Danning Yu, 305087992
CS M152A, Lab 2
TA: Logan Kuo

Project 5 Report

Introduction

The purpose of this lab is to model a finite state machine (FSM) by designing a parking meter that counts down the amount of time remaining, allows users to add more time or reset the time, and displays the time remaining using a 7 segment display and 4 digit binary coded decimal (BCD) display. Through this lab, we will learn about FSMs, how to design one in the Xilinx ISE using Verilog, and how to write test bench code to test one in Isim. We will also learn how to work with the 7 segment display on the Nexsys3 Spartan-6 FPGA board and provide the appropriate signals to it so that particular digits are displayed. FSMs are important because they can be used to model a wide variety of real world devices, and 7 segment displays are commonly used in real life to display human readable digits.

The parking meter should start off with no time in it and be capable of having a remaining time of up to 9999 seconds. It should display the time remaining on a 4 digit 7 segment display and 4 BCD digits. Users should be able to reset the time left to 0, 15 or 150 seconds, as well as add time in increments of 60, 120, 180, and 300 seconds. The time cannot be exceed 9999 s, so if adding time would cause it to exceed this value, the time left should simply be set to 9999 s. Once the meter has reached 0 s left, it stays at that value until its time is changed to a nonzero value. When the time left is greater than or equal to 180 s or equal to 0 s, the 7 segment display should flash the time left at 1 Hz with a 50% duty cycle. Otherwise, if the time left is between 1 and 179 s, inclusive, the 7 segment display should flash at a frequency of 0.5 Hz with a 50% duty cycle, only displaying the even values. The 4 BCD outputs correspond to the digits of the time left, and they will always display the current value (unlike the 7 segment display). All 4 digits will always be displayed on the 7 segment display and 4 digit BCD outputs, so leading zeroes are added as needed. The system clock frequency is 100 Hz.

Design

Parking Meter Module (`parking_meter`, Top Level Module)

The parking meter was implemented in a top level module called `parking_meter` whose function is to implement the behavior given in the introduction with the inputs shown in Table 1 on the next page, and then output the correct outputs shown in Table 2 on the next page. This means taking in a variety of reset and add time inputs and then displaying the remaining time on a 7 segment display and 4 BCD output digits. It should also adhere to the state machine shown in Figure 1. To achieve this, it includes a variety of submodules which handle the add and reset time inputs, keep track of the time remaining, determine when to display the time (as the 7 segment display alternates between on and off), convert the internal time counter to 4 BCD digits, convert the BCD digits into their 7 segment representations, and handle the logic for displaying the 7 segment representation, as one must cycle through the 4 digits rapidly. Each module described in more detail after Figure 2, which shows the RTL schematic for this top level module, which is simply made up of its submodules.

Input	Description
clk	System clock signal; frequency of 100 Hz
rst	Signal to reset the parking meter; when high, overrides all other signals and resets time left to 0 s
rst1	Signal to reset parking meter; when high, resets time left to 15 s
rst2	Signal to reset parking meter; when high, resets time left to 150 s
add1	Signal to add time to parking meter; when high, adds 60 s to time remaining
add2	Signal to add time to parking meter; when high, adds 120 s to time remaining
add3	Signal to add time to the parking meter; when high, adds 180 s to time remaining
add4	Signal to add time to the parking meter; when high, adds 300 s to time remaining

Table 1: Inputs to the parking meter and their effect on the time remaining when high. These inputs are all synchronous, so they only take effect on the rising edge of the clock signal.

Note: If multiple input signals are high at the same time of a positive clock edge, the hierarchy of signals is rst > rst2 > rst1 > add4 > add3 > add2 > add1 > clk. For example, this means that if rst1 and add2 both are high, then when a positive clock edge occurs, rst1 will take precedence over add2 and add2 will have no effect. The timer will be set to 15 s rather than having 120 s added to it.

Output	Description
val1[3:0]	Ones digit of time remaining, BCD
val2[3:0]	Tens digit of time remaining, BCD
val3[3:0]	Hundreds digit of time remaining, BCD
val4[3:0]	Thousands digit of time remaining, BCD
led_seg[6:0]	7 segment cathode signals, where CA is the most significant bit and CG is the least significant bit; if a bit is low or 0, means turned on
a1	Anode for ones digit of 7 segment display; 1'b0 means turned on
a2	Anode for tens digit of 7 segment display; 1'b0 means turned on
a3	Anode for hundreds digit of 7 segment display; 1'b0 means turned on
a4	Anode for thousands digit of 7 segment display; 1'b0 means turned on

Table 2: Outputs of the parking meter. The time remaining is displayed in both BCD and 7 segment formats.

From these tables of inputs and outputs, along with the design specification, a Moore FSM was created to model the behavior of the parking meter. A Moore FSM was chosen because the clock has a frequency of 100 Hz, so an input will only have a maximum delay of 10 ms, which is negligible compared to the 1 second resolution of the display. Also, it helps to ensure output stability. Note that state transitions and inputs taking effect only occur on positive edges of the clock, which means that if a button is pressed for an extremely short amount of time (< 10 ms) between positive edges of the clock, it will not have any effect (this is tested later). Because a Moore FSM is used, the 7 segment outputs led_seg, a1, a2, a3, and a4 will update 1 clock cycle later, which will be seen in the simulation results. The state machine is shown on the next page.

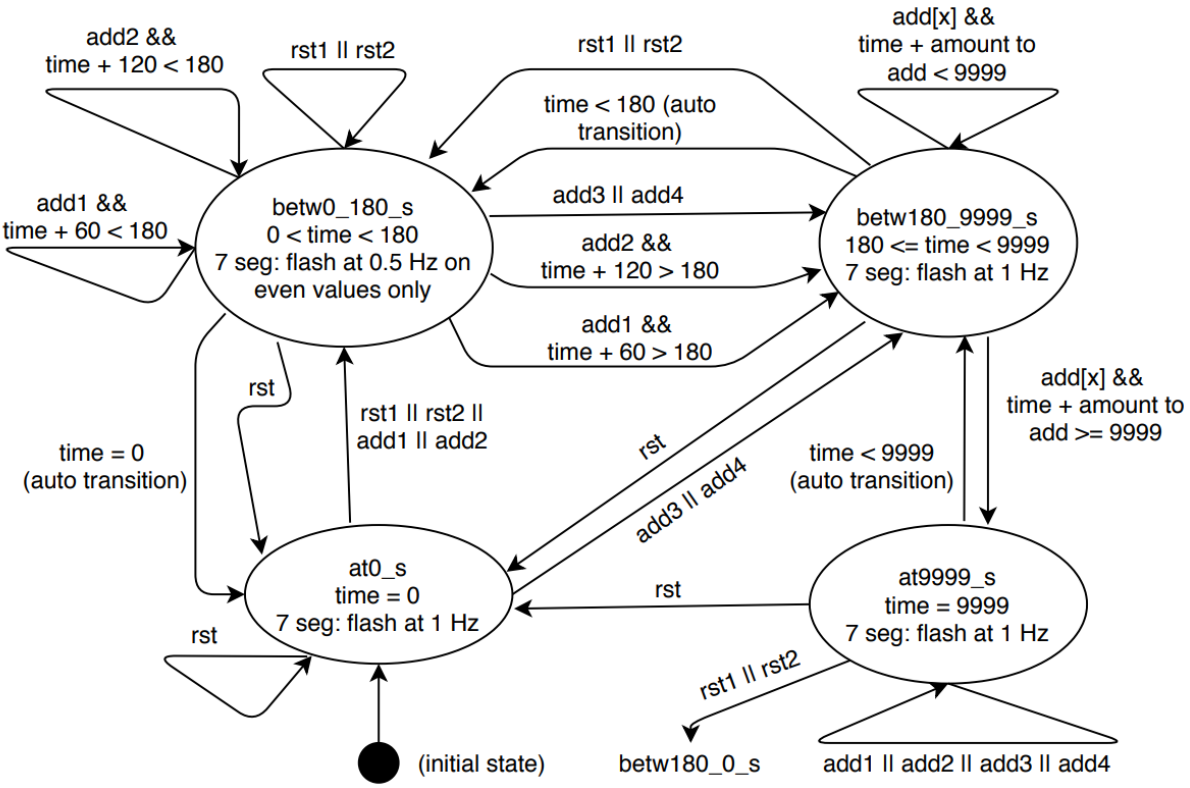


Figure 1: Moore FSM for the parking meter. All values for time are in seconds and `add[x]` means any one of `add1`, `add2`, `add3`, or `add4`. The auto transitions are for when the time left satisfies that constraint. The 7 segment output is always displayed with a 50% duty cycle and `val1`, `val2`, `val3`, and `val4` are always on and correspond to the amount of time left, so these outputs are not listed on the state machine. The state machine starts in the `at0_s` state, and any attempt to add more time when it is in the `at9999_s` simply causes it to stay in that state, as the maximum amount of time has been reached. Each state is described in detail in Table 3 below.

State	Description
`at0_s`	• Initial state of the machine • No time left in meter, time left is 0 s • 7 segment display flashes “0000” at 1 Hz with 50% duty cycle • `val1`, `val2`, `val3`, and `val4` are all 4’d0 • Entered upon `rst` being pressed or time running out (time = 0 s) • Exit to `betw0_180_s` or `betw180_9999_s` when time is not 0 s
`betw0_180_s`	• When time left is between 0 s and 180 s, exclusive • 7 segment display flashes time left at 0.5 Hz with 50% duty cycle on even amounts of time left only • `val1`, `val2`, `val3`, and `val4` correspond to the digits of the time left • Entered upon time left being between 0 s and 180 s, exclusive • Exited when time left becomes 0 s or greater than or equal to 180 s due to elapsing of time or certain inputs becoming high

(table continued on next page)

(table continued from previous page)

State	Description
`betw180_9999_s`	• When time left is between 180 s and 9999 s, inclusive of 180 s, exclusive of 9999 s • 7 segment display flashes time left at 1 Hz with 50% duty cycle • `val1`, `val2`, `val3`, and `val4` correspond to the digits of the time left • Entered upon time left being between 180 s and 9999 s, inclusive of 180 s, exclusive of 9999 s • Exited when time left becomes less than 180 s or equal to 9999 s due to elapsing of time or certain inputs becoming high
`at9999_s`	• When time left is 9999 s • 7 segment display flashes "9999" at 1 Hz with 50% duty cycle • `val1`, `val2`, `val3`, and `val4` all are 4'd9 • Entered upon time remaining being equal to 9999 s, or attempting to increase time remaining to a value beyond 9999 s • Exit to `betw180_9999_s` when time is no longer 9999 s, or `at0_s` when `rst` is high

Table 3: Description of each state from Figure 1 and its inputs and transitions.

Using the state machine and Tables 1 and 2, the parking meter was broken down into multiple submodules that collectively implemented the desired behavior. First, the inputs `clk`, `rst`, `add1`, `add2`, `add3`, `add4`, `rst1`, and `rst2` are passed into `timer` module, a sequential module that contains the logic for handling adding more time or resetting the time, an internal counter to keep track of the time remaining, a counter that divides the input clock by 100 to decrement the time left once per second, and logic to determine when the 7 segment display should be on. It has 2 outputs: a 14 bit output `counter` that represents the time left and a 1 bit output `shouldDisplay` that determines whether the 7 segment display should be on.

Then, the `bcd_vals` combinational module takes in the 14 bit output `counter` from `timer` as an input and uses division to output the ones, tens, hundreds, and thousands digits of the amount of time left as 4 4 bit BCD outputs `val1`, `val2`, `val3`, and `val4` respectively. These outputs are passed on as outputs of the top level module and to the `bcd_to_7seg` module.

There are then 4 `bcd_to_7seg` combinational modules that take in a 4 bit BCD input `val` from the `bcd_vals` module and convert it to corresponding 7 bit value called `led_seg` for the 7 segment display, which is the output of this module.

Finally, the `seven_seg` sequential module takes in the `clk` and `rst` inputs from the top level module, as well as a 1 bit input `shouldDisplay` from the `timer` module and 4 7 bit inputs `thousands`, `hundreds`, `tens`, and `ones` from the 4 `bcd_to_7seg` modules that represent the values to display on the 7 segment displays. It then outputs a 7 bit value `led_seg` for the cathodes of the 7 segment display, as well as 4 1 bit values `a1`, `a2`, `a3`, and `a4` for the anodes of the 7 segment display that correspond to the ones, tens, hundreds, and thousands digits. Only one of `a1`, `a2`, `a3`, and `a4` will ever be low (representing on) at any given instance. The outputs `led_seg`, `a1`, `a2`, `a3`, and `a4` are passed on to the top level module as outputs for the top level module.

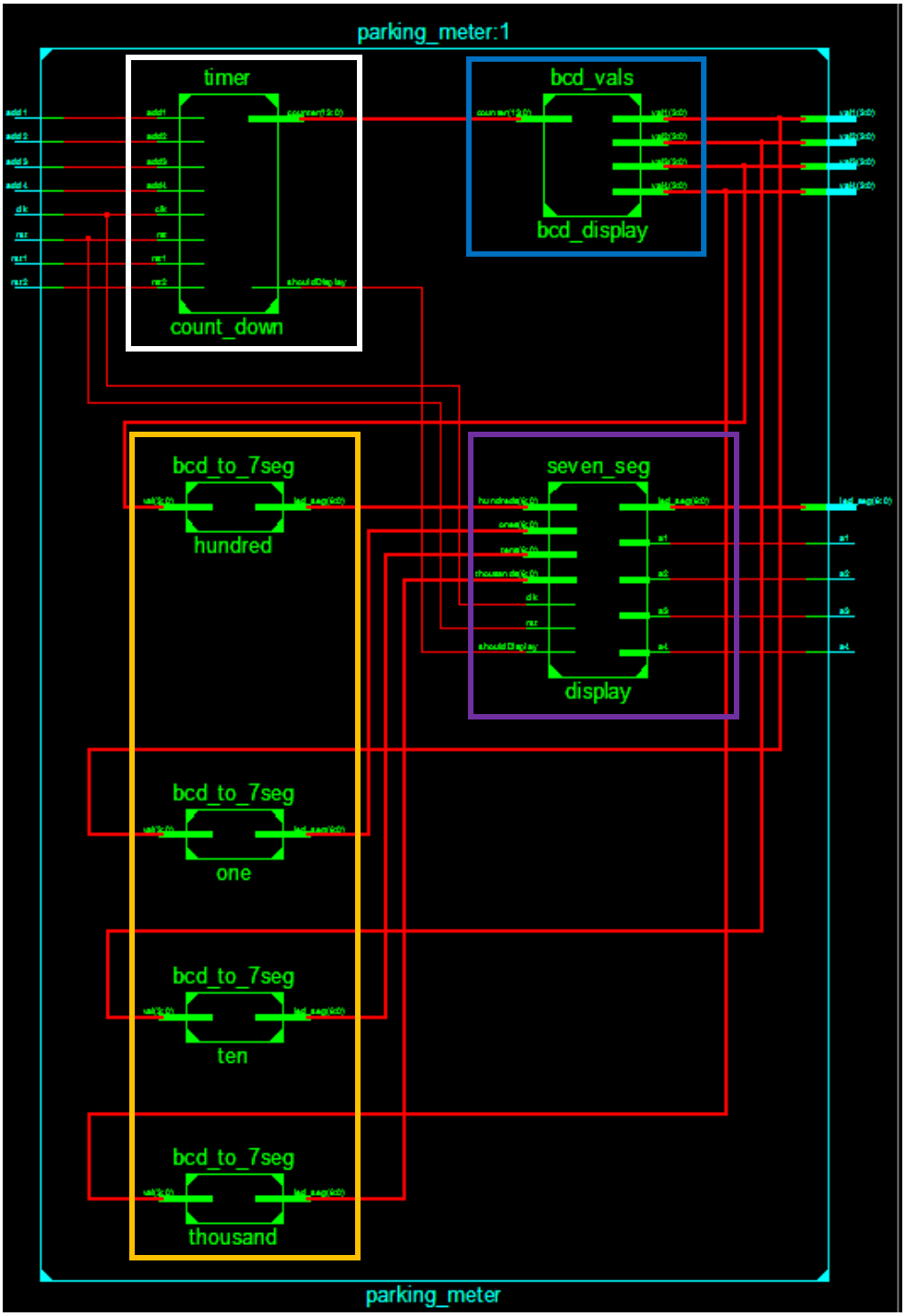


Figure 2: RTL schematic of the `parking_meter` module after synthesis. It is comprised of a `timer` module called `count_down` (white) that keeps track of the time remaining, a `bcd_vals` module called `bcd_display` (blue) for converting the time left into 4 BCD values that are outputted, and then 4 `bcd_to_7seg` modules (orange) to convert these BCD values to 7 segment encodings for the ones, tens, hundreds, and thousands digits. Finally, a `seven_seg` module called `display` (violet) takes in these 4 7 segment encodings and combines them together to output the top level module's cathode and anode signals for the 7 segment display.

