

Phase2

+

Improvement Report

Team members:

1: Ramy Mohsen ID: 201900961

2: Mohamed Helmy ID: 201900859

3: Youssef Mahmoud ID: 201901093

Project design

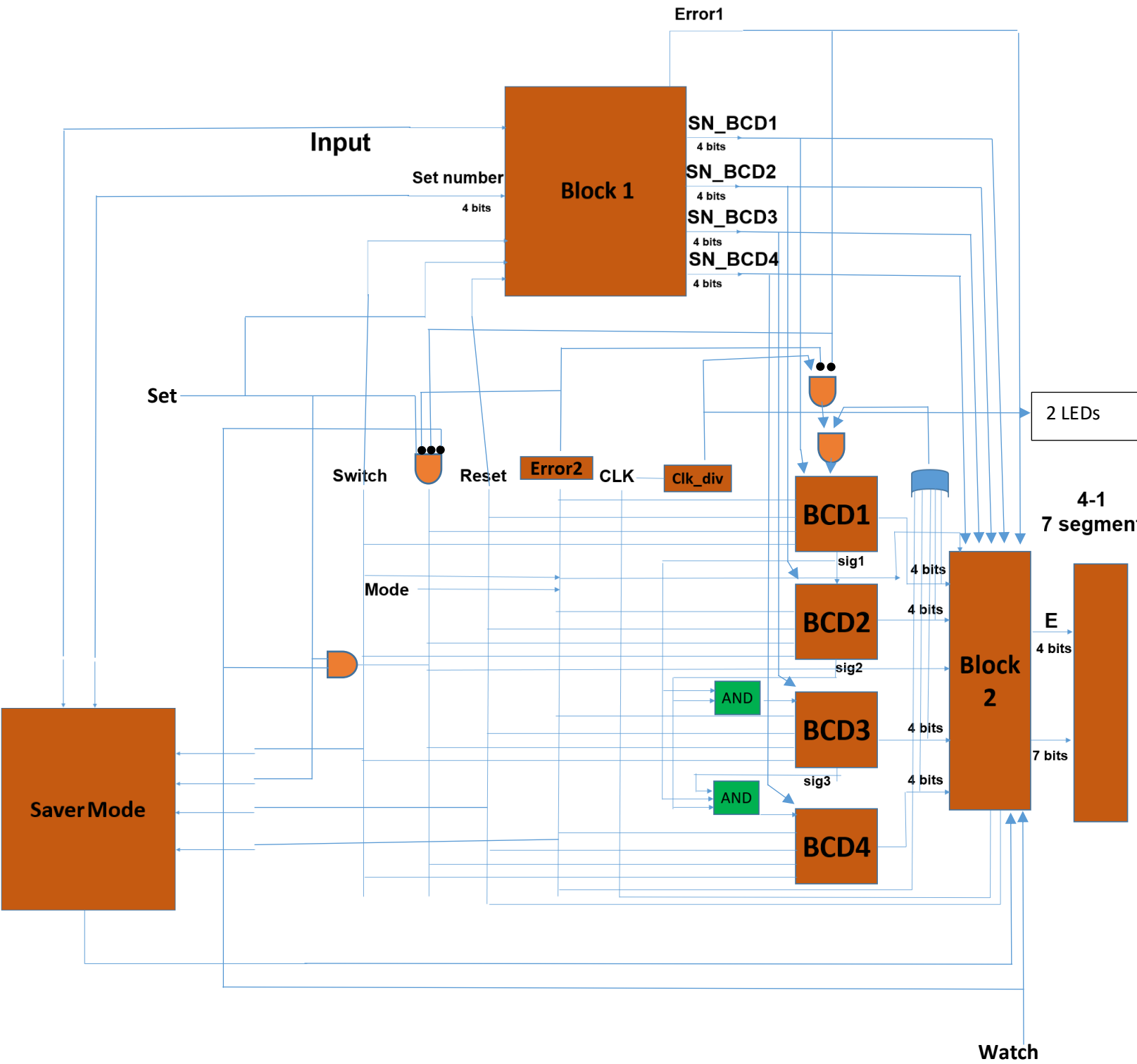


Figure 1

Saver Mode

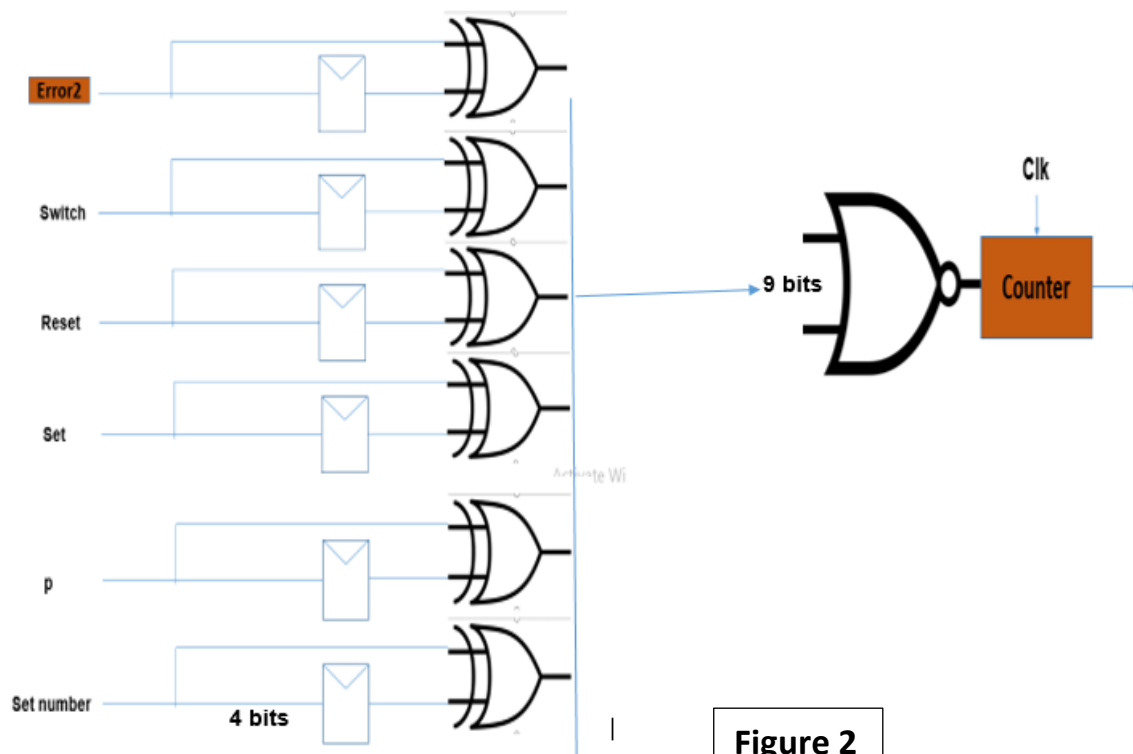


Figure 2

BCD

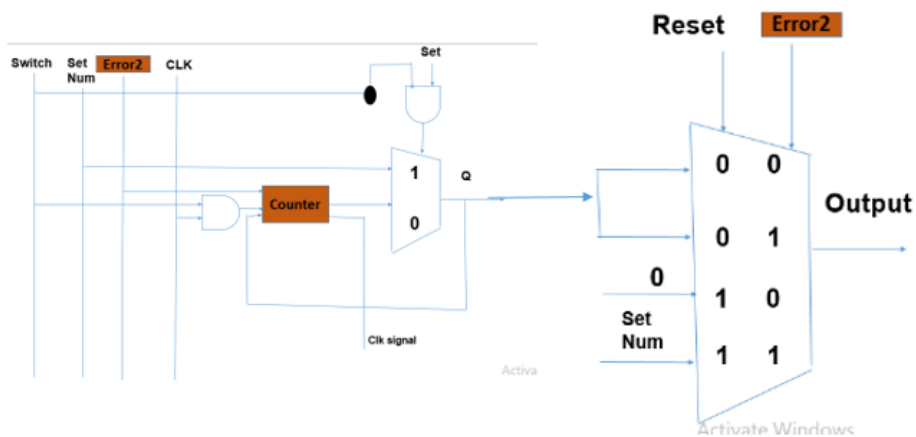


Figure 3

Team Member: Mohamed Helmy

Progress description:

The time for the work was about **30 hours** the **4 BCD blocks** took most of this time because of the challenges that faced me while connecting them together.

I met some **difficulties** to understand how **always_ff** work in the modelsim using system Verilog ,so it was a challenge to solve the impedance problem while connecting the blocks together, so **negege switch added** – will illustrated further below-.

At first the difficulties was also in **changing the counters from behavioral to structural** as it was increment the out put each time not the input.

BCD Block : As seen in the drawing block **(figure 3)**

The first part (The counter)

The inputs:

- 1- **Mode**
 - a. If the mode is 1 in the counter will begin to count up and if 0 the counter will begin to count down.
- 2- **Switch and clock**
 - a. If the clock is on and the switch is on the counter will count up or down if not the counter will stop and the last value will displayed
- 3- **The output from the Mux_R** will be discussed in the part after that.

The **output** of the counter is a clock signal and **4 bit output**.

The second part in the Block is (Mux_C)

The inputs:

- 1- **4 bits set number**
- 2- **4 bits that outed from the counter**

The condition:

- 1- Switch inverted and with set means that **only when the switch off the user can set a value**

Third part in the Block is (Mux_R)

The input:

- 1- **The 4 bits out from the Mux_C**
- 2- **The 0**
- 3- **The 4 bits set number entered by the user**

The condition:

- 1- Reset
- 2- Mode

If reset 1 and the mode 1 (timer), the value is 0 and if the reset 1 and mode 0 (stopwatch) the output is the set number value Other than that the output equal the value outed from the Mux_C

This output value from the Mux_R will enter the counter as mentioned before.

--- This block is repeated for the 4 other BCDs but the clock signal for each one will get it from the previous BCD of it.

The code for the counters: counter_9 AND counter_5

- For counter_9

```
1 module Counter_9 (switch,mode,clk,in,out,clk2);
2   input logic switch,mode,clk;
3   input logic [3:0] in;
4   output logic [3:0] out;
5   output logic clk2;
6
7   always_ff @(posedge clk)
8   begin
9     if(switch)
10    begin
11      if(mode) //timer
12      begin
13
14        if(in < 10)
15        begin
16          clk2 <= 0;
17          out <= in + 1;
18          if(in == 9)
19          begin
20            out <= 0;
21            clk2 <= 1;
22          end
23        end
24      end
25    end
26  else //stopwatch
27  begin
28
29    if(in + 1 > 0)
30    begin
31      clk2 <= 0;
32      out <= in - 1;
33      if(in == 0)
34      begin
35        out <= 9;
36        clk2 <= 1;
37      end
38    end
39  end
40
41  //end
42
43  end // endelse
44  end
45  else
46  begin
47    out<=in;
48  end
49  end //end always
50  endmodule
51
```

- For counter_5

```
1 module Counter_5 (switch,mode,clk,in,out,clk2);
2   input logic switch,mode,clk;
3   input logic [3:0] in;
4   output logic [3:0] out;
5   output logic clk2;
6   |
7   always_ff @ (posedge clk)
8   begin
9     if(switch )
10    begin
11      if(mode) //timer
12      begin
13
14        if(in < 6)
15        begin
16          clk2 <= 0;
17          out <= in + 1;
18          if(in == 5)
19          begin
20            out <= 0;
21            clk2 <= 1;
22          end
23        end
24      end
25    end
26    else //stopwatch
27    begin
28
29      if(in + 1 > 0)
30      begin
31        clk2 <= 0;
32        out <= in - 1;
33        if(in == 0)
34        begin
35          out <= 5;
36          clk2 <= 1;
37        end
38      end
39    end
40  end // endelse
41  end
42  else
43  begin
44    out <= in;
45  end
46  end //end always
47 endmodule
48
49 |
```

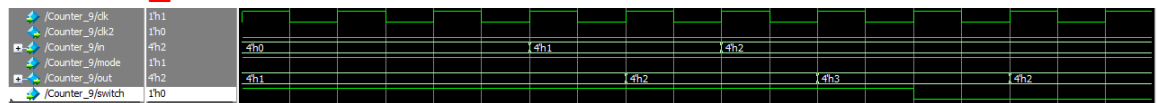
Edited: The only edit to this code is to add negative edge of the (switch) to be as a trigger for the counters in the ModelSim (system Verilog) to avoid the impedance that happens as the only trigger for them is the clock. If the is still impedance as it is generated from the previous clock the values will always be impedance.

