EECS 2070 02 Digital Design Labs 2020
Lab 4

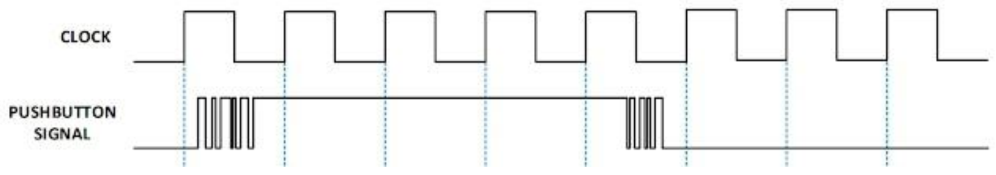
學號：108062281　　姓名：莊晴雯

1. 實作過程

Debounce and One Pulse

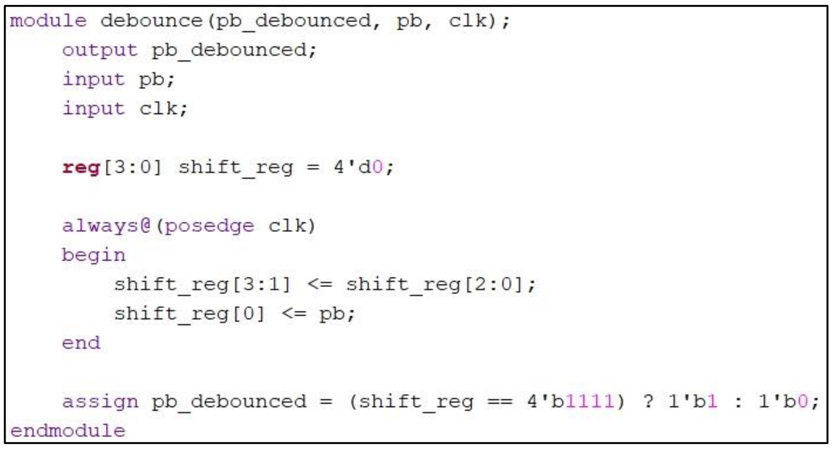
Debounce:

Initially, each pushbuttons generate a low (0) output when they are at rest and high (1) output when they are pressed. However, each pushbuttons contains a metal spring such that when the pushbutton is pressed and released, the switch contact will bounce several times before stabilizing and will generate a random number of unwanted signal pulses.



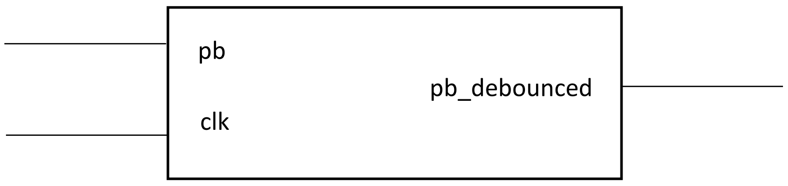
Therefore, we need to implement the module debounce to generate only a single pulse when pressing a button on the FPGA board.

The debounce module receives two inputs, pb and clk, and has another shift_reg of four bits. There is an always@(posedge clk) and inside the always block we begin shifting the shift_ref by assigning shift_reg[3:1] to shift_ reg[2:0]. Then for the shift_reg[0], we assign it to the value of pb. As for the output pb_debounced, it will become 1'b1 only when shift_reg is equal to 4'b1111, and 1'b0 otherwise.



```
module debounce(pb_debounced, pb, clk);
    output pb_debounced;
    input pb;
    input clk;

    reg[3:0] shift_reg = 4'd0;

    always@(posedge clk)
    begin
        shift_reg[3:1] <= shift_reg[2:0];
        shift_reg[0] <= pb;
    end

    assign pb_debounced = (shift_reg == 4'b1111) ? 1'b1 : 1'b0;
endmodule
```

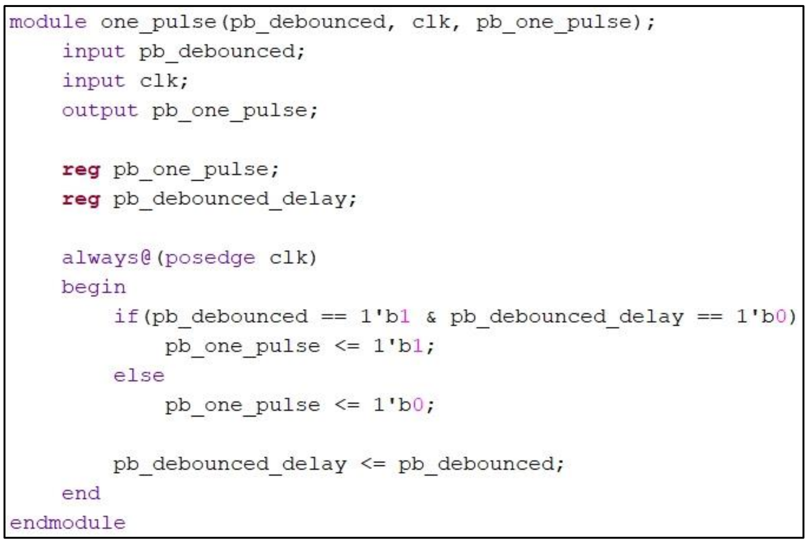
Debounce block diagram:



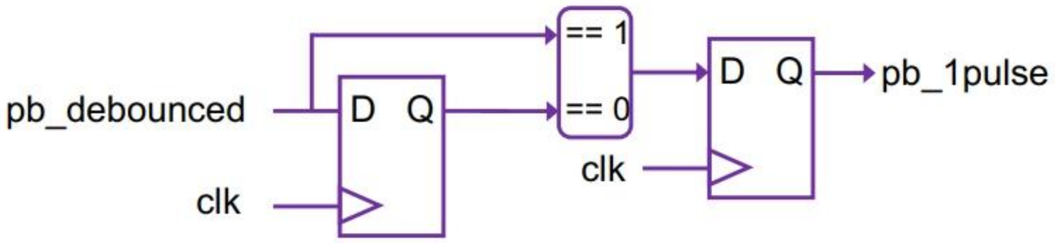
One Pulse:

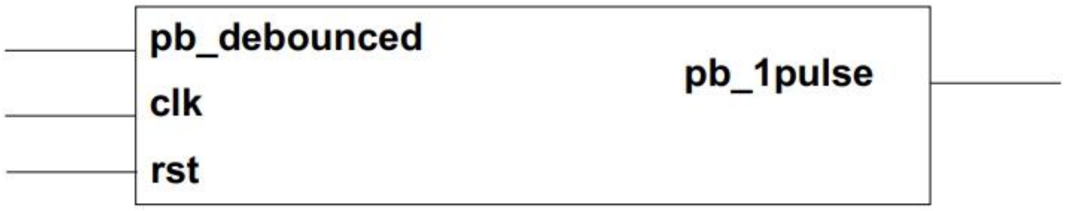
When a pushbutton is pressed for a short moment, the time to turn on the switch is usually much longer than one clock period, and that the circuit will see that we are supplying a string of ones as the input. Therefore, we also need to implement the one-pulse generator so that it can generate only a one-clock-period-long pulse every time a pushbutton is pressed.

The one_pulse module receives two inputs, pb_debounced and clk, and generates one output pb_one_pulse. We add one additional register namely pb_debounced_delay in the module. There is an always@(posedge clk) block and inside there is an if else statement. The if 's condition is when pb_debounced == 1'b1 and pb_debounced_delay == 1'b0, then pb_one_pulse will be assigned to 1'b1, else pb_one_pulse will be assigned to 1'b0. Lastly, outside the if else statement but inside the always block, pb_debounced_delay will be assigned to pb_debounced.


```
module one_pulse(pb_debounced, clk, pb_one_pulse);
    input pb_debounced;
    input clk;
    output pb_one_pulse;

    reg pb_one_pulse;
    reg pb_debounced_delay;

    always@(posedge clk)
    begin
        if(pb_debounced == 1'b1 & pb_debounced_delay == 1'b0)
            pb_one_pulse <= 1'b1;
        else
            pb_one_pulse <= 1'b0;

        pb_debounced_delay <= pb_debounced;
    end
endmodule
```

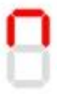
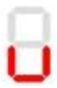
One pulse block diagram:

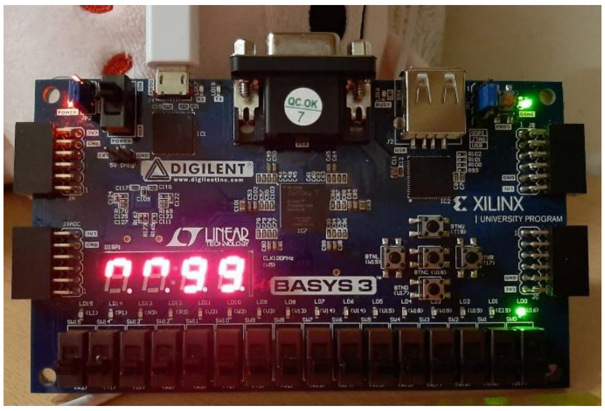


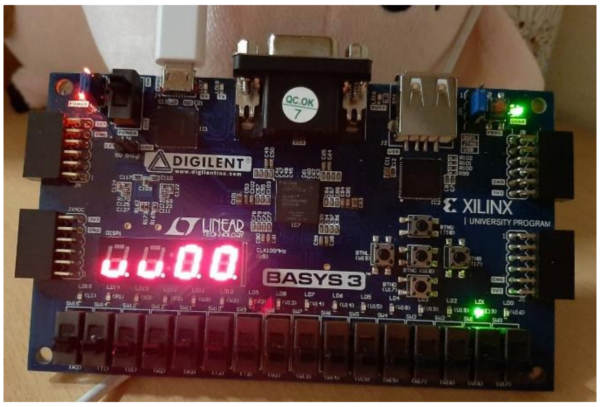


lab4_1

In this lab, we are required to design a 2-digit BCD up/down counter using seven segment display on the FPGA board which count up from 00 to 99, and vice versa, count down from 99 to 00, and when the counter reaches 99, it will stop and max will change from 0 to 1, which will light up the LED light LED0. The same thing for min, when the counter reaches 00, it will stop and min will be activated which will turn on LED1. The two rightmost digits on the seven segment display were reserved for the counter, meanwhile the two leftmost digits of the seven segment display are used to display the counting direction, which will either count up or count down.

