TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAF70LT

Viljar Indus 132328IASM

RESEARCH ON WEIGHTED PSEUDO-RANDOM TESTING

Master thesis

Raimund-Johannes Ubar

PhD

Professor

Tallinn 2015

Author’s declaration of originality

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

Author: Viljar Indus

04.06.15

Abstract

The goal of this thesis is to provide insight on weighted pseudo random testing methods. Different weight generation methods are discussed and compared. As a result of this thesis a simulator for one of these methods is created. The simulator will be used to analyze the impact weighted testing has on circuits of different size and complexity. Also the experimental results will be compared against standard pseudo-random testing methods. In addition to this a lab assignment is proposed that incorporates the simulator. The aim of this lab exercise is for students to learn about the effects of weighted pseudo-random testing can have on testing circuits.

This thesis is written in English and is 42 pages long, including 7 chapters, 15 figures and 5 tables.

Annotatsioon

Kaalutud juhuslike vektoritega testimismeetodite uurimine

Selle lõpputöö eesmärgiks oli uurida juhuslike kaalutud vektoritega testimismeetodeid. Töö käigus tuuakse välja erinevad kaalude arvutamisalgoritmid. Ühe algoritmi põhjal luuakse testi simulaator. Selle simulaatoriga viiakse läbi erinevaid eksperimente, et uurida, millist mõju avaldab testidele sisendite genereerimistõenäosuste muutmine. Kaaludega testimise efektiivsuse hindamiseks võrreldakse eksperimentide tulemusi standardse juhuslike vektorite meetodiga. Lisaks sellele esitatakse töös potentsiaalne labori ülesanne, milles kasutatakse lõputöös loodud testimise tööriista.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 42 leheküljel, 7 peatükki, 15 joonist, 5 tabelit.

Table of abbreviations and terms

ATE	Automatic Test Equipment
ATPG	Automatic Test Patern Generator
BDD	Binary Decision Diagrams
BILBO	Built-in Logic Block Observer
BIST	Built-in Self-test
CUT	Circuit under Test
ISCAS	IEEE International Symposium on Circuits and Systems
LFSR	Linear Feedback Shift Register
MISR	Multiple Input Signature Register
MUX	Multiplexer
NCV	Non-Controlling Value
NDI	Number of Device Inputs
PRPG	Pseudo-random Pattern Generator
SA-0	Stuck-at-0
SA-1	Stuck-at-1
SSBDD	Structurally Synthesized Binary Decision Diagrams
STUMPS	Self-test Using MISR and PRPG Structures
TT	Turbo Tester
WF	Weight Factor

WPRPG	Weighted Pseudo-random Generator
WV	Weighted Value

Table of contents

1. Introduction 10
2. Methods for Calculating Weights 14
   2.1. Bit Fixing 14
   2.2. Sample rate increasing method 15
   2.3. Structural Analysis 18
      2.3.1. Global Weight Calculation Algorithm 19
3. Generation of Weighted Patterns 21
4. Simulator for Weighted Pseudo Random Generator 24
   4.1. Overview 24
   4.2. Structure 25
      4.2.1. Graph conversion algorithm 26
   4.3. Commands 27
5. Experimental Results 29
6. A Possible Lab Exercise 33
7. Conclusion 34
References 35
Appendix 1 – Implementation for the graph conversion algorithm 36
Appendix 2 – Custom Circuits used in experiments 38
Appendix 3 – Lab Exercise: Weighted Pseudo-random Testing 40

List of figures

Figure 1 LFSR (Type II) and its generated sequences 11

Figure 2 Typical fault coverage curve for PRPG 11

Figure 3 Standard pseudo-random test compared to test with reseeding 12

Figure 4 Implementation of a bit fixing generator[3] 15

Figure 5 Weight Calculation Example 19

Figure 6 STUMPS architecture 21

Figure 7 Implementation of a weight generator 22

Figure 8 SSBDD and its equivalent circuit 24

Figure 9 Workflow of the simulator 25

Figure 10 Weight conflict example 30

Figure 11 Calculation of weights with conflicts 32

Figure 12 Schematic T1 38

Figure 13 Schematic T2 38

Figure 14 Schematic T3 39

Figure 15 Schematic T4 39

List of tables

Table 1 Initial weights 16

Table 2 Improved Weights 17

Table 3 Formulas for calculating weights 20

Table 4 Test results 29

Table 5 Use of multiple weights 31

1. Introduction

The developments in microelectronics have made it so that computers are being embedded to everywhere around us. Since computers influence our daily lives so much, it is important to make sure that it is also safe and reliable. Therefore the importance of testing has increased. However keeping up with the exponential growth of Moore's Law has been a constant struggle for testing.

In the previous century a lot of circuit testing was done by ATE-s. These are machines that generate test patterns and insert them externally to the circuit. External ATE-s were however very expensive and had trouble keeping up with the speeds of the circuits that they were testing. As the number of gates in a microchip grew exponentially over the years, test apparatus were started to be embedded to the designs themselves, thus creating BIST.

BIST provided a cheap and efficient way of testing complex circuits. However it provided a new constraint to the testing process. In addition to having high fault coverage, testing methods also need to have a low area overhead. This resulted in deterministic methods going out of favor as the number of test vectors needed to be stored onboard the chips memory proved to be too high.

Instead methods that used pseudo-random test patterns started to grow more important. The advantage of pseudo-random testing is that it only requires a simple shift register, commonly known as LFSR, to create any number of test patterns. Essentially an n-bit LFSR is a circular state machine with 2^n-1 states.

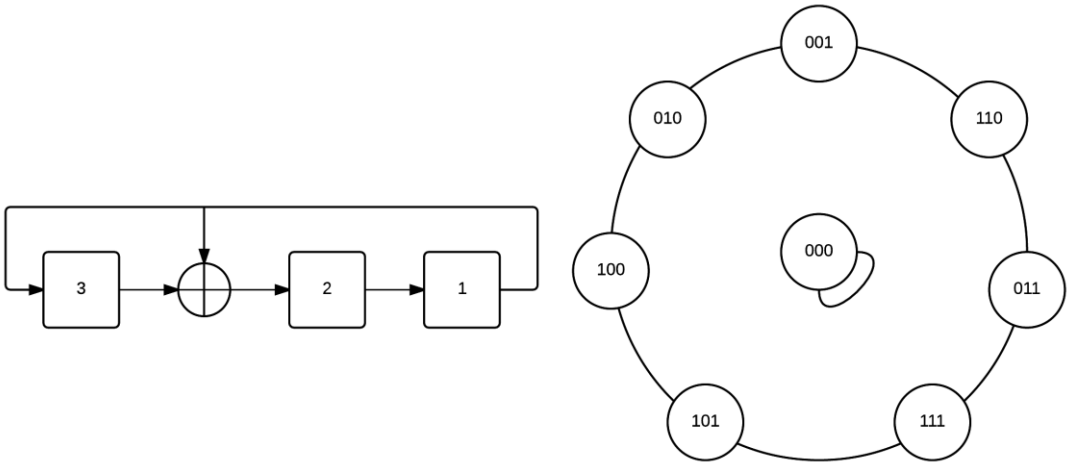

Figure 1 LFSR (Type II) and its generated sequences

Even though the generated patterns appear to be random, they are generated in a deterministic order. Actually, for non-exhaustive testing, the difference between the lengths of random and pseudo-random tests can be neglected [1]. This is an important feature as the actual length of the test can be predetermined during simulation.

Using random vectors to test circuits is a surprisingly effective technique as relatively high fault coverage can be achieved with only a small number of vectors. The reason behind this is that most faults can be detected by multiple test patterns. Not having to store vectors that cover these faults is a major advantage over deterministic testing methods.

