Jonathon Schnell
EE 281
12/10/19
Section 16

Final Project Report

Project #1: Circuit for checking if a list of numbers is sorted

Part A: Register File

The purpose of a register file is to store data that can be written to and read from by a finite state machine. In this case the register file contains eight 4-bit registers meaning we can store eight 4-bit numbers. These registers can be parallely written to and read from. The register file has 2 parallel read read ports so that we can compare two numbers in one clock cycle.

The center 4x8 array of D flip-flops (DFF) is where the actual bit data is stored. (figure A3). The rows of DFFs make up a 4-bit unsigned integer *(A3, A2, A1, A0)*. Hexadecimal will be used to represent the value stored in each address (0 - F). the columns of DFFs represent the address of each number in the register file.

At the top of the register (figure A2) file there are eight 3 to 8 multiplexers, each multiplexer is responsible for reading one bit from a specified register. Each set for four multiplexers has a 3 bit parallel select line, this is used to determine which register is being read from. The output from each individual multiplexer is the data stored in the selected D flip-flop. This allows the FSM to read two numbers from the register file in one clock cycle.

On the left of the register (figure A3) file there is one 3x8 decoder this decoder is used to select where the input data, coming from the parallel load line at the bottom, is to be stored. Finally each output of the decoder is ANDed and wired to the CLK of each

register. This allows the data that is presented on the data load line to be written to the correct location when write enable is pushed. Initially the load functionality was a part of the register file. This feature was moved to the decoder by implementing an enable line.

Figure A4 shows a 3x8 multiplexer. The multiplexer allows a 3 bit number to select which of the inputs gets passed to the output. The rightmost part of this block diagram represents a 1x2 multiplexer which similarly allows a 1 bit input to select between two 1-bit inputs. The 1x2 multiplexer is a simple design using 2 AND gates 1 NOT gate and 1 OR gate. If the select line is 0 the and gate with a not infort of it is selected to be passed to the output. Consider the truth table for a 1x2 mux:

select	data1	data2	output
0	D1	D2	D1
1	D1	D2	D2

Multiplexers can easily be stacked to create larger multiplexers as shown in figure A4.

3x8mux select	D0	D1	D2	D3	D4	D5	D6	D7	out
000	D0	D1	D2	D3	D4	D5	D6	D7	D0
001	D0	D1	D2	D3	D4	D5	D6	D7	D1
010	D0	D1	D2	D3	D4	D5	D6	D7	D2
011	D0	D1	D2	D3	D4	D5	D6	D7	D3
100	D0	D1	D2	D3	D4	D5	D6	D7	D4
101	D0	D1	D2	D3	D4	D5	D6	D7	D5
110	D0	D1	D2	D3	D4	D5	D6	D7	D6
111	D0	D1	D2	D3	D4	D5	D6	D7	D7

Figure A5 contains the verilog code for a 3x8 one hot decoder which is essentially the opposite of a multiplexer. The decoder is passed a 3 bit value which is decoded to activate 1 output of the decoder. Consider its truth table for a 3x8 decoder with enable:

EN	W2	W1	W0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	**1**
1	0	0	1	0	0	0	0	0	0	**1**	0
1	0	1	0	0	0	0	0	0	**1**	0	0
1	0	1	1	0	0	0	0	**1**	0	0	0
1	1	0	0	0	0	0	**1**	0	0	0	0
1	1	0	1	0	0	**1**	0	0	0	0	0
1	1	1	0	0	**1**	0	0	0	0	0	0
1	1	1	1	**1**	0	0	0	0	0	0	0

Figure A1. Register File.

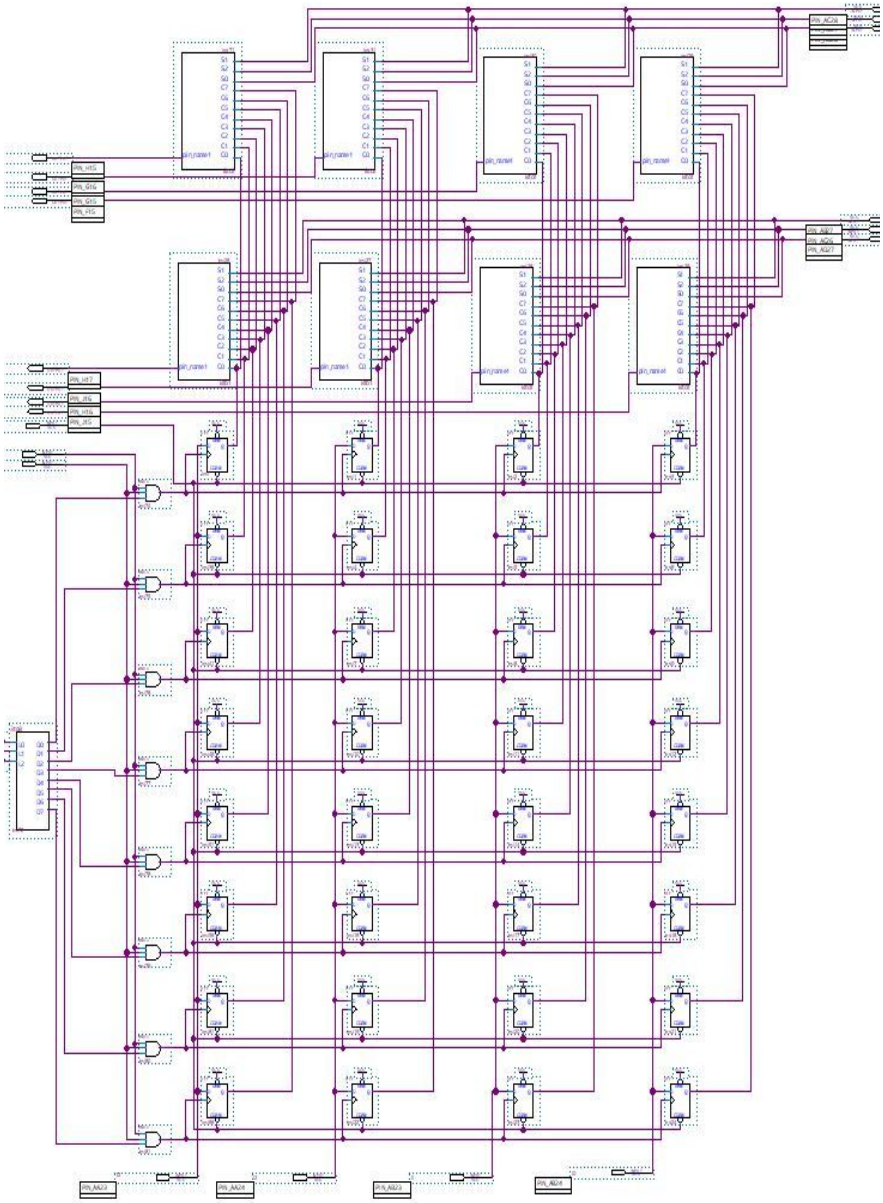


Figure A2. Multiplexer Read Banks.