The modification is as follow:

```
always_ff @(negedge switch, posedge clk )  
begin
```

The simulation for Them:

- Counter_9



- Counter_5 : The same as counter_9 except the last value is 5 instead of 9

The Mux_R

- Code:

```
1 module Mux_R (reset, Mode, setNum,in, out);  
2  
3 input logic reset,Mode;  
4 input logic [3:0] setNum;  
5 input logic [3:0] in;  
6 output logic [3:0] out;  
7 always_comb  
8 begin  
9     if((reset == 0 && Mode == 0) || (reset == 0 && Mode == 1))  
10        begin // if 00 || 01 reset & Mode  
11            out = in ;  
12        end //end of if reset & Mode  
13  
14        else if (reset == 1 && Mode == 0) // 10 reset & Mode in stopwatch mode  
15            begin  
16                out = setNum;  
17            end // end of stopwatch mode reset  
18  
19            else // timer mode reset  
20                begin  
21                    out = 0;  
22                end // end of timer mode  
23        end // end of the always statment  
24    end  
25 endmodule  
26
```

- **Simulation:**

Signal	Width	Value
/MUX_R/Mode	1h0	0
/MUX_R/in	4h1	0
/MUX_R/out	4h1	0
/MUX_R/reset	1h0	0
/MUX_R/setNum	4h5	0

Mux_counter:

- **Code:**

```

1
2 module Mux_Counter (set, switch, setNum, in , out);
3
4 input logic set,switch;
5 input logic [3:0] setNum;
6 input logic [3:0] in;
7 output logic [3:0] out;
8 always_comb
9 begin
10     if((switch == 1 && set == 1)||(switch == 1 && set == 0)||(switch == 0 && set == 0))
11     begin // if 00 || 10 || 11 switch & set
12         out = in ;
13     end //end of if switch & set
14
15     else // 01 switch & set
16     begin
17         out = setNum;
18     end
19 end // end of the always statment
20
21 endmodule
22

```

- **Simulation:**

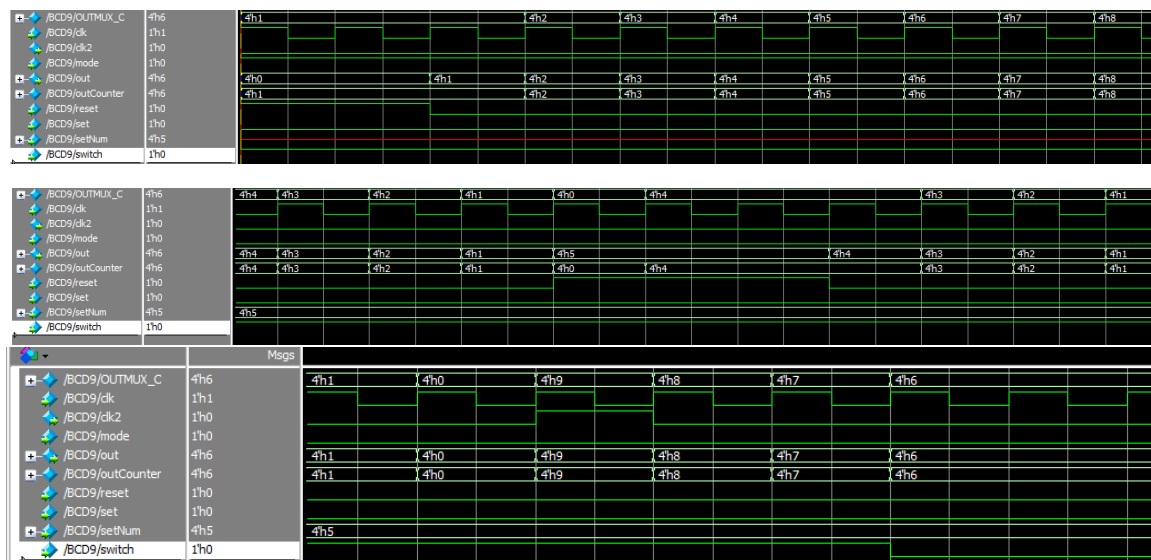
+ /Mux_Counter/in	4'h2	4'h2					
+ /Mux_Counter/out	4'h2	4'h2	4'h5	4'h2			
/Mux_Counter/set	1'h0						
+ /Mux_Counter/setN...	4'h5	4'h5					
/Mux_Counter/switch	1'h1						

BCD9 as a whole block Code:

```
//This the collection for the counter & muxes to include the reset and set
module BCD9 (switch,set,reset,mode,clk,setNum,out,clk2);
input logic switch,set,reset,mode,clk;
input logic [3:0] setNum;
output logic [3:0] out;
output logic clk2;
logic [3:0]outCounter; //output of the counter 9
logic [3:0]OUTMUX_C; // output of the mux after counter of set & switch
// this line is make the values equal each other instantly for the reset button
assign outCounter = out;
// counter module to increment or decrement
Counter_9 count(switch,mode,clk,out,outCounter,clk2);
//this module is to make the set only when the switch is off
Mux_Counter mux_count(set, switch, setNum, outCounter , OUTMUX_C);
//To choose the proper value for reset for the timer and the stop watch
Mux_R mux_reset(reset, mode, setNum,OUTMUX_C, out);

endmodule
```

BCD9 as a whole block simulation:



BCD5 as a whole block Code:

```
//This the collection for the counter & muxes to include the reset and set
module BCD5 (switch,set,reset,mode,clk,setNum,out,clk2);
input logic switch,set,reset,mode,clk;
input logic [3:0] setNum;
output logic [3:0] out;
output logic clk2;
logic [3:0]outCounter; //output of the counter 5
logic [3:0]OUTMUX_C; // output of the mux after counter of set & switch
// this line is make the values equal each other instantly for the reset button
assign outCounter = out;
// counter module to increment or decrement
Counter_5 count(switch,mode,clk,out,outCounter,clk2);
//this module is to make the set only when the switch is off
Mux_Counter mux_count(set, switch, setNum, outCounter , OUTMUX_C);
//To choose the proper value for reset for the timer and the stop watch
Mux_R mux_reset(reset, mode, setNum,OUTMUX_C, out);

endmodule
```

BCD5 as a whole block simulation:

[illegible]

The Block BCD as a whole block that contains all of the previous Code:

```
//This is the collection for all of the BCD Blocks
module Block_BCD (switch,set,reset,mode,clk,setNum1,setNum2,setNum3,setNum4,out1,out2,out3,out4,clk2);
input logic switch,set,reset,mode,clk;
input logic [3:0] setNum1;
input logic [3:0] setNum2;
input logic [3:0] setNum3;
input logic [3:0] setNum4;
output logic [3:0] out1;
output logic [3:0] out2;
output logic [3:0] out3;
output logic [3:0] out4;
output logic clk2;
logic CLKSIGNAL1; //output of the BCD 9 sec
logic CLKSIGNAL2; //output of the BCD 5 sec
logic CLKSIGNAL3; //output of the BCD 9 min

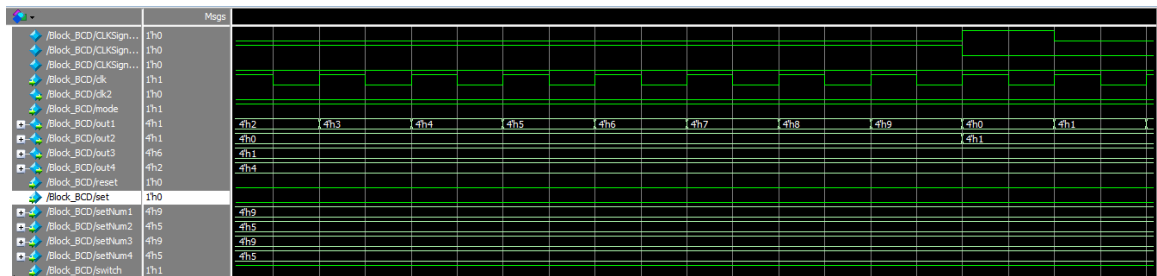
BCD9 BCD_9sec(switch,set,reset,mode,clk && ((|out1 || |out2 || |out3 || |out4) || mode),setNum1,out1,CLKSIGNAL1);
BCD5 BCD_5sec(switch,set,reset,mode,CLKSIGNAL1,setNum2,out2,CLKSIGNAL2);

BCD9 BCD_9min(switch,set,reset,mode,CLKSIGNAL2,setNum3,out3,CLKSIGNAL3);
BCD5 BCD_5min(switch,set,reset,mode,CLKSIGNAL3,setNum4,out4,clk2);

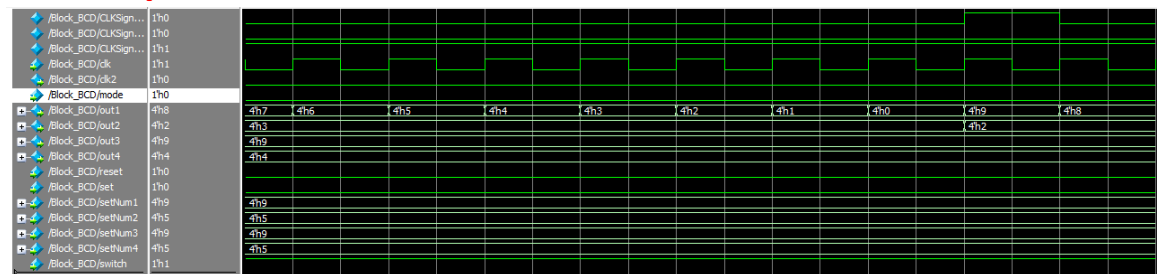
endmodule
```

The Block BCD as a whole block that contains all of the previous simulation:

- As a timer:



- As a stopwatch:



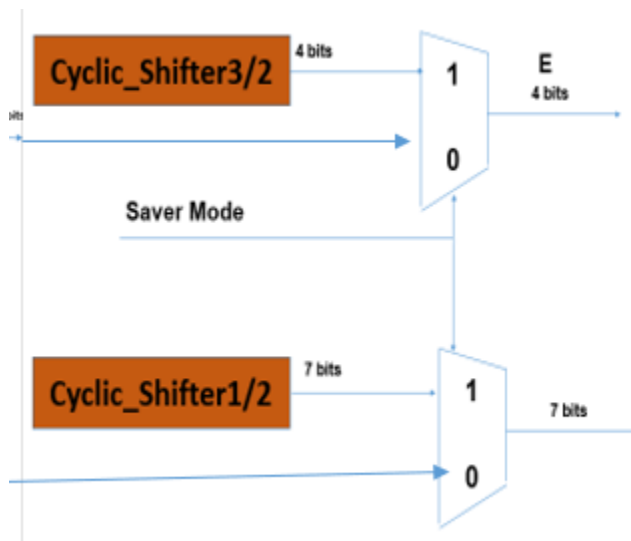
The saver Mode Part for display when the user doesn't push any button for 30 sec & the switch is off

To display the data there is a system of 2 cyclic shifters and 2 mux

One cyclic shifter for 3.5 sec to enable each 7 segment from the 4 (7 segments) -> the other cyclic shifter for 0.5 sec for each segment in the 7 segments

The first mux to choose between the cyclic shifter to enable the saver mode or to enable the ordinary status of the display with other shifter and the other mux to choose between the data from the shifter of 7 bits that enable each led at 0.5 sec or the 7bits that comes from the other Blocks.

