EE2016: Microprocessors Lab
Experiment # 1

Amizhthni PRK, EE21B015
Collaborator:
Nachiket Dighe, EE21B093

October 1, 2022

Contents

List of Figures

1 Series-Parallel Multiplier

Implementation of a 4-bit Serial-Parallel Multiplier in FPGA (Xilinx's Spartan 3E Board)

1.1 Approach

- Multiply the multiplicand by Least Significant Digit of multiplier, shift right, then repeat for the next digit to writeout the product beneath the first product, keep doing till Most Significant Digit and add them now. The only difference is that for each multiplication above, use repeated addition and use the same accumulator to hold the product.
- Of the different kinds of circuits, we use Serial - parallel multipliers for hardware simplicity and moderate speed.
- An extra bit at the left end of the product register temporarily stores any carry generated when the multiplicand is added to the accumulator
- The current multiplier bit is denoted by M while C denotes carry.

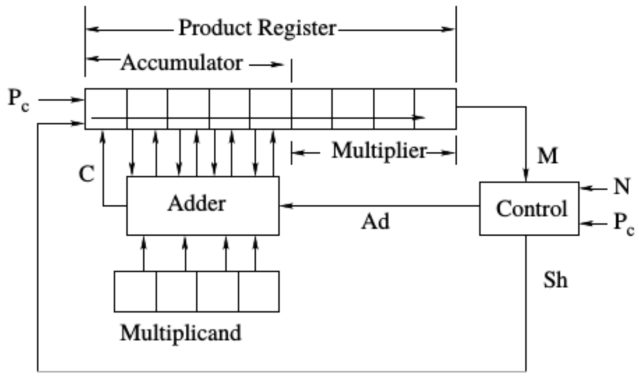

Figure 1: Serial Parallel multiplier

1.2 Code

The code used for Series-Parallel Multiplier and its testbench is given below in listing 1 and 2.

```
module noradd(
  input [3:0]A,
  input [3:0]B,
  output reg [7:0]product
    );
always@(A or B)
begin

   product = 8'b00000000;
```

```
    if(B[0] == 1'b1)
      product = product + (A<<0);
    if(B[1] == 1'b1)
      product = product + (A<<1);
    if(B[2] == 1'b1)
      product = product + (A<<2);
    if(B[3] == 1'b1)
      product = product + (A<<3);

end
endmodule
```

Listing 1: Code used for Series Parallel Multiplier

```
module testi;

 // Inputs
 reg [3:0] A;
 reg [3:0] B;

 // Outputs
 wire [7:0] product;

 // Instantiate the Unit Under Test (UUT)
 noradd uut (
  .A(A),
  .B(B),
  .product(product)
 );

 initial begin
  // Initialize Inputs
  A = 0;
  B = 0;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  A = 1;
  B = 1;
 end

endmodule
```

