=============================================================

**Lab 3 Report**

**Combinational Building Blocks**

**Date \_\_\_\_\_\_3/4/17\_\_\_\_\_\_\_\_\_**

**by**

**Name \_\_\_\_Anahit Sarao\_\_\_\_\_\_ SID \_\_008435583\_\_\_\_\_\_**

**Name \_\_\_\_Maxwell Cheshier\_\_\_\_\_\_ SID \_\_009193717\_\_\_**

**Lab Record**

| **Tasks** | **Designed by (print name)** | **Verified by (print name)** | **\*Completion Status** |
| --- | --- | --- | --- |
| **1** | Anahit | Ryan | A |
| **2** | Maxwell | Ryan | A |
| **3** | Anahit | Ryan | B |

**\* Enter the following:**

**A – if the task was successfully completed**

**B – if the task was partially completed**

**X – if the task was failed or not performed**

**If you entered B or X, detailed description about the incompletion or failure must be given in the report.**

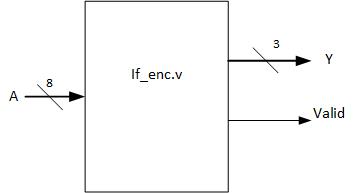
***Introduction***

This lab is divided into three sub components, priority encoder, 4 bit shift rotator, and a 4 bit arithmetic logic unit or ALU. To fully complete and understand the lab three tasks per component are implemented: design, simulation validation, FPGA validation. In order to validate the waveform testbench code was written in relation to the design files. The waveform outputs were compared to the function tables for each component. To validate using hardware the designs were testing by created an xdc file and imported to the board.

***Design Methodology***

Encoder- The 8-3 bit encoder is designed to take in a signal that is 8 bits long and convert to a signal that is 3 bits. For this lab, three different implementations were used if, for, and casez each with their own module. In order to test each of these on the board, a constraint file was created. The constraints utilized 8 dip switches on the board and three on board LEDs were used to verify output. The validation procedure simply relied on using each switch which would turn on the corresponding LEDs representing the 8 bits in three binary bits.

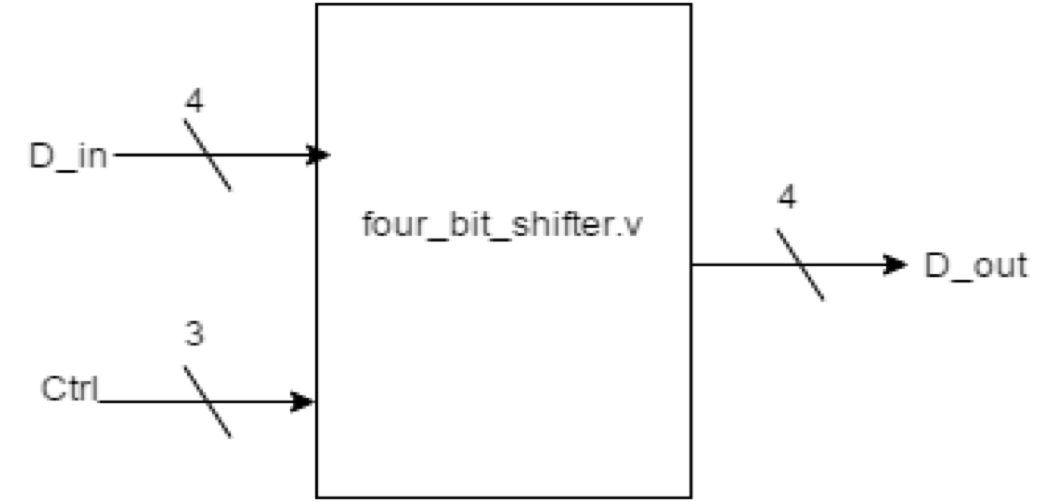
*Figure 1****:*** *Block Diagram of Encoder*

**

*Table 1: Encoder Function Table*

8-bit Input 3-bits Output 1-bit Valid Output   
D[7] D[6] D[5] D[4] D[3] D[2] D[1] D[0] Y [2:0] Valid  
1 x x x x x x x 111 1  
0 1 x x x x x x 110 1  
0 0 1 x x x x x 101 1  
0 0 0 1 x x x x 100 1  
0 0 0 0 1 x x x 011 1  
0 0 0 0 0 1 x x 010 1  
0 0 0 0 0 0 1 x 001 1  
0 0 0 0 0 0 0 1 000 1  
0 0 0 0 0 0 0 0 xxx 0

