

**Probabilistic approaches for verification of unlikely inserted errors in
Hardware Description Languages**

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in
the Graduate School of The Ohio State University

By

Venkata Sai Manoj Pasupuleti, B.E.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2016

Master's Examination Committee:

Prof. Steven Bibyk, Advisor

Prof. Wladimiro Villarroel

Copyright by
Venkata Sai Manoj Pasupuleti
2016

Abstract

With technology becoming more advanced each day and CMOS scaling being done proportionately to accommodate the size, speed and advancement in technology, more transistors are packed on to the same chip, thereby making fault tolerance, reliability and error detection an increasingly important design specification for processors. Unfortunately, for all practical purposes it is not possible to test the functionality of many digital integrated circuits chips exhaustively, since they can involve billions of components. Thus, there are no guarantees that all possible errors in the chip design and manufacture can be found. Therefore, random test vectors are usually generated which would help to test the functionality for most parts of the IC design. This leads to hardware being vulnerable to the insertion of malicious errors, similar to the problem of software viruses embedded into large software programs. Hence, in this thesis, prior and posteriori probabilities are developed to explore code in Hardware Description Language (HDL) designs of digital entities that are most vulnerable to errors.

This thesis develops a probabilistic approach which is based on the assumption that lines of code with very low probabilities of execution are the best place to insert malicious errors, such as a Hardware Trojan. To explore this idea, two versions of 16bit ALU's, one a Golden ALU (error Free) and second a Modified ALU (which has errors inserted into it) are used.

From the designed ALU entities, the probability of execution of each ALU section is determined to be as follows

$P(\text{AND-XOR}) < P(\text{AND-OR}) < P(\text{SUM-XOR}) < P(\text{SUM-OR}) < P(\text{No error loop})$. And the vulnerabilities associated with each of these loops was found to be

$(\text{AND-XOR}) > (\text{AND-OR}) > (\text{SUM-XOR}) > (\text{SUM-OR}) > (\text{No-Error})$.

By design, the exhaustive testing probabilities are derived as priori probabilities in this thesis. However, these results have been verified against posteriori probabilities for discrete finite test cases. The results showed that loops with low probability (e.g., SUM-XOR loop, has probabilities very close to 1.3×10^{-4}) are consistent in priori and posteriori values, while high probability loops (e.g., No-error, has probabilities around 0.99) are not. The recommendation section in this thesis outlines the next step to improve the result consistencies. These test vectors used on DUT had a Fault detection accuracy of 0.8, with scope of improvement.

Dedication
Dedicated to Amma, Nanna, Akka and Bava

Acknowledgments

I am hugely indebted to my advisor, Prof. Steven Bibyk, for providing me the opportunity to work under him. His constant support, precious advice and generous guidance motivated me immensely in my research at The Ohio State University.

I am also grateful to Prof Wladimiro Villarroel for serving on my thesis committee and for his questions and suggestions. I would also like to thank Adam Kimura for his valuable suggestions and guidance throughout my project.

I would like to thank Altera Inc., for their Model Sim Software which had helped me throughout this project.

Finally, I would like to thank my sister and brother-in-law for their unconditional support and guidance during my Master's Program. I am immensely grateful to my parents for their love, affection and confidence in me.

Vita

May, 2011..... Bachelors of Engineering.
Electrical and Electronics Engineering,
Manipal Institute of Technology, India.

July 2011- July 2014..... Senior Engineer,
Larsen & Tourbo,

August, 2014 - presentGraduate Student,
Electrical and Computer Engineering,
The Ohio State University

August, 2015 - presentGraduate Teaching Associate,
Geography Department,
The Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

Abstract.....	ii
Dedication	iv
Acknowledgments.....	v
Vita.....	vi
Table of Contents.....	vii
List of Tables	x
List of Figures	xi
Chapter-1 Introduction	1
1.1 Problem Statement.....	1
1.2 Objective of Thesis	2
1.3 Requirement Specification	3
1.4 Organization of Thesis	3
Chapter -2 Literature Survey.....	4
2.1 Trojan Classification	4
2.2 Trojan Detection Methodologies.....	8
2.2.1 Trojan Detection Using Side-Channel Power-Based Analysis	8

2.2.3 Trojan Detection Using Side-Channel Timing-Based Analysis	13
2.3 Trojan Activation Methods.....	15
2.4 Design for Hardware Trust.....	19
2.5 Technical Background	20
2.5.1 Need For RTL Coding	21
2.5.2 Behavioral Modeling.....	22
2.5.3 Probability of any Line of Code	23
2.5.4 Test Bench.....	26
2.6 Summary	29
Chapter 3 Mathematical Model and Tech Bench Setup	31
3.1 Mathematical Model	31
3.2 Test Setup	33
3.3 Test Vector Generation	35
Chapter-4 Experimental Results and Simulation.....	36
4.1 Modified ALU probabilities of execution	36
4.2 DUT Simulation.....	44
Chapter-5 Conclusion & Future Work.....	53
5.1 Advantages and Conclusion	55
5.2 Future Scope of Work.....	55
References	57

Appendix A: Golden ALU	60
Appendix B: Modified ALU	62
Appendix C: Test Bench.....	66
Appendix D: Test Vectors Generation Code	71

List of Tables

Table 1	Summary of possible values of num1 and num2.	25
Table 2.	Summary of Key Trojan detection methods	27
Table 3	Summary of ALUCOUNT and Corresponding Operations	32
Table 4.	Summary of Probabilities of various loops	39
Table 5	Summary of number of events in favor of of execution of each loop Sample Space : 400,000	40
Table 6	Summary of number of events in favor of of execution of each loop Sample Space : 4,000,000	40
Table 7	Summary of number of events in favor of of execution of each loop Sample Space : 20,000,000	40
Table 8	Exhaustive Testing Probabilities(Priori) and Posteriori	40
Table 9	Exhaustive Testing Probabilities(Priori) and Posteriori.....	40

List of Figures

Figure 2.1 Detailed Taxonomy showing physical, activation and action characteristics	5
Figure 2.2 Three components of a hardware Trojan Horse (HTH).....	7
Figure 2.3 Design flow of HTH	7
Figure 2.4 Simple Power Analysis of Trojan Circuits.....	10
Figure 2.5 Current Charge Integration Method.....	11
Figure 2.6 Path delay measurement architecture using a Shadow Register	13
Figure 2.7 Ring Oscillator Temperature Monitor	14
Figure 2.8 Illustration of the concept of Region and Radius in a circuit.....	17
Figure 2.9 Pseudo Code to Isolate Gate having Trojan activity	18
Figure 2.10 Voltage Inversion Scheme for Trojan Detection	20
Figure 2.11 Analog Waveform.....	21
Figure 2.12 Digital Waveform	22
Figure 2.13 Full Adder behavioral Code	23
Figure 2.14 Comparator Code.....	24
Figure 2.15 Architecture of Test Bench	24
Figure 2.16 Pseudo code for reading Input Test Vectors	24
Figure 3.1 ALU Overview	31
Figure 3.2 Pseudo Code for Golden ALU	35
Figure 3.3 Pseudo Code for Modified ALU	36

Figure 4.1 Pseudo Code for AND-XOR loop in Modified ALU	38
Figure 4.2 Pseudo Code for AND-OR loop in Modified ALU	39
Figure 4.3 Pseudo Code for AND-XOR loop in Modified ALU	40
Figure 4.4 Pseudo Code for SUM-OR loop in Modified ALU	41
Figure 4.5 Pseudo Code for NO-Error loop in Modified ALU	43
Figure 4.6 Simulation results for number of events in favor of execution of each loop	
Sample Space: 400,000	45
Figure 4.7 Simulation results for number of events in favor of execution of each loop	
Sample Space: 4,000,000.....	45
Figure 4.8 Simulation results for number of events in favor of execution of each loop	
Sample Space: 20,000,000.....	45
Figure 5.1 Flow diagram for Error detection using probabilistic approach.....	49

Chapter-1 Introduction

1.1 Problem Statement

Economic and market conditions are forcing most of the manufacturers to outsource their IC fabrication to cheaper fabrication facilities abroad [3]. With the globalization of chip manufacturing industries IC's are becoming more and more vulnerable to malicious activities such as inclusion of Trojans. These Trojans are designed to be hard to detect by normal testing procedures. This has raised serious concerns regarding possible threats to military systems, financial infrastructures, transportation, security and even household appliances. An adversary can introduce a Trojan to either disable or destroy a system at some point in the future or the Trojan might be used to leak confidential information secretly to an adversary.

Unfortunately the detection of these Trojans is difficult for several reasons including the following: The nanometer IC feature sizes and system complexity made detection through physical inspection and destructive reverse engineering difficult and costly. Secondly, Trojan circuits are by design activated under very specific conditions, which makes it difficult to fully activate them using random and functional stimuli. Moreover the existing automatic test pattern generation (ATPG) methods used in manufacturing tests for detecting defects do so by Operating on the net list of the Trojan free circuit specifications. Therefore existing ATPG algorithms cannot target Trojan activation directly. [4]

Although there has been a significant amount of work done on testing circuits for Trojan detection and prevention, no systematic approach has been developed to assess a circuit's susceptibility to Trojan insertions. Adversaries usually target sections in the circuit with low controllability and observability and implant stealthy Trojans. This necessitates a thorough circuit analysis to identify potential Trojan locations. Furthermore, there has been little or no work done on creating a metric to determine the difficulty of detecting a Trojan in a circuit. Therefore a comprehensive vulnerability analysis at the behavioral level is required to quantify the difficulty of activation of each circuit portion.

1.2 Objective of Thesis

The motivation behind this thesis is to develop a Probabilistic approach for error detection, which overcomes the drawbacks of existing methods. This thesis develops a mathematical model to determine the difficulty of finding an insertion error in any line/loop of code, by using a probabilistic approach. With the help of this mathematical model one can point to the locations where a stealthy error can be introduced. In order to ensure the code is error free extra random vectors are generated to validate the lines of code specified by the model

1.3 Requirement Specification

This thesis aims to address the following requirements. It develops a mathematical model which can be used to detect the lines of code which are vulnerable to errors. This mathematical model is then tested with the help of few Simulations to check if the results are in line with our expectations. In these experiments, a 16-bit Golden ALU (Trojan free) and a Modified 16-bit ALU (which has Trojans) are used to verify if an Insertion-error could be detected in the lines of code which have the least probability.

1.4 Organization of Thesis

This thesis is organized as follows.

- 1) An insight on established research work done on Trojan detection and prevention is covered in Chapter 2. At the end of this chapter, a brief technical background on the need for RTL Coding, behavior modeling and how to calculate probability of a given line of code, which will provide a context for the subject research.
- 2) Following this in chapter 3, a mathematical model is derived to associate the vulnerability of a line of code to its Probability, and a Test Bench has been setup to test this model.
- 3) In chapter 4, Experimental results and Simulations are discussed.
- 4) Finally the thesis is concluded in chapter 5, with a brief summary and recommendations for future scope.

Chapter -2 Literature Survey

2.1 Trojan Classification

This literature survey presents the current state of knowledge on Trojan detection schemes and designing methodologies for improving Trojan detection techniques. At the end of this chapter various detection methods are compared and analyzed based on their effectiveness in detecting various Trojans.

Wang, Tehranipoor and Plusquellic [1] have developed the first detailed taxonomy for the hardware Trojans. They have decomposed the taxonomy into three main categories (see in figure 2.1) according to their physical activation and action characteristics. They further divided Trojans into six types, of which four are physical, one activation and one action attribute. It is also possible for Trojans to be of hybrid in nature e.g., having more than one activation characteristic. This taxonomy helps us in understanding elemental characteristics of Trojans and thereby helps us in detecting these Trojans [1] [13].

The physical characteristics category describes the various hardware manifestations of Trojans. The type category divides Trojans into functional and parametric classes.

