I. Introduction

Finite state machines are used prolifically in digital electronics as mechanisms for controlling system input and output. State machines can be seen in various everyday electronic objects such as traffic lights, vending machines, and ATM machines.

In this project, we will explore state machines in detail by designing a functional prototype of a parking meter. The parking meter takes input from a user to add time to the and reset to certain times. Internally, our parking meter handles counting down the time remaining at 1Hz. Finally, the parking meter will display the time remaining on the 4 seven-segment displays on our board. A high-level view of the input and output is shown below.

Inputs	Function
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
clk	frequency of 100 Hz
rst	resets to the initial state

Figure 1: *Input description of parking meter*

Outputs	Description
val1	Value in thousands position (1st seven-segment LED display)
val2	Value in hundreds positions (2nd seven-segment LED display)
val3	Value in tens position (3rd seven-segment LED display)
val3	Value in ones position (4th seven-segment LED display)
led_cathodes	7-bits used to control number displayed on LED display
led_anodes	4-bits used to multiplex between 4 displays

Figure 2: *Output description of parking meter*

II. Design

The core logic of the parking meter is handled by a Mealy finite state machine that depends on both the state and the input. Additionally, two submodules – BCD_converter and LED_controller – are used to support the output of the device. All timing is done within a within a timing control block called Timing_controller.

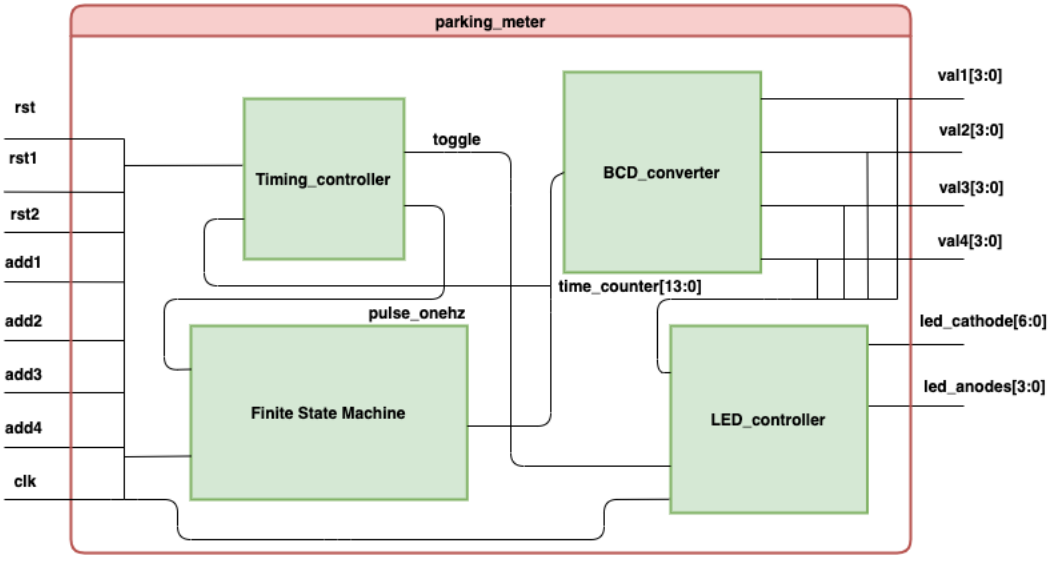


Figure 3: *Detailed high-level design for parking meter* - The parking meter is comprised of a Mealy FSM, two submodules for output logic, and a timing control block that determines when to flash and when to decrement the counter.

