THE UNIVERSITY OF CALGARY

# A New Approach For Modular Test Generation

by

Abdel–Fattah Yousif

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

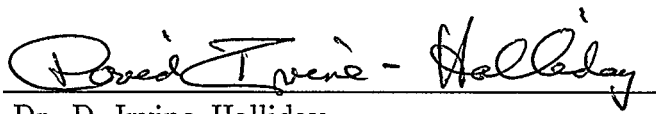
AUGUST, 1995

THE UNIVERSITY OF CALGARY

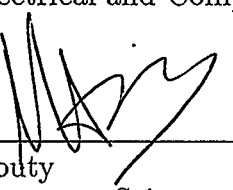
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "A New Approach For Modular Test Generation" submitted by Abdel–Fattah Yousif in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

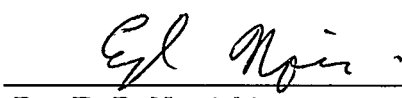
Supervisor, Dr. Jun Gu
Dept. of Electrical and Computer Engineering



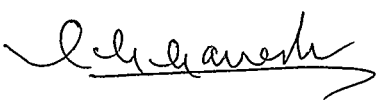
Dr. D. Irvine–Halliday
Dept. of Electrical and Computer Engineering



Dr. N. Bshouty
Dept. of Computer Science



Dr. E. P. Nowicki
Dept. of Electrical and Computer Engineering



Dr. G. Gopalakrishnan, External Reader
Dept. of Computer Science, University of Utah

Date: August 15, 1995

# ABSTRACT

Advances in VLSI technology have now made it possible to integrate increasing number of devices on a single chip. The reliability of a chip is of foremost importance to a VLSI design engineer. Due to the limitations on the number of pins and the increasing circuit complexity, it is very difficult to test a chip within a reasonable cost.

Most design systems support hierarchical design methods in order to contain and reduce the design complexity. Although testing is considered one of the most complex problems in VLSI design, it has not been yet incorporated in the hierarchical design cycle of VLSI circuits. This thesis presents a novel approach for modular test generation of VLSI circuits. The test activities in our modular test generation system are hierarchical. This implies that the modular test system presented in this thesis can be integrated in most of the existing CAD design tools. The benefits of this integration are enormous, such as reduced design cycles for improved testability, better control over the test quality, and better test strategy planning for the product at a higher level of abstraction.

In order to accomplish this task, we present a new and powerful gate–level framework for test generation. This framework is referred to as the *Global Automatic Test Pattern Generation* (GATPG). Experimental results on our GATPG system shows that it is fast, efficient, and guarantees a high test quality. We have extended the domain of the GATPG system by generating tests that implicitly cover multiple faults in a circuit. Finally, we present test assembly procedures which hierarchically gener-

ate chip tests from module's tests. These procedures implement a novel framework which guarantees the hierarchical control of test activities in circuit design. Unlike current approaches, tests are generated incrementally at different levels in the circuit hierarchy. In order to show the benefits of modular test generation over gate–level test generation, a cost model for our test system is presented. The cost model shows that the speedup factor of the modular test generation system outperforms other existing systems.

The impact of the modular test generation framework on the design cycle are discussed at the end of this thesis. We also propose a framework for test strategy selection at the chip level aiming at reducing the test application time and minimizing the hardware that might be needed to improve the chip testability.

## Acknowledgement

This work would not have been possible without the help of many other people. I am extremely grateful to my supervisor Jun Gu, who has been an invaluable source of guidance and friendship throughout my graduate research work. Jun helped me in improving my research quality and in developing a writing style for effectively communicating my research results. Jun was a constant inspiration and confidence builder for everything I did throughout my research work. I cannot thank him enough for his efforts in supporting my work right from the start.

I am indebted to Prof. Graham Birtwistle for his constant encouragement and technical advice on many occasions. I would like to thank Prof. Jim Hasslett for his support as a member of my supervisory committee. I also thank Prof. Nader Bshouty, Prof. Ed Novicki, Prof. Irvine Halliday and Prof. Ganesh Gopalkrishnan for serving on my thesis examination committee.

I am also grateful to my colleagues R. Puri, H. Kenawi, A. Handa, L. Ying and Bin Du whom I had lengthy discussions which helped me develop new directions in my work. A crucial step in the evolution of this work came during the review of a report with H. Kenawi who pointed out to the modular aspects of the back propagation algorithm. R. Puri gave me valuable comments on the completeness of my test generation algorithm. I owe my education to my parents and my wife who gave me the strength to overcome the difficulties encountered during the long road to this Ph.D.

Finally, I would like to acknowledge the financial support provided by the Electrical

and Computer Engineering Department at the University of Calgary and the NSERC Strategic Grant MEF0045793 and the NSERC Research Grant OGP0046423.

To

my family

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The advances in VLSI technology during the last decade have had a great impact on testing. Due to the increase in circuit size and the limited accessibility to the internal nodes of a circuit, the costs of testing a chip have become a substantial part of the overall chip costs. The testing cost is justifiable because it is much less than the cost of having the chip fails in the field.

## 1.1 Overview of VLSI Testing

Test techniques are introduced into the process of VLSI design in order to discover defects in digital systems. Test activities are interwoven with the VLSI design. Architectural design consists of partitioning a VLSI chip into realizable functional blocks. The logic design of these blocks should be synthesized in a testable form or the synthesized logic should be analyzed and improved for testability.

Faulty VLSI chips could be produced during manufacture because of photolithography errors, deficiencies in process quality, or improper design. Even if the chip is manufactured perfectly, it could subsequently wear-out in the field due to electromigration, hot-electron injection, or other reasons. Environmental effects, such as alpha particles and cosmic radiation can also cause a circuit to produce erroneous data.

Testing is experienced at various stages in the production of a system: the dies are tested during fabrication, the packaged chips before insertion in the boards, the boards

after assembly, and the entire system when complete. As far as the level of VLSI chip testing is concerned, a test generation algorithm is used to provide the necessary test vectors which, if applied to the chip, will expose most of the faults occurring at this level of manufacturing. The test cost at this level is primarily determined by the cost of generating these test vectors. Consequently, a new discipline has emerged to probe the testability problem of a circuit more thoroughly in order to give the designer feedback without taking the risk of submitting a circuit design which is not testable. Indeed, design for testability has been very well recognized and served by many researchers and integrated into commercial design methodologies.

When considering which test patterns to generate for testing a complex circuit, one should first consider how good the patterns are for detecting the possible physical failures in the circuit. It may be impossible to consider all possible physical failures. Hence, test patterns are generated to detect some set of modeled faults in the circuit. For example, any line in the gate-level representation of the circuit permanently stuck at logic 0 or 1. The measure of test quality in this case could be the percentage of the stuck-at faults detected by the patterns, and is called *fault coverage* for the fault class. A typical goal might be to achieve a fault coverage for single stuck faults of 99% for the chip.

Fault coverage is determined by a fault simulation program. Simulation of all faults in a large circuit with many tens of thousands of gates may take a prohibitive amount of computer time. Statistical sampling procedures for simulating a fraction of the total faults are commonly used for measuring the effectiveness of the test patterns.

## 1.2 Testing Cost and Testability Analysis

For decades, designers have regarded testability as a troublesome activity, necessary to support the manufacturing process. Extra hardware for testability has been considered as an area overhead and test pattern generation effort has been considered as limiting the time for creative design. However, the continuous growth of circuit complexity made testing difficult and time consuming. On the other hand, quality assurance and reliability have gained much in importance. Better quality testing is required, which complicates the test process even further and by several orders of magnitude. Therefore, the cost of testing has become a critical part of the total chip production cost. It can be as high as 70%. Needless to say, testability has become an irrefutably important part of the design trajectory.

Attempts to understand circuit attributes that influence testability have produced the two concepts of *observability* and *controllability.*Observability refers to the ease with which the state of internal signals can be determined at the circuit output leads. Controllability refers to the ease of producing a specific internal signal value by applying signals to the circuit input leads. Many of the Design For Testability (DFT) techniques are attempts to increase the observability or controllability of a circuit design. A straight forward approach to do this is to introduce test points, that is, additional circuit inputs and outputs to be used during testing. There is always a cost associated with adding test points. For circuit boards adding test points is often well justified. On the other hand, for ICs, the cost of test points can be prohibitive because of IC pin limitations.

A straightforward method for determining the testability of a circuit is to use an Automatic Test Pattern Generation (ATPG) program to generate the tests and determines the fault coverage. The running time of the program, the number of test

patterns generated, and the fault coverage then provide a measure of the testability of the circuit. The difficulty with this approach is mainly the large expense involved in running the ATPG program. Also, the ATPG program may not provide sufficient information about how to improve the testability of a circuit with poor testability. To overcome these difficulties, a number of programs have been written to calculate estimates of the testability of a design without actually running an ATPG program such as TEMAS (Testability Measure Program) and SCOAP (Sandia Controllability/Observability Analysis Program) [25].

These Testability Measure (TM) programs implement algorithms that attempt to predict for a specific circuit the cost (running time) of generating test patterns. In the process of calculating the testability measure, information is developed identifying those portions of the circuit which are difficult to test. This information can be used as a guide to circuit modifications that improve testability.

No accurate relationship between circuit characteristics and testability has yet been demonstrated. Thus the circuit parameters calculated by the TM programs are heuristic and have been chosen on the basis of experience and study of existing ATPG programs. It is not surprising that the various authors of TM programs have chosen different circuit characteristics for their estimates of testability. The technique used to demonstrate that a given TM program does indeed give an indication of circuit testability is to run both the TM program and also an ATPG program on a number of different circuits. A monotonic relation between the TM and the ATPG run time is offered as a proof that the TM program produces a good estimate of circuit testability. The difficulty with this validation technique is the high cost of running enough examples to be reliable. Some interesting results obtained by using statistical methods to evaluate the testability measure program approach are presented in [3].

## 1.3 Faults in VLSI Systems

As systems increase in complexity, it is useful to be able to describe faults at various levels of abstraction in the system. A fault which is described at a very low level, for example the level of transistors, may very accurately describe the physical phenomena causing the fault but, because of the extremely large number of transistors in a VLSI chip, the model may be intractable for the purpose of deriving tests for the fault. The two requirements for fault models are accuracy and tractability. Accuracy means realistic faults should be modeled, while tractability implies that very complex systems should be handled. These requirements are in some sense contradictory. Recent research, therefore, deals with deriving realistic models at higher levels which can accurately capture the faults at lower levels.

As an example, consider a contact between two conducting lines in a VLSI circuit. If the contact is faulty, then the fault can be described at this level of abstraction as a break between two lines. It may also turn out that the break is equivalent to the input of a gate being permanently set to logic 0. The fault can then be described at the gate level of abstraction as a stuck-at 0 fault. It would be simpler for the purpose of analysis to consider the fault at the highest possible level of abstraction.

A physical failure can also lead to the output of a module being at a nonlogical value (for example, indeterminate level between logic 0 and 1). Such faults are difficult to describe and detect, but the errors due to these faults may also be detected by error detection techniques.

### 1.3.1 Fault Models

Fault models are descriptions of the effect of a defect or failure in a circuit. As discussed earlier, fault models are driven by the requirement to derive high quality tests for complex circuits. Thus a useful fault model will naturally lead to a test generation procedure for the fault.

#### 1.3.1.1 Transistor-level Fault Models

Defects in present day integrated circuits can be abstracted to shorts and opens in the interconnects and degradation of devices. Fault models at the transistor level, therefore, can characterize physical failures quite accurately. Unfortunately, as the complexity of VLSI increases, the number of potential faults at the device and interconnect level also increase drastically. Nevertheless, it is necessary to study the effects of failures at the transistor level and to develop accurate fault models at this level. Better understanding of the effects of failures can be used to develop accurate fault models at higher levels which can be applied to complex systems. This approach is analogous to that used in the hierarchical design of VLSI systems where complex circuits are built from smaller cells.

Fault models proposed at the transistor level incorporate one or more of the following classes of faults:

- shorts and opens of transistors or interconnections.
- delay effects of failures.
- coupling or crosstalk between nodes of a circuit.
- degradation of elements.

Shorts and opens are included in most fault models while the more accurate and more complex models include delays. Fault models where activity on one node affects the logic values on another node in the circuit are primarily applied to memories. Fault models which incorporate degradations of elements (for example, transistor parameter changes, or changes in the value of a resistor) are usually used in analog circuits.

#### 1.3.1.2 Gate-level Fault Models

Early fault models were developed at the logic gate level. The popularity of this approach can be attributed to several reasons.

- Such models are simple to design and use.
- Many faults in discrete technologies can be represented by faults at the logic gate level.
- Use of such fault models allows many of the powerful results in mathematics relating to Boolean algebra to be applied to deriving tests for complex systems.
- A fault model at the logic gate level can be used to represent faults in many different technologies if, in fact, defects and faults in these technologies can be mapped to gate faults.

One of the earliest and still widely used fault models at the gate level of abstraction is the stuck-at model. In this model, it is assumed that physical defects and faults will result in the lines at the logic gate level of the circuit being permanently stuck-at logic 0 or 1. This model has been the source of a great deal of research. It is still very popular since it has been shown that many defects at the transistor and circuit

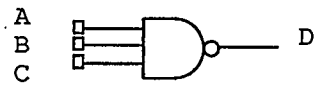



Figure 1.1. A three-input NAND gate example

level can be modeled by the stuck-at fault model at the logic level. In practice, only single stuck faults are considered in a circuit.

A subset of the stuck fault model is the pin fault model, where only input/output pins of a module are assumed to be stuck-at 0 or 1 under failure. This has been used sometimes when testing printed circuit boards with many VLSI devices. Unfortunately, this fault model does not even include a high percentage of gate level stuck faults within the module in most cases and is, therefore, inappropriate for VLSI.

## 1.4 Fault Equivalence and Dominance

Consider the three input NAND gate shown in Figure 1.1. This gate has four lines (three inputs and one output) and would, therefore, have eight stuck-at faults, each line stuck at 0 or 1. However, the faults A, B, or C stuck-at 0 would result in the output D being permanently 1 and, therefore, it is impossible to distinguish between an input stuck at 0 from the output stuck at 1. These faults are said to be *equivalent*. Now consider the fault A-stuck-at-1. In order to detect this fault, a 0 has to be applied on A, and 1s at B and C so that the effect of the fault can be propagated to D. The correct value of D will be a 1 and it will be a 0 under fault. This test for A-stuck-at-1 will, therefore, also detect the fault D-stuck-at-0. Hence, A-stuck-at-1 is said to *dominate* D-stuck-at-0.

Using the relations of equivalence and dominance allows many faults to be com-

Table 1.1. Tests for 3-input NAND gate.

| A | B | C | D | Fault Class |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | A/0,B/0,C/0,D/1 |
| 0 | 1 | 1 | 1 | A/1,D/0 |
| 1 | 0 | 1 | 1 | B/1,D/0 |
| 1 | 1 | 0 | 1 | C/1,D/0 |

bined into a single class, reducing the number of faults to be considered in a complex system. A three-input NAND gate, therefore, will have four different fault classes and the tests for these faults are shown in Table 1.1. In the table, the fault consisting of one line $l$ stuck-at-0 is shown as $l/0$.

The notion of equivalence and dominance can be applied to more complex circuits. Thus two faults which are in different parts of a larger circuit could possibly be equivalent. Figure 1.2 shows a simple circuit with four inputs and one output. Stuck-at-1 faults on the two lines marked $a$ and $b$ are equivalent; that is, the function under either faults is the same. However, equivalences such as these are more difficult to detect and, in practice, only equivalences and dominances around a gate are normally considered. More information on the concepts of the fault equivalence and dominance, as well as the idea of reducing the number of fault classes by fault collapsing, are found in [38, 49].

## 1.5 Scope of the Thesis

The thesis develops and implements a number of test generation frameworks at the gate and modular levels of abstraction. The procedures associated with this implementation aim at integrating the test generation process within the hierarchical framework of designing VLSI circuits. This requires developing an understanding for

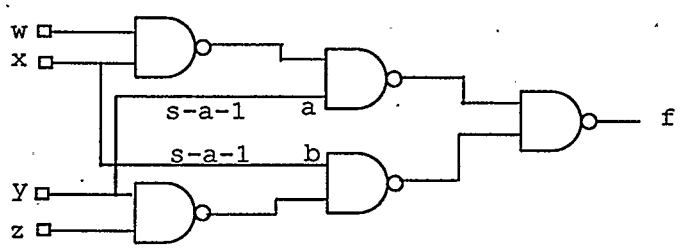



Figure 1.2. Two faults which are functionally equivalent.

the interface between gate level and modular level test generation approaches.

The test problem is well understood in combinational and synchronous sequential circuits. Thus, we limit the discussion in this thesis to combinational and synchronous sequential circuits. Asynchronous circuits are beyond the scope of this thesis. Another restriction is that the thesis is primarily concerned with the stuck–at fault model during test generation. Other fault models can be used at low level of abstraction. The resultant tests can then be used by the modular test system to create chip tests. This approach is not part of the thesis work.

The thesis also deals with single faults (except in Chapter 5). This means that only one fault might exist in the chip during the test application. The purpose of the test generation system is to find test vectors for the modeled single faults in a circuit. Multiple fault existence is dealt with only in Chapter 5.

## 1.6 Motivations and Goals

There is much effort that has been spent on research for powerful Computer Aided Design (CAD) tools to support the design of VLSI and ASICs with a fast turnaround time. Unfortunately, most of these tools regard the test pattern generation problem as

a back–end process, that is looking at the testability issues after the circuit is designed. This is the classical way of solving the testability problem: as an afterthought. Not surprisingly, users have had poor experiences with such tools.

This explains the need for more powerful test tools that support the test development during the design stage. This is especially true for the development of complex ASICs. These chips are tailored towards their application and require dedicated test generation. This is time consuming and therefore costly. Since they are often produced in limited quantities, the relative cost of test program development becomes excessive. Even worse, these test development times place a serious burden on the development of competitive ASICs that require a short time–to–market.

The need for efficient techniques for testing VLSI circuits arises due to the fact that companies are continuously faced with decisions to change and modify their designs. Each chip in the new design must be tested properly in order to eliminate chips with physical failures. A small changes in any of the circuit modules might invalidate the efforts spent on generating tests for the original design. More test efforts will then be needed to level up the test quality of the chip. With the current approaches in test generation, this effort, although automated, is enormous in terms of computer resources and the man–hour involved in it. In order to keep pace of the short design cycle, efficient ATPG tools must be readily available for the test engineer. Accordingly, the decisions made in choosing the test strategy will be highly influenced by these tools.

This thesis aims at providing the test engineer with powerful ATPG algorithms which will provide a range of test strategies at any level of abstraction. Our goal is to build powerful tools for testing VLSI circuits hierarchically. Cost and test quality will always be considered in any of these tools. In short, this thesis describes an

ATPG system for combinational circuits, an implicit maximal multiple fault coverage – single phase ATPG system , and modular (hierarchical) test procedures.

The ATPG system for combinational circuits has two major features which are highly desirable by test engineers. Unlike other ATPG systems, it is single phase which implies that the random test generation phase is not included in the test system. All test patterns generated by the test system are deterministic. The other feature which is unprecedented in other test systems is the global test approach in which tests are generated. Global test generation implies that more than one target fault are considered by the test algorithm, as opposed to the single target fault strategy that is currently adopted by other test algorithms. These two features not only generated quality test vectors with large fault coverage but also enabled us to produce test primitives for modules under test. The term test primitive is used to describe test sets for different modules. The efficiency of the test generation system was enhanced by including a powerful procedure which will allow the generated test sets to have maximal multiple fault coverage. As far as we know, no existing test system was able to provide such test sets.

The final step in this thesis is the use of the above test system to hierarchically generate test patterns for a modular design using the test primitive of each module. In order to achieve this objective, symbolic paths between modules must be created in order to move freely from one module's inputs/outputs to another module's inputs/outputs. Current hierarchical ATPG systems use the transfer (functional) mode of modules to create these paths. In our approach, we did not separate the test set from the symbolic paths of a module. This is one of the most important achievements of this work because it shows that our test system is truly modular and inherently efficient. We start by generating test primitives which serves both as test

patterns and symbolic paths for faults across a module. Then, we describe a new procedure for modular testing using the generated test sets.

If full scan is chosen as the strategy for testing a chip, then, the circuit in its modular form will be dealt with as combinational circuit. Therefore, the combinational test generator will be applied at each module separately. Each test primitive will be attached to a module. A modular test procedure will then be applied in order to assemble the chip tests at the primary inputs.

## 1.7 Contributions of the Thesis

A major contribution of this thesis is a novel and efficient modular test generation methodology that significantly reduces the complexity in testing large VLSI circuits. The most significant elements of this thesis are as follows:

- The identification of the requirements for a truly modular test generation system. The test interface between one level of abstraction and a higher level is clarified. The performance failure of current hierarchical test systems is explained. We have formally characterized the *test primitives* and stated the conditions under which test primitives can provide a complete test and functional description for a module (Chapter 3).

- An efficient global automatic test pattern generation algorithm (GATPG) to generate the test primitives with the required specifications is presented. The algorithm is capable of generating tests with a 100% fault coverage in a very short time compared to other approaches. The GATPG approach provides 1.6 to 47 speed–up factors over current approaches. This performance is achieved because of the application of the novel global search strategy where faults are searched collectively using shared search spaces for faults. Also, an efficient tree

pruning technique is applied to the algorithm in order to limit the memory size during its execution (Chapter 4).

- A direct relationship between the global test generation framework and multiple fault testing is established. A new view point in the analysis of multiple faults behavior is presented. This analysis is later adopted in our GATPG system to generate tests that implicitly cover multiple faults in a circuit (Chapter 5).

- An efficient modular test generation methodology is presented. The test activities in this methodology is hierarchical making it the first known approach with the potential of being integrated into the hierarchical framework for designing VLSI circuits. This methodology requires no extra heuristics for modular/hierarchical test generation. Thus, it can be integrated into VLSI CAD tools with minimal programming efforts (Chapter 6).

- A cost model for hierarchical test generation is presented. The speed up factor using our modular test generation system over low level test systems is shown to be increasing with the increase in the circuit size (Chapter 6).

- A revision of the test strategies at the chip level in the context of the modular test generation system is presented. We propose different strategies to optimize the test quality of the chip. A novel approach for minimizing the hardware addition to the chip design through macro testing is presented. The purpose of this approach is to improve the test quality of chips without loosing large silicon area.

## 1.8 Structure of the Thesis

The thesis is organized as follows:

Chapter 1 is an introduction to this work. In Chapter 2, some background about

the test problem and the related issues are presented. In Chapter 3, the new model for global Automatic Test Pattern Generation (ATPG) system will be presented. In Chapter 4, a combinational test generation algorithm based on the global ATPG model is presented. The GATPG algorithm will be presented with two implementations, namely, two phase and single phase test generation systems. In the two phase implementation, a random test generator is used as a front end during test generation. Random testing will cover most of the easy to detect faults in the circuit. In the single phase implementation, the random phase is not considered in the test generation process. The two implementations will enable us to compare our results adequately with other existing algorithms.

In order to enhance the efficiency of the combinational test generation algorithm, we have modified our algorithm to generate test sets with maximal multiple fault coverage. In order to ensure that the whole test set achieves a maximal multiple fault coverage, all test vectors must be generated deterministically, i.e., the single phase implementation is used in this part of the thesis. The single phase test system with implicit maximal multiple fault coverage is presented in Chapter 5. Also in Chapter 5, the necessary analysis for generating tests with maximal multiple fault coverage is presented.

Chapter 6 discuses the modular test generation procedures that we propose in this thesis. The modular test procedures and the hierarchical test control are presented in this chapter. In Chapter 7, we present the framework for test strategy selection at the chip level, in the context of the modular test generation system. We propose some new techniques to minimize the test application time and control the hardware addition as well. Chapter 8 concludes the thesis.

## 1.9 Summary

In this chapter, motivations that initiated the interest in the testing problem have been introduced. The cost of manufacturing a VLSI chip is shown to be very much affected by the testability figure of the chip. Design for testability, testability analysis programs, and new test generation algorithms are a normal consequence for the test process requirements.

The large number and complex nature of physical failures dictates that a practical approach to testing should avoid working directly with the physical failures. In most cases, in fact, one is not usually concerned with discovering the exact physical failure; what is desired is merely to determine the existence of (or absence of) any physical failure. One approach for solving this problem is to describe the effects of physical failures at some higher levels of abstraction. This description is called a fault model. The stuck–at fault model will be used throughout this thesis to generate test patterns which cover the physical failures in VLSI circuits.

# CHAPTER 2

# BACKGROUND AND PREVIOUS WORK

In this chapter, the test generation problem for combinational circuits is presented. Section 1 presents the test generation terminologies used throughout this thesis. The test problem complexity is identified and formulated in Section 2. The test strategies for current test pattern generation algorithms will be presented in Section 3. Although numerous approaches to test generation have been reported, only a few of these approaches are used in test systems. Section 4 presents some of these approaches such as the *D*-Algorithm, PODEM (Path Oriented DEcision Making), and FAN. The test strategies for these algorithms will be explored. The relationship between combinational test generation and the modular testing approach will then be discussed in Section 5.

## 2.1 Preliminaries and Notations

Common terminology pertaining to test generation for logic circuits is readily introduced with an example. Figure 2.1 shows a combinational logic circuit and a test for a single stuck fault that causes node $h$ to permanently assume a 0 state. A *stuck-at-1* (s/1) fault on a signal node causes that node to permanently assume the 1 state. A *stuck-at-0* (s/0) fault causes a permanent 0 on the faulted node. The five valued logic (0, 1, $X$, $D$, $\overline{D}$) is used to describe the behavior of a circuit with failures. The logic value $D$ designates a logic value 1 for a node in the error free circuit and a 0

for the same node in the failing circuit, $\overline{D}$ is the compliment of $D$, and $X$ designates a DON'T CARE value. A behavior difference between the good circuit and the failing circuit propagates along a *sensitized path.* In Figure 2.1, the signal path h, j (the bold line) is referred to as a sensitized path. Externally controllable nodes are referred to as *primary inputs.* Externally observable nodes are referred to as *primary outputs.* In Figure 2.1 assignment of the values 1, 1, $X$, $X$, 0 to the primary inputs $a$, $b$, $c$, $d$, $e$, respectively, constitutes a test for the fault $h$ s/0.

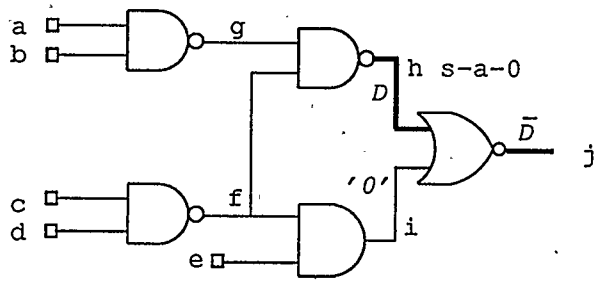



Figure 2.1. Example to illustrate test generation terminology.

*Definition 1 :* Two faults are said to be compatible if there exists at least one test vector which detects both faults.

*Definition 2 :* Two faults are said to be collapsed if the detection of one fault implies the detection of the other fault. The two faults can also be referred to as *indistinguishable* faults.

*Definition 3 :* The $D$-drive refers to the node with a logic value $D$ or $\overline{D}$ and is used by the test generation algorithm to bring it closer to the primary outputs. In Figure 2.1, node $h$ represents a $D$-drive to the test generation process. If at any time in the test generation procedure, more than one node carries the logic values $D$ or $\overline{D}$, then we refer to these nodes as the $D$-frontier. The test generation algorithm picks up one of these nodes to drive the test process, i.e., selecting the $D$-drive node.

*Definition 4 :* The implication procedure refers to the process of using the implication rules of logic gates to propagate signal values at gate input nodes to their output nodes. This procedure is used to check the implication of logic assignments made during the test generation procedure. The result is used as a guide to the next step in the test procedure.

*Definition 5 :* Consistency check is a procedure used by test generation algorithms to check if the previously made decisions meet some objectives set by the algorithm. The decisions made by the test generation algorithm are referred to as *inconsistent* if they don't meet the objectives set by the algorithm. It must be noted that these objectives vary during the test procedure.

## 2.2 The Test Generation Problem

With the progress of VLSI technology, the problem of fault detection for logic circuits is becoming more and more difficult. In developing tests for digital circuits, the faults that will actually occur are unknown. Instead, test sets are developed to detect a specific set of faults.

### 2.2.1 Problem Formulation

As Goel [24] stated in his paper, the test generation problem can be formulated as a search of the n-dimensional 0-1 state space of primary input patterns of an n-input combinational logic circuit. For example, in Figure 2.2, g is an internal node and the objective is to generate a test for the stuck fault g s/0. The logic value at g can be expressed as a Boolean function of the primary inputs $X_1$, $X_2$, ..., $X_n$. Similarly, each primary output ($y_j$, $j = 1, 2, ..., m$) can be expressed as a Boolean function of the state on node g as well as the primary inputs $X_1$, $X_2$, ..., $X_n$.

Let $g = G(X_1, X_2, ..., X_n)$

and $y_j = Y_j(g, X_1, X_2, ... X_n)$

where $1 \leq j \leq m$ and $X_i = 0$ or 1 for $1 \leq i \leq n$.

The problem of test generation for g s/0 can be stated as one of solving the following set of Boolean equations:

$G(X_1, X_2, ..., X_n) = 1$

$Y_j(1, X_1, X_2, ..., X_n) \oplus Y_j(0, X_1, X_2, ..., X_n) = 1$

for at least one j, $i \leq j \leq m$ and $X_i = 0$ or 1 for $1 \leq i \leq n$.

The first equation implies that a s/0 fault is first excited to logic 1 (opposite to the stuck-at level), while the second equation implies that the change of the logic value at the fault location can be observed at the primary outputs. The set of equations for g s/1 are the same as above except that $G$ is set equal to 0.

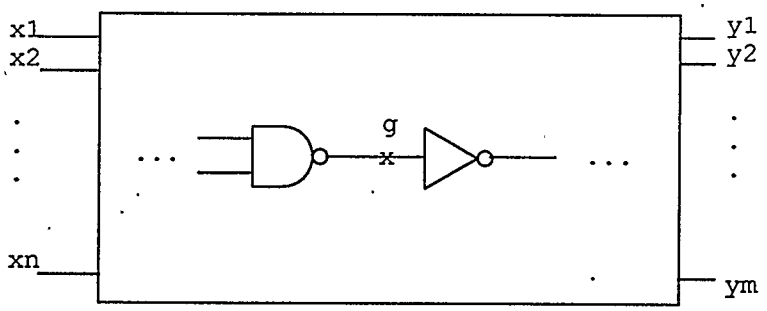



Figure 2.2. A Combinational circuit used in formulating test generation as an n-dimensional 0-1 state space search problem.

In short, test generation can be viewed as a search of an n-dimensional 0-1 space defined by the variables $X_i$ $(1 \leq i \leq n)$ for points that satisfy the above set of equations. More generally, the search will result in finding a k-dimensional subspace $(k \leq n)$ such that all points in the subspace will satisfy the above set of equations.

### 2.2.2 NP-Completeness of Test Generation

The concept of NP-Completeness is used to prove that the amount of time required to solve a specific problem is beyond a certain practical limit [4]. The problem of test generation, which is known to belong to the class of NP-complete problems, can be viewed as a finite space search problem [24]. For a circuit with $N$ primary inputs, there exists $2^N$ combinations of input assignments. Automatic Test Generation (ATG) algorithms basically search for a point in the primary input space that corresponds to a test pattern and consequently, to a solution of the search problem.

The NP-completeness property of the test generation problem necessitates that various heuristics be developed to create practical solutions for it. The PODEM [24] and FAN [21] algorithms are elegant examples in this regard. Many other fault analysis problems, such as the determination of the size of minimal test sets, coverage of multiple faults by single-fault test sets, and coverage of faults by randomly generated test sets are similarly besieged by their inherent complexity, and their solutions require thoughtful insights.

## 2.3 Test generation strategies

The goal of any Automatic Test Pattern Generation (ATPG) system is to to be able to detect the *existence* of faults in a circuit. It might be helpful to be able to pinpoint the exact *nature* and *location* of a fault within a circuit, but this is not necessary for most purposes. A common strategy for ATPG systems has been established through the last two decades. Within the context of this strategy, the test generation task is divided into two phases. In the first phase, random test vectors are generated and simulated to cover as many faults in a circuit as possible. In the second phase, a deterministic algorithm is applied to the rest of the undetected faults in the circuit.

The deterministic algorithm applies its search strategy on a single target fault and the resultant vector (if any) is simulated to cover any other fault that can be detected using the same vector. The single target fault strategy implies that there is at most one fault in a circuit.

### 2.3.1 Path Sensitization

Most ATPG algorithms apply a *path sensitization technique* as the basis for many detailed procedures during the deterministic test generation phase. Sensitization is a technique where a path consisting of many nodes is created to help propagate a stuck-at fault in a circuit. Searching the input space for a test pattern is equivalent to searching for a single (or multiple) sensitizing path.

Consider the circuit of Figure 2.3 and the fault 7 s/0. In order to detect this fault by a procedure that allows access only to the primary input lines (1, 2, 3, 4, 5, and 6) and the primary output line (15), it is essential that a test vector must somehow create a change on line 7 and ensure that the change can be seen on line 15. That is, the test vector must produce a 1 on line 7, and line 15 should be *sensitized* to line 7 in the sense that the output created on line 15 clearly shows whether the signal on line 7 is 0 or 1. If the path from line 7 to line 15 is traced in Figure 2.3, the first condition for sensitization is that line 10 be a 0. Indeed, if line 10 is a 1, then line 13 would be 1 irrespective of the value on line 7. In other words, a 1 on line 10 would desensitize line 7 to line 13. Moreover, since there is no other path to transmit the value on line 7 to line 15, line 10 being a 1 will also desensitize line 7 to line 15. Thus assuming that line 10 is a 0, the next condition for the sensitization is that line 14 be a 1. If both of these conditions exist in the circuit, then when a 0(1) is applied to line 7, the circuit output is going to be a 0(1). In other words, any input vector that

can create a 1 on line 7, a 0 on line 10, and a 1 on line 14 will in the fault free circuit produce a 1 on the output line, and in the faulty circuit a 0 on the output line, and will, therefore, be a test vector for 7 s/0.

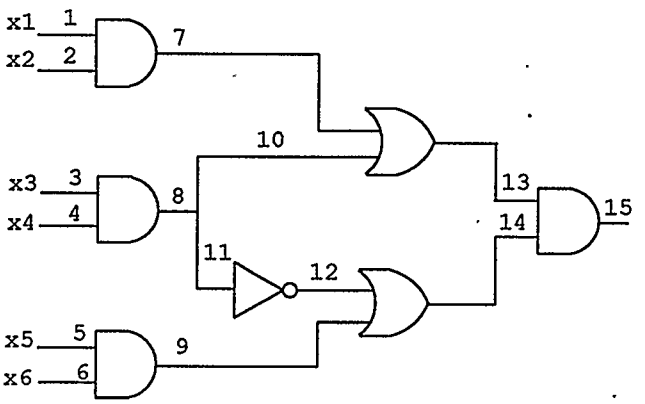



Figure 2.3. A simple circuit to describe sensitization.

