TECHNIQUES FOR TEST GENERATION AND VERIFICATION OF VLSI SEQUENTIAL CIRCUITS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M91/73

3 September 1991

COVER PAGE

TECHNIQUES FOR TEST GENERATION AND VERIFICATION OF VLSI SEQUENTIAL CIRCUITS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M91/73

3 September 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

TECHNIQUES FOR TEST GENERATION AND VERIFICATION OF VLSI SEQUENTIAL CIRCUITS

by

Abhijit Ghosh

Memorandum No. UCB/ERL M91/73

3 September 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

List of Tables

List of Figures

Contents

Acknowledgments

I am grateful to my research advisor, Prof. A. Richard Newton, for his guidance, inspiration, criticism, and constant support during my study at Berkeley. I am also grateful to Prof. Srinivas Devadas of MIT for providing similar guidance, impetus, direction, and necessary criticism during my research.

I would like to thank Prof. Robert K. Brayton and Prof. Alberto Sangiovanni-Vincentelli for many stimulating discussions on test generation, verification, and logic synthesis. Special thanks to Prof. Brayton for being on my qualifying examination and thesis committees. I would also like to thank Prof. Jack Silver of the Mathematics Department for being on my qualifying examination and thesis committees and for his helpful suggestions.

Being a part of the CAD-group at Berkeley has been a unique experience. Various people have contributed in different ways to make this experience enjoyable and edifying. I would like to thank my erstwhile colleagues Jeff Burns, George Jacob, Karti Mayaram, Theo Kelessoglou, and Don Webber for answering many of my questions when I was a newcomer in the group. Special thanks to all the unix-experts, namely, Wendell Baker, Brad Krebs, Chuck Kring, and Rick Spicklemier for pulling me out of trouble many a times. I would like to thank Pranav Ashar, Brian Lee, Bill Lin, Abdul Malik, Rajeev Murgai, Brian O'Krafka, Alex Saldanha, Hamid Savoj, Ellen Sentovich, Narendra Shenoy, K.J. Singh, Hérve Touati, Yosinori Watanabe, and Greg Whitcomb for many interesting discussions on a wide variety of topics. Special thanks to Rajeev Murgai for reading pre-publication manuscripts. Hi-Keung Tony Ma has been very helpful both inside and outside the department. In addition to giving me his software, fixing bugs for me, and his healthy criticism, he has also been a great tennis and badminton partner. Thanks also to Wayne Christopher, Andrea Casotto, Mark Beardslee, Chris Lennard, Jaijeet Roychowdhury, and Lorraine Layer for their friendship. I would also like to thank Kia Cooper, Elise Mills, and Flora Oviedo for their help with travel grants, mailings, and in general making my life easier.

There are some wonderful people who have been instrumental in making my life, especially in Berkeley, very pleasurable. I would like to thank them for what they have done for me — David, Paul, and Laurent for being silly and for their interest in agriculture; Lorna, Salima, Eliane, and Romella for their friendship, support, and love; Roberto and the rest of the gang for the parties in SF; Brinda for many entertaining and scintillating moments;

and Rhonda, Barbara, Micheline, Deborah, Cheryl, Ceri, Cynthia, Marilyn, Silvia, Sara, Martha, Kathy, Claudia, Giovanna, Susanna, Babu, Noeman, Rabi, Smarajit, Prashanta, Harald, Kinsuk, Kamal, Hitesh, Milind, Steve, James, Aditya, Ahmed, Joseph, Marco, and the rest of the group for being such good friends. Special thanks to my host family, Bob and Linda Mahley, for their help and their love.

I would like to thank my family, especially my parents and my uncle, for their monumental support, encouragement, and enthusiasm in the work that I was doing. Without their help, this work might not have been possible.

This research was supported in part by the Defense Advanced Research Projects Agency under contract JFBI90-073, Digital Equipment Corporation, and AT&T Bell Laboratories. Their support is gratefully acknowledged.

Techniques for Test Generation and Verification of VLSI Sequential Circuits

Abhijit Ghosh

Ph.D.

Department of Electrical Engineering
and Computer Science

Abstract

Very large-scale integrated circuits contain thousands of circuit components within a very small area. The design of such circuits is a complicated and time consuming process, and automatic design tools are used wherever possible to help or complement the designer. In addition to performance, the reliability of the manufactured product is of utmost importance. Testing is the process of ensuring that there are no defects in the manufactured circuit. One of the key problems in testing is that of automatic test pattern generation, especially for sequential circuits. Two new test generation algorithms for sequential circuits have been developed as a part of this dissertation. The first one uses novel ideas and heuristics for circuits described at the gate level. The second one uses and exploits the properties of higher level descriptions, namely, Register-Transfer level descriptions, for efficient test generation.

The design process involves the transformation of a design from one representation to another or a transformation within the same representation, using automatic optimization tools. The probability of introducing errors in a circuit during the design phase is high. Therefore, automatic tools that verify that the current representation of a circuit is the same as the original representation are needed. An algorithm for verification of sequential circuits described at the gate level has been developed. This algorithm uses implicit enumeration of the input as well as the state space.

To ease the task of test generation, it is necessary to synthesize circuits to be fully and easily testable, and these are the objectives of the synthesis for testability process. A new approach to synthesis for testability for sequential circuits is presented. It uses logic partitioning and exploits invalid and equivalent-state don't-cares to obtain a fully testable implementation of a circuit. An associated problem is that of minimization of Boolean relations. An algorithm to obtain a minimal implementation of a Boolean relation is presented. This algorithm uses well known testing techniques for logic optimization.

A. Richard Newton

Prof. A. Richard Newton
Thesis Committee Chairman

Chapter 1

INTRODUCTION

Very Large Scale Integrated (VLSI) circuits are an integral part of any modern electronic system. Such circuits contain from thousands to millions of transistors, diodes, and other devices, resistors, capacitors, and interconnections within a very small area. The design of such circuits is a complicated and time consuming process. *Synthesis* refers to the process of (automatically) designing or re-designing a circuit from a specification of the circuit. There are many sources of error that can produce an incorrectly functioning circuit. One of them could be an error in the specification. *Design verification* is the process of determining whether what the designer specified is what she/he wants. Once the specification is verified, an implementation of the circuit is derived. The error that produces an incorrectly functioning circuit could be in the design phase where either a human designer or an automatic design tool makes an error (probably due to an undetected bug). *Implementation verification* is the process of determining whether the designed circuit is the same as what was specified. *Logic verification*, which is a part of implementation verification, is the process of verifying the equivalence of two logic-level circuits, usually the optimized and the unoptimized ones. If an implementation of a design is correct and there are no manufacturing defects, then the manufactured circuit should function as per specifications. However, the manufacturing process may introduce defects in the circuit (*e.g.*, short circuits, open circuits, missing transistors, *etc.*). Even though a circuit has no manufacturing defects, it may become defective due to a variety of reasons (*e.g.*, physical or thermal stress, radiation, *etc.*) during the operation of the circuit. *Production verification* is the process of verifying whether the manufactured circuit is the same as what was specified. *Testing*, which is a part of production verification, is the process of determining whether a fabricated circuit is defective, and if so, identifying the location of the defect.

VLSI circuits can be divided into two classes – combinational circuits (without memory) and sequential circuits (with memory). Techniques for the automatic synthesis and verification of both kinds of circuits have been under investigation for a long time. Until recently, combinational circuits have received the bulk of researchers attention. Today, there are techniques that can synthesize [15, 16, 57] and generate tests [60, 65, 84, 91, 94, 119] for combinational circuits efficiently. There has been some effort in solving the sequential logic synthesis and test problem using combinational techniques [56, 87, 89]. The focus of this dissertation is the testing and logic verification of digital synchronous sequential logic, and the relationship of these techniques to synthesis of such circuits.

With the advances in integrated circuit (IC) technology, the number of devices that can be put on a chip has increased rapidly. This has greatly increased the complexity of the synthesis, verification, and testing process. It is necessary to find new synthesis strategies to synthesize circuits with minimal area, maximal performance, and improved testability. Previous approaches are not effective for these large circuits for a variety of reasons. However, without high-quality logic synthesis and verification tools, it might not be possible to design, implement, and market a reliable product. In this chapter the major issues involved in synthesis, verification, and testing are presented. In Section 1.1 a typical synthesis system is described. Issues in design verification and testing are the topics of Sections 1.2 and 1.3 respectively.

1.1 IC Design Systems

Synthesis involves a series of alternate mapping and optimization steps. These steps in a typical synthesis process are shown in Figure 1.1. Starting with an idea about what the designer wants, a behavioral description of the circuit in a high-level language like ELLA [104], VHDL [113], or ISPS [12] is written. The first step in the synthesis process is to convert the behavioral specification into a *Register-Transfer level* (RTL) description of the circuit. This process could be as simple as mapping the behavioral specification to a Register-Transfer level description. However, during the conversion process, in addition to mapping, behavioral synthesis tools [41, 69, 80, 111, 128, 132, 134] can be used to optimize the circuit to use the minimum amount of hardware within the required performance constraints. The design space is large and the tools explore various design trade-offs and try to produce

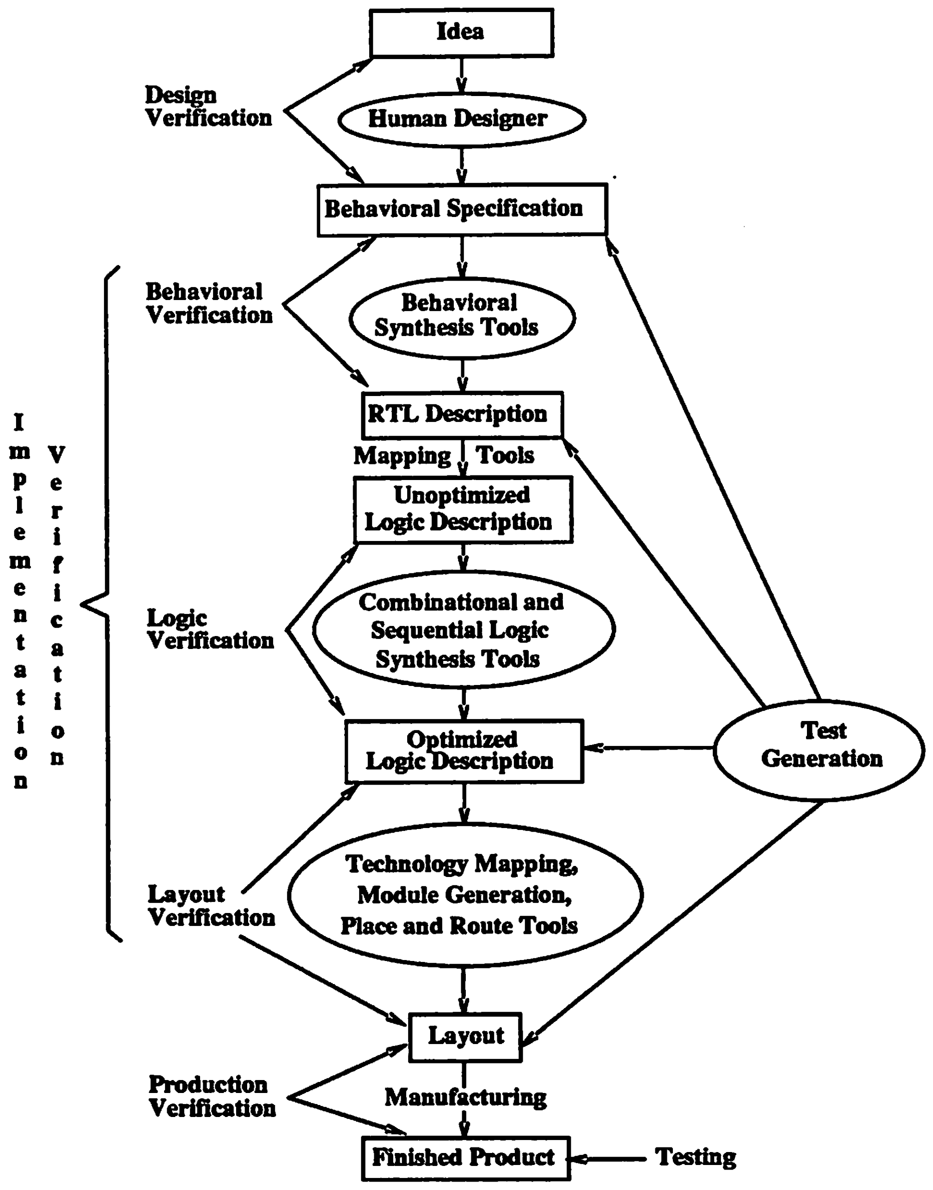


Figure 1.1: A typical synthesis pipeline

an optimal solution. It should be noted that the use of behavioral synthesis tools in the industry is currently quite limited.

After a Register-Transfer level description has been obtained, it is mapped into logic equations using mapping tools like BDSYN described in [122]. The next step is logic synthesis, which is an optimization step. Typically, a Register-Transfer level description is an interconnection of pre-defined modules like adders, multipliers, or finite state machines that implement controllers. This description often has redundant logic, and the task of logic synthesis tools [10, 15, 16, 40, 57] is to transform this description into a more optimal description of the circuit in terms of logic gates. The goal of logic optimization is to minimize the area while meeting the performance constraints. Another objective is to improve the testability of the circuit [45, 49].

The result of logic optimization is an optimized gate-level or logic-level description of the circuit. The next step in the synthesis process is to produce a mask-level description or a layout of the circuit in a given technology. Technology mapping [39, 82] refers to the mapping of an arbitrary logic-level description into an implementable logic network using a set of gates from a standard library of gates. This process involves both mapping the logic description into an implementable network, as well as optimizing the mapped description in order to meet area and performance constraints. Module generators [55, 109] may be used to produce a layout for each module in the design. Finally, the modules or gates are placed and routed using placement and routing tools like [31, 33, 114, 121]. This is both a mapping and an optimization phase that produces the final mask-level description of the circuit. This description can be used to manufacture the final product.

In a typical design process, the designer might have to iterate over these steps before an acceptable final circuit is produced. This is because constraints imposed on the design are not satisfied in the first pass. Information from any level could be fed back to a higher level to enable the tools working at that level to come to better decisions in order to satisfy the design constraints. For example, a behavioral synthesis tool might not know the cost of a module (in terms of area and performance) when it chooses the module. After layout, the cost of the module can be accurately estimated and used by the synthesis tools to make a better decision. The synthesis process can be made fully automatic. However, designer intervention and insight is often necessary to produce high-quality circuits. The synthesis process involves the solution of many optimization problems, most of which are conjectured to be intractable. Therefore, most tools use heuristics to obtain close-to-optimum solutions.

1.2 Implementation Verification

During the design phase many descriptions of the same circuit are produced. For example, starting from the behavioral description of the circuit, the RTL description, the logic-level description, and, finally, the mask-level description are produced. In addition, optimization tools produce many alternate designs at the same level. If the steps of synthesis are performed manually, there is always a high probability of introducing errors. Since bug-free software cannot be guaranteed, there is also a chance that automatic synthesis tools will produce an erroneous result for a particular circuit. Design errors might also be introduced due to the misunderstanding and the resulting misuse of an automatic tool. Since the probability of introducing errors is non-zero in both the automatic and the manual design environment, it is necessary to verify, at each design step, that the resulting description and the original description are identical. Manual verification is not possible due to the size of the circuits and due to the relatively high probability of error during verification. Reliable, automatic, and *independent* verification tools are necessary to ensure the correctness of the final design.

As shown in Figure 1.1, verification tools can be used at various steps in the design process. Design verification has relied mostly on simulation, though formal methods are slowly emerging [70]. The problem of verifying the equivalence of a behavioral description and an RTL description (behavioral verification) has been the subject of extensive investigation (*e.g.*, [4, 24, 42, 59]). Algorithms for verifying the equivalence of an RTL description and a logic-level description for both combinational and sequential circuits are presented in [46, 92]. Algorithms for verifying equivalence of alternate logic-level descriptions for circuits (logic verification) are presented in [36, 46, 92, 94]. Algorithms for verifying the equivalence of logic-level descriptions and layout (layout verification) can be found in [27, 29, 28], and algorithms for verifying timing behavior can be found in [110].

A *decision problem* [61] is a problem that has a *yes* or a *no* answer. Some decision problems form an equivalent class called non-deterministic polynomial-time complete (NP-complete). The characteristic of these problems is that a solution to any problem can be transformed into a solution to another problem in polynomial time. Also, no known polynomial-time algorithm can solve any of the problems. In addition, these problems are not provably intractable. Many optimization problems can be transformed into decision problems, and the corresponding decision problem can be proved to be NP-complete (*e.g.*, the Traveling Salesman Problem). There is another class of decision problems that are not NP-complete, but are at least as hard as NP-complete problems. Such problems are called NP-hard.

The logic verification problem can be stated in the following manner. Given two circuits $\mathcal{A}$ and $\mathcal{B}$, are $\mathcal{A}$ and $\mathcal{B}$ identical, *i.e.*, under all input conditions, do $\mathcal{A}$ and $\mathcal{B}$ produce the same outputs, and if not, then under what conditions do $\mathcal{A}$ and $\mathcal{B}$ produce different outputs ? This is a decision problem and can be proved to be NP-hard. Despite its complexity, it is possible to verify a large class of practical circuits (*e.g.*, [30, 36, 46, 94]). In this dissertation, the problem of verification of logic-level sequential circuits is examined, new approaches are developed, and their merits and demerits are evaluated in the light of previous approaches.

1.3 Testing

Implementation verification ensures that the design process is error free. However, manufacturing defects are almost invariably introduced. No manufacturing process can guarantee 100% yield, and therefore some manufactured circuits are bound to have defects. The types of defects depend on the technology. Across various technologies, the most common types of defect during manufacturing are short-circuits, open-circuits, open bonds, open interconnections, bulk shorts, shorts due to scratches, shorts through dielectric, pin shorts, cracks, and missing transistors [21]. Also, the larger the circuit in terms of area, the greater the chances of it having a defect. It is necessary to separate the bad circuits from the good ones after manufacture. From the point of view of economics, it has been shown that the cost of detecting a faulty component is lowest before the component is packaged and becomes a part of a larger system. Therefore testing is a very important aspect of any VLSI manufacturing system.

There are two aspects to testing; one is fault detection and the other is fault diagnosis. In fault detection only the presence of a fault is detected, but in diagnosis the exact location of the fault has to be identified. The testing process involves the application of test patterns to the circuit and comparing the response of the circuit with a pre-computed expected response. Any discrepancy constitutes an error, the cause of which is said to be a *physical fault* [21]. Such faults, for digital circuits, can be classified as *logic* or *parametric* [21]. A logic fault is one which causes the logic function of the circuit element (elements) or an input signal to be changed to some other function. Parametric faults alter the magnitude of a circuit parameter causing a change in some factor such as circuit speed, current, or voltage levels. In this dissertation the focus is only on the detection of logic faults.

Testing must be performed throughout the life of a circuit, since faults may be introduced in the circuit during assembly, storage, and service. The most commonly occurring faults during storage and service are due to temperature, humidity, aging, vibration, and voltage or current stress [21].

Generation of test patterns is a very important problem and has been under investigation for a long time [1, 60, 65, 84, 90, 95, 108, 119, 123]. As shown in Figure 1.1, test generation may be performed at various levels during the design process. The average case complexity of test generation, the fault model, and the fault coverage obtained depends on the representation used. An important issue is the fault model used in test generation. Physical faults are often modeled as logic faults. By doing so, the problem of fault analysis becomes a technology-independent logic problem. In addition, tests derived for logic faults may be useful for physical faults whose effect on circuit behavior is not well understood or too complex to be analyzed otherwise. The main requirement for the fault model is that the model should be able to capture the change in functionality caused by most of the commonly occurring physical defects in the circuit. The fault model used most often in practice today is the *single stuck-at* fault model, where a single gate input or output in the circuit gets stuck at a 1 or 0 value.

Another important issue in test pattern generation is fault diagnosis. It is not only important to identify the presence of a fault, but also to locate the fault and find a reason for the fault. Fault location, which is one aspect of fault diagnosis, is used to debug circuits and fix manufacturing errors. Tests for one fault can simultaneously detect other faults in the circuit. Two faulty circuits might also have identical responses for a particular test pattern. Therefore a test set has to be derived which not only identifies all the faults, but can also help in locating the fault from the analysis of the response.

Tests are applied to circuits using Automatic Test Equipment (ATE). This equipment is usually very expensive. The amount of time that each circuit requires for testing is therefore very important. This time is determined by the amount of time required to apply the test vectors and the time required to compare the data with the expected response.

Test generation for combinational circuits as well as sequential circuits is an NP-hard problem [85]. There is no known polynomial-time algorithm that can be used to generate tests for circuits. However, for a large class of circuits, it is possible to generate tests efficiently using various heuristic search techniques.

The problem of test generation is intimately related to the problem of verification. In fact, for both combinational and sequential circuits, the problem of test generation can be formulated as a verification problem and vice versa. The test generation problem can be converted into the problem of verifying whether the fault-free and faulty circuits are identical. If they are not, the test pattern is the (sequence of) input vector(s) that differentiates the two circuits. On the other hand, the problem of verification can be converted into finding a test for a fault, assuming one circuit is a true circuit and the other circuit is a faulty circuit. If no tests can be found, the circuits are identical.

In this dissertation, two new test generation algorithms are presented. The first one uses purely gate-level information to generate tests for sequential circuits. Using better techniques and heuristics, the algorithm can handle larger circuits than previous approaches. The second algorithm uses RTL information for test generation and can be applied to circuits that cannot be handled using a purely gate-level approach.

1.4 Synthesis For Testability

Generating tests for sequential circuits is a difficult problem. The algorithms that will be presented in Chapters 2 and 3 provide means for handling large circuits. However, some circuits have redundant faults, and while generating tests for such circuits, a significant fraction of the test generation time could be spent in the identification of redundant faults. The task of test generation will be greatly simplified if circuits are synthesized to be fully and easily testable.

It is well known that optimal logic synthesis can produce fully testable combinational circuits [13]. Similar results were derived for sequential circuits in [49]. These methods rely on extracting a set of don't-care conditions for the circuit and using the don't-cares during logic optimization to derive a fully testable implementation of the circuit. The primary issue in this approach to synthesis for testability is the efficient derivation of the don't-care conditions. For sequential circuits, these don't-cares have been traditionally obtained from the State Transition Graph [49]. Recently, State Transition Graph traversal techniques based on Binary Decision Diagrams have been used to derive the set of don't-cares [87]. Another major issue is the exploitation of these don't-cares in logic synthesis. As has been shown in [18], traditional don't-cares cannot be used to capture all forms of incomplete specification. The solution to this is the use of Boolean relations [18] in synthesis. In this dissertation, both of these issues are addressed. An algorithm for efficient determination of the don't-cares needed for synthesizing a fully testable machine is presented. In addition, a logic partitioning scheme is outlined that can help in reducing the size of the don't-care set. Also, a procedure that uses testing techniques for the minimization of Boolean relations is presented.

1.5 Organization of this Dissertation

In Chapter 2, a test generation algorithm for sequential circuits specified at the gate level is presented. This algorithm is extended to consider circuits described at the RT level in Chapter 3. In Chapter 4, algorithms for synthesizing circuits to ensure full testability are presented. Chapter 5 deals with the verification of sequential circuits. Efficient algorithms for traversal and enumeration of State Transition Graphs using techniques similar to those in Chapter 2 are presented. In Chapter 6, the use of compact STGs for synthesis is indicated. In Chapter 7, the problem of minimization of Boolean Relations arising in synthesis for testability is addressed. Conclusions are presented in Chapter 8.

Chapter 2

TEST GENERATION FOR SEQUENTIAL CIRCUITS

Test generation for sequential circuits has long been recognized as a difficult problem [21, 75, 102]. In particular, unstructured, random, sequential digital designs are very difficult to test. The test generation problem is difficult mainly because the input space that must be searched to obtain a test vector sequence is huge. Since most circuits have a large number of inputs, it is not possible to enumerate the input space explicitly. The structure of the circuit must be exploited to search the input space implicitly and reduce the complexity of the search process for the expected cases. Another difficulty in circuits without reset lines is the necessity of initializing the latches to a known value. One common approach to improving the testability of the circuit is to add extra test points to improve the controllability and observability of internal nodes of the circuit, thereby easing the task of test generation (*e.g.*, [135]). However, this method is not systematic and relies greatly on the ingenuity of the designer. This approach also has its associated area and performance penalty.

The problem of test generation for combinational circuits is NP-hard. However, by exploiting the structure of logic functions and using the power of modern computers, it is possible to generate tests for most combinational circuits. Therefore, a popular approach to solving the testability problem for sequential circuits has been the Scan Design methodology [2, 56]. In this approach all the memory elements of the machine are made scannable, *i.e.*, their values can be directly controlled and observed. The problem of sequential test generation is thereby converted into the problem of combinational test generation. Though this eases the task of test generation greatly, two major drawbacks of this approach are the associated area and performance penalties. The latches are special Scan latches and are usually larger than normal non-scan latches. The increased area may lead to either degraded performance or reduced circuit yield. The specific penalty that is most important is very technology (*e.g.*, CMOS or bipolar) and design style (*e.g.*, static or dynamic) specific. Also, for circuits with a large number of latches, the total time required to test the circuit using Scan is large. Since tester time is costly, this translates into higher cost for the manufactured product, often doubling the cost of a complex chip. Scan design also constrains the design to be fully synchronous and free of critical races in the normal operation of the circuit. In situations where there is a strict area and power budget, Scan design cannot be used. Even though the problem of non-scan sequential test generation is difficult, there are a large number of circuits for which non-scan test generation can be quite effective. Therefore adopting the Scan design methodology without investigating non-scan test generation may incur unnecessary area and performance penalties.

The rest of this chapter is organized as follows. In Section 2.1, the problem of test generation for non-scan sequential circuits is examined and some basic definitions are presented. In Section 2.2, previous work in this area is reviewed and their merits and disadvantages are pointed out. In Section 2.3, the overall test generation strategy is outlined. Section 2.4 deals with cover extraction and combinational test generation. In Section 2.5, the justification procedure is presented, followed by the differentiation procedure in Section 2.6. Identification of redundant faults is the topic of Section 2.7. Finally, results using this test pattern generation strategy are presented in Section 2.8. Conclusions are presented in Section 2.9.

2.1 Preliminaries

A sequential circuit is a circuit with memory [83]. Such circuits are capable of storing information and performing some mathematical or logical operations upon the stored information. A ***finite state machine*** is an abstract model describing the behavior of a sequential circuit. Formally, a finite state machine $\mathcal{M}$ is defined as a 5-tuple [83]:

$$\mathcal{M} = (\mathcal{I}, \mathcal{S}, \mathcal{O}, \delta, \gamma)$$

where $\mathcal{I}$ is a finite, non-empty set of input symbols, $\mathcal{S}$ is a finite, non-empty set of states, $\mathcal{O}$ is a finite, non-empty set of output symbols, $\delta : \mathcal{I} \times \mathcal{S} \to \mathcal{S}$ is the next-state function, and γ is the output function. For a Mealy machine [83], $\gamma : \mathcal{I} \times \mathcal{S} \to \mathcal{O}$, while for a Moore machine [83], $\gamma : \mathcal{S} \to \mathcal{O}$. Note that a Mealy machine is more general than a Moore machine, and any Moore machine can be converted into a Mealy machine with the same number of states and state transitions. The algorithms described in this dissertation are applicable to both Mealy and Moore machines.

Sequential circuits are often referred to as finite state machines, or simply *machines*. A sequential circuit is said to be synchronous if the internal state of the machine changes at specific instants of time as governed by a clock. A general synchronous sequential circuit at the logic level is shown in Figure 2.1. It consists of a combinational logic block and state registers (latches or flip-flops) that hold the state information. The combinational logic block is an interconnection of gates that implements the mapping between the primary input (PI) and present-state (PS), and primary output (PO) and next-state (NS). The behavior of this circuit is often represented using a State Transition Table (STT) or a State Transition Graph (STG) [83], as in Figure 2.2. The names synchronous sequential circuit, finite state machine, and machine will be used interchangeably throughout this dissertation. Most of the remaining definitions in this section are taken from [49, 63, 83].

In general, a **state** is a symbol indicating the internal state of the machine. For the circuit of Figure 2.1, a **state** is a bit vector of length equal to the number of memory elements (latches or flip-flops) in the sequential circuit. Each state has a unique bit vector representing that state, and this bit vector is known as the *state code*. The process of assigning a code to each state is known as *state assignment* or *state encoding*. A state with only 0s and 1s as bit values is called a **minterm** state. In general, a state could be a **cube**, *i.e.*, the values in the different bit positions may be 0, 1 or $-$ (don't-care). A cube state therefore represents a group of minterm states.

A state is said to **cover** another state if the value of each bit position in the first state is either a $-$ (don't-care) or is equal to the value of the corresponding bit position in the second state. For example, state 110 is covered by state 11$-$. State 11$-$ also covers state 111. State Transition Graph **enumeration** is the process of deriving the STG description of a machine from its logic-level description. There are various methods for doing this and one such method will be presented in this chapter.

To illustrate the problem of test generation, consider a sequential circuit, like the

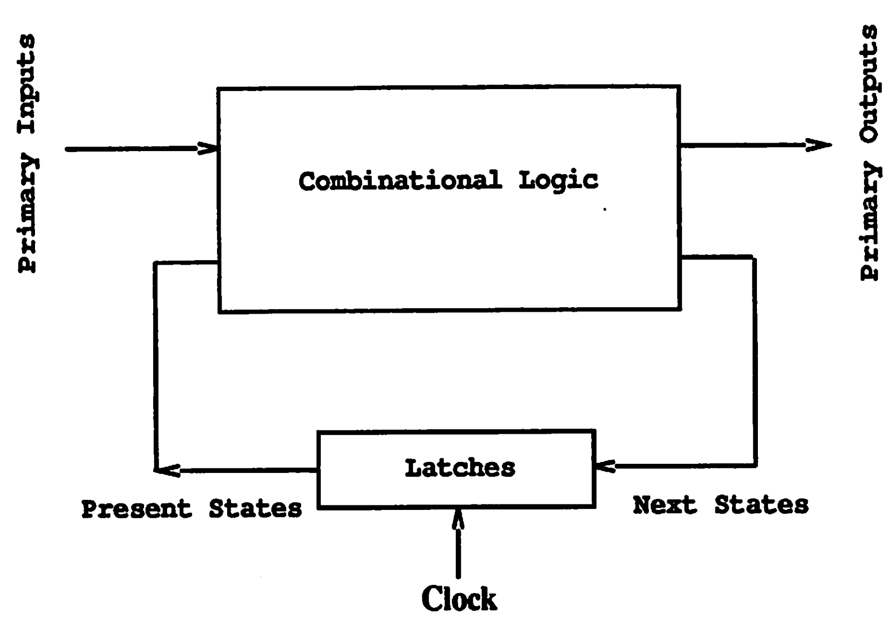


Figure 2.1: A general synchronous sequential circuit

one shown in Figure 2.1, whose STG is shown in Figure 2.2. Assume that the present-state and the next-state lines are neither directly *controllable* nor *observable*. The objective of test generation is to find a sequence of vectors that when applied to the primary input, will produce different responses in the correct and the faulty circuit. A fault often manifests its effect by changing the STG of the machine. Consider a fault in the circuit that changes the behavior of the circuit from that shown in Figure 2.2 to that shown in Figure 2.3. The faulty STG is characterized by corrupted edges. An edge in the State Transition Graph is said to be **corrupted** by a fault if a different output is asserted by the edge or the faulty machine goes to a different next state. Therefore the effect of the fault can be propagated to the primary outputs or next-state lines by the input vector corresponding to the corrupted edge, with the present-state lines set to the fanin state of the edge. An input vector to the combinational logic part of the faulty machine that excites and propagates the effect of the fault either to the primary output or the next-state lines is called an **excitation vector** for the fault. The present-state part of the excitation vector is the state whose fanout edge is corrupted by the fault and is called the **excitation state.** In Figure 2.3, the corrupted

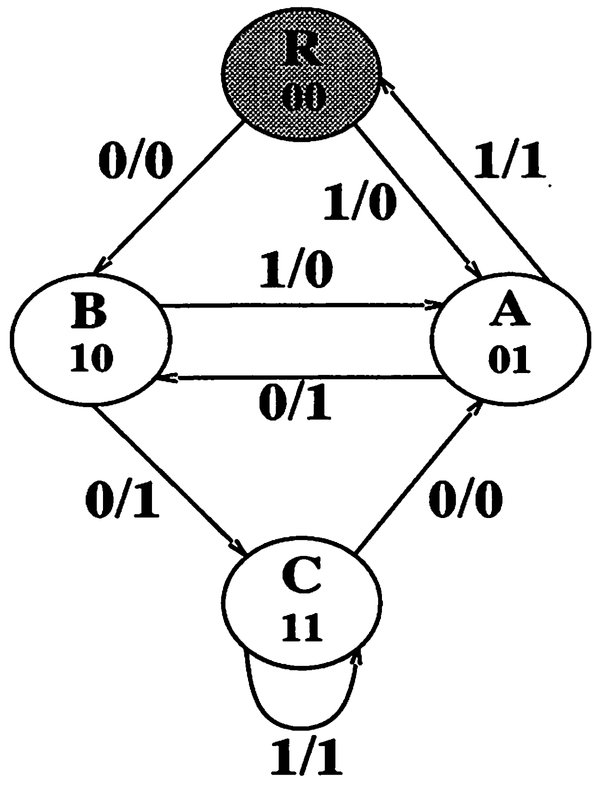
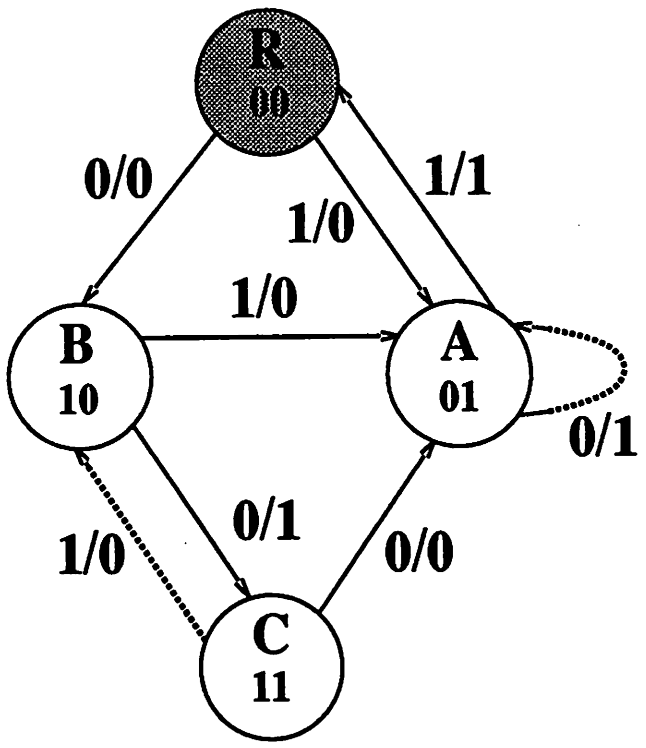


Figure 2.2: An example State Transition Graph

edges are shown using dotted lines. The excitation states are states $\mathcal{A}$ and $\mathcal{C}$. Note that the excitation vector is a combinational test for the fault.

Most sequential circuits have a starting state, the state to which the machine goes upon power-up. Often, the machine can be set to this state very easily (for example, by using a reset line). The starting state of the machine is called its **reset state.** All test vector sequences are applied with the machine starting in the reset state. For the machine used in this example, state $\mathcal{R}$ is the reset state. Throughout this dissertation the reset state will be indicated by a shaded circle in the figures. All states in the machine reachable from the reset state using an input sequence of any length, and the reset state are said to be valid states.

Having determined the excitation state, the next step in test generation is to take



Excitation State C
J = 0, 0
E = 1
Test = J + E = 0, 0, 1

Excitation State A
J = 1
E = 0
D = 1
Test = J + E + D = 1, 0, 1

Figure 2.3: STG of faulty machine

the machine from the reset state to the excitation state. The process of finding an input sequence which takes the machine from the reset state to the excitation state is called **state justification**. The corresponding input sequence is called the **justification sequence**, and the set of states and edges traversed during justification constitute the **justification path**. State justification may be **forward** or **backward**, depending on whether the search is conducted from the reset state to the excitation state or vice versa. A justification sequence for state C is the input vector sequence $\{0, 0\}$ which takes the machine from state $\mathcal{R}$ through state $\mathcal{B}$ to $\mathcal{C}$. A justification sequence for state $\mathcal{A}$ is $\{1\}$.

If state $\mathcal{C}$ is used as the excitation state, then on applying the input vector 1 (the excitation vector) with the machine in state $\mathcal{C}$, the true (or fault-free) machine asserts the output 1 while the faulty machine asserts the output 0. Therefore the effect of the fault is observable at the POs, and a test sequence can be formed by concatenating the justification sequence and the excitation vector. In this case, the test sequence is $\{0, 0, 1\}$. The situation is different if state $\mathcal{A}$ is used as the excitation state. The true and faulty circuits assert the output 1 in state $\mathcal{A}$ on application of the excitation vector 0. However, the true machine goes to state $\mathcal{B}$, while the faulty machine stays in state $\mathcal{A}$. States $\mathcal{B}$ and $\mathcal{A}$ are called the **true/faulty** or **fault-free/faulty state pair**. In order to propagate the effect of the fault to the POs, it is necessary to differentiate between state $\mathcal{B}$ in the true machine and the state $\mathcal{A}$ in the faulty machine. A **differentiating sequence** for a pair of states (S_1, S_2) in a sequential circuit is a sequence of input vectors, such that if the sequence is applied to the circuit when the circuit is initially in S_1, the last vector in the sequence produces a different primary output combination than if the circuit were initially in S_2. The process of finding such a sequence is called **state differentiation**. In this example, a differentiating sequence is the single vector $\{1\}$, where state $\mathcal{B}$ in the fault-free machine asserts the output 0 while the state $\mathcal{A}$ in the faulty machine asserts the output 1. The test sequence is obtained by concatenating the justification sequence, the excitation vector, and the differentiating sequence. In this case it is $\{1, 0, 1\}$.

Justification and differentiation are the bottlenecks in test pattern generation for sequential circuits. Determining the justification sequence for a state is conjectured to be an NP-hard problem and so is finding a differentiating sequence for a state pair. For both these steps, the entire STG of the machine might have to be enumerated (especially if the excitation state is not reachable from the reset state or if the true/faulty states are equivalent).

It can be shown that a fault in a general, fully specified, synchronous sequential circuit, like the one shown in Figure 2.1, may require a test sequence of up to $2^{n+1} - 1$ input vectors, where n is the number of memory elements in the machine. To show this, consider the true and faulty machines and their STGs. Each of these STGs can have a maximum of 2^n states. The two STGs can be concatenated, and a test (if it exists) will be the differentiating sequence for the two reset states. Since there can be 2^{n+1} states in the concatenated machine, the longest differentiating sequence can have as many as $2^{n+1} - 1$

vectors [83]. This shows that the search space for sequential test generation is very large. In fact, if there are i primary inputs, then the total search space is 2^{i+n+1}. To add to the complexity of test generation, some faults in the circuit may be **redundant**, *i.e.*, the behavior of the circuit does not change in the presence of the fault. There are two classes of redundancies [90] in a sequential circuit — **combinational redundancies** and **sequential redundancies**. For a **combinationally-redundant** fault (CRF), the effect of the fault cannot be propagated to the primary outputs or the next-state lines, beginning from any state and using any input vector. A **sequentially-redundant** fault (SRF) is a fault which cannot be excited or whose effect cannot be propagated to the primary outputs using any sequence of input vectors starting from the reset state of the machine. Since such faults are very difficult to identify, large amounts of effort can be spent in trying to generate tests for them.

For the combinational logic (Figure 2.1) in the sequential circuit, there are p primary inputs, n present-state and next-state lines, and q primary outputs. The combinational logic implements a multiple-output Boolean function, $f : B^{p+n} \rightarrow B^{q+n}$, where $B = \{0, 1\}$. Each of the primary outputs or next-state lines are single-output functions of $p+n$ variables. B^{p+n} is referred to as the input space and B^{q+n} is referred to as the output space for the function [16]. The **ON-set**, $X_{ON} \subseteq B^{p+n}$, of a primary output or next-state line is the complete set of input values such that the primary output or next-state line is 1. Similarly, the **OFF-set**, $X_{OFF} \subseteq B^{p+n}$, is the complete set of input values for which the corresponding line is 0. The set of cubes C, is said to be a **cover** for a ON-set if $X_{ON} \subseteq C$ and C does not intersect X_{OFF}.

The **fanout** of a gate $\mathcal{G}$ (or a wire) is defined as the set of gates that use the value generated by $\mathcal{G}$ as an input. The **transitive fanout** of $\mathcal{G}$ is defined recursively as follows. If $\mathcal{G}$ is a gate generating only a primary output, then its transitive fanout is the null set. Else, the transitive fanout of $\mathcal{G}$ is the union of the fanouts of $\mathcal{G}$ and the transitive fanout of every element in the fanout of $\mathcal{G}$.

The initial state of the machine before a test vector is applied or right after it is powered up may or may not be known. The fault model used in test generation also varies. The following assumptions are made regarding the sequential circuit to be tested:

1. The machine is assumed to have a **reset state**, R. All test sequences are applied with this state as the starting state.

2. The fault model is assumed to be **single stuck–at**. Since the state justification and differentiation parts of the test generation algorithm are independent of the fault model, the algorithm is not restricted to only this fault model. Other fault models like multiple stuck-at, bridging faults, *etc.*, may be used with minor modifications to the procedure described.

3. The memory elements are considered as distinct logic primitives and faults inside the memory elements are not considered. However, all faults on present-state and next-state lines are considered. In general, these faults model a large fraction of the faults inside the memory elements.

2.2 Previous Work

Initial work in tackling the problem of test generation involved the use of both random [20, 118] and deterministic techniques [95, 97, 108, 123]. In random test pattern generation techniques, a sequence of random patterns is generated and applied to the circuit. In general, all primary inputs to the circuit are excited equally, *i.e.*, the average number of transitions on each primary input is the same over a long period of time. In some random test pattern generators, inputs are assigned different weights and inputs with more weights are exercised more frequently than others [118]. Different distributions can be used for the random number generator from which the random patterns are derived. Though the test generation time is small, the fault coverage obtained is not satisfactory. Also, the number of test vectors required is often too large to be used as a practical test set. Moreover, redundant faults cannot be identified.

Some test generation algorithms were developed that use the iterative array model [21] of the circuit and combinational test generation techniques to generate tests for sequential circuits [95, 97, 108, 123]. All these approaches start with one copy of the combinational logic block of the machine and use combinational test generation algorithms like PODEM [65] or FAN [60] to find a test for the fault. If a test is not found, another copy of the circuit is added and the process continues. The major drawback of this approach is the complexity of finding a test vector in an iterative array model of the circuit. Since the length of the test vector is not known *a priori*, a large amount of effort may be wasted in trying to generate tests in an iterative array model with an inappropriate number of copies of the circuit. In

[21], it was shown that the complexity of the extended D-Algorithm is 4^n where n is the number of latches in the circuit. Though minimum length test vectors are produced, this approach can only be applied to small sequential circuits.

A heuristic, simulation-based test pattern generation algorithm is described in [1]. Starting with an initial vector, the final test sequence is derived using a fault simulator. For each vector applied, a cost function is computed which indicates whether the fault is excited or if the effect of the fault is propagated closer to a primary output. This cost function guides the selection of subsequent test vectors to be applied. The process continues until the fault is detected or a limit is reached. This is a pseudo-random approach and therefore does not guarantee that a test will be found, even when one exists. Also, redundant faults cannot be identified. However, this approach does not require the existence of a reset state. The simulation-based algorithm can find input vector sequences that set the required latches to a known value. However, there is a danger in not assuming a reset state for a certain class of circuits. Figure 2.4 shows a machine whose latches cannot be initialized using the simulation technique of CONTEST. In such a case, CONTEST would indicate that 100% of the faults in the circuit are untestable. In real life, the designer would ensure that the machine starts in one of the valid states, and assuming any of the valid states as reset state, 100% fault coverage can be obtained. It has also been shown [34] that even if a machine has a synchronizing sequence, then a fault may prevent the initialization of some latches in the circuit making the task of test generation difficult, if not impossible.

Recently, there has been considerable progress in this area. A PODEM-based deterministic approach to sequential test generation is described in [90]. This approach uses the iterative array model for fault excitation and propagation and makes intelligent use of a partial State Transition Graph (STG) of the circuit while generating justification sequences for the faults under test. Before test generation, the STG of the fault-free machine is extracted using the STG enumeration technique of [46]. The STG is used for finding justification sequences for the excitation states. Fault excitation and propagation is performed using the iterative array model of the circuit, considering the effect of the fault in each time frame. This approach can test circuits more efficiently than the approaches described previously in this section. However, this approach is not applicable to circuits with more than 100 latches. For such circuits, only a partial STG can be generated. Since the excitation states for the faults are not known *a priori*, the parts of the STG to be enumerated are not known. Therefore, a significant number of excitation states might not be justifiable.

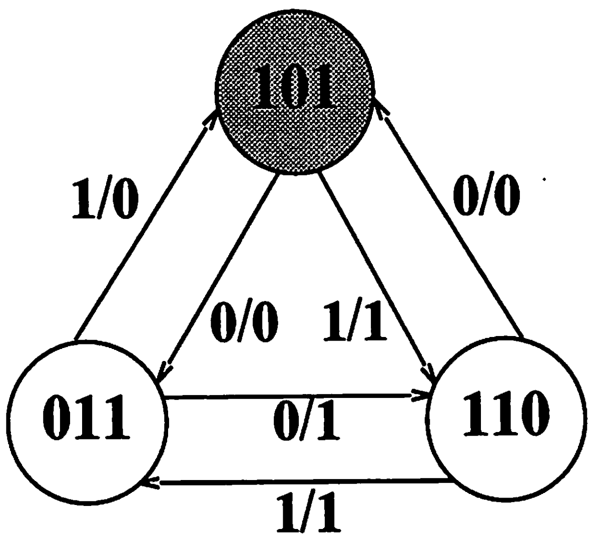


Figure 2.4: A machine illustrating problem with initialization

2.3 Test Generation Strategy

Three major characteristics of the test generation algorithm to be described distinguish it from previous approaches. They are:

- Decomposition of the testing problem into *three* subproblems of derivation of the excitation state, state justification, and state differentiation. As will be seen later, this decomposition makes the reuse of information possible, thereby improving the efficiency of test generation.
- *Fault-free* state justification and differentiation.
- *Selective* enumeration of the state transition graph, as required for test generation.

In this section the overall test pattern generation strategy that uses these ideas is presented.

Information about state transitions in sequential machines is traditionally represented using State Transition Graphs. It is also possible to represent this information using the ON-sets and OFF-sets of all the next-state lines and the primary outputs. *Connectivity* (*i.e.*, connectivity between states) can be represented by the ON and OFF-sets of the next-state lines only.

The first step in test generation is the enumeration of the partial or complete (memory and CPU time permitting) ON and OFF-sets of each next-state line and primary output of the sequential circuit to be tested. Cover enumeration can be performed using a PODEM-based [65] or D-Algorithm-based [115] enumeration algorithm. A limit on the number of cubes in each ON or OFF-set can be placed. This limit is used to restrict the amount of memory and CPU time used for enumeration. Cover enumeration is fast and full covers of moderately large circuits can be extracted easily.

Given the complete (or partial) covers, test generation is a three-step process. These three steps are:

1. Derivation of the excitation state for a fault using combinational test generation, treating the present-state (PS) lines also as primary inputs (PIs) and the NS lines also as POs.
2. State justification.
3. State differentiation.