Listing 2: Testbench used for Series Parallel Multiplier

1.3 Code Inputs and Outputs

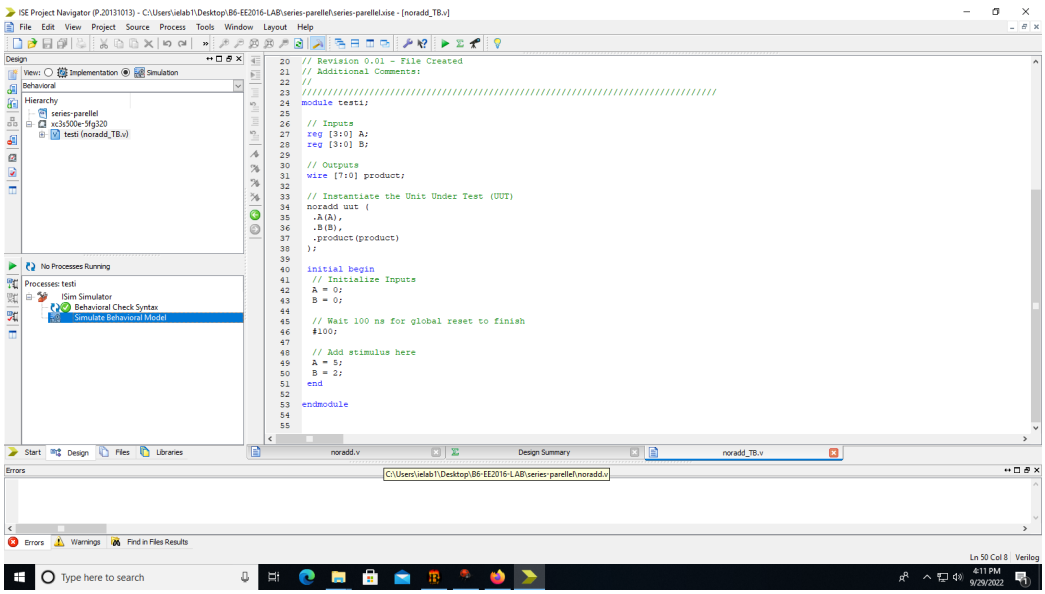

Figure 2: Sample input (A=5 , B=2)

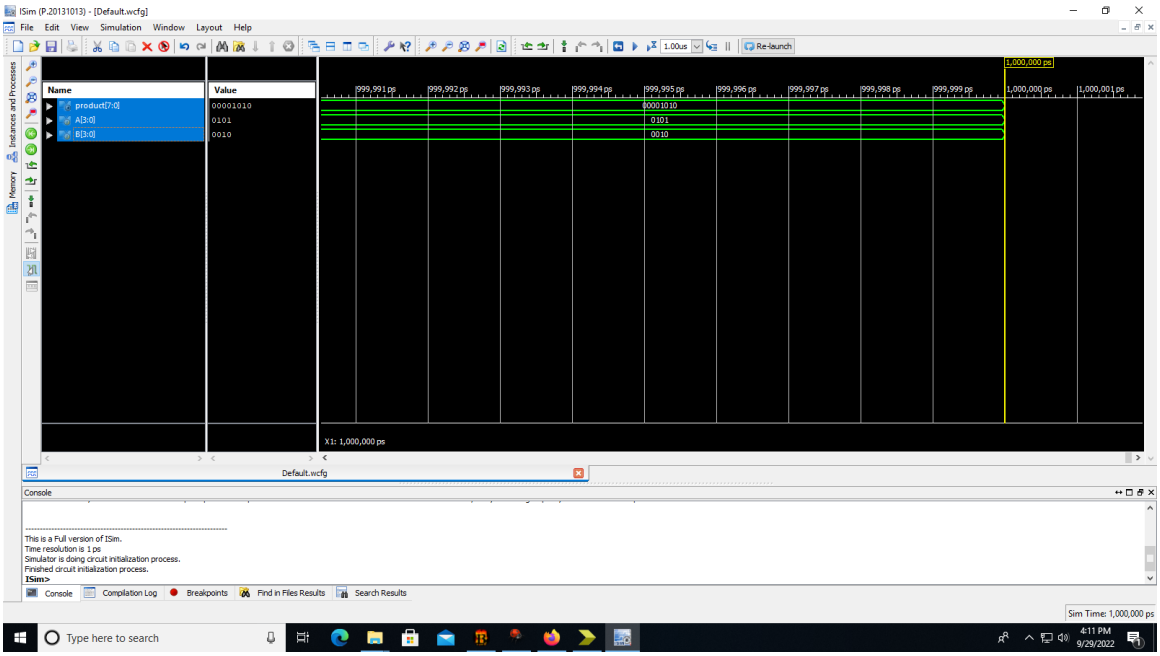

Figure 3: Output (product=10)

1.4 Verilog Code explanation

1. **Step 1:** Load the initial values for the registers. A = Multiplicand , B = Multiplier and an output register(product) = 0.
2. **Step 2:** Check for the Least significant bit of the multiplier. If its 1 then the output register should add the Multiplicand.
3. **Step 3:** Repeat this process for 2nd least significant bit of Multiplier but this time shift the Multiplicand by left once.
4. **Step 4:** Repeating for 3rd and 4th bit of Multiplier our result (A*B) is stored in register product.
5. **Step 5:** Stop.

