Adam Cole
UID: [redacted]

Project 5 Report

1. Introduction and Requirements

1.1 Introduction

For project 5, our goal was to successfully implement a design for a parking meter in Xilinx ISE. In this design, the user has the option to add time with the four “add” buttons, and there are internal signals such as a system clock and reset buttons that we also must integrate into our design. The design requirements are given in the following section.

1.2 Design Requirements

The inputs and outputs of our parking meter module follow the following specification, given by the project spec.

INPUT	BITS	BEHAVIOR
add1	1	Adds 60 seconds to the meter.
add2	1	Adds 120 seconds to the meter.
add3	1	Adds 180 seconds to the meter.
add4	1	Adds 300 seconds to the meter.
rst	1	Resets meter to initial state.
rst1	1	Resets meter to 15 seconds.
rst2	1	Resets meter to 150 seconds.
clk	1	Frequency = 100 Hz system clock.

OUTPUT	BITS	BEHAVIOR
val1 <3:0>	4	Value of the 1000s place digit on the meter
val2 <3:0>	4	Value of the 100s place digit on the meter

val3 <3:0>	4	Value of the 10s place digit on the meter
val4 <3:0>	4	Value of the 1s place digit on the meter
a1	1	Anode bit for 1000s place digit
a2	1	Anode bit for 100s place digit
a3	1	Anode bit for 10s place digit
a4	1	Anode bit for 1s place digit
led_seg <6:0>	7	7-segment display cathode bits for the current digit value

In addition to our parking meter interface, the project specification gave us a description of the parking meter's requirements to be considered functional.

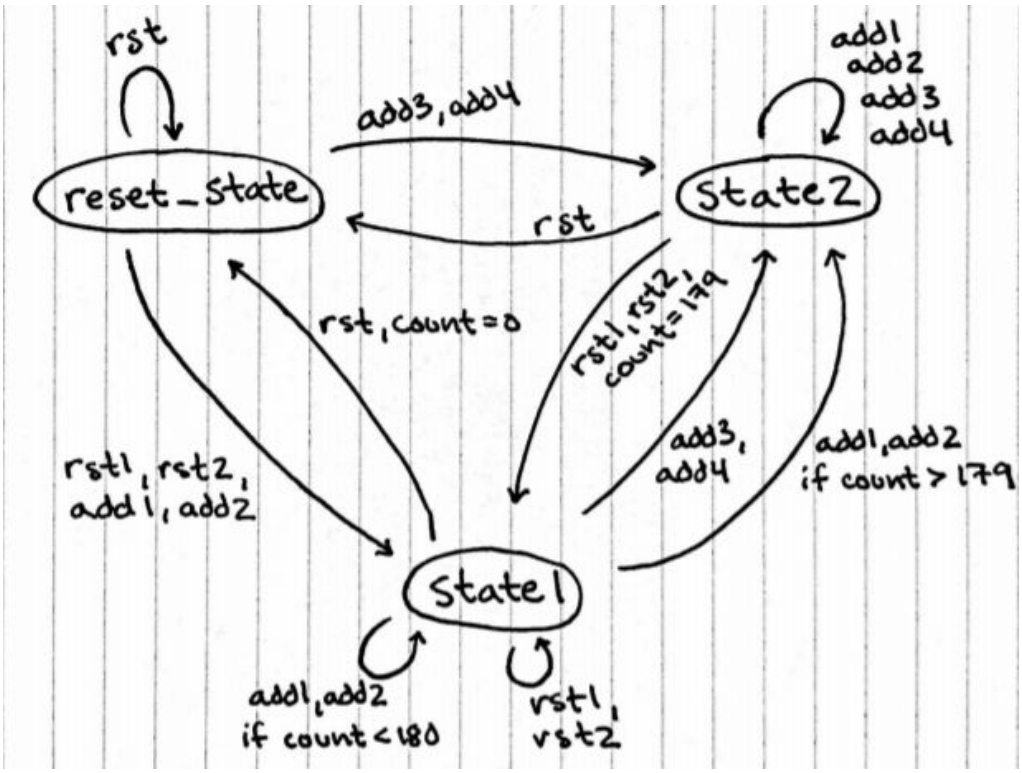
The parking meter will always display the remaining time left to park in seconds. The maximum amount of time allowed on the meter is 9999 seconds, or about 2 hours and 45 minutes. The display will be shown in 4 digits, each digit represented by a 7-segment display. The parking meter must begin in the initial state, the state when the parking meter is waiting for a user to insert money and start the parking timer. In this state, the 7-segment display should be flashing 0000 with a *T = 1sec* 50% duty cycle. This means that the meter should display 0000 seconds for *0.5sec* and nothing for the other *0.5sec*. A global reset will take the parking meter back to this initial state, and this occurs when the `rst` input is set to 1.

Given the interface for our parking meter, users have added 60 seconds of parking when the `add1` input is set to 1, 120 seconds of parking when the `add2` input is set to 1, 180 seconds of parking when the `add3` input is set to 1, and 300 seconds of parking when the `add4` input is set to 1. When a user adds parking time, our meter should increase the time value and begin counting down immediately, flashing in the correct instances. When our meter has less than 180 seconds of time remaining, the meter should show the remaining time for *1sec*, and show nothing for another *1sec*. The meter should show even time values, and flash blank for odd time values. In the spec, this is described as a *T = 2sec* 50% duty cycle. In the final case when the meter has more than 180 seconds remaining, the meter should constantly display the time, counting down every second. This is described in the project spec as counting down with frequency *1 Hz*.

2. Design Description

2.1 Finite State Machine Diagram

In order to implement the parking meter design, I created a verilog module which implemented a finite state machine in order to loop through the states of the design. In order to compute the correct representations of the current time, I created three submodules: clk_div_1s.v, dec_to_BCD.v, and LED_controller.v. Below is my Finite State Machine Diagram for the parking meter.



My states were encoded in the following order:

STATE	ENCODING
reset_state	00
state1	01
state2	10

I used three always blocks to implement the finite state state machine displayed above. The first always block updates the current state of the machine to the next state at each system clock cycle. The second always block updates the counter at each button press, or lack thereof resulting in the timer decrementing the time remaining. The third always block investigates the timing threshold between the three states, and updates our `NEXT_STATES` variable to the correct next state based on the time remaining. In addition to the finite state machine, I created three submodules to display the current remaining time on the 7-segment display. I describe these submodules in sections **2.3** and **2.4** below.

2.2 Reset State, State1, and State2

I decided to create three states in order to easily identify the result of each button press given the current state. The first state, reset_state, is when the parking meter has no time remaining left of the machine. In this state, the meter should be flashing on and off with period *T = 1sec* 50% duty cycle. The second state, state1, is when the parking meter has less than 180 seconds left on the clock. In this state, the meter should be flashing on and off with period *T = 2sec* 50% duty cycle. By this cycle, every other number should be displayed. For our meter, only the even numbers will be displayed. The final state, state2, is when the meter has more than 179 seconds remaining. In this case, the clock should not be flashing at all - and display the timer counting down. I implemented the state transitions with if statements depending on the current state in an always block.