*Shifter/Rotator:* The four bit shifter and rotator required the ability to shift up to four bits of a four bit LED sequence or rotate up to 3 bits to the right of the same sequence. The module was designed based on the table below. According to the constraint file, the first three switches were designed to be the control switches and the next four switches were designed to be the input switches. The control switches would shift up to four bits, represented by 0-3 in binary, or from 4-6, the bits would be rotated to the right. The LEDs above the switches represented the output signals. In the design, an array was used for both the input and the output in which the output array’s signal was operated on by the “CTRL” signal.

******

*Figure 2:**Diagram for 4-bit Shifter/Rotator*

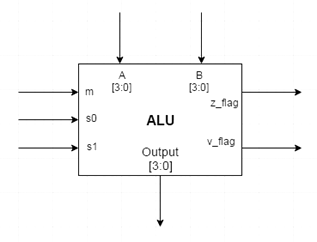
*Table 2: 4-bit Shift Rotator Function Table*

| ***Ctrl [2:0]*** | ***Operation*** | ***Input*** | ***Output*** |
| --- | --- | --- | --- |
| *000* | *Pass* | *a b c d* | *a b c d* |
| *001* | *Shift right 1 bit* | *a b c d* | *0 a b c* |
| *010* | *Shift right 2 bit* | *a b c d* | *0 0 a b* |
| *011* | *Shift right 3 bit* | *a b c d* | *0 0 0 a* |
| *100* | *Shift right 4 bit* | *a b c d* | *0 0 0 0* |
| *101* | *Rotate right 1 bit* | *a b c d* | *d a b c* |
| *110* | *Rotate right 2 bit* | *a b c d* | *c d a b* |
| *111* | *Rotate right 3 bit* | *a b c d* | *b c d a* |

*ALU:* The 4-bit arithmetic logic unit takes in two inputs A and B. The unit can perform any type of logical or arithmetic operation, the operation is controlled by a 3-bit control signal. After the operation the results are sent to the output. Since the control signal is 0-2 or 3-bit it can perform a maximum of eight logical operations or eight arithmetic operations. The input are 4-bit meaning that some arithmetic and logical operations will result in overflow and or zero output. These two condition are handled by the overflow flag (ov) and the zero flag (z). Table three shows the different logical and arithmetic functions the ALU is able to perform.

*Table 3: 4-bit ALU Function Table*

| m | s0 | s1 | Out |
| --- | --- | --- | --- |
| 1 | 0 | 0 | A-1 |
| 1 | 0 | 1 | A+B |
| 1 | 1 | 0 | A-B |
| 1 | 1 | 1 | A+1 |
| 0 | 0 | 0 | ~A |
| 0 | 0 | 1 | A&B |
| 0 | 1 | 0 | A^B |
| 0 | 1 | 1 | A|B |