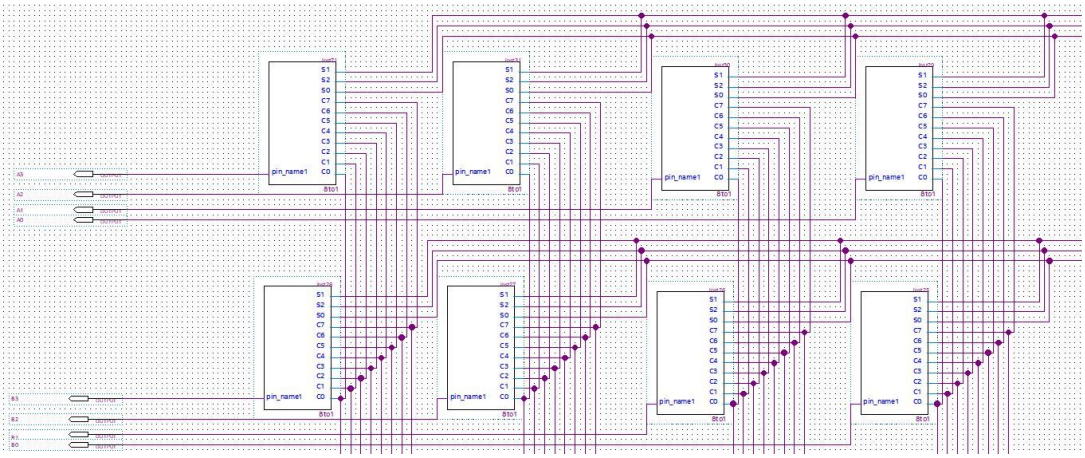


Figure A3. DFF's and Decoder

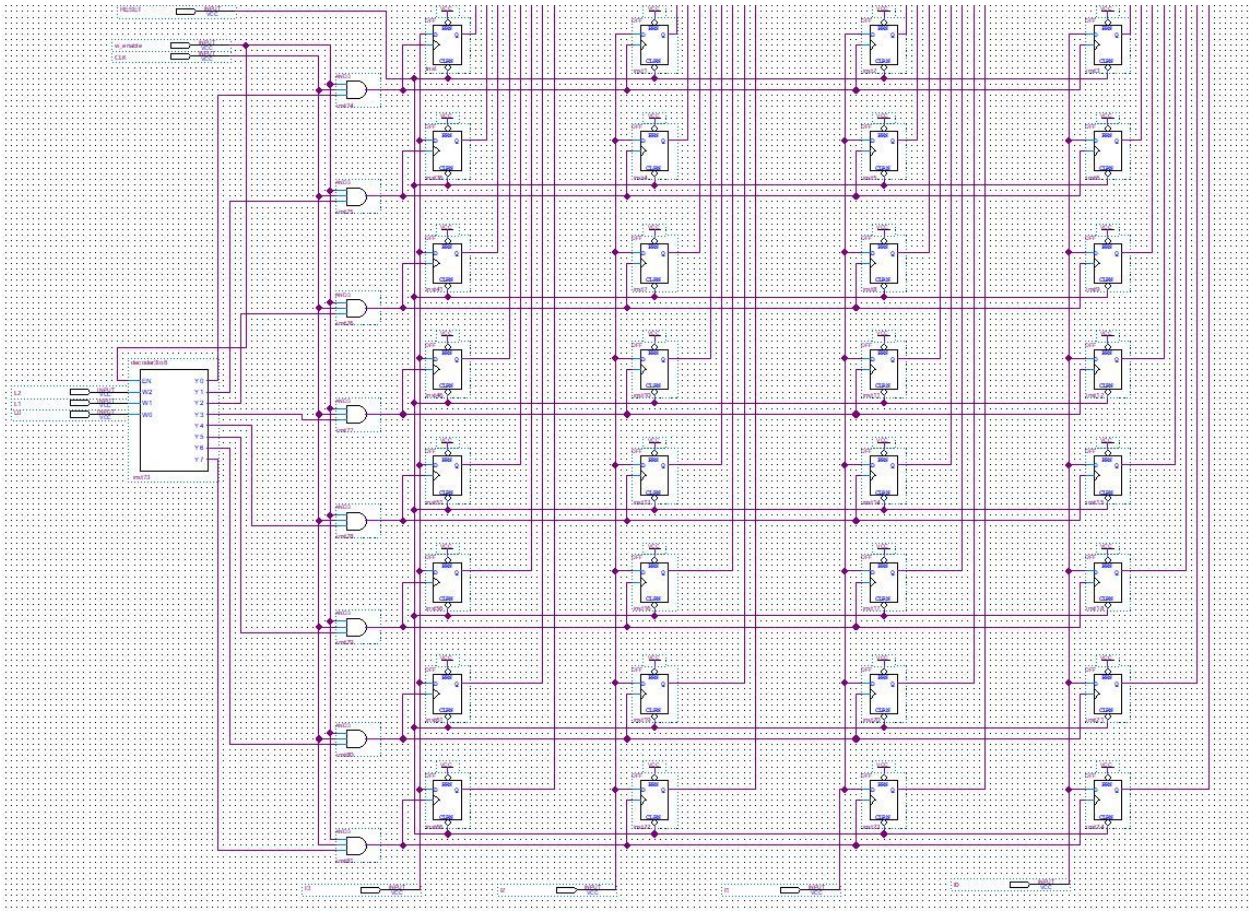


Figure A4. 3x8 MUX.
(NOTE THE INPUTS TO THE LEFT OF THIS MULTIPLEXER ARE IN REVERSE AND WERE CORRECTED IN THE FINAL DESIGN C7=C0 C0=C7)

