to the scan-in pin and the last slot represents the FF closest to the scan-out pin. The chaining problem is to assign one FF to each slot, i.e., to construct a **position mapping** $P : FS \rightarrow [1, N]$, where the notation $[x, y]$ represents the set $\{\text{integers } i \mid x \leq i \leq y\}$.

The idea used in our approach is to determine, for every FF F_j, a range of slots $Range(F_j) \equiv [low(F_j), high(F_j)]$ in which it may be placed in order for the chain ordering to be optimal. We will show that given this information it is simple to determine (based on the well-known *perfect matching problem*) an optimal solution, i.e., a mapping P that satisfies the range constraints, *if* such a solution exists.

8.3.1 Flip-Flop Position Ranges

Consider the circuit in Figure 8.2. There are three kernels, A, B, C, with test lengths 20, 30, 100, respectively, and the sequence of kernels in order of nondecreasing test length is $KSEQ = (A, B, C)$. All the five registers shown, $R1 \ldots R5$, are to be connected in a single scan chain so as to minimize the test time under the overlapped scheme. For simplicity we assume that every register has exactly one FF. Since $KSEQ$ has three kernels, the test for the circuit is organized into three sessions TS_1, TS_2, TS_3 as described in Section 8.2.3.

Let KS_i denote the set of kernels that are tested in session TS_i. Thus $KS_1 = \{A, B, C\}$, $KS_2 = \{B, C\}$, and $KS_3 = C$. In a fully compatible circuit, each scan FF can serve to drive test patterns into exactly one kernel and to receive test results from exactly one kernel. In a given session TS_i, depending on what subset of kernels are being tested, each FF may or may not be involved in driving test patterns, and may or may not be involved in receiving test results. Thus we can partition the set of scan FFs into four sets according to what roles they play in test session TS_i. The **idle set** I_i is the set of scan FFs that neither drive nor receive from any kernel in KS_i; in other words, they play no role in this session. The **driver set** D_i (**receiver set** R_i) is the set of scan FFs that drive (receive from) some kernel in KS_i. Finally, the **driver-receiver set** C_i is the set of scan FFs that drive some kernel in KS_i as well as receive from some kernel in KS_i. Based on these four sets we define $i_i \equiv |I_i|$, $d_i \equiv |D_i|$, $r_i \equiv |R_i|$, and $c_i \equiv |C_i|$.

For example, in session TS_3 only C is under test; thus $D_3 = \{R2, R4\}$; $R_3 = \{R5\}$; and $C_3 = \phi$. Since there are two driver FFs and one receiver FF, it means that every time a test is applied in session TS_3, two bits of useful data must be shifted into the scan chain and at least one bit of useful data must be shifted out. It is clear that irrespective of the ordering of the scan chain, at least two shift operations are required to achieve this. We refer to this number as the *minimum session cycle*, defined as follows.

Definition In a given test session TS_i, the **minimum session cycle** MSC_i is the minimum number of clock cycles, over all possible orderings of the scan chain, that are required to shift in a new test pattern while shifting out the result of the previous test.

In other words, the minimum session cycle MSC_i is the lowest possible *chain cycle* that could be used in session TS_i assuming that the chain could be ordered solely so as to minimize this chain cycle. In our example $MSC_3 = 2$, and the general rule is stated below:

$$\text{If } C_i = \phi \text{ then } MSC_i = \max(d_i, r_i). \tag{8.1}$$

There may be a large number of possible ways to position the non-idle scan FFs in the scan chain so as to achieve the minimum session cycle for this session. One such assignment is $P(R2) = 1$, $P(R4) = 2$, and $P(R5) = 4$; another is obtained by exchanging the positions of $R2$ and $R4$. In fact any assignment such that all driving FFs lie in the first two slots and all receiving FFs lie in the last two slots must satisfy the minimum session cycle. The positioning of FFs that are idle in this session is immaterial since they do not influence the test time in this session. In general for every FF F_j, if F_j serves as a driver it is desirable to position it within the *first* MSC_i slots in the chain; and if it serves as a receiver it is desirable to position it within the *last* MSC_i slots in the chain. Thus we can compute $Range_i$, the range of positions to which F_j may be assigned so as to minimize the test time in session TS_i, as follows:

$$Range_i(F_j) = \begin{cases} [\ 1 \ , \ N \ ] & \text{if } F_j \in I_i \\ [\ 1 \ , \ MSC_i \ ] & \text{if } F_j \in D_i \\ [\ N - MSC_i + 1 \ , \ N \ ] & \text{if } F_j \in R_i \\ [\ N - MSC_i + 1 \ , \ MSC_i \ ] & \text{if } F_j \in C_i \end{cases} \tag{8.2}$$

Note that FFs in the driver-receiver set C_i must lie in the middle portion of the chain; this will be elaborated on shortly. Thus for session TS_3 we have $Range_3(R2) = Range_3(R4) = [1,2]$; $Range_3(R5) = [4,5]$; and $Range_3(R1) = Range_3(R3) = [1,5]$. These ranges are illustrated in Figure 8.3(a).

Now consider session TS_2. We have $D_2 = \{R2\}$, $R_2 = \phi$, and $C_2 = \{R4, R5\}$. Once again we wish to compute the minimum session cycle, i.e., the lowest possible chain cycle among all orderings of the chain. Clearly this can be achieved by placing $R2$ nearest to the scan-in pin, placing $R4$ and $R5$ close to the middle of the chain, and placing the remaining idle FFs arbitrarily. For example, the ordering $(R2, -, R4, R5, -)$ (where "$-$" represents some idle FF) leads to a minimal chain cycle, i.e., 4. In this case it is not possible to have a chain cycle lower than 4 with any ordering, hence the minimum session cycle must be 4. Below we show how to formally derive this value.

In general, to compute the minimum session cycle MSC_i when C_i is non-empty, we can construct a hypothetical chain that is optimal for the current session

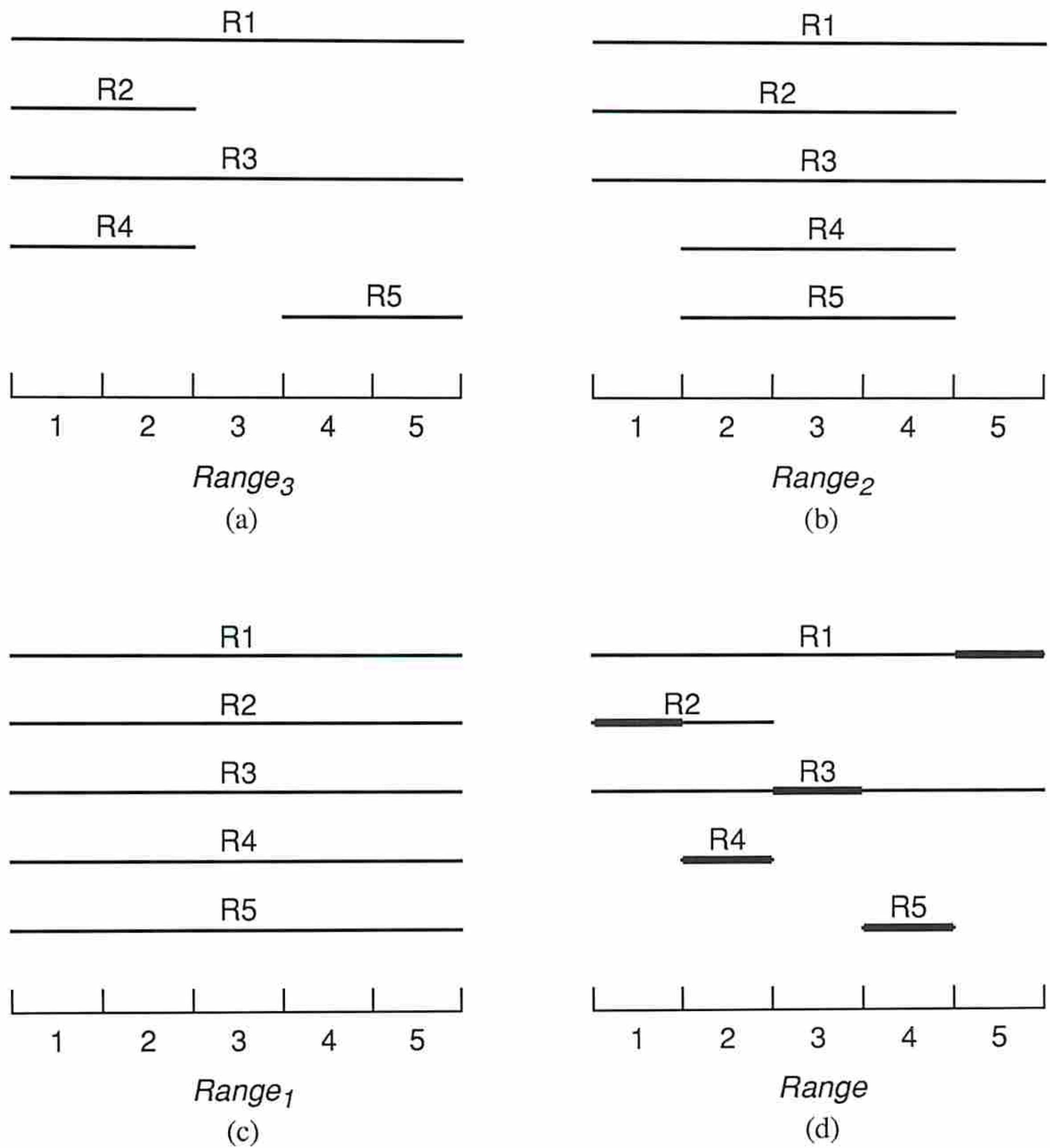


Figure 8.3: Ranges for placement of scan FFs. (a) For session TS_3, (b) for session TS_2, (c) for session TS_1, (d) combined ranges.

as follows. Ideally the c_i driver-receiver FFs in C_i must be placed in the middle of the scan chain, and the remaining $N - c_i$ FFs (including the FFs in D_i and R_i) must be distributed equally on either side. However, depending on the values of d_i and r_i, the hypothetical chain may not have the C_i FFs in the exact middle.

Case 1 If both r_i and d_i are less than or equal to $\lceil (N - c_i)/2 \rceil$, then the c_i FFs in C_i must be placed in a group in the middle of the chain, with all the D_i FFs preceding this group and all the R_i FFs following this group in the chain. This is illustrated in Figure 8.4(a). Thus the minimum session cycle is $\lceil (N - c_i)/2 \rceil + c_i$.

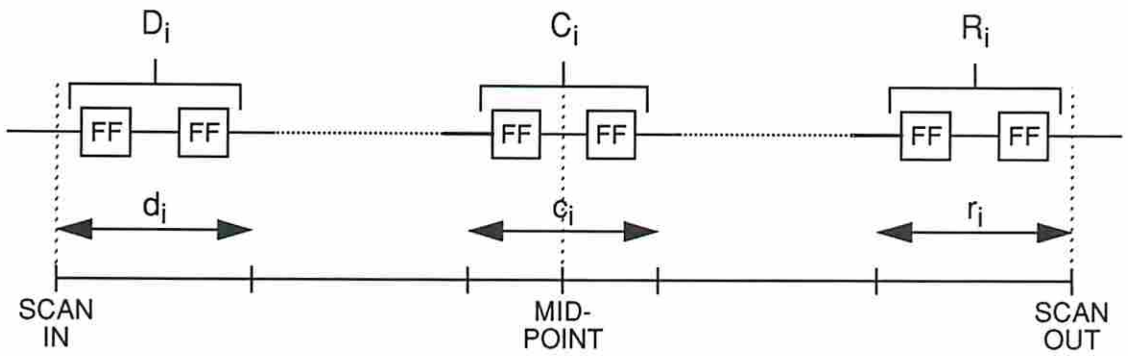
Case 2 If $d_i > \lceil (N - c_i)/2 \rceil$, then the chain must have the D_i FFs in the first d_i slots, followed by the C_i FFs in the next c_i slots. This is illustrated schematically in Figure 8.4(b). Thus the drive cycle for the current session must be $d_i + c_i$. Note that there must be a valid assignment of the R_i FFs in the remaining slots. Clearly the group of driver-receiver FFs C_i is skewed from the middle of the chain towards the scan-out pin, hence the receive cycle cannot be greater than the drive cycle. Thus the chain cycle for this configuration, which corresponds to the minimum session cycle, is $d_i + c_i$.

Case 3 If $r_i > \lceil (N - c_i)/2 \rceil$, then by an argument analogous to that in Case 2, the minimum session cycle is $r_i + c_i$. This case is illustrated in Figure 8.4(c).

