Lab 1 Multi-Barrel Shifter

Name: Beverly Abadines

I Goals

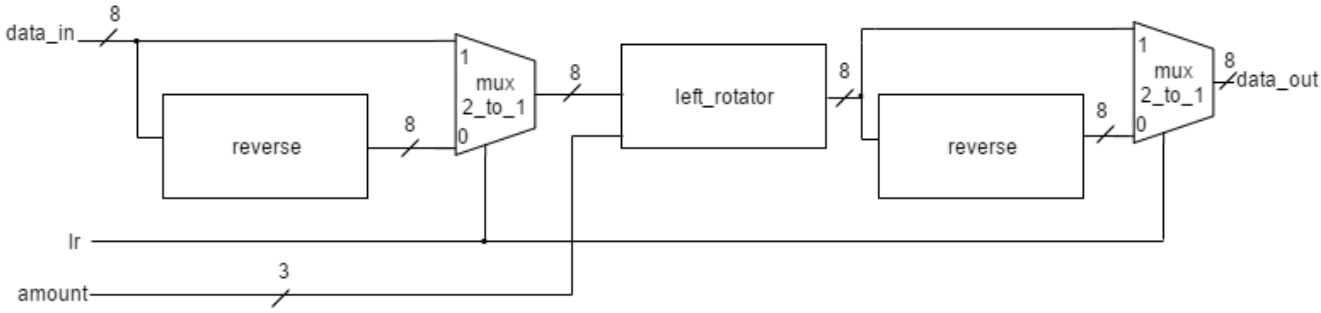
1) Implement Verilog software onto FPGA board using ucf.
2) Practice bit shift and rotations
4) Distinguish when to use procedural and continuous assignment
5) Design RTL Combinational Circuit
6) Understand the conventions used on various datatypes

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).

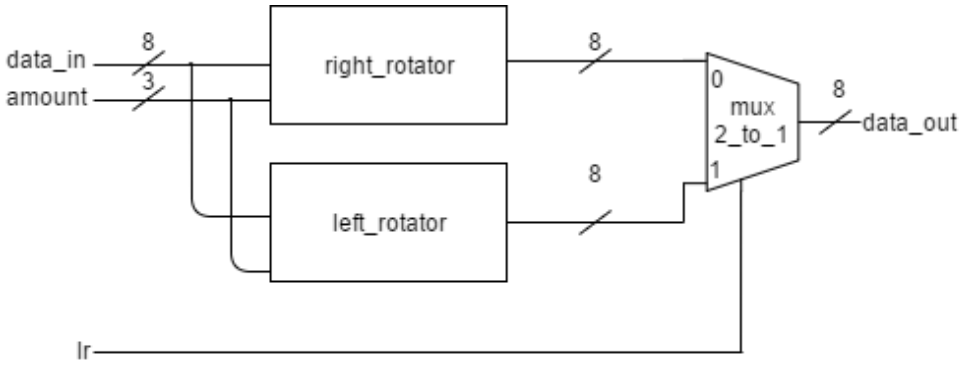
Bidirectional (pre and post reverse)

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

left_rotate		
d_in	amt	d_out
abcdefgh	1	bcdefgha
abcdefgh	2	cdefghab
abcdefgh	3	defghabc
abcdefgh	4	efghabcd
abcdefgh	5	fghabcde
abcdefgh	6	ghabcdef
abcdefgh	7	habcdefg
abcdefgh	8	abcdefgh

right_rotate		
d_in	amt	d_out
abcdefgh	1	habcdefg
abcdefgh	2	ghabcdef
abcdefgh	3	fghabcde
abcdefgh	4	efghabcd
abcdefgh	5	defghabc
abcdefgh	6	cdefghab
abcdefgh	7	bcdefgha
abcdefgh	8	abcdefgh

