TENOR: An ATPG for Transition Faults in Combinational Circuits

by

Dhawal Tyagi

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

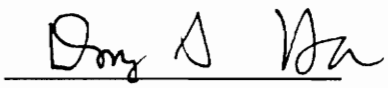
in partial fulfillment of the requirements for the degree of

Master of Science

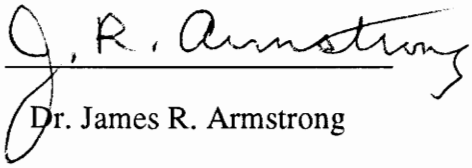
in

Electrical Engineering

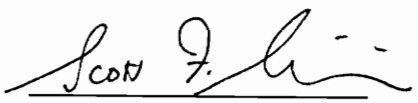
APPROVED:



Dr. Dong S. Ha



Dr. James R. Armstrong



Dr. Scott F. Midkiff

June, 1994

Blacksburg, Virginia

TENOR: AN ATPG FOR TRANSITION FAULTS IN COMBINATIONAL CIRCUITS

by

Dhawal Tyagi

Dr. D.S. Ha, Chairman

Department of Electrical Engineering

(ABSTRACT)

Delay fault testing of high speed VLSI circuits is becoming increasingly important. This thesis presents an Automatic Test Pattern Generator (ATPG), called TENOR, for transition faults. Transition faults are a special case of gate delay faults. Test generation is based on the FAN algorithm. The approach taken in this thesis is to map a transition fault into two stuck-at faults, and then generate test patterns for the stuck-at faults. A fault simulator, based on parallel pattern single fault propagation, was also developed. The problem of generating both non-robust and robust tests has been addressed. Experimental results indicate that TENOR is one of the fastest ATPGs among similar previous works, with comparable fault coverage. Experiments were also done to determine the effectiveness of stuck-at test sets and random test patterns in detecting transition faults.

Acknowledgements

I would like to thank Dr. Dong Ha for his guidance and support that he provided throughout the duration of this thesis and my master's program at Virginia Tech. I am particularly grateful for his help when I needed it most. Also, I would like to thank him for all the knowledge that he gave me, I am indebted for my life. I would also like to thank Dr. James Armstrong and Dr. Scott Midkiff for serving on my committee. Also, thanks goes to Communication Network Services which provided me with financial support during my stay at Virgina Tech and to the wonderful people there, especially Ms. Debbie Smith, Mr. David Doolittle, Mr. Robert Calley and Mr. Garry Goad (some of the finest people I have met in life).

I am grateful to Mr. Hyung Lee for error free maintenance of the computing facilities for this thesis. And lastly, my parents and my brother for their infinite love and caring (needless to say that I thank them the most). I hope that my quest for knowledge continues.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

As the speed of VLSI chips increases, testing of chips under a static fault model such as the stuck-at fault model may be insufficient. A chip which passes testing under the stuck-at fault model works at the testing clock rate, but may fail to work at the desired higher clock rate. This type of failure necessitates the timing testing of chips as well as logic testing. The timing behavior of a circuit is tested by creating a transition at the fault site (or path) and by observing the transition at an output at some specific time. If the expected transition does not occur at the observation time, a defect causing the slower operation, called a delay fault, has occurred in the circuit. To create a transition at the fault site (or path), it is necessary to apply two consecutive test patterns to the circuit under test.

Two different fault models, the path delay fault model and the gate delay fault model, are widely used to test delay faults [8], [9], [13], [16], [18]. In the path delay fault model, delay faults are associated with the structural paths of the circuit. Even if each gate is within the specified delay, the accumulated delays of the gates and the signal lines along a path may exceed the system clock period and fail the timing specification of the

circuit. If all the paths of the circuit are delay fault free, the circuit is guaranteed to comply with the timing specification. However, the testing of all the paths is impractical due to the exponential growth of the number of the paths with increase in the circuit size. This is the major disadvantage of the path delay fault model. The gate delay fault model assumes that the fault is lumped at the inputs and/or the output of the faulty gate. The main advantage of the gate delay fault model lies in the smaller number of faults, which lead to shorter processing time as compared to the path delay fault model. However, the gate delay fault model is incomplete in the sense that even if a circuit passes the test, correct timing operation is not guaranteed. This is because even if the extra delay in each gate is within the specified limits, the delays of gates on a path may accumulate to generate faulty behavior. This shortcoming may be circumvented by testing the longest path which contains the gate delay fault under consideration. In this thesis, we employ the single gate delay fault model which is computationally feasible.

The focus of the research is to develop an automatic test pattern generator (ATPG) for transition faults in combinational circuits. Transition faults are a special case of gate delay faults in which the failures at faulty gates exceed the clock period of the circuit. This implies that the testing of a transition fault can be accomplished by sensitizing any path, not necessarily the longest path. Detection of transition faults is similar to detection of transistor stuck-open faults in several aspects. A transistor stuck-open fault is a failure that turns off the faulty transistor permanently. The detection of both transition faults and stuck-open faults requires the application of two test patterns. Any test detecting a transition fault at a gate input or output also detects one or more transistor stuck-open faults for the gate and vice versa.

Lee and Ha have developed an ATPG for transistor stuck-open faults called SOPRANO at Virginia Tech [12]. SOPRANO converts a CMOS switch level circuit into an equivalent gate level circuit and the stuck-open faults into equivalent stuck-at faults. The gate level model and stuck-at faults are used to generate test patterns for stuck-open faults. Our ATPG for transition faults, called TENOR, has been developed based on SOPRANO. Like SOPRANO, TENOR converts the transition faults into equivalent stuck-at faults and then generates test patterns for the stuck-at faults.

This thesis is organized as follows. Chapter 2 presents terms and definitions and reviews previous work on delay fault testing. In Chapter 3, important features of SOPRANO and the relation between stuck-open faults and transition faults are described. In Chapter 4, key ideas and features employed in TENOR are presented. Chapter 5 presents the experimental results on the performance of TENOR. Chapter 6 concludes the thesis.

Chapter 2. Background and Related Research

In this chapter, terms and definitions necessary to understand our work are given and related work is reviewed. Section 2.1 explains the terms and definitions. Section 2.2 describes various fault models used for delay faults. Section 2.3 gives an overview of related work. Section 2.4 develops the motivation for the proposed work.

2.1 Preliminary Terms and Transition Fault Model

A *stuck-at-1* (s-a-1) *fault* on a signal line causes the faulty line to permanently assume a logic 1. A *stuck-at-0* (s-a-0) *fault* causes a permanent 0 on the faulty line. Five-valued logic {0, 1, X, D, $\overline{D}$ }is often used to describe the behavior of a circuit employing the stuck-at fault model. The *value D* designates a logic value 1 for a signal line in the good circuit and a 0 for the same signal line for the faulty circuit under the stuck-at fault model. D is also denoted as 1/0. The value $\overline{D}$ is the complement of D and is also denoted as 0/1. X designates an unspecified value.

The *D-frontier* consists of all gates whose output values are currently X, but there are one or more error signals (either D's or $\overline{D}$'s) on their inputs. Error propagation consists of selecting one gate from the D-frontier and assigning values to the unspecified

gate inputs to generate D or $\overline{D}$ at the gate output. This procedure is referred to as the *D-drive* operation. A line that is reachable from at least one fanout stem is called *bound* . A line that is not bound is said to be *free*. A *head line* is a free line that directly drives a bound line. For details on bound, head and free lines, refer to [4].

A gate g_i is a *dominator* of a gate g_j if every path from the gate g_j to the primary outputs of the circuit must pass through the gate g_i. Also by definition, every gate is a *dominator* of itself. Let a gate g_i be a dominator of a gate g_j. If there is no other dominator on the path between g_j and g_i, the gate g_i is called the immediate dominator of the gate g_j.

The delay defect size is the additional delay compared to the good circuit delay at a gate input, output or a signal line. The slack of a path is the difference between the system clock period and the propagation delay of the good circuit along the path. If the accumulated defect sizes along a path is greater than the slack of the path, the delay fault causes timing failure at the clock rate. In other words, the delay fault is detectable.

Two types of delay fault models, gate delay fault model and path delay fault model, have been widely used [8]-[10], [13], [16]. A gate delay fault assumes that the delay fault is lumped at the faulty gate at its inputs and/or output [18]. A path delay fault assumes that the effect of the fault is distributed over the faulty path [13]. A special class of the gate delay fault model is the transition fault model in which the delay defect size is greater than the clock period [20]. As the defect size is greater than the slack of any path, a transition fault can be detected by sensitizing any path to a primary output. In this thesis, we consider transition faults. There are two types of transition faults: *slow-to-rise* (STR) faults and *slow-to-fall* (STF) faults. A slow-to-rise fault causes a slower transition

to logic 1 from logic 0. A slow-to-fall fault causes a slower transition to logic 0 from logic 1.

Suppose that a delay fault is sensitized to a primary output along a path P. If an input of a gate on path P is not on the path P, the input is called an off-path sensitizing input. For example, the inputs i and k in Figure 2.1 are off-path sensitizing inputs, but the input j is not.

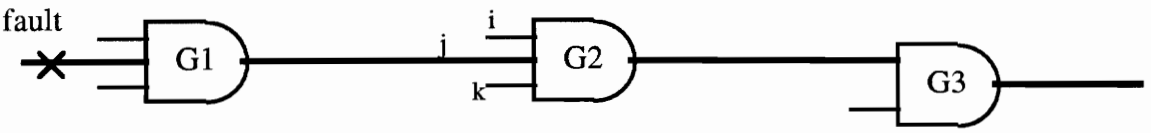


Figure 2.1. Off-path sensitizing inputs.

2.2 Detection of Delay Faults

Detection of delay faults requires application of a pair of patterns for each fault. The first pattern T_1 is called the initialization pattern and the second pattern T_2 the test pattern. The initialization pattern creates the necessary initial value for a given transition and the test pattern propagates it to one or more of the primary outputs. In Figure 2.2, the Input Clock latches the inputs and the Output Clock latches the outputs of the combinational circuit. The clock interval (t_2 - t_1) is the clock period of the combinational network. For correct operation, the delay of the combinational network should be less than the clock interval. The initializing vector T_1 is applied at time t_0 and the test vector T_2 is applied at time t_1. The output is sampled at time t_2. In the presence of a transition fault, the delay of the combinational network is greater than the clock interval. Hence, a

faulty value, which is the output for the intialization pattern, is latched in the output latch at time t_2.

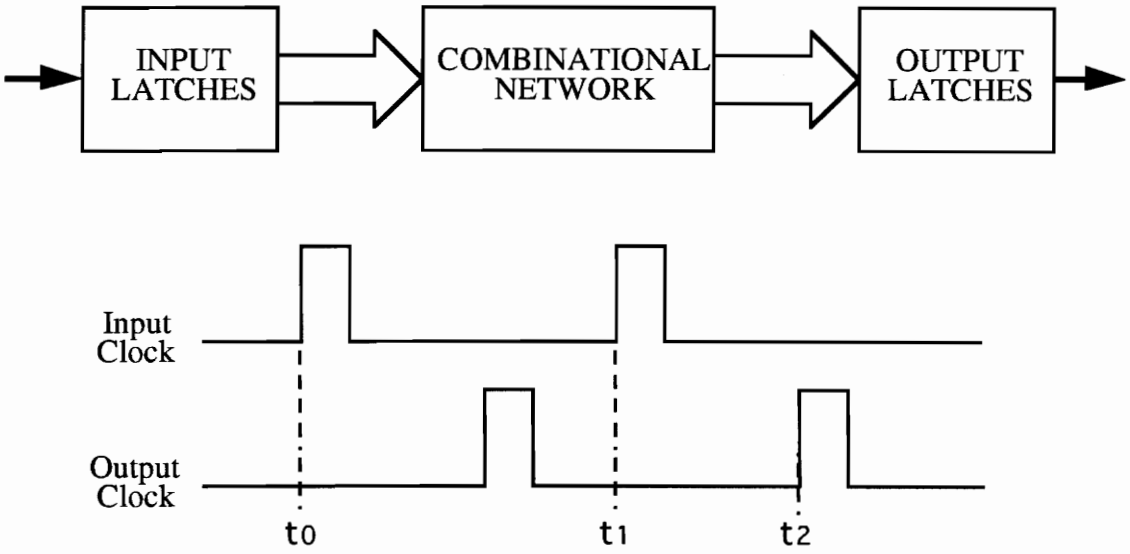


Figure 2.2. Illustration of a two pattern test.

Detection of transition faults can be easily understood by considering a simple example of an AND gate, as shown in Figure 2.3. Consider an STR fault at the input A. The initialization pattern, AB = 01, sets the input A to 0, and the test pattern, AB = 11, creates a rising transition at input A and propagates it to the output of the gate. If the output is 0 at the sampling time, the STR fault is detected.

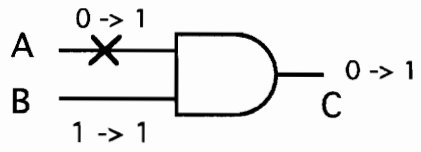


Figure 2.3. Detection of transition faults.

Detection of transition faults can be classified into three categories depending on the robustness of the test:

- non-robust test,
- robust test,
- hazard-free test.

For the non-robust test, detection of a fault can be invalidated by delays in other part of the circuit, while for the robust test detection of a fault is valid independent of delays for the rest of the circuit. If a test avoids hazards on the transition propagation path, it is called a hazard-free robust test, in short, a hazard-free test. The three test modes are illustrated in the following. Delays for both falling and rising transitions for NOR gates are assumed to be 1 unit for fault free circuits in the illustrations. Figure 2.4 shows a non-robust test for the STR fault of defect size 2 time units at the output i of gate G3.

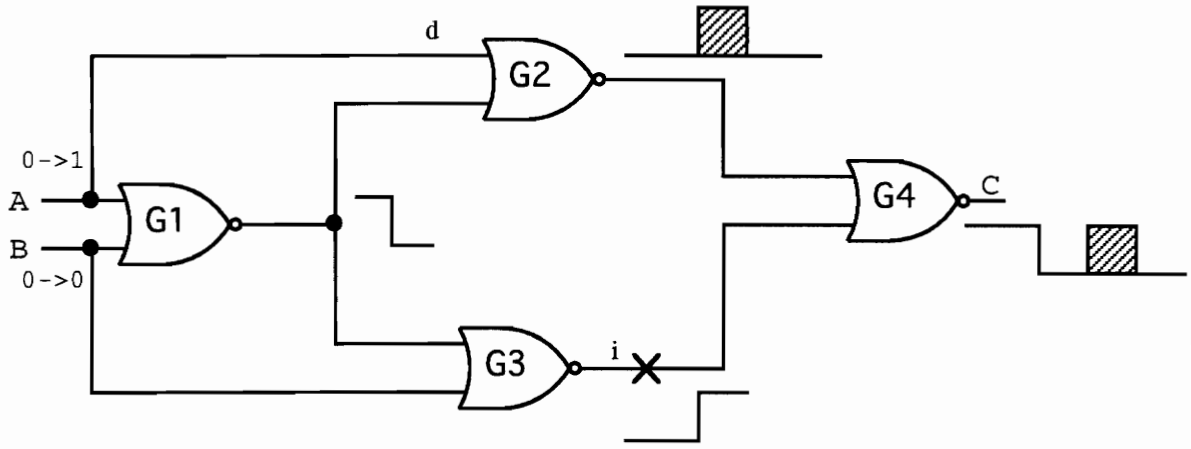


Figure 2.4. A non-robust test.

A rising transition on line **i** is propagated to the primary output C in the figure. Consider the tests, AB = 00 followed by 10. The first pattern initializes line **i** to logic 0

and the second pattern creates the desired rising transition on **i**. This transition is then propagated to the primary output C, hence the fault is detected. Figure 2.5 shows the timing diagram for the detection of this fault.

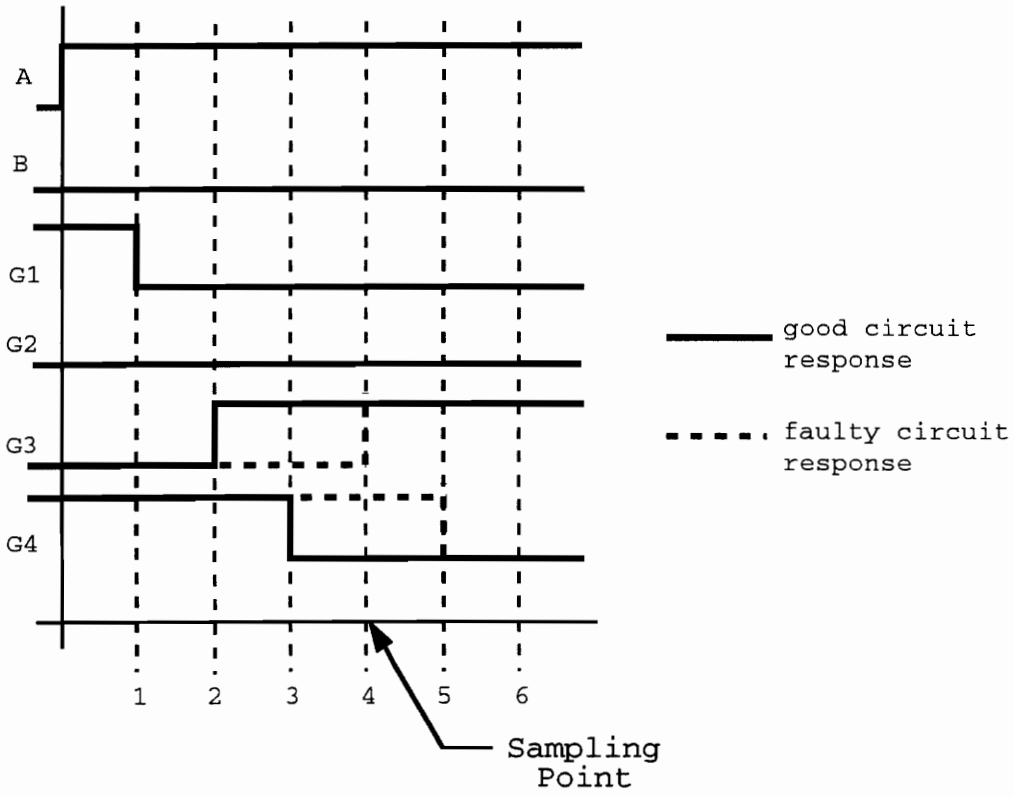


Figure 2.5. Timing diagram for primary inputs and gate outputs in the circuit of Figure 2.4.