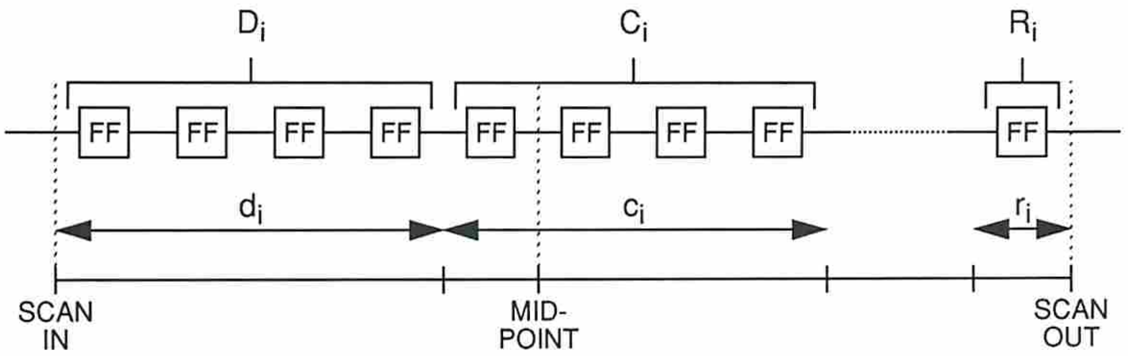
Combining the above cases, we get the following rule for the minimum session cycle for session TS_i when C_i is nonempty:

$$\text{If } C_i \neq \phi \text{ then } MSC_i = \max\left(d_i,\ r_i,\ \left\lceil \frac{N - c_i}{2} \right\rceil\right) + c_i. \tag{8.3}$$

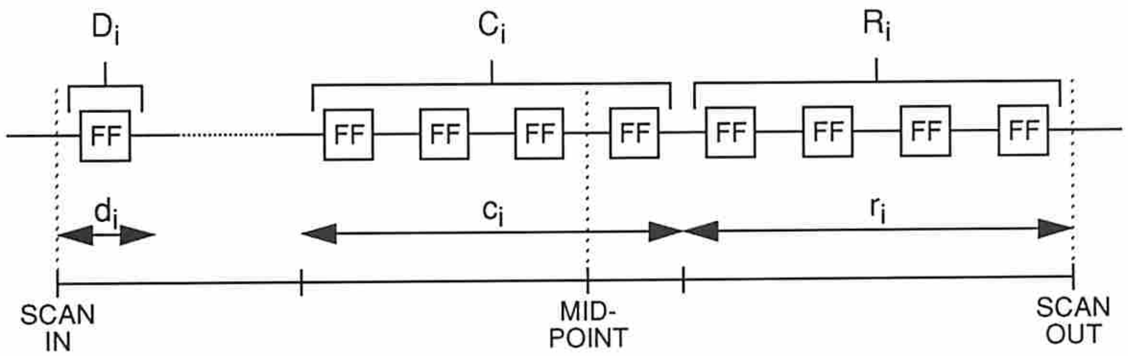
Together, equations 8.1 and 8.3 provide a complete rule for determining the minimum session cycle in all cases. These equations can be used to compute MSC_i for all sessions TS_i. Based on these computed values, equation 8.2 can be used to determine the values $Range_i(F_j)$ for all FFs F_j.



(a)



(b)



(c)

Figure 8.4: Illustration of computation of minimum session cycle. (a) Case 1, (b) case 2, (c) case 3.

Returning to the current example, we have $MSC_2 = 4$, and the corresponding ranges for the FFs for optimizing session TS_2 are shown in Figure 8.3(b). Finally, in the remaining session TS_1, we have $D_1 = \{R1\}$, $R_1 = \phi$, and $C_1 = \{R2, R3, R4, R5\}$. This results in a minimum session count $MSC_1 = 5$. Since this is equal to the length of the chain, the ranges for all FFs are $[1, 5]$ in this session, as shown in Figure 8.3(c).

Thus for each test session, not only have we shown how to determine some ordering of the scan chain that minimizes the test time in that session, but in fact we have characterized all possible orderings that achieve the minimum session time, in the form of the ranges for all FFs as shown in Figures 8.3(a,b,c). Given any particular ordering of the FFs, it is easy to check whether or not it satisfies the optimal ranges for a given session. For example, the ordering $(R1, R4, R2, R5, R3)$ satisfies the optimal ranges for TS_2 and also for TS_1, but not for TS_3. Given the ranges for the various sessions, the question arises: is there any ordering that satisfies the optimal ranges for *all* the sessions?

Figure 8.3(d) shows the result of taking the intersection, for each FF, of the ranges in Figures 8.3(a,b,c). The computation of these ranges is shown below.

$$\forall F_j \in FS, \;\; Range(F_j) = \bigcap_{1 \le i \le |KSEQ|} Range_i(F_j).$$

We will refer to $Range(F_j)$ as the **ideal range** of F_j. Note that in this example all the ideal ranges are nonempty. (It is possible that some ideal ranges may be empty; this situation will be discussed in Section 8.3.3.) From the manner in which the various ranges have been derived, it follows that if any ordering of the chain satisfies the ideal ranges, then that ordering must achieve the lowest possible test time for every session, and consequently the lowest possible test time for the entire test. This result is stated in the following theorem.

Theorem 7 *Given a position mapping $P : FS \rightarrow [1, N]$, and given $\forall F_j \in FS$, $Range(F_j) \neq \phi$, the ordering of the scan chain corresponding to P is optimal (i.e., leads to minimum test time using the overlapped scheme) if $\forall F_j \in FS$, $P(F_j) \in Range(F_j)$.* □

We will refer to any ordering of the chain that satisfies all the ideal ranges as an **ideal ordering** or an **ideal solution**. The result of Theorem 7 will be utilized in the next section to help in solving the chaining problem.

8.3.2 Single Chain Algorithm

We present in this section a heuristic procedure, called **singleChain**, that returns an ideal ordering if one exists, otherwise returns an ordering in which a maximal number of FFs lie in their ideal ranges. If an ideal ordering exists, then according to Theorem 7 the overall test time for the circuit is minimized, i.e., the ordering is optimal. If however not all FFs lie in their ideal ranges, then the ordering returned by the procedure may or may not be optimal, i.e., there may exist another ordering not found by this procedure that actually leads to a lower test time using the overlapped test scheme. In the latter case there is no obvious way to determine whether or not the solution returned is optimal. However, the procedure computes a *confidence* value, which is the fraction of the FFs that lie in their respective ideal ranges. This value indicates the degree of confidence in the optimality of the solution. A confidence value 1 indicates that the solution is definitely optimal.

The procedure **singleChain** maps the problem of finding an ideal ordering to the well-known *maximum-size bipartite matching problem* [23][Chapter 9]. Given a bipartite graph (V, U, A) where V and U are two sets of nodes and $A \subseteq V \times U$ is a set of arcs, the maximum-size bipartite matching problem is to find a maximum-cardinality set $M \subseteq A$ such that for any two arcs $a, b \in M$, a and b have no node in common. This problem can be solved in $O(\sqrt{n} \cdot e)$ time by mapping it to the maximum network flow problem [23]. If the resulting matching is a *perfect matching*, i.e., $|M| = |V| = |U|$, the corresponding ordering of the FFs is an ideal solution, and a confidence value of 1.0 is returned. If not, the match M yields a partial placement of the FFs in the chain. In this case the procedure places the remaining FFs in the vacant slots, each as close as possible to its ideal range. This is carried out in a greedy fashion by starting with FFs with empty or small ranges (i.e., more strongly constrained FFs) and proceeding to FFs with larger ranges. The confidence value

returned is $|M|/N$, where $|M|$ is the number of FFs lying within their ideal ranges and N is the total number of FFs.

procedure singleChain (Number of FFs N; set of FFs FS; ideal ranges $Range(F_j)$ for all $F_j \in FS$): Returns P, a mapping of FFs to $[1, N]$, and a confidence value c, $0 \leq c \leq 1$.

1. Construct bipartite graph $B \equiv (V, U, A)$, where V, U are two sets of nodes and A is the set of arcs, as follows.

   $V \leftarrow U \leftarrow [1, N]$;
   $A \leftarrow \{(j, l) \in V \times U \mid l \in Range(F_j)\}$.

2. Obtain $M \subseteq A$, the maximum-size bipartite match of B.

3. $SRC \leftarrow \{j \in [1, N] \mid (j, l) \in M\}$; $DST \leftarrow \{l \in [1, N] \mid (j, l) \in M\}$.

4. Determine positions for FFs that are placed within their ideal ranges:

   For each $j \in SRC$ do:

   (a) Determine (j, l), the arc in M that is connected to node j in B.
   (b) $P(F_j) \leftarrow l$.

5. If the solution is ideal, i.e., $|M| = N$, then **return** P along with $c = 1.0$.

6. Determine positions for FFs that cannot be placed within their ideal ranges:

   $FSEQ \leftarrow$ sequence of FFs $F_j \in (FS - SRC)$, ordered according to increasing size of $Range(F_j)$.

   For each $F_j \in FSEQ$ in order do:

   (a) Let p be the integer in the set of integers $([1, N] - DST)$ that is nearest to the middle of $Range(F_j)$ if $Range(F_j) \neq \phi$, or nearest to the value $(N + 1)/2$ otherwise.
   (b) $P(F_j) \leftarrow p$.
   (c) $DST \leftarrow DST \cup \{p\}$.

7. The function P is now defined for all FFs in the domain FS, and represents an ordering of the FFs such that a maximum number of FFs lie within their ideal ranges.

   Confidence level, $c \leftarrow \frac{|M|}{N}$.

   Return P along with c. □

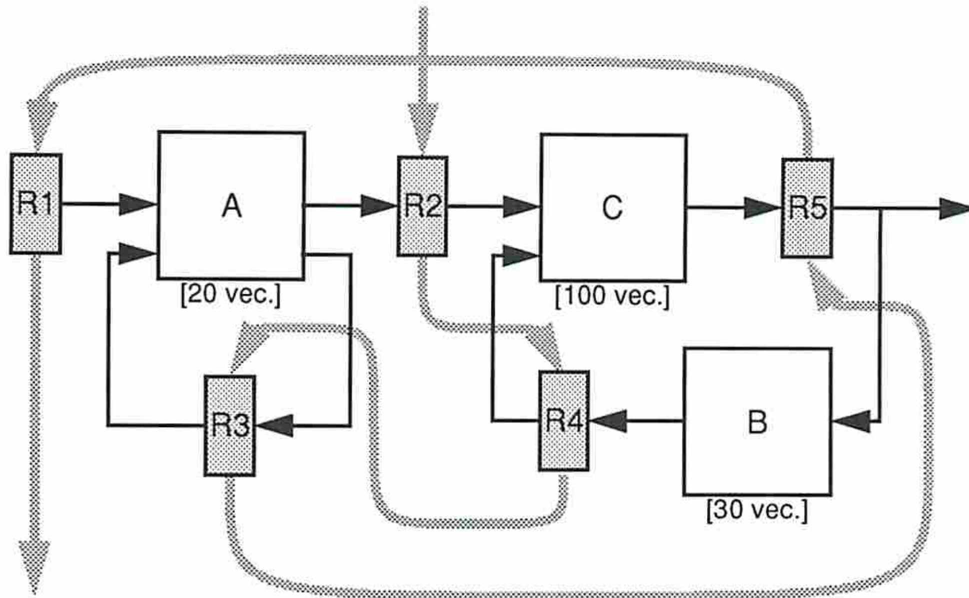


Figure 8.5: Example of single scan chain ordering.

The actual ordering of FFs returned by the algorithm depends on the implementation of the maximal-size bipartite matching algorithm. One possible solution for the circuit of Figure 8.2 is indicated in Figure 8.3(d), where the thickened portion of each range represents the slot in which the corresponding FF is placed. Note that this is a perfect matching of FFs to slots, i.e., the solution is ideal. The scan path for this ordering, $(R2, R4, R3, R5, R1)$, is shown in Figure 8.5.

8.3.3 Non-Ideal Solutions

In the above discussions we used an example for which an ideal solution exists, i.e., the procedure returns an optimal solution with confidence level 1.0. There are two possible cases in which no ideal solution can be found. (1) One or more of the ideal ranges for the FFs is/are empty; (2) the ideal ranges for the FFs are all nonempty, but no perfect matching of FFs to chain slots exists. For completeness, we present below an example of each of these two cases.

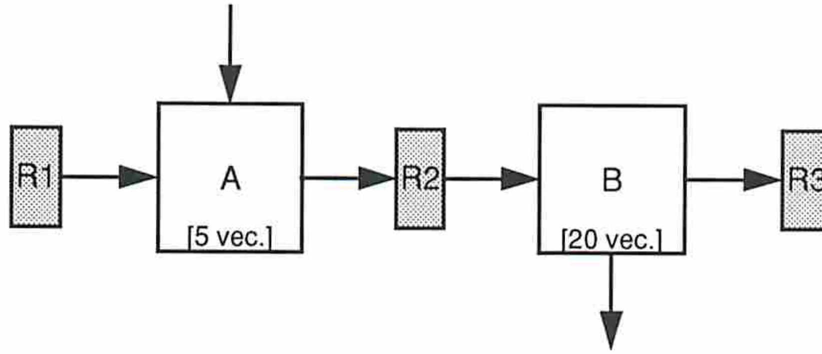


Figure 8.6: Example of single scan path chaining with empty ideal range for a register.