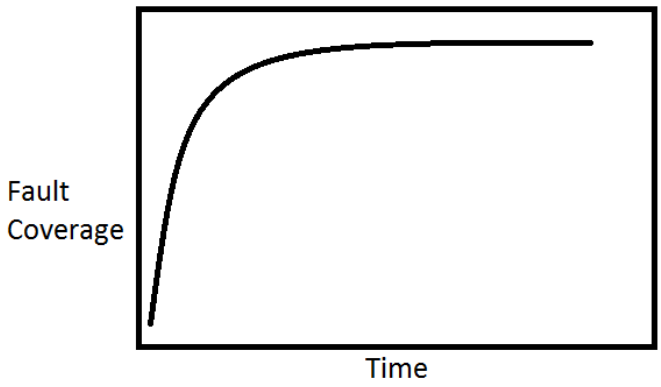

Figure 2 Typical fault coverage curve for PRPG

However the test lengths increase drastically if there are faults that can be detected only by very specific input combinations. In fact, only faults that are the hardest to detect

contribute to the overall length of a pure pseudorandom test. Faults which have a detection probability that is more than ten times greater than the fault that is the hardest to test can be ignored while analyzing the length of a pseudo-random test [1].

There are many ways to improve the quality of a pseudo-random test. A hybrid method was proposed where a pseudo-random test used to reach a high level of fault coverage and the final faults are covered by deterministic vectors. Another approach to reduce the length of a pseudo-random test is to use multiple seeds for the LFSR. By reseeding the pattern generator we are reducing the distance between two vectors that cover random pattern resistant faults which otherwise would have been further apart in the pseud-random sequence.

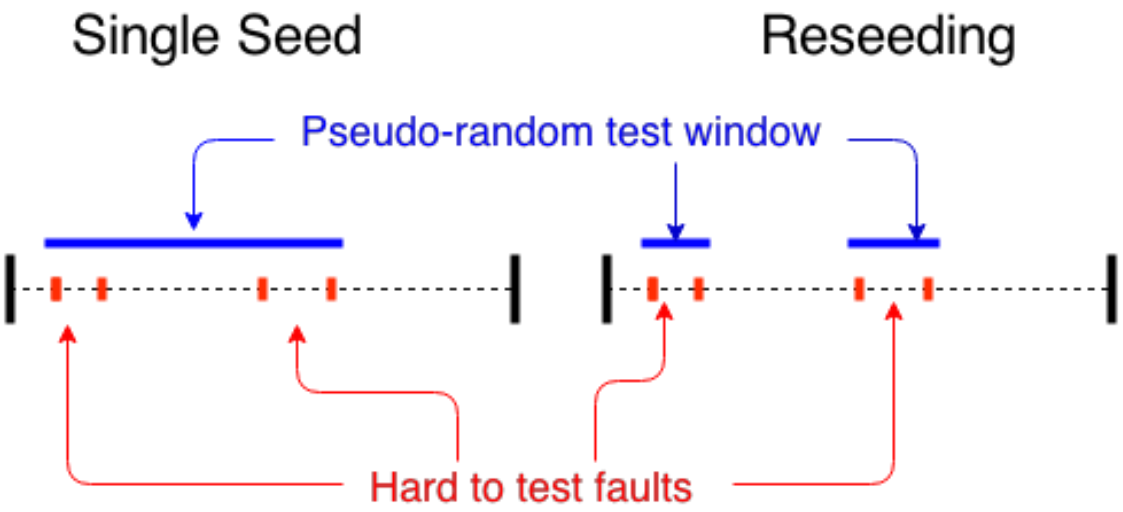

Figure 3 Standard pseudo-random test compared to test with reseeding

An alternative approach would be to change the behavior of the pattern generator itself. The standard LFSR has an equal probability of generating any of any input combinations. Since only the faults that are the hardest to test matter, it would be advantageous to increase the probability of vectors that cover them. This is done by skewing the probability each input towards the values of these vectors. As a result weight set for each input is created. This method is known as weighted pseudo-random testing.

The purpose of this thesis is to study different weight generation methods as well as methods for generating different weights. A simulator for one of the weight generation algorithms was created. This simulator was later used to conduct experiments to assess

the impact of the generated weights had on the test length compared to a standard pseudo-random test.

2. Methods for Calculating Weights

Many researchers have tackled the problem of generating efficient weights for a pseudo-random test. This has resulted in multiple algorithms that have a different approach for deriving these weights. An overview of some of the weight generation algorithms is presented in this chapter.

2.1. Bit Fixing

There are multiple ways of deriving weighted patterns to test circuits. Since random pattern resistant faults require very specific combinations on certain inputs then one approach would be to fix these bits to the required values and let the other bits be pseudo random. In this way you would get the mixture of the two worlds. You would get weight sets that are able to test multiple patterns thus minimizing the amount of memory needed compared to a deterministic test. At the same time the patterns you would generate from these seeds would be able to detect random pattern resistant faults faster than a pure pseudorandom test would.

Because the initial fault coverage of the pseudorandom tests is high then these methods start of by having random bits on all of the inputs. Once this fully random set of weights is unable to improve the fault coverage over a certain number of test vectors a new set of weights is generated. Usually by that time the fault coverage is already high and all the faults that are left are most considered random pattern resistant. The next set of weights will have some of its bits fixed to constant values. The inputs that are to be fixed are determined by deterministic vectors that are able to test the remaining faults.

A method is proposed in [2] to generate weight sets. At the start a deterministic test set T is generated for the CUT that covers the set all the testable faults F. The measure of random inputs in a weight set K is set to the number of inputs. Each weight set is set to run for N vectors. All the faults that are detected by a weight set are removed from F. If F is not empty then K is decremented and a new weight set is generated. This process continues until F is empty. Since K is reduced on every iteration it is guaranteed that all the faults in a circuit are detected as in the worst case a weight set will become a deterministic vector.

Weight sets with K random inputs are created by intersecting the vectors of the deterministic test set. First the remaining faults are ranked based on how many times they are detected in the deterministic test set. The fault that is detected by the least amount of vectors is selected. A vector is chosen that is able to detect the largest amount of faults while also detecting the chosen fault. After that the fault is marked as checked. This vector is intersected with the vector that covers the next unchecked fault. Intersection u of two vectors s and t are performed as follows. $If\ t(i) = s(i)\ then\ u(i) = t(i)\ else\ u(i) = "X"$. Where "$X$" denotes that the bit position is random. Vectors are selected until the amount of vectors reaches K.

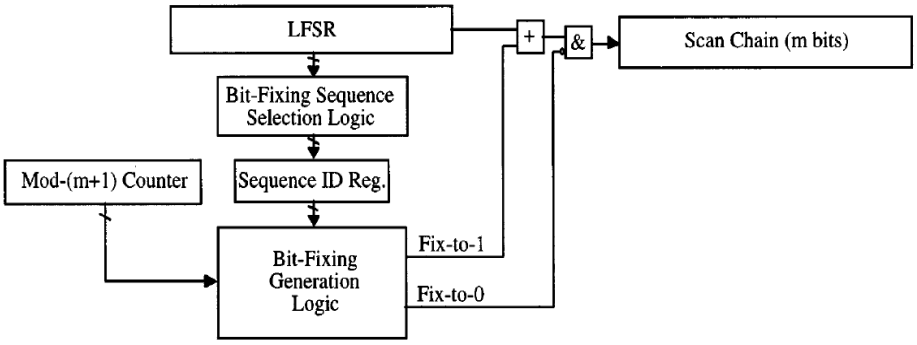

Figure 4 Implementation of a bit fixing generator[3]

2.2. Sample rate increasing method

Another approach for generating weights for a pseudo-random test is to embed deterministic test vectors inside the weights[4]. Weights are selected in a matter that would increase the likelihood of deterministic vectors generated in the test set. This method usually ends up using multiple sets of weights. Only vectors that would have a high sample rate are selected to be embedded to a single set of weights.

$$w_i = \frac{\left|\{t_j \in T | t_j[i] = 1\}\right|}{\left|\{t_j \in T | t_j[i] \neq X\}\right|} \quad (1)$$

First a deterministic pattern generator creates a set of test cubes for the CUT. Let us consider a circuit that can be tested by 10 test cubes presented in Table 1. A test cube is a three valued vector of constants “1” and “0” and “X” which symbolizes don’t care. The weight w_i of an input i is calculated as the average of all the test vectors t_j that have

a specified value in test set T (1). Don't cares in input positions are ignored as they do not have any bias towards either weight value (0 or 1).

$$P_j = \prod_{t=1, t_j[i] \neq X}^{m} \{(w_i \times t_j[i]) + (1 - w_i) \times (1 - t_j[i])\} \quad (2)$$

Once the weights have been calculated for all of the inputs, sampling probabilities P_j based on these weights are calculated for all of the test cubes. Only the test cube positions that have a constant value are considered in this calculation.