Count up:  Count down: 





This counter is also able to perform pause and resume, also change the counting direction, and reset. The pause/resume, direction, and reset each were controlled by a button:

- BTNC for reset (rst): press to reset the counter to 00 and the direction to count up, and the counter will be in the pause mode initially.
- BTNU for resume/pause (en): press to stop counting (pause), press again to continue counting (resume).
- BTND for directions (dir): press to change the direction of counting (count up to count down, vice versa, count down to count up. Directions can also be changed while in pause mode.)

Before designing the fully complete counter along with the buttons, LED lights, and displaying counting direction, I first designed the original BCD counter which counts up from 00 to 99 and counts down from 99 to 00.

As per usual, I implemented the clock_divider module that divides the frequency of the input clock by 2**25 to get the output clock that will be used for the BCD counter. To design the BCD counter that displays on the board, I set the rightmost digit’s name as ones, the second rightmost digit as tens. For the last two leftmost digits I did not assign a name to it since it only takes one of either two cases: direction up or direction down.

If we look at how numbers count up, it will count up from ones first, until ones reaches 9, then the tens will increase by one. Using this way of thinking, I designed my counter up like this:

```
if(ones !== 4'd9)
begin
    ones <= ones + 4'd1;
    tens <= tens;
end

else if(ones == 4'd9 && tens !== 4'd9)
begin
    ones <= 4'd0;
    tens <= tens + 4'd1;
end
```

Using the same way of thinking, I designed my counter down like this:

```
if(ones !== 4'd0)
begin
    ones <= ones - 4'd1;
    tens <= tens;
end

else if(ones == 4'd0 && tens !== 4'd0)
begin
    ones <= 4'd9;
    tens <= tens - 4'd1;
end
```

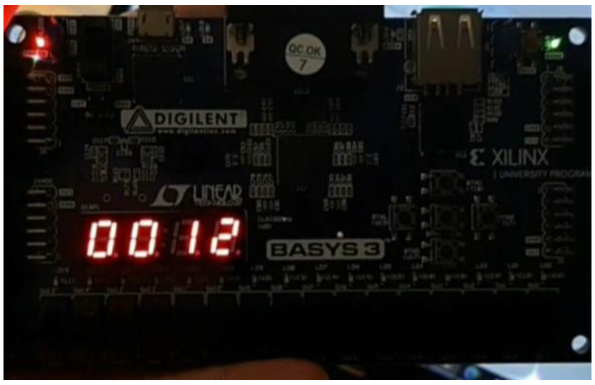
Lastly, if ones and tens both reaches 00 or 99, they will stop, so I assign each of their value to their value itself:

```
//reach 00 or 99
if((ones == 4'd9 && tens == 4'd9 && count_up == 1'b1) ||
(ones == 4'd0 && tens == 4'd0 && count_up == 1'b0))
begin
    ones <= ones;
    tens <= tens;
end
```

To assign the value of max and min, I used this method that if ones is equal to 4’d8 and tens is equal to 4’d9, then max will be assigned to 1’b1, and in all other cases, max will be assigned to 1’b0. Same thing for min, when ones is equal to 4’d1 and tens is equal to 4’d0, then min will be assigned to 1’b1, and in all other cases, min will be assigned to 1’b0. The light will light up at the correct timing (exactly when the counter hits 00 or 99) when the condition is like this, otherwise, the light will not light up, because I used the non-blocking assignment here.

```
if(ones == 4'd1 && tens == 4'd0 && count_up == 1'b0)
    min <= 1'b1;

else
    min <= 1'b0;


if(ones == 4'd8 && tens == 4'd9 && count_up == 1'b1)
    max <= 1'b1;

else
    max <= 1'b0;
```

Then I find a way to display all the digits simultaneously first without the counting direction and can be displayed on the four digits of the seven segment display on the FPGA board simultaneously, which looked like the picture below.



To display all four digits simultaneously, I need to use refresher here. The refresher I implemented here takes the two most significant bits of the 20-bit refresh_counter, which will increase by 20’d1 every time a positive edge of a clock occurs.

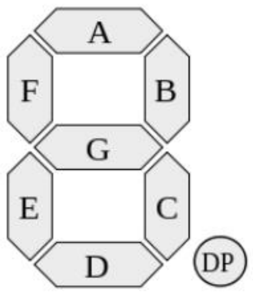
```
always@(posedge clk)
begin
    refresh_counter <= refresh_counter + 20'd1;
end

assign refresher = refresh_counter[19:18];
```

```
always@(*)
begin
    case(ACT_DIGIT) //digit
        4'd0: DISPLAY <= 7'b0000001;
        4'd1: DISPLAY <= 7'b1001111;
        4'd2: DISPLAY <= 7'b0010010;
        4'd3: DISPLAY <= 7'b0000110;
        4'd4: DISPLAY <= 7'b1001100;
        4'd5: DISPLAY <= 7'b0100100;
        4'd6: DISPLAY <= 7'b0100000;
        4'd7: DISPLAY <= 7'b0001111;
        4'd8: DISPLAY <= 7'b0000000;
        4'd9: DISPLAY <= 7'b0000100;
        4'd11: DISPLAY <= 7'b0011101; //up
        4'd12: DISPLAY <= 7'b1100011; //down
        default: DISPLAY <= 7'b1111111;
    endcase
end
```

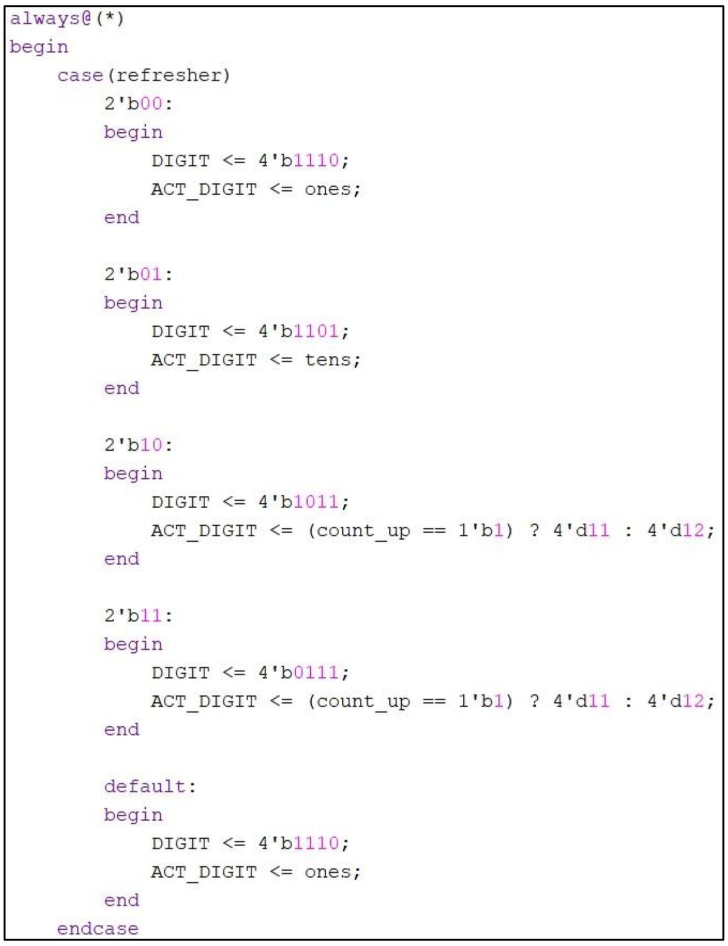
As for the numbers to appear correctly on the digits of the seven segment display, because this time to the binary digit 0 is used activate a single segment instead of 1, so I designed the case which will take ACT_DIGIT to determine which number to display on the digit, as shown below, and modified a bit of the constrains on this part for it to match with my main source code. The case is put inside an always@(*) block because this part will be used every time it is called.

Constraints for display, as referred from the lecture PPT:

```
# 7 segment display
 set_property PACKAGE_PIN W7 [get_ports {DISPLAY[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[6]}]
 set_property PACKAGE_PIN W6 [get_ports {DISPLAY[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[5]}]
 set_property PACKAGE_PIN U8 [get_ports {DISPLAY[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[4]}]
 set_property PACKAGE_PIN V8 [get_ports {DISPLAY[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[3]}]
 set_property PACKAGE_PIN U5 [get_ports {DISPLAY[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[2]}]
 set_property PACKAGE_PIN V5 [get_ports {DISPLAY[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[1]}]
 set_property PACKAGE_PIN U7 [get_ports {DISPLAY[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[0]}]
```



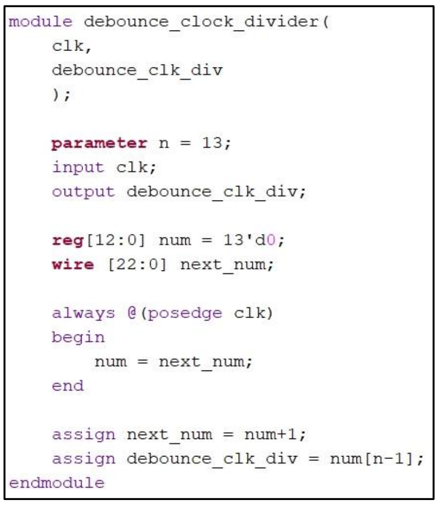
The ACT_DIGIT used as the case for the seven segment display is obtained from this part of code, which takes ones to be displayed on the rightmost digit (DIGIT <= 4’b1110), tens to be displayed on the 2nd rightmost digit (DIGIT <= 4’b1101), and either 4’d11 or 4’d12 (DIGIT <= 4’b1011) and (DIGIT <= 4’b0111), to fill in the two leftmost digits to match the case from ACT_DIGIT.



```
always@(*)
begin
    case(refresher)
        2'b00:
        begin
            DIGIT <= 4'b1110;
            ACT_DIGIT <= ones;
        end

        2'b01:
        begin
            DIGIT <= 4'b1101;
            ACT_DIGIT <= tens;
        end

        2'b10:
        begin
            DIGIT <= 4'b1011;
            ACT_DIGIT <= (count_up == 1'b1) ? 4'd11 : 4'd12;
        end

        2'b11:
        begin
            DIGIT <= 4'b0111;
            ACT_DIGIT <= (count_up == 1'b1) ? 4'd11 : 4'd12;
        end

        default:
        begin
            DIGIT <= 4'b1110;
            ACT_DIGIT <= ones;
        end
    endcase
```

For the case to determine which digit of the seven segment display will be activated, I used refresher which I have designed and was explained before, to display all the digits simultaneously. Then, as matched to how the digits were activated, I assigned ACT_DIGIT with the correct variables, which will be used as the case to activate the seven segment display correctly. For the last two cases, if count_up == 1’b1, then ACT_DIGIT will be assigned as 4’d11, which is the case to display direction up, and if count_up == 1’b0, which means it will count down, then we assign ACT_DIGIT as 4’d12, which is the case to display direction down. The case is put inside an always@(*) block because this part will be used every time it is called.
For the default I can assign it to anything since if the counter works perfectly the default case will not be used.