Example 1: Empty Range Figure 8.6 shows a circuit example with two combinational kernels, A (requiring 5 test vectors) and B (requiring 20 test vectors). Each register consists of a single FF. This circuit has $KSEQ = (A, B)$. It is tested in two sessions, TS_1 and TS_2, with minimum session cycles $MSC_1 = 2$ and $MSC_2 = 1$. For the FF $R2$ this results in $Range_1(R2) = [2, 2]$ and $Range_2(R2) = [1, 1]$. Hence the ideal range $Range(R2)$ is empty, i.e., ϕ. However, the ideal ranges for the other two FFs are nonempty; $Range(R1) = [1, 1]$ and $Range(R3) = [3, 3]$. Hence the chaining procedure first places $R1$ in slot 1 and $R3$ in slot 3. Subsequently $R2$ is placed in the remaining slot 2. Thus the ordering generated by **singleChain** is $(R1, R2, R3)$, and the confidence level is $\frac{2}{3}$.

The confidence value lower than 1.0 implies that the ordering may or may not be optimal; in our example the solution is actually nonoptimal. For the solution $(R1, R2, R3)$, the total test time using the overlapped scheme is 62 clock cycles. Now consider another ordering $(R2, R1, R3)$. The total test time in this case is 53 clock cycles, which indicates that the solution obtained by **singleChain** is not optimal.

Consider an interesting variation on the example in which A requires 15 test patterns instead of 5. Since the relative test lengths of A and B are unchanged, $KSEQ$ is still (A, B). The procedure **singleChain** returns the same non-ideal solution $(R1, R2, R3)$, for which the total test time is again 62 clock cycles. With the alternative ordering $(R2, R1, R3)$, however, the test time is now 73 clock cycles; this implies that the solution from the procedure could actually be optimal.

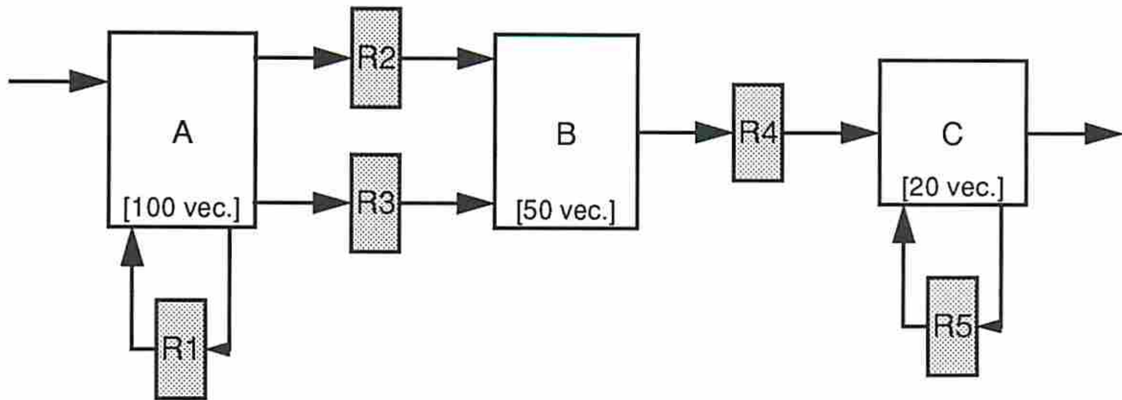


Figure 8.7: Example of single scan path chaining with no perfect matching.

This variation on the example highlights a weakness in the procedure **singleChain**, in that it does not consider the actual test lengths of the various kernels, only the relative test lengths, in computing an ordering. If an ideal ordering exists, then the actual test lengths are irrelevant since the ordering must be optimal. When no ideal ordering exists, however, the solution may suffer from the lack of consideration of the actual test lengths. The issue of taking the exact test lengths into account when constructing a non-ideal solution is an open problem requiring further analysis.

Example 2: Nonempty Ranges, No Perfect Matching Another example of single scan path chaining is shown in Figure 8.7. There are three combinational kernels with test lengths shown, and five FFs. Consider the FFs $R1$, $R2$ and $R3$. Analysis shows that $Range(R1) = [3,3]$; $Range(R2) = Range(R3) = [3,4]$. Even without considering the ranges of other FFs, it is evident that no perfect matching of the FFs to the chain slots can exist, since the three FFs $R1$, $R2$ and $R3$ cannot be squeezed into the two slots at positions 3 and 4.

In this case the procedure would first produce a partial matching with four of the five FFs in their ideal ranges, e.g., $(R5, -, R1, R2, R4)$. The remaining FF $R3$ would then be placed outside its ideal range in the remaining slot 2. The associated confidence level is 0.8.

8.3.4 A Simplifying Transformation

Above we have presented a procedure for constructing a near-optimal single scan chain for a fully compatible circuit. The procedure views the circuit as a sequence of n kernels, $KSEQ \equiv (K_1, \cdots, K_n)$, in ascending test length order. Each kernel K_i has certain scan registers driving it and certain scan registers receiving from it. Each gives rise to a test session TS_i during which kernels $K_i, \ldots, K_n$ are tested. In certain circuits, however, a subset of the kernels in $KSEQ$, say $(K_i, \ldots, K_j)$, may have exactly the same test length; hence the test sessions $TS_{i+1}, \ldots, TS_j$ have zero duration under the overlapped scheme.

In this case the problem can be simplified by merging $K_i, \ldots, K_j$ into a single kernel, say $K_{i \ldots j}$. Any register that drives (receives from) a kernel in $(K_i, \ldots, K_j)$ is also a driver (receiver) for the merged kernel. This merging of kernels can be carried out for every subsequence within $KSEQ$ of kernels having the same test length, resulting in a shorter sequence of kernels. However, the ideal range of each FF is exactly the same for the modified problem. Thus the result of the chaining algorithm for the modified set of kernels is also a solution to the original chaining problem. This transformation can be used to reduce the computation complexity whenever possible without altering the solution space.

8.3.5 Case Study

The single scan path chaining analysis of this section was applied to a single bit slice (i.e., a single butterfly circuit) of the Viterbi decoder described in Section 3.9. Based on full scan design, this circuit had 12 mutually compatible combinational kernels of various sizes, each being essentially a *cloud* of the circuit. These kernels were connected together via a total of 28 scan FFs. Due to similarities among some groups of kernels, there were only four distinct test lengths among the 12 kernels. Hence after the simplifying transformation of merging equal-test-length kernels the number of kernels for the chaining analysis was reduced to four.

These kernels and their interconnections are illustrated schematically in Figure 8.8(a). The four merged kernels and their respective test lengths are A [2], B

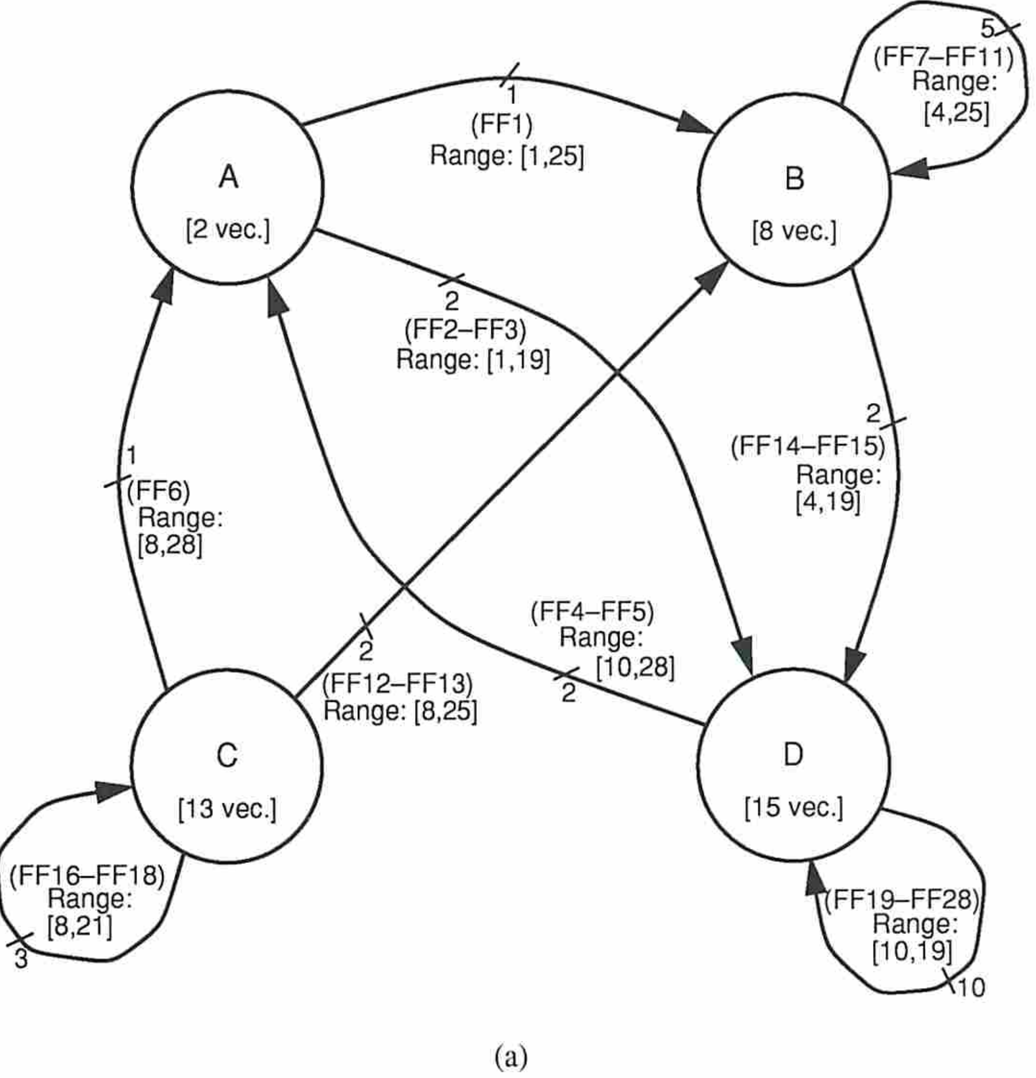
[8], C [13], and D [15]. Each arrow in the diagram represents a set of scan FFs that receive test results from the source kernel and drive test patterns into the destination kernel. Thus for example the self-loop at node B indicates that there are five scan FFs that serve as driver-receivers for kernel B. The parentheses alongside an arrow contain the names of the FFs, e.g., ($FF7$–$FF11$). There are a total of 28 scan FFs to be connected in a single scan chain.

In this example, if the combined test scheme is used with an arbitrary scan chain ordering, a chain cycle of 28 must be used for applying all 15 tests to the circuit. The resulting test time is **463** clock cycles. Below we determine a scan chain ordering and the resulting test time based on the overlapped test scheme.

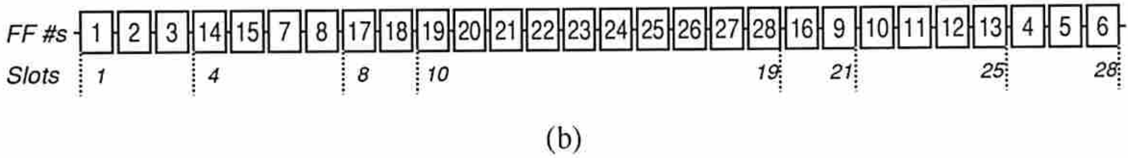
For the four test sessions, $TS_1 \ldots TS_4$, the respective minimum session cycles are 28, 25, 21 and 19. Using these values the ideal ranges can be computed for all the 28 FFs. In every session there is at least one driver-receiver FF, hence equation 8.3 must be used. Note that FFs that lie along the same arrow in Figure 8.8(a), i.e., that receive from the same kernel and drive the same kernel, must have exactly the same range. The range for a group of FFs covered by each arrow in the diagram is indicated alongside it, e.g., $[4, 25]$ for the self-loop at kernel B. Since there are only nine arrows in the diagram, the range computation needs to be carried out only nine times.

For this circuit the procedure **singleChain** is able to find an ideal chain ordering, i.e., an assignment of FFs to chain slots such that every FF is within its ideal range. The actual ordering returned by the procedure depends on which of the many possible perfect matchings is found during the execution of the maximum matching step. One such chain ordering is shown in Figure 8.8(b). Based on the overlapped test scheme, the test time using this scan chain is **392** clock cycles. According to Theorem 7 this must be an optimal solution. Note that this solution achieves a saving of 15% in the test time compared to the combined test scheme, at the cost of a more restricted scan path ordering.

Thus in general, the overlapped scheme trades off routing area against test time compared to the combined scheme. The actual increase in routing area, if any, cannot be determined until the final physical design is carried out. If in fact the



(a)



(b)

Figure 8.8: Case study for single scan chain. (a) Schematic description of circuit, (b) optimal scan chain ordering.

wiring due to the optimal scan chain ordering is taken into account in the placement stage, the actual area cost may not be very high. Even better, if the ordering generation in **singleChain** could be made sensitive to the physical placement of the scan FFs, say by altering the process of obtaining a maximum matching, the additional routing cost could be minimized or even eliminated in some cases. The latter idea is a subject for future study.

