TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Md Younus Ali 184623IASM

TRANSITION DELAY TEST GENERATION USING STUCK-AT-FAULT TEST PATTERNS

Master's thesis

Supervisor: Prof. Raimund- Johannes Ubar

D.Sc. Institute of Computer Engineering
Tallinn University of Technology
Professor, Chair of Computer systems Test and Verification

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Md Younus Ali 184623IASM

TESTIDE GENEREERIMINE VIIVITUSRIKETELE DIGITAALSKEEMIDES

Magistritöö

Juhendaja: Prof. Raimund Ubar

Tehnikateaduste doktor

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Md Younus Ali

18.05.2020

Abstract

Testing of delay faults is one of the important problems in the semiconductor industry. Most common delay fault has been transition delay fault (TDF) model due to the easy measurability and closeness to the standard stuck-at-fault (SAF) model. The main difficulty of the test generation for these models is related to the exponential complexity of the task that makes the methods for SAF and TDF test generation not scalable.

In this thesis, a new method, algorithms, and tools are developed, which have linear complexity, and are therefore scalable and can be efficient also for complex circuits. Differently from the traditional approach of generating tests by structural analysis of circuits, which is the reason of the exponential complexity, the new proposed method is based on transforming the SAF test directly to the TDF test. The transformation is based on analysis of the fault table of SAF test and recombining the SAF test patterns to satisfy the conditions of TDF testing. The proposed approach is implemented by creating two tools: a novel fault simulator for evaluating the TDF cover of the given SAF test, and a generator of additional test pairs to improve the TDF cover close to the value of SAF cover.

Experimental research with using traditional ISCAS'85 benchmark circuits showed the feasibility and efficiency of the new approach, and demonstrated a speed-up of TDF test generation compared to the traditional structural approach.

Annotatsioon

Testide genereerimine viivitusriketele digitaalskeemides

Viivitusrikete testimine on elektroonikatööstuses üks olulisi probleeme. Kõige levinum viivitusrikke tüüp on siirdeviivitusrike (SVR) tänu lihtsale mõõdetavusele ja standardse konstantrikke (KR) mudeli lähedusele. Testide genereerimise peamised raskused nende mudelite kasutamisel on seotud ülesande eksponentsiaalse keerukusega, mille tõttu KR- ja SVR-testide genereerimise meetodid skaleeruvad halvasti keerukate skeemide puhul.

Selles magistritöös on väljatöötatud uus meetod, algoritmid ja tööriistad, milliste omaduseks on lineaarne keerukus ja mis seetõttu skaleeruvad hästi ja on efektiivsed ka keerukate skeemide jaoks. Erinevalt traditsioonilisest lähenemisest, milleks on testide genereerimine skeemide struktuurse analüüsi abil, mis ongi eksponentsiaalse keerukuse põhjuseks, põhineb uus väljatöötatud meetod KR-testi teisendamisel otse SVR-testiks. Teisendus põhineb KR-testi rikete tabeli analüüsil ja KR-testi vektorite rekombineerimisel, et täita SVR testimise tingimusi. Uue lähenemisviisi rakendamiseks projekteeriti töös kaks tööriista: uudne rikete simulaator etteantud KR-testi poolt saavutatava SVR-katte hindamiseks ja täiendavate testipaaride generaator SVR-katte parandamiseks, et saavutada KR-katte tase.

Eksperimentaaluuringud, milles kasutati traditsioonilisi ISCAS'85 katseskeeme, demonstreerisid uue lähenemisviisi teostatavust ja tõhusust ning tõestasid SVR-testide genereerimise suuremat kiirust, võrreldes traditsioonilise struktuurse lähenemisviisiga testide genereerimisele.

Acknowledgment

Firstly, I would like to thank God for His unconditional love and the power He gives me to complete this thesis.

Secondly, I would like to thank my supervisor, Prof. Raimund-Johanned Ubar, for his supervision, advice, and full support throughout the whole thesis. It has been an honor as well as privilege for me to work with you. I am forever grateful to you, sir.

To my beloved parents, who had always stood by me, encouraged me and supported me throughout my whole life.

A special thanks goes to friends who have always been stood by my side during this whole study period. I must say a massive thanks to Angel, Azad, and Shanto, who help me during my tough time.

God bless you all.

List of abbreviations and terms

TDF	Transition delay fault
SAF	Stuck-at-fault
VLSI	Very Large Scale Integration
BDD	Binary Decision Diagram
ATPG	Automated Test Pattern Generation
RTG	Random Test Generation
TT	Turbo Tester

Table of contents

List of figures

List of tables

1 Introduction

The aim of this thesis is test sequence generation for testing transition delay faults (TDF) in combinational circuits from the given set of test patterns, which generated for stuck-at-faults (SAF). Alongside of generating test sequence this work also focuses on improving fault coverage for transition delay fault (TDF).

The rest of this chapter discusses on different test methods and delay faults simulation methods.

1.1 Background and Problem

Transition delay is a common issue in the semiconductor industry where the manufacturing process costs a large amount of money due to chip identification and elimination. To reduce the cost as well as the number of faulty chips, proper investment in chip testing and diagnosis can be a game-changing decision for the manufacturing process. The need for delay testing arises from here. The main objective of the delay test is to detect the faults and to make sure that the design meets the desired performance specifications [1].

Nowadays, there have been plenty of fault models to perform the transition delay tests. Here, the single stuck-at-fault model is considered as the classical fault [2][3]. Other fault models such as transition delay fault, gate delay fault, and bridging fault, which are mainly non-classical faults.

A vital aspect of guaranteeing high-quality chip fabrication is effective fault analysis through a transition delay test. A better fault diagnosis algorithm, along with a better test generation system, can bring efficiency and achievement as well. Timing issues and cost efficiency are the main reason for the transition delay test generation. The use of stuck-at fault pattern generation [4] in the diagnosis of transition delay test generation is the main contribution of this thesis.

1.2 Goal of thesis

The goal of this work was to develop a new method for generating tests of Transition Delay Faults (TDF) using the tests generated for Stuck-at-Faults (SAF) and the SAF fault table. For this work, we had to develop two separate algorithms where the first algorithm works on generating transition delay faults (TDF) test, and the second algorithm helps to improve the fault coverage by using additional test patterns.

Alongside these two algorithms, we also worked on another algorithm which helps to update the fault vectors and calculate the fault coverage.

1.3 Organization of thesis

The thesis is organized as follows.

In chapter 2, a brief discussion about digital test methods and fault simulation and fault diagnosis. Chapter 2 also discuss about Turbo-Tester.

Chapter 3 discusses different delay fault simulation method.

In chapter 4, a brief discussion about developed new method for transition delay test generation and the algorithm for improving delay fault coverage.

Chapter 5 is about the experimental results.

Conclusions or summary of work is presented in chapter 6.

Appendix and References are listed after the conclusion.

2 Overview of digital test methods

2.1 Fault models

While generating the transition delay test and fault diagnosis processes, we need to represent corresponding test patterns and logical faults. Here we discuss the popular fault models from stuck-at fault models to bridging fault models. These are also divided into transistor-level faults, gate-level faults, delay faults, and many more.

2.1.1 Stuck-at-fault model

Stuck-at Fault Model is the most used gate-level fault model. It is a signal line stuck at a logic ‘0’ or ‘1’, which is referred to as stuck-at 0 and stuck-at 1, respectively. Also, assuming that the value cannot be changed anyway.

It is a logic fault model that is related to timing issues. The stuck-at fault generally represents a slight physical defect where for a circuit with ‘N’ signal lines, the number of possible stuck-at faults is ‘2N’.

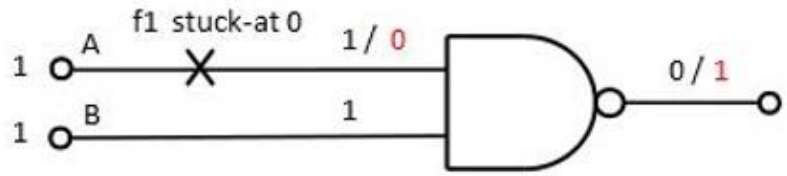


Figure 1 A NAND gate stuck-at fault illustration.

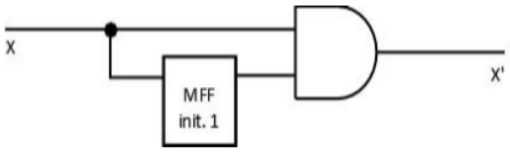
From figure 1, the input value of the NAND gate is ‘11’and the output value is ‘0’. Due to the stuck-at-0 fault in input line 1, the actual input of this NAND gate is stuck at ‘01’. And so the output response becomes ‘1’. To sharpen this stuck-at 0 faults in input line 1, we must apply ‘1’ at that line, and on the other front, we need to use non-effect signal ‘1’.

2.1.2 Transition Delay Fault Model

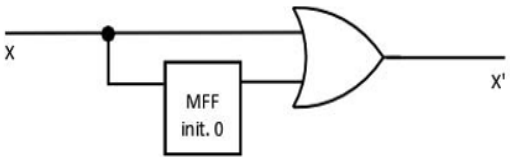
In general, transition delay fault models are divided into two categories; the first one is path delay fault, and the second one is the transition delay fault. Path delay fault model spreads the circuit path from flip-flop to flip-flop where transition delay fault occurs at a single gate input or output in the circuit, which also has an excessive delay. Figure 2 shows a transition delay fault as ‘slow-to-rise’ and ‘slow-to-fall.’ It is Similar to a stuck-at fault fixed at ‘0’ or ‘1’.

A pair of sequence test vectors has to be applied to detect the delay faults for both. Here, the first vector set considered to be the first state on the target signal line, and the second one is the specific fault along with the effect to a primary output. The model consists of a modeling flip-flop which is shown in Figure 2(a) initializes to 1 and an AND gate. When we apply ‘00’, ‘11’, and ‘10’, we will get a corresponding output from the model. But, when we use ‘01’, the output response will be ‘00’.

Similarly, modeling flip-flop initialized to 0 in Figure 2(b). Here, when we apply ‘10’, the output comes ‘11’, which is supposed to be ‘10’. This is how slow-to-fall fault is modelled.