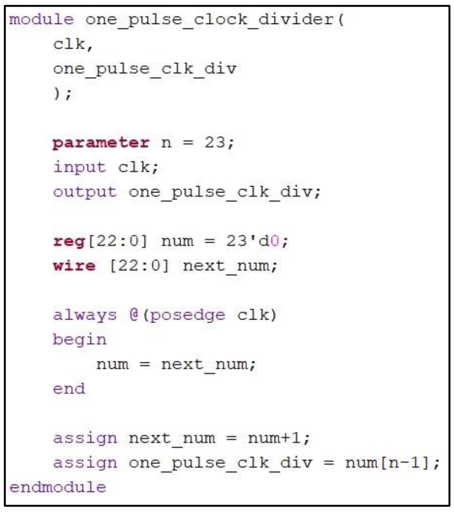
After settling the counter and displaying the counter part, it is time to work with the buttons. We need to make 3 buttons work for this lab, namely BTNC for rst (reset), BTNU for en (resume/pause), and BTND for dir (directions).



```
module debounce_clock_divider(
    clk,
    debounce_clk_div
    );

    parameter n = 13;
    input clk;
    output debounce_clk_div;

    reg[12:0] num = 13'd0;
    wire [22:0] next_num;

    always @(posedge clk)
    begin
        num = next_num;
    end

    assign next_num = num+1;
    assign debounce_clk_div = num[n-1];
endmodule
```

For the debounce module in lab4_1, we need to make a clock_divider module that divides the frequency of the input clock by 2**13 to get the output clock. This is used to prevent delay when pressing the button, or in other words, to avoid pressing the button in a longer time for it to generate a high output (1) and work properly.

For the one-pulse module, we also need to specially make a clock_divider module that divides the frequency of the input clock by 2**23 to get the output clock. This is used to prevent delay when pressing the button, or in other words, to avoid pressing the button in a longer time for it to generate a high output (1) and work properly.



```
module one_pulse_clock_divider(
    clk,
    one_pulse_clk_div
    );

    parameter n = 23;
    input clk;
    output one_pulse_clk_div;

    reg[22:0] num = 23'd0;
    wire [22:0] next_num;

    always @(posedge clk)
    begin
        num = next_num;
    end

    assign next_num = num+1;
    assign one_pulse_clk_div = num[n-1];
endmodule
```

(Debounce and one pulse explanation on page 1 to 3)

After finishing with the debounce and one_pulse module, we can look at the main module of lab4_1. I declared rst, en, dir as wire and DIGIT and DISPLAY as a reg of 4 bits and 7 bits respectively, also with max and min. I added many additional wires and registers to support this BCD counter namely:
For the wire:

- clk_div: divides the frequency of the input clock by 2**25 to get the output clock that will be used for the BCD counter.
- debounce_clk_div: divides the frequency of the input clock by 2**13 to get the output clock that will be used for the debounce module.
- one_pulse_clk_div: divides the frequency of the input clock by 2**23 to get the output clock that will be used for the debounce module.
- rst_debounced: rst generated from the debounce module that will be passed to the one_pulse module.
- en_debounced: en generated from the debounce module that will be passed to the one_pulse module.
- dir_debounced: dir generated from the debounce module that will be passed to the one_pulse module.
- rst_one_pulse: rst generated from the one-pulse module which indicates that BTNC is pressed or not. If BTNC is pressed, then rst_one_pulse will become 1’b1. Otherwise, if BTNC is at rest, then the value of rst_one_pulse is 1’b0.
- en_one_pulse: en generated from the one-pulse module which indicates that BTNU is pressed or not. If BTNU is pressed, then en_one_pulse will become 1’b1. Otherwise, if BTNU is at rest, then the value of en_one_pulse is 1’b0.

- dir_one_pulse: dir generated from the one-pulse module which indicates that BTND is pressed or not. If BTND is pressed, then dir_one_pulse will become 1’b1. Otherwise, if BTND is at rest, then the value of dir_one_pulse is 1’b0.
- refresher: takes the value of the two most significant bits of the 20-bit refresh_counter.

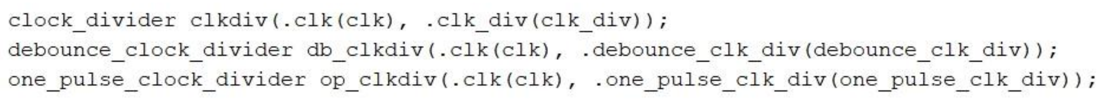
For the reg:

- refresh_counter: uses 20 bits, increase by 20’d1 every time a positive edge of a clock occurs, and the two most significant bits will then be assigned to refresher.
- ones: uses 4 bits, the variable that is used to represent the rightmost digit of the seven segment display on the FPGA board.
- tens: uses 4 bits, the variable that is used to represent the second rightmost digit of the seven segment display on the FPGA board.
- ACT_DIGIT: takes the value of ones, tens, and either 4’d11 or 4’d12 that will be used as the case for displaying the seven segment display.
- resume: indicates that the counter resumes counting when the value is 1’b1 and pause mode when the value is 1’b0. The value changes every time BTNU is pressed, or in other words, en_one_pulse == 1’b1.
- count_up: indicates counting up or counting down. The counter will count up when the value of count_up == 1’b1 and counts down when the value of count_up == 1’b0. The value of count_up changes every time BTND is pressed, or in other words, dir_one_pulse == 1’b1. count_up will also indicate whether the two leftmost digits of the seven segment display will show count up or count down direction. It will show count up direction when count_up == 1’b1 and it will show count down direction when count_up == 1’b0.

Since I have already explained about how I designed the BCD counter and how to activate and display the counter on the seven segment display, all that is left to be explained is how to implement the buttons.

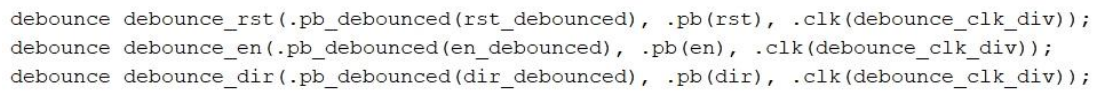
To implement the buttons, we need to use many instantiations:

- Three clock dividers for 2**25 which is used for the BCD counter, 2**13 which is used for the debounce, and 2**23 which is used for the one_pulse.



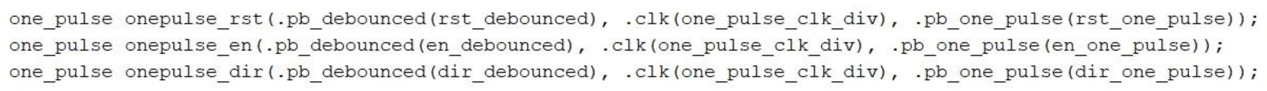
```
clock_divider clkdiv(.clk(clk), .clk_div(clk_div));
debounce_clock_divider db_clkdiv(.clk(clk), .debounce_clk_div(debounce_clk_div));
one_pulse_clock_divider op_clkdiv(.clk(clk), .one_pulse_clk_div(one_pulse_clk_div));
```

- Debounce module instantiation each for rst, en, and dir.



```
debounce debounce_rst(.pb_debounced(rst_debounced), .pb(rst), .clk(debounce_clk_div));
debounce debounce_en(.pb_debounced(en_debounced), .pb(en), .clk(debounce_clk_div));
debounce debounce_dir(.pb_debounced(dir_debounced), .pb(dir), .clk(debounce_clk_div));
```

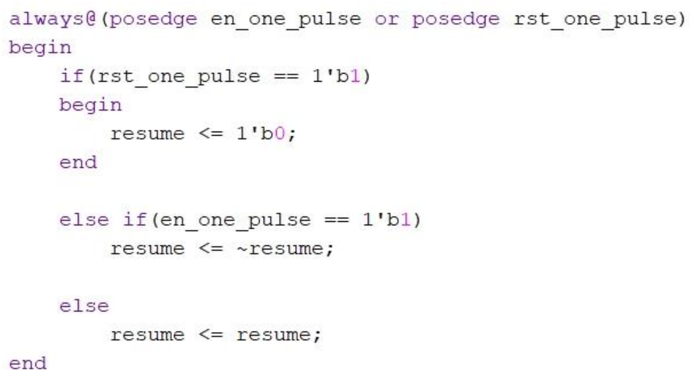
- One pulse module instantiation each for rst, en, and dir.



```
one_pulse onepulse_rst(.pb_debounced(rst_debounced), .clk(one_pulse_clk_div), .pb_one_pulse(rst_one_pulse));
one_pulse onepulse_en(.pb_debounced(en_debounced), .clk(one_pulse_clk_div), .pb_one_pulse(en_one_pulse));
one_pulse onepulse_dir(.pb_debounced(dir_debounced), .clk(one_pulse_clk_div), .pb_one_pulse(dir_one_pulse));
```

