EC310: Testing and Diagnosis of Digital System Design

2020 Even Semester Project

Hardware Design of Booth's Multiplier circuit with Built-in Self-Test Feature

Prepared for:

Ms. Kriti Suneja

Assistant Professor,

Dept. of ECE, DTU

Prepared by:

Abhishek Chaurasia	(2K17/EE/08)
Abhishek Choudhary	(2K17/EE/09)
Animesh Kulshreshtha	(2K17/EE/38)
Avisekh Ghosh	(2K17/EE/62)

ABSTRACT

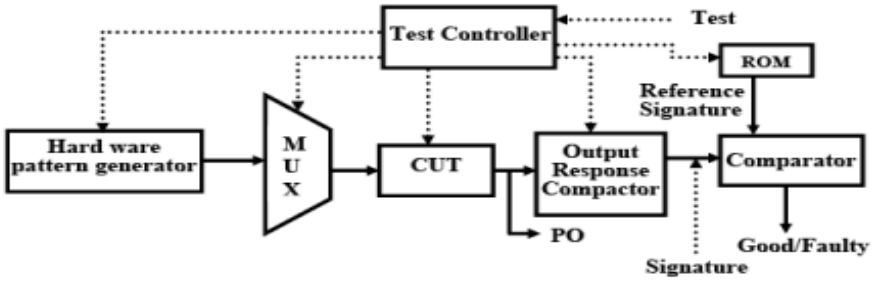
Very Large-Scale Integration (VLSI) has made a dramatic impact on the growth of integrated circuit technology. It has not only reduced the size and the cost but also increased the complexity of the circuits. The positive improvements have resulted in significant performance/cost advantages in VLSI systems.

There are, however, potential problems which may retard the effective use and growth of future VLSI technology. Among these is the problem of circuit testing, which becomes increasingly difficult as the scale of integration grows. Because of the high device counts and limited input/output access that characterize VLSI circuits, conventional testing approaches are often ineffective and insufficient for VLSI circuits.

Built-in self-test (BIST) is a commonly used design technique that allows a circuit to test itself. BIST has gained popularity as an effective solution over circuit test cost, test quality and test reuse problems. In this paper we are presenting implementation of BIST on Booth's Multiplier.

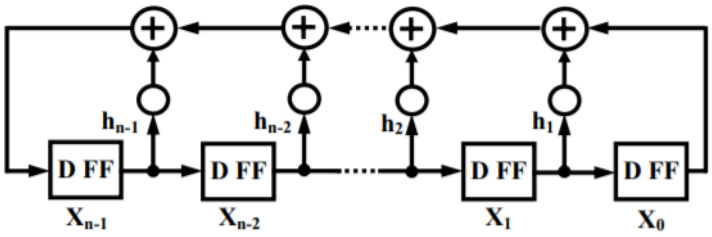
Built-in Self-Test

Built-in Self-Test (or BIST) is a DFT technique in which testing is accomplished through built in hardware. The basic idea is to have a VLSI chip that tests itself. BIST is a design-for-testability technique that places the testing functions physically with the circuit under test (CUT). The basic BIST architecture requires the addition of three hardware blocks to a digital circuit: a test pattern generator, a response analyser, and a test controller. The test pattern generator generates the test patterns for the CUT. Examples of pattern generators are a ROM with stored patterns, a counter, and a linear feedback shift register (LFSR). A typical response analyser is a comparator with stored responses or an LFSR used as a signature analyser. It compacts and analyses the test responses to determine correctness of the CUT. A test control block is necessary to activate the test and analyse the responses. However, in general, several test-related functions can be executed through a test controller circuit.



LFSR: Linear Feedback Shift Register

The LFSR is just a register formed from standard flip-flops, with the outputs of selected flip-flops being fed back to the shift register's inputs. When used as a test generation, an LFSR is about to cycle rapidly through an outsized number of its states. These states, whose choice and order depend upon the planning parameters of the LFSR, define the test patterns. during this mode of operation, an LFSR is seen as a source of (pseudo) random tests that are, in theory, applicable to any fault and circuit types. There are n flip-flops (X_{n-1}.....X_0) and this is called n-stage LFSR. It can be a near-exhaustive test pattern generator as it cycles through 2^n-1 states excluding the zero state.



Standard LFSR model

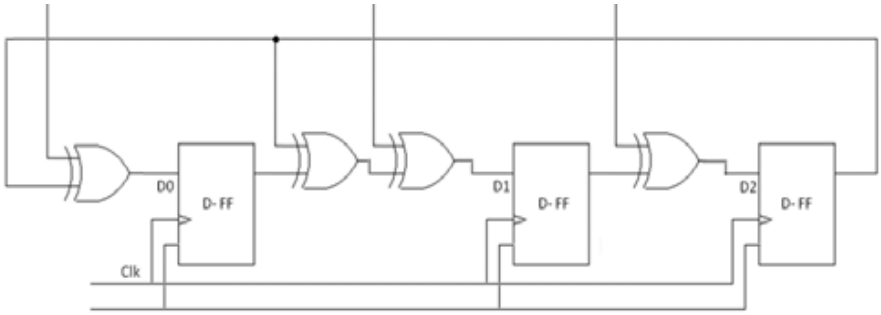
Output Response Compactor

There is huge amount of data involved in CUT response. It is uneconomical to store all these responses on the chip and compare during the BIST operation, therefore it is mandatory to compact responses before compaction. Large volume of response is compacted into a signature before comparison. The signature is stored on the chip. There are two approaches available.

1. In first approach we compact a single serial bit stream into a LFSR based compactor, called signature analyser.
2. In the second approach several bit streams are compacted into a special LFSR based compactor called multi-input signature register (MISR).

MISR: Multiple-Input Signature Register

A serial-input signature register can only be used to test logic with a single output. The idea of a serial input signature register can be extended to multiple-input signature register (MISR). There are several ways to connect the inputs of LFSRs to form an MISR. Since the XOR operation is linear and associative, (A XOR B) XOR C = A XOR (B XOR C), as long as the result of the additions are the same then the different representations are equivalent. If we have an n -bit long MISR we can accommodate up to n inputs to form the signature. If we use m < n inputs we do not need the extra XOR gates in the last n – m positions of the MISR. MISR reduce the amount of hardware required to compress a multiple bit stream. MISR circuit is implemented using a memory already existing in a circuit to be tested.



