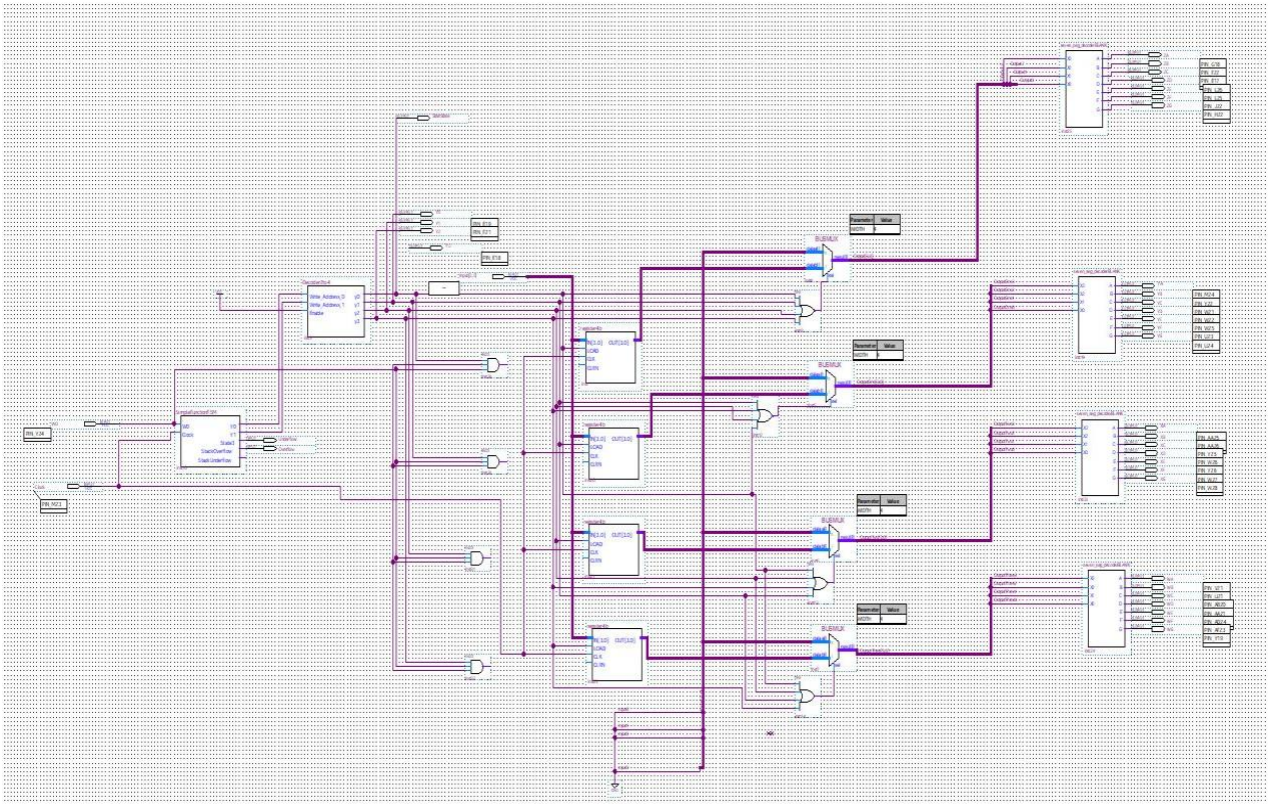
Anshul Bharath
Section 16
December 10, 2019