(a) Modeling slow-to-rise fault



(b) Modeling slow-to-fall fault

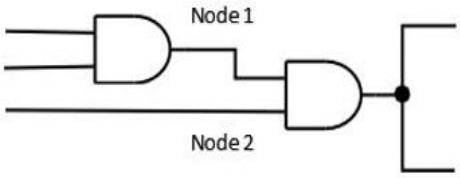
Figure 2 Modelling of transistion delay fault

2.1.3 Bridging Fault Model

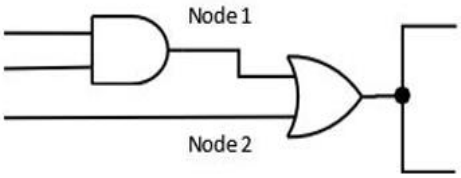
During the process of VLSI, two segments in the interconnection area may result in logic errors while getting too close to each other. Here, to create a stuck-at fault bridging between a signal line to VDD is analogous.

Figure 3 contains the most common bridging fault models, which are wire-AND, wire-OR, and dominant [5]. In the wire-AND model, logic 0 controls the two bridged nodes. That means either of two nodes being '0' or will lead to '0'.

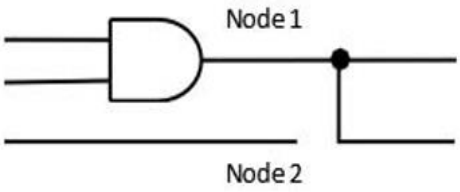
Similarly, if the wire-OR model provides a 1-dominant bridging fault, the value of the two shorted wire is '1' and also generates '1' on both lines. In the predominant spanning deficiency, the sign an incentive on one line is driven by the impulse on the other line with more grounded capacity.



(a) Modeling of wire-AND fault



(b) Modeling of wire-OR fault



(c) Modeling of dominant fault, node1 dominate node 2

Figure 3 Modeling of bridging fault

2.1.4 Other Fault Models

There are lots of variety of fault models; for instance, we can classify transistor-level stuck-at fault [6] as stuck-on and stuck-off . Stuck-off can be modelled as a conventional stuck-at fault as well. The performance of a stuck-on fault is never conducted because a stuck-on fault always conducts a transistor.

IDDQ fault [7] power supply current is larger than the expectation, but the circuit may pass logical testing but it's in a static condition, and it can be utilized in any transistor-level stuck-at fault testing.

2.2 Test generation methods

Test generation is a way of producing an effective set of vectors that will achieve high fault coverage for a specific fault model. Due to imperfect design or manufacturing process, it is possible to have defects in chip. The main purpose of test generation is to produce a set of test vectors that will help to detect any fault in the chip. Figure 4 illustrates a high-level concept of test generation. In this figure, the circuit at the top is defect free, and for any defective chip, which is functionally different from the defect-free one there must exist some input that can differentiate the two [8].

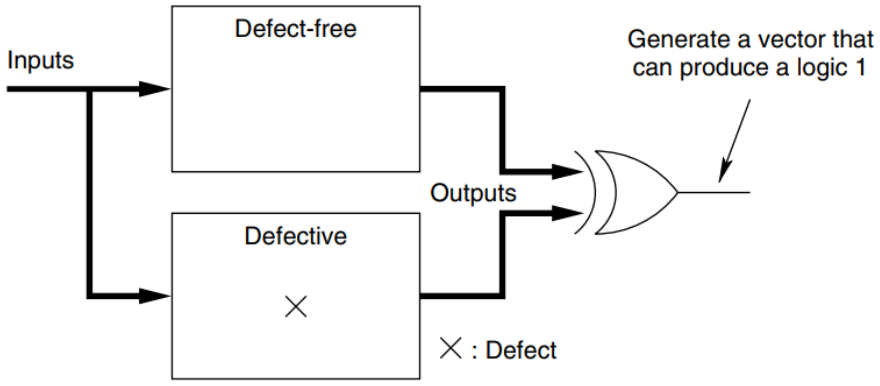


Figure 4 Conceptual view of test generation

If the ATPG engine is capable of delivering high-quality test patterns that achieve high fault coverages and small test sets, DFT would no longer be necessary. It is difficult and unrealistic to generate vectors targeting all possible defects that could potentially occur during the manufacturing process.

2.2.1 Random test generation

Random test generation is one of the most simple methods for generating vectors. The disadvantages of RTG are that due to difficult-to-test-faults, the test set size may grow enormous, and the fault coverage may not be adequately high. At the primary inputs, logic values are randomly generated in RTG. Since the pseudo-random number generator is used, the random test set is not truly random [8].

A random test set T is measured as the probability that a random test set can detect entire stuck-at faults in the circuit. For N random vectors, the test quality t_N defines the probability that all the detectable stuck-at faults are detected by these random vectors N. Therefore, the test quality of a random test set greatly depends on the CUT [8].

For example, consider a circuit with an eight-input AND gate (or equivalently a cone of seven two-input AND gates), illustrated in Figure 5 While achieving a logic 0 at the output of the AND gate is quite easy but to get a logic 1 is bit difficult. A logic 1 requires all the inputs to be at logic 1 itself. If the random pattern generation assigns each primary input value with an equal probability of logic 1 or logic 0, the chance of getting eight logic 1's simultaneously would only be $0.5^8 = 0.0039$. Especially, the AND gate output stuck-at-0 fault would be challenging to test by the RTG. These faults are known as random-pattern resistant faults [8].

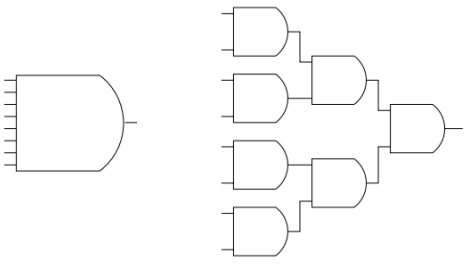


Figure 5 Two equivalent circuits

The quality of a random test set totally depends on the underlying circuit. More random-pattern resistant faults will reduce the quality of the random test set. In order to overcome the problem of targeting random-pattern resistant faults, biasing is required so that the input vectors are not viewed as uniformly distributed [8].

Another issue with the random test generation is the number of random vectors needed. Given a circuit with *n* primary inputs, there are clearly 2^n possible input vectors [8]. The probability of detecting fault f by any random vector can be expressed as:

$$d_f = \frac{T_f}{2^n}$$

where T_f is the set of vectors that detects fault f. As a result, the probability that a random vector will not detect f is:

$$e^f = 1 - d_f$$

Thus, for given *N* random vectors, the probability that none of the *N* vectors detects fault *f* is:

The probability that at least one out of *N* vectors will detect fault *f* is:

$$1 - (1 - d_f)^N$$

If the detection probability, d_f, for the hardest fault is known, *N* can be calculated by the following inequality

$$1 - (1 - d_f)^N \geq p$$

where *p* is the probability that *N* vectors should detect fault f.

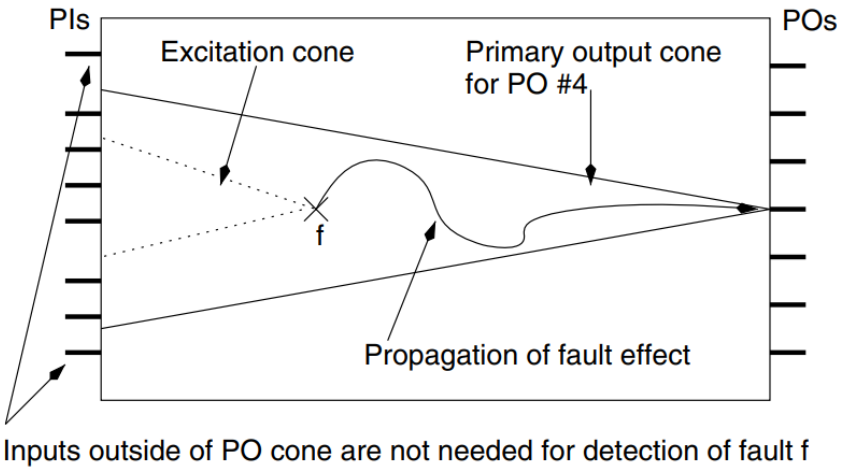


Figure 6 Detection of a fault

2.3 Fault simulation and fault diagnosis

Fault simulation is the process that figures out the detected and undetected faults, which mainly comes for testing and simulating stuck-at failures than can be easily adapted to test and simulate transition faults [9]. It is more challenging than logic simulation because when affecting one, the amount of computation is proportional to the number of test patterns, circuit size, and the number of modeled faults. It also noted that fault simulation is a reverse process of ATPG to a certain extent [10].

Basically, a fault simulator applies all the generated test patterns to the target fault as well as the fault set. After then simulate failures and observe output responses. We can call a circuit a suitable circuit if the output of the faulty circuit is different from the expected; otherwise, we get a detected fault. To improve fault simulation performance, various fault simulation techniques have been developed. We have Serial Fault Simulation, Parallel Fault Simulation, Parallel-Pattern Fault Simulation, Deductive Fault Simulation, Concurrent Fault Simulation, Differential Fault Simulation to be mentioned.

Now, in any example, we simply illustrate fault simulation, and fault detection is a natural process due to the outputs of the fault-free and faulty circuits being either 1 or 0, and they are different. But in practical cases, the decision of fault detection is a bit more complicated than we think. For instance, suppose the stuck-at-zero fault occurs at the enable input and is forced to 0. Then the tri-state buffer's output is floating. Now, the fault is detected or not is unclear, because the logic value of a floating signal accidentally

corrects or maybe the same as the correct value. Finally, some faults may cause the faulty circuit behavior to deviate significantly from the right response; for example, stuck-at faults on clock signals.

2.4 Turbo-Tester as a toolbox for test

Turbo tester (TT) is a powerful tool for testing digital circuits. It can work on both Windows and Linux operating systems. Turbo tester (TT) can be used for different purposes such as test patterns generation, Build in self-test emulation, Fault simulation, Multi valued simulation, Test set optimization, and many more [11].

Figure 7 presents the flow of the turbo tester system. Test pattern generators and fault simulators are available for combinational and sequential circuits. Combinational circuits can be tested by random, deterministic and genetic algorithm based methods, while for sequential designs a random ATPG is available [11].

.