MISR model

CUT (Circuit Under Test)

Circuit under test is a 4-bit Booth's Multiplier circuit. It is a very fast multiplication algorithm also capable of signed number multiplication. The Booth's algorithm the state machine used is described in the later sections.

Higher bit Booth's Multiplier can also be used, but to keep the testing simple, 4-bit Booth's Multiplier is used.

LFSR and MISR used in the design

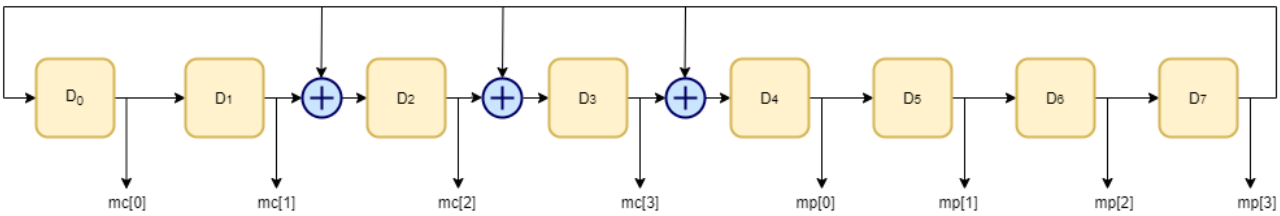
4-bit Booth's Multiplier has two inputs of 4-bit each. Hence a an LFSR capable of generating at least 8-bit pattern is required.

The output of 4-bit Booth's Multiplier is 8-bits, so there are 8 output patterns. So, at least 8-bit MISR is required for compaction.

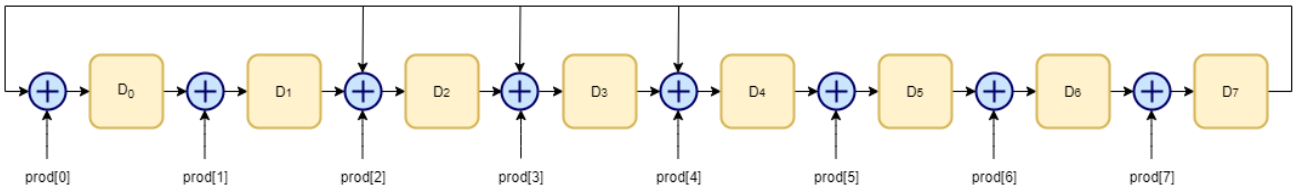
An 8-bit internal XOR (type 2/ modular) LFSR and MISR of same characteristic polynomial is used in the design.

Characteristic Polynomial: x^8 + x^4 + x^3 + x^2 + 1

This is a **primitive polynomial**. It can generate all patterns from 1 to 2^n-1 exhibited by an 8-bit register except zero. Moreover, the Golden Section will be more unique when using a primitive polynomial MISR as compactor and probability of aliasing will also be minimal.



LFSR output pattern into *mc* (multiplicand) and *mp* (multiplier) of multiplier



prod output of multiplier into MISR compactor

Software Environment

The complete project is done in Xilinx Vivado Software Environment using Verilog as HDL. The various tools used include the text editor, Vivado Simulator, Synthesis and RTL Analysis. It is recommended to use the same software for evaluation purposes.

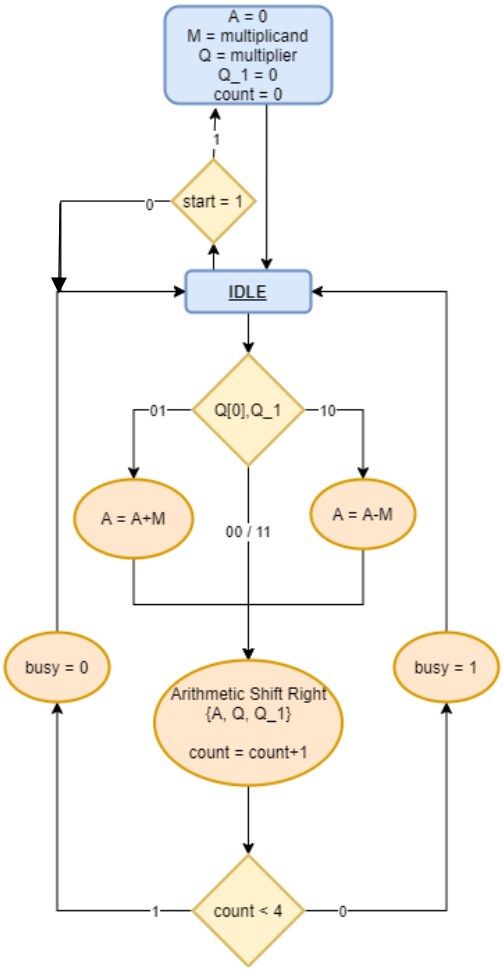
The Design

Source code of the design is provided in the **Appendix A.** The top module consists of two sub modules, *mult* and *tester*. The design hierarchy is provided on the left.

When operating the multiplier in normal mode, the *tester* circuit is completely removed from the picture using multiplexers. Clock Gating technique is used when tester is not used to decrease the dynamic power.

When test mode is set, the I/O ports of multiplier is completely disconnected from user's control, and the tester gets invoked to control the multiplier and begin testing.

The whole circuit operates on two controllers, Booth's Algorithm Controller and BIST Controller. Hence there are two different state machines. In test mode, both the controller functions together by virtue of a **Handshaking Protocol** using various control signals between them.



