Kai Heng Gan

Section 2

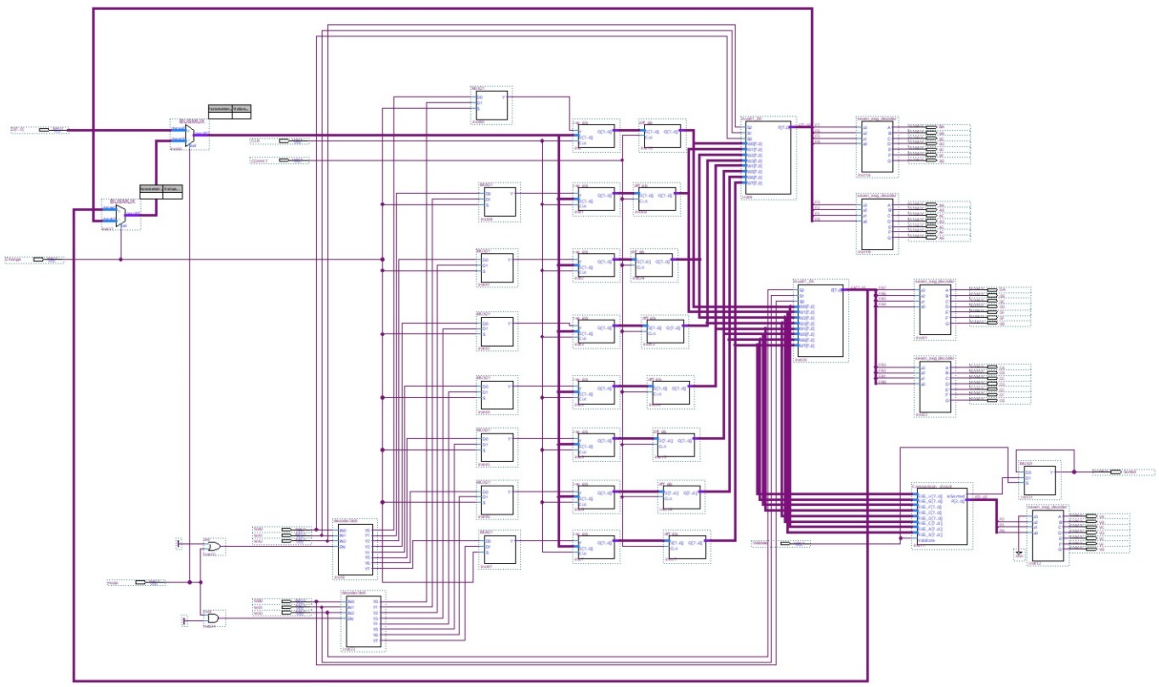
CPR E 281

Student ID: 787575850

Final Project Report

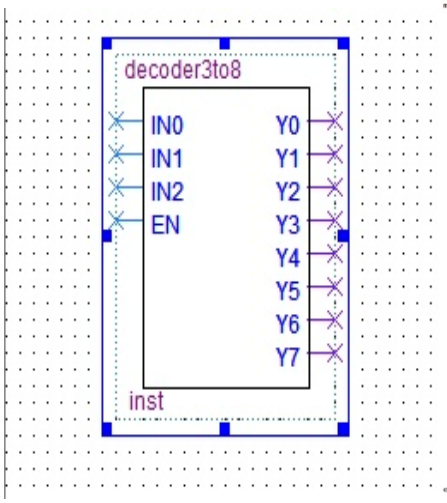
I completed project option number 1 (Manual Sorting Circuitry). This report details all the workings and derivations required to build the circuit and Verilog modules contained in the project. It contains a general overview of every top-level module, followed by a multilevel description of the inner workings of each module that explains every submodule involved.

