An Integrated Test Controller Synthesis System

Debaditya Mukherjee

CENG Technical Report 94-21

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4469

December 1994

AN INTEGRATED TEST CONTROLLER SYNTHESIS SYSTEM

by

Debaditya Mukherjee

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

December 1994

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

This dissertation, written by

DEBADITYA MUKHERJEE

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 requirements for the degree of*

DOCTOR OF PHILOSOPHY

Alice C. Parker
Dean of Graduate Studies

Date September 1, 1994

DISSERTATION COMMITTEE

Mel Breuer
Chairperson

M. Pedram

Ming-Deh Huang

Dedication

To
my wife, Sangeeta
and
my parents.

Acknowledgements

I would like to thank my advisor Prof. Melvin Breuer for his encouragement, guidance and support during the course of this dissertation. His scholarship has been a great source of inspiration and his invaluable constructive criticisms have enhanced the quality and presentation of my research over the years. I consider it my privilege to have worked with him. I express my deep sense of appreciation to Prof. Massoud Pedram for providing me with a clear understanding of the synthesis aspect of my research. His keen intellect, enthusiasm and support have gone a long way towards making my doctoral work a rewarding experience. I would like to thank Prof. Alice Parker for being part of my qualifying committee and providing me with invaluable advice and help during the entire course of my stay at USC. Special thanks go out to Prof. Sandeep Gupta for also being part of my qualifying committee, for his help and long discussions on various topics. I wish to extend my thanks to Prof. Ming-Deh Huang for serving on my qualifying and dissertation committees.

Thanks go to a number of friends and colleagues at USC with whom I have interacted over the years. My association with Dr. Charles Njinda has greatly enriched my outlook and I would like to thank him for his interest in my research efforts and helpful suggestions. As my friend and officemate, Sridhar Narayanan made the past few years enjoyable through interminable discussions over endless cups of coffee. Rajesh Gupta, Kuen-Jong Lee, Jung-Cheun Lien, Amit Majumdar, Rajagopalan Srinivasan, Ishwar Parulkar, Pravil Gupta, Atul Ahuja, Chih-Tung Chen, Sen-Pin Lin and Mody Lempel have all contributed to making my stay at USC memorable. I am also indebted to the staff members of the Electrical Engr. Dept. at USC for their help, particularly Mary Zittercob.

I gratefully acknowledge the financial support provided by the Advanced Research Projects Agency (ARPA) through Contract No. J-FBI-90092 (monitored by the Federal Bureau of Investigation).

My sincere gratitude goes to my best friend, companion, and wife Sangeeta for her faith, love, support and encouragement over the years that has contributed significantly to the successful completion of my doctorate program. Her positive outlook to life brightened the long hours of hard work. Last but not the least, I would like to thank my parents for instilling in me the value of education and their endless care, support and dedication throughout my life.

Contents

List Of Tables

List Of Figures

Abstract

A VLSI chip is made testable by modifying the functional circuitry and/or by adding extra hardware to serve as sources (destinations) of test patterns (responses) in the test mode of operation. On-chip hardware is needed to control the often complex test activation sequences. Additional control hardware is required to establish communication with the off-chip test support environment. With the increasing demands on silicon area and shorter design cycles it is important to minimize the logic and routing area of the test control hardware and to automate the controller synthesis process. We have developed an integrated test controller synthesis system, called CONSYST, that employs control schemes operating under a novel IEEE 1149.1 boundary scan standard compliant test control architecture. In this architecture controllers (decoders) for scannable registers are distributed while the test controllers for data transport paths are merged together. This merged controller can also be combined with the on-chip functional controller. Merging controllers is accomplished by using new and efficient FSM merging techniques that exploit certain unique features of the test and functional controllers. Different bus schemes have been devised to transmit control information from an Integrated Test Access Port Controller (ITAPC) to the distributed decoders. These schemes lead to a tradeoff between the number of control lines routed and the implementation cost of the ITAPC and distributed decoders. Algorithms have been developed to encode the symbols transmitted on the bus with the objective of minimizing the implementation cost of the distributed decoders and/or the ITAPC. Various characteristics of a testable design are represented in a Test Plan Description Language developed for CONSYST. Using this description CONSYST automatically synthesizes the test control and test source (destination) hardware. CONSYST can handle a wide range of scan and BIST methodologies that employ reconfigurable scan chains, complex data transport paths, pipelined test application and multifunction and/or reconfigurable test registers.

Chapter 1

Introduction

The objective of this research is the development of an integrated system to synthesize on-chip hardware for controlling Design-for-Test (DFT) and/or Built-in Self-Test (BIST) circuitry in a testable VLSI chip. Various mechanisms for controlling test circuitry have been presented in the past, but minimizing the logic and routing area of the control hardware and the task of efficiently controlling complex DFT/BIST methodologies has remained an open problem.

1.1 Design for Testability and Built-in Self-Test

Designers use DFT and BIST techniques to simplify the problem of testing VLSI chips. DFT/BIST techniques at the chip level (or at the die level in multi-chip modules) also provide the capability of building easily maintainable and diagnosable modules, boards, subsystems and systems. Examples of DFT techniques are test point insertion and global initialization capability [1]. However these are *ad hoc* approaches and *algorithmic* techniques such as adding scan capability to all flip-flops in a design (full scan) are usually preferred. DFT approaches add extra hardware to the circuit under test (CUT) to improve its testability and require substantial off-chip support (e.g., automatic test equipment) to provide test vectors and control the test application process. State of the art application-specific integrated circuits (ASICs) and instruction processor (IP) chips have over three million transistors. The use of DFT techniques alone often leads to unacceptably large test application time and significant test vector storage cost. Moreover, in scan based designs, test patterns cannot be applied every clock cycle since they need to be shifted in through a scan chain. Thus at-speed testing is not possible with scan based DFT techniques. This has led to the development of BIST techniques where on-chip hardware is used to test parts/all of a chip. A common feature of BIST approaches is the addition of on-chip test pattern generators (TPGs) and signature analyzers (SAs). Sufficient test patterns have to be generated in the test mode to achieve a satisfactory fault coverage. Though BIST techniques require less off-chip support, the area overhead incurred in incorporating these techniques is in general higher than that incurred in incorporating DFT techniques. Various parameters, such as test application time, test generation time, area overhead, performance degradation and extra I/O pins, are used to characterize the different DFT/BIST techniques. These techniques are also referred to as Test Design Methodologies (TDMs).

1.2 Test Design Methodologies

There are two aspects of a TDM, the first is the structural aspect and the other is the operational aspect. The structural aspect of a TDM is modeled by a *template.* A template specifies the type of structure or *kernel* (e.g., combinational or sequential) to which the TDM is applicable and the source (destination) of test (response) vectors. A circuit is usually partitioned into kernels or sub-circuits to simplify the testing process. To test a kernel using a TDM we need to determine how to apply test vectors and capture the test responses. Usually on-chip hardware is either modified or added to provide the source and destination of test vectors/responses and transport paths are set up to apply (capture) the vectors (responses) to (from) the kernel.

A commonly used BIST TDM is the BILBO (Built-in Logic-Block Observation) [2]. In this methodology, certain functional registers are modified into BILBO registers. BILBO registers are linked by a scan chain. In addition, the inputs of a kernel are driven by a BILBO register and the outputs drive another BILBO register. A BILBO register usually has four functions, namely parallel load (normal mode of operation), TPG (test pattern generation), SA (signature analysis) and shift (scan). The functions performed by a BILBO register is selected by the values applied to the control lines of the register. When the BILBO register is driven from the output of a kernel it operates as a SA and compresses the responses, while the BIBLO register that drives a kernel operates as a PG. A BILBO register may not be needed to operate in all four modes for testing a circuit. In this case a simpler version of the BILBO register is used where either the SA or the PG mode is dropped.

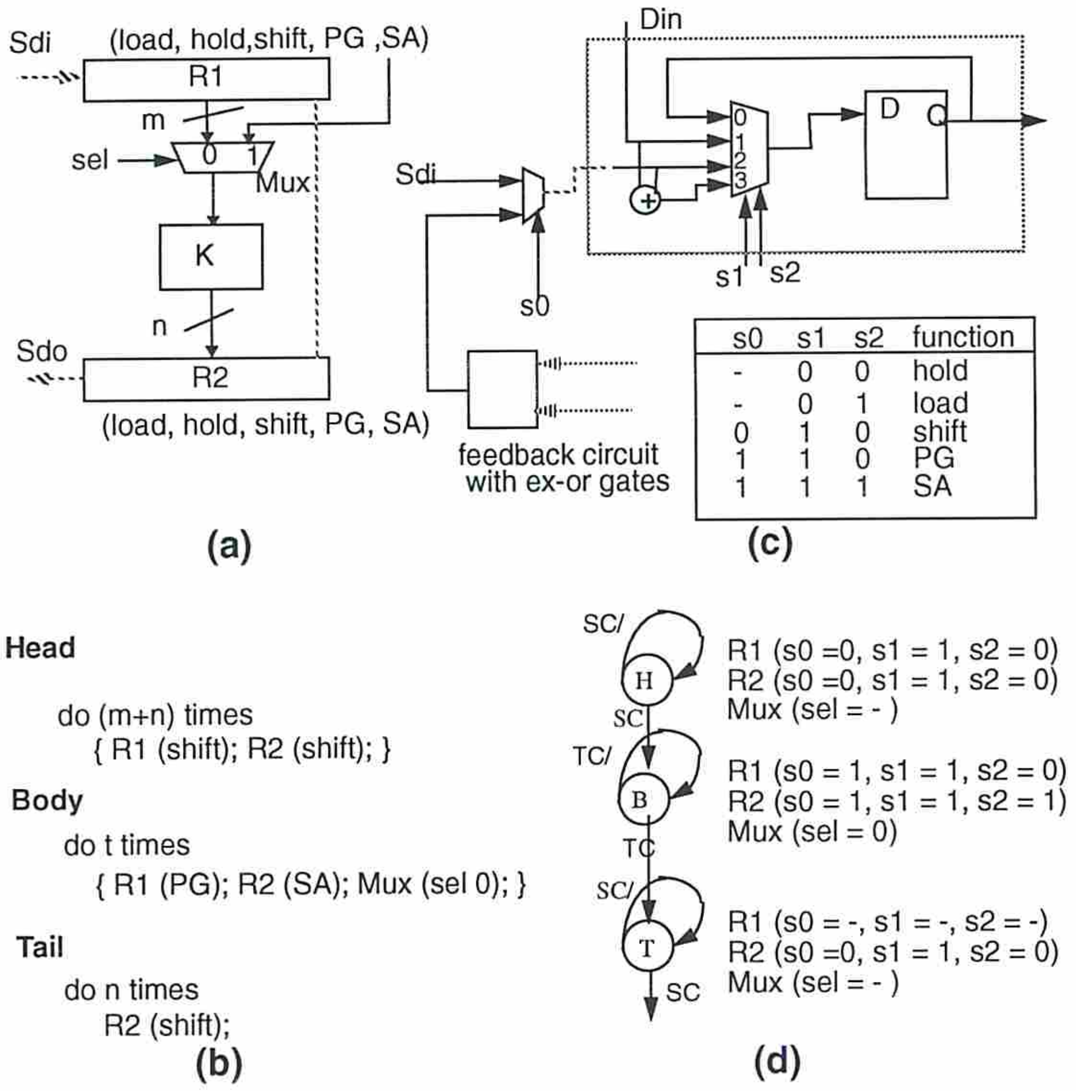
A datapath often contains random logic blocks, as well as multiplexers and buses that can be used a switching elements (or switches) to transport test (response) vectors. A switch has a set of *identity modes* or I-modes [3], where each I-mode is represented as a pair of input-output ports (P_i, P_o) of the switch. Data can be transferred unaltered from P_i to P_o in each I-mode by properly setting up values on control lines of the switch. Other common circuit structures having I-modes are registers, adders, multipliers, and ALUs. A data transfer path between the output of a circuit structure and the input of another structure is called an *identity-transfer path* or I-path [3] if data is transferred unaltered from source to destination. Complementation of data is also allowed. To be part of an I-path intermediate circuit structures must have I-modes. Test patterns and output responses can be transferred to and from a kernel if I-paths exist between the kernel and its associated TPGs and SAs. The methodology that enhances the basic BILBO TDM by employing I-paths is called an *extended* BILBO or EXTBILBO [4].

Fig. 1.1(a) shows an example of kernel K being tested using the EXTBILBO TDM. Registers R1 and R2 are functional registers which have been modified into BILBO registers. These registers also have the hold function. The registers are connected in a scan chain shown by the heavy broken line. An I-mode of the multiplexer (*Mux*) is used to create an I-path from R1 to K. The kernel K along with registers R1, R2 and the multiplexer constitute an *embedding* of the EXTBILBO TDM.

Definition 1.1 *The set of circuit components that satisfy the requirements of a TDM template constitute an* **embedding** *of the TDM.*

1.3 Test Plans

In addition to the structural component of a TDM, there is an operational component.


Sdi
(load, hold,shift, PG ,SA)
R1
m
sel
0
1
Mux
K
n
Sdo
R2
(load, hold, shift, PG, SA)
(a)
Din
Sdi
s0
0
1
2
3
D
Q
s1
s2
feedback circuit
with ex-or gates
(c)
SC/
H
SC
TC/
B
TC
SC/
T
SC
R1 (s0 =0, s1 = 1, s2 = 0)
R2 (s0 =0, s1 = 1, s2 = 0)
Mux (sel = -)
R1 (s0 = 1, s1 = 1, s2 = 0)
R2 (s0 = 1, s1 = 1, s2 = 1)
Mux (sel = 0)
R1 (s0 = -, s1 = -, s2 = -)
R2 (s0 =0, s1 = 1, s2 = 0)
Mux (sel = -)
(d)

s0	s1	s2	function
-	0	0	hold
-	0	1	load
0	1	0	shift
1	1	0	PG
1	1	1	SA

```
Head

    do (m+n) times
      { R1 (shift); R2 (shift); }
Body
    do t times
      { R1 (PG); R2 (SA); Mux (sel 0); }
Tail
    do n times
      R2 (shift);
```

(b)

Figure 1.1: (a) BILBO TDM embedding; (b) Test plan; (c) 1 bit register implementation; (d) Test plan represented as a directed graph

Definition 1.2 *A* **test plan** *for a TDM embedding describes the control activation sequence of the various components (modules) that are part of the embedding and used to test a kernel.*

In general, a test plan consists of three sections: a head, a body and a tail. The head specifies a set of actions that need to take place (initialization) before a test vector can be applied to a kernel. The body specifies a set of actions necessary for applying one or more test vectors to the kernel and capturing the results. The tail specifies closing actions.

Fig. 1.1(b) shows a test plan for the EXTBILBO TDM embedding. Since the registers are m and n bits long the head specifies that the registers must be initialized by performing a shift operation for m+n clock cycles. The initialization phase is needed to place the registers in a known state. Assume that t test vectors are needed for testing the kernel. The body specifies that register R1 (R2) should be in the PG (SA) mode of operation and the multiplexer control line should be set at a specific value (to set up an I-path from R1 to K) for t clock cycles. Once t clock cycles (and therefore t test vectors) have been applied the tail specifies that the final signature should be shifted out by setting R2 in the shift mode for n clock cycles. This test plan specification is a high level description of the test application process.

Fig. 1.1(c) shows an implementation of 1 bit registers R1 and R2. The table associated with the figure specifies the operation modes of the registers corresponding to specific control line values. Once an implementation of the hardware is fixed, the test plan can be written in a more specific manner. Fig. 1.2(d) shows the test plan represented as a directed graph. SC and TC are shift and test completion signals, respectively. The labels beside the nodes specify the values of the control signals activated while in the corresponding state. The test plan specified in this form is called a *specific* test plan and provides enough information to implement a test controller to control the test of the kernel. In the rest of this thesis we will refer to a test plan represented as a directed graph as simply a *test plan.*

Fig. 1.2(a) shows an example of a scan TDM embedding. The shift capability has been added to the registers R1 and R2. Assume that t test vectors are needed to test kernel K. A test vector is shifted into R1 by setting it up in the shift mode of operation for m clock cycles. Then one clock cycle is applied with register R2 in the load mode. At the end of the clock period the results of the test vector are loaded into R2. This response is then shifted out by setting R2 in the shift mode of operation for n clock cycles. The processes of shift-in/shift-out are usually overlapped and both registers need only be in the shift mode of operation for max(m,n) clock cycles. An implementation of a 1-bit register with load, hold and shift functions is given in Fig. 1.2(c) and a specific test plan for this embedding is shown in Fig. 1.2(d). The labels SC and TC correspond to shift completion and test signals, respectively. Note that for scan TDMs there is no explicit initialization, the shifting-in/out of test vectors (results) is considered to be part of the test process and therefore included in the body of the test plan as shown in Fig. 1.2(d).

Test plans can be *single phase* or *multiple phase.*

Definition 1.3 *A* **single phase** *test plan has at most two nodes or states in its body. If there are two states, then one must control the shift process.*

Definition 1.4 *A* **multiple phase** *test plan has two or more states in its body. If there are two states then none should be involved in the shift process.*

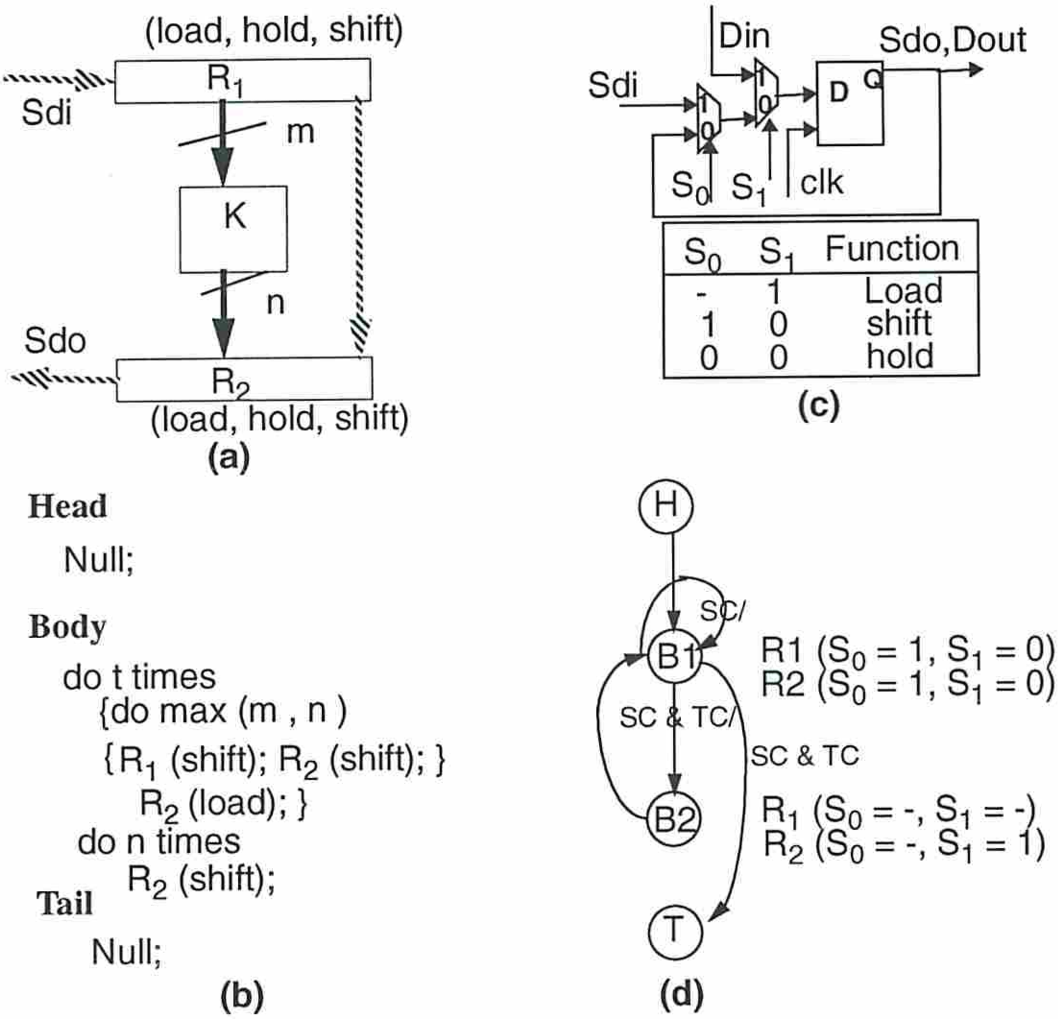
The test plans in Figs. 1.1(d) and 1.2(d) are examples of single phase tests. Note that the (body of the) test plan in Fig. 1.1(d) has 2 states and one is involved with the scan-in/out process. Some modules remain in the same mode of operation during application of test vectors, whereas others switch modes. Scannable registers *always* switch modes i.e., scan and/or hold and/or load.

Definition 1.5 *If all modules other than scan registers controlled by a test plan remain in the same mode during application of a test vector, then the test plan is* **single valued**.

Definition 1.6 *If any module (other than a scan register) controlled by a test plan changes modes during application of a test vector, then the test plan is* **multiple valued**.

1.4 Structural Classification of Controllers

Each TDM embedding needs an entity that controls the circuit components (i.e., enables certain control lines) in the test mode of operation. The test plans usually



```
Head
  Null;
Body
  do t times
    {do max (m , n )
     {R_1 (shift); R_2 (shift); }
       R_2 (load); }
   do n times
       R_2 (shift);
Tail
    Null;
```

S_0	S_1	Function
-	1	Load
1	0	shift
0	0	hold

Figure 1.2: (a) Full scan TDM embedding; (b) Test plan; (c) 1 bit register implementation; (d) Test plan represented as a directed graph

specify a sequence of control values that need to be asserted on control lines and the test plans correspond directly to state transition graphs (STGs) of Finite State Machines (FSMs). However, in certain instances a sequence of actions is not required and only one set of control values is needed for the entire duration of a test plan. In these cases the test controller can be purely combinational. Fig. 1.3 represents a structural classification of controllers. This classification is applicable to both test and functional controllers.

In Fig. 1.3, *vacuous* controllers correspond to the case where hardware on a chip is controlled directly from primary inputs. Examples of *combinational* controllers are decoders. Sequential controllers can either be implemented using FSMs, *Control Memory* or *Register-decoder pairs.* A well known example of Control Memory based controllers are microprogrammed controllers [5]. In such controllers the control values that need to be asserted on the control lines are usually stored in a read-only memory (ROM) in an encoded (vertical) or unencoded (horizontal) format. These memory locations are read out under the control of a counter (microprogram counter) and used to control on-chip hardware. *Register-decoder pairs* are used in *data-stationary* [6] controller designs. Data stationary controllers are widely used in RISC processors to control instruction/data pipelines.

FSMs are classified as linear or non-linear depending on whether the next state and output function is a linear or non-linear combination of the present states and inputs. Linear FSMs have certain attractive properties, such as easy testability, a simplified synthesis process and in some cases, a smaller implementation cost as compared to non-linear FSMs. However, for a machine to be linearly realizable, its state transition graph (STG) must possess certain properties [7, 8]. Most FSMs do not posses these properties and hence are not linearly realizable. However circular shift registers are examples of linear machines and can be used where the controller for each test plan is implemented separately.

A *Moore* FSM is one where the outputs depend only on the present state, while in a *Mealy* machine, the outputs depend on the present state as well as the inputs. The shaded area in Fig. 1.3 refers to the controller styles that we will consider in this thesis.

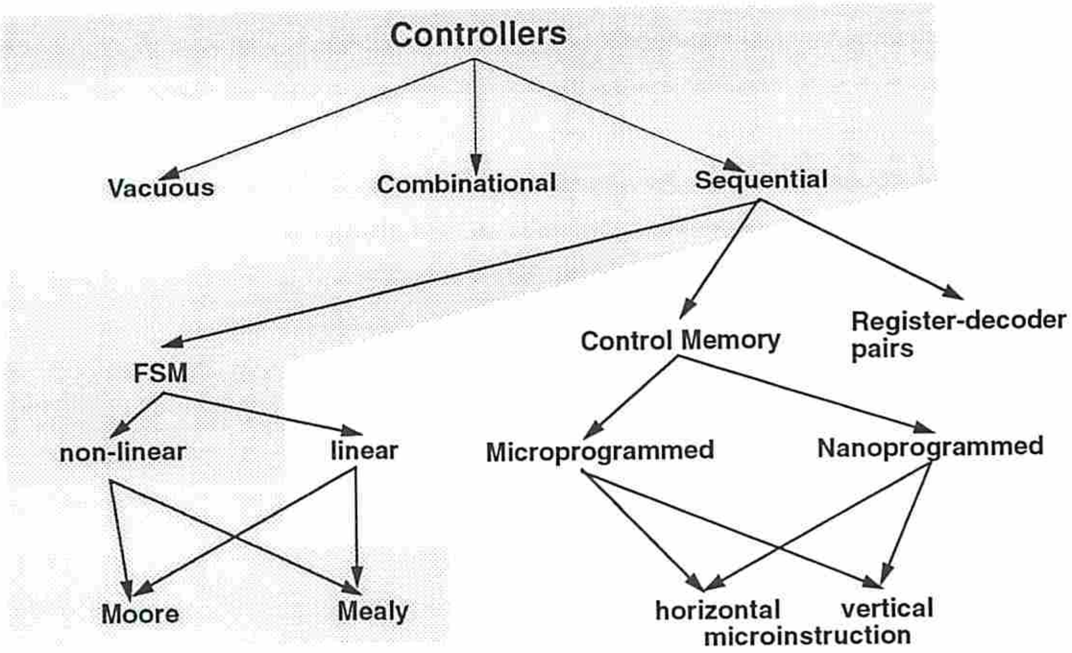


Figure 1.3: Structural classification of controllers

1.5 The Test Control Problem

A circuit can have a large number of embeddings of TDMs of the same or different types, each giving rise to a test plan. The nature of a TDM as well as the complexity of the data transfer paths between source and destination influences the complexity of the test plans. These transfer paths can be simple I-paths or other types of paths such as S-paths, T-paths and F-paths [1]. To save test resource area, resources are often shared between embeddings. Moreover a module that is in the I-path of one kernel may also be in the I-path of another kernel. Since embeddings can share data drivers, receivers and data transfer paths, the control of test plans leads to a complex design problem.

The test controller adds area overhead and may add delays on critical control paths in the chip. The area is contributed both by the logic gates needed to implement the test controller and also by the routing area of the control lines. A testable chip also requires additional control hardware to establish communication with the off-chip test support environment. This support is needed among others to shift in (out) test data (results), to initiate the on-chip controller and to test chip to chip interconnects on a loaded board. With the increasing demands on chip area and the inexorable push towards high speed designs, it is important to implement the test controller(s) with minimal area and minimal impact on chip performance.

1.6 Research Goals

The main goals and salient features of this research are as follows.

- Develop a structured test control architecture that is powerful enough to support the control requirements of various complex TDMs embedded in a circuit.
- Focus on implementing control hardware with minimal logic and routing area. Employ existing combinational and sequential synthesis tools and where applicable develop new and innovative synthesis techniques to exploit specific features of the test controller design problem.
- Conform to the IEEE 1149.1 boundary scan standard and support the set of mandatory instructions.
- Incorporate all ideas in a software system that is able to automatically synthesize and embed control circuitry in a testable design.

1.7 Organization of this Dissertation

In this chapter we have introduced the test control problem and outlined our research goals. The remainder of this dissertation is organized as follows.

Chapter 2 provides the background for this research and summarizes past work on various aspects of the test control problem. This chapter also describes the IEEE 1149.1 [9] boundary scan architecture.

Chapter 3 specifies the on-chip and off-chip partition of test control functions. This chapter also introduces and explains our IEEE 1149.1 boundary scan standard compliant *partially distributed* test control architecture. Various mechanisms for controlling scan and BIST registers have also been presented.

Chapter 4 presents implementational details of the local controllers associated with scan and BIST registers as well as techniques for implementing test buses.

Chapter 5 integrates the concepts presented in Chapters 3 and 4 and describes in detail how various scan and BIST TDMS are controlled in a testable chip. Mechanisms for controlling these chips at the board level have been presented.

Chapter 6 focuses on the problem of implementing the test controllers for a number of test plans corresponding to TDMs embedded in a testable chip. In this chapter an efficient technique for merging these test controllers into one merged controller has been presented.

Chapter 7 deals with the problem of efficiently merging a functional controller with a merged test controller.

Chapter 8 proposes a new test control architecture and presents a technique for merging a number of test controllers into a merged test controller where the merged test controller has a 1-hot coded state assignment. This chapter shows that the 1-hot coded merged controller can also be incorporated in IEEE 1149.1 boundary scan standard compliant architectures.

Chapter 9 presents a description of the test controller synthesis system CONSYST including various subsystem modules. Examples of various circuits processed by the controller synthesis system are also presented.

Chapter 10 concludes this dissertation with a summary and a list of future research topics.

Chapter 2

Background

2.1 Introduction

Research in designing testable chips using DFT/BIST techniques has been lopsided. Researchers have either concentrated on devising efficient DFT/BIST techniques without devoting much effort on efficient test control schemes, or have addressed the test control problem for simplistic (and inefficient) DFT/BIST techniques. Our test controller synthesis system on the other hand is geared towards efficiently (in terms of controller logic and routing area) handling an entire spectrum of DFT/BIST techniques. Past research on the synthesis of test controllers has been heavily biased towards a modular and hierarchical approach. While this approach simplifies the design problem, it ignores the global optimality of the test controller. We argue that in an automated design environment, it is possible to move away from a fully modular approach and design a test controller whose implementation is fine tuned to the control requirements of each design.

2.2 Overview of the Test Control Process

A chip is the smallest component of a system. A system (e.g., a workstation) may also be part of a larger distributed system (e.g., network of workstations). Since the control of test hardware of a chip is initiated at higher levels of the system hierarchy we will first present a top down view of the test control process.

2.2.1 The Hierarchical Test Methodology

A hierarchy of test controllers is usually employed in a testable and maintainable system. A model of such a hierarchical test system is presented in [10]. In such a system, referred to as an *Hierarchical Test and Maintainable* (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 finally each system has a *system test and maintenance controller* (SMP). These controllers participate in all system test and maintenance activities, and communicate via test buses.

Fig. 2.1 shows part of the test hierarchy for the four levels of system hierarchy. Different buses may be used for communication at different levels. The SMP communicates with the SuMP through a Level-2 (L2) bus; a SuMP communicates with the MMCs through a Level-1 (L1) bus; and an MMC communicates with CMCs through a Level-0 (L0) bus. Bus interfaces are required for both the controlling element (master) and the controlled elements (slaves). The master and slaves can be FSMs that understand the protocol used by a bus at each level. If the SMP, SuMP and MMCs are designed to be testable chips, then they themselves should have CMCs and connection to L0 buses. In [10] the L0-master in the MMC is called a *Test Channel*. The IEEE 1149.1 [9] boundary scan bus is used as the L0-bus in [10, 11].

2.2.2 On-chip and Off-chip Distribution of Test Control Hardware

The CMC represents all the test control hardware at the chip level and the TAP [9] is an example of the L0-slave. The next question is how much of the control hardware needs to be present on-chip (and hence be apart of a CMC) and how much can be off chip (part of the MMC). In [11] a model of the possible partitions of the chip test control hardware between the CMC and the MMC is given. These partitions are shown in Fig. 2.2. In this model it is assumed that the DFT structure of an application circuit consists of one or more scan chains. It is assumed that a new test pattern is generated from the block labeled PG (pattern generator) and then shifted

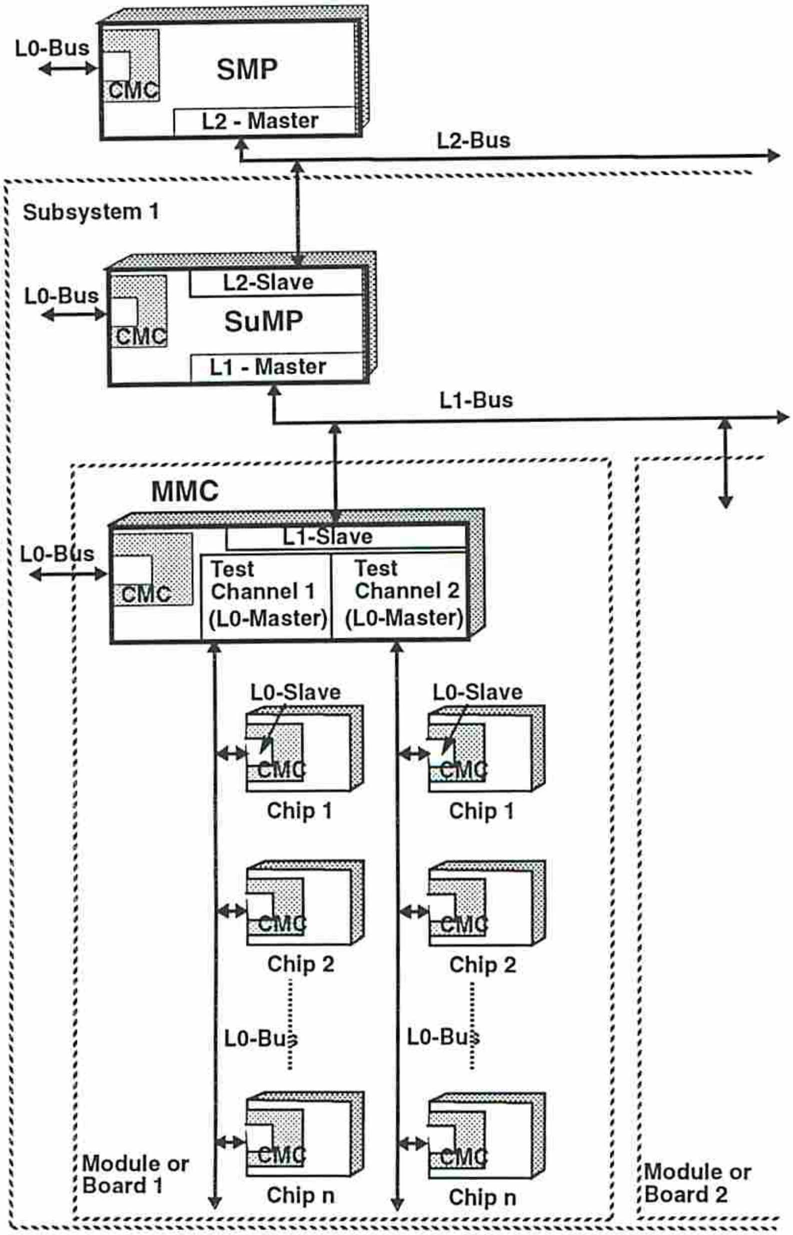


Figure 2.1: Architecture of a HTM system

in to a register that feeds a kernel. A normal mode clock is applied and then the results that are captured in register R2 are shifted out and compressed in the block labeled SA (signature analyzer). A counter SC1 and a register SC2 are used to keep track of the number of shifts required for each test vector. Test Counter (TC) is used to keep track of the total number of vectors applied to each kernel. A register SCS is used for scan chain selection and a FSM is used to control all these blocks (the control lines from the FSM to the controlled elements are shown as dotted lines).

Partition 1 puts all the test resources in the CMC. Such a CMC is capable of executing a test process on its own. *Partition 2* incorporates the seeds, correct signatures and the comparator into the MMC while leaving the rest of the resources in the CMC. *Partition 3* keeps the PG and the SA in the CMC while incorporating the rest of the control circuitry in the MMC. All control signals for the on-chip hardware must be derived from the L0-bus. *Partition 4* puts all the test control resources in the MMC and the CMC reduces to a L0-bus slave interface.

The control partition model presented in Fig. 2.2 provides a clear picture of the various choices that exist in implementing the test control hardware. Choice of a control partition affects the complexity of the on-chip test controller.

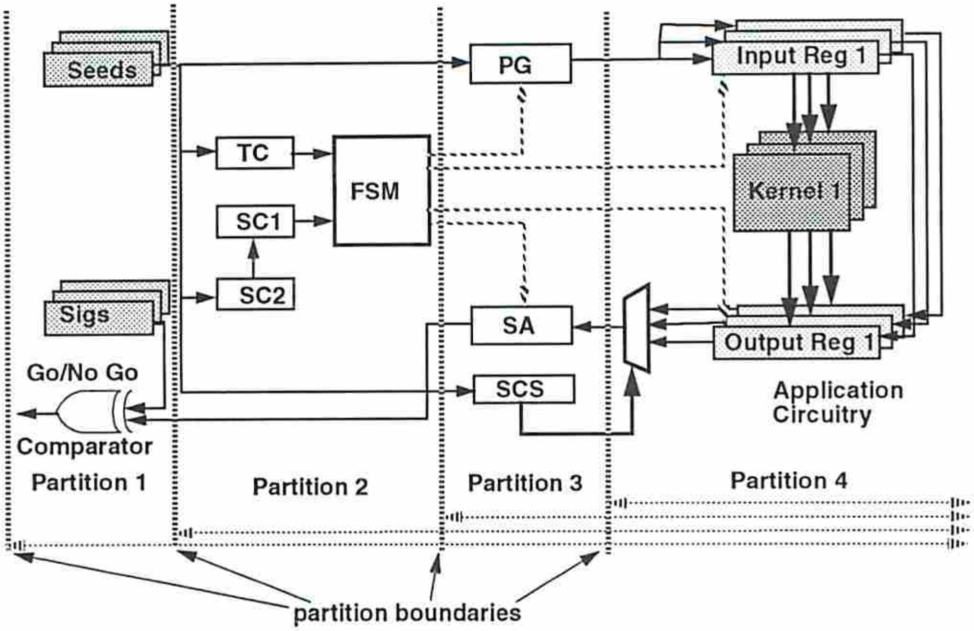


Figure 2.2: Partitioning chip test control hardware

2.3 Chip-level Test Control

2.3.1 Test Control Architectures and Control Schemes

In this section we will present various issues and approaches addressed by past researchers dealing with the on-chip test control hardware.

2.3.1.1 Test Control Architectures

In [12] Riessen *et al.* present the design and implementation of a hierarchical testable architecture using boundary scan. The design has a number of self testable *macros*. Macros are blocks of logic that are self contained as far as testing is concerned, i.e., they have their own pattern generators and signature analyzers and test resources are not shared between different blocks. The macros are isolated from one another in the test mode by registers called Test Interface Elements (TIEs) that are transparent in normal mode. Test processors (controllers) are present at the macro and at the chip level. Standardized interfaces are used to communicate between the chip level controller and the macro level controllers. For BIST testable macros, the TIEs consist of BILBO type registers and extra flip-flops that determine the mode of operation of the register. These extra flip-flops are initialized before the start of a macro test. The macro level test processor thus has inputs from the chip level test processor and the mode control bits and controls the various control lines associated with the BILBO type registers.

Beenker *et al.* [13] deal with the synthesis of a hierarchical test controller. The design is also partitioned into self testable macros. Each macro has a local controller called a test control block (TCB). A chip level test controller controls all macro level TCBs. To keep the synthesis task simple, standard interfaces are used for communication between the TCBs.

Haberl and Korpf [14] have built a system called HIST for automatically embedding self-test hardware and control circuitry in hierarchically designed circuits. Distributed self-test controllers are inserted in each functional module. These modules are called *base modules* and the self-test circuitry for base modules is not shared with any other module. A *compound module* is composed of several base modules and has a test controller controlling the base module test controllers. The controller for the compound module is controlled by the TAP controller. Standard interfaces are used for communicating between the controllers.

In [15] Zorian presents a distributed BIST control scheme for VLSI devices. The design is partitioned into self testable blocks of logic. A large number of these blocks are memory elements such as RAMs, ROMs and register files. The test operation of each self testable block is controlled by a *BIST Resource Controller* (BRC). BRCs are implemented as FSMs and are customized to implement the test process (similar to test plans) of each testable block. A BIST control network carries control information from the TAP to the distributed BRCs. Interfaces between the BRCs and the control network is done through *Scheduled BIST Resource Interface Controllers* (SBRICs). An SBRIC is also implemented as a FSM.

A number of other papers also deal with the hierarchical self-test concept [16, 17]. The ideas presented in these papers are very similar to one another. All the above techniques suffer from the following drawbacks.

- The self containment of the macros or base modules precludes the possibility of sharing test resources between various modules. This leads to high area overhead in terms of pattern generators and signature analyzers.

- Very few test methodologies are supported and the resource allocation mechanism for the modules are antiquated compared to currently available techniques [18].

- The use of dedicated test controllers for each module does not allow logic sharing between test controllers and the use of standard interfaces does not allow optimization in terms of control line routing or logic area.

2.3.1.2 Test Control Line Distribution

The optimality of test control line distribution has been investigated by Beausang and Albicki [19]. An algorithm has been presented which, given a set of BILBO registers for testing a chip and the test sessions, determines the minimum number of control lines that must be distributed throughout the chip. This approach ignores the implementation cost of the test controller.

2.3.1.3 Specialized Test Control Schemes

Breuer *et al.* [20] present control graphs for common built-in test (BIT) structures and hardwired and microprogrammed implementation of these control graphs. Designs for activating multiple BIT structures are also presented. I-paths are not considered and the area overhead is computed in terms of flip-flops. The controller designs are "template" based rather than bottom-up, i.e., logic synthesis tools are not used. Moreover, the problem of incorporating the BIT controllers in a boundary scan environment is not addressed.

2.3.2 Merging Test Controllers

The problem of merging test controllers to reduce controller area overhead has been discussed by Marinissen [21, 22]. A design is partitioned into self-testable macros. The test application for each of these macros is controlled by a Moore type FSM controller called the *Test Control Block (TCB)*. The TCBs are merged into one controller to reduce the area overhead incurred in implementing each of the TCBs separately. The merging procedure starts with two states of two TCBs and checks the compatibility of the outputs. If outputs are incompatible, then an attempt is made to make them compatible by inserting dummy states in one of the TCBs. The algorithm proceeds from those two starting states and forms a merge-tree of compatible states (and dummies). When it fails to merge any more states it has obtained a *maximal mergeable subgraph (merge-tree)*, and then starts with a new pair of starting states and continues to build merge-trees. Costs are assigned to each merge-tree in terms of dummies, "free arcs" and profits in terms of states merged. A heuristic then finds the minimal set of merge trees that cover the original pair of TCBs. All pairs of TCBs are merged and a greedy heuristic is used to merge all the TCBs. The minimization algorithm is part of the SPHINX test tools in the PYRAMID [13] silicon compiler. The main drawback of this approach is that it only focuses on minimizing the number of states and completely ignores the impact of a minimal realization on the eventual implementation cost, either for two-level or multi-level logic implementation. Moreover, results are presented only for two example circuits.

Agrawal and Cheng [23] use a heuristic for merging a test machine with a functional machine. The test machine is used for testing the functional machine. The test machine has the same number of states as the functional machine and can be set to any of its states and all outputs can be observed by short predetermined input sequences. The machines are merged such that a minimal number of edges are added to the functional machine. However, a procedure has not been presented for the merge process nor has any justification been provided for using this heuristic.

Smith and Kohavi [24] describe a technique for synthesizing one machine from two synchronous state machines with the same input I and outputs Z_1 and Z_2. First a composite machine M_c is found from the two initial machines M_1 and M_2. The composite machine is next examined to determine the largest non-trivial common factor machine. The existence of such a common factor machine was (in an earlier paper) shown to depend on the existence of equivalent implication graphs having disjoint node subsets. If the implication graphs have overlapping node subsets, then a common machine can be found by node-splitting. Once a common factor machine is found, it is factored out of each of the original machines and two smaller machines are appended to the common factor machine. Each of these machines then produce the required outputs Z_1 and Z_2 corresponding to the original machines. The main disadvantage of this approach is that state-splitting might increase the total number of states in the composite machine.

The classical approach to merging a number of FSMs is to obtain their Cartesian product and then minimize the number of states in the product machine (state minimization). Some results that deal with exact techniques for state minimization of incompletely specified sequential machines can be found in [25, 26, 27]. STAMINA [28] is one of the better known software packages that uses heuristic techniques for state minimization. A number of state minimal machines with different implementation costs exist for the same product machine, but current state minimization tools (e.g., STAMINA) are unable to automatically choose a machine with the minimal implementation cost.

2.4 Testability Buses and Boundary Scan Architectures

To achieve an acceptable degree of testability and diagnosability, a system must be testable at every level of integration. The design of testable chips, boards, subsystems and systems that can be connected to standard test buses has thus drawn much attention. Three major standards and proposals exist for controlling and accessing DFT/BIST hardware on chips, boards, subsystems or systems. One of these is the IEEE Boundary Scan Standard (IEEE 1149.1) [9], also referred to as *Minimum Serial Digital Subset (MSDS)*. The other two are currently proposals and will probably be converted into standards in the near future. They are the *Extended Serial Digital Subset (ESDS)*(IEEE 1149.2) and the *Standard Backplane Module Test and Maintenance Bus Protocol* (IEEE 1149.5).

2.4.1 The IEEE 1149.1 Boundary Scan Architecture

The IEEE 1149.1 boundary scan architecture of a chip is shown in Fig. 2.3(a). The shaded area labeled *app. circuit* is a circuit which may have some DFT/BIST hardware. The unshaded area is the *Test Access Port* (TAP). The TAP consists of a *TAP controller* (TAPC), an *instruction register* (IR), a *boundary scan register* (BSR), a one-bit *bypass register* (BYPR), an optional *device identification register*, output buffer and multiplexers. The boundary scan register is made up of 1-bit boundary scan cells. These cells may have different implementations depending on whether they serve input, output or bidirectional pins. These cells may also have additional functions to test application circuitry. Based on their functionality, these boundary scan cells may be grouped together into registers, which in turn constitute the boundary scan register. We refer to these constituent registers simply as boundary scan registers or BSRs. The TAP has four pins, TCK (test clock input), TMS (test mode select input), TDI (test data input), and TDO (test data output). The TRST (test reset input) is optional.

The TAPC is a 16 state FSM that operates under the control of the TMS signal (Fig. 2.3(b)). There are two outgoing edges from every state in the TAPC corresponding to the 0 and 1 values for TMS. The *test-logic reset* (tlr) state is entered whenever the TAPC is reset, either synchronously (using TMS) or asynchronously (using TRST). The *run-test/idle* (rti) state is entered when the chip is in the self-test mode, or is in test mode with no ongoing test activity. In the TAPC, 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 in the *UpdateIR* state. The IR contents (instructions) together with the TAPC states determine various on-chip test operations. One major function of an instruction in the IR is to select a data register for scanning. When transmitting data, both the states *CaptureDR* and *UpdateDR* 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 data register (DR).

By suitably controlling TMS, a board (module) level test controller (such as the *Module Maintenance Controller* [10]) can send both instructions and data to and receive results from a chip. Instructions EXTEST, SAMPLE and BYPASS are mandatory. These instructions together with the boundary scan hardware allow interconnects to be tested, signal values at the input/output pins to be sampled in a functionally operational chip (on-line monitoring), and bypass the boundary scan registers on one or more chips.

2.4.2 The IEEE 1149.2 Boundary Scan Proposal

The IEEE 1149.2 is also intended to be implemented at the chip level. This proposal offers test features similar to the IEEE 1149.1 but has some significant differences. First, unlike the IEEE 1149.1 standard where the boundary scan cells cannot be shared with functional registers, the IEEE 1149.2 proposal allows boundary-scan register cells to be shared with the functional logic of the chip and does not require the cells to have separate serial-shift and parallel-update stages. The IEEE 1149.2 boundary scan registers are called the *I/O registers*. Second, the IEEE 1149.2 uses a direct, parallel access method to enable different test operations. Thus the TAP controller and the IR are not needed. This proposal mandates the use of a minimum of two signals to specify the operation modes of a chip.

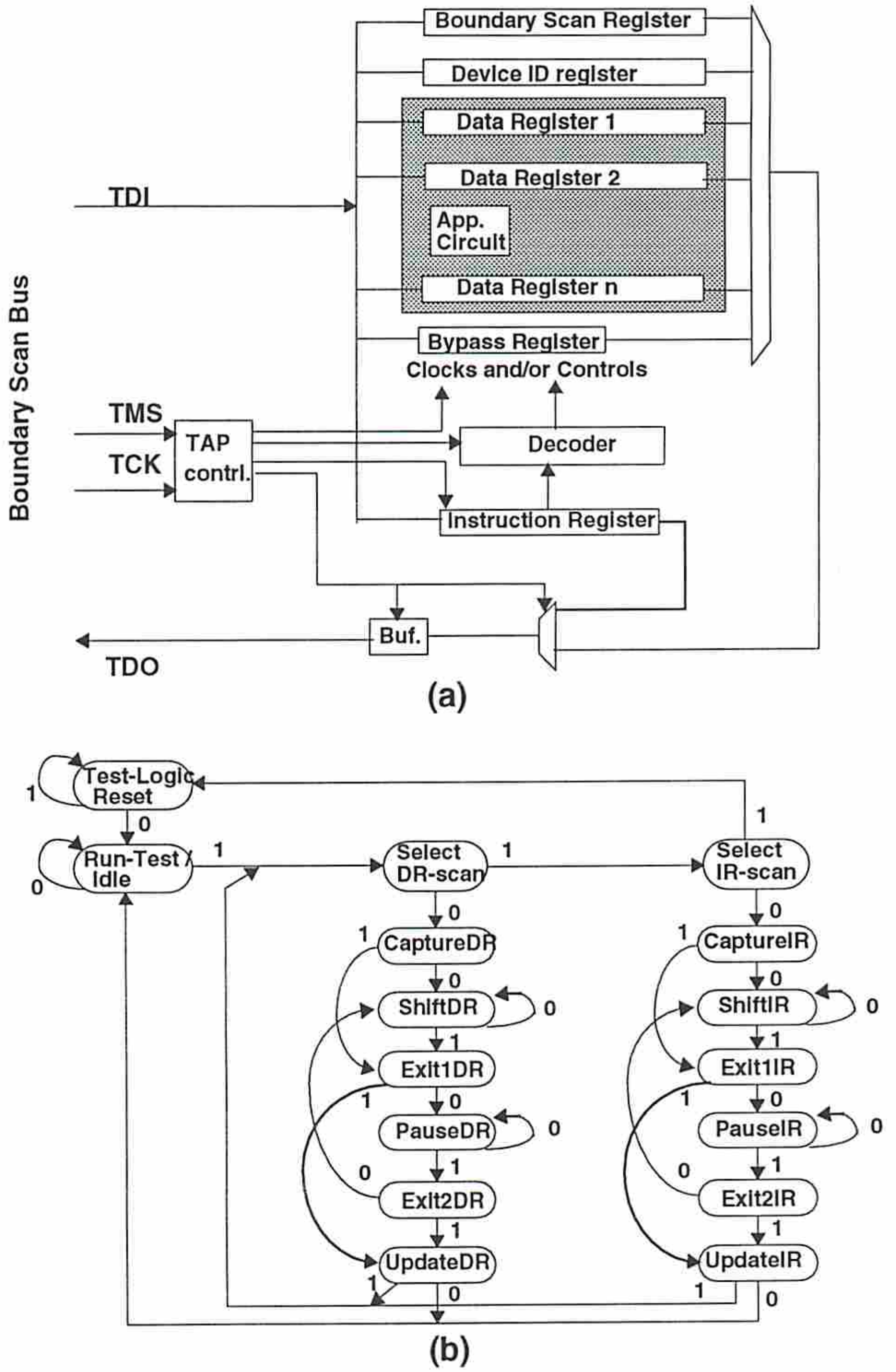


Figure 2.3: (a) IEEE 1149.1 Boundary Scan Architecture; (b) TAP controller state diagram

The IEEE 1149.2 architecture has four major elements: the I/O register, Implementation Detail Register (IDR), Bypass Register (BYPR), and the Scan Access Port (SAP). The test bus consists of TDI, TDO, TCK and a minimum of two mode control wires, STM0 and STM1. The IDR provides a mechanism for accessing some of the implementation specific details. The IEEE 1149.2 has the following advantages over the IEEE 1149.1. Test logic area overhead is reduced by (1) sharing functional and boundary scan registers, and by (2) eliminating the TAP controller and the IR. Additionally, (3) since the dedicated boundary scan registers are not used, interconnect testing can be performed at-speed. The drawbacks are (1) more pins are required, and (2) on-line monitoring of signals at chip boundary is not possible.

2.4.3 Boundary Scan Bus Masters

The Test Channel implemented by Lien [29] is an example of a IEEE 1149.1 boundary scan bus master. A bus master has also been developed by Jarwala and Yau [30].

2.5 Summary

In this chapter we first presented an overview of the hierarchical test methodology and on-chip/off-chip distribution of test control hardware. Then past research on various chip-level test control problems was summarized. Finally a brief description of test buses and boundary scan architectures was provided.

Chapter 3

Test Control Architecture

3.1 Introduction

In the previous two chapters we have provided an overview of the test control problem and past research in this area. However specifics of the hardware structures and/or the test methodologies that need to be controlled or supported by a test control/support system have not been provided so far. In this chapter we first present a detailed description of the TDMs and hardware structures that our test controller synthesis system needs to support. Then we describe the test control functions that we have decided to incorporate on a chip. Essentially a partition between on-chip and off-chip test control functions has been made and we will present the underlying motivation for such a partition. Next we will present our on-chip test control architecture in which certain control functions are distributed throughout the chip, and other functions are centralized. Finally we will describe various mechanisms for controlling scan and BIST registers.

3.2 Functions of the On-chip Controller

The first step in solving the test control problem is to identify the nature of the on-chip test hardware and/or methodologies that need support from test controller(s). The following on-chip test hardware/methodologies need to be supported.

- TAP controller and boundary scan registers.
- The TDMs incorporated in a circuit by the scan system SIESTA [31].

- The TDMs incorporated in a circuit by the BIST system BITS [18].

The IEEE 1149.1 boundary scan standard is being increasingly adopted by different ASIC and custom processor vendors and is fast becoming an industry standard. The simplicity of the interface (4 wires) makes it an attractive choice for the L0-bus. Therefore it is imperative that we also support this standard. Adopting this standard implies that our L0-slave interface is fixed and will be handled by the TAP controller.

3.2.1 Scan TDMs Requiring Support

SIESTA is an integrated scan design system that can incorporate various scan TDMs in a circuit. Fig. 3.1 is a directed acyclic graph that represents a classification of the scan based DFT techniques that are currently supported by SIESTA or will be supported in future. This classification highlights the wide variation of choices that are available for scan designs, e.g., full scan vs. partial scan. In this section we will provide a brief overview of different scan TDMs.

Partial scan [32, 33, 34] is a DFT methodology in which only a subset of the flip-flops in the design are converted to scan registers. Cheng and Agrawal [32] represent a circuit as a graph where the flip-flops are the nodes of the graph and interconnections between flip-flops (through combinational logic) are the edges. A heuristic is used to select the minimal number of flops (nodes) to make the resulting circuit graph acyclic. Self-loops, i.e., the case where the output of a flip-flop feeds back to the input (possibly through combinational logic), are not broken. The selected scan flops now become additional primary inputs (PIs) and primary outputs (POs) during the scan mode of operation. Making a sequential circuit acyclic reduces the complexity of the sequential test generation problem.

In the BALLAST [34] methodology, Gupta *et al.* add an extra constraint to the partial scan approach. Scan flip-flops are selected such that not only should the resulting circuit be acyclic but the circuit should also be *balanced.* In a balanced circuit a pair of paths between two combinational logic blocks should go through the same number of flip-flops (registers). SIESTA also has the option of only creating acyclic circuits without balancing them (ACYST methodology [35]) . These are called *unbalanced* acyclic partial scan circuits. The CYCLIST methodology in

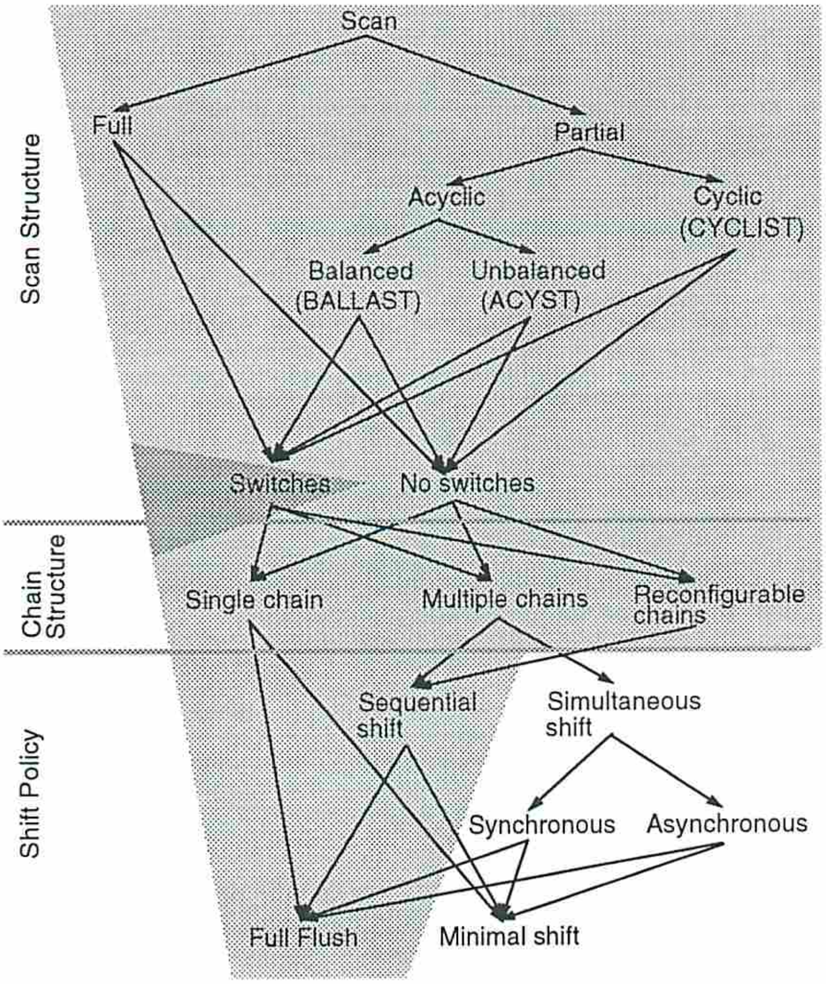


Figure 3.1: Classification of scan DFT techniques

SIESTA creates partial scan circuits where all loops are not broken and hence the circuit is cyclic. The application of a test vector to a kernel may or may not take advantage of switching structures such as multiplexers. Currently SIESTA does not support the switch option.

Test application time is a major concern for scan based DFT techniques. In a scan design all scannable flip-flops can either be connected together in one scan chain (*single chain*) or configured as a number of scan chains (*multiple chains*, *reconfigurable chains*). These correspond to different *chain structures* or chain configurations. The multiple chains can either be loaded simultaneously (*simultaneous shift*) or one after the other (*sequential shift*). The simultaneous shift option is only viable if multiple shift data input (SDI) and shift data output (SDO) pins are available on the chip. Since the IEEE 1149.1 standard does not support multiple SDIs and SDOs, we will not consider the simultaneous shift option for multiple chains. However chains that are shifted one after the other or reconfigured between shifts (*reconfigurable chains*) will be supported.

A shift chain can either be shifted for a number of clock cycles equal to the number of flip-flops in it (*full flush*) or may be shifted for only part of its length (*minimal shift*). Details about the chain structures and the shift policy can be found in [36]. Each path from the root to the leaves of the classification DAG in Fig. 3.1 describes various aspects of a scan TDM embedding. The shaded region corresponds to the various scan options that we have studied and will support.

SIESTA uses deterministic test generators to generate tests for combinational and sequential kernels. Due to the large volume of test data for scan designs, it is impractical to store test data on-chip. Therefore test vectors/expected responses are stored off chip and mechanisms are provided to shift vectors into and results out of a chip.

3.2.2 BIST TDMs Requiring Support

BITS is an integrated BIST system that supports a number of BILBO oriented methodologies. Apart from BILBO and EXTBILBO, BITS supports *concurrent* BILBO (CBILBO) [37], *combined* BILBO (COMBILBO) [38] and the *balanced* BILBO (BIBS) [39]. The CBILBO TDM uses BILBO registers that consist of back-to-back registers, one acting as a PG and the other as a SA. This TDM is often used in circuits with *self-loops*, where a self loop consists of combinational logic block that feeds itself through a single register. In the COMBILBO TDM, a test register acts as a SA for some kernel feeding it. The outputs of this register are used as test patterns for another kernel. Analytical and experimental results presented in [38] showed that after an initialization period, the characteristics of the SA register are similar to a random pattern generator.

In [39] Lin *et al.* have proposed a low cost BIST methodology that employs *balanced BISTable structures* as kernels. A kernel is balanced BISTable if (1) it does not contain any cycles, (2) two paths between a pair of combinational logic blocks go through the same number of registers, and (3) the registers feeding the inputs of a kernel and the registers fed by the outputs of a kernel are distinct.

For all the BIST TDMs the PG and the SA need to be initialized to known states. Usually the PG need only be initialized to a known *non-zero* state and the SA to a known state. This can be accomplished by adding specific hardware features to the PG and the SA. However we assume that the registers will be initialized by shifting in seeds from off-chip.

The following is our rationale for storing seeds/expected responses off-chip.

- Dedicated ROMs are needed to store seeds and expected responses on-chip and this may lead to an unacceptably high area overhead. If registers are not provided with the shift capability then dedicated buses are needed to load seeds to registers from a ROM. The on-chip comparator also adds to the area overhead. To reduce the cost of an on-chip comparator, a serial comparator may be employed. To use a serial comparator, all SAs need the shift capability and should be part of a scan chain. For registers that have the PG capability, the shift mode comes for free. Since a large percentage of BILBO registers in a BIST design have the PG capability, there is little hardware overhead in providing the shift capability to *all* BILBO registers. Signatures and seeds then can be stored off-chip. This leads to significant savings in hardware for seed/signature storage, distribution and collection.

- For random testing, Lempel *et al.,* [40], propose a scheme whereby the test vectors needed for a set of hard-to detect faults are automatically generated by a PG. They determine the smallest sequence generated by a PG that contains all these test vectors. This entails initializing the PG to a non-trivial initial seed. Moreover, to shorten test application time, the set of test vectors may even be embedded in disjoint sequences of a PG requiring multiple-seeds. Hardcoding multiple seeds into a PG or storing multiple seeds in a ROM will result in a high area overhead. An easier solution is to store seeds off-chip and shift them in as needed.

- Memory structures constitute a large percentage of modern VLSI chips. BIST approaches to testing memory structures are gaining popularity. In most cases, registers on the inputs/outputs of memory structures are modified into specialized PGs/SAs. Since the memory test algorithms have several stages, a complex test controller is usually employed that loads PGs/SAs with specific seeds (stored on-chip) and controls each phase of the test process. The seeds can be stored off-chip and the memory test can be partitioned into several sessions. This greatly simplifies the complexity of the test controller as well as saves on the hardware required for storing multiple seeds on-chip.

- To reduce the aliasing problem SAs may need to be initialized to a non-trivial seed. This initialization is easy if seeds are shifted in.

3.3 The On-chip Test Controller

Recall that in Chapter 2 a model was presented in Fig. 2.2 that illustrated various partitions between the on-chip and the off-chip controller. Our control partition is shown in Fig. 3.2. This partition shows that the seeds (test vectors) and the expected signatures (responses) are stored off chip. The original FSM is partitioned into two, one on-chip and the other off-chip. The on-chip FSM represents the TAP controller (and additional decoding circuitry) while the off-chip FSM corresponds to the test channel in a MMC.

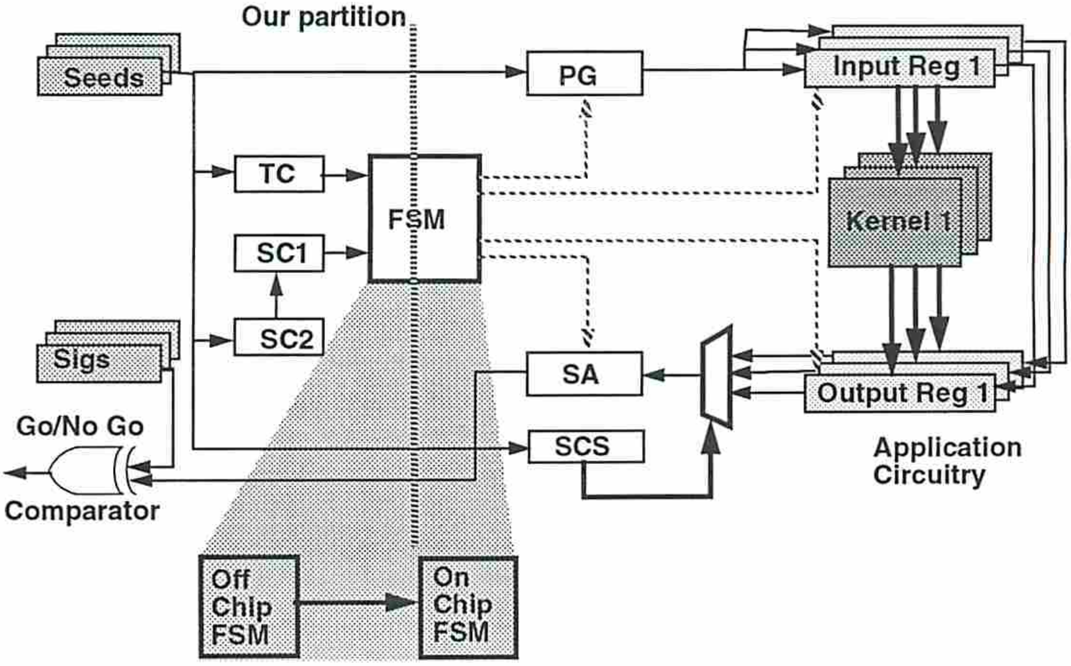


Figure 3.2: Our control partition

3.3.1 The Partially Distributed Test Control Architecture

One method of controlling all the on-chip circuitry is to control all the control lines directly from the FSM shown in Fig. 3.2. We will show in the next chapter that this control mechanism also referred to as *Direct Control* often leads to a large number of control lines. In the worst case the number of control lines are directly related to the number of scan chains for scan designs or the number of BIST registers in BIST designs.

Fig. 3.3 shows the model of our partially distributed architecture. This is a *partially* distributed architecture because while some control circuitry is distributed throughout the design, other control circuitry that could have been implemented separately is combined together. We assume that a chip is partitioned into *datapath* and *control* sections. To comply with the IEEE 1149.1 boundary scan standard boundary scan registers are inserted at the PIs and POs of the circuit. The datapath and control partitions are processed separately by SIESTA or BITS and various TDMs are embedded to make the circuit testable. As a result of these embeddings a subset of the registers are converted into scan registers. Note that the boundary scan registers are always scannable. In Fig. 3.3, *BSR1* and *BSR2* are boundary scan registers at the inputs and outputs of the datapath respectively. *R1* represents a functional register in the datapath that is made scannable. Scannable registers correspond to simple scan registers for scan TDMs or BILBO registers for BIST TDMs. All scannable registers have local controllers.

Instead of individual control lines emanating from the FSM as shown in Fig. 3.2, a test bus is routed to all test hardware. Examples of test resources are all scannable registers. The scannable registers in turn have local controllers that decode information from the internal test bus to control the registers. These controllers can be combinational or sequential. Sequential controllers contain *mode* and/or *configuration* flip-flops and combinational logic (decoders and possibly multiplexers). More details about these flip-flops will be presented Section 3.3.2.

I-paths need to be set up in the datapath to apply (collect) test vectors (results) to (from) kernels. The test data (results) are transported from (to) test registers. Test registers can be simple scan registers or they can have PG and/or SA capability. The test data/result transport paths are set up by controlling multiplexers, buses and functional control units. The control for all these paths are combined together into one merged test controller called the *internal test controller*. Test registers that act as PGs (SAs) in test-mode may need to generate (compact) a new pattern (result) every t clock cycles and hold the previously generated (compacted) pattern in between. In this case, the internal test controller has control lines to the local controllers to hold registers t clock cycles between each step of pattern generation (signature compaction).

The internal test controller is either implemented separately or merged (partially or fully) with the functional controller. The FSM driving the test bus performs the functions of the TAP controller. Different encoding schemes are used to broadcast control information over the test bus. These encoding schemes minimize a two-level implementation of the integrated TAP controller and/or the local decoders (controllers).

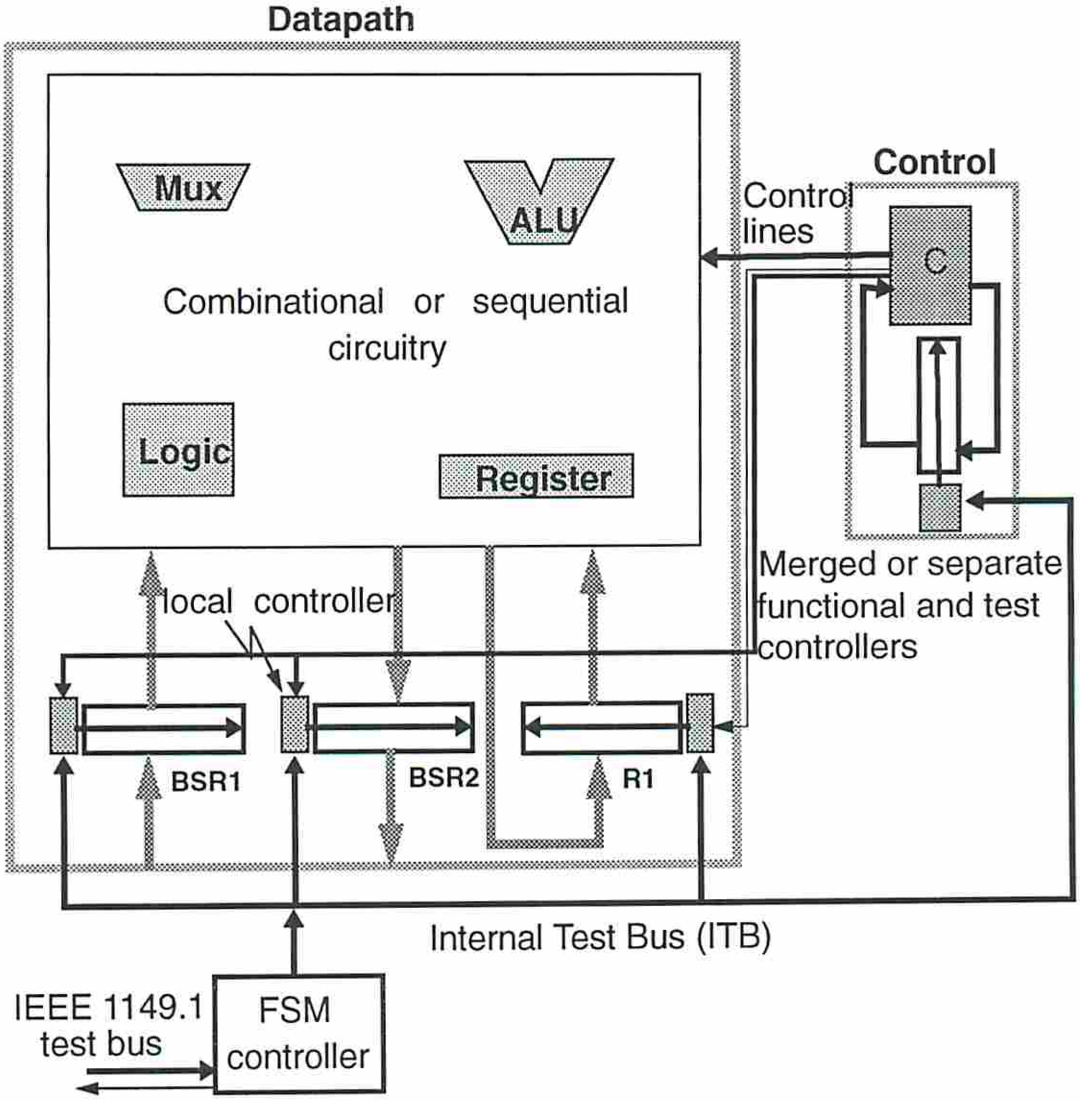


Figure 3.3: The partially distributed test control architecture

3.3.2 Control Models

Fig. 3.4(a) shows the relevant details of the distributed control architecture. In this figure registers $Reg\,1, Reg\,2, \ldots, Reg\,n$ represent n scannable registers in the datapath. $LC_1, LC_2, \ldots, LC_n$ represent the local controllers associated with these registers. The internal test bus is an input to all these local controllers. The information transmitted on the internal bus is a function of the set of instructions (for a particular design) and certain states of the FSM driving the bus. The exact nature of the information transmitted on the bus will be described in detail in the next chapter (Chapter 4).

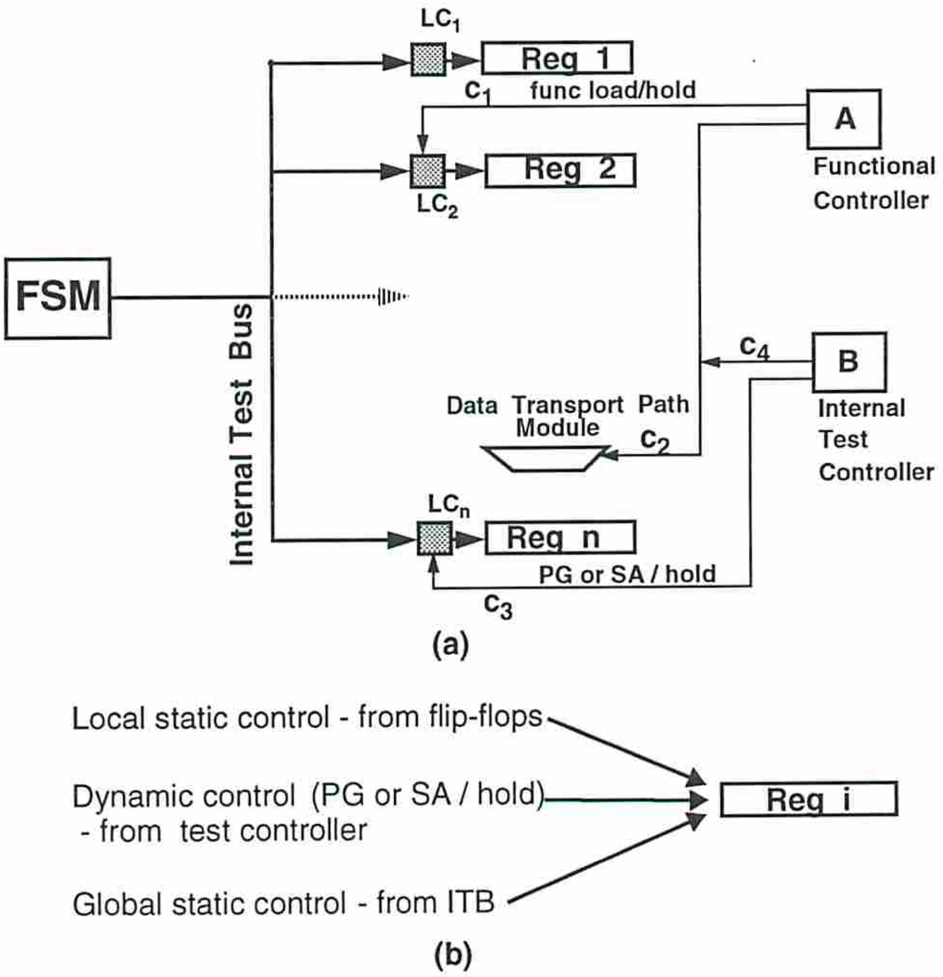


Figure 3.4: Simplified control model

Some of the local controllers have inputs from other sources as well. For example, a register may operate as a load/hold register in the functional mode of operation.