However, an extra delay in the path A-G2-G4 can invalidate the test. Suppose the input d has an extra delay, as shown in Figure 2.6, such that transition at d occurs at 2.5 time units later. This causes a static-zero hazard at the output of the G2 gate. The faulty output of gate G4 in this case, is the same as the good output at the sampling instant. Hence the test fails to detect the STR fault on line i under the presence of an extra delay

in the other path of the circuit. Hence the test AB = 10 followed by 10 is a non-robust test for a STR fault on line **i**.

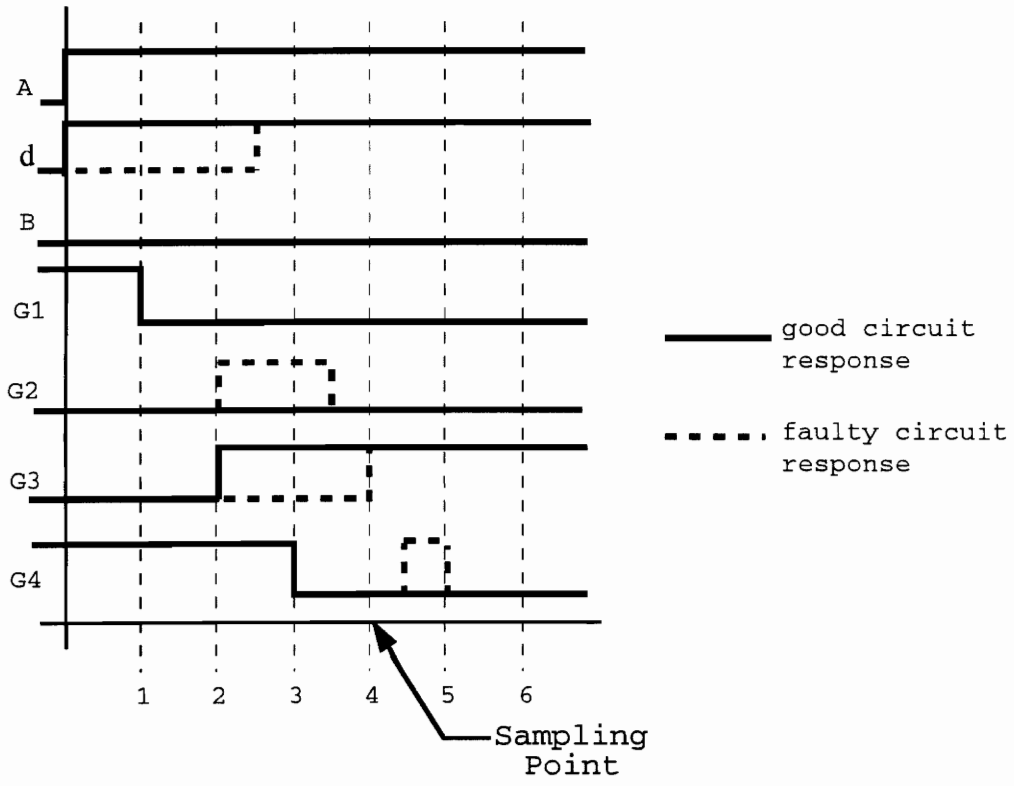


Figure 2.6. Timing diagram for the circuit of Figure 2.4 with extra delay.

Now consider the STF fault on line A as shown in Figure 2.7. One of the test patterns detecting the fault is AB = 10 followed by 00. The fault is propagated through path A-G1-G3-G4-C. Figure 2.8 shows the timing diagram for the detection of this fault.

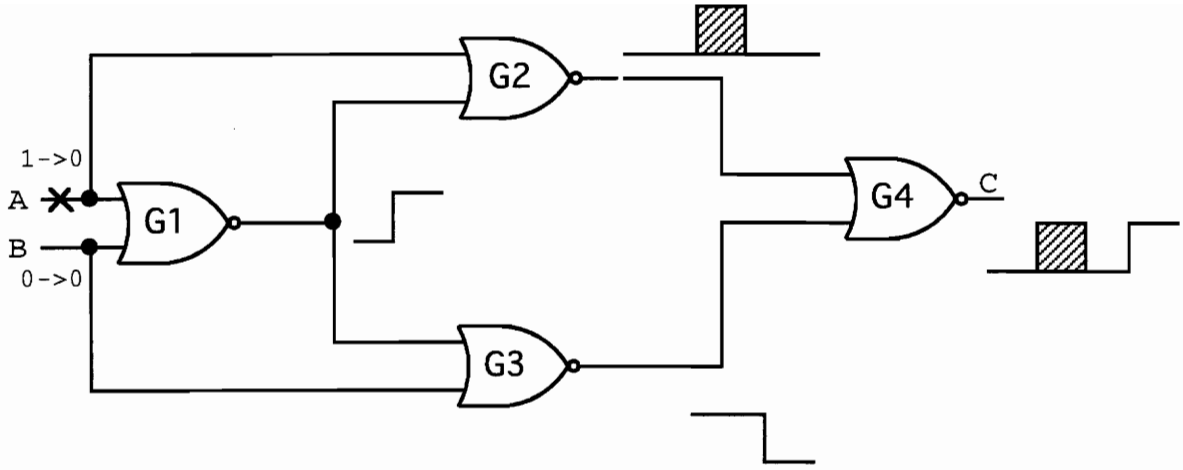


Figure 2.7. A robust test.

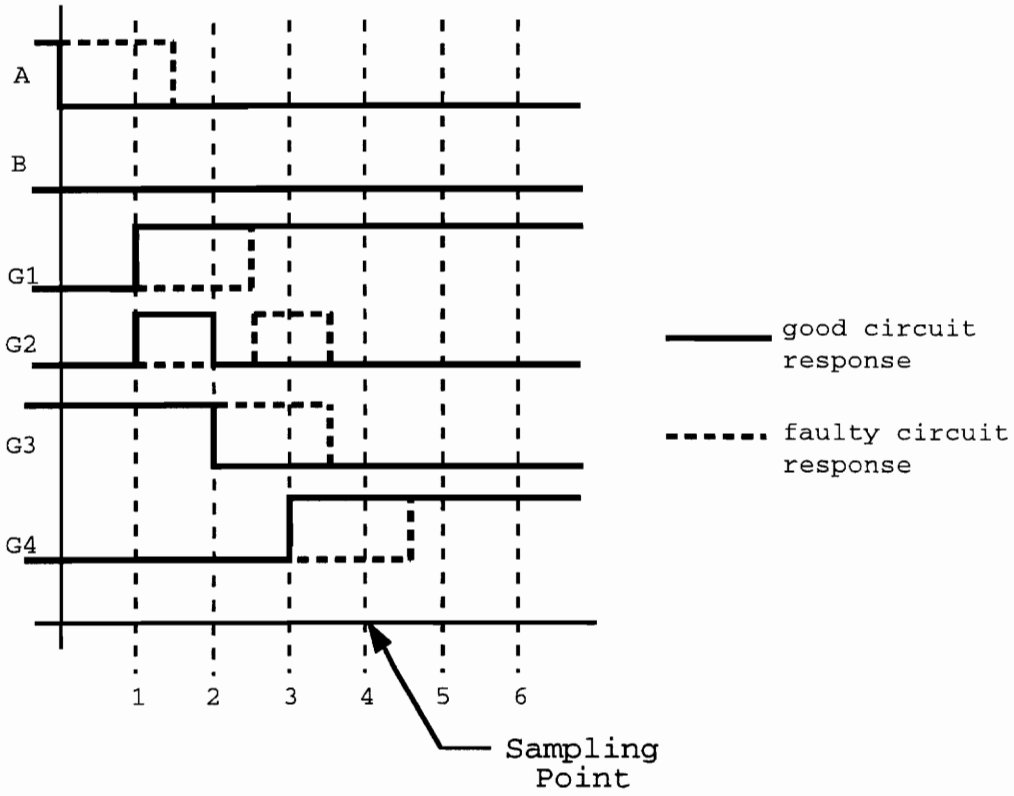


Figure 2.8. Timing diagram for the circuit of Figure 2.7.

This test will not be invalidated by any extra delays in the rest of the circuit other than the propagation path, hence, the test is robust. However, it is still not free from hazards. Suppose, the output of gate G2 has an extra delay, such that the transition is delayed by 1.5 time units. This creates a dynamic 1 hazard at the primary output C, but the test still detects the STF fault on line A. Figure 2.9 shows the timing diagram for this test with an extra delay at the output of gate G2.

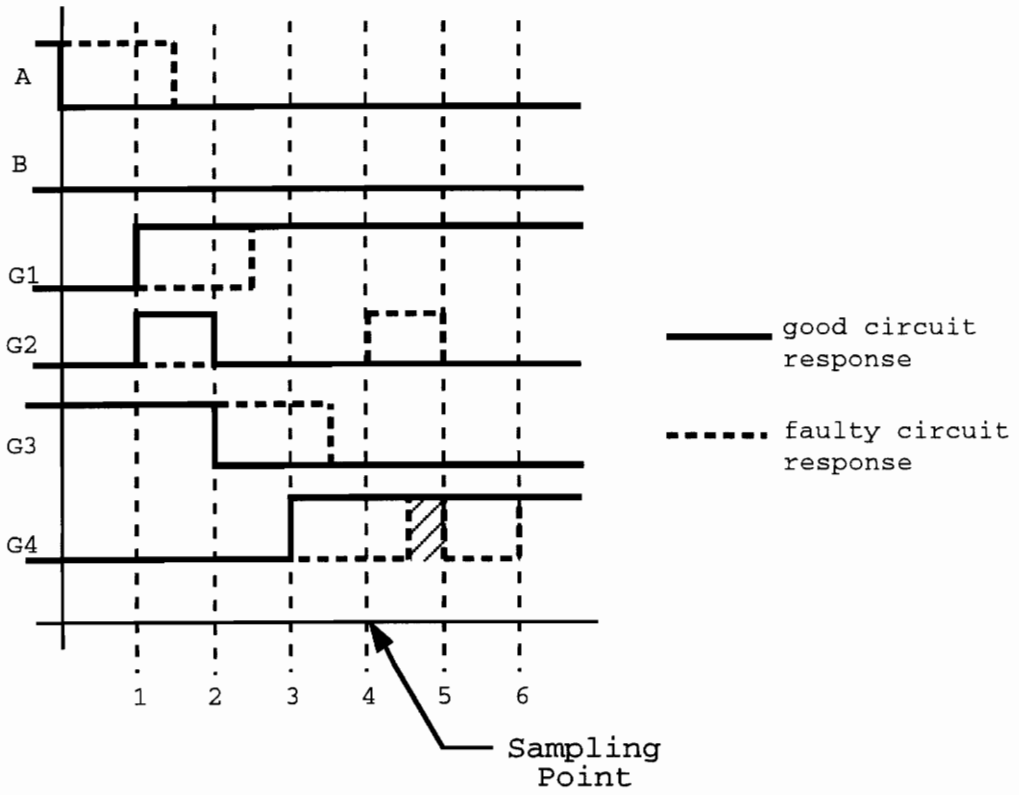


Figure 2.9. Timing diagram for the circuit of Figure 2.7 with extra delay.

Now consider the test patterns AB = 10 followed by 11 for the same STF fault on line A, as shown in Figure 2.10. The fault is now propagated through path A-G2-G4-C and the test completely avoids hazards. Hence, the test is hazard-free robust. In this

thesis, the test generation problem for only non-robust tests and robust tests has been addressed.

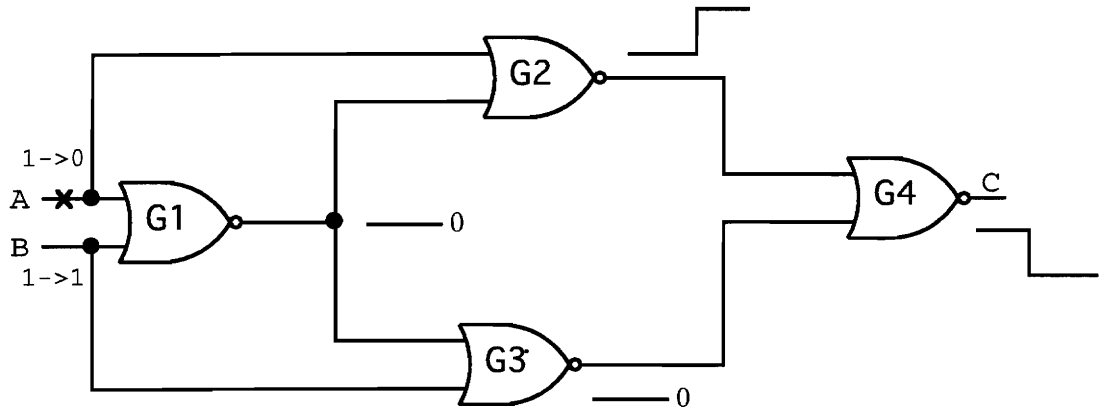


Figure 2.10. A hazard-free robust test.

2.3 Review of Existing Test Generation Methods

In this section some relevant ATPGs for both gate delay faults and path delay faults are reviewed. First, three ATPGs for gate delay faults are presented and then an ATPG for path delay faults is discussed.

Pramanick and Reddy [18] developed an ATPG for gate delay faults based on a nine-valued logic system. For the generation of robust tests, the earliest arrival time (EAT) of a transition is calculated and then this transition is robustly propagated using

the concept of a "tagged" transition. For all the leads, which are neither in the input cone nor in the output cone of the faulty lead, the EAT is assigned as -∞. So, only the transitions that are in the input cone or the output cone of the faulty lead can have non-negative EAT values. These are called "tagged" transitions. Two methods for finding the EAT have been described, namely an optimistic method and a pessimistic method. The optimistic method assumes that while propagating the transition through a gate, the transition at the output of the gate is controlled by the transition itself and not by the hazards, if present, on the other inputs of the gate. While for the pessimistic method, the EAT is calculated based on the gate type and the signals on the off-path sensitizing inputs. The aim of the two methods is to obtain a bounded value for the fault detection size. The pessimistic method gives the upper bound and the optimistic method gives the lower bound. Both methods are used to create a transition at the faulty site and to propagate the transition. The test generation is based on PODEM. Experimental results for four ISCAS85 circuits show that the proposed method achieves the same fault coverage for both robust and non-robust test patterns for 200 randomly selected faults.

Iyengar, *et al.* developed a test generator for gate delay faults [9], [10]. Unlike the transition fault model, the modeled delay faults incorporate the quantitative effect of faults. The test generator tries to find a test along a long path, so as to detect faults with small defect size. The issue of hazard-free test generation is addressed using a nine-valued logic system. In addition to logic values, three attributes, *timing, persistence* and *propagation*, are associated with each gate input and output. The test generation takes place in two phases. Phase I, algebraic test generation, is based on the FAN algorithm [4] and is similar to conventional stuck-at test generation. The purpose of this phase is to generate a candidate test for Phase II, based on a given algebraic criteria. Phase II, numerical test generation, simulates the circuit using the test pattern obtained during

Phase I for the target fault. Phase II essentially performs the numerical calculations about the arrival and stabilization times. Some of the logic values which might have been unspecified in Phase I are assigned either 0 or 1 using the coercion buffer. The coercion buffer is used to hold those gates whose inputs are X and need to be assigned either a 1 or a 0. These values are assigned in Phase II. The justification and implication of the assigned values are done immediately. This test generator tries to find a robust test first, whenever possible.

Park and Mercer proposed a gate delay test generation system, called DTEST_GEN system [16]. To alleviate the shortcomings of the gate delay fault model, DTEST_GEN system considers gross delay testing and small delay testing separately. The DTEST_GEN system consists of two delay test pattern generators, namely GROSS_DTEST_GEN and SMALL_DTEST_GEN. Gross delay testing is essentially the same as transition faults testing in which the fault size is greater than the system clock period. For faults whose fault size is smaller than the system clock period, small delay testing is employed. The overall approach is to detect gross delay faults first using GROSS_DTEST_GEN. If an acceptable defect level is achieved, the delay testing is terminated. Otherwise SMALL_DTEST_GEN is used to achieve the desired defect level. In SMALL_DTEST_GEN, the test tries to propagate a fault through the longest structural path. The longest path information for each fault is calculated in the preprocessing stage. First, it performs a gate-level timing analysis via a functionality check. The aim of the timing analysis is to find whether the longest structural path is a functional path or not. All mandatory values are obtained using heuristics such as static learning and dynamic unique sensitization. The mandatory values are implied in both forward and backward direction. If the longest structural path is not a functional path, the next longest path is considered. The SMALL_DTEST_GEN algorithm is based on PODEM [7]. The test generation algorithm selects paths whose delay is greater than a certain threshold value.

After the test is generated, the propagation delay of the path is calculated. The propagation delay is used to find the detectable delay fault size and a metric for the delay test grading.

Recently, another ATPG for gate delay faults, called DELTEST, was proposed by Mahlstedt [14]. Like Park and Mercer's approach, DELTEST also propagates faults along the longest path. The key idea is to partition a circuit into different cones. The input cone of a primary output is represented as a graph that contains all of the directed paths from primary inputs to the primary output. A transition graph for the faulty site is then constructed based on the graph. The transition graph contains only the paths from the primary input to the primary output which pass through the faulty site. Then a delay graph is constructed to keep track of all the paths from the primary inputs to the primary outputs that can possibly sensitize the given fault for a given set of values. A test is generated for each cone containing the faulty site. If the test is found then the next cone is selected, until the best quality test has been obtained. If the test generation has never been aborted, then the test generated is considered to be "optimal. " Otherwise, if a test has been found but test generation was aborted at least once, then a "good" test has been generated, but this test may not be a test along the longest path through that faulty gate. Experimental results show that the test generation time is large for circuits with a large number of paths. For example, test generation could not be completed for the circuit c6288 which has approximately 10^{20} structural paths. Fault coverages, which include both "optimal" and "good" tests, achieved for ISCAS85 circuits are low compared to Park and Mercer's results.