The RTL schematic shown on the previous page was created from the following Verilog code for the top level `parking_meter` module, which takes in the inputs listed in Table 1, creates the previously mentioned submodules and passes the top level inputs to these submodules as needed, declares various wires to pass output signals from one submodule as input signals to another module, and then collects the outputs from the `bcd_vals` and `seven_seg` modules to outputs them as top level module outputs in accordance with Table 2.

```
----------------------parking_meter Module Code----------------------
wire shouldDisplay;
wire [13:0] counter;
wire [6:0] thousands;
wire [6:0] hundreds;
wire [6:0] tens;
wire [6:0] ones;

timer count_down(.clk(clk), .add1(add1), .add2(add2), .add3(add3),
                 .add4(add4), .rst1(rst1), .rst2(rst2), .rst(rst),
                 .counter(counter), .shouldDisplay(shouldDisplay));
bcd_vals bcd_display(.counter(counter), .val1(val1), .val2(val2),
                     .val3(val3), .val4(val4));
bcd_to_7seg thousand(.val(val4), .led_seg(thousands));
bcd_to_7seg hundred(.val(val3), .led_seg(hundreds));
bcd_to_7seg ten(.val(val2), .led_seg(tens));
bcd_to_7seg one(.val(val1), .led_seg(ones));
seven_seg display(.clk(clk), .rst(rst), .shouldDisplay(shouldDisplay),
                  .thousands(thousands), .hundreds(hundreds),
                  .tens(tens), .ones(ones), .led_seg(led_seg),
                  .a1(a1), .a2(a2),.a3(a3), .a4(a4));
---------------------------------------------------------------------
```

The following sections contain details about the function of each submodule and how it implements its desired functionality.

Timer Module (`timer`)

The function of this module is to handle inputs for adding or resetting the time left, determine how much time is left in the parking meter, decrement the time left every second, and determine when to display the time left on the 7 segment display. It does this by taking in the inputs system clock `clk`, global reset `rst`, additional inputs `rst1`, `rst2`, `add1`, `add2`, `add3`, and `add4`, and it outputs a 14 bit value `counter` representing the time left in the parking meter and a 1 bit value `shouldDisplay` that says if the 7 segment display should display the current value. Within this module, 2 7 bit internal registers `divide_1hz` and `one_second` are used as mod 100 counters to divide the input clock by 100 and keep track of when 1 second has elapsed, respectively. Both of these registers start off with a value of 7'd0.

The Verilog code for this module is given on the next page. The module is comprised of an `always` block that triggers on the positive edge of the 100 Hz input clock `clk`, and when this occurs, it follows the signal hierarchy listed below Table 1 to determine which action to take. Thus, within the `always` block, it first checks if `rst` is active, and then if `rst2` is active, and then if `rst1` is active, and then `add4`, so on, until the last else statement, which is if none of the inputs (except for `clk`) are high. If `rst` is high, then it sets `counter` to 14'd0 and sets the registers for keeping track of when to display and decrement the time left (`one_second` and `divide_1hz`) to 7'd0, as well as setting `shouldDisplay` to 1'b1 so that the time is displayed. This represents resetting the parking meter. Otherwise, if `rst1` or `rst2` is high, `counter` is set to the corresponding value (14'd15 or 14'd150) and uses this value to determine if `shouldDisplay` should be set to 1'b0 or 1'b1. If an `add<x>` input is high instead (and no `rst<x>` input is high), then the module checks if adding this number to the current time left would cause overflow past 9999 s. If so, it simply sets the time left to 14'd9999. Otherwise, it adds the appropriate amount of time. Then, it uses the updated time left to determine whether to display the time or not (`shouldDisplay` = 1'b1 or 1'b0, respectively) in accordance with the behavior given in the introduction. It checks the least significant bit of counter, `counter[0]`, to quickly determine if the time remaining is even or odd. Finally, if any of the `rst<x>` or `add<x>` inputs go high, `divide_1hz` and `one_second` are set to 7'b0, as we have just changed the value of the time left and thus should restarting the counting of elapsed clock cycles.

If none of the `rst<x>` or `add<x>` inputs are high, then the timer module does its default operation of using `one_second` to determine when to decrement the time left. The counter `one_second` is used to keep track of when to decrement the time left, so when it reaches 7'd99, if the time remaining is greater than 0, then it is decremented by 1 and `one_second` is set to 7'd0. Otherwise, `one_second` is simply incremented by 7'd1 every clock cycle.

The `divide_1hz` counter controls the logic that decides whether the 7 segment display should show a value or not. If the time left is 14'd0 or ≥ 14'd180, then `shouldDisplay` is set to 1'b1 when `divide_1hz` becomes 7'd0 and then set to 1'b0 when `divide_1hz` becomes 7'd50, thus creating a 50% duty cycle for the 7 segment display. `divide_1hz` is a mod 100 counter, so upon reaching 7'd99, it is reset to 7'd0 for the next clock cycle; otherwise, it is simply incremented by 1. If the time remaining is instead between 0 and 180 seconds, exclusive, then the 7 segment display should flash at 0.5 Hz with a 50% duty cycle, showing only even values. The counter `divide_1hz` moves in sync with the value of `one_second`, so if `divide_1hz` is 7'd99 and `counter` representing the time remaining is currently odd, then that means the time remaining will turn even on the next clock cycle, so `shouldDisplay` is set to 1'b1. Otherwise, if `divide_1hz` is 7'd99 and `counter` is currently even, then `shouldDisplay` is set to 1'b0 to reflect the fact that the time remaining is about to turn odd. If `divide1_hz` is not 7'd99, then the time is not about to change, and thus the value of `shouldDisplay` should not change and `divide1_hz` should simply be incremented by 7'd1. A special case is needed to handle displaying for the value of 14'd180, as the previous paragraph only covers the initial transition to the time left into 14'd180. Once we reach 180 seconds left and 50 clock cycles have elapsed, `shouldDisplay` is set to 1'b0, as 14'd180 should be displayed with a 1 Hz frequency with a 50% duty cycle.

----------timer Module Code----------

```
reg [6:0] divide_1hz = 7'd0;
//divide by 100 for flashing
reg [6:0] one_second = 7'd0;
//divide by 100 for decrementing

always @(posedge clk) begin
  if(rst) begin
    one_second <= 7'd0;
    divide_1hz <= 7'd0;
    counter <= 14'd0;
    shouldDisplay <= 1'b1;
  end
  else if(rst2 || rst1 || add4 ||
          add3 || add2 || add1)
  begin
    if(rst2) begin
      counter <= 14'd150;
      shouldDisplay <= 1'b1;
    end
    else if(rst1) begin
      counter <= 14'd15;
      shouldDisplay <= 1'b0;
    end
    else if(add4) begin
      shouldDisplay <= 1'b1;
      if(counter <= 14'd9699)
        counter <=
          counter + 14'd300;
      else
        counter <= 14'd9999;
    end
    else if(add3) begin
      shouldDisplay <= 1'b1;
      if(counter <= 14'd9819)
         counter <=
           counter + 14'd180;
      else
        counter <= 14'd9999;
    end
    else if(add2) begin
      if(counter <= 14'd9879)
        counter <= counter + 14'd120;
      else
        counter <= 14'd9999;
      // check if even or odd
      if(counter[0] == 1'b0 ||
         counter + 14'd120 >= 14'd180)
        shouldDisplay <= 1'b1;
      else
        shouldDisplay <= 1'b0;
    end
    else if(add1) begin
      if(counter <= 14'd9939)
        counter <= counter + 14'd60;
      else
        counter <= 14'd9999;
      if(counter[0] == 1'b0 ||
         counter + 14'd60 >= 14'd180)
        shouldDisplay <= 1'b1;
      else
        shouldDisplay <= 1'b0;
    end
    one_second <= 7'd0; //reset
    divide_1hz <= 7'd0; //counters
  end
  else begin //no inputs
    // decrement counter every 1 s
    if(one_second == 7'd99) begin
      one_second <= 7'd0;
      if(counter > 14'd0)
        counter <= counter - 14'd1;
    end
    else
      one_second <= one_second + 7'd1;
    end
```

----------timer module code continues on next page----------

```
----------------------timer Module Code (continued)--------------------
// always @(posedge clk) begin
// ... code on previous page...
// else begin //no buttons pressed
// code for determining when
// to display
    if(counter == 14'd0 ||
       counter > 14'd180) begin
      // display at 1 Hz
      if(divide_1hz == 7'd99) begin
        divide_1hz <= 7'd0;
        shouldDisplay <= 1'b1;
      end
      else begin
        divide_1hz <=
          divide_1hz + 7'd1;
        if(divide_1hz == 7'd49)
          shouldDisplay <= 1'b0;
      end
    end
    else begin
      // display at 0.5 Hz
      if(divide_1hz == 7'd99)
      begin
        divide_1hz <= 7'd0;
        if(counter[0] == 1'b1)
          shouldDisplay <= 1'b1;
        else
          shouldDisplay <= 1'b0;
        end
      else begin
       divide_1hz <= divide_1hz + 7'd1;
        if(divide_1hz == 7'd49 &&
           counter == 14'd180)
          shouldDisplay <= 1'b0;
      end
    end
  end // end else clk
end // end always @(posedge clk)
-----------------------------------------------------------------------
```

After synthesis, a RTL schematic was produced, but due its large size and complexity, only parts of it will be displayed here. The full schematic is given in Appendix A.

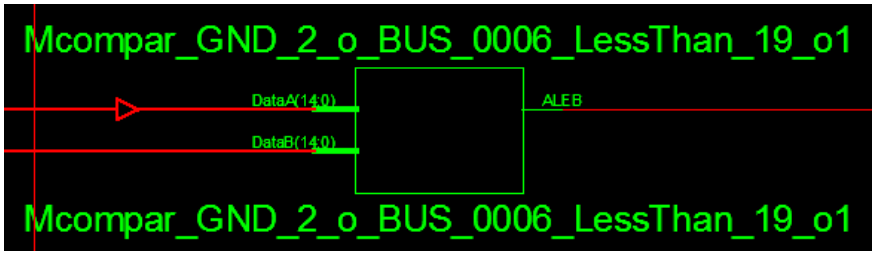


Figure 3: A comparator that compares to see if `counter` is less than 14'd180. This comparator arises from the many comparisons that are made between `counter` and constant values in the Verilog code shown above, and various other comparators exist for the other constant values that are compared against (such as 14'd9699, 14'd9819, 14'd9879, etc.)

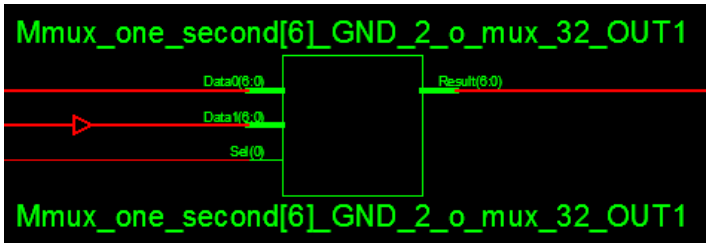


Figure 4: 7 bit 2 to 1 multiplexer to determine the next value of `one_second`. Depending on the value of the selection bit, either the current value of `one_second` (which is incremented somewhere else) is passed through, or 7'd0.

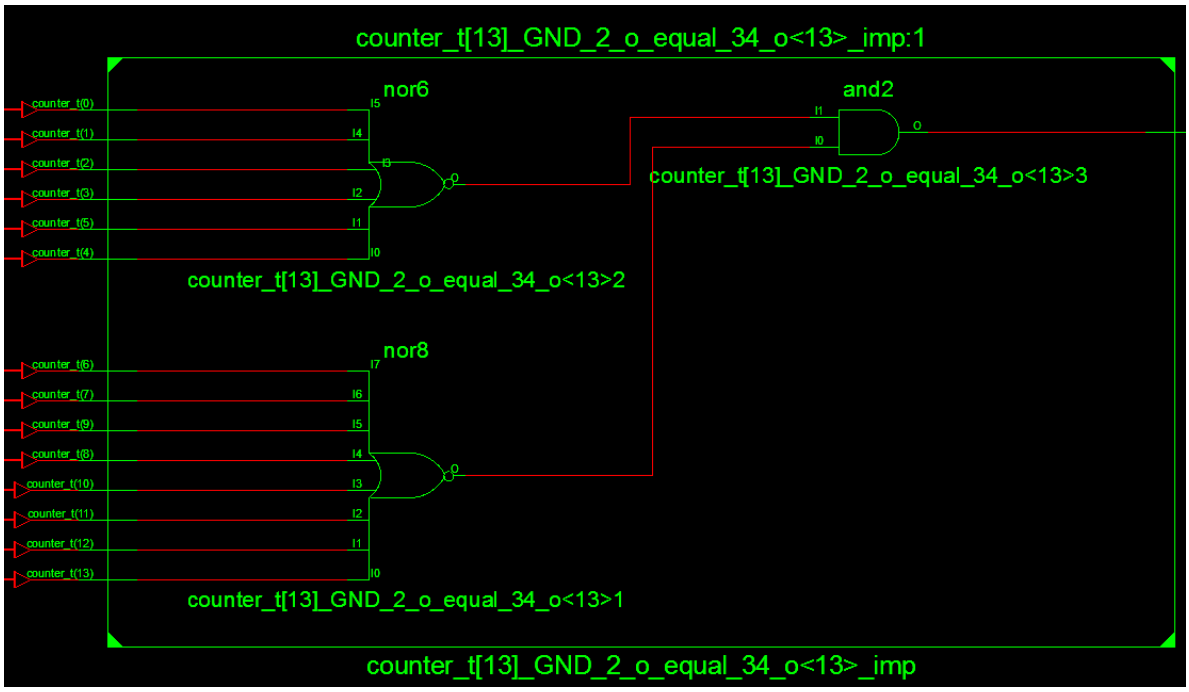


Figure 5: This combinational circuit puts every bit of `counter` through a NOR gate, thus checking if `counter` is equal to 14'd0. The value 14'd0 is used to determine what frequency to display the remaining time at and whether there is time that remains to be decremented.

