**Lab 1 Multi-Barrel Shifter**

**Name:** Beverly Abadines

**I Goals**

1) Implement Verilog software onto FPGA board using ucf.

2) Practice bit shift and rotations

4) Distinguish when to use procedural and continuous assignment

5) Design RTL Combinational Circuit

6) Understand the conventions used on various datatypes

**II Design Process**

Both designs take in data\_in, lr, and amount in order to output the rotated value, data\_out. The number of bits to rotate is set by amount, while lr dictates whether to perform left rotation (lr ==1) or right rotation (lr ==0).

**Bidirectional (pre and post reverse)**

This design uses only a single left\_rotator, two mux\_2\_to\_1 and and two reverse. If lr == 1, then the datapath is as follows: data\_in is the output of the first mux, which is the input of left\_rotator and the input and output of the second mux. If lr == 0, then the reverse of data\_in is the output of the first mux, which is the input of the left\_rotator. It goes through a reversal again and is the input and output of the second mux. It is important to note that regardless whether the left rotate or right rotate is desired the reverse procedures still executes.

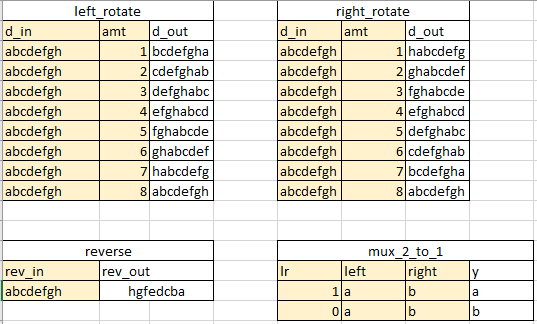


**Barrel\_rotator (left and right)**

This uses both the right rotator and left rotator as well as a mux\_2\_to\_1. If lr ==1 then the output is the left rotated value. If lr ==0, then the output is the right rotated value. Again regardless of the value of lr, both rotations will occur, but there will be a single output due to the mux.



**Truth Table**



**III Program**

// Connects the Barrel\_Rotator code to the physical layout onto the FPGA.

//////////////////////////////////////////////////////////////////////////////////

module **FPGA\_Barrel\_Rotator**(

output wire [7:0] led,

input wire [3:0] btn, // 3:1 amount, 0 lr

input wire [7:0] sw

);

Barrel\_Rotator BR\_Test(

.data\_out(led),

.lr(btn[0]),

.amount(btn[3:1]),

.data\_in(sw)

);

endmodule // FPGA\_Barrel\_Rotator

//// Connects the Bidirection\_Rotation code to the physical layout onto the FPGA.

//////////////////////////////////////////////////////////////////////////////////

module **FPGA\_Bidirection\_Rotation**(

output wire [7:0] led,

input wire [3:0] btn, // 3:1 amount, 0 lr

input wire [7:0] sw

);

Bidirection\_Rotation BR\_Test(

.data\_out(led),

.lr(btn[0]),

.amount(btn[3:1]),

.data\_in(sw)

);

endmodule//FPGA\_Bidirection\_Rotation

// This is the complete design for the Barrel\_rotator (left and right),

// which is suitable for simulation.

//////////////////////////////////////////////////////////////////////////////////

module **Barrel\_Rotator**(

output [7:0] data\_out,

input wire lr,

input wire [2:0] amount,

input [7:0] data\_in

);

// internal signals

wire [7:0] left\_out, right\_out;

left\_rotator left\_rotator1(

.d\_out(left\_out), //output

.d\_in(data\_in), //inputs

.bit\_amount(amount)

);

right\_rotator right\_rotator1(

.d\_out(right\_out), //output

.d\_in(data\_in), //inputs

.bit\_amount(amount)

);

mux\_2\_to\_1 mux\_2\_to\_1\_1(

.y(data\_out), //output

.left(left\_out),//inputs

.right(right\_out),

.lr(lr)

);

endmodule // Barrel\_Rotator

//// This is the complete design for the Bidirection\_Rotation (pre and post reverse),

// which is suitable for simulation.