Lin and Reddy [13] proposed a test generator for path delay faults. The major focus of their work is generation of robust tests based on random pattern testing and

deterministic testing. The probability of robust detection of a path delay fault by random test patterns is calculated in the paper. Two necessary and sufficient conditions for robust test patterns are identified. First, the pattern should create the desired transition in the path. Second, all the off-path sensitizing inputs of a gate should have non-controlling values which propagate the transitions to the output of the gate. A CONE(O_x) of a primary output O_x is defined as the cone generated by starting from O_x and going backward into the circuit. The computational complexity of the proposed algorithm is proportional to the size of CONE(O_x), where O_x is the output of the path being tested. Several methods are also proposed in their paper to reduce the complexity at the cost of reduced accuracy. An algorithm based on PODEM [7] is used for deterministic robust test pattern generation.

2.4 Motivation for the Proposed Work

In this thesis, the transition fault model, which is a special case of gate delay fault model, has been employed. Although the path delay fault model is more attractive in terms of detecting delay faults, it is nearly impossible to completely test all the path delay faults. Table 2.1 shows the total number of structural paths and the longest path for each ISCAS85 benchmark circuit [5]. As can be seen, the total number of paths increases exponentially with the number of gates in the circuit. Moreover, finding the longest functional path passing through a given node is an NP-hard problem [16]. To circumvent these difficulties associated with the path delay fault model, most of the work employing the path delay fault model investigates only a subset of paths.

Table 2.1. Number and Longest Paths of the ISCAS85 Benchmark Circuits [5]

Circuits	Number of Structural Paths	Longest Structural Path
c432	83926	27
c499	9440	18
c880	8642	35
c1355	4173216	41
c1908	729057	48
c2670	679884	44
c3540	28676671	58
c5315	1341305	56
c6288	$\approx 10^{20}$	218
c7552	726493	49

The gate delay fault model does not encounter the above problems. Test generation with reasonably high fault coverage can be done in a reasonable time. This fact motivated us to employ the transition fault model in the proposed work. Both robust and non-robust test generation was considered in our test generator, TENOR. Robust tests that are not invalidated by gate delays in the rest of the circuit are more desirable, but it takes more CPU time to generate robust tests.

Chapter 3. Overview of SOPRANO and Stuck-open Faults versus Transition Faults

The detection of stuck-open (SOP) faults and transition faults are similar in nature. A stuck-open ATPG, SOPRANO, was used as the basis for developing TENOR, an ATPG for transition faults. In this chapter, an automatic test pattern generator for stuck-open faults, SOPRANO, is briefly described and the relation between stuck-open faults and transition faults is discussed. Section 3.1 gives an overview of SOPRANO. Section 3.2 presents the relationship between detection of stuck-open faults and detection of transition faults. Section 3.3 summarizes this chapter.

3.1 Overview of SOPRANO

SOPRANO is an automatic test pattern generator for transistor stuck-open faults developed by Lee and Ha [12]. The key idea employed in SOPRANO is to convert a switch level CMOS circuit into the equivalent gate level circuit and to map the switch level SOP faults to equivalent gate level stuck-at faults. Then a gate level test pattern generation algorithm, FAN, is used to generate test patterns for the stuck-at faults.

The overall structure of SOPRANO is shown in Figure 3.1. Test generation in SOPRANO is divided into three sessions as shown in the figure. In the preprocessing session, a given CMOS circuit is converted into an equivalent gate level circuit and SOP faults are mapped to equivalent gate level stuck-at faults. Session II consists of random test pattern testing and deterministic test pattern generation. Test patterns generated are compacted in the third session. The role of each session is briefly described below.

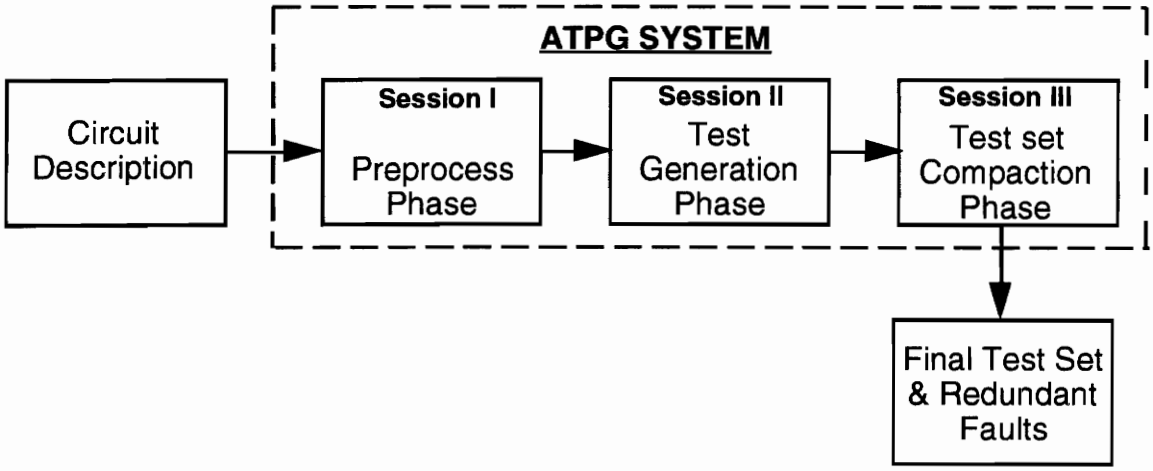


Figure 3.1. Overview of SOPRANO.

3.1.1 Preprocessing

In the preprocessing session, a CMOS switch level circuit under test is transformed into the equivalent gate level circuit. The procedure for obtaining the equivalent gate level circuit is based on the method proposed by Reddy, *et al.* [19]. The basic idea is to take only the n-type network (pull-down) of the circuit, and to find the logic function for the n-type network. The equivalent gate level circuit is then obtained

from the logic function. Since the n-type network is used, an inverter is necessary at the output of the gate level circuit.

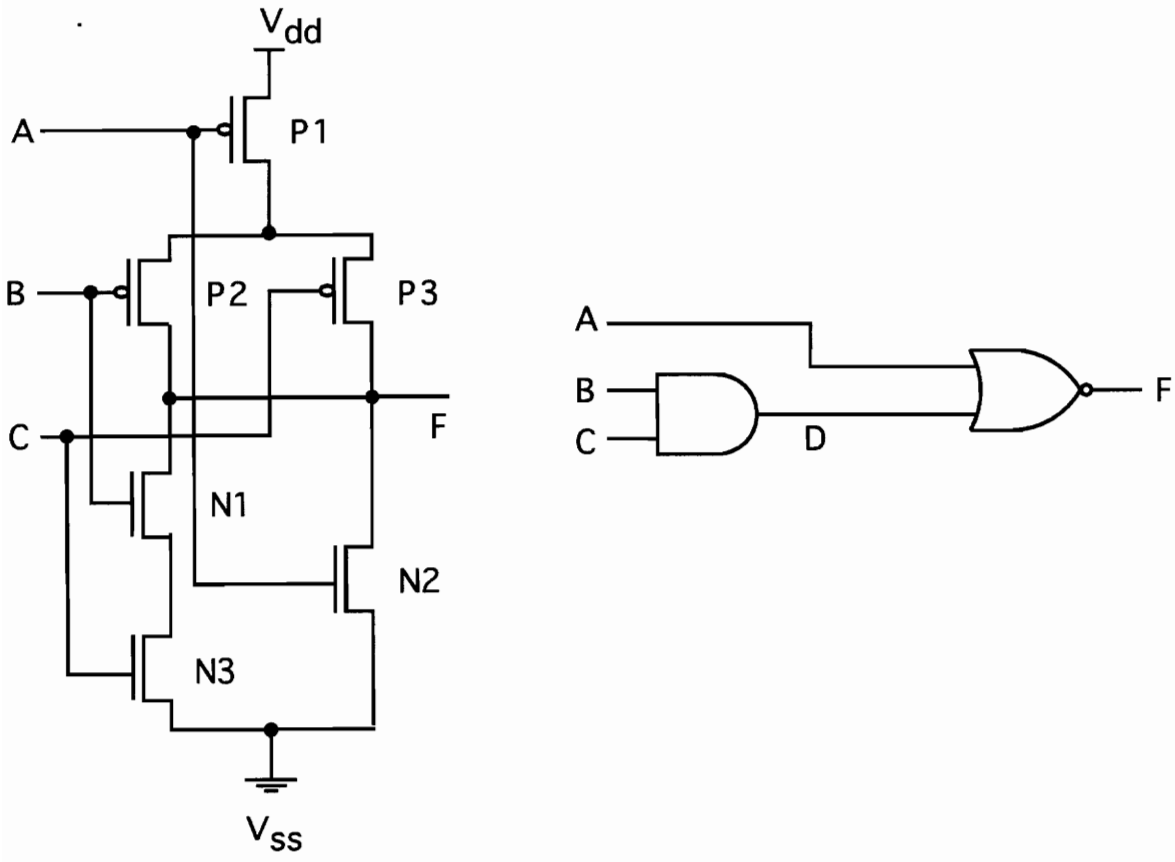


Figure 3.2. A CMOS circuit and its equivalent gate level model [12].

Figure 3.2 gives an example of obtaining equivalent gate level circuit from a CMOS switch level circuit. The Boolean logic function for the n-FET part of the circuit is F=(a+bc)'. An equivalent gate level circuit implementing the function is also shown in Figure 3.2.

Next, the SOP faults at the switch level circuit are mapped to stuck-at faults at the gate level. The following mapping definition is used. For an n-type transistor SOP fault, the equivalent stuck-at fault is the corresponding input line stuck-at-0 fault. For a p-type transistor SOP fault, the equivalent stuck-at fault is the corresponding input line stuck-at-1 fault. So a test for the corresponding stuck-at fault will also detect the SOP fault, provided the faulty line is properly initialized. It should be noted that if two stuck-at faults are equivalent, then the corresponding SOP faults are also equivalent.

3.1.2 Random Pattern Testing

In the random pattern testing session (RPT), a packet of 32 random patterns is generated, and fault free simulation is performed for the 32 random patterns. Then the circuit is fault simulated using the dominator concept, i.e. explicit fault simulation is performed only for fanout stems [16]. If the test pattern T_2 is found for a fault, it is marked as a found test pattern. All faults that are initialized by the test patterns are marked as found by the T_1 pattern. If both the test patterns, T_1 and T_2, are found for a fault, then it is removed from the fault list. A parallel pattern fault simulator is used for both fault free simulation and fault simulation. This sessions ends if the fault list is exhausted or two consecutive packets (64 patterns) do not detect new fault.

3.1.3 Deterministic Test Pattern Generation

If there are any undetected faults left after the RPT session, the test patterns for the remaining faults are generated during the deterministic test pattern generation session. The deterministic test pattern generation algorithm FAN finds the T_2 patterns of the remaining faults. If the pattern is the T_2 pattern for any other fault, the fault is marked as

found by T_2. Then the fault free simulation is done to find all the faults initialized by this test pattern. If both the test patterns, T_1 and T_2, have been found for a fault, the fault is deleted from the fault list.

3.1.4 Test Compaction

In the test compaction session, the test patterns generated are concatenated to achieve the maximum overlapping of T_1 and T_2 patterns. This is done by generating a directed graph. Vertices of the graph are the test patterns T_1, T_2 , etc. There exists an edge from V_i to V_j if the two vertices are a pair of test patterns, T_1 and T_2, detecting some fault f_k. The minimal string is then generated by traversing this graph, starting from any arbitrary node and ending at a node which has no remaining untraversed outgoing edges. This results in a minimal string of test patterns of which each sequence of (T_i,T_j) is a substring.

Using this sequence, the circuit is simulated in both forward and reverse directions to further compact the test set. After the forward simulation, each test pattern which detects a new fault is marked. If a test pattern T_i does not detect any new fault, it is discarded only if the pattern T_{i+1} also does not detect any new fault. The condition is necessary since T_i can be an initializing pattern for T_{i+1}. The remaining test patterns are divided into subsequences of test patterns. These subsequences are applied in reverse order to further identify any test pattern(s) which do not detect any new fault.

3.2 Relation Between Detection of Transition Faults and SOP Faults

Detection of transition faults is similar to detection of SOP faults since they both require a pair of test patterns to detect a fault. A pair of test patterns detecting a transition fault can detect one or more corresponding SOP faults and vice versa. In the following, we illustrate the relation using two primitive gates, NAND and NOR, and two non-primitive gates, AND and OR.

3.2.1 Primitive Gates

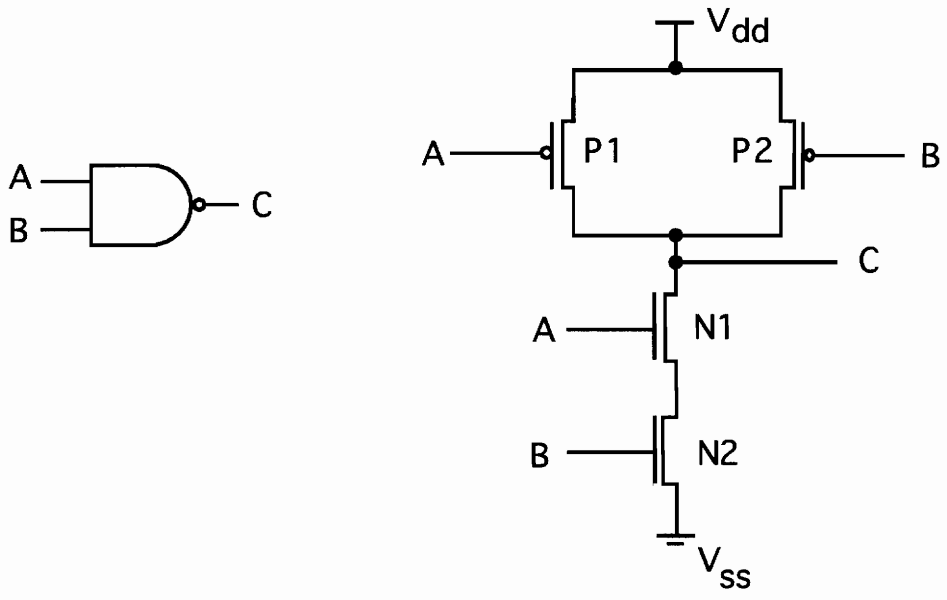


Figure 3.3. A NAND gate and its CMOS switch level realization.

Consider the NAND gate in Figure 3.3 and its equivalent CMOS realization. Table 3.1 lists all the transition faults with the NAND gate and the test pattern pairs

detecting the faults. In the table, X STR (STF) denotes a slow-to-rise (slow-to-fall) transition fault on line X. From Table 3.1, a minimal test set detecting all the transition faults is easily deduced, such as {(00,11), (11,00)}. Thus a minimal length sequence detecting all the transition faults would be {00,11,00}. Table 3.2 lists all the SOP faults for the NAND gate at the switch level and the test pattern pairs detecting the faults. Table 3.3 gives the coverage of transition faults for SOP faults.

Table 3.1. Transition Faults and Test Patterns for NAND Gate

Transition Faults	**Test Patterns (AB)**	
	T_1	**T_2**
A STR	0X	11
A STF	11	0X
B STR	X0	11
B STF	11	X0
C STR	11	(0X, X0)
C STF	(0X, X0)	11

Table 3.2. SOP Faults and Test Patterns for NAND Gate

SOP Faults	**Test Patterns (AB)**	
	T_1	**T_2**
P1 Open	11	01
P2 Open	11	10
N1 Open	(0X, X0)	11
N2 Open	(0X, X0)	11

As is evident from Table 3.1 and Table 3.2, test patterns detecting transition faults also cover SOP faults. The relation is summarized in Table 3.3. In the table, a test pattern pair which detects a transition fault in a row also covers SOP faults in the same row. A SOP fault which is marked by "*" denotes that the fault may or may not be detected depending on the test pattern pair chosen for the transition fault. For example, a test pattern pair detecting the A STF fault detects the P1 SOP fault in Table 3.3. However, the P2 SOP fault is detected only if the test pattern pair for the A STF fault is (11, 00). Note that a test pattern pair (11, 01) detects the A STF fault, but not the P2 SOP fault.

Table 3.3. Covering of SOP Faults by Transition Faults for NAND Gate

Transition Faults	SOP Faults
A STR	N1, N2
A STF	P1, P2*
B STR	N1, N2
B STF	P1*, P2
C STR	P1*, P2*
C STF	N1, N2

Following property can be derived from the covering relation obtained in Table 3.3.

Property 1: *Any set of test patterns which detects all the transition faults of a NAND gate, also detects all the SOP faults of the NAND gate.*

Proof:

Let set A = { f_i : f_i is a transition fault of a NAND gate }, and set B = { f_j : f_j is a SOP fault of NAND gate }. Let ψ be a function such that set A is the domain of ψ and set B is the range of ψ . The function ψ can be defined as ψ : A→B = {(x,y), x ∈ A and y ∈ B, if y is covered by x }. This means that if a fault in set A is detected, then one or more fault(s) in set B is (are) also detected. From Table 3.3, it can be easily seen that ψ on A is a one to many and onto mapping. Let set C = { t_i : t_i is a test pattern detecting at least one transition fault }. Let set C be such that it detects all the transition faults in set A. Since ψ on A is one to many and onto and all the faults in set A are detected, it implies that all the faults in set B are also detected. Hence, any set of test patterns which detects all the transition faults of a NAND gate, also detects all the SOP faults of the NAND gate.

Now consider a NOR gate as shown in Figure 3.4, and its equivalent CMOS realization. Table 3.4 lists all the transition faults for the NOR gate and shows the relation between the detection of transition faults and SOP faults. In the table, any test pattern pair detecting a transition fault also detects the SOP faults in the same row. A SOP fault marked "*" denotes that the fault may or may not be detected. The following property also holds for the NOR gate.

Property 2: *Any set of test patterns which detects all the transition faults of a NOR gate also detects all the SOP faults of the NOR gate.*

It is easy to show that test patterns detecting transition faults for an inverter also detect all the SOP faults of the inverter.