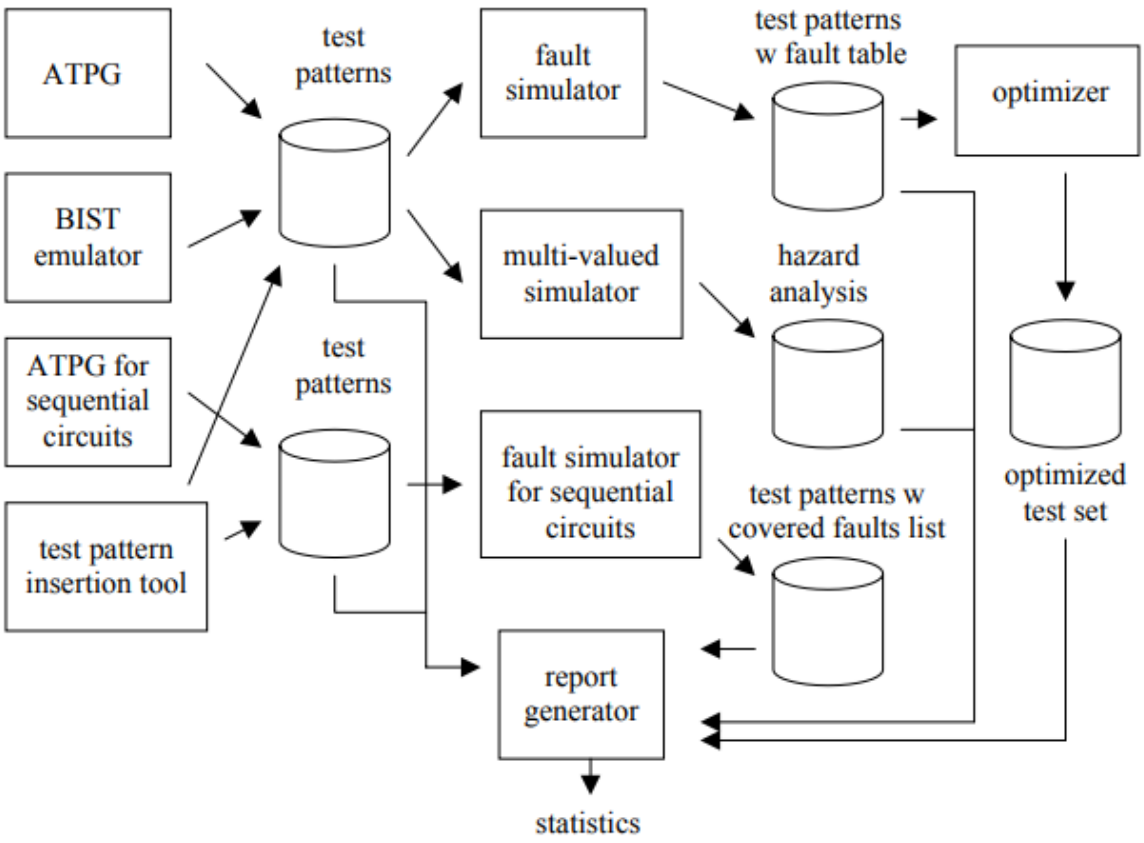


Figure 7 System flow of Turbo Tester [11]

All the tools in the Turbo Tester system operate on the design model of Structurally Synthesized Binary Decision Diagrams (SSBDD). Unlike Binary Decision Diagrams (BDDs), they are capable of representing gate-level structural faults [11].

In Turbo Tester, there are two options for generating SSBDDs:

1. Macro-level SSBDDs
2. Gate-level SSBDDs

In the gate-level SSBDDs, each gate is represented by corresponding BDD, and the circuit model does not differ from ordinary gate-level model.

In Figure 8, the design interface of the Turbo Tester system is presented.

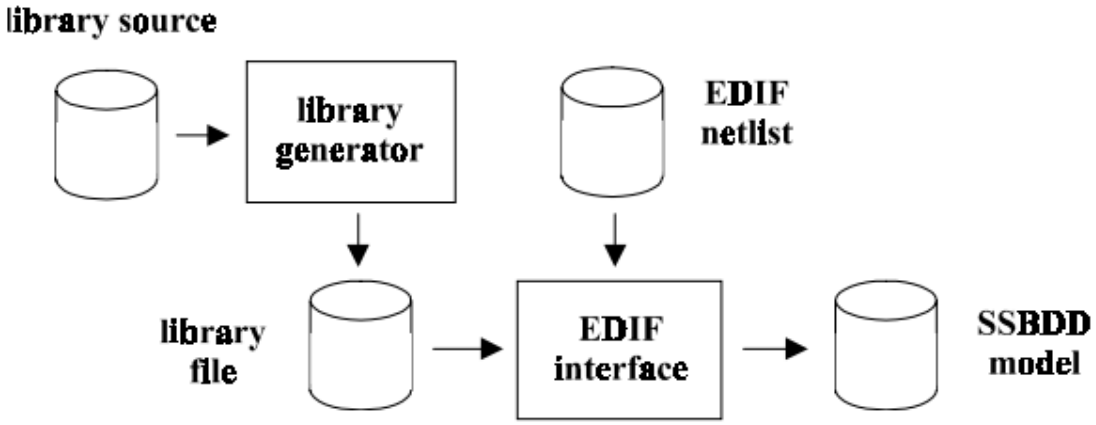


Figure 8 Turbo Tester Design Interface [11]

All of the Turbo Tester Automated Test Pattern Generators (ATPG) are able to reading information about detected faults from a test pattern file that are already exists. This means that the ATPGs of the system can be run in an arbitrary sequence, where one ATPG generates test patterns covering a set of faults, and another test pattern generator continues to target the faults not detected by the previous one [11].

In this thesis, we work with automated test pattern generators (ATPG), where we generate a table of test patterns for a circuit that helps us to detect transition delay faults.

3 Overview of delay fault simulation methods

3.1 Delay fault model

There are three delay fault models to be considered: The first one is the transition fault model, the second gate-delay fault model, and the final one is the path-delay fault model. In the simple design, it is assumed that each gate has a given fall delay from almost every input to the output pin, and here the interconnects have given rise delays. This happens because the gate pin to pin delays and the interconnect delays can be combined. Transition Gate-delay models are used to defects lumped at gates when the path-delay model addresses deficiencies that are distributed over several barriers for representing delay.

The delay fault for the transition fault model affects only one gate in the circuit—generally two transition faults associated with each of the gates. The first one is a slow-to-rise fault, and the second one is a slow-to-fall fault. It seems that each fault-free circuit has some simple delay. Under the transition fault model, the extra delay is large enough to prevent the transition from reaching any primary output. No matter if it is at the time of observation or not, and the delay faults increase this delay frequently.

In other words, we can observe the delay fault independently. Whether the transition propagates through a long or a short path to any primary output, this model is also referred to as the gross-delay fault model. Now, to detect a transition fault from a combinational circuit, it is always necessary to apply two input vectors, V =<1 2>. Here the first vector initializes the channel, which is 1, and the second vector activates the fault and propagates, which is 2. Also, Vector 2 can be found using stuck-at fault test generation tools.

For instance, while testing a slow-to-rise transition, we have to initialize the first vector fault site to 0, and the second vector fault site to a stuck-at-0 fault at fault. What are the advantages then? The main advantage of the transition fault model is that the number of faults in the circuit is relatively small; also, the stuck-at fault test generation and fault simulation tools can be easily modified for handling transition faults.

The gate-delay fault model is quantitative. It takes into account that the circuit delays to determine the ability of a test to detect a gate-delay defect. But, it is necessary to specify the delay size of the fault. The limitations of the gate-delay fault model are as similar as the transition fault model. The main advantage of this model is that the number of faults is linear, which is also identical to the number of gates in the circuit. Here, under the path-

delay fault, if the delay of any of its paths exceeds a specified limit, then the model of a circuit is considered faulty.

A delay defect on a path can be observed through the path by propagating a transition. However, a path-delay fault specification consists of a physical path and a transformation that will be applied at the beginning of the path. Where the delay or length of the path represents the sum of the setbacks of the gates and interconnections on that path. The major limitation of this fault model is, the number of paths in the circuit can be vast, and for this reason, testing all path-delay faults in the circuit is considered to be impractical.

3.2 Transition delay fault simulation

In the transition fault model, the delay fault occurs either in a rising and falling transition. There are two kinds of transition faults.

1. Slow-to-rise
2. Slow-to-fall

Generally, the transition faults depend on the input and outputs of logic gates. As time becomes larger, the slow-to-rise fault behaves like an s-a-0 fault. In the same way, the slow-to-fall transition behaves like an s-a-1 fault. In addition, to propagate the effect of the transition, a transition follows that a test for transition fault should create an applicable transition at the pointy of the fault [12].

Initialization and *transition propagation* are the two definite patterns required for the transition fault. As the name illustrates, the initialization pattern places the initial transition value at the point of the fault. Then final transition value is placed at the point of the fault by the transition propagation pattern, and after that, it propagates the effect to a primary output. Figure 9 illustrates how the two pattern sequence creates a transition.

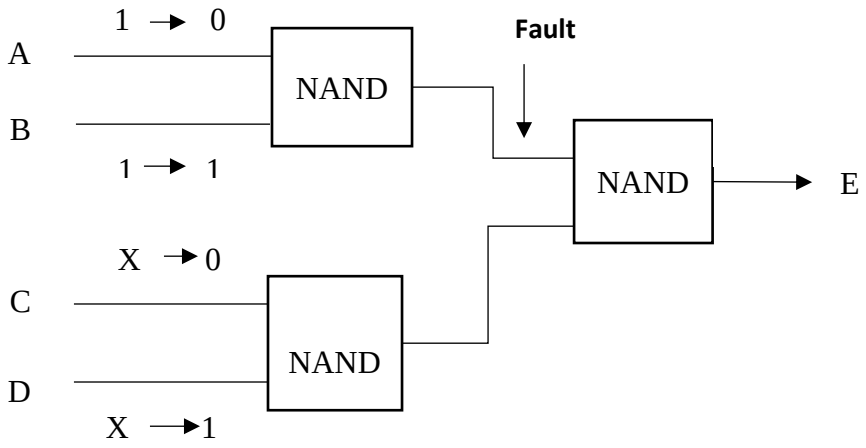


Figure 9 Two pattern sequence creating a transition

From figure 9, we can be able to see that in the inputs, initial transition values are placed, and then the final transition values are placed by the transition propagation pattern. The pattern that detects the corresponding stuck-at-fault and the transition propagation is identical [12].

Let us discuss the path delays on the size of the delay faults, the minimum delay size detectable by the test of the transition fault will always differ . Let us consider an example shown in figure 10.

