# BIST

## Abstract

The basic idea of BIST, in its most simple form, is to design a circuit so that the circuit can test itself and determine whether it is “good” or “bad” (fault-free or faulty, respectively). This typically requires that additional circuitry and functionality be incorporated into the design of the circuit to facilitate the self-testing feature. This additional functionality must be capable of generating test patterns as well as providing a mechanism to determine if the output responses of the circuit under test (CUT) to the test patterns correspond to that of a fault-free circuit.

1. Block diagram and explanation of your complete design with CUT and BIST Logic. Please include your selection criteria of BIST components and also how you calculated the golden (expected) signature.

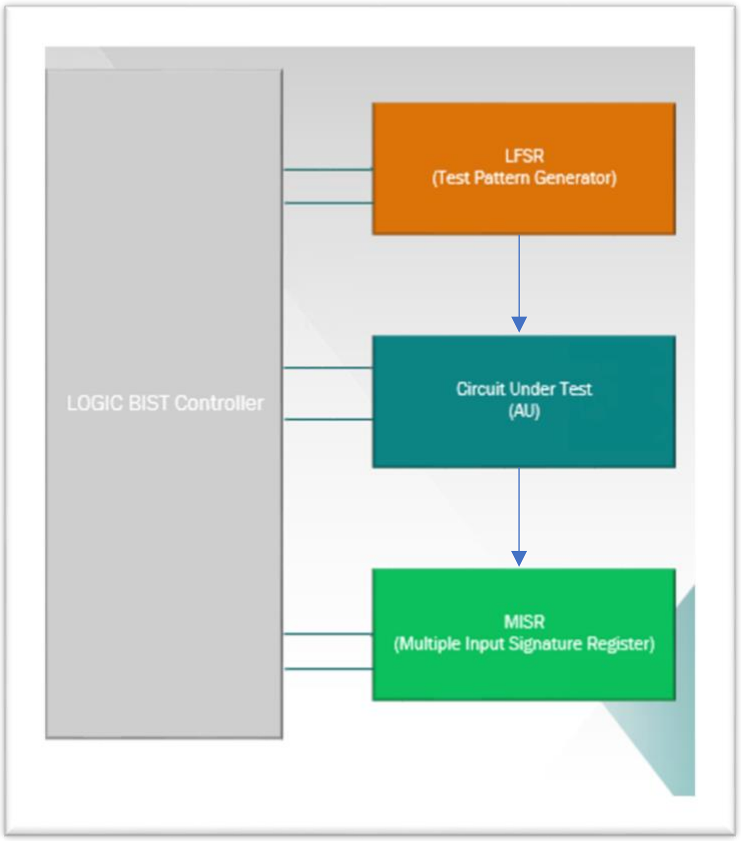


Fig 1: BIST Block Diagram

Basic BIST Architecture:

A representative architecture of the BIST circuitry as it might be incorporated into the CUT is illustrated in the block diagram of Figure 2. This BIST architecture includes two essential functions as well as two additional functions that are necessary to facilitate execution of the self-testing feature while in the system. The two essential functions include the test pattern generator (TPG) and output response analyzer (ORA). While the TPG produces a sequence of patterns for testing the CUT, the ORA compacts the output responses of the CUT into some type of *Pass/Fail* indication. The other two functions needed for system-level use of the BIST include the test controller (or BIST controller) and the input isolation circuitry. Aside from the normal system I/O pins, the incorporation of BIST may also require additional I/O pins for activating the BIST sequence (the *BIST Start* control signal), reporting the results of the BIST (the *Pass/Fail* indication), and an optional indication (*BIST Done*) that the BIST sequence is complete and that the BIST results are valid and can be read to determine the fault-free/faulty status of the CUT.