In theory, the justification sequence should be derived using the faulty machine. Also, the differentiating sequence should be derived using the fault-free and the faulty machines. It has been observed that a fault typically modifies a few edges in the State Transition Graph. If a justification or differentiating sequence is derived using the fault-free machine, the probability of that sequence being valid in the faulty machine is high. Therefore, the *fault-free state justification and differentiation* heuristic is used. There are several advantages of using this heuristic. Firstly, covers of the fault-free circuit can be used to generate these sequences on demand, and covers do not have to be extracted for each fault. Secondly, parts of the justification and differentiating sequence can be reused, saving time. Consider the examples used to illustrate the problem of test generation in Figures 2.2 and 2.3. The fault-free justification sequences derived for state $\mathcal{A}$ and $\mathcal{C}$ are $\{1\}$ and $\{0, 0\}$ respectively and both are valid in the faulty machine. A fault-free differentiating sequence between state $\mathcal{B}$ and $\mathcal{A}$ is $\{1\}$, and it is also valid in the faulty machine. Of course, there are faults that modify the STG drastically, but test generation for such faults is relatively simple.

After combinational test generation, the excitation vector is examined to see if the present-state part of the excitation vector covers the reset state. If the excitation state covers the reset state, then the fault can be excited from the reset state of the machine. If not, the excitation state is justified using a backward justification algorithm. Backward justification is performed by first finding all the fanin states of the excitation state (states that have an edge in the STG going from them to the excitation state) using repeated cube intersections. If the reset state is a member of the set of fanin states, then a one vector justification sequence is found. Otherwise, the process is repeated for some state in the fanin of the state being currently justified. Once a justification sequence is found, it is fault simulated to see if the required state is justified. If the required state is justified, then the justification sequence is a valid justification sequence in the faulty machine. If the required state is not justified, then some edge in the justification path must have been corrupted. A part of the justification sequence can then be used as a justification sequence for the state whose fanout edge was corrupted by the fault. State differentiation can then be performed between the corresponding true and faulty states. Thus, only one fault-free justification has to be performed to obtain a true and faulty state pair (*cf.* Section 2.5).

If the effect of the fault under test has been propagated to the primary outputs by the combinational test vector and if the excitation state can be justified in the faulty machine, then a successful test for the fault has been generated. If, however, the effect of the fault is propagated only to the next-state lines, then the fault-effect has to be propagated to some primary output by state differentiation. This is performed by first finding an input vector that produces a different output on at least one primary output line for the true and the faulty states. Such a vector constitutes a single-vector differentiating sequence between the true and faulty states. If a single-vector differentiating sequence cannot be found, all the fanout states of the true and faulty states are found using repeated cube intersections. Then, for each pair of fanout states a single-vector differentiating sequence is sought. If no such pair exists, then a pair of states fanning out from some fanout state pair is picked and differentiation between this pair is attempted. This differentiating sequence obtained is valid under fault-free conditions. After the differentiating sequence is obtained, the entire test sequence is fault simulated to see if the fault under test is detected. As shown later, experimental evidence gathered to date indicates that most of the time, the sequence generated is actually a test for the fault.

If a test is not valid for the fault under consideration, then the fault-free differentiating sequence is not a valid differentiating sequence under faulty conditions. Sometimes a fault-free differentiating sequence might not be obtainable because the true and faulty states are equivalent in the true machine. For such rare cases, the covers of the ON and OFF sets of the PO and NS lines are extracted assuming that the fault is present in the circuit. Differentiation is now attempted using the true and the faulty covers of the circuit. The differentiating sequence obtained can be guaranteed to be valid under faulty conditions (*cf.* Section 2.6).

At this point it is necessary to look at the assumption of having a reset state in more detail. The reset state acts as a reference state from which all test vectors are applied. The justification procedure looks for a sequence of vectors that will take the machine from this reference state to the fault excitation state. If there is no such reference state, it might not be possible to justify excitation states by the procedure outlined here. In Section 2.2, the problem of not assuming a reset state for the test generator CONTEST was presented. In addition, to ensure that the machine starts in some proper state, most designers incorporate a reset state. Therefore, the assumption of having a reset state is realistic. This reference state can be produced in a variety of ways. It could be provided as a reset line that sets the machine to a particular state or it could be derived by applying a sequence of vectors (called the synchronizing sequence) to the machine that will take the machine, irrespective of its starting state, to a known state. However, faults can invalidate a synchronizing sequence and make it impossible to generate tests. In general, the reset state can be a cube state, *i.e.*, there might be don't-care entries in the reset state. Such a reset state is interpreted to mean that all the minterm states covered by the cube reset state are valid reset states for the machine, as the machine can be driven to any of those states using a known sequence of input vectors. For a certain class of circuits, like microprocessors, only the latches in the controller are initialized first, and then the other registers in the processor are initialized. These other registers can be set to any value by selecting the right inputs during the initialization phase. For such circuits, the reset state is a cube state.

As in some combinational test generators [20] and in some sequential test generators [118], random test vector generation is used as a front end to the deterministic test generation algorithm. Random vector test generation helps in detecting some of the easy to detect faults without much effort and therefore reduces test generation time. However, unlike combinational circuits where sometimes a large percentage of the faults can be detected using random techniques, for sequential circuits, only a moderate percentage of the faults can be detected.

The test generation algorithm based on the ideas presented above is as follows:

1. Extract the complete or partial covers of the ON-set and OFF-set of primary outputs and next-state lines.

2. Generate a (new) combinational test vector for the fault under test. If test vectors for the fault have already been generated and some of the excitation states were not justifiable, the new test vector has the present-state part disjoint from the present-state part of all such previously generated test vectors. If no new test vector can be found, then exit without a test.

3. Generate a (new) fault-free justification sequence J_S for the excitation state. If no new justification sequence is found, go to Step 2.

4. Fault simulate the potential justification sequence J_S. If it detects the fault, generate a test sequence T_S directly from J_S. If it is a valid justification sequence, then go to Step 5. If it is not a valid justification sequence, then find the *first* state whose fanout edge was corrupted by the fault. Use part of J_S as the justification sequence for that state, and try to differentiate between the new true and faulty state pair.

5. Generate a fault-free differentiating sequence D_S for the true-faulty state pair. Concatenate J_S, the excitation vector, and D_S to obtain the test sequence T_S. Fault simulate T_S. If the fault under test is detected, exit with test vector T_S. If the fault is not detected, try to generate another D_S. If unsuccessful or if D_S is not found, go to Step 6.

6. Extract the faulty cover of the PO and NS lines. Set the unspecified inputs in the justification sequence to 1 or 0 and simulate the circuit to derive a true and faulty state pair. Generate a differentiating sequence D_S under faulty conditions. Concatenate J_S, the excitation vector, and D_S to obtain the test sequence T_S. Fault simulate T_S to find other faults that this sequence detects. T_S is guaranteed to be a test for the fault under consideration. Exit with test vector T_S. If D_S cannot be found and all possible assignments to unspecified inputs have been exhausted, go to Step 3. Else, set the unspecified inputs in the justification sequence to a different value and repeat this step.

By checking for the validity of the justification sequence before state differentiation, the need for generating more than one justification sequence in each pass of the algorithm is obviated. The algorithm is complete, *i.e.*, if a test exists for a fault, then given enough time and memory the algorithm will find it.

Since justification and differentiating sequences are derived using the fault-free machine, a significant fraction of these sequences can be reused and are therefore stored. In effect, parts of the State Transition Graph (STG) that are required for test generation are explicitly enumerated. Enumeration of only the required parts of the STG results in significant memory and CPU time savings over the approach of [90].

An important part of any test pattern generator is the fault simulator used to grade the test patterns. Significant amount of research has been done in this area. To date, there exist efficient algorithms for fault simulation, both for combinational and sequential circuits [6, 88, 99, 107, 120, 130]. Since the focus of this research was the development of efficient test pattern generation algorithms, a simple, parallel, event-driven sequential fault simulator was used to grade the test patterns. More sophisticated fault simulation algorithms like *concurrent fault simulation* can help in speeding up the total time required for test pattern generation.

Finally, though covers are used for justification and differentiation, it is also possible to use alternate representations, like reduced, ordered Binary Decision Diagrams [26], for the implementation of the general principles presented in this section and the three sections that follow.

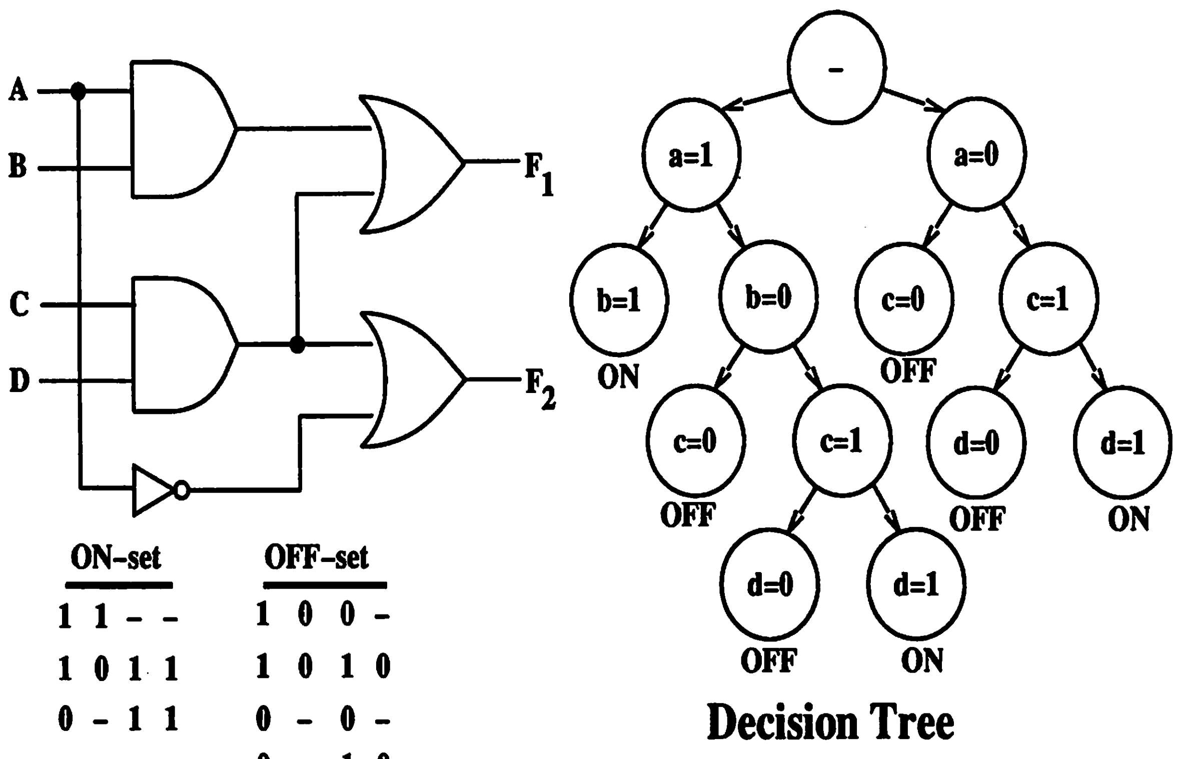
2.4 Cover Extraction and Combinational Test Generation

The input to the test generator is the combinational logic specification of the finite state machine with the latch inputs and outputs properly identified. For each primary output and next-state line, the ON-set and OFF-set are derived by setting the corresponding line to a 1 or 0 and using PODEM [65] to implicitly enumerate the input combinations that can set the line to a 1 or 0. For example, for the circuit shown in Figure 2.5, F_1 is set to 1 and the backtrace procedure of PODEM sets input A to 1 and then input B to 1. Upon simulation, it is found that F_1 is 1 and therefore $11--$ is a cube in the ON-set of F_1. Both the inputs and their values are stored in a decision tree. After the output is set to a value, the last decision in the tree is reversed, *i.e.*, B is assigned the value 0. In order to set F_1, more inputs have to be set. The backtrace procedure identifies input C and sets it to 0. Upon simulation, the value of F_1 is determined to be 0, and therefore

100– is a cube in the OFF-set of F_1. The backtracking and backtracing continues until the decision tree becomes empty. The entire decision tree and the resulting cover is shown in Figure 2.5. Since on every backtrack PODEM sets an input line to a value different from what it had previously, the cover of the ON and OFF-sets are guaranteed to be single cube containment minimal. The algorithm also tries to set as many primary input lines as possible before setting the present-state line to either 0 or 1. For large circuits, where entire covers cannot be enumerated, a partial cover is generated so that different portions of the entire input space are sampled. This is performed by backtracking more than one variable in the decision tree for every N backtracks, where N is usually set according to the estimated size of the cover and the total space allocated to store the cover. For most of the examples tried, complete covers could be generated. For some, complete covers were too large and partial covers had to be generated, but were found to be quite effective for test pattern generation. In most cases, cover enumeration took a small fraction of the total test generation time. However, there exist circuits for which even partial cover generation can take a significant amount of time and memory (*e.g.*, xor trees, combinational multipliers). Covers of the circuit when a fault is present are extracted using the same procedure, except that the faulty line is always assumed to have the corresponding stuck-at value.

As mentioned earlier, for certain circuits, it might not be possible to generate and store the complete covers. In certain cases, even when complete covers can be generated, the user might be interested in using only partial covers because of memory restrictions. For all circuits, a bound is set on the memory and CPU time required to enumerate the covers of each PO and NS line. As soon as the bound is reached, cover extraction for the particular PO or NS is aborted and a new PO or NS line is considered. *Note that the inability to generate complete covers or even a good partial cover will limit the applicability of this approach to very large circuits.*

Given a fault for which a test sequence has to be generated, the first step in sequential test generation is to derive an excitation vector using combinational test generation. The circuit is considered to be combinational with inputs being the PIs and the PS lines and the outputs being the POs and NS lines. A cube test vector is generated having as many don't-care entries in the present-state part as possible, because state justification is easier for states represented by large cubes. Combinational test generation is based on the decision tree concept of the test pattern generation algorithm PODEM. It uses 9-valued simulation as used in [105]. The fault is first excited by setting the faulty wire to a value



ON-set	OFF-set
1 1 - -	1 0 0 -
1 0 1 1	1 0 1 0
0 - 1 1	0 - 0 -
	0 - 1 0

Figure 2.5: Cover enumeration example

different from the faulty value. At first, this value is justified by setting some of the inputs to the combinational logic block. The algorithm then tries to propagate the effect of the fault to the primary outputs, failing that it tries to propagate the effect of the fault to the next-state lines. If the fault is combinationally redundant, then the effect of the fault cannot be propagated to either the NS or PO lines. Since the goal is to generate a maximal cube for the PS lines, all NS and PO lines may not be set to a 1 or 0 at the end of test generation — some lines may still be left unknown.

If a new test has to be generated because the previous excitation state could not be justified, then the new excitation state should be *disjoint* from the previous excitation state, *i.e.*, it should not intersect the previous excitation state. Thus, for a particular fault, each test generated has an excitation state that is disjoint from all previous excitation states

N_1 (ON)	N_1 (OFF)	N_2 (ON)	N_2 (OFF)
0 – 0	1 – 0	1 – 0	0 0 –
0 0 –	1 0 –	– 1 –	– 0 1
1 1 1	0 1 1		

Figure 2.6: ON and OFF-sets for NS lines of example machine

for which justification failed. This is performed by setting a state variable to a different value in the backtracking process.

2.5 Justification

The present-state part of the combinational test vector is the excitation state S_1. Combinational test generation produces a test vector with as many don't-care entries in the present-state part as possible. Therefore, the excitation state is generally a cube state. Any minterm state in the group of states S_1 has to be justified. If the reset state R is already covered by S_1, then the fault can be excited from the reset state and a justification sequence is not needed.

The state justification algorithm first attempts to find a single-vector justification sequence from the reset state R of the machine to any of the states (minterms) in S_1. If complete covers of the next-state lines are available, the entire fanin of S_1 can be found via cube intersections. S_1 is represented as a bit-vector with 0, 1, and – (don't-care) entries. If the position corresponding to a PS line has a 1 (0), the ON-set (OFF-set) of the corresponding NS line is picked. Bit positions with – are ignored. The intersection of the ON and OFF-sets of the NS lines picked gives the fanin edges (both PI and PS vectors) to the states in S_1. The intersection can be computed dynamically, checking each cube, c, produced to see if the PS part of the cube covers the reset state R. If such a cube(s) is found, a single-vector justification sequence from R to S_1 is obtained. The justification sequence corresponds to the PI part of the cube c. In Figure 2.6, the covers of the ON-set and OFF-set for the two NS lines for the machine of Figure 2.2 are shown. In trying to justify state $\mathcal{A}$ (code 01), the OFF-set of the first NS line and ON-set of the second NS line are intersected, and the cubes in the intersection (Figure 2.7) are the fanins of state $\mathcal{A}$. The first and the second cubes in the fanin contain the reset state, and a single-vector justification sequence for state $\mathcal{A}$ is $\{1\}$.

If the reset state is not covered by any state in the fanin, then a single-vector justification sequence does not exist for any of the states in S_1. Thus, an N-vector sequence with $N > 1$ has to be found. This is achieved by heuristically selecting a cube state S_2 which exists in the fanin of S_1 and attempting to justify some minterm state in S_2 using a single-vector justification sequence. The fact that S_1 can be a cube state implies an *implicit* traversal of the STG of the machine.

While inspecting each cube c formed from the intersection of the ON and OFF-sets, the *largest* cube (cube with most don't-care entries) that is *not covered* by S_1 and which is not already in the potential justification path is picked to be S_2. The first condition is required, since if $S_2 \subseteq S_1$, then all states in the fanin of S_2 are also in the fanin of S_1, and a single-vector justification sequence does not exist for S_2. Also, to prevent cycles during justification, the state selected should not be in the potential justification path built so far. For example, to justify state $\mathcal{C}$, the ON-sets of both the NS lines have to be intersected. The fanins of $\mathcal{C}$ contain only two cubes, which are $\{010, 111\}$. Since $\mathcal{C}$ has the code 11, the only cube in the fanin that can be chosen is the cube 010. The state with code 10 is state $\mathcal{B}$, and the justification process can be repeated for state $\mathcal{B}$ giving the final justification sequence $\{0, 0\}$.

In general, the excitation state is a cube state which might contain some valid as well as invalid states. The fanin of the excitation state contains all states that can reach any minterm state covered by the cube state. If the reset state is covered by some state in the fanin, then it means that one or more of the excitation states are reachable from the reset state. The exact state reachable can be found by simulating the justification sequence. Therefore if a justification sequence for a cube state is found, it *does not* mean that all states in the cube state are justifiable. While reusing justification sequences, this fact has to be considered. Whenever a new state has to be justified, it is checked to see if it *covers* any of the states that have been justified. If so, the justification sequence for that state can be used. In fact, the first step in justification is to make this check before generating the justification sequence.

Many primary inputs in the justification sequence are don't-cares. By setting these entries to different 1 or 0 values, different justification sequences to different minterm

$$\begin{array}{ccc} \multicolumn{3}{c}{N_1\ (\text{OFF})} \\ 1 & - & 0 \\ 1 & 0 & - \\ 0 & 1 & 1 \end{array} \quad \cap \quad \begin{array}{ccc} \multicolumn{3}{c}{N_2\ (\text{ON})} \\ 1 & - & 0 \\ - & 1 & - \\ & & \end{array} \quad = \quad \begin{array}{ccc} \multicolumn{3}{c}{\text{Fanins}} \\ 1 & - & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}$$

Figure 2.7: Justifying state A

excitation states can be found. During test generation, all possible assignments of 1 or 0 to the don't-care inputs have to be looked at, if necessary. In practice, it is seldom necessary to look at more than one or two such assignments.

The process of finding the justification sequence, as illustrated here, is a mixed depth-first/breadth-first method. This is because at each step all fanins of the current state being justified are found. Then one of the cube-states in the fanin is selected and all the fanins for the states contained in the cube-state are found. This method can be easily modified to do a complete depth-first or a complete breadth first search. In the complete depth first search, only one cube in the fanin is generated at a time. Then the corresponding fanin state is justified. This method has the advantage of using less memory, but could produce long test sequences and may take a longer time to find justification sequences. On the other hand, in the complete breadth-first method, at each step the fanin of all the states in the fanin of the excitation state is calculated. This obviously requires more memory, but guarantees the shortest justification sequence. In practice, it has been found that the mixed depth-first/breadth-first procedure produces the best result in terms of test generation time and the length of the test vector.

The justification sequence that is constructed is valid under fault-free conditions because the covers of the fault-free circuit were used to derive it. This sequence may be invalid under faulty conditions. If a justification sequence is invalid under faulty conditions, it means that the effect of the fault has already been propagated to the NS lines or the POs. Empirical evidence gathered to date has shown that over 99% of the time, in real circuits, a justification sequence valid in a fault-free machine is also valid in the faulty machine or is in itself a *test sequence* for the fault. Consider the unlikely event that a justification sequence is

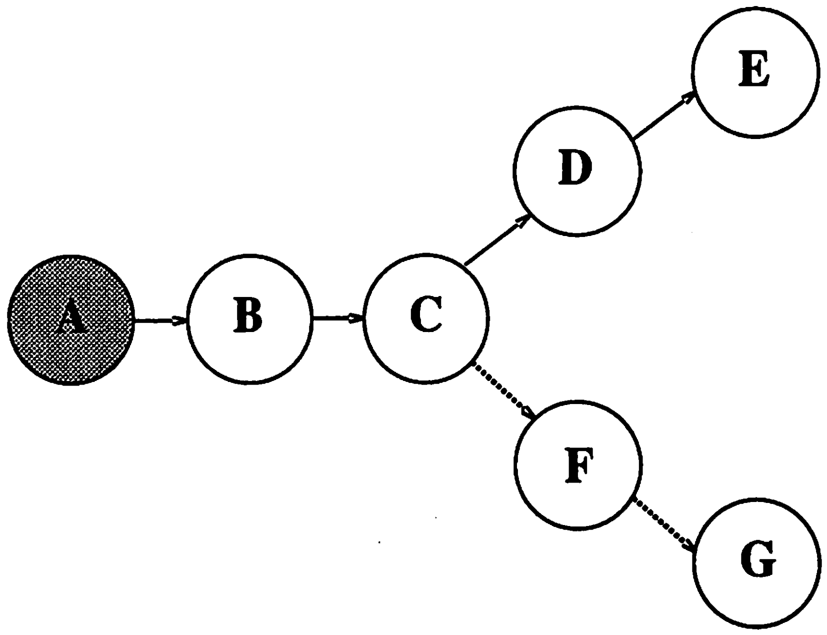


Figure 2.8: Fault-free state justification

neither valid in the faulty machine nor a test sequence in itself. This situation is depicted in Figure 2.8, where the fault-free justification sequence takes the correct machine along path $\mathcal{A} \rightarrow \mathcal{B} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{E}$, but takes the faulty machine along path $\mathcal{A} \rightarrow \mathcal{B} \rightarrow \mathcal{C} \rightarrow \mathcal{F} \rightarrow \mathcal{G}$. The fanout edge of state $\mathcal{C}$ is corrupted, and therefore, a part of the justification sequence, namely, the part that takes the machine from $\mathcal{A}$ to $\mathcal{C}$, is a correct justification sequence. Also states $\mathcal{D}$ and $\mathcal{F}$ correspond to the new true/faulty state pair. The fault simulator is used to derive the correct justification sequence and the true/faulty state pair.

All cubes are represented as bit vectors which makes storage and operations on cubes very efficient. Data structures used for representing cubes and covers are similar to those used in ESPRESSO [16]. Using proper bit notations for 0s, 1s, and –'s, cube intersection can be performed efficiently by bitwise AND operations. The cubes in each cover are ordered so that the cubes that cover the reset state are before those that do not. This helps in finding a fanin state that covers the reset state as early as possible. Cubes are also sorted so that the larger ones are placed before the smaller ones. This helps in finding

```
Justify_State(State)
{
  if (Reset State covered by State)
    return (FOUND);
  /* Check if information can be re-used */
  if (State is already justified)
    return (FOUND);
  /* Fanins is all the fanins of the excitation state */
  Fanins = universal cube;
  for (each PS line that is a 1 or 0){
    Fanins = Fanins ∩ (ON or OFF set of corresponding NS line);
  }
  if (Reset State covered by some state in Fanins)
    return (FOUND);
  /* $P_{JS}$ is the potential justification sequence */
  while (there are still cubes in Fanins){
    /* Select state amongst the state in Fanins so that it is not covered by State
    and so it is not in the potential justification sequence $P_{JS}$ */
    FaninState = select_state();
    if (FaninState is found){
      $P_{JS} = P_{JS} \cup$ FaninState;
      Justify_State(FaninState);
      if (Justification Sequence is found){
        return (FOUND);
      }
      else{
        $P_{JS} = P_{JS} -$ fanin_state;
      }
    }
  }
  return (NOT_FOUND);
}
```

Figure 2.9: Justification algorithm

the largest cube in the fanin easily.

The pseudo-code for the justification algorithm is given in Figure 2.9.

2.6 State Differentiation

This is the third and the final step in test generation and is required if the initial combinational test vector generated for the fault under test propagates the effect of the fault only to the NS lines.

Typically, in sequential test pattern generators, a propagation sequence that propagates the effect of the fault to the POs is found using a test generation algorithm like PODEM on multiple time-frames (or clock cycles) using the iterative-array model [21]. The first vector in this sequence propagates the effect of the fault only to the NS lines. Since the fault is present in each time-frame, propagation from the PS lines of the second time-frame to the POs of the second time-frame is attempted under faulty conditions (to take into account the multiple-fault effect corresponding to the presence of the fault in each time frame).

Since a fault typically modifies a few edges in the STG, a method of *fault-free state differentiation* is used. However, if a fault-free state differentiating sequence is not found or if such a sequence is not a valid sequence under faulty conditions, then state differentiation is performed considering the effect of the fault in all time frames.

In the general case, differentiation is attempted between disjoint groups of states rather than a minterm state pair. The existence of a differentiating sequence between two groups of states means that if *any* state A from the true group is chosen along with a *corresponding* state A' from the faulty group, then the differentiating sequence will be able to differentiate between the states A and A'. Since this is a strong requirement it is often impossible to find a differentiating sequence between the state groups. This does not mean that a test for the fault does not exist. To find a test, it is necessary to set the unspecified inputs in the justification sequence to either 1 or 0 and then fault simulate the sequence to find the true and faulty state pairs. Differentiation is then attempted between the minterm state pairs. All possible assignments to the unspecified inputs have to be made before it can be concluded that a test for the fault under consideration does not exist. At first, an attempt is made to find a fault-free differentiating sequence for the true/faulty state pair. If it is not found, then the faulty covers are extracted and differentiation is once again attempted with the true and faulty covers.

After justification, the true/faulty state pair ($S_1{}^T$, $S_1{}^F$) given by the excitation vector has to be differentiated. In reality, $S_1{}^T$ is a state in the fault-free machine and $S_1{}^F$

N_1 (ON)			N_1 (OFF)		
0	1	0	-	0	0
-	0	1	1	-	0
1	-	1	0	1	1

Figure 2.10: ON and OFF-set for PO of example machine

is a state in the faulty machine. It is assumed initially that ${S_1}^T$ and ${S_1}^F$ are states in the fault-free machine and a differentiating sequence for them is obtained. Note that ${S_1}^T$ and ${S_1}^F$ are guaranteed to differ in at least one bit.

State differentiation can be performed using the complete or partial covers of the POs and the NS lines. The procedure for single-vector differentiation is as follows:

1. Pick a (new) output.

2. Inspect the covers of the ON-set and OFF-set of the output and search for a PI combination, i_1, which appears concatenated with ${S_1}^T$ (or $c \supseteq {S_1}^T$) in the ON-set and concatenated with ${S_1}^F$ (or $c \supseteq {S_1}^F$) in the OFF-set (or vice versa). If such an input combination is found for some output, then a fault-free state differentiating sequence can be constructed. Exit with the input combination. If not, a single-vector state differentiating sequence cannot be found for $({S_1}^T, {S_1}^F)$. If the true and the faulty covers of the circuit are used, then the ON-set of the true machine is used together with the OFF-set of the faulty machine (and vice versa) instead of using the ON-set and OFF-set of the true machine, as outlined above. For example, consider the machine shown in Figure 2.2. One true/faulty state pair is the pair $\{\mathcal{B}, \mathcal{A}\}$. The ON and OFF-set of the only output are shown in Figure 2.10. The second cube in the ON-set of the output contains state $\mathcal{A}$ and the second cube in the OFF-set of the output contains state $\mathcal{B}$. Also, the cubes have a common input minterm, which is 1. Therefore a single-vector differentiating sequence can be found between $\mathcal{A}$ and $\mathcal{B}$.

Multiple-vector differentiating sequences can be searched for in the following fashion. N is the number of vectors in the current sequence. Note that the procedure uses depth-first search.

3. $N = 1$.

4. Pick a NS line and attempt to find a PI vector (using the method outlined in Step 2 above), i_N, that produces a 1 (0) when concatenated with $S_N{}^T$ and a 0 (1) when concatenated with $S_N{}^F$. Try another NS line if a vector cannot be found for the one that was picked. If an input combination cannot be found for any such NS line, then a differentiating sequence cannot be found for $(S_N{}^T,\ S_N{}^F)$.

5. Find by simulation or cube intersections the state pair $(S_{N+1}{}^T,\ S_{N+1}{}^F)$ which is the fanout state pair of the state pair $(S_N{}^T,\ S_N{}^F)$ for the PI vector i_N. Now set $N = N + 1$. Attempt to find a single-vector propagation sequence for the state pair $(S_{N+1}{}^T,\ S_{N+1}{}^F)$.

The procedure attempts to find a single-vector sequence, then a two-vector sequence, and so on. The NS lines are selected in a heuristic order that uses topological information about the location of the fault with respect to the different NS lines and POs. Only NS lines that are in the transitive fanout-cone of the line on which the fault resides are chosen. Also, the NS lines that are *closest* to the fault are chosen first. The rationale behind this choice is that if the NS line is close to the faulty line, it will be easier to propagate the effect of the fault to that NS line.

Since the justification and differentiation procedures are based on cube intersections, the cover sizes determine the time required for justification and differentiation. For circuits with very large covers, justification and differentiation can take a long time. Also, the mixed depth-first/breadth-first approach usually produces test vectors that are longer than the minimal length test vector that can be obtained for a fault.

2.7 Identification of Redundant Faults

The difficulty in sequential test generation lies not only in the generation of tests for difficult-to-detect but testable faults, but also in the identification of redundant faults. Low fault coverage on certain circuits does not necessarily mean that the test generator is inadequate, if it can be shown that the fault coverage is close to the maximum possible value. In general, identification of redundant faults may require astronomical amounts of CPU time as the total input and state space has to be enumerated before a fault can be pronounced redundant. As defined in Section 2.1, there are two classes of redundant faults –

combinationally-redundant and sequentially-redundant. Combinationally-redundant faults are detected during combinational test generation and are easier to find than sequentially-redundant faults. Sequentially-redundant faults (SRFs) can be divided into **invalid-SRFs**, **equivalent-SRFs** and **isomorph-SRFs**. Detailed definition and examples of these faults are provided in Chapter 4. Brief definitions are presented here. A fault is said to be an invalid-SRF if for all possible combinational tests for the fault, none of the excitation states are valid. Similarly, a fault is said to be an equivalent-SRF if none of the true and faulty state pairs have a differentiating sequence. In the third kind of sequentially-redundant fault, namely, isomorph-SRF, the STG of the faulty machine is isomorphic to the STG of the true machine (*cf.* Chapter 4).

It is possible to detect a subset of the sequentially-redundant faults (namely the invalid-SRFs) using **Theorem 1** of [90]. This theorem states that if all excitation states for a fault are invalid states in the fault-free machine, then the fault is sequentially redundant. The same theorem is used here for the detection of invalid-SRFs. However, the conditions of **Theorem 1** are verified differently. A state (or group of states) cannot be justified if the total number of fanin cubes determined during the justification procedure is zero or if all the fanin states of the state are unjustifiable. Also, a group of states is unjustifiable if the fanin states of all states in the group are states within the group, and the reset state is not a member of the group. Such groups of states can be identified during the justification procedure. An unjustifiable state is a state that cannot be reached in the true machine (invalid state). All possible combinational tests for a fault are generated with disjoint excitation states. If all the excitation states are unjustifiable, then the fault under test is redundant. If complete covers are available, then given enough time, all unjustifiable states in the machine can be detected. Even when complete covers of every next-state line are not available, if the covers necessary to find the justification sequence are complete, it might be possible to establish sequential redundancies. In the next section it is shown that it is possible to establish redundancy for a larger fraction of undetected faults using the approach outlined above as opposed to the method of [90].

It is also possible to detect equivalent and isomorph-SRFs. If a fault-free differentiating sequence is not valid or is not found, then the test generator extracts the faulty cover of the circuit and attempts to find a differentiating sequence using the true and the faulty covers of the circuit. This is performed using the procedure outlined in Section 2.6. If a differentiating sequence cannot be found using this procedure, then it can be concluded

CKT	#Inp	#Out	#Gate	#Lat	#Faults
cse	7	7	192	4	519
sse	7	7	130	6	368
planet	7	19	606	6	1482
sand	9	6	555	6	1425
scf	27	54	959	8	2456
mult4	9	9	170	15	336
sbc	40	56	1011	28	1985
stage	131	64	2700	64	7227
key	62	48	1342	56	8520
pewd	117	101	1873	112	3483
dsip	228	197	3654	224	6781

Table 2.1: Statistics for example circuits

that the true and faulty state pairs are not differentiable. If for all possible combinational tests for the fault the true and faulty state pairs are not differentiable, then the fault is sequentially-redundant. Identification of such redundancies is potentially very expensive, as it might require the enumeration of the entire STG of the true and the faulty machines.

2.8 Test Generation Results Using STEED

The test generation algorithm described in the previous sections has been implemented in the program STEED. It consists of about 10,000 lines of C code and runs in a VAX-UNIX^{TM} environment.

Results and time profiles using STEED for eleven finite state machines that are described in Table 2.1 are given in Tables 2.2 and 2.3 respectively. In Table 2.4, comparisons are made with STALLION [90] and the test generator CONTEST described in [1]. In the tables m and s stands for minutes and seconds, respectively. For each example in Table 2.1, the number of inputs (#Inp), number of outputs (#Out), number of gates (#Gate), number of latches (#Lat), and number of non-equivalent faults for which tests have to be generated(#Faults) are shown.

The first five examples are finite state machines obtained from various industrial sources. The example **sbc** is a snooping bus controller in the SPUR chip set. Examples **stage** and **key** are finite state machines in a data encryption chip. For the examples **dsip** and **key**, which have extremely large STGs, STALLION is unable to establish sequential

CKT	#Test Seq.	#Vec	Max. Seq. Len.	Success (%)	Fault Cov. (%)	Red. fault (%)	TFC (%)	Mem (Kb)	TPG time
cse	83	397	9	98.0	99.61	0.39	100.0	1.2	29.5s
sse	52	336	11	100.0	80.98	19.02	100.0	0.8	25.1s
planet	57	1046	28	100.0	96.56	3.44	100.0	2.7	5.85m
sand	109	722	33	95.0	92.77	6.66	99.43	7.6	7.77m
scf	164	2400	24	95.0	93.81	5.00	98.81	18.1	57.58m
mult4	9	94	46	100.0	98.51	1.19	99.70	3.3	11.9s
sbc	102	1364	31	95.0	95.42	3.22	98.64	118.7	75m
stage	58	155	19	100.0	92.36	7.64	100.0	3328.0	50.87m
key	454	1669	17	100.0	94.75	5.25	100.0	44.7	419m
pewd	11	221	28	100.0	100.0	0	100.0	108.1	6.58m
dsip	8	212	39	100.0	99.99	0.01	100.0	404.0	25.8m

Table 2.2: Test generation results for circuits

CKT	Cover Enum.	Justify	Differ	Test Generation	Fault Simulation	Miscell. Set up	Total Time
cse	0.6s	0.6s	0.7s	4.4s	24.3s	0.2s	29.5s
sse	0.3s	0.8s	0.3s	10.7s	13.9s	0.2s	25.1s
planet	1.9s	3.0s	3.0s	36.9s	311.9s	0.7s	5.85m
sand	3.5s	8.8s	48.8s	4.54m	3.16m	0.7s	7.77m
scf	8.7s	2.29m	4.28m	10.63m	47.01m	1.2s	57.58m
mult4	1.4s	3.5s	0.01s	4.59s	5.12s	0.2s	11.9s
sbc	1.71m	21m	26m	50m	23m	1.4s	75m
stage	3.38m	15.3m	0.0s	27.3m	20.07m	10.1s	50.8m
key	21.3s	21.27s	174.75s	59.0m	354.6m	6.8s	419m
pewd	31.9s	0.23s	8.0s	9.49s	5.18m	2.3s	6.58m
dsip	120.2s	0.34s	12.8s	14.9s	23.46m	5.1s	25.8m

Table 2.3: Time profiles for example circuits

redundancy for faults but STEED can.

In Table 2.2, the number of test sequences (#Test Seq.), the total number of test vectors (#Vect), the maximum test sequence length (Max. Seq. Len), the percentage of times when a potential test sequence generated for a fault actually detected the fault (%Success), the fault coverage (Fault Cov.), the percentage of provably redundant faults (Red. Fault) (using the redundancy identification procedure), the total fault coverage including detected and provably redundant faults (TFC), the total memory required for storing the covers (Mem) in kilo-bytes, and the time required (TPG Time) on a VAX 11/8800 are

CKT	CONTEST		STALLION		STEED	
	TFC	Time	TFC	Time	TFC	Time
cse	–	–	100.0	32.8s	100.0	29.5s
sse	99.56	291s	99.8	26.6s	100.0	25.1s
planet	98.42	52m	99.95	6.58m	100.0	5.85m
sand	–	–	99.36	12.2m	99.43	7.77m
scf	–	–	98.23	59.9m	98.81	57.58m
mult4	97.41	838s	99.21	42.4s	99.70	11.9s
sbc	–	–	98.34	80.0m	98.64	75m
stage	–	–	100.0	154m	100.0	50.8m
key	–	–	35.95	> 900m	100.0	419m
pewd	–	–	100.0	117m	100.0	6.58m
dsip	–	–	99.99	350m	100.0	25.8m

Table 2.4: Comparisons with STALLION and CONTEST

shown for each example. For the example **stage**, it was not possible to store the entire cover, and only parts of the cover were generated. For this example, all redundant faults were combinationally-redundant, and the partial covers were as effective as total covers for test pattern generation. The quality of test patterns is determined by the total amount of test data (bits) that have to be stored in the tester and the time required to apply these vectors to the circuit under test. The total number of test vectors suggest that the amount of test data is within reasonable limits for these circuits.

CPU times for enumeration of covers, justification, differentiation, total test generation, fault simulation, miscellaneous setup operations, and for the entire test generation process are given in Table 2.3. The test generation time includes the time required for justification and differentiation. It is noteworthy that justification and differentiation, which are traditionally the bottlenecks in sequential test pattern generation, take a small fraction of the total test pattern generation time for most of the examples.

The fraction of time spent in cover enumeration is very small, much smaller than the corresponding time required for even partial STG enumeration in STALLION. Test generation times are in most cases small, and fault simulation dominates the total test generation time. A better fault simulation algorithm can help in decreasing the time required for test pattern generation. It is worthwhile to note that the success rate, *i.e.*, the percentage of times that a potential fault-free test sequence is valid, is 100% for the larger examples.

CKT	#Scan design Testing Cycles	#Non-scan design Testing Cycles
cse	368	397
sse	486	336
planet	834	1046
sand	1290	772
scf	2008	2400
mult4	420	94
sbc	4536	1364
stage	9216	155
key	26208	1669
pewd	5376	221
dsip	12096	212

Table 2.5: Number of clock cycles needed for testing

In Table 2.4, total test generation time (Time) and fault coverage (TFC) of STALLION [90], CONTEST [1], and STEED are compared. As can be seen, this test generation technique obtains close to the maximum possible fault coverage in all the examples. It takes significantly less time than STALLION [90] and CONTEST [1] to achieve the same fault coverage. For the large examples, significant speed-ups were obtained. The results for STALLION shown here have been derived using a newer and improved version of the program that uses better fault collapsing and hierarchical STG enumeration. Thus the results quoted here for STALLION reflect an improvement over those in [90].

To determine the quality of the test patterns, the time required for the application of the test patterns have to be evaluated. In Table 2.5, the times for testing the sequential circuits using the popular Scan approach and the non-scan approach used here are compared. The number of clock cycles required for non-scan testing is equal to the number of test vectors required. For scan design, it is the total number of combinational test vectors multiplied by the number of latches in the circuit. For the large examples, testing is considerably faster for non-scan design than it is for scan design, anywhere between a factor of 4 to 60 times.

Finally, the results on a subset of the ISCAS-89 benchmarks [23] are presented in Table 2.6. For each example, the number of test vectors required, the memory required to store the covers, the percentage of provably redundant faults, and the total fault coverage including the detected as well as the provably redundant faults is shown. The time required

CKT	#Test Vec.	Memory (Kbytes)	Red. Fault (%)	TFC (%)	TPG time
s27	23	0.086	0.00	100.00	0.2s
s208	195	0.330	26.51	97.02	12.5s
s298	280	0.681	11.36	99.02	17.7s
s344	125	3.354	1.50	100.00	15.7s
s349	120	3.354	2.10	100.00	16.9s
s382	1633	3.771	4.50	95.23	66m
s386	238	0.740	18.20	100.00	11.3s
s400	409	4.161	5.89	95.75	60m
s420	808	0.836	43.72	91.16	22m
s444	994	3.186	7.17	95.56	1.66h
s510	733	1.230	0.00	99.82	19.8s
s526	2037	1.755	14.41	90.99	53m
s526n	2287	1.755	14.10	90.95	52m
s641	327	39.505	6.90	93.08	8.5h
s713	315	43.335	11.53	93.11	8.7h
s820	1304	4.374	4.17	100.00	6m
s832	1344	4.338	5.80	99.65	6m
s838	290	2.508	43.87	80.50	75m
s953	1050	4.408	0.90	100.00	86s
s1196	545	98.730	0.00	98.71	3.4h
s1238	576	99.012	5.00	98.96	3h
s1423	4026	503.98	0.00	56.43	9h
s1488	1310	2.920	2.60	100.00	400s
s1494	1374	2.942	3.38	100.00	440s
s5378	1037	712.67	30.25	99.25	10h

Table 2.6: Test generation results for ISCAS sequential benchmarks

for test pattern generation is shown in the last column. For all examples except **s1423** and **s5378**, complete covers were generated. *None of the ISCAS-89 benchmark examples have a specified reset state.* It is difficult to conjecture what the correct reset state for each of the examples could be, given only the logic-level description of the circuits. The fault coverage obtained and the test pattern generation time is very much dependent on the reset state of the machine. If an invalid state is chosen as the reset state, only a small portion of the State Transition Graph of the original machine will be exercised, thereby producing a large number of redundant faults. Stemming from the lack of information about the functionality of the circuits, for each example, a vector of all zeros was assumed to be the reset state.

This could be a very bad choice for some circuits, notably, the circuit **s5378**. It has a very large number of sequential redundancies, *assuming a reset state of all zeros*, which were identified. For some examples, where the entire covers cannot be generated, *e.g.*, **s1423**, the lack of proper reset state information is largely responsible for the poor fault coverage. The high percentage of redundant faults shows that the corresponding circuits were either not optimally designed or that the choice of reset state was a poor one. The size of the total test vector set for each example shows that the tests are of acceptable quality. For the examples in the benchmark set that are not mentioned here, covers could not be generated due to lack of memory.

2.9 Conclusions

A novel approach to test generation for sequential circuits using selective state transition graph enumeration and fault-free justification and differentiation heuristics has been presented. Fault-free justification and differentiation can be performed much more efficiently than the same under faulty conditions since *information can be reused.* Further, selective enumeration of the STG using the intersection of sum-of-product forms, which forms the basis for justification and differentiation, can be performed efficiently using sophisticated data structures. (Other representations, *e.g.*, Binary Decision Diagrams can be used instead of covers and is the subject of ongoing work). Splitting the sequential test generation problem into *three* subproblems, rather than the traditional two, also improves efficiency by enabling the use of parts of the justification and differentiation sequences. These factors combine to give up to fifteen times improvement in performance over previous approaches for large sequential machines. Experimental results also show that faults that require long input sequences are handled efficiently. This approach has been used to successfully generate tests for finite state machines with a large number of latches within reasonable amounts of CPU time, and close to the maximum fault coverage has been obtained. It has been demonstrated that this approach requires significantly smaller time than the test generator described in [90] and [1] while maintaining or improving fault coverage. It was also demonstrated that a larger class of sequentially redundant faults can be determined during test generation.

Despite these advantages, this approach has some drawbacks. The major drawback is the memory required to store the covers for the PO and NS lines. For a class of large circuits, partial covers are not effective for test generation. Though this approach can handle larger circuits than previous approaches, the larger circuits in the ISCAS sequential benchmark set and complete chips are still out of reach. In some cases, very short justification or differentiating sequences exist, but using the mixed depth-first/breadth-first approach, longer length sequences are found. To solve the problem of size, a different approach that uses register-transfer level descriptions for generating justification and differentiating sequences can be used, and is the subject of the following chapter.

Chapter 3

SEQUENTIAL TEST GENERATION USING RTL DESCRIPTIONS

The test generation algorithm described in the previous chapter used novel ideas to improve the efficiency of test generation for sequential circuits. The main drawback of the approach is that complete covers cannot be generated for very large circuits. Though partial covers are adequate for test generation for certain circuits, for a class of large circuits it is not possible to obtain high fault coverage using partial covers. Also, with the increase in cover sizes, the time taken to perform all the intersections increases, thereby increasing the time required for justification and differentiation. To date, there is no known way of generating partial covers so that they can be made as effective as complete covers for test generation. Though the approach presented in Chapter 2 was shown to be more efficient than previous approaches, there is still a need for test generators that can handle large circuits, especially entire chips.

In most cases, large designs can be divided into a *control* portion and a *datapath* portion. The datapath portion performs all the data processing operations under the supervision of the control portion. The major part of most real-life chips is the datapath which contains adders, registers, *etc.* The control portion is usually a finite state machine. The entire circuit (datapath and control) can be thought of as one finite state machine or as a set of interacting finite state machines. An important characteristic of many datapath portions of large circuits is the presence of arithmetic modules. The arithmetic properties of these modules can be exploited to generate tests efficiently.

Large circuits are often synthesized from their high-level or behavioral description. As pointed out in Chapter 1, an intermediate representation that is often used is the *Register-Transfer Level* (RTL) description (*e.g.*, [113]). At this level, the circuit is described as an interconnection of well-defined modules. The logic-level description of the circuit could be huge and cumbersome to manipulate, but the RTL description is often small and easier to manipulate. These properties and some algebraic properties of arithmetic modules in the RTL description can be used to generate tests for circuits described both at the RT and logic level.

The rest of this chapter is organized as follows. In Section 3.1, definitions and some new notions in RTL test generation are presented. Previous work is reviewed in Section 3.2. The global test generation strategy is presented in Section 3.3. State justification, indexed backtracking, conflict resolution, and differentiation are the topics of Sections 3.4, 3.5, 3.6, and 3.7. Results and conclusions are presented in Section 3.8 and Section 3.9 respectively.

3.1 Preliminaries

In a typical design process, as illustrated in Section 1.1, the designer starts with a behavioral description of the circuit and converts it into a structural description, which is an interconnection of well-defined modules like adders, multipliers, multiplexors, registers, finite state machines, *etc.* A description of the system in terms of such modules is called the **Register-Transfer Level** or RTL description. Note that the RTL description might contain arbitrary combinational and sequential functions as modules. These modules are said to be well-defined because either a *Truth Table* or a *State Transition Graph* is associated with such modules. The RTL description is converted into an interconnection of logic gates that is finally implemented. The implementable description is also called the **gate-level** or the **logic-level** description.