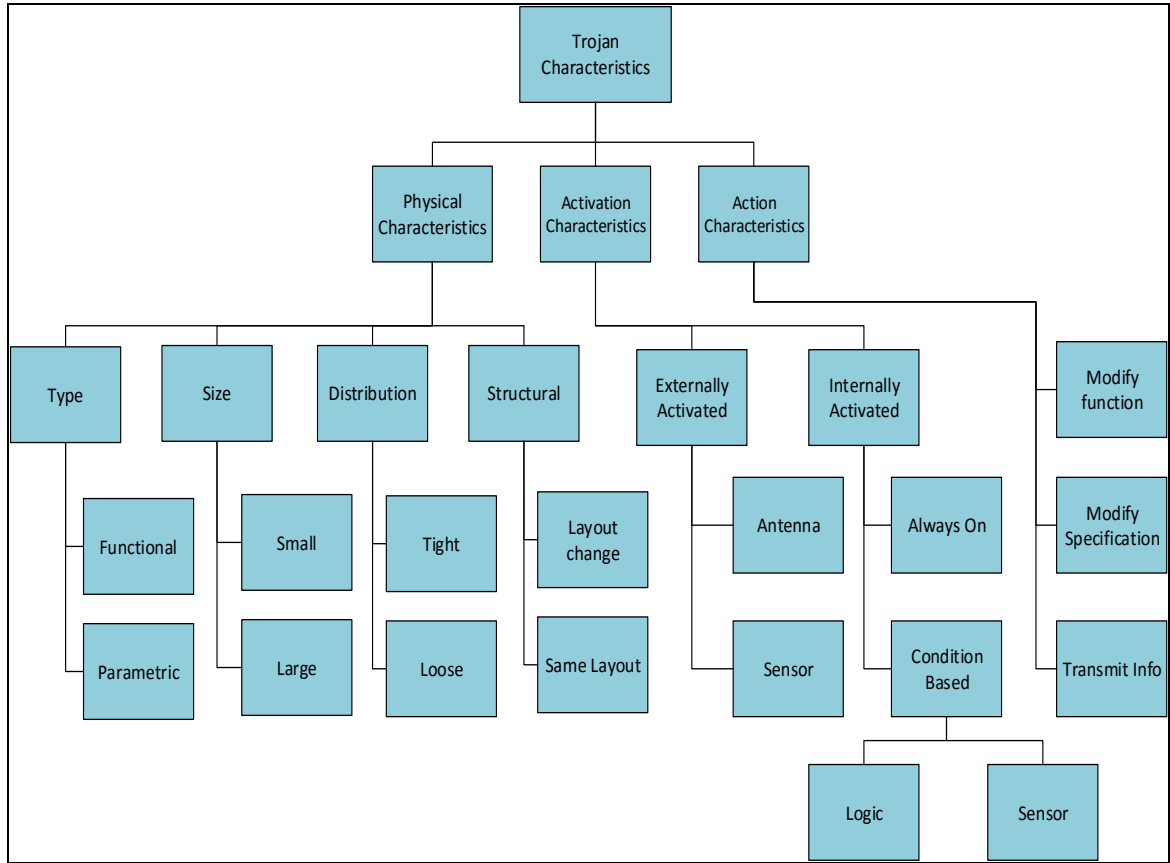


Figure 2.1: Detailed Taxonomy showing physical, activation and action characteristics [1]

The functional class includes Trojans that are realized either by addition, deletion of transistors or gates, whereas parametric class refers to Trojans that are realized through modification of existing wires and logic e.g., thinning of a wire, weakening of a transistor etc. The size category counts for the number of transistors that are either added or deleted from the circuit. The distribution category describes the location of the Trojans on the chip. It can be further classified as tight distribution wherein all the Trojan components are close to each other in the layout, whereas loose distribution describes a Trojan whose components are distributed across the layout of the chip. The structure category refers to a scenario in which by addition of Trojan there is a change in the dimension of the chip. This

change in physical layout can easily affect the delay and power characteristics of the chip [1] [13].

Action characteristics refer to the criteria that cause the Trojan to become active and carry out its disruptive function. Trojan activation characteristics can be broadly divided into Externally-activated and Internally-activated. In externally-activated category the Trojan gets activated because of external sources like antenna. And internally-activated can be further classified as always-on and condition-based. Always-on Trojans are implemented by modifying the geometrics of the chip such that certain nodes or paths have a higher susceptibility for failure. The condition based Trojans are usually inactive until a specific condition is met. The activation conditions can be an output of a sensor or environmental conditions or internal logic state [1] [13].

Action Characteristics identify the types of disruptive behavior introduced by the Trojan. As shown in the figure 2.1, this category can be further classified into three types- modify function, modify specification and transmit information. The modify function class refers to the type of Trojans that change the chip's function by either adding new logic or by removing or by passing existing logic. The modify specification class refers to the Trojans that focus on changing the chip's properties like delay, power. Finally the transmit info class refers to Trojans that transmit key information to an adversary [1] [13].

According to Alkabani and Koushanfar [2] the components needed for a Hardware Trojan Horse (HTH) is divided in to three categories: trigger, storage and driver. A trigger incites the planned HTH. Once the trigger is initiated, the action to be taken place can be

stored in the memory or a sequential circuit. Finally, the driver implements the action to be performed after the Trojan has been triggered.

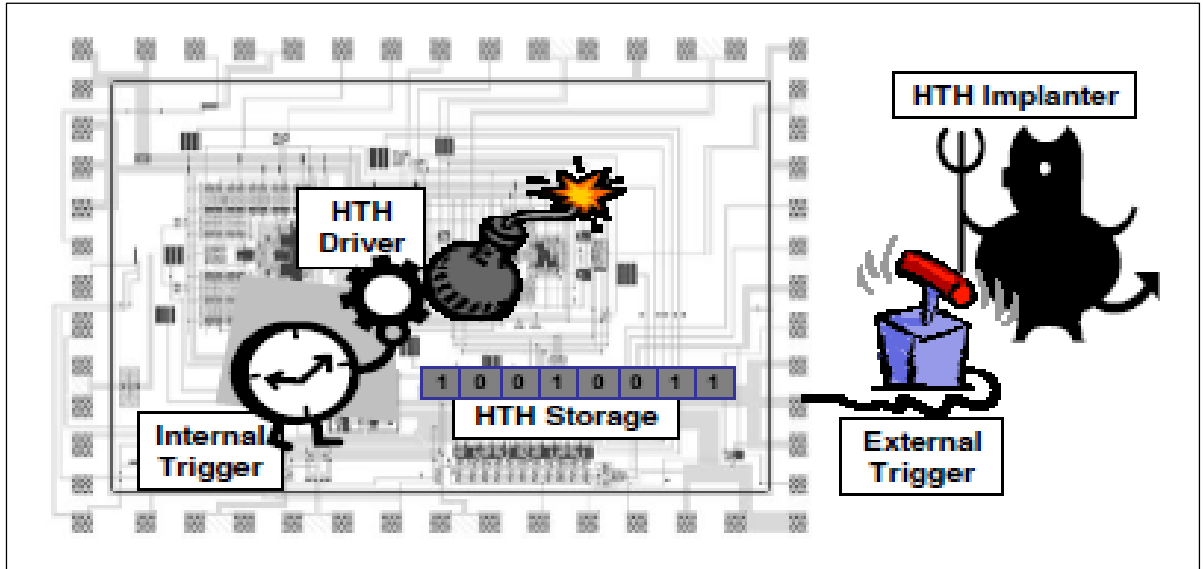


Figure 2.2: Three components of a hardware Trojan Horse (HTH) [2]

Alkabani and Koushanfar presented a systematic approach to insertion of hardware Trojan into the IC. Figure 2.3 shows an abstracted view of the design process. The Trojan designer composes a high level description of the computation model of the circuit that a Finite State Machine (FSM) can represent. HTH can be inserted into the FSM by adding new states.

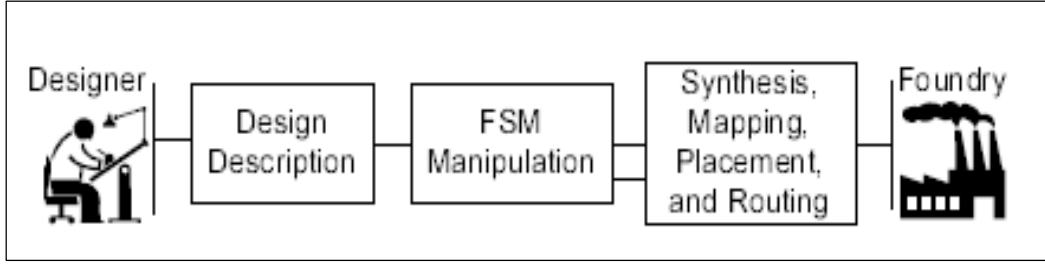


Figure 2.3: Design flow of HTH [2]

2.2 Trojan Detection Methodologies

Several Trojan methodologies have been developed over the past few years. They can be broadly classified as either side-channel analysis or Trojan activation [13]. According to M. Tehranipoor, F. Koushanfaride [13] Side-channel analysis of any circuit looks for change in performance characteristics like timing and power. Trojans typically change circuit characteristics thereby changing power or delay characteristics of wires and gates. Power based side-channel signals help us understand the internal structure and activities within the IC thereby enabling detection of Trojans without fully activating them. Timing-based side-channel analysis can detect a Trojan's presence if the chip is tested using delay test that are sensitive to small changes in the circuit delay.

2.2.1 Trojan Detection Using Side-Channel Power-Based Analysis

Trojan detection theory:

Consider an IC I , that executes a calculation C and consumes power M for this computation. The power trace obtained in this measurement, $r(t; I; C; M)$ can be modeled as consisting of four components, the mean power consumption $p(t; C)$, the process noise $n_p(t; I; C)$, the measurement noise $n_m(t; M)$ and the Trojan power leakage noise $\tau(t; I; C)$. The total power trace is given by,

$$r_G(t; I; C; M) = p(t; C) + n_p(t; I; C) + n_m(t; M) \quad (1)$$

The total power trace in an trojan IC is given by,

$$r_T(t; I; C; M) = p(t; C) + n_p(t; I; C) + n_m(t; M) + \tau(t; I; C) \quad (2)$$

The measurement noise $n_m(t; M)$ is a random noise and would vary with each measurement.

This can be eliminated by averaging over a large number of measurements and therefore equation (1) and (2) can be simplified into

$$r_G(t; I; C) = p(t; C) + n_p(t; I; C) \quad (3)$$

$$r_T(t; I; C) = p(t; C) + n_p(t; I; C) + \tau(t; I; C) \quad (4)$$

The process noise $n_p(t; I; C)$ can be eliminated by having access to multiple genuine ICs and hence the mean power consumption $p(t; C)$ can be calculated, that occurs during the calculation C. Since the mean power consumption is common among the power traces obtained from both the genuine and the Trojan ICs, it can also be subtracted from all the power traces and the Trojan power consumption can be calculated as below.

$$r_G(t; I; C) = n_p(t; I; C) \quad (5)$$

$$r_T(t; I; C) = n_p(t; I; C) + \tau(t; I; C) \quad (6)$$

1. Simple Power Analysis

Agrawal et al. [3] were the first to use side-channel based finger printing methodology for detecting Trojan contribution to circuit power consumptions. This approach does not require any changes to current processes and practices regarding the design and fabrication ICs and most importantly this would not require trusted fabrications to obtain the power

signature of Trojan-free (i.e. Golden) ICs, random patterns were applied and power measurement if performed. The power consumption of the circuit which was measured consists of measurement noise, process variation and Trojan contribution. These patterns are applied on a selected few ICs that are randomly selected from a family of ICs which are reverse-engineered to ensure they are Trojan free. Once the reference signature is obtained, the same random patterns are applied to other ICs under authentication (IUA). If the IUAs power signature is different form the golden IC, it is considered suspicious and might contain a Trojan. If the Trojan size is comparable to the size of the circuit then its impact on the circuit transient current will be significant and could be measured easily. However, if the Trojan size is very small when compared to the size of the circuit its effect is easily masked because of process variations [3] [13].

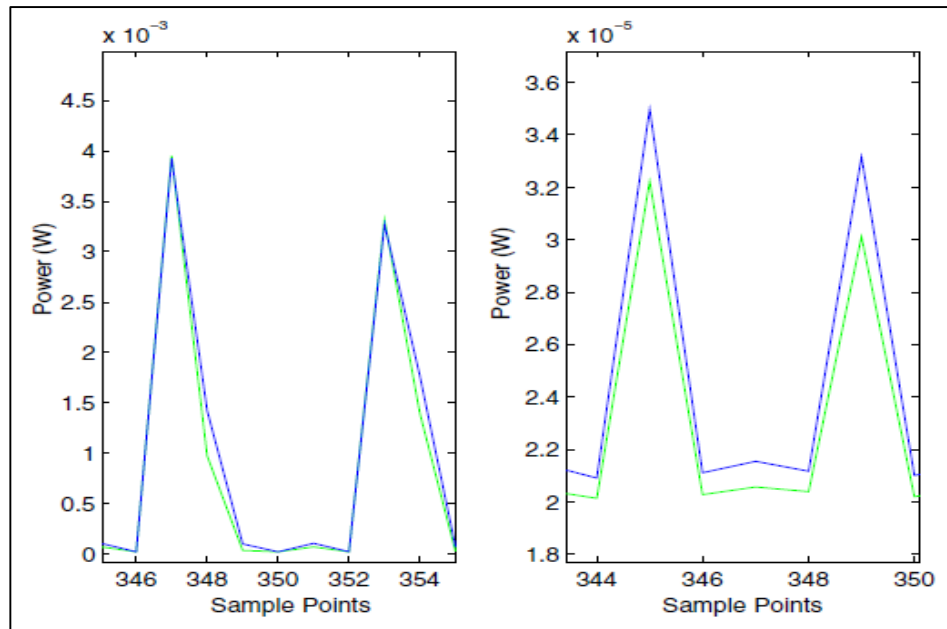


Figure 2.4: Simple Power Analysis of Trojan Circuits

Genuine (green/gray) and Trojan (blue/black) AES signals at 100MHz (left) and 500KHz(right) [3]

$$P = \left(\frac{1}{2} \cdot C \cdot V_{DD}^2 + Q_{se} \cdot V_{DD} \right) \cdot f \cdot N + I_{leak} \cdot V_{DD}$$

$$\text{Dynamic Power} = \left(\frac{1}{2} \cdot C \cdot V_{DD}^2 + Q_{se} \cdot V_{DD} \right) \cdot f \cdot N ;$$

$$\text{Leakage Power} = I_{leak} \cdot V_{DD}$$

2. Current Integration Method

According to Wang et al. [4] most Trojans inserted into the chip require power supply and ground to operate. The authors developed a multi supply transient current integration methodology to detect a hardware Trojan. A Trojan when inserted on a chip would consume power. However the Trojans contribution to the total power consumption would depend on its size and type. Fully activation of a Trojan using structural and functional patterns would be extremely challenging and prohibitively expensive. Hence partial activation of Trojans would be an effective way for Trojan detection using transient current based side-channel analysis methods similar to current integration method [4] [13].

