INSTITUTO SUPERIOR TÉCNICO

Design Test and Reliability of Electronic Systems

Project 2 - Circular Bist

Group 2

n° 77973 - Pedro Pina

n° 84779 - Afonso Muralha

Professor Fernando Gonçalves

May 31, 2020

Contents

I. Introduction and Objectives

The goal of this second project of the Design Test and Reliability of Electronic Systems course is to implement a Circular built-in self-test (BIST) system to circuit assigned.

The developed system must be submitted to a fault simulator that will inject faults on the top module of the system an will check for fault detection. The fault coverage must be higher than 90%.

II. Circular BIST design

The circular BIST system must be applied to a given circuit, different for each group. The circuit in question is named ”circuito06” and its as follows:

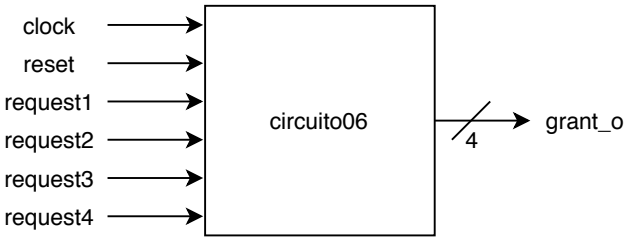

Figure 1: "circuito06" interface.

In order to make it compatible with the BIST scan chain, scan functionality needed to be added. This was done using cadence tools and it added the ”scan_in” and ”mode” inputs and the ”scan_out” output.

LFSR and MISR

In order to generate pseudo-random inputs for the circuit, an external linear feedback shift register (LFSR) is implemented.

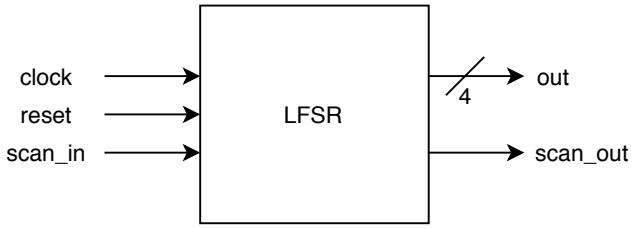

Figure 2: LFSR block diagram.

Since there are 4 inputs for the circuit, the LFSR features 4 flip-flop stages with XOR feedback on the forth and third flip-flops in order to make the output polynomial maximal-length:

$$\begin{aligned} \text{Polynomial} &= x^4 + x^3 + 1 \\ \text{Period} &= 2^4 - 1 = 15 \end{aligned} \tag{1}$$

Finally, to make it scan compatible, the scan_in was added to the XOR chain and a scan_out port using the last flip-flop state was added . This adds another degree of randomness to the system and makes it more likely for the system to detect faults. Additionally a temporary seed input was added for testing purposes.

The LFSR is represented in verilog:

```
module lfsr #
(
    parameter NBIT = 4,
    parameter SEED = 4'b1111
) (
    input clk,
    input rst,
    input scan_in,
    // input [3:0] seed, //this is only used for testing
    output [NBIT-1:0] out,
    output scan_out
);

    reg [NBIT-1:0] dff;

    assign out[0] = dff[0];
    assign out[1] = dff[1];
    assign out[2] = dff[2];
    assign out[3] = dff[3];

    assign scan_out = dff[3];

    always @ (posedge clk) begin
        if(rst) begin
            dff <= SEED;
        end
        else begin
            dff <= {dff[4-2:0], dff[3] ^ dff[2] ^ scan_in};
        end
    end
```

```
endmodule
```

Code excerpt 1: LFSR code.

A test-bench was created to ensure the periodicity of the LFSR. The test-bench is able to set a user determined seed and will cycle the LFSR until the output patterns repeats. When it does, the iteration number is printed and the program execution stops. In this case, the scan input is disabled. Since the LFSR has 4 bits, the number of cycles until repetition should be $2^N - 1 = 2^4 - 1 = 15$

Test result:

```
linuxdev@linuxdev-VirtualBox:/media/sf_GitHub/PTFSE-Classes/Project2$ make lfsr_test
Initial seed:       15
LFSR output: 15
LFSR output: 14
LFSR output: 12
LFSR output: 8
LFSR output: 1
LFSR output: 2
LFSR output: 4
LFSR output: 9
LFSR output: 3
LFSR output: 6
LFSR output: 13
LFSR output: 10
LFSR output: 5
LFSR output: 11
LFSR output: 7
End in        15 iterations
Should be 2^4-1 =      15
```

Code excerpt 2: LFSR test-bench.