Table 1 Initial weights

Vectors	**Inputs**					**Sample Rate**	
	I1	**I2**	**I3**	**I4**	**I5**	**Weighted**	**LFSR**
T1	0	0	1	X	X	0.188	0.125
T2	0	1	0	X	X	0.156	0.125
T3	X	1	X	0	0	0.104	0.125
T4	X	1	X	0	1	0.208	0.125
T5	X	X	0	1	1	0.111	0.125
T6	X	0	1	1	X	0.125	0.125
T7	X	1	X	0	1	0.208	0.125
T8	1	0	1	X	X	0.063	0.125
T9	X	X	1	1	1	0.222	0.125
T10	0	1	X	X	0	0.156	0.125
W_i:	**0.250**	**0.625**	**0.667**	**0.500**	**0.667**	**-**	

If the sampling probability of a test cube would be greater with an LFSR (w_i = 0.5) then this cube is flagged as unused. The purpose of this step is to filter out conflicting weights. A single set of weights is not optimal for all the vectors of the deterministic test set. Vectors that are hard to generate with these weights are therefore removed. By

doing this we have selected the candidate list of vectors that are likely to be generated with these weights. The weights are then recalculated without the flagged vectors. As can be seen in Table 2, this step increases the likelihood of generating the remaining deterministic vectors even further.

Table 2 Improved Weights

Vectors	**Inputs**					**Sample Rate**	
	I1	**I2**	**I3**	**I4**	**I5**	**Weighted**	**LFSR**
T1	0	0	1	X	X	0.250	0.125
T2	0	1	0	X	X	0.167	0.125
T4	X	1	X	0	1	0.250	0.125
T6	X	0	1	1	X	0.125	0.125
T7	X	1	X	0	1	0.250	0.125
T9	X	X	1	1	1	0.281	0.125
T10	0	1	X	X	0	0.167	0.125
Weights:	0.000	0.667	0.750	0.500	0.750	-	

The quality of the weight set is measured by calculating the variance of the sample rates for all the used vectors. By reducing the variance of the weight set, we are equalizing the sample probabilities of the test vectors that were used for calculating the weights. A correction measure E_i can be calculated for each of the weights (3). Where A is the average sample rate calculated over all selected vectors.

$$E_i = \sum_{t=1, t_j[i] \neq X}^{m} \left\{ \left((A - P_j) \times t_j[i] \right) + \left(P_j - A \right) \times \left(1 - t_j[i] \right) \right\} \quad (3)$$

Correction rate α is used on the calculated weights to reduce the overall variance of the weight set. Weights that have a positive E_j are multiplied by α whereas weights with negative E_j are divided by α.

Finally the generated weights are rounded to the closest weight of the weight generator. The weights used in this method are 1/16, 1/8, 1/4, 3/8, 1/2, 5/8, 3/4, 7/8 and 15/16.

2.3. Structural Analysis

Another approach would be to analyze the structure of the CUT and tamper with the input probabilities of each input to minimize the amount of random patterns required to test the circuit.

In order to test a 16 input AND gate only 17 patterns are needed. Sixteen patterns will have only one of its bits as "0" to test all the SA-1 faults for the inputs and a single vector of ones to test the output SA-0. As it can be seen a high number of ones are required to test this circuit. The probability to test a single fault for that gate with an equiprobable random pattern generator is around $1.52*10^{-5}$. However if would change the probability of generating a "1" for all the bits to the probability of 0.95 then the probability of detecting a SA-1 fault in one of the inputs becomes $2.31*10^{-2}$. As it can be seen the probability of detecting a fault has increased by an order of magnitudes. The same effect can be observed with an OR gate while increasing the probability of zeroes generated for the inputs by the pattern generator. From this we can derive that increasing the probability of its Non-Controlling Value (NCV) would be advantageous for improving the quality of a pseudo-random test[5].

Based on these ideas a method was proposed where the bias of each gate would be propagated to the circuit inputs to derive the best set of weights for the circuit. This however is a more difficult task as not only the faults of the gate itself should be taken to account but also all the faults that are passing through it. *Generally, the more faults that must be test through a gate input, the more the other inputs should be weighted towards the NCV*[5]. In order to measure how many faults are detected through a gate a parameter for each gate is created called Number of Device Inputs (NDI). NDI is a measure that shows how many signals from the device inputs propagated through this gate. For each input of the gate the ratio R_i can be calculated by dividing the NDI of the gate with the NDI of the input (4). R_i is the measure of how many times more the NCV has to be applied to the input compared to its opposite value.

$$R_i = NDI_g \,/\, NDI_i \tag{4}$$

2.3.1. Global Weight Calculation Algorithm

In this chapter a weight generation algorithm is presented which was described in [5]. An example of how the weights are calculated through out the circuit can be seen in Figure 5.

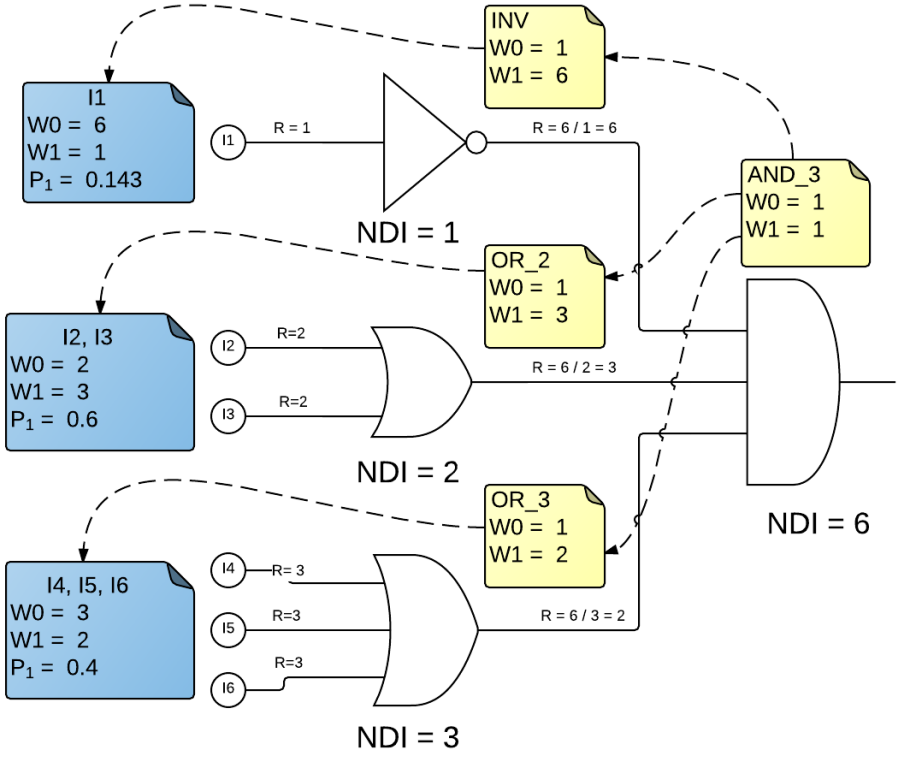

Figure 5 Weight Calculation Example

Algorithm:

- Define NDI for all gates of the circuit.
- Assign two numbers W0 and W1 to each logic gate and input pin and initialize both of them to 1.
- Perform a backtrace from each output of the circuit.
    - While moving backwards from gate *g* to gate *i* calculate $W0_i$ and $W1_i$ based on the weight propagation formulas from Table 3. The new value of W_i is the maximum of the previous and the calculated value.
- Determine the following values for each circuit input:

- Weighted Value (WV) which shows which logical value the input is biased towards. The value is either zero or one based on witch of the two weights is greater.
- Weight Factor (WF) this is the ratio between the two weights. It is a measure to show how much biasing is needed for the given input. It is calculated by dividing the larger of the two with the smaller. The WF is later used to find the closest weight of the generator.

