

**Design of Hierarchically Testable  
and Maintainable Systems**

Jung-Cheun Lien

CEng Technical Report 91-19

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 August 1991)

Electrical Engineering Systems  
University of Southern California  
Los Angeles, CA. 90089-2562

July 15, 1991

DESIGN OF HIERARCHICALLY TESTABLE AND MAINTAINABLE  
SYSTEMS

by

Jung-Cheun Lien

---

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)

August 1991

Copyright 1991 Jung-Cheun Lien

UNIVERSITY OF SOUTHERN CALIFORNIA  
THE GRADUATE SCHOOL  
UNIVERSITY PARK  
LOS ANGELES, CALIFORNIA 90007

*This dissertation, written by*

*JUNG-CHEUN LIEN*

*under the direction of his..... Dissertation  
Committee, and approved by all its members,  
has been presented to and accepted by The  
Graduate School, in partial fulfillment of re-  
quirements for the degree of*

DOCTOR OF PHILOSOPHY



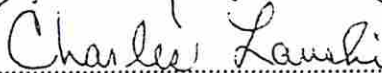
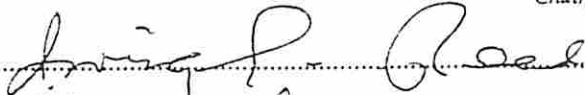
*Dean of Graduate Studies*

Date ..... July 18, 1991.....

DISSERTATION COMMITTEE



*Chairperson*



# Dedication

To  
My wife Jung-Yu and our parents



## Acknowledgements

I am very grateful to Prof. Melvin Breuer for his encouragement, guidance and support during my dissertation work. I consider it my privilege to have worked with him.

I would also like to thank Prof. Irving Reed and Prof. Charles Lanski for serving on my dissertation committee.

During my years at USC, I benefited greatly from interacting with many colleagues and friends. In particular I would like to mention Kuen-Jong Lee, Dr. Rajesh Gupta, Rajagopalan Srinivasan, Amitava Majumdar, Sing-Ban Tien, Dr. Rajiv Gupta and Dr. Charles Njinda.

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 No. JFBI90092, monitored by the Federal Bureau of Investigation.

Finally, I would like to thank my wife Jung-Yu for her endless love and support. This dissertation would never have been completed without her.

# Contents

|   |           |
|---|-----------|
| Dedication                                      | ii        |
| Acknowledgements                                | iii       |
| Abstract  | xiii      |
| <b>1 Introduction</b>                           | <b>1</b>  |
| 1.1 Background . . . . .                        | 1         |
| 1.2 Previous Work . . . . .                     | 3         |
| 1.2.1 System Architecture . . . . .             | 3         |
| 1.2.2 Testability Bus . . . . .                 | 4         |
| 1.2.3 Testable Chip Control Model . . . . .     | 5         |
| 1.2.4 Testable Chip Design . . . . .            | 6         |
| 1.2.5 Testable Module Design . . . . .          | 7         |
| 1.2.6 Interconnect Test Generation . . . . .    | 7         |
| 1.2.7 Test Scheduling . . . . .                 | 9         |
| 1.3 The Design Approach . . . . .               | 10        |
| 1.4 The Hierarchical Test Methodology . . . . . | 10        |
| 1.5 Test Controllers for a Circuit . . . . .    | 13        |
| 1.6 Thesis Outline . . . . .                    | 15        |
| <b>2 Controllers for Testable Chips</b>         | <b>17</b> |
| 2.1 The Model . . . . .                         | 17        |

|          |   |           |
|----------|---|-----------|
| 2.1.1    | Boundary Scan Architecture . . . . .              | 17        |
| 2.2      | Bus-Dependent BIT Controllers . . . . .           | 23        |
| 2.2.1    | BIT Controller for a LSSD Kernel . . . . .        | 26        |
| 2.2.2    | BIT Controller for a BILBO Kernel . . . . .       | 27        |
| 2.2.3    | BIT Controller for a Complex Kernel . . . . .     | 29        |
| 2.2.4    | A Mapping Algorithm . . . . .                     | 33        |
| 2.3      | Autonomous BIT Controller . . . . .               | 38        |
| 2.3.1    | Serial BIT Controllers . . . . .                  | 38        |
| 2.3.1.1  | A Hard-Wired Serial BIT Controller . . . . .      | 38        |
| 2.3.1.2  | A Microprogrammed Serial BIT Controller . . . . . | 40        |
| 2.3.2    | Parallel BIT Controllers . . . . .                | 43        |
| 2.3.2.1  | Interleaved FSM Controller . . . . .              | 44        |
| 2.3.2.2  | Tree of Counters Design . . . . .                 | 48        |
| 2.3.2.3  | Counter Sharing Design . . . . .                  | 53        |
| 2.3.2.4  | Comparison of Three Designs . . . . .             | 55        |
| <b>3</b> | <b>Controller for Testable Modules</b>            | <b>60</b> |
| 3.1      | Requirements for an MMC . . . . .                 | 61        |
| 3.2      | MMC Architecture . . . . .                        | 62        |
| 3.2.1    | Test Channel Design . . . . .                     | 63        |
| 3.2.2    | Bus Driver/Receiver . . . . .                     | 73        |
| 3.2.3    | Functional Bus Interface . . . . .                | 74        |
| 3.2.4    | Testability Register . . . . .                    | 74        |
| 3.2.5    | Analog Test Interface . . . . .                   | 75        |
| 3.2.6    | L1-Slave . . . . .                                | 75        |
| 3.2.7    | Processor . . . . .                               | 75        |
| 3.2.8    | Memory . . . . .                                  | 80        |
| 3.2.9    | Stand-Alone MMC . . . . .                         | 81        |
| 3.3      | MMC Self-Test . . . . .                           | 81        |

|          |  |            |
|----------|--|------------|
| 3.4      | Discussion of MMC Design . . . . .                                 | 83         |
| 3.5      | An MMC Prototype . . . . .   | 85         |
| 3.5.1    | Test Channel . . . . .   | 85         |
| 3.5.2    | Processor and Memory . . . . .                                     | 86         |
| 3.5.3    | Processor and Test Channel Interface . . . . .                     | 86         |
| 3.5.4    | Discussion . . . . .   | 89         |
| 3.6      | Testing a Kernel Using MMC and CMC . . . . .                       | 90         |
| <b>4</b> | <b>Test Program Synthesis</b>                                      | <b>93</b>  |
| 4.1      | Languages . . . . .  | 95         |
| 4.1.1    | CTL - The Chip Test Language . . . . .                             | 95         |
| 4.1.1.1  | Formal Definition of the CTL . . . . .                             | 103        |
| 4.1.2    | MTL - The Module Test Language . . . . .                           | 104        |
| 4.1.2.1  | Formal Definition of the MTL Syntax . . . . .                      | 109        |
| 4.2      | Synthesizers . . . . .   | 111        |
| 4.2.1    | C2C Synthesizer . . . . .  | 111        |
| 4.2.2    | M2C Synthesizer . . . . .  | 111        |
| 4.3      | An Example . . . . .   | 121        |
| 4.3.1    | An MTL-file . . . . .  | 121        |
| 4.3.2    | CTL-files . . . . .  | 122        |
| 4.3.3    | The Interconnect Information . . . . .                             | 124        |
| 4.3.4    | Synthesized Test Programs . . . . .                                | 125        |
| 4.3.5    | Activities between Processor and Test Channel . . . . .            | 127        |
| 4.3.6    | Activities on the Test Bus . . . . .                               | 128        |
| 4.4      | Results . . . . .  | 129        |
| <b>5</b> | <b>Global Controller Minimization Using Test Program Synthesis</b> | <b>132</b> |
| 5.1      | Tradeoff Curve-Based Minimization . . . . .                        | 134        |
| 5.2      | Algorithm-Based Minimization . . . . .                             | 140        |
| 5.2.1    | One Test Channel . . . . .   | 141        |

|          |   |            |
|----------|---|------------|
| 5.2.2    | Multiple Test Channels . . . . .                      | 144        |
| 5.2.3    | Results . . . . .                                     | 147        |
| 5.3      | Discussions . . . . .                                 | 148        |
| <b>6</b> | <b>Interconnect Test Generation</b>                   | <b>150</b> |
| 6.1      | Introduction . . . . .                                | 150        |
| 6.2      | Preliminaries . . . . .                               | 152        |
| 6.2.1    | Fault Model . . . . .                                 | 152        |
| 6.2.2    | Notation and Definitions . . . . .                    | 154        |
| 6.2.3    | Non-Diagnosable Faults . . . . .                      | 155        |
| 6.2.4    | Diagnostic Resolution . . . . .                       | 156        |
| 6.2.5    | Previous Results . . . . .                            | 157        |
| 6.2.6    | Deficiencies in Previous Approaches . . . . .         | 159        |
| 6.3      | One-Step Diagnosis . . . . .                          | 161        |
| 6.4      | Two-Step Diagnosis . . . . .                          | 165        |
| 6.4.1    | Adaptive Algorithm A1 . . . . .                       | 165        |
| 6.4.2    | Adaptive Algorithm A2 . . . . .                       | 167        |
| 6.4.3    | Comparison with Other Adaptive Algorithms . . . . .   | 169        |
| 6.5      | Diagnosis Using Structural Information . . . . .      | 171        |
| <b>7</b> | <b>Interconnect Test Scheduling</b>                   | <b>175</b> |
| 7.1      | Introduction . . . . .                                | 175        |
| 7.2      | Testing Model . . . . .                               | 177        |
| 7.3      | The Problem . . . . .                                 | 180        |
| 7.3.1    | The Use of Multiple Scan Chains . . . . .             | 180        |
| 7.3.2    | Scheduling Problem in Testing Interconnects . . . . . | 183        |
| 7.4      | Optimal Test Scheduling Theorems . . . . .            | 185        |
| 7.5      | An Algorithm for Generating Schedules . . . . .       | 192        |
| 7.6      | An Extension to Full Scan . . . . .                   | 200        |

|          |  |            |
|----------|--|------------|
| <b>8</b> | <b>Conclusions and Future Research</b> | <b>202</b> |
| 8.1      | On-Chip Test Controller . . . . .      | 202        |
| 8.2      | Module Test Controller . . . . .       | 203        |
| 8.3      | Test Program Synthesis . . . . .       | 204        |
| 8.4      | Controller Minimization . . . . .      | 204        |
| 8.5      | Interconnect Test Generation . . . . . | 205        |
| 8.6      | Interconnect Test Scheduling . . . . . | 206        |
| 8.7      | Future Research . . . . .              | 206        |
| 8.7.1    | On-chip Test Controller . . . . .      | 207        |
| 8.7.2    | Module Test Controller . . . . .       | 208        |
| 8.7.3    | Test Program Synthesis . . . . .       | 209        |
| 8.7.4    | Controller Minimization . . . . .      | 210        |
| 8.7.5    | Interconnect Test . . . . .            | 211        |

## List Of Tables

|     |   |     |
|-----|---|-----|
| 2.1 | Test schedule for the complex kernel. . . . .                             | 31  |
| 2.2 | Microinstruction List for Controller. . . . .                             | 41  |
| 3.1 | Counter usage. . . . .  | 70  |
| 3.2 | Processor instruction set. . . . .  | 78  |
| 4.1 | The numbers used in the shifting. . . . .                                 | 116 |
| 4.2 | Synthesis results for some modules. . . . .                               | 130 |
| 7.1 | Typical results; (a) $N_S$ , (b) saving $SS$ (in %) on test time. . . . . | 199 |

## List Of Figures

|      |   |    |
|------|---|----|
| 1.1  | The architecture of a HTM system. . . . .                         | 12 |
| 1.2  | Partitioning the test controller between the CMC and MMC. . . . . | 13 |
| 2.1  | A typical boundary scan cell. . . . .                             | 18 |
| 2.2  | A module having a boundary scan path. . . . .                     | 18 |
| 2.3  | Test bus architecture of a module. . . . .                        | 19 |
| 2.4  | The model of a chip with boundary scan architecture. . . . .      | 20 |
| 2.5  | The boundary scan bus state transition diagram. . . . .           | 21 |
| 2.6  | Control model for an addressable register. . . . .                | 22 |
| 2.7  | The control signals during test mode. . . . .                     | 23 |
| 2.8  | A LSSD kernel; (a) control signals, (b) control graph. . . . .    | 25 |
| 2.9  | A general model for the bus-dependent BIT controller. . . . .     | 26 |
| 2.10 | A BILBO kernel: (a) control signals, (b) control graph. . . . .   | 28 |
| 2.11 | A complex kernel; (a) control signals, (b) control graph. . . . . | 30 |
| 2.12 | An autonomous BIT controller for a LSSD kernel. . . . .           | 39 |
| 2.13 | Testing many kernels in sequence. . . . .                         | 40 |
| 2.14 | Microprogram controller. . . . .                                  | 42 |
| 2.15 | Interleaved FSM controller. . . . .                               | 45 |
| 2.16 | A controller for interleaved test execution. . . . .              | 48 |
| 2.17 | A Tree-Of-Counter Design. . . . .                                 | 50 |
| 2.18 | Three Trees-Of-Counters. . . . .                                  | 52 |
| 2.19 | A counter-Sharing design. . . . .                                 | 54 |
| 2.20 | Hardware complexity for different designs. . . . .                | 56 |



|      |   |     |
|------|---|-----|
| 2.21 | Time complexity for different designs. . . . .                          | 58  |
| 3.1  | The architecture of an MMC. . . . .                                     | 63  |
| 3.2  | The architecture of the test channel. . . . .                           | 65  |
| 3.3  | The state transition diagram for a test channel. . . . .                | 70  |
| 3.4  | The state transition diagram (cont.). . . . .                           | 71  |
| 3.5  | The Bus Driver/Receiver. . . . .  | 74  |
| 3.6  | A Testability Register; (a) block diagram (b) circuit of bit i. . . . . | 76  |
| 3.7  | Analog Test Interface. . . . .  | 77  |
| 3.8  | Control signals for MR and MRMW instructions. . . . .                   | 79  |
| 3.9  | Testable design features for a test channel. . . . .                    | 82  |
| 3.10 | Physical configuration of the MMC prototype. . . . .                    | 86  |
| 3.11 | The data bus adaptor. . . . .   | 88  |
| 3.12 | Overview of the test control. . . . .                                   | 91  |
| 4.1  | Overview of the test hardware/software hierarchy. . . . .               | 94  |
| 4.2  | Test control model used in CTL. . . . .                                 | 97  |
| 4.3  | Test control model used in MTL . . . . .                                | 105 |
| 4.4  | Scanning a data register in a ring. . . . .                             | 108 |
| 4.5  | Generating test programs for a module. . . . .                          | 112 |
| 4.6  | The structure of the M2C. . . . .                                       | 113 |
| 4.7  | Model for calculating the number of shifting. . . . .                   | 116 |
| 5.1  | Possible partitions of test resources. . . . .                          | 135 |
| 5.2  | Test time versus controller complexity. . . . .                         | 137 |
| 5.3  | Tradeoff curve: Test time versus controller complexity. . . . .         | 139 |
| 5.4  | Test time estimated by algorithm TC1. . . . .                           | 144 |
| 6.1  | A soft stuck-at 1 case. . . . .   | 153 |
| 6.2  | A short to an opened net. . . . .                                       | 153 |
| 6.3  | An open that is non-diagnosable. . . . .                                | 155 |

|      |   |     |
|------|---|-----|
| 6.4  | A short that is non-diagnosable. . . . .  | 156 |
| 6.5  | A short that cannot be identified by a diagonally independent sequence. . . . .                 | 159 |
| 6.6  | An open that cannot be identified by a diagonally independent sequence. . . . .                 | 159 |
| 6.7  | A short that cannot be identified by an independent test set. . . . .                           | 160 |
| 6.8  | Achieving maximal diagnosis using a set-cover independent sequence. . . . .                     | 164 |
| 6.9  | A deficiency in the W-Test Algorithm. . . . .   | 170 |
| 6.10 | A deficiency in the C-Test Algorithm. . . . .   | 171 |
| 6.11 | Example 6-2: (a) the NG, (b) the ANG, (c) the colored graph. . . . .                            | 173 |
| 7.1  | Test interconnect via two boundary scan chains; (a) block diagram, (b) graph model. . . . .     | 178 |
| 7.2  | The test controller model. . . . .  | 179 |
| 7.3  | Different classes of interconnect. . . . .  | 181 |
| 7.4  | Two schemes for testing interconnect; (a) distributed control, (b) centralized control. . . . . | 182 |
| 7.5  | Deriving test schedules for several examples. . . . .   | 184 |
| 7.6  | Example: Deriving an optimal schedule. . . . .  | 193 |
| 7.7  | Testing a circuit via two scan chains; (a) block diagram, (b) graph model. . . . .              | 201 |

## Abstract

The cost associated with the test and maintenance of a complex system, which includes test generation, and detection, location and repair of faulty components, represents a significant portion of the overall life-cycle cost. This cost can be drastically reduced if testability and maintainability techniques are properly incorporated in the system design. Despite its importance, most systems are built with insufficient or no concern for testability and maintainability. This is mainly due to costs related to design-for-test structures, in designing test controllers, and the development of test programs.

This thesis presents a tool called BOLD that assists in the design of a system with extremely high degree of testability and maintainability. A system designed using BOLD is testable and maintainable at every level of the design hierarchy, i.e., chips, modules, subsystems and the system, since test controllers are employed in the hierarchy. Test busses are used to allow communication between test controllers. Methods for designing various test controllers are also presented along with some example designs. To describe the test aspects of various hardware units, a set of high-level languages are provided. These languages can be used by designers with little or no knowledge of testability. Tools are provided to synthesize test programs from these descriptions. These test programs are then compiled and executed by controllers so that hardware units can be tested. These test programs include test for the interconnect between chips. The test synthesis approach also provides the capability of predicting test time before the controllers are actually built. Given a bound on system test time and a requirement on system testability and maintainability, BOLD can quickly search the whole design space to provide the designer with the best solution satisfying the requirements of test time and hardware overhead.

# Chapter 1

## Introduction

### 1.1 Background

The increased complexity of modern digital systems has significantly increased the costs associated with the activities of test generation, detecting, locating and repairing faulty components. These activities are referred to as the test and maintenance of a system. For a complex system, these costs constitute a significant portion of its overall life-cycle costs.

Current approaches to system test and maintenance often employ a three level maintenance scheme. System self-test is initiated by personnel with little training. If a faulty field replaceable unit is found, it is replaced with a working spare. The faulty unit is then sent to a shop, where a technician uses sophisticated equipment to isolate the fault to a shop replaceable unit. The faulty unit is then discarded or sent back to a depot, where a well trained technician can locate a faulty component down to a minimum replaceable unit, such as a chip, and repair the unit.

This form of testing often encounters several problems such as those listed next: (1) the unreliability of the tester; (2) the increased time and skill required of maintenance personnel; (3) the loss of system capability when the tester failed; (4) the cost and quality in developing test software; and (5) the so-called Cannot Duplicate (CND) and Retest Okay (RTOK) problems. The problem of Cannot

Duplicate refers to the situation where one level of test and maintenance indicates a failure in a unit, while the next level cannot detect a fault in the unit. This is in part due to intermittent faults and/or differences in test procedures used at various level of testing. The problem of Retest Okay refers to the situation where the fault isolation capability is insufficient. Therefore, instead of sending the faulty unit to the next level of test, a good unit is sent. Thus no fault can be identified at the next level. It has been pointed out in [49] that the *false alarm*, which is the major factor in lowering the system readiness and availability, is closely related to both Cannot Duplicate and Retest Okay.

The major causes of these problems, which have been identified in [11], are (1) mode of operation dependency, referring to test procedures which can be executed at one level of the system hierarchy but not at other levels; (2) environmental dependency, which means that a failure is caused by such conditions as temperature or vibration; (3) false alarms due to design error or transient faults; (4) inadequate fault isolation; (5) incompatibility of tests and test tolerances, which is caused by the inconsistent testability and maintainability techniques used in different levels of testing; (6) system parameters which are out of specification; (7) faults in Built-In Test (BIT) hardware; (8) test data inaccessible; and (9) other causes which we have no knowledge about.

To reduce the cost of test and maintenance, one needs to design a system so that the above mentioned problems and their causes can be eliminated or at least reduced. Such a system would then have a high degree of testability and maintainability. In addition, the system should be able to perform self-test with a high degree of fault coverage. This leads to a two level maintenance scheme, where the shop level is eliminated, resulting a tremendous reduction in test and maintenance cost.

In summary, the ideal system should be able to (1) improve fault isolation capability; (2) eliminate the mode of operation dependency; (3) unify test and test tolerance among different levels of testing; and (4) detect faults in BIT hardware. Various issues related to the design of such a system are addressed in this work.



## 1.2 Previous Work

Previous work provides partial solutions for the problem of designing a system with a high degree of testability and maintainability. Work has been done in the areas of system architecture, testability busses, testable chip design, testable module design, interconnect test and test scheduling. However, a complete solution that deals with both the hardware and software aspects of the problem is not available. The BOLD system presented in this thesis can provide such a complete solution. Previous work that addresses various aspects of the problem is briefly described below.

### 1.2.1 System Architecture

Several system architectures to support system level design for testability and maintainability have been proposed. These architectures share features such as hierarchical design and component built-in self-test (BIST) capability.

Haedtke et al. [27] proposed a multilevel self-test architecture. Each level of integration of the system is assumed to have BIST capability and is controlled by a maintenance processor. A standardized test bus and a standardized BIST control protocol are required to enable simultaneous self-test of all chips. These same standardized interfaces and protocols are used by the maintenance processor in both the factory and field test. The lower level tests remains valid at each higher level of integration. Using this architecture, the system test and maintenance cost can be greatly reduced. The system self-test is carried out by the maintenance processor, which must be first tested by an external tester. It is not clear how the maintenance processor will be tested if no external tester is available. Thus the capability of system self-test is not assured.

IBM's Common Signal Processor architecture [19] is designed with built-in test and maintenance capability. Each chip has an on-chip monitor to control the chip's BIST circuitry. Each functional module is associated with an element supervisor unit and an element control bus. The former can access the on-chip monitor

through an element maintenance bus. A subsystem manager controls all the element supervisor units via a pi-bus. The Common Signal Processor architecture incorporates the test and maintenance hierarchy in accordance with the system functional hierarchy.

TRW employs a hierarchical architecture for the system test and maintenance [48]. A system maintenance node controls several functional maintenance nodes via a subsystem maintenance bus. A functional maintenance node controls several module maintenance nodes via a module test bus. A module maintenance node controls several device maintenance nodes via a device test bus.

The TEA system [62] also employs a hierarchical test methodology. Each board is made testable by inserting a *Test Switch* between every pair of *Ambiguity Groups*. An Ambiguity Group is the basic unit of circuitry to be tested. The test process of the board is controlled by a maintenance node, which is controlled by a subsystem test control unit, which is in turn controlled by a system test control unit.

### 1.2.2 Testability Bus

A great amount of effort has recently been focused on the interoperability among products from different vendors. Interoperability can be assured by using a standard test bus. Some major initiatives for the standardization of test busses are listed below.

**TM/ETM initiatives** [20, 60] A VHSIC subcommittee consisting of IBM, Honeywell and TRW proposed the TM/ETM test busses. The Element Test and Maintenance (ETM) bus [20] was to be the standard testability bus for VHSIC chips. It consists of 6 signal lines, namely CLK, DI, DO, Mode, Control, and Interrupt. Recently the ETM bus protocol has been modified to be compatible with the IEEE Std. 1149.1, which combines the Mode and Control into a single line (TMS) to reduce the overhead of pin count.

The Test and Maintenance (TM) bus [60] was proposed to be used as the backplane test bus. It consists of four lines, namely Clock, Control, MD and SD.

The bus configuration uses a multi-drop architecture, so that the addition/removal of other modules to/from a backplane will not affect the testability operation of other modules.

**IEEE Std. 1149.1** [33] The Joint Test Action Group (JTAG) proposed a standard test architecture, which became the IEEE Std. 1149.1. The bus consists of four signal lines, namely TCK, TMS, TDI and TDO. The TCK is the test clock line; the TDI is the serial data input line; the TDO is the serial data output line; and the TMS is the control line which controls the state of a Test Access Port (TAP) that resides on each chip. One of the main objectives of the IEEE Std. 1149.1 was to minimize both pin count and area overhead associated with the test bus.

**IEEE P1149.x initiatives** [61] The IEEE P1149 Testability Bus Standardization Committee was formed in an effort to standardize one or more testability busses [33, 61]. The P1149.x (x=2,3,4) proposal includes four subsets that could be used individually or in any combination. These subsets are the P1149.2 Extended Serial Digital Subset, the P1149.3 Real Time Digital Subset and the P1149.4 Real Time Analog Subset. These initiatives, once approved, will become the IEEE standard 1149.

### 1.2.3 Testable Chip Control Model

Craig et al. [18] proposed a hierarchical test control scheme. In this scheme, a piece of automatic test equipment (ATE) controls a level 2 supervisor, which controls several level 1 supervisors which in turn controls the Test Control Logic in self-testable units, which are the basic unit for test scheduling and test application. In addition to the Test Control Logic, each self-testable unit contains several test resources and a circuit block under test. A problem occurs when a test resource, such as a signature analyzer (SA), is shared among several units. Each Test Control Logic in these units must provide a set of control signals to the test resource. Since the test resource can only be controlled by a single set of control signals, control signals provided by different Test Control Logic blocks must be combined (either



wired ORed or ANDed). This not only increases area overhead but also slows down the functional operation.

Beausang and Albicki [7] proposed a model for self-testable chips. Their model describes test resources, the test distribution network, the test controller and the test procedure in mathematical form. Necessary and sufficient conditions for a test controller to implement a test procedure are also derived. The problem of interfacing this model to a standard test bus is not treated. Thus, it is not clear how to incorporate their work with an external test controller, such as an MMC.

#### 1.2.4 Testable Chip Design

Avra [5] proposed a design for an ETM-BUS compatible on-chip test and maintenance controller (TMC). The controller, under the control of an ETM-BUS, can direct the chip under test to perform five different operations, namely functional, debug, reset, serial scan and built-in self-test. The controller consists of control logic, command decode logic, parity logic and three registers, namely transfer register, Command Register and Status Register. The Transfer register and the Status register are converted into a test pattern generator (TPG) and a parallel signature analyzer (PSA) during built-in self-test mode. The overhead of a TMC is 6 I/O pins and about 500 gates. The problem of synchronizing the system functional clocks and the the test bus clock is not considered.

Whetsel [33, 65] proposed a design for the Test Access Port used in the boundary scan architecture. In this design, edge-triggered flip-flops are used as the basic storage elements. Clock inputs of flip-flops are allowed to be gated. The total gate count for this design is about 80 and the pin count is 4. Compared to the design of the controller proposed by Avra, the overhead of this design is small. This design is suitable for most chips. Whetsel also did not address the problem of synchronizing functional and test clocks.

LeBlanc [41] reported on a built-in self-test technique, called LOCST, for chips designed at IBM using level sensitive scan design (LSSD). LOCST utilizes on-chip pseudorandom pattern generation, on-chip signature analysis, boundary scan

and an on-chip monitor as the test controller. The monitor includes a standard maintenance interface with seven dedicated signal lines. The major functions of an monitor are scan string control, error monitoring and reporting, chip configuration control, clock event control, run/stop single cycle and stop on error. The boundary scan chain can be used to test external logic, which cannot be tested by the monitor. The advantages of this technique are low area overhead ( $< 2\%$ ), design independent implementation and effective static testing. The key drawback of using a monitor is the high I/O pin count overhead.

### **1.2.5 Testable Module Design**

Budde [15] presented a board test controller called Testprocessor. Testprocessor can control the test process of a chip through a dedicated test bus. Since a Testprocessor is designed for use on either a printed circuit board or a VLSI chip, its functionality is limited by area constraints. The only data processing unit in a Testprocessor is a fault-secure comparator. Due to its limited data processing capability, diagnostic programs cannot run on a Testprocessor.

The TEA [62] system employs a Maintenance Node to make a board testable. Test Switches are added to a board to increase its testability and controllability. Ambiguity groups are first identified. A Test Switch is then inserted between every pair of Ambiguity Groups. The Maintenance Node uses 10 signal lines to control all the maintenance activities of the board.

Babiak et al. [48] reported on a module BIST scheme using Module Maintenance Nodes, one of which is embedded in every module and is controlled by a Maintenance And Diagnostic System that contains a processor to run both test and diagnosis programs.

### **1.2.6 Interconnect Test Generation**

The IEEE Std. 1149.1 requires that every chip be built with the boundary scan architecture, where each I/O pin is associated with a scan cell. By shifting data

along a chain consisting of these scan cells, the interconnect between chips can be easily tested. This helps subdivide the test problem and leads to increased fault isolation capability.

Kautz [37] derived a minimal test set for detecting opens and shorts in a wiring network. The number of test required is  $p - 1 + \lceil \log_2 q \rceil$ , where  $p$  is the number of terminals in the largest interconnect net in the network, and  $q$  is the total number of nets. This result has become the foundation of later work on interconnect testing.

Wagner [64] presented a method for testing interconnections using boundary scan registers. Both stuck-at faults and shorts are considered. Stuck-at faults are tested for free while testing for shorts. By complementing the test vector set, faults can be located. For  $n$  2-point nets,  $2 \times \log_2 \lceil (n + 2) \rceil$  tests are required. The order of the I/O pins in the boundary scan chain must be given. If tristate pins and bi-directional pins are included, additional tests are required.

Hassan et al. [29, 30] extended the minimal test set for interconnect test developed by Wagner [64] to a generalized test set, where information on the order of the I/O pins is not required. He also presented several BIST schemes for interconnect test using the boundary scan architecture. Walking ones and zeroes sequences are proposed as efficient test patterns. Several in-place diagnosis schemes are also presented, including a modifier sequence for area efficient built-in diagnosis.

Jarwala and Yau [34] have developed a comprehensive framework for dealing with the test and diagnosis of interconnect. In addition, a diagonally independent property is identified. They also proposed the C-Test algorithm which can generate a minimal test set for identifying all shorts and opens in a network. Their results are valid only if a net is not involved in both opens and shorts at the same time.

Cheng et al. [16] researched the self-diagnosis property. Constant weight codes are proposed to achieve self-diagnosis. In addition, several optimal adaptive algorithms are proposed for deriving test sets that achieve the self-diagnosis property.

All these results are based on the assumption that a net cannot be involved in both open and short faults. When this assumption is removed, these results are invalid.

### 1.2.7 Test Scheduling

Abadir and Breuer [3] solved the problem of optimizing the execution schedule of a test plan for a single test block. A resource conflict graph is used to indicate the sharing of a resource at different steps of a test plan. They showed that the lower bound on the time delay ( $D$ ) between the initiation of two tests is equal to the chromatic number of the conflict graph. No-operation steps (no-ops) are inserted into a test plan to get an optimal test schedule. The problem of scheduling multiple test blocks was not addressed.

Craig, Kime and Saluja [18] addressed the problem of scheduling multiple test blocks. Unlike Abadir and Breuer's work, they assumed that all test blocks have their  $D$  values equal to one. Assuming all test blocks have the same test length, they constructed an algorithm to minimize the number of concurrent test sets, which are defined as a set of tests that can be executed in the same test session. The same algorithm is then extended to solve the scheduling problem for test blocks with unequal test lengths. The problem of test blocks with  $D$  value greater than 1 was not dealt with.

Sayah and Kime [58] dealt with the problem of scheduling multiple test blocks with a complex test plan. Unlike previous work, which considers only a single aspect of test parallelism, a broader consideration including both time and space parallelism is taken. Based on a resource allocation graph and a so-called *delta graph*, they found a good heuristic algorithm for the scheduling of tests.

The work discussed above does not take into consideration the control structure that implements the scheduling process. Also, the problem of scheduling test at the module and subsystem level has not yet been addressed.



### 1.3 The Design Approach

A systematic design technique, called the *hierarchically testable and maintainable* (HTM) design methodology, is addressed in this work. Adopting the HTM methodology at all levels of the physical hierarchy of a design, i.e. chip, module, subsystem and system, will increase system availability and significantly lower hardware life cycle costs. A system designed with such a methodology is called an HTM system.

A design-for-test tool called BOLD is presented in this work. BOLD is a tool that supports both the hardware and software design of an HTM system. In the hardware support, a set of test controllers to execute the test process of different testable units are provided. In the software counterpart, a set of high level languages to describe the test aspects of these units are provided. Tools are also provided such that these descriptions can be translated into executable code for the test controller. BOLD also provide the necessary tool for automatically testing interconnects among different units.

In designing an HTM system, there exists tradeoffs between the test time and the hardware complexity of test controllers. From a designer's point of view, the goals are to reduce both the test time and hardware overhead. These are two conflicting requirements since in general more hardware has to be added to the test controllers to reduce test time. Using the capability of automatic synthesis of test programs provided by BOLD, the overall test time of an HTM system can be quickly predicted. Hence the whole design space is quickly explored in choosing a feasible solution.

### 1.4 The Hierarchical Test Methodology

A hierarchy of test controllers is employed in a system designed with the hierarchical test methodology. These controllers are distributed among each level of the physical hierarchy. In such a system, which is referred to as an HTM system, each VLSI chip has an on-chip test and maintenance controller (CMC); each module (or board) has a module test and maintenance controller (MMC); each subsystem has a subsystem

test and maintenance controller (SuMP); and each system has a system test and maintenance controller (SMP). These controllers participate in all system test and maintenance activities, and communicate via test busses.

Figure 1.1 shows part of the test hierarchy for four levels of test hierarchy. Different busses may be used for communication between different levels. The SMP communicates with SuMPs through a Level 2 bus (L2-bus); a SuMP communicates with MMCs through a Level 1 bus (L1-bus); and an MMC communicates with CMCs through a Level 0 bus (L0-bus). The IEEE 1149.1 [33] boundary scan bus is used as the L0-bus throughout this work.

**Pros and cons** Advantages of the hierarchical test methodology include the following:

1. Lower level tests remain valid at higher levels of the design hierarchy.
2. Interoperability at each level of subassembly due to standardized interfaces.
3. Increased fault isolation capability provided by boundary scan.
4. Reduced testability and maintainability (T&M) design time.
5. Reduced maintenance time due to consistent T&M techniques.
6. Increased system availability due to reduced maintenance time.
7. Increased reliability due to increased test effectiveness.
8. Reduced overall test and maintenance cost.

The disadvantages of such a hierarchical test methodology are the controller hardware overhead and also that components with standardized interface are required.

The following example shows how a circuit can be tested when the HTM methodology is used.

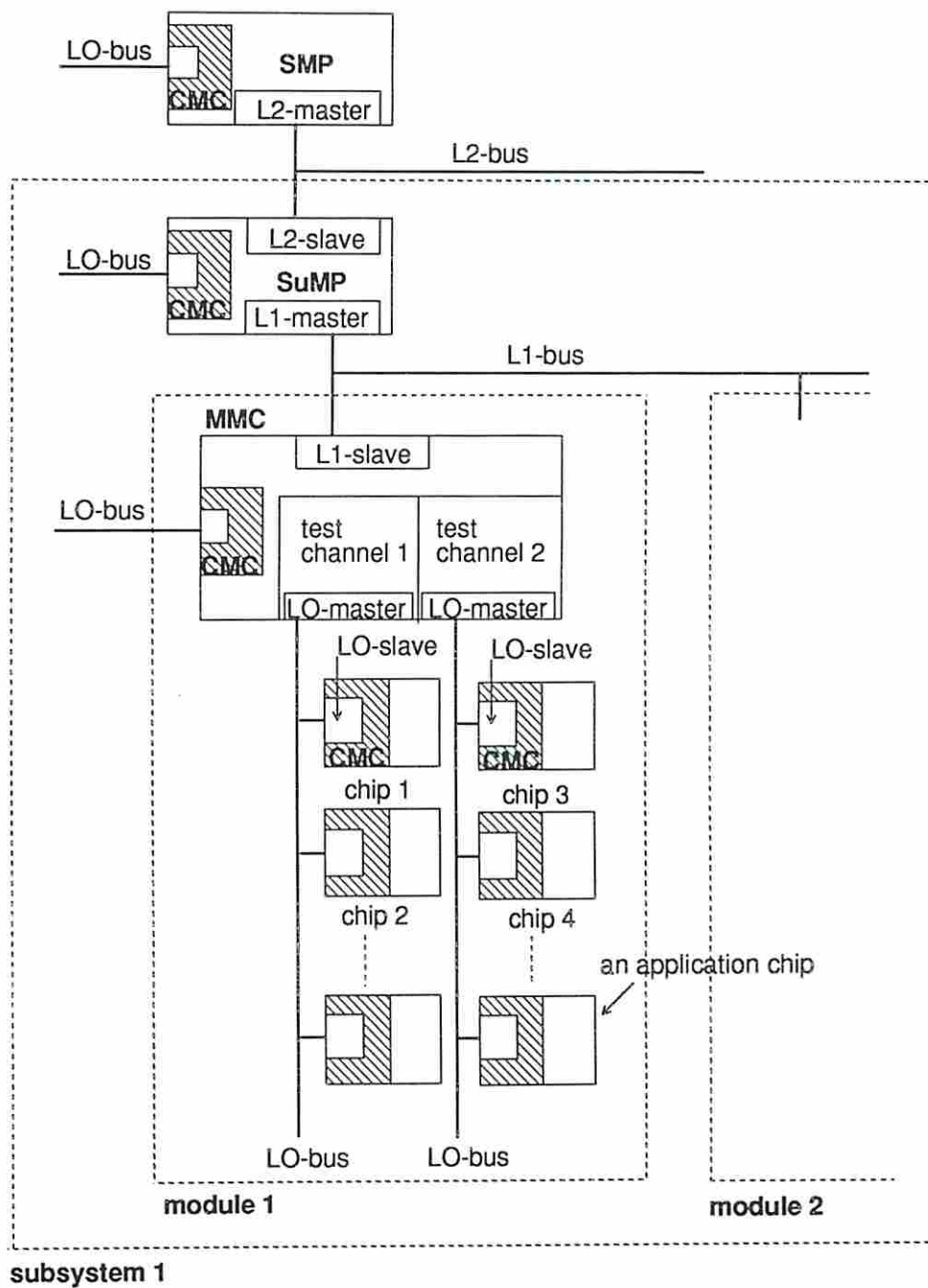


Figure 1.1: The architecture of a HTM system.

## 1.5 Test Controllers for a Circuit

There are numerous testable design methodologies (TDMs) that can be used to make a circuit testable. In selecting a TDM, criteria such as total test time, area overhead, and circuit performance degradation must be considered. Tradeoffs often need to be made so that design constraints and goals are satisfied. It is possible to automate such a selecting process by employing an expert system [2, 69]. Once a TDM for a circuit is selected, the associated BIT structure and test plan can be derived.

To execute such a test plan a test controller is required. The test controller configures the circuit for testing and controls the execution of the test. In addition, it may also generate test data and analyze test results. Not all aspects of the controller need be on-chip. Some of the possible tradeoffs are considered in this example.

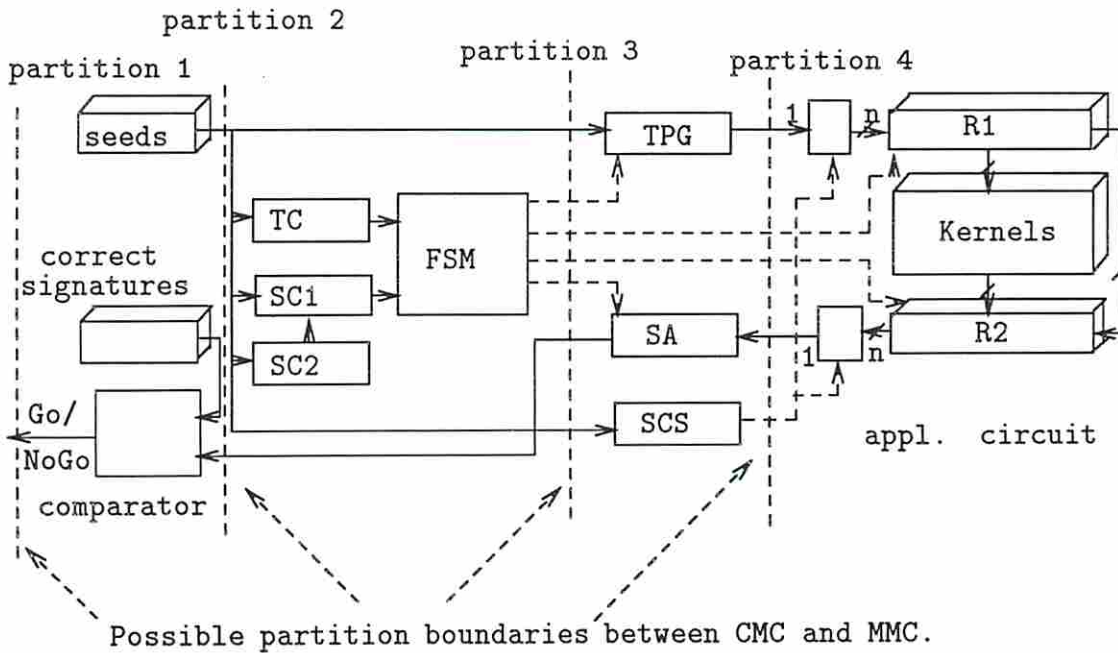


Figure 1.2: Partitioning the test controller between the CMC and MMC.

**Partitioning the test controller:** Suppose the BIT structure of an application circuit consists of one or more scan loops (see Figure 1.2). The test plan for a loop is as follows: (1) shift in a test vector; (2) latch the response data into the scan



register; and (3) shift the test results out of the loop while shifting in a new test vector. This process is repeated  $t$  times, where  $t$  is the number of test vectors.

Assume test vectors are to be generated by a serial test pattern generator (TPG) and the results compressed by a serial signature analyzer (SA). A complete test controller must have the following hardware facilities (or test resources) to carry out this test process: a TPG to provide the test vectors, a SA to compress the test results, a counter TC to keep track of number of the test vectors, one counter SC1 and a register SC2 to keep track of the number of shifts for each vector, seeds vectors for various registers such as the TPG and SA, a stack containing correct signatures, a signature comparator, a register SCS for scan chain selection, and a finite state machine to control the test process.

Several different configurations for these test resources are possible. Figure 1.2 shows four possible partitions of these resources between on-chip and off-chip controllers, i.e., CMC and MMC. Once partitioned along some boundary, an interface or bus is required to connect the two partitions. For example, a boundary scan bus is used for an MMC to communicate with a CMC. The CMC must have a slave interface and the MMC must have a master interface.

*Partition 1* puts all resources into the CMC. Such a CMC is capable of executing a test process completely on its own, once initiated by the MMC. After  $t$  test vectors have been applied to the kernel, the signature is compared with the correct one. The CMC then reports only the Go/NoGo status to the MMC through the boundary scan bus.

*Partition 2* incorporates the seeds, correct signatures and the comparator into the MMC while leaving the rest of the resources in the CMC. The MMC first loads the seeds into the TPG, SA, SCS, TC, SC1 and SC2 registers in the CMC via the boundary scan bus. The CMC then generates and applies test vectors to the test kernel and the test results are compressed in the SA. After  $t$  test vectors have been applied, the CMC requests that the MMC read the signature in the SA via the boundary scan bus. The MMC then compares the signature with the correct one and determines the health status of the kernel by generating a Go/NoGo indication.

*Partition 3* keeps the TPG and the SA in the CMC while putting the rest of the test resources in the MMC. The MMC must provide control signals for the BIT structures, the TPG and the SA. All control signals must be derived from the test bus directly during the test process. The MMC keeps track of the number of shifts for each test vector and the total number of the test vectors that have been applied to the kernel. After  $t$  vectors have been applied, the MMC then reads the signature out of the CMC. A comparison is made against the correct signature to determine the health status of the kernel.

*Partition 4* puts all the before mentioned test resources into the MMC. The CMC is simply a boundary scan bus slave interface. The complexity for this MMC is maximal and corresponds to the concept of a test channel, which will be described later. The MMC must provide test vectors and collect test results through the test bus. Control signals for the on-chip BIT structures are also provided by the MMC through the boundary scan bus.

An MMC can control several CMCs through a bus. The hardware partitions may be any of the four mentioned above. In fact they can be different for each chip. To be able to control any CMC, the MMC must not only have the resources dictated by partition 4, but even additional capabilities in order to control BIT configuration other than just scan chains using random data. Test resources in the MMC can be shared by the CMCs, while resources in a CMC cannot be shared by other CMCs. Obviously, the more test resources in the CMC the higher degree of test execution parallelism that can be achieved, thus leading to a reduction in total test time.

## 1.6 Thesis Outline

This thesis is organized as follows. The designs of test controllers are presented first. On-chip test controllers are described in chapter 2, followed in chapter 3 by a test controller for testable modules. The generation of test programs is described in chapter 4, where test description languages are presented along with various synthesis tools. The results of test program synthesis for several examples are also

shown. Chapter 5 deals with how the synthesis technique facilitates the minimization of test controller complexity. The issue of testing interconnect among different hardware units is described in chapters 6 and 7. In chapter 6, a new fault model is presented along with theorems and algorithms for deriving test sets to identify all faults. In chapter 7, the problem of actually applying a test set to test interconnect is investigated. Theorems and algorithms are provided so that a schedule can be constructed to achieve minimal test time. Conclusions and future work are given in chapter 8.

## Chapter 2

### Controllers for Testable Chips

#### 2.1 The Model

In this chapter the design of one or more on-chip test controllers (CMC) for testable chips are presented. A testable chip is assumed to have some DFT or BIST features which can be controlled through a boundary scan bus. It consists of a CMC and an application circuit. The CMC in turn contains a bus interface, called L0-slave, and a Built-In Test (BIT) controller. The IEEE Std. 1149.1 boundary scan bus is used as the test bus in this work. A detailed description of this standard can be found in [33]. For convenience, a brief introduction of the boundary scan architecture is given below.

##### 2.1.1 Boundary Scan Architecture

The boundary scan technique requires the inclusion of a scan cell for each I/O pin of a chip. A typical boundary scan cell is shown in Figure 2.1. A boundary scan register is formed by cascading all the boundary scan cells. During normal operation, the scan cells are transparent to the operation of the chip except that a multiplexer delay is added to each I/O pin. During test operation, the logic values of these I/O pins can be captured into the first flip-flop (Q1) and then shifted out for observation; meanwhile, new values can be shifted in and transferred to the output

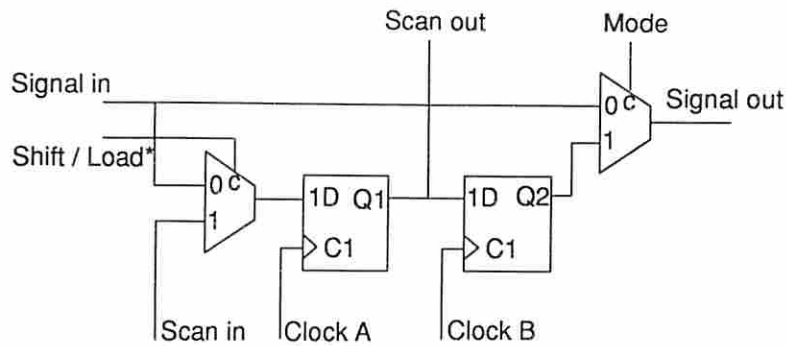


Figure 2.1: A typical boundary scan cell.

of these cells. A boundary scan module contains chips that have this boundary scan architecture. A scan path can be formed by cascading all the boundary scan registers (see Figure 2.2). This scan path can be used in two ways: (1) to allow the interconnects between the various chips to be tested, and (2) to allow the chip on the module to be tested.

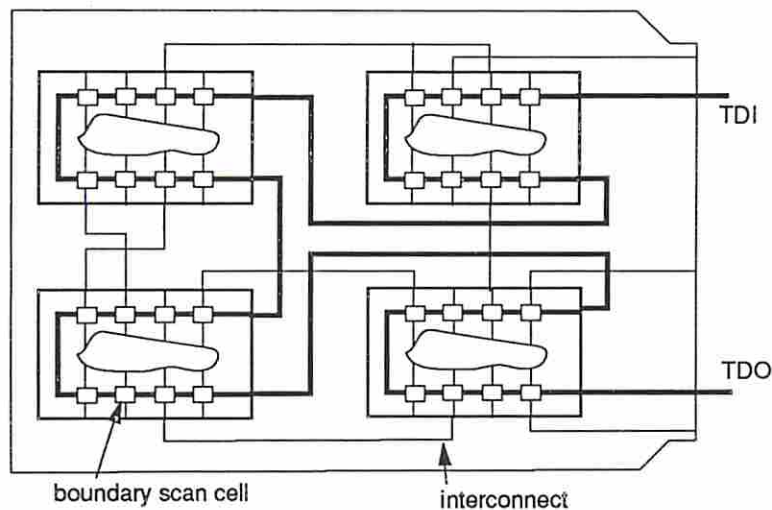


Figure 2.2: A module having a boundary scan path.

The boundary scan bus consists of at least four signal lines, namely TDI, TDO, TMS, and TCK. A fifth signal line TRST\*, which is not shown here, is optional. The TCK line provides the clock for the test logic in the L0-slave. The logic value



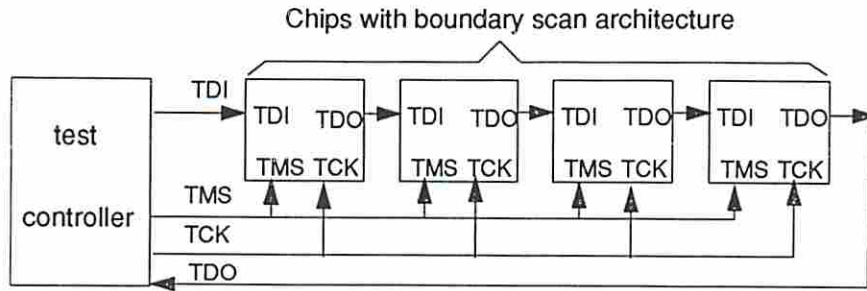


Figure 2.3: Test bus architecture of a module.

of the TMS line is decoded by an on-chip test controller to control test operations. The TDI line provides serial instruction and data to be received by the test logic of the chip. The TDO line is the serial output for test instruction and data from the test logic of the chip. One test bus architecture is shown in Figure 2.3.

The boundary scan architecture of a chip is shown in Figure 2.4. The shaded area labeled as application circuit is the circuit designed with a predefined BIT methodology. During the test mode, many scan registers are formed in the application circuit. These scan registers can be used to control the test process of the circuit. The unshaded area consists of the Test Access Port (TAP), which can also be referred to as the L0-bus slave, and is required in a chip with the boundary scan architecture. The TAP consists of a TAP controller, an instruction register (IR), a boundary scan register, a one-bit bypass register, an optional device identification register (ID), and multiplexers. The hatched area labeled as the *BIT controller* contains additional (optional) test facilities for controlling the test process of the application circuit. The CMC consists of the TAP and the BIT controller.

The state transition diagram of the TAP controller is shown in Figure 2.5. The states are represented by the values of the flip-flops used in the TAP controller. The state transition is controlled by the logic value of the TMS line. Each state has two possible next states, designated by the two outgoing directed edges. The state transition follows the edge with label 1 if the current value of the TMS line is 1, otherwise the edge with label 0 is followed. The *Reset* state is entered whenever

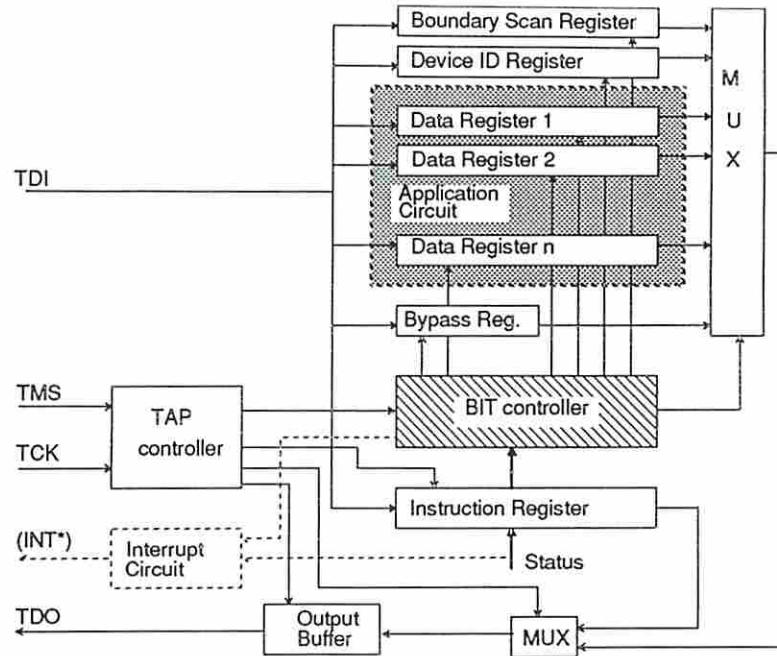


Figure 2.4: The model of a chip with boundary scan architecture.

the TAP controller is reset, which can occur when the system power-on-reset is activated or the TMS line is held high for more than 5 consecutive TCK clock cycles. The Run-Test-Idle state is entered when executing the self-test activities, or when the chip is in test mode with no ongoing test activity.

Two major branches are used to transmit instructions and data. When transmitting instructions, the number of activations of the state ShiftIR equals the number of instruction bits sent. A new instruction is loaded into the IR register when the UpdateIR state is activated. The contents of the IR determines the operation of the on-chip test controller. One major function of the data in the IR is to select a data register for scanning. When transmitting data, both the CaptureDR and UpdateDR states are activated exactly once for each transmission. The number of activations of the ShiftDR state equals the number of data bits sent to the selected DR. By properly driving a TAP controller, a module level test controller can send/receive both instructions and data to/from a chip. This information controls the test execution of the chip.