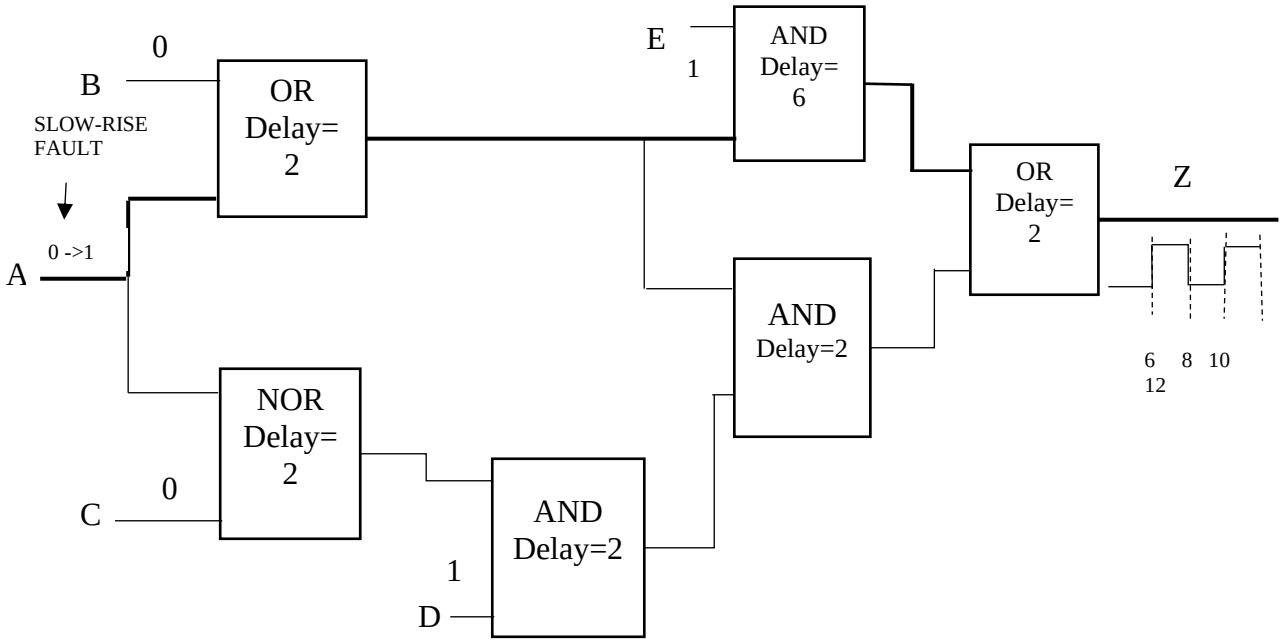


Figure 10 Effects of path delays on the size of detectable delay faults

The logic values and transitions are given on the signals. From figure 10, we can see, the transitions at the primary inputs A, B, C, D, and E occurs at time 0 while the time at the primary output Z is 12 units. It is very difficult to say how small a delay fault could be before it is not detectable. The path delay is defined as the sum of the delay through the gates along the path [12]. All the delay faults on this path of size should be greater than the difference between the measured time, i.e., the time at the primary output and the path delay should always be detectable. Nevertheless, it not always true in all the case. To have a clear look, see figure 10, the slow-to-rise transition fault on the input A creates a sensitized path to the primary output Z. Here the measured time is 12 units, and the delay along the sensitized path is 10 units. Therefore the difference is 2 units. Hence this pair of patterns cannot detect delay faults of 4 units [12].

3.3 Simulation strategy

The transition fault simulator is a stuck-at-fault simulator intensified to identify the logic gates that experience a transition in a pattern relative to the prior pattern. The Parallel Pattern Single Fault Propagation (PPSFP) simulator is used here because of its efficiency. Here, the concepts of multiple pattern evaluation and single-fault propagation are merged

to perform a simulation involving two values: Zero delay and 256 patterns per pass. Let us see how the PPSFP fault simulator works by the following algorithm.

1. Create a list of transition faults whose test coverage is to be evaluated
2. Assign the logic values to be placed at the primary inputs for the set of 256 patterns,
3. Regulate the previous states of primary inputs for this set of 256 patterns.
4. Perform a machine rank order simulation for the current set of 256 patterns along with the 256 previous states.
5. Simulate for each remaining transition fault by single fault propagation.

   For each fault perform the following steps

   a. Determine which of the 256 patterns accomplished a transition at the point of the fault by comparing the current state values with the previous state values.
   b. Consider only the patterns that experienced a transition and propagate fault values at the forward beginning at the point of the fault and continued until there are no longer different values from the good machine ranked values
   c. Eliminate the fault from further simulation when a fault becomes detectable at an observable point.
6. Repeat the steps from 2-5 until all the patterns are simulated.

Transition fault simulation makes sure that a delay fault of arbitrary size is observed at a measurable time by a set of test patterns. Compare to the traditional method our proposed method is much more efficient. We described the proposed method, and the experiment results in chapters four and five.

3.4 Transition delay fault test generation

Transition delay fault test generation is the opposite task to fault simulation, where for each fault, a test pair must be generated, where the first pattern initializes the state of the circuit, and the second pattern produces the transition of the signal from the state value to the opposite value. Then, the transition of the signal (the opposite value) must be propagated to the observable output of the circuit.

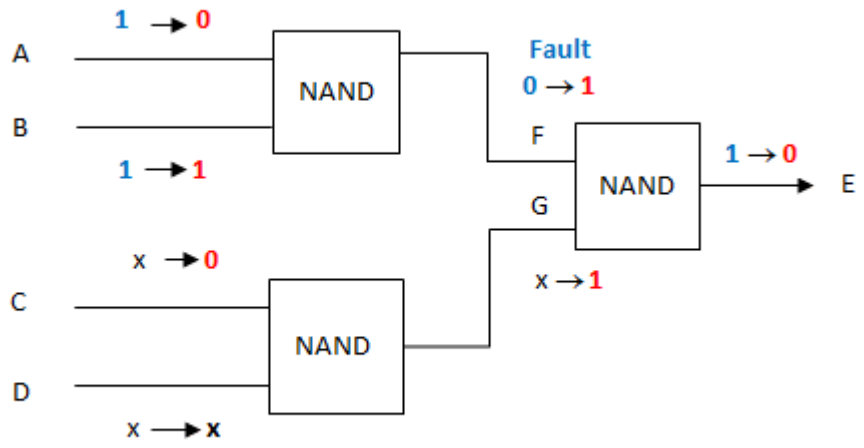


Figure 11 Test generation process

The figure 11 illustrates the test generation process for the fault "slow-to-rise" 0→1 on the line F of the circuit. For initialization of the state 0 on the line F, we have to apply the input signals A = 1 and B= 1. This is the first step of the process.

On the second step we have to produce the change of the signal 1→0 on the input A, and propagate this transition via NAND gate to the fault site F, and produce there the rise of the signal 0→1. For that, we need to keep the state on the input B = 1.

Further, to propagate the transition on F to the output E, we need the signal G = 1, and for that, in its turn, we have to apply the input signal C = 0 (or D = 0).

If there is delay of the signal on the line F (due to the signal delay in the previous NAND gate), this delay will manifest himself as the fault "slow-to-rise" 0→1 on the site F. The delay fault propagates further to the output E, and because of the invertor of the output gate, the delayed signal transforms to the delay "slow to fall".

The signals generated in the first step for initialization purposes are notated in the circuit in blue color, and the signals generated in the second step of SAF generation are notated in the circuit in red color. The values "x" of signals mean "don't care", because they are not involved in the test generation process.

From this example we can see, that the TDF test generation (as example, in case of "slow-to-rise" 0→1) for the given node F can be considered as a procedure consisting of two tasks:

1. initialization of the state of the circuit on F = 0, using backward signal justification procedure, and

2. test generation for detecting if there is a fault „slow-to-rise“, i.e. that instead of the transition of the signal on F, it remains constant.

It is easy to see, that the second step is equivalent to the test generation for the SAF type of fault $F \equiv 0$.

The same consideration takes place also for detecting the TDF fault “slow-to-fall” $1 \rightarrow 0$.

It is well known that the SAF test generation procedure, consisting of three tasks: signal initialization, signal propagation, and signal backward propagation for justification of assigned signals, has exponential complexity [2]. The same is valid for the first step of the described procedure of initialization, because it involves also the task of signal justification.

From that, the motivation of this research work grew out: to avoid the cost expensive exponential TDF test generation using structural circuit analyses, investigate the feasibility of deriving the TDF test directly from the test set generated already for the SAF type of faults, which is the standard approach regarding test generation for digital circuits.

The following chapters of the thesis are devoted to development of this new approach to TDF test generation.

4 Development of a new method for transition delay test generation

4.1 General idea of the method

The goal of the master thesis is test sequence generation for testing transition delay faults (TDF) in combinational circuits from the given set of test patterns, which were generated for stuck-at-faults (SAF). While for testing each SAF fault, a single test pattern t_j is sufficient, then for testing each TDF, a pair $T = (t_i, t_j)$ of test patterns is needed. The first pattern t_i is for initialization of the circuit, whereas the second pattern t_j activates a TDF fault and propagates its effect to some primary output.

The test pair $T = (t_i, t_j)$ can be also generated in a deterministic way by an Automated test Pattern Generation (ATPG) tool if available. However, the complexity of the generation of two pattern sequences for TDF is much higher than single pattern generation for SAF. In this thesis the goal is to extend the SAF test, generated by single pattern ATPG to the TDF test. To solve this task, two algorithms were developed, described in the next sections. In this Section, the concept of these algorithms is present.

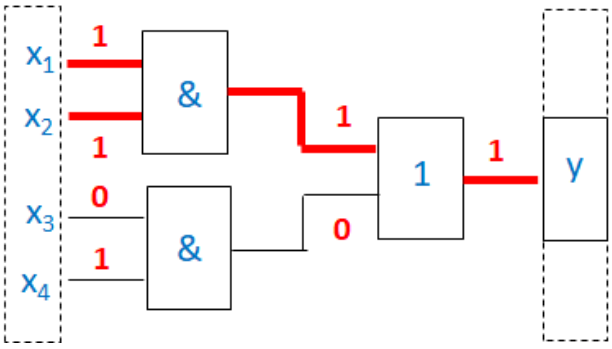


Figure 12 Combinational circuit between two registers with applied single input pattern

Consider a circuit in Fig.12, which consists a combinational part

$$y = (x_1 \,\&\, x_2) \vee (x_3 \,\&\, x_4)$$

between 4-bit registers *X* and a register *Y* with highlighted trigger *y* as the output of the circuit. The pattern t_j can be found using Stuck-at-Fault test generation tools. For example, in Fig 12, a pattern X = (1101) is applied with expected value 1 at the output y. The pattern activates two SAF faults SAF/0 at the inputs x_1 and x_2. In this circuit the faults are propagated along the highlighted in red paths. Note, all other faults on the

internal lines of these two paths are also detected. The faults $x_1 \equiv 0$ and $x_2 \equiv 0$ serve as representative faults of all faults detected by the pattern X = (1101).