III. Implementation

a. Finite State Machine

Our Mealy FSM has a total of 8 states, implemented in the following manner

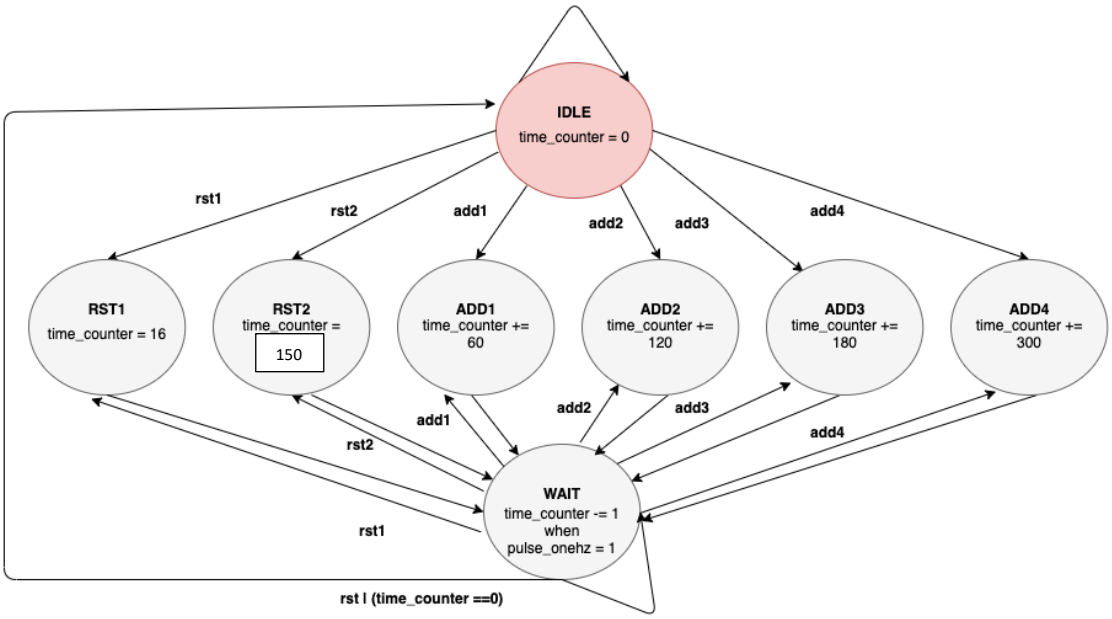


Figure 4: *Finite State Machine* – State machine with 8 states that handles all the possible inputs into the system. The red state – IDLE – is the default state of the machine. Additionally, the 4 add stages also accounts for when adding to time_counter exceeds 9999. Arrows with no label, refers to behavior of the FSM when none of the other inputs are active.

- **IDLE:** The initial state of the machine. When in this state, we set the time_counter to zero. When there is any input is active that is not rst, the next state corresponds to that inputs name and action. When rst is pushed in any state, we return to this state.
- **RST1:** State that resets the time_counter to 16 seconds. The next stage is the transition and decrementing stage, WAIT.
- **RST2:** State that resets the time_counter to 150 seconds. The next state is WAIT.
- **ADD1:** State that adds 60 seconds to time_counter. The next state is WAIT.
- **ADD2:** State that adds 120 seconds to time_counter. The next state is WAIT.
- **ADD3:** State that adds 180 seconds to time_counter. The next state is WAIT.
- **ADD4:** State that adds 300s seconds to time_counter. The next state is WAIT.
- **WAIT:** Transition and decrementing stage. The stage decrements the time_counter whenever the pulse_onehz input signal from the Timing_controller block is active. If any input is active during this state, the next state will be the inputs corresponding state. The next state is also set to IDLE when the time_counter is 0.

Note: For ADD? states there is logic to handle the case when adding causes time_counter to exceed 9999. In this case, the time_counter is set to 9999.

b. Timing_controller Block

The Timing_controller block was implemented as another procedural block in the top-level module. However, it helps to be able to visualize it as another submodule with inputs and outputs.

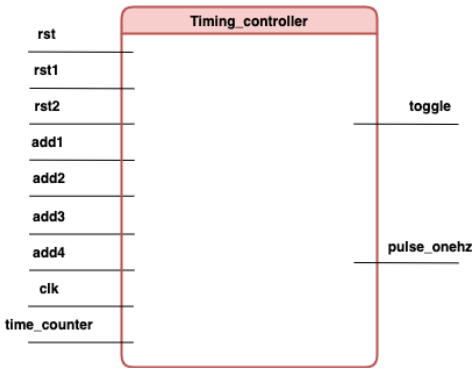


Figure 5: *Timing controller block*

- ***Input*** – The input is similar to the to the top module with the addition on a time_counter
  - When rst, rst1, or rst2 is pushed we reset the internal counter
  - When add1, add2, add3, or add4 is pushed and time_counter is equal to zero, reset internal counter
  - Resets ensure that each decrement is exactly 1 second apart
- **Output**
  - *toggle* – Variable clock used for flashing control
    - 1 Hz (1 sec period) when time_counter == 0
    - ½ Hz (2 sec period) when 0 < time_counter ≤ 180
    - 1 when time_counter > 180
  - *pulse_onehz* – Pulse used to decrement time_counter in FSM
    - Output pulse when 100 cycles of input clock (100Hz) passes

c. BCD_converter

The BCD_converter simply takes in the time_counter as input and converts it to binary-coded decimal with the modulo and division operations.