In that case, there is a load/hold control line to the local controller driven from a functional controller. Note that the functional controller can be vacuous (i.e., control lines driven directly from the PIs), combinational or sequential. The functional controller also controls various modules such as multiplexers, ALUs and other logic blocks in the datapath. In Fig. 3.4(a) a multiplexer in the datapath is shown being controlled from the functional controller, represented by A. If this multiplexer is also used to set up I-paths for testing datapath modules, then this control line also needs to be controlled by a test controller (represented by B).

Consider *Reg n* in Fig. 3.4(a) and assume that this register is part of a BIST design and is employed as a PG in one session and as a SA in another. We need some mechanism to control these modes of operation in different sessions. In our test control architecture this is accomplished by using a flip-flop (*mode* flip-flop) that resides inside the local controller and is part of the scan chain. This flip-flop is initialized when the register is seeded and the value in this flip-flop now determines the mode of operation of the register. The test plan for testing a kernel using this register may be such that this register may need to hold for a number of clock cycles between successive activations of the PG or SA mode. In that case an external signal is provided to the local controller that is driven from the test controller. This signal is called *PG or SA/hold.*

Fig. 3.4(b) models all the control sources to a register. A register can have at most three control sources. These are

1. Global static control from the internal test bus.

2. Local static control from flip-flops.

3. Dynamic control from the test controller.

The control from the internal test bus is *global* because the test bus input is common to all scannable registers. This control is also *static* because the bus state does not change during a particular instruction and a particular TAP controller state. The control from the flip-flops is *local* and *static* because the flip-flops are part of the local controllers and their state does not change during the test mode of operation (when there is an instruction corresponding to RUNBIST and the TAP controller is in the *run-test/idle* state). The *PG or SA/hold* signal from the test controller is called *dynamic* because the control value changes during the test mode of operation.

Examples of various types of local controller for registers in scan or BIST designs will be presented next.

3.3.3 Example Local Controllers

3.3.3.1 Scan Register Controller

Fig. 3.5(a) is an example of a circuit tested using the full scan TDM. The circuit consists of registers R_1, R_2 and R_3 and kernels k_1 and k_2. The three registers are configured into two scan chains, Chain 1 consists of all three registers and Chain 2 consists of R_2 and R_3. The two scan chains are used to first apply a number of test vectors to both the kernels and then Chain 2 is used to apply the remaining set of vectors to kernel k_2. Fig. 3.5(b) shows the local controller corresponding to register R_1. R_1 has three functions, shift, hold and load. This register is completely controlled by the internal test bus and a functional load/hold control line (if present). The test bus transfers control to the functional load/hold line in the functional mode. A chip is in the functional mode when BYPASS and SAMPLE are loaded in the IR.

The local controller for R_2 is shown in Fig. 3.5(c). Note a multiplexer is used to reconfigure the two scan chains. In our control approach we assume that the reconfigurable multiplexers are part of the local controllers of certain registers. In this case the 2-1 multiplexer is part of the local controller of R_2. The decoder in the local controller of R_2 controls the multiplexer select line based on the instruction information transmitted over the test bus.

The simplified control model for both registers R_1 and R_2 is given in Fig. 3.5(d). This model shows that for both the registers there is only *global static* control from the internal test bus.

3.3.3.2 BIST Register Controllers

Fig. 3.6(a) is an example of a circuit tested using the BILBO TDM. In this example registers R_1 and R_3 act as PG and SA in the test mode, respectively. Register R_2 operates as a PG and as a SA in different sessions. k_1 and k_2 are the kernels tested.

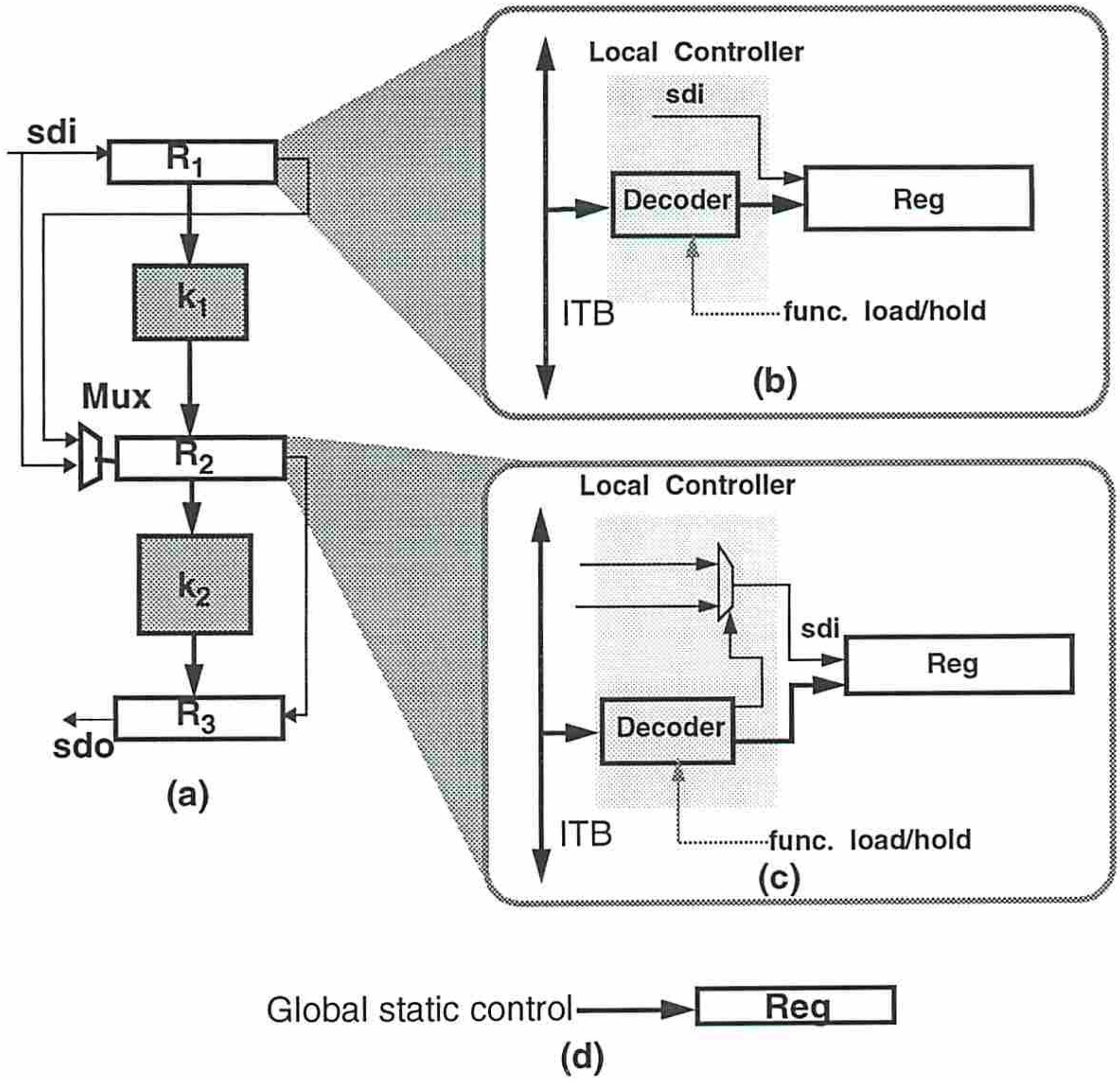


Figure 3.5: Examples of local controllers for scan designs

The three registers R_1, R_2 and R_3 are configured in a single scan chain. The local controller for R_1 is shown in Fig. 3.6(b). This controller consists of a simple decoder and a 2-1 multiplexer. The multiplexer is used to select between the scan chain and the output of a set of Ex-or gates that implement the feedback polynomial for the pattern generator. The Ex-or gates are fed by certain bits of R_1 (represented by Q in the figure). The multiplexer select line is controlled by the decoder. The internal test bus provides *global static* control. R_1 acts as a PG when an instruction corresponding to RUNBIST [9] is loaded in the instruction register and when the TAP controller is in the *run-test/idle* state. Control transfers to the functional load/hold line (if present) in the the functional mode (SAMPLE or BYPASS loaded in the instruction register).

Fig. 3.6(c) shows the local controller for R_2. Since this register has functions PG and SA, a mode flip-flop is needed in the local controller. This flip-flop is on the same scan chain as the register. This flip-flop has two modes of operation, scan and hold. These modes are controlled by the decoder associated the the R_2's local controller. Fig. 3.6(d) is the simplified control model for R_1. Global static control is provided by the test bus. Fig. 3.6(e) is the simplified control model for R_2, which shows that in addition to global static control from the test bus this register also has local static control from the mode flip-flop.

Fig. 3.7(a) shows an example circuit that uses the EXTBILBO TDM to test the kernel k_1. A test pattern is generated by R_1, driven on to the bus and applied to k_1. The response is loaded in R_3. The output of R_3 is then driven on to the bus and compacted in R_2. Since the bus is used in two clock periods, the minimal latency of a pipelined test application for k_1 is 2. Both R_1 and R_2 hold on alternate clock cycles. Registers R_1 and R_2 are configured in a scan chain. We will focus on the local controller for register R_2 (Fig. 3.7(b)). Since this register has PG and SA modes of operation, a mode flip-flop is needed. In addition, a *SA/hold* signal (driven from a test controller) is needed because R_2 needs to hold on alternate clock cycles during signature compaction. A multiplexer is used to select between the scan chain and the output of a set of Ex-or gates that implement the feedback polynomial for PG/SA. We assume that both the PG and SA modes employ the same polynomial. Fig. 3.7(c) is the simplified control model of R_2. This is an example where all three control sources are needed.

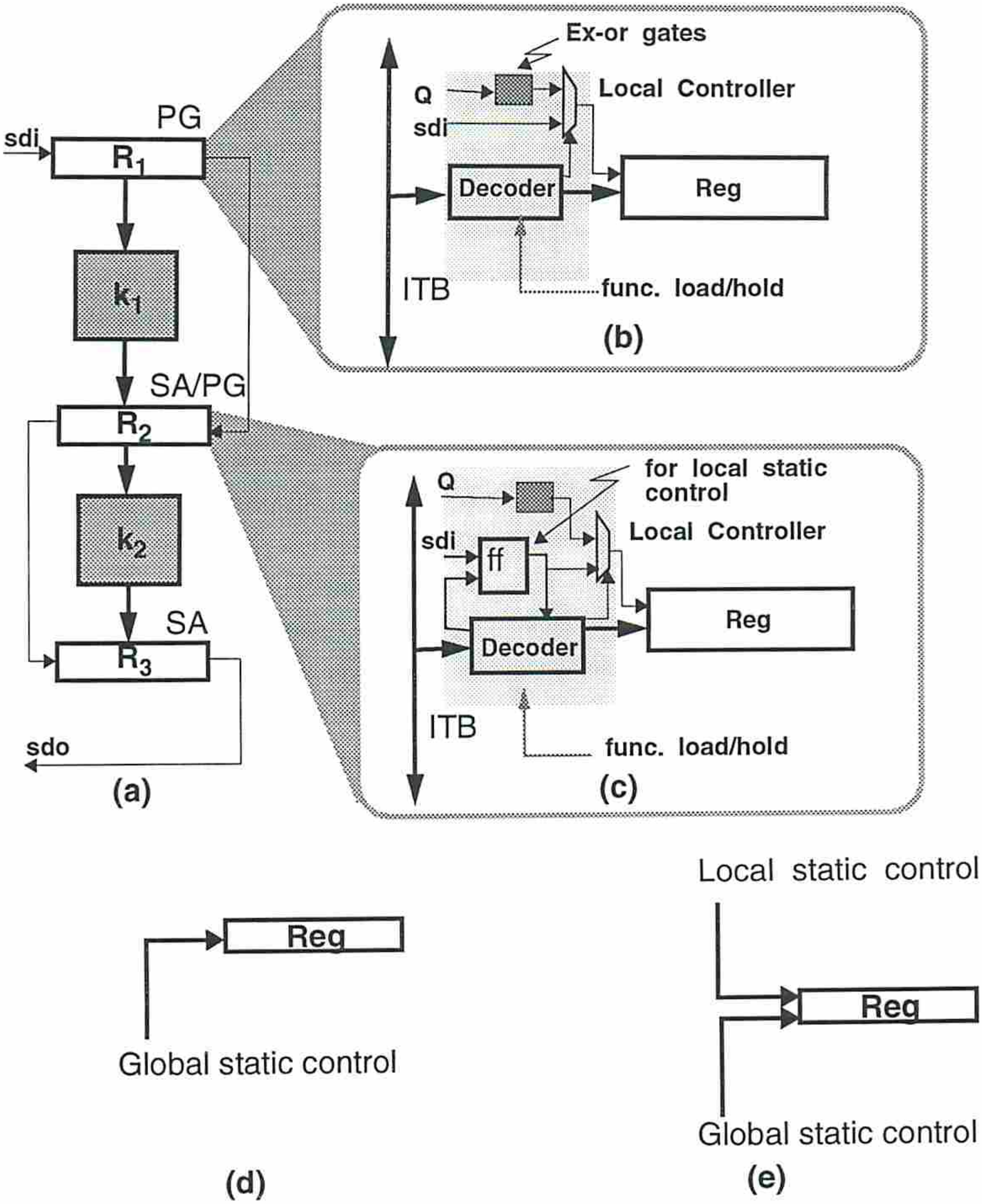


Figure 3.6: Examples of local controllers for BIST registers

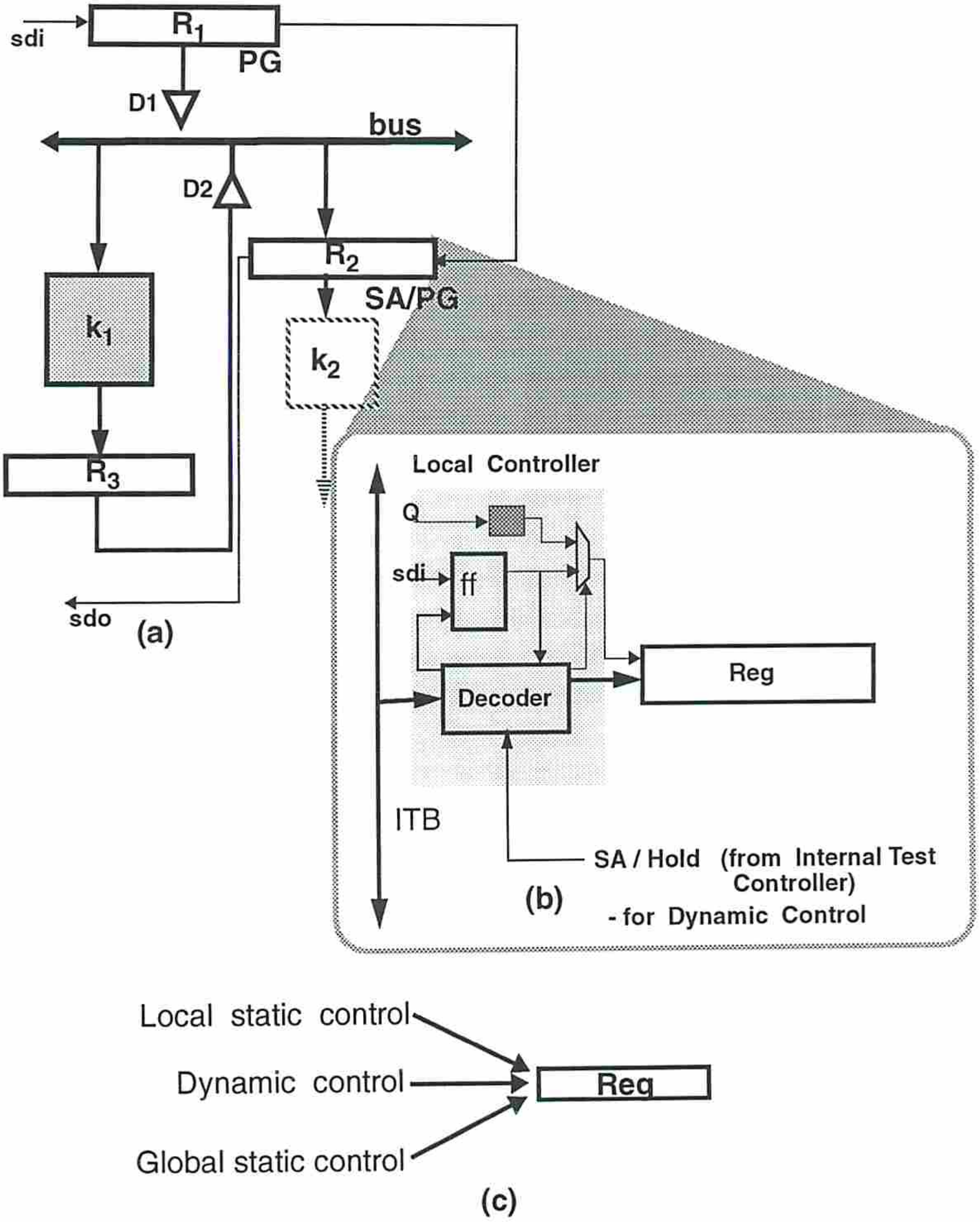


Figure 3.7: Example of a local controller for a BIST register having SA/hold control

Fig. 3.8(a) is a more complex example of a circuit tested using the EXTBILBO TDM. In this example, registers R_1 and R_2 are configured to operate as one PG for testing kernel k_2 in one session. In the other session R_1 operates as a PG on its own to test kernel k_1. The feedback polynomials for the two cases are different. R_1 also acts as a SA to test some other kernel that is not shown in the figure. Fig. 3.8(b) shows the local controller for R_1. Apart from one mode flip-flop, an additional flip-flop is needed to select between the two different polynomials. This flip-flop is called a *configuration* flip-flop. A *SA/hold* is also needed since R_1 acts as a SA and holds for alternate clock cycles. The simplified control model for R_2 is shown in Fig. 3.8(d).

3.3.4 General Control Models

The general model of the local controller for scan registers in given in Fig. 3.9(a). This figure shows that the local controller for a scannable register consists of a decoder and a n-way multiplexer. This controller has inputs from the test bus and (possibly) has a functional load/hold control line. The n-way multiplexer is used to select the relevant serial data input for a certain scan chain configuration.

The general control model of a BIST register is given in Fig. 3.9(b). This model shows the mode control and the configuration flip-flops. The decoder receives inputs from the mode and configuration flip-flops and the ITB. The decoder controls the data-select multiplexer and the BIST register. One or more of the following are the inputs to the data-select multiplexer.

1. Outputs of sets of Ex-or gates that form the feedback polynomials.

2. Serial data outputs of different registers.

3. Outputs from mode or configuration flip-flops.

3.4 Summary

In this chapter we described various scan and BIST TDMs that require test control support. Various reasons for storing seeds/test patterns/responses/signatures off

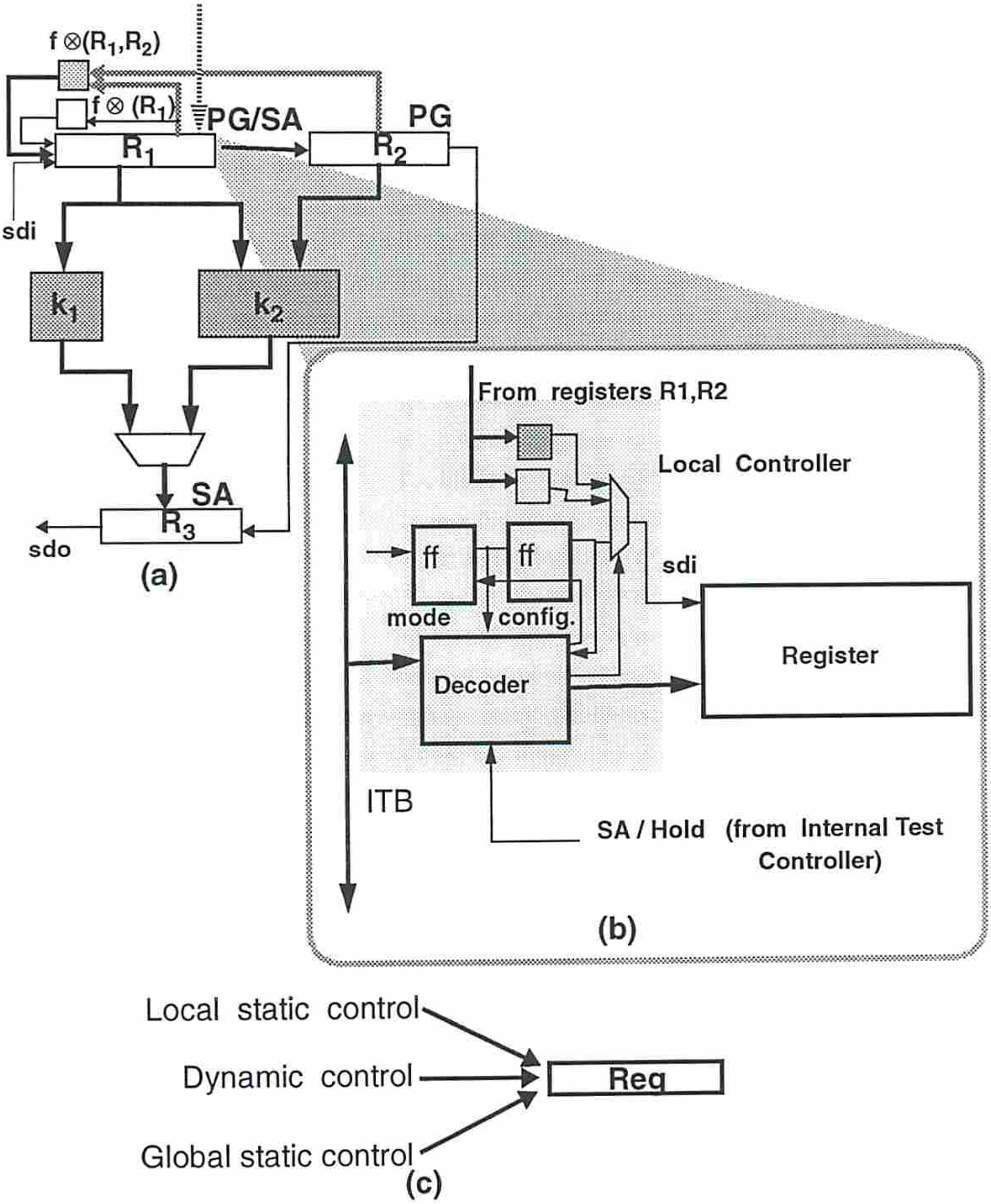


Figure 3.8: Example of a local controller for a reconfigurable BIST register

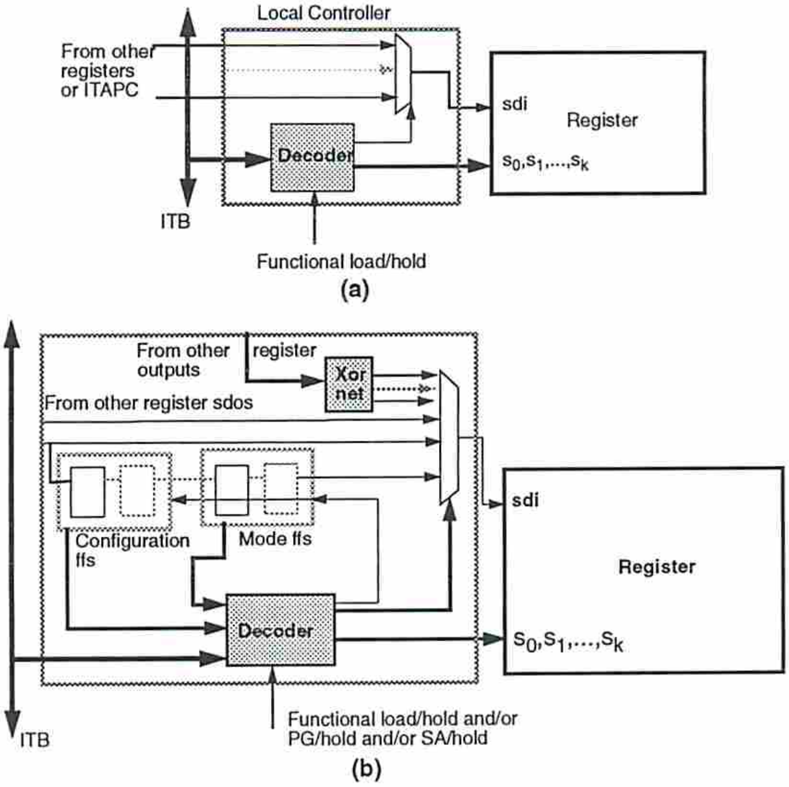


Figure 3.9: General model of local controllers for (a) scan registers; (b) BIST registers

chip were enumerated. We also presented our *partially distributed* test control architecture. Examples of specific local controllers as well as general controller models for scan and BIST registers were provided.

Chapter 4

Bus-based Control Schemes

4.1 Introduction

In Chapter 3 we introduced the *partially distributed* test control architecture. In this architecture each scannable register has a local controller that decodes information from an internal test bus (ITB). In this chapter we will describe in detail various bus styles as well as the algorithms employed to encode the information transmitted over the bus. These encoding algorithms are geared towards minimizing the implementation cost of the distributed decoders and/or the TAP controller. We will also present the implementation procedure for the decoders that are associated with the local controllers. An abridged version of this chapter is presented in [41].

This chapter is organized as follows. Section 4.2 presents a technique for integrating boundary scan on a chip. Section 4.3 analyzes the complexity of controlling on-chip test resources directly from the TAP controller. Different types of bus-based control schemes are examined in Section 4.4. Section 4.5 presents various techniques for encoding the symbols transmitted over a bus. Experimental results and a summary of this chapter are presented in Sections 4.6 and 4.7, respectively.

4.2 Integrating Boundary Scan on a Chip

In this section we first present our approach for implementing the TAP controller. We then employ an example circuit to illustrate the control requirements of various test resources in a boundary scan environment.

4.2.1 The Integrated TAP Controller

Fig. 4.1(a) is a canonical representation of the control signal generation scheme for on-chip test resources from the TAP Controller (TAPC) states and instruction register (IR) contents. In this figure, blocks *nsl, ol* and *cl* refer to next-state, output and control (decode) logic, respectively. Logic sharing among blocks is not possible since each block is implemented separately. Fig. 4.1(b) shows the block diagram of an Integrated TAP Controller (ITAPC) which consists of flip-flops (*state*) and combinational logic C_1. C_1 is responsible for generating the next state signals and the control signals for test data registers and data scan chains. Test data registers consist of the Boundary Scan Registers (BSRs) and functional registers that are used in testing application circuitry. ITAPC and TAPC have the same set of states. A state transition table (STT) [42] description of the ITAPC is created based on the control requirements of all the test resources. This STT is provided as input to sequential synthesis tools to perform state assignment and logic minimization [43]. The state assignment tool encodes the ITAPC states to minimize the implementation cost of C_1.

Various logic blocks are associated with the TAP (see Section 2.4.1). These are the Instruction Register (IR), the Bypass Register (BYPR), a multiplexer to select between the instruction and the data chains, and clock selection circuitry. The clock selection circuitry is used to control the functional and test clocks. The IR, BYPR, output buffer, clock selector and the instruction/data chain selection multiplexer are controlled by signals from logic block C_2. The inputs of C_2 are the outputs of the state flip-flops of the ITAPC. The two blocks C_1 and C_2 are implemented separately because the signals that C_1 drives go all over the chip while the signals that C_2 drives are local to the TAP. Experimental results have shown that implementing these blocks separately as opposed to combining them leads to smaller area for the TAP controller.

4.2.2 Controlling an Example Datapath

Demo (Fig. 4.1(c)) is an example datapath with two combinational logic blocks K_1 and K_2, and two functional registers R_2 and R_3. Boundary scan registers R_1 and R_4 are added to the primary data inputs and outputs. R_2 and R_3 each have a signal *ld* (load) which are primary control inputs and are tied to boundary scan register R_5. The functional registers are in the load (hold) mode of operation when ld is a 1 (0). An embedding of the full scan TDM adds shift capability to R_2 and R_3.

Let t_1 (t_2) be the number of vectors required for testing K_1 (K_2). Assume that $t_2 >> t_1$. To reduce test application time, the test data registers are configured into two chains:

1. Chain 1 - consists of R_1, R_2, R_3 and R_4.

2. Chain 2 - consists of R_2, R_3 and R_4.

Furthermore, to comply with the IEEE 1149.1 standards, R_1, R_5 and R_4 must be configured as a *boundary scan chain* (chain 0). Two 2-1 multiplexers M_1 and M_2 are needed to configure the three chains. Two instructions FSCAN1 and FSCAN2 are added to the instruction set. The instruction set now consists of FSCAN1, FSCAN2, EXTEST, BYPASS and SAMPLE, where the last three instructions are the mandatory boundary scan instructions. When FSCAN1 is loaded in IR, a test vector is shifted into registers R_1, R_2, R_3 and R_4 in the *shiftDR* state and the result captured in the *captureDR* state. t_1 test vectors are applied to both the kernels K_1 and K_2. Instruction FSCAN2 is then loaded into the IR to select chain 2 and $(t_2 - t_1)$ vectors are applied to K_2.

4.2.2.1 Test Register Implementation

Implementation details of the input BSRs (e.g., R_1, R_5), output BSRs (e.g., R_4), scannable functional registers (e.g., R_2, R_3) and the IR are shown in Figs. 4.2(a),(b), (c) and (d), respectively. Types *1, 2* or *3* in Fig. 4.2 refer to the different types of basic flip-flops shown in Figs. 4.3(a),(b) and (c). These flip-flop types support both scan and BIST methodologies. In particular, R_1 and R_5 are composed of 1-bit register cells as shown in Fig. 4.2(a), where the register is of Type 2. R_4 is composed of 1-bit register cells, where the first flip-flop is also of Type 2. The functional registers R_2 and R_3 are composed of 1-bit register cells shown in Fig. 4.2(c). Three flip-flop types have been specified. A Type 1 flip-flop has only load and hold functions. Type 2 augments the functions of Type 1 by adding a shift mode. Type 3 augments the functions of Type 2 by adding the SA mode. Note that the PG mode comes for free in flip-flops that have the shift mode. In fact, for Type 1 the load mode can also be called a shift mode if the load input is connected to a scan chain.

Note that the implementations of these registers differ from the suggested implementations in [9] where signals *shiftDR(IR)*, *clockDR(IR)*, *updateDR(IR)* and *mode* are generated by the TAPC and used to control various modes of operations of the registers. The most significant difference in our implementations is that the clocks to the test data registers (e.g., *clockDR*) are no longer generated by the TAPC; instead an explicit hold mode is added to the flip-flops and the clock inputs are driven by free running clock signal(s). For scan TDMs the TCK signal is applied to the BSRs and the TCK / FCK (functional or system clock) signal is applied to the functional registers with the help of a clock synchronization mechanism. Clock switching is more complex for BIST designs where the boundary scan registers may be modified to support pseudo-random pattern generation and signature compaction capabilities.

4.2.2.2 Enumerating Control Line Values

The control line values for the various registers in the presence of all instructions and certain ITAPC states are given in columns R_1, R_5, R_4, R_2 and R_3 of Table 4.1. This table enumerates the control line values for all test data registers and is called a *Control Table.* It is not necessary to enumerate the control line values for all states of the TAP controller. It is sufficient to examine only those states that are responsible for shifting/loading data and ignore all states that are related to shifting/loading the instruction register. Moreover, even among the set of states related to shifting/loading data, there are a number of states in which no relevant test activity occurs as far as the test data registers are concerned. Therefore states such as *pauseDR* and *exit1DR* in which no relevant test activity occurs are represented by (mapped to) a pseudo-state *dead.* For this full scan example there are five "relevant" TAP states namely, *test-logic reset (tlr), scanDR, updateDR, captureDR* and *dead.* In the *dead* state all registers selected by the current instruction are in the hold mode. For balanced partial scan designs, under certain conditions, the *run-test/idle*

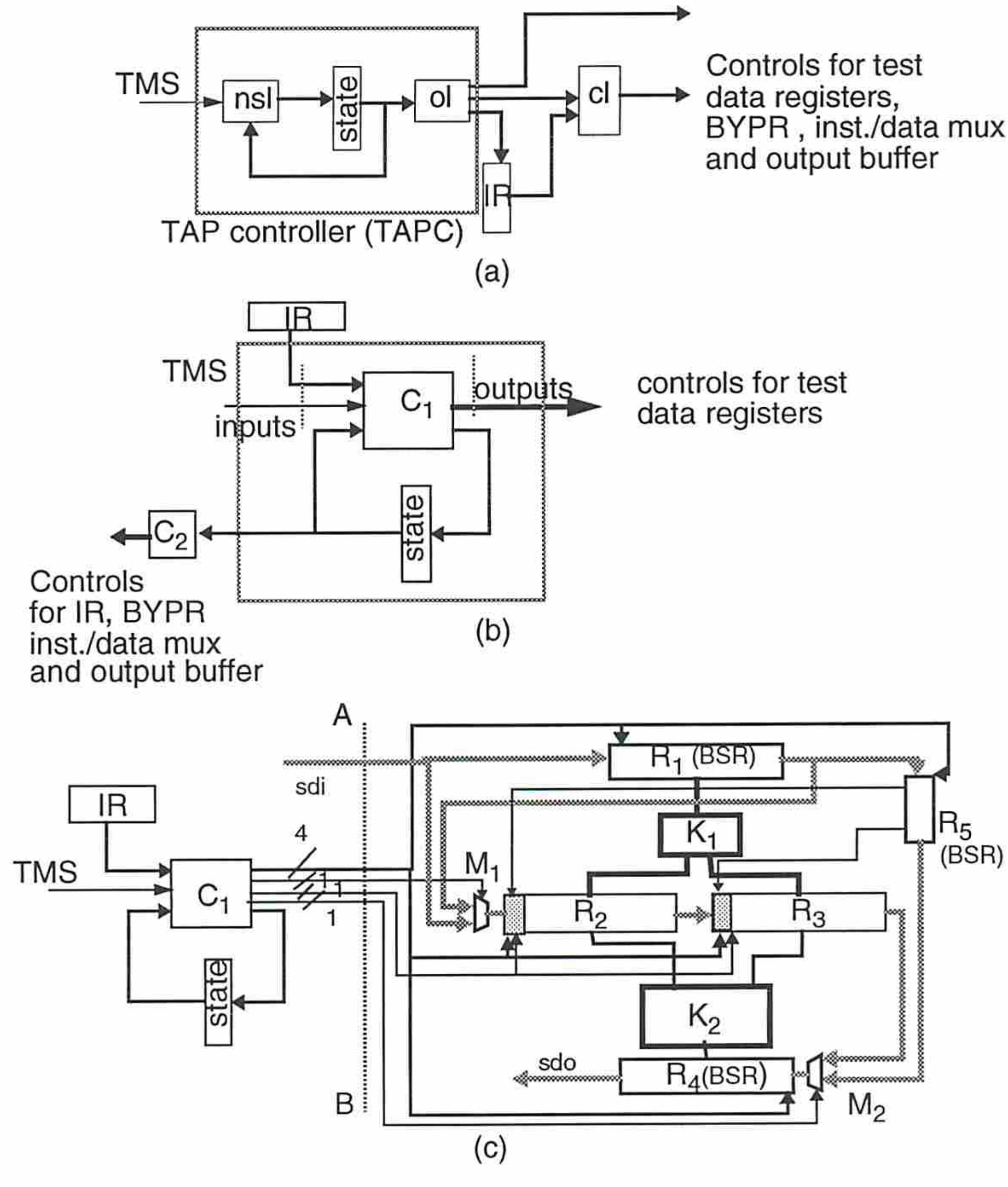

TMS
nsl
state
ol
cl
IR
Controls for test data registers, BYPR , inst./data mux and output buffer
TAP controller (TAPC)
(a)
IR
TMS
C1
outputs
inputs
controls for test data registers
state
C2
Controls for IR, BYPR inst./data mux and output buffer
(b)
A
sdi
R1 (BSR)
R5 (BSR)
IR
4
K1
TMS
C1
M1
1
1
1
R2
R3
state
K2
sdo
R4(BSR)
B
M2
(c)

Figure 4.1: (a) Canonical TAPC implementation; (b) The Integrated TAP Controller (ITAPC); (c) The ITAPC directly controlling registers in *Demo*

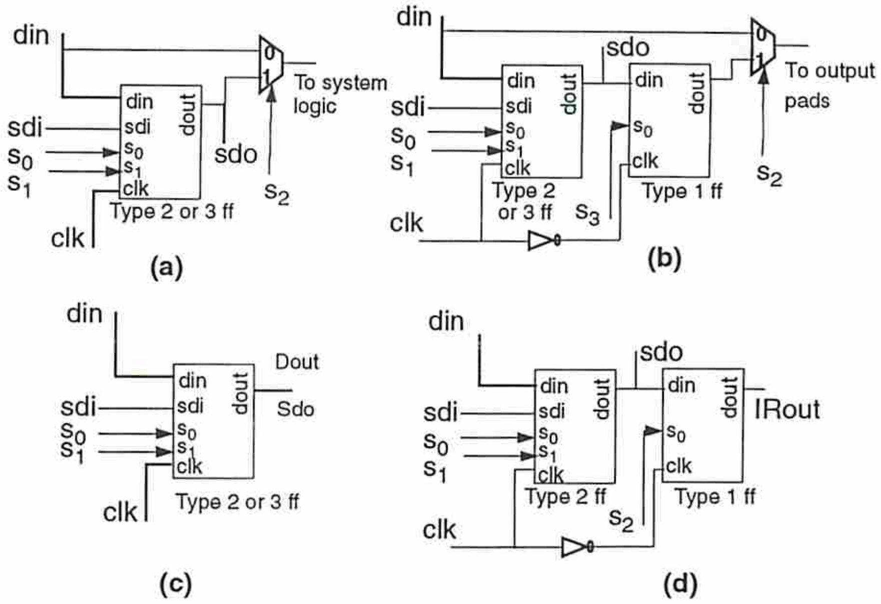


Figure 4.2: 1 bit implementation of (a) Input Boundary Scan Register; (b) Output Boundary Scan Register; (c) Scannable functional register; (d) Instruction Register

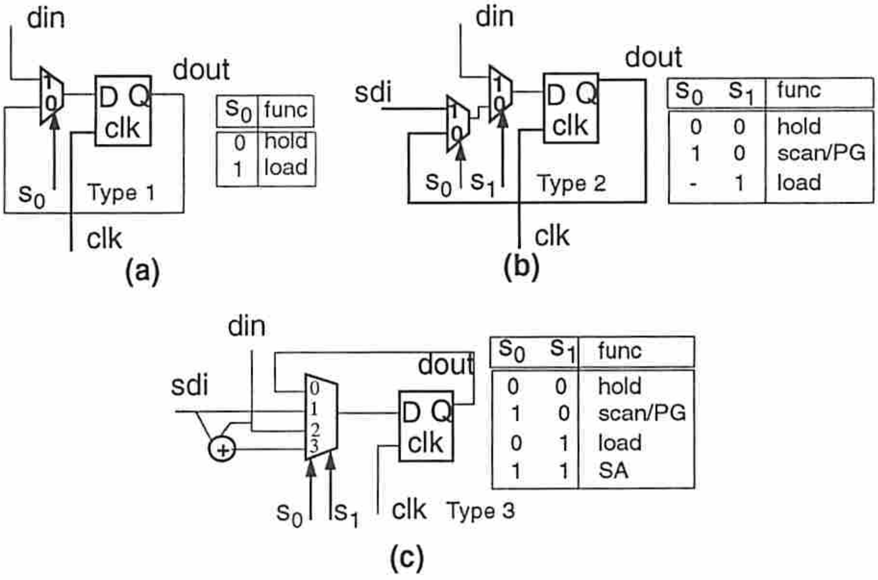


Figure 4.3: Basic flip-flop types: (a) Type 1; (b) Type 2; (c) Type 3

state may be removed from the set of "dead" states and added to the set of "relevant" states. These conditions will be presented in Chapter 5. For BIST designs this state will always be part of the "relevant" set of states.

We will now explain some of the entries in Table 4.1. Consider the lines s_0, s_1 and s_2 for register R_1. In the presence of EXTEST and *dead* the control values are 001. A 00 on s_0 and s_1 holds the flip-flop (in each of the 1-bit registers that comprise R_1) and the 1 on s_1 controls the output multiplexer such that the flip-flop drives some of the inputs of the circuit connected to R_1. In the presence of EXTEST and *captureDR*, the values asserted on the control lines are -11, where - and 1 on s_0 and s_1, respectively, are used to load the flip-flop. Note that the output multiplexer for register R_1 is always set to transmit the content of the register during EXTEST and FSCAN1, while it transmits the (primary) input when SAMPLE or BYPASS are selected (a 0 on s_2).

The boundary scan standard mandates that normal circuit function should be unaffected when SAMPLE and BYPASS instructions are loaded into the IR. Since R_2 and R_3 are functional registers with load signals, the logic values of control lines s_0 and s_1 for instructions SAMPLE and BYPASS are functions of *ld* and are represented in Table 4.1 by f_0 and f_1, respectively. f_0 is 0 (-) when *ld* is 0 (1) while f_1 is 0 (1) when *ld* is 0 (1). Thus control transfers to the *ld* signal and the registers hold (load) when *ld* is 0 (1). The logic values for m_1 and m_2 (control lines for the scan chain configuration multiplexers M_1 and M_2) are also presented in the table. Note that the multiplexer select lines need to have valid control values only during the shift-in/shift-out process (i.e., when the TAP controller is in *shiftDR*). For all TAP states other than *shiftDR* the multiplexer control lines are set to don't cares. The control signals listed under each register are generated by a local controller.

The local controllers for R_2 and R_4 contain the scan reconfiguration multiplexers in addition to decoders that generate the register and multiplexer control signals. Since the BYPASS instruction is automatically loaded in the IR whenever the ITAPC enters the *tlr* state, the control line values for all registers in the *tlr* state in presence of all instructions except BYPASS are set to don't cares. Furthermore, note that the update flip-flop of the output BSRs should always be loaded with "safe" values (using the SAMPLE operation) before starting the internal scan tests. This is because the output lines may be controlling tristate buffers or memory enable lines. Appropriate logic values are applied to control lines s_2 and s_3 of the output BSRs to ensure that the output pins are always driven from the update flip-flops during internal scan tests. Certain registers are not selected during FSCAN1 and FSCAN2. The values of the control lines for these registers are therefore set to don't cares. For example, register R_5 is not selected when FSCAN1 or FSCAN2 are loaded, therefore the control lines s_1, s_2 and s_3 are set to don't cares in the table. Similarly, register R_1 is not selected when FSCAN2 is selected and the control lines s_0, s_1 and s_2 are set to don't cares.

4.2.3 Two Control Strategies

Since the ITAPC, BYPR, IR, output buffer, clock selector and instruction/data chain selection multiplexer can all be placed close to each other, routing area for interconnect between these blocks does not constitute a major problem. However, since the ITAPC acts as a central control point and the test data registers are distributed throughout the chip, the control line routing area to these registers may contribute a significant area overhead to the overall chip area. Two strategies for controlling test registers from the ITAPC have been identified. These are

1. Direct control

2. Bus-based control

In the next section we will analyze the direct control scheme and in the rest of this chapter we will concentrate on the bus-based approach.

4.3 The Direct Control Scheme

In direct control, control lines are routed directly to the test data registers from the ITAPC. Since some test data registers may have identical control signals, certain control lines from the ITAPC can be fanned out to multiple registers. To objectively evaluate the cost of the direct control scheme, it is necessary to determine the minimal number of control lines that are needed for a testable circuit.

Inst.	ITAPC	R_1	R_5	R_4		R_2		R_3
	state	$s_0s_1s_2$	$s_0s_1s_2$	$s_0s_1s_2s_3$	m_2	s_0s_1	m_1	s_0s_1
EXTEST	dead	001	001	0010	-	- -	-	- -
EXTEST	captureDR	-11	-11	-110	-	- -	-	- -
EXTEST	shiftDR	101	101	1010	0	- -	-	- -
EXTEST	updateDR	001	001	0011	-	- -	-	- -
EXTEST	tlr	- - -	- - -	- - - -	-	- -	-	- -
SAMPLE	dead	000	000	0000	-	$f_0\,f_1$	-	$f_0\,f_1$
SAMPLE	captureDR	- 10	-10	-100	-	$f_0\,f_1$	-	$f_0\,f_1$
SAMPLE	shiftDR	100	100	1000	0	$f_0\,f_1$	-	$f_0\,f_1$
SAMPLE	updateDR	000	000	0001	-	$f_0\,f_1$	-	$f_0\,f_1$
SAMPLE	tlr	- - -	- - -	- - - -	-	- -	-	- -
BYPASS	dead	- -0	- -0	- -0-	-	$f_0\,f_1$	-	$f_0\,f_1$
BYPASS	captureDR	- -0	- -0	- -0-	-	$f_0\,f_1$	-	$f_0\,f_1$
BYPASS	shiftDR	- -0	- -0	- -0-	-	$f_0\,f_1$	-	$f_0\,f_1$
BYPASS	updateDR	- -0	- -0	- -0-	-	$f_0\,f_1$	-	$f_0\,f_1$
BYPASS	tlr	- -0	- -0	- -0-	-	$f_0\,f_1$	-	$f_0\,f_1$
FSCAN1	dead	001	- - -	0010	-	00	-	00
FSCAN1	captureDR	-11	- - -	-110	-	-1	-	-1
FSCAN1	shiftDR	101	- - -	1010	1	10	0	10
FSCAN1	updateDR	001	- - -	0010	-	00	-	00
FSCAN1	tlr	- - -	- - -	- - - -	-	- -	-	- -
FSCAN2	dead	- - -	- - -	0010	-	00	-	00
FSCAN2	captureDR	- - -	- - -	-110	-	-1	-	-1
FSCAN2	shiftDR	- - -	- - -	1010	1	10	1	10
FSCAN2	updateDR	- - -	- - -	0010	-	00	-	00
FSCAN2	tlr	- - -	- - -	- - - -	-	- -	-	- -

Table 4.1: Controls asserted by the ITAPC

Consider the control lines shown in Table 4.1. Let $R_i(s_j)$ represent the control line s_j of register R_i. The column vector consisting of 1's, 0's and don't cares corresponding to column s_j in R_i fully defines the control line $R_i(s_j)$.

Definition 4.1 *Two control lines $R_i(s_j)$ and $R_k(s_l)$ are* **compatible** *if there are no conflicting logic values in the same bit positions of the column vectors corresponding to these two control lines.*

Analyzing the compatibility between the control lines, we determine that they can be mapped into 6 unique control lines $c_0, c_1, \ldots, c_5$ as follows.

1. $c_0 \leftarrow \{R_1(s_0), R_4(s_0), R_5(s_0)\}$
2. $c_1 \leftarrow \{R_1(s_1), R_4(s_1), R_5(s_1)\}$
3. $c_2 \leftarrow \{R_1(s_2), R_4(s_2), R_5(s_2)\}$
4. $c_3 \leftarrow R_4(s_3)$
5. $c_4 \leftarrow m_2$
6. $c_5 \leftarrow m_1$

In the above, $c_i \leftarrow \{S\}$ implies that the set of control signals $\{S\}$ are mapped to control line c_i. An additional control line c_6 is needed to distinguish BYPASS and SAMPLE modes of operation from other modes such that the *ld* line can assume control of R_2 and R_3. After checking whether the control lines for R_2 and R_3 are compatible with some control lines of other registers for all instructions other than BYPASS and SAMPLE, the following equations are obtained.

$$R_2(s_1) = R_3(s_1) = c_6.c_1 + \overline{c_6}.ld \tag{4.1}$$

$$R_2(s_0) = R_3(s_0) = c_6.c_0 + \overline{c_6}.ld \tag{4.2}$$

where $c_6 = 0(1)$ for BYPASS and SAMPLE (FSCAN1 and FSCAN2). For this case, the TAP controller outputs cannot be directly connected to s_1 and s_2. Combinational logic is needed to implement the above equations. Thus true "direct" control is not possible. Fig. 4.1(c) shows *Demo* being controlled directly by the ITAPC. There are 7 control lines emanating from the ITAPC to control test registers. Cut line AB represents the boundary between datapath and control partitions.

4.3.1 Controlling Scan Designs

For scan designs the control lines in the direct control scheme control can be logically grouped into three sets.

1. Control lines for the boundary scan registers (set $\mathcal{A}$).
2. Control lines for functional registers modified as scan registers (set $\mathcal{B}$).
3. Control lines for controlling the multiplexers for reconfigurable scan chains (set $\mathcal{C}$).

Some of the control lines in these sets are compatible and can be combined.

4.3.1.1 Bounds on the Number of Control Lines

Proposition 4.1 *For designs that incorporate the test registers presented in Fig. 4.2 and have no scannable functional registers the minimum number of control lines in $\mathcal{A}$ is 4.*

Proof : A design needs at *least* one output and one input boundary scan register. An output boundary scan register has 4 control lines and from the control values asserted on these control lines (sufficient to consider only the values asserted for EXTEST in Table 4.1) it can be seen that none of these control lines are compatible to one another and therefore cannot be combined with one another. An input boundary scan register can be designed with either one flip-flop or two flip-flops. If it is a double flip-flop design, then there are four control lines which are compatible with the four control lines of an output boundary scan register. If an input boundary scan register has a single flip-flop then there are three control lines which are compatible to three control lines of an output boundary scan register. This can easily be checked from Table 4.1. Even though this table is specific to a particular example, the register control line values asserted for EXTEST, BYPASS and SAMPLE are valid for any design. The only difference for boundary scan register control values for different designs is when the boundary scan registers are used as part of different scan chains. If a boundary scan register is not used in a particular scan chain then don't cares are assigned to the control lines of the corresponding boundary scan register. Thus the inclusion of different boundary scan registers in different chains makes no difference to the compatibility of the boundary scan control lines. Therefore for designs where no internal register is made scannable, the minimum number of control lines is 4. □

Proposition 4.2 *For designs that incorporate the test registers presented in Fig. 4.2 and have at least one scannable functional register the minimum number of control lines in $\mathcal{A} \cup \mathcal{B}$ is 5.*

Proof : There are two cases to consider.
Case 1 - *None of the scannable functional registers has a load/hold control line.* In this case a scannable functional register has only two control lines s_0 and s_1. The control line s_0 is compatible with one of the control lines for a boundary scan register. A functional register needs to be in the normal (load) mode of operation for all TAP states when SAMPLE and BYPASS are loaded in the IR. Therefore s_1 is set to 1 when these two instructions are loaded. This requirement makes s_1 incompatible with any boundary scan register control line. Therefore a minimum of 5 control lines are needed.

Case 2 - *At least one scannable functional register has a functional load/hold control line.* When a functional load/hold control line exists, the control lines s_0 and s_1 are functions of boundary scan control lines and an additional signal that distinguishes the SAMPLE and BYPASS instructions from the other instructions (Equations 4.1 and 4.2). Thus a minimum of 5 control lines are again needed. □

The number of control lines in $\mathcal{C}$ depends on the number and sizes (in terms of the number of inputs) of the multiplexers used to reconfigure the scan chains.

Lemma 4.1 *A lower bound of the number of control lines for multiplexers is $\lceil log_2(N) \rceil$ and an upper bound is $min(\sum_{i=1}^{k} \lceil log_2(l(M_i)) \rceil,\ 2^N - 2)$, where $M_1, M_2, \ldots, M_k$ represent the k multiplexers used to configure N data chains and $l(M_i)$ is the number of distinct selectable inputs to M_i.*

Proof : *Upper bound*: It is not possible to use more than $\sum_{i=1}^{k} \lceil log_2(l(M_i)) \rceil$ control lines to control the multiplexers and hence this is an upper bound. Consider the case where all k muxes are 2 input muxes. This is a worst case scenario because each multiplexer can only select between two inputs and requires one control line. Let the control lines for these muxes be $m_1, m_2, \ldots, m_k$. Consider a matrix where the columns correspond to the multiplexer control lines and different configurations of the scan chains correspond to the rows. Each scan chain configuration defines a k bit control vector (a row of the matrix), where each control line can be a 1, 0 or a don't care. For N scan chains there are N such control vectors (rows). The values that each control line m_i is assigned over N reconfigurations defines a column vector. Two control lines m_i and m_j are incompatible if the corresponding column vectors differ in at least one non-don't care bit position. There can be 2^N distinct column vectors corresponding to N reconfigurations. Column vectors with all 0's and all 1's are not allowed because this implies that the corresponding multiplexers are redundant. Thus there can be at most $2^N - 2$ distinct control lines. Therefore the minimum of $(\sum_{i=1}^{k} \lceil log_2(l(M_i)) \rceil,\ 2^N - 2)$ is an upper bound on the number of control lines.

Lower bound: In the best case, a single N way multiplexer can be used to configure N scan chains requiring $\lceil log_2(N) \rceil$ control lines. □

Both the upper and lower bounds are achievable for certain scan chain configurations. The use of a single N-way multiplexer to configure N chains is *sufficient* to achieve the lower bound. The lower bound can also be achieved for other configurations that use distributed multiplexers. For example, Fig. 4.4(a) shows a circuit where two (2-1) multiplexers M_1 and M_2 are used to configure two chains. One select line is enough to control both the multiplexers. Fig. 4.4(b) presents a case where the upper bound on the number of multiplexers is reached. In this example three (2-1) multiplexers are used to configure three scan chains and three distinct control lines are needed. It is easy to show that for a set of $2m$ scannable registers, if a 2-1 multiplexer is inserted between every pair of adjacent registers and used to bypass the register immediately preceding it, then a set of m scan chains created by bypassing exactly one register R_i in chain i requires m distinct control lines, thereby always achieving the upper bound.

Corollary 4.1 *A lower bound on the number of control lines in the Direct Control scheme is max(C, $\lceil log_2 N \rceil$), where C is a constant and is equal to 5 for designs that incorporate the test register implementations presented in Fig. 4.2.*

Proof : Follows from the lower bound computations presented in Proposition 4.2 and Lemma 4.1. □

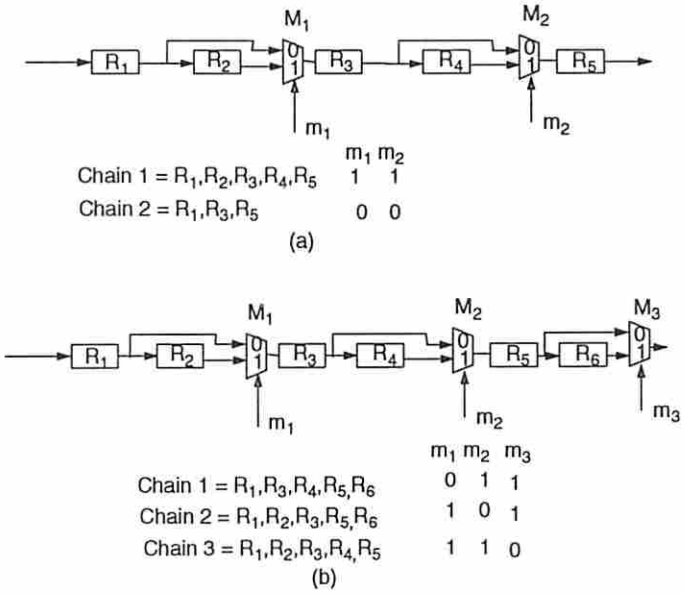

M1
M2
R1
R2
0
1
R3
R4
R5
m1
m2
m1 m2
Chain 1 = R1,R2,R3,R4,R5 1 1
Chain 2 = R1,R3,R5 0 0
(a)
M1
M2
M3
R1
R2
R3
R4
R5
R6
m1
m2
m3
m1 m2 m3
Chain 1 = R1,R3,R4,R5,R6 0 1 1
Chain 2 = R1,R2,R3,R5,R6 1 0 1
Chain 3 = R1,R2,R3,R4,R5 1 1 0
(b)

Figure 4.4: (a) Only one control line needed for controlling two scan chains; (b) Three control lines used to control three scan chains

4.3.2 Controlling BIST Designs

All BIST designs are assumed to have two scan chains, namely

1. a boundary scan chain, and

2. an internal test data chain.

All registers that participate in the BIST process are included in the internal test data chain. Reconfigurable scan chains are not considered for BIST designs because the time required for shifting in (out) test data (results) is insignificant as compared to the test application time. The serial data outputs of both chains are routed to the TAP block and the multiplexer selecting between these two chains is implemented as part of the TAP. Therefore, unlike scan designs, there is no multiplexer control line distribution problem.

4.3.2.1 Simplified BIST Model

In this section we focus on a simplified BIST model, where we consider only the BILBO TDM and assume that there are no I-paths [1]. Some functional registers and some boundary scan registers are modified to have PG and/or SA capability. Any register that has PG and/or SA capability is called a BIST register. In the direct control approach, none of the BIST registers have *mode* (refer to Chapter 3) control flip-flops. The control lines s_i of the BIST register are controlled directly from the ITAPC to set a BIST register in the load, hold, shift, PG or SA modes. There are (N+3) instructions : EXTEST,BYPASS,SAMPLE and $RBIST_1, RBIST_2, \ldots, RBIST_N$. When $RBIST_i$ is loaded into the IR and the ITAPC is in the *run-test/idle* state, certain test registers operate as PGs and others as SAs.

Consider the embedding of BILBO TDMs in circuit *Demo* to test kernels K_1 and K_2. Assume that there are no functional load control lines, hence R_5 does not exist. R_1 has PG, R_4 has SA and R_2 and R_3 have both PG and SA capability. The kernels are tested in two sessions.

- In session 1 ($RBIST_1$) K_1 is tested and the function of each of the registers is as follows: R_1 (PG), R_2 (SA), R_3 (SA) and R_4 (don't care).

- In session 2 ($RBIST_2$) the functions are: R_1(don't care),R_2 (PG),R_3(PG) and R_4(SA).

The control line values corresponding to these two instructions are shown in Table 4.2. Note that the *run-test/idle* state is explicitly specified. There are two scan chains :

- Chain 0 (boundary scan chain) - R_1,R_4

- Chain 1 - R_1, R_4, R_2, R_3.

Inspection of the table reveals that control lines $R_i(s_0)$ for i=1,2,3,4 are compatible, $R_i(s_2)$ for i=1,2 are compatible and $R_4(s_3)$ is not compatible with any control line. A minimum column cover of $R_i(s_1)$ for i=1,2,3,4 requires two additional control lines.

4.3.2.2 Bounds on the Number of Control Lines

The minimum number of control lines for BIST designs that conform to the simplified model presented in this section is $3 + \mathcal{Q}$, where $\mathcal{Q}$ is the number of columns in a minimum column cover of $R_i(s_1)$ for $i = 1, 2, \ldots, n$, and n is the number of BIST registers.

Inst.	ITAPC	R_1	R_4	R_2	R_3
	state	$s_0s_1s_2$	$s_0s_1s_2s_3$	s_0s_1	s_0s_1
RBIST1	dead	001	0010	00	00
RBIST1	captureDR	001	0010	-1	-1
RBIST1	shiftDR	101	1010	10	10
RBIST1	updateDR	001	0010	00	00
RBIST1	rti	101	--10	11	11
RBIST1	tlr	---	----	--	--
RBIST2	dead	001	0010	00	00
RBIST2	captureDR	001	0010	-1	-1
RBIST2	shiftDR	101	1010	10	10
RBIST2	updateDR	001	0010	00	00
RBIST2	rti	---	1110	10	10
RBIST2	tlr	---	----	--	--

Table 4.2: Controls asserted by the ITAPC for direct control of BILBO TDM embeddings for a modified version of *Demo*

Lemma 4.2 *A lower bound of Q is 2 and the upper bound is min(n, 2^N), where n and N are the number of BIST registers and the number of sessions, respectively.*

Proof : *Lower bound*: In a test session at least one register R_i (R_j) must be in the PG (SA) mode of operation. Therefore there will be at least one bit position where $R_i(s_1)$ will be a 0 and $R_j(s_1)$ will be a 1. Therefore these two control lines are incompatible and the size of a minimum column cover is 2.

Upper bound: Given n BIST registers an obvious upper bound is n. Note that the compatibility of control lines $R_i(s_1)$ is determined by logic values in bit positions corresponding to the *tlr* state of the ITAPC. Therefore, construct a matrix C with n columns and N rows. An entry $C_{i,j}$ corresponds to the logic value assigned to $R_j(s_1)$ in session i. A minimum column cover of $R_i(s_1)$ $i = 1, 2, \ldots, n$ is therefore equivalent to a minimum column cover of C. The maximum number of distinct column vectors possible is 2^N. Therefore if min(n,2^N) is an upper bound on the size of the column cover. □

The lower or upper bounds are achievable for certain designs. For *Demo*, Q is 2, which is equal to the lower bound. Consider a design with n BIST registers that are used to test a number of kernels in n test sessions. In session i, register R_i acts as a SA and the other $(n-1)$ registers act as PGs. Thus Q equals n which is equal to the upper bound.

4.4 The Bus-based Control Scheme

In this section we describe different types of buses, techniques for implementing descriptions of the distributed decoders and mechanisms for reducing the number of symbols transmitted over a bus.

4.4.1 Overview

4.4.1.1 Standard Bus

One bus scheme is the *Standard bus (STB)* , in which a set of control lines transmit instructions and another set of control lines carry "relevant" or "useful" ITAPC state information. The number of control lines in the Standard bus is proportional to the logarithm of the number of instructions, while in direct control scan (BIST) designs, the number of control lines may be directly proportional to the number of scan chains (number of BIST registers). For example, a scan design with 8 scan chains (including boundary scan) may need as many as 13 control lines from the ITAPC (8 lines for 2-way muxes and 5 lines for register controls. On the other hand a Standard bus only requires 7 control lines (4 lines for 10 instructions and 3 lines for 5 useful ITAPC states).

4.4.1.2 Compact and Sub-compact Buses

To reduce the number of control lines in a bus-based design, we propose two other bus schemes referred to as the *Compact bus (CB)* and the *Sub-compact bus (SCB)*. Instead of transmitting useful ITAPC states and instructions on separate sets of control lines, this information is compressed and transmitted on one set of control lines. The three buses, Standard, Compact and Sub-compact, provide tradeoffs between the number of control lines and the implementation cost of the control hardware. Fig. 4.5(a) and (b) show examples of the Standard and Compact (or Sub-compact) buses, respectively, for the scan testable circuit *Demo*. Compared to the direct control scheme which requires 7 control lines, the Standard bus has 6 control lines (3 lines to transmit 5 instructions and 3 to transmit 5 useful ITAPC states), while the Compact and Sub-compact buses both have 4 control lines.

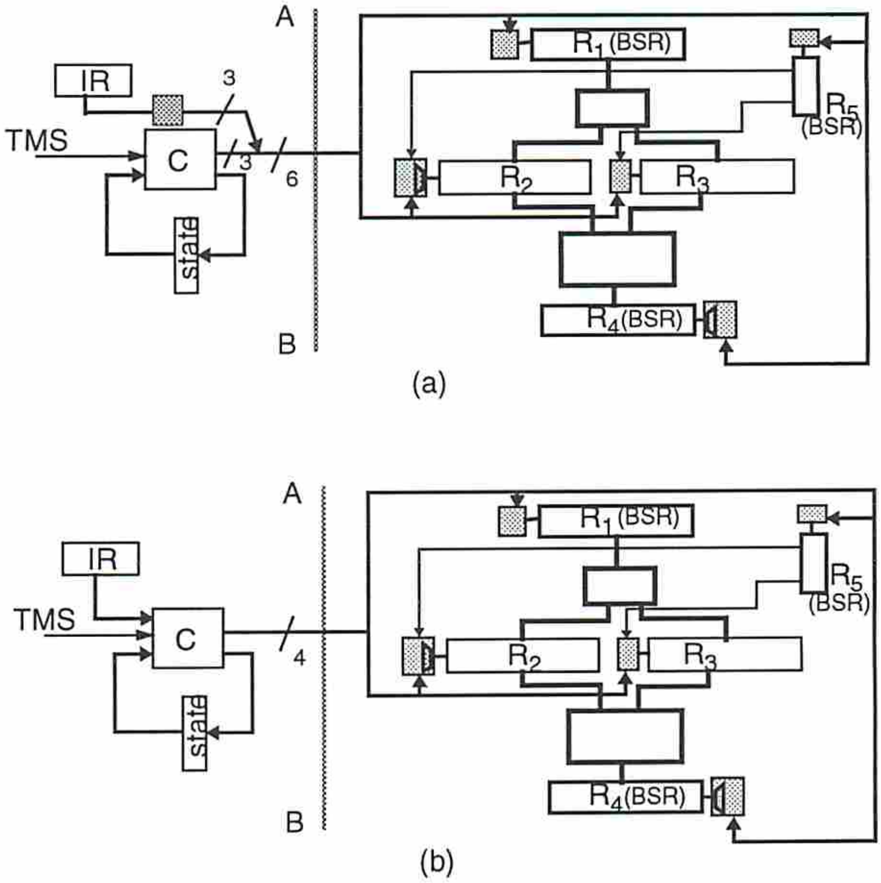


Figure 4.5: Bus-based control for scan testable *Demo*: (a) Standard bus; (b) Compact or Sub-compact bus.

4.4.2 Formalizing the Bus-based Control Scheme

Let $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ ($\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$) denote a set of n test registers (decoders). Decoder d_i is associated with register R_i. The registers include boundary scan registers and functional registers that have been modified with test capabilities. Each register has a local controller associated with it. The local controller contains a decoder, and may additionally contain a k input multiplexer and *configuration* and *mode* flip-flops. The multiplexer is used for configuring scan chains and the flip-flops are used in BIST designs to select the operational modes of the test registers. The relevance of these flip-flops have been presented in the previous chapter (Chapter 3).

4.4.2.1 Creating Decoder Descriptions

The first step in implementing a test bus consists of creating an initial symbolic description (symbolic cover) of each decoder. In this description, there are two symbolic input variables, $\mathcal{I}$ and $\mathcal{S}$, and possibly zero or more binary valued input variables, $i_1, i_2, \ldots, i_{bi(d_j)}$, where $bi(d_j)$ denotes the number of binary valued inputs corresponding to decoder d_j. The binary valued variables correspond to the functional *ld* signal (for functional registers with load and hold capability) and the outputs of the configuration and mode flip-flops (in BIST designs). The symbolic variable $\mathcal{I}$ ($\mathcal{S}$) represents the instructions (relevant tap states). $\mid \mathcal{I} \mid$ ($\mid \mathcal{S} \mid$) represents the number of symbols corresponding to variable $\mathcal{I}$ ($\mathcal{S}$) and corresponds to the number of instructions (relevant tap states). Each decoder has binary outputs $o_1, o_2, \ldots, o_{bo(d_j)}$, where $bo(d_j)$ corresponds to the number of binary valued outputs of decoder d_j. The outputs of a decoder control the associated register, and if present, the multiplexer and the configuration and mode flip-flops.

The number of implicants (rows) in the initial symbolic description of a decoder depends on $\mid \mathcal{I} \mid, \mid \mathcal{S} \mid$ and the number of binary valued inputs. For decoders with no binary valued inputs, the number of rows is $\mid \mathcal{I} \mid * \mid \mathcal{S} \mid$. For decoders with binary valued inputs, outputs corresponding to instructions for which the binary valued inputs are care inputs must be enumerated for all possible input values. For example, the *ld* signal (if present) is a care input for SAMPLE and BYPASS, and decoders d_2 and d_3 for *Demo* have 35 rows in their initial symbolic descriptions (5 rows each for EXTEST, FSCAN1, FSCAN2 and 10 rows each for SAMPLE, BYPASS). However, decoders d_1, d_4 and d_5 have no binary valued inputs and symbolic descriptions of these decoders each have 25 rows. Table 4.3 (a),(b),(c) and (d) shows the initial symbolic covers C_1, C_5, C_4 and C_2 of decoders d_1,d_5,d_4 and d_2, respectively. The symbols in $\mathcal{I}$ and $\mathcal{S}$ are represented in the positional cube notation. Thus symbols EXTEST,SAMPLE,BYPASS,FSCAN1,FSCAN2 (dead,capDR,shiftDR,updateDR,tlr) in $\mathcal{I}$ ($\mathcal{S}$) are represented by 10000,01000,…,00001 (10000,01000,…,00001), respectively.

(a) C_1	(b) C_5	(c)C_4	(d) C_2
10000 10000 001	10000 10000 001	10000 10000 0010-	- 10000 10000 ---
10000 01000 -11	10000 01000 -11	10000 01000 -110-	- 10000 01000 ---
10000 00100 101	10000 00100 101	10000 00100 10100	- 10000 00100 ---
10000 00010 001	10000 00010 001	10000 00010 0011-	- 10000 00010 ---
10000 00001 ---	10000 00001 ---	10000 00001 -----	- 10000 00001 ---
01000 10000 000	01000 10000 000	01000 10000 0000-	0 01000 10000 00-
01000 01000 -10	01000 01000 -10	01000 01000 -100-	1 01000 10000 -1-
01000 00100 100	01000 00100 100	01000 00100 10000	0 01000 01000 00-
01000 00010 000	01000 00010 000	01000 00010 0001-	1 01000 01000 -1-
01000 00001 ---	01000 00001 ---	01000 00001 -----	0 01000 00100 00-
00100 10000 --0	00100 10000 --0	00100 10000 --0--	1 01000 00100 -1-
00100 01000 --0	00100 01000 --0	00100 01000 --0--	. ...
00100 00100 --0	00100 00100 --0	00100 00100 --0--	. ...
00100 00010 --0	00100 00010 --0	00100 00010 --0--	
00100 00001 --0	00100 00001 --0	00100 00001 --0--	1 00100 00001 -1-
00010 10000 001	00010 10000 ---	00010 10000 0010-	- 00010 10000 00-
00010 01000 -11	00010 01000 ---	00010 01000 -110-	- 00010 01000 -1-
00010 00100 101	00010 00100 ---	00010 00100 10101	- 00010 01000 100
00010 00010 001	00010 00010 ---	00010 00010 0010-	- 00010 00010 00-
00010 00001 ---	00010 00001 ---	00010 00001 -----	- 00010 00001 ---
00001 10000 ---	00001 10000 ---	00001 10000 0010-	- 00001 10000 00-
00001 01000 ---	00001 01000 ---	00001 01000 -110-	- 00001 01000 -1-
00001 00100 ---	00001 00100 ---	00001 00100 10101	- 00001 00100 101
00001 00010 ---	00001 00010 ---	00001 00010 0010-	- 00001 00010 00-
00001 00001 ---	00001 00001 ---	00001 00001 -----	- 00001 00001 ---

Table 4.3: Initial symbolic covers for d_1, d_5, d_4 and d_2

4.4.2.2 Reducing Bus Symbols

It is possible to further reduce the number of control lines in a bus-based scheme by treating the instructions and the TAP states as one symbolic variable E. Initially the number of values (symbols) for this symbolic variable is $\mid \mathcal{I} \mid * \mid \mathcal{S} \mid$. Our objective is to reduce the number of symbols in E. This can be done by first determining the control values that need to be asserted for all the decoder outputs for each symbol. The set of control line values for all decoders corresponding to each symbol is represented compactly by a *Control Vector (CV)*. A CV for a design with n decoders $d_i, d_2, \ldots, d_n$ has $\sum_{i=1}^{n} bo(d_i)$ bits, where each bit can be a 0, 1 or -. Two symbols e_i and e_j for which the corresponding control vectors CV_i and CV_j are compatible can be replaced by one symbol. We discuss the compatibility of CV's as well as techniques for symbol minimization in the next subsection.

4.4.3 Determining the Minimal Number of Bus Symbols

In reducing the number of CVs (and hence the number of bus symbols), we can either preserve the don't cares (Compact bus) or convert don't cares into 1's or 0's (Sub-compact bus). Preserving don't cares in the CVs also preserves the don't cares in the outputs of the decoders and thus allows synthesis tools to take advantage of them during logic minimization. Converting don't cares into 1's or 0's leads to fewer symbols that need to transmitted on the bus, but reduces don't cares available to the logic minimization tools. We therefore consider both of these approaches to minimize the number of CVs.

4.4.3.1 Compact Bus

If none of the decoders has binary valued inputs then creating CVs and determining the number of distinct CVs is straight forward. CV_i for e_i is created by simply concatenating the outputs of all decoders corresponding to e_i. For example $001 - - - - - 0010 - 001$ is a CV obtained by concatenating the outputs 001, $- - -$, $- -$, 0010$-$, and 001 of the decoders for R_1,R_2,R_3,R_4 and R_5, respectively, corresponding to EXTEST and *dead* (see Table 4.1). The distinct CVs are found as follows.

1. Perform a lexical sort (Time complexity O($p$log$p$)).

e_0 11	e_0 01	1101	1101	e_0 11	e_0 01	- e_0 01	1101
e_1 -1	e_1 01	-101	-101	e_1 -1	e_1 01	- e_1 01	-101
e_2 01	e_2 -1	01-1	01-1	e_2 01	e_2 -1	0 e_2 -0	01--
e_3 01	e_3 -1	01-1		e_2 01	e_2 -1	1 e_2 -1	01-1
						- e_3 -1	
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)

Table 4.4: Illustration of symbol minimization for Compact bus

2. Determine unique CVs (Time complexity O(p)).

In the above $p = \mid \mathcal{I} \mid * \mid \mathcal{S} \mid$. However, the presence of binary valued inputs (as in *Demo*) complicates the situation.

Consider the covers of two simple decoders d_1 and d_2 shown in Table 4.4 (a) and (b). The initial CVs are 1101, -101, $01-1$ and $01-1$ (Table 4.4(c)). It is obvious that there are 3 unique CVs as shown in Table 4.4(d). The modified decoder descriptions are given in Table 4.4 (e) and (f). However consider the case when d_2 has a binary input asserting different outputs for symbol e_2 (Table 4.4(g)).

Definition 4.2 *A supercube $S(C)$ of a set of cubes C is defined as the smallest cube that contains all the cubes in C.*

To create the CV corresponding to e_2, the supercube $--$ of -0 and -1 (outputs for e_2 in d_2) is concatenated to 01 (the output for e_2 in d_1), that is, the output bit that depends on the binary valued inputs for a certain symbol is replaced by a don't care in the corresponding CV. From Table 4.4(h) we see that 4 symbols are now needed instead of 3.

The following procedure is used to determine the symbols for the Compact bus and to create appropriate decoder descriptions for scan based designs. Minor modifications need to be made to this procedure to handle BIST designs (discussed in Section 4.5.2.4). In the following, the cover (description) of a decoder d_i is represented by C_i.

```
    procedure Create_decoders_for_CB ($C_1, C_2, \ldots, C_n$){
  $\mathcal{CV}$ = construct_CV_CB ($C_1, C_2, \ldots, C_n$)
  $\mathcal{CV}$ = lexical_sort ($\mathcal{CV}$)
```

```
    $\mathcal{CV}_{CB}$ = pick_unique_CVs($\mathcal{CV}$)
    for each decoder $d_i$
        $C_i$=modify_decoder_descriptions($C_i$, $\mathcal{CV}_{CB}$) }
```

The procedure *construct_CV_CB* creates CV_i by concatenating the outputs of the decoders for symbol e_i as described above. The number of symbols in the Compact bus is equal to the cardinality of the set $\mathcal{CV}_{CB}$. The procedure *modify_decoder_descriptions* updates the decoder descriptions. This is done by replacing one or more input symbol pairs in a decoder cover by a new symbol. For example, two symbol pairs such as (10000 10000) and (10000 0001) in two implicants of a decoder may be replaced by 100000.