Test pattern 1			
x_1	x_2	x_3	x_4
1	1	0	1

SAF table			
x_1	x_2	x_3	x_4
1	1	x	x

TDF table			
x_1	x_2	x_3	x_4
x	x	x	x

Figure 13 Test information regarding the test pattern applied to circuit in Fig 12

In Fig.13, a test information regarding the test pattern applied to circuit in Fig.12 is presented, which consists of the Table of test patterns, SAF table and TDF table. In SAF table, the detected faults $x_1 \equiv 0$ and $x_2 \equiv 0$ are highlighted in red (note, the notation of faults is opposite to the signal values on the lines). No faults at x_3 and x_4 are detected, which is notated by symbol x. Also, no TDF fault is detected, because this is the first pattern, and the circuit is not initialized.

Fig. 14 illustrates the situation of testing a slow-to-rise Transition Delay Fault on the input x_1. Two test patterns (a pair of patterns) X_1 = (1101) and X_2 = (0101) are applied to the inputs of the combinational part of the circuit with ` The pattern activates a path from x_1 to y , highlighted in red , along which all TDF are detected.

Note, the second pattern X_2 detects also two SAF $x_1 \equiv 1$ and $x_3 \equiv 1$ (the path from x_3 is highlighted with black bold lines). However, at the input x_3 no TDF is detected, because there is no transition at x_3 produced by this test pair.

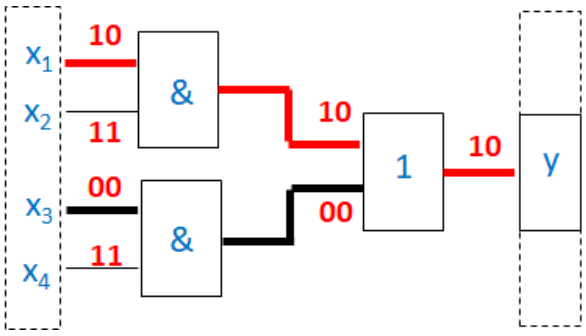


Figure 14 Combinational circuit between two registers with applied two input patterns

Test pattern 2 with SAF table			
x_1	x_2	x_3	x_4
1	1	0	1
0	1	0	1

TDF table			
x_1	x_2	x_3	x_4
x	x	x	x
0	x	x	x

Figure 15 Test information regarding the test pattern applied to circuit in fig 11

SAF Test patterns

TDF generation (Algorithm 1)

T	Test patterns with SAF cover				TDF table				TDF cover			
	x_1	x_2	x_3	x_4	x_1	x_2	x_3	x_4	x_1	x_2	x_3	x_4
1	1	1	0	1	x	x	x	x	x	x	x	x
2	0	1	0	1	0	x	x	x	0	x	x	x
3	0	1	1	1	x	x	1	x	0	x	1	x
4	1	0	1	0	x	0	x	0	0	0	1	0

Figure 16 TDF test generation for the circuit depicted in Fig.12 using Algorithm 1

In the following, the idea of manipulations of the test information (test pattern table and fault table) generated by ATPG for detecting a set of single patters for detecting SAF faults. Such an information is presented in the leftmost table in Fig.16, where the test pattern table and fault table are combined. The table represents 4 test patterns and 4 input variables. The entries of the table represent the values of variables for the given 4 patterns. By red the cases, where a fault is detected, are marked. For example in the first row two 1s mean that the SAF-type faults $x_1 \equiv 0$ and $x_2 \equiv 0$ are detected. If a SAF is detected, this will be a prerequisite, that there is a possibility that also a TDF is detected. The condition of detecting the TDF is to have in this place a transition.

Algorithm 1 was developed for creating the TDF Fault table as a matrix with entries 0, 1 and x, where 0 means that at this pattern the TDF/(1→0) is detected, 1 means that at this pattern the TDF/(0→1) is detected, and x, if no TDF is detected.

The first row in the TDF fault table will contain only symbols x, because no TDF can be detected by the first test pattern. The latter can serve only as initialization of the circuit. In the second row we analyses the variables where SAF is detected and check if there is also a transition. If yes, then we mark the respective TDF as detected, if not then no TDF is detected. The middle table in Fig.16 illustrates the result of this analysis for all the rows

of the SAF table. The right-most TDF cover table stores step-by-step the current states of the analysis carried out for all rows of the SAF table.

There entries of the TDF cover table mean the following: x – means no TDF not yet detected, 0 – means TDF/(1→0) is detected, 1 – means TDF/(0→1) is detected, and ja – means that both faults TDF/(1→0) and TDF/(0→1) are detected.

From the presented example, it can be seen that the sequence of 4 test patterns, which detects 100% of SAF faults (with total of 8 faults), only half of TDF faults, i.e.50% are covered.

The second part of TDF generation consists of adding additional test patterns to the TDF test sequence by selecting useful patterns from the initial SAF test set generated by the single pattern ATPG.

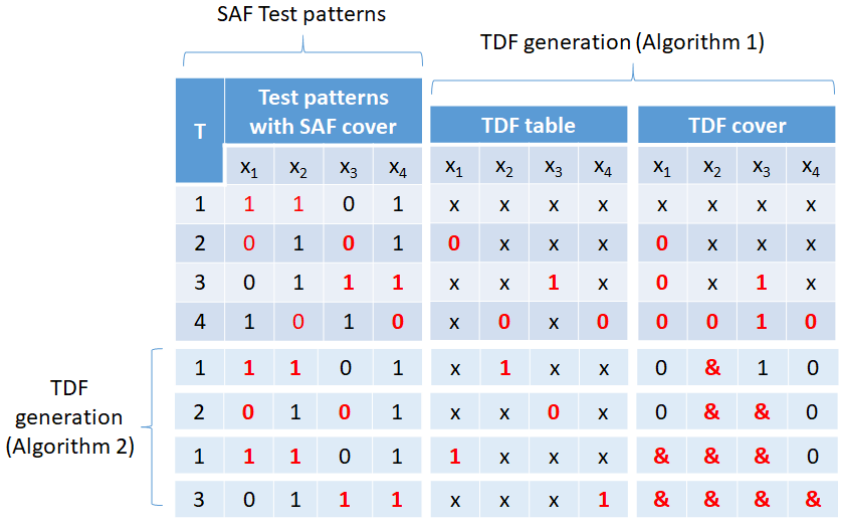


Figure 17 Full process of TDF test generation for the circuit depicted in Fig.12 using Algorithms 1 and 2

The idea of Algorithm 2 is illustrated in Fig.17. The initial test sequence of 4 patterns will be now extended. In each step, a pattern is chosen which satisfies at least for one variable the following two conditions:

- The fault TDF/(0→1) is not detected, but SAF/1 is detected,
- The fault TDF/(1→0) is not detected, but SAF/0 is detected.

If one of these conditions is satisfied, the pattern will be included into the sequence, otherwise the search continues. The algorithm is finished if either all TDF are detected, or a scan through the given test pattern table will not be able to find any more pattern which would satisfy one of these two conditions.

4.2 Using Turbo-Tester for generating input data for delay test generation

Turbo tester is an automated test pattern generator (ATPG) and capable of reading information about already detected faults from a test pattern file.

In this thesis, we use Turbo-Tester for generating input data, more specifically test patterns and Stuck at Fault (SAF) table. We use deterministic test pattern generator in our case [11].

4.2.1 Deterministic test pattern generator

Command: generate

Input: SSBDD model file (.agm)

Output: test pattern file (.tst), list of redundant faults (.red)

Syntax: generate [options] <design>

Design: Name of the design file without .agm extension.

options:

-backtracks <*number*>	Maximal number of backtracks. Default is 10.
-test_per_fault	Generates test for every fault in the circuit. Preserves don't care values.
-vector_limit<*limit*>	Maximal number of generated patterns. Default is 1000.
-fault_table	Perform fault simulation for the final patterns.
-infile<*file*>	Read data about covered faults from test patterns file *file*.

[11]

For generating the test patterns and SAF table, the command looks like below :

generate -backtracks 1000 -fault_table C880

To get better fault coverage, the number of *backtracks* depends on different circuits. By using this command, Turbo-tester generates the following report:

Reading SSBDD-model file c880.agm... OK
Allocating test patterns... OK
Generating tests...
Tested 1550
Untestable 0
Aborted 0
Fault coverage: 100.000000

Fault efficiency: 100.000000
84 Vectors
30 Backtracks
Time, used by process: 0.014000
Fault Simulation... OK
Time, used by process: 0.030000
Writing test patterns file c880.tst... OK

In the .tst file, test patterns and SAF patterns table are generated, which we use as input for delay test generation. Alongside the patterns, it also generated the fault coverage vector and the number of fault coverage as well [11].

4.3 Algorithm for converting stuck-at-fault table to delay fault table

This algorithm was developed for creating the transition delay fault(TDF) table as a matrix with entries 0, 1 and x, where 0 means that at this pattern the TDF/(1→0) is detected, 1 means that at this pattern the TDF/(0→1) is detected, and x, if no TDF is detected.

……
Algorithm 1- Converting stuck-at-fault table to delay fault table
……
.

Input: Test patterns table, SAF table, Mapping table.

Output: Delay fault table.

Notation:

t- Number of total pattern.

i- nodes

j- variables

VT- Variable table.