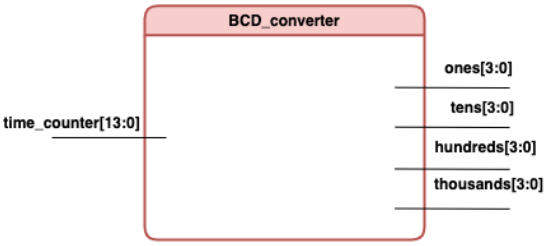


Figure 6: *BCD_converter*

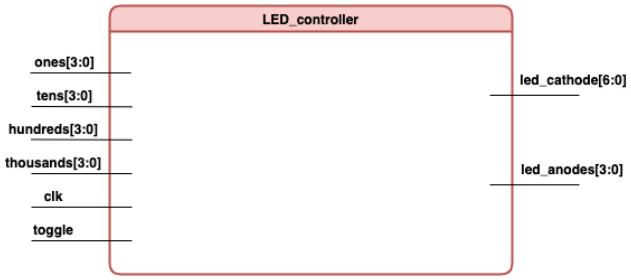


Figure 7: *LED_controller* – Controls the 4 seven-segment LEDs and the flashing

d. LED_controller

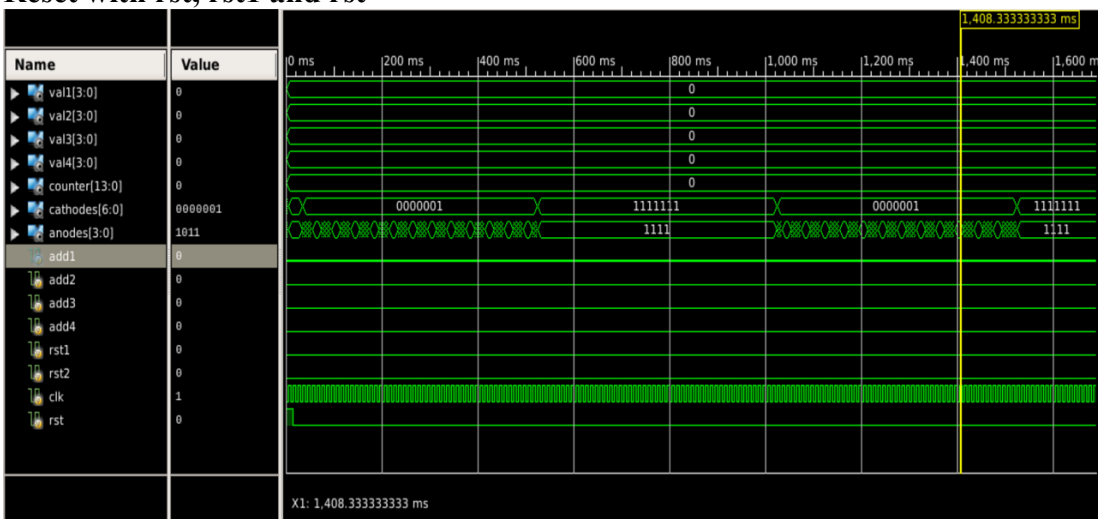
The LED_controller encodes val1-val4 into the 7-bit cathode value used to control the number displayed on the seven-segment LED display. Toggle is the input used to control the flashing logic. When toggle is active, we multiplex across the 4 seven-segment LEDs with the led_anodes. Multiplexing is done by selecting the displays one by one (this is done with the led_anodes by setting it to 1110, 1101, 1011, and 0111). Since we multiplex with a 100 Hz clock the display is refreshed at 25Hz. When the toggle is not active, we set the led_anodes to 4'b1111 and the led_cathodes to 4'b1111111. This creates the flashing behavior that we want since the anodes and the cathodes of the LEDs are active low.

IV. Testbench

For our testbench, a 100Hz clock was stimulated to match the specification of the project description. Though there is no mention of a need to output the counter value, it has been included to allow greater readability of the waveforms.