The concept of sensitization needs to be explained further in the situations involving more than one path from the faulty line to a primary output line, and in the case of multiple stuck-at faults. In summary, the concept of sensitization is fundamental to understanding how a fault is detected from the input and output lines only. However, the process of determining a sensitized path(or paths) in a general situation is not a simple procedure.

### 2.3.2 Consistency

As shown above, some logic assignments and conditions are needed to carry out the sensitization process. However, just formulating such conditions does not always guarantee that an input vector satisfying such conditions also exists. Thus, formulating conditions to create a change and to propagate the change along a sensitized path

is just one step. The second equally important step is to determine which, if any, vector(s) satisfies such conditions. When this process is carried out by exploring the circuit structure, it is often referred to as the line justification or consistency process.

An ideal line justification algorithm will, at each step, make a decision that will not have to be changed. In general, however, this is not possible since making an irreversible decision requires knowledge which is not available at the time of decision and can be obtained only by reversing the decision and starting again. The most one can do in this situation is to use some insights or heuristics so that as few decisions as possible are changed. Actually, it is due to this decision process that the test generation problem is NP-complete [21].

### 2.3.3 Redundancy and undetectability

A fault is said to be undetectable if there is no vector to detect this fault, and the line associated with the fault is called a redundant line. For instance, in the trivial circuit of Figure 2.4, the fault 5 s/1 is undetectable, since sensitizing it would require that each of lines 3, 4, and 6 be a 1, implying in turn that $x_1 = 1$, $x_2 = 1$, and $\overline{x_1.x_2}$ = 1. These being contradictory requirements, one can conclude that if 5 s/1 existed in the circuit, then as far as the input/output behavior is concerned, the circuit is going to behave as if there is no fault in it. Such an undetectable fault would seem to be harmless when not probed further. However, as previous research in the area has shown, in order to be able to carry out an effective detection for the detectable faults, one must know where the redundant lines in the circuit are. For example, in the circuit of Figure 2.4, the input vector (1, 1, 0) is a test vector for a 1 s/0. However, (1, 1, 0) cannot test 1 s/0 in the presence of the undetectable fault 5 s/1. Thus, an undetectable fault can invalidate the testing of some detectable faults if both are

present simultaneously.

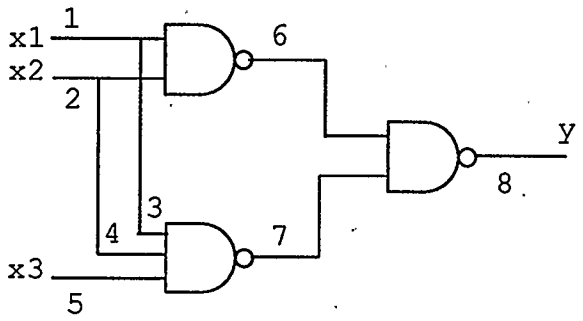



Figure 2.4. Example of redundancy.

Another effect of an undetectable fault is its impact on the test generating efforts for a given circuit and a fault set. If a fault set is undetectable, any resources spent in trying to obtain a test vector are wasted. It is thus useful to remove all the undetectable faults from the fault set before the test generation step. As it turns out, even the process of determining whether a fault is detectable or not is as complex as the test generation process which is NP-complete. The best hope, therefore, is to avoid the appearance of redundant lines during the design phase of the circuit under consideration.

## 2.4 Current Test Generation Approaches

### 2.4.1 Random Test Generators

The concept of generating test vectors for a digital circuit by some random process probably provides the simplest approach to the test generation problem [2, 52]. The major current issues for random test pattern generation are: selecting the test length, determining the fault coverage, and identifying random-pattern resistant faults (faults that are hard to detect with random patterns). These could, in principle, all be ac-

complished by a full single-stuck fault simulation of the network to be tested [56]. The development of special-purpose equipment is decreasing the cost of fault simulation. Despite this cost reduction, full fault simulation remains expensive for large circuits that require long random test sequences for adequate fault coverage.

The only viable alternative to full fault simulation appears to be the use of a probabilistic model of random test generation [39]. Probabilistic methods do not give exact fault coverage values, but they do provide more insight into the relations between circuit characteristics and test parameters.

### 2.4.2 Deterministic Test Pattern Generators

The problem of deterministically generating a test pattern for a given fault is to find a combination of assignments of logic values (0 or 1) to the primary inputs which:

- excite the target fault,
- monitor the target fault at, at least one of the primary outputs.

Since the properties of deterministic test generation fulfill the requirements for a systematic search problem, automatic test generation algorithms usually build a decision tree and apply a backtracking search procedure [21, 24], in order to find a solution for problem.

The $D$-algorithm [44, 45] is probably the most known test generation algorithm. It develops a five-valued $\{0, 1, X, D, \overline{D}\}$ calculus to be able to carry out the sensitization and the line justification procedures in a very formal manner. In this calculus, each line can be either a 0, 1, X (unknown), $D$, or $\overline{D}$. The faulty line is assigned a $D$ or $\overline{D}$ depending on the fault on the line. The next step is to use the calculus and

the circuit structure information to determine values on the other lines so that the $D$ or $\overline{D}$ can be sensitized to the primary output line. A line justification step is then carried out to justify the values assigned in the preceding step. Both the sensitization and line justification steps may have to be carried out many times before a test vector is obtained.

The PODEM (Path-Oriented Decision Making) algorithm was introduced in particular to perform better than the $D$ algorithm for circuits containing mostly XOR gates. It was, however, demonstrated to have a better performance than the $D$-algorithm for various other types of circuits as well. The approach taken by PODEM appears to be the first to treat the test generation problem as a classic branch-and-bound problem. More fundamentally, the algorithm starts by assigning a value of 0 or 1 to a selected primary input (Pi) line, and then determines its implication on the propagation of $D$ or $\overline{D}$ to a primary output line. If no inconsistency is found, it again somehow selects another Pi line and, assigns a 0 or 1 to it, and then repeats the process, which is referred to as *branching.* Since DALG ($D$-Algorithm) and PODEM are complete algorithms, given enough time, both will generate tests for each testable fault.

It is obvious that to accelerate an algorithm for test generation, it is necessary to reduce the number of occurrences of backtracks (branching-bounding cycles) in the algorithm and to shorten the processing time between backtracks. Based on that, the FAN [21] algorithm started with the basic conjecture that the PODEM does not fully exploit its framework. FAN has employed a better heuristic in the bounding-and-branching steps to speedup the test generation process. Results show that FAN is more efficient and faster than PODEM. The average number of backtracks in FAN is lower compared to that of PODEM.

Schulz et al. further improved the performance of FAN by improving the implication procedure and built a test generation system called SOCRATES [51]. They described a unique sensitization procedure and an improved multiple backtrace procedure. Marques and Sakallah [53] have presented several new techniques to prune the search space in path sensitization problems. These techniques explore dynamic information provided by the search process, both before and after inconsistencies are detected. In other approaches, a forward propagation procedure has been used in providing the necessary information to guide the test generation process. Jone and Madden [28] have applied this technique to generate a minimal single fault tests for fanout–free combinational circuits. They have also proved that this minimal test set covers all the multiple faults in the circuit. This technique is proved to be much more difficult for the general class of combinational circuits [13]. In general, using different strategies for test generation lowers the testing time [40].

There are other algorithms which have used formal methods for test generation. Larrabee [33] applied a satisfiability (SAT) algorithm to Boolean formulae which express the Boolean difference between the correct and faulty circuits. Chakradhar et al. gave a transitive closure algorithm for test generation [12].

It is expected that the trend for new approaches and improvements over current approaches will continue.

## 2.5 Modular Test Generation

Modular testing has been proposed as an alternative to the brute testing of VLSI chips. The goal of modular testing is to simplify the chip test by partitioning the chip into modules and test each module separately. This technique is compatible with the hierarchical approach in designing VLSI circuits which is available on most CAD

tools today.

The typical VLSI circuit or ASIC (Application Specific Integrated Circuit) contains not only random logic but also RAMs, ROMs, PLAs, and complex macros such as microprocessor cores, data paths, and multipliers. Designers create some of these blocks with logic synthesis or module generator; others are predesigned macros. Because of this variety of structures and functions, such ICs are called heterogeneous circuits. A test engineering system must cover a large variety of highly complex, heterogeneous circuits, but most available tools handle random logic only. Applying a modular test strategy is one of the most efficient ways to tackle these heterogeneous circuits. Within the module concept, each block is made controllable and observable independently. Then, the testing of the chip is reduced to testing the modules separately, where each of the modules is best tested with its own dedicated test technique. This approach is also referred to as *Macro Testing.* Macro testing completely solves the chip level test problem and ensures high fault coverage.

The success of modular testing depends entirely on the technique used in achieving full controllability and observability for the primary inputs and outputs of each module in a chip. In this context, module testing does pose some challenging problems: partitioning, selecting a test technique suited to the separate module, assembling module tests up to a chip test, and executing a module test independently of its environment. Solving these problems may lead to a chip with a significant test quality, not only at the chip level but also at the board level. Nevertheless, resolving these issues will always come at undesirable cost. We have seen many examples where the chip manufacturer compromise the chip quality, by not adding DFT (Design For Testability) measures such as modular testing, in order to reduce the cost of production.

Researchers have tried to solve these problems with brilliant ideas but with little success due to the costs associated with their techniques. The most notable work in this direction was presented in [18] where each module is made fully controllable and observable independently through busses and extra hardware. This hardware is added at the modular level so that the accessibility of each module is guaranteed. Although this technique is practically sound, the drawback in most cases is the extra cost (not only in terms in area but also in terms of delay) associated with this technique.

Other researchers [43, 41] used algorithmic approaches to generate tests for each module and use the functionality of other modules to create chip tests from a module's tests. In [54], instead of using functional heuristics to generate chip tests, symbolic paths which represent the onto mapping between the PIs and POs of a module and the PIs and POs of a chip are created. These symbolic paths, representing the controllability and observability of a module with respect to the chip's PIs and POs, are used to generate the chip tests from the module tests. This avoids the drawbacks of adding extra hardware to the chip, but it adds the extra cost of running the modular test assembly algorithm. In the matter of fact, the major drawback of algorithmic techniques for modular testing is that the test quality of the chip is lower than the test quality using extra hardware techniques. The reason being that the extra hardware technique improves the controllability and observability of modules while the algorithmic approach does not. This fact, although extremely important seems to be gone unnoticed by the researchers in this field.

Another fact which we believe favors adding extra hardware for better testability over the algorithmic approach is the resultant test length for the chip; a crucial figure in determining the test cost. The test length of the chip under test using extra hardware techniques equals to the total number of tests for all the modules in the

chip. On the other hand, in the algorithmic approach, the test assembly will map the module tests into chip tests. Some of the test vectors of a module may not be mapped due to unjustifiable logic assignments (controllability or observability problem). In this case, the designer will either compromise the test quality by ignoring this vector or decide to run ATPG system to cover the faults originally detected by the module test vector. Normally, the ATPG algorithm will be low level (not modular) and may results in more than one test vectors. It is safe to say that such approaches do not realize truly modular or hierarchical test systems.

## 2.6 Summary

In this chapter, the test generation problem has been presented and formulated. It has been shown that the test generation problem is a complex problem and is considered to be NP-complete. Different approaches have been used to tackle the test problem, either by randomly generating test vectors or by using other deterministic test generation methods. Test generation, as a space search problem, has evolved in designing efficient algorithms as the case in PODEM and FAN. Most of the test systems reported for this decade are based on these two algorithms. SOCRATES [51] algorithm, for instance, is an improved version of FAN. Based upon the sophisticated strategies of the FAN algorithm, an improved implication procedure, an improved unique sensitization procedure, and an improved multiple backtrack procedure are described. In general, however, most of the work in the testing area is useful under very specific circumstances. Despite the steady growth in the area of digital system testing, it has yet to witness the development of a consistent framework which can provide efficient testing algorithms for large and complex systems. Modular (hierarchical) testing is shown to be one of the most promising approaches for solving the test problem for large and complex circuits.

After researching the above problems, we have came to some conclusions concerning modular or hierarchical test generation. The most important conclusion is that any modular testing system can survive only if the cost issues are considered within its test strategy. Secondly, test quality should not be compromised, otherwise, the test efforts cannot be justified. In order to achieve that, we have proposed in this thesis new powerful techniques for the test generation of test vectors for VLSI circuits. The framework on which these tools are built is global test generation. It is a new and powerful framework which creates new directions in defining the test generation problem. This tools have been designed so that their benefits go beyond the chip level testing to the modular (hierarchical) level of testing. We propose different strategies for modular testing to suite different applications and DFT techniques. These tools and procedures will be presented in Chapters 3, 4, 5, 6, and 7.

# CHAPTER 3

# GLOBAL TEST–BASED MODEL FOR TEST GENERATION

In this chapter, we present a new model for test generation. We developed a test model that is based on a non–target fault strategy. We refer to such strategy as *global test generation.* We have also established the first formal characterization of the test primitives generated within the proposed test model. This characterization will ensure that each test primitive can be dealt with as a *test entity.* The test entity should be an integral part in any module design. First, we define the term Global ATPG (GATPG) within the context of the test problem. Then, we identify the necessary requirements to build such a system. This must be done in the light of our goal, namely, how to build a test generation system that can be integrated efficiently in a hierarchical[1] test system?

## 3.1 Global Testing and Backtracking

Our model for test generation aims at eliminating backtracking during test generation. Backtracking is the most time consuming procedure in current ATPG algorithms. In order to solve this problem, we have developed a new technique called global test generation. Global testing means that tests are generated collectively for

[1]The terms modular and hierarchical will be used interchangeably throughout this thesis. Modular test generation will be used whenever test generation for modules at some level in the design hierarchy is considered. Hierarchical test generation will be used when referring to the hierarchical control of the test activities at different levels in the circuit hierarchy.

all the testable faults in a circuit, i.e., the only input to the test algorithm is the circuit structure (there is no input set of modeled faults). We may also refer to this system as a non-target fault test generation system because, unlike other approaches, it does not start execution with a target fault. We believe that this is the first approach that models with the test problem in this way.

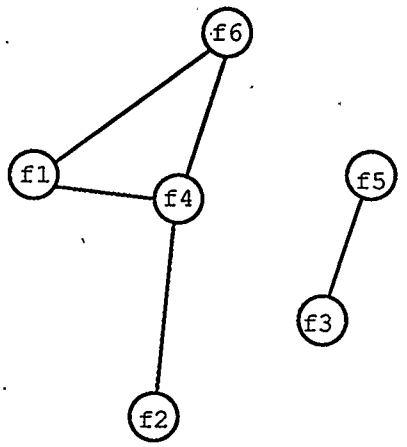



Figure 3.1. An example of a set of faults.

Global testing also offers an alternative to search strategy switching which is based on target fault testing. Consider the set of faults shown in Figure 3.1. A line between $f_i$ and $f_j$ means that fault $f_i$ is compatible with fault $f_j$. There are three compatible set of faults in Figure 3.1: ($f_2$, $f_4$), ($f_3$, $f_5$), and ($f_1$, $f_4$, $f_6$). Let us assume that each set of compatible faults requires a distinct search strategy to cover the faults that belongs to this set. Then, three search strategies must be employed in a test generation system to achieve the minimum number of backtrackings. On the other hand, in the context of global testing, all faults are considered collectively for testing. Tests will be generated for all testable faults while redundant faults will be automatically singled out during the test process. The difference between our approach and conventional test generation is that we require the generation of all unique sensitization paths,

without reference to the faults which they sensitize.

## 3.2 Global Automatic Test Pattern Generation (GATPG)

Although we claim that GATPG has not been presented before, global ATPG is a term that is used by test engineers to describe the test generation process for heterogeneous circuits. The term *global* refers to the ability to generate tests for different design applications, such as memories, combinational circuits, and finite state machines, that exists on the same chip. Therefore, as far as the test generation process at the gate level is concerned, global test generation is not considered before, and hence, our claim still holds. In order to explain what we mean by global ATPG, it is better to look at current test strategies in more detail and then evolve with a clear idea about the concept of global testing in test generation algorithms.

There are some distinct features that are common to most ATPG systems. Among these features is that the ATPG system resources are directed to searching the space of a circuit to find a test cover for one particular fault, referred to as the *target fault.* As explained in the previous chapter, the search process is complex and time consuming. Therefore, it is much easier to consider only one target fault during the search process. There has been so much dedication to solve the test problem within this framework. Such framework succeeded because of its simplicity in relating to the complex test problem. Another feature that is common in current ATPG systems is the two phase approach. The first phase in any of the existing ATPG systems is the random test generation phase. Over 90% of the modeled faults in a circuit are covered during this phase. The second phase, which normally takes most of the test generation time, is the deterministic pattern generation phase. In this phase, faults that are not covered by the random vectors are searched. As a result, a cover for a

fault is generated or the fault is proved to be redundant, that is, a test which covers this fault does not exist.

As the circuit size and complexity increases, less number of faults are detected by the random phase. More faults in complex circuits happen to be random vectors resistant. In order to control the time–to–market issue, the efficiency of the deterministic test generator must be increased to contain the increase in the number of uncovered faults in random phase. Judging by the published results for current ATPG systems, it is easy to see that these systems take large amounts of time to generate tests and to prove redundancy for a limited number of faults. This does not mean that current ATPG systems do not solve the test problem, but it simply means that they cannot be used efficiently, i.e., cost–wise, with complex circuits.

We propose a completely different framework for solving the test problem. Our approach is based on what we call global ATPG. The approach is global because it considers the problem of generating tests for more than one fault simultaneously. We do not use the term target fault because the GATPG does not use any explicit set of faults as target faults. In the context of GATPG, tests are generated without referring to the faults they cover. At any time during the GATPG process, many faults may evolve as candidates to be covered simultaneously. The concept of having more than one fault explored at any time in the search space makes the computations in the search space universal and cover the sub–search space of many faults simultaneously. This does not necessarily means that such framework will target multiple–faults, but it rather aims at utilizing common search spaces for different faults to generate common sub–solutions.

Consider the circuit shown in Figure 3.2 and the single faults $a/1$ and $b/1$ ($a$ and $b$ are stuck–at 1). These two faults are not compatible because they have different logic

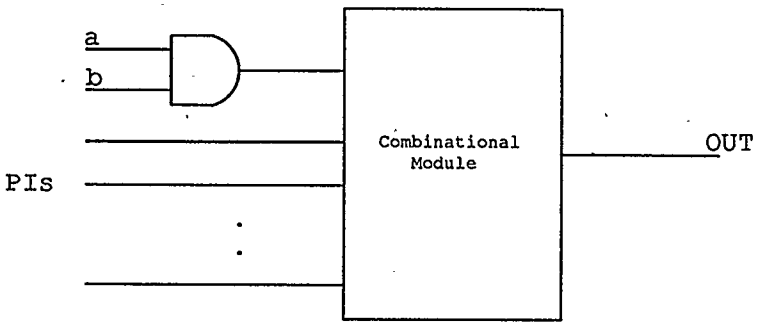



Figure 3.2. A combinational circuit block example.

assignment requirements at the inputs of the AND gate. As depicted from the figure, if a test cover exists for one fault, a test for the other fault will also exist. The logical search space for both faults is the same in the combinational module but different only at the AND gate. A typical ATPG system will search the combinational module space twice before a test for each fault is generated, as long as the generated cover for one fault does not cover the other fault. Global search, on the other hand, would allow the search of the combinational module only once. This will create a common sub–solution for both faults by assigning logic values to all primary inputs except at nodes $a$ and $b$. It is clear that to implement such methodology, the search space for both faults must be explicitly exposed to the search algorithm. This requirement will be provided in Chapter 4.

## 3.3 Important Issues in the GATPG Framework

Now that the concept of global ATPG is defined, it is worthwhile investigating the possibility of modifying current ATPG approaches so that they might incorporate the global test concept within their framework. We believe that this is an important point, otherwise we will not be able to justify the need for a new test algorithm.

Global testing can be applied to the test generation problem for current ATPG algorithms in many ways. One way is to search the circuit for test covers for more

than one fault simultaneously. Another way is to apply different search strategies, in a multiple search strategy system, globally. These approaches will decrease either the number of search cycles or the average time needed to cover a single fault. Eventually, the overall test generation time will decrease as well.

Current test generation systems use only single target fault strategy. The idea is that they generate a cover for a target fault and then fault simulate this cover to reduce the number of faults in the list of target faults. Let us assume that an ATPG system has been modified to search for covers for more than one fault simultaneously. In order to apply global testing techniques in such systems, one should make sure that the starting set of target faults, which will be searched simultaneously contains incompatible faults. Otherwise, faults which can be covered with a single test vector might be searched by exposing their combined search space. This definitely will cause waste in computer resources. However, the problem of searching a circuit for incompatible sets of faults is itself NP–hard which makes it even difficult to apply GATPG to the test problem with the current approaches. The multiple search strategy approach can be applied to current ATPG systems using multiprocessor environment in which each processor uses a separate search strategy for the fault under test. Large speed up factors may be achieved on the expense of more complicated hardware systems. Therefore, none of the GATPG approaches can be implemented efficiently within the context of current ATPG systems.

Since we have justified the need for a new global test framework, it is helpful to look at the expectations and challenges that lie ahead in implementing GATPG systems. Two facts can be extracted from the discussion of Figure 3.2. It is expected that the test generation time will depend entirely on the algorithm's implementation because the search space of each part in a circuit will be explored only one time in the

context of global search for tests. Therefore, the efficiency of the test procedures that are applied to the different parts of a circuit will determine the overall efficiency of the test system. Of course, the circuit complexity will have a great impact on the system's performance but it is better to think of an implementation which simplifies the test process rather than being concerned about how complex a circuit is. An implementation which incorporates test procedures that put different circuit complexities on equal footage will be highly desired.

The other fact which appears to be very challenging in the implementation of global ATPG systems is the memory requirements. The very thought that the search space of many faults will be explored simultaneously by the test algorithm makes it appear very difficult to manage large size circuits. But again, one thing we learn from other ATPG systems, there will always be some constraints imposed on the search space. For instance, the limit on the number of backtracks used by most ATPG algorithms aims at reducing the search space for the test problem. Our challenge is to devise a way to contain the space explosion during the test process in such a way that optimal performance is achieved without degradation in test quality.

## 3.4 Modular Aspects in the GATPG Framework

The ultimate goal of our GATPG algorithm is to generate tests that can be used efficiently in a modular test generation system. The GATPG algorithm will be referred to as a *low level test system* since it generates tests at the gate level, while modular testing will be referred to as a *high level test system.*

A system is modular when it can be described as a collection of modules with limited, well–defined interfaces. A test system is modular if it can use the set of test vectors which covers all the faults in the module and a description of well–defined

interfaces of modules to generate tests at the primary inputs of a chip. The test set of a module is referred to as the *test primitive* of the module. The well–defined interfaces are accessible through high level description (netlist) of the system's modules.

The most important aspect in the design of modular test generation systems is how well defined is the interface between the test generation algorithm for the internal circuitry of the modules (low level testing) and the modular test procedures which translate the test primitive to the chip's primary inputs (modular testing level). This particular point, although ignored by researchers, has a great impact on how truly modular the test system is. The description of the interface between the two levels of testing should be sufficient to assure that the modular test procedure will function completely at the system level, without reference to internal circuitry of modules. Any referral to the internal circuitry of modules will cause the system to flip back to low level testing. It should be noted that the test interface at the two levels of testing is represented merely by the test primitives of modules, i.e., the output from low level testing will be the only input to high level testing.

It must be emphasized here that current hierarchical test systems fail to deliver the above mentioned requirements for modular test systems. Although the term test primitives is used regularly in these approaches, it has never been formally defined or looked at in more depth to determine the conditions and constraints that if imposed on the test primitive would render the test system fully hierarchical. Instead, researchers have looked at the two levels of testing (low and modular testing levels) differently and devised ways to solve each testing level separately. The results from these efforts showed a gap between what the system is specified to do and what it can deliver. This fact manifests itself in different ways. For example, in [54], test primitives are not the only data that define the interface between modules. A procedure which

calculate the observability and controllability at the modular level is incorporated in order to completely define the modules test interfaces. The time to run such procedure and the memory space occupied by data will definitely increases the cost of test generation. Another manifestation of the problem can be seen in [43] where the heuristics used for modular testing is based on the functionality of modules. At any time during modular testing, the system functional heuristics may fail due to the limitations imposed on these heuristics. In that case, the test system flattens the circuit to the low level of abstraction and then applies low level test generation heuristics to cover the fault under test. Such behavior, if repetitive, makes us question the efficiency of such systems. We must keep in mind that the purpose of modular testing is to control the test complexity of VLSI chips. In considering any module in the circuit, irrelevant details about the other modules can be suppressed by hiding them behind the interfaces. Eventually, a truly hierarchical test system will use the test primitives of a low level test system as descriptions of input data at the next level.

## 3.5 Characterization of Test Primitives

It is imperative that for a modular test system to be successful, its underlying low level test generator must inherently support modular testing. Since our GATPG system represents the low level test algorithm which will generate the test primitives, it is reasonable to characterize the required information in a test primitive and develop a strategy to generate this information during test generation.

In order to characterize the test primitives, we need to determine first what kind of heuristics are necessary at the modular test system. Then, we will try to match these heuristics with corresponding requirements from the test primitives. Once all

heuristics at the modular level are matched with corresponding information in the test primitives from the low level of testing, only then will the proper interface between the two levels of testing be achieved.

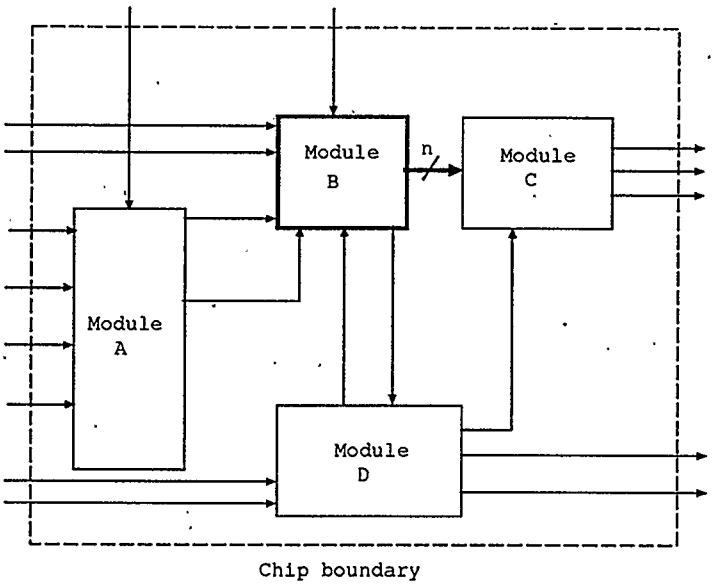



Figure 3.3. An example to illustrate modular heuristics.

Consider the circuit shown in Figure 3.3. If the test primitive of module $B$ is to be mapped into the primary inputs/outputs at the chip boundary, we expect a number of heuristics to be applied. These heuristics involve the propagation and justification of the logic values in the test primitive of module $B$ across the inputs/outputs of other modules. The number of successfully mapped test vectors will determine how many faults in module $B$ will be covered by the chip test vectors. Some test vectors in the test primitive may not be mapped to the chip boundary due to the lack of controllability from the chip's primary inputs, which is responsible for setting the logic values at the inputs of module $B$ to the values specified in the test primitive.

In order to match these complex heuristics at the modular test level, the test primitive should include information on the one to one onto mapping across the

module's inputs/outputs. This information is usually referred to as symbolic paths. Generally, the problem of generating symbolic paths is dealt with as a separate issue from the problem of generating test primitives. In our approach, on the other hand, we did not separate these two problems but rather dealt with them as a single entity. Therefore, our GATPG framework aims at generating test primitives according to the following criteria:

- Each pattern in the test primitive represents a sensitization path that is generated using our GATPG algorithm. In other words, the test primitive includes the test vectors for the module under test.
- Propagation and justification heuristics are represented symbolically within the test primitive. This representation allows the modular heuristics to be applied without reference to the internal circuitry of the module. This representation is complete, i.e., there is no need for any other procedures or data representation during modular testing.
- The representation of symbolic paths is achieved using test procedures in our GATPG system. Therefore, no additional functional heuristics are needed to generate them.

The first criterion is the only one that we share with other existing hierarchical test systems. The other criteria are two further improvements in the direction of fully modular test generation system. The second criterion states that the interface between the low and modular levels of testing is well–defined, which guarantees successful modular heuristics without exception. The third criterion states that in order to generate symbolic paths, we really do not need any extra procedures. This is because the heuristics used to generate these paths is the same one used in our test

generation system. Since test vectors and symbolic paths use the same heuristics, they are generated globally as well. This fact adds another level of simplicity in our test generation system. The impact of these representations in the test primitives is substantial as will be seen in the coming chapters.

## 3.6 Test Quality

Since we have developed a general idea about our test strategy, it is equally important to look at the impact of GATPG on the test quality. Test quality will be affected by the efficiency of the test algorithm and the fault model used in the test procedure. The efficiency of the GATPG algorithm will be judged by the fault coverage and the time it takes to generate the test vectors. These issues will be dealt with after the implementation of the GATPG algorithm. The fault model, on the other hand, is a priori condition and should be considered before the implementation of the GATPG. However, we have decided to take a challenging step in the design of our ATPG system in order to ensure the test quality, that is eliminating the random test generation phase as a front end in our system. Our GATPG is single phase, which means that the GATPG is responsible for generating all the tests required to cover all the modeled faults in a circuit. In this way, we ensure that the test quality of our system is always maintained regardless of the circuit complexity and however resistant its behavior is to random testing.

The accuracy of the fault model is defined as the number of physical defects that are captured by the modeled faults in respect to the total number of possible manufacturing defects. The stuck–at fault model is the most popular model among others and can represents most of the physical defects in VLSI systems. However, it has been shown that the classical gate level stuck–at model is inaccurate for some faults

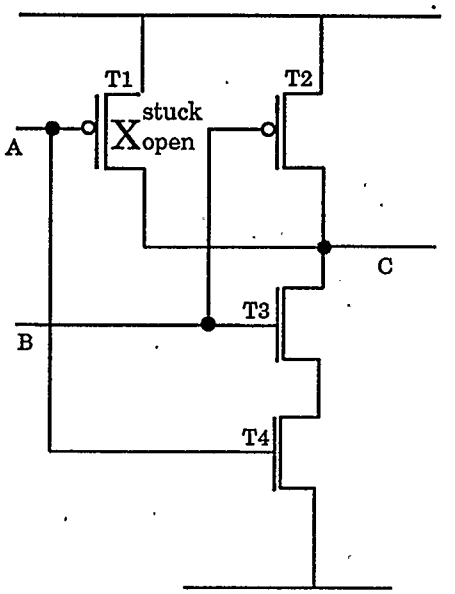



Figure 3.4. An example of a transistor level fault that cannot be described using the stuck–at model.

in today's CMOS process. Consider, for example, the NAND gate shown in Figure 3.4. If transistor $T_1$ is permanently open, then this fault can only be detected by a two–pattern test. The initial pattern sets the output C to a low value ($A$=1, $B$=1). Then, the evaluation pattern tries to turn $T_1$ on ($A$=0, $B$=1). In case $T_1$ fails, the output remains low, otherwise, the output becomes high. This type of faults cannot be captured using the stuck–at model.

As far as our modular test system is concerned, any fault model used at lower test level should not alter the way the test primitives are generated. As a matter of fact, any low level test system may use any suitable test procedures with any fault model as long as it generates the test primitives as specified before. Since we are dealing with circuits at the gate level, the stuck–at fault model has been chosen for implementation in our GATPG algorithm.

## 3.7 Summary

In this chapter, we have presented a new framework and a test model for solving the problem of test pattern generation. Our model for test generation aims at eliminating backtracking. In order to solve this problem, we have developed a new technique called global test generation. Global testing means that tests are generated collectively for all the testable faults in a circuit.

Since our objective is to build a modular test generation system, we have analyzed the different modular aspects in the context of our proposed global test generation framework. We have identified the required features in our GATPG system that will lead us to a successful implementation of a single fault and a modular test generation system. Accordingly, the test primitives have been characterized in the light of our objective. The impact of our test strategy on the test quality has been discussed as well.

# CHAPTER 4

# AN EFFICIENT GATPG ALGORITHM FOR COMBINATIONAL CIRCUITS

The global test–based model described in the previous chapter will be used as a guideline for the design of an efficient global automatic test pattern generation algorithm for combinational circuits. The GATPG is a single phase global test generation system. In this chapter, we will describe the necessary design steps needed to implement the global framework in the test problem. Again, it is emphasized that decisions at this stage will be highly dependent on the final goal of this algorithm. The interface between procedures needed at the modular test level and those needed at the GATPG level will always be in mind in order to achieve a complete and sufficient description for the test primitives generated by the GATPG algorithm. Experimental results will be given at the end of this chapter. The core of this chapter will be presented in the ICCD'95 [58].

## 4.1 The Test Generation Model

The test generation model is a detailed description of how the global test framework is going to be applied to the test problem. We will first discuss some global testing issues, the test generation problem, and then presents an outline of how to approach and implement the test generation algorithm.

### 4.1.1 Global Testing Issues

By looking back into the desired description of a test primitive which will guarantee the creation of a fully modular test level, it is imperative that the justification and propagation procedures at the modular level must be represented in the GATPG algorithm. The question which can be asked is what exactly is meant by propagation and justification? At the modular level, the propagation procedure can be defined as a procedure which propagates some logic values at the inputs of a module to its output nodes (probably using a simulator). The justification procedure does the same thing, but, the other way around, that is generating input logic assignments that will justify some known output logic values. Both procedures represent mapping logic values between the inputs and outputs of modules. Therefore, the keyword here is mapping.

Let us now consider the test problem and see how can we incorporate this key issue in the test generation procedures. First, such a mapping as explained above has never been thought of in any of the existing ATPG systems. The reason is that the purpose of such systems is to generate tests for the whole chip without support for any higher test levels. Even when some of these systems are used to generate tests for modules, they usually use another procedure to achieve higher test levels, as explained in Chapter 3.

