# Weighted Pseudo-Random Test Pattern Generation

*Under the guidance of*

**Prof. Indranil Sengupta**

Department of Computer Science and Engineering,

Indian Institute of Technology Kharagpur

*Submitted by*

**Siddharth Seth (02EC1032)**

**Gaurav Sahni (02CS1033)**

Indian Institute of Technology Kharagpur

## ACKNOWLEDGEMENTS

It is with great reverence that we wish to express our deep gratitude towards our VLSI Testing and Verification instructor, **Professor Indranil Sengupta,** Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, under whose supervision we completed our work. His astute guidance and invaluable suggestions, in spite of the fact that he was busy with other pressing assignments has been a great motivating factor all throughout our work.

We would be accused of ingratitude if we failed to mention the consistent guidance and help extended by Mr. Shibaji and Mr. Santosh Biswas, Graduate Research Assistants, during our Term-Project work. The brainstorming sessions at AVDL spent discussing various possible approaches for our system were very educative for us.

Our experience in working together, given that our departmental backgrounds were different, was wonderful. We hope that the knowledge, practical and theoretical, that we have gained through this term project will help us in our future endeavors.

Siddharth Seth
Gaurav Sahni

## 1. Problem Statement

Weighted Pseudo-random Test Pattern Generation: Write a program that will read a circuit in ISCAS-85 format, and calculate the weights of the signal lines, starting with 0.5 at the primary outputs. It will then carry out fault simulation using a LFSR based weighted pseudo-random pattern generator, and will produce an output data file showing the variation of percentage fault coverage with the number of patterns.

## 2. Introduction

### a. Motivation and need for weighted pseudo random patterns

A digital system is tested and diagnosed during its lifetime on numerous occasions. Such a test and diagnosis should be quick and have very high fault coverage. One way to ensure this is to specify such a testing to as one of the system functions, so now it is called Built In Self Test (BIST). With properly designed BIST, the cost of added test hardware will be more than balanced by the benefits in terms of reliability and reduced maintenance cost.

For BIST, we would require that the test patterns be generated on the system/chip itself. However, this should be done keeping in mind that the additional hardware is minimized. One extreme is to use exhaustive testing using a counter and storing the results for each fault simulation at a place on the chip (in the form of ROM). An n input circuit would then require $2^n$ combinations which can be very tiresome on the system with respect to the space and the time. Also, more the number of transitions, the power consumed will be more. On the other extreme, we can use a completely deterministic set of test vectors. This would again require us to store the test patterns in a ROM and the chip. However, it is relatively easy to use a relatively better to use a middle approach wherein we use a pseudo random generator made using Linear Feedback Shift Register (LFSR). These patterns generated using LFSR have all the desirable properties of random numbers, but are algorithmically generated by the hardware pattern generator and are therefore repeatable, which is essential for BIST. We no longer cover all the $2^n$ combinations, but a large number of test pattern sequences will still be necessary to attain sufficient fault coverage.

In general, pseudo random pattern generation requires more patterns than completely deterministic Automatic Test Pattern Generation (ATPG), but obviously, fewer than the exhaustive testing.

However, it was found that the stuck-fault coverage rises in a logarithmic fashion towards 100%, but at the cost of enormous numbers of random patterns.

On top of it, certain circuits are random pattern resistant circuits in that they do not approach full fault coverage with an unbiased random pattern. Such circuits require extensive insertion of testability hardware or a modification of random pattern generation to ‘weighted pseudo random pattern generation’ in order to obtain an acceptable fault percentage.

This desire to achieve higher fault coverage with shorter test lengths and therefore shorter test times led to the invention of the weighted pseudo random pattern generator.