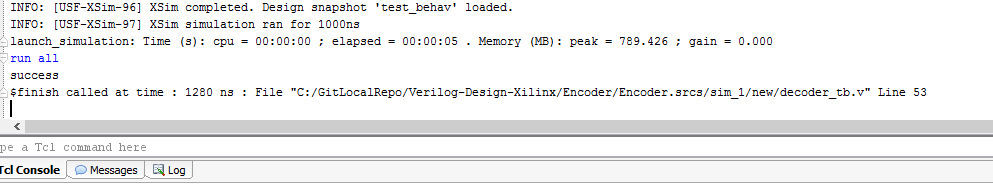
*Figure 2: ALU Block Diagram*

***Simulation Test Plan***

*Encoder:*

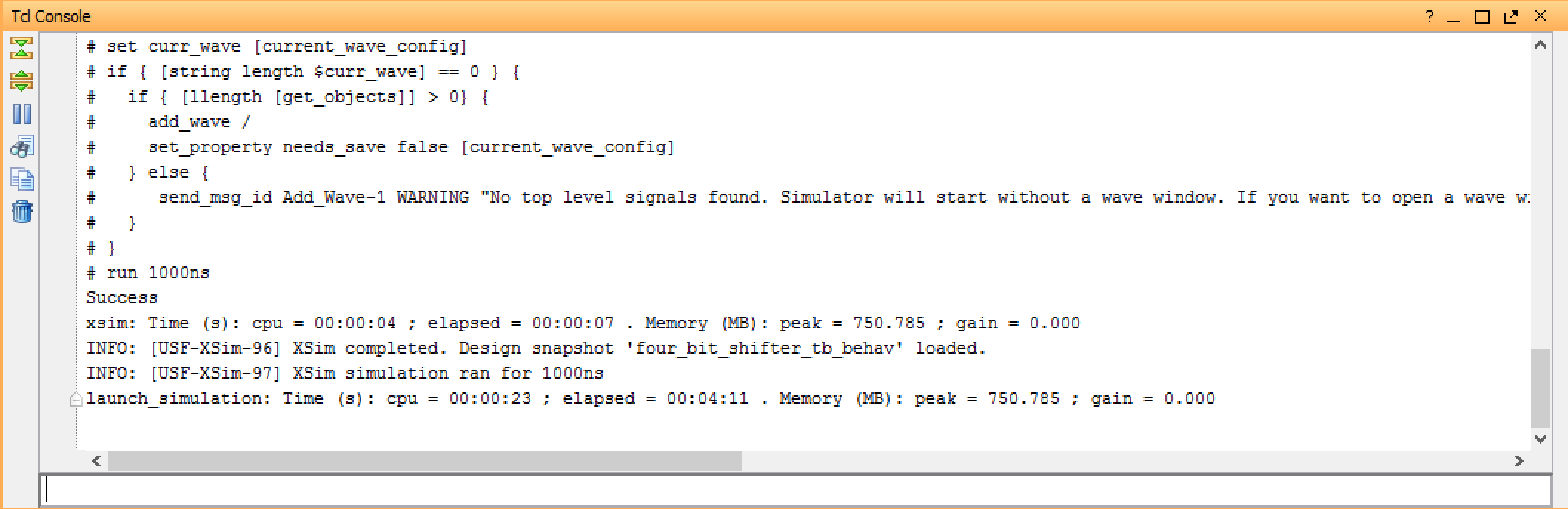
The encoder testing was done using a 8-bit input that ranges from zero to seven. The output is 3-bit and follows the priority of the highest bit being enabled. This will ensure that the simulation is same regardless how the encoder is designed. Figure three shows the confirmed success of the self testing testbench when the full simulation is ran.

*Figure 3: Output of Encoder Simulation*

**

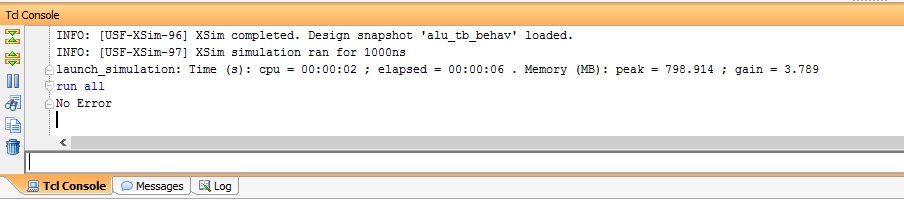
*Shifter:*

Figure four shows the successful output for the shifter.

*Figure 4: Output of Shifter Simulation*

*ALU:*

To test the ALU through simulation a self-testing test bench was programmed. It looped through all arithmetic and logical functions from binary 000 to binary 111. The binary counter was utilized as a control signal. To utilize proper looping and checking nested loops with case statements were used. From figure five we can see that the test bench fully passed and succeeded.

