Omar Muhammetkulyyev

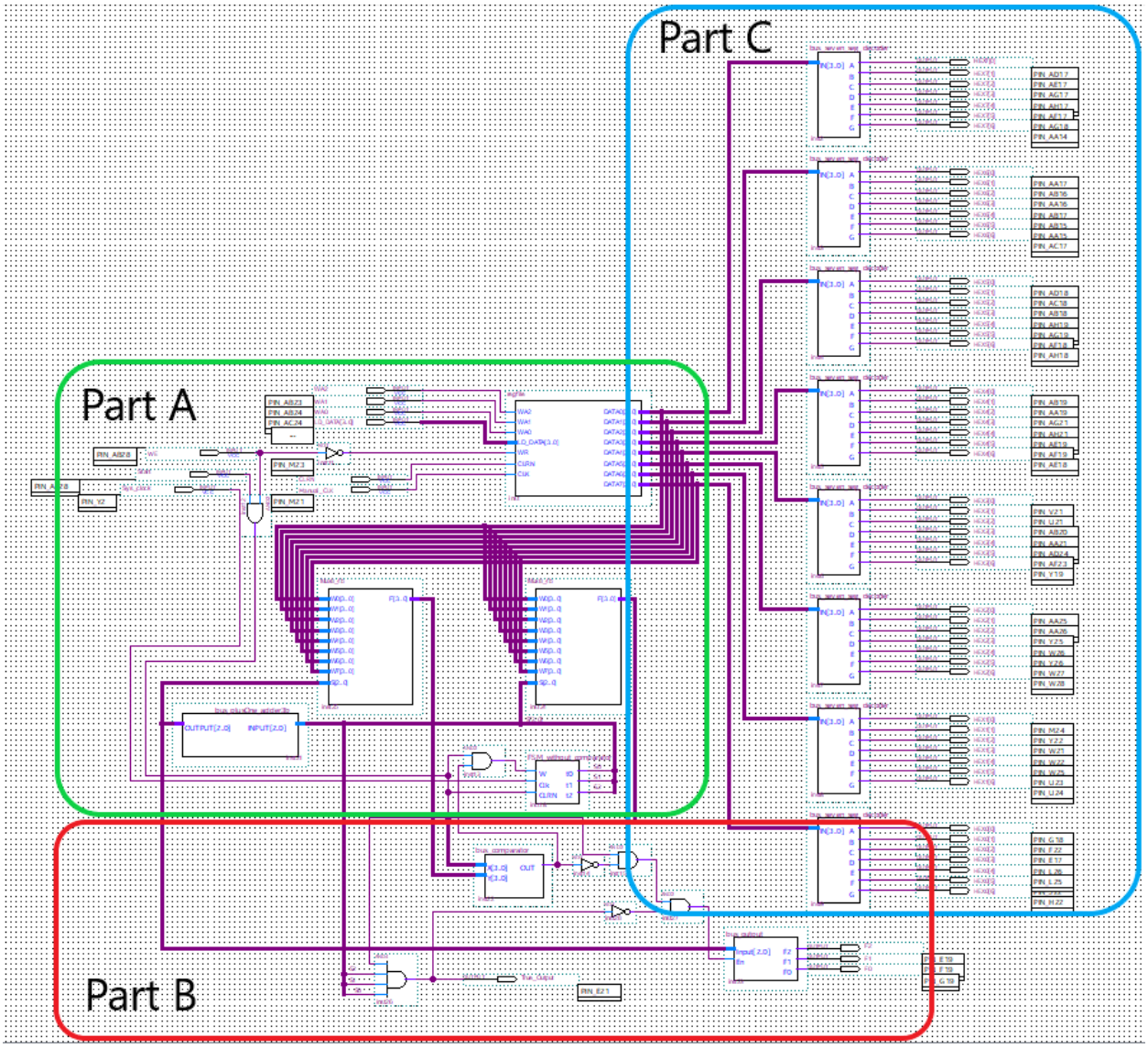
Section 9

CPRE 281

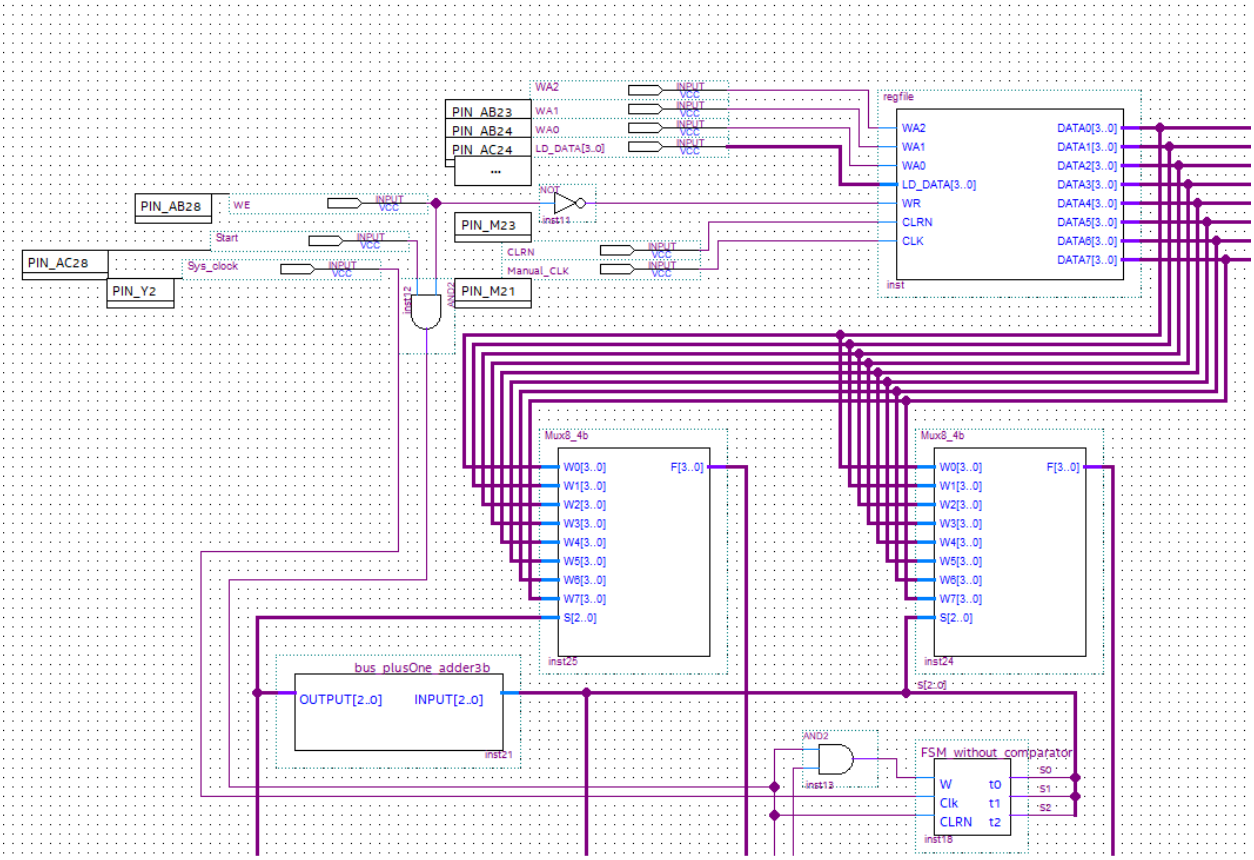
Student ID: 410026642

Final Project Report

I completed the Project number 1: Circuit for checking if a list of numbers is sorted. The report includes the top-level diagram of my circuit divided into 3 parts: A, B and C. Each of these parts is described thoroughly in their following corresponding sections.



Part A



This part includes the following which will be explained afterwards in the given order.

1. Inputs
2. Register File (regfile)
   2.1. 3-to-8 Decoder (Decoder3to8)
   2.2. 4-bit Parallel-Access Register(reg4)
3. Two 8-to-1 4-bit wide bus multiplexers (Mux8_4b)
4. Plus-one 3-bit wide bus adder circuit (bus_plusOne_adder3b)
   4.1. Adder 4-bit (adder_4bit)
5. Finite State Machine without comparator (FSM_without_comparator)
6. Other logic gates used

1. Inputs

Part A contains all of the inputs to the top-level diagram of the project which are **WA2**, **WA1**, **WA0**, **LD_DATA[3..0]**, **CLRN**, **Manual_CLK**, **WE**, **Start**, **Sys_clock**.