**b. Work described in this report**

In section 3 we discuss the ISCAS85 format and how we decipher relevant information about all the circuits from the format and what inferences can be drawn from the individual fields. In section 4 we show the data structure we have used while coding for the project and the platform that we have used. In section 5 we discuss how we read the ISCAS85 circuits and then form the corresponding .v files that will be used later in the Tetra Max tool while doing fault simulation. In section 6 we show how we compute the probabilities of the input lines and thus the weights by starting with signal probabilities of 0.5 at each of the output lines. In section 7 we finally show the design of the weight generating combinational circuit. It is here that we discuss the basics of the Linear Feedback Shift Registers (LFSR) and then discuss how we compute the length of the LFSR needed for each of the circuits and then how we decide upon the feedback taps that are used for each LFSR for both the weighted as well as the unbiased pseudo random pattern generation case. In section 8 we discuss how the entire system is run to generate the patterns and how the final .stil files are written to be used in Tetra Max tool for fault simulation. In section 9, the steps taken to perform the fault simulation in Tetra Max are enumerated. In section 10, the parameters and the computed fault coverage for each circuit for weighted pseudo random pattern generation as well as pseudo random pattern generation are given. We also discuss the limitations and the additional things that can be done as an improvement. Finally, in section 11, we give the references that were used during the course of the project.

## 3. ISCAS 85 Format

The ISCAS 85 benchmark circuits are a set of combinational circuits provided to authors at the 1985 International Symposium on Circuits and Systems. They subsequently have been used by many researchers as a basis for comparing results in the area of test generation.

For explaining the format we use a small six-NAND-gate circuit known as c17.isc, for which the netlist is as follows:

```
      *c17 iscas example (to test conversion program only)
*---------------------------------------------------
*
*
*  total number of lines in the netlist ..............    17
*  simplistically reduced equivalent fault set size =     22
*        lines from primary input  gates .......     5
*        lines from primary output gates .......     2
*        lines from interior gate outputs ......     4
*        lines from **     3 ** fanout stems ...     6
*
*        avg_fanin  =  2.00,     max_fanin  =  2
*        avg_fanout =  2.00,     max_fanout =  2
*
*
*
*
*
    1     1gat inpt    1   0      >sa1
    2     2gat inpt    1   0      >sa1
    3     3gat inpt    2   0 >sa0 >sa1
    8     8fan from     3gat      >sa1
    9     9fan from     3gat      >sa1
    6     6gat inpt    1   0      >sa1
    7     7gat inpt    1   0      >sa1
   10    10gat nand    1   2      >sa1
     1     8
   11    11gat nand    2   2 >sa0 >sa1
     9     6
   14    14fan from    11gat      >sa1
   15    15fan from    11gat      >sa1
   16    16gat nand    2   2 >sa0 >sa1
     2    14
   20    20fan from    16gat      >sa1
   21    21fan from    16gat      >sa1
   19    19gat nand    1   2      >sa1
    15     7
   22    22gat nand    0   2 >sa0 >sa1
    10    20
   23    23gat nand    0   2 >sa0 >sa1
    21    19
```

The valid delimiters between the fields of data are: spaces, horizontal tabs and new-lines. The definitions for each field starting from the left to the right for each line are:

- *address*: unique number that differentiates each node from the rest
- *name*: string of characters used to provide more meaningful information
- *type*: function performed by the gate driving this node. The legal node types are: inpt, and, nand, or, nor, xor, xnor, buff, not, from
- *fanout*: number of gates driven by this node

- *fanin*: number of nodes driving the gate that drives this node
- *faults(s)*: the stuck at fault(s) on this node that are included in fault set

This type of line is called a node line and provides the basic information for every node in the circuit. However there are two other types of line that may be associated with it. The *fanin* line provides a list of the addresses of the nodes that fan in to the gate driving this node. The *fanout* branch lines appear immediately after the *node* line and its *fanin* line with which it is associated.

## 4. Data Structures Used

The only data structure used for storing the iscas-85 net-list is a WIRE link-list. The nodes of the link-list are wires, inputs and outputs with details such as address, name, type, number of fan-in and fan-out, level, probability, a list of fan-outs (if they exist) etc.

```
typedef struct wire  {
     int addr;       /* address of the wire */
     char name[NAME_LEN]      /* name of the wire */
     char type[TYPE_LEN];     /* type of wire */
     int level;   /* wire level after levelization */
     float prob;  /* signal probability of the wire */
     short  fanin, fanout;/* number of fanout & fanin */
     struct fanlist   *flist;/* list of Inpts of this wire
*/
     struct wire  *fanParent;    /* pointer to parent if
this wire is a fanout */
     struct wire  *outGate;
     struct wire   *next; /* pointer to the next node in
linklist */
     } WIRE;
```