## BIST

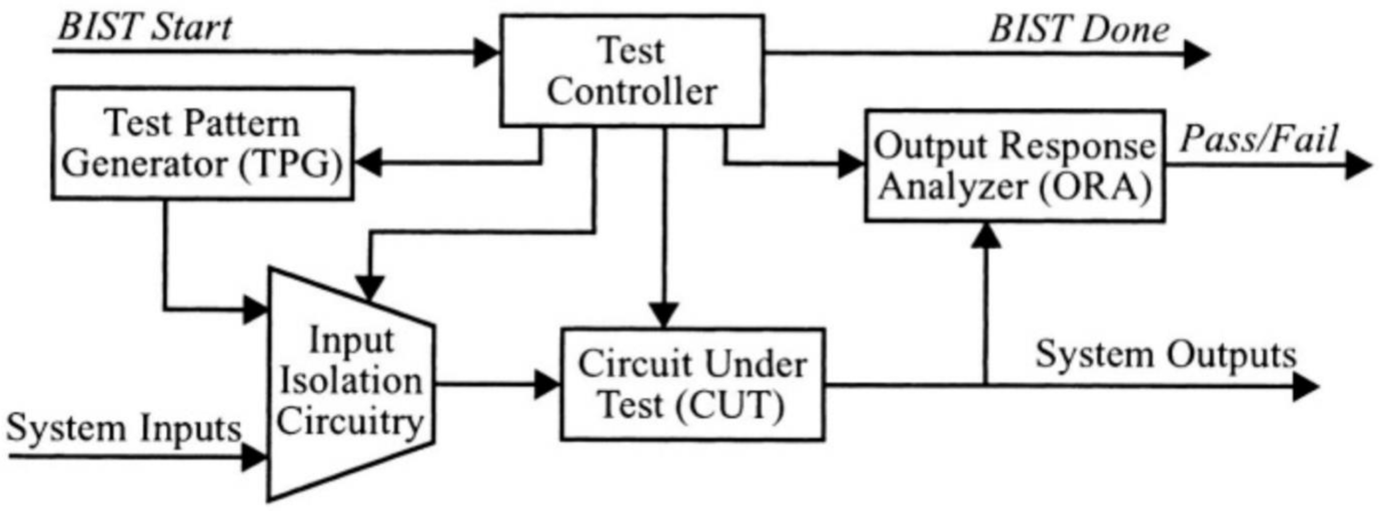


Fig 2: BIST Architecture

<u>Features of testing:</u>

- In BIST, we use the random pattern test approach as using LFSR's. The patterns generated are not completely random, they are seeded by a sequence of inputs as an initial seed hence becoming a pseudo random test pattern. These pseudo random test patterns are repeated after few clock cycles.
- The fault coverage of the random pattern testing can be determined using fault simulation.
  - The test length is large
  - Much faster test generation
  - Continue until fault reaches 70-80%, the switched to ATPG

<u>Calculation of Golden Signature:</u>

The golden signature is a process where we basically divide one polynomial with other polynomial and the remainder of this operation results in the final Golden Vector.

The two polynomials in turn are known as the input polynomial and characteristic polynomial.

In this lab, I have taken the polynomials as mentioned below:

Input sequence: 11010010
Input polynomial: $x_6 + x_3 + x + 1$

characteristics polynomial: $x_2 + x_2 + 1$

From the above given polynomials, the 8-bit input polynomial is divided by the 4 bit characteristics polynomial and the remainder obtained is the final golden signature.

here, we find the golden signature to be: 0101

This golden signature is further used to match with the final test output and if there is any mis-match then the signal *fault_detected* will go high. Otherwise with the proper functioning of circuit as well as when the *BIST_mode* is low, the output of *fault_detected* is logic low.

Below we see the tabular representation of the golden signature calculations:

| Inputs | x^0 | x^1 | x^2 | x^3 |
|---|---|---|---|---|
| Initial State | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |

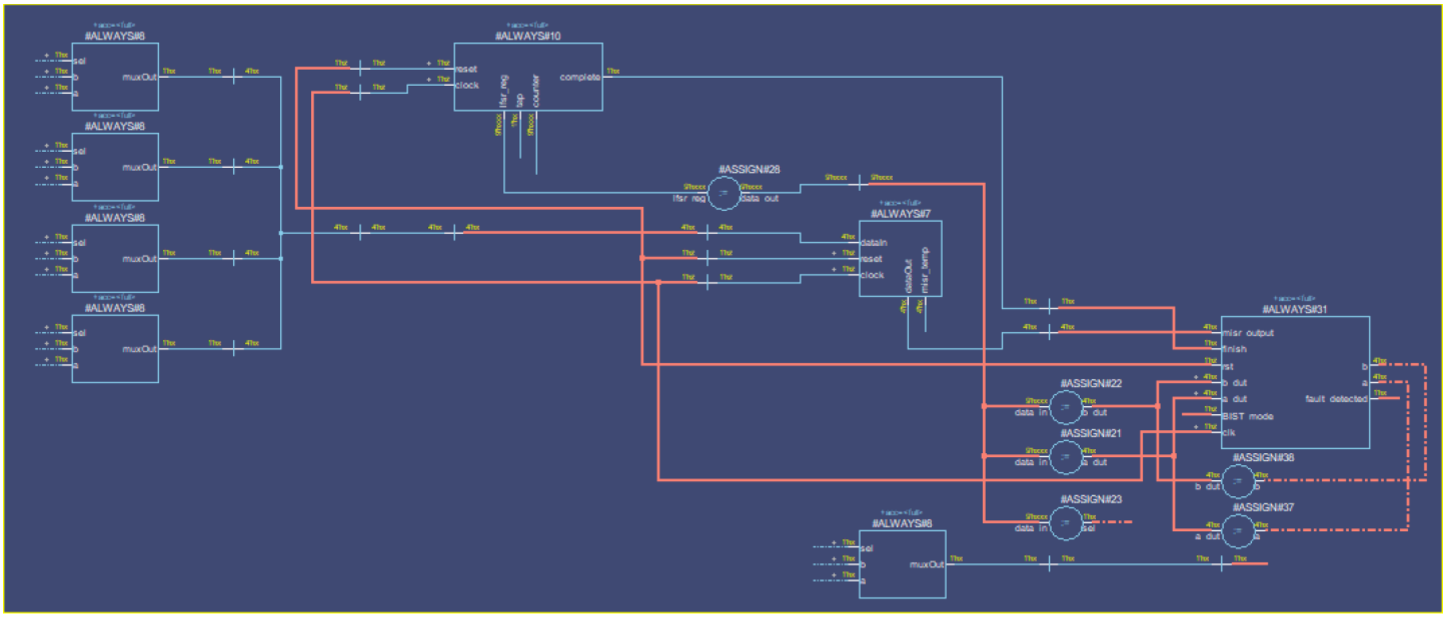


Fig 2: BIST controller Dataflow.

## 2. VHDL/Verilog implementation of each of the three major BIST components.

In this lab I have used Verilog HDL for the design of Test Pattern Generator, Output Response Analyzer as well the Controller which serves as the backbone of the entire implementation.

❖ The Verilog code for Test Pattern Generator is provided below:

```
module ML_lfsr(data_out,complete,reset,clock, counter);
input reset;
input clock;
output [8:0] data_out;
output complete;
reg complete;
reg [8:0] lfsr_reg;          //lfsr_reg is intialized as a 9 bit register
output reg [8:0] counter;
reg tap;
always@(posedge clock or posedge reset)
begin
  if(reset == 1)
```

```
begin
lfsr_reg <= 9'b101001010;
counter <= 9'b000000000;  //counter has been initialized to 0 and it will start counting up
end
  else
begin
tap = lfsr_reg[8] ^ lfsr_reg[5];   //tapping has been specified here for the operation of LFSR
lfsr_reg[8:0] <= { lfsr_reg[7:0], tap };
counter = counter + 1;
if(counter < 9'b111111110) //until the counter reaches the point 2^n-1, the complete signal is logic low
     complete = 0;
else
      complete = 1;
end
  end
assign data_out = lfsr_reg;   //the output port data_out has been assigned the final sequence of lfsr_reg
endmodule
```

❖ The Verilog code for Ouput Response Analyzer is provided below:

```
module MISR_4bit(dataIn,reset,clock,dataOut);
input [3:0] dataIn;
input reset,clock;
output reg [3:0] dataOut;
reg [3:0] misr_temp;
always@(posedge clock or posedge reset)
begin
  if(reset == 1)
dataOut <= 4'b0000;
else
 begin
misr_temp = dataOut;
dataOut[0] = misr_temp[3] ^ dataIn[0];
dataOut[1] = misr_temp[3] ^ misr_temp[0] ^ dataIn[1];
dataOut[2] = misr_temp[1] ^ dataIn[2];
dataOut[3] = misr_temp[2] ^ dataIn[3];
 end
end
endmodule
```

❖ The Verilog code of controller is provided below:

```
module controller_new (clk, rst, BIST_mode, data_in, finish, fault_detected, result_dut, misr_output);
input [8:0] data_in;
input clk; //clock
input rst; //reset
input finish; //finish flag
input BIST_mode; //BIST mode
output reg fault_detected; //fault detected flag
wire [3:0] a_dut; //4 bits of 9 bit LFSR output
wire [3:0] b_dut;
wire sel;   //assign 1 bit of LFSR output to select
wire output_dut; //carry output of DUT
input [3:0] result_dut; //result of DUT fed to MISR for compaction to compare with the Golden Signature
parameter golden_signature = 4'b0101; // golden signature as derived from the tapping of LFSR bits
output [3:0] misr_output;  //output of the misr
reg [3:0] a,b;

//port-mapping LFSR with controller
ML_lfsr lfsr (.data_out(data_in), .complete(finish), .reset(rst), .clock(clk));

//assignment of data_in bits to different bits of a, b and Sel
```

```
assign a_dut = data_in[8:5];
assign b_dut = data_in[4:1];
assign sel = data_in[0];

//port-mapping DUT with controller
AU DUT (.A(a), .B(b), .Sel(sel), .Result(result_dut), .Output(output_dut));

//port-mapping MISR with controller
MISR_4bit MISR (.dataIn(result_dut), .reset(rst), .clock(clk), .dataOut(misr_output));

always @ (posedge clk)
begin
        if(BIST_mode == 0)        //when bist mode is not on, faults can't be detected
        fault_detected = 0;
else
begin
        assign a = a_dut;  //assigning values of a
        assign b = b_dut;  //assigning values of b
        if(rst == 1)          //when reset is high, faults can't be detected
        fault_detected = 0;

        else
        begin
        if(finish == 1)
        begin

//match the golden signature with misr_output, if the mtch is found then fault is not detected and fault_detected bit is 0

                if(golden_signature == misr_output)
                fault_detected = 0;
                else
                fault_detected = 1;
end
end
end
end
endmodule
```