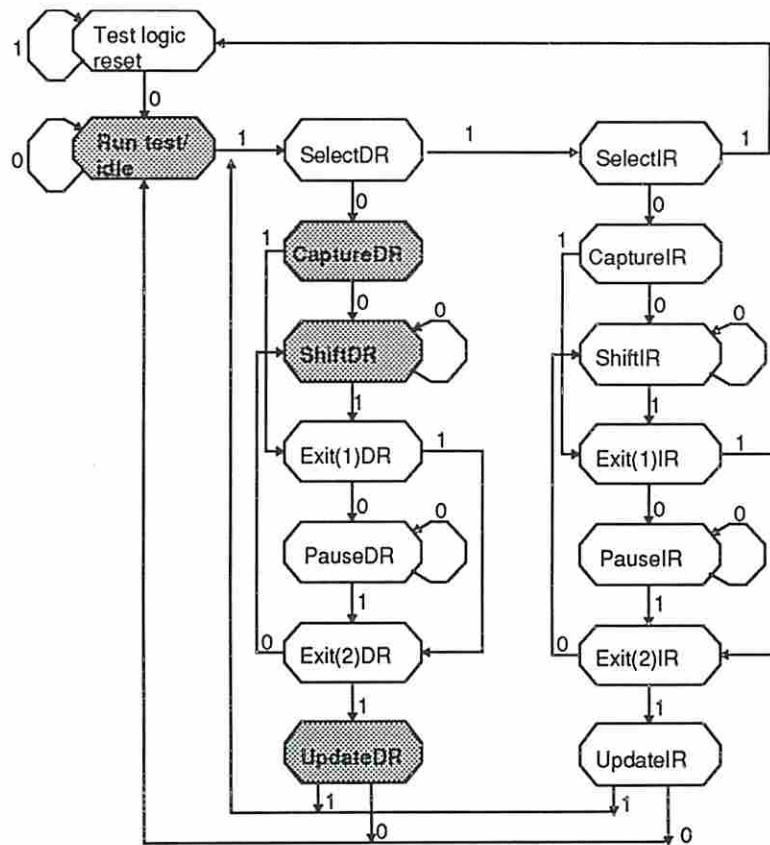


Figure 2.5: The boundary scan bus state transition diagram.



During the test mode, all test control signals are controlled by the BIT controller, which in turn is controlled by the contents of the IR and the current state of the TAP controller. Signals generated from the TAP controller include RunTest, Capture, Shift and Update, and are active during the state Run-Test-Idle, CaptureDR, ShiftDR, and UpdateDR, respectively. Only one of these four signals can be active at a time. Also, the sequence of activation of these signals must be consistent with the state transition diagram described previously.

When a register is selected for scan, the control signals must be designed in such a way that the logic values of the inputs to this register are captured when the TAP state is in CaptureDR. The selected register is shifted whenever the TAP state is in ShiftDR. Throughout this work it is assumed that the output of an addressable register is updated only when the TAP state is ShiftDR, i.e., an implicit HOLD mode is assumed for all addressable registers. Figure 2.6 shows a general model for an addressable register.

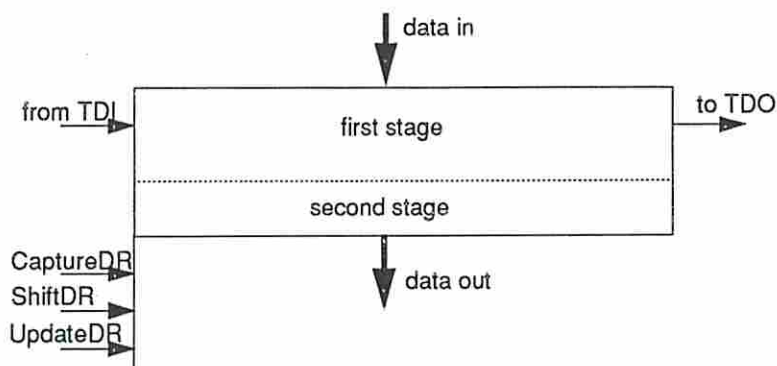


Figure 2.6: Control model for an addressable register.

The control signals of a testable chip during the test mode are shown in Figure 2.7. The signals  $C1$ ,  $C2$ ,  $\dots$ ,  $Cn$  control the test execution of the application circuit, which has been built using some BIT methodologies. The signals  $IR1$ ,  $IR2$ ,  $\dots$ ,  $IRm$  are the output of the IR. Registers in the application circuit must hold their data when the TAP controller is in certain states, such as Exit(1)DR, PauseDR, SelectDR. This can be done using an explicit hold control signal or by disabling the clock.

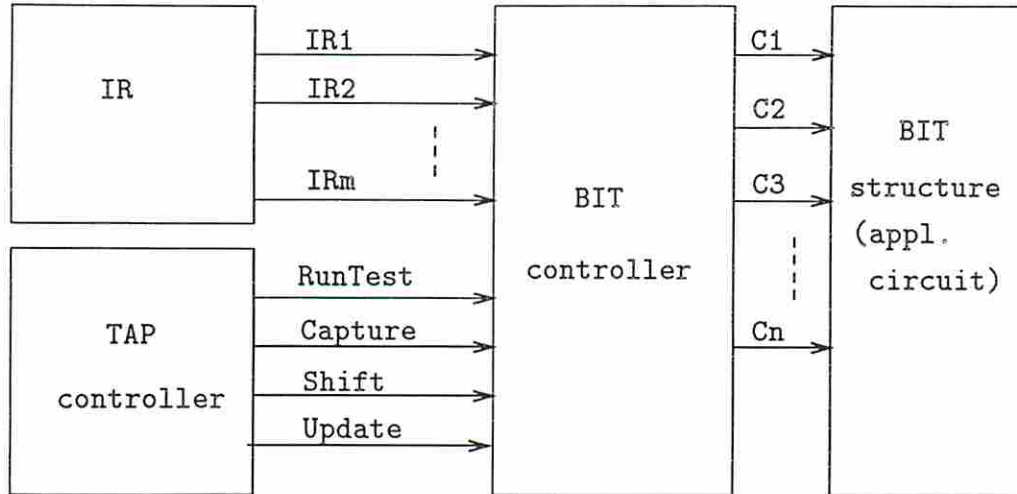


Figure 2.7: The control signals during test mode.

Based on the dependence of the test bus, BIT controllers can be grouped into two categories; *bus-dependent* and *autonomous* BIT controllers. During the testing of the application circuit, a bus-dependent BIT controller uses the lines *Capture*, *Shift*, *Update* and *RunTest*; while an autonomous BIT controller uses only the *RunTest* line. In the other words the operation of a bus-dependent BIT controller depends on the state transitions of the test bus during the entire test execution process. The operation of an autonomous BIT controller is independent of the bus state transitions once it has been properly initiated. In general, an autonomous controller has a higher hardware complexity than a bus-dependent controller.

## 2.2 Bus-Dependent BIT Controllers

In this section, the design of BIT controllers for various test structures are presented. These controllers use the state of the TAP controller as a source of the control signals. When testing a kernel, the state transitions of the BIT controller must follows the control graph of the kernel. A control graph that can be used to test a LSSD kernel [21] is shown in Figure 2.8(b).

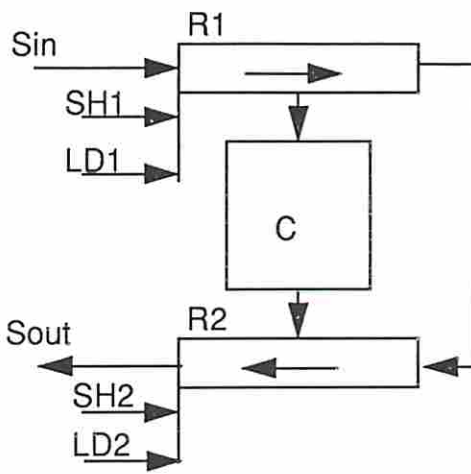
In a control graph, a node  $S_i$  represents a control state of a BIT controller, which is associated with a signal  $FS_i$  (not shown in the graph) that is active in

this state; an arc represents a state transition; and the label of an arc represents the number of iterations associated with that transition. A rectangle decision box determines the state transitions. The arc with label 1 in the box is taken when this box is first entered. The arc with the next highest number will be taken only when a sufficient number of iterations has been taken in the currently selected arc. For example, in Figure 2.8(b), once state S2 is entered, the second arc (to state S1) is taken after the first arc (self-loop) has been taken  $s$  times. Similarly, the third arc (to exit) is taken after the second arc has been taken  $t$  times. For those states that have only one possible next state, no decision box is needed. Thus the decision box corresponds to a nested loop of the form

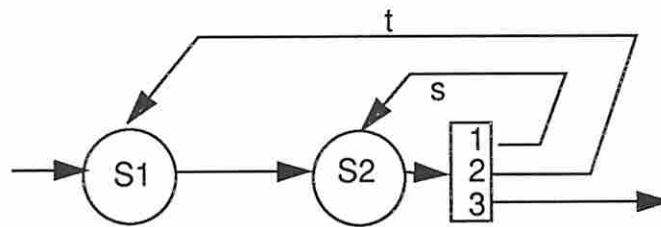
```
do once
  do for j=1,...,t
    do for i=1,...,s.
```

The control signals  $C_1, \dots, C_n$  are decoded from the signals  $FS_i, i=1, \dots, k$ . Thus the implementation of a bus-dependent BIT controller deals with the activation of the signals  $FS_1, \dots, FS_k$ , in the sequence as described by the control graph. A general model of the bus-dependent BIT controller is shown in Figure 2.9. The BIT controller consists of two combinational decoders and an optional finite state machine FSM. The decoder dec2, which generates control signals  $C_1, \dots, C_n$  from the signals  $FS_1, \dots, FS_k$ , consists of a set of OR gates. The decoder dec1 generates the signals  $FS_1, \dots, FS_k$  from three sources, namely the contents of the IR, the TAP controller state signals and the output  $PH_1, \dots, PH_p$  of the finite state machine FSM. Note that the finite state machine is not needed if the controller can be implemented using a combinational circuit.

The finite state machine is implemented as a programmable counter, which, when enabled, can count from 1 to  $c$  ( $c \leq p$ ) repeatedly. The signal  $PH_i$  is active only when the counter value is  $i$ , where  $1 \leq i \leq c$ . Thus the signals on  $PH_1, \dots, PH_c$  form a one-hot code, and these values are generated repeatedly as long as the FSM is enabled. When the finite state machine is disabled, the value of the counter is 0 and the outputs of the machine are all disabled. The value  $c$  is determined by a register in the machine which can be modified by shifting a new value into it.



(a)



(b)

Figure 2.8: A LSSD kernel; (a) control signals, (b) control graph.



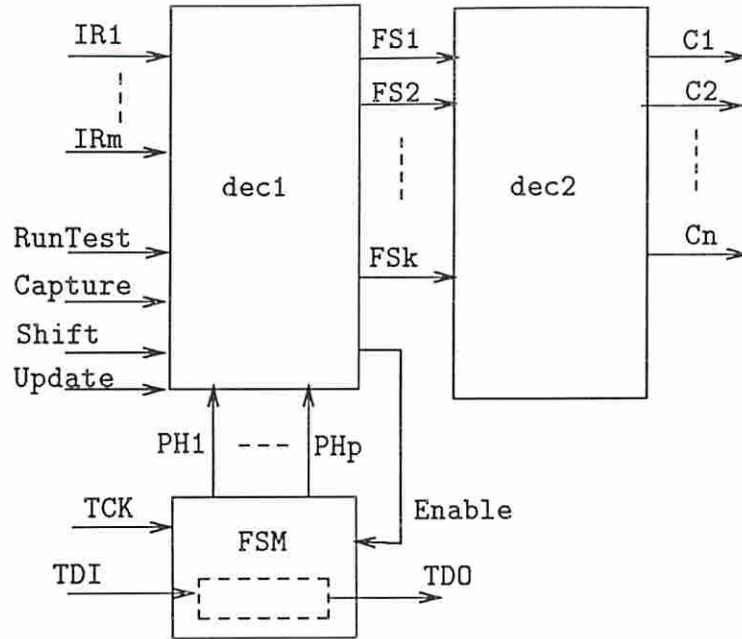


Figure 2.9: A general model for the bus-dependent BIT controller.

Note that this register can be part of the instruction register or an addressable data register.

### 2.2.1 BIT Controller for a LSSD Kernel

If a kernel is made testable using the LSSD technique [21], the BIT structure consists of two registers (R1, R2) and a combinational circuit C (see Figure 2.8(a)). (The registers R1 and R2 can be combined into a single register). These two registers form a scan register that is selected if the contents of the IR is 011 (denoted as [IR=011]). During the test mode, the control signals of the LSSD kernel are LD1, LD2, SH1, SH2. The signals LD1, LD2, which control the parallel loading of new data into registers R1, R2, respectively, are activated while in state S1. The signals SH1, SH2, which control the shifting of data along registers R1 and R2, are activated in state S2. When none of these signals are active, both registers R1 and R2 retain their values, i.e., remain in the HOLD mode. To test the LSSD kernel properly, the control signals should be activated according to the control graph shown in Figure 2.8(b).



To test the LSSD kernel properly, it is necessary to derive the control signals as follows.

$$C1 = LD1 = LD2 = FS1;$$

$$C2 = SH1 = SH2 = FS2.$$

From the model described in Figure 2.7, it is clear that a circuit that implements the following functions can be used as a BIT controller.

$$FS1 = \text{Capture} * [IR=011];$$

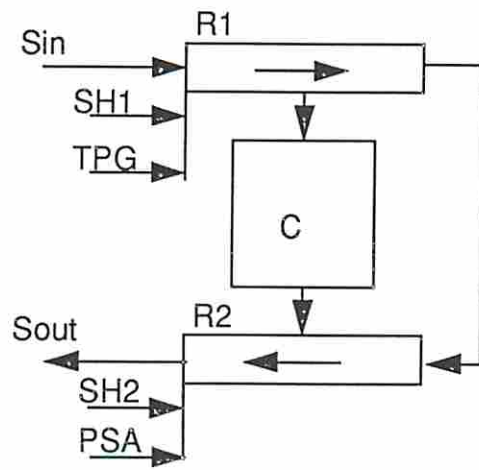
$$FS2 = \text{Shift} * [IR=011].$$

When such a BIT controller is used, an external controller can test the LSSD kernel by first loading the IR with the proper value (011 in this case) and driving the TAP to the states CaptureDR and ShiftDR, which in turn activate FS1 and FS2 according to the control graph. Therefore, the control signals LD1, LD2, SH1 and SH2 are properly activated and the LSSD kernel is tested. Note that the external controller must have at least two counters to keep track of the values of  $t$  and  $s$  required in the control graph.

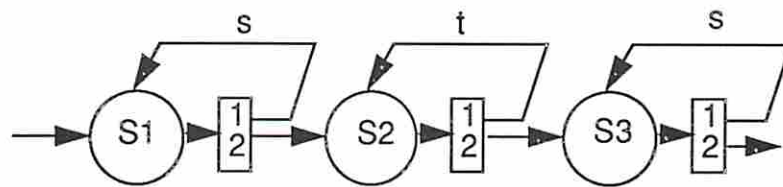
The mapping between the TAP controller states and the test control of the application circuit may not be obvious in some cases. An algorithm that facilitates this mapping properly will be given later in this chapter.

## 2.2.2 BIT Controller for a BILBO Kernel

In the case of a BILBO kernel [38] (see Figure 2.10(a)), the BIT structure consists of two BILBO registers (R1, R2). During the test mode, the control signals of R1 are TPG and SH1, while those of register R2 are PSA and SH2. These two registers form a scan register that can be selected when the value in the instruction register is 101, that is [IR=101]. To test a BILBO kernel properly, the control signals must be activated as illustrated in the control graph in Figure 2.10(b). The signals SH1 and SH2, which control the shift operation in registers R1 and R2, respectively, are both active in states S1 and S3. When the signal TPG is active,



(a)



(b)

Figure 2.10: A BILBO kernel: (a) control signals, (b) control graph.

register R1 acts as a test pattern generator. When the signal PSA is active, register R2 functions as a parallel signature analyzer. Both TPG and PSA are active in state S2. Note that to correctly execute a test according to the control graph, different instructions must be used in states S1 and S2. One reason for this is that in going from the Reset state to the ShiftIR state in the TAP control, one enters the RunTest state for at least one clock cycle. According to the control graph, it is clear that the decoder dec2 should be implemented as follows.

```
C1 = TPG = PSA = FS2;
C2 = SH1 = SH2 = FS1 + FS3.
```

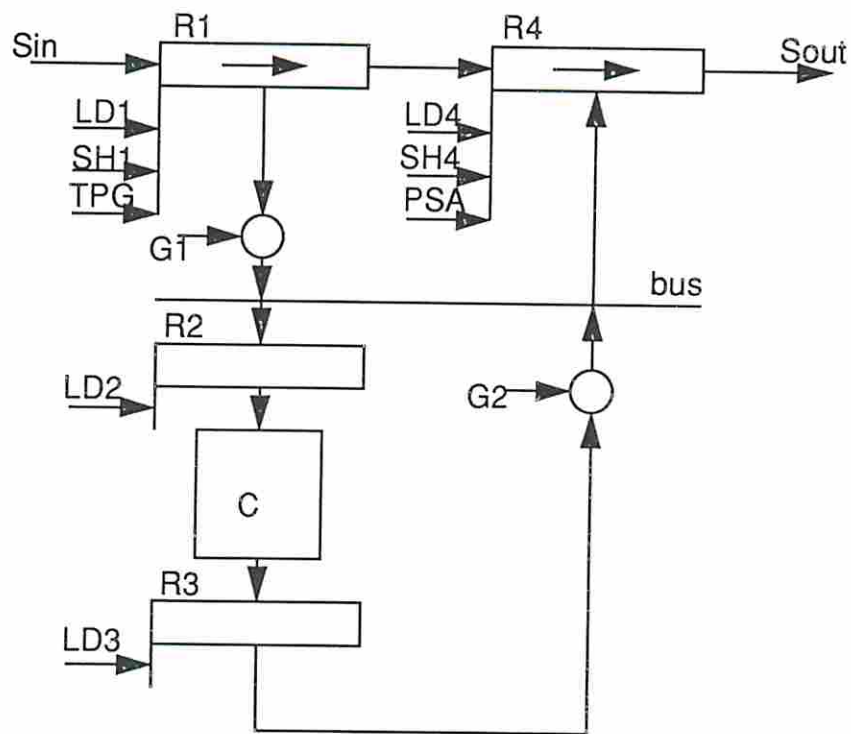
Using the control model shown in Figure 2.7, it is clear that the decoder dec1 should be implemented as follows.

```
FS2 = RunTest * [IR=011];
FS1 = FS3 = Shift * [IR=101].
```

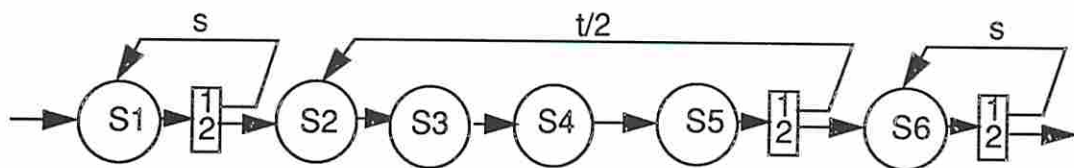
Note that in the above two examples, the BIT controller is simply a combinational circuit consists of two decoders. This is because the mapping mechanism is simple. If the mapping scheme is very complex, the algorithm presented in section 2.2.4 can be used to construct a BIT controller.

### 2.2.3 BIT Controller for a Complex Kernel

Consider the complex kernel in Figure 2.11(a), where the BIT structure consists of four registers (R1, R2, R3, R4), a combinational circuit C and a bus with its associated controls. During the test mode, the control signals are: LD1, SH1, TPG1, LD2, LD3, LD4, SH4, PSA, G1 and G2. A test vector is generated when the TPG signal is active. The test vector is transferred to the register R2 and then applied to C by activating both signals G1 and LD2. The results are then transferred to register R3 by activating the signal LD3. Finally, the results can be compressed in the register R4 if both the signals G2 and PSA are activated. This process is repeated  $t$  times, where  $t$  is the number of vectors required to test C.



(a)



(b)

Figure 2.11: A complex kernel; (a) control signals, (b) control graph.

To reduce the test time, a new test vector can be applied before the completion of the previous vector. However, resource conflicts must be avoided. For example, to avoid any conflict on the bus, the signals G1 and G2 cannot be activated in the same clock phase.

A minimal time test schedule for this kernel is shown in Table 2.1, where nine steps are required to apply four vectors (v1, v2, v3, v4). Each table entry represents a set of control signals that should be active at each time step for applying a test vector.

| time | v1      | v2      | v3      | v4      |
|------|---------|---------|---------|---------|
| 1    | TPG     |         |         |         |
| 2    | LD2, G1 | TPG     |         |         |
| 3    | LD3     | LD2, G1 |         |         |
| 4    | PSA, G2 | LD3     |         |         |
| 5    |         | PSA, G2 | TPG     |         |
| 6    |         |         | LD2, G1 | TPG     |
| 7    |         |         | LD3     | LD2, G1 |
| 8    |         |         | PSA, G2 | LD3     |
| 9    |         |         |         | PSA, G2 |

Table 2.1: Test schedule for the complex kernel.

From the table, one can conclude that the activation of control signals can be classified into four phases. In the first phase, the activated control signals are TPG, G2 and PSA. In the second phase, the activated signals are TPG, LD2 and G1. In the third phase, the activated signals are LD2, G1 and LD3. In the fourth phase, the activated signals are LD3, G2 and PSA. Two vectors are applied for each iteration of these four phases. The control graph that can be used to execute the test schedule is shown in Figure 2.11(b). The BIT controller that implements this control graph can be either a sequential or a combinational circuit. These two approaches are described next.

### Sequential approach

In this approach, the finite state machine FSM is used to derive the required control signals. The control signals FS1 and FS6 are active when the instruction is



IR=[0010] and the bus signal Shift is active. The control signals FS2, FS3, FS4, FS5 are derived from the signals PH1, PH2, PH3, PH4, respectively. The finite state machine is programmed as a counter that repeatedly counts from 1 to 4, thus activating PH1, PH2, PH3 and PH4 in sequence when the instruction is IR=[0111] and the bus state is RunTest. The counting continues until the signal PH4 has been activated  $t/2$  times. This means that the test bus must stay in the RunTest state for exactly  $2t$  clock cycles. The decoder dec1 is implemented as follows:

```

FS1 = FS6 = Shift * [IR=0010];
FS2 = PH1;
FS3 = PH2;
FS4 = PH3;
FS5 = PH4;

```

The decoder dec2 is implemented as follows:

```

SH1 = SH4 = FS1 + FS6;
TPG = FS2 + FS3;
PSA = G2 = FS2 + FS5;
LD2 = G1 = FS3 + FS4;
LD3 = FS4 + FS5;

```

An external test controller can thus execute the test by driving the bus states according to the control graph. Note that counters are required for the external controller to keep track of the values of  $s$  and  $t$ . In this example, the required finite state machine is a simple sequencer, which upon activation, repeats a sequence of steps which equals the number of phases in the control graph.

### Combinational Approach

In this case, the BIT controller is implemented using combinational circuits and the FSM part is not used. Each control signal must be generated by using a separate instruction. For example, the control signals FS2, FS3, FS4 and FS5 can be generated using the instructions IR=[0101], [0110], [0111] and [1000], respectively. In this case, the decoder dec1 is implemented as follows.

```

FS1 = FS6 = Shift * [IR=0010];
FS2 = RunTest * [IR = 0101];
FS3 = RunTest * [IR = 0110];
FS4 = RunTest * [IR = 0111];
FS5 = RunTest * [IR = 1000].

```

The decoder `dec2` is the same for both the sequential and combinational approaches.

An external controller can thus execute the test by driving the bus state according to the control graph. Again, counters are required for the external controller to keep track of the values of  $s$  and  $t$ . Note that a new instruction is required for the activation of each new control signal. Therefore, the test execution time is increased dramatically compared to the sequential approach.

From these two approaches, one can conclude that there is a relation between the test time and the complexity of the BIT controller. In general, the more complex the BIT controller is, the shorter the test time.

## 2.2.4 A Mapping Algorithm

When designing a BIT controller for a single kernel, as shown in the above examples, a mapping algorithm is required to ensure the correctness of the BIT controller. During the test mode, the signals  $C_1, C_2, \dots, C_n$  should be controlled by the BIT controller so that the kernel can be properly tested. The test procedure for the kernel is described in a control graph, which has been previously defined. Each state (or node) of the control graph is associated with a set of control signals, which are the signals that are active in that state. The inputs to the BIT controller is the contents of the IR and the signals directly derived from the bus state, namely `Capture`, `Shift`, `Update` and `RunTest` (see Figure 2.7). The BIT controller outputs the signals  $C_1, C_2, \dots, C_n$ .

The input of this algorithm is a control graph, where each state is associated with a set of control signals that should be active in that state. The output of this algorithm is a set of boolean functions along with a set of instructions for the TAP.

All control signals are represented in terms of the bus signals Capture, Shift, Update, RunTest, the contents of the IR, and possibly the output of the FSM. A state signal ( $FS_i$ ) is associated with the state  $S_i$ . The state signals are used as the intermediate form for generating the control signals  $C_1, C_2, \dots, C_n$ . Once the state signals are generated, all control signals  $C_i$  associated with that state are active when  $FS_i$  is active. If a control signal  $C_i$  is associated with various states ( $S_i, S_j, S_k$ ) then  $C_i = FS_i + FS_j + FS_k$ .

The mapping algorithm is used for the design of a bus-dependent BIT controller. Therefore, it is assumed that one of the four bus state signals, namely RunTest, Shift, Update, Capture, must be used to derive any control signal.

The Mapping Algorithm:

Input: A control graph and a list of control signals associated with each state.

Output:  $C_i$  in terms of  $FS_j$ 's, which is a function of the IR contents, the bus state signals and (optionally) the output of the FSM.

1. Repeat this step for all shifting states.
  - 1.1. Assign a distinct instruction to each scan chain.
  - 1.2. For each state  $S_i$ ,  $FS_i = \text{Shift} * [\text{IR} = \text{assigned instruction}]$ , and mark this state.
2. Repeat this step for all unmarked self-loop states.
  - 2.1. Assign a distinct instruction to each self-loop state.
  - 2.2. For each state  $S_i$ ,  $FS_i = \text{RunTest} * [\text{IR} = \text{assigned instruction}]$ , and mark this state.
3. If the only next state of an unmarked state ( $S_i$ ) is a shifting state, let  $FS_i = \text{Capture} * [\text{IR} = \text{instruction of the shifting state}]$ , and mark this state.
4. If the only previous state of an unmarked state ( $S_i$ ) is a shifting state, let  $FS_i = \text{Update} * [\text{IR} = \text{instruction of the shifting state}]$ , and mark this state.
5. If all neighboring states of an unmarked state ( $S_i$ ) have been marked, let  $FS_i = \text{RunTest} * [\text{IR} = \text{instruction of one of its next states}]$  and mark this state.

6. Do this step for all unmarked states.

6.1. Partition these unmarked states into groups such that for an unmarked state  $S_i$  in group  $j$ , if the next or previous states of  $S_i$  are also unmarked then they are also in group  $j$ , i.e., no neighboring state of a state in group  $j$  is unmarked.

Do step 6.2 for each group.

6.2. If the group contains more than one state, go to step 6.3, else assign a new instruction to this state  $S_i$ , and let  $FS_i = \text{RunTest} * [IR = \text{assigned instruction}]$ . Mark this state.

6.3. Sequential Approach: Assign a new instruction to this group. Let the number of states in this group be  $c$ . Program the FSM such that it counts from 1 to  $c$  when  $\text{Enable} = \text{RunTest} * [IR = \text{the selected instruction}]$  is active. Assign the FSM state signals  $PH_1, \dots, PH_c$  to the appropriate  $FS_i$ 's so that all  $FS_i$ 's in this group are properly activated. Mark all states in this group.

7. Instruction merging:

The control graph can be reduced to a subgraph for a given set of nodes if all other nodes and their associated arcs are removed. Generate a subgraph for each group of nodes formed in step 6. For all subgraphs that are isomorphic and are associated with the same control signals, an instruction can be used for all these groups. The number of assigned instructions can thus be reduced.

8. Generate all control signals  $C_i$  from the  $FS_j$ 's. If a signal is active in several states, say  $S_i$ ,  $S_j$  and  $S_k$ , then  $C_i = FS_i + FS_j + FS_k$ .

Initially, all states of the control graph are unmarked. After the execution of this algorithm, all states are marked and their associated signals are assigned with a boolean function. Since data must be sent and received when shifting a register in the application circuit, the shifting states in the control graph can only be mapped into the bus state  $\text{ShiftDR}$ . In step 2, a self-loop state is mapped into the  $\text{RunTest}$  bus state since this state can be held for many consecutive times without changing the contents of  $IR$ . In step 3, 4 and 5, more intermediate signals



are generated without assigning any extra instruction. In step 6, the state signals for all the unmarked states are generated by using RunTest. The BIT controller can either be a combinational circuit or a sequential circuit. If step 6.3 is entered, a sequencer is created, the BIT controller is a sequential circuit; otherwise it is a combinational circuit. After step 6, all the states have been marked (or processed). Instructions that can be shared among different groups formed in step 6 are found in step 7. The total number of instructions assigned is then further reduced. In step 8 all control signals ( $C_i$ ) are generated from the intermediate state signals ( $FS_i$ ). The correctness of the BIT controller is guaranteed since all control signals  $C_i$  are controlled by the contents of IR and the bus states.

**Example** The Mapping Algorithm is illustrated using the complex kernel shown in Figure 2.11(b). The associated control graph is shown in Figure 2.11(b).

- 1.1. Assign instruction  $i_1$  to the only scan register in this example.
- 1.2. There are two shift states  $S_1$  and  $S_6$ , therefore,  $FS_1=FS_6=Shift*[IR=i_1]$ . Mark states  $S_1$  and  $S_6$ .
2. This step is skipped since there are no unmarked self-loop nodes.
3. Skipped.
4. Skipped.
5. Skipped.
- 6.1. Only one group consisting of states  $S_2$ ,  $S_3$ ,  $S_4$  and  $S_5$  is formed.
- 6.2. Skipped.
- 6.3. (Use Sequential approach) Assign instruction  $i_2$  to this group. The FSM is enabled when  $Enable = RunTest*[IR=i_2]$  is active. The FSM is programmed to keep counting from 1 to 4 when enabled. Let  $FS_2=PH_1$ ,  $FS_3=PH_2$ ,  $FS_4=PH_3$ ,  $FS_5=PH_4$ .
7. No instruction merging can be done. Two instructions are assigned.
8. Assign all control signals.  $TPG = FS_2 + FS_3$ ,  $PSA = FS_2 + FS_5$ ,  $G_2 = FS_2 + FS_5$ ,  $LD_2 = FS_3 + FS_4$ ,  $G_1 = FS_3 + FS_4$ ,  $LD_3 = FS_4 + FS_5$ ,  $SH_1 = FS_1 + FS_6$ ,  $SH_4 = FS_1 + FS_6$ . □



Let the number of scan chains formed by shift registers be  $I_c$ , the number of self-loop states be  $I_s$ , and the number of groups formed in step 6 be  $I_g$ . The total number of instructions assigned in this algorithm is  $I_{seq} = I_c + I_s + I_g$ . Note that the public instructions defined in the standard are not included here.

**Lemma 1** *The number of instructions assigned by the mapping algorithm is minimal.*

Proof:

The number of instructions is the sum of  $I_c$ ,  $I_s$ , and  $I_g$ . The first two numbers are defined by the structure of the control graph. The third number  $I_g$  is minimal since the number of groups derived from step 6.1 is minimal. Thus it is clear that the number of instructions ( $I_{seq}$ ) assigned by the mapping algorithm is minimal.  $\square$

**Combinational Approach:** The BIT controller designed using the mapping algorithm may be a sequential circuit since a finite state machine may be used in step 6.3. However, because of design constraints it might be necessary to implement the BIT controller as a combinational circuit. If this is the case, step 6.3 can be modified as follows.

6.3. Combinational Approach: Assign a new instruction to each states in the group. For state  $S_i$  in this group, let  $FS_i = \text{RunTest} * [IR = \text{assigned instruction}]$ . Mark the state.

Let the number of unmarked states at the beginning of Step 6 be  $I_u$ , then the number of instructions assigned in the combinational approach is  $I_{com} = I_c + I_s + I_u$ .

For a given control graph, it is obvious that  $I_{com} \geq I_{seq}$ . Since loading a new instruction to the IR of a chip also loads new instructions to all chips on the same test ring, i.e., all chips share the same TMS line, the test time is increased significantly if many instructions are required in testing a kernel. Therefore the sequential approach of designing the BIT controller can reduce the test time at the expense of adding a finite state machine to the controller.

## 2.3 Autonomous BIT Controller

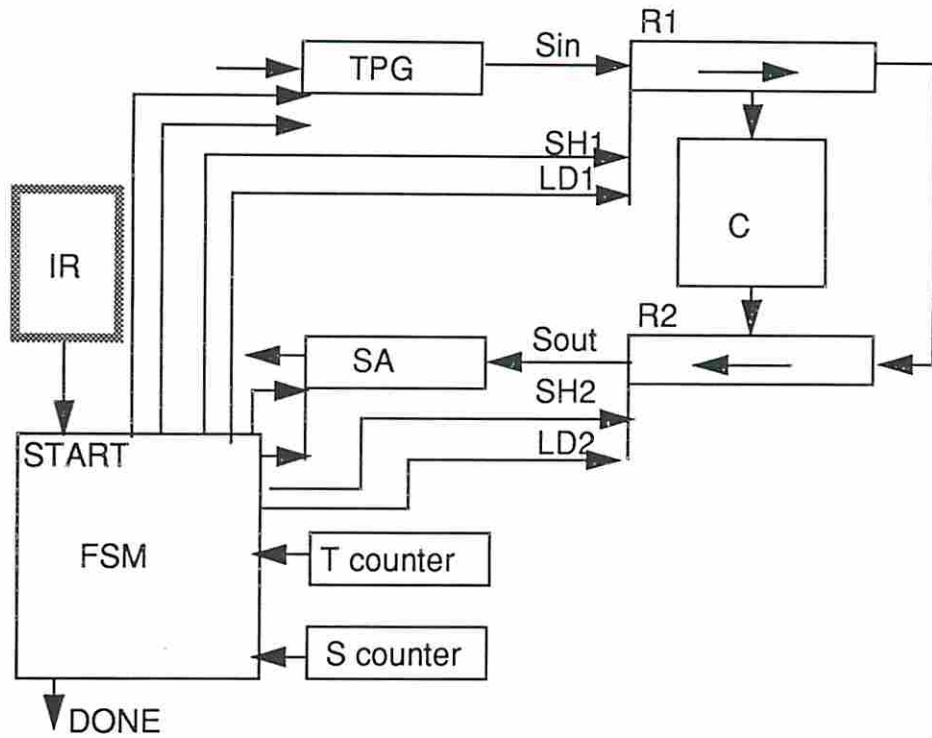
An autonomous controller has all the test control facilities required to execute the test process of the kernel under test. Typical facilities include a test pattern generator, a signature analyzer, a T counter, which is used to keep track of the number of test vectors applied, an S counter, which is used to keep track of the number of bits shifted during a shifting operation, and a finite state machine to control the test sequence. For example, an autonomous BIT controller for testing an LSSD kernel (see Figure 2.8) is shown in Figure 2.12. The correct seeds must be loaded into both TPG and SA before the test. The START signal is generated by loading a special instruction to the IR. Once START is activated, the controller can execute the testing for the LSSD kernel autonomously. Upon completion, a signal DONE is activated. The signature can then be collected.

For the rest of this chapter, it will be assumed that the application circuit contains  $n$  BIT structures, which are all LSSD kernels. An autonomous BIT controller is needed to test these  $n$  BIT structures. Depending on the approach used in testing these kernels (serial or parallel), the BIT controller used is called a Serial BIT controller or a Parallel BIT controller. The design philosophy of these two controllers are addressed next.

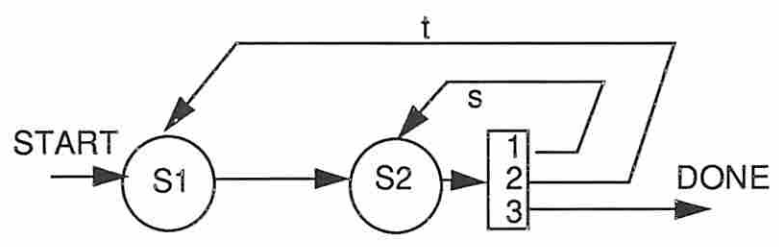
### 2.3.1 Serial BIT Controllers

#### 2.3.1.1 A Hard-Wired Serial BIT Controller

A single BIT controller can be employed to test  $n$  BIT structures in sequence. Such a controller is called a serial BIT controller. A BIT controller that can be used to test  $n$  BIT structures (LSSD kernels) is illustrated in Figure 2.13. A counter SS is used to keep track of which BIT structure is to be activated. This counter requires only  $\lceil \log n \rceil$  bits. Since the same controller is shared among all the BIT structures, the area required for it will not grow linearly with  $n$  and thus, the total area for the controller (not counting the area required for storing the  $t_i$ 's and  $s_i$ 's) will increase only logarithmically with  $n$ . However, the time required for testing is now



(a)



(b)

Figure 2.12: An autonomous BIT controller for a LSSD kernel.

proportional to  $\sum[s_i \times t_i]$ , which may become prohibitively large if the number of BIT structures to be exercised is significant.

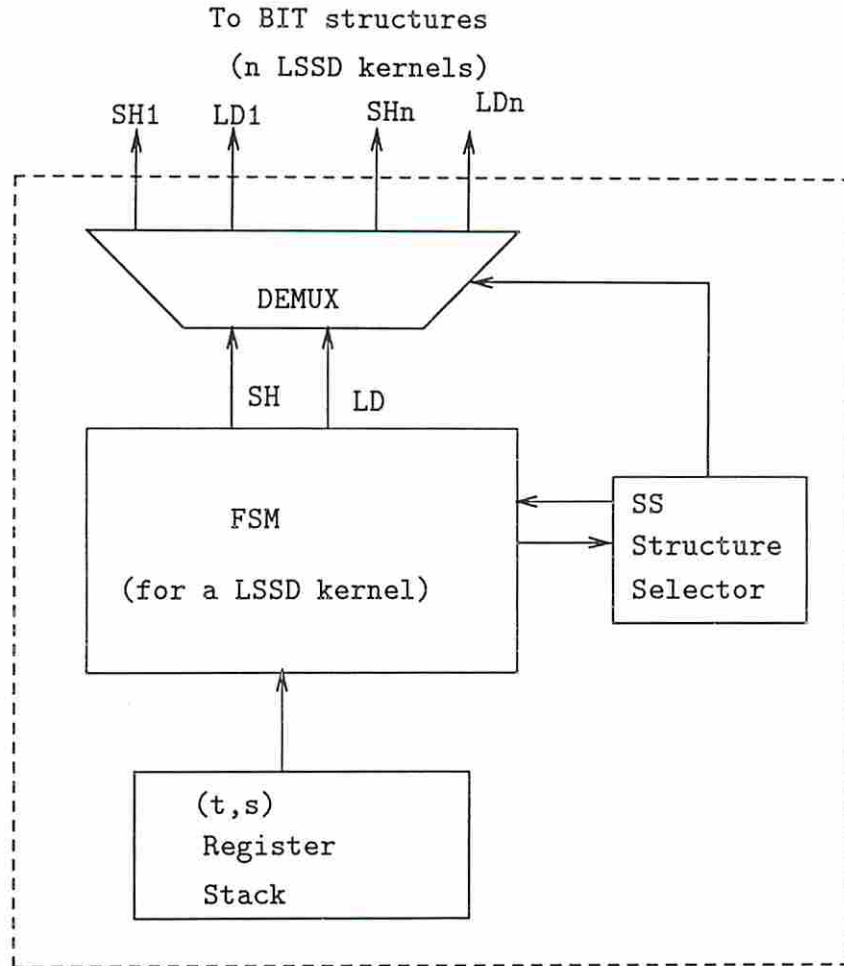


Figure 2.13: Testing many kernels in sequence.

### 2.3.1.2 A Microprogrammed Serial BIT Controller

The general architecture for a microprogrammable test controller, suitable for executing the test for an LSSD kernel, is shown in Figure 2.14. The design, which is first presented in [11], is similar to that of a conventional microcontroller, supplemented with circuitry for keeping track of the loops associated with  $t$  and  $s$  (shown within the dashed rectangle in the figure). The register stack contains constants,



such as  $t$  and  $s$ , which can be loaded into the accumulator register (ACC), decremented (DCR), and stored back into a temporary register in the stack. The logic C determines if the content of the accumulator is zero.

The fields of the instruction register (IR) are - (1) the opcode, (2) the control field, (3) the branch condition address, (4) the miscellaneous field, used for any environment specific control signals, and (5) the address field. These fields may be either horizontally or vertically encoded. A few basic microinstructions are described in Table 2.2.

| field       |              | function                                      |
|-------------|--------------|---|
| 1<br>opcode | 5<br>address |   |
| LOAD        | I            | load ACC from register I of stack             |
| STORE       | I            | load register I of stack with contents of ACC |
| DBNZ        | N            | decrement ACC; if $ACC \neq 0$ , branch to N  |
| NOP         | -            | no operation                                  |

Table 2.2: Microinstruction List for Controller.

For simplicity the branch condition address (field 3) has not been used. Hence for the DBNZ instruction, if  $ACC \neq 0$  then N is forced into ADR1, else ADR1 takes its normal next value which is  $ADR1+1$ .

The microprogram for the control graph of Figure 2.12(b) is shown below. Here  $s$  and  $t$  are permanently stored in stack registers 0 and 2, respectively. The program requires only 7 instructions and 4 different operation codes.

| No. | Opcode | Control | Address | Comments                    |
|-----|--------|---------|---------|-----------------------------|
| 1   | LOAD   | 0       | 0       | $t$ in register 0           |
| 2   | STORE  | 0       | 1       | Test vector count in reg. 1 |
| 3   | LOAD   | 0       | 2       | $s$ in register 2           |
| 4   | DBNZ   | SH      | 4       | Issue SH; loop $s$ times    |
| 5   | NOP    | LD      | 0       | Issue LD signal             |
| 6   | LOAD   | 0       | 1       | Test Vector count to ACC    |
| 7   | DBNZ   | 0       | 2       | Repeat major loop           |



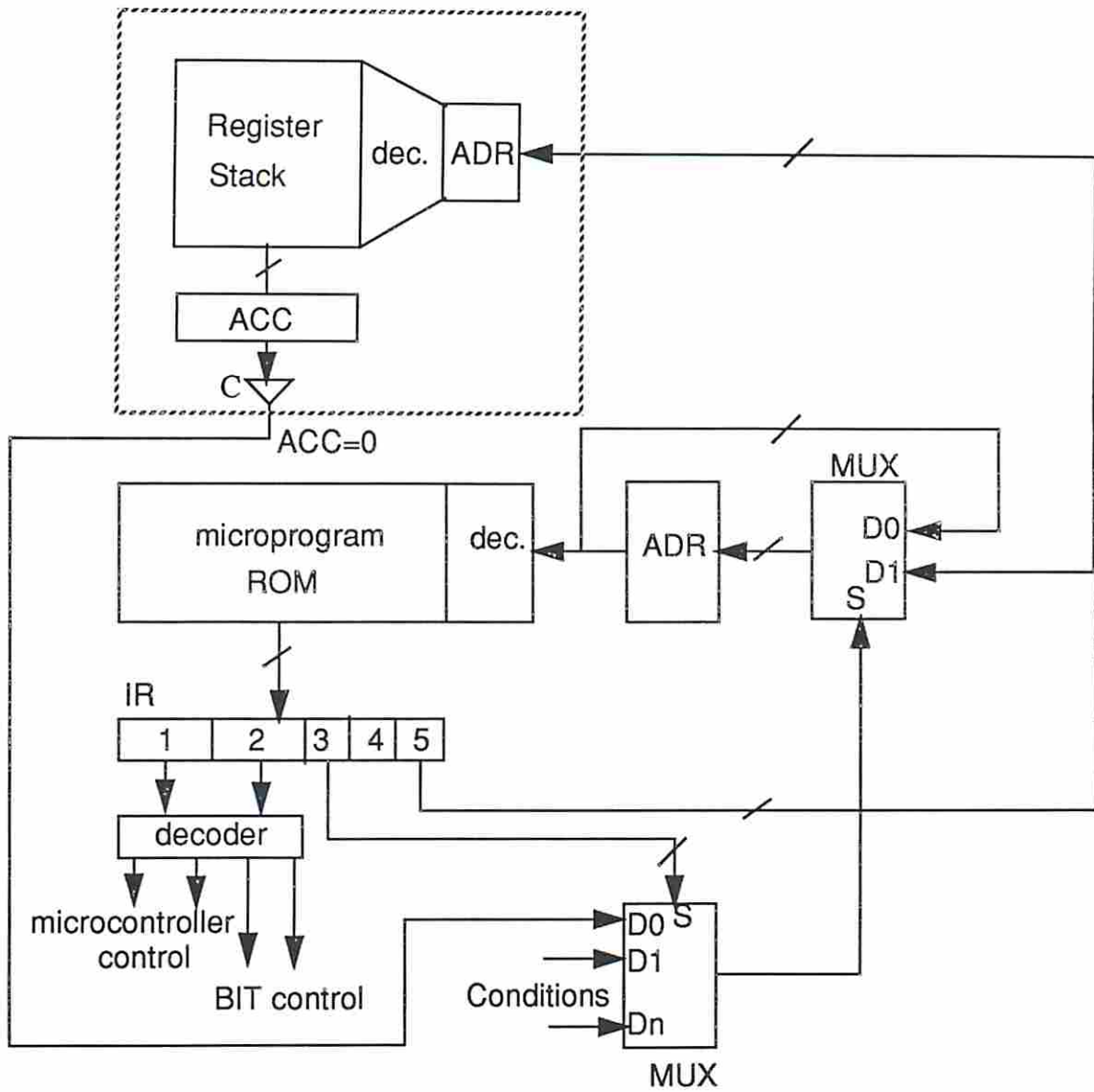


Figure 2.14: Microprogram controller.

If the chip being tested already has a microprogram control unit, then the test control procedure requires very little additional overhead. The latch and shift controls already exist. The data processor portion of the controller may need to be added to the chip if it does not already have an accumulator/ALU structure. An additional advantage of having a microprogrammed control is that one can implement microdiagnostic routines and functional tests. However, if the chip under test does not have a microprogrammable control unit, a hard-wired test controller previously described may be employed.

### 2.3.2 Parallel BIT Controllers

A parallel BIT controller can autonomously test multiple kernels in parallel. The test time is minimal since all kernels are tested simultaneously. Due to the similarity between the kernels, the complexity of the controller can be reduced if the controller is properly designed. The design technique for such controllers is illustrated by using an application circuit consisting of many BIT structures. All these BIT structures are assumed to be LSSD kernels. It is assumed that these kernels are independent, i.e., they do not share any resources such as busses or registers, and hence the test can be executed concurrently. The  $i^{th}$  kernel requires  $t_i$  test vectors and  $s_i$  shifts per vector. Thus  $n$  independent controllers of the type shown in Figure 2.12 can be used for each structure. In this approach, the test time would be  $\max [s_i \times t_i]$ , and the controller area will increase linearly with  $n$ .

Three hard-wired designs which do not have excessive test time as found in the sequential controllers, or require excessive area as may be the case when  $n$  controllers are used are presented next. The first design superimposes the FSMs of individual controllers into one FSM and interleaves the activation of the BIT structures. This design can potentially test all  $n$  BIT structures in the same time as required by  $n$  independent controllers. In addition the area overhead is comparable to that of a sequential controller which exercises the BIT structures one by one. Unfortunately, in this scheme, the time required for testing depends on the problem at hand (i.e., the values of  $t_i$  and  $s_i$ ), and for some pathological cases this design may entail long test times.

Two other designs to be presented rectify this problem by running all BIT structures simultaneously without interleaving. The controller area is reduced by sharing common factors among the  $t_i$ 's and  $s_i$ 's. These designs are referred to as the "Tree-of-Counters" and "Counter-Sharing" controller designs.

It should be remarked at the outset that none of these designs is clearly superior to the others. That is, depending on the BIT structures to be controlled, area constraints, and test time objectives, one controller may be more beneficial than the others.

### 2.3.2.1 Interleaved FSM Controller

The design technique is illustrated with an example consisting of three BIT structures BIT1, BIT2 and BIT3. The extension to  $n$  BIT structures is straightforward. Assume that  $s_1 = 41$ ,  $s_2 = 48$ ,  $s_3 = 62$ ,  $t_1 = 900$ ,  $t_2 = 1000$ , and  $t_3 = 1150$ . To each BIT structure, the LD and SH control signals must be issued at appropriate times. For example, BIT2 must receive 48 consecutive SH pulses, followed by a LD pulse, in order to apply one test vector. It is assumed that if a register is not in the SH or LD mode then it is in the HOLD mode.

The FSMs for the individual BIT structures, which are similar to the FSMs shown in Figure 2.12 with different values of  $s_i$  and  $t_i$ , can be combined into a single FSM as follows.

To interleave the execution of these FSMs, the controller can issue 41 SH pulses to all three BIT structures, then 7 SH pulses to BIT2 and BIT3, followed by 14 SH pulses to BIT3 alone. At this point all the test vectors are loaded into their proper registers and a LD pulse can be issued to the appropriate BIT structures. This process, when repeated 900 times, will apply all the test vectors to BIT1 and the first 900 vectors to BIT2 and BIT3. This cycle, with control signals to BIT1 disabled is repeated another 100 times to finish testing BIT2; and another 150 repetitions with control signals to both BIT1 and BIT2 disabled, will conclude the testing process.

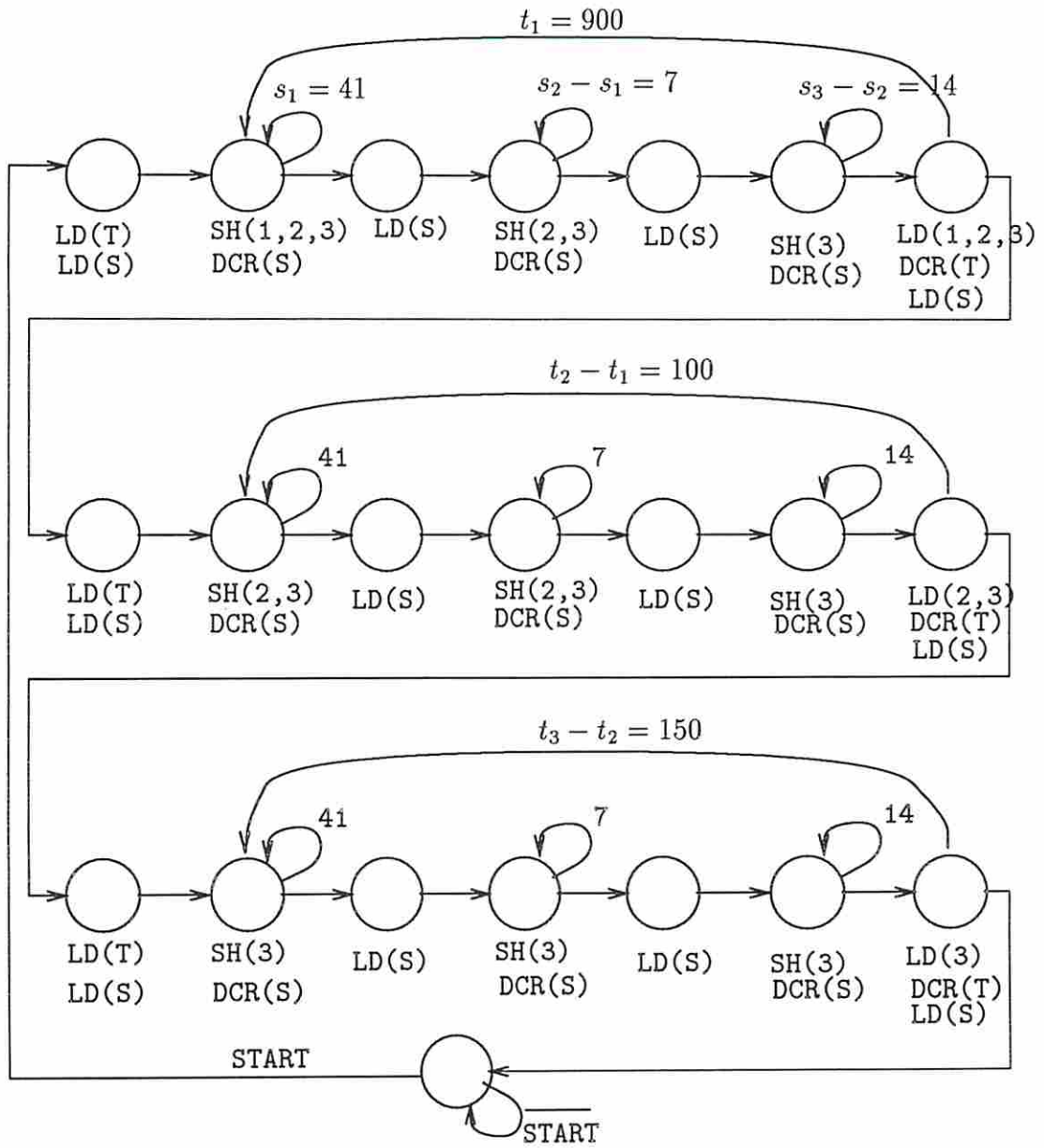


Figure 2.15: Interleaved FSM controller.

The interleaved FSM is shown in Figure 2.15. It consists of three (in general  $n$ ) phases. Each phase completely activates one BIT structure and executes some of the test vectors of other BIT structures which as yet have not completed their test cycle. In this example, the first phase (the top row of states) tests BIT1 and parts of BIT2 and BIT3; the second phase completes the testing of BIT2 and part of BIT3; and the third phase finishes testing BIT3.

A procedure for deriving an interleaved FSM for an arbitrary value of  $n$  is presented next. Without loss of generality, assume that  $s_i \leq s_{i+1}$ ,  $1 \leq i < n$ . Also let  $\sigma$  be a permutation of  $(1, 2, 3, \dots, n)$  such that  $t_{\sigma(i)} \leq t_{\sigma(i+1)}$ ,  $1 \leq i < n$ . Define  $t_0 = 0$ ,  $s_0 = 0$  and  $\sigma(0) = 0$ . The interleaved FSM has  $n$  phases. Phase  $i$  applies  $t_{\sigma(i)} - t_{\sigma(i-1)}$  vectors and after it is over, BIT $\sigma(1)$  through BIT $\sigma(i)$  have been completely tested. If  $t_{\sigma(i)} - t_{\sigma(i-1)} = 0$  this phase can be ignored. The  $i^{\text{th}}$  phase starts with loading  $t_{\sigma(i)} - t_{\sigma(i-1)}$  and  $s_1$  (which is the minimum  $s_i$  value) into two working registers,  $T$  and  $S$ . Register  $S$  is then decremented and a SH signal is issued to the appropriate BIT structures, namely BIT $\sigma(i+1)$ , BIT $\sigma(i+2)$ , ... BIT $\sigma(n)$ . This is done repetitively until register  $S$  contains a zero. The value  $s_2 - s_1$  is then loaded into  $S$  and this count is used to issue SH pulses to all those BIT structures for which a sufficient number of SH pulses have not been issued so far. This process is continued until all the test vectors are shifted into their respective registers. A LD pulse is then applied to the appropriate BIT structures and the above cycle is repeated  $t_{\sigma(i)} - t_{\sigma(i-1)}$  more times.