In any state, selecting `rst1` or `rst2` will automatically take the parking meter to state1 since this resets the remaining time to less than 180 seconds, but not 0. Similarly, in any state selecting `rst` will automatically take the parking meter to the reset_state since this resets the timer to 0 seconds. In any state, selecting `add3` or `add4` will automatically transition the meter to state2. In state2, adding any amount of time will keep the parking meter in state2. The other state transitions were based on the current counter value. If through selecting either `add1` or `add2`, the counter ever exceeded 0 or 179, a state transition was made. Oppositely, if the counter ever decremented past 180 or past 1, a state transition was made the opposite direction. This logic was all implemented through always blocks in parking_meter.v.

2.3 Clk_div_1s and dec_to_BCD Submodules

Once our finite state machine was implemented, I created three other submodules to continuously display the correct time and output the correct values. Two of these

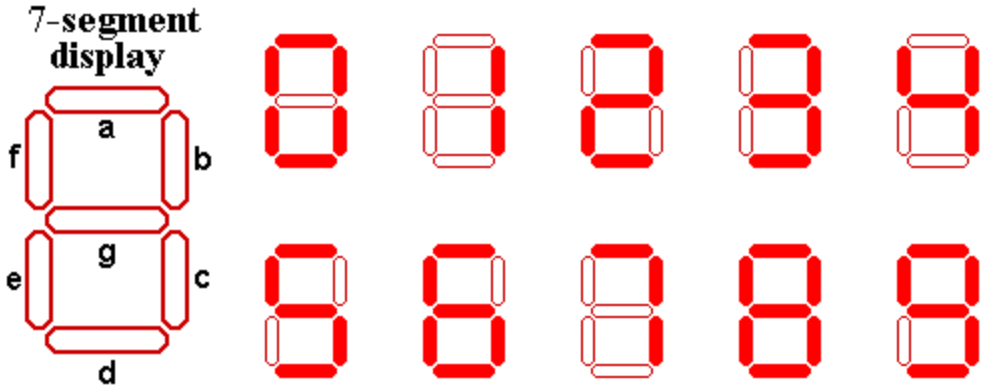
modules were clk_div_1s.v and dec_to_BCD.v. My clock divider submodule was easy to implement - the module reads in the system clock and increments a counter at each positive edge. Since I wanted to create a clock that cycled every second, I used a 3-bit counter that overflowed at 10 and toggled at 5. In this way, I transformed the *100 Hz* system clock into a *T=1sec* 50% duty cycle clock. I used this clock in my LED_controller.v submodule that I describe in the next section **2.4**.

In addition to my clock divider submodule, I created another submodule that translated the current remaining time of the counter into four values - one for each 10s place in the counter. Since we are developing with hardware, modulo is not a valid operation. In order to extract the four digits, I started by dividing the counter by 1000 and storing the result. This was our first digit, `val1`. In order to extract the 100s place digit, I divided the current counter by 100 and subtracted `val1` multiplied by 10. This was our second digit, `val2`. I continued on in the same fashion to extract the remaining values.

2.4 LED_controller Submodule

The LED_controller submodule was responsible for outputting the correct values of the 7-segment LED display anodes and cathodes. In the Nexys 3 7-segment display, there are 7 segments arranged in the shape shown below. Each segment is assigned a corresponding letter. For each segment, there is a cathode wire that is attached. In order to light the segment up, this cathode value must be set to 0. This is because the transistors in the segments only light up in the presence of voltage (potential difference across the transistor). Since one side of the transistor is constantly 1, turning the cathode wire to 0 creates the voltage that gives the segment its light.

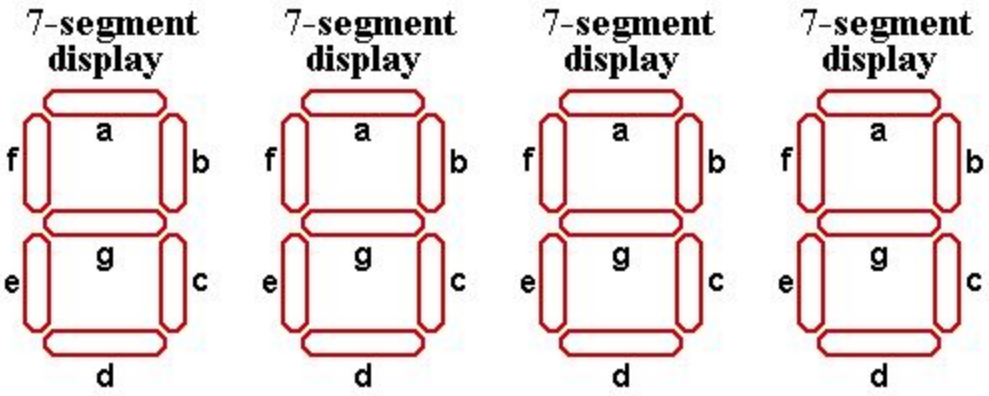
Below is the Nexys 3 7-segment display documentation, also showing the correct segment displays for each integer.



The parking meter design outputs the LED cathode bits in the `led_seg <6:0>` output, with the most significant bit corresponding to a's cathode wire and the least significant bit corresponding to g's cathode wire. The `led_seg <6:0>` values for each LED value are given in the table below, and the equivalent unsigned decimal representation of the binary value.

LED VALUE	Ca	Cb	Cc	Cd	Ce	Cf	Cg	Decimal Value
LED0	0	0	0	0	0	0	1	1
LED1	1	0	0	1	1	1	1	79
LED2	0	0	1	0	0	1	0	18
LED3	0	0	0	0	1	1	0	6
LED4	1	0	0	1	1	0	0	76
LED5	0	1	0	0	1	0	0	36
LED6	0	1	0	0	0	0	0	32
LED7	0	0	0	1	1	1	1	15
LED8	0	0	0	0	0	0	0	0
LED9	0	0	0	0	1	0	0	4

In addition to the cathodes, the parking meter requires four anode bits in order to signify which of the four 7-segment displays the current cathode values correspond to. Our parking meter uses four Nexys 3 7-segment displays, arranged in the following fashion.



Here, the leftmost 7-segment display displays `val1`, the next one displays `val2`, and so on until the rightmost displays `val4`. Anode bit `a1` corresponds to the `val1` display, `a2` corresponds to the `val2` display, `a3` corresponds to the `val3` display, and `a4` corresponds to the `val4` display. Similarly to cathodes, anodes must also be set to 0 in order for the corresponding 7-segment display to light up. Refer to the chart below to see the anode bit configurations to display the given values.

7-Seg Display	a1	a2	a3	a4
val1	**0**	**1**	**1**	**1**
val2	**1**	**0**	**1**	**1**
val3	**1**	**1**	**0**	**1**
val4	**1**	**1**	**1**	**0**
None	**1**	**1**	**1**	**1**

In order to display the counter value 6278, the `led_seg <6:0>` output must be set to the cathode bits for value **`LED6`** when anode bits correspond to those shown in the chart above for `val1`. On the next system clock cycle, the `led_seg <6:0>` output must be set to the cathode bits for value **`LED2`** and anode bits corresponding to those shown in the chart above for `val2`. Again, on the next system clock cycle, the `led_seg <6:0>` output must be set to the cathode bits for value **`LED7`** and anode bits corresponding to those shown in the chart above for `val3`. Again, on the next system clock cycle, the `led_seg <6:0>` output must be set to the cathode bits for value **`LED8`** and anode bits corresponding to those shown in the chart above for `val4`.