The **WA2**, **WA1** and **WA0** inputs indicate the write address – number from 0 to 7 - of the register in the register file to write the given input to. And **LD_DATA[3..0]** is the load data provided by the user which essentially is a hexadecimal digit from **0** to **F**.

CLRN is pinned to the pushbutton that is used to clear/reset the contents of all 8 registers in the register file. **Manual_CLK** is used to enter the given input **LD_DATA[3..0]** to the register specified by **WA2**, **WA1**, **WA0** in the register file on every positive edge.

WE is the write enable switch that serves as the mode switcher for the system to switch between the *Initialization* and *Checking* Modes: **0** for *Initialization* and **1** for *Checking*. **Start** is a VALIDATE input pinned to a switch that when asserted initiates the checking process. **Sys_clock** is connected to PIN_Y2 and functions as 50MHz clock for the Finite State Machine.

2. Register File (regfile)

```
1   module regfile(WA2, WA1, WA0, LD_DATA, WR, CLRN, CLK, DATA0, DATA1, DATA2, DATA3, DATA4, DATA5, DATA6, DATA7);
2      input WA2, WA1, WA0, WR, CLRN, CLK;
3      input [3:0] LD_DATA;
4      output [3:0] DATA0, DATA1, DATA2, DATA3, DATA4, DATA5, DATA6, DATA7;
5      wire [7:0] YY;
6      wire [3:0] VALUE0, VALUE1, VALUE2, VALUE3, VALUE4, VALUE5, VALUE6, VALUE7;
7
8      Decoder3to8 my_decoder(.W({WA2, WA1, WA0}), .EN(WR), .Y(YY));
9
10     reg4 myregister0 (.IN(LD_DATA), .LD(YY[0]), .CLK(CLK), .OUT(VALUE0), .CLRN(CLRN));
11     reg4 myregister1 (.IN(LD_DATA), .LD(YY[1]), .CLK(CLK), .OUT(VALUE1), .CLRN(CLRN));
12     reg4 myregister2 (.IN(LD_DATA), .LD(YY[2]), .CLK(CLK), .OUT(VALUE2), .CLRN(CLRN));
13     reg4 myregister3 (.IN(LD_DATA), .LD(YY[3]), .CLK(CLK), .OUT(VALUE3), .CLRN(CLRN));
14     reg4 myregister4 (.IN(LD_DATA), .LD(YY[4]), .CLK(CLK), .OUT(VALUE4), .CLRN(CLRN));
15     reg4 myregister5 (.IN(LD_DATA), .LD(YY[5]), .CLK(CLK), .OUT(VALUE5), .CLRN(CLRN));
16     reg4 myregister6 (.IN(LD_DATA), .LD(YY[6]), .CLK(CLK), .OUT(VALUE6), .CLRN(CLRN));
17     reg4 myregister7 (.IN(LD_DATA), .LD(YY[7]), .CLK(CLK), .OUT(VALUE7), .CLRN(CLRN));
18
19     assign DATA0 = VALUE0;
20     assign DATA1 = VALUE1;
21     assign DATA2 = VALUE2;
22     assign DATA3 = VALUE3;
23     assign DATA4 = VALUE4;
24     assign DATA5 = VALUE5;
25     assign DATA6 = VALUE6;
26     assign DATA7 = VALUE7;
27
28  endmodule
```

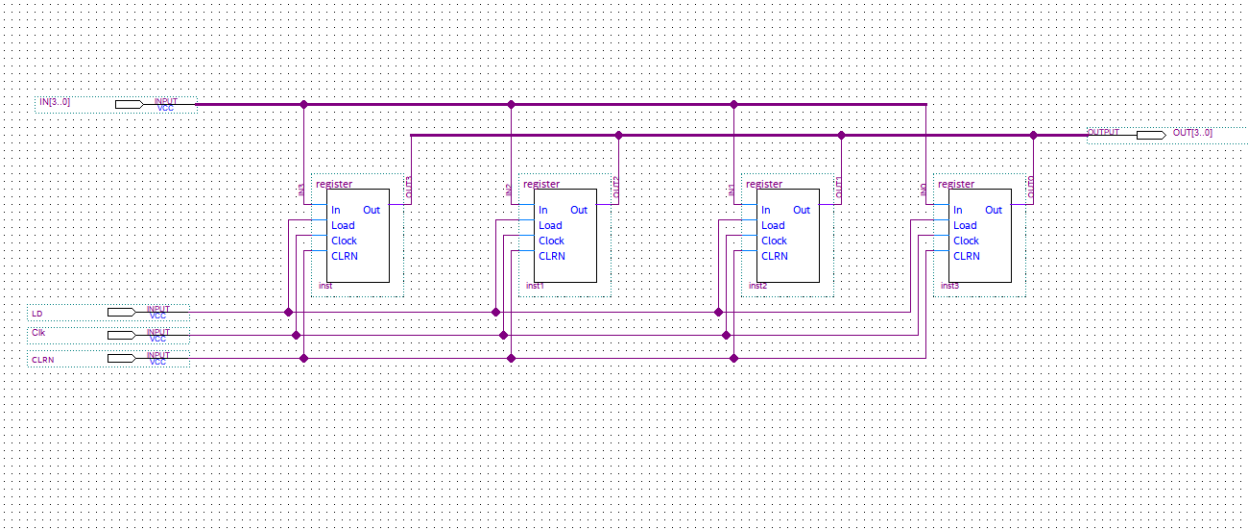
Regfile is a register file module with inputs **WA2**, **WA1**, **WA0**, **LD_DATA[3..0]**, **WR**, **CLRN**, **CLK** and outputs **DATA0[3..0]**, **DATA1[3..0]**, **DATA2[3..0]**, **DATA3[3..0]**, **DATA4[3..0]**, **DATA5[3..0]**, **DATA6[3..0]**, **DATA7[3..0]**. It has 8 4-bit parallel-access registers (reg4) to load the 4-bit **LD_DATA[3..0]**. The given data is loaded to the write address specified by **WA2**, **WA1** and **WA0**. 3-to-8 Decoder is used to produce 8-bit one-hot-encoded line (**YY[7:0]**) to decode the given write address where each of those 8 bits are fed as load enable line for each of 8 registers. **WR** input is fed to the decoder as a write enable line.

2.1. 3-to-8 Decoder(Decoder3to8)

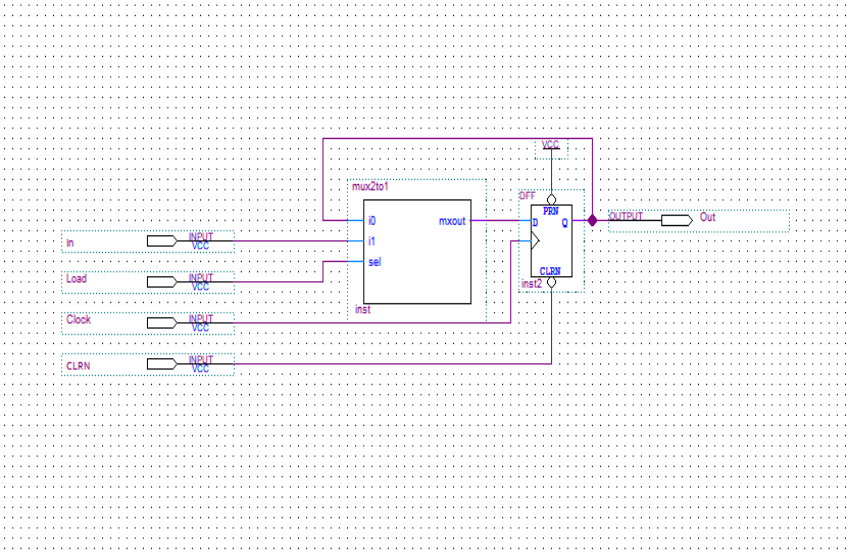
```
module Decoder3to8(W, EN, Y);
    input [2:0] W;
    input EN;
    output reg [7:0] Y;

    always @(W, EN)
        case({EN, W})
            4'b1000: Y = 8'b00000001;
            4'b1001: Y = 8'b00000010;
            4'b1010: Y = 8'b00000100;
            4'b1011: Y = 8'b00001000;
            4'b1100: Y = 8'b00010000;
            4'b1101: Y = 8'b00100000;
            4'b1110: Y = 8'b01000000;
            4'b1111: Y = 8'b10000000;
            default: Y = 8'b00000000;
        endcase

endmodule
```

3-to-8 Decoder with select line **W[2..0]**, enable line **EN** and output **Y[7..0]**. One of 8 bits of output is asserted according to the given **W[2..0]** and only when **EN** is **1**. Otherwise **Y** is all **0**'s which means that none of the registers will accept parallel-load.

