Joshua Aymett

COM SCI M152A-6

03/14/2021

Project 4: Parking Meter

Behavioral Summary

This project involved the design and implementation of a parking meter. The meter takes 8 inputs: a clock pulse, three reset switches, and four inputs that add various amounts of time to the counter.

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

Image from CS m152A Project 4 assignment

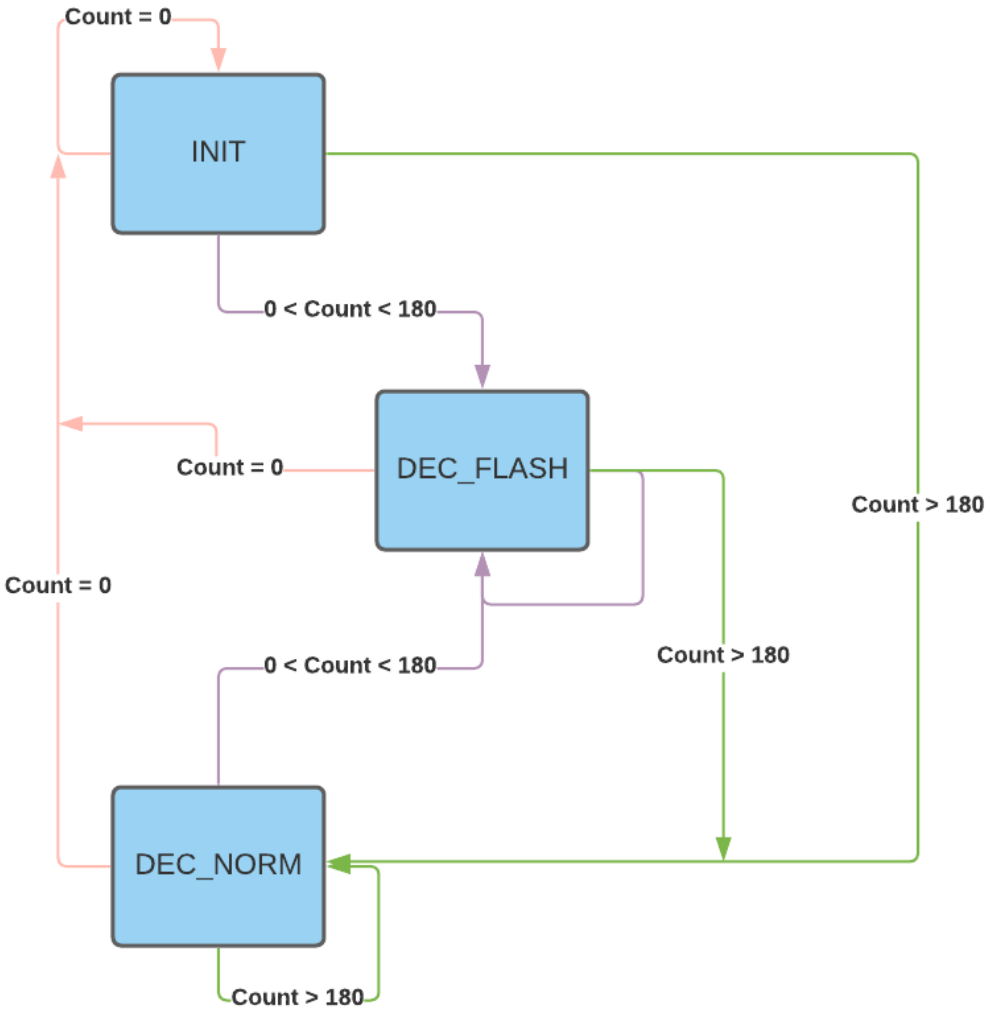
The meter outputs data for four seven-segment displays and are summarized below.

led_seg	7-bit vector labeled A-G which controls which segments are active (active low)
a1	1-bit anode selector for right-most digit (active low)
a2	1-bit anode selector for middle-right digit (active low)
a3	1-bit anode selector for middle-left digit (active low)
a4	1-bit anode selector for left-most digit (active low)
val1	4-bit binary coded decimal output for right-most digit
val2	4-bit binary coded decimal output for middle-right digit
val3	4-bit binary coded decimal output for middle-left digit
val4	4-bit binary coded decimal output for left-most digit

In addition to counting down, the meter also flashes when the remaining time is below 180 seconds. While more than 0 seconds remain, it flashes with a 50% duty cycle two second period. Once the clock reaches 0 seconds either via countdown or reset, it flashes with a 50% duty cycle and one second period. The maximum value of the display is 9999 and any attempts to exceed this value will result in the counter remaining at its maximum value.