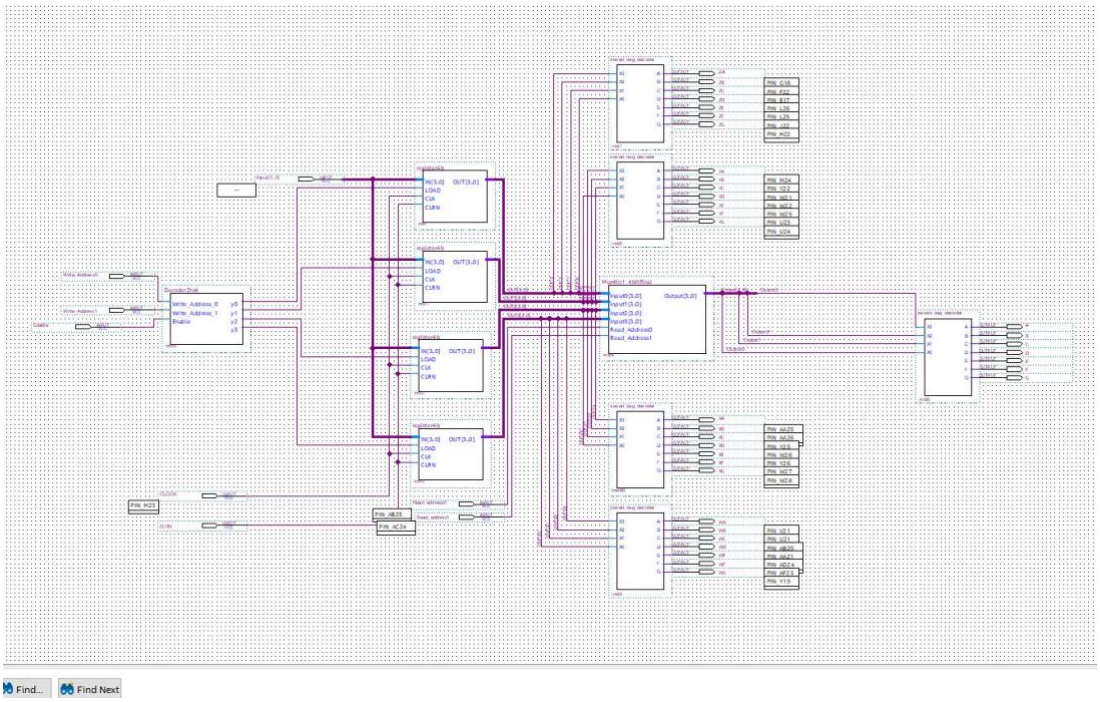
Final Project Report

For the final report I attempted to implement option three which was the Stack arithmetic. This lab report consists of the calculations, circuits and workings that arrived at the final circuit that I had to demo. In this project, the goal was to show our basic understanding of digital logic. I thought I was going to be able to finish this project, but I went into it a little confident, running into many problems along the way. My first goal was to get a working register file. The second goal was to create the finite state machine that would control which function the circuit would run at the given time. The third goal was to add some advanced pop functions. The final goal was to put it all together and include error handling. This report will include a basic overview of my work on the project including all circuits, modules and Verilog code that was used. This will be accompanied by any calculations done to arrive at certain circuits.

Top Level Diagram

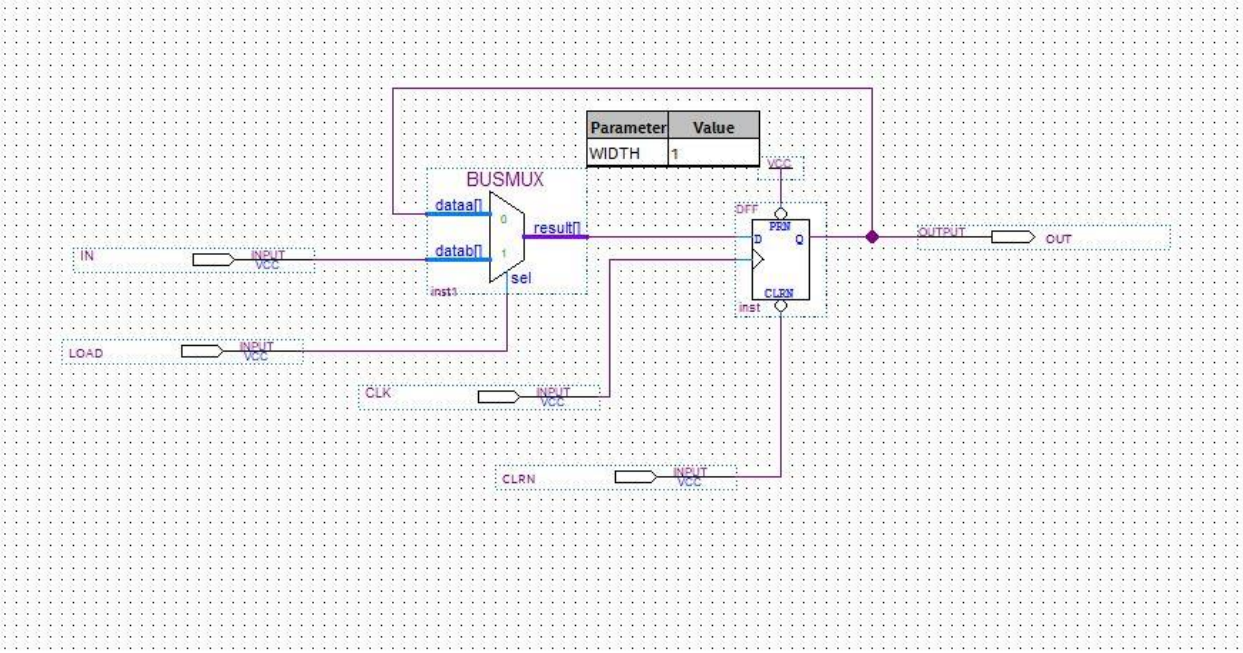


This is a top level diagram of the circuit that I have. I will split the diagram into three parts. Part A will be an explanation of the register file and how it works. Part B will be an explanation of the Finite State Machine that I used and the calculations used to arrive at the solution. Finally, Part C will be an explanation of my attempt at putting it together and my ideas of what I would have done if I had finished.

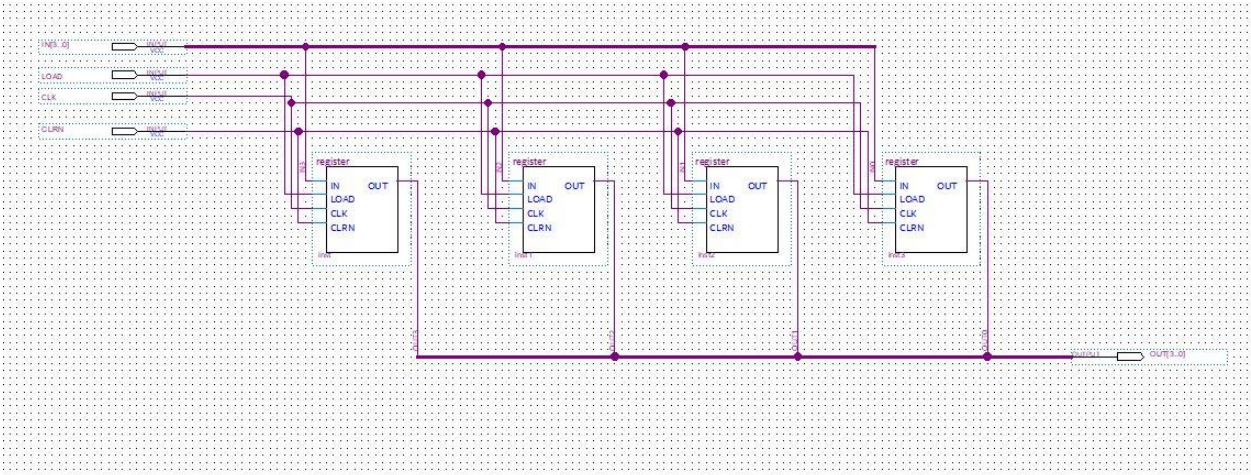


Part A: Register File

Part A will be an explanation of the register file that I used to store my values. The first image is the register file by itself. This circuit was used to test the register file to see if it was working up to the standards of a normal register. The modification I made to this register file in the end product was a change in multiplexing. In the test register file, I had one 4 to 1 multiplexer that only output one value. This would be ideal when I had to output all four of the registers at the same time. This was helpful to see how the register file worked although.