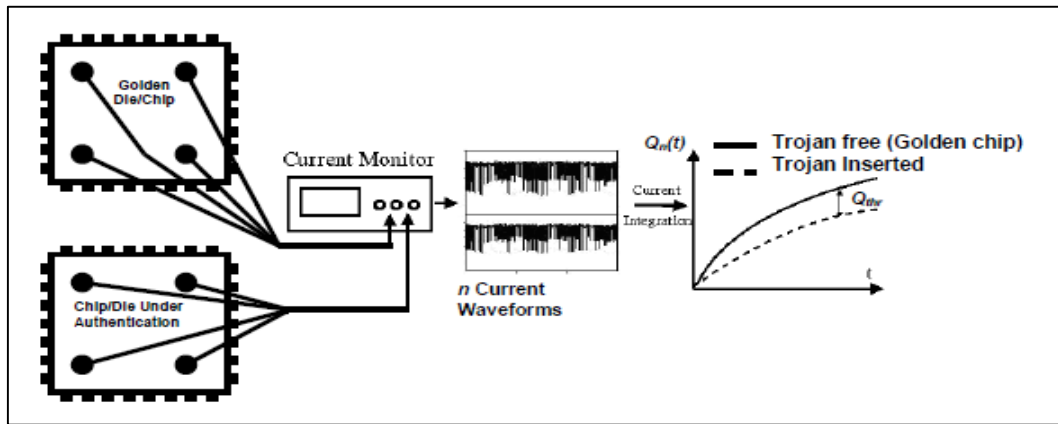


Figure 2.5: Current Charge Integration Method [4]

The amount of current which a Trojan draws can be so small that it can be submerged into an envelope of noise and process variation effects and thus not be detected by conventional methods. However Trojan detection capabilities are greatly enhanced by measuring currents locally and from multiple power ports/pads. Figure 4 shows our current (charge) integration methodology presented by Wang et al. for detecting hardware Trojans. The die includes four power ports. The golden die can be selected after subjecting it to exhaustive number of test cases. If the same results are obtained for all the cases, the chip can assumed as Trojan free. After identifying the golden chips, the worst case will be obtained for a particular test vector. The worst case is considered because of the process variations in any of the genuine IC's [4] [13].

Now the Trojan infected IC, is subjected to n number of current waveform patterns as shown in the figure, the charge variations for all the current waveforms with time is obtained after applying the test patterns. $Q_n(t)$ denotes the charge accumulation after applying n test patterns. Q_{thr} is the threshold charge to detect a Trojan. When applying the test patterns, the increase in charge is continuously compared with the worst case charge calculated for the golden chips. Once the difference between the two curves, ΔQ is greater than Q_{thr} , a Trojan is detected. If $I_{trojan_free}(t)$ and $I_{trojan_inserted}(t)$ denote the instantaneous supply current drawn by Trojan free and Trojan inserted circuit at time t respectively, then the integrated current at time t for Trojan free and Trojan inserted ($Q_{trojan_free}(t)$, $Q_{trojan_inserted}(t)$) can be expressed by equations (1) and (2).

$$dq = I \cdot dt$$

$$Q_{trojan_free}(t) = \int I_{trojanfree}(t) dt \quad (7)$$

$$Q_{\text{trojan_inserted}}(t) = \int I_{\text{trojaninserted}}(t) dt = \int (I_{\text{trojanfree}}(t) + I_{\text{trojan}}) dt \quad (8)$$

Where $I_{\text{trojan}}(t)$ denotes the current drawn by Trojan. The difference between the two currents $I_{\text{trojan_free}}(t)$ and $I_{\text{trojan_inserted}}(t)$ is equal to the additional current drawn by the Trojan gates and the changes in the circuit current due to process variations [4] [13].

2.2.3 Trojan Detection Using Side-Channel Timing-Based Analysis

Li and Lach [5] proposed a new method for IC authentication and HTH detection based on delay characteristics. This delay based model uses physical unclonable function (PUF) for hardware Trojan detection. This method uses a sweeping-clock-delay measurement technique to measure selected register to register path delays. This method was originally developed to characterize the process variations. Trojans can be detected when one or more path delay measurements are beyond the threshold limit. The figure 2.6 shows the path delay measurement architecture [5] [13].

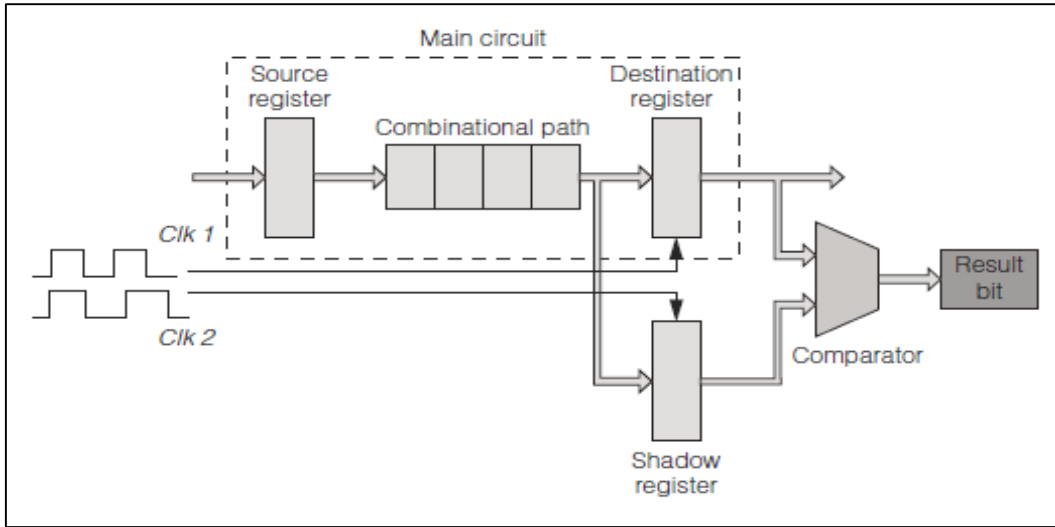


Figure 2.6: Path delay measurement architecture using a Shadow Register [5]

As shown in the Figure 2.6, the main circuit is a one register-to-register combinational path delay that is to be characterized and the registers on this path are triggered by the main system clock (CLK1). The components outside the box are part of the testing circuitry. The shadow register takes the same input as the destination register, but is triggered by the shadow clock (CLK2), which runs at the same frequency as main system clock, but with a controlled phase offset. The results latched by the destination register and the shadow register is compared during every clock period until a discrepancy is identified. If the comparison result is unequal, the path delay is characterized with the precision of the skew step size [5] [13].

In the technique described above, the functionality of the main circuit is not affected by the delay characterization testing. This main feature enables to test at-speed without adding extra complexity to the testing process [5] [13].

An issue arises when nonfunctional characteristics like delay and power are used for IC authentication, is that they are dependent on temperature and voltage variations. To overcome this problem, this method employs an on die temperature monitor to overcome the problem of temperature affecting the path delay. As shown in the figure, the monitor uses a ring oscillator as the clock input of a counter to measure operating temperature.

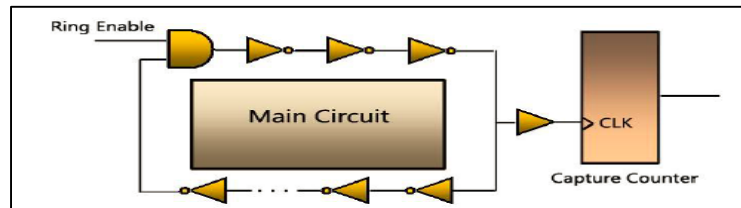


Figure 2.7: Ring Oscillator Temperature Monitor [5]

Because the oscillator, is embedded within the main circuitry, and its switching frequency is temperature dependent, the authenticator can calculate the effective response from the reported temperature and delay signature [5] [13].

Jin and Makris [6] proposed a new fingerprint generating method using the path delay information of the entire chip. A chip has many delay paths and each representing one part of the whole circuit. The timing features in this model can generate a series of path delay fingerprints. The entire testing process includes three steps

1. **Path delay gathering of nominal chips** – In this step, many chips are selected from an IC design. High coverage input patterns are run on the sample chips and high dimension path delay information is collected. Then these chips are checked via reverse engineering to ensure they are genuine circuits [6] [13].
2. **Fingerprint generation** – According to path delays, a series of delay fingerprints are generated and mapped to a lower dimension space [6] [13].
3. **Trojan Detection** – All the other chips are checked under the same test patterns and their delay information is reduced to a low dimension when compared to the delay fingerprints [6] [13].

2.3 Trojan Activation Methods

According to M.Tehranipoor, F.Koushanfar [13] Trojan activation strategies can accelerate Trojan detection process. If a portion of the Trojan circuitry is activated, the Trojan circuit will consume more dynamic power, which will further help in differentiating the power traces of Trojan inserted and Trojan free circuits. The Trojan activation schemes can be classified as Region free Trojan activation and Region aware Trojan activation.

Region free Trojan activation:

These methods do not rely on the region but depend on accidental or systematic activation of Trojans. According to Jha & Jha [7], a randomization based probabilistic approach can be used to Trojans. They showed that it is possible to construct a unique probabilistic signature of a circuit on the basis of a specific probability for patterns applied to its inputs. They apply input patterns based on the specific probability to the IUA and compare its output with the original circuit. If there are differences in the outputs, a Trojan is present [7] [13].

Region aware Trojan activation:

Banga and Hsiao [8] developed a two stage test generation technique that targets magnifying the difference between IUA and the genuine design power waveforms. In the first stage (circuit partitioning), a region aware pattern helps to identify the potential Trojan insertion regions. To detect a Trojan, the activity in a particular region is magnified and for the rest of the circuit is simultaneously minimized. In the next stage (activity magnification), new test patterns are applied to magnify the differences between original and the Trojan inserted circuits [8] [13].

In the figure given below, the circuit is partitioned into five regions. Its radius defines the extent of a region. For a single gate region, the radius is zero, the immediate fan in and fan out gates connected to it will have a radius of one (G1, G2, G3, G4, FF1, G6 and G7) and so on.

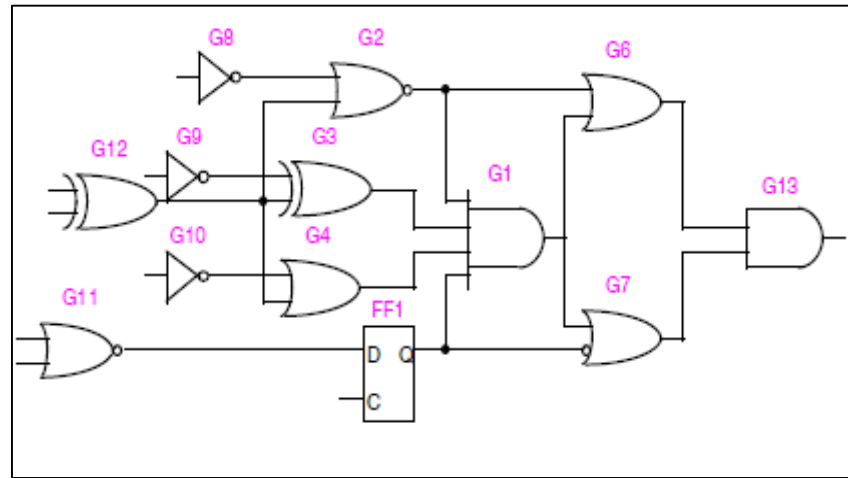
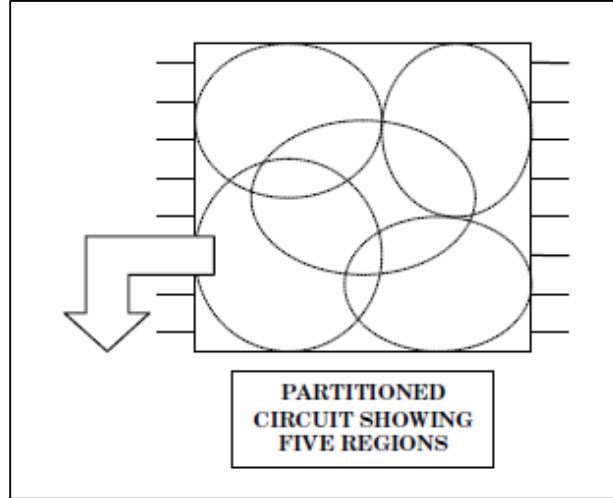


Figure 2.8: Illustration of the concept of Region and Radius in a circuit [8]

$$F = \max (\text{In-Region activity} - \text{Out-Region activity})$$

If the behavior of a Trojan is perceivable only if the difference in the activity of the Trojan infected chip and the genuine chip is above the process variation.