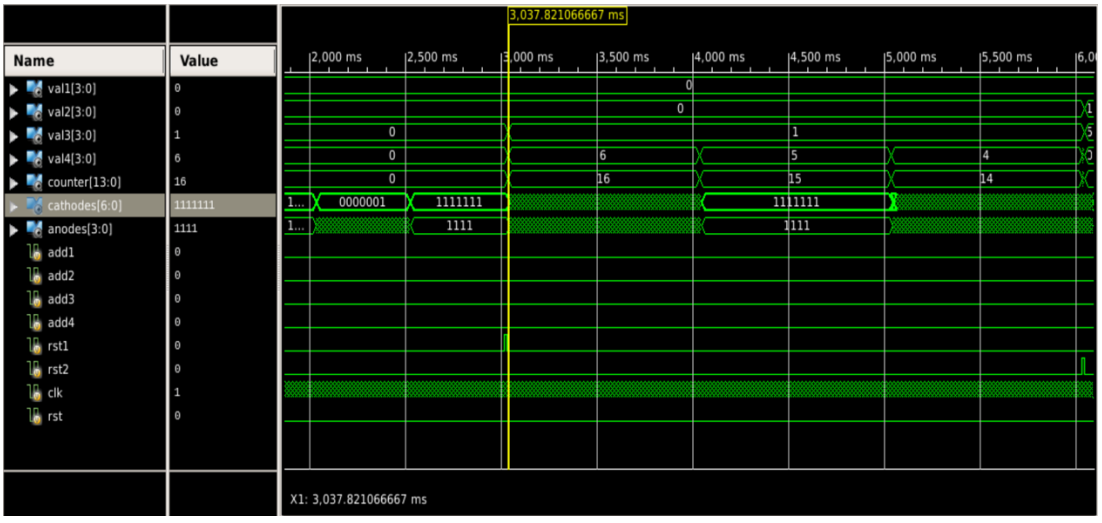
Base Cases

a. Reset with rst, rst1 and rst



Waveform 1: *Reset to 0 (rst)*

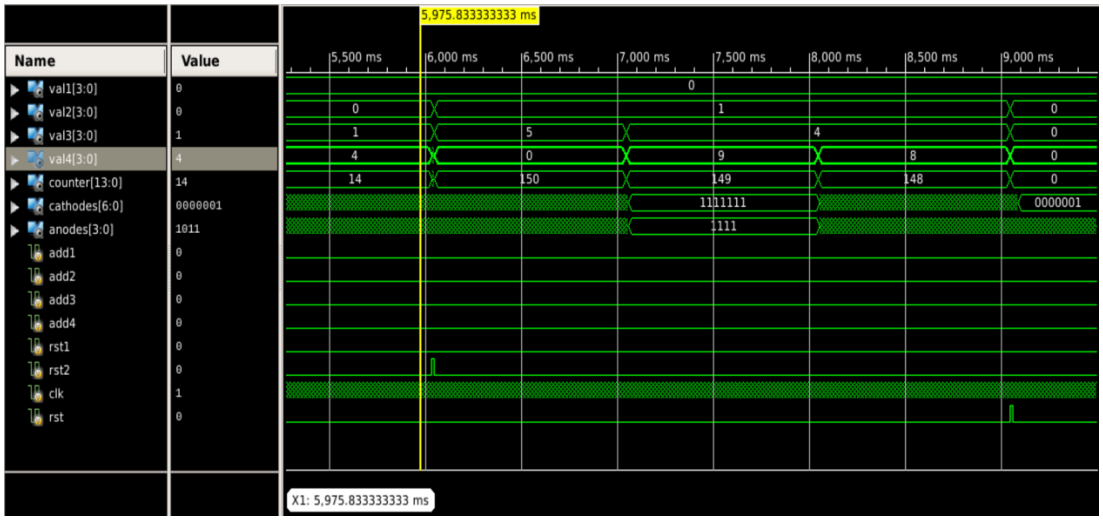
- Resets counter to 0 and flashes 0000 with a period of 1 second and 50% duty cycle
- Cathodes output is correct since 0000001 corresponds to middle light off (= # 0)
- Anodes output switches from multiplexing output (unclear result occurs due to switching between 1110, 1101, 1011, and 0111 every 10 ms) to off (1111) every 500 ms



Waveform 2: *Reset to 16 (rst1)*

- Resets counter to 16 and flashes with a period of 2 seconds and 50% duty cycle
- The counter is decremented every 1 second

- The time in which anode switches from multiplexing to off is now 1 second, which is consistent with flashing at a period of 2 seconds and 50% duty cycle
- Another way to see this is to see that the even numbers are displayed, and the odd numbers are hidden