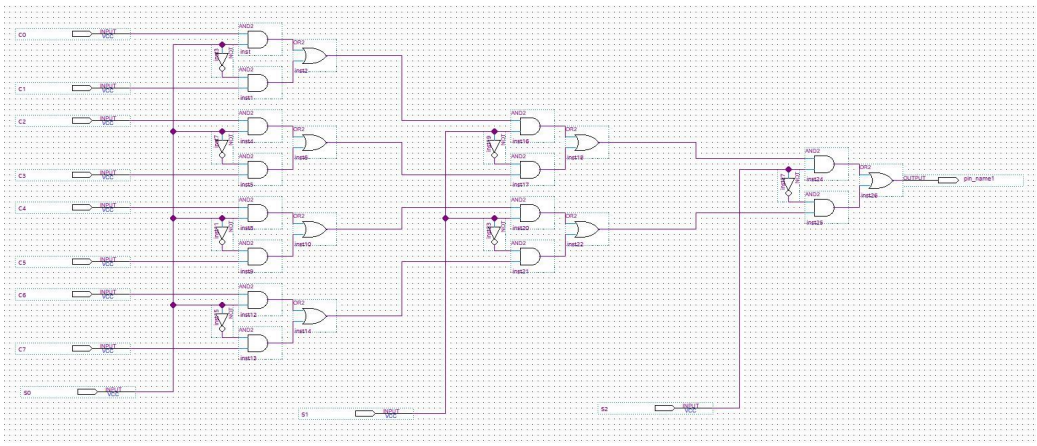


Figure A5. 3x8 Decoder Verilog.

```
module decoder3to8 (EN, W2, W1, W0, Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7);
    input EN, W2, W1, W0;
    output reg Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7;

    always @(EN, W2, W1, W0)
    Begin
        {Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7} = 8'd0;
        if (EN)
        Begin
            case({W2, W1, W0})
            3'b000 : Y0 = 1'b1;
            3'b001 : Y1 = 1'b1;
            3'b010 : Y2 = 1'b1;
            3'b011 : Y3 = 1'b1;
            3'b100 : Y4 = 1'b1;
            3'b101 : Y5 = 1'b1;
            3'b110 : Y6 = 1'b1;
            3'b111 : Y7 = 1'b1;
            Endcase
        End
    End
endmodule
```

Part B: Comparator

The purpose of the comparator is the compare two 4-bit numbers(A[0-4], B[0-4]). In the case of this machine we need to compare the magnitude of the integers. One way to do this is by subtracting the numbers A-B. If the result of this subtraction is negative or zero we know that A less than or equal to B and therefore the two numbers are sorted. A simpler way to solve this is with a boolean expression. Consider the table below comparing two 1-bit numbers:

A	B	A<=B
0	0	1
0	1	1
1	0	0
1	1	1

This same logic can be applied to a 4-bit number: (see figure B1).

Looking at the block diagram one can observe the data in line at the top left and the output F or (A<=B) on the right. The xnor's on the bottom are responsible for the equality of each bit of numbers A and B. The rest of the logic is to determine if A<B (derived in figure B1). the output of this circuit is a logical high if A<=B. The flag will be used in the finite state machine to allow comparison to continue if the list is sorted (A<=B) at the selected addresses.

EE 281
12/10/19
Section 16

Figure B1. Comparator Boolean Solution.