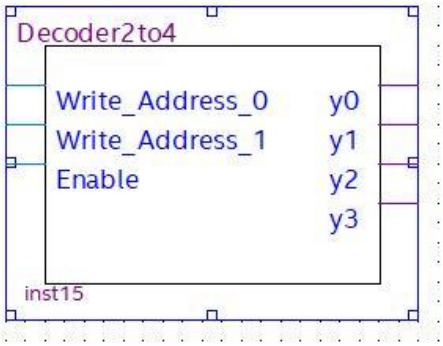


At the lowest level, the register file consisted of a 1-bit register that was able to hold 1-bit of information for any given clock cycle if the user was able to load the data in. This was the first Step in creating the register file.



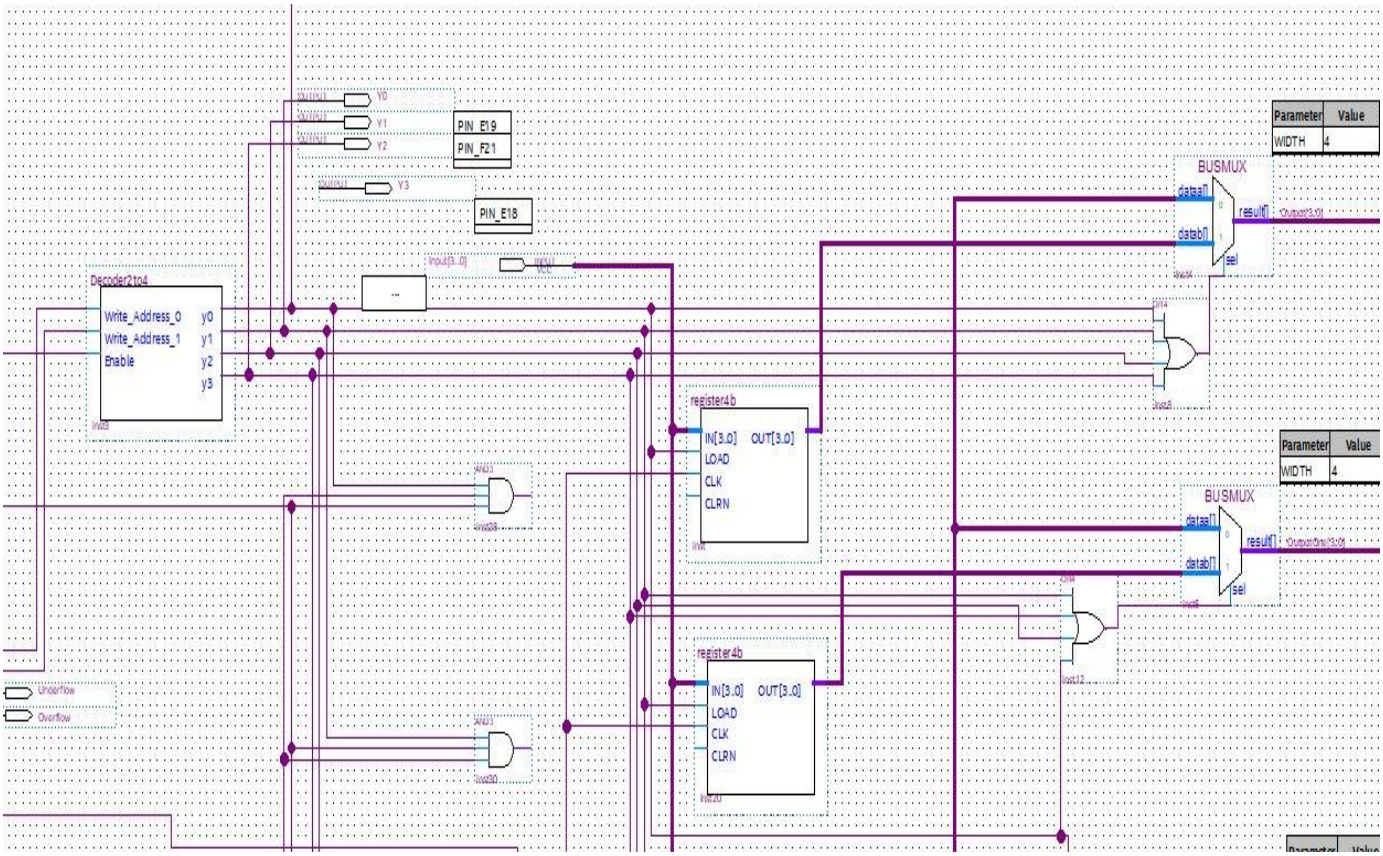
The next step up in the register file would be to put four of these registers together to create a 4-bit register. This is a parallel access 4-bit register where the user can directly

load the data in using a 4-bit wide bus. The other inputs include a Clock, ClearN and Load functionality.