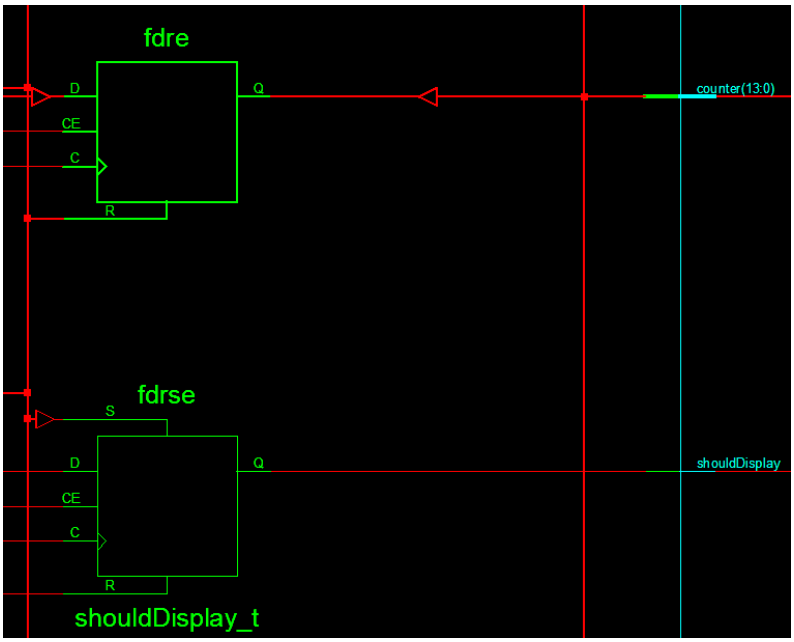


Figure 6: 1 bit register to store the current values of `shouldDisplay` (bottom) and 14 bit register to store the current value of `counter`. These are the outputs of the `timer` module (the blue text on the right).

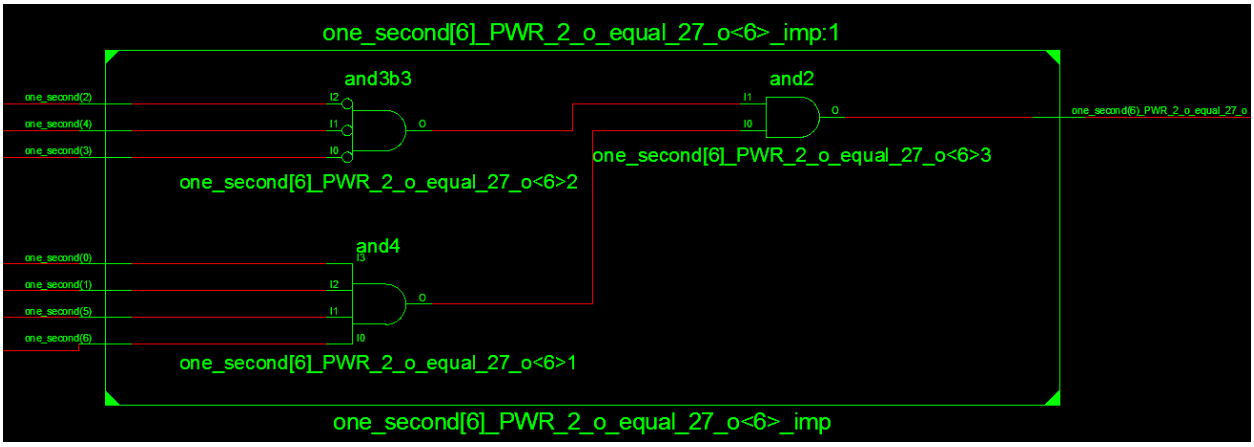


Figure 7: Gates to determine if `one_second` is equal to 7'd99 (7'b1100011). The output of this set of gates is used to determine when to set `one_second` back to 7'd0, as it is a mod 100 counter. Other sets of similar gates exist to determine if `one_second` or `divide_1hz` are equal to 7'd49.

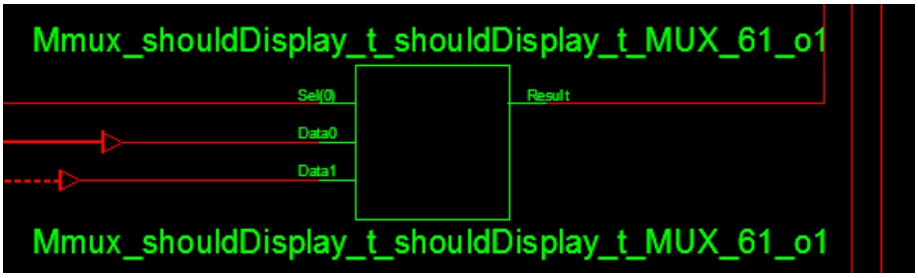


Figure 8: 2 to 1 1 bit multiplexer used to pick between 2 values for `shouldDisplay`. The selection bit comes from `counter`, and either 1'b0 or 1'b1 is outputted from this multiplexer.

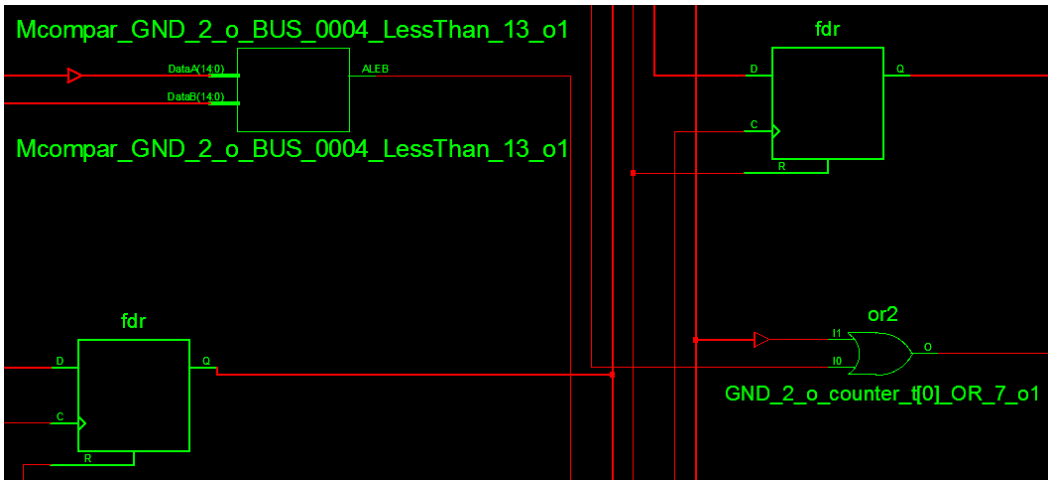


Figure 9: 7 bit registers to hold the current values of `divide_1hz` (lower left) and `one_second` (top right). There is also a comparator (top left) to compare if the value of `counter` is less than 14'd180, similar to the one in Figure 3.

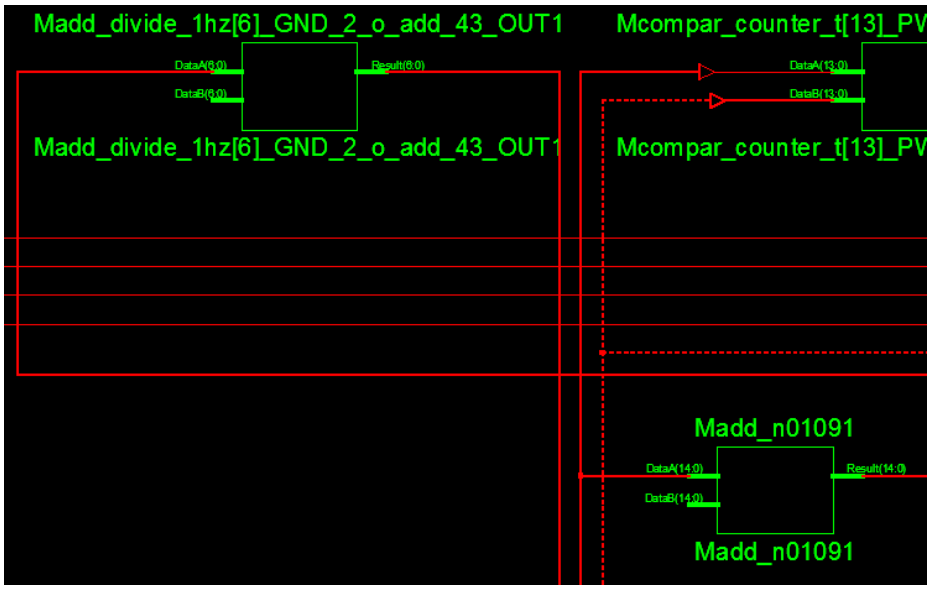


Figure 10: Adders used to increment the values of `divide_1hz` (white) and `counter` (blue). DataA contains the current value and DataB is the constant that is added.

BCD Values Module (`bcd_vals`)

The function of this module is to convert the current time remaining in the parking meter to 4 BCD values, one for each digit of the time remaining. It takes in a 14 bit input `counter` from the `timer` module and then outputs 4 4 bit BCD values `val1`, `val2`, `val3`, and `val4` representing the ones, tens, hundreds, and thousands digits of the time remaining, respectively. As shown in the Verilog code below, this module is simply a combinational circuit that uses division to extract each digit of the `counter` input. Because a 14 bit input is being divided, 4 14 bit registers are created to temporarily store the values of the division, and then the BCD outputs are assigned to the lowest 4 bits. This is so that Xilinx does not report a warning about a 14 bit value being truncated to fit in a 4 bit target; however, truncation will not actually occur, as `counter` is no more than 9999 and it is divided by 1000, so it will definitely fit in a 4 bit target.

```
------------------------bcd_vals Module Code------------------------
reg [31:0] val1_14; //ones
reg [31:0] val2_14; //tens
reg [31:0] val3_14; //hundreds
reg [31:0] val4_14; //thousands

assign val1 = val1_14[3:0];
assign val2 = val2_14[3:0];
assign val3 = val3_14[3:0];
assign val4 = val4_14[3:0];
```

```
always @(*) begin
  val4_14 = counter / 10'd1000;
  val3_14 = (counter -
             10'd1000*val4_14) / 7'd100;
  val2_14 = (counter -
             (10'd1000*val4_14 +
              7'd100*val3_14)) / 4'd10;
  val1_14 = (counter -
             (10'd1000*val4_14 +
              7'd100*val3_14 +
              4'd10*val2_14));
end
```

```
-------------------------------------------------------------------
```

After the module was synthesized, the resulting RTL schematic is given on the next page.

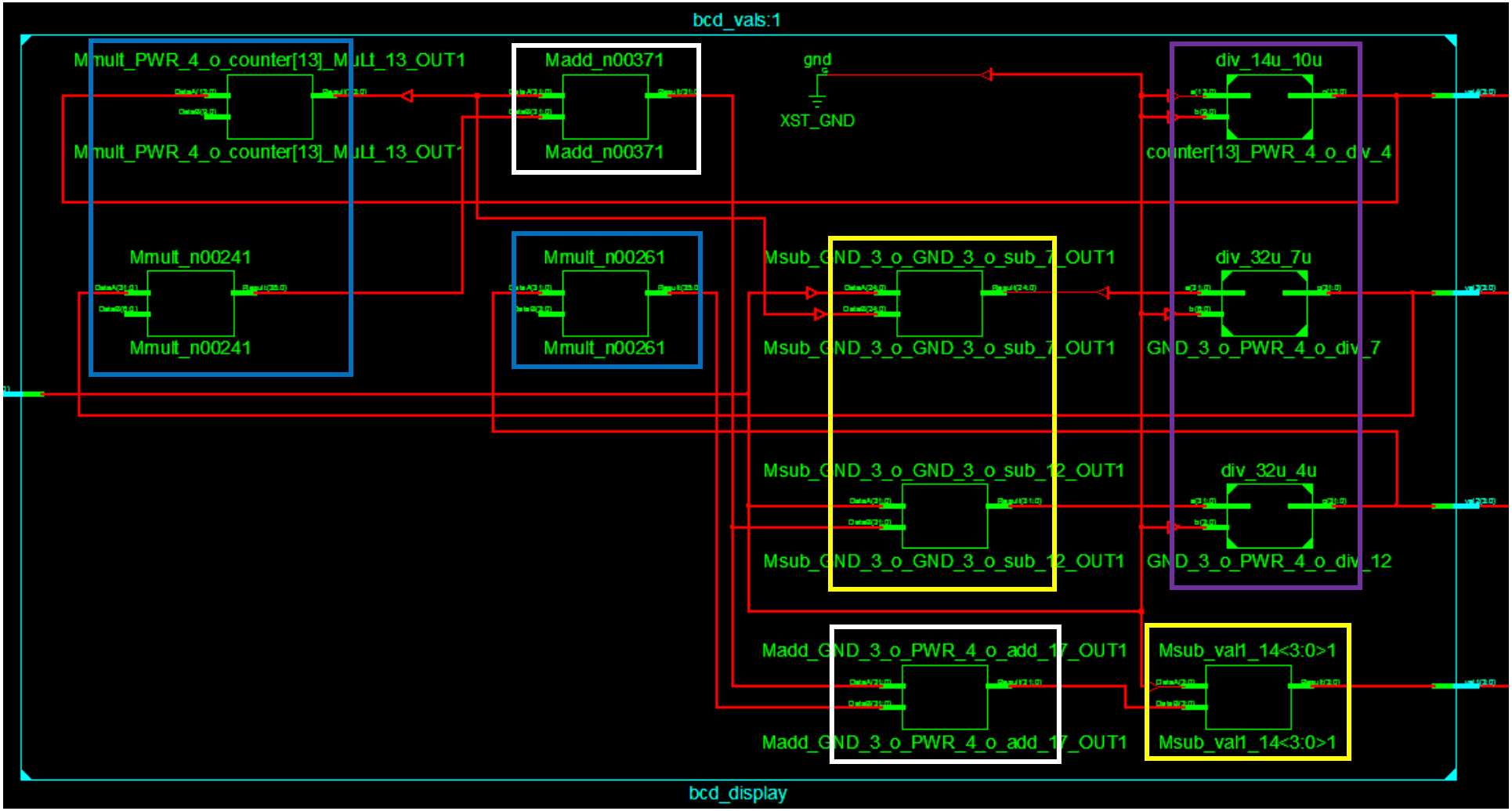


Figure 11: RTL schematic generated after synthesis for the `bcd_vals` module. It takes in a 14 bit input `counter` on the left side, and passes it through a series of multipliers (blue), adders (white), subtractors (yellow), and dividers (violet) to get each digit of the input, and it outputs these digits on the right as (from top to bottom) `val4`, `val3`, `val2`, and `val1`, which correspond to the thousands, hundreds, tens, and ones digits respectively. All 4 basic arithmetic operations are present in the Verilog code for this module, and thus they show up in the synthesis as well. The divider units (violet) are actually encapsulating combinational logic within them that is too complex to show here.

BCD to 7 Segment Display Module (`bcd_to_7seg`)

The function of this module is to convert a 4 bit BCD value into its corresponding 7 bit value for the 7 segment display. Thus, this was implemented using a simple combinational circuit that takes in a 4 bit BCD value `val` and outputs `val`'s corresponding 7 bit value `led_seg` that has the appropriate bits set and unset so that it shows up as the correct number on the 7 segment display. This was achieved in Verilog by using a case statement on `val`, defaulting to 7'b1111111 when `val` is not a digit between 0-9 (inclusive), which corresponds to all the LED segments being off. To figure out the corresponding values for the 7 segment display, I consulted the user manual for the Nexsys3 board; the relevant image is shown below. For the 7 segment display output `led_seg`, a bit being 0 (low) means that the segment is lit up, and `CA` is the most significant bit, while `CG` is the most significant bit. The Verilog code that implemented this module is shown on the next page. The anodes will be used in the `seven_seg` module (see next section). 4 of these modules are used, one for each digit of the time remaining (thousands, hundreds, tens, ones).