3. Test benches for each component and simulation waveforms.

❖ The test bench code for Test Pattern Generator is provided below:

```
module tb_ML_lfsr;
reg clock,reset;
wire [8:0] dt_out;
wire [8:0] counter;
ML_lfsr lfsr1(dt_out, complete, reset, clock,counter);
initial
begin
clock = 1'b0;
end
always
#5 clock = ~clock;
initial
begin
reset = 1'b1;
#10 reset = 1'b0;
end
endmodule
```

➢ Waveform of TPG test bench

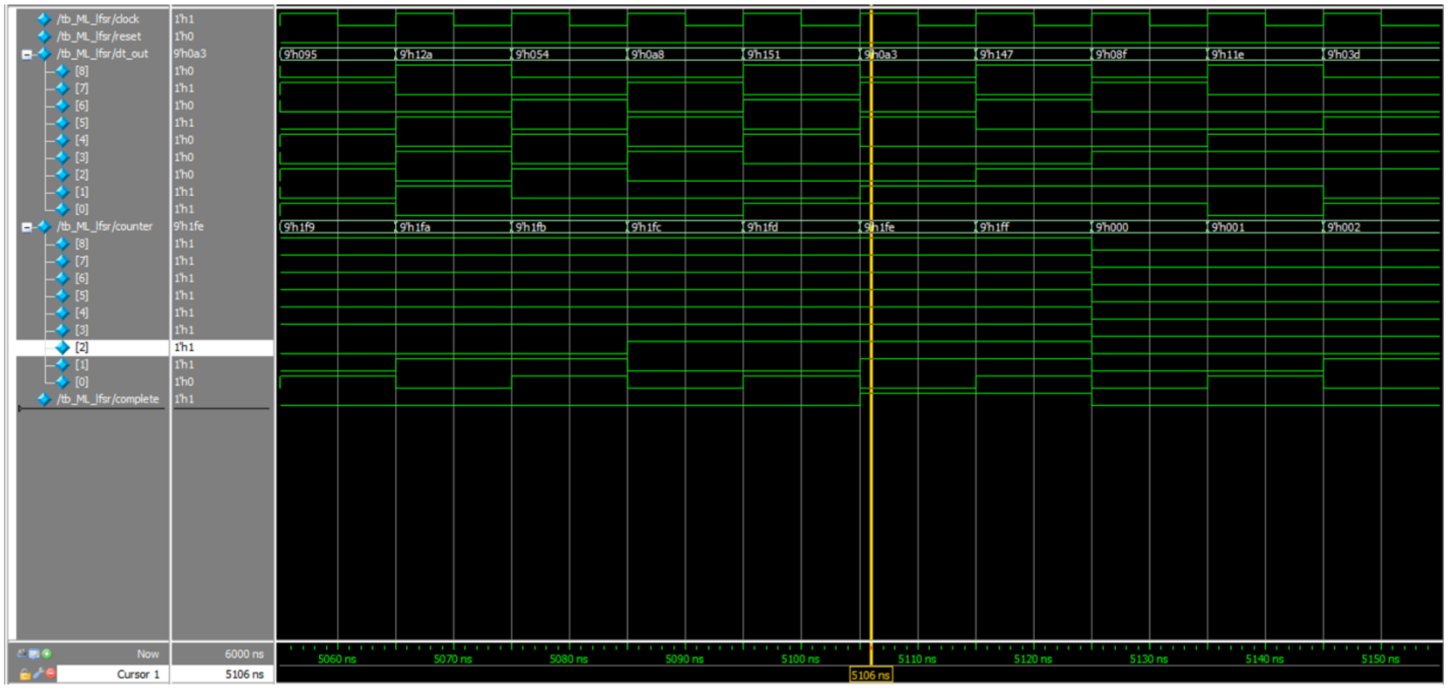


Fig 3:  LFSR simulation response

❖ The test bench code for Output Response Analyzer is provided below:

```
module tb_MISR4bit;
reg clock,reset;
wire [3:0] dataOut;
reg [3:0] dataIn;
MISR_4bit misr4bit1(dataIn,reset,clock,dataOut);
initial
begin
clock = 1'b0;
end
always
#5 clock = ~clock;
initial
begin
reset = 1'b1;
#10 reset = 1'b0;
dataIn = 4'b0111;
#10 dataIn = 4'b0000;
#10 dataIn = 4'b1100;
#10 dataIn = 4'b1010;
#10 dataIn = 4'b0111;
#10 dataIn = 4'b0001;
#10 dataIn = 4'b1101;
#10 dataIn = 4'b1010;
end
endmodule
```