Lemma 4.3 *For scan designs, procedure* Create_decoders_for_CB *generates valid decoder descriptions, i.e., the ON-sets and the OFF-sets of the decoders are orthogonal.*

Proof : A supercube of a set of cubes C corresponding to a symbol e_i has a don't care in all bit positions where the cubes differ from one another. CV_i can only be combined with CV_j if they agree in all bit positions. Assume that CV_i corresponds to a symbol for which a set of outputs of at least one decoder, d_k, depend on the binary valued input and have been set to don't cares. Suppose there exists another CV_j which corresponds to a symbol for which no decoder outputs depend on the binary valued inputs and CV_j is compatible to CV_i. Therefore, the same symbol e_i is now assigned to at least three cubes, $cube_x$,$cube_y$ and $cube_z$, of d_k by procedure *modify_decoder_descriptions*. $cube_x$ has don't cares in bit positions where $cube_y$ and $cube_z$ differ from each other. Since a don't care in an output bit position has no meaning during minimizing with Espresso with the *.type fr* option, $cube_x$ effectively carries no information and does not influence the minimization. However consider the case, where CV_j also corresponds to a symbol for which the outputs of d_k depend on a binary valued input. If CV_i and CV_j are compatible then the validity of the resulting decoder description depends on the outputs asserted by different logic values of the binary valued input. For scan designs, a decoder can have at most one binary valued input corresponding to the functional *ld* signal. For the set of symbols that *ld* is not a don't care the same outputs are always asserted for 0 (1) values of *ld*. Therefore the above procedure always results in valid decoder descriptions. □

4.4.3.2 Sub-compact Bus

The problem of assigning 1's and 0's to don't cares to obtain a minimum cover of CVs is NP-hard. (It can be reduced to the clique-cover problem which is NP-hard.) We transform this problem to a logic minimization problem and use Espresso in the exact (heuristic) mode to determine which CVs can be merged together to obtain a minimum (minimal) set of CVs.

As in the previous procedure, an output that depends on binary valued inputs is replaced by a don't care.

Definition 4.3 *A CV is* **constrained** *if at least one of its output bits has been set to a don't care.*

Definition 4.4 *Let k_i denote the set of positions in a control vector CV_i where the bits are set to don't cares. CV_i and CV_j are* **compatible** *if the two vectors have don't cares for all bits in $k_i \cup k_j$ and are bit-wise compatible for all other bits.*

Consider the CVs in Table 4.4(h). $01--$ is a constrained CV because its last bit depends on the binary valued input in decoder d_2. $01--$ is incompatible with all the other CVs because the last bit of $01--$ is incompatible with the corresponding bit of the other vectors. A minimum cover of the CVs is given in Table 4.5(a). The modified covers of the two decoders are shown in (b) and (c). Note that if we had merged $01--$ with 0101 (Table 4.5(d)), we would obtain an incorrect cover of d_2 as shown in Table 4.5(f) since cubes $(-\ e_1\ -1)$ and $(0\ e_1\ -0)$ conflict.

The following procedure is used to determine the symbols for the Sub-compact bus and create appropriate decoder descriptions.

```
 procedure Create_decoders_for_SCB $(C_1, C_2, \ldots, C_n)$\{
$\mathcal{CV}$ = construct_CV_SCB $(C_1, C_2, \ldots, C_n)$
$\mathcal{CV}$ = lexical_sort $(\mathcal{CV})$
$\mathcal{CV}$ = pick_unique_CVs $(\mathcal{CV})$
$\mathcal{C}$ = create_espresso_description $(\mathcal{CV})$
$\mathcal{M}$ = espresso $(\mathcal{C})$
```

1101	e_0 11	- e_0 01	1101	e_0 11	- e_0 01
0101	e_1 01	- e_1 01	0101	e_1 01	- e_1 01
01--	e_2 01	0 e_2 -0		e_1 01	0 e_1 -0
	e_1 01	1 e_2 -1		e_1 01	1 e_1 -1
		- e_1 -1			- e_1 -1
(a)	(b)	(c)	(d)	(e)	(f)

Table 4.5: Illustration of symbol minimization for Sub-compact bus

```
$\mathcal{CV}_{SCB}$ = $create\_Sub\text{-}compact\_codes$ $(\mathcal{M}, \mathcal{CV})$;
for each decoder $d_i$
    $C_i$ = $modify\_decoder\_descriptions$ $(C_i, \mathcal{CV}_{SCB})$; }
```

Procedure *construct_CV_SCB* creates the set $\mathcal{CV}$ of CVs for the Sub-compact bus using the concept of compatibility in Definition 4.4. The procedure *create_espresso_file* maps the CV compaction problem into a logic minimization problem and creates the cover, $\mathcal{C}$, of a logic function. This function has p input variables, where $p = \mid \mathcal{I} \mid * \mid \mathcal{S} \mid$, corresponding to the p initial CVs. There are p minterms in the ON-SET of this function and each minterm has exactly one bit position set to 1 and the others set to 0. Each pair of incompatible CVs, CV_i and CV_j, are represented by a minterm in the OFF-SET of this function. This minterm has 1's in bit positions i and j and 0's in other positions. Espresso is then run to obtain the minimal cover $\mathcal{M}$ of the logic function. Each row in this cover defines a maximal set of CVs (maximal compatibles). An initial CV may be compatible with more than one maximal compatible with different impacts on the eventual implementation cost of the decoders. The procedure *create_Sub-compact_codes* assigns initial CVs to the maximal compatibles such that a minimal number of don't cares are converted to 0's or 1's, and creates a set of compacted CVs. The following is an outline of this procedure.

1. Start with the empty set of compacted CVs. Obtain a set of compacted CVs from the set of CVs that can be uniquely assigned.

2. Pick a CV, CV_i from the set of *unassigned* CVs. Evaluate the cost assigning CV_i to each of the maximal compatibles in terms of the number of -'s that need to be changed to 1's or 0's.

3. Assign CV_i to the maximal compatible which results in least cost and update the set of compacted CVs.

4. Repeat this steps 2 and 3 until all CVs have been assigned.

Lemma 4.4 *The procedure* Create_decoders_for_SCB *generates valid decoder descriptions.*

Proof : Similar to proof of Lemma 4.3. □

For *Demo* the number of CVs for the Compact (Sub-compact) bus is 16 (10). Thus the width of the ITB for both cases is 4. Table 4.6 presents the initial CVs and compacted CVs for the Compact and Sub-compact buses and the caption explains each of the columns in the table. Table 4.7 shows the modified decoder covers for the Sub-compact bus. Note that the symbols have been represented as s_i for clarity.

4.4.4 Comparison with the *Hudson* Control Scheme

Hudson *et al.* [44] have proposed a test control scheme for BIST designs. In this approach all functional registers are implemented as BILBO registers, i.e, they have the following modes of operation : load (normal), shift, PG and SA. Two control lines *TST0* and *TST1* are routed to all the registers and used to control the following three modes : load, shift and PG/SA. The mode of operation of a specific BILBO register in a test session is controlled by a *BIST mode-control* bit that is associated with each BILBO register. All registers are configured in a single scan chain and the mode-bit precedes each register in the scan chain. The mode-control bits are initialized along with all BILBO registers by means of the scan chain. In this section we refer to this control scheme as the *Hudson* approach. Fig. 4.6 is a model of this approach. This figure shows the BILBO registers with the mode-control bit associated with each register.

10000 10000	0010010010------	0010010010------	e_1	e_1	0010010010-00-00	e_1	e_1
10000 01000	-11-11-110------	-11-11-110------	e_2	e_2	-11-11-110--1--1	e_2	e_2
10000 00100	10110110100-----	10110110100-----	e_3	e_3	10110110100-----	e_3	e_3
10000 00010	0010010011------	0010010011------	e_4	e_4	0010010011------	e_4	e_4
10000 00001	----------------	----------------	e_5	e_5	0000000000------	e_5	e_3
01000 10000	0000000000------	000 000 0000------	e_6	e_6	-10-10-100------	e_6	e_5
01000 01000	-10-10-100------	-10-10-100------	e_7	e_7	10010010000-----	e_7	e_6
01000 00100	10010010000-----	10010010000-----	e_8	e_8	0000000001------	e_8	e_7
01000 00010	0000000001------	0000000001------	e_9	e_9	101---1010110010	e_9	e_5
01000 00001	----------------	--0--0--0-------	e_{10}	e_5	------1010110110	e_{10}	e_3
00100 10000	--0--0--0-------	001---0010-00-00	e_{11}	e_{10}			e_5
00100 01000	--0--0--0-------	-11----110--1--1	e_{12}	e_{10}			e_5
00100 00100	--0--0--0-------	101---1010110010	e_{13}	e_{10}			e_5
00100 00010	--0--0--0-------	------0010-00-00	e_{14}	e_{10}			e_5
00100 00001	--0--0--0-------	-------110--1--1	e_{15}	e_{10}			e_5
00010 10000	001---0010-00-00	------1010110110	e_{16}	e_{11}			e_1
00010 01000	-11----110--1--1			e_{12}			e_2
00010 00100	101---1010110010			e_{13}			e_9
00010 00010	001---0010-00-00			e_{11}			e_1
00010 00001	----------------			e_5			e_3
00001 10000	------0010-00-00			e_{14}			e_1
00001 01000	-------110--1--1			e_{15}			e_2
00001 00100	------1010110110			e_{16}			e_{10}
00001 00010	------0010-00-00			e_{14}			e_1
00001 00001	----------------			e_5			e_3
(a)	(b)	(c)		(d)	(e)		(f)

Table 4.6: Determining the CVs for *Demo*: (a) initial symbols represented in positional cube notation; (b) initial CVs; (c) compacted CVs where don't cares have been preserved; (d) correspondence between symbols for the Compact bus and initial symbols in (a); (e) compacted CVs where 1's and 0's have been assigned to don't cares and; (f) correspondence between symbols for the Sub-compact and initial symbols in (a).

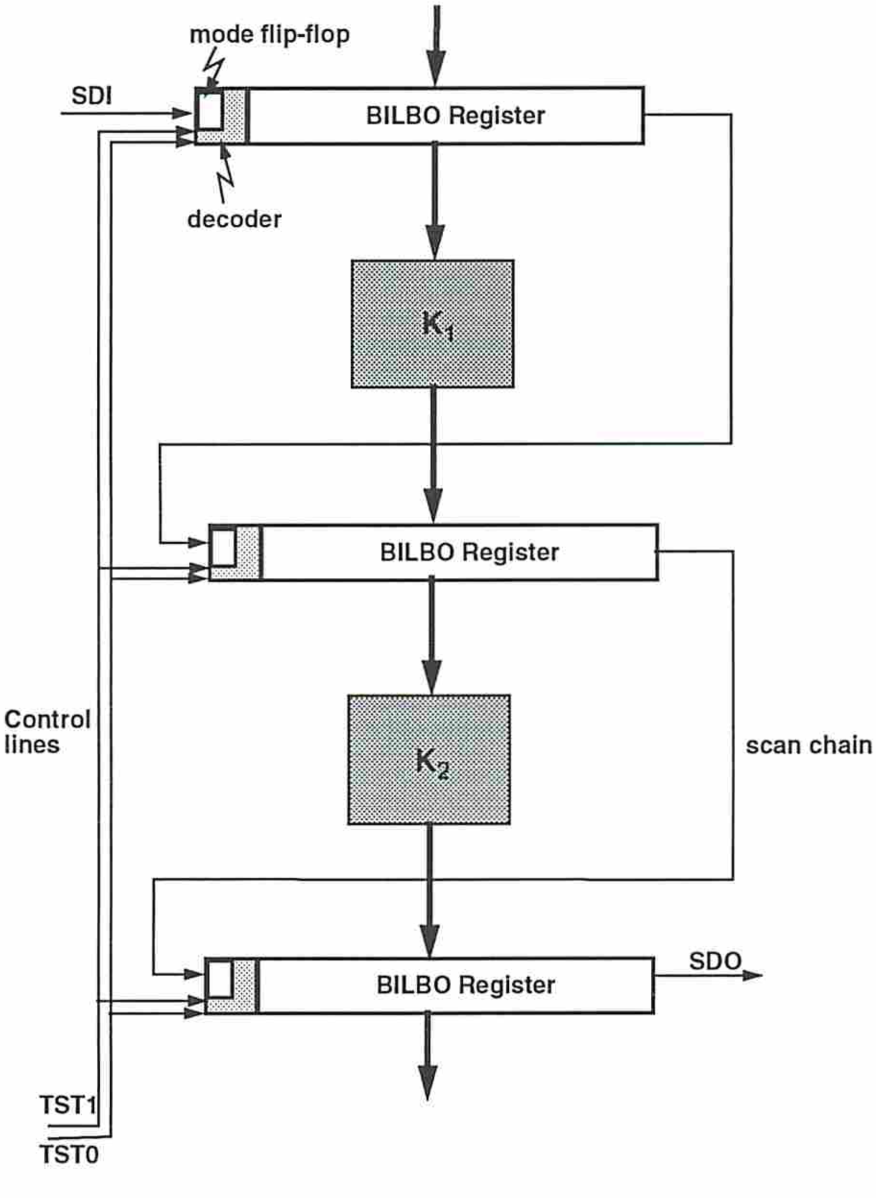


Figure 4.6: The *Hudson* control scheme

e_1 001	e_1 001	e_1 0010-	- e_1 00-	- e_1 00
e_2 -11	e_2 -11	e_2 -110-	- e_2 -1-	- e_2 -1
e_3 101	e_3 101	e_3 10100	- e_3 ---	- e_3 --
e_4 001	e_4 001	e_4 0011-	- e_4 ---	- e_4 --
e_5 000	e_5 000	e_5 0000-	0 e_5 00-	0 e_5 00
e_6 -10	e_6 -10	e_6 -100-	1 e_5 -1-	1 e_5 -1
e_7 100	e_7 100	e_7 10000	0 e_6 00-	0 e_6 00
e_8 000	e_8 000	e_8 0001-	1 e_6 -1-	1 e_6 -1
e_9 101	e_9 ---	e_9 10101	0 e_7 -0-	0 e_7 00
e_{10} ---	e_{10} ---	e_{10} 10101	1 e_7 -1-	1 e_7 -1
			0 e_8 00-	0 e_8 00
			1 e_8 -1-	1 e_8 -1
			- e_9 100	- e_9 10
			- e_{10} 101	- e_{10} 10
(a)	(b)	(c)	(d)	(e)

Table 4.7: Symbolic decoder covers for Sub-compact bus for *Demo* : (a) C_1; (b) C_5; (c) C_4; (d) C_2; (e) C_3.

We will now show that our bus based scheme for BIST designs also needs two control lines and at most a single mode control flip-flop under the following restrictions.

1. The IEEE 1149.1 boundary scan hardware is not present on the chip.

2. Reconfigurable registers are not considered.

3. A BIST register can have at most two functions in test mode : PG and SA.

The starting point of our bus-based scheme is the control table in Section 4.2.2.2. We enumerate the logic values for register control lines for all pairs of relevant TAP states and instructions. Each TAP state and instruction pair defines an "input condition" for which certain actions occur in the registers. These actions are captured in the form of register control line values. In the case where the TAP controller is absent we have to define an exhaustive set of "input conditions". These conditions are (1) normal (mode), and (2) test (mode). The test mode is a "macro" condition which represents the following : (1) shift, (2) hold, and (3) BIST operation. Therefore the following constitute an exhaustive set of input conditions : (1) normal, (2) shift, (3) hold, and (4) BIST operation. Two control lines are needed to distinguish between these conditions. The mode control bit is additionally used to distinguish between the PG and SA modes in BIST operation. Therefore if the restrictions enumerated above are met then our bus-based approach also requires two control lines. However, our approach can also deal with functional load/hold, PG/hold and SA/Hold signals since our decoder synthesis is geared to the requirements of individual registers. Moreover we also support complex data transport paths and the internal test controller. Thus though our scheme reduces to the Hudson approach in terms of number of control lines in the bus and the single mode control flip-flop, it is much more powerful in terms of supporting complex BIST TDMs. Note that even if restrictions (2) and (3) were removed, our control scheme would still require two control lines, while the Hudson approach does not even consider reconfiguration and modes other than PG and SA.

4.5 The Internal Test Bus Encoding Problem

Input (output) encoding is the process of assigning codes to symbolic input (output) variables of a function such that a cost function measuring the optimality of a two-level or multi-level implementation is minimized. Fig. 4.7(a) presents a model of our encoding problem for scan designs. In this model, the ITAPC is represented by a shell corresponding to its state transition table (STT). IR (instructions), and p_s (present state) are symbolic input variables while tms (test mode select) is a binary valued input variable. n_s (next state) and E are symbolic output variables. The symbolic variable E corresponds to the set of symbols that are transmitted over a Compact or Sub-compact bus. Decoders are represented in this model by the shells of their symbolic covers (truth tables).

Each decoder has the following inputs: (1) a symbolic variable E , corresponding to the internal bus, and possibly (2) a binary valued variable ld. The internal test bus is k bits wide and is an input to all n decoders. Fig. 4.7(b) models a design with a Standard bus. Each decoder has two symbolic input variables I and S, which represent the instructions and the relevant TAP states. The relevance of the block $IR\_code\_translator$ is explained in Section 4.5.2.4. The only difference in this coding model for BIST designs is the possible presence of additional input/output variables

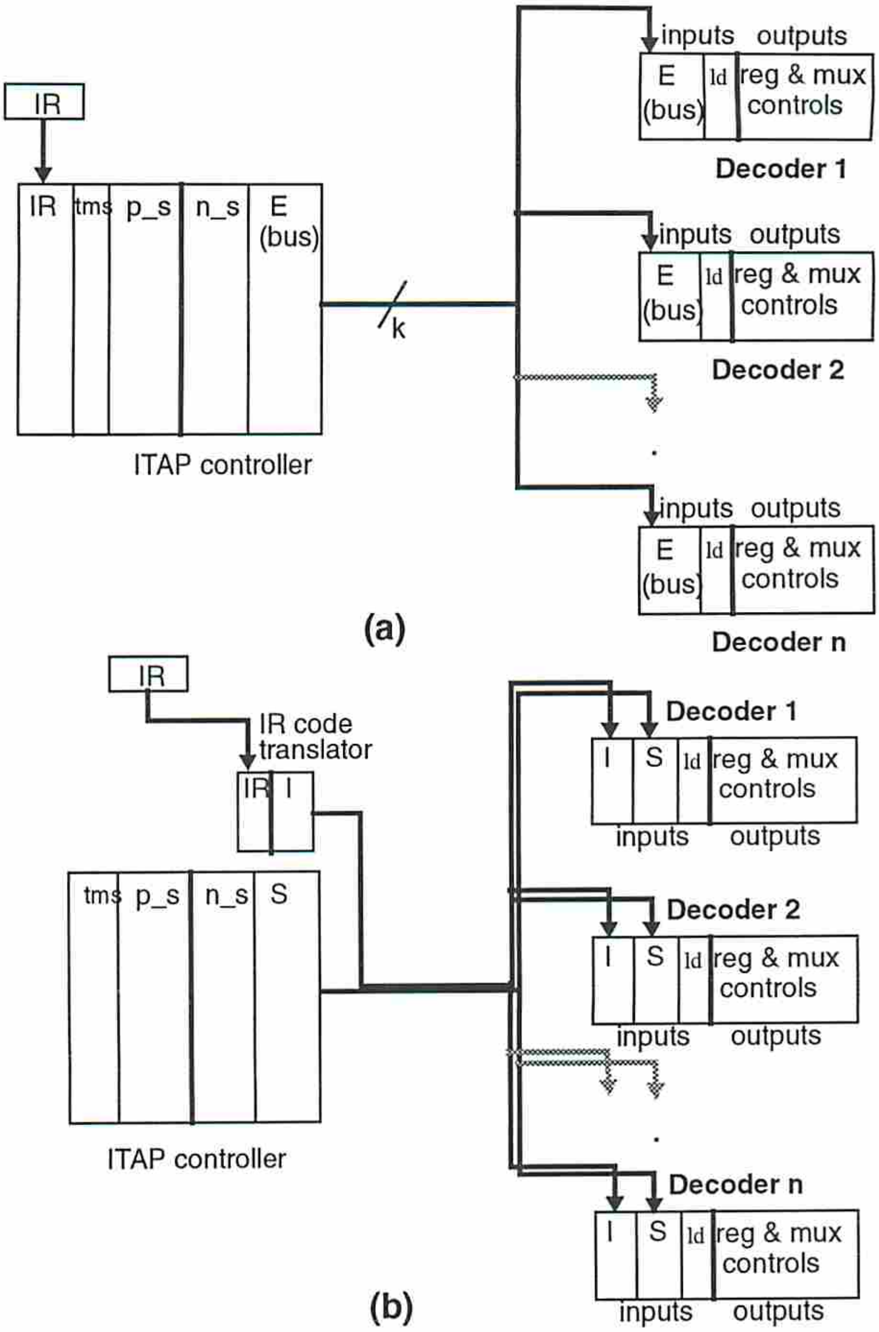


Figure 4.7: (a) Model of the Compact or Sub-compact bus; (b) Model of the Standard bus

for the decoders. Note that there is no longer an input from the IR to the ITAPC. This is because the information of the IR is transmitted explicitly. Our objective is to encode the symbols transmitted on the ITB such that the overall controller cost, i.e., cost of the ITAPC as well as the distributed decoders, is minimized.

We have developed heuristic techniques to obtain "good" encodings for the ITB. These include

1. Encoding the ITB to minimize the implementation cost of the distributed decoders without considering the ITAPC (*input encoding*).

2. Encoding the ITB to minimize the implementation cost of the ITAPC without considering the distributed decoders (*output encoding*).

3. Encoding the ITB to minimize the implementation cost of both the ITAPC and the distributed decoders (*input-output encoding*).

In all these techniques, the objective is to minimize the number of product terms in a two-level realization of the ITAPC and/or the decoders.

4.5.1 Background and Motivation

The input encoding problem for a decoder (or a logic function) with a symbolic input variable, consists of assigning binary codes to the symbols of the input variable to minimize the implementation cost of the decoder. For two-level implementation, this problem has been solved efficiently by representing the input symbols in the positional cube notation and then by obtaining a minimal multiple-valued (MV) cover of the decoder [42]. MV minimization groups together input symbols that assert the same outputs. The resulting MV cover imposes certain constraints (face-constraints or input constraints) on the codes of the symbols which, if satisfied, guarantee the existence of a binary cover with the same cardinality as the MV cover. Techniques for satisfying the face-constraints either use a graph-embedding approach [42, 43, 45] or a dichotomy-based approach [46]. Heuristic encoding techniques try to satisfy as many constraints as possible given a bound on the number of bits.

The input encoding techniques in [42, 43, 45, 46] encode input symbol(s) for a *single* logic block. However, our input encoding problem is different because we are concerned with minimizing the implementation cost of *multiple* distributed decoders, where logic sharing between the blocks is not possible.

In [42] the state assignment problem of FSMs with binary valued outputs, has been formulated as an input encoding problem. This problem is solved by simultaneously minimizing the multiple-valued input functions related to each next state and each of the binary outputs and then by satisfying the induced face constraints on the present state symbols. In this approach, the encoding of the next state symbol (output encoding) is not taken into account. [45] and [43] address the problem of encoding the next state symbols. These approaches use the ON-SETs of some next state symbols as the DON'T-CARE-SET of other symbols to reduce the cardinality of the FSM symbolic cover. This leads to the creation of bit-wise *dominance* constraints on the codes of the output symbols. The resulting encoding problem now consists of simultaneously satisfying a set of face and dominance constraints. [47] introduces additional types of constraints (*disjunctive* and *nested dominance-disjunctive*) between output symbols and presents an exact procedure for FSM encoding. In [46] a dichotomy-based approach is used to satisfy a set of face, dominance and disjunctive constraints.

Most of the research on output encoding has been confined to the case where the primary outputs are binary valued. In our case, the primary outputs of the ITAPC is a symbolic variable E and we are concerned with determining an efficient encoding of E to minimize the cost of the ITAPC. Due to its exhaustive nature, the procedure presented in [47] cannot be applied in practice. In [46] the entire output part (next states and binary or symbolic primary outputs) of the FSM is treated as one symbolic variable. Thus there may be as many as $\mid S \mid * \mid E \mid$ output symbols, for a STT with a single symbolic primary output variable E and state variable S. Moreover an exponential time procedure is used to extract an exhaustive set of dominance and disjunctive constraints. Thus no fast and efficient heuristic techniques exist for our output encoding problem.

4.5.2 Encoding Algorithms

4.5.2.1 Input encoding - Minimizing Decoder Cost

Given a set of face constraints to be satisfied, efficient heuristic graph-embedding techniques exist [43] for encoding a set of symbols. Therefore, we concentrate on the problem of encapsulating the final cost of the *distributed* decoders in a set of input constraints to be satisfied. Let IC_i be the set of input constraints obtained for decoder d_i using *Espresso* [48, 49]. One option is to satisfy the constraints generated for all the decoders, i.e., satisfy $\mathcal{IC} = \cup_{i=1}^{n} IC_i$. The problem with this approach is that all constraints are weighted equally. Suppose a constraint ic_i is a member of the constraint set of two decoders while ic_k occurs in the constraint set of only one decoder. Then $\mathcal{IC}$ does not reflect the fact that satisfying ic_i is more important than satisfying ic_k. The logical solution to this problem is to weight each constraint in $\mathcal{IC}$ by its frequency of occurrence in the constraint set of different decoders. Thus ic_i is given a weight 2 while ic_k is given a weight 1. The potential drawback of this approach is that it evenly weights constraints for different decoders. Consider a decoder with x outputs that has a constraint ic_i and another with y outputs that has a constraint ic_j, where $x > y$. In an encoding, ic_j may be satisfied while ic_i may not. The unsatisfaction of ic_i may lead to the addition of one (or more) product term(s) for the first decoder (over a minimal cost implementation) and the resultant increase in total cost may be more than that due to unsatisfaction of ic_j. We have formulated the following weight functions where w_i is the weight of a constraint $ic_i \in \mathcal{IC}$.

$$w_i = \sum_{j=1}^{n} W_j \mid W_j = 0 \;\; if \;\; ic_i \notin IC_j \;\; else \;\; W_j = 1 \tag{4.3}$$

$$w_i = \sum_{j=1}^{n} W_j \mid W_j = 0 \;\; if \;\; ic_i \notin IC_j \;\; else \;\; W_j = 2(\lceil log_2 \mid E \mid \rceil + bi(d_j)) + bo(d_j) \tag{4.4}$$

In Equation 4.3 each constraint in $\mathcal{IC}$ is weighted by its frequency of occurrence in the constraint set of different decoders. In Equation 4.4 each constraint is weighted by the number of inputs and outputs of each decoder, where $\lceil log_2 \mid E \mid \rceil + bi(d_j)$ is the total number of inputs of decoder d_j. The factor of 2 is used because the PLA area of a logic function is computed as $(2 \star i + o) \star p$, where i, o and p are the number of inputs, outputs and product terms, respectively.

The input encoding scheme is summarized in the following procedure.

```
**Procedure** *inp_enc_with_constraints* ($C_1, C_2, \ldots, C_n$,E){
    for each decoder $d_i$
        $IC_i$ = extract_input_constraints ($C_i$,E)
    $\mathcal{IC}$ = assign_weights ($IC_1, IC_2, \ldots, IC_n$,E)
    E = encode_considering_weights ($\mathcal{IC}$,E)
    return(E) }
```

The input constraints for each decoder is obtained by *extract_input_constraints.* The procedure *assign_weights* creates a set of distinct constraints obtained from the various decoders and assigns weights according to one of the weighting functions 4.3 or 4.4 given above. Finally, the symbols are encoded using the procedure *encode_considering_weights* that prioritizes the satisfaction of constraints based on their weights. This procedure calls a heuristic face-embedding algorithm used in NOVA [43].

An example of the input encoding problem applied to *Demo* for the Sub-compact bus is given in Table 4.8. In column (a), the multiple valued cover of each decoder is given. Each row of a decoder that has more than one 1 in its input plane defines an input constraint. In column (b) the input constraints for each decoder are given. For example the row 0010001010 100 for d_1 results in the input constraint e_3, e_7, e_9. The set of weighted constraints corresponding to the weighting function given in Equation 4.3 is given in column (c). A set of encodings for the 10 symbols is given in column (e) and in column (d) a Y (N) specifies whether the corresponding input constraint is satisfied (unsatisfied). These codes are inserted into the descriptions of the decoders and the final boolean cover for the different decoders is given in column (f). Note that even though one constraint was unsatisfied, the size of the final cover for each decoder still equals the size of the corresponding multiple-valued cover. This can be explained as follows.

Consider one of the constraints that was unsatisfied. e_3, e_7, e_9 obtained form d_1 is unsatisfied. But note that e_3, e_7, e_9, e_{10} is satisfied and there exists the following implicant in the symbolic cover of d_1 (Table 4.7(a)) e_{10} $---$. If this implicant

0010001010 100	$\{ (e_3, e_7, e_9),$	(e_2, e_6) 3	Y	e_1 0101	-0-- 100
0100010000 010	$(e_2, e_6),$	(e_1, e_2, e_3, e_4) 2	Y	e_2 0001	-0-1 010
1111000010 001	$(e_1, e_2, e_3, e_4, e_9)\}$	(e_5, e_6, e_7, e_8) 2	Y	e_3 0000	--0- 001
C_1	IC_1	(e_9, e_{10}) 2	Y	e_4 0100	C_1
0010001000 100	$\{ (e_3, e_7),$	(e_3, e_7, e_9) 1	N	e_5 0111	-0-- 100
0100010000 010	$(e_2, e_6),$	$e_1, e_2, e_3, e_4, e_9)$ 1	Y	e_6 0011	-0-1 010
1111000000 001	$(e_1, e_2, e_3, e_4)\}$	(e_3, e_7) 1	Y	e_7 0010	--0- 001
C_5	IC_5	(e_3, e_7, e_9, e_{10}) 1	Y	e_8 0110	C_5
0010001011 10000	$\{(e_3, e_7, e_9, e_{10}),$	(e_4, e_8) 1	Y	e_9 1000	-0-- 10000
0100010000 01000	$(e_2, e_6),$			e_{10} 1010	-0-1 01000
1111000000 00100	$(e_1, e_2, e_3, e_4),$				--0- 00100
0001000100 00010	$(e_4, e_8),$				1--- 00101
0000000011 00101	$(e_9, e_{10})\}$				-1-0 00010
C_4	IC_4				C_4
- 0000000010 100					-1--- 100
1 0000111100 010	$\{ (e_5, e_6, e_7, e_8)\}$				10-1- 010
- 0100000000 010					--001 010
- 0000000001 101					---1- 001
C_2	IC_2				C_2
- 0000000011 10	$\{(e_9, e_{10}),$				-1--- 10
- 0100000000 01	$(e_5, e_6, e_7, e_8)\}$				--001 01
1 0000111100 01					10-1- 01
C_3	IC_3				C_3

Table 4.8: Encoding the Sub-compact bus for *Demo* using only input constraints: (a) multiple valued covers of the decoders; (b) input constraints for each decoder; (c) weighted constraints; (d) constraints satisfied by the encoding in (e); (f) minimal binary covers of the decoders.

is now considered as e_{10} $1--$, then e_3, e_7, e_9, e_{10} is an input constraint instead of e_3, e_7, e_9. During logic minimization Espresso is able to use this fact and the size of the cover remains 3.

Lemma 4.5 *The number of product terms in a minimal encoded cover of a decoder d_i may be equal to the number of product terms in a minimal symbolic cover even if an initial constraint $ic_j \in IC_i$ is not satisfied by an encoding.*

Proof : Follows from the preceding discussion. □

4.5.2.2 Output encoding - Minimizing ITAPC Cost

In this section we present two heuristic output encoding procedures. The first is a fast greedy heuristic that attempts to reduce the number of 1's in the output of the ITAPC. It is based on the premise that reducing the number of 1's in the output (the ON-SET) of a logic function leads to smaller implementation cost. The second procedure considers only *dominance* constraints and is based on decomposing the problem of encoding two symbolic output variables of a logic function into the problem of recursively encoding a logic function with one symbolic output variable. We consider only dominance constraints because efficient procedures are not available for obtaining other types of output constraints.

- **Encoding based on output symbol frequencies**

```
Procedure out_enc_with_freq (STT,E) {
    F = determine_frequency_of_occurrence (STT,E)
    E = sort_output_symbols (F,E)
    E = assign_weighted_codes (E)
    return(E) }
```

Procedure *determine_frequency_of_occurrence* determines the number of rows (the frequency) in which a particular symbol occurs in the STT of the ITAPC. The procedure *assign_weighted_codes* processes the sorted symbols in E and assigns the first symbol in E the all 0's code. The other symbols in the list are then assigned codes with progressively higher weights. The number of 1's determines the weight of a code. All codes of a particular weight are assigned before moving to a higher weight code.

An example of codes assigned by this procedure in *Demo* for the Sub-compact bus is given in Table 4.9. The symbols are sorted in non-increasing order of their occurrences in the STT of the ITAPC (Table 4.9(a)). The frequency of occurrence of the symbols in E and the codes assigned to them are given in Table 4.9(b) and (c), respectively. The number of product terms in an encoded STT of the ITAPC has 32 product terms. As a comparison, the frequency of occurrences of the symbols in a multiple valued cover Table 4.9(d) and the weighted codes are given in Table 4.9(e).

e_1	76	0000	3	0001	0101	1000
e_5	56	0001	2	0010	1001	0100
e_3	10	0010	3	0000	1111	1110
e_2	6	0100	2	0100	1010	1011
e_{10}	2	1000	1	1000	0010	0011
e_9	2	0011	1	0011	1101	0000
e_8	2	0101	1	0101	1000	1101
e_7	2	0110	1	0110	0011	1010
e_6	2	1001	1	1001	1110	0111
e_4	2	1010	1	1010	0100	0001
a	*b*	*c*	*d*	*e*	*f*	*g*

Table 4.9: Encoding based on frequency for Sub-compact bus in *Demo*: (a) symbols sorted in non-increasing order; (b) frequency of occurrence in the STT ; (c) weighted code; (d) frequency of occurrence of symbols in a MV cover of the STT; (e) weighted codes for the frequency in (d); (f) and (g) codes assigned randomly.

This code results in 38 product terms in the encoded STT. This implies that computing the frequency of occurrence of the symbols in the STT rather than in a MV cover leads to better results. Two random encodings of the symbols are provided for comparison in columns (f) and (g). The encoded covers both have 42 product terms. Thus our encoding strategy produces FSMs with 24% less area product terms than random encoding.

- **Encoding based on output constraints**

```
Procedure out_enc_with_constraints (STT,E) {
  C = create_symbolic_cover (STT)
  E = encode_using_dominance_constraints (C,E)
  return(E)}
```

Procedure *create_symbolic_cover* creates a cover, C, by deleting the column corresponding to the next state entry in the STT of the ITAPC. Thus C represents the cover of only the output logic of the ITAPC. *encode_using_dominance_constraints* calls procedures in NOVA to generate a set of dominance constraints and to encode

e_1		0000
e_2	$> e_1$	1000
e_3	$> e_5,\ e_1$	1101
e_4	$> e_1$	0100
e_5	$> e_4,\ e_2,\ e_1$	1100
e_6	$> e_2,\ e_5,\ e_1$	1110
e_7	$> e_3,\ e_5,\ e_1$	1111
e_8	$> e_5,\ e_1$	0010
e_9	$> e_7,\ e_1$	1010
e_{10}	$> e_3,\ e_1$	0110
a	b	c

Table 4.10: Output encoding using constraints for Sub-compact bus in *Demo*: (a) the set of symbols in the Sub-compact bus; (b) list of dominance constraints; (c) the codes assigned.

E. The decoder and ITAPC STT descriptions are then modified with the encoded symbols.

Table 4.10 shows the codes assigned to the symbols of the Sub-compact bus for *Demo*. An encoded cover of the ITAPC has 32 product terms. Thus this approach produces an ITAPC with 23.8% less product terms a random encoding. Table 4.10(b) lists the dominance constraints extracted by the output encoding procedure. Each pair of symbols defines a dominance constraint and there are 19 dominance constraints. The number of constraints satisfied by the encoding are 16. It is easy to check that the constraints $e_8 > e_5$,$e_9 > e_7$ and $e_{10} > e_3$ are not satisfied.

4.5.2.3 Input-Output encoding - Minimizing Combined ITAPC and Decoder Cost

- **Encoding with input bias**

```
Procedure io_enc_with_inp_bias ($C_1, C_2, \ldots, C_n$,STT,E) {
   E = inp_enc_with_constraints ($C_1, C_2, \ldots, C_n$)
   e = symbol_with_max_freq (STT)
   E = flip_bits (E,e)
```

```
    return(E) }
```

io_enc_with_input_bias is essentially a procedure oriented towards satisfying input constraints. E is the encoding resulting from an encoding procedure satisfying input constraints. The procedure *symbol_with_max_freq* returns the symbol that occurs the most in the STT of the ITAPC. The corresponding symbol in E is then assigned the all 0's code by flipping all 1's in the initial encoding to 0's. The corresponding bits in the code for all other symbols are also flipped. This procedure does not compromise on the input encoding procedure and at the same time attempts to reduce the size of the ITAPC by reducing the number of 1's in the most frequently occurring symbol.

- **Encoding satisfying input-output constraints**

```
Procedure io_enc_with_constraints ($C_1, C_2, \ldots, C_n$,STT,E){
    for each decoder $d_i$
        $IC_i$ = extract_input_constraints ($C_i$,E)
    $\mathcal{IC}$ = assign_weights ($IC_1, IC_2, \ldots, IC_n$,E)
    C = create_symbolic_cover (STT)
    E = encode_using_io_constraints ($\mathcal{IC}, C, E$)
    return(E) }
```

In this procedure, first a set of weighted constraints are obtained. Then the cover of a logic function is created by deleting the next state column in the STT of the ITAPC. Finally, the dominance constraints among the output symbols in the cover C are extracted and a heuristic procedure (in NOVA) is used to satisfy the set of dominance constraints and the weighted input constraints. The procedure gives higher priority to satisfaction of input constraints over output constraints.

Table 4.11 shows the codes that are assigned by this procedure to the symbols of the Compact bus for example *Demo*. The encoded STT has 36 product terms and the costs of each of the decoders is equal to the 1-hot coded cost. The input constraints are the same as a simple input encoding case and the dominance constraints are the same as a output encoding case. An analysis of the codes reveals that 13 out of 14 input constraints and 15 out of 19 output constraints were satisfied.

e_1	0000
e_2	0100
e_3	0001
e_4	0101
e_5	0010
e_6	0110
e_7	0011
e_8	0111
e_9	1001
e_{10}	1011
a	b

Table 4.11: Encoding satisfying input-output constraints for Sub-compact bus in *Demo*: (a) the symbols; (b) the code assigned to these symbols.

4.5.2.4 Encoding the Standard Bus

The encoding algorithms presented above encode symbols corresponding to one symbolic variable. The extension to encoding two symbols for the Standard bus is simple. For input encoding, symbolic minimization is performed once for each decoder and the set of constraints for each of the symbolic variables can be obtained and assigned weights. Then each variable is encoded separately to satisfy the corresponding set of input constraints.

The boundary scan standard mandates that the EXTEST and BYPASS instructions be assigned the all 0's and all 1's code, respectively. However, an encoding satisfying the input constraints may not assign these codes to EXTEST and BYPASS. A coding constraint for only one instruction can be easily satisfied by bit-flipping, but satisfying the coding requirements for two instructions may not always be possible. We solve this problem by using a code translator between the instruction register and the standard bus. EXTEST and BYPASS are assigned the all 0's and all 1's code and the remaining codes are assigned decimal code values starting form 1. The code translator performs a one-to-one mapping between the codes assigned to the instructions in the instruction register and the codes that need to be transmitted on the bus. The cost of this translator is negligible as compared to the savings accrued by minimizing the cost of the decoders. A model of the code translator labeled *IR_code_translator* is shown in Fig. 4.7(b).

Output (input-output) encoding for the standard bus can use either of the two output (input-output) encoding procedures presented. Note that output and input-output encoding algorithms are relevant only for encoding symbols in $\mathcal{S}$. The symbols in $\mathcal{I}$ are always encoded using the procedure *inp_enc_with_constraints*. Recall that the procedures for creating the Compact and Sub-compact buses created valid decoders for scan designs. For BIST designs the same procedures can be used with the restriction that the symbol e_i corresponding to the RBIST instruction and the *run-test/idle* state cannot be combined with any other symbol if any register has a functional *ld* signal as well as mode and/or configuration flip-flops. In the absence of this restriction there may be a conflict in the control line settings (leading to invalid decoder descriptions) if e_i is merged with a symbol e_j for which the *ld* signal is a care input.

4.6 Experimental Results

We have implemented prototype software for CONSYST which synthesizes the control circuitry for various scan and BIST TDMs. In this chapter the results for various scan designs have been presented. The results of running CONSYST on several BIST designs will be reported later.

4.6.1 Characteristics of Example Circuits

CONSYST was run on several example datapaths. The characteristics of these examples are presented in Table 4.12. Entries *Regs*, *BSRs*, *ld* and *Chains* represent the total number of scannable registers (including boundary scan), the number of boundary scan registers, the number of functional registers with *load* control lines and the number of scan chains in a design, respectively. Entry *in* (*out*) specifies the number of input (output) boundary scan registers. Entry *Muxes* provides information about the multiplexers that are used to reconfigure the scan chains. The entries in this column are of the form $x(y)$, where x refers to the number of 1-bit y input multiplexers. Table 4.13 records the bit widths required by the different buses for

Example	Regs	BSRs		ld	Chains	muxes
		in	out			
demo	5	2	1	2	3	2(2)
case3	6	1	1	2	5	2(2)
scan6	6	2	2	2	9	1(2),2(3),1(4)
uscdp_fs_10	10	2	2	3	4	2(2)
partscan	17	13	1	2	6	4(2)
large	18	2	2	3	9	1(2),2(3),1(4)
uscdp_fs_20	20	10	1	5	7	5(2),2(3)

Table 4.12: Characteristics of example circuits

Example	STB			CB		SCB	
	$\mid I \mid$	$\mid S \mid$	bits	$\mid E \mid$	bits	$\mid E \mid$	bits
demo	5	5	6	16	4	10	4
case3	7	5	6	22	5	9	4
scan6	11	5	7	34	6	13	4
uscdp_fs_10	6	5	6	19	5	10	4
partscan	8	5	6	25	5	12	4
large	11	5	7	34	6	13	4
uscdp_fs_20	9	5	7	28	5	14	4

Table 4.13: Number of symbols and bits required by different buses

each design. The entries *STB*, *CB* and *SCB* in Table 4.13 represent the number of symbols ($\mid I \mid$ and $\mid S \mid$, or $\mid E \mid$) and the number of bits (*bits*) required for the Standard, Compact and Sub-compact buses, respectively. From the table we see that the Sub-compact bus results in a large reduction of the number of symbols as compared to the Compact bus.

4.6.2 Detailed Results for an Example Circuit

Tables 4.14 and 4.15 record the number of product terms (*pt*), PLA area (*pla*) and standard cell layout area (*area*) for register decoders and the ITAPC in example *case3* for the Sub-compact bus. The rows labeled $d_1, d_2, \ldots, d_6$ correspond to the

	1-hot		RND			IC			OW		
	pt	pla	pt	pla	area	pt	pla	area	pt	pla	area
d_1	3	51	5	55	4.39	5	55	2.74	6	66	4.80
d_2	4	72	6	72	4.81	7	84	4.00	8	96	6.02
d_3	4	76	6	78	4.22	4	52	1.47	5	65	3.54
d_4	3	54	5	60	4.58	3	36	1.47	5	60	3.04
d_5	3	51	5	55	2.30	4	44	1.32	4	44	1.86
d_6	2	32	4	40	1.86	3	30	1.14	4	40	1.63
dtot		336		360	22.16		301	12.14		371	20.89
TAP			46	1104	57.57	36	864	56.54	37	888	46.92
Tot				1464	79.73		1165	68.68		1259	67.81
%red			n.a	n.a	n.a		20.4	13.8		14.0	14.9
CPU			0.83			2.07			2.13		

Table 4.14: Results of IC and OW algorithms for Sub-compact bus for example circuit *case3*

	OC			ICOZ			ICOC		
	pt	pla	area	pt	pla	area	pt	pla	area
d_1	6	66	3.25	5	55	2.87	4	44	2.46
d_2	7	84	4.99	7	84	4.73	5	60	3.05
d_3	4	52	2.85	4	52	1.71	4	52	2.06
d_4	3	36	1.85	3	36	1.55	3	36	1.39
d_5	4	44	2.22	4	44	1.07	3	33	1.02
d_6	3	30	1.52	3	30	0.92	2	20	0.97
dtot		312	16.68		301	12.85		**275**	**10.95**
TAP	**32**	**768**	**43.49**	38	912	50.85	33	792	46.98
Tot		1070	60.17		1213	63.7		**1067**	**57.93**
%red		26.9	24.5		17.1	20.1		**27.1**	**27.3**
CPU	29.43			3.38			69.62		

Table 4.15: Results of OC, ICOZ and ICOC algorithms for Sub-compact bus for example circuit *case3*

4.6.3 Summary of Results for Example Circuits

Tables 4.16, 4.17 and 4.18 (Tables 4.19, 4.20 and 4.21) provide a short summary of the results of applying the encoding schemes to the various examples for the Sub-compact (Standard) bus. The columns have the same meaning as in Tables 4.14 and 4.15. As noted in the preceding subsection, the input and input-output oriented algorithms (IC,ICOZ and ICOC) are effective in reducing both the PLA and gate layout areas of the decoders. Similarly, the output and input-output oriented algorithms (OW,OC,ICOZ and ICOC) are effective in reducing the cost of the ITAPC. For some examples ICOC produces lower cost decoders than IC. This is because the set of codes assigned to the symbols are different in these two cases. The difference in codes arises because ICOC also attempts to satisfy output constraints.

Decoder and ITAPC areas for the Compact bus were also obtained. In most cases the results were inferior to the Sub-compact bus because of the large differences in the number of symbols transmitted over these buses (see Table 4.13). Therefore we do not consider the Compact bus as a viable scheme for scan testable designs. However, a couple of experiments for BIST testable designs indicate that the difference between the number of symbols in the two buses is small and the Compact bus may lead to smaller implementation costs as compared to the Sub-compact bus. For the Standard bus, the symbols corresponding to I are always encoded satisfying input constraints.

The results in Tables 4.19, 4.20 and 4.21 show that: (1) the PLA areas for the decoders are the same for all encoding schemes, and (2) OW seems to produce much better layout areas for the decoders than the input oriented encoding schemes such as IC. For IC both I and S are encoded satisfying input constraints. An analysis of the input constraints reveals that for all examples there are no input constraints for S. Since there are no input constraints IC randomly encodes the symbols of S. In OW, I is assigned the same codes as in IC. Additionally, S is encoded with minimal weight codes. This reduces the number of 1's in the *input* of the decoders. For all examples I is always encoded such that all constraints are satisfied. Thus the number of product terms (and hence PLA) areas for decoders for all approaches are the same. However, the number of gates (and therefore the layout area) depends on the number of literals in the input cubes of a two-level minimal solution. The codes assigned to S by OW result in less literals than a random encoding (assigned by IC), thereby minimizing the layout area.

Table 4.22 records the average reductions in total decoder (*%red_dec*), ITAPC (*%red_*TAP) and combined decoder and ITAPC (*%red_tot*) PLA and layout areas for the different encoding schemes. These results are reported for the Sub-compact (SCB) and Standard (STB) buses. For the Sub-compact bus, ICOC is most effective in reducing the area of the decoders and OC (OW) results in greatest reductions of the PLA (layout) areas of the ITAPC. The overall reduction of area for an encoding algorithm depends on the relative sizes of the decoders and the ITAPC. It is expected that for a design with a large number of scannable registers, where the total decoder area is larger than the ITAPC area, input oriented algorithms should perform better than output oriented algorithms. For the Sub-compact bus the results show that ICOC produces control hardware that has on average 34% (24%) less layout (PLA) area than random encoding. For the Standard bus OW (ICOZ) results in the largest reduction in decoder (ITAPC) area. An overall reduction of 26% (16%) is obtained by OW (ICOZ) for layout (PLA) area.

4.6.4 Comments

The results presented in the various tables show that for different types of buses, depending on the number and the complexity of the registers and scan chains, certain encoding schemes produce better results than others. For the Sub-compact bus, ICOC appears to perform well for all the examples because it is able to reduce both the cost of the decoders and the ITAPC. However, for large designs, the CPU time may prove to be a bottleneck, specially since the controller synthesis system may have to evaluate the controller costs for a large family of testable designs. In that case, a faster encoding algorithm is needed. From the experimental results it is seen that ICOZ performs reasonably well for a wide range of designs and is much faster than ICOC. Therefore ICOZ may be used when a good and fast solution is needed. For Standard bus, OW performs well for a wide range of designs and is also reasonably fast.

Examp.		pla	area	pla	area	pla	area
		RND		IC		OW	
	dtot	339	18.83	219	8.86	328	18.76
	TAP	936	52.05	768	49.47	792	45.00
	Tot	1275	70.88	987	58.33	1120	63.76
	%red	n.a.	n.a.	22.6	17.7	12.2	10.0
demo	CPU	0.78		3.90		2.05	
		OC		ICOZ		ICOC	
	dtot	363	26.08	219	8.33	219	8.95
	TAP	768	50.62	864	38.06	864	42.95
	Tot	1131	76.71	1083	46.39	1083	51.9
	%red	11.3	-8.2	15.1	34.6	15.1	26.8
	CPU	27.56		4.75		81.37	
		RND		IC		OW	
	dtot	360	22.16	301	12.14	371	20.89
	TAP	1104	57.57	864	56.54	888	46.92
	Tot	1464	79.73	1165	68.68	1259	67.81
	%red	n.a	n.a	20.4	13.8	14.0	14.9
case3	CPU	0.83		2.07		2.13	
		OC		ICOZ		ICOC	
	dtot	312	16.68	301	12.85	275	10.95
	TAP	768	43.49	912	50.85	792	46.98
	Tot	1070	60.17	1213	63.7	1067	57.93
	%red	26.9	24.5	17.1	20.1	27.1	27.3
	CPU	29.43		3.38		69.62	
		RND		IC		OW	
	dtot	630	47.74	519	31.92	579	42.89
	TAP	1326	71.70	1248	64.13	1040	51.21
	Tot	1956	119.44	1767	96.05	1619	94.10
	%red	n.a.	n.a.	9.7	19.9	17.2	21.2
scan6	CPU	1.25		28.62		6.98	
		OC		ICOZ		ICOC	
	dtot	605	41.57	519	31.26	401	20.24
	TAP	1014	58.20	910	56.00	858	49.21
	Tot	1619	99.77	1429	87.25	1259	69.45
	%red	17.2	16.5	26.9	27.0	35.6	41.9
	CPU	92.00		34.15		283.37	

Table 4.16: PLA and layout areas for Sub-compact bus for examples *demo*, *case3* and *scan6*

Examp.		pla	area	pla	area	pla	area
uscdp fs_10		RND		IC		OW	
	dtot	679	37.89	604	37.95	679	43.98
	TAP	1032	66.67	1176	54.45	840	46.39
	Tot	1711	105.56	1780	92.4	1519	90.37
	%red	n.a.	n.a	4.0	12.5	11.2	14.3
	CPU	1.48		6.25		2.58	
		OC		ICOZ		ICOC	
	dtot	619	32.19	612	40.63	407	15.39
	TAP	744	43.57	792	48.62	912	52.26
	Tot	1363	75.76	1404	89.25	1319	67.55
	%red	20.3	28.2	17.9	15.5	22.9	36.0
	CPU	32.28		7.00		129.35	
partscan		RND		IC		OW	
	dtot	1270	98.37	725	33.79	1189	86.04
	TAP	1008	55.57	816	58.16	864	45.60
	Tot	2278	153.94	1541	91.96	2053	131.64
	%red	n.a.	n.a.	32.4	40.2	9.9	14.5
	CPU	2.60		22.67		4.17	
		OC		ICOZ		ICOC	
	dtot	1074	84.24	725	34.48	725	35.41
	TAP	912	56.76	792	46.84	864	60.41
	Tot	1986	141.02	1517	81.32	1589	95.83
	%red	12.8	8.4	33.4	47.2	28.0	37.8
	CPU	61.00		23.75		231.17	
large		RND		IC		OW	
	dtot	1038	65.11	977	57.12	1067	64.7
	TAP	1326	73.52	1196	71.37	1040	49.27
	Tot	2364	138.63	2173	128.49	2107	113.97
	%red	n.a	n.a	8.08	7.31	10.87	17.79
	CPU	2.90		61.95		8.62	
		OC		ICOZ		ICOC	
	dtot	983	55.81	977	57.85	756	37.41
	TAP	1014	58.22	936	59.68	1300	64.75
	Tot	1997	114.03	1913	117.53	2056	102.16
	%red	15.52	17.74	19.07	15.22	13.02	26.3
	CPU	95.28		64.72		302.92	

Table 4.17: PLA and layout areas for Sub-compact bus for examples *uscdp_fs_10*, *partscan* and *large*

Examp.		pla	area	pla	area	pla	area
uscdp fs_20		RND		IC		OW	
	dtot	1504	108.81	888	37.35	1728	127.36
	TAP	1222	56.84	1014	53.49	1014	54.60
	Tot	2726	165.65	1902	90.84	2742	181.96
	%red	n.a.	n.a.	30.23	45.16	-5.87	-9.80
	CPU	3.42		42.03		5.58	
		OC		ICOZ		ICOC	
	dtot	1331	85.64	888	38.25	901	39.85
	TAP	936	56.81	1040	46.47	1040	55.06
	Tot	2267	142.54	1928	84.72	1941	94.90
	%red	16.84	13.95	29.27	48.85	28.80	42.7
	CPU	91.73		42.86		407.43	

Table 4.18: PLA and layout areas for Sub-compact bus for example *uscdp_fs_20*

4.7 Summary

In this chapter we have described both direct and bus-based approaches for controlling on-chip test resources. We have focused specifically on the bus-based approach. In particular, three different bus schemes have been proposed and we have analytically shown that a bus-based approach may result in a significant reduction in the number of control lines as compared to a direct control approach. Five different techniques for encoding the symbols transmitted over a bus have been formulated and it has been experimentally shown that the standard cell and PLA layout areas of the test control logic is reduced by 34% and 24% (26% and 16%), respectively, for a Sub-compact (Standard) bus as compared to random encodings. We observe that the efficacy of the different encoding schemes depend on the type of bus used as well as the relative sizes of the decoders and the integrated TAP controller.

Examp.		pla	area	pla	area	pla	area
		RND		IC		OW	
	dtot	325	9.11	291	9.09	291	6.59
	TAP	483	35.70	460	31.57	460	29.29
	Tot	785	44.81	751	40.66	751	35.88
	%red	n.a.	n.a.	4.3	9.3	4.3	19.9
demo	CPU	0.98		2.00		2.03	
		OC		ICOZ		ICOC	
	dtot	291	10.97	291	8.32	291	10.97
	TAP	460	28.36	414	29.20	437	28.66
	Tot	751	39.33	705	37.52	728	39.63
	%red	4.3	12.2	10.2	16.3	7.3	11.6
	CPU	2.58		2.08		7.48	
		RND		IC		OW	
	dtot	501	19.90	343	10.38	343	7.90
	TAP	506	31.29	437	33.40	460	29.36
	Tot	1007	51.19	780	43.78	803	37.26
	%red	n.a	n.a	22.5	14.5	20.2	27.2
case3	CPU	1.26		2.20		2.55	
		OC		ICOZ		ICOC	
	dtot	343	13.18	343	10.36	343	10.64
	TAP	437	29.76	414	28.25	437	32.54
	Tot	780	42.94	757	38.61	780	43.18
	%red	22.5	16.1	24.8	24.5	22.5	15.6
	CPU	10.02		3.15		17.17	
		RND		IC		OW	
	dtot	704	25.65	569	13.03	569	11.37
	TAP	500	32.71	450	28.81	500	30.34
	Tot	1204	58.36	1019	41.84	1069	41.71
	%red	n.a.	n.a.	15.4	28.3	11.2	28.5
scan6	CPU	1.95		6.22		7.23	
		OC		ICOZ		ICOC	
	dtot	569	15.48	569	13.11	569	11.31
	TAP	475	28.50	450	29.78	500	29.71
	Tot	1044	43.98	1019	42.89	1069	41.02
	%red	13.3	24.6	15.4	26.5	11.2	29.7
	CPU	17.62		7.6		28.12	

Table 4.19: PLA and layout areas for Standard bus for examples *demo*, *case3* and *scan6*

Examp.		pla	area	pla	area	pla	area
uscdp fs_10		RND		IC		OW	
	dtot	655	25.37	543	17.08	543	12.85
	TAP	506	35.65	437	31.32	460	32.60
	Tot	1161	61.02	980	48.40	1003	45.45
	%red	n.a.	n.a.	15.6	20.7	13.6	25.5
	CPU	2.17		3.01		3.45	
		OC		ICOZ		ICOC	
	dtot	543	20.84	543	16.38	543	17.97
	TAP	437	31.90	414	28.96	437	32.32
	Tot	980	52.74	957	45.34	980	50.29
	%red	15.6	13.6	17.6	25.7	15.6	17.6
	CPU	9.27		3.4		15.15	
partscan		RND		IC		OW	
	dtot	1229	37.06	1011	28.97	1011	25.14
	TAP	506	32.39	483	32.72	460	29.96
	Tot	1735	69.45	1494	61.89	1471	55.10
	%red	n.a.	n.a.	13.9	10.9	15.2	20.7
	CPU	3.4		5.28		5.11	
		OC		ICOZ		ICOC	
	dtot	1011	41.46	1011	31.70	1011	27.73
	TAP	437	30.15	437	26.98	460	31.45
	Tot	1448	71.61	1448	58.67	1471	59.18
	%red	16.5	-3.1	16.5	15.5	15.2	14.8
	CPU	13.53		5.4		21.8	
large		RND		IC		OW	
	dtot	1248	37.23	1151	30.5	1151	23.46
	TAP	525	35.97	475	35.07	500	29.69
	Tot	1773	73.20	1626	65.57	1651	53.15
	%red	n.a.	n.a.	8.3	10.4	6.9	27.4
	CPU	4.65		8.47		9.05	
		OC		ICOZ		ICOC	
	dtot	1151	37.62	1151	30.43	1151	31.10
	TAP	475	28.09	450	27.81	475	30.75
	Tot	1626	65.71	1601	58.24	1626	61.86
	%red	8.3	10.2	9.7	20.4	8.3	15.5
	CPU	19.55		9.65		30.02	

Table 4.20: PLA and layout areas for Standard bus for examples *uscdp_fs_10*, *partscan* and *large*

Examp.		pla	area	pla	area	pla	area
uscdp fs_20		RND		IC		OW	
	dtot	1553	48.68	1260	32.95	1260	23.78
	TAP	550	35.82	475	31.73	500	29.33
	Tot	2103	84.50	1735	64.68	1760	53.11
	%red	n.a.	n.a.	17.5	23.4	16.3	37.1
	CPU	4.23		11.08		11.50	
		OC		ICOZ		ICOC	
	dtot	1260	40.39	1260	32.12	1260	33.07
	TAP	475	28.70	450	27.83	475	33.05
	Tot	1735	69.10	1710	59.95	1735	66.09
	%red	17.5	18.2	18.69	29.1	17.5	21.79
	CPU	11.83		12.15		11.98	

Table 4.21: PLA and layout areas for Standard bus for example *uscdp_fs_20*

Bus		IC		OW		OC		ICOZ		ICOC	
		pla	area	pla	area	pla	area	pla	area	pla	area
SCB	%red_dec	24.1	39.3	-0.3	-0.5	7.3	9.1	23.9	37.9	35.1	55.7
	%red_TAP	11.0	5.7	18.3	20.8	22.3	14.2	20.8	18.6	16.4	13.6
	%red_tot	18.1	22.3	9.9	11.9	17.3	14.4	22.6	29.7	24.4	34.1
	Av_CPU	23.14		4.19		56.2		24.41		203.17	
STB	%red_dec	17.5	28.8	17.5	44.8	17.5	5.7	17.5	29.7	17.5	26.4
	%red_TAP	10.3	4.3	7.9	11.8	10.8	13.9	15.1	16.8	10.1	8.5
	%red_tot	14.0	16.8	12.6	26.6	14.0	13.1	16.1	22.6	13.9	18.1
	Av_CPU	5.34		5.61		11.13		5.97		17.26	

Table 4.22: Comparisons of the various encoding schemes for Sub-compact and Standard buses

Chapter 5

Controlling IEEE 1149.1 Compliant Scan and BIST Designs

5.1 Introduction

In Chapter 3 the test control architecture and control models for various scan and BIST registers were presented. Implementational details of the local controllers associated with these registers as well as techniques for implementing test buses and the integrated test controller (ITAPC) were detailed in Chapter 4. In this chapter we will describe in detail how various scan and BIST TDMs are controlled in a testable chip. We will also show how these testable chips are controlled at the board level. Essentially this chapter integrates the concepts presented in the preceding two chapters and shows how they are applied in controlling specific scan and BIST TDMs

5.2 Controlling Scan Designs

As mentioned in Section 3.2.1 of Chapter 3, scan TDMs are classified as full, balanced partial and unbalanced (general) partial scan [35]. Any of these TDMs can be incorporated in a circuit by the scan system SIESTA [35, 31]. To evaluate the control requirements of each of these TDMs, a high level abstraction of the control characteristics of these TDMs is necessary. This high level model is provided by *generic* test plans.

Definition 5.1 *A* **generic** *test plan specifies the behavior (modes of operation) of various modules of a design corresponding to a specific TDM without specifying individual control lines and their values.*

Thus *generic* test plans can capture the control requirements of different TDMs before these TDMs are actually embedded in a design. The modules are also specified at a very high level, e.g., scan registers and non-scan registers. The test plans shown in Figs. 5.1(b) 5.2(b) and 5.3(b) are generic plans for full scan, balanced partial scan and unbalanced partial scan TDMs, respectively.

5.2.1 Controlling Full Scan TDM

Fig. 5.1(b) is the generic test plan for the full-scan circuit shown in Fig. 5.1(a). The generic plan for a full scan design has two states, $s1$ and $s2$. In state $s1$ all registers are in the shift mode of operation and in state $s2$ all registers are in the load mode. Scan registers in full scan designs are controlled from the ITAPC via the test bus and the local decoders. A scan instruction (e.g., FSCAN1) is first shifted into the IR, then the ITAPC goes to the *shiftDR* state (under the control of the MMC) to shift in the first test vector. The ITAPC may occasionally transit to the *pauseDR* state to enable the MMC to replenish (read out) its serial data transmission (receive) buffer. All scan registers are in the hold mode in the *pauseDR* state as it corresponds to a "dead" state.

The MMC is responsible for counting the number of clock cycles that are needed to shift-in/out a test vector/result from the chip. Thus there are no on-chip shift counters. After shifting in a test vector the ITAPC moves through at least three intermediate states (*exit1DR, updateDR* and *selectDR*) before entering the *captureDR* state. The intermediate states are dead states and therefore the scan registers are in the hold mode between the shift and capture phases. Fig. 5.1(c) shows the part of the TAP STG that is used for shifting-in/out the scan vectors and for applying the full scan tests. The arrows drawn with thicker lines show the sequence of states traversed in applying full scan tests. The labels beside some of states specify the mode of operation of the scan registers.

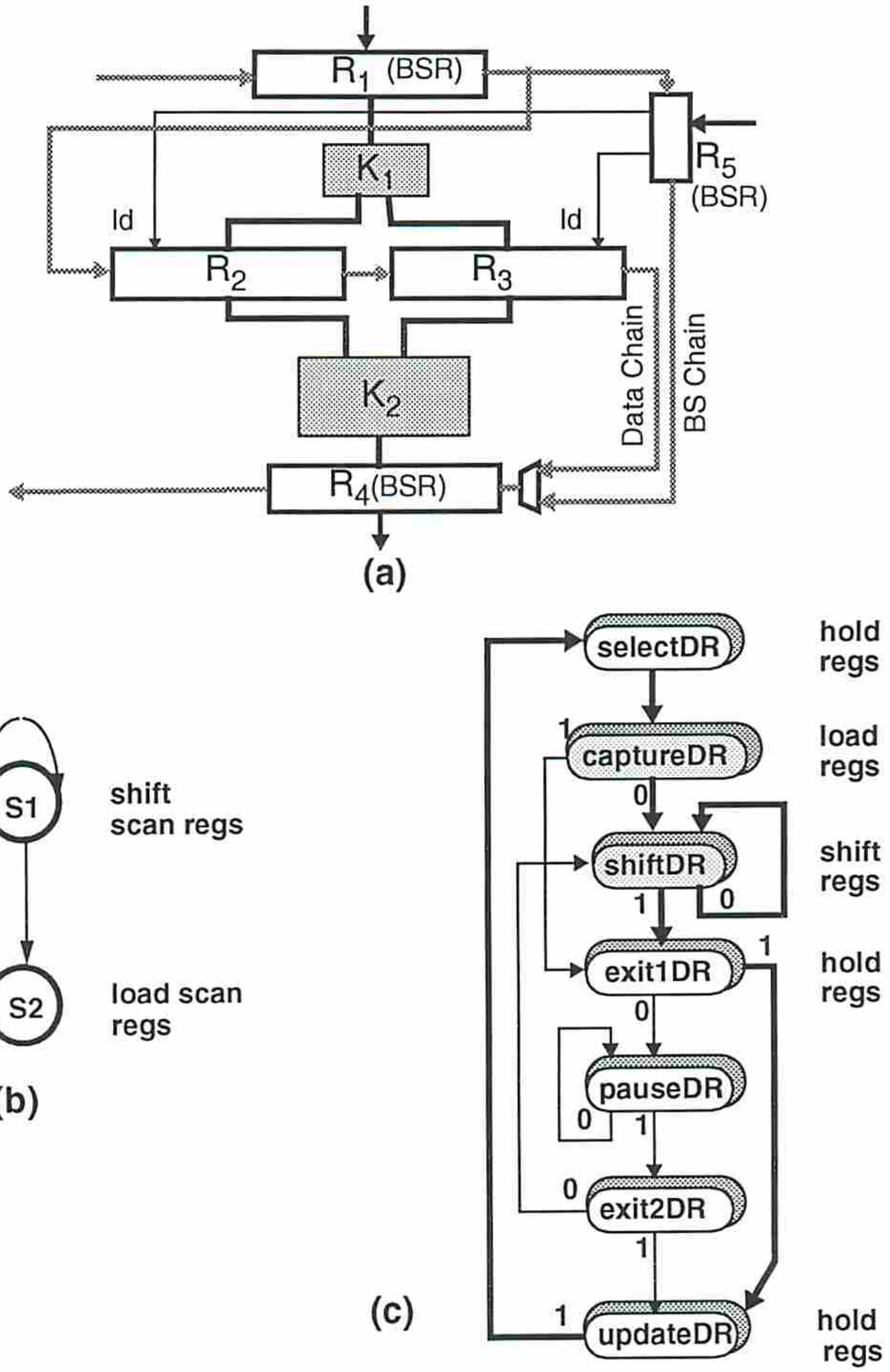


Figure 5.1: (a) Full scan circuit; (b) generic test plan for full scan circuits; (c) TAP controller activation sequence for controlling full scan circuits

5.2.2 Controlling Balanced Partial Scan TDM

Fig. 5.2(b) is the generic test plan for a balanced partial scan circuit shown in Fig. 5.2(a). In state $s1$ all *scannable* registers are in the shift mode. Non-scan registers that have load/hold control can be in any mode of operation. In state $s2$ all scan registers must be in the hold mode and all non-scan registers must be in the load mode. Each balanced circuit is characterized by its *sequential depth*, d.

Definition 5.2 *The sequential depth of a kernel (balanced or unbalanced) is the maximum number of registers in any path from input to output.*

All non-scan registers must be in the load mode (scan registers in hold mode) for d clock cycles to enable test vectors to propagate to the inputs of a scannable register (primary output of the circuit). Finally, in state $s3$ the scan registers must be in the load mode to capture the results. The non-scan registers can be in any mode (don't care).

From the test plan it is clear that the non-scan registers must be in the load mode between the shift and capture phases. This requirement is easy to meet for non-scan registers that have no load/hold control line - they are always in the load mode of operation. The test control mechanism must enforce this requirement for non-scan registers that have load/hold control lines. In normal mode these lines are either controlled from primary inputs or from a functional controller on the chip. Load/hold signals connected to primary inputs are controlled in test mode from a boundary scan register.

For scan designs, signals from a functional controller are controlled in test mode by a transparent register (has same logic as a boundary scan register) that is inserted between the functional controller and the datapath. This scannable register is also used to observe the functional controller outputs. The register(s) (boundary scan or datapath-control interface register(s)) that control the load/hold control lines is (are) part of every scan chain and at the end of the shift phase, the register(s) contains the enabling value for all load/hold control lines. In effect, the non-scan registers are in the load mode at all times during test application (states $s2$ and $s3$ of the test plan) and are in an indeterminate state during shift.

The 1-bit input boundary scan register shown in Fig. 4.2(a) has no update flip-flop. However an update flip-flop is needed if this boundary scan register is used

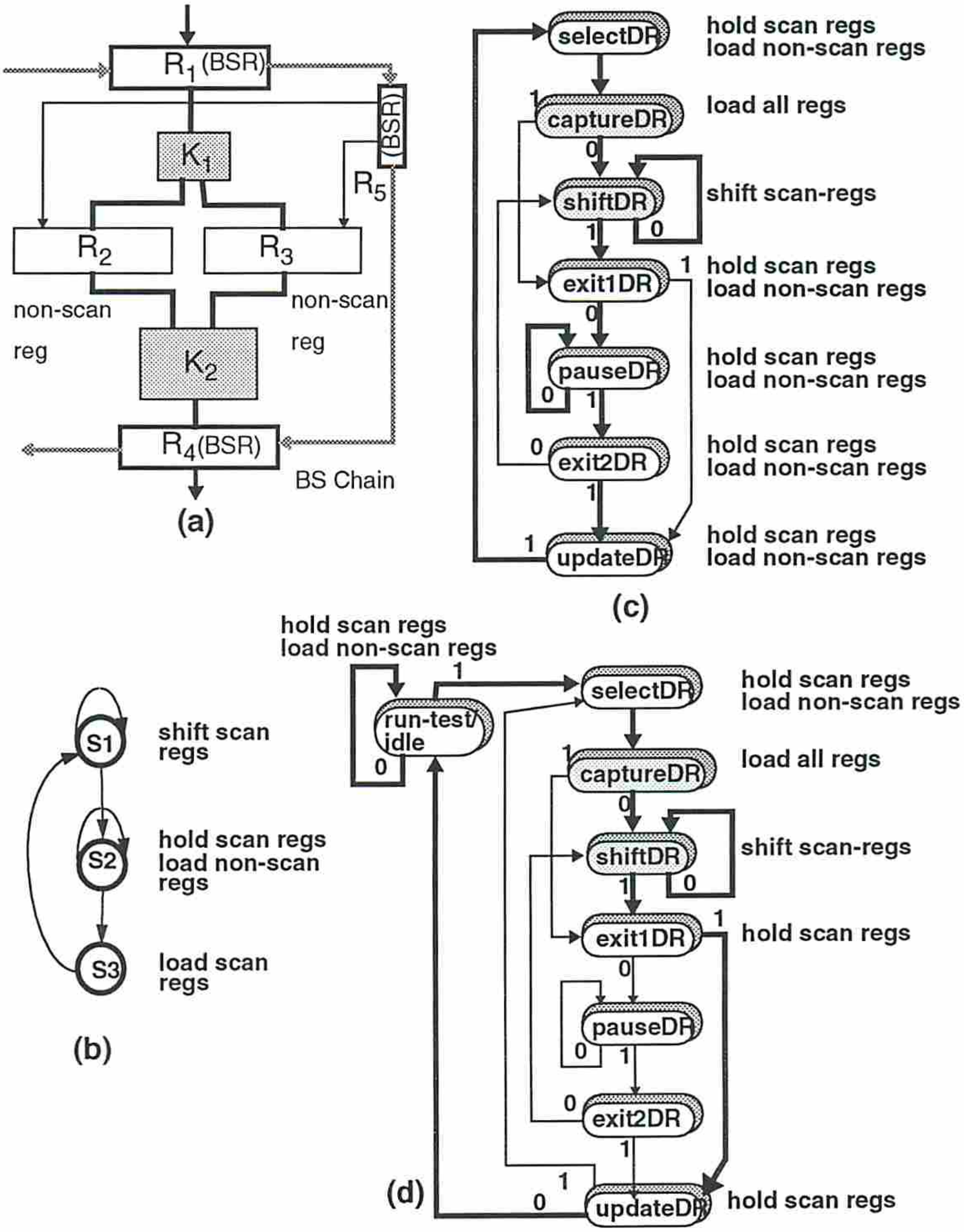


Figure 5.2: (a) Balanced partial scan circuit; (b) generic test plan for balanced partial scan circuits; (c) and (d) TAP controller activation sequences for controlling balanced partial scan circuits

to control tri-state buffers where indeterminate values during shift may damage the buffers. If none of the input boundary scan registers have a separate update flip-flop, then the kernels can be tested by the ITAPC in the *pauseDR* state for one or more clock cycles based on d, the sequential depth of the kernel(s) under test. If $d \leq 3$ then the normal transitions of the ITAPC from the *shiftDR* to the *captureDR* state (through *exit1DR, updateDR* and *selectDR*) will be sufficient to propagate the results to the scan registers. Then the results are loaded into scan registers in the *captureDR* state. Note that since the intermediate TAP states are "dead" states, the non-scan functional registers with load/hold lines will be kept in the load mode of operation. Fig. 5.2(c) shows the sequence of activation of the ITAPC states that are used for applying these tests.

If any of the input boundary scan registers have an update flip-flop the control values will not be valid until the *updateDR* state. Thus applying multiple clocks in the *pauseDR* is no longer a viable option. In this case, the ITAPC has to remain in the *run-test/idle* state for $d - 1$ clock cycles. Since the ITAPC transits through the *selectDR* state on its way to the *captureDR* state, an additional clock cycle is obtained. This results in a total of d cycles being applied. Since the *run-test/idle* (and *selectDR*) state is a "dead" state for scan designs, all scan registers will in the hold mode and all non-scan registers will be the load mode. Fig. 5.2(d) shows the states of the ITAPC that are activated in applying tests for the case where some of control lines are not valid until *updateDR*.

The number of clock cycles that the ITAPC spends in either the *pauseDR* state or the *run-test/idle* state is controlled by the MMC. Note that during the application of these scan tests all registers are driven by the TCK signal. In an ideal scan test scenario, FCK should be used to load test results so that some delay faults can be detected. However since *shiftDR* and *captureDR* states are separated from one another by at least three intermediate states, the one cycle normal mode clock application is not possible.

5.2.3 Controlling Unbalanced Partial Scan TDM

Fig. 5.3(b) is the generic test plan for the unbalanced partial scan design shown in Fig. 5.3(a). This is the most general form of partial scan. In state $s1$ of the test plan the non-scan registers are in hold mode while the scan registers are in shift mode. In state $s2$ all registers are in load mode. For balanced partial scan TDMs the state of non-scan registers during shift is immaterial, but for unbalanced partial scan TDMs these registers must hold during shift mode.

