Khoa Quach

Parking Meter

1. Introduction

a. Background information

The project includes building a finite state machine to implement a parking meter Verilog. Finite State Machines (FSMs)are used to model sequential circuits and are used in many low level systems. There are two types of FSMs: Mealy and Moore. A Mealy FSM is one where the output depends on the present state and input, while for a Moore FSM, the output only depends on only the present state. In this project, we are instructed to implement a **parking meter** that takes input of adding different amounts of time with coins and incorporates a seven segment display as it would work with an actual Nexys3 Spartan 6 FPGA board.

b. Design requirements

The inputs of the system are:

Inputs	Function
clk	100Hz clock
rst	Reset to initial state
add1	Add 60 seconds
add2	Add 120 seconds
add3	Add 180 seconds
add4	Add 300 seconds
rst1	Reset time to 16 seconds
rst2	Reset time to 150 seconds

Four seven segment displays are used in this project. It is required to have some output module that will consists of:
led_seg: seven segment vector that displays the actual value fed to the 4 segments corresponding to the digits being displayed.
a1,a2,a3,a4: The anodes driving each of these segments(there are four seven segment displays) The seven segment displays are multiplexed with the anodes.
val1,val2,val3,val4: These four display the actual digit in the segments in BCD (binary coded decimal)

The output module is based on the Nexys3 Spartan6 configurations.

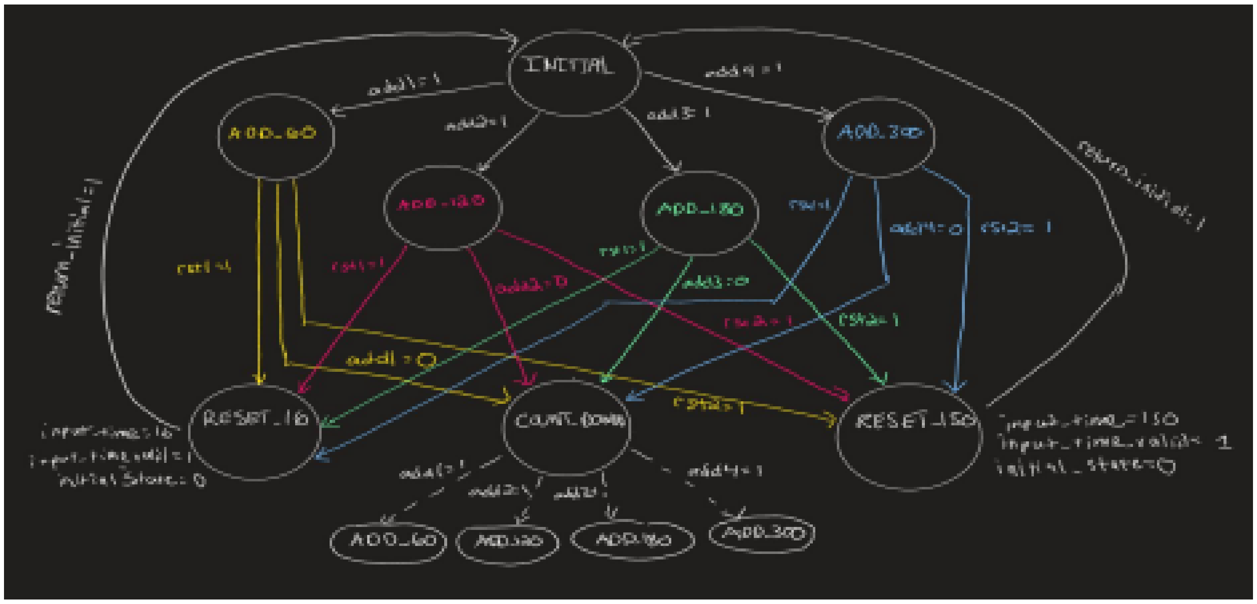
Here are some design requirements mentioned:

- In the initial state, the displays should be flashing 0000 with period 1 second and 50% duty cycle.
- The add buttons add the specific amount when pressed, and the timer starts to count down afterward at 1Hz
- When there are less than 180 seconds remaining, the display should flash with period 2 seconds and 50% duty cycle.
- When time runs out, it should also be flashing 0000 with period 1 and 50% duty cycle. ● The max value of the time displayed is 9999. If attempted to go above 9999 by the user, the counter should latch back to 9999 and start counting down from there

A FSM diagram is required to be drawn and explained.

2. Design Description

FSM Diagram



The 8 states are the following:

- INITIAL STATE (S0)
- ADD_60 (S1)
- ADD_120 (S2)

- ADD_180 (S3)
- ADD_300 (S4)
- COUNT_DOWN (S5) (starting counting down the time because no other inputs)
- RESET_16 (S6)
- RESET_150 (S7)

Transition analysis:

- S0 -> S1,S2,S3,S4 if add1= 1,add2 = 1,add3 = 1,add4 =1, respectively. Otherwise, loop back to the same state, this also covers rst case.
- S1 -> S5 if add1 = 0
- S1 -> S6 if rst1 = 1
- S1 -> S6 if rst2 =2
- The other S2,S3,S4 have similar transitions but with their add input index. ● S6 and S7 (reset 16and reset 150 states) go back to S0 (initial state) if return_initial is 1 ● The count down state S5:

  S5 -> S1 if add1 = 1, S5->S2 if add2 = 1, S5->S3 if add3 = 1, S5 -> S2 if add4 = 1

There are **internal variables** made within the code. Here are the purposes of these internal variables:

input_time : This signals holds the coin value captured from the FSM

return_initial = 1 (means in the FSM it should go to the initial state.)

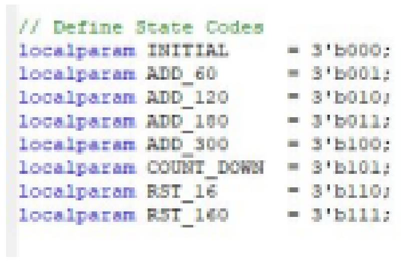
go_initial = 0 (state will remain in the same state.)

input_time_valid = 1 (if a new coin is captured by the FSM this signal will be 1). This signal is to show the validity of the current counter value.

input_time_captured = 1 (After capturing the current counter value from FSM, this will be set to 1. For example, in a case, where the driver is holding a 120 second coin for 5 seconds. In this case the FSM counter value will remain at 120 for 5 seconds; which we do not want to happen. Afterward, the timer section of the code will keep capturing 120 for 5 seconds. In the end the timer will be (120 x 5) which is incorrect. So after each correct insertion of a valid input, this signal will be set to 1 to avoid taking in the same counter value input again.

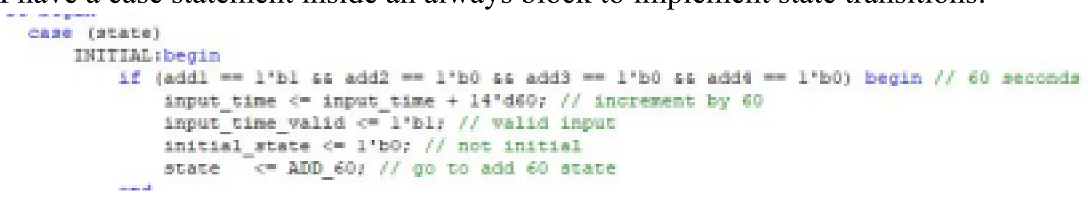
FSM Code Design

I created local parameters to define state numbers to have more readable code:



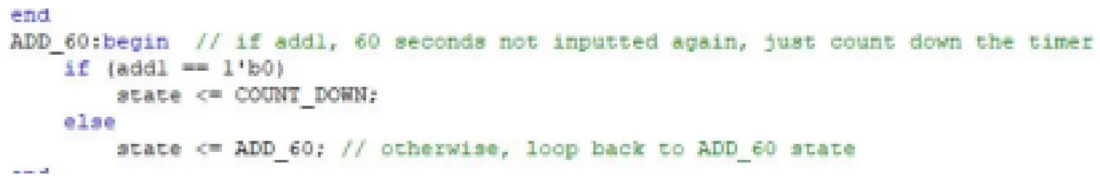
```
// Define State Codes
localparam INITIAL      = 3'b000;
localparam ADD_60       = 3'b001;
localparam ADD_120      = 3'b010;
localparam ADD_180      = 3'b011;
localparam ADD_300      = 3'b100;
localparam COUNT_DOWN   = 3'b101;
localparam RST_16       = 3'b110;
localparam RST_160      = 3'b111;
```

I have a case statement inside an always block to implement state transitions:



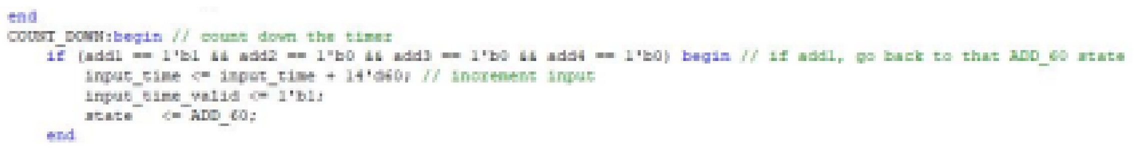
```
  case (state)
      INITIAL:begin
          if (add1 == 1'b1 && add2 == 1'b0 && add3 == 1'b0 && add4 == 1'b0) begin // 60 seconds
              input_time <= input_time + 14'd60; // increment by 60
              input_time_valid <= 1'b1; // valid input
              initial_state <= 1'b0; // not initial
              state    <= ADD_60; // go to add 60 state
```

In this example, it is implementing one part of S0 transitions. If add1 is high, the system goes from INITIAL state (S0) to ADD_60 (s1) state



```
      end
      ADD_60:begin  // if add1, 60 seconds not inputted again, just count down the timer
          if (add1 == 1'b0)
              state <= COUNT_DOWN;
          else
              state <= ADD_60; // otherwise, loop back to ADD_60 state
```