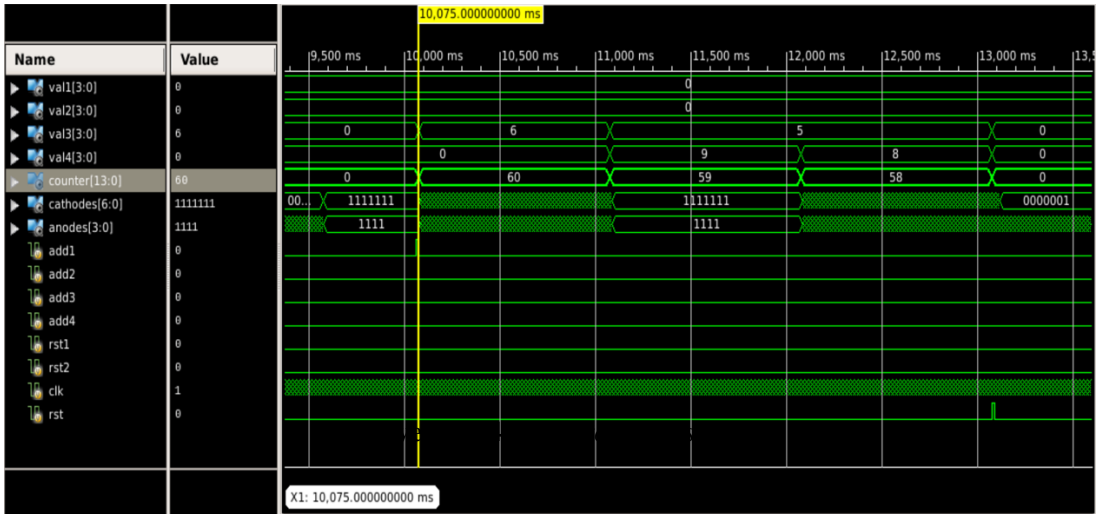


Waveform 3: *Reset to 150 seconds (rst2)*

- Resets counter to 150 and flashes with a period of 2 seconds and 50% duty cycle
- The counter is decremented every 1 second

b. Add with add1 and add4 from 0

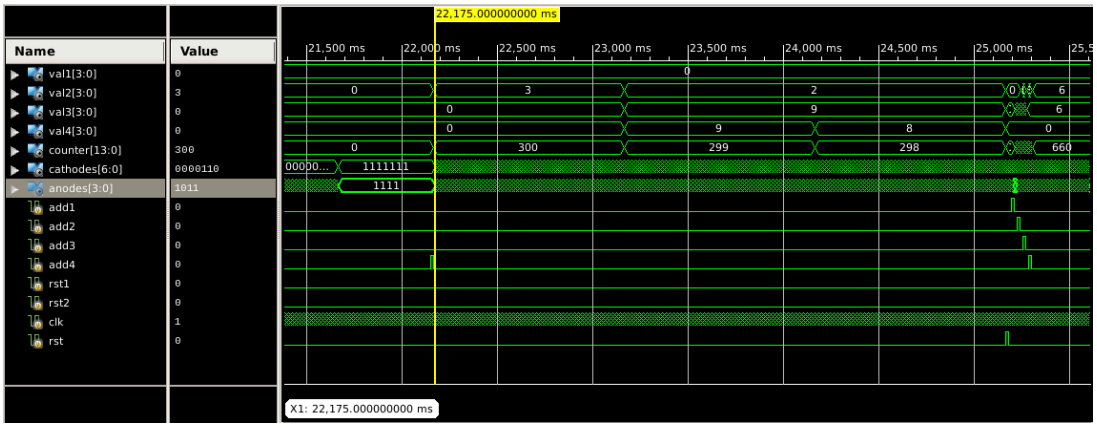
The waveforms for add2 and add3 are not included, because they are very similar to the waveform for add1 with the BCD values and the cathodes representing the different counter values.



Waveform 4: *Add 60 seconds (add1)*

- Adds 60 to counter and flashes with a period of 2 seconds and 50% duty cycle
- The counter is decremented every 1 second
- Here we can see the importance of resetting the timing control when add is pressed and the counter is 0

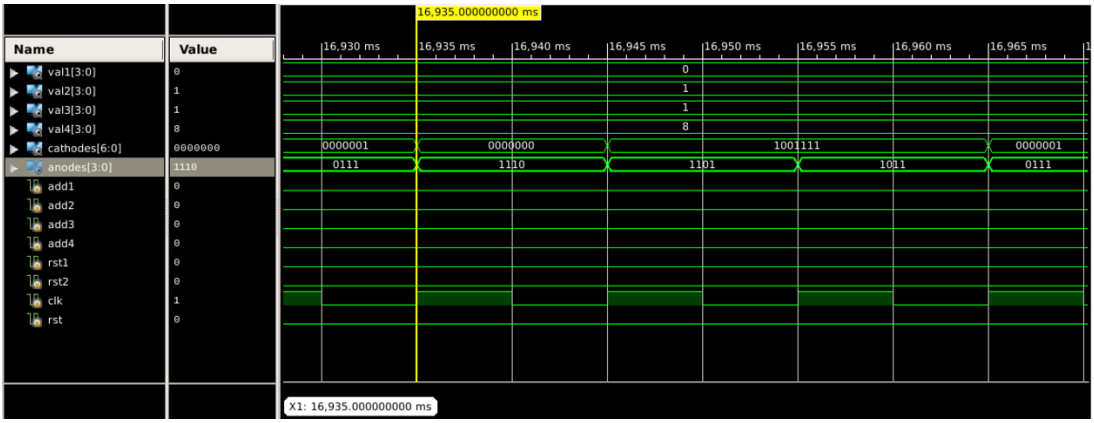
- It ensures that each number is decremented after 1 second and the even numbers are displayed while the odd numbers are skipped