In general, there are  $n$  phases, each with  $2n + 1$  states (2 for each difference  $(s_i - s_{i-1}) > 0$  and one final state). Hence the total number of states in the combined FSM is  $n \times (2n + 1)$ , requiring  $\lceil \log(n \times (2n + 1)) \rceil$  flip flops. The interleaved FSM requires  $2n$  constants (the differences  $(s_i - s_{i-1})$  and  $(t_{\sigma(i)} - t_{\sigma(i-1)})$  for  $i = 1$  to  $n$ ). We assume that they are stored off-chip and can be loaded by invocation of appropriate LD signal.

It can be noticed that the above controller does *not* have the minimum number of states required to control the  $n$  BIT structures. In fact, the number of states can be reduced by almost a factor of two. For example, consider the second phase in Figure 2.15. Since BIT1 has already been tested, the S register



can be loaded with 48, instead of 41 followed by 7. This results in a reduction of 2 states from the second phase. Similarly, 4 states can be deleted from the third phase. In general, the number of states which can be eliminated is given by  $2 + 4 + \dots + 2(n - 1) = n(n - 1)$ . Thus, instead of requiring  $\lceil \log n(2n + 1) \rceil$  flip flops,  $\lceil \log(n(2n + 1) - n(n - 1)) \rceil = \lceil \log n(n + 2) \rceil$  flip flops are sufficient. Thus the saving in terms of storage is just one flip flop. On the other hand, this change introduces asymmetry in the phases as each phase now has a different number of states. Because of this asymmetry, a different decoder will be needed to issue SH and LD signals in each phase, and hence the controller area will start increasing linearly with  $n$ . Therefore, the FSM shown in Figure 2.15 requires less area even though it does not have the minimum number of states.

Three factors contribute toward an efficient implementation of an interleaved controller. First, only three registers S, S' and T are required as opposed to  $2n$  registers in the case of  $n$  independent controllers. Secondly, the increase in the number of states in the FSM over a single controller does not lead to very much additional area. Since the state information can be encoded using just  $\lceil \log(n \times (2n + 1)) \rceil$  flip-flops, the variable part of the design (with the constants stored off-chip) increases only logarithmically with  $n$ . Thirdly, the proposed FSM is highly symmetric. Thus one can use a single  $2n + 1$  state machine in order to issue SH and LD pulses to all the BIT structure. A  $\lceil \log n \rceil$  bit counter, referred to as the template register, may be used to store which phase is being executed and, depending on its value, the control signals to the BIT structures that have already been tested may be disabled. Figure 2.16 shows a schematic for this implementation.

Under certain conditions the interleaved controller may entail a penalty in test time. Since the application of test vectors to different BIT structures is interleaved, it takes  $\max[s_i]$  units of time to apply one vector to all BIT structures. Thus the total test time is proportional to  $\max[t_i] \times \max[s_i]$ . Typically, the BIT structures with large value of  $s_i$  will also require the most number of test vectors, and hence  $\max[t_i] \times \max[s_i]$  will equal  $\max[t_i \times s_i]$ , the time required by  $n$  independent controllers. However, if the  $s_i$ 's and  $t_i$ 's are not well matched, there may be an unacceptably high penalty in test time. For example, if  $s_1 = 50$ ,  $s_2 = 100$ ,

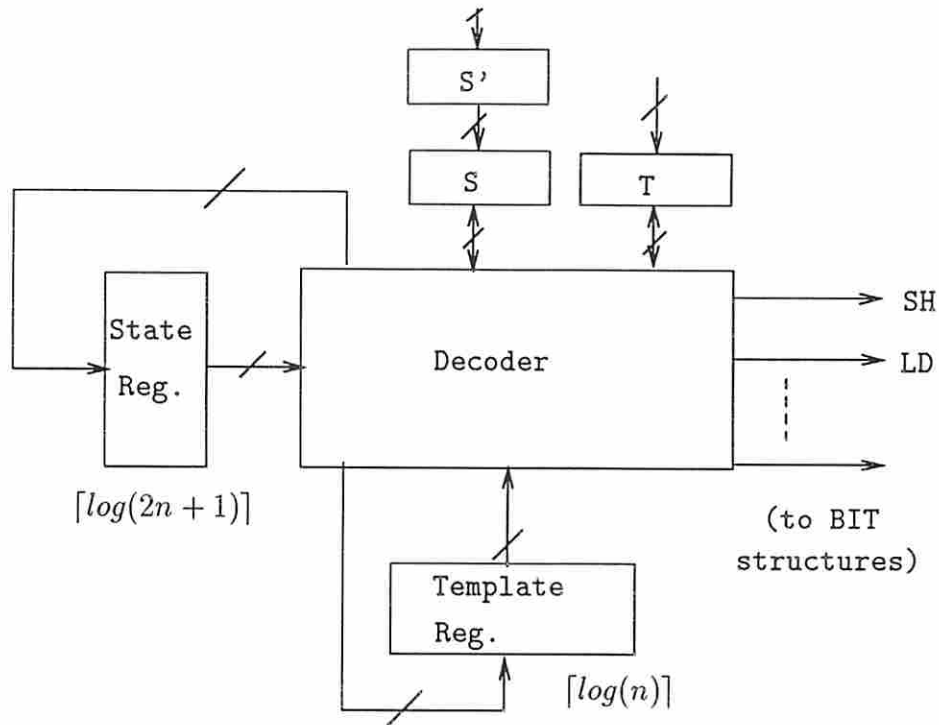


Figure 2.16: A controller for interleaved test execution.

$t_1 = 5000$  and  $t_2 = 1000$ , the interleaved controller will take  $100 \times 5000$  units of time instead of  $50 \times 5000$  required by two independent controllers.

### 2.3.2.2 Tree of Counters Design

The test time problem just discussed can be alleviated by executing all  $n$  BIT structures simultaneously without interleaving. The number of flip-flops used in the count-down registers can be reduced by sharing the common factors in the numbers  $t_i$  and  $s_i$  which need to be decremented. This scheme will be most effective when the number of common factors is large. Even in the worst case, this design methodology will be no worse than employing  $n$  independent controllers.

Consider once again the example given in the previous section. The controller has to provide a SH control signal to BIT1, BIT2, and BIT3 for 41, 48, and 62 clock cycles respectively; provide a LD control signal to the respective BIT structures after every 41, 48, and 62 clock cycles, respectively; and repeat this process

until all test vectors have been applied. To apply one test vector to BIT $_i$  requires  $s_i$  SHs followed by a LD, i.e., it is  $s_i + 1$  clock cycles. Notice that the LD signal is dependent on SH, i.e. LD = NOT(SH).

It is apparent from the above discussion that the task of generating appropriate control signals to the BIT structures is in essence that of repetitively (and simultaneously) counting a set of numbers. One needs to count from  $s_i$  to 0 to generate the SH signals and from  $(s_i + 1) \times t_i$  to zero to generate the DONE signals.

To illustrate this approach, assume that modulo 41, 48, and 62 counters are required. Since 7 is a common factor of these numbers a 3-bit modulo-7 counter that can be shared among all three counters can be used. This 'root' counter may be a hard-wired modulo-7 counter. This shared counter produces a terminal count signal ( $tc$ ) every 7 clock cycles that can be used to decrement modulo 6, 7 and 9 counters. Sharing of common factors can be continued recursively using factors of 6, 7, and 9, giving rise to a tree like structure. This logic is called a *Tree-of-Counters*. Figure 2.17 shows the optimal Tree-of-Counters for this example. No constants need be stored in this design approach since  $t_i$  and  $s_i$  are actually hard-wired into the modulo counters.

Clearly, considerable savings can be accrued by such sharing if the numbers to be counted have many common factors. In this example, if one used three separate counters, then  $\lceil \log 42 \rceil + \lceil \log 49 \rceil + \lceil \log 63 \rceil = 18$  flip-flops will be required. On the other hand, the proposed design uses only  $\lceil \log 7 \rceil + \lceil \log 3 \rceil + \lceil \log 7 \rceil + \lceil \log 2 \rceil + \lceil \log 3 \rceil = 11$  flip-flops.

The problem of designing an optimal Tree-of-Counters for a given set of numbers can be formalized as follows. Let the numbers to be counted be  $s_1, s_2, \dots, s_n$ , and let  $F_1, F_2, \dots, F_n$  be the sets containing the factors of these numbers. ( $F_i$ 's may contain multiple factors.) Also let  $f_{i,j}$ ,  $i = 1, 2, \dots, n$  and  $j = 1, \dots, |F_i|$  denote the  $j^{th}$  factor of  $s_i$ . The optimal Tree-of-Counters corresponds to a tree with  $n$  leaves, rooted at 1, such that

1. the product of the  $f_{i,j}$ 's on the path from the root to the  $i^{th}$  leaf is  $s_i$ , and
2.  $\sum \lceil \log f_{i,j} \rceil$  for all  $f_{i,j}$  in the tree is minimum over all possible trees.

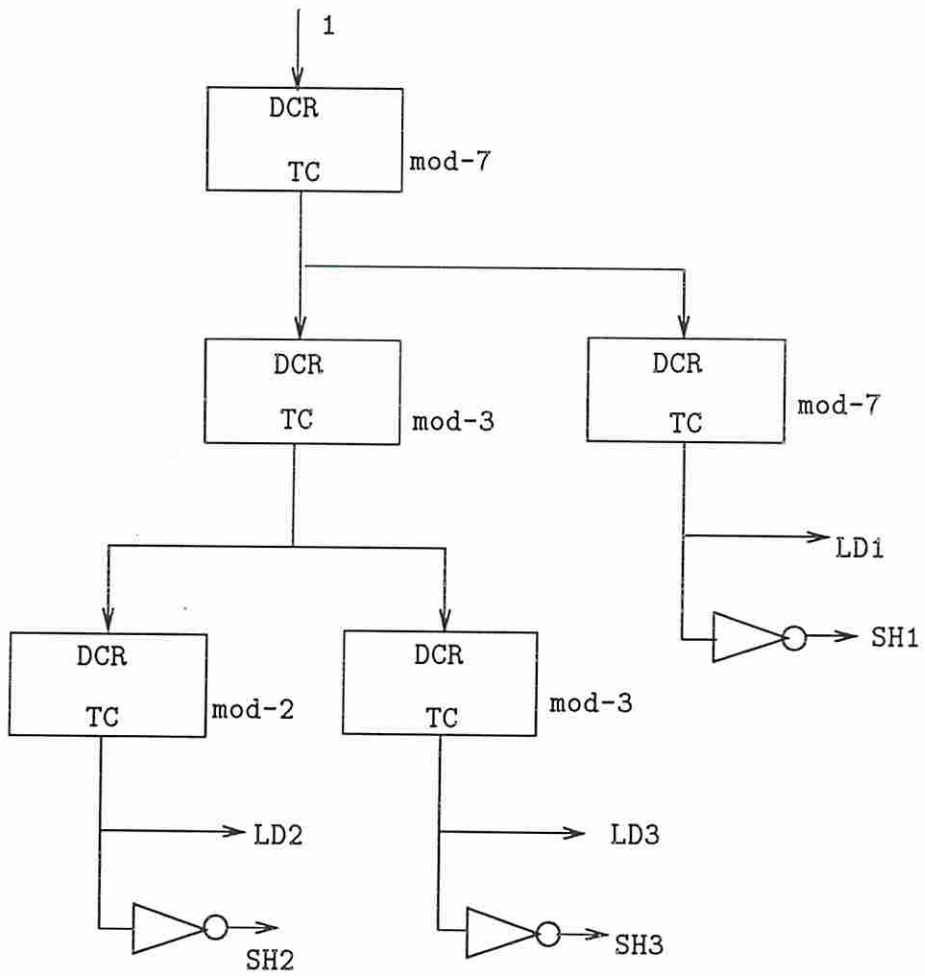


Figure 2.17: A Tree-Of-Counter Design.



The first item in the above definition ensures that the frequency of the terminal count signal of the  $i^{\text{th}}$  leaf counter is  $s_i$ . Because common factors may be shared in different ways, several trees are possible for a given set of numbers. For example, Figure 2.18 shows three alternative trees for the  $s_i$  values of 510, 714, 595, and 78. The dashed rectangles in the figure represent a counter for the product of the factors in the rectangle. The second item in the above definition requires that the total cost of constructing these trees in terms of the number of flip flops be minimum.

The problem of constructing the optimal Tree-of-Counters appears to be computationally intractable. In fact it is not known whether it is even in the NP class. A complete discussion of the computational complexity of this problem is beyond the scope of this work. In the remainder of this section, a ‘greedy’ heuristic, that attempts to obtain a good solution by locally optimizing the savings associated with any proposed counter sharing, is presented.

To make counters for  $s_i$  and  $s_j$  one can have a mod- $\alpha$  counter,  $\alpha = GCD(s_i, s_j)$  feeding counters for  $\frac{s_i}{GCD(s_i, s_j)}$  and  $\frac{s_j}{GCD(s_i, s_j)}$ . This sharing results in a saving of  $\lceil \log(GCD(s_i, s_j)) \rceil$  and two counters having  $\lceil \log \frac{s_i}{GCD(s_i, s_j)} \rceil$  and  $\lceil \log \frac{s_j}{GCD(s_i, s_j)} \rceil$  bits respectively have to be constructed. These two counters will be driven by a signal from the mod- $\alpha$  counter. Thus, with each pair  $s_i$  and  $s_j$ , we can associate a score  $S$  given by  $S(s_i, s_j) = \lceil \log(GCD(s_i, s_j)) \rceil$  which weights the sharing of common factors between  $s_i$  and  $s_j$ . A greedy procedure for generating a near optimal Tree-of-Counters is presented below.

```

TreeOfCounters(C)
{if |C| = {} then return;
  else if |C| = {a} then
    {GenCounter(a, 1); return; }
  else {find a, b in C such that S(a,b) is maximum;
    if S(a,b) = 0 then
      {for all x in C, GenCounter(a, 1); return; }
    GenCounter(a/GCD(a,b), GCD(a, b));
    GenCounter(b/GCD(a,b), GCD(a, b));
  }
}

```



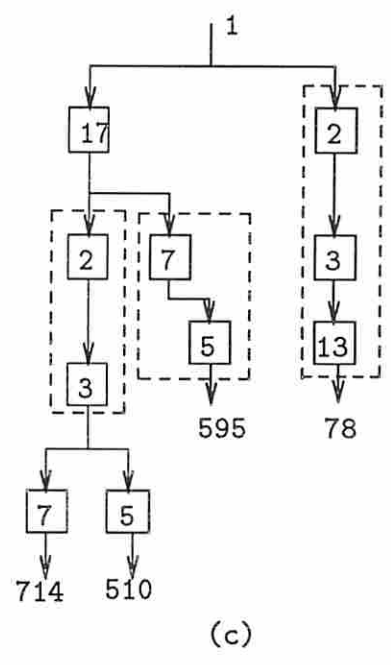
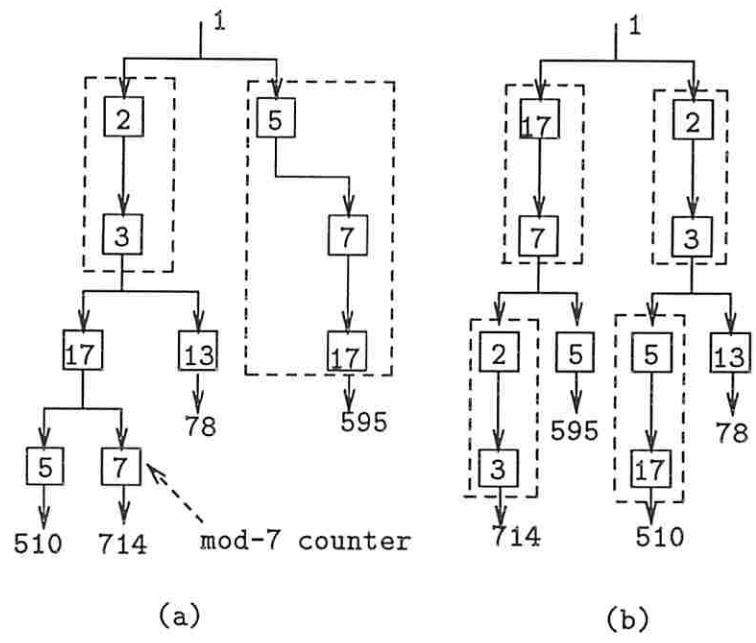


Figure 2.18: Three Trees-Of-Counters.

```

delete(a, C); delete(b, C);
insert(GCD(a,b), C);
TreeOfCounters(C); }

```

This algorithm works bottom up from leaves to the root. For any given set  $C$  of numbers the algorithm computes the score  $S(s_i, s_j)$  for all  $s_i, s_j \in C$ . It then picks the pair with the maximum score, generates a  $\text{mod}(\frac{s_i}{\text{GCD}(s_i, s_j)})$  and  $\text{mod}(\frac{s_j}{\text{GCD}(s_i, s_j)})$  counter, deletes  $s_i$  and  $s_j$  from  $C$  and inserts  $\text{GCD}(s_i, s_j)$  into  $C$ . These two counters are decremented by the terminal count signal of  $\text{GCD}(s_i, s_j)$ . The procedure now recursively calls itself with the modified set  $C$  of numbers. It terminates when there is only one number left in  $C$ , or the maximum score for any pair is 0 indicating that all elements of  $C$  are relatively prime. In the procedure, the routine  $\text{GenCounter}(A, B)$  generates mod- $A$  counter which is enabled by the terminal count signal of the counter for mod- $B$ . Figure 2.18(c) shows an example Tree-of-Counters constructed using this procedure.

The step “find  $a, b$  in  $C$  such that  $S(a, b)$  is maximum” is the most complex step in the algorithm. Initially,  $O(n^2)$   $\text{GCD}$  computations will be required in order to find the maximum value of  $S(a, b)$ . Since in each recursive step the size of  $C$  diminishes by one, there will be  $n$  iterations and the above procedure can be easily executed in  $O(n^3)$  time. It is possible to reduce this time complexity to  $O(n^2)$  if in each iteration  $a$  and  $b$  are replaced by  $\text{GCD}(a, b)$ . Thus in each successive iteration the score  $S$  need be recomputed only for the pair formed by this new entry and the old numbers. Hence it is possible to find the maximum value of  $S(a, b)$  in  $O(n)$  time in all iterations except the first one, resulting in an  $O(n^2)$  overall time complexity.

### 2.3.2.3 Counter Sharing Design

In the previous design, specific common factors of the form  $\text{GCD}(A, B)$  were employed and used to enable the counters for  $A/\text{GCD}(A, B)$  and  $B/\text{GCD}(A, B)$ . In the Counter Sharing design scheme a set of common factors are obtained, and these factors are combined to derive the desired SH and LD signals. Let  $tc_x$  denote the terminal count signal of a mod- $x$  counter. One can generate signals of any frequency by simply ANDing together appropriate factors of a desired frequency. For example, a

signal that goes high every 100 clock cycles can be generated by ANDing  $tc_{25}$  and  $tc_4$ , denoted by  $AND(tc_{25}, tc_4)$ . Notice that repeated prime factors, e.g.  $\{5, 5\}$  and  $\{2, 2\}$  in this case, have to be multiplied and a larger common factor counter must be used. That is, it is wrong to implement  $AND(tc_5, tc_5, tc_2, tc_2)$ , since this would produce a signal every 10 clock cycles instead of every 100 clock cycles. Figure 2.19 shows how to derive  $tc_{10}$  and  $tc_{100}$  from mod-2 and mod-5 counters. In general, if  $\prod_{i=1}^r \alpha_i^{\beta_i}$  is the unique prime factorization of  $N$ , then a signal that goes high every  $N$  cycles can be produced by  $AND(tc_{\alpha_1^{\beta_1}}, \dots, tc_{\alpha_r^{\beta_r}})$ .

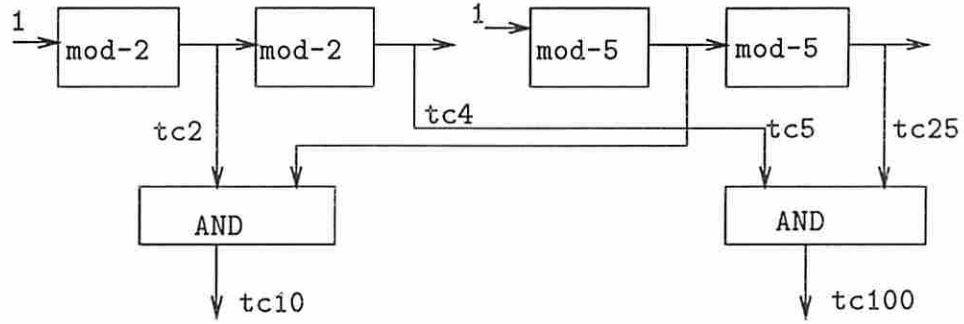


Figure 2.19: A counter-sharing design.

Let  $T_j = t_j \times (s_j + 1)$  denote the total test time for the  $j^{\text{th}}$  BIT structure. To control this structure, we need a signal  $tc_{s_j+1}$  to issue appropriate LATCH signals and a signal  $tc_{T_j}$  to mark the end of testing. Let  $factor(X) = \{\alpha^i | \alpha^i \text{ occur in the prime factorization of } X\}$  and  $U = \bigcup_{j=1}^n (factor(T_j) \cup factor(s_j))$  be the set of all the factors. Here the union operation  $\bigcup$  takes the maximum power for each replicated prime factor, i.e.,  $\bigcup(\alpha^i, \alpha^j) = \alpha^{\max(i,j)}$ . The control signals for all the BIT structures can now be generated from the common pool  $U$  of factors by appropriately ANDing the appropriate signals. For example, if  $T_j = \alpha_{j1}^{\beta_{j1}} \alpha_{j2}^{\beta_{j2}} \dots \alpha_{jr}^{\beta_{jr}}$  is the prime factorization of  $T_j$  then  $tc_{T_j} = tc_{t_j \times s_j + 1} = AND(tc_{\alpha_{j1}^{\beta_{j1}}}, \dots, tc_{\alpha_{jr}^{\beta_{jr}}})$  will generate the DONE signal for BIT $_j$ . Signals  $s_j$ 's can be similarly generated.

This design requires  $2n$  AND gates (one for each  $s_j$  and the other for each  $T_j$ ). Notice that no other circuitry is required, either for decoding or for maintaining the state information. The total test time required by this design is  $\max_{j=1}^n T_j = \max_{j=1}^n (t_j \times (s_j + 1))$  which is considerably better than  $\sum_{j=1}^n T_j$ , the time required by a sequential controller.

#### 2.3.2.4 Comparison of Three Designs

The interleaved FSM controller requires on the order of  $\max[s_i] \times \max[t_i]$  clock cycles to completely activate all BIT structures. For a 'balanced' problem, i.e. the one in which  $\max[s_i]$  and  $\max[t_i]$  occur for the same BIT structure, this will be optimal. In addition, the area occupied by the FSM excluding the area required for storing the constants, increases only logarithmically with the number of BIT structures.

The Tree-of-Counters design and the Shared-Counter design are useful when  $\max[s_i] \times \max[t_i]$  is much larger than  $\max[s_i \times t_i]$ , and when the  $s_i$ 's and  $t_i$ 's have many factors in common. Both designs operate the BIT structures concurrently, without any penalty in test time. Also the area overhead for these designs is considerably less than that incurred when independent controllers are used. This is achieved through register sharing and elimination of decoding circuitry.

The basic idea in both the Tree-of-Counters and Counter-Sharing designs is the same. In the former the terminal count signal of one counter is used to decrement a set of other counters giving rise to a tree structure. The amount of sharing in this scheme depends heavily on the problem at hand and may be limited for some problem instances. For example, if counters for  $11 \times 13$ ,  $11 \times 31$ , and  $31 \times 13$  have to be designed, only two of these numbers can share a counter. The Counter-Sharing design alleviates this problem by constructing a common pool of counters which is shared among all numbers. For each number appropriate terminal count signals are simply ANDed together. For most problems this design will be superior to the Tree-of-Counters design. However, there exist situations when a Tree-of-Counters is more desirable. For example, if counters for  $11 \times 13$  and  $11 \times 31$  are to be constructed, both design schemes will require modulo counters for 11, 13, and 31. The Tree-of-Counters design will use mod-11 to drive mod-13 and mod-31. On the other hand, the Counter-Sharing design will need extra AND gates to produce signals of appropriate frequencies. It is this saving in terms of AND gates which may make Tree-of-Counters more desirable for some problem instances.

For any BIT structure the value of  $s_i$  is fixed by the length of its scan-chain or internal shift registers. The number of vectors ( $t_i$ ), however, can in general be increased, if doing so will result in reduced area without excessive penalty in test

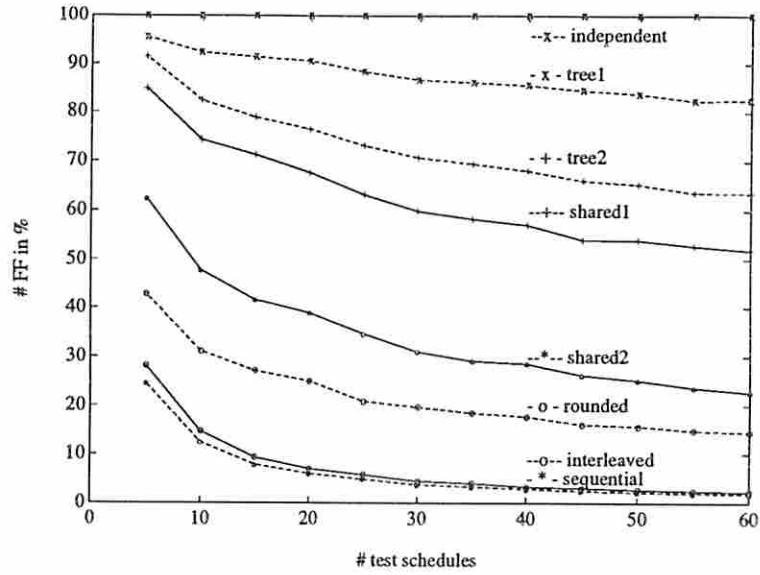


Figure 2.20: Hardware complexity for different designs.

time. Since common factors in the  $t_i$ 's are shared, by modifying the value of  $t_i$ , it should be possible to significantly reduce the controller area. The simulations experiments described below show that this indeed is the case. In particular, we will see that if the  $t_i$ 's are increased to become multiples of some number, say 100, both the Tree-of-Counters and Counter-Sharing designs result in reduced area. This clearly increases test time. However, this increase may be acceptable in exchange for reduced area and associated increase in fault coverage.

Controller area and test times have been determined for each of the designs described. The results are shown in Figures 2.20 and 2.21. Figure 2.20 shows the average number of flip flops required by the various designs relative to that of  $n$  independent controllers. In order to plot these curves,  $n$  random pairs of  $s_i$  and  $t_i$  values were generated. The number of flip flops required by each design was then estimated (as described below) and averaged over 20 iterations. The results indicated correspond to the following cases, described from top to bottom in the figure. The value of  $s_i$  range from 50 to 250; that of  $t_i$  from 1000 to 50,000.



1. `independent` – represents the area occupied by  $n$  independent controllers with no optimization. The area is estimated to be  $\sum_{i=1}^n (\lceil \log s_i \rceil + \lceil \log t_i \rceil)$  and is used as a normalization factor for all other curves. The area of the FSM is ignored.
2. `tree1` – represents the area occupied by the tree generated using the procedure `TreeOfCounters`.
3. `tree2` – same as above except  $t_i$ 's are incremented to the next multiple of 100.
4. `shared1` – indicates the area occupied by the Shared Counter design; the area for the AND gates is ignored.
5. `shared2` – same as above with  $t_i$ 's advanced to the next multiple of 100.
6. `rounded` – same as 4 above with  $t_i$ 's increased to the next power of 2.
7. `interleaved` – represents the area occupied by the interleaved controller estimated by  $2\lceil \log(\max s_i) \rceil + \lceil \log(\max(t_i)) \rceil + \lceil \log n \rceil + \lceil \log(2n + 1) \rceil$ . Here the terms represent the area occupied by the S and T registers, the state register, and the template register (see Figure 2.16). The area occupied by the decoder is neglected and all constants are assumed to be stored off-chip.
8. `sequential` - represents the area occupied by a sequential controller which is shared among all the BIT structures. In this case we need an additional counter to distinguish between various BIT structures and the area is given by  $2\lceil \log(\max s_i) \rceil + \lceil \log(\max(t_i)) \rceil + \lceil \log n \rceil$ .

It should be emphasized that the above area estimates only consider the area for counters and flip flops required for storing the state information. All the constants are assumed to be stored off-chip and loaded through activation of appropriate LD signals. These estimates nonetheless are representative of the total controller area. When the area for storing the constants  $t_i$ 's and  $s_i$ 's are considered, those designs using mod counters become even more attractive because these constants are hard-wired into the counters themselves.

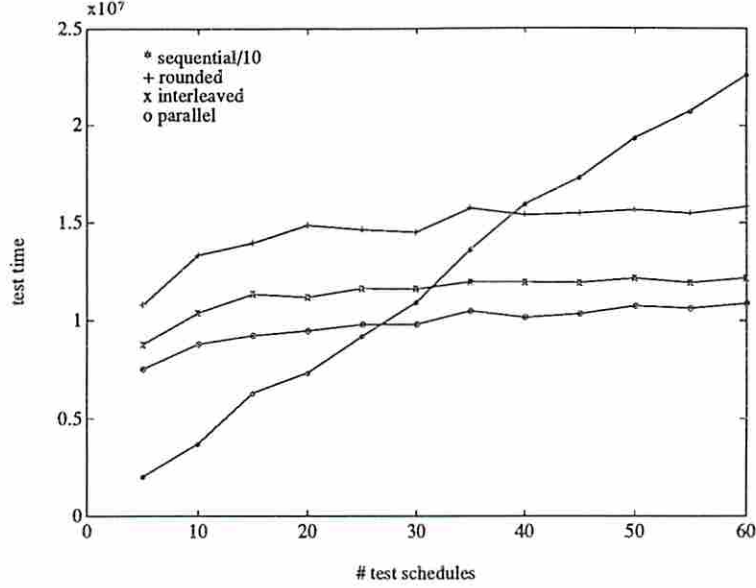


Figure 2.21: Time complexity for different designs.

Figure 2.21 shows the test times for these designs. Not surprisingly, the  $n$  independent controllers, the Tree-of-Counters Design, and the Shared Counter designs all take the same time indicated by the bottom most curve (labeled `parallel`) in the figure. This time corresponds to the maximum time taken by any BIT structure and is given by  $\max[(s_i + 1) \times t_i]$ . The test time for the interleaved design (labeled `interleaved`) is proportional to  $\max[(s_i + 1)] \times [\max t_i]$ . If the values of the  $t_i$ 's are changed to the next higher power of 2, the design area can be reduced, as shown by the curve marked 'rounded' in Figure 2.20. The corresponding increase in test time is given by the curve labeled `rounded` in Figure 2.21. Finally, the time taken by a sequential controller, which is given by  $\sum (s_i + 1) \times t_i$ , is shown. This curve is labeled `sequential/10` and is scaled by a factor of  $\frac{1}{10}$  in order to more clearly show its relationships with the other curves. The test time for the sequential controller is significantly greater than any other design, and increases monotonically with  $n$ .

Several conclusions can be drawn from these plots. First, all the designs occupy an area which is between that of using  $n$  independent controllers and the sequential design. The interleaved FSM design may be viewed as an enhancement

to the sequential design. The area occupied by it is only marginally more than the sequential design. At the same time, its test time is comparable to that of parallel controllers. Secondly, on the average, the Shared Counter design performs better than the Tree-of-Counters design. However, the Tree-of-Counter design should not be totally ruled out because in some cases both may use the same number of flip flops while the later will require additional AND gates for decoding, which have not been accounted for in these plots. Thirdly, the area occupied by both the common factor sharing designs can be further reduced by appropriately selecting  $t_i$  values. The reduction in area appears to be proportional to the corresponding increase in test time. Lastly, these designs reduce the rapid rise in test time as  $n$  increases. It is clear that if the  $t_i$ 's can be either forced to have more common factors, or better yet to take on values from a small set, then the area overhead can be considerably reduced. The same is true for the  $s_i$  values. Though it is not feasible to make a scan chain longer, pseudo test bits can be added to each vector to make the apparent length of a scan chain longer. For example, a test vector of 47 bits can be preceded by 3 garbage bits to produce a test vector of 50 bits. After 50 SHIFT pulses, only the desired 47 bits will reside in the corresponding 47 flip flop shift register chain. Thus by forcing the  $s_i$  values also to be elements of a small set the area of the controller can be further reduced.

## Chapter 3

### Controller for Testable Modules

This chapter deals with the design and implementation of an MMC. An MMC is used to control the self-test process of a module (or board) by accessing each chip's BIT structures through an L0-bus. The proposed MMC is universal in the sense that the same basic design is used for all modules. MMCs differ by the test programs they execute, the number of test busses they control, and the expansion units they employ. Test programs are used to control the processor in an MMC in the execution of the Built-In Self-Test (BIST) process for the entire module. The test results are then reported to a SuMP via an L1-bus. A SuMP can initiate the self-test process of a module by sending a "begin test" command to the MMC on that module. The MMC then reports the "health status" of that module back to the SuMP.

An MMC contains bus interface units (such as an L1-slave and an L0-master), a processing unit (such as a processor), a memory unit (consisting of RAMs and ROMs), one or more test channels, a Bus Driver/Receiver, one or more expansion units (such as testability registers and analog test interface), and a CMC.

A simple yet novel design, called the *test channel*, is used in an MMC. Since every testable chip has an L0-slave in its CMC, a test channel, which contains an L0-master, can communicate over an L0-bus with the CMC. The MMC's processor can control a test channel by reading from or writing to its internal registers. Once initiated by the processor, a test channel can completely control an L0-bus and

the testing of a chip. The separation of processor and test busses provided by test channels prevents the processor from dealing with detailed bus timing activities. A test channel translates processor instructions into proper timing sequences for an L0-bus. A test process can now be represented as high level processor instructions.

In [15], Budde reported on the design of the Testprocessor which is similar to our MMC. The Testprocessor is intended to carry out some of the functions of the CMC and the MMC. Since it may be part of an application chip, it must be simple. The Testprocessor is programmed at the microinstruction level. All peripheral devices are controlled directly by the control signals provided by these microinstructions. The number of expansion units is limited by the total number of control signals the control unit can provide. Data can be moved directly between the test pattern RAM and the test interfaces without going through the processor register. Obviously, this is an efficient approach for data movement. However, due to the limitation of the bus, only one serial interface can run at a time. Comparisons are done using a fault-secure comparator. There is no other data processing unit in the Testprocessor. Due to the limited processing capability, diagnostic programs cannot run on the Testprocessor.

### 3.1 Requirements for an MMC

An MMC must be able to respond to requests from a SuMP, to carry out tests for every chip on the module, and to report test results to a SuMP. The requirements for an MMC are stated below, followed by a description of its architecture in the next section. In summary, an MMC should be able to support the following functions:

1. Access the on-chip BIT structures via an L0-bus.
2. Provide proper control sequences for the execution of a chip's BIT structures.
3. Generate test data and collect test results if necessary.
4. Analyze test results to monitor the health status of chips.



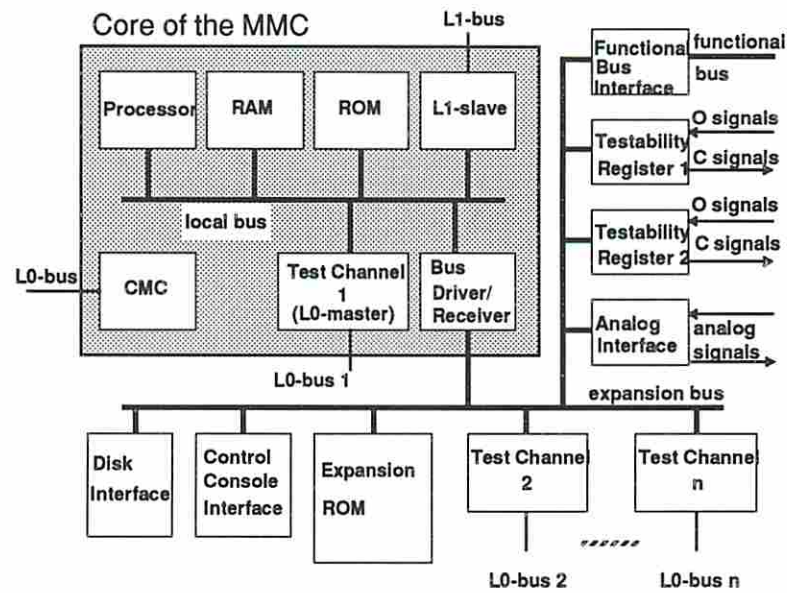
5. Test the interconnects among different chips on the module via the boundary scan registers.
6. Provide controllability and observability for non-testable chips and analog circuits.
7. Interface with a SuMP or the control console.

In addition, an MMC must have memory to store test data and/or test results if deterministic test data is needed. This requirement on memory is relaxed for random or exhaustive test methodologies since only seed data and signatures need to be stored.

## 3.2 MMC Architecture

Figure 3.1 shows the architecture of an MMC. It consists of a 16 bit general or special purpose processor, a ROM, a RAM, a test channel, a CMC with an L0-slave, an L1-slave, and a Bus Driver/Receiver (BDR). The BDR supports an expansion bus, i.e., it allows extra units to be added to the MMC. For example, a functional bus interface, two testability registers, an analog test interface, several test channels, an expansion ROM, a control console interface and a disk interface are shown to communicate with the MMC through the BDR in the figure. The components shown in the shaded region (which can be implemented as a single ASIC chip) are required for every MMC. CMCs for these chips are not shown.

All units on the local and expansion bus are accessed by the processor in a *memory-map* schema. That is, every accessible register of each unit occupies one location in the global address space. The processor can read from or write into these registers by first addressing the appropriate registers. Each unit must be able to decode the address lines. Once the register is selected, an enable signal is generated to initiate a read or write operation.



Use IEEE 1149.1 as L0-bus

Figure 3.1: The architecture of an MMC.

### 3.2.1 Test Channel Design

A CMC may have a pseudorandom test pattern generator (TPG) and a signature analyzer (SA), which can be implemented using linear feedback shift registers [51]. In this case only control signals need be supplied by a test bus during self-test. An example of such a design is presented in [5]. However, if the chip does not have these facilities and is to be tested using pseudorandom test data, then a TPG and an SA must be made a part of the MMC. For chips tested by deterministic test vectors, an MMC must be able to provide test vectors and obtain test results via a test channel.

Once initialized by the processor, the primary function of the test channel is to control an L0-bus autonomously. The processor can then be used for other tasks. As a result, high test parallelism can be achieved through running several test channels at the same time.

The major functions of the test channel are listed below.

1. Serve as an L0-master.
2. Transmit instructions to and receive status from chips.
3. Generate and transmit pseudorandom test data and receive and compact test results.
4. Transmit deterministic test vectors to and receive test results from chips.
5. Generate interrupts and also direct interrupts from chips to the processor.
6. Keep count of the number of tests applied, and the number of bits of each test or instruction transmitted.

### **Organization of the Test Channel:**

Figure 3.2 shows a block diagram of the test channel. The test channel consists of a Transmitter Register (TxR) for transmitting data over the TDI line; a Receiver Register (RxR) for receiving data on the TDO line; two polynomial control and buffer registers PA and PB; a control register (CR) which specifies the operational mode, selection and function enabling information; a status register (SR) which contains the current chip status; three counters, namely TC, which stores the total number of test vectors to be sent, SC which keeps track of the number of bits in a test vector which have been transmitted, and DC which keeps track of the elapse idle time between two vectors; a register CNR which contains the initial values for SC and DC; a register select circuit for processor read/write control; an interrupt circuit to request service from the processor; and a control unit FSM1 which implements the L0-master protocol and is used to send and receive information via an L0-bus under the control of the CR and the three counters. If the test channel is implemented as a stand-alone unit, then it should also have a CMC.

The I/O pins of the test channel consists of /WR, /RD, /CS, PA, PD, Direct, TDI, TDO, TMSi, TCK, and other interrupt signals. The processor can write a word of data from the data bus PD to a register, addressed by PA, in the test channel by simultaneously activating the signals /WR and /CS. Similarly, the processor can read a word of data from a register in the test channel to the data bus by simultaneously activating the signals /RD and /CS.

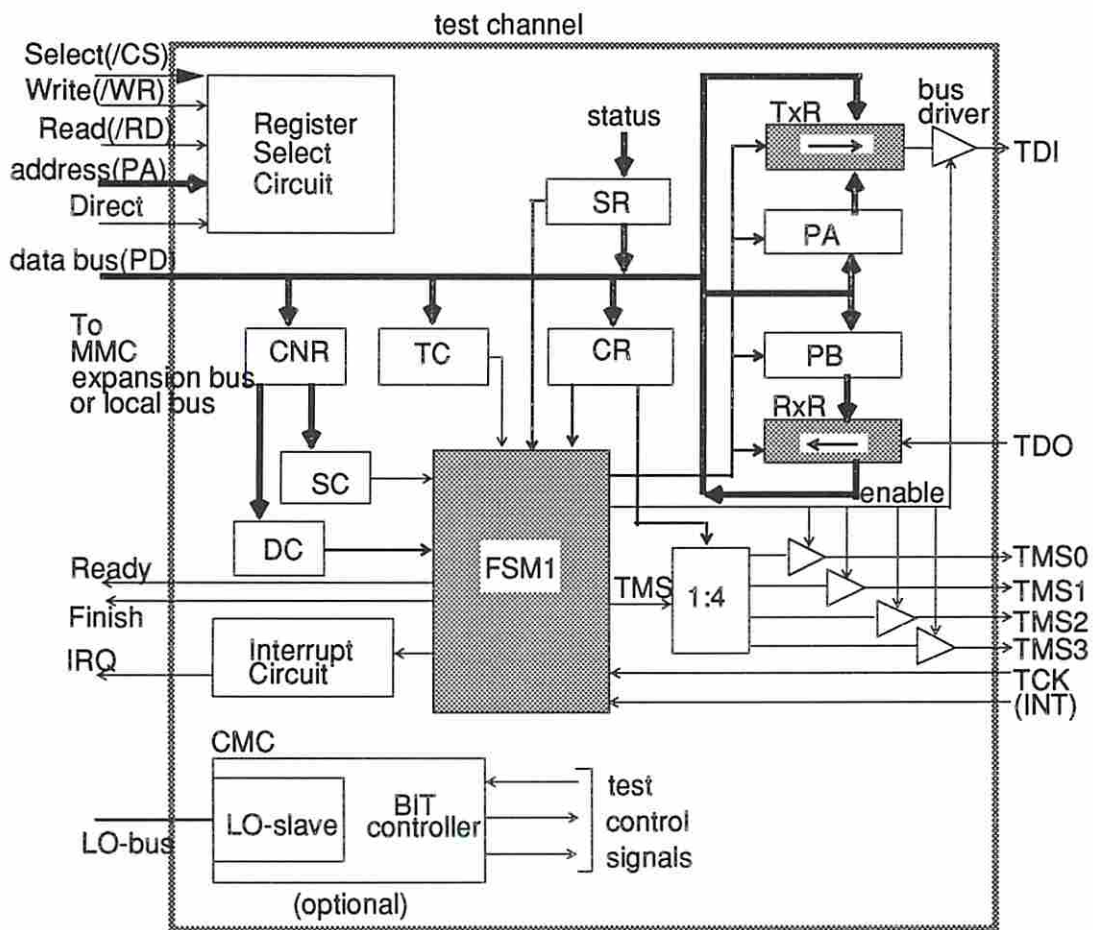


Figure 3.2: The architecture of the test channel.

Output signals, such as TDI, TMS are all driven through a tri-state buffer thus allowing two or more test channels to be connected to an L0-bus. This enhances the reliability of the test process as well as enables external testing of a module by another MMC [12]. A more detailed description of the major blocks follows.

1. TxR (Transmitter Register). The TxR is a 16 bit register with parallel LOAD, SHIFT and TPG capabilities. It is used to transmit data over the TDI line. During pseudorandom data transmission the TxR acts as a TPG. The feedback polynomial of the TPG is controlled by the PA. Any feedback polynomial can be realized since the PA is directly writable by the processor. The seed value for the TPG can also be loaded by the processor. During instruction or deterministic data transmission the TxR acts as a shift register. It must be loaded with a new word of data before transmission is initiated. The PA serves as a buffer for transmission. Once the TxR is empty, the next word of data, which is already in the PA, is copied into the TxR. Processor service is then requested in order to load a new word of data into the PA. Transmission over the L0-bus is not interrupted during the 16 clock cycle window in which the PA may receive a new data word. If the data transfer rate is not fast enough, or when the TxR is empty the PA does not contain a new word of data, the L0-bus enters a pause state until the PA is loaded.
2. RxR (Receiver Register). The RxR is a 16 bit register with parallel READ, SHIFT and SA capabilities. It is used to receive data from the TDO line. Received data is either read by the processor or compressed into a signature. During pseudorandom data transmission the RxR acts as an SA. The feedback polynomial is controlled by the PB. The final signature in the RxR can be read out via a processor *read* operation. During transmission of status or deterministic results, data on the TDO line is shifted into the RxR. The PB serves as a buffer. Once the RxR is full, its contents is copied into the PB. A service request is generated to signal the processor to read the PB and store the data in the RAM. If the previous result in the PB has not yet been read, the L0-bus enters a pause state. Transmission cannot start again until the PB is read and the RxR transfers its data to the PB.



3. PA, PB (polynomial control registers): Both registers are 16 bit wide and have parallel LOAD capability. They can be accessed by the processor via the data bus. Their functions have already been described.
4. CR (Control Register). The CR is a 7 bit register. Symbolic names used for the CR bits are *FSMen*, *INTen*, *MS0*, *MS1*, *BS0*, *BS1* and *Scan*. *FSMen* and *INTen* enable FSM1 and the interrupt circuit, respectively; *MS0* and *MS1* specify the operational mode; *BS0* and *BS1* select one of the TMS<sub>i</sub> (i=0,1,2,3) signals; and *Scan* determines the scan or non-scan operation.
5. SR (Status Register). The SR register consists of 4 bits namely, *Finish*, *IRQ*, *Ready* and *Wait*. The *Ready* bit is cleared whenever the content of the PA is copied into the TxR, and is set whenever the processor loads new data into the PA. The *Finish* bit is set only when the required information has been transferred, or TC reaches 0. The *IRQ* bit is set when the INT line from the test bus is active. The *Wait* bit is set when both the TxR and PA are empty, and is in the reset state when the TxR is loaded. A processor SR *read* operation also reads the content of CR, i.e. 11 bits are read. This operation can be performed independent of the state of the FSM1. Bits *Finish* and *IRQ* are cleared whenever the SR is read.
6. TC (Test Counter). The TC counts the number of test vectors transmitted during the execution of one test session. The TC is a 22-bit down counter; it is able to count down to 0 from any number between 1 and 4,194,303. It requires two processor write operations to load: one of the write operations loads part of this counter and part of the CR, another loads the rest of the counter.
7. SC (Scan Counter). The SC is used to keep count of the number of bits of a test vector or instruction which have been transmitted. SC is a 10-bit down counter and can count down to 0 from any number between 1 and 1023. Its initial value is loaded from the CNR. A terminal count signal will be activated whenever the value in SC reaches 0, and the value *s* in CNR will be copied into SC. In transmitting *t* test vectors to a chip during one test session, SC must be re-initialized (to the value *s*) *t* times.

8. DC (Delay Counter). The DC is a 5 bit down counter and is used to count the number of clock cycles between the transmission of two consecutive test vectors. Its initial value can be loaded from the CNR. The DC can count down to 0 from any number between 1 and 31. A terminal count signal will be activated whenever DC reaches 0, and the value  $d$  in the CNR will be copied into the DC.
9. CNR (Count Number Register). This buffer is used to store the initial value of the constants for both SC and DC, i.e.  $s$  and  $d$  referred to above. As discussed above, these counters destroy their original contents after a test vector is transmitted. Thus CNR is used to restore the values in both counters so that the next vector can be transmitted. The CNR is 15 bits long. It can be loaded by a single processor write operation.
10. Register Select Circuit. This circuit is driven by the processor and is used to control the access to various registers in the test channel. Registers CNR, TC, CR, SR, TxR, RxR, PA, PB are accessible to the processor. When the *Direct* signal is inactive, the registers are selected by address. When the *Direct* signal is active, this circuit interprets a processor *read* operation as a *write to the PA* operation, thus ignoring the address lines. In addition, the address and *Read* signals are used to read a word from the memory unit. Thus a word of data is transferred from the memory unit to the PA of the selected test channel. Similarly, when *Direct* is active a processor *write* operation is interpreted as a *read from the PB* operation. The address and *Write* signals are used to write the contents of the PB into the memory unit.
11. FSM1. This circuit controls the operation of the test channel and acts as an L0-master. It receives control signals from the CR and conditional signals from the counters TC, SC, and DC. When the *FSMen* bit is set, a processor generated *write* operation is used to issue a *Start* signal which in turn initiates the FSM1.

The initialization of the test channel includes the loading of the following register of the test channel: CR, TC, CNR, PA, PB, and TxR. The contents of the

registers SR and RxR, which are the results of the previous operation, should be read before the test channel is again enabled for the next operation.

### Operation of the Test Channel:

The operation of a test channel is controlled by its FSM1. The FSM1 controls the state of a test bus via signal line TMS (see Figure 3.2). The possible bus states are shown in Figure 3.3.

A test channel provides two types of operation: *RunTest* and *Scan*. During *RunTest*, the test bus enters the *Idle/RunTest* state for a pre-determined number of clock cycles. The TC counter keeps track of this number. No data is transmitted on either the TDI or TDO lines. This type of operation is used when a chip under test has BIST capability and the BIST hardware has been properly initialized through the test bus. The chip's BIST controller runs the self-test as long as the bus stays in the *Idle/RunTest* state.

During *Scan* operation, the test channel transfers either pseudorandom test data (PTD), deterministic test data without results compression (DTD), deterministic test data with results compression (DRC), or instructions (INS). The operation of the test channel is controlled by the CR and three counters. These counters are used for all types of information transfer. During the different operational modes, these counters may be used for different purposes. For example, in PTD transmission, the TC keeps track of the number of test vectors applied, the SC keeps track of the number of bits transmitted, and the DC keeps track of the number of elapsed clock cycles between two consecutive test vectors. Table 3.1 indicates how these counters are used. The operational modes of the TxR and RxR are also shown in the table.

Figure 3.3 shows the state transitions carried out by the FSM1 of the test channel. The dashed rectangle represents a wait for processor service. The operations indicated in the solid rectangles are executed in one clock cycle. The protocol of this state transition diagram is consistent with the IEEE 1149.1 boundary scan protocol.

The *FSMen* bit is cleared during the power up process, and the test channel enters the *idle* state at this time. The processor can read from and write into



|     | PTD          | DTD       | DRC       | INS       | RunTest      |
|-----|--------------|-----------|-----------|-----------|--------------|
| TC  | # tests      | # tests   | # tests   | set to 1  | # clk cycles |
| SC  | # bits       | # bits    | # bits    | # bits    | ---          |
| DC  | # clk cycles | set to 15 | set to 15 | set to 15 | ---          |
| TxR | TPG          | SHIFT     | SHIFT     | SHIFT     | ---          |
| RxR | SA           | SHIFT     | SA        | SHIFT     | ---          |

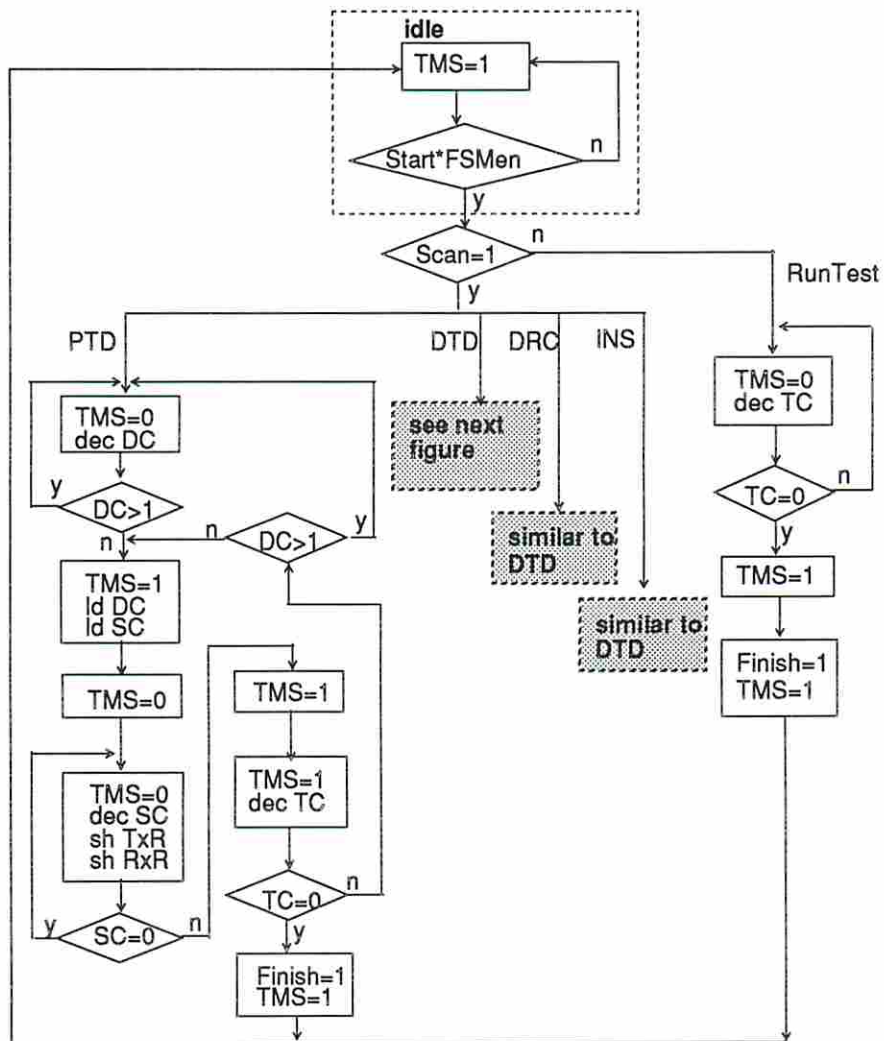


Figure 3.3: The state transition diagram for a test channel.





internal registers of the test channel while in this state. After initializing the appropriate set of registers, setting the *Start* signal and *FSMen* bit will initiate the operation of the FSM1. Depending on the setting of *Scan*, *MS0* and *MS1* bits, the FSM1 follows one of the five major branches as shown in Figure 3.3.