➢ Waveform of ORA test bench

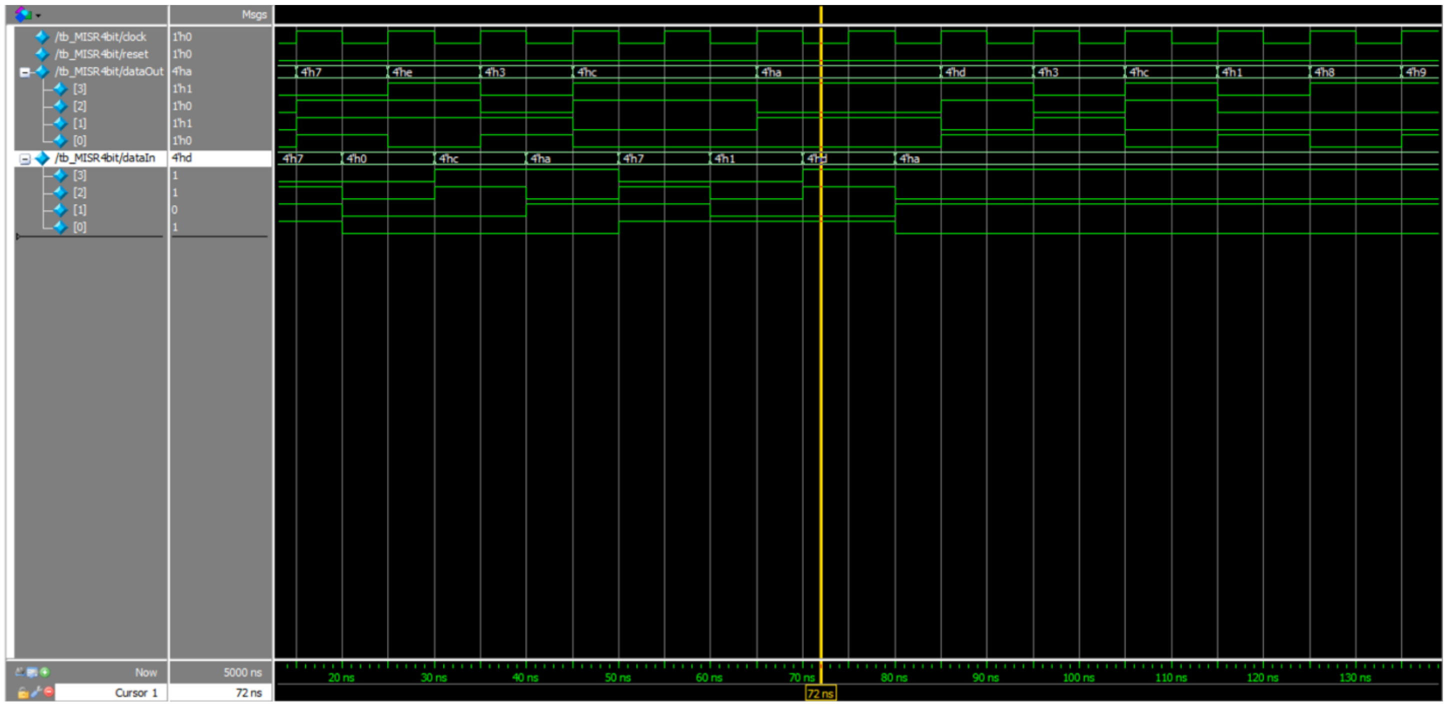

Msgs
/tb_MISR4bit/clock 1'h0
/tb_MISR4bit/reset 1'h0
/tb_MISR4bit/dataOut 4'ha
[3] 1'h1
[2] 1'h0
[1] 1'h1
[0] 1'h0
/tb_MISR4bit/dataIn 4'hd
[3] 1
[2] 1
[1] 0
[0] 1
4'h7 4'he 4'h3 4'hc 4'ha 4'hd 4'h3 4'hc 4'h1 4'h8 4'h9
4'h7 4'h0 4'hc 4'ha 4'h7 4'h1 4'hd 4'ha
Now 5000 ns
Cursor 1 72 ns
20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns 110 ns 120 ns 130 ns
72 ns