As in the test generator of the previous chapter, a gate-level fault model is used. A portion of the work required for test generation is performed at the gate level and the rest is performed at the RT level. Therefore, it is frequently necessary to move from the logic level to the RT level and vice versa. Wires at the RT level could be single wires or buses, and therefore they correspond to multiple logic-level wires. In the test generation procedure to be described, it will be necessary to maintain a correspondence between single wires or buses at the RT level and wires at the logic level. This correspondence is required only for the wires corresponding to primary inputs, present-state lines, primary outputs, and next-state lines. Correspondence between intermediate wires is neither necessary nor desired. This is because logic-synthesis tools may alter the structure imposed by the RTL description of the circuit so that certain intermediate wires might disappear and some new ones might appear. Integers are used as the data type to represent values on wires at the RT level and Boolean values are used at the logic level. Therefore, moving from the RT level to the logic level and back involves the **translation** of numbers from one radix to another, namely, decimal to binary and vice versa. There is an important issue in translation, especially of negative integers. If a circuit consists only of adders and arbitrary Boolean functions, then during translation the particular encoding of the integer (either as 1's-complement or 2's-complement) does not matter. However, if a circuit contains multipliers, since multipliers are built for a certain encoding, that encoding has to be used during translation. Also, the underlying encoding limits the range of integer values on an RTL wire. For example, for a circuit that consists of a single 3-bit multiplier that uses 2's-complement, the input wires will have a range of -4 to 3 and the output wire will have a range of -16 to 15. For all circuits with arithmetic modules, the same encoding that is used for integers in the circuit is used during translation and to determine the range of integer values on all RTL wires.

Consider the circuit shown in Figure 3.1. Assume that wire g has a certain value and module A is an adder. To **justify** the value on wire g is to find an assignment of values to wires b and c so that their values add up to the value on wire g. In this example, initially, wires b and c are *free* to take any value. However, once either b or c is assigned a value, the other wire is forced or *bound* to take a fixed value in order to justify the value on g. A wire in the circuit is said to be **free** if it can take more than one value without causing any conflicts. A wire that is not free is said to be **bound**. In other words, a wire is said to be free if its value cannot be uniquely implied, either by forward or backward implication. Implication, as will be seen later, is performed each time a wire is assigned a value. A **conflict** is defined to be a situation where a particular wire (called the **primary wire** of the conflict) is required to have two or more different values simultaneously in order to justify the values on other wires (these wires are said to be forcing requirements on the value of the primary wire and are called the **secondary wires** of the conflict). For the circuit of Figure 3.1, assume that wires h and i have the values 20 and 1 respectively. Let

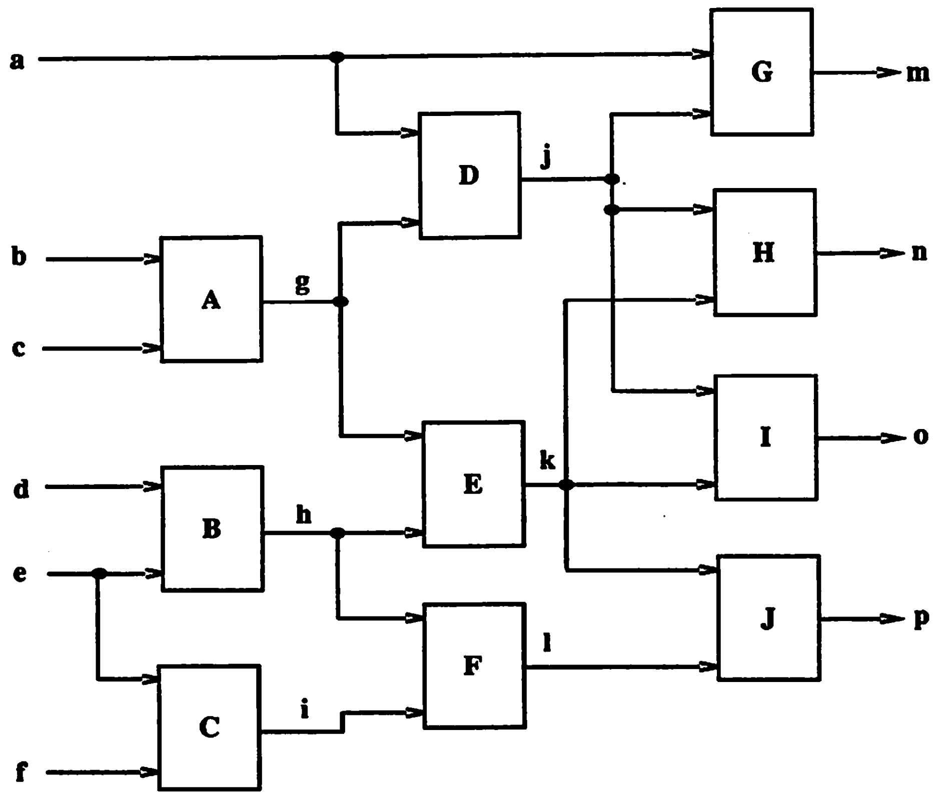


Figure 3.1: An example RTL description

module *B* be a multiplier and module *C* be a comparator. Assume wires *d* and *f* were assigned the values 4 and 16 respectively. In order to justify the value on wire *h*, wire *e* is required to have the value 5, but to justify the value on wire *i* it is required to have a value greater than 16. This is a conflict with wire *e* as the primary wire and wires *h* and *i* as the secondary wires. In the circuit of Figure 3.1, assume that wires *n*, *o*, and *p* have values on them that have to be justified. Wire *m* is unimportant and can assume any value. Initially, wires *j*, *k*, *l*, and *a* are free. Assume that wire *l* is assigned a value. This forces wire *k* to have a fixed value in order to justify the value on wire *p*. Therefore wire *k* is bound. Forcing *k* to have a fixed value also forces wire *j* to have a fixed value. Wire *k* is said to be **directly bound** by the free wire *l*, while wire *j* is said to be **transitively bound.** Note that wire *a* is not bound because the value on *m* is not important. If wire *m* were to have a value that had to be justified, a too would have been transitively bound by l.

Consider a path from the input f to the output p consisting of wires i and l, in the example shown in Figure 3.1. To propagate a value on f to the wire p, three modules have to be encountered, namely C, F, J. The other inputs to these modules are called the **side-inputs** to the path, and they are e, h, and k.

The **fanout** of a gate $\mathcal{G}$ is defined as the set of gates that use the value generated by the gate $\mathcal{G}$. The **transitive fanout** of $\mathcal{G}$ is defined recursively as follows. If $\mathcal{G}$ is a gate generating only a primary output, then its transitive fanout is the null set. Else, the transitive fanout of $\mathcal{G}$ is the union of the fanout of $\mathcal{G}$ and the transitive fanout of every element in the fanout of $\mathcal{G}$. The **fanin** of a gate $\mathcal{G}$ is defined as the set of gates whose output value is used by the gate $\mathcal{G}$. The **transitive fanin** of $\mathcal{G}$ is defined recursively as follows. If $\mathcal{G}$ is a gate whose inputs are only primary inputs, its transitive fanin is the null set. Else, the transitive fanin of $\mathcal{G}$ is the union of the fanin of $\mathcal{G}$ and the transitive fanin of every element in the fanin of $\mathcal{G}$. Similar terms can be defined for a wire in the circuit. For any wire that is an input to a module(s), the **co-fanin** of the wire is defined as the set of wires that are also inputs to the same module(s), *e.g.*, the co-fanins of e are d and f.

As in the previous chapter, it is assumed that the circuit has a reset state. If the circuit is simulated with the PS lines set to the reset state of the circuit and the primary inputs set to unknown, some wires in the circuit might get set to a known value. This value is called the **reset value** or **guide value** for the wire. Wires which do not get set to a known value during this simulation do not have a reset value. For any RTL module, forward and backward implication (or justification and differentiation) is performed using the description of the module. If the module has an associated operator, the algebra of the operator is used. Otherwise, the Truth Table of the module is used. This implies that if a large number of modules are arithmetic, then the time taken for implication is relatively smaller.

The assumptions about the sequential circuit for which test patterns have to be generated are the same as in the previous chapter. For the ease of reading, they are repeated here.

1. The machine is assumed to have a **reset state,** $\mathcal{R}$. All test sequences are applied with this state as the starting state.

2. A gate-level fault model is used, and it is the **single stuck–at** fault model. Since

the state justification and differentiation parts of the test generation algorithm are independent of the fault model, the algorithm is not restricted to only this fault model. Other fault models like multiple stuck-at, bridging faults, *etc.*, may be used with minor modifications to the procedure described.

3. The memory elements are considered as distinct logic primitives and faults inside the memory elements are not considered. However, all faults on present-state and next-state lines are considered.

4. It is assumed that an RTL description of the circuit is available.

3.2 Previous Work

In Chapter 2, previous work in the area of test pattern generation for sequential circuits was reviewed. In the rest of this section, recent work in the use of high-level information in test pattern generation is briefly presented.

In [3], a symbolic test generation system for hierarchically modeled digital systems is described. The system under test is modeled as a *data path* and a *control section.* Often, partitioning the circuit into data path and control sections is somewhat arbitrary. The modules in the datapath section are defined as an interconnection of other modules or described behaviorally. The behavioral model is a simplified and incomplete model of the module and is used for the purpose of fault-effect propagation and justification. The control section is always modeled as a finite state machine. Since a hierarchical description of the controller is not allowed, large controllers for which State Transition Graphs cannot be generated, cannot be handled by this approach. There is a fundamental assumption that the control section has been tested using existing techniques and is found to be fault-free. This can be an artificial assumption, as the system designer often wants to test the control and the datapath together. Also, for control dominated chips, this approach will not be applicable because it assumes that the control portion is handled by some other test generator. Instead of using binary or integer values, symbols are used to represent values on the wires. However, in any computer implementation, the symbols have to be represented as a code, and therefore integers or a similar representation have to be used for the encoding. Therefore the advantage of using symbols instead of integers is not clear. Very few results were quoted for this approach and it is difficult to gauge its effectiveness.

In [117], the gate-level combinational test pattern generator SOCRATES [119] is augmented to handle circuits described hierarchically. The only high-level primitives supported are decoder, demultiplexor, bus, encoder, multiplexor, tri-state driver, and one-bit adder. The implication, unique sensitization, and multiple backtrace procedures used in SOCRATES are extended to handle these high-level primitives. A gate-level fault model is used and modules inside which faults have to be considered are dynamically expanded to their gate-level circuit. Therefore the structure of the circuit changes dynamically during test generation. Apart from these modifications, the procedures are similar to well known procedures in combinational test pattern generation. This approach handles combinational circuits only. A maximum speed-up of a factor of two is reported for this approach.

Module-oriented decision making is introduced in [32]. Modules and faults were captured at both the functional and the gate level. The problem of hierarchical test generation is viewed as an extension of the gate-level problem. This approach works only for combinational circuits and for a limited number of high-level primitives. Some other work in exploiting high-level information (*e.g.*, [76, 81]) has been restricted to combinational or Scan-based sequential circuits and does not offer guarantees as to stuck-at fault coverage.

One of the major problems encountered in using purely high-level information is in the fault modeling process. Almost all approaches that use high-level information in the test generation process use a gate-level fault model. Gate-level fault models have the advantage of being accurate and easy to use. On the other hand, modeling faults at higher levels is an extremely complex process. There is not only a significant loss in accuracy but the fault model is often too complex to be of any use. In [22], functional level primitives for test generation are obtained and the problem of functional fault modeling is addressed. Though this problem can be solved for simple modules, there are modules for which there are no known good methods for obtaining functional fault models.

3.3 Global Strategy for Test Generation

The key ideas in the test generation algorithm are similar to those used in the previous chapter. The problem of test generation is divided into three sub-problems. Combinational test generation is used to derive the excitation state, state justification is used to justify this state, and state differentiation is used to propagate the effect of the fault to the POs. The *fault-free* heuristic is used for justification and differentiation. Unlike the algorithm in the previous chapter, justification and differentiation is not performed using the covers of ON and OFF sets. Instead, the RTL description of the circuit is used. To motivate the use of RTL descriptions for justification, consider a 32-bit combinational multiplier with a prime value at its output. There is only one way to justify this value; by assigning one of the inputs the prime value and the other input the value 1. To discover this from the logic-level description might require significant amounts of work, but at the RT level this is easily performed. Similarly, for the example circuit of Figure 3.2, it is easy to see that a value can be propagated from the wire C to wire G by setting the values of A and B to 0. This global knowledge available at the RT level can be exploited to perform justification and differentiation. Given an RTL as well as a logic-level description of the circuit, the deterministic sequential test pattern generation algorithm consists of the following steps:

1. Derivation of the excitation state for a fault using combinational test generation, treating the PS lines also as primary inputs (PIs) and the NS lines also as POs. This is performed at the gate level using a standard test pattern generation algorithm (*cf.* Section 2.4).

2. *Fault-free* state justification performed at the RT level.

3. *Fault-free* state differentiation performed at the RT level.

The RTL description is represented as a *graph* whose nodes are modules and edges represent wires. The nodes are annotated with the information about the module and the edges are annotated with information about the wires. Information at a node consists of:

- Fanin wires for the module.
- Fanout wires for the module.
- Type information. This field is used to indicate whether the module is arithmetic, logical, an arbitrary Boolean function, or finite state machine.
- Function-type information. This field indicates whether the module is one of the primitives or has a model in terms of the primitive modules.
- Truth Table or STG of the module, if necessary.
- Model information, if necessary.

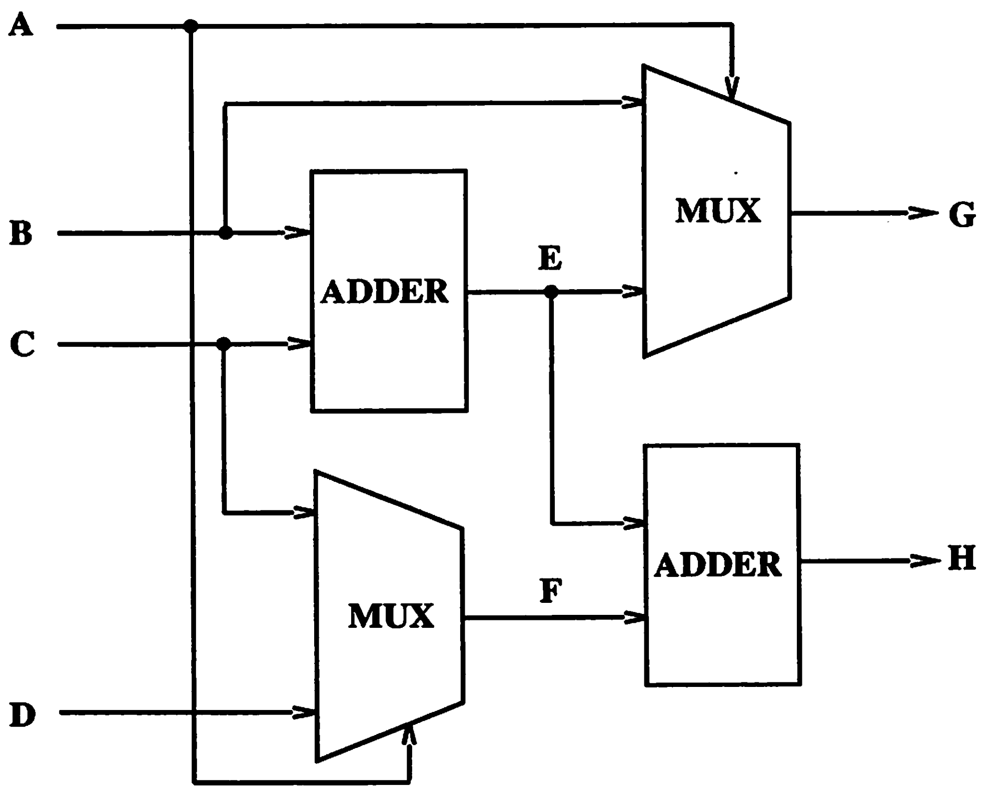


Figure 3.2: An example circuit

Information at a wire consists of:

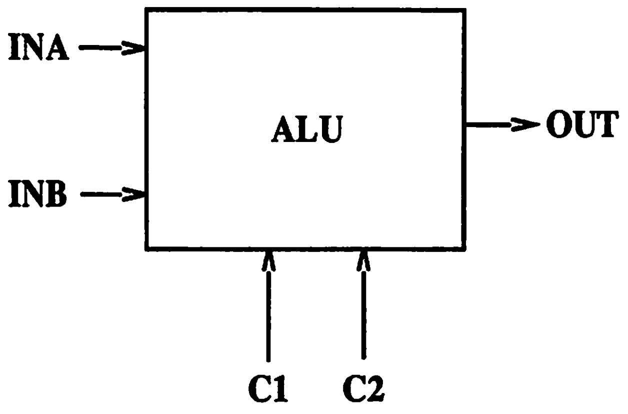
- Fanin module of the wire.
- Fanout modules of the wire.
- Correspondence with logic-level wires, if necessary.
- The width of the wire in terms of the number of logic-level wires needed to implement the RTL wire.
- Encoding type, which is the type of value on the RTL wire. Integers are used for almost all the wires in the circuit except for some modules whose input/output behavior is specified using symbolic values and their associated encodings.

- Range of integer values, if the encoding type is integer. Otherwise the symbolic values and their encodings are specified.

The graph representing the RTL description can be easily obtained from the connectivity graph of the system and the process of translation is straightforward. The primitives supported by the test generation system are given in Figure 3.4. For an arbitrary Boolean function or a finite state machine, it is required that a *Truth Table* or a *State Transition Graph* be specified and it is stored at the node corresponding to the module. Not all modules can be represented using primitives or truth tables. In such cases, a *model* is used for the module. A model is always defined in terms of primitives. An example of such a model is the Arithmetic-Logic Unit (ALU). The model for the ALU is shown in Figure 3.3. Depending on the values of wires $C1$ and $C2$, the appropriate primitive is inserted into the circuit during justification and differentiation. These models are supplied by the user.

After combinational test generation, the excitation vector is examined to see if the present state part of the excitation vector covers the reset state. If the excitation state is covered by the reset state, then the fault can be excited from the reset state of the machine. If not, the excitation state is justified using a backward justification algorithm. Backward justification is performed by considering the excitation state as a next state and using an RTL justification algorithm to justify the values on the NS wires over multiple time frames (*cf.* Section 3.4). Once a justification sequence is found, it is fault simulated to see if the required state is justified. If the required state is justified, then the justification sequence is valid in the faulty machine. If the required state is not justified, then a correct justification sequence can be obtained easily from the invalid justification sequence using the technique described in Section 2.5. Thus only a single fault-free justification has to be performed to obtain a true/faulty state pair.

If the effect of the fault is propagated only to the next-state lines, then the effect has to be propagated to some primary output by state differentiation. State differentiation is performed using the RTL description by sensitizing a path from a present-state wire having the true and faulty value to a primary output. If an input combination exists that can sensitize such a path, then it serves as the state differentiation vector. Multiple-vector differentiation sequences can also be obtained (*cf.* Section 3.7). The differentiation sequence obtained is valid under fault-free conditions. After the differentiation sequence is obtained, the entire test sequence is fault simulated to see if the fault under test is detected.



C1 C2 = 0 0	OUT = INA+INB
C1 C2 = 0 1	OUT = INA–INB
C1 C2 = 1 0	OUT = INA & INB
C1 C2 = 1 1	OUT = INA || INB

Operation of ALU

C1	C2	Operation
0	0	OUT = INA (ADD) INB
0	1	OUT = INA (SUB) INB
1	0	OUT = INA (AND) INB
1	1	OUT = INA (OR) INB

Model of the ALU

Figure 3.3: An ALU and its model

Experimental evidence gathered to date indicates that more than 95% of the time, a test sequence obtained is actually a test for the fault (*cf.* Section 3.8).

The fault-free heuristic enables the use of RTL descriptions for justification and differentiation without using any kind of functional fault modeling. If the fault-free heuristic were not used, it would be necessary to identify the module in which the fault occurs. This would require a correspondence between modules at RT level and a set of gates and wires at the logic level. This is an undesirable requirement because logic synthesis tools can easily blur the boundaries between modules during operations like factoring and resubstitution. Another problem is that of functional fault modeling. The effect of the fault has to be modeled as a discrepancy in the behavior of the module. It was indicated in the previous section that this is a hard problem and there are no known good solutions.

adder	decoder	and
subtractor	comparator	or
multiplier	multiplexor	not
divider	demultiplexor	nand
finite state machine	arbitrary function	buffer
encoder	xor	nor
		xnor

Figure 3.4: List of primitives used in test generation

A test generation algorithm based on the ideas discussed above is as follows:

1. Generate a (new) combinational test vector for the fault under test to derive the excitation state S. If test vectors for the fault have already been generated and some of the excitation states were not justifiable, the new test vector has its present-state part disjoint from the present-state part of all such previously generated test vectors. If no new test vector can be found, then exit without a test.

2. Generate a (new) fault-free justification sequence J_S for the excitation state. If no justification sequence is found, go to Step 1.

3. Fault simulate the potential justification sequence J_S. If it detects the fault, generate a test sequence T_S directly from J_S. If it is a valid justification sequence, then go to Step 4. If it is not a valid justification sequence, then find the *first* state whose fanout edge was corrupted by the fault. Use part of J_S as the justification sequence for that state and try to differentiate between the new true/faulty state pair.

4. Generate a fault-free differentiation sequence D_S for the true/faulty state pair. Concatenate J_S, the excitation vector, and D_S to obtain the test sequence T_S. Fault simulate T_S. If the fault under test is detected, exit with test sequence T_S. If the fault is not detected, try generating another D_S. If not successful, go to Step 2.

In each pass of the algorithm, for a given excitation state, a justification sequence is derived. Backtrack limits are set for justification and differentiation that control the amount of time spent in each of these steps. These limits are user controllable. By checking to see if the justification sequence is valid prior to state differentiation, the need for generating more than one justification sequence in each pass of the algorithm is obviated. Now all possible differentiation sequences are generated for the fault-free/faulty state pair. If the fault is not detected, the procedure goes back to Step 2 to generate another justification sequence and initiate another pass of the algorithm. Therefore, for each excitation state, all possible justification and differentiation sequences are generated. Also, for each fault, all possible excitation states are tried, if necessary. Thus the algorithm will find a test for a fault if it exists in the fault-free machine. In this sense, the algorithm is complete. All justification and differentiation sequences generated are stored for possible reuse. Thus, parts of the State Transition Graph that are required for test generation are enumerated, as in the algorithm described in the previous chapter.

This test generation algorithm is not guaranteed to find a test for a fault if one exists. This is due to the nature of fault-free differentiation. Firstly, a fault-free differentiation sequence might not be a valid differentiation sequence in the faulty machine. Secondly, though a fault-free differentiation sequence might not be obtainable, a test for a fault may exist, as noted in Chapter 2. To obtain such a test vector, the effect of the fault has to be modeled at the RT level using some form of functional fault modeling, and then a true and a faulty copy of the RTL description has to be used to propagate the effect of the fault to the primary outputs, as in the iterative array approaches described in Section 2.2. Due to the inaccuracies in functional fault modeling, this approach is also not guaranteed to generate a test. However, as experimental evidence gathered to date indicates, more than 95% of the time a fault-free test sequence is valid. If one such sequence is not valid, another fault-free test sequence may be generated for the fault under test. Use of multiple fault-free sequences produces 100% fault coverage in all the examples that have been considered.

This approach works best for circuits that have an STG where any state can be reached easily from any other. This is true for almost all datapath circuits. Of course, by exploiting the higher-level arithmetic properties, the approach is most efficient when the majority of the modules in the circuit are arithmetic. This property is used during the justification, differentiation, and conflict resolution phases of the algorithm as illustrated in the sections that follow.

```
justify_state()
{
  Assign all unknown logic-level NS wires a value;
  Store all assignments in binary decision tree;

  while (binary decision tree != NULL) {
    value_set = values on each RTL NS wire;
    if (value_set covers Reset State )
      return (NULL justification sequence);
    flag = justify_values(value_set, 1);
    if (flag) {
      return (justification sequence);
    }
    else {
      /* Justification sequence not found */
      assign last variable on binary decision tree
      a different value;
    }
  }
  /* No justification sequence obtainable */
  return (justification not possible);
}
```

Figure 3.5: Main justification procedure

3.4 State Justification

Combinational test generation produces a test vector with as many don't-care entries in the present state part as possible. The present state part of the excitation vector is the excitation state S_1. In general, S_1 is a cube state. Any state covered by S_1 has to be justified.

The main justification procedure is shown in Figure 3.5. Justification is performed using the RTL description of the circuit. The objective of the justification algorithm is to find a justification sequence to any one of the minterm excitation states covered by S_1. To justify a state, it is necessary to think of it as a next state and use a backward justification algorithm. Therefore, the first step in justification is to transfer the values from the present-state wires to their corresponding logic-level next-state wires. The next step is to find a single or a minterm excitation state. A particular excitation state is selected by assigning the unknown logic-level next-state wires a random 0 or 1 value. All NS wires not having a value on them are assigned values, with one exception. If a set of NS wires do not have a value and they correspond to a single wire at the RT level, then those wires are not assigned values. This ensures that the corresponding next-state RTL wire is not assigned a value, thereby giving more freedom to the justification algorithm. Next, the Boolean values on the logic-level wires are translated into integer values on the corresponding RTL wires. This set of values on the next-state wires of the RTL description, after translation, is called the *value set.* A *value set* is therefore an integer representation of a minterm state. The state defined by the value set is justified by the routine **justify_values()**. It is possible that the state represented by *value set* is not a valid state. Therefore, the assignments to the unknown logic-level next-state wires are stored in a decision tree. If a state cannot be justified, then the last assigned NS wire is assigned a different value and the process of justification is repeated by calling the procedure **justify_values()** with the new *value set.* Thus all possible minterm excitation states for a fault are tried, if necessary. The decision tree used in this procedure is called the binary decision tree because all variables in the tree can only take two values. For large datapath-type circuits, almost any state is reachable from any other and the number of invalid states is not large. For such circuits, no backtracking is usually necessary on the binary decision tree. However, there are circuits that have the opposite characteristic. Such circuits are often small and can be best handled by STEED. This approach is not likely to perform better than STEED for such circuits.

The procedure for justifying a particular state is shown in Figure 3.6. Before justification, the circuit is levelized from the primary outputs backwards. Primary output and next-state wires are assumed to be of level 0. The level of any module is the minimum of the levels of all its fanout wires. The level of any intermediate wire is the minimum of the levels of its fanout modules plus one. Wires at a particular level are assigned values to justify the values on wires at lower levels, and the procedure is called recursively until the primary inputs are reached. During justification, only next-state wires, which are level 0, have values on them that have to be justified. Justification starts with wires at level 1. At

```
justify_values(v_set, Level)
{
  /* Level is the level of RTL wires being considered */
  /* v_set is the values on wires at previous level */
  do {
    Select free wire at Level, assign value, store in decision tree;
    Assign value to bound wires, simulate circuit;
    if (conflict)
      conflict_resolve();
    if (conflict resolved or no conflict) {
      if (all free wires assigned values) {
        if (all wires at Level are PI or PS) {
          Store vector as part of justification sequence;
          if (reset state covered)
            return (TRUE);
          else{
            Form new_state;
            flag = justify_values(new_state, 1);
            if (flag)
              return (TRUE);
          }
        }
        else { /* All wires are not PI or PS */
          Form new_v_set;
          flag = justify_values(new_v_set, Level+1);
          if (flag)
            return (TRUE);
        }
      }
      else { /* All free wires not assigned values */
        continue;
      }
    }
    /* Conflict not resolved or justification sequence not found */
    /* Use indexed backtracking if necessary */
    Assign last node in decision tree a different value;
  }while (decision tree is not empty)
  return (FALSE); /* No justification sequence found */
}
```

Figure 3.6: Procedure for justifying a state

first a free wire at the given level is identified and assigned a value (heuristically) and this decision is stored in a decision tree. All wires which are bound (directly or transitively) by the free wire are then assigned values. After assigning values, the circuit is simulated to detect any conflicts (this is similar to forward implication performed in D-Algorithm or PODEM). If there is a conflict, the conflict resolution procedure is called (*cf.* Section 3.6). This procedure tries to form a system of equations with the values of wires involved in the conflict. In this system of equations, all wires involved in the conflict are assigned a variable and the system of equations represents their interdependence. This system of equations is solved to obtain the correct assignment of values for the wires. Sometimes no solution can be obtained using this procedure. In such cases, it can be assumed that the *value set* cannot be justified under the given conditions. Therefore, the procedure must backtrack to a previous node in the decision tree. If the conflict resolution procedure fails (*i.e.*, the system of equations cannot be assembled or a solution cannot be found), then the last free wire in the decision tree is assigned a new value and the process repeated. The backtracking procedure goes through all the values of a free wire before backtracking to the previous node in the decision tree. If no solution is found, backtracking continues till the last node in the decision tree is reached, *i.e.*, to the first free wire that was assigned a value. During backtracking, one of two things might happen. A satisfactory assignment might be found or the decision tree may become empty. If the decision tree becomes empty, it can be concluded that the state represented by value set cannot be justified. Such a state is an invalid state and is stored in the set of invalid states.

After all free wires at a given level are assigned values without any conflicts, the procedure **justify_values()** is called recursively to justify the newly assigned values. At each level all essential assignments are done first, *i.e.*, wires that *have* to be assigned a certain value to justify values at lower levels are considered first. This is similar to backward implication in the D-Algorithm. However, unlike the D-Algorithm where implications can be performed for many levels at a time, here it is performed for only one level at a time. The reason for this is that implications usually can be performed for only one level in an RTL description. During justification and differentiation, forward and backward implications are performed as soon as a wire is assigned a value, for early detection of conflicts. Note that none of the essential assignments are stored in the decision tree as they are considered to be bound wires (they are forced to have a single value). The *decision index* assigned to any such wire is the largest of the decision indices of its fanout wires as well as its co-fanins (*cf.* Section 3.5).

During justification, the frontier of assigned values moves from the outputs of the RTL description to the inputs (referring both to the PI and PS), and when all inputs have been successfully assigned a value, the one-step justification procedure is done. Note that if the backtracking procedure backtracks to a node of level i, then all decisions below level i have already been undone, and the backward justification procedure starts once again from level i. If the reset state is covered by the values on the PS wires, then a single-vector justification sequence has been found. If not, the state corresponding to the values on the PS wires is justified by calling the routine **justify_values()** recursively. In this manner multiple-vector justification sequences might be derived. After deriving the justification sequence, the RTL values are translated to binary values and the justification sequence is fault simulated.

Heuristics are used to speed up justification and the detection of invalid states. From the set of free wires that can be selected at any given point, the one with the maximum number of fanouts is selected first. Also, wires with guide values are given priority during selection. A wire with a large fanout binds a large number of other wires. This helps in early detection of conflicts and also in keeping the size of the decision tree small so that less effort is spent in backtracking. For wires having the same number of fanouts, the one closest to the primary inputs is selected. This is because a wire closer to an input is more easily justified than one further away. While setting free wires to a value, the guide value for the wire is selected first. If a guide value does not exist for a wire, then a value is selected in the following manner. An RTL wire that has to be implemented as n logic-level wires can have values ranging from 0 to $2^n - 1$, assuming an encoding where only positive integers are allowed. If an RTL wire does not have a guide value, then the first value selected for the wire is a value midway in this range, *i.e.*, 2^{n-1}. In subsequent backtracks, the value chosen is $2^{n-1} + (-1)^k \lceil k/2 \rceil$, where k is the number of times the procedure has backtracked to the node in the decision tree corresponding to that wire ($\lceil x \rceil$ means the smallest integer greater

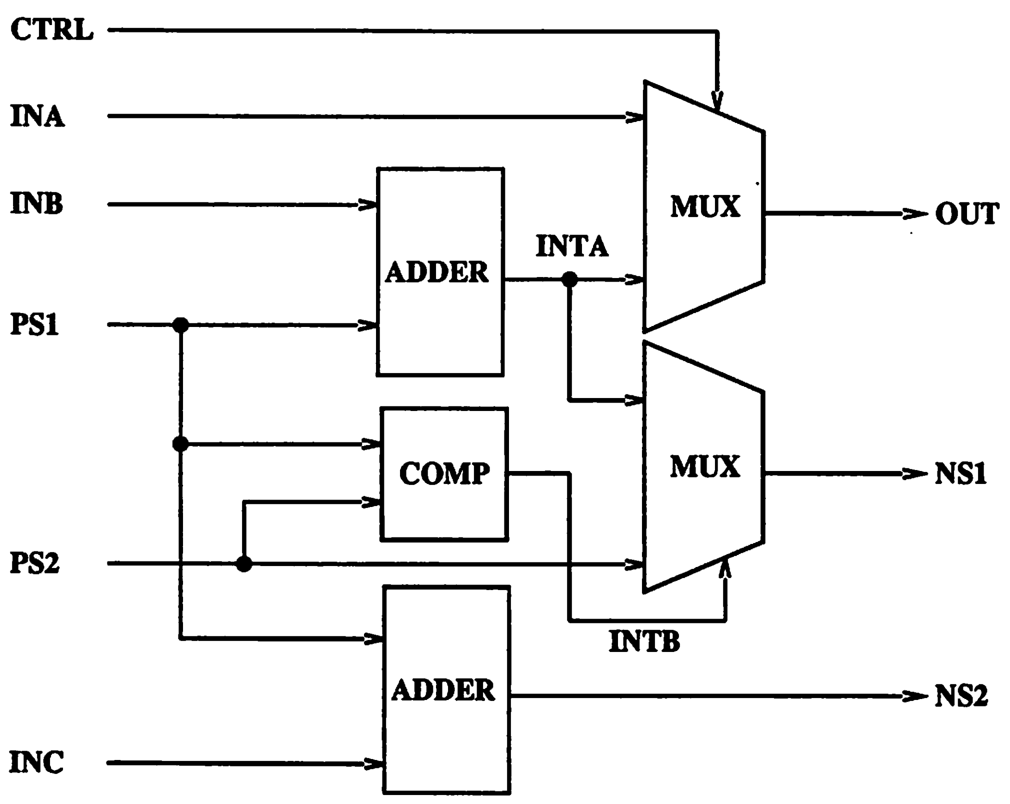


Figure 3.7: Circuit to illustrate justification and differentiation

than or equal to x). For example, if the RTL wire corresponds to 3 logic-level wires, the first choice would be the value 4. Subsequent choices, in order, would be 3, 5, 2, 6, 1, 7, and 0. For other encodings, the process is similar. Modules like multiplexors are handled easily by noting that the free wire is the control input to the multiplexor and the corresponding input to the multiplexor is bound. The assignment to the input wire and the control wire for the multiplexor is stored as a single node of the decision tree. In addition, indexed backtracking as in [95], with modifications, is used to speed-up backtracking by eliminating non-critical nodes in the backtracking process (*cf.* Section 3.5).

As an example, consider the circuit of Figure 3.7. Wires INA, INB, INC and $CTRL$ are primary inputs, $PS1$ and $PS2$ are present-state lines, OUT is the primary output, and $NS1$ and $NS2$ are next-state lines corresponding to $PS1$ and $PS2$ respectively. The modules are adders, multiplexors (denoted as MUX), and a comparator (COMP). The output of the comparator is 0 if and only if $PS1$ is strictly greater than $PS2$, otherwise it is 1. Assume that the reset state of the machine is all zero, and that it is required to find a justification sequence for the value set $NS1 = 7$ and $NS2 = 29$. In the first step the control input of the multiplexor, wire $INTB$, is set to 1. This binds $PS2$ to the value 7. To justify the value on $NS2$, $PS1$ is chosen as the free wire, and since it has a reset value, the value chosen for it is 0. This binds INC to the value 29. On simulation, it is seen that the value on $INTB$ is justified. The values on the present-state wires are 0 and 7, and this state does not cover the reset state. Therefore, this new state has to be justified. In the next step the value set is $NS1 = 0, NS2 = 7$. Once again, assigning $INTB$ to 1 forces $PS2$ to have the value 0. Choosing the guide value for $PS1$, the value on $NS2$ can be justified by assigning the value 7 to INC. The value on the present-state wires is $PS1 = 0, PS2 = 0$, and this covers the reset state. Therefore a two vector justification sequence for the initial value set is $\{[INA = INB = CTRL = -, INC = 7], [INA = INB = CTRL = -, INC = 29]\}$. In this justification sequence, the values of the inputs $CTRL$, INA, and INB are don't-cares.

Another interesting example is the circuit of Figure 3.2. In this circuit, wires A and B are the primary inputs, C and D are the present-state lines, G and H are the next-state lines, and G is also the only primary output. Let the value set be $G = 20, H = 16$. Setting $A = 0, B = 20$ justifies the value on G, and C is forced to take the value -2 to justify the value on H. The value of D is left unspecified. In the next step of justification, the value set is $G = -2, H = -$. This value on G is easily justified by setting $A = 0, B = -2$. The value of both the present-state wires is left unspecified. Therefore the present state is the universal state, and the reset state must be covered by the universal state. Therefore a two-vector justification sequence for the original value set is $\{[A = 0, B = 20], [A = 0, B = -2]\}$.

The justification sequence that is constructed is valid under fault-free conditions and may be invalid under faulty conditions. Experimental evidence gathered to date has shown that over 99% of the time, in real-life circuits, a justification sequence that is valid in a fault-free machine is also valid in the faulty machine or is in itself a test sequence for the fault. In the unlikely event that a justification sequence is neither valid in the faulty machine nor a test sequence in itself, a valid justification sequence may be obtained from the invalid one using the procedure of Section 2.5.

The procedure for finding a justification sequence uses a depth-first search method. This approach is therefore best suited for circuits whose STGs are not too deep. Most datapath-type circuits meet this condition. However, this does not mean that this approach fails for circuits that require long justification and differentiation sequences. As will be shown in Section 3.8, some such circuits are handled efficiently. In theory, finding a justification sequence for a state can involve the enumeration of the entire STG of the machine. Therefore, detection of invalid states can be computationally very expensive. In most cases invalid states can be identified very quickly because the number of invalid states are small and they form relatively small groups of interconnected states. However, there might exist examples where justification and detection of invalid states can take significant amounts of time and impair the efficiency of this approach.

3.5 Indexed Backtracking

One of the main objectives during justification and differentiation is to keep the number of backtracks small. Each wire at the RT level can have a large number of values. Going through all possible values of a wire is a time consuming process and should be avoided wherever possible. Consider a situation where a decision tree has three nodes, and a conflict is detected that cannot be solved using the conflict resolution procedure. This means that the last node in the decision tree should be examined and a new decision should be made at that node. If the last decision made has nothing to do with the conflict, then the procedure would unnecessarily go through all possible values of the last wire in the decision tree. Therefore, it is necessary to reach the node in the decision tree which affects the wires involved in the conflict in a single step so as to avoid the unnecessary backtracking.

To achieve this goal the technique of indexed backtracking is used. The main steps in the process are:

- Create an array called the **decision array**. This array grows or shrinks dynamically.
- Each time a free wire is assigned a value, the pointer to the decision tree node corre-

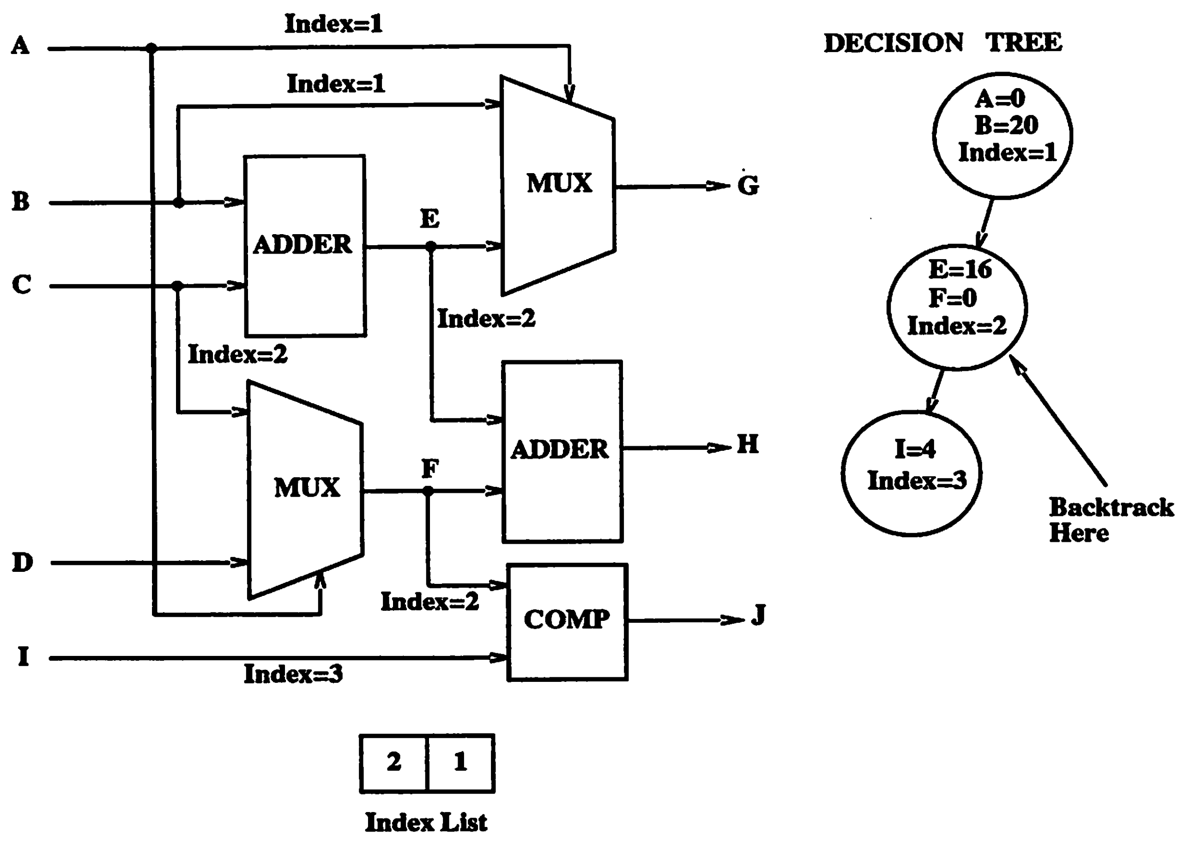


Figure 3.8: Circuit to illustrate indexed backtracking

sponding to the decision is stored in the first free position in the decision array. The index of this entry is stored in the decision tree, and the free wire and all wires bound by the free wire are given the same index. This index is called the **decision index.**

Indexed backtracking is used only when a conflict happens and cannot be resolved by the conflict resolution procedure. The wires involved in the conflict are first identified. The index of the wires are gathered in a list, called the **index list**, and sorted in descending order. The head of the list (*i.e.*, the most recent decision affecting the conflict) is examined and the procedure backtracks to the node corresponding to that index and not to the last node in the decision tree. If the conflict cannot be resolved by assigning different values to the the wire at the node, then the next node in the index list is used for backtracking. The critical step in the procedure is to detect the set of wires involved in the conflict. This set of wires, called the set C, is derived using the following rules:

- The primary and secondary wires of the conflict are in this set.
- For each wire, w, in the set, wires whose values are justified by w are also in the set.
- For each wire, w, in the set, the wires in its co-fanin that have been assigned a value and also force w to have a particular value are also in the set. In other words, if w is considered as a free wire, the co-fanins of w that are bound and have been already assigned values are in the set. Those that are not bound are not in the set, unless selected by some other rule.
- For each wire in the set, if the wire is the fanout of a module whose fanin wires have been assigned values, then the fanin wires are also in the set.
- For each wire in the set, all wires with the same decision index are also in the set.

It is easy to see that for any wire, the value on that wire is directly determined only by one or more of the following: the value of its fanout wires, the values of its fanin wires, or by the values of its co-fanin wires. The rules are formulated so that all wires that directly determine the values of the primary and secondary wires of the conflict are selected first in set C, and then a transitive closure of this set is obtained. Therefore, it is easy to see that for any wire in the set C, all wires that affect its value, directly or indirectly, are also in C. It can also be easily seen that the primary wire of the conflict and the secondary wires of the conflict are separated by at most one level. In justification, all wires at a level are assigned values before moving on to wires at the next higher level. If there is a conflict on a wire at level p, it must be due to the requirements enforced on the wire by wires at level $p-1$. Note that all wires at level $k < p-1$ have their values justified, and there can be no conflict involving those wires as secondary wires.

A conflict can be resolved in one of two ways. It can be resolved if the requirements on the primary wire are changed by changes in the values of the secondary wires of the conflict. It can also be resolved by a change in the value of the co-fanins of the primary wire, whereby the primary wire can be assigned a different value to justify the values on the secondary wires. To show that the procedure only avoids looking at portions of the space where a justifiable assignment does not lie, it is necessary to show that for any set of values on the wires in C for which the conflict is not resolved, the conflict will not be resolved by changing the values on wires not in C. This is proved in the following theorem.

Theorem 3.5.1 *Given the primary wire, W, involved in the conflict, the set of wires involved in the conflict C, if wires in C have values for which the conflict is not resolved, then the conflict cannot be resolved by any possible assignment to wires not in C.*

Proof: Assume that instead of indexed backtracking, normal backtracking is performed. At some stage when the conflict has not yet been resolved, the wires both in and outside the set C have some value. Now assume that the conflict is resolved by assigning a wire not in C a different value. Since the conflict gets resolved, the wire (call it K) must have affected the values of the secondary wires of the conflict or the co-fanin of W. In that case, by the rules used for forming the set C, wire K should also be in C, which is a contradiction. ■

For example, consider the circuit of Figure 3.8. It is required to justify the values $G = 20$, $H = 16$, $J = 0$. The choices made are shown in the decision tree with the corresponding index associated with the decisions and the wires. Note that to justify the value on wire J, wire I can assume more than one value (as long as it is greater than the value on wire F) and is therefore not bound by the value on wire F. When wire C is assigned the value -4 to justify the value on wire E, a conflict is detected on wire C. Note that wire C is not a free wire as its value is forced by the values assigned to wires B and E. This is an example where a wire is bound by more than one wire, and such wires are often the sources of conflicts. The wires involved in the conflict are C (the primary wire of the conflict), wire B (as it is the co-fanin of C), wire A (same decision index as B), wires E and F (secondary wires of the conflict), and wires G, H and J (fanouts of E and F). Wire I is not chosen as it does not bound the value of F. Note that wire C is assigned the index 2, which is the largest of the indices of its fanout and co-fanin wires. The index list corresponding to these wires is also shown in the figure. The procedure first backtracks to the node with index 2, as shown in the figure, avoiding the node with index 3 which has nothing to do with the conflict.

Consider another situation where the values on wires G, H and J are 20, 16, and 1 respectively. Assume that the decision tree is the same as shown in Figure 3.8 but with wire I assigned the value -1 to justify the value on wire J. When the justification process moves back one level to assign a value on wire C, a conflict is detected and resolved using indexed backtracking as illustrated in the previous paragraph. The value assigned to wire F is now -2. This gives rise to a conflict on wire F, as it is required to have a value less than that of wire I. Now indexed backtracking can be used again to resolve this conflict. This is an example of how the resolution of one conflict can give rise to another. The resolution of a conflict can give rise to other conflicts or a conflict on the wire that was involved in the first conflict. This indicates that there is a possibility of oscillation. However, each time new values are assigned to the nodes in the decision tree and the number of different values at each node (on each wire) is finite. Therefore, the backtracking process will either converge to an assignment that resolves the conflict or stop once no such assignment can be found.