Combinational logic is needed to control the load/hold control lines for non-scan registers. Since the control lines are either PIs (hence are boundary scanned) or are outputs of a functional controller (cut by a boundary scan type register) we have a simple mechanism whereby the load/hold control line is controlled by augmenting the 1-bit boundary scan register with an AND gate. Implementations of augmented 1-bit boundary scan registers are presented in Figs. 5.4(a) and (b), where an AND gate is inserted between the output of the scan (update) flip-flop and the output multiplexer. A single flip-flop boundary scan register has only a *scan* flip-flop whereas a double flip-flop boundary scan register has both a *scan* and an *update* flip-flop. One input of the AND gate is connected to the output of the flip-flop whereas the other input is connected to the control line s_1 of the local decoder. s_1 is a 1 in *captureDR* and is a 0 otherwise. Thus when the ITAPC is in *shiftDR* and the dead states, a 0 is asserted on the load/hold line of the non-scan registers. This keeps these registers in the hold mode. At the end of the shift (for single flip-flop boundary scan registers) or after *updateDR* (for two flip-flop boundary scan registers) we ensure that a 1 is present in the scan or update flip-flop. Since s_1 is a 1 in *captureDR* and a 1 is also present in the scan or update flip-flop, the non-scan registers are set to the load mode. Note that a 0 may be also shifted in the scan or update flip-flop and applied to the load/hold line to check for a stuck-at 1 on this line.

The presence of dead states between *shiftDR* and *captureDR* complicates the control problem. The hardware modification to the boundary scan registers described above solves the control problem of the non-scan registers that have load/hold control. However the problem of controlling non-scan registers that have no load/hold control needs to be addressed. First a hold mode is added to these registers. The load/hold control for all such registers is driven by combinational logic which implements the following logic function:

$$f = SAMPLE + BYPASS + captureDR \tag{5.1}$$

Definition 5.3 *A* **session test plan** *is composed of the test plans for a set of kernels that can be tested concurrently.*

A session boundary is characterized by shifting out (in) the results (seeds) for the old (new) session. A kernel can have multiple test plans executing in different sessions. For example, a memory structure (RAM/ROM/CAM) can be tested in multiple test sessions, where each session corresponds to one of the steps in a memory test algorithm. The shift-in/shift-out process at session boundaries not only carries data to initialize registers but also control information to mode and configuration flip-flops. Kernels can have unequal test lengths. The optimal scheduling of unequal test length kernels have been addressed by various researchers. Tests for kernels that have large test lengths may be split across multiple sessions. Thus when a shift out is performed, the state of the PGs and SAs for kernels that have not completed their test process is stored and then restored when registers for the next session are initialized.

In BIST designs we may incorporate an on-chip test counter. In Section 5.4 we show that it is advantageous to incorporate this on-chip test counter for certain configurations of chips on a board. This counter is on the scan chain and is initialized at the beginning of a session. The counter starts down-counting in the *run-test/idle* state and asserts a signal *test_complete* on terminal count. The *test_complete* signal is an input to the ITAPC and the *run-test idle* state is transmitted on the ITB when *test_complete* is not true and "dead" is transmitted otherwise. All BIST registers perform their appropriate functions when the RBIST instruction is loaded and the *run-test/idle* state is transmitted on the internal test bus. When *test complete* is asserted, all BIST registers go to the hold mode. The (test channel in the) MMC services the chip after a certain interval. *test_complete* can additionally be used to interrupt the MMC to process the on-chip data. In the *run-test/idle* state, the test counter, BIST registers and all other functional registers are driven by FCK. Fig. 5.6 illustrates an ITAPC with a test counter. Combinational logic is used to detect all zeros and generate the *test_complete* signal. C_2 generates the control signals for the test counter.

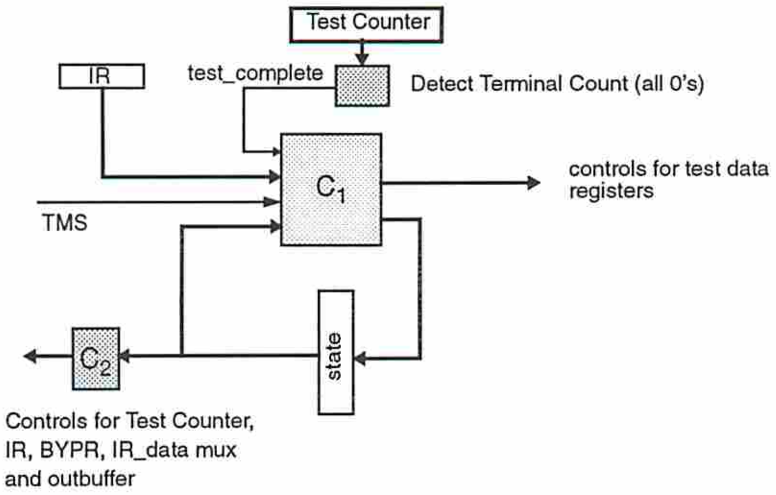

Test Counter
IR
test_complete
Detect Terminal Count (all 0's)
C$_1$
controls for test data registers
TMS
state
C$_2$
Controls for Test Counter, IR, BYPR, IR_data mux and outbuffer

Figure 5.6: Test counter incorporated in BIST designs

5.4 Overview of the Test Process for Multiple Chips on a Board

We assume that a number of chips with boundary scan hardware will be present on a board. These chips can be connected in a number of configurations to an on-board MMC. In the simplest configuration all the chips are connected in a single ring (i.e., daisy chained with the TDO of one chip connected to the TDI of the other) and there is only one TMS and TCK signal connected to all the chips. This case corresponds to the situation where the MMC has only one test channel. More complex schemes are possible where the chips can be connected in two or more rings with a TMS line servicing each of the rings. If the TDI is common to all the rings then the resulting configuration can still be handled by the test channel described in [10]. Separate TDI lines to each of the rings can only be accommodated when multiple test channels are present in the MMC.

Figs. 5.7(a) and (b) show examples where a number of chips on a board are connected in different configurations. Fig. 5.7(a) shows n chips connected in a single ring. The TMS and TCK lines are common to all chips. Fig. 5.7(b) shows a number of chips connected in a number of rings. All the chips in a ring are driven by different TMS line. The test channel in [10] drives a number of TMS lines. However only one TMS line can be actively driven by the FSM in the test channel at a time. All the other TMS lines are held at specific values. We assume a single test channel in the MMC. Thus the number of rings are limited by the number of TMS lines available from a test channel. The TDO of the test channel is fanned out to the TDI's of the first chip in each ring. The TDO's of each of the rings are connected together and provided as input to the TDI of the test channel. This is possible because the TDO of a chip is driven by a tristate buffer which is active only when the chip is in the shift mode (i.e, ITAPC in the *shiftDR* state). Thus chips in two rings cannot be simultaneously in the shift mode. In the following we will first focus on the single ring configuration and show how the chips are tested via the test channel. Then we will briefly describe how chips are tested in the multiple ring configuration.

5.4.1 Chips Connected in a Single Ring

5.4.1.1 Mandatory Boundary Scan Instructions

In the normal mode of operation, all chips have the BYPASS instruction loaded in their instruction registers. To test interconnect, the EXTEST instruction must be loaded in all (some) of the chips. This is accomplished by controlling the TMS signal such that the TAP controllers in all the chips move through the set of states related to shifting in instructions. Note that the TAP controllers for all the chips transit through exactly the same set of states at the same time. In the *shiftIR* TAP state all the chips are connected in one long scan chain and the appropriate instructions (EXTEST for some chips and BYPASS for others) are shifted in. Then the TMS line is controlled to shift in the appropriate test vectors to all relevant (those that have EXTEST loaded) chips. These vectors are used to test the interconnect and possibly some glue logic that does not have boundary scan. The EXTEST operation can be performed simultaneously on all chips.

To set one (or more) chips in the sample mode, the test channel controls the TMS line such that the ITAPC of *all* chips transit through the set of states related to shifting in instructions. Instructions are shifted in and at the end of the shift process only those chips that need to be in the sample mode contain the SAMPLE instruction and the other chips contain the BYPASS instruction.

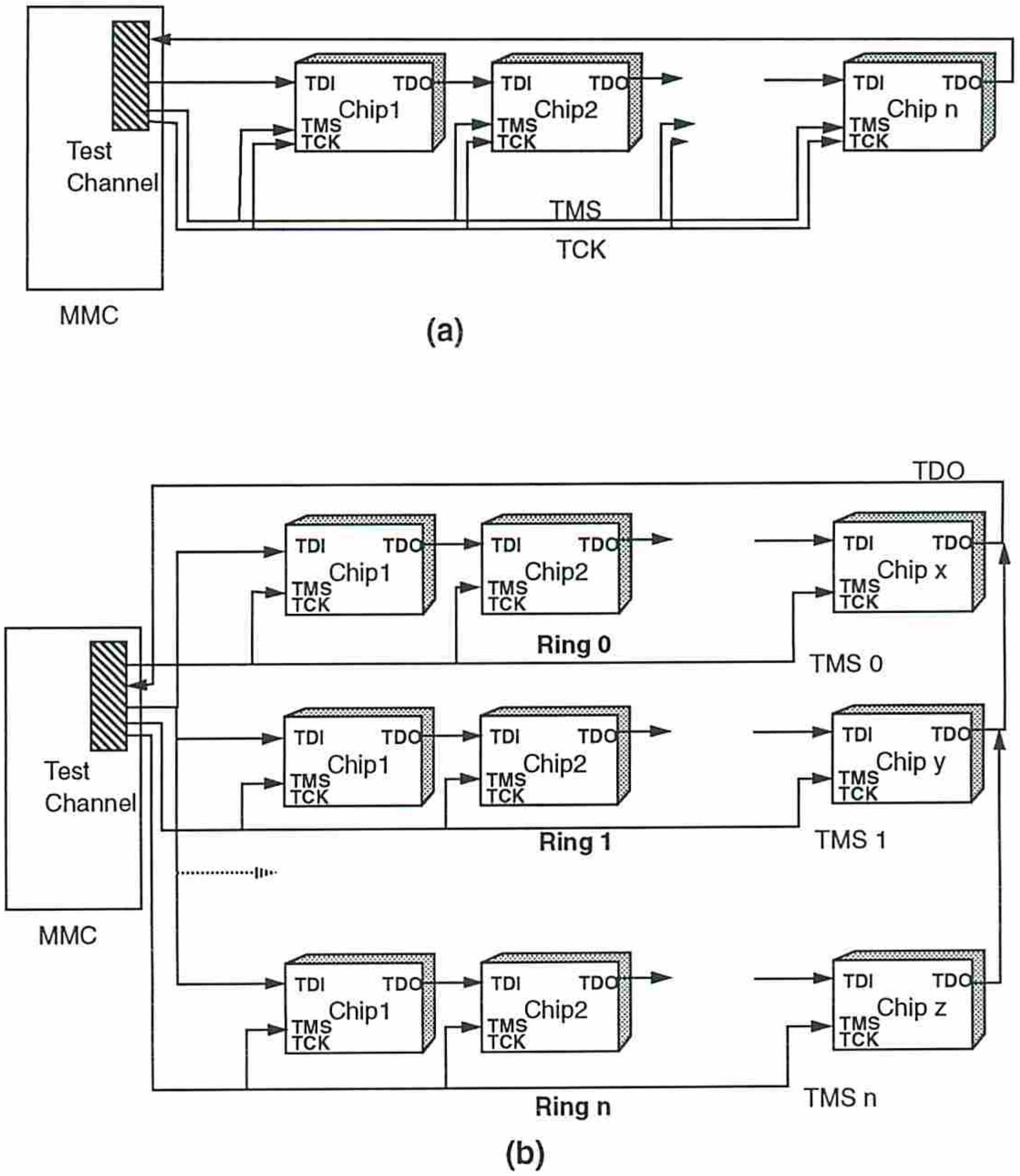


Figure 5.7: Chips on a board configured in (a) a single ring; (b) multiple rings

5.4.1.2 Controlling Scan Designs

If all chips have full scan capability then all chips can be tested simultaneously. By "simultaneous" we mean that a single shift operation is used to shift-in/out test vectors/results to/from all chips and the results of a test vector are captured in the same clock cycle in all chips. An instruction such as FSCAN is first loaded in all chips, then all chips are set in the shift mode such that test vectors are loaded in all chips. Then all ITAPC's move to the *captureDR* state such that one test vector is applied to all the chips. This process is repeated. If all vectors have been applied to a particular chip, then an instruction shift phase is entered, whereby the BYPASS instruction is loaded in this particular chip and the testing process resumes for all other chips.

Now consider the case where some of the chips have full scan capability and some of the chips have embeddings of the balanced partial scan TDM. The test vectors for all the chips can still be loaded in a single shift operation. The TMS line now has to be controlled (by keeping the ITAPC in the *pauseDR* state or the *run-test/idle* state) for sufficient clock cycles such that one test vector is applied to the chip with the largest sequential depth. This automatically applies one test vector to kernels of other chips with less sequential depth. Full scan designs can be viewed as special cases of balanced partial designs where the sequential depth is reduced to 0.

In fact any combination of chips with (embeddings of) full / balanced partial / unbalanced partial scan TDMs can be tested in the above manner. Consider two chips Chip1 and Chip2 on a board. Chip1 has an embedding of the balanced partial scan TDM with a sequential depth d while Chip2 has an embedding of the unbalanced partial scan TDM. After the shift phase is over a test vector is loaded in Chip1 and Chip2. TMS is controlled such that a test is applied to Chip1. Depending on the value of d, for Chip2 the test process appears as if additional *dead* states have been inserted between end of shift and the *captureDR* state. Thus the application of a test vector to Chip1 also results in the application of a test vector to Chip2. An alternate approach is to test the chips sequentially - each chip is tested separately while keeping the other chip in the bypass mode. It is easy to show that (ignoring the time required to set each of the chips in the bypass mode) testing the two chips together (*simultaneous* test application) until all tests have been applied to one of the chips will always be less expensive in terms of test time as compared to the case where the two chips are tested sequentially.

5.4.1.3 Controlling BIST Designs

In this case we assume that all chips on the board have BIST capability.

Case 1 : *None of the chips have a test counter*

Consider two chips in a board, Chip1 and Chip2. The test sessions of each of the chips are shown in Figs. 5.8(a) and (b). The test times t_1, t_2, t_3 and t_4 represent *number* of clock cycles. If Chip1 is tested on its own then the test channel first shifts in seeds to initialize the PG/SAs for kernels a and b. Then the ITAPC of Chip1 is set in the *run-test/idle* state for t_1 clocks. The *Test Counter (TC)* in the Test Channel [10] is used to keep track of the t_1 clock cycles to be applied. At the end of this period the complete signature for kernel b and a partial signature and state of the pattern generator for kernel a is shifted out. A shift-in is then conducted that restores the states of the PG/SAs for kernel a and initializes the PG/SAs for kernel c. Chip1 is then set in the *run-test/idle* state for $t_2 - t_1$ clocks. Chip2 can be tested in a similar manner. A complete description of the test channel can be found in [10].

However when the two chips are connected in a single ring on a board then there are 5 shift boundaries instead of the 3 individually for each of the chips (see Fig. 5.8). At each of the boundaries the signature for all kernels that have completed test are shifted out. The state of each PG/SA for kernels whose test are not completed are restored and seed data for PG/SAs for kernels whose test is to be started are shifted in.

Case 2 All chips have a test counter

If there is only one chip on a board with a dedicated test channel driving it, then the on-chip test counter is not necessary. This is because the test channel can keep track of the number of test vectors applied and stop the test when all patterns have been applied.

Again consider Chip1 and Chip2 on a board and assume that both have test counters. Since $t_3 < t_1$, (see Fig. 5.9) the test counter for Chip2 can be loaded with the value t_3. Both the chips are then set in the *run-test/idle* state and BIST tests are applied. Chip2 holds after t_3 cycles. The test channel however allows t_1 clocks

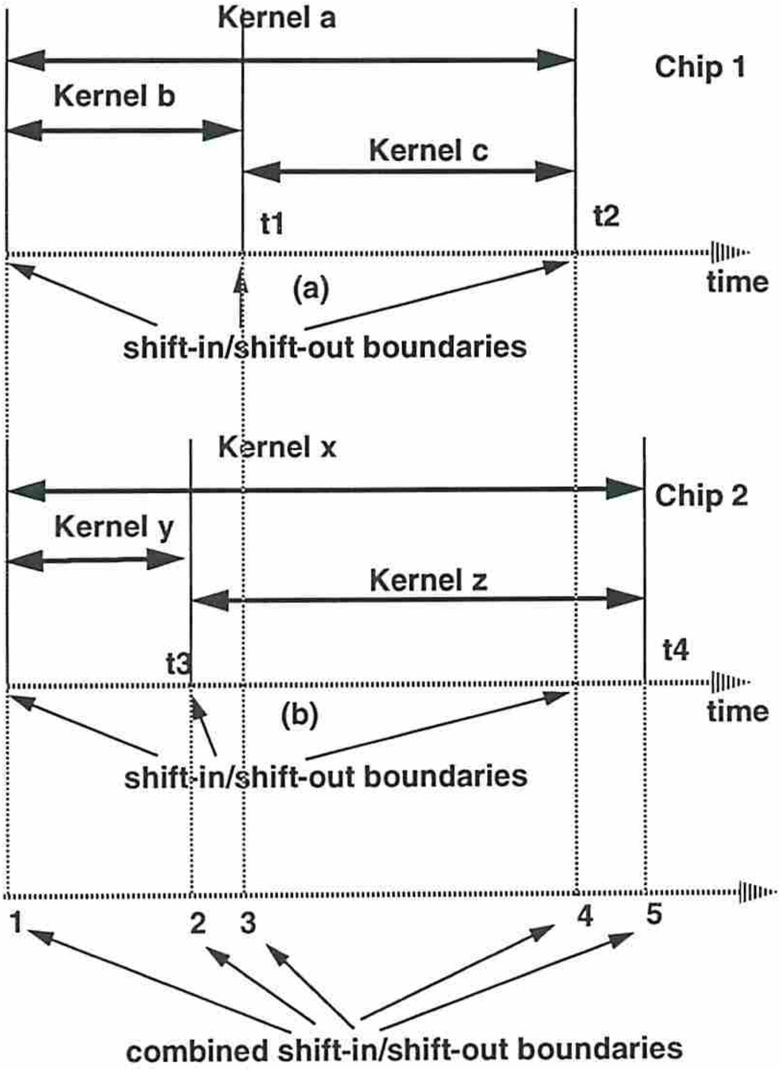

Kernel a
Chip 1
Kernel b
Kernel c
t1
t2
time
(a)
shift-in/shift-out boundaries
Kernel x
Chip 2
Kernel y
Kernel z
t3
t4
time
(b)
shift-in/shift-out boundaries
1
2
3
4
5
combined shift-in/shift-out boundaries

Figure 5.8: Test sessions and shift-in/shift-out (session) boundaries for (a) Chip1 by itself; (b) Chip2 by itself; (c) combined session boundaries for Chip1 and Chip2 connected in a single ring

to elapse (using its Test Counter) before servicing the chips. Then the test counter for Chip1 is loaded with the value $(t_2 - t_1)$. Both the chips are again allowed to run in the *run-test/idle* state. Chip1 holds when its counter reaches its terminal count. However the test channel only services the chips after the test for Chip2 is complete. Figs. 5.9(a) and (b) show the modified session boundaries for Chip1 and Chip2, respectively. This example shows how the use of test counters enables the alignment of session boundaries and therefore reduces the number of times shift-in/shift-out is needed. There is a tradeoff between the number of shift-in/shift-outs required and the test application time. Reducing the number of shift-in/shift-outs reduces the amount of support needed from the test channel. This in turn reduces the amount of support needed by the test channel from the MMC.

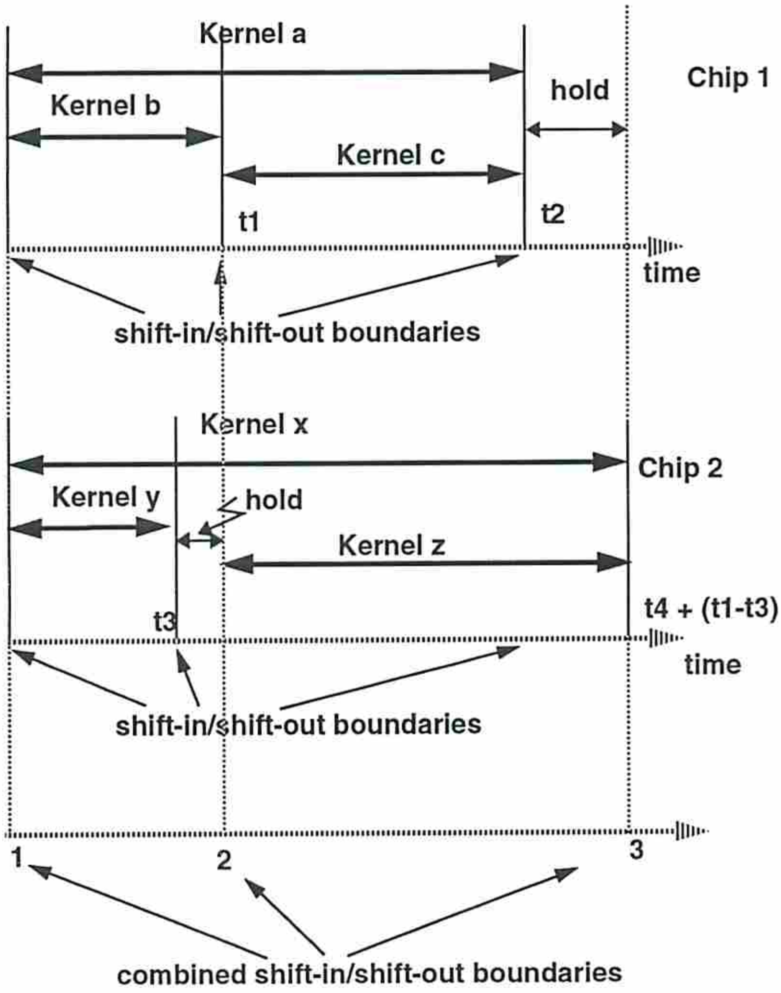


Figure 5.9: Chips on a board with test counters (a) modified session boundary for Chip1; (b) modified session boundary for Chip2; (c) combined shift-in/shift-out boundaries for the two chips with test counters used to align boundaries

5.4.2 Chips Connected in Multiple Rings

5.4.2.1 Controlling Scan Designs

The multiple ring model with a single test channel driving multiple TMS lines via buffers has no advantages over the single ring model for scan designs. This is because the TMS line needs full support from the test channel for scan testing. Therefore even if there are multiple rings, the test channel can only service one ring at a time.

5.4.2.2 Controlling BIST Designs

The advantage of the multiple ring model of connecting chips has advantages for the BIST case. The test channel can initiate the test for one ring by loading up the seeds for all the chips and the test counters with appropriate values for each of their sessions and sets all chips in the *run-test/idle* state. The test channel then starts servicing another ring. The advantage of this approach over a single ring case is that to service one chip in a ring, only the chips in that ring have to be disturbed. Chips on other rings can continue their tests.

5.5 Summary

In this chapter we presented techniques for controlling specific on-chip scan and BIST TDMs. These techniques operate within our test control framework. We also described two board level configurations for controlling a set of testable chips and showed how the on-chip test control mechanisms can be controlled from a module maintenance controller.

Chapter 6

Merging Test Plan Controllers

6.1 Introduction

In Section 5.3 of Chapter 5 we showed an example datapath with a number of TDMs embedded in the circuitry. The test plans corresponding to these embeddings were also given. Each of these test plans requires a test controller. In this chapter we focus on an efficient approach to implementing these controllers.

Fig. 6.1(a) shows three test controllers that control the body of three session test plans. As defined in Section 5.3, a session test plan describes the test control activation sequence of a set of kernels that can be tested concurrently. In the rest of this chapter we will refer to session test plans as simply test plans. The outputs of the controllers in Fig. 6.1(a) are multiplexed and the select line of this multiplexer (*t_mux*) is controlled by a *session register*. This session register is part of the scan chain and is initialized with a binary string corresponding to the test session under execution. A chip usually has a functional controller that controls various datapath elements. Since the functional controller and the test controllers control a common set of datapath elements, the outputs of *t_mux* is again multiplexed with the outputs of the functional controller. The select line of this multiplexer (*tf_mux*) is controlled by a test mode signal (*TM*). This signal hands over control of the datapath to the functional controller in normal mode and to the test controllers in test mode. TM is generated by a decoder that decodes information from the test bus and is able to distinguish between test and normal modes from the information transmitted over the bus.

State transition graphs (STGs) of the various controllers are provided for illustrative purposes only. The STGs of the test controllers have a very specific structure, i.e., there is only one outgoing arc (edge or transition) from every state and only one incoming arc into every state. Fig. 6.1(b) shows a test controller that is formed by merging the three separate test controllers in Fig. 6.1(a). *t_mux* is no longer needed to select among the different test controllers and the output of the session register serves as the primary inputs to the merged test controller. Finally, Fig. 6.1(c) illustrates the case where the merged test controller is combined with the functional controller [53]. Note that now *tf_mux* is also eliminated. The merged functional and test controller has two sets of inputs, one from the session register and the other set is the primary inputs to the functional controller.

In this chapter we focus on the problem of efficiently merging a number of test controllers. This problem is stated as follows. *Given* n *FSMs, $M^1, M^2, \ldots, M^n$, controlling the body of* n *test plans, obtain a minimal state merged test controller, M^m, that (after performing state assignment and logic minimization) has minimal implementation cost in terms of product terms.* These controllers have the following properties:

- Single fan-in into and single fan-out of every state.
- The machines are temporally disjoint, i.e., only one machine is active at a time.

We solve this merging problem by sequencing the test controllers in non-increasing order of the number of their states and then optimally merging machines one at a time to an intermediate merged machine. The intermediate merged machine, M^{m_j}, is built by the merger of the first j test controllers, where $1 \leq j < n$. We have developed an A^* algorithm that merges the $j+1$st test controller with M^{m_j} such that the number of implicants (product-terms) in a two-level logic implementation of $M^{m_{j+1}}$ is minimal.

This chapter is divided into eight sections. Section 6.2 walks though a simple example to show that current state minimization techniques are inadequate for the test controller merging problem. Section 6.3 summarizes symbolic minimization techniques. Section 6.4 presents lower and upper bounds on the implementation cost of merged machines. Section 6.5 formally introduces our cost function and outlines the merger technique. Section 6.6 provides a detailed description of the A^*

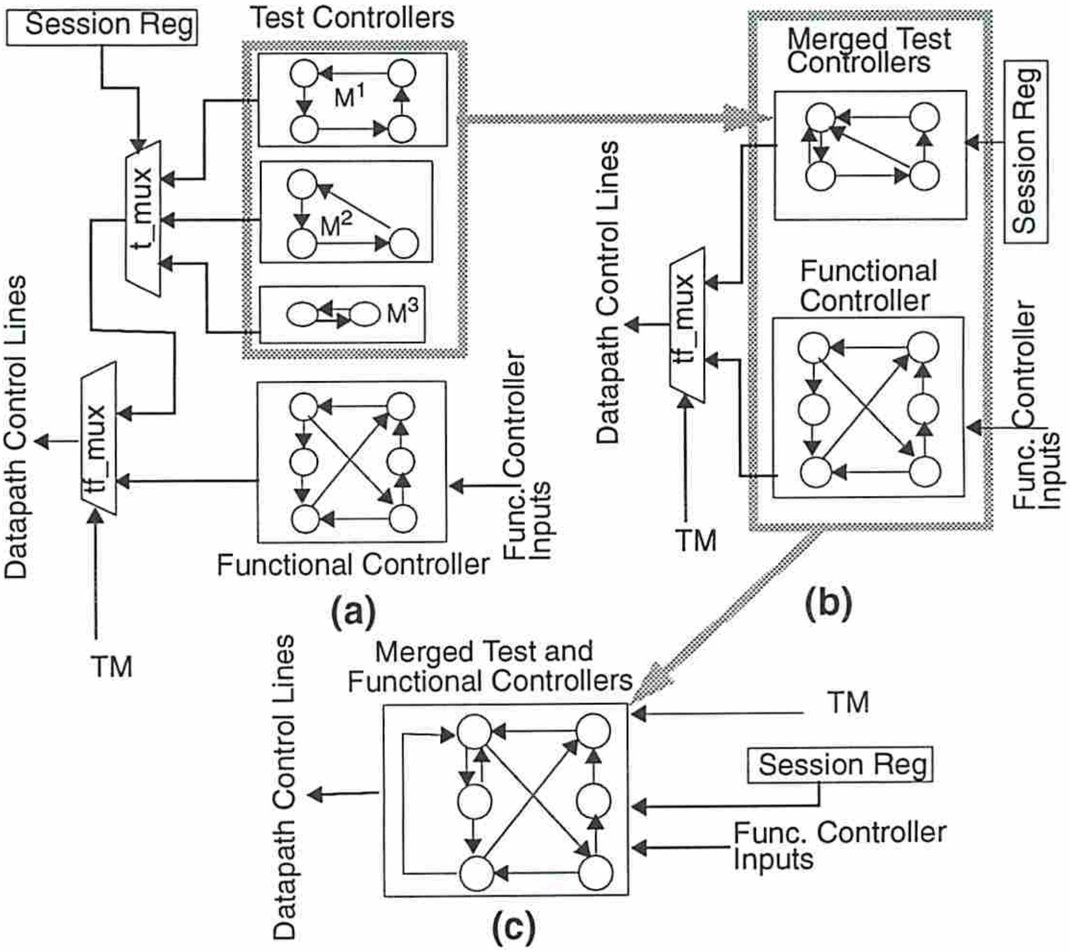


Figure 6.1: Various mechanisms for controlling a testable datapath

algorithm used to merge a pair of machines. Finally, Sections 6.7 and 6.8 present results and a summary of this chapter, respectively. An abridged version of this chapter appears in [54].

6.2 Motivation

A FSM can be represented by two equivalent structures, a *State Transition Graph (STG)* and a *State Transition Table (STT)*. Let $I = \{i_0, i_1, \ldots, i_{p-1}\}$ and $O = \{o_0, o_1, \ldots, o_{r-1}\}$ represent the set of primary inputs and outputs of n FSM test controllers, $M^1, M^2, \ldots, M^n$. The $p \geq \lceil log_2 n \rceil$ input lines (variables) are common to all the FSMs. The input lines are assigned values 0 or 1 to represent n input symbols (one symbol per FSM). $S^i = \{s^i_0, s^i_1, \ldots, s^i_{l_i-1}\}$ represents the set of states of machine [illegible] l_i is the number of states in machine M^i. In general, a FSM can be defined [illegible] where $\delta : I$ x $S \rightarrow S$ and $\lambda : I$ x $S \rightarrow O$ are the next state [illegible] ly. If the test controllers are implemented separately [illegible] then each machine is implemented as [illegible] state transition depends only [illegible] are ordered such

algorithm used to merge a pair of machines. Finally, Sections 6.7 and 6.8 present results and a summary of this chapter, respectively. An abridged version of this chapter appears in [54].

6.2 Motivation

A FSM can be represented by two equivalent structures, a *State Transition Graph (STG)* and a *State Transition Table (STT)*. Let $I = \{i_0, i_1, \ldots, i_{p-1}\}$ and $O = \{o_0, o_1, \ldots, o_{r-1}\}$ represent the set of primary inputs and outputs of n FSM test controllers, $M^1, M^2, \ldots, M^n$. The $p \geq \lceil log_2 n \rceil$ input lines (variables) are common to all the FSMs. The input lines are assigned values 0 or 1 to represent n input symbols (one symbol per FSM). $S^i = \{s^i_0, s^i_1, \ldots, s^i_{l_i-1}\}$ represents the set of states of machine M^i, where l_i is the number of states in machine M^i. In general, a FSM can be defined as a 5-tuple (I, S, O, δ, λ), where $\delta : I$ x $S \rightarrow S$ and $\lambda : I$ x $S \rightarrow O$ are the next state and output functions, respectively. If the test controllers are implemented separately and their outputs multiplexed (Fig. 6.1(a)), then each machine is implemented as an *autonomous* FSM, i.e., for any machine the next state transition depends only on the present state. We assume that machines, $M^1, M^2, \ldots, M^n$, are ordered such that $l_1 \geq l_2 \geq \ldots \geq l_n$.

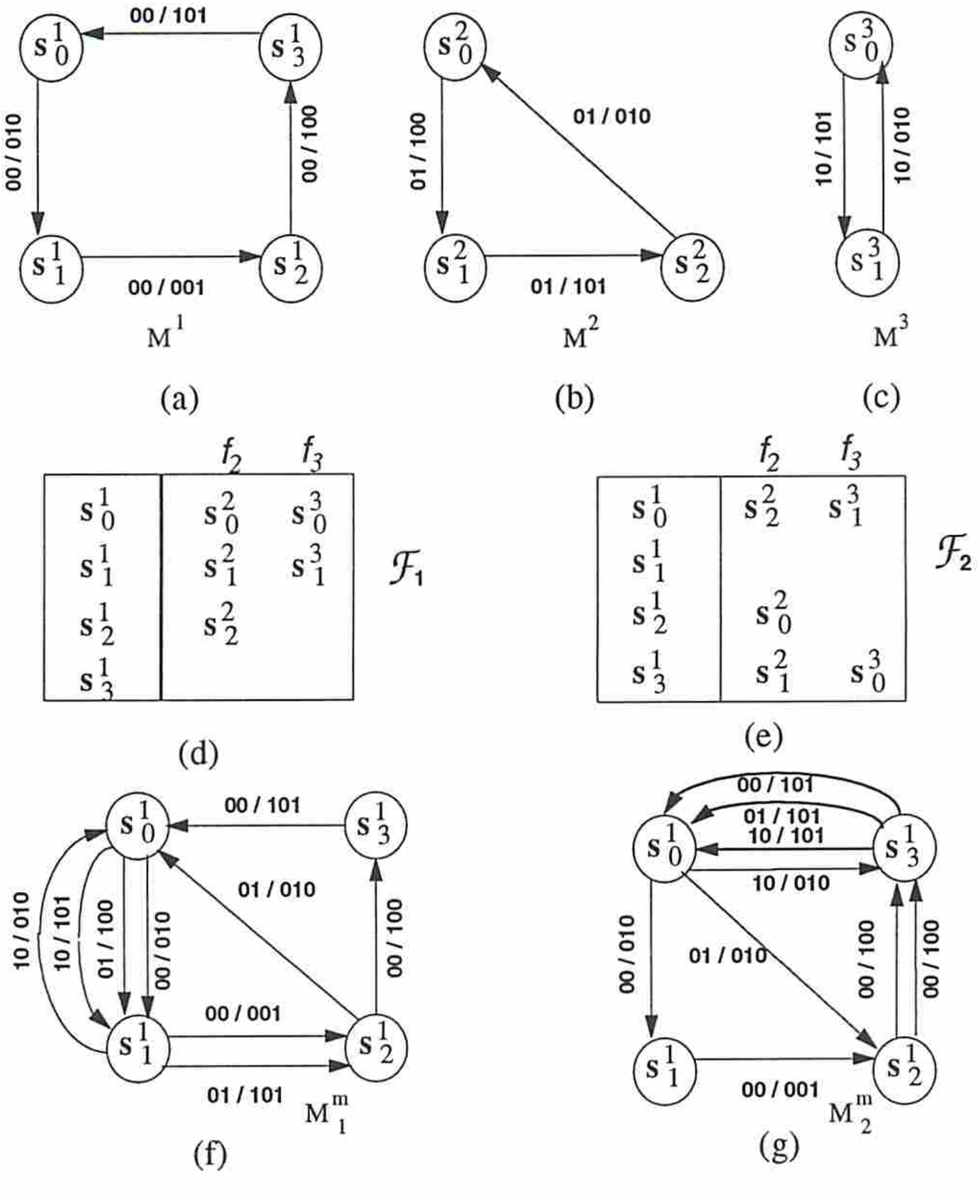
In general, the merger of n machines with $l_1, l_2, \ldots, l_n$ states produces as many as $l_1 * l_2 * \ldots * l_n$ states prior to state minimization. However, the FSMs that we target for merger are orthogonal in the sense that only one machine is active at a time and the input values are constant throughout the activation of a particular machine. State minimization becomes trivial for this case, since any state of one machine can be merged with any state of another machine using an input symbol to condition any outputs that are incompatible and to specify the next state transition. The number of states in the merged machine is thus equal to l_1, which is the number of states of the largest machine, M^1. The merged test controller state flip-flops as well as the session register are part of a scan chain. Making the controller flip-flops scannable enables us to (1) initialize the controller to any state, and (2) provide an effective solution to testing the controller itself [55]. A simple scheme for testing the controller is the *Staggered Self Observation* technique [55], where controller outputs

00 s_0^1 s_1^1 010	01 s_0^2 s_1^2 100	10 s_0^3 s_1^3 101
00 s_1^1 s_2^1 001	01 s_1^2 s_2^2 101	10 s_1^3 s_0^3 010
00 s_2^1 s_3^1 100	01 s_2^2 s_0^2 010	
00 s_3^1 s_0^1 101		
M^1	M^2	M^3

Table 6.1: *Demo*

are partitioned into a number of groups and the groups are observed one at a time in the controller flip-flops and session register.

The merger of n orthogonal machines is the process of determining a mapping vector $\mathcal{F}$ consisting of n-1 components $f_2, f_3, \ldots, f_n$. Each component f_i defines a mapping between states of machines M^i and M^1. Any mapping $\mathcal{F}$ will produce a minimal state machine. However, the minimality of next state and output logic is not guaranteed. Consider the example shown in Table 6.1 which shows the STTs of three machines M^1, M^2 and M^3. This trio of machines will be referred to as *Demo* throughout this chapter. The inputs to all three machines are i_0 and i_1, and the outputs are o_0, o_1 and o_2. Input symbol 00 (01, 10) is present in all rows of the STT of M^1(M^2, M^3). These STTs represent incompletely specified machines, because transitions out of a state for some input symbols are left unspecified. Note that if each of the machines is implemented separately, then it does not matter whether the inputs of some machine M^i are left unspecified (don't cares) or set to (symbol) i, because a boolean (or multiple-valued) cover of each of the machines obtained by synthesis tools will set the input bits to don't cares. In general, some of the outputs can be don't cares. The STGs of the three machines are shown in Figs. 6.2(a), (b) and (c). Now consider the case where the three machines are merged into one machine. Let $\mathcal{F}_1$ and $\mathcal{F}_2$ represent two mapping vectors which lead to two different merged machines M_1^m and M_2^m. $\mathcal{F}_1$ maps states s_0^2, s_1^2, s_2^2 to s_0^1, s_1^1, s_2^1 and states s_0^3, s_1^3 to s_0^1, s_1^1. $\mathcal{F}_2$ maps states s_0^2, s_1^2, s_2^2 to s_2^1, s_3^1, s_0^1 and states s_0^3, s_1^3 to s_3^1, s_0^1. These mappings are shown in Figs. 6.2(d) and (e). The STGs of the two merged machines are shown in Figs. 6.2(f) and (g) and the corresponding STTs of the two merged machines are given in Table 6.2(a) and (b).



	f_2	f_3
s^1_0	s^2_0	s^3_0
s^1_1	s^2_1	s^3_1
s^1_2	s^2_2	
s^1_3		

$\mathcal{F}_1$

(d)

	f_2	f_3
s^1_0	s^2_2	s^3_1
s^1_1		
s^1_2	s^2_0	
s^1_3	s^2_1	s^3_0

$\mathcal{F}_2$

(e)

Figure 6.2: STGs of M^1, M^2, M^3 and some merged machines

00	s_0^1	s_1^1	010
01	s_0^1	s_1^1	100
10	s_0^1	s_1^1	101
00	s_1^1	s_2^1	001
01	s_1^1	s_2^1	101
10	s_1^1	s_0^1	010
00	s_2^1	s_3^1	100
01	s_2^1	s_0^1	010
00	s_3^1	s_0^1	101

(a) M_1^m

00	s_0^1	s_1^1	010
01	s_0^1	s_2^1	010
10	s_0^1	s_3^1	010
00	s_1^1	s_2^1	001
00	s_2^1	s_3^1	100
01	s_2^1	s_3^1	100
00	s_3^1	s_0^1	101
01	s_3^1	s_0^1	101
10	s_3^1	s_0^1	101

(b) M_2^m

Table 6.2: Merged machines for *Demo*

00	s_0^1	s_1^1	010
00	s_1^1	s_2^1	001
00	s_2^1	s_3^1	100
00	s_3^1	s_0^1	101
01	s_0^2	s_1^2	100
01	s_1^2	s_2^2	101
01	s_2^2	s_0^2	010
10	s_0^3	s_1^3	101
10	s_1^3	s_0^3	010

(a) Input to STAMINA

00	s_0^1	s_1^1	010
01	s_0^1	s_1^1	100
10	s_0^1	s_1^1	101
00	s_1^1	s_2^1	001
01	s_1^1	s_2^1	101
10	s_1^1	s_0^1	010
00	s_2^1	s_3^1	100
01	s_2^1	s_0^1	010
10	s_2^1	s_0^1	010
00	s_3^1	s_0^1	101
01	s_3^1	s_0^1	010
10	s_3^1	s_0^1	010

(b) Output of STAMINA

Table 6.3: (a) Machine descriptions provided to STAMINA; (b) merged machine description obtained from STAMINA

Minimum cost two-level implementations of M_1^m and M_2^m using a 1-hot coded state assignment have 9 and 6 product terms, respectively. A state assignment with minimum number of flip-flops [43] results in 9 and 5 product terms for M_1^m and M_2^m, respectively. NOVA [43] and Espresso [48, 49] are used for state assignment and logic minimization, respectively. $\mathcal{F}_2$ is therefore a better mapping.

The STT of a merged machine can also be obtained by using a state minimization tool such as STAMINA [28]. Given a machine that does not have a minimum number of states, STAMINA provides the option of using exact (heuristic) techniques for obtaining a minimum (minimal) closed set of compatibles that covers all the states of the machine. STTs of M^1, M^2 and M^3 are concatenated together as shown in *Table 6.3*(a) and provided as input to STAMINA. This description corresponds to an incompletely specified state machine with 9 states. STAMINA produces a state minimal machine with 4 states which is equal to the number of states of both M_1^m and M_2^m. An analysis of the merged machine produced by STAMINA, shown in Table 6.3(b), reveals a close resemblance to M_1^m. We also observe that additional transitions for states s_2^1 and s_3^1 have been specified. For example, rows 10 s_2^1 s_0^1 010, 01 s_3^1 s_0^1 010 and 10 s_3^1 s_0^1 010 are clearly unnecessary. The effect of specifying additional transitions is to potentially increase the cost of the FSM synthesized by STAMINA over the straightforward mapping $\mathcal{F}_1$. The number of product terms for 1-hot encoding is 9 but the number of product terms using 2 flip-flops (minimum length state assignment) is 10. Comparing the implementation costs of STAMINA with that of M_2^m, we see that STAMINA produces a merged machine with 50% and 100% more product terms for a 1-hot coded and minimum length state assignment, respectively. The implementation costs for merged machines is summarized in Figs. 6.3(a) and (b).

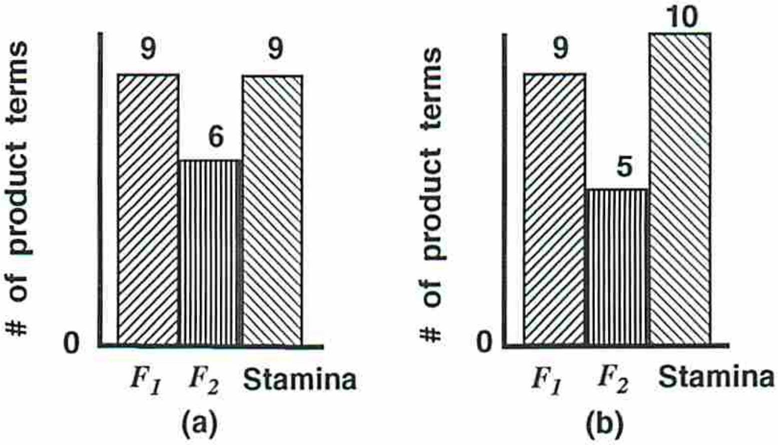


Figure 6.3: Implementation costs of merged machines obtained by different mappings and by STAMINA (a) 1-hot code used for state assignment; (b) minimum number of bits used for state assignment

We have seen that different mappings influence the implementation cost of the merged test plan controllers. It has also been shown that an existing popular tool is inadequate for the test plan controller merging problem. The number of possible merged machine is $\prod_{j=2}^{n} l_1!/(l_1 - l_j)!$. It is impractical to exhaustively generate all merged machines, synthesize them and then pick the best merger. In this chapter we present a technique to find a merged machine which leads to a minimal implementation cost.

In the next section, we review symbolic minimization techniques since these techniques are used for state assignment and input encoding of FSMs.

6.3 Symbolic Minimization

In FSM synthesis, logic minimization is usually performed both before and after state assignment. Before state assignment, logic minimization is performed on a symbolic (code independent) representation of the combinational component of the FSM : the *symbolic* cover. The concept of a symbolic cover is a generalization of the logic cover representation of combinational logic functions. A symbolic cover represents a symbolic function. In general, symbolic functions are switching functions whose variables take a finite set of values. Each value is represented by a word (or mnemonic). A symbolic cover consists of a set of symbolic implicants and a STT for a FSM is a collection of symbolic implicants, each of which can be represented as a 4-tuple $(\sigma_I \; \sigma_{ps} \; \sigma_{ns} \; \sigma_O)$, where each element is a string of characters. $(\sigma_I \; \sigma_{ps})$ represent the symbolic implicant input part and $(\sigma_{ns} \; \sigma_O)$ represent the symbolic implicant output part. σ_I can either represent the values of binary valued primary input variables or a symbolic primary input variable. If the FSM has p binary valued inputs, then σ_I has p components, each component $\sigma_{I,i}, i = 0, 1, \ldots, p-1$, takes values from the set $\{0, 1, -\}$. If the FSM has one n-valued symbolic input variable then $\sigma_I \subseteq V$, where V represents (in terms of strings of characters) the set of n values for the symbolic input variable. σ_O represents the values of binary valued primary output variables. σ_{ps}, $\sigma_{ns} \in S$ represent the present and next states, respectively, where S represents the states of a FSM in symbolic form.

In general, a symbolic implicant can represent a transition from one or more states to a next-state under some input conditions. Therefore, there exist several symbolic cover representations that are equivalent to each other. A minimum symbolic cover is one of minimum cardinality. Symbolic minimization consists of finding such a minimum symbolic cover, i.e., it is equivalent to determining a minimum sum-of-product representation which is independent of the encoding of the symbolic strings. The symbolic cover representation is related to a *multiple-valued logic* representation,

001 1000 0100 010	001 1000 0100 010
010 1000 0100 100	010 1000 0100 100
100 1000 0100 101	100 1000 0100 101
001 0100 0010 001	011 0100 0010 001
010 0100 0010 101	010 0100 0000 100
100 0100 1000 010	100 0100 1000 010
001 0010 0001 100	001 0010 0001 100
010 0010 1000 010	010 0010 1000 010
100 0001 1000 101	111 0001 1000 101
(a) M_1^m	(b) Minimum cover for M_1^m

Table 6.4: Initial and minimized multiple valued covers of M_1^m

of choosing the binary encodings, we will use symbols (represented by a positional cube notation) in the input field (σ_I) of each symbolic implicant and perform input encoding after multiple-valued minimization. Thus we perform multiple-valued minimization on a FSM cover where both the inputs and states are represented by the positional cube notation (1-hot coded). Tables 6.4 and 6.5 present the STTs of M_1^m and M_2^m, respectively, along with their minimum multiple-valued covers (obtained by Espresso in exact mode). Note that some of the symbolic implicants in a multiple valued cover may have a null entry in the next state field (σ_{ns}). For example, the implicant 010 0100 0000 100 in Table 6.4(b) has a null entry (0000) in the next state field.

In the next section, we present upper and lower bounds on the implementation cost of a merged machine.

6.4 Bounds on Implementation Cost

Let M^{m_j} be the merged machine obtained by merging the first j test controllers. Merging M^{j+1} with M^{m_j} consists of determining a mapping f_{j+1} of states of M^{j+1} to states of M^{m_j} such that the cost of implementing $M^{m_{j+1}}$, in terms of the number of implicants in its multiple-valued cover, is minimal. Since states of the merged machine are the same as the states of M^1, the domain of mapping f_{j+1} is comprised of the l_{j+1} states in M^{j+1} and the range is comprised of l_1 states. Merging M^{j+1}

001 1000 0100 010	001 1000 0100 010
010 1000 0010 010	010 1000 0010 010
100 1000 0001 010	100 1000 0001 010
001 0100 0010 001	111 0100 0010 001
001 0010 0001 100	111 0010 0001 100
010 0010 0001 100	111 0001 1000 101
001 0001 1000 101	
010 0001 1000 101	
100 0001 1000 101	
(a) M_2^m	(b) Minimum cover for M_2^m

Table 6.5: Initial and minimized multiple valued covers of M_2^m

to M^{m_j} consists of replacing a state, s_x^{j+1} in the implicants of M^{j+1} by $f(s_x^{j+1})$, where $f(s_x^{j+1})$ represents the state of M^{m_j} to which s_x^{j+1} is mapped. The modified implicants are added to the STT of M^{m_j} to create $M^{m_{j+1}}$. Thus a STT of $M^{m_{j+1}}$ has $\sum_{i=1}^{j+1} l_i$ implicants. Note that the subscript for f has been dropped for the sake of simplicity.

In the following discussion, M^m represents a machine formed by the merger of some j controllers, where $1 \leq j \leq n$. The STT of M^m can be represented by a set of implicants $\mathcal{P}$. The symbolic strings representing states in M^m are also the same as those representing states in M^1. Therefore, for an implicant of M^m, σ_{ps} and $\sigma_{ns} \in \{s_0^1, s_1^1, \ldots, s_{l_1-1}^1\}$. $\mathcal{P}$ can be split into l_1 slices or sets $\mathcal{P}_i, i = 0, 1, \ldots, (l_1 - 1)$. Every implicant in slice $\mathcal{P}_i$ has the same entry in the present state field, i.e., for every implicant $\sigma_{ps} = s_i^1$.

The following two theorems provide lower (upper) bounds on the number of implicants in a minimal (minimum) multiple valued cover of a specific merged machine, i.e., after a mapping vector $\mathcal{F}$ has been determined for merging the n test controllers.

Theorem 6.1 *Given a merged machine M^m, a lower bound on the number of implicants in a multiple-valued cover of M^m is $\sum_{i=0}^{l_1-1} k_i$, where k_i is the number of distinct next state symbols in $\mathcal{P}_i$.*

Proof : A paradigm for obtaining the minimum number of implicants that cover a logic function (exact minimization) consists of first determining all prime implicants of the function and then choosing a minimum set of these prime implicants to cover the function. Consider a particular slice $\mathcal{P}_i$ with k_i distinct next state symbols. This slice represents a symbolic function with the same number of input variables and binary valued output variables as the symbolic function representing the entire machine M^m. Additionally, this slice has k_i output variables representing the k_i distinct next state symbols. The number of implicants in this slice may be greater than k_i. This implies that there are some implicants with the same next state symbol. All implicants that have the same next state symbol have different input symbols, the same present state symbol and may or may not have the same values for the binary output variables. We determine all prime implicants of the function defined by $\mathcal{P}_i$. The input space corresponding to all present state symbols other than the current one is assumed to be in the OFF-set of the function. For each of the next state variables, there exists a prime implicant that covers its input cubes. These prime implicants may additionally cover part or all of the (input cubes of the) boolean output variables. Thus a cover for $\mathcal{P}_i$ can be represented by a set of $k_i + y$ prime implicants, k_i of which cover the k_i distinct output state variables and y implicants have a null entry in the next state field and cover the boolean output variables that are not covered by the k_i implicants.

For the general problem of multiple-valued minimization it is possible to cover two implicants in different slices that have the same next state symbol by the same implicant. This happens when there exist transitions from different states to the same next state under the same inputs. However, in merged machines created by merging test controllers, there will never exist two transitions that go to the same next state under the same input. That is, a prime implicant of a slice $\mathcal{P}_i$ that covers an output state variable s_j^1 is also a prime implicant of the entire machine and will cover all input cubes of s_j^1 which have s_i^1 as present state entries. Thus $\sum_{i=0}^{l_1-1} k_i$ is a lower bound on the number of implicants in a multiple valued cover of a machine M^m. This is a lower bound because $\sum_{i=0}^{l_1-1} k_i$ implicants may not be sufficient to cover all the boolean outputs. □

If $\sum_{i=0}^{l_1-1} k_i$ implicants cover all the binary valued output and next state variables of a merged machine then this lower bound is achievable. For example, for M_2^m, $\sum_{i=0}^{3} k_i$=6 (see Table 6.4(a)) and the number of implicants in a minimum cover of M_2^m is equal to this lower bound. Thus $\sum_{i=0}^{l_1-1} k_i$ is a good lower bound on the implementation cost of a merged machine. Fig. 6.4(a) shows the slices of M_2^m as well as the values of k_i for i=0,1,2 and 3. Fig. 6.4(b) shows the slices of M_1^m and values of k_i.

Theorem 6.2 *Given a merged machine M^m, an upper bound on the number of implicants in a minimum multiple-valued cover of a merged machine M^m is $\sum_{i=0}^{l_1-1} q_i$, where q_i is the number of implicants in a minimum multiple-valued cover*[1] *of $\mathcal{P}_i$.*

Proof : Minimizing the slices individually precludes the possibility of reducing the number of implicants that cover only boolean valued outputs. Consider the minimum covers, $\mathcal{M}_i$ and $\mathcal{M}_j$, of two slices $\mathcal{P}_i$ and $\mathcal{P}_j$, respectively. These slices may each have an implicant with the same primary output part, a null entry in the next state field and the same input (or input combination) in the primary input field. However, if the merged machine is considered as a whole, these two implicants may be covered by a single implicant. This implicant will have the same primary outputs and inputs as the two implicants being covered. Additionally, the present state field will contain the symbols of the present states corresponding to the two slices. Thus $\sum_{i=0}^{l_1-0} q_i$ is an upper bound on the number of implicants in a multiple-valued cover of M^m. □

For a merged machine, it may not be possible to cover the input cubes of binary valued outputs that have different entries in their present state field by one (or more) implicants. In this case the minimum implementation cost of a merged machine equals $\sum_{i=0}^{l_1-1} q_i$. For example, for M_1^m, $\sum_{i=0}^{3} q_i=9$ and for M_2^m, $\sum_{i=0}^{3} q_i = 6$ (see Figs. 6.4(a) and (b)). Minimum multiple-valued covers of M_1^m and M_2^m also have 9 and 6 implicants, respectively (see Tables 6.4(b) and 6.5(b)). Experimental results indicate this to be also true for a number of other examples. Therefore $\sum_{i=0}^{l_1-1} q_i$ is a good upper bound on the implementation cost of a merged machine.

In the next section we formally introduce our cost function.

6.5 Formulation of the Merging Problem

In merging M^{j+1} to M^{m_j} and obtaining a merged machine $M^{m_{j+1}}$, we define the following as our objective: *Determine a mapping f which minimizes $\sum_{i=0}^{l_1-1}(q_i^{m_{j+1}} -$*

[1] A minimum multiple-valued cover of slice $\mathcal{P}_i$ is obtained by minimizing $\mathcal{P}_i$ independently of the other slices using Espresso in the exact mode. The input space corresponding to all present state symbols other than s_i^1 is assumed to be in the OFF-set of the function represented by $\mathcal{P}_i$.

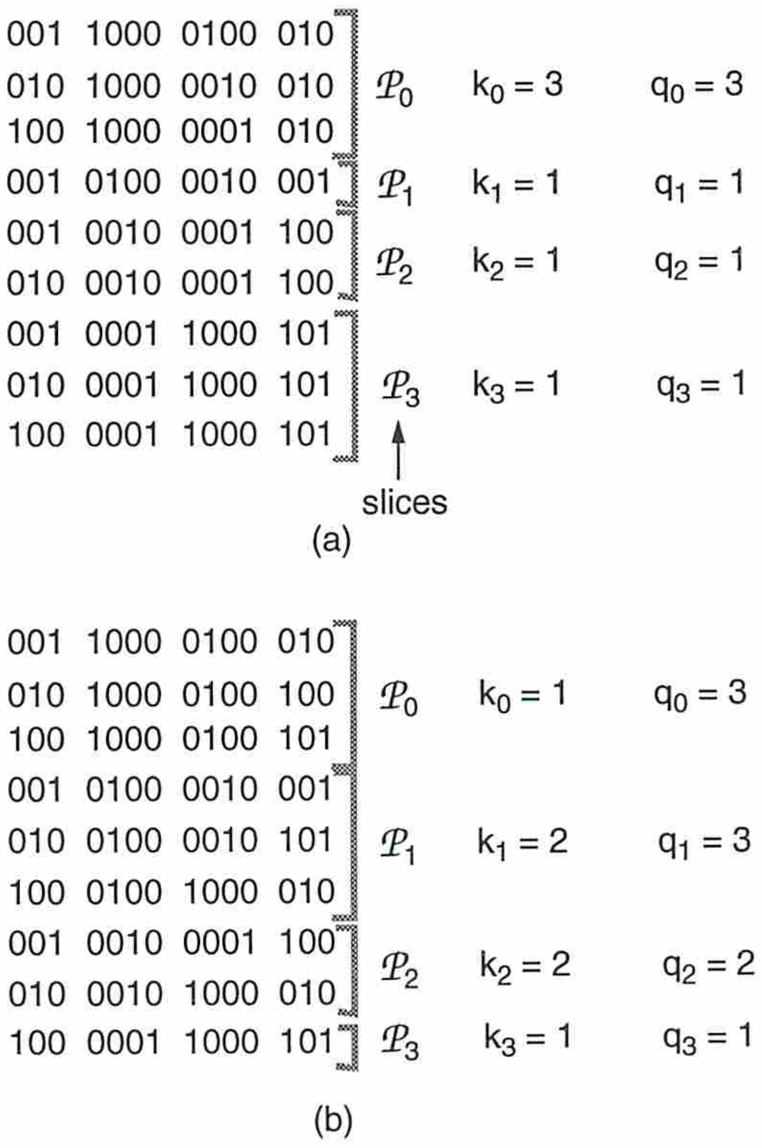


Figure 6.4: The number of implicants in minimal multiple valued covers and number of distinct next states in slices of merged machine (a) M_2^m (b) M_1^m

$q_i^{m_j}$) *where* $q_i^{m_{j+1}}$ *and* $q_i^{m_j}$ *are the number of implicants in minimum multiple valued covers of slices* $\mathcal{P}_i^{m_{j+1}}$ *and* $\mathcal{P}_i^{m_j}$, *respectively.* Note that since $M^{m_{j+1}}$ is formed by *adding* implicants of M^{j+1} to M^{m_j} and $\sum_{i=0}^{l_1-1} q_i^{m_j}$ is a constant, the above objective is also equivalent to minimizing $\sum_{i=0}^{l_1-1} q_i^{m_{j+1}}$.

Suppose the state s_x^{j+1} is mapped to s_i^1 under a mapping f. Let $\mathcal{NS}_i^{m_j}$ be the set of distinct next states in (the implicants of) $\mathcal{P}_i^{m_j}$ and $\mid \mathcal{NS}_i^{m_j} \mid = k_i^{m_j}$. The slice $\mathcal{P}_i^{m_{j+1}}$ is formed by concatenating the single implicant p_x belonging to the cover of M^{j+1} to $\mathcal{P}_i^{m_j}$. Present (next) state s_x^{j+1} (s_y^{j+1}) in p_x is replaced by s_i^1 ($f(s_y^{j+1})$). The following two propositions quantify the effects of different mappings on the minimum cover of a slice.

Proposition 6.1 *If* $f(s_y^{j+1}) \notin \mathcal{NS}_i^{m_j}$ *then* $q_i^{m_{j+1}} - q_i^{m_j} = 1$.

Proof : First we will prove that $q_i^{m_{j+1}} - q_i^{m_j} > 0$. In the cover of $\mathcal{P}_i^{m_{j+1}}$, one implicant will always be needed to cover the output variable representing the state $f(s_y^{j+1})$. In minimizing $\mathcal{P}_i^{m_j}$, the input space corresponding to primary input symbols $j+1, j+2, \ldots, n$ and present state s_i^1 is appended to the don't care set. The appended implicant p_x specifies that some of the boolean output functions are ON or OFF (instead of don't care) for the point in the input space corresponding to input symbol $j+1$ and state s_i^1. The net effect of the addition of the new implicant for the boolean valued outputs is a reduction in the don't care points available for minimizing $\mathcal{P}_i^{m_{j+1}}$. Therefore the number of implicants to cover the binary valued outputs and the outputs corresponding to $\mathcal{NS}_i^{m_j}$ next states (in minimizing $\mathcal{P}_i^{m_{j+1}}$) can never be less than $q_i^{m_j}$. In addition, one implicant will be needed to cover the output variable representing the state $f(s_y^{j+1})$ (since $f(s_y^{j+1}) \notin \mathcal{NS}_i^{m_j}$). Thus $q_i^{m_{j+1}} - q_i^{m_j} > 0$.

The additional implicant needed for covering $f(s_y^{j+1})$ also covers any boolean output that is ON for the input $j+1$ and the state s_i^1. The $q_i^{m_j}$ implicants in a cover of $\mathcal{P}_i^{m_j}$ can still be used to cover the $\mathcal{NS}_i^{m_j}$ next states and the rest of the ON-set of the boolean outputs. This is possible because the inputs are represented using the positional cube notation and any implicant which used input $j+1$ as a don't care point for a boolean output that is now OFF for $j+1$ need only drop a 1 from the $j+1$ th bit position in its input field, σ_I. Therefore $q_i^{m_{j+1}} - q_i^{m_j} = 1$. □

Note that if the primary inputs were not 1-hot coded, then the difference between $q_i^{m_{j+1}}$ and $q_i^{m_j}$ may be greater than 1. This is because $\mathcal{P}_i^{m_j}$ is minimized with the binary code corresponding to the input $j+1$ in the don't care set. When $j+1$ becomes part of the OFF or ON set for some of the binary outputs then the input cubes have to be made smaller and the number of additional implicants needed may be greater than 1.

Proposition 6.2 *If* $f(s_y^{j+1}) \in \mathcal{NS}_i^{m_j}$ *then* $q_i^{m_{j+1}} - q_i^{m_j} = 0$ *or* 1.

Proof : Follows from the proof of Proposition 6.1. □

The basic idea behind our algorithm for optimally merging a pair of machines M^{m_j} and M^{j+1} is as follows. Consider a pair of states s_x^{j+1} and s_i^1 to be merged. Obtain $k_i^{m_j}$ minimum covers of $\mathcal{P}_i^{m_{j+1}}$ by setting σ_{ns} in p_x (the implicant contributed by M^{j+1}) to each of the states in $\mathcal{NS}_i^{m_j}$. Suppose $q_i^{m_{j+1}} - q_i^{m_j} = 0$ for a subset $\mathcal{S}_{i,x}$ of $\mathcal{NS}_i^{m_j}$. This implies that the cost of the slice $\mathcal{P}_i^{m_{j+1}}$ is minimized only if the next state s_y^{j+1} of s_x^{j+1} is merged with one of the states in $\mathcal{S}_{i,x}$. The pair i, x in the subscript of $\mathcal{S}$ denotes that the elements of $\mathcal{S}$ are determined by the implicants to which the state pair belong. If $q_i^{m_{j+1}} - q_i^{m_j} = 1$ for all the $k_i^{m_j}$ covers, then it does not matter how s_y^{j+1} is merged. Thus the merger of a pair of states imposes certain restrictions on the merger of the next states. All pairs of states belonging to the two merging machines are evaluated and the results are recorded. An A^* procedure is then used to determine the optimal mapping for a pair of machines. Once the optimal mapping is found, the present and next state fields in the implicants of M^{j+1} are modified to reflect the mapping and the implicants are added to the STT of M^{m_j} to construct $M^{m_{j+1}}$. The pairwise merge procedure is continued until all machines are merged.

An example of how the merger of the next state influences the cost of a slice in the merged machine is shown in Figs. 6.5(a), (b) and (c). Fig. 6.5(a) shows M^{m_2} which is formed by the merger of the first two machine in *Demo*. Now we want to merge M^3 to this intermediate merged machine. Assume that s_0^3 is merged with s_3^1 of M^{m_2}. Since s_3^1 belongs to slice $\mathcal{P}_3$, we will evaluate the cost of the resulting slice $\mathcal{P}_3$ of M^{m_3} when s_1^3 (the next state of s_0^3) merges with s_0^1 and when it merges with some other state. Fig. 6.5(b) corresponds to the first case when s_1^3 merges with s_0^1. The slice $\mathcal{P}_3$ of M^{m_2} (shown shaded in Fig. 6.5(a)) is now augmented with one additional implicant due to the merger of slice containing the single implicant of M^3. This slice is shown in Fig. 6.5(b). Note that in the additional implicant added state s_0^3 is replaced by s_3^1 and the state s_1^3 is replaced by s_0^1. This slice can be minimized to only one implicant, i.e., $q_3 = 1$. However consider another case where s_1^3 does not merge with s_0^1 but merges with another state s_1^1. The resulting slice is given in Fig. 6.5(c). Multiple valued minimization of this slice results in two implicants as shown. This example therefore illustrates the fact that knowledge of merge information of both the present and the next states of a slice is necessary to determine the cost of merger.

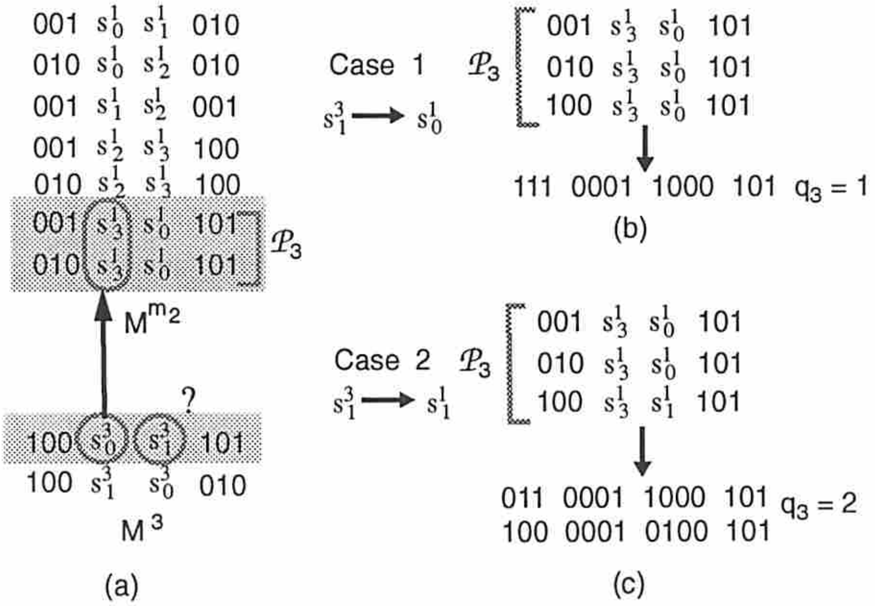


Figure 6.5: Example showing how the merger of the next state influences the implementation cost of the merged machine

From Theorem 6.2 we see that by minimizing the sum of the number of implicants in slices of the merged machine we are guaranteeing an upper bound on the number of implicants in a multiple valued cover of the entire merged machine. This premise is similar to state assignment programs that consider only input constraints [42] and guarantee that the number of implicants in a boolean cover of a state encoded machine is upper bounded by the number of implicants in a multiple-valued cover of the machine. Our results indicate that our cost function does indeed have a very strong correlation to the actual implementation cost. In almost all cases the number of implicants in minimum multiple valued covers of the merged machines equals the sum of the number of implicants in minimum multiple-valued covers of the slices.

In the following section, we will present in detail the A^* algorithm that we use to optimally merge a pair of machines.

6.6 The Merge Procedure

We have developed COMPOSER (COMPact cOntroller SynthesizER), a program that merges a number of test controllers to create the STT of a merged test controller with the minimal number of product terms in its multiple-valued cover. The core of COMPOSER is an A^* algorithm called MERGE that optimally merges a pair of controllers. MERGE is called $n-1$ times by COMPOSER to merge n test controllers.

6.6.1 Preprocessing

Prior to a call to MERGE to merge M^{j+1} and M^{m_j}, a preprocessing phase is executed. In this phase COMPOSER calls Espresso repeatedly to create the sets $\mathcal{S}_{i,x}, i = 0, 1, \ldots, l_1 - 1, x = 0, 1, \ldots, l_{j+1} - 1$, and a matrix A. A has l_1 rows and l_{j+1} columns, where rows represent the states of M^{m_j} and columns represent the states of M^{j+1}. Entry $A_{i,x} = 0$ if $\mathcal{S}_{i,x} \neq \emptyset$, otherwise it is 1. The following pseudocode provides a clearer picture of this preprocessing phase.

Algorithm : Preprocess

```
For merging machine $M^{j+1}$ to $M^{m_j}$
   {  For each slice  $\mathcal{P}_x$  of  $M^{j+1}$
       {  For each slice  $\mathcal{P}_i$  of  $M^{m_j}$
           {  For each distinct next state in  $\mathcal{P}_i$
               {  Merge  $\mathcal{P}_x$  with  $\mathcal{P}_i$
                    to create new slice;
                  Minimize newly created slice;
                  Record in  $\mathcal{S}_{i,x}$  the set of next states
                   of  $\mathcal{P}_i$  that lead to zero cost mappings;
               }
           }
       }
   }
```

6.6.2 Basics of the Merge Procedure

In the following x and $f(x)$ denote states s_x^{j+1} and $f(s_x^{j+1})$, respectively. MERGE finds a mapping f such that $\sum_{x=0}^{l_{j+1}-1}(A_{f(x),x} + b_{f(x),x})$ is minimized, where b is a boolean variable which is 1 if $A_{f(x),x} = 0$ and the next state of s_x^{j+1} is not merged with any state in $\mathcal{S}_{f(x),x}$, otherwise it is 0. Note that a 0 entry in A is "optimistic" because it "hopes" that the next state of s_x^{j+1} will be merged with some state in $\mathcal{S}_{f(x),x}$. The variable b is used to adjust this "optimistic" entry based on actual merger information[2].

MERGE creates a tree of nodes. Each node, N, (except the root) represents the merger of a pair of states from the two machines, M^{m_j} and M^{j+1}. Excluding the root, the tree has l_{j+1} levels corresponding to the number of states in M^{j+1}. The root node is at level 0. A node represents a partial or complete merger. The merger information can be obtained by retracing a path from N to the root. Each node has the following attributes: *s1, s2, g, h, c, chklist.* $s1$ is a state of M^{m_j}, $s2$ is a state of M^{j+1}, g is the cost for the current node N, h is an optimistic estimate of the cost to the goal, and $c = g + h$, is the evaluation function for the node. This evaluation function is used to prune the search space. The subscript from the symbolic state names are actually assigned to *s1, s2.* Thus if s_2^{j+1} merges with s_3^1, then $s2$=2 and $s1$=3 and *chklist*=$\mathcal{S}_{s1,s2}$. A list, *open*, contains the nodes to be processed (expanded). *low_b* is a global variable that keeps track of the cost of the best merger. The notation, N.x refers to the attribute x of node N. Entries in the A matrix are used to compute $N.h$. Suppose N is not a leaf node and represents a partial merger, where states $s_0^{j+1}, \ldots, s_r^{j+1}$ have been assigned to a set of states of M^{m_j}. Then $N.h = \sum_{x=r+1}^{l_{j+1}-1} min(A_{i,x\ \forall i | s_i^1\ is\ an\ unassigned\ state})$. Since A represents the cost of merging two machines without considering next states, and since the computation of $N.h$ uses the least cost "unassigned" entry in each column of A, $N.h \leq N.h^*$, where $N.h^*$ is the actual cost from N to the goal.

[2] This is required in order to guarantee that the A^* algorithm will find the optimum solution.

6.6.3 Example Merger

The operation of MERGE can be best explained with the help of an example. Consider the optimal merger of M^1 and M^2 in *Demo.* Figure 6.6 shows the A matrix and the nodes that are created. Initially *open*=$\{N_{00}\}$, where N_{00} is the root node. This node is expanded. The first child is N_{10}. This node corresponds to a merger of s_0^2 with s_0^1. The various attributes of N_{10} are : $s1$=0, $s2$=0, h=$\sum_{x=1}^{2} min(A_{i,x_{i=1,2,3}})$=$A_{3,1} + A_{2,2}$=1, g=$A_{0,0}$=1, c=2 and *chklist*=$\{\oslash\}$. *chklist* records the fact that the value of $N_{10}.g$ is independent of where the next state s_1^2 merges. Nodes N_{11},N_{12} and N_{13} are then created one after the other. After each node is created, it is inserted into *open*. The insertion is done such that the nodes in *open* are always sorted in non-decreasing order of their evaluation function, c. Thus *open*=$\{N_{12}, N_{11}, N_{13}, N_{10}\}$. A node, N, is pruned if $N.c \geq low\_b$. N_{12} is expanded next to create N_{20}, N_{21} and N_{22}. For N_{22}, $s1$=3, $s2$=1, h=min($A_{i,2_{i=0,1}}$)=0, g=$N_{12}.g + A_{3,1}$=0, c=0 and *chklist*=$\{0\}$. Since $N_{12}.chklist = \{3\}$, and the assignment of states in N_{22} satisfies this constraint, $N_{22}.g$ needs no adjustment. However, consider the computation of $N_{20}.g$. This node represents the merger of s_1^2 with s_0^1. $A_{0,1} = 1$, and the constraint imposed by $N_{12}.chklist$ is not satisfied. Therefore $N_{20}.g = 2$, and since $N_{20}.h = 1$, $N_{20}.c = 3$.