As shown:



The code of the 2Mux:





- 4bits:

```
module MUX_Display(shifter, BCD, saver , out);

input logic saver;
input logic [3:0] shifter;
input logic [3:0] BCD;
output logic [3:0] out;
always_comb
begin
    if(saver)
    begin // if no change in the system
        out = shifter ;
    end //end of if saver

    else // if there's a change the savermode = 0
    begin
        out = BCD;
    end
end // end of the always statment
```

- endmodule

 /MUX_Display/saver	1'h1													
 /MUX_Display/shifter	4'h1	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h0
 /MUX_Display/BCD	4'h1	4'h1												
 /MUX_Display/out	4'h1	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h1	4'h0	4'h0

- 7bit:

```
module MUX_Display(shifter, BCD, saver , out);

input logic saver;
input logic [6:0] shifter;
input logic [6:0] BCD;
output logic [6:0] out;
always_comb
begin
    if(saver)
    begin // if no change in the system
        out = shifter ;
    end //end of if saver

    else // if there's a change the savermode = 0
    begin
        out = BCD;
    end
end // end of the always statment
```

- endmodule

- The code of the 2shifters:

```
// this module shifts the one in the enable 1 bit to the left in a cyclic order
module shifter(input logic clk,reset, output logic[3:0] out);
    logic[3:0] enable;
    always_ff @(posedge clk)
        if(reset) enable <= 4'b0001; //if reset is one reset the enable and output to 0001
        else enable <= {enable[2:0],enable[3]}; // else shift the second, third and forth bit to the left and put the first(most significant) bit in
    assign out = enable;
endmodule

// this module shifts the one in the enable 1 bit to the left in a cyclic order
module shifter(input logic clk,reset, output logic[6:0] out);
    logic[6:0] enable;
    always_ff @(posedge clk)
        if(reset) enable <= 7'b0000001; //if reset is one reset the enable
        else enable <= {enable[5:0],enable[6]}; // else shift the second
    assign out = enable;
endmodule
```

- Before them there is a clock divider for each :

```
//this module converting the 87.5 megaHertz clk into 1 Hertz clk -- 3.5sec
module clkdiv (input logic clk, output logic clk2=0);
    integer counter = 0;
    always_ff @(posedge clk)
        if(counter==87500000) //when the count reached 87.5 mega the counting starting from the beginning and reverse the bit of the clk2
            begin
                counter <= 1; //this line equivalent to counter=0; counter++;
                clk2 <= !clk2;
            end
        else counter++;
endmodule

//this module converting the 12.5 megaHertz clk into 1 Hertz clk -- 0.5 sec
module clkdiv (input logic clk, output logic clk2=0);
    integer counter = 0;
    always_ff @(posedge clk)
        if(counter==12500000) //when the count reached 12.5 mega the counting starting from the beginning and reverse the bit of the clk2
            begin
                counter <= 1; //this line equivalent to counter=0; counter++;
                clk2 <= !clk2;
            end
        else counter++;
endmodule
```

The Bonus Part (added) : → Watch

Watch block that contains the following:

3 BCD counters

The first counter is 4 bits that counts to 9, the second counter is also 4 bits that counts to 5 and the third one is 5 bits that counts to 24.

Note : The difference between these BCD counters here in the watch and the other counters in the main module of the timer and stop watch is that it increment the out put itself, but in the timer and the stop watch it was incrementing or decrementing the input every time by 1 as the input could change.

This value is added to this counters to be able to work as the other circuit is working -timer & stop watch-.

Note that this block is designed behaviorally as the set option is added to the same block.

First the 4 bit that counts to 9:

- The code:

```
module watch_9 (watch,set,clk,in,out,clk2);
input logic watch,set,clk;
input logic [3:0] in;
output logic [3:0] out = 0;
output logic clk2;
always_ff @(posedge clk, posedge (watch && set))
begin
//this is to set the watch
if (set == 1 && watch == 1)
begin
out <= in;
end
else
begin
out<=out;
if(out < 10)
begin
clk2 <= 0;
out <= out + 1;
if(out == 9)
begin
out <= 0;
clk2 <= 1;
end
end
end
else
begin
if(out > 9)
begin
out <= 0;
clk2 <= 1;
end
end
end
end
end
```

Second the 4 bit that counts to 5:

- The code:

```
module watch_5 (watch,set,clk,in,out,clk2);
input logic watch,set,clk;
input logic [3:0] in;
output logic [3:0] out = 0;
output logic clk2;

always_ff @(posedge clk, posedge (watch && set) )
begin
//this is to set the watch
if (set == 1 && watch == 1)
begin
out <= in;
end
else
begin
out <= out;
if(out < 6)
begin
clk2 <= 0;
out <= out + 1;

if(out == 5)
begin
out <= 0;
clk2 <= 1;
end
end
end
else
begin
if(out > 5)
begin
out <= 0;
clk2 <= 1;
end
end
end
end
```


Third the 5 bit that counts to 24:

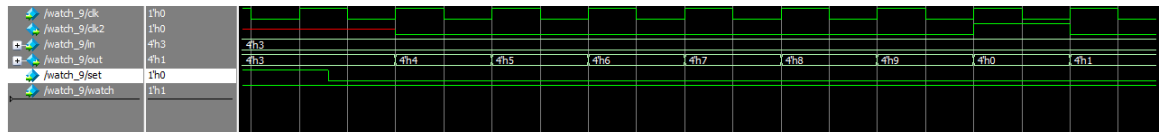
- The code:

```
module watch_2 (watch,set,clk,in,out,clk2);
input logic watch,set,clk;
input logic [4:0] in;
output logic [4:0] out = 0;
output logic clk2;

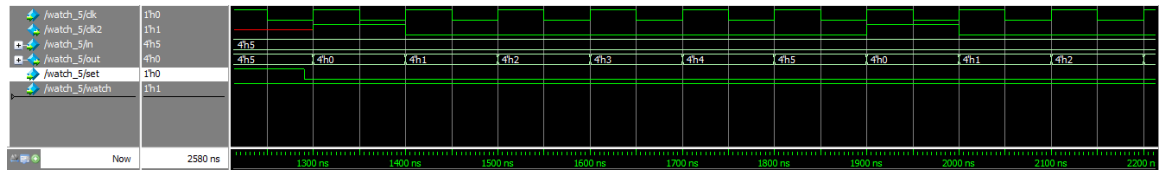
always_ff @(posedge clk ,posedge (watch && set))
begin
//this is to set the watch
if (set == 1 && watch == 1)
begin
out <= in;
end
else
begin
out <= out;
if(out < 25)
begin
clk2 <= 0;
out <= out + 1;
if(out == 24)
begin
out <= 0;
clk2 <= 1;
end
end
else
begin
if(out > 25)
begin
out <= 0;
end
end
end
end
end
```

Hint : as there is no check error for the bonus module and it is behaviorally designed I just put as condition to make the value 0 if it is greater that the value limit.

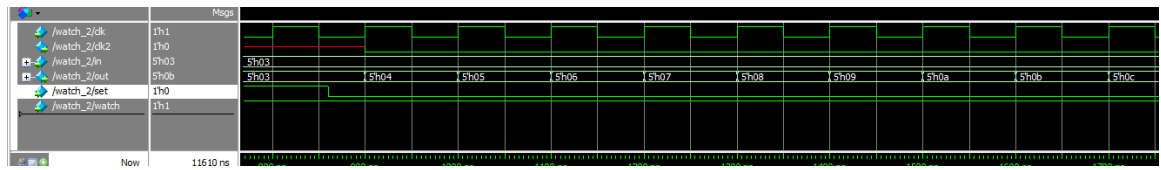
- The simulation for the counter 9:



- The simulation for the counter 5:



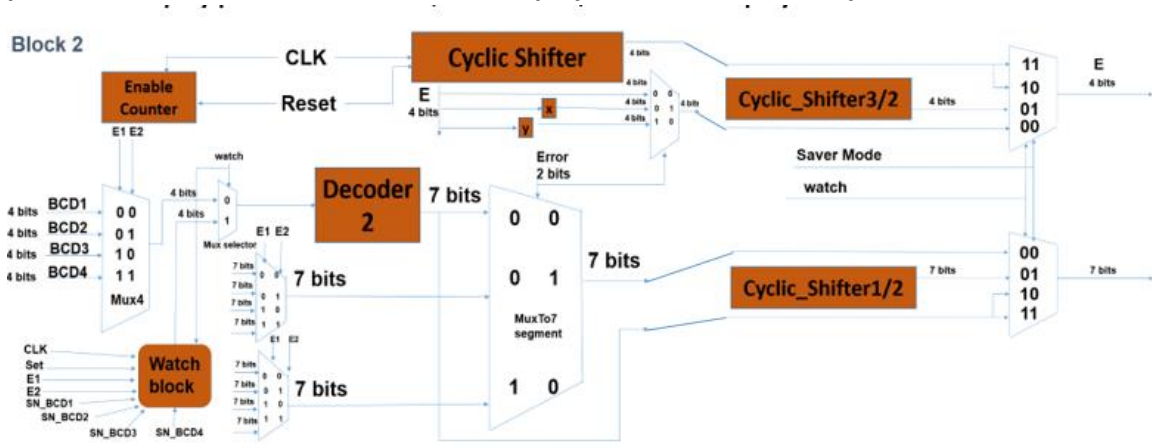
- The simulation for the counter 24:



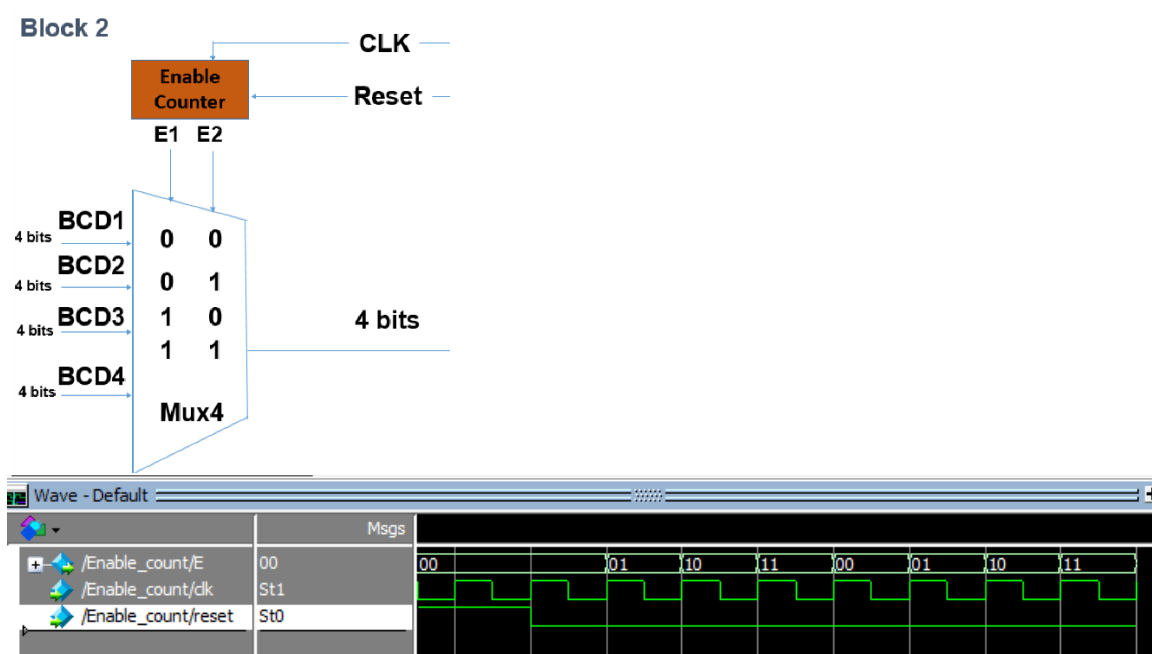
Team Member: Youssef Mahmoud

Block 2 and Mode block

Block 2 is supposed to be **the last stage** of the project, it includes **the display part** either in **error, saver mode, normal display, or the Watch bonus part**.



Enable Counter in Block 2:

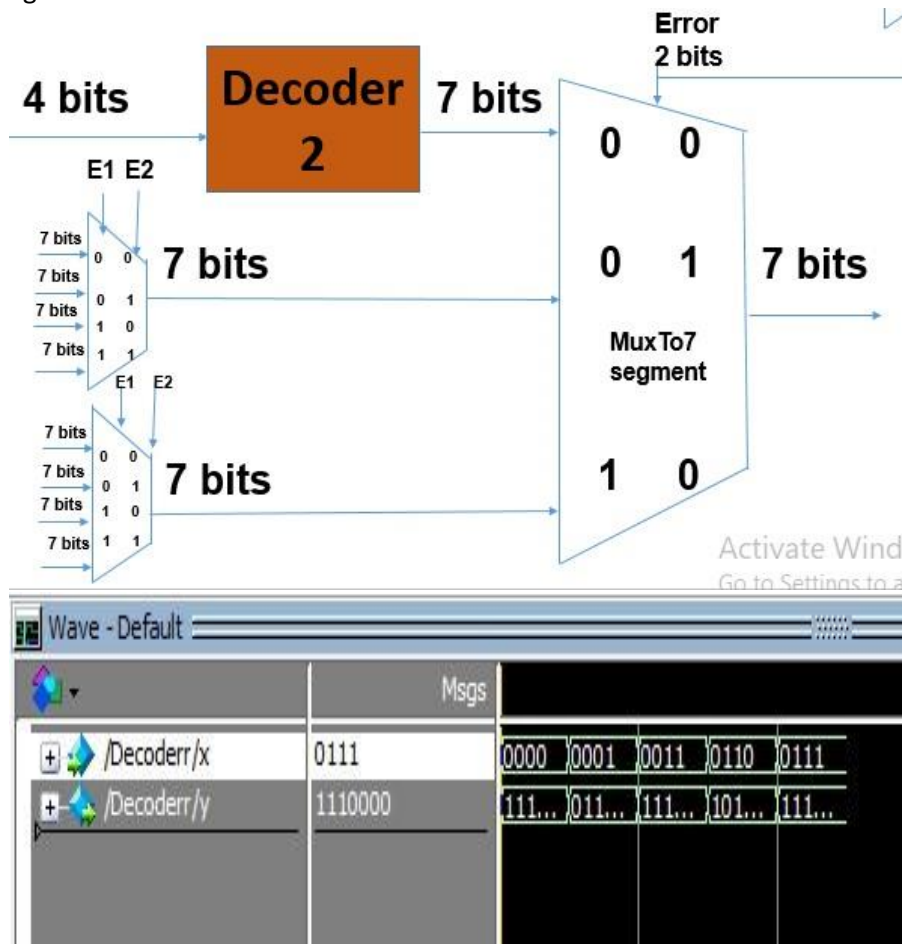


This counter is used to alternate a two bits Enable in order to be used as a selective condition for mux 4.

It is alternating between (00, 01, 10, 11)

Decoder 2 in Block 2:

Decoder 2 here is used to **produce 7 bits output** to the 4-1 7 segments to light its bits with the desired number in case there is no errors



- Inputs:

1- x 4 bits for the desired number (from 0 to 9)

- outputs:

1- x 7 bits

MuxE1 & MuxE2 in block2:

They are only different in the shape of the error

Error1: [] 59, Error2: -E 01

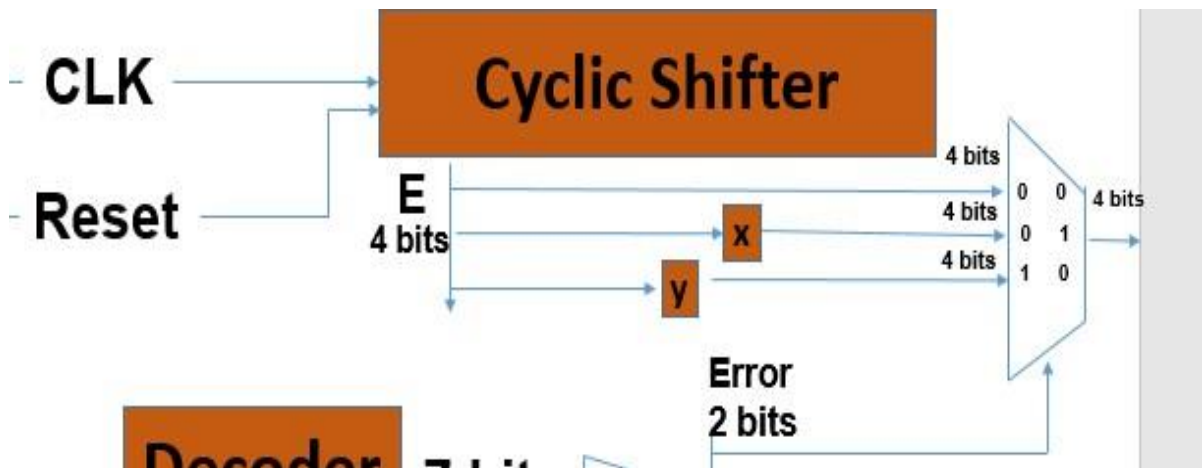
They are muxs that gets 2 Enable bits from the enable counter **in order to set its 7 bits output with a certain shape depending on the directed enable.**

Wave - Default		Msgs			
+ /muxE1/Efinal	1110011	1001110	1111000	1011011	1110011
+ /muxE1/Enable	11	00	01	10	11

MuxTo7segments is a mux that gets 3*7 bits inputs from MuxE1, MuxE2, and Decoder2, and two bits Error selectors to choose which case will be displayed on the 7 segments.

Wave - Default		Msgs			
+ /muxTo7Segment/E0	0101010	0101010			
+ /muxTo7Segment/E1	1111111	1111111			
+ /muxTo7Segment/E2	0000000	0000000			
+ /muxTo7Segment/E...	0000000	0101010	1111111	0000000	
+ /muxTo7Segment/E...	10	00	01	10	

Blinking mode process:



In the blinking time; Enable bits shall vary from their normal number to 0000 and vice versa.

After the 4 bits of the cyclic shifter are produced, they enter three paths as 3*4 input bits to mux2, this mux has two bits Error selectors.

Wave - Default		Msgs		
/mux2/E0	0001	0001		
/mux2/E1	0010	0010		
/mux2/E2	0100	0100		
/mux2/Efinal	0100	0001	0010	0100
/mux2/Error	10	00	01	10

In case there is no errors: 00: the 4 enable bits will pass as they are,

In case of Error1: 01: the 4 enable bits will enter blockX for the 3 times blinking process, and then pass through the mux

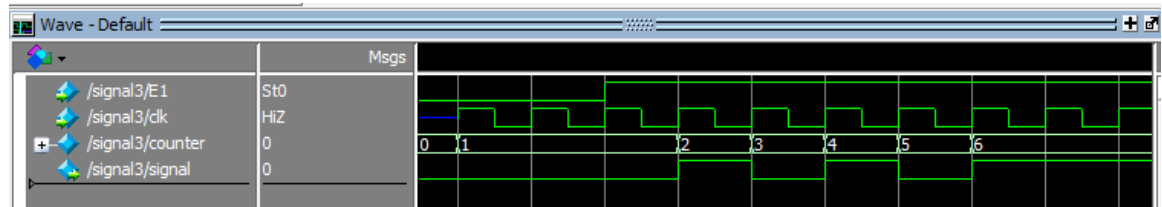
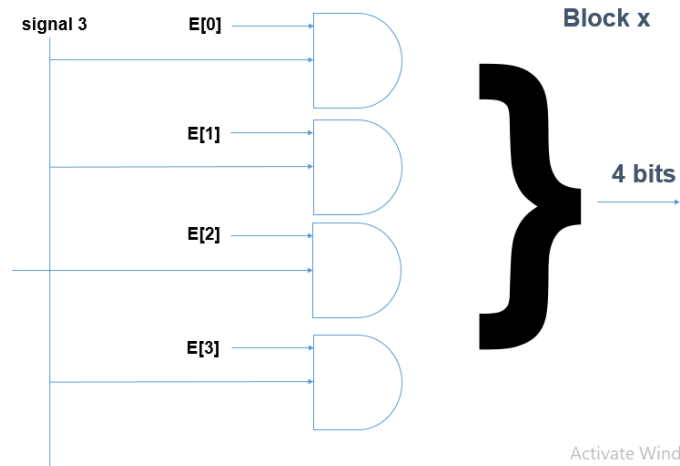
In case of Error2: 10: the 4 enable bits will enter blockY for the 1 time blinking process, and then pass through mux2

- blockX, and blockY are only different in the number of blinking times.