At this point, if we want the timer to continue displaying the correct time, we keep one of the segments lit up so to the user it still displays. If we want the timer to flash, we set the anode bits to all 1s to make the display go blank.

Understanding how the Nexys 3 7-segment display works with our parking meter, implementing the LED_controller submodule was straightforward. As inputs, the submodule took in the system clock, `rst`, `val1`, `val2`, `val3`, `val4`, and an ‘off’ wire that goes high when the 7-segment display should be blank. As outputs, the submodule returns anode bits `a1`, `a2`, `a3`, `a4`, and cathodes encapsulated in `led_seg <6:0>`. The “off” bit was created using combinatorial logic dependent on the current state in the overarching parking meter module.

In order to synchronize the display with the rest of the parking meter, I displayed 5 digits every *0.5sec*, or half clock cycle of the *T=1sec* clock. I did this in order to align the display with the common flashing cycles and to synchronize the underlying counter that decided which anode bit to set. I chose the 5th cycle to repeat `val4`.

3. Simulation Documentation

3.1 Testbench and Special Cases

In order to test my simulation, I created a testbench verilog module to run my parking_meter through sample test cases. While unable to test every combination of `add1`, `add2`, `add3`, and `add4`, I ran the parking meter through all special cases and edge cases. I document the special cases in the following list:

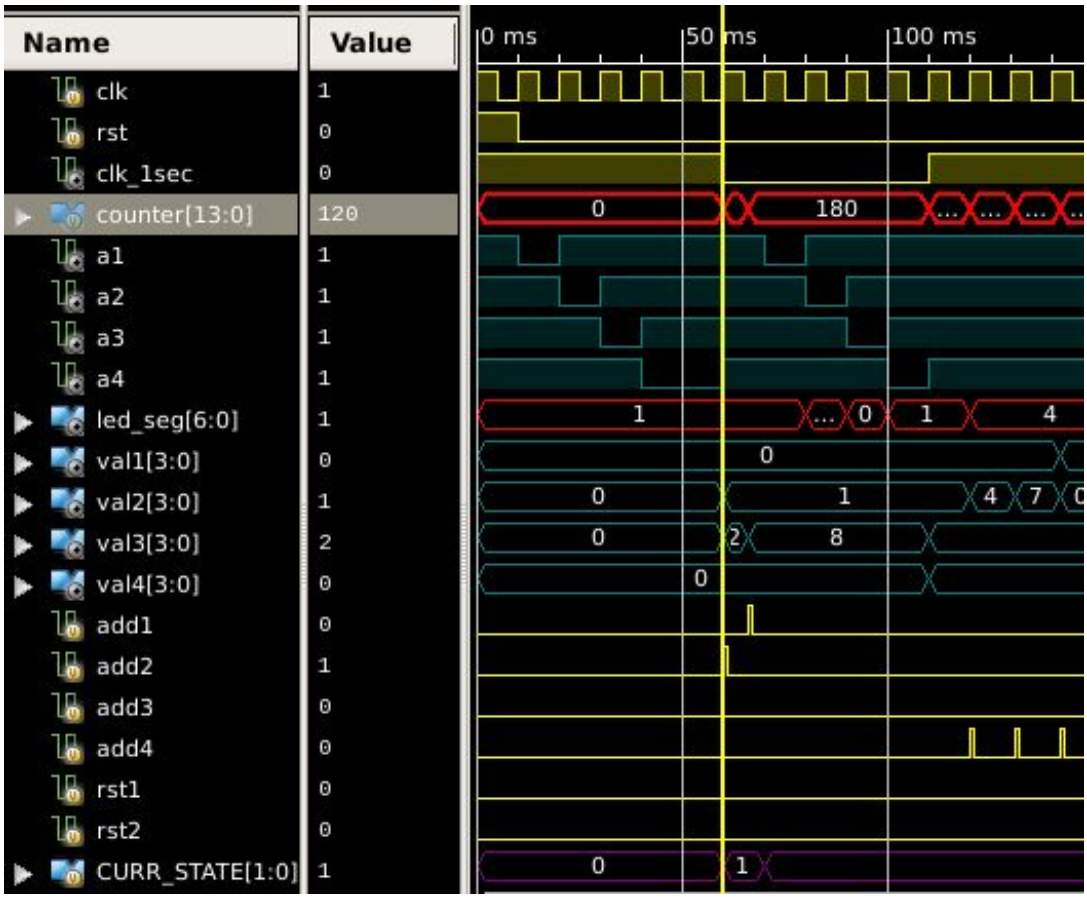
- Attempting to add time past 9999
- Counter decrementing below 0000
- Correct flashing periods in all three states
    - Reset_state *T = 1sec* 50% duty cycle flashing
    - State1 *T = 2sec* 50% duty cycle flashing
    - State2 no flashing, display number decrementing
- Use of all reset buttons: `rst`, `rst1`, and `rst2`.
- State change transition from 180 to 179 seconds remaining.
- State change transition from 1 to 0 seconds remaining.

I simulate my parking meter through all of these cases in one large testbench run. I do this by initializing my parking meter to the initial reset_state, and select `add1` and `add2` to demonstrate their correct use. After that, I use a for loop to continuously `add4` until the time remaining is constant at 9999. I then let the parking meter count down a few seconds to display the correct absence of flashing when the time exceeds 180 seconds.

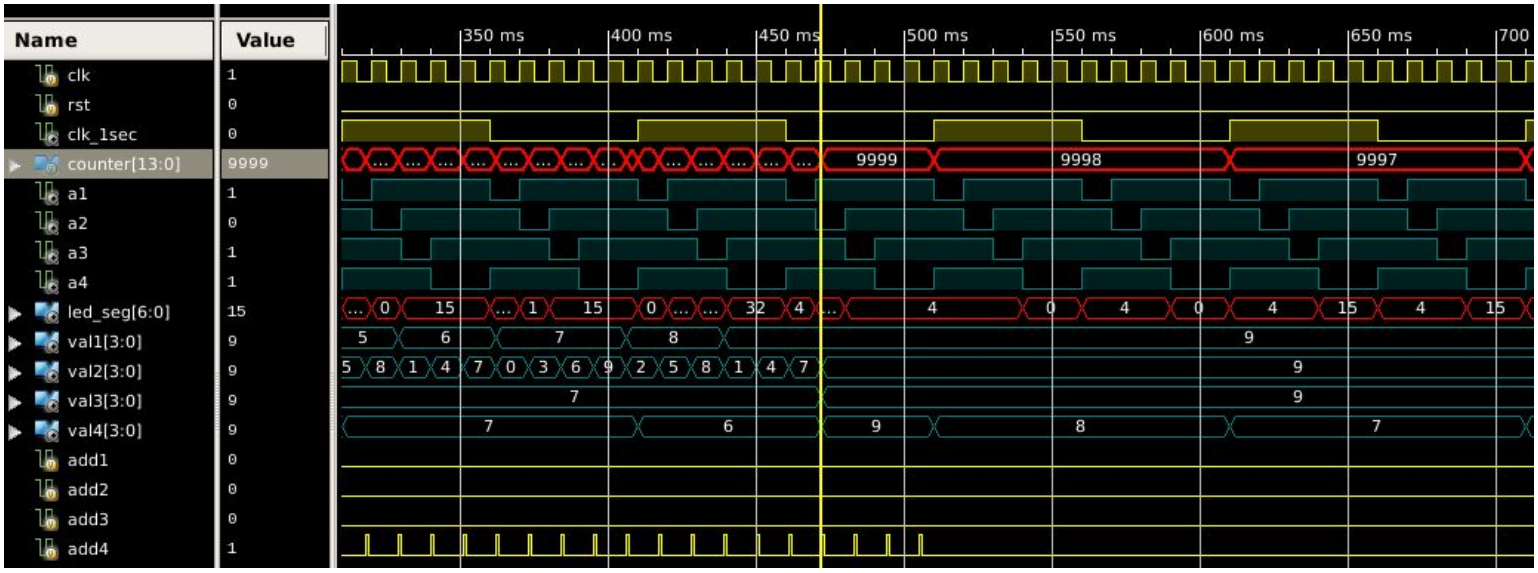
From here I select `rst2`, showing the correct updated time and flashing when the time remaining is less than 180 seconds. After this, I hard reset the parking meter by selecting `rst`, bringing the timer back to 0000 and displaying the correct flashing in the reset_state. Lastly, I `add3` from the reset_state to display the smooth state transition from 180 to 179. Selecting `rst1`, I finally reset the counter to 15 and let the time run down to 0 seconds. This displays the final special case, which is the state transition from 1 to 0 seconds remaining.