2.2. 4-bit Parrallel-Access Register (reg4)



4-bit Parallel-Access Register that comprises 4 1-bit parallel-access registers, inputs **IN[3..0], LD, Clk** and **CLRN** and output **OUT[3..0]**. It is a basic register with parallel load enable line **LD** that when asserted accepts the data from **IN[3..0]** into its 4 registers on each positive clock edge. Contents of the registers are then fed to **OUT[3..0]** regardless of **LD**.



To the left is the 1-bit register module of the above reg4 module. It uses simple 2-to-1 multiplexer to select between **Out** and **In** (to keep the current value or to load new input) based on **Load**. D flip-flop is used to keep the input in this register with preset option disabled. **CLRN** forces to the data in this register be set to 0 on negative edge regardless of **Clk**. Data is loaded or the current state is kept on each positive edge of the **Clk**.

To the right is the mux2to1 module of the above register module. Simple 2-to-1 multiplexer that assigns **mxout** value of **i0** when **sel** is **0** and **i1** when **sel** is **1**.

```
1  module mux2to1(i0, i1, sel, mxout);
2      input i0, i1, sel;
3      output mxout;
4
5      assign mxout = sel == 0 ? i0 : i1;
6
7  endmodule
```

3. Two 8-to-1 4-bit wide bus multiplexers (Mux8_4b)

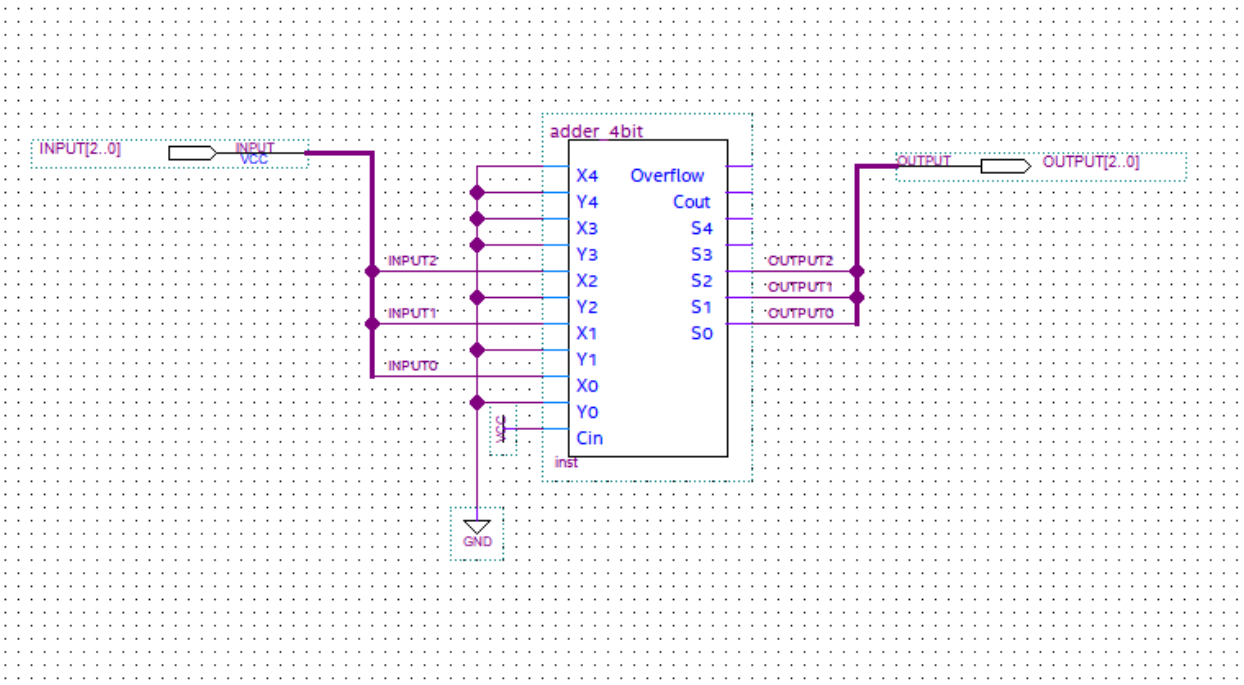
The purpose of the two 8-to-1 bus multiplexers is to select the register to read the data from. Then the data read from **register$_i$** selected with the help of the first(right) bus multiplexer is compared with its neighbor **register$_{i+1}$** which is selected by the second(left) multiplexer. The comparator module is detailed in the Part B of the report. Select lines to the first multiplexer is given by the Finite State Machine which is described in the following Section 5 of Part A. Select line to the second multiplexer is produced by adding **1** to the select line of the first one which is done via bus_plusOne_adder3b module described in Section 4 of Part A.

```
1   module Mux8_4b(W0, W1, W2, W3, W4, W5, W6, W7, S, F);
2      input [3:0] W0, W1, W2, W3, W4, W5, W6, W7;
3      input [2:0] S;
4      output [3:0] F;
5      wire [3:0] X1, X2, X3, X4, Y1, Y2;
6
7      Mux2_4b Mux1_1 (W0, W1, S[0], X1);
8      Mux2_4b Mux1_2 (W2, W3, S[0], X2);
9      Mux2_4b Mux1_3 (W4, W5, S[0], X3);
10     Mux2_4b Mux1_4 (W6, W7, S[0], X4);
11
12     Mux2_4b Mux2_1 (X1, X2, S[1], Y1);
13     Mux2_4b Mux2_2 (X3, X4, S[1], Y2);
14
15     Mux2_4b Mux3_1 (Y1, Y2, S[2], F);
16
17  endmodule
18
19  module Mux2_4b(A, B, s, Out);
20     input [3:0] A, B;
21     input s;
22     output [3:0] Out;
23
24     assign Out = s == 0 ? A : B;
25
26  endmodule
```

The above is the Verilog code for 8-to-1 bus multiplexer with inputs **WN[3..0]** (0 <= **N** <= 7), **S[2..0]** and output **F[3..0].** It essentially selects one of the given 8 4-bit inputs based on the select line **S[2..0**] and assigns it to output **F[3..0]**. It is done by first selecting between **W0** and **W1**, **W2** and **W3**, **W4** and **W5**, **W6** and **W7** and assigning them to **X1**, **X2**, **X3** and **X4** based on **S[0]** with the help of the helper module Mux2_4b which is a 4-bit 2-to-1 multiplexer. With the same module then **X1** and **X2**, **X3** and **X4** are being selected based on the value of **S[1]** assigning the outputs to **Y1** and **Y2** respectively. Finally, the output **F[3..0]** is chosen between **Y1** and **Y2** based on the value of **S[2]**.