3.6 Conflict Resolution

Consider the RTL description of Figure 3.2. Assume that wire G has the value 20 and wire H has the value 16. In the first step of justification, let the values assigned to wires A, B, E, and F be 0, 20, 16, and 0 respectively. In the next step wire C has to be assigned a value to justify the values on wire E and F. To justify the value on E wire C must have the value -4, while to justify the value on F it has to have the value 0. This is a conflict. At this point the justification procedure is forced to backtrack on the assignment of a value to wire E and ultimately reach a justifiable assignment of $E = 18$. Since the number of values on wire E can be large, backtracking can take a long time. This conflict can be resolved efficiently by noting that all the modules involved in the conflict are arithmetic and their interdependence can be represented by the following system of equations:

$$B + C = E$$

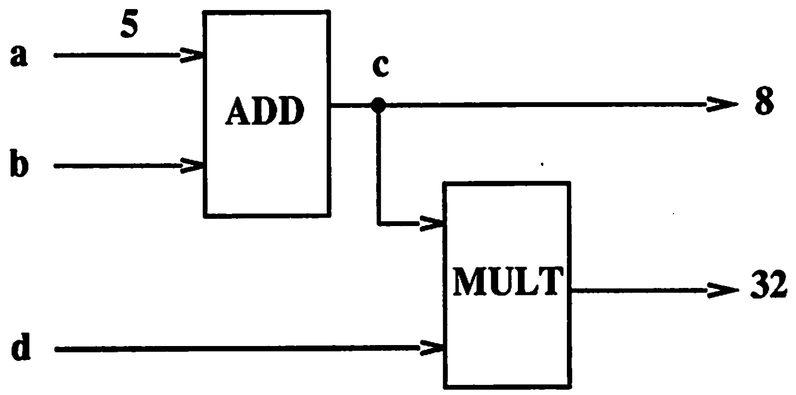


Figure 3.9: Circuit illustrating conflict resolution

$$\begin{aligned} C &= F \\ E + F &= H \\ B &= 20 \\ H &= 16 \end{aligned}$$

This system of equations can be assembled *only* if the modules involved in the conflict are arithmetic. In this example, the system of equations is linear, but in general, the system of equations is non-linear and is solved using Newton-Raphson (NR) methods [58]. The process of solution is standard and will not be discussed here. The equations are assembled using techniques used in circuit simulators like SPICE [106].

3.6.1 Assembling the equations

The first step in conflict resolution is to identify the wires involved in the conflict. The procedure for doing this is the same as outlined in Section 3.5. The modules involved in the conflict are defined to be modules whose input wires and output wires are wires involved in the conflict. Wires are said to be *boundary wires* of the conflict if they are not both an input as well as an output for some modules involved in the conflict. If all the modules involved in the conflict are arithmetic and their operation can be represented algebraically using an operator, only then will the system of equations be assembled. Information regarding the suitability of a module for conflict resolution comes from the user and is stored in the function-type field of the module. Having identified the wires and the modules, a matrix called the *coefficient matrix* is built. This is a square matrix. The number of columns in the matrix is equal to the number of wires (n) involved in the conflict, while the number of rows is equal to the number of modules involved in the conflict plus the number of boundary wires that have been assigned values. Each wire is assigned a variable and the variable is assigned a column in the matrix. Together with the coefficient matrix, another $n \times 1$ matrix called the *Right Hand Side (RHS)* is constructed. For boundary wires having a value assigned to them, an equation of the form

$$variable = constant$$

is formulated. For each equation a row is allocated in the coefficient and the RHS matrices. For the equation above, all the columns in the row of the coefficient matrix are 0 except the one for the variable in the equation, which is given the value 1. The entry in the RHS matrix is the constant. For modules, the equation is of the form

$$f(var_1, var_2, \ldots, var_n) = constant$$

and another row is allocated in the matrix, and the entry in each column is the partial derivative of the function f with respect to the variable corresponding to the column. These partial derivatives can be pre-computed and used as *stamps* to form the coefficient matrix, as in circuit simulators like SPICE [106].

The derivation of the *stamps* will be illustrated with the help of the circuit of Figure 3.9. The system of equations representing this circuit is:

$$\begin{aligned} a+b-c &= 0 \\ cd-32 &= 0 \\ c-8 &= 0 \\ a-5 &= 0 \end{aligned}$$

The variables are ordered as a, b, c, d; a is assigned the first column in the matrix, b the second, and so on. This system of equations can be represented as $F(x) = 0$, where x is the vector of values on the wires. The formula for Newton-Raphson iteration is:

$$J(x^k)\,(x^k - x^{k+1}) = F(x^k)$$

where x^k is the vector of values on the wires after the k^{th} iteration, and the values to be computed are x^{k+1}. $F(x^k)$ is the value of $F(x)$ and $J(x^k)$ is the coefficient matrix (also called Jacobian matrix) after the k^{th} iteration. Applying this to the system of equations above, the following linearized system of equations is obtained:

$$\begin{bmatrix} 1 & 1 & -1 & 0 \\ 0 & 0 & d^k & c^k \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a^k - a^{k+1} \\ b^k - b^{k+1} \\ c^k - c^{k+1} \\ d^k - d^{k+1} \end{bmatrix} = \begin{bmatrix} a^k + b^k - c^k \\ c^k d^k - 32 \\ c^k - 8 \\ a^k - 5 \end{bmatrix}$$

On inspection of this system of equations, the stamps for the adder and the multiplier can be easily identified. For the adder whose two inputs are v_p and v_q and whose output is v_r, the stamp in the coefficient matrix is:

$$\begin{bmatrix} N_p & \dots & N_q & \dots & N_r \\ \vdots & & \vdots & & \vdots \\ 1 & \dots & 1 & \dots & -1 \\ \vdots & & \vdots & & \vdots \end{bmatrix}$$

The stamps for other modules like multipliers can be formed accordingly.

In the case when the system of equations is non-linear, and there are no guarantees about the existence and the uniqueness of the solution. The initial guess (which is zero) in the NR-iteration is very important in reaching a solution. Despite these uncertainties, in most cases, the system of equations can be solved and the conflict resolved without backtracking. Note that inequalities cannot be handled, and modules like comparators, though arithmetic in nature, cannot be handled during conflict resolution. For circuits that are purely arithmetic, the conflict resolution procedure can be used for implication in the circuit, as PI values can be directly implied from PO and NS values.

3.7 State Differentiation

Combinational test generation produces fault-free/faulty state pairs. It is assumed that both the fault-free and the faulty states are states in the fault-free machine and a differentiating sequence is sought for that pair of states. Usually, due to the large number of unknown values on the primary inputs and present state wires in the combinational test vector, the values on some of the next state wires are unknown. In the general case therefore, it is necessary to find a differentiating sequence between groups of states rather than a minterm state pair. However, fault simulation of the justification sequence produces a minterm state pair to differentiate. If the state pair cannot be differentiated using the procedure described in this section, then the fault-free/faulty state pair corresponds to equivalent states in the fault-free machine. Such state pairs are stored in the set of equivalent states.

Fault-free differentiating sequences are generated using the RT level description of the circuit. Fault simulation or combinational test generation produces the true/faulty state on the next-state wires. The first step is to transfer these values to the corresponding present-state wires. The values are now translated to integer values to obtain a fault-free present state as well as a faulty present state at the RT level. During translation, a D on any present state wire is given the value 1 in the true circuit and 0 in the faulty circuit. The opposite is done for a $\overline{D}$, where D and $\overline{D}$ are used to represent true/faulty values on wires at the gate-level, as in the D-Algorithm [115].

The state differentiation procedure is shown in Figure 3.10. It uses the fact that if an input combination exists for which a particular output depends on the value of a PS wire, then if the effect of the fault is propagated to that PS wire, the true and faulty states can be differentiated in one time frame. The corresponding input combination is a single-vector differentiating sequence.

A primary output is said to be sensitive to the value of a PS wire if on changing

```
differentiate()
{
  /* Given the NS wires that has true/faulty value */
  if (differentiating input comb. exists for NS wire)
    Store differentiation sequence; return (TRUE);
  for (each PS wire with true/faulty value) {
    for (each output) {
      for (each path from the PS wire to output) {
        Sensitize path;
        simulate assignments;
        justify all values on intermediate wires;
        if (all values justified) {
          Store differentiation sequence;
          return (TRUE);
        }
      }
    }
  }
  /* True/Faulty value cannot be propagated to output */
  for (each PS wire with true/faulty value) {
    for (each NS wire) {
      for (each possible path from the PS to NS) {
        Sensitize path;
        simulate assignments;
        justify all values on intermediate wires;
        if (all values justified) {
          flag = differentiate();
          if (flag) {
            make up differentiation sequence;
            return (TRUE);
          }
        }
      }
    }
  }
  /* Cannot differentiate this state pair */
  return (FALSE);
}
```

Figure 3.10: Procedure for state differentiation

the value of the PS wire, the value of the output changes. The value of the primary inputs and other PS lines for which this happens is called the **sensitizing input combination.** If a sensitizing input combination for a PS wire does not require the other PS wires to have any particular value, the corresponding input combination is called a **differentiating input combination** for that PS wire. If a differentiating input combination exists for a PS wire to which the effect of the fault is propagated, then that input combination is a single-vector fault-free differentiating sequence. For example, for the circuit of Figure 3.2, a differentiating input combination for C is $A = 0, B = 0$. There is no such combination for wire D.

Differentiating input combinations and differentiating sequences are found using similar techniques. If there exists a path from a PS wire to a primary output such that the path can be sensitized by setting the primary inputs only, then the primary input combination is the differentiating input combination. For each path from a PS wire to each primary output, values are set on the side inputs so that the path is sensitized. As soon as a side input is assigned a value, it is stored in a decision tree and this value is justified using a modified version of the procedure **justify_values()** of Section 3.4. If all the values on the side inputs are justified without setting any PS wires to a value, a differentiating input combination is obtained. The side inputs to the path being sensitized are arranged in increasing order, according to their levels. Wires of a lower level, *i.e.*, those closer to the POs are assigned values before wires at higher levels. This reduces the chances of conflict in the later stages of the sensitization process. All the side inputs that are assigned values are stored in a decision tree, and all possible assignments to these wires are looked at in order to find a differentiating input combination or a differentiating sequence. Differentiating input combinations are found before actual test generation begins and can be used for the differentiation of any fault whose effect is propagated to a PS wire with a differentiating input combination. This improves the efficiency of the differentiation algorithm.

If a differentiating input combination does not exist for the PS wire to which the effect of the fault has been propagated, a single-vector differentiating sequence is searched for in a manner similar to the one described above. However, there are some differences. Firstly, during the sensitization of paths, the values on the PS wires have to be taken into account. Also, the side inputs to the path are now arranged in decreasing order, thereby side inputs closest to the primary inputs are assigned values before side inputs further away. This is required to ensure that the effect of the fault is propagated to the next level of the circuit so that the value of the side inputs at that level can be accurately determined. In case a single-vector differentiating sequence cannot be found for the true and faulty state pairs, an effort is made to propagate the effect of the fault to the NS wires. The process is similar to the one that propagates the effect of the fault to the primary outputs. The next-state line to which the effect of the fault has to be propagated is selected heuristically. If a particular PS wire has a differentiating input combination, the corresponding NS wire is chosen. If no PS wire has a differentiating input combination, the one that requires the least number of side inputs to be set to sensitize a path from that wire to the output is a good candidate to which the effect of the fault should be propagated. This is because a small number of side inputs can be assigned sensitizing values more easily than a large number. Once the fault effect has been propagated to the NS lines, an attempt is made to find a single-vector differentiating sequence between the new true/faulty state pairs, *i.e.*, to propagate the effect of the fault from the new PS wire to one of the POs. The algorithm thus attempts to find a single-vector sequence, then a two-vector sequence, and so on.

As an example, consider the circuit of Figure 3.7. Let the true state be $PS1 = 18$, $PS2 = 23$ and the faulty state be $PS1 = 18$, $PS2 = 5$. There is no path from $PS2$ to the primary output. Therefore, the effect of the fault has to be propagated to an NS line. Since $PS1$ has a differentiating input combination (which is $INB = 0$, $CTRL = 1$), the obvious choice for the next-state line to which the effect of the fault should be propagated is $NS1$. There is only one side input to the path, namely, wire $INTB$, which has to be set to 1. With the fault-free state $PS1 = 18$, $PS2 = 23$, the value of $INTB$ is 1, as required. On the other hand, for the faulty state $PS1 = 18$, $PS2 = 5$, the value of $INTB$ is 0. Since the fault-free value on $NS1$ was already determined to be 23, the procedure has to ensure that the faulty value on $NS1$ is different from 23. Therefore, $INTA$ is set to a value different from 23. Say the value chosen is 16. Now the procedure **justify_values()** is called to justify the value on wire $INTA$, which is easily done by assigning $INB = -2$. Therefore, a two-vector differentiating sequence is $\{[INA = INC = CTRL = -,\ INB =$

$-2], [INA = INC = -,\ INB = 0,\ CTRL = 1]\}$.

For the circuit of Figure 3.2, present-state wire C has a differentiating input combination. But for wire D there is no such combination, and moreover, there is no way to propagate a true/faulty value on wire H to the primary output or any other next-state line. Therefore, all states which have the same value on wire C but different values on wire D are equivalent.

If a differentiating sequence exists in the fault-free circuit, but the sequence is not valid in the faulty machine, the fault is not detected. Therefore, even when a test exists for a fault in the fault-free machine, the fault may be redundant. The opposite is also true, *i.e.*, a test might exist for a fault, but no fault-free differentiating sequence might exist for it.

This procedure involves a depth-first search of the STG of the machine. Therefore, it is best suited for circuits where any state can be differentiated from any other within a few time frames. Once again, most datapath-type circuits meet this condition. In Section 3.8, it will be shown that this approach also works for some circuits that require long differentiating sequences. Potentially, the entire State Transition Graph of the circuit might have to be enumerated in order to verify that the fault-free/faulty state pair are equivalent. Since the STG could be very large, detection of equivalent states might require a large amount of CPU time for some examples. However, experimental evidence gathered to date suggests that in most cases it is possible to detect equivalent states within a reasonable number of time frames. This happens because the fanouts of the fault-free and the faulty states become identical in a few time frames.

3.8 Test Generation Results Using ELEKTRA

The test generation algorithm described in the previous sections has been implemented in the program ELEKTRA. It consists of about 14,000 lines of C code and runs in a VAX-UNIX^{TM} environment.

Results and time profiles using ELEKTRA for six sequential circuits which are described in Table 3.1 are given in Tables 3.2 and 3.3 respectively. In the tables m and s

CKT	#Inputs	#Outputs	#Gates	#Latches	#Total Faults
ex1	130	32	1073	64	3410
ex2	140	96	3422	256	11572
ex3	220	146	5216	480	20198
des	113	64	1439	64	4886
key	258	193	1812	228	5926
viterbi	94	43	1717	640	5202

Table 3.1: Statistics for example circuits

stands for minutes and seconds respectively. For each example in Table 3.1, the number of inputs (#Inputs), number of outputs (#Outputs), number of gates (#Gates), number of latches (#Latches), and number of faults for which tests have to be generated (#Total Faults) are indicated.

Example **ex1** and **ex2** are datapath portions of digital signal processing chips. Example **ex3** is a datapath and a controller for a digital signal processor. Examples **key** and **des** are data encryption standard chips described in [129]. Example **viterbi** is a word processing chip that forms a part of a speech recognition system [125].

In Table 3.2, the total number of test vectors (#Vect), the maximum test sequence length (Max. seq. len), the percentage of times when a potential test sequence generated for a fault actually detected the fault (%Success), the fault coverage, the percentage of provably redundant faults, the total fault coverage including detected and provably redundant faults (TFC), and the test pattern generation (TPG) time on a VAX 11/8800 are indicated for each example. The quality of test patterns is determined by the total amount of test data (bits) that have to be stored in the tester and the time required to apply these vectors to the circuit under test. The total number of test vectors suggest that the amount of test data is always within reasonable limits even for the large examples.

CPU times for justification, differentiation, total test generation, fault simulation, miscellaneous setup operations, and for the entire test generation process are given in Table 3.3. The test generation time includes the time required for justification and differentiation. It is noteworthy that justification and differentiation takes, in most cases, a small fraction of the total test pattern generation time using the algorithms described in

CKT	#Vect	Max. seq. len.	Success (%)	Fault Cov. (%)	Red. Fault (%)	TFC (%)	TPG time
ex1	208	4	100	100	0	100	226.7s
ex2	590	4	100	100	0	100	129m
ex3	1033	4	100	100	0	100	316m
des	202	6	100	99.4	0.6	100	256m
key	203	9	96	100	0	100	23m
viterbi	2045	8	99	99.8	0.2	100	674m

Table 3.2: Test generation results for circuits

CKT	Justify	Differ	Test Generation	Fault Sim	Misc. Set up	Total Time
ex1	0.49s	0.36s	9.17s	212s	5.53	226.7s
ex2	7.47s	3.23s	102s	126m	38s	129m
ex3	11.3s	8.5s	300s	309m	120s	316m
des	36m	0.0s	40m	210m	300s	256m
key	3m	2m	8m	14m	43s	23m
viterbi	40m	3m	462m	202m	10m	674m

Table 3.3: Time profiles for example circuits

this chapter. The fraction of time spent in justification and differentiation is small, much smaller than the corresponding time required in STEED. Moreover, there is no need for cover enumeration. Test generation times are in most cases small, and fault simulation dominates the total test generation time. A better fault simulation algorithm can decrease the time required for test pattern generation. It is noteworthy that the success rate, *i.e.*, the fraction of times that a potential test sequence is valid, is close to 100% for most of the examples.

In Table 3.4, total test generation time and fault coverage of STEED are compared with ELEKTRA. As can be seen, the algorithm described in this chapter obtains close to the maximum possible fault coverage in all the examples. It takes significantly smaller time than STEED to achieve the same fault coverage. For some of the examples, very significant speed-ups, up to a factor of 100 were obtained.

To determine the quality of the test patterns, the time required for the application of the test patterns has to be evaluated. In Table 3.5, the times for testing the sequential

CKT	STEED		ELEKTRA	
	Fault Cov.	TPG time	Fault Cov.	TPG time
ex1	85.68	> 240m	100	226.7s
ex2	55.32	>1200m	100	129m
ex3	40.44	>1200m	100	316m
des	97.93	1423m	100	256m
key	100	256m	100	23m
viterbi	88.95	>1200m	100	674m

Table 3.4: Comparisons with STEED

CKT	#Scan Design Testing Cycles	#Non-scan design Testing Cycles
ex1	4032	208
ex2	29696	590
ex3	55680	1033
des	8768	202
key	11856	203
viterbi	15168	2045

Table 3.5: Clock cycles needed for testing

circuits using the Scan approach and the non-scan approach used here are compared. The testing time required for a non-scan implementation is significantly smaller than a Scan implementation. This is because values have to be clocked in and out of flip-flops sequentially in a Scan design and each test vector takes multiple clock cycles to apply. The larger the number of flip-flops in a Scan chain, the longer the testing time. This time can be reduced by using multiple Scan paths but at the cost of higher overhead. On the other hand, in non-scan testing, each vector takes only one clock cycle to apply. For the examples above, ELEKTRA generates test sequences that require a factor of 7 to 58 smaller testing times than compacted, combinational tests applied using Scan design techniques.

3.9 Conclusions

A new approach to test generation for sequential circuits was presented in this chapter. The key ideas of the previous chapter were used together with the exploitation of the arithmetic properties of many RTL descriptions to design a test pattern generator that can handle larger circuits than previous approaches, and which is more efficient. Good heuristics play a major role in the efficiency of this algorithm. It was possible to generate tests for an entire chip of medium complexity.

Despite these advantages, there are certain drawbacks. This approach works best for circuits whose STGs have a strong connectivity, and there are various paths from one state to another. Also, this approach is not suitable for circuits without arithmetic modules. In addition, some arithmetic modules (*e.g.*, multipliers with round-off ability) cannot be handled. For pure controllers, the approach described in the previous chapter may be better. Due to the nature of fault-free justification and differentiation, it might be necessary to operate strictly at the logic level to generate tests for certain faults. Also, this approach requires that the Computer-Aided Design (CAD) system used to design the chip maintains and relates the RTL and logic-level descriptions. However, this requirement is easily met in most modern CAD frameworks.

Despite having more powerful test generators, the problem of testing is not fully solved. If a circuit has a large number of redundant faults, even a very good test generator might spend a significant amount of time in identifying those faults. The synthesis process can help in easing the task of test generation by synthesizing the circuit to be fully and easily testable. Though it is difficult to characterize easy testability and make the circuit easily testable, there are procedures to synthesize circuits to be fully testable. One such procedure is the topic of the next chapter.

Chapter 4

SEQUENTIAL SYNTHESIS FOR TESTABILITY

Test generation for VLSI circuits has traditionally been a post-design activity. In most cases, test generators like those described in the previous chapters are used after the design is complete. Most circuits are designed with only area and performance under consideration, and quite often, the circuits are not fully or easily testable. Therefore, a significant amount of effort might be spent in generating tests for such circuits. Furthermore, design for testability approaches, such as *Level Sensitive Scan Design* (LSSD [56]), have been primarily used to transform the difficult sequential test generation problem into a more tractable problem of combinational test generation by enhancing the *controllability* and *observability* of the sequential circuit. Such approaches have an associated area or performance penalty or both, because the latches used are special scan latches, which are larger than simple non-scan latches.

Synthesis for testability approaches attempt to completely integrate the tasks of logic design, optimization, and test generation for combinational as well as for sequential circuits. The goal of synthesis is to meet area, performance, and testability requirements. Ideally, test vectors to detect the targeted faults in the circuit are obtained as a by-product of the synthesis step. Testability-directed synthesis strategies for combinational circuits are well developed (*e.g.*, [13, 43, 68]). It is well known that for a two-level *prime* and *irredundant* circuit, all single stuck-at faults are testable. Similar results are also known for multi-level circuits. Logic synthesis systems like MIS [15] and BOLD [57] can produce fully testable two-level or multi-level combinational circuits, for single as well as multiple stuck-at faults. Various approaches for synthesis of fully testable non-scan sequential circuits (*e.g.*, [44, 48, 49, 50]) have been proposed in the past. These approaches use logic optimization under *don't-cares* to derive a fully testable implementation of the circuit.

Previous approaches like [44, 48, 49, 50] for synthesis of fully testable non-scan sequential circuits have used State Transition Graph descriptions of finite state machines. These approaches are therefore limited in applicability to small controller circuits that have small STGs. Datapath-controller circuits, as well as communication chips and digital signal processors have STGs that require too much memory to store and are too cumbersome to manipulate. For such circuits, a different synthesis for testability approach is needed.

In this chapter, a synthesis for testability approach is presented that can be used for large sequential circuits. This approach uses *covers* or a *Register-Transfer Level description* of a sequential circuit instead of State Transition Graphs. The rest of this chapter is organized as follows. Terminology is introduced in Section 4.1. Previous work is briefly reviewed in Section 4.2. Some essential theoretical results are presented in Section 4.3. The overall synthesis for testability strategy is outlined in Section 4.4. The identification of invalid and equivalent state don't-cares is the topic of Sections 4.5 and 4.6. Experimental results are presented in Section 4.7, with conclusions in Section 4.8.

4.1 Preliminaries

A general synchronous sequential circuit is shown in Figure 2.1. The behavior of such a circuit is often represented using a State Transition Graph. An example State Transition Graph is shown in Figure 4.1. This machine has five states. In an implementation of the machine, each state is assigned a binary code called the *state code*. Since there are five states, at least three bits have to be used to assign codes to the states of the machine. However, three bits can be used to assign eight state codes, which means that in this machine three state codes are not used. As in the previous chapters, assume that the machine has

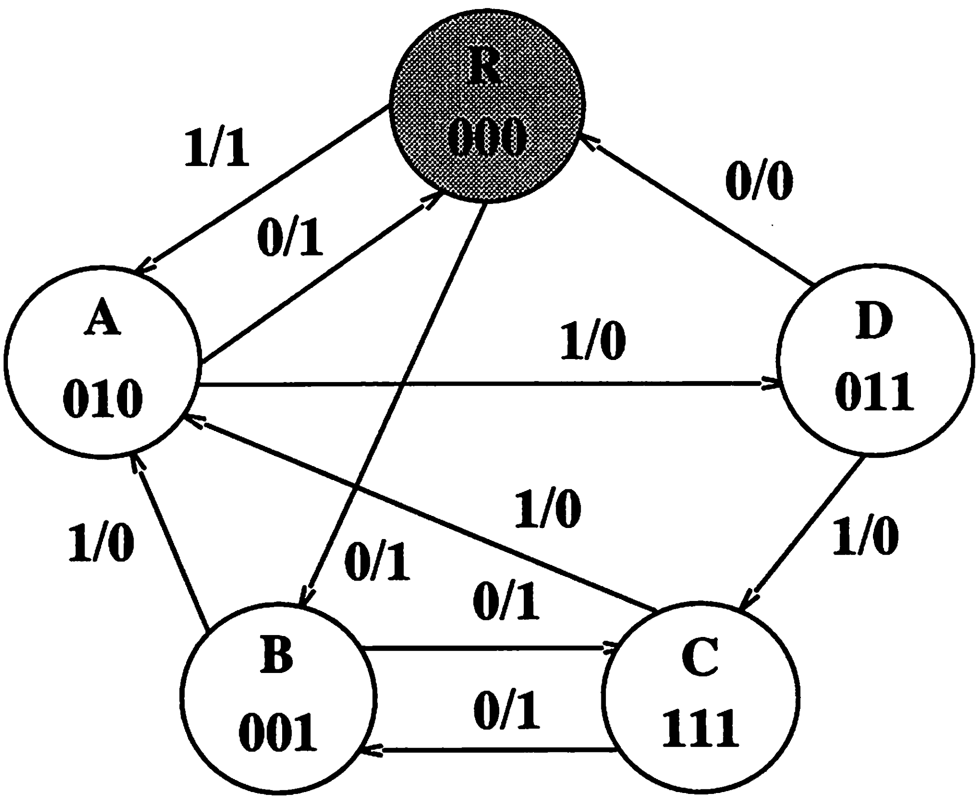


Figure 4.1: An example STG

a reset state, state *R* in the figure. In any correctly functioning implementation of the machine, only the five states given in the STG of Figure 4.1 are visited. However, the three other states exist in the logic-level implementation of the machine. Assume that the entire STG of the implemented machine is as shown in Figure 4.2. A state in this STG is said to be **valid** if it is either the reset state or it can be reached from the reset state given that the machine is functioning properly. Otherwise, the state is said to be **invalid**. The portion of the STG consisting of all the valid states and all edges emanating from the valid states is called the *valid portion* of the STG. The rest of the STG is called the *invalid portion*. For all possible implementations of a fully specified machine, the valid portion of the STG is the same but the invalid portion might depend on how the machine is synthesized. For the

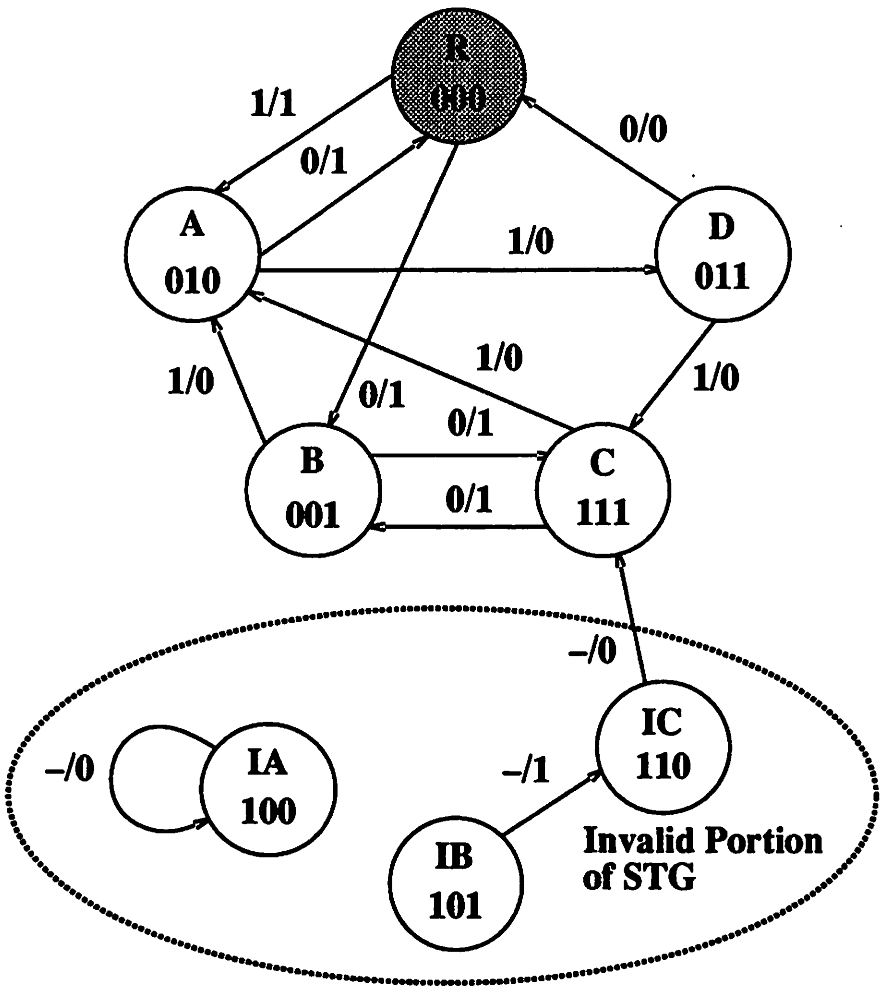


Figure 4.2: The STG of an implemented machine

STG of Figure 4.2, the invalid portion of the STG is explicitly shown.

A **differentiating sequence** for a pair of states (S_1, S_2) in a sequential circuit is a sequence of input vectors such that if the sequence is applied to the circuit when the circuit is initially in S_1, the last vector in the sequence produces a different primary output combination than if the circuit were initially in S_2. If there is no differentiating sequence for the pair of states (S_1, S_2), then they are said to be **equivalent**. In the machine of Figure 4.2, states B and C are equivalent.

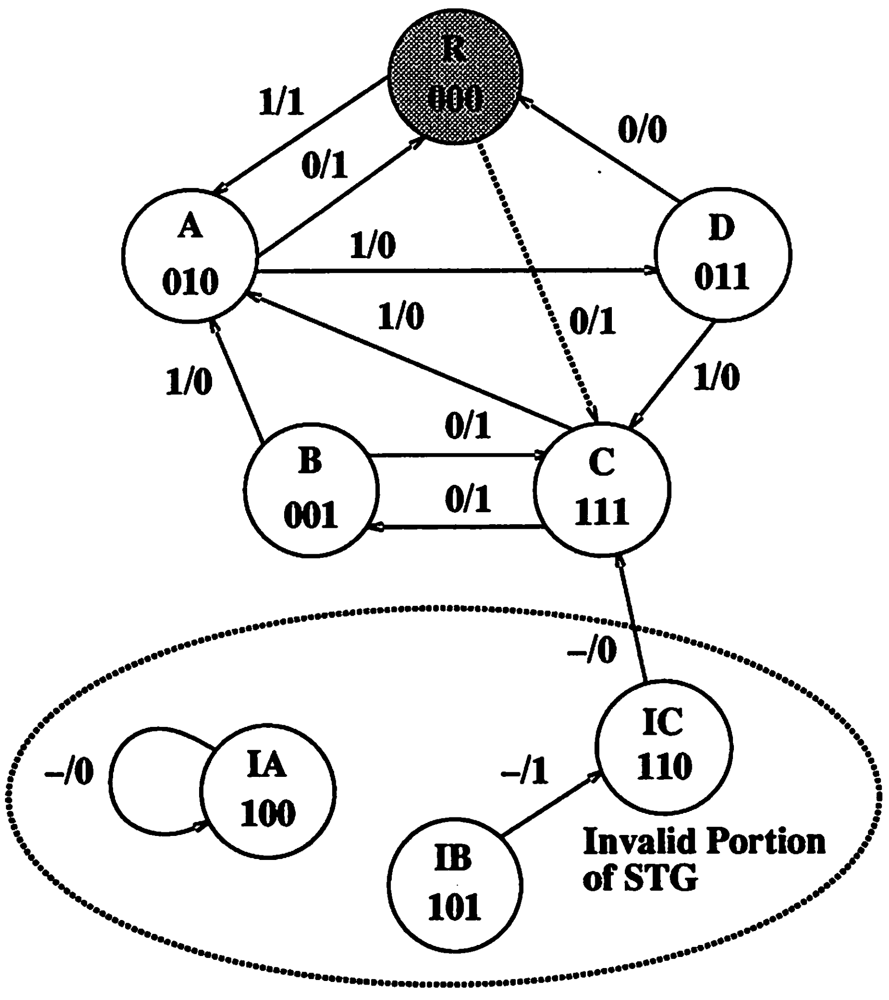


Figure 4.3: Equivalent-SRF

Faults in a sequential circuit may occur on the inputs or outputs of the combinational logic block or may occur inside the combinational logic block. Faults on the inputs and outputs are either primary input faults, present-state faults, primary output faults, or next-state faults. All other faults are called **internal** faults. In the following paragraphs, some terms that were mentioned briefly in Chapter 2 are defined.

A fault in a sequential circuit is said to be **redundant** if the behavior of the fault-free and the faulty circuit is identical under all possible input sequences. Redundant faults can be divided into two categories — **combinationally-redundant** and **sequentially-redundant**. A combinationally-redundant fault (CRF) is one whose effect cannot be observed at the PO or NS wires using any primary input and any present state. Synthesizing the combinational logic block of the circuit to be prime and irredundant can ensure that there are no CRFs. Sequentially redundant faults can be classified into three categories as in [49]:

1. **Equivalent-SRF**: The fault causes the interchange/creation [1] of equivalent states in the STG.

2. **Invalid-SRF**: The fault does not corrupt any fanout edge of a valid state in the STG.

3. **Isomorph-SRF**: The fault results in a faulty machine that is isomorphic to the original machine but with a different state encoding.

Consider the circuit whose STG is shown in Figure 4.2. Let a fault modify this circuit to another one having the STG shown in Figure 4.3. The corrupted edge is shown using a dotted line. In the faulty machine the corrupted edge goes from state R to C instead of B. Since states B and C are equivalent, the true and faulty machines behave identically. This is an equivalent-SRF. On the other hand, if the fault were to change the STG to that of Figure 4.4, then only the invalid portion of the STG would be corrupted. Since the invalid portion of the STG is never visited, this fault will never be exerted and therefore it is redundant. Such a fault is called an invalid-SRF. The third kind of SRF, namely, isomorph-SRF, is illustrated with the STG of Figure 4.5. In this figure, the state codes of state B and D and that of C and IC are interchanged.

In addition to these SRFs, there is a complicated equivalent-SRF. Consider a portion of an STG shown in Figure 4.6(a). In this STG state B is an invalid state and is not equivalent to state C. Let a fault modify only the same portion of the STG to that shown in Figure 4.6(b). The fault-free state C in the fault-free machine is equivalent to the faulty state B in the faulty machine. Therefore, the fault is redundant. In [49], it is shown that apart from the three kinds mentioned above, no other kind of sequential redundancy can exist.

[1]Replacement is included as a form of interchange.

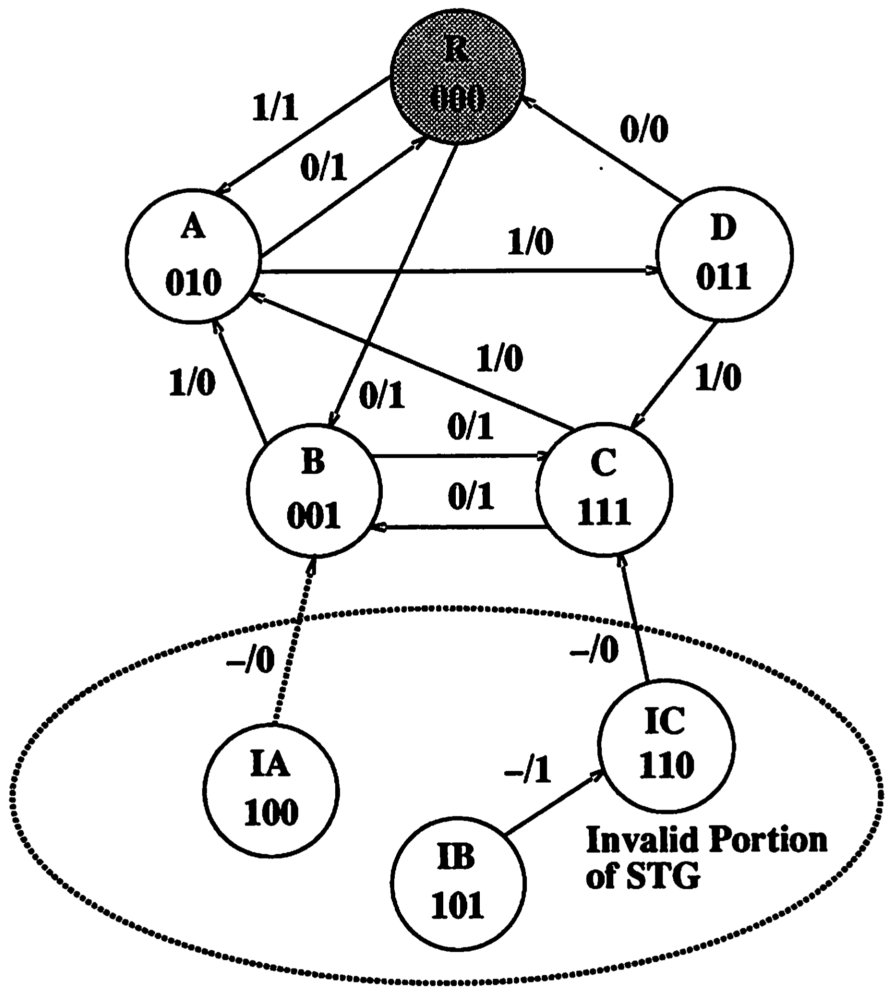


Figure 4.4: Invalid-SRF

Complicated equivalent-SRFs as well as isomorph-SRFs are very rare. In fact, in all practical circuits seen so far, not a single instance of these faults have been encountered. On the other hand, invalid-SRFs and simple equivalent-SRFs are very common. The approach to be described here is therefore restricted only to these SRFs. Full testability therefore means that there are no invalid or simple equivalent-SRFs in the machine.

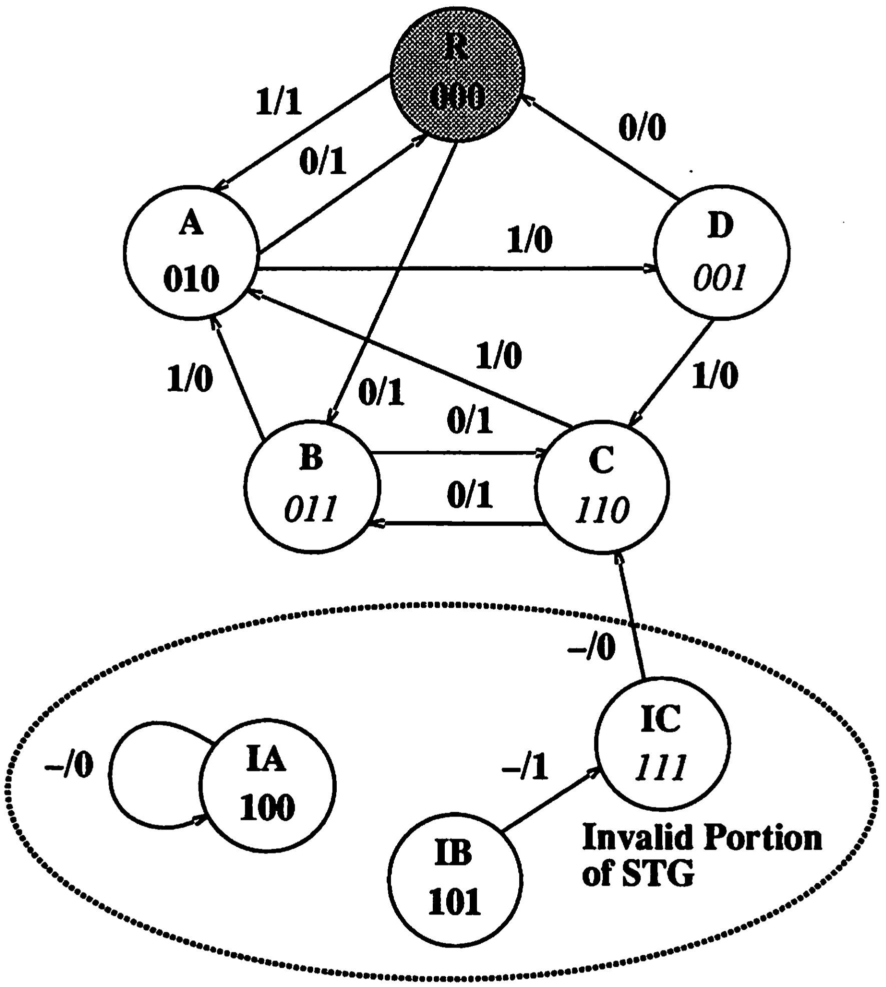


Figure 4.5: Isomorph-SRF

4.1.1 Eliminating Sequential Redundancies Using Don't-Cares

In general, in an n-latch machine, there are invalid states as well as equivalent states. To eliminate invalid-SRFs it has to be ensured that no fault requires only invalid states as excitation states. This can be achieved using invalid states as don't-cares during combinational logic optimization [49]. For example, a two-level cover of the combinational logic block implementing the machine of Figure 4.1 is shown in Figure 4.7. The use of invalid

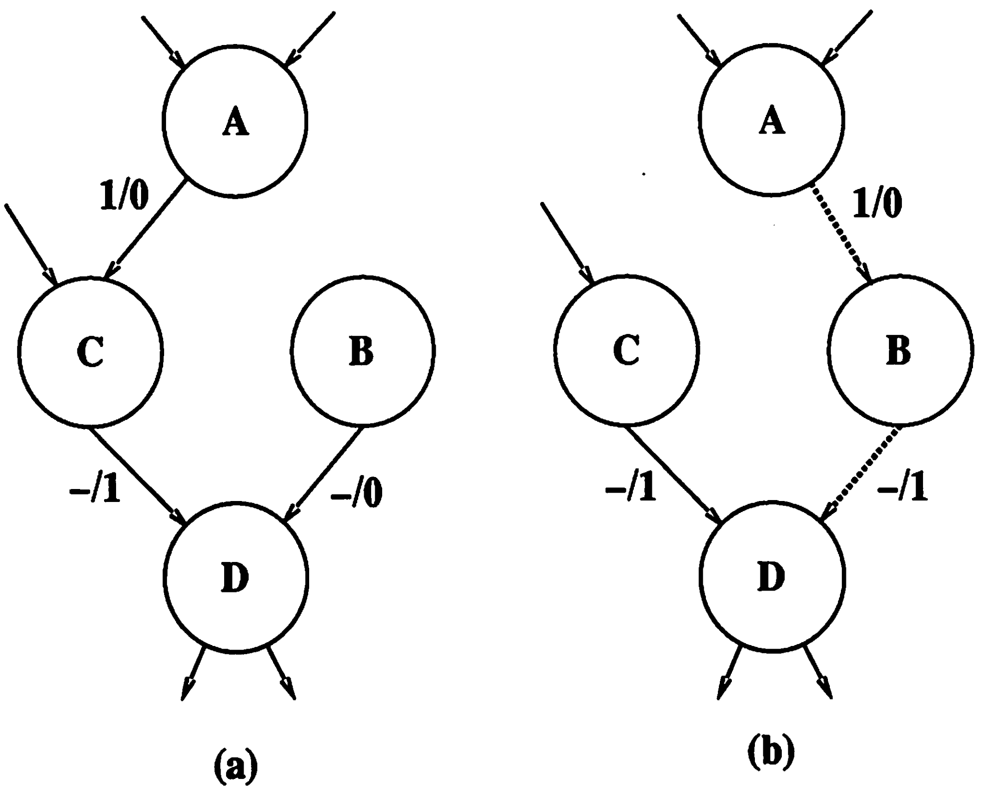


Figure 4.6: Complicated equivalent-SRF

states as don't-cares is illustrated in the last three lines of the cover. Making a network prime and irredundant *under* a don't-care set ensures that there exists a test vector outside the don't-care set. That is, there will exist at least one valid state for each fault that will excite and propagate the effect of the fault to the next-state lines and/or primary outputs.

In many cases, the effect of a fault is propagated only to the next-state lines of a sequential circuit. In such cases a fault-free/faulty state pair is obtained. In order to detect the fault it is necessary to be able to differentiate between the fault-free/faulty state pair. To ensure that there are no equivalent-SRFs, it is necessary to ensure that some faulty state in the faulty machine is not equivalent to the corresponding true state in the fault-free machine. This is achieved by using an equivalent state don't-care set [49] or Boolean

PI	PS	PO	NS
1	000	1	010
0	000	1	001 $\rightarrow$ {001, 111}
1	010	0	011
0	010	1	000
1	001	0	010
0	001	1	111 $\rightarrow$ {001, 111}
1	111	0	010
0	111	1	001 $\rightarrow$ {001, 111}
1	011	0	111 $\rightarrow$ {001, 111}
0	011	0	000
Invalid State Don't-Cares			
-	100	-	- - -
-	101	-	- - -
-	110	-	- - -

Figure 4.7: Two-level cover of the FSM

relations [18]. In the example of Figure 4.1, states B and C are equivalent. For every edge that fans in to state B or C, the next-state function could either produce state B or C, *i.e.*, the next state for the corresponding present state is a set of states, not a single state. In this example, a Boolean relation has to be minimized. Minimization of Boolean relations is discussed in Chapter 7. If the state codes of B and C were different in only one bit, then they could have been combined into one cube and a Boolean function with output don't-cares would have to be minimized, as in [49]. Using these don't-cares or minimizing the associated Boolean relation guarantees the existence of a test vector outside the don't-care set. That is, for each fault there is a fault-free and faulty state that are not equivalent.

The two main issues in using don't-cares for synthesis are the efficient derivation of the don't-cares and the use of the don't-care sets in synthesis. If the number of latches in a machine is large, then even a listing of all invalid states may not be possible, let alone optimization under the don't-care set. Similarly, explicitly detecting and listing equivalences between all pairs of states is virtually impossible. Therefore, STG-based techniques for synthesis for testability will fail for such circuits. In Sections 4.5 and 4.6 it will be shown how to find a subset of invalid and equivalent state don't-cares that will ensure that the circuit will have no equivalent or invalid-SRFs. This don't-care information is extracted using the covers of PO and NS lines or the RTL description of the circuit.

4.2 Previous Work

In [48], a procedure is given for implementing fully and easily testable finite state machines. This approach represents approaches that constrain the synthesis procedure by restricting the implementation of the logic. In this approach the next-state logic block is separated from the output logic block. Also, the logic for each next-state line is implemented as a single cone. Using a *distance-2* state assignment strategy, it is shown that the resultant circuit is fully testable. The attractive feature of this approach is that the resultant circuit is very easy to test and the length of each test sequence is bounded. An obvious drawback of this approach is that the resultant logic is usually larger and the area of the machine might increase significantly. In a related work, a synthesis method for easily testable PLAs using a crosspoint fault model is proposed in [45].

The classification of redundancies presented in the previous section was first presented in [49]. It is shown that all SRFs are one of the three kinds discussed. This classification serves as a basis for approaches that use optimal synthesis procedures to guarantee full testability, instead of constraining the synthesis process. The STG of the machine to be synthesized is used to derive the don't-care sets. A three-step synthesis procedure is outlined. The first step is state minimization, the second is state assignment, and the third one is combinational logic optimization under a proper don't-care set. *Locally optimal* state assignment is introduced, and it is shown that a locally optimal state assignment can guarantee that no single stuck-at fault is an isomorph-SRF. An iterative synthesis procedure is used to synthesize a fully testable implementation of the circuit. In the first step of the procedure the equivalent don't-care set is empty and only the invalid state don't-cares are used. In each iteration the circuit is synthesized to be combinationally prime and irredundant under the don't-care sets. Now all invalid states that are equivalent to valid states are identified and are used as don't-cares. Therefore, in the next iteration, some previously invalid states might become valid and vice versa. The new invalid and equivalent state don't-cares are extracted and the same set of operations is repeated. This carries on until the circuit doesn't change from one iteration to the next. The resultant circuit can be proved to be fully testable, and it can also be shown that the procedure converges. The fundamental ideas introduced in this paper are used in this chapter. The major practical drawback of this approach is the use of the STG for the purpose of synthesis. As has been pointed out already, the STG of certain circuits like datapath-controllers and digital signal processors can be huge, thereby precluding the use of this approach.