//////////////////////////////////////////////////////////////////////////////////

module **Bidirection\_Rotation**(

output wire [7:0] data\_out,

input wire lr,

input wire [2:0] amount,

input wire [7:0] data\_in

);

// internal signals

wire [7:0] pre\_rev\_out, post\_rev\_out;

wire [7:0] pre\_mux\_out;

wire [7:0] right\_out;

reverse pre\_rev(

.rev\_out(pre\_rev\_out), //output

.rev\_in(data\_in) //input

);

mux\_2\_to\_1 pre\_mux(

.y(pre\_mux\_out), //output

.right(data\_in), //inputs

.left(pre\_rev\_out),

.lr(lr) // lr = 1 -> prev\_rev\_out, lr = 0 -> data\_in

);

right\_rotator right\_rotator1(

.d\_out(right\_out), //output

.d\_in(pre\_mux\_out), //inputs

.bit\_amount(amount)

);

reverse post\_rev(

.rev\_out(post\_rev\_out), //output

.rev\_in(right\_out) //input

);

mux\_2\_to\_1 post\_mux(

.y(data\_out), //output

.right(right\_out),//input

.left(post\_rev\_out), // lr = 1 -> post\_rev\_out, lr = 0 -> right\_out

.lr(lr)

);

endmodule // Bidirection\_Rotation

// Left rotates the value d\_in based on the bit\_amount. The output is d\_out.

//////////////////////////////////////////////////////////////////////////////////

module **left\_rotator**(

output reg [7:0] d\_out, // 8'b

input wire [7:0] d\_in,

input wire [2:0] bit\_amount // 3'b, enough to specify #'b to rotate

);

//In Verilog you can't use a variable as the end of range.

// https://verificationacademy.com/forums/systemverilog/range-must-be-bounded-constant-expressions

//assign d\_out = {d\_in[bit\_amount +: 0], d\_in[7+: 8]};

always @(\*) begin

case(bit\_amount)

3'd1: d\_out = {d\_in[6:0], d\_in[7]};

3'd2: d\_out = {d\_in[5:0], d\_in[7:6]};

3'd3: d\_out = {d\_in[4:0], d\_in[7:5]};

3'd4: d\_out = {d\_in[3:0], d\_in[7:4]};

3'd5: d\_out = {d\_in[2:0], d\_in[7:3]};

3'd6: d\_out = {d\_in[1:0], d\_in[7:2]};

3'd7: d\_out = {d\_in[0], d\_in[7:1]};

default: d\_out = d\_in; // case 0

endcase

end

endmodule//left\_rotator

/// Right rotates the value d\_in based on the bit\_amount. The output is d\_out.

//////////////////////////////////////////////////////////////////////////////////

module **right\_rotator**(

output reg [7:0] d\_out, // 8'b

input wire [7:0] d\_in,

input wire [2:0] bit\_amount // 3'b, enough to specify #'b to rotate

);

//In Verilog you can't use a variable as the end of range.

// https://verificationacademy.com/forums/systemverilog/range-must-be-bounded-constant-expressions

//assign d\_out = {d\_in[bit\_amount +: 0], d\_in[7+: 8]};

always @(\*) begin

case(bit\_amount)

3'd1: d\_out = {d\_in[0], d\_in[7:1]};

3'd2: d\_out = {d\_in[1:0], d\_in[7:2]};

3'd3: d\_out = {d\_in[2:0], d\_in[7:3]};

3'd4: d\_out = {d\_in[3:0], d\_in[7:4]};

3'd5: d\_out = {d\_in[4:0], d\_in[7:5]};

3'd6: d\_out = {d\_in[5:0], d\_in[7:6]};

3'd7: d\_out = {d\_in[6:0], d\_in[7]};

default: d\_out = d\_in; // case 0

endcase

end

endmodule//right\_rotator

// Y, outputs the value based on lr. y = left if lr ==1, and right otherwise.

//////////////////////////////////////////////////////////////////////////////////

module mux\_2\_to\_1(

output wire [7:0] y, // output of mux

input wire [7:0] left, right, // input a = 1'b1, b = 1'b0

input wire lr // select, left or right

);

assign y = lr ? left: right; // the type after assign must be WIRE

// if lr = 1, then y = left

// if lr = 0, then y = right

endmodule//mux\_2\_to\_1

// Outputs the reverse order of rev\_in as rev\_out.