Table 3 Formulas for calculating weights

Function of *g*	**$W0_i$**	**$W1_i$**
AND	$W0_g$	$R_i * W1_g$
OR	$R_i * W0_g$	$W1_g$
NAND	$W0_g$	$R_i * W0_g$
NOR	$R_i * W1_g$	$W0_g$

3. Generation of Weighted Patterns

One of the main problems of weighted pattern generation is finding an efficient way of generating the weights. The problem is that multiple shift registers are required to generate weighted inputs that do not have any correlation with other inputs. Therefore having more weighted inputs for a test-per-clock type of test method would drastically increase the area overhead of the test circuitry. Also having different kind of weight cells means that the more effort needs to be directed to designing the test apparatus. This obviously has an impact on the overall testing cost. An example of target device independent approach would be BILBO that uses uniform cells in its implementation.

Typically a test-per-scan implementation, like STUMPS, is used, where the bits are shifted to a boundary scan circuit one by one[6]. First the test vector is shifted into the scan chain from the pattern generator, the CUT is run for a single clock cycle and finally the response is shifted out to the signature analyzer. This is considerably slower than the test-per-clock method where the responses are stored on every clock cycle.

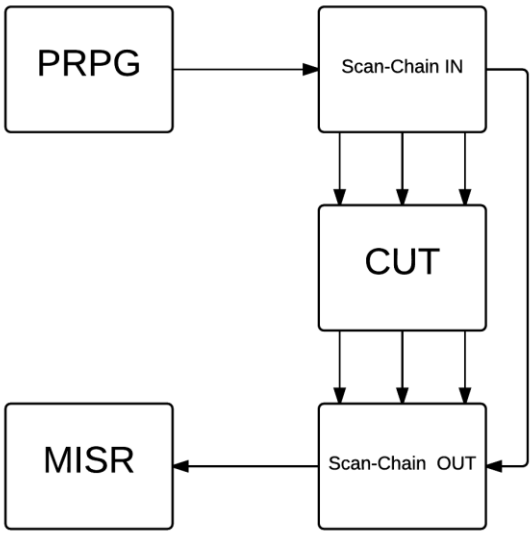

Figure 6 STUMPS architecture

Typical weighted PRPG designs consist of an LFSR to generate random bit patterns, some weighing logic to modify the generated series of bits and an input counter to control which weight is applied to the current weight.

A weight generator design is proposed in [5]. It consists of an LFSR of arbitrary length. An LFSR has a probability of 0.5 to generate a "1" in any of its inputs. By combining the last two elements of an LFSR with an AND gate we are able to generate a weight of 0.25. If we use another AND gate and combine the previous AND gate and another LFSR register then we are able to generate the weight of 0.125. By using this pattern any weight that is a factor of 0.5 can be created. The desired weight will later be selected by a MUX block which is controlled by the input decoder. All of the weights created before have a bias towards creating a "0". An XOR gate is added to the output in order to also create weights that have a probability of creating a "1" higher than 0.5.

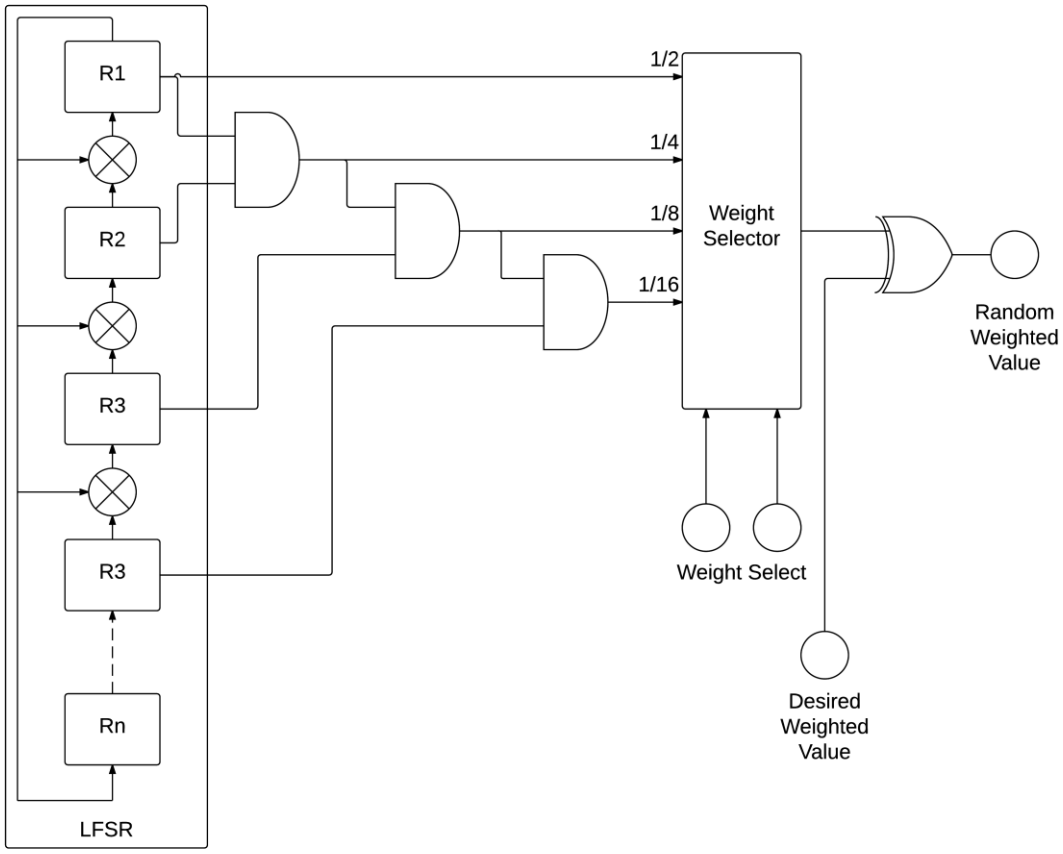

Figure 7 Implementation of a weight generator

It is important to note that this design has a flaw. For example let us use the weight generator configuration used in Figure 7. Let us assume that we wanted to generate a weight of 0.25 and the generator was able to create value "1". For this to happen both of the registers R1 and R2 need to be "1". Now let us try to generate a weighted value with a weight of 0.5. Due to the feedbacks of the LFSR used in this configuration there is no

possible way to generate value “1” therefore a combination of “…11…” can never be created for these weights.

In order to prevent correlation between consecutive generated values it is important to shift out the content from the registers that were used to produce that weight. Therefore every weight of 0.5 is followed by a single shift and a weight of 0.25 must be followed by two shifts, etc. This means that this generator is unable to generate a weighted value on every clock cycle. This means that the process of generating test vectors can dramatically increase if the number of heavily weighted inputs is high.

4. Simulator for Weighted Pseudo Random Generator

4.1. Overview

The purpose of the simulator analyzes the structure of a combinational circuit and creates weights for its inputs and then generates test vectors based on these weights. The algorithm described in paragraph 2.3.1 was used to generate the weights. This algorithm was chosen as it focuses on a strategy for generating a single set of weights. As mentioned in the previous chapter weighted testing will always be slower and have a bigger area overhead than regular pseudo-random testing. If using a single set of weights proves to be effective enough it would mean that additional chip area would not have to be used to store the additional weight sets.

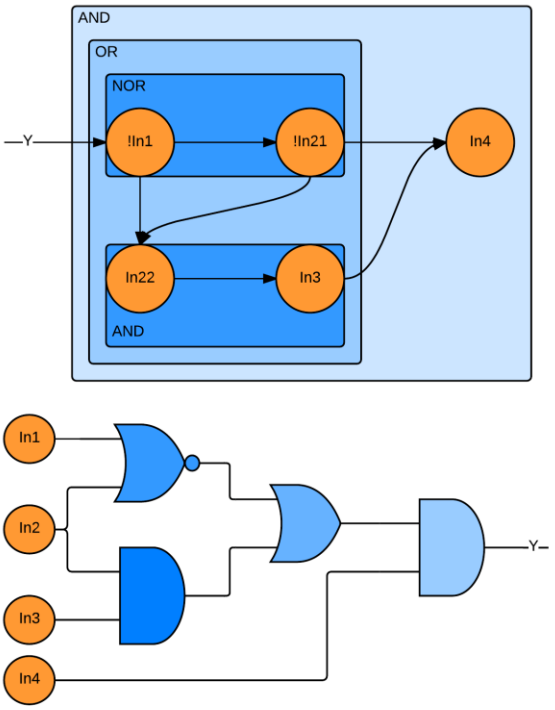

Figure 8 SSBDD and its equivalent circuit

The models for the simulator are described using the ".agm" format common for the Turbo Tester Toolset[7]. This format is special as the circuit is described by a series of SSBDD-s.