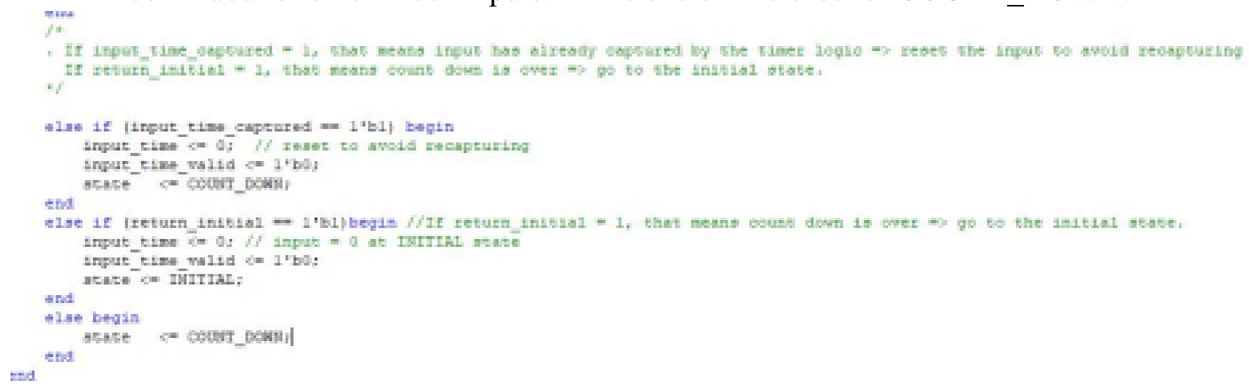
In this example, it is implementing the adding of a 60 second input by the user. If there is no other input, the system goes to the COUNT_DOWN state to start decrementing the time remaining. Otherwise, the state self loops back to its own state.



```
  end
  COUNT_DOWN:begin // count down the timer
      if (add1 == 1'b1 && add2 == 1'b0 && add3 == 1'b0 && add4 == 1'b0) begin // if add1, go back to that ADD_60 state
          input_time <= input_time + 14'd60; // increment input
          input_time_valid <= 1'b1;
          state    <= ADD_60;
      end
```

..

continued for other “add” inputs. At the end of this case for COUNT_DOWN:

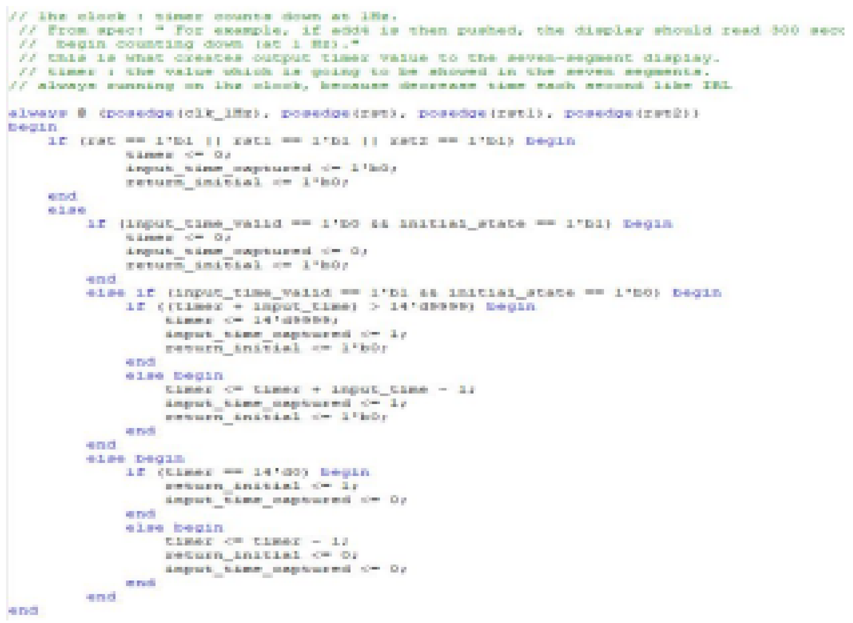


```
    /*
    . If input_time_captured = 1, that means input has already captured by the timer logic => reset the input to avoid recapturing
      If return_initial = 1, that means count down is over => go to the initial state.
    */

    else if (input_time_captured == 1'b1) begin
        input_time <= 0;  // reset to avoid recapturing
        input_time_valid <= 1'b0;
        state    <= COUNT_DOWN;
    end
    else if (return_initial == 1'b1)begin //If return_initial = 1, that means count down is over => go to the initial state.
        input_time <= 0; // input = 0 at INITIAL state
        input_time_valid <= 1'b0;
        state <= INITIAL;
    end
    else begin
        state    <= COUNT_DOWN;
    end
end
```

If **input_time_captured** = 1, that means the input has already been captured by the timer logic so the system resets the counter to avoid recapturing. If **return_initial** = 1, that means the countdown is over (time is zero) so the system should go to the INITIAL state.

The reset logic for **rst1** and **rst2** just follows the FSM diagram explained above.

Time Control Logic (Decrementing)

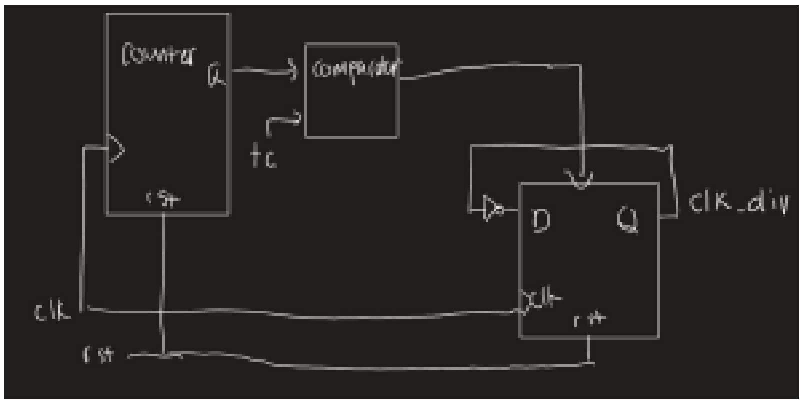


```
// the clock : timer counts down at 1Hz.
 // From spec: " For example, if add4 is then pushed, the display should read 300 secs
 //  begin counting down (at 1 Hz)."
 // this is what creates output timer value to the seven-segment display.
 // timer : the value which is going to be showed in the seven segments.
// always running on the clock, because decrease time each second like IRL

always @ (posedge(clk_1Hz), posedge(rst), posedge(rst1), posedge(rst2))
begin
    if (rst == 1'b1 || rst1 == 1'b1 || rst2 == 1'b1) begin
            timer <= 0;
            input_time_captured <= 1'b0;
            return_initial <= 1'b0;
    end
    else
        if (input_time_valid == 1'b0 && initial_state == 1'b1) begin
            timer <= 0;
            input_time_captured <= 0;
            return_initial <= 1'b0;
        end
        else if (input_time_valid == 1'b1 && initial_state == 1'b0) begin
            if ((timer + input_time) > 14'd9999) begin
                timer <= 14'd9999;
                input_time_captured <= 1;
                return_initial <= 1'b0;
            end
            else begin
                timer <= timer + input_time - 1;
                input_time_captured <= 1;
                return_initial <= 1'b0;
            end
        end
        else begin
            if (timer == 14'd0) begin
                return_initial <= 1;
                input_time_captured <= 0;
            end
            else begin
                timer <= timer - 1;
                return_initial <= 0;
                input_time_captured <= 0;
            end
        end
end
```

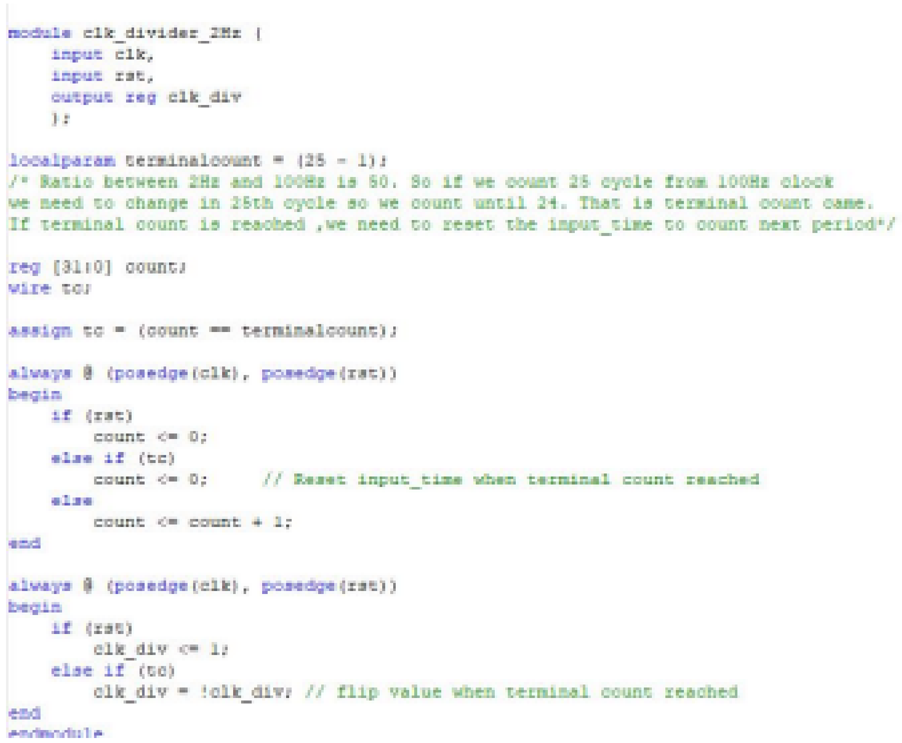
There are a series of if statements that deal with the time if one of the resets is high. There are if statements dealing with valid or invalid inputs when in or not in the initial state of the system. If the input is valid but the system is not in the initial state, that means a new input has been captured by the meter; thus, the current time has to be added as well. At the end, if these are none of the cases, the timer decrements by 1 each second since this always block is running on the clk_1Hz clock.

