CE100 Lab Report 3

Multiplexers

Justin Fortner

Lab 1C

TU/TH 1:30-3:20

10/24/17

Description:

In this lab several different multiplexers were built and implemented in order to create an 8 bit ripple carry adder/subtractor. This circuit takes two 8 bit two's complement numbers input by the switches and then either add or subtract them. Weather to add or subtract is determined by the btnU input. If there is on overflow the decimal points will light up. This lab is also meant to be an introduction to buses.

Methods:

Part 1- Multiplexer:

M4_1e:

This multiplexer takes 4 one bit inputs, 2 one bit selectors and one one bit enable while having one one bit output. First a truth table needed to be created in order to obtain the logic equations needed to implement this 4 to 1 multiplexer. Once this truth table was obtained, one expression can be derived in order to implement the multiplexer. The selectors will choose which input. Then this result is logical ANDed with e in order to give 0 if e=0 or the correct output if e=1.

M8_1e:

This multiplexer takes 8 one bit inputs, 3 one bit selectors and one one bit enable while having one one bit output. In order to obtain the logic for this multiplexer a truth table needed to be created. One logic expression can be obtained from this truth table that correctly implements this 8 to 1 multiplexer. Once again the selectors will chose the input and the result will be ANDed with e in order to give 0 if e=0 or the correct output if e=1.

M2_1x8:

This multiplexer takes 2 eight bit inputs and one one bit selector while having one one bit output. Once again a truth table is obtained in order to derive a logic equation for this 2 to 1 multiplexer. The selector bit of this multiplexer must be extended to 8 bits so that all 8 bits of in0 or in1 can be passed through to the output.

Part 2- Adder Subtractor:

The AddSub8 module takes in two eight bit inputs, the input of a button and outputs one 8 bit sum as well as the one bit overflow value. The adder subtractor is implemented using 8 full adders. The initial carry in bit is initialized to 0. After this the carry out of one full adder is passed into the next full adder. Thus creating the ripple carry adder. If there is an overflow on the last full adder that signal is discarded. Within this module is also the logic to turn a positive binary number into its negative two's complement form. This is done by negating each input bit and then adding one to this. The m2_1x8 multiplexer is used to decide whether to use the positive or negative input thus creating a subtractor as well. The negative form is achieved by pressing btnU.

Full Adder:

The fullAdder module has 3 one bit inputs and two one bit outputs. The full adder is made up of two 4 to 1 multiplexers with their enables permanently set to 1. In order to obtain the logic expressions for this full adder a truth table must be made. Two selectors and a carry in make up the inputs. A sum is the output of one multiplexer while the carry out is the output of the other.

Part 3- 7-Segment Display:

The hex7seg module takes in one 4 bit input and on one bit enable switch while output in all 7 segments of the display. The hex7seg consists of seven 8 to 1 multiplexers. The inputs for these multiplexers are obtained through a several step process. First, a truth table using n3, n2, n1, and n0 as inputs representing a number 0-15. In order for these numbers to be represented on the 7-segment display specific segments of the display must be lit up. The 7 segments, A-G are on when high. From these necessary outputs that correspond to the input number K-maps are able to be produced. Each segment of the display, A-G, will have their own K-map. Choose 3 n values to represent selectors of the 8 to 1 multiplexer. The inputs to the 8 to 1 multiplexer are derived from this K-map. Once all eight inputs are found for all 7 segments of the display they are input into the seven separate 8 to 1 multiplexers. The enable switch of the 8 to 1 multiplexer is dependant on the enable switch input of the hex7seg.

Part 4- Top Level Schematic:

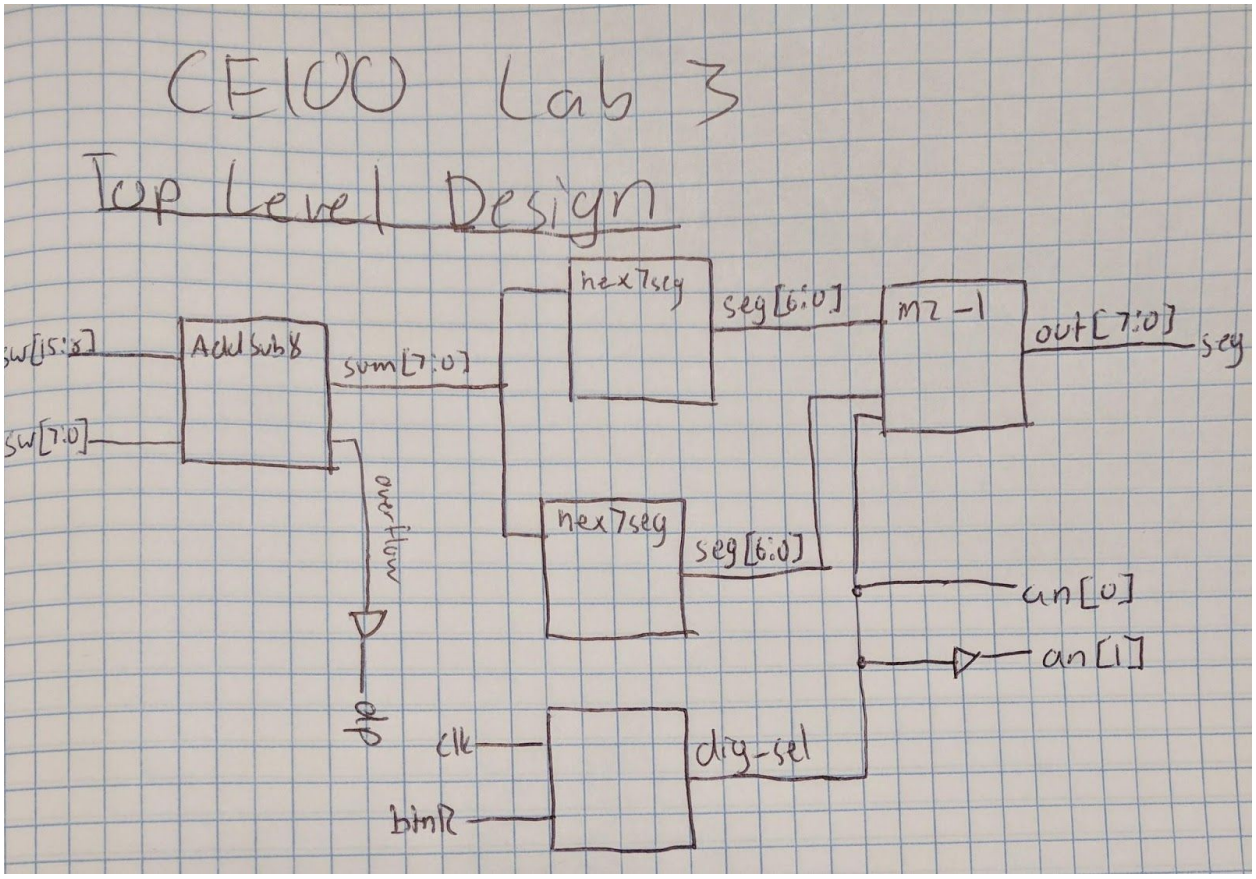
The Lab3Top module takes in the input of all 16 switches, the btnU, btnR and clock while outputting the 6 bit value for the 7 segment display, the overflow indicator and the enables for the display to turn on. The first two units of the display are permanently turned off. The other two units alternate refresh whenever the dig_sel is changed. The logic for the overflow is inverted because the decimal point indicator is on when low and off when high. This module consists of one AddSub8 instance and two hex7seg instances, one for each display unit that is turned on. The provided lab3_digsel instance is also added to this module. Switches 15 to 8 are tied to input A of the AddSub8 while 7 to 0 are tied to input B of the AddSub8. The sum output of the add sub is connected by wires to the two hex7seg instances. The first 4 bits of the sum into one instance and the last 4 bits into the other. A 2 to 1 mux is added in order to decide which hex7seg instance to display. One will display when dig_sel is high and one when dig_sel is low.

Results:

Design:

Lab3Top:

The top level design takes an input from the switches on the board and splits them into two separate numbers. The sum is then calculated and output into the hex 7 segments. The number is then displayed on one of the two units turned on on the 7 segment display.



AddSub8:

The adder consists of 8 full adders connected together in order to create a ripple carry adder.

sel1	sel0	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1

1	1	1	1	1

Overflow:

The overflow logic looks at the leftmost bit of the sum, A and B. This will calculate iof there is an overflow or not.

S[7]	A[7]	B[7]
1	0	0
0	1	1

Overflow = ((S[7]&~A[7]&~B[7])|(~S[7]&A[7]&B[7]))

FullAdder:

The full adder is comprised of two 4 to 1 multiplexers. one mux is used to calculate the sum and the other is used to calculate the carry out.

M4_1e:

The 8 to 1 multiplexer is made up of the SOP equations derived from the following truth table.

sel1	sel0	in3	in2	in1	in0	e	o
0	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0
1	0	0	1	0	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	in0
0	1	0	0	1	0	1	in1
1	0	0	1	0	0	1	in2
1	1	1	0	0	0	1	in3

o = e &
((in[0]&~sel[1]&~sel[0])|(in[1]&~sel[1]&sel[0])|(in[2]&sel[1]&~sel[0])|(in[3]&sel[1]&sel[0]))

Hex7seg:

The hex7seg takes the output of the ripple carry adder and converts it into a hex value that can be displayed on the board. The 7 segment display is made up of 7 8 to 1 multiplexers. Each one is responsible for one of the 7 segments in each unit. The inputs for the multiplexers is derived from K-maps filled with values from the truth table.

#	n3	n2	n1	n0	A	B	C	D	E	F	G
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
A	1	0	1	0	1	1	1	0	0	1	1
B	1	0	1	1	1	1	1	0	1	1	1
C	1	1	0	0	0	0	1	1	1	1	1
D	1	1	0	1	0	1	1	1	1	0	1
E	1	1	1	0	1	0	0	1	1	1	1
F	1	1	1	1	1	0	0	0	1	1	1

K-Map Results

A

1'b0, 1'b1, 1'b0, 1'b0, 1'b0, ~n[0], 1'b0, n[0]

B

1'b1, ~n[0], 1'b0, 1'b0, ~n[0], n[0], 1'b0, 1'b0

C

1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, ~n[0], 1'b0

D

n[0], 1'b0, 1'b1, n[0], n[0], ~n[0], 1'b0, n[0]

E

1'b0, 1'b0, ~n[0], n[0], n[0], 1'b1, n[0], n[0]

F

1'b0, n[0], 1'b0, 1'b0, n[0], 1'b0, 1'b1, n[0]

G

1'b0, 1'b0, 1'b0, 1'b0, n[0], 1'b0, 1'b0, 1'b1

M8_1e:

The 8 to 1 multiplexer is made up of the SOP equations derived from the following truth table.

sel2	sel1	sel0	in7	in6	in5	in4	in3	in2	in1	in0	e	o
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	in0
0	0	1	0	0	0	0	0	0	1	0	1	in1
0	1	0	0	0	0	0	0	1	0	0	1	in2
0	1	1	0	0	0	0	1	0	0	0	1	in3
1	0	0	0	0	0	1	0	0	0	0	1	in4
1	0	1	0	0	1	0	0	0	0	0	1	in5

1	1	0	0	1	0	0	0	0	0	0	1	in6
1	1	1	1	0	0	0	0	0	0	0	1	in7

o = e & ((in[0]&~sel[2]&~sel[1]&~sel[0])|(in[1]&~sel[2]&~sel[1]&sel[0])
|(in[2]&~sel[2]&sel[1]&~sel[0])|(in[3]&~sel[2]&sel[1]&sel[0])
|(in[4]&sel[2]&~sel[1]&~sel[0])|(in[5]&sel[2]&~sel[1]&sel[0])
|(in[6]&sel[2]&sel[1]&~sel[0])|(in[7]&sel[2]&sel[1]&sel[0]))

M2_1x8:

The 2 to 1 multiplexer is made up of the SOP equations from the following truth table.

sel	in1	in0	e	o
0	0	1	0	0
1	1	0	0	0
0	0	1	1	in0
1	1	0	1	in1

o = (~extend & in0) | (extend & in1)

Testing & Simulation:

I tested my design to make sure it worked by using various combinations of buttons and switches to result in my desired output. I chose inputs that changed the values of both my displays to all possible outcomes. These inputs were chosen to make sure my displays and hex7seg were working correctly. I then implemented a case where overflow would occur in order to test my overflow logic. I then started to introduce negative numbers. This included subtracting negative from negative as well as negative from positive numbers. Lastly I tested adding various numbers together to solidify that my adder was working. The biggest problem that I encountered during testing was I had originally thought that I could just use the carry out from my last full adder to signify overflow. The issue with this is that you can get a carry out without going over the allowed output for the display. This made me realize that I needed to look at the leftmost bits of my sum as well as my inputs in order to correctly display the overflow indicator.

Lab Questions:

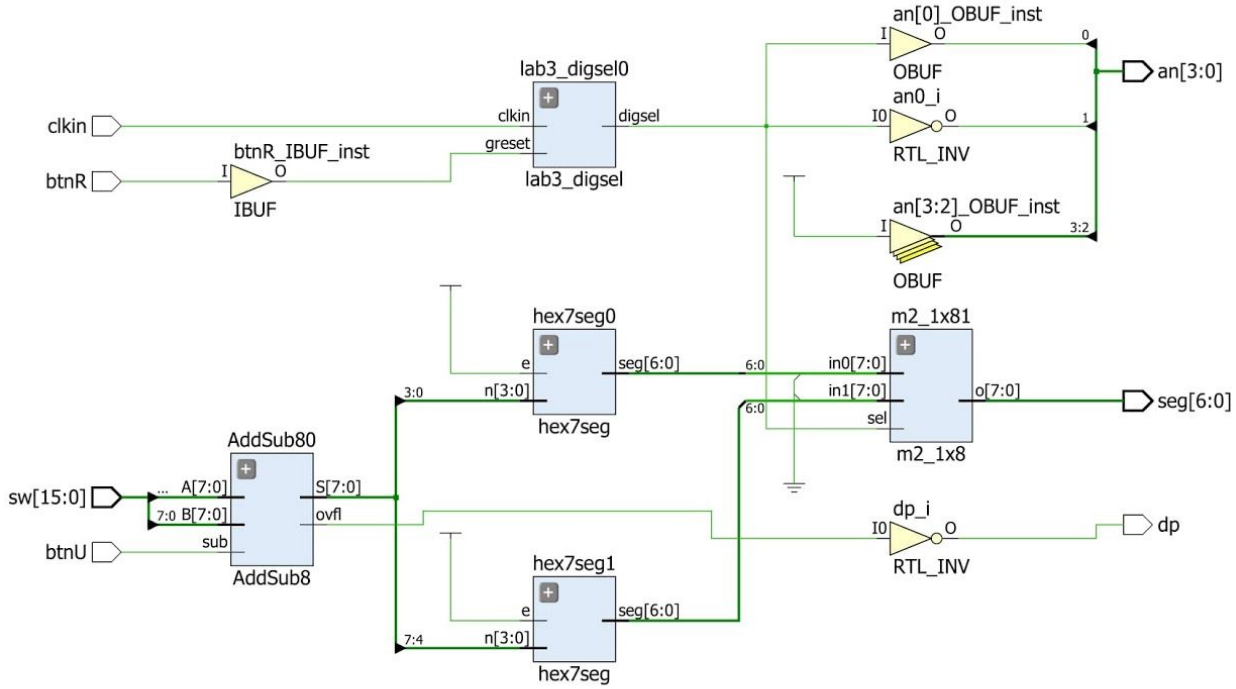
1) Digsel is changing at a rate of 1 time every 4 microseconds.
2) I did notice flickering in the 7-segment.

Conclusion:

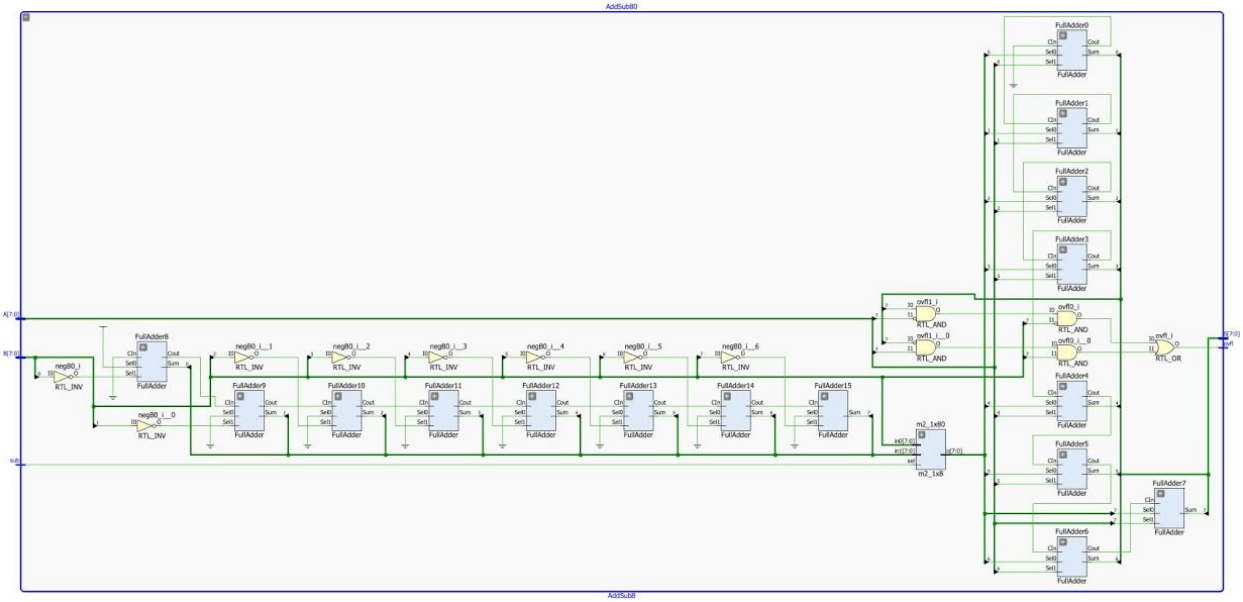
In this lab I learned how useful buses can be. They allow code to be simplified and allow for the use of multiple inputs/outputs at once. I also learned how multiplexers can be used to implement adders. Clocks were a small but vital portion of this lab. I learned how to deal with those as well. One of the best skills I learned from this lab was how to use the simulator. This is an extremely useful tool when trying to debug code. At first I had a lot of difficulties. I had the original impression that this lab would be easy and did not follow the lab manual closely. This led to many careless mistakes and errors. So many in fact that I restarted the entire lab. This time following the lab manual very closely and I was able to understand the material better, finish faster, and complete the lab as it was designed. I would love to go back in this lab and optimize the conversion of a positive number to its negative two's complement form.

Appendix: Started Next Page

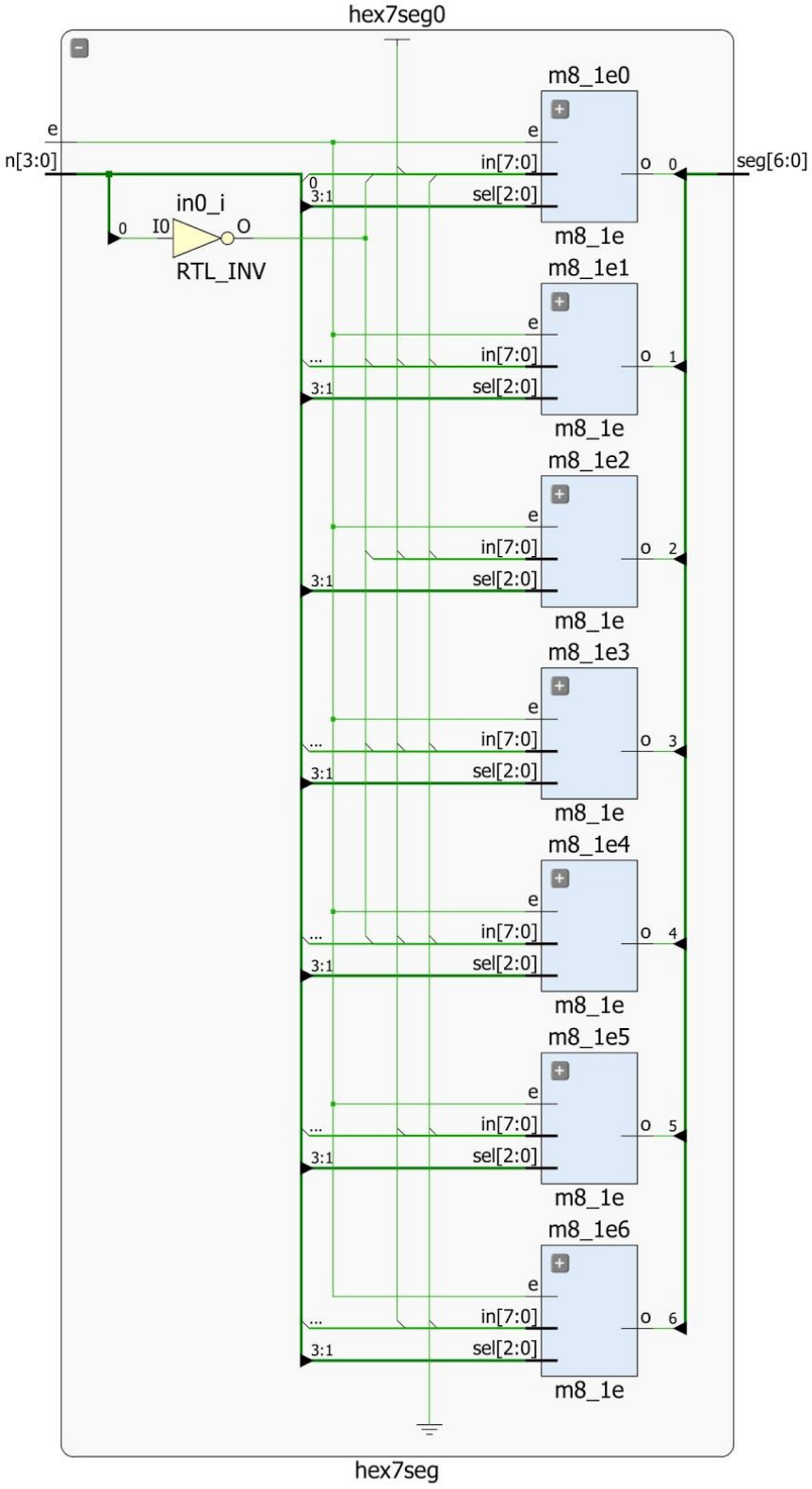
Appendix A: Top Level Schematic:



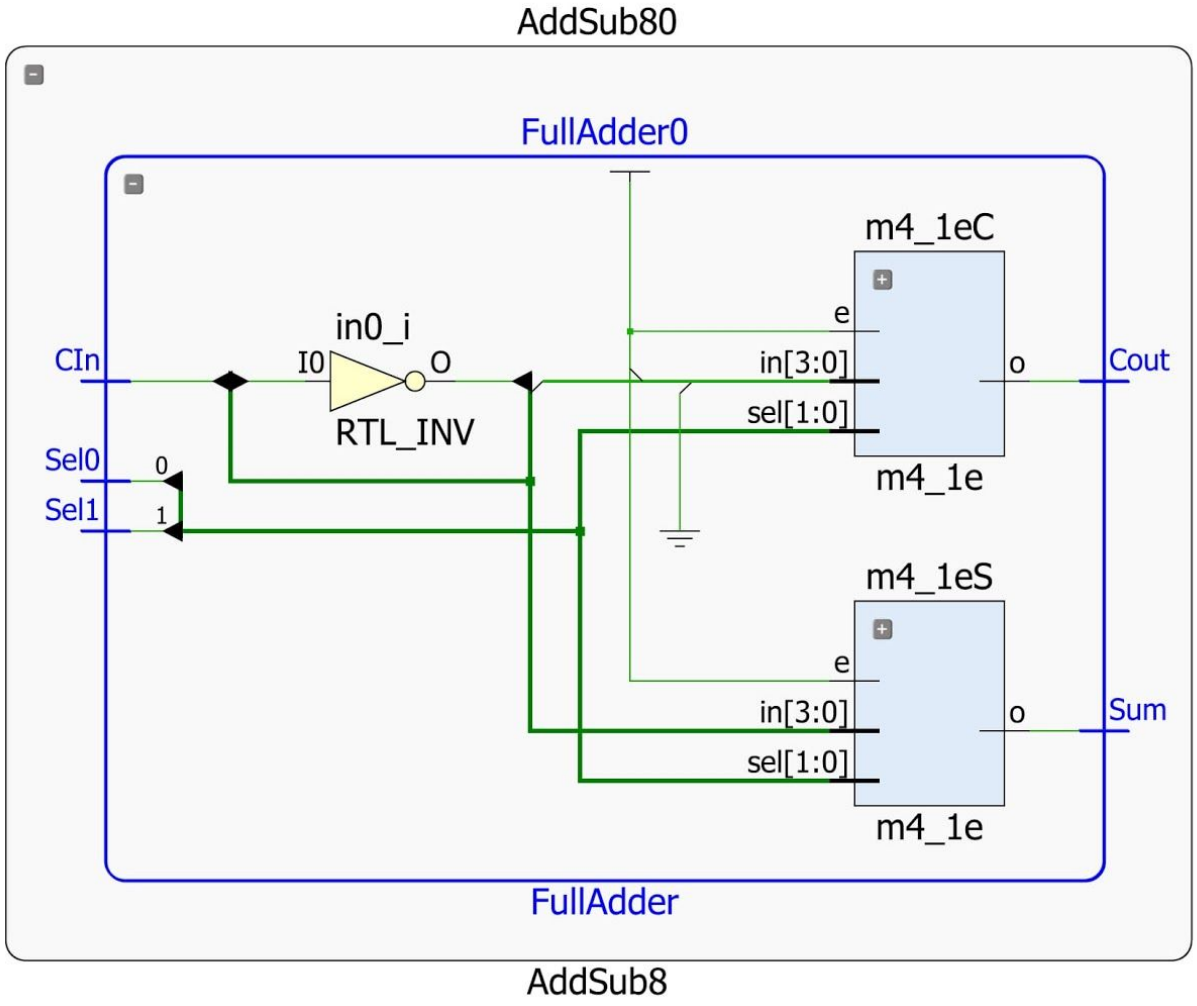
Appendix B: Addsub8 Schematic:



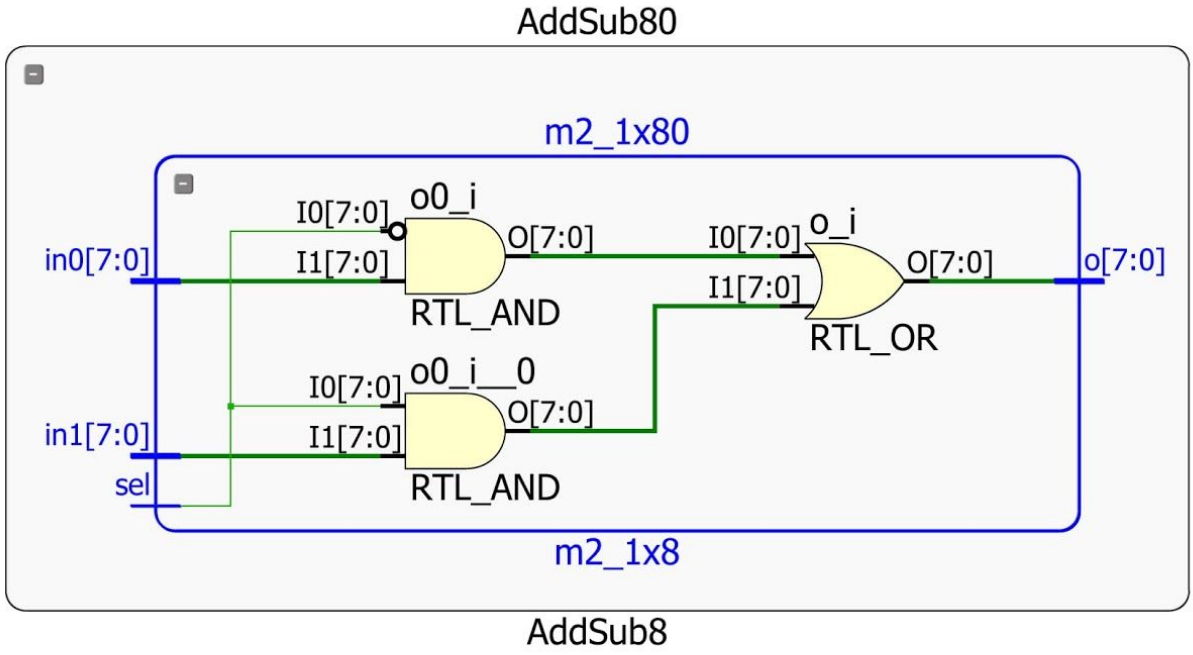
Appendix C: Hex7seg Schematic:



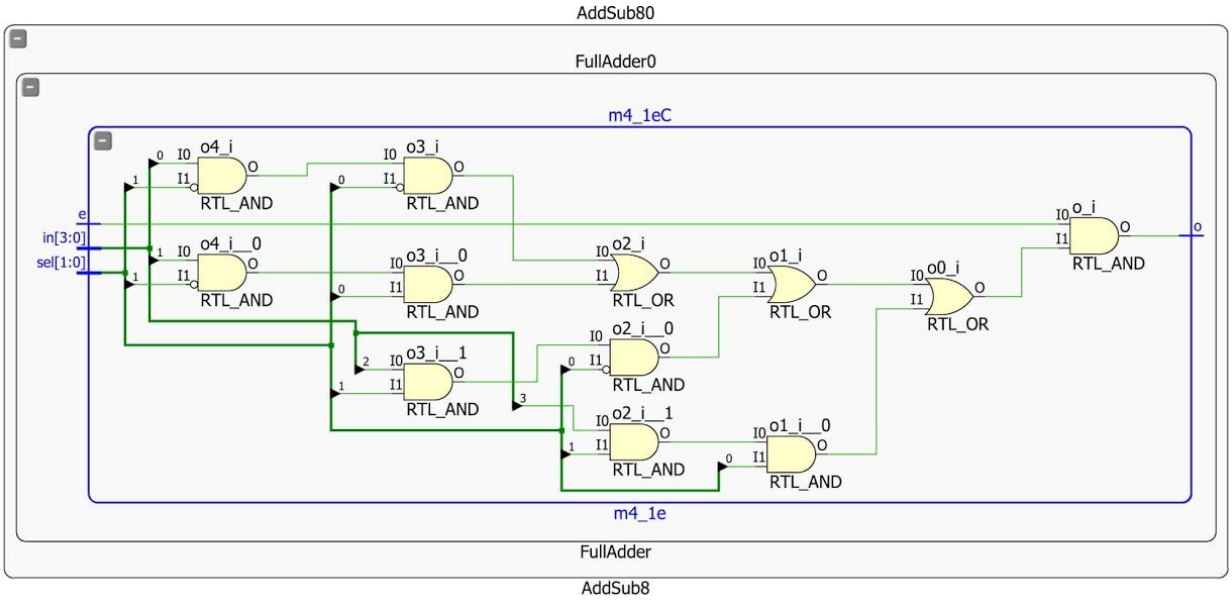
Appendix D: Full Adder Schematic:



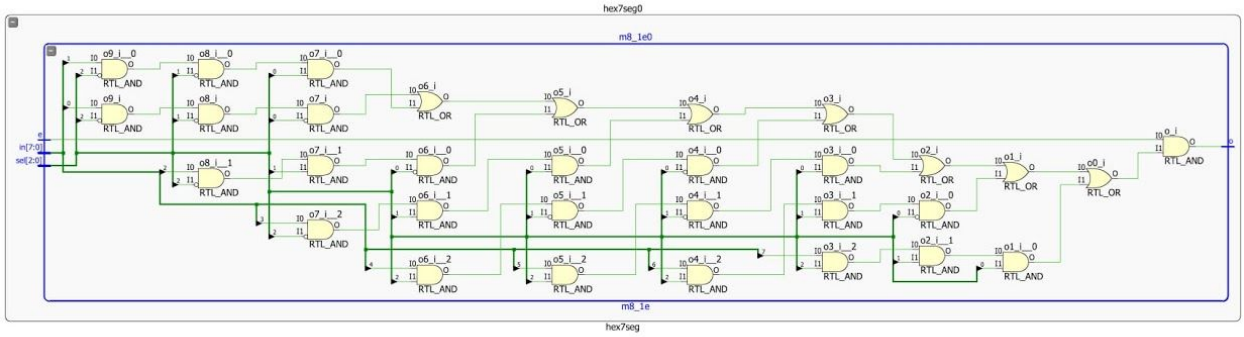
Appendix E: 2 to1 Multiplexer Schematic:



Appendix F: 4 to 1 Multiplexer Schematic:



Appendix G: 8 to 1 Multiplexer Schematic:



Appendix H: Top Level Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 09:42:18 PM
// Design Name:
// Module Name: Lab3Top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Lab3Top(
    input [15:0] sw,
    input btnU,
    input btnR,
    input clkin,
    output [6:0] seg,
    output dp,
    output [3:0] an
    );

    wire [7:0] sum;
    wire [6:0] seg0;
    wire [6:0] seg1;
    wire dig_sel;
    wire overflow;

    lab3_digsel lab3_digsel0 (.clkin(clkin), .greset(btnR), .digsel(dig_sel));
    AddSub8 AddSub80 (.A(sw[15:8]), .B(sw[7:0]), .sub(btnU), .S(sum), .ovfl(overflow)
    hex7seg hex7seg0 (.n(sum[3:0]), .e(1'b1), .seg(seg0));
    hex7seg hex7seg1 (.n(sum[7:4]), .e(1'b1), .seg(seg1));
    m2_1x8 m2_1x81 (.in0({1'b0, seg0}), .in1({1'b0, seg1}), .sel(dig_sel), .o(seg));
    assign an[0] = dig_sel;
    assign an[1] = ~dig_sel;
    assign an[2] = 1;
```

```
    assign an[3] = 1;
    assign dp = ~overflow;

endmodule
```

Appendix I: AddSub8 Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 07:26:37 PM
// Design Name:
// Module Name: AddSub8
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module AddSub8(
    input [7:0] A,
    input [7:0] B,
    input sub,
    output [7:0] S,
    output ovfl
    );

    wire [6:0] cw;
    wire [6:0] cw1;
    wire [7:0] Bsel;
    wire [7:0] negB;
    wire [7:0] negBFinal;
    wire overflow;
    wire overflow1;

    assign ovfl = ((S[7]&~A[7]&~B[7])|(~S[7]&A[7]&B[7]));
    assign negB = {~B[7], ~B[6], ~B[5], ~B[4], ~B[3], ~B[2], ~B[1], ~B[0]};


    FullAdder FullAdder8 (.Sel1(negB[0]), .Sel0(1'b1), .CIn(1'b0), .Cout(cw1[0]),
.Sum(negBFinal[0]));
    FullAdder FullAdder9 (.Sel1(negB[1]), .Sel0(1'b0), .CIn(cw1[0]), .Cout(cw1[1]),
.Sum(negBFinal[1]));
```

```
    FullAdder FullAdder10 (.Sel1(negB[2]), .Sel0(1'b0), .CIn(cw1[1]), .Cout(cw1[2]),
.Sum(negBFinal[2]));
    FullAdder FullAdder11 (.Sel1(negB[3]), .Sel0(1'b0), .CIn(cw1[2]), .Cout(cw1[3]),
.Sum(negBFinal[3]));
    FullAdder FullAdder12 (.Sel1(negB[4]), .Sel0(1'b0), .CIn(cw1[3]), .Cout(cw1[4]),
.Sum(negBFinal[4]));
    FullAdder FullAdder13 (.Sel1(negB[5]), .Sel0(1'b0), .CIn(cw1[4]), .Cout(cw1[5]),
.Sum(negBFinal[5]));
    FullAdder FullAdder14 (.Sel1(negB[6]), .Sel0(1'b0), .CIn(cw1[5]), .Cout(cw1[6]),
.Sum(negBFinal[6]));
    FullAdder FullAdder15 (.Sel1(negB[7]), .Sel0(1'b0), .CIn(cw1[6]),
.Cout(overflow1), .Sum(negBFinal[7]));


    m2_1x8 m2_1x80 (.in0(B), .in1(negBFinal), .sel(sub), .o(Bsel));

    FullAdder FullAdder0 (.Sel1(A[0]), .Sel0(Bsel[0]), .CIn(1'b0), .Cout(cw[0]),
.Sum(S[0]));
    FullAdder FullAdder1 (.Sel1(A[1]), .Sel0(Bsel[1]), .CIn(cw[0]), .Cout(cw[1]),
.Sum(S[1]));
    FullAdder FullAdder2 (.Sel1(A[2]), .Sel0(Bsel[2]), .CIn(cw[1]), .Cout(cw[2]),
.Sum(S[2]));
    FullAdder FullAdder3 (.Sel1(A[3]), .Sel0(Bsel[3]), .CIn(cw[2]), .Cout(cw[3]),
.Sum(S[3]));
    FullAdder FullAdder4 (.Sel1(A[4]), .Sel0(Bsel[4]), .CIn(cw[3]), .Cout(cw[4]),
.Sum(S[4]));
    FullAdder FullAdder5 (.Sel1(A[5]), .Sel0(Bsel[5]), .CIn(cw[4]), .Cout(cw[5]),
.Sum(S[5]));
    FullAdder FullAdder6 (.Sel1(A[6]), .Sel0(Bsel[6]), .CIn(cw[5]), .Cout(cw[6]),
.Sum(S[6]));
    FullAdder FullAdder7 (.Sel1(A[7]), .Sel0(Bsel[7]), .CIn(cw[6]), .Cout(overflow),
.Sum(S[7]));


endmodule
```

Appendix J: Hex7Seg Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 08:22:58 PM
// Design Name:
// Module Name: hex7seg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module hex7seg(
    input [3:0] n,
    input e,
    output [6:0] seg
    );

    //m8_1e m8_1e0 (.in({~n[0], 1'b1, n[0], 1'b1, 1'b1, 1'b1, 1'b0, 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[0]));
    //m8_1e m8_1e1 (.in({1'b1, 1'b1, ~n[0], n[0], 1'b1, 1'b1, n[0], 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[1]));
    //m8_1e m8_1e2 (.in({1'b1, n[0], 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[2]));
    //m8_1e m8_1e3 (.in({~n[0], 1'b1, n[0], ~n[0], ~n[0], 1'b0, 1'b1, ~n[0]}),
.sel(n[3:0]), .e(e), .o(seg[3]));
    //m8_1e m8_1e4 (.in({~n[0], ~n[0], 1'b0, ~n[0], ~n[0], n[0], 1'b1, 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[4]));
    //m8_1e m8_1e5 (.in({~n[0], 1'b0, 1'b1, ~n[0], 1'b1, 1'b1, ~n[0], 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[5]));
    //m8_1e m8_1e6 (.in({1'b0, 1'b1, 1'b1, ~n[0], 1'b1, 1'b1, 1'b1, 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[6]));
    //m8_1e m8_1e0 (.in({1'b1, 1'b0, 1'b1, 1'b1, 1'b1, n[0], 1'b1, ~n[0]}),
.sel(n[3:0]), .e(e), .o(seg[0]));
    //m8_1e m8_1e1 (.in({1'b0, n[0], 1'b1, 1'b1, n[0], ~n[0], 1'b1, 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[1]));
```

```
    //m8_1e m8_1e2 (.in({1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, n[0], 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[2]));
    //m8_1e m8_1e3 (.in({~n[0], 1'b1, 1'b0, ~n[0], ~n[0], n[0], 1'b1, ~n[0]}),
.sel(n[3:0]), .e(e), .o(seg[3]));
    //m8_1e m8_1e4 (.in({1'b1, 1'b1, n[0], ~n[0], ~n[0], 1'b0, ~n[0], ~n[0]}),
.sel(n[3:0]), .e(e), .o(seg[4]));
    //m8_1e m8_1e5 (.in({1'b1, ~n[0], 1'b1, 1'b1, ~n[0], 1'b1, 1'b0, ~n[0]}),
.sel(n[3:0]), .e(e), .o(seg[5]));
    //m8_1e m8_1e6 (.in({1'b1, 1'b1, 1'b1, 1'b1, ~n[0], 1'b1, 1'b1, 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[6]));
    m8_1e m8_1e0 (.in({1'b0, 1'b1, 1'b0, 1'b0, 1'b0, ~n[0], 1'b0, n[0]}),
.sel(n[3:1]), .e(e), .o(seg[0]));
    m8_1e m8_1e1 (.in({1'b1, ~n[0], 1'b0, 1'b0, ~n[0], n[0], 1'b0, 1'b0}),
.sel(n[3:1]), .e(e), .o(seg[1]));
    m8_1e m8_1e2 (.in({1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, ~n[0], 1'b0}),
.sel(n[3:1]), .e(e), .o(seg[2]));
    m8_1e m8_1e3 (.in({n[0], 1'b0, 1'b1, n[0], n[0], ~n[0], 1'b0, n[0]}),
.sel(n[3:1]), .e(e), .o(seg[3]));
    m8_1e m8_1e4 (.in({1'b0, 1'b0, ~n[0], n[0], n[0], 1'b1, n[0], n[0]}),
.sel(n[3:1]), .e(e), .o(seg[4]));
    m8_1e m8_1e5 (.in({1'b0, n[0], 1'b0, 1'b0, n[0], 1'b0, 1'b1, n[0]}),
.sel(n[3:1]), .e(e), .o(seg[5]));
    m8_1e m8_1e6 (.in({1'b0, 1'b0, 1'b0, 1'b0, n[0], 1'b0, 1'b0, 1'b1}),
.sel(n[3:1]), .e(e), .o(seg[6]));
    //m8_1e m8_1e0 (.in({n[0], 1'b0, ~n[0], 1'b0, 1'b0, 1'b0, 1'b1, 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[0]));
    //m8_1e m8_1e1 (.in({1'b0, 1'b0, n[0], ~n[0], 1'b0, 1'b0, ~n[0], 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[1]));
    //m8_1e m8_1e2 (.in({1'b0, ~n[0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1}),
.sel(n[3:0]), .e(e), .o(seg[2]));
    //m8_1e m8_1e3 (.in({n[0], 1'b0, ~n[0], n[0], n[0], 1'b1, 1'b0, n[0]}),
.sel(n[3:0]), .e(e), .o(seg[3]));
    //m8_1e m8_1e4 (.in({n[0], n[0], 1'b1, n[0], n[0], ~n[0], 1'b0, 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[4]));
    //m8_1e m8_1e5 (.in({n[0], 1'b1, 1'b0, n[0], 1'b0, 1'b0, n[0], 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[5]));
    //m8_1e m8_1e6 (.in({1'b1, 1'b0, 1'b0, n[0], 1'b0, 1'b0, 1'b0, 1'b0}),
.sel(n[3:0]), .e(e), .o(seg[6]));

endmodule
```

Appendix K: Full Adder Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 07:38:04 PM
// Design Name:
// Module Name: FullAdder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module FullAdder(
    input Sel0,
    input Sel1,
    input CIn,
    output Cout,
    output Sum
    );

    m4_1e m4_1eS (.in({CIn, ~CIn, ~CIn, CIn}), .sel({Sel1, Sel0}), .e(1'b1), .o(Sum))
    m4_1e m4_1eC (.in({1'b1, CIn, CIn, 1'b0}), .sel({Sel1, Sel0}), .e(1'b1), .o(Cout)

endmodule
```

Appendix L: 2 to 1 Multiplexer Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 07:01:14 PM
// Design Name:
// Module Name: m2_1x8
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module m2_1x8(
    input [7:0] in0,
    input [7:0] in1,
    input sel,
    output [7:0] o
    );

    wire [7:0] extend;
    assign extend = {sel, sel, sel, sel, sel, sel, sel, sel};

    assign o = (~extend & in0) | (extend & in1);

endmodule
```

Appendix M: 4 to 1 Multiplexer Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 06:39:33 PM
// Design Name:
// Module Name: m4_1e
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module m4_1e(
    input [3:0] in,
    input [1:0] sel,
    input e,
    output o
    );

    assign o = e &
((in[0]&~sel[1]&~sel[0])|(in[1]&~sel[1]&sel[0])|(in[2]&sel[1]&~sel[0])|(in[3]&sel[1]&
sel[0]));

endmodule
```

Appendix N: 8 to 1 Multiplexer Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 06:54:14 PM
// Design Name:
// Module Name: m8_1e
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module m8_1e(
    input [7:0] in,
    input [2:0] sel,
    input e,
    output o
    );

    assign o = e &
((in[0]&~sel[2]&~sel[1]&~sel[0])|(in[1]&~sel[2]&~sel[1]&sel[0])|(in[2]&~sel[2]&sel[1]
&~sel[0])|(in[3]&~sel[2]&sel[1]&sel[0])|(in[4]&sel[2]&~sel[1]&~sel[0])|(in[5]&sel[2]&
~sel[1]&sel[0])|(in[6]&sel[2]&sel[1]&~sel[0])|(in[7]&sel[2]&sel[1]&sel[0]));

endmodule
```

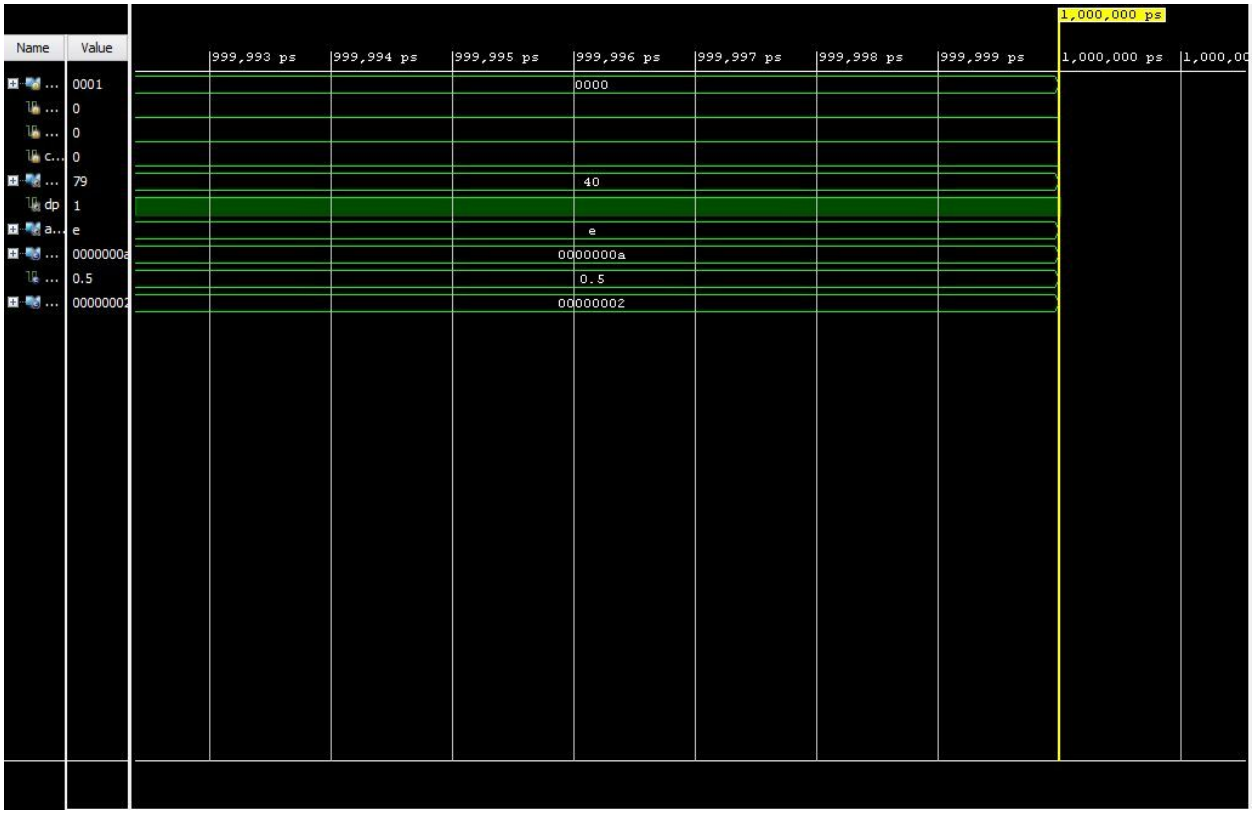
Appendix O: Testbench Verilog:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/19/2017 10:19:05 PM
// Design Name:
// Module Name: Lab3Sim
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Lab3Sim(

    );

    reg [15:0] sw;
    reg btnU, btnR, clkin;
    wire [6:0] seg;
    wire dp;
    wire [3:0] an;

    Lab3Top Lab3Top0(.sw(sw), .btnU(btnU), .btnR(btnR), .clkin(clkin), .seg(seg),
.dp(dp), .an(an));

     parameter PERIOD = 10;
       parameter real DUTY_CYCLE = 0.5;
       parameter OFFSET = 2;

       initial    // Clock process for clkin
       begin
           #OFFSET
             clkin = 1'b1;
           forever
           begin
               #(PERIOD-(PERIOD*DUTY_CYCLE)) clkin = ~clkin;
```

```
    end
 end

initial
begin
btnU = 1'b0;
btnR = 1'b0;
  // add your stimuli here
  // to set signal foo to value 0 use
  // foo = 1'b0;
  // to set signal foo to value 1 use
  // foo = 1'b1;
  //always advance time my multiples of 100ns
  // to advance time by 100ns use the following line


  sw [15:0] = 16'b0;
  #1000;
  //1
  sw[0] =  1'b1;
  #100;
  //2
  sw[0] =  1'b0;
  sw[1] = 1'b1;
  #100;
  //3
  sw[0] =  1'b1;
  #100;
  //4
  sw[0] = 1'b0;
  sw[1] = 1'b0;
  sw[2] = 1'b1;
  #100;
  //5
  sw[0] = 1'b1;
  #100;
  //6
  sw[0] = 1'b0;
  sw[1] = 1'b1;
  #100;
  //7
  sw[0] = 1'b1;
  #100;
  //8
  sw[0] = 1'b0;
  sw[1] = 1'b0;
```

```
        sw[2] = 1'b0;
        sw[3] = 1'b1;
      end

endmodule
```

Appendix P: Testbench Waveform:



Appendix Q: Lab Notebook Pages:

Lab 3 CE100

Calculate sum or difference of two numbers represented in 8-bit 2's comp

a[7:0] = sw[7] to sw[0]

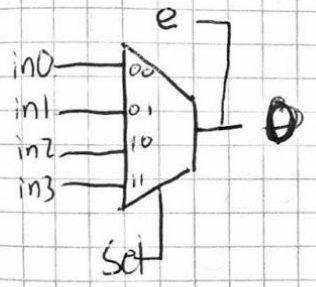
b[7:0] = sw[15] to sw[8]

~~btnU = sub~~

sub = btnU

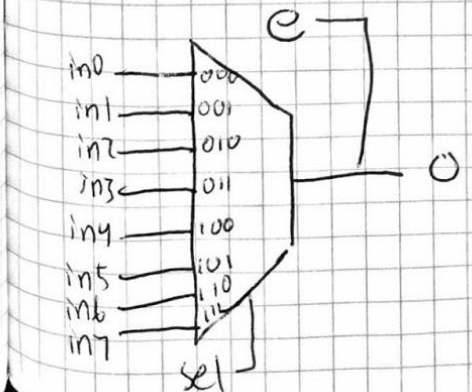
overflow = dp

m4_1e in: in[3:0], sel[1:0], e
out: o



sel1	sel0	in3	in2	in1	in0	e	o
0	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0
1	0	0	1	0	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	in0
0	1	0	0	1	0	1	in1
1	0	0	1	0	0	1	in2
1	1	1	0	0	0	1	in3

m8_1e in: in[7:0], sel[2:0], e
out: o



Lab 3
7/27

Truth table on next pg

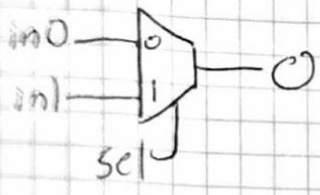
2:20 pm
20 Oct 2017

m8-1e truth table

s2	s1	s0	in7	in6	in5	in4	in3	in2	in1	in0	e	o
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	in0
0	0	1	0	0	0	0	0	0	1	0	1	in1
0	1	0	0	0	0	0	0	1	0	0	1	in2
0	1	1	0	0	0	0	1	0	0	0	1	in3
1	0	0	0	0	0	1	0	0	0	0	1	in4
1	0	1	0	0	1	0	0	0	0	0	1	in5
1	1	0	0	1	0	0	0	0	0	0	1	in6
1	1	1	1	0	0	0	0	0	0	0	1	in7

CE100 Lab 3

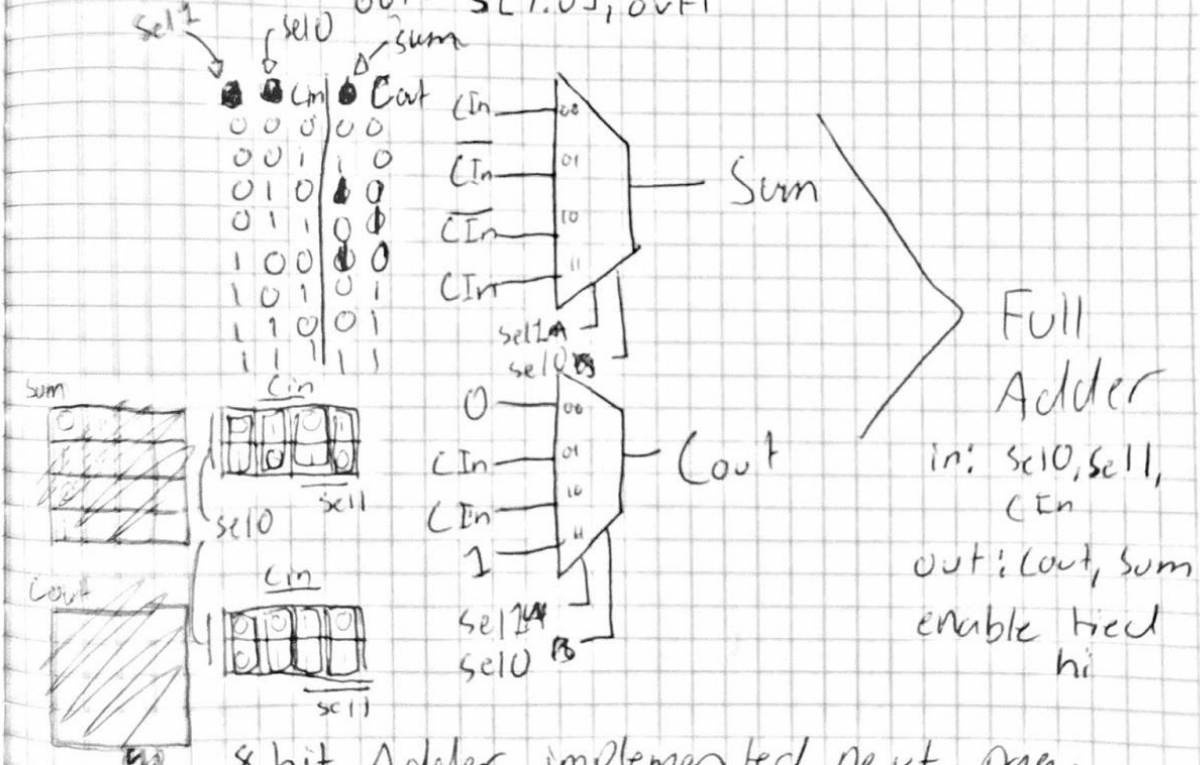
m2_1x8 in: in0[7:0], in1[7:0], sel,
out: o[7:0]