4. Plus-one 3-bit wide bus adder circuit (bus_plusOne_adder3b)

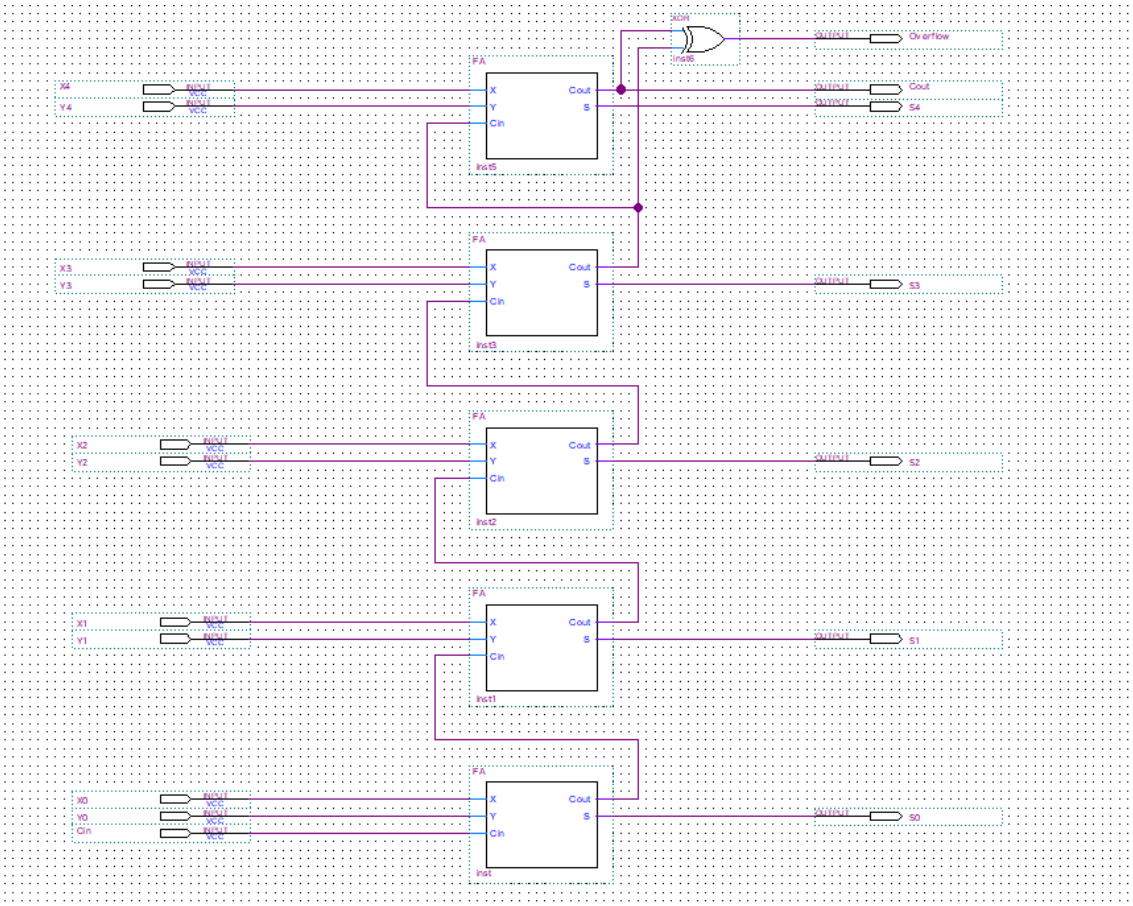
This is a simple 3-bit adder module that adds **1** to the provided input. It is used to provide the select line to the second(left) Mux8_4b multiplexer to read the data from **register**$_{i+1}$.



It uses the 4-bit adder circuit with inputs **XN** (0 <= **N** <= 4), **YN** (0 <= **N** <= 4) and **Cin** and outputs **Overflow**, **Cout** and **SN** (0 <= **S** <= 4). Even though it has 5 bits I decided to call 4-bit adder because the given numbers should be in 2's complement(signed). The inputs **YN** (0 <= **N** <= 4) are all connected to **GND** (logic 0) and **Cin** is connected to **VCC** (logic 1) because the purpose of this circuit is to add only **1**. Inputs **X3** and **X4** are also connected to **GND** as this circuit accepts only 3-bit inputs. The inputs are read from **INPUT[2..0]** and assigned to **OUTPUT[2..0]** after being increased by **1**.

4.1. Adder 4-bit (adder_4bit)

Below is the adder_4bit module. It is a ripple-carry adder with the inputs and outputs as described above. Overflow is detected by taking the **XOR** of **$Cout_4$** and **$Cout_5$**.



Here is the Verilog code full adder FA module

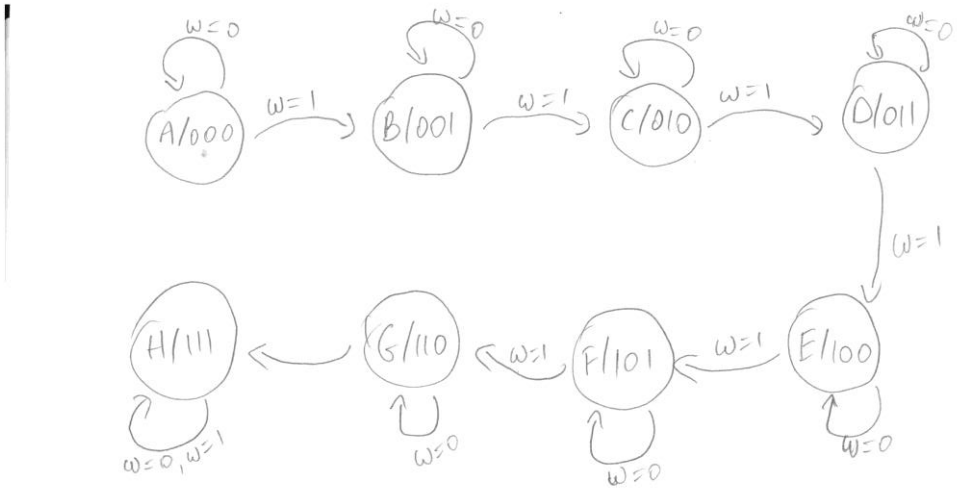
```
module FA(X, Y, Cin, Cout, S);
    input Cin, X, Y;
    output S, Cout;

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

endmodule
```

A	B	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

Sum is the XOR of **Cin, X** and **Y** whereas **Cout** occurs if any two of the inputs are asserted.

5. Finite State Machine without comparator (FSM_without_comparator)

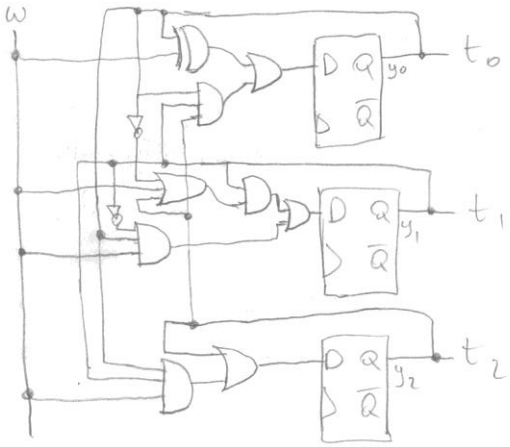
In this project Finite State Machine is used to get the address, that is the number of the register [0, 7] to be compared in the next iteration. Below are the state diagram, state table, state assigned table and the derivation of the circuit with K-maps as well as the actual derived circuit for FSM.