reverse	
rev_in	rev_out
abcdefgh	hgfedcba

mux_2_to_1			
lr	left	right	y
1	a	b	a
0	a	b	b

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)
                );
```

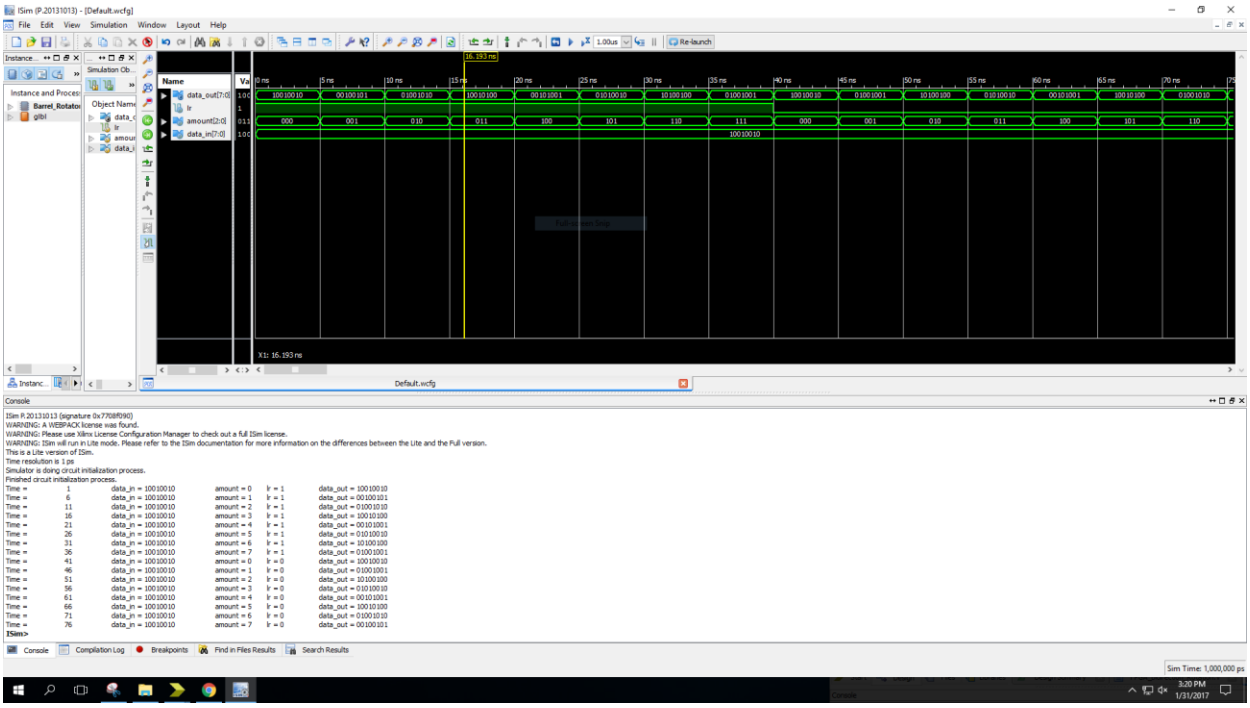
```
        reverse post_rev(
                .rev_out(post_rev_out),         //output
                .rev_in(right_out)                      //input
                );