In [44], a unified approach for the synthesis of testable sequential circuits is given. Equivalent-SRFs are identified as the most difficult fault to remove, and the concept of *fault-effect disjointness* is introduced. Procedures for retaining fault-effect disjointness are used to ensure that all fault-free/faulty state pairs can be differentiated, thereby guaranteeing that there are no equivalent-SRFs. The results of [49] are extended and the relationship between redundancies and don't-cares are further explored in [50]. A classification of redundancies in circuits consisting of single or interacting finite state machines is proposed. For each class of redundancies, don't-care sets are defined which if properly exploited during logic synthesis can implicitly remove the redundancies. The steps followed by the synthesis procedure are similar to those of [49].

4.3 Theoretical Results

Before describing the synthesis procedure, it necessary to lay the theoretical groundwork for some results that will be used in the synthesis process. The first result in this section concerns a special class of circuits and a logic-partitioning strategy such that the circuits can be synthesized to be fully non-scan testable through well known procedures for synthesizing combinational circuits to be fully testable. The second part of this section deals with strategies that can be used to prune the size of the don't-care sets necessary to ensure full testability. These strategies exploit the already existing partitions in the logic.

4.3.1 An Unconditional Testability Theorem

Logic partitioning strategies have been used in the past to synthesize more testable circuits. It is possible to synthesize a certain class of circuits to be fully non-scan testable by partitioning the next-state and output logic blocks and ensuring that each logic block is free of CRFs. The following theorem provides the theoretical foundation for such a synthesis strategy.

Theorem 4.3.1 : *Given a sequential machine with N latches and 2^N valid states like the one shown in Figure 4.8, if each state can be differentiated from every other state using a single vector, then the machine is fully sequentially testable, provided that each combinational logic block is fully testable.*

Proof: Since both combinational logic blocks are fully testable, no CRFs can exist. Now it has to be shown that for each fault a test can be generated. Since the fault is not a CRF, an excitation state can be obtained for the fault. Since all states are valid, this excitation state is definitely a valid state, and therefore the excitation state can be justified. Assume that the fault is in the output logic block. In that case the effect of the fault can be observed at the POs by a test vector consisting of the justification sequence and the excitation vector. If the fault is in the next-state logic block, a fault-free/faulty state pair will be produced and the two states will differ in at least one bit. Since every state is differentiable from every other by just one vector, the output logic block can be made to assert different outputs for the individual states in any state pair. Therefore, the fault-free/faulty state pair can be differentiated with one vector and the effect of the fault can be propagated to the POs. ■

As a first step in testability-directed synthesis, the circuit is checked to see if it satisfies the properties required by Theorem 4.3.1. This information has to be provided by the user of this tool. These conditions are satisfied in some pure datapath-type circuits. If the conditions are satisfied, the combinational logic block is partitioned and each block is synthesized to be prime and irredundant, either using test generation algorithms [14] or using synthesis procedures [13]. The resulting sequential machine is guaranteed to be non-scan testable for all single stuck-at faults. No constraints are placed on the synthesis procedure for each block, neither are there any associated don't-care sets.

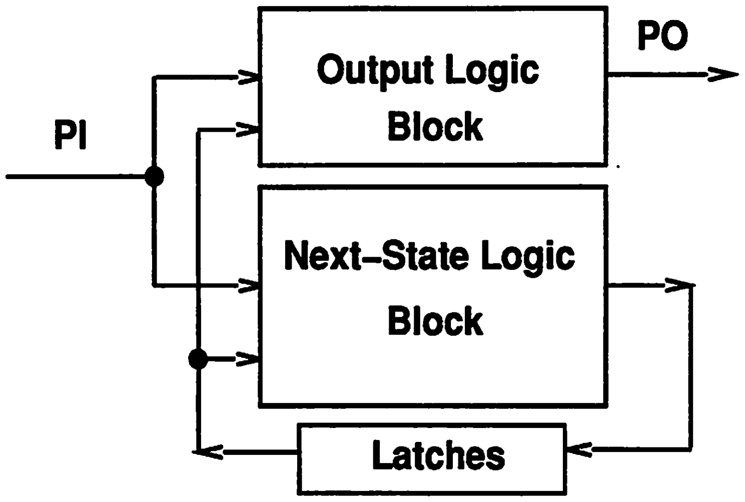


Figure 4.8: An implementation of a sequential circuit

4.3.2 Logic Partitioning

In general, in VLSI designs, logic is partitioned and implemented as separate modules in order to contain the complexity of the design as well to improve the performance of the circuit. A useful side effect of partitioning is that it helps to prune the don't-care sets associated with a sequential circuit. Typically, in a multiple-output circuit, the different outputs are functions of a subset of the inputs. For a primary output O_i, $support(O_i)$ is defined to be the set of inputs that O_i is a function of. In Figure 4.9(a), for instance, $support(F_1) = \{A, B, C\}$.

The logic required to implement a function corresponding to an output may be shared with other outputs or may be separate. In Figure 4.9(a), F_1 and F_2 share logic but F_3 does not. The set of logic gates which implement the function for any output O_i of a multiple-output function is defined as $logic(O_i)$. For example, in the circuit of Figure 4.9(a), $logic(F_1) \bigcap logic(F_3) = \phi$, but $support(F_1) \bigcap support(F_3) \neq \phi$. Any such separation can be exploited to prune the size of the set of equivalent states that is required to be specified as don't-cares to ensure full sequential testability. The following theorems form

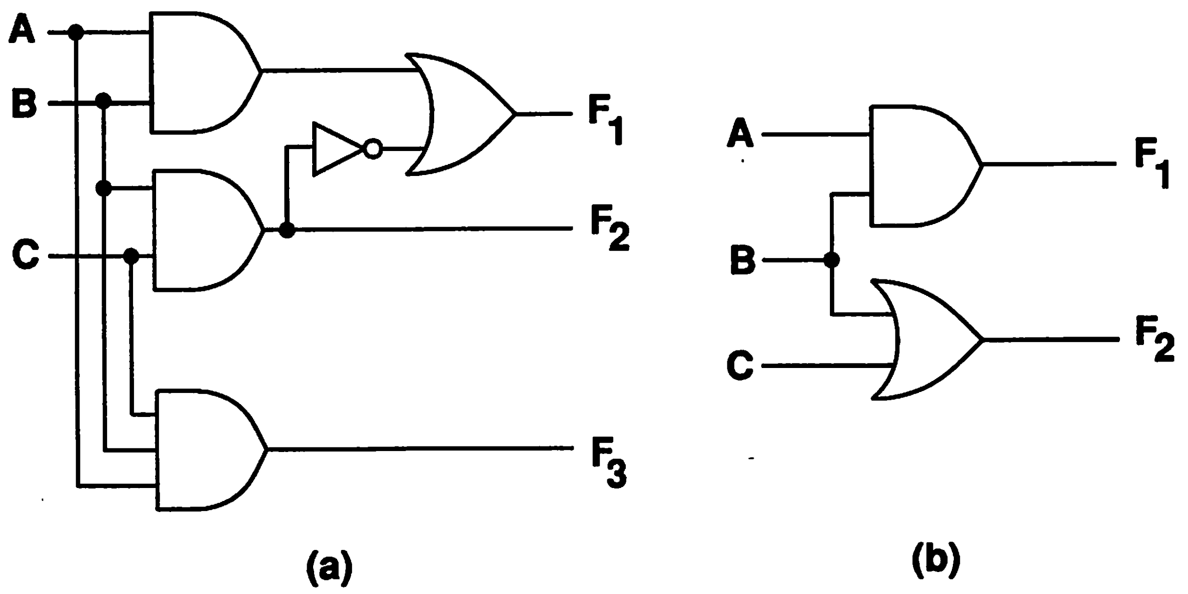


Figure 4.9: Partitioned logic blocks

the basis of the strategy for pruning the set of don't-cares required to synthesize the circuit so that all *internal* faults are testable.

Theorem 4.3.2 : *If two next-state lines N_1 and N_2 are such that they have disjoint logic, i.e., logic(N_1) $\bigcap$ logic(N_2) $= \phi$, then the equivalent state don't-care corresponding to any pair of states s_a and s_b that differ in bit positions corresponding to both N_1 and N_2, is not required to ensure that any internal fault is not an equivalent-SRF.*

Proof: For an internal fault to be an equivalent-SRF, it has to result in an equivalent fault-free/faulty state pair. For any internal fault to be propagated to the next-state lines N_1 and N_2, the fault has to be present both in *logic*(N_1) and *logic*(N_2). Since their logic is disjoint, this cannot occur. Therefore, fault-free/faulty state pairs that differ in bit positions corresponding to *both* N_1 and N_2 will never occur. Therefore, don't-cares corresponding to pairs of states that are equivalent and that differ in these bits are not necessary. ∎

For instance, assume that a machine has three next-state lines N_1, N_2, and N_3, such that $logic(N_1) \bigcap logic(N_3) = \phi$. Then, if state pairs like {110, 011} are equivalent, they need not be considered in the equivalent state don't-care set that is required to ensure testability for internal faults.

It is also possible that two inputs never appear simultaneously in the support of any output in a multiple-output function. For example, in the circuit shown in Figure 4.9(b), $support(F_1) = \{A, B\}$, and $support(F_2) = \{B, C\}$. Inputs A and C never appear in the same support for this multiple-output function. This information can be exploited to prune the invalid state don't-care set. In the following theorem, U_P corresponds to the set of all present-state lines.

Theorem 4.3.3 : *If two present-state lines P_1 and P_2 are such that there does not exist any next-state line N_i or primary output O_i where $support(N_i) \supseteq \{P_1, P_2\}$ or $support(O_i) \supseteq \{P_1, P_2\}$, then the invalid state don't-care set can be limited to cubes that do not have positions corresponding to both P_1 and P_2 set to either 0 or 1, i.e., it can be restricted to combinations on lines $U_P - P_1$ and $U_P - P_2$.*

Proof: This theorem implies that the cubes in the invalid state don't-care set are such that the positions corresponding to P_1 and P_2 in the cube should not have a 0/1 value together. Consider an internal fault in the combinational logic block of the machine. The fault will be present in several $logic(N_i)$'s or $logic(O_i)$'s. To excite and propagate the effect of the fault to the next-state lines or outputs, a specific combination of $support(N_i)$ or $support(O_i)$ is required. It is guaranteed that none of these supports contain both P_1 and P_2. Therefore, the test vector for the fault will have a don't-care entry for either P_1 or P_2 (or both). This implies that to preclude the occurrence of invalid-SRFs, it is necessary to consider only invalid states corresponding to combinations of $U_P - P_1$ or $U_P - P_2$. ∎

For instance, given a finite state machine with three present-state lines, P_1, P_2, and P_3, if P_1 and P_3 satisfy the conditions of the theorem above, then it is not necessary to consider invalid states corresponding to all three present-state lines. If 111 happens to be an invalid state, then the only kind of don't-care information that will be relevant is 11− or −11. If 110 and 011 are valid states, then 11− and −11 are not invalid state cubes and therefore it is not necessary to use 111 as a don't-care. On the other hand, if 110 is also an invalid state, then 11– is an invalid combination for lines $U_P - P_3$, namely, P_1, P_2, and this should be used as a don't-care during synthesis.

As has been stated in the theorems, only internal faults are considered, *i.e.*, no faults on primary inputs, present-state lines, next-state lines, and primary outputs of the circuit are considered. These faults have dramatic effects and are easy to generate tests for. The conditions for the above theorem can be used to prune the set of don't-cares required to remove redundant internal faults. For some circuits, the set of don't-cares can be very large. The time required for synthesis often depends on the size of the don't-care set; the larger the set, the larger the time taken. Pruning the set of don't-cares is therefore an important step in improving the efficiency of the synthesis process.

4.4 The Synthesis and Test Strategy

The synthesis for testability approach to be outlined involves the use of a sequential test pattern generator to derive the essential set of don't cares and combinational logic optimization using this don't-care set. The test generators described in Chapters 2 and 3 can be used. The overall synthesis for testability procedure is as follows:

1. If the RTL description of the circuit is available, it is analyzed to see if it satisfies the conditions of Theorem 4.3.1. If so, the logic block is partitioned and each block is synthesized to be combinationally prime and irredundant. The resulting sequential machine is fully testable, non-scan.

2. Otherwise, the circuit is synthesized to be combinationally prime and irredundant using the procedure outlined in [13] or using any redundancy identification procedure. This guarantees that there are no CRFs in the circuit.

3. A fault list for all single stuck-at faults in the circuit is derived and tests are generated for faults in the list. If an RTL description of the circuit is available, ELEKTRA is used, otherwise STEED is used.

4. For every fault requiring invalid states for excitation, the invalid states are stored. Pruning techniques like those of Theorem 4.3.3 are used, if necessary, to decrease the

size of the don't-care set. Procedures for detection of invalid states are presented in Section 4.5.

5. For every fault that produces fault-free/faulty state pairs that are equivalent, the equivalent states are stored. Pruning techniques like those described in Theorem 4.3.2 can be used, if necessary. Procedures for the detection of equivalent states are presented in Section 4.6.

6. The combinational logic is synthesized to be prime and irredundant under the invalid and equivalent state don't-care set. Not all equivalent state pairs can be represented using traditional don't-cares, as was illustrated in Figure 4.7. In such cases, Boolean relations have to be minimized while synthesizing the logic.

7. After re-synthesis, some previously invalid states might become valid states and the set of invalid and equivalent state don't-cares might change, together with the network topology. Therefore, Steps 3 to 6 are repeated until the circuit becomes fully testable or the circuit structure and the set of don't-cares don't change from one iteration to the next. In [49] it is shown that this procedure must converge.

Faults that require only invalid states for justification are redundant, as are those that only produce equivalent fault-free/faulty state pairs. After all don't-care information has been derived, the circuit is optimized using MIS by specifying all possible don't-care information that MIS can make use of. Currently, MIS cannot use all the don't-care information, neither does it guarantee a prime and irredundant combinational circuit. Therefore, after logic optimization, a combinational test generator is used to make the circuit prime and irredundant under the don't-care set using techniques described in Chapter 7.

An alternative to this synthesis strategy would be to explicitly remove the redundancies in the circuit, as soon as they are detected, using a sequential test pattern generator like STEED or ELEKTRA. Explicit removal of redundancies would entail the replacement of the redundant line with a 0/1, propagating the value through the network, and eliminating all wires that get set to a 0/1 value. Therefore, for each redundant fault, the amount of logic that can be removed would vary with the fault. Since redundant faults have to be removed one at a time [38], after each removal all previous faults have to be checked to ensure that their status (redundant or testable) haven't changed. Removal of each redundant fault forces a new iteration where test generation has to begin all over again. Rather than removing the redundancies explicitly, don't-cares are used to *implicitly* eliminate them. Test generators like ELEKTRA are not able to establish equivalent-SRFs but can identify equivalent states. Therefore, equivalent-SRFs would not be removed using the explicit approach but would be removed using the implicit approach.

The procedure above guarantees that the synthesized logic-level circuit will not have any invalid-SRFs, neither will the kind of simple equivalent-SRFs shown in Figure 4.3 occur. Isomorph-SRFs and complicated equivalent-SRFs may be present since these redundancies can occur in theory even after using the don't-care sets above. However, even a single instance of such a redundancy has not been encountered to date in the experiments of Section 4.7 and in the experiments reported in [49, 50]. It is not hard to see why such faults are rare. For a complicated equivalent-SRF, a single stuck-at fault has to modify only two edges in the STG, one from a valid state and the other from an invalid state. It is difficult to find a fault that does this. For an isomorph-SRF, the original state assignment should be such that there is another state assignment whose cost is only one literal less than the original one. Since most state assignments tend to be locally optimal, such a situation is also difficult to come by. Therefore, such faults are not considered.

4.5 Detection of Invalid States

Invalid states are detected during state justification. In Sections 2.7 and 3.4, the procedures for the detection of invalid states using either the *covers* of the ON and OFF-sets of PO and NS lines or an *RTL* description of the circuit was briefly described. Details are provided in this section.

Consider a circuit described only at the logic level. For such a circuit, STEED would be used for the purpose of test generation. The first step in checking whether a state is invalid is to compute its fanin. If the fanin is empty, the state cannot be reached from the reset state and is therefore invalid. However, this is not the only criterion for detecting invalid states. Given any excitation state that has to be justified, it is marked temporarily invalid. Now all its fanin states are computed, and if the reset state is not covered by some state in the fanin, then all the states are stored in a *transitive fanin set.* For every state in the transitive fanin set that has not been considered before, the fanin of that state is generated and added to the transitive fanin set. In this manner, all states in the transitive fanin of the initial excitation state can be generated. Once all the states in the transitive fanin are obtained, they can be said to be invalid if the reset state is not in the transitive fanin set. For example, for the machine whose STG is shown in Figure 4.2, if IC is chosen as an excitation state, the transitive fanin of IC contains only IB, and thus states IC and IB can be declared invalid.

The same procedure is used when justification is performed using the RTL description of the circuit. If while justifying a state the decision tree becomes empty, then the state has no fanin states and is definitely invalid. Otherwise the transitive fanin of the excitation state is computed, as described in the previous paragraph, and is used in the detection of invalid states. The test generation procedure might end up detecting all invalid states, but in many cases only those that are required as excitation states and therefore required to ensure full testability are computed.

If the invalid states form small disjoint groups of states, it is easy to detect them. On the other hand, if the invalid portion of the STG is large and well connected, it might not be possible to identify invalid states as it would entail the enumeration of a large STG. It has been observed that there exist circuits where the valid portion of the STG is a small fraction of the total STG and the invalid portion of the STG is well connected. For such machines it is not possible to list all the invalid states and use them as don't-cares during synthesis. Such circuits are inherently hard to test and hard to synthesize to be fully testable. Fortunately, they are very rare and can be re-designed to be highly testable by starting from the behavioral description of the circuit.

Recently, State Transition Graph traversal techniques [30, 35] based on BDD representations have been used to obtain all the valid states of a finite-state machine [87]. The set of invalid states of the machine can therefore be easily obtained. However, not all of these states will be necessary to ensure full testability. The algorithms described in this section extract only the necessary set of invalid state don't-cares.

4.6 Detection of Equivalent States

Procedures for the identification of equivalent states were presented in Sections 2.6 and 3.7. As pointed out in Section 2.6, for every fault-free/faulty state pair, at first a single-vector differentiating sequence is sought using cube intersections. Instead of using cube intersections, ELEKTRA (*cf.* Section 3.7) uses a D-Algorithm-type fault effect propagation from a PS line to a PO. If a single-vector differentiating sequence is not found, the states that are in the fanout of each state are found. If the fanout states of the two states are identical, then the two states are equivalent. If not, differentiation is now attempted using the new state pair. In both cases a depth-first search technique is used, where in the worst case the entire STG of the machine has to be enumerated to find a differentiating sequence or to establish that two states are equivalent. For large machines, enumeration of the entire STG is not possible. However, it has been observed that the worst-case situation almost never arises. In most cases the equivalence of two states can be established in a few time frames. This is because the fanout states of the fault-free/faulty states become identical within a few time frames.

Equivalences are detected between states that are produced as a fault-free/faulty state pair. The synthesis procedure might have to use all equivalent state pairs to ensure full testability, but in the average case, not all of them are needed to ensure full testability. Therefore, as in the invalid state case, only those equivalent state pairs that are necessary to ensure full testability are computed.

4.7 Experimental Results

In this section, experimental results for the synthesis and test procedure described here are presented. In Table 4.1, for each example used, the number of inputs (#Inputs), outputs (#Outputs), and latches (#Latches) are given. Example **dsip** was obtained from [63]. The example **ex1** is a data path of a simple computer with no associated control.

CKT	#Inputs	#Outputs	#Latches
dsip	228	197	224
ex1	130	32	64
des	113	64	64
key	258	193	228
viterbi	94	43	640

Table 4.1: Statistics for example circuits

Examples **des** and **key** are large finite-state machines from a data-encryption chip [129]. Example **viterbi** is digital signal processing chip in a speech recognition system [125].

The examples described in Table 4.1 are synthesized without using the extended set of don't-cares, and the results of test generation are presented in Table 4.2. In this table the number of test sequences (# Test Seq.), the total number of test vectors (# Vect), the percentage of redundant faults (Red. fault), the total fault coverage including detected and provably redundant faults (TFC), and the TPG time on a VAX 11/8800 are indicated for each example. Also, the times required for logic optimization (LO time) and don't-care determination (both invalid state and equivalent state don't-cares), which is a part of TPG time, is indicated (DC time). Logic optimization is performed using MIS [15] and a combinational test generator. The circuits were synthesized to be combinationally prime and irredundant and the number of literals in factored form (as a measure of the area) for each of these circuits is shown in the last column of the table. For the example **dsip** the test generator STEED was used. For the rest of the examples ELEKTRA was used. Example **ex1** satisfied the conditions of Theorem 4.3.1 and was easily synthesized to be fully testable. In **key**, though certain states were detected to be invalid, they were not essential for test pattern generation. For all examples, a large fraction of the test pattern generation time was spent in combinational test generation and fault simulation. All redundant faults detected in the last four examples are invalid-SRFs.

In Table 4.3, the number of iterations required (#Iterations) during logic optimization to obtain a fully testable circuit, the additional synthesis time (LO time) using the extended don't-care set, the fault coverage (TFC), and the time required for test pattern generation (TPG time) is shown. Test pattern generation time is significantly smaller than

CKT	#Test Seq.	#Vect	Red. fault (%)	TFC (%)	TPG time	LO time	DC time	Area
dsip	8	212	0.01	100	25.8m	24m	2.0s	2766
ex1	72	208	0	100	226s	26m	0	4671
des	129	202	0.6	100	256m	2h	36m	2312
key	33	203	0	100	23m	1.5h	3m	2960
viterbi	329	2045	0.17	100	674m	4.5h	40m	2075

Table 4.2: Test generation results for circuits

CKT	#Iterations	LO time	TFC	TPG time	Reduction in Area
dsip	1	45s	100	2m	0.8%
des	3	100m	100	21m	10.2%
viterbi	4	136m	100	48m	6.5%

Table 4.3: Results of logic optimization

those in Table 4.2 because the test patterns generated previously gave a high initial fault coverage. Only the examples that were not fully testable are shown in Table 4.3. As can be seen, the examples have been synthesized to be fully testable within a reasonable amount of CPU time. The percentage reduction in the number of literals after the use of the extended don't-care set is shown in the last column of the table. In all cases the circuits were smaller. Since Scan-based approaches would have required the area indicated in Table 4.2 (and perhaps more, because of the Scan latches), this approach compares favorably with respect to Scan-based approaches in terms of area (as well as performance, since they are intimately related).

4.8 Conclusions

A synthesis procedure that produces an optimized and fully testable logic implementation of a sequential circuit starting from an RTL description or a logic-level description of the sequential circuit was presented. Unlike previous approaches, this is not based on the STG description of the system and is not restricted to small controller circuits. Datapath-controller circuits as well as digital signal processors, whose STGs are very large, can be synthesized to be fully testable. The essential invalid-state and equivalent-state don't-cares are derived and are used in synthesis. The procedure involves the use of combinational logic optimization and test generation techniques for the detection of invalid and equivalent states. Though a purely gate-level description of the circuit can be used as a starting point, using an RTL description often facilitates the detection of invalid and equivalent states in large circuits. The procedure does not introduce any area or performance overhead like scan design and can be used for circuits an order of magnitude larger than those handled by previous State Transition Graph based approaches. The memory and CPU time requirements for synthesis and test generation are also reasonable.

As has been pointed out, in some cases, detection of invalid states and equivalent states can take an exponential amount of time. Though such circuits have not been encountered, their existence cannot be ruled out. Similarly, there might exist examples for which faults give rise to isomorph-SRFs or complex equivalent-SRFs, which are not handled by the approach presented in this chapter. Despite these drawbacks, this approach works for a class of large circuits.

Chapter 5

VERIFICATION OF SEQUENTIAL CIRCUITS

The problem of verifying the equivalence of two implementations of a sequential circuit and the problem of test generation for such circuits are intimately related. The verification problem can be formulated as a decision problem where given two descriptions of a circuit, the question asked is whether the two descriptions have the same functionality. With the increasing use of sophisticated, automatic optimization tools in the synthesis of combinational and sequential logic circuits, it has become essential to be able to verify efficiently that the optimized and original descriptions represent the same machine, *i.e.*, the synthesis process has not introduced any errors in the circuit. As observed in Chapter 1, verification is required at all stages of the design process and different verification techniques are used at different stages for verification. One of the most important phases of the design process is sequential logic optimization. At this level, various techniques like retiming [86, 93], decomposition [10], optimization under don't cares [40, 87], and re-encoding [40] are used to transform the logic-level description of the circuit to a more optimal description. The focus of this chapter is the verification of circuits described at the logic level. The main application of the techniques described in this chapter is in verifying the correctness of the optimization and synthesis tools.

A significant amount of research has been performed in the area of verification of combinational circuits [27, 67, 91, 92, 94, 133]. To date, one of the most efficient approaches for verification of such circuits use Binary Decision Diagram-based canonical representation, whereby the verification problem can be converted into a graph isomorphism problem [27, 94]. However, there exist circuits for which the size of the BDDs can be proved to grow exponentially with the number of inputs to the circuit (*e.g.*, multipliers) [26]. For such circuits, enumeration-simulation approaches [91, 133] work better due to their smaller memory requirement. In such an approach, the truth table of one circuit is enumerated and simultaneously simulated on the other circuit. The time required for verification can grow exponentially with the number of inputs to the circuit. However, using implicit enumeration, the number of cubes to be simulated can be significantly reduced. In a different approach, the use of multi-level tautology for verification has also been proposed [67]. In this technique a product network is built from the individual networks and the product network is checked for tautology.

The sequential logic verification problem is more complicated than the combinational verification problem. One approach to sequential verification is exhaustive simulation, where each path in the State Transition Graphs (STGs) of the machines being verified is simulated. Since the number of input patterns to be simulated in an equivalence check *always* grows exponentially with the number of inputs to the logic circuit, this approach is only feasible for small circuits. To reduce the number of paths to be enumerated and simulated, implicit enumeration techniques are used [46, 62]. Formal verification methods also use some form of implicit search mechanism in order to prove or disprove logic equivalence [79]. In this chapter, an algorithm for the implicit enumeration of the STG of a circuit is described and its use in verification is presented. This algorithm is memory and CPU time efficient and requires only the explicit storage of the logic-level descriptions of the circuits and the valid states in the finite state machines. Unlike previous algorithms as in [46], the input as well as the state space is enumerated implicitly. The algorithms described are especially applicable to interconnected finite state machines.

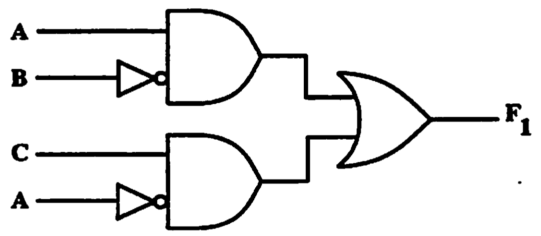
The rest of the chapter is organized as follows. Definitions are presented in Section 5.1. Previous work in this area is reviewed in Section 5.2. An implicit STG traversal algorithm is presented in Section 5.3 and its use in verification is described. This algorithm uses the covers of the ON and OFF-sets of the primary outputs and next-state lines for traversal. In Section 5.4, an implicit STG enumeration algorithm is presented and its use in verification is illustrated. Results using both approaches are presented in Section 5.5, followed by conclusions in Section 5.6.

5.1 Preliminaries

A general sequential circuit is shown in Figure 2.1. In order to show that two such circuits are not equivalent, it is necessary to find a primary input sequence which when applied to the two machines results in the machines asserting different output sequences. If no such sequence exists, the machines are said to be **equivalent**. Before trying to determine equivalence, correspondence between at least one state in each machine is required. Therefore each machine is assumed to have a **reset state**. The problem of verification of sequential machines is thereby reduced to the problem of determining the equivalence of the reset states of the two machines.

Consider a Boolean function of r input variables and n output variables. This function is a mapping from the r-dimensional Boolean input space to the n-dimensional Boolean output space. The process of finding the output value of the function (or range) for each point in the input space (or domain) is called an **enumeration** of the function [41]. Given a logic-level description of a circuit implementing a function, **minterm enumeration** refers to the use of minterms for the purpose of enumeration. Each input applied to the logic-level circuit is a minterm and enumeration involves the simulation of the circuit for each of the minterms. On the other hand, **implicit enumeration** refers to the use of cubes (with one or more don't-care entries) for enumeration. In minterm enumeration all 2^r input combinations have to be considered, but in implicit enumeration the number of cubes to be considered can be significantly less than 2^r. For example, for the 3-input single output function of Figure 5.1, minterm enumeration requires 8 input vectors but implicit enumeration requires only 4 vectors.

The combinational logic block of the sequential circuit takes both primary inputs as well as present-state variables as input. If there are p primary inputs, q primary outputs, and



IMPLICT ENUMERATION

ON	OFF
1 0 -	1 1 -
0 - 1	0 - 0

Figure 5.1: Circuit illustrating explicit and implicit enumeration

n present-state and next-state lines, the combinational logic block implements a mapping from the $(p+n)$-dimensional input space to the $(q+n)$-dimensional output space. The total input space for the combinational logic block can be partitioned into the p-dimensional primary input space or simply the **input space** and the n-dimensional **state space.**

Given finite state machines M_1 and M_2, the **product** of the two machines is defined to be the single machine obtained by connecting the machines in parallel, as shown in Figure 5.2. The inputs of both machines are connected to the primary inputs. The product machine has a single output which is obtained by connecting the corresponding outputs of the individual machines to an XNOR (or equivalence) gate and connecting the outputs of the XNOR gate to an AND gate. For machines M_1 and M_2 to be equivalent, the output of the product machine should always be true (*i.e.*, 1). Alternately, the product machine can have the same number of outputs as the original machines, and this is obtained by not performing the final AND in the previous step. The main use of the product machine is for the purposes of verification or to establish the equivalence of two states.

If there exists for a pair of states (S_i, S_j) a differentiating sequence of length k, the states S_i and S_j are said to be **k-differentiable.** States that are not i-differentiable for any $i \leq k$ are said to be **k-equivalent.** If a finite differentiating sequence does not exist

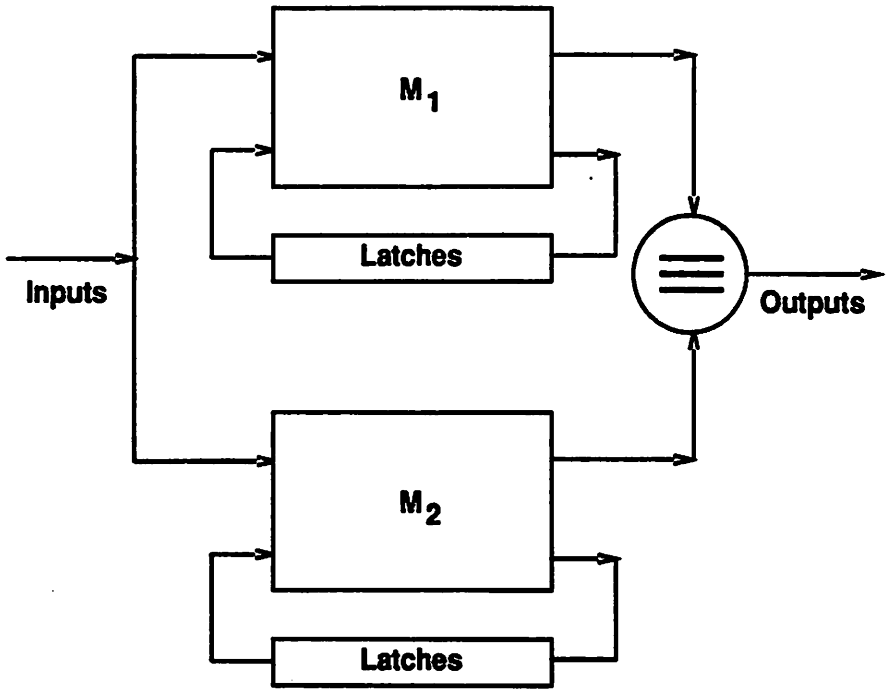


Figure 5.2: Product machine

for a state pair, the states in the pair are said to be **equivalent**. Two states are said to be **output-equivalent** or **single-cycle equivalent** if they are 1-equivalent.

Enumeration of the STG of a machine is the process of obtaining the STG description of the machine. In such a description, each state is represented with a unique symbolic code, and from any state, given any input vector, the next state and the output can be uniquely determined. On the other hand, in the **traversal** of the STG of a machine, each state might not have a unique symbolic code and the next state for every input combination might not be uniquely determined. However, all valid states are visited to check if a certain property is true for all such states. Examples in Sections 5.3 and 5.4 will illustrate the difference between the two.

5.2 Previous Work

The problem of verification of sequential circuits has been under investigation for a long time. There exist algorithms that require the explicit storage of the State Transition Graph of the sequential machine in order to check for equivalence [25, 79, 126]. These algorithms can only be used for circuits with less than 10 latches since the STG descriptions of large circuits typically require huge amounts of memory to store.

To avoid storing the STG of the machine, the enumeration-simulation approach was first presented in [46]. In this approach the STG of one machine is enumerated and simultaneously simulated on the other machine. During enumeration every path in the STG and all valid states have to be visited. One way of doing this would involve explicit or minterm enumeration of the input as well as the state space. This is similar to exhaustive simulation and therefore can be applied only to small circuits. However, the memory requirement of such a method is small and the entire STG does not have to be stored. In the algorithm described in the afore-mentioned paper, cubes are used to implicitly search the input space thereby coalescing multiple edges between states to one edge. A depth-first enumeration procedure is used whereby only one path in the STG has to be stored at any point of time, making the approach memory efficient. Implicit enumeration of the input space is performed using a PODEM-based [65] algorithm.

Recently, efficient *symbolic STG traversal* algorithms have been developed that can be used to traverse the STG of a machine and verify whether a certain property is true for all the valid states of the machine [30, 35]. In these approaches, a breadth-first technique is used and the input as well as the state space is implicitly enumerated. Such a traversal algorithm can be used for verification by constructing and traversing the STG of the product machine, ensuring that every reachable state in the product machine asserts the output 1. The algorithms are best implemented using Binary Decision Diagrams (BDDs) [27]. These methods work best for circuits that have a certain regularity in the structure of the STG. Such regularity is displayed in datapath-type circuits. For such circuits, the STG is very wide but not deep, *i.e.*, there are a large number of states but almost any state can be reached from any other. Therefore any state can be reached in a small number of clock cycles. Most large circuits contain datapath portions and can be handled efficiently by this approach. However, for certain circuits, the BDDs are so large that they cannot be built. Also, for certain machines, the transition relations become very complex causing each iteration to take a long time. Improvements on these techniques have been presented in [37, 66, 127]. Most of these improvements are in the form of better ordering strategies, recursive range computation methods, and the use of ATPG techniques to fine tune the traversal process.

5.3 Implicit State Transition Graph Traversal

In this section, an algorithm for the traversal of the STG of a sequential circuit is described. This algorithm enumerates the input as well as the state space implicitly, giving rise to significant savings in CPU time for verification. Before going into the details of the algorithm, the use of implicit state methods will be motivated with an example.

Consider the interconnection of machines shown in Figure 5.3. The STGs of machines M_1 and M_2 are shown underneath each machine. State A in M_1 and state X in M_2 are the reset states. Each of the machines is state minimized, and the output of M_1 is fed as the input of M_2 through a pipeline latch. The reset state of the pipeline latch is 1. Pipeline latches are introduced to preserve timing constraints and often do not add much complexity to the sequential behavior of the circuit. If this interconnection of machines is considered as a single machine, then the enumeration approach of [46] that implicitly enumerates only the input space yields an STG with 9 states and 13 edges as shown in Figure 5.4(a). Each state in the machine has a 5-bit code, where the first two state bits correspond to the state of M_1, the next two correspond to the state of M_2, and the last bit corresponds to the state of the pipeline latch. As can be easily seen from Figure 5.4(a), the states in the state pairs $\{01001, 01000\}$ and $\{01101, 01100\}$ that differ only in the state of the pipeline latch are equivalent, and can be combined into single cube states 0100– and 0110–. During the enumeration process, if the state bits are allowed to have the don't-care value, then a smaller STG with 7 states and 9 edges can be obtained, as shown in Figure 5.4(b). This STG is not only smaller but takes less time to enumerate.

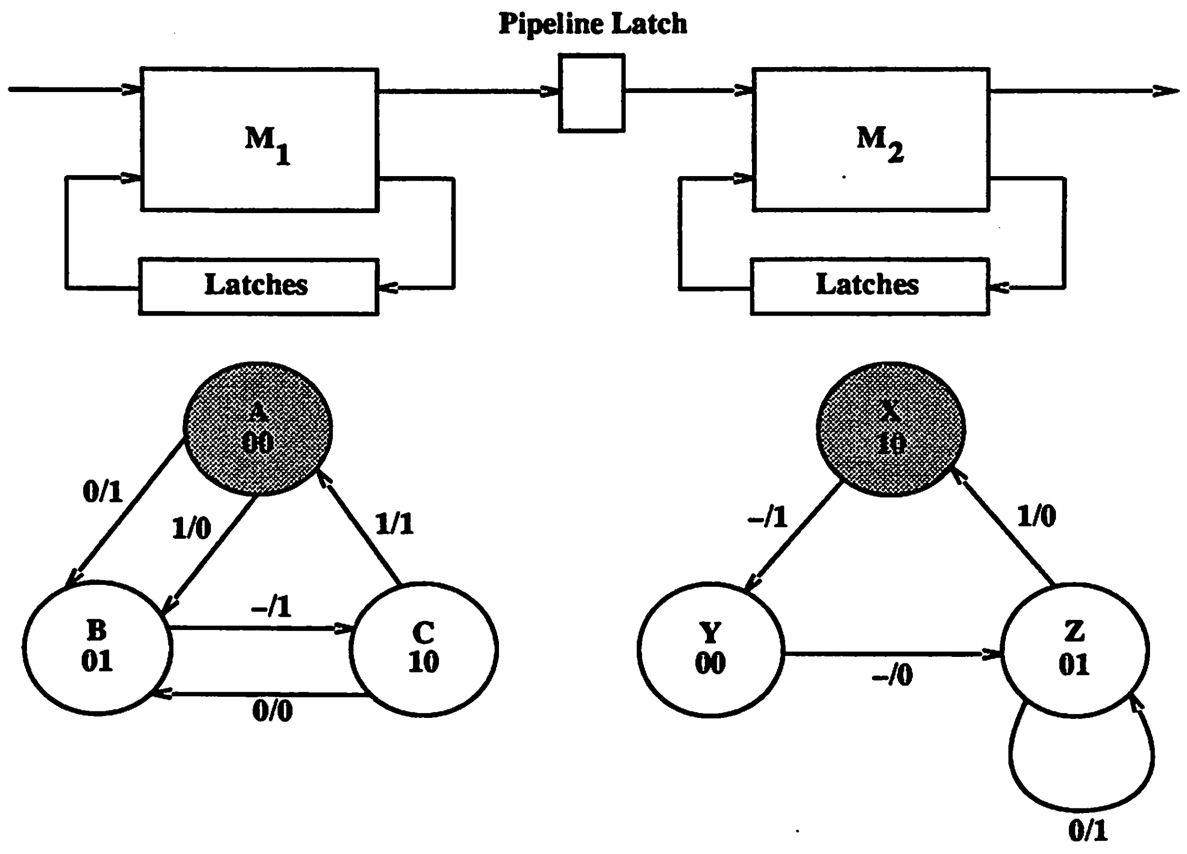


Figure 5.3: A cascade of two machines

As illustrated with this example, implicit enumeration of the state space together with the input space can give rise to savings in terms of CPU time to generate the STG and can produce more compact STGs. In the compact STG representation, a cube state implies that all minterm states covered by that cube state are equivalent. Note that only states that are equivalent and have codes that can be combined into a cube (without including other states that are not equivalent to it) can be represented as a cube state. Most complicated sequential circuits are an interconnection of simpler machines. As illustrated with this example, interacting machines connected through pipeline latches have the property that there are many states that are equivalent and have codes that can be combined into a cube.

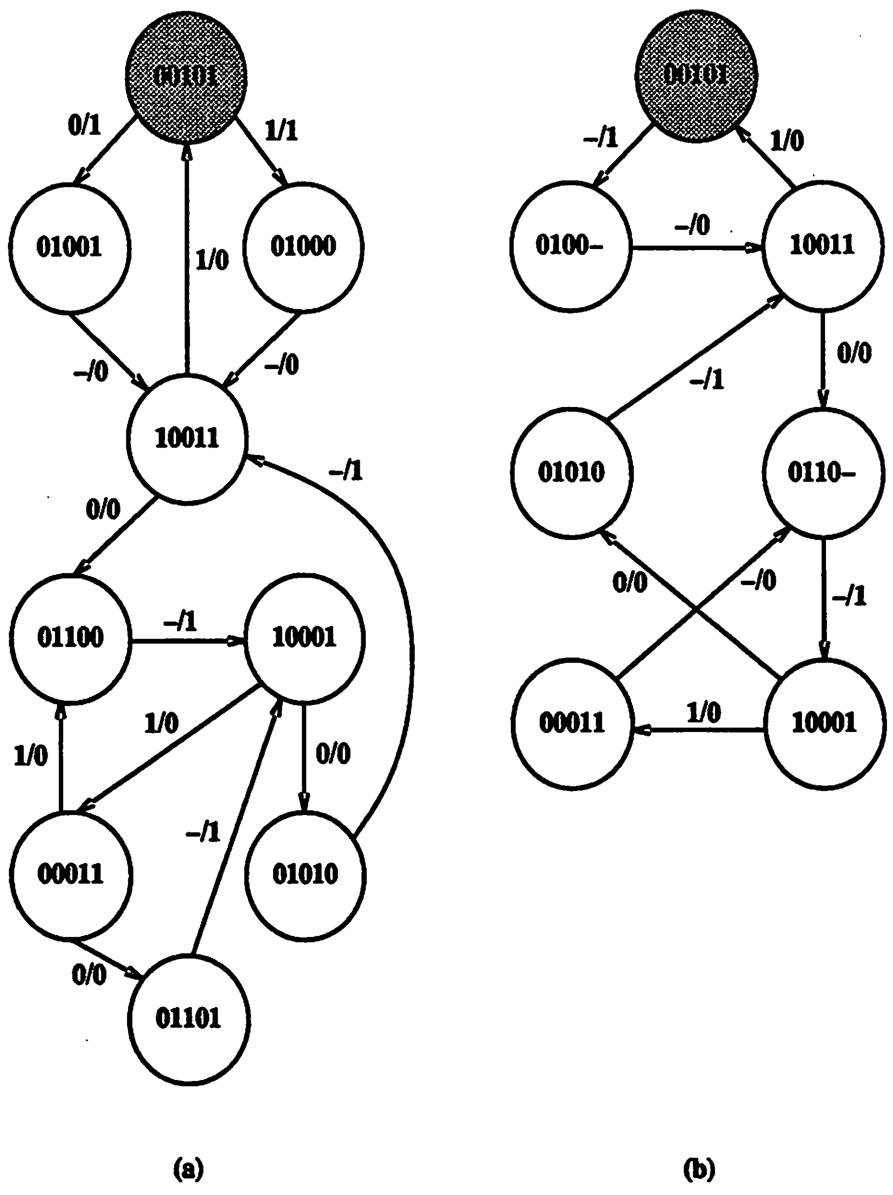


Figure 5.4: STGs using explicit and implicit state enumeration

Therefore the approach presented is especially suitable for complex interacting machines.

Before going into the details of the traversal algorithm, its use for the purpose of verification will be illustrated. The main verification procedure is shown in Figure 5.5.

```
verify_equivalence($M_1$, $M_2$)
{
  /* $M_1$ and $M_2$ are the machines to be verified */
  productMachine = form_product_machine ($M_1$, $M_2$);
  flag = stg_traverse (resetState);
  if (flag is TRUE)
    Machines are same;
  else
    Machines are different;
}
```

Figure 5.5: Main verification procedure using traversal

The first step in the procedure is to form the product machine using the two machines to be verified. This is a simple and standard procedure and will not be discussed. If the two machines are identical, then all valid states in the product machine should assert the output 1. To check this, the product machine is traversed using the routine **stg_traverse()**. The initial state of the product machine is the concatenation of the reset states of the two component machines. During traversal, all valid states are visited, and if any valid state asserts the output 0, the machines are different. Otherwise, the machines are equivalent.

The STG traversal algorithm is geared towards the efficient traversal of the product machine. Before traversal begins, the covers of the ON and OFF-sets of each PO and NS line in the product machine are extracted using the method used in Chapter 2. The traversal algorithm shown in Figure 5.6 makes use of these covers.

The procedure is best described with an example. Since the procedure uses covers, a real example should use covers. However, the fundamental ideas are better illustrated with an STG. Therefore, the STG of an example product machine, as shown in Figure 5.7, will be used. In the following section, the use of covers for all the procedures will be illustrated with another example.

The reset state of the product machine is derived as the concatenation of the reset states of the individual machines. Traversal starts with the initial state S_1 = 010000. The

```
stg_traverse($S_1$)
{
  /* $S_1$ is the current state */
  if ($S_1$ is in the current path)
    return (TRUE);
  if ($S_1$ has all its fanout edges enumerated)
    return (TRUE);
  Add $S_1$ to current path;
  if ($S_1$ is covered by some state in List)
    restore saved decisionTree;
  else
    decisionTree = NULL;
  do{
    ($S_1'$, flag) = get_fanout_edge ($S_1$, decisionTree);
    if (flag is TRUE){ /* Some state variables were set */
      if saveFlag is set, then save decisionTree and $S_1$ in List;
      return (BACKTRACK);
    }
    flag = stg_traverse ($S_1'$);
    if (flag is BACKTRACK){
      LABEL :
      ($S_1'$, flag) = set_state_variables ($S_1$);
      if (flag is TRUE){
        /* State variables cannot be set without setting other state variables */
        if saveFlag is set, then save decisionTree and $S_1$ in List;
        return (BACKTRACK);
      }
      else{
        flag = stg_traverse ($S_1'$);
        if (flag is BACKTRACK){
          /* Cannot enumerate further without setting some more state variables */
          goto LABEL;
        }
      }
    }
    Assign last variable in decisionTree a different value and set saveFlag;
  } while (decisionTree is not empty);
  Store $S_1$ as a state whose fanout edges have been enumerated;
}
```

Figure 5.6: Procedure for traversing the STG of a machine

path variable is initialized to zero. The first step is to check if the current state S_1 is already in the current path. It is also necessary to check whether all fanout edges from S_1 have been enumerated or not. If so, there is no need to enumerate further from S_1. If not, the next step is to enumerate an edge in the STG of the product machine from state S_1. At this point it is also checked to see if S_1 is covered by any state in the set of states stored in the variable *List*. The use of this check will be illustrated later.

The enumeration of a fanout edge is performed in the routine **get_fanout_edge()**. The objective of this procedure is to set a minimal number of inputs so that all the outputs of the product machine are set to a value. This is performed using cube intersections and a minimal covering procedure that will be illustrated in Section 5.4. The inputs that are set determine the part of the input space that has been enumerated for the current state. These inputs are stored in a decision tree. The fanout state corresponding to the input is also determined through a series of cube intersections or simulation. In this case the outputs can be set without setting any primary inputs. The next state obtained is $S_1' = 00-01-$.