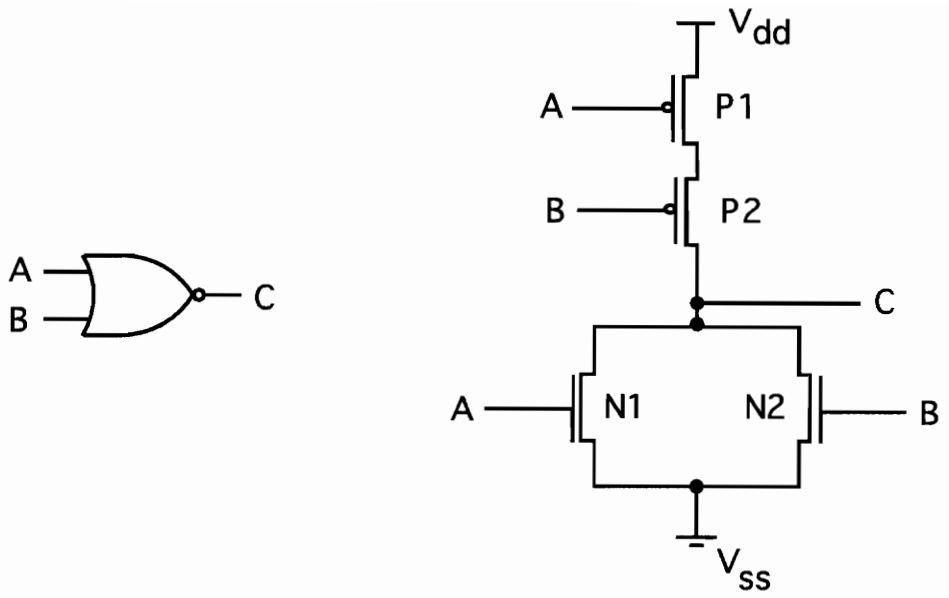


Figure 3.4. A NOR gate and its CMOS switch level realization.

Table 3.4. Covering of SOP Faults by Transition Faults for NOR Gate

Transition Faults	SOP Faults
A STR	N1, N2*
A STF	P1, P2
B STR	N1*, N2
B STF	P1, P2
C STR	P1, P2
C STF	N1*, N2*

3.2.2 Non-primitive Gates

Now consider an AND gate as shown in Figure 3.5. At the switch level it is composed of a NAND gate and an inverter. The list of transition faults for an AND gate is the same as that of a NAND gate. The test pattern pairs detecting each fault in an AND gate are given in Table 3.5. Table 3.6 lists all the SOP faults for the switch level circuit ofthe AND gate shown in Figure 3.5. From Tables 3.5 and Table 3.6, a fault covering relationship of SOP faults by transition faults can be deduced. This relationship is summarized in Table 3.7. Again all the faults that are marked by "*" may or may not be detected, depending upon the test pattern selected.

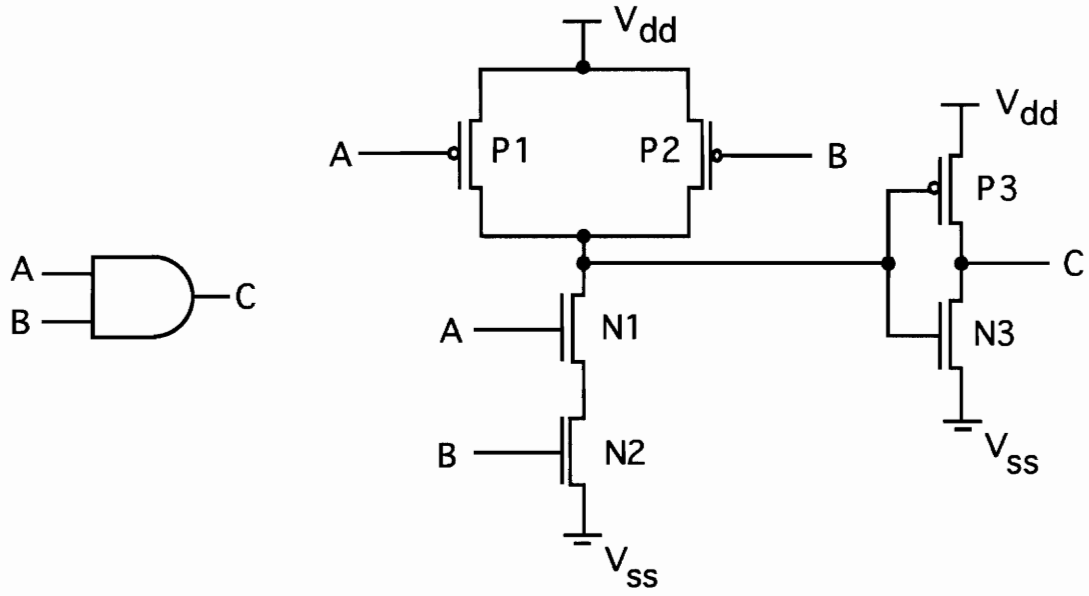


Figure 3.5. An AND gate and its CMOS switch level realization.

Table 3.5. Transition Faults and Test Patterns for AND Gate

Transition Faults	Test Patterns (AB)	
	T_1	T_2
A STR	0X	11
A STF	11	0X
B STR	X0	11
B STF	11	X0
C STR	(0X, X0)	11
C STF	11	(0X, X0)

Table 3.6. SOP Faults and Test Patterns for AND Gate

SOP Faults	Test Patterns (AB)	
	T_1	T_2
P1 Open	11	01
P2 Open	11	10
P3 Open	(0X, X0)	11
N1 Open	(0X, X0)	11
N2 Open	(0X, X0)	11
N3 Open	11	(0X, X0)

Table 3.7. Covering of SOP Faults by Transition Faults for AND Gate

Transition Faults	SOP Faults
A STR	N1, N2, P3
A STF	P1*, N3
B STR	N1, N2, P3
B STF	P2*, N3
C STR	N1, N2, P3
C STF	N3, P1*, P2*

Suppose that we define a function h similar to ψ in the proof of Property 1. An inspection of Table 3.7 reveals that the function h is not an onto function. Hence, even if all the transition faults are detected for an AND gate, all the SOP faults may not be detected. For example, consider the test sequence T = {00,11,00}. This sequence is a minimal test sequence that can detect all the transition faults for an AND gate, but fails to detect all the SOP faults. If we choose a robust test sequence T = {01,11,10,11,01}, which is not a minimal non-robust test sequence for transition faults, all the SOP faults for the AND gate as well as the transition faults, are detected.

Now consider an OR gate, as shown in Figure 3.6. Table 3.8 lists the transition faults of an OR gate along with the test pattern pairs detecting them. Table 3.9 lists the SOP faults and test pattern pairs detecting the SOP faults.

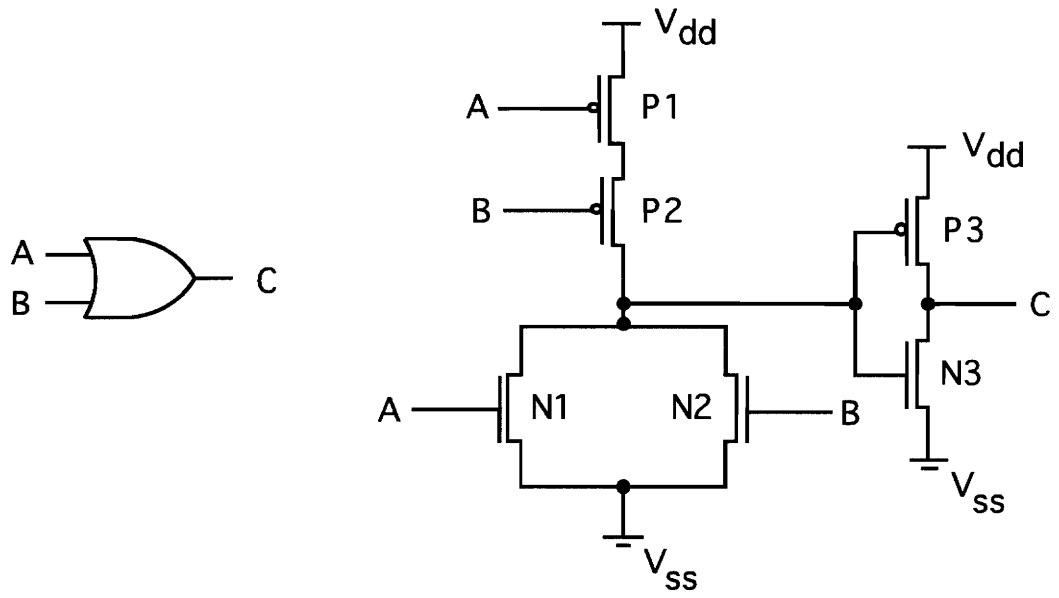


Figure 3.6. An OR gate and its CMOS switch level realization.

Table 3.8. Transition Faults and Test Patterns for an OR Gate

Transition Faults	Test Patterns (AB)	
	T_1	T_2
A STR	00	1X
A STF	1X	00
B STR	00	X1
B STF	X1	00
C STR	00	(X1,1X)
C STF	(X1,1X)	00

Table 3.9. SOP Faults and Test Patterns for an OR Gate

SOP Faults	Test Patterns (AB)	
	T_1	T_2
P1 Open	(1X,X1)	00
P2 Open	(1X,X1)	00
P3 Open	00	(1X,X1)
N1 Open	00	10
N2 Open	00	01
N3 Open	(1X,X1)	00

Similar to the case of the AND gate, there exists a minimal test set which detects all the transition faults of the OR gate, but fails to detect all the SOP faults. This is again due the function mapping from the set of transition faults to the set of a SOP faults for the OR gate, not being an onto mapping. The covering relationship for the OR gate is indicated in Table 3.10. We can find a test pattern set which detects all the transition faults but not all the SOP faults. One such test is T = {00,11,00} which detects all the transition faults, but not N1 SOP and N2 SOP faults. But, if the test set is T={00,10,00,01,00}, then all the transition faults and all the SOP faults are detected.

From the results of AND gate and OR gate, it can be inferred that a given set of test patterns for transition faults detects most of the SOP faults, but does not guarantee detection of all the SOP faults. Hence, for a circuit consisting of AND, NAND, OR, NOR and inverters, if a given test set detects all the transition faults, then it would detect most of the SOP faults also, but does not necessarily detect all the SOP faults.

Table 3.10. Covering of SOP Faults by Transition Faults for an OR Gate

Transition Faults	SOP Faults
A STR	P3, N1*
A STF	P1, P2, N3
B STR	P3, N2*
B STF	P1, P2, N3
C STR	P3, N1*, N2*
C STF	P1, P2, N3

3.3 Summary

Detection of transition faults and SOP faults are similar and closely related. The similarity and closeness motivated us to modify SOPRANO (an ATPG for stuck-open faults) [18] to develop TENOR, an ATPG for transition faults. Details of TENOR are described in the next chapter.

Chapter 4. Test Generation for Transition Faults

In this chapter, TENOR, our ATPG for transition faults, is described. Section 4.1 explains fault collapsing for transition faults. Section 4.2 presents the random test pattern generation session and fault detection algorithm of TENOR. Section 4.3 explains the deterministic test pattern generation session of TENOR. Section 4.4 describes the test set compaction algorithm. Finally, Section 4.5 describes detection of transition faults using a stuck-at test set.

4.1 Fault Collapsing

When creating a list of transition faults, the rules of fault equivalence for stuck-at faults are no longer valid. As indicated in Section 2.2, the test for a transition fault requires two successive patterns, while the test for a stuck-at fault needs only one pattern. This results in two stuck-at faults being equivalent, but their corresponding transition faults are not.

Figure 4.1 illustrates the difference. Consider the input A stuck-at-0 fault and output C stuck-at-0 fault for a two-input AND gate. These two faults are equivalent, but

their corresponding transition faults, namely input A STR and output C STR, are not equivalent. The test for input A STR fault consists of the initialization pattern, in which input A is 0, followed by the test pattern in which A is set to 1. The initialization pattern for output C STR fault can have either input A set to 0 or input B set to 0, while the test pattern remains the same. Since the initialization pattern can be different for these two faults, they are not equivalent.

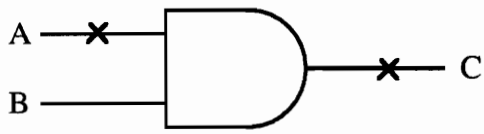


Figure 4.1. Fault equivalence of stuck-at and transition faults.

The transition fault equivalence rules are stricter than those for stuck-at faults. Two transition faults are equivalent if one of the following conditions is satisfied:

- If a gate has one input, then the input transition faults are equivalent to the output transition faults.
- If a gate has only one fanout, then the output transition faults are equivalent to the input transition faults on the fanout gate.

In this thesis, we have assumed that all faults are lumped at the output of the gate. Therefore, if a gate has only one fanout, then the two faults on the output line are lumped as faults at the output of the gate, instead of the equivalent input transition faults on the fanout gate. Some special cases of the ISCAS85 benchmark circuits have been taken care of in this manner. For example, if a primary input is directly connected to the primary output, then the two faults of this line are associated with the primary input. But if a primary input is connected to a primary output by a buffer or an inverter, then the two

faults are associated with the output of the buffer or inverter, since the faults at the input of the buffer or inverter are equivalent to the faults at its output.

Because of these more restrictive conditions, the ability to group faults in equivalence classes is very much reduced. It has been seen that the total number of transition fault equivalence classes is about 50 percent larger than stuck-at fault equivalence classes [25]. For a small circuit like c432, the total number of transition faults to be considered is 784, while the number of stuck-at faults is 524. For a large circuit, like c7552, the total number of transition faults is 12,284, while number of stuck-at faults is just 7,550. Table 4.1 gives the size of the collapsed fault lists for the ISCAS85 benchmark circuits.

Table 4.1. Number of Transition Faults for Benchmark Circuits

Circuit Name	Number of Transition Faults
c432	784
c499	918
c880	1582
c1355	2566
c1908	2938
c2670	4306
c3540	5654
c5315	8842
c6288	12512
c7552	12284

4.2 Random Test Pattern Generation Session

After the preprocessing phase, TENOR employs random pattern test generation only for non-robust tests. In this session, a packet of 32 patterns is generated and the circuit is simulated to find out if any faults are detected. Then all the faults which are initialized by the test patterns in this packet are identified. All the test patterns which detect or initialize some fault are stored and others are rejected. All the faults for which both the test patterns have been found are marked detected and are deleted from the fault list. Then another packet is generated and the same procedure is performed. If two consecutive packets do not detect any new fault, then the random session is terminated. Figure 4.2 gives a flowchart of the random pattern test generation session.

In Figure 4.2, the RANDOM_LIMIT is by default set to 2, but can be changed by the user. If the test mode is robust, i.e., ROBUSTMODE is set, the random test pattern generation session is skipped. This is due to the fact that robust fault coverage of random test patterns is very low [22]. The procedures *Dominant_Test_Detect* and *Check_Initialization* are used to find the detected faults and the initialized faults, respectively. These are described in detail in the following subsections.

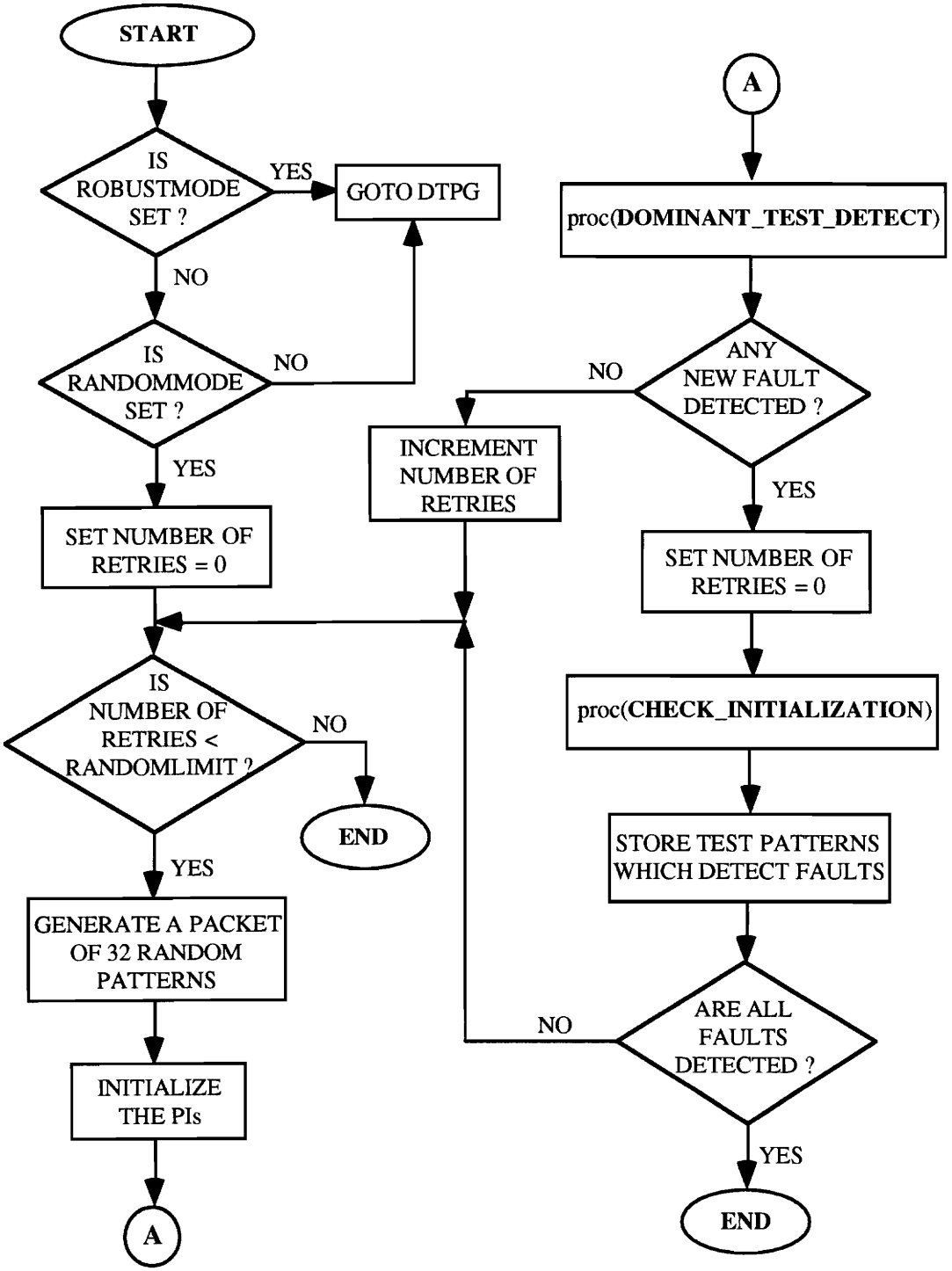


Figure 4.2. Flowchart of random test pattern generation session.