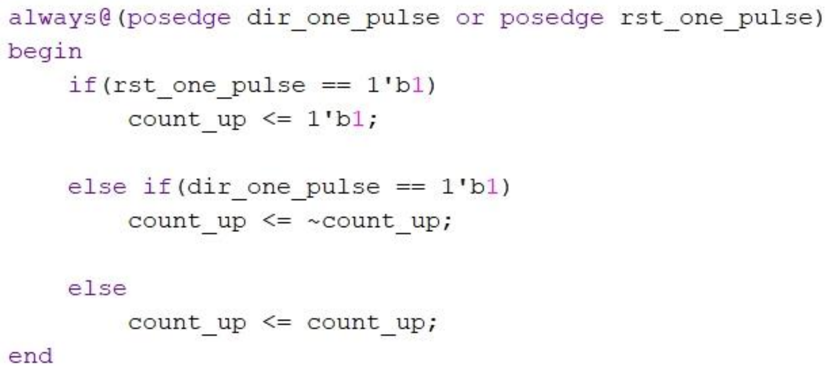
After getting the rst_one_pulse, en_one_pulse, and dir_one_pulse, we can work on the logic of the buttons. I used different always block for different buttons, but included rst_one_pulse in each always block because every time rst_one_pulse is activated, everything, including every variable are being reset back to its first place, in which different variables were put inside different always blocks because we cannot have one same variable inside different always blocks.

For the always@(posedge en_one_pulse or posedge rst_one_pulse) block, this will deal with the en button which is BTNU, and it will deal with resume. As stated in the PDF, when reset button is pressed then the counter will be in pause mode. Therefore if rst_one_pulse == 1'b1, then resume will become 1'b0. Two remaining conditions were if en_one_pulse is equal to 1'b1 or 1'b0. If en_one_pulse is equal to 1'b1, then resume will change its value by negating itself, changing the value from 1'b1 to 1'b0 (resume to pause) or from 1'b0 to 1'b1 (pause to resume). Else, if en_one_pulse == 1'b0, which means that BTNU is not pressed, then resume will stay at its current value.



```
always@(posedge en_one_pulse or posedge rst_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
    begin
        resume <= 1'b0;
    end

    else if(en_one_pulse == 1'b1)
        resume <= ~resume;

    else
        resume <= resume;
end
```

Next is the always@(posedge dir_one_pulse or posedge rst_one_pulse) block. This will deal with the dir button which is BTND, and it will deal with count_up. When the reset button is pressed then the counter go back to its initial state. For count_up, intially its state is to count up, so we set count up to 1’b1 during when reset occurs. Therefore if rst_one_pulse == 1’b1, then count_up will become 1’b1, which is activated. Two remaining conditions were if dir_one_pulse is equal to 1’b1 or 1’b0. If dir_one_pulse is equal to 1’b1, then count_up will change its value by negating itself, changing the value from 1’b1 to 1’b0 (count up to count down) or from 1’b0 to 1’b1 (count down to count up). Else, if dir_one_pulse == 1’b0, which means that BTND is not pressed, then count_up will stay at its current value.

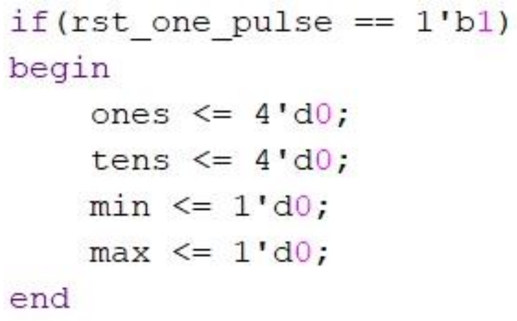


```
always@(posedge dir_one_pulse or posedge rst_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
        count_up <= 1'b1;

    else if(dir_one_pulse == 1'b1)
        count_up <= ~count_up;

    else
        count_up <= count_up;
end
```

For the BCD counter that we have implemented before but excluding buttons, we can finally add the logic for buttons in the BCD counter logic. Kindly refer to the description regarding the BCD counter above (starting from page 4) because in this part I will only explain about how I implemented the buttons and its logic.

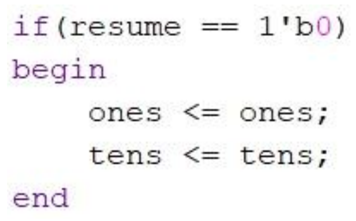
Firstly, aside from clk_div, we add rst_one_pulse to always@(posedge clk_div). Then inside the always block, we first deal with the reset before dealing with the counter itself. There will be a big if else statement, one for when rst_one_pulse == 1'b1 and the other one when rst_one_pulse == 1'b0.

When reset occurs (rst_one_pulse == 1'b1), the counter along with max and min will be set to 1’b0. Therefore, if rst_one_pulse == 1’b1, then ones, tens, min, and max will be assigned to 0.



```
if(rst_one_pulse == 1'b1)
begin
    ones <= 4'd0;
    tens <= 4'd0;
    min <= 1'd0;
    max <= 1'd0;
end
```

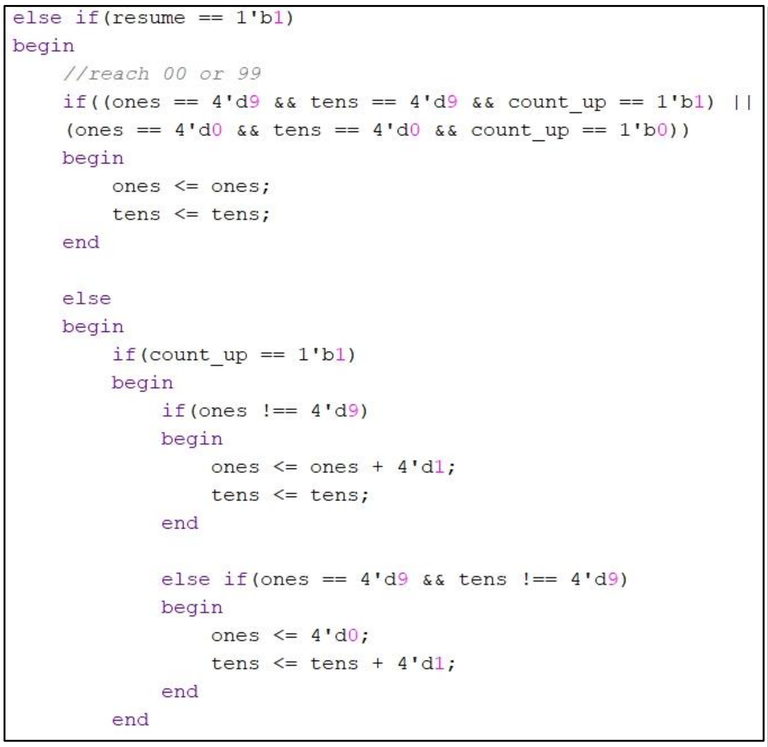
The else statement would be referring to when rst_one_pulse == 1'b0. Inside here is when the BCD counter works. We still need to make an if else statement for when the BCD counter is in resume or pause mode, which is controlled by BTNU.



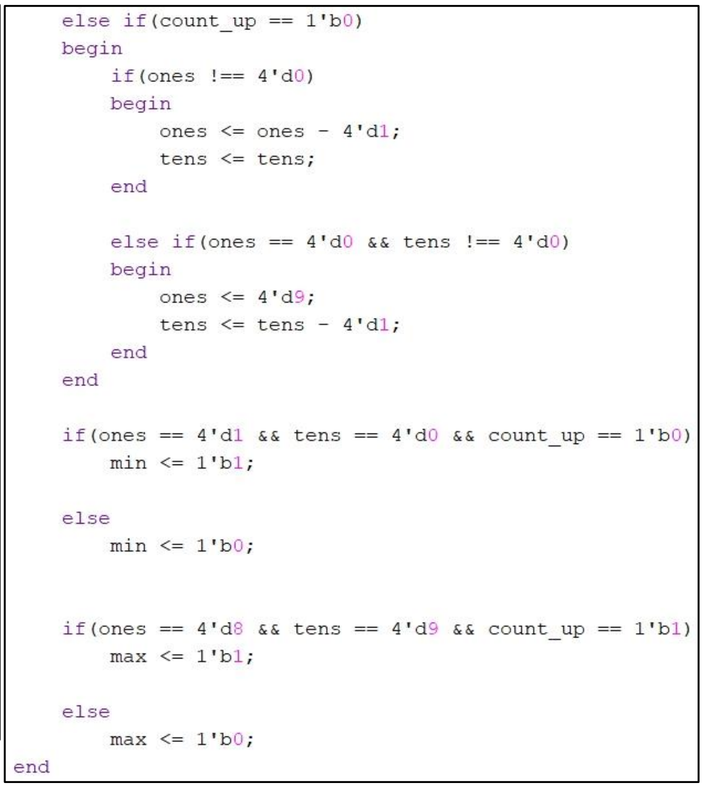
```
if(resume == 1'b0)
begin
    ones <= ones;
    tens <= tens;
end
```

If the BCD counter is in pause mode, which is resume == 1’b0, then ones and tens values will remain the same. Therefore, we can assign ones and tens as its own value.

Else, if resume == 1’b1, which means that the BCD counter is in resume mode and it starts counting, then we can start working with the BCD counter. We only need to put what is done regarding the BCD counter inside the else if(resume == 1’b1) statement, and then assign min to the value 1’b1 if the counter reaches 00, else min will be 1’b0, and assign max to be 1’b1 when the counter reaches 99, else max will be 1’b0.