Top Level Diagram

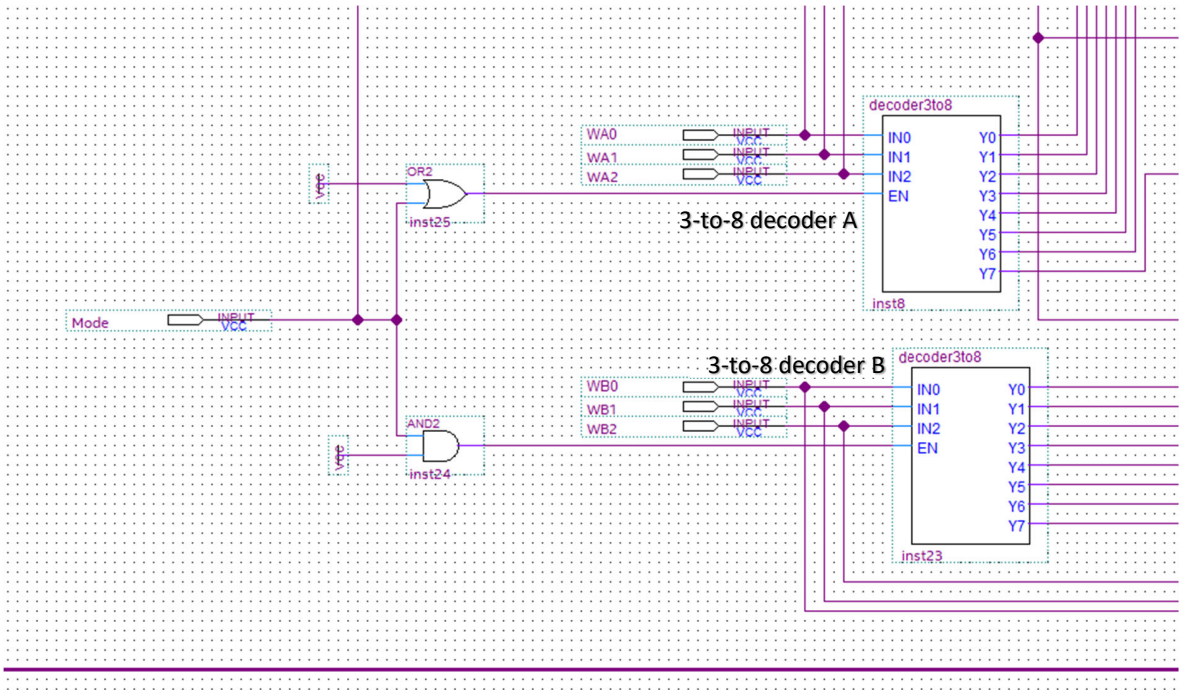


This is a complete Manual Sorting Circuitry. The circuit is hard to view from the image and it is complex. Therefore, I am going to briefly explains the circuit in this part. Then, I am going to split them into individual component and explain the logic of each of them. At first, the circuit contains a register file which have eight 8-bits registers. They are used to store the value that insert by the users. In order to store the value into the respective address of the 8-bit register, I use two 3-to-8 decoders. During INITIALIZATION mode, I am only using a 3-to-8 decoder which is the upper 3-to-8 decoder. During SORTING mode, I will then use two 3-to-8 decoders. And also, there are two 8-to-1 multiplexers which are use to output the value from the address of the register that selected by the user that stored in the D flip-flop. Then, there is a comparison circuit which used to check the order of the values in the eight registers. There are five seven segment decoders had been used in this circuit. They are used to display the output of the selected registers value and the position of the first register file that does not have its data sorted; that is, the first register that has a lower value than the preceding register. The seven segment decoders are very helpful for the user when the user are running SORTING mode.

3-to-8 Decoder

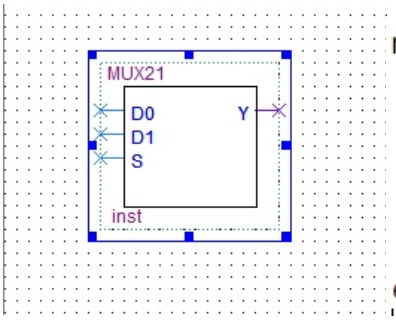



```
module decoder3to8(IN0,IN1,IN2,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,EN);
        input  IN0,IN1,IN2; //The three Input lines of the decoder

        input EN; //The decoder Enable input. High active.

        output reg Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;

    always @ (EN,IN2,IN1,IN0)

        begin

         Y0=0;
         Y1=0;
         Y2=0;
         Y3=0;
         Y4=0;
         Y5=0;
         Y6=0;
         Y7=0;

        if (EN==1'b1)
        begin
         case ({IN2,IN1,IN0})
                  3'b000: Y0=1;
                  3'b001: Y1=1;
                  3'b010: Y2=1;
                  3'b011: Y3=1;
                  3'b100: Y4=1;
                  3'b101: Y5=1;
                  3'b110: Y6=1;
                  3'b111: Y7=1;

        default:
                begin
                        Y0=0;
                        Y1=0;
                        Y2=0;
                        Y3=0;
                        Y4=0;
                        Y5=0;
                        Y6=0;
                        Y7=0;
                end
         endcase
        end
        end
endmodule
```



This decoder3to8 is a 3-to-8 decoder. And also, I have attach the Verilog code for the 3-to-8 decoder. The decoder has an Enable input, 3 inputs, and 8 outputs. The users will input a 3-bits number which is the address of the 8-bits register. The most significant bit will be input into IN2, the second bit will be input into IN1, and the least significant bit will be input into IN0. The Enable input which is used to activate the 3-to-8 decoder. When the user is running INITIALIZATION mode, the user only can use the 3-to-8 decoder A to assign the value to the user's desired 8-bits register and the 3-to-8 decoder B will be remains deactivate. The 3-to-8 decoder B will be activated only if the user is running SORTING mode. The outputs of the 3-to-8 decoder are "one-hot" encoded. For example, if the user enters 001 as the input, the output Y1 will output 1 and the rest of the outputs will output 0.

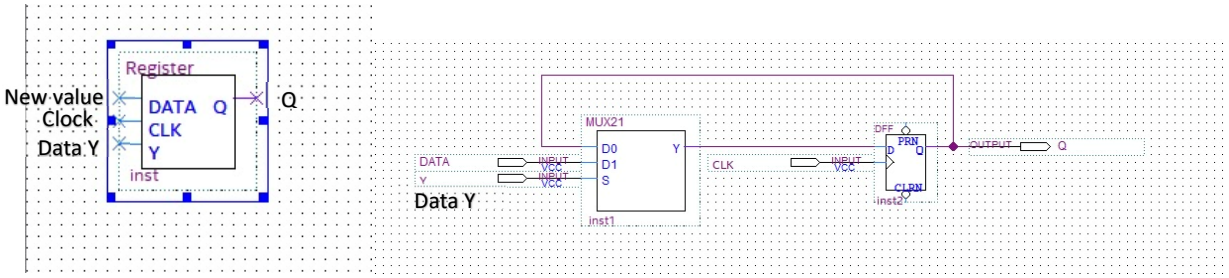
2-to-1 Multiplexer




```
module MUX21(D0, D1, S, Y);
        output Y;
        input D0, D1, S;

        assign Y=S?D1:D0;

endmodule
```

The images shown above are the symbol file and the Verilog code for 2-to-1 multiplexer. It contains two data inputs, a selector input, and an output. When the selector input is 0, the output Y will output the data from D0 input; when the selector input is 1, the output Y will output the data from D1 input. The meaning of the formula of the Y in the Verilog code is when S is true (1), Y equal D1; when S is false (0), Y equal D0.