4.2.1 Detection of Transition Faults

The procedure *Dominant_Test_Detect* (in Figure 4.2) is based on parallel pattern single fault propagation (PPSFP) fault simulation. It performs fault simulation for a packet of 32 patterns in parallel assuming zero gate delays. As, by definition, the fault size of a transition fault is greater than the system clock period, the zero gate delay assumption would not invalidate the test generation. Explicit fault simulation is performed only for stem faults. Procedure *Delay_Fault_Simulation* essentially computes the detectability of stem faults from primary outputs toward primary inputs.

The *Check_Detection* procedure is used to determine the faults detected of a given gate, based on the observability of the gate. The leftmost pattern detecting each fault is saved as the T_2 pattern of that fault. The pseudo-code for the procedures is given below.

```
procedure Dominant_Test_Detect ( )
    fault_free_simulation ( ) ;
    for ( each gate in the reverse_order_gate_list ) do
        if ( gate is a primary output ) then
            gate->observe = 1 ;
        elseif ( the fanout of gate is 1 ) then
            complement gate output ;
            evaluate ( fanout gate ) ;
            if ( output of fanout gate changes AND fanout gate is observable )
            then        gate->observe = 1 ;
            else        gate->observe = 0 ;
```

```
            else
                gate->observe = Delay_Fault_Simulation ( gate ) ;
            endif ;
            if ( gate is observable AND there is an undetected fault of this gate ) then
                Check_Detection ( gate ) ;
                if ( all the faults of this gate have been detected ) then
                    delete this gate from the reverse_order_gate_list ;
                endif ;
            endif ;
        endfor ;
end procedure Dominant_Test_Detect ( ) ;

procedure Check_Detection ( gate )
        for ( each fault of gate ) do
            if ( fault is redundant or detected ) then
                continue with next fault ;
            endif ;
            if ( it is an output fault ) then
                if ( fault is STR ) then
                    observe = gate->observe AND gate->output ;
                elseif ( fault is STF ) then
                    observe = gate->observe AND ~(gate->output) ;
                endif ;
            elseif ( it is an input fault ) then
                evaluate_faulty_gate ( ) ;
                if ( faulty output is different from good output ) then
```

```
                    observe = gate->observe ;
                else
                    observe = 0 ;
                endif ;
            endif ;
            if ( all bits of observe are not zero ) then
                fault is detected ;
                find the leftmost pattern detecting the fault ;
                store it as T2 pattern of the fault ;
            endif ;
        endfor ;
end procedure Check_Detection ( gate ) ;
```

4.2.2 Initialization of Transition Faults

The procedure *Check_Initialization* (in Figure 4.2) finds all the faults initialized by the test patterns in a given packet of 32 test patterns. It uses the good value outputs of gates to determine which faults have been initialized by the patterns. In non-robust mode, if a pattern creates an initial value of a transition at a line, the corresponding fault on the line is initialized. While for robust test patterns, all the off-path sensitizing inputs of a gate on the path of fault propagation should have non-controlling values for both the initialization and test patterns. Hence, it is necessary to examine all off-path sensitizing inputs to determine if a pattern is an initializing pattern or not. Pseudo-code for the *Check_Initialization* procedure is given below.

```
procedure Check_Initialization ( )
    for ( each fault in the Delay_Fault_List ) do
        if ( it is an output fault ) then
            if ( fault is STR ) then
                observe = ~( gate->output ) ;
            elseif ( fault is STF ) then
                observe = gate->output ;
            endif ;
        elseif ( it is an input fault ) then
            if ( fault is STR ) then
                if ( faulty_line is 0 ) then
                    observe = 1 ;
                else
                    observe = 0 ;
                endif ;
            elseif ( fault is STF ) then
                if ( faulty_line is 1 ) then
                    observe = 1 ;
                else
                    observe = 0 ;
                endif ;
            endif ;
        endif ;
        for ( each bit i of observe ) do
            if ( observe_i is 1 ) then
                fault has been intialized ;
```

```
                    store pattern i as T1 pattern of this fault ;
                    break ;
                endif ;
            endfor ;
            if ( both patterns of this fault have been found ) then
                delete fault from Delay_Fault_List ;
            endif ;
        endfor ;
end procedure Check_Initialization ( ) ;
```

4.3 Deterministic Test Pattern Generation Session

All the remaining faults after the random test pattern generation session are processed in the deterministic test pattern generation session. (Note that if the test mode is robust test, random pattern testing session is skipped.) Figure 4.3 shows the flowchart for the deterministic test pattern generation session. After the random test pattern generation session, the fault list contains only the undetected faults for which pattern T_2 or pattern T_1 or both have not been found. If a T_2 pattern has not been found, the transition fault is mapped to an equivalent stuck-at fault using Table 4.2. The equivalent stuck-at fault is passed to FAN, which generates a test pattern for the fault. For example, an STR fault on a signal line is equivalent to the line stuck-at 0 fault. FAN finds a test pattern which sets a logic 1 at the faulty site, which is needed to create a rising transition at the faulty site.

Table 4.2. Mapping of Transition Faults to Stuck-at Faults

Transition Faults	Equivalent Stuck-at Faults
Slow-to-rise	Stuck-at-0
Slow-to-fall	Stuck-at-1

After all the faults have been processed to find the test pattern T_2, the faults for which pattern T_2 has been found but pattern T_1 has not been found are processed. The transition fault is mapped to the equivalent stuck-at fault to find the initializing pattern using Table 4.3.

4.3.1 Generation of Robust Tests

If the robust test mode option is set, only the deterministic test pattern generation session is used to generate robust test patterns. For robust tests, both the initializing pattern and the test pattern values should be observable at a primary output. For the test to be robust, the off-path sensitizing inputs of all the gates on the excitation path and the propagation path should have non-controlling values for both the patterns [22]. This ensures that the effect of extra delays in other parts of the circuit do not reach the transition propagation path and thus invalidate the test. In fact, FAN tries to assign non-controlling values to all the off-path sensitizing inputs, if possible, for non-robust tests as well as robust tests. In this way, the chances of having hazards in the circuit are minimized. For the robust tests, if a test does not satisfy the assigned values for the off-

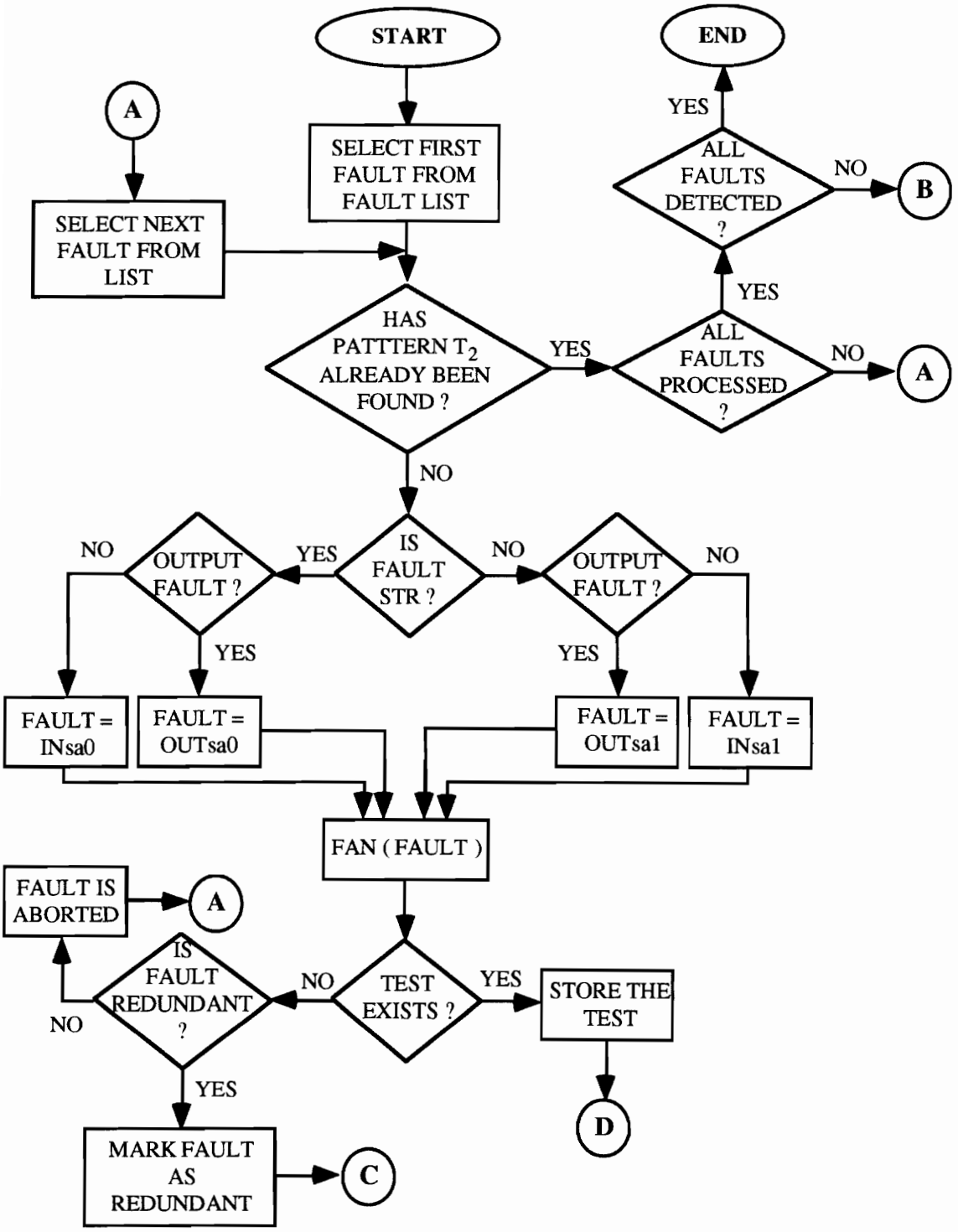


Figure 4.3. Flowchart of deterministic test pattern generation.

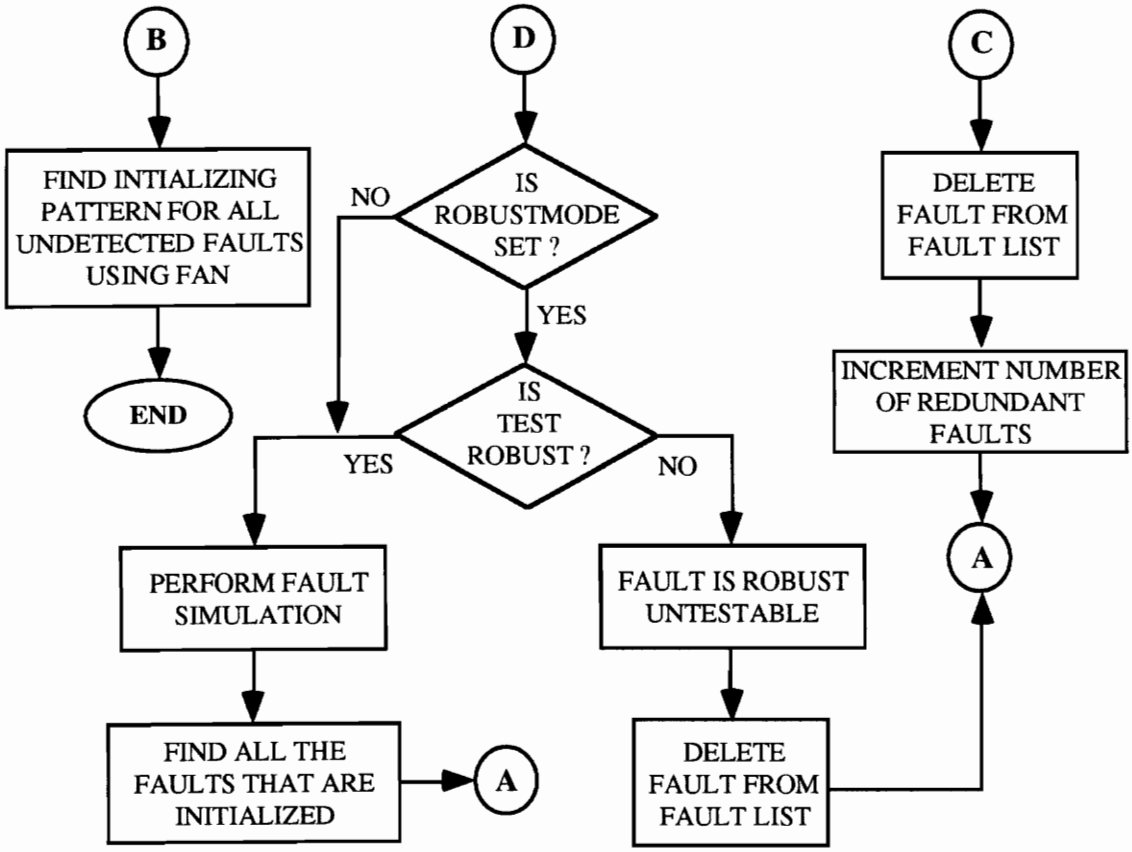


Figure 4.3 (continued). Flowchart of deterministic test pattern generation.

Table 4.3. Mapping of Transition Faults to Stuck-at Faults

Transition Faults	Equivalent Stuck-at Faults
Slow-to-rise	Stuck-at-1
Slow-to-fall	Stuck-at-0

path sensitizing inputs, then it is rejected and the fault is marked as robustly undetectable. While for non-robust test, first the non-controlling values are tried, if there is any conflict, then the controlling values may be assigned to generate a test.

Since the generation of robust tests requires stricter conditions, it is more time consuming to generate robust tests. As robust tests detect a given fault even in the presence of delays in other parts of circuit, it is more desirable than the non-robust tests.

4.4 Test Compaction Session

The first part of the test compaction session is to obtain a minimum length sequence of test patterns of which the $< T_1 , T_2 >$ pair of each fault is a substring. The algorithm proposed in [3] was implemented to find a minimal length superstring. For details of the algorithm, refer to [3]. To further compact the test set, both forward and reverse fault simulation is done using this superstring. This is similar to the technique employed in SOPRANO, described in Section 3.1.4.

4.5 Detection of Transition Faults Using Stuck-at Faults Test Set

The equivalence relation developed in Section 4.3 (Tables 4.2 and 4.3) suggests that since each transition fault can be mapped to an equivalent pair of stuck-at faults, a stuck-at test set is likely to cover many transition faults, albeit non-robustly. To explore the detectability of transition faults using stuck-at fault test sets, TENOR also accepts test patterns generated by an ATPG for stuck-at faults such as ATALANTA [24]. TENOR

performs fault simulation for the stuck-at test set to measure the transition fault coverage. Several experiments were performed on the transition fault coverage of stuck-at test sets. Details of the experiments appear in Chapter 5.

Chapter 5. Experimental Results

In this chapter, we describe the performance of our ATPG, TENOR, for transition faults in combinational circuits. TENOR has been implemented in the C language and runs on SUN workstations.

We conducted several experiments with TENOR. Two major objectives of experiments are: 1) to measure the performance of TENOR as an ATPG, and 2) to measure the transition fault coverage of stuck-at test sets. Section 5.1 presents experimental results on the performance of TENOR. Two cases, robust and non-robust test generations, were considered in the experiments. Section 5.2 presents the experimental results on the effectiveness of stuck-at test sets in detecting transition faults.

Throughout the experiments, ISCAS85 combinational benchmark circuits and ISCAS89 sequential benchmark circuits with full scan design were used. The characteristics of the ISCAS85 and ISCAS89 circuits are given in Table 5.1. All CPU times for TENOR were measured on a SUN SPARC 2 workstation.

Table 5.1. Characteristics of Benchmark Circuits

Name	# Gates	# PI's	# PO's	# Transition Faults
c432	160	36	7	784
c499	202	41	32	918
c880	383	60	26	1582
c1355	546	41	32	2566
c1908	880	33	25	2938
c2670	1193	233	140	4306
c3540	1669	50	22	5654
c5315	2307	178	123	8842
c6288	2416	32	32	12512
c7552	3512	108	207	12284
s208	115	19	10	346
s298	136	17	20	508
s344	193	24	26	552
s349	194	24	26	566
s382	182	24	27	646
s386	172	13	13	690
s400	186	24	27	688
s420	231	35	18	692
s420.1	252	34	17	760
s444	205	24	27	764
s510	236	25	13	956
s526	217	24	27	948
s526n	218	24	27	944
s641	433	54	42	730
s713	447	54	42	918
s820	312	23	24	1574
s832	310	23	24	1614
s838	457	67	34	1378
s838.1	512	66	33	1560
s953	440	45	52	1738
s1196	561	32	32	2110
s1238	540	32	32	2316
s1423	748	91	79	2512
s1488	667	14	25	2770
s1494	661	14	25	2810
s5378	3050	214	213	6988
s9234.1	5844	247	250	11328
s13207	8679	700	790	15602
s15850.1	10386	611	684	19046
s35932	18148	1763	2048	63502
s38584	20869	1464	1730	61254
s38417	23921	1664	1742	49738

5.1 Performance of TENOR

The performance of TENOR was measured for two test modes, robust and non-robust test generation. The experimental results for both of these test modes are presented in detail in the following.

5.1.1 Non-robust Test Generation

The performance of TENOR for non-robust test generation is summarized in Table 5.2. In the table, the fault coverage is obtained as the number of faults detected divided by the total number of faults. The fault efficiency is calculated as the fraction of the sum of the detected faults and the identified redundant faults out of the total number of collapsed faults. The total CPU time taken for each circuit is indicated in the table. The backtrack limit for the FAN algorithm was set to 100. The column for aborted faults lists the number of faults which were aborted by FAN.

As can be seen from the table, TENOR achieves high fault efficiency ($\geq$ 98 percent) for most of the circuits with reasonable CPU time. Among the 42 circuits in the experiments, TENOR achieves 99 percent or higher fault efficiency for 35 circuits. The largest test generation time is about 35 minutes for s38584 which is small enough for practical purposes. Most faults aborted by TENOR are redundant faults. The lowest fault efficiency, 96.7 percent is achieved for s298. The high fault efficiency for c6288 is notable, as the circuit is known to be a difficult to test circuit for path delay faults (c6288 contains around 10^{20} structural paths). It is a good example showing the usefulness of the gate delay fault model over the path delay fault model.