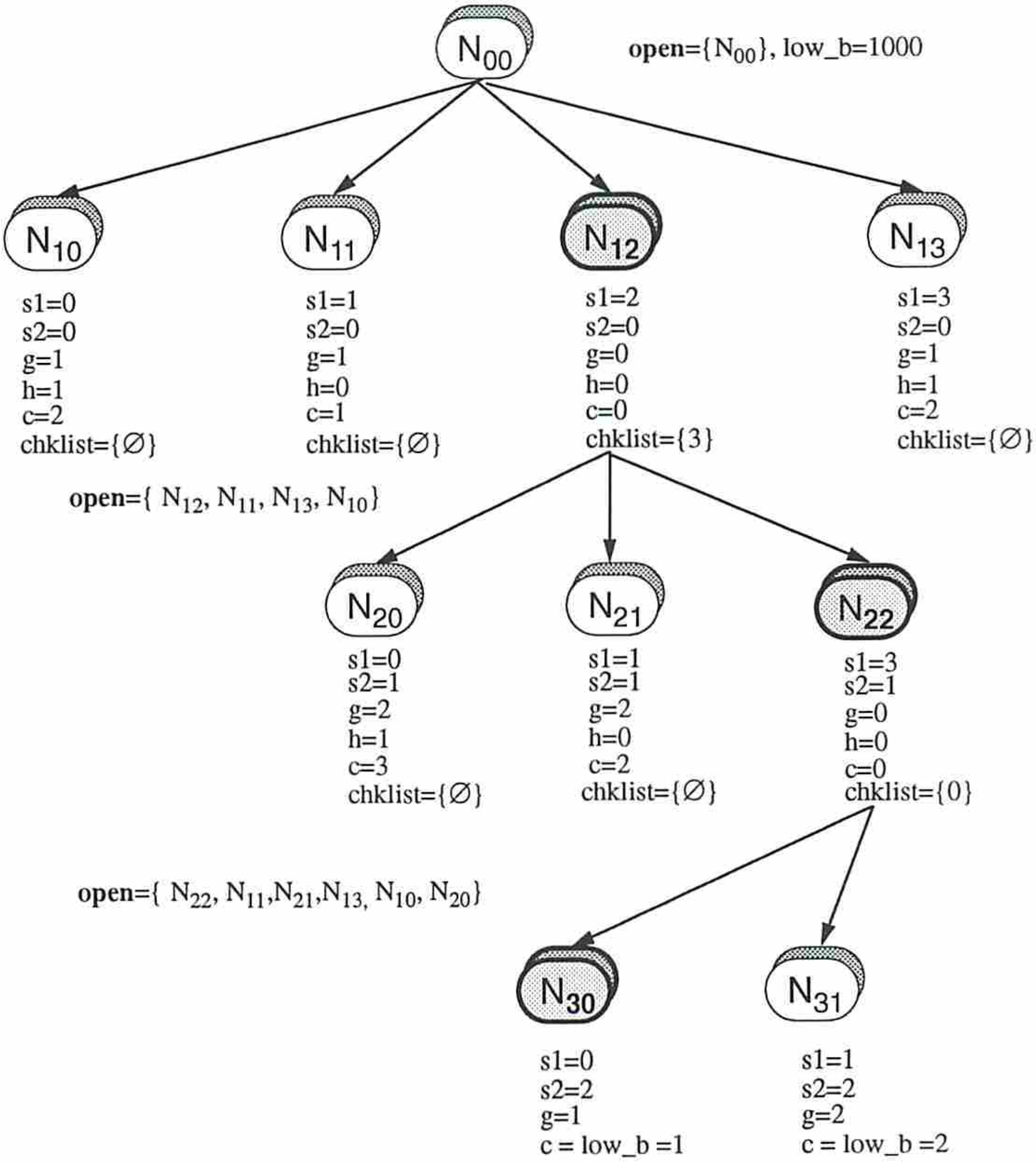
After all the children of N_{12} have been created, *open*=$\{N_{22}$, N_{11}, N_{21}, N_{13}, N_{10}, $N_{20}\}$. The next node expanded is N_{22} and the first child is N_{30} with $s1$=0, $s2$=2, and c=1. This is a leaf node and *low_b*=1. The second child is N_{31} with $s1$=1, $s2$=2, and c=2. c=2 because firstly the chklist entry for N_{22} is not satisfied and $A_{1,2}$=1. Since N_{31}.c > low_b, this node is deleted. Furthermore, we look at the first element in *open*. The first element in *open* is N_{11} and since N_{11}.c ≥ low_b, all nodes in *open* can be deleted and the search stopped. Retracing a path from N_{30} to the root provides all the information about a complete merger and the STT of M^{m_2} is obtained from this information. M^3 can now be merged with M^{m_2} using the same procedure.

6.6.4 Discussion

The worst case time complexity of MERGE is exponential. However, since it is an A^* algorithm, it is guaranteed to return an *optimum* solution whenever a solution exists

	0	1	2
0	1	1	0
1	1	1	1
2	0	1	1
3	1	0	1

(a)



(b)

Figure 6.6: (a) The A matrix; (b) Search tree for merging M^2 to M^1 in *Demo*

and it is consistently better than other equally informed search strategies in that it always explores the least number of nodes [56]. Furthermore, since test controllers are of restricted size in terms of states, the run time of MERGE can be expected to be quite small and this has been observed to be the case for our example circuits. Once all of the controllers have been merged, COMPOSER calls NOVA [43] to encode the inputs and/or perform state assignment.

We now summarize the significance of the various theorems and propositions. The upper bound on the implementation cost of a merged machine derived in Theorem 6.2 is used as our objective function. The results derived in Propositions 6.1 and 6.2 are used to check the validity of the slice minimization technique. The lower bound on the implementation cost of a merged machine derived in Theorem 6.1 can be used in conjunction with the optimistic cost estimator currently employed in MERGE to prune the search space.

6.7 Experimental Results

COMPOSER was run on several examples. Each example consists of a group of test controllers to be merged. The characteristics of these example controllers are presented in Table 6.6. The entries *controllers*, *states* and *outputs* represent the number of controllers being merged, the number of states in each of the machines, and the number of binary valued outputs, respectively. *Ckt1* is a set of controllers for the testable datapath in [57]. *Ckt2* and *Ckt3* are controllers for other testable datapaths. *Small*, *Ex1* and *Ex2* are sets of FSMs where the outputs are arbitrarily assigned values 0, 1 and -. Note that none of the standard benchmark circuits have the special characteristics of our test controllers.

6.7.1 1-hot Coded Primary Inputs and States

Table 6.7 shows characteristics of the merged machines produced by STAMINA and COMPOSER where the primary inputs and states are 1-hot encoded. COMPOSER produces the STT of a merged machine, which is then provided to Espresso to perform both exact and heuristic multiple-valued minimization. Since STAMINA does not accept symbolic inputs, each input symbol was assigned a binary encoding.

Ex	controllers	states	outputs
Small	2	6,6	3
Demo	3	4,3,2	3
Ckt1	3	5,3,2	10
Ckt2	4	6,5,4,3	12
Ex1	5	7,4,4,2,1	3
Ckt3	6	6,5,4,3,2,1	15
Ex2	8	10,8,6,4,3,2,2,1	5

Table 6.6: Characteristics of example circuits

	STAMINA		COMPOSER					
Ex	pe	ph	lb	ub	pe	ph	%ph	cpu
Small	11	11	6	8	7	7	36	2.9
Demo	9	9	6	6	6	6	33	2.3
Ckt1	8	8	7	7	7	7	13	4.3
Ckt2	10	10	9	9	9	9	10	12.4
Ex1	16	16	11	11	11	11	31	9.4
Ckt3	14	14	11	12	12	12	14	23.7
Ex2	37	41	19	22	22	22	46	80.8

Table 6.7: 1-hot coded primary inputs and states

	STAMINA			COMPOSER				
Ex	i,s	p	a	i,s	p	%p	a	%a
Small	1,3	11	17.91	1,3	7	36	12.62	30
Demo	2,3	9	14.63	2,2	6	33	6.49	56
Ckt1	2,3	8	21.16	2,3	7	13	16.80	21
Ckt2	3,3	10	21.77	3,3	9	10	21.39	2
Ex1	4,4	13	33.23	3,3	9	31	22.66	32
Ckt3	3,4	14	32.90	4,3	12	14	28.83	12
Ex2	NOVA timed out			4,4	22	n.a	58.67	n.a

Table 6.8: Optimum state and input encoding using NOVA (satisfying all input constraints)

The FSMs were then concatenated and passed to STAMINA as one large FSM. A state minimal machine was obtained from STAMINA and the binary encodings were replaced by a 1-hot encoding. Each example was run through STAMINA four times, each time with a different *map (-m)* option. All runs produced identical STTs of the state-minimal machine. The STT description was then provided to Espresso to perform both exact and heuristic multiple-valued minimization. In the table, *pe* (*ph*) represents the number of implicants in a minimum (minimal) multiple-valued cover of the machines produced by Espresso. Entry *lb* represents the lower bound of the implementation cost of a merged machine (i.e., $\sum_{i=0}^{l_1-1} k_i$). *ub* represents the upper bound of the implementation cost of a merged machine (i.e., $\sum_{i=0}^{l_1-1} q_i$), and is the cost minimized by COMPOSER. *%ph* is the percentage reduction obtained by COMPOSER in the number of product terms over STAMINA. *cpu* refers to the CPU time in seconds taken by COMPOSER to merge all machines on a Sun SPARCstation 1. From these results we see that COMPOSER produces better results than STAMINA for all examples. Savings in implicants is on average 26% and ranges from 10% (*Ckt2*) to 46% (*Ex2*). Except for *Small* the entries in *ub*, *pe* and *ph* are identical. This indicates that our cost function (based on minimizing slices of a FSM STT) has a strong correlation with the actual implementation cost (based on minimizing the entire machine STT). The merged machine produced by COMPOSER for *Demo* is the same as M_2^m in Table 6.5(a).

6.7.2 Optimum Encoding of Inputs and States

Table 6.8 shows the number of product terms and layout area of merged machines produced by STAMINA and COMPOSER using NOVA to encode the inputs and perform state assignment. The *iexact* algorithm of NOVA was run on the STTs of the merged machines to obtain an encoding of the inputs and the states. This is an exact algorithm that finds an encoding of symbolic variables satisfying all input constraints and minimizing the encoding length. In some cases, the number of bits required for the inputs and states to satisfy all constraints may be greater than the minimum number of bits required to encode all input (state) symbols. The number of implicants in a minimal boolean cover is in some cases less than the number of implicants in a minimal multiple-valued cover because the dominance [45] and disjunctive [47] relationships between bits representing the next states are not taken into account. The encoded and two-level minimized merged machines are mapped into the *msu.genlib* and *msu.genlib_latch* gate libraries using the *map* function in SIS [50]. Layouts were obtained using TimberWolf [58] and YACR [52]. Since TimberWolf uses simulated annealing, three runs were made with each example and the average area (gate area and routing) has been reported. In the table, i (s) represents the number of bits required for optimally encoding the inputs (states). Entries p and a represent the number of implicants in a minimal cover of the encoded FSM and the layout area, respectively. The unit for area is 10^4 μm^2. $\%p$ ($\%a$) denotes the percentage reduction obtained by COMPOSER in product terms (area) as compared to STAMINA.

6.7.3 Inputs and States Encoded Using Minimum Number of Bits

Table 6.9 shows the number of product terms and layout area of the merged machines where both the inputs and states are coded with the minimum number of bits. Thus merged machines produced by STAMINA and COMPOSER for a set of controllers have the same number of input, state and output bits. In this mode of operation, NOVA uses a heuristic procedure to satisfy as many input constraints as possible.

The savings in implicants and area for COMPOSER as compared to STAMINA is on average 33% and 24% , respectively.

6.7.4 General Comments

Note that COMPOSER's performance over STAMINA in terms of product-terms improves for input and state encoded machines as compared to 1-hot coded machines. The layout areas for some of the merged machines produced by COMPOSER and STAMINA also differ substantially. Since most controllers are now implemented as standard cells rather than PLAs, the layout area figures for the merged machines produced by COMPOSER provide a very strong case for our merge procedure. We conjecture that the number and/or the complexity of input constraints to be satisfied for machines produced by COMPOSER is less than that for machines produced by STAMINA.

Fig. 6.7 summarizes the synthesis procedure used to merge the test controllers. In this figure, the input is the STGs of the n test controllers $M^1, M^2, \ldots, M^n$ that are to be merged. The bold arrow depicts the sequence of steps in our synthesis procedure, whereas the lighter arrows represent the synthesis path taken by a traditional approach to this problem. In our synthesis procedure COMPOSER creates the STG of the merged machine. COMPOSER can be thought of performing state minimization. Then standard synthesis tools are used for the rest of the sequential synthesis steps. The double arrows between *composer* and *Espresso* represent the fact that COMPOSER calls *Espresso* repeatedly during the preprocessing steps (for merging a pair of machines) to perform slice minimization.

6.7.5 Sensitivity of the Results to Machine Ordering

In our procedure the machines were ordered in non-increasing order of the number of states and then merged. Experiments were conducted to determine the sensitivity of the results produced by COMPOSER on the ordering of the machines. For each example (except *Ex2*), all machines except the largest were ordered in all possible ways ((n-1)! orders) and COMPOSER was run on each order. For *Ex2* 25 random orderings were used. The results of these experiments are given in Table 6.10. Entries *permutations*, *Cp* and *distribution* refer to the number of orderings, the

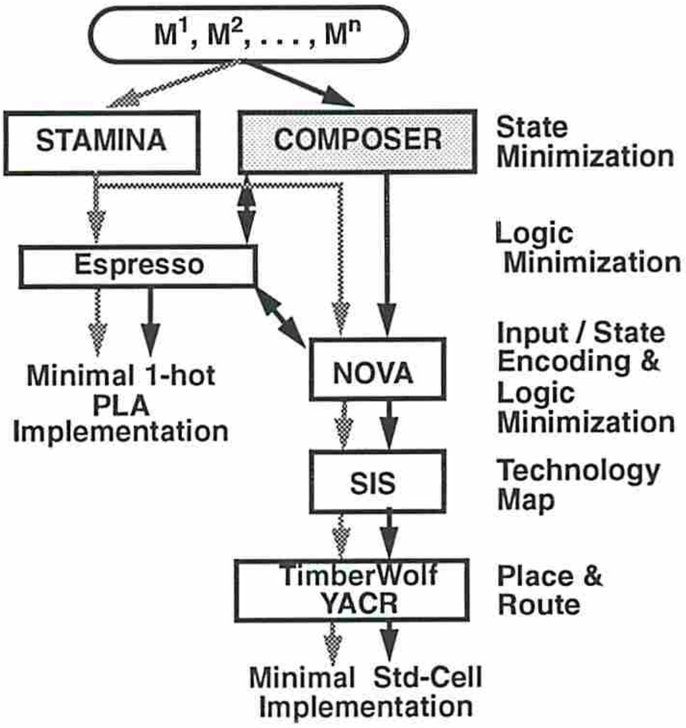


Figure 6.7: The synthesis process

	STAMINA		COMPOSER			
Ex	p	a	p	%p	a	%a
Small	11	17.79	6	45	12.73	28
Demo	9	15.35	6	33	6.94	55
Ckt1	9	23.18	7	22	16.23	30
Ckt2	13	27.33	10	23	25.90	5
Ex1	15	25.69	9	40	23.12	10
Ckt3	16	37.23	13	19	35.37	5
Ex2	47	102.85	23	51	65.78	36

Table 6.9: Minimum bit length input and state encoding using NOVA

		1-hot coded	
Ex	permutations	Cp	distribution
Small	1	7	7
Demo	2	6	6_2
Ckt1	2	7	7_2
Ckt2	6	9	9_6
Ex1	24	11	11_{24}
Ckt3	120	12	12_{120}
*Ex*2*	25	22	22_{11},23_8,24_6

Table 6.10: Effect of machine ordering on merged machine cost

implementation cost of merged machines produced by COMPOSER for our fixed order, and the distribution of the implementation cost of merged machines produced by COMPOSER for different orderings, respectively. The subscripts in *distribution* specify the number of merged machines that result in the associated number of implicants. These experiments were performed for 1-hot (*1-hot coded*) encodings. *Ex2* is starred to indicate that a subset of all possible orderings were explored. From the results we conclude that (1) for 1-hot coded encodings, there is practically no variation in the quality of the results as a function of the order in which the machines are processed, and (2) in the case where there is a variation, sequencing the machines in non-increasing order of the number of states and then processing them leads to the optimal solution.

6.8 Summary

We have presented a technique for merging a set of test controllers. The test controllers are orthogonal FSMs with a single transition out of (into) any state. The controllers are ordered in non-increasing order of the number of states and then merged one at a time to form an intermediate merged machine. Tight upper and lower bounds on the cost of merged machines have been derived. A cost function based on minimizing the multiple-valued covers of slices of the merged machine has been proposed for optimally merging a test controller to an intermediate merged machine. We show that this cost function has a very strong correlation with the implementation cost of a merged machine obtained by minimizing a multiple-valued cover of the entire machine. These ideas have been implemented in C and incorporated in a program called COMPOSER. Experimental results indicate that COMPOSER outperforms an existing state minimizer STAMINA by a wide margin. We observe that the effect of minimizing the number of implicants in a multiple-valued cover of a merged machine (our cost function) percolates through all the subsequent sequential synthesis steps.

merged controller while in this chapter the test and functional controllers are general FSMs whose states can have multiple fanin and/or fanout edges.

We use NOVA [43] (JEDI [59]) to perform state assignment for two-level (multi-level) implementation. We have compared our results with STAMINA, and found that our merging technique produces designs that have about 20% less literals in the factored form and gate area after technology mapping.

7.2 Motivation

7.2.1 Preliminaries

A FSM can be represented by two equivalent structures, (1) a *State Transition Graph (STG)* and, (2) a *State Transition Table (STT)*. We refer interchangeably to rows in the STT as *transitions* or *edges*.

Definition 7.1 *A* **completely specified** *machine has both next state and outputs for all input symbols from every state completely defined.*

Definition 7.2 *An* **incompletely specified** *machine has the next state and/or some outputs for an input symbol from some state not specified.*

M^1 and M^2 are two FSMs where $I^j=\{i_0^j,i_1^j,\ldots,i_{|I^j|-1}^j\}$, $S^j=\{s_0^j,s_1^j,\ldots,s_{|S^j|-1}^j\}$ and $O^j=\{o_0^j,o_1^j,\ldots,o_{|O^j|-1}^j\}$ represent the set of inputs, states and outputs of machine M^j, j=1,2. $\mid S^1 \mid = p$ and $\mid S^2 \mid = q$, where $p \geq q$. I^1 and I^2 are disjoint since M^1 and M^2 correspond to a pair of test and functional controllers. If the number of states in the functional controller is larger than or equal to the number of states in the test controller, then M^1 is the functional controller and M^2 is the test controller, else M^1 and M^2 are the test and functional controllers, respectively.

7.2.2 The Merging Process

In general, the merger of two machines with p and q states produces as many as p_*q states prior to state minimization. However, the FSMs we target for merger are temporally disjoint and also input disjoint. State minimization becomes trivial

0	s_0^1	s_0^1	00	1	s_0^2	s_0^2	11
1	s_0^1	s_1^1	00	0	s_0^2	s_1^2	11
-	s_1^1	s_2^1	1-	-	s_1^2	s_2^2	--
-	s_2^1	s_3^1	--	-	s_2^2	s_0^2	00
1	s_3^1	s_3^1	00				
0	s_3^1	s_0^1	00				
	M^1				M^2		

Table 7.1: *Mrgex*

for this case, since any state of one machine can be merged with any state of the other machine using the mode control signal T to condition any outputs that are incompatible. The number of states in the merged machine is thus equal to the number of states in M^1.

The merger of two orthogonal machines is the process of determining an injective mapping $\mathcal{F}$ that maps (merges) the states of M^2 to the states of M^1. Any mapping will produce a state minimal machine. However, the minimality of next state and present state logic is not guaranteed. Consider the example shown in Table 7.1 which depicts the STTs of a pair of machines M^1 and M^2. This pair of machines will be referred to as *Mrgex* throughout this chapter. The inputs to M^1 and M^2 are i_0^1 and i_0^2, respectively. Each machine has two outputs that drive two datapath control lines c_0 and c_1 through a multiplexer.

Consider the three mappings, $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$, where $\mathcal{F}_1$ maps states s_0^2, s_1^2, s_2^2 to states s_3^1, s_1^1, s_2^1; $\mathcal{F}_2$ maps s_0^2, s_1^2, s_2^2 to s_1^1, s_2^1, s_3^1; and $\mathcal{F}_3$ maps s_0^2, s_1^2, s_2^2 to s_3^1, s_2^1, s_1^1. Table 7.2 shows the STTs of the three merged machines M_1^m, M_2^m and M_3^m obtained by $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$, respectively. The inputs of the merged machines are i_0^1, T, i_0^2 and the outputs are c_0, c_1. In general, a merged machine has l outputs, where $l = max(\mid O^j \mid)$, j=1,2. The output cubes of the machine with the smaller number of outputs are padded with don't cares. Figs. 7.2(a) through (e) show the STGs of M^1, M^2 and the merged machines. Fig. 7.2(f) shows state mappings used for different merged machines.

Suppose M^1 and M^2 are implemented separately and the output lines are multiplexed. A gate level implementation of M^1, M^2 and the multiplexer has 28, 27 and 8 units of area, respectively, leading to a total of 63. In comparison, a gate level

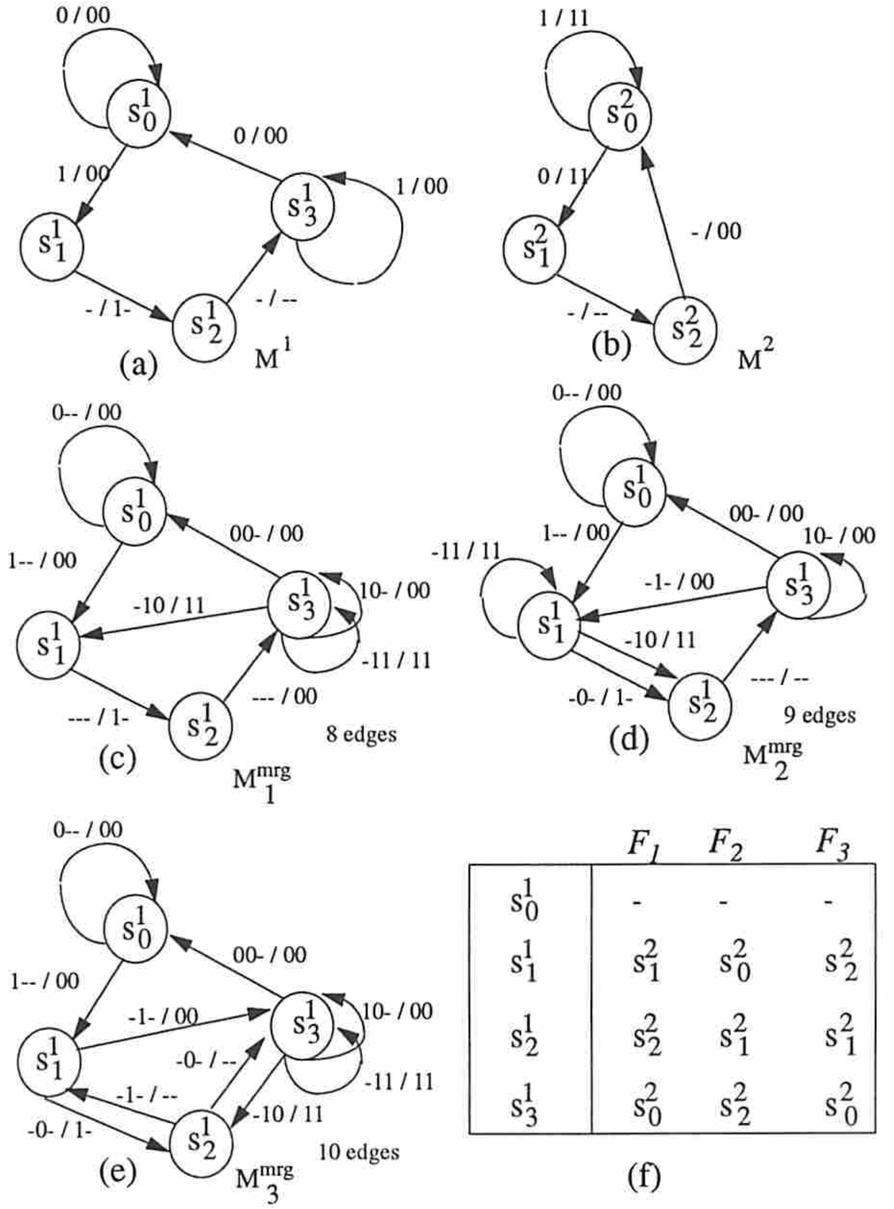
0-- s_0^1 s_0^1 00	0-- s_0^1 s_0^1 00	0-- s_0^1 s_0^1 00
1-- s_0^1 s_1^1 00	1-- s_0^1 s_1^1 00	1-- s_0^1 s_1^1 00
--- s_1^1 s_2^1 1-	-0- s_1^1 s_2^1 1-	-0- s_1^1 s_2^1 1-
--- s_2^1 s_3^1 00	-11 s_1^1 s_1^1 11	-1- s_1^1 s_3^1 00
10- s_3^1 s_3^1 00	-10 s_1^1 s_2^1 11	-0- s_2^1 s_3^1 --
00- s_3^1 s_0^1 00	--- s_2^1 s_3^1 --	-1- s_2^1 s_1^1 --
-11 s_3^1 s_3^1 11	10- s_3^1 s_3^1 00	10- s_3^1 s_3^1 00
-10 s_3^1 s_1^1 11	00- s_3^1 s_0^1 00	00- s_3^1 s_0^1 00
	-1- s_3^1 s_1^1 00	-11 s_3^1 s_3^1 11
		-10 s_3^1 s_2^1 11
M_1^m	M_2^m	M_3^m

Table 7.2: Merged machines for different mappings

implementation of M_2^m has 40 units of area. Merging the machines thus reduces the gate area by 37%.

Two-level implementations of M_1^m, M_2^m and M_3^m, obtained by running NOVA, have 6, 6 and 8 product terms respectively. A multi-level implementation using JEDI to assign states and SIS to perform logic minimization yields 17, 14 and 22 factored form literals for the three merged machines. There are 24 different mappings which merge the three states of M^2 to the four states of M^1, each mapping corresponds to a different merged machine. Two-level and multi-level synthesis of all 24 merged machines were performed. M_3^m with 22 literals is the worst and M_4^m with 13 literals is the best among all merged machines. M_4^m (mapping $\mathcal{F}_4$) corresponds to the merger of s_0^2, s_1^2, s_2^2 to s_1^1, s_2^1, s_0^1. The number of product terms for two-level implementations varies between 6 and 8 for different mergers. Fig. 7.3 graphically represents the costs associated with the different mappings. Mapping $\mathcal{F}_5$ corresponds to the merged machine produced by STAMINA (discussed in the following paragraph).

A merged machine can also be obtained by using a state minimization tool such as STAMINA. The STTs of M^1 and M^2 are concatenated together as shown in Table 7.3 and provided as input to STAMINA. Note that the second input, T, is used to distinguish between the two machines. This description corresponds to an incompletely specified machine with 7 states. STAMINA produces a state minimal machine with 4 states. A multi-level implementation of this machine has 20 literals.



	F_1	F_2	F_3
s_0^1	-	-	-
s_1^1	s_1^2	s_0^2	s_2^2
s_2^1	s_2^2	s_1^2	s_1^2
s_3^1	s_0^2	s_2^2	s_0^2

Figure 7.2: STGs of M^1, M^2 and some merged machines

00-	s_0^1	s_0^1	00
10-	s_0^1	s_1^1	00
-0-	s_1^1	s_2^1	1-
-0-	s_2^1	s_3^1	--
10-	s_3^1	s_3^1	00
00-	s_3^1	s_0^1	00
-11	s_0^2	s_0^2	11
-10	s_0^2	s_1^2	11
-1-	s_1^2	s_2^2	--
-1-	s_2^2	s_0^2	00

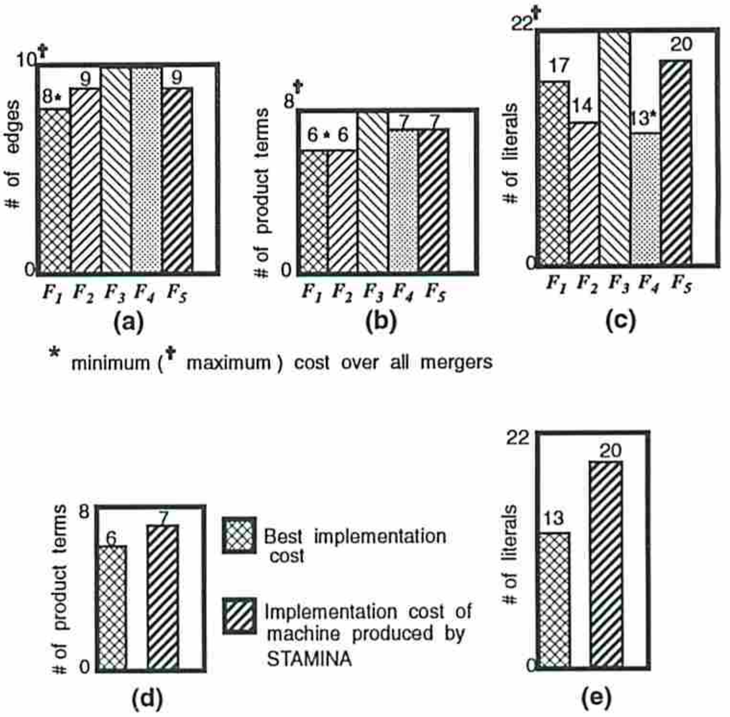
Table 7.3: Concatenated STTs of M^1 and M^2

Analyzing the merged machine produced by STAMINA we see that the states s_0^2, s_1^2, s_2^2 are simply merged with s_0^1, s_1^1, s_2^1, i.e., STAMINA picks the first possible merger.

We have seen that different mappings influence the implementation cost of the merged machine. For *Mrgex*, the worst merged machine has 70% more literals or 33% more product terms than the best merged machine. It has also been shown that an existing popular tool is inadequate for the merger problem. The number of possible merged machines is $\prod_{i=0}^{q-1}(p-i)$, which is $\mathbf{O}$(p!). It is impractical to exhaustively generate all merged machines, synthesize them and then pick the best merger. In this chapter we present efficient techniques to find merger(s) which lead to machines with minimal or near minimal implementation cost.

7.3 The Objective Functions

The optimization of the combinational component of the FSM depends heavily on the state encoding. The cost function that estimates the optimality of an encoding depends on the target implementation: two-level or multi-level. Two-level implementations minimize the number of product terms or the area of a PLA. Multi-level implementations minimize the number of literals of a factored form representation of the logic. Good state assignment tools for two-level implementations exist. This is not the case for multi-level implementation, since it is considerably more difficult to quantify the number of literals before actually performing logic minimization.



* minimum († maximum) cost over all mergers

Figure 7.3: Statistics for various mappings : (a) number of edges; (b) number of product terms in two-level implementation; (c) number of literals in multi-level implementation; Comparison with STAMINA : (d) product terms; (e) literals

Most multi-level state assignment tools, such as MUSTANG [60] and JEDI, estimate the *ease of sharing* literals for different assignments. The assumption is that there is a correlation between the degree of literal sharing and the number of literals after state assignment and logic minimization.

In contrast to the state assignment problem, where the state transition table does not change, in our problem the state transition table itself changes when states of one machine are merged to different states of another machine. Thus our problem of predicting which merger will produce a machine with the least cost is as difficult as the optimal state assignment problem which is NP-hard. To choose between different merged machines, we propose the following cost functions:

1. Minimization of the number of edges (in the STG) of M^m (*edge-minimal*),

2. Minimization of the number of factored form literals in a multi-level implementation of the merged machine by considering pairs of states (state-pairs) from M^1 and M^2 (*literal-minimal*).

The *edge-minimal* criterion is used for two-level implementation and the *literal-minimal* criterion is used for multi-level logic implementation.

7.4 Edge Minimization

Merging M^2 to M^1 adds a certain number of edges to M^1. An edge minimal merger thus adds a minimal number of edges to M^1.

7.4.1 Mechanism for Merging Edges

Let $e_{s^1_x}$ and $e_{s^2_y}$ represent the number of fanout edges from states s^1_x and s^2_y, respectively.

Definition 7.3 *A* **single fanout edge** *refers to a fanout edge of a state* s *such that* $e_s = 1$.

Definition 7.4 *A* **multiple fanout edge** *refers to a fanout edge of a state* s *such that* $e_s > 1$.

Suppose M^2 is merged with M^1 under a mapping $\mathcal{F}$, where $\mathcal{F}(s^2_y) = s^1_x$ or $s^2_{\mathcal{F}(y)} = s^1_x$ denotes that state s^2_y is merged with state s^1_x. We assume that each of the machines M^1 and M^2 have next states specified for all input symbols. Thus a single edge such as "1 s^1_0 s^1_1 -" out of state s^1_0 is not allowed. Such a single edge means that the transition out of s^1_0 for input value 0 has been left unspecified. We also assume that all transitions out a state are represented by a minimal number of fanout edges. Thus two edges such as "1 s^1_0 s^1_1 1" and "0 s^1_0 s^1_1 -" are represented as "- s^1_0 s^1_1 1". The following theorem states conditions under which fanout edges of a state in M^2 are either added to M^1 or merged with existing edges.

Theorem 7.1 *Due to the mapping $\mathcal{F}(s^2_y) = s^1_x$ no edges are added to M^1 if (1) $e_{s^2_y} = 1$ and $e_{s^1_x} = 1$, (2) the edges are output compatible, and (3) the next state of s^2_y is also mapped to the next state of s^1_x, otherwise $e_{s^2_y}$ edges are added.*

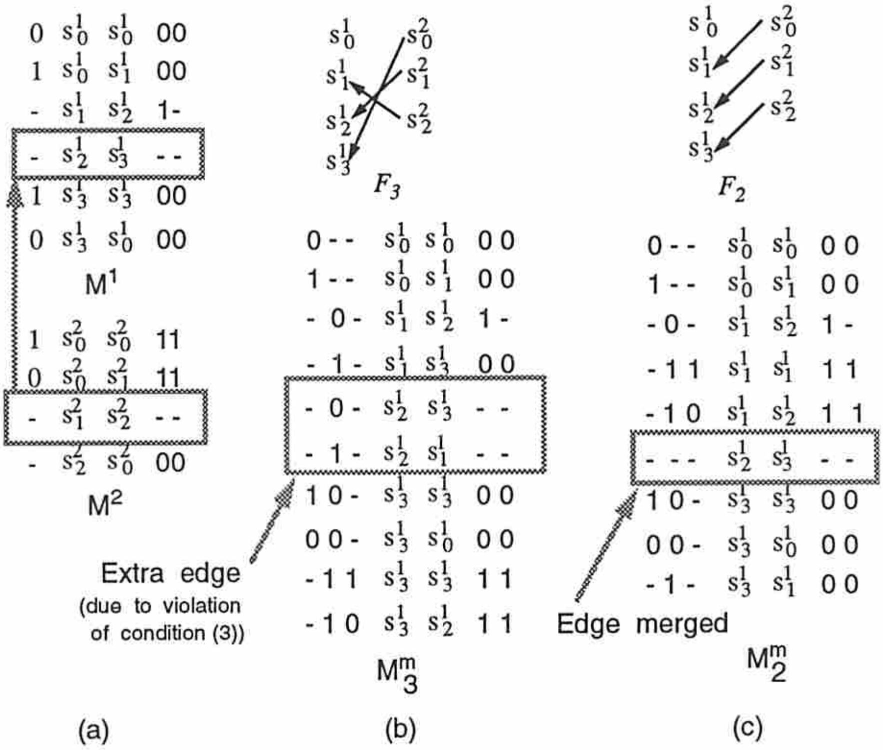
Proof : $e_{s^2_y} > 1$ implies that a non empty set of variables in I_2 are 1's or 0's in the input cubes of all the fanout edges of s^2_y. However, in the merged machine, all variables in I_2 are don't cares in the input cubes of the edges corresponding to M^1. Similarly, all variables in I_1 are don't cares in the input cubes of edges corresponding to M^2. Therefore $e_{s^2_y}$ edges have to be added. If $e_{s^2_y} = 1$ and $e_{s^1_x} = 1$, then all variables in the input cubes of the fanout edges of s^2_y and s^1_x are don't cares. However, if the outputs asserted by these states are incompatible, then an additional edge will have to be added to distinguish between the outputs asserted by the two machines. If the outputs are compatible and if the next state of s^2_y is also mapped to the next state of s^1_x, then an additional edge is not needed. The original edge in M^1 is sufficient for both the machines. The outputs will have to be suitably modified, i.e., don't cares have to be changed to 1's or 0's as needed. □

Fig. 7.4 illustrates Theorem 7.1 by showing how an edge of M^2 is either added to M^1 or merged with an existing edge of M^1. Consider the case shown in Fig. 7.4(b) (M^m_3) where s^2_1 merges with s^1_2, $e_{s^2_1} = 1$ and $e_{s^1_2} = 1$, the outputs are compatible, but the next state of s^2_1 is not merged with the next state of s^1_2. Therefore, the edge "- s^2_1 s^2_2 --" cannot be merged with "- s^1_2 s^1_3 --" and is added as an extra edge. However, in Fig. 7.4(c) (M^m_2), since s^2_2 merges with s^1_3, the edge "- s^2_1 s^2_2 --" can be merged with "- s^1_2 s^1_3 --" and the composite edge is "--- s^1_2 s^1_3 --".

7.4.2 Analysis of Edge Minimization

Our ultimate goal is to obtain a merged machine that has minimal implementation cost in terms of number of product terms. In Section 6.3 of Chapter 6 we have presented a detailed description of general FSM synthesis techniques. In this section we will therefore summarize some of the relevant concepts.

A symbolic cover is a code independent (symbolic) representation of the combinational component of the FSM and consists of a set of primitive elements called *symbolic implicants*. For example, consider the STT of M^1 in Table 7.1. The row "0 s^1_0 s^1_0 00" in the STT is a symbolic implicant. There are 6 rows in the STT, each row is a symbolic implicant and these implicants taken together form a symbolic cover of M^1. Usually the symbols of a variable are assigned different values and the resulting symbolic cover is referred to as a *multiple-valued* cover. Symbolic minimization is usually performed by simultaneously minimizing the multiple-valued input functions related to each next state and each of the binary outputs. This is called *output*



F_3
F_2
s_0^1 s_1^1 s_2^1 s_3^1
s_0^2 s_1^2 s_2^2
Extra edge
(due to violation of condition (3))
Edge merged
(a)
(b)
(c)

0 s_0^1 s_0^1 00
1 s_0^1 s_1^1 00
- s_1^1 s_2^1 1-
- s_2^1 s_3^1 --
1 s_3^1 s_3^1 00
0 s_3^1 s_0^1 00

M^1

1 s_0^2 s_0^2 11
0 s_0^2 s_1^2 11
- s_1^2 s_2^2 --
- s_2^2 s_0^2 00

M^2

0-- s_0^1 s_0^1 00
1-- s_0^1 s_1^1 00
-0- s_1^1 s_2^1 1-
-1- s_1^1 s_3^1 00
-0- s_2^1 s_3^1 --
-1- s_2^1 s_1^1 --
10- s_3^1 s_3^1 00
00- s_3^1 s_0^1 00
-11 s_3^1 s_3^1 11
-10 s_3^1 s_2^1 11

M_3^m

0-- s_0^1 s_0^1 00
1-- s_0^1 s_1^1 00
-0- s_1^1 s_2^1 1-
-11 s_1^1 s_1^1 11
-10 s_1^1 s_2^1 11
--- s_2^1 s_3^1 --
10- s_3^1 s_3^1 00
00- s_3^1 s_0^1 00
-1- s_3^1 s_1^1 00

M_2^m

Figure 7.4: Illustration of theorem

disjoint multiple valued minimization because each of the next states is considered to be a different component of the original symbolic function. We refer to output disjoint minimization simply as *multiple valued (MV)* minimization.

Suppose the states, s_0^1, s_1^1, s_2^1 and s_3^1 of M^1 in *Mrgex* are assigned codes 1000, 0100, 0010 and 0001, respectively. States s_0^2, s_1^2 and s_2^2 of M^2 are assigned codes 100, 010 and 001, respectively. The minimum MV covers of M^1 and M^2 are shown in Table 7.4. The implicant "0 1001 1000 00" implies that when the input is 0, states s_0^1 and s_3^1 both assert outputs 00 and go to the same next state s_0^1. The pair of states s_0^1 and s_3^1 is an input constraint. The implicants "– 0010 0001 ––" and "1 0001 0001 00" cannot be combined together into "1 0011 0001 00" because this does not cover "0 0010 0001 ––". They cannot also be combined in any other manner. The implicant "0 1001 1000 00" can also be represented as "0 (s_0^1, s_3^1) s_0^1 00".

In order to analyze edge minimization we introduce the concept of *symbolic prime implicants (SPIs)*.

0 1001 1000 00	1 100 100 11
1 1000 0100 00	0 100 010 11
- 0100 0010 1-	- 010 001 - -
- 0010 0001 - -	- 001 100 00
1 0001 0001 00	
(a) MV cover for M^1	(b) MV cover for M^2

Table 7.4: Minimum MV covers for M^1 and M^2

Definition 7.5 *A SPI represents a maximal group of fanout edges (of a set of states) that go the same next state and assert compatible binary valued outputs under some inputs.*

Thus "0 (s_0^1, s_3^1) s_0^1 00" is a SPI and it covers the implicants (edges) "0 s_0^1 s_0^1 00" and "0 s_3^1 s_0^1 00". The edge "1 s_0^1 s_1^1 00" can only be covered by itself and is a SPI. Note that in the definition of an SPI we assume that the binary valued outputs of a symbolic implicant are processed as a group and not as individual outputs as is done by some MV minimizers [49]. An exact solution to the minimum MV cover problem where the binary valued outputs of each symbolic implicant are processed as a group therefore requires the identification of all SPIs of a FSM, and then the selection of a minimum subset of SPIs that cover the FSM. For M^1, the set of all 5 SPIs are needed for the minimum cover, shown in Table 7.4(a).

Lemma 7.1 *Let $\mathcal{SPI}_{all}$ represent the set of all SPIs of a machine. $\mathcal{SPI}_{all}$ can be partitioned into two sets $\mathcal{SPI}_a$ and $\mathcal{SPI}_b$ such that $\mathcal{SPI}_a$ ($\mathcal{SPI}_b$) covers all and only all the single fanout edges (multiple fanout edges) of the machine.*

Proof : All single fanout edges have don't cares in all bit positions in their input cubes. Multiple fanout edges however have at least one bit in their input cubes that is not a don't care. Therefore there cannot exist an SPI that covers a single fanout edge and also a multiple fanout edge. All the SPIs obtained by considering only the single fanout edges constitute $\mathcal{SPI}_a$. Similarly all SPIs obtained by considering only the multiple fanout state edges constitute $\mathcal{SPI}_b$. Since there is no edge that is covered by a member of both $\mathcal{SPI}_a$ and $\mathcal{SPI}_b$, $\mathcal{SPI}_a \cap \mathcal{SPI}_b = \oslash$ and $\mathcal{SPI}_{all} = \mathcal{SPI}_a \cup \mathcal{SPI}_b$. □

For example, for M^1, $\mathcal{SPI}_a$={- s_1^1 s_2^1 1-, - s_2^1 s_3^1 --} and $\mathcal{SPI}_b$={0 (s_0^1, s_3^1) s_0^1 00, 1 s_0^1 s_1^1 00, 1 s_3^1 s_3^1 00}.

Let C_a^i and C_b^i denote the cardinalities of the minimum covers of single and multiple fanout edges of M^i, obtained by using members of sets $\mathcal{SPI}_a^i$ and $\mathcal{SPI}_b^i$, respectively, for i=1,2. Let C^m be the cardinality of the minimum cover of a merged machine M^m, E_a^2 the number of single fanout edges of machine M^2, and E^1 the total number of edges in M^1. We present bounds on C^m in terms of the cardinalities of the minimum covers of the single and multiple fanout edges of the two machines.

Lemma 7.2 $C^m \geq max(C_a^1, C_a^2) + C_b^1 + C_b^2$.

Proof : A merged machine is obtained by adding edges of M^2 to M^1. In the merged machine the input cubes of all multiple fanout edges of (contributed by) M^2 are qualified by a 1 (if M^2 is the test controller) or by a 0 (if M^2 is the functional controller) in bit position i, corresponding to the input T. The input cubes of all edges of M^1 are qualified either by a 0 (1) or a don't care in the corresponding bit position i. The set of SPIs of M^m that cover the multiple fanout edges of M^2 thus cannot cover any edge of M^1. The cardinality of the minimum cover of this set of SPIs is C_b^2.

Suppose state s_y^2 merges with a state s_x^1 with $e_{s_x^1} > 1$, then the input cubes of all fanout edges of s_x^1 are changed in the ith bit position, corresponding to T, from a don't care to a 0 (1). Suppose a SPI P, covered one fanout edge each from states s_x^1 and s_z^1. After state merger, P can no longer cover these edges, unless another state from M^2 merges with s_z^1, in which case the fanout edge of s_z^1 under consideration will also have a don't care in bit position i changed to a 0. Therefore C_b^1 is a lower bound on the number of SPIs needed to cover the multiple fanout edges of M^1.

Consider the case $C_a^1 > C_a^2$. In the best case all single fanout edges of M^2 are merged with the single fanout edges of M^1. Therefore C_a^1 is a lower bound on the number of SPIs needed to cover the single fanout edges of both the merging machines. The cardinality of the minimum cover of the merged machine is thus lower bounded by $C_a^1 + C_b^1 + C_b^2$ and this bound is achievable. The case $C_a^2 \geq C_a^1$, can be proved in a similar manner. □

Among all merged machines, M_1^m in Table 7.2 has the least number of implicants (C^m=8) in its minimum MV cover. For M_1^m, max(C_a^1, C_a^2)=2, C_b^1=3 and C_b^2=2.

Lemma 7.3 $C^m \leq E^1 + C_b^2 + E_a^2$.

Proof : All the multiple fanout edges of M^2 can be covered by C_b^2 SPIs. The merging may occur in a manner such that no SPI can cover two edges of M^1, thus requiring E^1 SPIs to cover all edges of M^1. E_a^2 is a loose upper bound on the SPIs needed to cover the single fanout edges of M^2. □

M_3^m is one of the machines which has the largest number of implicants (C^m=10) in its minimum MV cover. For M_3^m, E^1=6, C_b^2=2 and E_a^2=2.

We will show that under certain conditions there is a correlation between minimizing edges in a merged machine and the size of a minimal MV cover where each of the next states are considered to be a different component of the original symbolic function and the binary valued outputs of each implicant are processed together. We will refer to such a cover as *strictly output disjoint.*

Lemma 7.4 *If all fanout edges of M^1 and all single fanout edges of M^2 are SPIs (when merging M^2 into M^1), then a merged machine has a strictly output disjoint MV cover of minimum cardinality* if *and* only if *it has a minimum number of edges.*

Proof : In any M^m, E^1 SPIs are needed to cover the edges corresponding to M^1 and C_b^2 SPIs are needed to cover the multiple fanout edges corresponding to M^2. Edge minimization results in merging a maximum number, k, of single fanout edges of M^2 to the single fanout edges of M^1. Therefore, a minimum number, $E_a^2 - k$, of SPIs are needed to cover the unmerged single fanout edges of M^2. Thus edge minimization leads to a minimum MV cover. The necessity follows from the fact that a minimum MV cover implies that a minimum number, $C^m - E^1 - C_b^2$, of SPIs need to cover the unmerged single fanout edges of M^2, where C^m is the cardinality of a minimum MV cover of M^m. This implies that a maximum number of single fanout edges on M^2 have been merged with single fanout edges on M^1 which in turn implies that edge minimization has been done. □

From Lemma 7.3 we see that edge minimization attempts to reduce the upper bound on C^m and this cost function is most effective when conditions given in Lemma 7.4 are met. The size of a cover obtained by strictly output disjoint MV minimization is an upper bound on the size of a binary cover of the FSM that is obtained by replacing the present and next states by a binary encoding (which satisfies all input of face constraints) and running a multiple-output binary valued minimizer such as Espresso.

Consider M^1 and M^2 of Table 7.1. All single fanout edges of M^2 are SPIs, whereas some edges of M^1 are not SPIs. However, M_1^m is the only merged machine with the least number of edges (8 edges) and the minimum MV cover (8 implicants) and one of the five machines with the least number of product terms (6 terms). This example shows that edge minimization is an efficient cost function even when the conditions in Lemma 7.4 are not met.

7.4.3 Computing the Number of Added Edges for State-pairs

Let A be a matrix with p rows and q columns, where rows represent states of M^1 and columns represent states of M^2. An entry $A_{x,y}$ denotes the number of edges added to M^1 when s_y^2 is merged with s_x^1:

- $A_{x,y}$=$e_{s_y^2}$ if ($e_{s_x^1} = 1$ and $e_{s_y^2} > 1$) or ($e_{s_x^1} > 1$ and $e_{s_y^2} \geq 1$),
- $A_{x,y}$= 1 if $e_{s_x^1} = e_{s_y^2} = 1$ and the edges are *output incompatible*,
- $A_{x,y}$= 0 if $e_{s_x^1} = e_{s_y^2} = 1$ and the edges are *output compatible.*

Note that in the above formulations, the cost of merging s_y^2 with s_x^1 ignores how the other states are merged. This leads to no problems for the first two cases. But setting $A_{x,y} = 0$ for the third case is an "optimistic" assumption. Let $\mathcal{C}_{edges}$ represent the cost in terms of additional edges for a mapping $\mathcal{F}$. Then

$$\mathcal{C}_{edges} = \sum_{y=0}^{q-1}(A_{\mathcal{F}(y),y} + b_{\mathcal{F}(y),y}) \qquad (7.1)$$

where b is a boolean variable which is 1 if $A_{\mathcal{F}(y),y}$=0 and next states of s_y^2 and $s_{\mathcal{F}(y)}^1$ are not merged, otherwise it is 0. This variable is used to adjust the "optimistic" entry. Thus edge minimization corresponds to determining $\mathcal{F}$ such that C_{edges} is minimal.

The A matrix for *Mrgex* is given in Table 7.5. Only the mapping corresponding to $\mathcal{F}_1$ (entries shown in bold face in A) minimizes the number of added edges ($\mathcal{C}_{edges}$=2).

	0	1	2
0	2	1	1
1	2	**0**	1
2	2	0	**0**
3	**2**	1	1

Table 7.5: The A matrix corresponding to *Mrgex* for minimizing edges

7.5 Literal Minimization

A merger that produces a minimal two-level implementation may not lead to a minimal multi-level implementation. For example, in Section 7.2.2 it was shown that $\mathcal{F}_1$ leads to a minimal two-level implementation while $\mathcal{F}_4$ leads to a minimal multi-level implementation. The outputs and next state of some state s^1_x can be expressed in terms of the inputs and s^1_x itself. s^1_x is considered as a symbolic literal. For example, consider edge "- s^1_1 s^1_2 1". We express next state s^1_2 and output c_0 in terms of the present state s^1_1 as follows: $s^1_2 = s^1_1$ and $c_0 = s^1_1$. Merging a state s^2_y with s^1_x may add fanout edges and/or alter the input and output cubes of the original fanout edges of s^1_x. This may lead to an increase in the literal count for implementing the fanout edges of s^1_x after the merger.

Suppose "- s^1_1 s^1_2 0" and "- s^2_1 s^2_2 1" are two fanout edges in M^1 and M^2, and s^2_1, s^2_2 are merged with s^1_1, s^1_2. The fanout edges of s^1_1 after merger are "-0- s^1_1 s^1_2 0" and "-1- s^1_1 s^1_2 1". The next state s^1_2 and output c_0 are expressed in terms of s^1_1 and T as follows: s^1_2=s^1_1 and $c_0 = Ts^1_1$. Thus three literals are required. However, the original fanout edge of s^1_1 only required one literal, i.e., $s^1_2 = s^1_1$. Merger of different states of M^2 to s^1_1 may add different number of literals.

In general, we can extract common cubes and express the outputs and next states in factored form expressions and determine the number of literals added for mergers of various pairs of states from M^1 and M^2. For state-pairs with single fanout edges, sometimes the merger information of the next states is also needed. We *estimate* the number of literals added when merging various state-pairs based upon properties such as the number of fanout edges, the output cubes of the edges, and their next states. A mapping is then found that minimizes the sum of the added literals. In this chapter, we focus on Moore type machines, where the output is only a function of the present states, not the inputs. This is not a shortcoming since any Mealy machine can be converted to an equivalent Moore machine.

7.5.1 Estimation of the Number of Added Literals for State-pairs

A is a matrix with p rows and q columns. The entry $A_{x,y}$ represents the estimated number of literals added when s_y^2 is merged with s_x^1. Let $c_{x_i}^1$ and $c_{y_i}^2$ represent the ith output bit of states s_x^1 and s_y^2 respectively. Recall that a merged machine has l output bits, where $l = max(\mid O^j \mid)$, j=1,2, and the output cubes of the machine with the smaller number of outputs are padded with don't cares. m_{xy} is an integer variable that depends on the outputs of the merged states as follows:

$$m_{xy} = \lfloor \sum_{i=0}^{l-1} C(c_{x_i}^1, c_{y_i}^2) \rfloor \tag{7.2}$$

where

- $C(c_{x_i}^1, c_{y_i}^2) = 2$ if $c_{x_i}^1 = 0$ and $c_{y_i}^2 = 1$,
- $C(c_{x_i}^1, c_{y_i}^2) = 1$ if $c_{x_i}^1 = 1$ and $c_{y_i}^2 = 0$,
- $C(c_{x_i}^1, c_{y_i}^2) = 0.5$ if $c_{x_i}^1 = -$ and $c_{y_i}^2 = 1$,
- $C(c_{x_i}^1, c_{y_i}^2) = 0$ for all other combinations.

For example, we have seen that when the state s_1^2 with fanout edge "- s_1^2 s_2^2 1" merges with the state s_1^1 with fanout edge "- s_1^1 s_2^1 0", two additional literals are needed. However, if the edges were "- s_1^2 s_2^2 0" and "- s_1^1 s_2^1 1", then only one additional literal would be needed.

Consider the merger of "- s_1^2 s_2^2 1" with "- s_1^1 s_2^1 -", one additional literal is needed. But there is a possibility that the don't care in the output of s_1^1 will be changed to a 1 during multi-level synthesis. Therefore we penalize this case less than when $c_{x_i}^1 = 1$ and $c_{y_i}^2 = 0$. Now consider the case where s_2^2 is not merged with s_2^1, but with some other state s_x^1. Then the state s_1^1 has the following fanout edges after merger: "-0-

	0	1	2
0	4	3	3
1	**0**	0	1
2	1	**0**	0
3	4	3	**3**

Table 7.6: The A matrix corresponding to *Mrgex* for minimizing literals

s_1^1 s_2^1 1", "-1- s_1^1 s_x^1 0". Define a cube $k = s_1^1\overline{T}$. Thus s_2^1=k, $c_0 = k$ and $s_x^1 = s_1^1 T$. Six literals are need as opposed to two in the unmerged case.

Therefore, in some cases, the next state merger also influences the literal count for a pair of states. A constant, w, represents the influence of the next states in the matrix A and the cost function.

The entries in A are as follows:

- $A_{x,y} = m_{xy}$ if $e_{s_x^1} \geq 1$ and $e_{s_y^2} > 1$,
- $A_{x,y} = m_{xy} + w$ if $e_{s_x^1} > 1$ and $e_{s_y^2} = 1$,
- $A_{x,y} = m_{xy}$ if $e_{s_x^1} = 1$ and $e_{s_y^2} = 1$.

Let C_{lits} represent the cost in terms of additional literals of a mapping $\mathcal{F}$. Then

$$C_{lits} = \sum_{y=0}^{q-1}(A_{\mathcal{F}(y),y} + b_{\mathcal{F}(y),y}.w) \tag{7.3}$$

where b is a boolean variable that is 1 if $e_{s_y^2}$=1 and $e_{s^1_{\mathcal{F}(y)}}$ =1 and the next states of s_y^2 and $s^1_{\mathcal{F}(y)}$ are not merged together, otherwise it is 0. $w = 3$ is used to ensure that C_{lits} is influenced by the next state mergers. The objective is to find a mapping $\mathcal{F}$ that minimizes C_{lits}.

The A matrix for *Mrgex* is given in Table 7.6. Only the mapping corresponding to $\mathcal{F}_2$ (entries shown in bold face), leads to the minimum of C_{lits}=3.

7.6 Merge Procedure

We have developed OMEN (Optimal MErged machiNe synthesis), a program that merges a functional (test) controller with a test (functional) controller. OMEN has two phases.

- Phase 1 is a preprocessing phase in which the A matrix for either of the cost functions is computed.

```
Procedure Preprocess
    { For each state $s_x^1$ in $M^1$
        { For each state $s_y^2$ in $M^2$
              Compute $A_{x,y}$ using the expressions in
              Section 7.4.3 for minimizing edges
              or Section 7.5.1 for minimizing literals;
        }
    }
```

- Phase 2 is a minimization process that minimizes either C_{edges} (Equation 7.1) or C_{lits} (Equation 7.3) using an A* algorithm. The STTs of k merged machines with the least cost are created. k is a user defined parameter. A switch setting chooses between the two cost functions, C_{edges} and C_{lits}. All of the k machines are synthesized and the one with the least number of product terms in the encoded cover (two-level) or factored form literals (multi-level) is selected. There may be a number of merged machines with the same minimum cost in terms of C_{lits} or C_{edges}, but different implementation cost, i.e., actual cost after state assignment and logic minimization. Therefore, instead of picking just one machine with minimum cost, it is prudent to look at a number of least cost machines. The A* algorithm used in this phase is very similar to the MERGE procedure described in Section 6.6.2 of Chapter 6 and therefore is not being explained here.

The A matrix for either of the cost functions is computed in $\mathbf{O}(p_*q)$ time. The worst case time complexity of the search procedure in OMEN is $\mathbf{O}(p!)$. However, since it is an A^* algorithm, it is consistently better than other equally informed

Example	states	inputs	outputs
Mrgex	4,3	1,1	2,2
Small	5,5	2,3	16,16
Newtrap	7,5	3,2	13,8
2comp	9,5	3,3	12,16
Ex3	9,9	3,3	16,16
Mark.1	15,9	5,3	16,16
Planet.1	48,9	8,3	19,16

Table 7.7: Characteristics of the example circuits

search strategies in that it always explores the least number of partial or complete solutions in the search space.

7.7 Experimental Results

OMEN was run on seven pairs of example FSMs. The characteristics of these example circuits are given in Table 7.7, where the first and second entries in each column refer to M^1 and M^2 respectively. *Newtrap* and *2comp* are each a pair of functional and test controllers that control hardware for the Newton-Raphson algorithm and 2's complement multiplication, respectively. The machines in *Small* (*Ex3*) are similar to M^2 (M^1) in *Newtrap* and *2comp.* M^1 for the last two examples have been chosen from the MCNC FSM benchmarks and M^2 for these examples are the same as M^1 in *2comp.*

Our experimental results indicate that different mergers lead to machines with different implementation costs. Let "best" ("worst") refer to the minimum (maximum) implementation cost for merging a pair of machines. In most cases, there is a significant difference between "best" and "worst". For the first three examples, we obtain "best" and "worst" by exhaustively generating and synthesizing all merged machines. For larger examples it is not possible to generate and synthesize all merged machines. Therefore we generated 100 random mergers, synthesized each merged machine, and obtained "best" and "worst". We found that, setting k=1 for two-level and k=5 for multi-level implementations produced merged machines with

Example	e	mv	en	pla	cpu	best	worst
Mrgex	8	8	6	96	0.1	96*	128*
Small	14	14	12	420	0.2	420*	455*
Newtrap	17	17	16	512	0.7	480*	544*
2comp	21	21	19	684	0.3	684	720
Ex3	28	28	25	1000	4.7	1000	1040
Mark.1	38	37	34	1496	3.1	1496	1628

Table 7.8: Results for two-level implementation

implementation cost equal to or very close to the "best" cost (see Section 7.6 for details of k).

Table 7.8 shows the characteristics of the merged machines produced by OMEN for two-level implementation with k=1. The entries refer to the number of edges (e), the cardinality of the minimum MV cover (mv), the cardinality of the minimum encoded cover (en), and the area of a PLA implementation of the merged machine produced by OMEN (pla). NOVA is used in its default mode for state assignment. *cpu* refers to the CPU time in seconds taken by OMEN on a Sun SPARCstation1. An asterisk after entries in the *best* and *worst* columns refers to the fact that these values have been obtained by exhaustively generating all merged machines. OMEN generated the optimal solution for all but one case.

Table 7.9 shows the characteristics of the merged machines produced by OMEN for multi-level implementation with k=5. The five least cost merged machines produced by OMEN are synthesized using JEDI and SIS, and the machine with the least number of factored form literals is selected. The entries refer to the number of factored form literals of the merged machine chosen (*lit*), the CPU time taken by OMEN to produce the STTs of the five merged machines (*cpu O*), the CPU time for synthesizing these machines using JEDI and SIS (*cpu J+Sis*), and the total CPU time (*cpu total*). For *Mrgex*, synthesizing only one machine with the least cost (k=1) would produce a machine with 14 literals. However, by synthesizing more than one machine with least cost (k=5), we are able to pick the best machine with 13 literals. The runtime of OMEN is sublinear in k. However, the synthesis time using JEDI and SIS increases linearly as a function of k. Therefore, k should be as small as possible.

Example	lit	best	worst	cpu O	cpu J+Sis	cpu total
Mrgex	13	13*	22*	0.1	20.5	20.6
Small	38	34*	52*	0.1	33.5	33.6
Newtrap	38	36*	64*	0.1	32	32.1
2comp	59	55	96	0.2	53.5	53.7
Ex3	108	96	147	0.7	81	81.7
Mark.1	153	150	237	0.8	127	127.8
Planet.1	503	503	625	148	1701	1849

Table 7.9: Results for multi-level implementation

Table 7.10 compares the results of STAMINA with OMEN for multi-level implementation. For each of the examples, the number of states in the state minimized machines produced by STAMINA is equal to the number of states in M^1. *O lit* (*S lit*) and *O area* (*S area*) refer to the number of factored form literals and logic area, respectively, of the merged machine produced by OMEN (STAMINA). Entries *O_syn cpu* and *S_syn cpu* refer to the time taken by OMEN, JEDI, SIS to synthesize the five machines and the time taken by STAMINA, JEDI, SIS to synthesize one machine, respectively. Only one machine needs to be synthesized in the latter case because STAMINA only produces one state minimal machine. The gate area has been obtained by performing technology mapping (in the area mode) using the *mcnc.genlib* and *mcnc_latch.genlib* libraries. The results show that the machines produced by OMEN have on average about 20% less factored form literals and gate area than those produced by STAMINA.

Finally, Table 7.11 illustrates the advantages of merging machines as opposed to implementing them separately. The entries represent the gate areas of M^1, M^2 and the multiplexer (*mux*) needed on the common control lines. *total* reflects the total area for separate implementation of the machines and M^m the area of the merged machine produced by OMEN. In all cases, the merged machines are significantly smaller than the machines implemented separately.

Example	O	S	O	S	O_syn	S_syn
	lit	lit	area	area	cpu	cpu
Mrgex	13	20	40	49	20.6	5.5
Small	38	42	83	91	33.6	8
Newtrap	38	52	87	109	32.1	8.1
2comp	59	83	130	161	53.7	13.3
Ex3	108	125	199	240	81.7	20.8
Mark.1	153	201	264	336	127.8	66.6
Planet.1	503	JEDI could not process STAMINA output				

Table 7.10: Comparison with STAMINA

Example	M^1	M^2	mux	total	M^m	ratio
Mrgex	28	27	8	63	40	0.63
Small	55	50	64	169	83	0.49
Newtrap	80	60	32	172	87	0.51
2comp	104	47	48	199	130	0.65
Ex3	124	106	64	294	199	0.68
Mark.1	160	130	64	354	264	0.75
Planet.1	669	133	64	866	785	0.91

Table 7.11: Merged versus separate implementation

7.8 Summary

We have shown that merging two orthogonal machines, as opposed to implementing them separately, provides significant savings in area. For two-level implementation, a cost function based on minimizing the number of edges in the merged machine has been proposed. If all the fanout edges of M^1 and all single fanout edges of M^2 are SPIs (refer to Section 7.4.2), then the proposed cost function is exact and is completely equivalent to minimizing the cardinality of a strictly output disjoint MV cover. Otherwise, the cost function is a heuristic measure, which according to our experience, reflects a good match between our predicted cost and the final implementation cost. For the multi-level case, a cost function based on minimizing the number of literals by considering all state pair mergers has been proposed. Even though this cost function is heuristic in nature, experimental results show a strong correlation between this cost function and the final implementation cost. These ideas have been incorporated in a program called OMEN. The run time for the examples is quite small. Even for the largest example with a search space of 10^8 mergers, the run time for OMEN is less than three minutes.

Chapter 8

Synthesis of 1-hot Coded Test Controller

8.1 Introduction

In this chapter we first present (in Section 8.2) a test control architecture that is based on Partition 2 shown in Fig. 2.2 in Chapter 2. This is a different partition between the off-chip and the on-chip test control circuitry as compared to the partition proposed in Chapter 3 and adopted throughout this dissertation. The architecture presented in this chapter incorporates a shift counter on the chip and assumes that the chip does not support the IEEE 1149.1 boundary scan architecture. In Section 8.2.1 we present the model of a number of test controllers controlling test plans in this architecture. In other sections of this chapter we present an approach to combining these test controllers into a merged controller, where the merged controller has a 1-hot coded state assignment. We show that a 1-hot encoding provides certain tangible benefits and may be an attractive alternative to a merged test controller implemented with a minimum number of flip-flops (Section 8.2.3). The benefits manifest themselves in smaller area, easy testability and ability to distribute elements of the controller near the controlled elements. The architecture presented in Section 8.2 is meant to provide an alternative to the bus-based IEEE 1149.1 compliant test control approach. However in Section 8.9 we show that the 1-hot coded test controller can be integrated with both the bus-based and the non bus-based IEEE 1149.1 compliant test control architectures.

8.2 Test Control Model

Our primary objective is to implement the test control circuitry for a number of test plans corresponding to various TDM embeddings in a circuit. The following are the salient features of the test control model.

- There are n test plans to be controlled and the number of states (phases) in test plan i is q_i.

- There are three counters on the chip. These are :

  1. A scannable *test counter* (TCNTR). A value specifying the number of test vectors to be applied in plan i is scanned in this counter prior to the execution of the plan. TCNTR has an input DEC-TC that decrements the counter. TCNTR asserts a signal TC when it counts down to 0.

  2. A non-scannable *shift counter* (SCNTR). It is reset by a signal RST-SC at the start of execution of a test plan and has an input INC-SC that increments the counter. Output SC is generated when the counter reaches a value equal to the length of the shift chain.

  3. A non-scannable *test plan counter* (TPCNTR) that specifies the test plan being executed. This counter is reset by a signal RST-TPC and incremented by a signal INC-TPC. When TPCNTR reaches the value n it generates a signal TPC which indicates that all test plans have been executed. The output of this counter is fully decoded and generates n control signals $tp_1, tp_2, \ldots, tp_n$.

- A Test-Mode (TM) signal is provided from off-chip that is 0 in *normal mode* and is 1 in *test mode*.

- Test data is stored off chip and shifted on-chip via a SDI (Shift-Data Input) pin. Test responses are shifted off-chip via a SDO (Shift-Data Output) pin and compared with stored valid response vectors.

Fig. 8.1 is a model of the test control environment.

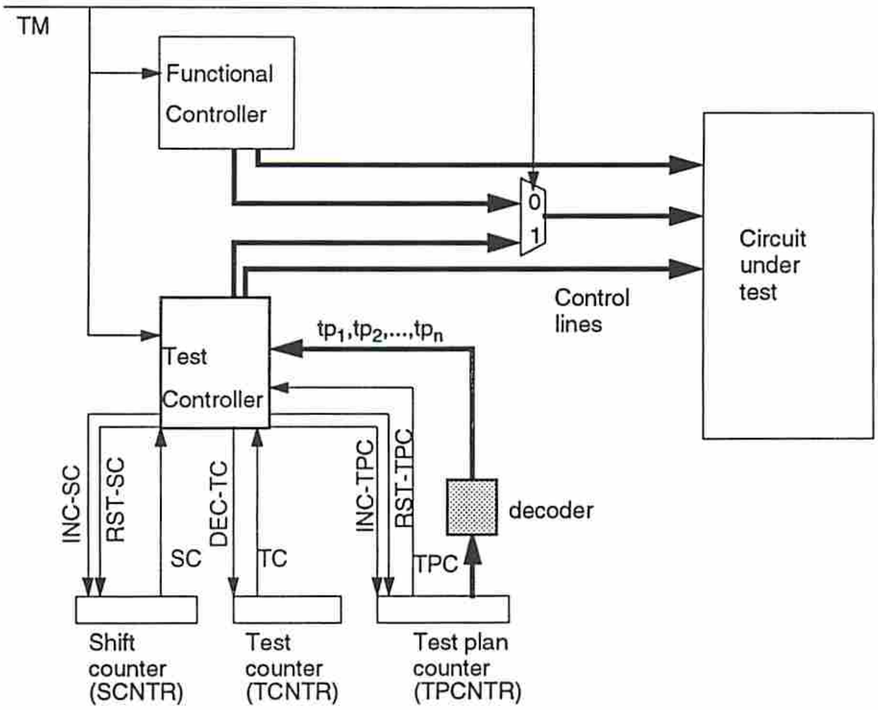


Figure 8.1: The control model

8.2.1 State Diagram of Overall Test Controller

The state diagram of the overall test controller (corresponding to the block *Test Controller* in Fig. 8.1) that sets up the test environment and executes each of n test plans is shown in Fig. 8.2. It is modeled as a Moore machine, M, represented by the 5-tuple $(I, O, S, \delta, \lambda)$. I is the set of input lines, O the set of output lines, S the set of states, δ the state transition function and λ is the output function.

- I={TC,SC,TPC,TM,$tp_1, tp_2, ..., tp_n$}, where tp_i is activated for test plan i.

- O={$c_1, c_2, c_3, ..., c_k$,INC-SC,DEC-TC,INC-TPC,RST-SC,RST-TPC,SHIFT} where $c_1, c_2, c_3, ..., c_k$ are the set of control lines controlling the circuit under test.

- S={$Idle_1$,$Idle_2$, Head, Tail, $s_a, s_b, s_1^1, ..., s_{q_1}^1, s_1^2, ..., s_{q_2}^2, ..., s_1^n, ..., s_{q_n}^n$}

- δ: I_{val} x S → S | I_{val} set of all input values in the space spanned by I

- λ : S → O_{val} | O_{val} set of all output values in the space spanned by O

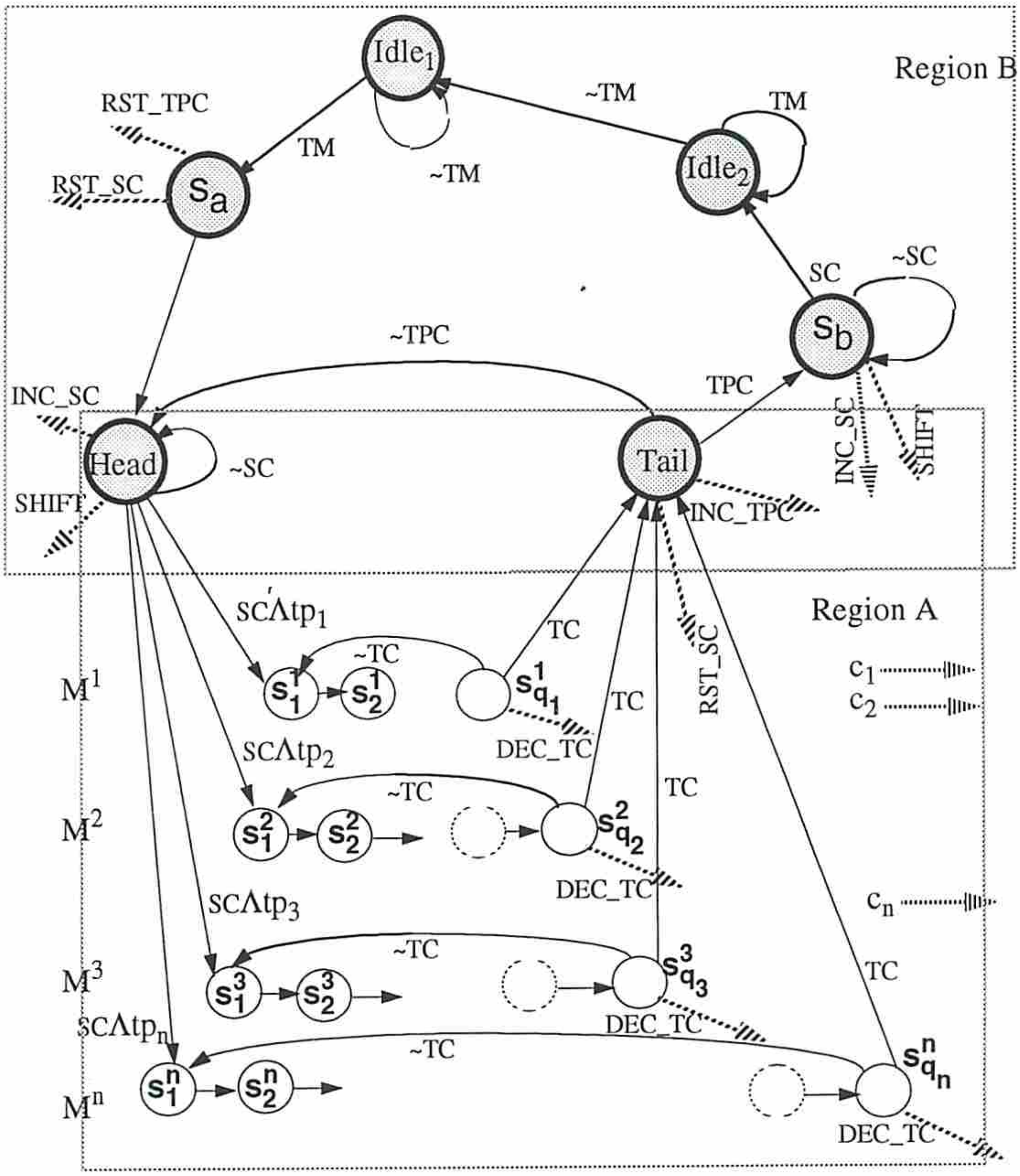


Figure 8.2: The overall test controller state diagram

In normal mode, when TM=0, M stays in state $Idle_1$. When TM=1, M goes to s_a enabling RST-TPC and RST-SC to reset TPCNTR and SCNTR. Then M remains in state Head until shift in(out) is completed. On completion of initial shift (SC=1) M goes to the state s_1^1 corresponding to *state 1* of test plan 1. M moves through the states of test plan 1 and in the last state ($s_{q_1}^1$) makes a decision either to loop back to s_1^1 or go to the Tail. This decision is based on the value of TC. If M is in Tail, and if all the test plans have not been executed (TPC=0), then M goes back to Head and shifts in the new seed for test plan 2 and shifts out the result for test plan 1. If TPC=1 and M is in Tail, then it moves to s_b and shifts out the results of the last test plan. Then M moves to $Idle_2$ and finally goes to $Idle_1$ when TM=0. Note that throughout this chapter the notation $\sim x$ is used to represent the negation of x in the figures.

This machine can be decomposed into two parts. One part consisting of states $Idle_1$, $Idle_2$, Head, Tail, s_a and s_b, deals with the set up process and is common to all test plans. The second part deals with the actual execution of the test plans. We can therefore model the overall controller as interacting submachines M^c (a common machine) and M^i,i=1,2,...,n, where M^i executes test plan i. Our primary focus will be on merging the submachines M^i into one machine, M^m.

8.2.2 Implementation Details of Submachines

M^c can be implemented using any classical technique. In addition to the Head and Tail states, M^i has q_i states. These states will be implemented using the 1-hot encoded state assignment, (one of the many choices in state assignment) and thus q_i flip-flops are required. From the examples used in this research, we have observed that the output logic for a 1-hot coded controller is trivial. Each control line is either driven directly by a flip-flop or at most by an OR(NOR) gate. We have thus restricted the output logic such that each output is driven at most by one OR/NOR gate. This restriction helps to prune the search space of candidate merged machines and from the examples we have looked at, this restriction is quite realistic.

8.2.2.1 Definitions

Following are some definitions of transitions (arcs) and states of a submachine M^i.

Definition 8.1 *δ_{entry} is an* **entry** *transition (arc) where δ_{entry} : $(SC \wedge tp_i, Head) \rightarrow s_1^i$. Each* (a,b) *represents an ordered pair where* a *is a function of inputs and* b *is the present state.*

Definition 8.2 *s_1^i is called the* **start** *state of M^i and is reached from* Head *via the entry transition.*

Definition 8.3 *δ_{exit} is an* **exit** *transition (arc) of M^i where δ_{exit} : $(TC, s_{q_i}^i) \rightarrow$ Tail.*

Definition 8.4 *$s_{q_i}^i$ is called the* **end** *state of M^i and* Tail *is reached from this state via a exit transition.*

Definition 8.5 *$\delta_{feedback}$ is called a* **feedback** *transition of M^i where $\delta_{feedback}$: $(\overline{TC}, s_{q_i}^i) \rightarrow s_1^i$.*

8.2.3 Advantages of 1-hot Encoding

Some advantages of a 1-hot code over an assignment using a minimum number of flip-flops are as follows.