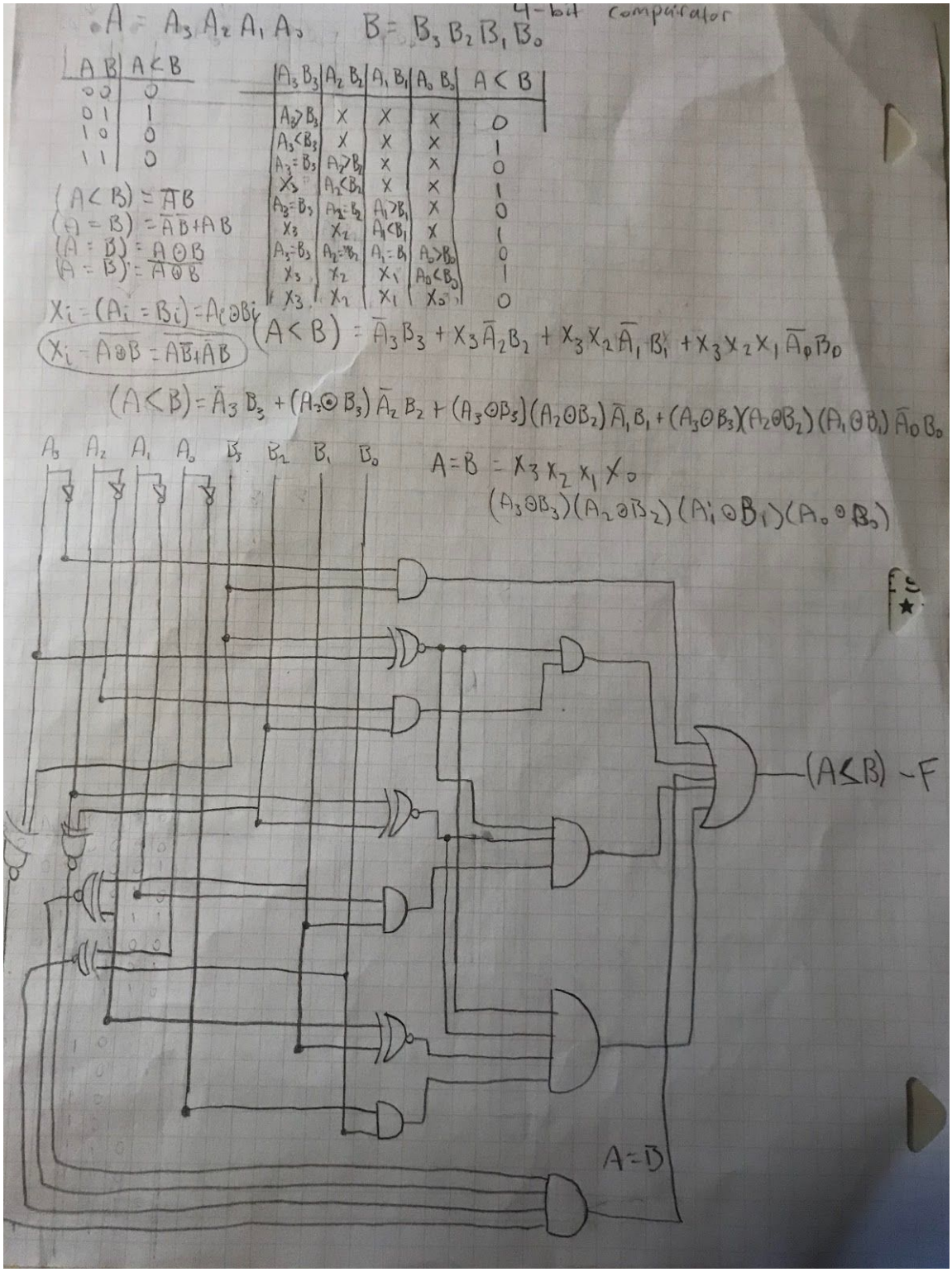
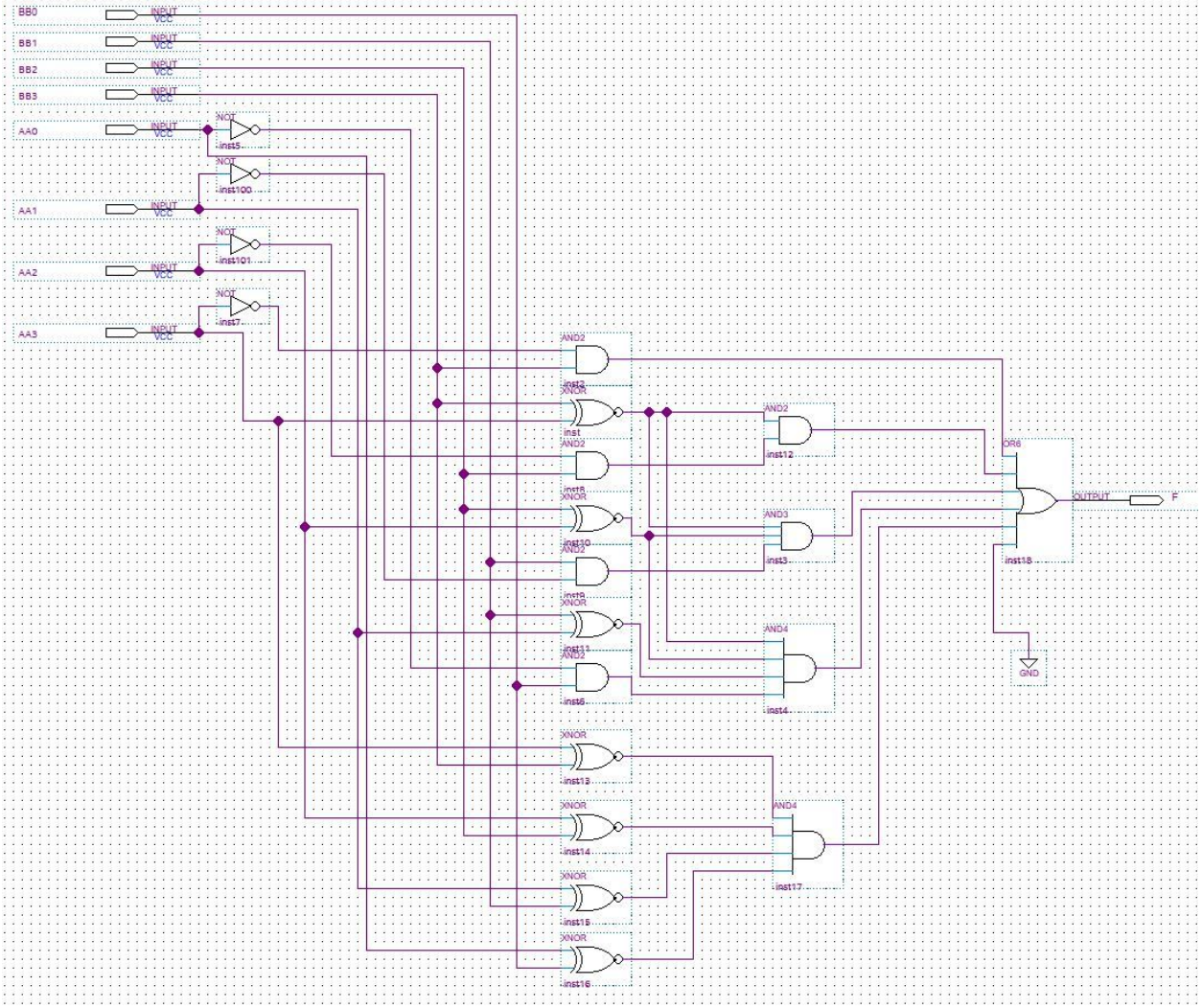


Figure B2. Comparator.



Part C: Moore Finite State Machine

The purpose of the finite state machine (FSM) is to take user inputs and control the register file as well as the comparator. Figure C1 shows that states table as well as the state diagram to get a better idea of how this FSM will function. The start state takes a user input and allows the machine to start comparing all the numbers in the list by counting. If the comparator outputs a true meaning the two numbers being compared are

sorted the FSM is allowed to continue counting. If the numbers are not sorted the counter will be put into a failed state. This failed state will stop the counter and output the address at which the list of numbers is not sorted. The end state is only reached once all numbers are compared. This state will output a green led indicating that a list of numbers is sorted in ascending order.

When the stated are listed in a state table it is clear that a number of five variable K-maps would be required to determine the logic for this FSM. there are a couple ways one can simplify this FSM if one recognizes that we are outputting 2 three bit numbers and that the second number is always one greater than the first. Therefore a 3-bit adder hard coded to 001 or 1 will output the second 3-bit number needed to run the comparison. This means the FSM only needs to count one 3-bit number. This makes it much easier to simplify into a circuit see figure C2 & C4.

The other major simplification that can be made is to the type of counter being used. Figure C2 shows a synchronous up counter. In a synchronous counter all of the clock lines are wired together. Figure C3 shows an asynchronous counter which is much simpler to implement because it will always count up we just need to reset the flip flops when our target value is reached. In an asynchronous counter, the first flip-flop is driven by a pulse from an external clock and each successive flip-flop is driven by the output of the preceding flip-flop in the sequence.