Table 5.2. Performance of TENOR for Non-robust Tests

Circuit Name	# of Transition Faults	# of Identified Redund. Faults	# of Aborted Faults	# of Test Patterns	Fault Coverage (%)	Fault Efficiency (%)	CPU Time (seconds)
c432	784	2	10	228	96.68	96.97	2.83
c499	918	16	12	206	96.95	98.70	1.58
c880	1582	0	0	228	100.00	100.00	1.87
c1355	2566	9	8	387	99.34	99.69	4.65
c1908	2938	11	10	514	99.29	99.66	7.12
c2670	4306	115	91	532	95.22	97.89	58.2
c3540	5654	223	17	767	95.76	99.70	24.63
c5315	8842	66	22	806	98.94	99.75	23.43
c6288	12512	71	37	262	99.14	99.70	42.03
c7552	12284	99	90	1013	98.40	99.27	246.83
s208	346	0	8	117	97.69	97.69	0.30
s298	508	0	17	140	96.65	96.65	0.38
s344	552	0	13	99	97.65	97.65	0.37
s349	566	6	6	109	97.88	98.94	0.42
s382	646	0	9	140	98.61	98.61	0.58
s386	690	0	7	268	98.99	98.99	0.83
s400	688	15	9	148	96.51	98.69	0.55
s420	692	0	5	244	99.28	99.28	0.88
s420.1	760	0	9	203	98.82	98.82	0.92
s444	764	27	9	164	95.29	98.82	0.60
s510	956	0	10	242	98.95	98.95	0.80
s526	948	1	13	243	98.52	98.63	1.00
s526n	944	0	18	235	98.09	98.09	0.93
s641	730	0	5	206	99.32	99.32	1.03
s713	918	97	23	212	86.93	97.5	1.32
s820	1574	0	19	602	98.79	98.79	2.40
s832	1614	19	17	602	97.77	98.95	2.40
s838	1378	0	11	407	99.2	99.2	3.00
s838.1	1560	0	11	460	99.3	99.3	3.85
s953	1738	0	4	419	99.77	99.77	2.68
s1196	2110	0	11	555	99.48	99.48	4.45
s1238	2316	91	11	606	95.6	99.53	5.73
s1423	2512	25	5	357	98.77	99.8	4.53
s1488	2770	0	18	753	99.35	99.35	4.43
s1494	2810	16	19	766	98.75	99.32	4.62
s5378	6988	77	46	1184	98.03	99.34	34.48
s9234.1	11328	580	104	1841	93.91	99.08	507.28
s13207	15602	225	102	2135	97.8	99.35	339.12
s15850.1	19046	678	61	2074	96.01	99.68	313.67
s35932	63502	7196	50	440	88.59	99.92	1321.02
s38584	61254	2325	265	4462	95.77	99.57	2095.83
s38417	49738	250	240	5007	99.02	99.52	2000.60

In Table 5.3, the performance of TENOR is compared with that of an ATPG called DTEST_GEN developed by Park and Mercer [16], as this is the only ATPG available with results for test generation for transition faults. In the table, CPU times were measured on a SPARC 2 (SUN 4/75) for TENOR, and on a Sun 4 system for DTEST_GEN. The model number of Sun 4 system is not shown in the paper.

Table 5.3. Comparison of Performance of TENOR and DTEST_GEN

Circuit	Fault Coverage (%)		Fault Efficiency (%)		CPU Time (seconds)		Speed-up
Name	TENOR	DTEST_ GEN	TENOR	DTEST_ GEN	TENOR	DTEST_ GEN	Ratio
c432	96.68	98.72	98.94	98.90	2.83	9.3	3.32
c499	96.95	99.13	98.70	100.00	1.58	17.4	11.01
c880	100.00	100.00	100.00	100.00	1.87	30.7	16.42
c1355	99.34	99.69	99.69	100.00	4.65	97.2	20.9
c1908	99.29	99.63	99.66	99.90	7.12	104.6	14.69
c2670	95.22	96.45	97.89	99.40	58.2	230.0	3.95
c3540	95.76	96.64	99.70	99.60	24.63	386.6	15.69
c5315	98.94	99.3	99.75	100.00	23.43	450.9	19.24
c6288	99.14	99.26	99.70	99.90	42.03	305.7	7.27
c7552	98.40	98.84	99.27	99.60	246.83	1168.8	4.73
Average	97.97	98.77	99.33	99.70	41.31	280.12	6.78

As can be seen from the table, TENOR and DTEST_GEN achieve comparable fault coverage for most of the circuits. The difference is less than 1% for most of the circuits. The largest difference is 2.18% for c499. This is because the circuit consists of a large number of exclusive-or gates which are difficult for TENOR to test. The fault efficiency of TENOR is slightly lower than DTEST_GEN due to the abortion of some

redundant faults by TENOR. The fault coverages of TENOR and DTEST_GEN are compared in Figure 5.1.

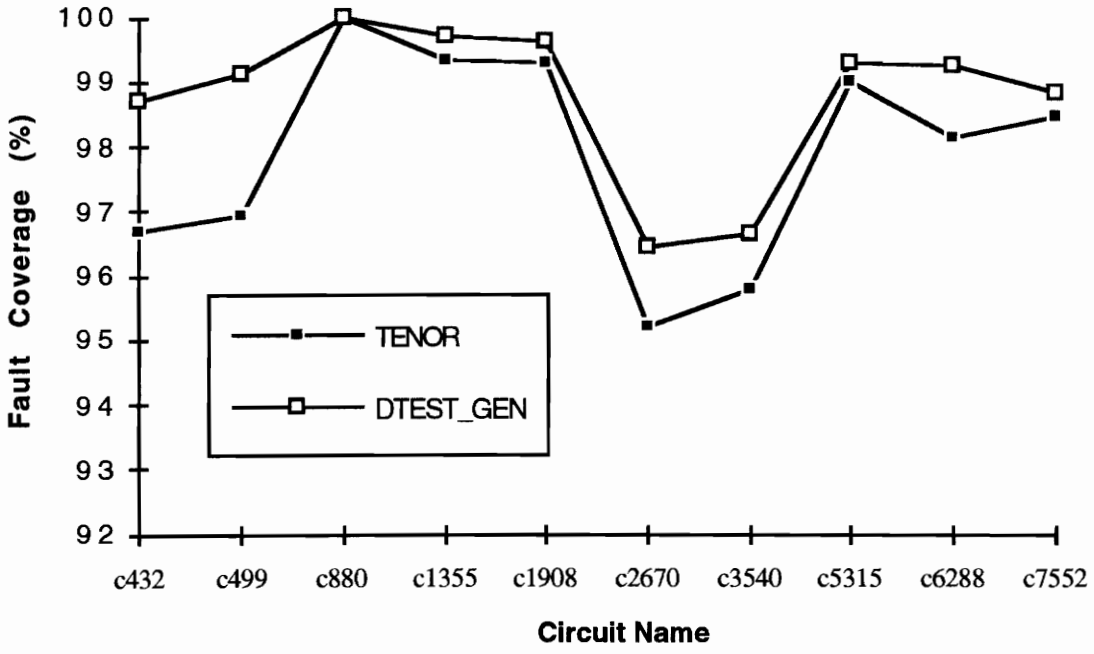


Figure 5.1. Comparison of fault coverages for TENOR and DTEST_GEN.

The major advantage of TENOR lies in its speed. From Table 5.3, TENOR is faster than DTEST_GEN by 3 to 20 times and on average by 7 times. For example, it takes 1168.8 seconds for DTEST_GEN for the largest circuit c7552. It takes only 259.1 seconds for TENOR for the same circuit. Even if the possibility is considered that a slower Sun 4 model was used for DTEST_GEN, we believe that TENOR would be at least as fast as DTEST_GEN. The main reason for the speedup is the FAN algorithm and parallel pattern fault simulation used in TENOR, while DTEST_GEN employs PODEM which is known to be less efficient than FAN and single pattern fault simulation. A comparison of CPU times between TENOR and DTEST_GEN is shown in Figure 5.2.

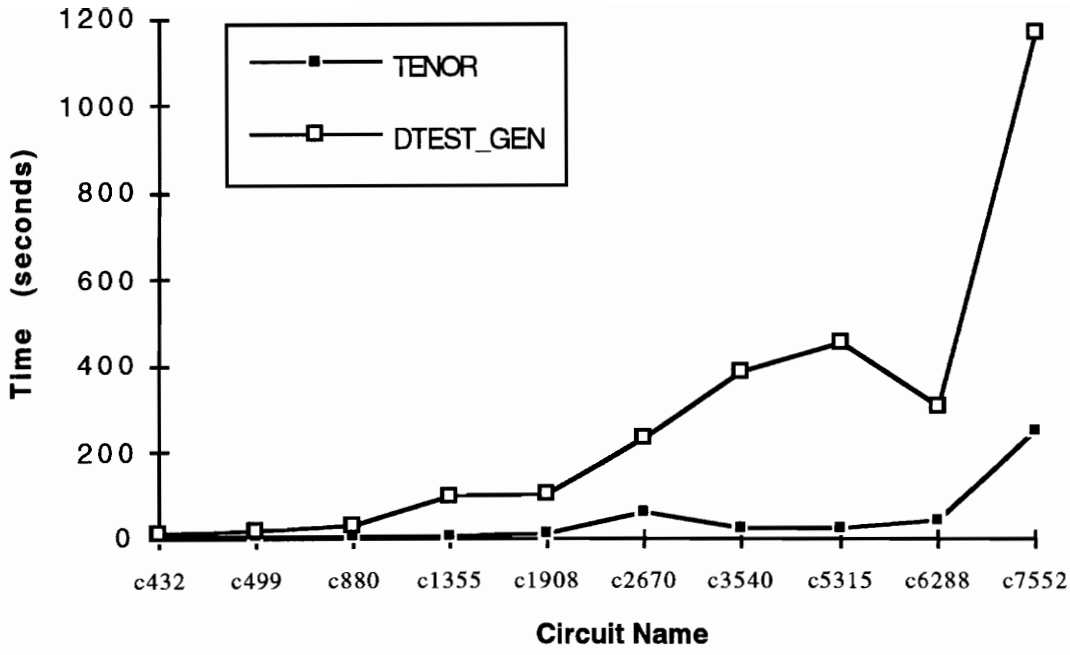


Figure 5.2. Comparison of CPU times for TENOR and DTEST_GEN.

5.1.2 Robust Test Generation

The performance of TENOR for robust test is summarized in Table 5.4. Since test generation for robust tests has to satisfy more stringent requirements compared with non-robust test generation, it is expected to achieve lower fault coverage with longer test generation time. As can be seen from the table, TENOR achieves over 83 percent fault coverage for all the circuits used in the experiment but two. The fault coverage is over 90 percent for nine of the circuits. We believe that the performance of TENOR is quite good. At this time, there is no other work available in the literature for robust test generation for

transition faults. As can be seen from the number of test patterns, TENOR generates less than 400 test patterns for most of the circuits. As compared to non-robust tests, the robust tests have a larger test set, even though the fault coverage is lower than non-robust tests. Although the CPU time of TENOR for robust tests is much longer than that for non-robust tests, the CPU time is still within practical limits. For example, robust test generation for the largest circuit, s38417, takes about 4 hours, which does not pose any practical problem. A "*" in the columns indicate that the simulation for that circuit could not be completed.

Table 5.5 gives a comparison between robust testing and non-robust testing. As expected, the fault efficiency of robust testing is lower than that for non-robust testing for every circuit. The fault efficiency of robust testing is on average, about 10 percent lower than that for non-robust testing. The largest discrepancy occurs due to the s9234.1 circuit in which the difference in fault efficiency between robust testing and non-robust testing is about 18 percent. Also, it can be seen that the average number of test patterns for robust testing is more than that for non-robust testing. This is because a test pattern pair which detects a fault robustly, may not detect faults which are not on the propagation path of this fault. This results in more test patterns, even though the fault coverage is low. Robust testing increases the CPU times. The average CPU time for robust testing is 679.43 seconds, and that for non-robust testing is 121.42 seconds. The longer CPU time for robust testing is due to the fact that only FAN is used to generate each pattern since the robust coverage of random patterns is known to be very low [22]. Figure 5.3 and 5.4 show the CPU time comparison of robust and non-robust tests for ISCAS85 and ISCAS89 circuits, respectively. As can be seen, the CPU time taken for test generation of robust tests is as much as 8 times that of non-robust tests.

Table 5.4. Performance of TENOR for Robust Tests

Circuit Name	# of Transition Faults	# of Redund. Faults	# of Aborted Faults	# of Test Patterns	Fault Coverage (%)	Fault Efficiency (%)	Total Time (seconds)
c432	784	2	90	166	88.27	88.52	5.67
c499	918	16	66	211	91.07	92.81	5.12
c880	1582	0	272	276	82.81	82.81	7.97
c1355	2566	9	130	397	94.62	94.93	16.83
c1908	2938	11	212	395	92.41	92.78	29.35
c2670	4306	112	484	807	86.16	88.76	212.22
c3540	5654	216	376	542	89.53	93.35	71.87
c5315	8842	61	843	1232	89.78	90.47	180.33
c6288	12512	74	113	346	98.51	99.10	149.32
c7552	12284	99	755	1391	93.07	93.85	745.72
s208	346	0	57	105	83.53	83.53	0.45
s298	508	0	84	124	83.47	83.47	0.67
s344	552	0	56	122	89.86	89.86	0.93
s349	566	6	70	122	89.05	87.63	0.92
s382	646	0	80	129	87.62	87.62	0.97
s386	690	0	81	172	88.26	88.26	1.37
s400	688	15	75	152	86.92	89.10	1.07
s420	692	0	118	159	82.95	82.95	1.60
s420.1	760	0	122	169	83.95	83.95	1.90
s444	764	27	73	157	86.91	90.45	1.08
s510	956	0	74	190	92.26	92.26	1.28
s526	948	1	116	226	87.66	87.76	1.58
s526n	944	0	116	234	87.71	87.71	1.80
s641	730	0	103	149	85.89	85.89	2.73
s713	918	101	91	149	79.09	90.09	2.55
s820	1574	0	179	395	88.63	88.63	5.03
s832	1614	19	273	379	81.91	83.09	6.08
s838	1378	0	207	348	84.98	84.98	6.18
s838.1	1560	0	257	282	83.53	83.53	6.90
s953	1738	0	143	303	91.77	91.77	4.80
s1196	2110	0	239	436	88.67	88.67	9.73
s1238	2316	90	283	445	83.90	87.78	11.90
s1423	2512	24	341	432	85.47	86.43	18.68
s1488	2770	0	218	526	92.13	92.13	13.68
s1494	2810	16	250	487	90.53	91.10	14.20
s5378	6988	66	803	1597	87.56	88.51	151.20
s9234.1	11328	568	2075	1603	76.67	81.68	850.62
s13207	15602	219	1727	2622	87.53	88.93	1060.92
s15850.1	19046	678	2112	2874	85.39	88.91	1566.77
s35932	63502	7196	3125	7780	83.57	95.08	8089.95
s38584	61254	*	*	*	*	*	*
s38417	49738	250	6523	9146	86.42	86.89	14594.67

Table 5.5. Robust vs. Non-robust Performance of TENOR

Circuit Name	# of Test Patterns		Fault Efficiency (%)		CPU Time (seconds)	
	Robust	Non-Robust	Robust	Non-Robust	Robust	Non-robust
c432	166	228	88.52	96.97	5.67	2.83
c499	211	206	92.81	98.70	5.12	1.58
c880	276	228	82.81	100.00	7.97	1.87
c1355	397	387	94.93	99.69	16.83	4.65
c1908	395	514	92.78	99.66	29.35	7.12
c2670	807	532	88.76	97.89	212.22	58.2
c3540	542	767	93.35	99.70	71.87	24.63
c5315	1232	806	90.47	99.75	180.33	23.43
c6288	346	262	99.10	99.70	149.32	42.03
c7552	1391	1013	93.85	99.27	745.72	246.83
s208	105	117	83.53	97.69	0.45	0.30
s298	124	140	83.47	96.65	0.67	0.38
s344	122	99	89.86	97.65	0.93	0.37
s349	122	109	87.63	98.94	0.92	0.42
s382	129	140	87.62	98.61	0.97	0.58
s386	172	268	88.26	98.99	1.37	0.83
s400	152	148	89.10	98.69	1.07	0.55
s420	159	244	82.95	99.28	1.60	0.88
s420.1	169	203	83.95	98.82	1.90	0.92
s444	157	164	90.45	98.82	1.08	0.60
s510	190	242	92.26	98.95	1.28	0.80
s526	226	243	87.76	98.63	1.58	1.00
s526n	234	235	87.71	98.09	1.80	0.93
s641	149	206	85.89	99.32	2.73	1.03
s713	149	212	90.09	97.5	2.55	1.32
s820	395	602	88.63	98.79	5.03	2.40
s832	379	602	83.09	98.95	6.08	2.40
s838	348	407	84.98	99.2	6.18	3.00
s838.1	282	460	83.53	99.3	6.90	3.85
s953	303	419	91.77	99.77	4.80	2.68
s1196	436	555	88.67	99.48	9.73	4.45
s1238	445	606	87.78	99.53	11.90	5.73
s1423	432	357	86.43	99.8	18.68	4.53
s1488	526	753	92.13	99.35	13.68	4.43
s1494	487	766	91.10	99.32	14.20	4.62
s5378	1597	1184	88.51	99.34	151.20	34.48
s9234.1	1603	1841	81.68	99.08	850.62	507.28
s13207	2622	2135	88.93	99.35	1060.92	339.12
s15850.1	2874	2074	88.91	99.68	1566.77	313.67
s35932	7780	440	95.08	99.92	8089.95	1321.02
s38584	*	4462	*	99.57	*	2095.83
s38417	9146	5007	86.89	99.52	14594.67	2000.60
Average	921	632	88.68	98.98	679.43	121.42

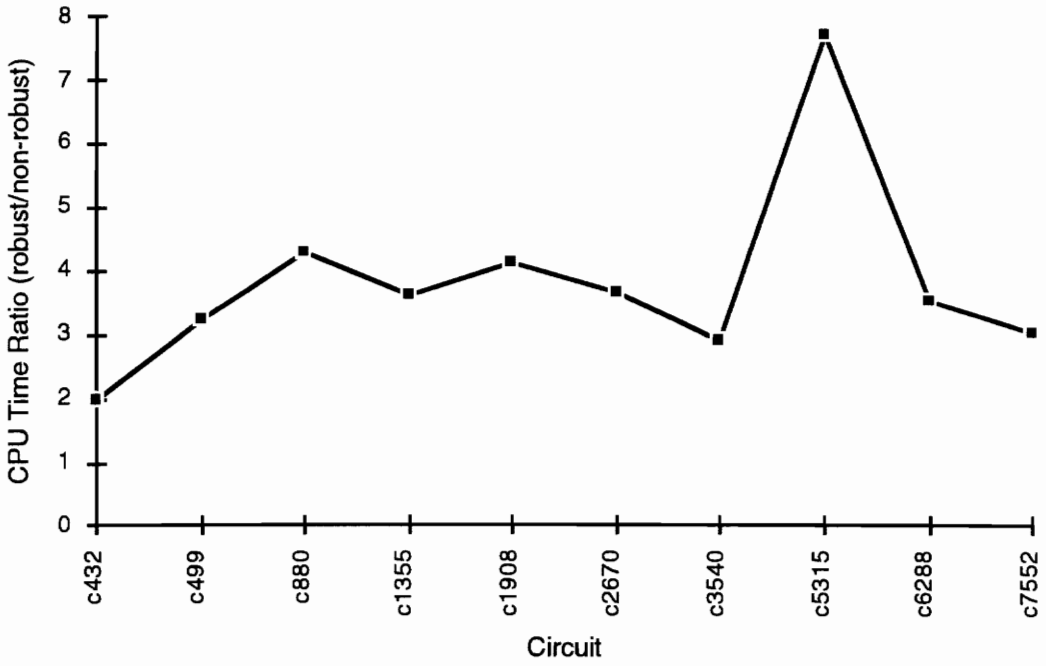


Figure 5.3. Comparison of CPU time ratios for ISCAS85 benchmark circuits.