In order to elaborate on the mapping issue, consider the module shown in Figure 4.1. In order to propagate a set of logic values from the primary inputs to the primary output, an implication procedure should be applied where the output of a module is determined by the input logic values. The implication procedure is simple and straightforward. The justification procedure, on the other hand, requires the back

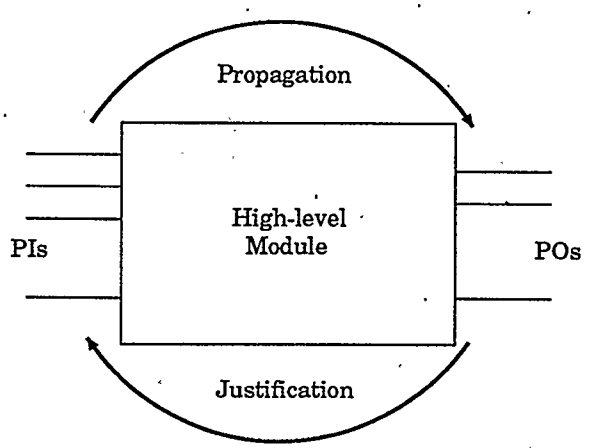



Figure 4.1. An example showing the propagation and justification procedures at the modular test level.

propagation of the logic values at the primary outputs to the primary inputs. This step is very complicated and time consuming when applied to large circuits. The outcome from the justification step depends on the circuit structure, i.e., given some logic assignments at the primary outputs, there may be one or more vectors that can be mapped to the primary inputs while there might be a case where no vectors can be mapped to the module's primary inputs. We conclude from the above discussion that mapping can be achieved at the gate level using implication and back propagation procedures. Throughout the rest of the thesis, the term *mapping information* will be used to express the data generated by the GATPG algorithm which represent the data needed to perform the propagation and justification procedures at the high level of test, without reference to the module's internals. This term must not be confused with the term *fault mapping* which is used to express the translation of a fault value at a primary output of a module to a number of fault patterns at the primary inputs of the same module.

Although our aim is to create a test system, there is no discussion so far on how the test vectors are generated. As mentioned earlier, high level test interface issues

are resolved first and then the test strategy will be adapted accordingly. Now, it is important to decide which strategy we are going to use to generate the test vectors. The first thought is to use any known strategy for test generation, but then we will end up with a system that has many of the disadvantages of other hierarchical ATPG systems. The cost issue comes to the picture if we try to incorporate one strategy for test generation and another strategy for creating the mapping information because we will be loosing time in running two different algorithms, one for generating the mapping information and the other for test generation. Instead, we have adapted and *embedded* the concept of global test generation within the framework of the mapping procedures. In this way, all the necessary information (test vectors and mapping) in the test primitive are derived from a single algorithm. As a matter of fact, we will generate a single set of patterns within a test primitive which represents both the test and mapping information.

### 4.1.2 Test Generation Framework

Since the mapping procedures deal primarily with inputs and outputs of modules, it is better to direct our attention to a test framework which put more emphasis on the fault behavior across the inputs/outputs of modules. In order to adapt the test generation strategy within this framework, we have to explain the difference between the problem of generating a test for a fault inside a circuit and a fault at the boundary of a module. A fault internal to the circuit needs to be excited (its logic value set to the opposite of its fault value, i.e., a s/1 fault is excited to 0) and be observed at one of the primary outputs of the chip. The fault excitation step is achieved through the forward propagation of logic values at the primary inputs to the fault location. The fault observation step requires not only the propagation of fault from its location to a

primary output node, but also requires justification of logic values so that no conflict in logic assignments is created. On the other hand, a fault at the primary output of a circuit needs only to be excited for the fault to be detected. Accordingly, it is much easier to deal with faults at the primary outputs rather than those internal to the circuit.

The question that now arises is that if we start with a fault at a primary output, is it possible to generate tests for all the faults inside a circuit? The answer is definitely yes, because any path that sensitize a fault internal to the circuit must eventually end up at one primary output. Put in test generation terms, any testable fault in a circuit must be compatible with one or more primary output faults. The next question is how can we achieve the creation of such tests? Given a fault at a primary output, it is possible to trace back the different paths between the primary output and the internal nodes in a circuit. The procedure which can achieve this purpose is the back propagation algorithm described above in the mapping procedure. When applied to the test generation problem, the back propagation algorithm will create many sensitization paths evolving from a primary output node and extend through the circuit internals and terminate at the primary inputs. Each sensitization path represents a series of faults that are detected by the resultant test vector. The existence of many paths simultaneously makes the test system global. This test strategy can also be described as the problem of mapping a fault logic value to its equivalent set of fault and control logic values at the primary inputs. Therefore, the back propagation procedure is used to generate test vectors and as a part in generating mapping information. The forward propagation procedure is used only in generating mapping information. In this way, a complete test primitive can be generated within a single procedure and without any software overhead. The challenge is to develop

an efficient implementation for the GATPG algorithm.

## 4.2 Problem Formulation

The problem of mapping a fault logic value at a primary output of a circuit to a set of test vectors at the primary inputs is formulated. The test procedures on the basic logic gates will then be defined.

### 4.2.1 Problem representation

Consider the combinational module shown in Figure 4.2. The black box represents a circuit structure with only one primary output. From the basic definition of testing, for any internal fault ($D$ or $\overline{D}$) inside the black box, this fault must propagate to and be observed at the primary output to be covered. The fault may be observed at the primary output as a $D$ or a $\overline{D}$ logic vlaue.

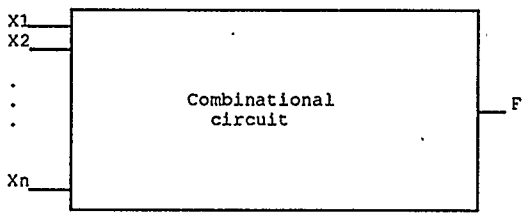



Figure 4.2. A combinational circuit block.

The logic value at the primary output can be expressed as a Boolean function of the primary inputs $X1$, $X2$,.., $Xn$. Global testing can be achieved by tracing back the different sensitization paths between the primary inputs and outputs starting at a primary output node. The problem of global test generation can be stated as one of solving each of the following two equations:

$$F(X_1, X_2, ..., X_n) = D \tag{4.1}$$

$$F(X_1', X_2', ..., X_n') = \overline{D} \quad (4.2)$$

where $X_1$, $X_2$, ..., $X_n$ may assume any combination of the logic values 0, 1, X, $D$, or $\overline{D}$, where X is the don't care logic value. We will refer to the logic values 0, 1, and $X$ as the *control logic values*, while $D$ and $\overline{D}$ as the *fault logic values*. The test vector $X_1$, $X_2$,..., $X_n$ will be referred to as a fault pattern because it might include a fault logic value. In each fault pattern, the control logic values allow the fault logic value to propagate to the primary output. This creates a sensitization path between the primary inputs and the primary output with all the compatible faults on that path covered by the same fault pattern. This definition of global testing reduces the test generation problem to that of searching a circuit for all the sensitization paths between the primary inputs and outputs.

### 4.2.2 Logic Representation in the GATPG Algorithm

The five–valued logic representation has been used in the above discussion to explain some basic issues in our GATPG algorithm. However, the need for the generation of a complete test primitive has lead us to the conclusion that the five–valued logic representation will not be adequate. In current ATPG systems, the fault logic values $D$ and $\overline{D}$ are used to represent a node stuck–at–0 and stuck–at–1, respectively. For example, in the fault–free circuit, the logic value of a node with a $D$ fault value is 0, and in the faulty circuit the logic value of the node is 1.

Since we are dealing with the problem of mapping faults across modules, we are not really interested in the absolute type of faults, but rather interested in the relative fault values between two nodes during the mapping process. Therefore, in our approach, a $D$ or a $\overline{D}$ fault value will be used to express both stuck–at–0 and stuck–at–1

faults. A node which carries a $D$ or a $\overline{D}$ logic value will have the potential of both types of faults being covered for this particular node. Let us assume that the GATPG algorithm starts the back propagation process with a $D$ fault value at the primary output. Any internal node that carries a similar $D$ value is compatible with the fault at the primary output. In other words, the sensitization path which passes through the two nodes will cover faults with the same polarity at the two nodes (both s/0 or both s/1). On the other hand, if their polarities are different, i.e., the internal node has a $\overline{D}$ fault value, then, the sensitization path covers two compatible faults with different polarities (one s/1 and the other s/0). Since the algorithm always knows what value the primary output started with, it is easy to determine the fault–free and faulty response of any node by looking at the fault value it holds. Accordingly, the generated test vector not only carries information about the test path but also specifies its forward implication on the output logic value. This explains why at the low level test generation using our GATPG algorithm, the forward propagation step is not needed.

We have considered the situation where both types of faults can be covered by the test system. There are other situations where only one type of fault (s/1 or s/0) can be detected while the other type is redundant. It seems that adding two more fault values to the logic representation would include this situation in our test system. For instance, one fault value describes a s/0 redundant case, while the other describes the s/1 fault redundancy case. However, the algorithm has the extra task of determining the fault response at the primary output as well. For instance, the algorithm should determine which type of fault is covered and what is the output value in the faulty and fault–free circuit. Therefore, four fault values has been included with the five–valued logic representation. The resultant 9–valued logic representation is used in our

GATPG to express the logic values at different nodes in a circuit. The logic values 0, 1, and X are referred to as *control* logic values, while the logic values $D$, $\overline{D}$, $FD$, $F\overline{D}$, $TD$, and $T\overline{D}$ are referred to as *fault* logic values. The fault logic values are used to express the relative fault values on a sensitization path between an internal node in the circuit and a primary output. They are also used to determine the response at the primary output (faulty and fault–free values). The fault logic values $FD$, $F\overline{D}$, $TD$, and $T\overline{D}$ express a node with the potential of having only one type of stuck–at fault ($s/0$ or $s/1$) being detected. For instance, $FD$ and $F\overline{D}$ represent a node carrying a $s/1$ fault logic value with the primary output having a logic value of 0 and 1, respectively, for the fault–free response.

### 4.2.3 Extensions and Simplification of the Test Problem

It is interesting to know that with the above representation of logic values, it is possible to further simplify the test problem. Since we interpreted a $D$ or a $\overline{D}$ as fault values that represent both types of stuck–at faults, then, starting with either value at a primary output means that we propagate both types of faults simultaneously (globally). As a result, we need to solve only one of the above two equations. Therefore, in the GATPG algorithm, a $D$ fault logic value will be assigned to the primary output and mapped into a number of input fault patterns at the primary inputs. To probe more on this concept, the following definition and proposition will give more insight into the above discussion.

**Definition 1** *The partial complement of an input fault pattern which retains a $D$ or a $\overline{D}$ fault value is another fault pattern with its only fault logic value complemented and each control logic value kept unchanged. For instance, the partial complement of the fault pattern (0, 1, D) is (0, 1, $\overline{D}$).*

**Proposition 1** *If the input fault pattern ($X_1$, $X_2$, ..., $X_n$) which retains a $D$ or a $\overline{D}$ fault value is a test cover for the fault $s/0$ (or $s/1$) at any node on a sensitization path, then, the partial complement of this fault pattern is a test cover for the fault $s/1$ (or $s/0$) at the same node.*

A partial complement of the fault pattern at the primary inputs causes only the fault logic values to be complemented at all nodes residing on the path and retaining one of the fault logic values $D$ or $\overline{D}$. The intuition of this proposition is that all faults on the sensitized path can be covered for both s-a-0 and s-a-1 faults, using the fault pattern and its partial complement, provided that the input fault pattern retains one of the fault values $D$ or $\overline{D}$. The partial complement is equivalent to the problem of back propagating a $\overline{D}$ at the primary output (instead of the $D$ logic value) to map it into primary input fault patterns.

If, on the other hand, there exist some input fault patterns where none of the primary inputs has a $D$ or a $\overline{D}$ fault logic value, then, the input fault pattern will retain one of the other four fault values. In such case, the input fault pattern will support only one type of stuck–at fault to be detected and the partial complement step will not be applied.

For *multiple output circuits*, we need to extend the definition of the global testing problem. A sufficient condition for testing an internal fault in a circuit is to observe the fault at one (or more) node of the primary outputs. Accordingly, test generation can be achieved by iteratively generating tests for each output cone. This can be achieved by assigning a fault logic value (a $D$ or a $\overline{D}$) to one primary output and don't care values to the other outputs.

We can then formulate the test generation problem for the general class of combinational circuits as one of solving the following set of equations:

$$F_i(X_1, X_2, ..., X_n) = D \tag{4.3}$$

$$F_r(X_1, X_2, ..., X_n) = X \tag{4.4}$$

for all $r$, $1 \leq r \leq m$ and $r \neq i$; where $m$ is the number of primary outputs in a circuit.

## 4.3 The GATPG Algorithm

The purpose of the GATPG algorithm is to generate all possible fault patterns at the primary inputs of a circuit. To achieve this goal, a Back–Fault–Propagation procedure has been developed and is presented in this section.

### 4.3.1 Back–Fault–Propagation for Logic Gates

Back–propagation of logic values is a well known technique that has been used both by combinational and sequential ATPG algorithms. In the context of current ATPG algorithms, back–propagation and backtracing are used to create nodes in a decision tree and to justify some previously assigned logic values in a branch–and–bound search environment. In our approach, back–propagation is used to enumerate the possible logic assignments at some nodes in a circuit. No decision tree or justification steps are required.

The basic operation of the test algorithm is to back–propagate a fault or a control logic value at the output of a logic gate to its inputs. Unique fault and control logic assignments are used during the back–propagation process across the gate's input/output nodes. Figure 4.3 shows a NAND gate with its output being assigned different logic values. The algorithm uses all the possible combinations of fault and

control logic values at the gate's inputs which uniquely imply the logic value at the gate's output. As shown in Figure 4.3, for instance, the possible fault patterns at the NAND gate inputs ($a$ and $b$, respectively) are (1, $\overline{D}$) and ($\overline{D}$, 1) with the gate's output being assigned a $D$ fault value. Back fault assignments with a similar form can be defined for OR, XOR, and other types of gates (or modules).

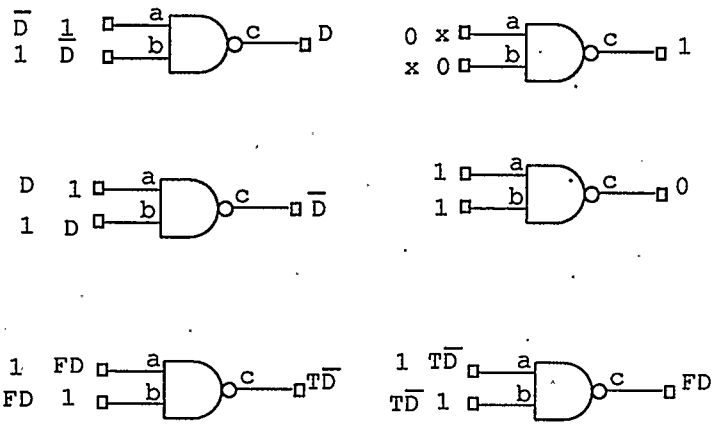



Figure 4.3. The back fault assignments for a NAND gate.

From Figure 4.3, it can be seen that single path sensitization (SPS) is supported by the test algorithm. This implies that faults are covered using only one path, even though multiple paths for a fault might exist which will simultaneously cover the fault. To elaborate on this point, consider the circuit structure shown in Figure 4.4. Let us assume that there is a fault residing at the source node of the fanout structure, as shown in figure. If this fault propagates in the forward direction, more than one fault path will be created. As the circuit structure converges at the output NAND gate, many possibilities exist which allow for any combination of faults (or no faults at all) to appear at the inputs of the NAND gate. If the fault is successfully propagated to the output of the NAND gate with more than one fault value assigned to the inputs of the NAND gate, the fault is said to be sensitized through multiple paths. Multiple

paths may also cause the faults at the inputs of the NAND gate to mask each other and thus destroying the sensitization structure. For instance, if one input carries a $D$ fault logic value and the other input carries a $\overline{D}$ fault logic value, the output of the NAND gate will always be one, even though the effect of the fault is propagated from the fault location to the inputs of the gate.

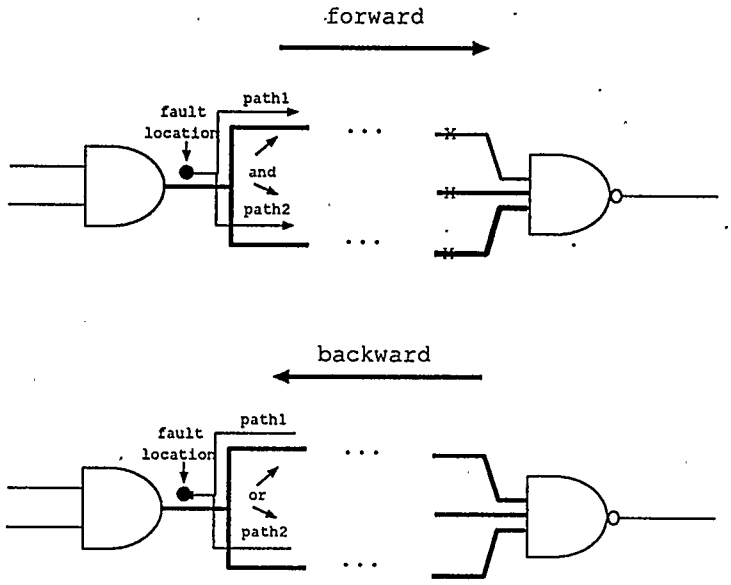



Figure 4.4. Single and multiple path sensitization of faults.

In the GATPG algorithm, only the back propagation is allowed. Therefore, looking at the same example in the backward direction, the GATPG enforces the inputs of the NAND gate to carry one and only one fault at a time. For this particular example, three choices will be allowed for the 3–input NAND gate. Each one of these choices represents a path with one fault value at one of the inputs and control logic values for the other two inputs. When the fault at one of the inputs propagates to the fanout structure, only one stem node will carry the fault logic value, hence, multiple path sensitization is not possible in the back propagation procedure.

In some cases, however, multiple path sensitization (MPS) must be applied to

cover a fault. One way of achieving MPS, for instance, is to add the assignment $(\overline{D}, \overline{D})$ to the inputs of the NAND gate shown in Figure 4.3 (when the output is $D$). Such assignment will have the effect of increasing the size of the assignment tree in the GATPG algorithm. Hence, multiple fault assignments at the inputs of logic gates are not considered in the GATPG algorithm. However, an efficient procedure which supports MPS and uses only the SPS assignments will be presented later in the discussion.

### 4.3.2 The Back–Fault–Propagation Procedure

The key to fast performance is the use of necessary assignments during test generation [14]. The Back–Fault–Propagation procedure guarantees that only consistent logic assignments at the different nodes be progressively propagated toward the primary inputs. Redundant faults cause conflict assignments to appear in the node assignment tree. In this procedure, the detection and removal of conflict assignments occurs at the fanout stems. Therefore, the back–propagation process stops at fanout stems to resolve conflicts in the logic assignment tree.

The back–fault–propagation procedure starts at a primary output node and ends by collecting the fault patterns at the primary inputs. During this procedure, a tree of logic assignments is created. Each node in a circuit will be assigned one or more logic values. We refer to the string of logic values assigned to a circuit node as the *logic queue* of that node.

**Definition 2** *The logic queue is an ordered set of logic values assigned to a circuit node. The first logic value in the set is at the head of the queue while the last element in the set is at the tail of the queue.*

For instance, the logic queue of input $a$ of the NAND gate shown in Figure 4.3 (with its output tied to D) is denoted by $q_a$ and has a length of two with logic $\overline{D}$ at the head of the queue and logic 1 at the tail of the queue. This can be written as $q_a = \{\overline{D}, 1\}$.

A general outline for the Back–Fault–Propagation (BFP) procedure is shown in Figure 4.5. As discussed earlier, test generation is achieved by generating tests for each output cone in a circuit. This is indicated by the first *for* statement in Figure 4.5. An arbitrarily selected primary output is assigned an arbitrary fault logic value ($D$ or $\overline{D}$). The remaining primary outputs are assigned X logic values.

During its execution, the algorithm allows some logic queues to back–propagate their logic contents while delaying the propagation of some other logic queues until certain requirements are met; a *marking* procedure is used by the algorithm to achieve this purpose.

**Definition 3** *A node is said to be marked if the logic value, at the head of its logic queue, is enabled for back–propagation.*

The back–fault–propagation procedure continues the back–propagation of faults as long as there exists at least one marked node in the output cone (the *while* statement). The back_propagate function in Figure 4.5 assigns uniquely implied logic values to the inputs of a logic gate according to the gate's output logic value. After each back–propagation step, the BFP procedure checks if any of the newly assigned nodes is a stem of a fanout structure (the first **if** statement). The back–propagation process stops at fanout stem nodes. This occurs by unmarking all the newly assigned fanout stem nodes from the preceding back–propagation step.

Consider the circuit example and its associated assignment tree shown in Figure

```
Input  : A circuit's netlist.
Output : A set of test patterns.

Procedure Back_Fault_propagation( ) {
  for each primary output;
    {
    assign a D logic value to a selected primary output;
    assign don't care to other primary outputs;
      while there is a marked node i in the output cone;
        {
        back_propagate(q_{i_h}, gate_type);
        if any of the assigned nodes is a stem node;
          if all the stems in a fanout point are assigned;
            {
            compare(q_{s_1}, q_{s_2},..., q_{s_m});
            mark nodes associated with the fanout structure;
            }
          else unmark the stem nodes;
        }
        partial_complement(fault patterns);
        fault_simulate( );
    }
}
```

Figure 4.5. **BFP:** a back–fault–propagation algorithm that globally sensitizes output cones.

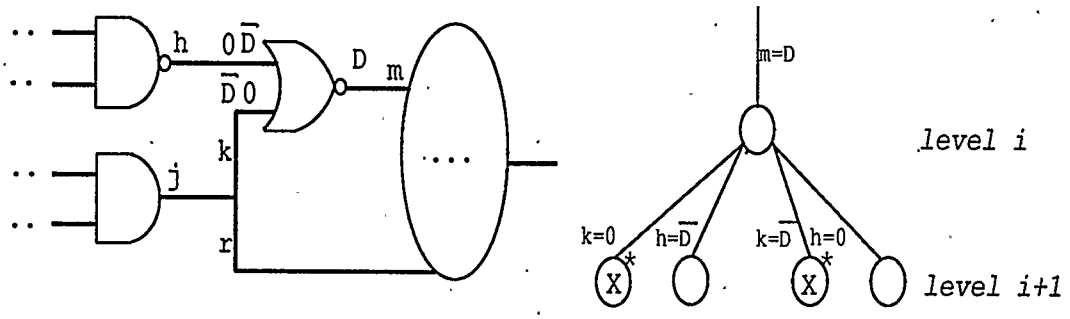



Figure 4.6. A circuit example for marking nodes associated with fanout structures and the sub–tree of logic assignments at node $m$.

4.6. Each node in the tree represents the marking status of a circuit node after it is assigned a logic value. An "X" sign inside a node represents an *unmarked* node, otherwise it is marked. A line drawn between two nodes in a tree represents the logic assignment of the circuit node associated with that edge. An asterisk besides a tree node indicates that this node is a fanout stem. Each node in a tree has a level of assignment. For instance, node $m$ is at the $i_{th}$ level of assignment, while nodes $h$ and $k$ are at the $i_{th} + 1$ level of assignment.

Suppose that node $m$ has been assigned a fault logic value $D$. Then, after applying the back_propagate function to this node, nodes $h$ and $k$ will be assigned as shown in the figure. The algorithm unmarks node $k$. The reason is that it is not known which logic assignment will be consistent with the logic assignments at the other stem node. Hence, the algorithm delays the back propagation of this logic assignment until conflicts are resolved between the fanout stems.

Only when all the fanout stems are assigned logic values, the algorithm uses the *compare*( ) procedure, as shown in Figure 4.5, to compare the logic values assigned to the fanout stems. The consistent logic values will be extracted and assigned to the source node of the fanout structure. In the above circuit example, if the *compare*( ) procedure results in a conflict assignments between the fanout stems, then, the subtree

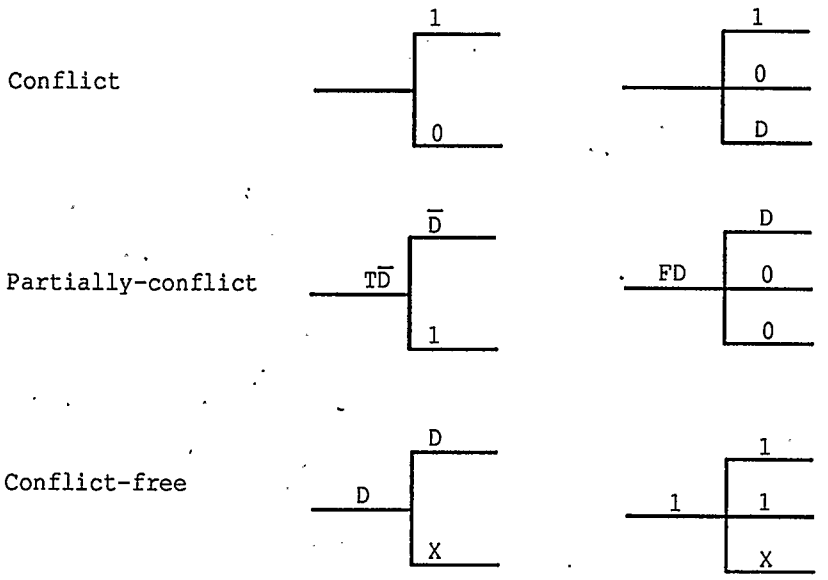



Figure 4.7. Comparison outcomes for logic assignments at fanout stems.

of logic assignments at node $h$ will be discarded. A comparison example between some logic assignments in fanout structures is shown in Figure 4.7. As shown in this figure, three outcomes may result:

- *Conflict assignments*: In this case, there exists no current assignment to the source node that sets the logic value at each stem node to the desired value. Hence, no logic value is assigned to the source node. The logic values at the stem nodes and their accompanying gate inputs will be removed from the assignment tree. For example, if the logic values assigned to node $r$ in Figure 4.6 are in conflict with those at node $k$, then, the logic assignments of nodes $h$, $k$, and $r$ will be discarded.

- *Partial conflict*: In this case, a sensitization path with only one type of stuck–at fault can be supported by the current stem nodes assignments. Consequently, a $D$ or a $\overline{D}$ fault value at a stem node will disappear and be replaced by a single fault logic value that is compatible with the control logic value assigned to the

other stem. For instance, the fault logic value $FD$ in Figure 4.7 is assigned to the source node of the fanout structure after comparing the fault logic value $D$ with the control logic value 0. Hence, $FD$ represents a $s/1$ fault with a fault–free logic value of 0 at the primary output.

- *Conflict–free*: In this case, either a complete match occurs between the logic values at the stem nodes or one of the stems has an X logic value. The source node can then be assigned logic values as shown in the bottom of Figure 4.7.

It can be concluded that there exists only one fault value associated with each sensitized path. Therefore, although different sensitization paths may share some control logic values, each one retains its fault logic value.

The marking of nodes is always revisited after the execution of each compare procedure as shown in Figure 4.5. For instance, if a successful comparison, i.e., either partial conflict or conflict–free, occurs, the source node of the fanout structure is marked. On the other hand, if the comparison results in inconsistent logic assignments, this node will be unmarked.

### 4.3.3 Multiple Path Sensitization

Multiple Path sensitization occurs due to the existence of fanout structures in a circuit. In the GATPG algorithm, MPS is approached by the local search of fanout structures to find a set of consistent logic assignments which allow more than one stem node to carry a fault logic value. This fault logic value will be assigned to the source node of the fanout structure.

The back–fault–propagation procedure accounts for faults which can only be covered by MPS using the cube intersection of the logic assignments associated with each

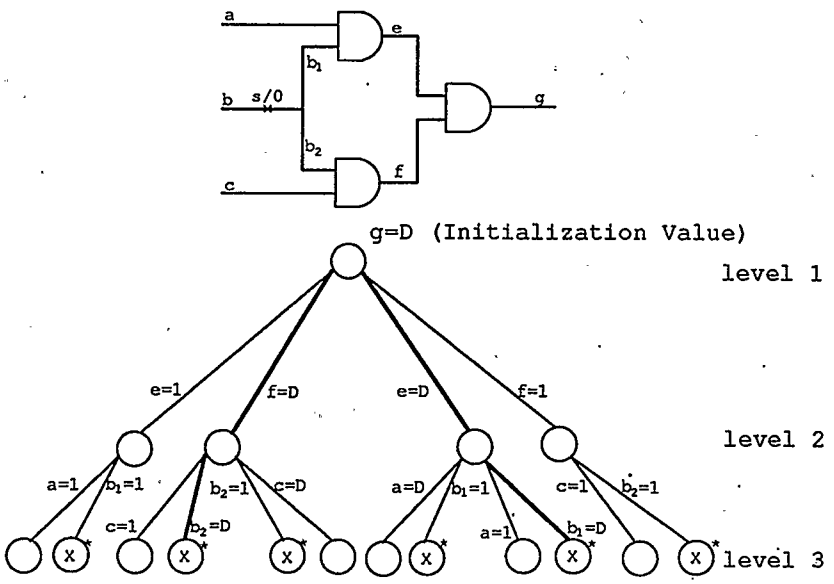



Figure 4.8. A MPS example.

stem node in a fanout structure. In order to achieve MPS using cube intersection, all stem nodes must carry consistent fault logic values in order to generate a new sensitization structure which allows more than one stem node to carry fault values simultaneously. Consider the circuit shown in Figure 4.8. The single faults $b_1/0$ and $b_2/0$ can be covered through SPS, however, the fault $b/0$ can only be covered using MPS. The logic assignment tree, shown in Figure 4.4, behaves as a multiple–valued tree. Each bundle of nodes proceed together and form a subtree of logic assignments. The two bolded paths in the logic assignment tree represent the paths where the stem nodes $b_1$ and $b_2$ carry fault logic values. The logic assignments associated with these nodes are ($a$=1, $b_1$=$D$, $c$=1) and ($a$=1, $b_2$=$D$, $c$=1). The cube intersection of these two patterns results in the assignments ($a$=1, $b$=$D$, $c$=1). The resultant test vector (1, $D$, 1) and its partial complement cover the faults $b/0$ and $b/1$. These faults can only be covered through MPS. In this way, MPS can be achieved using SPS logic assignment and without further increasing the size of the assignment tree.

The procedure which implements multiple path sensitization is shown in Figure 4.10. The MPS procedure terminates if a multiple path(s) which sensitize both type of nodes ($s/0$ and $s/1$) at the source point of the fanout structure is created, otherwise, the procedure continues. If cube intersection results in conflict of assignments between two stem node paths, the process continues with other stem nodes until a multiple path sensitization with maximal number of stem nodes is created. This process can be illustrated using the example shown in Figure 4.9. The search space for a three-stem fanout structure is shown in the figure. If the multiple path procedure starts with stem 1 in the multiple path sensitization list, the path associated with stem 1 is compared with that of stem 2. The conflict may occur if the two fault entries of the stem nodes are not consistent or any control logic assignment in the region of intersection between the stem nodes is in conflict. If such conflict exists, it means that it is not possible to support a sensitization structure which allows stem 1 and stem 2 to carry fault logic values simultaneously. Accordingly, the search space for stem 2 will not be considered any further, as shown in Figure 4.9. The procedure will then try the cube intersection between the sensitization paths of stem 1 and stem 3.

In order to determine which fault entries for stem nodes are consistent with each other, faults are compared at different entries. For example, a $DF$ fault logic value on a stem node will result in a conflict if compared with another stem node carrying any of the fault values $F\overline{D}$, $TD$, or $T\overline{D}$. An interesting case may occur when one stem node carries, for instance, an $FD$ fault value while the other stem carries a $\overline{D}$. Since $\overline{D}$ represents both type of faults while $FD$ represents only a s/1 type of fault, it is expected that comparing these two fault values would results in the sensitization of a s/1 fault. However, although both fault values support a s/1 sensitization path, they produce different logic values at the primary output of the circuit. An $FD$ fault

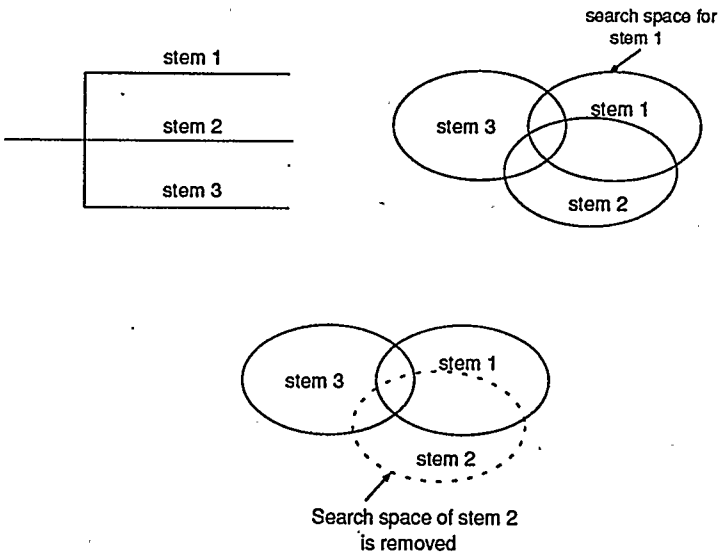



Figure 4.9. Search space representation for a 3–stem fanout structure.

value covers a s/1 fault with a fault–free output logic value of 0, while a $\overline{D}$ fault value would support a s/1 fault with a fault–free output logic of 1. This simply show that although faults at the fanout stems can be excited properly, the faults propagated from the stem nodes will mask each other before they successfully propagate to the primary output. This process is called path sensitization failure or destruction. In short, the following list of fault pairs represent the fault comparison, for each pair, which results in a conflict–free fault assignments at two stem nodes. These pairs are: $(D, D)$, $(\overline{D}, \overline{D})$, $(D, FD)$, $(D, TD)$, $(\overline{D}, F\overline{D})$, $(\overline{D}, T\overline{D})$, $(FD, FD)$, $(TD, TD)$, $(F\overline{D}, F\overline{D})$, and $(T\overline{D}, T\overline{D})$. Any other pair of faults will result in conflict fault assignment.

For a fanout structure with $n$ number of stems, the MPS procedure takes a maximum of $(n\text{-}1) + (n\text{-}2) + \ldots + 1$ cube intersection steps to terminate. Hence, the procedure is linear with the number of stems and does not cause any degradation in the GATPG algorithm's performance.

```
Input    : Stem nodes sensitization paths.
Output : MPS-list A set of stems with MPS.

Procedure Mult-Path-Sens.( ) {
   for each stem node i;
      {
      Let j=i+1;
      Let MPS-list=i;
         while j is less than the number of stems;
            {
            cube-intersect(SPS-path[i], SPS-path[j]);
            if conflict free cube intersection;
               MPS-list=j ;
               if MPS-list covers the source node for both type of faults;
                  exit;
               else continue search;
            else j=j+1;
      }
   }
}
```

Figure 4.10. The multiple path sensitization procedure.