Booth's Algorithm for Multiplier Circuit

1. Initially the multiplier is in undefined state. On *start = 1*, the multiplier resets *A, Q_1* and *count* to zero.
2. Next state is **Idle**, now the *start* signal needs to be zero otherwise it will again move to **reset state** and ask for parameters multiplier and multiplicand again.
3. Check for *{Q[0], Q_1}*. If *{0,1}* move to step 4, if {1,0} move to step 5, else skip to step 6.
4. Assign *A = A + M*. Move to step 6.
5. Assign *A = A - M*. Move to step 6.
6. Arithmetic Shift Right, the triplet *{A, Q, Q_1}*. Decrement *count*.
7. Check for *count* < 4, if true set *busy=0* otherwise set *busy=1*. Move to **Idle** state and continue to step 3.

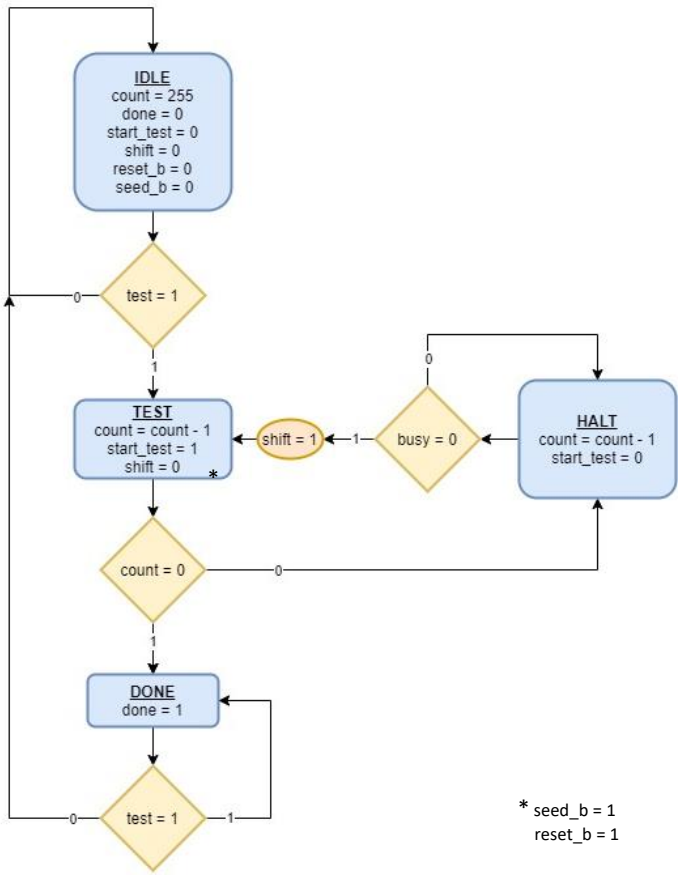
Note: All of the steps from 3 to 7 are done in 1 clock cycle. It reduces the no. of states and increases the speed of FSM. But it may impose high timing constraints.

BIST Algorithm for tester circuit

1. Reset all the signals to default zero value in **Idle State**. Reset to initial seed in LFSR and MISR to 1111_1111 and 0000_0000 by applying *seed_b = 0* and *reset_b = 0* respectively. The LFSR must be seeded to a non-zero value initially to generate pattern.
2. On *test = 1*, move to next state i.e. **Test State.** Issue *start = 1* on Booth's Multiplier. Set *shift = 0* to pause the pattern generator LFSR and compactor MISR. Set *reset_b* and *seed_b* as 1, as LFSR and MISR are already seeded.
3. If count = 0 jump to step 4, else jump to step 6.
4. This is **Halt State**, the state machine remains in this state for most amount of time. Set the start *signal = 0* to begin multiplying.
5. If *busy = 0*, move to step 2 with *shift = 1* as mealy output. This signal resumes the LFSR generator and MISR compactor to shift and accept new values. Else stay in the current state.
6. This is **Done State**, the testing is completed hence set the *done* signal as 1.

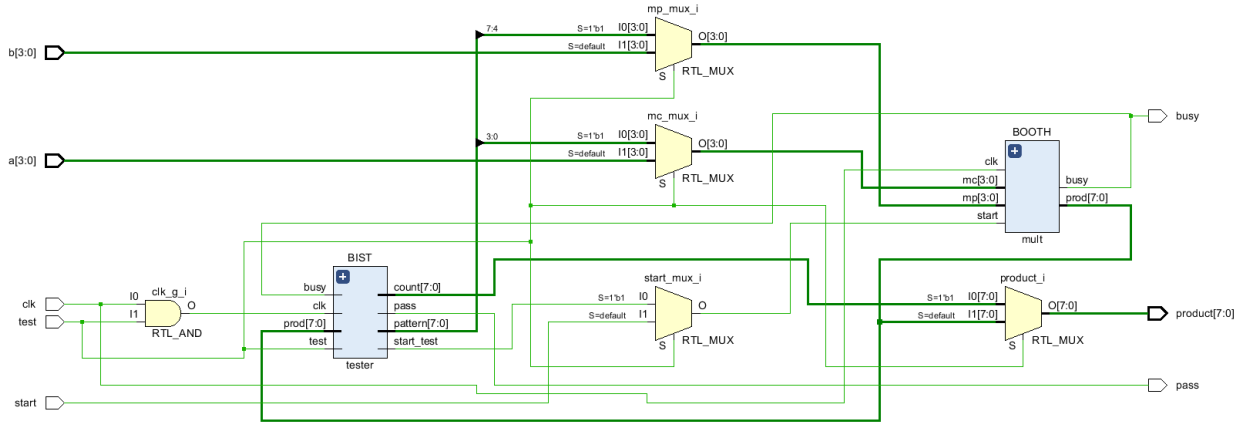
After the test has been completed, the *done signal* is set to one and the obtained signature in MISR is compared with the valid signature, i.e. **sign[7:0] = 0101_1111B or 95D**. This valid signature can't be calculated manually, because test pattern polynomial is too large with maximum degree = 255. Hence the valid signature is obtained from the waveform in Vivado Simulator when there is not fault in the circuit. Later this signature is compared with the obtained signature in faulty circuit.

When the comparison is true, the *pass* signal is set 1 by the tester to indicate the circuit is not faulty.