Banga and Hsiao [9] discussed magnifying Trojan contributions by minimizing a circuit activity. Different portions of the circuit can be explored by changing input vectors to localize a Trojan. At the same time, each gate is equipped with two counters,

Trojan count and Non-Trojan count. With each vector, if the number of transitions at the gates' output exceeds a specific threshold, its Trojan count would increase and vice versa.

```

Require: GenuineCkt, TrojanCkt, InputVector

Ensure: Plot of gate weights corresponding to their probability of Trojan association

PowerDifferentialThreshold<=5.0

TrojanCount<=0

NonTrojanCount<=0

ToggledGateList<=Simulate(GenuineCkt,InputVector)

PowerNumbers(GenuineCkt)=PowerSimulate(GenuineCkt,InputVector)

PowerNumbers(TrojanCkt)=PowerSimulate(TrojanCkt,(InputVector))

for all( $V_i, V_{i+1}$ ) $\in$ InputVector d
  %PowerDifferential<=(abs
  ( $\frac{PowerNumbers(TrojanCkt,(V_i,V_{i+1}))-PowerNumbers(GenuineCkt,(V_i,V_{i+1}))}{PowerNumbers(GenuineCkt,(V_i,V_{i+1}))}$ )) * 100

  If %PowerDifferential>PowerDifferentialThreshold then

    IncrementWeight(TrojanCount,ToggledGateList( $V_i, V_{i+1}$ ))

  else

    IncrementWeight(NonTrojanCount,ToggledGateList( $V_i, V_{i+1}$ ))

  end if

BuildPowerProfilePlots(PowerNumbers(GenuineCkt),PowerNumbers(TrojanCkt))

For all  $g_i \in$  GenuineCkt do

  GateWeight =  $\frac{TrojanCount(g_i)}{NonTrojanCount(g_i)}$ 

end for

```

Figure 2.9: Pseudo Code to Isolate Gate having Trojan activity [9]

A high gate weight ratio means that the gate has been considerably been impacted by a Trojan.

2.4 Design for Hardware Trust

According to M.Tehranipoor and F.Kaushanfar [13], the current design practices do not support effective side channel signal analysis or pattern generation for Trojan detection. Hence the methods proposed by hardware security and trust community to improve Trojan detection and isolation by changing the design flow should be given a closer look. These methods are called as Design for Hardware Trust [12] [13].

Banga and Hsiao [12], proposed an inverted voltage scheme to magnify the Trojan activity. Because the Trojan is assumed to be active only under certain rare conditions, IC inputs can be changed so that rare combinations are created to activate the Trojan.

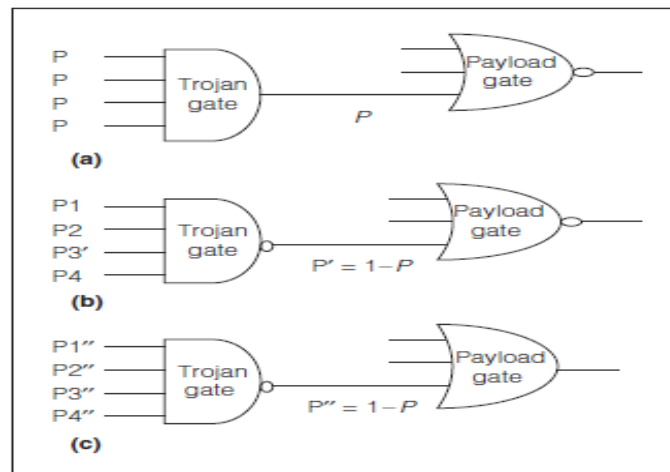


Figure 2.10: (a) Trojan logic under normal Voltage supply (b) Only the Trojan gate is effected by voltage inversion (c) both Trojan and Payload are affected by voltage Supply [12]

This approach is based on two principles,

1. Gate Logic Inversion – This creates a frequent triggering scenario for the Trojan gate [12] [13].
2. Sustained Vector Simulation – This decreases overall circuit activity and magnifies the extra toggles created by the Trojan [12] [13].

For a random gate g , if the logical value L ($L \in \{0, 1\}$) appears more frequently than the other logical value \bar{L} under a test vector set T , then L is called as the majority value and value \bar{L} is called as the minority value for g under T . The majority value for an AND gate is 0 and minority value is 1, similarly majority value for an OR gate is 1 and minority value is 0 [12] [13].

For example, in the figure above, for the AND gate with four inputs, the rare condition would be when all the four inputs are 1 (probability is 1/16). The goal of this process is to remove this rare condition. This can be achieved by reversing the gates' power supply voltage (V_{DD}) and ground (GND). Thus the AND gate is changed into a NAND gate and 1 at the output of the NAND gate is never a rare value (as the probability is now 15/16).

$$P(\text{trigger}) = \prod_{i=0}^{i=n} P[i]_{NC} \quad (9)$$

2.5 Technical Background

This section provides a technical background by giving a brief overview on the need for RTL coding, how behavioral modeling done and finally how the probability of a given line

of code is calculated. After establishing these basics we can, we can dive into deriving a mathematical model for this thesis.

2.5.1 Need For RTL Coding

RTL stands for Register Transfer level. Typically a IC which we manufacture has two sections

1. Analog: This section of the IC interacts which all sections of the outer world and uses all voltage levels.

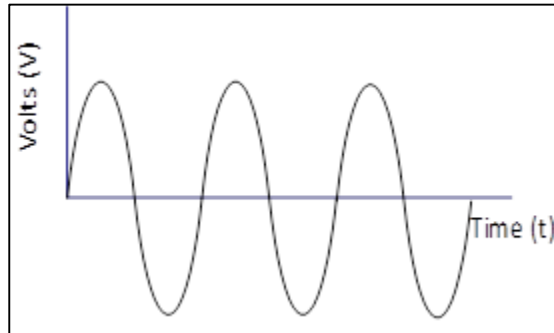


Figure 2.11: Analog Waveform

2. Digital: This section forms the core part of IC and deals with only two voltage levels 0 and 1. All data transfer between various sections, data processing and all computations are accomplished using digital design.

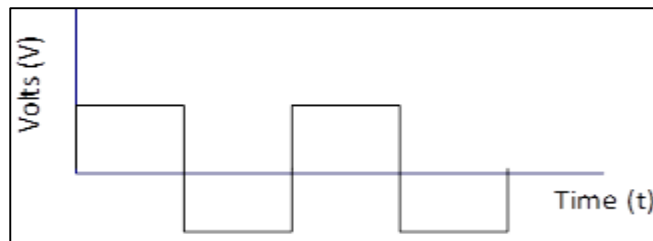


Figure 2.12: Digital Waveform

RTL description is a lower level abstraction than behavioral model but higher level abstraction than netlist description of a circuit. The RTL description of a circuit can be written using any of the HDL (Hardware Descriptive Language)

Like VHDL or Verilog. In RTL description, the circuit is described in terms of registers and data is transferred between them using logical operations and hence the name Register Transfer Level.

RTL is the first real dig at the detailed circuit design. All the subsequent steps in the design process would depend on the quality of the RTL design. Better the quality of RTL, the quicker the design can be sent into the market.

I have used VHDL for RTL design in my thesis. VHDL was first published as IEEE standard in 1987 and since then revised versions have been launched in 1994, 2000, 2002 and 2009.

2.5.2 Behavioral Modeling

Behavioral Modeling is the highest form of abstraction possible with the HDL. It is mainly used for system analysis, simulation and partition stage. Behavioral model cannot be synthesized into logic gates automatically like in RTL.

Behavioral modeling of any circuit mainly has processes, sequential statements. Processes run code sequentially. Subprograms is another behavioral construct which allows us to reuse the code. Packages are another important component in VHDL they contain subprograms, constants definitions to be used through one or more design items.

```

Entity full_adder is
(A,B,Cin: in Bit;
 Cout,Sum:out Bit
);
End full_adder;
Architecture behavioral of full_adder is
Begin
Process (A,B,Cin)
Sum<= A xor B xor Cin;
Cout<= (A and B) or (B and C) or (C and A);
End Process;
End behavioral;

```

Figure 2.13: Full Adder behavioral Code

2.5.3 Probability of any Line of Code

Probability is a branch of mathematics which deals with the calculation of the likelihood of occurrence of an event. It is the ratio of number cases favorable towards the occurrence of an event to the total number of all possible cases.

$$P(X) = \frac{\text{Number of favorable outcomes}}{\text{Total Number of outcomes}} \quad (10)$$

where X is a random event.

The sum of the probabilities all possible outcomes is one.

$$\sum_{k=1}^N P(X_k) = 1 \quad (11)$$

```

entity compare is
port(  num1 :  in std_logic_vector(3 downto 0); --input 1
      num2 :  in std_logic_vector(3 downto 0); --input 2
      first:  inout std_logic; -- indicates first number is larger
      second: inout std_logic; --indicates second number is larger
      equal : inout std_logic);
end compare;
architecture Behavioral of compare is
begin
process(num1,num2)
begin
if (num1 > num2) then
    first <= '1'; second <= '0'; equal<='0';
    elsif ((num2 > num1) then
        first <= '0'; second <= '1'; equal<='0';
    else
        first <= '0'; second <= '0';equal<='1';
    End if;
End process;
End behavioral;

```

Figure 2.14: Comparator Code

For the code given above we have two input signals *num1* and *num2* and three output signals *first*, *second* and *equal*.

Signal	Possible Outcomes	No.of Outcomes
num1	000,001,010,011	4
num2	000,001,010,011	4
first	0,1	2
second	0,1	2
equal	0,1	2

Table 1: Summary of possible values of *num1* and *num2*.

From the table above we can see that num1 & num2 each have four possible values.

Let " X_1 " be the event of " $\text{num1} > \text{num2}$ ".

Number of possible outcomes of event $X_1 = \{(011,000), (011,001), (011,010), (010,000), (010,001), (001,000)\}$

Let " X_2 " be the event of " $\text{num2} > \text{num1}$ ".

Number of possible outcomes of event $X_2 = \{(011,000), (011,001), (011,010), (010,000), (010,001), (001,000)\}$

Let " X_3 " be the event of " $\text{num1} = \text{num2}$ ".

Number of possible outcomes of event $X_3 = \{(011,011), (010,010), (001,001), (000,000)\}$

Therefore from the definition of probability we can see that, the probability of occurrence of event " X_1 " is

$$P(X_1) = \frac{6}{16} = 0.375$$

Similarly, the probability of occurrence of event " X_2 " is

$$P(X_2) = \frac{6}{16} = 0.375$$

The probability of entering the loop " X_3 " is

$$P(X_3) = \frac{4}{16} = 0.25$$

2.5.4 Test Bench

A test bench is used to simulate this design. To simulate, the test bench provides the Design Under Test (DUT) and the stimulus. The test bench is the HDL code, which generates the stimulus and applies this stimulus to the DUT and generates the output responses. These output responses are compared to the expected values to check for errors.

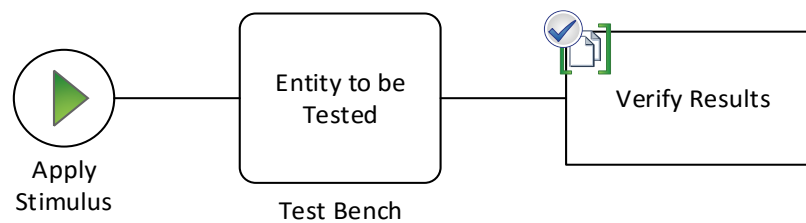


Figure 2.15 Architecture of Test Bench

The basic building blocks of any test bench are scheduler, driver, receiver, UUT interfaces and text I/O sub programs. The scheduler acts as the heart of the test bench. Its functions include reading stimulus and expected response files. The driver acts as a message queuing ENTITY. It receives messages and control information from the scheduler and queues messages to the appropriate UUT input. The receiver entity monitors the output of the UUT, and reports activity to the scheduler and to the result file. The UUT interfaces format the data passed to and from the UUT [14].

The text I/O sub programs are shared routines that format input and output information and interface them directly with the standard VHDL STD.TEXTIO routines. The Text I/O interfaces carry text information into and out of the test bench. The interface between the scheduler and the Text I/O routines contains two paths. The first is

the input scheduler path consisting of Text I/O routines which pass the scheduler data from the stimulus/response file. This data is of standard type STD.TEXTIO.LINE, and is read using standard procedure STD.TEXTIO.READLINE [14].

The scheduler parses the line using the standard procedure STD.TEXTIO.READ

```
file test_vectors: TEXT open read_mode is "C:\Modeltech_pe_edu_10.4a\examples\work\Cleaned_Vectors.txt";
```

```
WHILE (NOT ENDFILE(test_vectors)) LOOP
```

```
  READLINE (test_vectors, current_line);
```

```
  READ(current_line, a_input_test);
```

```
  READ(current_line, b_input_test);
```

Figure 2.16 Pseudo code for Reading Input Test Vectors

Test vectors generation

With the advent of VLSI technology and increasing the complexity of VLSI chips, ensuring correctness of behavioral models before they are released for synthesis and logic design phase is essential to achieve a high quality product with tight deadlines. The current state of testing complex behavioral models for bugs is a tedious process. Millions or even billions of test vectors are applied to behavioral model before they are shipped. This *test-it-to-death* approach is wasteful and does not direct limited resources effectively to potential buggy portions of the model. Most designers still use a manual test generation process that typically provides test coverage of between 50% and 70% of the potential faults. If highest coverage is desired, the time and cost for developing the test vectors would increase exponentially [14].

Various statistical techniques have been used to determine the number of test vectors needed to achieve a particular test objective. Stopping rules can be used to save testing time and costs while still maintaining the expected quality of the model. There are various models proposed to find an efficient and economical stopping rules, like Bayesian stopping rule which uses Poisson counting process compounded with logarithmic series distribution (LSD). These stopping rule methods are complex and are difficult to apply to all the models [14].

Fault coverage

Fault coverage is an important concept in VLSI test. Fault coverage is used to evaluate the efficiency of the test, the efficiency of different test structures, different test algorithms and different test strategies in terms of fault coverage [16].

People usually intend to pursue high fault coverage to achieve high chip quality.

$$Fault\ Coverage = \frac{Number\ of\ detected\ Faults}{Total\ Number\ of\ Faults}$$

Different Stopping rules evaluation methods are very complex and are difficult to apply for all the behavioral model structures. Therefore, random vector generation process is used to test the Modified ALU in this thesis. Although the test vectors randomly are generated manually, the Fault coverage value with these test vectors is maintained very high. For this purpose we are generated around 400,000 test vectors which gave us a fault coverage value of 0.8.