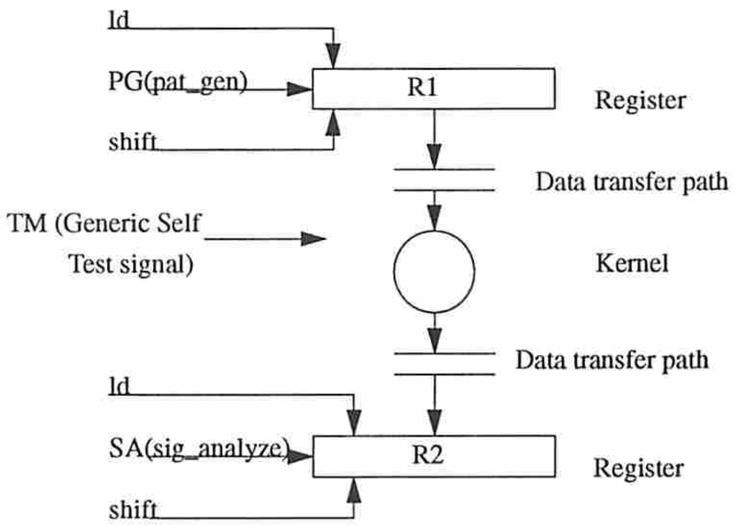
- The complexity of the next state and output logic is reduced as compared to a FSM encoded with minimum number of bits.
- The flip-flops in the controller can be distributed throughout the circuitry to reduce control wire routing overhead. In a test plan with multiple states controlling switch based scan or non-pipelined BIST TDMs, test data and results move from one register to another in each of the states. Each of these registers need to be controlled. Moreover data selectors such as multiplexers and sometimes complex combinational logic between sets of registers also need to be controlled. Functional registers can be increased in length by one or more bits and these extra bits can be used as the states of the 1-hot controller. The test control lines are thus generated very close to the modules that are being controlled.
- All cycles in the controller can be broken by appropriately controlling the primary inputs. This simplifies the sequential test generation problem.

- Output logic, if any, is fully tested during the execution of the test plans. The controller generates a walking 1 pattern in test mode. This constitutes a complete test set for output logic synthesized with OR/NOR gates.

It has been shown in [61] that there exists a close relationship between the synthesis approach chosen for an FSM and its testability. Appropriate state assignments enhanced the testability of some benchmark FSMs and resulted in a hardware overhead of 0%-30%. In [62] a new test algorithm for FSMs synthesized using the 1-hot code state assignment has been presented. The paper presents area, delay and test coverage data for a set of MCNC benchmark FSMs that are synthesized using the 1-hot code assignment and also synthesized using minimum number of flipflops. MUSTANG [60] is used in the latter case. All flip-flops in the minimum flip-flop FSMs have been made scannable. The FSMs have been placed and routed. It is interesting to note that, with the exception of one example, all the 1-hot coded FSMs have less area and delay than the minimum flip-flop ones.

8.3 Example Circuit

The control signals routed to BIST registers are shown in Fig 8.3. This figure shows a BILBO TDM embedding. Three control lines *shift*, *PG (SA)*, *TM* and possibly a load/hold signal *ld* are the inputs to the register acting as a PG (SA) in test mode. Note that in this control scheme there is no test bus and mode or configuration flip-flops are not associated with a register. Extra control lines are needed to reconfigure registers or provide different modes of operation. These control lines are driven from the test controller for the test architecture defined in Section 8.2. The truth table in Fig. 8.3 defines the functional relationship between the various signals. For example, when *TM* is 0, the *ld* signal has control over the registers. When *TM* is 1, a precedence relationship is present between the *shift* and the PG/SA signals. In this example the control line controlling the PG/SA mode also provides hold. If the hold mode is not needed then this control line can control a register having both the PG and SA modes. Even though there is no test bus local decoding is needed for every test register to control the actual control lines of the 1-bit cells constituting each register.



Truth table for test register

TM	ld	PG/SA	shift	Register Function
0	1	x	x	load
0	0	x	x	hold
1	x	x	1	shift
1	x	1	0	pat gen/sig analyze
1	x	0	0	hold

Figure 8.3: The BILBO TDM structure

Fig. 8.4 shows an example circuit. The circuit has three combinational blocks (kernels) $L1$,$L2$ and $L3$ to be tested. $PG1$, $PG2$ and $PG3$ are pattern generators and $SA1$ is a signature analyzer. The *ld* signal of each register and multiplexer select lines are activated by the functional controller in normal mode and by the test controller(s) in test mode. Test plans 1, 2 and 3, corresponding to the three embeddings of the BILBO TDM, are shown in Fig. 8.5. In *Test Plan 1*, the transition from Head to phase (state) 1 occurs when all registers have been initialized. At this time SC is set to 1 for one clock period.

In *state 1* of the test plan PG1 is activated by c_1 and SA1 is in the hold mode ($c_{10} = 0$). In *state 2* the test vector x_{inp} generated by PG1 is loaded into R1. In *state 3*, the response vector x_{out} from $L1$ is loaded in R1 through MUX1. In *state 4*, x_{out} is loaded in R2 from R1 through MUX3. In *state 5*, SA1 captures x_{out} from R2 through MUX4. The test plan sequences through the 5 states k times, where k is the *test length. Test length* of a test plan is the number of test vectors applied to a kernel. A signal TC is set to 1 when all tests have been applied. Test plans 2 and 3 operate in a similar way.

The test controllers associated with the test plans do not perform the set up task for the test environment e.g., initialize the BILBO registers or scan out the final signature. The test controller for a kernel will therefore not only have to execute the actual test plan shown, but will also have to set up the test environment. Since set-up is a common feature of all test plans, we can have a master or common controller (M^c) that performs all the housekeeping functions and hands over control to simpler controllers to execute different test plans.

In this chapter we focus on the problem of merging test controllers controlling test plans for BILBO TDMs and assume that test plans are executed serially, i.e. one after the other. Each test plan is assumed to be non-pipelined. The methodology is applicable to other TDMs (such as the SCAN TDM) and to test plans that operate in a pipelined manner.

8.4 Problem Formulation

The sequencing and activation of the submachines depends on the value of the test plan counter. Intuitively it appears that, instead of implementing each of the

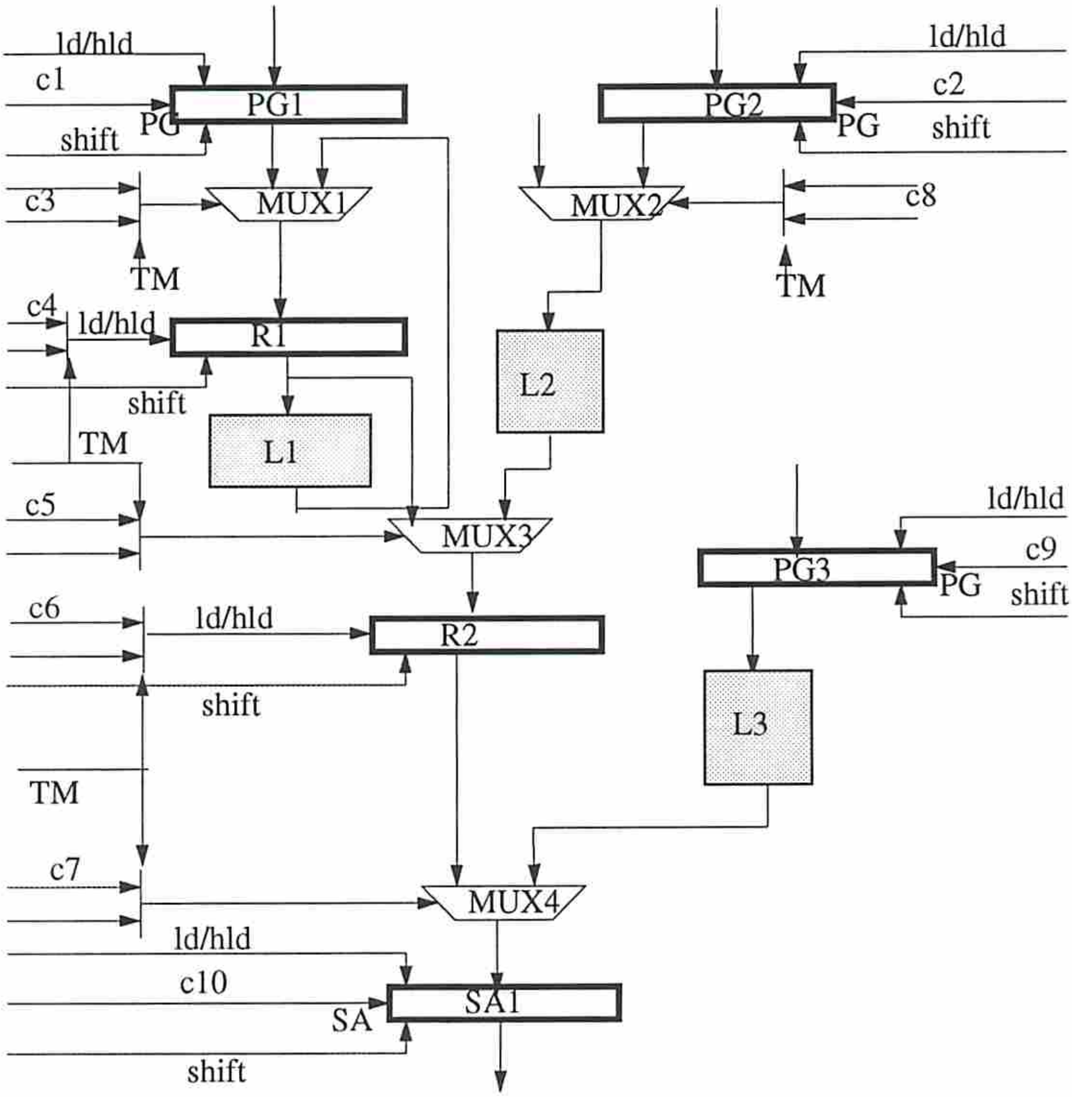


Figure 8.4: Ckt 1

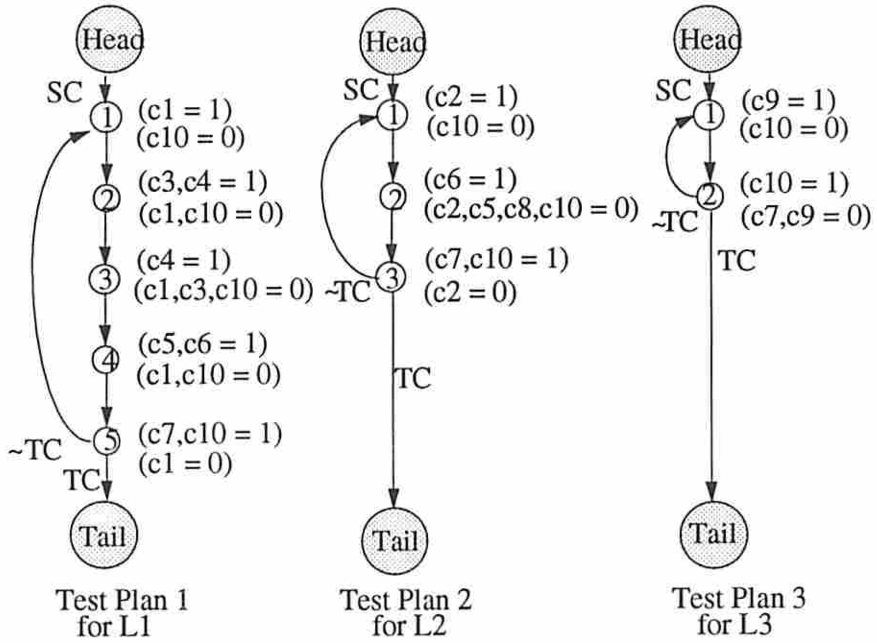


Figure 8.5: Test plans 1,2, and 3 for Ckt 1

submachines as individual entities, we may save logic by sharing logic among the submachines. This sharing, which can be done via a merging process. The degree of sharing will depend on the degree of interaction between the control lines activated by the submachines.

We address the following merging problem.

Given n *submachines controlling* n *test plans, obtain one machine* M^m *that has, (1) Head and Tail as its entry and exit states, (2) controls each of the test plans in turn in accordance to the input from the test plan counter, and (3) has a 1-hot coded implementation with minimal next state and output logic.*

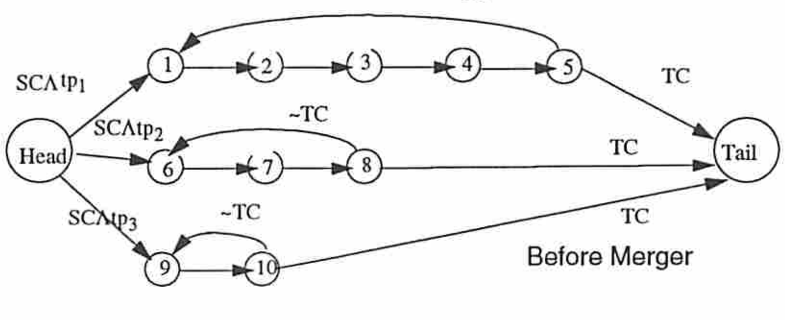


Figure 8.6: An example of an unoptimized merger

The portion of the state diagram in Region A of Fig. 8.2 satisfies conditions (1) and (2) of the merging problem. In our running example, Fig. 8.6 represents the state transition graph of M^m. The number of states in this merged machine is $\sum_{i=1}^{n} q_i = 10$, where $n = 3$. Each of these states activate some control lines. If all the states of a test plan are compatible with all the states of other test plans, then Fig. 8.7 is an example of a M^m which satisfies conditions (1) and (2) and has minimal next state logic.

8.5 Solution Approach

Since satisfying the first two conditions in the merging problem is trivial, we will focus our attention on satisfying the third condition. The merged machine M^m of Fig. 8.6 will be used as a starting point. The state transition table of this machine is shown in Table 8.1. The unspecified outputs appear as -'s (don't cares) in the table. Note that the states have been simply represented by numbers.

We will use the pair chart technique to find all pairs of compatible states. Some observations are helpful.

- The states of any one M^i are mutually incompatible.
- It is not necessary to make multiple passes over the pair chart as is required for general machines. This is because for a pair of output compatible states s_i and s_j (from two different submachines), an input tp_x specifies the next state only for s_i or s_j but not both.

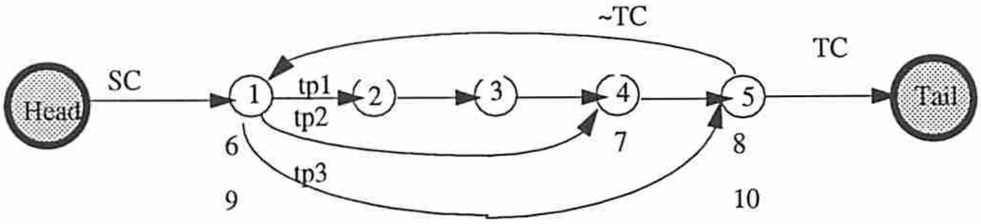
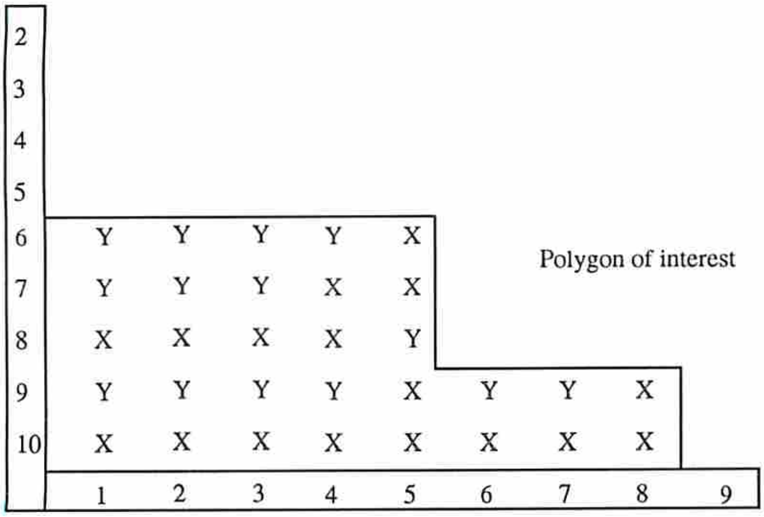


Figure 8.7: An example of a best case merger



2									
3									
4									
5									
6	Y	Y	Y	Y	X				
7	Y	Y	Y	X	X				
8	X	X	X	X	Y				
9	Y	Y	Y	Y	X	Y	Y	X	
10	X	X	X	X	X	X	X	X	
	1	2	3	4	5	6	7	8	9

Figure 8.8: The pair chart

inputs			states		outputs									
tp1	tp2	tp3	p_s	n_s	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
1	0	0	1	2	1	-	-	-	-	-	-	-	-	0
1	0	0	2	3	0	-	1	1	-	-	-	-	-	0
1	0	0	3	4	0	-	0	1	-	-	-	-	-	0
1	0	0	4	5	0	-	-	-	1	1	-	-	-	0
1	0	0	5	1	0	-	-	-	-	-	-	-	-	1
0	1	0	6	7	-	1	-	-	-	-	-	-	-	0
0	1	0	7	8	-	0	-	-	0	1	-	0	-	0
0	1	0	8	6	-	0	-	-	-	-	1	-	-	1
0	0	1	9	10	-	-	-	-	-	-	-	-	1	0
0	0	1	10	9	-	1	-	-	-	-	-	-	0	1

Table 8.1: The transition table for Ckt 1

Fig. 8.8 is the pair chart where only the entries in the polygon are of interest. The other entries denote compatibility between states in the same test plan and as mentioned above, states in the same plan are mutually incompatible. In the pair chart Y stands for a compatibility and X denotes an incompatibility.

From the pair chart the *maximum compatibles* (MCs) and the *prime compatibles* (PCs) can be found [63]. The PCs for this problem are the same as the MCs because the *class sets* [63] of the MCs are null sets. Fig. 8.9 shows the 2-compatibles, 3-compatibles and the PCs.

In a classical state minimization technique the next step would be to find *one* minimal PC cover for the set of states in the state transition table [63]. Then state, input and output encodings are determined so as to minimize the resulting logic implementation [48, 42, 60, 59]. Herein lies the major difference between classical techniques and ours. *We have already decided to use the 1-hot code.* To find the simplest hardware realization, it is necessary to consider *all* the minimal PC covers for this machine. We shall show later that the actual search space in most cases is, in fact, larger than the space of all minimal PC covers.

2 compatibles :

1-6,
1-7,
1-9

2-6,
2-7,
2-9

3-6,
3-7,
3-9

4-6,
4-9

5-8

6-9,

7-9

3-compatibles :

1-6-9, 1-7-9, 2-6,-9, 2-7-9, 3-6-9, 3-7-9, 4-6-9 ,

PC's :: 1-6-9, 1-7-9, 2-6-9, 2-7-9, 3-6-9, 3-7-9, 4-6-9, 5-8, 10

Figure 8.9: The compatibles and prime compatibles

To illustrate how the choice of a minimal covers affects the implementation of our merged 1-hot coded controller, consider the six possible combinations of PCs shown in Fig. 8.10, all of which give a minimal cover. Figs. 8.11(a) and (b) are the state transition graphs of the merged controller corresponding to two different choices of minimal covers. Hardware implementation of the two graphs shows that the implementation of the state transition graph of Fig. 8.11(a) has less logic than the one for Fig. 8.11(b). The output logic is the same in both cases. Counting the edges(arcs) of the state transition graphs shows that Fig. 8.11(a) has less arcs then

Fig. 8.11(b). In the next section we compute bounds on states, arcs, next state and output logic of a merged machine. In Section 8.7 we present a branch and bound procedure that builds a state transition graph of the minimal cost merged machine.

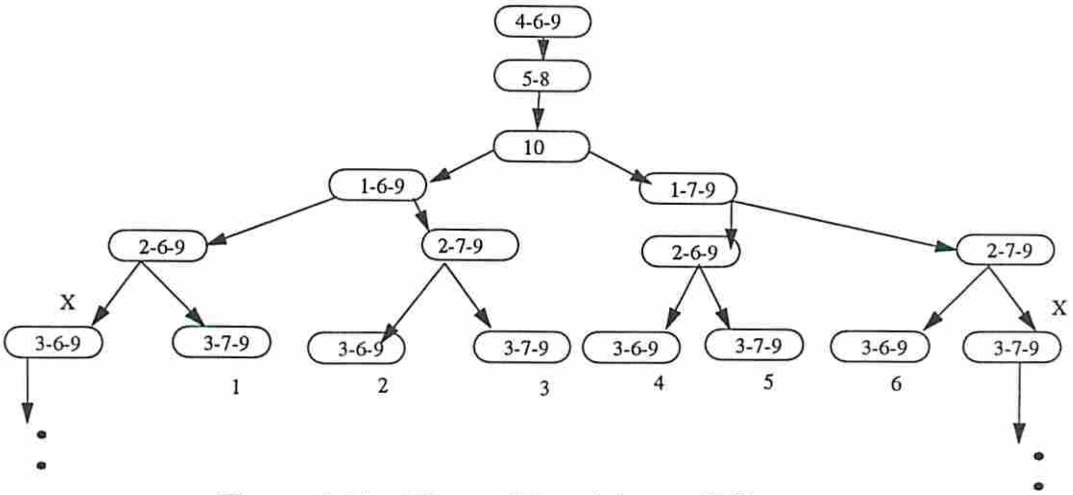


Figure 8.10: All possible minimum PC covers

8.6 Bounds on Implementation Cost

Recall that M^i for i=1,2,...,n has q_i states. Two cases need to be considered.

1. Each of the q_i are distinct and can be ordered in a strictly decreasing manner, i.e.

$$q_1 > q_2 > q_3 > ... > q_n \tag{8.1}$$

2. The q_i are not distinct and the states can be ordered in a non-increasing manner, e.g.

$$q_1 = q_2 = ... = q_{t_1} > q_{t_1+1} = q_{t_1+2} = ... = q_{t_2} > ... > q_{t_{S-1}+1} = q_{t_{S-1}+2} = ... = q_{t_S} \tag{8.2}$$

The M^i with equal number of states are grouped together. In case 2, there are S such *state groups.* For case 1, *S=n.*

Fig. 8.12 illustrates the concept of state groups. The STG's of six submachines are shown in Figs. 8.12(a),(b),...,(f). The number of states of each of the machines are given in the figures and they are partitioned into four state groups, i.e., S=4.

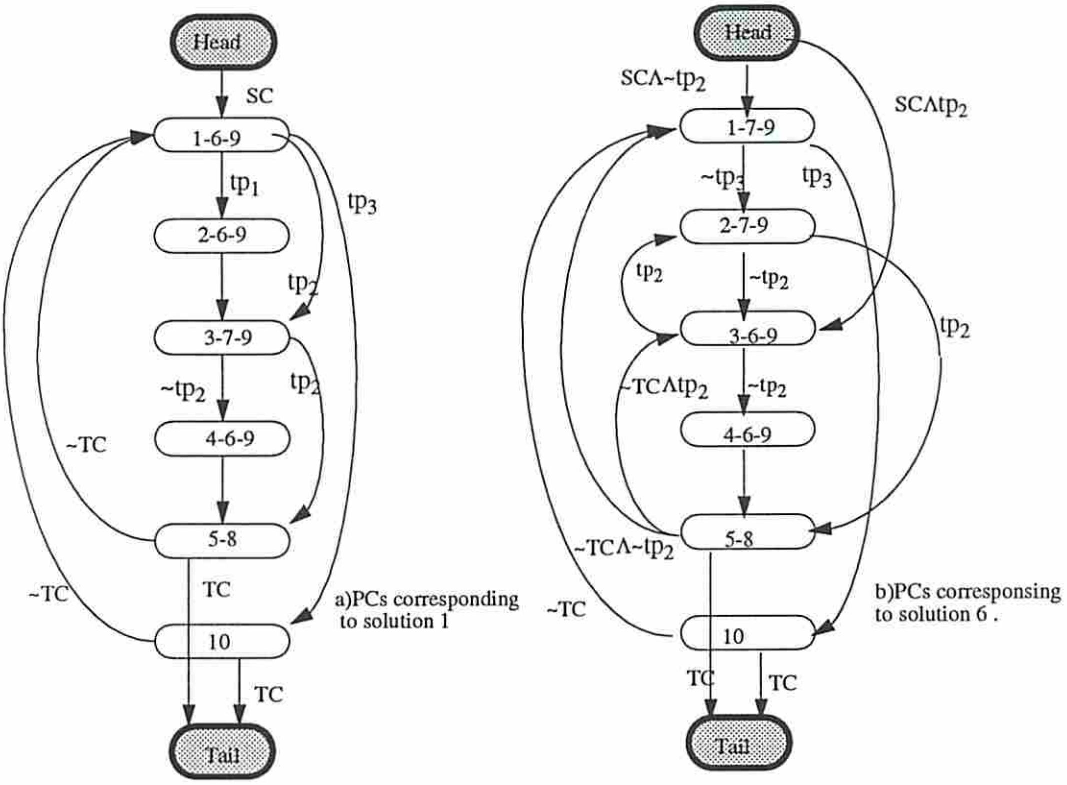


Figure 8.11: State transition graphs for two choices of minimum covers

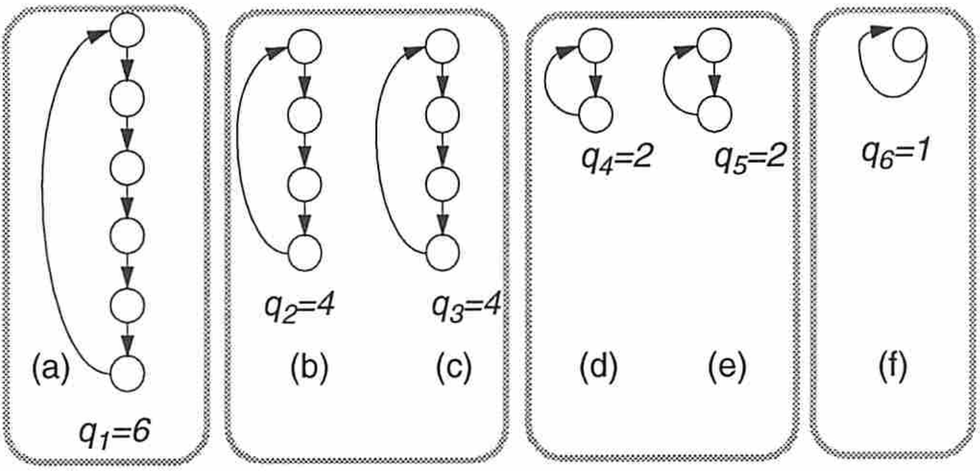


Figure 8.12: Illustration of state groups

8.6.1 Computing Implementation Cost

We can think of transition arcs between two states being broken up into two parts, the outgoing arc (from a state) and the incoming arc (to a state). The existence of a single outgoing transition arc from a state implies that no logic is needed for the implementation of that outgoing arc. However when a state has more than one outgoing transition arc, then each transition is a function of some subset of $\{tp_1, ..., tp_n\}$, and a lower bound of the logic needed to "implement" each outgoing arc is a single 2 input AND gate. Such a gate implements the conjunction (product) of two literals. We will assign a cost of 1 to the conjunction of 2 literals (e.g., 2 input AND gate). So each outgoing arc has a cost of at least 1. The conjunction of k literals (e.g., k input AND gate) has cost $k - 1$. To implement two incoming transition arcs to a state we need the disjunction of 2 literals (2 input OR gate). Assign a cost of $k - 1$ to the disjunction of k literals. Thus the cost of implementing k incoming arcs to a state is $k - 1$. The negation operation is implemented by an inverter with a cost of 1.

8.6.1.1 Next State Logic

Fig. 8.13(a) represents the STG of a merged machine. This is a merged machine corresponding to four submachines partitioned into four state groups and activated by tp_1, tp_2, tp_3 and tp_4. Fig. 8.13(b) is a gate level implementation of the merged machine. The flip-flops in this figure are provided the same labels as the states in the merged machine. We focus on state 2 and see how the outgoing arcs are implemented. Each outgoing arc is implemented by a 2 input AND gate with inputs as shown in the figure. Each 2 input AND gate contributes 1 unit towards the total cost of 3. Fig. 8.13(c) is another implementation of the three outgoing arcs. In this implementation the arc from state 2 to state 3 is implemented as (state 2)*$\overline{tp_2} * \overline{tp_3}$, where $\overline{tp_i}$ represents the complement of tp_i. We assume that complements of all inputs are available. Since the cost of this implementation is 4 we prefer the previous implementation (Fig. 8.13(b)).

Fig. 8.14(a) is a partial STG of a merged machine where the eleven submachines are partitioned into four state groups. The arcs that distinguish between the various state groups emanate from state 2 as shown. Figs. 8.14(b) and (c) show two methods of implementing the outgoing arcs from state 2. In the first method (Fig. 8.14(a)), each outgoing arc for all state groups except the largest is implemented as the conjunction (AND) of state 2 with the disjunction (OR) of all the test plan inputs that correspond to that particular state group. The arc corresponding to the largest state group is implemented as the conjunction of state 2 and the negation of the disjunctions of sets of test plans that correspond to each of the other state groups. Fig. 8.14(c) is a straightforward implementation of the outgoing arcs. The cost associated with these two alternate approaches are shown in the figures and for this example the first approach leads to a smaller implementation cost. In implementing the outgoing arcs of every state of a machine we will evaluate both these approaches and use the one leading to a smaller implementation cost.

8.6.2 Lower Bound of States, Arcs and Next State Logic

Proposition 8.1 *The greatest lower bound or* inf *of the number of states in M^m is* max*(q_i) which, in terms of the ordered sequences in Equations 8.1 and 8.2, is q_1.*

We next state a theorem that gives the *greatest lower bound* of the number of arcs in (the state transition graph of) M^m. We will then establish a relationship between the number of arcs and the implementation complexity of the next state logic of M^m.

Theorem 8.1 *The* inf *of the number of arcs in the state transition graph of M^m is q_1+S+1, where q_1 is* max*(q_i) and* S *is the number of state groups.*

Proof : The proof is by induction. Without loss of generality we can assume that the submachines have been ordered in a non-increasing sequence with respect to the number of states and subscripted as 1,2,...,n.

Let n=1. Then S=1. There are $q_1 - 1$ arcs on the path connecting s_1^1 to $s_{q_1}^1$. There is one arc from the *Head* to s_1^1, one from $s_{q_1}^1$ to *Tail* and the last one is the feedback arc connecting $s_{q_1}^1$ to s_1^1. Since n=1, M^1 and M^m are the same. Thus the total number of arcs is q_1+2 = q_1+S+1.

For n=2, S could be either 1 or 2. S=1 implies that both M^1 and M^2 have the same number of states. Assuming that there are no incompatibility between any pair of states in M^1 and M^2, both submachines can be represented by the same state

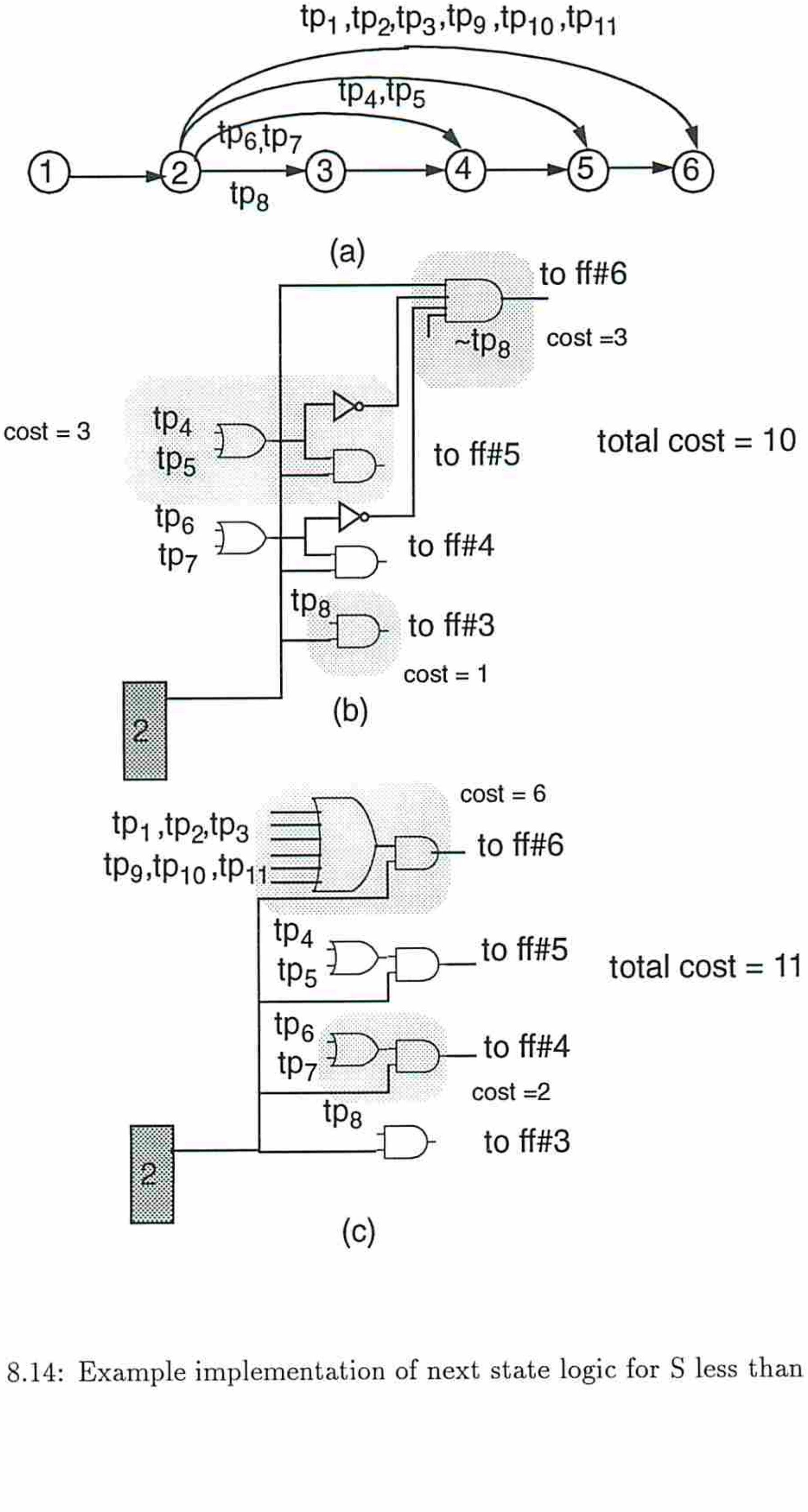


Figure 8.14: Example implementation of next state logic for S less than n

transition graph, in which case the total number of arcs is q_1+S+1. The situation S=2 is interesting because the two submachines to be merged have unequal number of states. By our ordering principle, $q_1 > q_2$. Now q_1+2 = q_1+S arcs are *required* for M^1. When we merge M^2 to M^1 we will have to add *at least* one additional arc over and above the arcs already used for M^1. This arc is necessary to distinguish between the longer and the shorter test plans. Thus the lower bound on the number of arcs is q_1+S+1.

Assume that the bound holds for some $n = k - 1$. We will show that the same bound holds when we add another submachine. There are two cases. If q_k equals q_{k-1} then no additional arcs need be added, and since the lower bound for $n = k-1$ was q_1+S+1, and S has not changed, the lower bound for $n = k$ is correct. If $q_k > q_{k-1}$, then we need to add *at least* one arc to the merged machine built up so far. Thus the lower bound is q_1+S+1 where the arc to be added is reflected in the value of S.

This proves that $q_1 + S + 1$ is a lower bound. Given a set of machines M^i, i=1,2,...,n, we can construct a merged machine M^m having $q_1 + S + 1$ arcs. Thus $q_1 + S + 1$ is the *inf.* □

Corollary 8.1 *Merging start states s_1^i to form s_1^m and merging end states $s_{q_i}^i$ to form $s_{q_1}^m$ for i=1,2,...,n is a* necessary *condition for achieving the* inf *of arcs in M^m, where s_j^m represents a state of M^m.*

Proof : Only the outline of the proof is given. Without loss of generality consider the case n=2 and $q_1 > q_2$. If the start and end states are not merged together, then three cases need to be considered. (1) Start states are not merged but end states are merged, (2) end states are not merged but start states are merged and (3) both start and end states are not merged. Recall the proof of Theorem 8.1 and note that case 1 requires at least *two* additional (entry and feedback for tp_2) arcs, case 2 requires at least *two* additional arcs (feedback and exit for tp_2) and case 3 requires at least *three* additional arcs (entry, exit and feedback for tp_2). □

Fig. 8.15 illustrates the concept of start and end states. In the figure, the STGs of five submachines have been shown. The start and end states have been shown shaded in vertical lines and crosshatched, respectively. The merged machine shown in the figure has been implemented with all the start states and the end states of the submachines merged into the start and end states of the merged machine.

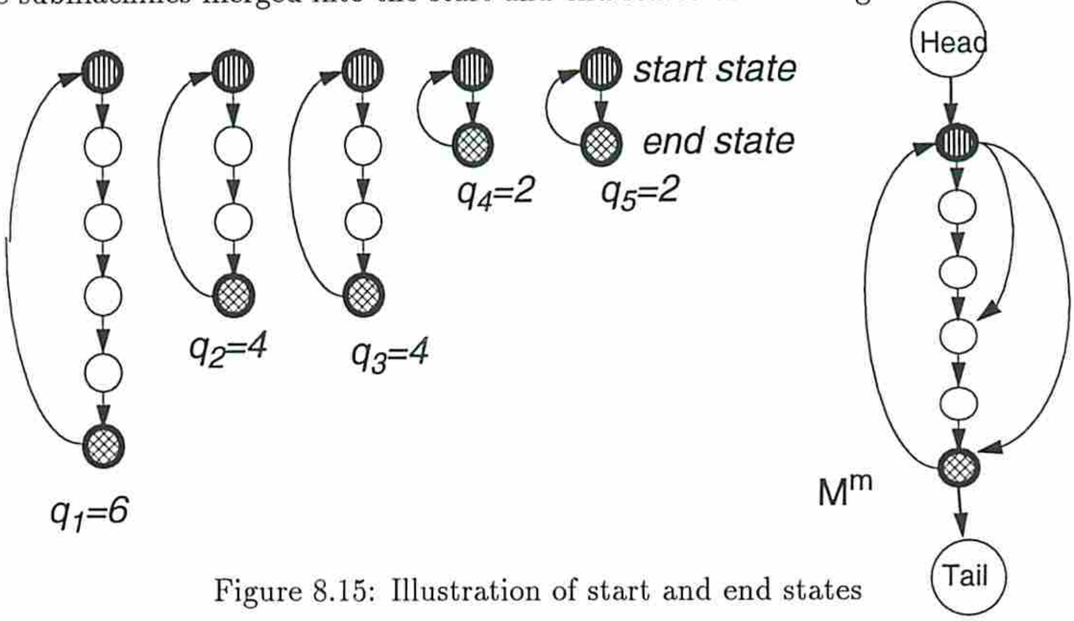


Figure 8.15: Illustration of start and end states

Theorem 8.2 *When merging a number of submachines into a merged machine M^m, the cost of the next state logic when all the arcs that distinguish different state groups emerge from the same state s_i^m is less than or at most equal to the cost of the next state logic when the arcs emerge from different states.*

Proof : Let $K_1, K_2, \ldots, K_S$ represent sets of test plans (submachines) in each of the S state groups. k_i is the number of test plans in set K_i. Let $k_1 \geq k_2 \geq ... \geq k_S$. r is the number of groups with only one member if $S < n$, else if $S = n$ then $r = n - 1$. Consider first the case when all the outgoing transition arcs emerge from the same state. Let L_{same} be the total cost of implementing the outgoing arcs. Then

$$L_{same} = min(n, \sum_{i=2}^{S} k_i + 2(S-1) - r) \tag{8.3}$$

This formulation is derived easily from Section 8.6.1.1.

Now assume that the S groups have outgoing arcs from some t ($t > 1$) states in M^m. Thus the groups are divided into t sets, $V_1, V_2, ..., V_t$. Let p_i be the number of groups assigned to V_i and r_i be the number of groups in V_i having only one test plan each. Thus $0 \leq r_i \leq p_i$, $i = 1, ..., t$. $V_i \cap V_j = \oslash, i \neq j$.

Note that the set V_t will contain exactly one group i with k_i test plans. Thus the state s_j^m which corresponds to the set V_t, has only one outgoing arc and this arc does not need any logic for its implementation. Also if s_x^m corresponds to the set V_{t-1}, then s_x^m "comes before" s_j^m.

The total logic needed to implement all these outgoing arcs is
$L_{diff} = min\{n, \sum_{K_i \in V_1} k_i + 2p_1 - r_1\} + min\{(n - \sum_{K_i \in V_1} k_i), \sum_{K_i \in V_2} k_i + 2p_2 - r_2\} +$
$min\{(n - \sum_{K_i \in V_1 \cup V_2} k_i), \sum_{K_i \in V_3} k_i + 2p_3 - r_3\} + ...$
$+ min\{(n - \sum_{K_i \in V_1 \cup ... \cup V_{t-2}} k_i, \sum_{K_i \in V_{t-1}} k_i + 2p_{t-1} - r_{t-1}\}$

If $(n \leq \sum_{K_i \in V_1} k_i + 2p_1 - r_1)$ then $L_{diff} > n \geq L_{same}$ because L_{same} is at most n,
else if $((n > \sum_{K_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{K_i \in V_1} k_i) \leq \sum_{K_i \in V_2} k_i + 2p_2 - r_2))$ then $L_{diff} > n \geq L_{same}$,
else if $((n > \sum_{K_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{K_i \in V_1} k_i) > \sum_{K_i \in V_2} k_i + 2p_2 - r_2) \wedge ((n - \sum_{K_i \in V_1 \cup V_2} k_i) \leq \sum_{K_i \in V_3} k_i + 2p_3 - r_3))$ then $L_{diff} > n \geq L_{same}$,

.

.

.

else if $((n > \sum_{K_i \in V_1} k_i + 2p_1 - r_1) \wedge ((n - \sum_{K_i \in V_1} k_i) > \sum_{K_i \in V_2} k_i + 2p_2 - r_2) \wedge ... \wedge ((n - \sum_{K_i \in V_1 \cup ... \cup V_{t-2}} k_i) \leq \sum_{K_i \in V_{t-1}} k_i + 2p_{t-1} - r_{t-1}))$ then $L_{diff} > n \geq L_{same}$
else $L_{diff} = \sum_{K_i \in V_1 \cup ... \cup V_{t-1}} k_i + 2\sum_{i=1}^{t-1} p_i - \sum_{i=1}^{t-1} r_i$.

Lets look at the last case in more detail and try to minimize L_{diff}. Max$\sum_{i=1}^{t-1} r_i = S - 1$ and $min(\sum_{K_i \in V_1 \cup ... \cup V_{t-1}} k_i)$ occurs when K_1 is allocated to V_t. Now $\sum_{i=1}^{t-1} p_i = S - 1$. Therefore L_{diff} in this case is $\sum_{i=2}^{S} k_i + (S-1)$.

Consider two cases. Case (1) S=n. Then $L_{same} = min(n, \sum_{i=2}^{S} k_i + 2(n-1) - (n-1))$. Since $\sum_{i=2}^{n} k_i = n-1$, $L_{same} = min(n, (2n-2))$. $L_{diff} = 2n - 2$. If $2n - 2 \leq n$ then $L_{same} = L_{diff}$, else $L_{same} < L_{diff}$. But $2n - 2 \leq n$ implies that $n \leq 2$ which is the case when the outgoing arcs will always emerge from the same state and there is no meaning of L_{diff}. Therefore for S=n $L_{diff} > L_{same}$. Case (2) $S < n$. The minimum value of the term $\sum_{i=2}^{S} k_i + 2(S-1) - r$ in L_{same} occurs when $r = S - 1$ (r cannot be equal to S since that means S=n). Thus $L_{same} = min(n, \sum_{i=2}^{S} k_i + (S-1))$. Comparing L_{same} and L_{diff} we conclude that $L_{diff} \geq L_{same}$.

The cost of implementing the incoming arcs is the same in both cases. Thus the complexity of next state logic when all the arcs that distinguish different groups of test plans emerge from the same state is less than or at most equal to the complexity of arcs emerging from different states. □

Figs. 8.16(a) and (b) show two merged machines with the same number of arcs. In Fig. 8.16(a) there are two outgoing arcs that emanate from the same state (the start state) whereas in Fig. 8.16(b) the outgoing arcs emanate from two different states (states 1 and 2). By the preceding theorem the machine in (b) has more implementation cost than the machine in (a).

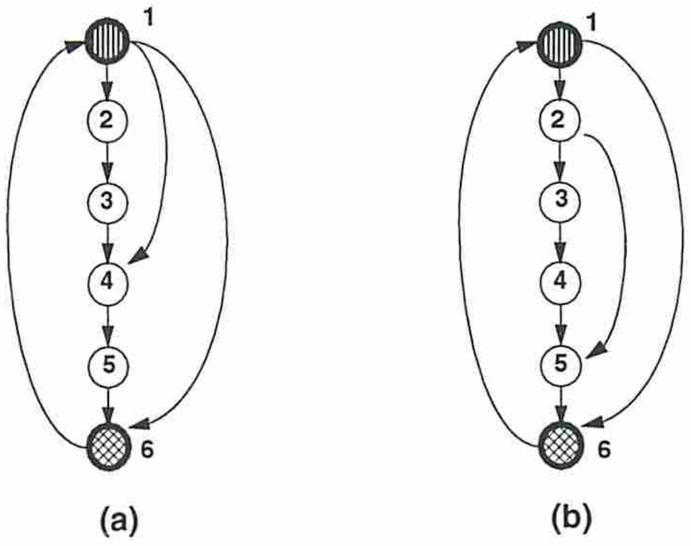


Figure 8.16: Machine with outgoing arcs emanating from different states

Lemma 8.1 *The following two conditions are* sufficient *for achieving the minimum cost next state logic in a merged machine M^m: (1) the number of arcs in M^m equals the* inf *of arcs, and (2) all the outgoing transition arcs that distinguish various state groups emerge from the same state s_i^m.*

Proof : In any merged machine the number of arcs has to at least equal the *inf* of arcs. From Theorem 8.2 we know that the implementation cost of the outgoing arcs for a machine M^m where all arcs emerge from one state is less than or equal to the implementation cost of the outgoing arcs for another machine $\mathcal{M}^m$ with the same number of arcs but which do not all emerge from the same state. Consider the following two cases.

- Case 1. Let the number of arcs in machine M^m equal the *inf* of arcs and all arcs emerge from the same state. Let $\mathcal{M}^m$ be another machine with arcs also emerging from the same state but the number of arcs is greater than the *inf* of arcs. We will show that the cost of next state logic c_{M^m} of M^m is less than the cost of the next state logic $c_{\mathcal{M}^m}$ of $\mathcal{M}^m$. The cost of M^m ignoring the entry, exit and feedback arcs is

  $$c_{M^m} = min(n, \sum_{i=2}^{S} k_i + 2(S-1) - r) + (S-1) \qquad (8.4)$$

  where the last term (S-1) represents the cost of the incoming arcs (OR gate cost). The cost of $\mathcal{M}^m$ is given by

  $$c_{\mathcal{M}^m} = min(n, \sum_{i=2}^{S'} k'_i + 2(S'-1) - r') + (S'-1) \qquad (8.5)$$

  where S' represents the number of modified state groups, r' is the number of such state groups having one test plan each and k'_i is the number of test plans in the ith modified state group. Modified state groups reflect the fact that the original state groups have been split, i.e., instead of one arc representing an entire state group, more than one arc is used in $\mathcal{M}^m$. It is obvious that $S' > S$. We will now show that $(\sum_{i=2}^{S'} k'_i + 2(S'-1) - r') > (\sum_{i=2}^{S} k_i + 2(S-1) - r)$ or equivalently, $(\sum_{i=2}^{S'} k'_i + S' + (S' - r')) > (\sum_{i=2}^{S} k_i + S + (S - r))$. Note that $\sum_{i=2}^{S} k_i$ is equal to n minus the number of test plans in the largest state group. For $\mathcal{M}^m$ the number of test plans in the largest (modified) state group cannot be greater than that for M^m because this would lead to an illegal machine. Therefore $\sum_{i=2}^{S'} k'_i \geq \sum_{i=2}^{S} k_i$.

  We will show that $S' - r' \geq S - r$ or $S' - S \geq r' - r$. $\mathcal{M}^m$ has more arcs than M^m because state groups are split. In the extreme case all the $S' - S$ extra arcs represent single test plans which results in $S' - S = r' - r$. Else $S' - S > r' - r$. Since $S' > S$, we have shown that

  $$(\sum_{i=2}^{S'} k'_i + 2(S'-1) - r') > (\sum_{i=2}^{S} k_i + 2(S-1) - r) \qquad (8.6)$$

Using Equation 8.6 and the fact that $S' > S$, we prove that $c_{M^m} < c_{\mathcal{M}^m}$.

- Case 2. Let M^m be the same machine as defined in Case 1, and let $\mathcal{M}^m$ be another machine which has more arcs than the *inf* of arcs and these arcs do not emerge from the same state in $\mathcal{M}^m$. We will show that $c_{M^m} < c_{\mathcal{M}^m}$ for this case also.

  Let $\mathcal{M}^{m'}$ be another merged machine with the same number of arcs as that in $\mathcal{M}^m$ but the arcs of $\mathcal{M}^{m'}$ emerge from the same state in $\mathcal{M}^{m'}$. From Theorem 8.2 we can show that $c_{\mathcal{M}^{m'}} < c_{\mathcal{M}^m}$. But $c_{M^m} < c_{\mathcal{M}^{m'}}$ by Case 1 of this proof. Therefore $c_{M^m} < c_{\mathcal{M}^m}$.

The proof of the lemma therefore follows from Cases 1 and 2 and Theorem 8.2. □

Corollary 8.2 *Let $k_1, k_2, ..., k_S$ be the number of test plans in the S state groups, where these groups are ordered such that $k_1 \geq k_2 \geq ... \geq k_S$. Let* r *be the number of groups which have only one test plan each if $S < n$ else $r = n - 1$. Then the minimal cost of the next state logic, denoted by C_{nsl} of a machine with number of arcs equal to the* inf*, is given by the following formula.*

$$C_{nsl} = min\{n, \sum_{i=2}^{S} k_i + 2(S-1) - r\} + S + 3 \tag{8.7}$$

Proof : From Equation 8.3 the cost of implementing the outgoing arcs excluding entry, exit and feedback arcs is $min(n, \sum_{i=2}^{S} k_i + 2(S-1) - r)$. The cost of implementing the incoming arcs for the S groups is *S-1*. The cost of implementing the three arcs mentioned above is 4. □

Lemma 8.2 *If the number of state groups S is equal to n, then a lower bound on the cost of next state logic of a machine M^m having p arcs more than the* inf *is given by the following formula.*

$$C_{lb} = 2n + 3 + 2p. \tag{8.8}$$

Proof : Consider the case when the number of arcs of M^m equals the inf of arcs. Then the minimum cost of next state logic is obtained by substituting $S = n$ in Equation 8.7 and is equal to *min(n,(n-1) + 2(n-1) - (n-1)) + n+3*. Simplifying the equation we get *min(n,2n-2)+n+3* which equals *2n+3* since *min(n,2n-2)* $= n$ (since n $\geq$ 2). Every arc beyond the absolute minimum necessary for implementing M^m (i.e., the *inf* of arcs) requires *at least* one AND gate with cost of 1 (outgoing arc) and an OR gate with a cost of 1 (incoming arc). Therefore C_{lb}=$2n + 3 + 2p$ and the lower bound is a monotonically increasing function of p. □

Lemma 8.3 *If the number of state groups S is less than n, then a lower bound on the cost of the next state logic of a machine M^m having p arcs more than the* inf *is given by the formula*

$$C_{lb} = min\{n, \sum_{i=2}^{S+p} k_i + 2(S + p - 1) - r'\} + (S + p) + 3 \qquad (8.9)$$

where r' is the total number of arcs that represent single test plans.

Proof : The total number of arcs distinguishing the machines constituting M^m is (S+p). Since the cost of outgoing arcs from the same state is less than or equal to the cost of the arcs emanating from different states (Theorem 8.2), we substitute (S+p) in place of S in Equation 8.7. C_{lb} increases monotonically with p (Case 1 Lemma 8.1). □

The lower bound computations in Equations 8.8 and 8.9 assume that the number of states (flip-flops) in the merged machine is equal to the *inf* of states. But this may not be the case since the merged machine is a Moore machine and the number of states in the merged machine may be larger than the *inf* of states. A correction factor c (a negative number) has to be added to the above equations to account for the extra states. This correction factor is needed for the following reasons. (1) A single incoming arc entering a flip-flip corresponding to an extra state does not need an OR gate. (2) In the worst case this flip-flop may correspond to an end state of a machine. End states have feedback and exit arcs in our test control model and therefore need an AND gate each (outgoing arc). Additionally, the exit arc requires an OR gate at *Tail.* To offset the cost of the extra flip-flops, in our synthesis procedure we attempt to merge the outgoing arcs from these flip-flops using a multi-level minimization approach. Merging the outgoing arcs often leads to reduced implementation cost for these arcs and this is incorporated in the correction factor. In the best case, the exit arc can be merged with the feedback arc. Case (1) contributes a 1 to the correction factor while Case (2) contributes a 3 to the correction factor. Therefore $c = -4$ and the lower bounds for the implementation cost when the number of states of the merged machine is greater than the *inf* of states are given by the (following) Equations 8.10 and 8.11 for the cases $S = n$ and $S < n$, respectively.

$$C_{lb} = 2n + 3 + 2p + c \tag{8.10}$$

$$C_{lb} = min\{n, \sum_{i=2}^{S+p} k_i + 2(S + p - 1) - r'\} + (S + p) + 3 + c \tag{8.11}$$

8.6.3 Lower Bound of Output Logic

The state transition table of the machines M^i, i=1,2,...,n, shows the values of the output lines c_j corresponding to the present state for every machine. Consider the part of the transition table corresponding to only one machine M^i. Let c_j, j=1,2,...,p, be the set of output lines corresponding to this machine. c_j corresponds to a column vector with q_i entries. If there is one or less 1 entries (the rest may be 0's or x's) or one or less 0 entries (the rest may be 1's or x's) in the column vector corresponding to c_j, then c_j comes directly from a flip-flop or a fixed logic level and needs no output logic. However if there are more than one 1 entries *and* more than one 0 entries in any column, then the corresponding control line has to be driven by either an OR gate or a NOR gate. Let a be the number of 1's and b be the number of 0's in c_j, where $a, b > 1$. If $a > b$ then a b input NOR gate is needed, and if $b > a$, then an a input OR gate is needed. If the cost of a 2 input OR/NOR gate is 1, then the cost of gates in the two cases above is $b - 1$ or $a - 1$. When more than one c_j has $a, b > 1$, then sharing of gates between them may be necessary to get a minimal implementation.

As stated earlier, we have restricted our output logic to one level of OR/NOR gate implementation. This restriction is realistic because all the examples that we have looked at have trivial output logic (for the 1-hot code implementation) and do not warrant using general 2-level or multiple level logic minimization algorithms.

Let OL be a q x p matrix where the columns correspond to the p control lines and rows to the q states. All columns where $a \leq 1$ or $b \leq 1$ are deleted. OL is now reduced to a q x p' matrix. A pair chart technique is used to find compatible columns. Each of the compatibles has a cost. We solve a covering problem to find a set of compatibles that covers all columns and has minimum cost.

Proposition 8.2 *Let L_{ol_i} be the minimum cost output logic for 1-hot coded submachines M^i, i=1,2,...,n. Then the greatest lower bound on the output logic for M^m is L_{ol_m}=max $\{L_{ol_i} \mid i{=}1,2,...,n\}$.*

Proof : Suppose a control line c_i, is driven by only one machine. Then another machine is merged with this machine and c_i is now driven by a merged machine. The process of merging states between two state machines has either no effect on the the original number of 1s and 0s in a column vector representation of c_i, or adds 1s and/or 0s. The cost of implementing the control line c_i thus can never be less than when there was no merger. This is also true when sharing of logic between outputs is considered. In a merged machine, all submachines must retain their functionalities. Thus the lower bound of output logic in the merged machine is $max\{L_{ol_i} \mid i = 1, 2, ..., n\}$.

It is conceivable that the merger of machines does not affect the 1 and/or 0 count of the outputs of machine M^i, where $L_{ol_i} = \max\{L_{ol_j} \mid j = 1, 2, ..., n\}$. L_{ol_i} is a valid output logic complexity of M^m. Thus it is the greatest lower bound. □

8.7 Synthesis Procedure

We have developed an implicit enumeration procedure SOHOT (*S*ynthesis of *O*ptimal 1-*HOT* controllers) that incrementally builds the transition graph of a state minimal merged machine with minimal next state and output logic. The basic data structure of SOHOT is a n-ary tree called *Search_tree*. Each node N_i in *Search_tree* has some properties attached to it. One of these properties is a directed graph DG. DG=(V,E) represents the state transition graph (partial or complete) of a M^m. We will present the basic concepts of SOHOT by illustrating how an optimal M^m is built for the example circuit given in Fig. 8.4. Pseudocode for this procedure can be found in [64].

The *Search_tree* along with the DGs for every node in this tree are shown in Figs. 8.17 and 8.18. *Node_list* is a list of nodes in *Search_tree*. Initially *Node_list*=$\{N_1\}$.

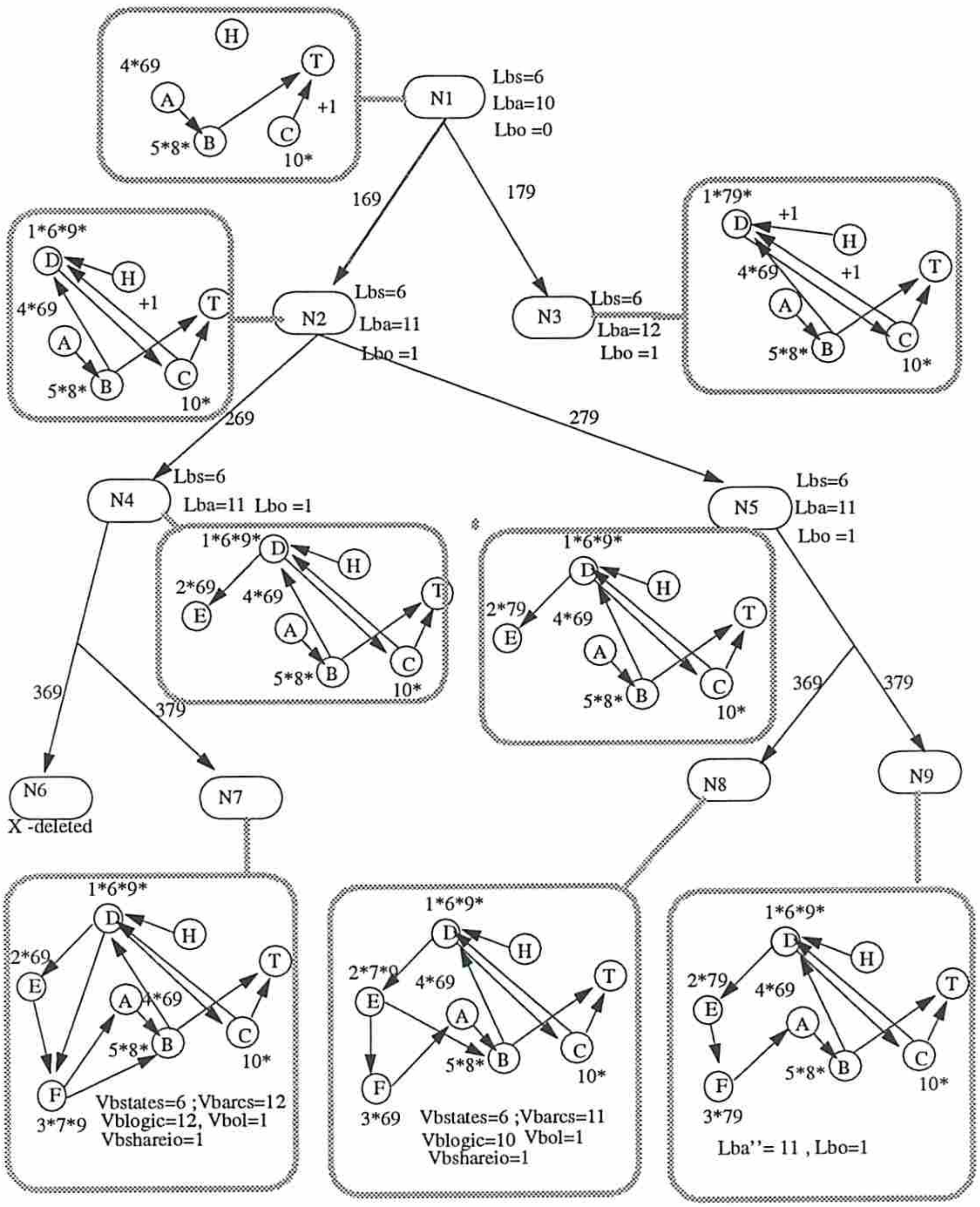


Figure 8.17: The search tree for Ckt 1 (continued in next figure)

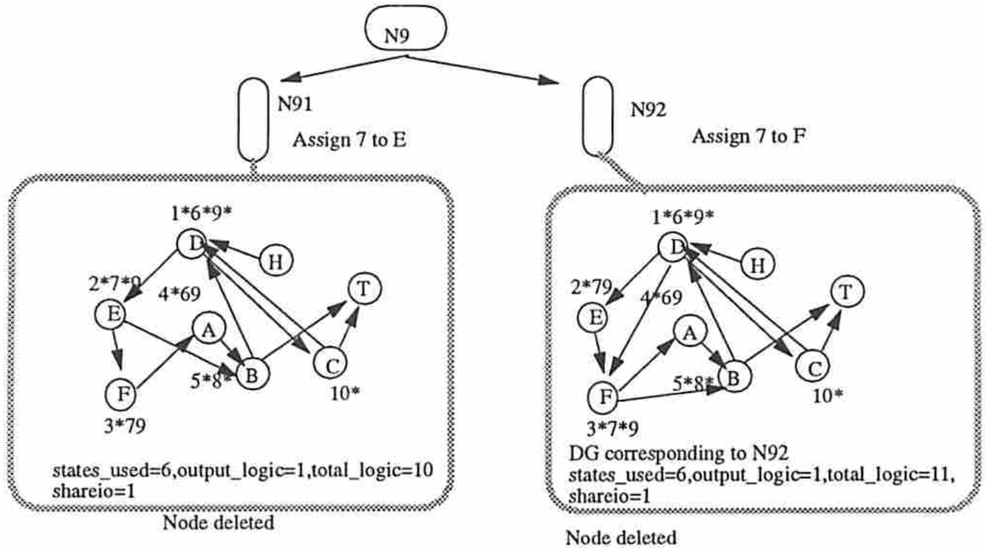


Figure 8.18: The search tree for Ckt 1

Freq_list is a list of states in M^i,i=1,2,3, sorted in nondecreasing order of their occurrences in the PCs shown in Fig. 8.9. States 4,5,8,10 have a frequency of 1. Thus the three PCs (4,6,9),(5,8) and (10) which cover these states *must* be used to build M^m and are selected first. These PCs are attached to the three new vertices, A,B and C created in the DG of N_1. Since the states 4,5,8,10 will not appear again, we *assign* them to the vertices A,B and C and mark this assignment by *starring* them. States 6 and 9 are not starred because they occur in other PCs. It is not known at this point if assigning 6 and 9 to vertex A will yield an optimal M^m. However, states 4,5,6,8,9 and 10 are deleted from *Freq_list.* Since a transition exists between states 4 and 5 in M^1, an edge is added between A and B. Similarly edges (B,T) and (C,T) are added to the DG of N_1 because transitions (5,Tail),(8,Tail) and (10,Tail) exist in all three submachines.

The information from the DG of N_1 is used to compute metrics that establish a lower bound on the number of states (*lbs*), number of arcs (*lba*) and output logic (*lbo*) of the M^m whose partial transition graph is DG. Theoretically, the *inf* of arcs is 9 (*Theorem 8.1*). To achieve this result it is necessary to merge all the end states 5,8,10 (*Corollary 8.1*). However, in the DG of N_1, states (5,8) and 10 are assigned to two different vertices and there are two edges instead of one going to T. Thus the new lower bound, *lba*, is 10 (9+1). From *Proposition 8.1*, the *inf* of the number of states is 5. The DG of N_1 has 3 vertices if H and T are excluded. However, 3 states from M^1 and 1 state from M^2 have not been accounted for. Assignment of the 3 states of M^1 will require *at least* 3 additional vertices in DG. Hence *lbs* is 6. An output matrix OL is created having three rows corresponding to the three vertices A,B and C in DG. The columns correspond to the control lines activated by the states 4,5,8 and 10 assigned to these vertices (4 to A, 5 and 8 to B, 10 to C). No gate is required to implement the output logic. So *lbo* is 0. These metrics are properties of N_1. *Freq_list* and OL are also properties of N_1.

State 1, the first element of the modified *Freq_list* with a frequency of 2, needs to be assigned to a vertex. Since PCs (1,6,9) and (1,7,9) both cover state 1, N_1 is expanded to create children N_2 and N_3. These nodes inherit all the properties of N_1. Vertex D is added to the DGs of both nodes. PCs (1,6,9) and (1,7,9) are attached to the vertex D in the DGs of N_2 and N_3 respectively. State 1 is starred for both nodes. The edges of the DG of N_2 is updated next. Since 1 is a start state in M^1, an edge (H,D) is added. Since merging the start states 1,6,9 is a necessary condition for achieving the *inf* of arcs in M^m, and since 6 and 9 appear in vertex D along with state 1, we assign 6 and 9 to D even though they appear in other PCs (and vertices). So 6 and 9 are starred. Edges (D,C),(B,D) and (C,D) are then added. The metrics *lba,lbs* and *lbo* need to be computed for this node. This DG has edges (B,D) and (C,D) corresponding to two feedback arcs instead of one required to achieve the *inf* of arcs. So *lba* is incremented by 1 and becomes 11. *lbs* of this node is 6, since 2 states in M^1 have yet to be accounted for. An additional row is added to the OL of N_2 to reflect vertex D and the entries in this row are the control lines activated by the states 1,6,9. Control line c_{10} requires a two input OR(NOR) gate, so *lbo*=1. For N_3, state 9 is starred and edges (H,D), (B,D),(C,D) and (D,C) are added to its DG. Similar to the DG of N_2, the DG of N_3 has two feedback arcs. In addition only two out of three start states are assigned to D, thus another arc will be needed to cover the (Head,6) transition in M^2. Thus *lba* is incremented by 2 and becomes 12. *lbs* is 6 and *lbo is 1.*

The node N_1 is deleted from *Node_list* and N_2 and N_3 are added. N_2 is the next candidate for expansion since *lba* of N_2 is less than the *lba* of N_3. Note that *lba* can be used directly as a measure of the predicted next state logic cost since for both nodes *lbs* is greater than the *inf* of states (Equation 8.10), else the actual lower bound needs computation using Equations 8.8 and 8.10. The *Freq_list* of N_2 only has states 2,3 and 7 since the other states have been deleted. State 2 has a frequency of 2, hence N_2 is expanded to create two children N_4 and N_5. PCs (2,6,9) and (2,7,9) are attached to vertex E in the DGs of nodes N_4 and N_5. State 2 is starred in both cases. Edges (D,E) are added to both. The metrics for both the nodes are 6,11 and 1 for *lbs*, *lba* and *lbo* respectively.

Node_list=$\{N_4, N_5, N_3\}$. Expansion of N_4 creates two nodes N_6 and N_7 corresponding to the two choices of PCs for state 3. Node N_6 is deleted because its *Freq_list* still contains state 7 whereas the *Freq_list* of N_7 is empty. Now all states are found to be starred in the DG of N_7 implying that all states have been assigned to vertices and this DG represents a complete transition graph of a M^m.

The next state logic, output logic, the number of states and the number of arcs used by the M^m corresponding to the DG of N_7 is determined. Sharing of logic (if any) between next state and output logic is taken into account and assigned to a local variable *shareio.* The shared logic, *shareio*, is subtracted from the sum of the logic needed to implement the next state and output logic and assigned to another local variable *Total_logic.* Global variables *Vbstates*, *Vbarcs*, *Vblogic*, *Vbol* and *Vbshareio* are assigned values for the states, arcs, total logic, output logic and shared logic respectively for the best DG found at any point in SOHOT. Since the DG of N_7 is the first complete transition graph found so far, *Vbstates*=6, *Vbarcs=12*, *Vblogic*=12, *Vbol*=1, *Vbshareio*=1 and *Best_Node*=N_7.

Node_list=$\{N_5, N_3\}$. The metrics of N_5 are compared against the best found so far and N_5 is expanded to produce N_8 and N_9 corresponding to the two PC choices for state 3. The DG of N_8 is updated and it is observed that it is also a complete transition graph of a M^m. This DG has less arcs and logic than the best found so far. So *Best_node*=N_8, *Vbstates*=6, *Vbarcs=11*, *Vblogic*=10, *Vbol*=1 and *Vbshareio*=1. After the DG of N_9 is updated, it is found that state 7 has not been starred even though it appears in vertices E and F. This implies that we have to evaluate the impact of the assignment of state 7 to each of these vertices. This makes the search space larger than the space of all minimal PC covers. *lba* is no longer a good metric for nodes such as N_9 which have unassigned states. Another metric *lba″* is calculated instead. This is a heuristic and is a lower bound on the number of arcs. It is more optimistic (gives smaller numbers) than *lba* but helps to order nodes in *Node_list* if more than one node similar to N_9 exists. Assume that E and F are lumped together to form a new vertex EF. Then the transition (6,7) is covered by edge (D,EF) but no edge exists from EF to B to cover (7,8). The DG has 10 edges and *at least* 1 additional edge will be required. So *lba″* is 11. *lbs* is 6 and *lbo* is 1.

Node_list=$\{N_9, N_3\}$, N_9 is inserted before N_3 because *lbs* for both the nodes are equal and *lba″* of N_9 is less than the *lba* of N_3. Expansion of N_9 is different from the expansions encountered so far. Here two nodes N_{91} and N_{92} are created corresponding to the two vertices E and F to which state 7 can be assigned. Node N_{91} is deleted because *total_logic* equals *Vblogic* (we do not keep track of more than one solution with the best cost) and node N_{92} is deleted because *total_logic* is greater than *Vblogic.*

Node_list=$\{N_3\}$. Lets assume that the best implementation of a M^m built up from the DG of N_3 has 12 arcs and shares 1 unit of logic between input and output. Using Equation 8.10 the lower bound on implementation cost is (2*3+3+2*3-4), which equals 11. However, the best M^m found so far has *Vblogic*=10, *Vbol*=1 and *Vbshareio*=1. Thus expanding N_3 will never give a better result. So N_3 is deleted and the hardware realization of the best M^m, corresponding to N_8, is shown in Fig. 8.19.

The procedure will terminate if at any leaf node it finds a DG whose next state and output logic equals the *inf*s of the next state and output logic equals the minimal logic cost given in Equation 8.7. This is a very powerful feature of SOHOT because it does not need to explore the solution space any further.

8.8 Experimental Results

Some results using SOHOT are presented in Fig. 8.20. The entry *Total states* in the table corresponds to $\sum_{i=1}^{n} q_i$, where n is the number of test plans for a particular circuit. The *inf* of states, arcs, next state logic (*nsl*) and output logic (*ol*) are also tabulated. It is important to note that the search space is not simply the space of all minimal PC covers. A state may occur in more than one PC in the minimal cover and an optimal assignment of this state to a PC increases the search space. For example, Ckt 1 has 6 PC covers, but the entire search space has 18 leaf nodes. Entries *states*, *arcs*, *lb nsl*, *nsl*, *ol* and *share* on the right side of the table represent the total number of states, arcs, lower bound on next state logic, the actual next

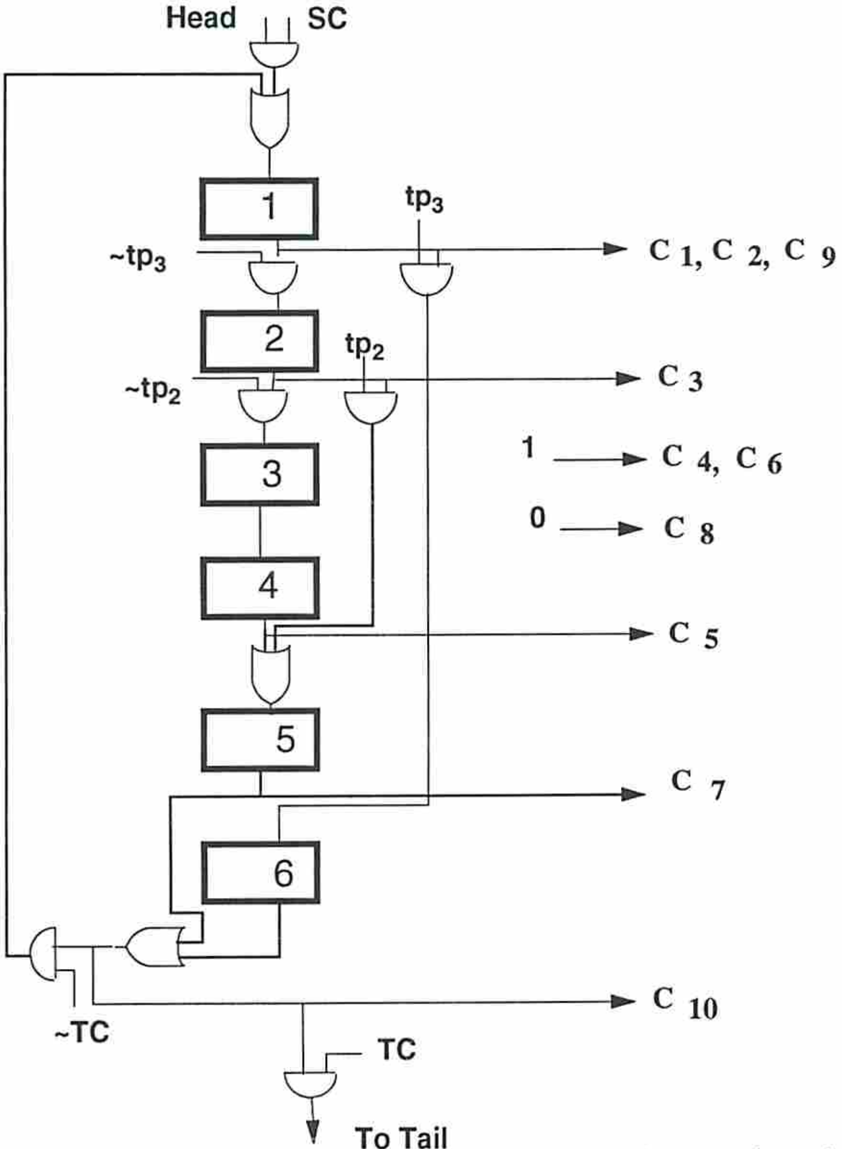


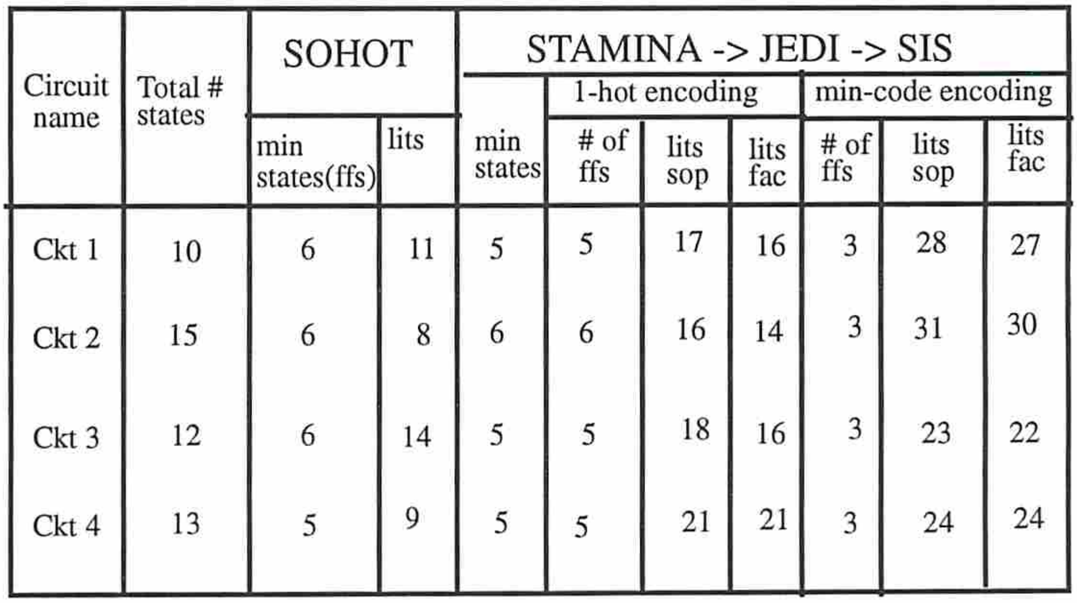
Figure 8.19: Hardware implementation of the minimal merged machine for Ckt 1

state logic, output logic and the logic shared between next state and output logic, respectively. As seen from the table, M^m for Ckt 1 has 11 arcs, 6 states, and 10 2-input gates, and the procedure only examined 4 leaf nodes out of a possible 18. Ckt 2 (3 machines with 6, 5 and 4 states) is interesting because the next state logic and output logic of M^m at one of the leaf nodes is equal to the *inf* of the next state logic and output logic, respectively, and the procedure terminated without searching any further. In this case, only 2 out of a possible 18 nodes were examined. For Ckt 3 (3 machines with 5, 4 and 3 states), the procedure examined only 2 leaf nodes out of 10. Ckt 4 (3 machines with 5, 4 and 4 states) has only 1 answer. Except for an OR gate driving a control line in Ckt 1 and Ckt 3, all other control lines of the circuits are driven directly from the flip-flops of their merged machines. Note that the lower bound (*lb nsl*) computation provides very close estimates of the final implementation cost (*nsl*). These circuits are detailed in [64].