8.4 Multiple Chains for Two Compatible Kernels

In this section we consider another restricted case of the general chaining problem defined in Section 8.1. Unlike in the previous problem, we now allow multiple scan chains. Note that this introduces two new degrees of freedom in the solution: the lengths of the various chains, and the actual assignment of FFs to the chains. To compensate for the vastly increased complexity of the solution space, in this section we will consider the following restrictions on the problem, and attempt to formulate a solution.

1. There are exactly two kernels.
2. As in the previous study, the kernels are mutually compatible.
3. The number of chains is prespecified.
4. Rather than the canonical overlapped test application scheme, we use a special *quasi-overlapped scheme* described below in which the ordering of FFs in the scan chains is irrelevant.

The **quasi-overlapped** scheme used in this problem is actually a hybrid of the combined and overlapped schemes. If the circuit has a single scan chain, this scheme is identical to the combined scheme, i.e., the chain is flushed completely every time a test pattern is applied, irrespective of the ordering of the scan FFs. If there are multiple scan chains, however, a modified form of the overlapped scheme is used. Let $KSEQ \equiv (K_1, K_2, \ldots, K_n)$ be the ordered sequence of kernels and

$TS_1, TS_2, \ldots, TS_n$ the test sessions exactly as defined under the overlapped scheme in Section 8.2.3. In the original overlapped scheme, a minimum amount of shifting is done to apply each test pattern, and the chain cycle used in session TS_i is $CC(K_i, \ldots, K_n)$, which is dependent on the FF ordering within the scan chains. In the quasi-overlapped scheme, however, all chains that are active in a given session are flushed completely. Thus the chain cycle used in session TS_i is

$$\max_{c \in Ch(K_i, \ldots, K_n)} |c| ,$$

where $Ch(K_i, \ldots, K_n)$ is the set of chains used for applying tests during session TS_i and $|c|$ is the number of FFs in chain c. Thus in this scheme the chain cycles in the various sessions do not depend on the ordering of FFs in the scan chains, only on the *assignment* of FFs to the scan chains.

8.4.1 Modeling the Problem

Under the above restrictions, the problem of this section can be stated as follows: given two kernels and the associated scan FFs, and given the number of scan chains to be constructed, determine an assignment of the scan FFs to chains such that the overall test time under the quasi-overlapped test scheme is minimized.

Figure 8.9 shows the two kernels under test, K_1 and K_2, and the associated scan FFs. There are three set of scan FFs. The first, R_1, contains n_1 FFs that are used for testing K_1 but not K_2. These FFs may be either drivers, receivers, or driver-receivers with respect to K_1. It is not important to distinguish among these types of FFs when the quasi-overlapped scheme is used, since the presence of any of these FFs anywhere in a given chain may require the whole chain to be flushed during test, irrespective of the FF type. Similar to R_1, the set of FFs R_2 contains n_2 FFs that are used for testing K_2 but not K_1. Finally, R_{12} contains n_{12} FFs that are to be used for testing *both* K_1 and K_2. Thus each FF in R_{12} *either* receives from K_1 and drives K_2 *or* drives K_1 and receives from K_2.

Let k be the number of chains to be constructed. Each chain c_i is essentially a set of FFs, i.e., a subset of $R_1 \cup R_2 \cup R_{12}$. The set of k chains in the design

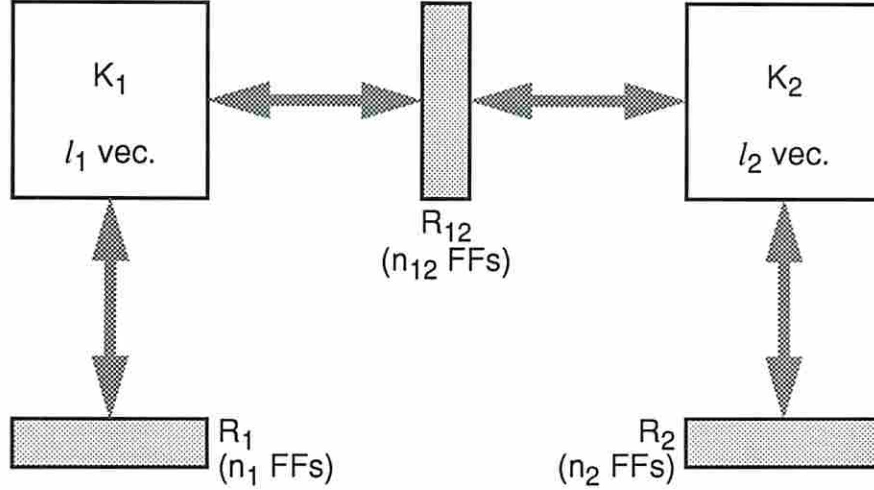


Figure 8.9: Circuit model for two-kernel multiple scan chain study.

under consideration is denoted by Ch. Our objective is to determine the set $Ch = \{c_1, c_2, \ldots, c_k\}$, where

$$\begin{aligned} \cup_{c_i \in Ch} c_i &= R_1 \cup R_2 \cup R_{12}; \\ \forall c_i, c_j \in Ch,\ i \neq j,\ & c_i \cap c_j = \phi; \end{aligned}$$

and the overall test time is minimized.

In the discussions below, we will use Ch_1, Ch_2 and Ch_{12} to denote three partitions of Ch defined as follows.

$$\begin{aligned} Ch_1 &= \{c \in Ch \mid c \subseteq R_1\}; \\ Ch_2 &= \{c \in Ch \mid c \subseteq R_2 \cup R_{12}\}; \\ Ch_{12} &= Ch - Ch_1 - Ch_2 . \end{aligned}$$

Thus each chain in Ch_{12} contains at least one FF from each of R_1 and $(R_{12} \cup R_2)$.

Under the quasi-overlapped test scheme described above, the test for the circuit consists of two sessions, TS_1 and TS_2. For brevity, let l_i denote the test length $T(K_i)$ of K_i; i.e., $l_1 \equiv T(K_1)$ and $l_2 \equiv T(K_2)$. Note that by assumption, $l_1 \leq l_2$. In TS_1, l_1 test patterns are applied to K_1 as well as K_2 in a combined

manner. During this session all the chains in Ch must be involved in applying tests. Thus the chain cycle for this session is $\max_{c \in Ch} |c|$. In TS_2, the remaining $l_2 - l_1$ test patterns are applied to K_2 alone. Clearly during this session all chains in Ch_1 must be idle since they contain no FFs connected to K_2. Only chains in Ch_2 and in Ch_{12} are involved in applying the tests, since each contains at least one FF connected to K_2. Accordingly, the chain cycle in this session is $\max_{c \in Ch_2 \cup Ch_{12}} |c|$.

Below we derive some interesting properties of any optimal chaining solution, i.e., any well-formed set of chains $Ch \equiv \{c_1, c_2, \ldots, c_k\}$ that has minimum associated test time using the quasi-overlapped scheme. We take advantage of these properties to prune the search space and efficiently obtain an optimal solution.

8.4.2 Problem Characteristics

In constructing the k scan chains for the circuit of Figure 8.9, each of the $n_1+n_2+n_{12}$ FFs could be assigned to any of k chains. Hence the total number of possible ways of constructing the chains is $k^{(n_1+n_2+n_{12})}$. For small problem sizes it may be possible to enumerate all the ways and compute the test time for each, since the test time computation for a fully compatible circuit using is relatively simple under any test application scheme. For larger problems, however, the solution space grows rapidly. For example, with 4 chains and 30 FFs, the total number of distinct chain configurations is approximately 10^{18}. Fortunately, as the study presented in this section demonstrates, it is possible to identify a limited subspace of solutions within which an optimal solution must exist. This helps to restrict the search for the solution.

We first show that we can ignore chaining configurations in which there is more than one chain in Ch_{12}, i.e., there is more than one chain that has FFs from R_1 as well as from $R_2 \cup R_{12}$. For example, consider the configuration in Figure 8.10(a), where each vertical bar represents a chain and its length represents the number of FFs in the chain. Ch_{12} contains two chains, c_3 and c_4. Assume that this configuration is optimal. The proof of the lemma below will show that another optimal configuration, shown in Figure 8.10(b), must exist. In the new optimal configuration, the FFs in

c_3 and c_4 are redistributed, without altering their respective lengths, such that the resulting set Ch_{12} has only one chain.

Lemma 10 *There exists an optimal chain configuration in which Ch_{12} has only one chain.*

Proof Suppose Ch is an optimal configuration of the chains with $|Ch_{12}| > 1$. Then transform the chain configuration Ch as follows.

Pick any two chains $c_i, c_j \in Ch_{12}$. Without loss of generality, assume that $|c_i| \leq |c_j|$, i.e., c_i is not longer than c_j. Define $N_1 = |(c_i \cup c_j) \cap R_1|$, i.e., the total number of FFs from R_1 present in the two chains. Define $N_2 = |c_i| + |c_j| - N_1$, i.e., the total number of FFs from R_2 and R_{12} present in the two chains. Note that since c_i is the shorter chain, $N_1 + N_2 = |c_i| + |c_j| \geq 2|c_i|$. Thus clearly $N_1 \geq |c_i|$ or $N_2 \geq |c_i|$. In either case, we reconstruct the two chains while keeping their respective lengths unchanged as described below.

Case 1: If $N_1 \geq |c_i|$, place $|c_i|$ FFs from R_1 in c_i, and the remaining $N_1 + N_2 - |c_i|$ FFs (from both R_1 and $(R_2 \cup R_{12})$) in c_j. c_i now belongs to Ch_1 instead of Ch_{12}.

Case 2: If $N_2 \geq |c_i|$, place $|c_i|$ FFs from $(R_2 \cup R_{12})$ in c_i, and the remaining $N_1 + N_2 - |c_i|$ FFs (from both R_1 and $(R_2 \cup R_{12})$) in c_j. c_i now belongs to Ch_2 instead of Ch_{12}.

In both cases, the chain configuration is altered such that one chain is removed from Ch_{12} and migrates to either Ch_1 or Ch_2. Under the quasi-overlapped test application scheme, all the chains in Ch are accessed in test session TS_1; since the length of each chain is unaltered, the test time in TS_1 is unchanged. In test session TS_2, only chains in $Ch_{12} \cup Ch_2$ are accessed. In the two cases above, this set of chains either remains the same or has one chain removed. Thus in both cases the chain cycle in TS_2 cannot increase, and the test time for TS_2 cannot increase. Thus we have shown that by transforming the chains as described above, the number of chains in Ch_{12} is reduced by one but the test time does not increase. Hence the new chain configuration resulting from this transformation must also be optimal.

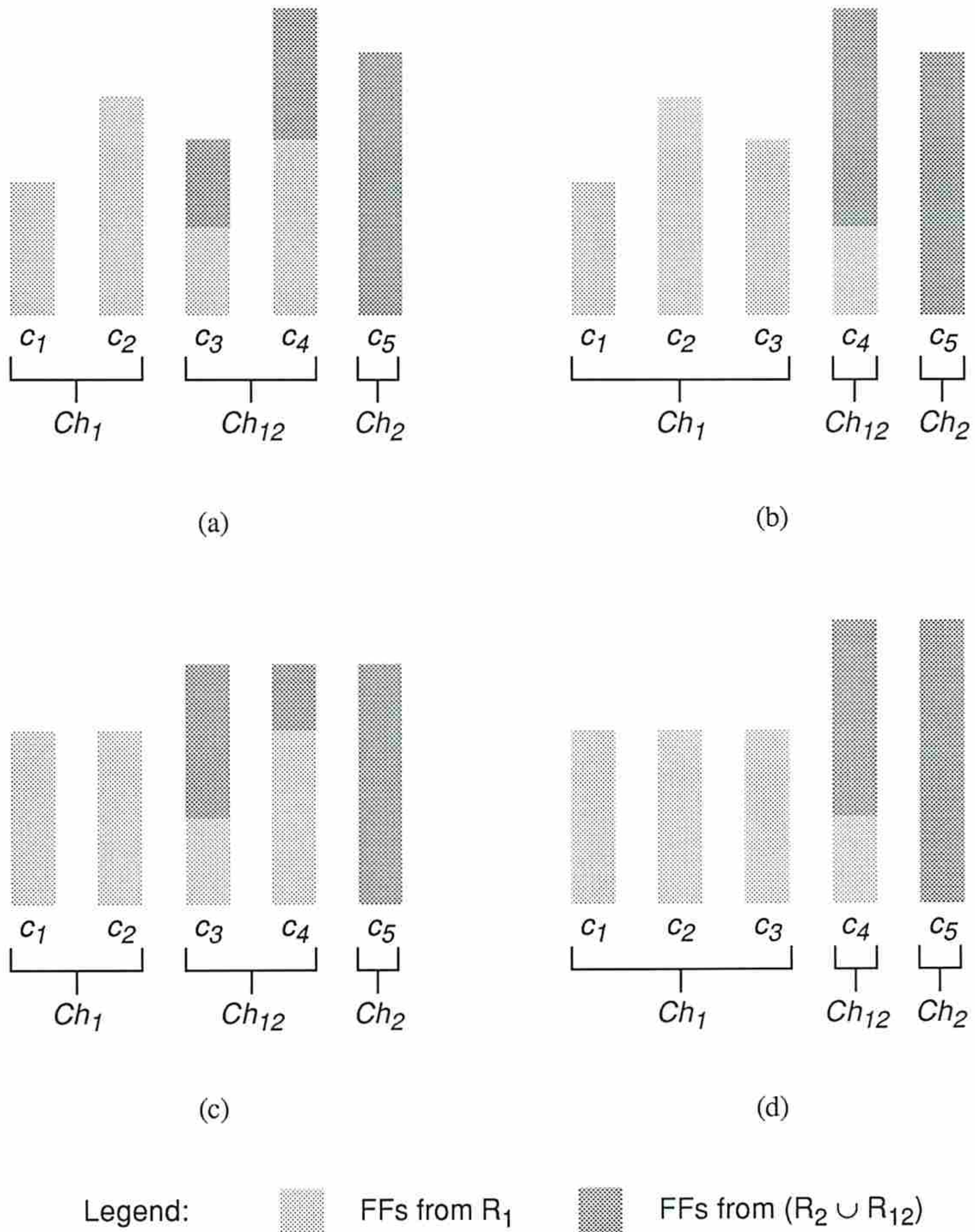


Figure 8.10: Optimal chaining solutions. (a) Original configuration, (b) in region A, (c) in region B, (d) in intersection region.