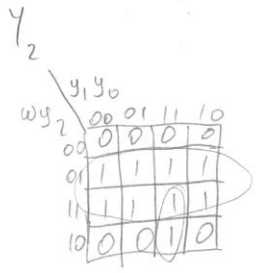


Present State	Next State w=0	Next State w=1	Output
A	A	B	0 0 0
B	B	C	0 0 1
C	C	D	0 1 0
D	D	E	0 1 1
E	E	F	1 0 0
F	F	G	1 0 1
G	G	H	1 1 0
H	H	H	1 1 1

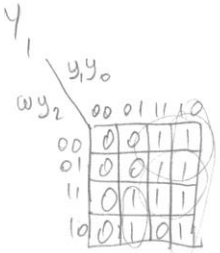
Present State	Next State $w=0$	Next State $w=1$	Output
$y_2\ y_1\ y_0$	$Y_2\ Y_1\ Y_0$	$Y_2\ Y_1\ Y_0$	$t_2\ t_1\ t_0$
0 0 0	0 0 0	0 0 1	0 0 0
0 0 1	0 0 1	0 1 0	0 0 1
0 1 0	0 1 0	0 1 1	0 1 0
0 1 1	0 1 1	1 0 0	0 1 1
1 0 0	1 0 0	1 0 1	1 0 0
1 0 1	1 0 1	1 1 0	1 0 1
1 1 0	1 1 0	1 1 1	1 1 0
1 1 1	1 1 1	1 1 1	1 1 1

$t_2 = y_2 \quad t_1 = y_1 \quad t_0 = y_0$

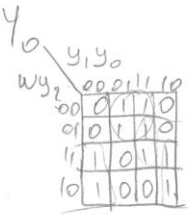




$Y_2 = y_2 + y_1 y_0 w$



$Y_1 = w\bar{y}_1 y_0 + y_1\bar{y}_0 + y_2 y_1 + \bar{w}y_1 = y_1(y_2 + \bar{y}_0 + \bar{w}) + w\bar{y}_1 y_0$

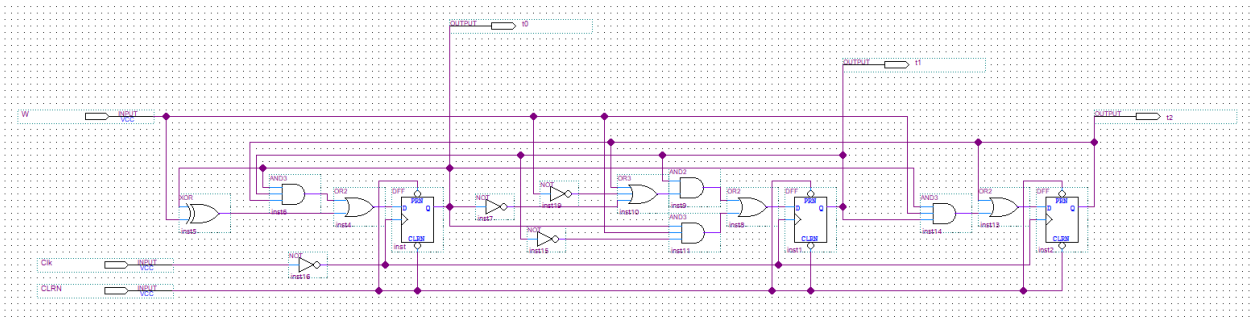


$Y_0 = w\overline{y_0} + \bar{w}y_0 + y_2 y_1 \bar{y_0} = w \oplus y_0 + y_2 y_1 y_0$

Each state except for H in the above provided diagram points to itself when **w = 0** and points to the next state when **w = 1** (i.e. when **w = 1**, **A -> B**, **E -> F** and **H -> G**). However, in state **H**, regardless of the input value **w**, it always points to itself. Here input **w** is provided by the comparator circuit which is detailed in Part B of this report: if **w = 0** then the register that is located at the address indicated by the output of current state is greater than its next register; if **w = 1** then the current register is smaller than or equal to the next one. Shortly, **w**(**t+1**) = **comparator**(**t**).

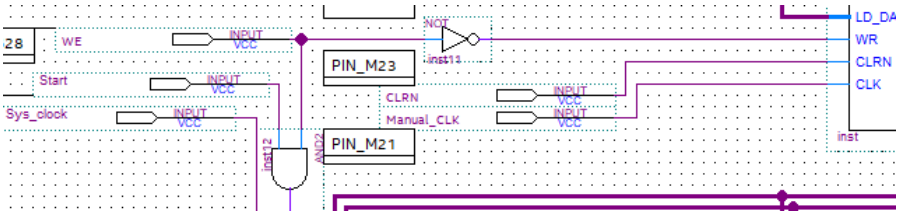
Moving on to the state assigned table, one can find it convenient to assign the letters **A**, **B**, **C**, **D**, **E**, **F**, **G** and **H** the numbers **000**, **001**, **010**, **011**, **100**, **101**, **110** and **111** respectively. The reason why it's convenient is the outputs can be uniquely identified as current state numbers ($t_2 = y_2$, $t_1 = y_1$ and $t_0 = y_0$) and all of these assigned state numbers can be implemented with 3 flip-flops. The outputs should be unique because the project requires the address of the first encountered register with lower value than its previous one displayed on the board which can happen in 7 out of 8 iterations (0^{th} excluded because it does not have a register before it). And if there is no such register then the true(or sorted) output should be asserted for which the FSM states are just enough to be done: the last 8^{th} output can be used to indicate this. To summarize, outputs of the first seven states **000**, **001**, **010**, **011**, **100**, **101** and **110** are used to indicate the address of the first register that caused the issue by simply adding 1 to it which is done via bus_plusOne_adder3b and explained in Section 4 of Part A. The output of the last state **111** can be used to assert the sorted/true output because there is no register left to compare.

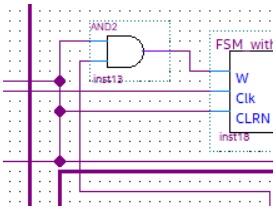
State assigned table is followed by the derivation of the circuit with K-maps and the actual FSM circuit. Below is the Quartus block diagram for FSM circuit(FSM_without_comparator).



6. Other logic gates used

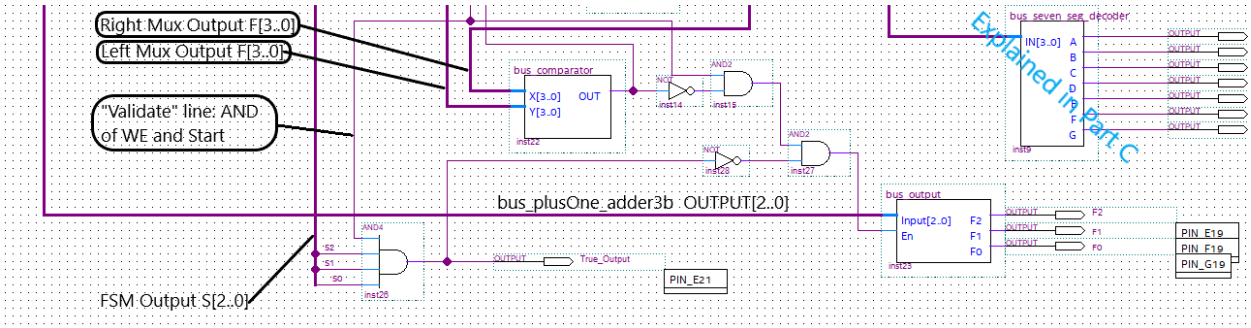
There are 3 other logic gates used in Part A which are shown and described below.





Since the *Initialization* mode is indicated via **WE = 0, inst11 NOT** gate is used to negate the mode input WE and enable the register's capability to write. Also, **inst12 AND** gate is used to and the values of **WE** and **Start** and feed the result to Finite State Machine and two outputs that will be described in Part B as "Validate/Enable" line. Another **inst13 AND** gate is shown to the right which is used to validate the output of comparator circuit with the "Validate/Enable" line. This is to make the FSM remain in its initial state **A** before **WE** and **Start** inputs are asserted.

Part B

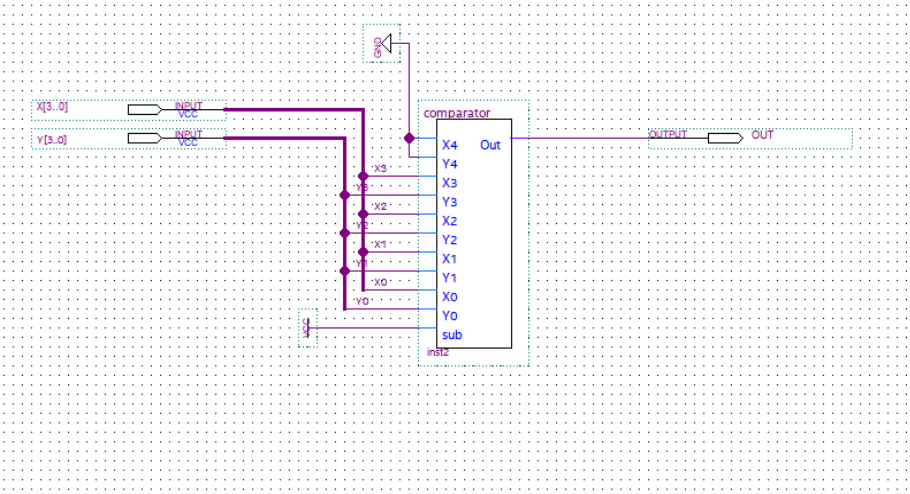


This part includes the following which will be explained afterwards in the given order.