//////////////////////////////////////////////////////////////////////////////////

module **reverse**(

output wire [7:0] rev\_out,

input wire [7:0] rev\_in

);

//Generate loop index i must be defined as a genvar

genvar i;

// generate for loop is generating an instance for each iteration

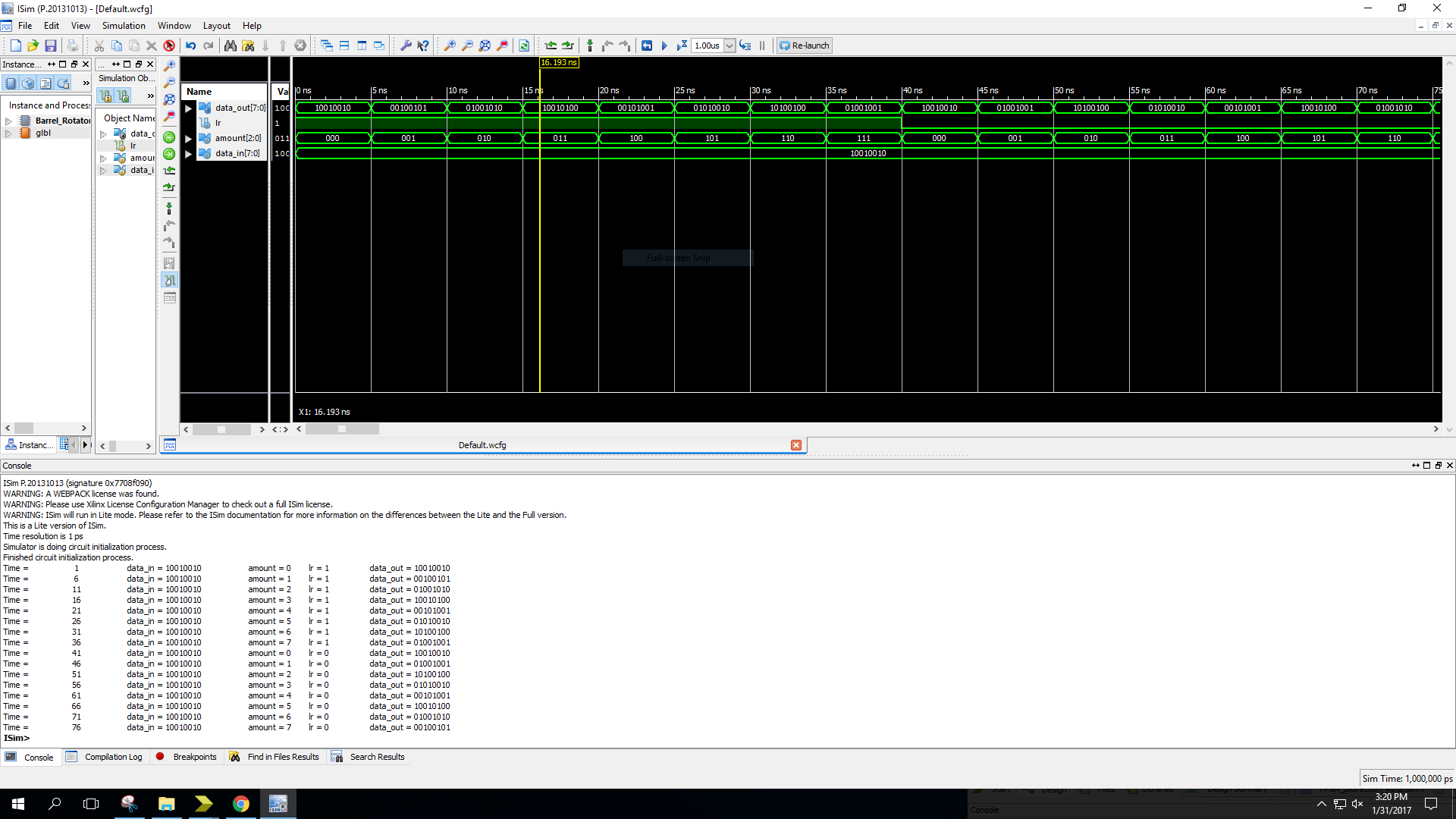
generate for(i = 0; i < 8; i = i + 1) begin

assign rev\_out[i] = rev\_in[7-i];

end endgenerate

endmodule // reverse

**IV Results**   
**Bidirection and Barrel\_Rotatator testbench**

****

module Bidirection\_Rotation\_testbench; // Similar to Barrel\_Rotator\_testbench