The branch labeled PTD is executed when pseudorandom testing is needed. Registers PA, PB, TxR, RxR, CNR and TC are assumed to have been initialized to their appropriate values, such as  $pa$ ,  $pb$ ,  $seed1$ ,  $seed2$ ,  $(s, d)$  and  $t$ . The TxR acts as a TPG with  $pa$  selecting the feedback polynomial and  $seed1$  as its initial value; RxR acts as an SA with  $pb$  selecting the feedback polynomial and  $seed2$  as its initial value. The test channel then autonomously transmits  $t$  random test vectors generated by TxR to TDI and compresses  $t$  test results in the RxR. Each test result is  $s$  bit long, and  $d$  clock cycles of delay exist between two consecutive test vectors. No service from the processor is required during pseudorandom testing. The *Finish* bit is set when process is completed. The processor then reads the signature stored in RxR to determine the test result.

The branch labeled DTD (see Figure 3.4) is executed when deterministic test data is used. Registers CNR and TC contain the values  $(s, d)$  and  $t$ . Note that  $d$  is always set to 15 for the DTD process. Its purpose is to clear the *Ready* bit after the transmission of 16 bits. For a test vector longer than 16 bits, the TxR is loaded with the first 16 bits of deterministic test data before the *Start* signal is activated. After 15 shift operations, the TxR contains the last bit of the test data. One clock cycle later the RxR is full. Two possible situations exist. After these shift operations have occurred it is possible that the PA is full ( $Ready=1$ ). Then the content of the PA is copied into the TxR and one clock cycle later the content of the RxR is copied into the PB. The *Ready* bit is cleared and transmission over TDI and TDO is not interrupted. The processor then has another 16 clock cycles to load the PA, read the PB and set the *Ready* bit.

Another possibility is that the PA is empty ( $Ready=0$ ). Transmission is then interrupted and the *Wait* bit is set to request service from the processor. Waiting for the processor to read the RxR and load TxR is indicated by a dashed rectangle in Figure 3.4. The test bus is in the pause state during the wait period.

Once the processor finishes the read/write process, it clears the *Wait* bit to allow the FSM1 to transfer another 16 bit of information. The *Finish* bit is set upon the completion of the DTD test, i.e., when the TC reaches zero.

The branch labeled DRC is selected when deterministic test data are used and the test results are compressed in the RxR. The volume of information flow between the memory unit and test channel is reduced by half over the DTD operation.

The branch labeled INS is followed when transmitting instructions. The content of TC is set to 1. The operations of the test channel are similar to that for DTD operations except that the sequence of values on the TMS line is different.

The branch labeled RunTest is used when the *RunTest* operation is required. The test channel transmits a specific sequence as specified by the boundary scan protocol over the TMS line such that all L0-slaves connected to the selected signal TMS<sub>i</sub> will enter the *Idle/RunTest* state for  $t$  clock cycles. The *Finish* bit is set before returning to the *idle* state again.

The loop conditions depend on the conditional signals ( $TC = 0$ ,  $SC = 0$ ,  $DC = 0$  and  $DC > 1$ ) generated by counters TC, SC and DC. The processor can stop or disable the operation of the FSM1 by loading a new word into the CR through a processor write operation. Resetting the *FSMen* bit will halt the operation of the FSM1. To maintain consistent operations, modifications of all other registers, except PA, PB, TxR and RxR, are prohibited until the *Finish* bit is set or an interrupt has occurred.

### 3.2.2 Bus Driver/Receiver

The Bus Driver/Receiver (BDR) is a bidirectional interface to the local bus of the MMC. It provides the driving capability for signals to/from the expansion bus. Figure 3.5 shows the basic architecture of the BDR. Signals *in* and *out* control the flow of information between the local bus and the expansion bus. These two signals are decoded from the address and control busses, which are subbusses of the local bus. When the addressed unit is not directly tied to the local bus, the BDR is used

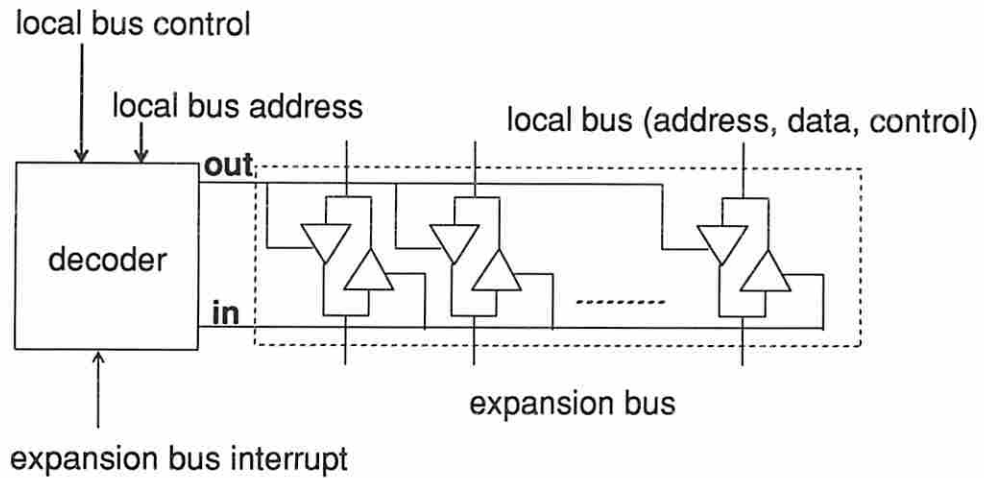


Figure 3.5: The Bus Driver/Receiver.

to select the appropriate unit on the expansion bus. To enable external interrupts from units tied to the expansion bus to reach the local bus, the expansion bus interrupt signals can also assert the *in* signal.

### 3.2.3 Functional Bus Interface

The Functional Bus Interface (FBI) allows communications between the module's functional bus and the MMC's expansion bus. Through the FBI, the MMC can execute functional tests for the module. Details of this interface will not be presented here. Further information on related interfacing techniques can be found in [10].

### 3.2.4 Testability Register

This is a 16 bit register used to increase the testability of modules containing chips which are either not designed to be testable, or do not have a test bus interface. The boundary scan registers on testable chips can be used to increase the testability of non-testable chips. However, in many cases, no boundary scan registers can be found to access signals between non-testable chips. The Testability Register can be



used to increase the testability of these chips and their signals in the following way. Signal points which need to be controlled (*C*) and/or observed (*O*) are cut and fed into the Testability Register. The *O signals* are connected to *C signals* during normal operation (see Figure 3.6). In test mode, the processor writes a word to the testability register which in turn applies this data to the *C signals*. Signals which need to be observed (*O signals*) are loaded into the testability register and then read by the processor. Thus both the controllability and observability of these cut points are enhanced. A technique for selecting these signal points is presented in [17].

### 3.2.5 Analog Test Interface

This circuit is used when there are analog circuits on the module under test (see Figure 3.7). To generate an analog signal, the processor writes a word to the analog test interface, the D/A converter then converts this data into an analog signal. For observability, an analog signal is converted into a digital word which can then be read by the processor.

### 3.2.6 L1-Slave

The MMC communicates with its higher level SuMP controller via an L1-bus, thus it must have an L1-slave. The design of a TM-slave is given in [13].

### 3.2.7 Processor

The processor's functions can be classified into five categories: (1) transfer data between memory and test channels; (2) transfer data between memory and an L1-slave; (3) compare test results with expected results; (4) transfer data between memory and expansion units; and (5) execute test and/or diagnostic programs.

A general or special purpose 16 bit processor can be used in the MMC. It controls all other units in the MMC. Through *read/write* operations, the processor

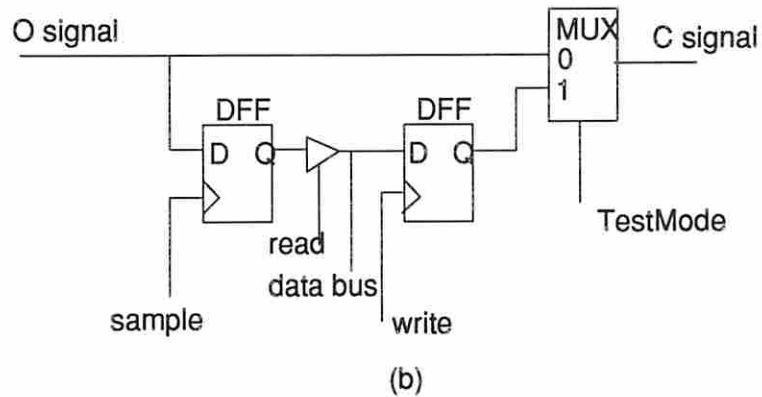
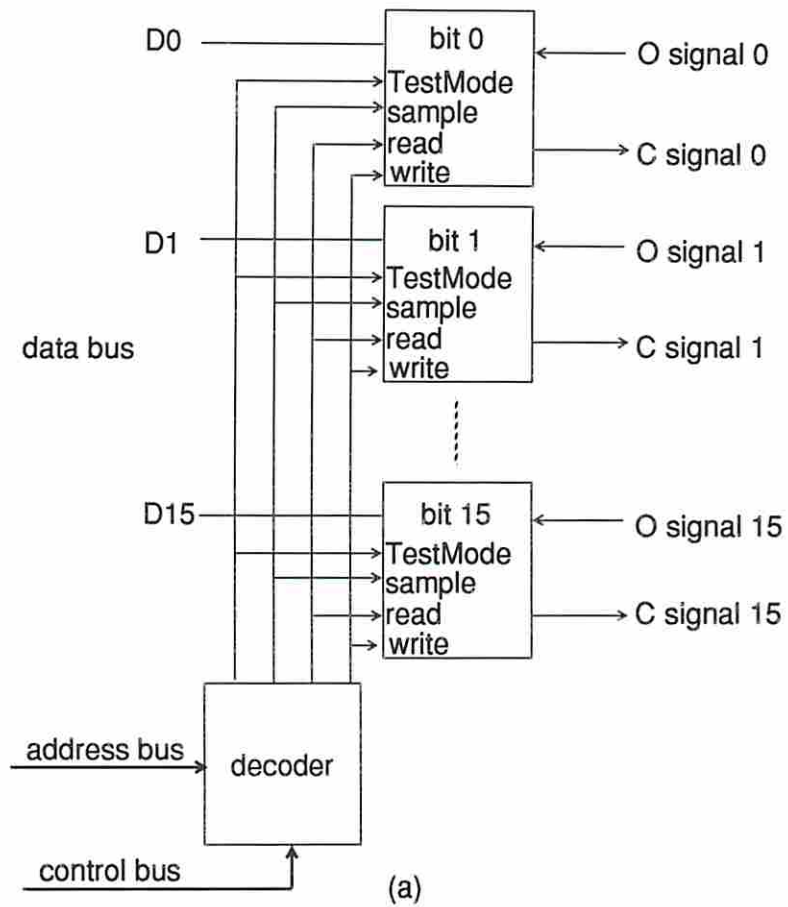


Figure 3.6: A Testability Register; (a) block diagram (b) circuit of bit  $i$ .



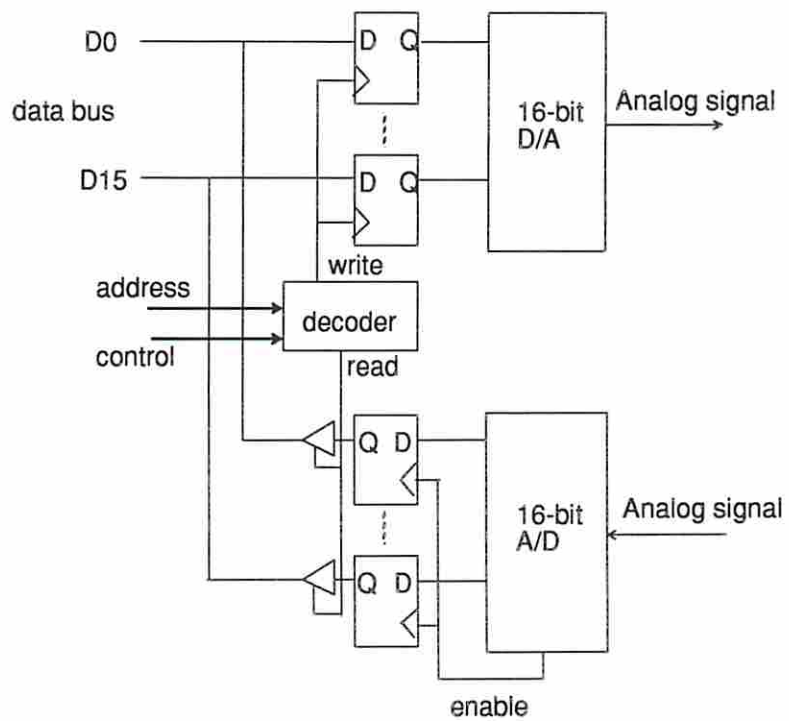


Figure 3.7: Analog Test Interface.

can access internal registers of a peripheral device, such as the L1-slave and test channels. Operations of a peripheral device can thus be controlled by a processor write to the control register of the peripheral device. Data exchange between memory and a peripheral device are controlled by processor *read/write* operations. Any processor that can execute the instruction set shown in Table 3.2 is powerful enough for the application of an MMC.

| instruction              | meaning  |
|--------------------------|--|
| <i>LDA R<sub>i</sub></i> | Load Acc with <i>R<sub>i</sub></i>                 |
| <i>LDA M</i>             | Load Acc with memory (M)                           |
| <i>STA R<sub>i</sub></i> | Store Acc to <i>R<sub>i</sub></i>                  |
| <i>STA M</i>             | Store Acc to memory (M)                            |
| <i>ADD R<sub>i</sub></i> | Add <i>R<sub>i</sub></i> to Acc                    |
| <i>AND R<sub>i</sub></i> | Bitwise And <i>R<sub>i</sub></i> with Acc          |
| <i>CMP R<sub>i</sub></i> | Compare Acc with <i>R<sub>i</sub></i>              |
| <i>NEG</i>               | Complement Acc                                     |
| <i>CLA</i>               | Clear Acc  |
| <i>BRZ R<sub>i</sub></i> | Branch to ( <i>R<sub>i</sub></i> ) if Acc not zero |
| <i>JMP R<sub>i</sub></i> | Jump to ( <i>R<sub>i</sub></i> )                   |
| <i>PUSH</i>              | Push Acc onto Stack                                |
| <i>POP</i>               | Pop Acc from Stack                                 |
| <i>NOOP</i>              | No operation                                       |
| <i>HALT</i>              | Halt the processor                                 |

Table 3.2: Processor instruction set.

The minimal architecture for a processor which is able to execute the above instruction set consists of an accumulator, four general purpose registers, an ALU, a program counter, a program status word, a stack with at least 4 words, an interrupt circuit, and a microprogrammed control unit.

If the MMC is implemented as a single chip ASIC, two additional instructions are useful to increase the data transfer efficiency between the memory unit and the test channel. The added instructions are *MR* (Multiple Read) and *MRMW* (Multiple Read and Multiple Write). The signal lines *Direct*, *Finish*, and *Ready* are used exclusively to support these two instructions (see Figure 3.8). Signal *Direct* is active when the microcontroller is executing any one of these two instructions.

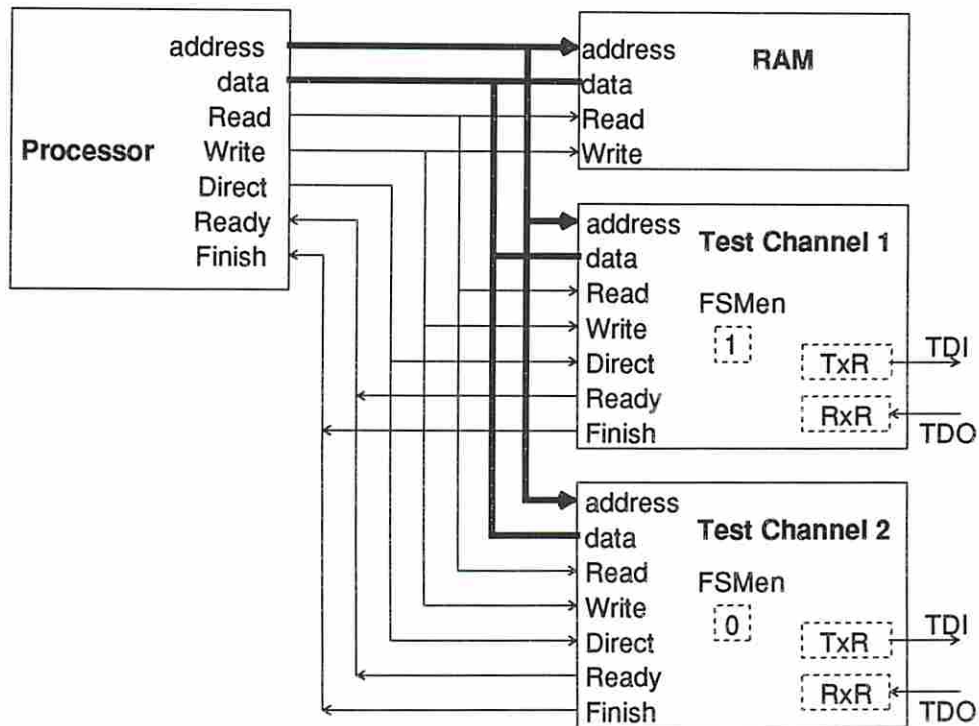


Figure 3.8: Control signals for MR and MRMW instructions.

Signals *Finish* and *Ready* are used as conditional signals for the microcontroller of the processor. All *Ready* (*Finish*) signals from test channels are wired-ORed together.

When executing an *MR* instruction, the processor waits until the *Ready* signal is cleared and then issues a *read* operation to the memory location addressed by the general purpose register R0. Meanwhile the test channel with *FSMen* bit set and operation mode being either DTD, DRC or INS can generate a load PA signal using signals *Direct* and *Read*. Thus a data word is moved directly from memory to a test channel. The value of R0 is increased by one after each read. The processor waits for the *Ready* signal to be deactivated, and then issues another *read* operation. This process is repeated until the *Finish* signal is set. Thus a block of information can be moved from the memory unit to the selected test channel and transmitted to a chip without any interruption.

When executing an *MRMW* instruction, the processor waits until the *Ready* signal is deactivated and then issues a *read* operation to the memory location addressed by R0. Meanwhile, the enabled test channel generates a load PA signal, and the data word from memory is loaded into the PA. The value of R0 is increased by one. The processor then issues a *write* operation to the memory location addressed by R1; meanwhile the enabled test channel generate a read PB signal, and a data word is read out of the PB and sent to the memory. The value of R1 is incremented. The processor waits for the *Ready* signal to be deactivated again and then issues another *read/write* operation. This process is repeated until the *Finish* signal is set. Thus a block of deterministic test data is moved from the memory unit to the selected test channel, and a block of test results is moved from the selected test channel to the memory unit.

### 3.2.8 Memory

The memory unit in an MMC is composed of a RAM unit and a ROM unit. The ROM unit contains test programs to test the entire module. These programs are compiled separately before testing. Some crucial information about the chips on the module is stored here. This information includes the number of chips to be tested, ordering of chips along the test bus ring, number and length of scan chains for each chip, number of random test vectors to apply to each chain, test instructions for each chip's CMC, TPG seeds and good signature for each test session. MMC functional self-test programs can also be stored in this unit. If the MMC is implemented using commercial ICs, then these programs are essential for MMC self-test. The Expansion ROM can be added whenever the module requires a large test program.

The RAM unit provides scratch pad memory for test program execution. Response signatures are stored here for latter evaluation. The RAM also provides storage for the Go/NoGo status for all chips, as well as for the entire module.

### 3.2.9 Stand-Alone MMC

The MMC can be used as a stand-alone mini ATE, provided that extra storage and console capabilities are added. For this application, a Console Control Interface and Disk Interface can be added to the MMC.

## 3.3 MMC Self-Test

If the MMC is implemented with an off-the-shelf “non-testable” processor, ROM and RAM, then some form of functional self-test is required. After finishing self-test, the MMC then reports its status to the control console or to a SuMP.

An MMC can also be tested either by an ATE or by another MMC. In the first case, an ATE can access the expansion bus of the MMC under test. The ATE invokes the self-test program of the MMC under test and waits until its completion. The test results, which are stored in the RAM, are then read by the ATE. In the second case, an MMC uses its Functional Bus Interface to access the expansion bus of the MMC under test. Again self-test programs can be invoked. Test results can be read and interpreted by the monitoring MMC.

If the MMC under test is implemented as a custom testable ASIC, then we assume it has a CMC. The MMC can thus be tested by another MMC. All units in the MMC, such as the processor, RAM, ROM, test channel and L1-slave, must be designed to be testable and their BIT structure need to be accessible via the L0-slave.

Some of the testability features of the test channel are described next.

**Testable test channel:** The testable design features of a test channel are shown in Figure 3.9. Major combinational logic blocks are indicated by rectangles having dotted lines. Registers are indicated by rectangles having solid lines. Some logic is associated with these registers, since some are counters and LFSRs. Normal functional connections are not shown. Instead the two scan chains formed during self-test are shown. Scan chain 1 is the boundary scan chain. All I/O signals can



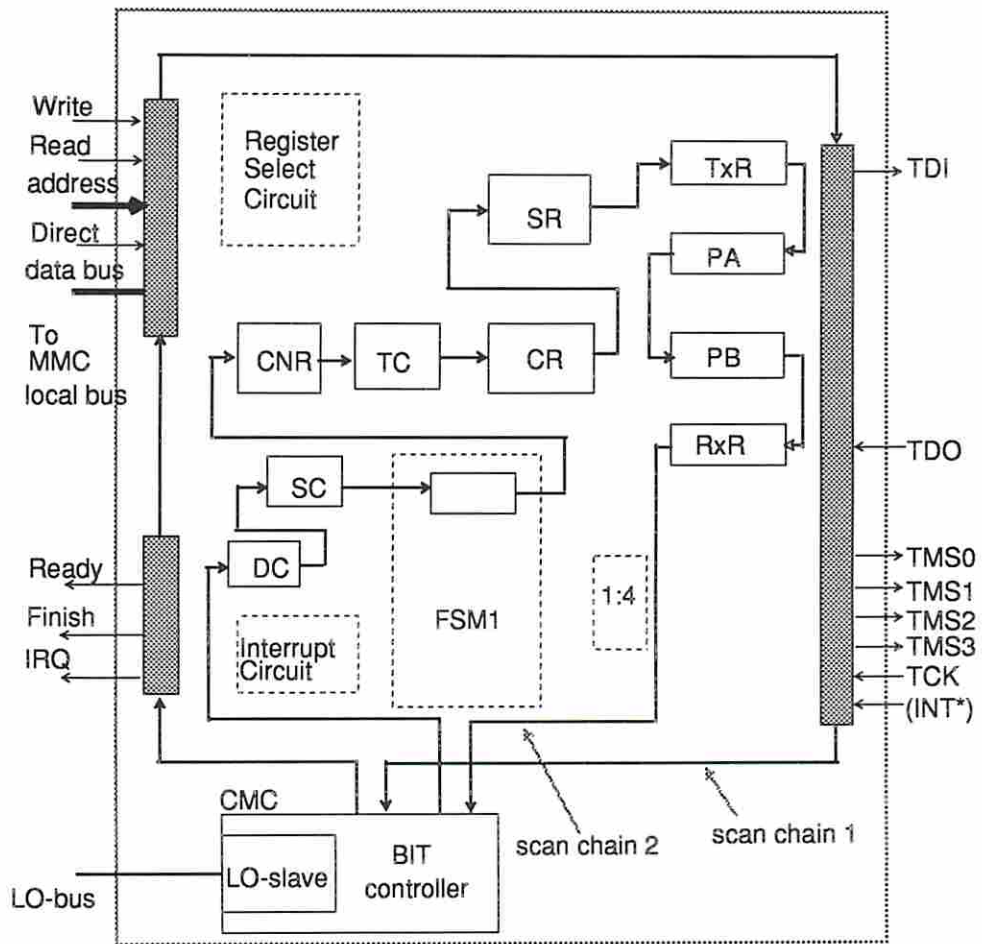


Figure 3.9: Testable design features for a test channel.

be controlled and observed by shifting test data or results along this chain. All other registers make up scan chain 2. The state of the test channel is controlled by shifting data along this scan chain. If a functional clock is activated, the next state of the test channel also can be observed by shifting out the content of this chain.

During testing, scan chain 1 is first loaded with test data which is held in place while the logic associated with scan chain 2 is tested. The module I/Os are tested using the boundary scan chains of this chip and those to which it is connected.

### 3.4 Discussion of MMC Design

An MMC design suitable for controlling the self-test process of a module has been described. The design uses the concept of test channels, which can run a test autonomously (in PTD case) once it is initialized by the processor. Because of the test channel, the processor need not deal with detailed control sequences over the JTAG boundary scan bus. Test execution sequences for chips can be generated in terms of processor *read/write* operations, which greatly simplifies the development of test programs.

The MMC architecture is expandable. More test channels can be added so that more chips can be tested in parallel. In addition, the MMC supports the functional testing of a module, the testing of clusters of chips which are not designed to be testable, and the testing of analog devices.

#### **Clock Synchronization:**

Four or more clocks may be applied to an MMC, viz. TCK for the CMC, FCK1 for the L1-slave, FCK2 for each test channel connected to an L0-bus, and FCK3 for the operation among processor and other peripheral devices. Synchronization problems will occur in a test channel where both FCK2 and FCK3 may access the same component, such as TxR and RxR. Techniques to solve this problem can be found in [39, 10]. In the design presented here, we use a common clock to drive all the clocks mentioned above, thus avoiding the clock synchronization problem.

**Portable Tester:**

The proposed MMC is designed to be part of a HTM system. It is assumed that each module contains an MMC, which under request from a SuMP can test all chips on the module and report back test results. However, it is possible to build an MMC as a portable stand-alone unit. In this case the L1-slave can be replaced by a control panel. A stand-alone MMC can test any module having an L0-bus. The module's built-in MMC is tested first through its L0-slave. Application chips on the module can be tested either by the built-in MMC or by the stand-alone MMC. For the latter case, the built-in MMC must be disabled to allow the stand-alone MMC to take control the module's L0-bus. An operator can start the test process via the control panel. Test programs stored in the ROM then take over control. After all chips have been tested, test results are shown on the control panel to indicate the Go/NoGo status of the module under test.

**Overhead:**

There are several ways of implementing an MMC. One or more test channels can be built on an ASIC chip. The processor, RAM and ROM can be implemented using standard chips. The other functions, which are optional, can be implemented using standard parts or an ASIC chip, excluding the expansion ROM. The application chip requires overhead to support testability, such as scan registers, as well as a L0-slave. For double latch designs, scan area overhead usually varies from 2.8 to 6.3%, depending on the ratio of gates to latches [52]. The overhead for an L0-slave depends on the length of the Instruction Register and the number of I/O pins. Assuming each shift register latch (SRL) is equivalent to 10 gates, an L0-slave with a 16 bit instruction register and a 60 bit boundary scan register requires about 1600 gates. For a 50000 gate ASIC chip, the total overhead for testability will typically be between 5-10%.

The boundary scan bus consists of 4 wires. Assuming 60 pins/chip prior to adding the bus, the routing overhead to support testability will be at least  $4/60 \cdot 100\% = 6.7\%$ . This is a lower bound since most pins on a chip are tied to only 2 or 3 point nets, while the test bus goes to all IC's. The wiring overhead is estimated to be closer to 10%.



**Fault Isolation:**

One of the important attributes of boundary scan is the ability to test the interconnect between chips. Assuming chips are also designed to be testable via DFT or BIST techniques, the MMC should be able to accurately locate hardware faults to a chip or interconnect.

**Analog Performance:**

Since the A/D and D/A conversion time is much smaller than the data transfer rate in the bus, the speed of observing or controlling an analog signal is determined by the data bus bandwidth. For example, an Intel 80186 processor running at 8MHz clock rate can transfer 4 MByte data from memory to the Analog Interface in 1 second.

## 3.5 An MMC Prototype

The implementation of an MMC prototype is described in this section. The major components of an MMC includes a processor, a memory unit and a Test Channel. The MMC prototype has been successfully used to execute the test procedures for a test chip. Programs that describe these test procedures are easy to develop since they can be written in a high level languages such as C.

### 3.5.1 Test Channel

The Test Channel is implemented using the Actel field programmable gate array (FPLA) technology. Some design changes have been made in the implementation. The main reasons for these changes are 1) the limited capacity of the device; 2) a change with the clocking scheme; and 3) the addition of DFT facilities. A detailed description of the changes in the design can be found in [44].

The implementation was aided by the Actel Action Logic System, which automatically performs placement, routing and the programming of ACT1020 devices. The Test Channel uses an ACT1020 device which is packaged in an 84 pin Plastic Leaded Chip Carrier (PLCC). The module utilization of this device is high. 513

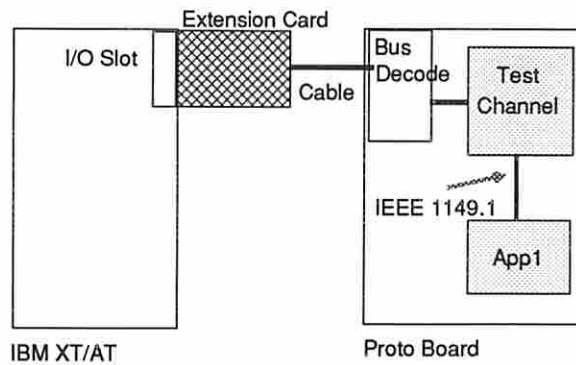


Figure 3.10: Physical configuration of the MMC prototype.

out of a total of 548 logic modules are used, and so are 47 out of 67 I/O modules. This chip can operate at 2.5 MHz and consumes less than 250 mW of power.

### 3.5.2 Processor and Memory

An IBM AT computer is used as the host computer of the prototype. It provides both the processor and the memory units required in an MMC. The physical configuration of the prototype is illustrated in Figure 3.10. A board which occupies a bus slot in the IBM AT is used to provide an extension of the I/O bus. Another board called the proto-board and developed at Stanford University [23], is used to decode the bus signals and to accommodate the Test Channel chip.

### 3.5.3 Processor and Test Channel Interface

Interfacing the Test Channel with the host processor requires very little effort. Only the following signal lines need to be connected: a 16 bit data bus, a 4 bit address bus, a chip enable line and two read/write control lines. These lines are available on the I/O bus of the host. Through these lines, the host can control the Test Channel by executing I/O read/write operations.



The proto-board provides I/O connection and bus decoding logics for interfacing with an IBM XT or AT computers that can serve as a host for the Test Channel chip. The width of the data bus on the proto-board is 8 bit only, which is incompatible with the 16 bit design of the Test Channel chip. Hence, a data bus adaptor that provides data buffering between the 8 bit and the 16 bit bus is required.

Figure 3.11 shows the data bus adaptor used in the prototype. The signal lines available on the proto-board include HD[7:0], HA[3:0], /PIOR, /PIOW and /POR. HD[7:0] is the 8 bit data bus from the host. HA[3:0] is the 4 bit address bus from the host. The /PIOR and /PIOW signal lines are derived from the host signal lines /IOR, /IOW and address lines. Both /PIOR and /PIOW are active only when I/O is selected in the address range from 300H to 30FH.

The signal lines of the Test Channel chip that need to be controlled by the host are PD[15:0], PA[3:0], /RD, /WR, /CS and Reset. PD[15:0] is a 16 bit data bus. PA[3:0] is a 4 bit address bus that selects the internal register to be accessed. A data buffer consisting of two 74LS373 is used to interface PD[15:8] with HD[7:0]. A bus transceiver 74LS245 is used to interface PD[7:0] with HD[7:0].

The address PA[3:0]=1111 is reserved for accessing the data buffer, which is used to store the high byte data. Two write operations are required to move a 16 bit data from the host to an internal register of the Test Channel. The first write operation, which is "outp(0x30f, high\_byte\_data);" in Microsoft C, loads the high byte data into the data buffer. The second write operation "outp(0x30i, low\_byte\_data);" loads both the high byte and low byte data into the internal register that is addressed by PA[3:0]=i. Similarly, two read operations are required to move a 16 bit data from an internal register of the Test Channel to the host. The first read operation "low\_byte\_data=inp(0x30i);" moves the low byte of the internal register into the host data bus and the high byte into the data buffer. The second read operation "high\_byte\_data=inp(0x30f);" moves the high byte from the data buffer into the host data bus.

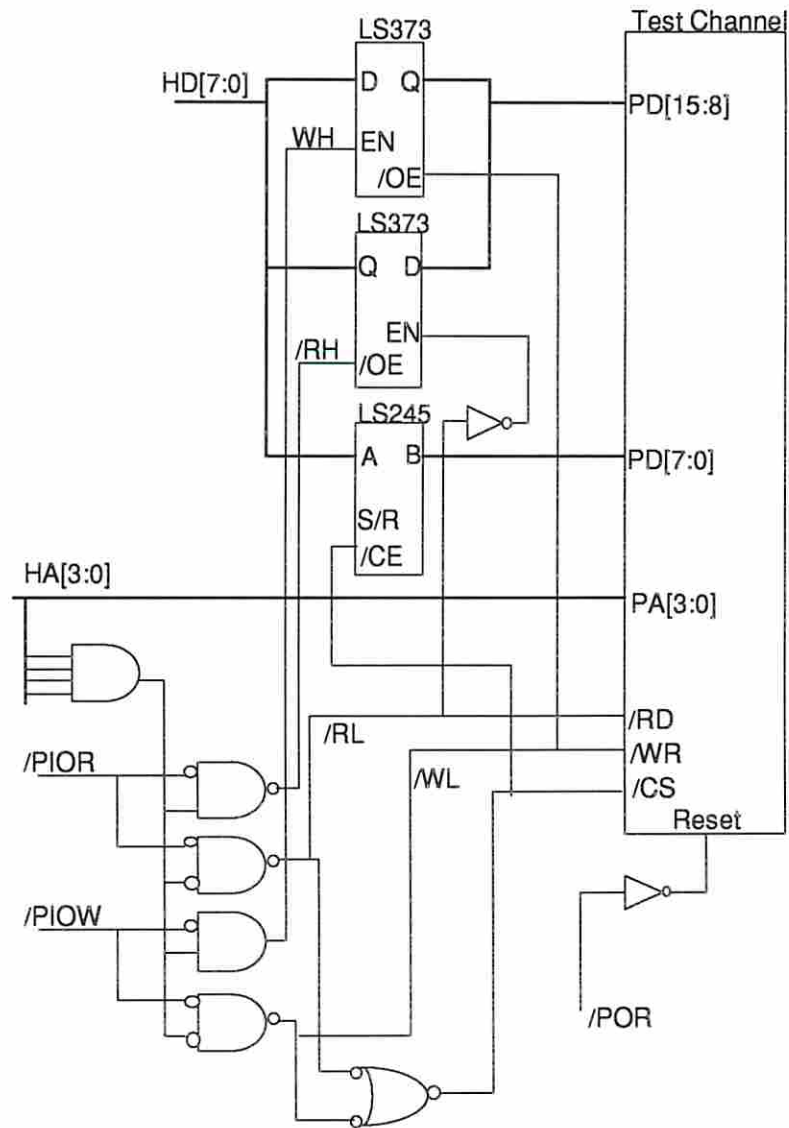


Figure 3.11: The data bus adaptor.

### 3.5.4 Discussion

The prototype has been successfully used to execute the test procedures for a test chip. Since the programs describing these test procedures are written in a high-level language like C, they can be easily developed and designed. Also, the prototype has been implemented using a PC, which costs much less than a conventional ATE. Furthermore, the performance of the prototype is superior to an ATE in testing boundary scan devices thanks to the Test Channel chip.

Using the developed Test Channel, it is possible to implement a minimal MMC by adding two chips such as an off-the-shelf RAM chip (e.g. Hitachi 6116), and a micro-controller (e.g. Intel 8048). Thus at most three chips are required to make a board completely self-testable.

The major drawback of the field programmable gate array technology is its limited capacity. Because the maximal capacity of the selected device (ACT1020) is less than 2000 gates, many functions in the original designs have been omitted. It is possible to implement the MMC using a different technology that has a larger capacity than an ACT1020, such as the ACT 2 devices. In such case, the performance of the MMC can be improved as follows.

1. Add the PA and PB registers, and the DC counter, omitted in this implementation, to the Test Channel.
2. Increase the length of the two counters TC and SC to account for a larger number of test vectors and more bits in each vector.
3. Incorporate a memory control circuit so that the Test Channel can access a local RAM. In this way it is possible for the Test Channel to send a long sequences of instructions and data without interruption.
4. Integrate the processor, the Test Channel and a memory unit into a chip. Handshaking circuit among the processor and the Test Channel can be avoided since they can be designed to be synchronous. However, to fit into a chip, the size of the processor instruction set and the on-chip memory should be bounded.

## 3.6 Testing a Kernel Using MMC and CMC

To test an application circuit consisting of one or more kernels, both the MMC and the CMC are required. Through a test bus the proposed MMC controls the execution of and provides all necessary test data for the testing of a chip having a CMC. During test mode the control signals of a kernel must be activated in the sequence specified by its associated control graph.

Figure 3.12 shows how the test control signals for a kernel can be generated and controlled by an MMC with the help of a CMC consisting of a TAP and a BIT-controller. The signals  $C_i$ ,  $C_j$ ,  $C_k$  are controlled by the state signals *Capture*, *Shift*, *Update*, *RunTest* corresponding to their associated TAP controller states, which are in turn controlled by the TMS line. The value of the TMS line is determined by the state of the FSM1 in the test channel. The FSM1 can control the TAP controller to any desired state via the TMS line. When sending data to the kernel, a predefined sequence of state transitions is generated by the FSM1 to activate the TAP states such that data can be received. Several predefined sequence of state transitions have been built into the FSM1. The processor in the MMC can select a sequence by loading proper data into the internal registers of the test channel. The loading of a register in the test channel is achieved by controlling the signals */CS*, */WR*, */RD*, *PA*, *PD*. These signals can be controlled by the processor executing a piece of code that contains I/O instructions.

By executing a program stored in the memory unit of an MMC, the processor can control the test of a kernel in the application circuit. The program directs the processor to control the test channel by loading its internal registers. The FSM1 is then activated and produces the proper sequence of values on the TMS line, which drives the TAP controller to appropriate states. This activates the state signals and, with the help of the BIT controller, the control signals  $C_i$ ,  $C_j$ ,  $C_k$  are activated according to the control graph. The kernel is thus tested.

Writing a program to test a kernel is a difficult problem since it involves a high degree of complexity resulting from many details at various level. To solve this problem a test program synthesis technique is used. This technique is presented in

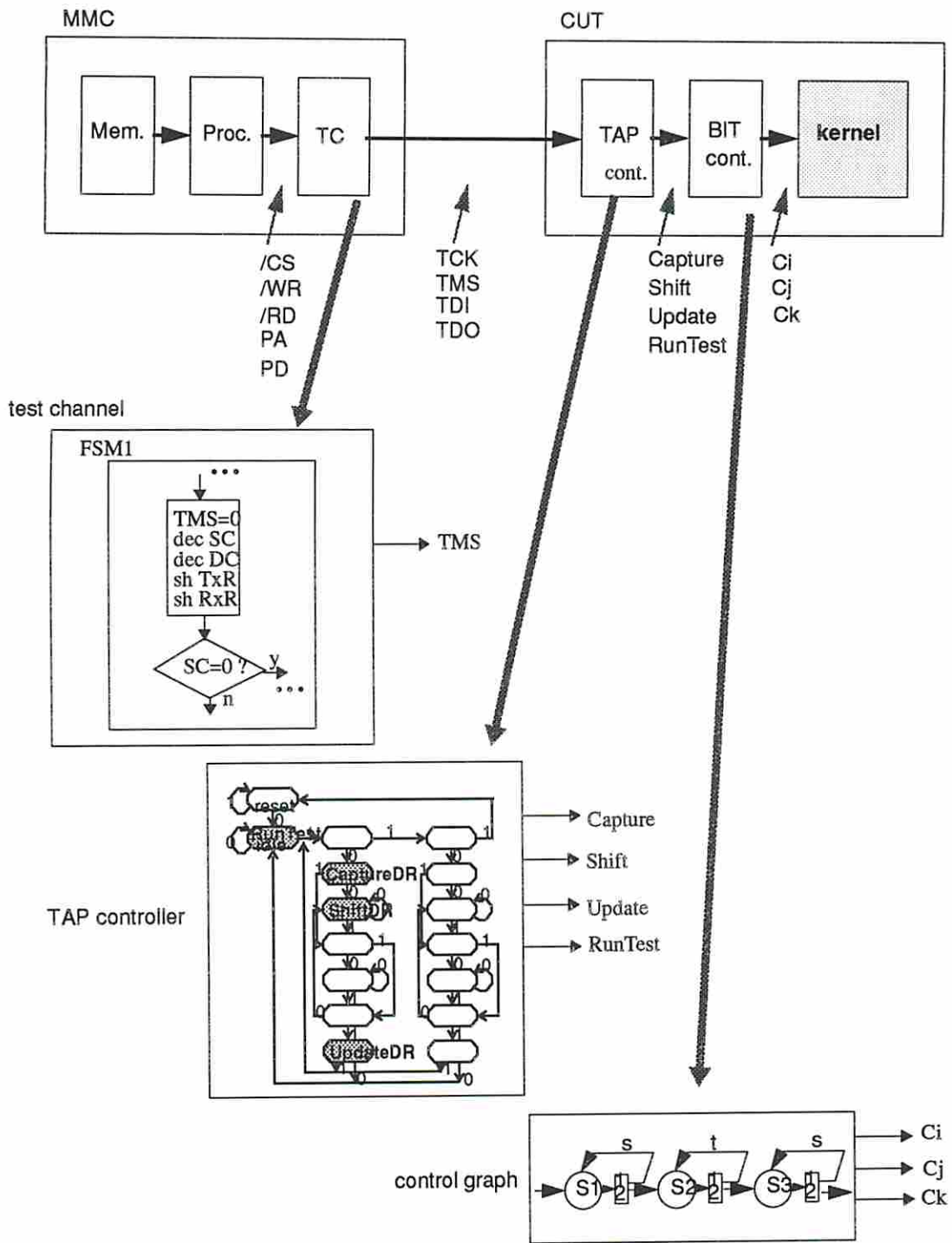


Figure 3.12: Overview of the test control.



chapter 4, where test description languages are provided for describing test procedures at both chip and module levels. Software tools are also provided so that test programs can be automatically synthesized from files written in these languages. The synthesized test programs are compiled and loaded into the memory unit of the MMC. The processor can then test the kernel by executing the test programs.

## Chapter 4

### Test Program Synthesis

When the proposed design methodology is properly adopted, test programs for the system can be easily synthesized. This is one of the most important aspects of the BOLD system. The ability to synthesize the test programs for a system represents a major advance in the reduction of the test development time.

The relationship between the test hardware and software in an HTM system is shown in Figure 4.1, where four axes are used to represent the hardware assembly units, the test description languages, the test programs, and the test controllers, respectively. Each axis again is represented by a hierarchy of four levels. The top hardware assembly unit is a system, which consists of several subsystems, which again consists of several modules, which again contains many chips. Each hardware assembly unit has a test controller associated with it. These test controllers include the system maintenance processor (SMP), the subsystem maintenance processor (SuMP), the module test and maintenance controller (MMC), and the chip test and maintenance controller (CMC). The test programs are classified into four levels according to their applications to the hardware units. These are the system test program (STP), the subsystem test program (SuTP), the module test program (MTP), and the chip test program (CTP). The test languages used to describe the test aspects of a hardware unit includes the system test language (STL), the subsystem test language (SuTL), the module test language (MTL) and the chip test language (CTL).

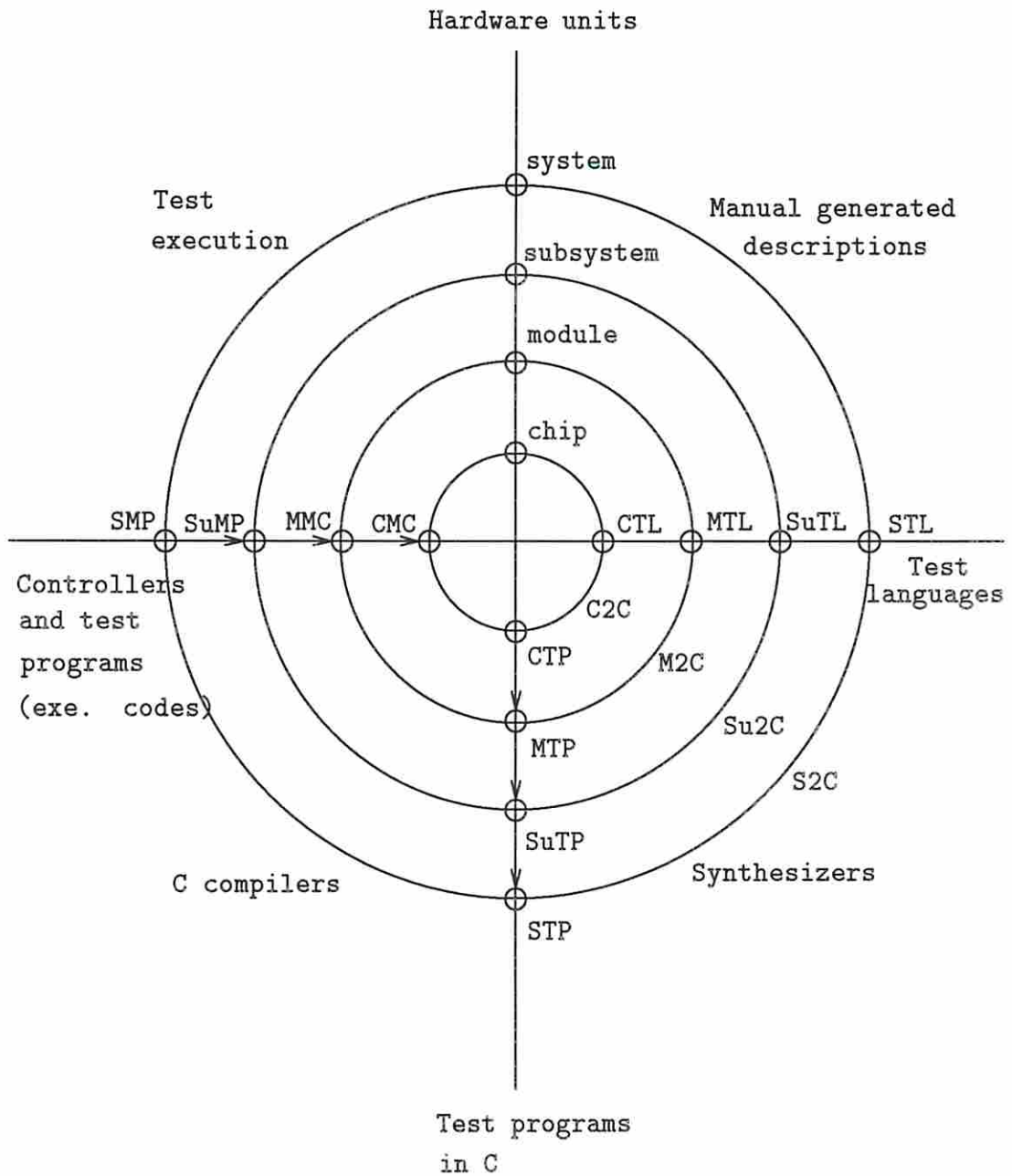


Figure 4.1: Overview of the test hardware/software hierarchy.

The synthesis process starts with the preparation of the test description files by a designer. These files are constructed using high level description languages that are easily understood by designers with little or no knowledge of testability. A set of synthesizers are then used to translate the input files into appropriate formats (which is in C language) for each hardware unit. These test programs are then translated down to executable code for the test controllers by an appropriate C compiler. A test controller can then test its associated hardware unit by executing the loaded executable codes. In such a manner, the entire system can be tested.

The major advantages of the synthesis approach are (1) consistent test methodology, that is, a chip is tested using the same test set during the chip test, module test, subsystem test and system test; (2) reduced time, effort, and errors in test program development; (3) test programs can be prepared by designers with little knowledge of testability; (4) interconnect testing is included automatically.

This chapter is organized as follows. In section 4.1 the test description languages are presented. In section 4.2 the test program synthesizers are described. In section 4.3 an example is used to illustrate the synthesis of test programs. In section 4.3 an example is used to show the details of test program synthesis procedure. In section 4.4 results of synthesizing several modules are presented.

## 4.1 Languages

Four set of languages are required in describing the test aspects of a system, namely Chip Test Language (*CTL*), Module Test Language (*MTL*), Subsystem Test Language (*SuTL*) and System Test Language (*STL*). Currently, only *CTL* and *MTL* have been developed and are supported. These two languages are described next.

### 4.1.1 CTL - The Chip Test Language

Due to its wide acceptance, the BSDL [55] has been adopted as the framework of the CTL. Effort is currently under way to make this language a new IEEE standard

to go along with the boundary scan standard. The syntax of the BSDL follows that of VHDL [32], which has been widely accepted as a hardware description language.

A very brief description of BSDL is given here. The BSDL can be used to describe the testability information of a boundary scan device (or chip) that conforms with the IEEE Std. 1149.1. The information described by the BSDL includes three major parts: *the pin I/O*, *the Test Access Port (TAP)* and *the boundary register*. The pin I/O part contains the definition of the logical port, the package pin mapping, and the definition of the scan port. The TAP part describes the instruction and the instruction register (IR), the identification register (ID) and the other registers that can be accessed. The boundary register part characterizes the cell type of each boundary cell, the ordering of the cells in the register and the control information for tri-state and bi-directional cells. A detailed description of the BSDL can be found in [55].

The BSDL describes the information about the on-chip test hardware; however, it does not provide the information about how the chip can be tested. To circumvent this problem, the CTL includes a part called Test Procedure in addition to the original BSDL description. The Test Procedure provides the information required for testing the chip. The incorporation of the test procedure is achieved by adding a VHDL attribute called TEST\_PROC. The following example shows how a test procedure is incorporated in a CTL-file.

```
attribute TEST_PROC of app1 : entity is
    "Test_Begin" &
    "TDM 1 = FULLSCAN;" &
    "REG=FBR, VECFILE=ap1_in2, RESFILE=ap1_out2;" &
    "REG=BOUNDARY, VECFILE=ap1_in1, RESFILE=ap1_out1;" &
    "CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;" &
    "Test_End ";
```

It is also possible to omit this attribute and describe the test procedure in a separate file, where the quotes and & are omitted. For example, the above test procedure can be written in a file called `app1.ctp` as follows.



```

TEST_BEGIN
TDM 1 = FULLSCAN;
REG=FBR, VECFILE=AP1_IN2, RESFILE=AP1_OUT2;
REG=BOUNDARY, VECFILE=AP1_IN1, RESFILE=AP1_OUT1;
CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;
TEST_END

```

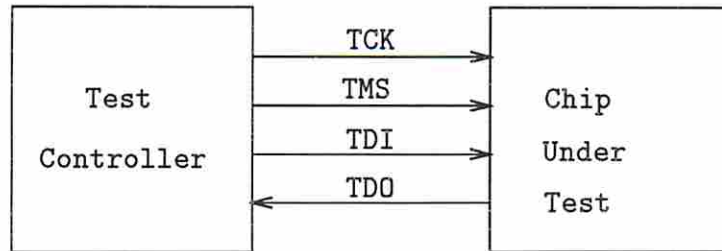


Figure 4.2: Test control model used in CTL.

The test control model used in CTL is shown in Figure 4.2, where a test controller can execute the test process of the chip under test (CUT), which is also referred to as the device under test (DUT), via a four-line boundary scan test bus. A Test Procedure consists of one or more test sessions. During a test session different parts of the chip are tested according to a predefined methodology, which is called a Testable Design Methodology (TDM). The procedure of testing a circuit designed with a specific TDM is referred to as a TDM in the Test Procedure. The procedure can be described as either template-based or user-defined TDMs. A template-based TDM is used to describe the procedure to test a circuit designed with a commonly used TDM such as fullscan or BILBO. A user-defined TDM, on the other hand, is used to describe an arbitrary procedure composed by the user using C codes or some test-specific statements provided by the CTL. A more detailed description of these TDMs follows.

### Template-Based TDMs

The template-based TDMs that are currently supported include: *Fullscan*, *FullscanN*, *BILBO*, *RUNBIST*, and *INTEST*.

- *Fullscan TDM:*

The circuit under test is designed with the full scan technique, where all storage elements in the circuit are made scannable and are cascaded to form a scan chain. The circuit thus is observable and controllable via the scan chain. The fullscan design structure of a circuit is shown in Figure 2.8, where the control graph for this circuit can also be found. A circuit designed with the fullscan technique can be tested using the following procedure.

Procedure to test a circuit using the Fullscan TDM:

1. Load a vector to the scan chain by shifting  $s$  times
2. Repeat for  $t - 1$  times  
Update the scan chain by applying a functional clock.  
Shift out the result while shifting in next vector.
3. Get the last result by shifting  $s$  times.

In addition to the number of test vectors ( $t$ ) and the length of each vector ( $s$ ), the test controller also needs to know where to get the test vectors and the correct response vectors such that they can be compared with the test responses. An example of the Fullscan TDM described in CTL is shown as follows.

```
TDM <tdm_id> = Fullscan;  
REG = <reg1>, VECFILE= <file1>, RESFILE= <file2>;  
CLOCK= FCK <number1> CYCLES_IN RUN_TEST_IDLE;
```

The <tdm\_id> identifies the TDM in a CTL-file. The selected scan register is <reg1>, which must be previously defined in the CTL-file. The test vectors are stored in the file <file1> and the expected result vectors are stored in the file <file2>. The number of test vectors and the length of each vector is contained in the file <file1>. After a test vector is loaded into the scan register <reg1>, it is necessary to apply <number1> cycles of FCK clock to the circuit under test before the test results are available for shifting out. Note that the test bus must be maintained in the RUN\_TEST\_IDLE state during the application of the clock FCK.

- *FullscanN TDM:*

This TDM is similar to the Fullscan TDM except that the scan registers are organized into more than one scan chain. All scan chains must be loaded with a new test vector before applying a system clock to capture the test result. The procedure for testing a circuit with full scan structure using multiple scan chains is as follows.

Procedure for testing a circuit using the FullscanN TDM:

1. Repeat for  $t$  times
  - 1.1 Load each scan chain with a test vector segment.
  - 1.2 Update all chains by applying a functional clock.
  - 1.3 Get a result segment from each scan chain.