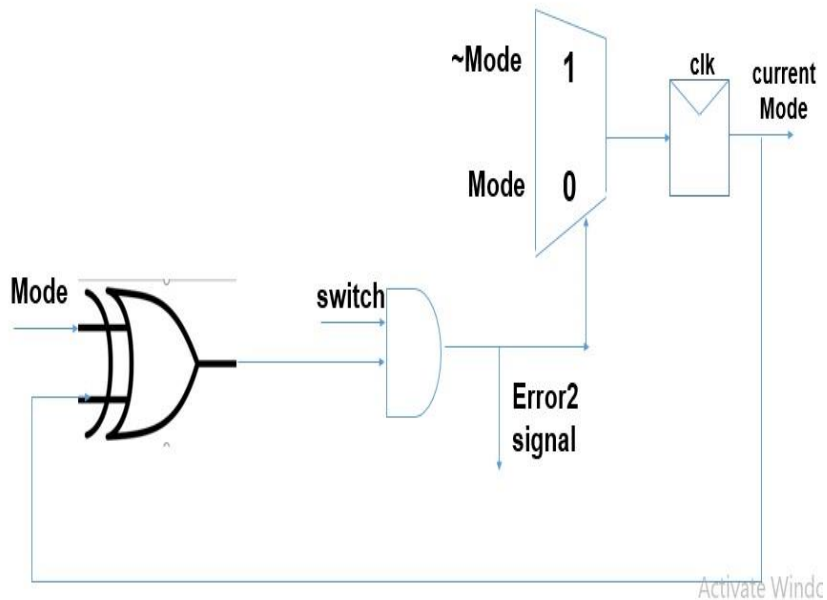
- blockX is composed of 4* 2inputs AND gates each AND gate gets one bit one bit of a signal3 (blink 3 times) block to work as a counter of 6 in Error1, and one bit from the 4 bits of the Enable bits.

- blockY is the same but with signal1 (blink one time) instead of signal3

- signal3 is taking Error1 bit as an input, and one second clk in order to alternate its output signal 3 times only when Error1 == 1, then keeps its signal = 1 till Error1 gets back to zero



error2 block “Mode”

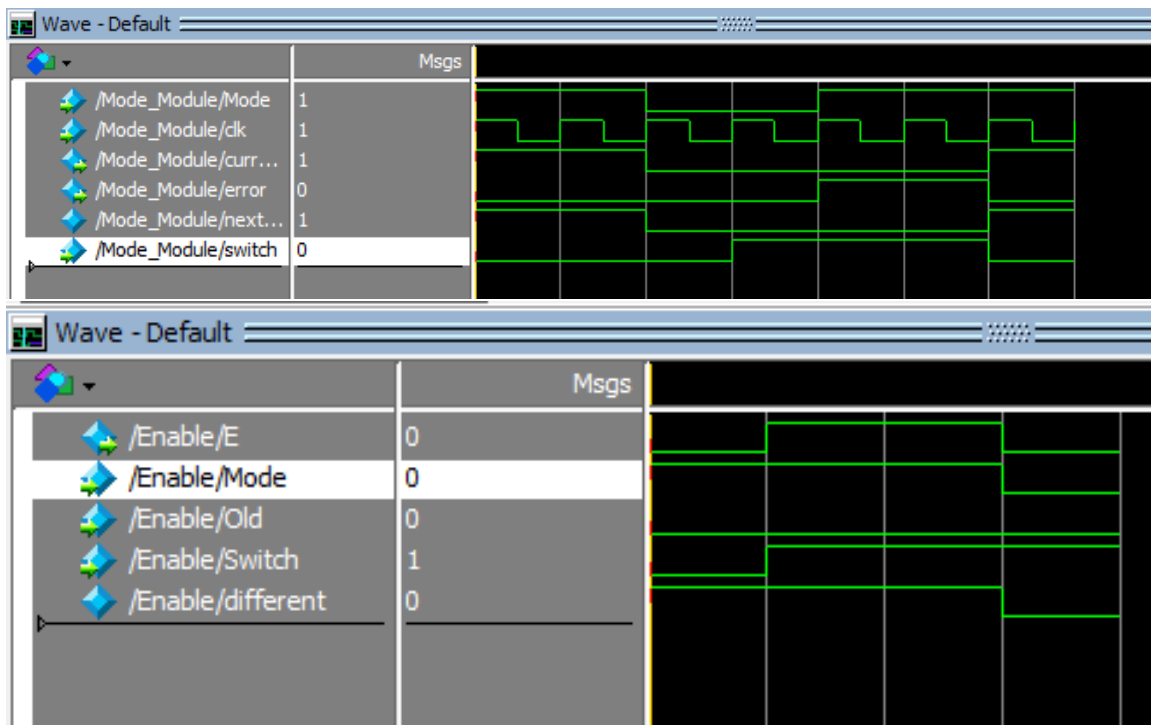


It is supposed to get a one mode bit, and one switch bit as inputs.

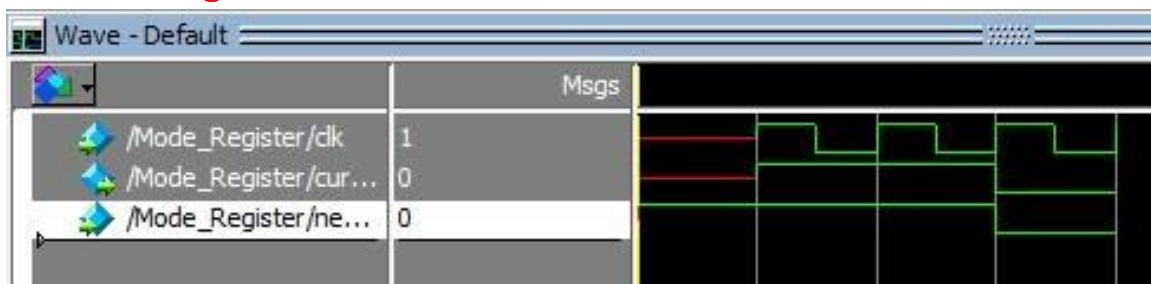
XOR gate is used to detect a variation between the current mode, and the new one, if they are different;

AND gate is used to see whether the switch is on or off, if AND gate output is 1, there is an error, so the mux, and the register keeps the current mode as it is, and the error signal is sent to block 2 with keeping the value of the old mode as it is

The whole mode module simulation:

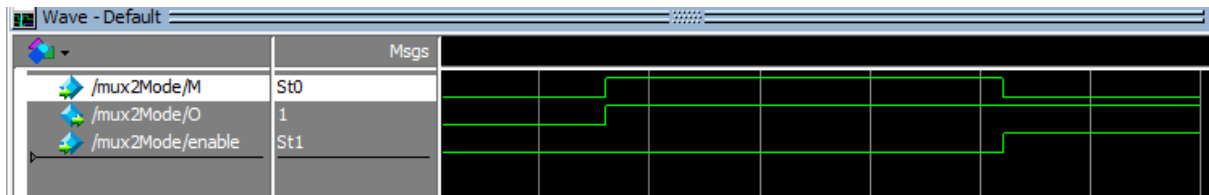


Where Enable module is responsible for producing the error signal

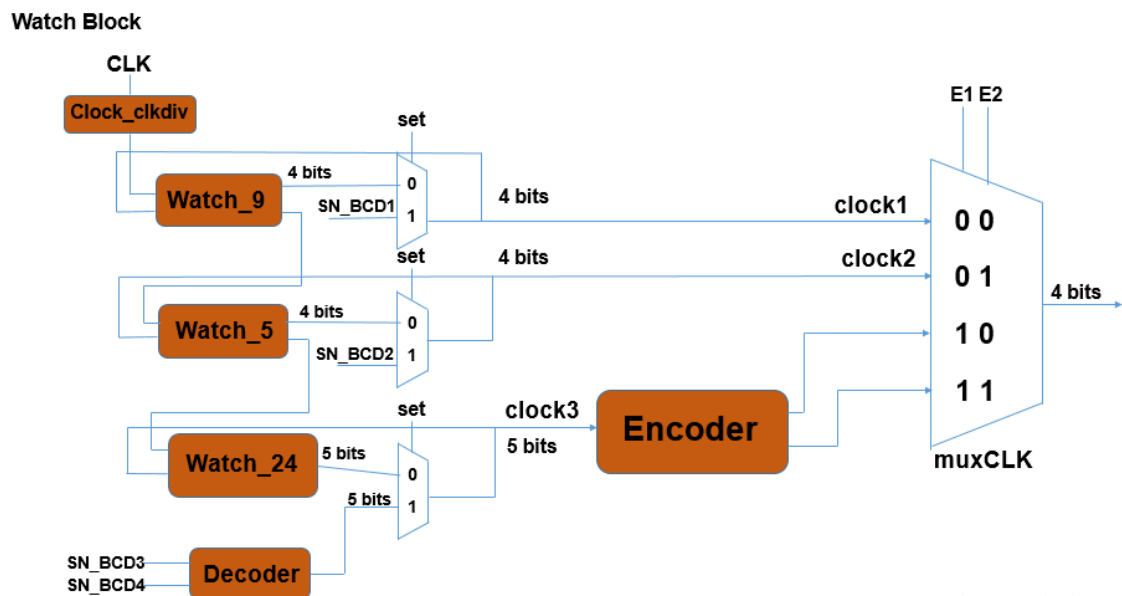


Mode Register is responsible on changing current mode with the mode value.

Mux2Mode is responsible to pass the mode or ~mode to the register depending on the error signal.



Improvement feature Watch block design and mechanism:



Activate Windows
Go to Settings to activate.

3 BCDs are used as counters just as we did in the 4 BCDs in phase 2, Watch_24 produces 5 bits, so it is responsible for two 7-segments. After the set muxs, I used an Encoder to split the 5 bits into two 4 bits to deal with its bits easily as the same done in phase 2.

After we got 4*4bits I used muxCLK with two enable selectors from the Enable counter to switch between the clock bits to produce 4 bits as an output from the mux

Encoder:

/Encoder/in	11000	00001	01101	11000
/Encoder/out3	0100	0001	0011	0100
/Encoder/out4	0010	0000	0001	0010

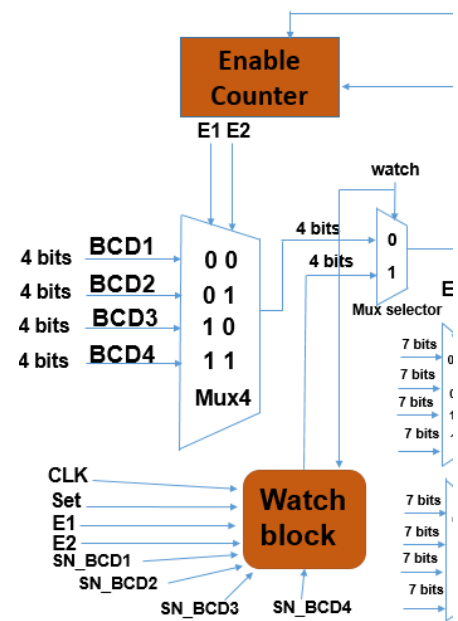
muxCLK:

Wave - Default		Msgs			
/muxCLK/Efinal	1111	0000	0101	1010	1111
/muxCLK/Enable	11	00	01	10	11
/muxCLK/clk1	0000	0000			
/muxCLK/clk2	0101	0101			
/muxCLK/clk3	1010	1010			
/muxCLK/clk4	1111	1111			

After that, I used a selector mux with one-bit selector indicating whether the watch mode is on, or off; if on, its output is the 4 bits of the muxCLK output, if off, its output

is the 4 bits mux4 output.