4.2. Structure

The simulator takes a model described in an ".agm" file as an input and then generates a test set of weighted vectors for it and stores the fault simulation results of these vectors to a ".tst" file. The process is broken into the following steps.

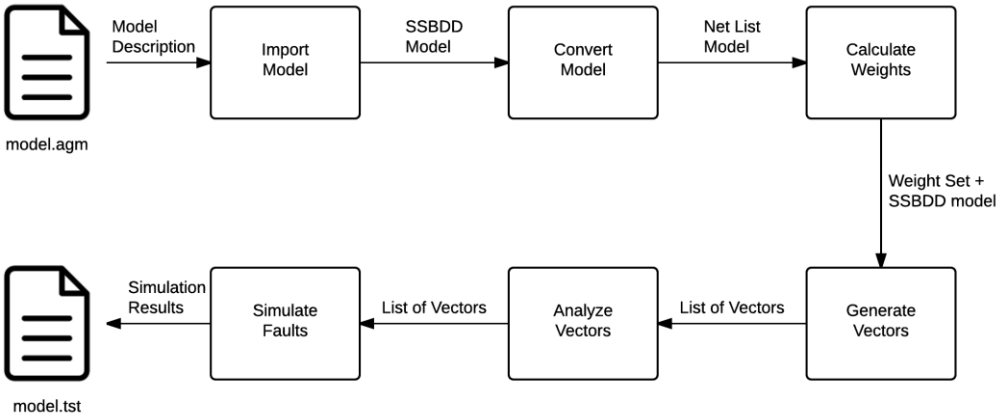

Figure 9 Workflow of the simulator

In the first step the ".agm" file is parsed. This file describes the CUT and it is used to create a local SSBDD model based on it. The SSBDD model is then passed to the model converter.

The model converter takes the SSBDD model and creates an equivalent gate level net list for it. This step is needed as the algorithm for deriving the weights operates on standard gates but the model itself is described by graphs. The ".agm" file can also be created with a gate level option, where each graph describes a single gate. However the advantage of using macro level graphs is a significant speed up during fault simulation. A detailed description of converting SSBDD graphs into a gate level structure is presented in paragraph 4.2.1. After all the graphs have been converted to the net list representation all of the net lists are merged to form a flat gate level structure. As a result a net list model is created which is passed onto the weight calculator.

The net list model is then passed on to the weight calculator which uses the weight generation method [5] described in paragraph 2.3.1 to generate the weights which would reduce the time of the pseudo random test. If the -no_weights option is selected then this step is passed and weights of 0.5 will be used for all of the inputs in the pattern generator. The generated weights can also be overridden by the user using the –modify option. To overide the weight of a specific input the user must supply the correct input number and weight pairs after –modify switch. As a result the weight set for the model is created.

This weight set is then sent to the pattern generator. In the pattern generator an LFSR is constructed based on the number of registers and weights it is supposed to generate. For up to the first 32 bits the LFSR will use primitive polynomials described in [8]. If the number of feedbacks is greater than 32 the primitive will use random polynomials. The generator will then create the number of test vectors specified. The vectors will then be printed to a ".tst" file. The weight generator will use the weight which has the closest input probability to it.

These vectors are then analysed if the "–report" flag is used. The report contains statistics for the frequency of ones generated by the weight generator for each input. For each input the report will contain the following values:

- The probability of the input calculated by the algorithm.
- The probability of used in the weight generator.
- The actual frequency of ones created by the generator in the test set.

Finally the created vectors are stored to the „.tst" file and the Analyze tool of the Turbo Tester toolset will be used on it to perform fault simulation.

4.2.1. Graph conversion algorithm

In order to convert SSBDD graphs back to standard gates lets analyze how they are first constructed. A BDD is called SSBDD, if there is a one-to-one correspondence between non-terminal nodes of the BDD and signal paths in the combinational circuit[9]. The non-terminal nodes of a SSBDD correspond to the input signals of the described combinational circuit. Typically SSBDD are drawn in a way where the "1" path of a node exits to the right and the "0" path exits to the bottom. Each standard gate is converted to its SSBDD equivalent which in terms of AND and NOR gates results in a

series of nodes running from left to right and in the case of OR and NAND gates in a series of nodes running from top down. The graphs themselves are constructed through the use of a graph superposition procedure to preserve the gate-level description. An example of a SSBDD constructed for a circuit is shown in Figure 8.

The algorithm:

1. Initialize the function for all the nodes of the graph to the input term of the node.
2. Push all the nodes of the graph into a queue.
3. Repeat the following until only one node is left in the queue
   3.1. Pop a node *n* from the queue.
   3.2. Check if n and its right neighbor *m* have a common down neighbor and that m is only referenced by *n*.
      3.2.1. If true join the functions of *n* and *m* with an AND function and set it as the new function of *n*.
      3.2.2. Remove *m* from the queue.
   3.3. Check if *n* and its down neighbor *m* have a common right neighbor and that *m* is only referenced by *n*.
      3.3.1. If true create a new node which function is the OR function of the original nodes functions
      3.3.2. Remove the down neighbor from the queue.
   3.4. Push node *n* back to the queue.

The implementation of the algorithm can be seen in Appendix 1.

4.3. Commands

Here is the syntax for using the simulator and possible switches that can be used.

Syntax: wprg [options] <design>

Design:	Name of the design file without the .agm extension
Options:	
-report	Displays a report of the weight information for all of the inputs. The report shows the calculated probability, the probability of the generators output, and measured

	frequency of ones generated in the generated patterns.
-vectors <amount>	Specifies how many vectors are to be generated. The default value is 1000.
-no_weights	Uses the 0.5 weights for all of the inputs.
-glen <length>	Specifies the number of stages of the LFSR used in the generator. The default value is 32. The generator uses primitive polynomials in the LFSR for lengths up to 32. After that random polynomials are used.
-weights <size>	The number of output stages of the LFSR generator.
-modify [<input> <probability>]	Sets the weight of the specified inputs to the desired probability. This probability is later rounded to the nearest generator output.

5. Experimental Results

The simulator described in the previous chapter was used to compare its effectiveness to standard pseudo-random testing. Deterministic methods were also used to show the maximum coverage that could be achieved on the tested circuit. Circuits that did not reach 100% coverage contained untestable faults in the circuits which were caused by redundancies in the circuits. The following ISCAS85 circuits were used c432, c880 and c5315. Also small sample circuits T1-4 (See Appendix 2) were used to illustrate the possible effects could have on a circuit.

Table 4 Test results

Circuit	Method	Average Vectors	Covered Faults	Total Faults	Coverage %	Reduction in test length
T1	Equiprobable	132.4	16	16	100	72.4%
	Weighted	36.6	16	16	100	
	Deterministic	8	16	16	100	
T2	Equiprobable	41	18	18	100	14.6%
	Weighted	35	18	18	100	
	Deterministic	8	18	18	100	
T3	Equiprobable	35.5	42	42	100	-7.04%
	Weighted	38	42	42	100	
	Deterministic	9	42	42	100	
T4	Equiprobable	445	28	28	100	86.1%
	Weighted	61.9	28	28	100	
	Deterministic	13	28	28	100	
c880	Equiprobable	13014.4	994	994	100	60.2%
	Weighted	5178.2	994	994	100	
	Deterministic	43	994	994	100	
c5315	Equiprobable	3102	5364	5424	98.894	15.7%
	Weighted	2615.4	5364	5424	98.894	
	Deterministic	104	5364	5424	98.894	
c432	Equiprobable	2036.2	573	616	93.019	-391%

	Weighted	10000	558	616	90.58442	
	Deterministic	45	573	616	93.019	

From the results it is possible to see that the weighted pseudo-random vectors have a clear advantage when applied to tree shaped circuits like T1. However circuits, T2 and T3, that have fan-outs which drive the weights in different directions in equal magnitude performed as well as the standard PRPG. If the magnitudes remain different like in T4 the weighted method seems to perform better again. The magnitudes of T4 are different because the fan-out is applied to an OR gate which offsets the weights from its successors once again. Balanced units like invertors or XOR gates propagate weights to their predecessors equally.