Note that the steps 1.1 and 1.3 can be executed simultaneously, i.e. shifting in a new vector segment while shifting out a result segment, provided that all the scan chains can be updated simultaneously. However, this is not possible in the IEEE Std. 1149.1 protocol, where the *capture* state precedes the *shift* state. However, as shown in chapter 7, steps 1.1 and 1.3 can be overlapped for some scan chains if there is no data dependency between these chains. An in depth analysis of this problem is presented in that chapter. As shown in [45], information about the data dependency among the boundary scan chains is needed. If this information is not available, one can separate the operations of steps 1.1 and 1.3 so that no conflict exists. However, in this case the test application time is not necessarily minimal. An example of the FullscanN TDM in CTL is shown below.

```
TDM <tdm_id> = FullscanN;  
REG = <reg1>, VECFILE = <file1>, RESFILE = <file2>;  
REG = <reg2>, VECFILE = <file3>, RESFILE = <file4>;  
CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;
```

- *BILBO TDM:*

The circuit under test has been designed using the BILBO methodology and consists of one or more BILBO kernels. For each BILBO kernel, the seed

and the correct signature must be provided. In addition, the number of the pseudorandom test vectors that are applied is required. A circuit designed using the BILBO TDM is shown in Figure 2.10, where the control graph can also be found. The procedure for testing a circuit using the BILBO TDM is as follows.

Procedure for testing a circuit using the BILBO TDM:

1. Load the seeds into the BILBO registers.
2. Apply a number of cycles of test clocks.
3. Get the signatures from the BILBO registers.

Note that more than one BILBO kernels can be tested in a single session. An example of the BILBO TDM in CTL is as follows.

```
TDM <tdm_id> = BILBO;
INITIALIZE <reg1> = <value1>, <reg2> = <value2>;
USE_INSTRUCTION = <ins1>;
CLOCK = <TCK or FCK> <number1> CYCLES_IN RUN_TEST_IDLE;
EXPECTED_RESULT <reg3> = <value3>, <reg4> = <value4>;
```

It is necessary to load both registers <reg1> and <reg2> with the initial values <value1> and <value2>, respectively, at the beginning of the test. The instruction register (IR) of the TAP must be loaded with the instruction <ins1> during the execution of the test. After the execution of the test, the final signature in the registers <reg3> and <reg4> must be <value3> and <value4>, respectively, for a fault-free circuit.

- *RUNBIST TDM:*

The circuit can be tested using the public instruction RUNBIST defined in the IEEE 1149.1 standard. Once the RUNBIST instruction is loaded into the IR of the TAP, the self-test procedure can be executed by simply applying the test clock TCK during the RUN\_TEST\_IDLE bus state. The result of the test is stored in a register <reg1>. The <value1> represents the result that should be in the register when the circuit is fault free. An example of the RUNBIST TDM is listed below.



```
TDM <tdm_id> = RUNBIST;  
CLOCK = TCK <number> CYCLES_IN RUN_TEST_IDLE;  
EXPECTED_RESULT <reg1> = <value1>;
```

- *INTEST TDM:*

The circuit is tested using the INTEST instruction defined in the IEEE 1149.1 standard. The instruction INTEST must be loaded into the IR of the TAP before the execution started. This TDM differs from the Fullscan TDM in that only the boundary scan register is included in the scan chain and no internal storage elements are scanned.

Procedure for testing a circuit using the INTEST TDM:

1. Repeat for  $t$  times
  - 1.1 Shift a test vector into the boundary scan register.
  - 1.2 Apply one or more functional clock cycles.
  - 1.3 Shift the results out of the boundary scan register.

An example of the INTEST TDM is shown below.

```
TDM <tdm_id> = INTEST;  
VECFILE = <file1>, RESFILE = <file2>;  
CLOCK = FCK <number1> CYCLES_IN RUN_TEST_IDLE;
```

## User-Defined TDM

In addition to the normal C code, some additional C functions have been provided for describing user-defined TDMs. The following paragraphs describe these functions.

- *ScanIR(outS);*

The content of the instruction register (IR) of the TAP can be updated by executing this function. The string *outS* is loaded into the IR after this function is executed. When scanning in the new instruction, a string of values called status is also scanned out. The status is the logic value on the parallel



data input to the instruction register before the shifting started. By executing this function, a test controller can control the test process of a chip. The format of the instruction is determined by the chip designer except for those that have been predefined by the IEEE Std. 1149.1. The format of the status is also defined by the chip designer.

- *ScanDR(outS);*

This function is similar to the *ScanIR* except that the selected data register is scanned. The selection of the data register is determined by the current content of the instruction register, which can be altered by the *ScanIR* function. When scanning a new string of data, the resulting string from the selected data register is also scanned out. The result is the logic value on the parallel data input to the data register before the shifting started.

- *ApplyClock(tck, n);*

This function can be used to provide test clocks to the circuit under test. Both the test clock TCK and the functional clock FCK can be applied using this function. For example, if the value of *tck* is 1 (or true), then the test clock TCK is applied for *n* cycles while the rest of the I/O pins are kept unchanged. If the value of *tck* is 0 (or false), then the functional clock FCK, which may consist of several phases, is applied for *n* cycles while keeping the input to the rest of the I/O pins unchanged.

- *Bring2state(i);*

The execution of this function drives the bus to state *i*, defined in the IEEE Std. 1149.1, regardless of the current state of the bus. It is easy to conclude, from the state transition diagram, that putting a 1 on the TMS line for five consecutive clock cycles will bring the TAP controller into the Reset state. The TAP controller can then be brought to any state *i* from the Reset state. If more than one possible state transition paths exist, the one that goes through the smallest number of states will be taken.

- *State2state(i,j);*

The execution of this instruction drives the bus from state *i* to state *j*. Again, the path taken is through the smallest number of states.

- *RepeatState(i,n);*

The execution of this instruction enables the bus to stay in state *i* for *n* consecutive clock cycles. Note that this instruction can be applied to only some of the states. For the *shiftDR* and *shiftIR* states, this instruction can be modified to *RepeatState(i,n,Sout,Sin)*, where *Sout* is the string sent to the TDO line and *Sin* is the string received from the TDI line.

- *RunTest(n);*

The bus is driven into the `RUN_TEST_IDLE` state and held there for *n* consecutive test clock cycles. During this period, the on-chip test controller executes the predefined test process for the built-in self-test structures.

#### 4.1.1.1 Formal Definition of the CTL

The CTL is a superset of the BSDL. The formal definition of the BSDL can be found in [55]. The definition for the chip test program is listed below using the YACC [35] input format.

%%

```
ctl: BSDL chip_test_program;
chip_test_program: _TEST_BEGIN test_procs _TEST_END;
test_procs: test_proc | test_procs test_proc;
test_proc
    : _TDM _INT_NUM _EQ _RUNBIST _SEMICOLON runbist_tdm
    | _TDM _INT_NUM _EQ _INTEST _SEMICOLON intest_tdm
    | _TDM _INT_NUM _EQ _FULLSCAN _SEMICOLON fullscan_tdm
    | _TDM _INT_NUM _EQ _BILBO _SEMICOLON bilbo_tdm
    | _TDM _INT_NUM _EQ _USER_DEFINE _SEMICOLON user_def_proc;
runbist_tdm: clock_part result_part;
clock_part
    : _CLOCK _EQ _TCK nums _CYCLES_IN _RUN_TEST_IDLE _SEMICOLON
    | _CLOCK _EQ _FCK nums _CYCLES_IN _RUN_TEST_IDLE _SEMICOLON
    | _CLOCK _EQ _FCK _SHIFTED
```

```

        | _CLOCK _EQ _NONE;
nums: _FLO_NUM | _INT_NUM;
result_part: _EXPECTED_RESULT res_lists _SEMICOLON;
res_lists: res_list | res_lists _COMMA res_list;
res_list: _IDENTIFIER _EQ _BIN_NUM;
intest_tdm: _VECFILE _EQ _IDENTIFIER _COMMA _RESFILE _EQ
            _IDENTIFIER _SEMICOLON clock_part;
fullscan_tdm: reg_Flists clock_part;
reg_Flists: reg_Flist| reg_Flists reg_Flist;
reg_Flist: _REG _EQ _IDENTIFIER _COMMA _VECFILE _EQ _IDENTIFIER
            _COMMA _RESFILE _EQ _IDENTIFIER _SEMICOLON;
bilbo_tdm: initialize_part use_ins_part clock_part result_part;
initialize_part: _INITIALIZE ini_lists _SEMICOLON;
ini_lists: ini_list | ini_lists _COMMA ini_list;
ini_list: _IDENTIFIER _EQ _BIN_NUM;
use_ins_part: _USE_INSTRUCTION _EQ _IDENTIFIER _SEMICOLON;
user_def_proc: _TOP;
%%

```

Note that the user-defined TDM is not described since the statements in this TDM are directly translated by LEX [42]. In addition, the syntax of C is not listed. The interested readers are referred to the source code of the program.

### 4.1.2 MTL - The Module Test Language

The MTL is a high level language that can be used to describe the test aspects of a module. The language has been designed in such a way that little testing expertise is required to use it. A module consists of many testable chips and a test controller, which can access these chips via one or more test busses. The controller can test these chips and the interconnect between them. A typical test control model used in MTL is shown in Figure 4.3, where five chips are organized into two test rings. The test clock TCK, which is connected to every chip, is not shown.

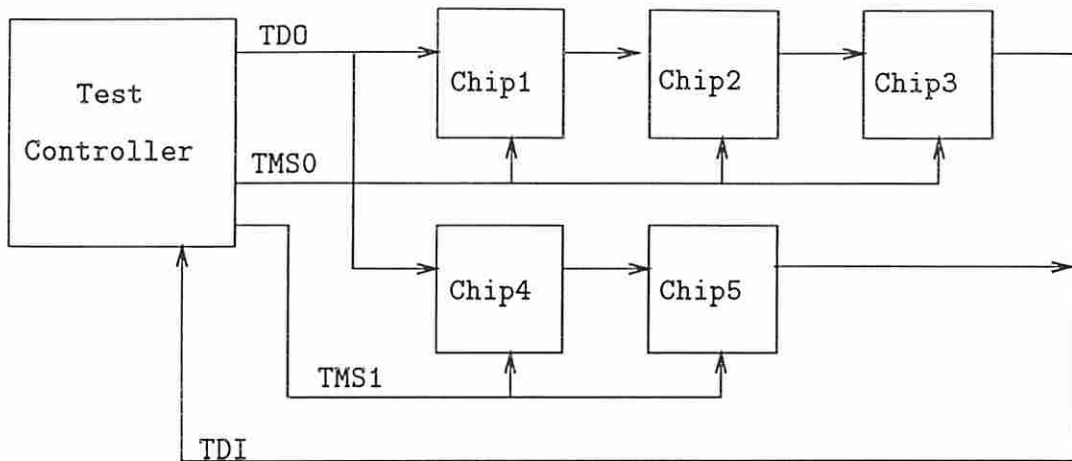


Figure 4.3: Test control model used in MTL

Each module is associated with an MTL-file written in MTL. This file contains all the information required for testing the module. When an MTL-file and its associated CTL-files are processed by the *synthesizer M2C*, a module test program is synthesized. This test program is written in C. With the help of a C compiler, the test programs can be translated to executable codes which can then be executed by the test controller. The entire module can thus be tested.

The test aspects of a module described by the MTL include the following parts, namely *library\_id*, *device\_list*, *test bus configuration*, *net\_list*, and *test procedure*. The *library\_id* points to the directory containing the CTL-files. The *device\_list* associates every device used in the module with a CTL-file in the library. An error is flagged when a device is associated with a CTL-file that does not exist in the library. The *device\_list* of an example module consisting of two devices is described as follows.

```
device_list =
(Chip1 adder)(Chip2 multiplier);
```

The *test bus configuration* describes how the devices on the module are connected via the boundary scan bus to the test channel. In the boundary scan architecture, the test bus can be configured as a ring, a star or a combination of both. In MTL a test bus is modeled as a multiple ring, which can be mapped into any



one of the above three configurations. A ring configuration is formed when only a single ring is used. A star configuration is formed when every ring contains only one device. All DUTs in a ring are controlled by the same TMS line. The test bus shown in Figure 4.3 is described as follows.

```
test bus =  
ring 0:  chip1 => chip2 => chip3,  
ring 1:  chip4 => chip5;
```

The *net\_list* describes how the devices on a module are interconnected functionally. The number of nets can be large. Two or more terminals are possible for each net. Each terminal is specified by two names, the first name gives the device name, while the second specifies the I/O port name or the pin number. A *net\_list* consisting of two nets is described below.

```
net_list =  
net 1:  (Chip1 inp1) (Chip2 outp1),  
net 2:  (Chip2 inp1) (Chip3 inp1) (Chip1 outp1);
```

The *test procedure* contains the information for testing a module by an MMC. This information is represented in terms of standard C code and some test-specific statements. These statements assume no knowledge about the test controller and can be translated to low level functions according to the architectural detail of the MMC. The low level functions are fully supported by a library of C code containing machine-dependent I/O functions. These I/O functions are used to control the test channel, which is responsible for all the low level activities in the boundary scan test bus.

If every chip used in a module conforms with the IEEE 1149.1 boundary scan architecture, it is possible for a designer to write the test procedure using only the test-specific statements. In such a situation, a test procedure can be easily written by a designer with little knowledge of testing, thus greatly reducing the program development time. However, if all chips used in the module do not conform with



the boundary scan architecture, then it is necessary for the designer to write the test procedure using C code in addition to the test-specific statements. In this case some knowledge about testing is required.

The test-specific statements that can be used to describe the test procedure in an MTL-file are listed below.

- *Testchip (chip\_id);*

A test controller, such as an MMC, can fully test a chip by executing this statement. If more than one session is required for testing the chip, the test results are reported only after all sessions are completed.

- *Testchip (chip\_id) Use TDM (tdm\_id);*

A test controller can test part of a chip by executing this statement. Usually, a chip may be tested in more than one session. During each session part of the chip is tested using a specific TDM. To fully test the chip, all test sessions must be executed. When a module under test cannot stay offline long enough to allow it to be fully tested, a partial testing approach is used. Also a piece of circuit can be tested several times using different test patterns or different seeds. In this approach, the chip is tested in different intervals. One or more test sessions are executed in each interval without exceeding the time limit.

- *TestInet();*

A test controller can test the interconnect on a module by executing this statement. Every net connecting two boundary scan chips is tested. The test set used in this test is a counting sequence which can determine if the entire interconnect is fault free. However, only the Go/NoGo information is produced in this test. No diagnostic information is provided.

- *DiagnosisInet();*

A test controller can test and diagnose the interconnect on a module by executing this statement. To achieve the maximal diagnostic resolution, the test set is a universal test set, which includes a walking ones sequence, a walking zeroes sequence, and the all-0 and all-1 vectors. As shown in chapter 6, all diagnosable faults can be identified by this test.

- *SampleRing(ring\_id);*

By executing this statement, a test controller can achieve a snap-shot of the logic values on the I/O pins of all chips that are connected to the selected ring. The returned value of this function is a string of 1s and 0s representing the current status of these chips.

- *ScanDR(ring\_id, preDR, postDR, outS, inS);*

By executing this statement, a test controller can exchange information with a selected data register in the DUTs on a selected ring specified by the *ring\_id*. The data sent out is the string *outS*. The data received, which is referred to as results, is a string *inS*. It is possible to exchange information with a particular DUT while bypassing all other DUTs in the ring. In this case the number of DUTs to be bypassed must be specified. In Figure 4.4, *DR* represents the data register of the selected DUT, *preDR* is the number of bypassed DUTs between the selected DUT and the TDO of the ring, and *postDR* is the number of bypassed DUTs between the TDI of the ring and the selected DUT. The number of shifts required for transmitting the string *outS* is calculated automatically once both *preDR* and *postDR* are known.

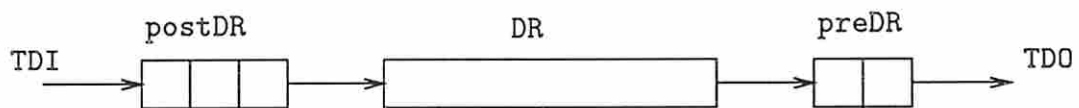


Figure 4.4: Scanning a data register in a ring.

- *ScanIR(ring\_id, preIR, postIR, outS, inS);*

This statement is similar to *ScanDR* except that the information transmitted is the instructions to the DUTs and the received results is the status of the DUTs. When sending instructions, all DUTs in the selected scan ring receive a new instruction. For those devices that are not being tested, the received instruction can either be a No-Op or the previous instruction.

- *ApplyClock(ring\_id, n);*

By executing this statement, a test controller can apply the test clock to the DUTs in a selected scan ring for a fixed number of cycles. The bus state of

the selected ring is kept in the *RunTest* state so that the last instruction can be executed. For example, when this statement is executed after the public instruction *RUNBIST* has been sent, the chips can be tested.

- *ChangeClockFreq(n);*  
By executing this statement, a test controller can change the frequency of the test clock TCK. The default TCK frequency can be divided by a factor specified by *n*.
- *EnableClock(on);*  
By executing this statement, a test controller can halt the application of the test clock TCK. When this statement is executed with *on* = 0, the test clock TCK is disabled (or halted); otherwise the test clock is enabled (or running freely). By default the test clock is always enabled.

#### 4.1.2.1 Formal Definition of the MTL Syntax

The formal definition of the module test language MTL is listed below. The language is described in the input format of the YACC.

```
%%  
mtl: conf_section test_section;  
conf_section : mtl_stmts;  
mtl_stmts: mtl_stmt | mtl_stmts mtl_stmt;  
mtl_stmt: module | lib | device_list | test_bus | net_list;  
module: _MODULE _EQ _IDENTIFIER _SEMICOLON;  
lib: _LIB _EQ _IDENTIFIER _LPR _INT_NUM _RPR _SEMICOLON;  
device_list: _DEVICE_LIST _EQ dev_pairs _SEMICOLON;  
dev_pairs: dev_pair | dev_pairs dev_pair;  
dev_pair: _LPR _IDENTIFIER chip_type _RPR;  
chip_type: _IDENTIFIER;  
test_bus: _TEST_BUS _EQ test_rings _SEMICOLON;  
test_rings: test_ring| test_rings _COMMA test_ring;  
test_ring: _RING ring_id _COLON device_chain;
```

```

ring_id: _INT_NUM;
device_chain: _IDENTIFIER| device_chain _RROW _IDENTIFIER;
net_list: _NET_LIST _EQ nets _SEMICOLON;
nets: net| nets _COMMA net;
net: _NET net_id _COLON pins;
net_id: _INT_NUM;
pins: pin| pins pin;
pin: _LPR _IDENTIFIER _IDENTIFIER _RPR;
test_section: _TEST_BEGIN mtp_stmts _TEST_END
mtp_stmts: mtp_stmt | mtp_stmts mtp_stmt;
mtp_stmt
    : test_inet
    | diagnosis_inet
    | test_chip
    | reset_ring
    | sample_ring
    | apply_clock
    | ch_clock_freq;
test_inet: _TESTINET _LPR _RPR _SEMICOLON;
diagnosis_inet: _DIAGNOSISINET _LPR _RPR _SEMICOLON;
test_chip: _TESTCHIP _LPR _IDENTIFIER _RPR _SEMICOLON
    | _TESTCHIP _LPR _IDENTIFIER _RPR _USE
        _TDM _INT_NUM _SEMICOLON;
reset_ring: _RESETRING _LPR int_num _RPR _SEMICOLON;
sample_ring: _SAMPLERING _LPR int_num _RPR _SEMICOLON;
int_num: _INT_NUM;
apply_clock: _APPLY_CLOCK clock_type int_num _CYCLES _SEMICOLON;
ch_clock_freq: _CHANGE_CLOCK_FREQ _LPR int_num _RPR _SEMICOLON;
clock_type: _TCK | _FCK;
%%

```

Note that the `mtp-stmt` can also be described in the C language. The formal definition of this part is not listed. An example of an MTL description and some CTL descriptions are given in section 4.3.1.

## 4.2 Synthesizers

In the current implementation, only the CTL and MTL have been defined. Hence, only the C2C, which is a program that can synthesize a test program for a chip using CTL, and the M2C, a program that synthesizes a test program for a module using CTL and MTL, are described.

### 4.2.1 C2C Synthesizer

The C2C synthesizer can generate a test program for a chip from the CTL description of the chip, i.e. a CTL-file. The output of the C2C is a program in the ANSI C format. A C compiler is needed to compile the test program into executable code for the test controller such as an MMC.

Comparing the models used in the CTL (see Figure 4.2) and the MTL (see Figure 4.3), it is clear that when a module contains only one chip then both models are the same. Therefore the M2C can be used as a C2C. Since the C2C is built as a subset of the M2C, no further description of the C2C will be given.

### 4.2.2 M2C Synthesizer

The M2C generates a test program for a module from the MTL description of that module and the CTL description of each chip on the module. The outputs of the M2C includes a test program in ANSI C format that can be used to test the module and a file describing the module interconnect. Testing of a module is considered complete when all individual chips on the module and the interconnect between these chips have been tested. As explained before, a C compiler is used to compile



the test program down to executable codes for the MMC. This process is depicted in Figure 4.5.

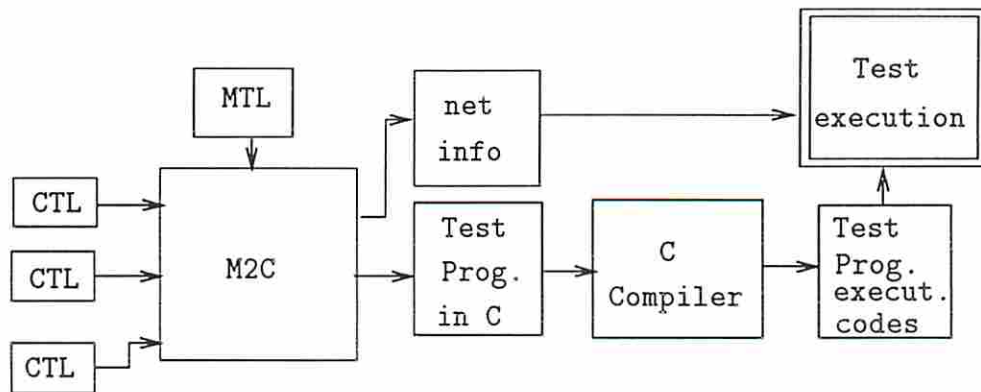


Figure 4.5: Generating test programs for a module.

The structure of the M2C is shown in Figure 4.6. The major components of the M2C include a multiple parser module, a template-based TDM module, a user-defined TDM module, an interconnect test and diagnosis module, a shift adjustment module and a test program manufacturing module. These components are described in more detail in the following paragraphs.

### The parsers

Both the MTL and CTL parsers are constructed using the well known compiler construction tools YACC [35] and LEX [42], developed at the Bell Laboratories. When the syntax of a language is properly described, these two utilities can generate a parser for the language. Due to the use of global variables in the parser, conflicts may exist in a program with more than one parser. This is a major limitation for applications using more than one input language, such as the M2C. To resolve this problem, a parser management technique is used. The technique takes advantage of the use of makefiles in the UNIX environment. Whenever a parser is generated, the name of its global variables are automatically changed before the compilation begins. Hence no two parsers have the same global variables. Using this technique, more than one parser can exist in the same program.

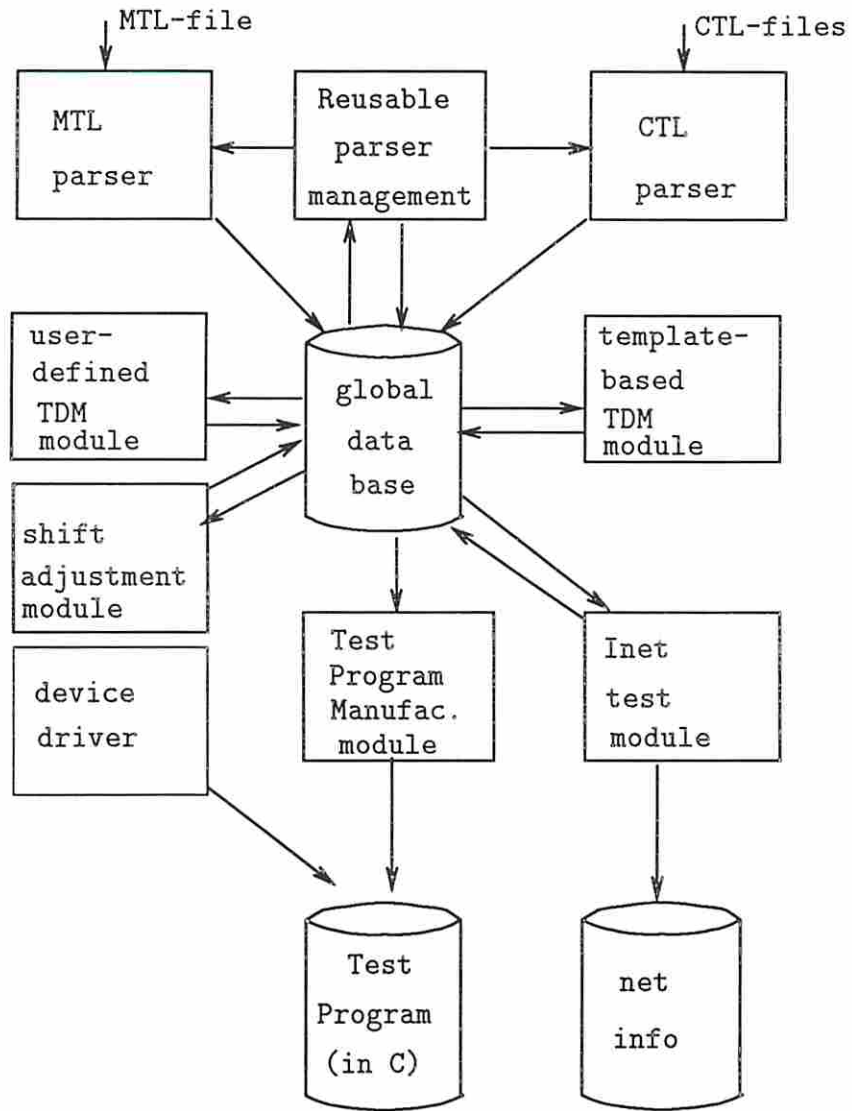


Figure 4.6: The structure of the M2C.

## Template-based TDM Module

Procedures that can generate C programs from a template-based TDM are provided. These procedures are referred to as meta-procedures. Their input is a test procedure written for a template-based TDM and they generate a C program for executing the test process of the selected TDM. These meta-procedures include *callbilbo*, *callfullscan*, *callfullscanN*, *callintest*, and *callrunbist*, which can generate programs for the BILBO, Fullscan, FullscanN, INTEST, and RUNBIST TDMs, respectively.

All information required in generating a test program must be provided to these meta-procedures. For example, when using the meta-procedure *callbilbo*, the following information is required.

- chipID*: the name of the current chip under test;
- chipType*: the type name of the current chip under test;
- ringID*: the test bus ring where the chip under test is located;
- pre*: the number of cells between the CUT and the controller when shifting data (see Figure 4.4);
- post*: the number of cells between the controller and the CUT when shifting data;
- ipre*: the number of cells between the CUT and the controller when shifting instruction;
- ipost*: the number of cells between the controller and the CUT when shifting instruction;
- iniNum*: the number of registers that should be initialized;
- iniIns*: the instructions that are used to select the registers;
- iniVal*: the initial values to be loaded into these registers;
- useIns*: the instruction used during the test execution;
- ckType*: the type of clock used for executing the test;
- ckNum*: the number of clock cycles to execute the test;
- resNum*: the number of registers used to store the test results;
- resIns*: the instructions that can be used to select these registers;

*expVal*: the expected results in these registers when the circuit under test is fault-free.

The values of items such as *iniNum*, *iniIns*, *iniVal*, *useIns*, *ckType*, *ckNum*, *resNum*, *resIns*, and *expVal* are directly available from the CTL description. However, the values of some of the information, such as *chipType*, *ringID*, *ipre*, *ipost*, *pre*, *post*, are not directly available and can only be obtained by processing the information in the MTL-file and the CTL-files.

### User-defined TDM Module

A user-defined TDM is basically a C program plus some test-specific statements. It is necessary to translate these test-specific statements into normal C statements that can be readily executed by a test controller. Due to the differences in the test control model used in CTL (see Figure 4.2) and MTL(see Figure 4.3), the test-specific statements are modified to reflect the change in the control model. For example, the test-specific statement *scanIR(outS)* is changed to *scanIR(RingID, ipre, ipost, outS)* so that the same string of data *outS* can now be properly sent to the chip under test. The value of the information *ringID*, *ipre*, *ipost* are computed in the Shift Adjustment Module.

### Shift Adjustment Module

Often it is desirable to send and receive data only to a single chip on a scan ring while keeping the other chips in their bypass mode. The model used here is shown in Figure 4.7. The data stored in register TxR is shifted into the chain, and the incoming data is shifted into register RxR. The length of both TxR and RxR are assumed to be unlimited since a buffering mechanism is employed to make sure that the TxR will never be empty and that the RxR will never be full.

When exchanging data with a single chip on a scan ring, the number of shifts needs to be adjusted so that (1) data in TxR is sent to the CUT properly, and (2) data in the CUT is received in RxR properly. The calculation of the number of

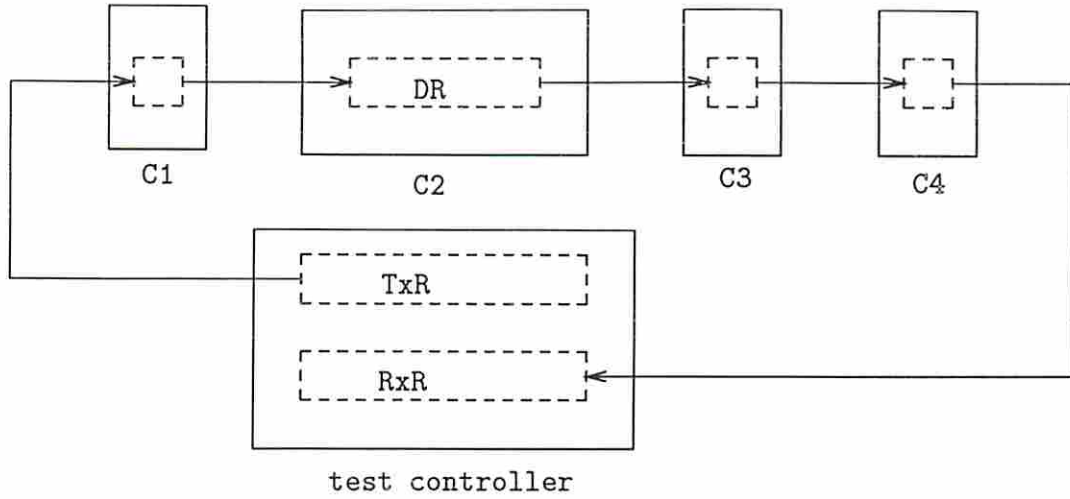


Figure 4.7: Model for calculating the number of shifting.

shifts is done as follows. Let the length of DR in C2 be  $len$ , the number of chips between C2 and RxR be  $pre$ , the number of chips between TxR and C2 be  $post$ . For example, in Figure 4.7,  $pre = 2$ ,  $post = 1$ . Depending on the values of  $pre$  and  $post$ , three cases are possible. These cases are summarized in Table 4.1.

| cases                         | $pre > post$ | $pre < post$ | $pre = post$ |
|-------------------------------|--------------|--------------|--------------|
| # of shifts                   | $len + pre$  | $len + post$ | $len + post$ |
| # TxR leading 0s added        | $pre - post$ | 0            | 0            |
| # TxR trailing 0s added       | $post$       | $post$       | $post$       |
| # RxR leading bits discarded  | $pre$        | $pre$        | $pre$        |
| # RxR trailing bits discarded | 0            | $post - pre$ | 0            |

Table 4.1: The numbers used in the shifting.

#### Case 1: $pre > post$

In this case the total number of shifts is  $len + pre$ . It is necessary to add  $pre - post$  leading 0s to the output string before loading it into the TxR, such that after the shifting the output string will be properly loaded into the DR. It is also necessary to discard the first  $pre$  number of bits received in the RxR such that the content of the DR is properly received.



#### Case 2: $pre < post$

In this case the total number of shifts is  $len + post$ . No leading 0s are needed for the output string. However, it is necessary to discard the first  $pre$  bits received in the RxR. It is also necessary to discard the last  $post - pre$  bits received in the RxR so that the contents of the DR is properly received.

#### Case 3: $pre = post$

In this case the total number of shifts is  $len + post$ . No leading 0s are needed for the output string. However, it is necessary to discard the first  $pre$  number of bits received in the RxR such that the content of the DR is properly received.

### Test Program Manufacturing Module

The synthesizer M2C generates C programs from the CTL and MTL descriptions. These C programs are then sent to an IBM AT computer where they are compiled and executed. The IBM AT serves as the host for the test controller in the present implementation of the MMC prototype. The purpose of the Test Program Manufacturing module is to reduce the manual operation in transferring and compiling the test programs, therefore, reducing the errors in the generation of the test programs. This module can generate a makefile that can manufacture the test program in the IBM AT. In addition, this module also copies all the C programs to a specific subdirectory so that minimal effort is required to transfer test programs from the SUN, which is used to synthesize the test programs, to the IBM AT.

### Interconnect Test Module

This module deals with the testing and diagnosis of the module interconnect. It contains four major parts, namely net\_list generation, test generation, test application and results analysis.

In the net\_list generation part, the net\_list, which is described in the MTL-file, is first read and the data structure established. The drivers and receivers of these nets are all part of the boundary scan registers of the chips. The CTL-files of these chips are then read such that the physical configuration of these boundary

scan registers are established. A mapping mechanism is then used to map the terminals of a net to the physical locations in the boundary scan register. The mapping information is then output to a file called `infofile.net` which is later transferred to the IBM AT. When executing the interconnect testing, this file is first read so that all information required to perform the test can be obtained without the need to consult both the MTL-files and CTL-files.

The syntax of the file `infofile.net` is as follows. The file consists of a *head-line* followed by one or more *net-lines*. The *head-line* consists of a net number which is the total number of nets in the module, a ring number which is the total number of test bus rings used in the module and one or more ring descriptions. A ring is described by three numbers; the first number is the identification of the ring, the second number is the sum of the IR length of all chips, and the third number is the sum of the length of the Boundary Register of all chips. A *net-line* consists of a number which is the identification of the net, the number of drivers of the net followed by a list of drivers, and the number of receivers followed by a list of receivers. A driver is described by a number which identifies the ring on which the driver is located; a number which is the location of the driver in the ring; a flag which indicates whether the driver is 2-state or 3-state; and an optional enabling information which is needed only when the driver is 3-state. The first number of the enabling information representing the location of the control cell, which must be in the same ring as the driver. The second number is the value that should be loaded into the control cell in order to disable the driver. A receiver is described by two numbers; the first number identifies the ring on which the receiver is located and the second number gives its location in the ring.

For example, the file `infofile.net` of the module shown in Figure 4.3 is as follows.

```

6 2 1 14 24 0 6 6
1 1   1 19 0   1 1 23
2 1   1 22 0   1 1 14
3 1   1 21 0   1 1 20
4 1   1 18 0   1 1 15
5 1   1 7 1 16 1   1 0 5
6 1   1 6 1 16 1   1 0 2

```

The first line indicates that there are six nets and two rings in the module. Ring 1 contains fourteen instruction register cells and twenty four boundary scan cells. Ring 0 contains six instruction register cells and six boundary scan cells. The data on the second line indicates that net 1 (which is not shown) has one 2-state output driver which is located in ring 1 at the nineteenth location. Also there is one receiver on the net. The receiver is located on ring 1 at the location 23. The data on other lines can be interpreted similarly.

In the test generation part, a test set that can identify all diagnosable faults is generated. This test set contains a walking ones and a walking zeros sequence. Detailed analysis of fault models and the theorems and algorithms for generating the test set are presented in chapter 6.

In the test application part, a test schedule, that can be used to apply all test vectors and collect the test results, is produced. Depending on the number of boundary scan rings used, and the connectivity among these rings, the application sequence of the test vectors can be properly determined in order to achieve the minimal test application time. Detailed analysis of this problem along with some theorems and algorithms that lead to the generation of minimal time schedules are presented in chapter 7.

In the results analysis part, the collected test results are compared with the test vectors and then analyzed. Based on this analysis, the faults in the module interconnects are identified. The analysis is closely related to the test generation techniques. In fact, both parts are treated in chapter 6.

## Device Driver

The device driver consists of a set of C functions that control the operation of the test channel. Two functions that are used in the MMC prototype developed at USC [44] are shown. A function that can load a two-byte data (*outword*) into a register of the test channel, addressed by *portid*, is implemented as a `writeReg` function in Microsoft C as follows:

```
void writeReg(portid, outword)
```

```

unsigned int portid, outword;
{
    int i;
    outp(0x30f, outword / 256); /* high byte */
    outp(portid, outword % 256); /* low byte */
    for(i=0; i<30; i++); /* tc synchronization */
}

```

Another function (`readReg`) that can read a two-byte data from a register of the test channel is implemented as:

```

unsigned int readReg(portid)
unsigned int portid;
{
    unsigned int lowByte, highByte;
    lowByte=inp(portid);
    highByte=inp(0x30f);
    return(lowByte+highByte*256);
}

```

These functions are hardware-dependent in that the I/O address `portid` is determined by the physical implementation of the test controller. For the MMC prototype built at USC [44], the valid address for `portid` ranges from 300 to 30f(hex). These functions are also C compiler-dependent. For example, if the Turbo C Compiler is used, then the function `outp` used in both `writeReg` and `readReg` should be replaced by `outportb`. Similarly, the function `inp` should be replaced by `inportb`.

Other functions that are related to the device drive may also be required. For example, if the communication between the test channel and the processor of the controller is done through an interrupt mechanism, then interrupt service functions are needed.

One of the major advantage of the MMC is that all test programs run by it can be written in portable code except the functions included in the device driver as shown above.

## 4.3 An Example

In the BOLD system, the process of testing a module consists of the following steps:

1. Prepare the input files. These files include a MTL-file and many CTL-files.
2. Synthesize the test program by running M2C. For convenience, the program source files are copied into a directory.
3. Transfer all files in the directory into the IBM AT computer, which serves as the host of the MMC prototype.
4. Manufacture the executable codes using the MAKE utility.
5. Execute the test program by the MMC.

The process is demonstrated using the following simple example. The module consists of only three chips. The first chip `app1` was developed at USC [44]. The second chip `TI8374` [55] is a product of Texas Instrument. The third chip `bilbo1` does not exist.

To synthesize test programs for this module, four input files are required, namely `ex1.mtl` `app1.ct1` `ti8374.ct1` and `bilbo1.ct1`. These files are described below.

### 4.3.1 An MTL-file

The MTL description (`ex1.mtl`) of the module is shown below.

```
MODULE = ex1;
LIB = lib1(3);
DEVICE_LIST = (Chip1 app1) (Chip2 TI8374) (Chip3 bilbo1);
TEST_BUS =
  RING 0: Chip1,
  RING 1: Chip2 => Chip3;
NET_LIST =
  NET 1: (Chip1 DIN) (Chip1 SUM),
```



```

    NET 2: (Chip1 C0) (Chip2 D2),
    NET 3: (Chip3 I1) (Chip2 Q2);
MODULE_TEST
#include <stdio.h>
#include <string.h>
#include "comp.h"
main()
{
    testinet();      /*test the entire interconnect network*/
    testchip(Chip1); /*test the entire chip */
    testchip(Chip2);
    testchip(Chip3);
}
END_TEST

```

The test bus is organized into two rings. Only one chip is located on ring 0. The other two chips are located on ring 1. To simplify the problem, only three nets are shown in the example. The test procedure for the module consists of four statements, which are self-explanatory.

### 4.3.2 CTL-files

The CTL description of the chip app1 is shown below. A detailed description of this chip can be found in [44].

```

-- CTL description of the app1 chip
entity app1 is
    generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");
    port (RESET, DIN: in bit; C0, SUM:out bit; VCC, GND:linkage bit);
    use STD_1149_1_1990.all; --1149.1-1990 attributes and definitions
    attribute PIN_MAP of app1: entity is PHYSICAL_PIN_MAP;
    constant DW_PACKAGE:PIN_MAP_STRING:="RESET:48,DIN:8,C0:5,SUM:3,"&
        "TDI:45, TDO:44, TMS:46, TCK:52, TRST:47, VCC:25, GND:49";
    attribute TAP_SCAN_IN    of TDI : signal is true;
    attribute TAP_SCAN_MODE  of TMS : signal is true;
    attribute TAP_SCAN_OUT   of TDO : signal is true;
    attribute TAP_SCAN_CLOCK of TCK : signal is (2.0e6, BOTH);
    attribute INSTRUCTION_LENGTH of app1 : entity is 3;
    attribute INSTRUCTION_OPCODE of app1 : entity is

```

```

        "BYPASS (001, 101, 011, 111)," &
        "EXTEST (000)," &
        "INTEST (100)," &
        "SAMPLE (010)," &
        "SCANFB (110)";
attribute INSTRUCTION_CAPTURE of app1 : entity is "101";
-- attribute INSTRUCTION_DISABLE of app1 : entity is "TRIBPY";
attribute REGISTER_ACCESS of app1 : entity is
--implicit      "BOUNDARY (EXTEST, INTEST, SAMPLE)," &
--implicit      "BYPASS (BYPASS)," &
        "FBR[2] (SCANFB)"; -- 2-bit FeedBack Register
attribute BOUNDARY_CELLS of app1 : entity is "BC_2";
attribute BOUNDARY_LENGTH of app1 : entity is 3;
attribute BOUNDARY_REGISTER of app1 : entity is
-- num cell port  function safe [ccell disval rslt]
        "2 (BC_2, DIN,  input, X)," &
        "1 (BC_2, SUM,  output2, X)," &
        "0 (BC_2, CO,  output2, X)";
attribute TEST_PROC of app1 : entity is
        "Test_Begin" &
        "TDM 0 = FULLSCAN;" &
        "REG=FBR, VECFILE=api_in2, RESFILE=api_out2;" &
        "REG=BOUNDARY, VECFILE=api_in1, RESFILE=api_out1;" &
        "CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;" &
        "Test_End ";
end app1;

```

The BSDL description of the chip TI8374 can be found in [55]. Hence only the test procedure part of CTL-file is shown in the following:

```

Test_Begin
TDM 0 = USER_DEFINE;
#include <stdio.h>
#include <string.h>
#define IR 1
#define DR 0
top()
{
char outs[20], ins[20];
char *p1, *p2;
    sprintf(outs, "00000011"); /* INTEST */
    scanIR(outs);
}

```

```

sprintf(outs, "000000001010101000");
scanDR(outs); /* load D=10101010, clk=0 */
sprintf(outs, "000000001010101001");
scanDR(outs); /* load D=10101010, clk=1 */
sprintf(outs, "00000000101010100");
strcpy(ins, scanDR(outs));
/*load D=01010101,clk=0 and get previous result*/
p1=ins+10;
p2=outs+2;
if(strncmp(p1,p2,8)!=0) {
    printf("error in 10101010 test.\n");
    exit(1);
}
sprintf(outs,"00000000101010101");
strcpy(ins, scanDR(outs));
strcpy(ins, scanDR(outs));
if(strncmp(p1,p2,8)!=0) {
    printf("error in 01010101 test.\n");
    exit(1);
}
else{
    printf("TI8374 is tested OK.\n");
}
}
Test_End

```

For the chip bilbo1, only the test procedure part is shown below.

```

Test_Begin
TDM 0 = BILBO;
INITIALIZE FBR=11B, BOUNDARY=100B;
USE_INSTRUCTION = BYPASS;
CLOCK = TCK 3000 CYCLES_IN RUN_TEST_IDLE;
EXPECTED_RESULT FBR=11B, BOUNDARY=001B;
Test_End

```

### 4.3.3 The Interconnect Information

The net list information that will be sent to the MMC is as follows:

```

3 2 0 3 3 1 11 21
1 1   0 1 0   1 0 2
2 1   0 0 0   1 1 17
3 1   1 9 1 19 1   1 1 2

```

#### 4.3.4 Synthesized Test Programs

The synthesized test program for the module `ex1` consists of seven files, namely `comp.h`, `ex1main.c`, `TI8374tops.c`, `driver.c`, `inetpc.c`, `template.c` and `infofile.net`. The file `comp.h`, shown below, defines all variables used in the main program.

```

#define BUFSIZE 1024
char chipID[30];
char chipType[30];
char regIns[BUFSIZ];
char vecFID[BUFSIZ];
char expFID[BUFSIZ];
char iniIns[BUFSIZ];
char iniVal[BUFSIZ];
char useIns[BUFSIZ];
char resIns[BUFSIZ];
char expVal[BUFSIZ];
#define IR 1
#define DR 0

```

The file `ex1main.c` contains the main function that is needed for any program. The prefix `ex1` is extracted from the name of the MTL file `ex1.mtl`. Four function calls are contained in the main program. The first one `inetpc` is used to test the interconnect. The second one `fullscanN` is used to test the chip `Chip1` by using fullscan TDM. The third one `TI8374top0` is used to test the chip `Chip2` by using a user-defined procedure that has the name `top0`. The prefix `TI8374` is extracted from the name of the CTL-file `TI8374.ctl`. The last one `bilbo` is used to test the chip `Chip3` by using the BILBO TDM. The file `ex1main.c` is as follows:

```

#include <stdio.h>
#include <string.h>

```

```

#include "comp.h"
main()
{
inetpc(1); /* interconnect test */

sprintf(chipID, "Chip1");
sprintf(chipType, "app1");
sprintf(regIns, "110000");
sprintf(vecFID, "AP1_IN2,AP1_IN1");
sprintf(expFID, "AP1_OUT2,AP1_OUT1");
fullscanN(chipID,chipType,0,0,0,0,0,2,regIns,vecFID,expFID);

TI8374top0(1, 3, 0, 1, 0);

sprintf(chipID, "Chip3");
sprintf(chipType, "bilbo1");
sprintf(iniIns, "110000");
sprintf(iniVal, "11100");
sprintf(useIns, "001");
sprintf(resIns, "110000");
sprintf(expVal, "11001");
bilbo(chipID,chipType,1,0,8,0,1,2,iniIns,iniVal,useIns,1,
      30000,1.000000,2,resIns,expVal);
}

```

The file TI8374tops.c is a translated version of the user-defined procedure for testing Chip2 that is of the type TI8374. Note that the statement scanIR(outs); in the file TI8374.ct1 has been translated into scan(TI8374rid, IR, TI8374ipre, TI8374ipost, outs);. The added information is based on the physical organization of the test bus.

```

#include <stdio.h>
#include <string.h>
#define IR 1
#define DR 0
TI8374top0(TI8374rid, TI8374ipre, TI8374ipost, TI8374pre,
          TI8374post)
int TI8374rid, TI8374ipre, TI8374ipost, TI8374pre, TI8374post;
{
char outs[20], ins[20];

```



```

char *p1, *p2;
    sprintf(outs, "00000011");
    scan(TI8374rid, IR, TI8374ipre, TI8374ipost,outs);
    sprintf(outs, "000000001010101000");
    scan(TI8374rid, DR, TI8374pre, TI8374post,outs);
    sprintf(outs, "000000001010101001");
    scan(TI8374rid, DR, TI8374pre, TI8374post,outs);
    sprintf(outs, "00000000101010100");
    strcpy(ins, scan(TI8374rid, DR, TI8374pre, TI8374post,outs));
    p1=ins+10;
    p2=outs+2;
    if(strncmp(p1,p2,8)!=0) {
        printf("error in 10101010 test\n");
        exit(1);
    }
    sprintf(outs,"00000000101010101");
    strcpy(ins, scan(TI8374rid, DR, TI8374pre, TI8374post,outs));
    strcpy(ins, scan(TI8374rid, DR, TI8374pre, TI8374post,outs));
    if(strncmp(p1,p2,8)!=0) {
        printf("error in 01010101 test\n");
        exit(1);
    }
    else{
        printf("TI8374 is tested OK.\n");
    }
}

```

The files `driver.c`, `inetpc.c` and `template.c` are not shown here. The `driver.c` contains all the functions that can be used to control a device. The `inetpc.c` contains the interconnect test program and `template.c` contains all the template-based TDM functions. By reading the file `infofile.net`, the `inetpc` procedure can be used to test different interconnect networks. These three files remain unchanged even when the MTL-file or the CTL-files are changed.

### 4.3.5 Activities between Processor and Test Channel

The test channel is controlled via processor read/write operations. For example, the function `scan(0,0,0,0,010)`; represents the operation of sending a string of

010 to the data register of a chip which is the only chip located on ring 0. This function is further translated into the following sequences of read/write operations. Once the test channel is properly initialized, it can be started and it carries out the required operation. When the operation is finished, the results are read.

```

writeReg(776, 0); /*disable FEN */
writeReg(768, 32); /*load CR with binary 10000 */
writeReg(769, 14); /*load CNR with 14 */
writeReg(772, 1); /*load TC with 1 */
writeReg(773, 16384); /*load TxR */
writeReg(777, 0); /*clear SR */
writeReg(774, 0); /*clear RxR */
writeReg(776, 1); /*enable FEN */
writeReg(776, 0); /*disable FEN */
writeReg(777, 0); /*clear SR */
readReg(774); /*read RxR */

```

The entire process of testing the module described by `ex1.mtl` contains 1059 read/write operations. It is clear that without the synthesis of test program, it would take a tremendous effort to write the test program for even a simple example.

#### 4.3.6 Activities on the Test Bus

The testing of a chip is realized by controlling the data on the test bus, which consists of four lines, namely TCK, TMS, TDI and TDO. Using the same example, the process of sending instruction 010 to a chip (`app1`) located in ring 0, requires the following binaries sequences.

```

TCK  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
TMS  0 0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0
TDO  0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0

```

The entire program for testing the module described in `ex1.mtl` executes for about 1000 TCK clock cycles. This kind of test program is hard to write and almost impossible to comprehend by a user. Consequently, the cost associated with the development and maintenance of such a program is very high. This is also the case when testing a chip using automatic test equipment (ATE) that is not specifically designed with the boundary scan control facilities. The ATE can

control and observe a number of I/O pins in parallel in the form of binaries (or timing waveforms). In this case, a test engineer has to deal with the testing of a chip in binaries. The superiority of the synthesis approach used in BOLD is clearly demonstrated in this example.

## 4.4 Results

Test programs for several modules have been synthesized. The results are presented in Table 4.2. To simplify the comparison, only two types of chips are used in the module. Each chip is tested via either the BILBO TDM or the LSSD TDM. A BILBO chip represents a chip that is tested via a BILBO TDM, while an LSSD chip represents a chip that is tested via a LSSD TDM. For example, the `bilbo1` is a BILBO chip and `app1` is an LSSD chip. The TDM for `bilbo1` requires the application of 3,000 test patterns when executing the test. On the other hand, the TDM for `app1` requires the application of only 8 test vectors via two scan chains, which have a total length of 5.

The test time required for testing interconnects is based on the application of a counting sequence. The data stored in the memory of the controller includes the instructions, the seeds and signatures, and the test vector and correct results for each chip.

The synthesis is done using a Sun 4/60 workstation. The time to synthesis the test program is shown in seconds. For a module containing twenty chips, the synthesis is completed in less than 1 second. The program size is shown in terms of bytes. Due to the characteristics of the machine, the size of an executable code in a Sun 4 is a multiple of 1024 bytes. This explains why the modules `mod2`, `mod20`, `mod3`, `mod30`, `mod4` and `mod40` all have the same program size. The column # `rw ops` represents the number of access from the processor to the test channel chip. Note that this number is not a good indication of test time. This is due to the inclusion of BILBO chips that have a long test time while requiring only a small number of read/write operations from/to the test channel to set up the test. The test time is the number of test clock cycles required to complete the test. The listed

| mod id | # ring | #BILBO chips | # LSSD chips | time to synthesis | program size | # net | # rw ops | test time |
|--------|--------|--------------|--------------|-------------------|--------------|-------|----------|-----------|
| mod0   | 1      | 0            | 2            | 0.3               | 57,344       | 2     | 1,518    | 621       |
| mod00  | 2      | 0            | 2            | 0.3               | 57,344       | 2     | 1,518    | 677       |
| mod1   | 1      | 1            | 1            | 0.4               | 57,344       | 2     | 858      | 3,333     |
| mod10  | 2      | 1            | 1            | 0.4               | 57,344       | 2     | 913      | 3,023     |
| mod2   | 1      | 10           | 1            | 0.7               | 73,728       | 21    | 1,935    | 31,800    |
| mod20  | 2      | 10           | 1            | 0.6               | 73,728       | 21    | 2,056    | 31,490    |
| mod3   | 1      | 1            | 10           | 0.6               | 73,728       | 6     | 7,288    | 9,126     |
| mod30  | 2      | 1            | 10           | 0.6               | 73,728       | 6     | 7,329    | 6,126     |
| mod4   | 1      | 10           | 10           | 0.9               | 73,728       | 30    | 8,391    | 36,710    |
| mod40  | 2      | 10           | 10           | 0.8               | 73,728       | 30    | 8,452    | 30,230    |

Table 4.2: Synthesis results for some modules.

figures reflect the case where the test bus never enters the pause state. This may occur when the test controller cannot supply data to the test bus fast enough.

The size of the test vector files for these examples are very small, therefore, its impact on the total storage size of the program is negligible. The number of nets are also small since both `app1` and `bilbo1` have very few I/O pins. In general the cases where two test rings are used have shorter test times when compared to those have only one ring. The impact of the test time can be much more significant if an example that has a large number of test vectors is examined. One exception shown in the table is that the test time required for `mod0` (one ring) is less than that of `mod00` (two rings). The reason is as follows. Both modules contain two chips that are identical. These chips are tested using the same number of test vectors and the same length for each vector. If they are on the same ring (as in the case `mod0`), fewer redundant bus states are traversed during the test. On the other hand, if each chip is located on a separate ring controlled by the same test channel, then only one ring can be tested at a time (This is the limitation of the MMC). Therefore, more redundant bus state transitions are traversed. This explains why the test time for the two rings case is longer than that of the one ring case.

If two test rings are employed in the module under test, then it is beneficial to arrange all LSSD chips on one ring and the all BILBO chips on the other. In this

way, the BILBO chips can be initialized and while the self-testing is in progress the LSSD chips are tested. This scheme can greatly reduce the test time. However, it can only be applied if the BILBO chips are designed with an autonomous on-chip controller.



## Chapter 5

# Global Controller Minimization Using Test Program Synthesis

Testable chips built with the boundary scan architecture are used in the design of a self-testable module. A module test controller controls the testing of a chip through an on-chip test controller. These controllers are connected via a test bus. The module test time is affected by the overall complexity of these controllers and the configuration of the test busses.