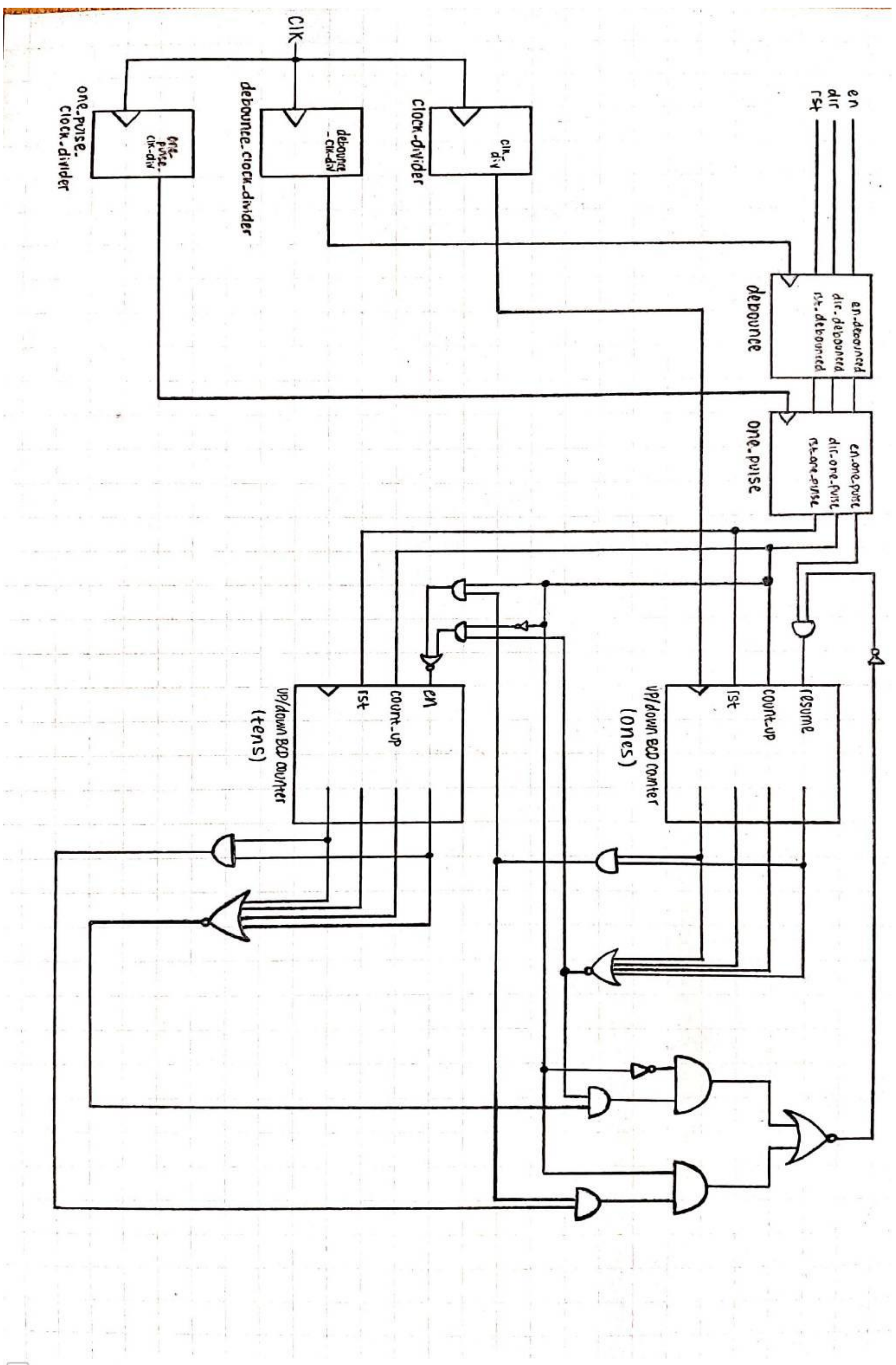
```
else if(resume == 1'b1)
begin
    //reach 00 or 99
    if((ones == 4'd9 && tens == 4'd9 && count_up == 1'b1) ||
    (ones == 4'd0 && tens == 4'd0 && count_up == 1'b0))
    begin
        ones <= ones;
        tens <= tens;
    end

    else
    begin
        if(count_up == 1'b1)
        begin
            if(ones !== 4'd9)
            begin
                ones <= ones + 4'd1;
                tens <= tens;
            end

            else if(ones == 4'd9 && tens !== 4'd9)
            begin
                ones <= 4'd0;
                tens <= tens + 4'd1;
            end
        end
```



```
        else if(count_up == 1'b0)
        begin
            if(ones !== 4'd0)
            begin
                ones <= ones - 4'd1;
                tens <= tens;
            end

            else if(ones == 4'd0 && tens !== 4'd0)
            begin
                ones <= 4'd9;
                tens <= tens - 4'd1;
            end
        end

        if(ones == 4'd1 && tens == 4'd0 && count_up == 1'b0)
            min <= 1'b1;

        else
            min <= 1'b0;


        if(ones == 4'd8 && tens == 4'd9 && count_up == 1'b1)
            max <= 1'b1;

        else
            max <= 1'b0;
end
```

(these two pictures are related to each other)

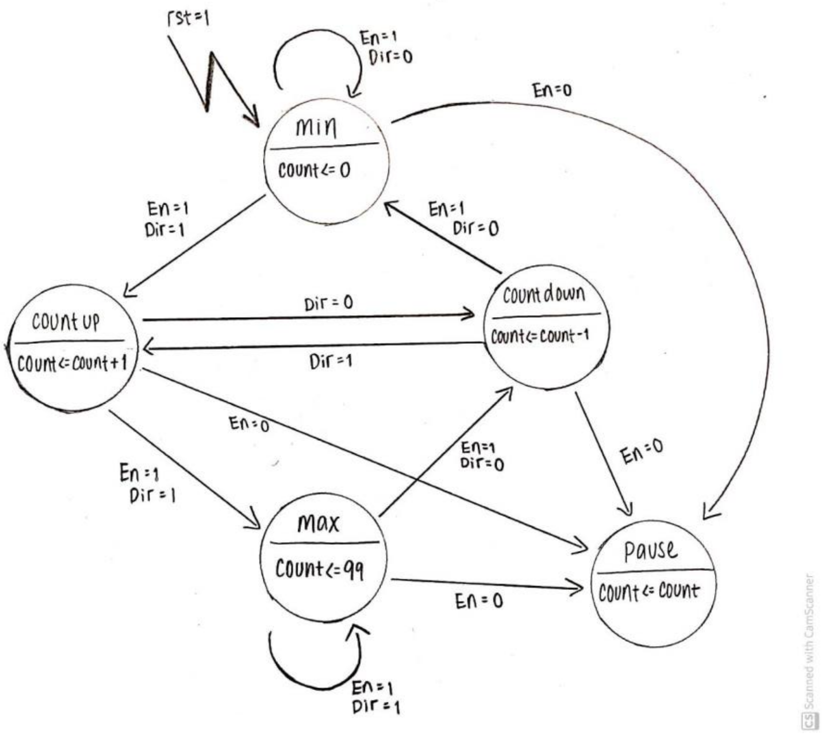
The entire code for the else if(resume == 1’b1) somehow looked like this. For max and min, I assigned their values to be 1’b1 when ones == 1’d1 && tens == 4’d0 and when ones == 4’d8 && tens == 4’d9 because the light will light up at the correct timing (exactly when the counter hits 00 or 99) when the condition is like this, otherwise, the light will not light up, because I used the non-blocking assignment here.


clk
clk_div
clock_divider
debounce_clk_div
debounce_clock_divider
one_pulse_clk_div
one_pulse_clock_divider
en
dir
rst
en_debounced
dir_debounced
rst_debounced
debounce
en_one_pulse
dir_one_pulse
rst_one_pulse
one_pulse
resume
count_up
rst
up/down BCD counter (ones)
en
count_up
rst
up/down BCD counter (tens)

The first part of the block diagram, is the clock and the clock dividers. We are provided with the input clk and I implemented the clock divider module to get the frequency of 2**13, 2**23, and 2**25 for debounce, one pulse, and the counter itself. Which is why the input clk goes into the block diagram of clock_divider, debounce_clock_divider, and one_pulse_clock_divider and generates the output clk_div, debounce_clk_div, and one_pulse_clk_div respectively. Next is for the inputs en, dir, and rst that needs to go through the debounce and one_pulse module, along with the clocks, as shown on the picture above. From the debounce module, each of them will then become en_debounced, dir_debounced, and rst_debounced, which will then directly go into the one_pulse module. From the one_pulse module, the outputs we finally have which can be used for the main module will be en_one_pulse, dir_one_pulse, and rst_one_pulse. En_one_pulse here will be the resume, which will determine that the counter will be in resume or pause mode, while dir_one_pulse will be the count_up that will indicate the counting direction (1 for counting up, 0 for counting down), and lastly we have rst_one_pulse that will be the rst here, and it will reset everything to the initial condition if rst == 1. We then have two up down BCD counters, one is for the ones (first digit), and the other one is for the tens (second digit). We can directly connect dir_one_pulse to both count_up inputs in both of the counter block diagrams, and rst_one_pulse to both rst inputs of the two block diagrams. As for the up down counter, it will generate the numbers in binary, and we have two cases here:

When the counter is counting up (count_up == 1), when the ones is equal to 9 (1001 in binary), then it will enable the tens BCD counter, so the tens will start counting. Other than that, the tens will stop counting. We take the MSB and LSB which are both 1 when the counter reaches 9, and then we put them inside an AND gate to generate 1 only when both of their values are equal to 1, then connect them to the en of the tens counter. But here we also need to consider the fact that the counter is able to perform count down. So we need to take the case when the ones are all equal to 0 for the tens counter to be enabled. So we take all digits of the binary and put them inside an XOR gate to generate 1 only when all the digits are equal to 0. Then we need to put this output with the negated count_up since count_up will be 0 when the counter is counting down, into an AND gate. Then we put count_up directly into an AND gate along with the output from putting LSB and MSB of ones in an or gate, since the count_up is 1 during this state of counting up. Then we put both outputs into an XOR gate since only one that should be activated to enable the tens counter.