Fig 4: ORA simulation response

❖ The test bench code for BIST controller is provided below:

```
`timescale 1ns/100ps
module tb_controller_new;
reg [8:0] data_in;
reg clk; //clock
reg rst; //reset
reg finish; //finish flag
reg BIST_mode; //BIST mode
//reg misr_ouput;
reg [3:0] a,b;
reg [3:0]golden_signature = 4'b0101;
wire [3:0] misr_output;
wire [3:0] result_dut;

controller_new controller_test(clk, rst, BIST_mode, data_in, finish, fault_detected, misr_output, result_dut);

initial
begin
        clk = 0;
        rst = 1;
        forever #5 clk = ~clk;
end

initial
begin
        forever #8 rst = ~rst;
end


always @(posedge clk)
begin
        BIST_mode = 0;
        a = 4'b0010;
        b = 4'b0101;
        #5 a = 4'b1011;
        #5 b = 4'b1111;
        #5 a = 4'b1011;
        #5 b = 4'b1011;
        #5 a = 4'b1001;
        #5 b = 4'b1110;
        data_in = 9'b 101100110;
        #10 finish = 1;
        #40 BIST_mode = 1;
        #12 data_in = 9'b 001101001;
        #12 data_in = 9'b 101101101;
        #12 data_in = 9'b 001111111;
        #12 data_in = 9'b 111101001;
        #5 a = 4'b1011;
        #5 b = 4'b1111;
        #5 a = 4'b1011;
        #5 b = 4'b1011;
        #5 a = 4'b1001;
        #5 b = 4'b1110;
end
endmodule
```

- Waveform of controller test bench

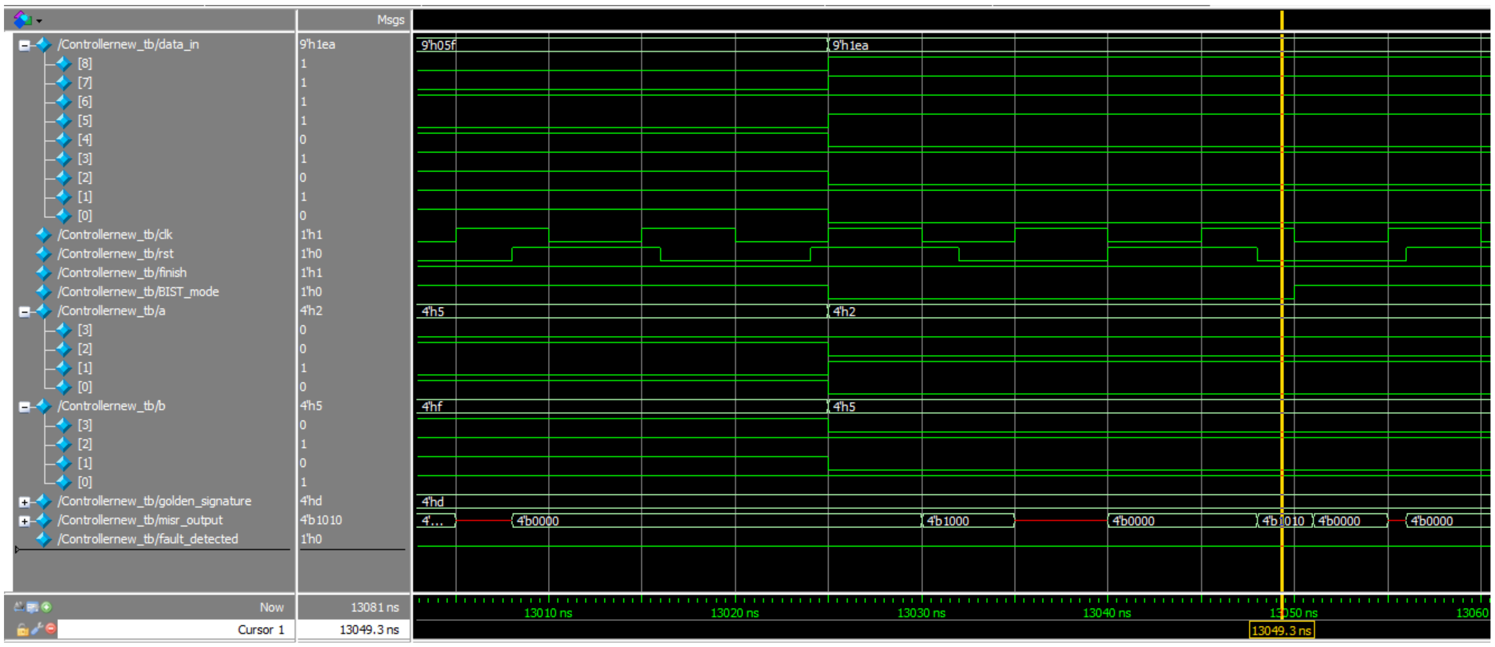


Fig 5: Controller simulation response

## 4.Simulation waveforms for test mode with FAULT-FREE (PASS) and FAULTY (FAIL) cases, and normal mode with some operations.