## 5. Writing Verilog Net-list

While writing the Verilog Net-List the whole WIRE link-list is traversed several times starting from the head to the end. And subsequently inputs, outputs and wires are identified on the basis of their unique properties. Since numerical names are not compatible with the Verilog compiler so we have added a prefix “n” in all the inputs, outputs and wire names. The wire types are:

**Primary Inputs:**

condition: fanin = 0, type = “inpt”

**Primary Outputs:**

    condition : fanout = 0

**Wire**:

    condition: !input && !output
                fanout != 0 && wire->type != “input”

**buff**:

    wire->type == “buff”
        assign wire to the input of the buffer.

**fanouts**:

    condition: wire->type == “from”
    for each fanout it is assigned to its fan-parent

**Exceptional wires**:

Primary Input is directly taken out as a Primary Output.

so in this case :

    wire->type == “inpt” && wire->level == 1

so such wires appear both in inputs are well as outputs. Name for output is changed by adding a prefix “o” in the wire name and later this is assigned to the original wire.

    ex:  for netlist c7552.isc
        assign o339 = n339;

## 6. Probability Computation

### a. Levelisation starting from output

We recursively move from PO (Primary Output) towards the inputs and assign levels to the wires until all PI have been leveled. The algorithm can be written as:

```
for each PO do
     PO->level = 1;
     Traverse(PO);
end

Traverse(OutWire) {
```

```
if OutWire == PI
     return;

for each input wire for this OutWire
     wire->level = OutWire->level + 1;
     if( wire isFanout ){
     // assign a level to its Parent also
          fanParent->level =
          MAX(fanParent->level, wire->level);
     // if level for parent is changed
          Traverse(fanParent);
     }else{
          Traverse(wire);
     }
}
```

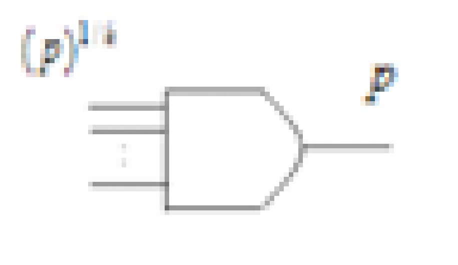
## b. Probability Computation

The ISCAS 85 benchmark circuits are a set of combinational circuits. We start with the assumption that the probability of observing a 1 at each of the outputs is 0.5. We then propagate this signal line probability back to the primary inputs. For this we use the following algorithm:

*Step1*: Assign to all the outputs a signal line probability (probability that that line is 1) equal to 0.5

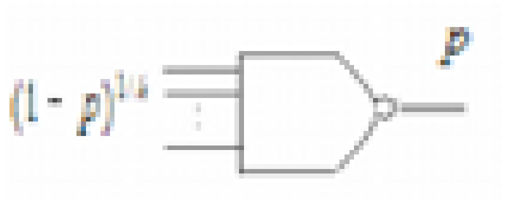
*Step2*: Moving backwards from each of the output lines being assigned in the Step1 above, compute the signal probability assignment at the primary inputs by propagating the signal probability values from the outputs to the inputs of gates according to the following formulas:

- For a k – input AND gate with output signal probability = p, each input has a signal probability given by:



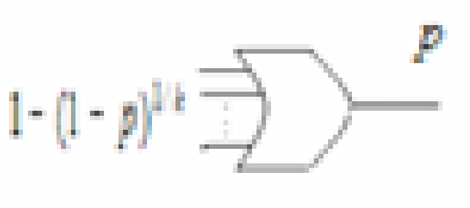
$$AND: \quad p_i = (p)^{1/k}$$

- For a k – input NAND gate with output signal probability = p, each input has a signal probability given by:



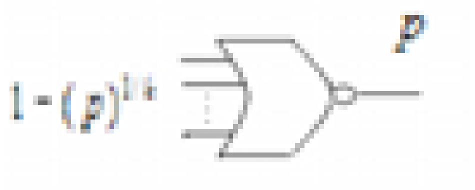
$$NAND: \quad p_i = (1-p)^{1/k}$$

- For a k – input OR gate with output signal probability = p, each input has a signal probability given by:



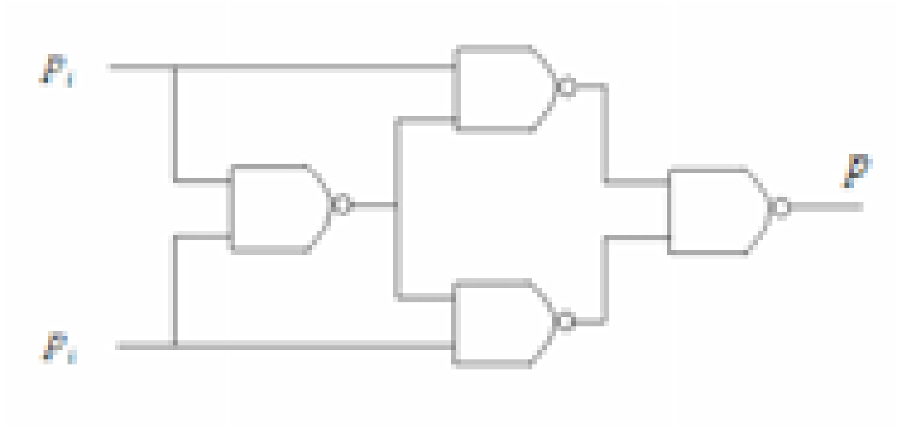
$$OR: \quad p_i = 1-(1-p)^{1/k}$$

- For a k – input NOR gate with output signal probability = p, each input has a signal probability given by:



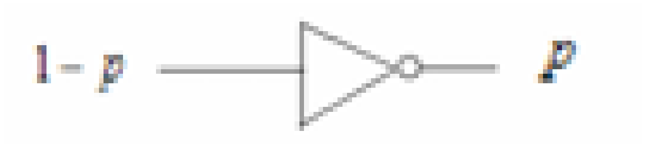
$$NOR: \quad p_i = 1-(p)^{1/k}$$

- For a 2 – input EXOR gate with output signal probability = p, each input has a signal probability given by:



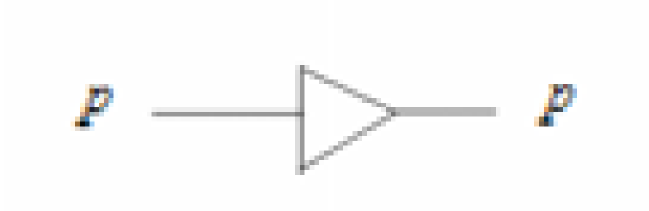
$$EXOR: \quad p_i = \frac{(1-(1-(1-p)^{1/2})^{1/2})^{1/2} + (1-(1-p)^{1/2})^{1/2}}{2}$$

- For a NOT gate with output signal probability = p, the input has a signal probability given by:



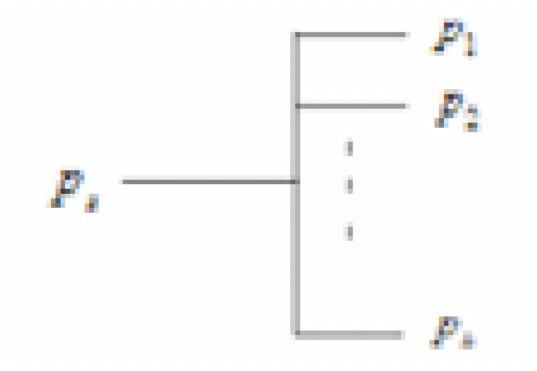
$$NOT: \quad p_i = 1 - p$$

- For a BUFFER with output signal probability = p, the input has a signal probability given by:



$$BUFF: \quad p_i = p$$

- For a fan out with branch signal probabilities $p_i$, i = 1, 2, 3, …, k, the stem signal probability $p_s$ is given by:



$$p_s = \frac{1}{k}\sum_{i=1}^{k} p_i$$

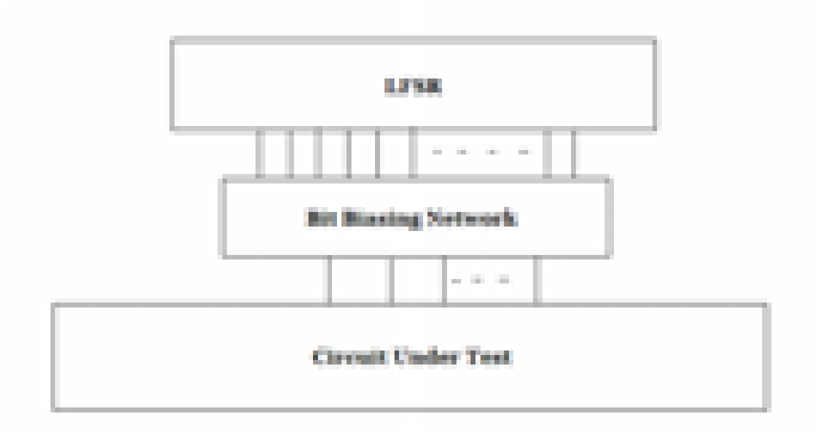
The signal probability assignment at any fan-out stem is completed only when all the branches have been computed. Only then can one go back from that fan-out stem.

*Step3*: Record the signal probability assignments computed for the primary inputs.

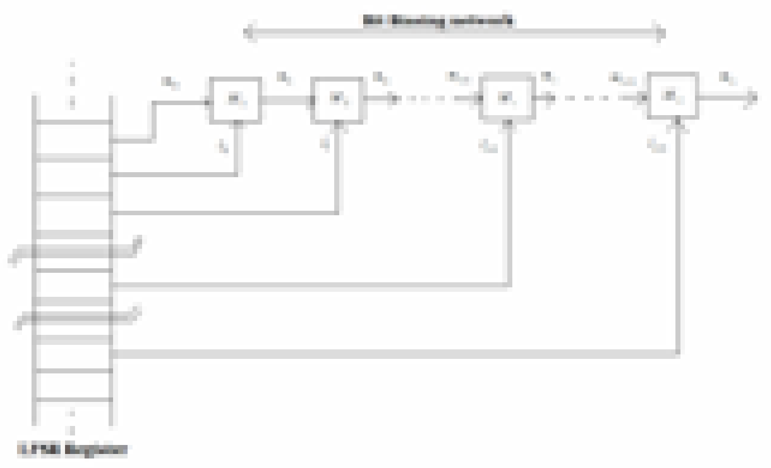
## 7. Weight Generating Combinational Circuit

### a. Computing the Combination Circuit

At this point we assume that the weights are available in the form of probabilities of each primary input. We then use the following scheme as illustrated in the diagram below for generating a weighted vector. This block inserted between the LFSR and the CUT represents the biasing circuitry needed to map the equi-probable patterns generated by the LFSR onto the weighted patterns that will best stimulate the CUT.



The general network structure used in conjunction with an LFSR for bit-wise weighted bit pattern generation is shown in the following diagram:



The modules $M_i$'s are simple 2-input NAND or AND gates arranged in a manner dependent on the bit bias values desired. The accuracy with which we are able to realize a weight is a function of n and is equal to $2^{-(n+1)}$. The signal line probabilities associated with AND and NAND modules, used to construct the network structure are given by the following equations:

$$M_i : AND - gate \rightarrow X_i = \frac{1}{2} X_{i-1} \ldots\ldots \text{ (1)}$$

$$M_i : NAND - gate \rightarrow X_i = 1 - \frac{1}{2} X_{i-1} \ldots\ldots\text{(2)}$$

where $X_i$ = Prob($m_i$ = 1) and Prob( $l_i$ = 1) = 0.5 for I = 1, 2, 3, …, n.