2.6 Summary

In this chapter, several Trojan detection methodologies which have been developed over the past years have been discussed. Power and timing based side channel signal analysis, have been analyzed. After which, various Trojan activation methods including Region free and Region aware Trojan activation techniques and their advantages have been examined. Finally, the need for Hardware Trust and methodology which helps us in achieving this has been surveyed. Although we have spoken about the ways by which the models help us detect Trojans, they do have their own drawbacks as listed below.

1. **Power Based Analysis** - In modern Nano scale technology based ICs, the amount of parameter variation can be much more than 7.5% whereas, the Trojan circuit area can be as low as 0.01%. In which case, the power generation would be too small [77]. In the current integration method that has been discussed, the amount of current drawn by a Trojan can be at times so small that it can be submerged into an envelope of noise and process variation effects and thus be undetectable by conventional measuring equipment. [13].
2. **Timing Based Analysis** – In the model proposed by Li and Lach [5], the model would require large considerable area overhead when targeting today's large designs with millions of parts especially the short ones would not be practical.[13]

Trojan Activation Methods – In order to activate any particular Trojan instance, we must trigger it. Thus the generation of the optimal set of test vectors will remain an important problem [11]. Along with these, the test engineer would not knowing the Trojan type or size initially, hence both region free and region aware methods would be needed. If the

Trojan input comes from a part of the circuit, which are functionally dependent, then region aware method would be effective else region free method would be more effective.

S.No	Paper	Test Modality	Drawbacks
1	Agarwal et al.[3]	Transient Power (Simple Power Analysis)	Would not work if size of Trojan is very small compared to the size of circuit
2	Wang et al [4]	Transient Power (Current Integration)	Would not work if size of Trojan is very small compared to the size of circuit
3	Li and Lach [5]	Delay	Would require large overhead and would be difficult to find delay for all the million paths in the circuit to develop fingerprint
4	Jin and Markis [6]	Delay	Would require large overhead and would be difficult to find delay for all the million paths in the circuit to develop fingerprint
5	Banga and Hsiao[8][9]	Transient power and pattern generation	Generation of optimal set of test vectors to activate a particular Trojan would be difficult
6	Banga and Hsiao[12]	Design for Trust and transient power	Switching Power Supply Voltage for each power supply Voltage and ground would be difficult

Table 2: Summary of Key Trojan detection methods

All these models have limitations be it with Side-channel analysis, whose effectiveness reduces with decreasing Trojan size and increasing process induced parameter variations. Logic testing loses its effectiveness in circuits with multi-millions of transistors, because one has to generate the exact input condition to activate the trigger function that causes the alteration in logic value. It can be very challenging to generate such input conditions, and none of these methods explain methods of Trojan detection at the behavioral level as most of these methods can be applied to the circuits post manufacturing. Hence we need an alternate approach which would help us detect the Trojans right at the behavioral level.

Chapter 3 Mathematical Model and Tech Bench Setup

In this chapter, a mathematical model is developed to associate, the probability of a given line of code to the vulnerability of that line of code. The correctness of this model is verified by implementing this model in an experimental setup. In this setup, the mathematical model is applied to two 16-bit ALU's, one a Golden ALU (error free) and second a Modified ALU (has Insertion-errors).

3.1 Mathematical Model

Two random variables A&B are defined, each with 'd' bits of length and would take values from 0 to 2^d .

“A” is a discrete uniform random variable with parameter $N_a = 2^d$

Where $P(A=a) = \frac{1}{N_a}$ and $a=0, 1, 2, 3, \dots, 2^d$

Similarly,

“B” is a discrete uniform random variable with parameter $N_b = 2^d + 1$

Where $P(A=b) = \frac{1}{N_b}$ and $b=0, 1, 2, 3, \dots, 2^d$

Let X_1 be a Bernoulli random variable denoting occurrence of a loop, and Q & R denote the set of values which A&B take for this particular loop X_1 . i.e, this loop gets executed if “A” takes values from set Q and B takes values from set R.

N_Q - denotes the number of values in set Q, and

N_R - denotes the number of values in set R.

$$P(X_1=1) = P_1, \text{ and}$$

$$P(X_1=0) = 1 - P_1$$

Where P_1 is the probability " X_1 " occurring.

$$P_1 = P(A \text{ and } B)$$

Since A & B are two independent events

$$P(A \text{ and } B) = P(A) * P(B) \tag{12}$$

$$= \frac{N_Q}{N_A} * \frac{N_R}{N_B} \tag{13}$$

Another function H is defined where

$$H = \{i: P_i = \min(P_1, P_2, P_3, P_4, P_5)\} , \text{ where } P_1, P_2, P_3, P_4, P_5 \text{ denote the probability of occurrence of loop } X_1, X_2, X_3, X_4, X_5 \tag{14}$$

And assume that H would give the loop which is the most vulnerable towards Insertion-errors.

This is a very generic mathematical model, which can be used for any number of loops and inputs vectors of any size.

3.2 Test Setup

The Test setup uses two 16 bit ALUs, one Golden version and other Modified Version. These 16-bit ALUs take two Vectors A&B, each of 16-bits in length, and ALUCOUNT, 3 bits in length, as inputs and performs different functions on A&B based on the value of ALUCOUNT. The output of the ALU is the RESULT vector which is again 16-bits in length.

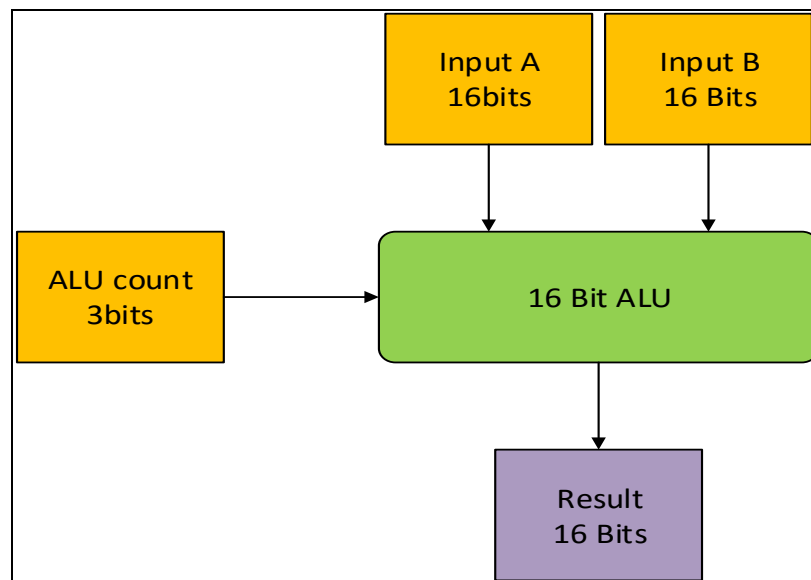


Figure 3.1: ALU Overview

3.2.1 Golden ALU

The following are the different operations which are performed on A&B based on the value of ALUCOUNT in the Golden ALU.

S.No	ALUCOUNT	Corresponding Operation
1	000	OR
2	001	AND
3	010	XOR
4	011	SUM

Table 3: Summary of ALUCOUNT and Corresponding Operations

```

Golden ALU:
Switch alucount(2 downto 0)
case 1: result <= a and b;
case 2: result <= a or b ;
case 3: result <= a xor b ;
case 4: result <= sum;
end Switch;

```

Figure 3.2 Pseudo Code for Golden ALU.

Note: Complete Code is attached in the appendix section.

3.2.2 Modified ALU

The Modified ALU is modified by changing the functionality of the Golden ALU. For e.g., Instead of performing “AND” operation when ALUCOUNT is one, the modified ALU will perform “OR” function etc.

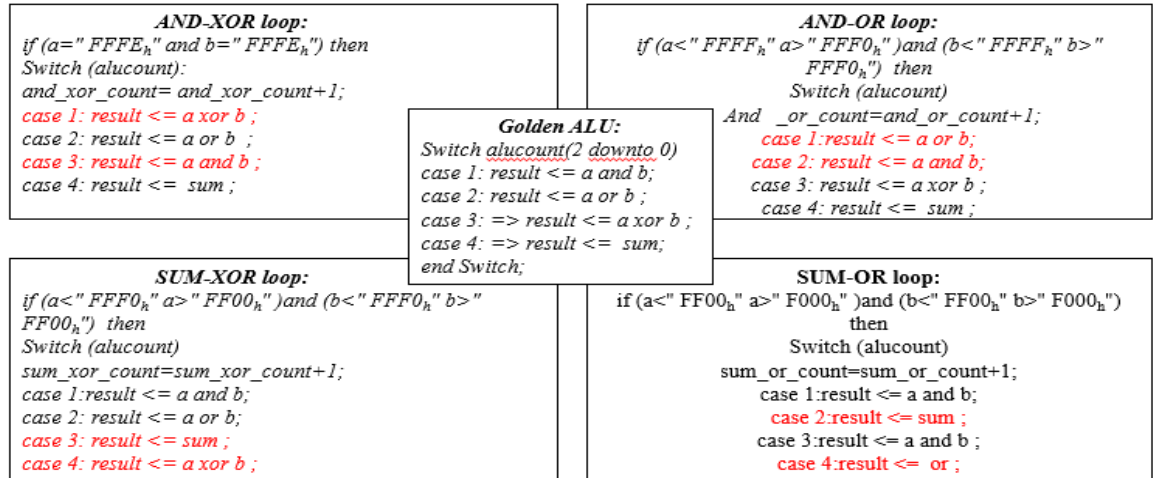


Figure 3.3 Pseudo Code for Modified ALU.

3.3 Test Vector Generation

Random vector generation process is used to test the Modified ALU in this thesis. Although the test vectors are generated randomly, the Fault coverage value with these test vectors is maintained very high (0.8). We have generated three sets of uniform random vectors with sample spaces of four hundred thousand test vectors , four million and twenty million[15].

Since, we are using 32 bit vectors (both A&B combined), the total number of test vectors to be generated for exhaustive testing is $2^{32} = 4.2 \text{ billion vectors}$. Although 4.2billion is a small quantity to test, this exhaustive testing would becomes prohibitive if the number of bits becomes more like 256, in which case the case the number of test vectors to be generated would be 2^{256} . Therefore we have generated random uniform test vectors.

Chapter-4 Experimental Results and Simulation

By using the concepts detailed in the mathematical model, the probability of execution and the number of events in favor of execution of each loop are considered in this chapter. In this chapter, simulations are run on the Modified ALU code to find out the number of events in favor of execution each loop and the relation between probability for a given line of code and the vulnerability associated with detecting errors in that line of code is found for a given set of input test vectors.

The Modified ALU consisting of five loops has been setup such the probabilities of each loop ranges from 0 to 1. These loops have errors inserted into them at different locations. Since we have a prior knowledge on the data distribution before making inferences on the probabilities obtained from each of these loops these probabilities are calculated for the Modified ALU are priori probabilities.

4.1 Modified ALU probabilities of execution

Loop1: AND-XOR loop: In this loop “and” and “xor” operations are interchanged and this loop gets executed only if the values of A&B are equal to "FFFEh" and every time this loop gets executes and xor_count value gets incremented by one. The following pseudo code best describes the loop.

AND-XOR loop:

if (a="FFFE_h" and b="FFFE_h") then

Switch (alucount):

and_xor_count= and_xor_count+1;

case 1: result <= a xor b ;

case 2: result <= a or b ;

case 3: result <= a and b ;

case 4: result <= sum ;

Figure 4.1 Pseudo Code for AND-XOR loop in Modified ALU.

$X_1 = \text{The Event of "AND-XOR" loop getting executed.}$

$N_A = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_B = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_q = \text{Total number of possible values of A for this event} = 1$

since, this loop gets executed for only one value of A

$N_r = \text{Total number of possible values of B for this event} = 1$

since, this loop gets executed for only one value of B

$P(X_1) = \text{Probability of execution of AND – XOR loop}$

$$P(X_1) = P(\text{AND-XOR}) = \frac{1}{65356} * \frac{1}{65356} = 2.34 * 10^{-10}$$

Loop2: AND-OR loop: In this loop “and” and “or” operations are interchanged and this loop is executed only if the values of A&B lie in between "FFFFh" and “FFF0h” and every time this loop gets executes *and_or_count* value gets incremented by one. The following pseudo code best describes the loop.

AND-OR loop:

```

if (a<"FFFFh" a>"FFF0h" )and (b<"FFFFh" b>"FFF0h") then
Switch (alucount)
and_or_count=and_or_count+1;
case 1: result <= a or b;
case 2: result <= a and b;
case 3: result <= a xor b ;
case 4: result <= sum ;

```

Figure 4.2: Pseudo Code for AND-OR loop in Modified ALU.

$X_2 = \text{The Event of "AND-OR " loop getting executed.}$

$N_A = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_B = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_q = \text{Total number of possible values of A for this event}=16$

Maximum Value of a = FFFFh,

Minimum value of a for this loop is =FFF0h

Range of values of a which satisfy this loop condition are

$$\text{FFFFh}-\text{FFF0h}=16$$

Similarly, N_r = Total number of possible values of B for this event = 16

$P(X_2)$ = Probability of execution of AND – OR loop

$$P(X_2) = P(\text{AND-OR}) = \frac{16}{65356} * \frac{16}{65356} = 5.97 * 10^{-8}$$

Loop3: SUM-XOR loop: In this loop “sum” and “xor” operations are interchanged and this loop is executed only if the values of A&B lie in between “FFF0h” and “FF00h” and every time this loop gets executes *sum_xor_count* value gets incremented by one. The following pseudo code best describes the loop.