In the fault-free case, the DUT is used normally and BIST is used on the DUT. The controller simulation response shows fault_detected signal as 0 in the fault free case.

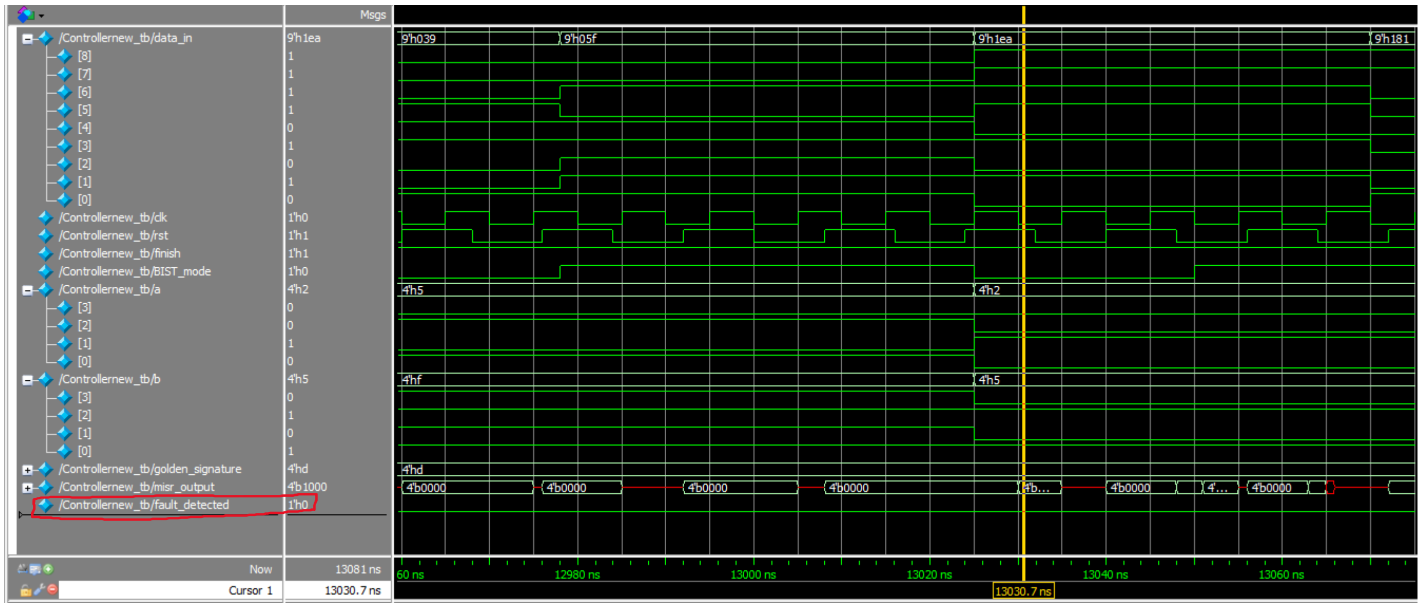
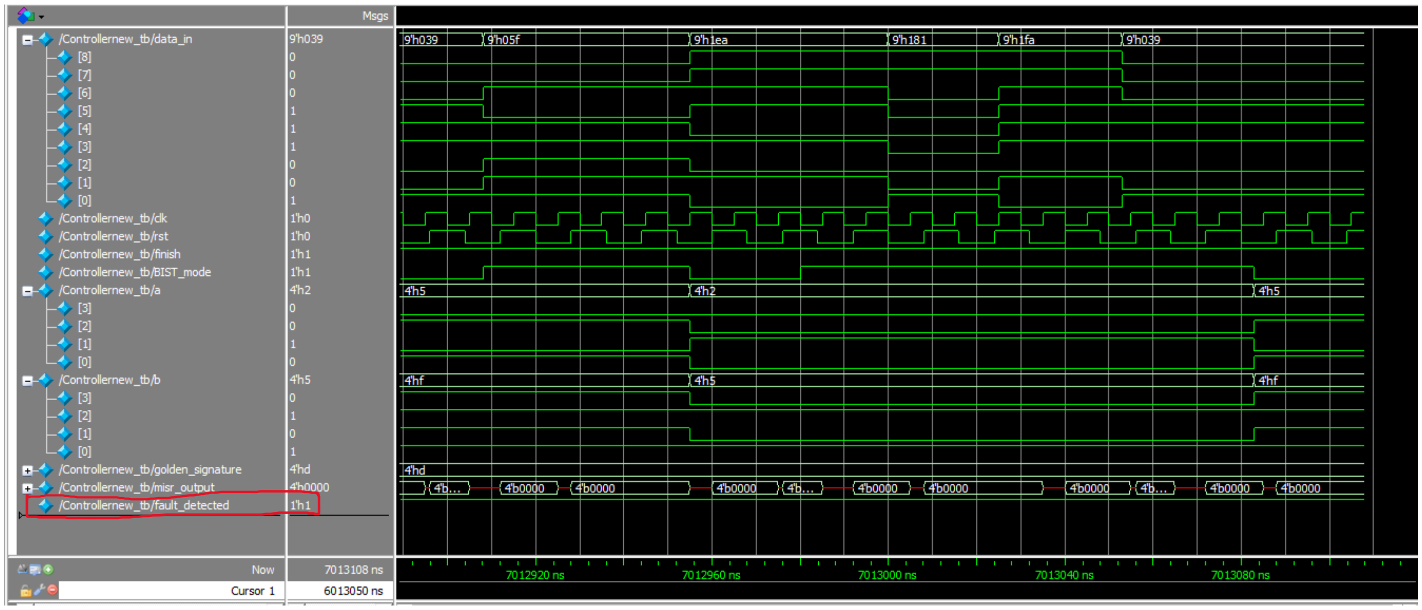