3.2 9999 Max and State2 Flashing

To begin the testbench, I use `add1`, `add2` and `add3` to increment the parking meter time to 9999. In doing this, I demonstrate the functionality of `add1`, `add2` and `add3`, as well as the successful 9999 timer maximum. The associated waveforms with this section are shown below.



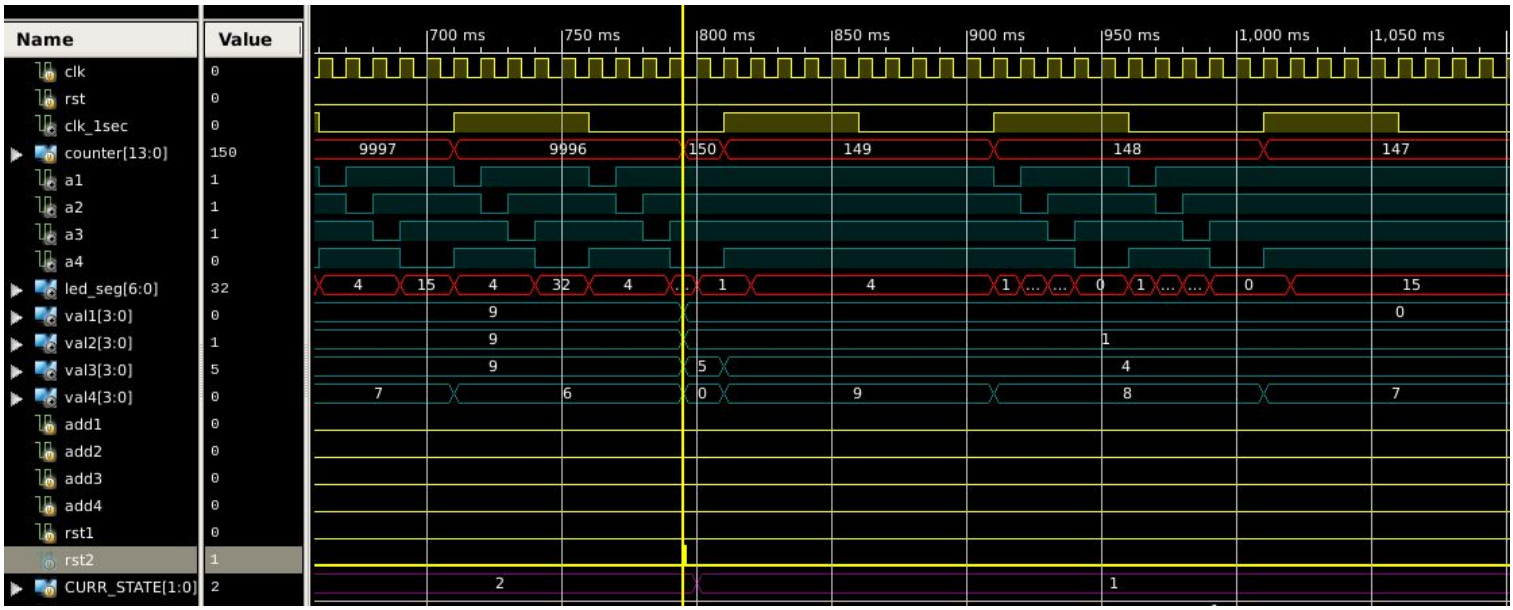
First, we show here the functional `add1` and `add2` bits. As shown in the waveform, when `add1` is selected, the counter value increases from 0120 to 0180, showing a change by 60. When `add2` is selected, the timer value increases from 0000 to 0120. We can also see a smooth state transition from the reset_state to state1 to state2.



In this waveform, we clearly see that while we continuously add 300 seconds to the timer, once the meter reaches 9999 it does not increase from here. Once we reach this maximum, our testbench lets the meter count down for a few seconds. As we see from the anodes `a1`, `a2`, `a3`, and `a4`, there is no flashing in this state. Each timer value is displayed constantly, and the timer decrements every clock cycle of the *1 Hz* clock.

3.3 rst2 Validation and State1 Flashing

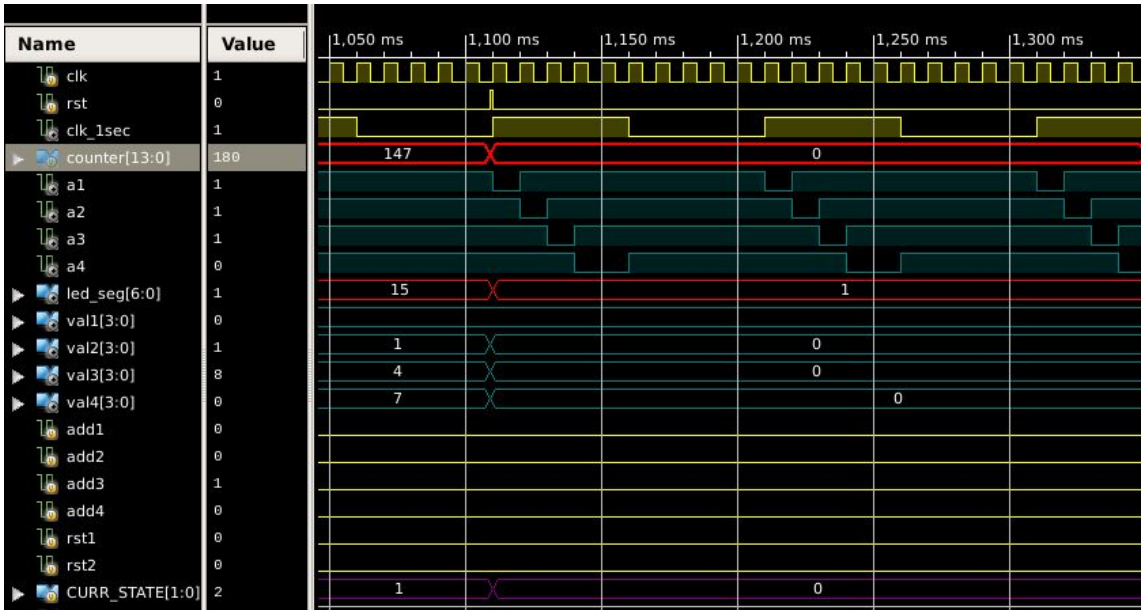
In the next section of the testbench, I select `rst2` to transition the timer from the 9000s to state1. In this section, we see a smooth transition from state2 to state1 as well as the correct state1 flashing display. The waveforms associated with this section are shown below. As per the parking meter spec, the meter flashes on and off with a period *T=2sec* 50% duty cycle, or every *1sec*.