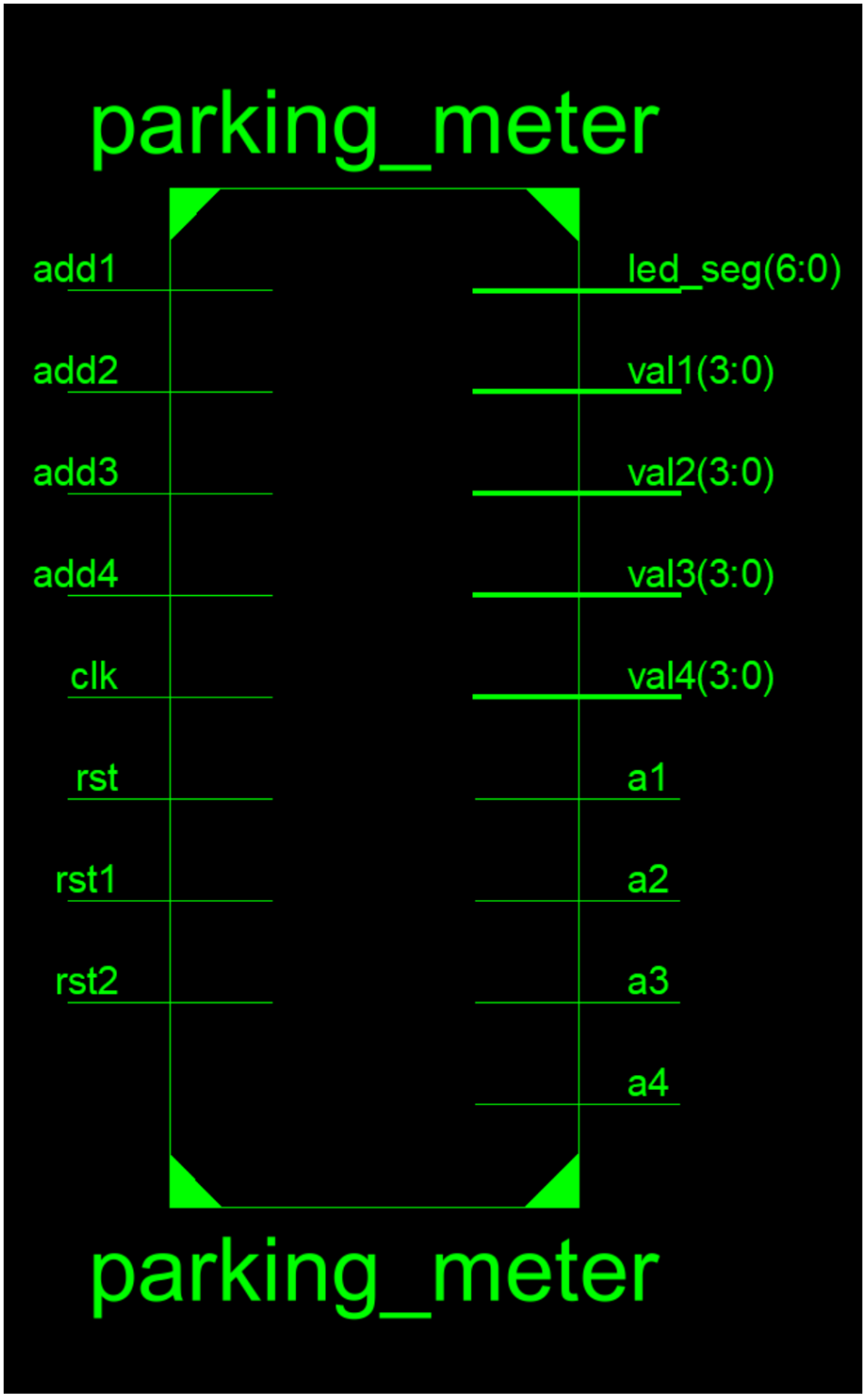
The parking meter is designed as a finite state machine where the value of the counter dictates the state transitions. It is a Moore machine and its outputs depend entirely upon state. INIT represents the initial

state of 0 seconds which flashes with a one second period. DEC_FLASH represents 0 < Count < 180 where the meter flashes with a two second period. DEC_NORM represents Count > 180 with no flashing.