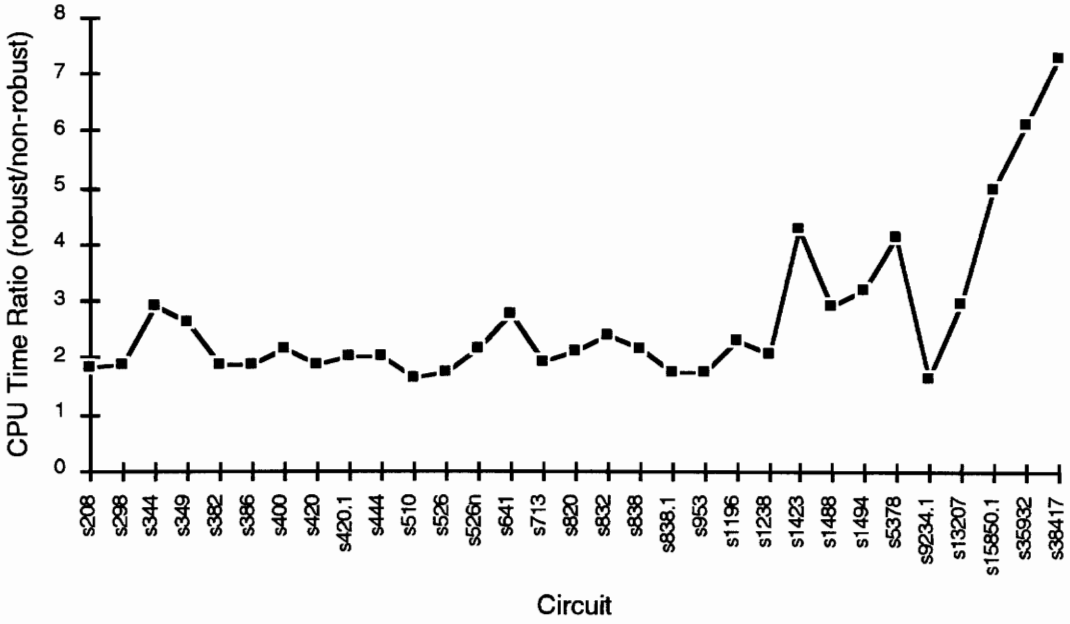


Figure 5.4. Comparison of CPU time ratios for ISCAS89 benchmark circuits.

The experiments described above have shown that TENOR achieves high fault coverage for non-robust testing with short processing time. The test length of TENOR is also within practical limits. The average test length for non-robust testing is 632. As this metric is not available for DTEST_GEN, a comparison is not possible. The experiments also show that robust testing results in lower fault coverage and longer CPU time than non-robust testing. Although robust testing is more desirable than non-robust testing, it may not be a practical solution for some large, complex circuits.

5.2 Effectiveness of Stuck-at Test Sets for Detecting Transition Faults

A test pattern that detects a stuck-at fault also detects the corresponding transition fault provided the transition fault has been properly initialized. This suggests that stuck-at test sets may be effective in detecting transition faults.

To investigate the above conjecture, a feature was added to TENOR to perform transition fault simulation for test patterns supplied by users. A stuck-at test set was generated by using ATALANTA [24], an ATPG for combinational circuits. The original number of test patterns for stuck-at faults and the increased test set sizes are given in Table 5.6. The column heading T_S in the table gives the number of test patterns obtained by ATALANTA. An increased test set size is generated by duplicating the original test set size by 2, 5 and 10 times, and then randomly shuffling the enlarged test set. For example, column $2T_S$ means that the original test set is doubled and then the doubled set is shuffled randomly.

Table 5.6. Number of Stuck-at Test Patterns

Circuit Name	Number of Test Patterns			
	T_S	$2T_S$	$5T_S$	$10T_S$
c432	48	96	240	480
c499	53	106	265	530
c880	52	104	260	520
c1355	85	170	425	850
c1908	120	240	600	1200
c2670	105	210	525	1050
c3540	157	314	785	1570
c5315	120	240	600	1200
c6288	30	60	150	300
c7552	204	408	1020	2040

Experimental results for transition fault coverage of various stuck-at test set sizes for non-robust testing are given in Table 5.7. The average transition fault coverage of the original stuck-at test set is 88.27 percent. The fault coverages for the enlarged test sets $2T_S$, $5T_S$, and $10T_S$ are 92.29 percent, 95.35 percent, and 96.82 percent, respectively. As the test set size increases, the fault coverage initially increases rapidly, but more or less saturates after that. For smaller circuits, the fault coverage increases sharply between T_S and $10T_S$. For example, for c432, fault coverage increases from 79.72 percent to 97.96 percent. But for larger circuits like c7552, fault coverage increases rather slowly. The above experiments show that stuck-at test sets achieve high transition fault coverage, especially if the test size is increased. When the original stuck-at test set size is increased by ten times, transition fault coverage achieved is over 90 percent for all the circuits.

Table 5.7. Transition Fault Coverage of Stuck-at Test Patterns

Circuit Name	Fault Coverage (%)			
	T_S	$R(2T_S)$	$R(5T_S)$	$R(10T_S)$
c432	79.72	90.31	95.15	97.96
c499	88.13	91.18	94.23	96.08
c880	87.8	92.79	97.66	98.42
c1355	92.48	94.47	96.57	97.39
c1908	89.59	91.87	93.16	95.34
c2670	87.34	90.34	93.73	95.01
c3540	83.57	88.66	92.59	94.43
c5315	89.36	92.97	96.46	97.89
c6288	93.42	96.91	98.47	98.91
c7552	91.28	93.43	95.49	96.73
Average	88.27	92.29	95.35	96.82

To further investigate the effectiveness of stuck-at test sets over random patterns, the fault coverage of random patterns is compared with that of stuck-at test set. The number of random patterns applied is identical to the size of stuck-at test set used for the comparison. Experimental results are given in Table 5.8. In the table, nR_S, where n=1, 2, 5, and 10, denotes random patterns whose size is equal to nT_S.

Table 5.8. Fault Coverage of Random Test Patterns

Circuit	R_S	T_S	$2R_S$	$2T_S$	$5R_S$	$5T_S$	$10R_S$	$10T_S$
c432	69.26	79.72	82.53	90.31	92.60	95.15	95.15	97.96
c499	71.68	88.13	76.14	91.18	85.40	94.23	92.59	96.08
c880	71.11	87.8	78.76	92.79	85.97	97.66	89.89	98.42
c1355	76.50	92.48	81.49	94.47	90.29	96.57	92.59	97.39
c1908	66.54	89.59	74.85	91.87	84.03	93.16	87.85	95.34
c2670	69.81	87.34	75.77	90.34	80.14	93.73	81.51	95.01
c3540	72.53	83.57	78.58	88.66	87.76	92.59	90.89	94.43
c5315	83.57	89.36	88.66	92.97	92.59	96.46	94.43	97.89
c6288	91.62	93.42	96.23	96.91	98.39	98.47	98.93	98.91
c7552	83.79	91.28	86.68	93.43	89.17	95.49	90.40	96.73

The experimental results indicate that a stuck-at test is more effective in detecting transition faults than random patterns for all circuits with any test set size. This fact is easily explained as a transition fault is detected by test patterns for the two corresponding stuck-at faults. The difference between the fault coverage of a stuck-at test set and that for the random patterns is large for small test set sizes, but the difference is smaller for larger test set sizes. For example, the difference in fault coverage for c2670 is 17.53 percent for R_S and T_S. When the test set is increased by ten times, the difference reduces to 13.5 percent. Figure 5.5 shows the fault coverages for the stuck-at and random test patterns for this circuit. This circuit was specifically chosen because it is a difficult circuit to test even for stuck-at faults, due to a large number of redundant faults. This graph clearly shows that the effectiveness of random test patterns is low compared to that of stuck-at test sets.

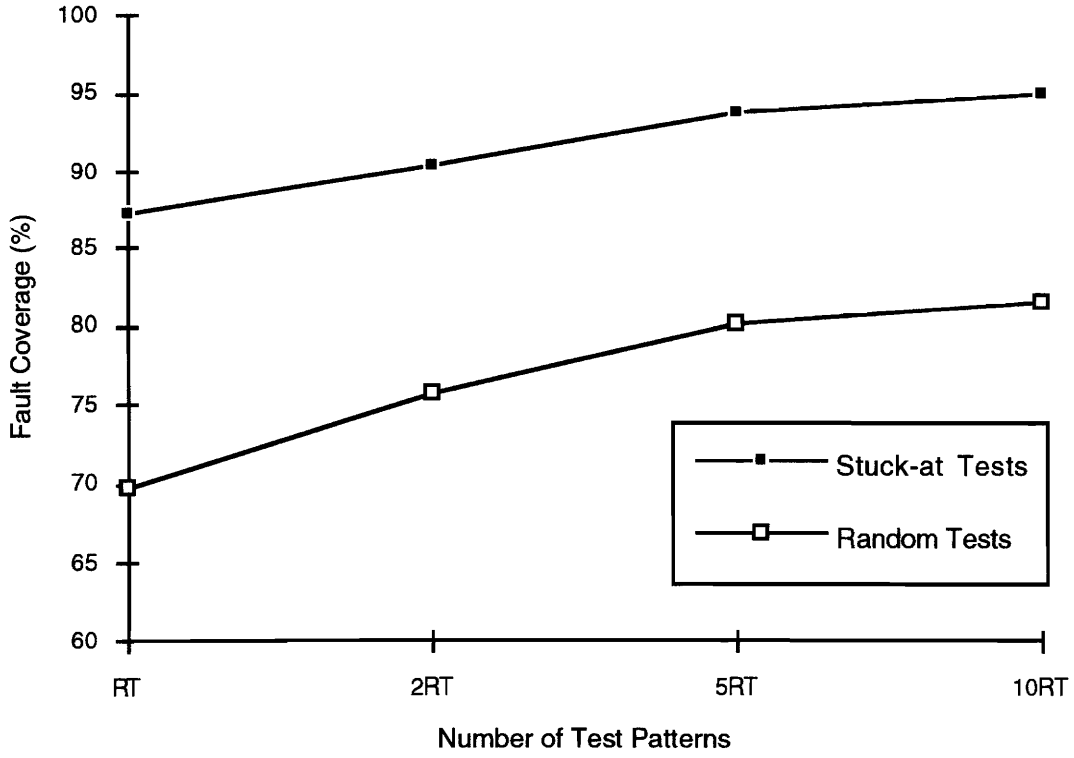


Figure 5.5. Comparison of fault coverages for c2670.

Although stuck-at test patterns are effective in detecting transition faults, they are not as effective as the ones generated by an automatic test pattern generator for transition faults such as TENOR. In Table 5.9, we compare the quality of random pattern test sets, stuck-at test sets and the test set generated by TENOR. For this experiment, the number of random patterns is equal to the number of the test patterns generated by TENOR.

Table 5.9. Fault Coverage Comparison of Three Test Sets

Circuit Name	Fault Coverage (%)			# of Test Patterns		
	Random	Stuck-at	TENOR	Random	Stuck-at	TENOR
c432	92.35	79.72	96.68	228	48	228
c499	83.88	88.13	96.95	206	53	206
c880	84.58	87.8	100.00	228	52	228
c1355	89.56	92.48	99.34	387	85	387
c1908	82.51	89.59	99.29	514	120	514
c2670	80.12	87.34	95.22	532	105	532
c3540	87.64	83.57	95.76	767	157	767
c5315	95.73	89.36	98.94	806	120	806
c6288	98.87	93.42	99.14	262	30	262
c7552	89.14	91.28	98.40	1013	204	1013

The above table shows the merit of using an ATPG for transition faults instead of stuck-at test sets or random patterns in detecting transition faults. The fault coverage of TENOR is far higher than that for either stuck-at test sets or random test patterns. As shown in the above experiments, even if the size of a stuck-at test set is increased by ten times, the fault coverage of stuck-at test sets is lower than that achieved by TENOR.

It is well known that random patterns are not effective for robust testing of transition faults [22]. This should also be true for stuck-at test sets, as it is unlikely that two test patterns detecting a transition fault set all of the off-path sensitizing inputs to their corresponding non-controlling values.

Chapter 6. Conclusion and Future Work

In this thesis, we presented an automatic test pattern generator, called TENOR, for transition faults in combinational circuits. We also studied the effectiveness of stuck-at test sets in covering transition faults. This chapter concludes this thesis by outlining the features of TENOR and providing suggestions for further improvement. Section 6.1 presents the features and capabilities of TENOR and Section 6.2 discusses the shortcomings and possible future enhancements of TENOR.

6.1 Features of TENOR

The key idea employed in TENOR is to map a transition fault to a pair of stuck-at faults and to then use a stuck-at fault test generation algorithm to generate the test pattern. The FAN algorithm [4] is employed for test generation and parallel pattern single fault propagation (PPSFP) is employed for fault simulation. The FAN algorithm and PPSFP method are known to be the most effective for combinational circuits [9]. Hence, TENOR is fast. The experimental results presented in Chapter 5 demonstrate the effectiveness of TENOR in speed. TENOR is 3 to 20 times faster than DTEST_GEN, a competing ATPG for non-robust testing [16].

TENOR can generate both non-robust and robust test patterns. TENOR achieves high fault coverage with short processing time for non-robust testing. A fault efficiency of 98.9 percent, on average, is achieved by TENOR for the ISCAS85 and ISCAS89 benchmark circuits. For robust testing, the fault coverage achieved is lower and CPU time greater than for non-robust testing. This is due to strict requirements for robust testing. Although robust testing is more desirable than non-robust testing, it may not be practical for large and complex circuits.

Through experiments we demonstrated that stuck-at test sets are effective in detecting transition faults, especially if the size of a stuck-at test size is increased by five to ten times. However, stuck-at test sets are inferior to test sets generated by an ATPG such as TENOR. Also, for larger circuits, the increased stuck-at test set may contain too many test patterns. As a test set covering transition faults also covers stuck-at faults, it is a better approach to use a test set for transition faults if covering both transition and stuck-at faults is needed.

Several features are available in TENOR to make it a versatile computer-aided test tool.

i) A stuck-at test set can be read by TENOR for transition fault simulation.

ii) Most of the test generation parameters can be controlled by the user, including the number of random packets generated, the backtrack limit of FAN, and others.

iii) TENOR can read both ISCAS85 and ISCAS89 circuit formats.

iv) Various log files and test pattern output files can be generated.

These features are described in more detail in Appendix A.

6.2 Future Enhancements

The capability of TENOR can be enhanced in a couple of aspects. First, TENOR can be extended to handle gate delay faults as well as transition faults. The extension would require the handling of gate delays in fault simulation and identifying the longest path through each fault site. Second, TENOR can be enhanced to identify all redundant faults. This would require the enhancement of the FAN algorithm. Several heuristics such as the learning algorithm and dynamic unique path sensitization [23] may be employed for the FAN algorithm.

Bibliography

[1] K. J. Antreich and M.H. Schulz, "Accelerated Fault Simulation and Fault Grading in Combinational Circuits," *IEEE Trans. on Computer-Aided Design*, vol. CAD-6, no. 5, pp. 704-712, September 1987.

[2] D. Brand and V.S. Iyengar, "Identification of Single Gate Delay Faults Redundancies," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 24-28, September 1992.

[3] S. Chakravarty and S.S. Ravi, "Computing Optimal Test Sequences from Complete Test Sets for Stuck-Open Faults in CMOS Circuits," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, pp. 329-332, March 1990.

[4] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, vol. C-32, no. 12, pp. 1137-1144, December 1983.

[5] M. Geilert, J. Alt, and M. Zimmermann, "On the Efficiency of the Transition Fault Model for Delay Faults," *IEEE International Conference on Computer-Aided Design*, pp. 272-275, November 1990.

[6] P. Girard, C. Landrault, and S. Pravossoudovitch, "Delay Fault Diagnosis by Critical-Path Tracing," *IEEE Design & Test of Computers*, vol. 9, no. 6, pp. 27-32, December 1992.

[7] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 215-222, March 1981.

[8] E.P. Hsieh, R.A. Rasmussen, L.J. Vidunas, and W.T. Davis, "Delay Test Generation," *Proc. 14th Design Automation Conf.*, pp. 486-491, June 1977.

[9] V.S. Iyengar, B.K. Rosen, and I. Spillinger, "Delay Test Generation 1 -- Concepts and Coverage Metrics," *Proc. of Intl. Test Conf.*, pp. 857-866, September 1988.

[10] V.S. Iyengar, B.K. Rosen, and I. Spillinger, "Delay Test Generation 2 -- Algebra and Algorithms," *Proc. of Intl. Test Conf.*, pp. 867-876, September 1988.