**

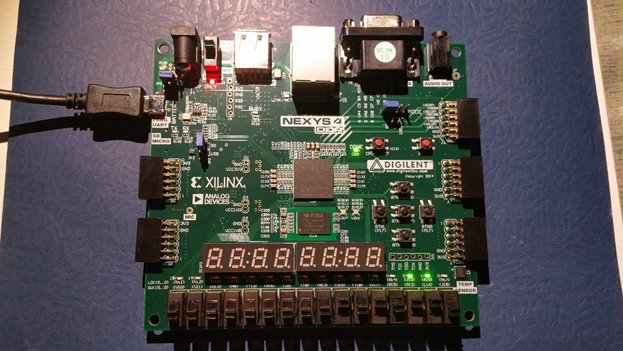
*Figure 5: Output of ALU Simulation*

***FPGA Test Plan***

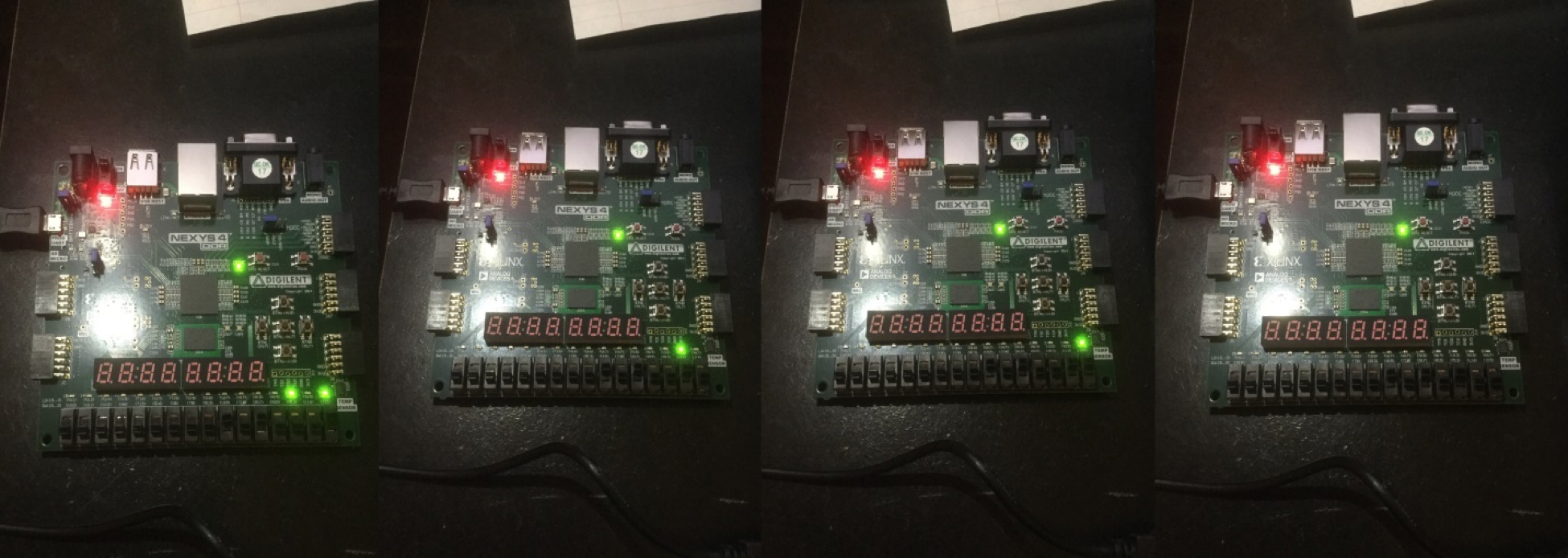
*ALU:* The ALU test plan was to perform the not function which would trigger the zero flag. However during the lab the constraints while would not work when trying to map the RGB led to as a check for overflow or zero output. This led to a failed test plan as the design was not testing upon the board itself fully.

*Encoder:* The encoder is tested by using the rightmost eight switches and 3 LEDs. All switches are considered one bit binary representation in which the three switches represents a 3 bit wide binary counter. This means if the eight switch is set all LEDs will light up as 8 is 111 in binary. The switches are flipped to a correspond binary signal for testing we used 110 which is 6 in decimal. In the figure below we can see that two of the LEDs are lit up representing 110 in binary. This shows a working encoder utilizing the design and proper constraints file.

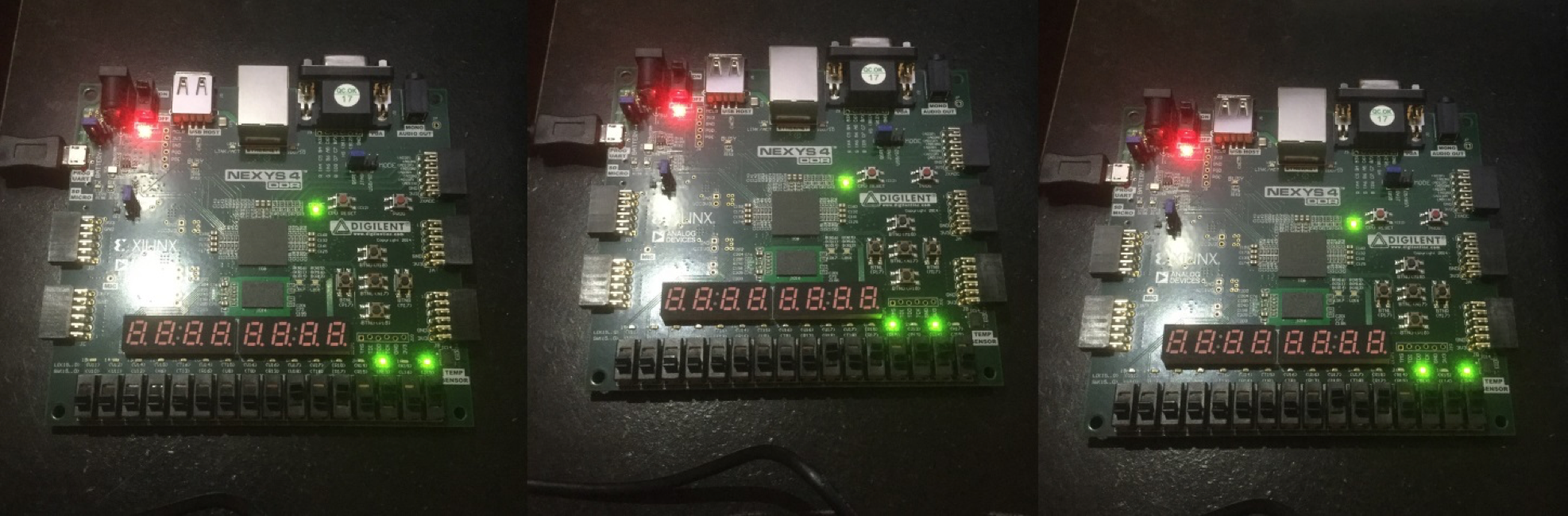
*Figure 6: Encoder Testing on Board*



*Rotator/Shifter:* To test the shifter and rotator, the three rightmost dip switches were used as control signals. From 000-100, the switches would shift to the right and that could be seen on the LEDs of the board. From 101-111, the LEDs would be rotated to the right.



*Figure 7: Encoder Shift Testing on Board*



*Figure 8: Encoder Shift Testing on Board*

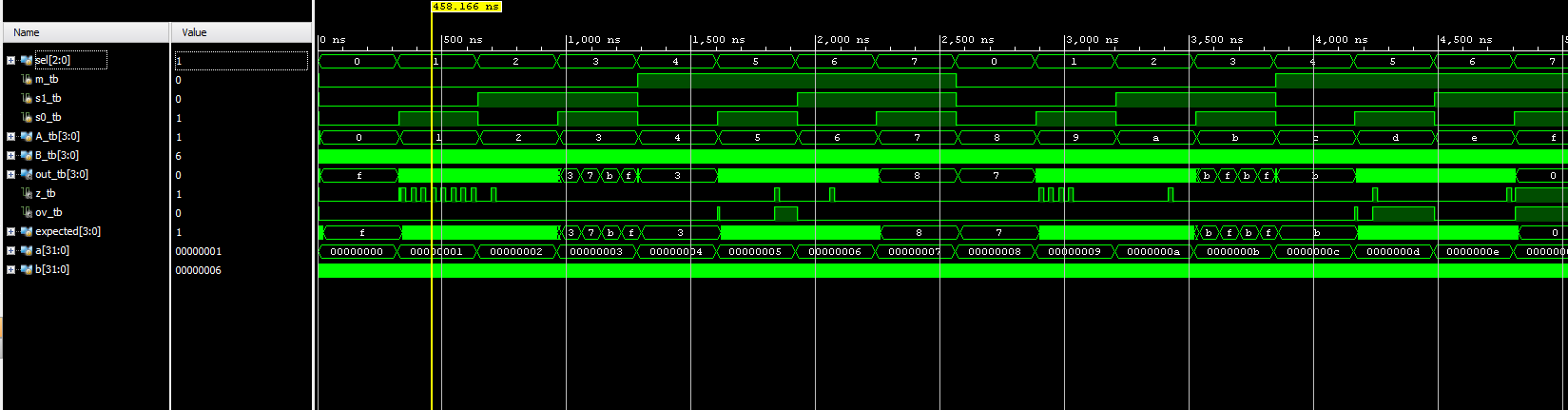
***Conclusion***

The encoder worked as expected. Each case performed properly in the simulation and they functioned properly on the board as well. The board properly generated a 3 bit LED signal based on the 8 inputs that were given to it by the dip switches. When testing the rotator, the results on the board matched what was found on the simulation. The board would shift when the inputs were from 000-100 and they would rotate from 101-111. Additionally, our simulation worked properly according to the console output. We were able to shift and rotate any four bit signal that was input. The ALU had near complete functionality in the end. We could not get the constraint file to work. The simulation worked as intended however the FPGA did not. There was an issue with the overflow and zero flag checking as they were not properly recognized through the RGB led.

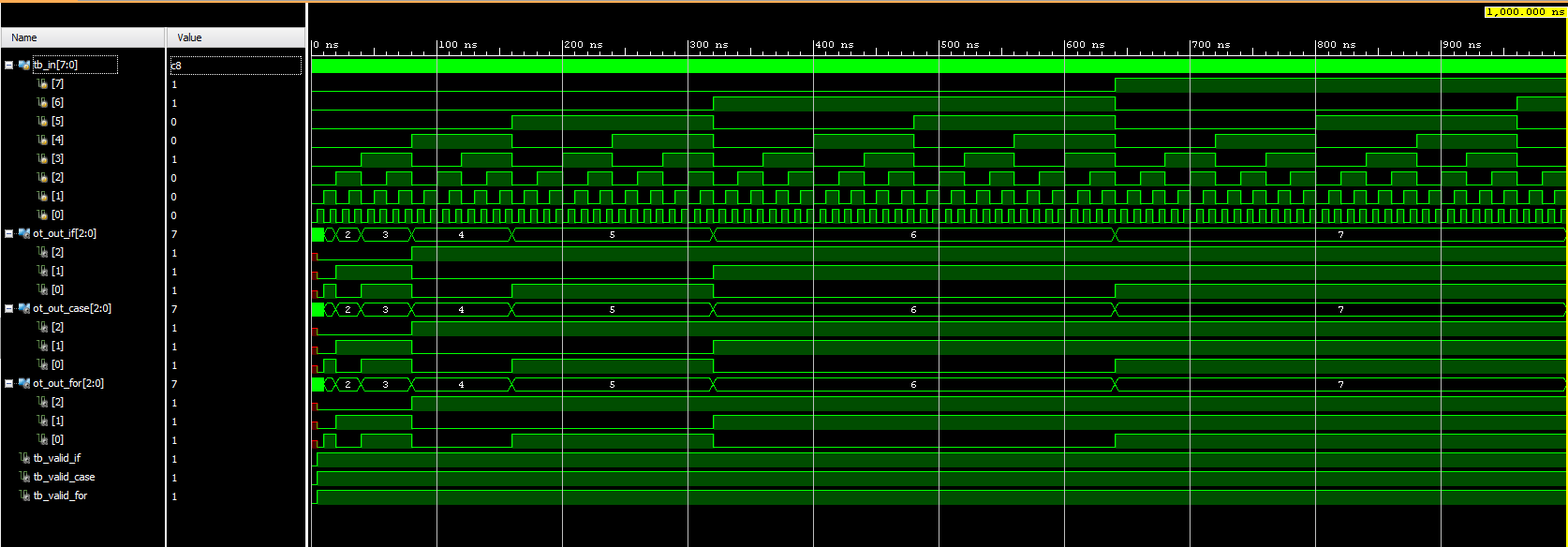
***Appendix***

Additional Simulation Waveforms

*Figure 9: ALU Simulation Waveform*



*Figure 10: Encoder Simulation Waveform*



***Source Code***

**Source Files:**

***encoder if.v***

module decoder\_if (A,Y, Valid);  
input [7:0] A;  
output reg [2:0] Y;  
output reg Valid;  
always@(A)  
begin  
Valid = 1;  
if (A[7]) Y = 7; // same as Y = (A[7] = = 1) ? Y = 7:  
 else if (A[6]) Y = 6; // (A[6] = = 1) ? Y = 6:  
 else if (A[5]) Y = 5; // (A[5] = = 1) ? Y = 5:  
 else if (A[4]) Y = 4; // (A[4] = = 1) ? Y = 4:  
 else if (A[3]) Y = 3; // (A[3] = = 1) ? Y = 3:  
 else if (A[2]) Y = 2; // (A[2] = = 1) ? Y = 2:  
 else if (A[1]) Y = 1; // (A[1] = = 1) ? Y = 1:  
 else if (A[0]) Y = 0; // 0;  
else  
 begin  
 Valid = 0;  
 Y = 3'bx;  
 end  
end  
endmodule