1Hz and 2Hz clock

A basic block diagram of how to implement a clock divider



A 1Hz and 2Hz clock is made to implement the flashing of the seven segments and also decrement the timer at the correct real time units.

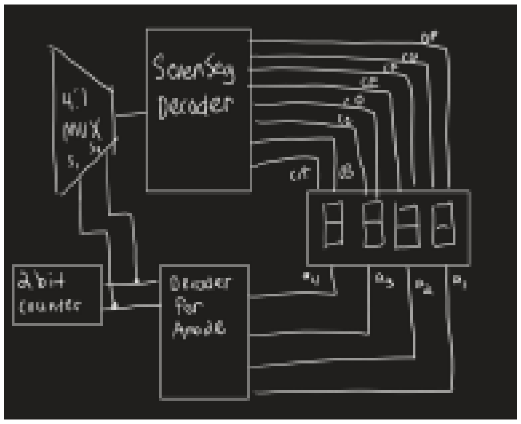


```
module clk_divider_2Hz (
    input clk,
    input rst,
    output reg clk_div
    );

localparam terminalcount = (25 - 1);
/* Ratio between 2Hz and 100Hz is 50. So if we count 25 cycle from 100Hz clock
we need to change in 25th cycle so we count until 24. That is terminal count came.
If terminal count is reached ,we need to reset the input_time to count next period*/

reg [31:0] count;
wire tc;

assign tc = (count == terminalcount);

always @ (posedge(clk), posedge(rst))
begin
    if (rst)
        count <= 0;
    else if (tc)
        count <= 0;     // Reset input_time when terminal count reached
    else
        count <= count + 1;
end

always @ (posedge(clk), posedge(rst))
begin
    if (rst)
        clk_div <= 1;
    else if (tc)
        clk_div = !clk_div; // flip value when terminal count reached
end
endmodule
```

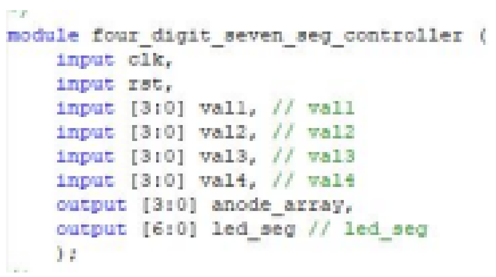
A terminal count variable is made and set to 24. This is because I want to change 100Hz to 2Hz. The ratio between these two frequencies is 50. Therefore, If it counts to the 25th cycle from 100Hz clock, the system needs to change at the 25th cycle, so it counts until 24. It takes 25 clock cycles for clk_div to change its value.
The same logic applies to a 1Hz clock, but it takes 50 clock cycles for clk_div to change its value; thus, the terminal count variable is set to 49.

Seven Segment Display Design

Here is a simple sketch of a block diagram showing how a common seven segment controller works:

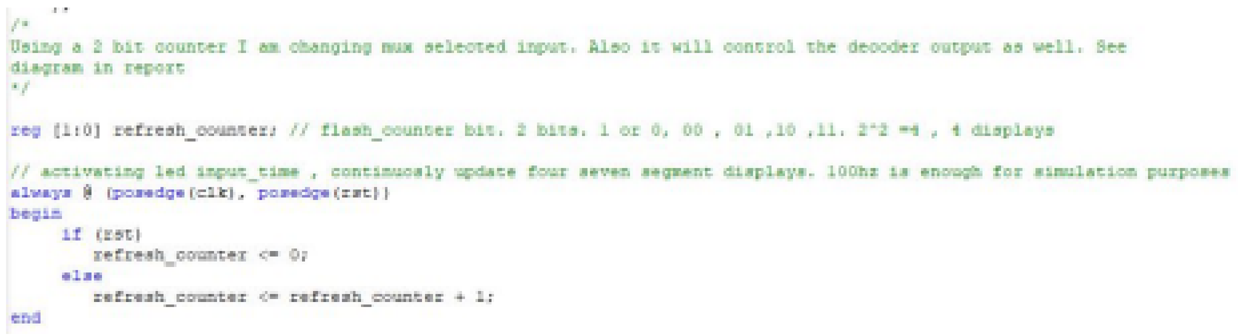


This is a traditional popular implementation of implementing a seven segment display design
In terms of this projects requirements:
The 4:1 **MUX** takes in **val1,val2,val3**, and **val4** as inputs.
The **decoder** for anode uses a variable called **anode_array** which is a 4 bit value [3:0].
The **2 bit counter refresh_counter** determines the select pin.
Based on the select pin, the mux will select one of the **vals** to be fed to the **seven segment decoder** and then to the displays. This is a red warning, but that is why the decoder for anode enable is used.
The **decoder** is used to turn off the other three segments (with **anodes**) and turn on the respective segment only respective to the anode to be activated.
This allows you to have all the right displays in each segment. Since this is happening with 100hz clock refresh rate, the human eye may not catch the effect of refreshing. The **refresh_counter** is used to go through all four **vals**; in a sense, it is serving as an activating led counter , continuously updating four seven segment displays.
Because of the 100hz master clock constraint, the **100hz refresh rate** is the best for simulation purposes.
This is the module definition layout:



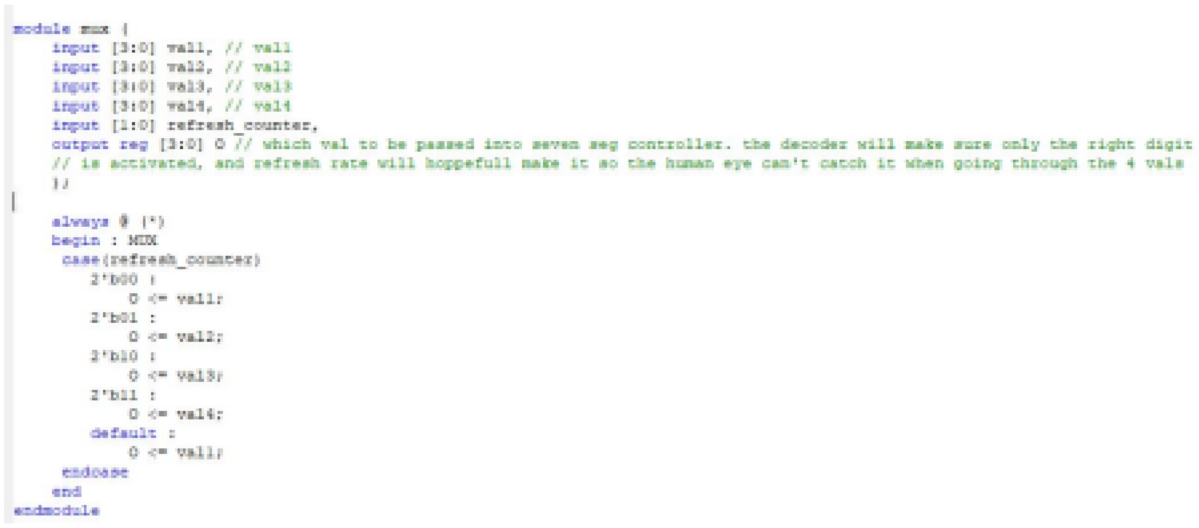
```
module four_digit_seven_seg_controller (
    input clk,
    input rst,
    input [3:0] val1, // val1
    input [3:0] val2, // val2
    input [3:0] val3, // val3
    input [3:0] val4, // val4
    output [3:0] anode_array,
    output [6:0] led_seg // led_seg
    );
```

//



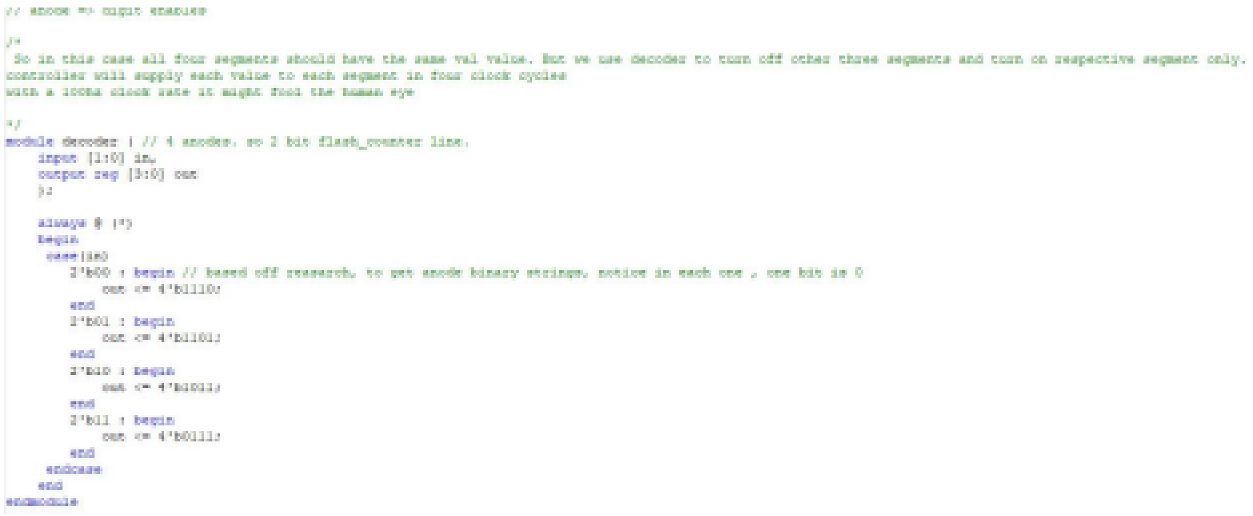
```
/*
Using a 2 bit counter I am changing mux selected input. Also it will control the decoder output as well. See
diagram in report
*/

reg [1:0] refresh_counter; // flash_counter bit. 2 bits. 1 or 0, 00 , 01 ,10 ,11. 2^2 =4 , 4 displays

// activating led input_time , continuosly update four seven segment displays. 100hz is enough for simulation purposes
always @ (posedge(clk), posedge(rst))
begin
    if (rst)
       refresh_counter <= 0;
    else
       refresh_counter <= refresh_counter + 1;
end
```