SUM-XOR loop:

if ($a < \text{"FFF0h"}$ $a > \text{"FF00h"}$) and ($b < \text{"FFF0h"}$ $b > \text{"FF00h"}$) then

Switch (alucount)

sum_xor_count = sum_xor_count + 1;

case 1: result \leq a and b;

case 2: result \leq a or b;

case 3: result \leq sum ;

case 4: result \leq a xor b ;

Figure 4.3 Pseudo Code for AND-XOR loop in Modified ALU.

$X_3 = \text{The Event of "SUM-XOR" loop getting executed.}$

$N_A = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_B = \text{Total number of values which A can take} = 2^{16} = 65356.$

$N_q = \text{Total number of possible values of A for this event} = 240$

Maximum Value of a = FFF0h.

Minimum value of a for this loop is = FF00h

Range of values of a which satisfy this loop condition are

$\text{FFF0h} - \text{FF00h} = 256$

Of these 512, 32 values are covered in the first loop so the remaining values are

$256 - 16 = 240$

$N_r = \text{Total number of possible values of B for this event} = 240$

$P(X_3) = \text{Probability of execution of SUM - XOR loop}$

$$P(X_3) = P(\text{SUM-XOR}) = \frac{240}{65356} * \frac{240}{65356} = 1.3 * 10^{-4}$$

Loop4: SUM-OR loop: In this loop “sum” and “or” operations are interchanged and this loop is executed only if the values of A&B lie in between "FF00h" and “F000h” and every time this loop gets executes “*sum_or_count*” value gets incremented by one. The following pseudo code best describes the loop.

SUM-OR loop:

if ($a < "FF00_h"$ $a > "F000_h"$) and ($b < "FF00_h"$ $b > "F000_h"$) then

Switch (alucount)

sum_or_count = sum_or_count + 1;

case 1: result $\leq a$ and b ;

case 2: result $\leq sum$;

case 3: result $\leq a$ and b ;

case 4: result $\leq or$;

Figure 4.4 Pseudo Code for SUM-OR loop in Modified ALU.

X_4 = The Event of "SUM-OR " loop getting executed.

N_A = Total number of values which A can take = $2^{16} = 65536$.

N_B = Total number of values which A can take = $2^{16} = 65536$.

N_q = Total number of possible values of A for this event = 3840

Maximum Value of $a = FF00_h$,

Minimum value of a for this loop is $= F000_h$

Range of values of a which satisfy this loop condition are

$$FF00_h - F000_h = 3840$$

Of these 3840, 256 values are covered in the first two loops so the remaining values are

$$3840 - 256 = 3584$$

Similarly, N_r = Total number of possible values of B for this event = 3584

$P(X_4) = \text{Probability of execution of SUM - OR loop}$

$$P(X_4) = P(\text{SUM-OR}) = \frac{3584}{65356} * \frac{3584}{65356} = 0.003$$

Loop5: No error loop: This loop is same as the Golden ALU and no operations are interchanged and this loop is executed for all other values of A&B and every time this loop gets executes “no_error_count” value gets incremented by one. The following pseudo code best describes the loop.

NO Error:

```

If ((a > "0000Eh" and a < "FFFFh") or (b > "0000Eh" and b < "FFFFh"))
and ! ((a = "FFFEh" and b = "FFFEh"))
and ! ((a < "FF00h" a > "F000h") and (b < "FF00h" b > "F000h"))
and ! ((a < "FFF0h" a > "FF00h") and (b < "FFF0h" b > "FF00h"))
and ! ((a < "FFFFh" a > "FFF0h") and (b < "FFFFh" b > "FFF0h"))
Switch alucount(2 downto 0)
no_error_count=no_error_count+1;
case 1: result <= a and b;
case 2: result <= a or b ;
case 3: => result <= a xor b ;
case 4: => result <= sum;

```

Figure 4.5 Pseudo Code for NO-Error loop in Modified ALU.

$X_5 = \text{The Event of "SUM-OR " loop getting executed.}$

$N_A = \text{Total number of values which A can take} = 2^{16} = 65356.$

$$N_B = \text{Total number of values which A can take} = 2^{16} = 65356.$$

$$N_q = \text{Total number of possible values of A for this event:-}$$

$$\text{Maximum Value of } a = \text{FFFFh},$$

$$\text{Minimum value of } a \text{ for this loop is } = 0000h$$

$$\text{Range of values of } a \text{ which satisfy this loop condition are}$$

$$\text{FFFFh} - 0000h = 65356$$

$$\text{Similarly, } N_r = \text{Total number of possible values of B for this event} = 65356.$$

$$\begin{aligned} \text{The total number of possible events for this loop to get executed is} \\ = 65356 * 65356 = 4294967296 \end{aligned}$$

$$\begin{aligned} \text{Of these 4294967296 events, 1 event gets used up in loop AND-XOR, 16 in loop} \\ \text{AND-OR, 240 in SUM-XOR, 3840 in SUM-OR.} \end{aligned}$$

$$\text{Therefore the remaining events executed in No-Error loop are}$$

$$= 4294967296 - (1 + 16 + 240 + 3840) = 4294963199$$

$$P(X_5) = \text{Probability of execution of No error loop}$$

$$P(X_5) = P(\text{No Error}) = \frac{4294963199}{(65356 * 65356)} = 0.999$$

The priori probabilities of different loops can be summarized as follows.

Loops	Probabilities
$P(X_1) = P(\text{AND-XOR})$	$2.34 * 10^{-10}$
$P(X_2) = P(\text{AND-OR})$	$5.97 * 10^{-8}$
$P(X_2) = P(\text{SUM-XOR})$	$1.3 * 10^{-4}$
$P(X_4) = P(\text{SUM-OR})$	0.003
$P(X_5) = P(\text{No error loop})$	0.999

Table 4: Summary of Probabilities of various loops

From the table:

$$P(X_1) < P(X_2) < P(X_3) < P(X_4) < P(X_5), \text{ or}$$

$$P(\text{AND-XOR}) < P(\text{AND-OR}) < P(\text{SUM-XOR}) < P(\text{SUM-OR}) < P(\text{No error loop}) \quad (15)$$

4.2 DUT Simulation

The number of events which are in favor of execution of each loop in the Design under Test (Modified ALU) for each set of input test vectors can be found by passing the input test vector sets with the help of Golden_TB (Test Bech) using ModelSim.

Here Golden_TB, supplies three sets of input test stimuli to the DUT (Modified ALU). One set containing 400,000 test vectors, other 4 million and the last set contains 20millions test vectors each of 32bits in length (Vectors A&B) [15].

The probability of execution of each loop will have posteriori probabilities and can be calculated using our Simulation results. The test vectors have been generated using uniform distribution have “a priori” estimations associated with them.

a. **Total Sample Space= 400,000 vectors.**

In the first case we supply 400,000 test vectors as inputs to the DUT. The Simulation below gives us the count of number of events favorable towards execution of each loop for the given randomly generated test vectors.

◆ /alu_tb/and_xor_lo...	32'h80000000	32'h80000000	
◆ /alu_tb/and_or_loo...	32'h80000000	32'h80000000	
◆ /alu_tb/sum_xor_lo...	32'h80000059	32'h80000059	
◆ /alu_tb/sum_or_loo...	32'h80001191	32'h80001191	
◆ /alu_tb/no_error_lo...	32'h8006089A	32'h8...	32'h8...

Figure 4.6 Simulation results for number of events in favor of execution of each loop

Sample Space: 400,000.

The following are the results obtained from the simulation of the DUT against the Test Vectors.

S.No	Loop Name	Number of events in favor of execution of each loop(in Hex)	Number of events in favor of execution of each loop(in Decimal)
1	AND-XOR	0	0
2	AND-OR	0	0
3	SUM-XOR	59	89
4	SUM-OR	1191	4497
5	No Error	6089A	395418

Table 5: Summary of number of events in favor of execution of each loop Sample Space: 400,000

From our simulation table,

Nearly 400,000 randomly generated test vectors which have uniform distribution have been supplied as inputs to the test bench.

The number of events which are in favorable for execution of loop “AND-XOR” are “0”.

Therefore, the estimated probability of execution of this loop would be $\frac{0}{400000} = 0$

Similarly, The number of events which are favorable towards execution of loop “AND-OR” are “5”.

Therefore, the probability of execution of this loop would be $\frac{0}{400000} = 0$

Similarly, The number of events which are favorable towards execution of loop “SUM-XOR” are “41”.

Therefore, the probability of execution of this loop would be $\frac{89}{400000} = 0.0002225$

Similarly, The number of events which are favorable towards execution of loop “SUM-OR” are “5813”.

Therefore, the probability of execution of this loop would be $\frac{4497}{400000} = 0.0112425$

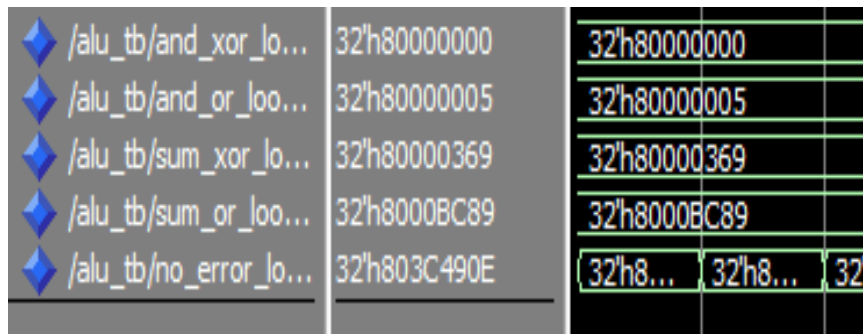
Similarly, The number of events which are favorable towards execution of loop “No-Error” are “394132”.

Therefore, the probability of execution of this loop would be $\frac{395418}{400000} = 0.988545$

Experiment-2

Total Sample Space= 4,000,000 Test Vectors

The number of events favorable towards the execution of each of these loops are



/alu_tb/and_xor_lo...	32'h80000000
/alu_tb/and_or_loo...	32'h80000005
/alu_tb/sum_xor_lo...	32'h80000369
/alu_tb/sum_or_loo...	32'h8000BC89
/alu_tb/no_error_lo...	32'h803C490E

Figure 4.7 Simulation results for number of events in favor of execution of each loop

Sample Space: 4,000,000.

The results from simulation can be summarized as follows.

S.No	Loop Name	Number of events in favor of execution of each loop(in Hex)	Number of events in favor of execution of each loop(in Decimal)
1	AND-XOR	0	0
2	AND-OR	5	5
3	SUM-XOR	369	873
4	SUM-OR	BC89	48265
5	No Error	3C490E	3950862

Table 6: Summary of number of events in favor of execution of each loop Sample Space: 4,000,000

The number of events which are in favorable for execution of loop “AND-XOR” are “0”.

Therefore, the probability of execution of this loop would be $\frac{0}{4000000} = 0$

Similarly, The number of events which are favorable towards execution of loop “AND-OR” are “5”.

Therefore, the probability of execution of this loop would be $\frac{5}{4000000} = 1.25 * 10^{-6}$

Similarly, The number of events which are favorable towards execution of loop “SUM-XOR” are “873”.

Therefore, the probability of execution of this loop would be $\frac{873}{4000000} = 2.18 * 10^{-4}$

Similarly, The number of events which are favorable towards execution of loop “SUM-OR” are “48265”.

Therefore, the probability of execution of this loop would be $\frac{48265}{4000000} = 0.012$

Similarly, The number of events which are favorable towards execution of loop “No-Error” are “3937054”.

Therefore, the probability of execution of this loop would be $\frac{3950862}{4000000} = 0.9877$

Experiment-3:

Total Sample Space= 20,000,000.

The number of events favorable towards the execution of each of these loops are

◆ /alu_tb/and_xor_lo...	32'h80000000	32'h80000000
◆ /alu_tb/and_or_loo...	32'h80000011	32'h80000011
◆ /alu_tb/sum_xor_lo...	32'h8000106D	32'h8000106D
◆ /alu_tb/sum_or_loo...	32'h8003A851	32'h8003A851
◆ /alu_tb/no_error_lo...	32'h812D7436	3... 3...

Figure 4.8 Simulation results for number of events in favor of execution of each loop

Sample Space: 20,000,000.

S.No	Loop Name	Number of events in favor of execution of each loop(in Hex)	Number of events in favor of execution of each loop(in Decimal)
1	AND-XOR	0	0
2	AND-OR	11	17
3	SUM-XOR	106D	4205
4	SUM-OR	3A851	239697
5	No Error	12D7436	19756086

Table 7: Summary of number of events in favor of execution of each loop Sample Space:

20,000,000

The number of events which are in favorable for execution of loop “AND-XOR” are “0”.

Therefore, the probability of execution of this loop would be $\frac{0}{20000000} = 0$

Similarly, The number of events which are favorable towards execution of loop “AND-OR” are “17”.

Therefore, the probability of execution of this loop would be $\frac{17}{20000000} = 0.00000085$

Similarly, The number of events which are favorable towards execution of loop “SUM-XOR” are “4205”.

Therefore, the probability of execution of this loop would be $\frac{4205}{20000000} = 0.00021025$

Similarly, The number of events which are favorable towards execution of loop “SUM-OR” are “239697”.

Therefore, the probability of execution of this loop would be $\frac{239697}{20000000} = 0.01198485$

Similarly, The number of events which are favorable towards execution of loop “No-Error” are “19756086”.