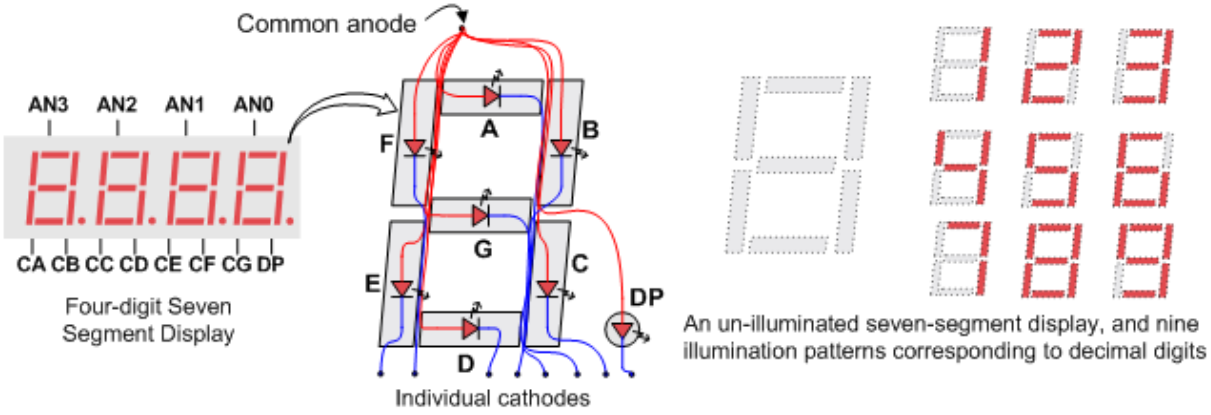


Figure 12: Signal diagram for the 7 segment display on the Nexsys3 FPGA board. There are 7 cathode signals `A`, `B`, `C`, `D`, `E`, `F`, and `G`, as well as a decimal point signal (`DP`), which is not used. Each cathode signal is preceded with a "`C`." Figure courtesy of Nexsys3 user manual[1].

Decimal Digit	7 Segment Representation
0	7'b0000001
1	7'b1001111
2	7'b0010010
3	7'b0000110
4	7'b1001100
5	7'b0100100
6	7'b0100000
7	7'b0001111
8	7'b0000000
9	7'b0000100

Table 4: Decimal digits 0-9 and their corresponding values for the 7 segment display.

[1] https://reference.digilentinc.com/reference/programmable-logic/nexys-3/reference-manual

```
------------------------bcd_to_7seg Module Code-----------------------
always @(*) begin
  case(val)
    4'd0: led_seg = 7'b0000001;
    4'd1: led_seg = 7'b1001111;
    4'd2: led_seg = 7'b0010010;
    4'd3: led_seg = 7'b0000110;
    4'd4: led_seg = 7'b1001100;
    4'd5: led_seg = 7'b0100100;
    4'd6: led_seg = 7'b0100000;
    4'd7: led_seg = 7'b0001111;
    4'd8: led_seg = 7'b0000000;
    4'd9: led_seg = 7'b0000100;
    default: led_seg = 7'b1111111;
  endcase
end
-----------------------------------------------------------------------
```

After synthesis, the following RTL schematic was generated.

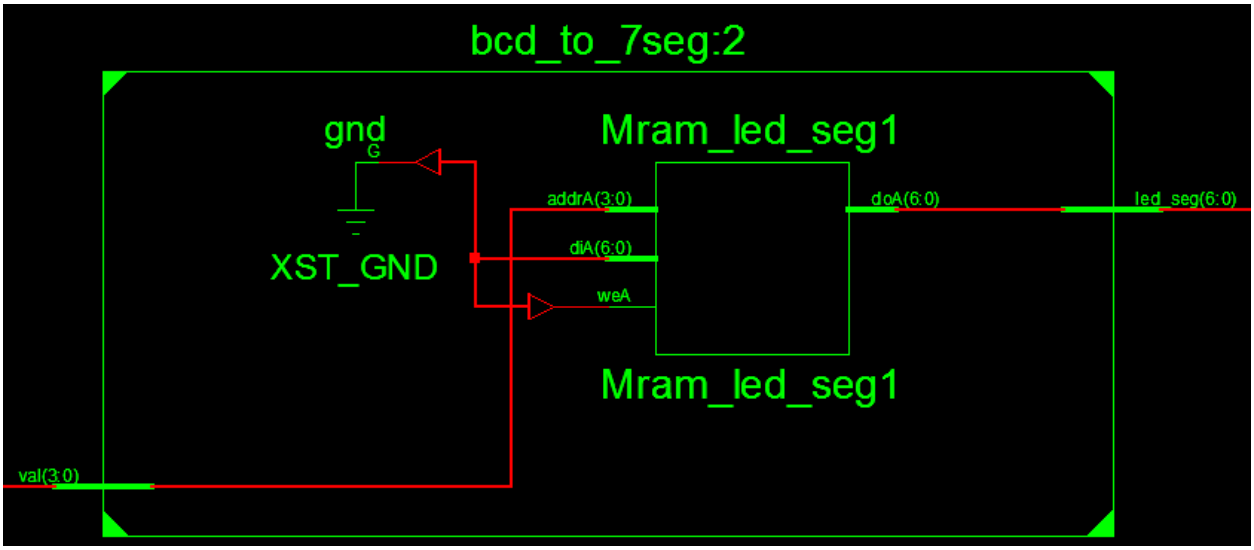


Figure 13: RTL schematic of `bed_to_7seg` module after synthesis. Instead of a multiplexer, a random access memory block was chosen instead, and examining the synthesis report shows that this is a 16x7-bit single port read only RAM, which makes sense, as the BCD input `val` is 4 bits wide for up to 16 selection options, and all `led_seg` outputs are 7 bits. A RAM block was most likely used because there is a static mapping of BCD values to 7 segment display cathode values that is the same across all 4 `bcd_to_7seg` modules.

Seven Segment Display Module (`seven_seg`)

The function of this module is to provide the anode and cathode signals for the 4 digit 7 segment display on the FPGA board. To do this, it takes in a clock signal `clk`, reset signal `rst`, a `shouldDisplay` signal from the `timer` module to know whether it should display something or not, 4 7 bit 7 segment display values `thousands`, `hundreds`, `tens`, and ones from the bcd_to_7seg modules, and outputs a 7 bit output `led_seg` for the cathode signal of the 7 segment display, as well as 4 1 bit signals `a1`, `a2`, `a3`, and `a4`, which are anode signals used to control the ones, tens, hundreds, and thousands digits of the 7 segment display, respectively.

The 7 segment display is driven by cathode (covered in the previous section) and anode signals. The cathode signals determine which segment of the 7 segment display lights up according to Table 4, while the anode signal determines which digit will to light up. For both the cathodes and nodes, a low signal (or 0) asserts that the digit or segment should be on. Note that there is a minor difference in anode names between the figure and what is used here: `AN0` in the figure corresponds to `a1` for this project, and `AN1` corresponds to `a2`, and so on. For example, to display '1' on the leftmost digit, a4 would be 1'b0 and the cathodes would be 7'b1001111 (see Table 4). The reason for why one cannot simply use the `led_seg` outputs from the 4 `bcd_to_7seg` modules is that every segment in the same place for each digit is actually connected to the same cathode wire. Thus, displaying something like "1234" is not possible, as you can only have one 7 bit cathode signal at any time to determine what digit is displayed. It is possible to display "9999," as all digits are the same. Thus, we rapidly cycle through each digit, enabling its display with its corresponding anode and setting the corresponding 7 bit cathode signal for that digit. Each digit is displayed for 1 clock cycle before it is turned off and the next one is displayed, and to the human eye, with a fast enough refresh rate, it will look as if all 4 digits are on all the time. Because the input clock is only 100 Hz, the refresh rate for this project's 7 segment display is 40 ms, which is slower than what is recommend by the FPGA board reference manual, but this is only a simulation, so the 40 ms refresh rate is acceptable.

To implement this cycling through of digits in Verilog, an `always` block that triggered on the rising edge of the clock was used, along with an internal 2 bit counter `digToDisplay` that kept track of which digit to turn on. If `rst` is high, then all anodes are set to 0 and `led_seg` is set to 7'b0000001, which corresponds to "0000" being displayed. The variable `digToDisplay` is also reset to the thousands digit next. Otherwise, on the rising edge of the clock, if `shouldDisplay` is 1'b1, then the code cycles through the digit that should be displayed using a case statement on `digToDisplay`, setting the appropriate `a<x>` output to low and updating the corresponding `led_seg` output from the correct input digit (`thousands`, `hundreds`, `tens`, or `ones`). If `shouldDisplay` is not high, then all anodes are set to high and the cathode signal is set to all high bits so that the 7 segment display is turned off. The code to accomplish this is shown on the next page.

```
-------------------------seven_seg Module Code-------------------------
reg [3:0] anodes; //pack a1, a2, a3, a4 together into a 4 bit register
assign a1 = anodes[0]; //ones
assign a2 = anodes[1]; //tens
assign a3 = anodes[2]; //hundreds
assign a4 = anodes[3]; //thousands

reg [1:0] digToDisplay = 2'd0;

always @(posedge clk) begin
  if(rst) begin
    digToDisplay <= 2'd0;
    led_seg <= 7'b0000001;
    anodes <= 4'b0000; //display 0000 until rst button is unpressed
  end
  else begin
    if(shouldDisplay) begin
      case(digToDisplay)
        2'd0: begin
          anodes <= 4'b0111;
          led_seg <= thousands;
        end
        2'd1: begin
          anodes <= 4'b1011;
          led_seg <= hundreds;
        end
        2'd2: begin
          anodes <= 4'b1101;
          led_seg <= tens;
        end
        default: begin
          anodes <= 4'b1110;
          led_seg <= ones;
        end
      endcase
      digToDisplay <= digToDisplay + 2'd1;
    end
    else begin
      led_seg <= 7'b1111111; // all off
      anodes <= 4'b1111; //all off
      digToDisplay <= 2'd0;
    end
  end
end
-----------------------------------------------------------------------
```

The RTL schematic generated after synthesis is shown on the next page.

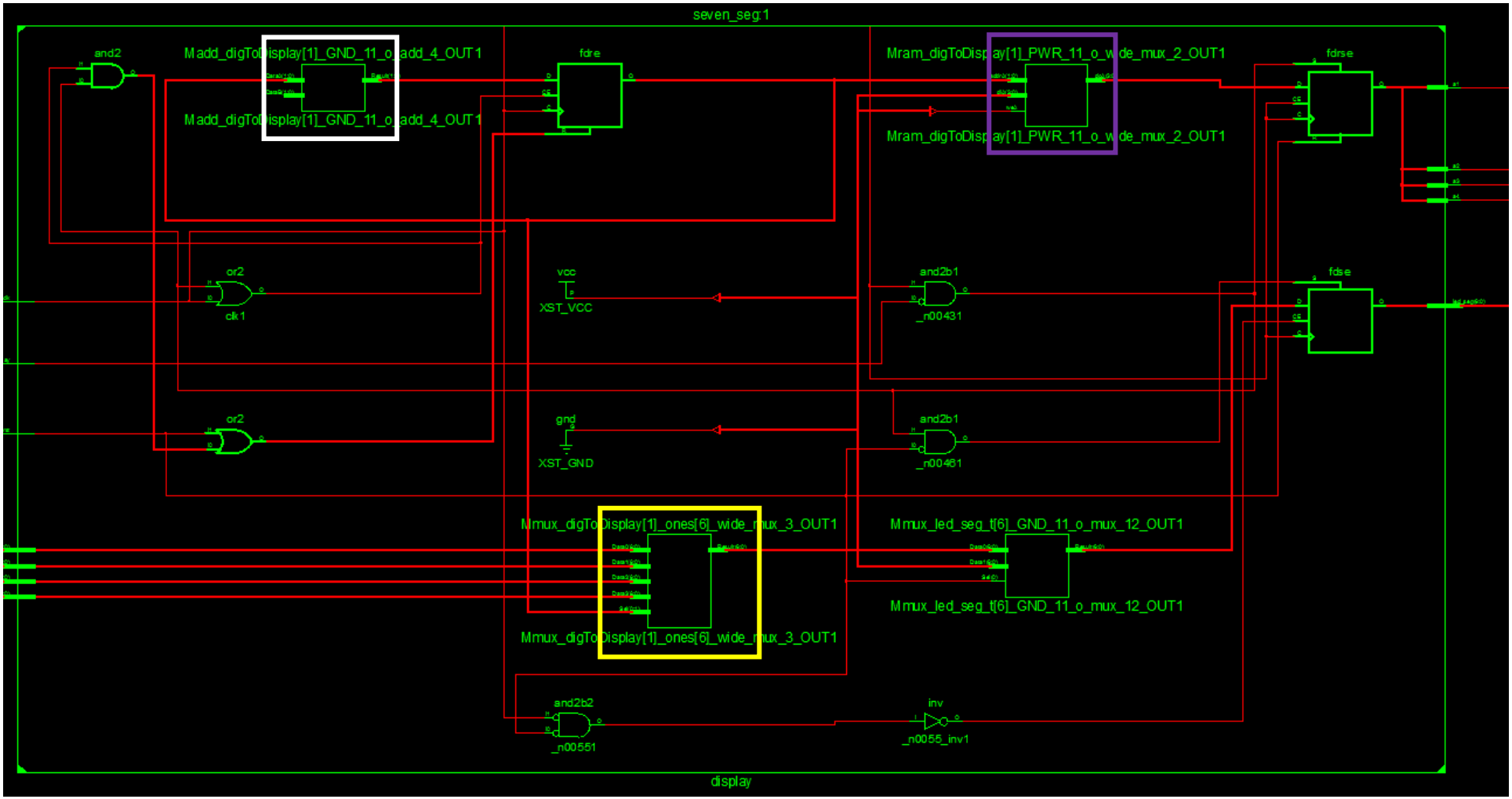


Figure 14: RTL schematic of `seven_seg` module after synthesis. It takes in 4 7 bit 7 segment display values from the `bcd_to_7seg` modules for the thousands, hundreds, tens, and ones digits, as well as `shouldDisplay` from the timer module and the system `rst` and `clk` inputs. It then outputs a single 7 bit 7 segment display value called `led_seg`, as well as 4 1 bit anode control signals `a1`, `a2`, `a3`, and `a4`. It uses an 2 bit adder (white) to cycle through picking the next digit to display, a 4 to 1 7 bit multiplexer to choose which 7 segment display input to display (yellow), a 4x4 bit read only RAM to get the anode signals based on which digit to display (violet), and various registers to hold the values of the 7 bit segment display bit vector, 4 bit anode bit vector, and 2 bit counter that keeps track of which digit will be displayed. Additional multiplexers and basic logic gates are used to handle the `rst` input being 1'b1 and `shouldDisplay` being 1'b0, in which case the anode and cathode signals are set to constant values.

Simulation Documentation

The requirement for the parking meter is that it follows the behavior specified in the introduction, which includes being capable of accepting input to reset or add time, counting downwards when there is no input, and correctly displaying the time remaining on the 7 segment display and BCD digit outputs. It should take in the inputs listed in Table 1 and provide the correct outputs with regards to both timing and value listed in Table 2. Although the parking meter is comprised of a variety of submodules, most of them are quite simple, so the unit was tested together as a whole with one single testbench for the top level module, and tests were done consecutively in the order below. The testbench code has a timescale of 1 ms (rather than 1 ns).