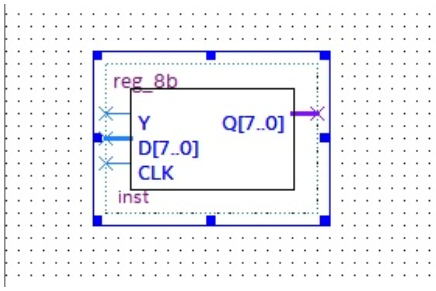
1-bit Register

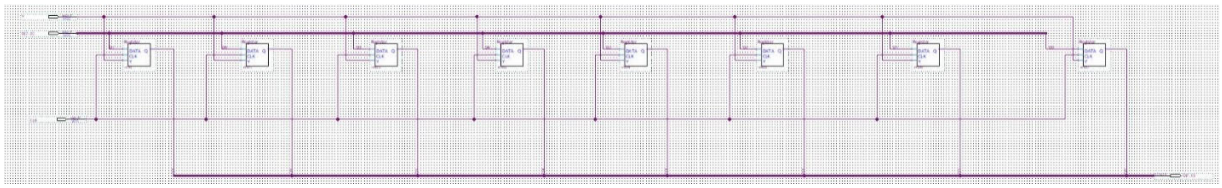


This is an 1-bit register which is used to store a 1-bit binary number. It contains a 2-to-1 multiplexer and a D flip-flop. According to the register symbol file, there are three inputs (DATA, CLK, and Y) and an output Q. The Data Y is the output from the 3-to-8 decoder. When the Data Y is 1, the output Y from MUX21 will output the D1 input (New value) that input by the user whilst the CLK (Clock) has to be 1, in order to store the value into the D flip-flop because the D flip-flop is a positive-edge D flip-flop. It only will load a new value when the clock has a positive edge. When Data Y is 0, the output Y from MUX21

will output the D0 input which is the output of the positive-edge D flip-flop. Hence, there is no new value will be stored into the D flip-flop.

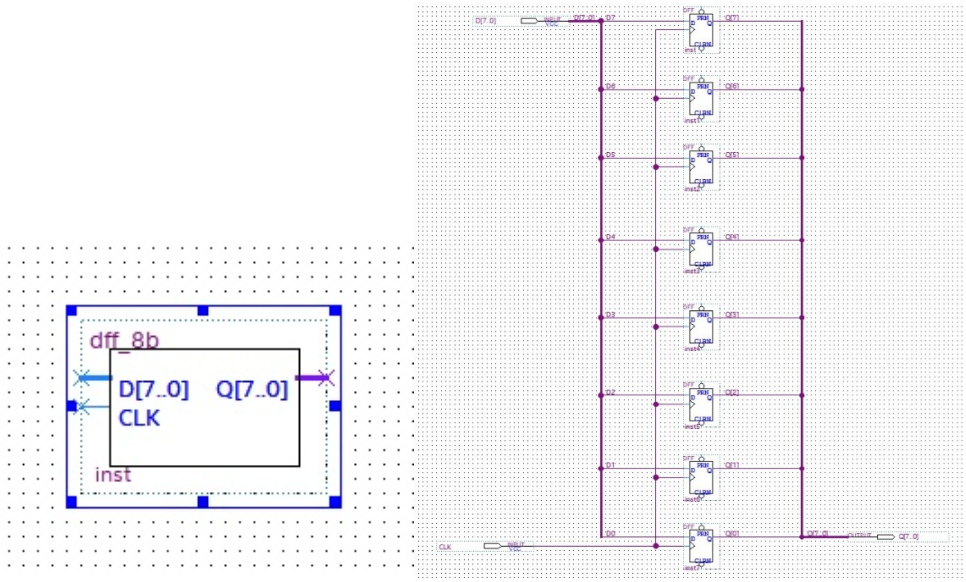
8-bits Register





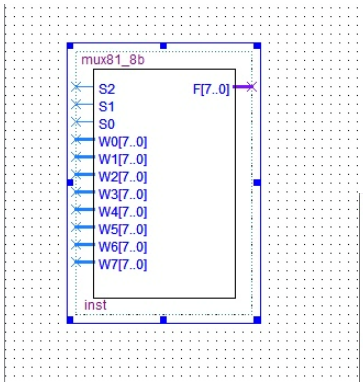
This is an 8-bits register which contains eight 1-bit registers. According to the 8-bits register symbol file, it also has three inputs (Y, D[7..0] and CLK) and an output Q[7..0]. The differences between the 1-bit register and 8-bits register are the user allows to enter an 8-bits value into the 8-bits register. And also, the 8-bits register will out an 8-bits value. Other than that, the logic of the 8-bits register works same as the 1-bit register.

8-bits D Flip-Flop



This is an 8-bits D flip-flop. Instead of using a 1-bit D flip-flop to store a 1-bit value, I created an 8-bits D flip-flop to load, store and output an 8-bits value that the user entered into an 8-bits register. The concept of the 8-bits D flip-flop is just the same as the 1-bit D flip-flop.

8-to-1 Multiplexer




```
module mux81_8b (S2,S1,S0,W0,W1,W2,W3,W4,W5,W6,W7,F);
        input [7:0] W0,W1,W2,W3,W4,W5,W6,W7;
        input S2,S1,S0;
        output [7:0] F;

        assign F = S2 ? (S1 ? (S0?W7:W6):(S0?W5:W4)):(S1 ? (S0?W3:W2):(S0?W1:W0));
endmodule
```

Part A Part B

Part AA Part AB Part BA Part BB

This is an 8-to-1 multiplexer which consist three selector inputs, eight 8-bits data inputs, and an 8-bits data output. It is the same concept as 2-to-1 multiplexer. The differences between 2-to-1 multiplexer and 8-to-1 multiplexer are 8-to-1 multiplexer allows the user to input 3-bits value for the selector, which

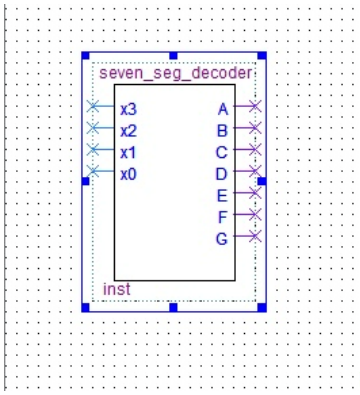
allow the user to output desired location's data from the eight 8-bits data inputs. The most significant bit of the 3-bits value that will be input for the selector will be input into S2 by the user, then the second bit of the 3-bits value will be input into S1 by the user, and the least significant bit of the 3-bits value will be input into S0 by the user.