2 Booth's Multiplier

Implementation of a 4-bit Booth's Multiplier in FPGA (Xilinx's Spartan 3E Board)

2.1 Approach

- Booth's multiplication algorithm is an algorithm that multiplies two signed binary numbers in fewer additions and subtractions than the normal multiplication algorithm.

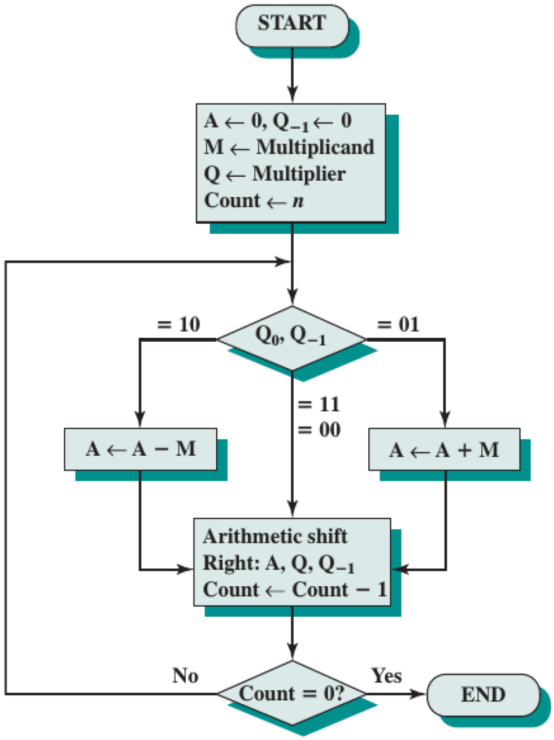

Figure 4: Booth's multiplier

2.2 Code

The code used for Booth's Algorithm and its testbench is given below in listing 3 and 4.

```
module BOOTHS_MULTIPLIER(
    output [7:0] prod,
    output busy,
    input [3:0] mc,
    input [3:0] mp,
    input clk,
    input start
    );
    reg [3:0] A, Q, M; // all registers are of 4 bits
    reg Q_1;
    reg [2:0] count;
```

```
    wire [3:0] sum, difference;

    always @(posedge clk)
    begin
        if (start)
        begin
            A <= 1'b0;
            M <= mc;
            Q <= mp;
            Q_1 <= 1'b0; // bit written to the left of lsb of number to be multiplied
            count <= 3'b0;
        end
        else
        begin
            case ({Q[0], Q_1})
                2'b0_1 : {A, Q, Q_1} <= {sum[3], sum, Q};
                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
    alu adder(sum, A, M, 0); // adder
    alu subtracter(difference, A,~M, 1); //subtractor using 2's compliment
    assign prod = {A, Q}; // make it fill up the arguments
    assign busy = (count < 5);
endmodule

// The following is an alu.It is an adder, but capable of subtraction:
// Recall that 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(out, a, b, cin);
    output [3:0] out;
    input [3:0] a;
    input [3:0] b;
    input cin;
    assign out = a + b + cin;
endmodule
```

Listing 3: Code used for Booth's Multiplier

```
module BOOTHS_MULTIPLIER_TB;

    // Inputs
    reg [3:0] mc;
    reg [3:0] mp;
    reg clk;
    reg start;
```

```
8
9       // Outputs
10      wire [7:0] prod;
11      wire busy;
12
13      // Instantiate the Unit Under Test (UUT)
14      BOOTHS_MULTIPLIER uut (
15          .prod(prod),
16          .busy(busy),
17          .mc(mc),
18          .mp(mp),
19          .clk(clk),
20          .start(start)
21      );
22
23      initial begin
24          // Initialize Inputs
25          mc = 4'b0011;
26          mp = 4'b0010;
27          clk = 1;
28          start = 1;
29          #10 clk = ~clk;
30          #10 clk = ~clk;
31          start = 0;
32          #10 clk = ~clk;
33          #10 clk = ~clk;
34          #10 clk = ~clk;
35          #10 clk = ~clk;
36          #10 clk = ~clk;
37          #10 clk = ~clk;
38          #10 clk = ~clk;
39          #10 clk = ~clk;
40
41
42
43          $finish;
44      end
45
46      initial begin
47        $dumpfile("BOOTHS.vcd");
48        $dumpvars(0,BOOTHS_MULTIPLIER_TB);
49      end
50
51  endmodule
```