Test #	Test Focus	Description and Expected Behavior
1	Output behavior when time is expired/initial state	• BCD digits should all be 4'd0 • 7 segment display should be flashing "0000" at 1 Hz with 50% duty cycle • Also check general behavior of 7 segment display
2	Inputs	• Confirm `rst`, `rst1`, `rst2`, `add1`, `add2`, `add3`, and `add4` inputs work and update the time correctly
3	Output behavior when time remaining is ≥ 180 s	• BCD digits correctly display the time remaining • 7 segment display flashes the remaining time at 1 Hz with 50% duty cycle
4	Auto decrease in time values	• Remaining time should decrease every second when no inputs are provided • Tested in conjunction with tests 3, 5, 6
5	Output behavior when time remaining is > 0 s and < 180 s	• BCD digits correctly display the time remaining • 7 segment display flashes the remaining time at 0.5 Hz with 50% duty cycle on even values only
6	Output behavior during transition of time left from ≥ 180 s to < 180 s	• 7 segment display flashes 180 at 1 Hz with 50% duty cycle, then does not display 179, and displays 178 for a full second • Tested in conjunction with test #5
7	Latch to 14'd9999	• Keep one of the `add<x>` inputs high until the time remaining is 14'd9999 and then try to add more time • Time remaining should never exceed 14'd9999 • Time remaining should not go down when the `add<x>` input is high
8	Multiple simultaneous inputs	• If 2 inputs are both at the positive edge of the clock, the one with higher precedence takes effect
9	Run time down to 0	• Let the parking meter run down all the way to 0 s from a value ≥ 180 s to ensure it works correctly
10	Add/reset time so resulting time is odd, < 180 s	• The time left should be increased but not displayed on the 7 segment output
11	Extremely short input duration	• Turn an input on and then off within 2 rising edges of the clock signal; should have no effect

Table 5: List of tests performed on the `parking_meter` module in the testbench code.

Test 1: Initial State/Time Expired Behavior

The purpose of this test is to check that the parking meter starts out in the idle state, and then in this state, it flashes "0000" on the 7 segment display with a 1 Hz frequency and 50% duty cycle, and that the BCD outputs are all 4'd0. This test also checks that the 7 segment display is correctly cycling through each of the anodes. Finally, this test checks the reset input `rst`, ensuring that it overrides all other inputs. This is also the first test in the testbench, so this also servers to initialize the inputs. Note that for this testbench, the timescale is 1 ms, rather than 1 ns.

```
------------------------Test 1 Testbench Code-----------------------
initial begin
  clk = 1;
  add1 = 1;
  add2 = 1;
  add3 = 1;
  add4 = 1;
  rst1 = 1;
  rst2 = 1;
  rst = 1; //test rst, precedence
  // over all other inputs
  #10; // wait 1 clock cycle
  rst = 0;
  rst1 = 0;
  rst2 = 0;
  add1 = 0;
  add2 = 0;
  add3 = 0;
  add4 = 0;
  #4500; //9 seconds
  // test bench code for
  // other tests
end

// clock signal
always begin
  #5; //100 Hz clock
  clk = ~clk;
end
----------------------------------------------------------------------
```

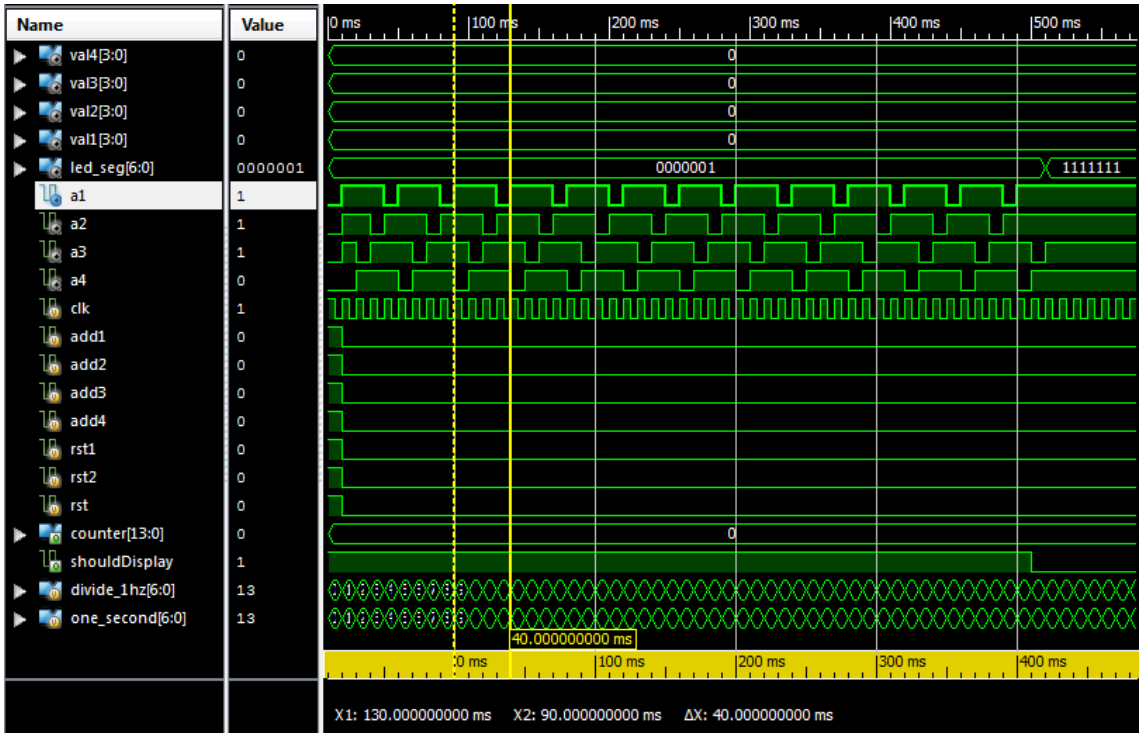


Figure 15: Simulation results for test 1. We can see anodes `a1`, `a2`, `a3`, and `a4` successively going low on each clock cycle, resulting in a 40 ms refresh rate. The `led_seg` value 7'b0000001 corresponds to the digit 0, and we are correctly displaying that no time is left.

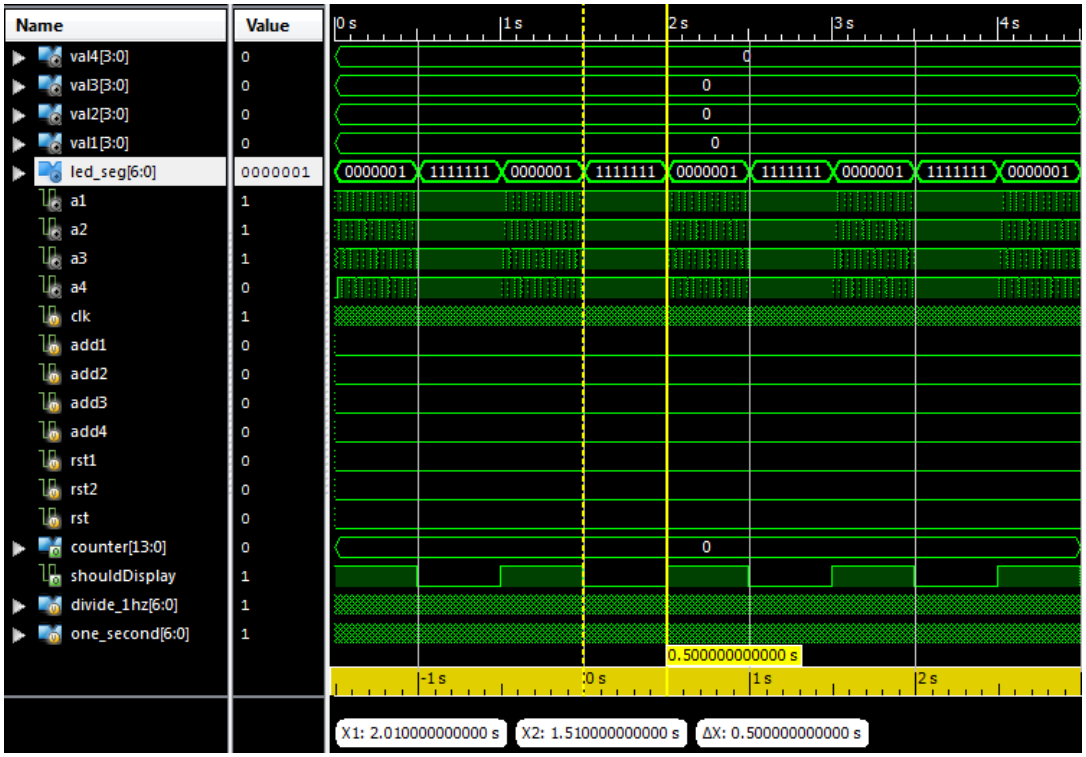


Figure 16: Simulation results for test 1. We can see that `led_seg` displays 0 for 0.5 s before turning off for 0.5 s (see time cursors), confirming the display is flashing at 1 Hz with a 50% duty cycle. The BCD outputs `val1`, `val2`, `val3`, and `val4` display the expected value of 0.

Test 2: Check Inputs Work

The purpose of this test is to check that the inputs `rst1`, `rst2`, `add1`, `add2`, `add3`, and `add4` all work. When each input goes high, the time remaining should be set or incremented by the correct amount, and the 7 segment display should start displaying the new time if it is greater than 180 s, or less than 180 s and even. Thus, the testbench code simply consists of turning these 6 inputs on and then off. The code to drive the clock and for other tests are not shown here.

```
------------------------Test 2 Testbench Code-----------------------
initial begin                          |   add2 = 1; //Test 2: test add2
  // ... Test 1 code ...               |   #10;
  rst2 = 1; //Test 2: test rst2        |   add2 = 0;
  #10;                                 |   #10;
  rst2 = 0;                            |   add3 = 1; //Test 2: test add3
  #10;                                 |   #10;
  rst1 = 1; //Test 2: test rst1        |   add3 = 0;
  #10;                                 |   #10;
  rst1 = 0;                            |   add4 = 1; //Test 2: test add4
  add1 = 1; //Test 2: test add1        |   #10;
  #10;                                 |   add4 = 0;
  add1 = 0;                            |   #10;
  #10;                                 |   // ...Test 3 code ...
-----------------------------------------------------------------------
```

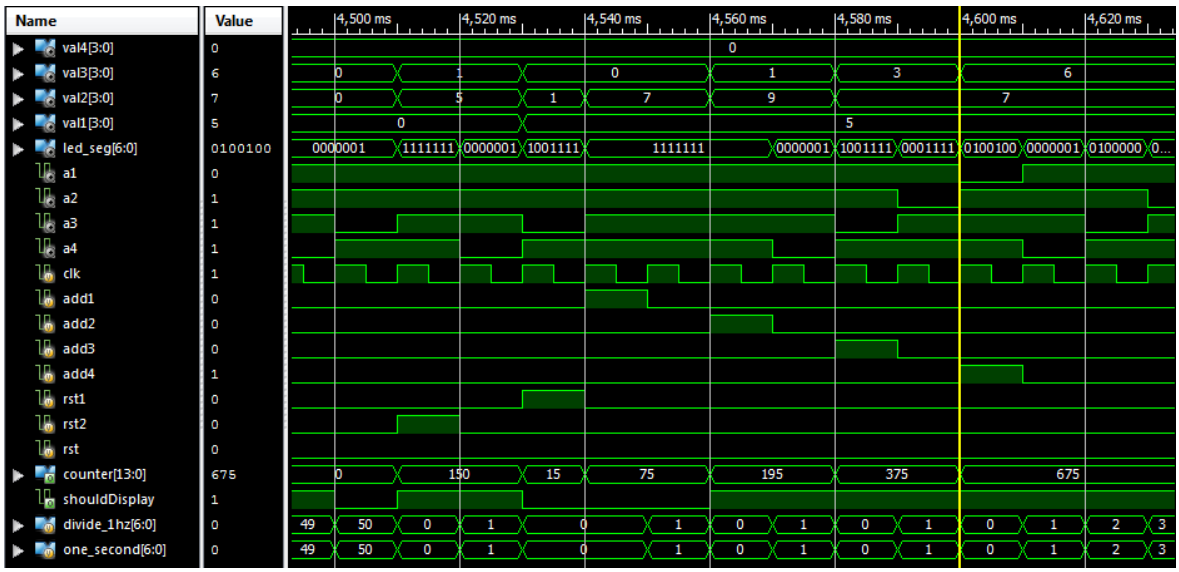


Figure 17: Simulation results for test 2. When `rst2` becomes high, `counter` is updated to 14'd150, and so are `val1`, `val2`, `val3`, and `val4`. The 7 segment display updates 1 cycle later, which is due to the fact that it gets the updated value of `counter` on the next clock cycle. `rst1` going high sets `counter` to 14'd15 as expected, and `add1`, `add2`, `add3`, and `add4` going high all update `counter` and the outputs correctly. The counters for decrementing the time (`one_second`) and turning off the display (`divide_1hz`) also reset to 0 every time an input goes high. Also note that nothing is displayed when `counter` is 14'd15 or 14'd75, as expected.

Test 3: Output Behavior, Time Remaining ≥ 180 s

The purpose of this test is to ensure that the outputs are correct when the time remaining is more than 180 s. The BCD outputs should display the current time left and the 7 segment display should flash the time left at 1 Hz with a 50% duty cycle. It also checks that the parking meter correctly counts down every second. Since test 2 finished with the counter set to 14'd675, the test code for this section simply consists of waiting for 10 seconds, so it is not shown here.

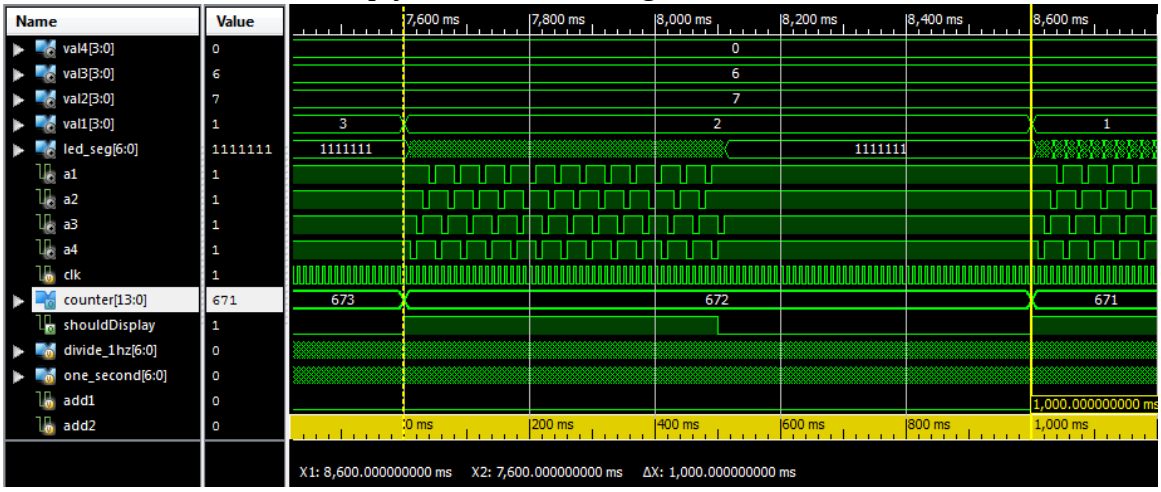


Figure 18: Simulation results for test 3. We see that the parking meter correctly spends 1 s with `counter` = 14'd672, and so does the BCD outputs (4'd0, 4'd6, 4'd7, 4'd2). The 7 segment display is on for 0.5 seconds and then off for 0.5 seconds, as expected.