Having obtained the fanout state of the current state, further enumeration from the fanout state is carried out by calling the routine **stg_traverse()** recursively with the fanout state S_1' as the input variable. In this case $S_1' = 00-01-$ becomes the current state. The operations outlined in the previous paragraph are now repeated. For $00-01-$, the routine **get_fanout_edge()** sets the primary input to zero thereby setting the output to 11. The next state obtained is 100001. This new edge shows that for the input condition 0, the states covered by the cube $00-01-$ all go to the same next state and assert the same output. Continuing further from the state 100001, another edge in the STG going from state 100001 to 010000 and asserting the output 11 is enumerated. The part of the STG of the product machine traversed so far is shown in Figure 5.8(a).

Continuing further, it is noticed that the current state 010000 is already in the current path. The procedure goes back to state 100001 and tries to explore other fanout edges from that state. Since the input condition corresponding to the edge from 100001 to 010000 is a don't-care, all input conditions from state 100001 have been explored. The procedure therefore goes back one more state to the state $00-01-$. From this state a part of the input space has already been explored and the input conditions explored are stored

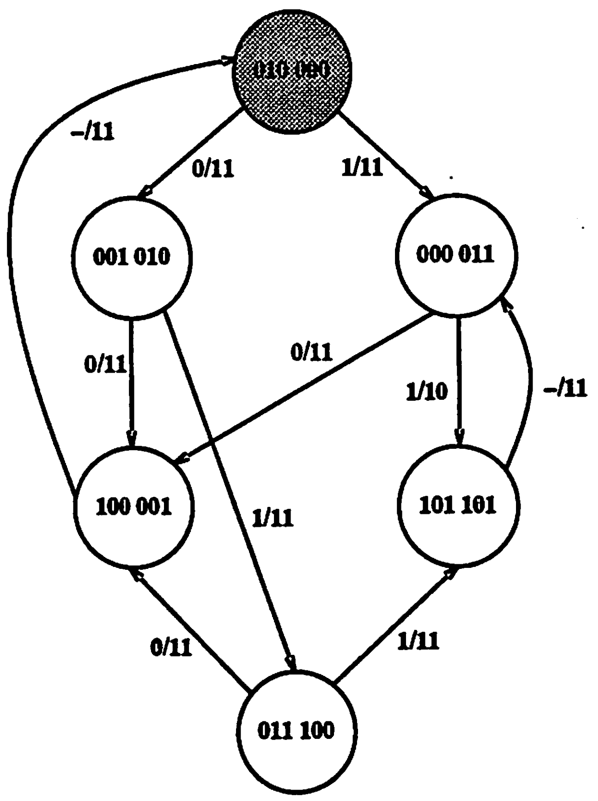


Figure 5.7: The STG of a product machine

in the decision tree. Now the last variable in the decision tree is assigned a different value, *i.e.*, the primary input variable is set to the value 1. Also, the *saveFlag* variable is set to indicate that part of the input space for that state has been enumerated. With the input set to 1, in order to set the outputs to a 0 or 1 value, the routine **get_fanout_edge()** has to set some state variables. With the current state being $00-01-$, the state variables that can be set are the third and the sixth variables. If the third and the sixth state variables are set to 0, the current state corresponds to an invalid state in the STG. Invalid states might assert the output 0, forcing the procedure to conclude wrongly that the machines are different. At this point two things should be done. At first, the routine **get_fanout_edge()** tries to see if the current state (obtained after setting the state variables) is justifiable. This is performed

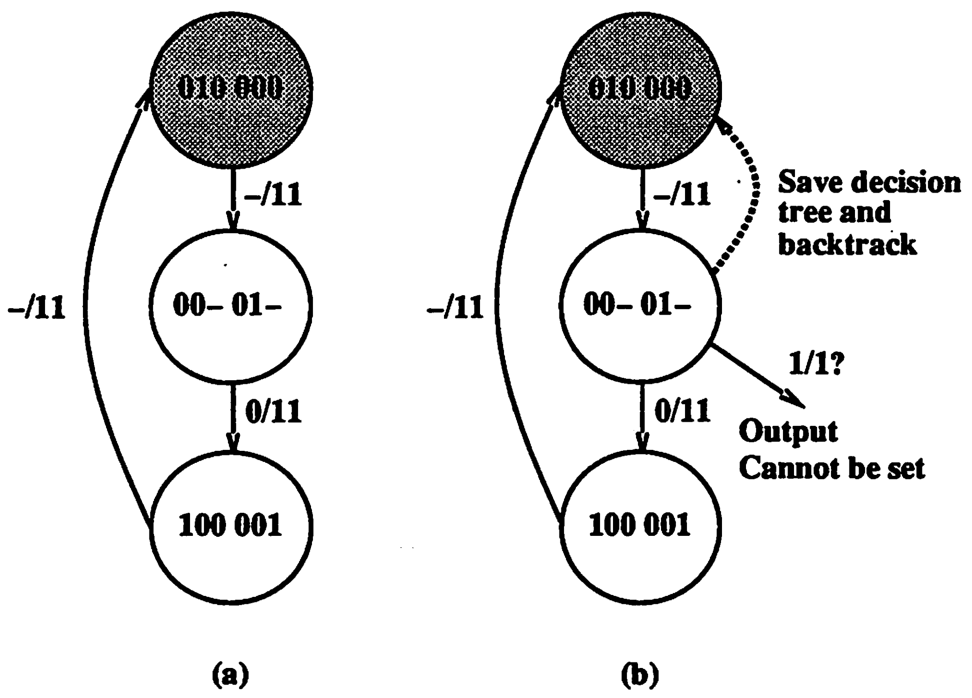


Figure 5.8: Parts of the STG enumerated during traversal

by obtaining the fanin states of the current state, using the cube intersection technique of the justification algorithm of Chapter 2. If there exists a state in the fanin of the current state that covers the fanin state of the current state, then the current state can be reached from any state in the fanin of the current state. Therefore, the current state must be a valid state. If justification succeeds, the routine **get_fanout_edge()** does not set a flag. In this example, the current state obtained after setting the third and sixth state variables to 0 is an invalid state and therefore justification fails. Note that justification might fail even if the current state is valid, as the justification requirement is very stringent. When justification fails (as in this example), the routine **get_fanout_edge()** sets a flag that forces *backtracking*. The procedure backtracks to the fanin state of 00 – 01–, *i.e.*, to state 010000. Since part of the input space from state 00 – 01– has been enumerated (as indicated by *saveFlag*, which is set), the decision tree and the state associated with it is saved in a list.

This is illustrated in Figure 5.8(b).

Having backtracked to the state 010000, the procedure **set_state_variables()** is used to set the required state bits by setting more inputs. For this purpose, once again, the covers are used. (Note that justification is performed in the routine **set_state_variables()** also). The details of this routine will be given in the next section. From state 010000, by setting the input to 0, the state 001010 is reached. In this state, both the third and the sixth state variables are set to a value. Further enumeration can proceed from this new state. However, this new state is covered by the state $00-01-$, for which half the input space has already been enumerated. This is discovered by checking to see if 001010 is covered by any state in *List*. Since it is covered by $00-01-$, the decision tree that was saved for state $00-01-$ is restored for the state 001010 and further traversal from state 001010 proceeds only with the edge corresponding to the input condition 1. In this manner, enumeration proceeds until state 000011 is reached. For this state the output corresponding to the input condition 1 is 10, indicating that the two machines are not equivalent. The traversal process stops and reports the non-equivalence and the path to the current state. The final STG obtained after traversal is shown in Figure 5.9. In this STG states do not have disjoint state codes. Moreover, some states only have fanout edges corresponding to the input condition 0 and some states only have fanout edges corresponding to the input condition 1.

Further discussion of this algorithm will be postponed until the enumeration algorithm is presented in the next section. It is necessary to point out that for large data-path like circuits, instead of considering all the outputs together, one output or a subset of the outputs can be considered at a time thereby exploiting some of the regularity of the topology and the natural partitions in the logic.

5.3.1 Incompletely-specified machines

For incompletely-specified machines it is expected that the user provides a set of don't-care sequences. Don't-care sequences could be of various kinds:

1. It could be a sequence of input vectors that will never be applied to the machine. The behavior of the machine when that sequence is applied is therefore not important.

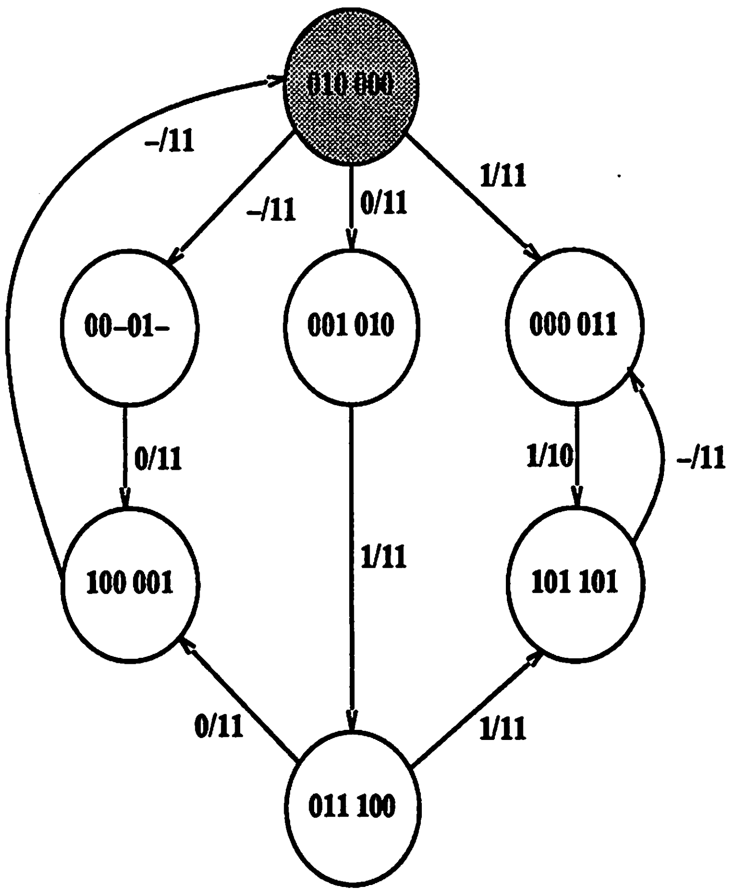


Figure 5.9: Final STG after traversal

2. Given a state of the machine, it could be input vectors that will never be applied when the machine is in that particular state. For such vectors, the fanout edges from the state corresponding to the don't-care input conditions are not important.

3. It could also be a set of vectors for which some or all of the outputs of the machine do not matter when the machine is in a particular state.

The procedure outlined in Figure 5.5 can handle incompletely-specified machines. It can be easily checked during traversal whether an input vector is a don't-care for a particular state. If so, the input vector is not considered. If a differentiating sequence is found and a don't-care sequence is a subsequence of the differentiating sequence, the differentiating sequence is rejected and the search for a new differentiating sequence is continued.

5.4 Implicit State Transition Graph Enumeration

Verification using the product machine approach works well for circuits that are not very large. Otherwise, the cover sizes become so large that storing the entire covers takes a significant amount of memory and cube intersections take a long time. For such large circuits a better approach is the enumeration-simulation approach, where the STG of one machine is enumerated and simultaneously simulated on the other machine. This approach is efficient for more than one reason. Firstly, only the covers of the single machine have to be stored thereby requiring smaller amount of memory and less time in cube intersections. Secondly, a depth-first search procedure is used whereby only one path in the STG of the machines has to be stored at any point of time.

The algorithm for STG enumeration, given a logic-level implementation of a circuit, is presented first. The main enumeration algorithm is shown in Figure 5.11 and will be illustrated with an example. Consider the machine shown in Figure 5.3. The covers of the ON and OFF-sets of each primary output and next-state line is shown in Figure 5.10. N_1 and N_2 correspond to the next-state lines for the latches in M_1, N_3 and N_4 correspond to the next-state lines for the latches in M_2, and N_5 corresponds to the pipeline latch.

At the beginning, the current state S_1 is the reset state of the machine. The path variable is initialized to zero. The procedure first checks to determine if the current state is covered by some state already in the path. It also checks to determine if all fanout edges from the current state has been enumerated. If so, no further enumeration is necessary. Otherwise, the current state is added to the current path and the next step is to enumerate a fanout edge from the current state. An input vector to the combinational logic block of the circuit is formed. This input vector has all its primary input variables set to don't-cares and the present-state variables set to the current state. The vector is now intersected with the ON and OFF-set of each primary output line. If the vector intersects only the ON-set

O (ON)	O (OFF)	N_1 (ON)	N_1 (OFF)
- - - 1 0 - - - - 0 1 0	- - - - 1 1 - - - 1 1 - - - - 0 0 -	- 0 1 - - -	- - 0 - - - - 1 - - - -
N_2 (ON)	N_2 (OFF)	N_3 (OFF)	N_3 (ON)
- 0 0 - - - 0 - 0 - - -	1 1 - - - - - - 1 - - -	- - - 0 1 1	- - - - 0 - - - - 1 - - - - - - - 0
N_4 (ON)	N_4 (OFF)	N_5 (ON)	N_5 (OFF)
- - - 0 - 0 - - - 0 0 -	- - - 1 - - - - - - 1 1	0 0 - - - - 1 1 0 - - - - 0 1 - - -	1 0 0 - - - 0 1 - - - - - 1 1 - - -

Figure 5.10: ON and OFF-sets of the PO and NS lines of a machine

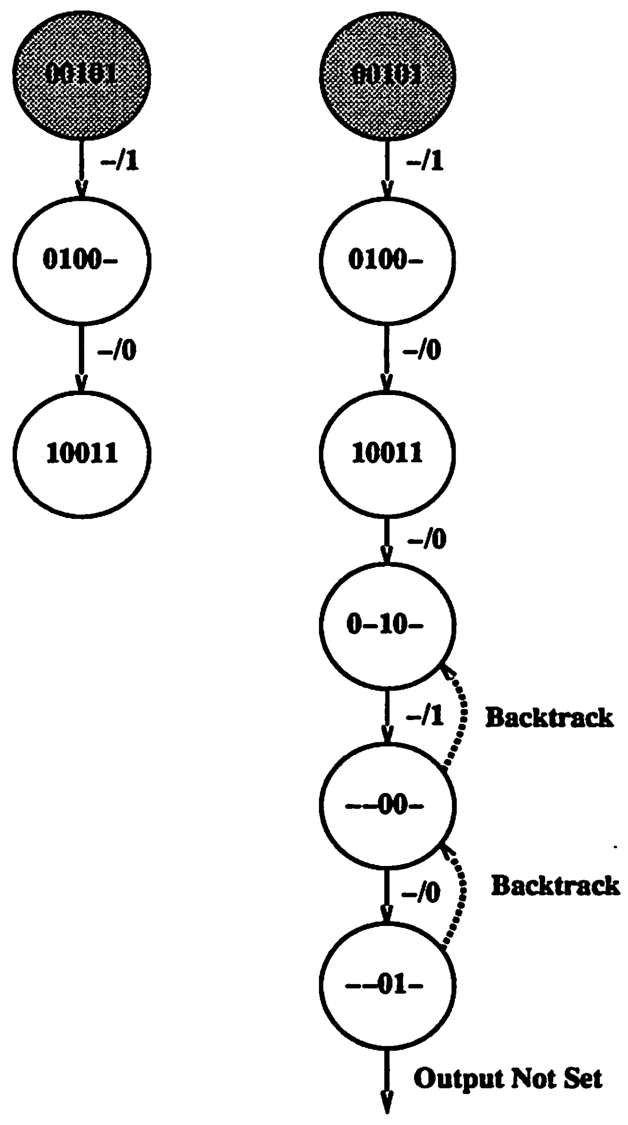
(OFF-set) of the PO line, the value of the line is 1 (0). In this example, the input vector is −00101 and it intersects only the ON-set of the PO line. Therefore the value of the output is 1. The values of the next-state lines are obtained by intersecting the input vector with the ON and OFF-sets of each next-state line. If the input vector intersects only the ON-set (OFF-set) the value of the corresponding next-state line is 1 (0). If it intersects both, then the value of the next-state line is a don't-care. Since a logic-level description of the machine is available, the next state can also be obtained by simulating the input vector. In this case the next state obtained is 0100−. These operations are performed in the routine **get_fanout_edge()**. However, this is only a part of what this routine does. Further operations will be illustrated later.

Having obtained the fanout state S_1' of the current state, further enumeration from the fanout state is performed by calling the routine **stg_enumerate()** recursively with S_1'

```
stg_enumerate($S_1$)
{
  /* $S_1$ is the current state */
  if ($S_1$ is in the current path)
    return (TRUE);
  if ($S_1$ has all its fanout edges enumerated)
    return (TRUE);
  Add $S_1$ to current path;
  decisionTree = NULL;
  do{
    ($S_1'$, flag) = get_fanout_edge ($S_1$, decisionTree);
    if (flag is TRUE) /* Some state variables were set */
      return (BACKTRACK);
    flag = stg_enumerate ($S_1'$);
    if (flag is BACKTRACK){
      LABEL :
      ($S_1'$, flag) = set_state_variables ($S_1$);
      if (flag is TRUE){
        /* State variables cannot be set without setting other state variables */
        return (BACKTRACK);
      }
      else{
        flag = stg_enumerate ($S_1'$);
        if (flag is BACKTRACK){
          /* Cannot enumerate further without setting some more state variables */
          goto LABEL;
        }
      }
    }
    Assign last variable in decisionTree a different value;
  } while (decisionTree is not empty);
  Store $S_1$ as a state whose fanout edges have been enumerated;
}
```

Figure 5.11: State Transition Graph enumeration algorithm

as the input variable. Therefore, in the next step of the enumeration process 0100– is the current state. The operations outlined above are repeated and another edge in the STG of the machine is enumerated. This edge goes from state 0100– to the state 10011 asserting



(a) (b) (c)

Figure 5.12: Example to illustrate STG enumeration

the output 0. The portion of the STG enumerated so far is shown in Figure 5.12(a). Note that 0100– covers two states, and both states under all possible input conditions go to the same next state and assert the same output. Therefore the states covered by 0100–

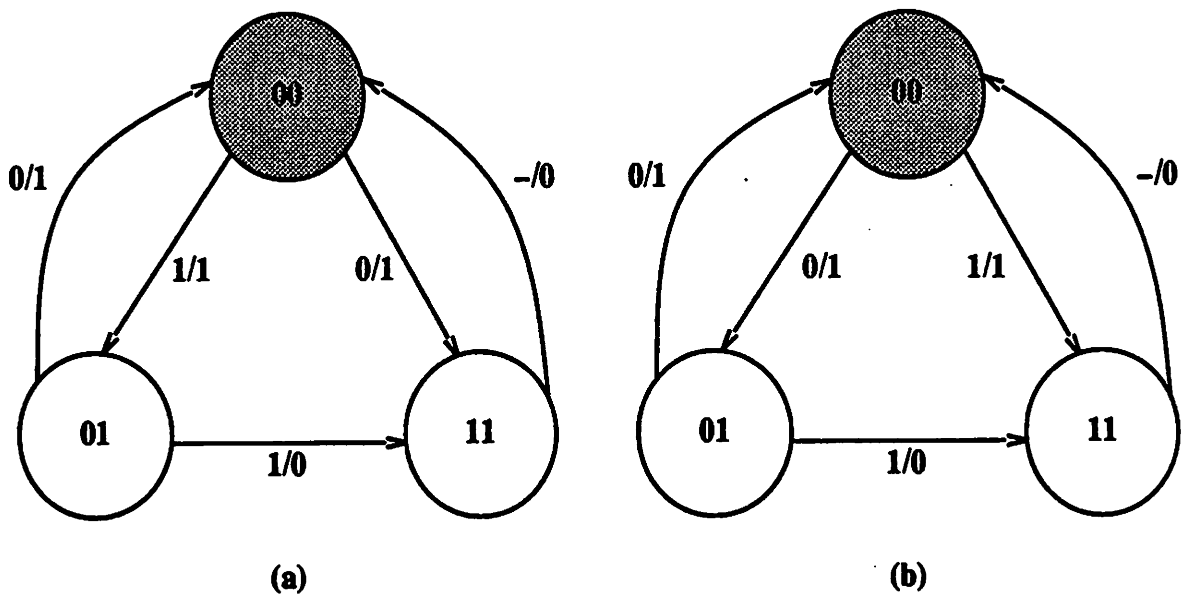


Figure 5.13: Machines illustrating difference between traversal and enumeration

are equivalent (in this case single-cycle equivalent), and the enumeration process implicitly detects this and combines them into one cube state. In order to detect such equivalent states it is necessary to reach, somehow, the cube state that covers the equivalent states. Otherwise the equivalent states will not be detected.

Continuing enumeration further, the routine **get_fanout_edge()** tries to find a fanout edge for the current state 10011. In this case, with the input vector is −10011, the output is set to 0 and the next state obtained after intersections is 0 − 10−. In the next step of enumeration 0 − 10− becomes the current state and the initial input vector is −0 − 10−. This vector intersects only the ON-set of the PO and therefore the output is set to 1. The fanout state derived for this state is − − 00−. This indicates that the states covered by 0 − 10− are output-equivalent. Enumeration continues in this fashion until the state − −01− is reached. The part of the STG enumerated so far is shown in Figure 5.12(b).

With the current state − − 01− the input vector initially is − − −01−. This input vector intersects both the ON and the OFF set of the PO line. Under such circumstances, the routine **get_fanout_edge()** tries to set some PI or PS variables that were don't-cares to a 0 or 1 value so that the input vector intersects only the ON or the OFF-set, *i.e.*, the output is set to either a 0 or a 1 value. To do this, the procedure examines how many variables have to be set to make the input vector orthogonal to first the ON and then the OFF-set. If the number of variables required to be set to make the input vector orthogonal to the ON-set is smaller than the number of variables required to be set to make the input vector orthogonal to the OFF-set, then the variables in the former group are set to a value and vice versa. To determine the variables that have to be set to a value to make the input vector orthogonal to the ON-set, the cubes in the ON-set that intersect the input vector are considered. All columns for which the input vector already has a variable set to a value are removed from the cubes. With the remaining set of columns, a minimal covering procedure is used to determine the set of variables which if set to a value will make the input vector orthogonal to the ON-set. This procedure is identical to the procedure for obtaining the *lowering set* in the EXPAND step of the logic minimizer ESPRESSO [16]. The only difference is that the covering procedure is biased to select primary input variables in the lowering set before selecting state variables. The rationale behind this is that if the output can be set by setting only primary input variables, then that should be done first. As will be seen shortly, setting state variables forces backtracking which splits up states that have been combined into cubes. This is undesirable and should be avoided whenever possible.

In this example, the ON-set cube that intersects the input vector is $---010$ and the OFF-set cube that intersects the input vector is $----11$. As can be seen, to keep the cube orthogonal to the ON-set or the OFF-set the fifth state variable has to be set to a value. Let this be called the required state variable. Since a state variable has to be set, the routine **get_fanout_edge()** sets a flag. The enumeration procedure now backtracks to the previous level where the current state is $--00-$. The procedure **set_state_variables()** tries to set the required state variable in the fanout state by setting only some more primary inputs. In this case, the procedure tries to obtain as the fanout state of $--00-$, some state like $--010$ or $--011$. This is similar to setting the output to a value as in **get_fanout_edge()**. However, instead of considering a primary output line, a next-state line is considered. If **set_state_variables()** succeeds in setting the required state variables without setting some other state variables, then enumeration continues with the new fanout state.

In this example, the procedure fails because in order to set the fifth state variable it is necessary to set the first or the second state variable. The enumeration procedure is forced to backtrack another level where the current state is $0-10-$. At this level, the routine **set_state_variables()** can set the fifth state variable to 1 by setting only the PI to 0. The new fanout state is $--001$ and enumeration proceeds further from this state as shown in Figure 5.12(c). In this fashion the entire STG of the machine can be enumerated and the final STG is shown in Figure 5.4(b).

For each state, the input variables set by the procedures **get_fanout_edge()** and **set_state_variables()** are stored in a decision tree. Whenever one path from a current state has been enumerated, the last variable in the decision tree is assigned a different value and other fanout edges from the current state are enumerated. Therefore, all possible input combinations for a particular state are implicitly but exhaustively enumerated. Once all fanout edges from a state has been enumerated, the state is stored in a special list of states all whose fanout edges have been enumerated.

This enumeration algorithm can be used in conjunction with a simulator for the purpose of verification. One of two methods can be followed. If the machines are small, the entire STG of one machine can be enumerated before simulating it on the other machine. If the machines are large, as soon as an edge is enumerated using one machine it is simulated on the other. If upon simulation, the output of the machine being simulated is not set, then the enumeration procedure starts to backtrack. If at any stage of the enumeration-simulation process the outputs from the enumerated and simulated machines don't match, the machines are different. Otherwise the machines are identical. In the enumeration-simulation approach, search along a particular path is terminated when the total state, consisting of the state of the enumerated machine and the simulated machine, already exists in the path.

Incompletely-specified machines can be handled in very much the same manner as in the product machine traversal case. It is assumed that the user provides the necessary don't-care sequences. As soon as a differentiating sequence is found it is checked to see that no don't-care sequence is a subsequence of the differentiating sequence. If so, the differentiating sequence is rejected and enumeration-simulation is continued.

The equivalent states that were detected in this example were single-cycle equivalent. Multiple-cycle equivalent states can also be detected. During the enumeration process output-equivalent states are grouped into cubes. Setting state variables corresponds to splitting of states combined into cubes. If during enumeration a cube state is never split, then all the states covered by the cube state are equivalent.

At this point it is necessary to distinguish between the enumeration and the traversal approach. A cursory perusal of the algorithms of Figure 5.6 and 5.11 shows that the difference between the two approaches is in the storing and retrieving of decision trees and in performing justification at certain times to verify that a state is valid. In the traversal algorithm it is necessary to visit only the valid states, and it is not important how the valid states are reached as long as all of them were reached. Therefore justification is used in certain cases to verify whether a state is valid or not. However, this cannot be done with enumeration, as illustrated with the example in Figure 5.13. In this case, for both machines, traversal would yield the same STG. The reason for this is the following. From state 00 the next state for all possible input conditions is -1 and the output is 1. When the state variable has to be set in the state -1, the don't-care state variable can be set either to 0 or 1, and both resulting states are valid states and their validity can be easily determined by justification. Backtracking would not be necessary under such circumstances. In both machines therefore, the edges going from state 00 to states 11 and 01 will never be revisited, after they have been visited once. However, one can easily determine that the machines are different and a differentiating input sequence is $\{1, 0\}$. Note that in trying to verify the equivalence of the two machines using the traversal method, the product machine will be traversed and in such a machine, the difference in behavior can be easily detected. In enumeration, however, the exact STG has to be enumerated and therefore justification cannot be used. The reason for storing a decision tree (or trees) during traversal is to avoid retracing portions of the STG that have already been traversed. But the use of stored decision trees in traversal produces an STG with state codes that are not disjoint and with many states having only parts of the input space enumerated. In enumeration it is required that

CKT	#I	#O	#L	Explicit State [46]			Implicit State		
				States	Edges	Time	States	Edges	Time
sse	7	7	6	13	60	0.3s	13	66	0.2s
sand	11	9	6	32	206	3.7s	32	163	1.01s
planet	7	19	6	48	135	3.0s	48	130	0.6s
scf	27	54	8	115	279	10.9s	115	280	3.35s
tlc	3	5	10	400	2026	5.3s	201	712	2.74s
mclc	11	6	11	35	1404	3.2s	35	1264	18.44s
viterbi	11	34	12	1863	214962	52m	1863	139087	49m
counter12	0	1	12	4096	4096	18.4s	4096	4096	10s
ac12	2	3	14	16384	65536	380s	16384	65536	290s
sbc11-12	35	51	33	–	–	–	6	70	180s

Table 5.1: Comparison of implicit and explicit state techniques

each state have a code disjoint from the rest and that fanout edges for all input conditions for each state be specified. Therefore, decision trees are not stored during enumeration.

The traversal as well as the enumeration-simulation approach uses a depth-first technique. However, since some states are combined into a single state during traversal, this technique can be thought of as a mixed depth-first/breadth-first technique. As will be shown in the results sections, circuits that have STGs that are not too wide but are deep, *i.e.*, there are not too many states but certain states require a large number of clock cycles to reach, are handled efficiently by the algorithms presented here.

5.5 Experimental Results

In this section, results obtained using the algorithms described in this chapter are presented. All examples have been run on a VAX 11/8800 computer. CPU times are quoted in seconds (symbol *s*), in minutes (symbol *m*), or in hours (symbol *h*).

In Table 5.1, the implicit state enumeration technique is compared with the explicit state technique of [46]. The number of states and edges enumerated using the approach of [46] are compared with the number of states and edges obtained using the enumeration algorithm of Section 5.4. The times required for enumeration of the STGs are also compared. For all the examples, the implicit state algorithm performed better or at least as good as the algorithm of [46]. For the example tlc, the advantage of the implicit state technique

CKT	#I	#O	#Gates	#L	#Valid States	#Edges in STG	CPU Time	Memory (kB)
sse	7	7	130	6	13	73	0.32s	214
sand	11	9	555	6	32	166	1.60s	288
planet	7	19	606	6	48	130	1.35s	283
scf	27	54	959	8	115	280	5.34s	485
tlc	3	5	76	10	201	712	4.07s	483
mclc	11	6	148	11	35	1264	29.5s	483
viterbi	11	34	232	12	1863	139087	1.23h	240
counter12	0	1	61	12	4096	4096	20s	311
ac12	2	3	152	14	16384	65536	640s	2074
adder6	6	6	54	6	64	4096	62.3s	272

Table 5.2: Verification of machines using traversal

over the explicit state technique is clearly demonstrated. For examples where the number of equivalent states were small or non-existent, the approach presented here is seen to do no worse than the approach of [46]. For the example **sbc11-12**, the STG could not be extracted using the approach of [46]. However, the implicit state approach was able to easily derive the STG.

Most of the examples presented here have been presented in the previous chapters. The examples **tlc** and **mclc** are traffic light controller chips. In the previous chapter the entire *Viterbi* speech processing chip [125] was used as an example. Here only the controller of the chip is used. It is an interconnection of smaller finite state machines. The example **sbc11-12** is the circuit corresponding to the 11th and 12th output of the snooping-bus controller discussed in Chapter 2. Examples **counter12**, **adder6**, and **ac12** are a 12-bit counter, a 6-bit adder, and a 2-bit ALU driven by a 12-bit counter respectively.

In Table 5.2, the results of verification of single machines using the product machine approach is presented. Both the statistics and the CPU times required for verification for ten finite state machine examples are given in Table 5.2. For each example, the number of inputs (#I), the number of outputs (#O), the number of gates (#Gates), and the number of latches in the initial implementation (#L) are indicated together with the time required for verification. For each example, verification was performed between different implementations of the same circuit and/or with different encodings, implying no correspondence

CKT	States	Edges	CPU Time	Memory (kB)
sse	13	66	0.4s	152
sand	32	163	2.0s	218
planet	48	130	1.2s	200
scf	115	280	7.7s	350
tlc	201	712	5.5s	346
mclc	35	1264	37.0	352
viterbi	1863	139087	1.6h	180
counter12	4096	4096	20s	220
ac12	16384	65536	610s	1600
adder6	64	4096	52s	256

Table 5.3: Verification of machines using enumeration-simulation

between the latches of the two circuits. Memory usage was restricted to a few megabytes for all these examples.

In Table 5.3, the results of verification of the same finite state machines using the enumeration-simulation approach is presented. For all examples, only the number of states and edges enumerated and the time required to verify the circuits using enumeration-simulation method is presented. As can be seen, the times required for verification using product machine traversal and enumeration-simulation is comparable for the examples presented here. The example **counter12** is a 12-bit counter which has a long and narrow STG. Example **adder6**, a 6-bit adder is another extreme, having an STG with a regular structure which is wide but not deep. Though traversal is slightly more efficient than enumeration in terms of visiting all the valid states, the advantage is offset by the fact that in the product machine the size of the covers are larger and therefore intersections take more time. For the product machine, about twice the number of intersections have to be done in the average case than in the enumeration of a single machine. Therefore traversal of the product machine is roughly twice as much work as enumeration of a single machine. On the other hand, simulation of a machine roughly requires the same amount of work as enumeration. Therefore the times required for verification using the two methods are comparable.

Finally, the enumeration-simulation approach presented here is compared with a recent approach in the verification of sequential machines [30, 35]. The results for the

CKT	Time taken by approach of [127]	Time taken using enumeration approach
sse	1.0s	0.4s
sand	7.7s	2.0s
planet	6.9s	1.2s
scf	42s	7.7s
tlc	2.9s	5.5s
mclc	2.66s	37.0s
viterbi	189s	1.6h
counter12	2.3h	20s
ac12	2.6h	610s
adder6	0.08s	52s

Table 5.4: Comparison of times for verification

symbolic traversal approach have been obtained using the program described in [127]. For the small examples the approach presented here works better than the approach of [30, 35]. However, as the size of the examples increase, the approach of [30, 35] gets better and soon shows much better performance than the approaches presented here. For the counter, however, the symbolic traversal approach does not perform better. It can be concluded from a study of these examples that for small to medium size controller circuits that have an irregular state space as well as circuits with long, narrow STGs, the approach of this chapter will be superior to that of [30, 35]. However, with larger circuits, especially ones that have a regular STG structure (*e.g.*, datapaths), the approach of [30, 35] will perform significantly better.

5.6 Conclusions

Two different algorithms for verification of sequential machines were presented and compared with two other algorithms. For all the examples seen, the implicit state approach is better than the approach of [46]. It was shown that verification using traversal and enumeration-simulation took about the same amount of time, though traversal required more memory. For some circuits that have STGs that are deep, the approaches presented here work better than symbolic traversal approaches. Almost all circuits for which the symbolic traversal approaches produce better results have the characteristic that they have

STGs that are regular and wide.

One of the major advantages of the approach presented here is its memory efficiency. The symbolic traversal approaches require significantly more memory than that required by approaches presented here.

The enumeration procedure presented in this chapter can produce STGs that are compact and partially state minimized. STGs of large machines which could not be derived previously can now be derived using this technique. This has obvious implications in the re-encoding and optimization of sequential circuits. The use of the techniques developed in this chapter in conjunction with other optimization techniques can be used to re-synthesize circuits to improve performance and/or testability and is the subject of the next chapter.

Chapter 6

SYNTHESIS OF SEQUENTIAL CIRCUITS

Techniques for optimizing, verifying, and testing finite state machines have traditionally relied on the use of State Transition Graph (or State Transition Table) descriptions of behavior. While the State Transition Graph is an easily manipulable representation of behavior, VLSI sequential circuits, consisting of large, interacting FSMs do not usually have compact representations in terms of STGs. In fact, STGs of even moderate-sized VLSI circuits typically require astronomical amounts of memory to store and large amounts of CPU time to generate from logic-level implementations.

The problem with conventional STG descriptions is twofold. First, the STG is a flattened sum-of-products representation. A variety of VLSI sequential circuits have combinational portions that require exponential amounts of storage in sum-of-products form (*e.g.*, xor-trees, adders). The second and the more severe problem, especially for controller-type circuits [1], is that in the STG representation corresponding to a logic-level implementation, all states are *minterm* states, *i.e.*, all state variables are set to 0/1 values in each state. This results in STGs that require a significant amount of time and space to generate.

In Chapter 5 a new algorithm for the enumeration of State Transition Graphs

[1]Typically, FSM controllers can be flattened quite easily.

was proposed. The main characteristic of the algorithm is that both the input space and the state space are enumerated implicitly, producing STGs where some equivalent states are merged into single cube states. The resultant STG (called an *Implicit State Transition Graph* or ISTG) is more compact and often has a smaller number of states and edges as opposed to those extracted using conventional techniques, as in [46]. Techniques for STG traversal (*e.g.*, [30, 35]) are geared towards visiting all the valid states in the most efficient manner possible and do not produce an STG description that can be used for synthesis. This is because information about the explicit connectivity between states and the output vector asserted for each edge is missing. However, the techniques of [30, 35] can be modified to generate STGs that can be used for synthesis.

In this chapter, sequential optimization algorithms based on ISTGs for FSMs described at the logic level will be presented. Given a logic-gate and flip-flop specification of an FSM, inherent logical decompositions into parallel or serial submachines are first identified. These may or may not be identifiable by a purely topological analysis. Depending on the size of the ISTG for a particular circuit, different synthesis strategies are used. If the ISTG has a manageable number of states and edges, existing encoding and decomposition programs are used. For ISTGs with a prohibitively large number of edges but a reasonable number of states, a dynamic re-encoding technique is proposed that does not require the storage of the entire ISTG. If a circuit has an ISTG that has both a large number of states and edges, a heuristic latch selection algorithm is used that selects submachines in the circuit corresponding to particular outputs and flip-flops as candidates for re-decomposition and re-encoding. These submachines are selected so as to have ISTGs that are of manageable size. The conventional STGs for these submachines are typically significantly larger, precluding the use of standard synthesis techniques.

The rest of this chapter is organized as follows. Previous work in the area of sequential logic synthesis is presented in Section 6.1. In Section 6.2 sequential optimization algorithms based on ISTGs for FSMs described at the logic level will be presented. Experimental results using the ISTG extraction and FSM optimization algorithms in the FLAMES system are presented in Section 6.3, followed by conclusions in Section 6.4.

6.1 Previous Work

In this section, the steps involved in sequential logic synthesis will be briefly reviewed and previous work in this area will be presented.

The first step in synthesizing sequential circuits from STG descriptions is state minimization. This has been a subject of extensive research for a long time and initial algorithms for completely specified machines were proposed in [77, 78, 103]. The most commonly used algorithm for state minimization for completely specified machines was proposed in [98]. Later, this work was extended to incompletely specified machines in [64, 96, 112].

The next important step is assigning codes to the symbolic states in a machine. The area and the performance of the resulting logic network depends strongly on the state assignment. Initial work in this area was done by Hartmanis and Stearns using algebraic structure theory [71, 73, 124]. In [5, 54] the problem of state assignment was treated from a different perspective using partitions. The concept of state splitting for obtaining better state assignment was introduced in [72, 73]. A description of all these techniques can be found in [74].

In the recent years, significant progress has been made in the area of state encoding. New algorithms for state assignment targeting two-level implementations were developed by De Micheli *et al* in [100, 101]. State assignment algorithms for multilevel IC-based implementations were first developed by Devadas *et al* in [47]. Exact algorithms for state assignment and associated problems of output encoding and four-level minimization were presented in [53]. It has been conjectured that the state assignment problem is NP-hard. For all medium-sized circuits, the above mentioned algorithms can find good state assignments. However, circuits with large number of states and edges are still not handled efficiently.

To alleviate the problem of size, while still being able to optimize large sequential circuits, approaches based on distributed-style STG representations of interacting FSMs (*e.g.*, [40, 52]) and retiming-based algorithms (*e.g.*, [86, 93]) have been used in the past. Unfortunately, to perform global optimization, the approaches of [40] require in the limit information corresponding to the entire STG of the interacting set of FSMs, which grows rapidly with circuit size. Logic-level sequential logic synthesis approaches like those in [93] hold promise as far as efficiency and accurate cost functions are concerned but to date are also lacking in global optimization capabilities. Recently, significant work has been done in the area of FSM decomposition both for performance improvement and testability [10, 8, 9]. The algorithms described in [10, 8, 9] will be used in this chapter in conjunction with new techniques for sequential logic optimization.

6.2 Optimizing Sequential Circuits

The main objective of sequential logic synthesis is to obtain an implementation of a circuit that meets area, performance, and testability requirements. Most often these conditions are not met by an initial implementation, and the designer is required to re-design the entire circuit or parts of the circuit to meet the specifications. The goal of this chapter is to provide optimization techniques that can transform a given circuit to a better one that meets the specifications.

Re-designing can be done in a number of ways. One way is to start from the very beginning, *i.e.*, from the high-level or behavioral description of the system and apply a new set of design tools or tricks, depending on the instincts of the designer, to arrive at a better design. This might involve making new decisions about the architecture of the entire system, and might require significant re-design of other components of the system. Even when re-designing other parts of the system is not required, designers are often reluctant to re-design the circuit from the very beginning, as it is a time consuming process. In the ASIC market where quick design time is of paramount importance, a designer often cannot afford the time for a complete re-design. Therefore, the ability to efficiently re-design parts of the circuit at the logic-level is of great importance.

To illustrate the necessity of re-designing parts of the circuit, consider the combinational logic block of a sequential circuit shown in Figure 6.1. The logic associated with each primary output and present state line is represented as a *cone* in the figure. Assume that after designing the circuit it is found that only output O_1 does not meet the timing specifications and that there is a redundant fault in the logic cone for that output. In this

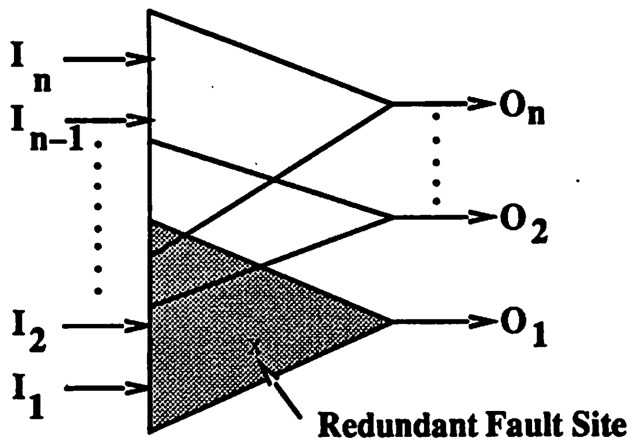


Figure 6.1: Combinational logic block of a sequential circuit

case, only the part of the circuit corresponding to the output O_1 (shown shaded) has to be re-designed to meet the testability and the performance specifications. Since O_1 shares logic with other outputs, during re-design it has to be ensured that the performance of the other parts of the circuit does not degrade.

The use of STG-based techniques to optimize logic-level sequential circuits has been severely limited by the CPU time and memory requirements posed by the size of the STG descriptions that correspond to even moderate-sized sequential circuits. Working at the logic level exclusively, on the other hand, precludes global optimization due to the lack of information necessary to optimally change the structure/encoding of the sequential machine. Global alteration of the circuit may be required to improve area, performance, or testability.

Implicit State Transition Graph extraction techniques presented in Chapter 5 have the characteristic that some equivalent states with uni-distant state codes are merged into single states thereby decreasing the number of states and edges in the STG obtained from the logic level description. One can argue that the same result can be obtained by first extracting the explicit STG and then minimizing the number of states. However, before

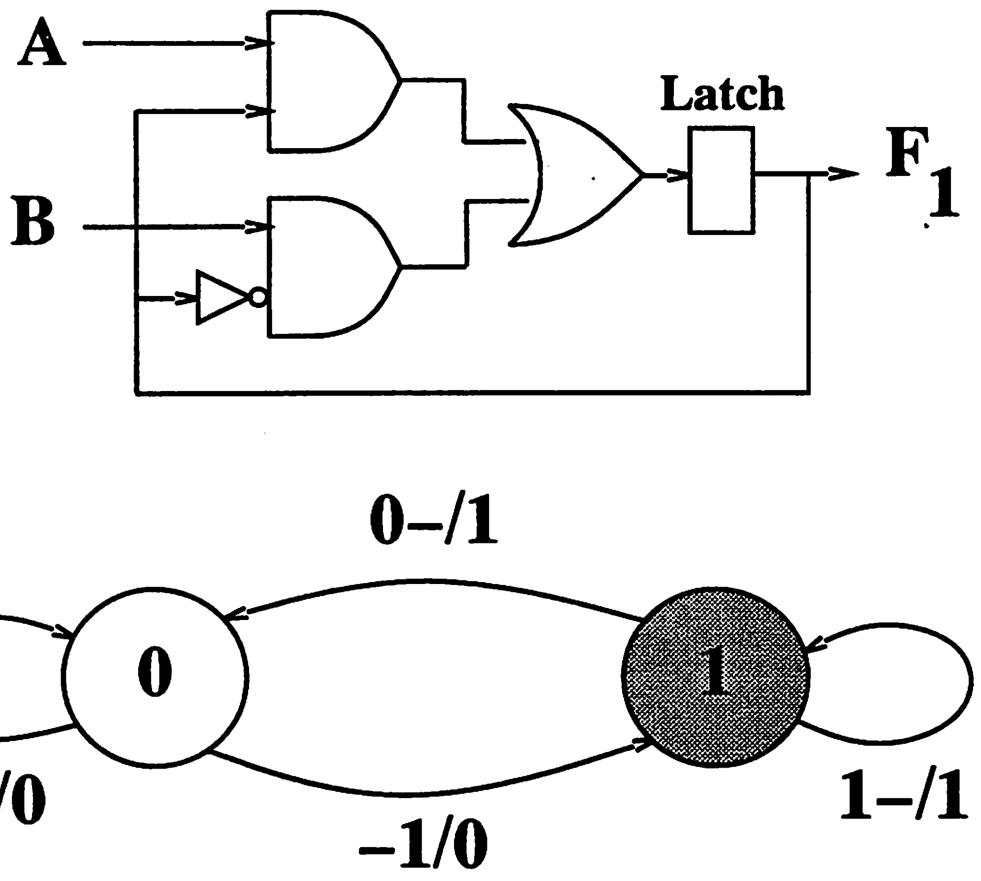


Figure 6.2: Sequential circuit before retiming

minimization, the storage of the entire explicit STG is necessary. Also, the time required for state minimization depends on the size of the explicit STG. As will be demonstrated in Section 6.3, the entire explicit STG might not be obtainable from the logic-level descriptions, though after minimization, the corresponding STG might consist of only a few states. Therefore ISTGs are important from the point of view of synthesis as they increase the size of the circuits that can be handled by traditional encoding and decomposition algorithms.