It is also important to note that even though the weights help increase the detection of some the weights it also starts decreasing the probability of others. The maximum effect that can be gained by weighing an input in the correct direction can reduce the test length to about two times. Whereas selecting the opposite weight can increase the test length up to 8 times[5]. If the initial probability of detecting those faults is high it has no effect on the test overall test length, however in worst case scenarios it is possible that it takes more time for weighted generator to cover all of the faults. This would explain why it was not possible to reach the max coverage (93.019%) with the calculated weights for c432.

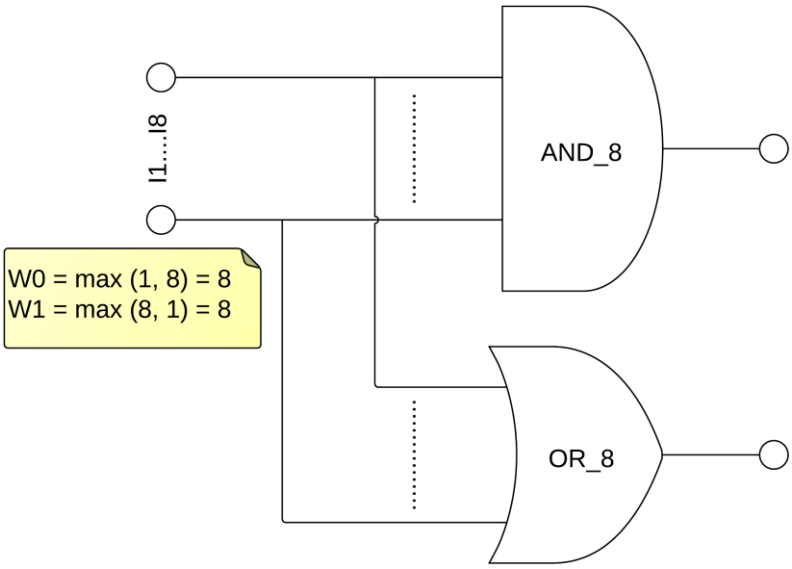

Figure 10 Weight conflict example

A good example of the effect cancelling weights have can be seen in a circuit shown in Figure 10. The circuit consists of an eight input AND gate and an eight input OR gate. If weights for these gates would have been calculated individually the algorithm would create weights for testing the AND gate to be W_0=1 and W_1=8 and W_0=8 and W_1=1 for the OR gate. However since only the maximum for each weight is considered in case of conflicts a weight of 0.5 will be assigned for each input. It would actually be faster to test this circuit by having two sets of weights. Having a weight of 7/8 will increase the likelihood of testing the AND gate and a weight of 1/8 will help test the OR gate. As can be seen in Table 5 using two sets of weights significantly enhanced the quality of the test. Even though in this example multiple weights are used, the probabilities for each input still average to around 0.5 over the entire data set.

Table 5 Use of multiple weights

Method	**Weights**	**Vectors**	**Coverage**
Single Weight	{0.5}	128	54.583%
Multiple Weights	{0.125, 0.875}	64+64	97.292%

In addition to two sets of weights canceling each other out, one set of weights can completely dominate the other. Let us consider the calculation of weights for the given input in Figure 11. The input is connected to three inverter gates which are connected to some sort of sea of gates. Inverters were chosen in this example because of their easy transfer functions. The ratio of inputs R_i is always 1 for them. The top inverter has the weight ratio of 10 / 30. This means that the optimal input probability for that input would be p_1 = 0.25. The bottom two inverters have the weight ratio of 7 / 1. For these paths the optimal input probability would be p_1 = 0.875. The algorithm chooses the maximum for both weights when resolving conflicts. This means that the final input probability for the input would be p_1 = 0.25. This input probability would favor detecting faults originating from the top branch as the input probability and the probability required for the branch are the same. However this significantly reduces the probability of detecting faults from the bottom two branches as the bottom branches require a high probability of generating ones. Therefore the selected weights could actually increase the time it takes to test the circuit.

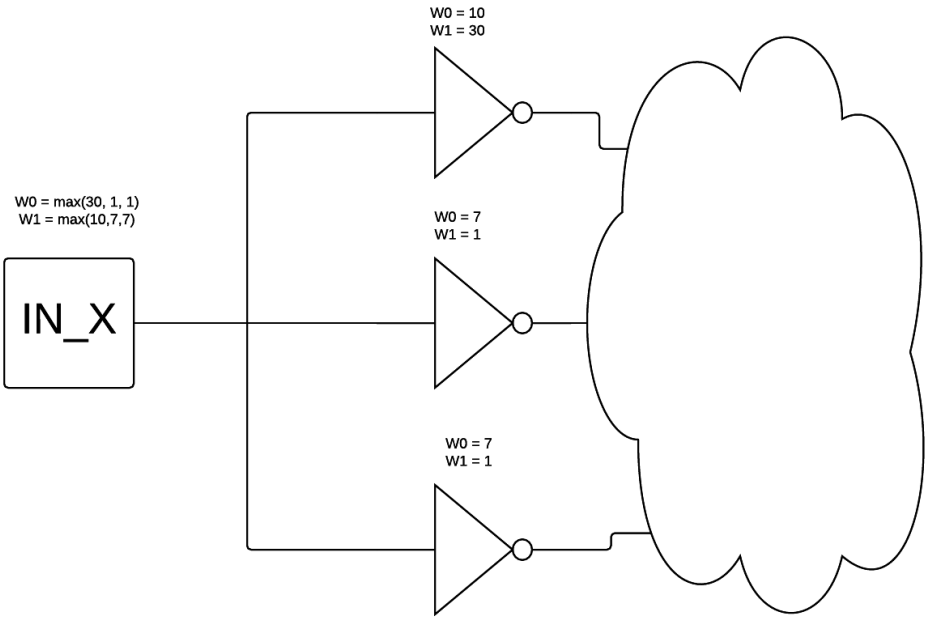

Figure 11 Calculation of weights with conflicts

6. A Possible Lab Exercise

From the experimental results it is possible to see that the weighted testing method can have mixed results depending on the complexity and structure. Studying these effects could prove to be an interesting lab exercise. The goal of the lab exercise would be to teach the students about how the changes to the probabilities for each input of the circuit can have on the test length. The task of the lab exercise would consist of two parts.

In the first part the student is given a small circuit, about 8 inputs. At first the student runs a standard pseudo-random test on the circuit. After that the student is tasked to make more experiments on the circuit again with different weight sets. The simulator created for this thesis will be used to carry out the experiments. The student can use the structure based algorithm explained in the lectures, try random weights or come up with his own strategy of deriving the weights. The goal would be either to reduce the test length by a given percentage or to pick weights that actually increase the test length. In the end the student writes a small report on how the changes to the input probabilities affected the length of the pseudo-random test.

In the second part of the lab exercise the student carries out experiments out on a few larger circuits. The ISCAS85 benchmark circuits could be used. The student then runs multiple tests with different seeds using both the standard pseudo-random generator and the weighted generator in the structure analysis mode. In the end the student writes a report on the results of the two methods compares their differences and evaluates the performance of the weighted algorithm on different sized circuits.

An example of a report that students would have to fill during this lab can be seen in Appendix 3.

7. Conclusion

As the result of this thesis a summary of different weighted pseudo-random testing methods was created. The key strengths and weaknesses of using weighted generation methods were analyzed. A tool was created to simulate process of using one of the discussed methods for testing a circuit. Experiments were made to compare the WPRPG against a pure PRPG. From the results it was possible to see that in tree structured circuits the test lengths for WPRPG were smaller in order of magnitudes. However in more complex circuits the performance of a WPRPG was similar to a standard PRPG and in some cases even worse.

As mentioned in some of the works on weighted test generation[5][4], a single set of weights is simply not enough to have strictly better performance over non-weighted testing. An experiment was carried which showed that in certain configuration having multiple weight sets is more beneficial than having a single set of weights where conflicting weights have evened the input probability to 0.5.