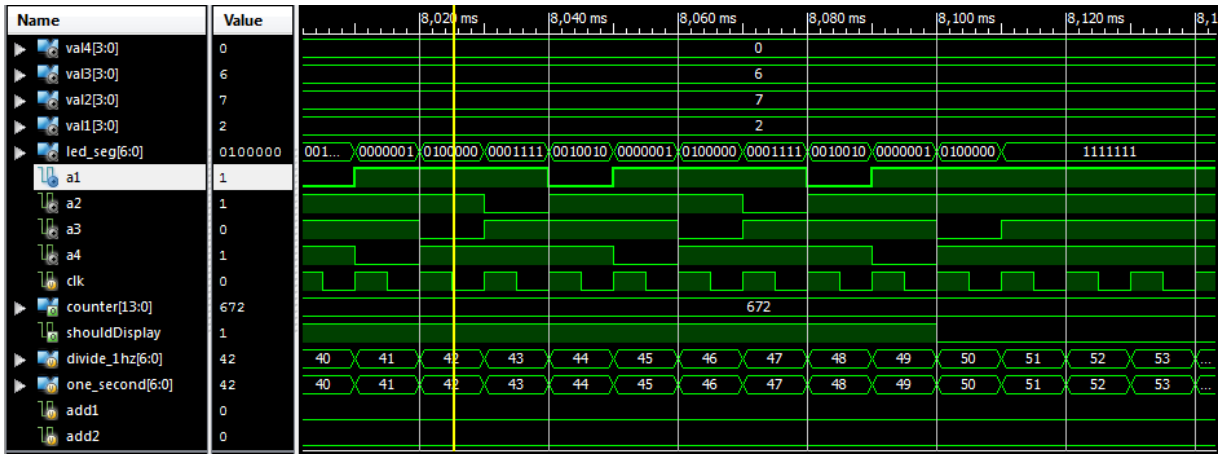


Figure 19: Simulation results for test 3. 7 segment display behavior for when the time remaining is ≥ 180s. The the 7 segment display correctly rapidly cycles through the values 0, 6, 7, and 2, with the values of the anodes matching the expected value of `led_seg` for when that particular anode is turned on (1'b0). When `divide_1hz` becomes 50, the 7 segment displays off on the next clock cycle, resulting in a 1 Hz frequency and 50% duty cycle.

Test 4: Auto Decrease in Time Values

The purpose of this test is to ensure that the internal timer decrements by 1 every second when there are no inputs. Figure 18 verifies this for when the time left is ≥ 180 s, and Figure 21 verifies this for when the time left is < 180 s.

Test 5: Output Behavior, Time Remaining > 0 s and < 180 s

The purpose of this test is to ensure the outputs are correct when the time remaining is less than 180 s and not 0 s. The BCD outputs should display the current time left and the 7 segment display should flash the time left at 0.5 Hz with a 50% duty cycle on even numbers. We use the `rst2` input to set the time remaining to 150 s, thus allowing us to test the output behavior in this time range. To do this is a couple lines in the testbench, so the code is not shown here.

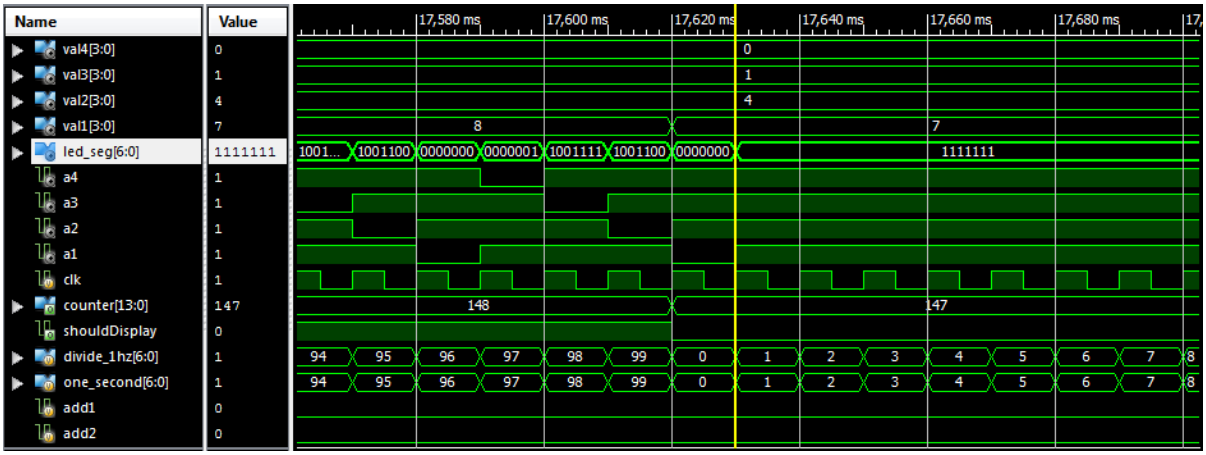


Figure 20: Simulation results for test 5. We see that the 7 segment display is on for when the time left is 14'd148, but then turns off when the time left is 14'd147. The digits that it displays, which are 0, 1, 4, and 7, are correct.

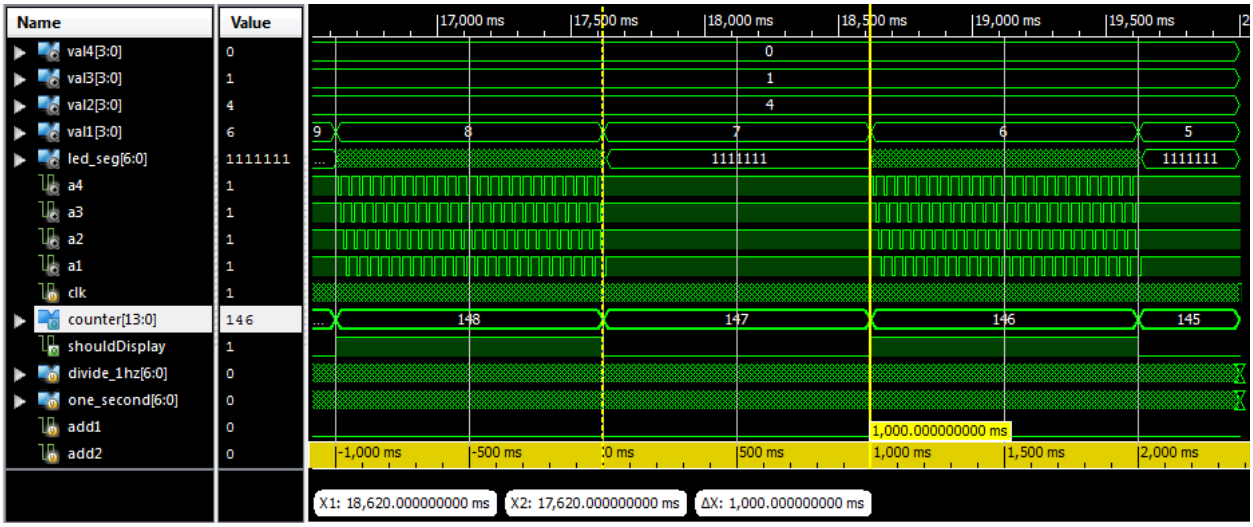


Figure 21: Simulation results for test 5. The 7 segment is turned off for the entire duration when the time left is 14'd147, and turns on again for the full duration of even times left, such as 14'd148 and 14'd146. The counting down behavior works correctly, transitioning from 14'd147 to 14'd146 after 1 s. The BCD outputs are also correct.

Test 6: Output Behavior During Transition from ≥ 180 s to < 180 s

The purpose of this test is to check the output behavior when the time remaining goes from more than 180 s to less than 180 s. At 181 s and 180 s, the 7 segment display should display at 1 Hz with a 50% duty cycle, and then it should be tuned off for 179 s, and then display for the full 1 second when the time remaining is 178 s. We enter this duration of the time remaining from setting `rst2` to high, which sets the time remaining to 150 s, and then adding 60 more seconds with `add1`. The code shown below is for tests 3-6.

```
----------------------Tests 3-6 Testbench Code----------------------
// ... Test 2 code ...                  //Tests 4 and 5: confirm
// counter is currently 675             // flashing at 0.5 Hz on even #s
#10000; //wait 10 s                     #10000; //wait 10 s
//Tests 3 and 4: confirm                add1 = 1;
// flashing at 1 Hz, > 180              #10;
rst2 = 1;                               add1 = 0;
#10;                                    //  Test 6: confirm transition
rst2 = 0;                               // from >= 180 to < 180 is correct
                                        #25000; // go from >= 180 to < 180
                                        // ...Test 7 code ...
--------------------------------------------------------------------
```

(simulation results on next page)

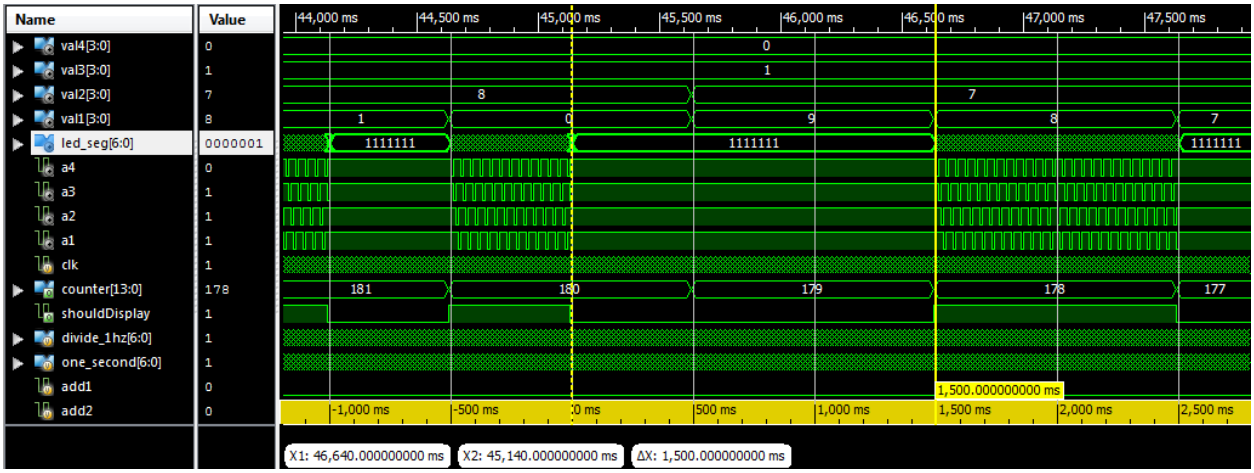


Figure 22: Simulation results for test 6. When the time remaining is 14'd181 or 14'd180, the 7 segment display flashes at 1 Hz with a 50% duty cycle, and it turns off when `counter` becomes 14'd179, and then is on for the entire duration of 14'178. The BCD outputs show the correct value. From when `counter` is 14'd180 to when it turns into 14'd178, the 7 segment display is off for 1.5 s total as expected, due to 0.5 from 14'd180 and then off for 1 s from 14'd179.

Test 7: Latch to 14'd9999

The purpose of this test is to ensure that the amount of time remaining, stored in `counter`, cannot exceed 14'd9999, and any attempts to increase it beyond that value will result in it latching to 14'd9999. This was achieved by keeping `add4` high for a long duration of time, thus ensuring that the time remaining reached 14'd9999, and then attempting to add more time via other `add<x>` inputs.

```
-------------------------Test 7 Testbench Code-------------------------
// ... Test 6 code ...                  | add1 = 1;
add4 = 1;                               | //Test 7: should latch to 9999
#3000; //Test 7: latch to 9999          | #10;
add4 = 0;                               | add1 = 0;
#5000; //wait 5 seconds                 | // ...Test 8 code ...
-----------------------------------------------------------------------
```

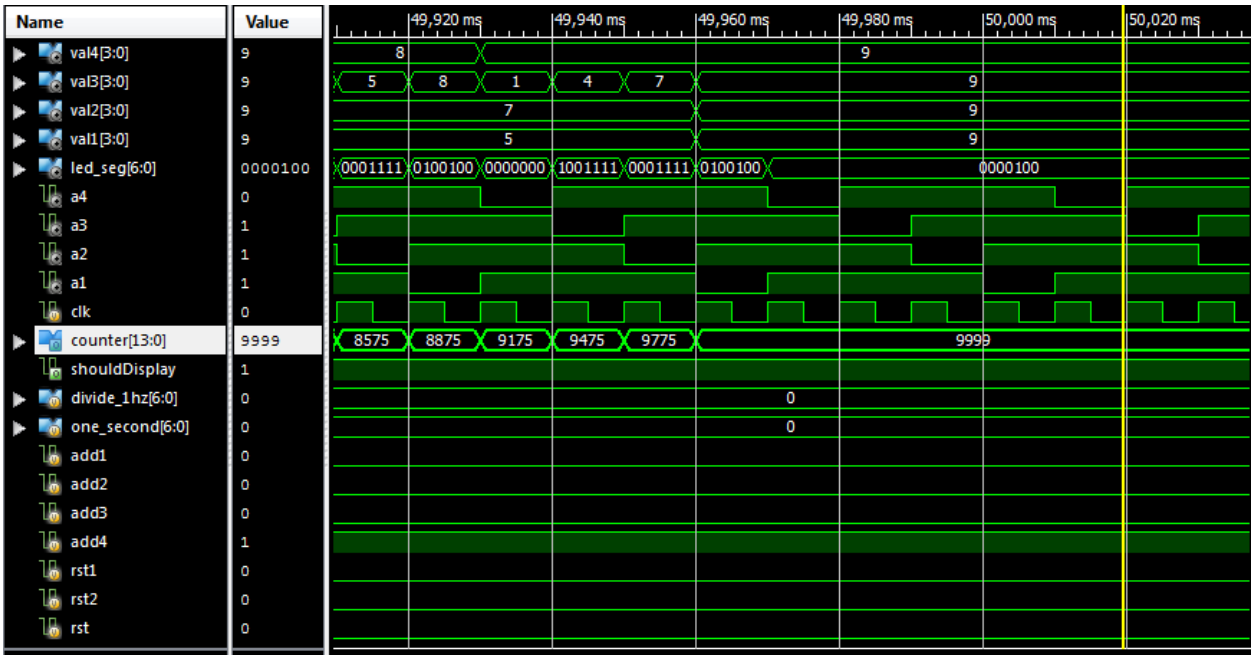


Figure 23: Simulation results for test 7. This shows how when `add4` is kept on, the value of counter increases by 400 every clock cycle until it reaches 14'd9999, where it then stays at that value. The 7 segment also stays on because we are "pressing" the button the whole time, and the BCD outputs update correctly as well.