Therefore, the probability of execution of this loop would be $\frac{19756086}{20000000} = 0.98739$

Summarizing the priori probabilities and the posteriori probabilities for the mathematical model and the DUT.

Loops	Probabilities (using exhaustive testing)	Probabilities using 400,000 test vectors	Probabilities using 4,000,000 test vectors	Probabilities using 20,000,000 test vectors
$P(X_1) = P(\text{AND-XOR})$	$2.34 * 10^{-10}$	0	0	0
$P(X_2) = P(\text{AND-OR})$	$5.97 * 10^{-8}$	0	$1.25 * 10^{-6}$	$8.7 * 10^{-7}$
$P(X_2) = P(\text{SUM-XOR})$	$1.3 * 10^{-4}$	$2.2 * 10^{-4}$	$2.1 * 10^{-4}$	$2.1 * 10^{-4}$
$P(X_4) = P(\text{SUM-OR})$	0.003	0.011	0.012	0.0119
$P(X_5) = P(\text{No error loop})$	0.999	0.9885	0.9877	0.9878

Table 8: Exhaustive Testing Probabilities (Priori) and Posteriori Probabilities.

In our test cases, we have established the results with priori and posteriori probabilities. It is important to note that the priori and posteriori probabilities of a given data set at different instances of time will not be identical because the distribution is always normal and random. Let P_{pr} be the priori probability and P_{po} be the posteriori probability then the condition

$P_{pr} = P_{po} + \Delta P$ where “ ΔP ” can either be a measurement error or be a case in which the loop is not executed due to random generation, or due to the design of the loops itself. So a more practical way to determine the convergence will be to get the results from priori probability and then do different experiments for random data sets at different instances of time. This will give a way for us reduce the error in measurement.

From the table we can see that

The number of events in favor of execution of the following loops is in increasing order as follows

$$(AND-XOR) < (AND-OR) < (SUM-XOR) < (SUM-OR) < (No-Error) \quad (16)$$

If the number of events in favor of execution of any loop is zero, it means that the loop never gets executed and it would not get tested for the given set of randomly generated test vectors. It means it's the best place to place an error, as it would never get detected.

Therefore the order of increasing Vulnerability of each loop from the above simulation results would be as follows:

$$(AND-XOR) > (AND-OR) > (SUM-XOR) > (SUM-OR) > (No-Error) \quad (17)$$

From equations (15) and (17) it's clearly understood that

The loops of code, which have lowest probability, are most vulnerable towards errors. This statement is in line with the mathematical model, which has been developed. In the next chapter we will discussing about the priori probabilities and posteriori probabilities.

Chapter-5 Conclusion & Future Work

In this thesis, a mathematical model developed which helps in detecting parts of code that are most vulnerable to errors. This thesis assumes that lines of code which have lower probabilities are the places that are most vulnerable. To check this mathematical model we have used two ALU's, one Golden ALU (which is error free) and the other Modified ALU (which has errors inserted into it). The modified ALU has five loops. The probability of execution of each loop is determined manually and the vulnerabilities associated with each loop can be found out using the Simulation results generated using ModelSim.

$$P(X_1) < P(X_2) < P(X_3) < P(X_4) < P(X_5), \text{ or}$$

$$P(\text{AND-XOR}) < P(\text{AND-OR}) < P(\text{SUM-XOR}) < P(\text{SUM-OR}) < P(\text{No error loop})$$

It was observed that, the vulnerability associated with each of these loops is in increasing order as follows.

$$(\text{AND-XOR}) > (\text{AND-OR}) > (\text{SUM-XOR}) > (\text{SUM-OR}) > (\text{No-Error}) \quad .$$

To test this Modified ALU we have calculated priori and posteriori probabilities of each loop and tried understanding the vulnerability associated with each of these loops. We have generated three sets of uniform random test vectors. The first set has 400,000 vectors in its

sample space, the second set has 4,000,000 vectors in its sample space and the third set has 20,000,000 vectors in its sample space. We have obtained a fault coverage of 0.8 with the help of these test vectors which is acceptable as per industry standards.

The following table summarizes the probabilities of each loop obtained from exhaustive testing which has priori probabilities against randomly generated test vectors have posteriori probabilities.

Loops	Probabilities (using exhaustive testing)	Probabilities using 400,000 test vectors	Probabilities using 4,000,000 test vectors	Probabilities using 20,000,000 test vectors
$P(X_1) = P(\text{AND-XOR})$	$2.34 * 10^{-10}$	0	0	0
$P(X_2) = P(\text{AND-OR})$	$2.23 * 10^{-7}$	$1.25 * 10^{-5}$	$1.25 * 10^{-6}$	$0.25 * 10^{-6}$
$P(X_2) = P(\text{SUM-XOR})$	$5.3 * 10^{-5}$	$1.02 * 10^{-4}$	$5.5 * 10^{-5}$	$6.285 * 10^{-5}$
$P(X_4) = P(\text{SUM-OR})$	$1.36 * 10^{-2}$	0.14	0.15	0.15
$P(X_5) = P(\text{No error loop})$	0.999	0.98533	0.98426	0.98439

Table 9: Exhaustive Testing Probabilities (Priori) and Posteriori Probabilities

From the table we can say that the loops that have lower probabilities have their priori and posteriori values converged and for the loops which have higher probabilities the priori and posteriori values do not converge. It is important to note that the priori and posteriori probabilities of a given data set at different instances of time will not be identical because the distribution is always normal and random.

The “ ΔP ” difference in the two probabilities can be due to a measurement error or be a case in which the loop is not executed because of the particular set of random generation

of test vectors, or due to the design of the loops itself. So a more practical way to determine the convergence and an absolute value of “ ΔP ”, will be to get the results from priori probability and then do different experiments for random data sets at different instances of time. This will give a way for us to reduce the error in measurement.

5.1 Advantages and Conclusion

This thesis developed a new probabilistic approach for determining the vulnerability of a line of code. Probability of execution of a given line of code would help in estimating the vulnerability associated to that line of code.

This model overcomes the difficulties faced in previous methodologies in regards to determining parts of code which are most vulnerable. Since, model does not use any power-based or time-based analysis, the method remains effective irrespective the size of Trojan or the number of paths in circuit. Another advantage with this method is, this methodology can be used to test for the presence Trojans at any stage during the manufacturing process, whereas other methodologies require the chip to be completely manufactured first to check for the presence of Trojans.

5.2 Future Scope of Work

Although this model gives a good place to start using probabilistic approach for determining parts of code which are most vulnerable, this methodology can be extended further to make the error detection process more robust.

1. Automate the Process of Finding the probability.

In this thesis, the probability of each line of code has been manually calculated. This process can be automated by using different scripting or programming languages. This

would help us in speeding up the process and at the same time it would help us in using this methodology for bigger circuits and more vulnerable circuits.

2. Generate Test Vectors Automatically for the loops which have low probability

This thesis has been confined to determining parts of code which are more vulnerable towards Insertion-errors. It would be very useful if new test vectors can be automatically generated for the lines of code whose probability is less than the threshold value set. This process would help in making the entire testing process more robust. The following flow diagram below can help in understand the same more clearly.

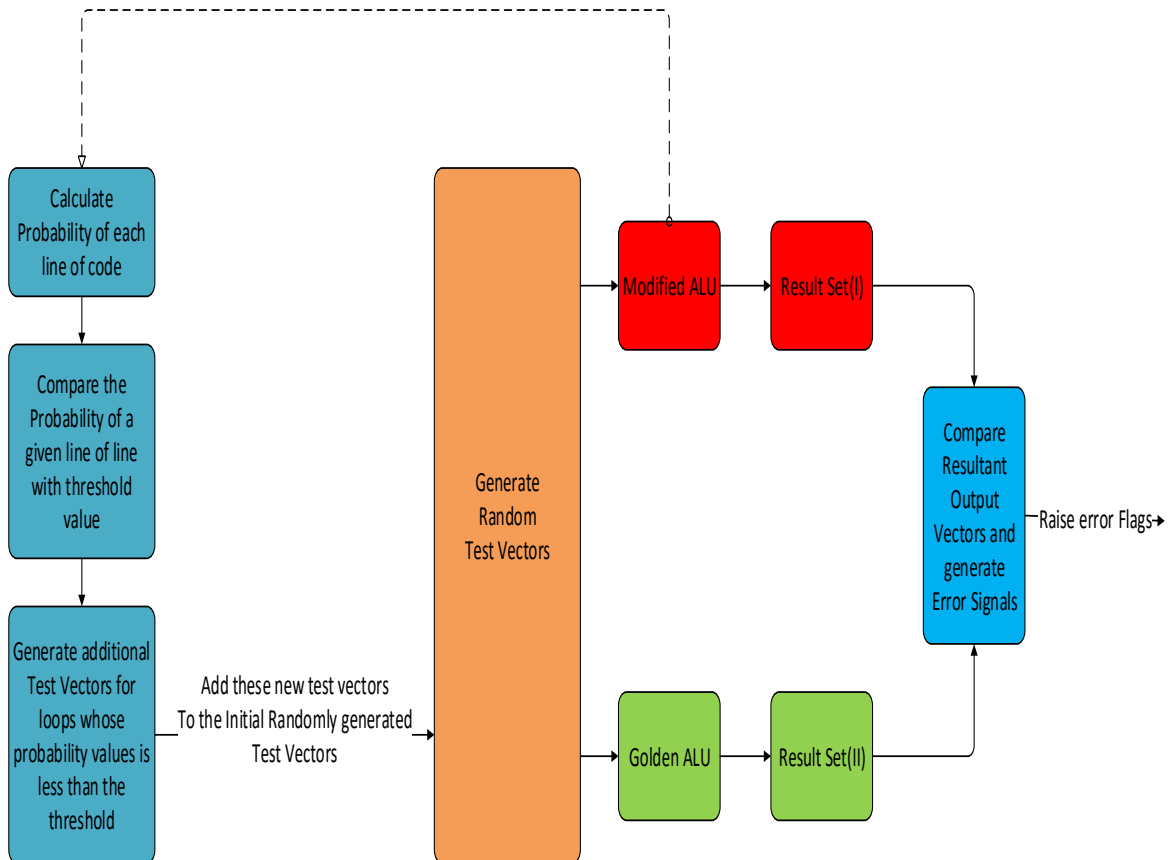


Figure 5.1: Flow diagram for error detection using probabilistic approach

References

- [1] X. Wang, M. Tehranipoor, and J. Plusquellic, “Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions,” Proc. IEEE Int’l Workshop Hardware- Oriented Security and Trust (HOST 08), IEEE CS Press, 2008, pp. 15-19.
- [2] Y. Alkabani and F. Koushanfar, “Extended Abstract: Designer’s Hardware Trojan Horse,” Proc. IEEE Int’l Workshop Hardware-Oriented Security and Trust (HOST 08), IEEE CS Press, 2008, pp. 82-83.
- [3] D. Agrawal et al., “Trojan Detection Using IC Fingerprinting,” Proc. IEEE Symp. Security and Privacy (SP 07), IEEE CS Press, 2007, pp. 296-310.
- [4] X. Wang et al., “Hardware Trojan Detection and Isolation Using Current Integration and Localized Current Analysis,” Proc. IEEE Int’l Symp. Defect and Fault Tolerance of VLSI Systems (DFT 08), IEEE CS Press, 2008, pp. 87-95.
- [5] J. Li and J. Lach, “At-Speed Delay Characterization for IC Authentication and Trojan Horse Detection,” Proc. IEEE Int’l Workshop Hardware-Oriented Security and Trust (HOST 08), IEEE CS Press, 2008, pp. 8-14.
- [6] Y. Jin and Y. Makris, “Hardware Trojan Detection Using Path Delay Fingerprint,” Proc. IEEE Int’l Hardware-Oriented Security and Trust (HOST 08), IEEE CS Press, 2008, pp. 51-57.

- [7] S. Jha and S.K. Jha, “Randomization Based Probabilistic Approach to Detect Trojan circuits,” Proc. 11th IEEE High Assurance Systems Engineering Symp., IEEE CS Press, 2008, pp. 117-124.
- [8] M. Banga and M. Hsiao, “A Region Based Approach for the Identification of Hardware Trojans,” Proc. IEEE Int’l Workshop Hardware-Oriented Security and Trust (HOST 08), IEEE CS Press, 2008, pp. 40-47.
- [9] M. Banga and M. Hsiao, “A Novel Sustained Vector Technique for the Detection of Hardware Trojans,” Proc. 22nd Int’l Conf. VLSI Design, IEEE CS Press, 2009, pp. 327-332.
- [10] I. Verbauwhede and P. Schaumont, “Design Methods for Security and Trust,” Proc. Design, Automation and Test in Europe Conf. (DATE 07), EDA Consortium, pp. 672-677.
- [11] G. Bloom, B. Narahari, and R. Simha, “OS Support for Detecting Trojan Circuit Attacks,” Proc. IEEE Int’l Workshop Hardware-Oriented Security and Trust (HOST 09), IEEE CS Press, 2009, pp. 100-103.
- [12] M. Banga and M. Hsiao, “VITAMIN: Voltage Inversion Technique to Ascertain Malicious Insertion in ICs,” Proc. 2nd IEEE Int’l Workshop Hardware-Oriented Security and Trust (HOST 09), IEEE CS Press, 2009, pp. 104-107.
- [13] M. Tehranipoor, F.Koushanfar, “A Survey of Hardware Trojan Taxonomy and Detection,” in IEEE Design and Test of Computers, 2010, vol., no., 27, issue number 1, Pages 10-25.
- [14] Phil May, “A Flexible VHDL Test Bench Architecture,” Motorola Inc., Government Electronics Group, Tactical Secure Communications Office.
- [15] Communications with Adam Kimura.