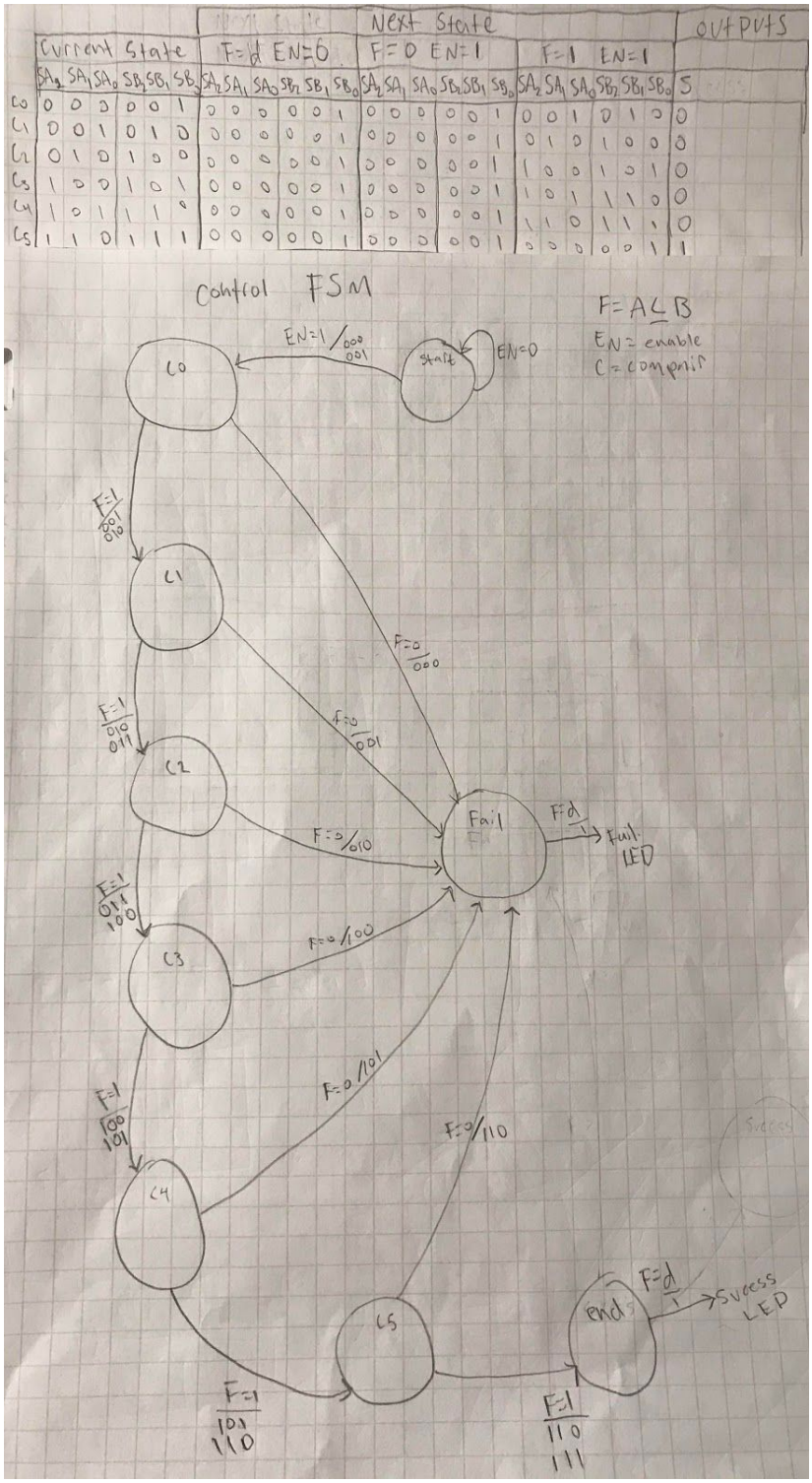
Figure C4 shows the FSM in its entirety. To the left we have the 3-bit asynchronous counter, the first flip-flop is driven by a pulse from an external clock and

each successive flip-flop is driven by the output of the preceding flip-flop in the sequence counter which counts from 000 to 110. On top there is an and gate that resets the counter if the output of the counter is 110 or if the reset line is pressed. Below the counter there is another and gate. This gate takes in a CLK, (A<=B), and count_enable. This signal reeds the clock input of the first DFF. To the right of figure C4 is the 3-bit adder made up with 3 full adders. One of the inputs of this adder is the output of the counter. The other input is hard coded to 001 or 1 therefore the output, on the bottom of the adder is input + 1. Finally the outputs of the adder are wired to an and gate which will enable a successful LED if the final value is reached (111). The output of the counter and adder are used to select which numbers are being read from the register file and passed to the comparator.

The adder in the circuit is made up of 3 full adders see figure C5. these full adders are wired to the output of the modulo 7 counter and hard coded to 001 or 1. This allows the FSM to read two different numbers from both ports of the register file in one clock cycle. Since this FSM only needs to output a 3 bit number that is 1+the number generated by the counter we don't need to worry about using xnors for subtraction functionality or c_out. It is however important to tie c_in to ground and connect the c_out of least significant bit full adder to the c_in of the next larger bit's full adder. The c_out of the greatest bit full adder can even be used to trigger a success state as 111 (B's read address) is the final state for witch comparison needs to occur between A and B before the machine determines that the list is sorted.

EE 281
12/10/19
Section 16

Figure C1. Complex FSM.



EE 281

12/10/19

Section 16

Figure C2. Simplified FSM.