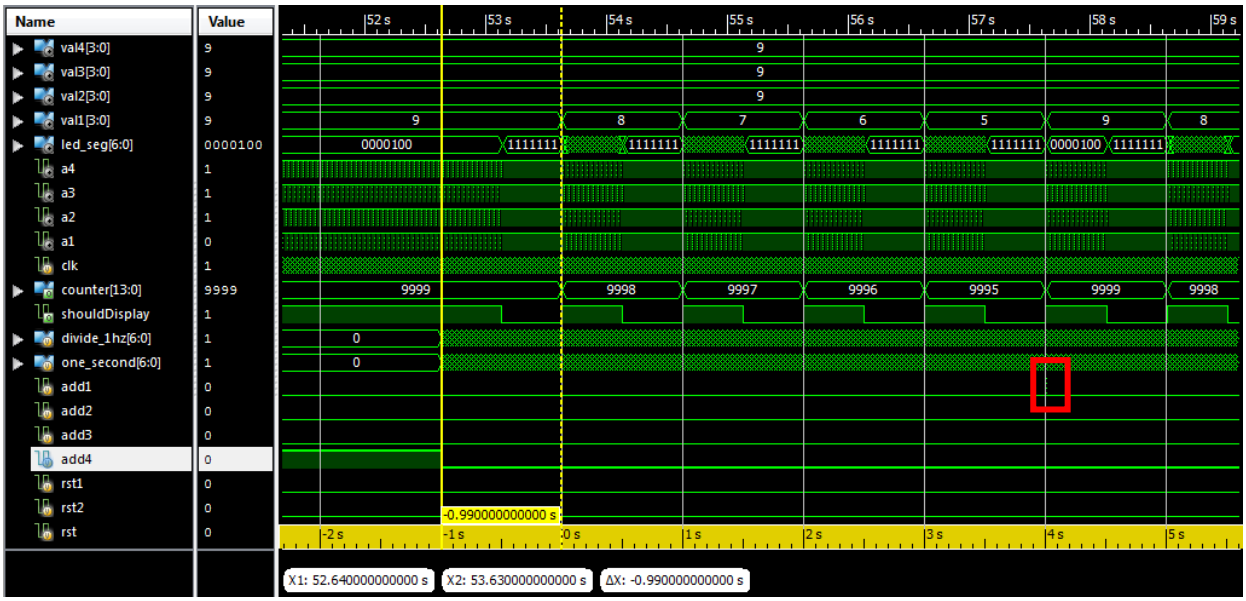


Figure 24: Simulation results for test 7. This shows how once `add4` becomes low, the regular counting down logic begins and it correctly transitions from 14'd9999 to 14'd9998 after 1 s. The two time cursors are 0.99 s apart because the left time cursor is positioned on the falling edge of add4, so the last time it was high on the rising edge of the clock was 1 clock cycle ago, or 0.01 s, so the behavior is still correct, as the parking meter spends 0.99 s + 0.01 s = 1 s in the 14'd9999 state since we tried to add time to it. When we try to add more time with `add1` (the barely perceptible spike in the red box), the time left does not go past 14'd9999, as expected.

Test 8: Multiple Simultaneous Inputs

The purpose of this test is to turn on multiple inputs at the same time and ensure that: (1) they are take effect according to the signal hierarchy given below Table 1, and (2) that the presence of simultaneous inputs does not result in undefined or unexpected behavior. To do this, multiple inputs were set high at the same time in the test bench code, and the simulation results were checked to make sure the correct one took precedence.

```
-------------------------Test 8 Testbench Code-------------------------
// ... Test 7 code …                    add4 = 0;
//Test 8: rst2 takes precedence         add3 = 0;
rst2 = 1;                               add2 = 0;
rst1 = 1;                               //Test 8: rst takes precedence
#10;                                    //over add
rst2 = 0;                               rst1 = 1;
rst1 = 0;                               add1 = 1;
//Test 8: add4 takes precedence         #10;
add4 = 1;                               rst1 = 0;
add3 = 1;                               add1 = 0;
add2 = 1;                               // ...Test 9 code ...
#10;
-----------------------------------------------------------------------
```

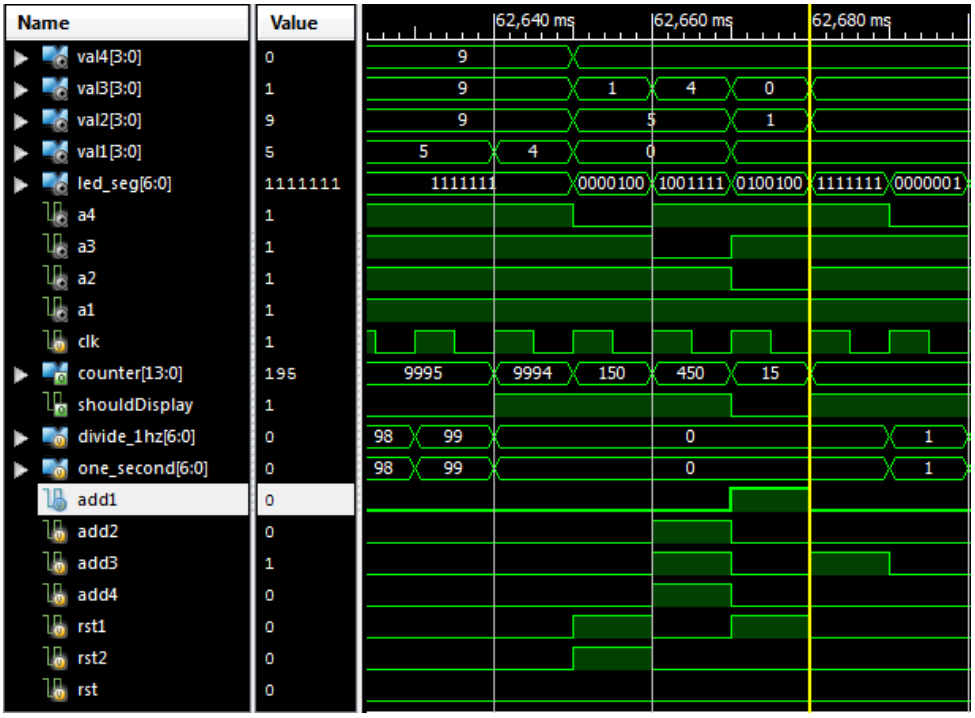


Figure 25: Simulation results for test 8. When both `rst1` and `rst2` are on, `rst2` takes correctly takes precedence, and likewise for the combination `add4`, `add3`, and `add2` and the combination `rst1` and `add1`. Although not all combinations were tested, the sign that these worked shows that other combinations would most likely work as well, as the code is structured identically for all inputs except for `rst`, which takes precedence over all other inputs.

Test 9: Normal Operation from ≥ 180 s to 0 s

The purpose of this test is to let the meter run from a starting amount of time left that is greater than 180 s all the way down to 0 s to verify that it works for normal operation. Above 180 s, the 7 segment display should flash at 1 Hz, and below 180 s but not at 0 s, it should flash at 0.5 Hz, lit up on even values. Once it reaches 0 s, it should go back to flashing at 1 Hz. The duty cycle is always 50%. To quickly get to the smallest time remaining larger than 180 s, `rst1` was turned high to set the time left to 15 s, and then 180 more seconds were added with `add3`. Then, the testbench waited 200 s to let the time run out.

```
-------------------------Test 9 Testbench Code-------------------------
// ... Test 8 code …                    add3 = 1; //Test 9: increase to
rst1 = 1; //Test 8: rst takes           195 s
precedence over add                     #10;
add1 = 1;                               add3 = 0;
#10; //timer now at 15 s                #200000; //Test 9: wait 200 s to
rst1 = 0;                               run down to 0
add1 = 0;                               // ...Test 10 code ...
-----------------------------------------------------------------------
```

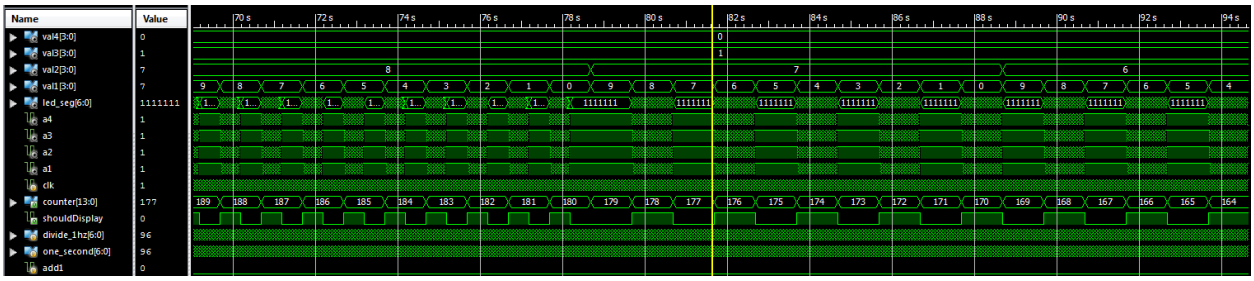


Figure 26: Simulation results for Test 9, showing time remaining values between 14'd189 and 14'd164. The display frequency of the 7 segment display correctly changes when the time remaining becomes less than 14'd180. The BCD outputs also display the correct digits.

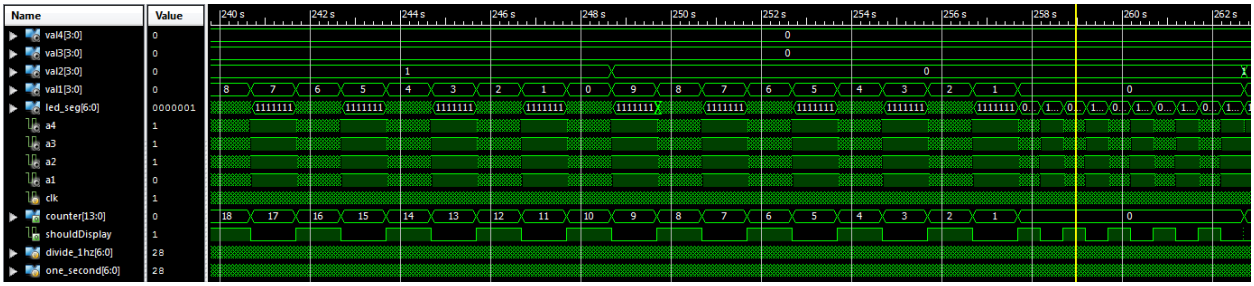


Figure 27: Simulation results for Test 9, showing time remaining values between 14'd18 and 14'd0. The display frequency of the 7 segment display correctly changes when the time remaining becomes 0. The BCD outputs also display the correct digits. Upon reaching 0 s left, the parking meter correctly stays in that state until another input is provided.

Test 10: Add Time, Resulting Time is < 180 s and Odd

The purpose of this test is to verify that when time is added to the parking meter, but the resulting time is less than 180 s and odd, it is not displayed, as only even values should be displayed when the time left is less than 180 s. To do this, the parking meter was set to have a time left of 15 s using `rst1`, and then 60 s were added using `add1` to cause it to have 75 s left. The 7 segment display should remain turned off until the time left becomes 14'd75.

```
------------------------Test 9 Testbench Code-------------------------
// ... Test 9 code ...                 | //Test 10: add 60 to make 75 s
rst1 = 1; //Test 10: reset to 15 s     | #10;
#10;                                   | add1 = 0; //Test 10: should
rst1 = 0;                              | // display on 74 s left
add1 = 1;                              | // ...Test 11 code ...
-----------------------------------------------------------------------
```

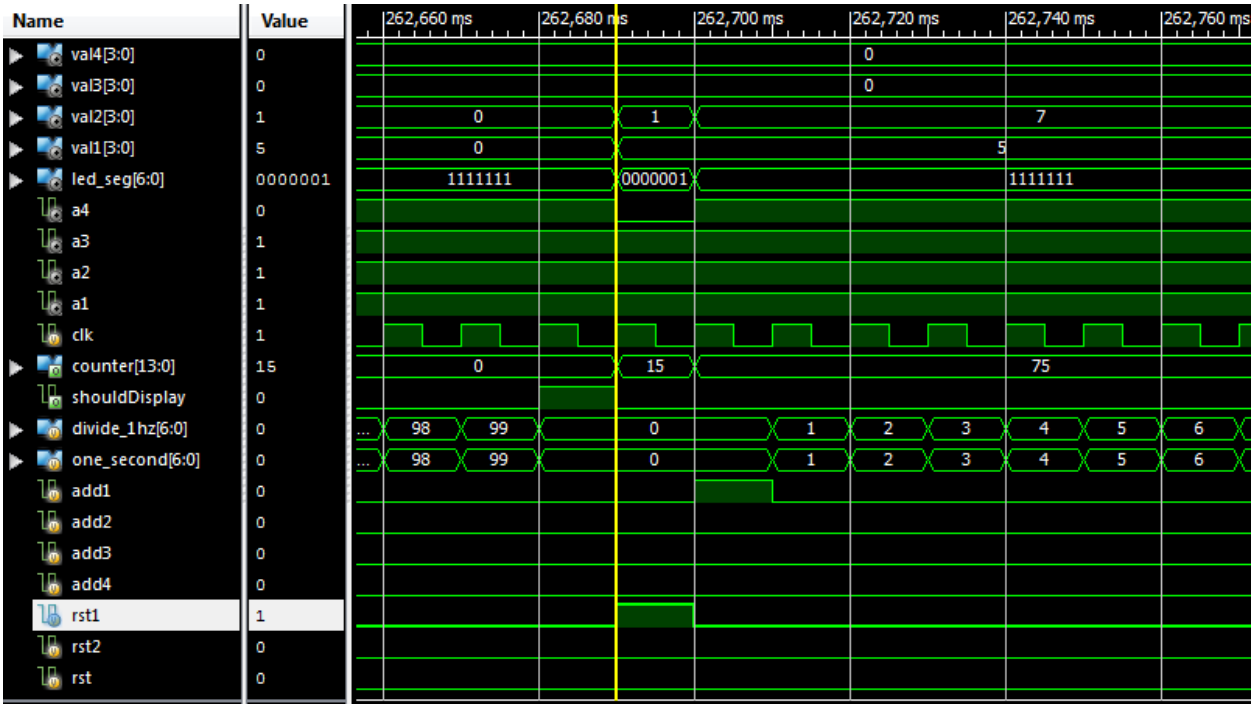


Figure 28: Simulation results for Test 10, showing that when the time left becomes 14'd75 after `rst1` and then `add1` become high, the 7 segment display does not turn on. When `rst1` is high, `led_seg` is not 7'b1111111 because the seven_seg sequential module updates its input 1 clock cycle after changes in `counter`. This 1 cycle delay is expected and has been previously observed and mentioned.

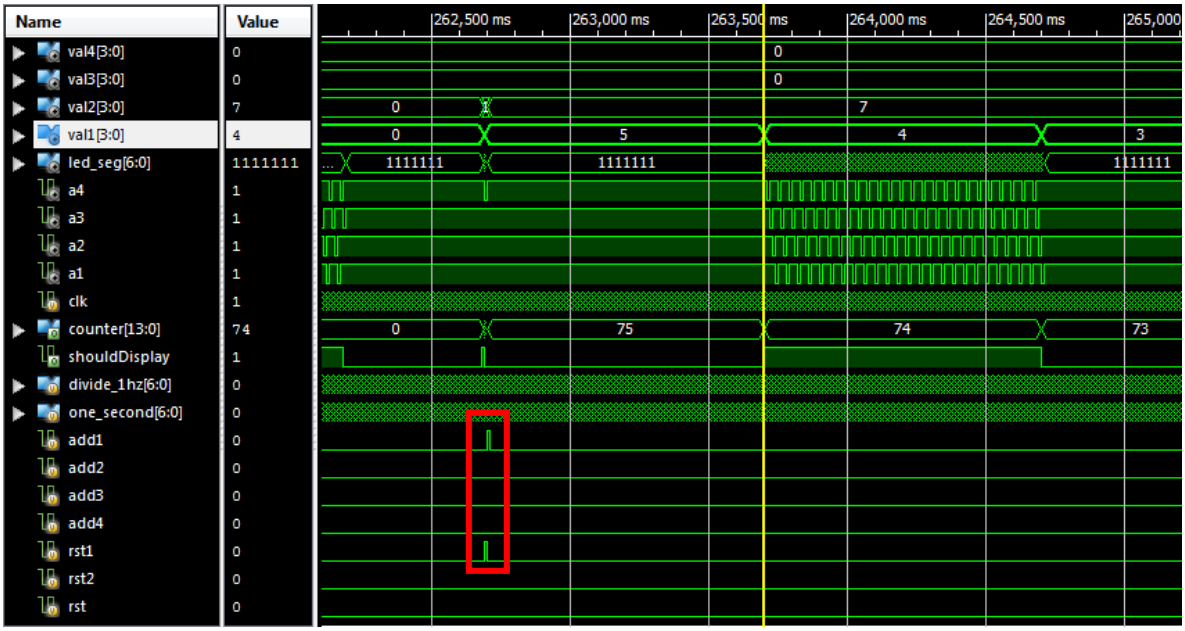


Figure 29: Simulation results for Test 10, showing that the 7 segment display does not turn on until the time remaining becomes 14'd74, as expected. The two inputs `rst1` and `add1` becoming high in rapid succession are highlighted in the red box.