Listing 4: Testbench used for Booth's Multiplier

2.3 Outputs and code inputs

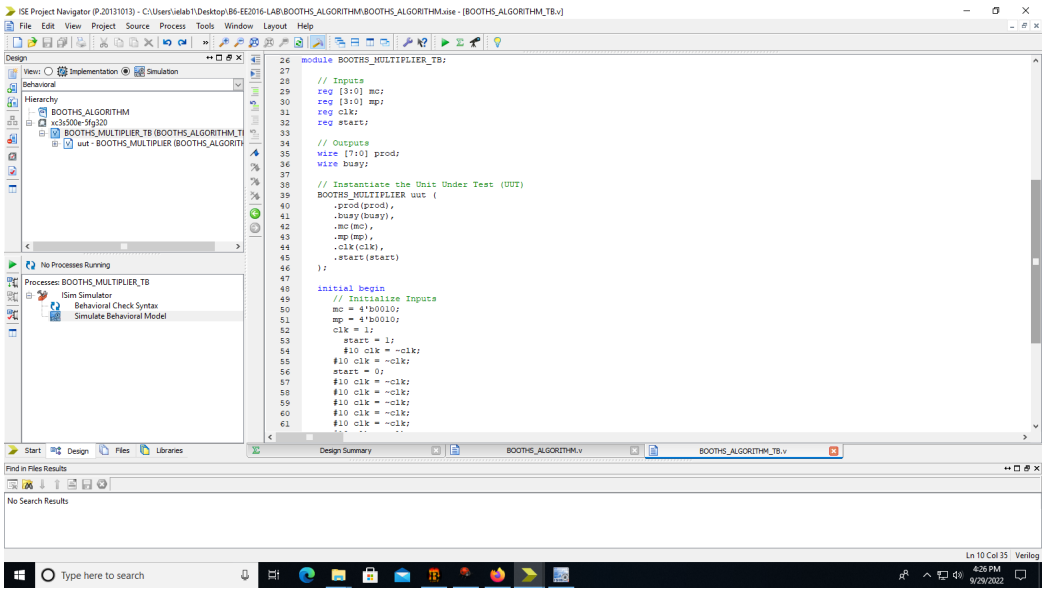

Figure 5: Sample input (mc=2 , mp=2)