We do the same by connecting the MSB and LSB of the output using an AND gate, also all digits of the binary number into an XOR gate. This time, it is used to determine whether we have reached max or min. After doing so, we connect the result from the AND gate to the result of the AND gate from the ones counter, and put them in another AND gate, since if both of them reaches 9, it indicates that the counter has reached 99. We do the same thing for when both counters reach 0. We then need to check if the count_up is at the correct value or not when the counter reaches 99 or 00. So what we have to do when both counters reach 9 is to connect the output resulted by putting MSB and LSB to the AND gate to another AND gate with count_up, and they will generate 1 when both of their values are 1. The same thing goes for 0, but we need to negate the count_up first since when counting down, the value of count_down will be 0. We then put both outputs generated to an XOR gate since only one value out of those two should be one in order for the counter to be correct. Then, we just need to negate it and AND it with en_one_pulse, since when the counter reaches 00 or 99, the last output from the XOR gate will be 1. After we negate it, the value will become 0 and after we AND it with en_one_pulse, in which the value of en_one_pulse is 1 because we did not perform pause in this case. The counter will stop automatically since 1 AND 0 results in 0.



This state diagram clearly shows the flow of the counter and how it works, on what condition it will count up, count down, pause or resume, reset, and what it will do when it reaches min or max.

lab4_2

In this lab4_2 we are required to implement a stopwatch that counts the 0.1 seconds and has the record function, up to two records. Any further records will not be recorded. The default record is 0:00.0. The stopwatch is facilitated with pause and resume mode and will count up to 2:00.0. When in pause mode, the user can check their records.

The stopwatch uses buttons to control reset, enable (resume/pause), record, and switches to control display_1 and display_2. BTNC for reset (rst): press to reset the counter to 00 and the direction to count up, and the counter will be in the pause mode initially.

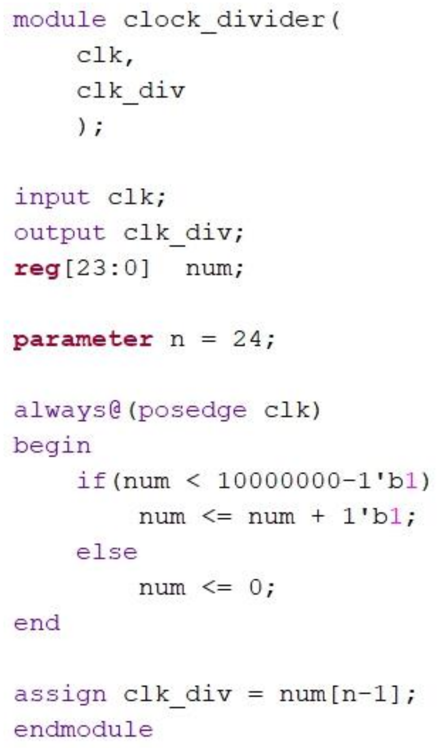
- BTNC for reset (rst): press to reset the stopwatch and records to 0:00.0, and the stopwatch will be in the pause mode initially.
- BTNU for resume/pause (en): press to stop counting (pause), press again to continue counting (resume).
- BTNR for record (record): press to record the current time when the stopwatch is counting.
- SW0 for display_1: show the first time record on the seven segment display when the stopwatch is either paused or finished counting to 2:00.0. If both SW0 and SW1 are slid up, display---- on the seven segment display.
- SW1 for display_2: show the second time record on the seven segment display when the stopwatch is either paused or finished counting to 2:00.0. If both SW0 and SW1 are slid up, display---- on the seven segment display.

This time, in order to fulfill the bonus question, I implemented the clock_divider module to get the clock frequency of 10Hz from 100MHz (the original clock frequency form the FPGA board) to get the output clock that will be used. I decided on 10 Hz because the stopwatch needs to count at the exact 0.1-second timestep. Using the formula of frequency and time, which is

frequency = 1 / (periodic time)

Therefore, frequency = 1/0.1 = 10 Hz.

To get 10Hz from 100MHz, we need to divide by 10,000,000.



```
module clock_divider(
    clk,
    clk_div
    );

input clk;
output clk_div;
reg[23:0]  num;

parameter n = 24;

always@(posedge clk)
begin
    if(num < 10000000-1'b1)
        num <= num + 1'b1;
    else
        num <= 0;
end

assign clk_div = num[n-1];
endmodule
```

The clock_divider itself takes one input clk and generates one output clk_div. I added one additional register num of 24 bits because 10,000,000 in binary numbers takes up to 24 bits, and I set the parameter n to 24. Then inside the always@(posedge clk) block, there is an if else statement, that if num is smaller than 10,000,000-1’b1, then num will be assigned to num + 1’b1, else, num will be assigned to 0. Outside the always block, I assigned clk_div as the value of num[n-1]. Therefore, we will have divided the clock by 10,000,000.

Before designing the fully complete stopwatch along with the buttons, LED lights, and switches, I first designed the original stopwatch which counts up to 2:00.0. To design the stopwatch itself, I need four variables to represent each digit of the seven segment display on the FPGA board, namely digit 4 (1 min) for the leftmost digit, digit3 (10 s) for the second leftmost digit, digit2 (1 s) for the second rightmost digit, and digit1 (0.1 s) for the rightmost digit. Digit 1 can count up to ten, and so as digit2. However, digit3 can only count up to 5, and both digit2 and digit1 should both be 9, so that digit1 can increase its value by 1. Until when digit1 reaches 2, then every digit will be assigned to their own value. Using this method, I implemented the stopwatch counter:

```
else if(digit4 != 4'd2)
begin
    if(digit1 != 4'd9)
    begin
        digit4 <= digit4;
        digit3 <= digit3;
        digit2 <= digit2;
        digit1 <= digit1 + 4'd1;
    end

    else if(digit1 == 4'd9)
    begin
        if(digit2 != 4'd9)
        begin
            digit4 <= digit4;
            digit3 <= digit3;
            digit2 <= digit2 + 4'd1;
            digit1 <= 4'd0;
        end

        if(digit2 == 4'd9)
        begin
            if(digit3 != 4'd5)
            begin
                digit4 <= digit4;
                digit3 <= digit3 + 4'd1;
                digit2 <= 4'd0;
                digit1 <= 4'd0;
            end
```

```
    else if(digit1 == 4'd9)
    begin
        if(digit2 != 4'd9)
        begin
            digit4 <= digit4;
            digit3 <= digit3;
            digit2 <= digit2 + 4'd1;
            digit1 <= 4'd0;
        end

        if(digit2 == 4'd9)
        begin
            if(digit3 != 4'd5)
            begin
                digit4 <= digit4;
                digit3 <= digit3 + 4'd1;
                digit2 <= 4'd0;
                digit1 <= 4'd0;
            end

            else if(digit3 == 4'd5)
            begin
                digit4 <= digit4 + 4'd1;
                digit3 <= 4'd0;
                digit2 <= 4'd0;
                digit1 <= 4'd0;
            end
        end
    end
end
```

After successfully implementing the stopwatch counter, I tried displaying the stopwatch on the seven segment display. Same thing as lab4_1, in order to display all four digits simultaneously, I need to use refresher here. The refresher I implemented here also takes the two most significant bits of the 20-bit refresh_counter, which will increase by 20’d1 every time a positive edge of a clock occurs.

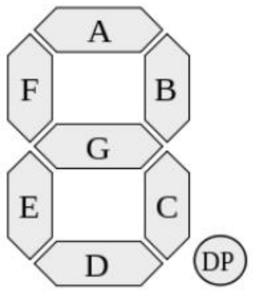
```
always@(posedge clk)
begin
    refresh_counter <= refresh_counter + 20'd1;
end

assign refresher = refresh_counter[19:18];
```

```
always@(*)
begin
    case(ACT_DIGIT) //digit
        4'd0: DISPLAY = 7'b0000001;
        4'd1: DISPLAY = 7'b1001111;
        4'd2: DISPLAY = 7'b0010010;
        4'd3: DISPLAY = 7'b0000110;
        4'd4: DISPLAY = 7'b1001100;
        4'd5: DISPLAY = 7'b0100100;
        4'd6: DISPLAY = 7'b0100000;
        4'd7: DISPLAY = 7'b0001111;
        4'd8: DISPLAY = 7'b0000000;
        4'd9: DISPLAY = 7'b0000100;
        4'd11: DISPLAY = 7'b1111110; //----
        default: DISPLAY = 7'b1111111;
    endcase
end
```

As for the numbers to appear correctly on the digits of the seven segment display, because this time to the binary digit 0 is used activate a single segment instead of 1, so I designed the case which will take ACT_DIGIT to determine which number to display on the digit, as shown below, and modified a bit of the constrains on this part for it to match with my main source code. The case is put inside an always@(*) block because this part will be used every time it is called.

Constraints for display, as referred from the lecture PPT:

```
# 7 segment display
 set_property PACKAGE_PIN W7 [get_ports {DISPLAY[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[6]}]
 set_property PACKAGE_PIN W6 [get_ports {DISPLAY[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[5]}]
 set_property PACKAGE_PIN U8 [get_ports {DISPLAY[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[4]}]
 set_property PACKAGE_PIN V8 [get_ports {DISPLAY[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[3]}]
 set_property PACKAGE_PIN U5 [get_ports {DISPLAY[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[2]}]
 set_property PACKAGE_PIN V5 [get_ports {DISPLAY[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[1]}]
 set_property PACKAGE_PIN U7 [get_ports {DISPLAY[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[0]}]
```



The ACT_DIGIT used as the case for the seven segment display is obtained from this part of code, which takes digit1 to be displayed on the rightmost digit (DIGIT <= 4’b1110), digit2 to be displayed on the 2nd rightmost digit (DIGIT <= 4’b1101), digit3 to be displayed on the 2nd leftmost digit (DIGIT <= 4’b1011), digit4 to be displayed on the leftmost digit (DIGIT <= 4’b0111).