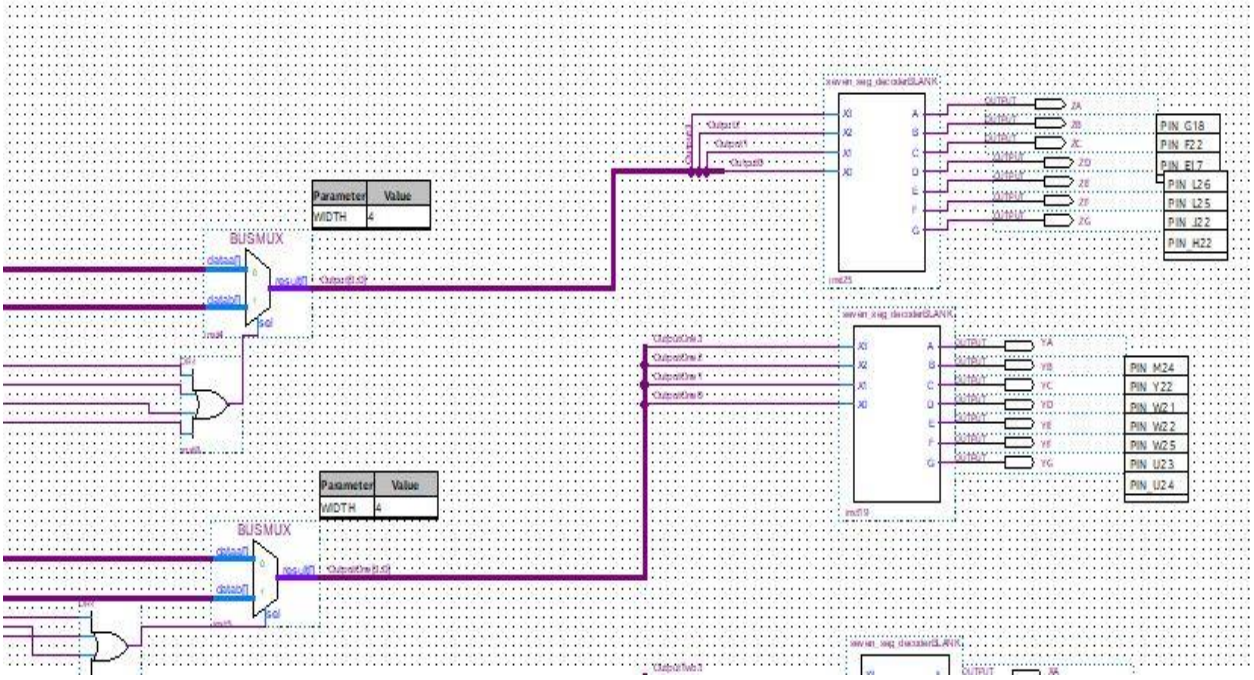


Putting four of the 4-bit registers together and a couple of other components and you arrive with the register file. The first step in the register file is a 2 to 4 decoder (this decoder will later be used to decode the output of the FSM into the register file). The decoder chooses which register to load into. There is a 4-bit wide bus that take the user input and feeds to all four of the 4-bit registers. Finally, to get the output of the registers multiplexers are used to show the output of the registers. Four multiplexers are used so that each register can have its own 7-segment display. The tricky part with the multiplexers, was to get the output to only show when there was something in the stack. My attempt at solving this was to change the *dataa* input of the multiplexer to a 4-bit ground bus and modify the 7-segment decoder to my needs. I also had to connect the

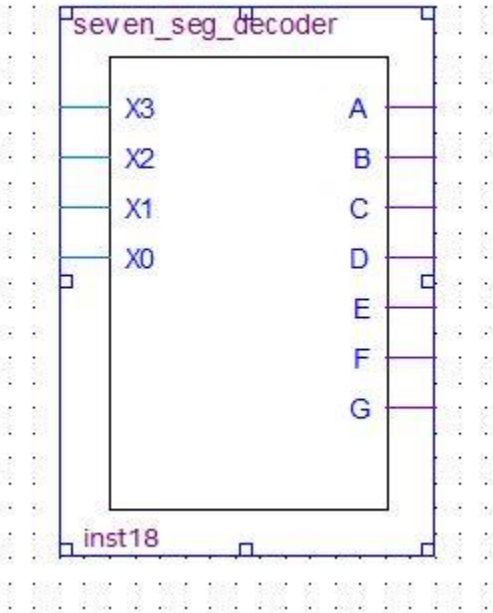
select lines of each multiplexer to the states that the FSM was in using four different or gates.