The **2-bit counter , refresh_counter,** is implemented using an always block running on the clock and incrementing the counter when reset is not high. The value of **refresh_counter** is used as the selector for the multiplexer and used in the decoder for anode enabling for the corresponding digit consistent with the **val** selected with the multiplexer. A **100Hz refresh rate** is used.



```
module mux (
    input [3:0] val1, // val1
    input [3:0] val2, // val2
    input [3:0] val3, // val3
    input [3:0] val4, // val4
    input [1:0] refresh_counter,
    output reg [3:0] O // which val to be passed into seven seg controller. the decoder will make sure only the right digit
    // is activated, and refresh rate will hoppefull make it so the human eye can't catch it when going through the 4 vals
    );

    always @ (*)
    begin : MUX
     case(refresh_counter)
        2'b00 :
            O <= val1;
        2'b01 :
            O <= val2;
        2'b10 :
            O <= val3;
        2'b11 :
            O <= val4;
        default :
            O <= val1;
     endcase
    end
endmodule
```

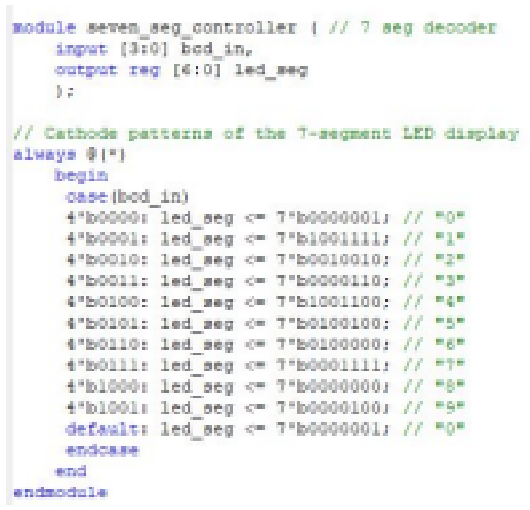
The code for multiplexer is just a case statement based on the value of **refresh_counter** translating what a multiplexer does in hardware into code. It's basically a series of if statements using the selector **refresh_counter** as the deciding factor of what **val** is chosen to be decoded with the right **cathodes**.



```
// anode => digit enabler

/*
 So in this case all four segments should have the same val value. But we use decoder to turn off other three segments and turn on respective segment only.
controller will supply each value to each segment in four clock cycles
with a 100hz clock rate it might fool the human eye

*/
module decoder ( // 4 anodes, so 2 bit flash_counter line.
    input [1:0] in,
    output reg [3:0] out
    );

    always @ (*)
    begin
     case(in)
       2'b00 : begin // based off research, to get anode binary strings, notice in each one , one bit is 0
           out <= 4'b1110;
       end
       2'b01 : begin
           out <= 4'b1101;
       end
       2'b10 : begin
           out <= 4'b1011;
       end
       2'b11 : begin
           out <= 4'b0111;
       end
     endcase
    end
endmodule
```

The decoder takes in the **refresh_counter** variable from the 2 bit counter. The output is **[3:0] anode_array**.

In the **blinking/flashing** logic section of the code, **a1-a4** are set to their corresponding index element of the **anode_array** when the displays should not be blanked out. The values of what to set **out/anode_array** to are from research about the actual hardware. Of course, ideally, with the **100Hz refresh rate**, it would look normal to the user when the controller will supply each value to each segment in **four clock cycles**



```
module seven_seg_controller ( // 7 seg decoder
    input [3:0] bcd_in,
    output reg [6:0] led_seg
    );

// Cathode patterns of the 7-segment LED display
always @(*)
    begin
     case(bcd_in)
     4'b0000: led_seg <= 7'b0000001; // "0"
     4'b0001: led_seg <= 7'b1001111; // "1"
     4'b0010: led_seg <= 7'b0010010; // "2"
     4'b0011: led_seg <= 7'b0000110; // "3"
     4'b0100: led_seg <= 7'b1001100; // "4"
     4'b0101: led_seg <= 7'b0100100; // "5"
     4'b0110: led_seg <= 7'b0100000; // "6"
     4'b0111: led_seg <= 7'b0001111; // "7"
     4'b1000: led_seg <= 7'b0000000; // "8"
     4'b1001: led_seg <= 7'b0000100; // "9"
     default: led_seg <= 7'b0000001; // "0"
     endcase
    end
endmodule
```

The seven segment decoder/controller takes in the **val** selected from the mux submodule And translates it to the appropriate **cathode patterns** that would be needed on actual hardware.

All of these **submodules** are called within a higher level **four_digit_seven_seg_controller** module**,** which is called inside the top module **parking_meter**.
The last part of my code converts a 14 bit binary (able to hold 9999) to **bcd_array[15:0]** using the famous **double dabble algorithm.**
In another section of my code, I set the **val** values to the corresponding range of 4 indices in the larger [15:0] array.

Blinking/Flashing Design

This section implements the requirement from the project manuscript:

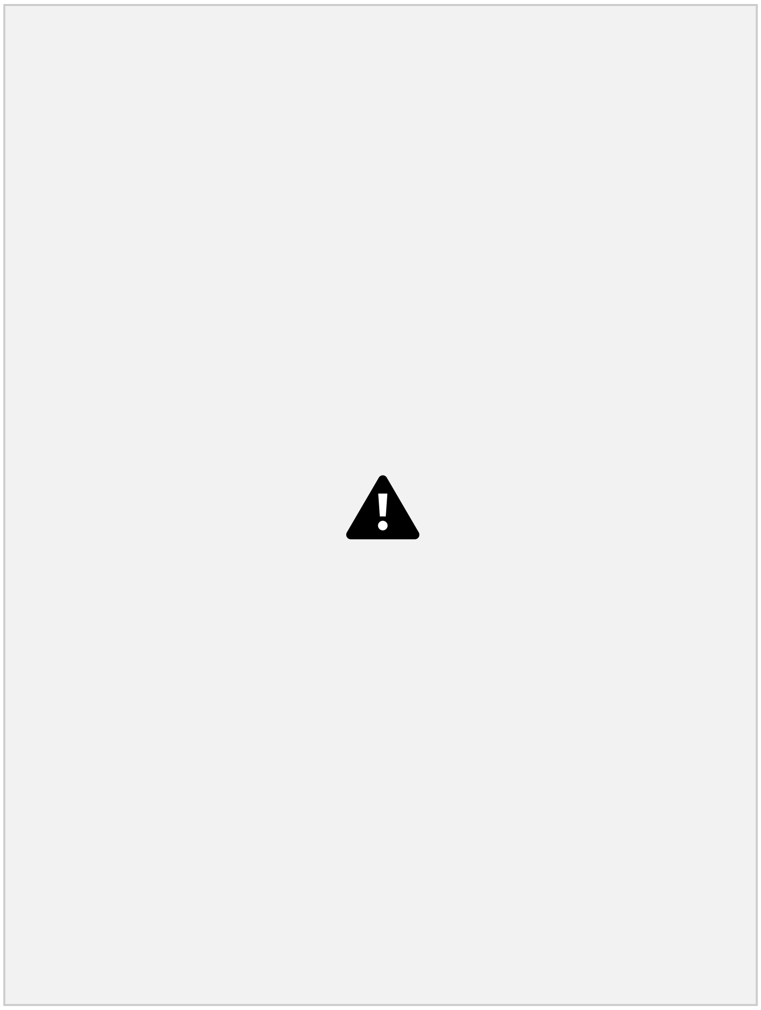
By flashing with period 1 sec, this means the seven segment displays should be on for 0.5 seconds and then blanked out for 0.5 second. In other words, in the first half of the 1Hz clock cycle the seven segment displays should be on, and on the other half (the low edge) the displays should be blanked out.

By flashing with period 2 second, this means the seven segment displays should be on for 1 second and then blanked out for 1 second. In terms of the 1Hz clock, the seven segments displays should be on for one clock cycle and off for one clock cycle. In terms of the 2Hz clock, the seven segment displays should be on for two clock cycles and off for two clock cycles.

In order to determine what flashing period to use, I created two internal variables and set them based on a condition:

A two sec flashing period is used if the **timer** is greater than 0, but less than 180. A 1 second flashing period is used if the timer is 0 ; in other words, the seven segment displays are 0000.

An always block running on the 2Hz clock is made that increments a two bit **flash_counter** variable which is used in combinational logic with the internal period variables described earlier.

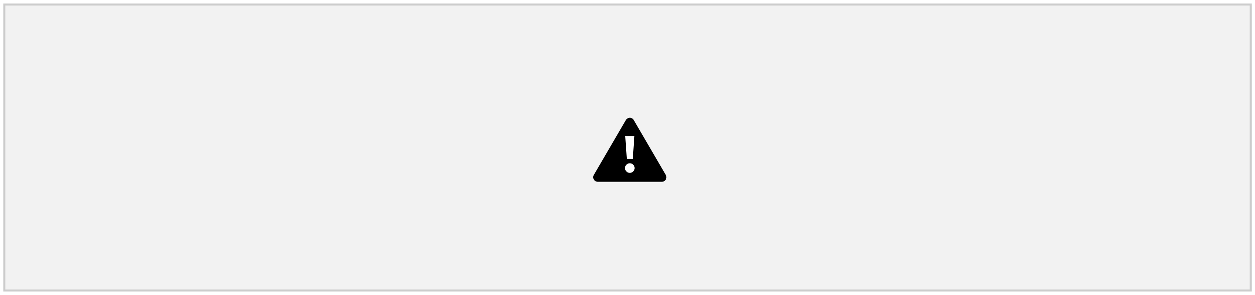


Combinational logic to set the **a1-a4** values depending on the flashing period used. The select variable determines whether to set it equal to **anode_array** and allow refreshing of the display segments or blank out the displays.