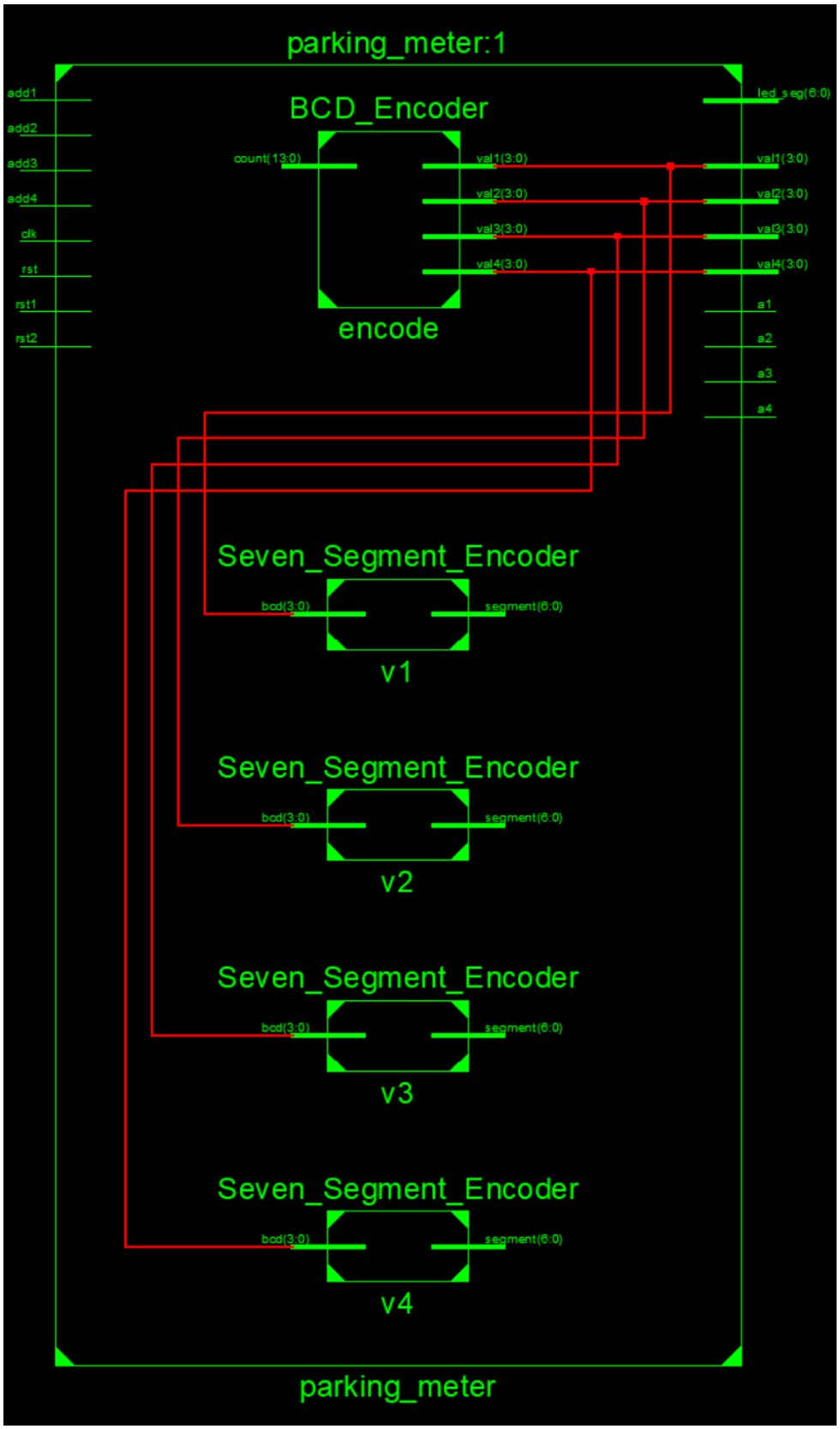


Finite State Machine Diagram displaying state transitions

Schematics



Black box of parking_meter module which displays inputs and outputs



Schematic displaying main module with submodules inside

parking_meter Project Status (03/14/2021 - 11:49:47)			
Project File:	Project4.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:**	36 Warnings (25 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)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	61	18,224	1%	
Number used as Flip Flops	33			
Number used as Latches	28			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	250	9,112	2%	
Number used as logic	246	9,112	2%	
Number using O6 output only	154			
Number using O5 output only	26			
Number using O5 and O6	66			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number used exclusively as route-thrus	4			
Number with same-slice register load	0			
Number with same-slice carry load	4			
Number with other load	0			
Number of occupied Slices	87	2,278	3%	
Number of MUXCYs used	76	4,556	1%	
Number of LUT Flip Flop pairs used	254			
Number with an unused Flip Flop	196	254	77%	
Number with an unused LUT	4	254	1%	
Number of fully used LUT-FF pairs	54	254	21%	
Number of unique control sets	12			
Number of slice register sites lost to control set restrictions	51	18,224	1%	
Number of bonded IOBs	35	232	15%	
IOB Latches	4			
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	1	16	6%	
Number used as BUFGs	1			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	248	0%	
Number of OLOGIC2/OSERDES2s	4	248	1%	
Number used as OLOGIC2s	4			
Number used as OSERDES2s	0			
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	32	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	3.65			

Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	
Timing Constraints:	All Constraints Met			

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Mar 14 21:37:22 2021	0	31 Warnings (20 new)	4 Infos (2 new)
Translation Report	Current	Sun Mar 14 21:37:37 2021	0	0	0
Map Report	Current	Sun Mar 14 21:38:01 2021	0	5 Warnings (5 new)	6 Infos (6 new)