[11] H.K. Lee, "On Detection of Stuck-open Faults Using Stuck-at Test Sets in CMOS Combinational Circuits," *M.S. Thesis, Dept. of Elec. Eng., Virginia Polytechnic Institute & State University*, March 1989.

[12] H.K. Lee and D.S. Ha, "SOPRANO: An Efficient Automatic Test Pattern Generator for Stuck-open Faults in CMOS Combinational Circuits," *Proc. of 27th Design Automation Conf.*, pp. 660-666, June 1990.

[13] C.J. Lin and S.M. Reddy, "On Delay Fault Testing in Logic Circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 5, pp. 694-703, September 1987.

[14] U. Mahlstedt, "DELTEST: Deterministic Test Generation for Gate Delay Faults," *Proc. of Intl. Test Conf.*, pp. 972-980, October 1993.

[15] B.G. Oomman and S.B. Akers, "Fault Simulation for Delay Faults," *Proc. of Intl. Test Conf.*, pp. 328-335, September 1987.

[16] E.S. Park and M. R. Mercer, "An Efficient Delay Test Generation System for Combinational Logic Circuits," *IEEE Trans. Computer-Aided Design*, vol. 11, no. 7, pp. 926-938, July 1992.

[17] I. Pomeranz, *Private Communications*.

[18] A.K. Pramanick and S.M. Reddy, "On Detection of Delay Faults," *Proc. of Intl. Test Conf.*, pp. 845-856, September 1988.

[19] S.M. Reddy, V.D. Agarwal, and S.K. Jain, "A Gate Level Model for CMOS Combinational Logic Circuits with Application to Fault Detection," *Proc. of 21st Design Automation Conference*, pp. 504-509, June 1984.

[20] J. Savir and W.H. McAnney, "Random Pattern Testability of Delay Fault," IEEE Trans. on Computers, vol. 37, no. 3, pp. 291-300, March 1988.

[21] J. Savir and S. Patil, "Scan-Based Transition Test," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 8, pp. 1232-1241, August 1993.

[22] M.H. Schulz and F.Brglez, "Accelerated Transition Fault Simulation," *Proc. 24th Design Automation Conf.*, pp. 237-243, June 1987.

[23] M.H. Schulz, E. Trischler, T.M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, vol. 7, no. 1, January 1988,

[24] S. Thakar, H.K. Lee, and D.S. Ha, "ATALANTA 1.1", *Dept. of Elec. Eng., Virginia Polytechnic Institute & State University*, Blacksburg, VA, December 1993.

[25] J.A. Waicukauski, E. Lindbloom, B.K. Rosen, and V.S. Iyengar, "Transition Fault Simulation," *IEEE Design & Test* , vol. 4, no. 2, pp. 32-38, April 1987.

Appendix A. User Manual

1. Input to TENOR

The input circuit file to TENOR can be in either ISCAS89 format or the older ISCAS85 format. If a stuck-at test pattern set is being used, then the input test vector file should be in ISCAS89 format.

2. Invoking TENOR

NAME

tenor - An Automatic Test Pattern Generator for transition faults in combinational circuits.

SYNOPSIS

tenor [options] circuit_file

DESCRIPTION

TENOR is an ATPG for transition faults in combinational circuits. It accepts the circuit file either in ISCAS89 or ISCAS85 format, the default is ISCAS89 format. Several options are available for TENOR to modify parameters. The various options are:

-b n The number of maximum backtrackings of FAN algorithm. The default is 100.

-f <filename> The options are read from <filename>.

-h g Displays the on-line users's guide.

-h n Displays an example of the ISCAS89 netlist format.

-h t Displays an example test pattern file.

-h a Displays the entire online manual.

-I Use the ISCAS85 netlist format.The default is ISCAS89.

-l <filename> <filename> is created as the log file. The default is no log file.

-r n Sets the random limit of the random test pattern (RPT) session to n. The RPT session stops if n consecutive packets of 32 random patterns do not detect any new fault. If n=0, an RPT session is not used. The default for n is 2.

-R Generate robust test patterns. The default is to generate non-robust test patterns.

-s n Use n as the initial seed for the random number generator. The default is 10. If n=0, the initial seed is current time.

-S n Use the test patterns from a stuck-at test set. The number of test patterns read are multiplied by n. The test pattern file should be named as circuit_name.test and should be in the current directory. The default is not to read test patterns from the external file.

-t <filename> The test patterns are put in <filename>. The default is *.dtest for the circuit *.bench.

In default mode, the summary of the test pattern generation is reported on standard output, and the test patterns are put in *.dtest for a circuit file *.bench. For example, for the circuit file c432.bench, TENOR can be invoked as:

% tenor c432.bench

In this case, the summary will be reported on standard output and the test patterns will be put in c432.dtest. The above example command is the same as:

% tenor -b 100 -r 2 -s 10 -t c432.dtest c432.bench

Appendix B. An Example Run of TENOR

TENOR generates test patterns for transition faults in the given circuit and puts the test patterns in the circuit_name.dtest file. This appendix gives an example run for the circuit c432.bench. The summary is by default sent to standard output.

Command: tenor c432.bench

Output Summary:

```
*********************************************
*                                           *
*     Welcome to tenor (version 1.0)        *
*                                           *
*        Copyright (C) 1994,                *
*              VPI&SU                       *
*                                           *
*********************************************

****** SUMMARY OF TEST PATTERN GENERATION RESULTS ******
1. Circuit structure
   Name of the circuit                : c432
   Number of gates                    : 196
   Number of primary inputs           : 36
   Number of primary outputs          : 7
   Depth of the circuit               : 18

2. ATPG parameters
   Test pattern generation mode       : RPT + DTPG + TC
   Limit of random patterns (packets) : 2
   Backtrack limit                    : 100
   Initial random number generator seed : 10
```

3. Test pattern generation results
   Number of test patterns before compaction : 537
   Number of test patterns after compaction : 228
   Fault coverage : 96.684 %
   Test Quality : 96.939 %
   Number of collapsed faults : 784
   Number of identified redundant faults : 2
   Number of undetected faults : 24
   Number of backtracking in FAN : 910

4. CPU time
   Initialization : 0.133 secs
   Fault simulation : 1.883 secs
   FAN : 2.283 secs
   Total : 4.300 secs

The test pattern output file is c432.dtest, which contains the test patterns and the fault free response for each pattern. Any line starting with '*' is a comment.

* Name of circuit: c432

1: 10110101101011011100001101001111110 0 1101101
2: 01110110010010010110001011111010101 0 1101110
3: 01110111010111100001010011111001001 00 1111000
4: 11011011010011001110100001111000111 1 1101101
5: 01110010110101000011011001011011100 0 1111110
6: 10110101101011011100001101001111110 0 1101101
7: 01110011011010110100010010110010101 0 1101011
8: 11011011010011001110100001111000111 1 1101101
9: 01101011110001011101000011001001 1001 1101101
10: 11000111010001010110110001001100011 1 1111101
11: 00001000011110001111110011101011100 0 1001110
12: 10100010000011001110010111110110000 0 1101101
13: 10111011101110111011101111110011101 1 0011010
14: 10001001101011111011001011101000111 0 1011000
15: 10111011101110111011101111110011101 1 0011010
16: 01111101011010100010011110001110010 0 1111000
17: 10111011101110111011101111110011101 1 0011010
18: 11011010111000000011001010010111100 1 1011110
19: 00011110000110101000010011100101110 1 1101011
20: 01111101011010100010011110001110010 0 1111000
21: 10111011101110111011101111110011101 1 0011010
22: 10110101010001101101110011011101100 0 1101100
23: 10111011101110111011101111110011101 1 0011010
24: 00110011111100110010111000111101001 1 1011011
25: 10111011101110111011101111110011101 1 0011010

```
26: 011101111000101010111011010011001111 1101001
27: 101110111011101110111011111100111011 0011010
28: 011100011000011100010110001011100111 1010101
29: 011101111000101010111011010011001111 1101001
30: 101110111011101110111011111100111011 0011010
31: 110111110011110010000111110100000111 1001011
32: 111001110110111011011010001101101111 1101100
33: 101110111011101110111011111100111011 0011010
34: 101100100011001010010011101111100101 1101000
35: 101110111011101110111011111100111011 0011010
36: 011010110111100110011011001000100111 1000000
37: 101110111011101110111011111100111011 0011010
38: 001100000000000110011011000010001110 1 0101000
39: 101110111011101110111011111100111011 0011010
40: 011100101110101100011111000101000000 1010000
41: 001100000000000110011011000010001110 1 0101000
42: 101110111011101110111011111100111011 0011010
43: 100011111100000010000111101001010001 1111001
44: 011010110111100110011011001000100111 1000000
45: 101110111011101110111011111100111011 0011010
46: 011100101101010000110110010110111000 1111110
47: 101110111011101110111011111100111011 0011010
48: 011010111100010111010000110010011001 1101101
49: 101110111011101110111011111100111011 0011010
50: 001110010101110010000011100011111111 1001001
51: 101110111011101110111011111100111011 0011010
52: 000111001110100011110001101100110100 1011100
53: 101110111011101110111011111100111011 0011010
54: 110101100000111100011111101101110111 1011101
55: 000111001110100011110001101100110100 1011100
56: 101110111011101110111011111100111011 0011010
57: 001110111011001011101001001111110011 1001100
58: 000111001110100011110001101100110100 1011100
59: 101110111011101110111011111100111011 0011010
60: 011101110101111000010100111100100100 1111000
61: 110101100000111100011111101101110111 1011101
62: 101110111011101110111011111100111011 0011010
63: 100010110001111000000111001011001111 1101001
64: 001110111011001011101001001111110011 1001100
65: 110101100000111100011111101101110111 1011101
66: 101110111011101110111011111100111011 0011010
67: 011101100100100101100010111110101010 1101110
68: 100010110001111000000111001011001111 1101001
69: 011101110101111000010100111100100100 1111000
70: 001110111011001011101001001111110011 1001100
71: 101110111011101110111011111100111011 0011010
72: 011100110110101101000100101100101010 1101011
```

```
 73: 00111011101100101110100100111111110011 1001100
 74: 10111011101110111011101111111100111011 0011010
 75: 11000101010000110011010110001010100 1111111
 76: 00111011101100101110100100111111110011 1001100
 77: 10111011101110111011101111111100111011 0011010
 78: 10110101101011011100001101001111111100 1101101
 79: 00011110011001001111111110101011101101 1101101
 80: 10111011101110111011101111111100111011 0011010
 81: 01011000000011111111100110010000011111 1100000
 82: 00111011101100101110100100111111110011 1001100
 83: 10111011101110111011101111111100111011 0011010
 84: 00010011110111101001110000101011011 0111011
 85: 01011000000011111111100110010000011111 1100000
 86: 11000101010000110011010110001010100 1111111
 87: 01110110010010010110001011111010101 1101110
 88: 00011110011001001111111110101011101101 1101101
 89: 10111011101110111011101111111100111011 0011010
 90: 11011010111011100101010100100011111 1111100
 91: 11000101010000110011010110001010100 1111111
 92: 10111011101110111011101111111100111011 0011010
 93: 10100100100000110111001000100101100 1101111
 94: 01011000000011111111100110010000011111 1100000
 95: 11111010111110100110001110000111100 1011100
 96: 11100101111011010000110110100111000 1111101
 97: 11011010111011100101010100100011111 1111100
 98: 11011010110000100000001000011010100 1111110
 99: 01000110100100001110011100000110101 1110000
100: 11110111011110011101000011000001001 1111010
101: 10100100100000110111001000100101100 1101111
102: 00010011110111101001110000101011011 0111011
103: 11100101111011010000110110100111000 1111101
104: 00000000011101110100111010101011010 1011110
105: 11011010111011100101010100100011111 1111100
106: 00010011110111101001110000101011011 0111011
107: 10111011101110111011101111111100111011 0011010
108: 11110111011110011101000011000001001 1111010
109: 11100101111011010000110110100111000 1111101
110: 11110111011110011101000011000001001 1111010
111: 11111010111110100110001110000111100 1011100
112: 01000110100100001110011100000110101 1110000
113: 01001111100000111010001110111101101 1100000
114: 11100101111011010000110110100111000 1111101
115: 11011010110000100000001000011010100 1111110
116: 11111010111110100110001110000111100 1011100
117: 01000110100100001110011100000110101 1110000
118: 01001111100000111010001110111101101 1100000
119: 11111010111110100110001110000111100 1011100
```

120: 000111100110010011111111010101110110 1101101
121: 110110011001100011011101110111011101 1101100
122: 000100010000001010110101110111000010 0111001
123: 110110011001100011011101110111011101 1101100
124: 110010101001110100011111101100101000 0111101
125: 110110011001100011011101110111011101 1101100
126: 000111100001101111111011110100110000 0111010
127: 110110011001100011011101110111011101 1101100
128: 100010011010111110110010111010001110 1011000
129: 110110011001100011011101110111011101 1101100
130: 001111001011101010010000001111110010 0101111
131: 110110011001100011011101110111011101 1101100
132: 000111111001011100110011101101110001 0011101
133: 110110011001100011011101110111011101 1101100
134: 110110100000000010001101100000010100 1 0100000
135: 110110011001100011011101110111011101 1101100
136: 100010011011011101010111000111011101 0111100
137: 110110011001100011011101110111011101 1101100
138: 100010010000100100000011111010100010 0001010
139: 110110011001100011011101110111011101 1101100
140: 000001100011011010000000001010100001 1011111
141: 110110011001100011011101110111011101 1101100
142: 110111011011101010010111100101011001 0100111
143: 110110011001100011011101110111011101 1101100
144: 100010111110001000110011101100101010 0011110
145: 101010110010000010000111101111100110 1011000
146: 010010000110111100010111101111110100 1100000
147: 110110011001100011011101110111011101 1101100
148: 001100000000000110011011000010001 1101 0101000
149: 100011111100000010000111101001010001 1111001
150: 011100101110101100011111000101000000 1010000
151: 110110011001100011011101110111011101 1101100
152: 000000000111011101001110101010110101 1011110
153: 000111100110010011111111010101110110 1101101
154: 100100010001000100010001000000010000 0000000
155: 011010110111110011001101100100010011 1 1000000
156: 100100010001000100010001000000010000 0000000
157: 100011111100000010000111101001010001 1111001
158: 000100110011011001011011011101000100 1111000
159: 100100010001000100010001000000010000 0000000
160: 011101110101111000010100111100100100 1111000
161: 100100010001000100010001000000010000 0000000
162: 011100110110101101000100101100101010 1101011
163: 010110000001111111111001100100001111 1 1100000
164: 100100010001000100010001000000010000 0000000
165: 000100111101111010011100001010110111 0111011
166: 100100010001000100010001000000010000 0000000

```
167: 10100100100000011011100100010010111001101111
168: 10010001000100010001000100000001000000000000
169: 00000000011101110100111010101011010110111110
170: 11011010110000100000001000011010010011111110
171: 10010001011101110111000101110001010101000101
172: 11110101101000011111101100010111010011001001
173: 10010001011101110111000101110001010101000101
174: 00100001000111000110100110001111110010011101
175: 01110010001011011100110101100011111001111011
176: 10010001011101110111000101110001010101000101
177: 00110101111110111100000010010111111011001001
178: 10001001101101110101011100011101110101110100
179: 01111111111111111111111010111111111111000000
180: 01110011011010110100010010110010101011011011
181: 11010001000001010001010001010101010011111101
182: 11111010011001100010001101101001001110111110
183: 11010001000001010001010001010101010011111101
184: 00111111000101100111001111010111011110111011
185: 11010001000001010001010001010101010011111101
186: 00110000000000011001101100001000111010101000
187: 11010001000100000101010001000101010011111100
188: 10111111000111110001100100001111010110011001
189: 11010001000100000101010001000101010011111100
190: 00011110011001001111111110101011101101101101
191: 10010111000100010101000100010001000001111100
192: 11000111111110011000000011011011000101001111
193: 10010111000100010101000100010001000001111100
194: 11011111001111001000011111101000000111010011
195: 11110101000100010001000100010001000001111111
196: 01010100010100001101001100010101100001110110
197: 11110101000100010001000100010001000001111111
198: 11100111011011101101101010001101101111011100
199: 11110101000100010001000100010001000001111111
200: 11010000000011111011000010110000011110011101
201: 11110101000100010001000100010001000001111111
202: 01101011011111001100110110010001001111000000
203: 11011011010011001110100001111000011111101101
204: 00111001010111001000001110001111111110011001
205: 11111110111101111111111111111111111111111110
206: 01110000001111101111111011011011110011010010
207: 10011001100110011001110100011001100101110111
208: 11111001000110100110111110010101111110011001
209: 10011001100110011001110100011001100101110111
210: 11110100111101110000011000011001000110111110
211: 10011001100110011001110100011001100101110111
212: 11000111010001010110110001001100011111110101
213: 00111011101100101110100100111111001110011001
```

214: 11101111011111111111111111111111111 1011111
215: 10111011101110111011101110111011110111110 0011000
216: 01101001011101101010111111011111010110 1000011
217: 01100101111101110101111101111010011 0 1101111
218: 10010001000100010111011101011001000 0 0101010
219: 11011111001111001000011111010000011 1 1001011
220: 01110011011010110100010010110010101 0 1101011
221: 11111111111111111111111111111101110 1011000
222: 01000110100100001110011100000110101 1 1110000
223: 11010011001100110011001100110011001 10010 0110000
224: 11011010110000100000001000011010010 0 1111110
225: 10011001110100011001100110011001 10011001 0111110
226: 11010001000100000101010100000101010 0 1111101
227: 01000110110000110001001100000100000 0 1110110
228: 11000001001001011100110111101111001 0 1101101

Vita

Dhawal Tyagi was born in Meerut, India on March 29, 1969. After completing his high school education at Apeejay Public School, Faridabad, India in June 1987, he began undergraduate study in Electronics Engineering at Motilal Nehru Regional Engineering College, University of Allahabad, Allahabad, India. After completing his B.E. degree in Electronics Engineering, Dhawal decided to pursue an M.S. degree in Electrical Engineering at Virginia Polytechnic Institute and State University. At Virginia Tech he concentrated his studies in design automation and testing and computer networking. His career goals are to work in the computer networking area.

Dhawal Tyagi