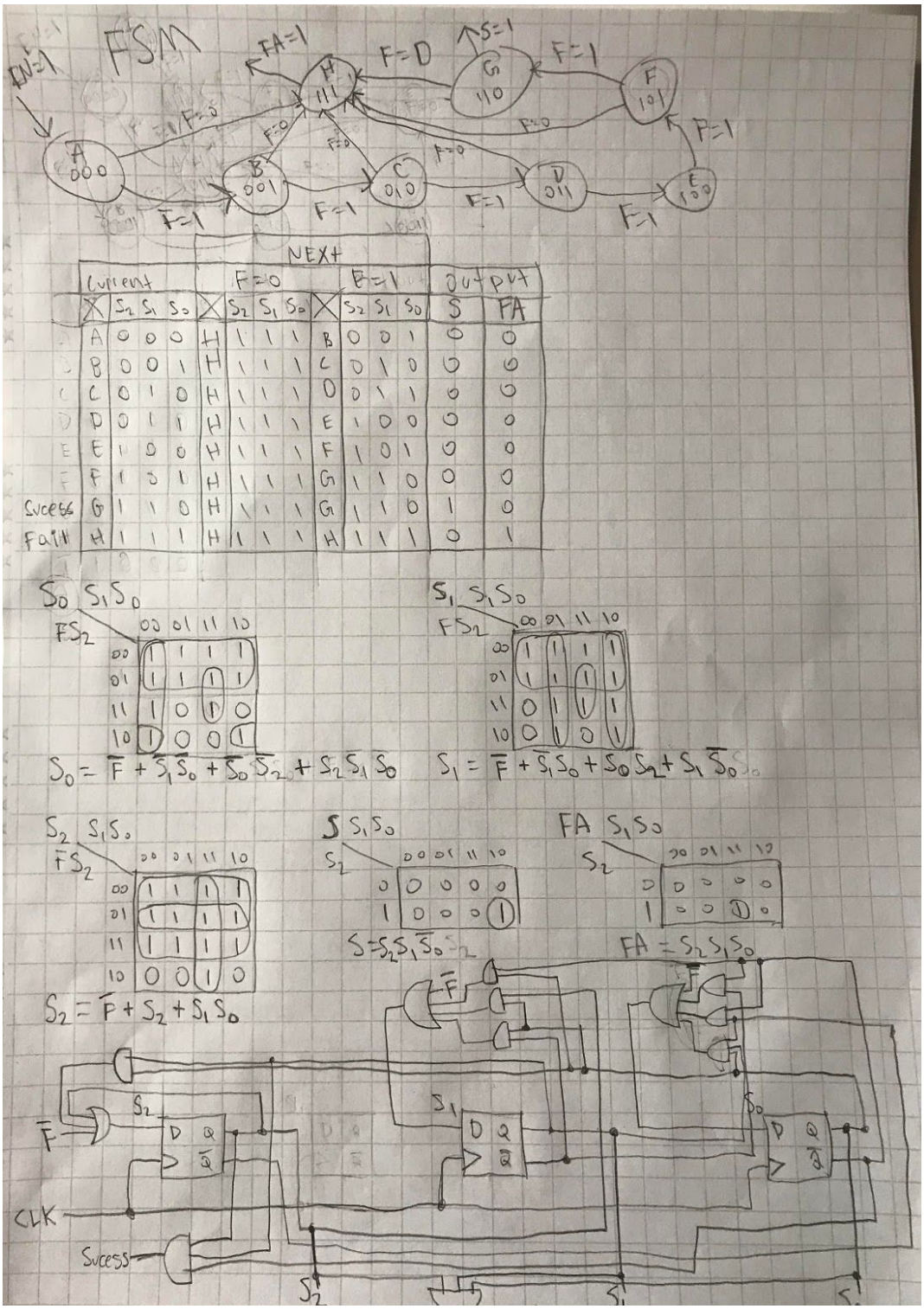


Figure C3. Simplest FSM.

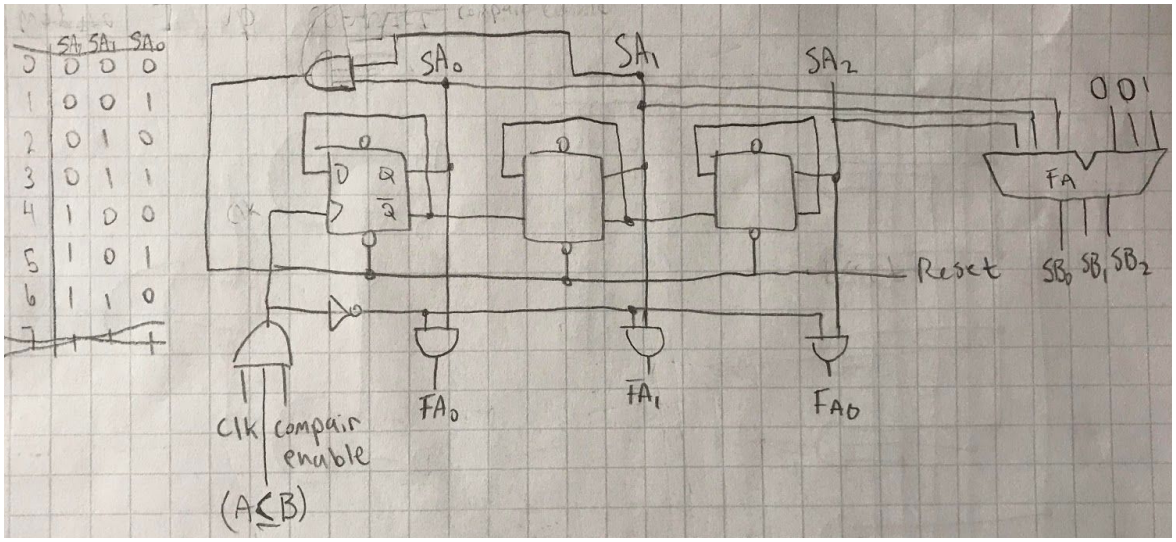


Figure C4. Simplest FSM.

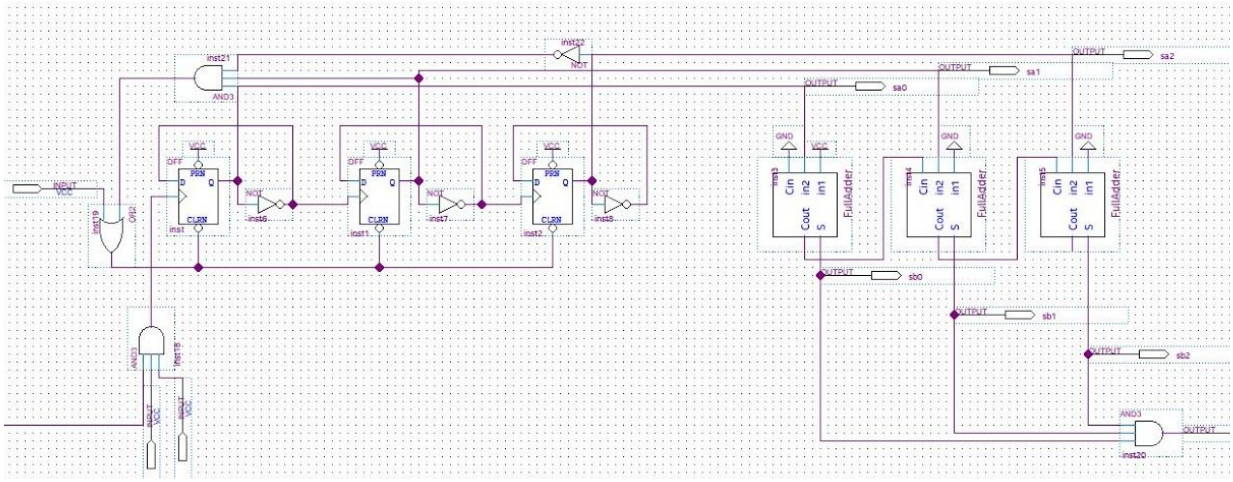
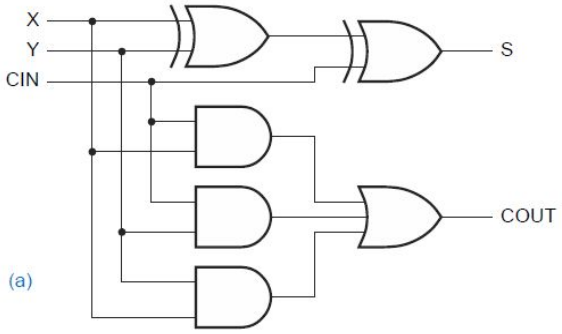


Figure C5. Full Adder.