With this, we can assure that the LFSR is working and can be used on the top module.

In order to register the output combinations of the circuit, an internal multiple input signature register is also created.

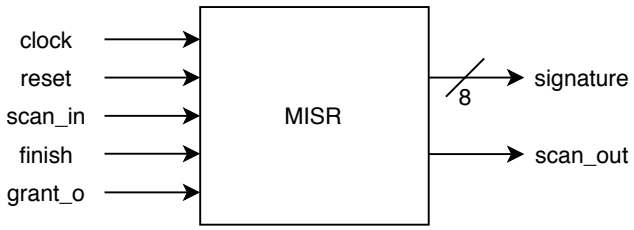

Figure 3: MISR block diagram.

With this the input sequence of the MISR will generate a quasi-unique signature for a BIST run. The length chosen for the MISR was 8, but other sizes were tested, for example, 16, but since it did not improve the fault coverage, the size was reset to 8, since more hardware can be worst for the fault coverage. With 8 flip-flops the XOR chain used in order to make output polynomial maximal-length was:

$$\text{Polynomial} = x^8 + x^6 + x^5 + x^4 + 1 \tag{2}$$

An additional input named ”finish” was added. This serves the purpose of holding the final signature. This input is controlled by the ”finish” output of the controller. The MISR is represented in verilog:

```
// `define misr16
`define misr8


module misr #
(
    `ifdef misr16
    parameter NBIT = 16
    `endif


    `ifdef misr8
    parameter NBIT = 8
    `endif
)(
    input clk,
    input rst,
    input scan_in,
```

```
    input[3:0] grant_o,
    input finish,
    output[NBIT-1:0] signature,
    output scan_out
);

    reg [NBIT-1:0] dff;

`ifdef misr16
<definition for 16 bit misr supressed>
`endif

`ifdef misr8

    assign signature = dff;
    assign scan_out = dff[7];

    parameter seed = 8'b11111111;

    always @ (posedge clk) begin
        if(rst) begin
            dff <= seed;
        end
            dff[0] <= grant_o[3] ^ dff[NBIT-1];
            dff[1] <= grant_o[2] ^ dff[0];
            dff[2] <= grant_o[1] ^ dff[1] ^ dff[NBIT-1];
            dff[3] <= grant_o[0] ^ dff[2] ^ dff[NBIT-1];
            dff[4] <= scan_in ^ dff[3] ^ dff[NBIT-1];
            dff[5] <=           dff[4];
            dff[6] <=           dff[5];
            dff[7] <=           dff[6];
        end
    end

`endif
```

```
endmodule
```

Code excerpt 3: MISR code.

To ensure that the MISR works correctly, a test-bench was also created.
Like the LFSR, the outputs of the MISR were read unitl the pattern repeated. In this case all of the MISR inputs were set to zero. For the MISR the number of cycles sould be $2^N - 1 = 2^8 - 1 = 255$

```
linuxdev@linuxdev-VirtualBox:/media/sf_GitHub/PTFSE-Classes/Project2$ make misr_test
Initial signature:     255
MISR output: 255
MISR output: 227
MISR output: 219
MISR output: 171
<output truncated>
MISR output: 246
MISR output: 241
End in       255 iterations
Should be 2^8-1 =     255
```

Code excerpt 4: MISR test-bench.

In order to make it easier to check for duplicates, a simple python script was created to accomplish such task.

```
import sys
import collections


with open(sys.argv[1], 'r') as f:
    content = f.readlines()


content = [x.strip() for x in content]


duplicates = [item for item, count in collections.Counter(content).items() if count
    > 1]
print("Found duplicates: ")
```

```
print(duplicates)
```

Code excerpt 5: Python code that detects duplicates.