The design of test controllers has been previously addressed. In chapter 3 the design of a universal module test controller (MMC) was presented. The MMC controls the test process of chips which have the boundary scan architecture via one or more test channels. Each test channel controls a test bus, which can be organized in one or two test rings. Data communication can only occur in one ring at a time. A test bus consisting of two rings is shown in Figure 4.3.

In chapter 2 the design of an on-chip controller for a circuit with various test structures, referred to as Testable Design Methodologies (TDMs), was presented. Depending on the assistance from the bus during test execution, on-chip controllers can be designed in two different styles, namely autonomous and bus-dependent. An autonomous controller can execute the test without any assistance from the test bus, once it is properly initialized. Chips with such controllers can be tested concurrently using only one test ring. A bus-dependent controller requires the assistance from the test bus during the entire test execution process. Thus if only one test channel

is used, two chips with such controllers must be tested in sequence even if they are on different test rings. In a module the more chips designed with autonomous controller, the shorter the module test time. This is achieved at the expense of increased overall controller complexity since autonomous controllers usually have a higher complexity.

An important objective for the module test design is to minimize the overall controller complexity while keeping the module test time within reasonable bounds. To achieve this it is crucial to quantify both the controller complexity and the test time for a module. The complexity of an on-chip controller can be easily calculated once its design is known. Since the complexity of an MMC is fixed, the overall controller complexity can be easily computed. On the other hand, the module test time cannot be directly obtained. This is true even when the time required to test each individual chip is known. Worse still, a chip's test time is not always available. For example, if a chip is designed with a user-defined TDM (defined in chapter 4), it is usually not clear how to estimate its test time.

The test program synthesis technique provides a way to calculate both the module and chip test time. This is done by examining the number of test clock cycles required in their corresponding test programs. In this chapter two approaches for controller minimization, referred to as *tradeoff curve-based minimization* and *algorithm-based minimization*, are presented. These two approaches can determine not only the complexity of each on-chip controller but also the test bus configuration. The derived results guarantee that a module can be tested within a given time while the overall complexity of the controllers is minimal.

While both approaches require the synthesis of test programs, they differ in the way the module test time is obtained. In the first approach, the module test time is calculated only after the module test programs are synthesized. In the second approach, the test time for each individual chip is first calculated from the synthesis of the chip test programs. The module test time is then estimated based on the chip test time and the module test configuration.

## 5.1 Tradeoff Curve-Based Minimization

This approach requires the plotting of a curve that relates the module test time to the overall controller complexity. For each design point on the curve, the module test time is calculated after the module test programs have been synthesized. The complexity for each design can be calculated once the design of the controllers are known. Using such a curve, a design point can be easily selected such that within a time bound the overall controller complexity is minimal.

This approach is illustrated by the example circuit in Figure 5.1 which has only one LSSD kernel. The kernel includes a 16 bit input register, a 16 bit output register and a combinational circuit to be tested. The test vectors are generated using a test pattern generator (TPG) and results are compacted using a signature analyzer (SA). It is also assumed that the fault simulation task has been previously performed and 1000 test vectors are needed to achieve the required fault coverage. The test control facilities for this kernel include: two 16 bits LFSRs (one used as a TPG and the other as an SA); a 10 bit counter TC used to count the number of test vectors applied; a 4 bit counter SC1 used to count the number of shift operations; a 4 bit register SC2 for holding the initial value for the SC1; a 3-bit finite state machine FSM to control the LSSD test procedure; two 16 bit data words, one being the seed of the TPG, the other the correct signature of the SA; and a 16 bit comparator. The hardware overhead for each control facility is estimated based on the following assumption: TPG: 32 units, SA: 32 units, TC: 20 units, SC1: 8 units, SC2: 8 units, finite state machine: 6 units, seed and signature storage: 32 units, comparator: 16 units. The total complexity of the test control hardware is 152 units.

These test facilities are provided by the MMC and the on-chip controller. The facilities on the left hand side of any one of the four partition lines, shown in Figure 5.1, can be provided by the MMC; while those at the right hand side are provided by the on-chip controllers. Since the design of an MMC include all the possible test facilities, the complexity of an MMC is fixed. However, the complexity of the on-chip test controller varies as the partition line changes.



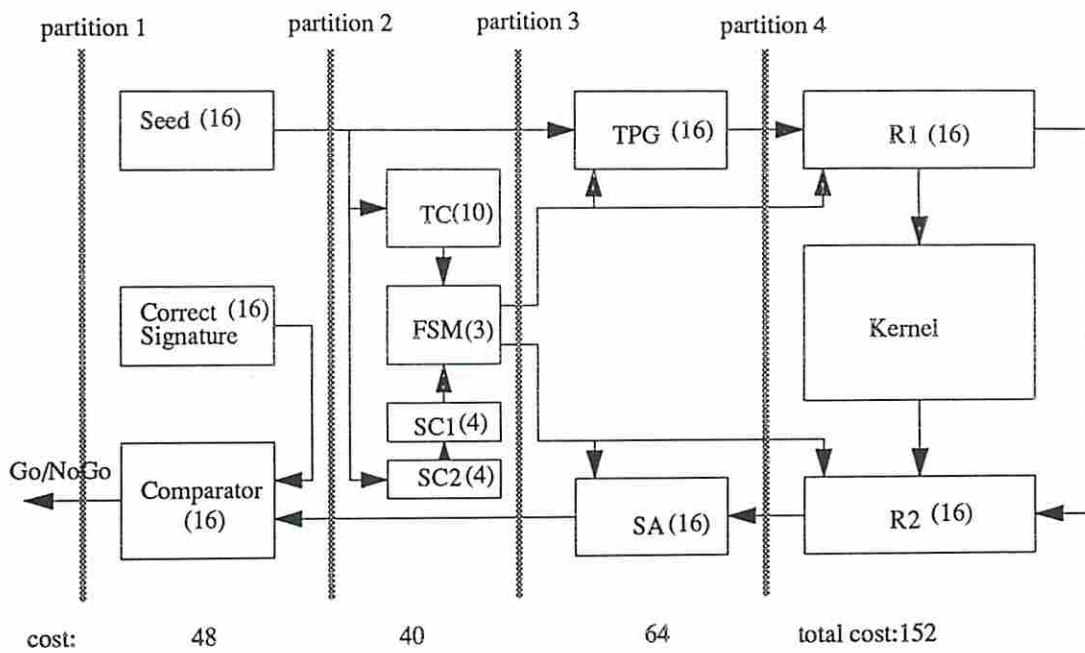


Figure 5.1: Possible partitions of test resources.

Let D1, D2, D3 and D4 be the four design styles representing the circuit at the right hand side of each of the four partition lines. For example, D4 contains the chip to be tested and no test facilities. The complexity of the on-chip controller for D1 is the highest among the four, while that for D4 is the lowest. In addition, the controllers for D1 and D2 are autonomous, while the controller for D3 and D4 are bus-dependent. The module under consideration contains two chips Chip1 and Chip2. Each of these chips can be designed using one of the four design styles. Therefore, the module can be designed in 10 different ways, namely d11, d12, d13, d14, d22, d23, d24, d33, d34, and d44, where  $d_{ij}$  represents the case when one chip has the design style  $D_i$  and the other  $D_j$ .

The objective is to find the design style for both chips such that the overall controller complexity is minimal and the test time is less than or equal to a given bound.

#### **Test time versus controller complexity**

Using BOLD, the required inputs are a CTL-file for each chip and an MTL-file for the module. These files can be easily prepared. The test programs are then automatically synthesized and the test time is calculated. Figure 5.2 shows the relationship between the test time and the controller complexity. In the figure there are four plots identified by the design style for Chip1. For example, in plot 1 denoted by "Chip1 uses D4", the data points illustrate the cases where Chip1 has the design style D4 while Chip2 can take any of the four other design styles (D1, D2, D3 and D4). The horizontal axis (controller complexity) represents the overall controller complexity of the module. The complexity of the bus interface (the Test Access Port) and the MMC are not included since they are fixed. The vertical axis represents the total test time for the module. Note that the interconnect test time is not included since it is fixed if the number of test rings remains unchanged. The data points '+' represent the cases where both Chip1 and Chip2 are located on the same test ring. The data points 'o' connected by the dashed line segments represent the cases where Chip1 and Chip2 are located on different test rings.

In general the test time decreases as the complexity of the on-chip controller increases. Also, the number of test rings may affect the test time. For example,



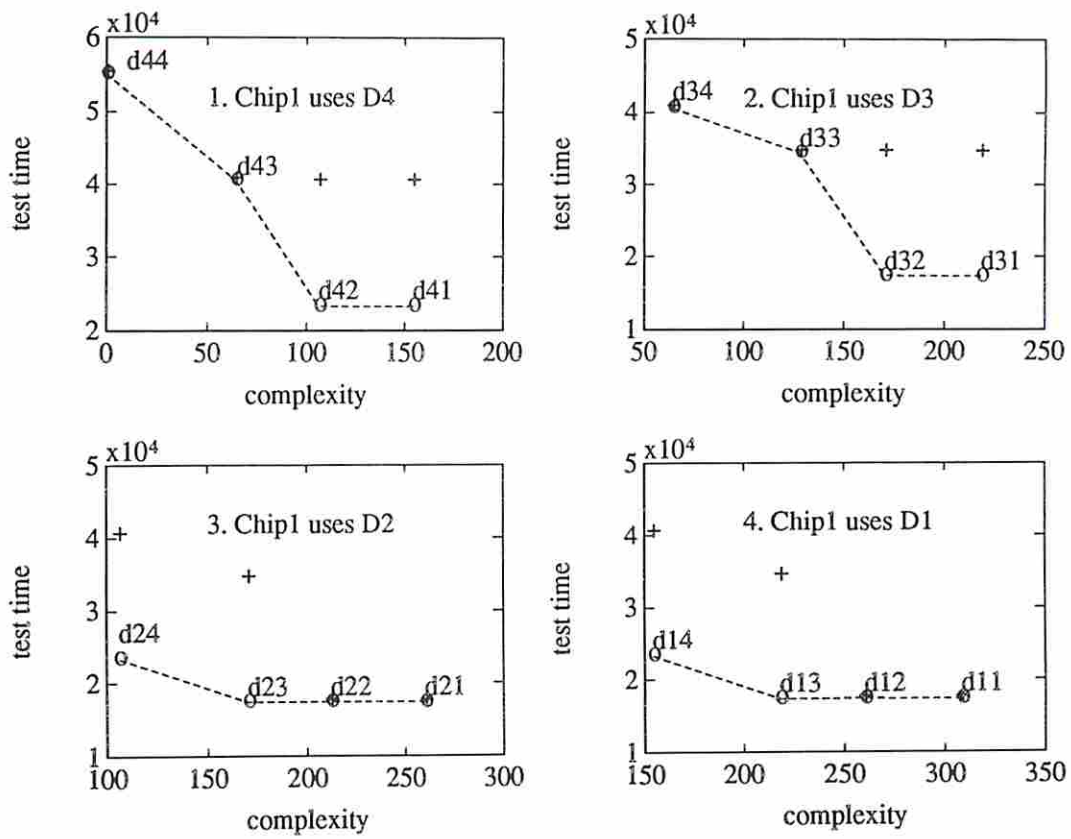


Figure 5.2: Test time versus controller complexity.

in plot 1, the test time is affected by the number of test rings only when Chip2 has the design styles D1 or D2. In both cases, Chip2 has autonomous controller. Once Chip2 is properly initiated, it can be tested concurrently with Chip1 since both chips are located on different test rings. On the other hand, the test time is not affected by the number of test rings when Chip2 has the design styles D3 or D4. In this case these two chips are tested in sequence since they both require the assistance from the test bus in executing the test. So the test time remains the same even when these two chips are located on different test rings.

The results shown in plot 2 are similar to that of plot 1 since Chip1 also has a bus-dependent on-chip controller. In both plot 3 and 4, Chip1 has an autonomous on-chip controller. The number of test ring affects the test time only if Chip2 has the design style D3 or D4. If two test rings are used, Chip1 and Chip2 can be tested in parallel. On the other hand, these two chips must be tested in sequence if only one test ring is used.

### Tradeoff Curve

In Figure 5.3 the test time and controller complexity of all the 10 possible design combinations for Chip1 and Chip2 are shown; that is  $d_{ij} = d_{ji}$ . The horizontal axis represents the overall controller complexity of the module (in units). The vertical axis represents the total test time for the module in test clock cycles. The data points '+' represent the cases where both chips are located on the same test rings. The data points 'o' along with the dashed line segments represent the cases where the two chips are located on different test rings. A curve consisting of these data points can be used to make design decision in selecting an optimal design point for given constraints in both test time and overall controller complexity.

As shown in the figure, the test time when using two test rings is always less than or equal to that of the one test ring case. It is interesting to note that even the design d33 (for the two test ring case) has a higher controller complexity than the design d24. The test time for the former is greater than that of the latter. Similarly the design d14 has a much higher complexity than d24, but there is little difference between the test time for both cases. Therefore both designs d33 and d14 are inferior to the design d24. Using the same argument, more inferior designs

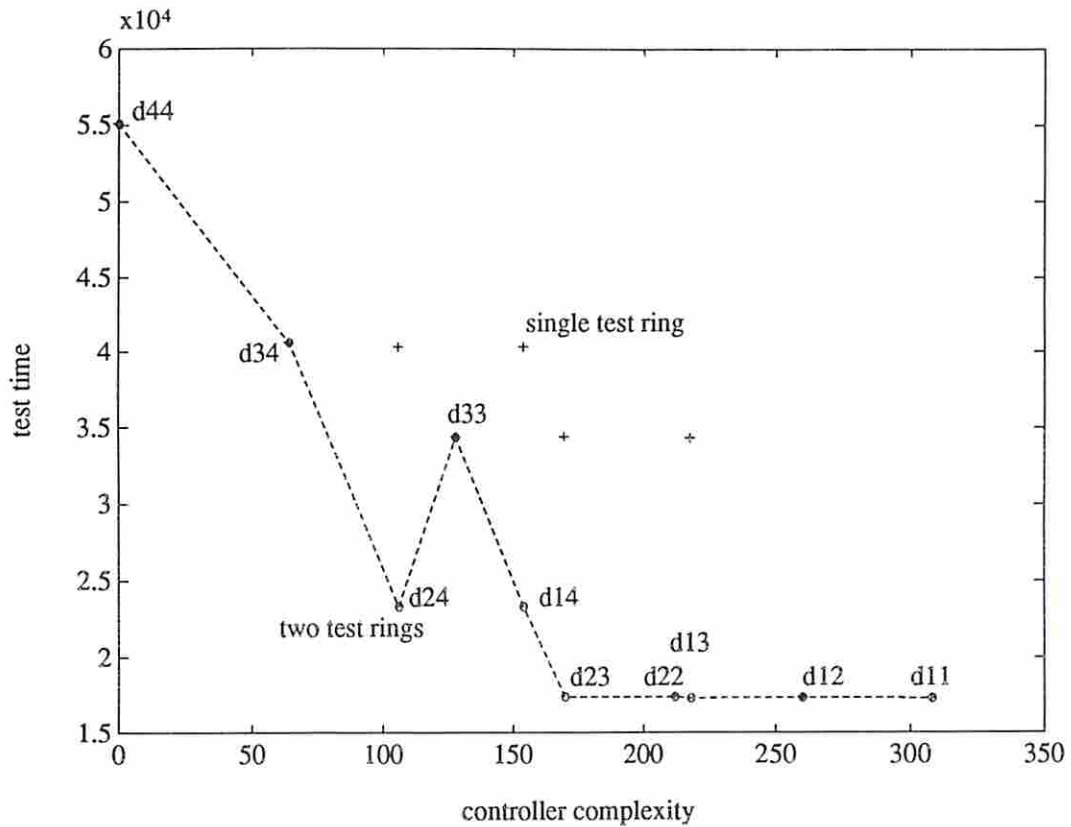


Figure 5.3: Tradeoff curve: Test time versus controller complexity.

can be identified. These inferior designs include d33, d14, d22, d13, d12 and d11. A curve, referred to as the tradeoff curve for a module, consists of all designs that are not inferior. These superior designs include d44, d34, d24 and d23. This curve can be used in selecting the most appropriate design satisfies both the constraints on test time and controller complexity.

The analysis of this example shows that (1) putting both TPG and SA on chip can greatly reduce test time, (2) putting the test control facilities, including FSM and two counters, on chip can also reduce test time, however, (3) putting the seed and correct signature on chip has little if any impact on reducing test time and often leads to an inferior design.

This approach can be generalized to the cases where a module contains many chips that have many different design styles. The optimal solution can be guaranteed if the design points include the entire design space. The drawback of this approach is that it is not feasible for a module containing a large number of chips, since the time required to plot the tradeoff curves in such cases grows exponentially as the problem size increases. To determine the optimal solution in such a case, an algorithmic approach is needed.

Two algorithms that can reduce the time required to generate an optimal solution are presented next. These algorithms do not remove the need of test program synthesis since they assume the test time for each design is known. The test time calculated by these algorithms is only an approximated value since the model used is different from the final design. Therefore the final results have to be verified by the synthesis approach in order to make sure that the module test time is actually less than the given bound.

## 5.2 Algorithm-Based Minimization

To simplify the problem an MMC is assumed to have only one test channel. We make this assumption to ensure that the MMC processor bandwidth will match that of the test channel. If a module has  $m$  test channels, then it will have  $m$  MMCs. Suppose there are  $n$  chips in the module under test. Let  $T$  be the given upper bound of the module test time. The on-chip controller for each chip can be either autonomous or bus-dependent. Let the controller hardware complexity for chip  $i$  be  $a_i$  if its on-chip controller is autonomous; otherwise, let the complexity be  $b_i$ . The test program synthesis technique can be used to calculate the test time for each individual chip. Suppose that the test time for chip  $i$  is  $t_{a_i}$  if its controller is autonomous and  $t_{b_i}$  otherwise.

### 5.2.1 One Test Channel

Assume there is a single test channel and it can control either one or two test rings. However, it can only transmit data to one ring at a time. It is clear that  $a_i \geq b_i$  for all chips. (This assumption can be justified from the discussion in chapter 2).

By connecting all chips that have autonomous on-chip controllers to one test ring, the self-test of all these chips can be executed in parallel with the testing of the chips with bus-dependent controllers. On the other hand, all chips that have bus-dependent controllers must be tested in sequence. The module test time is  $T_m = \max(t_a, t_b)$ , where  $t_a$  is the longest test time required by any chip with an autonomous controller, and  $t_b$  is the sum of the test time for all chips with bus-dependent controllers. In this work, it is assumed that  $t_{ai} \leq T, \forall i$ , where  $T$  is a given parameter representing the upper bound on the total test time.

The objective is to determine the controller design style for each chip such that  $T_m \leq T$  and the total complexity of all the controllers is minimal.

#### Problem Formulation

Let  $x_i$  be a variable associated with chip  $i$  defined as follows.

$$x_i = \begin{cases} 1 & \text{if chip } i \text{ has a bus-dependent on-chip controller,} \\ 0 & \text{otherwise.} \end{cases}$$

The objective is to minimize

$$\sum_{i=1}^n x_i b_i + (1 - x_i) a_i$$

subject to

$$T \geq \sum_{i=1}^n x_i t_{bi}.$$

Note that the objective function can be reformulated as

$$\sum_{i=1}^n (x_i b_i + (1 - x_i) a_i) = \sum_{i=1}^n a_i - \sum_{i=1}^n x_i (a_i - b_i) = C' - \sum_{i=1}^n x_i (a_i - b_i).$$



$C'$  is a constant. Let  $c_i = a_i - b_i$  be the difference in complexity between the autonomous and bus-dependent design for chip  $i$ . It is clear that  $c_i \geq 0$  since for a given chip the complexity of an autonomous controller is higher than that of a bus-dependent controller. The objective can be restated as follows.

Maximize

$$\sum_{i=1}^n x_i c_i$$

subject to

$$T \geq \sum_{i=1}^n x_i t_{bi}.$$

This formula is equivalent to that of the *Optimization 0-1 Knapsack* [53] and can be efficiently solved using dynamic programming. The following algorithm, proposed in [53], is used to solve this problem.

#### Algorithm DP

1. Let  $M_0 = \{(\emptyset, 0)\}$ .
2. For  $j = 1, \dots, n$  do
  - (a) Let  $M_j = \emptyset$ .
  - (b) For each element  $(S, c)$  of  $M_{j-1}$ , add to  $M_j$  the element  $(S, c)$  and also  $(S \cup \{j\}, c + c_j)$  if  $\sum_{i \in S} t_{bi} + t_{bj} \leq T$ .
  - (c) Examine  $M_j$  for pairs of elements  $(S, c)$  and  $(S', c')$  with the same second component. For each such pair, delete  $(S', c')$  if  $\sum_{i \in S'} t_{bi} \geq \sum_{i \in S} t_{bi}$ , otherwise delete  $(S, c)$ .
3. The optimal solution is  $S$ , where  $(S, c)$  is the element of  $M_n$  having the largest second component.

Starting with chip 1, the algorithm determines the design style for each chip in sequence. After the design style of chip  $n$  has been determined, the optimal solution is obtained. A solution is represented by  $(S, c)$ , where  $S$  is the set of chips which should have bus-dependent controllers and  $c$  is the total complexity for this set of chips. A solution set is feasible if  $\sum_{i \in S} t_{bi} \leq T$ .  $M_j$  is the set of all feasible solutions after the design of  $j$ th chip has been determined. It is obvious that the feasible set in  $M_n$  with the largest  $c$  is the optimal solution.

A C program has been implemented based on this algorithm. The program can solve this design problem in  $O(n^2c)$  time. This C program is used in the following algorithm to determine the design style for each chip and also connects the chip to one of the test rings of the test channel. The module can then be tested within the time bound  $T$  by an MMC having a single test channel.

If only one test channel is used to control the testing of a set of chips the following algorithm can be used to determine the design style for each chip such that the total test time is less than or equal to  $T$  and the overall control complexity is minimal. The algorithm also determines the chips are connected to the test bus.

**Algorithm TC1:**

1. Find the test time  $t_{bi}$  and  $t_{ai}$  for each chip using the test program synthesis technique. The test time is calculated based on the model used in describing a CTL-file for a chip (see Figure 4.2 in chapter 4).
2. Formulate the problem as above and find an optimal solution set using Algorithm DP.
3. If a chip is in the solution set, its controller is bus-dependent; otherwise, it is autonomous.
4. Connect all chips that have bus-dependent controllers to ring 0 of the test channel. Connect all chips that have autonomous controllers to ring 1 of the test channel.

**Lemma 2** *The solution set found by Algorithm TC1 is optimal.*

**Proof:**

The proof is obvious since the solution set is found by using a dynamic programming procedure, which implicitly searches through the entire solution space.  $\square$

Using Algorithm TC1, one can quickly obtain optimal controller designs. Figure 5.4 shows the results obtained by using both approaches. The data points connected by solid line segments are the actual values obtained by using the test program synthesis technique. The data points connected by dashed line segments are obtained by using Algorithm TC1. As observed, there is very little difference between the results obtained from the two approaches except for the design d44.

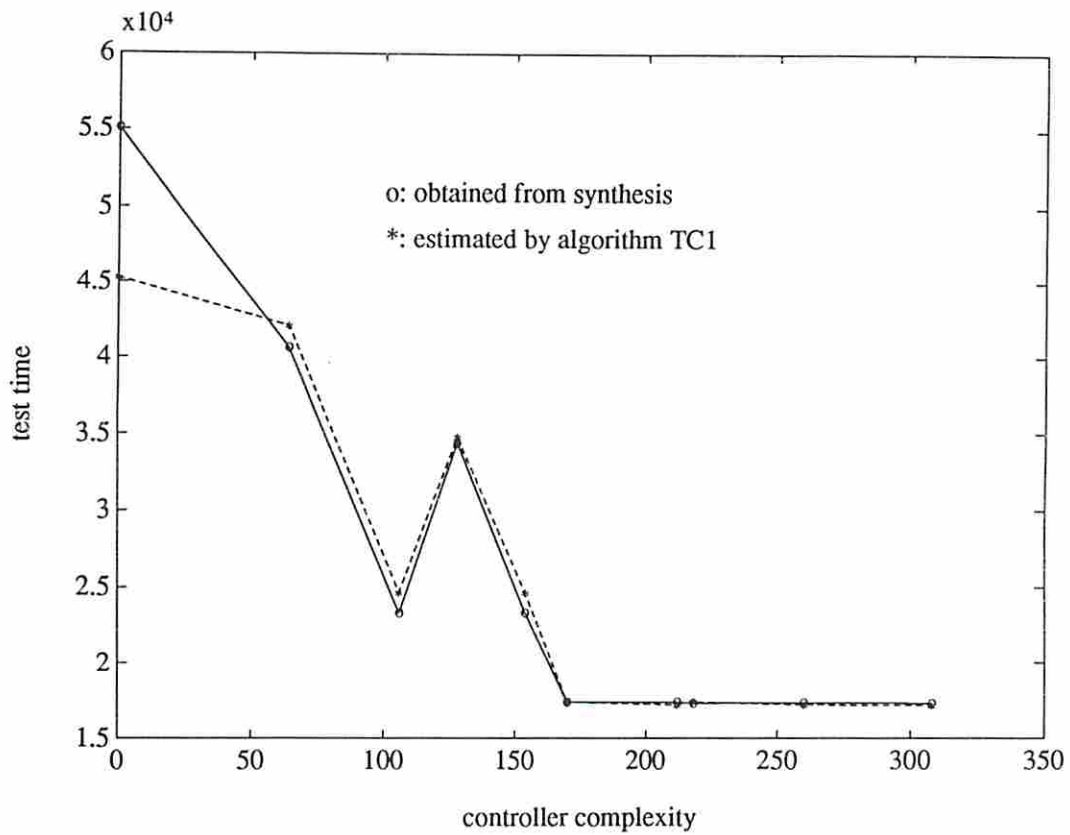


Figure 5.4: Test time estimated by algorithm TC1.

The discrepancy is caused by the simulation program that calculates the test time from the module test program.

### 5.2.2 Multiple Test Channels

The test time for a module can be further reduced by increasing the number of test channel while the overall controller complexity remains minimal. The following algorithm determines the controller complexity for each chip and the number of test channels required such that the module test time is less than or equal to  $T$  and the overall controller complexity is minimal. The controller complexity includes both the complexities of on-chip controllers and of the test channels.

Let  $c_{tc}$  be the complexity of a test channel. Let  $G = (V, E)$  be a graph. A set of nodes  $I \subset V$  is an independent set of  $G$  if  $\forall v_i, v_j \in I, e = (v_i, v_j) \notin E$ .  $Y$  is a maximal independent set of  $G$  if (1)  $Y$  is an independent set and (2)  $Y$  is not contained in another independent set.

**Algorithm TCM:**

1. Formulate the problem as above and use Algorithm DP to solve it. However, replace step 3 of Algorithm DP by the following: For every solution set  $(S_k, c_k)$  in  $M_n$ , if  $c_k > c_{tc}$ , mark it as a candidate. Let the number of candidates be  $z$ . Relabel the candidates such that they are represented as  $(S_k, c_k), k = 1, \dots, z$ .
2. Construct a weighted graph  $G=(V,E,C)$  as follows: For each candidate  $1 \leq k \leq z$  associated with  $(S_k, c_k)$ , there is a corresponding node  $v_k \in V$  and a label  $c_k \in C$ . An edge  $e = (v_i, v_j) \in E$  if  $(S_i, c_i) \cap (S_j, c_j) \neq \emptyset$ .
3. Let  $Q$  be the collection of all maximal independent sets of  $G$ . Find  $X \in Q$  so that  $\sum_{v_i \in X} c_i$  is maximal.
4. For each node  $v_i \in X$ , allocate a new test channel. For all chips in  $S_i$  of the corresponding candidate, assign the design style for their on-chip controllers as bus-dependent. Connect these chips to the newly allocated test channel. Repeat this step until all nodes in  $X$  are processed. Let  $m$  be the number of test channels allocated in this step.
5. Let  $U$  be the set of chips that do not belong to any candidate in  $X$ . If  $U \neq \emptyset$  allocate a new test channel. All chips in  $U$  are connected to this test channel. Apply Algorithm TC1 to  $U$  and find a set of chips  $U_b$  that have bus-dependent controller. Connect all chips in  $U_b$  to test ring 0, and all chips in  $U - U_b$  to test ring 1 of the newly allocated test channel. Note that all chips in  $U - U_b$  have autonomous controller. □

In step 1 a candidate is a feasible set found using Algorithm DP with  $c_k \geq c_{tc}$ . This means that making all chips in a candidate set as bus-dependent can satisfy the time constraint and the complexity is reduced since the cost of allocating a new test channel is less than that of making these chips autonomous. In step 2 the optimization problem is formulated as a graph so that it can be solved in step 3 by finding the maximal independent set for a graph. Computing  $X$  in step 3 in general requires exponential time since finding a maximal independent set of a graph is a known NP-Complete problem [24]. An algorithm (Algorithm MIS), that uses a branch and bound technique to find  $X$  efficiently, is given below. Efficiency

is obtained by pruning the solution space. The number of test channel allocated in step 4 is maximal. This leads to the minimization of the overall controller complexity. If the complexity of a test channel  $c_{ic}$  is very large such that no candidate can be found ( $z = 0$ ), then steps 2, 3 and 4 are not executed and the set  $U$  in step 5 consists of the entire set of chips. For this case both Algorithms TCM and TC1 generate the same result.

**Algorithm MIS:**

1. Initialization: Set  $X = Y = \emptyset$ .  $X$  contains the current “best” independent set.
2. Call  $BB(Y, 1)$ ;
3.  $X$  is a maximal independent set and  $\sum_{i \in X} c_i$  is maximal.

**Function  $BB(Y, i)$ :**

1. If  $i > n$  return; otherwise continue.
2.  $Y = Y \cup \{v_i\}$ .
3. If  $Y$  is an independent set then
  - (a) if  $(|X| < |Y|)$  or  $(|X| == |Y|$  and  $\sum_{v_i \in X} c_i < \sum_{v_i \in Y} c_i)$ , set  $X = Y$ .
  - (b) Call  $BB(Y, i + 1)$
4. Call  $BB(Y - \{v_i\}, i + 1)$ . □

Using a recursive approach, the function  $BB$  searches through the entire solution space for  $Y$  (branch) and intelligently prunes off a subspace if it does not contain a solution better than the best solution seen to date. Pruning is performed in step 3 of  $BB$ . When an independent set  $Y$  is found,  $X$  is updated only if an independent set with more nodes is found ( $|X| < |Y|$ ) or an independent set with higher value is found ( $|X| == |Y|$  and  $\sum_{v_i \in X} c_i < \sum_{v_i \in Y} c_i$ ). If  $Y$  is not independent, the subspace containing  $Y$  is no longer considered, e.g., if  $\{v_1, v_2\}$  is not independent, all set of nodes that contain both  $v_1$  and  $v_2$  will not be checked for independence.

**Lemma 3** *The solution found by Algorithm TCM is optimal.*



**Proof:**

The solution set  $X$  is optimal in that the test time for the resulting design is less than or equals  $T$  and the overall controller complexity is minimal. This is obvious since the algorithm implicitly searches through the entire space.  $\square$

### 5.2.3 Results

Let a chip be represented by a two-tuple  $(c_i, t_{bi})$ , where  $c_i$  and  $t_{bi}$  have been defined previously.

**Example 5.1:** Consider a module consisting of five chips, numbered from 1 to 5. These chips can be represented as  $(6,1)$ ,  $(11,1)$ ,  $(17,3)$ ,  $(3,2)$  and  $(9,2)$ , respectively. Using Algorithm TC1, the solution set found is  $S=\{1, 2, 3\}$ . Using Algorithm TCM, the results are as follows, where  $n=5$  and  $T=5$ .

| $c_{tc}$ | num. of candidates $z$ | num. of TC allocated $m$ |
|----------|------------------------|--------------------------|
| 10       | 10                     | 3                        |
| 15       | 6                      | 2                        |
| 20       | 3                      | 1                        |
| 30       | 1                      | 1                        |
| 40       | 0                      | 0                        |

The value in the last column is the number of test channels allocated in step 4 of Algorithm TCM. Note that even with  $m=0$ , Algorithm TCM will still allocate one test channel in step 5.

**Example 5.2:** Consider a module consists of seven chips, numbered from 1 to 7, and is represented as  $(299,4)$ ,  $(73,1)$ ,  $(159,2)$ ,  $(221,3)$ ,  $(137,2)$ ,  $(89,1)$  and  $(157,2)$ , respectively. Using Algorithm TC1, the solution set is  $S=\{1,2,3,6,7\}$ . Using Algorithm TCM, the results are as follows, where  $n=7$  and  $T=10$ .

| $c_{tc}$ | num. of candidates $z$ | num. of TC allocated $m$ |
|----------|------------------------|--------------------------|
| 500      | 44                     | 2                        |
| 550      | 32                     | 1                        |
| 600      | 27                     | 1                        |
| 700      | 10                     | 1                        |
| 800      | 0                      | 0                        |

### 5.3 Discussions

Algorithm TCM can also be used in the design of system level controllers. For example, if each MMC can only control a test channel, then Algorithm TCM can determine the number of MMC required such that a system can be tested in a predetermined time.

#### Chip Type Constraint

When two chips have the same application circuit, it is beneficial to let them have the same type of on-chip controller so that only one type of chip need to be manufactured. The algorithms presented above can deal with this constraint if the problem is modeled as follows. Replace the set of chips that have the same application circuit with a new pseudo-chip. The test time and the complexity of the pseudo-chip is the sum of the test time and the complexity of the chips deleted, respectively. Repeat this process until the constraint on the chip type is completely removed. The new problem can then be solved by the previously proposed algorithms.

#### Chips Having Autonomous Controller

The test time required by chips having an autonomous controller is neglected in both Algorithm TC1 and TCM. It is assumed that these chips are connected to a test ring and their tests are executed concurrently while the test ring stays in the *RunTest* bus state. The assumption is invalid if an autonomous chip cannot stop the self-test process by itself. In this case, a test ring should be assigned to each chip. The controller complexity is thus increased.

This problem can be solved if the test channel is modified such that it can support many test rings while data can be sent to only one ring at a time. In

such a case, the complexity of the test channel remains low and all chips having autonomous controller can be tested concurrently.

### Controller with Multi-level Complexity

For both Algorithm TC1 and TCM, the on-chip controller is assumed to have only two levels of complexity,  $a_i$  and  $b_i$ . However, it is possible to have controllers with many levels of complexity, as shown in section 5.1. For example, the complexity of chip  $i$  can be further classified into  $b_{ij}$ , where  $j = 1, \dots, i_b$  if it has a bus-dependent controller, and  $a_{ij}$ , where  $j = 1, \dots, i_a$  if it has an autonomous controller. The test time for the chip is  $t_{ij}^b$  if the complexity is  $b_{ij}$ , and  $t_{ij}^a$  if the complexity is  $a_{ij}$ . Using one test channel, the problem can be formulated into an integer linear programming problem as follows.

Let the variable  $x_{ij}$  be defined as

$$x_{ij} = \begin{cases} 1 & \text{if the complexity of the on-chip controller is } a_{ij} \\ 0 & \text{otherwise} \end{cases}$$

and the variable  $y_{ij}$  be defined as

$$y_{ij} = \begin{cases} 1 & \text{if the complexity of the on-chip controller is } b_{ij} \\ 0 & \text{otherwise.} \end{cases}$$

The objective is to minimize

$$\sum_{i=1}^n \left( \sum_{j=1}^{i_a} x_{ij} a_{ij} + \sum_{j=1}^{i_b} y_{ij} b_{ij} \right)$$

subject to

$$\begin{aligned} \sum_{j=1}^{i_a} x_{ij} + \sum_{j=1}^{i_b} y_{ij} &= 1, \quad \forall i, \text{ and} \\ \sum_{i=1}^n \sum_{j=1}^{i_b} y_{ij} t_{ij}^b &\leq T. \end{aligned}$$

## Chapter 6

### Interconnect Test Generation

Previous work on the diagnosis of faults in a wiring network has been based on the assumption that both open and short faults do not exist on the same net. When this assumption is relaxed, the results obtained fail to identify all diagnosable faults. The non-diagnosability of these faults, including shorts between nets, represents a deficiency. In this chapter the causes for this deficiency are analyzed and explained. New test algorithms and theoretical results for correcting this deficiency are also provided. Based on these results, a test set that is capable of identifying all diagnosable faults in a wiring network with arbitrary open and short faults is developed. Finally, two adaptive diagnosis algorithms which can reduce the number of test vectors while retaining the same level of diagnostic resolution are delineated.

#### 6.1 Introduction

Detecting and locating faults in wiring networks on a printed circuit board has drawn much attention since the emerging of the boundary scan architecture [33]. In this architecture each primary input/output pin of a chip is associated with a boundary scan (B-S) cell. Each chip has a boundary scan register consisting of all the B-S cells. During the test mode a scan chain is formed by cascading boundary scan registers of several chips. Through this chain a test controller can access the I/O pins of every chip. Thus a virtual bed-of-nails capability is achieved. With



this capability the wiring nets can be isolated from the chips and tested without the need to physically probe the board. In this way it is possible to test the new generation of boards which allow limited probing due to the use of surface mounted devices and tape automated bonding technology. Note that if non-boundary scan devices are used, physical probes may still be required.

Many papers have dealt with the problem of finding test sets for detecting and locating faults in a wiring network [16, 25, 29, 30, 34, 37, 56, 64, 68]. Usually opens are modeled as stuck-at faults and are diagnosed separately from the shorts. This is based on the assumption that a net cannot be simultaneously associated with both open and short faults. Very comprehensive results are presented by Jarwala and Yau [34], where a framework for the detection and diagnosis of wiring networks is discussed. In particular, the *diagonally independent* property is identified. It is shown that a test set with this property is sufficient for the diagnosis of all shorts in one-step, where the results are analyzed after all test vectors have been applied.

However, as shown in this chapter, when the assumption concerning open and short faults is relaxed, a test set having the diagonally independent property is insufficient for achieving complete diagnosis without repair. In fact, there are certain faults that cannot be diagnosed without repair. Some of these non-diagnosable faults are listed in section 6.2.3. Furthermore, it is shown that none of the previous results can identify all diagnosable faults. The causes for this deficiency are identified and characterized in Lemmas 4 and 5. A diagnostic level DR5, which refers to the case where all diagnosable faults can be identified, has been formulated. A term called maximal diagnosis is defined using two conditions. It can be shown that these conditions are both necessary and sufficient for achieving the diagnostic level DR5.

Both one-step and two-step diagnosis are addressed in this chapter. In the former case, responses are analyzed only after all test vectors have been applied. In the latter case, responses are analyzed after a fixed part of the test vectors have been applied. Based on this analysis, additional test vectors are then generated and applied. Final analysis is then carried out to identify the faults.



For one-step diagnosis, a property called *set-cover independent* is identified. Based on this property a fundamental theorem on diagnosing wiring faults is presented. The theorem gives both the necessary and sufficient conditions for identifying all diagnosable faults. A universal test set is also presented. This test set can achieve maximal diagnosis for an arbitrary network without assuming a specific fault model.

For the two-step diagnosis case, two adaptive diagnosis algorithms are presented. Compared with one-step diagnosis, these algorithms can reduce the number of test vectors while retaining the same level of diagnostic resolution by using a two-step scheme. In the first step, a detection sequence is applied and the responses are evaluated. Based on the initial results, a second sequence is applied to achieve the required diagnosis. The test vector size is reduced since some information about the network is employed in generating the second sequence.

## 6.2 Preliminaries

A wiring network consists of many nets. A net contains one or more drivers and one or more receivers. The logic value of a net can be controlled via one of its drivers and observed by all of its receivers. For a multi-driver net, only one driver can be enabled at a time; the others must be disabled. In addition, while testing a wiring network, only the drivers and receivers of nets are accessible. A fault-free net can transfer the logic value from an enabled driver to its receivers correctly. A receiver of a fault-free net can only receive from its associated drivers. The objective of diagnosis of a wiring network is to find a set of test vectors which can be applied to identify as many faults in the network as possible without repair.

### 6.2.1 Fault Model

Two types of physical faults, namely open and short, are assumed. More than one physical break is possible in an opened net. Ignoring fan-out nodes and shorts, multiple opens are modeled as a single open along a wire segment. Also more than

one physical bridge are possible between two shorted nets. Multiple shorts are modeled as a single short between two wire segments.

If two or more nets are shorted, the resulting behavior can be modeled as either (1) a wired-OR, (2) a wired-AND, or (3) a strong-driver, where one driver dominates the resulting behavior. In all cases, all nets involved in a short will have the same resulting logic value. The wired-OR fault model is assumed in this discussion unless otherwise stated.

A net shorted to a power line VCC (GND) will exhibit a stuck-at 1 (stuck-at 0) behavior. If a net contains an open, the logic value interpreted by all floating receivers of the opened net will be the same, which could be either a soft stuck-at 1 or a soft stuck-at 0. The logic value of a net shorted to a wire segment having a soft logic value cannot be forced to the soft value. The soft stuck-at 1 model is assumed for a floating net unless otherwise stated. In Figure 6.1, the logic value of the point A is soft stuck-at 1.

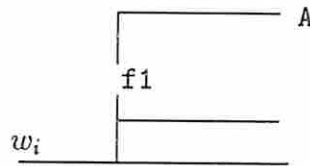


Figure 6.1: A soft stuck-at 1 case.

Both opens and shorts can occur on the same net. If a net contains both an open and a short, the logic value received by the receivers is determined by the combined effect of these faults. A short to an open net is illustrated in Figure 6.2, where A takes the logic value of B.

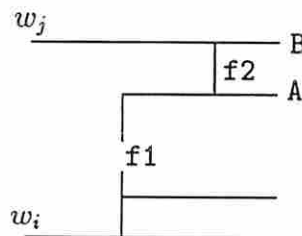


Figure 6.2: A short to an opened net.

## 6.2.2 Notation and Definitions

The notation and definitions used in this chapter follow the conventions established in [34]. For convenience, some of this information is repeated here.

- Parallel Test Vector ( $PTV$ ): the vector applied to all nets of a wiring network in parallel.
- Sequential Test Vector ( $STV$ ): the vector applied to a net, over a period of time, by a sequence of  $PTVs$ .
- Test Set (or test sequence)  $S$ : the collection of all  $STVs$ . Each column of  $S$  is a  $PTV$  and each row of  $S$  is a  $STV$ .
- Sequential Response Vector (SRV): the response of a net to a  $STV$ .
- Syndrome: the SRV of a faulty net.
- Aliasing syndrome: the resulting syndrome of a set of faulty nets is the same as the correct SRV of a net not in the set.
- Confounding syndrome: the syndromes that results from multiple independent faults are identical.

The following definitions are also used.

- OR-Cover: A vector  $V_i$  *OR-covers* another vector  $V_j$  if for every bit position in  $V_j$  that is 1, the corresponding bit in  $V_i$  is also 1. For example,  $STV_i=(1101)$  OR-covers  $STV_j=(0101)$ , or  $STV_j$  is OR-covered by  $STV_i$ . The OR-cover is used in the wired-OR fault model. In a similar fashion one can define an AND-cover for the wired-AND fault model. In this chapter, the term OR-cover is abbreviated as *cover*.
- Independent set: A test set  $S$  is an *independent set* if no  $STV_i$  is covered by another  $STV_j, j \neq i$ .

- Set-cover: Let  $V_J$  be the result of wire-ORing a set of vectors  $V_{j1}, \dots, V_{jk}$ . A vector  $V_i$  *set-covers* the vectors  $V_{j1}, \dots, V_{jk}$  if  $V_i$  covers  $V_J$ .
- Set-covering syndrome: A *set-covering syndrome* is a syndrome that results from a set of shorted nets ( $W'$ ) that either covers a SRV or is covered by the SRV of some net  $w_i$  not in  $W'$ .
- Set-cover independent: Let  $S = (STV_1, \dots, STV_n)^T$  be a test set for a set of nets  $W = (w_1, \dots, w_n)$ .  $S$  is *set-cover independent* if for  $i = 1, \dots, n$ ,  $STV_i$  is neither covered by nor covers the union (for wired-OR, intersection for wired-AND) of any subset of vectors in  $S - \{STV_i\}$ . In the other words, for every  $SRV_i$  in  $S$  no set-covering syndrome can exist.

### 6.2.3 Non-Diagnosable Faults

A fault  $f$  is said to be *non-diagnosable* if there is no test set  $S$  and an algorithm  $A$  such that by applying  $S$  to the network and processing the responses using algorithm  $A$ ,  $f$  can be identified. Note that all single faults are diagnosable. Based on the fault model presented, some faults are non-diagnosable. Some of these non-diagnosable faults are listed below.

- In a set of nets that are shorted with each other, there are some opens that are non-diagnosable. For example, in Figure 6.3 the open fault  $f1$  on net  $w_2$  is non-diagnosable. Since  $w_1$  and  $w_2$  are electrically common, it is impossible to find a test to detect the open.

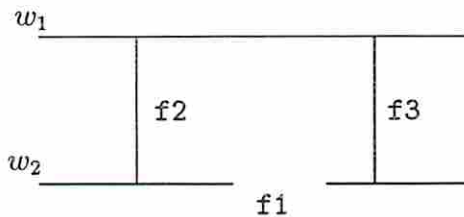


Figure 6.3: An open that is non-diagnosable.



- The short between a set of opened nets is non-diagnosable. For example, in Figure 6.4, it is impossible to identify the short f1 between  $w_1$  and  $w_2$  since no receivers are connected to the shorted wires.

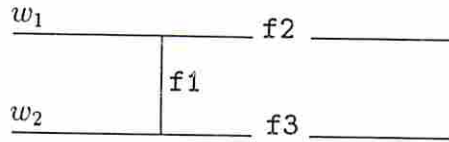


Figure 6.4: A short that is non-diagnosable.

- There exists three possible reasons for a faulty net that has all-1 responses to a test, namely (1) the net is shorted with a VCC power line, (2) the net is opened and floating (soft stuck-at 1 model), and (3) the net is shorted with other nets such that the combined result is an all-1 vector (wired-OR model). The third case can be distinguished from the others by applying an all-0 PTV. The first two cases cannot be distinguished.

#### 6.2.4 Diagnostic Resolution

Various levels of diagnostic resolution are possible in testing a wiring network without repair. Listed below are six such levels. They are listed in ascending order of their diagnostic resolution, i.e., DR1 has the lowest diagnostic resolution, and DR6 has the highest diagnostic resolution.

**DR1:** Determine whether the entire network is fault-free.

**DR2:** Identify all faulty nets.

**DR3:** For each and every net, determine whether it is fault-free without knowing the response of the other nets.

**DR4:** Identify all faulty nets. In addition, for nets without shorts, identify the existence of nets having opens. For a faulty net without open faults, identify all nets that are shorted to it.



**DR5:** Identify all faulty nets. In addition, identify all faults that are diagnosable.

**DR6:** Identify all faulty nets. In addition, identify all the opens and shorts in the network.

In DR1, one is only interested in determining the health status of the entire network. No further diagnostic information is provided. In DR2, all faulty nets are identified. No information about what type of faults associated with each net is provided. In DR3, all faulty nets are identified. In determining the health status of net, only its response is required. No information about the response of other nets are needed. This scheme is most suitable for a built-in self-test type of design. In DR4, all faulty nets are identified. The faults associated with each nets can be identified if they belong to those cases described above. In DR5, all faulty nets are identified. More faults can be identified than in the case of DR4. In DR6, all faulty nets are identified. In addition, all the faults, including opens and shorts, are identified.

For the purpose of repairing a wiring network, it is desirable that as many faults as possible be identified. Due to the fact that some faults cannot be identified without first repairing other faults, DR6 cannot be reached. An example would be a net with multiple opens. In this work, the focus is to achieve DR5 without repair.

### 6.2.5 Previous Results

Previous results focus on diagnostic resolution ranging from DR1 to DR4. Some typical results for the testing of a network consisting of 4 nets are listed below. These results are based on the assumption that both opens and shorts cannot exist on the same net.

*Counting Sequences:* [37]

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

This test set consists of a simple counting sequence, where the all-0 and all-1 STVs are not used. This test can achieve the DR1 diagnostic levels. The size of the test set is  $\lceil \log(n+2) \rceil$ , where  $n$  is the number of nets.

*Complementary Counting Sequence:* [64]

$$S = \left[ \begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{array} \right]$$

This test set consists of a counting sequence and its complement. The all-0 and all-1 STVs can be used. The size of the test set is  $2\lceil \log n \rceil$ . This test set can achieve diagnostic level DR3. By eliminating the aliasing syndromes, the *self-diagnosis* property is achieved. This allows the determination of the health status of a net by examining only its *SRV*.

*Maximal Independent Set:* [16]

$$S = \left[ \begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{array} \right]$$

Constant weight codes, where every STV has the same number of 1s, is a class of independent test sets. These test sets can achieve diagnostic level DR3. The size of the test set is minimal for self-diagnosis when the number of 1s in a SRV is half of the number of PTVs, which is referred to as a maximal independent set [16].

*Diagonally Independent Sequence:* [34]

$$S = \left[ \begin{array}{cccc} x & x & x & 1 \\ x & x & 1 & 0 \\ x & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array} \right]$$

The  $x$  represents either a 0 or a 1. This test set can achieve the diagnostic level DR4. Both aliasing and confounding syndromes can be eliminated. All pairs of nets that are shorted are identified.

### 6.2.6 Deficiencies in Previous Approaches

Recall that these results are based on the assumption that both opens and shorts cannot exist on the same net. However, when this assumption is relaxed, there exists certain types of opens and shorts that cannot be identified. For example, the diagonally independent sequence

$$S = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

cannot identify the short fault  $f1$  in Figure 6.5 nor the open fault  $f2$  in Figure 6.6.

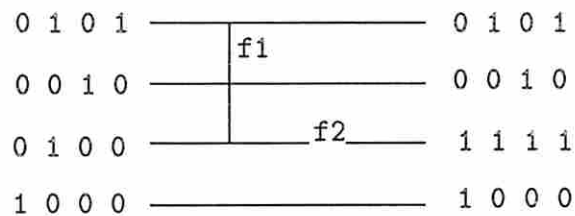


Figure 6.5: A short that cannot be identified by a diagonally independent sequence.

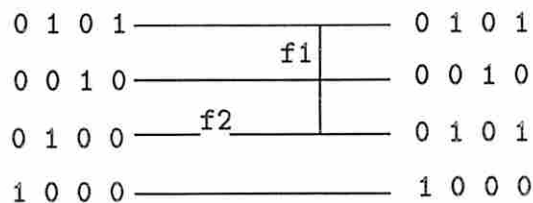


Figure 6.6: An open that cannot be identified by a diagonally independent sequence.



In summary, none of the previous approaches which include the diagonally independent sequence [34], the maximal independent set[16], and the complementary counting sequence[64] can identify all the faults described above in one pass without repair.

The existence of unidentifiable shorts and opens in a network represents a deficiency in diagnosis. A test set that can be used to identify these faults is presented next.

### 6.3 One-Step Diagnosis

Due to the existence of non-diagnosable faults in a wiring network, it is impossible to identify all faults without repair or access to points of the nets other than the drivers and receivers. The term **maximal diagnosis** is defined as follows.

Let  $W = (w_1, w_2, \dots, w_n)$  be a set of nets to be tested,  $D_i$  be the set of drivers of net  $w_i$ , and  $R_i$  be the set of receivers of net  $w_i$ . A test set  $S$  achieves *maximal diagnosis* for  $W$  if the following two conditions are verified.

- Condition C1: For  $i = 1, \dots, n$ , by analyzing the responses obtained from the application of  $S$ , the existence of the connection  $(D_i, R_i)$  can be determined. The *connection*  $(D_i, R_i)$  exists if for all  $k$ , each driver  $d_{ik} \in D_i$  can *transfer* its logic value to all receivers in  $R_i$  correctly. A driver in  $D_i$  is said to *transfer* its logic value to a receiver  $R_i$  if  $SRV_i$  covers  $STV_i$ . Note that only one driver can be enabled for a given net at one time. In other words, if the connection  $(D_i, R_i)$  exists, then the application of  $STV_i$  to the enabled driver of  $w_i$  will make one of the following statements true: (1)  $SRV_i = STV_i$ , or (2)  $SRV_i$  covers  $STV_i$ .
- Condition C2: For all  $i, j, i \neq j$ , by analyzing the responses obtained from the application of  $S$ , the existence of the connection  $(D_i, R_j)$  can be determined. The connection  $(D_i, R_j)$  does not exist if for all  $k$ , each driver  $d_{ik} \in D_i$  does not transfer its logic value to any receivers in  $R_j$ . In other words if the connection



$(D_i, R_j)$  does not exist, the application of  $STV_i$  to the enabled driver of  $w_i$  will make one of the following statements true: (1)  $SRV_j \neq STV_i$ , or (2)  $SRV_j$  does not cover  $STV_i$ .

**Theorem 1** *Diagnostic level DR5 can be achieved by a test set  $S$  iff  $S$  achieves maximal diagnosis.*

**Proof:** (*if* part)

The test set  $S$  verifies both Condition C1 and C2 since  $S$  achieves maximal diagnosis for a network  $W$ . By definition DR5 is achieved if all diagnosable faults in  $W$  can be identified. There are two type of faults in  $W$ , namely opens and shorts. The diagnosability of these faults by  $S$  are discussed separately.

(1) opens: An open on a net  $w_i$  can be diagnosed if the connection  $(D_i, R_i)$  does not exist. Since  $S$  can achieve maximal diagnosis, the connectivity is determined in Condition C1. Thus  $S$  can identify all diagnosable opens in  $W$ .

(2) shorts: A short between two nets  $w_i$  and  $w_j$  can be diagnosed if one of the following is true: (a) there exists a driver set  $D_k$  such that both connections  $(D_k, R_i)$  and  $(D_k, R_j)$  exist; (b) there exists a receiver set  $R_k$  such that both connections  $(D_i, R_k)$  and  $(D_j, R_k)$  exist. The existence of these connections can be determined by  $S$  using Condition C2. Thus  $S$  can identify all diagnosable shorts in  $W$ .

From (1) and (2), all diagnosable faults can be identified. Therefore the diagnostic level DR5 can be achieved.

(*only if* part)

Assume that  $S$  cannot achieve maximal diagnosis. By definition, there exist at least one connection for which the existence cannot be determined. Two cases are possible: (1) if the connection is of the form  $(D_i, R_i)$  then there can be an open fault on net  $w_i$  that cannot be identified; (2) if the connection is of the form  $(D_i, R_j), i \neq j$ , then there can be a short between  $w_i$  and  $w_j$  that cannot be identified. In both cases, the faults can be identified by a walking ones sequence. Thus, by definition, these faults are diagnosable. Therefore the diagnostic level DR5 cannot be achieved. Thus the maximal diagnosis property is necessary.  $\square$

In this chapter the generation of test sets that achieves maximal diagnosis is discussed. The generated test sets thus achieves diagnostic level DR5 in which all diagnosable faults are identified. This is the best diagnostics possible without accessing points on the nets other than the drivers and receivers.