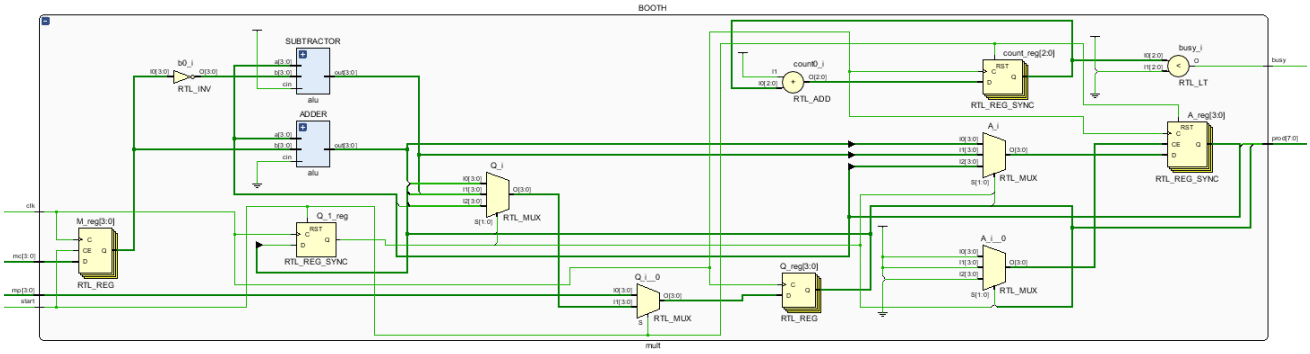


Elaborated Design

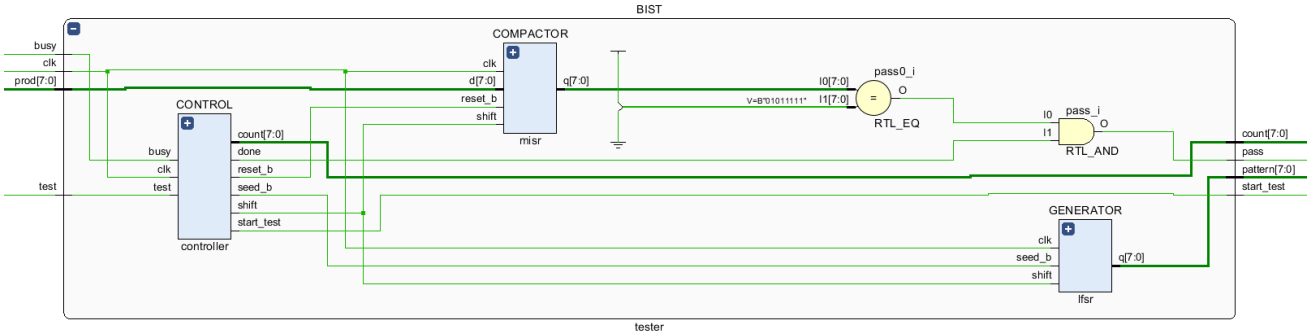
Top Module



Booth's Multiplier

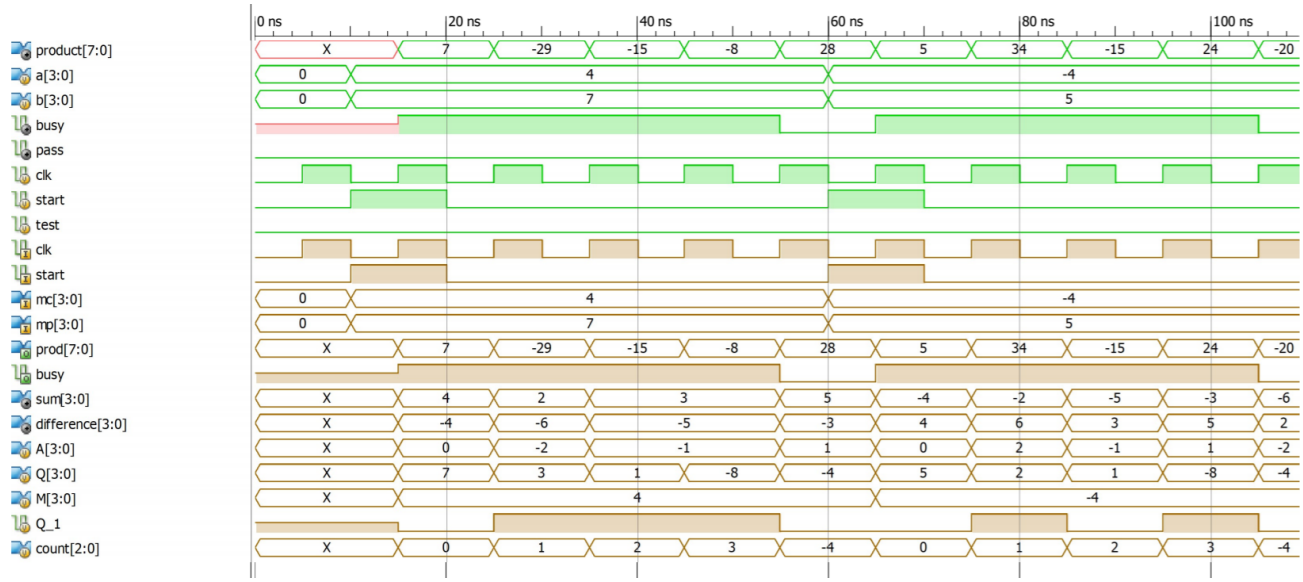


BIST Tester



Simulation Waveforms

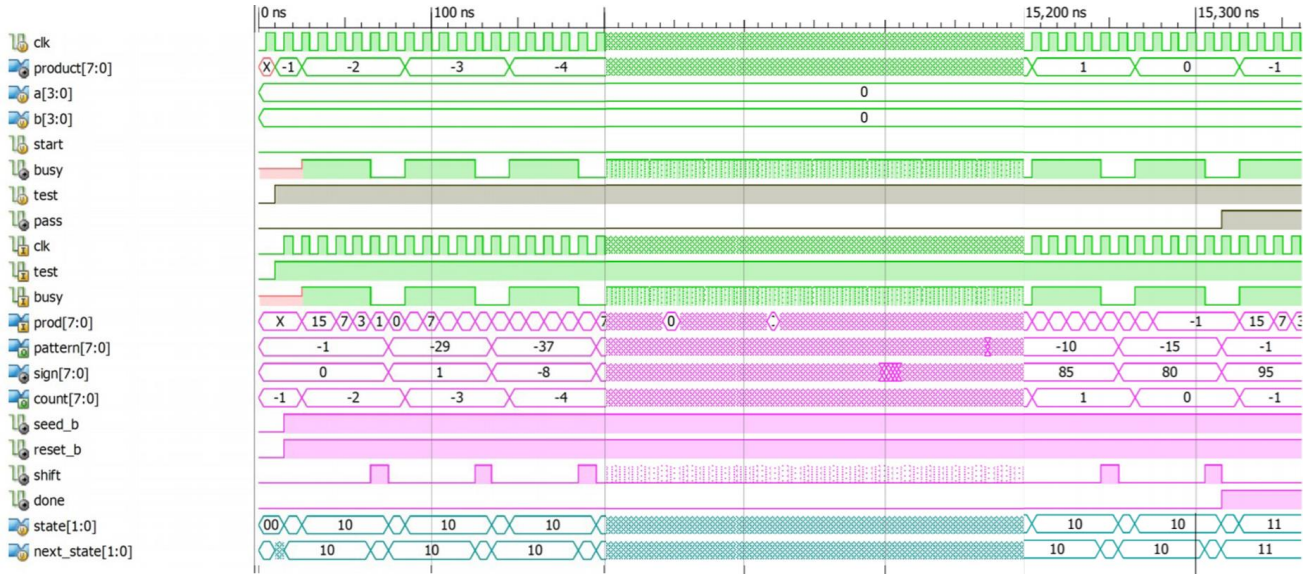
Testbench waveform for Booth's Multiplier



Testbench code for above simulation can be found at **Appendix B1**. The signals in green are the i/o signals of the top UUT i.e. *multiplier*. The signals in golden are some important signals in the *mult* UUT. The inputs *a* and *b* are loaded with respective values and the *start* signal pulse is given to start the multiplication. Since the Booth's multiplier takes O(n) time units, where n in the type of Booth's Multiplier (here n = 4), hence the multiplication takes 40 ns to complete. In the while, the *busy* signal is et 1 to indicate the multiplication is not complete. Just after *busy* signal gets low, we get the desired product as result. The testbench also yields the following result in TCL Console.

```
product:     x busy: x at time=              5000
first example: a = 4 b = 7
product:     7 busy: 1 at time=             15000
product:   -29 busy: 1 at time=             25000
product:   -15 busy: 1 at time=             35000
product:    -8 busy: 1 at time=             45000
product:    28 busy: 0 at time=             55000
first example done
second example: a = -4 b = 5
product:     5 busy: 1 at time=             65000
product:    34 busy: 1 at time=             75000
product:   -15 busy: 1 at time=             85000