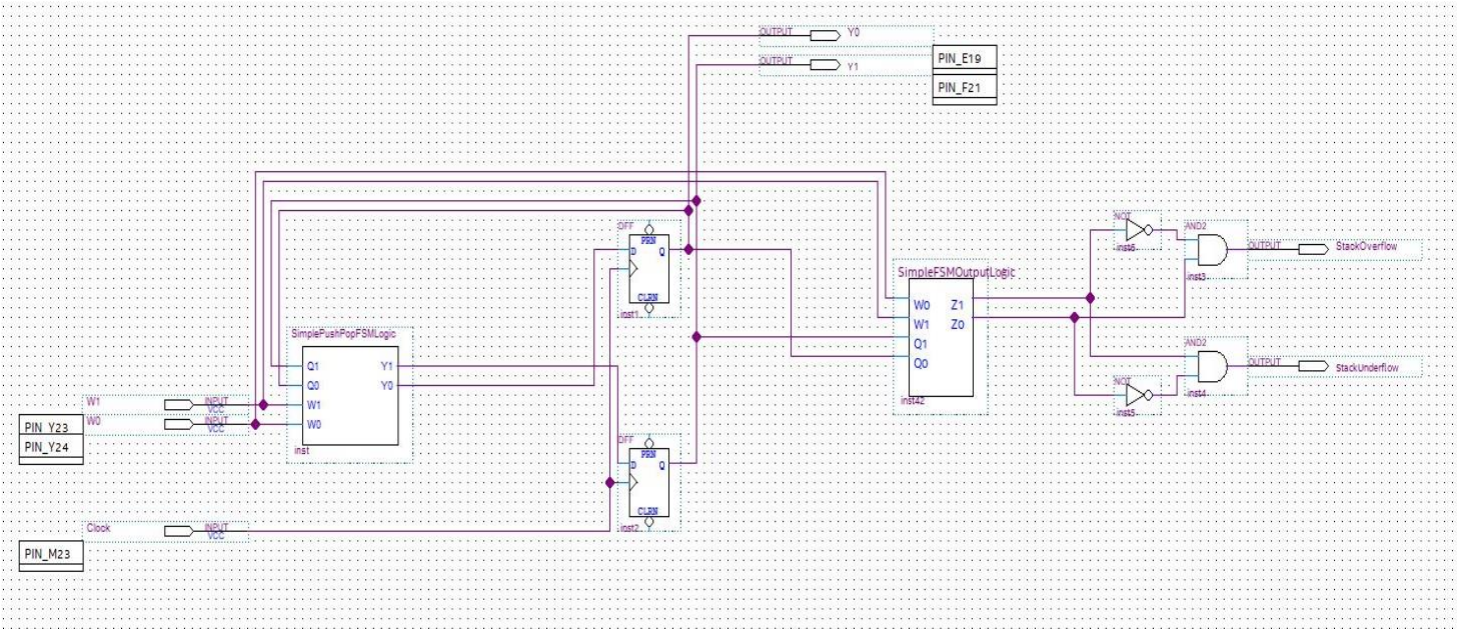
This image shows the first two registers of the register file. The inputs consist of a 4-bit *input*, *Load*, *CLK* and *CLRN*. The input is the user entered 4-bit number. This number is entered through the toggle switches on the FPGA board. The Load input is fed in through the decoder, this decoder is run by the finite state machine and rotates through the different states.



The 4-bit output from the registers then feed into the 4-bit 2 to 1 multiplexers here. This multiplexers then feed into the Seven segment display.

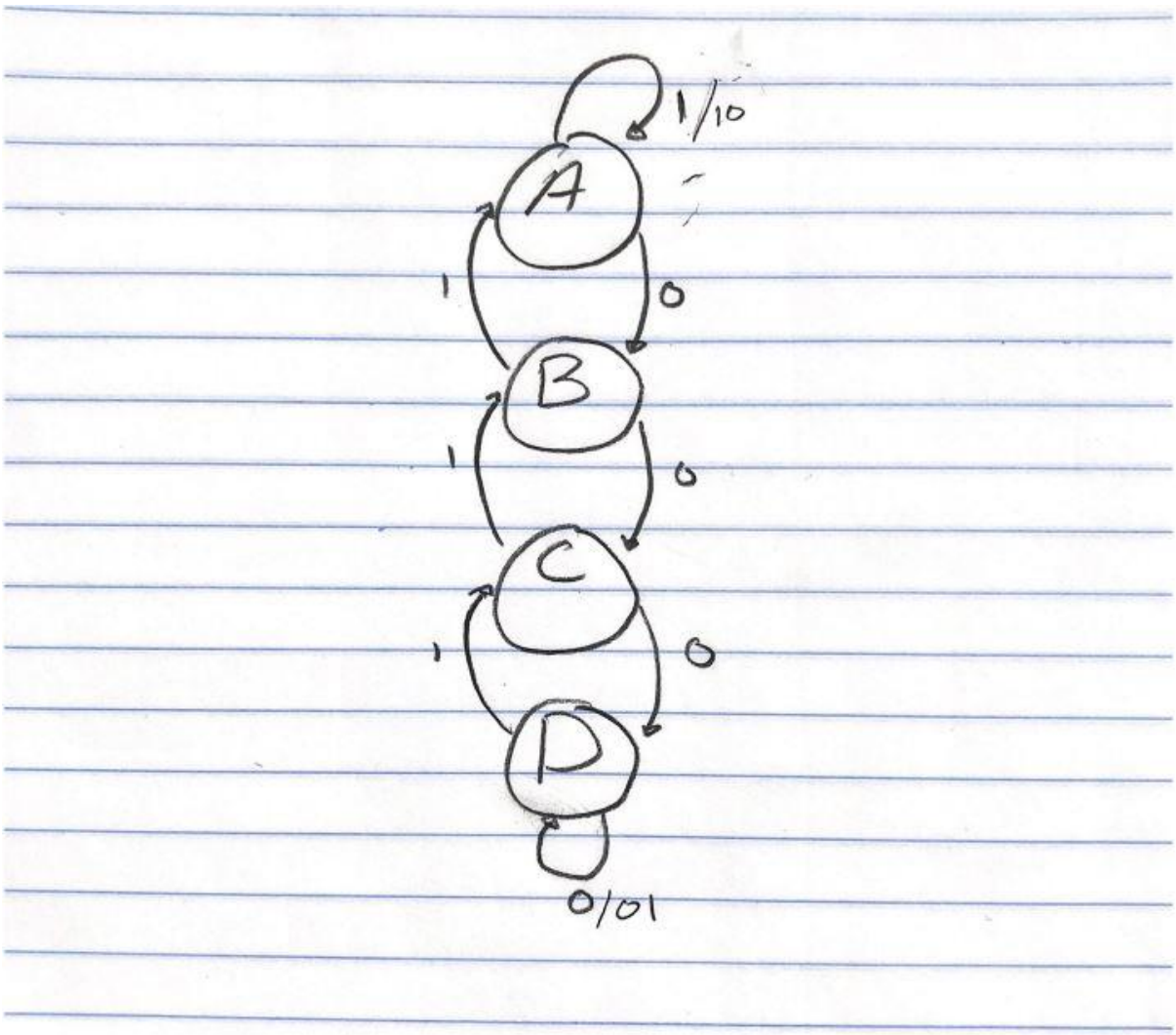


Dataa has been modified to go into the seven segment decoder and act as a null place holder that does not output anything when the select line is 0 on any of the multiplexers.