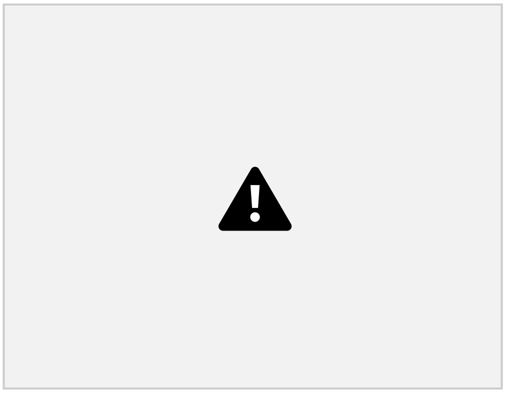
Waveforms are helpful to explain this logic, so here is a screenshot of the waves and the select variable and internal period variables

This is the case of when time is 58. As you can see every clock cycle of the 2Hz clock, the select bit increments as the always block described earlier implements in Verilog

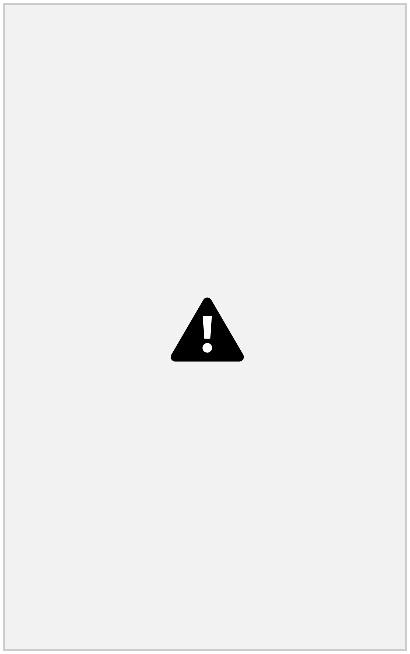
As you can **flash_counter**[1] is 0 or the MSB of flash_counter is 0, a1-a4 will be set to their corresponding values from the **anode_array**. When the MSB is 1, the values are blanked out. As you can see the flash_counter[1] bit changes every 2 clock cycles from the 2Hz clock or every clock cycle from the 1Hz clock.



This is the case when time is 0, so **one_sec_flash** should be 1. As you can see in the waveforms, the logic matches the code.**flash_counter**[0] changes every 0.5 second or half of the 1Hz clock cycle or every 2Hz clock cycle. Therefore, flash_counter[0] bit is changing twice in the 1Hz period. So, I can use the flash_counter[0] = 0 condition to blink and then we can use flash_counter [0] = 1 for the off/blank condition. When flash_counter[0] is 1, the display segments are blanked out which I'll explore in the simulation documentation section of the project report.

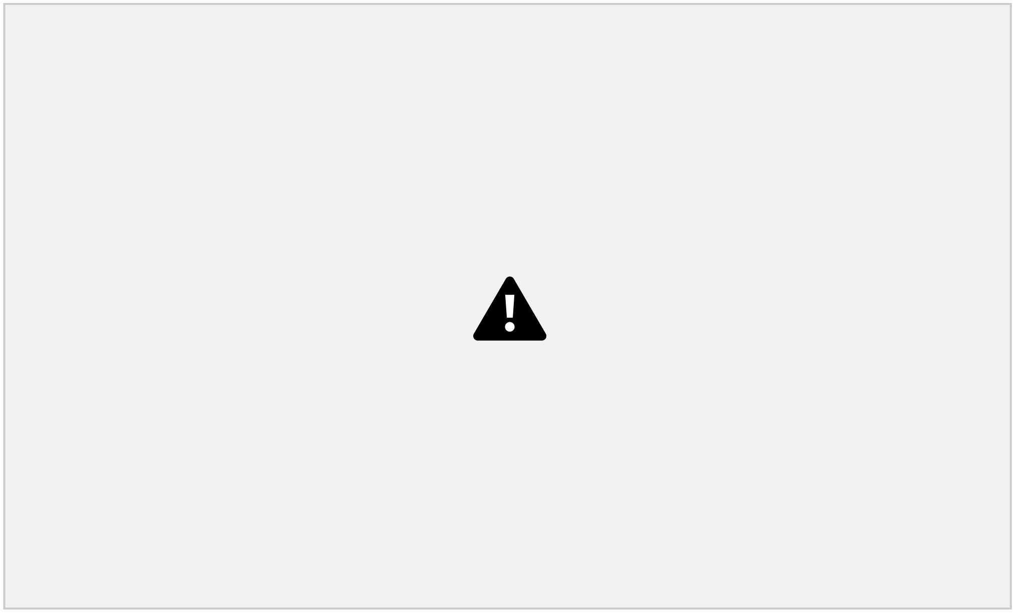


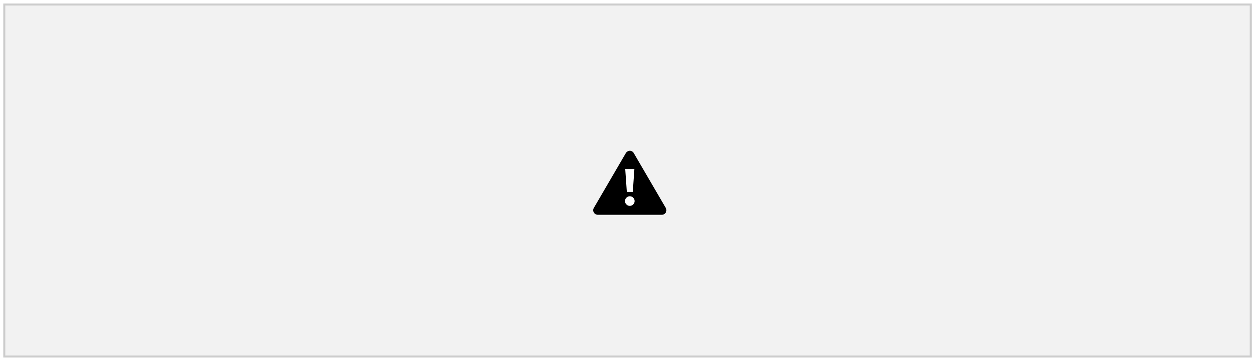
Based on this circuit, anodes are enabled when they are "0" and are off when "1" since PNP transistors are used.



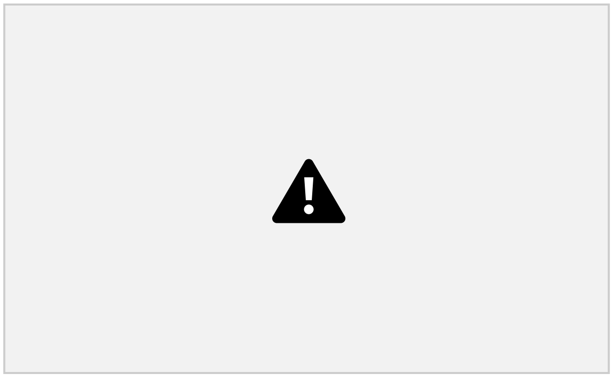
This is a high level schematic indicating the inputs and outputs of the system generated by Xilinx ISE

The lower level schematics are very large, but here are screenshots of key sections:

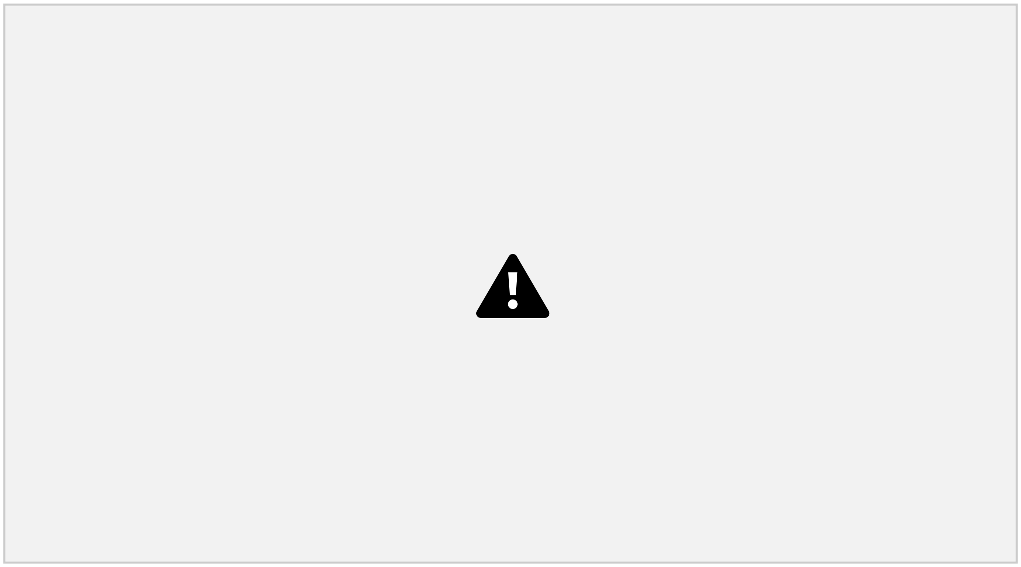




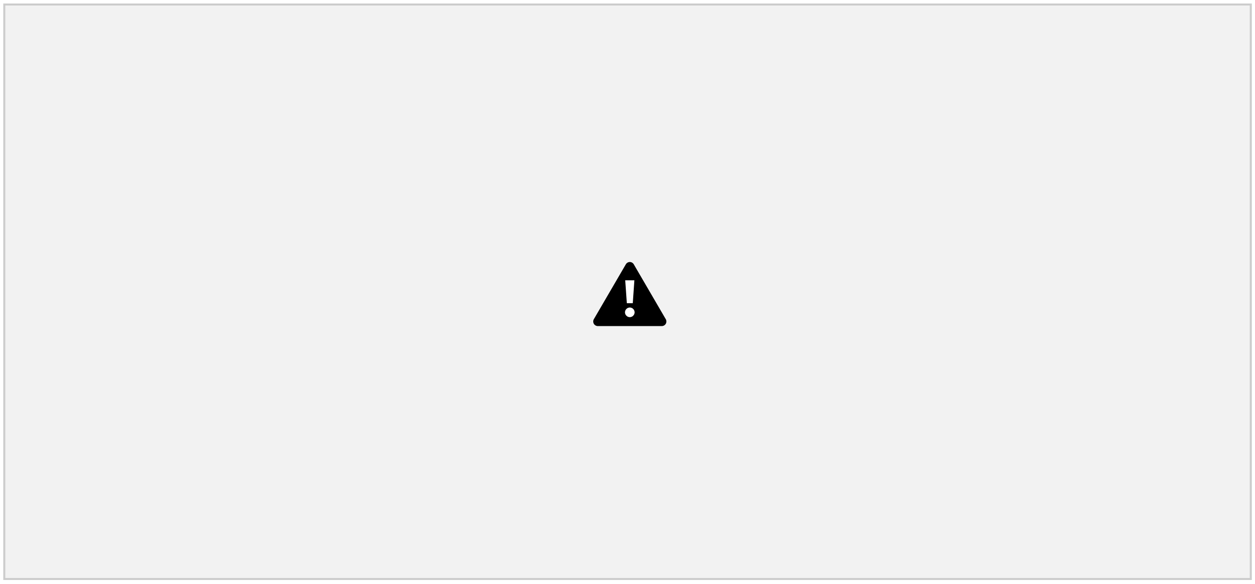
There is the flash counter used to determine which flashing period to use to activate the anodes which are determined in the anode logic earlier. That is why a mux is being shown.



A mux is being used to determine the states



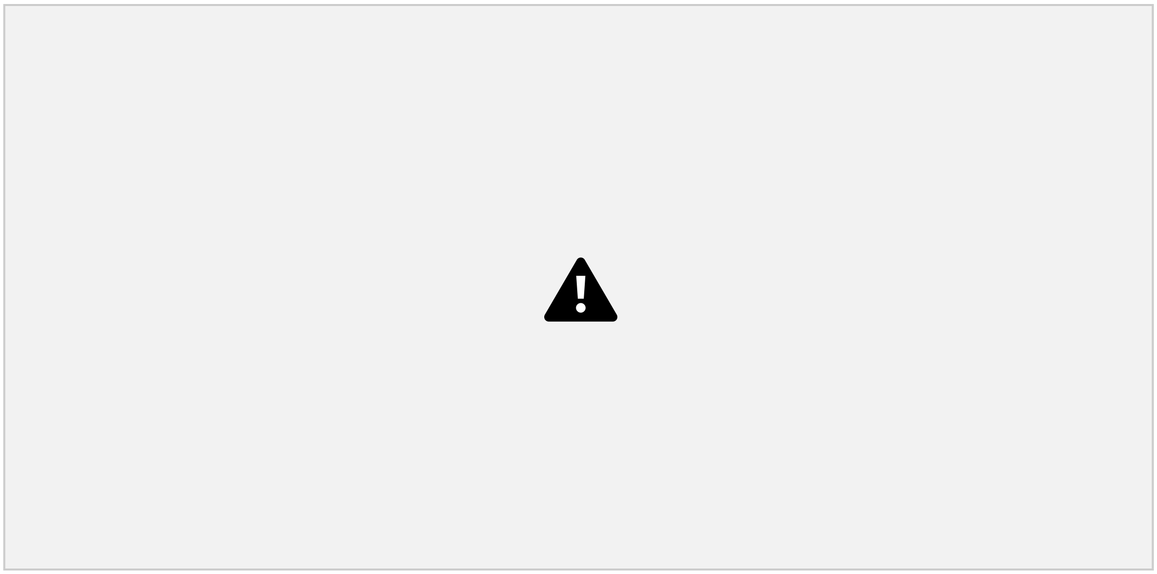
Similar to my simple block diagram, a counter and D flip flop is being used to implement the clock divider.



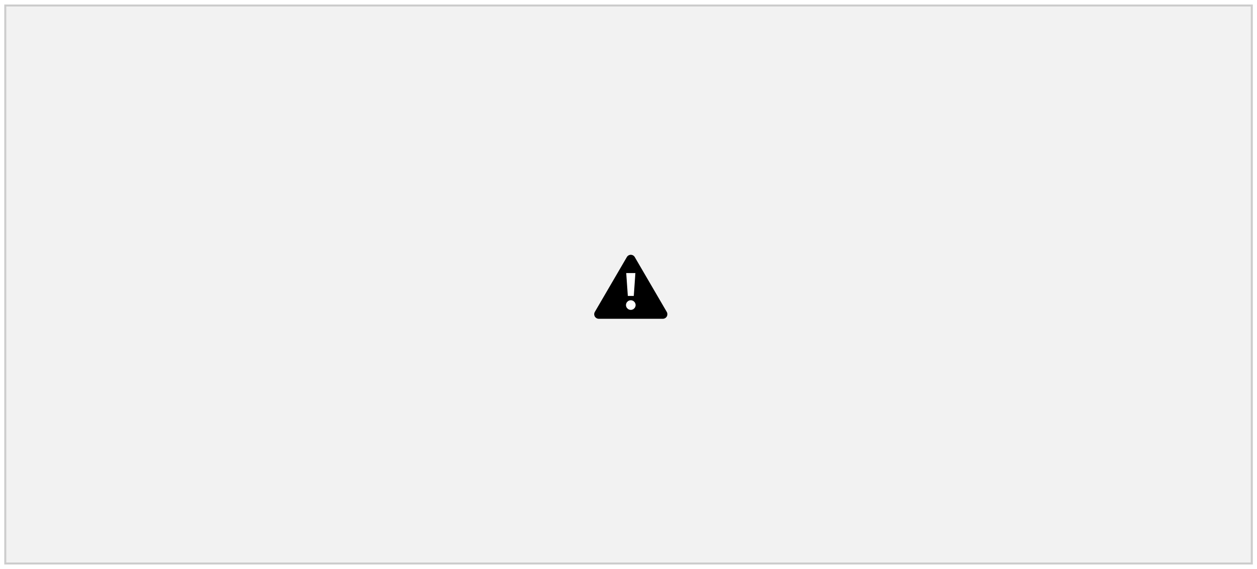
Similar to my drawing, there is a mux, and two decoders being used to implement the Seven segment displays: vals going to the mux, decode is used for anode enabling, and seven seg controller is used for cathode patterns for led_seg.

3. <u>Simulation Documentation</u>

To make things easier, I will show a zoom out version and zoomed in version before explaining my test cases



This is a waveform zoomed out.



This is what a waveform looks like zoomed in.

Hopefully, when I put it in one representation (zoomed in or zoomed out), you will understand what is going on in these marked sections.

Test Case 1

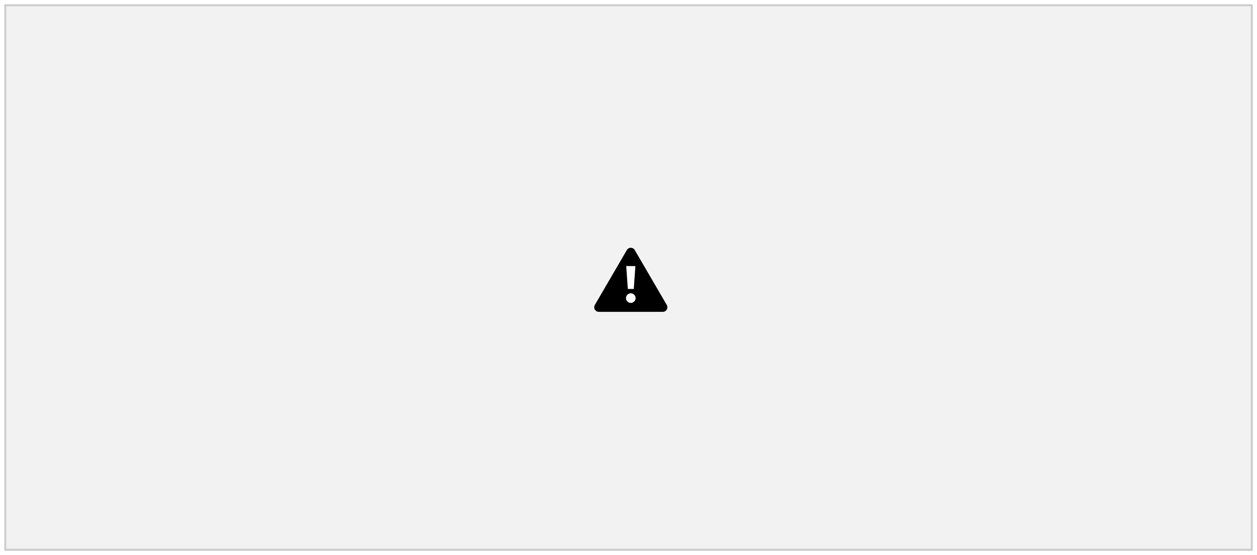


Figure 1: add1 input , 60 seconds

In this test case, I wanted to check if decrementing the time, anode enabling and refreshing, blanking out during odd times, and appropriate val outputs were working correctly. As you can see, the time decrements by 1 every second (1000ms = 1s).

In addition, the val1 - val4 match correctly with what would be displayed on the seven segments. Here are the values in binary form:

Figure 1.1: val1-val4 verification
For example for the time of 59 seconds remaining, val1 is 9, val 2 is 5, and val 3 and val4 are 0. In other words, the display would be 0059.

If zoomed in, you can see that led_seg is definitely changing:

Figure 1.2: led_seg and anode enabling verification

Figure 1.3: refresh rate verification. As you can see the refresh rate is 100Hz. 0 means activated for the anodes. 1 means off.