Waveform 5: *Add 300 seconds (add4)*

- Adds 300 to counter and displays digits continuously since counter is greater than 180
- When the counter is greater than 180, we multiplex continuously and refresh the display at 25 Hz (it takes 40ms to multiplex all 4 displays)

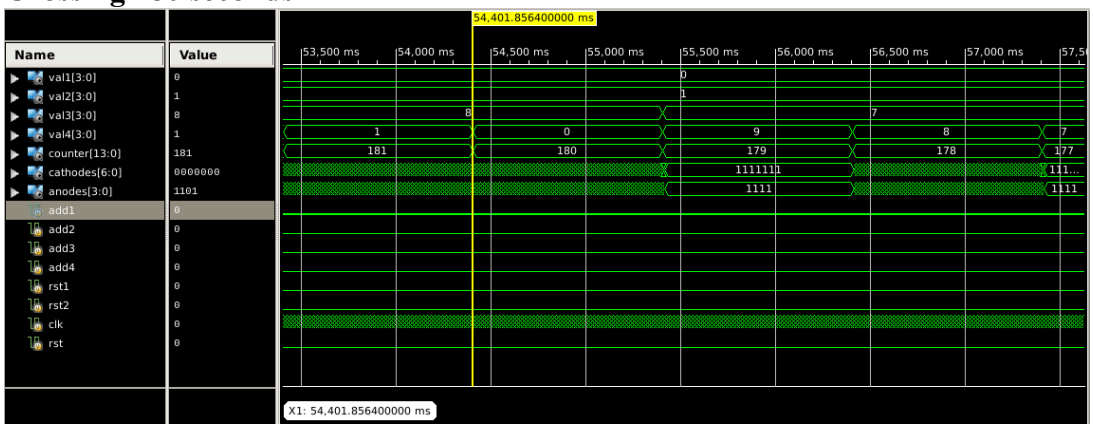
c. Check Cathodes and Anodes Check



Waveform 6: *Cathodes and anodes behavior check with count equal to 118*

- Anodes outputted corresponds to the ones position (1110) at 16,935 ms
- Cathodes 0000000 corresponds to 8 since all the lights have to be lit
- The anodes outputted in the next two periods correspond to the tens and hundreds positions (1101 and 1011)
- Cathodes 1001111 for the two periods is correct as well since only the two rightmost LEDs are lit when displaying 1

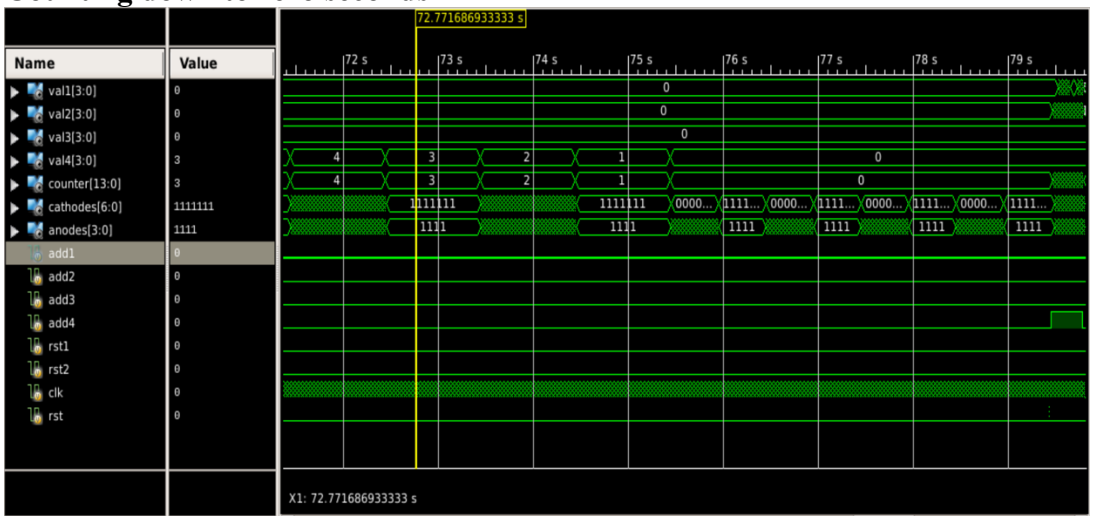
d. Crossing 180 seconds



Waveform 7: *Behavior of parking meter when we decrement into 180 seconds*

- Prior to the count being equal to 180 seconds, the display is refreshed continuously
- When we reach 180 seconds, the display flashes with 2 second periods at 50% duty cycle
- Even numbers are displayed while odd numbers are skipped (anode == 1111)