1. Comparator Circuit(bus_comparator)
   1.1. Add_Sub module
   1.2. Equal module
2. Outputs

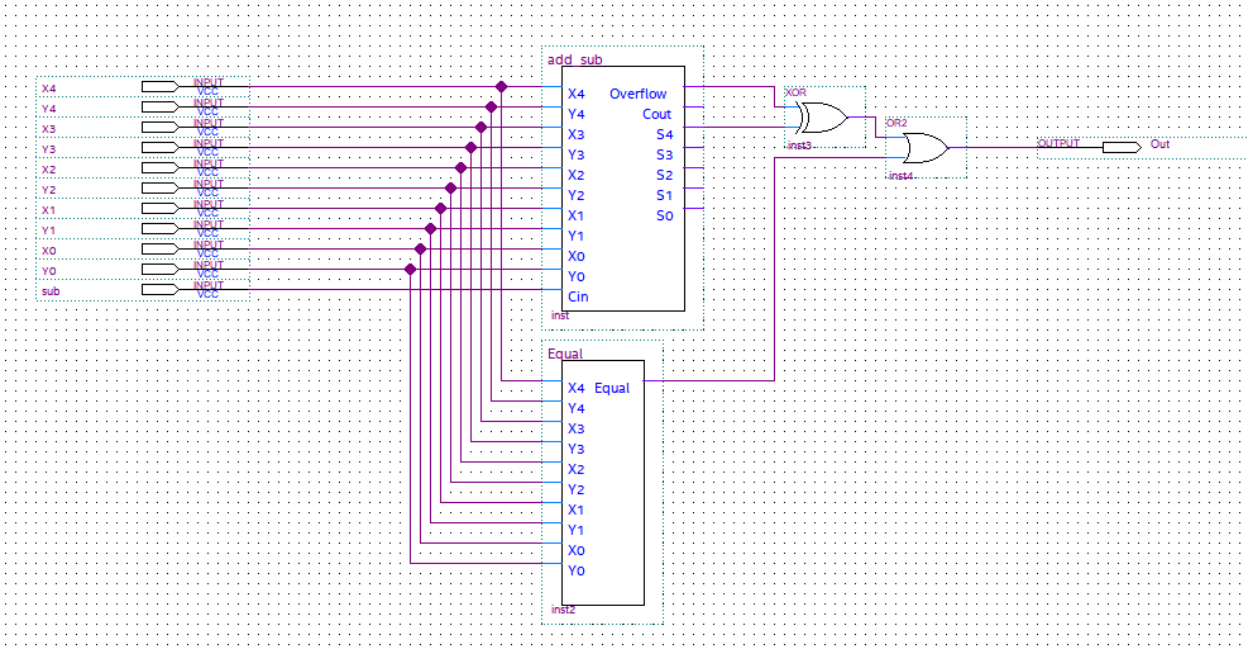
1. Comparator Circuit(bus_comparator)

Comparator circuit is enclosed by bus_comparator module for which the block diagram file is shown below.



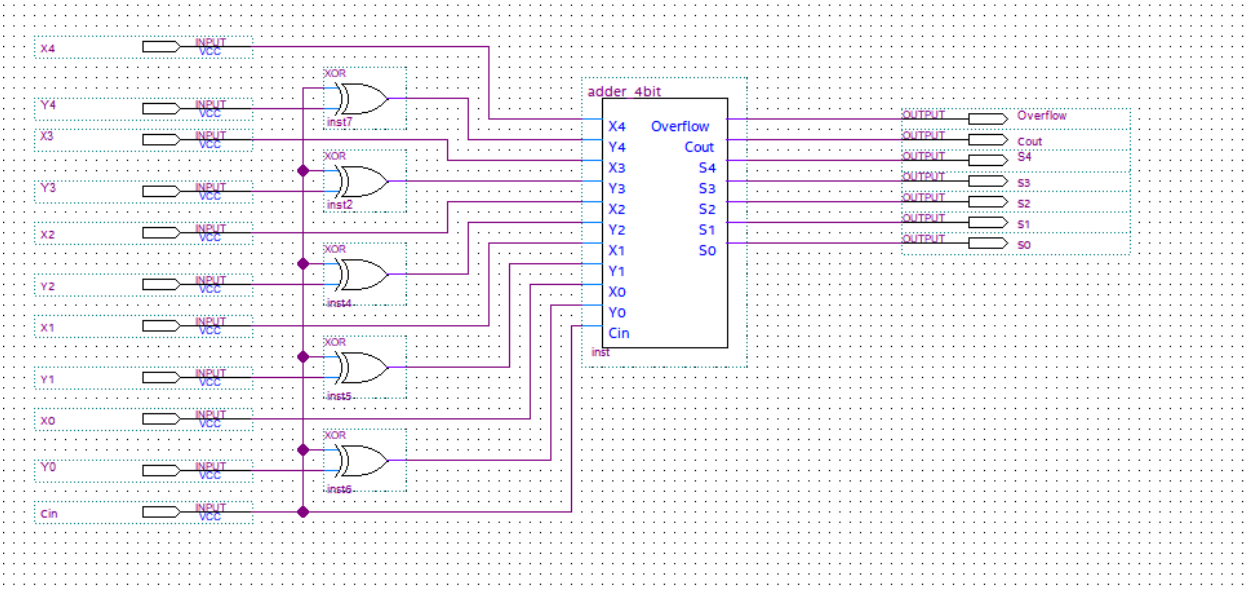
It has inputs **X[3..0]** and **Y[3..0]** and are connected to the inputs of comparator module as shown in the picture. **X4** and **Y4** are connected to **GND**(**logic 0**) because they are redundant. Also, **sub** input is connected to **VCC** (**logic 1**) because comparator circuit uses add_sub module to subtract **Y** from **X** which.

Single output **OUT** is asserted if **X** is smaller than or equal to **Y.**



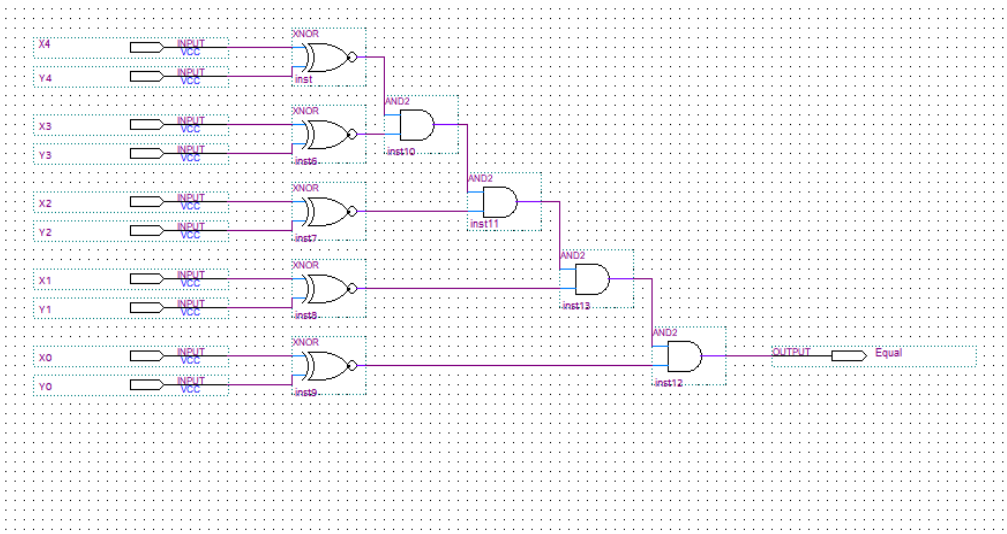
The above is the block diagram of comparator module with inputs **XN** (0 <= **N** <= 4), **YN** (0 <= **N** <= 4) and **sub**. As **X** <= **Y** can be written as (**X** == **Y**) or (**X** < **Y**), there are two modules in the diagram, namely Equal and add_sub. Equal asserts its output if the given two values are equal. However, its not so easy with add_sub module. For **X** < **Y** to be true, **X** - **Y** must be negative. One can see that if both **X** and **Y** have the same sign, then **Overflow = 0** and **S4 = 1** (negative) for **X** - **Y**. And if **X** is negative but **Y** is positive then if there is overflow, then **Overflow = 1** and **S4 = 0** (becomes positive); but if no overflow, then **Overflow = 0** and **S4 = 1** (must be negative). Hence one can conclude that for **X** < **Y** to be true **Overflow ^ S4** must be **1**. To summarize, output **Out = Equal + (Overflow ^ S4)**.