## 4.4 Data Structure and Tree Pruning

The BFP procedure allows all possibilities of path sensitization to be considered. Consequently, the queue sizes of some nodes in a circuit might explode exponentially. Since the test generation approach used in the BFP procedure is different from current ATPG systems where backtrack limit is used to impose space constraint in the search process, a new methodology should be employed to contain the space complexity and to prune the assignment tree during back propagation. The efficiency of a test generation algorithm greatly depends on its implementation, hence, a description of the data structure used in implementing the BFP procedure is presented, then tree pruning will be discussed.

### 4.4.1 Data Structure

Consider the circuit shown in Figure 4.11. The dynamic change of the data structure during the back–propagation process for the circuit example is also shown in Figure 4.12. Each segment in Figure 4.12 corresponds to a different level of assignment. As shown in this figure, the data structure of the BFP procedure is a two dimensional array of circuit nodes. The first column in the array represent current nodes which carry fault logic values (fault entries). All other nodes retain control logic values (control entries). Starting with a node carrying a fault logic value, a horizontal move represents a sensitization path logic assignment for a node on the path. A vertical move in the control entries, if it exists, represents another set of logic values which sensitizes different path for the same entry node which carries the fault value.

Figure 4.12 shows that the circuit requires three levels of assignment before all

the stem nodes are completely assigned. After applying the *Compare*(·) procedure, it is apparent that MPS from node $b$ is not possible because of the conflict fault logic entries at nodes $b_1$ and $b_2$ (cube intersection will result in conflict of assignment). On the other hand, SPS of faults at each entry in the first column in the data structure can be extracted. For instance, at level 3 of assignments, in the first fault entry (a=$D$), the test pattern ($a$=$D$, $b$=0) is generated due to the conflict–free assignments at the stem nodes $b_1$ and $b_2$. This fault pattern along with its partial complement cover the two faults $a/0$ and $a/1$. For the second entry ($b_1$=$D$), only the fault $b_1/1$ can be covered because of the partial conflict with the logic value at node $b_2$. After comparing the logic values at the stem nodes, node $b$ will be assigned the fault value $FD$ (not shown in figure).

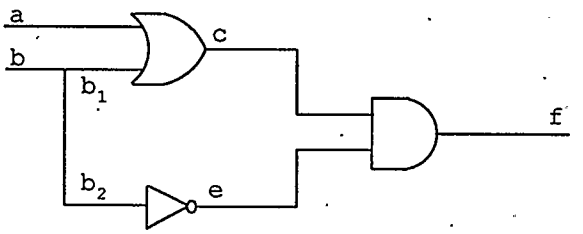



Figure 4.11. A circuit example.

It is clear that some node assignments are shared by sensitization paths for different faults. For instance, the first and second fault entries share the node assignment $b_2$=0. This is different from current ATPG approaches where test generation efforts are dedicated to single target faults. On the other hand, the GATPG algorithm allows for global test pattern generation in a shared search space environment.

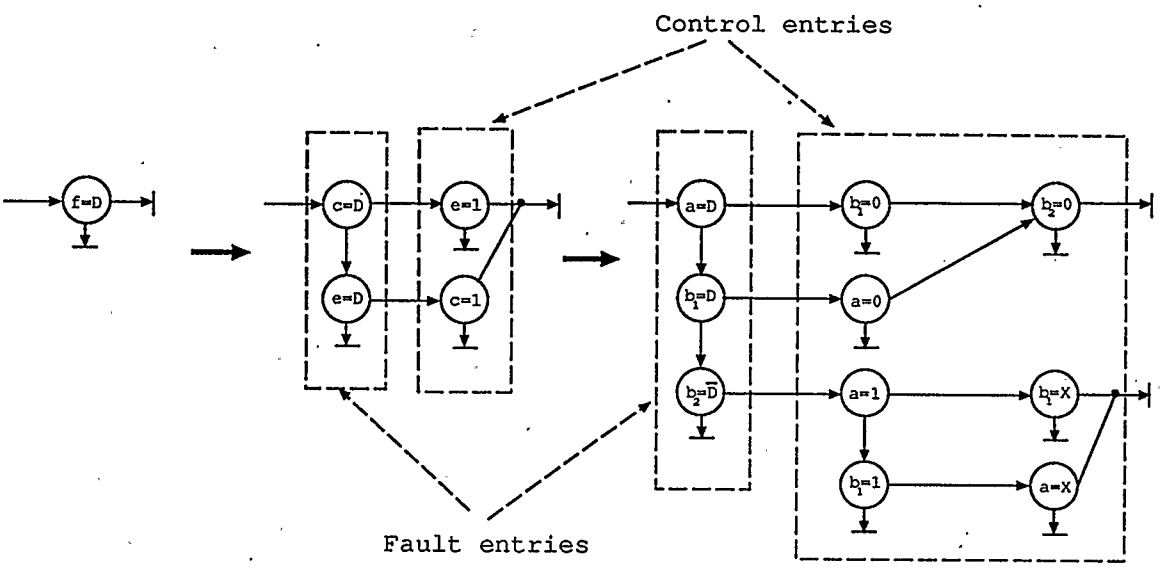



Figure 4.12. The data structure for the circuit example.

### 4.4.2 Pruning the Assignment Tree

The horizontal length (number of nodes in the horizontal direction) of the data structure used in the BFP procedure is limited by the number of nodes assigned at some level during back–propagation. The vertical dimension of the data structure, however, will explode exponentially if enumeration of logic values is allowed at all levels of assignment. For instance, consider the fault entry $b_2$=$\overline{D}$ in Figure 4.12. At level 3 of assignment, two sensitization paths are created which support the fault entry at node $b_2$. Eventually, as the number of levels of assignments increases, the number of paths associated with each fault entry will, in the worst case, increase exponentially.

A key factor in controlling the space complexity of the GATPG algorithm would be to limit the number of sensitization paths associated with each fault entry in the data structure. Therefore, the BFP procedure keeps track of the possible number of sensitization paths for each fault entry and compares it to a preset limit. If the

number of possible paths exceeds that limit, then only one pattern is allowed for each consequent back–propagation step. Otherwise, full enumeration of logic values is allowed.

The preset limit for the maximum number of sensitization paths associated with each fault in our implementation ranges from 2 to 5 depending on the circuit complexity. This limit is generally smaller than the backtrack limit used in current ATPG algorithms. The reason is that at some level of assignment, only a small portion of the search space is exposed and a small number of alternatives might be needed. If at any level, some of these alternatives are in conflict, the algorithm creates more options for the next levels of assignment by allowing full enumeration of logic values across the logic gates until the number of alternatives reaches the preset limit. In this way, the GATPG algorithm can deterministically generates tests globally without using large memory space. This approach also ensures that every testable fault will be covered while redundant faults will be singled out. A redundant fault will result in a complete conflict of assignments for all the control entries associated with it. Consequently, redundant faults nearest to the primary outputs are not processed any further by the algorithm. All paths between the primary inputs and the visited redundant faults will not be processed as well. This technique enhances the performance of our algorithm and saves test generation time.

## 4.5 Constructing the Test Primitives

Since the GATPG algorithm is described in details, an example which illustrates the procedures for test generation and the construction of the test primitives is presented. The example circuit is shown in Figure 4.13. The GATPG algorithm will be applied to the circuit example in order to generate the test primitive.

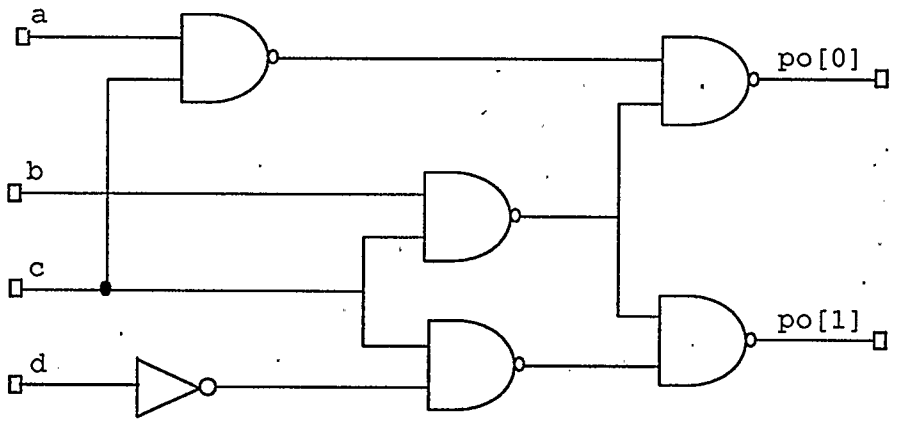
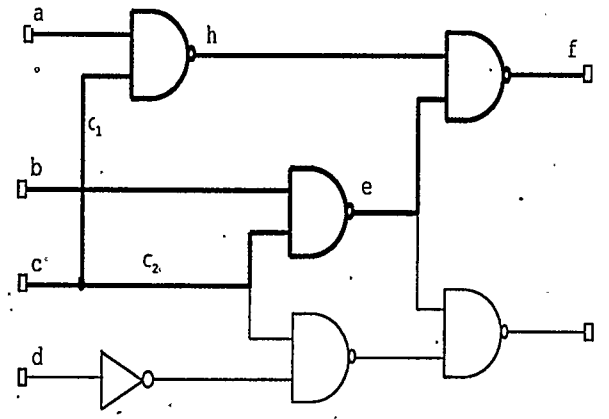



Figure 4.13. A combinational circuit example.

Figure 4.14 shows a highlight for the first output cone for the circuit and the corresponding data structure used by the GATPG algorithm. The GATPG algorithm takes three levels of assignments to completely map the fault value at the primary output to the circuit's primary inputs. For each fault entry, one sensitization path will be selected as an input vector which covers the fault entry location. A careful choice should be made to decide which values (path) to keep and which to ignore because we might be faced with different choices, as shown in Figure 4.14. The first fault entry will have only one sensitization path which is represented by the logic entries: $a$=$D$, $c_1$=1, $b$=0, and $c_2$=X. After the compare procedure is applied, the test pattern which represents this fault entry will be: D, 0, 1, X for $a$, $b$, $c$, and $d$, respectively. The first fault entry cannot propagate through the other path which carries the assignments $c_2$=0 and $b$=X, because, in this case, $c_1$ and $c_2$ are in conflict.

The second fault entry, on the other hand, may have two valid paths. The first is represented by the assignment $a$=1, $b$=0, and $c_2$=X. This path created the test pattern 1, 0, D, and X for $a$, $b$, $c$, and $d$, respectively. The other path has the assignments: $a$=1, $c_2$=0, and $b$=X. There is a partial conflicts between the assignments of the two stem nodes of the fanout point. This will result in the test pattern: 1, X, $DF$, and



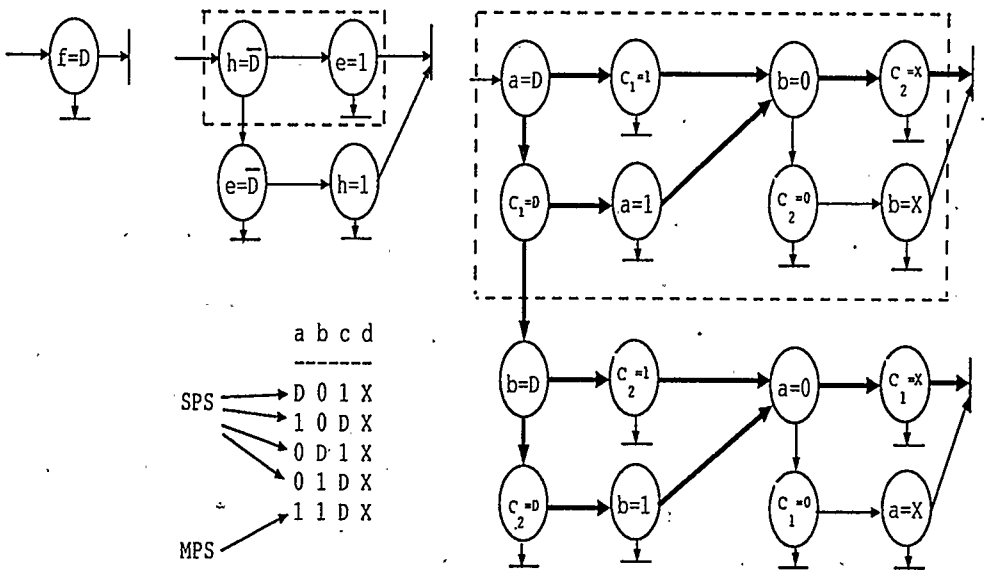




Figure 4.14. Test generation for the first output cone.

X, which implies that this test will cover only one type of stuck–at fault ($c$ s/1, for instance). The better choice is to chose the first path because it represents a cover for both type of stuck–at faults for all the nodes on that path. The bold lines in Figure 4.14 represent the different paths that are chosen by the GATPG algorithm to cover all the faults in the circuit example.

The test patterns shown in Figure 4.14 are generated by considering all the fault entries in the data structure. The first four entries in the test patterns table are the one extracted from each entry through SPS. The last entry is extracted by applying the MPS procedure. In this procedure, the sensitization paths associated with the fault entries for the two stem nodes are cube intersected. Since each entry has two possible paths, four cube intersections may be performed before a MPS pattern is generated. Currently, the ATPG algorithm halts if one MPS path is generated, otherwise continue with the cube intersection procedure.

Similar procedures are applied to the second cone of the circuit example as shown in Figure 4.15. Figure 4.16 shows the final test primitive for the module under test which corresponds to the circuit example. The first entry in the test primitive is the output cone number. This is followed by the list of generated test patterns. Following each test pattern is the logic values of the output nodes when the corresponding pattern is applied at the inputs of the circuit under test. This information is produced after the generation of each test pattern, where the test pattern is simulated (forward propagation) and evaluated at the primary outputs. In this particular example, with each output cone, one of the two primary outputs always carries an X logic value.

In this way, the test primitive totally characterizes the test behavior of the circuit. It can be seen that the test primitive can also be used to evaluate the functionality of the circuit. If the test primitive is to be used at the gate–level of testing, the test

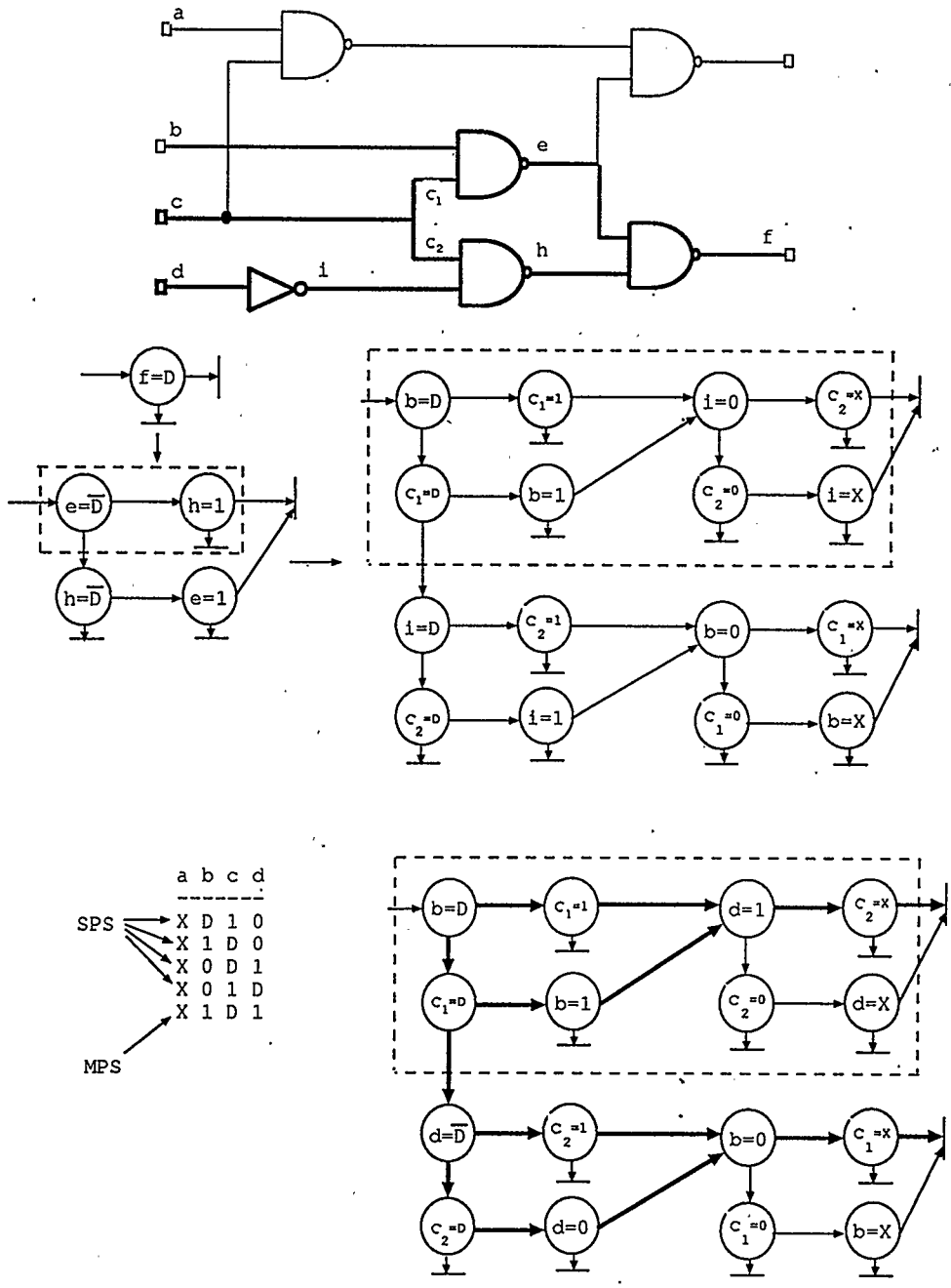
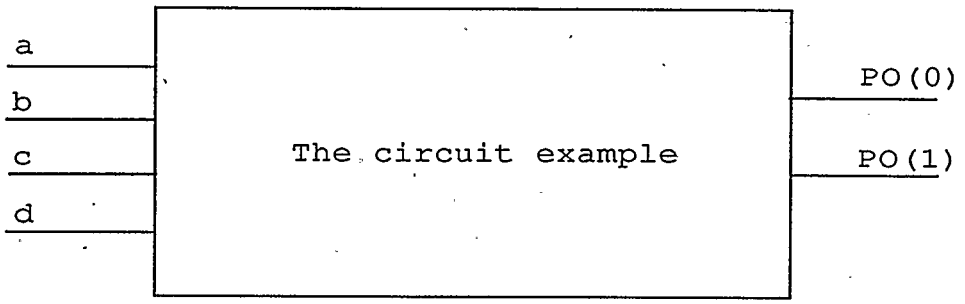



Figure 4.15. Test generation for the second output cone.




Test primitive for the circuit example.

```
Output Cone(0)
**************

abcd   PO(0)  PO(1)
D01X    D      X
10DX    D      X
0D1X    D      X
01Dx    D      X
11DX    D      X

Output Cone(1)
**************

abcd   PO(0)  PO(1)
XD10    X      D
X1D0    X      D
X0D1    X      D
X01D    X      D
X1D1    X      D
```

Figure 4.16. The generated test primitive.

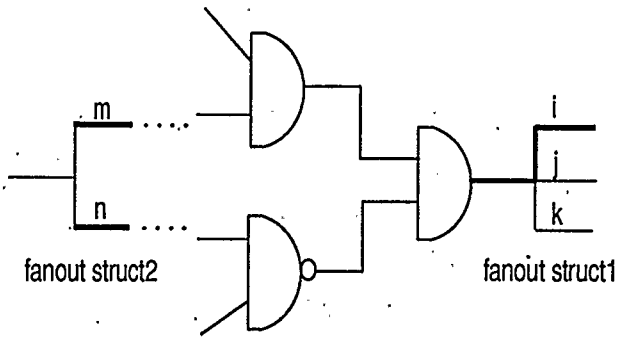



Figure 4.17. A circuit example for explaining the space complexity of the GATPG algorithm.

primitive will represent the test patterns for the circuit. If it is to be used at the modular level of testing, the test primitive will also serve as the mapping information across the module's terminals.

### 4.5.1 Algorithm Complexity

As shown in the above discussion, the GATPG algorithm minimizes the size of the logic assignment tree by allowing only the consistent logic assignments to occur at any node during the back–propagation process. It also applies pruning criteria which effectively reduces the space complexity of the test generation process. The space complexity of the GATPG algorithm is determined by the size of the data structure during the back propagation process. As discussed before, the horizontal length of the data structure is always limited by the number of nodes assigned during the back propagation process. At any time, during this process, the number of valid paths associated with each fault entry is less than a fixed number, i.e., the preset limit. Therefore, the number of fault entries in the data structure are the only nodes which are left unconstrained in order to ensure the coverage of all testable faults in a circuit.

It is interesting to notice that the number of fault entries in a data structure

depends mainly on the circuit complexity, not on the number of nodes in a circuit. This fact can be explained using the circuit example shown in Figure 4.17. In this example, during the back propagation process, any sensitization path from fanout structure 1 (struct1 in figure) must pass through the stem nodes of struct2. For instance, the bolded path at stem $i$ will be back propagated through two single path sensitizations; one time through node $m$ while the other one through node $n$. Of course, it is assumed that there are no more fanout structures in the path between struct1 and struct2 in the figure. Accordingly, the three paths at the first fanout structure will be translated into six single path sensitizations coming out of the second fanout structure. In general, a circuit with a maximum of $N$ consecutive fanout structures in a path, with each structure $i$ on the path having an $m(i)$ stem nodes, the expected maximum number of fault entries will be:

$$m(0) * m(1) * .... * m(N-1).$$

This number can be huge for large circuits. Fortunately, in our algorithm, we only deal with output cones instead of the whole circuit. This technique drastically decreases the number of fault entries in the data structure and inherently partitions the circuit under test. The above space complexity figure explains why the experimental results in the next section show a performance which does not reflect the size of the circuit under test, for instance, large circuits may take less test generation time than smaller ones.

The time complexity of the BFP algorithm is mainly determined by the time taken during the *Compare*( ) procedure. If a fanout structure has an $m$ number of stems, the compare procedure time complexity is $O(m)$. As mentioned before, the multiple path sensitization procedure has a linear time complexity with the number of stems

in a fanout structure, i.e., $O(m)$. Also, in the worst case, if the algorithm has to perform the compare procedure for each sensitized path in the circuit, then the time complexity will be:

$O(m)$ * Number of fault entries.

Therefore, the time complexity, once more, reflects the circuit complexity; not the circuit size. All the operations involved in building the data structure and the compare procedure are very simple and do not include any decision making procedures. Therefore, they are fast and efficient, which resulted in the superior time performance of our algorithm.

## 4.6 Experimental Results

Table 4.1. Real execution performance of our algorithm in a two-phase implementation on a SUN SPARC 2 workstation with the ISCAS'85 benchmark combinational logic circuits. Time units: seconds.

| Circuits | RTPG | | DTPG | | | Fault | # Test |
|---|---|---|---|---|---|---|---|
| | faults | CPU | Dfault | Rfault | CPU | coverage | vectors |
| C432 | 495 | 0.7 | 20 | 4 | 0.4 | 100% | 77 |
| C499 | 741 | 1.5 | 9 | 8 | 1.1 | 100% | 69 |
| C880 | 911 | 2.1 | 31 | 0 | 2.1 | 100% | 141 |
| C1355 | 1542 | 5.2 | 24 | 8 | 1.1 | 100% | 102 |
| C1908 | 1851 | 7.8 | 21 | 7 | 7.0 | 100% | 98 |
| C2670† | 2120 | 11.5 | 360 | 115 | 19.9 | 100% | 344 |
| C3540† | 3271 | 12.8 | 26 | 131 | 21.8 | 100% | 266 |
| C5315† | 5271 | 25.6 | 20 | 59 | 33.1 | 100% | 322 |
| C6288† | 7710 | 41.1 | 0 | 34 | 0.4 | 100% | 75 |
| C7552† | 7011 | 52.0 | 406 | 131 | 36.8 | 100% | 209 |

The GATPG test generator and a fault simulator were implemented in C on a SUN SPARC 2 workstation. The ISCAS'85 and '89 benchmarks were used to evaluate the performance of the test generation algorithm. We generate test vectors for the

combinational circuitry for the ISCAS'89 benchmark by breaking the loops connecting to the flip flops into primary inputs and outputs. The GATPG algorithm is used to generate tests for each circuit, then, the generated patterns in the test primitives are used to fault simulate the modeled faults reported in the benchmarks. We are reporting results for two system implementations of the GATPG algorithm. In the first implementation, we have purposely included a random test generation phase as a front end to the GATPG algorithm. In the second implementation, the GATPG is used as a single phase test algorithm without the random phase. In this way, we can fully evaluate the potential of the GATPG algorithm.

### 4.6.1 Two Phase Implementation

In the first implementation, the random test generation is first applied and is followed by a deterministic test generation based on the GATPG algorithm. Results using the ISCAS'85 and ISCAS'89 benchmark circuits obtained on a SUN SPARC 2 workstation are summarized in Tables 4.1 and 4.2. We require only the test generation for the combinational logic in the ISCAS'89 benchmark. The results obtained from random and deterministic test generation are shown in the two columns labeled RTPG and GATPG, respectively. Under the RTPG column, the label *fault* refers to the number of faults detected by the random generation of vectors, while CPU refers to the CPU time spent in random test generation. As a common practice, we generate random test vectors until a simulation run of 32 vectors does not detect any additional fault. For circuits marked with †, the random phase is continued until a 128 vector simulation runs did not detect any additional faults. The random test generation and simulation phase covers more than 90% of the faults in most circuits. In some cases like C6288, the random test generation phase covers 100% of the testable faults.

Table 4.2. Real execution performance of our algorithm in a two–phase implementation on a SUN SPARC 2 workstation with the ISCAS'89 benchmark combinational logic circuits.

| Circuits | RTPG | | DTPG | | | Fault | # Test |
|---|---|---|---|---|---|---|---|
| | faults | CPU | Dfault | Rfault | CPU | coverage | vectors |
| S27 | 32 | 0 | 0 | 0 | 0 | 100% | 18 |
| S208 | 198 | 0.3 | 17 | 0 | 2.3 | 100% | 50 |
| S298 | 308 | 0.3 | 0 | 0 | 0.1 | 100% | 59 |
| S344 | 332 | 0.4 | 10 | 0 | 1.8 | 100% | 52 |
| S349 | 343 | 0.4 | 5 | 2 | 0.8 | 100% | 41 |
| S382 | 392 | 0.4 | 7 | 0 | 0.8 | 100% | 47 |
| S386 | 370 | 0.5 | 14 | 0 | 1.2 | 100% | 78 |
| S400 | 411 | 0.4 | 7 | 6 | 0.6 | 100% | 63 |
| S420 | 401 | 0.8 | 29 | 0 | 1.4 | 100% | 100 |
| S444 | 460 | 0.6 | 0 | 14 | 0.2 | 100% | 46 |
| S510 | 564 | 0.9 | 0 | 0 | 0.2 | 100% | 66 |
| S526 | 530 | 0.7 | 24 | 1 | 1.0 | 100% | 113 |
| S641 | 457 | 0.9 | 10 | 0 | 2.2 | 100% | 120 |
| S713 | 532 | 1.6 | 11 | 38 | 2.9 | 100% | 141 |
| S820 | 790 | 3.9 | 60 | 0 | 1.6 | 100% | 159 |
| S832 | 812 | 3.2 | 44 | 14 | 2.3 | 100% | 177 |
| S838 | 709 | 2.7 | 148 | 0 | 16.4 | 100% | 181 |
| S953 | 974 | 4.1 | 105 | 0 | 2.8 | 100% | 149 |
| S1196 | 1120 | 4.2 | 122 | 0 | 10.9 | 100% | 196 |
| S1238 | 1229 | 4.8 | 47 | 69 | 24.1 | 100% | 260 |
| S1423 | 1442 | 4.3 | 59 | 14 | 8.2 | 100% | 151 |
| S1488 | 1482 | 4.6 | 5 | 0 | 0.6 | 100% | 230 |
| S1494 | 1476 | 4.2 | 18 | 12 | 2.0 | 100% | 221 |
| S5378 | 4383 | 51.6 | 180 | 40 | 75.5 | 100% | 392 |
| S9234† | 5781 | 142.4 | 694 | 452 | 189.6 | 100% | 790 |
| S13207† | 8452 | 212.8 | 1210 | 151 | 123.0 | 100% | 811 |
| S15850† | 10321 | 246.6 | 1015 | 389 | 274.3 | 100% | 755 |
| S35932† | 35077 | 1778.5 | 33 | 3984 | 81.1 | 100% | 214 |
| S38417† | 29198 | 1694.4 | 1817 | 165 | 375.4 | 100% | 1817 |
| S38584† | 34211 | 1516.9 | 587 | 1506 | 107.7 | 100% | 1330 |

In the GATPG phase, we have used a similar systematic approach to the one used in random test generation. For each primary output in a circuit, we generate the test patterns until a simulation run of 32 (128 for circuits marked with †) test patterns does not detect any additional fault in the output cone under test. This procedure is repeated until all output cones are tested. For each primary output in a circuit, we generate the test patterns with a preset limit of two. A preset limit of five is used for large circuits (marked with †). If any of the two fault logic values $D$ or $\overline{D}$ appears in the input fault pattern, both types of stuck–at faults are considered during simulation. If, on the other hand, one of the other four fault values exists, only the corresponding type of stuck–at fault is considered by the simulator.

Under the GATPG column, the label *Dfault* refers to the detected faults by the GATPG algorithm. If a fault is not detected by RTPG or GATPG, it is considered a redundant fault as the label *Rfault* indicates. The CPU refers to the time taken to generate the test patterns.

The final fault coverage is indicated under the column labeled *Coverage*. It represents the percentage of faults covered by the test system including both RTPG and GATPG. Finally, the column under *Vectors* is the total number of vectors generated by the test system.

As shown in the tables, the two phase test system covers all the testable faults in the benchmarks. The random phase covers at least 90% of the modeled faults in any circuit. As the circuit size increases, the random phase takes a large amount of the CPU time in order to reach the 90% fault coverage margin set by the test system. The GATPG algorithm, on the other hand, takes a very small CPU time, even when considering large size circuits.

### 4.6.2 Single Phase Implementation

In the single phase system implementation for the GATPG algorithm, we have considered the fact that the GATPG algorithm CPU run time is small and it can be used as a single phase system. In order to achieve this purpose, we have lifted the constraint on the number of generated patterns that when generated do not cover any other fault in the fault list of the circuit under test. This limit was 32 for small circuits and 128 for large circuits. Instead, we simply allowed the GATPG algorithm to generate all possible test patterns. The results of this implementation is shown in Tables 4.3 and 4.4.

It can be seen from these two tables that the single phase GATPG system also cover 100% of the testable faults in the benchmarks. The CPU run time has increased slightly from the results in the two phase system. The overall run time is greatly reduced compared to the two phase system. The only noticeable difference between the two implementations is the resultant number of test vectors. As expected, the test length has increased and in some cases is doubled. This is due to the lack of constraint on the number of generated test patterns in the single phase system.

Table 4.5 shows a performance comparison between our algorithm and the Transitive Closure (TC) algorithm [12]. We have chosen large circuit examples in this comparison where computational complexity is critical. In these particular examples, the TC algorithm uses a large number of backtracks during its execution. The data under the GATPG and TC columns show the average time taken by the GATPG algorithm and by the transitive closure algorithm, respectively, to cover a single fault in the corresponding circuit. Each entry under TC is calculated by dividing the time taken in the GATPG phase by the number of faults in the fault list after the RTPG

Table 4.3. Real execution performance of our algorithm in a single-phase implementation on a SUN SPARC 2 workstation with the ISCAS'85 benchmark combinational logic circuits. Time units: seconds.

| Circuits | BFP | | | Fault | # Test |
|---|---|---|---|---|---|
| | Dfault | Rfault | CPU | coverage | vectors |
| C432 | 515 | 4 | 0.5 | 100% | 91 |
| C499 | 750 | 8 | 1.2 | 100% | 89 |
| C880 | 942 | 0 | 2.2 | 100% | 188 |
| C1355 | 1566 | 8 | 1.2 | 100% | 192 |
| C1908 | 1872 | 7 | 7.1 | 100% | 162 |
| C2670† | 2480 | 115 | 29.9 | 100% | 393 |
| C3540† | 3297 | 131 | 27.8 | 100% | 324 |
| C5315† | 5291 | 59 | 39.1 | 100% | 429 |
| C6288† | 7710 | 34 | 10.4 | 100% | 145 |
| C7552† | 7417 | 131 | 45.3 | 100% | 511 |

phase. For the single phase GATPG system, each entry under GATPG is calculated by dividing the test generation time by the total number of faults covered by the algorithm. The Table also shows the resultant speed–up (SU) factor in each example. The Speed–up factor depends on the circuit structure but in general our approach is faster than the TC algorithm. For the other circuits in the ISCAS benchmarks, the GATPG algorithm shows a comparable performance to the TC algorithm.

Generally, for small and medium size circuits, the GATPG algorithm shows a comparable performance with best known algorithms. For large and complex circuits, the GATPG algorithm performs exceptionally well and outperforms other algorithms. This is due to the global search approach, the early detection and removal of inconsistencies, and by the use of efficient pruning techniques for the assignment tree. The results also show that although our algorithm is single phase, the number of generated patterns is slightly larger (but comparable) to those generated by other algorithms. Finally, the GATPG algorithm always achieves a 100% fault coverage.