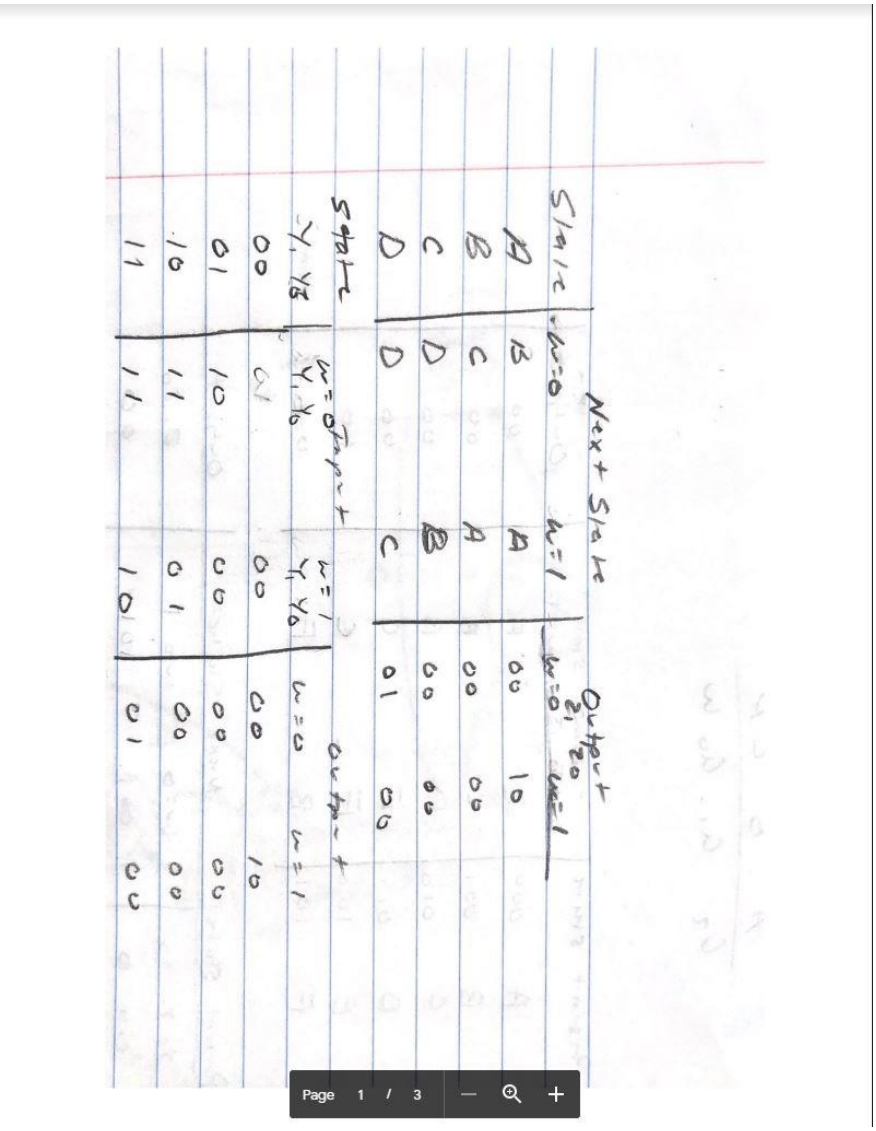


Part B: Finite State Machine

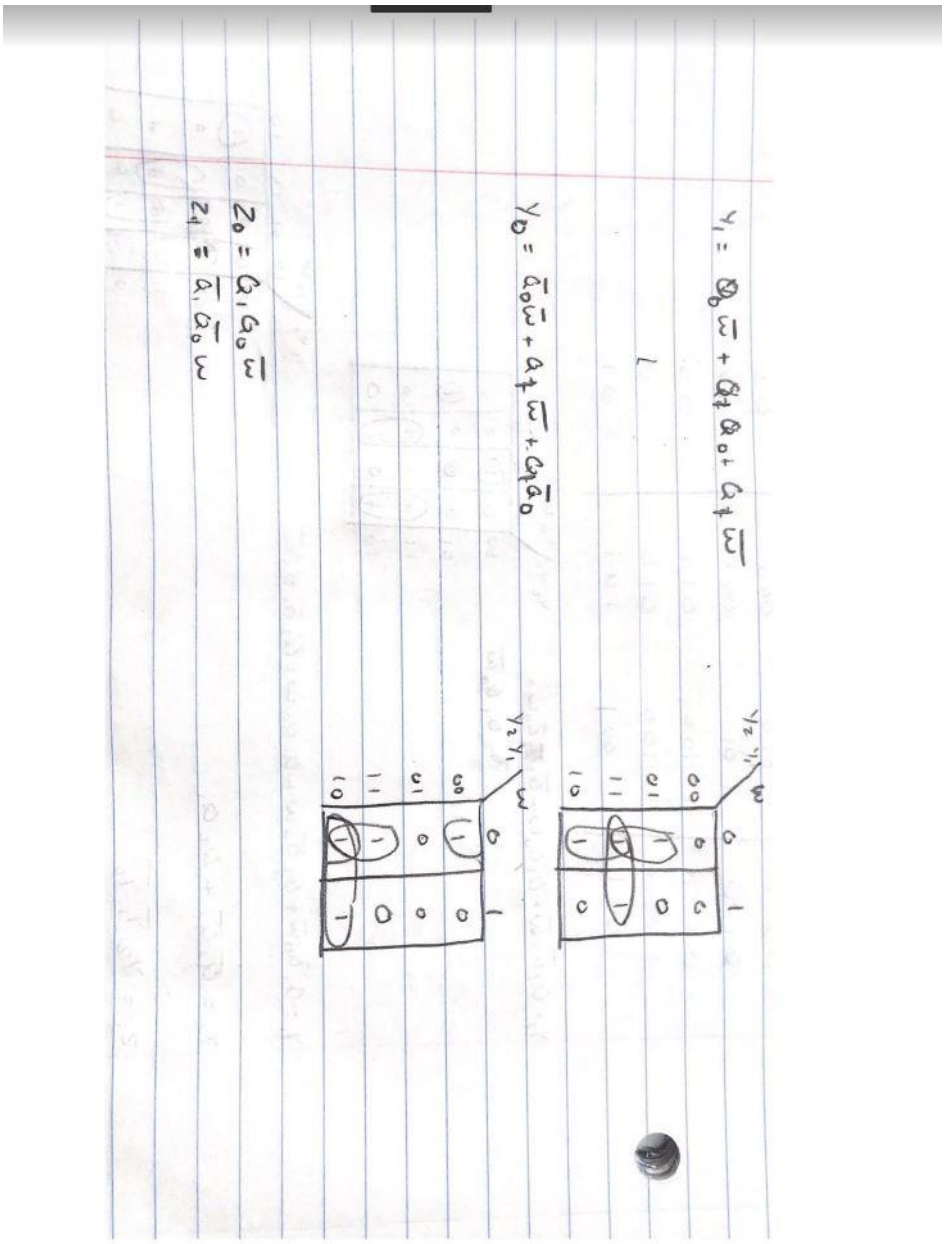
Here is the finite state machine that I used for the operations of push and pop. This finite state machine was able to handle both the push and pop operation with overload and underload detection. When developing this FSM, I made more than 10 different finite state machines all that did not work properly or were close to working, but not right. I ended up simplifying it and just giving the machine two inputs the push and pop functions that I set to 0 and 1. I decided to use a mealy machine that would output a 2- bit number. This number would output 10 if there overflow and 01 if there was underflow.