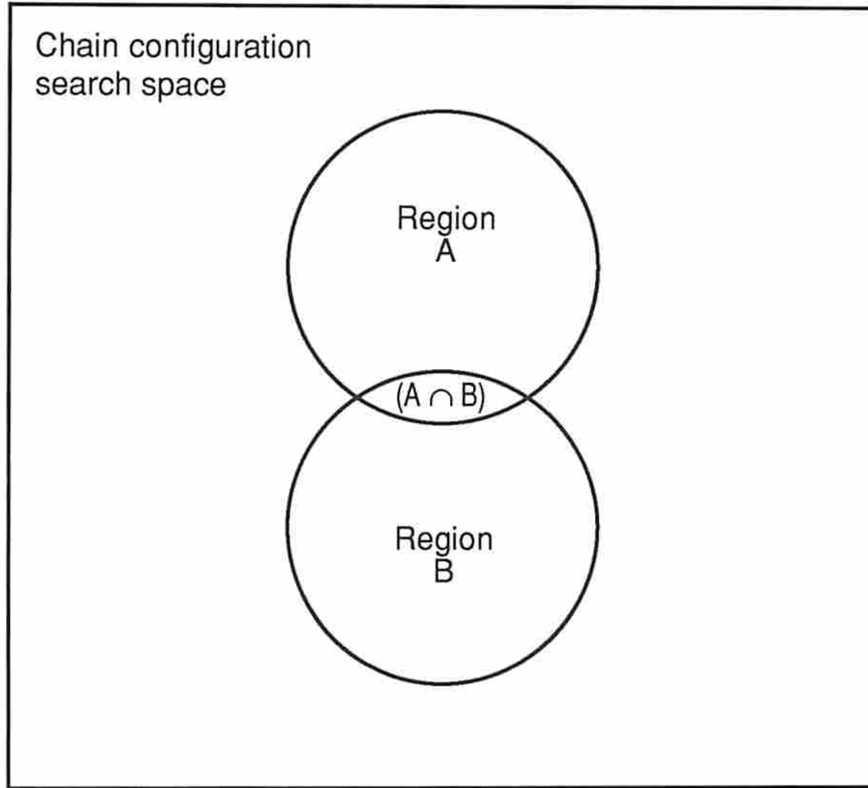


Figure 8.11: Venn diagram of search space for multiple chain design problem.

The transformation described above can be repeated as long as Ch_{12} has more than one chain. Each time a new optimal configuration is generated in which the number of chains in Ch_{12} decreases by at least one. Finally we must obtain a chain configuration with $|Ch_{12}| = 1$. □

Lemma 10 shows that in searching for an optimal chain configuration, it is sufficient to consider those configurations in which at most one chain has FFs from both R_1 and $(R_2 \cup R_{12})$, and for all other chains c, either $c \subseteq R_1$ or $c \subseteq (R_2 \cup R_{12})$. Figure 8.11 shows a Venn diagram of the search space, in which Region A is the restricted search space resulting from Lemma 10.

There are still many degrees of freedom in the configuration of the chains, both within Region A and outside it, because each chain is allowed to have an arbitrary number of FFs. Below we show another way in which the search space can be limited by considering only chaining configurations in which all chains in Ch_1 are

of equal length (± 1), and all chains in $Ch_2 \cup Ch_{12}$ are of equal length (± 1). For the optimal configuration in Figure 8.10(a), the proof of the lemma below will show that there must exist another optimal solution, shown in Figure 8.10(b), satisfying this restriction.

Lemma 11 *There exists an optimal chain configuration in which all chains in Ch_1 are of equal length (± 1) and all chains in $Ch_2 \cup Ch_{12}$ are of equal length (± 1).*

Proof Suppose Ch is an optimal configuration of the chains. Carry out the following transformations which preserve the optimality of the configuration.

First, consider the chains in Ch_1 and rearrange them such that all the FFs contained among them are evenly distributed among them. Formally, let $N_1 = \sum_{c \in Ch_1} |c|$; reconstruct the $|Ch_1|$ chains in Ch_1 such that each has either $\left\lceil \frac{N_1}{|Ch_1|} \right\rceil$ or $\left\lfloor \frac{N_1}{|Ch_1|} \right\rfloor$ FFs. Clearly none of the resulting chains can be longer than the longest of the original chains in Ch_1.

Next, consider the chains in $(Ch_2 \cup Ch_{12})$ and rearrange them such that all the FFs contained among them are evenly distributed among them. Let $N_2 = \sum_{c \in (Ch_2 \cup Ch_{12})} |c|$; reconstruct the $|Ch_2 \cup Ch_{12}|$ chains such that each has either $\left\lceil \frac{N_2}{|Ch_2 \cup Ch_{12}|} \right\rceil$ or $\left\lfloor \frac{N_2}{|Ch_2 \cup Ch_{12}|} \right\rfloor$ FFs. Clearly none of the resulting chains can be longer than the longest of the original chains in $Ch_1 \cup Ch_{12}$.

With the resulting chain configuration, the chain cycles to be used in both sessions TS_1 and TS_2 cannot be greater than the respective chain cycles with the original configuration. Thus the new configuration must also be optimal, and satisfies the statement of the lemma. □

Thus Lemma 11 can be used to restrict the search to a space labeled Region B in Figure 8.11. The two lemmas presented above have shown that it is sufficient to search for a solution in either Region A or Region B. We will now show that it is actually sufficient to search in only the intersection region of these spaces, by proving that an optimal configuration must exist within the intersection region. Figure 8.10(d) shows such an optimal configuration, which is derived from the optimal configuration in Figure 8.10(b).

Theorem 8 *There exists an optimal chain configuration in which Ch_{12} has at most one chain and all chains in Ch_1 are of equal length (± 1) and all chains in $Ch_2 \cup Ch_{12}$ are of equal length (± 1).*

Proof By Lemma 10 there exists some optimal chain configuration Ch in which Ch_{12} has at most one chain. Carry out the following transformations on this configuration.

First, using a method similar to that in the proof of Lemma 11, rearrange the FFs contained in the chains in Ch_1 so that they are evenly distributed among the chains. This is similar to the first transformation step in that proof. The lengths of the resulting chains in Ch_1 must range from $x_1 - 1$ to x_1, where x_1 is some integer (i.e., $x_1 = \max_{c \in Ch_1} |c|$).

Similarly, using the same method, rearrange the FFs contained in the chains in Ch_2 so that they are evenly distributed among the chains. Note that this differs from the second step in the earlier proof since the chain in Ch_{12} (if any) is not altered here. The lengths of the resulting chains in Ch_2 must range from $x_2 - 1$ to x_2, where x_2 is some integer (i.e., $x_2 = \max_{c \in Ch_2} |c|$).

If there is a chain in Ch_{12}, say c_{12}, let its length be x_{12}. If x_{12} equals either x_2 or $x_2 - 1$, or if there is no chain in Ch_{12}, then the present configuration satisfies the statement of the theorem. Otherwise consider the following two cases.

Case 1: If c_{12} is the longest of the chains in this configuration, i.e., $x_{12} > x_1$ and $x_{12} > x_2$, then c_{12} is the chain that determines the chain cycle in both sessions TS_1 and TS_2. Pick any FF in c_{12}, say f, and remove it from c_{12}. Depending on whether $f \in R1$ or $f \in (R_2 \cup R_{12})$, add f either to one of the chains in Ch_1 or Ch_2, respectively. Note that it must be possible to do so while maintaining all chains in Ch_1 of equal length (± 1) and all chains in Ch_2 of equal length (± 1). In the new configuration created by this migration of f, the length of c_{12} is closer to x_2 by at least one FF; however, c_{12} is still the longest chain, and since it has been shortened, the test time in each of the two test sessions could not have increased. Hence the new configuration is also an optimal configuration. The process described in this paragraph can be repeated, updating the values of x_1, x_2 and generating new optimal

configurations, until the length of c_{12} is equal to x_2 (± 1). This final configuration must satisfy the statement of the theorem.

Case 2: If c_{12} is not the longest chain in this configuration, then remove arbitrarily one FF f from a longest chain (which could be in either Ch_1 or Ch_2), and place f in c_{12}. Note that irrespective of which chain f came from, the test time in each of the two sessions could not have increased. This migration process can be repeated if necessary until the length of c_{12} is equal to x_2 (± 1). This final configuration must satisfy the statement of the theorem.

Thus the theorem is proved in all possible cases. $\square$

The preceding theorem allows us to restrict the search for an optimal configuration to a small fraction of the search space, i.e., the intersection of Regions A and B in Figure 8.11. In the following section we develop an efficient way to traverse this space.

8.4.3 Constructing Optimal Chains

Due to the restricted characteristics of any configuration in the intersection of Regions A and B, we can define a simplified search problem as shown below. We first formulate a nonlinear integer programming problem and then show how to tackle the problem by reducing it to a bounded set of linear problems.

8.4.3.1 Nonlinear Problem Formulation

The problem can be stated as follows. Given the test lengths l_1 and l_2 of the two kernels, the lengths n_1, n_2 and n_{12} of the three registers, and the total number of chains k that are to be constructed, determine four integers k_1, k_2, x_1 and x_2 such that the following relationships hold.

1. There are k_1 chains in Ch_1 and k_2 chains in $(Ch_2 \cup Ch_{12})$. That is,

$$0 \leq k_1, k_2 \leq k; \tag{8.4}$$

$$k_1 + k_2 = k. \tag{8.5}$$

In the configuration of Figure 8.10(d), $k_1 = 3$ and $k_2 = 2$.

2. Every chain in Ch_1 has either x_1 or $x_1 - 1$ FFs from R_1. At least one of these chains must have x_1 FFs. If there is a chain in Ch_{12} (with maximum length x_2), it may have up to $x_2 - 1$ FFs from R_1. These observations are embedded in the following constraints on k_1, x_1 and x_2.

$$k_1(x_1 - 1) + 1 \ \leq \ n_1 \ \leq \ k_1 x_1 + (x_2 - 1). \tag{8.6}$$

3. Every chain in Ch_2 and Ch_{12} has either x_2 or $x_2 - 1$ FFs. At least one of these chains must have x_2 FFs. If there is a chain in Ch_{12}, at least one of its FFs and up to $x_2 - 1$ of its FFs must be from $R_2 \cup R_{12}$. These observations are embedded in the following constraints on k_2 and x_2.

$$(k_2 - 1)(x_2 - 1) + 2 \ \leq \ n_2 + n_{12} \ \leq \ k_2 x_2 - 1. \tag{8.7}$$

4. The test time under the quasi-overlapped scheme is minimized. Depending on the relationship between x_1 and x_2, there are two possible expressions for the overall test time, which is the objective function.

   Case $x_1 \leq x_2$: The chain cycle in both sessions TS_1 and TS_2 is x_2, hence the overall test time is

$$TT \equiv (x_2 + d + 1) l_2. \tag{8.8}$$

   Case $x_1 > x_2$: In TS_1 the chain cycle is x_1 and in TS_2 the chain cycle is x_2. The overall test time is

$$TT \equiv (x_1 + d + 1) l_1 + (x_2 + d + 1)(l_2 - l_1). \tag{8.9}$$

The two expressions above serve as objective functions to be minimized. Since the relationship between x_1 and x_2 cannot be predicted in advance, both objective functions must be attempted separately, and in each resulting solution, the values of x_1 and x_2 must be compared to determine whether they are consistent with the objective function that was used. Note that according to the lemmas proved earlier there must exist some configuration of the chains

that satisfies our constraints, hence at least one of the two cases must yield a satisfactory solution.

In the above formulation of the problem, some of the inequalities contain products of more than one of the variables to be optimized, i.e., k_1, k_2, x_1, x_2. Since all the variables take on integal values, this is an integer nonlinear programming problem.