Following this, the algorithm used to get the NAND AND gate combinations is as follows:

*<u>Step1</u>: Determining the number of gates required by the Bias Network.*

If the desired accuracy is say a, then the maximum no. of gates is the smallest integer n satisfying the relationship, $a \leq 2^{-(n+1)}$ or equivalently:

$$n = \left\lceil -\frac{\log a}{\log 2} - 1 \right\rceil = \left\lceil -\log_2 a - 1 \right\rceil$$

*<u>Step2:</u> Select the proper function (NAND vs. AND)*

Insert the value of the desired weight in $X_n$ and calculate $X_{n-1}$ using the equations (1) and (2). Only one of these results will be in between 0 and 1. The other is always outside the probability range of [0, 1] and is rejected. The gate associated with the same value yielding the valid result is assigned to the module $M_n$.

Then, calculate the accuracy obtained so far using the following equation:

$$a_{[n,i]} = \left| \frac{1}{2} - X_{n-i} \right| .2^{-i}$$

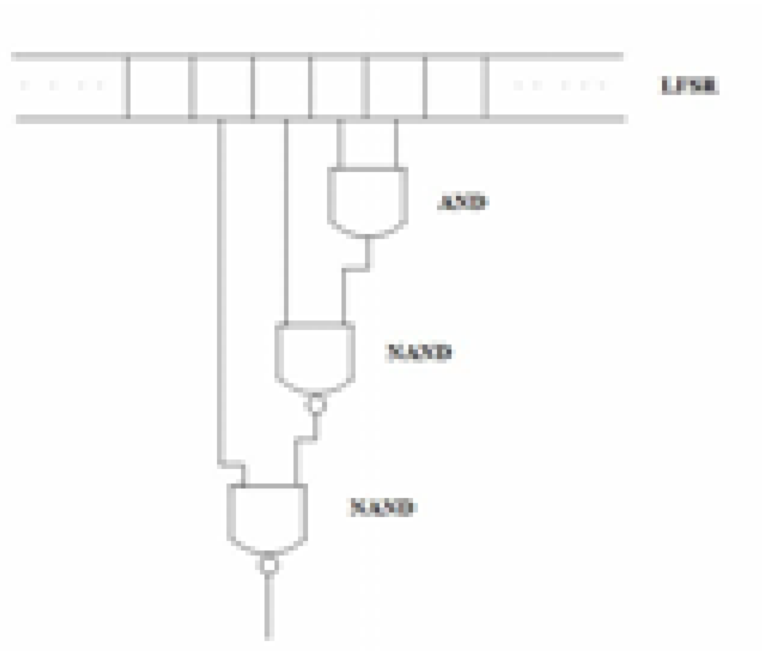
Then, using the value just obtained for $X_{n-1}$, one determines, following the same procedure, $X_{n-2}$and the type of gate associated with $M_{n-1}$. The process is carried out recursively until either $a_{[n-i]}$ of any given iteration becomes less than a or all n modules of the system have been determined. When module $M_1$ is reached, the required input to this module is taken directly from any stage of the LFSR Stage, and the accuracy sought, a, is constructively assured.

## b. Illustrative Example

In all our computation we have taken a = 0.04, giving us n = 5 which is the maximum no. of LFSR bits per generated weighted bit. Consider the simplest ISCAS85 circuit c17.isc. The probability computation shows that the last input bit has a probability of 0.541196. The Verilog code pertaining to the combinational bias generating circuit for this weighted bit is as follows:

```
wire in_0_0;
assign out[0] = in_0_0;
wire in_0_1;
assign in_0_0 = ~( (combin[0]) & (in_0_1) );
wire in_0_2;
assign in_0_1 = ~( (combin[1]) & (in_0_2) );
wire in_0_3;
assign in_0_2 = (combin[2]) & (in_0_3);
assign in_0_3 = combin[3];
```