I used the Verilog code to create the 8-to-1 multiplexer. I am going to explain the equation that I used to assign the output F value. S2 is the most significant bit of the 3-bits value for the selector, S1 is the second bit of the 3-bits value for the selector, and S1 is the least significant bit of the 3-bits value for the selector. First, if S2 is true (1), the system will proceed to Part A; if S2 is false (0), the system will proceed to Part B. In Part A, when S1 is 1, the system will proceed to Part AA; when S1 is 0, the system will proceed to Part AB. In Part AA, when S0 is 1, the system will choose W7 as the value of output F; when S0 is 0, the system will choose W6 as the value of output F. In Part AB, when S0 is 1, the system will choose W5 as the value of output F; when S0 is 0, the system will choose W4 as the value of output F. In Part B, when S1 is 1, the system will proceed to Part BA; when S1 is 0, the system will proceed to Part BB. In Part BA, when S0 is 1, the system will choose W3 as the value of output F; when S0 is 0, the system will choose W2 as the value of output F. In Part BB, when S0 is 1, the system will choose W1 as the value of output F; when S0 is 0, the system will choose W0 as the value of output F.

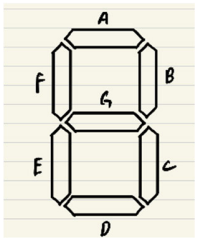
S2	S1	S0	F
0	0	0	W0
0	0	1	W1
0	1	0	W2
0	1	1	W3
1	0	0	W4
1	0	1	W5

1	1	0	W6
1	1	1	W7

Seven Segment Decoder



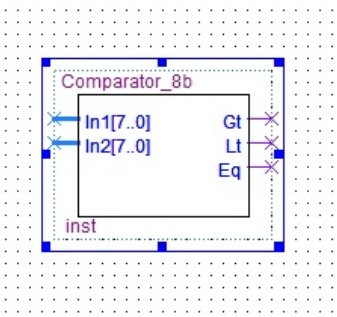

```
module seven_seg_decoder(x3,x2,x1,x0,A,B,C,D,E,F,G);
        input x3,x2,x1,x0;
        output A,B,C,D,E,F,G;
        reg A,B,C,D,E,F,G;

        always@(x3 or x2 or x1 or x0)
        begin
                case({x3,x2,x1,x0})
                        4'h0: {A,B,C,D,E,F,G} = 7'b0000001;
                        4'h1: {A,B,C,D,E,F,G} = 7'b1001111;
                        4'h2: {A,B,C,D,E,F,G} = 7'b0010010;
                        4'h3: {A,B,C,D,E,F,G} = 7'b0000110;
                        4'h4: {A,B,C,D,E,F,G} = 7'b1001100;
                        4'h5: {A,B,C,D,E,F,G} = 7'b0100100;
                        4'h6: {A,B,C,D,E,F,G} = 7'b0100000;
                        4'h7: {A,B,C,D,E,F,G} = 7'b0001111;
                        4'h8: {A,B,C,D,E,F,G} = 7'b0000000;
                        4'h9: {A,B,C,D,E,F,G} = 7'b0000100;
                        4'hA: {A,B,C,D,E,F,G} = 7'b0001000;
                        4'hB: {A,B,C,D,E,F,G} = 7'b1100000;
                        4'hC: {A,B,C,D,E,F,G} = 7'b0110001;
                        4'hD: {A,B,C,D,E,F,G} = 7'b1000010;
                        4'hE: {A,B,C,D,E,F,G} = 7'b0110000;
                        4'hF: {A,B,C,D,E,F,G} = 7'b0111000;
                endcase
        end
endmodule
```



This is a seven segment decoder. I used Verilog code to create this module. It consists four inputs (x3, x2, x1, and x0) and seven outputs (A, B, C, D, E, F, G). The four inputs allow the user to input 4-bits value. I used switch-case statement to assign the value of the seven outputs. In the switch-case statement, instead of writing 4'b0000 for the expression for case 0, I assign the 'b0000' to 'h0'. It makes me easily to track which case is displaying its respecting value. The seven segment display LEDs will only be triggered when its respecting input is 0. For example, when the selector input is 0, the value of A = 0, B = 0, C = 0, D = 0, E = 0, F = 0, G = 1. All the LEDs in seven segment display will be triggered, except the LED bar G. I have created a seven segment display truth table. It will easier for others to comprehend the concept of seven segment decoder and seven segment display through the table below.

x3 x2 x1 x0	A	B	C	D	E	F	G	DISPLAY
0000	0	0	0	0	0	0	1	0
0001	1	0	0	1	1	1	1	1
0010	0	0	1	0	0	1	0	2
0011	0	0	0	0	1	1	0	3
0100	1	0	0	1	1	0	0	4
0101	0	1	0	0	1	0	0	5
0110	0	1	0	0	0	0	0	6
0111	0	0	0	1	1	1	1	7
1000	0	0	0	0	0	0	0	8
1001	0	0	0	0	1	0	0	9
1010	0	0	0	1	0	0	0	A
1011	1	1	0	0	0	0	0	b
1100	0	1	1	0	0	0	1	C
1101	1	0	0	0	0	1	0	d
1110	0	1	1	0	0	0	0	E
1111	0	1	1	1	0	0	0	F

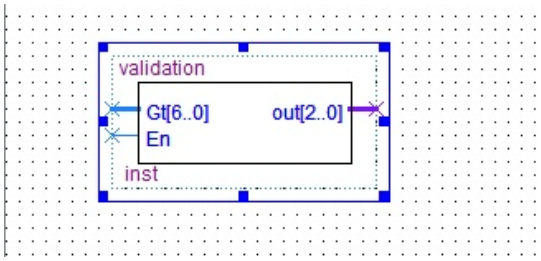
8-bits Comparator




```
module Comparator_8b (In1,In2,Gt,Lt,Eq);
        input [7:0]  In1,In2; //The two 8-bit Inputs In1 and In2
        output reg Gt,Lt,Eq; //The Outputs of comparison

        always @ (In1 or In2) //Check the state of the input lines
        begin
                Gt <= ( In1 > In2 )? 1'b1 : 1'b0;
                Lt <= ( In1 < In2 )? 1'b1 : 1'b0;
                Eq <= ( In1 == In2)? 1'b1 : 1'b0;
        end
endmodule
```

This is an 8-bits comparator. I used Verilog code to construct this module. It has two input lines (In1 and In2) which allow user to input 8-bits values. And also, it has three output lines (Gt, Lt, and Eq). The output Gt will output 1 if In1 input's value is larger than In2 input's value which mean the expression: In1 > In2 is true. Otherwise, the output Gt will output 0. The output Lt will output 1 if In1 input's value is smaller than In2 input's value which mean the expression: In1 < In2 is true. Otherwise, the output Lt will output 0. The output Eq will output 1 if In1 input's value and In2 input's value are equal which mean the expression: In1 == In2 is true. Else, the output Eq will output 0.