product:    24 busy: 1 at time=             95000
product:   -20 busy: 0 at time=            105000
second example done
```

Testbench waveform for BIST Tester Circuit with properly functioning Booth's Multiplier



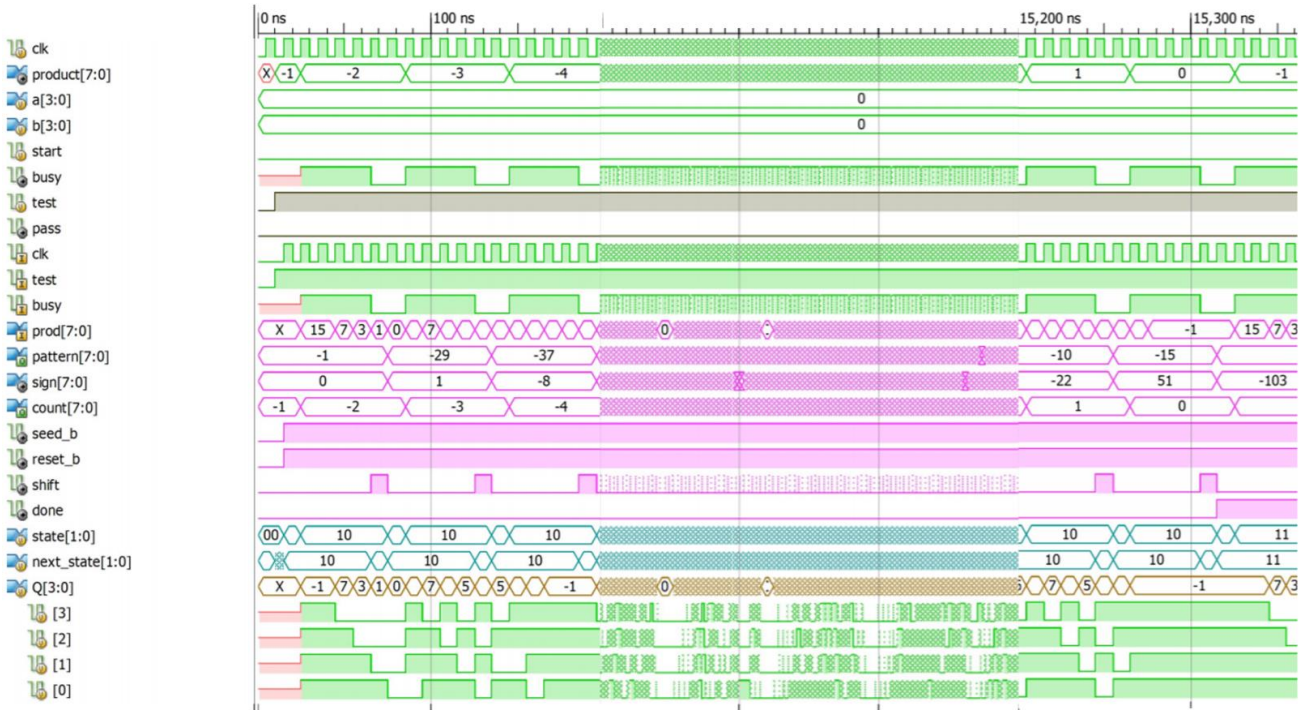
Testbench code for above simulation can be found at **Appendix B2**. The signals in green are the i/o signals of the top UUT i.e. *multiplier*. The signals in pink are some important signals in the *tester* UUT. The signals in blue indicates state and next state of the BIST *Controller* UUT. The test and pass signals are colored grey to highlight them. When the test signal is high, the multiplier inputs and outputs are completely disconnected from user's control and are connected to the BIST tester. The tester follows the state machine described earlier. The output *product* signal is connected to *count* of the BIST controller. This provides an extra feature to display the time remaining for the test to complete to the user. The product signal down counts from 255 to 0 in (unsigned decimal form). When count value reaches 1111_1111B or -1, it indicates the testing is complete. Since the *pass* signal is 1 in the waveform, this means the signature is matched with the valid signature and test is passed. The tester is reset when *test = 0*. Since same *mult* module is used in this testing purpose, it will obviously *show pass = 1* whenever testing is complete, because no internal modification has been done in the *mult* module.