sign extend sel to be 8 bits

sel	in0	in1	o
0	1	0	0
1	0	1	0
0	0	0	in0
1	1	1	in1

AddSub8 in: A[7:0], B[7:0], sub
out: S[7:0], ovfl



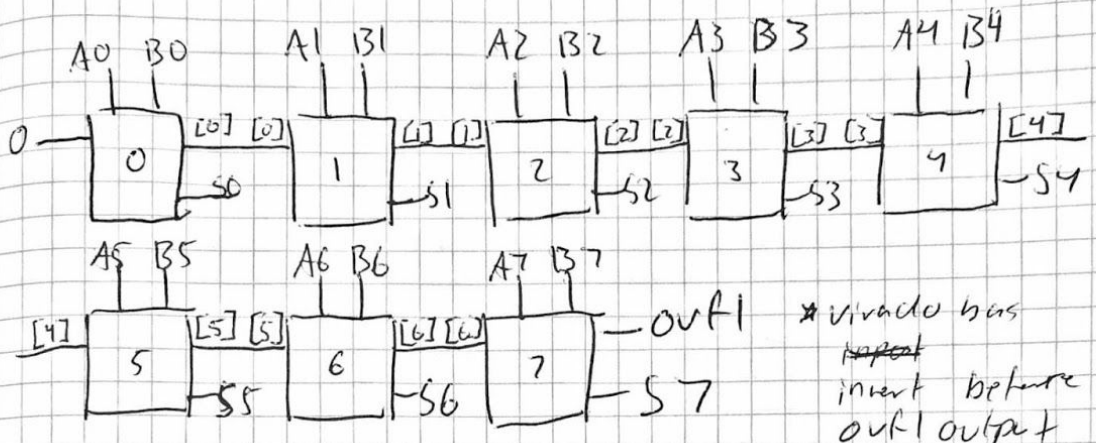
8 bit Adder implemented next page

Overflow logic

S[7]	A[7]	B[7]
1	0	0
0	1	1

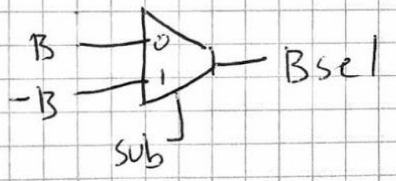
CE100 Lab 3

AddSub8 8bit adder B=Bsel



* vivado has ~~repeat~~ invert before ovfl output

AddSub8 m2_1x80



m2 hex7seg in: n[3:0], c
out: seg[6:0]

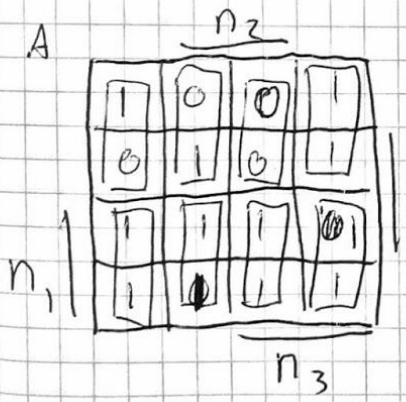
Continued Next Pg

CE100 Lab 3

hex7seg m&_le

#	n_3	n_2	n_1	n_0	A	B	C	D	E	F	G
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10	A	1	0	1	0	1	1	1	0	1	1
11	B	1	0	1	1	1	1	1	1	1	1
12	C	1	1	0	0	0	1	1	1	1	1
13	D	1	1	0	1	0	1	1	1	1	1
14	E	1	1	1	0	0	0	1	1	1	1
15	F	1	1	1	1	0	0	0	1	1	1

A



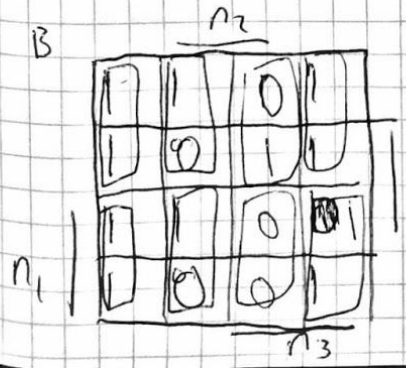
PIs

~n[0], 1, n[0], 1, 1, ~~n[a]~~, 1

~~n[0]~~, 1

0

B



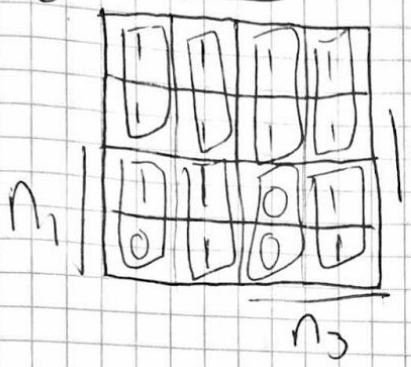
PIs

1, 1, ~n[0], n[0], 1, 1, n[0],

0

CE100 Lab 3

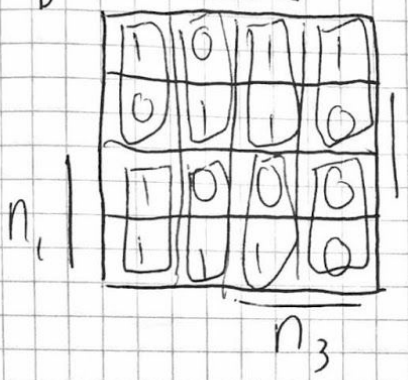
c



PIs

1, n[0], 1, 1, 1, 1, 1, 0

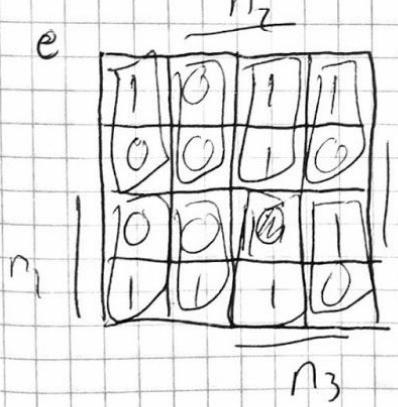
d



PIs

~n[0], 1, n[0], ~n[0], ~n[0],
0, 1, ~n[0]

e

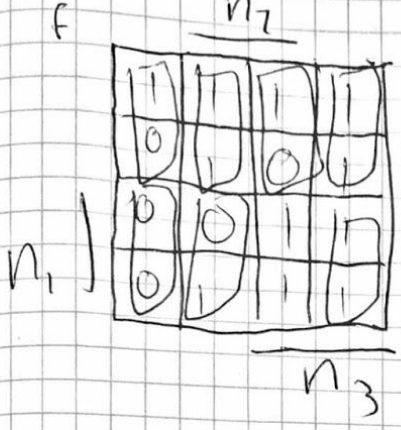


PIs

~n[0], ~n[0], 0, ~n[0], &
&, ~n[0], n[0], 1, 1

CE100 Lab 3

f

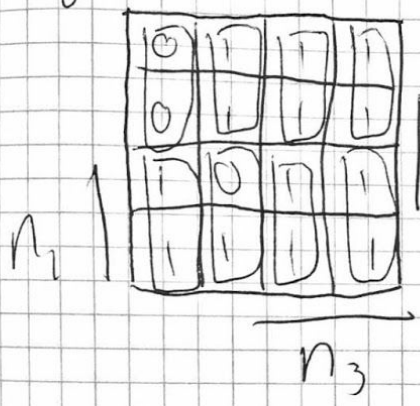


PIs

~n[0], 0, 1, ~n[0], 1, 1

~n[0], 1

g



PIs

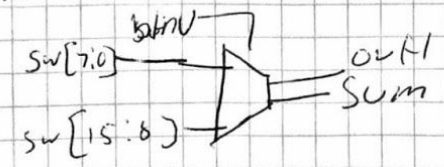
0, 1, 1, ~n[0], 1, 1, 1, 1

~~Top level schematic~~

Lab3Top in: [15:0]sw, btnU, btnR, clk_in

out: seg[6:0], dp, an[3:0]

AddSub8



Lab3CE100

hex7seg0/1

in n_3 n_2 n_1 n_0

sum3 2 1 0/7654

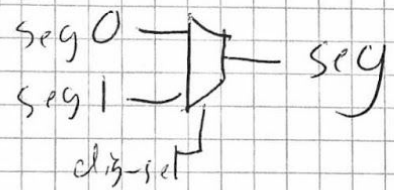
in c

dig_sel / dig_sel

out seg

seg0 / seg1

m2-1x8



Lab3Sim

CE100 Lab 3

Top Level Design