        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



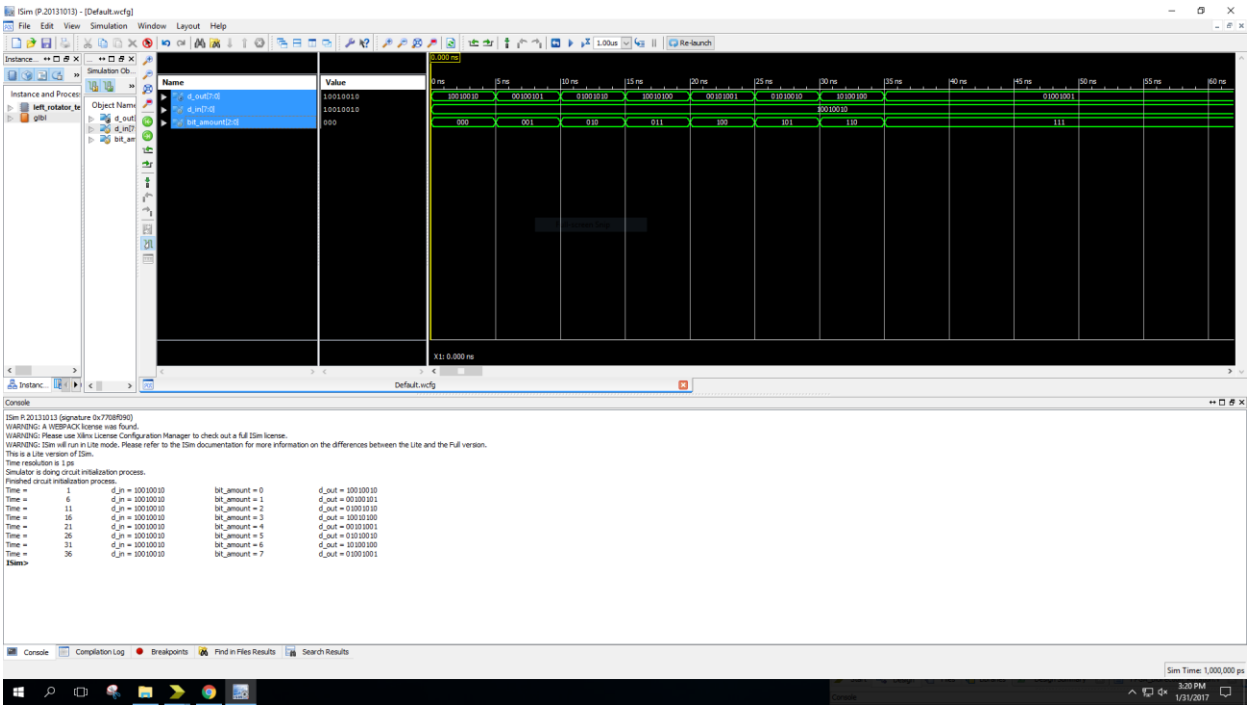
```
module Bidirection_Rotation_testbench; // Similar to Barrel_Rotator_testbench

        // 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

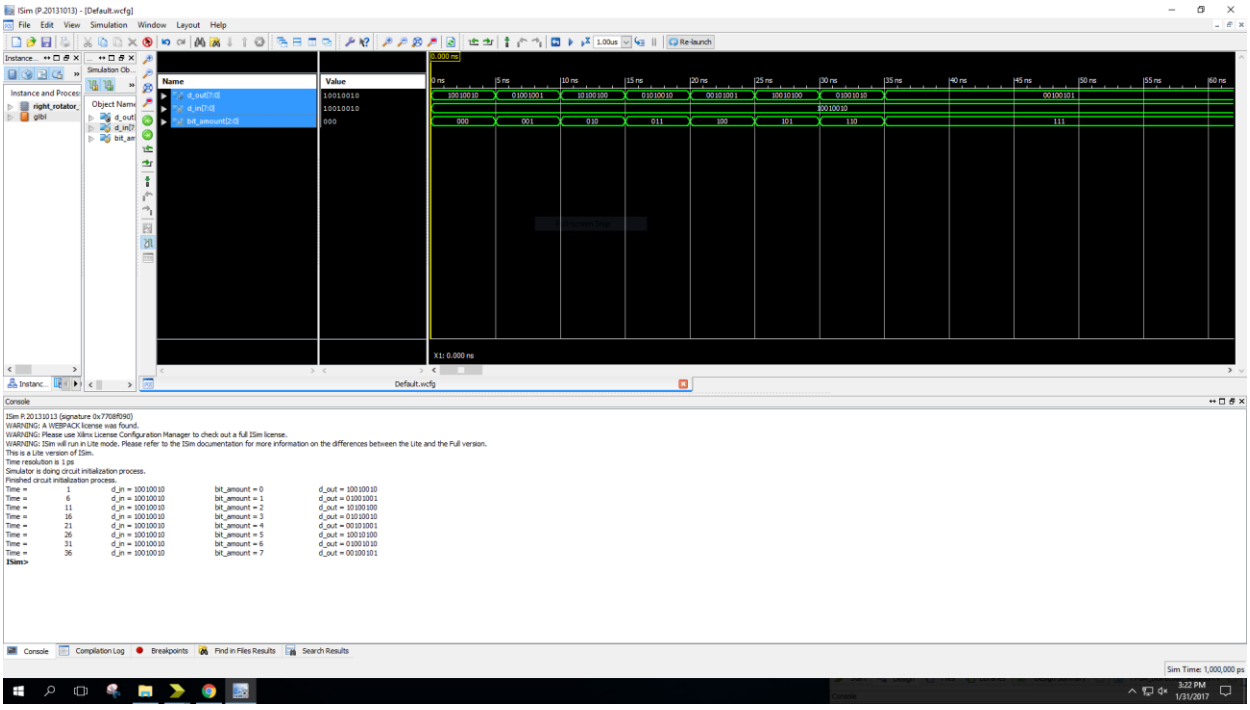
Left_rotator testbench



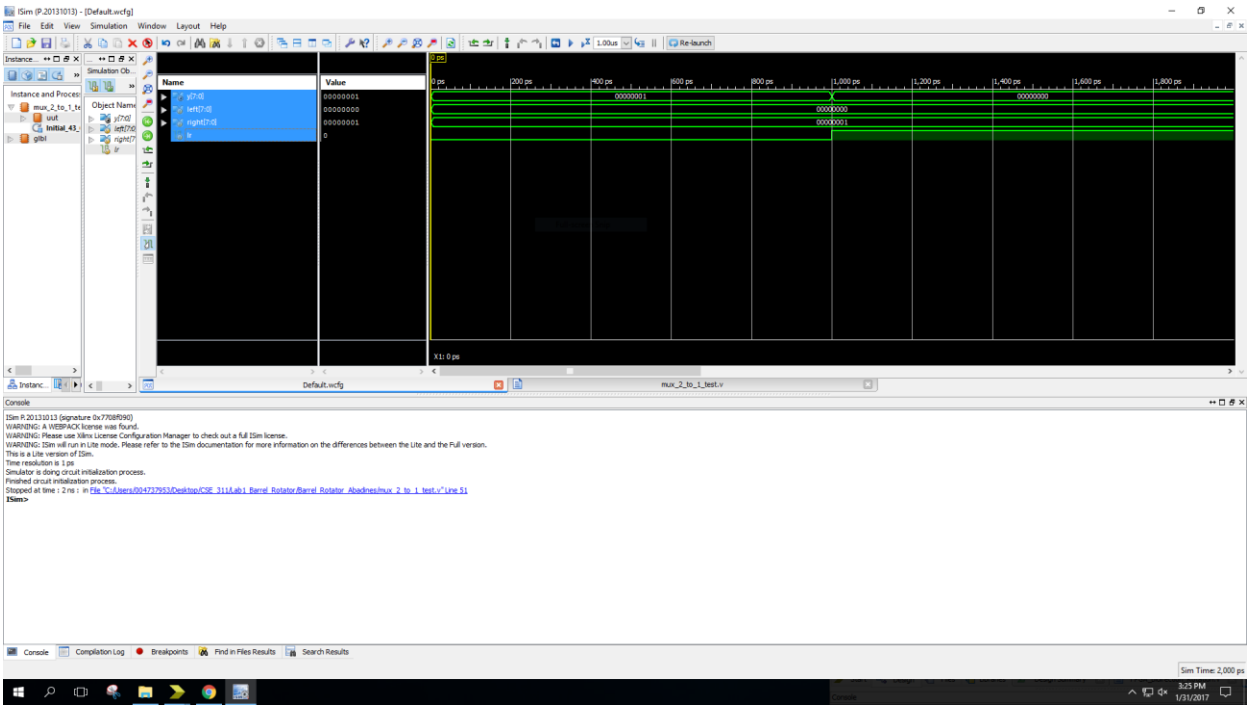
```
module left_rotator_testbench; // Similar to right_rotator_testbench