The time taken by this test to large for the waveform to fit in the report, hence the middle part of waveform has been shrinked to show to results.

Since there are 2^n-1 total patterns generated by the LFSR generator (excluding pattern 0000_0000B), the time taken to complete the test is approximately $O(2^{2n}-1)$ units, where n is the type of multiplier (n=4 for 4 bit multiplier). Each multiplication takes O(n) time units. And in test mode, 2 more time units are required for state transition of BIST controller. Hence total time for test can be calculated by using formula $(2^{2n}-1)(n+2)$ time units or **$(2^{2n}-1)(n+2)*10$** ns for 100 MHz clock as used in the simulation or typical FPGA considering zero timing violations.

Time taken for test to complete for 4-bit Booth's Multiplier = $(2^{2*4}-1)(4+2)*10$ ns = 15300 ns = 15.3 µs

Testbench waveform for BIST Tester Circuit with faulty Booth's Multiplier



Testbench code for above simulation can be found at **Appendix B3**. Using a normal testbench on the circuit will always show *pass = 1* when testing using BIST feature, as the circuit is never faulty in frontend pre-production process. Hence the testbench in Appendix B3 is empty with no stimulus. Faults must be created in *mult* module intentionally. To create stuck-at-0 or stuck-at-1, any individual signals inside *mult* module can be forced to 0 or 1 respectively using Vivado Simulator GUI. This task can also be automated using TCL scripts in the console. The used TCL scripts for simulation provided in **Appendix C**. The signals in green are the i/o signals of the top UUT i.e. *multiplier*. The signals in pink are some important signals in the *tester* UUT. The signals in blue indicates *state* and *next state* of the BIST *Controller* UUT. The *test* and *pass* signals are colored grey to highlight them. At last, an extra signal *Q[3:0]* with golden colour is added to waveform. This signal shows the value of Q register in *mult* module. Modifying or forcing this signal will interfere with the results of the product and hence change the final signature. In this case the signal *Q[1]* is forced to stuck-at-1 from time interval 6020 ns to 6140 ns (just for 120 ns). The force is removed and then testbench is run normally.

The product signal is down counting as always. When BIST testing is complete the product signal reaches 0 and the shows 1111_1111B or -1. At this moment the *pass* signal is still low, indicating that the circuit is faulty. The test takes same time (i.e. 15300 ns) to complete.

Conclusions

Probability of aliasing

It is the probability of that fault has occurred but not detected. It can be determined easily using the formula,

Probability of aliasing = $2^{-n} = 2^{-8} = 0.0039$ (n = 8, size of Compactor)

Utilisation

The hardware overhead or utilization can be manually calculated by analysing the source code in **Appendix A.**

The BIST multiplier circuit consists of 3 registers (A, Q, M) contributing to 3n FFs. The count register is sized ($\log_2 n + 1$). One Q_1 reg takes 1 FF more.

Booth's Multiplier hardware utilisation (FF) = $3n + (\log_2 n + 1) + 1 = 3n + \log_2 n + 2$

Similarly, for tester circuit compactor and generator contributes to 2*(2n) FFs. The counter in the controller needs 2n FFs. Two more FFs are required for 4-states.

BIST Tester hardware utilisation (FF) = 2*(2n) + 2n + 2 = 6n + 2

Total hardware utilisation (Booth + tester) = $(3n + \log_2 n + 2) + (6n + 2) = 9n + \log_2 n + 4$
(for an n-bit Multiplier Circuit)

Hardware (FF) utilisation without tester = $3n + \log_2 n + 2$ = 16
Hardware (FF) utilisation without tester = $9n + \log_2 n + 4$ = 42 (n = 4)

This can be verified using Utilization Reports from Vivado Synthesis Tool. The following table describes the utilization of various resources.

Resource	Utilization (without BIST)	Utilization (with BIST)	Times Increase
LUT	13	45	3.46x
FF	16	42	2.62x
IO	19	21	1.11x
BUFG	1	1	1.00x

IO and BUFG is irrelevant and LUTs can be assumed to be proportionate to FF consumption. This simplifies the calculation the hardware overhead for the tester.

Type of Multiplier	FF Utilization (without BIST)	FF Utilization (with BIST)	Times Increase
Booth 2-bit	9	23	2.55x
Booth 4-bit	16	45	2.81x
Booth 8-bit	29	79	2.72x
Booth 16-bit	54	152	2.81x
Booth 32-bit	103	297	2.88x

It can be concluded that incorporating BIST tester for a Booth's Multiplier will consume 2.7x more hardware resources. Hence percentage increase in resources almost remains constant with size of multiplier. Other parameters like static and dynamic power will also show similar characteristics.

Test Duration

The test duration can be calculated using the previous formula derived in Simulation Section.