1.1. Add_sub module



Add_sub module is a ripple-carry adder circuit that also enables subtraction. **Cin** indicates the type of operation to be done: if **1** subtraction, and addition if **0**. Each bit of input **Y** is negated if input **Cin** is **1** to convert it to **-Y** in 1's complement form and left as is if **Cin** is **0**. And then input **Cin** is fed as a carry in to adder_4bit module to convert **-Y** to 2's complement form(**Cin** has no effect if **0**). Adder_4bit module is detailed in section 4.1 of Part A. Outputs are **Overflow**, **Cout** and **SN** (0 <= **N** <= 4) for the Sum.

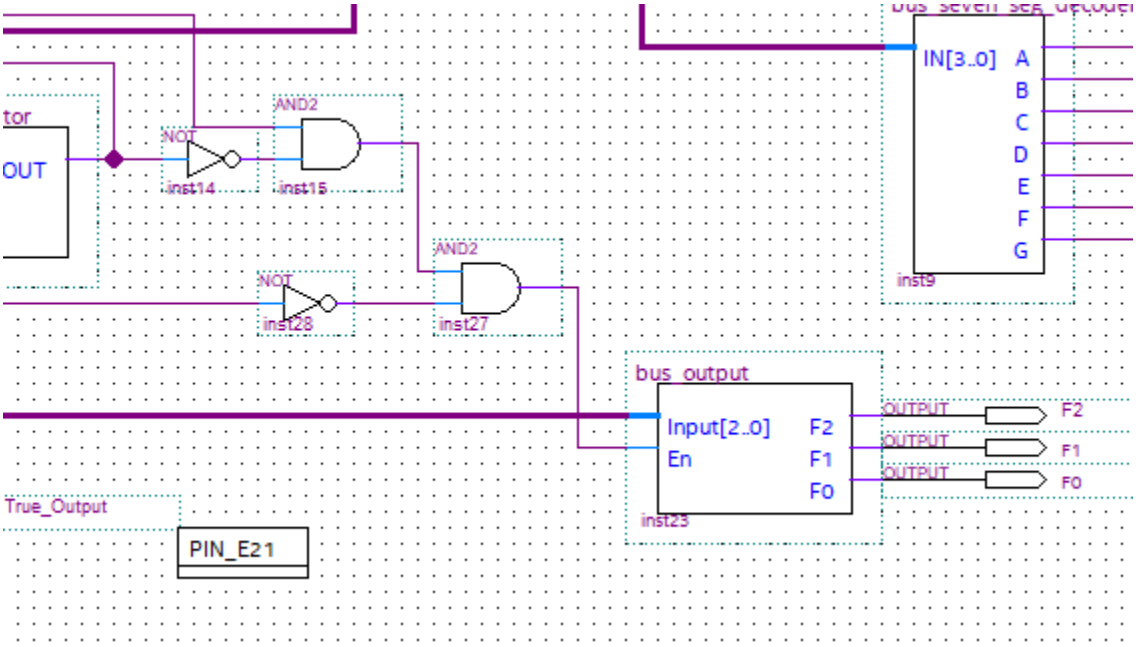
1.2 Equal module



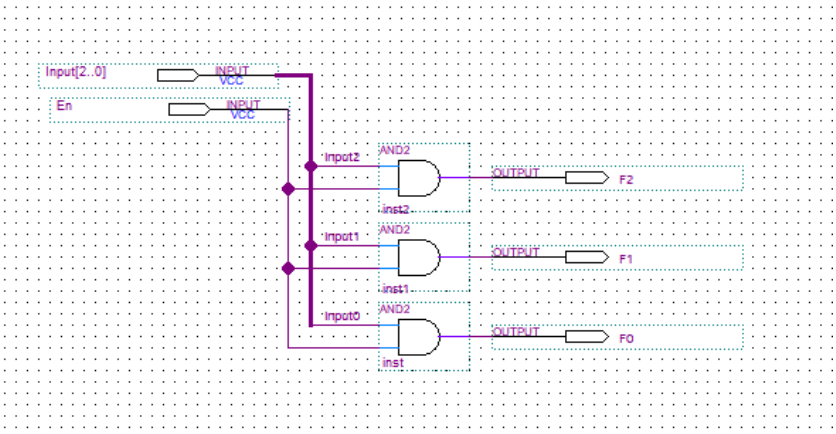
In this module two numbers **X** and **Y** are checked if they are equal as shown in the block diagram. It is done by taking **XNOR** of each bit and taking **AND** of all of the **XNOR**'s and assigning the result to output **Equal**.

Two numbers are equal only if each of their bits are equal and to check that **XNOR** is used as (**0 XNOR 0 = 1 XNOR 1 = 1**).

2. Outputs

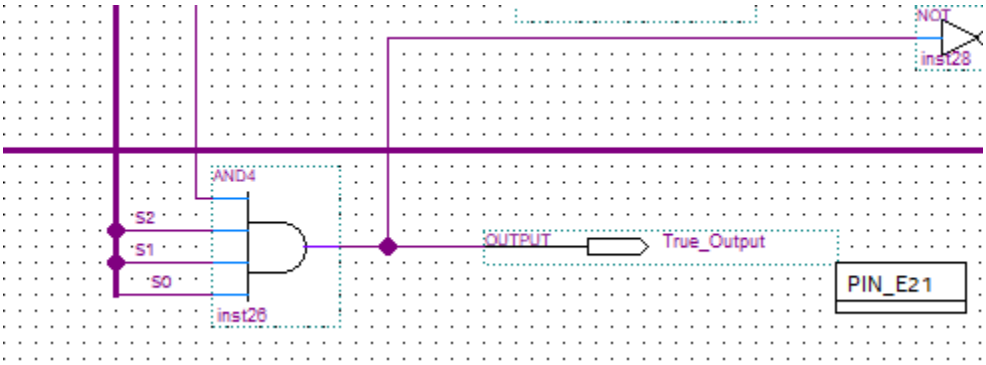


Above is the module to output the address of the first register that is smaller than its previous one. Input to the module is given by the output of bus_plusOne_adder3b module which is explained in Section 4 of Part A. The outputs are asserted to indicate the said address only when the "Validate/Enable" line is asserted and the comparator's output is **0** and the **True_Output** is **0**: **En** = "Validate/Enable" & **OUT** & (~**True_Output**) where **En** serves as the enable line for bus_output module. The said module's block diagram is shown below.