							Optimal Merged Machine							
Ckt Name	Total states	Inf states	Inf arcs	Inf nsl	Inf ol	Total Leaf nodes	states	arcs	lb nsl	nsl	ol	share	Total logic	Leaf nodes gen.
Ckt 1	10	5	9	9	0	18	6	11	9	10	1	1	10	4
Ckt 2	15	6	10	9	0	18	6	10	9	9	0	0	9	2
Ckt 3	12	5	9	9	0	10	6	12	11	13	1	1	13	2
Ckt 4	13	5	8	7	0	1	5	9	9	10	0	0	10	1

Figure 8.20: Results for example circuits

We have run our examples through a set of standard synthesis tools. For state minimization we used STAMINA [28]. JEDI [59] is used for state assignment, and SIS [50], has been used for logic minimization. STAMINA, however, produces a minimal state Mealy machine. Since we use the Moore model, in some cases the cardinality of the minimum state cover produced by STAMINA is smaller than that produced by SOHOT. However, this is the only state minimization tool that is integrated with SIS. For each of the example circuits we carried out state assignment using both the 1-hot code and the minimum number of flip-flops. Then logic minimization is performed using a standard script file. Fig. 8.21 shows the results. The descriptions given to the synthesis tools did not have the *Head* and *Tail* states. Therefore when we obtained the literal counts for the designs implemented by SOHOT, we did not count the literals contributed by the transitions to/from the Head and Tail states. From Fig. 8.21, we can see that the literal counts for designs obtained using SOHOT are much smaller than those obtained by standard synthesis tools. The entries *lits sop* and *lits fac* represent the number of literals in the sum of product form and factored form respectively.



Circuit name	Total # states	SOHOT		STAMINA -> JEDI -> SIS						
					1-hot encoding			min-code encoding		
		min states(ffs)	lits	min states	# of ffs	lits sop	lits fac	# of ffs	lits sop	lits fac
Ckt 1	10	6	11	5	5	17	16	3	28	27
Ckt 2	15	6	8	6	6	16	14	3	31	30
Ckt 3	12	6	14	5	5	18	16	3	23	22
Ckt 4	13	5	9	5	5	21	21	3	24	24

Figure 8.21: Comparison with standard synthesis tools

8.9 Interfacing with IEEE 1149.1 Boundary Scan Architecture

In Section 8.2 we presented a control model that assumes that the TAP controller is not present on-chip and there is no test bus. In this section we will show how the 1-hot coded controller can be integrated with the IEEE 1149.1 boundary scan architecture and also be compatible with the bus-based architecture proposed in Chapter 4. The basic objective of proposing the control model in Section 8.2 is to provide the designer with the option of choosing an alternate test control strategy when compatibility with the IEEE 1149.1 boundary scan architecture is not a requirement.

8.9.1 Direct Control

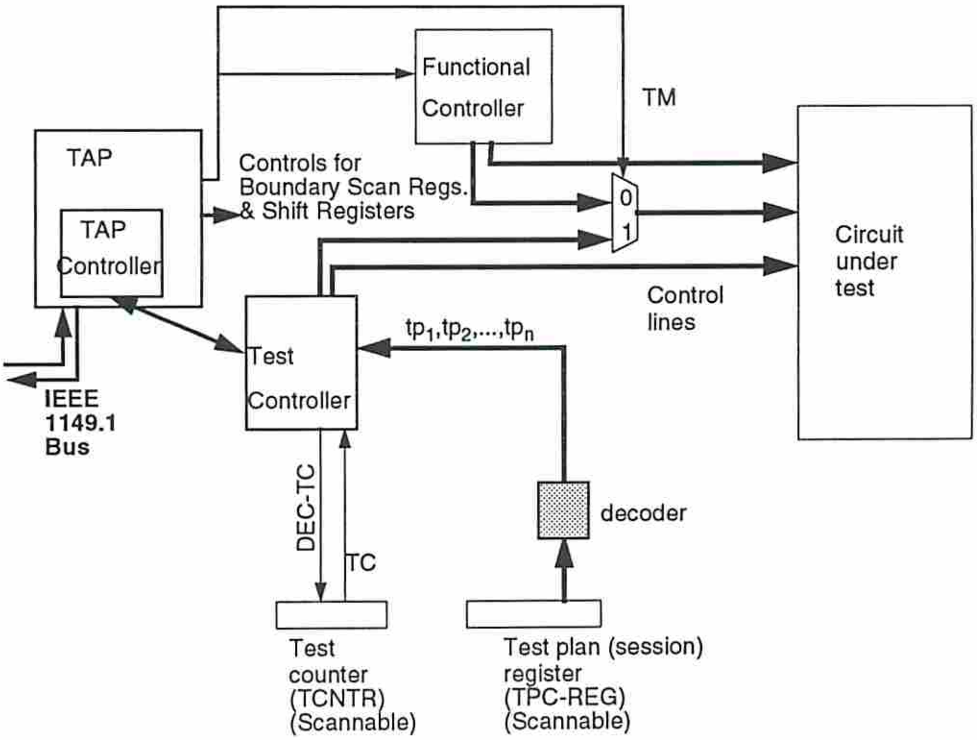


Figure 8.22: The control model for direct control

This subsection presents the control model and the overall model of the test controller for incorporating the TAP controller on the chip and controlling the test resources by routing control lines from the TAP controller and the 1-hot coded controller (test controller). Fig. 8.22 is a model of this architecture. This model shows the TAP controller interacting with the test controller. The TAP controller controls the shift/capture/update functions of the boundary scan registers as well as the shift/load operations of the scannable functional registers. Control transfers to the test controller when the TAP controller is in the *run-test/idle* state and the *RUNBIST* instruction is loaded in the IR. The exact mechanism is presented in Fig. 8.23. Since the TAP controller controls the processes of scan-in/scan-out of the data/results, the shift counter (SCNTR - in Fig. 8.1) is not needed. Moreover the test plan counter (TPCNTR) is made scannable and is initialized with a binary string corresponding to the current test plan (session) under execution.

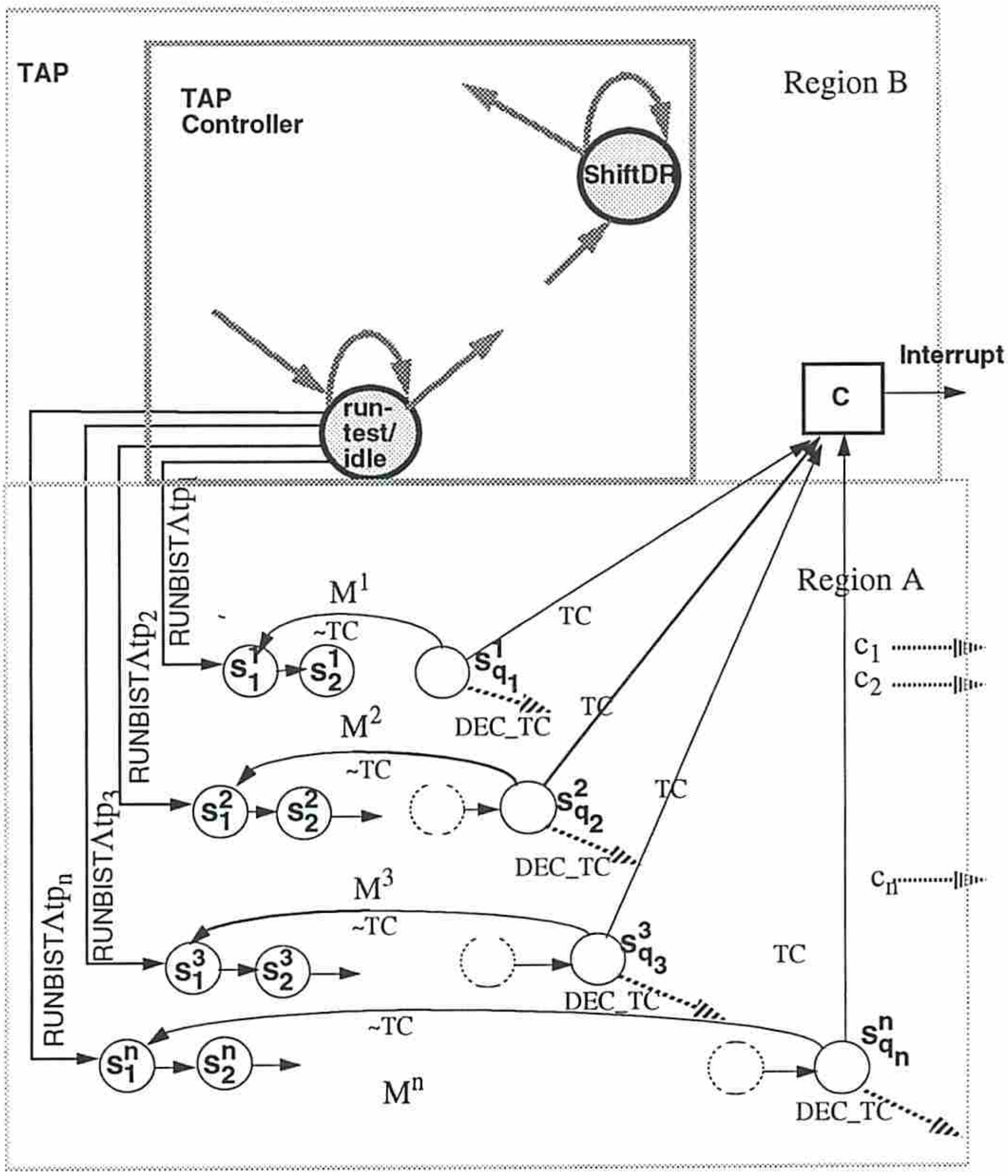


Figure 8.23: The overall test controller in direct control

The state diagram of the overall test controller is shown in Fig. 8.23. In this figure Region B corresponds to the TAP. The *run-test/idle* state of the TAP controller serves as the *Head* for the test controller. Individual machines in the test controller are activated when the TAP controller is in the *run-test/idle* state and the *RUNBIST* instruction is loaded in the IR. The *Tail* is replaced by logic block *C* which generates an interrupt to the test channel signifying end of test application for a particular test plan (session). The control lines activated by the test controller correspond to control lines for various data transport paths and the mode control (PG/SA) for the test registers. An example truth table for a test register that has PG or SA capability is given in Fig 8.3. If a register has both PG and SA capabilities then an extra control line can be added.

8.9.2 Bus-based Control

The 1-hot coded test controller can also be used in the bus-based approach. The control model of the bus-based approach is given in Fig. 8.24. This model shows the TAP controller (the ITAPC) driving an internal test bus. This bus is an input to both the functional and the test controllers as well as the datapath. The test counter and the test plan counter also have inputs from the bus since they are both scannable. The multiplexer selecting between the outputs of the functional and the test controller is controlled by a decoder that decodes off the test bus. The exact nature of the interaction between the test controller and the test bus is shown in Fig. 8.25. A decoder in the test controller decodes the code corresponding to the *run-test/idle* state of the ITAPC and the *RUNBIST (RBIST)* instruction. This decoder activates one of the constituent machines in the test controller depending on the test plan to be executed.

In the bus-based model the test controller controls the elements in the data transport paths and if necessary the PG/hold or SA/hold signals for the registers. The models for the registers is the same as that presented in Chapter 3 and the bus schemes as well as the encoding techniques are the same as that presented in Chapter 4.

8.10 Summary

Contemporary state machine synthesis techniques perform state minimization first and then do state, input and output encoding. However, in this chapter we have shown that for some specific problems, such as the test controller synthesis problem, it is possible to use certain properties specific to the nature of the controllers and come up with an implicit enumeration technique to achieve a optimal synthesized

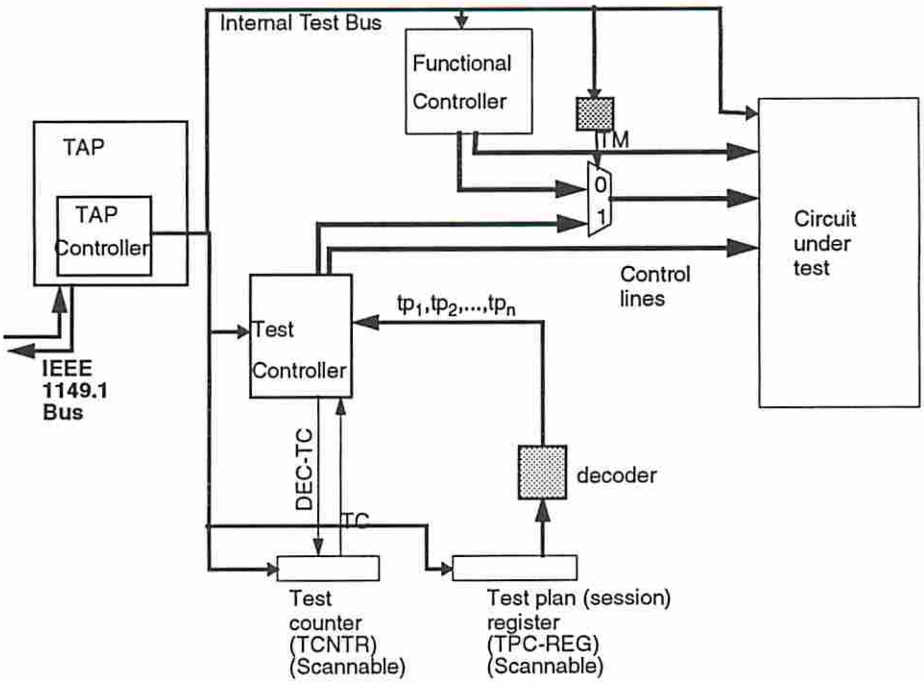


Figure 8.24: The control model for bus-based control

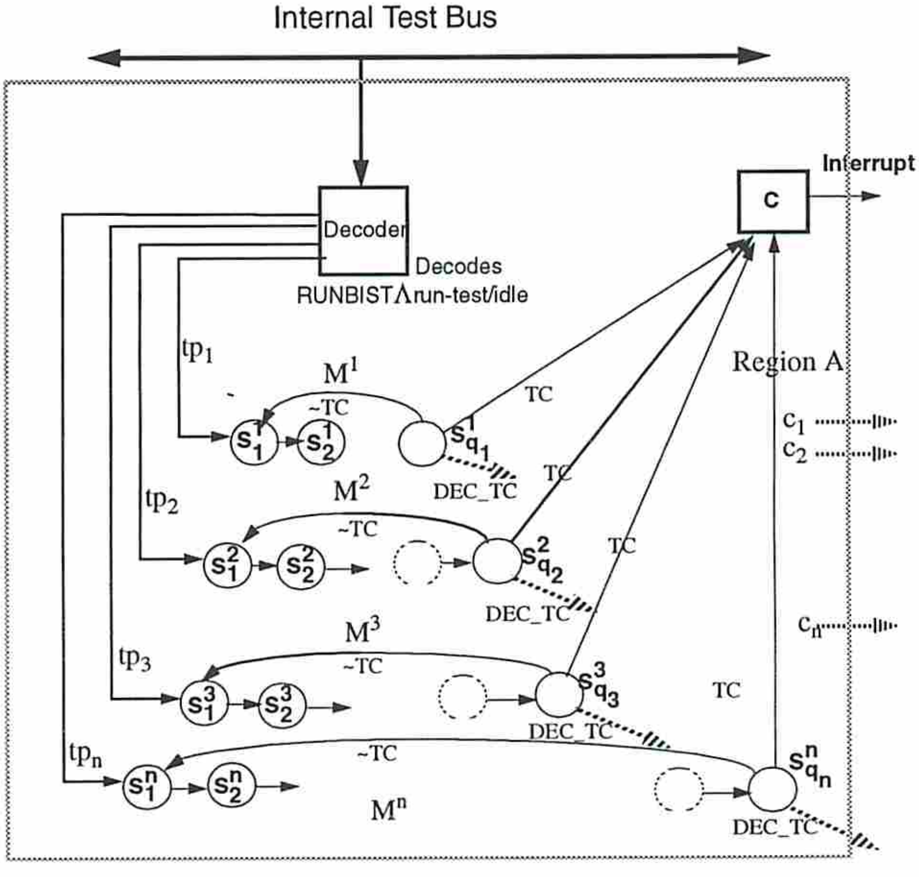


Figure 8.25: The overall test controller for bus-based control

machine. Some of the subproblems by themselves are NP-complete or NP-hard. For example, state minimization is a NP-hard problem. Determining the minimum output logic, even when it is restricted to one level, is an NP-complete problem. Since the problems in the test controller domain are of restricted size, in this chapter we have assumed that we will solve these subproblems exactly. We could have an option of using heuristic techniques for obtaining the PCs or use existing logic minimizers to obtain the output logic. However, using the 1-hot code makes the output logic trivial and consequently the output logic problem size is small enough to make exact techniques feasible. Also note that the size of a PC in our problem is upper bounded by the number of machines, n, to be merged, and the complexity of finding all the PCs is $\mathbf{O}((\sum_{i=1}^{n} q_i)^{2^n})$. The complexity is exponential in the number of machines being merged and not in the number of states.

We mentioned in Section 8.2.3 that one of the main advantages of using a 1-hot coded controller is the flexibility of distributing the flip-flops in the controller throughout the design such that control wire routing overhead is reduced. The flip-flops of the merged controllers for the four circuits given in the result can indeed be distributed throughout the circuits. Also, the single OR gate in the output logic of Ckt 1 and Ckt 3 is fully tested by the walking 1 pattern that is generated in the controller during the test mode.

We show that the 1-hot coded controller can be used in conjunction with a wide range of architectures. The architecture presented in Section 8.2 assumes that a chip does not support the boundary scan architecture. In Section 8.9.1 we showed how the 1-hot coded controller can be integrated with the boundary scan architecture using the direct control scheme. Finally in Section 8.9.2 we presented a scheme where the bus-based control scheme is used in conjunction with a 1-hot coded controller.

Chapter 9

System Description

9.1 Introduction

This chapter focuses on the implementational details of CONSYST (**Con**troller **Syn**thesis **S**ystem for **T**est). Section 9.2 provides an overview of CONSYST by describing its major subsystems. In Section 9.3 the grammar for a language that specifies various attributes of a testable circuit is presented along with example testable scan and BIST designs. Flowcharts of CONSYST are provided in Section 9.5. Examples of circuits synthesized by CONSYST are shown in Section 9.6. Section 9.7 illustrates some graphical menus developed for the system.

9.2 System Overview

A circuit to be made testable is processed by SIESTA or BITS. These systems generate a family of testable designs with different attributes in terms of test time, area overhead of the test hardware and fault coverage. Various characteristics of a testable design such as scan chains and test plans are specified in a high level test description language developed for CONSYST. Using this description, CONSYST synthesizes the test control logic and provides control area overhead information for each design. A selection system such as SAESS [65] can be used to help the designer make a choice between different designs. Once a decision is made to implement a particular testable design, CONSYST automatically incorporates all test control hardware as well as test resource hardware (such as pattern generators and signature compactors) into the circuit.

Circuits are represented in the Cbase (an object-oriented database developed by the USC test group [66]) format. Circuits in Cbase are hierarchical in nature and are either captured schematically by a designer using the Cbase graphical user interface (GUI), imported from other schematic capture tools using hierarchical EDIF, translated from a structural VHDL description or generated by a high-level synthesis system such as ADAM [67]. CONSYST is written in C and C++ (approximately 30,000 lines of code) and incorporates a GUI for interacting with the designer. Currently CONSYST is compiled under SunOs Release 4.1.3.

9.2.1 Major Subsystems

An overview of CONSYST is shown in Fig. 9.1. The major subsystems that constitute CONSYST or help in its functioning are briefly explained below.

- **OCTTOOLS :** This is a set of logic synthesis and layout tools developed mainly at the University of California, Berkeley [68]. CONSYST uses some of the sequential and combinational logic synthesis tools in OCTTOOLS. Specifically, STAMINA [28] is used for state minimization and NOVA [43] and JEDI [59] are used for state assignment targeted for two-level and multi-level logic implementation, respectively. Espresso [48] and SIS [50] are used for two-level and multi-level logic minimization, respectively. TimberWolf [51] and YACR [52] are used for standard cell placement and layouts of the control logic, respectively.

- **Merged Test Controller Synthesizer :** Given the STTs of individual test plan controllers this module obtains the STT of a merged test controller that has minimal two-level implementation cost. This module is called COMPOSER and is described in detail in Chapter 6.

- **Functional and Test Controller Merger Module :** This module optimally merges a test controller that controls a number of test plans with the on-chip functional controller. This module is called OMEN and can target either a two-level or multi-level implementation of the merged controller. It is presented in detail in Chapter 7.

- **ITAPC and Distributed Decoder Synthesizer :** This module synthesizes the Integrated TAP Controller (ITAPC) and the decoders necessary for

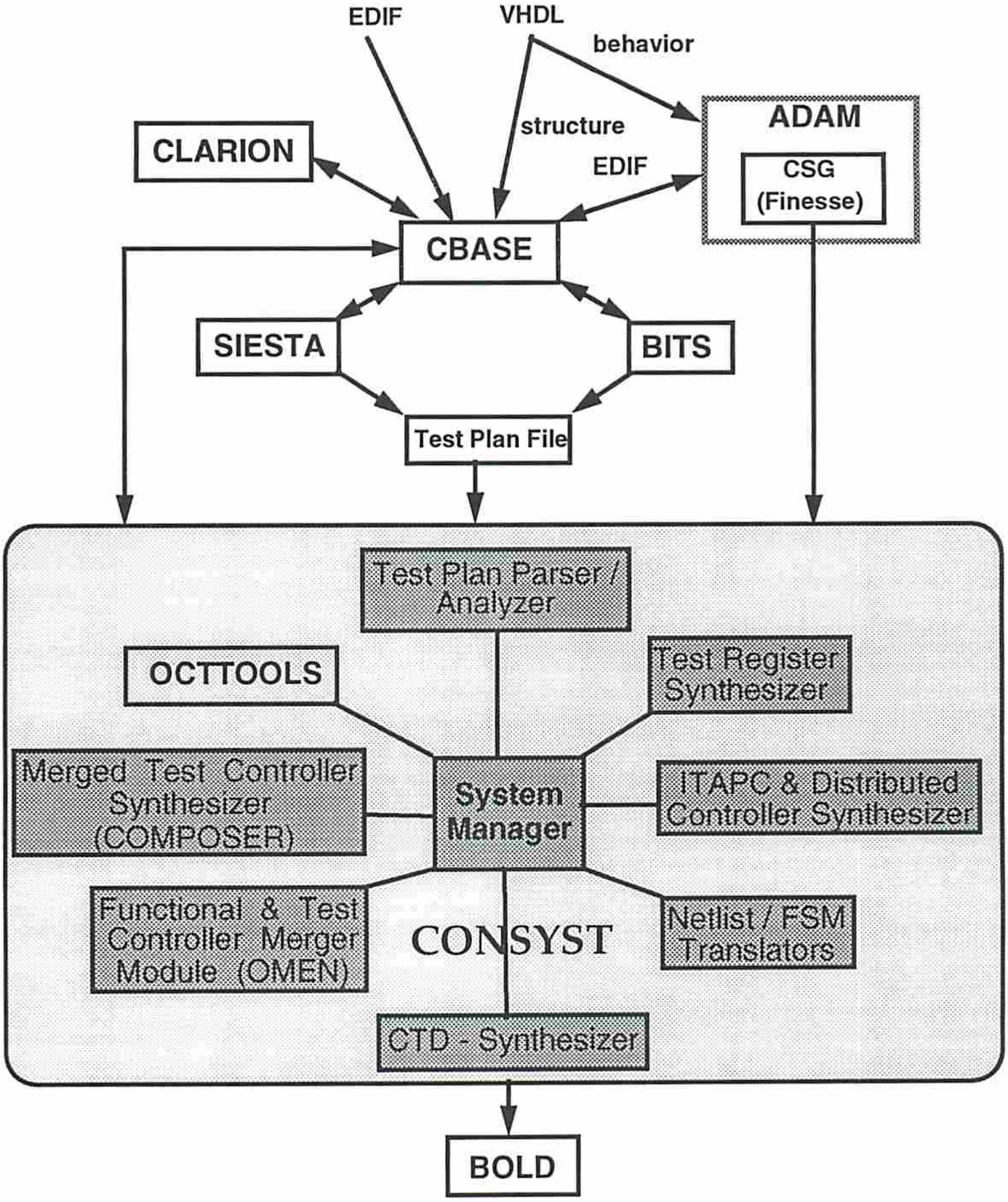

EDIF
VHDL
behavior
structure
EDIF
ADAM
CSG
(Finesse)
CLARION
CBASE
SIESTA
BITS
Test Plan File
Test Plan Parser / Analyzer
OCTTOOLS
Test Register Synthesizer
Merged Test Controller Synthesizer (COMPOSER)
System Manager
ITAPC & Distributed Controller Synthesizer
Functional & Test Controller Merger Module (OMEN)
CONSYST
Netlist / FSM Translators
CTD - Synthesizer
BOLD

Figure 9.1: System Overview - Subsystem Modules

controlling various scan and BIST registers. This module can minimize bus symbols as well as encode them to minimize the implementation cost of the distributed decoders and/or the ITAPC. More details about the techniques used by this module can be found in Chapter 4.

- **Test Register Synthesizer :** This module creates hierarchical gate level descriptions of the registers that are involved in testing a circuit. This includes implementing the components of the local controllers. Gate level netlists of decoders generated by the preceding module are incorporated in the local controllers. Furthermore data selection multiplexers, mode and configuration flip-flops and ex-or gates for PG/SA polynomials are also implemented.

- **CTD Synthesizer :** The BOLD [11] system developed at USC defines the architecture of hierarchical off-chip test and maintenance controllers. BOLD also has compilers that convert high level chip (module) test logic and test application descriptions into code that is executable by the off-chip controller(s). The CTD (Chip Test Description) synthesizer module creates a description of the testable chip and the test plans using the Chip Test Language (CTL) [11]. CTL has strong similarities to BSDL [69].

- **Netlist / FSM Translators :**

  - BLIF2C**base Translator :** This module takes a circuit described in BLIF (*Berkeley Logic Interchange Format*) and incorporates it into Cbase. CONSYST uses synthesis tools that produce a logic description of the test control hardware in BLIF.

  - **Finesse2**STT **Translator :** In ADAM, a program called CSG [70] creates a behavioral description of the functional controller which is provided as input to a FSM synthesis program (FINESSE) in the CONCORDE design tools. The Finesse2STT translator is used to convert a functional controller specification generated by CSG to the STT format recognized by CONSYST.

- **System Manager :** The System Manager invokes other modules and controls the flow of execution of CONSYST.

9.3 Test Plan Description Language

Systems such as SIESTA or BITS are *test analysis* tools, in that they determine attributes necessary to make a chip testable. However CONSYST performs the actual task of implementing the test control and resource hardware. Therefore a consistent mechanism is needed to transfer information from the test analysis process to CONSYST. Since test analysis can even be performed by a test engineer, it is important to describe the test features of a chip in a *consistent* and *unambiguous* manner. This is accomplished by the Test Plan Description Language (TPDL). The attributes of a testable chip such as scan chains, characteristics of test registers and the test plans are described in a Test Plan File (TPF) using TPDL. A grammar for this language has been specified and is presented in the following subsection.

A compiler is needed to translate the information in a TPF into a format recognizable by CONSYST. There are two major parts of any compiler : (1) a lexical analyzer, and (2) a parser. The lexical analyzer reads the source file and passes *tokens* or *terminal symbols* to the parser which checks if the string of tokens are valid according to rules of the grammar for the language in which the source file is written. Yacc (yet another compiler-compiler) [71] is a parser generator that simplifies the process of implementing a parser for a language. The context-free grammar for TPDL is written in the input format of Yacc and then Yacc is run to create a parser for TPDL in C. In the grammar presented in the following subsection, the words in capital letters are the terminal symbols while the words in lower case letters are the non-terminal symbols. In the rest of this chapter we use the term *keyword* to denote terminal symbols. Lex [72] is a tool used for simplifying the process of writing a lexical analyzer for a grammar. A specification of a lexical analyzer for the TPDL grammar is written in the Lex language, and then Lex is run on this specification to create a lexical analyzer for TPDL in C. Refer to [73] for more details of context-free grammars, lexical analysis and parsing.

9.3.1 Grammar for TPDL

```
test_procedure : _BEGIN_CHAIN_DEF  chains _END_CHAIN_DEF
                 _BEGIN_REG_GROUP_DEF reg_groups _END_REG_GROUP_DEF
                 _BEGIN_CONTROL_LINE_DEF control_lines _END_CONTROL_LINE_DEF
                 _BEGIN_FUNC_REG_DEF func_regs  _END_FUNC_REG_DEF
                 _BEGIN_SESSION_DEF  sessions _END_SESSION_DEF
                ;
chains          : chain
                | chains chain
                ;
chain           : _CHAIN _INT_NUM _SEMICOLON
                  _CHAIN_TYPE _EQ chain_type _SEMICOLON all_reg
                ;
chain_type      : _BOUNDARY_SCAN
                | _GENERAL
                ;
all_reg         : _BEGIN_CHAIN reg_defs _END_CHAIN
                ;
reg_defs        : reg_def
                | reg_defs reg_def
                ;
reg_def         : _REG_NAME _EQ _STRING _COMMA _REG_ID _EQ _INT_NUM _COMMA
                  _LENGTH _EQ _INT_NUM _SEMICOLON
                | _REG_NAME _EQ _STRING _COMMA _REG_ID _EQ _INT_NUM _COMMA
                  _LENGTH _EQ _INT_NUM _COMMA _TEST_ONLY _SEMICOLON
                | _REG_NAME _EQ _STRING _COMMA _REG_ID _EQ _INT_NUM _COMMA
                  _LENGTH _EQ _INT_NUM _COMMA _HAS_FUNC_HOLD _SEMICOLON
                | _REG_NAME _EQ _STRING _COMMA _REG_ID _EQ _INT_NUM _COMMA
                  _LENGTH _EQ _INT_NUM _COMMA bs_io _SEMICOLON
                ;
reg_groups      : reg_group
                | reg_groups reg_group
                ;
reg_group       : _REG_NAME _EQ _STRING _COMMA _CONSTITUENTS _EQ regs _COMMA
                  _FUNC _EQ functions _SEMICOLON
                | _NULL
                ;
regs            : reg
                | regs reg
                ;
reg             : _STRING
                ;
bs_io           : _INPUTBS
                | _OUTPUTBS
                ;
functions       : function
                | functions function
                ;
function        : _PG _L_BRAC polydescriptor _R_BRAC
                | _SA _L_BRAC polydescriptor _R_BRAC
                | _LOAD
                | _HOLD
                ;
polydescriptor : _STRING _COLON _POLY _EQ _STRING
                | _STRING _COLON _POLY _EQ _STRING feed_forwards
                ;
feed_forwards   : feed_forward
                | feed_forwards feed_forward
                ;
feed_forward    : _FEEDF _EQ _STRING
```

```
                ;
control_lines   : control_line
                | control_lines control_line;
                ;
control_line    : _LINE_NAME _EQ _STRING _COMMA _LINE_ID _EQ _INT_NUM _COMMA
                  _LINE_WIDTH _EQ _INT_NUM _SEMICOLON
                | _NULL
                ;
func_regs       : func_reg
                | func_regs func_reg
                ;
func_reg        : _REG_NAME _EQ _STRING _COMMA _REG_ID _EQ _INT_NUM _SEMICOLON
                | _NULL
                ;
sessions        : session
                | sessions session
                ;
session         : _SESSION  _INT_NUM _SEMICOLON session_def
                ;
session_def     : _TDM _EQ _STRING _SEMICOLON session_body
                ;
session_body    : _BEGIN_INITIALIZE initializations _END_INITIALIZE
                  _BEGIN_APPLICATION application   _END_APPLICATION
                ;
initializations: initialization
                | initializations initialization
                ;
initialization : _CHAIN _EQ _INT_NUM _COMMA _INT_VEC _EQ _STRING _COMMA
                  _RES_VEC _EQ _STRING _COMMA _SHIFT_CYCLES _EQ _INT_NUM
                  _SEMICOLON
                ;
application     : phases _APPLY_CYCLES _EQ _INT_NUM _SEMICOLON
                ;
phases          : phase
                | phases phase
                ;
phase           : _PHASE _EQ _INT_NUM _COMMA _CONTROL_LINE_VAL _EQ vals _COMMA
                  _REG_VAL _EQ reg_vals _SEMICOLON
                | _NULL
                ;
vals            : val
                | vals val
                ;
val             : _STRING _COLON _INT_NUM
                ;
reg_vals        : reg_val
                | reg_vals reg_val
                ;
reg_val         : _STRING _COLON session_func
                ;
session_func    : _PG _L_BRAC _STRING _R_BRAC
                | _SA _L_BRAC _STRING _R_BRAC
                | _LOAD
                | _HOLD
                ;
```

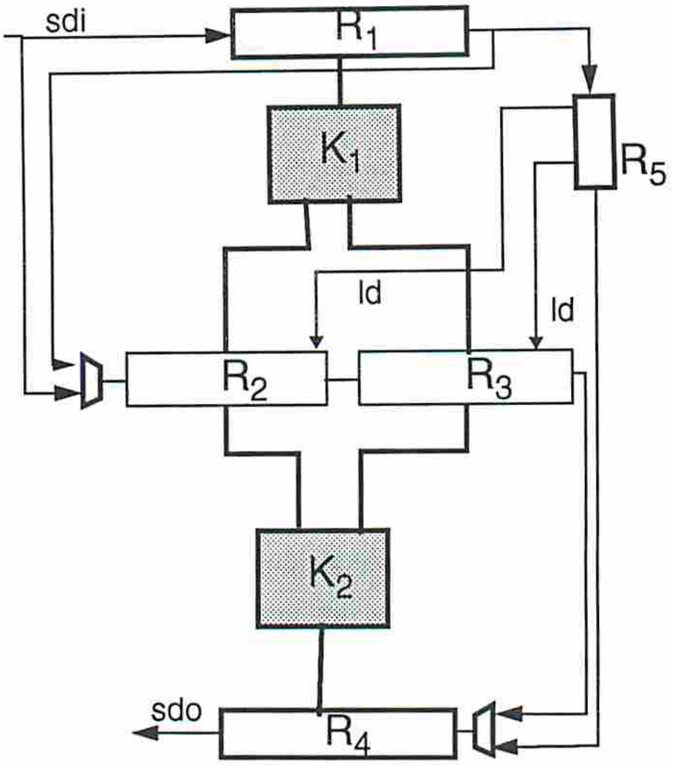


Figure 9.2: Scan_Ckt

```
      END_CHAIN
 CHAIN 2;
      CHAIN_TYPE = GENERAL;
      BEGIN_CHAIN
                 REG_NAME = R2, REG_ID = 97,  LENGTH = 2;
                 REG_NAME = R3, REG_ID = 112, LENGTH = 2;
                 REG_NAME = R4, REG_ID = 135, LENGTH = 4;
      END_CHAIN
END_CHAIN_DEF
BEGIN_REG_GROUP_DEF     NULL   END_REG_GROUP_DEF
BEGIN_CONTROL_LINE_DEF NULL   END_CONTROL_LINE_DEF
BEGIN_FUNC_REG_DEF      NULL   END_FUNC_REG_DEF
BEGIN_SESSION_DEF
  SESSION 0;   /* Test Kernels K1 and K2 */
    TDM = F_SCAN;
       BEGIN_INITIALIZE
             CHAIN = 1, INT_VEC = vector_file0, RES_VEC = result_vector_file0,
             SHIFT_CYCLES = 8;
        END_INITIALIZE
        BEGIN_APPLICATION
              NULL
              APPLY_CYCLES = 1;
        END_APPLICATION
  SESSION 1;  /* Test Kernel K2 */
     TDM = F_SCAN;
        BEGIN_INITIALIZE
```

```
                CHAIN = 2, INT_VEC = vector_file1, RES_VEC = result_vector_file1,
                SHIFT_CYCLES = 4;
          END_INITIALIZE
           BEGIN_APPLICATION
                NULL
                APPLY_CYCLES = 1;
           END_APPLICATION
END_SESSION_DEF
```

9.3.2.2 Discussion

The scan chains are defined first in a TPF. The scan chains are specified either as BOUNDARY_SCAN or GENERAL. In each scan chain the constituent registers are specified in the order of their appearance in the chain. The name (a string that follows keyword REG_NAME), Cbase identification number (an integer that follows keyword REG_ID), number of bits (an integer that follows keyword LENGTH) and a possible extra attribute are specified for every register. This extra attribute can be one of the following.

- INPUTBS - Specifies that the register is an input boundary scan register.
- OUTPUTBS - Specifies that the register is an output boundary scan register.
- TEST_ONLY - Specifies that the register is not part of the functional design and is added to the design for test purposes. The corresponding REG_ID can be any integer. Such registers are needed by PET [74] for pseudo-exhaustive testing.
- HAS_FUNC_HOLD - Specifies that the register has a functional load/hold control line.

Boundary scan registers for bidirectional or tristate input/output pins are not supported. Note that all the cells in a register are assumed to be homogeneous, i.e., all cells have the same functions.

Since this is a full scan design no control signals need to be specified and therefore the NULL keyword is inserted between the keywords BEGIN_CONTROL_LINE_DEF and END_CONTROL_LINE_DEF. For unbalanced partial scan designs, all control lines to non-scan registers need to be defined so that the boundary scan cell driving the corresponding control line can be modified as described in Chapter 5.

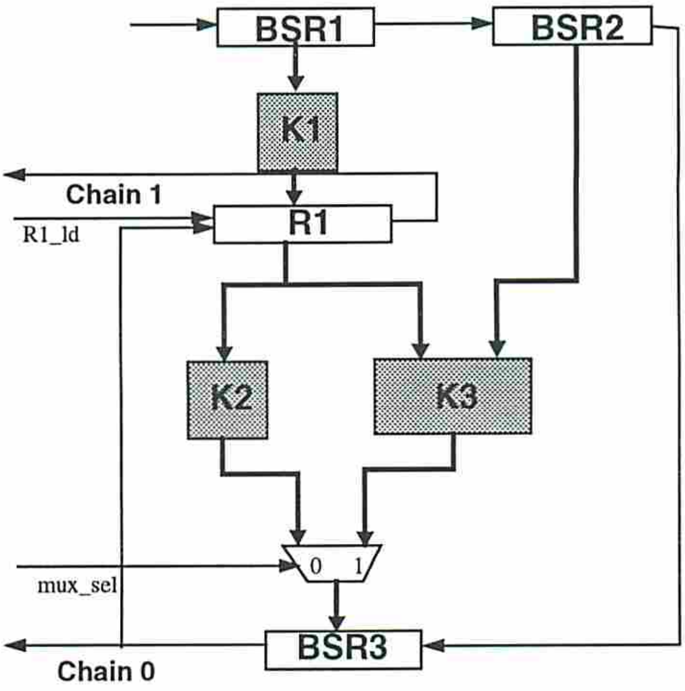


Figure 9.3: Bist_Ckt_1

9.3.3.1 Test Plan File for Bist_Ckt_1

```
BEGIN_CHAIN_DEF
  CHAIN 0;
    CHAIN_TYPE = BOUNDARY_SCAN;
    BEGIN_CHAIN
      REG_NAME = BSR1, REG_ID = 57,  LENGTH = 4, INPUTBS;
      REG_NAME = BSR2, REG_ID = 80,  LENGTH = 4, INPUTBS;
      REG_NAME = BSR3, REG_ID = 115, LENGTH = 4, OUTPUTBS;
    END_CHAIN
  CHAIN 1;
    CHAIN_TYPE = GENERAL;
    BEGIN_CHAIN
      REG_NAME = BSR1, REG_ID = 57,  LENGTH = 4, INPUTBS;
      REG_NAME = BSR2, REG_ID = 80,  LENGTH = 4, INPUTBS;
      REG_NAME = BSR3, REG_ID = 115, LENGTH = 4, OUTPUTBS;
      REG_NAME = R1,   REG_ID = 150, LENGTH = 4, HAS_FUNC_HOLD;
    END_CHAIN
END_CHAIN_DEF
BEGIN_REG_GROUP_DEF
  REG_NAME = BSR1, CONSTITUENTS = BSR1,    FUNC = PG (PG1: POLY = f1<3+4);
  REG_NAME = R1,   CONSTITUENTS = R1,      FUNC = PG (PG1: POLY = f1<3+4)
                                                  SA (SA1: POLY = f1<3+4);
  REG_NAME = RG1,  CONSTITUENTS = R1 BSR2, FUNC = PG (PG1: POLY = f1<2+3+7+8);
  REG_NAME = BSR3, CONSTITUENTS = BSR3,    FUNC = SA (SA1: POLY = f1<3+4);
END_REG_GROUP_DEF
BEGIN_CONTROL_LINE_DEF
  LINE_NAME = R1_ld,   LINE_ID = 713, LINE_WIDTH = 1;
  LINE_NAME = mux_sel, LINE_ID = 725, LINE_WIDTH = 1;
END_CONTROL_LINE_DEF
BEGIN_FUNC_REG_DEF  NULL  END_FUNC_REG_DEF
BEGIN_SESSION_DEF
    SESSION 0;      /* Test K1 */
       TDM = BILBO;
           BEGIN_INITIALIZE
              CHAIN = 1, INT_VEC = bbbbxxxxxxxxbbbb,RES_VEC = xxxxxxxxxxxxbbbb,
              SHIFT_CYCLES = 16;
           END_INITIALIZE
           BEGIN_APPLICATION
              PHASE = 0,
                  CONTROL_LINE_VAL = mux_sel:2,
                  REG_VAL = BSR1:PG(PG1) R1:SA(SA1);
              APPLY_CYCLES = 16;
           END_APPLICATION
    SESSION 1;     /* Test K2 */
       TDM = BILBO;
           BEGIN_INITIALIZE
              CHAIN = 1, INT_VEC = xxxxxxxxbbbbbbbb,RES_VEC = xxxxxxxxbbbbxxxx,
              SHIFT_CYCLES = 16;
           END_INITIALIZE
           BEGIN_APPLICATION
              PHASE = 0,
                  CONTROL_LINE_VAL = mux_sel:0,
                  REG_VAL = R1:PG(PG1) BSR3:SA (SA1);
              APPLY_CYCLES = 16;
           END_APPLICATION
    SESSION 2;     /* Test K3 */
       TDM = BILBO;
            BEGIN_INITIALIZE
               CHAIN = 1, INT_VEC = xxxxbbbbbbbbbbbb,RES_VEC = xxxxxxxxbbbbxxxx,
               SHIFT_CYCLES = 16;
            END_INITIALIZE
            BEGIN_APPLICATION
                PHASE = 0,
                   CONTROL_LINE_VAL = mux_sel:1,
                   REG_VAL = RG1:PG(PG1) BSR3:SA (SA1);
                APPLY_CYCLES = 256;
            END_APPLICATION
END_SESSION_DEF
```

9.3.3.2 Discussion

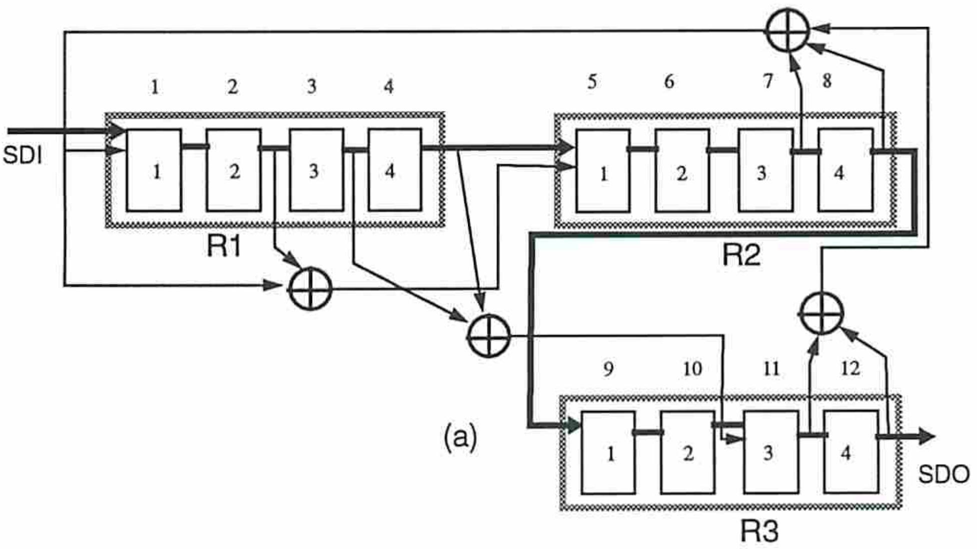
Register grouping information is necessary for BIST designs because different logical registers may be formed by concatenating physical registers in different test sessions (refer to Chapter 3). Each logical register is defined within the keywords BEGIN_REG_GROUP_DEF and END_REG_GROUP_DEF. The following information is required.

- A string specifying the name of a logical register (follows keyword REG_NAME). If a register is never combined with another register then this name may be the same as the register itself (as specified in the shift chain information), else a unique name should be assigned. For example the logical register comprised of R1 and BSR2 is assigned a unique name RG1.

- A set of strings specifying the names of registers that constitute a logical register (follow keyword CONSTITUENTS).

- A set of function (mode) specifications. Each function specification is of the form < *keyword*1 > (*polydescriptor*) or simply < *keyword*2 >, where *keyword*1 is one of PG, SA and *keyword*2 is one of LOAD,HOLD. *polydescriptor* completely defines a polynomial used for pattern generation or signature analysis and is written as follows.

  - A string specifying the name (a reference) of a particular polynomial.
  - A string that specifies the feedback taps (follows the keyword POLY). For example the string f1<3+4, specifies that the third and fourth bits of a logical register are ex-ored and fed to the first bit of the register.
  - Often PGs are implemented as LFSR/SRs. These PGs are used by the PET subsystem and sometimes by the BIBS TDM in BITS. These PGs often require *feedforwards* in addition to the feedback inputs. Feedforwards are described by a string that follows the keyword FEEDF. For example the string f5<2+3 states that the fifth bit of a logical register is fed by the ex-or of bits 2 and 3. A string such as f5<0+2 specifies that the feedback input into the first bit of the logical register is ex-ored with bit 2 and then fed to bit 5. An example of a logical register implemented with three functional registers is shown in Fig. 9.4(a). This logical register is a pattern generator and has feedforwards. The numbers inside each of the register cells specify the order within a particular register, while the numbers above a register specify the order of the cells in the logical register. The thick lines correspond to the scan chain. The definition of this logical register (named RG1) in a TPF is shown in Fig. 9.4(b).



(a)

```
BEGIN_REG_GROUP_DEF
.
.
REG_NAME = RG1,  CONSTITUENTS =  R1 R2 R3,
          FUNC = PG  (PG1 :  POLY = f1<7+8+11+12
                             FEEDF=  f5 < 0+2   FEEDF = f11< 3+4 );

.
.
END_REG_GROUP_DEF
```

(b)

Figure 9.4: (a) A logical register with feedforwards; (b) description of the register using TPDL

All functional single/multi-bit control lines (*carriers* in Cbase terminology) in a BIST design are described between the keywords BEGIN_CONTROL_LINE_DEF and END_CONTROL_DEF. Each control line (carrier) is specified by the following attributes.

- A string specifying the name of the control line (follows keyword LINE_NAME).
- The Cbase identifier (follows LINE_ID).
- An integer specifying the number of nets in the control line (follows the keyword LINE_WIDTH).

In BIST designs a functional register R without a load/hold control may need to hold for one or more clock periods to insert delays in a pipelined test application to achieve the minimal average latency of the pipeline. Since only scannable registers are referenced in the chain definition, a reference to R must be provided separately in the TPF. This is done by specifying the name and Cbase identification of R between the keywords BEGIN_FUNC_REG_DEF and END_FUNC_REG_DEF. An example of such a register is provided in Section 9.4.1.1.

The session definitions for BIST designs differ from scan designs in a few aspects. As noted earlier, for BIST designs the string following the keywords INT_VEC or RES_VEC represents a single initialization vector and is not a test vector file name. In this string, a x represents a don't care and a b represents a care bit which can be either 1 or 0. The string following the keyword TDM for BIST designs is either BILBO, EBILBO (extended BILBO) or BBILBO (balanced BILBO). The TDM type is only used to determine if a circuit is a scan design or a BIST design and all BILBO based TDMs can simply be specified as BILBO.

Multiple phases can be specified in the session description. A phase is specified as follows.

- Control lines that need to be set to specific values are represented by $< line\_name >:< value >$, where *line_name* references a control line defined earlier. *value* is 0, 1 or 2 (don't care). A set of such declarations follow the keyword CONTROL_LINE_VAL.
- The mode of logical registers are represented by $< register\_name >:< function\_name >$ or

$< register\_name >:< function\_name >(< polynomial\_name >)$. For example in Phase 0 of Session 0 the register modes are defined by $BSR1 : PG(PG1)$ and $R1 : SA(SA1)$. A register that simply holds is represented by $< register\_name >: HOLD$.

- The number of clock cycles for a session is specified by an integer that follows the keyword APPLY_CYCLES.

9.4 Processing Control Lines

The information for controlling each test session is provided by explicit control line and value declarations as well as by specifying the mode of logical registers in each phase. The following summarizes the control line processing actions performed by CONSYST for BIST designs.

- First determine if an internal test controller is needed. Check the number of phases for each session - if any session has more than one phase then an internal controller is needed.

- The second step is to determine if *PG/hold* or *SA/Hold* control lines need to be added to PGs or SAs. This is done by checking if a logical register is specified to be in the PG (SA) mode in one phase of a session and in the hold mode in another phase of the same session. If this register already has a load/hold control line then an extra control line is not needed. Note that if the logical register is comprised of multiple physical registers then a *PG/hold* or *SA/Hold* control line is needed for *every* constituent register.

- Create a table (*initial control table*) with the columns representing all control lines including the load/hold control lines (if any). Load/hold lines include both *PG/hold*(*SA/hold*) and load/hold lines for functional registers described within the keywords BEGIN_FUNC_REG_DEF and END_FUNC_REG_DEF. The number of rows in this table is $\sum_{i=1}^{n} p_i$, where n is the number of test sessions and p_i denotes the number of phases in session i.

- Determine control lines that can be driven directly from a boundary scan register (BR_C). This is done by scanning individual columns of the initial control table. If a control line has values 1 (0) in phase i and 0 (1) in phase j of any multi-phase session k, then this control line cannot be controlled from BR_C and must be handled by the internal test controller.

- Create a *reduced control table* by deleting columns in the initial control table that correspond to control lines controlled directly from BR_C.
- Create the STTs of n FSMs from the reduced control table. These FSMs are provided as input to COMPOSER.

9.4.1 Example BIST Circuit - Bist_Ckt_2

Consider an example BIST circuit (Bist_Ckt_2) tested using the BIBS TDM. This circuit is shown in Fig. 9.5. This circuit is referred to as *Ex1* in [18]. The TPF (generated by BITS) corresponding to a minimal area solution is presented in the following subsection. The initial control table is given in Table 9.2. Note that the composite register RG6 is constituted of registers BSR1 and R9. This register also holds for every alternate clock cycle and therefore a *SA/hold* control line is needed for *both* BSR1 and R9. Moreover R1 also holds on alternate clock cycles while generating test patterns. Therefore it needs a *PG/hold* line. Register RG1 (refers to R6 in the design) is a functional register that needs to hold for a clock period while testing B1 and B2, therefore it also needs a load/hold control line.

The initial control table has 10 columns, the first 6 correspond to the control lines defined in the test plan file and the last four correspond to load/hold lines that will be added to various registers. In Table 9.2 entries S and P correspond to the session and phase numbers, respectively. The *PG/hold*, *SA/hold* or *load/hold* control line associated with register R is represented by R_H and a 1 (0) on this line makes the register generate/compact/load patterns (hold).

The reduced control table corresponding to the initial control table is given in Table 9.3. The first four control lines in Table 9.2 can be driven directly from BR_C and hence do not appear in the reduced table. The FSMs created from the reduced control table are given in Table 9.4. Note that since COMPOSER merges machines from the largest to the smallest (in terms of the number of states) the FSMs in Table 9.4 also ordered accordingly. The letter e is a keyword denoting the end of

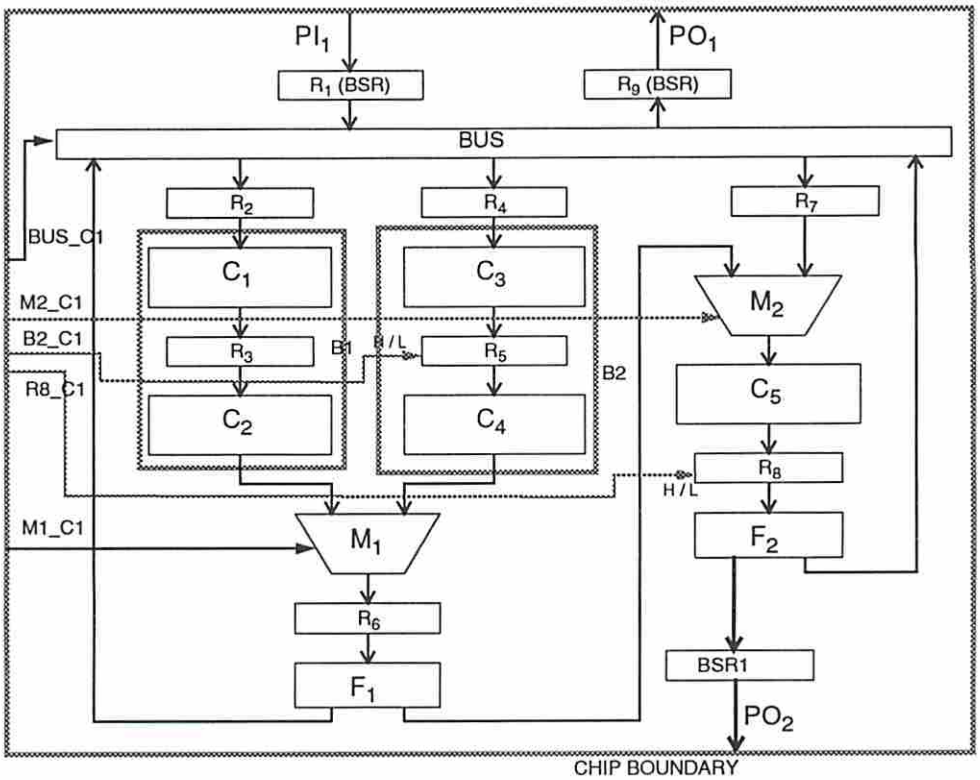


Figure 9.5: Bist_Ckt_2

a FSM description. The primary inputs are 1-hot coded. Both the inputs and the states are encoded by COMPOSER which returns a merged gate level internal test controller. For this example a 2 bit wide *session* register is used while a single flip-flop suffices for the state element. Both the session register and the state flip-flops are inserted in the scan chain (chain 1). COMPOSER returns the corresponding codes for the inputs and states and the state merger information so that the session and state registers can be properly initialized for a specific test session.

9.4.1.1 Test Plan File for Bist_Ckt_2

```
BEGIN_CHAIN_DEF
  CHAIN 0;  /* boundary scan chain */
    CHAIN_TYPE = BOUNDARY_SCAN;
    BEGIN_CHAIN
      REG_NAME = R1,   REG_ID = 42609, LENGTH = 12, INPUTBS;
      REG_NAME = R9,   REG_ID = 42827, LENGTH = 12, OUTPUTBS;
      REG_NAME = BSR1, REG_ID = 44211, LENGTH = 8, OUTPUTBS;
    END_CHAIN
  CHAIN 1;
    CHAIN_TYPE = GENERAL;
    BEGIN_CHAIN
```

```
      REG_NAME = R1,   REG_ID = 42609, LENGTH = 12;
      REG_NAME = R9,   REG_ID = 42827, LENGTH = 12;
      REG_NAME = BSR1, REG_ID = 44211, LENGTH = 8;
    END_CHAIN
END_CHAIN_DEF
BEGIN_REG_GROUP_DEF
  REG_NAME = RG3, CONSTITUENTS = R1, FUNC = PG (PG1: POLY = f1<12+7+4+3);
  REG_NAME = RG6, CONSTITUENTS = BSR1 R9, FUNC = SA (SA1: POLY = f1<20+3);
END_REG_GROUP_DEF
BEGIN_CONTROL_LINE_DEF
  LINE_NAME = M1_C1,  LINE_ID = 9293,  LINE_WIDTH = 1;
  LINE_NAME = M2_C1,  LINE_ID = 9315,  LINE_WIDTH = 1;
  LINE_NAME = B2_C1,  LINE_ID = 50197, LINE_WIDTH = 1;
  LINE_NAME = R8_C1,  LINE_ID = 44191, LINE_WIDTH = 1;
  LINE_NAME = BUS_C1, LINE_ID = 40258, LINE_WIDTH = 2;
END_CONTROL_LINE_DEF
BEGIN_FUNC_REG_DEF
  REG_NAME = RG1, REG_ID = 43874; /* RG1 refers to R6 via the Reg_id*/
END_FUNC_REG_DEF
BEGIN_SESSION_DEF
  SESSION 0;   /* Test C5 */
    TDM = BILBO;
      BEGIN_INITIALIZE
        CHAIN = 1, INT_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
                   RES_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
        SHIFT_CYCLES = 32;
      END_INITIALIZE
      BEGIN_APPLICATION
        PHASE = 0,
          CONTROL_LINE_VAL = BUS_C1:1 M2_C1:1 R8_C1:1,
          REG_VAL = RG3:PG(PG1) RG6:SA (SA1);
        APPLY_CYCLES = 256;
      END_APPLICATION
  SESSION 1;   /* Test B1 */
    TDM = BILBO;
      BEGIN_INITIALIZE
        CHAIN = 1, INT_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
                   RES_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
        SHIFT_CYCLES = 32;
      END_INITIALIZE
      BEGIN_APPLICATION
        PHASE = 0,
          CONTROL_LINE_VAL = BUS_C1:1 M1_C1:0,
          REG_VAL = RG3:PG(PG1) RG6:HOLD RG1:HOLD;
        PHASE = 1,
          CONTROL_LINE_VAL = BUS_C1:0 M1_C1:0,
          REG_VAL = RG3:HOLD RG6:SA(SA1) RG1:LOAD;
        APPLY_CYCLES = 256;
      END_APPLICATION
  SESSION 2;   /* Test B2 */
    TDM = BILBO;
      BEGIN_INITIALIZE
        CHAIN = 1, INT_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
                   RES_VEC = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb,
        SHIFT_CYCLES = 32;
      END_INITIALIZE
      BEGIN_APPLICATION
        PHASE = 0,
          CONTROL_LINE_VAL = B2_C1:1 BUS_C1:1 M1_C1:1,
          REG_VAL = RG3:PG(PG1) RG6:HOLD RG1:HOLD;
        PHASE = 1,
          CONTROL_LINE_VAL = B2_C1:1 BUS_C1:0 M1_C1:1,
          REG_VAL = RG3:HOLD RG6:SA(SA1) RG1:LOAD;
        APPLY_CYCLES = 256;
      END_APPLICATION
END_SESSION_DEF
```

9.4.2 Compacting the Reduced Control Table

The rows of the reduced control table correspond to both single and multiple phase sessions. In the initial control table, the rows corresponding to all single phase tests are incompatible (because sessions by definition are created by incompatibilities in control line values). However once some of the columns are deleted, rows corresponding to single phase sessions may become compatible and can be merged. Therefore we can either:

1. extract FSM descriptions from the reduced control table and hand the descriptions over to COMPOSER, or

2. compact the rows corresponding to single phase sessions, extract the FSM descriptions and then provide the descriptions to COMPOSER.

The advantage of approach (2) is that the number of bits in the session register may be reduced. Consider a reduced control table that represents 5 single phase sessions and 4 multi-phase sessions. The session register requires 4 bits to distinguish between the 9 sessions. However, even if two rows corresponding to the single phase sessions are merged, the number of bits in the session register is reduced to 3. The advantage of approach (1) is that since CONSYST takes a global view in merging FSMs, it will do a better job of minimizing the total number of product terms if rows corresponding to single phase sessions are not merged apriori (as in (2)). However the number of bits in the session register will be $log_2(\#of sessions)$. Currently CONSYST has both options. Rows are merged using a technique similar to that used for compacting control vectors for the Sub-compact bus (see Chapter 4).

S	P	Control Line Values									
		M1_C1	M2_C1	B2_C1	R8_C1	BUS_C1		R1_H	R9_H	BSR1_H	RG1_H
						a	b				
0	0	-	1	-	1	0	1	1	1	1	1
1	0	0	-	-	-	0	1	1	0	0	0
1	1	0	-	-	-	0	0	0	1	1	1
2	0	1	-	1	-	0	1	1	0	0	0
2	1	1	-	1	-	0	0	0	1	1	1

Table 9.2: Initial Control Table

S	P	Control Line Values				
		BUS_C1	R1_H	R9_H	BSR1_H	RG1_H
		b				
0	0	1	1	1	1	1
1	0	1	1	0	0	0
1	1	0	0	1	1	1
2	0	1	1	0	0	0
2	1	0	0	1	1	1

Table 9.3: Reduced Control Table

inputs	*p_s*	*n_s*	outputs
010	0	1	11000
010	1	0	00111
e			
001	0	1	11000
001	1	0	00111
e			
100	0	0	11111
e			

Table 9.4: Input to COMPOSER

9.5 System Flowcharts

In Section 9.2 we have described the important subsystems of CONSYST. In this section we will show how CONSYST uses these subsystems to synthesize and insert all test control and resource hardware into a testable chip.

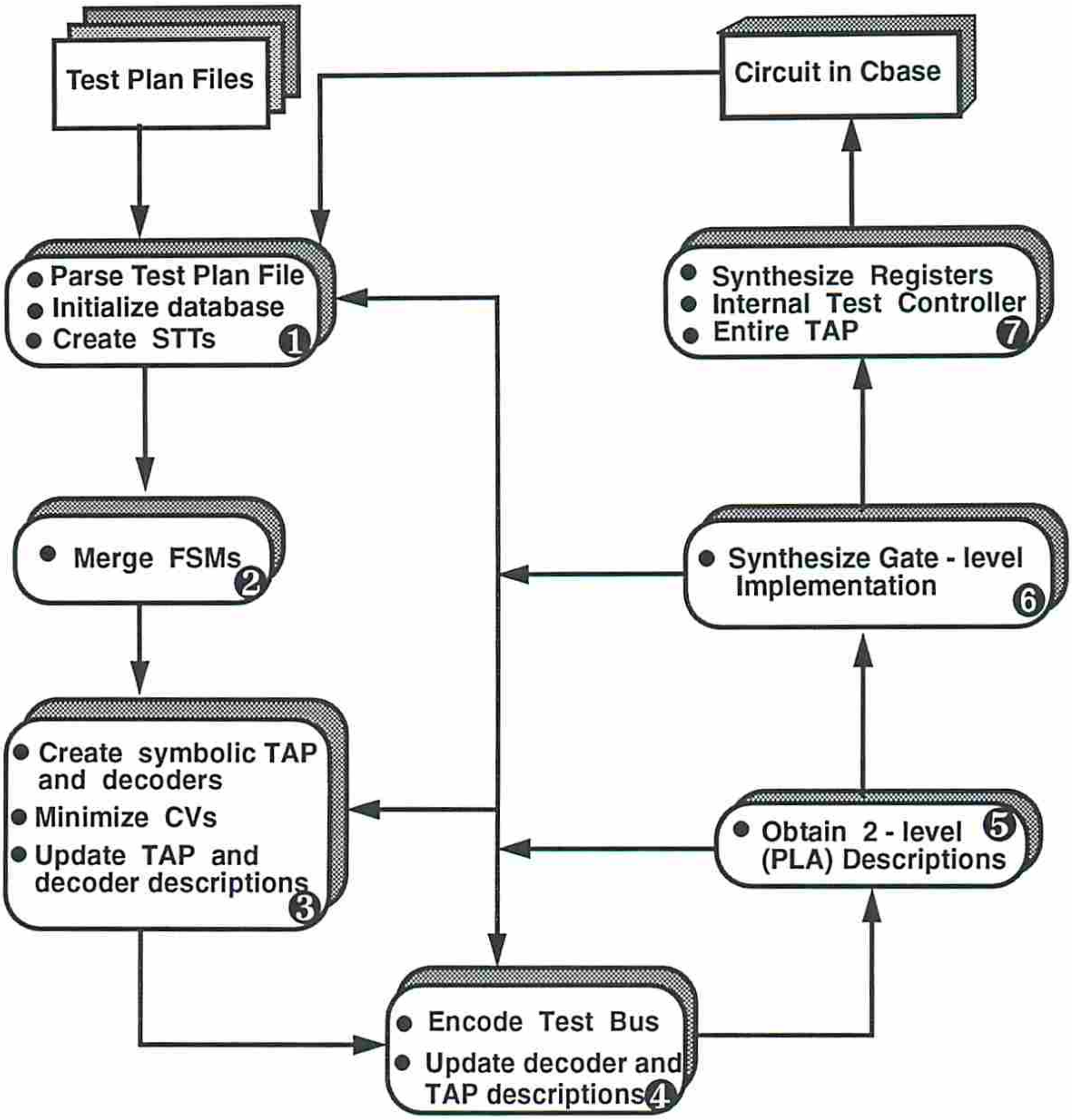

Test Plan Files
Circuit in Cbase
● Parse Test Plan File
● Initialize database
● Create STTs 1
● Synthesize Registers
● Internal Test Controller
● Entire TAP 7
● Merge FSMs 2
● Synthesize Gate - level Implementation 6
● Create symbolic TAP and decoders
● Minimize CVs
● Update TAP and decoder descriptions 3
● Obtain 2 - level (PLA) Descriptions 5
● Encode Test Bus
● Update decoder and TAP descriptions 4

Figure 9.6: Simplified flowchart of CONSYST

Fig. 9.6 is a simplified flow chart of CONSYST. The inputs to the system is a set of TPFs that correspond to various testable designs. In the block labeled **1** a TPF is parsed to allocate memory and initialize the internal database of CONSYST. The

Cbase circuit description is also checked for the presence of boundary scan registers on all chip primary inputs/outputs. Currently BITS does not insert the control boundary scan register BR_C so this register is inserted into Cbase at this stage. If the circuit is a BIST design then the phase information from the session and the control line database is used to determine if an internal test controller is needed. If an internal test controller is needed then the STTs of the various test plan controllers are created.

Block **2** is bypassed for scan designs and for BIST designs where an internal test controller is not needed. Otherwise COMPOSER is used to optimally merged the test plan controllers to synthesize a gate level description of the internal test controller. Block **3** deals with the problem of determining the number of bus symbols and creating descriptions of the decoders and the ITAPC. The user specifies the type of bus to be used. Control Vectors (CVs) are minimized and decoder descriptions are updated if either the Compact or the Sub-compact bus is chosen.

The bus symbols are encoded in block **4** using one of the encoding schemes described in Chapter 4. The encoding scheme to be used is a user defined parameter. Once the symbols are encoded, all decoder descriptions and the ITAPC description are updated. 2-level (PLA) and gate level descriptions of the decoders and the ITAPC are obtained in blocks **5** and **6**, respectively. The user has the option of viewing the costs and exploring other bus schemes or encoding strategies. If multiple TPFs are present (a family of testable designs available) then the whole process is repeated for each TPF. Once the user is satisfied with the costs then CONSYST proceeds to block **7**. In this block all the test registers are synthesized. This involves implementing gate level descriptions of the flip-flops comprising each register and the local controller. The local controller synthesis involves implementing the decoder, data selection multiplexers, the configuration and/or mode flipflops and the feedback polynomials. The components of the local controller conform to the local controller models presented in Fig. 3.9 in Chapter 3. The internal test controller (if present) is also inserted in Cbase. Moreover the session register and the multiplexer (and associated controls) used to cut the functional and test control lines are also implemented. The scan chains are connected. The entire TAP (components other than ITAPC) is synthesized, the TAP port created and relevant control lines are connected.

Fig. 9.7 is a detailed flowchart of the system. In this flowchart the oval blocks represent menu options while the rectangular blocks represent actions performed by CONSYST. The options such as IC or OC in the flowchart refer to encoding algorithms presented in Chapter 4. There is an option for selecting the number of bits to be used in encoding the bus symbols. There are three options (1) minimum, (2) as required, and (3) user. Option (2) can only be used in conjunction with input oriented algorithms. If this option is chosen, the encoding algorithm will increment the number of bits one at a time until all input constraints are satisfied. Option (3) asks the user to specify the number of bits to be used. After gate level implementations have been obtained the user has the option of graphically viewing the decoder netlists. This option uses a graphical display package linked to SIS. Once the test control and resource hardware have been synthesized, the user can view various components of the circuit such as scan chains and the internal test bus.

9.6 Examples Circuits Processed by CONSYST

In this section we present screendumps of the Cbase user interface to show the test control and resource logic incorporated in example circuits.

9.6.1 Scan Circuit - Scan_Ckt

The non-testable (original) version of the circuit is shown in Fig. 9.8. Fig. 9.9 shows a view of the top level of Scan_Ckt after processing through CONSYST using the TPF presented in Section 9.3.2.1. The internal test bus (Standard bus) is highlighted in Fig. 9.10. Scan chains 0, 1 and 2 and the registers associated with these chains are highlighted in Figs. 9.11, 9.12 and 9.13. Note that since the highlighting option in Cbase highlights entire carriers the figures may initially appear confusing.