However it is important to note that the performance here was measured in the number of vectors, not in time. As discussed earlier WPRPG methods are limited to using test-per-scan methods as no area efficient methods exist for test-per-clock implementations. These implementations are available for standard pseudo-random testing. Therefore weighted methods need to be significantly better than any unweighted generation method to compensate for the lack of a parallel implementation.

However weighted testing methods should not be discarded. As a completely different approach to pseudo-random testing, they still have academic value. A lab exercise was proposed in this thesis to teach and familiarize students with weighted testing methods. In these lab exercises students would use the simulator created for this thesis to compare the weighted PRPG methods to the standard one. The students are also encouraged to modify the weights themselves to and discover how the selection of weights can either decrease or increase the overall test length.

References

[1] H.-J. Wunderlich, “Multiple distributions for biased random test patterns,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 9, no. 6, pp. 584–593, 1990.

[2] I. Pomeranz and S. M. Reddy, “3-Weight Pseudo-Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 12, no. 7, pp. 1050–1058, 1993.

[3] N. a. Touba and E. J. McCluskey, “Bit-fixing in pseudorandom sequences for scan BIST,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 20, no. 4, pp. 545–555, 2001.

[4] H.-S. K. H.-S. Kirn, J. L. J. Lee, and S. K. S. Kang, “A new multiple weight set calculation algorithm,” *Proc. Int. Test Conf. 2001 (Cat. No.01CH37260)*, pp. 878–884, 2001.

[5] E. B. Eichelberger, E. Lindbroom, J. A. Waicukauski, and T. W. Williams, “Weighted Random Patterns,” in *Structured Logic Testing*, Englewood Cliffs, New Jersey: Prentice-Hall, 1991, pp. 136–155.

[6] “ECE 553: TESTING AND TESTABLE DESIGN OF DIGITAL SYSTEMS.” [Online]. Available: http://homepages.cae.wisc.edu/~ece553/handouts/slides/Lecture_19.pdf. [Accessed: 02-May-2015].

[7] G. Jervan, A. Markus, P. Paomets, J. Raik, and R. Ubar, “A CAD System for Teaching Digital Test,” in *Proc. of the 2nd European Workshop on Microelectronics Education*, 1998, pp. 287–290.

[8] R. Ward and T. Molteno, “Table of Linear Feedback Shift Registers,” Dunedin, New Zealand, 2007.

[9] A. Jutman, J. Raik, and R. Ubar, “SSBDDs: Advantageous model and efficient algorithms for digital circuit modeling, simulation & test,” *Proc. 5th Int. Work. Boolean …*, 2002.

Appendix 1 – Implementation for the graph conversion algorithm

```
private static NLModel convertGraph(SSBDDGraph graph){

    //  create a copy of all the nodes for reducing graph nodes to gates.
    LinkedList<SSBDDNode> nodes = copyNodes(graph.getNodes());

    // create initial NLModels for nodes. These are simple models that
    // consist of a NLInput & NLOutput & NLSignal joining them.
    for (SSBDDNode node : nodes){
        node.initNLModel();
        if (node.getRight() != null) node.getRight().incRefCount();
        if (node.getDown() != null) node.getDown().incRefCount();
    }

    while (nodes.size() > 1){
        SSBDDNode node = nodes.pop();

        boolean canMerge = true;

        while (canMerge) {
            if (node.getRight() != null &&
               node.getRight().getDown() == node.getDown() &&
               node.getRight().getRefCount() == 1){

                // remove the right node from the list
                SSBDDNode right = node.getRight();
                nodes.remove(right);

                // merge the functions of the two nodes
                SSBDDNode joinNode = mergeNodes(node,
                                                right,
                                                NLGate.GateType.AND);

                // change the references from the old node to the new merged
                // node
                for (SSBDDNode n : nodes){
                    if (node.equals(n.getDown())) n.setDown(joinNode);
                    if (node.equals(n.getRight())) n.setRight(joinNode);
                }

                //  reduce the amount of references to the node
                if (node.getDown() != null) node.getDown().decRefCount();

                node = joinNode;
            }
```

```
            else if ((node.getDown() != null) &&
                    (node.getRight() == node.getDown().getRight()) &&
                     node.getDown().getRefCount() == 1){

                // remove the down node from the list
                SSBDDNode down = node.getDown();
                nodes.remove(down);

                // merge the functions of the two nodes
                SSBDDNode joinNode = mergeNodes(node, down,
                NLGate.GateType.OR);

                // change the references from the old node to the new merged
                // node
                for (SSBDDNode n : nodes){
                    if (node.equals(n.getDown())) n.setDown(joinNode);
                    if (node.equals(n.getRight())) n.setRight(joinNode);
                }

                //  reduce the amount of references to the node
                if (node.getRight() != null) node.getRight().decRefCount();

                node = joinNode;
            }else {

                // add this node to the end of the list and start processing
                // a new node.
                nodes.addLast(node);
                canMerge = false;
            }
        }
    }

    NLModel res = nodes.getFirst().getNetList();
    createInversions(res);

    res.getOutputs().getFirst().setVariable(graph.getVariable());

    return res;
}
```