FT- Fault table

……

1. ***FOR*** *all test patterns* ***t = 2,3****, …, T*
2. ***FOR*** *all nodes* ***i = 0,1, …, n***
3. ***IF*** *in* ***FT*** *node* ***(t,i)*** $\neq$ *x* ***THEN*** *find* ***j = NVA (i)***
4. ***IF*** *in VT var* ***(t,j) = 0*** *THEN*
5. ***IF*** *in* ***VT*** *var* ***(t-1,j) = 0****, THEN in* ***FT (t,i) := x***
6. ***IF*** *in* ***VT*** *var* ***(t,j) = 1*** *THEN*

7. ***IF** in **VT** var **(t-1,j) = 1**, then in **FT (t,i) := x***

8. ***END** FOR*

9. ***Remove** the row **t=1** from **FT***

10. ***END** FOR*

……

4.3.1 Tables for SAF faults

Fault table (FT):

Table 1 Converting SAF to Delay Fault Table

Test	Nodes
t	*i* 0 1 … n
1	Node (i) ∈ {0,1,x}
2	
…	
T	NVA (i)

Variable table (VT) (simulated test patterns):

Test	Variables
t	*j* 0 1 … m
1	Var (j) ∈ {0,1}
2	
T	

Test	Variables
t	*j* 0 1 … m
1	Var (j) ∈ {0,1,}
2	
T	

The table above (Table 1) explains the working principle of the proposed algorithm step by step.

The main goal of algorithm 1 is to Converting stuck-at-fault table to delay fault table. As an input of this program three table needed to start analysis, variable table, simulation table, and mapping table. For creating the mapping table from agm model we had to develop another program.

The algorithm starts working from the second pattern of SAF table. First it check the node whether it is not X (**(t,i)** ≠ x). If it finds stuck at fault either stuck-at-0 or stuck-at-1 it go to the variable table using mapping value. In the variable table, it checked two patterns (t,j) and (t-1,j). If any transition happens (0 to 1 or 1 to 0) between those two patterns, we detect a delay fault and then update the Stuck at fault to delay fault. The process will continue through the last node of delay fault table. The first row in the TDF fault table will contain only symbols x, because no TDF can be detected by the first test pattern. The latter can serve only as initialization of the circuit. The new FT is the fault table for delay faults, where 0 means detection of the fault $1 \rightarrow 0$,1 means detection of the fault $0 \rightarrow 1$, and X means no fault is detected.

4.3.2 Mapping table

The aim of creating the mapping table is to work with Fault table and variable table. The number of nodes and variables are different, so we have to map the nodes from the fault table to variable in the variable table. For creating this table, we need the agm model of circuit. An agm model of C17 circuit is given below:

```
STAT#    15 Nods,  14 Vars,  9 Grps,  5 Inps,  0 Cons,  2 Outs

MODE#   STRUCTURAL


VAR#   0:  (i_______)  "i_5"
VAR#   1:  (i_______)  "i_4"
VAR#   2:  (i_______)  "i_3"
VAR#   3:  (i_______)  "i_2"
VAR#   4:  (i_______)  "i_1"

VAR#   5:  (________)  "i_3"
GRP#    0: BEG =   0, LEN =   1 -----
   0   0:  (____)      (       0       0)  V = 2     "i_3"

VAR#   6:  (________)  "inst_0>o"
GRP#    1: BEG =   1, LEN =   2 -----
   1   0:  (I___)      (       1       0)  V = 5     "inst_0>i_1"
   2   1:  (I___)      (       0       0)  V = 4     "i_1"
```

```
VAR#   7:  (________)  "inst_1>o"
GRP#    2: BEG =   3, LEN =   2 -----
   3   0:  (I___)      (       1       0)  V = 1     "i_4"
   4   1:  (I___)      (       0       0)  V = 5     "inst_1>i_2"

VAR#   8:  (________)  "inst_2>o"
GRP#    3: BEG =   5, LEN =   2 -----
   5   0:  (I___)      (       1       0)  V = 7     "inst_2>i_1"
   6   1:  (I___)      (       0       0)  V = 3     "i_2"

VAR#   9:  (________)  "inst_3>o"
GRP#    4: BEG =   7, LEN =   2 -----
   7   0:  (I___)      (       1       0)  V = 0     "i_5"
   8   1:  (I___)      (       0       0)  V = 7     "inst_3>i_2"

VAR#   10:  (________)  "inst_4>o"
GRP#    5: BEG =   9, LEN =   2 -----
   9   0:  (I___)      (       1       0)  V = 8     "inst_4>i_1"
   10   1:  (I___)     (       0       0)  V = 6     "inst_0>o"

VAR#   11:  (________)  "inst_5>o"
GRP#    6: BEG =   11, LEN =   2 -----
   11   0:  (I___)     (       1       0)  V = 9     "inst_3>o"
   12   1:  (I___)     (       0       0)  V = 8     "inst_5>i_2"

VAR#   12:  (_o______)  "o_2"
GRP#    7: BEG =   13, LEN =   1 -----
   13   0:  (____)     (       0       0)  V = 11     "inst_5>o"

VAR#   13:  (_o______)  "o_1"
GRP#    8: BEG =   14, LEN =   1 -----
   14   0:  (____)     (       0       0)  V = 10     "inst_4>o"
```

The agm model describes which node connected to which variable. GRP indicates the position of nodes, and V indicates the position of variables.

4.3.3 Fault coverage and fault vector

The Turbo-Tester generates the fault vector and fault coverage for stuck-at fault. By using algorithm 1 we got delay fault table from stuck at fault table. Then the next step is to update the fault vector to get fault coverage for delay fault. In this scenario, we had to develop another program that updates the fault vector and calculates the fault coverage for delay fault.

Equation to calculate the fault coverage is given below:

faultCoverage :=.COVERAGE<numberOfDetectedFaults> / <numberOfFaults> = <Percentage> %

……

Algorithm for update the fault vector and calculate the fault coverage

……

Input: Fault table.

Output: Fault vector, Fault coverage.

Notation:

FL=Length of Fault table

FV =Fault value
INDEX = Index of loop starting from 0
CL = variable

……

…….

1. Initialize count vector
2. FOR all test pattern i = 0, 1, 2... FL
3. FOR all test pattern j = FV[0], FV[1], FV[2]... FV[N-1]
4. IF j = 0 THEN count[0][INDEX] set to 1
5. IF j = 1 THEN count[1][INDEX] set to 1
6. IF j = X THEN count[2][INDEX] set to 1
7. Initialize res vector for store result
8. Initialize variables for count &, ZERO, and ONE
10. FOR all test pattern i = 0, 1, 2.... CL
11. if x and x then x
if 0 and x then 0
if 0 and 0 then 0
if x and 0 then 0
if 1 and x then 1
if 1 and 1 then 1
if x and 1 then 1
if & and x then &
if x and & then &
11. Input total number of fault
12 calculate Fault coverage

The principle of this algorithm is to update fault vectors and calculate fault coverage.

4.4 Algorithm for improving delay fault coverage by recombining the stuck-at-fault test patterns

Algorithm 2 was developed to get better fault coverage by adding additional test patterns to the
TDF test sequence by selecting useful patterns from the initial SAF test set generated by the
single pattern ATPG. a pattern is chosen which satisfies at least for one variable the following two conditions:

- The fault TDF/(0→1) is not detected, but SAF/1 is detected,
- The fault TDF/(1→0) is not detected, but SAF/0 is detected.

……
Algorithm 2- improving delay fault coverage by recombining the stuck-at-fault test patterns
…….

Input: Test patterns table, Fault table, Mapping table.

Output: Delay fault table.

Notation:

p=Number of new pattern added to FT and VT tables.

t= The number of the first pattern to be added.

k= The number of the second patterns to be added.

i- nodes

j- variables

VT- Variable table.

FT- Fault table

……

1. *p : = T*
2. ***FOR** all test patterns t = 1,2, …, T*
3. ***FOR** all test patterns k = 1,2,…, T*
4. ***Cop**y the pattern **FT(t)** into **FT(p+1)** (*** new first pattern into FT-table)*
5. ***Copy** the pattern **VT(t)** into **VT(p+1)** (*** new first pattern into VT-table)*
6. ***Copy** the pattern **FT(k)** into **FT(p+2)** (*** new second pattern into FT-table)*
7. ***Copy** the pattern **VT(k)** into **VT(p+2)** (*** new second pattern into VT-table)*

*8. Success := 0 (*** flag initialization to remember the event of new fault detection)*

9. ***FOR*** *all nodes i = 0,1, …, n (*** processing of the bits of a new added test pattern pair)*

10. ***IF*** *in FT node(p+2,i) ≠ x THEN find j = NVA (i) (*** bit analysis starts)*

11. ***IF*** *in VT var(p+2,j) = 0 THEN (*** condition for detection the fault 1→0)*

12. ***IF*** *in VT var(p+1,j) = 0, THEN in FT node(p+2,i) := x (*** fault is not detected)*

13. ***ELSE*** *Success := 1 (*** fault 1→0 is detected)*

14. ***IF*** *in VT var(p+2,j) = 1 THEN (*** condition for detection the fault 0→1)*

15. ***IF*** *in VT var(p+1,j) = 1, THEN in FT node(p+2,i) := x (*** fault is not detected)*

16. ***ELSE*** *Success :=1 (*** fault 0→1 is detected)*

17. ***IF*** *Success = 1 then p = p +2 (*** the added two patterns stay in the tables, otherwise*

they will be removed and p will be not changed)

18. ***END FOR***

29. ***END FOR***

20. ***END FOR***

4.4.1 Tables for recombinig the stuck-at-fault test patterns

Fault table (FT):

Table 2 Recombining the stuck-at-patterns

Test	Nodes
t	*i*
	0 1 … n
1	Node (i) ∈ {0,1,x}
2	
…	
T	NVA (i)
P+1	
P+2	

Variable table (VT) (simulated test patterns):

Test	Variables
t	*j*
	0 1 … m
1	Var (j) ∈ {0,1}
2	
T	
P+1	
P+2	

Test	Variables
t	*j*
	0 1 … m
1	Var (j) ∈ {0,1,}
2	
T	

The table above (Table 2) explains the working principle of the proposed algorithm 2 step by step.

The concept of the Algorithm 2 is as follows:

Take from both tables the first vectors and put them in the end of the tables, and apply for the two last vectors the Algorithm 1.

1. If no new faults will be detected by this pair of vectors, remove the pair.
2. If at least one new fault was detected, keep these two vectors.

This step will be repeated for all possible pairs of the patterns generated for SAF faults. The algorithm consists of three embedded loops:

1. The t-loop runs through all pairs (t,k) where t is changing t = 1,2,…, T
2. The k-loop runs through all pairs (t,k) where t = const, and k is changing k = 1,2,…,T
3. The i-loop runs through all bits of the given pattern pair (t,k), and checks if in the i-bit a new fault can be detected;

The algorithm contains a lot of redundancy, since the analysis of the same pattern pair (t,k) will be several times analyzed. On the other hand, the regularity of the approach keeps the algorithm simple. The length of the test can be further minimized using any fault coverage minimization algorithm, which allows to remove non-effective test pairs.