e. Counting down to zero seconds

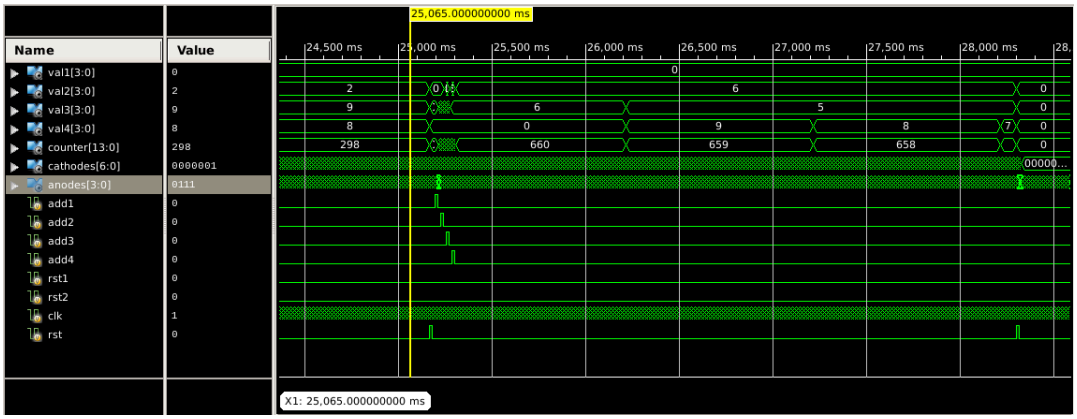


Waveform 7: *Counting down to zero*

- At around 75.5 s, the display counts down to zero
- Prior to zero, the display flashes with 2 second periods at 50% duty cycle
- At zero the display begins to flash with 1 second periods at 50% duty cycle
- In the first half cycle from around 75.5-76.0 seconds, we can see from the cross-hatched output of the anodes that the display is being multiplexed
- In the second half cycle from around 76.0-76.5 seconds, the display is off

Edge Cases

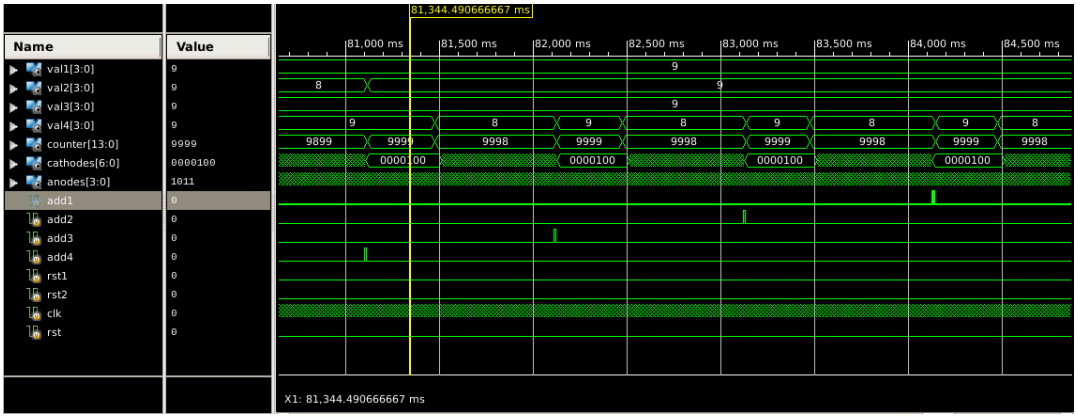
a. Consecutive adds



Waveform 8: *Consecutive adds* – Press add1, add2, add3, and add4 consecutively. Though this would be a little fast in real life, we press simulate pressing the buttons 30 ms apart.

- We begin adding at 25,400 ms with add1
- We follow the first add1 with the other adds one after the other in 30 ms intervals
- The cumulative sum of the counter, 660 seconds, is correct
- During the adding, the anodes and the cathodes output is also correct, since the count is greater than 180 seconds
- Note that the count is decremented at 26,400 ms, which means that pulse_onehz is only reset during the first add
- This is correct behavior since we do not want to reset on each consecutive add, which will cause the decrement to take longer than 1 second

b. Add exceeds 9999



Waveform 8: *Adding overflow* – When adding exceeds 9999, we want to set the counter to 9999.