In Chapter 5, an example circuit was presented to demonstrate how equivalent states are created in interacting sequential machines. Such equivalent states are often de-

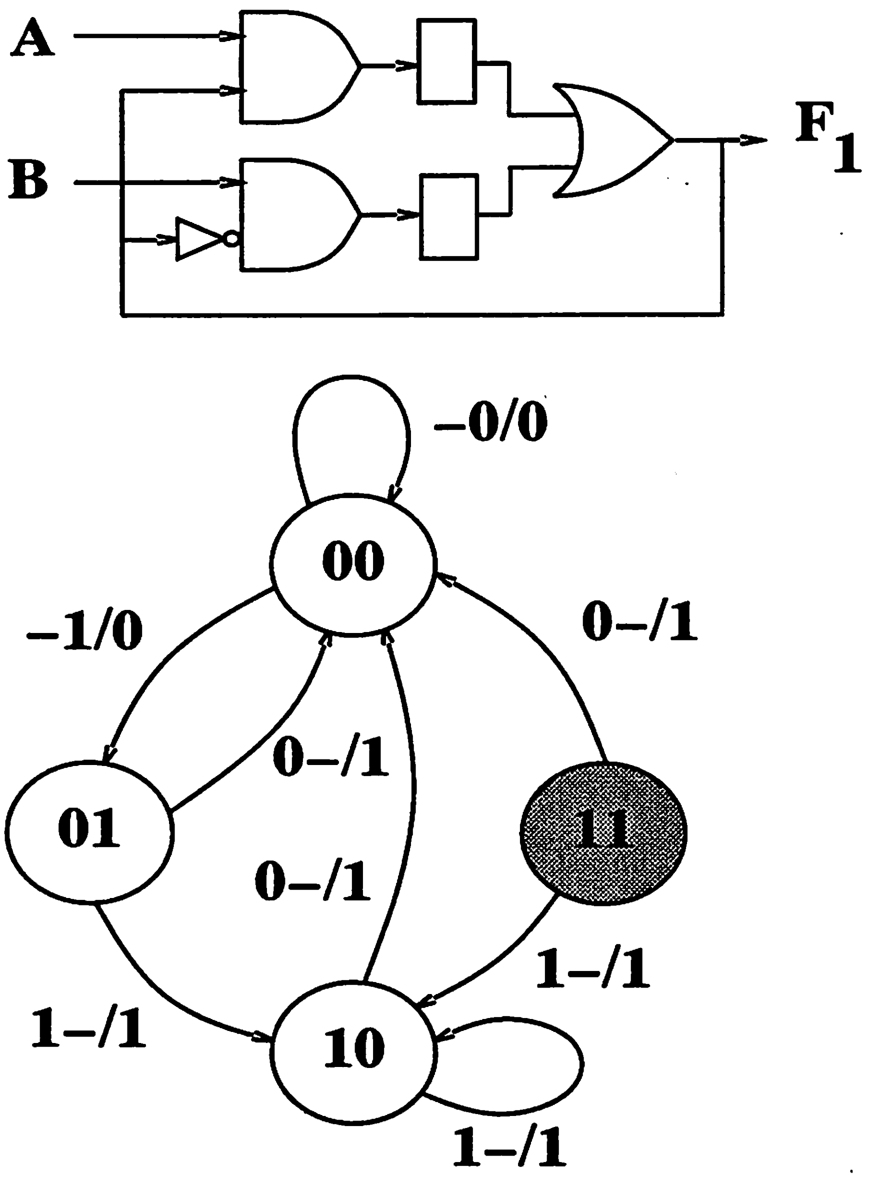


Figure 6.3: Sequential circuit after retiming

tected by the ISTG extraction routine. Apart from equivalent states arising from interacting machines, there is another reason why equivalent states arise in many sequential circuits. Consider the simple finite state machine shown in Figure 6.2. The state transition graph of

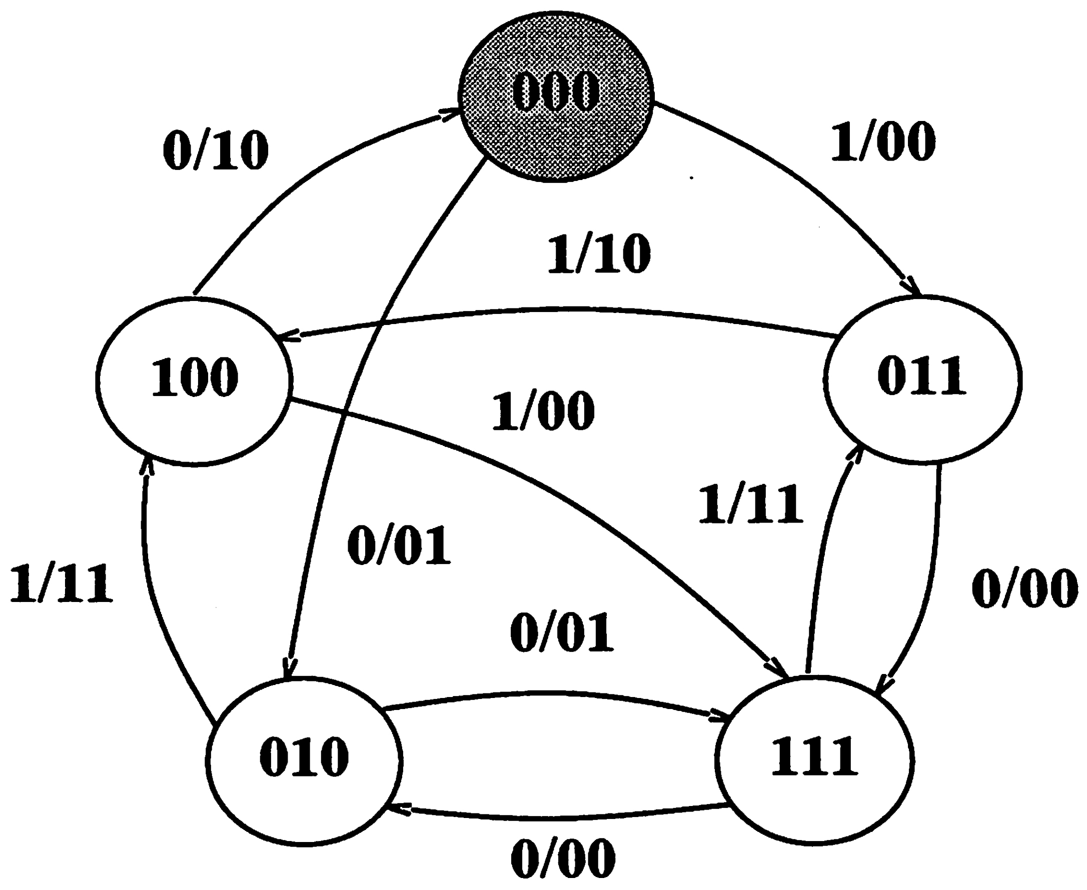


Figure 6.4: Example State Transition Graph

the machine is also shown in the figure. The reset state is state is labeled 1. This circuit can be re-timed [93] to the one shown in Figure 6.3 (though this might not be a very desirable re-timing). The latch at the output of the OR gate has been pushed to the inputs. For the OR gate, the inputs 01, 10, and 11 are equivalent, and it is expected that the states represented by these codes are equivalent. As shown in the STG of the circuit (also in Figure 6.3) that is indeed the case. Note that the reset state 11, can be merged with either 10 or 01, to form a cube state.

Since only parts of the circuit may be re-designed/re-synthesized, equivalent states

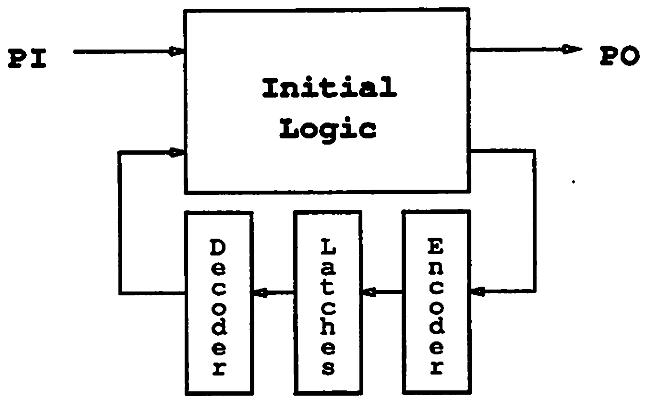


Figure 6.5: FSM with an encoder-decoder

may arise because of a different reason. Consider the FSM whose STG is shown in Figure 6.4. Considering all the outputs at a time, not a single state in this machine is equivalent to any other. However, considering only the first output, it is easily seen that states 010 and 011 are equivalent. While extracting STGs corresponding to subsets of outputs, this kind of equivalence can be utilized by the ISTG extraction program.

The optimization strategy proposed here begins with a logic-level description of a sequential circuit. Depending on the needs, either the ISTG of the entire circuit or a sub-circuit is extracted using the algorithm described in Section 5.4. This ISTG is used for subsequent re-encoding and re-decomposition. Depending on the size of the ISTG (*i.e.*, the number of states and edges), different options are exercised in the synthesis procedure. They are :

1. **Small number of states, small number of edges:** If the ISTG of the entire circuit has a small number of states (less than 1000) and a small number of edges (less than 10000), then existing decomposition [10, 8, 9] or state assignment programs [47, 131] that operate on the ISTG can be used for the global optimization of the circuit. It may so happen that only certain outputs in the circuit have small-sized ISTGs. In that case the alternate strategies given below have to be followed for the remaining

outputs.

2. **Small number of states, large number edges:** If the ISTG of a particular output has a small number of states (less than 1000) but a large number of edges (greater than $10,000$), then it is not CPU-time efficient to use decomposition programs like those in [10] and state assignment programs like NOVA [131]. This is because the CPU time requirement for these programs depends strongly on the number of edges as well as the number of states in the STG. In this case a dynamic state assignment strategy that does not require storage of the ISTG is used. It is a two-pass procedure. In the first pass, the ISTG extraction program is used to obtain all the valid states in the machine without storing each edge in the STG. In the second pass, given the symbolic state information, the edges in the ISTG are inspected one by one to determine a good adjacency-based coding for the states, much like the *counting algorithms* in MUSTANG [47]. Again, there is no need to store the entire set of edges. Once a new encoding for the states has been constructed, an encoder and a decoder are added to the sequential circuit as illustrated in Figure 6.5. For instance, a state originally may have the code $10-1-1$ and may be re-assigned the code 0000. The combinational logic (that now includes the encoder and decoder) of the sequential circuit is then optimized for area or performance using programs like MIS [15], potentially leading to an improved implementation.

3. **Large number of states, large number of edges:** In this case the first step is to reduce the number of states in the ISTG to be manipulated by attempting to find a good decomposition of the circuit. At first submachines in the circuit that correspond to a parallel decomposition inherent in the circuit are searched for. An example of a parallel decomposition is shown in Figure 6.6(a). Failing that, the next step is to find a cascade decomposition (*cf.* Figure 6.6(b)). If the circuit does not possess an inherent cascade decomposition, a good general decomposition (*cf.* Figure 6.6(c)) is obtained by suitably choosing subsets of latches that would form each component machine. Selecting any subset of latches corresponds to identifying a submachine in some general decomposition. It is of interest to select a subset of latches and outputs

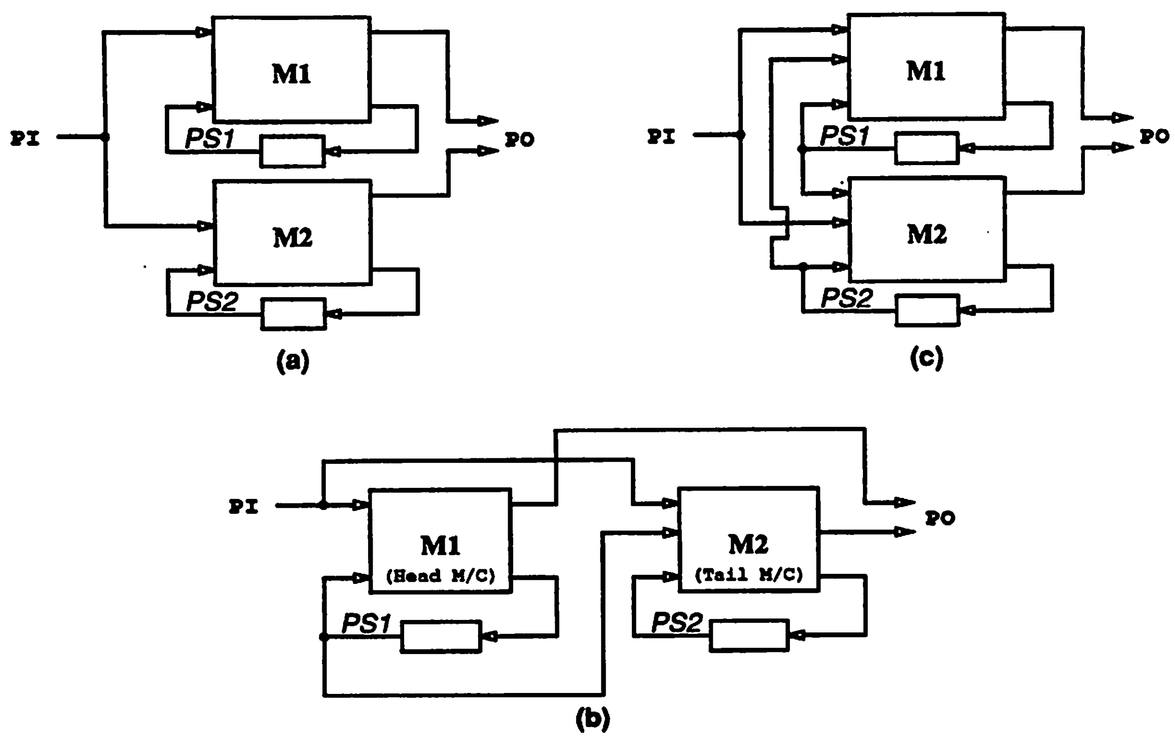


Figure 6.6: FSM decomposition types

such that the submachines that are so created interact minimally. Each of these submachines would have a much smaller number of states than the original FSM, making it possible to apply some of the techniques in options (1) and (2) above for re-encoding. Heuristics for the selection of latches are described in [7].

Counting algorithms like those of MUSTANG [47] compute a state assignment based on adjacency relations between states. It is not necessary to store an entire STG or ISTG in order to do this computation. Storing all the distinct states in the machine and a small set of edges currently being inspected is sufficient. After a new encoding is found, given that an encoder and decoder can be introduced as shown in Figure 6.5, large circuits can be handled.

If the circuit has an inherent decomposition, this can be used to advantage in re-

CKT	#I	#O	#L	#G	O/P #	Size ISTG		Size STG	
						States	Edges	States	Edges
viterbi	11	34	12	227	0-3	15	48	359	8421
					4-7	8	159	136	6464
sbc	35	51	33	1011	11-12[1]	6	70	*	*
					27-32	1	1	*	*
					48[1]	6	190	*	*
tlc	3	6	21	162	0-5[1]	34	865	*	*
s344	9	11	15	160	4	258	569	*	*

* Conventional STG could not be obtained.

Table 6.1: Example circuits

encoding. In general, state assignment strategies work better on decomposed, rather than lumped circuits [52]. In the case of a parallel decomposition, either of the two machines can be re-encoded without affecting the other. In the case of a cascade decomposition, the tail machine can be re-encoded without affecting the head machine, but if the head machine is re-encoded, then appropriate encoding logic has to be added to the tail machine, since the tail machine receives present state inputs from the head machine. Lastly, in the case of general decomposition, re-encoding either machine requires this additional encoding circuitry for the other.

Adding the extra logic in the form of encoders and decoders seems to increase the logic in the circuit, thereby adversely affecting the area and performance of the resulting circuit. In the algorithms proposed, the circuit obtained after adding the encoder and the decoder is optimized using logic synthesis systems like MIS[15]. If the new encoding corresponds to a better actual implementation, then after introducing the encoder and the decoder and re-optimizing the combinational logic block, it is expected that the total area will decrease and the performance will improve. The results of Section 6.3 validate this claim.

6.3 Results using FLAMES

The strategies described above have been implemented in a sequential synthesis system called FLAMES[11]. It is a powerful system incorporating various logic optimization and sequential logic synthesis algorithms. The sequential logic synthesis strategies described in this chapter have been tried on a number of example sequential circuits. In all the cases the starting point was a logic-gate and flip-flop description of the circuit. Some of these examples could not be described in terms of conventional STGs.

Table 6.1 gives the statistics of the examples and the sizes of their corresponding STGs. The first example is a set of FSMs forming the controller of the **viterbi** processor [125]. As it happens, the description for the controller contains three parallel FSMs asserting different sets of outputs but which are driven by the same set of primary inputs. After combinational logic optimization however, this decomposition could not be identified via a topological analysis. STGs of all of the outputs were obtainable using the program of [46] but were large. On the other hand, compact ISTGs for each output were obtainable using the techniques described in Section 5.4. Extracting the ISTG for any particular output takes advantage of the inherent parallel decomposition, *i.e.*, the fact that the output under consideration is functionally dependent only on a subset of the latches and therefore all the states with the same value in this subset of latches are equivalent as far as this output is concerned. In a sense, the ISTG extraction method attempts to extract the minimal amount of symbolic information necessary to be able to optimize that particular output. In an initial pass, ISTGs were extracted for each output separately. After this pass, the outputs that were found to be dependent on the same set of latches (the outputs that are asserted by the same component FSM in the parallel decomposition) were clustered together and a single ISTG was extracted for each such cluster of outputs. In the results presented here, only two such clusters are considered.

The next two examples are large FSM controllers **sbc** and **tlc**. Conventional STG descriptions for almost all of the outputs (considered individually) for either of these controllers are not obtainable. On the other hand, manipulable ISTG descriptions could be obtained for most of their outputs. In addition, the number of outputs for which manipulable ISTG descriptions could be extracted was further increased by only re-encoding an appropriate subset of latches. When it is required to re-encode only a subset of latches, only the symbolic information corresponding to that subset need be extracted. The latches not in the chosen subset can be considered to be primary inputs.

Both the examples have inherent two-way cascade decompositions (detectable by the algorithms in [7]) that can be identified after cover extraction but not via a topological analysis. The subsets of latches to be re-encoded were formed by grouping together the latches belonging to a single component machine in the inherent decomposition. After exploiting the cascade decomposition in the case of **sbc**, about two-thirds of the outputs had a moderate number of states and edges and did not require further latch selection. However, the remaining outputs either had too large a number of edges or states or both. In the case of **tlc**, once the inherent cascade decomposition had been identified, all the outputs could easily be re-encoded using ISTGs. The last example **s344** is from the ISCAS Sequential Testing Benchmark set.

Some of the significant synthesis results have been provided in Table 6.2. In both Table 6.1 and 6.2, #I is the number of primary inputs, #O the number of primary outputs, #L the total number of latches, #G the number of gates, and O/P # indicates the output(s) being optimized. The initial areas and delays correspond to the best area and delay figures obtained via combinational logic optimization on the initial logic-level description using MIS [15]. The final areas and delays correspond to the the figures obtained after sequential logic synthesis from the extracted symbolic information. The area was measured using the factored-form literal count in MIS and the delays were measured using the *mapped* delay model in MIS. Note that the optimization of the initial circuit and the re-synthesized circuit were performed using the same set of MIS commands (*mis script*).

Significant area and performance improvements were obtained for **viterbi** and **sbc**, as indicated in Table 6.2. The conventional STGs could not be obtained for any of the outputs of **sbc** while some of the ISTGs for the same outputs are extremely compact. *For instance, some of the outputs* **(27-32)** *became wires since all states are single-cycle equivalent.* Using the algorithm of [46] to extract a conventional STG for outputs **27-32** would result in a STG with millions of equivalent states. Using state minimization would eventually produce the same area gain but such a method would require exorbitant amounts of CPU time as well as memory.

The ISTGs for the examples **tlc** and **s344** are much more compact than the cor-

CKT	#I	#O	#L	#G	O/P #	Delay[2] Init./Fin./Impr.	Area Init./Fin./Impr.
viterbi	11	34	12	227	0-3	32/15.7/51%	321/69/78%
					4-7	30.5/23.9/22%	319/85/73%
sbc	35	51	33	1011	11-12[1]	11.8/10.5/11%	632/538/15%
					27-32	19.5/0/100%	397/0/100%
					48[1]	12.4/9.6/23%	630/550/13%
tlc	3	6	21	162	0-5[1]	18.5/18.5/0%	161/161/0%
s344	9	11	15	160	4	20.7/20.7/0%	131/131/0%

[1] Incorporating latch selection heuristics.
[2] Using the mapped delay-model in MISII [15].

Table 6.2: Synthesis results using FLAMES

responding conventional STGs. However, for these examples, re-encoding did not provide any area or performance gain. After extensive experimentation it has been found that given large ISTGs, encoding programs like MUSTANG [47] and NOVA [131] are either unable to find state assignments that are comparable to the initial literal count, or do not complete in reasonable amounts of CPU time. Different encoding strategies are necessary. Some larger circuits were tried but either ISTGs could not be extracted or the resulting STGs were too large for the encoding and the decomposition programs to handle.

6.4 Conclusions

The size of sequential circuits for which current sequential logic synthesis strategies are viable can be increased quite significantly using Implicit State Transition Graphs (ISTGs). In this chapter, the focus has been on a sum-of-products representation for ISTGs, since the most mature state assignment and decomposition strategies in use today target and use such a representation. However, the algorithms and ideas presented here can quite easily be modified to use alternate representations of Boolean functions as a base.

Certain classes of sequential circuits usually described at the logic-level, notably, ALUs interconnected with registers, are not amenable to sequential logic optimization.

While algorithms exist that can verify/test such circuits, improving the performance of such circuits using re-encoding or re-decomposition appears improbable. Selection strategies that can focus on the control portions of a logic-level sequential circuit, where the most room of sequential optimization exists, are a must for large, real-life chips. The algorithms described in this chapter are best suited for the control portions of circuits.

State-of-the-art state assignment programs like KISS, NOVA, and MUSTANG cannot handle really large circuits, *i.e.*, circuits that have a large number of states and edges. At the present moment, the ISTG extraction program can extract ISTGs of circuits which cannot be re-encoded by these state assignment programs. Finite state machine decomposition techniques should help in the future by producing smaller component machines that can be encoded. However, better encoding strategies are necessary.

Though cubes and covers were used for the enumeration and the synthesis procedure, other representations can be used. In particular, BDDs can be used to improve the efficiency of the ISTG extraction process. However, all state assignment programs to date use a State Transition Table type representation of the STG. There is a need for developing strategies that would use a BDD-based representation of the STG.

Though ISTGs are more compact and can be better used than conventional STGs, there are still some drawbacks. Firstly, all equivalent states are not detected, so some state minimization might be necessary. Moreover, if equivalent states do not have uni-distant state codes, they cannot be combined to obtain a more compact STG. This drawback can be removed by using BDDs to represent states. Another important drawback is that the detection of equivalent states is strongly dependent on the heuristics used in setting the input variables during the enumeration process. Lastly, this approach cannot handle circuits with more than 30 latches, as even the ISTGs become too large to manipulate.

Chapter 7

HEURISTIC MINIMIZATION OF BOOLEAN RELATIONS

Most of the work in logic synthesis has been on Boolean functions, which are one-to-one or many-to-one multi-output Boolean mappings. A Boolean relation is a one-to-many multi-output Boolean mapping. Boolean relations are a generalization of incompletely specified logic functions. Typically, for such functions, a set of input values is specified for which one or more outputs can be either 0 or 1, *i.e.*, a *don't-care*. Logic minimizers like ESPRESSO [16] can utilize this information to obtain smaller sum-of-products representations of functions. However, not all aspects of incomplete specification can be captured using don't-cares, especially for multi-level logic networks. Consider the example shown in Figure 7.1. In this figure there are two PLAs, with PLA_1 driving PLA_2. The logic function of PLA_1 is represented by the truth table shown in Figure 7.2. If f_1 and f_2 are intermediate variables, *i.e.*, they are used only as inputs to PLA_2, then it does not matter whether PLA_1 produces 00 instead of 11 because PLA_2 maps both 00 and 11 to the same value 0 at its output. Similarly, PLA_1 could produce either 01 or 10. The output patterns 00 and 11 are equivalent and so are 01 and 10. This kind of incomplete specification cannot be represented as output don't-cares but only as a Boolean relation as shown in Figure 7.3 [18].

For every minterm in the input space of a Boolean relation, there is a set of outputs,

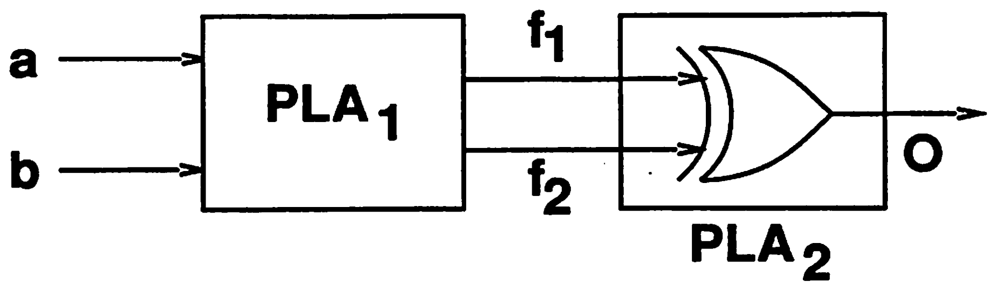


Figure 7.1: PLA driving another PLA

and any output from the corresponding set of outputs can be chosen for that minterm to form a function compatible with the Boolean relation. Note that output don't-cares can be easily represented using Boolean relations, as illustrated with the example in Figure 7.4.

If the function shown in Figure 7.2 is minimized, the minimized function is

$$\begin{aligned} f_1 &= ab + \bar{a}\bar{b} \\ f_2 &= a \end{aligned}$$

However, if a different function is chosen so that it has the output 10 for the first and third input minterms and the output 00 for the second and fourth, the minimized function becomes

$$\begin{aligned} f_1 &= \bar{b} \\ f_2 &= 0 \end{aligned}$$

The overall behavior of the PLA network remains unchanged. As is evident from this example, significant savings in network size may be obtained by exploiting Boolean relations for the minimization of PLA_1.

Boolean relations arise in several contexts. One of them is the situation described above where one PLA drives another and the driven PLA maps two input patterns x and y to the same output and therefore for the driver PLA x and y are equivalent outputs. The

a	b	f_1	f_2
0	0	1	0
0	1	0	0
1	0	0	1
1	1	1	1

Figure 7.2: Truth table of PLA_1

other situation was presented in Chapter 4 where a finite state machine has sets of equivalent states. For each present state and input, the machine can go to any one of the states in the set of equivalent next states. This situation can be easily represented using Boolean relations. Minimization of Boolean relations helps in deriving smaller logic networks and also in synthesizing circuits for testability [49].

For each minterm in the input space of a Boolean relation, any one of the patterns from the set of output patterns for that minterm can be chosen. For each choice of outputs a Boolean function [16] is obtained. In applications, the goal is to implement one of the functions, generally the one with the least cost. Thus the problem of minimization is two-fold. A function with the least cost after minimization has to be identified first and then minimized. In this chapter, the focus is on two-level sum-of-product representations of functions. In the sequel, a fast heuristic minimization algorithm is presented. This procedure makes use of test generation techniques to determine a prime and irredundant (minimal) implementation of a function compatible with a Boolean relation. The test generation based minimization techniques used here are somewhat similar to those used in [57].

The rest of the chapter is organized as follows. Definitions are presented in Section 7.1, followed by a brief review of previous work in Section 7.2. In Sections 7.3 and 7.4, the main minimization and network formation procedures are described. The expansion procedure is presented in Section 7.5, followed by the irredundant cover procedure in Section 7.6. In Section 7.7, the reduction procedure is described. In Section 7.8, the last

a	b	$\{f_1 f_2\}$
0	0	$\{10, 01\}$
0	1	$\{00, 11\}$
1	0	$\{01, 10\}$
1	1	$\{11, 00\}$

Figure 7.3: Boolean relation for PLA_1

procedure for minimization, makesparse, is presented. The procedure for choosing a function compatible with a Boolean relation is presented in Section 7.9. Results presented in Section 7.10 demonstrate the viability of this approach and its superiority over exact minimization approaches for large circuits. Conclusions are presented in Section 7.11.

7.1 Definitions

Throughout this chapter it is assumed that the reader is familiar with the terminology of [16] and [18].

Definition 7.1.1 *A Boolean relation [18] is a one-to-many multi-output Boolean mapping, $\mathcal{R} : B^r \rightarrow B^n$, where $B = \{0,1\}$. Thus $\mathcal{R}(\mathbf{x}) \subseteq B^n$ is a set. For each input minterm $c \in B^r$, a set of primary output vectors can be asserted by that minterm. The set of primary output vectors corresponding to that minterm c of the relation is called the* **equivalence class** *for the minterm and is denoted by $O_E(c)$.*

The **specification of a Boolean relation** is a set of cubes (which could be either minterms or cubes with don't-care entries) and their corresponding equivalence classes $(c|O_E(c))$. The specification is given as in Figure 7.3. The cubes in the specification might overlap. In such cases, for the minterms common to the overlapping cubes, the equivalence class is the set union of the equivalence classes of the cubes in which the minterms are present.

a	b	f_1	f_2		$\{f_1 f_2\}$
0	–	1	–	$\rightarrow$	$\{10, 11\}$
1	0	0	0	$\rightarrow$	$\{00\}$

Figure 7.4: Representation of don't-cares

Definition 7.1.2 *A multi-output Boolean function f is a* **function compatible with** *a Boolean relation R if for every minterm $x \in B^r$, $f(x) \in O_E(x)$.*

A set of cubes $(c|y)$, where $c \in B^r$ and $y \in O_E(c)$, constitute the **cover of a Boolean function.** The cover is said to be **valid** if the function corresponding to the cover is compatible with the relation R.

Definition 7.1.3 *Two implementations f_1 and f_2 of a Boolean relation R are* **equivalent** *if f_1 and f_2 are mappings compatible with R. Thus for any minterm x, $f_1(x)$ and $f_2(x)$ must be elements of the set $O_E(x)$.*

Definition 7.1.4 *A cube in the cover of a function f which is compatible with a Boolean relation R is said to be* **prime** *if raising any input or output literal causes the resulting function to be non-equivalent to f.*

Definition 7.1.5 *A cube in the cover of a function f which is compatible with a Boolean relation R is said to be* **irredundant** *if on removal of the cube from the cover, the resulting function is not equivalent to f.*

Definition 7.1.6 *A cover of a function f which is compatible with a Boolean relation R is said to be* **prime and irredundant** *if every cube in the cover is prime and irredundant. The function corresponding to the cover is therefore called a prime and irredundant function.*

7.2 Previous Work

In the mid-80's, researchers in logic synthesis believed that all kinds of incomplete specification could be represented using don't-cares. In [18], Brayton and Somenzi first showed that certain kinds of incomplete specification could not be captured using don't-cares. They were the first to use Boolean relations and to show that it was a generalization of traditional don't-cares.

The first attempt to minimize Boolean relations was also undertaken by Brayton and Somenzi in the work presented in [19]. A complete algorithm and results were presented in [17]. In the last paper, an exact procedure for the minimization of Boolean relations was given. The problem was formulated as a linear integer 0-1 program and a branch and bound covering method was given to find the minimum cover. The problem with this approach is its exponential complexity, both in terms of CPU time and memory required to generate and store all the *c-primes* [17] and performing the constraint generation and binate covering. Thus, it can be applied only to small examples. To date, no other method for minimization of Boolean relations has been published.

7.3 Minimization algorithm

Minimization of a Boolean relation can be viewed as a two-step process. The first step is the choice of a function and the second step is the minimization of the function. The objective in the first step is to choose the function which when minimized will have the optimum implementation. An optimum implementation is characterized by :

- minimum number of product terms in the cover
- minimum number of literals in each product term.

A cost function is defined in terms of the above factors and the minimization procedure tries to minimize the cost function.

A naive approach to minimization would be to form all possible functions compatible with a Boolean relation and minimize them (exactly or approximately) using logic

```
MINIMIZE(RelationCover)
{
  /* RelationCover is the specification of a Boolean Relation */
  /* FunctionCover is the cover of a Boolean Function */
  /* The optimization loop */
  /* First choose a particular function compatible with relation */
  FunctionCover = Choose_Function(RelationCover);
  Build_Interconnected_Network(FunctionCover, RelationCover);
  while (quality of solution keeps on improving){
    Expand (FunctionCover);
    Irredcover (FunctionCover);
    if (latest cover better than previous cover){
      save latest cover;
    }
    Reduce (FunctionCover);
  }
  FunctionCover = best cover;
  Makesparse (FunctionCover);
  return (FunctionCover);
}
```

Figure 7.5: Main minimization procedure

optimizers such as ESPRESSO [16]. This approach is a modified exact minimization approach (if ESPRESSO-exact is used) and is not suitable for real examples due to the large number of minimizations required.

The approach described in this chapter starts with an initial function compatible with the Boolean relation and through a series of iterations involving the procedures **Expand**, **Irredcover**, and **Reduce** obtains a function with a smaller cost. Instead of trying to explicitly minimize every possible function compatible with a Boolean relation, in each of these steps, the initial function is *implicitly* changed to a function equivalent to it but with a smaller cost. *Due to the lack of ways of predicting which initial function would produce the best result, the initial choice is made somewhat randomly.* However, it has to be ensured that the initial function chosen is compatible with the Boolean relation. The procedure for

f_1	f_2	O
0	0	0
1	1	0
0	1	1
1	0	1

Figure 7.6: Truth table for PLA_2

deriving a function compatible with a Boolean relation is given in Section 7.9. Since there is no known polynomial-time algorithm to solve the Boolean function minimization problem, which is conjectured to be NP-hard, the worst-case complexity of this approach is exponential. However, with clever heuristics, a solution can be found within reasonable amounts of CPU time in the average case. Note that the correspondence between the minimization algorithm presented here and that of ESPRESSO is in the general paradigm for minimization (*i.e.*, in the use of the **Expand**, **Irredcover**, and **Reduce** steps). The procedures used for each step are different. As will be pointed out later, these procedures when used for the minimization of a Boolean function are, in a sense, less powerful than the corresponding procedures in ESPRESSO.

The procedures make extensive use of test pattern generation techniques. To make the use of such techniques feasible, the first step in the main minimization procedure (Figure 7.5) is to build a network of gates called the *interconnected network*. This interconnected network is a connection of two PLAs, as in Figure 7.1. The first PLA implements the initial function compatible with the Boolean relation. The second PLA is built so that the interconnection of PLAs produce the same Boolean relation for the first PLA as the original Boolean relation. The procedure **Build_Interconnected_Network()** receives as its input the specification of a Boolean relation and a random initial function compatible with the Boolean relation, as chosen using **Choose_Function()**. Having derived the network, a gate-level automatic test pattern generator is used for the various steps in the main loop. The use of this network and test pattern generation techniques will be illustrated in

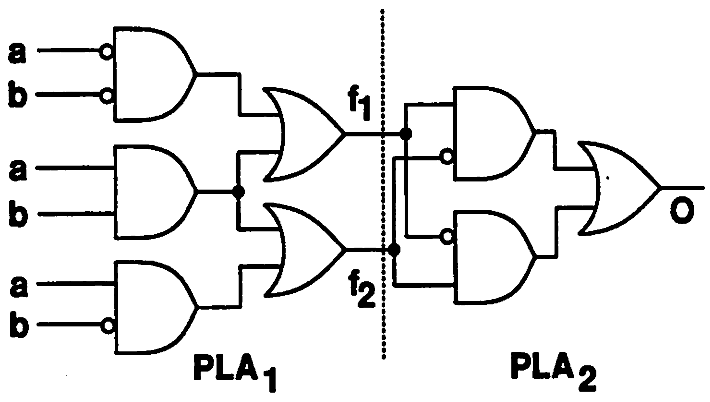


Figure 7.7: Interconnected PLA network

subsequent sections.

The goal of the **Expand** procedure is to make each cube in the cover prime while keeping the function compatible with the Boolean relation. **Irredcover** identifies a set of cubes that can be removed from the cover without affecting compatibility. Thus the cardinality of the cover is reduced and all cubes in the cover are necessary. In an attempt to get out of the local minima, **Reduce** is used to reduce the size of each cube as much as possible while maintaining compatibility with the Boolean relation. The reduced cubes can possibly re-expand to cover more cubes and reduce the cardinality of the cover. After reduction, the operations in the loop are repeated. Iteration continues as long as the quality of the solution keeps improving. When the quality of the solution does not improve any more, the loop is exited and **Makesparse** is applied to make the cover as sparse as possible. At all stages, various heuristics are used to guide the algorithm in order to obtain the best possible solution.

7.4 Network Formation

The main steps in the minimization algorithm are performed by a test pattern generator. To enable the use of a test pattern generator it is necessary to build a network

a	b	$\{f_1 f_2\}$
0	0	$\{00, 01\}$
0	1	$\{01, 10\}$
1	0	$\{10\}$
1	1	$\{11\}$

Figure 7.8: Example Boolean relation

of gates from the cover of the function. The objective of the network formation procedure is to form an interconnected network consisting of a driver network and a driven network as in Figure 7.1. This interconnection should be such that the Boolean relations for the driver network arising from this interconnection is the same as the original Boolean relation. Since we are interested in two-level sum-of-product representations, PLAs are the obvious choice for building such a network. The cover of the initial function chosen can be easily translated to a PLA where each product term corresponds to an AND gate, with the inputs to the gate denoting the input literals for the product term. This PLA is called the **driver** PLA. Following that, another PLA driven by the driver PLA has to be constructed. The second PLA is called the **driven** PLA.

Intuitively, the second PLA should be such that the output patterns of the first PLA which are in the same equivalence class (remember the outputs of the driver PLA are inputs to the driven PLA) should be mapped to a single pattern at the output of the driven PLA. Though this is the general principle, this step of deriving the driven PLA is performed in two different ways, depending on the Boolean relation. At first, all unique equivalence classes of the Boolean relation are identified. If a particular output pattern never occurs in more than one equivalence class, then the driven PLA should only map the patterns in the same equivalence class to a single pattern at its output. To illustrate this with an example, consider the Boolean relation shown in Figure 7.3. There are only two unique equivalence classes $\{01, 10\}$ and $\{11, 00\}$, and they do not have any patterns in common. Therefore the driven PLA should map 01 and 10 to a single pattern at its output and also map 11

a	b	f_1	f_2	O
0	0	0	0	1
0	0	0	1	1
0	1	0	1	1
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Figure 7.9: Truth table for PLA_2

and 00 to another pattern. In this manner, the truth table for the second PLA is derived and is shown in Figure 7.6. If the initial function chosen is as shown in Figure 7.2, the interconnected network that is formed is shown in Figure 7.7.

If a particular output pattern occurs in more than one equivalence class, then the inputs to the driver PLA also have to be considered as inputs to the driven PLA. In this case, the driven PLA is the *characteristic function* of the Boolean relation. To illustrate this, consider the Boolean relation shown in Figure 7.8. There are four unique equivalence classes and the pattern 01 occurs in two of them. The driven PLA has to be constructed so that under the input condition $a = 0$, $b = 0$, it maps $f_1 = 0$, $f_2 = 0$ and $f_1 = 0$, $f_2 = 1$ to a single pattern at its output. This is illustrated by the first two cubes in the truth table for PLA_2, as shown in Figure 7.9. Continuing in this manner, the truth table for the second PLA can be obtained. The resulting function is called the characteristic function of the Boolean relation. The interconnected network (with the initial function obtained by choosing 01 as the output for both the first and second minterm) is shown in Figure 7.10. It is easy to show that the Boolean relation that can be derived from the interconnected network is the same as the original one.

Note that this method of deriving the driven PLA could have been applied to the previous case where the equivalence classes were non-intersecting. However, the method applied there produces a smaller driven PLA in general. Since the time required for test generation depends on the size of the network, the previous method helps speed up the

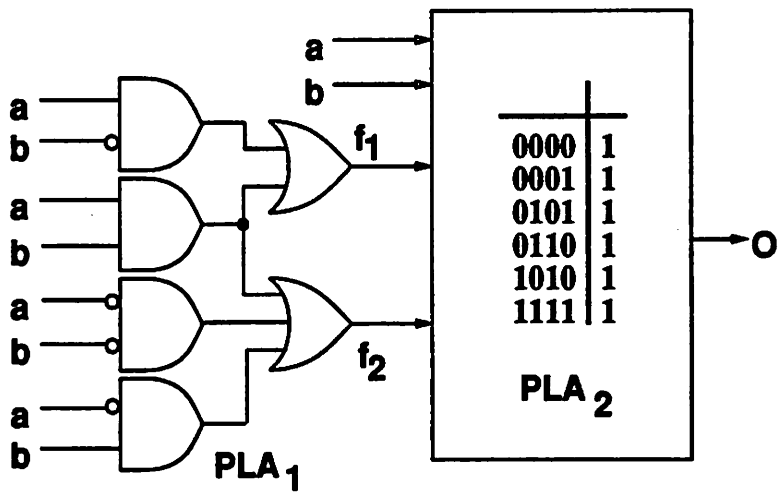


Figure 7.10: Interconnected PLA network

algorithm. The size of the driven PLA is not affected by the size of the driver PLA, and does not affect the quality of the final result.

7.5 Expand

In this section, the **Expand** procedure is described. This procedure tries to identify a function with a smaller cost and simultaneously minimizes that function.

It is well known for two-level AND-OR logic networks that if the function being implemented is prime and irredundant [16], then it is testable for all single stuck-at faults in the network. If a cube in the cover is prime, all input stuck-at-1 faults for the corresponding AND gate are testable. Conversely, if a cube is not prime, then for certain inputs to the corresponding AND gate, stuck-at-1 faults are not detectable. These inputs are therefore redundant and can be removed to make the cube prime.

Given a Boolean relation and an initial function compatible with the relation, a cube in that function is prime if and only if raising any literal (either in the input part or the output part) produces a function that is not equivalent to the original function. To

a	b	f_1	f_2
0	0	1	0
0	1	1	1
1	0	0	1
1	1	1	1

Figure 7.11: Function cover after Expand

determine whether a cube is prime or not, it is necessary to raise all the literals in the input part and the output part of the cube, one at a time, and check if the resulting function is compatible with the Boolean relation. Looking at this from the point of view of test pattern generation, an input literal in a cube is redundant if for all stuck-at-1 fault tests for the literal, the response of the fault-free network and the network with the stuck-at-1 fault are in the same equivalence class (which is the equivalence class for the test pattern). Since the driven PLA maps patterns in the same equivalence class to the same output, this means that the effect of the fault will not be observable at the outputs of the driven PLA. This gives a simple criterion for determining whether an input literal is redundant or not. If a stuck-at-1 fault for that literal in the interconnected network is undetectable, then the literal is redundant and can be removed. Another objective during this phase is to add redundant fanout stems to AND gates, *i.e.*, raise the output part of the cube. For every AND gate and OR gate between which a connection does not exist yet, the connection is made. If a stuck-at-0 fault on the connection is undetectable, it is retained.

To illustrate the procedure with an example, consider the Boolean relation of Figure 7.3. The initial function chosen is shown in Figure 7.2 and the resulting interconnected network is shown in Figure 7.7. Consider the second AND gate and the input a stuck-at-1. The only input vector that can excite the fault is $a = 0$ and $b = 1$. For this vector, the response of the fault-free driver PLA is 00 and the faulty driver PLA is 11. The equivalence class of the test vector, 01, contains both these responses. It can be easily determined that a stuck-at-1 fault on that input is undetectable in the interconnected network, and

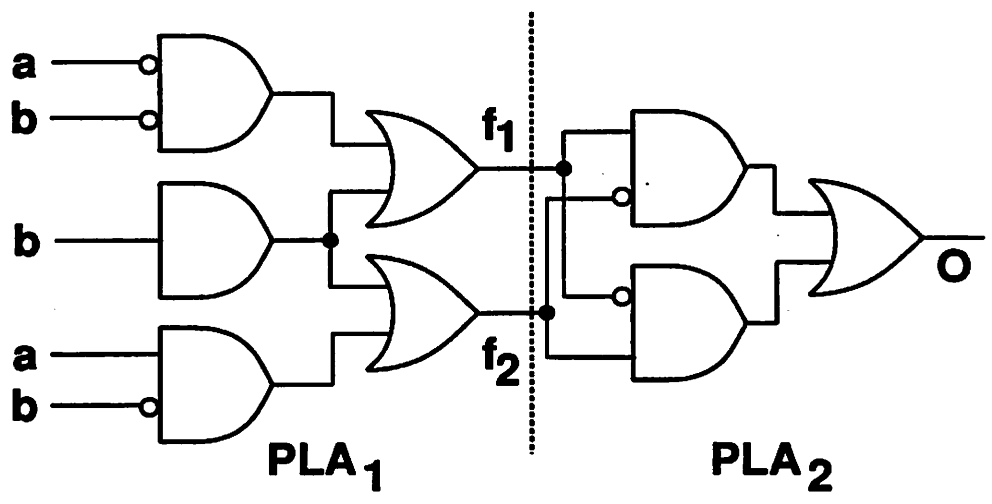


Figure 7.12: Network after Expand

therefore the corresponding literal can be removed from the cube. The driver PLA now implements the function shown in Figure 7.11, which is different from the initial function. Thus a new function with a smaller cost has been *implicitly* chosen. Note that the cube after expansion is a prime of the current function, indicating that the chosen function has been simultaneously minimized.

The **Expand** procedure is based on the aforementioned principles. The AND gates in the driver PLA are ordered and all literals in each gate are checked for redundancy. Also, new fanout stems are added to AND gates and checked for redundancy. All redundant inputs are removed as soon as they are detected. On the other hand, redundant fanout stems for AND gates are retained. The process continues *iteratively* until there are no more redundant literals to remove or fanout stems to add. The procedure, since it is iterative, stops only after going through all the cubes and fails to raise any literals. Since no literals in any of the cubes can be raised, all the cubes must be prime. For test pattern generation and efficient identification of undetectable faults, the test pattern generation algorithm PODEM [65] is used with modifications suggested in [60] and [119].

The result of **Expand** depends strongly on the order in which the cubes are expanded. Cubes are ordered in decreasing size of their equivalence class. The rationale behind this choice is that a cube with a larger equivalence class can possibly expand more and cover other cubes in the cover. The input variables in the cube are also ordered before expansion. Variables are ordered according to the number of gates they fanout to. Variables fanning out to a larger number of AND gates are considered first. This greedy strategy tries to produce PLAs with better folding characteristics [51] and works better in the average case. To add redundant fanout stems to an AND gate, the OR gates are ordered so that ones that have a maximum number of common input literals with the AND gate are considered first. Ties are broken arbitrarily. Since the support of the OR gate and the AND gate is similar, the chance of the connection being redundant is higher. This is the rationale behind this choice.

This procedure is different from the corresponding procedure in ESPRESSO. The major difference is that literals here are considered serially, while in ESPRESSO, sets of literals can be considered using *Blocking* and *Covering* matrices. Therefore the chances of this procedure getting stuck in a sub-optimal solution are higher. However, since there is no analog of an OFF-set for a test-pattern-generation-based algorithm, the ESPRESSO-type algorithm cannot be used.

7.6 Irredcover

After expansion, each cube in the cover is prime. However there may be some cubes in the cover that are redundant, *i.e.*, the cardinality of the cover may be decreased by deleting these cubes. Like the **Expand** procedure, the objective of this procedure is to identify a function with a lower cost and minimize that function simultaneously.

The principles used are similar to the ones used in the previous section. For a two-level AND-OR logic network, a cube is redundant if a stuck-at-0 fault at the output of the corresponding AND gate is undetectable. For a Boolean relation, the condition for being redundant can be expressed in the following manner. In the interconnected network, if a stuck-at-0 fault at the output of an AND gate in the driver PLA is undetectable, then the corresponding cube is redundant and can be removed from the cover. In this procedure,

a	b	f_1	f_2
0	0	1	0
0	1	0	0
1	0	0	1
1	1	0	0

Figure 7.13: Function cover after Irredcover

the AND gates in the driver PLA are ordered and then tests are generated for output stuck-at-0 faults. As soon as a redundant cube is detected, it is removed from the cover. The procedure *iterates* over all cubes in the cover until no cubes are redundant. Once again, since the procedure is iterative, it stops only after going through all the cubes and fails to remove any. Since none of the cubes can be deleted, the cover must be irredundant.

To see how the procedure implicitly changes the function chosen, consider the example used in the previous section. The cover of the function after expansion is shown in Figure 7.11 and the corresponding network is shown in Figure 7.12. (For the sake of this example, in the network shown, not every cube in the function is prime). A stuck-at-0 fault at the output of the second AND gate is untestable and therefore the cube can be removed from the cover. This new cover corresponds to the function shown in Figure 7.13 and is different from the function shown in Figure 7.11. Once again, the function is implicitly changed and the new function is minimized simultaneously.

The number of cubes removed depends on the order in which they are considered. A greedy ordering strategy is employed to reduce the number of literals in the final function. Cubes that have a larger number of literals are considered before cubes with a smaller number of literals.