This is the State Diagram that I used to base the finite machine off of. It has four states that the machine cycles through. Each state corresponds to the element that can be added during that state, so because there are four elements that can be added, there are four states.



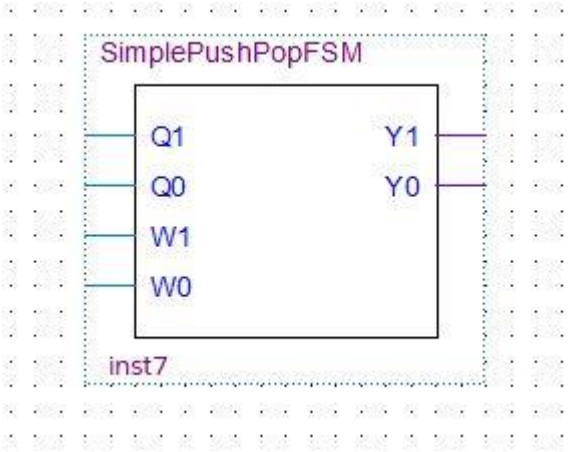
Next I transferred this data from the state diagram to the state table and state assignment table. Looking back at this, I should have organized this better to make my K-maps easier to fill in.

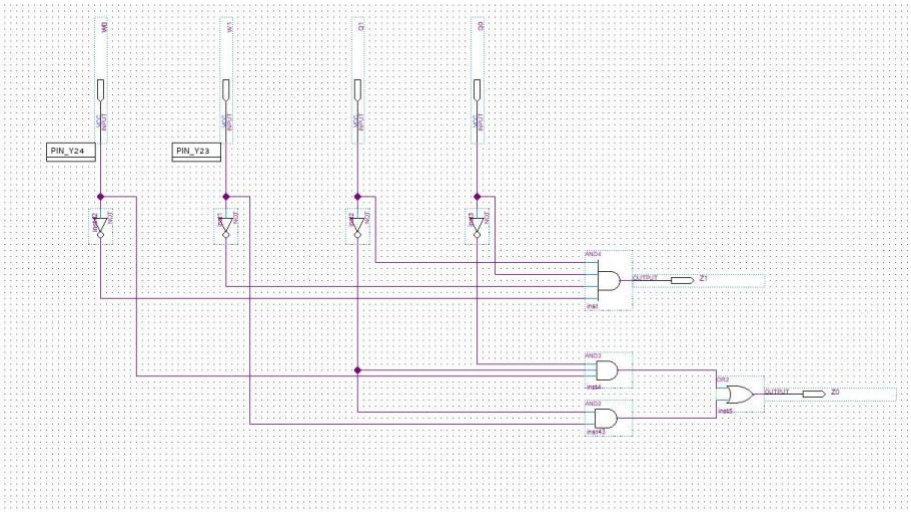


From there I calculated the functions needed for each of the state variables and the output variables, because it is a mealy machine. This was not too bad for this machine, but was probably the area I had the most trouble with the previous machines that I had created.