Test Time = $(2^{2n} - 1)(n + 2) * 10$ ns (for 100 MHz clock)

Type of Multiplier	Test Time	
	(ns)	(- units)
Booth 2-bit	600	600 ns
Booth 4-bit	15300	15.3 µs
Booth 8-bit	6553500	6.55 ms
Booth 16-bit	7.731×10^{11}	12 min 53 secs
Booth 32-bit	6.172×10^{21}	195.7 millenniums

Hence it is very impractical to incorporate BIST testing beyond a 16-bit multiplier circuit.

References

1. Hardware Modelling using Verilog, Prof-I. Sengupta, IT Kharagpur, NPTEL Courseware.
2. IJERT Special issue 2015, B. John and C. Mathew, Design and Implementation of Built in Self-Test (BIST) for VLSI circuit using Verilog
3. IEEE Design and Test of Computers,1993, Vishwani D Agarwal, Charles R Kime, Kewal K Saluja-A Tutorial on Built-in Self-Test
4. https://www.partow.net/programming/polynomials/index.html
5. Testing and Diagnosis of Digital Design Class Notes.

Source code and Testbenches

Although, the source code and testbenches are provided in the Appendix Section. For evaluation or prototyping purposes, it is recommended to import the codes from the following GitHub Repository to prevent any syntax errors.

https://github.com/avisekh007/booth-multiplier-bist

APPENDIX A: SOURCE CODE

Top Module (multiplier)

```
module multiplier(
    input clk,
    input [3:0] a, b,
    input start, test,
    output [7:0] product,
    output busy, pass
    );

          wire [3:0] mc_mux, mp_mux;
          wire [7:0] pattern, count, prod;

          // Booth's Multiplier Module
          mult BOOTH (
                 .clk(clk),
                 .start(start_mux),
                 .mc(mc_mux),
                 .mp(mp_mux),
                 .prod(prod),
                 .busy(busy)
          );

          // Built-in Self-Test Module
          tester BIST (
                 .clk(clk_g),
                 .test(test),
                 .busy(busy),
                 .prod(prod),
                 .start_test(start_test),
                 .pass(pass),
                 .pattern(pattern),
                 .count(count)
          );

          // If test=1, connect the tester module to the mult module
          assign mc_mux    = (test) ? pattern[3:0] : a,
                 mp_mux    = (test) ? pattern[7:4] : b,
                 start_mux = (test) ? start_test   : start,
                 product   = (test) ? count        : prod;

          assign clk_g = clk && test;  //clock gating to reduce dynamic power dissipation when
tester is not in use

endmodule
```

Booth's Multiplier and ALU

```
module mult(
          input clk, start,
          input [3:0] mc, mp,  //multiplicand and multiplier
          output [7:0] prod,   //product
          output busy          //when busy=1, result is not ready
          );

         reg [3:0] A, Q, M;
         reg Q_1;
         reg [2:0] count;
         wire [3:0] sum, difference;

         always @(posedge clk) begin
           if(start) begin   //reset all registers and initilise M and Q on start
               A <= 4'b0;
               M <= mc;
               Q <= mp;
               Q_1 <= 1'b0;
               count <= 3'b0;
           end

            else begin
               case({Q[0], Q_1})  //Booth's Algorithm
                2'b0_1 : {A, Q, Q_1} <= {sum[3], sum, Q};  //arithmetic shift left {A, Q, Q_1}
                2'b1_0 : {A, Q, Q_1} <= {difference[3], difference, Q};
                default: {A, Q, Q_1} <= {A[3], A, Q};
               endcase
                 count <= count + 1'b1;
            end
         end

         // Instantiation of alu as adder and subtractor
         alu ADDER (A, M, 1'b0, sum);
         alu SUBTRACTOR (A, ~M, 1'b1, difference);

         assign prod = {A, Q};
         assign busy = (count < 4);  //4-bit multiplier completes the opearation on 4th count,
make busy low on 4th count

endmodule

/* ALU as an adder, but capable of subtraction:
   Subtraction means adding the two's complement,
   a - b = a + (-b) = a + (inverted b + 1)
   The 1 will be coming in as cin (carry-in) */

module alu(
         input [3:0] a, b,
         input cin,
         output [3:0] out
         );

         assign out = a + b + cin;

endmodule
```

BIST Tester Top Module

```
module tester(
          input clk,
          input test, busy,
          input [7:0] prod,
          output start_test, pass,
          output [7:0] pattern, count
     );

     wire [7:0] sign;

     //Input Pattern Genarator
     lfsr GENERATOR (
                 .clk(clk),
                 .seed_b(seed_b),
                 .shift(shift),
                 .q(pattern)
          );

     //Output Response Compactor
     misr COMPACTOR (
                 .clk(clk),
                 .reset_b(reset_b),
                 .shift(shift),
                 .d(prod),
                 .q(sign)
          );

     controller CONTROL (
                 .clk(clk),
                 .test(test),
                 .busy(busy),
                 .seed_b(seed_b),
                 .reset_b(reset_b),
                 .shift(shift),
                 .start_test(start_test),
                 .done(done),
                 .count(count)
          );

      assign pass = (sign == 8'b0101_1111) && done;
      //Signature is compared with the valid one when the testing is complete.

endmodule
```

LFSR Pattern Generator and MISR Response Compactor

```
module lfsr(
    input clk, seed_b, shift,
    output reg [7:0] q
    );

        //Characteristic Polynomial : x^8 + x^4 + x^3 + x^2 + 1
        always @(posedge clk, negedge seed_b) begin
                if(~seed_b)
                        q <= 8'b1111_1111;  //initial seed != 0
                else if(shift) begin
                        q[0] <= q[7];
                        q[1] <= q[0];
                        q[2] <= q[1]^q[7];
                        q[3] <= q[2]^q[7];
                        q[4] <= q[3]^q[7];
                        q[5] <= q[4];
                        q[6] <= q[5];
                        q[7] <= q[6];
                end
        end

endmodule
```

```
module misr(
    input clk, reset_b, shift,
        input [7:0] d,
    output reg [7:0] q
    );

        //Characteristic Polynomial : x^8 + x^4 + x^3 + x^2 + 1
        always @(posedge clk, negedge reset_b) begin
                if(~reset_b)
                        q <= 8'b0;  //initial seed = 0
                else if(shift) begin
                        q[0] <= q[7]^d[0];
                        q[1] <= q[0]^d[1];
                        q[2] <= q[1]^q[7]^d[2];
                        q[3] <= q[2]^q[7]^d[3];
                        q[4] <= q[3]^q[7]^d[4];
                        q[5] <= q[4]^d[5];
                        q[6] <= q[5]^d[6];
                        q[7] <= q[6]^d[7];
                end
        end


endmodule
```