8.4.3.2 Linearizing the Problem

In our optimization problem, the range of values that k_1 and k_2 can take on is fairly restricted, since the total number of chains k is fixed and typically small. Note also that if k_1 and k_2 are treated as constants, the problem formulation above becomes an integer linear programming (ILP) problem [29], which is theoretically intractable but for which efficient software packages exist. We can take advantage of this fact to iterative over all k possible combinations of k_1 and k_2 values, and for each, determine locally optimal values of x_1 and x_2 by solving the corresponding ILP problem. The solution with the lowest overall test time is the global optimum. The procedure **multipleChains** listed below carries out this process.

procedure multipleChains $(l_1, l_2, n_1, n_2, n_{12}, k)$: Returns k_1, k_2, x_1, x_2.

1. Current "optimal" test time: $TT^O \leftarrow \infty$.
2. For all k_1, $0 \leq k_1 \leq k$, do the following:
   (a) $k_2 \leftarrow k - k_1$.
   (b) Treating k_1 and k_2 as constants, and x_1 and x_2 as variables, solve the integer linear programming problem consisting of the constraints 8.4 through 8.7 with the objective function 8.8 above, to obtain values of x_1 and x_2 that minimize this objective function.
   If $x_1 \leq x_2$, this is a valid local optimal solution for the current values of k_1 and k_2.
   Otherwise, solve the integer linear programming problem again, this time using the objective function 8.9 instead of 8.8, to obtain new values of x_1 and x_2 that minimize this objective function.

If $x_1 > x_2$, this is a valid local optimal solution for the current values of k_1 and k_2.

Otherwise no valid solution exists for the current values of k_1 and k_2; hence skip step 2c.

(c) The values k_1, k_2, x_1, x_2 represent a local optimal solution, with test time TT according to the appropriate equation 8.8 or 8.9.

If $TT < TT^O$, then store the current solution:

i. $TT^O \leftarrow TT$;

ii. $k_1^O \leftarrow k_1$;

iii. $k_2^O \leftarrow k_2$;

iv. $x_1^O \leftarrow x_1$;

v. $x_2^O \leftarrow x_2$.

3. **Return** $k_1^O, k_2^O, x_1^O, x_2^O$. □

8.4.4 Discussion

The algorithm presented above yields an optimal chaining configuration for fully compatible two-kernel circuits. It also applies to circuits with more than two kernels for which the simplifying transformation described in Section 8.3.4 results in two kernels.

If the circuit model resulting from this transformation still has more than two kernels, however, the algorithm **multipleChains** may still be useful for obtaining a *suboptimal* solution. To make this possible, a process similar to the simplifying transformation of Section 8.3.4 could be carried out to reduce the number of kernels to two. Rather than merging kernels that have exactly the same test length, any subset of kernels that have *approximately* the same length could be merged into a single kernel. This idea could be used to partition the set of kernels into two subsets, based on their test lengths and/or the number of FFs involved in testing them, and merge the kernels in each subset. Clearly the resulting chain configuration for the two merged kernels may not be optimal for the original set of kernels. The heuristic used for partitioning the set of kernels into two subsets would have an influence on the degree of optimality of the solution.

The number of chains to be constructed is required as an input parameter to the **multipleChains** algorithm. Every chain is associated with at least two I/O pins serving as the scan-in and scan-out pins respectively. When the quasi-overlapped test scheme is used, however, note that some of the chains may be required to operate asynchronously with respect to each other, i.e., at different chain cycles, during the test. Hence one or more additional I/O pins may be required for test control under this scheme.

8.4.5 Experimental Results

In this section we study the results of applying the above multiple chain design algorithm to circuit examples, and compare the results with the traditional approach of constructing equal-length chains.

Consider a circuit with two compatible kernels, K_1 and K_2, with test lengths $l_1 = 20$ and $l_2 = 80$, respectively. Let the number of FFs communicating with K_1 but not K_2 (i.e., n_1) be 15, and let the number of FFs communicating either with K_2 alone or with both K_1 and K_2 (i.e., $n_2 + n_{12}$) be 10. The 25 FFs in this circuit could be connected into a single scan chain, in which case the test time (based on the combined test application scheme) is 2105 clock cycles.

Table 8.1 (illustrated graphically in Figure 8.12) shows the results of using multiple chains. The notation used in the table is as follows. k is the number of scan chains to be constructed. k_1 and k_2 represent the numbers of chains in Ch_1 and $(Ch_2 \cup Ch_{12})$, respectively. max_1 $[max_2]$ represents the maximum length among the chains in Ch_1 $[(Ch_2 \cup Ch_2)]$. TT is the overall test time for this solution. TT_{eq} is the overall test time for the "traditional" chaining approach in which FFs are assigned arbitrarily to chains provided the lengths of the chains are equal (± 1).

For the case of two chains, the optimal solution shown in the first row of Table 8.1 has lengths of 15 and 10, respectively, for the two chains. This results in a saving in test time, compared to the "traditional" solution, of 12.2%. For other numbers of chains, ranging up to 10, the corresponding results are shown in the table, and the overall test time is plotted in Figure 8.12. For this circuit example,

k	k_1	k_2	max_1	max_2	TT	TT_{eq}	Saving
2	1	1	15	10	995	1133	12.2%
3	1	2	15	5	695	809	14.1%
4	2	2	8	5	548	647	15.3%
5	2	3	7	4	467	485	3.7%
6	2	4	7	3	407	485	16.1%
7	3	4	5	3	365	404	9.7%
8	3	5	5	2	305	404	24.5%
9	4	5	4	2	284	323	12.1%
10	5	5	3	2	263	323	18.6%

Table 8.1: Results of multiple scan path chaining for circuit with $l_1 = 20$, $l_2 = 80$, $n_1 = 15$, and $(n_2 + n_{12}) = 10$.

the saving in test time compared to the solution with equal-length chains ranges between 3.7% and 24.5%, depending on the number of chains. The average saving among the solutions for 2, 3, ... 10 chains is 16.0%.

Table 8.2 shows the average saving in test time for variations of the above example. It should be noted that in any individual solution for a given number of chains, there must be a non-negative saving. It is possible, however, for the saving to be 0 in some cases. Clearly the average saving varies widely depending on the circuit characteristics. In general, it appears from the results of the study that when the test lengths l_1 and l_2 of the two kernels differ widely, a greater saving is obtained—unless if the ratio $l_1 : l_2$ is similar to the ratio $n_1 : (n_2 + n_{12})$, in which case the saving is less significant. This information could be useful as feedback to the process of partitioning for test. To summarize, the results show that the algorithm **multipleChains** can indeed be used to reduce the overall test time; but they also indicate that the savings are dependent on the circuit characteristics, and more study is required to understand the dependency better.

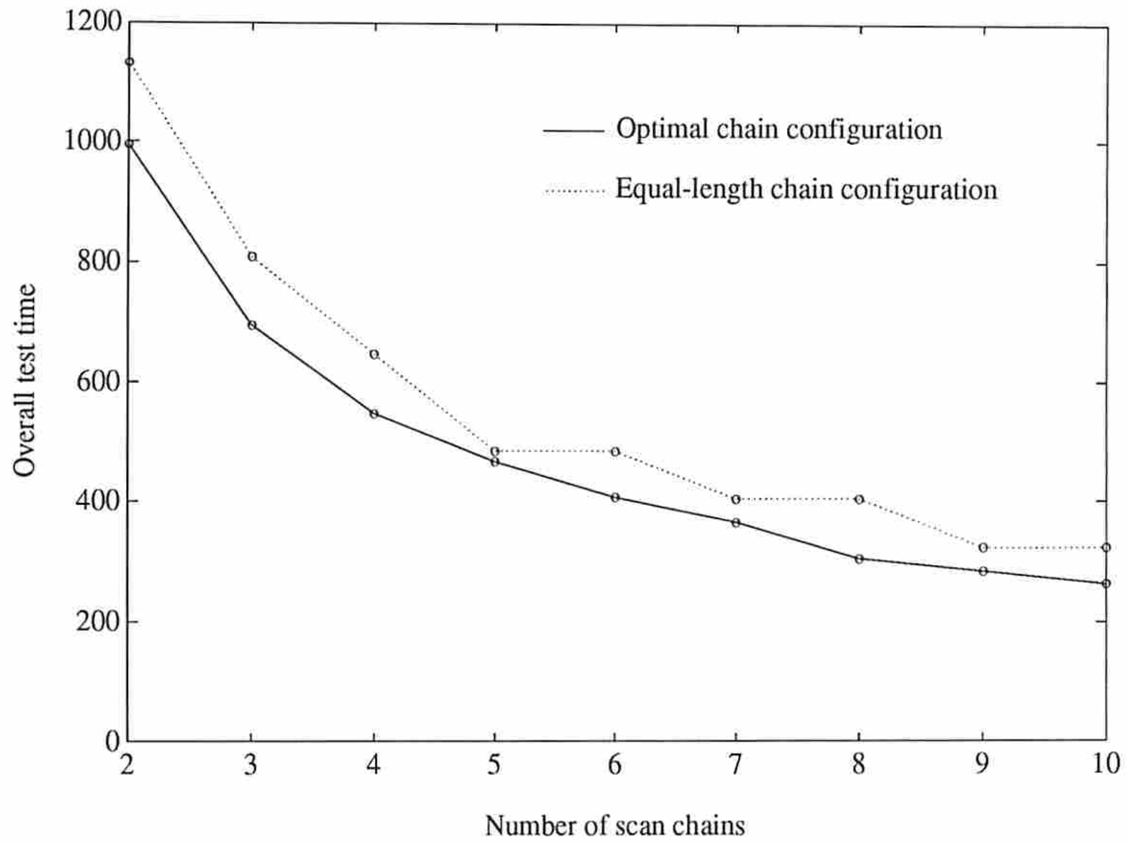


Figure 8.12: Decrease in overall test time with multiple scan chains.

l_1	l_2	n_1	$(n_2 + n_{12})$	Average saving
20	80	15	10	16.0%
20	80	10	15	6.6%
40	60	15	10	3.3%
40	60	10	15	2.2%
10	90	15	10	23.0%

Table 8.2: Average saving in test time for different circuit examples over various numbers of scan chains.

8.5 The Rest of the Iceberg

The study presented in this chapter is merely the tip of the iceberg of chaining subproblems. To get a handle on the chaining problem, we have started by looking only at fully compatible designs. The results obtained here are applicable to traditional full scan designs in which every cloud of combinational logic forms a kernel, or to partial scan designs in which size-based partitioning is used to form subkernels. Although many other chaining subproblems remain unsolved, the primary contribution of this work is in bringing out the implicit relationship between the scheduling problem (discussed in Chapter 7) and the chaining problem (which attempts to minimize test time based on the test scheduling and test application schemes to be used). One interesting result established in the multiple chain study is that contrary to traditional wisdom, equal-length chains are not necessarily optimal; depending on the kernel characteristics and the test application scheme, the lowest overall test time may actually be achieved by using *unequal*-length multiple chains.

The results obtained in this chapter can be used to provide insights into more areas of the chaining problem space. The following is a list of directions that can be taken in future research.

- Take geometric constraints such as maximum routing area, maximum wire length between scan FFs, and maximum clock wire length into account.
- Use the total number of pins available as scan-in, scan-out, or scan path control pins as a constraint, rather than assuming the number of chains as given.
- In the multiple chain case, deal effectively with the case of more than two kernels.
- Allow kernels to be incompatible, in both the single chain and multiple chain problems.
- In the multiple chain case, use the canonical overlapped test scheme rather than the quasi-overlapped test scheme.

- In the separate and overlapped schemes, reduce the amount of shifting to apply each test by "pipelining" the shifting in/out of test patterns/results respectively. The modified test schemes, *pipelined separate* and *pipelined overlapped*, would help to reduce test time, but would need certain sets of scan registers to inhibit loading data at their inputs during certain clock cycles.

- Allow the scan chains to be *reconfigurable* using multiplexers. The overall test time could be reduced, at the expense of control complexity, by bypassing certain segments of the chains in some sessions.

Each of the extensions above requires a fair amount of analysis. The last idea, which suggests reconfigurable chains, completely redefines the chain construction problem by adding a whole new degree of freedom. Clearly there is a wide expanse of chaining subproblems yet to be explored in addition to those that have been addressed in this work.

Chapter 9

Conclusion

"The greater our knowledge increases, the greater our ignorance unfolds."
—John F. Kennedy, 1962.

This thesis has attempted to solve many of the design issues associated with well-known serial scan design techniques. Scan design has traditionally been viewed within the design community as a rigid and expensive technique. However, the work presented here describes and analyzes many different ways in which scan design can be applied, leading to a more flexible design methodology in which high testability can be achieved and different design costs (such as area overhead, test time, I/O pin count) can be traded off against each other.