By implementing the algorithm 2 we got an updated fault table and variable table with additional row. Then we have to use the algorithm (4.3.3) to update the fault table for the latest result (New fault vector and fault coverage).

5 Experimental Results

5.1 The goals of experiments and characteristics of benchmark circuits

The goal of this work was to develop a new method for generating tests of Transition Delay Faults (TDF) using the tests generated for Stuck-at-Faults (SAF) and the SAF fault table. It is well known that the test generation for SAF is a very complex task and not scalable for complex circuits. The reason of the complexity is the combinatorics of the test generation task. Compared to SAF test generation, where the patterns are generated, the TDF test generation is even more complex, because instead of single patterns in the TDF case the test pairs (sequences of two patterns) have to be generated, which makes the combinatorics of the task even more complex.

The idea of the proposed method stands in recombining the test patterns generated for SAF, so that for each SAF pattern a proper other pattern from the same test pattern set was found to create a pair of patterns capable to detect the selected delay faults.

The main question of the feasibility and capability of the new method was, if there is in the test pattern set for each SAF a suitable pattern existing to form the needed pair for the respective TDF.

The task of TDF test generation was divided into two steps:

(1) a simulation phase, where for each detected SAF it was checked, if the previous test pattern in the test set had the proper signal value to produce signal transition for the respective SAF;

(2) a test generation phase, where for the not yet detected SAF, a suitable pattern was found from the same table to produce signal transition for the respective SAF.

The goal of the experiments was threefold:

(1) to demonstrate the feasibility and efficiency of the proposed new method,

(2) to compare the achieved TDF coverage with SAF coverage of the given set of test patterns, and

(3) to compare the time costs of the new proposed TDF test generation with SAF test generation time costs.

The experiments were carried out with Benchmark circuits' family ISCAS'85. The most important parameters of the circuits are highlighted in Table 3.

Table 3 ISCAS'85 benchmark circuits and their characteristics

Name	Number of input	Number of outputs	Number of Nods	Number of faults
1	2	3	4	5
C432	36	7	487	974
C880	60	26	775	1550
C499	41	32	1097	2194
C1355	41	32	1097	2194
C1908	33	25	1394	2788
C2670	233	140	2075	4150
C3540	50	22	2784	5568
C5315	178	123	4319	8638
C6288	32	32	4864	9728
C7552	207	108	5795	11590

Table 3 represents the following characteristics: numbers of inputs, numbers of outputs, numbers of nodes and numbers of faults. The number of faults is always equal to the double number of nodes, where each to each node two SAF corresponds (SAF/0, and SAF/1), and two TDF faults („slow-to-rise", and „slow-to-fall"). The parameters of circuits characterize in some extent the complexity of the circuit for test generation purposes. On one hand, the bigger is the number of nodes and the less is the number of outputs, the more difficult is to find a test pattern for faults. On the other hand, the bigger is the) number of inputs, the easier is to generate a test pattern. Another factor, which influences on the difficulty of test generation is the internal structure of embedded fanout reconvergencies which are the causes of the conflicts arising during propagation of faults to observable outputs. The arising conflicts in their turn will cause backtracks during test generation process for resolving the contradiction of signals.

The best parameter for characterising the complexity of the circuit is the time cost of deterministic test pattern generation. For SAF test generation experiments we used the Turbo Tester tool, which has been developed in the lab.

5.2 Discussion of the results of experimental research

The results of experiments are presented in Table 4, where the time costs for test generation and the achieved fault coverages are highlighted for both SAF test generation and TDF test generation.

Table 4 Fault Coverage and Time cost

Circuits		SAF generation (Reference data)		TDF generation (new method)					
				After 1st step		After 2nd step		Gain in time cost	
Name	# Faults	Fault cover %	Time, s	Fault cover %	Time, s	Fault cover %	Time, s	Total time, s	Gain, times
1	2	3	4	5	6	7	8	9	10
C880	1550	100	0.15	93.87	0.17	98.06	0.09	0.26	0,58
C5315	8638	99.29	34.10	95.70	1.42	98.29	0.79	2.21	15
C1908	2788	99.60	58.45	87.23	0.52	94.08	0.31	0.83	70
C499	2194	99.63	65.85	86.82	0.35	93.02	0.22	0.57	116
C1355	2194	99.63	66.45	86.28	0.32	92.98	0.21	0.53	125
C6288	9728	99.30	129	94.73	0.51	98.36	0.36	0.87	148
C432	974	95.55	36.70	64.98	0.09	84.49	0.05	0.14	262
C3540	5568	95.68	810	85.91	1.09	95.68	0.56	1.65	491
C2670	4150	95.68	1467	89.20	0.75	94.03	0.54	1.29	1137
C7552	11590	97.28	5544	93.44	2.65	97.15	1.47	4.12	1345

Columns 1 and 2 in Table 4 represent the circuits under test with the number of faults as the complexity parameter, columns 3 and 4 characterize the parameters of SAF test patterns – the fault coverage and time cost of generation, respectively, column pairs (5,6) and (7,8) represent the calculated in the work TDF test parameters (fault cover and test cost) for the first and second phases of test generation. The last two columns represent the main result of the work – the gain achieved by the new method compared to the reference method.

Regarding the pairs of columns (5,6) and (7,8), the efficiency of two test generation phases are compared. The column (5) represents the TDF cover achieved by SAF test generation without any influence of the test set, whereas the column (7) represents the TDF cover achieved by recombination of test patterns produced by SAF test generator (by inserting purposely selected patterns in different places in the same test pattern sequence.

The results of the column 5 are achieved by the special fault simulator developed in the thesis, and the results of column 7 are achieved by the second tool – TDF test generator – developed in the thesis.

The column 9 represents the total test generation time used by the two tools developed in the thesis.

In Table 4, two results of the thesis are compared with the reference SAF test generator: the fault coverage, and the time cost of test generation.

The column 10 illustrates the gain of time cost, achieved in this work compared with the reference SAF test generator. We see that the more complex is the circuit for structural deterministic test generation, the bigger is gain of using the TDF generator. In the best cases the gain is about three orders of magnitudes. Such a big difference between the methods is because the traditional deterministic test generation is using a search with exponential complexity, whereas the proposed method uses the search of linear complexity.

An exception is the circuit c880 with very low structural complexity, where the proposed method looses to the structural test generator. But the loss is less than a second and hence, has no practical meaning. Such exceptions may happen only in case of very low complexity of circuits regarding structural test generation.

Note, the TDF cover can never be better than SAF cover, because the condition of testing a TDF at a node in the circuit, the prerequisite for that is the existence of SAF test pattern. On the other hand, the ideal case of TDF test generation is to achieve the same fault cover as in case of SAF. In this experiment, this ideal result was achieved only for the circuit c3540 where both the SAF cover and TDF cover are equal 95.68%.

In general, if the TDF cover, achieved by the proposed method, is less than the SAF cover, then to achieve the same fault coverage as for SAF, traditional TDF generation methods should be used, which however have high complexity and are very slow.

6 Conclusion

This thesis aimed at proposing a new method for generating transition delay fault (TDF) tests using stuck-at-fault test (SAF) patterns. The goal for this research was to minimize the time cost of test generation and achieve delay fault coverage as much close to the SAF coverage as possible.

Chapter two discusses fault models, test generation methods, fault simulation and fault diagnosis methods and Turbo – Tester tools, which was used in the work as the reference tool. In chapter three, an overview of delay fault simulation methods was given. Chapter four covers the details about proposed method. Chapter five represents and discusses the experimental research results.

The practical results of the research work contribute with two new tools for development of delay test sequences:

- TDF fault simulator for calculating the TDF cover for the given test pattern sequence,
- TDF test generator for recombining the set of test patterns, developed for detecting SAF faults, into another test pattern sequence for detecting TDF faults

The experimental results show that the developed new method of TDF test generation for digital circuits can be characterized with the following properties:

- it has linear complexity, and outperforms the reference method, which has exponential complexity
- the method is well scaling due to its linear complexity, and can be efficiently use for high complexity digital circuits.

References

[1] S. K, “Transition Delay Faults,” in *IEEE*, 2004.

[2] M. Bushnell and V. Agarwal, Essential of Electronic Testing fro Digital, Memory & Mixed SIgnal VLSI Circuits, Boston, 2000.

[3] N. Jha and S. Gupta, Testing of Digital Systems, Cambridge University Press, 2003. Page 82.

[4] “Wikipedia,” [Online]. Available: http://en.wikipedia.org/w/index.php?title=Automatic test pattern generation &oldid=628793647. [Accessed 20 04 2020].

[5] A. Grout, Integrated Circuit Test Engineering: Modern Techniques, Springer Science & Business media, 2005.

[6] W. Sleszynski, J. Nieznanski and A. Cichowski, “Open-Transistor Fault Diagnostics in Voltage-Source Inverters by Analyzing the Load Currents, vol56, no 11, pp4681-4688,” in *IEEE Transactions on Industrial Electronics,*, Nov. 2009.

[7] J. Esch, “Prolog to IDDQ Testing for CMOS VLSI, vol. 88 n0. 4, pp. 542-543,” in *Proceedings of the IEEE* , April 2000.

[8] L.-T. Wang, C.-W. Wu and X. Wen, VLSI Test Principles And Architectures: Design For Testability, Morgan Kaufmann Publishers, 2006.

[9] A. Krstic and K.-T. Cheng, Delay Fault Tetsing for VLSI circuits, kluwer Academic Publishers, 1998.

[10] C. Alagappan, “Dictionary-Less Defect Diagnosis as Real or Surroagate Single Stuck-at-fault,” Auburn University, Auburn, 2013.

[11] “Turbo Tester Reference Manual. Version 3.99.03,” Tallinn Technical University, 2002.

[12] J. A. Waicukauski, E. Lindbloom, B. k. Rosen and I. Vijay S. Iyengar, “Tansition Fault Simulation,” in *IEEE Design & Test*, NewYork, April 1987.

[13] M. Geilert, J. Alt and M. Zimmermann, “On the Eficiency of the Transition Fault Model for Delay Faults,” Germany.

[14] M. M. V. Kumar, S. Tragoudas, S. Chakravathy and R. Jayabharathi, “Exact Delay Fault Coverage in Sequential Logic Under Any DElay Fault Model,” in *IEEE Transactions On Computer-Aided DEsign of Integrated Circuits And Systems, Vol. 25, No..12*, December 2006.