**Lemma 6** *For the wired-OR (wired-AND) model, any test set  $S$  is set-cover independent iff  $S$  has the walking ones (zeros) sequence as its subsequence.*

**Proof:** (*if* part)

This is obvious since the walking ones sequence is set-cover independent.

(*only if* part)

Suppose that the test set  $S$  is set-cover independent. For a given  $STV_i$ , there must exist a PTV such that its  $i$ th bit is 1 and all other bits are 0. This is true for all  $STV_i, i = 1, \dots, n$ . By arranging  $S$  properly (by swapping rows and columns), a walking ones sequence can be constructed. Therefore,  $S$  contains a walking ones subsequence.  $\square$

In the following, a theorem that characterizes the test set which achieves maximal diagnosis is presented.

**Theorem 2** *A test set  $S$  achieves maximal diagnosis for a network  $W$  in one-step iff  $S$  is set-cover independent.*

**Proof:** (*if* part)

Suppose that  $S$  is set-cover independent. No set-covering syndrome can exist. For each and every net  $w_i$ , Condition C1 can be verified by checking whether  $SRV_i$  covers  $STV_i$ . If this is not true then there is at least an open fault between the driver and the receivers of the net  $w_i$ . Since no set-covering syndrome can exist, no drivers of other nets can cover the  $STV_i$ .

Condition C2 can be verified as follows. If  $STV_i$  cannot cover  $SRV_i$ , then for every bit in  $SRV_i$  that is not covered by the  $STV_i$ , there is a short between the receiver of  $w_i$  and a driver  $w_j$  whose  $STV_j$  covers that bit.

Since both Conditions C1 and C2 can be verified, maximal diagnosis is achieved and the sufficiency aspect of the theorem has been demonstrated.

(only if part)

Suppose that S is not set-cover independent. There exists at least one  $STV_i$  that is covered by another  $STV_j$ . When the following two faults occur, the existence of the connection  $(D_i, R_i)$  cannot be determined: (1) a short between  $w_i$  and  $w_j$ , and (2) an open on  $w_i$  that is closer to the driver than the short. This means that Condition C1 cannot be satisfied. Thus, by definition, maximal diagnosis is not achieved.  $\square$

From Lemma 6 and Theorem 2, one can conclude that any test set that can achieve maximal diagnosis must have a walking ones (zeros) sequence as its subsequence. From Theorem 2 and 1 it can be concluded that a set-cover independent test set can achieve the diagnostic level DR5.

**Example 6-1:** The short that could not be identified by a diagonally independent sequence in Figure 6.5 can be identified by a set-cover independent sequence (see Figure 6.8).

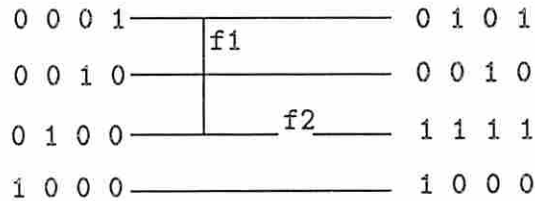


Figure 6.8: Achieving maximal diagnosis using a set-cover independent sequence.

**Universal Test Set:**

Assuming the wired-OR model and that a floating net is modeled as a soft stuck-at 1, a test set that can achieve maximal diagnosis for a network consisting of three nets can be constructed as follows.

$$\left[ \begin{array}{c|ccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{array} \right]$$

The all-0 PTV is used to distinguish between the cases (1) all nets are shorted together (all 1s for SRVs) and (2) all nets are opened.

Similarly, if the wired-AND model is used and an open and floating net is modeled as a soft stuck-at 0, a test set for maximal diagnosis can be constructed by a walking zeros sequence followed by an all-1 PTV.

In summary, a universal test set for maximal diagnosis, without making any assumption on the nature of the faults, is as follows.

$$S_{universal} = \left[ \begin{array}{c|ccc|c|ccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} \right]$$

## 6.4 Two-Step Diagnosis

Two-step diagnosis refers to the fact that diagnosis is done by applying two test sequences. The results of the first test sequence is used to generate the second test sequence. This type of diagnosis is also known as adaptive diagnosis.

Two adaptive algorithms that can achieve maximal diagnosis with a reduced number of PTVs are presented next. The test sets for both algorithms do not have the set-cover independent property since certain information about the network is employed in generating the second test sequence.

### 6.4.1 Adaptive Algorithm A1

1. Apply a maximal independent set ( $S_M$ ). Collect and analyze the responses. Stop if no faults are detected.
2. Partition the nets into two groups. The partitioning is done as follows. For a net  $w_i, i = 1, \dots, n$ , if (a)  $STV_i = SRV_i$  and (b)  $SRV_i$  is unique, include  $w_i$  into Group 0, else Group 1.



3. Apply a walking ones sequence  $S_F$  to all nets in Group 1, and all-0 vectors to all nets in Group 0.

The objective of the first sequence is to achieve the self-diagnosis property by eliminating the aliasing syndromes. The maximal independent set is the minimal size test set that can achieve this objective [16]. The number of PTVs required by the first sequence is  $p$ , where  $p$  is the smallest integer satisfying  $C_{\lfloor p/2 \rfloor}^p \geq n$ , and  $C_{\lfloor p/2 \rfloor}^p$  represents possible combinations choosing  $\lfloor p/2 \rfloor$  items out of  $p$  items. The total number of PTVs required by this algorithm is  $p + F$ , where  $F$  is the number of nets in group 1.

**Theorem 3** *The test set derived from Algorithm A1 achieves maximal diagnosis.*

**Proof:**

Let  $W_0$  and  $W_1$  be the set of nets in group 0 and 1, respectively, after the completion of step 2. Let  $D_{0i}$ ,  $R_{0i}$  and  $D_{0j}$ ,  $R_{0j}$  ( $i \neq j$ ), be the set of drivers and receivers of nets  $w_{0i}$  and  $w_{0j}$ , respectively, where  $w_{0i}$  and  $w_{0j} \in W_0$ . Similarly, let  $D_{1i}$ ,  $R_{1i}$  and  $D_{1j}$ ,  $R_{1j}$  ( $i \neq j$ ), be the set of drivers and receivers of nets  $w_{1i}$  and  $w_{1j}$ , respectively, where  $w_{1i}$  and  $w_{1j} \in W_1$ .

There are six types of connections need to be checked, namely  $(D_{0i}, R_{0i})$ ,  $(D_{0i}, R_{0j})$ ,  $(D_{0i}, R_{1j})$ ,  $(D_{1i}, R_{1i})$ ,  $(D_{1i}, R_{0j})$ ,  $(D_{1i}, R_{1j})$ , for all  $i \neq j$ .

The connections  $(D_{0i}, R_{0i})$  do exist since (a) in the sequence  $(S_M)$ , it is impossible to find a subset from  $S - \{STV_{0i}\}$ , whose wired-OR result equals  $STV_{0i}$ , (b)  $STV_{0i} = SRV_{0i}$  and (c)  $SRV_{0i}$  is unique.

The connections  $(D_{0i}, R_{1j})$  do not exist for the following reason. For each  $i$ , if the connections exist,  $SRV_{0i}$  must equal  $SRV_{1j}$  since there is no open on  $w_{0i}$ . However  $SRV_{0i}$  is unique, so we know that these connections do not exist.

The connections  $(D_{0i}, R_{0j})$ ,  $\forall i \neq j$  does not exist since the existence of them will make  $SRV_{0i} = SRV_{0j}$ , which contradicts the fact that  $SRV_{0i} \neq SRV_{0j}$ .

The connections  $(D_{1i}, R_{1i})$ ,  $(D_{1i}, R_{0j})$  and  $(D_{1i}, R_{1j})$  are checked by the walking ones sequence  $S_F$ .



Since the existence of all connections can be determined, Algorithm A1 achieves maximal diagnosis.  $\square$

**Example 6-2:** Let  $W$  be a network of four nets, where  $w_1$  and  $w_2$  are two nets in group 0 and  $w_3, w_4$  are in group 1. Let  $S_{A1}$  be the test generated by Algorithm A1,

$$S_{A1} = (S_F, S_M) = \left[ \begin{array}{cc|cccc} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{array} \right]$$

Maximal diagnosis can be achieved in this example since the existence of all connections can be determined. The efficiency of this algorithm depends on the value of  $F$ . In the case when  $F \approx n$ , the number of PTVs is close to that of a walking ones sequence.

### 6.4.2 Adaptive Algorithm A2

The second adaptive algorithm follows.

1. Apply a maximal independent set ( $S_M$ ). Collect and analyze the responses. Stop if no faults are detected.
2. Partition nets into Group 0 and 1 (as in Algorithm A1).
3. Partition nets in Group 1 such that all nets with the same  $SRV$  are in the same group. Number these new groups from 1 to  $G$ . Let  $K$  be the cardinality of the largest group.
4. Apply a walking ones sequence  $S_G$  to all groups in parallel except to Group 0 for which the all-0 vectors are applied.
5. Apply another walking ones sequence  $S_K$  across groups. That is, all nets in the same group are modeled as a single net. Again the all-0 vectors are applied to all nets in Group 0. The number of PTVs is  $G$ .

The total number of PTVs for Algorithm A2 is  $p + G + K$ . In general this algorithm requires fewer PTVs than Algorithm A1. This is because (1) the  $F$  nets in Group 1 of Algorithm A1 are now partitioned into  $G$  groups in Algorithm A2, and (2)  $F + 1 \geq G + K$ .

**Theorem 4** *The test set derived from Algorithm A2 achieves maximal diagnosis.*

**Proof:**

Let  $W_0$  be the group of fault-free nets and  $W_x, W_y$  be any two of the  $G$  groups formed in step 3. Let  $D_{ai}, R_{ai}$  and  $D_{aj}, R_{aj}$  ( $i \neq j$ ), be the set of drivers and receivers of nets  $w_{ai}$  and  $w_{aj}$ , respectively, where  $w_{ai}$  and  $w_{aj} \in W_a, a = 0, x, y$ .

There are twelve types of connections to check, namely  $(D_{0i}, R_{0i}), (D_{0i}, R_{0j}), (D_{0i}, R_{xj}), (D_{0i}, R_{yj}), (D_{xi}, R_{xi}), (D_{xi}, R_{0j}), (D_{xi}, R_{xj}), (D_{xi}, R_{yj}), (D_{yi}, R_{yi}), (D_{yi}, R_{0j}), (D_{yi}, R_{yj}), (D_{yi}, R_{xj}), \forall i \neq j$ .

The connections  $(D_{0i}, R_{0i}), (D_{0i}, R_{0j}), (D_{0i}, R_{xj}), (D_{0i}, R_{yj}), (D_{xi}, R_{0j}), (D_{yi}, R_{0j})$  are checked by  $S_M$  for the reasons stated in the proof of Theorem 3.

The existence of connections  $(D_{xi}, R_{xj})$  or  $(D_{yj}, R_{xj})$  or both is denoted by  $(D_{xi}|D_{yj}, R_{xj})$ . After the application of  $S_G$  the existence of  $(D_{xi}|D_{yj}, R_{xj})$  can be determined. The  $S_K$  can then easily distinguish between the three possible cases covered by  $(D_{xi}|D_{yj}, R_{xj})$ . Thus, with the application of both  $S_G$  and  $S_K$ , the existence of the remaining six type of connections can be checked.

Both Conditions C1 and C2 are verified since all the twelve types of connections are checked. This concludes that Algorithm A2 achieves maximal diagnosis.

□

**Example 6-3:** Let  $W$  be a network with seven nets. After the application of  $S_M$ , three groups are formed. Let  $w_1, w_2$  be in Group 0,  $w_3, w_4$  be in Group 1, and  $w_5, w_6, w_7$  be in Group 2. In this example  $n = 7, G = 2$  and  $K = 3$ . According to Algorithm A2, the total test set  $S_{A2}$  consists of a maximal independent set of five PTVs ( $S_M$ ), followed by three walking ones PTVs ( $S_G$ ), which are again followed by another two walking ones PTVs ( $S_K$ ).

$$S_{A2} = \left[ \begin{array}{cc|ccc|ccccc} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array} \right]$$

### 6.4.3 Comparison with Other Adaptive Algorithms

Several adaptive algorithms have been proposed previously. In the following these algorithms will be briefly reviewed and compared to Algorithms A1 and A2.

#### Method 3: [16]

This algorithm first applies a counting sequence, and based on the initial results a second sequence is applied. The STV of the second sequence represents the number of 0s in the corresponding STVs of the first sequence. The purpose of the second sequence is to make sure that the overall test set  $S$  is independent. This algorithm can achieve self-diagnosis. No confounding syndromes can be identified. Also, the fault f1 in Figure 6.7 cannot be detected by this algorithm.

#### W-Test Algorithm: [25]

This algorithm is similar to Algorithm A1 except that the first sequence is a counting sequence  $S_C$ , i.e., in the W-Test Algorithm the test set consists of only  $S_C$  and  $S_F$ . The W-Test Algorithm cannot achieve maximal diagnosis as shown by the following example.

Part of a wiring network is shown in Figure 6.9 where the counting sequence  $S_C$  have been applied. For  $i = 1, 2$   $STV_i = SRV_i$  and  $SRV_i$  is unique, thus both  $w_1$  and  $w_2$  will be put into Group 0. This means that the opens f1 and f2 and the short f3 will not be identified since during the application of a walking ones sequence, all nets in Group 0 will be kept at 0.

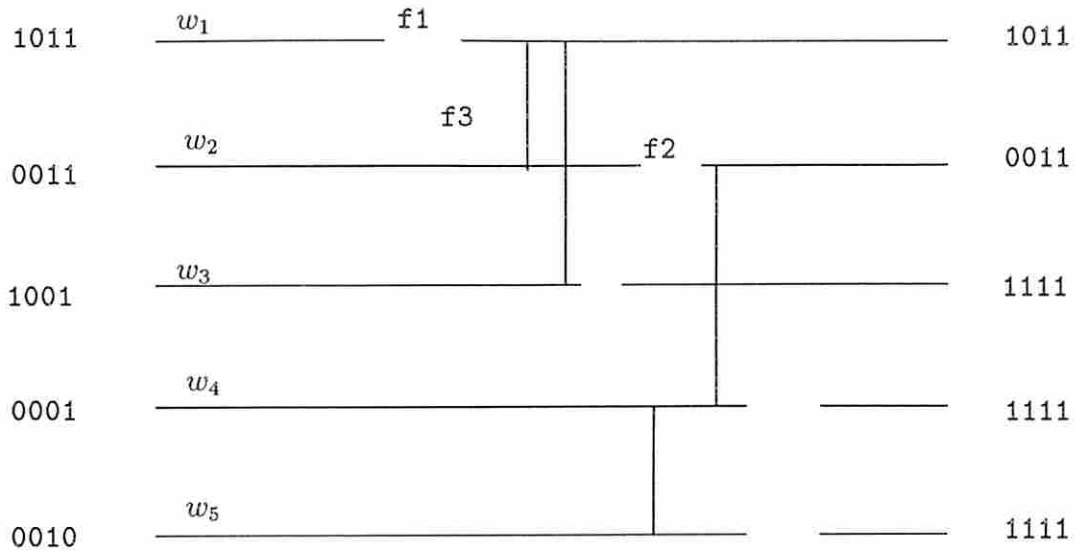


Figure 6.9: A deficiency in the W-Test Algorithm.

Therefore, to avoid putting a net into Group 0 by mistake, it is necessary to apply an independent set which can achieve the *self-diagnosis* property. Both Algorithm A1 and A2 will not put  $w_1$  and  $w_2$  into Group 0. Thus the faults associated with them can be identified by either  $S_F$  or  $(S_G, S_K)$ .

**C-Test Algorithm:** [34]

The C-Test Algorithm first applies a counting sequence, then based on the analysis of the syndromes, one or more PTVs are applied.

Part of a wiring network is shown in Figure 6.10 where a counting sequence has been applied. In the C-Test Algorithm both faults f3 and f4 can be identified immediately. However, since no aliasing or confounding syndromes are related to  $SRV_1 = 1011$ , the fault f1 and f2 cannot be identified.

Using the same example, the diagnosis sequence generated by both Algorithm A1 and A2 will apply a walking ones sequence to  $w_2, w_3, w_4$  since they are in the same group. Thus all faults in the network can be identified.

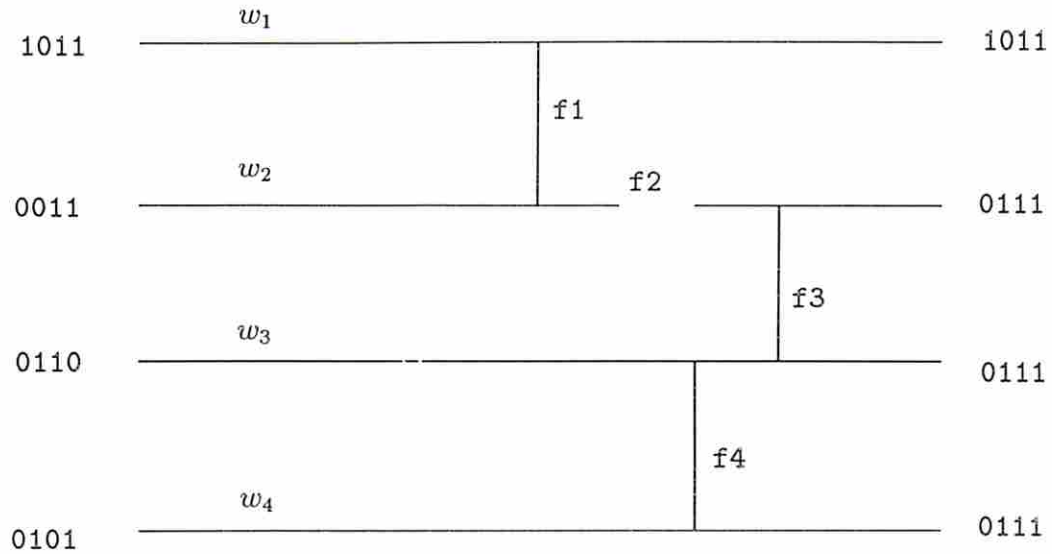


Figure 6.10: A deficiency in the C-Test Algorithm.

## 6.5 Diagnosis Using Structural Information

Up to this point all the test methods presented assume a net can be shorted to every other net in a network. In practice, however, one net can only be shorted to a set of neighboring nets. This set of nets is called its neighbors. The size of the neighboring nets is usually much smaller than the number of nets in the network. Therefore, using neighborhood information it is possible to generate a reduced test set that can still achieve maximal diagnosis. Other researchers have considered using neighborhood information [16, 68], but do not obtain maximal diagnosis because they do not consider that both opens and shorts can be associated with the same net.

A one-step diagnosis algorithm that incorporates the neighborhood information is presented below.

Let  $W = \{w_1, \dots, w_n\}$  be a network under test, and let  $Nbr(w_i) \subset W$  be the set of the neighboring nets of  $w_i$ . The algorithm for constructing a test set that can achieve maximal diagnosis in one-step is as follows.

**Algorithm A3:**



1. Construct a neighborhood graph  $NG = (V_1, E_1)$  as follows: (1) For each  $w_i \in W$  there is a corresponding  $v_i \in V_1$ , (2)  $E_1 = \{e | e = (v_i, v_j), \forall w_j \in Nbr(w_i), \forall w_i\}$ .
2. Construct an augmented neighboring graph  $ANG = (V, E)$  as follows: (1)  $V = V_1$ , (2)  $E = E_1 \cup E_2$ , where  $E_2 = \{e | e = (v_j, v_k), \forall v_j, v_k \in Nbr(w_i), \forall v_i \in V_1\}$ .
3. Label each node  $v_i$  of the  $ANG$  with a color  $Color(v_i)$  such that  $Color(v_i) \neq Color(v_j)$  if  $e = (v_i, v_j) \in E$  and that the number of colors  $c$  is minimal (this number is also referred to as the chromatic number of the graph). Let the colors used be  $C_1, \dots, C_c$ .
4. Associate each color  $C_i$  with a unique  $c$  bit binary vector  $CV_{C_i}, i = 1, \dots, c$ , such that only one bit in each vector is a 1.
5. Associate each net  $w_i \in W$  with a vector  $STV_i$  such that  $STV_i = CV_{Color(v_i)}$ . The test set  $S = (STV_1, \dots, STV_n)^T$ .

The problem to be solved in step 3 is NP-Complete [24]. In general, solving this problem requires an exponential time algorithm. The algorithm listed below, referred to as Algorithm Coloring, can be used to solve this problem. Some efficiency is achieved by pruning the search space. This algorithm makes use of Algorithm *MIS* presented in chapter 5.

**Algorithm Coloring:**

1. Construct a weighted graph  $G_2 = (V_2, E_2, C_2)$  such that (1)  $V_2 = V$ , (2)  $E_2 = \bar{E}$ , i.e.,  $E_2 = \{e | e = (v_i, v_j) \notin E, v_i, v_j \in E\}$ , (3)  $C_2 = \{c_i = 1, \forall i\}$ .
2. Apply Algorithm *MIS* using  $G_2$  as input. Let  $c$  be the size of the solution set. The size of the maximal independent set of  $G_2$  equals the size of the maximal clique of  $G$ . Therefore the chromatic number of  $G$  is  $c$ .
3. For nodes  $v_1, \dots, v_n$ , assign a color for each of them. This process can be done in  $O(nc)$  time since the chromatic number  $c$  is known.

**Theorem 5** *The test set derived from Algorithm A3 achieves maximal diagnosis for  $W$  and is minimal in size.*

**Proof:**

First, it is necessary to show that  $S$  can achieve maximal diagnosis. From the way the test set is generated in Algorithm A3, it is clear that the STVs of a net  $w_i$  and its neighboring nets  $Nbr(w_i)$  contain a walking ones subsequence. All opens faults in this set of nets and all shorts between any two net in this set are diagnosable. The same argument can be applied to each net  $w_i \in W$ . Thus  $S$  achieves maximal diagnosis for the network  $W$ . Second, it is necessary to show that  $S$  is of minimal size. The number of PTVs in  $S$  is  $c$ , which is the minimal number of colors required to color the ANG. For any test set with less PTVs than  $S$ , there exist at least one net  $w_i$  and its neighboring nets  $Nbr(w_i)$  that cannot be assigned a different CV. Hence the faults in this set of nets cannot be fully diagnosed. Thus  $S$  is a minimal test set that can achieve maximal diagnosis for the network  $W$ .  $\square$

**Example 6-4:** Let  $W = \{w_1, \dots, w_6\}$  be a network under test. Let  $Nbr(w_1) = \{w_2\}$ ,  $Nbr(w_2) = \{w_1, w_3\}$ ,  $Nbr(w_3) = \{w_2, w_4\}$ ,  $Nbr(w_4) = \{w_3, w_5\}$ ,  $Nbr(w_5) = \{w_4, w_6\}$  and  $Nbr(w_6) = \{w_5\}$ . The neighborhood graph NG is shown in Figure 6.11(a) and the augmented neighborhood graph ANG is shown in Figure 6.11(b). It is found that the chromatic number of ANG is 3 and one solution to the coloring problem is shown in Figure 6.11(c). The colors used in the graph are R, G and B.

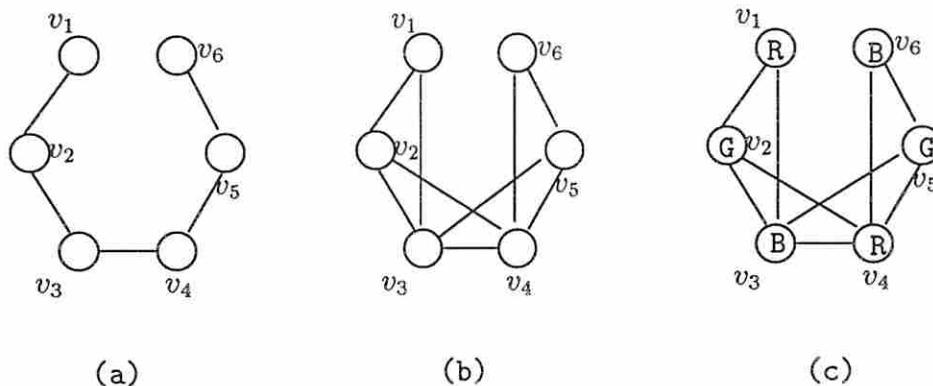


Figure 6.11: Example 6-2: (a) the NG, (b) the ANG, (c) the colored graph.

Let the vectors associated with the colors be  $CV_R=(100)$ ,  $CV_G=(010)$  and  $CV_B=(001)$ . Then the minimal test set  $S$  that can achieve maximal diagnosis is as follows.

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that there is a walking ones sequence for each net  $w_i$  and its neighboring nets  $Nbr(w_i)$ . Therefore, it is clear that  $S$  achieves maximal diagnosis for the network  $W$ .

## Chapter 7

### Interconnect Test Scheduling

Many digital systems will soon be built with ICs that conform with the IEEE 1149.1 boundary scan architecture. Due to the hierarchical nature of such systems, they may contain many boundary scan chains. These chains can be used to test the system, subsystem and board interconnect. To reduce test time, the application of test vectors to these scan chains must be carefully scheduled. This chapter deals with problems related to finding an optimal schedule for testing interconnect. This problem is modeled using a directed graph. The following results are obtained: 1) upper and lower bounds on interconnect test time; 2) necessary and sufficient conditions for obtaining the optimal schedule when the graph is acyclic; 3) sufficient condition for obtaining the optimal schedule when the graph is cyclic; and 4) an algorithm for constructing the optimal schedule for any graph.

#### 7.1 Introduction

Testing interconnect between I/O pins of ICs on a printed circuit board is facilitated by including boundary scan in the design of the ICs. In this chapter the term boundary scan is referred to the IEEE 1149.1 boundary scan architecture [33]. For this architecture a boundary scan cell is associated with each I/O pin. A boundary scan register in an IC is formed by concatenating these cells. The logical values of the input (output) pins of an IC can be observed (set) by using the boundary scan

register. By using the boundary scan registers a net connecting two I/O pins can be tested by scanning in test vectors and observing test results.

The impact of boundary scan on board test has been reported in [28, 50, 54]. This technique provides many benefits such as enhanced diagnosis, reduced test-repair looping, standardized testing, and reuse of tests. These benefits can be extended to the test of other levels of assembly of a system provided that the boundary scan philosophy is followed.

In many cases multiple boundary scan chains exist in a complex system composed of subsystems which, in turn, are composed of modules and boards. Therefore, when testing interconnect between these subassemblies, multiple boundary scan chains (BS-chains) are used. Test vectors can be applied to these BS-chains by using a test controller. To apply a test vector, the test controller must first partition the vector into segments and associate each BS-chain with a vector segment. The test controller then sequentially selects a BS-chain and applies the associated vector segment to it. The order in which chains are selected can significantly impact total test time. According to the IEEE 1149.1 protocol, for each scan operation the test result segment is loaded (in parallel) into the BS-chains immediately before a new test vector segment is shifted in. This result segment can be shifted out while shifting in a new test vector segment. The selection order is important since it determines the correctness of the test procedure and the overall test time.

The generation of tests to detect and diagnose interconnect faults on a board is discussed in [25, 29, 30, 34, 37, 64, 68]. In this chapter it is assumed that a set of test vectors has been generated and stored in a memory unit. Our main objective is to apply these test vectors so that the interconnect is correctly tested in minimal time. This objective can be achieved if the optimal schedule is found. For each test vector the schedule determines the order for applying test vector segments to the BS-chains. This implies that the interconnect is correctly tested by a test controller executing the schedule. If the optimal schedule is obtained, all test vectors can be applied in minimal time.

In this chapter several theorems which aid in identifying how to construct an optimal schedule are presented. It is shown that the optimal schedule can be



achieved if a proper order is followed in applying test vectors to the BS-chains. An algorithm for deriving an optimal schedule is presented. This schedule can be executed by the Module Maintenance Controller (MMC) developed at USC [43], or by the Scan Bus Master (SBM) developed by Texas Instruments [63]. The test time is greatly reduced using an optimal schedule. A reduction in the range of 30 to 50 percent has been achieved for the examples considered so far. From the way the problem is defined, the least upper bound in the reduction in test time is 50 percent.

The algorithms presented can also be applied to schedule tests of chips that are designed with full scan capability where the chips have more than one internal scan chains. Typical examples are chips designed with both the boundary scan architecture and the multiple scan chain technique, such as the CBT method and the MAST method [22, 57].

## 7.2 Testing Model

Every chip has a boundary scan register since it is assumed that all chips under test have the boundary scan architecture. A scan chain is formed by cascading the boundary scan registers of various chips during test mode. The length of the scan chain is the number of scan cells in the boundary scan registers. All cells in a scan chain share the same control inputs (except the Mode line) provided by the test controller. Therefore all scan cells in a scan chain always operate in the same mode. The control model of a boundary scan register is shown in Figure 2.6.

The control inputs of the scan chain are assumed to be activated in a fixed order (see Figure 2.5), namely an activation of *CaptureDR* followed by zero or more activations of *ShiftDR*, and lastly an activation of *UpdateDR*. A test controller must follow this order to access a scan chain. Also, only one scan chain can be accessed at a time. These assumptions are in accordance with the IEEE P1149.1 standard. A special function is defined which, when executed by a test controller generates the proper sequences of control signals. This function is denoted as *Scan(chainID, vecS, resS)*, where *chainID* is the chain selected to be scanned,

$vecS$  is the new data string that will be shifted into the scan cells, and  $resS$  is the data string originating from the scan chain that will be shifted out of it. Once this function is executed by a test controller, the logic values on the  $D$  inputs of the scan chain are collected as a string  $resS$ , and the  $Q$  outputs of the chain are updated to take on the values specified by the data string  $vecS$ .

A test schedule represented by this function can be executed by a Test Channel operating in the DTUR mode [44].

### Graph Model for Testing Interconnect

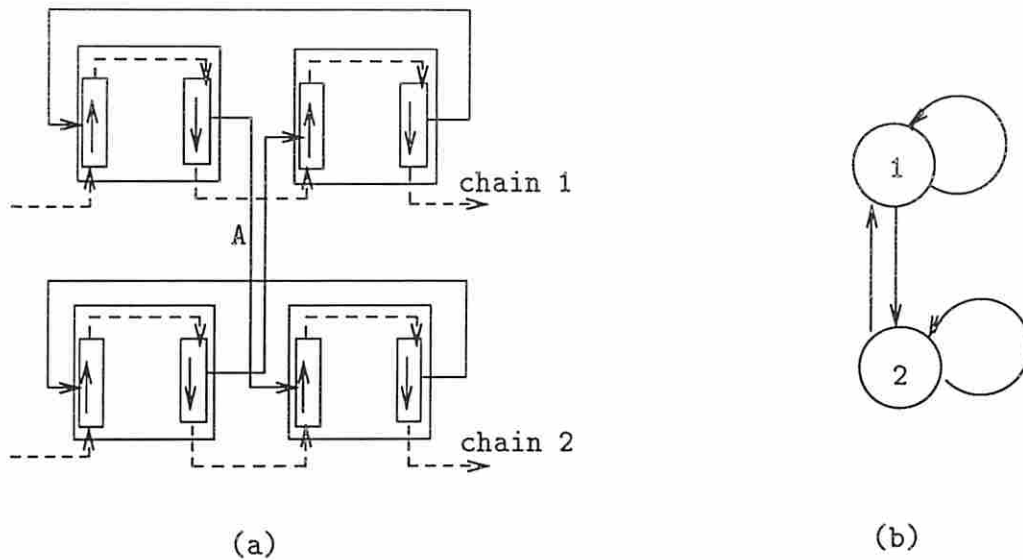


Figure 7.1: Test interconnect via two boundary scan chains; (a) block diagram, (b) graph model.

Figure 7.1(a) shows interconnects that can be tested via two boundary scan chains. Each scan chain is formed by cascading the boundary scan registers of two chips. Figure 7.1(b) shows a directed graph that is used in scheduling tests. Each node represents a scan chain. Each edge represents the data flow direction between scan chains. For example, an edge exist from node 1 to node 2 because an output pin in scan chain 1 drives an input pin in scan chain 2 - namely signal  $A$ .

Each net can be represented by one or more edges in the graph. For example, multi-input nets, multi-output nets and busses may all map into more than one edge in this graph model. When a bus structure is used, many edges may exist in the graph. To avoid dealing with a complete directed graph, the testing of a bus is carried out in several phases. During each phase, the bus is modeled as a single-input multi-output net.

## Test Controller Model

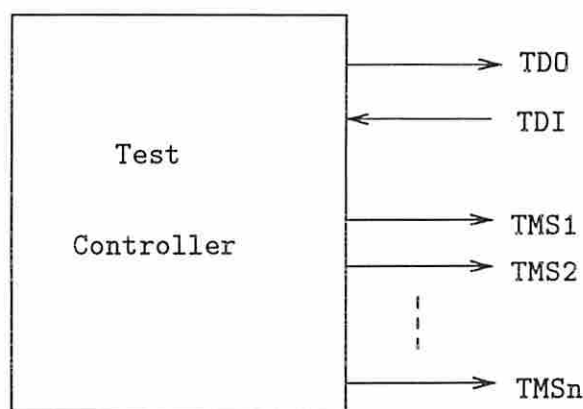


Figure 7.2: The test controller model.

A model for the test controller used in this work is shown in Figure 7.2. The controller contains a data port, which includes a TDI line and a TDO line, and several TMS lines labeled as  $TMS_i$ ,  $i=1, \dots, n$ . Only one TMS line is selected at a time. By controlling the selected TMS line and the data port, the test controller can send and receive information to/from the selected scan chain. The non-selected TMS lines are either (1) set high, which sets the associated scan chain to the “reset” state, or (2) set low, which sets the associated scan chain to the “run test/idle” state. In the latter state, the chips under test in that chain can execute a self-test. The controller can only alter the value of one TMS line at a time. At least two counters are included in the test controller. These counter are used to control the transmission of information between the data port and the selected scan chain. In addition, the controller can execute the *scan* function on a selected scan chain.

## 7.3 The Problem

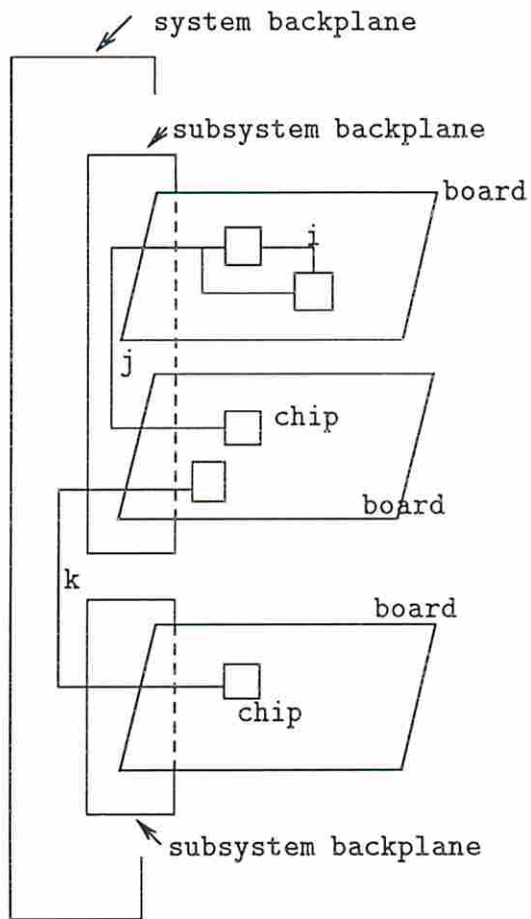
In this section the reasons for having multiple scan chains in testing interconnect are first presented. The scheduling problem associated with the application of tests via these scan chains is then identified. Solution methods for this problem will be presented in the next section.

### 7.3.1 The Use of Multiple Scan Chains

The system test hierarchy for a hierarchically testable and maintainable system has been discussed in [12, 27, 43]. With reference to the system discussed in [12, 43], each system has a system maintenance processor (SMP). Each subsystem, which consists of two or more modules, has a subsystem maintenance processor (SuMP). Each module (board) has a module maintenance controller (MMC). Each chip has an on-chip test controller (CMC), which can be as simple as a Test Access Port (TAP), plus a boundary scan register. Test busses are required to connect these controllers. The issue of checking these test busses is not dealt with in this chapter; only the testing of functional interconnect is considered.

Three classes of interconnect exist, namely system interconnect, subsystem interconnect and board (module) interconnect. A net can have one or more drivers and one or more receivers. These drivers and receivers can be located in one or more chips. If these chips are all located on the same board within a subsystem then this net belongs to the class of board interconnect. If these chips are located on different boards then this net belongs to the class of subsystem interconnect. If these chips are located on different subsystems then this net belongs to the class of system interconnect. Figure 7.3 shows nets belonging to the different classes.

Two control schemes, referred to as distributed control and centralized control, exist in testing a net connecting two or more units, where a unit can be either a subsystem, a board or a chip. The distributed control scheme is shown in Figure 7.4(a), where a net is tested by two local controllers LC1 and LC2 under the control of another controller C1. C1 first instructs LC1 to execute a *Scan* function in order



Board Interconnect: i  
 Subsystem Interconnect: j  
 System Interconnect: k

Figure 7.3: Different classes of interconnect.



to set net  $i$  to a logic value. It then instructs LC2 to read the value on the net via another *Scan* function. This value is then checked by C1. The centralized control scheme is shown in Figure 7.4(b), where the testing of the net is directly controlled by C2. C2 execute a *Scan* function to set net  $i$  to a logic value, and then executes another *Scan* function to read the value of the net. The test bus is configured in a star configuration so that unit 1 can be tested even if unit 2 is removed. In both schemes the net is tested via two scan chains.

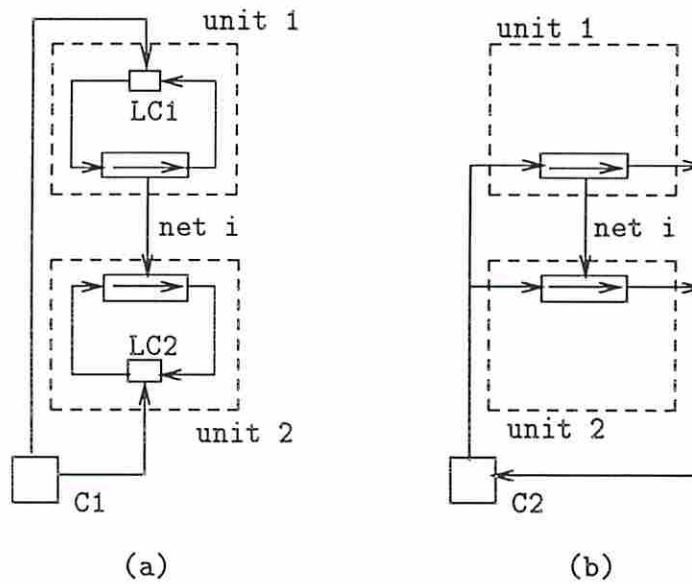


Figure 7.4: Two schemes for testing interconnect; (a) distributed control, (b) centralized control.

Because of the hierarchical nature of a system, it is very common to test interconnect, especially system and subsystem interconnect, using multiple scan chains. If a unit represents a chip, then Figure 7.4 represents a board having several chips that support boundary scan. In Figure 7.4(a) each chip has a fairly complex test controller. In Figure 7.4(b) each chip has a minimal test controller. It is believed that most designs will conform to the latter case. Several scan chains can be used to reduce test time. Chips associated with the same scan chain can be tested either in a sequential manner or concurrently. However, in the latter case the test procedure can be quite complex and inefficient. On the other hand, chips associated with different scan chains can more easily be tested concurrently and usually in an efficient manner. Therefore, it can be concluded that, in many

cases, interconnect is tested using multiple scan chains regardless of the class of interconnect.

### 7.3.2 Scheduling Problem in Testing Interconnects

The problem related to scheduling vectors to test interconnect is considered next. This problem is illustrated using both distributed and centralized control schemes. We assume that the test vectors have been formatted to conform with multiple scan chains.

#### Centralized Control Scheme

The scheduling problem is modeled using a directed graph. Figure 7.5 shows several directed graphs and their associated test schedules. Each test schedule consists of several *Scan* functions, where  $t_{i,j}$  ( $r_{i,j}$ ) represents the  $i$ th test vector (result) segment for scan chain  $j$ . For clarity and simplicity it is assumed that each schedule consists of only two test vectors.

In simple cases such as those shown in Figure 7.5 (a), (b) and (c) the schedule is easy to construct. For example, in Figure 7.5(b) a test vector is only applied to scan chain 1, while a result vector consists of two segments collected from scan chains 1 and 2. The first function  $Scan(1, t_{1,1}, -)$  loads chain 1 with the first test vector. The second function  $Scan(2, -, r_{1,2})$  gets the result segment associated with the first test vector out of scan chain 2. The third function  $Scan(1, t_{2,1}, r_{1,1})$  gets the result segment associated with the first test vector out of scan chain 1 and simultaneously loads the second test vector into scan chain 1. The fourth function  $Scan(2, -, r_{2,2})$  gets the result segment associated with the second test vector out of scan chain 2. Finally, a fifth function  $Scan(1, -, r_{2,1})$  is used to scan out the result segment associated with the second test vector in scan chain 1.

The situation is more complex for the cases shown in Figure 7.5 (d), (e) and (f), where it is more difficult to construct a test schedule. In general, if the graph contains many nodes with complex connectivity, finding a “good” schedule is difficult; the problem of finding an optimal schedule is NP-Complete.

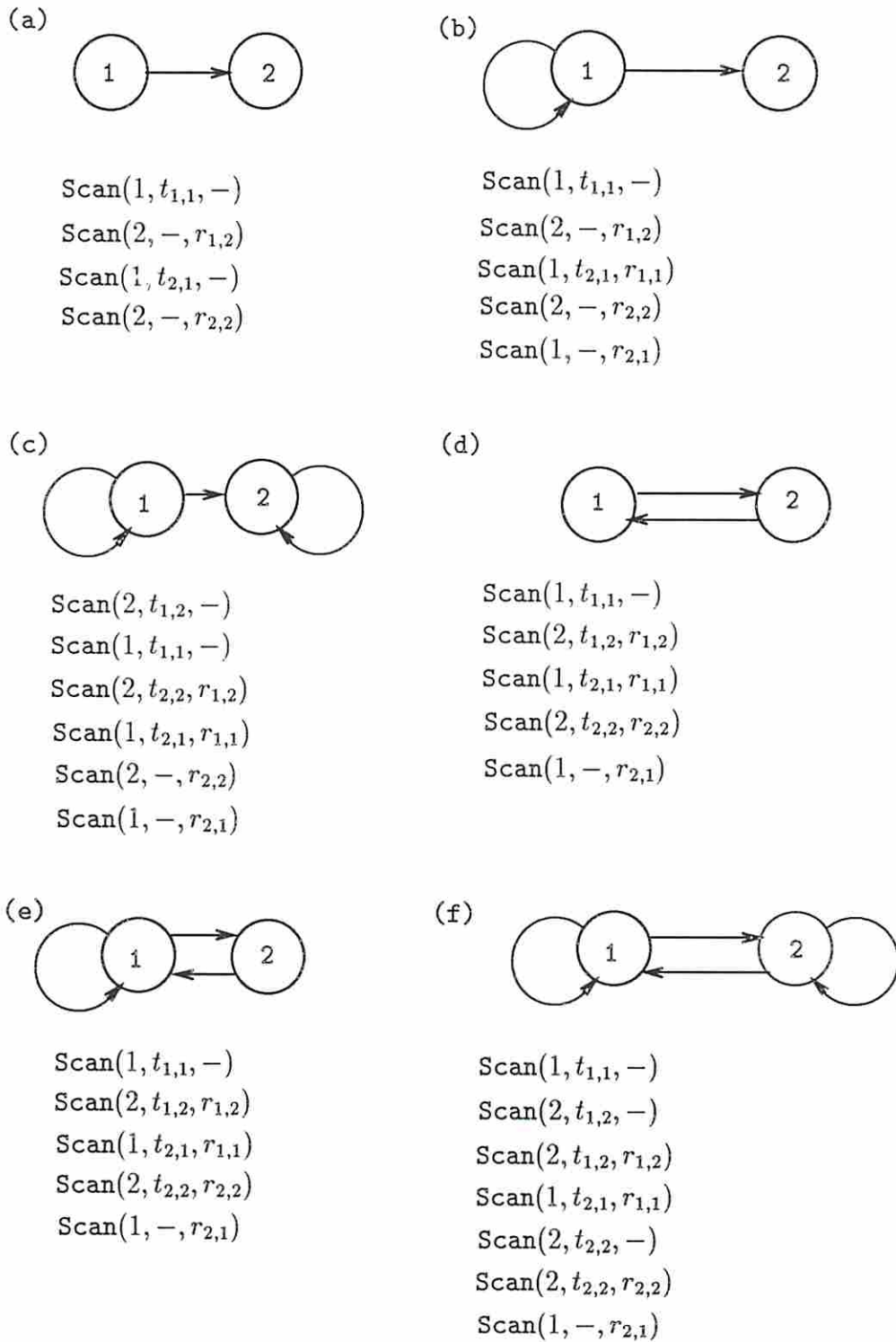


Figure 7.5: Deriving test schedules for several examples.

### Distributed Control Scheme

In Figure 7.4(a) the execution of *Scan* functions by LC1 and LC2 can be either synchronous or asynchronous. In the former case, the scheduling problem does not exist. The interconnect can be tested by letting LC1 and LC2 execute the *Scan* functions at the same time. In the latter case, LC1 and LC2 must execute the *Scan* functions in sequence. The order in which LC1 and LC2 execute this function is important since it dictates the correctness of the test and the overall interconnect test time.

In general it can be difficult to synchronize LC1 and LC2 because 1) C1 cannot send data to both LC1 and LC2 at the same; 2) clock skew between LC1 and LC2; and 3) difference in the length of the scan chain between LC1 and LC2. Therefore, the scheduling problem exists in this distributed control scheme.

Since the scheduling problems for both control schemes can be modeled in the same way, only one of them will be considered. For the rest of this chapter, the model under discussion is assumed to be the testing of board interconnect using the centralized control scheme shown in Figure 7.4(b). The theorems and procedures that lead to the generation of an optimal schedule are presented in the next section.

## 7.4 Optimal Test Scheduling Theorems

Throughout this section we shall use the following notation. Let  $B$  be a board containing  $n$  BS-chains,  $t_1, \dots, t_m$  be the test vectors for testing the interconnect on  $B$ , and  $r_1, \dots, r_m$  are the test results associated with  $t_1, \dots, t_m$ , respectively. Each test vector  $t_i$  is partitioned into  $n$  segments  $t_{i1}, \dots, t_{in}$  in accordance with the  $n$  BS-chains. Each vector segment is then applied to its associated BS-chain. The result  $r_i$  consists of segments  $r_{i1}, \dots, r_{in}$  collected from the  $n$  BS-chains. A BS-chain  $j$  is **scanned** if  $t_{ij}$  ( $r_{ij}$ ) is to be applied (observed). A vector (result) segment is said to be **required** if it is in the test (result) vector and it has not yet been applied (observed).



**Definition 1** A schedule for testing interconnects on a board  $B$ , denoted by  $\mathcal{S}(B)$ , is a sequence of Scan functions that can apply all test vectors  $t_1, \dots, t_m$  and collect all test results  $r_1, \dots, r_m$ .

The total number of shift operations of a schedule  $\mathcal{S}(B)$  is denoted by  $N_{\mathcal{S}}(B)$ . This number is a good measure of the test time since most operations of a Scan are shift operations. This notation is used throughout the chapter. Also the argument  $B$  will not be used if the meaning is clear.

**Definition 2** A schedule  $\mathcal{S}$  is optimal if  $N_{\mathcal{S}}$  is minimal, i.e. given a schedule  $\mathcal{S}_1$  for  $B$ ,  $N_{\mathcal{S}} \leq N_{\mathcal{S}_1}$ .

**Definition 3** For a pair of BS-chains  $v_i, v_j$ , a schedule  $\mathcal{S}$  has the scan order  $v_i \succ v_j$  if for each test vector  $t_k$  ( $k = 1, \dots, m$ ) that contains vector segments  $t_{ki}$  and  $t_{kj}$  for  $v_i$  and  $v_j$ , respectively,  $\mathcal{S}$  applies  $t_{ki}$  to  $v_i$  before applying  $t_{kj}$  to  $v_j$ .

**Definition 4** Let  $v_i$  and  $v_j$  be two BS-chains on a board  $B$ . If there is a net connecting  $v_i$  and  $v_j$  whose logic value can be set by a scan cell in  $v_j$  and observed by a scan cell in  $v_i$ , then  $v_i$  depends on  $v_j$ , denoted by  $v_i \mathbf{D}v_j$ , otherwise,  $v_i$  does not depend on  $v_j$ , denoted by  $v_i \overline{\mathbf{D}}v_j$ .

**Definition 5** Let  $X$  and  $Y$  be two sets.  $X + Y$  is the union of these two sets,  $X - Y$  is the difference of these two sets and consists of all elements in  $X$  which are not in  $Y$ .

**Definition 6** A dependency graph  $DG$  for a board  $B$  is defined as a 3-tuple  $(V, W, E)$ , where  $V$  is a set of nodes,  $W$  is a set of labels associated with the nodes, and  $E$  is a set of edges, such that (1) for each BS-chain  $c$  in  $B$ , there is a corresponding node  $v_c \in V$ , (2) for each BS-chain  $c$  in  $B$ , there is a  $w_c \in W$  representing the length of the BS-chain  $c$ , and (3) for every pair of BS-chains  $u, v$ , there is a corresponding edge  $e = (u, v) \in E$  if and only if  $v \mathbf{D}u$ .

Two types of nodes exist in a  $DG$ , namely type-I and type-II nodes. Type-I nodes refer to those nodes that have a self-loop; Type-II nodes refer to those nodes



that don't have a self-loop. The set of all type-I nodes is denoted as  $V_I$ . The set of all type-II nodes is denoted as  $V_{II}$ . Thus  $V = V_I + V_{II}$ .

**Definition 7**  $DG' = (V', E', W')$  is the **reduced form** of  $DG = (V, E, W)$  with respect to  $U$ , denoted as  $DG' = DG \perp U$ , where  $V' = V - U$ ,  $W' = W - \{w_i | v_i \in U\}$  and  $E' = E - \{e | e = (v_i, v_j), (v_j, v_i) \text{ or } (v_i, v_k), \text{ where } v_i, v_k \in U, v_j \in V'\}$ .

Let  $V_I$  ( $V_{II}$ ) be the set of all type-I (type-II) nodes in  $DG$ ,  $DG_I = DG \perp V_{II}$  and  $DG_{II} = DG \perp V_I$ .

**Definition 8**  $DG$  is **type-I acyclic** if  $DG_I$  is acyclic when all self-loops are ignored.  $DG$  is **type-II acyclic** if  $DG_{II}$  is acyclic.  $DG$  is **type-acyclic** if  $DG$  is both type-I acyclic and type-II acyclic.

**Definition 9** Let  $DG$  be type-acyclic. A schedule  $\mathcal{S}$  is **type-I proper** if for every pair of nodes  $v_i, v_j \in DG_I$ , 1)  $v_j \bar{D}v_i$ , and 2)  $\mathcal{S}$  has the scan order  $v_i \succ v_j$  (see Figure 7.5(a)).

A schedule  $\mathcal{S}$  is **type-II proper** if for every pair of nodes  $v_i, v_j \in DG_{II}$ , 1)  $v_i \bar{D}v_j$ , and 2)  $\mathcal{S}$  has the scan order  $v_i \succ v_j$  (see Figure 7.5(c)).

A schedule  $\mathcal{S}$  is **proper** if 1) it is both type-I proper and type-II proper and 2) for every pair of nodes  $v_i \in V_I, v_j \in V_{II}$ ,  $\mathcal{S}$  has the scan order  $v_i \succ v_j$  (see Figure 7.5(b)).

Note that if  $DG$  is not type-acyclic, it is impossible to find a schedule that is proper. For example, the schedules in Figure 7.5(d) and (f) are not proper.

Let  $DG$  be type-acyclic,  $K$  be the cardinality of  $V_I$ , and  $M$  be the cardinality of  $V_{II}$ . Let  $T_1 = (v_1, v_2, \dots, v_K)$  be a topological order for nodes in  $DG_I$  when self-loops are ignored and  $T_2 = (u_1, u_2, \dots, u_M)$  be a topological order for nodes in  $DG_{II}$ .

**Theorem 6** Let  $DG$  be type-acyclic. A schedule  $\mathcal{S}$  is optimal iff  $\mathcal{S}$  is proper.

**Proof:** (*if* part)

Let  $\mathcal{S}$  be a proper schedule. Suppose that  $\mathcal{S}$  is not optimal. There exists at least one node  $v_i \in V$  such that either 1)  $v_i \in V_I$  and the test vector segment for  $v_i$  is applied twice for each test vector, or 2)  $v_i \in V_{II}$  and there is at least one scan operation that does not contain both the required test vector segment and the required result segment. In case 1, there exists at least one node  $v_j \in V_I$  such that  $v_j \mathbf{D} v_i$  and  $\mathcal{S}$  has the scan order  $v_i \succ v_j$ . This implies that  $\mathcal{S}$  is not type-I proper. In case 2, there exists at least one node  $v_j \in V_{II}$  such that  $v_j \mathbf{D} v_i$  and  $\mathcal{S}$  has the scan order  $v_j \succ v_i$ . This implies that  $\mathcal{S}$  is not type-II proper. Both cases lead to the conclusion that  $\mathcal{S}$  is not a proper schedule. This contradicts the fact that  $\mathcal{S}$  is a proper schedule. This proves the *if* part of the theorem.