By piping the output of the verilog simulation to the python it is possible to check for duplicates:

```
linuxdev@linuxdev-VirtualBox:/media/sf_GitHub/PTFSE-Classes/Project2$ make misr_test
Found duplicates:
[]
```

Code excerpt 6: Python script output.

With this, we can also assure that the MISR is working as intended and can be used on the top module.

Controller

The controller was left mostly unchanged from the first project, apart from correcting some errors, for example, synthesis related. The ”init” output was used to reset the other modules, alongside with the global reset signal tied with OR logic. The toggle signal was used to control the scan mode of the circuit in order to increase the fault detection ratio.

Wrappers and additional logic

The last module left to be implemented was the mode select multiplexer. This multiplexer serves the purpose of choosing the input of the circuit being it the normal inputs in normal operation or the LFSR outputs in test mode. It’s implementation is rather simple:

```
module lfsrmux #
(
    parameter NBIT = 4
) (
    input [4-1:0] in,
    input [4-1:0] lfsr,
    input mode,

    output reg [4-1:0] outport
);
```

```
    always @(*) begin
        if(mode)
            outport = lfsr;
        else
            outport = in;

    end

endmodule
```

Code excerpt 7: Multiplexer code.

The mode is set by the ”running” signal of the BIST controller.

With all of the modules completed, the interconnections between them were made, additional logic was added for signature checking and for the ”pass_fail” signal. The final implemented system has the following diagram [1]:

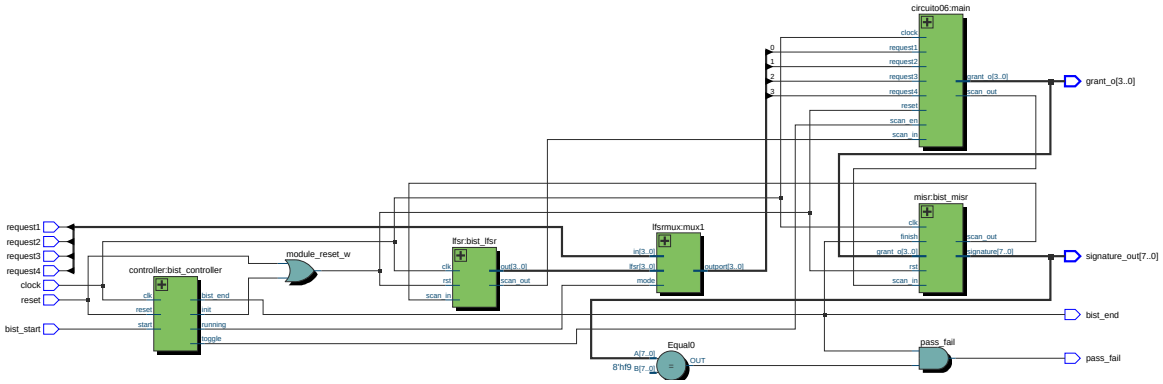

Figure 4: Circular BIST block diagram.

[1] The block diagram is in vector format and can be zoomed-in for better visualization. Intel Quartis was used to generate this block diagram

III. Circular BIST testing and validation

A multi-purpose testbech was created for the system's validation.

A battery of tests included in the test-bench were used in order to check if the system is working is working properly and compliant to the given specifications . These tests are:

- Single run, normal operation
- Multiple runs with reset sequence
- Multiple runs without reset sequence
- Mid start toggle
- Mid reset toggle
- Reset latch test
- Circuit operation with test vectors injection

The test to be run can be selected on the test-bench by setting the appropriate define condition. Only one test should be enabled at a time.

```
//VALIDATION TESTS
`define getfaultcoverage
// `define multiple_runs_reset
// `define multiple_runs
// `define mid_start
// `define mid_reset
// `define reset_latch
// `define vectest
```

Code excerpt 8: Configurable test-bench parameters.

Circuit operation with test vectors injection

The first test was to run the system in normal mode and a set of vectors were injected in order to check if the circuit is still working properly when embedded onto the BIST system.

```
Current vector: 0101
Circuit output: 4
Current vector: 1011
Circuit output: 8
Current vector: 0110
Circuit output: 4
Current vector: 1100
Circuit output: 8
Current vector: 1101
Circuit output: 8
Current vector: 0001
Circuit output: 1
Current vector: 0001
Circuit output: 1
Current vector: 1110
Circuit output: 8
Current vector: 0111
Circuit output: 4
Current vector: 0010
Circuit output: 2
```

Code excerpt 9: Normal operation output.