Place and Route Report	Current	Sun Mar 14 21:38:15 2021	0	0	3 Infos (3 new)
Power Report					
Post-PAR Static Timing Report	Current	Sun Mar 14 21:38:25 2021	0	0	4 Infos (4 new)
Bitgen Report					

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
ISIM Simulator Log	Out of Date	Sun Mar 14 21:07:32 2021	

Design Summary: The presence of slice registers indicate sequential logic

Logical Summary

The parking meter was implemented as a Moore machine where state transitions are handled based on remaining time as described previously.

After declaring module inputs and outputs, parameters were declared representing states and quantities of seconds to add. Immediately following, all registers are defined. These registers store the current remaining time (count), current and next state registers, a counter use as a clock divider (clock_count), an anode selector which constantly increments to select which digit of the display will be updated, multiplexed wires to carry seven segment encoded values to the 7-bit vector output, and a toggle to count parity of seconds for use in the DEC_FLASH state.

```
module parking_meter(
  // ….Inputs as described above
  );

        // State parameters
        parameter INIT = 2'b00;
        parameter DEC_NORM = 2'b01;
        parameter DEC_FLASH = 2'b10;

        // Add numbers
        parameter ADD_60 = 6'b111100;
        parameter ADD_120 = 7'b1111000;
        parameter ADD_180 = 8'b10110100;
        parameter ADD_300 = 9'b100101100;
        parameter MAX_COUNT = 14'b10011100001111;

        // Timer count
        reg [13:0] count = 0;

        // State registers
        reg [1:0] current_state;
        reg [1:0] next_state;

        // Clock divider
        reg [26:0] clock_count = 0;

        // Anode selector
        reg [1:0] anode_selector = 0;

        // Seven segment displays for each bcd
        wire [6:0] val1_seg;
        wire [6:0] val2_seg;
        wire [6:0] val3_seg;
        wire [6:0] val4_seg;

        // DEC_COUNT on/off flag
        reg dec_toggle_flag = 1'b1;
```