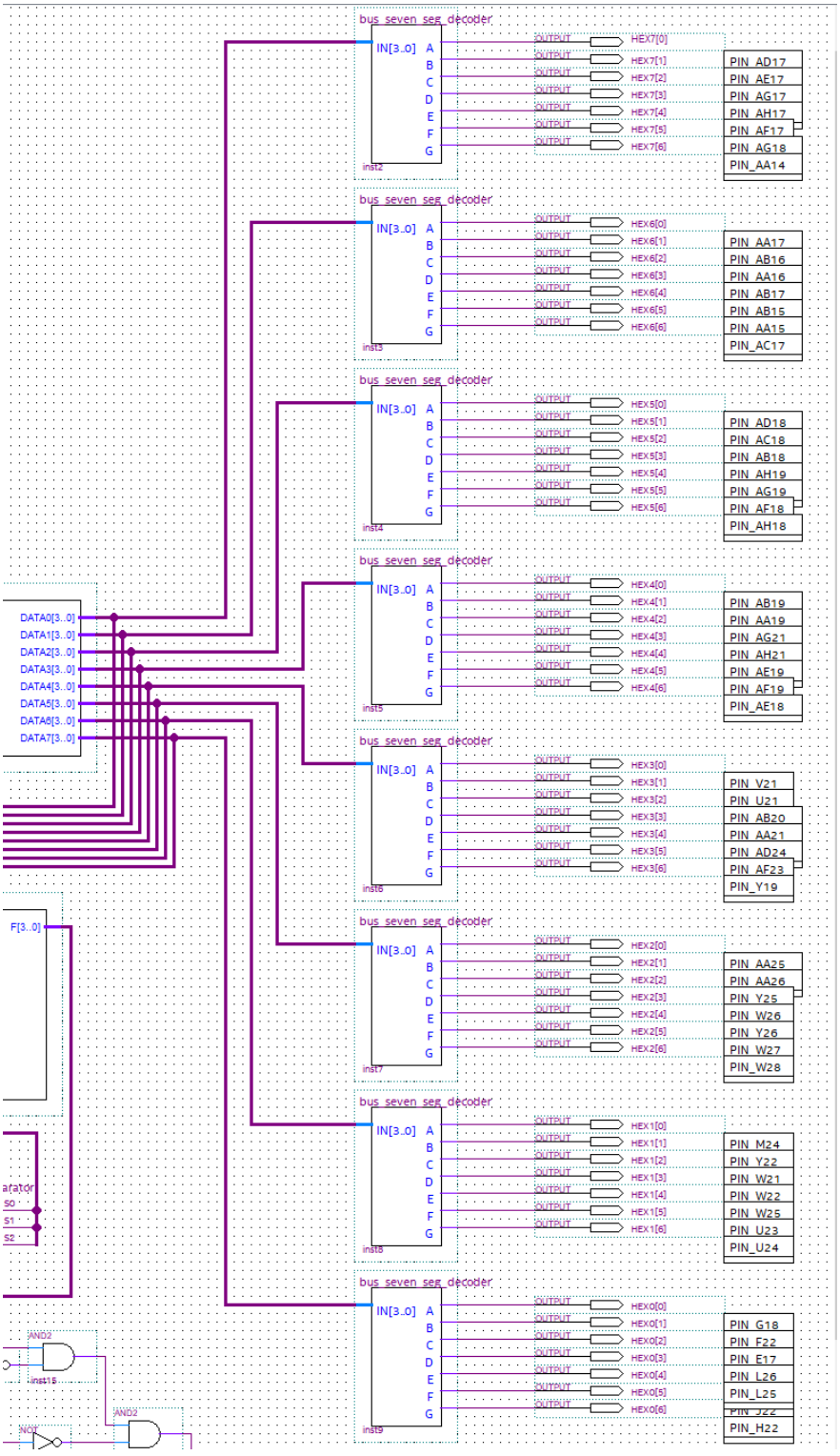
It simply takes **Input[2..0]** and outputs them after verifying the **En** line by taking **AND** of **En** and each bit if **Input**. Corresponding three outputs **F2, F1** and **F0** are asserted only when both **En** and **input2**, **input1** and **input0** are **1**, respectively.

The following is the output logic for **True_Output** that is asserted if the numbers in the registers are sorted.



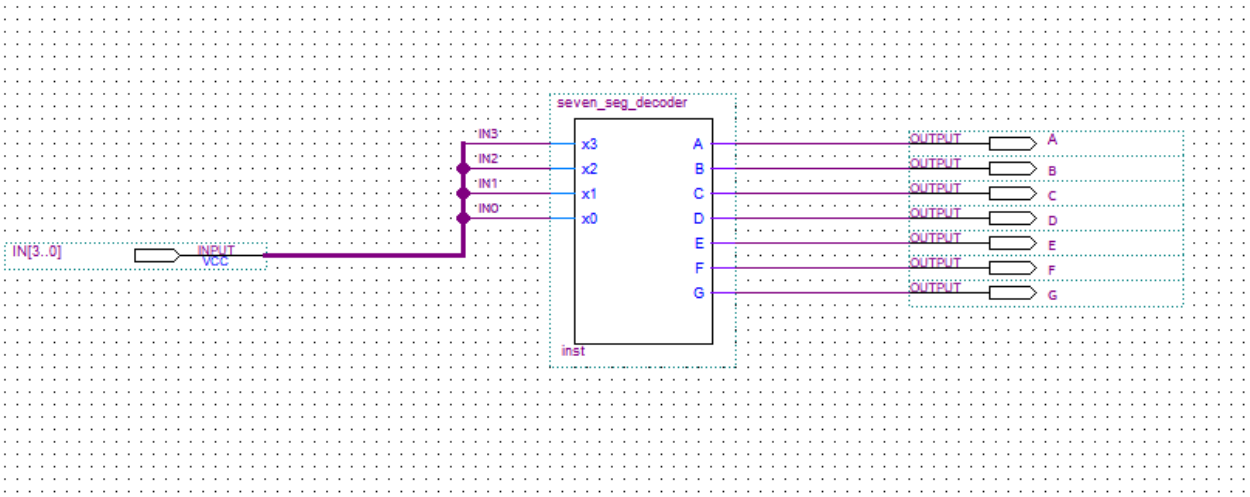
It is a simple 4-bit **AND** gate that takes in output of FSM **S[2..0]** and "Validate/Enable" line (to make sure that it is not asserted before **WE** and **Start** inputs are set to 1). They are **AND**-ed because output of FSM must be **S[2..0] = 111** for the **True_Output = 1**.

Part C



This part includes only 8 bus_seven_seg_decoder modules to output the numbers in 8 registers on 8 seven segment displays on the Altera Board.

Bus Seven Segment Decoders(bus_seven_seg_decoder)



As can be seen from the block diagram, bus_seven_seg_decoder module takes input **IN[3..0]** and assigns them to the corresponding inputs of seven_seg_decoder. And then takes the outputs **A, B, C, D, E, F** and **G** just like it's returned from the module to be displayed on the seven-segment display of the Altera Board.

Below is the Verilog code for seven_seg_decoder module that asserts **A, B, C, D, E, F** and **G** according to the given 4-bit number and its representation on seven-segment display.

```
1   module seven_seg_decoder(A, B, C, D, E, F, G, x3, x2, x1, x0);
2      input x3, x2, x1, x0;
3      output A, B, C, D, E, F, G;
4      reg A, B, C, D, E, F, G;
5
6      always@(x3, x2, x1, x0)
7         begin
8            case({x3, x2, x1, x0})
9
10              4'b0000: {A, B, C, D, E, F, G} = 7'b0000001;
11              4'b0001: {A, B, C, D, E, F, G} = 7'b1001111;
12              4'b0010: {A, B, C, D, E, F, G} = 7'b0010010;
13              4'b0011: {A, B, C, D, E, F, G} = 7'b0000110;
14
15              4'b0100: {A, B, C, D, E, F, G} = 7'b1001100;
16              4'b0101: {A, B, C, D, E, F, G} = 7'b0100100;
17              4'b0110: {A, B, C, D, E, F, G} = 7'b0100000;
18              4'b0111: {A, B, C, D, E, F, G} = 7'b0001111;
19
20
21
22              4'b1000: {A, B, C, D, E, F, G} = 7'b0000000;
23              4'b1001: {A, B, C, D, E, F, G} = 7'b0000100;
24              4'b1010: {A, B, C, D, E, F, G} = 7'b0001000;
25              4'b1011: {A, B, C, D, E, F, G} = 7'b1100000;
26
27              4'b1100: {A, B, C, D, E, F, G} = 7'b0110001;
28              4'b1101: {A, B, C, D, E, F, G} = 7'b1000010;
29              4'b1110: {A, B, C, D, E, F, G} = 7'b0110000;
30              4'b1111: {A, B, C, D, E, F, G} = 7'b0111000;
31
32           endcase
33        end
34
35  endmodule
```