Here we see `rst2` being pushed, and immediately the parking meter transitions to state1 and the correct digits are displayed. Furthermore, the even number 0150 is displayed, and correctly flashes off for the odd number of remaining time. This can be seen through the moment when the timer has 147 seconds remaining.

3.4 rst Validation and Reset_State Flashing

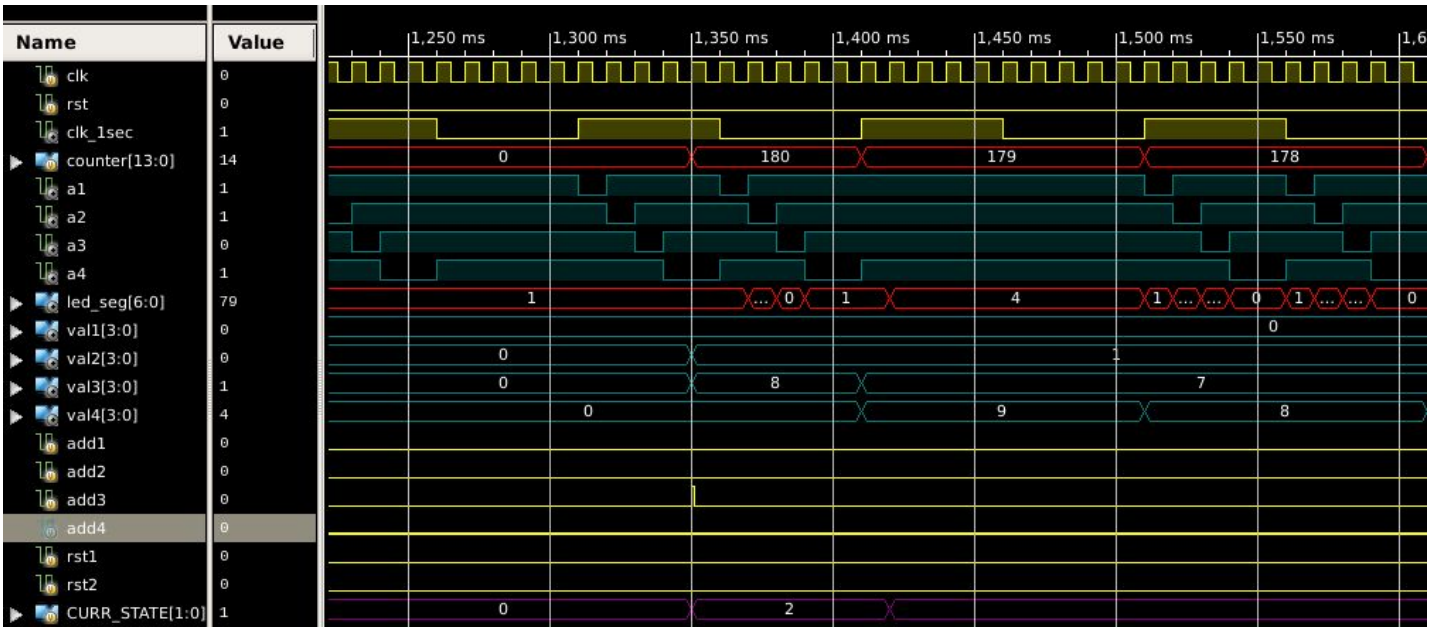
Next in the testbench we select `rst` in order to demonstrate the correct state transition, and the correct reset_state flashing. The associated waveforms for this section are:



As we can see, `rst` is selected early on in this waveform. Once set, the parking meter transitions to the initial state and updates the remaining time to 0000. This can be seen through the `val` bits below the red LED waveform. As per the parking meter spec, the meter flashes on and off with a period *T=1sec* 50% duty cycle, or every *0.5sec*.

3.5 State Transition - State2 to State1

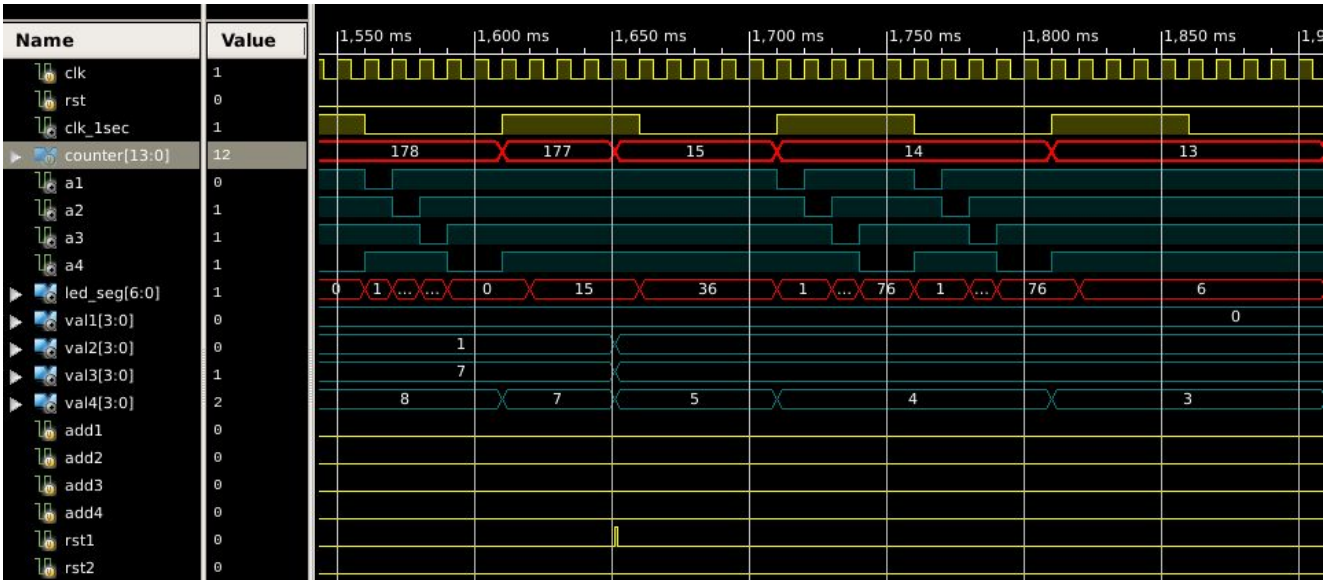
Next in the testbench, we demonstrate the parking meter's ability to process `add3` and the correct transition from 180 seconds remaining to 179 seconds remaining by the counter counting down. The associated waveform is shown below.