// Inputs

reg lr;

reg [2:0] amount;

reg [7:0] data\_in;

// Outputs

wire [7:0] data\_out;

// Instantiate the Unit Under Test (UUT)

Bidirection\_Rotation uut (

.data\_out(data\_out),

.lr(lr),

.amount(amount),

.data\_in(data\_in)

);

initial begin

// Initialize Inputs and check for left rotate first

data\_in = 8'b10010010;

lr = 1;

amount = 3'b000;

#5;

amount = 3'b001;

#5

amount = 3'b010;

#5

amount = 3'b011;

#5

amount = 3'b100;

#5;

amount = 3'b101;

#5

amount = 3'b110;

#5

amount = 3'b111;

#5

lr = 0; // Now check for right rotate

amount = 3'b000;

#5;

amount = 3'b001;

#5

amount = 3'b010;

#5

amount = 3'b011;

#5

amount = 3'b100;

#5;

amount = 3'b101;

#5

amount = 3'b110;

#5

amount = 3'b111;

end

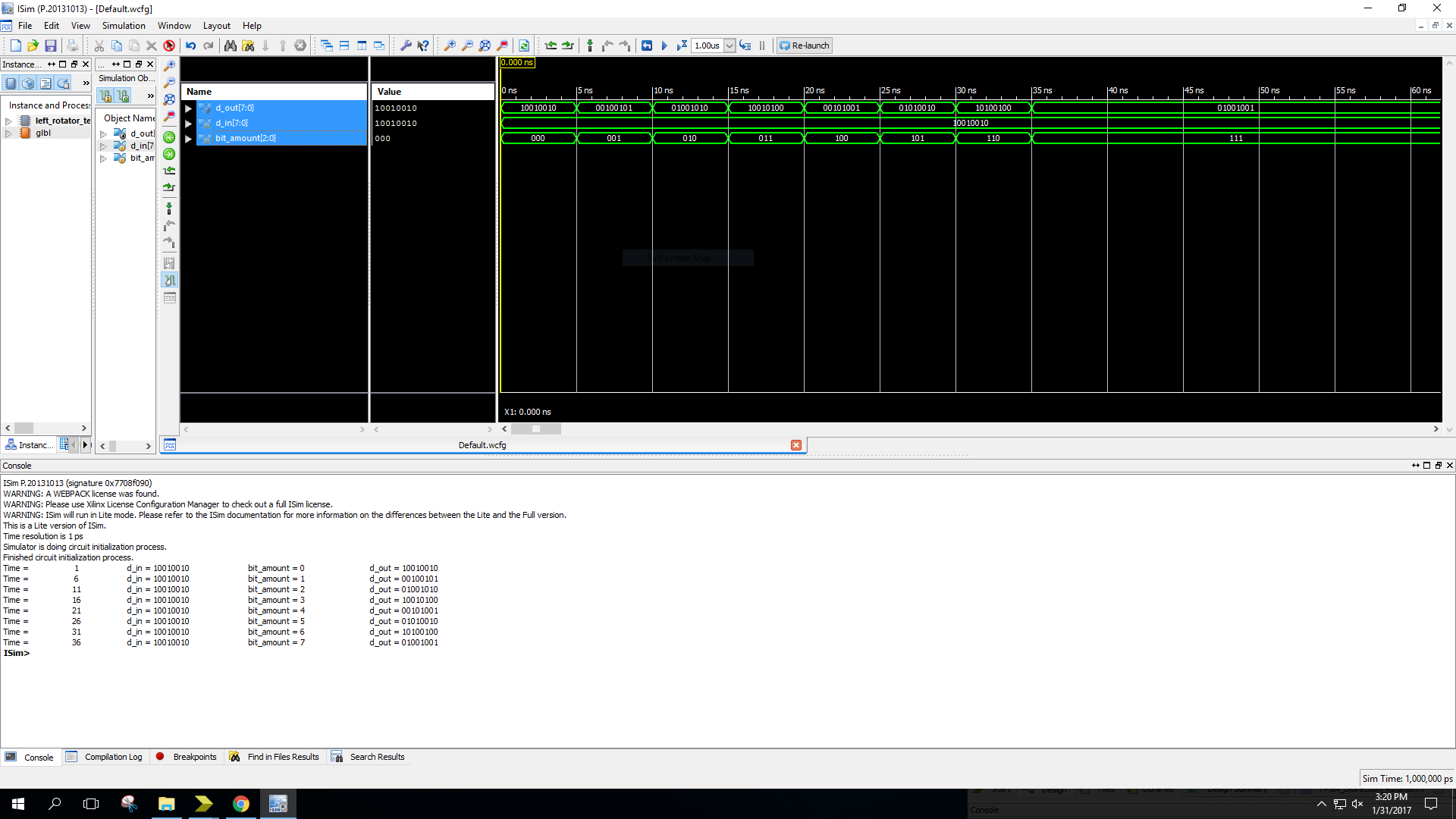
always @(data\_out)