[15] P. Bernardi, M. S. Reorda, A. Bosio, P. Girard and S. Pravossoudovitch, “On the modeling of Gate Delay Faults by means of Transition Delay Faults,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2011.

[16] S. majunder, b. B. Bhattacharya, v. D. Agarwal and M. L. Bushnell, “A Complete Characterization of Path Delay Faults through Stuck-at-Faults,” Inida.

[17] T. Storey and J. Barry, “Delay Test Simulation,” in *proc. 14th Design Automation Conference, pp.492-494*, 1997.

[18] J. Carter, V. Iyengar and B. Rosen, “Efficient Test Coverage Determination for Delay Faults,” in *Proc. Int. Test Conf.,*, 1987.

[19] M. Schulz and F. Brglez, “Accelerated transition Fault Simulation,” in *24th Design Automation Conference*, 1987, pp.237-243.

[20] C. Lin and S. Reddy, “On Delay Fault Testing in Logic Circuits,” in *IEEE Trans. on CAD* , September 187, pp.694-703.

[21] Y. Levendel and P. Menon, “Transition Faults in Combinational Circuits: Input Transition Test Generation and Fault Simulation,” in *Proceedings of International Fault Tolerant Computing Symposium, PP. 278-283*, July 1986.

[22] A. Majhi and V. Agarwal, “Tutorial: Delay Fault Models and Coverage,” in *11th International Conference VLSI Design, pp. 364-369*, 1998.

Appendix 1 – Program Description and Manual

This sections describes how our experiments were performed. It contains a step by step explanation of how transition delay fault generate and calculate the fault coverage.

Each experiment has its separate folder and have some steps.

1. The folders and files for the experiments can be downloaded through this link: https://github.com/ripon-cse109/tdf/tree/master/TRANSITION%20DELAY%20TEST%20GENERATION%20USING%20STUCK-AT-FAULT%20TEST%20PATTERNS

2. The experiment can be performed on both Windows and Linus operating system.

3. The folder name Circuits contains ISCA'S 85 circuits which can be use for generating stuck-at-fault test patterns by using Turbo-Testor.

4. The folder Mapping contains *mapping.py*, is a python code which is use to create the maping table from agm model. As a input *pre_var.txt* have to be contain with an agm model.

5. *Sol.py* in Algorithm 1 folder contain proposed algorithm 1 which generate the transition delay fault table in *output.txt file*. As a input we have to provide variable table, fault table from stuck-at-fault. Alongside this we also have to provide mapping table.

6. *Fcoverage.py* in fault coverage folder consist of python code to upadate fault vector and calculate fault coverage.

7. Algorithm 2 folder have *sol.py*, which execute the algorithm 2.

Appendix 2 – Source Code

Algorithm 1 *(sol.py)*

```
import time
start_time = time.time()
"""
Open fault table as fault_text
"""
with open("fault.txt") as fault_text:
    fault_val = list(fault_val.strip() for fault_val in fault_text)
#-----------------------------
"""
Open simulation table as sim_text
"""
with open("simulation.txt")  as sim_text:
    sim_val = list(sim_val.strip() for sim_val in sim_text)
#-----------------------------
"""
Open variable table as var_text
"""
with open("var.txt") as var_text:
    var_val = list(map(int, var_text.readline().split()))
#-----------------------------
"""
storing fault table length and
initiating variable for storing result
"""
fault_len = len(fault_val)
res = [[] for resi in range(fault_len)]
# initiate variables for count 0, 1, X
count = 0
countx = 0
# Loop through fault table row
for i in range(1, fault_len):
    fault_row_len = len(fault_val[i])
    # for all patterns in fault table rows
    for j in range (fault_row_len):
        # calculating result if zero or one found in fault table
        if fault_val[i][j] == '0' or fault_val[i][j] == '1':
            if (sim_val[i][var_val[j]] == '1' or sim_val[i][var_val[j]]
== 'h' or sim_val[i][var_val[j]] == 'H') and (sim_val[i-1][var_val[j]] ==
'0' or sim_val[i-1][var_val[j]] == 'l' or sim_val[i-1][var_val[j]] ==
'L'):
                res[i].append('1')
                count += 1
```

```
            elif (sim_val[i][var_val[j]] == '0' or sim_val[i][var_val[j]]
== 'l' or sim_val[i][var_val[j]] == 'L') and (sim_val[i-1][var_val[j]] ==
'1' or sim_val[i-1][var_val[j]] == 'h' or sim_val[i-1][var_val[j]] ==
'H'):
                res[i].append('0')
                count += 1
            else:
                res[i].append('X')
                countx += 1
        else:
            res[i].append('X')
            countx += 1
#-----------------------------
# appending X in first row of result
for i in range(fault_row_len):
    res[0].append('X')
    res_len = len(res)
# output final table as output.txt
with open("output.txt", "w") as output:
    for i in range(res_len):
        res_row_len = len(res[i])
        for j in range(res_row_len):
            output.write(res[i][j])
        output.write("\n")
fault_text.close()
sim_text.close()
var_text.close()
print('process finished with ---> ', time.time()-start_time, '
seconds...')
```

Fcoverage.py

```
with open("fault.txt") as fault_text:

    fault_val = list(fault_val.strip() for fault_val in fault_text)
# print(fault_val)
#-----------------------------
fault_len = len(fault_val)

count = [[] for i in range(3)]

for i in fault_val[0]:
    count[0].append(0)
    count[1].append(0)
    count[2].append(0)
# print(count)
for i in range(fault_len):
```

```
    index = 0
    for j in fault_val[i]:
        if j == '0':
            count[0][index] = 1
        elif j == '1':
            count[1][index] = 1
        elif j == 'X':
            count[2][index] = 1
        index += 1
# print(count)
res = []
count_len = len(count[0])
and_count = 0
zero_count = 0
one_count = 0
for i in range(count_len):
    if count[0][i] == 1 and count[1][i] == 1 and count[2][i] == 1:
        res.append('&')
        and_count += 1
    elif count[0][i] == 1 and count[1][i] == 1 and count[2][i] == 0:
        res.append('&')
        and_count += 1
    elif count[0][i] == 1 and count[1][i] == 0 and count[2][i] == 0:
        res.append('0')
        zero_count += 1
    elif count[0][i] == 0 and count[1][i] == 0 and count[2][i] == 1:
        res.append('X')
    elif count[0][i] == 0 and count[1][i] == 1 and count[2][i] == 0:
        res.append('1')
        one_count += 1
    elif count[0][i] == 0 and count[1][i] == 1 and count[2][i] == 1:
        res.append('1')
        one_count += 1
    elif count[0][i] == 1 and count[1][i] == 0 and count[2][i] == 1:
        res.append('0')
        zero_count += 1
# print(res)
    print('number of & counted: ', and_count)
print('number of one counted: ', one_count)
print('number of zero counted: ', zero_count)
total_num_of_fault = int(input('input total number of fault : '))
fault_coverage = (((and_count * 2) + one_count + zero_count) /
total_num_of_fault)*100
print(fault_coverage)
with open("output_1.txt", "w") as output:
    for i in res:
        output.write(i)
```

Mapping.py

```
var = []
with open("pre_var.txt") as pre_var:
    for i in pre_var:
        got = i.split()
        # if i[0] == 'V' and i[1] == ' ':
        temp = len(got)
        if temp > 6:
            # print('`', got[6], '`')
            if got[6] == 'V' and got[7] == '=':
                # print(got[8])
                var.append(i.split()[8])
var_len = len(var)
print(var, 'len: ', var_len)
with open("var.txt", "w") as output:
    for i in range(var_len):
            output.write(var[i])
            output.write(" ")
```

Algorithm 2 (*sol.py*)

```
import time
start_time = time.time()
"""
Open fault table as fault_text
"""
with open("fault.txt") as fault_text:
    fault_val = list(fault_val.strip() for fault_val in fault_text)
#-----------------------------
"""
Open simulation table as sim_text
"""
with open("simulation.txt")  as sim_text:
    sim_val = list(sim_val.strip() for sim_val in sim_text)
#-----------------------------
"""
Open variable table as var_text
"""
with open("var.txt") as var_text:
    var_val = list(map(int, var_text.readline().split()))
    #-----------------------------
"""
storing fault table, simulation table length
and then length of fault table row
```

```
"""
fault_len = len(fault_val)
sim_len = len(sim_val)
fault_row_len = len(fault_val[0])
"""
initial result table and make a temporary vector
for store 'X' in
"""
res = []
temp_res = ''
res_x = ''
for i in range(fault_row_len):
    res_x += 'X'
for fault in fault_val:
    res.append(fault)
res.append(fault_val[0])
for i in range(sim_len):
    sim_val.append(sim_val[i])
"""
initial variables for count zero and one
"""
count_zero = 0
count_one = 0
# Loop through fault table row
for i in range(1, fault_len):
    index = 0
    # for all patterns in fault table rows
    for j in fault_val[i]:
        # calculating result if zero or one found in fault table
        if j == '0' or j == '1':
            if (sim_val[i][var_val[index]] == '1' or
sim_val[i][var_val[index]] == 'h' or sim_val[i][var_val[index]] == 'H')
and (sim_val[i-1][var_val[index]] == '0' or sim_val[i-1][var_val[index]]
== 'l' or sim_val[i-1][var_val[index]] == 'L'):
                temp_res += '1'
            elif (sim_val[i][var_val[index]] == '0' or
sim_val[i][var_val[index]] == 'l' or sim_val[i][var_val[index]] == 'L')
and (sim_val[i-1][var_val[index]] == '1' or sim_val[i-1][var_val[index]]
== 'h' or sim_val[i-1][var_val[index]] == 'H'):
                temp_res += '0'
            else:
                temp_res += 'X'
        else:
            temp_res += 'X'
        index += 1
    # appending calculated result in result variable
    res.append(temp_res)
    temp_res = ''
res_len = len(res)
```

```
for i in range(res_len):
    if not i & 1:
        res[i] = res_x
# output final fault table as fault_output.txt
with open("fault_output.txt", "w") as output:
    for i in res:
        output.write(i)
        output.write('\n')
output.close()
# output final simulation table as simulation_output.txt
with open("simulation_output.txt", "w") as output:
    for i in sim_val:
        output.write(i)
        output.write('\n')
fault_text.close()
sim_text.close()
var_text.close()
output.close()
print('process finished with ---> ', time.time()-start_time, '
seconds...')
```