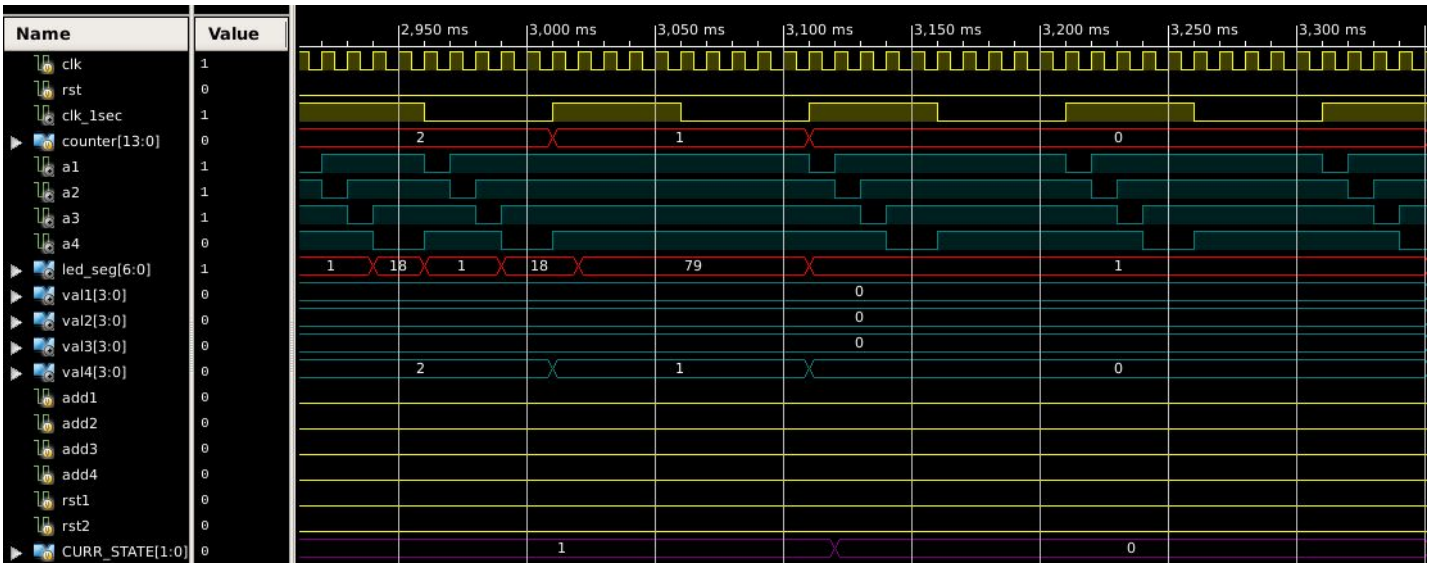
In this waveform, we see `add3` selected and immediately the counter is updated to 180 seconds remaining, with current state 'state2'. Since the current state is state2, 0180 is displayed. When the timer counts down and transitions from state2 into state1, the odd numbered 179 seconds is correctly not displayed. Therefore the state transition is smooth and successful, picking up the correct flashing requirements from the next state.

3.7 rst1 Validation and State Transition - State1 to Reset_State

Lastly, our testbench selects `rst1` to reset the timer back to 15 seconds, and allows the timer to run out. In this way, the functionality of `rst1` is demonstrated, as well as the state transition from state1 to the reset_state. The associated waveforms for this section are shown below.



In this waveform, we see `rst1` selected. As soon as this happens, the timer is updated with the current value of 0015 correctly, and continues to flash with the correct period, given that it is an odd value. From here, our testbench allows the meter to count down until 0000 is reached. The waveform showing this is shown next.

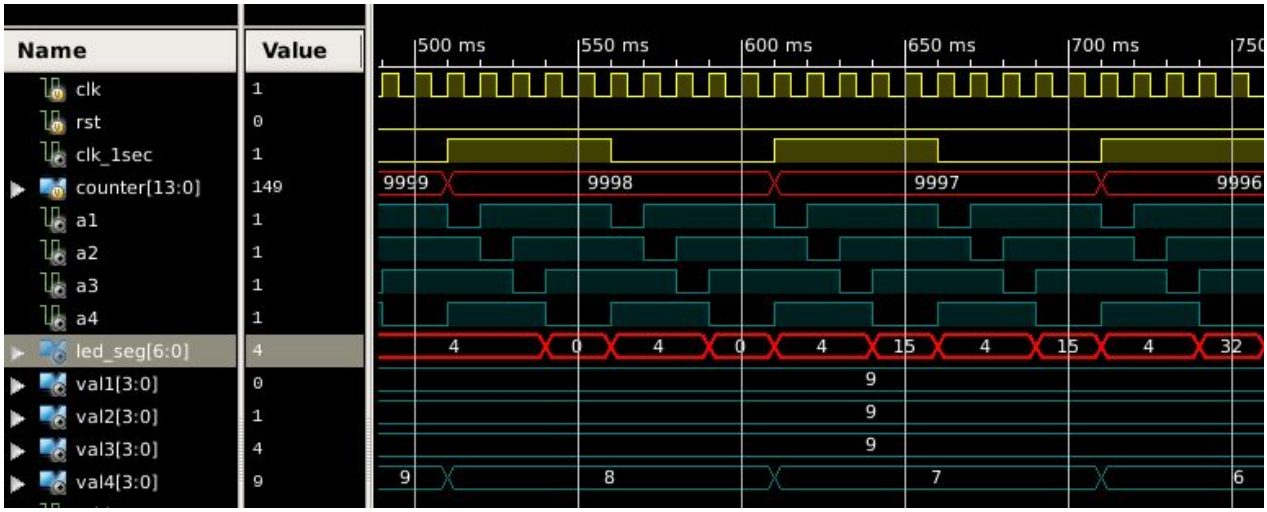


In this waveform, we see the timer flashing appropriately and decrementing to 0. Once the counter reaches 0, the correct reset_state flashing shown in an above waveform continues once again. In this way, there was a successful transition between the states when the meter counting down triggered the state change, and not a selected input.

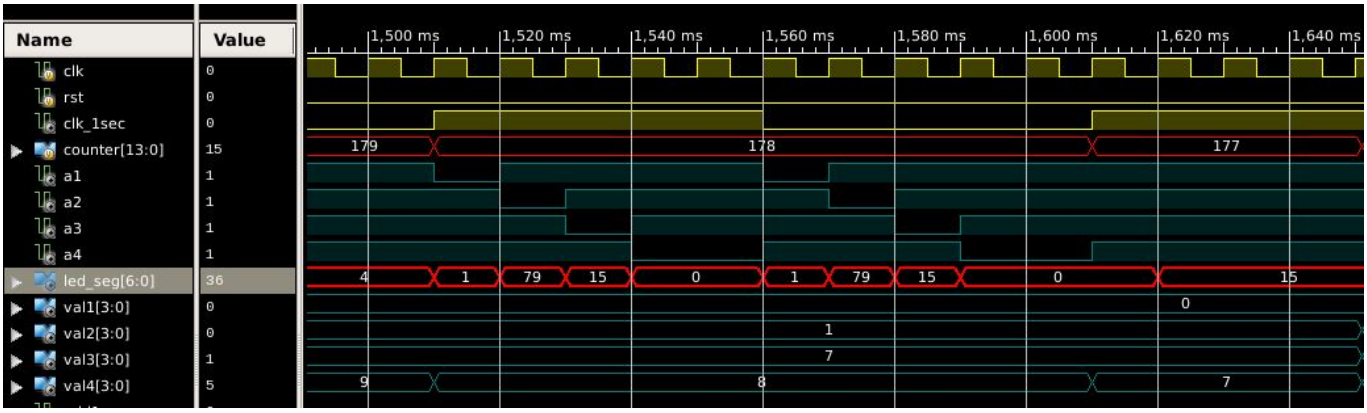
3.7 BCD and LED_controller Validation

For all of the previous simulation results, I relied on the accuracy of my `[13:0] counter` register to show the current time on the parking meter. This section investigates the accuracy of my counter with the `val1`, `val2`, `val3`, `val4`, and `led_seg <6:0>` waves displayed from simulating my parking meter with my testbench. Examples of these waveforms are shown below. As a reminder, the decimal representations of the LED values are shown here:

LED VALUE	Decimal Value
LED0	1
LED1	79
LED2	18
LED3	6
LED4	76
LED5	36
LED6	32
LED7	15
LED8	0
LED9	4



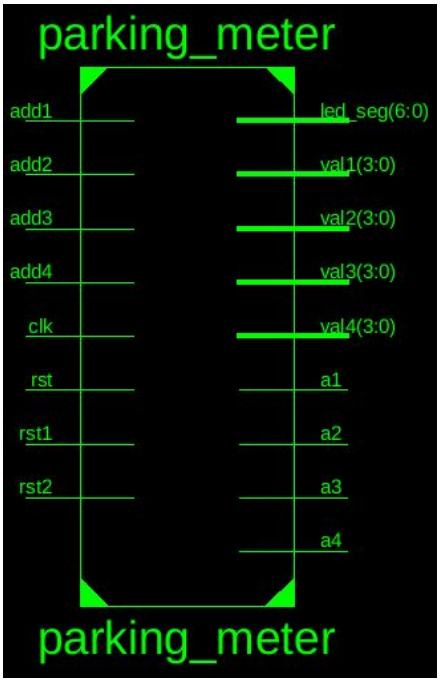
This screenshot was taken after allowing the counter to decrement from 9999, the maximum value. In this screenshot, the values of `val1`, `val2`, and `val3` remain at 9, which is correct. We can see in the chart that this value is represented as unsigned decimal 4, which is the value displayed in `led_seg <6:0>` throughout the clock cycles where `a1`, `a2`, and `a3` are set to 0. `val4` counts down with the counter correctly, and here we see the variation in `led_seg <6:0>` the clock cycles where `a4` is low. In this way, our LED and BCD submodules successfully perform their functions. Let's look at one more waveform.



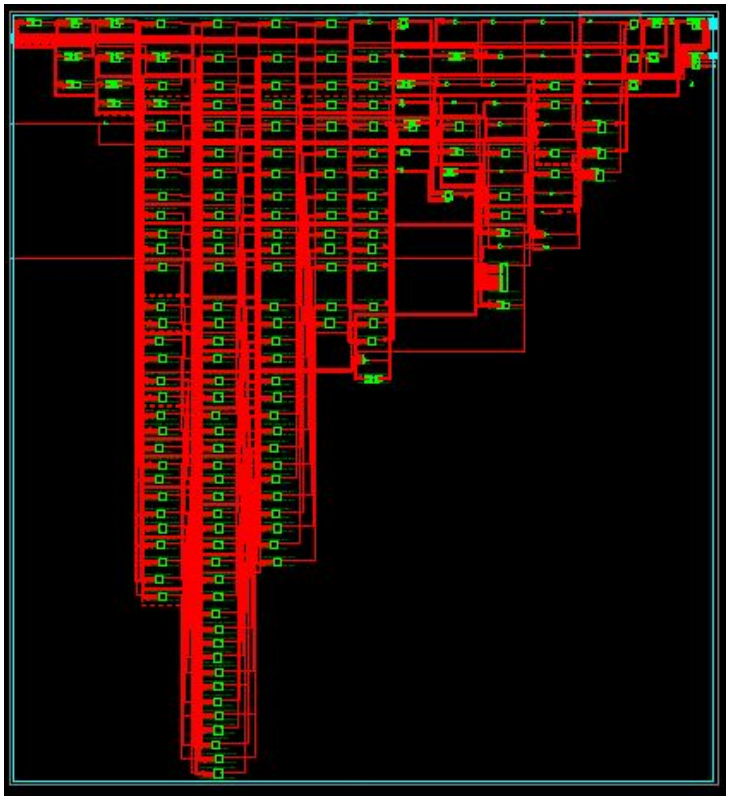
This screenshot gives us more values to test. From the chart, we see that unsigned decimals 1, 79, 15, and 0 correspond to LED values 0, 1, 7, and 8. Coupling these unsigned decimal values with the anode bits `a1`, `a2`, `a3`, and `a4` at each clock cycle, we confirm the values are equivalent to the values of `val1`, `val2`, `val3`, and `val4`.

4. Conclusion and Synthesis

After running our testbench and viewing the output waveforms, it is clear that our design and implementation of the parking_meter was a success. When synthesizing our verilog module, the high level RTL matches the one given to us in the spec. This is shown below:



When viewing the detailed RTL Schematic, the complexity of this abstract module becomes increasingly apparent. Shown below is a zoomed-out representation of our module being implemented by Xilinx ISE:



Most of these boxes shown are multiplexers (MUXs) and combinatorial logic. The RTL schematic shown above can be validated by the HDL Synthesis report output by the Xilinx ISE during Synthesis. The report is displayed below:

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================

Synthesizing Unit <parking_meter>.
    Related source file is "/home/ise/Xilinx_VM/Project5/parking_meter.v".
        reset_state = 2'b00
        state1 = 2'b01
        state2 = 2'b10
        add1val = 14'b00000000111100
        add2val = 14'b00000001111000
        add3val = 14'b00000010110100
        add4val = 14'b00000100101100
        maxval = 14'b10011100001111
        rst1val = 14'b00000000001111
        rst2val = 14'b00000010010110
        notime = 14'b00000000000000
    Found 3-bit register for signal <off_count>.
    Found 1-bit register for signal <zero_off>.
    Found 1-bit register for signal <one_off>.
    Found 1-bit register for signal <two_off>.
    Found 2-bit register for signal <CURR_STATE>.
    Found 1-bit register for signal <counter<13>>.
    Found 1-bit register for signal <counter<12>>.
    Found 1-bit register for signal <counter<11>>.
    Found 1-bit register for signal <counter<10>>.
    Found 1-bit register for signal <counter<9>>.
    Found 1-bit register for signal <counter<8>>.
    Found 1-bit register for signal <counter<7>>.
    Found 1-bit register for signal <counter<6>>.
    Found 1-bit register for signal <counter<5>>.
    Found 1-bit register for signal <counter<4>>.
    Found 1-bit register for signal <counter<3>>.
    Found 1-bit register for signal <counter<2>>.
    Found 1-bit register for signal <counter<1>>.
    Found 1-bit register for signal <counter<0>>.
    Found 14-bit subtractor for signal <counter[13]_GND_1_o_sub_54_OUT> created at
line 233.
    Found 15-bit adder for signal <n0232> created at line 131.
    Found 15-bit adder for signal <n0233> created at line 138.
    Found 14-bit adder for signal <counter[13]_GND_1_o_add_43_OUT> created at line
209.
    Found 14-bit adder for signal <counter[13]_GND_1_o_add_49_OUT> created at line
221.
```