Wave - Default		Msgs			
/muxSelector/BCD	0000	0000			
/muxSelector/Efinal	0000	1111	0000		
/muxSelector/button	0				
/muxSelector/clock	1111	1111			

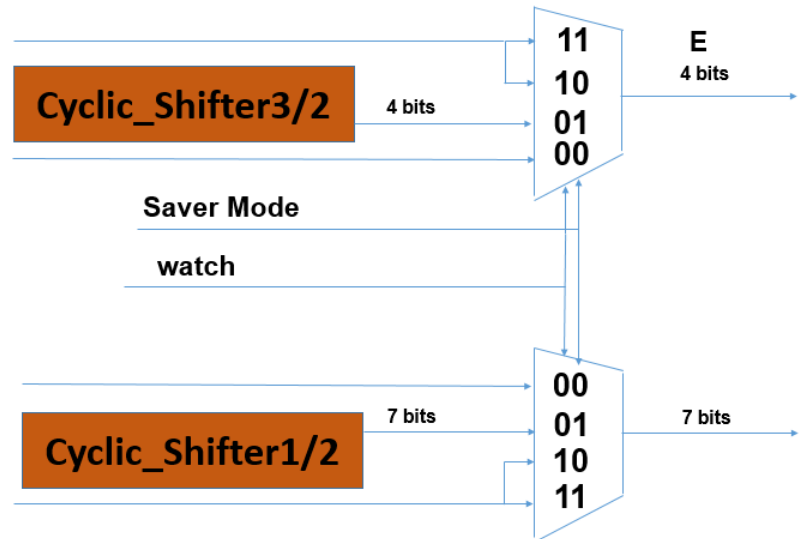


Then these 4 bits enters the decoder to convert them into 7 bits.

After that I added a watch selector to both muxs display at the final step:

If the watch button is off, the mux will perform its normal check whether there is a saver mode or it is normal display (no error, error1, error2).

But if the watch button is on, the 7 bits' output is the 7 bits output from the decoder, in order to neglect the errors issues, so do the 4 enable bits as they pass neglecting the blinking mode step.



Progress description

Block 2: it took nearly 11 hours

Mode Error Block "Error2": it took nearly 8 hours

Drawing the design on laptop: it took 5 hours

Improvement bonus feature: 3 hours

Total individual hours: 27

Team Member: Ramy Mohsen

Progress description:

I spend around **18 hours** coding these modules, fixing some **problems** with the software, and simulating and testing it. I completed what I started in the first phase, starting with the saver mode, with its registers, XOR gates, or gate, and a counter, then I worked with my colleagues in designing the blinking mode block, then we changed our coding style to be more structural.

Improvements: finishing the improvements of Block 1 took me 30 mins and 30 mins for drawing the new design on computer. The final module which is the combination of all modules took around 2 hours in the original project and around another hour to improve it. It took me around 2 hours drawing the design of the whole project. Testing the whole project also part of my job and it took hours of testing and reporting the problems found (I cannot determine an exact number of hours).

Watch Bonus: it took me 30 mins to combine the modules of the watch into one big module called watch and 2 hours to draw the modules on computer

Mux4:

- Description:

This module takes 4 4-bit inputs representing 4 digits and according to the enable the output will be 4-bit representing a digit from the 4 digits

the enable table: 00 -> the output will be the first digit

01 -> the output will be the second digit

10 -> the output will be the third digit

11 -> the output will be the fourth digit

- the code for the block:

```
module mux4 (input logic [3:0] BCD1,BCD2,BCD3,BCD4,logic [1:0] enable, output logic[3:0] O);  
    always_comb  
        if(enable == 0) O=BCD1;  
        else if(enable == 1) O=BCD2;  
        else if(enable == 2) O=BCD3;  
        else if(enable == 3) O=BCD4;  
endmodule
```

- The simulation of the block:

/mux4/BCD1	-No Data-	4'h1				
/mux4/BCD2	-No Data-	4'h2				
/mux4/BCD3	-No Data-	4'h4				
/mux4/BCD4	-No Data-	4'h8				
/mux4/enable	-No Data-	2'h0	2'h1	2'h2	2'h3	
/mux4/O	-No Data-	4'h1	4'h2	4'h4	4'h8	

As we see in the simulation when enable is 00 the output is the BCD1(the first digit). If it was 01 the output is the BCD2 (the second digit) and so on.

Cyclic shifter:

- Description:

this module shifts the one in the enable 1 bit to the left in a cyclic order

input: clk representing 50 mega Hertz clk, reset to reset the value of enable to 0001

output: out representing enable after the change

- the code for the block:

```
module shifter(input logic clk,reset, output logic[3:0] out);
    logic[3:0] enable;
    always_ff @(posedge clk)
        if(reset) enable <= 4'b0001;
        else enable <= {enable[2:0],enable[3]};
    assign out = enable;
endmodule
```

- The simulation of the block:

/shifter/dk	1'h1					
/shifter/reset	1'h0					
/shifter/out	4'h4	4'h1	4'h2	4'h4	4'h8	4'h1
/shifter/enable	4'h4	4'h1	4'h2	4'h4	4'h8	4'h1

As we in the simulation when the reset is 1, the enable and the output = 0001 or 1 and when the reset is 0, the 1 in 0001 is shifted cyclically from 0001 or 1 to 0010 or 2 to 0100 or 4 to 1000 or 8 to 0001 or 1 and so on

Clk div (clock divider):

- Description:

this module converting the 50 mega Hertz clk into 1 Hertz clk

input: clk representing 50 megahertz clk

output: clk2 representing the 1 hertz clk

- the code for the block:

```
module clkdiv (input logic clk, output logic clk2=0);
    integer counter = 0;
    always_ff @(posedge clk)
        if(counter==25000000)
            begin
                counter <= 1; //this line equivalent to counter=0; counter++;
                clk2 <= !clk2;
            end
        else counter++;
endmodule
```

- The simulation of the block:

In the simulation the number 25000000 is replaced with 3 to make the simulation more visible



As we see in the simulation clk2 makes a rising edge after 6 rising edges of clk.

BCD4: represent the set number for the fourth BCD (5 min BCD)

error1: represent whether there is an error or not comes out of block 1

This module has another 2 module one of them is cyclic shifter and I already explained it and the other one error 1 module, and I will explain this module later.

The Reset reset the bits that comes out of it to 0001 and the button serves as clk to the shifter every time the button is pressed and released the 1 is shifted to the left and the user can set another digit. The Button serves also as a clk for the change of the BCDs

Every bit in the output of the shifter ANDing it with switch inverted and e1 (that comes out of the error 1 block) represent a condition for the changing of the BCDs (the condition for the mux). when the clk comes to its rising edge and the condition is true, the SN is loaded to one of the BCDs.

The e1 will be ANDed with the set to represent if there is an error or not as if there is an error and the set is 1 then the error1 bit will be 1 represent that there is an error. (as we will see in code $error1 = e1 \& Set$)

- The code:

note1: that there are more comments in the code file.

note2: the mux module in the figure of the block 1 is replaced with only an if statement.

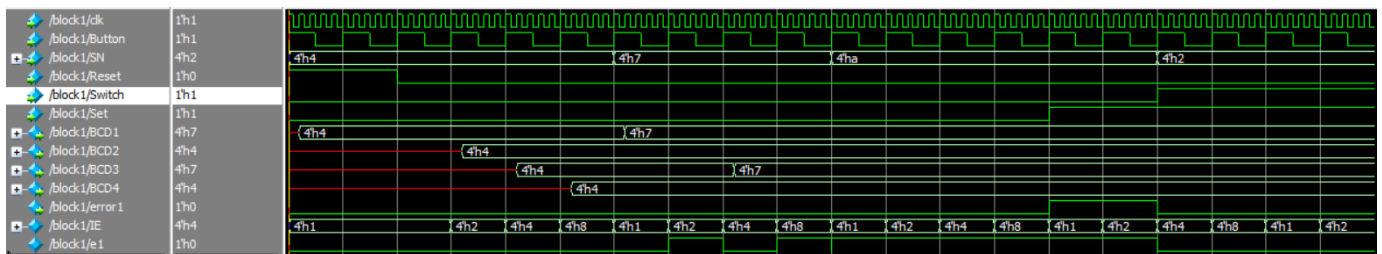
```
module block1 (input logic[3:0] SN, logic clk, Reset, Button, Switch, Set, output logic [3:0] BCD1=0, BCD2=0, BCD3=0, BCD4=0, logic error1);
logic[3:0] IE; //input enable
logic e1;
// moves to the next digit for setting every time the button is pressed
shifter s(Button, Reset, IE);

//error checking
Error1 e(SN, IE, e1);

always_ff@(posedge clk) //50 mega hertz clk
begin
    if(IE[3] & ~Switch & ~e1) BCD4 <= SN; //represent the mux that control the changing of the fourth digit (5 min digit)
    if(IE[2] & ~Switch & ~e1) BCD3 <= SN; //represent the mux that control the changing of the third digit (9 min digit)
    if(IE[1] & ~Switch & ~e1) BCD2 <= SN; //represent the mux that control the changing of the second digit (5 sec digit)
    if(IE[0] & ~Switch & ~e1) BCD1 <= SN; //represent the mux that control the changing of the first digit (9 sec digit)
end
assign error1 = e1 & Set;
endmodule
```

- The simulation:

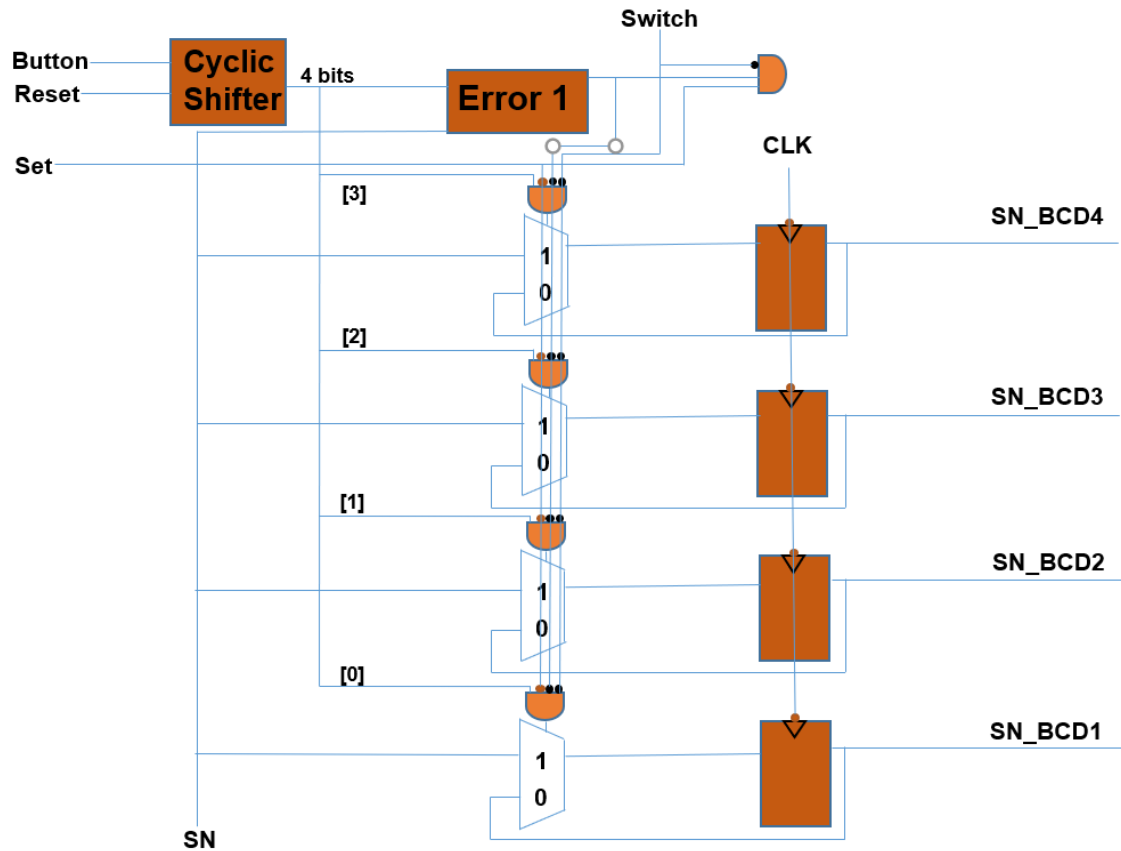
Note: there is only one difference between the code and the simulation. Which is the starting point for all BCD is 0 instead of Z.