Single run, normal operation

After this a single run in BIST mode was made to check the final signature in a perfect setup without faults. In our case the final signature is F9 (in hexadecimal format). This signature was updated on the design and another test was made in order to check if the ”pass_fail” signal was working as expected.

```
Output signature: f9
Pass/fail: 1
```

Code excerpt 10: BIST normal operation output.

Multiple runs with reset sequence

On this test, the circuit must perform two runs with a reset sequence in between. The both BIST runs must end with the same result, F9 signature and ”pass_signal” indicating ”pass”.

```
linuxdev@linuxdev-VirtualBox:/media/sf_GitHub/PTFSE-Classes/Project2$ make
    circular_bist_test
Output signature: f9
Pass/fail: 1
Output signature: f9
Pass/fail: 1
```

Code excerpt 11: Multiple runs with reset sequence output.

Multiple runs without reset sequence

Like before, on this test, the circuit must perform two consecutive runs but without a reset sequence in between. The both BIST runs must end with the same result, F9 signature and ”pass_signal” indicating ”pass”.

```
linuxdev@linuxdev-VirtualBox:/media/sf_GitHub/PTFSE-Classes/Project2$ make
    circular_bist_test
Output signature: f9
Pass/fail: 1
Output signature: f9
Pass/fail: 1
```

Code excerpt 12: Multiple runs without reset sequence output.

Mid reset toggle

On this test, the global reset signal is toggle during the BIST operation. In this case the module should stop operation and return to the default state.

Mid start toggle

On this test, the start signal is toggle during the BIST operation. In this case the module should not affect the current operation.

Reset Latch test

This test is used to check for proper operation of the reset latch. This test ensures that when start signal is sampled whilst the reset signal is on, the controller must wait for both

the signals to go low and should only start on the next start signal if the reset signal is not enabled.

IV. Synthesis

Once the circuit was fully tested, the top module was synthesized into one single module. The synthesized circuit was tested locally, and apart from some issues related with the tool-chain used (icarus + gtkwave), the circuit worked as intended.

Full synthesis reports are present alongside this document with the project files.

Synthesis reports were extracted for the original circuit and for the complete top module. Some comparisons can be made:

Gate count and circuit area

Report for the original circuit:

```
  Gate  Instances  Area        Library
-----------------------------------------------
AOI211          2   145.600   c35_CORELIB_TYP
AOI2111         2   182.000   c35_CORELIB_TYP
AOI221          9   819.000   c35_CORELIB_TYP
CLKIN3          4   145.600   c35_CORELIB_TYP
DFC3           13  4022.200   c35_CORELIB_TYP
DFEC1          12  4149.600   c35_CORELIB_TYP
IMUX20          1    91.000   c35_CORELIB_TYP
IMUX21          1    91.000   c35_CORELIB_TYP
INV0            1    36.400   c35_CORELIB_TYP
INV2            2    72.800   c35_CORELIB_TYP
INV3            1    36.400   c35_CORELIB_TYP
NAND22          5   273.000   c35_CORELIB_TYP
NAND31          1    72.800   c35_CORELIB_TYP
NOR21           5   273.000   c35_CORELIB_TYP
NOR31           7   509.600   c35_CORELIB_TYP
NOR40           1    72.800   c35_CORELIB_TYP
OAI2111         1    91.000   c35_CORELIB_TYP
OAI212          9   655.200   c35_CORELIB_TYP
OAI311          1    91.000   c35_CORELIB_TYP
TFC3            1   291.200   c35_CORELIB_TYP
```

```
------------------------------------------------
total         79  12121.200


   Type   Instances  Area  Area %
--------------------------------------
sequential     26  8463.000  69.8
inverter        8   291.200   2.4
logic          45  3367.000  27.8
--------------------------------------
total          79 12121.200 100.0
```

Code excerpt 13: Gate count and circuit area of the given circuit.