```
    Found 3-bit adder for signal <off_count[2]_GND_1_o_add_57_OUT> created at line
249.
    Found 1-bit 3-to-1 multiplexer for signal <CURR_STATE[1]_NEXT_STATE[1]_Mux_24_o>
created at line 116.
    Found 1-bit 4-to-1 multiplexer for signal <CURR_STATE[1]_GND_11_o_Mux_25_o>
created at line 116.
    Found 1-bit 3-to-1 multiplexer for signal <CURR_STATE[1]_NEXT_STATE[0]_Mux_26_o>
created at line 116.
    Found 15-bit comparator greater for signal <BUS_0001_GND_1_o_LessThan_9_o> created
at line 131
    Found 15-bit comparator greater for signal <BUS_0002_GND_1_o_LessThan_12_o>
created at line 138
    Found 14-bit comparator greater for signal <counter[13]_GND_1_o_LessThan_20_o>
created at line 160
    Found 14-bit comparator greater for signal <PWR_1_o_counter[13]_LessThan_31_o>
created at line 182
    Found 14-bit comparator greater for signal <PWR_1_o_counter[13]_LessThan_37_o>
created at line 194
    Found 14-bit comparator greater for signal <PWR_1_o_counter[13]_LessThan_43_o>
created at line 206
    Found 14-bit comparator greater for signal <PWR_1_o_counter[13]_LessThan_49_o>
created at line 218
    Summary:
       inferred   6 Adder/Subtractor(s).
       inferred  22 D-type flip-flop(s).
       inferred   2 Latch(s).
       inferred   7 Comparator(s).
       inferred 148 Multiplexer(s).
Unit <parking_meter> synthesized.
```

…

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Multipliers                                          : 6
 10x4-bit multiplier                                   : 1
 4x4-bit multiplier                                    : 3
 7x4-bit multiplier                                    : 2
# Adders/Subtractors                                   : 56
 14-bit adder                                          : 23
 14-bit subtractor                                     : 1
 15-bit adder                                          : 5
 15-bit subtractor                                     : 2
 16-bit adder                                          : 3
 17-bit adder                                          : 3
 18-bit adder                                          : 3
 19-bit adder                                          : 2
 20-bit adder                                          : 2
```

```
 21-bit adder                                          : 2
 22-bit adder                                          : 1
 23-bit adder                                          : 1
 24-bit adder                                          : 1
 3-bit adder                                           : 3
 32-bit subtractor                                     : 1
 4-bit subtractor                                      : 3
# Registers                                            : 28
 1-bit register                                        : 23
 2-bit register                                        : 1
 3-bit register                                        : 3
 7-bit register                                        : 1
# Latches                                              : 2
 1-bit latch                                           : 2
# Comparators                                          : 52
 14-bit comparator greater                             : 5
 14-bit comparator lessequal                           : 24
 15-bit comparator greater                             : 2
 15-bit comparator lessequal                           : 3
 16-bit comparator lessequal                           : 3
 17-bit comparator lessequal                           : 3
 18-bit comparator lessequal                           : 3
 19-bit comparator lessequal                           : 2
 20-bit comparator lessequal                           : 2
 21-bit comparator lessequal                           : 2
 22-bit comparator lessequal                           : 1
 23-bit comparator lessequal                           : 1
 24-bit comparator lessequal                           : 1
# Multiplexers                                         : 628
 1-bit 2-to-1 multiplexer                              : 606
 1-bit 3-to-1 multiplexer                              : 2
 1-bit 4-to-1 multiplexer                              : 1
 14-bit 2-to-1 multiplexer                             : 14
 3-bit 2-to-1 multiplexer                              : 3
 7-bit 2-to-1 multiplexer                              : 1
 7-bit 4-to-1 multiplexer                              : 1

=========================================================================
```

When synthesizing my verilog files, the synthesis report passed without errors - but the report did contain some warnings that detailed inconsistencies with some sensitivity lists. While the simulation proved successful, without a FPGA Board to implement my design on, my solutions cannot be truly verified as accurate in regards to timing issues.

Furthermore, an interesting thing I noticed was that the synthesis did not report that Xillinx found a finite state machine in my design. In this way, my finite state machine may not be flawlessly defined. Next, we investigate the Map report which details the implementation of my design on an FPGA board. The Map report is shown below:

Design Summary

Design Summary:

Number of errors: 0
Number of warnings: 15

Slice Logic Utilization:

Number of Slice Registers:	71	out of	18,224	1%
Number used as Flip Flops:	54			
Number used as Latches:	16			
Number used as Latch-thrus:	0			
Number used as AND/OR logics:	1			
Number of Slice LUTs:	460	out of	9,112	5%
Number used as logic:	457	out of	9,112	5%
Number using O6 output only:	362			
Number using O5 output only:	12			
Number using O5 and O6:	83			
Number used as ROM:	0			
Number used as Memory:	0	out of	2,176	0%
Number used exclusively as route-thrus:	3			
Number with same-slice register load:	0			
Number with same-slice carry load:	3			
Number with other load:	0			

Slice Logic Distribution:

Number of occupied Slices:	161	out of	2,278	7%
Number of MUXCYs used:	68	out of	4,556	1%
Number of LUT Flip Flop pairs used:	461			
Number with an unused Flip Flop:	393	out of	461	85%
Number with an unused LUT:	1	out of	461	1%
Number of fully used LUT-FF pairs:	67	out of	461	14%
Number of unique control sets:	51			
Number of slice register sites lost to control set restrictions:	338	out of	18,224	1%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element.
The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs:	35	out of	232	15%

Specific Feature Utilization:

Number of RAMB16BWERs:	0	out of	32	0%
Number of RAMB8BWERs:	0	out of	64	0%
Number of BUFIO2/BUFIO2_2CLKs:	0	out of	32	0%
Number of BUFIO2FB/BUFIO2FB_2CLKs:	0	out of	32	0%
Number of BUFG/BUFGMUXs:	2	out of	16	12%

```
    Number used as BUFGs:                       2
    Number used as BUFGMUX:                     0
  Number of DCM/DCM_CLKGENs:                    0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                  0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:        0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                  0 out of     248    0%
  Number of BSCANs:                             0 out of       4    0%
  Number of BUFHs:                              0 out of     128    0%
  Number of BUFPLLs:                            0 out of       8    0%
  Number of BUFPLL_MCBs:                        0 out of       4    0%
  Number of DSP48A1s:                           5 out of      32   15%
  Number of ICAPs:                              0 out of       1    0%
  Number of MCBs:                               0 out of       2    0%
  Number of PCILOGICSEs:                        0 out of       2    0%
  Number of PLL_ADVs:                           0 out of       2    0%
  Number of PMVs:                               0 out of       1    0%
  Number of STARTUPs:                           0 out of       1    0%
  Number of SUSPEND_SYNCs:                      0 out of       1    0%

Average Fanout of Non-Clock Nets:               3.89

Peak Memory Usage:  773 MB
Total REAL time to MAP completion:  27 secs
Total CPU time to MAP completion:   26 secs

Mapping completed.
```

The map report details the specific design of my module implementation in multiplexers, latches, and combinatorial gates. Although not shown in the design summary, the map report also displays the I/O pinnings to be used on an FPGA board, which would help me physically implement my design if I wanted to truly create the parking_meter.