As we see in the figure, if the reset is 1 it only can set the first digit (9 sec digit). If the reset is 0 then at every rising edge of the button, we can set another digit. If the SN is greater than 5 it only can set the first and the third digit (the second and the fourth digit will make e1 1 (which means, there is an error)). If the SN is greater than 9 then we cannot set anything and e1 will be 1. The error1 bit will be 1 only if there is an error (e1 is 1) and the Set bit is 1. Notice that we cannot set anything if the switch is 1.

- Improvement:

Block 1



The new code: the changes are shown by yellow highlighting

```
module block1 (input logic[3:0] SN, logic clk, Reset, Button, Switch, Set, output logic [3:0] BCD1=0, BCD2=0, BCD3=0, BCD4=0, logic error1);
    logic[3:0] IE; //input enable
    logic e1;
    // moves to the next digit for setting every time the button is pressed
    cyclic_shifter s(Button, Reset, IE);

    //error checking
    Error1 e(SN, IE, e1);

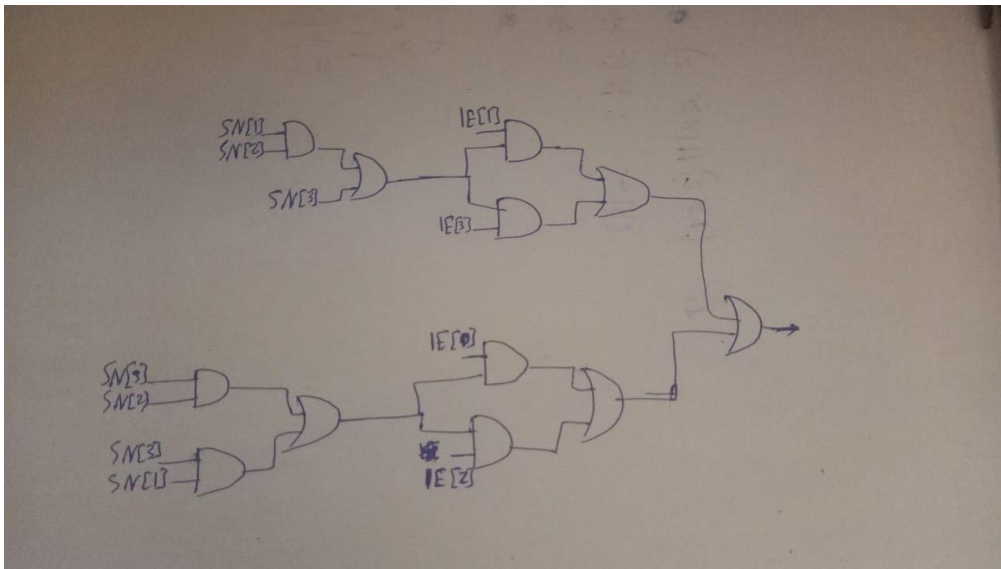
    always_ff@(posedge clk) //50 mega hertz clk
    begin
        if(IE[3] && ~Switch && ~e1 && Set) BCD4 <= SN; //represent the mux that control the changing of the fourth digit (5 min digit)
        if(IE[2] && ~Switch && ~e1 && Set) BCD3 <= SN; //represent the mux that control the changing of the third digit (9 min digit)
        if(IE[1] && ~Switch && ~e1 && Set) BCD2 <= SN; //represent the mux that control the changing of the second digit (5 sec digit)
        if(IE[0] && ~Switch && ~e1 && Set) BCD1 <= SN; //represent the mux that control the changing of the first digit (9 sec digit)
    end
    assign error1= e1 && Set && ~Switch;
endmodule
```

The reason:

- 1- The first update which ANDing Set with IE AND NOT Switch AND NOT e1 so the number used to set every BCD will not be updated with the value of SN unless the Set bit is 1.
- 2- The second update which ANDing NOT Switch with Set AND e1 to produce error 1 so the error1 will not be 1 if the switch is off

The simulation:

- Error 1:



Inputs: SN: 4 bits represent a number entered by the user

IE: the output from the shifter represent which digit will be set

Outputs: e1: represent whether there is an error or not

The error will rise if the SN is more than 5 in the first (IE=1000 or 0010) and third digit and more than 9 in the second and fourth digit (IE = 0001 or 0100)

- The code:

Note: that I didn't change the code I change the look of it (instead of using notepad, I use Modelsim)

- **The simulation:**

As we see, there is no error unless we tried to set a number bigger 5 in the first and the third digit or bigger than 9 in the second and the fourth digit

Inputs:

P_NM: represent the next state for the button that the user use to set different digits

This module has 8 registers (to restore the current states). ORing what gets out of XORing every current and next state for everything except the switch with the switch will be the reset for the counter that counts for 30 sec if nothing is change for 30 sec the counter will rises the SM to 1 if something is changed the counter will be reset and the SM will be 0

- The code:
- Savermode:

Note: that I did not change the code I change the look of it (instead of using notepad, I use Modelsim)

```
//CM for current state, NM for next state and SM for saver mode bit that represent if the conditions for saver mode satisfied or not
module savermode (input logic[3:0] setnumber_NM, logic clk, switch, mode_NM, reset_NM, set_CM, P_NM, output SM);
    // defining the current state for every input
    logic[3:0] setnumber_CM;
    logic mode_CM, reset_CM, set_CM, P_CM;
    //the reset of the SM_counter
    logic counter_R;

    // new XORing every next state and current state of every input to see if they changed or there is no change (next state = current state)
    // after that ORing every output of the XOR and the switch. this will be the reset of the counter that produces the signal
    assign counter_R = (switch) || (mode_NM ^ mode_CM) || (reset_NM ^ reset_CM) || (set_CM ^ set_CM) || (P_CM ^ P_CM)
        || (setnumber_NM[3] ^ setnumber_CM[3]) || (setnumber_NM[2] ^ setnumber_CM[2]) || (setnumber_NM[1] ^ setnumber_CM[1])
        || (setnumber_NM[0] ^ setnumber_CM[0]);

    // this counter produces the SM bit that is if no thing is changed for 30 sec (the counter counts for 30 times as the clk period is 1 sec)
    //makes the SM 1 otherwise makes the SM 0
    SM_counter c(clk, counter_R, SM);

    Register M(mode_NM, mode_CM, clk); //register for the mode to change the current state of the mode with the next state
    Register R(reset_NM, reset_CM, clk); //register for the reset to change the current state of the reset with the next state
    Register S(set_CM, set_CM, clk); //register for the set to change the current state of the set with the next state
    Register P(P_CM, P_CM, clk); //register for the button P to change the current state of the P with the next state

    // the next 4 lines are the registers for the 4 bits of the setnumber to change its current state with the next state
    Register SN3(setnumber_NM[3], setnumber_CM[3], clk); //the first bit (most significant)
    Register SN2(setnumber_NM[2], setnumber_CM[2], clk); //second bit
    Register SN1(setnumber_NM[1], setnumber_CM[1], clk); //third bit
    Register SN0(setnumber_NM[0], setnumber_CM[0], clk); //forth bit

endmodule
```

SM_counter: (for the simulation this counter will count up to 3 not 30S)

Note: that I did not change the code I change the look of it (instead of using notepad, I use Modelsim)

```
// SM for saver mode bit that represent if the conditions for saver mode satisfied or not
module SM_counter (input logic clk, reset, output logic SM);
    integer count = 0;
    always_ff @(posedge clk, posedge reset)
    if(reset) //if reset is 1 then reset the counter to 0 and make the output SM 0 which means that the saver mode will be off
    begin
        SM <= 0;
        count <= 1;
    end
    else if(count == 3) SM <= 1; // if reset is 0 and the count reaches 30 then the count will remain the same and SM will be 1 (saver mode is on)
    else
    begin // else the count will be incremented and SM will be (saver mode is off)
        SM <= 0;
        count++;
    end
endmodule
```

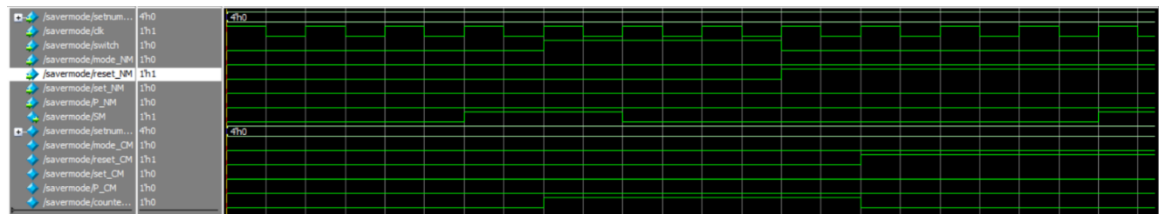
Register:

Note: that I did not change the code I change the look of it (instead of using notepad, I use Modelsim)

```
module Register(next_state, current_state, clk);
    input logic next_state, clk;
    output logic current_state;

    always_ff @(posedge clk)
        current_state <= next_state; // change the current state with the next state at every rising edge of the clk
endmodule
```

- The simulation:



Note: that the 2 rectangles mean 1 clk

As we see in the simulation, if nothing is changed for 3 clks the SM will be 1 if the switch is 1 the SM will always be 0 and the counter will always be reset. If anything, else is changed the counter will be reset for 1 time and SM will be 0 and wait another 3 clks to be SM if nothing is changed

Clock_clkdiv module:

Inputs: clk represent the 50-megahertz clk

Outputs: clk2 represent the 1 min clk

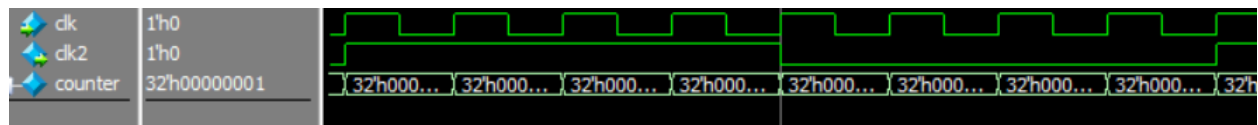
this module converting the 50 mega Hertz clk into 1 min clk

- The code:

```
//this module converting the 50 megaHertz clk into 1 min clk ()
module Clock_clkdiv (input logic clk, output logic clk2=0);
    integer counter = 0;
    always_ff @(posedge clk)
        if(counter >= 4) //when the count reached 1500000000 the counting starting from the beginning and reverse the bit of the clk2 (4 is just for simulation)
            begin
                counter <= 1; //this line equivalent to counter=0; counter++;
                clk2 <= !clk2;
            end
        else counter++;
endmodule
```

- The simulation of the block:

In the simulation the number 1500000000 is replaced with 4 to make the simulation more visible



As we see in the simulation clk2 makes a rising edge after 8 rising edges of clk.