The results of this work are being implemented in a prototype software system called SIESTA (System for IntEgrated Scan Test Application) [30]. SIESTA is built upon Cbase, an object-oriented framework for VLSI design and test applications [31]. Figure 9.1 shows the system architecture and its relationship to Cbase and other tools. Bold lines indicate portions of the work that are already implemented; broken lines indicate modules that are under development. SIESTA interacts with the user in text and graphic form by making use of the Cbase graphic user interface, as shown in Figure 9.2.

The research presented here can be classified into four major areas: partial scan, partitioning, test scheduling, and scan path chaining. Below we describe the

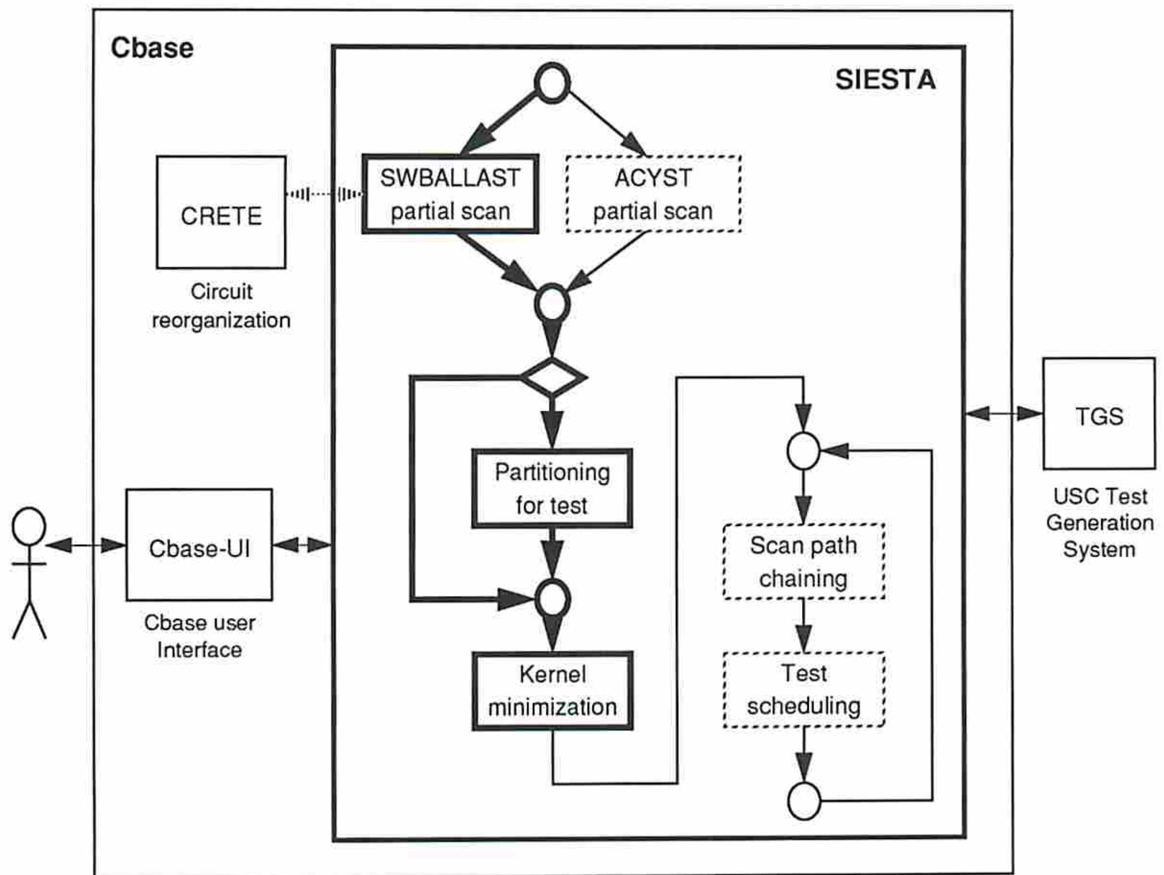


Figure 9.1: SIESTA 1.0 system architecture.

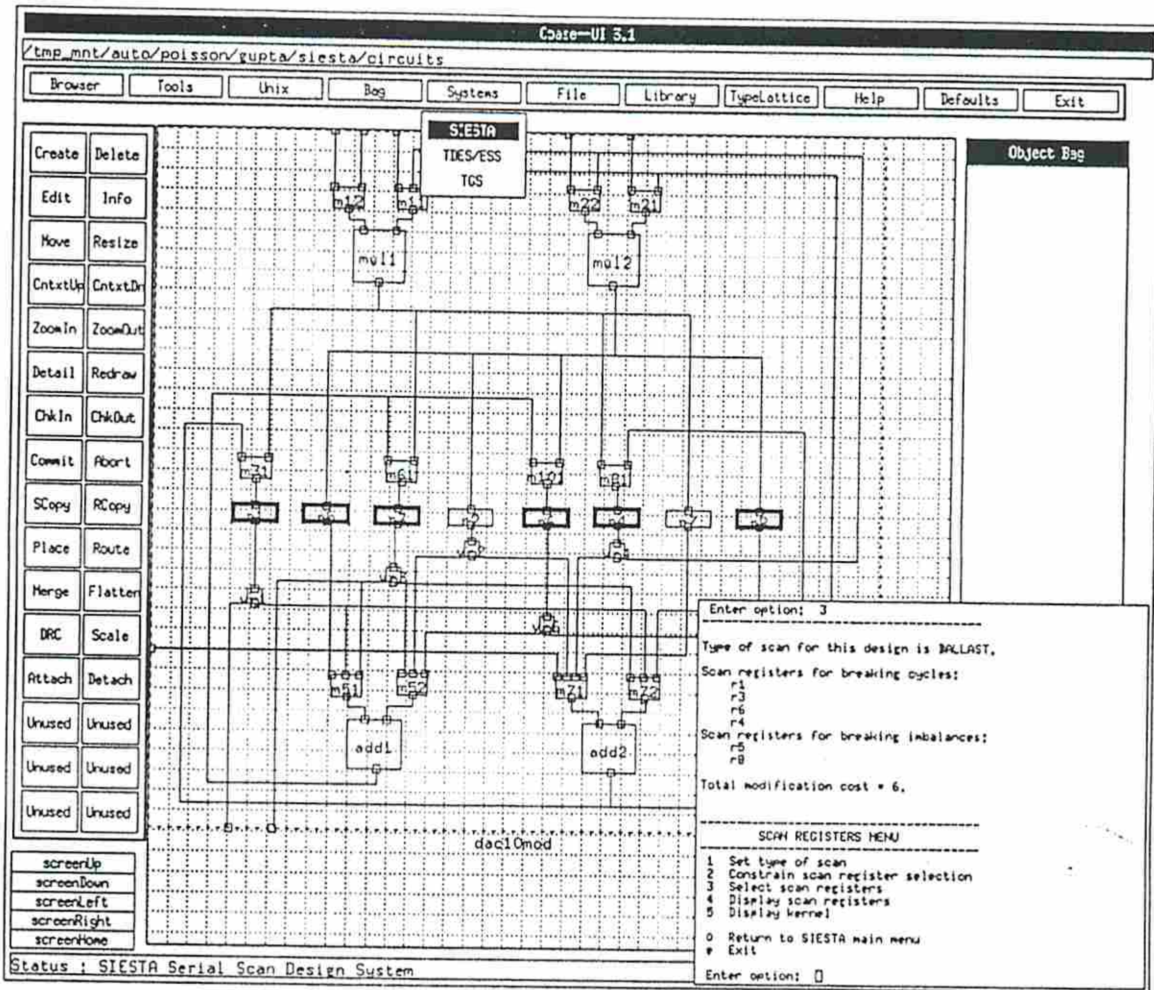


Figure 9.2: SIESTA user interface.

contributions made in each area by this thesis. We also list some of the open problems for future research, which in some cases are related problems that could not be fully addressed in this work, but in other cases arise as a direct consequence of the contributions in this work itself.

9.1 Partial Scan Design

The work on partial scan design has shown that there exist partial scan designs, with reduced overheads compared to full scan, that can achieve the main benefits of full scan itself. The partial scan approaches presented in this work can be applied to arbitrary synchronous circuits; however, they are most effective with data path and pipelined circuits. Although some area cost is unavoidable with partial scan design, in general the area cost is lower than with full scan. Perhaps more important, the partial scan design approach allows a range of designs to be generated, each with different area/performance characteristics, and in fact can allows area cost to be traded off against performance cost. The BALLAST partial scan technique, presented in Chapter 3, achieves full coverage of all detectable single stuck-at (SSA) faults using simple combinational ATPG, thus enjoying the main benefit of full scan design at reduced cost.

This work has also explored two different extensions to the BALLAST technique. The first, called ACYST, presented in Chapter 4, deals with kernels that are acyclic but allows them to be unbalanced. This leads to designs that have lower overheads compared to BALLAST but require some additional work in test generation. In particular, simple combinational ATPG cannot be used, although a combinational ATPG program that can handle multiple stuck-at (MSA) faults is sufficient. In general for an unbalanced kernel a given test consists of a sequence of test patterns. In the ACYST approach, the structure of the kernel is analyzed to determine an efficient way to compact the test sequences, i.e., to reduce the number of patterns in a given test sequence (for an arbitrary fault) to a minimal value. This helps to minimize the test application time. ACYST also generates a condensed *test generation model* (TGM). By carrying out combinational ATPG with an MSA fault

model on the condensed TGM, a complete set of minimal-length test sequences can be generated.

Another extension to BALLAST, present in Chapter 5, takes advantage of any *switches* (MUXes or buses) present in the circuit to reduce test-related costs whenever possible. Switches can be utilized for test in many different ways. Two main uses of switches are studied in this work. The first takes advantage of the fact that kernels that appear to be unbalanced may actually show balanced behavior in the presence of switches. By using this information, the overhead due to partial scan design can be reduced by taking advantage of switches that are present in the circuit for normal operation. The second use of switches is to set up data transfer paths called *I-paths* within the kernel. These can help to transport test data during test and thus gain access to inner portions of the kernel. This leads to reduced ATPG cost and can also help to increase the degree of sharing of test resources (i.e., scan path registers) among different parts of the circuit.

Figure 9.3 shows the space of partial scan designs for a given circuit. The horizontal axis represents the degree of scan, i.e., the fraction of the circuit's FFs that are included in the scan path. The vertical axis shows the cost of test generation for each given partial scan design. The space is divided into three regions. The leftmost region consists of partial scan designs generated by BALLAST, which have balanced (acyclic) kernels. The middle region consists of ACYST scan designs, which have acyclic but unbalanced kernels. The remaining region of the space consists of designs whose kernels contain cycles. Although the overheads in this region are lower than in the other regions, in general the ATPG cost for these circuits is orders of magnitude higher. Most earlier work has focussed on this region of the design space; however, no conclusive solutions have been found and this remains an area for future study. In this thesis, the main contribution of the work on partial scan presented has been to identify the various regions in the design space and to effectively generate solutions in the balanced and acyclic unbalanced regions.

Areas for future work in the partial scan domain can be summarized as follows:

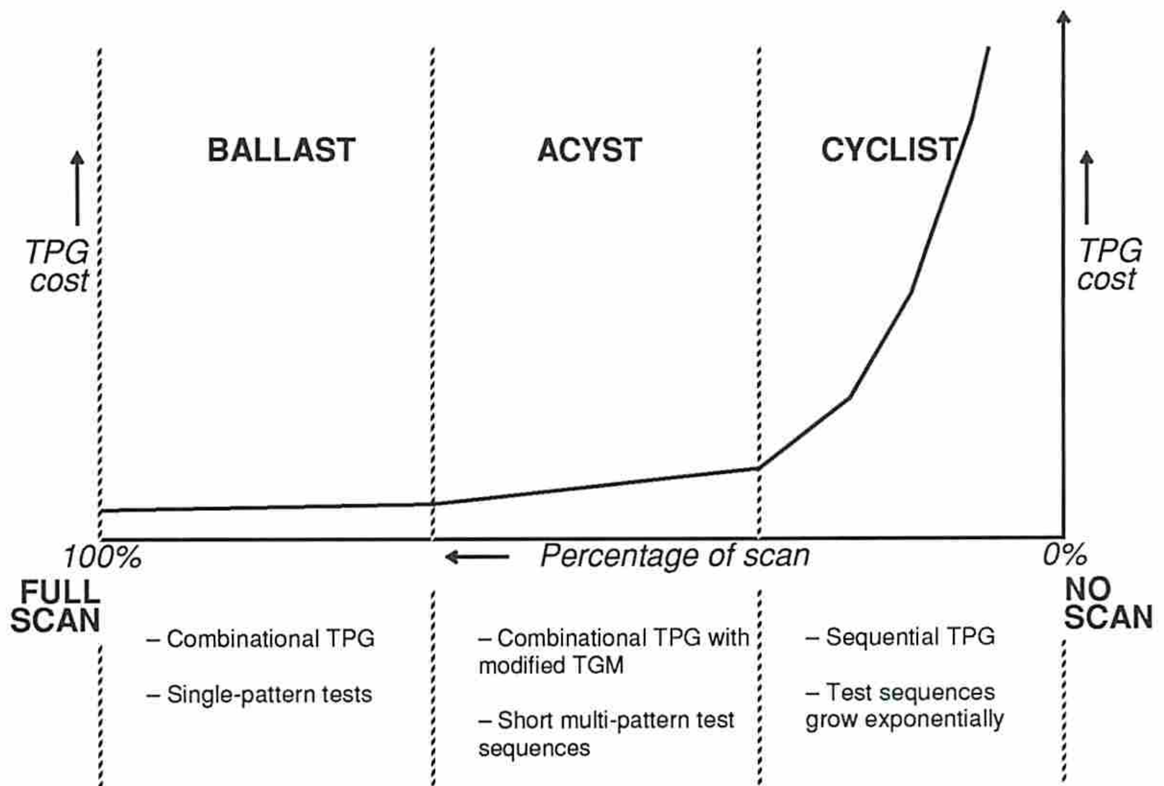


Figure 9.3: The partial scan design space.

- Extending the BALLAST and ACYST techniques to deal with circuits containing storage elements other than D-latches and D-FFs, e.g., J-K-FFs, T-FFs, and S-R-FFs.

- Allowing for register/switch controls that are derived within the circuit rather than fed by primary inputs, and also allowing for gated clocks and/or multiple clocks.

- Studying the coverage of non stuck-at faults and developing techniques for generating tests for these faults using partial scan.

- In this work I-paths consist of only switches and registers; they could be generalized to contain modules like adders that are capable of transmitting data unchanged or through a bijective transformation.

- Developing a partial scan strategy for the cyclic region of the design space.

9.2 Partitioning

In Chapter 6, different ways of subdividing an acyclic/balanced kernel in a partial scan design have been presented. The partitioning process is driven by two main objectives: reduction in ATPG cost and reduction in test application time. Unfortunately, with the current state of the art, it is not possible to accurately link the partitioning process to these circuit parameters. In the absence of good estimators for these parameters, any *elaborate* partitioning scheme would be meaningless, since it is analogous to a blind man taking pains to paint a beautiful picture when he cannot even see what he is doing.

What is clear, however, is that both the parameters tend to increase with increasing circuit size. In fact, the ATPG problem is theoretically intractable, although good heuristics tend to increase at a polynomial rate (between $O(n^2)$ and $O(n^3)$) in the average case. Test length is also generally higher for large circuits. The *size* of a circuit could be defined as one or a combination of several circuit features such as the number of gates, number of primary inputs, or the number of

gates in any input–output path. Thus for very large circuits even a rudimentary form of partitioning could potentially be useful for reducing test costs.

In Chapter 6 a heuristic approach for logically partitioning a kernel so as to bound the maximum size of each resulting subkernel is presented. The size function is assumed to be defined by the user based on the specific ATPG and test application environments. Three different types of partitioning are used to break down the kernel: output-based, switch-based, and size-based. We have demonstrated that partitioning a circuit can not only reduce ATPG cost but also reduce the overall test time (see Chapter 7), provided the registers in the scan path are chained appropriately so as to maximize the efficiency of the partitioned test.

The main contribution of the partitioning study presented here is to demonstrate the usefulness of partitioning, and to help generate partitioning solutions in the design space. However, further work is required in the following areas.

- Studying the relationship of kernel features, such as size, number of inputs/outputs, and test length, to costs like ATPG effort and test time. The results could be used for developing fast estimators of these costs.
- Improving the procedure for merging subkernels (**processKernels**), based on either theoretical or empirical studies, and possibly using test length as one of the criteria used for merging.
- Developing more effective ways of size-based partitioning, by using additional size criteria, and by making the partition sizes uniform wherever possible.
- Enhancing the global partitioning heuristic, to make it more sensitive to the actual ATPG cost and test time for the resulting design.
- Taking into account the organization of the scan chain(s), if this information is available, to influence the formation of partitions for minimum overall test time.

The work described above would help in better allocating resources such as chip area overhead, test time, and CPU time for ATPG, and in trading them off against each other using various different partitioning solutions.

9.3 Test Scheduling

Given the various subkernels produced by partitioning, we have developed an algorithm in Chapter 7 for scheduling the tests for them in a time-efficient manner. Previous research on the test scheduling problem has been focused mostly on circuits using the built-in self test (BIST) methodology. We have shown that scheduling problem for scan-testable circuits cannot be modeled using these earlier problems because in our problem, the test time for each kernel is not a constant; rather, it depends on what other kernels are to be tested at the same time. A new model based on a *test relationship graph* (TRG), which contains the information required for scheduling the tests, is presented in Chapter 7.

The manner in which the various scan registers in the circuit are organized into a single scan chain or into multiple scan chains has a strong influence on the scheduling of tests. The presence of multiple chains with unequal lengths leads to complex interactions among the kernels during test, depending on what chains are involved in testing each kernel. The TRG essentially captures the different types of interactions among the kernels and allows schedule information to be easily computed.

Based on the TRG model, an algorithm for searching the solution space to obtain the best test schedule is presented in Chapter 7. The algorithm assumes that the configuration of the scan chains is known. It allows one of two different scheduling disciplines to be specified. The first, *scheduling with interruptions*, allows the test for a kernel to be interrupted temporarily while the test control for the circuit is reconfigured to test a different subset of kernels. The second, *scheduling without interruptions*, requires that once a test for a kernel is started, it must be carried out in a continuous period of time with the same control configuration throughout, although the tests for other kernels compatible with it may be initiated or completed during this period. Although the same basic scheduling algorithm is used in both cases, the algorithm obtains an optimal schedule with the latter discipline but may sometimes result in a suboptimal schedule with the former discipline.

Future work on the test scheduling problem could be carried out in the following directions.

- Allowing for some or all scan registers to have individual HOLD controls, which permits tests to be applied in a pipelined manner with a lower chain cycle as defined in Section 7.2.

- Improving the efficiency of the implicit enumeration algorithm for scheduling, possibly by introducing a *bounding* criterion to further prune the search space. Alternatively, developing fast suboptimal heuristics.

- Expanding the search space for schedules by using something other than a "first fit" strategy, or by removing the constraint that kernels must be considered one at a time in sequence. Thus for example after a portion of a test is executed, the remainder of the test could be considered later in the sequence. Expanding the search space in this way could lead to better solutions.

9.4 Scan Path Chaining

The scheduling problem described above assumes that the configuration of single/multiple scan chains is fixed. The chaining problem, however, is to determine a configuration of the chains themselves such that it leads to the most optimal test schedule. A chain configuration includes both the assignment of scan FFs to chains (in the case of multiple scan chains) and the ordering of scan FFs in each chain (in both the multiple and single scan chain cases).

The optimal chaining problem in its most general form is extremely complex and frustrating because of the many degrees of freedom involved in constructing the scan chains. The main achievements in this work with respect to the chaining problem are to bring out its interaction with the scheduling problem, and to solve two constrained chaining problems. Both the constrained problems involve *fully compatible circuits*, i.e., circuits in which all kernels can be tested simultaneously, hence they are applicable to full scan designs.

The first study deals with constructing a single scan chain, and ordering the FFs within it such that the test time is minimized. An algorithm for solving this problem is presented, which returns an ordering of the FFs in the chain. It does not

always determine an optimal ordering; however, as a part of the solution it returns a confidence level which indicates the level of optimality of the ordering.

The second study deals with constructing a given number of scan chains for a circuit consisting of exactly two compatible kernels. The problem is to assign each scan FF to one of the chains such that the test time is minimized. A special test application scheme that does not depend on the ordering of FFs in each chain is used. This problem is tackled by reducing it to a finite and reasonably small number of integer linear programming problems. By solving each of these problems, the solution that has the lowest test time can be taken as the globally optimal solution.

Although only two constrained chaining problems have been considered in this work, it is hoped that the experience and insight gained from the analysis will help to mount an assault, in future research, on the most general form of the scan path chaining problem. Some of the issues to be tackled are listed below.

- Extending the model used in the multiple chain study to the overlapped test application scheme (rather than the quasi-overlapped scheme).
- Constraining FFs that belong to the same register to be assigned to the same scan chain and to be adjacent to each other within the chain.
- Developing chaining solutions for circuits with incompatibilities among kernels, for both single and multiple chains.
- Designing reconfigurable scan chains through the use of multiplexers, so that the chain configuration can be altered in different sessions.
- Taking into account the routing area overhead for different scan chain configurations, and generating a range of solutions that trade off routing overhead vs. test time.
- Allowing for different types of clocking schemes, which affect the way in which scan path registers fed by distinct clock signals can be connected to each other.

Reference List

[1] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Rockville, Md., 1976.

[2] B. Könemann, J. Mucha, and G. Zwiehoff. Built-in logic block observation techniques. In *Proceedings, IEEE International Test Conference*, pages 37–41, October 1979.

[3] E. B. Eichelberger and T. W. Williams. A logic design structure for LSI testability. In *Proceedings, 14th Design Automation Conference*, pages 462–467, June 1977.

[4] M. S. Abadir and M. A. Breuer. A knowledge-based system for designing testable VLSI chips. *IEEE Design & Test of Computers*, 2(4):56–68, August 1985.

[5] M. J. Y. Williams and J. B. Angell. Enhancing testability of LSI circuits via test points and additional logic. *IEEE Transactions on Computers*, C-22:46–60, 1973.

[6] J. H. Stewart. Future testing of large LSI circuit cards. In *Digest of Papers, IEEE Semiconductor Test Conference*, pages 6–15, October 1977.

[7] S. Funatsu, N. Wakatsuki, and A. Yamada. Designing digital circuits with easily testable consideration. In *Digest of Papers, IEEE Semiconductor Test Conference*, pages 98–102, 1978.

[8] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semiconductor Circuits.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[9] E. Trischler. Design for testability using incomplete scan path and testability analysis. *Siemens Forsch.- u. Entwickl.-Ber.*, 13(2):56–61, 1984.

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

[11] H.-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. An incomplete scan design approach to test generation for sequential machines. In *Proceedings, IEEE International Test Conference*, pages 730–734, September 1988.

[12] Rajiv Gupta, R. Srinivasan, and M. A. Breuer. A methodology for partitioning and hierarchical reorganization of sequential circuits for DFT and BIST. Technical Report CENG 90-19, University of Southern California, 1990.

[13] E. J. McCluskey and S. Bozorgui-Nesbat. Design for autonomous test. *IEEE Transactions on Computers*, C-30(11):866–875, November 1981.

[14] S. Bhawmik. *An Integrated CAD System for the Design of Testable VLSI Circuits.* PhD thesis, Indian Institute of Technology, Kharagpur, India, February 1988.

[15] S. Al-Hariri and F. Ozguner. An easily testable structure for LSI and VLSI circuits. In *Proceedings, National Electronic Conference*, pages 346–350, October 1982.

[16] K.-T. Cheng and V. D. Agrawal. An economical scan design for sequential logic test generation. In *Proceedings, Fault-Tolerant Computing Symposium (FTCS-19)*, pages 28–35, June 1989.

[17] A. Kunzmann. Produktionstest synchroner Schaltwerke auf der Basis von Pipelinestrukturen. In *Proceedings, 18. Jahrestagung der Gesellschaft für Informatik, Hamburg*, pages 92–105, 1988. Informatik-Fachberichte 188, Springer-Verlag.

[18] V. Chickermane and J. H. Patel. An optimization based approach to the partial scan design problem. In *Proceedings, IEEE International Test Conference*, pages 377–386, September 1990.

[19] P. P. Fasang, J. P. Shen, M. A. Schuette, and W. A. Gwaltney. Automated design for testability of semicustom integrated circuits. In *Proceedings, International Test Conference*, pages 558–564, 1985.

[20] A. Miczo. *Digital Logic Testing and Simulation.* Harper & Row, New York, 1986.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, 1979.

[22] H. W. Lenstra. The acyclic subgraph problem. Technical Report BW 26/73, Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam, July 1973.

[23] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1983.

[24] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[25] Rajesh Gupta, Rajiv Gupta, and M. A. Breuer. BALLAST: A methodology for partial scan design. In *Proceedings, Fault-Tolerant Computing Symposium (FTCS-19)*, pages 118–125, June 1989.

[26] R. Woudsma, F. P. M. Beenker, J. L. van Meerbergen, and C. Niessen. PIRAMID: An architecture-driven silicon compiler for complex DSP applications. In *Proceedings, International Symposium on Circuits and Systems*, pages 2596–2600, May 1990.

[27] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-30(3):215–222, March 1981.

[28] G. L. Craig, C. R. Kime, and K. K. Saluja. Test scheduling and control for VLSI built-in self-test. *IEEE Transactions on Computers*, 37(9):1099–1109, September 1988.

[29] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Chichester, England, 1990.

[30] Rajesh Gupta. *SIESTA 1.0 User's Manual.* University of Southern California, Department of Electrical Engineering–Systems, Los Angeles, CA 90089-0781, February 1991.

[31] Rajiv Gupta, W. H. Cheng, Rajesh Gupta, I. Hardonag, and M. A. Breuer. An object-oriented VLSI CAD framework: A case study in rapid prototyping. *IEEE Computer*, pages 28–37, May 1989.