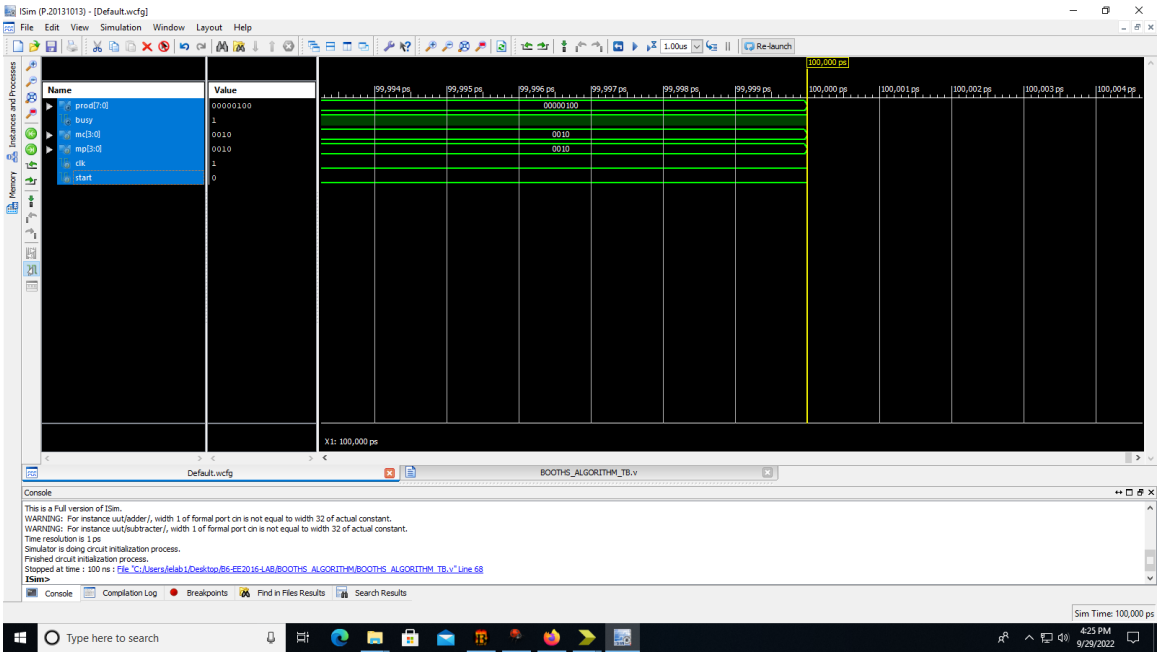

Figure 6: Output (prod = 4)

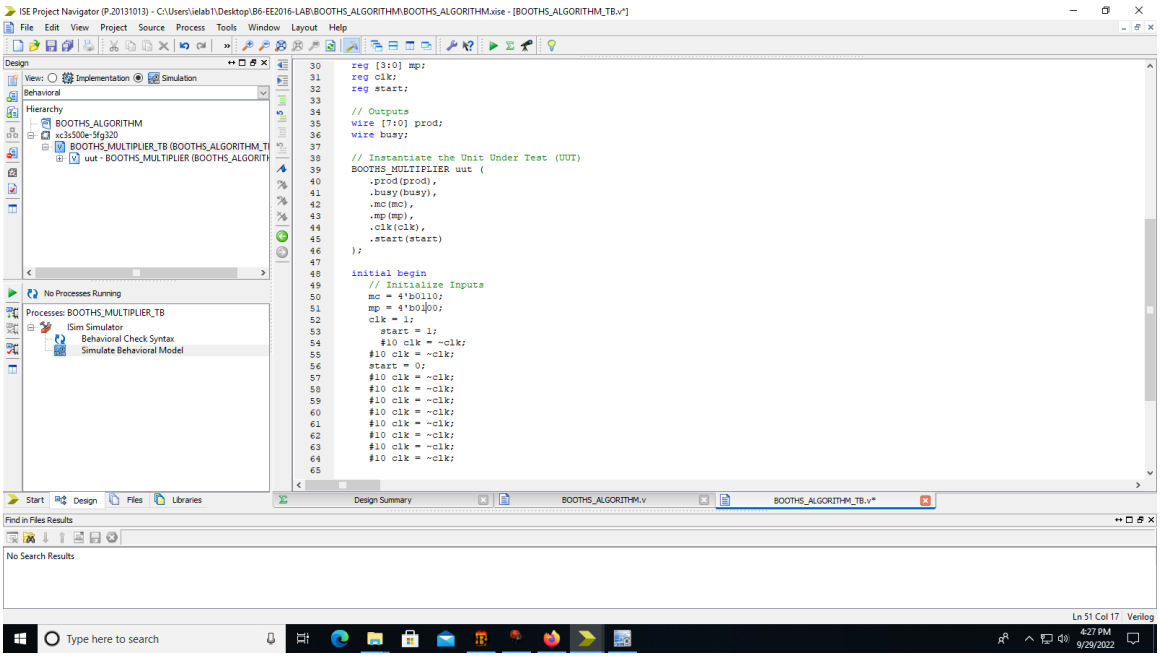

Figure 7: Sample input (mc=6 , mp=4)