Decoder module:

Input: SN_BCD3, SN_BCD4 (4 bits each) represent the 2 set numbers for hours of the watch

Output: out (5 bits) represent one number the sets the hours in the hours counter

The module converts 2 4-bit numbers to 5-bit number as the 2 4-bit represent 2 numbers the user entered to set the hours of the watch and the 5-bit number represent the number required by the hours counter to set the hours of the watch

- The code:

The 2 4-bit input represent 2 numbers in decimal so the code convert them to binary and to 5-bit output

```
module decoder (input logic [3:0] SN_BCD3, SN_BCD4 , output logic [4:0] out);
    assign out = SN_BCD3 + (SN_BCD4 * 10);
endmodule
```

- The simulation:

SN_BCD3	4'd3	4'd5	4'd8	4'd3
SN_BCD4	4'd2	4'd0	4'd1	4'd2
out	5'd23	5'd5	5'd18	5'd23

as we see in the simulation the number 05 8-bits represent 5 5-bits, 18 8-bits represent 18 5-bits, and 23 8-bits represent 23 5-bits.

Watch module:

Inputs:

watch: 1-bit represent the watch mode (1 means that we are in the watch mode).

set: 1-bit represent if the user wants to set the watch or not.

clk: 50-megahertz clk.

enable: condition for the mux inside the block and represent which 4-bits will get out of the module.

SN_BCD1, SN_BCD2, SN_BCD3, SN_BCD4: represents the numbers the used entered to set the watch with.

Outputs:

outwatch: 4 bits represent a digit of the watch.

- The code:

```

module watch (watch,set,clk,SN_BCD1,SN_BCD2,SN_BCD3,SN_BCD4,enable,outwatch,clk4);
input logic watch,set,clk;
input logic [1:0] enable;
input logic [3:0] SN_BCD1;
input logic [3:0] SN_BCD2;
input logic [3:0] SN_BCD3;
input logic [3:0] SN_BCD4;

output logic [3:0] outwatch;

logic clk1;
logic clk2;
logic clk3;

output logic clk4;

//watch clk divider
Clock_clkdiv ccd(clk, clk1);

//decoder
logic [4:0] outdecoder;
decoder D(SN_BCD3,SN_BCD4,outdecoder);

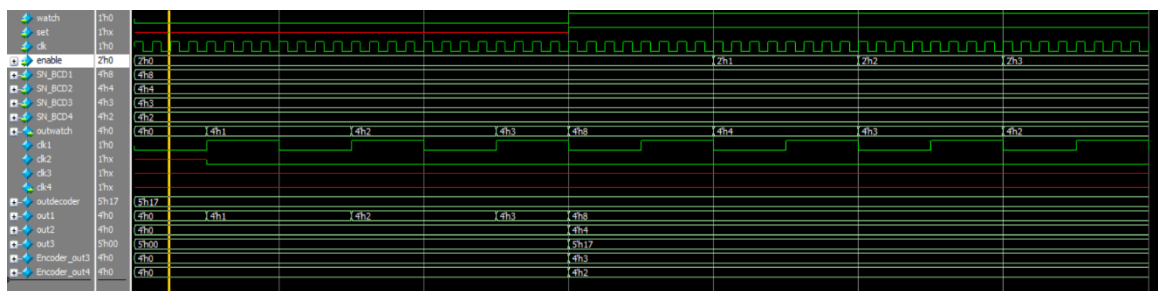
//watch counter
logic[3:0] out1,out2;
logic[4:0] out3;
watch_9 w9(watch,set,clk1,SN_BCD1,out1,clk2);
watch_5 w5(watch,set,clk2,SN_BCD2,out2,clk3);
watch_2 w12(watch,set,clk3,outdecoder,out3,clk4);

//Encoder
logic[3:0] Encoder_out3; logic[3:0] Encoder_out4;
Encoder EN(out3, Encoder_out3, Encoder_out4);
//MuxClock
muxCLK mu(out1, out2, Encoder_out3, Encoder_out4,enable, outwatch);

endmodule

```

- The simulation:



As we see in the simulation the watch will keep counting until the set and watch is 1 with represent that the user wants to set the digits of the clock. And with each value of the enable one digit of the watch digits will get out of the module

project progress description:

First, we met 2 times to draw the functional design of the project, then we focused on splitting the work among us as shown in details in this report. We redesigned the process after a feedback from our TA, then we worked till the improvement phase.

It took 3 days for the improvement phase to be done after (designing, coding, drawing, testing, and simulating)

Final module Collection (made by Ramy Mohsen):

```
module finalmodule(input logic switch,set,inmode,reset,clk,button,watch,logic [3:0] SN,output logic[3:0] enable_out,logic[6:0] digit_out, logic LED1,LED2);
//mode module
logic mode,error2;
Mode_Module m(inmode, switch, mode, error2, clk);

//clk divider module that generate 1 sec clk
logic clk1sec;
clkdiv c(clk,clk1sec);

//saver mode module
logic SM;
savermode s(SN,clk1sec,switch,mode,reset,set,button,SM);

//block 1 module
logic error1;
logic[3:0] SN_BCD1,SN_BCD2,SN_BCD3,SN_BCD4;
block1 b(SN, clk, reset, button, switch, set, SN_BCD1, SN_BCD2, SN_BCD3, SN_BCD4, error1);

//BCD module
logic [3:0] BCD1,BCD2,BCD3,BCD4;
Block_BCD bcd(switch,set ## ~error1 ## ~error2 ## ~watch,reset,mode,clk1sec ## ~error1 ## ~error2,SN_BCD1,SN_BCD2,SN_BCD3,SN_BCD4,BCD1,BCD2,BCD3,BCD4,clk2);

//block 2 module
Block3Module b2(clk, BCD1, BCD2, BCD3, BCD4, SN_BCD1, SN_BCD2, SN_BCD3, SN_BCD4, reset, {error2,error1} , SM, watch, set, digit_out, enable_out);

assign LED1=clk1sec;
assign LED2=clk1sec;
endmodule
```

module description:

The set number is coming from the user to block 1 then using block 1 connecting the set number with the corresponding BCD, after making sure that there is no error in setting. Each BCD is responsible for a digit from the 4 digits of the module (setting, resetting, counting up and down and start and stop the counting) and this happened synchronously with the clk that is coming from the clk div (this block converts the 50-megahertz clk of the FPGA in to 1 hertz clk) Then we got 2 main blocks: error2 block, and saver mode block, both of them sends signals to block 2, after that, the 4 BCD enter block 2 which output is 11 bits (7 for the 7 segment and 4 for enabling each 7 segment). Block 2 is divided into 2 parts the first one is a counter (with reset that reset the value with 0) that return 2-bit enable from 0 to 3 every rising edge from the undivided clk representing each 7 segment and after that the 4 BCD is entered a decoder to be 7 bits, two other multiplexers for the errors shapes are producing 7 bits, both multiplexers, and the decoder enters a mother multiplexers to choose between them depending on two error bits, after that they enter another mux with a selective saver mode one bit, if there is a saver mode the output 7 bits comes from a saver mode mux that has a cyclic shifter as a selective bits. Regarding the blinking mode, it is controlled by a mux with AND logic gates, and signal3, signal1 blocks for the blinking mode, this mux finally produces 4 bits, then they enter another mux controlled by a saver mode one bit, if there is a saver mode, the 4 bits for the enable comes from another mux controlled by a cyclic shifter for its selections, finally 11 bits are produced to the 4x1 7 segments.

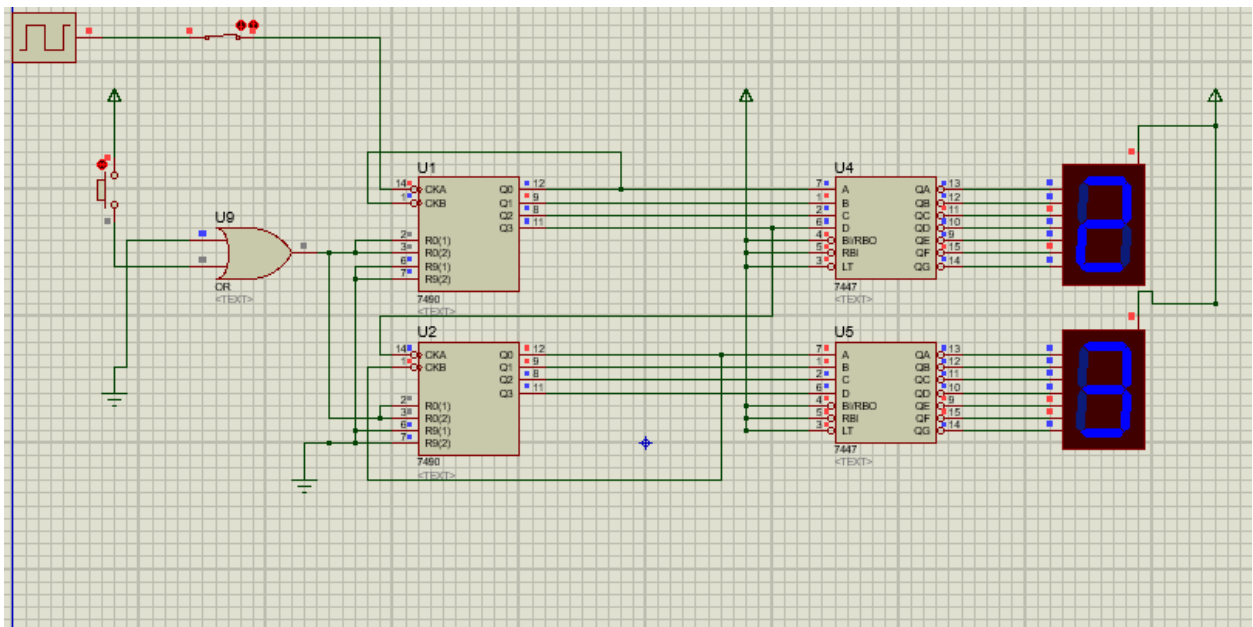
For the improvement phase Watch feature:

We added 3 BCDs for counting, then the Watch block is presented to produce 4 bits representing each 7 segment, after that there was a multiplexer that chooses the watch process or the original process, then at the end there is a watch mode selector to get the 7 bits and the 4 enable bits as they are neglecting the error issues in phase2.

Proteus feature is explained in detail above in Mohamed Helmy part.

Bonus Part 3: -> The counter 00 : 99 with restart and reset

The circuit as shown:



With 2 BCD counter 4bits (7490) that designed to count to 9 and return to 0 in bit 10

Connecting the first bit to increment every time by 1 as illustrated in the data sheet the last value in the counter by 1.

Also connecting the last bit to be a clock for the other counter as illustrated in the circuit to be as a clock for the other counter.

The button stop make the counting stop and resume to the last point.

The button restart enable the restart 1 & 2 (0) to clear the counters and make them start from 0. The 2 decoders (7447) to convert from 4 bits to 7 bits to enable each of the 7 segments.