Part D: Other Devices

There are a couple other devices what will be used in conjunction with the machine to make it easier to use. One of these devices is a clock divider and debounce circuit (figure D2 & D1). While this is not necessarily required to operate the machine, it makes it easier for a user to see what's happening and test the circuit. If the machine were connected directly to the clock of the altera board the comparison would happen in less than a second. While this is good for actually running the calculation, for testing and demonstration the circuit will have a manually advanced clock.

Figure D2 shows a clock divider witch is just a series of T flip-flops which are triggered on positive edge of a clock cycle meaning that each flip flop in the chain will divide the clock by 2 to the number of flip flops.

Figure D1 shows the debounce circuit. This circuit takes in the clock from the board and divides it by 2 to the 20th power since there are 20 flip flops. The manual clock signal is connected to the input of the DFF and the smoothed clock signal is output from the DFF. Essentially this circuit works by slowing the clock down and triggering on a human input which can cause a bounce in the signal (instead of just going high when presses the signal will oscillate between high and low when pressed). With a slower clock the user input has much more time to settle to high or low before the DFF is triggered with a debounced output.

The final accessory device used by this machine is a 7 segment decoder. There are two of these decoders in the machine, one for each of the values being compared (A &

B). The decoder works by mapping 4-bit inputs to a hexadecimal output on the 7 segment display see figure D3 & D4. for example, if we pass the decoder the number 1 display segments B and C should be lit to make a 1. (note that logical 0 represents a lit segment.)

Figure D1. Debounce.

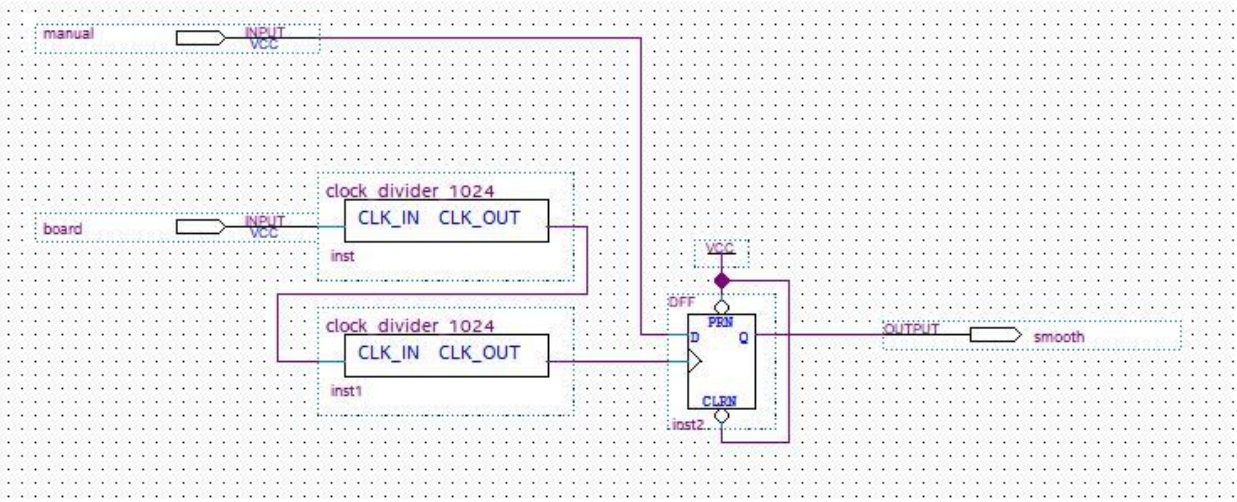


Figure D2. Clock Divider.

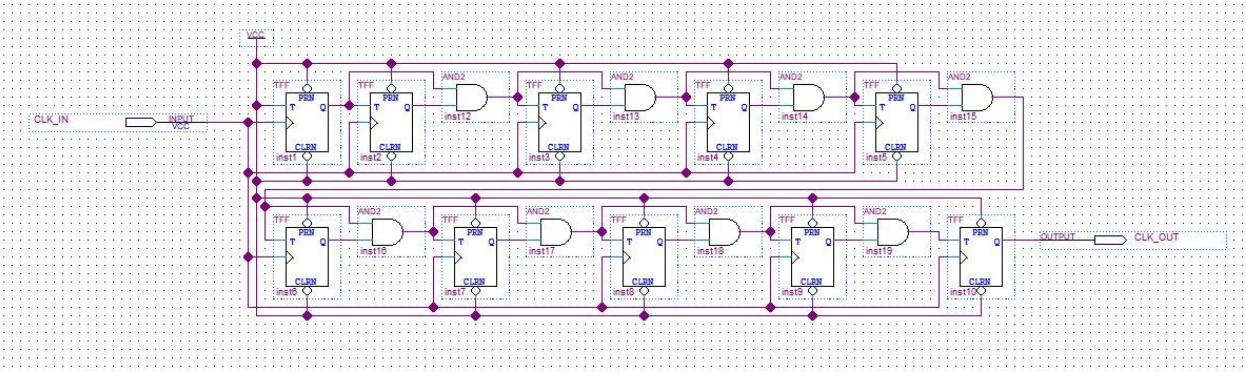


Figure D3. Seven Segment Decoder Verilog.

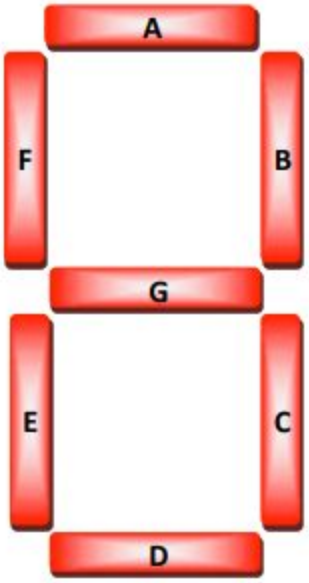
```
module 7segDecoder(A,B,C,D,E,F,G,X0,X1,X2,X3);
        input X0,X1,X2,X3;
        output A,B,C,D,E,F,G;
        reg A,B,C,D,E,F,G;

        always @(X3 or X2 or X1 or X0)
        begin
```

```
            case ({X3,X2,X1,X0})

                4'b0000: {A,B,C,D,E,F,G}=7'b0000001;
                4'b0001: {A,B,C,D,E,F,G}=7'b1001111;
                4'b0010: {A,B,C,D,E,F,G}=7'b0010010;
                4'b0011: {A,B,C,D,E,F,G}=7'b0000110;
                4'b0100: {A,B,C,D,E,F,G}=7'b1001100;
                4'b0101: {A,B,C,D,E,F,G}=7'b0100100;
                4'b0110: {A,B,C,D,E,F,G}=7'b0100000;
                4'b0111: {A,B,C,D,E,F,G}=7'b0001111;
                4'b1000: {A,B,C,D,E,F,G}=7'b0000000;
                4'b1001: {A,B,C,D,E,F,G}=7'b0000100;
                4'b1010: {A,B,C,D,E,F,G}=7'b0001000;
                4'b1011: {A,B,C,D,E,F,G}=7'b1100000;
                4'b1100: {A,B,C,D,E,F,G}=7'b0110001;
                4'b1101: {A,B,C,D,E,F,G}=7'b1000010;
                4'b1110: {A,B,C,D,E,F,G}=7'b0110000;
                4'b1111: {A,B,C,D,E,F,G}=7'b0111000;
            endcase
        end
Endmodule
```