#1 $display ("Time = %d \t data\_in = %b \t amount = %d \t lr = %b \t data\_out = %b",

$time, data\_in, amount, lr, data\_out);

endmodule

**Left\_rotator testbench**

****

module left\_rotator\_testbench; // Similar to right\_rotator\_testbench

// Inputs

reg [7:0] d\_in;

reg [2:0] bit\_amount;

// Outputs

wire [7:0] d\_out;

// Instantiate the Unit Under Test (UUT)

left\_rotator uut (

.d\_out(d\_out),

.d\_in(d\_in),

.bit\_amount(bit\_amount)

);

initial begin

// Initialize Inputs

d\_in = 8'b10010010;

bit\_amount = 3'b000;

#5;

bit\_amount = 3'b001;

#5

bit\_amount = 3'b010;

#5

bit\_amount = 3'b011;

#5

bit\_amount = 3'b100;

#5;

bit\_amount = 3'b101;

#5

bit\_amount = 3'b110;

#5

bit\_amount = 3'b111;

end

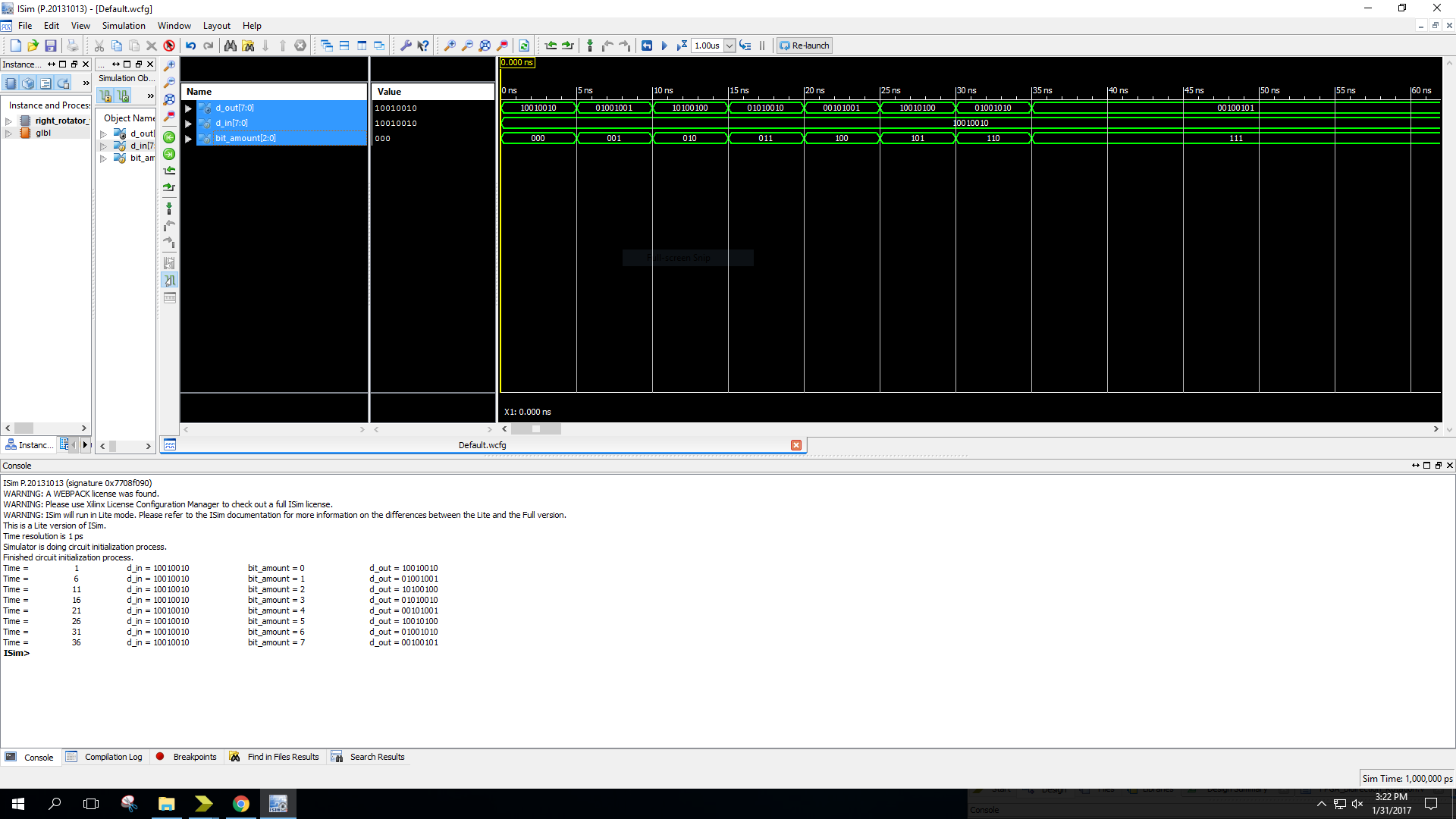
always @(d\_out)