For the case to determine which digit of the seven segment display will be activated, I used refresher which I have designed and was explained before, to display all the digits simultaneously. Then, as matched to how the digits were activated, I assigned ACT_DIGIT with display_digit1 for the first digit (rightmost seven segment), display_digit2 for the second digit, etc., along with the correct conditions, since we have a few conditions here, which will be used as the case to activate the seven segment display correctly. There is one set of if else statements with 1 if and 4 else ifs.

The first if is when the user wants to show the first recorded time of the stopwatch, that is, the condition would be if display_1 is slid up (display_1 == 1’b1) and both display_2 and resume should be 1’b0, since display_2 is not slid up and the stopwatch is in pause mode. The other condition that satisfies is when display_1 is slid up (display_1 == 1’b1) and display_2 should be 1’b0, since display_2 is not slid up and the stopwatch reaches 2:00.0, that is, digit4 == 4’d2. Under these conditions, display_digit1 will be assigned to the value of record1_digit1) the variable representing the first recorded time).

The second if else is when the user wants to show the second recorded time of the stopwatch. The conditions would be pretty much the same as when the user wants to show the first recorded time, that is, the condition would be if display_2 is slid up (display_2 == 1’b1) and both display_1 and resume should be 1’b0, since display_1 is not slid up and the stopwatch is in pause mode. The other condition that satisfies is when display_2 is slid up (display_2 == 1’b1) and display_1 should be 1’b0, since display_1 is not slid up and the stopwatch reaches 2:00.0, that is, digit4 == 4’d2. Under these conditions, display_digit1 will be assigned to the value of record2_digit1) the variable representing the second recorded time).

The third else if is when both switches (display_1 and display_2) are switched up simultaneously or at the same time. The condition for this is when both display_1 and display_2 is equal to 1’b1 and resume is equal to 1’b0 (pause mode), or when both display_1 and display_2 is equal to 1’b1 and the stopwatch reaches 2:00.0, that is, digit4 == 4’d2. Under these conditions, display_digit1 will be assigned to 4’d11 (-).

The last condition would be when the stopwatch is currently counting or when it is in pause mode without sliding up display_1 and display_2. The conditions would be when resume == 1’b1 and both display_1 and display_2 == 1’b1, or when resume == 1’b1 and one of display_1 or display_2 is equal to 1’b1 and the other one is equal to 1’b0, or when resume == 1’b1 and both display_1 and display_2 == 1’b1, or when resume == 1’b0 and both display_1 and display_2 are not slid up. Under these conditions, display_digit1 will be assigned to the value of digit1.

I did this exact same thing for the rest three digits (digit2, digit3, digit4), just changing the variable names according to their number (display_digit2 for digit2, etc.).

After settling the stopwatch and displaying it correctly on the seven segment display, it is time to work with the buttons. We need to make 3 buttons work for this lab, namely BTNC for rst (reset), BTNU for en (resume/pause), and BTNR for record (record).

For the one-pulse module, I also used the same clock_divider module as the one used for stopwatch counter, the clock divider that divides the frequency of the input clock by 10,000,000 to get the output clock frequency of 10Hz. This is the same clock we used for the stopwatch counter and the reason we used this clock is to prevent delay when pressing the button, or in other words, to avoid pressing the button in a longer time for it to generate a high output (1) and work properly.

(Debounce and one pulse explanation on page 1 to 3)

After finishing with the debounce and one_pulse module, we can look at the main module of lab4_2.

For the stopwatch counter that we have implemented before but excluding buttons, we can finally add the logic for buttons in the stopwatch logic. Kindly refer to the description regarding the stopwatch above (starting from page 18) because in this part I will only explain about how I implemented the buttons and its logic.

I declared rst, en, record, display_1, and display_2 as wire and DIGIT and DISPLAY as a reg of 4 bits and 7 bits respectively. I added many additional wires and registers to support this BCD counter namely:
For the wire:

- clk_div: divides the frequency of the input clock by 10,000,000Hz (10MHz) to get the output clock that will be used for the stopwatch.
- rst_debounced: rst generated from the debounce module that will be passed to the one_pulse module.
- en_debounced: en generated from the debounce module that will be passed to the one_pulse module.
- record_debounced: record generated from the debounce module that will be passed to the one_pulse module.
- rst_one_pulse: rst generated from the one-pulse module which indicates that BTNC is pressed or not. If BTNC is pressed, then rst_one_pulse will become 1’b1. Otherwise, if BTNC is at rest, then the value of rst_one_pulse is 1’b0.

- en_one_pulse: en generated from the one-pulse module which indicates that BTNU is pressed or not. If BTNU is pressed, then en_one_pulse will become 1’b1. Otherwise, if BTNU is at rest, then the value of en_one_pulse is 1’b0.
- record_one_pulse: record generated from the one-pulse module which indicates that BTNR is pressed or not. If BTNR is pressed, then record_one_pulse will become 1’b1. Otherwise, if BTNR is at rest, then the value of record_one_pulse is 1’b0.
- refresher: takes the value of the two most significant bits of the 20-bit refresh_counter.

For the reg:

- digit1, digit2, digit3, digit4: each uses 4 bits, used to represent the digits on the seven segment display from leftmost side to rightmost side.
- display_digit1, display_digit2, display_digit3, display_digit4: each uses 4 bits, used to pass the value of the digits to be displayed on the seven segment display from leftmost side to rightmost side.
- record1_digit1, record1_digit2, record1_digit3, record1_digit4: each uses 4 bits, to store the value of each digit when the first record is performed.
- record2_digit1, record2_digit2, record2_digit3, record2_digit4: each uses 4 bits, to store the value of each digit when the second record is performed.
- ACT_DIGIT: takes the value of digit1, digit2, digit3, and digit4 that will be used as the case for displaying the seven segment display.
- refresh_counter: uses 20 bits, increase by 20’d1 every time a positive edge of a clock occurs, and the two most significant bits will then be assigned to refresher.
- resume: indicates that the counter resumes counting when the value is 1’b1 and pause mode when the value is 1’b0. The value changes every time BTNU is pressed, or in other words, en_one_pulse == 1’b1.
- recorded1: works as a flag to indicate that the first time is already recorded. If the first time is recorded, that is, BTNR is pressed for the first time (record_one_pulse == 1’b1), recorded1 will change its value to 1’b1.
- recorded2: works as a flag to indicate that the second time is already recorded. If the second time is recorded, that is, BTNR is pressed for the second time (record_one_pulse == 1’b1), recorded2 will change its value to 1’b1.

Since I have already explained about how I designed the stopwatch and how to activate and display the stopwatch on the seven segment display, all that is left to be explained is how to implement the buttons.

To implement the buttons, we need to use a few instantiations:

- One clock divider that divides the clock frequency by 10MHz, which is used for the stopwatch, debounce, and one_pulse module.

```
clock_divider clkdiv(.clk(clk), .clk_div(clk_div));
```

- Debounce module instantiation each for rst, en, and record.

```
debounce debounce_rst(.pb_debounced(rst_debounced), .pb(rst), .clk(clk));
debounce debounce_en(.pb_debounced(en_debounced), .pb(en), .clk(clk));
debounce debounce_record(.pb_debounced(record_debounced), .pb(record), .clk(clk));
```

- One pulse module instantiation each for rst, en, and record.

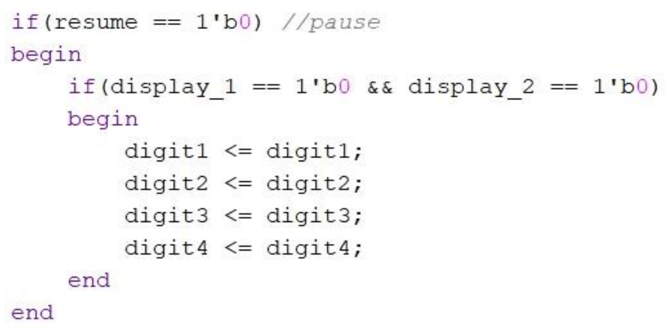
```
one_pulse onepulse_rst(.pb_debounced(rst_debounced), .clk(clk_div), .pb_one_pulse(rst_one_pulse));
one_pulse onepulse_en(.pb_debounced(en_debounced), .clk(clk_div), .pb_one_pulse(en_one_pulse));
one_pulse onepulse_record(.pb_debounced(record_debounced), .clk(clk_div), .pb_one_pulse(record_one_pulse));
```

After getting the rst_one_pulse, en_one_pulse, and record_one_pulse, we can work on the logic of the buttons. I used different always block for different buttons, but included rst_one_pulse in each always block because every time rst_one_pulse is activated, everything, including every variable are being reset back to its first place, in which different variables were put inside different always blocks because we cannot have one same variable inside different always blocks.

```
if(rst_one_pulse == 1'b1)
begin
    digit1 <= 4'd0;
    digit2 <= 4'd0;
    digit3 <= 4'd0;
    digit4 <= 4'd0;

    recorded1 <= 1'b0;
    recorded2 <= 1'b0;

    record1_digit1 <= 4'd0;
    record1_digit2 <= 4'd0;
    record1_digit3 <= 4'd0;
    record1_digit4 <= 4'd0;

    record2_digit1 <= 4'd0;
    record2_digit2 <= 4'd0;
    record2_digit3 <= 4'd0;
    record2_digit4 <= 4'd0;
end
```

For the always@(posedge en_one_pulse or posedge rst_one_pulse) block, this will deal with the en button which is BTNU, and it will deal with resume. As stated in the PDF, when reset button is pressed then the counter will be in pause mode. Therefore if rst_one_pulse == 1'b1, then resume will become 1'b0. The remaining condition is if en_one_pulse is equal to 1'b1. If en_one_pulse is equal to 1'b1, then resume will change its value by negating itself, changing the value from 1'b1 to 1'b0 (resume to pause) or from 1'b0 to 1'b1 (pause to resume).

For the always@(posedge clk_div or posedge rst_one_pulse) block, there is one set of big if else statement, for which the if condition will be if rst_one_pulse == 1’b1, then everything will be reset to its original default value, including the variables for digits, recorded digits, and the flag that indicates that a value is recorded. Else, it would be reset == 1’b0, and inside this else statement we would have the conditions for when resume == 1’b0 (pause mode) and resume == 1’b1 (resume mode). When resume == 1’b0, then each digit will hold its current value.