9.6.1.1 Hierarchical View of a Register in Scan_Ckt

A view of the lower levels of the hierarchy of register R_2 is shown in Fig 9.14. In this figure, the labels in bold face have been manually added. Fig 9.14(a) shows the local controller (*l_controller*) and the register flip-flops (*reg_2bit*). Fig 9.14(b) provides a lower level view of the local controller. This controller has a decoder and

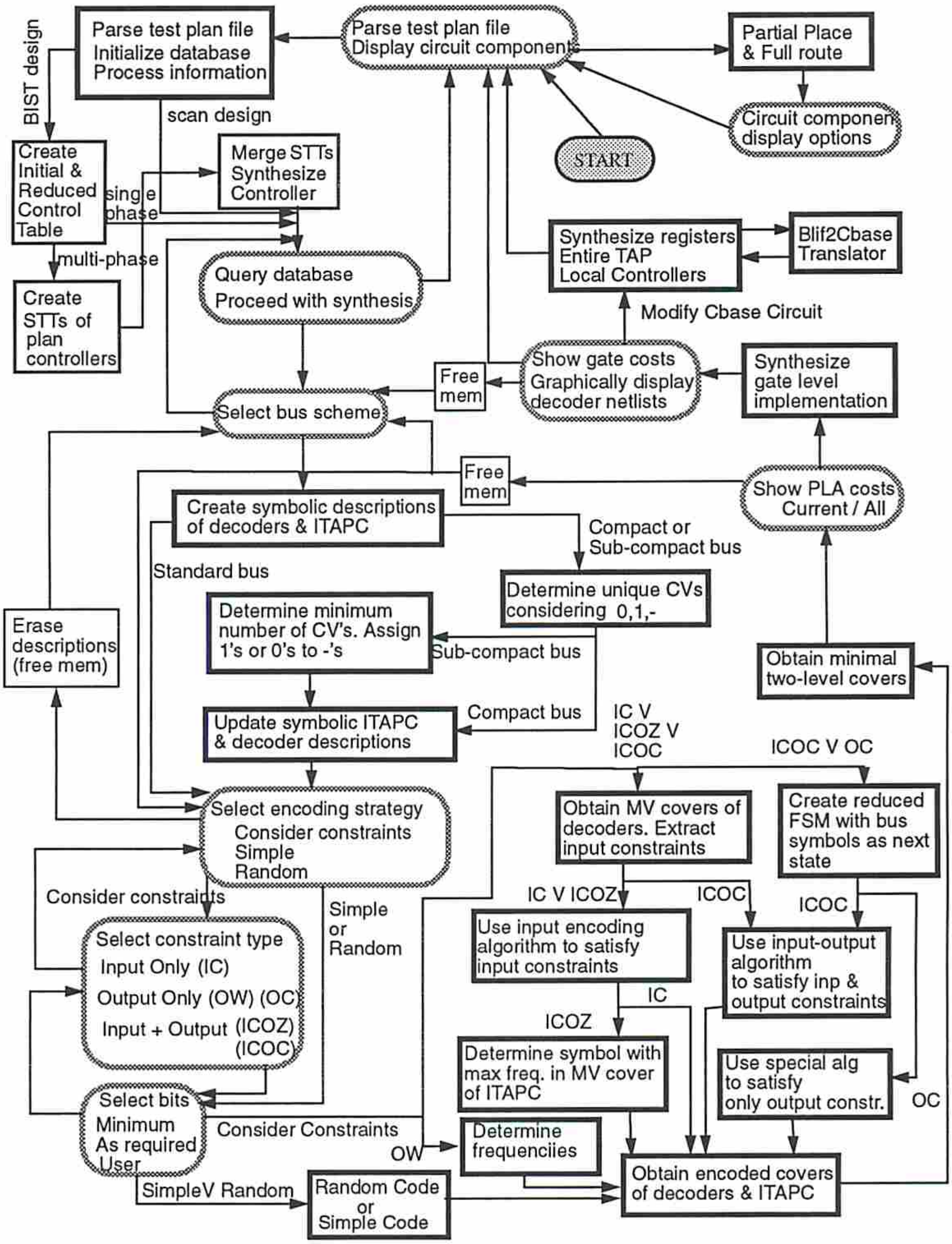
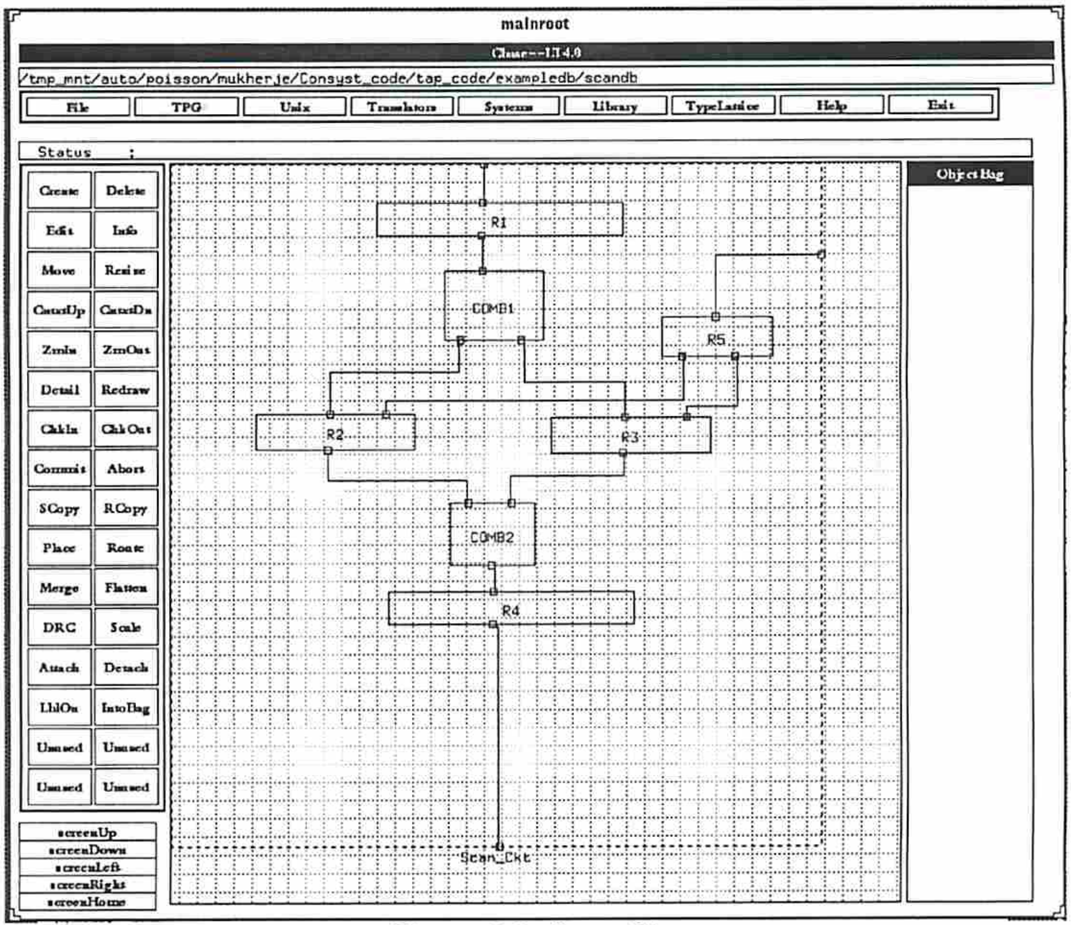


Figure 9.7: Detailed flowchart of CONSYST



/tmp_mnt/auto/poisson/mukherje/Consyst_code/tap_code/exampledb/scandb

Status :

Figure 9.8: Scan_Ckt

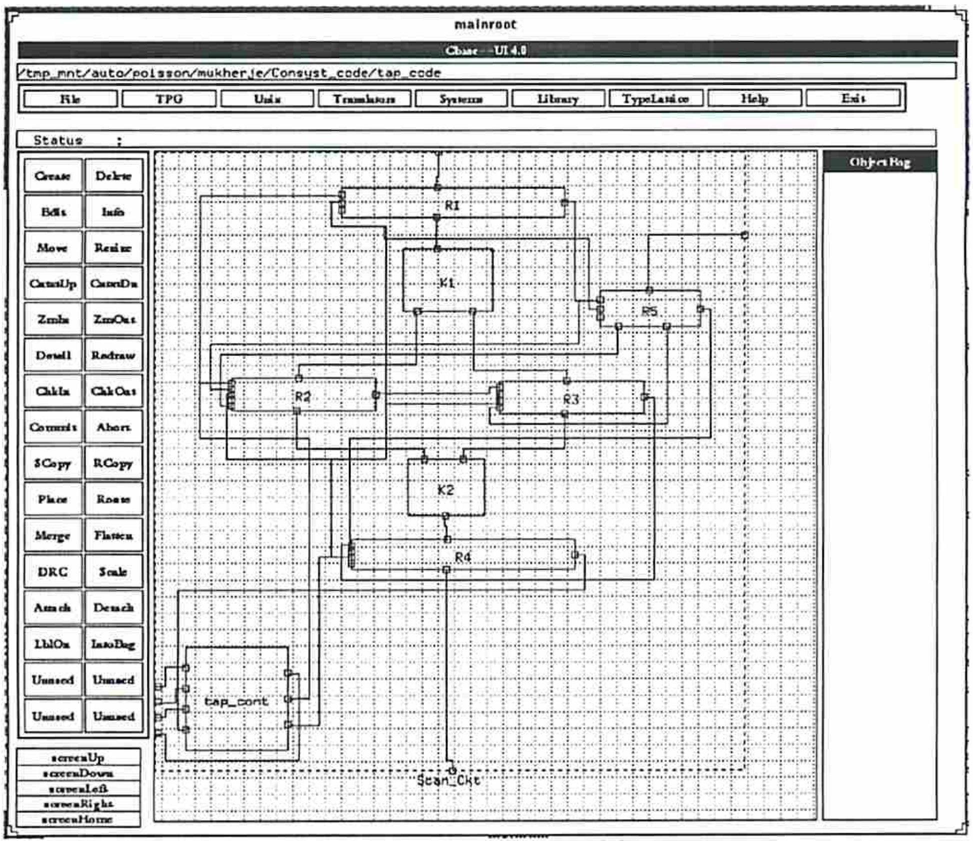


Figure 9.9: Scan_Ckt processed by CONSYST

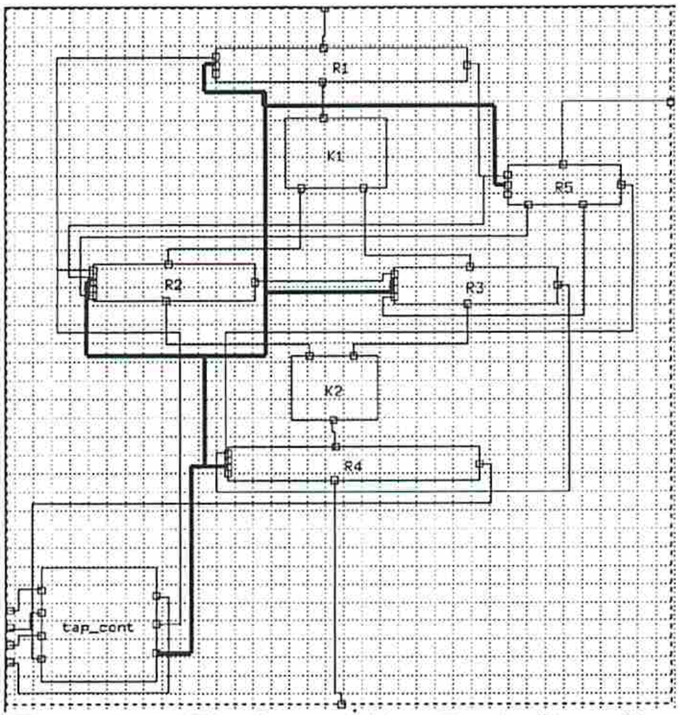


Figure 9.10: The Internal Test Bus in Scan_Ckt

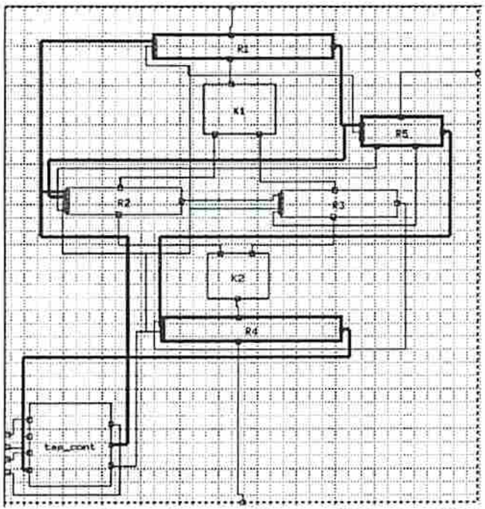


Figure 9.11: Boundary Scan Chain (Chain 0) in Scan_Ckt

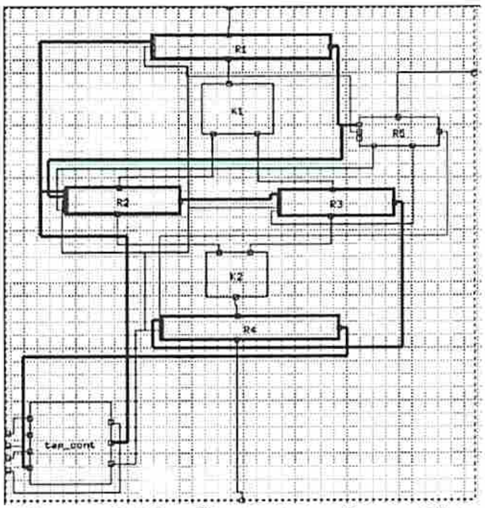


Figure 9.12: Chain 1 in Scan_Ckt

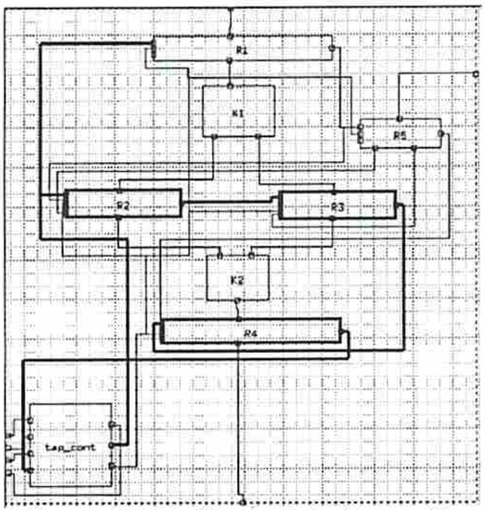


Figure 9.13: Chain 2 in Scan_Ckt

a 2-way multiplexer that selects between scan chains 1 and 2. The ITB which is 6 bits wide is shown connected to 4 input ports on the decoder. This implies that R2's decoder only uses a subset of the inputs from the internal test bus. CONSYST recognizes don't care inputs to logic blocks after logic minimization and drops them from consideration. Figs 9.14(c),(d) and (e) show implementations of the 2 bit register, decoder and the multiplexer, respectively.

9.6.1.2 Implementation of the TAP block

Screendump of the TAP block for *Scan_Ckt* is shown in Fig 9.15. Fig 9.16 is a cleaner representation of the TAP. The lower levels of the hierarchy have not been shown. Since the Standard bus is used, the *IR_code_translator* (see Chapter 4) is implemented. This logic block has input from the IR and the output drives part of the internal test bus. The number of bits in the IR and the number of inputs of the multiplexer (and the implementation of the mux select block C_4) in *mux_block* vary from design to design depending on the number of instructions and the number of data scan chains, respectively. The implementation of the ITAPC and the *TAP_decoder* depends on the bus scheme and the bus encoding used for a particular design.

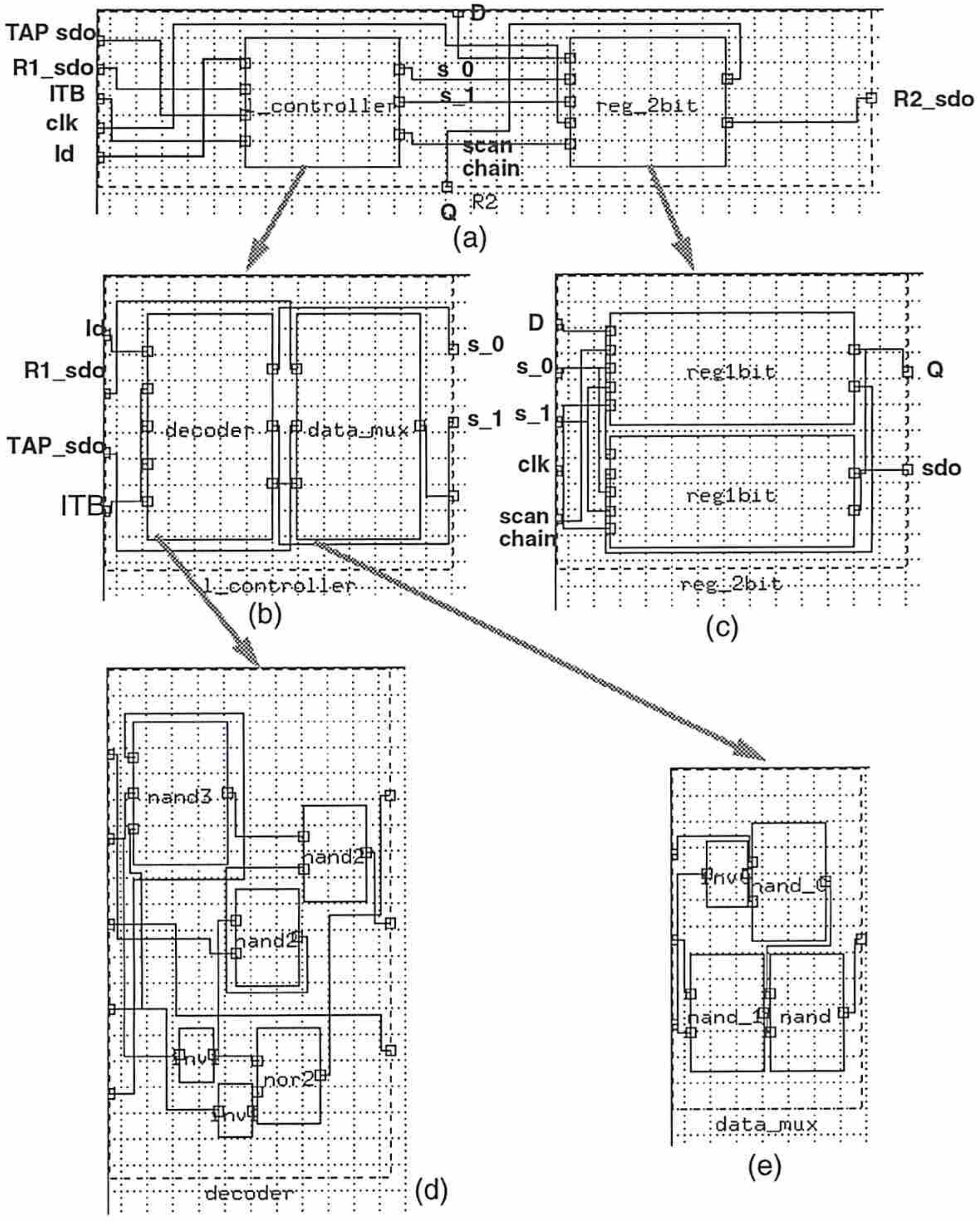


Figure 9.14: Hierarchical View of R2 in Scan_Ckt

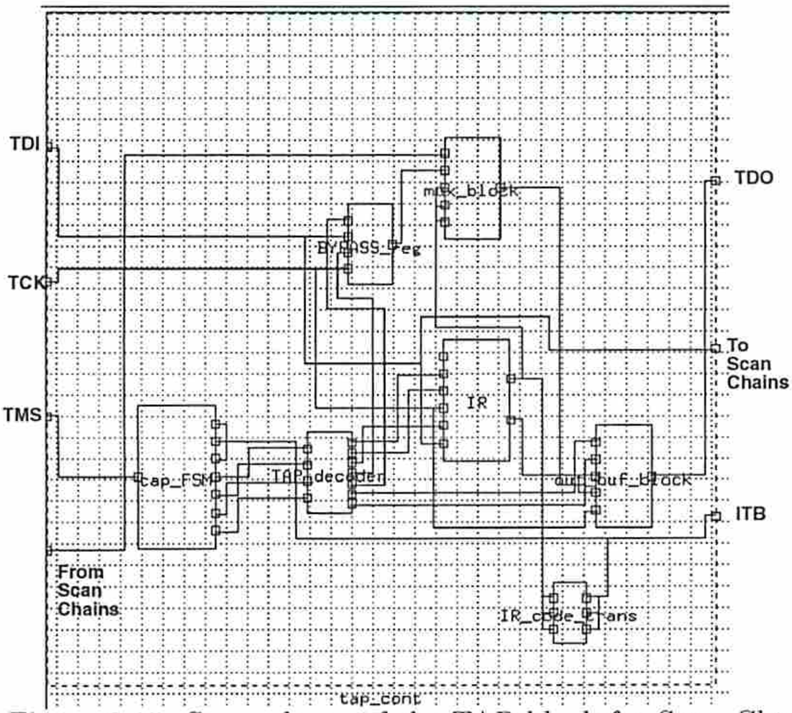


Figure 9.15: Screendump of the TAP block for Scan_Ckt

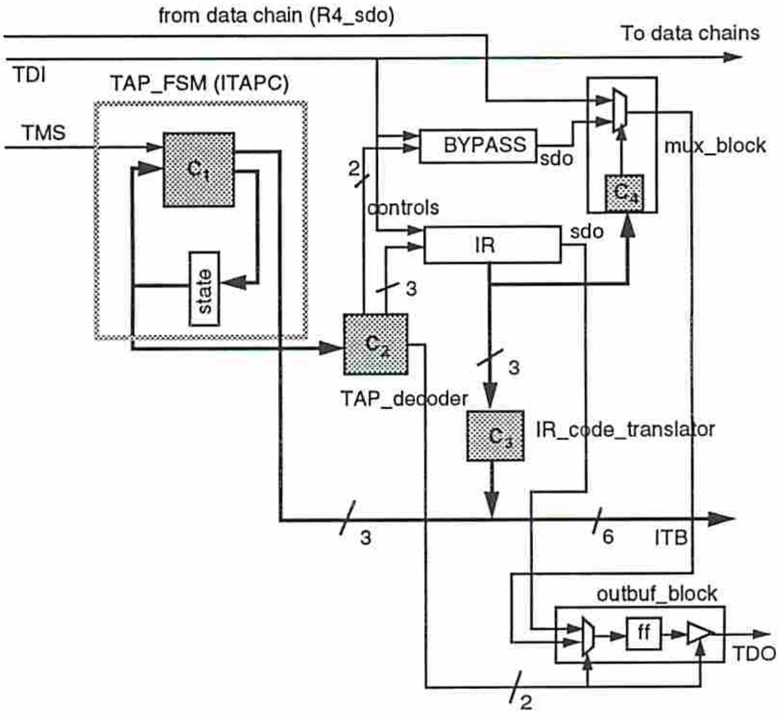
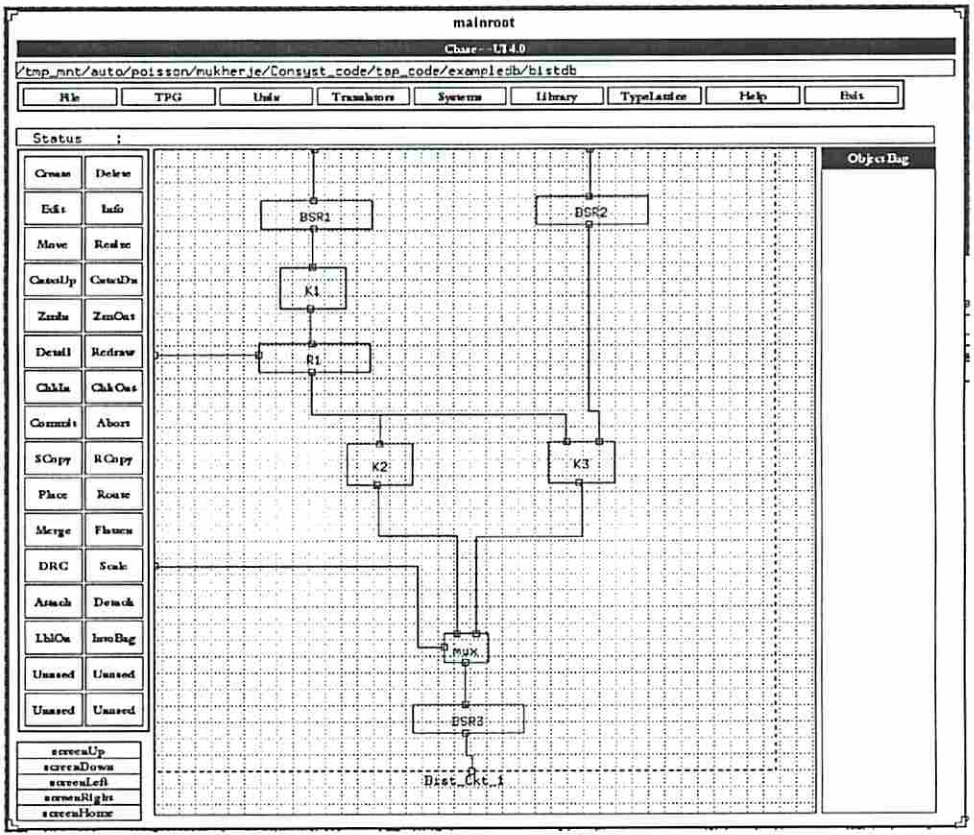


Figure 9.16: The TAP block for Scan_Ckt

9.6.2 BIST Circuit - Bist_Ckt_1

The non-testable (original) version of the circuit Bist_Ckt_1 is shown in Fig. 9.17. Fig. 9.18 shows a view of the top level of Bist_Ckt_1 after processing through CONSYST using the TPF presented in Section 9.3.3.1. The internal test bus (Sub-compact bus) is highlighted in Fig. 9.19. Scan chains 0 and 1 and the registers associated with these chains are highlighted in Figs. 9.20 and 9.21.



/tmp_mnt/auto/poisson/mukherje/Consyst_code/tap_code/exampledb/bistdb

Status :

Figure 9.17: Bist_Ckt_1

9.6.2.1 Hierarchical View of a Register in Bist_Ckt_1

A view of the lower levels of the hierarchy of register R_1 is shown in Fig 9.22. In this figure, the labels in bold face have been manually added. Fig 9.22(a) shows the local controller (*l_controller*) and the register flip-flops (*reg_4bit*). The input *poly_inp* is the output of an ex-or gate in BSR2. Fig 9.22(b) provides a lower level view of the local controller. This controller has a decoder, a 3-way multiplexer, a block (*reg_2bit*) containing one mode and one configuration flip-flop and a block (*poly_cell*)

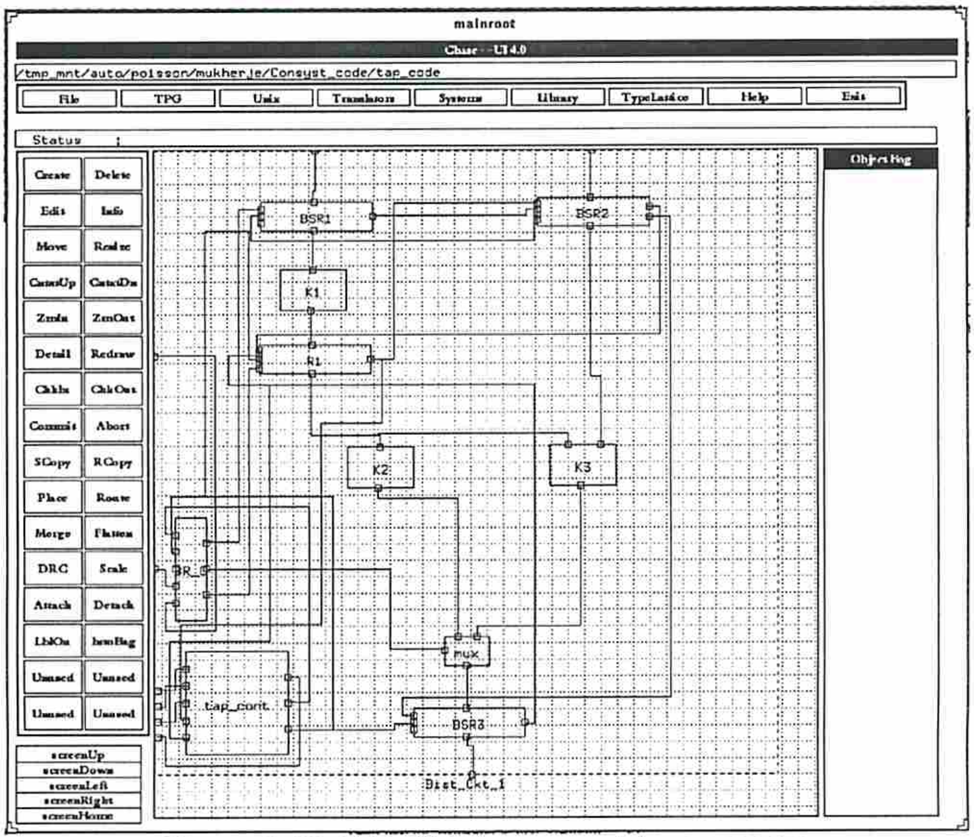


Figure 9.18: Bist_Ckt_1 processed by CONSYST

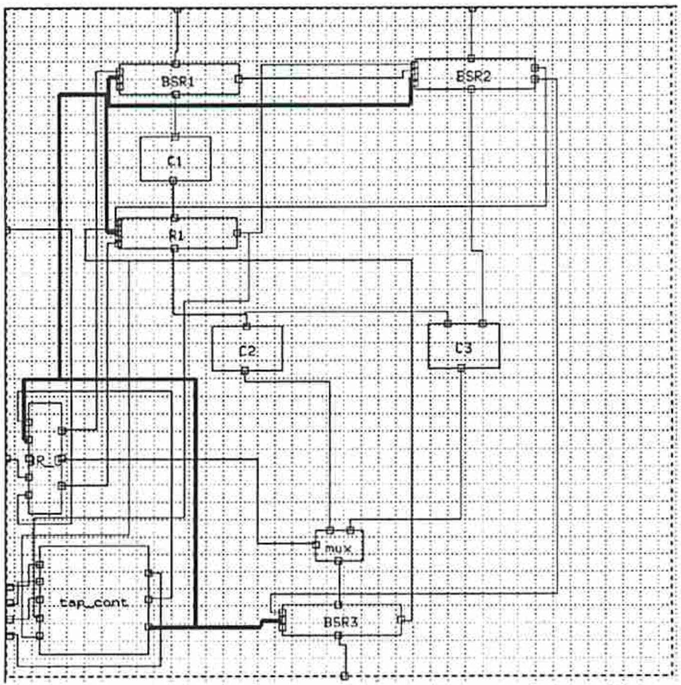


Figure 9.19: The Internal Test Bus in Bist_Ckt_1

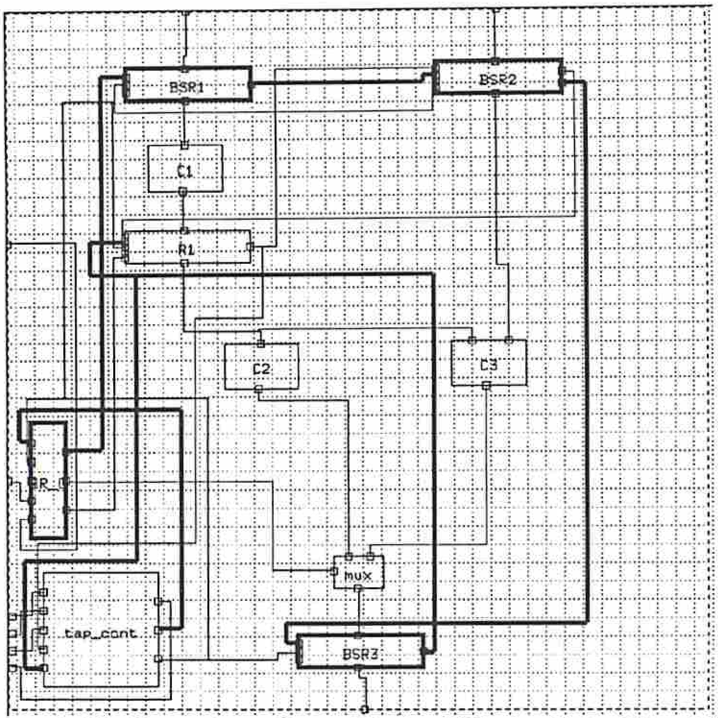


Figure 9.20: Boundary Scan Chain (Chain 0) in Bist_Ckt_1

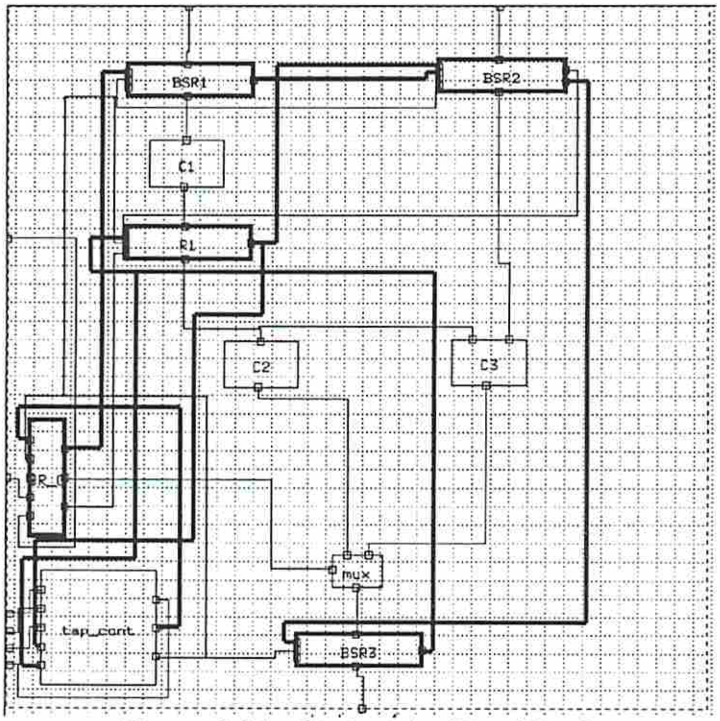


Figure 9.21: Chain 1 in Bist_Ckt_1

containing ex-or gates that implement the PG/SA polynomials. The multiplexer selects between the scan chain (chain 1) and outputs of ex-or gates. The ITB which is 4 bits wide is shown connected to 3 input ports on the decoder. Fig 9.23 shows two ex-or gates implementing feedback polynomials.

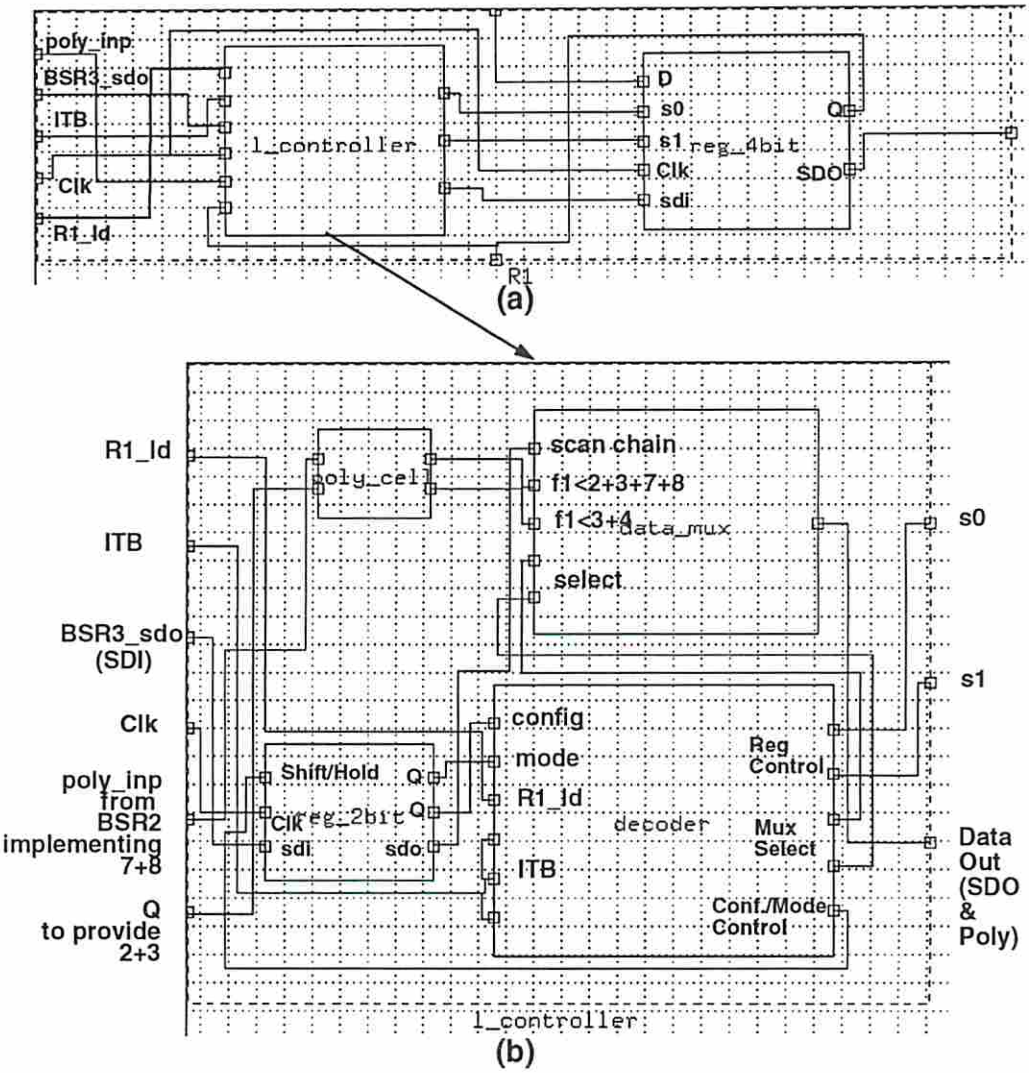


Figure 9.22: Hierarchical view of register R1 and the local controller

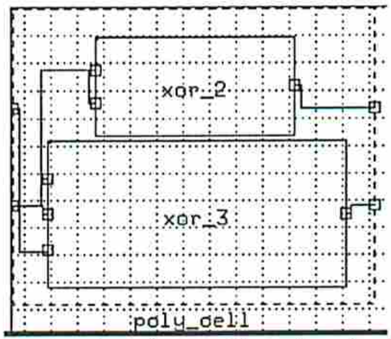


Figure 9.23: Ex-or gates implementing the feedback polynomials

9.6.3 BIST Circuit - Bist_Ckt_2

The non-testable (original) version of the circuit Bist_Ckt_2 is shown in Fig. 9.24. Figs. 9.25 and 9.26 show top level views of Bist_Ckt_2 after processing through CONSYST using the TPF presented in Section 9.4.1.1. The internal test bus (Sub-compact bus) is highlighted in Fig. 9.27. The internal test controller (labeled *int_controller*) is also shown. Details of the internal controller for this example are provided in Section 9.4.1. A careful inspection of Fig. 9.26 also reveals that certain control lines are driven directly from the boundary scan register BR_C, while others are driven from the internal test controller. The multiplexer that cuts the least significant bit of the bus control line is inside the block *int_controller*. The *session* register is also inside this block. The session register and the single flip-flop constituting the internal controller are appended to the data scan chain (Chain 1). Both the session register and the internal controller have local decoders and therefore the internal test bus is an input to *int_controller* (Fig. 9.27).

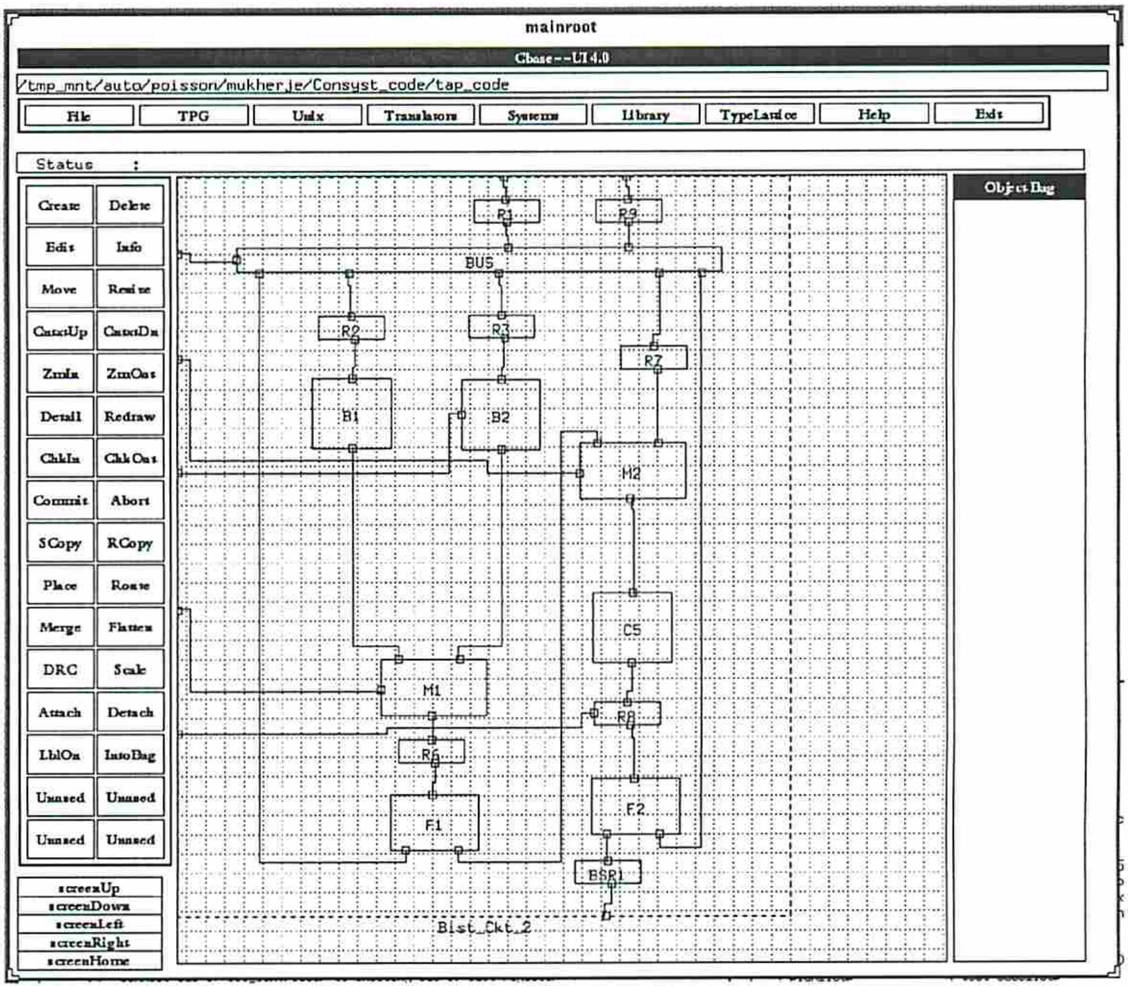


Figure 9.24: Bist_Ckt_2

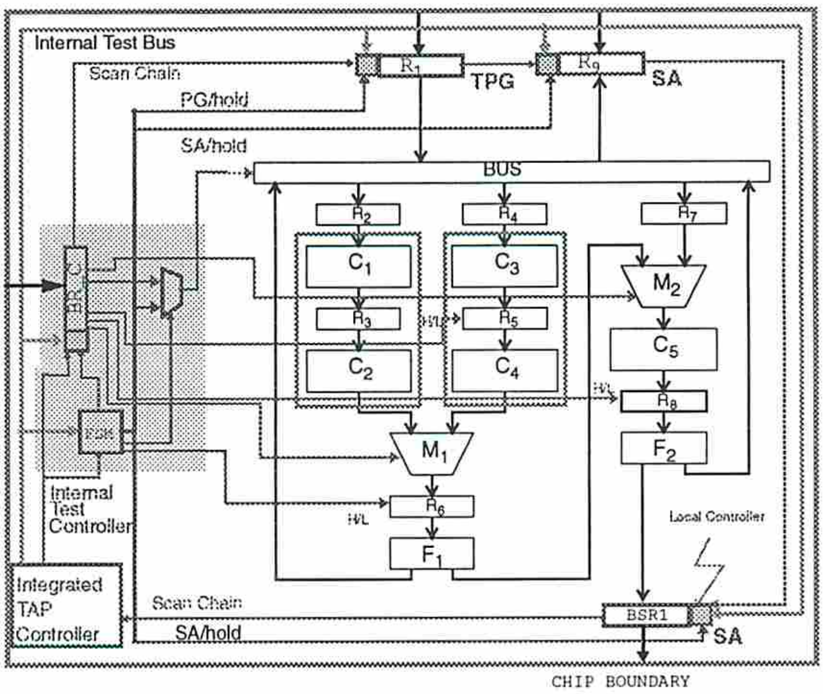
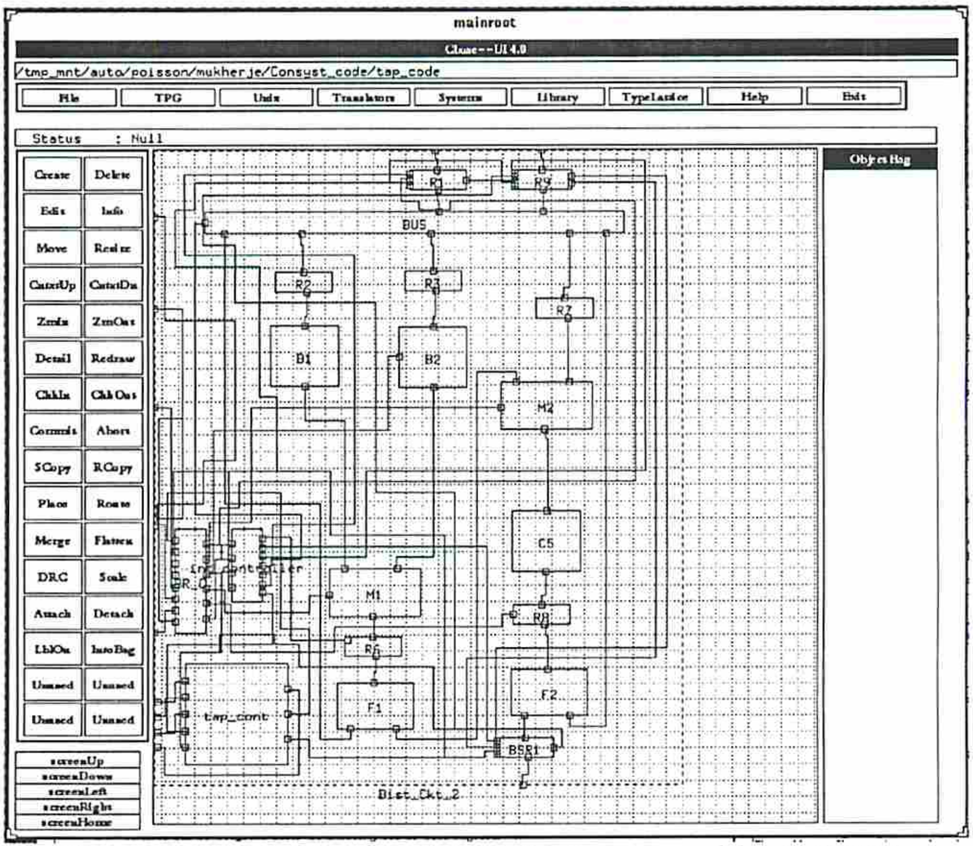


Figure 9.25: Bist_Ckt_2 processed by CONSYST



/tmp_mnt/auto/poisson/mukherje/Consyst_code/tap_code

Status : Null

Figure 9.26: Cbase view of Bist_Ckt_2 processed by CONSYST

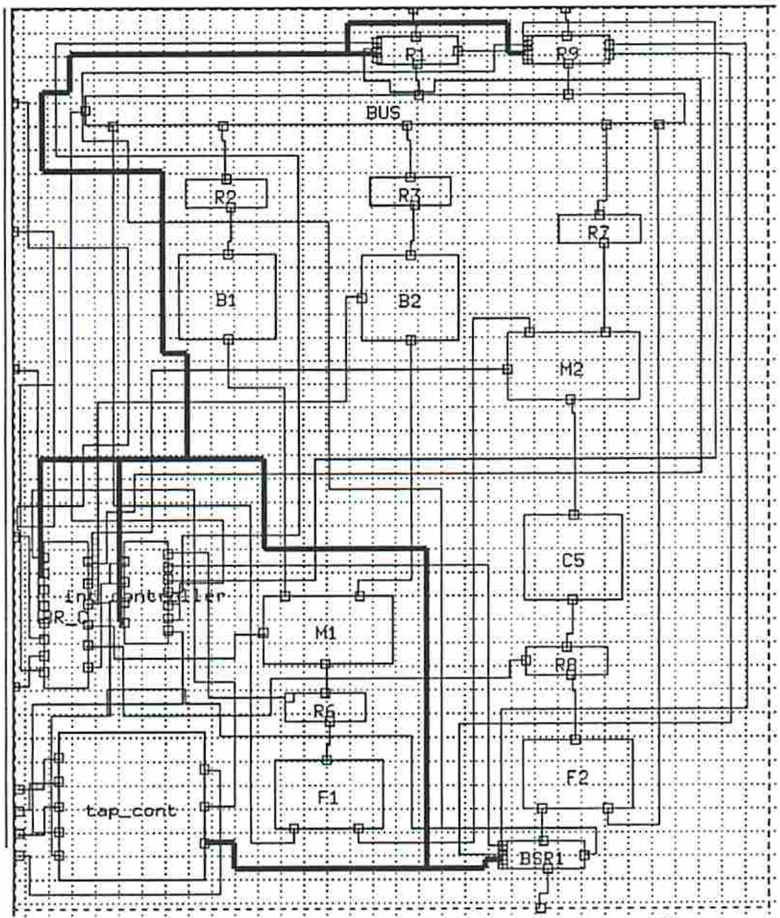
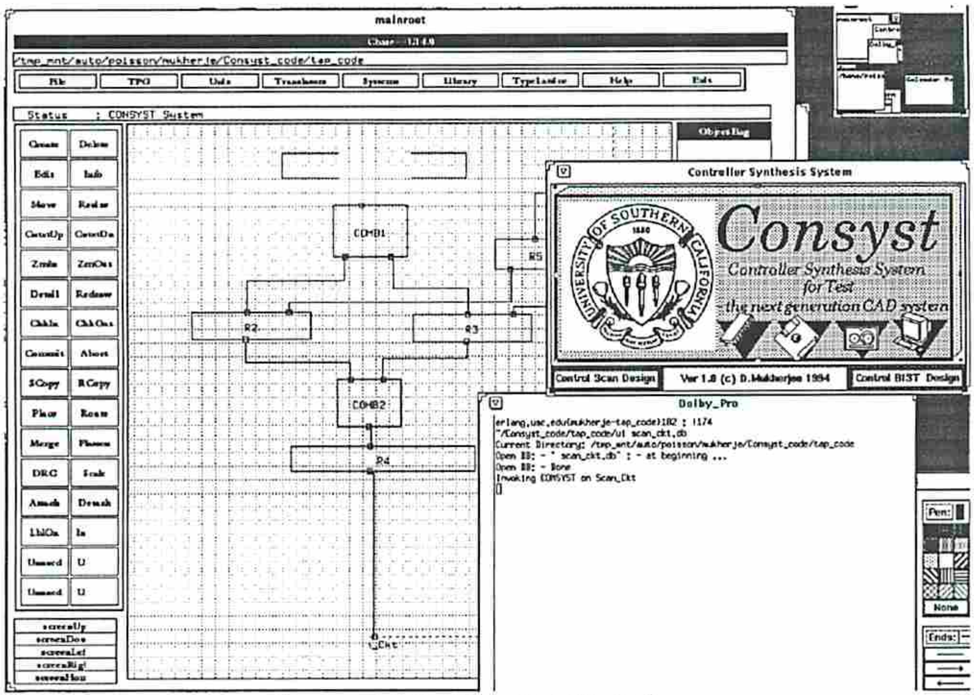


Figure 9.27: The Internal Test Bus in Bist_Ckt_2

9.7 Incorporating a Graphical User Interface

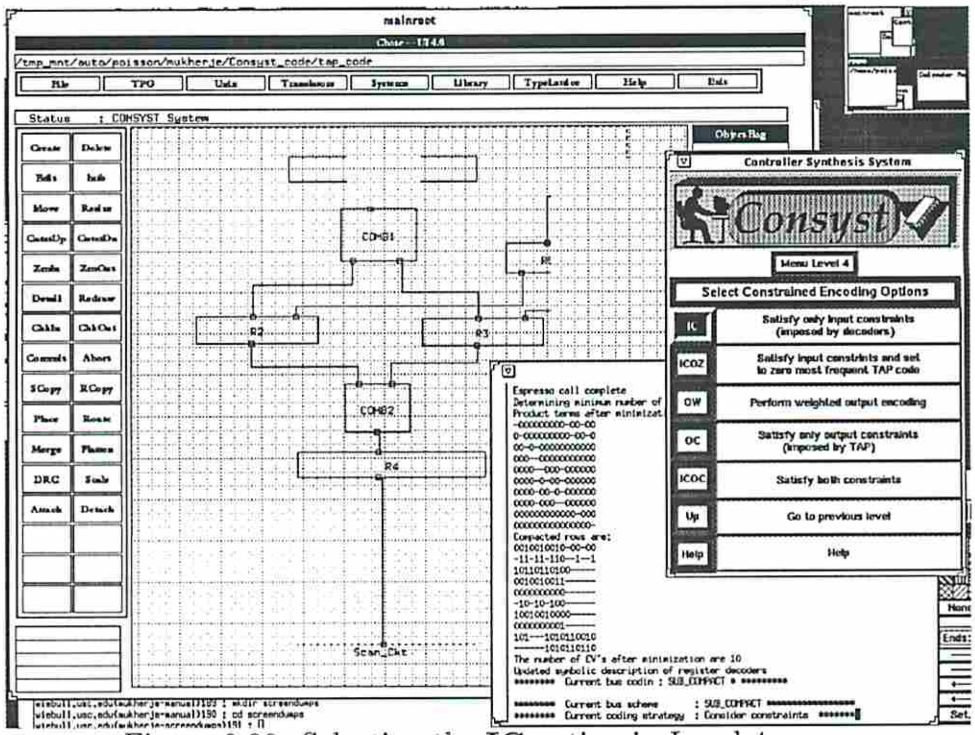
To simplify user interaction with the various subsystems and options provided by CONSYST we have developed graphical menu driven user interfaces. Various GUI building tools were evaluated and the Tcl/Tk/XF packages were chosen. XF [75] is an integrated programming environment that accelerates the development of GUIs. XF uses Tk [76], a Motif-like widget set that is accessible through Tcl [77, 76], which is a very efficient interpreted programming language. Tcl and Tk were developed at the University of California, Berkeley. We use XF to graphically create the layouts of the user interfaces used in CONSYST. The layout descriptions are saved as Tcl/Tk code. Additional Tcl/Tk code is then added to these descriptions to customize the interfaces for CONSYST. The Tcl command interpreter is called directly from the C/C++ code in CONSYST.

Examples of menus developed for CONSYST are given in Figs 9.28 and 9.29 [78].



```
erlang,usc,edu(mukherje-tap_code)182 : 1174
"/Consyst_code/tap_code/ul scan_ckt,db
Current Directory: /tmp_mnt/auto/poisson/mukherje/Consyst_code/tap_code
Open DB: - " scan_ckt.db" : - at beginning ...
Open DB: - Done
Invoking CONSYST on Scan_Ckt
```

Figure 9.28: The CONSYST logo



```
Espresso call complete
Determining minimum number of
Product terms after minimizat
-000000000-00-00
0-000000000-00-0
00-0-0000000000
000--0000000000
0000--000-00000
0000-0-00-000000
0000-00-0-000000
0000-000--000000
00000000000-000
00000000000000-
Compacted rows are:
0010010010-00-00
-11-11-110--1--1
10110110100-----
0010010011------
000000000-------
-10-10-100------
10010010000-----
000000001-------
101---1010110010
------1010110110
The number of CV's after minimization are 10
Updated symbolic description of register decoders
********  Current bus codin : SUB_COMPACT * *********

********  Current bus scheme      : SUB_COMPACT ****************
********  Current coding strategy  : Consider constraints  *******
```

```
wiebull.usc.edu(mukherje-manual)189 : mkdir screendumps
wiebull.usc.edu(mukherje-manual)190 : cd screendumps
wiebull.usc.edu(mukherje-screendumps)191 : []
```

Figure 9.29: Selecting the **IC** option in Level 4 menu

9.8 Summary

In this chapter we described the high level structure of the test controller synthesis system. Examples were presented to highlight various aspects of the Test Plan Description Language. The hardware incorporated into testable designs after processing through CONSYST was also shown.

Chapter 10

Conclusions

In this thesis we have addressed the problem of efficiently designing the test control circuitry for a testable chip. We have developed a novel test controller synthesis system that automatically synthesizes and incorporates test control and resource hardware into a circuit. This system is unique in its global outlook at the test control problem and in its approach to tightly coupling the processes of logic synthesis and testable circuit design. Some of the synthesis techniques presented in this thesis also have applications beyond the test domain. In the following subsections we summarize the contributions of the research presented in this thesis and discuss future research problems.

10.1 Summary of Contributions

10.1.1 Test Control Architecture

- We have justified the adoption of a specific on-chip/off-chip partition of test hardware, where the test vectors/results are stored off chip while pattern-generators/signature compactors and all test control hardware is incorporated on-chip.

- A *partially distributed* test control architecture has been developed. In this scheme all scannable registers have local controllers while the controllers for data transport paths have been centralized into one controller called the internal test controller. A test bus transmits control information to the distributed decoders and the internal test controller.

- Models of local controllers for scan and BIST registers have been developed. These models are general enough to represent multifunction and/or reconfigurable BIST registers.

10.1.2 Bus-based Control Schemes

- A different view of the traditional TAP controller has been incorporated in the form of an Integrated TAP Controller (ITAPC).

- We have analyzed the complexity of controlling test resources on a chip directly from the ITAPC by computing bounds on the number of control lines in scan and BIST designs. It has been shown that the direct control scheme is not a viable approach for large and complex designs.

- Three bus schemes have been proposed that tradeoff the number of lines in the bus with the implementation cost of the distributed decoders and the ITAPC. The *Standard* bus transmits information pertaining to relevant TAP states and instructions in the IR separately.

- The number of bus symbols are reduced by first determining Control Vectors (CVs) and then compacting them either by preserving the don't cares (leading to the Compact bus) or by optimally assigning 1's and 0's to don't cares (leading to the Sub-compact bus). Preserving the don't cares in the CVs preserves don't cares in the outputs of the decoders thereby aiding the task of synthesis tools, while converting don't cares to 1's and 0's usually leads to a reduced number of bus symbols. We have developed procedures to minimize the CV's for both these approaches. The significance of these procedures is in their ability to handle binary valued inputs. Proofs of the validity of these minimization techniques have been presented.

- We have addressed the problem of encoding the bus symbols to (1) minimize the implementation cost of the distributed decoders without considering the cost of the ITAPC (*input encoding*), (2) minimize the implementation cost of the ITAPC without considering the cost of the decoders (*output encoding*), and (3) minimize the implementation cost of both the distributed decoders and the

ITAPC (*input-output encoding*). We have developed a number of experimentally verified good heuristic techniques to solve the input, output and input-output encoding problems. For input encoding we have developed weighting functions that weight each input constraint (in the set of input constraints of all decoders) based on its occurrence in the constraint set of different decoders and the number of inputs and outputs of the corresponding decoders. For output encoding we have developed two algorithms. The first assigns codes to symbols with weights based on the frequency of occurrence of these symbols in the primary (symbolic) output of the ITAPC STT. The second satisfies *dominance* constraints and is based on decomposing the problem of encoding two symbolic output variables of a logic function into the problem of recursively encoding a logic function with one symbolic output variable. Two algorithms have also been developed for input-output encoding. The first employs the input encoding algorithm to obtain an initial encoding of the symbols and then assigns the all 0's code to the symbol that occurs most frequently in the output of the ITAPC STT. The second algorithm satisfies a set of weighted input constraints and dominance constraints.

- We have experimentally shown that our encoding algorithms reduce the standard cell and PLA areas of the test control logic by 34% and 24% (26% and 16%) for a Sub-compact bus (Standard bus), respectively, as compared to random encodings.

10.1.3 Controlling IEEE 1149.1 Compliant Scan and BIST Designs

- We have presented generic test plans for full, balanced partial and unbalanced partial scan designs as well as techniques for controlling the test hardware of these designs from the TAP controller. Hardware modifications of input boundary scan registers to control load/hold lines of registers in unbalanced partial scan designs have been proposed.

- We show how a test counter is incorporated in BIST designs and describe the impact of the presence or absence of this counter on the test sessions and test activation process of chips. This analysis is performed for different board level configurations.

10.1.4 Merging Test Plan Controllers

- We have presented a technique for merging a set of test plan controllers. The test controllers are orthogonal FSMs with a single transition out of (into) each state. The controllers are ordered in non-increasing order of the number of states and then merged one at a time to form an intermediate merged machine. Tight upper and lower bounds on the cost of the merged controller have been derived. A cost function based on minimizing the multiple-valued covers of slices of the merged machine has been proposed for optimally merging a test controller with an intermediate merged machine. We show that the cost function has a very strong correlation with the implementation cost of a merged machine obtained by minimizing a multiple-valued cover of the entire machine.

- Experimental results have shown that our merging procedure produces merged machines that, after state and input encoding using minimum number of bits, have on average 33% and 24% less product terms and area, respectively, than machines produced using an existing state minimizer.

- We have observed that the effect of minimizing the number of implicants in a multiple-valued cover of a merged machine (our cost function) percolates through all subsequent sequential synthesis steps.

- We have also experimentally shown that our merge procedure is relatively insensitive to the order in which the machines are merged.

10.1.5 Merging Test and Functional Controllers

- An efficient technique has been developed to merge a FSM that controls a number of test plans with the on-chip functional controller. The two machines operate in disjoint time frames and have disjoint sets of inputs. Our technique targets either two-level or multilevel implementation. For two-level implementation, a cost function based on minimizing the number of edges in the merged machine has been formulated. It has been shown that under certain conditions edge-minimization is equivalent to minimizing the cardinality of a strictly output disjoint multiple-valued cover. For multi-level minimization a cost function based on minimizing the number of literals by considering all state pairs has been developed.

- Experimental results show a strong correlation between our cost functions and the final implementation cost. For multi-level implementation our technique produces merged machines that have on average 20% less factored form literals than machines produced by an existing state minimizer. It has also been shown that merging the controllers, as opposed to implementing them separately and accessing the datapath control lines through a multiplexer, leads to significant savings in area.

10.1.6 Synthesis of 1-hot Coded Test Controller

- An implicit enumeration procedure has been developed to merge a number of test plan controllers where the merged machine has a 1-hot code state assignment. Bounds on the number of states, arcs, next state logic and output logic of the merged machine have been derived. Sufficient conditions have been developed for obtaining the minimum cost next state logic.

- Experimental results show that the literal count of designs synthesized by our approach is 20% to 50% less than designs synthesized using standard synthesis tools.

10.1.7 CONSYST System

- The grammar for a test plan description language has been defined. This language is powerful enough to handle the descriptions of reconfigurable scan chains, reconfigurable/multifunctional registers, registers that are implemented as LFSR/SRs and a spectrum of scan and BIST TDMs.

- The system employs a wide range of synthesis tools and completely automates the process of incorporating test control and resource hardware in a chip.

10.2 Future Research and Development

10.2.1 Bus-based Control

- The problem of efficiently encoding multiple symbolic outputs of a FSM is an open problem. An efficient solution to this problem will also result in an efficient solution to the problem of encoding the bus to minimize the implementation cost of the ITAPC. The ITAPC has two symbolic outputs, one being the test bus and the other the next state. The first step in solving this problem is to develop techniques to obtain a set of "good" dominance constraints. Then a weighted compatibility graph may be used to extract a set of compatible constraints that result in a maximal reduction in the size of the FSM cover.

- A number of encoding algorithms have been proposed in this thesis, but it is difficult to predict which encoding scheme will produce the best results, i.e., minimize the cost of the decoders and the ITAPC. Therefore efficient estimators need to be developed to predict the efficacy of the encoding algorithms for a particular design. The estimation can also be extended to take into account the routing area of the test bus, thereby solving the problem of selecting between one of the three different bus schemes.

- Currently each scannable register has its own local controller. Test control logic area may be reduced by sharing local controllers between registers that have the same functions and are placed close to each other. This can be handled by adding a post processor to CONSYST that merges local controllers based on information about register functions and predicted placement.

10.2.2 Controller Merging

- The procedure for merging test controllers optimally merges a pair of machines at a time. This procedure needs to be extended to consider the simultaneous merger of all the machines.

- The problem of efficiently merging two (or more) arbitrary FSMs is still open. Current state minimizers are unable to explore the space of all state minimal machines and pick the one that leads to a minimal implementation cost. The first step in solving this problem is to extend the procedure for merging test and functional controllers to handle non-disjoint sets of primary inputs. This will lead to a procedure for efficiently merging two controllers that operate in disjoint time frames and even though this is not the most general case, applications can be found in real designs. For example, a bus master that switches between two protocols depending upon an input condition can use such a procedure to merge the two FSMs that handle each of the protocols.

- Microprogrammed controllers are used in many large chip designs. Test microcode is needed to make such a design testable. Currently test microcode is usually appended to the functional microcode and it may be advantageous to develop efficient techniques to merge test and functional microcode. Specifically the problem is as follows : given individually compacted horizontal functional microprograms, merge the test microprograms with the functional microprogram to minimize the total number of microinstructions.

Reference List

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, New York, NY, 1990.

[2] B. Konemann, J. Mucha, and G. Zwiehoff. Built-in logic block observation technique. In *Proc., Int'l. Test Conf.*, pages 37–41, 1979.

[3] M. S. Abadir and M. A. Breuer. A knowledge based system for designing testable VLSI chips. *IEEE Design and Test of Computers*, pages 56–68, August 1985.

[4] S. P. Lin, C. Njinda, and M. A. Breuer. A systematic approach for designing testable VLSI circuits. In *Proc., Int'l. Conf. on Computer-Aided Design*, pages 496–499, November 1991.

[5] A. W. Nagle, R. Cloutier, and A. C. Parker. Synthesis of hardware for the control of digital systems. *IEEE Trans. on Computers*, 31(10):201–212, October 1982.

[6] P. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corporation, New York, NY, 1981.

[7] J. Hartmanis. Two tests for the linearity of sequential machines. *IEEE Trans. on Electronic Computers*, pages 781–786, December 1965.

[8] W. A. Davis and J. A. Brzozowski. On the linearity of sequential machines. *IEEE Trans. on Electronic Computers*, 15(2):21–29, February 1966.

[9] IEEE Standard 1149.1-1990. *IEEE Standard Test Access Port and Boundary Scan Architecture.* IEEE Standards Board, New York, NY, 1990.

[10] J. C. Lien and M.A. Breuer. A universal test and maintenance controller for modules and boards. *IEEE Trans. on Industrial Electronics*, 36(2):231–240, May 1989.

[11] J. C. Lien. *Design of Hierarchically Testable and Maintenable Systems.* PhD thesis, Univ. of Southern California. Dept. of Electrical Engr. - Systems. Technical Report CENG 91-19, June 1991.

[12] R. P. van Riessen, H. G. Kerkhoff, and A. Kloppenberg. Designing and implementing an architecture with boundary scan. *IEEE Design and Test of Computers*, 7(2):9–19, February 1990.

[13] F. Beenker, R. Dekker, R. Stans, and M. van der Star. Implementing macro test in silicon compiler design. *IEEE Design and Test of Computers*, 7(3):41–51, April 1990.

[14] O. F. Haberl and T. Kropf. HIST : A methodology for the automatic insertion of a hierarchical self test. In *Proc., Int'l. Test Conf.*, pages 732–741, September 1992.

[15] Y. Zorian. A distributed BIST control scheme for complex VLSI devices. In *Proc., VLSI Test Symp.*, pages 4–9, April 1993.

[16] J Maierhofer. Hierarchical self-test concept based on the JTAG standard. In *Proc., Int'l. Test Conf.*, pages 127–134, September 1990.

[17] J. Leenstra and L. Spaanenberg. Hierarchical test assembly for macro based VLSI design. In *Proc., Int'l. Test Conf.*, pages 520–529, September 1990.

[18] S. P. Lin. *A design system to support built-in self test of VLSI circuits using BILBO oriented test methodologies.* PhD thesis, Univ. of Southern California. Dept. of Electrical Engr. - Systems, May 1994.

[19] J. Beausang and A. Albicki. Towards determining an optimal test control line distribution scheme for a self-testable chip. Technical Report Tech. Rep. EL-86-04, Dept. of Electrical Engr., The Univ. of Rochester, March 1986.

[20] M.A. Breuer, R. Gupta, and J. C. Lien. Concurrent control of multiple BIT structures. In *Proc., Int'l. Test Conf.*, pages 431–442, September 1988.

[21] E. J. Marinissen. Automated test control block generation and minimization. Master's thesis, Eindhoven Univ. of Technology, January 1990.

[22] E. J. Marinissen and R. Dekker. Minimization of test control blocks. In *Proc., European Test Conf.*, pages 427–436, April 1991.

[23] V. D. Agrawal and K.T. Cheng. Test function specification in synthesis. In *Proc., 27th Design Automation Conf.*, pages 235–240, June 1990.

[24] E. J. Smith and Z. Kohavi. Synthesis of multiple sequential machines. In *Proceedings, 7th Annual Symp. on Switching and Automata Theory*, pages 160–171, October 1966.

[25] A. Grasseli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Trans. on Electronic Computers*, 14(6):350–359, June 1965.

[26] C. V. S. Rao and N. N. Biswas. Minimization of incompletely specified sequential machines. *IEEE Trans. on Computers*, 24(11):1089–1100, November 1975.

[27] M. Yamamoto. A method for minimizing incompletely specified sequential machines. *IEEE Trans. on Computers*, 29(8):732–736, August 1980.

[28] G. D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *Proc., European Design Automation Conf.*, pages 184–191, March 1991.

[29] J. C. Lien. A module maintenance controller prototype. Technical Report CENG 90-14, Univ. of Southern California, 1990.

[30] N. Jarwala and C. W. Yau. The boundary-scan master: Architecture and implementation. In *Proc., European Test Conf.*, pages 1–10, April 1991.

[31] S. Narayanan, C. Njinda, R. Gupta, and M. Breuer. SIESTA: A multi-facet scan design system. In *Proc., European Design Automation Conf. (EURODAC)*, pages 246–251, September 1992.

[32] K.-T. Cheng and V.D. Agrawal. A partial scan method for sequential circuits with feedback. *IEEE Trans. on Computers*, 39(4):544–548, April 1990.

[33] V. Chickermane and J.H. Patel. An optimization based approach to the partial scan design problem. In *Proc., Int'l. Test Conf.*, pages 377–386, September 1990.

[34] R. Gupta, R. Gupta, and M. A. Breuer. The BALLAST methodology for structured partial scan design. *IEEE Trans. on Computers*, 39(4):538–543, April 1990.

[35] R. Gupta. *Advanced Serial Scan Design for Testability.* PhD thesis, Univ. of Southern California. Dept. of Electrical Engr. – Systems. Technical Report CENG 91-10, 1991.

[36] S. Narayanan. *Scan Chaining and Test Scheduling in an Integrated Scan Design System.* PhD thesis, Univ. of Southern California. Dept. of Electrical Engr. – Systems, December 1994.

[37] L. T. Wang and E. J. McCluskey. Concurrent built-in logic block observer (CBILBO). In *Proc., Int'l. Symp. On Circuits and Systems*, pages 1054–1057, 1986.

[38] K. Kim, D. S. Ha, and J. G. Tront. On using signature registers as pesudorandom pattern generators in built-in self testing. *IEEE Trans. on Computer-Aided Design*, 7(8):919–928, August 1988.

[39] S. P. Lin, S. K. Gupta, and M. A. Breuer. A low cost BIST methodology and a novel test pattern generator design. In *Proc., the European Design and Test Conf.*, pages 106–112, February 1994.

[40] M. Lempel, S. K. Gupta, and M. A. Breuer. Test embedding with discrete logarithms. In *Proc., VLSI Test Symp.*, pages 74–80, April 1994.

[41] D. Mukherjee, M. Pedram, and M. A. Breuer. Control strategies for chip-based DFT/BIST hardware. In *Proc., Int'l. Test Conf.*, October 1994.

[42] G. DeMicheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment of finite state machines. *IEEE Trans. on Computer-Aided Design*, 4(3):269–285, July 1985.

[43] T. Villa and A. Sangiovanni-Vincentelli. NOVA, state assignment of finite state machines for optimal two-level logic implementation. *IEEE Trans. on Computer-Aided Design*, 9(9):905–924, September 1990.

[44] C. L. Hudson and G. D. Peterson. Parallel self-test with pseudo-random test patterns. In *Proc., Int'l. Test Conf.*, pages 954–963, 1987.

[45] G. DeMicheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Trans. on Computer-Aided Design*, 5(4):597–616, October 1986.

[46] M. J. Ciesielski and J-J. Shen. A unified approach to input-output encoding for FSM state assignment. In *Proc.*, 28th *Design Automation Conf.*, pages 176–181, June 1991.

[47] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment, and four level Boolean minimization. *IEEE Trans. on Computer-Aided Design*, 10(1):13–27, January 1991.

[48] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, MA, 1984.

[49] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. on Computer-Aided Design*, 6(5):727–750, September 1987.

[50] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on Computer-Aided Design*, 6(6):1062–1081, November 1987.

[51] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE J. of Solid State Circuits*, 20(4):510–522, April 1985.

[52] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro. A new symbolic channel router: YACR2. *IEEE Trans. on Computer-Aided Design*, 4(2):208–219, March 1985.

[53] D. Mukherjee, M. Pedram, and M. A. Breuer. Minimal area merger of finite state machine controllers. In *Proc., European Design Automation Conf. (EURODAC)*, pages 278–283, September 1992.

[54] D. Mukherjee, M. Pedram, and M. A. Breuer. Merging multiple FSM controllers for DFT/BIST hardware. In *Proc., Int'l. Conf. on Computer-Aided Design*, pages 720–725, November 1993.

[55] D. Mukherjee, C. Njinda, and M. A. Breuer. A partially distributed control scheme for DFT/BIST hardware. In *Proc., WESCON*, pages 673–679, November 1992.

[56] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of the A* algorithm. *Journal of the Association of Computing Machinery*, 32(3):505–536, July 1985.

[57] D. Mukherjee, C. Njinda, and M. A. Breuer. Synthesis of optimal 1-hot coded on-chip controllers for BIST hardware. In *Proc., Int'l. Conf. on Computer-Aided Design*, pages 236–239, November 1991.

[58] C. Sechen and K. W. Lee. An improved simulated annealing algorithm for row-based placement. In *Proc., Int'l. Conf. on Computer-Aided Design*, pages 478–481, November 1987.

[59] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proc., IFIP Int'l. Conf. on VLSI*, pages 187–196, August 1989.

[60] S. Devadas, H. K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic

implementations. *IEEE Trans. on Computer-Aided Design*, 7(12):1290–1300, December 1988.

[61] S. Devadas, H. K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Synthesis and optimization procedures for fully and easily testable sequential machines. In *Proc., Int'l. Test Conf.*, pages 621–630, September 1988.

[62] R. Z. Makki, S. Bou-Ghazale, and C. Tianshang. Automatic test pattern generation with branch testing. *IEEE Trans. on Computers*, 40(6):785–791, June 1991.

[63] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, 1975.

[64] D. Mukherjee, C. Njinda, and M. A. Breuer. Synthesis of optimal 1-hot coded on-chip controllers for BIST hardware. Technical Report CENG 91-20, Univ. of Southern California, November 1991.

[65] S. P. Lin, M. A. Breuer, and C. Njinda. A self adaptive expert selection system (SAESS) and its application to selection systems. In *Proc., Third Int'l. Conf. on Software Engr. and Knowledge Engr.*, pages 116–121, June 1991.

[66] R. Gupta, W. H. Cheng, R. Gupta, I. Hardong, and M. A. Breuer. An object-oriented VLSI CAD framework : A case study in rapid prototyping. *IEEE Computer*, 22(5):28–37, May 1989.

[67] R. Jain, K. Kucukcakar, M. Mlinar, and A. Parker. Experience with the ADAM design system. In *Proc., 26^{th} Design Automation Conf.*, pages 56–61, June 1989.

[68] A. Casotto (editor). *Octtools 5.1.* Electronics Research Laboratory, Univ. of California, Berkeley, 1991.

[69] K. P. Parker and S. Oresjo. A language for describing boundary scan devices. In *Proc., Int'l. Test Conf.*, pages 572–581, October 1990.

[70] J. P. Weng and A. Parker. CSG: Control path synthesis in the ADAM system. Technical Report CENG 92-03, Univ. of Southern California, 1992.

[71] S. C. Johnson. Yacc : Yet Another Compiler-Compiler. In *Unix Programmer's Manual, 7th Edition*. Bell Laboratories, Murray Hill, NJ, 1978.

[72] M. E. Lesk and E. Schmidt. Lex : A Lexical Analyzer Generator. In *Unix Programmer's Manual, 7th Edition*. Bell Laboratories, Murray Hill, NJ, 1978.

[73] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1988.

[74] R. Srinivasan, S. K. Gupta, and M. A. Breuer. Novel test pattern generators for pseudo-exhaustive testing. In *Proc., Int'l. Test Conf.*, pages 1041–1050, October 1993.

[75] S. Delmas. Xf : Design and implementation of a programming environment for interactive construction of graphical user interfaces. Technical report, Technische Universitat Berlin, Institut fur Angewandte Informatik, 1993.

[76] J. K. Ousterhout. *Tcl and the Tk Toolkit*. To be published by Addison-Wesley, Reading, MA, 1994.

[77] J. K. Ousterhout. Tcl : an embeddable command language. In *Proc., USENIX Winter Conf.*, pages 133–146, January 1990.

[78] D. Mukherjee. CONSYST 1.0 User's Manual. Internal Report. Univ. of Southern California, August 1994.