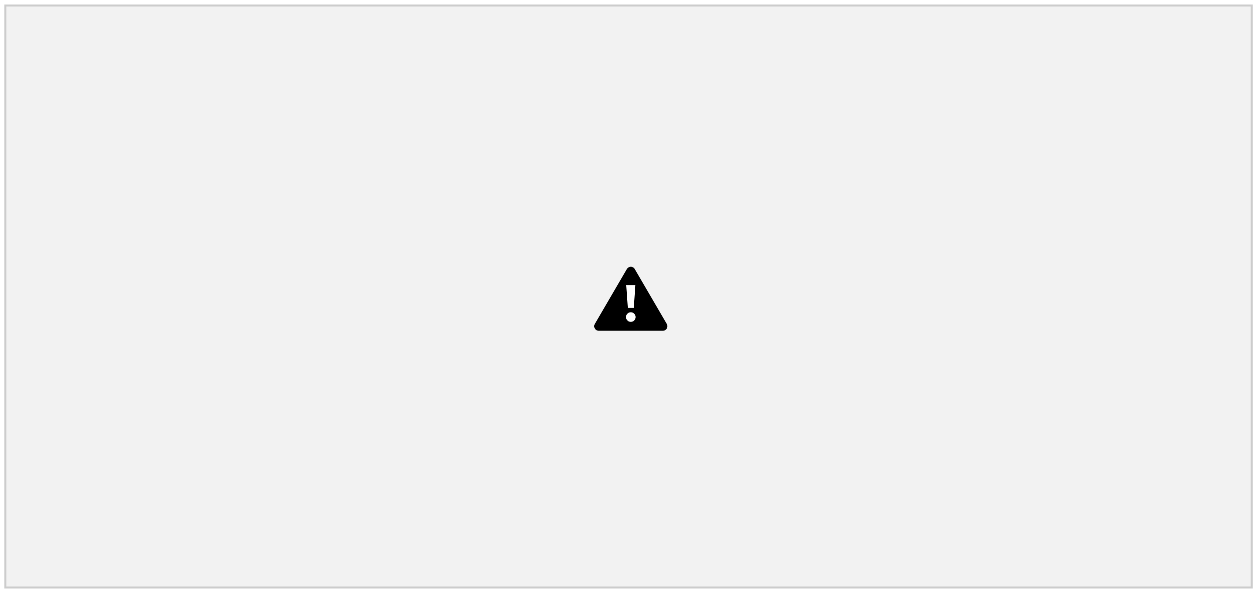


Figure 1.4: Two second period flashing verification

As you can see, the anodes are all off, meaning the values are blanked out when the time is odd (i.e 49 seconds). The duration matches the clk_1hz wave in the screenshot. This means it is off for 1 second and on for 1 second where the segments are being refreshed at 100Hz.

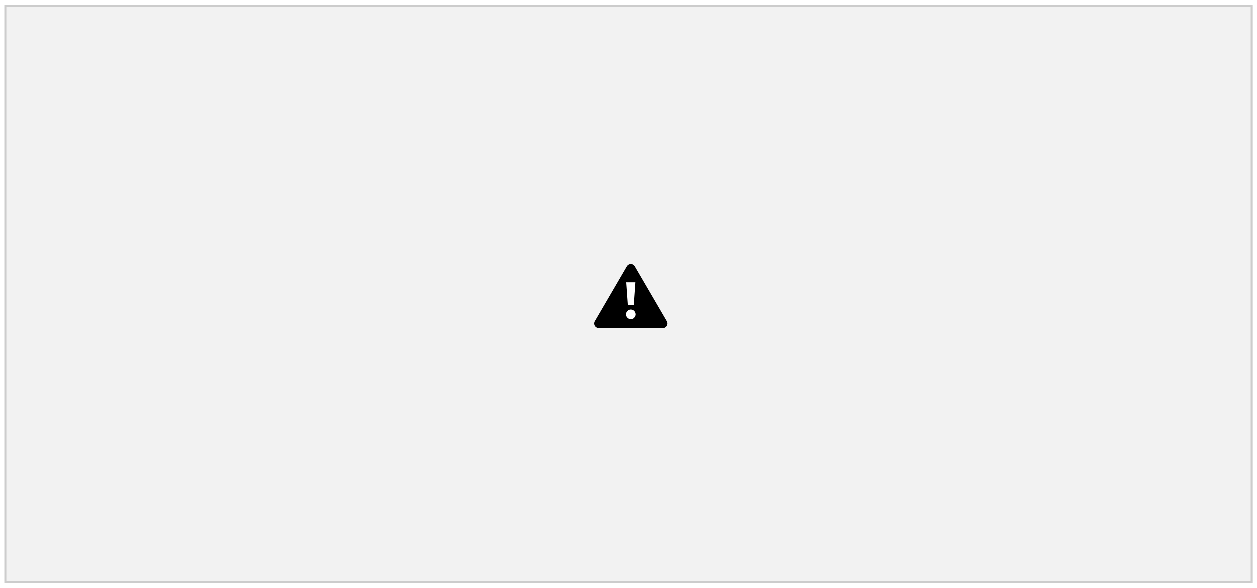


Figure 1.5: 1 second period flashing verification

When time has expired or in INITIAL state, the displays should flash with period 1 sec and 50% duty cycle. As you can in the screenshot, the segments are on for 0.5 seconds and off for 0.5 seconds.

<u>Test Case 2</u>

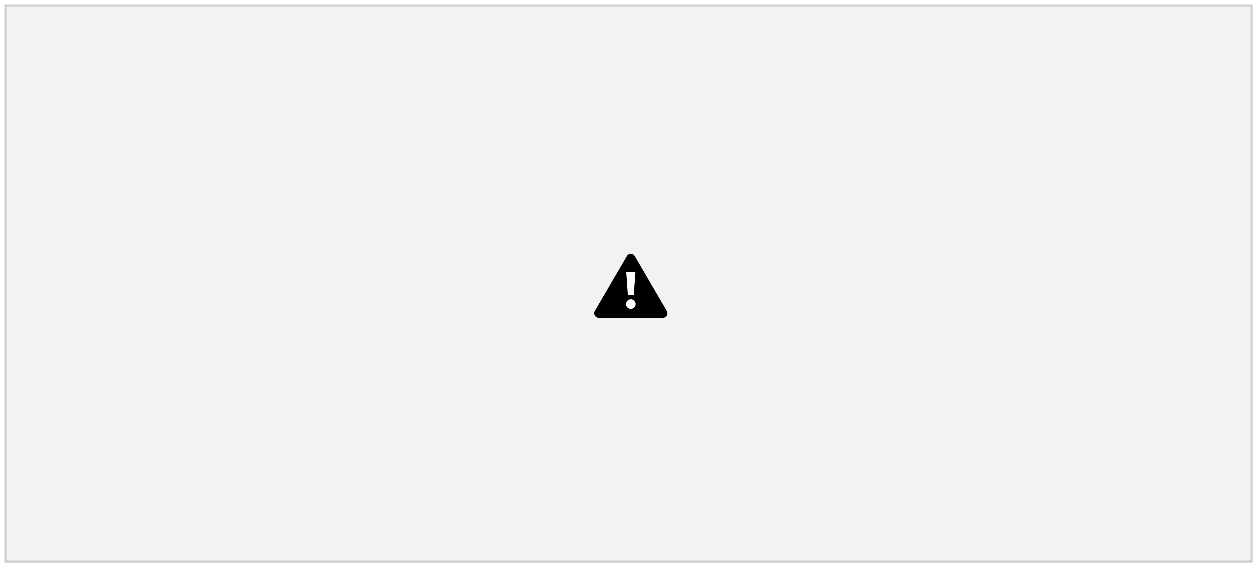


Figure 2: add1 (60 seconds) and add2(120 seconds) inputted within 1 second

In this case, since the two coinos came within one second, they will add up as a total time of 180 since 60 + 120 = 180. Because the time is less than 180 seconds, the segments are blinking with a period of 2 sec. The segments are on for 1 second and then off 1 second.

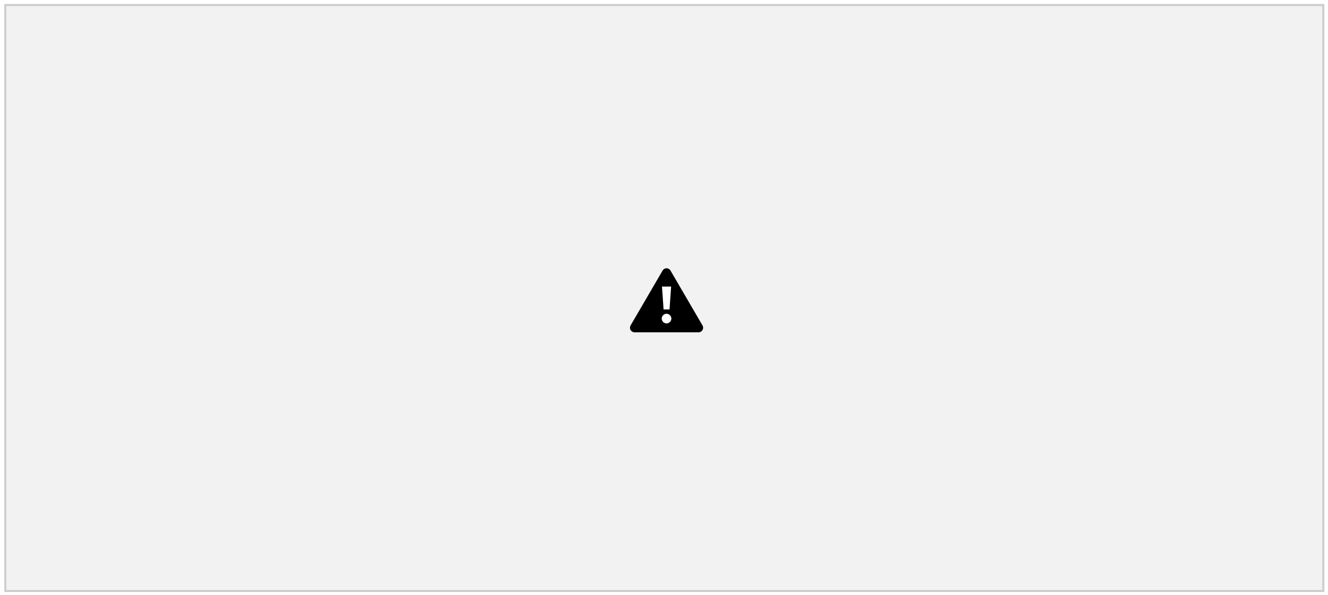


Figure 3: add 3 (180 seconds) and add1(60 seconds) after 10 seconds.