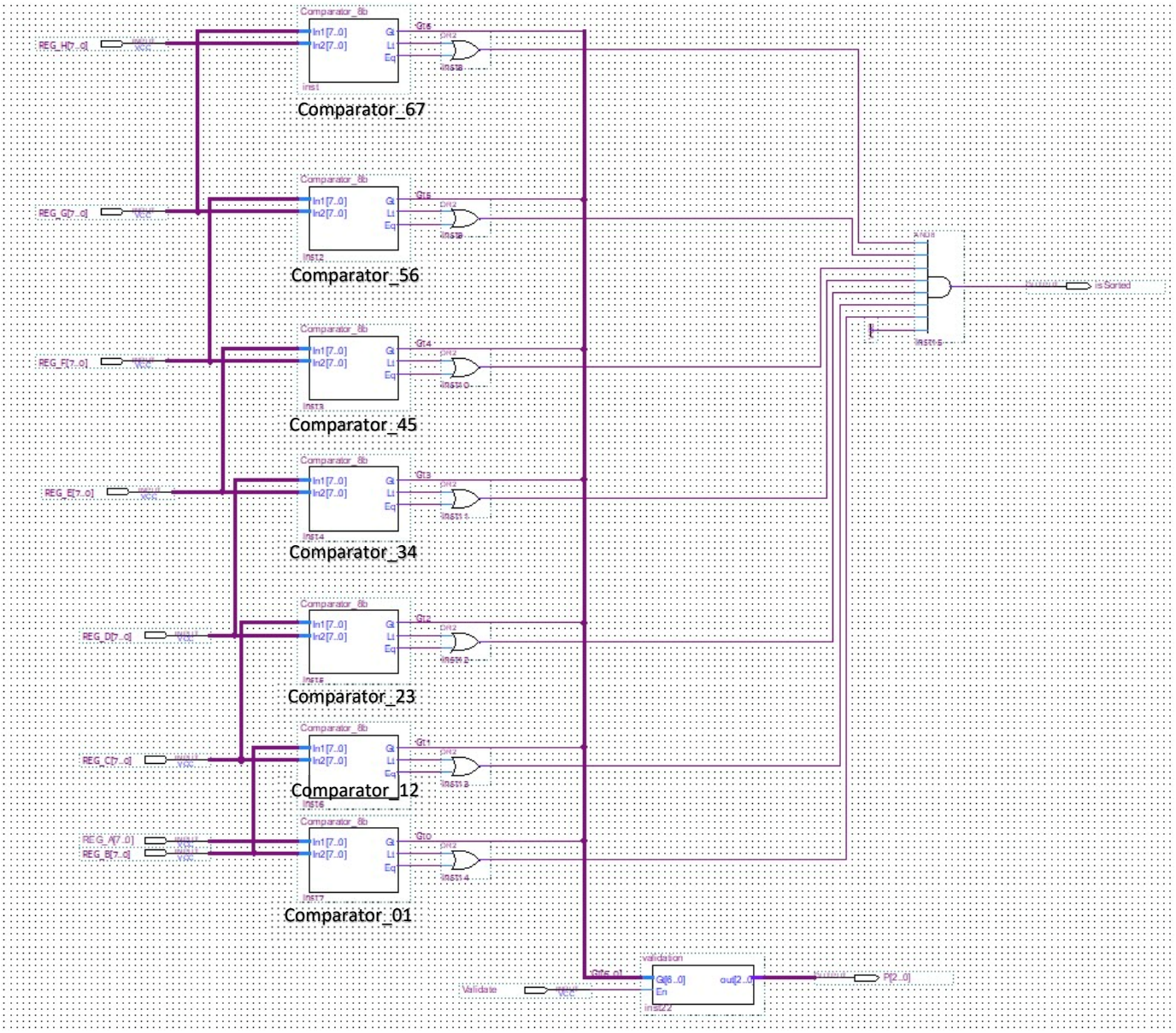
Validation




```
module validation(Gt, En, out);
        input En;
        input [6:0] Gt;
        output reg [2:0] out;

        always@(En)
        if(En)
        begin   Expression A
                if(Gt[0])
                        out = 3'b001;   Statement A
                Expression B
                else if(Gt[1])
                        out = 3'b010;   Statement B
                Expression C
                else if(Gt[2])
                        out = 3'b011;   Statement C
                Expression D
                else if(Gt[3])
                        out = 3'b100;   Statement D
                Expression E
                else if(Gt[4])
                        out = 3'b101;   Statement E
                Expression F
                else if(Gt[5])
                        out = 3'b110;   Statement F
                Expression G
                else if(Gt[6])
                        out = 3'b111;   Statement G
                Default
                else
                        out = 3'b000;   Statement H
        end
endmodule
```

This is a validation module. I created it with Verilog code. It has an enable input line (En), a 7-bits data input line (Gt), and a 3-bits data output line (out). The module will only be activated if the En input is 1. I used if-else statement to assign the 3-bits out output value. Gt[0] is the least significant bit of 7-bits data. Gt[6] is the most significant bit of 7-bits data. The order of significant bit of Gt[1], Gt[2], Gt[3], Gt[4], Gt[5] is in descending order. For example, the Gt[5] is the second significant bit of the 7-bits data. If the 7-bits data has been input into Gt and the En input is 1, the program will first check the least significant bit (Gt[0]) of the 7-bits data. If the least significant bit if the 7-bits data is 1, the program will assign out equal to 3'b001 which means the out = 1. If Gt[0] is 0, the Expression A is false because 0 in coding represent false. If Expression A is not satisfied, the system will ignore Statement A. Then, the system will proceed to Expression B. If Expression B is satisfied which means Gt[1] is 1, the system will do the Statement B, the program will assign out equal to 3'b010 which means the out = 2. The system will proceed to the next expression until it found an expression is satisfied if the previous expressions are not satisfied. If all the expressions are not satisfied, the system will do the Statement H.