- Add inputs are stimulated each approximately one second apart
- In all the cases, the add operation would exceed 9999
- However, in each case the count is limited to 9999
- Another important detail to note is that, the add operations do not reset the pulse_onehz
- Decrementing still happens every 1 second

IV. Design Summary

Synthesis Report Design Summary

```
=========================================================
*                    Design Summary                     *
=========================================================

Top Level Output File Name         : parking_meter.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                              : 898
#      GND                         : 1
#      INV                         : 3
#      LUT1                        : 2
#      LUT2                        : 22
#      LUT3                        : 34
#      LUT4                        : 51
#      LUT5                        : 140
#      LUT6                        : 460
#      MUXCY                       : 75
#      MUXF7                       : 27
#      VCC                         : 1
#      XORCY                       : 82
# FlipFlops/Latches                : 68
#      FDC                         : 42
#      FDR                         : 13
#      FDRE                        : 2
#      FDS                         : 11
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 34
#      IBUF                        : 7
#      OBUF                        : 27

Device utilization summary:
---------------------------

Selected Device : 6slx16csg324-3


Slice Logic Utilization:
 Number of Slice Registers:          68  out of  18224     0%
 Number of Slice LUTs:              712  out of   9112     7%
    Number used as Logic:           712  out of   9112     7%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:   736
   Number with an unused Flip Flop:    668  out of    736    90%
   Number with an unused LUT:           24  out of    736     3%
   Number of fully used LUT-FF pairs:   44  out of    736     5%
   Number of unique control sets:        6

IO Utilization:
 Number of IOs:                  35
 Number of bonded IOBs:          35  out of    232    15%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:        1  out of     16     6%
```

Map Report Design Summary

```
Design Summary
--------------
Number of errors:     0
Number of warnings:   0
Slice Logic Utilization:
  Number of Slice Registers:              68 out of  18,224    1%
    Number used as Flip Flops:            68
    Number used as Latches:               0
    Number used as Latch-thrus:           0
    Number used as AND/OR logics:         0
  Number of Slice LUTs:                  715 out of   9,112    7%
    Number used as logic:                714 out of   9,112    7%
      Number using O6 output only:       681
      Number using O5 output only:         2
      Number using O5 and O6:             31
      Number used as ROM:                  0
    Number used as Memory:                 0 out of   2,176    0%
    Number used exclusively as route-thrus:     1
      Number with same-slice register load:     0
      Number with same-slice carry load:        1
      Number with other load:                   0

Slice Logic Distribution:
  Number of occupied Slices:             252 out of   2,278   11%
  Number of MUXCYs used:                  84 out of   4,556    1%
  Number of LUT Flip Flop pairs used:    733
    Number with an unused Flip Flop:     668 out of    733   91%
    Number with an unused LUT:            18 out of    733    2%
    Number of fully used LUT-FF pairs:    47 out of    733    6%
    Number of unique control sets:        5
    Number of slice register sites lost
      to control set restrictions:       28 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:                1 out of      16    6%
    Number used as BUFGs:                 1
    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:                     0 out of      32    0%
  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%
```

When we synthesize the code into hardware and map to the device, we can see that the parking meter utilizes 68 registers. This is expected as the logic for the system relies heavily upon

sequential logic. We can also see in our mapping report that the code utilizes 715 slice LUTs (Lookup Tables), which serves as the building blocks for combinatorial logic. This is also consistent with the content of parking meter since it also relies upon combinatorial logic to convert our binary counter to BCD, encode our BCD to its led_cathode equivalent, and determine the next state in our FSM.

V. Conclusion

parking_meter Project Status (11/29/2020 - 07:10:30)			
Project File:	Project3.xise	**Parser Errors:**	No Errors
Module Name:	parking_meter	**Implementation State:**	Placed and Routed
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	7 Warnings (0 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Figure 8 : *High Level Design Summary* - 7 warnings due to truncation warnings

The purpose of this project was to utilize the principles of finite state machines to build a working prototype of a parking meter. The project merges together many different aspects of the digital design. From this project, I learned how to implement a FSM in Verilog and how to synchronize different modules in Verilog together. Additionally, I learned how to multiplex in Verilog to control the 4 seven-segment displays on our device. The main challenge with this project was implementing the timing mechanism for flashing. Initially, I built two different submodules that outputted a 1Hz clock and a 0.5Hz. However, having 3 clocks to work with make timing and synchronization very confusing. When I aggregated all the timing control into a single block, it made synchronization a lot easier.