	// 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



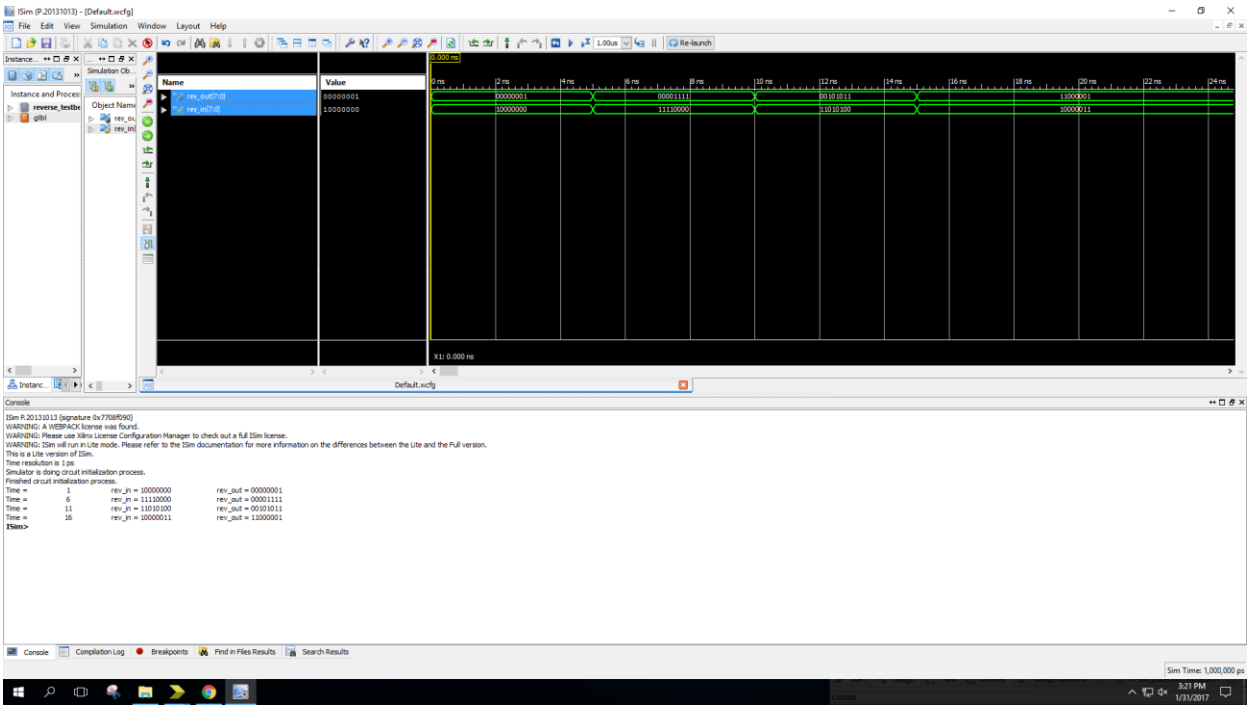
Mux_2_to_1testbench



```
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.