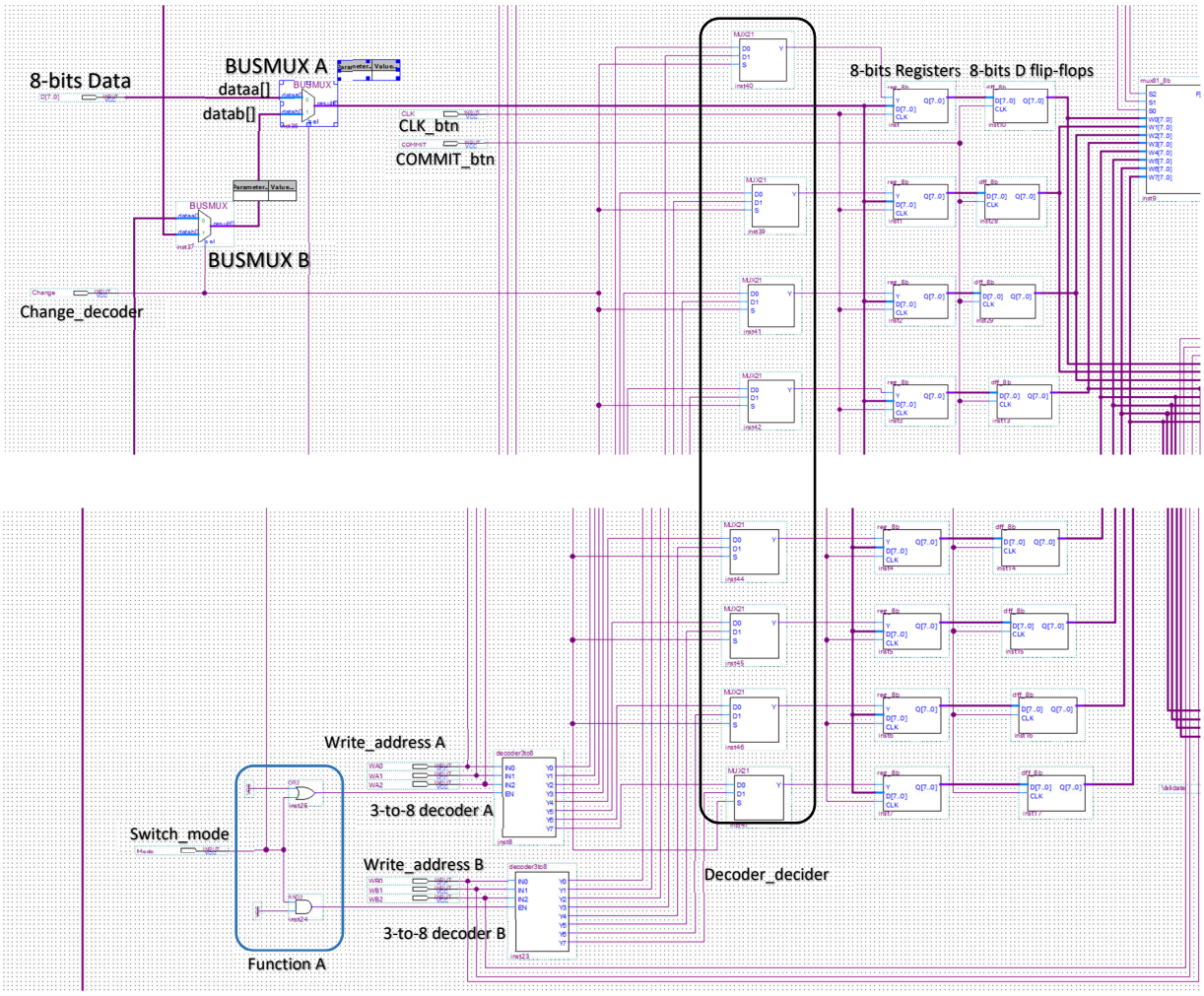
Comparison Circuit



This a comparison circuit. I used seven 8-bits comparators, seven two-input line OR gates, a validation, and an eight-input line AND gate. I have defined the variable name for each comparator in the circuit. For the Comparator_01, it compares the values between 8-bits register A and 8-bits register B. In other word, it compares the value between the first register's stored value and the second register's stored value. For the Comparator_12, it compares the value between the second register's stored value and

the third register's stored value. The rest of the comparators do the same things as Comparator_01 and Comparator_12 with their respective registers. The OR gates will output 1 as long as one of the output of Lt and Eq is 1. If all of the OR gates output 1, which means the data are sorted in an ascending order because the AND gate will output 1. If the values between the two 8-bit registers are the same, it still considers as in ascending order as long as the current register's stored value is not lower than the preceding register's stored value. If all of the Gt output 1 and the Validate input is 1, the validation will only output the position of the first register file that does not have its data sorted, which mean the position of the second register file position (3'b001) will be output.