And for the full top module:

```
  Gate  Instances  Area        Library
-------------------------------------------------
ADD22          6   873.600   c35_CORELIB_TYP
AOI211         1    72.800   c35_CORELIB_TYP
AOI2111        3   273.000   c35_CORELIB_TYP
AOI221         4   364.000   c35_CORELIB_TYP
AOI311         2   182.000   c35_CORELIB_TYP
CLKIN2         2    72.800   c35_CORELIB_TYP
CLKIN3         2    72.800   c35_CORELIB_TYP
DF3           27  7371.000   c35_CORELIB_TYP
DFS1           1   364.000   c35_CORELIB_TYP
DFSC1         13  4968.600   c35_CORELIB_TYP
DFSEC1        12  5241.600   c35_CORELIB_TYP
IMUX20        23  2093.000   c35_CORELIB_TYP
IMUX21         4   364.000   c35_CORELIB_TYP
IMUX30         1   182.000   c35_CORELIB_TYP
INV2          18   655.200   c35_CORELIB_TYP
INV3           9   327.600   c35_CORELIB_TYP
MUX22          5   546.000   c35_CORELIB_TYP
NAND22        27  1474.200   c35_CORELIB_TYP
NAND30         1    72.800   c35_CORELIB_TYP
NAND31         4   291.200   c35_CORELIB_TYP
```

```
NAND40          1     91.000   c35_CORELIB_TYP
NOR20           1     54.600   c35_CORELIB_TYP
NOR21          22   1201.200   c35_CORELIB_TYP
NOR31          11    800.800   c35_CORELIB_TYP
NOR40           4    291.200   c35_CORELIB_TYP
OAI211          2    145.600   c35_CORELIB_TYP
OAI2111         4    364.000   c35_CORELIB_TYP
OAI212          5    364.000   c35_CORELIB_TYP
OAI222          6    546.000   c35_CORELIB_TYP
OAI311          1     91.000   c35_CORELIB_TYP
TFSC1           1    345.800   c35_CORELIB_TYP
XNR21           2    218.400   c35_CORELIB_TYP
XNR31           1    200.200   c35_CORELIB_TYP
XOR31           3    600.600   c35_CORELIB_TYP
-----------------------------------------------
total         229  31176.600


   Type   Instances  Area  Area %
--------------------------------------
sequential      54 18291.000 58.7
inverter        31  1128.400  3.6
logic          144 11757.200 37.7
--------------------------------------
total          229 31176.600 100.0
```

Code excerpt 14: Gate count and circuit area of the full BIST system.

That make out an increase of 157.21%.

After the circuit was successfully synthesised, the fault simulator was used to check for the current fault detection coverage:

```
Stuck-At (0/1) Fault Table
                      Total #       Prime #
Untestable                0             0
Detected                344           344
Potentially_detected      7             7
Undetected               36            36
```

```
Unobserved_detected       0             0
Unobserved_undetected     0             0
Dangerous_detected        0             0
Dangerous_undetected      0             0
Not_simulatable         173           173
Not_injected             32            32
Total                   592           592
```

Code excerpt 15: Fault simulator results.

Using the following formula, it is possible to compute the fault coverage:

$$\frac{\text{Detected} + \text{Potentially detected}}{\text{Detected} + \text{Potentially detected} + \text{Undetected}} \times 100 = \frac{344 + 7}{344 + 7 + 36} \times 100 = 90.7\% \qquad (3)$$

With this we can confirm that the developed system achieves the required 90% fault coverage.

V. Conclusion

This project objective was to learn how to program and develop a Circular BIST to a given circuit which could also work with the controller developed for the first project of this course. Several tests were performed to the system and its modules, and corrections were made to the controller to guarantee this system works as it was intended to.

The fault coverage surpasses the minimum requirement of 90%.

Most of the difficulties of this project landed upon a simulation difference between the toolchains used, icarus vs cadence. The system that was proven to be working locally did not work after it was synthesized. Our team spent a large amount of time in order to track down the issue which led us to have not much time left to improve the performance of the system. But overall, the developed system worked as expected and accomplishes all of the requirement set by the teacher.

References

[1] Verilog Tutorial (Course slides)

`https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382240212/My%20Verilog%20Tutorial.pdf`

[2] Xillinx LFSR maximum length polynomial table

`https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf`