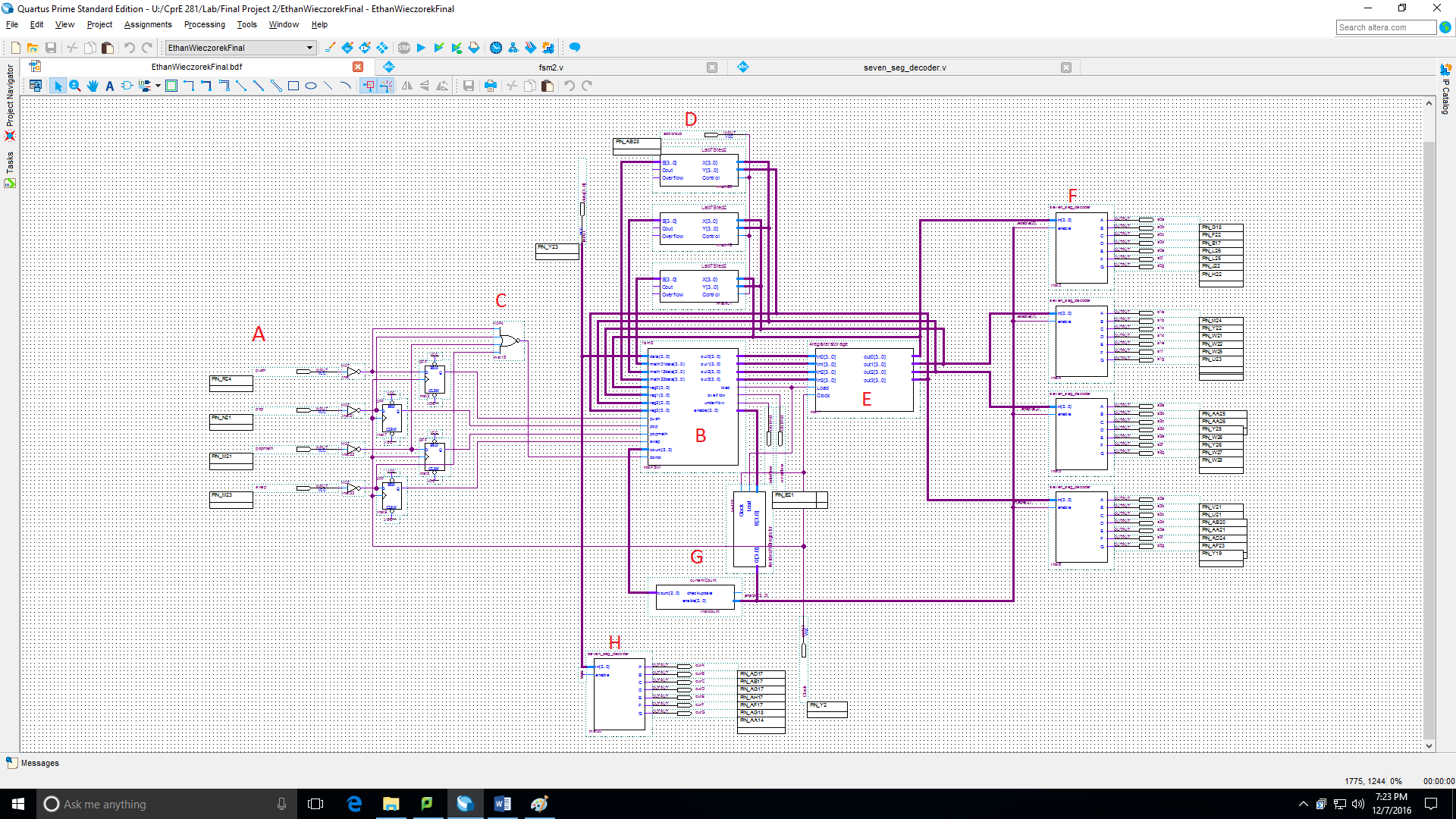
Ethan Wieczorek Final Project: Stack

Top level design entity



**A:** These four input lines are the pushbuttons. Top to bottom they are: push, pop, popmath, and swap. These are fed into a not gate because push buttons are 0 when being pressed. The not gate then feeds into a D flip flop. This is to help debounce the circuit and keep the one for the entire clock cycle. These four buttons feed into B.

**B:** This is the finite state machine. I commented all of my code for this with what I was doing so I’m just including the code.

**module fsm2(data, math01data, math12data, math23data, reg0, reg1, reg2, reg3, push, pop, popmath, swap, count, donot, out0, out1, out2, out3, load, overflow, underflow, enable);**

**input [3:0] data, math01data, math12data, math23data, reg0, reg1, reg2, reg3; //these lines are all of my input buses.**

**input [2:0] count; //this is my input for the current amount of items in the stack**

**input push, pop, popmath, swap, donot; //these are the push buttons. donot activates on the negative edge of the button presses.**

**output reg [3:0] out0;**

**output reg [3:0] out1;**

**output reg [3:0] out2;**

**output reg [3:0] out3;**

**//all output registers for parallel access register.**

**output reg load, overflow, underflow;**

**//load becomes 1 when you want to write to the access register**

**output reg [3:0] enable; //this is telling which seven segment displays to turn on.**

**reg en0, en1, en2, en3; //this is temporary storage for enable.**

**always @(push or pop or popmath or swap or donot) //the code underneath here will run whenever any of these hit a positive edge.**

**if(donot) //first of all, if donot is 1 (all buttons are 0) i want it to do nothing**

**begin**

**load <= 1'b0; //don't load the registers, in other words, do nothing**

**end**

**else if(push) //otherwise if push is one it will write to the first open register**

**begin**

**//the following cases are my FSM, each case represents what the "count" is currently at. The count represents how many items are currently stored.**

**case (count)**

**0:begin //0 items stored**

**out0 <= data; //data goes in first register**

**overflow <= 1'b0; //overflow and underflow are off**

**underflow <= 1'b0;**

**en0 <= 1'b1; //first seven segment display becomes enabled.**

**end**

**1:begin //1 item in the stack**

**out0 <= reg0; //first register stays the same**

**out1 <= data; //data goes in second register.**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en1 <= 1'b1; //second display is turned on**

**end**

**2:begin //2 items in the stack**

**out0 <= reg0; //first 2 registers stay the same**

**out1 <= reg1;**

**out2 <= data; //data goes in thrid register**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en2 <= 1; //third display turns on**

**end**

**3:begin //3 items in stack**

**out0 <= reg0; //first 3 registers all stay the same**

**out1 <= reg1;**

**out2 <= reg2;**

**out3 <= data; //data goes in the fourth one**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en3 <= 1'b1; //fourth display turned on**

**end**

**4, 5, 6, 7:begin //if the count is any higher it should set off overflow, and all registers should stay the same.**

**out0 <= reg0;**

**out1 <= reg1;**

**out2 <= reg2;**

**out3 <= reg3;**

**overflow <= 1'b1;**

**underflow <= 1'b0;**

**end**

**endcase**

**//all code above this has been blocked code. meaning that verilog will go through the code and store everything you want it to do until you reach the end statement.**

**//after the block ends the two lines below this will run unblocked**

**//it will enable the displays and write to the registers.**

**enable = {en3,en2,en1,en0};**

**load = 1'b1;**

**end**

**else if(pop) //if it wasn't a push we check to see if it was a pop**

**begin**

**case (count)**

**0:begin //if the count is 0, underflow**

**overflow <= 1'b0;**

**underflow <= 1'b1;**

**end //if the count is 1, display 1 gets turned off**

**1:begin**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en0 = 1'b0;**

**end**

**2:begin //if the count is 2, display 2 gets turned off, 1 stays the same**

**out0 <= reg0;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en1 = 1'b0;**

**end**

**3:begin //if the count is 3, display 3 gets turned off, 1 & 2 stay the same**

**out0 <= reg0;**

**out1 <= reg1;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en2 = 1'b0;**

**end**

**4:begin //if count is 4, display 4 turns off, 1-3 stay the same**

**out0 <= reg0;**

**out1 <= reg1;**

**out2 <= reg2;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en3 = 1'b0;**

**end**

**5, 6, 7:begin //these cases should never happen, they are all overflow**

**out0 <= reg0;**

**out1 <= reg1;**

**out2 <= reg2;**

**out3 <= reg3;**

**overflow <= 1'b1;**

**underflow <= 1'b0;**

**end**

**endcase**

**enable = {en3,en2,en1,en0};**

**load = 1'b1;**

**end**

**else if(popmath) //otherwise if popmath was pressed**

**begin**

**case (count)**

**0:begin //if count is 0, underflow**

**overflow <= 1'b0;**

**underflow <= 1'b1;**

**end**

**1:begin //if count is 1, underflow**

**out0 <= reg0;**

**overflow <= 1'b0;**

**underflow <= 1'b1;**

**end**

**2:begin //if count is 2, reg 1 becomes the math data from 1 and 2.**

**out0 <= math01data;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en1 = 1'b0;**

**end**

**3:begin //if count is 3, reg 2 becomes the math from 2 and 3.**

**out0 <= reg0;**

**out1 <= math12data;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en2 = 1'b0;**

**end**

**4:begin //if count is 4, reg 3 becomes the math between 3 and 4.**

**out0 <= reg0;**

**out1 <= reg1;**

**out2 <= math23data;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**en3 = 1'b0;**

**end**

**5, 6, 7:begin //these cases should never happen, they are all overflow**

**overflow <= 1;**

**end**

**endcase**

**enable = {en3,en2,en1,en0};**

**load = 1'b1; //load the blocked code**

**end**

**else if(swap) //if swap is 1, we are going to be swapping the top two items**

**begin //the size of the stack should never change from swapping numbers so there is no mention of enable in here**

**case (count)**

**0:begin //if count is 0, underflow**

**overflow <= 1'b0;**

**underflow <= 1'b1;**

**end**

**1:begin //if count is 1, underflow**

**out0 <= reg0;**

**overflow <= 1'b0;**

**underflow <= 1'b1;**

**end**

**2:begin //if count is 2, swap 1 and 2**

**out0 <= reg1;**

**out1 <= reg0;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**end**

**3:begin //if count is 3, swap 2 and 3**

**out0 <= reg0;**

**out1 <= reg2;**

**out2 <= reg1;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**end**

**4:begin //if count is 4, swap 3 and 4**

**out0 <= reg0;**

**out1 <= reg1;**

**out2 <= reg3;**

**out3 <= reg2;**

**overflow <= 1'b0;**

**underflow <= 1'b0;**

**end**

**5, 6, 7:begin //these cases should never happen, they are all overflow**

**overflow <= 1'b1;**

**end**

**endcase**

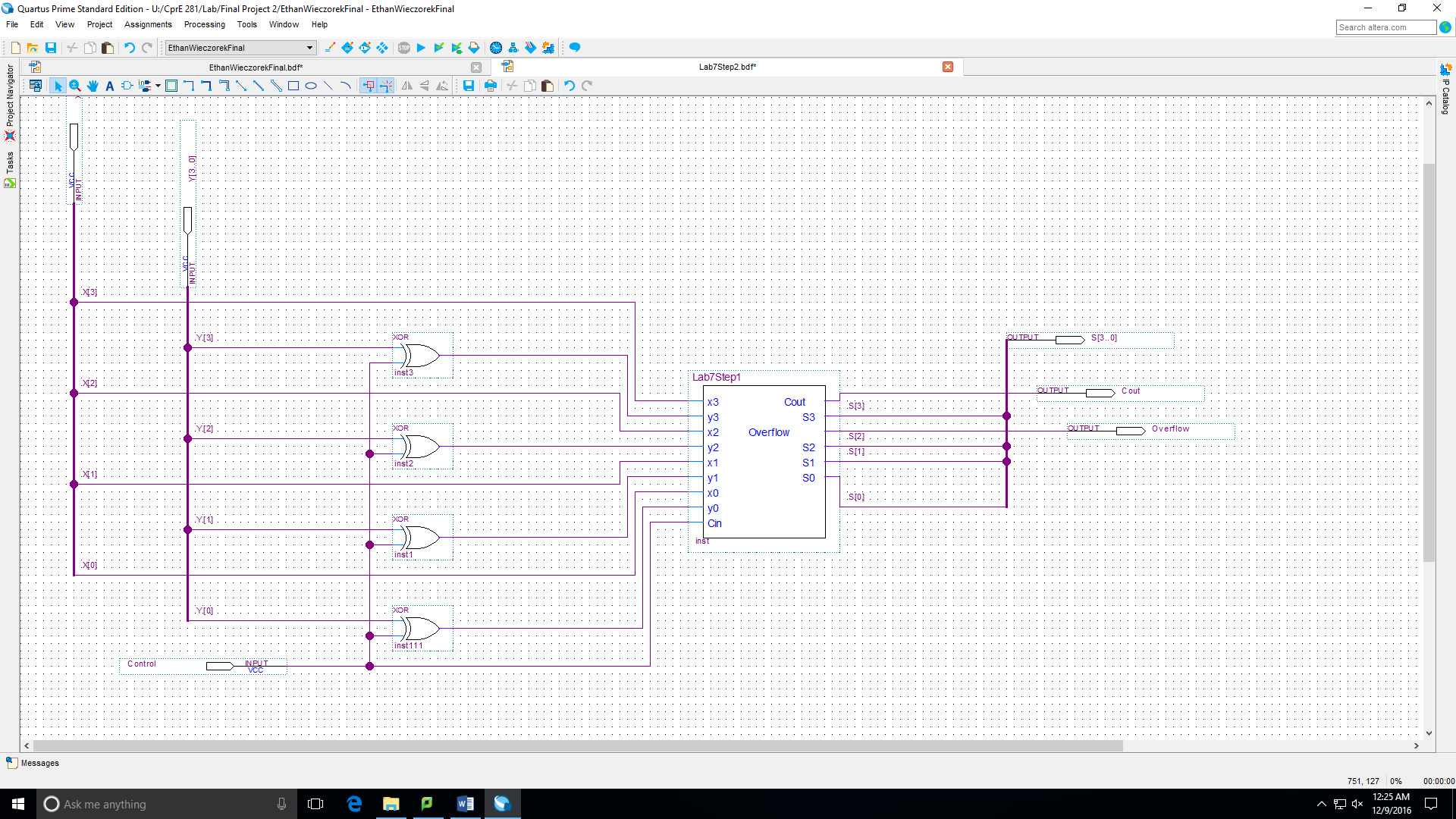
**load = 1'b1; //load the blocked code**

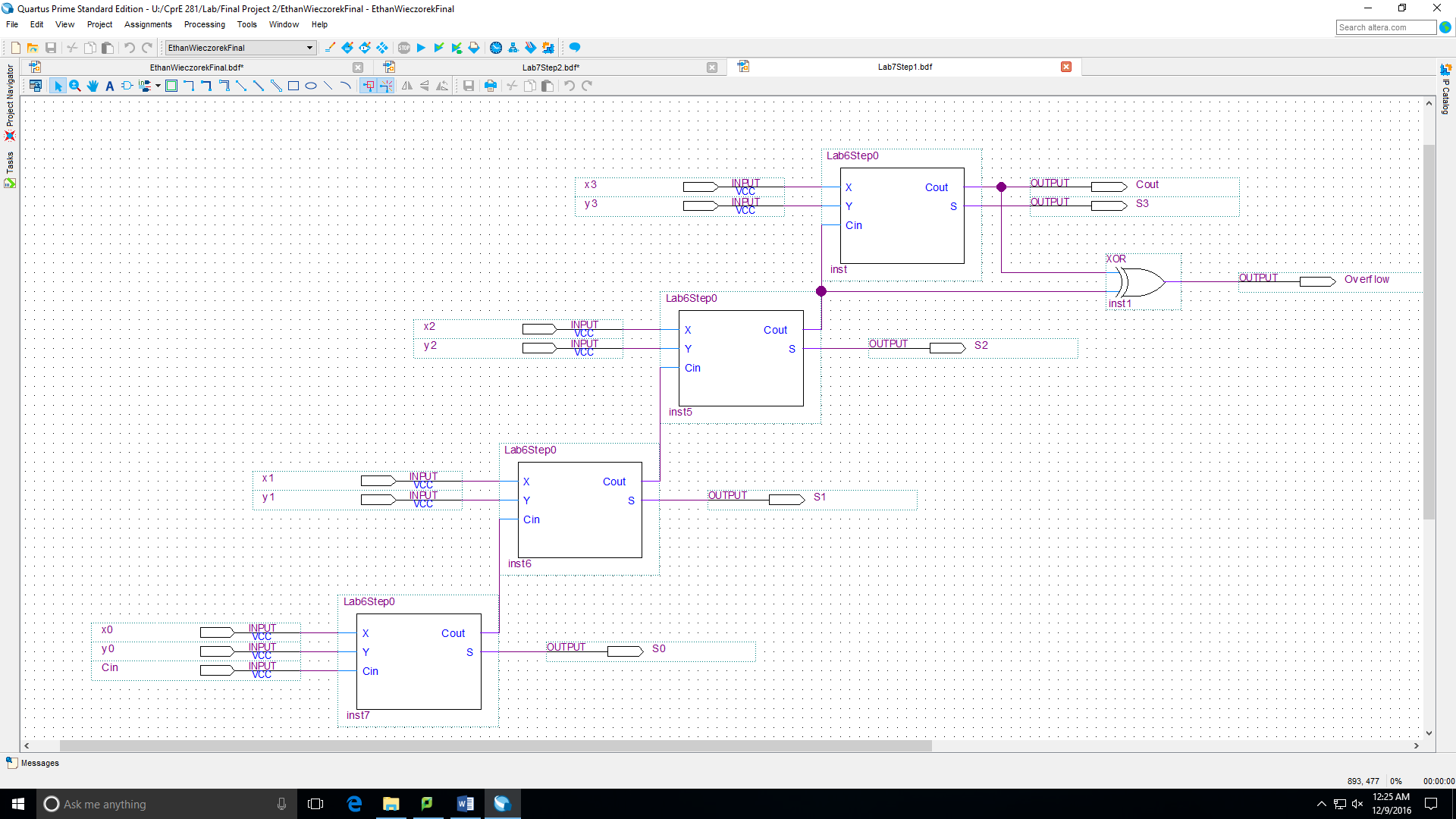
**end**

**endmodule**

**C:** This feeds into the donot in B.

**D:** These are the adder/subtracters from lab 7. When addorsub (mapped to switch 0) is 0, you add, when it is 1, you subtract.





module Lab6Step0(X,Y,Cin,Cout,S);

input Cin, X, Y;

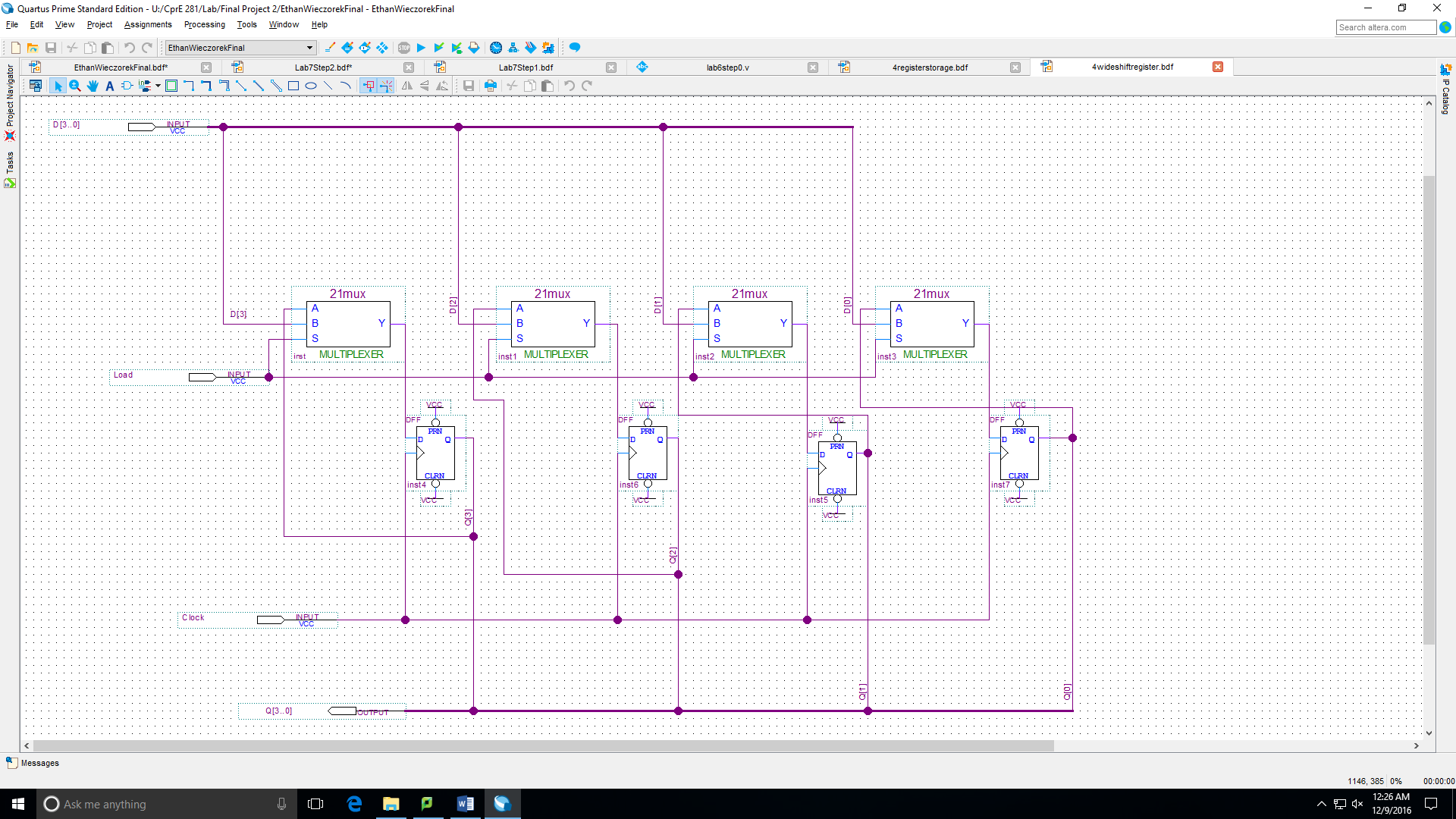
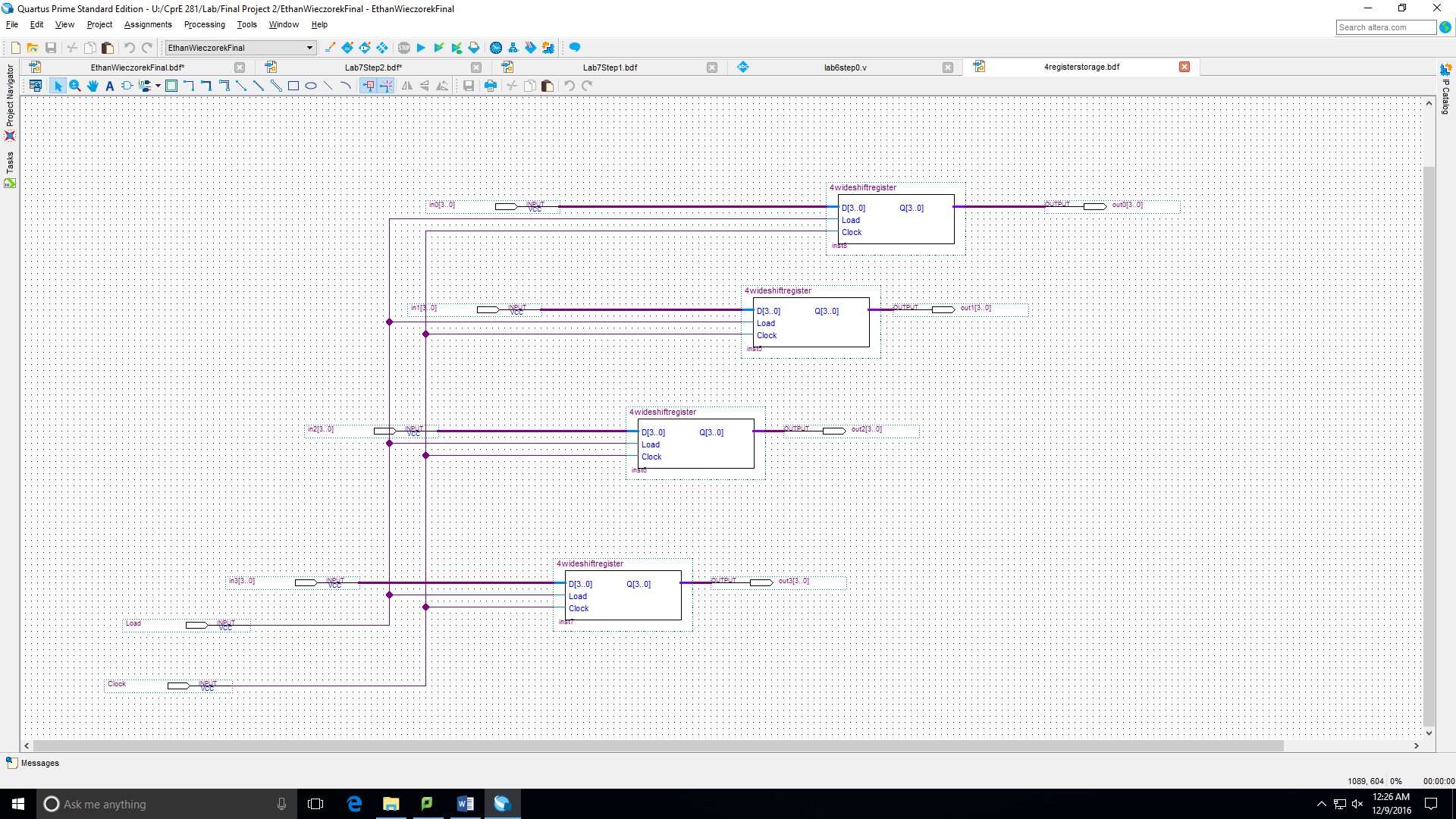
output Cout, S;