#1 $display ("Time = %d \t d\_in = %b \t bit\_amount = %d \t d\_out = %b",

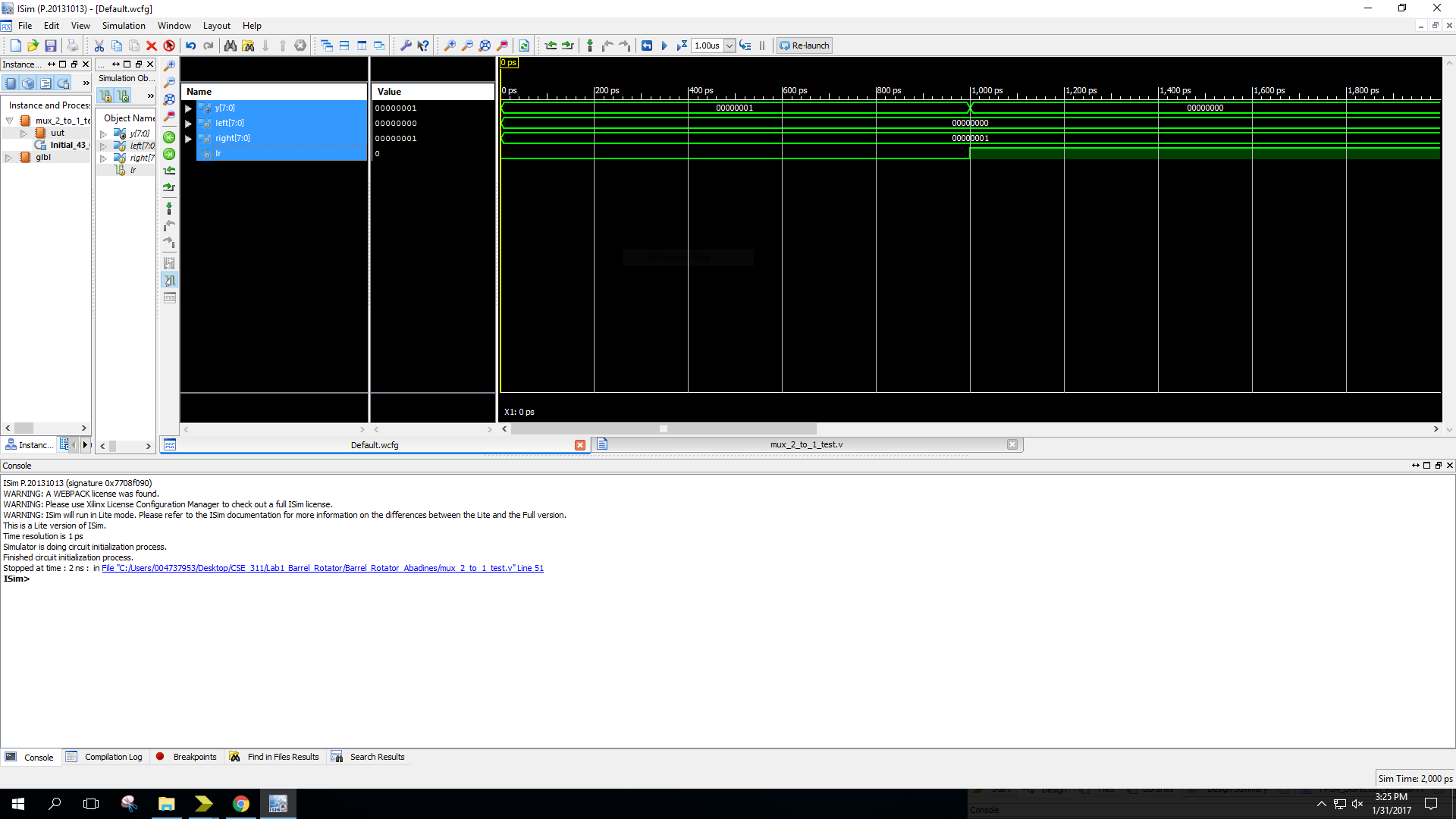
$time, d\_in, bit\_amount, d\_out);

endmodule

**Right\_rotator testbench**

****

**Mux\_2\_to\_1testbench**

****

module mux\_2\_to\_1\_test;

// Inputs

reg [7:0] left;

reg [7:0] right;

reg lr;

// Outputs

wire [7:0] y;

// Instantiate the Unit Under Test (UUT)

mux\_2\_to\_1 uut (

.y(y),

.left(left),

.right(right),

.lr(lr)

);

initial begin

// Initialize Inputs

left = 0;

right = 1;

lr = 0; //output right

#1;