Table 4.4. Real execution performance of our algorithm in a single-phase implementation on a SUN SPARC 2 workstation with the ISCAS'89 benchmark combinational logic circuits.

| Circuits | BFP | | | Fault | # Test |
|---|---|---|---|---|---|
| | Dfault | Rfault | CPU | coverage | vectors |
| S27 | 32 | 0 | 0.1 | 100% | 13 |
| S208 | 215 | 0 | 2.9 | 100% | 52 |
| S298 | 308 | 0 | 3.1 | 100% | 79 |
| S344 | 342 | 0 | 3.2 | 100% | 71 |
| S349 | 348 | 2 | 1.8 | 100% | 62 |
| S382 | 399 | 0 | 2.1 | 100% | 63 |
| S386 | 384 | 0 | 2.4 | 100% | 81 |
| S400 | 410 | 6 | 2.6 | 100% | 77 |
| S420 | 430 | 0 | 3.4 | 100% | 109 |
| S444 | 460 | 14 | 3.2 | 100% | 45 |
| S510 | 564 | 0 | 4.2 | 100% | 86 |
| S526 | 554 | 1 | 4.0 | 100% | 127 |
| S641 | 467 | 0 | 3.0 | 100% | 117 |
| S713 | 543 | 38 | 4.9 | 100% | 177 |
| S820 | 850 | 0 | 4.6 | 100% | 181 |
| S832 | 856 | 14 | 4.3 | 100% | 169 |
| S838 | 857 | 0 | 17.9 | 100% | 202 |
| S953 | 1079 | 0 | 6.8 | 100% | 169 |
| S1196 | 1242 | 0 | 19.1 | 100% | 221 |
| S1238 | 1276 | 69 | 26.6 | 100% | 299 |
| S1423 | 1501 | 14 | 8.9 | 100% | 161 |
| S1488 | 1487 | 0 | 9.6 | 100% | 255 |
| S1494 | 1494 | 12 | 6.0 | 100% | 241 |
| S5378 | 4563 | 40 | 81.5 | 100% | 444 |
| S9234† | 6475 | 452 | 190.0 | 100% | 812 |
| S13207† | 9662 | 151 | 144.1 | 100% | 807 |
| S15850† | 11336 | 389 | 288.1 | 100% | 723 |
| S35932† | 35110 | 3984 | 96.0 | 100% | 394 |
| S38417† | 31015 | 165 | 411.0 | 100% | 2501 |
| S38584† | 34798 | 1506 | 120.5 | 100% | 2110 |

Table 4.5. Performance comparison between the BFP algorithm and the Transitive Closure (TC) algorithm on a SUN SPARC 2 workstation for large ISCAS benchmark circuits. Time unit: seconds.

| Circuits | GATPG | TC | SU |
|---|---|---|---|
| C2670 | 0.05 | 0.19 | 40 |
| C7552 | 0.09 | 0.47 | 47 |
| S9234 | 0.27 | 0.52 | 2 |
| S13207 | 0.008 | 0.3 | 36 |
| S38417 | 0.2 | 0.78 | 3.9 |
| S38584 | 0.17 | 0.29 | 1.6 |

## 4.7 Practicality of the GATPG Algorithm

Since the GATPG is presented, its space and time complexities are explored, and experimental results were reviewed, it is useful to give some remarks about the practicality of the GATPG. The way VLSI circuits are designed today dictates that VLSI test generation algorithms must be simple, fast, and efficient in order to keep on with the pace of increasing design complexities. We believe that the GATPG algorithm provides these requirements. Simplicity is manifested in the simple test procedures employed in our algorithm. These procedures are not based on any decision making processes. This fact made it possible to direct our attention to the implementation issues, rather than being worried about the test quality of the algorithm. The test quality is always guaranteed.

The fast performance of the GATPG algorithm has been attained because of the same factors that contributed to the simplicity of the algorithm. It is known that decision making–based algorithms suffer from time consuming procedures which, although dealing with a single target fault, try different choices (sometimes based on preprocessing setps) and then check the validity of these choices. Our approach, on

the other hand, is based on a none decision making procedures to minimize the test generation time.

This discussion brings the efficiency issue into consideration. If the efficiency of a test system is defined as the ability of the test system to cover all faults in a circuit (regardless of test generation time), then all test approaches are efficient. If, on the other hand, efficiency is defined as the ability of the test algorithm to cope with different circuit complexities and generate tests in a reasonable amount of time, the GATPG will emerge as the most efficient. This has been proved by the ability of the GATPG algorithm to generate tests for large circuits, without random test vectors. The space and time complexities of the GATPG algorithm show that the test generation time is highly dependent on how complex the circuit is. But, as we stated earlier, as long as the test procedures are simple and efficient, the circuit complexity can always be contained in the context of the GATPG algorithm. This fact renders the GATPG algorithm as one of the most practical ATPG algorithms for the ever increasing complexity of VLSI circuits.

## 4.8 Summary

The present test generation algorithms can generate test vectors for complex combinational circuits and guarantee 100 percent coverage for all the testable faults in a circuit. However, in the worst case, the test generation time increases exponentially with the circuit complexity. A new approach which combines simplicity and efficiency has been developed for the test generation of large combinational circuits. This approach is based on tracing back a fault at a primary output node in order to generate test patterns which sensitize all paths between the primary inputs and the primary output node. This system is referred to as global testing or a non–target

fault system.

A two phase and a single phase GATPG algorithm implementations have been implemented for generating input fault patterns which sensitize all paths between primary input and output nodes. The algorithm creates a tree of logic assignments by back–propagating a fault at a primary output node. During its execution, the algorithm uses uniquely implied logic values across the input and output nodes of the logic gates. A key feature of the GATPG algorithm is that conflicts are detected and removed incrementally from the logic assignment tree during the back–propagation process. The results on large circuits suggest that our algorithm outperforms other test generation algorithms in terms of computing time.

# CHAPTER 5

# THE GENERATION OF TEST PATTERNS WITH MAXIMAL MULTIPLE FAULT COVERAGE

The new framework which is proposed in previous chapters for solving the problem of test generation suggests that it would be worthwhile revisiting other related test problems and issues in its context. Some of these problems are extremely difficult and have been put aside by test generation researchers. One of these problems is the multiple fault test generation, in which the assumption that only one fault exists in a circuit at the time of testing does not hold any more. Instead, it is assumed that more than one physical failure has occurred in the circuit during fabrication, which results in the existence of more than one stuck–at fault in a circuit.

As far as the problem of generating tests with maximal multiple fault coverage for the general class of combinational circuits is concerned, no algorithm exists in the literature which generates such tests. In this chapter, we will extend the domain of the GATPG system by showing how it can generate tests with maximal multiple fault coverage without adding any new procedures to our test system. We will first present our analysis on the multiple faults behavior. This analysis will then be used in improving the multiple fault coverage for any test set. We have taken this step to show that this analysis can be applied to any existing ATPG system; a fact which helps in building a better test system integration that uses different test models for

different applications. Then, we will show how can we apply the results of this analysis to our GATPG system. The core of this chapter will be presented in ASIC'95 [59].

## 5.1 Introduction

Multiple fault detection has been discussed many times [22, 8, 16, 42]. The major complexity of the problem lies in the number of multiple faults that may exist in a circuit. Most Automatic Test Pattern Generation (ATPG) algorithms guarantee a 100% fault coverage for single stuck–at faults in a circuit. Single fault test sets have been used in simulating multiple faults in a circuit. The multiple fault coverage varies among the different test sets for the same circuit. This is due to the different heuristics employed in different ATPG systems. Presently researchers have avoided the direct test generation for multiple faults because of the extremely large number of multiple fault combinations that exist in a circuit.

In this chapter, the multiple fault testing problem is analyzed in the context of path sensitization algorithms such as PODEM, FAN, and SOCRATES. We specifically examine the self–masking faults because they represent most of the undetectable faults using single fault test sets. We examine the strategies that are generally used in path sensitization algorithms for single fault test generation. These strategies may lead to incomplete cover of some multiple faults in a circuit. A procedure which can be used during single fault test generation will then be presented. It implicitly guarantees maximum multiple fault coverage for a given set of test vectors. This procedure first partitions the set of primary inputs in a circuit into three subsets, namely, control, excitation, and persistency sets of primary inputs. Then, the problem of multiple fault testing can be formulated as a problem of optimizing these subsets so as to achieve maximal multiple fault coverage for each test. Based on the analysis given in this chapter, two different procedures for augmenting any single fault test set will be

presented. We have tested these procedures on the 74LS181 ALU circuit using twelve test sets generated by different methods.

This chapter is organized as follows. In the next section, we will briefly review the previous work. In Section 5.3, some preliminary discussion on the multiple fault testing problem is presented. The analysis of multiple faults behavior in the context of path sensitization algorithms for single fault test generation is presented in Section 5.4. In Section 5.5, we present two models for the analysis and test set augmentation for multiple faults. In Section 5.6, two procedures based on the analysis of Section 5.4 are given. Experimental results on the 74LS181 ALU is given in Section 5.7. In Section 5.8, we will show how can we modify the GATPG algorithm to generate test patterns with implicit maximal multiple fault coverage.

## 5.2 Previous Work

There have been several works discussing the problem of multiple stuck–at fault testing. Some researchers approach the fanout–free circuits [30, 19, 7, 29]. Most of the work done so far on multiple fault testing emphasized on the issues of reducing the number of faults considered during test generation and generating test sets without explicitly considering all combinations of multiple faults [10, 24, 45, 50]. Jone and Madden [29] developed an algorithm which generates tests for single stuck–at faults. It guarantees the detection of all multiple stuck–at faults in fanout–free circuits. Their test sets were shown to be minimal in size. Instead of generating test patterns for multiple faults, the work of Agarwal and Fung [1] investigated the multiple fault coverage of single fault tests. Results on fanout–free circuits containing gates with fan–in of two cover all multiple faults of size two and three, at least 99% multiple faults of sizes four and five, and at least 98.5% multiple faults of size six. GEMINI, proposed

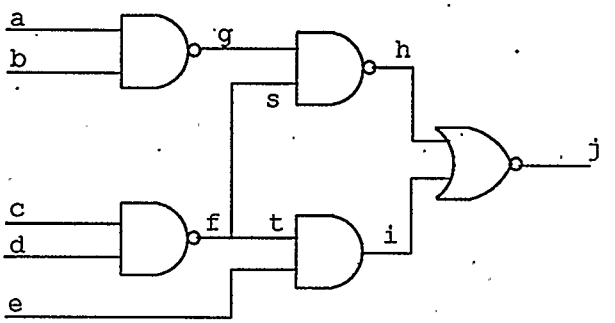



Figure 5.1. An example to illustrate testing terminology.

by Cox and Rajski [13], is a multiple fault test generation system for general circuits. GEMINI allows multiple faults of all multiplicities to be considered implicitly.

The work of Kubiak and Fuchs [32] indicated that multiple faults simulation is a time–consuming process. As the number of gates in a circuit increases, it is not practically acceptable to fault simulate single fault test sets against multiple fault occurrences in the circuit. All present approaches handling the multiple fault testing problem face a higher order of time and space complexities.

To overcome these difficulties, procedures used during single fault test generation which implicitly guarantee a maximum multiple fault coverage for the generated test set must be employed. Another approach for solving the multiple fault test generation problem is to augment the generated test sets from single fault ATPG systems.

## 5.3 Preliminaries

In this section, some common terminologies pertaining to multiple fault test generation are introduced with an example.

**Definition 4** *A convergence point (CP) is a node at which two or more faults , i.e.* $\{f_1, f_2, \ldots, f_m\}$, *interact. These faults may be faults in the CP or have propagated*

*from some other node to the CP.*

**Definition 5** *When multiple faults meet at a CP, the fault or faults which can propagate through CP for some test will be denoted as dominant faults for that node.*

For example, consider the double faults $\{t/0, g/1\}$ in Figure 5.1. The test vector "11010" (for inputs $a$, $b$, $c$, $d$, and $e$, respectively) covers these two faults. The CP point for $\{t/0, g/1\}$ is node $j$. The fault $g/1$ *dominates* the fault $t/0$ because the later is masked by the input $e$.

**Definition 6** *A set of faults $F = \{f_1, f_2, \ldots, f_m\}$ is said to be self-masking under a test set $T = \{t_1, t_2, \ldots, t_n\}$ if and only if:*

*1) For each $t_i$, $i = 1, 2, \ldots, n$, $t_i$ detects each single fault $f_j$, $j = 1, 2, \ldots, m$.*

*2) For each $t_i$, $i = 1, 2, \ldots, n$, $t_i$ does not detect the multiple fault $f_1, f_2, \ldots, f_m$. Multiple faults composed of fewer than $m$ elements of $F$ may or may not be detected by the test set $T$.*

If the test set $T$ contains only a single test vector, the relationship will be referred to as single vector self-masking; if $T$ contains more than one vector, then multiple vector self-masking. For example, the double faults $\{a/1, e/1\}$ in Figure 5.1, are single vector self-masking under the test $T$ = "01000".

## 5.4 Multiple Faults Analysis

In order to detect a set of multiple faults, it is only necessary that, with all faults excited properly, at least one fault from this set must propagate to the primary outputs. The following question can now be asked concerning multiple faults detection.

Given two single stuck–at faults $f_1$ and $f_2$ which are self–masking under a test $t$, is there any way where the test vector $t$ can be modified so that it might cover the set of multiple faults $\{f_1, f_2\}$?

Let us assume that $t$ sensitizes the single fault $f_1$ through a sensitization path $s_1$ and sensitizes $f_2$ through $s_2$. The only condition which makes $t$ unable to cover the multiple faults $\{f_1, f_2\}$ is that the sensitization structures of $s_1$ and $s_2$ are destroyed due to the existence of $f_2$ and $f_1$ simultaneously. For example, in the circuit shown in Figure 5.2, the test vector "1001" covers the single faults $b/1$ and $c/1$. In the existence of the double faults $\{b/1, c/1\}$, the sensitization paths of $b/1$ and $c/1$ mask each other at the inputs of gates $G_3$ and $G_4$. The problem of covering a set of multiple faults can be solved if there exists a test which does not destroy the sensitization structure of at least one fault in a fault set. To probe more on this issue, we present the following definitions:

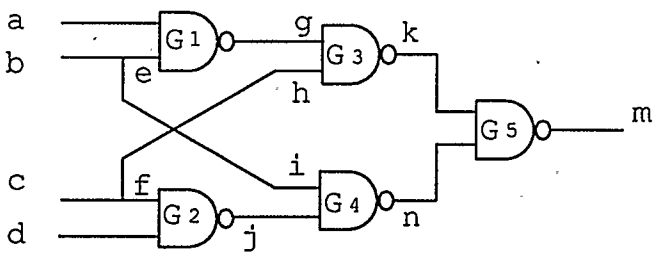



Figure 5.2. Example for the different PI sets of a fault under test.

**Definition 7** *A sensitization path is referred to as persistent under a set of primary inputs $S_p$ if it is not destroyed with the variations in the logic values of the primary inputs in the $S_p$ set. $S_p$ is referred to as the persistency set.*

The following definition further divides the set of primary inputs into an excitation

set and a control set.

**Definition 8** *The set of primary inputs which controls the excitation logic value at a fault location is referred to as the excitation set $S_e$. Any primary input which does not belong to $S_p$ or $S_e$ is referred to as a controlling primary input. The set which comprises all controlling primary inputs is referred to as the control set $S_c$.*

In the example shown in Figure 5.2, the fault $b/1$ is covered by the test vector "1001", then: $S_p = \{a, d\}$, $S_e = \{b\}$, and $S_c = \{c\}$. The logic values for the entries in $S_p$ do not affect the sensitization structure of the fault $b/1$; entries in $S_e$ excites the fault; while the entries in $S_c$ guides the sensitization path to its destination at the primary output. Each fault covered by a test vector in a test set has unique sets of primary inputs. The contents of each set depends on the way test generation is carried out. The primary input sets can be determined during the test generation process or during fault simulation. A simple procedure for identifying the different sets of primary inputs for a fault under test using a single fault simulator is shown in Figure 5.3.

In the next section, we will use the above sets in analyzing and improving the multiple fault coverage of single fault test sets.

## 5.5 Two Models for Test Set Augmentation

The main objective in single fault test generation algorithms is to generate tests that cover all the testable faults in a circuit in a reasonable amount of time. Many efforts are used to find a sensitization path for a single target fault with the minimum number of logic assignments to the circuit's nodes. Although this has proven to be helpful in reducing the test generation time, tests generated using this method does

```
Input  : A single fault f and its cover t.
Output : The PI sets for f under t.

Procedure Identify(S_e, S_c, S_p) {
   for each primary input PI_i
      {
   Complement(PI_i) ;
   Simulate(PI_i) against f;
   if f is not excited
     PI_i ∈ S_e;
   else if f is excited but not covered
      PI_i ∈ S_c;
   else PI_i ∈ S_p
   Complement(PI_i) ;
      }
}
```

Figure 5.3. **Identify( ):** a procedure used to determine the different primary input sets for fault $f$ under $t$.

not guarantee to cover 100% of multiple faults (especially for circuits with many internal fanout nodes). In order to account for multiple faults during single fault test generation, the relationship between a fault sensitization path and the other nodes in a circuit must be taken into account. The relationship between a sensitizing path and the different nodes in a circuit is encoded in the generated test vector. Each generated test can be analyzed to determine such relation. The different sets of primary inputs presented in the previous section can be used for this purpose.

Let $T = \{t_1, t_2, .., t_n\}$ be the set of single fault tests which was generated to cover the single target faults $f_1$, $f_2$, .., and $f_n$, respectively. In order to obtain a maximum multiple fault coverage from a single fault test set, it is necessary that each vector $t_i$ covers a maximum number of multiple faults which represent any combinations of $f_i$ and the other faults in a circuit. This can be achieved by modifying the test $t_i$ so as to minimize the number of nodes which, when changing their logic values, destroy the sensitization structure of $f_i$. The problem of augmenting single fault test sets to achieve maximal multiple fault coverage can be formulated as the one of maximizing the number of primary inputs in the control set of each cover in a test set. This is also equivalent to the problem of minimizing the number of primary inputs in the persistency set.

The process of maximizing the control set of a test vector requires that some elements be removed from the $S_p$ set and be added to the $S_c$ set. These elements are identified as primary inputs with a potential to create stronger sensitization structure for the fault under test. Maximizing the control set in each vector increases the sensitization path persistency against the variation of logic values at some nodes in a circuit due to the existence of multiple faults. This is obviously in conflict with some of the strategies used in single fault test generation where a minimum number

of logic assignments during the search process is always used in order to reduce the search time. Therefore, test augmentation must be applied after the single fault test generation phase in order to ensure maximal multiple fault coverage.

## 5.6 Two Procedures for Test Set Augmentation

It has been shown that most undetected multiple faults are self–masking faults. A procedure for identifying the potentially self–masking faults in a circuit can be found in [27]. Test set augmentation can be applied on the resultant self–masking faults. In this way, test augmentation can be applied without the extensive use of multiple fault simulation.

In the next subsections, we give two new procedures for augmenting single fault test sets for obtaining maximum multiple fault coverage. The first procedure aims at maximizing the control set of a single fault under a test vector without considering any multiple faults existence. This method can be directly implemented during the single fault test generation phase. A second procedure which can only be applied on a set of uncovered multiple faults is also presented. This procedure aims at changing the dominance relation between the different faults in a multiple fault set. During this procedure, the sensitization structures of a subset of faults in a multiple fault set are destroyed while allowing other fault sensitization path(s) to terminate at the primary output nodes, hence, covering the set of multiple faults.

#### 5.6.0.1 The Maximum Control Set Procedure

The first procedure for augmenting a single fault test set is based on the maximization of the control set of each test vector in the test set. Consider the circuit shown in Figure 5.2, The test "1001" excites both faults $b/1$ and $c/1$ but it fails to

propagate the effect of any of these faults to the primary output. This behavior occurs because the two faults mask each other at the inputs of gates $G_3$ and $G_4$. Note that the primary input $c$ is the only element in the $S_c$ set for the fault $b/1$ under the test "1001". A procedure is needed to search the $S_p$ set of $b/1$ so that a new input might exist which if combined with $c$ creates a stronger sensitization structure for $b/1$. In order to achieve this, the *Max_Control( )* procedure, shown in Figure 5.4, is applied. This procedure can be applied on each single target fault and its cover until a 100% multiple fault coverage is achieved or the single fault test set is exhausted.

The *Max_Control( )* procedure starts by identifying different sets of primary inputs for a single stuck–at fault $f$ under a test $t$. The circuit is then simulated against each single logic variation in the primary inputs which belongs to the persistency set. If the logic implication of this variation meets some control objectives set by the original test vector, then the corresponding primary input from the persistency set is moved to the control set of the same test vector. The value assigned to this particular input is the one which, if implied, meets the control objectives. The control objectives are determined by simulating the primary inputs which belongs to the original control set, i.e., simulating $c = 0$ in the above example. In Figure 5.5, the *Max_Control( )* is applied to the fault $b/1$. The control objectives set by the test vector "1001" for covering the fault $b/1$ are $k = j = 1$ as shown in Figure 5.5.

As shown in Figure 5.5, the implication of the logic variation at node $a$ does not provide any additional control support to the sensitized path and does not meet any of the objectives set by the original test. On the other hand, changing the logic value of node $d$ to 0 adds more control to the sensitizing path and meets the control objective of setting the logic value of node $j$ to 1. Hence, node $d$ can now be moved from $S_p$ to $S_c$ with its new logic value. The modified test vector will now be "1000". This test

```
Input  : A single fault f and its cover t.
Output : A modified cover with maximal control set.

Procedure Max_Control( ) {
   Identify (S_e, S_c, S_p);
   Fault_Simulate(S_c);
   Set the objectives;
      for each PI_i ∈ S_p
         {
         Simulate the logic variation in PI_i;
            if the implication meets the objectives
               Move PI_i from S_p to S_c;
            else continue;
      }
      Return(S_c);
}
```

Figure 5.4. **Max_Control:** the procedure used to determine a maximal control set for a single fault $f$ under a test $t$.

Input : The single fault $b/1$ under "1001".
Output : A modified cover with maximal control set.

Procedure **Max_Control( )** {
1) **Identify** ($S_e$, $S_c$, $S_p$);
$S_e$={$b$}, $S_c$={$c$}, $S_p$={$a$, $d$};
2) **Fault_Simulate**($c$=0);
The objectives are $k$=$j$=1;
3) **Simulate** the logic variations in $S_p$;
Input $a$ changes from 1 to 0: Objectives met? NO;
Input $d$ changes from 1 to 0: Objectives met? YES ($j$=1);
4) **Determine** the maximal ($S_c$);
$S_c$={$c$, $d$} and the new test is "1000";
}

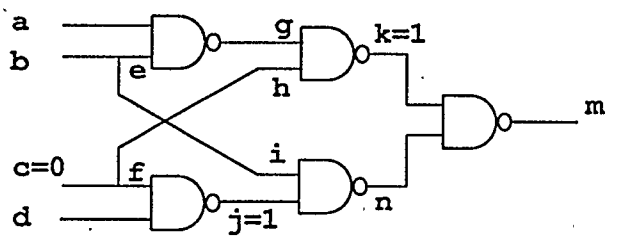



Figure 5.5. An example to illustrate the Max_Control( ) procedure.

vector has a maximal control set under the fault $b/1$ and has a stronger sensitization structure compared to the original test. It also covers the double faults $\{b/1, c/1\}$. It must be noted that the modified test vector excites both faults but only allows the fault b/1, under which the procedure was applied, to propagate to the primary output. Thus, the *Max_Control( )* procedure extends the dominance relation of the single fault under test so that it dominates the maximum possible number of nodes in case of multiple faults occurrence.

#### 5.6.0.2 Sensitization Path Elimination Procedure

Given a set of multiple faults which are single–vector or multiple–vector self–masked, we need to modify the test vector(s) so that at least one fault dominates the others in the fault set. As discussed earlier, any single fault test set provide covers to single faults with minimal control requirements for the sensitized paths. It can then be argued that changing the logic value of a non–exciting [1] primary input might results in destroying one or more paths. This process might also leads to the detection of the multiple faults which are individually covered by the same test. This is the idea behind the sensitization path elimination procedure for multiple faults detection.

Consider the simple circuit example shown in Figure 5.6. The primary input sets for each of the single faults $a/0$ and $r/0$ are also shown in this figure. The double faults $\{a/0, r/0\}$ are self–masking under "1010". The logic values of the primary inputs which belong to the exciting sets of both faults, i.e., nodes $a$ and $c$, must be kept unchanged in order to excite both faults properly. As shown in Figure 5.6, the logic variation at the primary input node $d$, which belongs to $S_p$ of $a/0$, destroys

[1] The first requirement to cover a set of multiple faults is to excite all fault locations properly, i.e., if a fault location is s/0, then the fault site must be activated to logic 1.

Input : The double faults $a/0$ and $r/0$ under "1010".
Output : A cover for the double faults.

Procedure **Sens._Path_Elemination( )** {
1) **Identify** ($S_e$, $S_c$, $S_p$) for each fault;
$a/0$: $S_e$={$a$}, $S_c$={$b$, $c$}, $S_p$={$d$};
$r/0$: $S_e$={$c$}, $S_c$={$a$, $d$}, $S_p$={$b$};
2) **Combine** the exciting set of PIs;
$S_{exc}$ = {$a$, $c$};
3) **Fault Simulate** the logic variation for each non-exciting PI;
Input $b$ changes from 0 to 1;
Input $d$ changes from 0 to 1;
4) **Determine** the cover for the double faults;
Covers: "1110" and "1011";
}

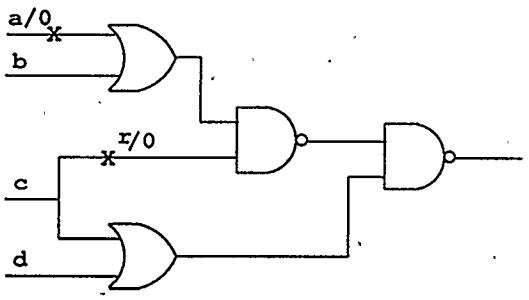



Figure 5.6. An example which illustrates the sensitization path elimination procedure.

the sensitization structure of $r/0$ from the fault location to the primary output[2]. Accordingly, the new test vector "1110" is a cover for the double faults. The same result can be achieved if the logic value of node $d$ is complemented where another test cover "1011" will be generated. In the former case, $a/0$ dominates $r/0$, while in the later case $r/0$ dominates $a/0$.

The main advantage of this method over the previous one is that test modification is applied without monitoring the logic values of any internal nodes in a circuit. The success of the above procedure lies in the fact that the logic variation in only one primary input would lead to the detection of the multiple fault set under test. From our experience, this fact seems to be holding for double and triple faults. For larger set of faults, more than one logic variation in the primary inputs might be needed. Fortunately, the sensitization path elimination procedure can be applied hierarchically. In such case, the resultant vectors, which guarantee the detection of a subset of faults in a larger set of faults, can be used by the same procedure to cover the remaining subset of faults.

## 5.7 Experimental Results on the 74LS181 ALU Circuit

In the previous section, two procedures for augmenting single fault test sets to maximally cover multiple faults has been presented. The first procedure aims at maximizing the control set of a single fault under a test vector without considering any multiple faults existence. This method can be directly implemented during the single fault test generation phase. The second procedure can only be applied on a set of uncovered multiple faults. The procedure aims at changing the dominance relation between the different faults in a multiple fault set. During this procedure, the sensitization structures of a subset of faults in a multiple fault set are destroyed while

[2]This is because $d \in S_c$ for $r/0$.

allowing other fault sensitization paths to terminate at the primary output nodes, hence, covering the set of multiple faults.

In order to evaluate the *Max_Control*( ) procedure, a test augmentation study was performed on the 74LS181 4-bit ALU. The circuit diagram for the 74LS181 is shown in Figure 5.7. The 74LS181 was selected because there are 16 single fault test sets available for this particular example. Different approaches have been used in generating these tests. The list of the sixteen test sets and the description of the test generation approach for each set can be found in [26]. A summary for the simulation study done by J. Hughes on the 74LS181 ALU is shown in Table 5.1. In this Table, the length of each test and number of uncovered double faults are listed. Among the sixteen test sets available for the 74LS181, only four sets achieved a 100% coverage for double faults. The remaining twelve sets will be examined using the procedures presented in the previous section in order to achieve maximum multiple fault coverage.

Our goal is to extend all the test sets which do not achieve an 100% double fault coverage. Therefore, this experiment aims at applying the *Max_control*( ) procedure on each cover in a test set. We select a single fault that is covered by the test vector and generate a new test which has maximal control set under the selected fault. For simplicity, selected faults are chosen from the set of primary inputs. The experiment is done for each test in a test set or until an 100% double fault coverage is achieved. Table 5.2 shows the results of this experiment. Under the *new tests* column is the number of new tests added to the original test set. Under the column *new cov.* is the double fault coverage of the augmented test set. As shown in Table 5.2, an 100% double fault coverage is always achieved. The number of new tests depends on the original set of tests and on the order of which these tests are listed.

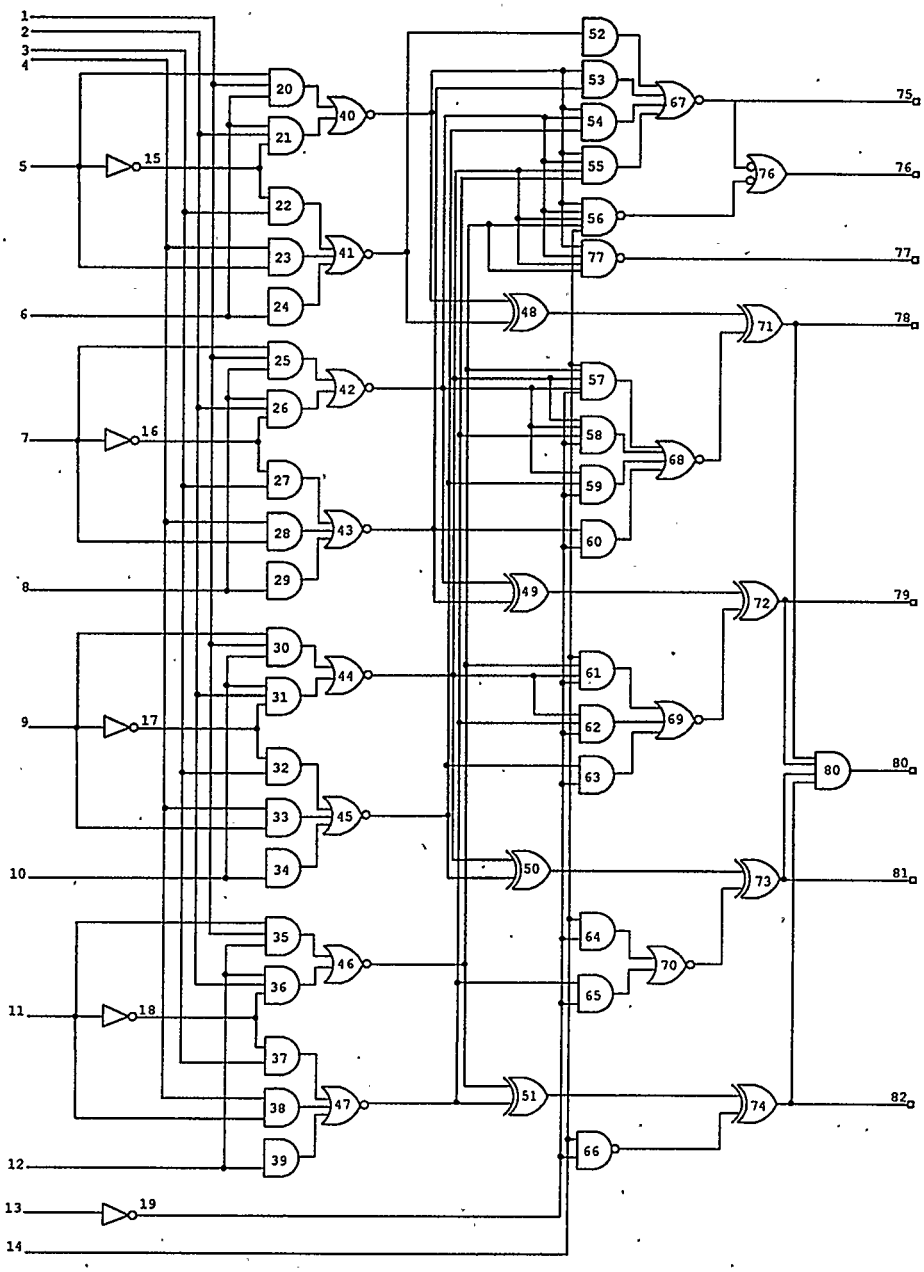



Figure 5.7. The 74LS181 ALU circuit diagram.

Table 5.1. A summary for the simulation study done by Hughes.

| Test set | Length | Uncov. faults |
|---|---|---|
| Krishnamurthy | 12 | 9 |
| Bryant1 | 14 | 4 |
| Bryant2 | 14 | 14 |
| Bryant3 | 14 | 11 |
| Bryant4 | 12 | 8 |
| Bryant5 | 12 | 1 |
| Bryant6 | 12 | 9 |
| Bryant7 | 12 | 28 |
| Bryant8 | 12 | 13 |
| Bryant9 | 12 | 19 |
| Miczo1 | 17 | 3 |
| Miczo2 | 17 | 30 |
| Goel | 35 | 0 |
| McCluskey1 | 124 | 0 |
| McCluskey2 | 352 | 0 |
| Hughes | 135 | 0 |

Table 5.2. Results obtained after applying the first experiment on the 74LS181 ALU.

| Test set | New tests | New cov. |
|---|---|---|
| Krishnamurthy | 5 | 100% |
| Bryant1 | 3 | 100% |
| Bryant2 | 9 | 100% |
| Bryant3 | 9 | 100% |
| Bryant4 | 7 | 100% |
| Bryant5 | 1 | 100% |
| Bryant6 | 4 | 100% |
| Bryant7 | 11 | 100% |
| Bryant8 | 7 | 100% |
| Bryant9 | 12 | 100% |
| Miczo1 | 2 | 100% |
| Miczo2 | 17 | 100% |

If the fault under investigation propagates to multiple outputs, all sensitizing paths from the fault site to the primary outputs are guaranteed to exist after applying the *Max_control*( ) procedure. Consider maximizing the control set of the test vector "01101100100100", from Miczo 1 and 2 test sets, under the single fault 27/0 (output node of gate 27 is s/0). This fault propagates to the primary outputs 79 and 80. The control objectives set by the test vector "01101100100100" are determined by implication. All logic implication values meet these objectives except at the circuit portion which involves gates number 61 and 62. The original test vector implies a logic value 1 at the output of gate 63 and hence a logic value of 0 at the output of gate 69. The fact that only one node (output of gate 63) controls the objective control logic value (output of gate 69) of the sensitizing path makes the fault 27/0 undetectable in the existence of the fault 63/0, i.e., the double fault {27/0, 63/0} is undetectable by the test vector. In order to maximize the control set of the original test vector under 27/0, the output of gates 61 and 62 must be set to logic one. This can be achieved by tracing back the logic values at the inputs of these gates to the primary input lines. Changing the logic values of the primary input nodes 11, 12, and 14 will meet the above requirements. Hence, the new test "01101100101001" is obtained. This new test covers the double faults {27/0, 63/0}.

As mentioned earlier, the above procedure has the disadvantage of being done by monitoring the logic values of some internal nodes in a circuit. If the test set contains a large number of tests, the above procedure may take some time before processing all the tests in the set. On the other hand, this procedure can be implicitly employed in single fault test generation systems by allowing a unique control assignments to be applied during the search process. These control assignments allow the inputs of each gate to maximally control the logic value of the gate's output.

The second experiment on the 74LS181 ALU is performed using the sensitization path elimination procedure. In this experiment, sets of multiple faults which are not covered by the original test set are explicitly investigated by the test augmentation procedure. The procedure starts by identifying the test(s) under which a set of multiple faults is self–masking. The implication of logic variations in each primary input of a test vector is simulated against the set of faults. For example, let us consider the double faults {27/0, 26-1/1} which are self–masking under "01101100100100". Our experiment has shown that any single logic variation in the primary input 1, 4, 5, 6, 9, 10, 12, 13, or 14 will result in a cover for the double faults. On the average, the effect of these faults is observed at three different primary outputs. In all twelve test sets, we were able of covering all the double faults in the 74LS181 and achieve a 100% double fault coverage.