Appendix 2 – Custom Circuits used in experiments

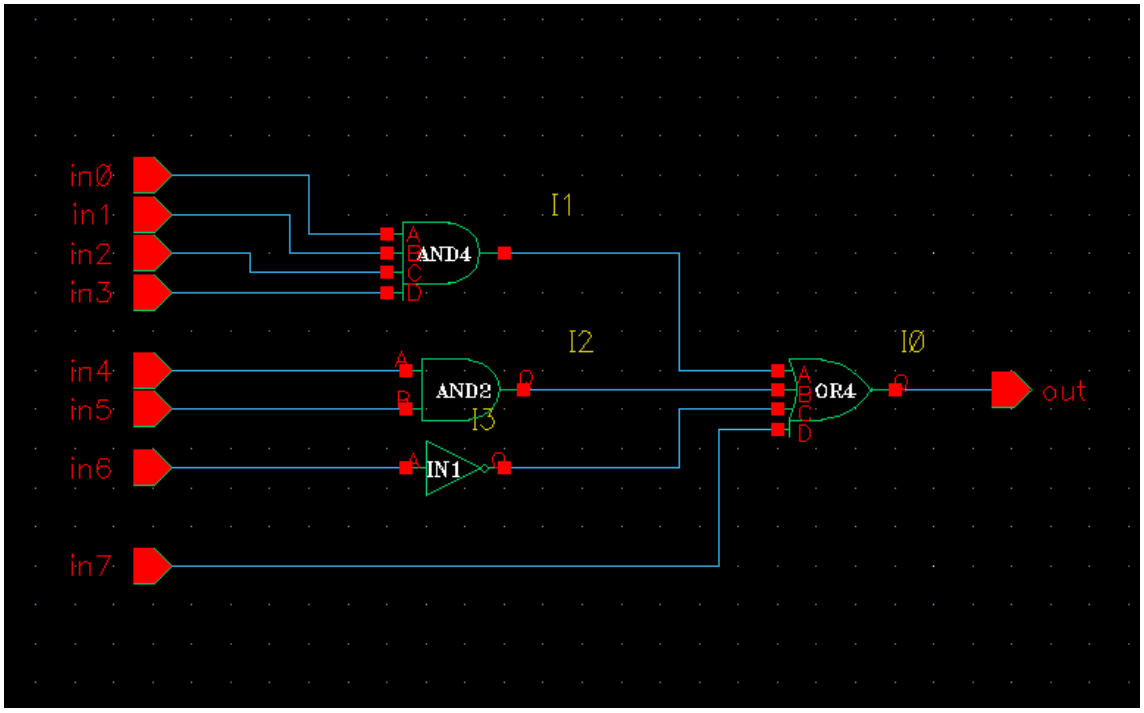

Figure 12 Schematic T1

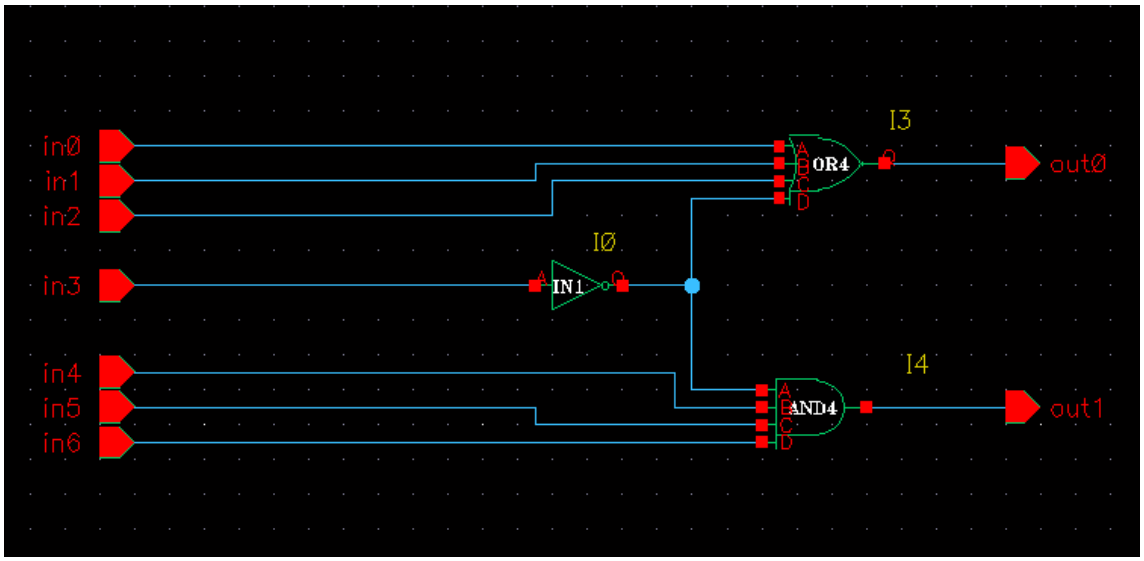

Figure 13 Schematic T2

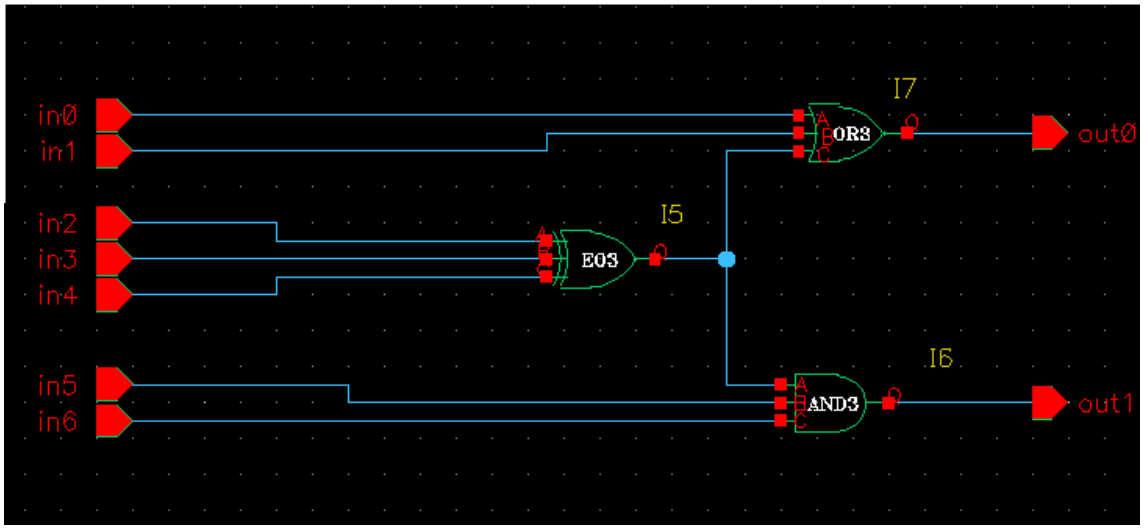

Figure 14 Schematic T3

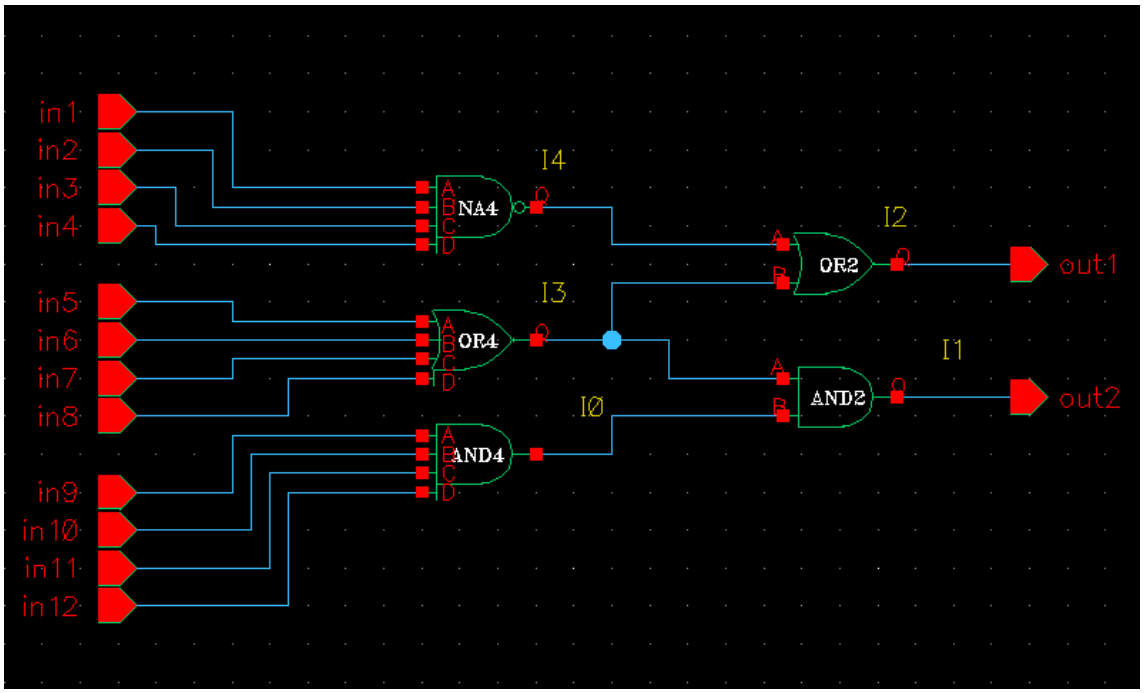

Figure 15 Schematic T4

Appendix 3 – Lab Exercise: Weighted Pseudo-random Testing

Name: _____

Student code: _____

Version: _____

Task

In this lab you are to carry out pseudo-random tests with different sets of weights in order to learn how changes to the input probabilities of the circuit can affect the overall test length. For the first experiment use the WPRG with an equiprobable weight set.

wprg <design_name> -no_weights

As the second experiment use the weights calculated by the structure based algorithm that was described in the lectures.

wprg <design_name>

In the following experiments test the circuit by manually changing some of the inputs.

wprg_sim <design_name> -modify {<Input_number> <input_probability>}

Experiment by increasing the offset on some of the calculated weights. Also try inverting the values of the generated weights. See if you are able pick better weights than the calculated by the weight calculation algorithm used in the simulator. Also make a few counter examples where you try to pick weights in a way that would actually increase the test length

In the end write a conclusion on how the changes to the input probabilities affected the test length. Also write about your strategy on picking the weights in order to reduce or increase the test length.

Experiments

No	Input probabilities									Fault coverage	Test Length
	I1	I2	I3	I4	I5	I5	I6	I7	I8		
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											

Conclusion

Carry out experiments on two of the given ISCAS85 benchmark circuits with both the weighted pseudo-random test method and the standard pseudo-random test which uses equiprobable weights. First calculate the maximum fault coverage for both the circuits. Then carry out ten experiments using both of the methods. Insert the number of vectors it was need to reach maximum fault coverage on each test. Calculate the average test length for both of the methods. In the end write a small summary on the results of the experiments and explain how changing the weights affected the test length.

No	**Circuit _____**	**Fault coverage ___**	**Circuit _____**	**Fault coverage ___**
	Equiprobable	**Weighted**	**Equiprobable**	**Weighted**
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
Avg				

Conclusion