Figure D4. Seven Segment Display Diagram.



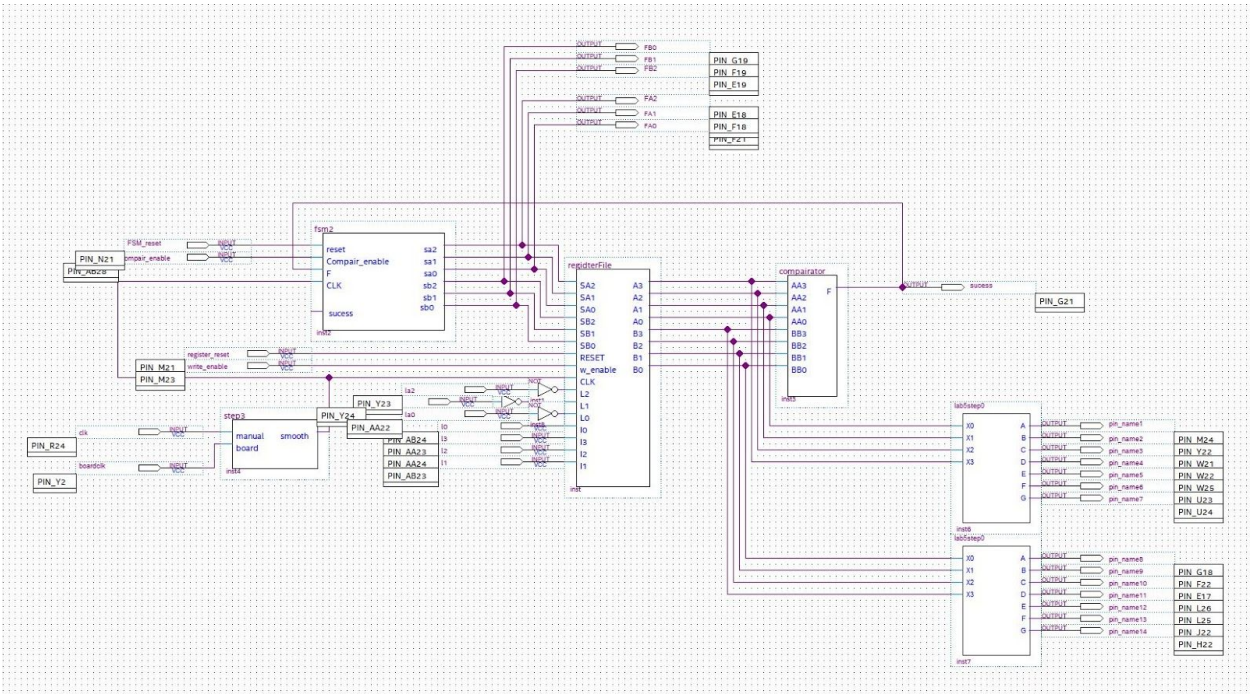
Part E: Overall

The overall block diagram is much easier to understand after taking a look at the individual components. Figure E1 shows the block diagram of the finished machine. At the top of the block diagram are the outputs for the memory location of the two numbers being compared. This output takes two 3-bit signals between the FSM and the register file because the FSM is what determines these read addresses. To the left is the FSM which takes an input from the user Compair_Enable and an output from the comparator, F. if conpair_enable and F are a logical 1 the FSM will begin comparing numbers. The output of the FSM has two 3 bit read addresses which select witch two numbers of the register file are to be compared.

Moving to the right is the register file itself. On the left are the read address inputs as previously discussed. Below those inputs we have the user inputs for state 1 which consist of a 3-bit write address, a 4-bit parallel load line, and a write enable. These inputs are used in state 1 (loading data). On the right of the register file there are two 4-bit parallel read ports that go to the comparator and each 4-bit line also goes to a 7 segment decoder to display to the user the two numbers being compared.

EE 281
12/10/19
Section 16

Figure E1. Overall Block Diagram.



Part F: What would I do differently?

This design underwent a couple slight modifications in order to demonstrate its capabilities. The seven segment decoders were relocated to the register file and each of the eight displays of the altera board were hard wired to the output of each register in the register file. With this update the user was able to see the value stored in each register as it was loaded.

Similar to the 7 segment display the key push buttons are inverted in a normally closed position creating a negative edge rather that a positive edge to occur when pressed. This is particularly an issue when writing to the register file because the DFF's are triggered on a positive edge. To fix this the key inputs were ran through a not gate to invert their signal.

Organization with block diagrams became an issue as the machine grew. Take the multiplexer mistake for example figure A4. to correct the labeling mistake I had to re label each multiplexer then adjust the symbol to easily fit it back into the register. Instead of making a 3x8 MUX with primitive logical gates I should have made it out of 1x2 multiplexer symbols (or verilog) the wire the 1x2's into a 3x8. By doing this to fix the labeling issue all I would have to do is go back to the 1x2 symbol and reverse the not gate, then update the symbols and recompile.

Another example of where a block diagram would have been useful was in the FSM the counter and adder should each have their own symbols so they can easily be put together to make the entire FSM. This would help keep track of where the most and least significant bit belong. It also would have been smart to include a user input block in the FSM. to organize and understand how the machine works. The same can even be said for each register in the register file as any changes done would need to be made in 4, 8 or even 32 different spots instead of just 2 or 4 saving a lot of time while debugging.

Busses in quartus could improve the debugging and demonstration process as well. Busses make it easier to assign your individual bit output and keep track of addresses and data in the machine. If you ever need to make a change you don't need to move 4 individual wires instead you can just move the bus.