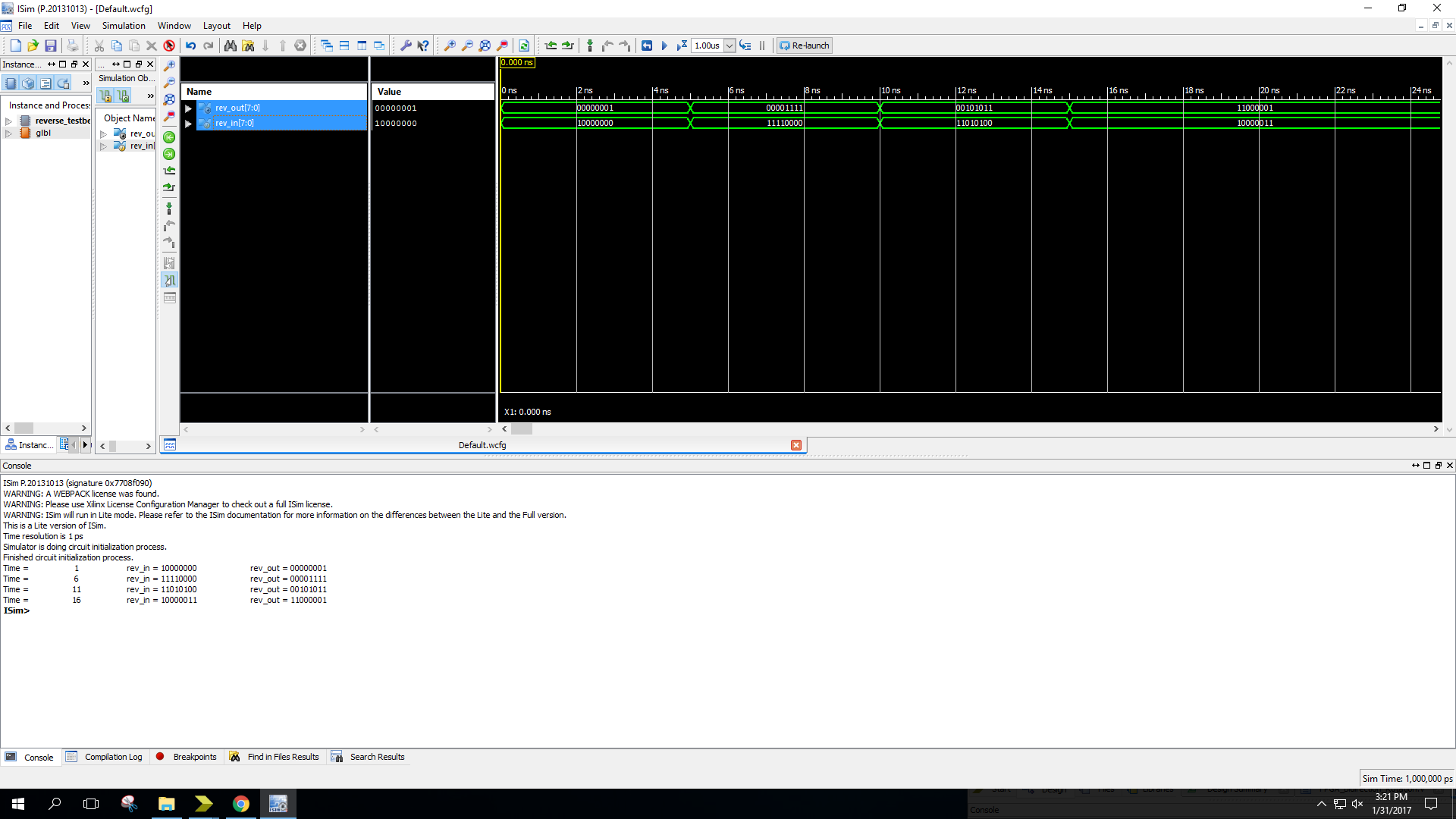
lr =1; // output left

#1;

$stop;

end

endmodule

**Reverse testbench**

module reverse\_testbench;

// Inputs

reg [7:0] rev\_in;

// Outputs

wire [7:0] rev\_out;

// Instantiate the Unit Under Test (UUT)

reverse uut (

.rev\_out(rev\_out),

.rev\_in(rev\_in)

);

initial begin

// Initialize Inputs

rev\_in = 8'b10000000;

#5

rev\_in = 8'b11110000;

#5

rev\_in = 8'b11010100;

#5

rev\_in = 8'b10000011;

end

always @(rev\_out)

#1 $display ("Time = %d \t rev\_in = %b \t rev\_out = %b",

$time, rev\_in, rev\_out);

Endmodule

**V Problem**

1. I attempted to develop an algorithm for the left and right rotation instead of hardcoding the values based on the bit\_amount, but Verilog was incapable of having a variable as the end range of the array. To solve the problem I hardcoded the values into a case statement.
2. Again, I did not want to hardcode values for the reverse module as well. In planning, I wished to develop an algorithm that can handle dynamic changes in parameters. I used a for loop to try to index values dynamically. Overall, I was able to run and compile the program using *genvar* index and a *generate* for loop.
3. I did not know how to connect the I/O signals to the I/O pins on the board. I used a default.ucf and commented out unnecessary ports.

**VI What you learned**

As robust as Verilog is as HDL, and as practical as it is for its ability to be used for real-world applications, it is not suitable for every algorithm. My first problem displays how an algorithm can work in theory, but fail in application.

Secondly, I learned that to distinguish little nuances based on datatypes and conditionals, and recognize the differences between combination logic circuits versus sequential circuits. (for loop, generate for loop, net group versus variable group, continuous versus procedural).

Lastly, I learned how to implement Verilog software into a tangible application, such as the FPGA.