***encoder\_for.v***

module decoder\_for (A,Y, Valid);  
input [7:0] A;  
output reg [2:0] Y;  
output reg Valid;  
integer N;  
always @ (A)  
 begin  
 Valid = 0;  
 Y = 3'bx;  
 for (N = 0; N < 8; N = N + 1)  
 if (A[N])  
 begin  
 Valid = 1;  
 Y = N;  
end  
end  
endmodule

***Encoder\_case.v***

module decoder\_case (A,Y, Valid);  
input [7:0] A;  
output reg [2:0] Y;  
output reg Valid;  
always @ (A)  
begin  
Valid = 1;  
casex (A)  
8'b 1xxxxxxx: Y = 7;  
8'b 01xxxxxx: Y = 6;  
8'b 001xxxxx: Y = 5;  
8'b 0001xxxx: Y = 4;  
8'b 00001xxx: Y = 3;  
8'b 000001xx: Y = 2;  
8'b 0000001x: Y = 1;  
8'b 00000001: Y = 0;  
default:  
begin  
Valid = 0;  
Y = 3'bX;  
end  
endcase  
end  
Endmodule

***Alu.v***

// Verilog Code: An Arithmetic Logic Unit (ALU)  
  
module ALU (m, s1, s0, A, B, out, z, ov);  
   
input m, s1, s0; //This hold 3 bits selection for m, s1, s0  
input [3:0] A, B; //This hold 4 bits input  
output reg [3:0] out; //This hold 4 bits output  
output reg z, ov; //This hold check zero and overflow  
   
always @ (m, s1, s0, A, B)  
begin  
 ov=0;  
 if (m == 1'b0) //logic operation  
 case ({s1, s0})  
 2'b00: out = ~A; //bitwise negation  
 2'b01: out = A & B; //bitwise AND  
 2'b10: out = A ^ B; //bitwise XOR  
 default: out = A | B; //bitwise OR  
 endcase  
 else //arithmetic operation  
 case ({s1, s0})  
 2'b00:  
 begin  
 out = A - 1; // decrement  
 if(A == 0)  
 ov = 1;  
 end   
 2'b01:  
 begin  
 out = A + B; //addition  
 if ((A+B > 15)) // This check overflow  
 ov=1;  
 end   
 2'b10:  
 begin  
 out = A - B; //subtraction  
 end  
 default:  
 begin  
 out = A + 1; // increment  
 if ((A+1 > 15)) //check overflow  
 ov=1;  
 end  
 endcase  
 if (out==0000)  
 z=1; //set z flag=1 if out==0  
 else  
 z=0; //set z flag=0 if out!==0  
end  
endmodule

**Testbench Files:**

***Encoder\_tb.v***

`timescale 1ns / 1ps  
  
module test;  
// declaring variables  
reg[7:0] tb\_in;  
wire [2:0] ot\_out\_if, ot\_out\_case, ot\_out\_for;  
wire tb\_valid\_if;  
wire tb\_valid\_case;  
wire tb\_valid\_for;  
  
// passing in inputs based on function table  
  
decoder\_if DUTA (.A(tb\_in),.Y(ot\_out\_if), .Valid(tb\_valid\_if));  
  
decoder\_case DUTB (.A(tb\_in),.Y(ot\_out\_case), .Valid(tb\_valid\_case));  
  
decoder\_for DUTC (.A(tb\_in),.Y(ot\_out\_for), .Valid(tb\_valid\_for));  
  