## 5.8 Multiple Fault Detection Using the GATPG Framework

The analysis and procedures presented so far dealt with the problem of extending a test set to cover a maximal number of multiple faults. The attention will now be directed to the generation of test sets that implicitly cover maximal number of multiple faults in a circuit.

### 5.8.1 The Approach

The GATPG seems to be the natural framework for detecting multiple faults in a circuit because it explores the search space for many faults simultaneously. In this context, there are two ways by which we can approach the problem of generating multiple fault tests in the GATPG system. The first approach can be explained using the structure shown in Figure 5.8. As with the case of multiple path sensitization dis-

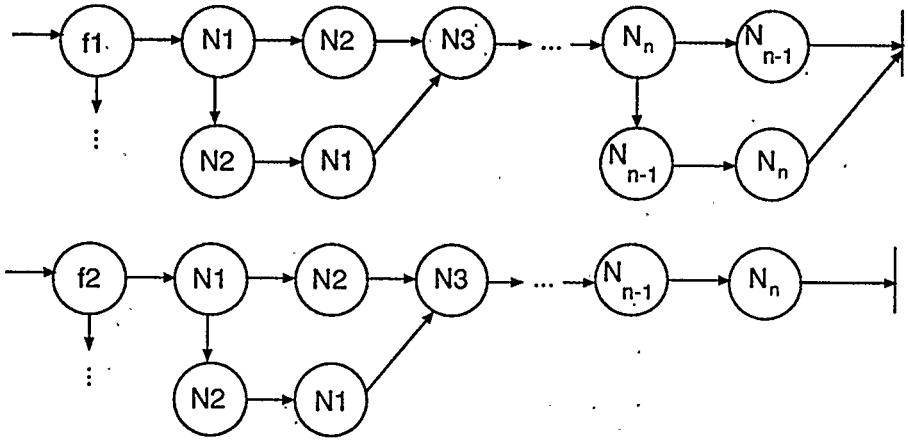



Figure 5.8. A general data structure for two faults in a circuit.

cussed in the previous chapter, multiple fault test generation requires the availability of the sensitization paths for more than one fault simultaneously. Figure 5.8 shows the sensitization paths of two faults in the data structure created by the GATPG algorithm. In order to find a test for the double faults $f_1$ and $f_2$, the cube intersection of the fault values and the control values ($N_1$ to $N_{n-1}$) for the different paths associated with each fault must be performed. This is exactly the same method used in generating multiple path sensitization patterns. In the matter of fact, multiple path sensitization is one form of multiple fault behavior because it allows more than one stem node in a fanout structure to carry a fault logic value. However, the multiple path sensitization procedure is applied locally, i.e., to a fanout structure. On the other hand, multiple faults, such as $f_1$ and $f_2$ in Figure 5.8, may occur between any two (or more) nodes in a circuit. It would be extremely difficult and time consuming to apply the multiple path sensitization procedure to all the multiplicities of faults in large (or even moderate) size circuits.

The second approach, on the other hand, uses the analysis and results presented so far in this chapter to implicitly generate tests that cover any multiplicities of faults without adding any new procedures in the GATPG system.

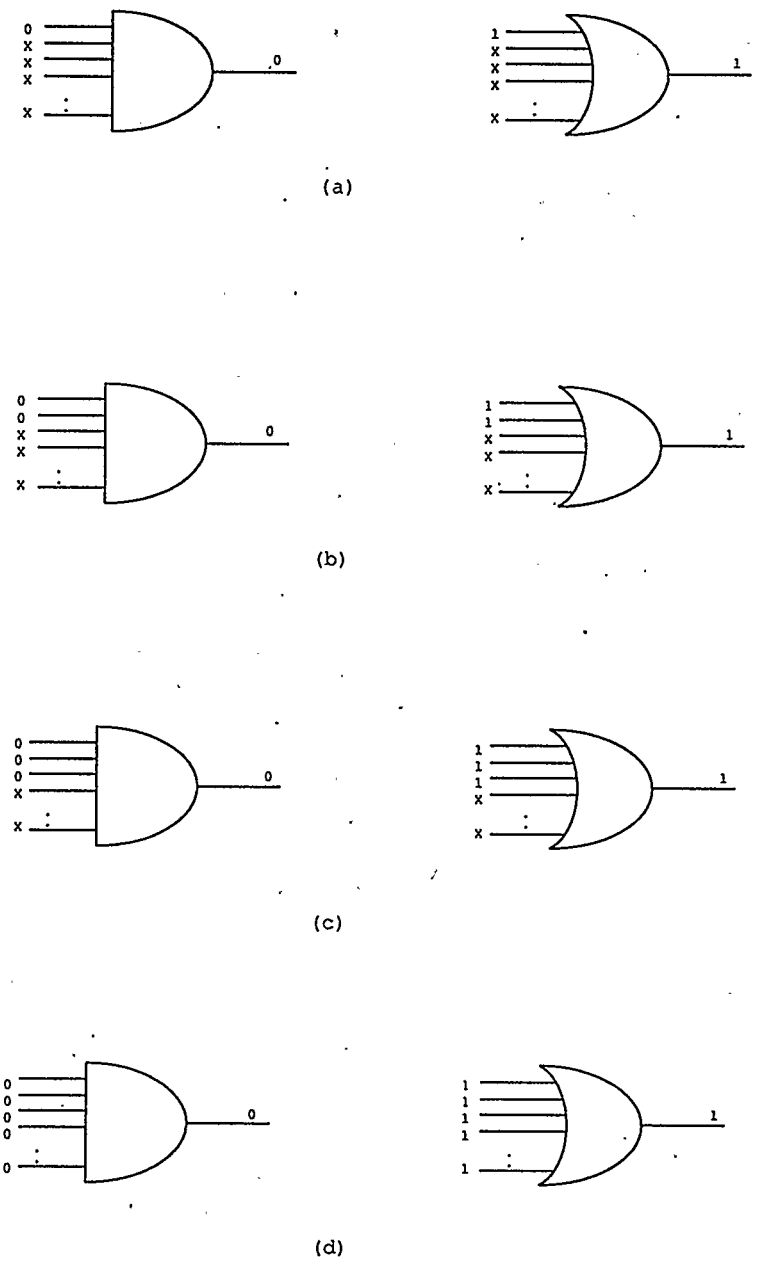



Figure 5.9. Control logic assignments for implicit multiple fault coverage (a) single fault coverage (b) double fault coverage (c) triple fault coverage (d) all multiple fault coverage.

In order to achieve maximal multiple fault coverage, the GATPG applies a set of control logic assignments during the back propagation process which will guarantee that each generated test pattern has a maximal control set $S_c$. So far, we have concentrated on how a fault sensitization path is created during the back propagation process. We would like to direct the attention now on how the control logic values associated with each fault entry in the data structure are created. In case of single faults, the GATPG uses control logic assignments like the one shown in Figure 5.9.a, where the control logic value of a gate's output is determined with a minimum number of deterministic (0's and 1's) control logic assignments to the gate's inputs. This assignments results in a test pattern that has minimal control set $S_c$. On the other hand, the minimal number of deterministic control logic assignments makes it much easier and more likely to find a test for a fault in the circuit.

Consider the problem of generating a test pattern with a maximal control set against all double faults multiplicities in a circuit. In order to achieve this goal, control logic assignments such as the one shown in Figure 5.9.b must be used to guarantee that if a line that contributes to the propagation of a fault from its location along the sensitization path fails (stuck–at opposite control logic value), another line with the required control logic value will still support the fault propagation. Hence, a cover for the fault and the failed line will always exists. Similarly, a test pattern with a maximal control set against all triple multiplicities of faults can be achieved using assignments similar to the one shown in Figure 5.9.c. Generally, a maximal control set for any multiplicity of faults can be obtained using the assignments shown in Figure 5.9.d. The assignments shown in Figure 5.9 are not unique, i.e., there can be other enumeration of logic values which achieves the same purpose.

In this way, any multiplicities of faults can be expressed within the the GATPG

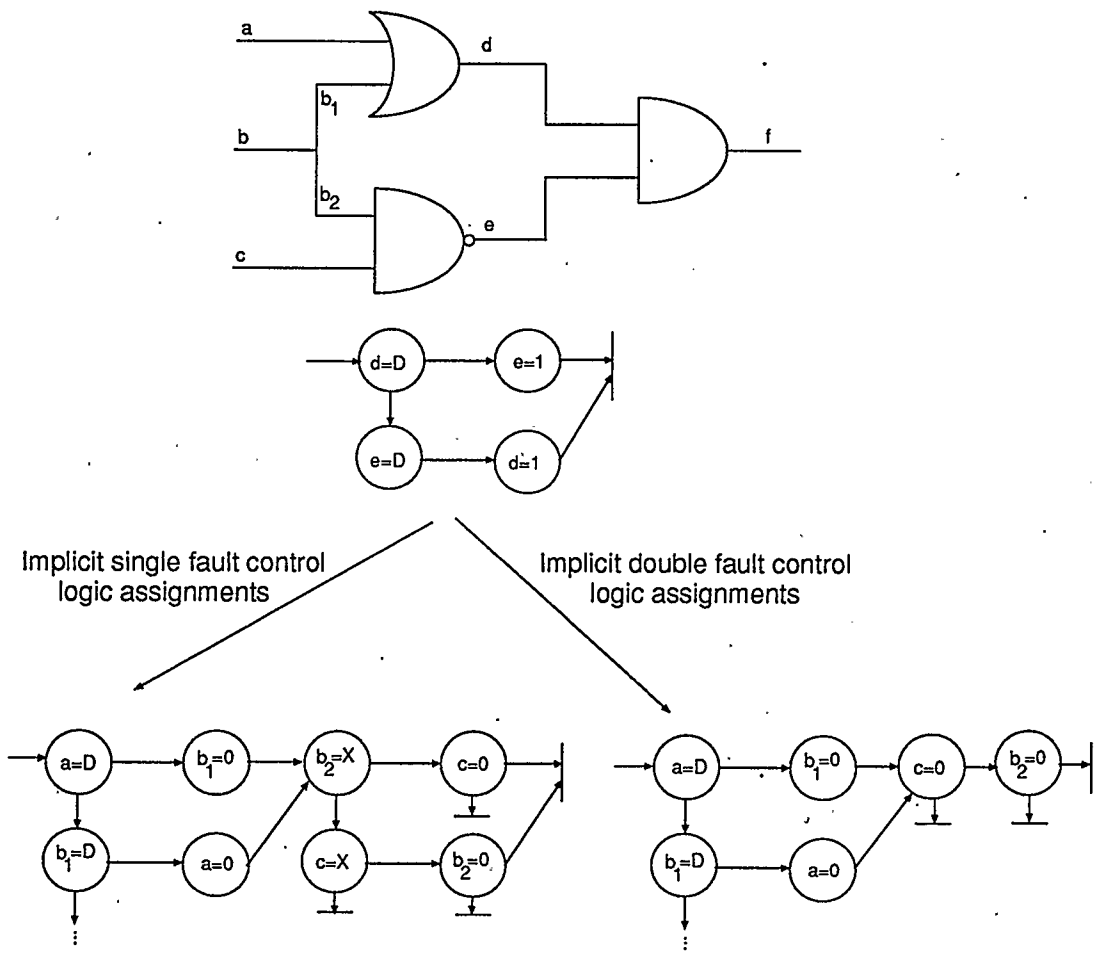



Figure 5.10. The impact of implicit multiple fault control assignments on the data structure.

system. The test system can be tailored to generate tests with maximal double, triple, or any multiplicities of faults. This technique can be used in any other ATPG algorithm during the test generation phase. The problem with these systems is that they may guarantee the implicit maximal coverage of multiple faults for tests generated deterministically, but will fail to provide such tests during the random phase. In our approach, this problem does not exist because the GATPG system is single phase.

### 5.8.2 Implementation and Results

The control logic assignments described above will have some impact on the implementation of the GATPG system with implicit maximal multiple fault coverage. Generally, these assignments will impose some constraints on the search space for tests and might degrade the quality of the generated test patterns. To elaborate on this point, consider the circuit example shown in Figure 5.10. In this figure, the data structure at the second level of assignments is shown twice, one time with the single fault control logic assignments and the other with implicit double fault control logic assignments. In the former case, node $e$ (at the first level of assignment) has a logic value of one. This logic value is enumerated at the inputs of the NAND gate with one input carrying a 0 logic value while the other is carrying an $X$ logic value. It it obvious that this enumeration will enable us to cover both types of faults at node $b_1$ (because $b_2$ has an X logic value in one of the enumerated inputs of the NAND gate). In the later case, implicit double fault control logic assignments dictates that both inputs of the NAND gate must carry the same 0 logic value. It can be seen that, in this case, only one type of fault (s/1 fault) at node $b_1$ may be covered. This will definitely decreases the fault coverage of the generated test set.

In order to avoid the degradation of the test quality of the GATPG system, implicit

Table 5.3. Real execution performance of our algorithm in a single-phase implementation with implicit double fault maximal coverage on a SUN SPARC 2 workstation with the ISCAS'85 benchmark combinational logic circuits. Time units: seconds.

| Circuits | BFP | | | Fault | # Test |
|---|---|---|---|---|---|
| | Dfault | Rfault | CPU | coverage | vectors |
| C432 | 515 | 4 | 0.7 | 100% | 91 |
| C499 | 750 | 8 | 1.6 | 100% | 89 |
| C880 | 942 | 0 | 2.8 | 100% | 188 |
| C1355 | 1566 | 8 | 1.5 | 100% | 192 |
| C1908 | 1872 | 7 | 7.9 | 100% | 162 |
| C2670† | 2480 | 115 | 36.8 | 100% | 393 |
| C3540† | 3297 | 131 | 31.1 | 100% | 324 |
| C5315† | 5291 | 59 | 46.7 | 100% | 429 |
| C6288† | 7710 | 34 | 14.2 | 100% | 145 |
| C7552† | 7417 | 131 | 55.6 | 100% | 511 |

single and multiple fault control logic assignments are both allowed and enumerated during the back propagation procedure. The GATPG system allow a maximum of two enumeration of multiple fault control logic assignments at the inputs of a gate during the back propagation procedure. This is followed by the full enumeration of the original single fault control logic assignments. Since such logic assignments will increase the length of the number of paths associated with each fault entry in a data structure, we have decided to increase the preset limit from 2 – 5 in the previous implementation to 4 – 7 in this implementation. In this way, no compromise is made toward the test generation for single faults. If a cover with a maximal control set exists, at any level during back propagation, the GATPG system will find it. Otherwise, it will generate a pattern with implicit single fault coverage at that level of assignment.

As a result of the above strategy, we would expect that the generated number of test patterns will be the same as in the previous implementation. The only difference between the two sets of patterns is that in the new implementation each test pattern

Table 5.4. Real execution performance of our algorithm in a single-phase implementation with implicit double fault maximal coverage on a SUN SPARC 2 workstation with the ISCAS'89 benchmark combinational logic circuits.

| Circuits | BFP | | | Fault | # Test |
|---|---|---|---|---|---|
| | Dfault | Rfault | CPU | coverage | vectors |
| S27 | 32 | 0 | 0.1 | 100% | 13 |
| S208 | 215 | 0 | 3.3 | 100% | 52 |
| S298 | 308 | 0 | 3.5 | 100% | 79 |
| S344 | 342 | 0 | 3.8 | 100% | 71 |
| S349 | 348 | 2 | 2.5 | 100% | 62 |
| S382 | 399 | 0 | 2.7 | 100% | 63 |
| S386 | 384 | 0 | 3.0 | 100% | 81 |
| S400 | 410 | 6 | 3.3 | 100% | 77 |
| S420 | 430 | 0 | 4.0 | 100% | 109 |
| S444 | 460 | 14 | 3.9 | 100% | 45 |
| S510 | 564 | 0 | 5.1 | 100% | 86 |
| S526 | 554 | 1 | 4.9 | 100% | 127 |
| S641 | 467 | 0 | 4.0 | 100% | 117 |
| S713 | 543 | 38 | 6.1 | 100% | 177 |
| S820 | 850 | 0 | 5.7 | 100% | 181 |
| S832 | 856 | 14 | 5.2 | 100% | 169 |
| S838 | 857 | 0 | 19.2 | 100% | 202 |
| S953 | 1079 | 0 | 8.2 | 100% | 169 |
| S1196 | 1242 | 0 | 20.8 | 100% | 221 |
| S1238 | 1276 | 69 | 27.9 | 100% | 299 |
| S1423 | 1501 | 14 | 9.9 | 100% | 161 |
| S1488 | 1487 | 0 | 10.7 | 100% | 255 |
| S1494 | 1494 | 12 | 7.3 | 100% | 241 |
| S5378 | 4563 | 40 | 88.4 | 100% | 444 |
| S9234† | 6475 | 452 | 224.1 | 100% | 812 |
| S13207† | 9662 | 151 | 176.5 | 100% | 807 |
| S15850† | 11336 | 389 | 341.6 | 100% | 723 |
| S35932† | 35110 | 3984 | 112.3 | 100% | 394 |
| S38417† | 31015 | 165 | 498.0 | 100% | 2501 |
| S38584† | 34798 | 1506 | 157.9 | 100% | 2110 |

has a maximal control set which guarantees that the total set of tests has a maximal multiple fault coverage.

We have applied the above implementation of the single phase GATPG system using the implicit double fault control assignments on the ISCAS'85 and ISCAS'89 benchmarks. The results are shown in Tables 5.3 and 5.4. It can be seen that all the data are similar to those in the single phase GATPG implementation for single faults, except for the running time. The run time has increased due to the increase in the preset limit. The generated test set is guaranteed to have a maximal double fault coverage.

## 5.9 Summary

In this chapter, an analysis based on the sensitization structure behavior in the existence of multiple faults has been given. The purpose of this analysis was to identify the conditions under which a set of multiple faults are self-masking. Having done that, the ultimate goal is to use this information to guide the process of single fault test generation and/or in extending single fault test sets in order to achieve a maximal multiple fault coverage.

We have presented two different procedures for augmenting any single fault test set. An experiment has been carried out on the 74LS181 ALU using twelve single fault test sets. It has been shown that different fault classes can be covered using any of the above procedures. Although these procedures proved very efficient in achieving a 100% double fault coverage, we believe that test augmentation and multiple fault simulation can be totally avoided if the problem of maximizing the control set of each generated test in a single fault test generator is taken into account. The current approaches for single fault test generation can easily apply some unique control as-

signment during the search process. The fact that test generation is carried out on a specific target fault makes it easier to apply such technique.

Finally, we have extended our GATPG algorithm to generate test patterns with implicit maximal multiple fault coverage. We have achieved that by using unique control logic assignments at the gate inputs during back propagation which guarantees that each generated pattern will have a maximal control set, and hence, covers a maximal number of multiple faults. Our system is the only known system that generates test vectors with maximal multiple fault coverage for the general class of combinational circuits.

# CHAPTER 6

# THE MODULAR TEST GENERATION SYSTEM

In previous chapters, the framework and the implementation of a single phase global test generation system have been presented. It has been shown that the test primitives generated using the GATPG system can be used as test patterns for the module under test or for mapping information across the interfaces of a module. In this chapter, both representations of the test primitives will be used in a modular test generation system. We will first introduce the modular test generation problem, and then present the modular test generation procedures. The different features of the modular test system will be explored using an adequate circuit example. Differences and similarities of the proposed modular test approach with other approaches will be identified. A cost model for the modular test system will then be presented. The purpose of the cost model is to predict the possible speedup in test generation for modular test systems against low level testing.

## 6.1 Introduction

A growing class of integrated circuits is designed using libraries of large subcircuit modules, which are not readily decomposable into logic gates or whose gate–level design is unavailable to the test engineer. These include circuits designed by silicon compilers, as well as standard cell systems. Most design systems also support hierarchical design methods employing both high (module or macro–based) and low (bit)

level circuit models. Experience in a variety of domains suggests that using hierarchy can reduce design complexity. There is increasing evidence that this is true for test generation as well, but this particular problem is still poorly understood.

The purpose of modular test generation is to simplify the test generation process for large circuits. The importance of hierarchical or modular test generation has been recognized as early as 1975 [6, 55, 35]. Some hierarchical techniques have been proposed for test generation and fault simulation to avoid explosive cost increases. Krishnamurthy [31] and Calhoun *et al.* [11] proposed a new framework for hierarchical ATPG, Daseking *et al.* [15] developed a multilevel test generation technique which extensively uses the circuit hierarchy in test generation, Murray and Hays [43] developed module–level testing using stimulus–response pairs, and Sarfert *et al.* [47] extended their gate–level SOCRATES to process high–level modules. Hyoung *et al.* [41] incorporated dynamic hierarchical circuit reconfiguration, and heuristic mechanism to directly perform propagation, backtracing, and implication with high level functional models.

The other aspect of hierarchical test generation is its cost prediction. A few attempts have been made to predict the test generation cost [23, 40, 20, 17]. Goel [23] estimates the cost of parallel and deductive fault simulation, and test generation cost for gate–level circuits with no backtracking. Min and Rogers [40] generalize and extend Goel's model by incorporating the cost of backtracking. Fisher *et al.* [20] have significantly improved this model, which can now be used to predict ATPG run time, fault coverage, and test length. Hyoung *et al.* [41] developed a hierarchical cost model that is a hierarchical extension of these previous models.

Although these techniques support different approaches to modular testing, they are not providing efficient solutions to the problem because they all depend on a

common test generation framework that does not support all aspects of modular test generation. For example, in [43], the test generation method at the gate–level is not analysed and it is assumed that each module retains its test primitive regardless of the technique used in generating them. Although this approach seems adequate in the context of modular test generation, it will have the effect of increasing the complexity of the test procedures at the modular level. It is important to unify the way test generation techniques are used at any level of hierarchy, so that the test engineer can concentrate on the tests and their quality without having to switch to different heuristics at different levels of hierarchy. In [41, 34], the modular test systems did not keep the test cost to a minimum because different heuristics are used at the modular and gate levels of hierarchy.

### 6.1.1 The Modular Test Generation Approach

As mentioned in Chapter 3, a system is modular if it can be described as a collection of modules with limited, well–defined interfaces. A test system is modular if it can use the set of test vectors which covers all the faults in the module and a description of well–defined interfaces of modules to generate tests at the primary inputs of a chip. Current ATPG systems *uses* the hierarchy of a circuit in generating tests, i.e., the design *must* be completed before the test procedures are applied. This raises a question about how truly hierarchical these systems are. We view hierarchical test generation as an interactive process in which tests are developed during the circuit design development cycle. It is imperative that such systems are more likely to be integrated in a circuit design CAD tools. Other conventional approaches may or may not be integrated in such tools because they deal with the test problem after designs are completed.

Currently, if the design needs to be modified to improve the test quality, the test engineer will then have to deal with the completed design as a single entity. This increases the difficulty of debugging the design for testability problems. Therefore, these approaches are inefficient not only because of their inability to apply truly modular test techniques but also of the difficulties they introduce in the test quality improvement process.

In our modular test generation approach, we provide a truly modular/hierarchical test generation framework. This framework allows the designer to examine the test quality of the design at any level in the circuit hierarchy. Test evaluation and quality improvement can be achieved at any level of hierarchy, regardless of the details at lower levels of hierarchy. This process is accomplished during the design cycle of the product. This means that tests are generated for the designed portions of the chip, i.e., we do not require the completion of the chip design to apply our test procedures. Once the top level of hierarchy is reached, then, the modular tests generated at that level will represent the chip tests. The impact of this approach on CAD tools design will be substantial because it matches the current practice in the design cycle of VLSI circuits. Test quality improvement is also incremental, i.e., the designer can look at the different aspects of his/her design (optimality, verification, testing, etc.) and explore different approaches to attain the design goals at one level in the hierarchy without reference to lower or higher levels.

Current modular test generation systems uses the circuit hierarchy to create a symbolic test path between a fault location and the primary outputs of the chip. A symbolic test path can be established by putting the intermediate modules (relative to the module under test) in a special mode, a so called *transfer mode.* In a transfer mode, the data is transferred unchanged from a module's inputs to its outputs. The

test paths can also be represented by functional relations that are calculated by the circuit rules. In [43, 5, 48], these paths are found by propagation and justification algorithms that are mainly based on similar heuristics used at the gate–level test algorithms. Generally, it is a tedious job to derive the necessary transfer modes for arbitrary modules in any of the proposed approaches. These difficulties arise because the test interface at different levels of hierarchy are not defined properly. The system must first derive the transfer mode for each module and then use time consuming procedures to generate chip tests from module's tests.

In our approach, at the gate–level, the GATPG generates test primitives which represent not only the test patterns for a module but also represent the transfer mode as well. We defined the interface between low and modular level test generation and used simple test procedures to generate the test primitives. The characterization of the test primitives was mainly based on the type of heuristics that are commonly used at the modular test level, i.e., information mapping. We will reiterate here the characterization of the test primitives generated at the gate–level using the GATPG system:

- Each pattern in the test primitive represent a sensitization path that is generated using our GATPG algorithm. In other words, the test primitive includes the test vectors for the module under test.

- Propagation and justification heuristics are symbolically represented within the test primitive. This representation allows the modular heuristics to be applied without reference to the internal circuitry of the module.This representation is complete, i.e., there is no need for any other procedures or data representation during modular testing.

- The representation of symbolic paths is achieved using test procedures in our GATPG system. Therefore, no additional functional heuristics are needed to generate them.

This characterization shows that the test primitives retains functional information for the module. Therefore, deriving the transfer modes for modules is not required in the proposed modular test generation system. Also, no extra heuristics will be needed (such as justification and backtracing) at the modular level. This will substantially improves the performance of our modular test generation system. Therefore, the purpose of our modular test generation system is not the application of any new procedures at the modular level but the *assembly* of test patterns from modules to the chip's primary inputs/outputs using the test primitive of each module.

### 6.1.2 System level test assembly

In order to describe the assembly of test patterns at the chip's boundary, a system approach must be determined first. In this approach, the modular test system will determine which module to select and to map (assemble) its test set to the primary inputs/outputs of the chip. Of course, module selection will depend on the way modular test generation is carried out.

Assume that the hierarchy of the circuit under test is a *K-ary* tree as shown in Figure 6.1. The root of the tree is on the $0_{th}$ level and the leaves are on the $K$-1 level. In current hierarchical ATPG systems, every leaf node represents a gate with $f_g$ modeled faults to be covered by the test system. In the context of current hierarchical ATPG systems, a path from the fault location at one of the leaf nodes is created across the hierarchy of the circuit. This means that each time a cover for

a fault is generated, modular heuristics must be applied to hierarchically generate a corresponding test at the chip's primary inputs. As the number of faults increases, these heuristics will pose a serious time constraints on the system's performance. The major drawback of such approach is that since the gate–level abstraction is allowed in the circuit hierarchy, it is possible that the system will generate tests for identical modules whose unique description at the modular level are not exposed by the test system during gate–level test generation. Consequently, the test system will not be able to use its full potential in reducing the test generation time. This is a typical consequence of dealing with the one fault at a time framework which is still adopted in current hierarchical ATPG approaches.

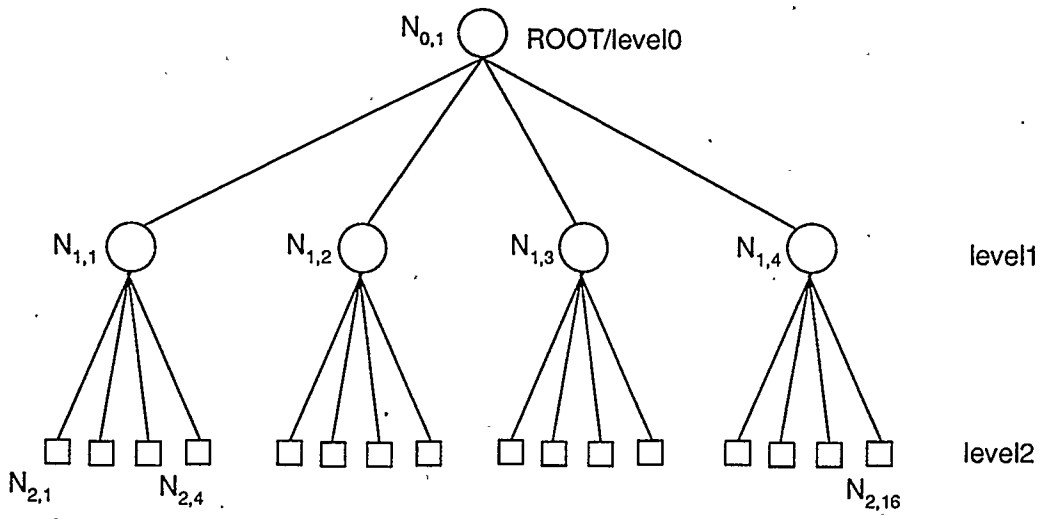



Figure 6.1. Circuit hierarchy in modular test generation.

A final remark can be made on the module selection technique in modular test generation in current approaches. Since the single target fault strategy is used in these approaches, and modular heuristics are separated from gate–level heuristics, it can be seen that such systems may perform hierarchical test generation on any module in the circuit. The only requirement is to perform modular test generation

for all the modules in a circuit, i.e., to apply the modular heuristics to all the leaf nodes in the hierarchy. Consequently, the module selection in these approaches can be random.

In our approach, on the other hand, all nodes in the circuit hierarchy represent modules, including the leaf nodes. All the information related to the internal circuitry of the leaf modules are hidden behind their interfaces and is represented only by the test primitives generated by the GATPG. Now, consider the problem of assembling the test primitive of one of the leaf node modules at the circuit's primary inputs/outputs. Our modular test system starts with all the leaf node modules having their test primitives generated. The system (unlike other approaches) does not retain any other information about the functionality of modules at the upper levels of hierarchy. Therefore, it is not possible for our modular test system to create paths similar to the one created by other test systems. Instead, the system generates test primitives at one hierarchy level using the test primitives of the son level in the hierarchy. Such strategy is highly desirable because it matches the VLSI design practice in today's CAD tools. For example, consider the problem of designing an ASIC with three levels of hierarchy as shown in Figure 6.2. The ASIC design consists of four major modules; each module is subsequently divided into a number of smaller modules. The designer will develop each of the leaf modules separately and assembles them into the four major modules, and then interconnects the large modules to complete the design. Similarly, our modular test system takes the test primitives of the leaf modules and generates test primitives for the larger modules, and then generates test primitives for the ASIC chip using the test primitives of the four major modules in the chip.

Since our modular test system works totally at the modular level (no gate–level abstraction), it is easy to identify similar modules in the hierarchy. For example, if the

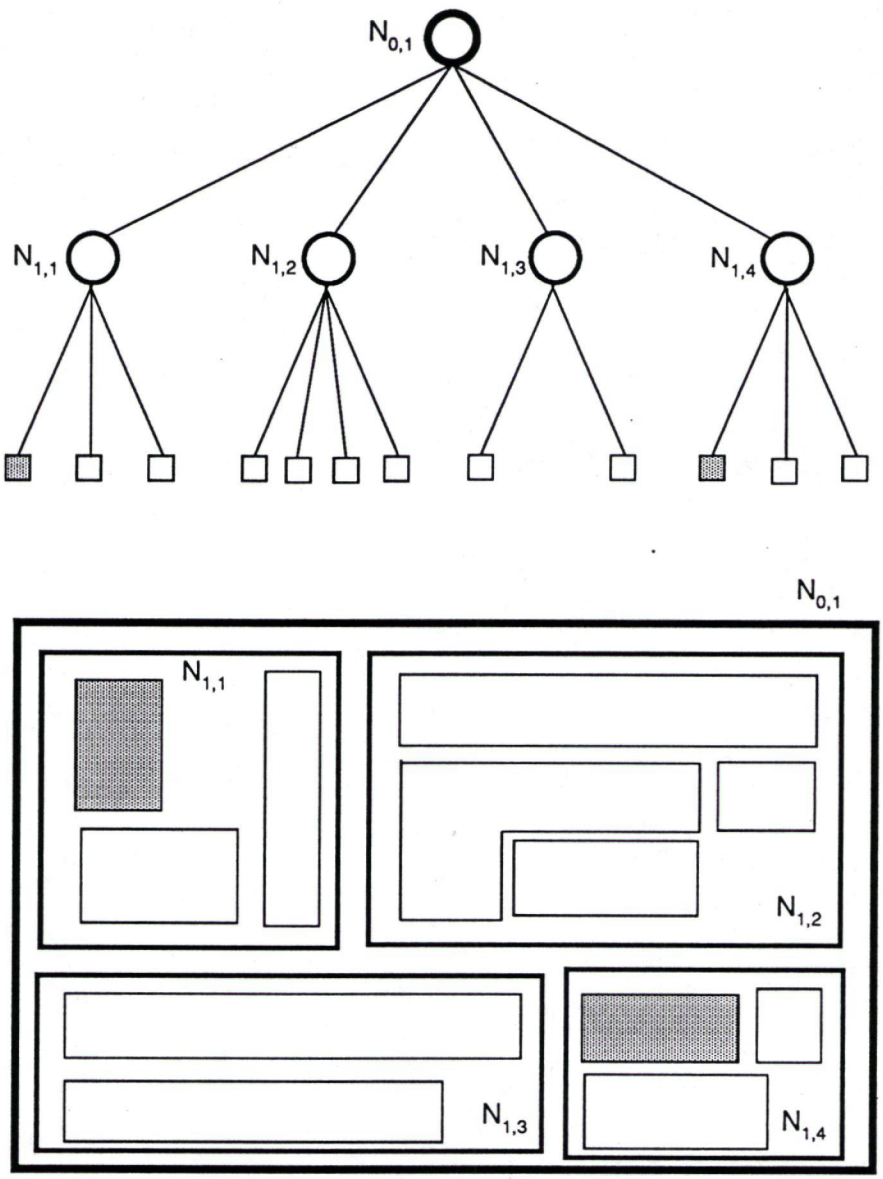



Figure 6.2. The modular decomposition of large ASICs in the design stage.

two shaded modules in Figure 6.2 are identical, the modular test system will allow the test primitive of one of these modules to be generated (using the GATPG algorithm) and then copies it to the other module. Similarly, if at any level in the hierarchy, two modules are identical, the assembled tests for one module will be copied to the other one. This framework not only simplifies the test generation process but also saves a significant amount of time, both in test generation at low level and test assembly at the modular level.