assign S = (~X&~Y&Cin)|(~X&Y&~Cin)|(X&~Y&~Cin)|(X&Y&Cin);

assign Cout = (Y&Cin)|(X&Y)|(X&Cin);

endmodule

**E:** This is 4 parallel access registers.



**F:** The four seven segment displays. They are only on when enable = 1, otherwise they display 7’b111111.

module seven\_seg\_decoder(in,A,B,C,D,E,F,G,enable);

input [3:0] in;

input enable;

output A,B,C,D,E,F,G;

reg [6:0] DISPLAY;

always @(in[3] or in[2] or in[1] or in[0])

begin

if(enable == 1'b1)

begin

case ({in[3],in[2],in[1],in[0]})

4'b0000: DISPLAY = 7'b0000001;

4'b0001: DISPLAY = 7'b1001111;

4'b0010: DISPLAY = 7'b0010010;

4'b0011: DISPLAY = 7'b0000110;

4'b0100: DISPLAY = 7'b1001100;

4'b0101: DISPLAY = 7'b0100100;

4'b0110: DISPLAY = 7'b0100000;

4'b0111: DISPLAY = 7'b0001111;

4'b1000: DISPLAY = 7'b0000000;

4'b1001: DISPLAY = 7'b0000100;

4'b1010: DISPLAY = 7'b0001000;

4'b1011: DISPLAY = 7'b1100000;

4'b1100: DISPLAY = 7'b0110001;

4'b1101: DISPLAY = 7'b1000010;

4'b1110: DISPLAY = 7'b0110000;

4'b1111: DISPLAY = 7'b0111000;

endcase

end

else

begin

DISPLAY = 7'b1111111;

end

end

assign {A,B,C,D,E,F,G} = DISPLAY;

endmodule

**G:** This turns the amount of enabled segments into a count for B to use.

There is also a register storing the data before it goes in to count.

module currentCount(checkupdate,enable,count);

input [3:0] enable;

input checkupdate;

output reg [2:0] count;

reg r0, r1, r2, r3;

always @(enable or checkupdate)

begin

case(enable)

4'b0000: count <= 3'b000;

4'b0001: count <= 3'b001;

4'b0010: count <= 3'b010;

4'b0011: count <= 3'b010;

4'b0100: count <= 3'b011;

4'b0101: count <= 3'b011;

4'b0110: count <= 3'b011;

4'b0111: count <= 3'b011;

4'b1000: count <= 3'b100;

4'b1001: count <= 3'b100;

4'b1010: count <= 3'b100;

4'b1011: count <= 3'b100;

4'b1100: count <= 3'b100;

4'b1101: count <= 3'b100;

4'b1110: count <= 3'b100;

4'b1111: count <= 3'b100;

endcase

end

endmodule

**H:**

This is just another seven segment display. This just shows whatever your data currently is so you know what you are about to push.

**Test Cases:**

**Test push each number** (some numbers weren’t adding to count at first, especially when I pushed 0, so I used this to make sure every number works in every spot.

Push 0 4 times. popadd, then popsubtract, then pop twice.

Push 1 4 times. popadd, then popsubtract, then pop twice.

Push 2 4 times. popadd, then popsubtract, then pop twice.

…….

Push F 4 times. popadd, then popsubtract, then pop twice.

**Current year with math:**

Push 6, then 1, then 0, then 2.

Display is 2016.

Popadd shows 216.

Popadd shows 36.

Popsubtract shows 3.

Pop shows nothing.