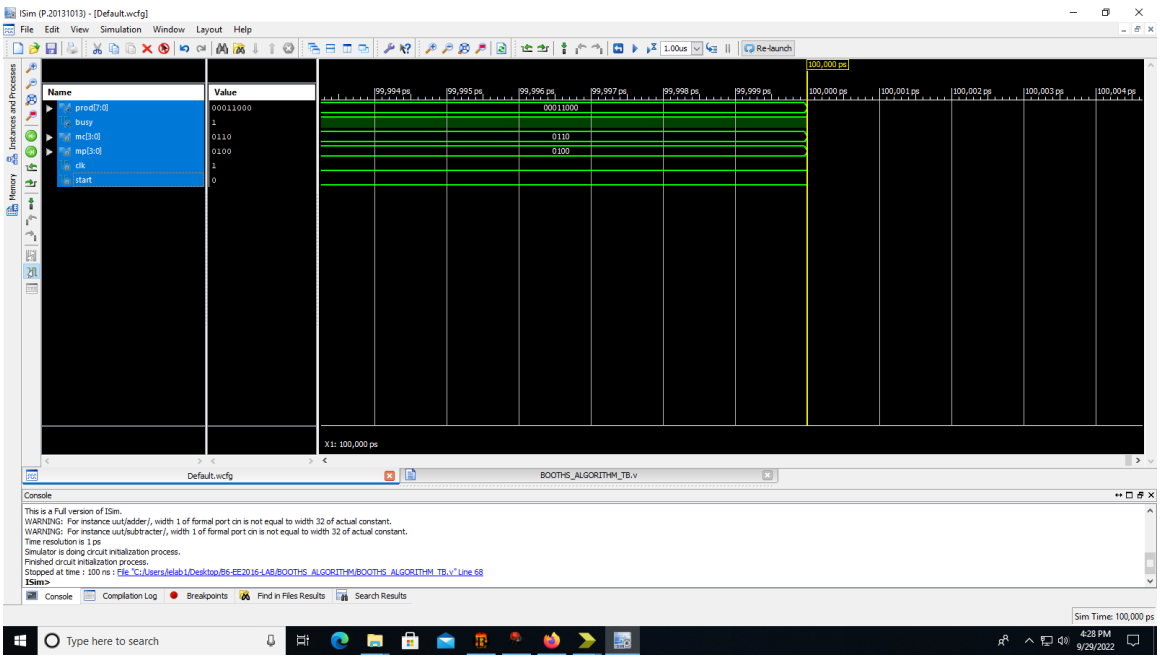

Figure 8: Output (prod=24)

2.4 Verilog Code explanation

1. **Step 1:** Load the initial values for the registers. A = 0 (Accumulator), Q1 = 0, M = Multiplicand, Q = Multiplier and n is the count value which equals the number of bits of multiplier.
2. **Step 2:** Check the value of Q0,Q1. If 00 or 11, goto step 5. If 01, goto step 3. If 10, goto step 4.
3. **Step 3:** Perform A = A + M. Goto step 5.
4. **Step 4:** Perform A = A - M.
5. **Step 5:** Perform Arithmetic Shift Right of A, Q, Q1 and decrement count.
6. **Step 6:** Check if counter value n is zero. If yes, goto next step. Else, goto step 2.
7. **Step 7:** Stop.

In **IDLE** state, we wait for the start pulse. Upon its arrival, move to **START** state. In **START** state, we follow the same algorithm and perform the computations using verilog logic as long as counter < 3. (Incrementing counter is used). Upon counter completion, we send out valid pulse and goto **IDLE** state.

3 Comparison of both the outputs

1. **Serial-Parallel Multiplier:** One operand is fed to the circuit in parallel while the other is in serial. N partial products are formed for each cycle. On successive cycles, each cycle does the addition of one column of the multiplication table of M*N Partial Products. The final results are then stored in the output register after completing N+M cycles.

2. **Booth's Algorithm:** Our Multipliers should consume less power and less space for efficient performance . Booth algorithm provides the procedure of multiplication of binary integers with 2's complement representation, hence uses of additions and subtractions would be reduced.