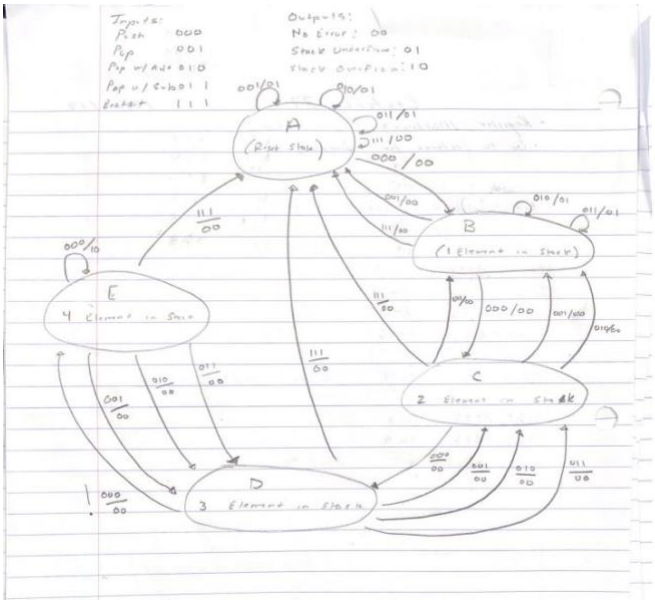
```
1   module SimplePushPopFSMLogic(Y1,Y0,Q1,Q0,W1,W0);
2
3      input Q1,Q0,W1,W0;
4      output Y1,Y0;
5
6      //~Q2,~Q1,~Q0,~W1,~W0
7
8      //assign Y2 = ((Q1&Q0&~W1)|(Q2&~Q1&~W1)|(Q2&Q1&~W0)|(~Q2&~Q1&~Q0&~W0));
9      //assign Y1 = ((Q2&~W0)|(Q1&Q0&~W0)|(Q1&Q0&~W0)|(~Q2&~Q1&Q0&~W1)|(~Q2&Q1&~Q0&~W1));
10     //assign Y0 = ((~Q0&~W1)|(Q2&~W1)|(Q2&~Q1&~W0)|(~Q2&Q1&~Q0&W0));
11
12     assign Y1 = ((Q1&Q0)|(Q0&~W1&~W0)|(Q1&~W1&W0));
13     assign Y0 = ((Q1&~W1)|(~Q0&~W1&~W0)|(Q1&Q0&~W0)|(Q1&~Q0&W0));
14
15  endmodule
16
17
```

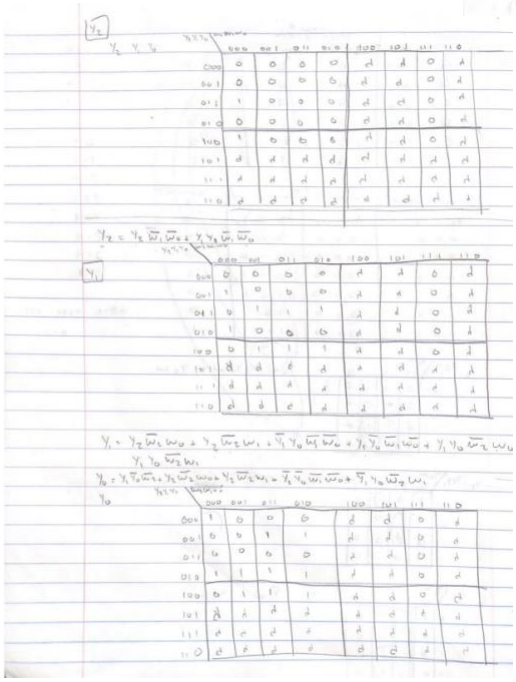
Here is the Verilog for the logic that ran the finite state machine. This was taken straight from the k-maps and were just two assignment statements.





I then made this into a symbol file and connected it to two D flip flops that the output would come back into the module. The second image is the output logic for the finite state machine that handled the errors that came out of the finite state machine. This output with additional gates were used as two different outputs.





The first finite state machine that I made had a 3-bit input 5 states and a two bit output. This made the k-maps very complex with 6 variables and the finite state machine did not work properly. I believe that I was on the right track with this finite state machine, but overcomplicated and confused myself.

Part C: Problems and Other Ideas

When starting this project, I first started with the register file. I was able to finish the register file with no issues and had it working within the first few hours of starting the project. The first issue that I faced was conceptualizing how I would use this register file as a storage element along with my finite state machine. The first problem that I had when attempting to solve this project was that I couldn't really wrap my head around how the states worked and what I should use as my states. The first idea I had was to use each of the functions of the Stack as my states and have an input of which state I wanted to go to. After tinkering around with this idea, it made no sense. I also did not think of the errors as outputs at the time. I came around and realized that the states would have to be how many elements would be in the stack at that moment, the inputs would be the functions of the stack and the output would be the errors that occurred at unique times.

I spent way to long figuring all four of the functions at the same time and kept coming up with wrong outputs and the finite state machine not working properly. I decided that I was just going to have a two-function finite state machine with no reset. I was able to successfully do this, but then ran into problems with the seven-segment display and the read lines of the multiplexers that I used. When I got this part down, I was able to load a value, but all the displays would light up at the same time because the load was enabled for all of the registers. I had to fix this by connecting the finite state machine to the registers' load input. My next issue was having the hex display does not display anything when there was nothing on the stack. This was done in a sneaky way, but I should have done it differently. What I did was turn the 0 input as a null value, so I am not able to

push any zeros onto the stack. What I should have done is modify the seven-segment decoder to have an enable function and added logic to that logic to allow it to show or not.

In conclusion, this was quite the comprehensive project. I put a lot of time into it that was not well used and not productive toward the end goal. Even with these setbacks though, I feel like I learned a lot and solidified the concepts that I have learned in this class. Given a couple more days, I could have finished a much more complete project. This project did teach me to work smarter and not harder.