Like the **Expand** procedure, this procedure is also quite different from the corresponding ESPRESSO procedure. Once again, the main difference is that cubes are considered serially here. In ESPRESSO, sets of cubes can be considered for removal using an auxiliary function and appropriate covering procedures. Again, since partially-redundant

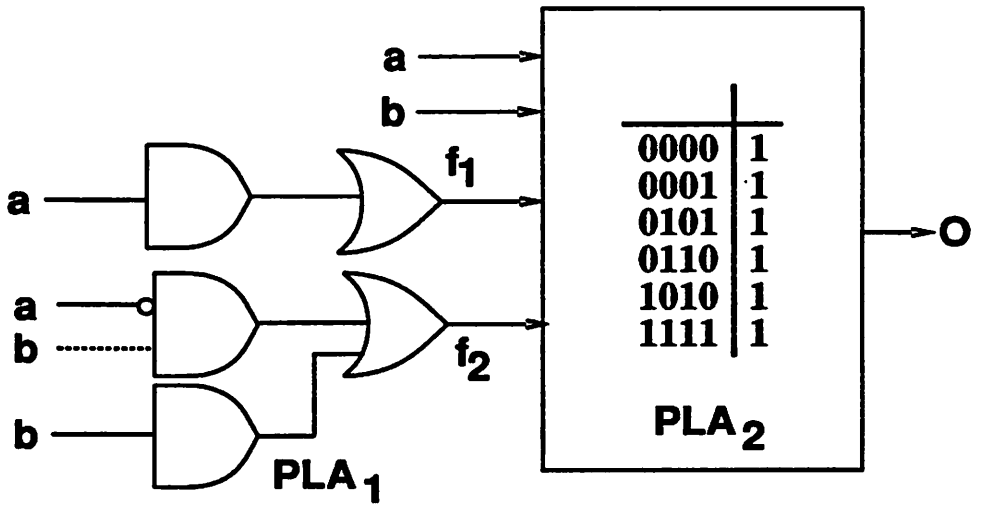


Figure 7.14: Network after Expand-Irredcover

primes as defined in [16] cannot be defined for a test-pattern-generation-based algorithm, the ESPRESSO-type procedure cannot be used.

7.7 Reduce

After **Expand** and **Irredcover**, the cover is prime and irredundant. No more literals can be raised, nor can any more cubes be deleted from the cover. This is a locally optimal solution but may be a poor optimal solution. To move from a local optimum to a better local optimum, it is necessary to start with a different function of (possibly) higher cost. Reduction is the operation that transforms a prime cover F into a new (in general, non-prime) cover F', by replacing each cube by a (smaller) cube contained in it. Like the two previous procedures, this operation implicitly changes the function of the driver PLA without destroying the compatibility with the Boolean relation. However, unlike those procedures, it increases the cost of the resulting function. The reduced function acts as a new starting point for the **Expand** and **Irredcover** operations from where a better local optimum might be reachable. In fact, since some of the cubes of F' are not prime, **Expand**

a	b	f_1	f_2
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	1

Figure 7.15: PLA_1 after Expand and Irredcover

can be applied to F' to yield a different prime cover that may have a fewer number of cubes than F.

In order to convert a prime cover to a non-prime cover, cubes in the cover have to be made non-prime. A non-prime cube has redundant literals and this suggests the use of test generation techniques for reduction. In this procedure, all inputs not connected to an AND gate under consideration are connected to it one by one. If a stuck-at-1 fault on the connector is undetectable in the interconnected network, it is retained. If not, the connector is removed. Thus literals are added to each cube, one literal at a time. Also, for each AND gate connected to an OR gate, if a stuck-at-0 fault on the connector is undetectable in the interconnected network, it is removed. The procedure continues as long as redundant literals can be added to the circuit or fanout stems removed. Note that an ESPRESSO-style Reduce can only reduce the cubes while keeping the function of the driver PLA the same. In this procedure, each cube could be reduced further and a different function for the driver PLA may be chosen during reduction. However, unlike the corresponding procedure in ESPRESSO, redundant literals are added one at a time.

The procedure is illustrated with the help of an example. Consider the Boolean relation shown in Figure 7.8 and the interconnected network after expansion and irredundant cover operation shown in Figure 7.14. The function of the driver PLA is shown in Figure 7.15. If input b is connected to the second AND gate (dotted line), a stuck-at-1 fault on the connector is undetectable and the connection is retained. The resultant function for the driver PLA is shown in Figure 7.16 and is different from the one shown in Figure 7.15.

a	b	f_1	f_2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Figure 7.16: PLA_1 after Reduce

The choice of cubes to be reduced and the method of reduction have a crucial effect on solution improvement. The heuristic used here is similar to the one used in ESPRESSO. Cubes in F are processed sequentially, maximally reducing each one without destroying compatibility. A crude heuristic ordering strategy is used before reducing cubes. Cubes are ordered in decreasing order of size. The inputs to be connected to the AND gate are sorted so that the input variable that occurs in most other cubes comes first. This is motivated by the fact that the literal chosen first is already present in some other cubes. Thus that literal can probably be lowered in the cube under consideration.

7.8 Makesparse

The objective of this routine is to make the PLA matrix as sparse as possible. This enhances the ability of the PLA to be folded and improves some of its electrical properties. After the main loop in the minimization program, the cover of the PLA is prime and irredundant but might not be so with respect to each individual output. The first step in this procedure is to lower the outputs. For each cube in the cover, it is necessary to change as many 1's in the output plane to 0's as possible. If it is possible to lower a 1 in the output plane to a 0, it means that the connection between the AND gate representing the cube and the OR gate implementing the output function can be deleted. This suggests a way of implementing the lowering of outputs procedure. For the cube under consideration, the corresponding AND gate has to be identified. Now for each fanout stem of the AND gate, tests have to be generated for the fault where the stem is stuck-at-0. If the fault is undetectable, the connection is redundant and can be removed. This is repeated for all the cubes in the cover.

Having lowered the outputs, the cubes may not be prime any more. The next step in the **Makesparse** procedure is to raise the input literals in the cubes to make them prime. The procedure followed is similar to the one described in Section 7.5. However, instead of raising both the input and the output literals, only the input literals are raised. In this way the input part of the PLA is made as sparse as possible. The procedure ensures that the resultant cover is prime for each output function, as well as irredundant. However, **Makesparse**, like **Expand**, could change the function implicitly and provide another (and perhaps better) starting point for the minimization process.

7.9 Choosing an Initial Function

Choosing an initial function compatible with a Boolean relation is very important from the point of view of the correctness of the procedure. For a minterm-based specification of a Boolean relation, *i.e.*, where every input condition is a minterm, as in Figure 7.3, choosing an initial function can be easily performed by choosing a single output pattern from the equivalence class for each input minterm. The same is not applicable for a cube-based specification, as illustrated with the example of Figure 7.17. If the same criterion is used, then for the first and the third cubes the outputs 10 and 01 might be chosen. For the minterm 1011 that is common to the cubes, the output asserted is 11 which is not in the equivalence class of the minterm.

The procedure for choosing an initial function for a cube-based specification is described here. At first, a matrix is formed with the patterns in the equivalence class of each cube. In this matrix, there is a column for each output variable. For each cube in the specification, there are as many rows as there are patterns in its equivalence class. Each row therefore corresponds to an output pattern (for a cube) and the entry in any column for that row is the value of the output variable in that pattern. Having constructed the matrix, the number of one's and zero's in each column are counted, and the column with the maximum number of one's or zero's (depending on the count) is chosen and the

a	b	c	d	$\{f_1f_2\}$
–	0	1	–	{10, 01}
0	–	–	1	{00, 11}
1	0	–	1	{01, 10}
0	1	0	–	{11, 00}

Figure 7.17: Cube-based specification of a Boolean relation

corresponding output variable is assigned the value 1 or 0. If the value 1 is assigned to the variable, then all rows that have a 0 in that column are marked and are not considered in the same pass. This process is repeated until all output variables are assigned a value, *i.e.*, until a particular output pattern is chosen. The rows remaining correspond to cubes that have the chosen output pattern in their equivalence class. These cubes are assigned the output pattern, marked as selected, and removed from the specification and from the matrix (for each selected cube, multiple rows may be removed from the matrix as each cube has multiple row entries). All cubes in the specification that intersect the selected cubes but have not been selected are saved in a set L and their corresponding rows are removed from the matrix. Now all the marked cubes in the matrix are unmarked and the process of selection of output patterns continues until the matrix becomes empty. For the selected cubes (with their associated output patterns), it can be guaranteed that the output for each minterm is in the equivalence class of that minterm.

The complement of the set of selected cubes is now found using the logic minimizer ESPRESSO. The complement is intersected with the set of cubes saved in the set L. This produces cubes containing minterms in the original specification for which no output pattern has been selected yet. These cubes can be treated as a new Boolean relation and the procedure of selection using the matrix can be applied. These steps are repeated until L becomes empty or the intersection of L with the complement of the selected set becomes empty. The resulting selected set is a function that is compatible with the Boolean relation. At each step of the process, more minterms are added to the selected set and therefore this procedure must converge when all minterms for which equivalence classes have been specified are selected.

The correctness of the minimization procedure can be proved as follows.

Theorem 7.9.1 : *Given a Boolean relation, the minimization procedure described here produces a function compatible with the Boolean relation.*

Proof: The procedure starts with a function that is compatible with the Boolean relation. In the procedure wires are either added or deleted. The addition or deletion is based on the redundancy of a fault in the interconnected network. Assume that during the procedure, at a certain step when a wire is either added or deleted, the function becomes incompatible with the Boolean relation. Therefore, the fault considered in the operation must have been redundant. Since the new function is not compatible with the Boolean relation, there must exist a minterm for which the outputs of the driver PLA before the operation and after the operation are not in the same equivalence class. Since the driven PLA maps patterns only in the same equivalence class to a single pattern at its output, the output of the driven PLA before the operation would be different from that after the operation. Therefore that minterm constitutes a test for the fault considered in the operation and therefore the fault is not redundant. This is a contradiction that disproves the assumption. ■

7.10 Experimental Results

The algorithm described in this chapter has been implemented in a system called HERB. As mentioned previously, the test generation algorithm used is a variation of the PODEM [65] test generation algorithm with modifications suggested in [60] and [119].

In this section, fifteen examples are presented, some of which have been taken from [17] and the rest from various industrial and university sources. For each example in Table 7.1, the number of inputs and the number of outputs and the final number of product terms and literals in the sum-of-product representations after minimization is presented. The final column indicates the total minimization time on a DECstation 3100.

In Table 7.2, the results are compared with the results obtained from an *exact minimizer for Boolean relations* described in [17]. Though in some cases the number of

CKT	#Inputs	#Outputs	#Product terms	#Literals	CPU Time
int1	4	3	8	24	0.2s
int8	4	3	8	37	0.5s
c17a	5	3	5	15	0.6s
c17b	5	3	7	25	0.4s
int10	6	4	30	157	5.0s
she1	7	3	9	58	120.4s
she2	5	5	13	74	30.0s
she3	7	4	10	41	800.2s
she4	5	6	27	154	44s
int7	6	4	16	68	4.5s
yosi	5	13	14	100	24.2s
int15	24	14	145	1573	421.8s
b9	16	5	452	5052	892.9s
gr	15	11	126	1240	400.4s
vtx	27	6	424	6404	1001.6s

Table 7.1: Experimental results

CKT	EXACT			HERB		
	#Terms	#Lits	Time	#Terms	#Lits	Time
int1	5	13	31.7s	8	24	0.2s
int8	8	35	0.1s	8	37	0.5s
c17a	5	11	0.9s	5	15	0.6s
c17b	7	19	26.2s	7	25	0.4s
int10	25	113	40044.1s	30	157	5.0s
she1	6	34	9.0s	9	58	120.4s
she2	11	43	68.5s	13	74	30.0s
she3	8	31	71.9s	10	41	800.2s
she4	Out of Memory			27	154	44s
int7	Out of Memory			16	68	4.5s
yosi	Out of Memory			14	100	24.2s
int15	Out of Memory			145	1573	421.8s
b9	Out of Memory			452	5052	892.9s
gr	Out of Memory			126	1240	400.4s
vtx	Out of Memory			424	6404	1001.6s

Table 7.2: Comparison with exact minimization

literals obtained by the heuristic method is double the exact minimum, the time required for heuristic minimization is significantly smaller. In all cases considered so far, the number of product terms obtained using the heuristic approach is close to the exact minimizer's

CKT	#Product terms	#Literals	CPU Time
she4	27	154	0.1s
int7	16	68	0.2s
yosi	18	141	0.6s
int15	151	1912	240s
b9	452	5054	14s
gr	130	1401	10.9s
vtx	424	6404	120.9s

Table 7.3: Comparison with exact function minimization

results. For all the large examples, the exact minimization approach ran out of memory after running for several hours. The heuristic approach was successful in minimizing all the relations in reasonable amounts of time. The memory requirements for all examples were very small as this approach is not at all memory intensive. Despite the dramatic reduction in time for some examples, there were some for which the time taken by the heuristic approach was larger. This can be attributed to the fact that the heuristic approach can spend a significant amount of time in trying to search for a good solution in a part of the space where no good solution exists. This might happen because of a poor initial choice.

To compare the quality of results obtained using the heuristic approach for those examples for which the exact algorithm did not complete due to lack of memory, the examples were minimized using an exact logic minimization algorithm (ESPRESSO-EXACT [116]). The functions were derived from the Boolean relations by using **Choose_Function()**. It is time consuming to manually choose a good function compatible with the Boolean relation. Specifying extra don't-care information for the function is even more difficult. These are the drawbacks of using function minimizers for minimizing Boolean relations. The results of function minimization are presented in Table 7.3. As can be easily seen, the quality of results obtained by the heuristic procedure are in most cases better or at least as good as those obtained using exact function minimization. Also, there are examples for which the difference in the quality of the result is not much.

7.11 Conclusions

Minimization of Boolean relations is important from the point of view of logic synthesis and synthesis for testability. A fast and memory efficient heuristic test generation based algorithm for the minimization of Boolean relations was presented. This algorithm uses iterative logic improvement to derive a function of minimal cost that is compatible with the Boolean relation. Using this approach, it is possible to minimize larger circuits than with the exact minimization approach. Moreover, the circuits are testable for all single stuck-at faults. It is also guaranteed that for all faults there exists a test such that the responses of the true and the faulty networks are not in the same equivalence class. The latter property is useful for sequential logic synthesis for testability.

This approach suffers from some drawbacks. First, since redundant literals and cubes are removed sequentially, the ordering of the literals and cubes becomes very critical for obtaining better quality results. Second, only the removal of one literal or cube is considered at a time. Considering the removal of more than one literal or cube at a time (as in ESPRESSO) could give better results. This implies the use of multiple faults instead of single faults during the minimization process. Since the number of multiple faults is very large (exponential in the number of wires in the circuit), the time required for minimization could be very large. Third, in this approach the entire space of functions is not explored and thus given enough time, this approach might not be able to find the exact minimum function compatible with the Boolean relation.

Chapter 8

CONCLUSIONS

The problems of test generation and verification of sequential circuits were addressed in this dissertation. In addition, the relationship between test generation, verification, and sequential logic synthesis was explored.

Sequential test generation is considered a difficult problem. One of the main reasons for this is that a fault typically modifies a few edges in the STG of a machine. Therefore, finding an input sequence that distinguishes between the fault-free and the faulty machines requires an enormous amount of searching. In Chapter 2, a novel approach to test generation for sequential circuits was presented. The algorithm uses selective STG enumeration together with the *fault-free circuit* heuristic. The fault-free circuit heuristic is used to generate justification and differentiation sequences for faults using only the fault-free machine. This heuristic tries to take advantage of the fact that a fault typically modifies a few edges in the STG of a machine. In addition, fault-free justification and differentiation can be performed much more efficiently than the same under faulty conditions, since *information can be reused*. Further, selective enumeration of the STG using the intersection of sum-of-product forms, which forms the basis for justification and differentiation, can be performed efficiently by using sophisticated data structures. Another characteristic of the test generation algorithm proposed is that the problem of test generation is split into three subproblems rather than the traditional two. This results in improved efficiency.

The algorithm described makes intelligent use of covers of the ON and OFF-sets of each PO and NS line for test generation. Cubes are represented as bit vectors and cube intersections, which form the basis of the justification and differentiation algorithms, are performed efficiently using bitwise AND operations. This is both a strength and a weakness of this approach. If circuits have a moderate number of inputs and latches and the cover sizes are not large, the number of intersections performed is within reasonable limits and therefore justification and differentiation takes a small fraction of the total test generation time. However, as the cover sizes grow with circuit size, the times for justification and differentiation grow with it. For large circuits, where complete covers cannot be generated because of memory restrictions and even partial covers are large, justification and differentiation can become the bottleneck in test pattern generation.

Despite these drawbacks, it was shown that this approach is significantly faster than the best existing approach and could handle larger circuits more efficiently. This approach has been used to successfully generate tests for finite state machines with a large number of latches within reasonable amounts of CPU time and close to the maximum fault coverage has been obtained. In addition, a larger class of sequentially redundant faults can be identified. This is because invalid states can be easily detected during the justification process, given the complete covers. Though circuits larger than those handled by the approaches of [1] and [90] can be handled by this approach, the very large circuits are still out of reach.

To alleviate the problem of size, an approach that does not require the storage of covers was presented in Chapter 3. The key ideas of Chapter 2 were used together with the exploitation of the properties of Register-Transfer Level (RTL) descriptions to design a test pattern generator that can handle larger circuits and is more efficient. RTL descriptions are interconnections of well-defined modules that perform a specific information processing task. For many such modules, there is an associated *algebra*. For example, adders can be represented using the + operator, and for a network of adders, a system of simultaneous equations can be written to represent the behavior of the system mathematically. This system of equations can be used to determine uniquely the values on each wire, given the primary inputs to the system. The opposite is also easily done, *i.e.*, given the primary output values, primary input values that justify the output values can be obtained by solving this system of equations. In the test pattern generator, this algebra is exploited during justification and differentiation. Good heuristics play a major role in the efficiency of this algorithm and a set of heuristics was presented. It is possible to generate tests for some chips, of intermediate complexity, for which covers are too large to generate, store, and use effectively in test pattern generation.

As is often the case, the strongest point of an algorithm also happens to be its weakest. For circuits consisting of a large number of arbitrary Boolean function modules which do not have an associated algebra, the test generator has to resort to time consuming backtracking methods. Though indexed backtracking improves the efficiency of the process by avoiding backtracking over unrelated decisions, the number of backtracks could still be large and backtracking could be time consuming. However, it is expected that the test generator STEED will perform well for such circuits. In general, it can be said that for circuits with a regular structure, ELEKTRA will perform better than STEED. Such circuits are pure datapaths or datapaths with small associated controllers. These circuits have STGs that are highly connected, *i.e.*, it is possible to reach almost any state from any other. For circuits like pure controllers that have a sparse STG, STEED will perform better. Therefore, two viable tools that cover more or less the entire spectrum of most commonly designed circuits have been developed.

Despite having more powerful test generators, the problem of testing is not fully solved. If a circuit has a large number of redundant faults, even a good test generator might spend a significant amount of time in identifying these faults. The synthesis process can help in easing the task of test generation by synthesizing the circuit to be fully and easily testable. Though it is difficult to characterize easy testability and make the circuit easily testable, there are procedures to synthesize circuits to be fully testable. One such procedure was presented in Chapter 4.

The structure of the synthesis for testability algorithm presented in Chapter 4 is similar to that of [49]. The main issue in this algorithm is the efficient derivation of the set of don't-cares necessary to make the circuit fully testable. The objective is to derive the minimum set of don't-cares as efficiently as possible. Ideas from Chapters 2 and 3 were used to formulate an algorithm that determined the necessary set of don't-care conditions. An iterative logic optimization and don't-care derivation strategy was used. The testability of the circuit was iteratively improved until the circuit became fully testable. Unlike previous approaches, this is not based on the STG description of the circuit and therefore not restricted to small controller type circuits. In fact, the RTL description can be exploited (as in Chapter 3) to generate the required don't-cares. Therefore, datapath-controller circuits as well as digital signal processors, whose STGs are very large, can be synthesized to be fully testable. Moreover, the procedure does not introduce any area or performance overhead like scan design and can be used for circuits an order of magnitude larger than those handled by previous STG-based synthesis approaches. A complete chip of intermediate complexity and a data encryption chip were synthesized to be fully testable. The memory and CPU time requirements for synthesis are also reasonable.

Despite its advantages and superiority over other approaches for a class of circuits, this procedure is not the remedy to all testing woes. Determination of invalid and equivalent states is by no means a solved problem. In the worst case, the algorithms have exponential complexity. Therefore the dream of designing circuits to be fully testable is not yet a reality. A significant amount of effort is still required.

Connected intimately to the problem of test generation is the problem of verification. Many of the techniques used for test generation can be used for verification and vice versa. In the field of verification, there has been a gradual evolution towards better and more sophisticated algorithms. Starting from purely explicit techniques like exhaustive enumeration, today we have completely implicit techniques for verification. Exhaustive simulation techniques can be applied only to the smallest examples because of its exponential average-case complexity. The first step towards a better algorithm was taken in [46] where the input space was implicitly enumerated. Two different algorithms for verification of sequential machines were presented in Chapter 5. These algorithms represent the next step in the evolutionary chain, *i.e.*, purely implicit enumeration of the input space and partially implicit enumeration of the state space. Two different paradigms for the use of these algorithms were presented. One used the product machine approach and an STG traversal technique, and the other used an enumeration-simulation approach. The main characteristic of these algorithms is that equivalent states with uni-distant codes can be merged into single cube states, thereby requiring the enumeration of a smaller number of states. Also, multiple edges from different states are coalesced into one edge, producing a more compact STG. The other characteristic of these algorithms is that they are based on depth-first search. Therefore they are memory efficient, much more so than the recently developed completely implicit techniques [30, 35]. However, the completely implicit techniques are quite sophisticated and are able to handle a large class of circuits efficiently. The experiments performed indicate that for circuits that have STGs that are narrow and deep, the approaches presented here work better than purely implicit approaches. For almost all other classes of circuits, the implicit approaches produce better results, especially for circuits that have STGs that are regular and wide.

Arising out of the need to efficiently verify circuits, the Implicit State Transition Graph (ISTG) extraction procedure was developed. ISTGs are more compact than conventional STGs, yet they have all the information that a conventional STG has. Also, ISTG extraction often takes a smaller amount of time and memory than conventional STG extraction. In addition, the resultant STG is partially state minimized, so less time is spent in state minimization. This has obvious implications for re-synthesis of circuits described at the logic level.

A strategy for the re-synthesis of circuits was presented in Chapter 6. The ISTGs of circuits described at the logic level were extracted and used in conjunction with existing state encoding and decomposition techniques to obtain better implementations of the circuits. Methods for re-synthesizing only parts of a circuit and changing the encoding of only a subset of the latches for the sake of area, performance, or testability improvement were proposed. It was shown that the size of the sequential circuits for which current sequential logic synthesis strategies are viable can be significantly increased using ISTGs. Parts of circuits were re-synthesized to obtain improvements in both area and performance. These algorithms are particularly suitable for the control portions of logic circuits where most of the room for sequential logic optimization lies.

Finally, the problem of minimization of Boolean relations, which arise in sequential logic synthesis is addressed in Chapter 7. Boolean relations, a relatively new concept, is a generalization of traditional don't-cares. Minimization of Boolean relations is important from the point of view of logic synthesis and synthesis for testability. A fast and memory efficient heuristic test-generation-based algorithm for the minimization of Boolean relations was presented. This algorithm uses the same minimization paradigm as the logic minimizer ESPRESSO [16]. However, the individual algorithms are different. A desirable characteristic of the circuits obtained after minimization is that they they are testable for all single stuck-at faults. It is also guaranteed that for all faults there exists a test such that the responses of the true and the faulty networks are not in the same equivalence class. The latter property is useful for sequential logic synthesis for testability. Using this approach, it is possible to minimize larger circuits than with the exact minimization approach.

For this dissertation, two test generation programs, one verification program, two synthesis programs, and a program for the minimization of Boolean relations were developed. Some of these tools fit in as a part of a larger sequential synthesis system, FLAMES, being developed jointly at University of California at Berkeley and Massachusetts Institute of Technology.

8.1 Future Work

The focus of my research has been to solve some problems as best as I can. However, as in any research, I have probably opened more doors than I have closed.

In the area of test pattern generation, one obvious extension is to use alternate representations (instead of covers) in the same framework of algorithms as in STEED. One representation that could be particularly useful is BDDs. Performing justification and differentiation using BDDs will have its associated problems, which must be solved. The State Transition Graph Traversal techniques of [30, 35] will be useful for this purpose.

The problem of size will still plague the test pattern generator, as it will not be possible to generate BDDs for all circuits. Currently, there is no nice and clean way for generating BDDs for circuits with multipliers. Also, for large circuits, BDDs might be very large. Like partial covers, *partial BDDs* would have to be used. Currently, there is no notion of partial BDDs. Therefore this work would involve defining partial BDDs and operations on them. Another alternative is to replicate variables and use a more general, though non-canonical, BDD.

The main drawback of the test pattern generator ELEKTRA is the lack of a clean *algebra* for all kinds of modules. There is a need to develop either algebra's for such modules or to develop better backtracking strategies. It is also necessary to formalize what is meant by an RTL description and provide a calculus for working with such descriptions. Such a calculus will make the task of test generation and fault simulation easier. It also could have a significant impact on verification at the RT level.

Though ISTGs are more compact and can be better used than conventional STGs, there are still some drawbacks. Firstly, all equivalent states are not detected and some state minimization might be necessary. Moreover, if equivalent states do not have uni-distant state codes, they cannot be combined to obtain a more compact STG. This drawback can be removed by using alternate representations (like BDDs) to represent states. Another important drawback is that the detection of equivalent states is strongly dependent on the heuristics used in setting the input variables during the enumeration process. Lastly, this approach cannot handle circuits with more than 30 latches, as even the ISTGs become too large to manipulate. Therefore, alternate, efficient, extraction algorithms that do not have the above mentioned drawbacks are necessary.

It has been observed that state of the art state assignment programs like KISS, NOVA, and MUSTANG cannot handle really large circuits, *i.e.*, circuits that have a large number of states and edges. At the present moment, the ISTG extraction program can extract ISTGs of circuits which cannot be re-encoded by these state assignment programs. Finite state machine decomposition techniques should help in the future. However, better encoding strategies are necessary. In the future, STG extraction programs would use BDDs as opposed to cubes and covers and states and edges in the STG would be represented as BDDs. All state assignment programs to date use a State Transition Table type representation of the STG. There is a need for developing strategies that would use a BDD based representation of the STG for state encoding.

The approach described for the minimization of Boolean relations suffers from some drawbacks. First, since redundant literals and cubes are removed sequentially, the ordering of the literals and cubes becomes very critical for obtaining better quality results. Second, only the removal of one literal or cube is considered at a time. Considering the removal of more than one literal or cube at a time (as in ESPRESSO) could give better results. This implies the use of multiple faults instead of single faults during the minimization process. Since the number of multiple faults is very large (exponential in the number of wires in the circuit), the time required for minimization could be very large. However, there is scope for further improvement here, as all possible multiple faults do not have to be considered. Therefore, an algorithm could be developed that would intelligently use multiple fault effects. Also, it is necessary to improve the algorithm so that the entire space of solutions is searched, so that if the user desires, the exact solution can be found.

Bibliography

[1] V. D. Agrawal, K-T. Cheng, and P. Agrawal. CONTEST: A Concurrent Test Generator for Sequential Circuits. In *Proceedings of the 25th Design Automation Conference*, pages 84–89, June 1988.

[2] V. D. Agrawal, S. K. Jain, and D. M. Singer. Automation in Design for Testability. In *Proceedings of the Custom Integrated Circuit Conference*, pages 159–163, May 1984.

[3] P. N. Anirudhan and P. R. Menon. Symbolic Test Generation for Hiearchically Modeled Digital Systems. In *Proceedings of the International Test Conference*, pages 461–469, October 1989.

[4] K. Apt and D. Kozen. Limits for Automatic Verification of Finite State Concurrent Systems. In *Information Processing Letters*, pages 307–309, 1986.

[5] D. B. Armstrong. A Programmed Algorithm for Assigning Internal Codes to Sequential Machines. In *IRE Transactions on Electron Computers*, volume EC-11, pages 466–472, August 1962.

[6] D. B. Armstrong. A Deductive Method for Simulating Faults in a Circuit. In *IEEE Transactions on Computers*, volume C-21, pages 464–471, May 1972.

[7] P. Ashar. *Sequential Logic Optimization.* PhD thesis, University of California, Berkeley, December, 1991.

[8] P. Ashar, S. Devadas, and A. R. Newton. A Unified Approach to the Decomposition and Re-decomposition of Sequential Machines. In *Proceedings of the 27th Design Automation Conference*, pages 601–606, June 1990.

[9] P. Ashar, S. Devadas, and A. R. Newton. Testability-Driven Synthesis of Interacting FSMs. In *Proceedings of the International Conference on Computer Design*, pages 273–276, September 1990.

[10] P. Ashar, S. Devadas, and A. R. Newton. Optimum and Heuristic Algorithms for an Approach to Finite State Machine Decomposition. In *IEEE Transactions on Computer-Aided Design*, pages 296–310, March 1991.

[11] P. Ashar, A. Ghosh, S. Devadas, and A. R. Newton. Implicit State Transition Graphs and its Application to Synthesis and Test. In *Proceedings of the International Conference on Computer-Aided Design*, pages 84–87, November 1990.

[12] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek. The ISPS Computer Description Language. Technical report, Dept. of EECS, CMU, Pittsburgh, PA, August 16, 1979.

[13] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer-Aided Design*, pages 723–740, June 1988.

[14] D. Brand. Redundancy and Don't Cares in Logic Synthesis. In *IEEE Transactions on Computers*, volume C-32, pages 947–952, October 1983.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer-Aided Design*, pages 1062–1081, November 1987.

[16] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[17] R. K. Brayton and F. Somenzi. An Exact Minimizer for Boolean Relations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[18] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *Proceedings of the VLSI-89 Conference*, Munich, August 1989.

[19] R. K. Brayton and F. Somenzi. Minimization of Boolean Relations. In *Proceedings of the International Symposium on Circuits and Systems*, Portland, Oregon, May 1989.

[20] M. A. Breuer. A Random and an Algorithmic Technique for Fault Detection and Test Generation. In *IEEE Transactions on Computers*, volume C-20, pages 1366–1370, November 1971.

[21] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[22] M. A. Breuer and A. D. Friedman. Functional Level Primitives in Test Generation. In *IEEE Transactions on Computers*, pages 223–235, March 1980.

[23] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings of the International Symposium on Circuits and Systems*, Portland, Oregon, May 1989.

[24] M. C. Browne and E. M. Clarke. A High Level Language for the Design and Verification of Finite State Machines. In *IFIP WG 10.2 Int. Work. Conf. From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 269–292, 1986. Grenoble, France.

[25] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. In *IEEE Transactions on Computers*, volume C-35, pages 1035–1044, December 1986.

[26] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[27] R. E. Bryant. Symbolic Verification of MOS Circuits. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 419–438, December 1985.

[28] R. E. Bryant. Algorithmic Aspects of Symbolic Switch Network Analysis. In *IEEE Transactions on Computer-Aided Design*, pages 618–633, July 1987.

[29] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS : A Compiled Simulator for MOS Circuits. In *Proceedings of the 24th Design Automation Conference*, pages 9–16, June 1987.

[30] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th Design Automation Conference*, pages 46–51, June 1990.

[31] J. Burns, A. Casotto, M. Igusa, F. Marron, F. Romeo, A. Sangiovanni-Vincentelli, C. Sechen, H. Shin, G. Srinath, and H. Yaghutiel. MOSAICO: An integrated Macrocell Layout System. In *Proceedings of the VLSI-87 Conference*, Vancouver, Canada, August 1987.

[32] J. D. Calhoun and F. Brglez. A Framework and Method for Hierarchical Test Generation. In *Proceedings of the International Test Conference*, pages 480–490, October 1989.

[33] C.B.Shung, R.Jain, K. Rimey, R.W.Brodersen, E.Wang, M.B.Srivastava, B.Richards, E. Lettang, L. Thon, S.K.Azim, P.N.Hilfinger, and J.Rabaey. An Integrated CAD System for Algorithmic-Specific IC Design. In *IEEE Transactions on Computer-Aided Design*, volume 10, pages 447–463, April 1991.

[34] K-T. Cheng. On Removing Redundancy in Sequential Circuits. In *Proceedings of the 28th Design Automation Conference*, pages 164–169, June 1991.

[35] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[36] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Machines*, 1989. Grenoble, France.

[37] O. Coudert and J.C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[38] R. Dandapani and S. M. Reddy. On the Design of Logic Networks with Redundancy and Testability Considerations. In *IEEE Transactions on Computers*, volume C23, pages 1139–1149, November 1974.

[39] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology Mapping in MIS. In *Proceedings of the International Conference on Computer-Aided Design*, pages 116–119, November 1987.

[40] S. Devadas. Approaches to Multi-level Sequential Logic Synthesis. In *Proceedings of the 26th Design Automation Conference*, pages 270–276, June 1989.

[41] S. Devadas. *Techniques for Optimization-Based Synthesis of Digital Systems.* PhD thesis, University of California, Berkeley, August, 1988. UCB ERL Memo No. M88/54.

[42] S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, November 1990.

[43] S. Devadas and K. Keutzer. Necessary and Sufficient Conditions for Robust Delay-Fault Testability of Logic Circuits. In *Proceedings of the Sixth MIT Conference on Advanced Research on VLSI*, pages 221–238, April 1990.

[44] S. Devadas and K. Keutzer. A Unified Approach to the Synthesis of Fully Testable Sequential Machines. In *IEEE Transactions on Computer-Aided Design*, volume 10, pages 39–51, January 1991.

[45] S. Devadas and H-K. T. Ma. Easily Testable PLA-based Finite State Machines. In *IEEE Transactions on Computer-Aided Design*, pages 604–611, June 1990.

[46] S. Devadas, H-K. T. Ma, and A. R. Newton. On the Verification of Sequential Machines at Differing Levels of Abstraction. In *IEEE Transactions on Computer-Aided Design*, pages 713–722, June 1988.

[47] 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. In *IEEE Transactions on Computer-Aided Design*, pages 1290–1300, December 1988.

[48] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines. In *IEEE Transactions on Computer-Aided Design*, pages 1100–1107, October 1989.

[49] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant Sequential Machines Via Optimal Logic Synthesis. In *IEEE Transactions on Computer Aided Design*, pages 8–18, January 1990.

[50] S. Devadas, H-K. Tony Ma, and A. R. Newton. Redundancies and Don't Cares in Sequential Logic Synthesis. In *Journal of Electronic Testing: Theory and Applications*, pages 15–30, January 1990.

[51] S. Devadas and A. R. Newton. GENIE : A Generalized Array Optimizer for VLSI Synthesis. In *Proceedings of the 23rd Design Automation Conference*, pages 631–637, June 1986.

[52] S. Devadas and A. R. Newton. Decomposition and Factorization of Sequential Finite State Machines. In *IEEE Transactions on Computer-Aided Design*, pages 1206–1217, November 1989.

[53] S. Devadas and A. R. Newton. Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization. In *IEEE Transactions on Computer-Aided Design*, pages 13–27, January 1991.

[54] T. A. Dolotta and E. J. McCluskey. The Coding of Internal States of Sequential Machines. In *IEEE Transactions on Electronic Computers*, volume EC-13, pages 549–562, October 1964.

[55] A. Domic, S. Levitin, N. Phillips, C. Thai, T. Shiple, D. Bhavsar, and C. Bissell. CLEO : A CMOS Layout Generator. In *Proceedings of the International Conference on Computer-Aided Design*, pages 340–343, November 1989.

[56] E. B. Eichelberger and T. W. Williams. A Logic Design Structure for LSI Testability. In *Proceedings of the 14th Design Automation Conference*, pages 462–468, June 1977.

[57] D. Bostick et. al. The Boulder Optimal Logic Design System. In *Proceedings of the International Conference on Computer-Aided Design*, pages 62–65, November 1987.

[58] G. Dahlquist et. al. *Numerical Methods.* Prentice Hall, 1974.

[59] T. E. Fuhrman and D. E. Thomas. Verification of High Level Synthesis Designs through Gate Level Simulation of Compiled Module Implementations. In *Proceedings of the Workshop on High Level Synthesis*, October 1989. Kennebunkport, Maine.

[60] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. In *IEEE Transactions on Computers*, pages 1137–1144, December 1983.

[61] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[62] A. Ghosh, S. Devadas, and A. R. Newton. Verification of Interacting Sequential Circuits. In *Proceedings of the 27th Design Automation Conference*, pages 213–219, June 1990.

[63] A. Ghosh, S. Devadas, and A. R. Newton. Test Generation and Verification for Highly Sequential Circuits. In *IEEE Transactions on Computer-Aided Design*, pages 652–667, May 1991.

[64] S. Ginsburg. A Synthesis Technique for Minimal State Sequential Machines. *IRE Transactions on Electronic Computers*, pages 13–24, March 1959.

[65] P. Goel. An Implicit Enumeration Algorithm to generate tests for combinational logic circuits. In *IEEE Transactions on Computers*, volume C30, pages 215–222, March 1981.

[66] G. Hachtel H. Cho, S-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG Aspects of FSM Verification. In *Proceedings of the International Conference on Computer-Aided Design*, pages 134–137, November 1990.

[67] G. D. Hachtel and R. M. Jacoby. Verification Algorithms for VLSI Synthesis. In *IEEE Transactions on Computer-Aided Design*, pages 616–640, May 1988.

[68] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On Properties of Algebraic Transformations and the Multifault Testability of Multilevel Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 422–425, November 1989.

[69] L. J. Hafer and A. Parker. Register-Transfer Level Digital Design Automation : The Allocation Process. In *Proceedings of the 15th Design Automation Conference*, pages 213–219, June 1978.

[70] Z. Har'El and R.P. Kurshan. Software for Analysis of Coordination. *Proceedings of the International Conference on System Science and Engineering*, pages 382–385, 1988.

[71] J. Hartmanis. Symbolic Analysis of a Decomposition of Information Processing. In *Inform. Control*, volume 3, pages 154–178, June 1960.

[72] J. Hartmanis and R. E. Stearns. Some Dangers in the State Reduction of Sequential Machines. In *Information and Control*, volume 5, pages 252–260, September 1962.

[73] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[74] F. C. Hennie. *Finite State Models for Logical Machines*. Wiley, New York, 1968.

[75] F. C. Hennie. Fault Detecting Experiments for Sequential Circuits. In *Proceedings of the 5th Annual Symposium on Switching Theory and Logical Design*, pages 95–110, November 1974.

[76] F. J. Hill and B. Huey. A Design Language Based Approach to Test Sequence Generation. In *Computer Journal*, pages 28–33. IEEE Computer Society, June 1977.

[77] D. A. Huffman. *The Synthesis of Sequential Switching Circuits.* Journal of the Franklin Institute, 1956. pages 161-190.

[78] D. A. Huffman. *The Synthesis of Sequential Switching Circuits.* Journal of the Franklin Institute, 1956. pages 275-303.

[79] S. Hwang and A. R. Newton. An Efficient Design Correctness Checker for Finite State Machines. In *Proceedings of the International Conference on Computer-Aided Design*, pages 410–413, November 1987.

[80] C. Y. Hitchcock III and D. E. Thomas. A Method of Automatic Data Path Synthesis. In *Proceedings of the 20th Design Automation Conference*, pages 484–489, June 1983.

[81] M. Kawai, H. Shibano, S. Funatsu, S.Kato, T. Kurobe, K.Ookawa, and T. Sasaki. A High Level Test Pattern Generation Algorithm. In *Proceedings of the International Test Conference*, pages 346–352, October 1983.

[82] K. Keutzer. DAGON: Technology Mapping and Local Optimization. In *Proceedings of the 24th Design Automation Conference*, pages 341–347, June 1987.

[83] Z. Kohavi. *Switching and Finite Automata Theory.* Computer Science Press, 1978.

[84] T. Larabee. Efficient Generation of Test Patterns Using Boolean Difference. In *Proceedings of the International Test Conference*, pages 795–801, August 1989.

[85] T. Larabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability.* PhD thesis, Stanford University, February 1990.

[86] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Proceedings of the Third CalTech Conference on VLSI*, March 1983.

[87] B. Lin, H. Touati, and A. R. Newton. Don't Care Minimization of Multi-Level Sequential Logic Networks. In *Proceedings of the International Conference on Computer-Aided Design*, pages 414–417, November 1990.

[88] C-Y. Lo, H. N. Nham, and A. K. Bose. Algorithms for an Advanced Fault Simulation System in MOTIS. In *IEEE Transactions on Computer-Aided Design*, volume CAD-6, pages 232–240, March 1987.

[89] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. An Incomplete Scan Design Approach to Test Generation for Sequential Circuits. In *Proceedings of the International Test Conference*, pages 730–734, September 1988.

[90] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test Generation for Sequential Circuits. In *IEEE Transactions on Computer-Aided Design*, pages 1081–1093, October 1988.

[91] H-K. T. Ma, S. Devadas, R-S. Wei, and A. Sangiovanni-Vincentelli. Logic Verification Algorithms and Their Parallel Implementation. In *IEEE Transactions on Computer-Aided Design*, pages 181–189, February 1989.

[92] J-C. Madre and J-P. Billon. Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour. In *Proceedings of the 25th Design Automation Conference*, pages 205–210, June 1988.

[93] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimizing Sequential Circuits Using Combinational Techniques. In *IEEE Transactions on Computer-Aided Design*, pages 74–84, January 1991.

[94] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[95] S. Mallela and S. Wu. A Sequential Test Generation System. In *Proceedings of the International Test Conference*, pages 57–61, October 1985.

[96] M. P. Marcus. Deriving Maximum Compatibles Using Boolean Algebra. *IBM Journal*, pages 537–538, November 1964.

[97] R. Marlett. EBT: A Comprehensive Test Generation System for Highly Sequential Circuits. In *Proceedings of the 15th Design Automation Conference*, pages 332–338, June 1978.

[98] G. H. Mealy. A Method of Synthesizing Sequential Circuits. *Bell System Technical Journal*, pages 1045–1079, September 1955.

[99] P. R. Menon and S. G. Chappell. Deductive Fault Simulation With Funtional Blocks. In *IEEE Transactions on Computers*, volume C-27, pages 689–695, August 1978.

[100] G. De Micheli. Symbolic Design of Combinational and Sequential Logic Circuits implemented by Two-Level Macros. In *IEEE Transactions on Computer-Aided Design*, pages 597–616, September 1986.

[101] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State Assignment of Finite State Machines. In *IEEE Transactions on Computer-Aided Design*, pages 269–285, July 1985.

[102] A. Miczo. The Sequential ATPG: A Theoretical Limit. In *Proceedings of the International Test Conference*, pages 143–147, October 1983.

[103] E. F. Moore. *Gedaken-Experiments on Sequential Machines*. Princeton University Press, Princeton, N.J., 1956. pages 129-153.

[104] J. D. Morison, N. E. Peeling, and T. L. Thorp. ELLA: Hardware Description or Specification ? In *Proceedings of the International Conference on Computer-Aided Design*, pages 54–56, November 1984.

[105] P. Muth. A Nine-Valued Circuit Model for Test Generation. In *IEEE Transactions on Computers*, volume C-25, pages 630–636, June 1976.

[106] L. W. Nagel. *SPICE2 : A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California at Berkeley, 1975. Memo Number ERL-M520.

[107] T. M. Niermann, W-T. Cheng, and J. H. Patel. PROOFS : A Fast, Memory Efficient Sequential Circuit Fault Simulator. In *Proceedings of the 27th Design Automation Conference*, pages 535–540, June 1990.

[108] S. Nitta, M. Kawamura, and K. Hirabayashi. Test Generation by Activation and Defect-Drive (TEGAD). In *INTEGRATION Journal*, volume 3, pages 2–12, March 1985.

[109] C-L. Ong, J-T. Li, and C-Y. Lo. GENAC : An Automatic Cell Synthesis Tool. In *Proceedings of the 26th Design Automation Conference*, pages 239–244, June 1989.

[110] J. K. Ousterhout. Crystal : A Timing Verifier for Digital MOS VLSI. In *IEEE Transactions on Computer-Aided Design*, pages 336–349, July 1985.

[111] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU Design Automation System. In *Proceedings of the 16th Design Automation Conference*, pages 73–79, June 1979.

[112] M. C. Paul and S. H. Unger. Minimizing the Number of States in Incompletely Specified Sequential Circuits. In *IRE Transactions on Electronic Computers*, volume EC-8, pages 356–357, September 1959.

[113] IEEE Press. *IEEE Standard VHDL Language Reference Manual.* The IEEE, Inc., New York, 1987.

[114] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro. A New Symbolic Channel Router: YACR2. In *IEEE Transactions on Computer-Aided Design*, pages 208–219, July 1985.

[115] J. P. Roth. Diagnosis of Automata Failures: A Calculus and a Method. In *IBM journal of Research and Development*, volume 10, pages 278–291, July 1966.

[116] R. Rudell and A. Sangiovanni-Vincentelli. Exact Minimization of Mutiple-Valued Functions for PLA Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 352–355, November 1986.

[117] T. M. Sarfert, R. Markgraf, E. Trischler, and M. H. Schulz. Hierarchical Test Pattern Generation Based on High-Level Primitives. In *Proceedings of the International Test Conference*, pages 470–479, October 1989.

[118] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter. The Weighted Random Test-Pattern Generator. In *IEEE Transactions on. Computers*, volume C-24, pages 695–700, July 1975.

[119] M. Schulz, E. Trischler, and T. Sarfert. SOCRATES : A Highly Efficient Automatic Test Pattern Generation System. In *IEEE Transactions on Computer-Aided Design*, pages 126–137, January 1988.

[120] M. D. Schuster and R. E. Bryant. Concurrent Simulation of MOS Digital Circuits. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, pages 129–138, January 1984.

[121] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and Routing Package. In *Proceedings of the 1984 Custom Integrated Circuit Conference*, pages 522–527, Rochester, NY, May 1984.

[122] R. Segal. BDSYN : Logic Description Translator; BDSIM : Switch-Level Simulator. Master's thesis, University of California, Berkeley, May, 1987. UCB ERL Memo No. M87/33.

[123] S. Shteingart, A. W. Nagle, and J. Grason. RTG: Automatic Register Level Test Generator. In *Proceedings of the 22nd Design Automation Conference*, pages 803–807, June 1985.

[124] R. E. Stearns and J. Hartmanis. On the State Assignment Problem for Sequential Machines II. *IRE Transactions on Electronic Computers*, pages 593–604, December 1961.

[125] A. Stolzle. A VLSI Wordprocessing Subsystem for a Real Time Large Vocabulary Continuous Speech Recognition System. In *U. C. Berkeley, ERL Memo M89/133*, January 1990.

[126] K.J. Supowit and S. J. Friedman. A New Method for Verifying Sequential Circuits. In *Proceedings of the 23rd Design Automation Conference*, pages 200–207, June 1986.

[127] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni Vincentelli. Implicit State Enumeration of Finite State Machines Using BDDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[128] C-J. Tseng and D. P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. In *IEEE Transactions on Computer-Aided Design*, pages 379–395, July 1986.

[129] U. S. Department of Commerce, National Bureau of Standards. *Data Encryption Standard*, January 1977. Federal Information Processing Standards Publication (FIPS PUB 46).

[130] E. Ulrich and T. Baker. Concurrent Simulation of Nearly Identical Digital Networks. In *Computer*, volume 7, pages 39–44, April 1974.

[131] T. Villa. Constrained Encoding in Hypercubes: Applications to State Assignment. In *U. C. Berkeley, ERL Memo 86/44*, May 1986.

[132] R. A. Walker and D. E. Thomas. Behavioral Transformation for Algorithmic Level IC Design. *IEEE Transactions on Computer-Aided Design*, 8(10):1115–1128, October 1989.

[133] R-S. Wei and A. Sangiovanni-Vincentelli. A Logic Verification System for Combinational Circuits. In *Proceedings of the International Test Conference*, 1987.

[134] G. Whitcomb and A. R. Newton. Abstract Data Types and High-Level Synthesis. In *Proceedings of the 27th Design Automation Conference*, pages 680–685, June 1990.

[135] M. J. Y. Williams and J. B. Angell. Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic. In *IEEE Transactions on Computers*, volume C22, pages 46–60, January 1973.