- [16] S.Moazzeni, S. Poormozaffari , A. Emami, “An Optimized Simulation-based Fault Injection and Test Vector Generation Using VHDL to Calculate Fault Coverage”, 10th International Workshop on Microprocessor Test and Verification.

Appendix A: Golden ALU

```
-----  
-- Test Article 8-bit MIPS ALU GOLDEN  
-- Adam Kimura & Venkata Sai Manoj  
-- Department of Electrical & Computer Engineering  
-- The Ohio State University  
-- July 24, 2014  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
-----
```

```
-- Entity Declarations  
-----
```

```
entity alu is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than  
    generic(width: integer :=8);  
    port(a, b:   in STD_LOGIC_VECTOR(width-1 downto 0);  
          alucont: in STD_LOGIC_VECTOR(2 downto 0);  
          result: out STD_LOGIC_VECTOR(width-1 downto 0));  
end;
```

```
-- Architecture Declarations
```

```
architecture synth of alu is
```

```
    signal b2, sum: STD_LOGIC_VECTOR(width-1 downto 0);
```

```
begin
```

```
    b2 <= not b when alucont(2) = '1' else b;
```

```
    sum <= a + b2 + alucont(2);
```

```
    Process(a,b,alucont)
```

```
    begin
```

```
        case alucont(2 downto 0) is
```

```
            when "000" => result <= a and b ;
```

```
            when "001" => result <= a or b ;
```

```
            when "011" => result <= a xor b ;
```

```
            when "010" => result <= sum ;
```

```
        end case;
```

```
    End Process;
```

```
end;
```

Appendix B: Modified ALU

-- Test Article 8-bit MIPS ALU GOLDEN

-- Adam Kimura & Manoj

-- Department of Electrical & Computer Engineering

-- The Ohio State University

-- July 24, 2014

library IEEE;

use IEEE.STD_LOGIC_1164.all;

use IEEE.STD_LOGIC_ARITH.all;

use IEEE.STD_LOGIC_UNSIGNED.all;

-- Entity Declarations

entity Modified_ALU is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than

generic(width: integer :=16);

port(a, b: in STD_LOGIC_VECTOR(width-1 downto 0);

alucont: in STD_LOGIC_VECTOR(2 downto 0);

result: out STD_LOGIC_VECTOR(width-1 downto 0);

and_xor_count, and_or_count, sum_xor_count, sum_or_count, no_error_count: out
integer);

END Modified_ALU;

-- Architecture Declarations

architecture synth of Modified_ALU is

signal b2, sum, slt: STD_LOGIC_VECTOR(width-1 downto 0);

begin

b2 <= not b when alucont(2) = '1' else b;

sum <= a + b2 + alucont(2);

Process(a,b,alucont)

variable counter_1,counter_2,counter_3,counter_4,counter_5: integer;

begin

if (a="1111111111111110" and b="1111111111111110") then

counter_1:=counter_1+1;

and_xor_count<= counter_1;

case alucont(2 downto 0) is

when "000" => result <= a xor b ;

when "001" => result <= a or b ;

when "011" => result <= a and b ;

when "010" => result <= sum ;

when others => result <= slt ;

end case;

elsif ((a<"111111111111111" and a>"1111111111110000") and
(b<"111111111111111" and b>"1111111111110000")) then

counter_2:=counter_2+1;

and_or_count<=counter_2+1;

case alucont(2 downto 0) is

when "000" => result <= a and b ;

when "001" => result <= a or b ;

when "011" => result <= a and b ;

when "010" => result <= sum ;

when others => result <= slt ;

```

end case;

elsif (a<"1111111111110000" and a>"1111111100000000") and
(b<"1111111111110000" and b>"1111111100000000") then
    counter_3:=counter_3+1;
    sum_xor_count<=counter_3+1;
    case alucont(2 downto 0) is
        when "000" => result <= a xor b ;
        when "001" => result <= a or b ;
        when "011" => result <= a and b ;
        when "010" => result <= sum ;
        when others => result <= slt ;
    end case;

elsif (a> "1111000000000000" and a<"1111111100000000") and
(b<"1111111100000000" and b>"1111000000000000") then
    counter_4:=counter_4+1;
    sum_or_count<=counter_4+1;
    case alucont(2 downto 0) is
        when "000" => result <= a xor b ;
        when "001" => result <= a or b ;
        when "011" => result <= a and b ;
        when "010" => result <= sum ;
        when others => result <= slt ;
    end case;

else if ((a>"0000000000000000" and a<"1111111111111111") or (
b>"0000000000000000" and b<"1111111111111111"))
    and !(((a<"1111111111111111" and a>"1111111111110000") and
(b<"1111111111111111" and b>"1111111111110000"))

```

```

        and !((a<"1111111111110000" and a>"1111111100000000") and
(b<"1111111111110000" and b>"1111111100000000"))

        and !((a> "0000000000000000" and a<"1111111100000000") and
(b<"1111111100000000" and b>"0000000000000000"))

        and !((a="111111111111110" and b="111111111111110"))

        counter_5:=counter_5+1;
        no_error_count<=counter_5+1;
        case alucont(2 downto 0) is
            when "000" => result <= a or b ;
            when "001" => result <= a and b ;
            when "011" => result <= a xor b ;
            when "010" => result <= sum ;
            when others => result <= slt ;
        end case;
    end if;
End Process;

end;

```

Appendix C: Test Bench

```
-----  
--  
-- Adam Kimura & Venkata Sai Manoj  
-- Department of Electrical & Computer Engineering  
-- The Ohio State University  
-----  
  
library IEEE;  
library STD;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;  
use STD.TEXTIO.ALL;  
  
entity ALU_TB is  
    generic(width: integer :=16);  
end ALU_TB;  
  
architecture Testbench of ALU_TB is  
    signal A_TB, B_TB: STD_LOGIC_VECTOR (width-1 DOWNT0 0);  
    signal alucont_TB: STD_LOGIC_VECTOR(2 DOWNT0 0);  
    signal error :      integer range 0 to 1;  
    signal result_TB: STD_LOGIC_VECTOR(width-1 DOWNT0 0);  
    signal test_number, vector_number : integer := 0;
```

```

    signal  and_xor_loop_frequency,  and_or_loop_frequency,  sum_xor_loop_frequency,
    sum_or_loop_frequency, no_error_loop_frequency: integer;

```

component Modified_ALU is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than

```

    generic(width: integer :=16);

    port(a, b:  in STD_LOGIC_VECTOR(width-1 downto 0);
          alucont: in STD_LOGIC_VECTOR(2 downto 0);
          result: out STD_LOGIC_VECTOR(width-1 downto 0);
          and_xor_count,and_or_count,sum_xor_count,sum_or_count,no_error_count: out
integer
          );
end component;

```

BEGIN

-- Instatiating the Device Under Test (DUT)

```

DUT:Modified_ALU    port    map(A_TB,    B_TB,    alucont_TB,    result_TB,
and_xor_loop_frequency,    and_or_loop_frequency,    sum_xor_loop_frequency,
sum_or_loop_frequency, no_error_loop_frequency);

```

Vector_injection: PROCESS

-- Specifying the file which will read the test vectors via STD.TEXTIO.vhdl package

```

file      test_vectors:      TEXT      open      read_mode      is
"C:\Modeltech_pe_edu_10.4a\examples\work\Cleaned_Vectors.txt";

```

-- Specifying the file which will write the expected output vectors via STD.TEXTIO.vhdl package

```

file      expected_result:      TEXT      open      write_mode      is
"C:\Modeltech_pe_edu_10.4a\examples\work\Expected_Result(1).txt";

```

variable current_line : LINE;

variable a_input_test, b_input_test : BIT_VECTOR (width-1 DOWNT0 0);


```

variable output : BIT_VECTOR (width-1 DOWNT0 0);
variable test_number_int, vector_number_int, error_number : integer := 0;

BEGIN

    WHILE (NOT ENDFILE(test_vectors)) LOOP -- Going through the entire Test Vector
    File to read in

        -- Acquiring the next test vector line
        READLINE (test_vectors, current_line);
        READ(current_line, a_input_test);
        READ(current_line, b_input_test);

        -- Tracking the Vector number
        vector_number_int := vector_number_int + 1;
        vector_number <= vector_number_int;

        -- Converting the A and B input vectors from BIT_VECTOR type to
        STD_LOGIC_VECTOR before assigning it to the ENTITY inputs.
        A_TB <= to_stdlogicvector(a_input_test);
        B_TB <= to_stdlogicvector(b_input_test);

        -- Talley of the number of input vector combinations being used.
        alucont_TB <= "000"; -- Testing the Logical AND Operation
        test_number_int := test_number_int + 1;
        test_number <= test_number_int;
        wait for 25 ns;
        WRITE(current_line, bit_vector'(to_bitvector(result_TB)));
        WRITELINE(expected_result, current_line);
        wait for 25 ns;

```

```

-- Tracking the Vector number
vector_number_int := vector_number_int +1;
vector_number <= vector_number_int;

-- Converting the A and B input vectors from BIT_VECTOR type to
STD_LOGIC_VECTOR before assigning it to the ENTITY inputs.
A_TB <= to_stdlogicvector(a_input_test);
B_TB <= to_stdlogicvector(b_input_test);

alucont_TB <= "001"; -- Testing the Logical OR Operation
test_number_int := test_number_int + 1;
test_number <= test_number_int;
wait for 25 ns;
WRITE(current_line, bit_vector'(to_bitvector(result_TB)));
WRITELINE(expected_result, current_line);
wait for 25 ns;

-- Tracking the Vector number
vector_number_int := vector_number_int +1;
vector_number <= vector_number_int;

-- Converting the A and B input vectors from BIT_VECTOR type to
STD_LOGIC_VECTOR before assigning it to the ENTITY inputs.
A_TB <= to_stdlogicvector(a_input_test);
B_TB <= to_stdlogicvector(b_input_test);

```

```

alucont_TB <= "010";  -- Testing the sum Operation
test_number_int := test_number_int + 1;
test_number <= test_number_int;
wait for 25 ns;
WRITE(current_line, bit_vector'(to_bitvector(result_TB)));
WRITELINE(expected_result, current_line);
wait for 25 ns;

    -- Tracking the Vector number
vector_number_int := vector_number_int + 1;
vector_number <= vector_number_int;

    -- Converting the A and B input vectors from BIT_VECTOR type to
STD_LOGIC_VECTOR before assigning it to the ENTITY inputs.
A_TB <= to_stdlogicvector(a_input_test);
B_TB <= to_stdlogicvector(b_input_test);

alucont_TB <= "011";  -- Testing the sum Operation
test_number_int := test_number_int + 1;
test_number <= test_number_int;
wait for 25 ns;
WRITE(current_line, bit_vector'(to_bitvector(result_TB)));
WRITELINE(expected_result, current_line);
wait for 25 ns;

END LOOP;
END PROCESS;

end Testbench;

```

Appendix D: Test Vectors Generation Code

```
# Written by Adam Kimura
# Aprl 22, 2015
# Department of Electrical & Computer Engineering
# The Ohio State University
#
# This script will generate a set of two N-bit random test vectors. This is ideal for going
into a 2-input ALU or
# 2-input Floating Point Adder Unit. The script will prompt the user to enter the number
of bits that each vector
# should contain and the number of random test vector sets that should be generated.

#-----
-----

## Function for generating the test vectors

def Vector_Generation(Number_of_Test_Vectors, N):
    from random import randint

    Generated_Test_Vectors = open("Generated_Test_Vectors.txt", 'wb')    # Creating
    TXT file to save vectors into

    vector_A = []    # Initializing empty Vector A Array
    vector_B = []    # Initializing empty Vector B Array

    for x in range (0,Number_of_Test_Vectors): # Selecting the vector number that loop is
    creating
        for i in range (0,N):    # Selecting the bit in the selected vector to write
```

```

vector_A.append(randint(0,1))    # random 0 or 1 selection for vector A
vector_B.append(randint(0,1))    # random 0 or 1 selection for vector B

print (vector_A, vector_B)      # Printing to the monitor to be cute

print >> Generated_Test_Vectors, vector_A, vector_B    # Saving the vectors to
the file

vector_A = []                   # Re-initializing Vector A to empty for the next vector
creation

vector_B = []                   # Re-initializing Vector B to empty for the next vector
creation

#-----
#####

## Function for cleaning the array list of the generated test vectors in order to translate it
into a format that can
## be read into the VHDL Architecture Model

def clean_text(line):
    line = line.replace("[", "")
    line = line.replace("]", "")
    line = line.replace(", ", "")
    return line

#-----
#####

## Main Program

# Specifying the number of bits each vector should have
N = int(input('Specify the number of bits that each Test Vector should have: '))

```

```

## Input the number of test vectors that need to be generated for each inputs A and B
Number_of_Test_Vectors = int(input('Specifying the number of Test Vectors to be
generated: '))

Vector_Generation(Number_of_Test_Vectors,N)      # Generate the test vectors based
upon the user information

# Opening up the Generated Test Vectors in order to clean them with the cleaning script
with open("Generated_Test_Vectors.txt", 'rt') as vector_file:
    Cleaned_Vectors = open("Cleaned_Vectors.txt", "w")
    for line in vector_file:
        line = clean_text(line)
        print >> Cleaned_Vectors, line,

```