(*only if* part)

Let  $\mathcal{S}$  be an optimal schedule. Suppose that  $\mathcal{S}$  is not a proper schedule. If  $\mathcal{S}$  is not type-I proper, then there exists  $v_i, v_j \in V_I$  such that 1)  $v_j \mathbf{D} v_i$ , and 2)  $\mathcal{S}$  has the scan order  $v_i \succ v_j$ . This means that  $v_i$  must be scanned twice for each test vector, while all other  $v_j$  need only be scanned once. It is possible to find another schedule  $\mathcal{S}_1$  that has the same scan order as  $\mathcal{S}$  except that  $\mathcal{S}_1$  has the scan order  $v_j \succ v_i$ . This implies that  $N_{\mathcal{S}_1} < N_{\mathcal{S}}$ , thus  $\mathcal{S}$  is not optimal. This leads to a contradiction and thus proves the *only if* part of Theorem 6.  $\square$

**Corollary 1** *Let  $DG$  be type-acyclic. If  $\mathcal{S}$  has the scan order  $(v_K \succ \dots \succ v_2 \succ v_1 \succ u_1 \succ u_2 \succ \dots \succ u_M)$ , then  $\mathcal{S}$  is optimal.*

**Proof:**

From Definition 9 it can be concluded that  $\mathcal{S}$  is proper. By Theorem 6  $\mathcal{S}$  is an optimal schedule.  $\square$

Procedure P1, which is based on Corollary 1, constructs an optimal schedule  $\mathcal{S}$  for a type-acyclic  $DG$ . In this procedure,  $T_1 = (v_1, v_2, \dots, v_K)$  is a topological order for nodes in  $DG_I$  when self-loops are ignored, and  $T_2 = (u_1, u_2, \dots, u_M)$  is a topological order for nodes in  $DG_{II}$ .

**Procedure P1:**

- (1) For  $i = 1$  to  $m$  do  
 (1.1) For  $j = K$  down to 1 do  $Scan(v_j, t_{i,j}, r_{i-1,j})$ .  
 (1.2) For  $j = 1$  to  $M$  do  $Scan(u_j, t_{i,j}, r_{i,j})$ .  
 (2) For  $j = K$  down to 1 do  $Scan(v_j, x, r_{m,j})$ . □

Since the function  $Scan(v_i, t_j, r_j)$  contains  $w_i$  shift operations, from Procedure P1 it is obvious that

$$\begin{aligned} N_S &= (m+1) * \sum_{v_i \in V_I} w_i + m * \sum_{v_i \in V_{II}} w_i \\ &= \sum_{v_i \in V_I} w_i + m * \sum_{v_i \in V} w_i \end{aligned} \quad (7.1)$$

$$\equiv N_{ac} \quad (7.2)$$

**Lemma 7 [Lower Bound]** Let  $\mathcal{S}$  be a schedule for  $DG$ .  $N_S \geq LB \equiv m * \sum_{v_i \in V} w_i$ .

**Proof:**

From Equation (1) it is obvious that  $N_S = m * \sum_{v_i \in V} w_i$  when  $V = V_{II}$  and  $DG$  is acyclic. For this case a schedule can be found such that each scan chain is scanned exactly once for each test vector. It thus follows that a schedule  $\mathcal{S}$  for any other  $DG$  has a greater or equal number of scan operations. □

If a given  $DG$  is not type-acyclic, then Theorem 6 cannot be applied. This type of  $DG$  is dealt with next.

**Definition 10** A **type-I cycle** in  $DG_I$  is a directed cycle that consists of two or more type-I nodes. A **type-II cycle** in  $DG_{II}$  is a directed cycle that consists of two or more type-II nodes.

A cycle  $C$  in  $DG_I$  ( $DG_{II}$ ) is **broken** if a node  $v \in C$  is removed from  $DG_I$  ( $DG_{II}$ ). Removing  $v$  from  $DG_I$  is represented as  $DG_I \perp \{v\}$  (Definition 7). In a schedule  $\mathcal{S}$ , the removal of a node  $v_i$  from  $DG_I$  is achieved by scanning the vector segment into chain  $i$  twice for each test vector. The first scan operation loads a vector segment into the scan chain. The second one gets the result segment out

of the scan chain while loading the same vector segment back into it. Thus node  $v_i$  provides a correct test vector segment throughout the application of the rest of vector segments. Hence it can be removed from  $DG_I$ .

The removal of a node  $v_i$  from  $DG_{II}$  is achieved by putting  $v_i$  into the same category as type-I nodes when applying the vector segment for  $v_i$ . This enables the vector segment in scan chain  $i$  to remain valid throughout the application of the remaining segments of the current test vector. Thus node  $v_i$  can be treated the same as a type-I node. Let  $DG_I = DG \perp V_{II}$  and  $DG_{II} = DG \perp V_I$  be two subgraphs of  $DG$ .

**Definition 11** *If  $DG = (V, W, E)$  is cyclic and  $DG' = DG \perp Z$  is acyclic when self-loops are ignored, then  $Z$  is called a feedback vertex set (FVS) of the  $DG$ .*

For example, in Figure 7.5 (d),  $Z = \{2\}$  is a FVS of the  $DG$ .

**Definition 12** *Given a  $DG=(V, W, E)$ , a set of nodes  $Z \subset V$  is a minimal FVS of  $DG$  if for any  $Z' \subset V$  that is a FVS of  $DG$ ,  $\sum_{v_i \in Z} w_i \leq \sum_{v_i \in Z'} w_i$ .*

**Definition 13** *Let  $Z_I \subset V_I$ ,  $Z_{II} \subset V_{II}$  be two sets of nodes in  $DG$ . If (1)  $DG_I$  is cyclic and  $DG_I \perp Z_I$  is acyclic when self-loops are ignored, and (2)  $DG_{II}$  is cyclic and both  $DG_{II} \perp Z_{II}$  and  $DG \perp (V - (Z_{II} + V_I - Z_I))$  are acyclic, then  $(Z_I, Z_{II})$  is called a joint-FVS of  $DG$ .*

For example, in Figure 7.6(a),  $Z_I = \{6\}$  and  $Z_{II} = \{1\}$  is a joint-FVS of the  $DG$ .

**Definition 14** *Given a  $DG$ ,  $(Z_I, Z_{II})$  is called a minimal joint-FVS of  $DG$  if (1)  $(Z_I, Z_{II})$  is a joint-FVS of  $DG$ , and (2) for any  $(Z'_I, Z'_{II})$  that is a joint-FVS of  $DG$ ,  $(m - 1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i \leq (m - 1) * \sum_{v_i \in Z'_I} w_i + \sum_{v_i \in Z'_{II}} w_i$ .*

Let  $(Z_I, Z_{II})$  be a joint-FVS of  $DG$ . Let  $T_1 = (u_1, u_2, \dots, u_K)$  be a topological order for nodes in  $DG \perp (V - (Z_{II} + V_I - Z_I))$  when self-loops are ignored,  $T_2 = (v_1, v_2, \dots, v_{z_2})$  be a topological order for nodes in  $DG \perp (V - (V_{II} - Z_{II}))$ , and  $T_3 = (y_1, y_2, \dots, y_{z_1})$  be an arbitrary order for nodes in  $DG \perp (V - Z_I)$ . Also



let  $K$ ,  $M$ ,  $z_1$  be the cardinality of  $Z_{II} + V_I - Z_I$ ,  $V_{II} - Z_{II}$ , and  $Z_I$ , respectively. A schedule  $\mathcal{S}$  for  $DG$  can be constructed as follows.

**Procedure P2[B]:**

- (1) For  $i = 1$  to  $m$  do
  - (1.1) For  $j = K$  down to 1 do  $Scan(u_j, t_{i,u_j}, r_{i-1,u_j})$ .
  - (1.2) For  $j = 1$  to  $z_1$  do  $Scan(y_j, t_{i,y_j}, x)$ .
  - (1.3) For  $j = 1$  to  $M$  do  $Scan(v_j, t_{i,v_j}, r_{i,v_j})$ .
  - (1.4) For  $j = 1$  to  $z_1$  do  $Scan(y_j, t_{i,y_j}, r_{i,y_j})$ .
- (2) For  $j = K$  down to 1 do  $Scan(u_j, x, r_{m,u_j})$ . □

Since the function  $Scan(v_i, t_j, r_j)$  contains  $w_i$  shift operations, it follows that

$$\begin{aligned}
 N_{\mathcal{S}} &= (m+1) * \sum_{v_i \in (Z_{II} + V_I - Z_I)} w_i + 2m * \sum_{v_i \in Z_I} w_i + m * \sum_{v_i \in (V_{II} - Z_{II})} w_i \\
 &= (m-1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i + \sum_{v_i \in V_I} w_i + m * \sum_{v_i \in V} w_i \\
 &= (m-1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i + N_{ac} \tag{7.3}
 \end{aligned}$$

Note that  $N_{ac} = \sum_{v_i \in V_I} w_i + m * \sum_{v_i \in V} w_i$  is independent of the selection of  $(Z_I, Z_{II})$ . If  $DG$  is type-acyclic then  $Z_I = Z_{II} = \emptyset$ . In this case Equation (3) reduces to Equation (1).

**Theorem 7** *Let  $(Z_I, Z_{II})$  be a minimal joint-FVS of  $DG$ . If  $\mathcal{S}$  is a schedule constructed by Procedure P2, then  $\mathcal{S}$  is optimal.*

**Proof:**

The schedule  $\mathcal{S}$  applies (collects) all required test vector (result) segments, so it is indeed a schedule. We still needed to show that  $\mathcal{S}$  is optimal. It is obvious that  $DG' = DG \perp (Z_I + Z_{II})$  is of type-acyclic. Since by construction  $\mathcal{S}$  is proper with respect to  $DG'$ ,  $\mathcal{S}$  is optimal with respect to  $DG'$  (Theorem 6). Next we show that  $\mathcal{S}$  is optimal with respect to  $DG$ . From Definition 14 it is clear that for any schedule  $\mathcal{S}_1$



that removes another joint-FVS of  $DG (Z'_I, Z'_{II})$ ,  $(m-1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i \leq (m-1) * \sum_{v_i \in Z'_I} w_i + \sum_{v_i \in Z'_{II}} w_i$ . Thus  $N_S \leq N_{S_1}$ . From Definition 2, we conclude that  $\mathcal{S}$  is an optimal schedule.  $\square$

**Corollary 2 [Upper Bound]** *If  $\mathcal{S}$  is optimal then  $N_S \leq UB \equiv 2m * \sum_{v_i \in V} w_i$ .*

**Proof:**

From Equation (3), it is obvious that  $N_S$  is maximal when  $Z_I = V$ , i.e., all nodes in  $DG$  are type-I and they must all be removed. But  $(m+1) * \sum_{v_i \in (Z_{II} + V_I - Z_I)} w_i + 2m * \sum_{v_i \in Z_I} w_i + m * \sum_{v_i \in (V_{II} - Z_{II})} w_i \leq 2m * \sum_{v_i \in V} w_i$  is true for any  $Z_I$  and  $Z_{II}$ . Thus  $N_S \leq 2m * \sum_{v_i \in V} w_i$ .  $\square$

Note that one can always construct a schedule for any  $DG$  regardless of its connectivity. For example, if  $Z_I = V$  then the scan order is not important. The problem is that the schedule constructed may not be optimal.

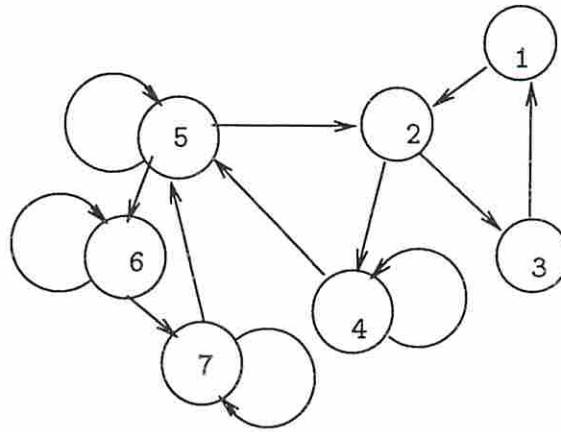
Theorem 7 provides a way to find an optimal schedule for a board modeled as a cyclic DG. Since an acyclic DG can be viewed as a cyclic DG with an empty  $FVS$ , Theorem 7 is applicable to acyclic DGs as well. In this case, both Procedure P1 and P2 produce the same schedule.

**Example:** Optimal schedule for a cyclic  $DG$ .

Figure 7.6(a) shows a cyclic  $DG$  with 7 nodes. There are 4 type-I nodes, i.e.,  $V_I = \{4, 5, 6, 7\}$  and 3 type-II nodes, i.e.,  $V_{II} = \{1, 2, 3\}$ . The minimal joint-FVS  $(Z_I, Z_{II})$  found is  $Z_I = \{6\}$  and  $Z_{II} = \{1\}$ . Using Procedure P2, an optimal schedule can be constructed (see Figure 7.6(c)).  $N_S$  equals  $(2-1) * 90 + 50 + (80 + 90 + 100 + 110) + 2 * (560) = 1640$ .  $\square$

## 7.5 An Algorithm for Generating Schedules

A Test Scheduling Algorithm (TSA) based on Theorem 7 is described next. This algorithm can find an optimal schedule for an arbitrary  $DG$ . Because of its complexity ( $O(n * 2^n)$ ), it may not be suitable for problems where  $n \geq 15$ . For large problems the user can direct the TSA to find a sub-optimal schedule at a reduced



(a)

$m = 2, n = 7$   
 $w_1 = 50, w_2 = 60, w_3 = 70, w_4 = 80$   
 $w_5 = 100, w_6 = 90, w_7 = 110$   
 $V_I = \{4, 5, 6, 7\}$   
 $V_{II} = \{1, 2, 3\}$   
 $Z_I = \{6\}$   
 $Z_{II} = \{1\}$   
 $V_I - Z_I = \{4, 5, 7\}$   
 $V_{II} - Z_{II} = \{2, 3\}$   
 $T_1 = (1 \succ 4 \succ 7 \succ 5)$   
 $T_2 = (2 \succ 3)$

(b)

Test Schedule S

$Scan(5, t_{1,5}, -)$   
 $Scan(7, t_{1,7}, -)$   
 $Scan(4, t_{1,4}, -)$   
 $Scan(1, t_{1,1}, -)$   
 $Scan(6, t_{1,6}, -)$   
 $Scan(2, t_{1,2}, r_{1,2})$   
 $Scan(3, t_{1,3}, r_{1,3})$   
 $Scan(6, t_{1,6}, r_{1,6})$   
 $Scan(5, t_{2,5}, r_{1,5})$   
 $Scan(7, t_{2,7}, r_{1,7})$   
 $Scan(4, t_{2,4}, r_{1,4})$   
 $Scan(1, t_{2,1}, r_{1,1})$   
 $Scan(6, t_{2,6}, -)$   
 $Scan(2, t_{2,2}, r_{2,2})$   
 $Scan(3, t_{2,3}, r_{2,3})$   
 $Scan(6, t_{2,6}, r_{2,6})$   
 $Scan(5, -, r_{2,5})$   
 $Scan(7, -, r_{2,7})$   
 $Scan(4, -, r_{2,4})$   
 $Scan(1, -, r_{2,1})$

(c)

Figure 7.6: Example: Deriving an optimal schedule.

complexity ( $O(n * MAX(n, e))$ ), thereby reducing computation time. This can be done by replacing Procedure Find-Min-Joint-FVS in step (3) by Procedure Find-Joint-FVS.

**Algorithm TSA:**

*Input* : A  $DG = (V, W, E)$ ,  $V_I, V_{II}$  and a test set  $t_1, \dots, t_m$ .

*Output* : A schedule  $\mathcal{S}$ .

*Method* :

(1) For  $i=1$  to  $m$  do the following

Partition  $t_i$  into  $t_{ik}$  ( $k = 1, \dots, n$ ) as follows.

Let the first  $w_1$  bits of  $t_i$  be  $t_{i1}$ ,

let the next  $w_2$  bits of  $t_i$  be  $t_{i2}$ ,

...

let the next  $w_j$  bits of  $t_i$  be  $t_{ij}$ ,

...

let the last  $w_n$  bits of  $t_i$  be  $t_{in}$ .

(2)  $DG_I = DG \perp V_{II}$ ,  $DG_{II} = DG \perp V_I$ .

(3) Run Procedure Find-Min-Joint-FVS (or Procedure Find-Joint-FVS).

Let  $(Z_I, Z_{II})$  be the derived output.

(4) Let  $DG' = DG \perp (V - (Z_{II} + V_I - Z_I))$ .

(5) Find the topological order  $T_1$  of  $DG'$ .

(6) Let  $DG'' = DG \perp (V - (V_{II} - Z_{II}))$ .

(7) Find the topological order  $T_2$  of  $DG''$ .

(8) Generate a schedule  $\mathcal{S}$  using Procedure P2. □

A problem is modeled as a DG in TSA. Test vectors are partitioned into segments in step (1). The procedure Find-Min-Joint-FVS is used to find a minimal joint-FVS  $(Z_I, Z_{II})$  for  $DG$  in step (3). This procedure is described in more detail later on. In step (4) the  $DG'$ , which is acyclic, is derived. The topological order of  $DG'$  is found in step (5). The method listed in [67] is used to derive the topological order. In step (6) the  $DG'' = DG \perp (V - (V_{II} - Z_{II}))$ , which is acyclic, is derived. The topological order of  $DG''$  is derived in step (7). Finally, Procedure P2 is used to

derive a schedule for  $DG$ . According to Theorem 7 the derived schedule is optimal if the  $(Z_I, Z_{II})$  found in step (3) is a minimal joint-FVS of  $DG$ .

Finding  $Z_I, Z_{II}$  is still a problem. For clarity, the problem is restated as follows.

**Problem:** Find a minimal joint-FVS.

Given a cyclic graph  $DG = (V, E, W)$ , find two sets of nodes  $Z_I \subset V_I$ ,  $Z_{II} \subset V_{II}$  such that 1)  $DG' = (DG \perp V_{II}) \perp Z_I$  is acyclic, 2)  $DG'' = (DG \perp V_I) \perp Z_{II}$  is acyclic, 3)  $DG''' = DG \perp (V - (V_I - Z_I + Z_{II}))$  is acyclic, and 4)  $(m - 1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i$  is minimal.

This problem can be shown to be NP-Complete. Due to the facts that 1) the problem of finding a minimal FVS is a special case of the problem of finding a minimal joint-FVS, and 2) the problem of finding a minimal FVS is NP-Complete, it can be concluded that the problem of finding a minimal joint-FVS is NP-Complete by using the **restriction** technique described in [24]. For a small problem, it is possible to compute the optimal solution exhaustively. The following procedure finds a minimal joint-FVS for a given  $DG$ .

**Procedure Find-Min-Joint-FVS:**

*Input:*  $DG = (V, W, E)$ ,  $V_I$  and  $V_{II}$ .

*Output:* A minimal joint-FVS  $(Z_I, Z_{II})$ .

*Method:*

(1) Let  $n_1$  ( $n_2$ ) be the cardinality of  $V_I$  ( $V_{II}$ ).

Let  $Z' = Z'' = \emptyset$ ,  $min =$  a large number.

(2) For  $i = 0$  to  $n_1$  do

(2.1) For all  $C_i^{n_1}$  combinations do

(2.1.1) Generate the next combination ( $U_1$ ).

(2.1.2) If  $DG_I \perp U_1$  cyclic then goto (2.1.1)

(2.1.3) else for  $j = 0$  to  $n_2$  do

(2.1.3.1) For all  $C_j^{n_2}$  combinations do

(2.1.3.1.1) Generate the next combination ( $U_2$ ).



(2.1.3.1.2) If  $DG_{II} \perp U_2$  cyclic then goto (2.1.3.1.1)

(2.1.3.1.3) else if  $DG \perp (V - (V_I - Z_I + Z_{II}))$  cyclic  
then goto (2.1.3.1.1)

(2.1.3.1.4) else if  $min < (m - 1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i$   
then goto (2.1.3.1.1)

(2.1.3.1.5) else  $min = (m - 1) * \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i$ ,  
 $Z' = U_1, Z'' = U_2$ .

(3)  $Z_I = Z', Z_{II} = Z''$ . □

The complexity of Procedure Find-Min-Joint-FVS is derived next. In the worst case  $n_1 + n_2$  operations are needed in step (2.1.3.1.4), which can be repeated  $2^{n_2}$  times in one pass of step (2.1.3). Also, step (2.1.3) can be repeated  $2^{n_1}$  times in the worst case. So the complexity of Procedure Find-Min-Joint-FVS is  $O(2^{n_1+n_2}(n_1 + n_2))$  or  $O(n2^n)$ . Since the most time consuming step in Algorithm TSA is Procedure Find-Min-Joint-FVS, the complexity of Algorithm TSA is  $O(n2^n)$ .

For a large problem it is not computationally feasible to use Procedure Find-Min-Joint-FVS. Therefore, a heuristic procedure that can find a good solution (not necessarily optimal) in a reasonable amount of time is needed. A version of Procedure Find-Min-Joint-FVS that can be used in such cases is described next.

#### **Procedure Find-Joint-FVS:**

*Input:*  $DG = (V, W, E)$ ,  $V_I$  and  $V_{II}$ .

*Output:* A joint-FVS  $(Z_I, Z_{II})$ .

*Method:*

(1) Let  $n_1$  ( $n_2$ ) be the number of nodes in  $V_I$  ( $V_{II}$ ).

(2) Let  $Z'_{II}$  be a FVS of  $DG_{II}$  (Use Procedure Find-FVS).

(3)  $DG_{III} = DG \perp (V_{II} - Z'_{II})$ .

(4) Let  $Z'_I$  be a FVS of  $DG_{III}$  (Use Procedure Find-FVS).

(5) Let  $Z_I = Z'_I$  and  $Z_{II} = Z'_{II}$ . □

The Procedure Find-Joint-FVS first uses Procedure Find-FVS to derive a FVS ( $Z'_{II}$ ) for  $DG_{II}$ , then it derives a FVS ( $Z'_I$ ) for  $DG \perp (V_{II} - Z_{II})$ . Since  $Z'_I$



and  $Z'_{II}$  are derived separately, a minimal joint-FVS cannot be guaranteed. Furthermore, Procedure Find-FVS uses a greedy strategy to find a FVS. This means that whenever a node must be removed from a cycle, the one with least weight is selected. The Procedure Find-FVS is presented below.

A strongly connected component (*scc*) is a set of nodes that have directed edges among them, and at least one directed path exists from each node to every other node. One or more cycles exist in a *scc*. If no cycle exists in a directed graph then there is no *scc* containing more than one nodes in this graph. At least one node must be removed from a *scc* in order to break a cycle.

**Procedure Find-FVS:**

*Input:* A *DG*.

*Output:*  $Z$ , which is a *FVS* of the *DG*.

*Method:*

- (a) Mark all nodes in  $V$  as “white”, and let  $Z$  be an empty set.
- (b) Call  $H(DG)$ .
- (c) Put all nodes marked as “black” into  $Z$ .

**Procedure  $H(DG)$ :**

- (1) Find all *scc* of *DG*.

Let  $SCC = \{ \text{all } scc \text{ found with } |scc| > 1 \}$ .

- (2) If  $|SCC| = 0$ , then RETURN.
- (3) For each  $scc \in SCC$  do the following:

(3.1) Pick a node  $v_i \in scc$ , such that  $w_i < w_j, \forall v_j \in scc$ .

(3.2) Mark  $v_i$  as “black”.

(3.3) Remove  $v_i$  from  $scc$ , i.e.  $scc' = scc \setminus \{v_i\}$ .

(3.4) Call  $H(scc')$ . □

In step (1) the algorithm given in [4] is used to find all *scc* of a *DG*. Procedure  $H$  calls itself recursively. The major function of procedure  $H$  is to find all *scc* of a graph. A node having a minimal weight is removed from each *scc*. The remaining graph is again checked for *scc*. More nodes are removed if more *scc* are found. This process continues until no more *scc* containing more than one node are found. The

solution set  $Z$  consists of all the nodes removed during the process, i.e., those nodes that are marked as “black”.

The complexity of Procedure Find-FVS is equal to that of Procedure H. The complexity of step (1) in Procedure H is  $O(MAX(n, e))$  since the procedure described in [4] is used. In the worst case, only one node is removed from the *scc* in step (3). If there are  $n$  nodes in the *DG*, the complexity of Procedure H is  $O(n * MAX(n, e))$ , i.e., the complexity of Procedure Find-FVS is  $O(n * MAX(n, e))$ .

The complexity of Procedure Find-Joint-FVS is  $O(n_1 * MAX(n_1, e_1)) + O(n_2 * MAX(n_2, e_2))$  or  $O(n * MAX(n, e))$ . Therefore the complexity of Algorithm TSA is  $O(n * MAX(n, e))$  when Procedure Find-Joint-FVS is used in step (3) of Algorithm TSA.

In conclusion, the complexity of Algorithm TSA is  $O(n2^n)$  when an optimal solution is required, and  $O(n * MAX(n, e))$  when a sub-optimal solution can be used.

TSA has been applied to several examples. The results are shown in Table 7.1(a). Each example is modeled as a *DG* and is described by a 4-tuple  $(n, e, w, m)$ , where  $n$  is the number of nodes,  $e$  is the number of edges,  $w$  is the total weight of the *DG*, and  $m$  is the number of test vectors. The connectivity of the *DG* is not shown. Column *UB* indicates the worst case situation, i.e., where each node is scanned twice to make sure that test results correspond to appropriate test vectors. The columns *Heuristic* and *Optimal* indicate the values for schedules derived by using Procedures Find-Joint-FVS and Find-Min-Joint-FVS, respectively. All the values found by Procedure Find-Min-Joint-FVS are minimal. For the Examples 1, 2, 3, 7 and 8, Procedure Find-Joint-FVS also finds the optimal solution. Table 7.1(b) shows the results in percentage. The saving *SS* is calculated from Equation (4).

$$\begin{aligned}
 SS &= 1 - \frac{N_S}{UB} \\
 &= 1 - \frac{(m-1) \sum_{v_i \in Z_I} w_i + \sum_{v_i \in Z_{II}} w_i + \sum_{v_i \in V_I} w_i + m \sum_{v_i \in V} w_i}{2m \sum_{v_i \in V} w_i} \quad (7.4)
 \end{aligned}$$

| Ex. | $n$ | $m$ | $e$ | $w$   | <i>Heuristic</i> | <i>Optimal</i> | UB     |
|-----|-----|-----|-----|-------|------------------|----------------|--------|
| 1   | 3   | 15  | 1   | 500   | 7,500            | 7,500          | 15,000 |
| 2   | 3   | 20  | 2   | 700   | 14,700           | 14,700         | 28,000 |
| 3   | 3   | 20  | 3   | 300   | 8,100            | 8,100          | 12,000 |
| 4   | 4   | 20  | 6   | 355   | 8,880            | 8,880          | 14,200 |
| 5   | 5   | 30  | 8   | 500   | 23,562           | 21,068         | 30,000 |
| 6   | 6   | 15  | 11  | 1,900 | 40,200           | 35,100         | 57,000 |
| 7   | 5   | 25  | 12  | 1,350 | 39,420           | 39,420         | 67,500 |
| 8   | 4   | 30  | 11  | 1,450 | 60,900           | 60,900         | 87,000 |
| 9   | 7   | 20  | 15  | 830   | 25,505           | 22,845         | 33,200 |

(a)

| Ex. | <i>Heuristic</i> | <i>Optimal</i> |
|-----|------------------|----------------|
| 1   | 50               | 50             |
| 2   | 47.5             | 47.5           |
| 3   | 32               | 32             |
| 4   | 37               | 37             |
| 5   | 21               | 30             |
| 6   | 29               | 38             |
| 7   | 42               | 42             |
| 8   | 30               | 30             |
| 9   | 23               | 31             |

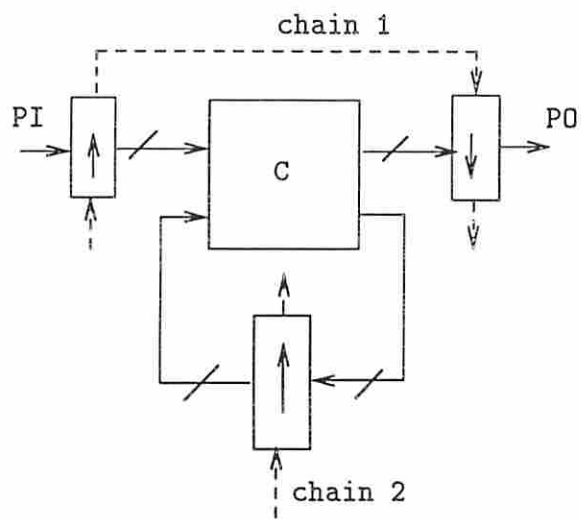
(b)

Table 7.1: Typical results; (a)  $N_S$ , (b) saving  $SS$  (in %) on test time.

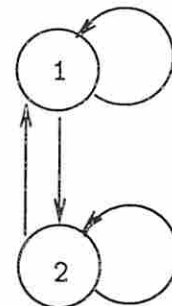
Note that  $SS$  cannot exceed 50 percent. This is because at least one shift operation is required for applying each bit of the test data and there are  $m * \sum_{v_i \in V} w_i$  bits in the test data. So  $SS$  is limited to  $1 - \frac{m * \sum_{v_i \in V} w_i}{2m \sum_{v_i \in V} w_i} = 0.5$ . Since the connectivity of these examples are different, it is misleading to compare one example with another. However, it is clear that the smaller the value of  $(m - 1) \sum_{v_i \in Z} w_i + \sum_{v_i \in Z_{II}} w_i + \sum_{v_i \in V_I} w_i$ , the larger the saving.

## 7.6 An Extension to Full Scan

The TSA Algorithm can also be used to schedule tests for a chip designed with full scan capability. Figure 7.7(a) shows a chip designed with the boundary scan architecture. Two scan chains are used to test the circuit  $C$ . Scan chain 1 is the boundary scan register. Scan chain 2 consists of all internal scan cells. Figure 7.7(b) shows a directed graph that can be used to schedule tests. Test vectors for  $C$  are applied and observed via these scan chains. Once a test vector is generated, it must be partitioned into two segments before being applied. The *Scan* function described earlier can be used to apply test vectors to these scan chains. It is necessary to find a schedule which can properly apply vectors to test  $C$  in minimal time. This problem belongs to this new class of scheduling problems, and can be solved using algorithm TSA.



(a)



(b)

Figure 7.7: Testing a circuit via two scan chains; (a) block diagram, (b) graph model.



## Chapter 8

### Conclusions and Future Research

In this work a design-for-test tool, called BOLD, has been described. BOLD is applicable at the chip, module, subsystem and system level. Using a hierarchical design methodology BOLD can deal with both hardware and software test issues and achieves a very high degree of testability and maintainability. A system designed using BOLD can support fault detection and isolation in a timely and cost effective manner. Hence system availability is increased and the hardware life-cycle costs are decreased.

Various issues related to the design of a hierarchically testable and maintainable system are dealt with in BOLD. These issues include the support of both test hardware and software. In particular, the following material has been described in depth: (1) design of on-chip and module test controllers; (2) definitions of test languages and the support of test synthesizers for these languages; (3) algorithms to evaluate various tradeoffs between test time and controller complexity; and (4) algorithms that lead to enhanced interconnect testing.

#### 8.1 On-Chip Test Controller

The design of the on-chip test controllers presented are based on the boundary scan architecture so as to conform to the IEEE Std. 1149.1. These designs include both bus-dependent and autonomous controllers. A bus-dependent controller requires

the assistance from the test bus during the entire test process. Controllers for two commonly used kernels and a more complex kernel have been designed. When designing such a controller, there exists a problem of mapping the test bus states to the test control signals of the kernel. A mapping algorithm has been provided to solve this problem. Using this algorithm a controller can be designed that requires a minimal number of instructions.

Two design styles (serial or parallel) for the autonomous on-chip controllers have been presented. A serial controller can test many kernels in sequence. It requires less hardware overhead than a parallel controller. Designs for both the hard-wired and microprogrammed serial controllers have been illustrated. A parallel controller can test many kernels simultaneously and reduces the overall chip test time at the expenses of extra hardware. Three techniques have been illustrated in the design of a parallel controller for testing many scan-type kernels. These techniques, referred to as interleaved design, tree-of-counters design, and counter-sharing design, reduce the hardware overhead by sharing resources. A comparison of the performance of these three designs shows that no one design is always better than the other two.

## 8.2 Module Test Controller

The design of a family of universal module test controller was presented. The module controllers differ by the test programs they execute, the number of test busses they control, and the expansion units they employ. One important aspect of their design is the use of a test channel. A test channel contains a boundary scan master that can communicate with an on-chip controller over the boundary scan bus. In addition, test vectors can be generated and results can be compressed in the test channel.

The processor used in the module controller can control the test channel by reading from or writing to its internal registers. Once initiated by a processor, a test channel can completely control a boundary scan bus, thus eliminating the need

for the processor to deal with detailed bus activities. The test process can thus be represented as high level processor instructions.

A prototype of the module test controller has been built and tested. The prototype is based on an IBM AT computer and a test channel built using the Actel field programmable gate array technology. Compared to most conventional automatic test equipment, the cost of the proposed module test controller is much cheaper and the performance is far superior.

### 8.3 Test Program Synthesis

One of the major contribution of this work is the synthesis of test programs for the system under test. The synthesis process starts with the preparation of test description files, represented in high level languages, for each chip and module. Synthesis softwares have been provided to translate these descriptions into test program, which can then be used to drive the test controllers embedded at various hardware units. These controllers can then control the testing process of their associated hardware units. The entire system is therefore tested.

The languages have been designed in such a way that test files can be understood by a designer with little or no knowledge in testing. Major advantages of this approach include the reduced time in developing test software, and the increased maintainability and reliability of the test software since it can be checked and synthesized quickly.

### 8.4 Controller Minimization

The time required to test a module is related to the complexity of the module and chip test controllers. In general, the more complex the controllers are, the shorter is the module test time. Tradeoffs can be made so that the module test time is bounded and the overall controller complexity is minimized. The test program synthesis technique provides a way to calculate the time required to test a module



or chip. This ability facilitates the design tradeoff between the module test time and the overall controller complexity. Two approaches that can be used to determine the complexity for each controller have been presented. Both approaches can minimize the overall controller complexity while keeping the module test time bounded.

## 8.5 Interconnect Test Generation

The results presented for test and diagnosis of interconnects are superior to all previous approaches in that all diagnosable faults can be identified. It has been shown that there exists diagnosable faults in a wiring network which cannot be identified by any of the previous approaches, including the complementary counting sequence [64], the independent set [16], the diagonally independent sequence [34], the W-Test Algorithm [25], the C-Test Algorithm [34] and Method 3 in [16]. The faults that lead to the deficiencies in these previous approaches are summarized and explained.

Various levels of diagnostic resolution have been defined. In particular, a diagnostic level where all diagnosable faults are identified is defined. Two maximal diagnosis conditions have been presented and proved to be both necessary and sufficient for identifying all diagnosable faults.

A property called set-cover independent is introduced. A test sequence that is set-cover independent must have a walking ones sequence (for wired-OR model) as its subsequence. It has been shown that a set-cover independent set is both necessary and sufficient for achieving maximal diagnosis. In addition, a universal test set has been proposed to identify all diagnosable faults in a network regardless of the fault model used.

Two adaptive algorithms that achieve maximal diagnosis have been presented. They can reduce the size of the test set by employing a two-step diagnosis scheme. Both algorithms first apply a maximal independent set to eliminate aliasing syndromes. The responses are analyzed and based on the initial results, the second part of the test set is generated. Without the information from the first

part, it is impossible to reduce the size of the test set. However, it is not clear whether these algorithms can generate minimal test sets.

In practice, a net can only be shorted to a set of neighboring nets due to the physical structure of the network. When neighborhood information is employed, it is possible to generate a reduced test set. A one-step diagnosis algorithm that uses this information has been presented. It has been shown that this algorithm can generate a minimal size test set to achieve maximal diagnosis.

## 8.6 Interconnect Test Scheduling

The problem of applying interconnect test vectors via multiple boundary scan chains was investigated. The objective is to apply test vectors in such a way so as to minimize the total test time. This problem leads to a new class of scheduling problems. Theorems pertaining to optimal schedules are derived. Based on these results an algorithm has been constructed that generates an optimal schedule. The test time is greatly reduced when the optimal schedule is adopted. A reduction in the range of 30 to 50 % has been achieved in the examples examined so far. The search procedure used in this algorithm can be further improved. However, no effort has been made to find a better search procedure. This is due to the following two reasons: (1) the problem is NP-Complete; and (2) the current procedure performs well when the problem size  $n$  is less than 15, which includes most foreseeable applications.

## 8.7 Future Research

The enhancement of BOLD includes both hardware and software aspects. The proposed MMC is not as efficient as possible due to limitations in the test channel. The inefficiency could be a problem if the volume of data transfer between the memory and the test channel is large. Also, the test channel does not support an arbitrary sequence of values on the TMS line. This could be a problem in some applications. To make the BOLD system more general, some suggestions are listed



below. This includes the realization of a more efficient MMC, which can be done through the redesign of the test channel.

### 8.7.1 On-chip Test Controller

Currently, BOLD supports the synthesis of test software but not the test controllers. The system can be improved if the CMCs could be automatically synthesized. The synthesis of a CMC should contain two parts, namely the Test Access Port (TAP) and the BIT controller.

**TAP:** The inclusion of a TAP to each chip should be done automatically, i.e., independent of the logic design of the chip. The synthesis of a TAP can be achieved as follows.

1. Generate the TAP controller, which is a machine defined by the IEEE Std. 1149.1.
2. Generate the boundary register consisting of a boundary scan cell for each I/O pin of the chip. The scan cells should be properly connected and controlled so that it satisfies the IEEE Std. 1149.1 requirements. In particular, the Boundary Register should support the predefined public instructions EX-TEST, INTEST, and SAMPLE.
3. Generate the instruction register (IR). The length of the IR should be determined first. This can be done if the total number of instructions required in controlling the BIT controller is known. The opcode of the instructions should also be determined to allow the BIT controller to be synthesized. The mapping algorithm proposed in chapter 2 can be used to determine the total number of instructions required.
4. Generate the bypass register and the identification register. The latter is optional.

The synthesis of the TAP should be treated as a development project since it is not very difficult and requires little research.

**BIT Controller:** The synthesis of a BIT controller should start with the representation of test schedules. Given a circuit that has been partitioned into testable kernels, it is necessary to organize the test into sessions. During each session a test schedule is required. The difficult part of the synthesis is the minimization of the area overhead of the BIT controller. This problem is further complicated by the fact that the area overhead of the TAP, which varies as the length of the IR changes, should also be considered. The proposed control graph, which represents a test procedure in terms of test control signals, can be used to represent a test schedule. Further research is needed to find the best way for representing test schedules.

**CTL Generation:** The CTL description of a chip can be automatically generated along with the CMC. This will be an important feature in the integration of both chip and module testing.

## 8.7.2 Module Test Controller

### Improved Test Channel

The current implementation of the MMC prototype is not ideal. The major reason is that the test channel of the MMC is implemented using an ACT1020 device, which has a very limited capacity. The interface between the processor and the test channel is also not ideal since the data transfer between the memory and the test channel is not fast enough without interrupting the test activities. In addition, many proposed features of the test channel designs are not included.

An ideal test channel should contain the following features.

1. The test channel should contain a large memory so that all the test data for each test session can be loaded into the test channel without the need to access the external memory.
2. The data transfer between the test channel and the external memory unit should be very high. Either the direct memory access (DMA) operations or the use of Direct signal proposed in this work are desirable.

3. The FSM1 of the test channel should not be limited to only some predefined state transition sequences. The test channel should be able to set an arbitrary sequence of values on the TMS line.
4. The test channel itself should be testable, i.e., it should contain a CMC.
5. A clock control circuitry should be added to the test channel. This circuitry should be able to control the application of the test clock TCK and the system clock.

This ideal test channel can be realized using either full-custom or standard cell VLSI design approaches. Both approaches allow for a large number of gates and a large memory to be built onto a single chip.

#### **MMC for a Self-Testable Module**

To make a module fully self-testable, a complete MMC should be built into the module. To be practical, it is desirable to have a complete MMC including a test channel, a processor and a large memory unit packaged in from one to three chips. When both BILBO and RUNBIST TDMs are used in a module as the primary means of testing, the memory requirements can be small. In this case, an MMC can be built from two chips, namely a microcontroller chip that contains an internal RAM, and a test channel chip.

#### **MMC in a Chip**

A more ambitious project would be to develop a single chip MMC. This MMC should contain three major units, i.e., a processor, a RAM and a test channel. The instruction set of the processor can be very small. In fact both the processor and the test channel can be closely coupled, i.e., no obvious boundary need exist between these two units.

### **8.7.3 Test Program Synthesis**

The test program synthesis aspect of BOLD can be improved as follows.

1. The synthesis of test programs in BOLD requires test description files as the inputs. Currently, these files are manually prepared. The automatic generation of these description files should be a major goal for improving the capability of BOLD. Initial investigation indicates that the test description of a chip can be generated as a by-product of the on-chip test controller synthesis, and that the test description of a module can be generated from a computer-aided design system. The latter assumes that the module contains all boundary scan devices.
2. Currently, the synthesizers do not fully support the capability of putting an arbitrary sequence of values on the TMS line. The reason for this is that the implemented test channel is not ideal. The required operations are only partially supported by the hardware. When the test channel is properly redesigned, the synthesizers will be enhanced by supporting this capability.
3. Currently, the lowest level of hardware that can be dealt with using BOLD is a chip. However, most of the design-for-test tools, such as TDES [2] and SIESTA [26], operate on circuit blocks or kernels of a chip. More work needs to be done to enhance BOLD with the capability of describing the test aspects of a circuit block. By so doing, BOLD can be integrated with these design-for-test tools.

#### 8.7.4 Controller Minimization

The minimization algorithms presented in this work assumed that an MMC contains a single test channel. However, when employing a very fast processor, an MMC can control more than one test channel simultaneously without interrupting their operations. New controller minimization algorithms should be developed to incorporate these changes.

### 8.7.5 Interconnect Test

1. The two-step diagnosis algorithms, presented in chapter 6, do not guarantee the generation of a minimal size test set. Further work is required to develop an algorithm that can generate a minimal size test set for the two-step diagnosis approach.
2. The presented interconnect test methods are based on the assumption that all chips contained in the module have the boundary scan architecture. However, most existing modules do not satisfy such requirement. New test methods need be developed such that the interconnect can be tested under the incomplete boundary scan environment. In addition, the testing of the glue logic between boundary scan chips should be considered.
3. The proposed test methods investigate only the DC behavior of the interconnect. Modern systems are designed to operate at such a high speed that the testing for the DC correctness of the interconnect is no longer sufficient. Therefore faults related to the AC behavior, such as crosstalk between signal lines, transmission line effects, line delay faults, should be dealt with. Future work should investigate the possibility of testing these faults in the boundary scan framework.



## Reference List

- [1] M.S. Abadir and M.A. Breuer, "Constructing Optimal Test Schedules for VLSI Circuits Having Built-In Test Hardware", *Proc. 15th Int'l Symp. on Fault-Tolerant Computing*, pp. 165-170, June 1985.
- [2] M.S. Abadir and M.A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips", *IEEE Design & Test of Computers*, pp. 56-68, August 1985.
- [3] M.S. Abadir and M.A. Breuer, "Test Schedules for VLSI Circuits Having Built-In Test Hardware", *IEEE Trans. on Computers*, Vol. C-35, No. 4, pp. 361-367, April 1986.
- [4] A.V. Aho, J.E. Hopcroft and J.D. Ullman, "The Design and Analysis of Computer Algorithms", *Addison-Wesley*, Readings, Massachusetts, pp. 193-194, 1974.
- [5] L. Avra, "A VHSIC ETM-BUS Compatible Test and Maintenance Interface", *Proc. Int'l Test Conf.*, pp. 964-971, 1987.
- [6] P.H. Bardell and W. McAnney, "Self-Testing of Multichip Logic Modules", *Proc. Int'l Test Conf.*, pp. 200-204, 1982.
- [7] J. Beausang and A. Albicki, "A Methodology for Designing Self-Testable VLSI Chips: Synthesis, Part I, A Model for Self-Testable Chips", *Technical Report EL-87-05*, Department of EE, University of Rochester, 1987.
- [8] F. Beenker, K. Eerdewijk, R. Gerritsen, F. Peacock and M. van der Star, "Macro Testing: Unifying IC and Board Test", *IEEE Design & Test of Computers*, pp. 26-32, December 1986.
- [9] F. Beenker, "Systematic and Structured Methods for Digital Board Testing", *VLSI System Design*, pp. 50-58, January 1987.
- [10] G. Borriello and R. H. Katz, "Synthesis and Optimization of Interface Transducer Logic", *Proc. Int'l Conf. Computer-Aided Design*, pp. 274-277, 1987.

- [11] M.A. Breuer, "On-Chip Controller Design for Built-In-Test", *Technical Report CRI-88-04*, Department of EE-Systems, University of Southern California, December 1985.
- [12] M.A. Breuer and J.C. Lien, "A Methodology for the Design of Hierarchically Testable and Maintainable Digital Systems", *Proc. 8th Digital Avionics System Conf.*, pp. 40-47, 1988.
- [13] M.A. Breuer and J.C. Lien, "A Test and Maintenance Controller for a Module Containing Testable Chips", *Proc. Int'l Test Conf.*, pp. 502-513, 1988.
- [14] M.A. Breuer, R. Gupta, and J.C. Lien, "Concurrent Control of Multiple BIT Structures", *Proc. Int'l Test Conf.*, pp. 431-442, 1988.
- [15] W.O. Budde, "Modular Testprocessor for VLSI Chips and High-Density PC Boards", *IEEE Trans. on CAD*, Vol. 7, No. 10, pp. 1118-1124, October 1988.
- [16] W.T. Cheng, J.L. Lewandowski and E. Wu, "Diagnosis for Wiring Interconnects", *Proc. Int'l Test Conf.*, pp. 565-571, 1990.
- [17] K.K. Chua and C.R. Kime, "Selective I/O Scan: A Diagnosable Design Technique for VLSI Systems", *Comput. Math. Applic.*, Vol. 13, No. 5/6, pp. 485-502, 1987.
- [18] G.L. Craig, C.R. Kime, and K.K. Saluja, "Test Scheduling and Control for VLSI Built-In Self-Test", *IEEE Trans. on Computers*, Vol. C-37, No. 9, pp. 1099-1109, September 1988.
- [19] C.A. Dennis, "Common Signal Processor: Application and Design", *IBM Technical Directions*, Federal Systems Division, Vol. 13, No. 1, pp. 14-20, 1987.
- [20] IBM, Honeywell and TRW, "VHSIC Phase 2 INTEROPERABILITY STANDARDS", *ETM-BUS Specification*, December 1986.
- [21] E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability", *Proc. 14th Design Automation Conf.*, pp. 462-467, June 1977.
- [22] P.P. Fasang, J.P. Shen, M.A. Schuette and W.A. Gwaltney, "Automated Design for Testability of Semicustom Integrated Circuits", *Proc. Int'l Test Conf.*, pp. 558-564, 1985.
- [23] A. El Gamal, "Protozone: The PC-Based ASIC Design Frame", *EE218 Handout No. 7*, Stanford University, Winter 1990.
- [24] M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", *W.H. Freeman and Company*, New York, 1974.

- [25] P. Goel and M.T. McMahon, "Electronic Chip-In-Place Test", *Proc. Int'l Test Conf.*, pp. 83-90, 1982.
- [26] Rajesh Gupta, "Advanced Serial Scan Design for Testability", *Ph.D. Dissertation*, Department of EE-Systems, University of Southern California, 1991.
- [27] J.E. Haedtke and W.R. Olson, "Multilevel Self-Test for the Factory and Field", *Proc. Annual Reliability and Maintainability Symp.*, pp. 274-279, 1987
- [28] P. Hansen, "The Impact of Boundary-Scan on Board Test Strategies", *Proc. ATE & Instruments Conf. East*, pp. 35-40, Boston, June 1989.
- [29] A. Hassan, J. Rajska, and V.K. Agarwal, "Testing and Diagnosis of Interconnects using Boundary Scan Architecture", *Proc. Int'l Test Conf.*, pp. 126-137, 1988.
- [30] A. Hassan, V.K. Agarwal, J. Rajska and B.N. Dostie, "Testing of Glue Logic Interconnects Using Boundary Scan Architecture", *Proc. Int'l Test Conf.*, pp. 700-771, 1989.
- [31] C.L. Hudson, Jr. and G.D. Peterson, "Parallel Self-Test with Pseudo-Random Test Patterns", *Proc. Int'l Test Conf.*, pp. 954-963, 1987.
- [32] IEEE Standard 1076-1987, "IEEE Standard VHDL Language Reference", *IEEE Standards Board*, 345 East 47th Street, New York, NY 10017, March 1988.
- [33] IEEE Standard 1149.1-1990, "IEEE Standard Test Access Port and Boundary Scan Architecture," *IEEE Standards Board*, 345 East 47th Street, New York, NY 10017, May 1989.
- [34] N. Jarwala and C.W. Yau, "A New Framework for Analyzing Test Generation and Diagnosis Algorithms for Wiring Interconnects", *Proc. Int'l Test Conf.*, pp. 63-70, 1989.
- [35] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler", in B.W. Kernighan and M.D. McIlroy, *UNIX Program's Manual*, Bell Laboratories, 7th Edition, 1978.
- [36] N. Kanopoulos, et al., "A New Implementation of Signature Analysis for Board Fault Isolation Testing", *Proc. Int'l Test Conf.*, pp. 730-736, 1987.
- [37] W.H. Kautz, "Testing for Faults in Wiring Networks", *IEEE Trans. on Computers*, Vol. C-23, No. 4, pp. 358-363, April 1974.
- [38] B. Konemann, J. Mucha and G. Zwiehoff, "Built-In Logic Block Observation Techniques", *Proc. Int'l Test Conf.*, pp. 37-41, 1979.



- [39] S.Y. Kung, S.C. Lo, S.N. Jean and J.N. Hwang, "Wavefront Array Processors - Concept to Implementation", *IEEE Computer*, pp. 18-33, July 1987.
- [40] D. van de Lagemaat and H. Bleeker, "Testing a Board with Boundary Scan", *Proc. Int'l Test Conf.*, pp. 724-729, 1987.
- [41] J.J. LeBlanc, "LOCST: A Built-In Self-Test Technique", *IEEE Design & Test of Computers*, pp. 45-52, November 1984.
- [42] M.E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator", in B.W. Kernighan and M.D. McIlroy, *UNIX Program's Manual*, Bell Laboratories, 7th Edition, 1978.
- [43] J.C. Lien and M.A. Breuer, "A Universal Test and Maintenance Controller for Modules and Boards", *IEEE Trans. on Industrial Electronics*, Vol. 36, No. 2, pp. 231-240, May 1989.
- [44] J.C. Lien, "A Module Maintenance Controller Prototype", *Technical Report CENG 90-14*, Department of EE-Systems, University of Southern California, June 1990.
- [45] J.C. Lien and M.A. Breuer, "An Optimal Scheduling Algorithm for Testing Interconnect Using Boundary Scan", *Journal of Electronic Testing: Theory and Applications*, Vol. 2, No. 1, pp. 117-130, March 1991.
- [46] J.C. Lien and M.A. Breuer, "Maximal Diagnosis of Wiring Networks", *Technical Report CENG 91-2*, Department of EE-Systems, University of Southern California, February 1991.
- [47] T.S. Liu, "The Role of a Maintenance Processor for a General Purpose Computer System", *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 507-517, June 1984.
- [48] TRW, "MMN Architecture", Private Correspondence.
- [49] J.G. Malcolm, "BIT False Alarms: An Important Factor In Operational Readiness", *Proc. Annual Reliability and Maintainability Symp.*, pp. 206-212, 1982.
- [50] C. Maunder and F. Beenker, "BOUNDARY-SCAN: A Framework for Structured Design-For-Test", *Proc. Int'l Test Conf.*, pp. 714-723, 1987.
- [51] E.J. McCluskey, "Built-In Self-Test Techniques", *IEEE Design & Test of Computers*, pp. 21-28, April 1985.
- [52] M.J. Ohletz, T.W. Williams and J.P. Mucha, "Overhead in Scan and Self-Testing Designs", *Proc. Int'l Test Conf.*, pp. 460-470, 1987.

- [53] C.H. Papadimitriou and K. Steiglitz, "Combinatorial Optimization, Algorithms and Complexity", *Prentice-Hall, Inc.*, Englewood Cliffs, New Jersey, pp. 421-422, 1982.
- [54] K.P. Parker, "The Impact of Boundary Scan on Board Test", *IEEE Design & Test of Computers*, pp. 18-30, August 1989.
- [55] K.P. Parker and S. Oresjo, "A Language for Describing Boundary Scan Devices", *Proc. Int'l Test Conf.*, pp. 222-234, 1990.
- [56] G.D. Robinson and J.G. Deshayes, "Interconnect Testing of Boards with Partial Boundary Scan", *Proc. Int'l Test Conf.*, pp. 572-581, 1990.
- [57] K. Sakashita, T. Hashizume, T. Ohya, I. Takimoto and S. Kato, "Cell-Based Test Design Method", *Proc. Int'l Test Conf.*, pp. 909-916, 1989.
- [58] J. Sayah and C.R. Kime, "Test Scheduling For High Performance VLSI System Implementations", *Proc. Int'l Test Conf.*, pp. 421-430, 1988.
- [59] J.H. Stewart, "Application of Scan/Set for Error Detection and Diagnostics", *Proc. Semiconductor Test Conf.*, pp. 152-158, 1978.
- [60] IBM, Honeywell and TRW, "VHSIC Phase 2 INTEROPERABILITY STANDARDS", *TM-BUS Specification*, December 1986.
- [61] J. Turino, "IEEE P1149 Proposed Standard Testability Bus - An Update with Case Histories", *Proc. Int'l Conf. Computer Design*, pp. 334-337, 1988.
- [62] N. Vasanthavada, "TEA Design Review, Built-In Test", Research Triangle Institute, June 1987.
- [63] S. Vining, "Tradeoff Decisions Made for P1149.1 Controller Design", *Proc. Int'l Test Conf.*, pp. 47-54, 1989.
- [64] P.T. Wagner, "Interconnection Testing with Boundary Scan", *Proc. Int'l Test Conf.*, pp. 52-57, 1987.
- [65] L. Whetsel, "A Proposed Standard Test Bus and Boundary Scan Architecture", *Proc. Int'l Conf. on Computer Design*, pp. 330-333, 1988.
- [66] T.W. Williams and K.P. Parker, "Design For Testability - A Survey," *The Proc. of the IEEE*, Vol. 71, No. 1, pp. 98-112, January 1983.
- [67] N. Wirth, "Algorithm + Data Structures = Programs", *Prentice-Hall*, Englewood Cliffs, New Jersey, pp. 188-189, 1976.



- [68] C.W. Yau and N. Jarwala, "A Unified Theory for Designing Optimal Test Generation and Diagnosis Algorithms for Board Interconnects", *Proc. Int'l Test Conf.*, pp. 71-77, 1989.
- [69] M.A. Breuer and X. Zhu, "A Knowledge-Based System for Selecting Test Methodology for a PLA", *Proc. 22nd Design Automation Conf.*, pp. 259-265, June 1985.