INITIALIZATION Mode

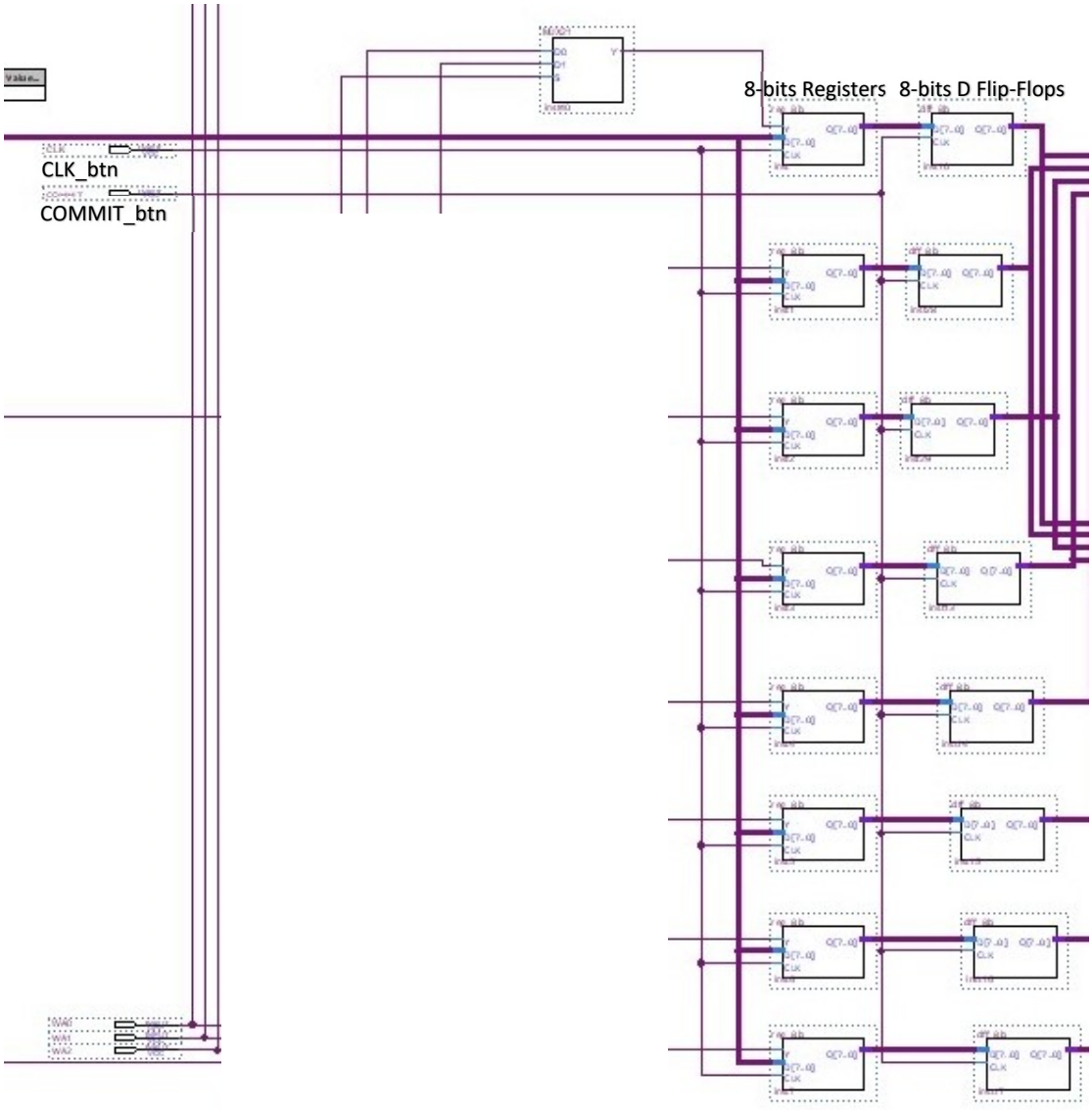


Now, I am going to explain the concept of INITIALIZATION mode. I use a rocker switch to control the Switch_mode. When the Switch_mode is 0, the circuit will run INITIALIZATION mode; when the Switch_mode is 1, the circuit will run SORTING mode. The Switch_mode will start with 0 because the user to initialize all the switches on the DE2-115 into 0. Therefore, when the user run the program, the circuit will automatically start with INITIALIZATION mode. There is a Function A. It is used to activate the 3-to-8 decoders. I use an OR gate for the 3-to-8 decoder A's Enable input. The VCC symbol represents 1, no matter the input of Switch_mode is 0 or 1, the output of the OR gate will always be 1. Therefore, the 3-to-8 decoder A will always be activated. For 3-to-8 decoder B, I use an AND gate for its Enable input.

Because of the circuit is running INITIALATION mode, the Switch_mode is 0, the 3-to-8 decoder B will not be activated. The 3-to-8 decoder B will only be activated when the Switch_mode is 1 because the AND gate will only output 1 when its two inputs are 1. Besides that, there is an important switch called Change_decoder. The Change_decoder has to be initialize to 0 in order to let the Decoder_decider outputs the outputs from 3-to-8 decoder A. If the Change_decoder is 1, the Decoder_decider will output the outputs from 3-to-8 decoder B but the 3-to-8 decoder B was not activated and the outputs of 3-to-8 decoder B are 0. It will not trigger any one of the 8-bits register.

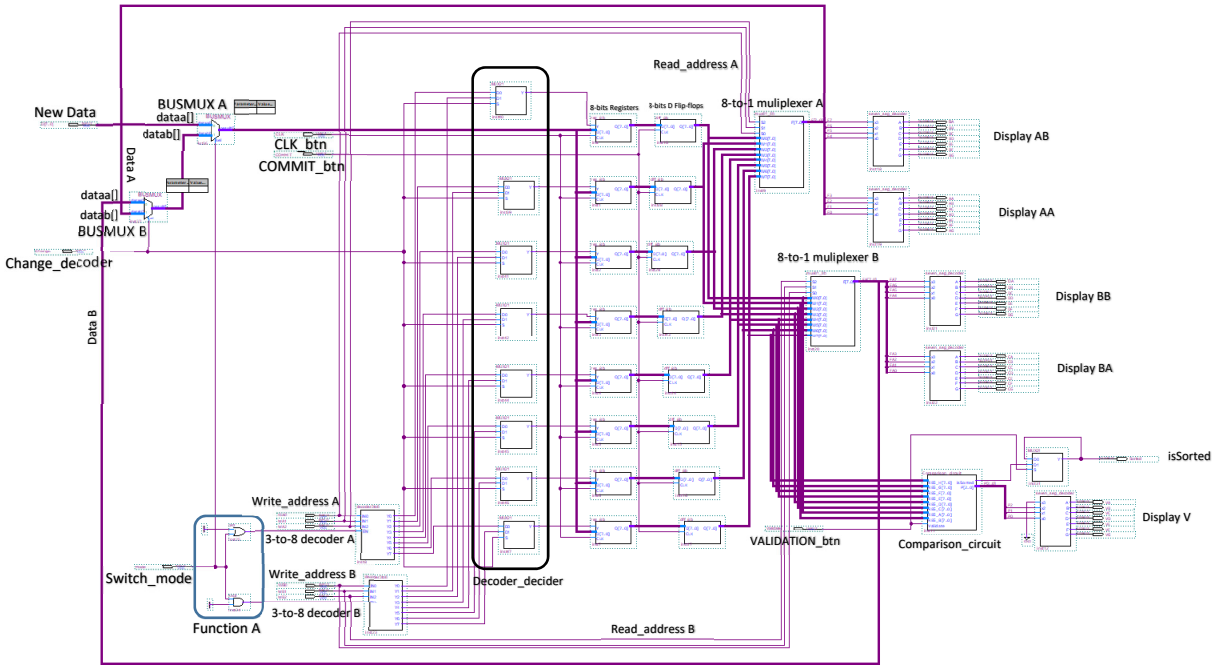
I use two bus multiplexers, BUSMUX A and BUSMUX B. Both of them are with WIDTH value: 8 because the user will input 8-bits values into the 8-bits registers. The BUSMUX B will only be fully use in SORTING mode because the Switch_mode is 0 (INITIALIZATION mode) and the selector of BUSMUX A is 0. The BUSMUX A will only output the data from the dataa[] input which is the 8-bits Data that entered by the user. Because of the INITIALIZATION mode and only the output of 3-to-8 decoder A trigger one of the 8-bits register, the user only has to enter the address of the 8-bits register into the Write_address A input. And also, the user has to press CLK_btn when the user entered the address of the 8-bits register and the 8-bits Data every time because the 8-bits registers will only be triggered when the clock has positive edge. Once the user done with entering 8-bits values into the 8-bits registers, the user has to press COMMIT_btn to trigger the clock in the 8-bits D flip-flops to store the values from the 8-bits registers.