Test 11: Extremely Short Input Duration

The purpose of this test is to verify that the button inputs are synchronous; that is, they only trigger edge of the input clock signal `clk`. Thus, if a button is pressed for a very short time in between rising edges of a clock, it should have no effect. The test bench code below tests this, where the `add4` input goes high 2 ms after the rising edge of the clock and goes low 2 ms after, which is before the next rising edge of the clock.

```
------------------------Test 11 Testbench Code-----------------------
  // ... Test 9 code …                  #2;
  #2;                                   add4 = 0;
  add4 = 1;                          end //end initial always block
  //Very short input (2 ms)
---------------------------------------------------------------------
```

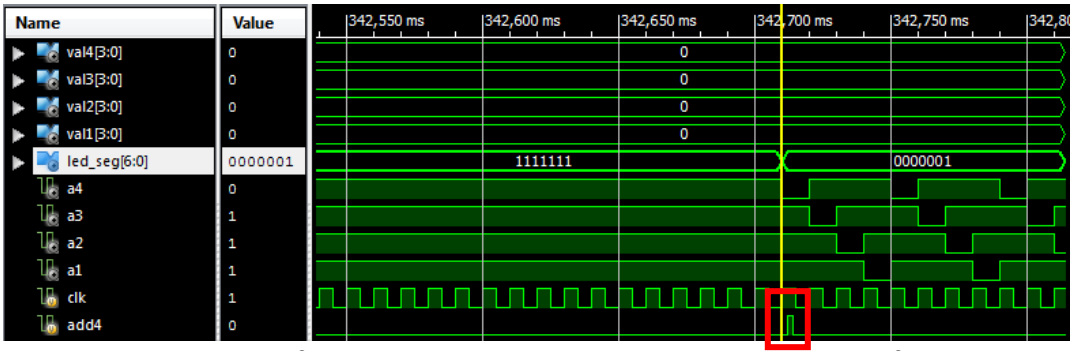


Figure 30: Simulation results for Test 11, showing a very short input pulse of `add4` (red box) that does not cause it to be high on a positive edge of the clock has no effect. The time remaining stays at 14'd0, as expected.

Finally, an attempt was made to test the parking meter counting down from its maximum of 9999 s all the way to 0 s, but it seems that Isim does not allow a simulation time of more than 9000 s, so unfortunately, this test could not be done.

After synthesis, the design summary section of the text report shows that 116 of 18224 slice registers were used, as well as 3213 of 9112 slice lookup tables (LUTs). Of the 3289 flip flop pairs used, 3173 had an unused flip flop, 76 had an unused LUT, and 40 used both the LUT and flip flop. There are 35 bonded IOBs, corresponding to the total number of input and output bits for the top level parking_meter module. Notable macros that were used were a 16x7 bit single port distributed read only RAM for the `bcd_to_7seg` module (see Figure 13), a 4x4 bit version of the same type of RAM for the `seven_seg` module (see Figure 14), multipliers for the `bcd_vals` module, and a 14 bit comparators for the `timer` module to compare if the time left was less than or greater than particular constants (14'd180, 14'd0, etc., see Figures 3 and 9). A variety of adders, subtractors, and up counters were used to handle adding and subtracting values in various submodules, as well as multiplexers to handle choosing between various values. Finally, a large number of macros were synthesized to synthesize the division used in the `bcd_vals` module. A macro does not seem to exist for division, so the synthesis tool had to manually create one from comparators, multiplexers, other macros, and logic gates.

After implementation, the mapping report showed that the number of slice registers used increased to 117 of 18224, where 116 were used as flip flops (consistent with synthesis report) and 1 was used for AND/OR logic. The number of slice LUTs was reduced to 3074 of 9112, and of these 3074, 3072 were used as logic, while 2 were used only for route thrus. 959 of 2278 slices are occupied, and of the 3111 LUT flip flop pairs used (a reduction from the synthesis amount of 3289), 2999 had an unused flip flop, 37 had an unused LUT, and 75 used both. The number of bonded IOBs remained at 35.

Overall, the slice register usage is low, while the LUT usage is roughly 1/3rd of the resources available on the board. This is much higher than previous projects primarily due to the use of division, which is an expensive computation to implement on this FPGA board. However, the resource limits are not exceeded, so this design is feasible to implement on a real board. The design summary section of the synthesis report is given in Appendix B and the mapping report is given in Appendix C.

(conclusion on next page)

Conclusion

In this lab, a parking meter was modeled using Verilog that allows users to set the meter's time to particular values or add more time to the meter up to a maximum of 9999 s, automatically counts down the time remaining, and displays the time remaining on a 7 segment display and 4 BCD outputs was created. The parking meter was designed by modeling it as a Moore machine and then implementing it with submodules that kept track of the time remaining and automatically decremented it, handled user inputs, decided when the 7 segment display should turn on, and converted the remaining time to 4 BCD digit outputs and a 7 segment display output with cathode and anode signals. Testing this parking meter with a test bench shows that it satisfies the expected behavior, displaying the remaining time with the expected frequency and values for the 7 segment display and with the expected values for the BCD outputs. It correctly handles user inputs, limits the maximum amount of time to 9999 s, handles when time runs out, and changes the 7 segment display frequency correctly depending on the time remaining. Through this lab, I learned how to model a FSM in Verilog, test it with testbench code and Isim, and write Verilog code for a 7 segment display.

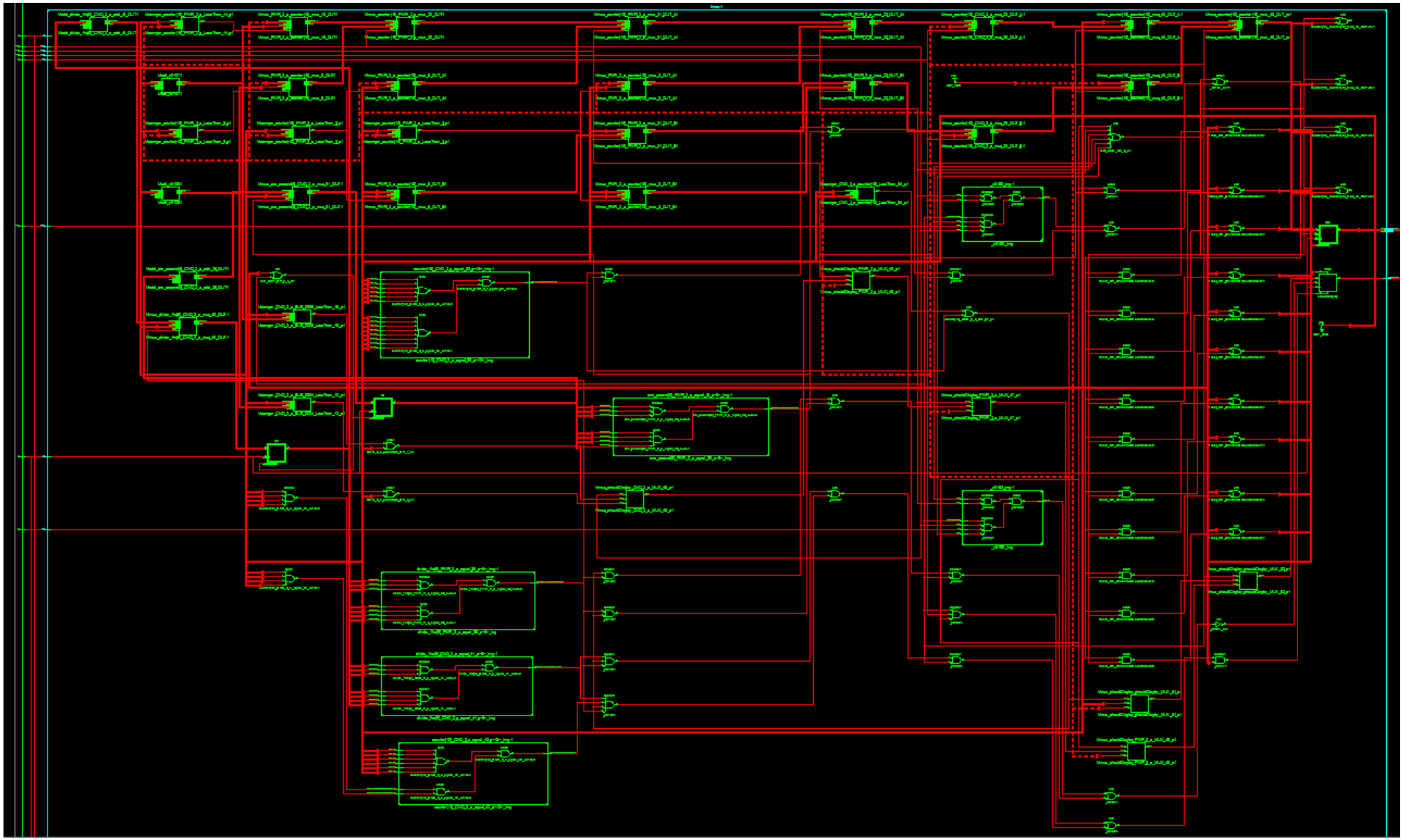
The main difficulty I encountered was figuring out how the 7 segment display worked and writing code for it. It took some time to understand that only 1 unique digit can be displayed on the 7 segment display at any time, and that even though cycling through the 4 digits mean that 3 of them will be turned off while 1 is turned on, because of the clock speed, this will be imperceptible to the human eye. Also, it felt odd that for the 7 segment display, a low value (0) turns on the segment or enables the display for that digit. I resolved this problem by rereading the Nexsys3 documentation until I understood how the 7 segment display worked, as well as listening to my TA explain it.

Overall, this project was interesting and a good way to end the quarter, as I felt that I used concepts from all the previous projects to implement the parking meter. Doing this lab helped me reinforce the concepts I learned about state machines and also exposed me to the 7 segment display, which I've never worked with before. However, like Project 4, when this project specification was initially released, it was unclear and was missing some vital components, such as a system clock input and specification for the bit order of the cathodes in the `led_seg` output, and this prevented me from starting on the lab until there was only 1 week left, as I had to wait for my TA to address these missing pieces of information with the updated project specification document. However, this information has since been added to the project specification document, so future students will have a clearer project specification to work with.

Appendix

Appendix A: Full RTL Schematic for `timer` Module

The full RTL schematic shown below is essentially unreadable due to its large size, but I have included it for the sake of completeness to give the reader a sense of the full result when the code is synthesized. The inputs are on the left and the outputs are on the right. There are two registers on the right side for storing the values of `counter[13:0]` and `shouldDisplay` to output.



Appendix B: Synthesis Report

Note: Only the advanced HDL synthesis and design summary sections are included. Some text has been omitted from the advanced HDL synthesis section.

```
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
                        *** Some Text Omitted ***
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                  : 5
 16x7-bit single-port distributed Read Only RAM        : 4
 4x4-bit single-port distributed Read Only RAM         : 1
# Multipliers                                           : 3
 14x10-bit multiplier                                  : 1
 32x4-bit multiplier                                   : 1
 32x7-bit multiplier                                   : 1
# Adders/Subtractors                                    : 86
 14-bit adder                                          : 14
 14-bit addsub                                         : 1
 15-bit adder                                          : 2
 25-bit subtractor                                     : 1
 32-bit adder                                          : 65
 32-bit subtractor                                     : 1
 4-bit adder                                           : 1
 4-bit subtractor                                      : 1
# Counters                                              : 3
 2-bit up counter                                      : 1
 7-bit up counter                                      : 2
# Registers                                             : 26
 Flip-Flops                                            : 26
# Comparators                                           : 88
 14-bit comparator greater                             : 1
 14-bit comparator lessequal                           : 9
 15-bit comparator lessequal                           : 3
 16-bit comparator lessequal                           : 1
 17-bit comparator lessequal                           : 1
 18-bit comparator lessequal                           : 1
 19-bit comparator lessequal                           : 1
 20-bit comparator lessequal                           : 1
 21-bit comparator lessequal                           : 1
 22-bit comparator lessequal                           : 1
 23-bit comparator lessequal                           : 1
 24-bit comparator lessequal                           : 1
 32-bit comparator lessequal                           : 55
 33-bit comparator lessequal                           : 2
```

```
 34-bit comparator lessequal                       : 2
 35-bit comparator lessequal                       : 2
 36-bit comparator lessequal                       : 2
 37-bit comparator lessequal                       : 1
 38-bit comparator lessequal                       : 1
 39-bit comparator lessequal                       : 1
# Multiplexers                                     : 2042
 1-bit 2-to-1 multiplexer                          : 2016
 14-bit 2-to-1 multiplexer                         : 18
 32-bit 2-to-1 multiplexer                         : 6
 7-bit 2-to-1 multiplexer                          : 1
 7-bit 4-to-1 multiplexer                          : 1

=========================================================================
*                            Design Summary                             *
=========================================================================
Top Level Output File Name         : parking_meter.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 5121
#      GND                         : 1
#      INV                         : 73
#      LUT1                        : 15
#      LUT2                        : 121
#      LUT3                        : 360
#      LUT4                        : 164
#      LUT5                        : 1099
#      LUT6                        : 1381
#      MUXCY                       : 963
#      MUXF7                       : 63
#      VCC                         : 1
#      XORCY                       : 880
# FlipFlops/Latches                : 116
#      FDR                         : 13
#      FDRE                        : 95
#      FDS                         : 8
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 34
#      IBUF                        : 7
#      OBUF                        : 27
# DSPs                             : 3
#      DSP48A1                     : 3
```

```
Device utilization summary:
---------------------------
Selected Device : 6slx16csg324-3

Slice Logic Utilization:
 Number of Slice Registers:             116  out of  18224     0%
 Number of Slice LUTs:                 3213  out of   9112    35%
    Number used as Logic:              3213  out of   9112    35%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:   3289
   Number with an unused Flip Flop:    3173  out of   3289    96%
   Number with an unused LUT:            76  out of   3289     2%
   Number of fully used LUT-FF pairs:    40  out of   3289     1%
   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%
 Number of DSP48A1s:                     3  out of     32     9%
=====================================================================
```

Appendix C: Mapping Report

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                117 out of  18,224    1%
    Number used as Flip Flops:              116
    Number used as Latches:                   0
    Number used as Latch-thrus:               0
    Number used as AND/OR logics:             1
  Number of Slice LUTs:                   3,074 out of   9,112   33%
    Number used as logic:                 3,072 out of   9,112   33%
      Number using O6 output only:        2,400
      Number using O5 output only:           17
      Number using O5 and O6:               655
      Number used as ROM:                     0
    Number used as Memory:                    0 out of   2,176    0%
    Number used exclusively as route-thrus:   2
      Number with same-slice register load:   0
      Number with same-slice carry load:      2
      Number with other load:                 0
```

```
Slice Logic Distribution:
  Number of occupied Slices:                  959 out of   2,278   42%
  Number of MUXCYs used:                    1,116 out of   4,556   24%
  Number of LUT Flip Flop pairs used:       3,111
    Number with an unused Flip Flop:        2,999 out of   3,111   96%
    Number with an unused LUT:                 37 out of   3,111    1%
    Number of fully used LUT-FF pairs:         75 out of   3,111    2%
    Number of unique control sets:              6
    Number of slice register sites lost
      to control set restrictions:             20 out of  18,224    1%
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:                           3 out of      32    9%
  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:               4.71
```