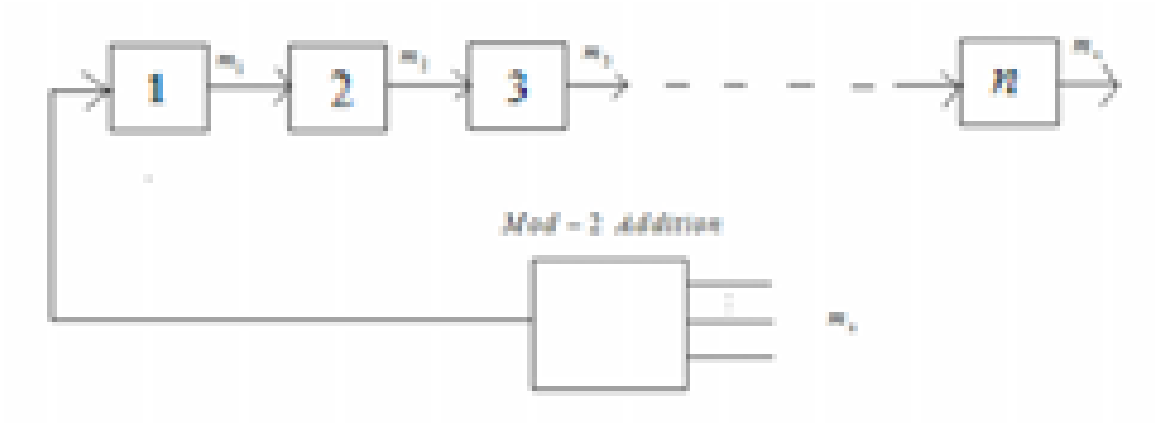
This bit uses 4 bits of an LFSR which corresponds to a circuit as follows:



## c. Linear Feedback Shift Registers

The LFSR are the basic building blocks of the pseudo random test pattern generators. In unbiased pseudo random testing, the outputs from the LFSR is fed directly to the CUT and thus the no. of LFSR stages required is equal to the number of inputs to the CUT. For a weighted pseudo random testing we however require much more LFSR stages than the inputs to the CUT. This is so because each weighted bit usually requires more than one equi-probable bit (exact number determined by the accuracy that we assume) coming in from an LFSR stage for the generation of its weighted bit.

Now we assume that for both the unbiased and the weighted case we have the total number of LFSR shift registers required for each of the CUTs. In that case the LFSR can be made as follows:



For pseudo random testing it is necessary that the period of the basic sequence is as high as possible. i.e. we always try to make a maximal length sequence. Now, in our case we observe that the required no. of stages (n) for the 11 circuits under consideration varies from 5 to 801. To find the feedback terminals for each of the LFSRs, we refer to the appendix given in [3] for a table of the simplest primitive polynomials for degrees up to 300. For n greater than 300, (in our case here we require primitive polynomials for LFSR with n = 635, 678 qnd 801), we make use of a heuristic. We express these values of n as a multiple of two numbers within 300 and then combine the primitive polynomials of these two multiples. As an illustration, consider: n = 801

We can write: n = 801 = 9 x 89. The taps corresponding to n = 9 are [5, 9] and the taps corresponding to n = 89 are [51, 89]. As such, the taps corresponding to n = 801 will be [407, 445, 763, 801].

**d. Preventing Temporal Correlation**

In case of weighted pseudo random testing, in order to ensure randomness, we need to eliminate any direct correlation in time among the individual weighted bits $B_i$. In the case of LFSR, this is done by considering the generator every b clock cycles where b is the maximum number of LFSR bits used in the formation of a single weighted bit. This operation ensures that none of the LFSR bits used to generate the weighted bits $B_i(t)$ are not used again for generating the bits $B_i(t+1)$. Therefore no direct correlation in time will exist in $B_i$.

# 8. <u>Pattern Generation</u>

In the last section we see how, depending upon the no. of inputs of the CUT and the associated probabilities we can make an LFSR configuration for both, the unbiased and the weighted pseudo random testing part. We do this by writing a Verilog file with the entire configuration written in it. After that we run the Verilog simulator and get the raw patterns in a .ptrn file. After this we need to parse this file to get a file in the .stil format which the Tetramax Tool can understand and take in as an input for the fault simulation.

For this we make use of two parser files: read_weights.c which reads the patterns generated from an LFSR working for weighted pseudo random testing. In this case the parser file takes every b'th pattern and wrties it into the stil format so as to prevent the temporal correlation as shown above. The other parser file reads the patterns generated from an LFSR working for unbiased pseudorandom testing. In this case we take every pattern and put it into the ,stil file.