Storing 8-bits Values from 8-bits Registers into 8-bits D Flip-Flops



In this part, I am going to explain how the user can store the values from the 8-bits registers into the 8-bits D flip-flops. Once the user stored the values into all the 8-bits registers, the user has to press COMMIT button in order to store the values from the 8-bits registers into the 8-bits D flip-flops. The purpose of creating eight 8-bits D flip-flops is to allow the user to swap the value between two registers. The 8-bits D flip-flops play an important role in sorting the order of the values that stored in the eight 8-bits registers.

SORTING Mode



In SORTING mode, the input of the Switch_mode is 1. Therefore, the AND gate in Function A will output 1 and the 3-to-8 decoder B will be activated. In the meantime, the New Data will not be load into the circuit because the selector of the BUSMUX A is 1 and it only will output the output of BUSMUX B. Now, two 3-to-8 decoders are validated and I use two 8-to-1 multiplexers to output the respective register's value. I defined the two data that output from the two 8-to-1 multiplexers as Data A and Data B. The Data A will input into BUSMUX B datab[] input line. The Data B will input into BUSMUX B dataa[] input line.

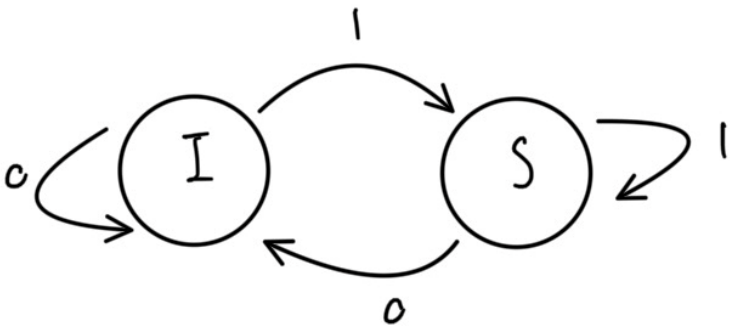
The user will input two values; the values will both be the addresses. Then, each 8-to-1 multiplexers will output an 8-bits data. Because of the output is 8-bits data, I need two seven segment decoder for each 8-to-1 multiplexer. Each seven segment decoder will connect to a seven segment display. It helps the user to observe the value in the respective register. And now, Data A and Data B, each of them is handling an 8-bits data which is the value from its respective register. The Change_decoder can be

either initialize to 0 or 1 because the two 3-to-8 decoders are working. The user can either change the value of the address A register to the value of the address B register first or change the value of the address B register to the value of the address A register first due to the Change_decoder and the Decoder_decider. If the user starts with the input of Change_decoder is 0, the circuit will open a road for the Data B to the address A 8-bits register. The user has to press the CLK_btn in order to input the Data B into the address A 8-bits register. Once the user pressed the CLK_btn, the data in the address A 8-bits register will change to Data B. But, the data in address A 8-bits D flip-flop will not change until the user pressed the COMMIT_btn. Following on, the user will change the input of Change_decoder to 1, the circuit will open a road for the Data A to the address B 8-bits register. The user has to press the CLK_byn in order to input the Data A into the address B 8-bits register. Once the user pressed the CLK_btn, the data in the address B 8-bits register will change to Data A. Now, the user has to press the COMMIT_btn in order to store the latest data into the 8-bits D flip-flops.

The user allows to check whether the data in the register file has been sorted or not anytime by pressing the VALIDATION_btn. If the data in the first register is larger than the data in the second register, the Comparison_circuit will output the position of the second register. Then, there is a seven segment decoder which connected with the Comparison_circuit output P[2..0]. It will then display the position to a seven segment display. If the data in the second register is larger than the data in the third register, and the data in the third register is larger than the data in the fourth register, the Comparison_circuit will out the position of the third register because it has been set to output the first register that has a lower value than the preceding register in the validation module. Once the user sorted the data in the register file, the user can press the VALIDATION_btn to check whether the data in the register file is completely sorted. If the data in the register file is completely sorted, the isSorted will output 1. The isSorted is let user know whether the register file has been completely sorted or not. The isSorted output will connect to a LED. If the data in the register file is completely sorted, the isSorted will output

1. The LED will be triggered to light. If the data in the register file is not sorted, the isSorted will output 0 and the connected LED will not be triggered.

State Diagram



Present state	Next state		output
	w=0	w=1	
I	I	S	0
S	S	I	0

In the state diagram and the state table, the I represents the INITIALIZATION mode. The S represents the SORTING mode. The w in the state table represents the Switch_mode. There is no output when the user switches the mode. The user allows to switch the mode anytime as the user want to by changing the value of Switch_mode. The user allows to switch from SORTING mode to INITIALIZATION mode if the user want to assign a new data into the register.