There is one more issue that has to be resolved before we can present the procedures used in the assembly of tests hierarchically. This issue relates to the strategy of module selection for test assembly in our modular test generation system. Let us classify the module selection process as random and deterministic. Random selection means that the modular test system will assemble tests starting at any module in a circuit. Deterministic selection dictates that a certain order of modules must be followed during the test assembly process in order to completely generate chip tests without failure. As explained earlier, current hierarchical ATPG systems performs on the gate level of abstraction, and create symbolic test paths using a precomputed functional information (transfer mode) to hierarchically generate the test for a target fault. The term target fault here implies that *any fault* can be dealt with in a similar fashion. Therefore, random selection of modules (gates, in this case) can be applied to current hierarchical ATPG systems.

In the proposed modular test system, test assembly can only be performed between two consecutive levels of hierarchy. The reason is that it is assumed that the full chip design is not available at the time of test assembly. The test assembly process is incremental and only proceeds with the creation of a new hierarchy level. Therefore, random selection of modules cannot be applied in our modular test approach. This

implies that the target fault strategy at the modular level cannot be adopted as well. Therefore, the global test strategy that has been used in our low–level GATPG system is also used in the modular test approach. Global test generation was achieved at the gate–level of abstraction by iteratively generating tests for each output cone in a circuit. Similarly, global test generation at the modular level can be achieved at the $i_{th}$ level of hierarchy by selecting a module at the $i_{th} + 1$ level of hierarchy that is attached to a primary output of the parent module. Then, the test assembly procedure is applied on the test primitives of the selected module to generate tests for the parent module. This process continues until all tests for modules at some level in the hierarchy are assembled. When the the hierarchy level reaches the root level, the chip tests will be generated and test assembly is completed. This strategy is explained in more details in the next section.

## 6.2 The Test Assembly Procedures

As mentioned earlier, the purpose of our modular test system is to assemble tests from the leaf modules to the inputs/outputs of the chip. We have shown that our modular test approach supports global test generation at the modular level. The test assembly procedures for the modular test system are shown in Figures 6.3 and 6.4. In Figure 6.3, the System_level_assembly( ) procedure shows how the test system perform at each level in the hierarchy. This procedure ensures that similar modules are not processed more than one time and allows for the tests of one module to be copied to a similar module.

The Module_level_assembly( ) procedure organizes the module selection process and the assembly of tests at the boundary of a module. To illustrate this point, consider the modular representation shown in Figure 6.5.. In this figure, the parent

```
Procedure System_level_assembly( ) {
   hierarchy_level=k-2 (level above leaf nodes);
   while (hierarchy_level != root level);
      {
      Current_module_list1 is empty;
      j=0 ;
         while module(j) exists at the current hierarchy_level;
            {
            if module(j) is in the Current_module_list;
               copy test_primitive of similar module to module(j);
            else
               {
               Assemble_tests;
               add module(j) to Current_module_list1;
               }
            j=j+1;
            }
            hierarchy_level=hierarchy_level-1;
      }
}
```

Figure 6.3. The system–level test assembly procedure.

```
Procedure Module_level_assembly( ) {
  for each primary output PO_q in module m(j) at level i;
    {
    find a module m(k) at level i+1 thet is connected to PO_q;
    Current_module_list2 = m(k) ;
    Current_assembled_tests =  test primitive of PO_q in m(k);
      while current_module_list2 is not empty;
        {
        for each test pattern in Current_assembled_tests;
              {
              back_propagate patterns across modules;
              if back_propagation successful;
                 add mapped pattern to a new_test_primitive;
              else continue with other pattern;
              }
        Current_assembled_tests = new_test_primitive;
        Update Current_module_list2;
        }
    }
}
```

Figure 6.4. Test assembly procedure at the module level.

module m(j) is at level $i$ of hierarchy and has four primary outputs. At the $i_{th}$+1 level of hierarchy, there exists five sub–modules. The Module_level_assembly( ) procedure selects a primary output for the parent module $m(j)$, for instance, $PO(0)$. It then searches for a module at the $i_{th}$+1 level of hierarchy which is connected to that primary output. In this case, module $m(k)$ will be selected.

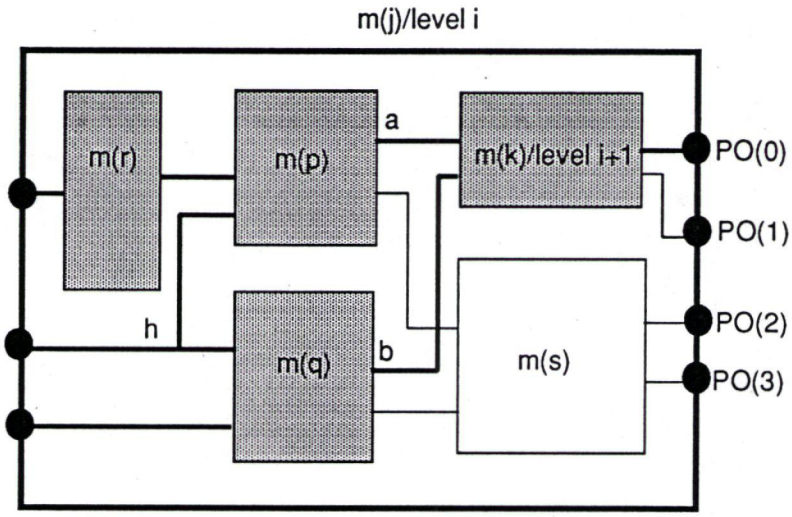



Figure 6.5. An example showing module selection in the test assembly procedures.

The test assembly will be performed on the test primitive of the output cone $PO(0)$ for $m(k)$. The test assembly starts by the back propagation of the test primitive of output cone $PO(0)$ across the two modules $m(p)$ and $m(q)$. The test primitives of these two modules will be used as the mapping (functional) information which maps output logic values to input logic values. For example, if node $a$ (connected to the output of module $m(p)$) in the figure carries a $D$ fault logic value, while node $b$ (connected to the output of module $m(q)$) carries a logic one, then, the test primitive of module $m(p)$ will be cube-intersected with the *true state primitive* of node $b$ across module $m(q)$. The *true (false) state primitive* represent all the entries in the test primitive of a module which generates an output logic of one (zero) for the fault–

free response of the circuit. True and false state primitives represent the functional behavior of a module.

Cube intersection is necessary during test assembly because different modules may share some inputs, and therefore conflict of assignments might occur. For example, node $h$ in Figure 6.5 is a common input to the two modules $m(p)$ and $m(q)$. The result of the cube intersection of the two primitives at node $h$ will determine if the mapping of some pattern is successful or not. Once all successful mapping occur, the procedure starts another cycle to map the assembled tests at the inputs of modules $m(p)$ and $m(q)$ to the inputs of $m(j)$. The shaded modules and the bold lines in the figure are the one involved in the back propagation of the test primitive of module $m(k)$.

### 6.2.1 An Example

To probe more on the back propagation of test patterns during the test assembly, consider the circuit example shown in Figure 6.6. This figure shows a hierarchical description of a 3–to–8 decoder. There are three levels of hierarchy in Figure 6.6, with all the leaf nodes representing 1–to–2 decoders. Therefore, the GATPG can be applied to only one of these modules. The test primitive for the 1–to–2 decoder is shown in Figure 6.7. A copy of this test primitive will be loaded into each one of the leaf node modules. At level one in the hierarchy, the left most node is 1–to–2 decoder. The System_level_assembly( ) procedure will copy the test primitive of a 1–to–2 decoder to that node. The two other nodes represent 2–to–4 decoders, hence, the System_level_assembly( ) procedure will assemble tests for one of them and copies these tests to the other node.

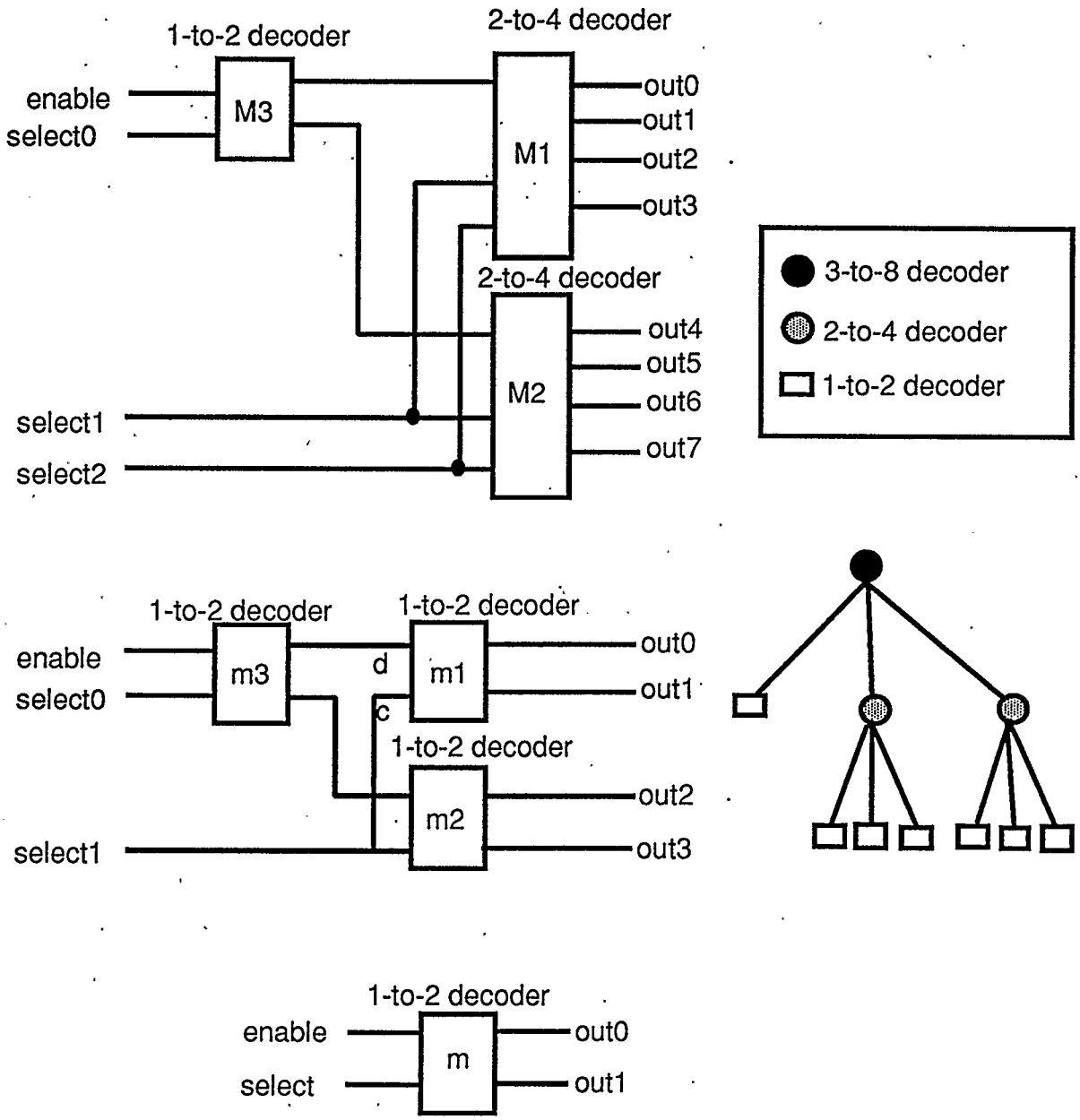
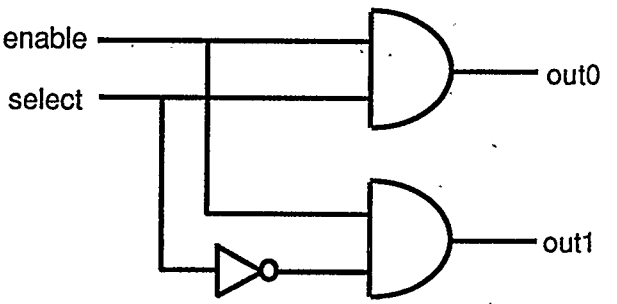



Figure 6.6. Hierarchical description of a 3–to–8 decoder circuit.




Test primitive (fault mapping)

Cone(0)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | D | D | $\overline{D}$ |
| D | 1 | D | 0 |

Cone(1)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | $\overline{D}$ | $\overline{D}$ | D |
| D | 0 | 0 | D |

True state primitive

Cone(0)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | 1 | 1 | 0 |

Cone(1)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | 0 | 0 | 1 |

False state primitive

Cone(0)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |

Cone(1)
======

| enable | select | out0 | out1 |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 |

Figure 6.7. The circuit diagram and the test primitive for a 1–to–2 decoder.

The Module_level_assembly( ) procedure will assemble tests at level one of hierarchy for the 2–to–4 decoder using the test primitives of the three 1–to–2 decoders at level two of hierarchy as shown in Figure 6.6. It starts by back propagating the test primitive of the output cone at *out*0. The test primitive of this cone includes two input fault patterns (1, $D$) and ($D$, 1) assigned to nodes $c$ and $d$ of module $m1$; shown in Figure 6.6. The assembly of the first pattern can be achieved by the back propagation of the logic value "1" across module $m3$. Since node $c$ is a primary input (select1) of the parent module, it needs not to propagate any further. In order to back propagate the logic value one across module $m3$, the Module_level_assembly( ) procedure substitutes each $D$ fault logic value in the test primitive with logic value one in order to create the *true state primitive* of module $m3$ at the output cone of node $d$. The *true* and *false* state primitives of module $m3$ (a 1–to–2 decoder) is also shown in Figure 6.7. They represent the mapping of control logic values across a module. Since all test primitives were generated with the primary output assigned a $D$ fault logic value, then a true state primitive, for instance, is generated by substituting all $D$ values in the test primitive with "1", $\overline{D}$ value with "0", $TD$ value with "1", and $F\overline{D}$ with 0. The true state primitive for node $d$ across the module $m3$ has the entry (1, 1) which correspond to the values at the signal lines enable and select0 for the 2–to–4 decoder shown in Figure 6.6. The cube intersection step is not required in this case because all input lines involved in the test assembly are disjoint. The assembled pattern from this process will then be (1, 1, $D$), for the signal lines enable, select0, and select1, respectively.

In order to back propagate the other pattern in the test primitive of module $m1$, the test primitive of module $m3$ at the output cone of node $d$ will map the $D$ fault logic value at node $d$. Therefore, two more patterns, (1, $D$, 1) and ($D$, 1, 1), will be

Cone(0)

| enable | select0 | select1 | out0 | out1 | out2 | out3 |
|---|---|---|---|---|---|---|
| 1 | 1 | D | D | $\overline{D}$ | 0 | 0 |
| 1 | D | 1 | D | $\overline{D}$ | $\overline{D}$ | 0 |
| D | 1 | 1 | D | 0 | 0 | 0 |

Cone(1)

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | $\overline{D}$ | $\overline{D}$ | D | 0 | 0 |
| 1 | D | 0 | 0 | D | 0 | $\overline{D}$ |
| D | 1 | 0 | 0 | D | 0 | 0 |

Cone(2)

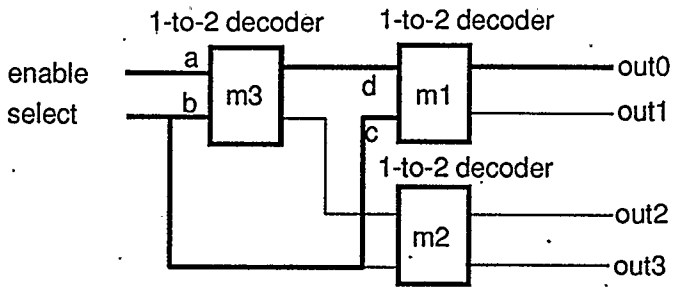
| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | D | 0 | 0 | D | $\overline{D}$ |
| 1 | $\overline{D}$ | 1 | $\overline{D}$ | 0 | D | 0 |
| D | 0 | 1 | 0 | 0 | D | 0 |

Cone(3)

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | $\overline{D}$ | 0 | 0 | $\overline{D}$ | D |
| 1 | $\overline{D}$ | 0 | 0 | $\overline{D}$ | 0 | D |
| D | 0 | 0 | 0 | 0 | 0 | D |

Figure 6.8. The test primitive for the 2–to–4 decoder.

assembled. The three assembled tests represent the test primitive for the first output cone for the parent module. This process is repeated for all other primary outputs in the parent module. The total test sets will represent the final test primitive of the 2–to–4 decoder. The test primitive for the 2–to–4 decoder is shown in Figure 6.8. This test primitive will be copied to the other 2–to–4 decoder module at hierarchy level one. Using the test primitives for modules at level one, the Test_assembly( ) procedure will assemble tests for the 3–to–8 decoder in a similar fashion to the one described above.




select = b ∩ c (cube intersection)

| a b c | enable | select |
|---|---|---|
| 1 1 D | 1 | TD |
| 1 D 1 | 1 | TD |
| D 1 1 | D | 1 |

Figure 6.9. A modified circuit diagram to illustrate the cube intersection process.

In order to illustrate the case where cube intersection of logic values on common nodes is applied, let us modify the structure of the 2–to–4 decoder circuit to the one shown in Figure 6.9. In this figure, nodes $b$ and $c$ are connected together. The corresponding modification on the assembled tests for the first primary output $PO(0)$ is also shown in figure. The second and third entries in each assembled test must be cube intersected before they are assigned to inputs of the parent module. For example, the test pattern (1, 1, $D$) will be modified to (1, $1 \cap D$) = (1, $TD$). This process is

similar to the *compare* ( ) procedure in the GATPG algorithm.

### 6.2.2 Test Length

It is important to note that the number of assembled patterns at the inputs of each module can increase dramatically as we get closer to the root of the hierarchy tree. In order to solve this problem (without degrading the test quality), a similar strategy to the one applied to the GATPG system is used during the test assembly procedures. The number of assembled tests across a module from the back propagation of a single test pattern at the output of a module is limited to a preset number. In order to ensure the test quality, this limit is applied *only* at high levels of hierarchy where the circuit complexity (interconnections) is reduced and the test length is more likely to explode. This level of hierarchy is left for the designer to decide, but it will always be in the vicinity of the Register Transfer Level (RTL) where the different major design components are easily identified. This approach also implies that the test assembly procedures will generate a minimal test length if the chip at high levels of hierarchy are partitioned to large number of modules.

In general, it can be seen that the test assembly procedures use the exact same heuristics in the GATPG system. Therefore, we do not need any other heuristics to completely generate tests hierarchically. The only difference between the modular test procedures and the low level test procedures is that they are performed at a modular level, hence, speed up is guaranteed. In order to estimate the benefits of modular test generation as opposed to gate-level test generation, a cost model for our modular test procedures is presented in the next section.

## 6.3 Modular Test Cost

This section explores the cost analysis of our modular test generation system. The total cost of test generation equals the cost of low level test generation and the cost of test assembly in the modular test system. In the low level test generation, the GATPG system does not involve backtracing, justification, or any decision making procedures. Therefore, the cost model presented by Goel [23] for gate–level single fault test generation can be applied to our approach. Hence, the low level cost model ($C_l$) for the GATPG algorithm can be simply expressed as:

$$C_l = \mu f$$

where, $f$ is the total number of faults for which the GATPG algorithm is applied, and $\mu$ is the average cost of test generation for a fault in the circuit. Since the GATPG algorithm is single phase, $f$ represent all the modeled faults in a circuit. Also, it is reasonable to consider that each fault takes an equal amount of time by the test generation algorithm to be covered since the GATPG generates tests globally, i.e., without reference to the faults it sensitizes. In other words, an *easy* or *difficult* to detect fault can be treated equally for the external observer.

In calculating the cost of the test assembly procedures, consider the circuit hierarchy shown in Figure 6.10. Modules at the leaf nodes are in alphabetic format to show modules that are identical. In order to calculate the modular test generation cost, we start at first level in the hierarchy. The test assembly cost at this level $C_h(1)$ can be expressed as:

$$C_h(1) = C_h(S_{1,0}) + C_h(S_{1,1}) + C_h(S_{1,2}) + C_h(S_{1,3})$$

where, $C_h(S_{1,j})$ represents the cost of applying the test assembly procedures to modules in the leaf nodes to generate tests for the parent modules $S_{1,j}$. This cost can be

expressed as:

$$C_h(S_{1,0}) = C_h(A + B + C)$$

$$C_h(S_{1,1}) = C_h(A + D + E)$$

$$C_h(S_{1,2}) = C_h(C + F)$$

$$C_h(S_{1,3}) = C_h(A + B + C)$$

Since identical modules are recognized by the modular test generation system, the cost of test assembly at the first level of hierarchy is contributed to the first three equations while no cost will be attributed to the module $S_{1,3}$ (since $S_{1,0}$ and $S_{1,3}$ are identical as shown in Figure 6.10).

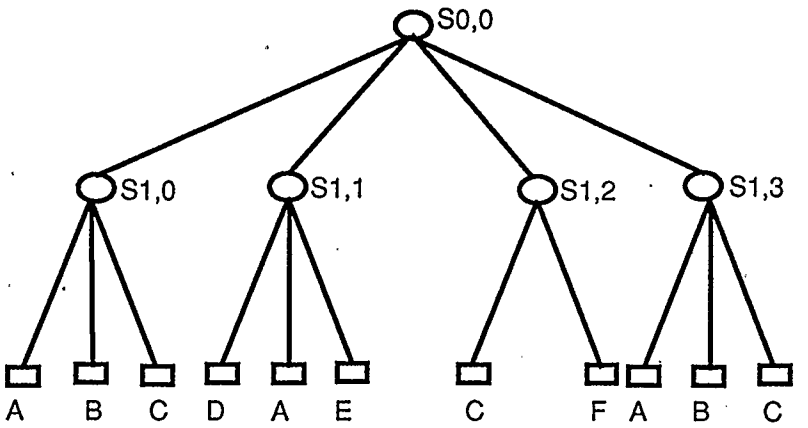



Figure 6.10. A circuit hierarchy for explaining the cost model.

The cost of test assembly at level $i$ in the hierarchy can now be expressed as:

$$C_h(i) = \sum_{a=0}^{N_{i+1}-M_{i+1}} C_h(S_{a,i+1})$$

where, $N_{i+1}$ is the number of modules at level $i+1$ in the hierarchy and $M_{i+1}$ is the number of modules that have identical description and are not processed by the test assembly procedures. This equation can be expanded to cover the total cost of modular test generation for all the levels in the hierarchy as follows:

$$C_h(t) = \sum_{i=k-2}^{0} \sum_{a=0}^{N_{i+1}-M_{i+1}} C_h(S_{a,i+1})$$

where, k is the number of hierarchy levels. The summation starts at $k$-2 because it represents the hierarchy level right above the leaf nodes.

To simplify the cost model, assume that the hierarchy of the circuit is a complete *m-ary* tree (every non leaf node has $m$ children). Also, it is a reasonable assumption that all modules in the circuit hierarchy can be evaluated for test assembly using their true and false state primitives in constant time. The highest cost occurs when all the modules at different levels of hierarchy are not identical. With these assumptions, the term $\sum_{i=k-2}^{0} \sum_{a=0}^{N_{i+1}-M_{i+1}}$ will approach and always be less than $m^{k-1}$. Combining the assumptions and the worst case condition will result in the following test cost model:

$$C_h = m^{k-1} * S_m$$

where, $k$ is the number of levels in the hierarchy, and $S_m$ is the average module test assembly evaluation cost. Let $G$ be the number of primitive gates in the circuit. The number of leaf nodes in the circuit hierarchy can be expressed as $m^{k-1}/A$, where $A$ is a constant representing the average number of logic gates in each module[1]. Therefore, $G = A\ m^{k-1}$. Similarly, the average cost of evaluating a module ($S_m$) can be expressed in terms of the number of gates in the circuit. We start by recognizing that the test assembly cost across a module depends on the number of test patterns in the test primitive of that module. Therefore, assuming that there is a fixed number of generated test patterns for each gate in a module, the number of these tests will be proportional to $g$ (the number of gates in a module). The test assembly cost increases at levels of hierarchy near to the root node. Therefore, the test assembly cost will be somewhere between the two cases:

[1]Consider the case where all the leaf nodes representing logic gates ($A = 1$). The number of leaf nodes will then equal the number of gates $G$ in the circuit, hence, $G = m^{k-1}$.

*Case 1:* considering the case of $S_m$ being proportional to $G/m$ which represents the gate count per module at the highest level for test assembly in the hierarchy. In this case, the total cost can be expressed as:

$$C_h = G^2 * s_h/m$$

where, $s_h$ is the cost of assembling a gate test at the boundary of a module hierarchically. Note that the test cost is proportional to $G^2$ which represents the lower bound on generating tests at the gate–level [23].

*Case 2:* the test assembly cost can be reduced if $m$ is large. In such case, $S_m$ is being proportional to $log_m\ G$ [41]. Therefore, the test assembly cost can be expressed as:

$$C_h = G\ log_m\ G\ s_h.$$

where $s_h$ accumulate all the constants and expresses the average evaluation cost for any gate hierarchically, and the other term represents the effect of different hierarchical representation and the gate count on the cost of modular test generation.

The actual performance of the modular test generation system is between the above two cases. In order to express the speedup factor of modular test generation over gate level test generation, the cost of gate level test generation can be expressed as $C_g = s_g\ G^2$, where $s_g$ is the average cost of generating a test for a fault at gate level of description. This model represents the lower bound (best performance) for the test generation at the gate level [23]. The speedup factor can be expressed as:

*Case 1:* Speedup $= C_g/C_h = m\ s_g/s_h.$

*Case 2:* Speedup $= C_g/C_h = s_g\ G/s_h\ log_m\ G.$

In Case 1, the speed up is constant and depends primarily on the way the chip is partitioned. The advantage of not using time consuming heuristics at the modular

level as opposed to those used at the gate level appears in the ratio $s_g/s_h$. We believe that this ratio contributes significantly to the speedup factor. For instance, considering $s_g = 10\ s_h$ and $m = 10$, a constant speedup factor of 100 can easily be attained.

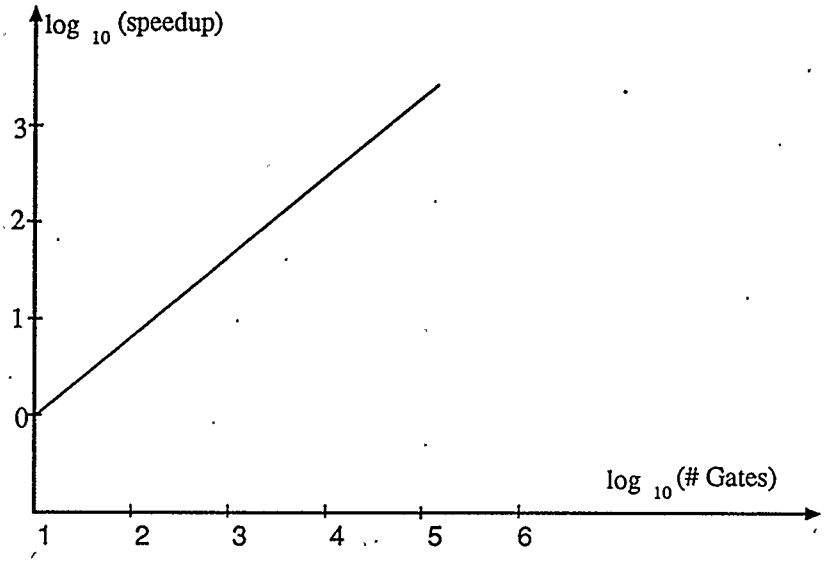



Figure 6.11. A graph showing the speedup factor for modular test generation over gate level test generation.

The speedup factor in case 2 is similar to Hyoung's speedup ratio [41] for hierarchical test generation. Figure 6.11 shows the speedup ratio of the modular test generation procedures over gate level test procedures. It is assumed that $s_g = 10\ s_h$ and $m = 10$ for the graph. The speedup is estimated as 33.3 at $10^3$ gates, 250 at $10^4$ gates, and 2000 at $10^5$ gates. These speedup factors are much higher than those reported in [41]. The graph also suggests that as the number of gates increases, modular test generation clearly shows a substantial cost reduction in test generation. The high cost reduction is due to the efficient test generation strategies at the low level of test generation. This strategy has lead to the elimination of the complicated test heuristics at the modular test generation level. Also contributed to the large

speedup factor is the modular test generation approach which uses the full potential of the hierarchical representation of the circuit.

## 6.4 Summary

We have presented in this chapter the first known *truly modular* test generation system. Our system allows for the test control activities to be hierarchical. This feature has not been presented before in the literature. We have used this feature to assemble tests hierarchically. The test primitives at one hierarchy level are used to generate tests for the upper level in the hierarchy. We have shown that such framework allows our system to be integrated efficiently in today's CAD design tools. We have also presented a new cost model based on our test generation framework. The speed up factors are shown to be substantial.

# CHAPTER 7

# TEST STRATEGIES IN MODULAR TEST GENERATION ENVIRONMENT

The impact of our modular test generation procedures on the integration of test algorithms into CAD design tools has been pointed out in the Chapter 6. It has been shown that our modular test approach provides an efficient framework for such integration, where all design activities can now be performed hierarchically. In this chapter, we will explore the impact of the modular test generation system on the test strategy selection at the chip level. Test strategy selection is the process of identifying the most suitable approach for the test pattern generation and their application to the chip. Accordingly, the test application cost and the cost of adding hardware to the chip for improving testability is determined during test strategy selection.

In this chapter, different test strategies will be explored and analyzed in the context of the modular test generation system presented in Chapter 6. The impact of each strategy on the cost of test application and added hardware will be discussed. Finally, we are proposing a new framework for the automation of the test strategy selection with the objective of minimizing the chip test cost.

## 7.1 High level strategy selection

It has been demonstrated that the chip functionality plays an important role in the test strategy selection early in the design stage. Figure 7.1 shows one classification of

test strategy selection at high level of circuit description. We classify the test strategy into *full chip* and *macro* testing strategies.

In full chip testing, the designer looks at the chip as a single testing entity. The test activities should be carried out until the root of the hierarchy tree is reached, as shown in Figure 7.1. This approach is very useful when applied to chips such as multipliers, ALUs, and other designs that do not include varieties of design styles.

In macro testing, the designer faces the problem of dealing with heterogeneous styles of design where different macros of completely different functionality exist on the chip. For example, a heterogeneous design may include a Finite State Machine (FSM), a Programmable Logic Array (PLA) structure, a RAM, and an ALU on the same chip as shown in Figure 7.1. Each one of these macros has its own test generation technique, fault models, and its unique attributes which guarantee efficient test generation and application. Therefore, it is reasonable to review the test activities at some level of the chip hierarchy where the unique test representation of each macro can be explored. The task of the test engineer is to find a way to deal with each of these macros separately and to be able to do that from the accessible nodes in the chip, i.e., the primary inputs and outputs and a small number of dedicated test pins. We will discuss these different strategies in the context of our modular test generation system and then present a new framework aimed at minimizing the total test cost of the chip.

### 7.1.1 Full Chip Testing

In full chip testing, test activities halt at the root node of the circuit hierarchy. The circuit design can be combinational or sequential. For combinational circuits,

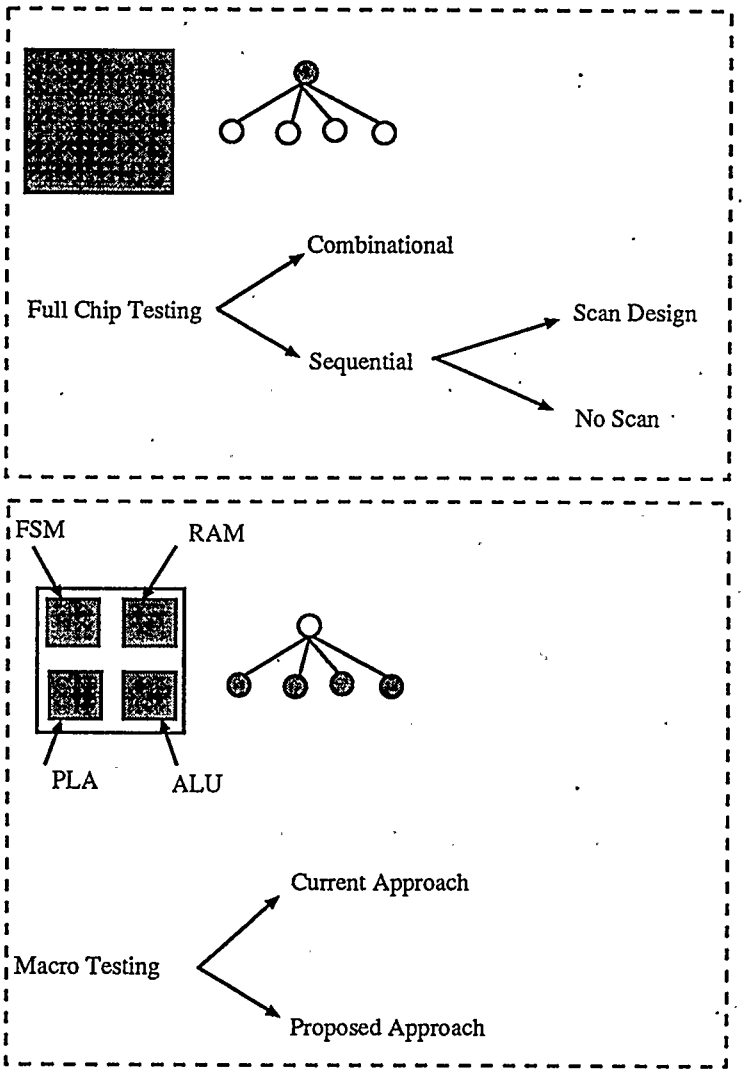



Figure 7.1. A classification of high level test strategies.

test generation can be achieved through the direct application of single fault test generation algorithm to the chip circuitry. On the other hand, our modular test generation procedures can be applied to save test time.

If the chip under test is a synchronous sequential, e.g., contains storage elements such as flip flops, there are two options as far as the test generation process is concerned. First, the designer applies a sequential test generation algorithm [36, 9, 37] to cover faults in the circuit. Sequential test generators are not popular because they take an excessive amount of time to generate tests for a circuit. Also, a fault in a sequential circuit requires a sequence of test vectors to be detected as opposed to the one vector for each fault in combinational circuits. This might result in a huge number of test vectors for even a moderate size circuit. Sequential test generation algorithms are only applied to circuits with a small number of flip flops (20 or less).

The other option which is the more likely to be taken by the designer in generating tests for sequential circuits is to apply a design for testability technique which simplifies the test activities. The most well known technique is the scan path design [57]. In scan path design, the circuit is forced into a combinational mode of operation during the testing procedure. Therefore, during test generation, the circuit may be considered combinational and our modular test generation system can be applied to the circuit as explained in the Chapter 6.

Scan path design can be explained using the circuits shown in Figure 7.2 and Figure 7.3. Figure 7.2 shows the general model for sequential circuits with only three flip flops in the circuit. The output from the flip flops represent the present state lines while the inputs to the flip flops represent the next state lines. In order to force the circuit into a combinational behavior, each flip flop is modified as shown in Figure 7.3. A flip flop can now accepts input from two different sources through the input

multiplexer. In the functional mode ($T = 0$), the flip flop inputs are connected to the circuit. In the scan mode ($T = 1$), the output of a flip flop is connected to the input of another flip flop, creating a shift register that can be scanned in and out from an external pin on the chip.