Msgs
/Controllernew_tb/data_in 9'h1ea
/Controllernew_tb/clk 1'h0
/Controllernew_tb/rst 1'h1
/Controllernew_tb/finish 1'h1
/Controllernew_tb/BIST_mode 1'h0
/Controllernew_tb/a 4'h2
/Controllernew_tb/b 4'h5
/Controllernew_tb/golden_signature 4'hd
/Controllernew_tb/misr_output 4'b1000
/Controllernew_tb/fault_detected 1'h0
Now 13081 ns
Cursor 1 13030.7 ns
9'h039
9'h05f
9'h1ea
9'h181
4'h5
4'h2
4'hf
4'h5
4'hd
4'b0000
12980 ns
13000 ns
13020 ns
13040 ns
13060 ns
13030.7 ns

Fig 5: Controller simulation response of fault-free DUT

In the faulty case, the DUT has been injected with a stuck-at fault and BIST is used on the DUT. The controller simulation response shows fault_detected signal as 1 in the fault free case.


Msgs
/Controllernew_tb/data_in 9'h039
/Controllernew_tb/clk 1'h0
/Controllernew_tb/rst 1'h0
/Controllernew_tb/finish 1'h1
/Controllernew_tb/BIST_mode 1'h1
/Controllernew_tb/a 4'h2
/Controllernew_tb/b 4'h5
/Controllernew_tb/golden_signature 4'hd
/Controllernew_tb/misr_output 4'h0000
/Controllernew_tb/fault_detected 1'h1
Now 7013108 ns
Cursor 1 6013050 ns
9'h039
9'h05f
9'h1ea
9'h181
9'h1fa
9'h039
4'h5
4'h2
4'h5
4'hf
4'h5
4'hf
4'hd
4'b0000
7012920 ns
7012960 ns
7013000 ns
7013040 ns
7013080 ns

## 5. DC compiler script for synthesis and synthesis results.

While using the DC compiler script for synthesis errors were being thrown. The changes were made in the file dc_syn.tcl and the database files were added in the directory. Still there were read errors on the DC compiler part. Hence, synthesis was not performed on the BIST circuitry.

## 6. Please elaborate also on what you've learnt and how your approach can be extended to create a complete DFT EDA software.

BIST is very commonly used at industries which specialize in avionics, medical devices, automotive electronics, and integrated circuits. Built-In-Self-Test is used to make faster, less-expensive integrated circuit manufacturing tests. The IC has a function that verifies all or a portion of the internal functionality of the IC. In some cases, this is valuable to customers, as well. For example, a BIST mechanism is provided in advanced fieldbus systems to verify functionality. At a high level this can be viewed similar to the PC BIOS's power-on self-test (POST) that performs a self-test of the RAM and buses on power-up.

This Lab was a learning opportunity to understand the basic functionality and implementation of BIST. The generation of Golden Vectors was the most challenging part for me and understanding how the polynomials work mathematically to provide one vector out of an exhaustive set of vectors was a great learning experience.

This implementation of BIST can be extended of more complex circuits and machines where these machines are self-capable of testing their functionality and faults while being offline. This gives the end user comfort of distinguishing between a faulty and a fault-free machine or circuit.

## REMARKS

- This lab was carried out using ModelSim PE student edition software.
- The source codes and test benches are written in verilog using ModelSim.
- The block diagrams on this report are made using Microsoft Visio Software.
- In this Lab, the BIST architecture was implemented which is very useful to test the functionality of any given circuit weather be it offline or online testing.
- The DC complier was not working properly so, synthesis part couldn't be performed in this lab.