As you can see, when add3 is inputted, it starts counting down. Since the time is less than 180s the seven segments are blinking with a period of 2 second (on for 1 second and off for 1 second).

When the add1 (60 seconds) is inputted, the time adds to 229 (170 + 60 -1). Since the time is now greater than 180 seconds, the seven segments are not blinking anymore. They are refreshing continuously.

Test Case 4

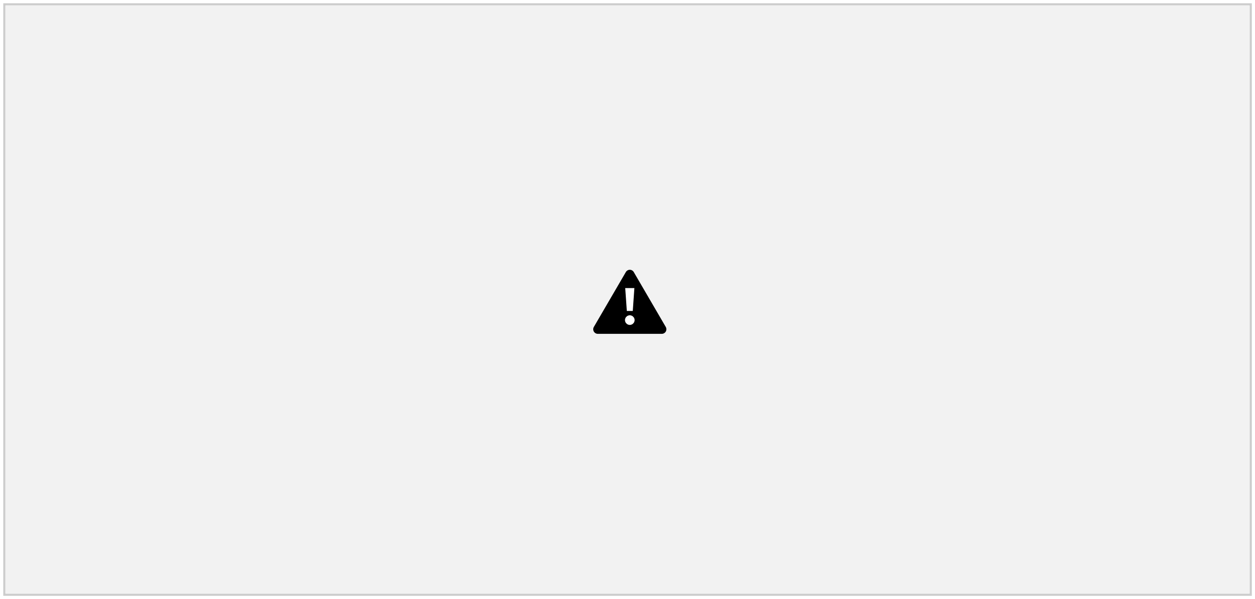


Figure 4: add4 and then rst1

A 300 second input is made then after some time rst1 input is made. As you can see, the parking meter starts counting down from 300s, but after rst1 is HIGH, the timer resets to 16 seconds and counts down until zero. Since the time is greater than 180s, there is continuous refreshing until the time rst1 is set to high, where the timer starts counting down from 16.

Test Case 5

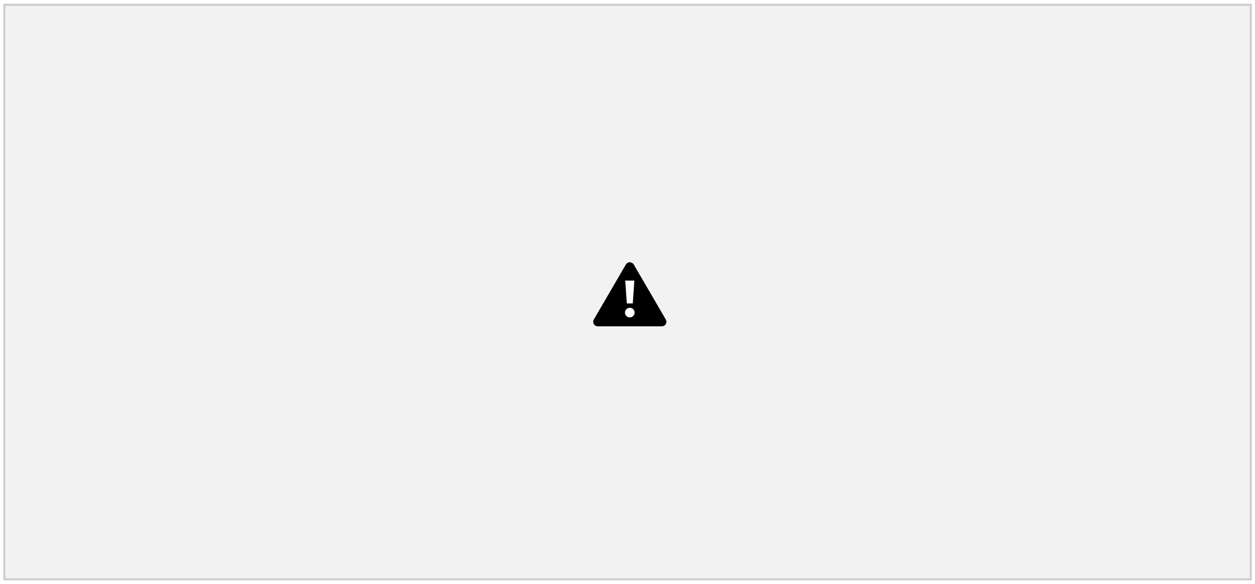


Figure 5: add3 and then rst2

Same idea as the previous test case, but add3 is set to HIGH and then rst2 is set to HIGH. After rst2 is set to high, the parking meter starts counting down from 150. Once again, the blinking rates are correct with period 2 since the time is less than 180.

Test Case 6

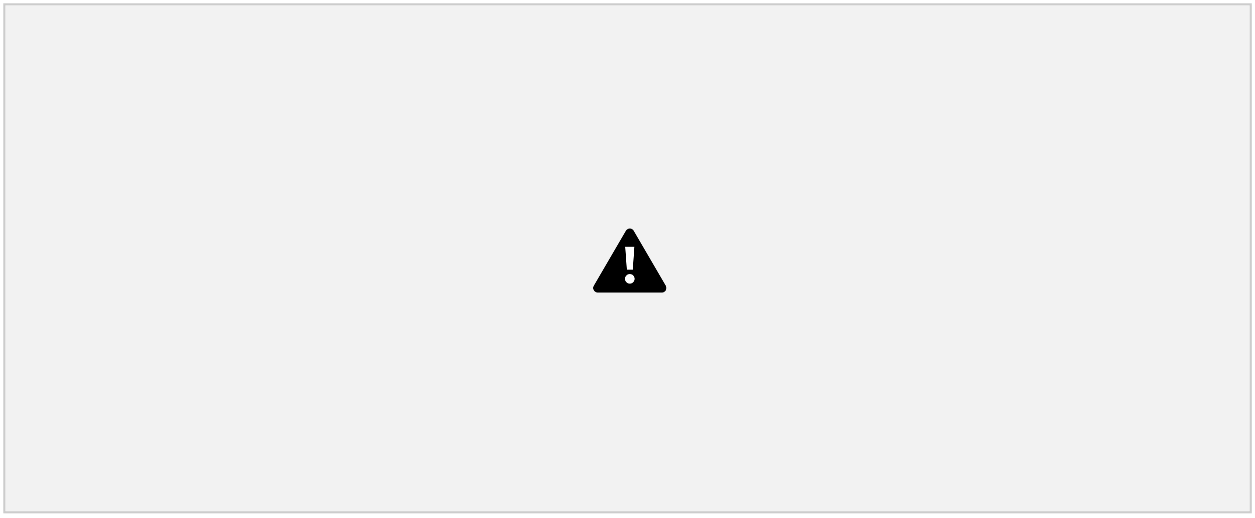


Figure 6: beyond 9999

In this test case, I input enough add4's until it reaches 9999. As you can see in the screenshot, when I try to input another add4 after reaching 9999, the system loops back to 9999 where it starts counting down again.

Design Summary and Implementation Map

From what I can see in the maps and summaries, lots of Flip flops and latches are used to implement the parking meter. I assume it is because of the transition states and counters. I assume the buffers are used to deal with the intermediate points due to the user inputs. The flip flops are probably used to implement the clock dividers and used alongside comparators.

4. Conclusion

My design has 8 states for the Finite State Machine part of my system. There is a multiplexer and two decoders used to implement the seven segment displays with refreshing, anode enabling, and cathode pattern deciphering. I have a section of code that deals with decrementing the time remaining on the parking meter which runs on a 1Hz clock. I used a counter and conditional variables to implement what flashing period to use for the 0000 display or when there is less than 180 seconds remaining. The 2 bit counter increments on a 2Hz clock. It takes advantage of how many cycles it takes to change the MSB or LSB of the 2 bit binary string. Difficulties I encountered during the lab was understanding the simulation. I saw that, for example, the user inputs a 60 second coin. In my simulation, it showed 59 right after. After researching and just analyzing possibilities, I understood that the user has the capability to give inputs at any time. The time they input is not synchronized with the 1Hz clock which is the rate the time is decrementing at. As in real application, we are not in control of the clocks. These clocks are inputs. The user inputs are asynchronous and the timer output is synchronous with the

1Hz clock.