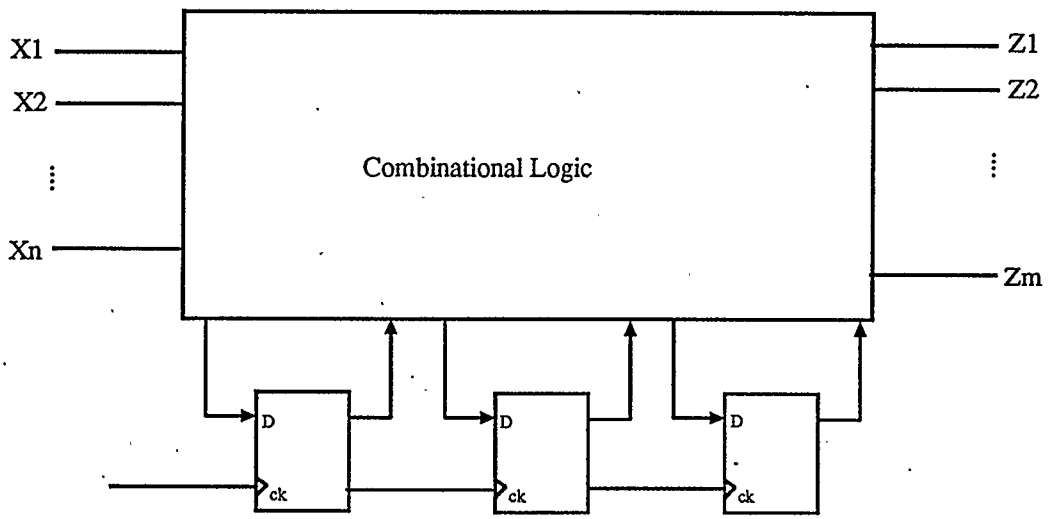
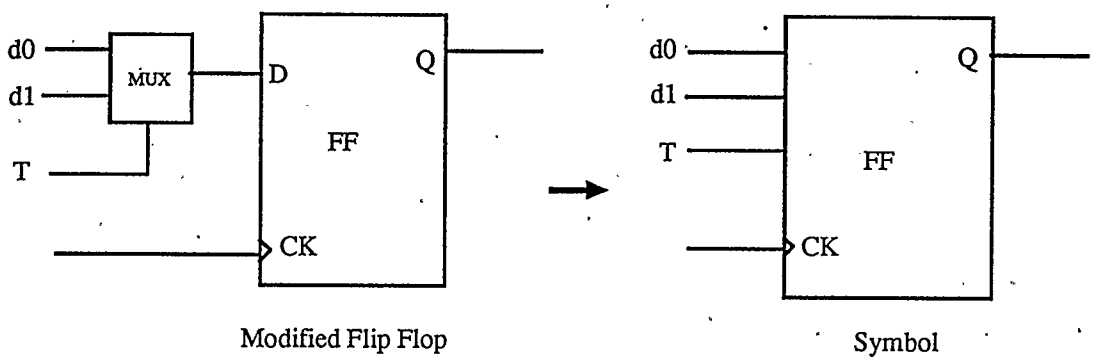



Figure 7.2. A general model for sequential circuits.

In the context of scan path designs, the test patterns generated from our modular test generation system can be applied to the circuit as follows:

- Setting $T = 1$ (Scan mode).
- Shifting the test pattern $y_j$ values into the flip flops.
- Setting the corresponding test values on the $x_i$ inputs.
- Setting $T = 0$ and, after a sufficient time for the combinational logic to settle, checking the output $Z_k$ values.
- Applying a clock signal to $CK$.
- Setting $T = 1$ and shifting out the flip flop contents via $Z_m$.



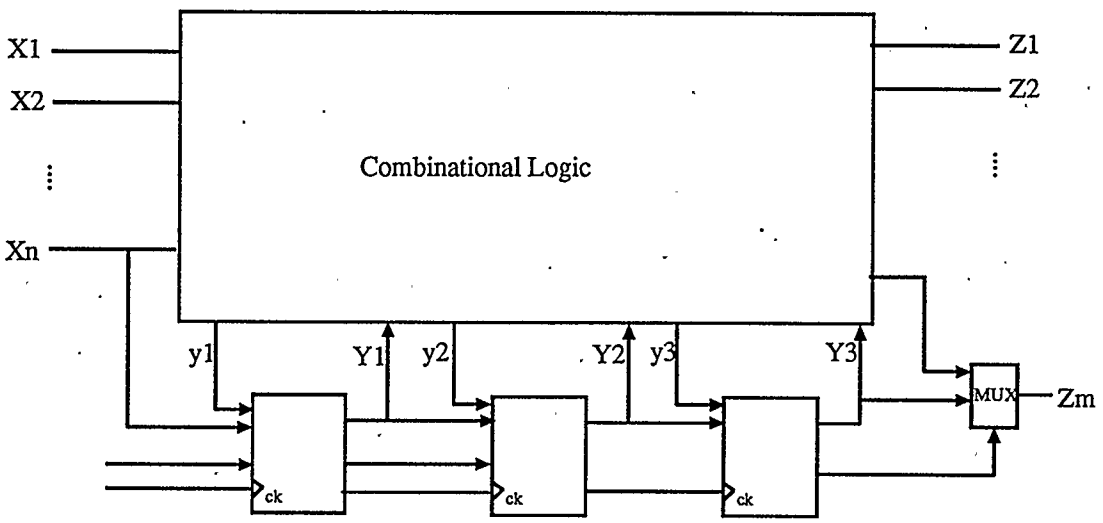




Figure 7.3. A sequential circuit modified for scan path design technique.

The logic values on the lines $y_j$ and $X_i$ are generated by the modular test generation system. The shifted output values from the flip flops represent the circuit response to the input pattern. This output can be compared to the correct response to determine if the circuit is faulty or not.

Scan path techniques proved to be very helpful in testing complex sequential circuits. The main drawback of this approach is the extra hardware added to the design of the flip flops. This hardware can be as high as 65% of the original design of the flip flop. Also, the signal going through the flip flop suffers an extra delay due to the existence of the multiplexer circuitry. There are other approaches for the modification of the circuit design to achieve the scan path modes of operation. All of them suffer from the addition of hardware into the original design. As the number of flip flops in the circuit increases, the number of clock cycles required to scan in and out the flip flop logic values increases. This poses a serious problem on the effectiveness of scan path design as the number of test patterns increases with the advancement in the VLSI technology. In order to ease this problem, macro testing has been proposed as a way of partitioning large designs into smaller macros and test each macro separately. Macro testing will be discussed in the next section.

### 7.1.2 Macro Testing

Macro testing is the process of breaking the chip design into separate macros (or modules) and applying test patterns to each macro independently. Unlike the full chip testing approach, macro testing is applied to the chip at any hierarchy level except the root level. This approach is particularly useful with heterogeneous circuit designs such as ASICs where different design styles exist on the chip. It is also helpful when the test quality of the chip is degraded due to the lack of controllability and

observability of the internal nodes in a chip, which makes it very difficult to test the internal structure of the circuit under test.

#### 7.1.2.1 The Current Approach in Macro Testing

The best known approach for macro testing was presented in [18]. This approach is considered a pure macro test technique because it adds hardware to the design to make each macro completely controllable and observable from the input/output pins of the chip. Each macro uses its own design for testing technique, while the purpose of the added hardware is to create paths between the chip pins and the inputs/outputs of each macro. In this way, test generation, test application, simulation, and all other test activities are applied to one macro rather than to the full chip.

In pure macro testing, test interface elements (TIEs) are inserted between the outputs of one macro and the inputs of next macro. A block diagram of a TIE element and its typical use in macro testing is shown in Figure 7.4. Each TIE has three modes of operation, namely, a *transparent* mode ($Y = F$), *collect* mode ($Y = F$ after one clock cycle) which allows the TIE element to collect an output value from a macro, and a *shift* mode in which the TIE elements are connected serially in a shift register fashion. It has been shown that the addition of the TIE elements would increase the design area by about 9% over the original design. The performance of the chip is not affected because the TIE elements do not introduce any significant delays to the signals in the transparent mode. A 12.4% increase in the test application time was also reported [18].

These results show that macro testing can be very helpful in testing large circuits without excessive hardware overhead. However, the real problem which faces the IC designer today is not only in the cost of adding the TIE elements to the chip but also

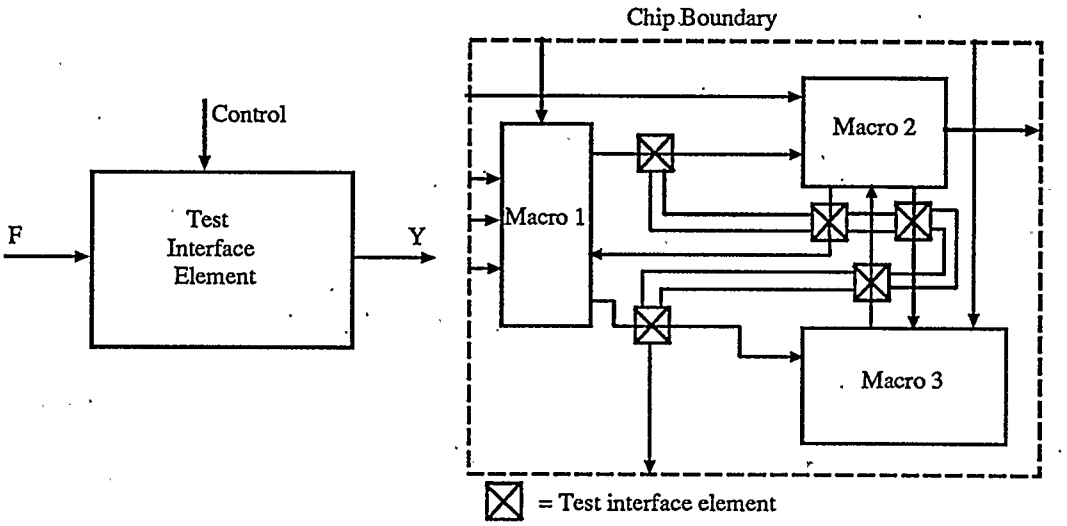



Figure 7.4. A test interface element and its application in macro testing.

to the design for test (DFT) techniques embedded in each macro. Therefore, the real testing cost and performance degradation is largely contributed to the cost of DFT techniques, such as scan path designs, in each macro. As has been mentioned before, the cost of building a scannable flip flop can go as high as 65% in area penalty, and the path delay introduced using such technique can sometimes be unacceptable. We close this argument with two different views on scan path techniques from a discussion on test economics published in [46]:

*"As the chip becomes larger, when you double the ASIC technology, you get double the number of flip flops and double the number of scan loads. So double-sized ASICs need four-sized test capability. Meanwhile, the RAM technology has only advanced by a factor of 2. So, we are faced with a situation where the requirements for scan test are outstripping the tester capability.", Richard Illman, ICL.*

*"A lot of people think that area overhead is going to cost them something, so they won't consider it even though the benefits of using DFT are still going to save them much more money.", Tony Ambler, Brunel University.*

#### 7.1.2.2 A New Framework for Macro Testing

In pure macro testing, all macros are treated equally, in the sense that if one macro is easily controllable and observable from the chip's boundary, it still has to be tested independently through the TIE elements. Also, current pure macro testing techniques cannot be applied at different levels of hierarchy. This will necessitate that the test control activities must be hierarchical. Our modular test generation system is the only known system that provides such control. Therefore, the test control activities in our modular test generation procedure can provide a suitable macro testing framework using not only software procedures but also the addition of hardware for improved testability. We will refer to our macro testing approach as a *mixed macro testing.* It is mixed because macros will be tested through software procedures and hardware addition. Accordingly, the proposed approach can be divided into two steps, namely, *soft testing* and *hard testing.* It will be shown later in this section that each one of these two steps can be used as a stand alone test strategy. Efficient test strategy with high test quality and minimal hardware addition can be achieved if the two strategies are mixed together by the designer.

#### 7.1.2.3 Soft Testing

In soft testing, the modular test generation system presented in the previous chapter will be applied to generate tests for the entire chip using the test primitive of each macro. In order to achieve this goal, each macro must be forced to have a combinational behavior, i.e., all flip flops in the design should be disconnected during the test application time. This requirement will be provided through the hard testing step. This process is similar to the full chip testing approach discussed in the previous section.

It is also assumed that each macro has its test set in the same format as the test primitives generated by our modular test generation system. Accordingly, the modular test generation procedures can be applied up to the chip level of hierarchy. The assembled tests from these procedures can then be simulated to determine the fault coverage for the chip. If the fault coverage is satisfactory, the designer may stop the test activities at this point.

If the assembled tests at the chip level do not provide an adequate fault coverage, the designer should perform preprocessing analysis before going on to the hard testing step in which hardware additions are necessary. The purpose of this analysis is to determine which macros have poor test quality. These macros will be the target for hardware addition in the hard test step. This is similar to the analysis at any other level in the circuit hierarchy during the application of our modular test generation procedure. It simply generates an estimate on the number of covered faults in each macro. Once a macro with poor test quality is identified, the designer should look at alternate ways to design that macro to improve its test quality at the current level in the circuit hierarchy. If this is not possible, the last resort would be to apply the hard test step on that macro by adding extra hardware to increase its test quality. It should be noted that our test approach can perform this process hierarchically, i.e., the test quality analysis and design modification are performed at one level in the hierarchy without reference to the underlying levels in the hierarchy.

#### 7.1.2.4 Hard Testing

Hard testing is performed if one of the following conditions arises.

- The soft testing step cannot be performed because of the lack of test primitives for some of the modules on the chip.

- The soft testing step is performed but did not provide adequate test quality for some of the macros. In order to reduce the cost of adding hardware to the design, only these macros will be targeted by the hard test step.

In the first case, the chip test assembly through modular test procedures will not be possible because not all the macros retain test primitives. Accordingly, our test generation system can be used in generating tests for some macros on the chip. The designer can choose other test solutions for the remaining macros. Then, a pure macro test approach such as the one reported in [18] can be used to test each macro independently.

In the second case, soft testing is applied to the macros on the chip in the same manner described in the Chapter 6. If the fault coverage after test assembly is not sufficient, then, the macro test approach should be applied to a subset of macros on the chip. These macros are the one that include most of the uncovered faults on the chip. It must be noted that, at this stage, the fault coverage obtained by applying the modular test generation system will equal that of applying single fault test generation at the gate level of the chip. Also, the fault coverage is typically over 90% and only a number of faults on the order of tens need to be covered by the macro testing approach. It would then be a waste in silicon area if we apply macro testing at a high level of abstraction to cover a number of localized faults in a large macro. Instead, we propose a different test strategy in which the circuit hierarchy is used to determine the lowest level of hierarchy at which all the uncovered faults are explored and then apply the macro testing strategy at that level.

As an example, in Figure 7.5, a macro has four uncovered faults. If these faults are traced to a lower level of hierarchy, two sub–modules will exist which carries this

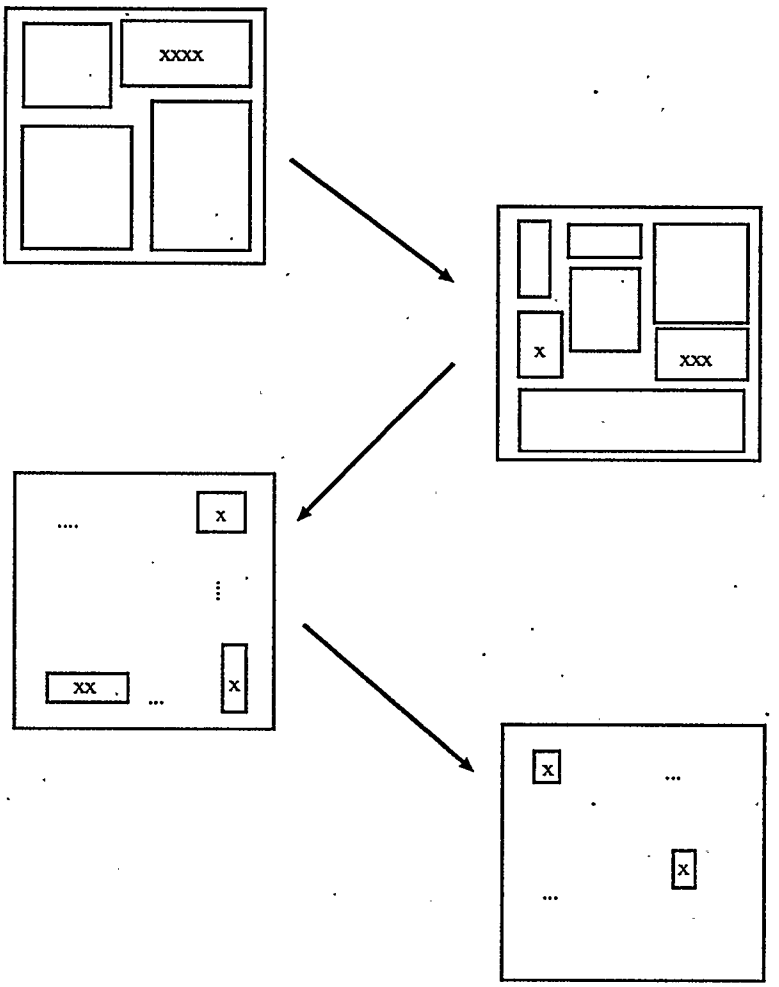



Figure 7.5. An example showing how can we determine the lowest level in the circuit hierarchy at which macro testing is applied.

set of faults. Again, tracing back the set of faults at a lower level of hierarchy reveals that three sub–modules with the fault set. If we further moved to a lower level in the hierarchy, part or all of the fault set may disappear because the subcircuit in which these faults first appeared does not exist any more. Therefore, the level at which the three modules were created is the lowest level of hierarchy at which the fault set exist. Our strategy is to apply the macro testing technique for these three modules. The bus width and the number of signals passing through these modules is much less than that at the upper macro level. When these signals are interfaced with other macros through the TIE elements, they will not cause any routing problem and the added hardware will be minimal. Therefore, minimal routing area and hardware additions are achieved using this technique. We have to emphasize here that such an approach would have not been possible without our modular test assembly procedures which cover most of the faults in the macro under test and allowed for less hardware addition at a hierarchy level other than the macro level.

## 7.2 Summary

In this chapter, we have explored the different test strategies at the chip level within the context of our modular test generation algorithm. We explained the use of our test system with each of the presented test strategy. We have also explored the macro testing approach which offers an elegant framework for test generation and application. A new test strategy for macro testing, based on our modular test generation system, has been presented as well. It is shown that this strategy would results in minimal hardware addition during macro testing.

# CHAPTER 8

# CONCLUSIONS

Most of the problems in VLSI system design are very complex and sometimes intractable. Test generation and application is one of the most challenging problems in VLSI, not only because it is complex but also because of the cost associated with it. With the advances in technology, testing is getting harder and more costly.

This thesis describes a modular/hierarchical test generation approach for the general class of VLSI circuits. We have built, from the ground up, a framework and an implementation of an automatic test pattern generation system which guarantees the full automation and integration of test activities in today's CAD tools. This integration is possible because the test activities in our framework is hierarchical.

The first task was the definition of what could be a truly modular test generation system. This had lead us to the conclusion that the existing systems fail in providing the requirements for truly modular test systems. The main reason was that many heuristics are used to solve the test problem at different levels of hierarchy. Therefore, it was very important to define the test interface at different levels of hierarchy. The representation of this interface is manifested in the way we characterize our test primitives. We have defined the test primitive of a module as a set of patterns which carries test and functional information.

The next task was to implement a test generation system which can generate such information in the test primitive. We described a novel framework for test generation at the gate level of description which achieves this purpose. This system is based on a new framework, namely, global test generation. This framework has enabled us of generating the required test primitives. In order to extend the efficiency of our global test pattern generation system, we have modified the test system to generate tests with implicit multiple fault coverage. Multiple fault testing is much more difficult than single fault testing. Our test generation framework is used successfully in solving this problem.

The description of a novel approach for modular test generation is presented. This system is based on test assembly procedures which use the test primitives and a description of modules interconnections to generate tests at higher levels in the hierarchy. We have shown that no extra information or heuristics are needed to achieve this purpose. Consequently, truly hierarchical test system can be built with minimal programming efforts and less memory requirements.

Finally, test strategy selection at the chip level was reviewed in the light of our new test framework. We have discussed the different test scenarios which might face the designers today. We have put more emphasis on macro testing technique because it has the potential of testing large varieties of complex and heterogeneous circuits. Since this approach requires the addition of hardware to improve the system's testability, we have proposed a new approach in which minimal hardware can be added to the original design.

## Future Work

As with most research, this thesis raises more questions than it answers. These are some of the areas which need further study.

**Efficiency:** There are a number of improvements that can be made to the test system to make it more efficient. An area which obviously needs to be investigated is that of minimizing the test length generated by the test procedures. Although our test system generates tests with a 100% fault coverage in a very small run time, it still suffers from a larger test length over other approaches. The problem with large test length is that it takes so much time to apply these tests in the test equipment and it also consumes a large amount of the tester's memory. We think that the test length may be minimized by applying interactive fault simulation program during test generation.

**DFT Improvement:** Design for testing techniques have come to the point where their existence became a hurdle for fast and efficient designs. Scan path techniques, for example, are now attacked because of the huge cost associated with them. It takes so much time to serially scan in and out the scan path structure. In fact, this is one of the reasons that lengthy test sets are not desirable.

Area penalty, performance degradation, and long test application time are a few reasons for the reluctant use of scan path techniques. We believe that the separation of the test attributes of a design from its functional attributes will solve some of the DFT problems. Currently, one might think of DFT circuitry as *embedded* into the design. For instance, during test application, the system must switch between a test mode and a functional mode. Consequently, this framework puts some constraints on the way test generation is carried out. We should search for a new technique

which allows the parallel application of test patterns to the chip so that the test application time is minimized. The separation of the test circuitry from the functional circuitry will minimize performance degradation. It will also separate the tester's clock from the system's clock and thus eliminates the need for clock synchronization which contributes to the complexity of the test application process.

**MCM Testing:** Multi–Chip Module (MCM) testing is one of the most challenging problems in VLSI design. It can be seen that our modular test generation system combined with the macro testing approach are very suitable to apply to this problem. The modular test system should provide the test set for each module, while the macro test approach will provide access to each module in the MCM. Because of the complexity of the MCM circuitry, we expect that the test length and the test application time will be the limiting factor in applying these techniques to the MCM testing problem. So, it is the solutions of the above problem that will lead to an efficient implementation for an MCM testing technique.

**Partial Scan:** We have discussed the application of our test generation system to the full scan technique in which all the flip flops in the design are connected as a shift register in the test mode. Partial scan is another technique in which a subset of flip flops are connected in the scan chain. The flip flops for the scan chain are selected in such a way that the extra DFT area is minimized while still sufficient controllability and observability are guaranteed to test the circuit with a desired fault coverage.

The existence of unscanned flip flops poses a problem during the test application phase. The order of applying the test bits in a single pattern will depend on the location of the unscanned flip flops in the design. We need to extend the domain of our test generation system to this type of design. This requires the addition of delay information into the test primitives. For instance, each time an unscanned flip flop is

encountered, a delay unit should be added to the sensitized path.

**Sequential ATPG:** Our test generation system is based on combinational test generation procedures. We would like to extend our system to solve the sequential test generation problem. We believe that this can be achieved by interfacing the GATPG algorithm with sequential justification and differentiation procedures. These procedures ensure that each generated pattern from the GATPG system is justified from the reset state. The purpose of the differentiation procedure is to ensure that if a fault is propagated to a next state line, it will be rerouted to one of the primary outputs.

# REFERENCES


[1] V. K. Agarwal and A. S. F. Fung. Multiple Fault Testing of Large Circuits by Single Fault Test Sets. *IEEE Trans. Comp.*, C-30:855–865, Nov. 1981.

[2] P. Agrawal and V. D. Agrawal. Probabilistic Analysis of Random Test Generation Method for Irredundant Combinational Logic Networks. *IEEE Trans. Comp.*, C-24(7):691–695, July 1975.

[3] V. D. Agrawal and M. R. Mercer. Testability measures - what do they tell us? *IEEE Test Conf., Chirry Hill, Phil.*, pages 391–396, 1982.

[4] A. V. Aho, E. Hopcroft, and J. D. Ullman, editors. *The Design and Analysis of Computer Algorithms.* Addison - Wisley, Mass., 1974.

[5] P. N. Anirudhan and P. R. Menon. Symbolic Test Generation for Hierarchical Modelled Digital Systems. *Proc. International Test Conference*, pages 461–469, 1989.

[6] R. G. Bennetts, D. C. Brittle, A. C. Prior, and J. L. Washington. A Modular Approach to Test Sequence Generation for Large Digital Networks. *Digital Processes*, 1:3–23, 1975.

[7] I. Berger and Z. Kohavi. Fault Detection in Fanout-Free Combinational Networks. *IEEE Trans. Comp.*, C-22:908–914, Oct. 1973.

[8] D. C. Bossen and S. J. Hong. Cause–Effect analysis for Multiple Fault Detection in Combinational Circuits. *IEEE Trans. Comp.*, C-20:1252–1257, Nov. 1971.

[9] W. G. Bouricius, E. P. Hsieh, G. R. Putzolu, J. P. Roth, P. R. Schneider, and C. J. Tan. Algorithms for detection of Faults in Logic Circuits. *IEEE Trans. Comp.*, C-20(11):1258–1264, Nov. 1971.

[10] W. G. Bouticius. Algorithms for Detection of Faults in Logic Circuits. *IEEE Trans. Comp.*, C-20:1258–1264, Nov. 1971.

[11] J. D. Calhoun and F. Bigliz. A Framework and Method for Hierarchical Test Generation. *Proc. Int. Test Conference*, pages 480–490, Aug. 1989.

[12] S. Chakradhar, V. D. Agrawal, and S. G. Rothweiler. A Transitive Closure Algorithm for Test Generation. *IEEE Trans. On CAD*, 12(7):1015–1028, July 1992.

[13] H. COX and J. Rajski. A Method of Fault Analysis for Test Generation and Fault Diagnosis. *IEEE Trans. on Comp.*, 7(7):813–833, July 1988.

[14] H. Cox and J. Rajski. On Necessary and Nonconflicting Assignments in algorithmic Test Pattern Generation. *IEEE Trans. on Comp.*, 13(4):515–530, April 1994.

[15] H. W. Daseking, I. R. Gardner, and G. B. Weil. VISTA: VLSI CAD System. *IEEE Trans. on CAD*, CAD-1:36–52, Jan. 1982.

[16] M. W. Du and C. D. Weiss. Multiple Fault Detection in Combinational Circuits. *IEEE Trans. Comp.*, C-22:235–240, March 1973.

[17] E. B. Eichelberger and T. W. williams. A Logic design Structure for VLSI Testing. *Proc. 14th Design Automation Conference*, pages 462–468, June 1977.

[18] F. P. M. Beenker et. al. Macro Testing: Unifying IC and Board Test. *IEEE Design and Test of Computers*, pages 26–32, Dec. 1986.

[19] G. Fantauzzi and A. Marsella. Multiple–fault Detection and Location in Fanout Free Combinational Circuits. *IEEE Trans. Comp.*, C-23:48–55, Jan. 1974.

[20] L. Fisher, W. A. Rogers, M. Abadir, and H. B. Min. A Quantitative Prediction Model for Combinational Test Generation. *The Economics of Design and Test for Electronic Circuits and Systems*, pages Chap. 5.2:, Ellis Horwood, 1992.

[21] H. Fujiwara and T. Shimono. On the Acceleration of Test GenerationAlgorithms. *IEEE Trans. Comp.*, C-32:1137–1144, Dec. 1983.

[22] J. W. Gault, J. P. Robinson, and S. M. Reddy. Multiple Fault Detection in Combinational Networks. *IEEE Trans. Comp.*, C-21(1):31–36, Jan. 1972.

[23] P. Goel. Test Generation Costs Analysis and Projections. *Proc. 17th Design Automation Conference*, pages 77–84, June 1980.

[24] P. Goel. An Implicit Enumeration Algorithm to Generate Tests For Combinational Logic Circuits,. *IEEE Trans. Comp.*, C-30:215–222, March 1981.

[25] L. H. Goldstien and E. L. Thigpen. Scoap: Sandia controllability/observability analysis program. *Des. Aut. Conf., Minneapolis, Minn.*, June 1980.

[26] J. A. Hughes. Multiple Stuck–at Fault Coverage of Single Stuck–at Fault Test Sets. *Tech. Rep. No. JH85-2*, Palo Alto Research Associates, Palo Alto, Dec. 1985.

[27] J. A. Hughes. Multiple Fault Detection Using Single Fault Test Sets. *IEEE Trans. on CAD*, 7(1):100–108, Jan. 1988.

[28] W. Jone and P. Madden. Multple-Fault Testing Using Single Fault Test Set for Fanout-Free Circuits. *IEEE Trans. on Comp.*, 12(1):149–157, Jan. 1993.

[29] W. Jone and P. Madden. Multple-Fault Testing Using Single Fault Test Set for Fanout-Free circuits. *IEEE Trans. on Comp.*, 12(1):149–157, Jan. 1993.

[30] K. L. Kodandapani and S. C. Seth. On Combinational Networks with restricted fanout. *IEEE Trans. Comp.*, C-27:309–318, April 1978.

[31] B. Krishnamurthy. Hierarchical Test Generation: Can AI Help? *Proc. Int. Test Conference*, pages 694–700, Sep. 1987.

[32] K. Kubiak and W. K. Fuchs. Multiple–Fault Simulation and Coverage of Deterministic Single–Fault Test Sets. *International Test Conf.*, pages 956–962, September 1991.

[33] Tracy Larrabee. Test Generation Using Boolean Satisfiability. *IEEE Trans. Computer–Aided Design*, 11:4–15, Jan 1992.

[34] J. Leenstra and L. Spaanenburg. Hierarchical Test Assembly for Macro Based VLSI Design. *Proc. of International Test Conference*, pages 520–529, 1990.

[35] Y. H. Levendel and P. R. Menon. Test Generation Algorithms for Computer Hardware Description Languages. *IEEE Trans. Comp.*, 31:557–588, July 1982.

[36] Hi-Keung Tony Ma, Srinivas Devadas, A. Richard Newton, and Alberto Sangiovanni-Vincentelli. Test generation for Sequential Finite State Machines. *Proc. of Int. Conf. on CAD*, pages 288–291, Nov. 1987.

[37] S. Mallela and S. Wu. A Sequential Circuit Test Generation System. *Proc. of Int. Test Conference*, pages 57–61, Oct. 1985.

[38] E. J. McCluskey and F. W. Clegg. Fault Equivalence in Combinational Logic Networks. *IEEE Trans. on Comp.*, C-20:1286–1293, Nov. 1971.

[39] E. J. McCluskey, S. Makar, S. Mourad, and K. D. Wagner. Probability Models for Pseudorandom Test Sequences. *IEEE Trans. on CAD*, 7(1), Jan. 1988.

[40] Hyoung B. Min and William A. Rogers. Search Strategy Switching : An Alternative to Increased Backtracking. *Int. Conf. on Testing*, Sept. 1989.

[41] Hyoung B. Min, Hwei tsu A. Luh, and William A. Rogers. Hierarchical Test Pattern Generation: A Cost Model and Implementation. *IEEE Trans. On CAD*, 12(7):1029–1038, July 1993.

[42] S. Mourad and E. J. McCluskey. Testability of Parity Checkers. *IEEE Trans. Ind. Electron*, 36:254–262, May 1989.

[43] Brian T. Murray and John P. Hayes. Hierarchical Test Generation Using Precomputed Tests for Modules. *IEEE Trans. On CAD*, 9(6):594–603, June 1990.

[44] D. K. Pradhan. *Fault-Tolerant Computing.* Prentice Hall, 1986.

[45] J. P. Roth. Diagnosis of Automata Failures, A Calculus and A Method. *IBM J. Res. Dev.*, 10:278–291, July 1966.

[46] A D& T Roundtable. Test Economics. *IEEE Design & Test of Computers*, pages 70–77, Fall 1994.

[47] T. M. Sarfert, R. Markgraf, E. Trishler, and M. H. Schulz. Hierarchical Test Pattern Generation Based on High–Level Primitives. *Proc. Int. Test Conference*, pages 470–479, Aug. 1989.

[48] J. Van Sas, F. Catthoor, P. Vandeput F. Rossaert, and H. De Man. Automated Test Pattern Generation for the CATHEDRAL-II/2nd Architectural Synthesis Environment. *Proc. of EDAC*, pages 208–213, Feb. 1991.

[49] D. R. Schertz and G. Metze. A New Representaion for Faults in Combinational Digital Circuits. *IEEE Trans. on Comp.*, C-21:858–866, August 1972.

[50] D. R. Schertz and G. Metze. A New Representaion for Faults in Combinational Digital Circuits. *IEEE Trans. on Comp.*, C-21:858–866, August 1972.

[51] Michael M. Schulz, Erwin Trischler, and Thomas M. Sarfert. SOCRATES: A Highly Efficient Test Pattern Generation System. *IEEE Trans. On CAD*, 7(1), Jan. 1988.

[52] J. J. Shedletsky and E. J. McCluskey. The Error Latency of A Fault in A Sequential Digital Circuit. *IEEE Trans. Comp.*, C-25:655–659, June 1976.

[53] J. P. Marques Silva and Karem A. Sakallah. Dynamic Search–Space Pruning Techniques in Path Sensitization. *Proc. of the 31st Design Automation Conference*, pages 705–711, June 1994.

[54] J. Steensma, W. Geurts, F. Catthoor, and H. De Man. Testability Analysis in High Level Synthesis. *Journal of Electronic Testing: Theory and Applications*, First issue 1993.

[55] S. M. Thatte and J. A. Abraham. Test Generation for Microprocessors. *IEEE Trans. Comp.*, 1:429–441, June 1980.

[56] J. A. Waicukauski, E. A. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault Simulation for Structured VLSI. *VLSI Systems Design*, page 20, Dec. 1985.

[57] M. J. Williams and J. B. Angel. Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic. *IEEE Trans. Comp.*, C-22(1):46–60, Jan. 1973.

[58] Abdel-Fattah S. Yousif and Jun Gu. Concurrent Automatic Test Pattern Generation Algorithm for Combintational Circuits. *International Conference on Computer Design*, Oct. 1995.

[59] Abdel-Fattah S. Yousif and Jun Gu. On the Augmentation of Single Fault Test Sets for Maximal Multiple Fault Coverage. *International Conference on ASIC Design*, Oct. 1995.