BIST Controller

```
module controller(
    input clk,
    input test, busy,
    output reg seed_b, reset_b, shift,
    output reg start_test, done,
    output reg [7:0] count
    );

    parameter IDLE  = 2'b00, // not testing
              TEST  = 2'b01, // multiplier ready
              HALT  = 2'b10, // multiplier not ready
              DONE  = 2'b11; // testing finished

    reg [1:0] state, next_state;

    always @(posedge clk, negedge test) begin
        if (~test) begin  // disable testing on test signal low
            state <= IDLE;
            count <= 255; // 2^8-1 = 255 combinations of pattern to be checked
        end
        else begin
            state <= next_state;
            if(state == TEST) count <= count - 1'b1; // decrease count on every sucessful compaction
        end
    end

    // next-state combinational logic
    always @(*) begin
        next_state = state;
        case (state)
            IDLE : if(test) next_state = TEST;
            TEST : begin
                      if(count==0) next_state = DONE; // finish testing when all 255 combinatons of pattern
are checked
                      else next_state = HALT; // halt and wait for multiplier to get ready
                   end
            HALT : if(busy==0) next_state = TEST; // resume testing when multiplier is ready
        endcase
    end

    // output combinational logic
    always @(*) begin
        reset_b    = 1'b1;  // default value for all control signals
        seed_b     = 1'b1;
        done       = 1'b0;
        start_test = 1'b0;
        shift      = 1'b0;

        case (state)
            IDLE  : begin
                        reset_b = 1'b0;
                        seed_b  = 1'b0;
                    end
            TEST  : start_test = 1'b1;  // start testing as multilpier is ready
            HALT  : if(busy==0) shift = 1'b1; // shift LFSR & MISR when multiplier operation is finished
            DONE  : done = 1'b1; // indicates testing is finished
        endcase
    end

endmodule
```

Appendix B: Testbench Code

B1: Testbench for normal multiplier functionality

```
`timescale 1ns / 1ps

module multiplier_tb;

	// Inputs
	reg clk;
	reg [3:0] a;
	reg [3:0] b;
	reg start;
	reg test;

	// Outputs
	wire [7:0] product;
	wire busy;
	wire pass;

	// Instantiate the Unit Under Test (UUT)
	multiplier uut (
		.clk(clk),
		.a(a),
		.b(b),
		.start(start),
		.test(test),
		.product(product),
		.busy(busy),
		.pass(pass)
	);

	always #5 clk = ~clk;

	always @(posedge clk) $strobe("product: %d busy: %d at time=%t", $signed(product),
busy, $stime);

	initial begin
		// Initialize Inputs
		clk = 0;
		a = 0;
		b = 0;
		start = 0;
		test = 0;
		// Add stimulus here

		#10;
			$display("first example: a = 4 b = 7");
			a = 4; b = 7; start = 1; #10 start = 0;
			#40 $display("first example done");
			$display("second example: a = -4 b = 5");
			a = -4; b = 5; start = 1; #10 start = 0;
			#40 $display("second example done");
			$finish;
	end

endmodule
```

B2: Testbench for Tester Circuit with properly functioning Booth's Multiplier

```
`timescale 1ns / 1ps

module multiplier_tb;

	// Inputs
	reg clk;
	reg [3:0] a;
	reg [3:0] b;
	reg start;
	reg test;

	// Outputs
	wire [7:0] product;
	wire busy;
	wire pass;

	// Instantiate the Unit Under Test (UUT)
	multiplier uut (
		.clk(clk),
		.a(a),
		.b(b),
		.start(start),
		.test(test),
		.product(product),
		.busy(busy),
		.pass(pass)
	);

	always #5 clk = ~clk;

	initial begin
		// Initialize Inputs
		clk = 0;
		a = 0;
		b = 0;
		start = 0;
		test = 0;
		// Add stimulus here

		  #10;
		  test = 1;

		  #100;
		  wait(product == 8'b1111_1111);

		  #10;
		  if(pass == 1)
			$display("Test Passed!");
		  else
			$display("Test Failed!");

		  #30;
		  test = 0;

	$finish;

	end

endmodule
```

B3: Testbench for Tester Circuit with faulty Booth's Multiplier

```
1   `timescale 1ns / 1ps
2
3   module multiplier_tb;
4
5           // Inputs
6           reg clk;
7           reg [3:0] a;
8           reg [3:0] b;
9           reg start;
10          reg test;
11
12          // Outputs
13          wire [7:0] product;
14          wire busy;
15          wire pass;
16
17          // Instantiate the Unit Under Test (UUT)
18          multiplier uut (
19                  .clk(clk),
20                  .a(a),
21                  .b(b),
22                  .start(start),
23                  .test(test),
24                  .product(product),
25                  .busy(busy),
26                  .pass(pass)
27          );
28
29          always #5 clk = ~clk;
30
31          initial begin
32                  // Initialize Inputs
33                  clk = 0;
34                  a = 0;
35                  b = 0;
36                  start = 0;
37                  test = 0;
38                  // Add stimulus here
39          end
40
41  endmodule
```

Appendix C: TCL Commands for forcing multiplier to behave faulty

```
restart
run 10 ns
add_force {/multiplier_tb/test} -radix bin {1 1ps}
run 10 ns
run 6000 ns
add_force {/multiplier_tb/uut/BOOTH/Q[1]} -radix bin {1 1ps}
run 60 ns
run 60 ns
remove_forces { {/multiplier_tb/uut/BOOTH/Q[1]} }
run 60 ns
run 60 ns
run 9120 ns
add_force {/multiplier_tb/test} -radix bin {0 1ps}
run 60 ns
```