initial  
begin  
//tb\_in=0;  
for(integer i=0; i<256; i=i+1)  
 begin  
 tb\_in = i;  
 #5  
 begin  
 casex(tb\_in)  
 8'b 1xxxxxx:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd7 || ot\_out\_for != 3'd7 || ot\_out\_case != 3'd7)  
 $display("error for testbench");  
 8'b 01xxxxx:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd6 || ot\_out\_for != 3'd6 || ot\_out\_case != 3'd6)  
 $display("error for testbench");  
 8'b 001xxxx:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd5 || ot\_out\_for != 3'd5 || ot\_out\_case != 3'd5 )  
 $display("error for testbench");  
 8'b 0001xxx:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd4 || ot\_out\_for != 3'd4 || ot\_out\_case != 3'd4 )  
 $display("error for testbench");  
 8'b 00001xx:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd3 || ot\_out\_for != 3'd3 || ot\_out\_case != 3'd3 )  
 $display("error for testbench");  
 8'b 000001x:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd2 || ot\_out\_for != 3'd2 || ot\_out\_case != 3'd2 )  
 $display("error for testbench");  
 8'b 0000001:  
 if (tb\_valid\_if != 1 || tb\_valid\_for != 1 || tb\_valid\_case != 1 || ot\_out\_if != 3'd1 || ot\_out\_for != 3'd1 || ot\_out\_case != 3'd1 )  
 $display("error for testbench");  
 endcase  
 end  
 end  
 $display("success");  
 $finish;  
end  
endmodule

***Alu\_tb.v***

`timescale 1ns / 1ps

module alu\_tb;

reg [2:0] sel; //This hold 3 bits selection for m, s1, s0

reg m\_tb, s1\_tb, s0\_tb; //This hold 1 bits selector

reg [3:0] A\_tb, B\_tb; //This hold 4 bits input

wire [3:0] out\_tb; //This hold 4 bits output

wire z\_tb, ov\_tb; //This hold check zero and overflow

reg [3:0] expected; //This hold for selftest

integer a; //This hold selector loop

integer b; //This hold input loop

ALU DUT(.m(m\_tb), .s1(s1\_tb), .s0(s0\_tb), .A(A\_tb), .B(B\_tb), .out(out\_tb), .z(z\_tb), .ov(ov\_tb));

initial

begin

for(a = 0; a < 16; a = a + 1)

begin

for(b = 0; b < 16; b = b + 1)

begin

// Assign bits to m, s1, s0

sel = a;

#5;

{m\_tb, s1\_tb, s0\_tb} = sel;

#5;

A\_tb = a;

#5;

B\_tb = b;

#5;

if (m\_tb == 1'b0) //logic operation

case ({s1\_tb, s0\_tb})

2'b00: expected = ~A\_tb; //bitwise negation

2'b01: expected = A\_tb & B\_tb; //bitwise AND;

2'b10: expected = A\_tb ^ B\_tb; //bitwise XOR

default: expected = A\_tb | B\_tb; //bitwise OR

endcase

else //arithmetic operation

case ({s1\_tb, s0\_tb})

2'b00: expected = A\_tb - 1; // decrement

2'b01: expected = A\_tb + B\_tb; //addition

2'b10: expected = A\_tb - B\_tb; //subtraction

default: expected = A\_tb + 1; // increment

endcase

if(out\_tb != expected) //error checking

begin

$display("Error Out");

$stop;

end

end

end

$display("No Error");

$stop;

end

endmodule

**XDC Files:**

***encoder\_fpga.v***

set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports {A[0]}];

set\_property -dict { PACKAGE\_PIN L16 IOSTANDARD LVCMOS33 } [get\_ports {A[1]}];

set\_property -dict { PACKAGE\_PIN M13 IOSTANDARD LVCMOS33 } [get\_ports {A[2]}];

set\_property -dict { PACKAGE\_PIN R15 IOSTANDARD LVCMOS33 } [get\_ports {A[3]}];

set\_property -dict { PACKAGE\_PIN R17 IOSTANDARD LVCMOS33 } [get\_ports {A[4]}];

set\_property -dict { PACKAGE\_PIN T18 IOSTANDARD LVCMOS33 } [get\_ports {A[5]}];

set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports {A[6]}];

set\_property -dict { PACKAGE\_PIN R13 IOSTANDARD LVCMOS33 } [get\_ports {A[7]}];

set\_property -dict { PACKAGE\_PIN H17 IOSTANDARD LVCMOS33 } [get\_ports { Y[0] }];

set\_property -dict { PACKAGE\_PIN K15 IOSTANDARD LVCMOS33 } [get\_ports { Y[1]}];

set\_property -dict { PACKAGE\_PIN J13 IOSTANDARD LVCMOS33 } [get\_ports { Y[2]}];

set\_property -dict { PACKAGE\_PIN R18 IOSTANDARD LVCMOS33 } [get\_ports { Valid }];