## 9. <u>Using Tetra Max</u>

For every circuit we will need to do fault simulation using the patterns generated in the last stage. For this we use the Tetra Max tool. We firstly read the net-list for that circuit in the Verilog format. We showed in section 5 how this is done. We then build the net-list and run a design rule check to see if there are any violations. After that we run ATPG to generate a .stil file which we later edit with our own generated patterns. We can control the number of patterns that get simulated and thus do fault simulation using the tool and find out the how the percentage fault coverage changes as the number of test patterns used are changed.

## 10. <u>Results</u>

## c. Conclusion

A comparative study was made between unbiased pseudo random test pattern generation and weighted pseudo random test pattern generation for all the combinational circuits under the ISCAS85 format. It was observed that for some the cases, the unbiased pseudo random testing is actually better than the weighted pseudo random testing. i.e. it takes a lesser number of test patterns generated using unbiased pseudo random testing to cover a larger number of faults as compared to the weighted pseudo random testing. A number of analyses could be used to understand this. Firstly, for the small circuit c17.isc, there's not much difference between the weighted and unbiased test patterns. It might happen that in such small circuits, the initial seed might play an important role in determining the fault coverage behavior of the test patterns. Next, overall we need to be sure of the following constraints or limitations with respect to our design:

1. The probabilities computed are not realized perfectly. As in there's only a certain accuracy up to which we can go (In our case, it is determined by the factor 'a' which we have taken as 0.04). The implication of this hasn't been studied
2. In certain cases where a large LFSR was to be used, we didn't use primitive polynomials as they were not directly available. The methods to generate primitive polynomials for such LFSR where the number of bits is large should be studied.
3. The probabilities computed were assuming that all the outputs have a signal probability of 0.5 simultaneously. Such a condition will lead to a lot of clashes at the fan-out stems. Because of this, if we use the heuristic used in our case here (that of taking the minimum signal probability among all the fan-out branches), we might end up with an input probability assignment which if again forward traced to the output won't give us signal probabilities at the output as 0.5. As such, we need to explore other methods for signal line probabilities and clash resolution. Methods involving different heuristics, like taking an average or the maximum of the branches' signal line probabilities, or taking a weighted average of the branches' signal line probabilities that are weighed according to the distance of that branch from the outputs or the inputs; or iteratively assigning the input probabilities so as to minimize the mismatch between the forward trace and backward trace of the signal line probabilities; can be studied and implemented to improve the fault coverage.

4. We should keep in mind that in an unbiased pseudo random testing process, if the number of test vectors generated is less than the period of the LFSR, no two test vectors will be repeated in that set. However, in our scheme wherein m LFSR bits map into n weighted bits (where m is about 3-4 times n), there will be repetition. Because of this, the effective number of test vectors in the weighted pseudo random case is lesser than the number of test vectors generated.

5. The method used is good for demonstration purposes but is inefficient if we look with respect to hardware and time efficiency. Hardware-wise, we use a lot more number of LFSR bits compared to the number of inputs of the CUT leading to an increase in the hardware. Time-wise, we generate one test vector per clock cycle of the LFSR in an unbiased pseudo random testing scheme while in a weighted scheme we generate one test vector every 'b' clock cycles of the LFSR where 'b' is the maximum number of LFSR bits used by the weighted bits. Such inefficiency can be reduced by either involving the use of deterministic test patterns that are known a-priori or by making use of correlated test vectors whose correlation is complemented by suitable means.

6. As a future check on the system, one can simply generate the inputs using the weighted pseudo random technique described in the report and find the corresponding outputs. Then one can run a statistical check on the data obtained so as to confirm whether the probabilities match with those designed for. However the sample set needs to be large to make such computations.

## 11. <u>References</u>

[1] Silvio Bou-Ghazale and Peter N. Marinos, "Testing with correlated test vectors," pp. 254 – 262, FTCS-22. Digest of Papers., Twenty-Second International Symposium on Fault Tolerant Computing., July 1992

[2] Miguel A. Maranda and Carlos A. Lopez- Barrio, "*Generation of Optimized Single Distribution of Weights for Random Built In Test,*" Proceedings of the IEEE International Test Conference, 1993.

[3] Paul H. Bardell, and Jacob Savir, "*Built in Test for VLSI – Pseudo random Techniques,*" (Text Book).