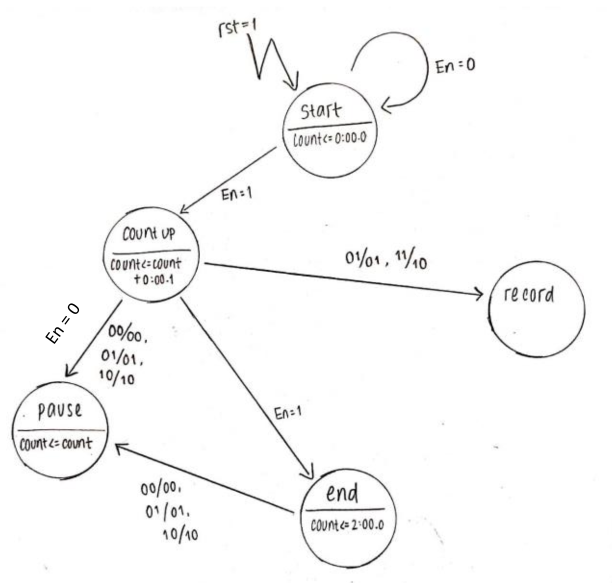
```
if(resume == 1'b0) //pause
begin
    if(display_1 == 1'b0 && display_2 == 1'b0)
    begin
        digit1 <= digit1;
        digit2 <= digit2;
        digit3 <= digit3;
        digit4 <= digit4;
    end
end
```

When resume == 1’b1, the stopwatch can perform record for the first and second time. The conditions for the first record would be when BTNR is pressed (record_one_pulse == 1’b1), and both recorded1 and recorded2 has the value 1’b0, which means that nothing was recorded yet. Inside this if condition, we assign 1’b1 to recorded1 and assign the current digit1, digit2, digit3, and digit4 to record1_digit1, record1_digit2, record1_digit3, and record1_digit4 respectively.

Then the conditions for the second record would be pretty much the same ad the first record, which is when BTNR is pressed (record_one_pulse == 1’b1), and recorded1 has the value 1’b1 since first record was already used, and recorded2 has the value 1’b0, which means that second record had not been used yet. Inside this if condition, we assign 1’b1 to recorded2 and assign the current digit1, digit2, digit3, and digit4 to record2_digit1, record2_digit2, record2_digit3, and record2_digit4 respectively.

```
//first record
if(record_one_pulse == 1'b1 && recorded1 == 1'b0 && recorded2 == 1'b0)
begin
    recorded1 <= 1'b1;

    record1_digit1 <= digit1;
    record1_digit2 <= digit2;
    record1_digit3 <= digit3;
    record1_digit4 <= digit4;
end

//second record
else if(record_one_pulse == 1'b1 && recorded1 == 1'b1 && recorded2 == 1'b0)
begin
    recorded2 = 1'b1;

    record2_digit1 <= digit1;
    record2_digit2 <= digit2;
    record2_digit3 <= digit3;
    record2_digit4 <= digit4;
end
```

Other than these two record conditions, the stopwatch will start counting up to 2:00.0 as previously explained.



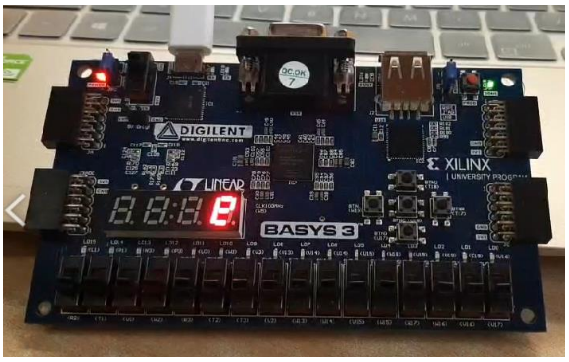
The stopwatch was initially at the start state, and when en is still 0, it stays at the start state and does not start counting. Only when en is equal to 1 when the stopwatch changes its state to count up and start counting, and the output will be count <= count + 1. While the stopwatch was still counting, the user can record the current time by pressing the record button. The first 01 indicates that the user records the first time, so recorded2 was still 0 and recorded1 is 1, and the output will be the first recorded time (record1 <= count), and 11 means that the user has pressed the button again so both slots have been used for recording, and the output will be 10 which indicates the second recorded time (record2 <= count). Then if the user presses pause, which means that en == 0, then the user can choose to show both the recorded time one by one, or not to show at all. If the user chooses not to show, it is indicated by 00, therefore the output itself will be 00 since no numbers are shown. If the user chooses to show the first recorded time, then it will be indicated by 01, and the output is 01 which indicated that the first recorded time is shown, and if the user chooses to show the second recorded time, then it is indicated by 10, and the output is 10 which indicates that the second recorded time is shown.

When the user is in count up state and en is still 1, then the stopwatch will count up to 2:00.0 and it will stop there, which is also the same as going to the pause state since the value stays the same, and we also need to go to the pause state to display the recorded time. So even though en is equal to 1 and the stopwatch has reached the end state, it will keep displaying the same number, and in every state, if we press the rst button, the stopwatch will automatically go to the start state.

2. 學到的東西與遇到的困難

<u>Displaying all four digits simultaneously on the seven segment display</u>

I was at first not sure on how to work with the buttons on the FPGA board, so I designed the counters first for both lab4_1 and lab4_2. Then I figured the way to display the digits on the seven segment display. At first, I only succeed in displaying the rightmost digit, and the first digit I displayed was 0, so I thought everything worked fine until I tried displaying other numbers and ended up getting an upside down number.



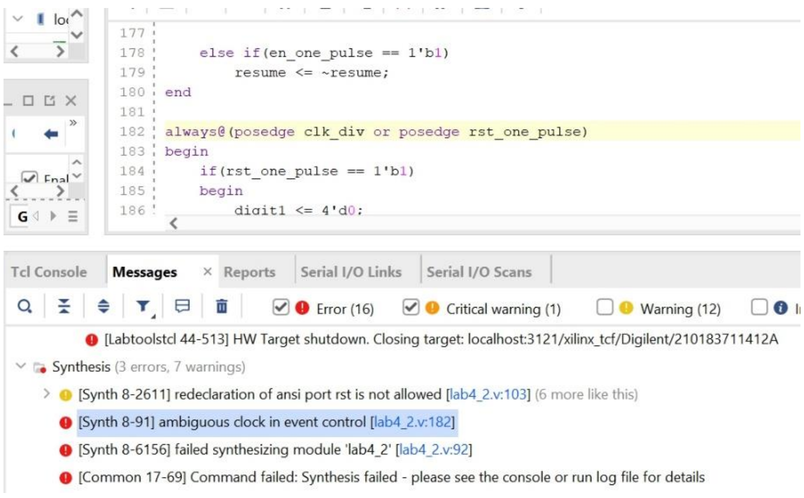
I then re-checked with the constraint and implemented it on my own until I displayed each numbers correctly.

After solving this problem, I need to figure out how to display all four digits on the seven segment display simultaneously, until I realized that I need to use a refresher. After looking at examples from the internet, I tried implementing a refresher and I finally succeed in displaying all four digits simultaneously.

Lesson learnt: Re-check whether your constraints match your main source module. Also learned how to make use of a refresher.

Ambiguous clock in event control

I encountered several times of ambiguous clock in event control. I asked the TA regarding this problem and he pointed out that I may not have one variable in two different always blocks. I tried fixing this and my problem was solved. There was one time too that I did not put an else after an if statement, and it results in ambiguous clock in event control too.



```
177
178     else if(en_one_pulse == 1'b1)
179         resume <= ~resume;
180 end
181
182 always@(posedge clk_div or posedge rst_one_pulse)
183 begin
184     if(rst_one_pulse == 1'b1)
185     begin
186         digit1 <= 4'd0;
```

```
Tcl Console | Messages | Reports | Serial I/O Links | Serial I/O Scans

[✓] Error (16)   [✓] Critical warning (1)   [ ] Warning (12)

[Labtoolstcl 44-513] HW Target shutdown. Closing target: localhost:3121/xilinx_tcf/Digilent/210183711412A
Synthesis (3 errors, 7 warnings)
  [Synth 8-2611] redeclaration of ansi port rst is not allowed [lab4_2.v:103] (6 more like this)
  [Synth 8-91] ambiguous clock in event control [lab4_2.v:182]
  [Synth 8-6156] failed synthesizing module 'lab4_2' [lab4_2.v:92]
  [Common 17-69] Command failed: Synthesis failed - please see the console or run log file for details
```

Lesson learnt: Do not have one variable in two different always blocks and arrange your if else statements neatly.

Button delay

Before, I need to press my button in lab4_1 for quite a long time for it to start working because I used the original clock from the FPGA board and a 2**25 clock for debounce and one_pulse. Me and my friend also asked the TA regarding this issue and they told us to fix the timing (clock divisor) for each debounce and one_ pulse. We tried several possible timing conditions for debounce and one_pulse and ended up with 2**13 and 2**23 which is the best one so far.

Lesson learnt: try and don’t stop finding the perfect timing for the buttons to work with as little to none delay as it could.

Debounce and One Pulse

I learnt a new concept of working with buttons on FPGA board with debounce and one pulse. I just knew that the buttons on the FPGA board contains a metal spring such that when the pushbutton is pressed and released, the switch contact will bounce several times before stabilizing and will generate a random number of unwanted signal pulses.

Lesson learnt: we need to make use of debounce to generate only a single pulse when pressing a button on the FPGA board, and one pulse to generate only a one-clock-period-long pulse every time a pushbutton is pressed.

3. 想對老師或助教說的話

This lab 04 had been quite challenging for me, but through this I have learnt a lot of things I have never learned before. The difficulty of the labs has increased gradually but in a good pace which allows us to have the time to learn the new things that are used in each labs. I hope that the pace will somehow be maintained like this so we would not have the difficulty to catch up to the teaching materials and what methods to use in each lab.