Immediately following this code is an always block which servers to update the state and count. The clock count is incremented each positive edge of the clock as is the anode selector. Reset flags are checked and the counter is set to its corresponding value depending on which reset flag is high. If no reset flag is high, the current state is set to the next state, the clock count used to divide the 100HZ frequency into 1 HZ is checked, and the counter is decremented if one second has passed. Notice how the dec_toggle_flag is complemented every second. This is used by later code to check the parity of the counter and only display digits on even numbers while in the DEC_FLASH state.

```
always@(posedge clk) begin
        // Increment clock divider
        clock_count <= clock_count + 1'b1;

        // Increment anode selector to update at clock frequency
        anode_selector <= anode_selector + 1'b1;

        if(rst) begin
                current_state <= INIT;
                count <= 0;
                clock_count <= 0;
        end
        else if (rst1) begin
                current_state <= DEC_FLASH;
                count <= 16;
                clock_count <= 0;
        end
        else if (rst2) begin
                current_state <= DEC_FLASH;
                count <= 150;
                clock_count <= 0;
        end
        else begin
                current_state <= next_state;

                // Decrement every second
                if (clock_count >= 100 & count >= 14'b00000000000000) begin
                        if (count != 14'b00000000000000) // don't reset to negative
                                count <= count - 1'b1;
                        clock_count <= 0;

                        // Complement flag to track parity of second count
                        dec_toggle_flag <= ~dec_toggle_flag;
                end
        end
```

After checking for reset conditions and updating the state, add flags are checked and their corresponding values are added to the counter. If the added value exceeds the maximum capabilities of the display, MAX_COUNT = 9999, the display is set to MAX_COUNT. Once this is completed, the machine checks for the current count and updates its next state accordingly.

```
        // Increment count based on inputs while accounting for max limits
        if (add1) begin
                if (count + ADD_60 > MAX_COUNT)
                        count <= MAX_COUNT;
                else count <= count + ADD_60;
        end
        else if (add2) begin
                if (count + ADD_120 > MAX_COUNT)
                        count <= MAX_COUNT;
                else count <= count + ADD_120;
        end
        else if (add3) begin
                if (count + ADD_180 > MAX_COUNT)
                        count <= MAX_COUNT;
                else count <= count + ADD_180;
        end
        else if (add4) begin
                if (count + ADD_300 > MAX_COUNT)
                        count <= MAX_COUNT;
                else count <= count + ADD_300;
        end

        if (count == 0) begin
                next_state <= INIT;
        end
        else if (count <= 180)
                next_state <= DEC_FLASH;
        else
                next_state <= DEC_NORM;
end
```

Following the state transitions, the machine instantiates two submodules: BCD_Encoder and Seven_Segment_Encoder. The BCD encoder takes a binary value and outputs four 4-bit BCD values corresponding to the four separate digits of the display. The Seven Segment Encoder accepts the BCD value outputted by BCD_Encoder and encodes it into seven segment display format with an active low.

```
// Update BCD outputs
BCD_Encoder encode(.count(count), .val1(val1), .val2(val2), .val3(val3), .val4(val4));

// Update registers for seven segment vectors for each display digit
Seven_Segment_Encoder v1(.bcd(val1), .segment(val1_seg));
Seven_Segment_Encoder v2(.bcd(val2), .segment(val2_seg));
Seven_Segment_Encoder v3(.bcd(val3), .segment(val3_seg));
Seven_Segment_Encoder v4(.bcd(val4), .segment(val4_seg));
```

After updating the wires which store the seven segment values, the output is decoded and flashing is handled as described above.

```
//Output Decoder
always @(*) begin
	// Update seven segment value
	case (anode_selector)
		// ...
		// Multiplexes which 7-segment value should be sent to the 7-bit vector output
	endcase

	// Handle which segment is on
	case(current_state)
		INIT: begin
			if (clock_count > 50) begin
				case (anode_selector)

				endcase
			end
			else begin
				// Output 1's for a1-a4 indicating all segments off
			end

		end
		DEC_FLASH: begin
			if (dec_toggle_flag) begin
				case (anode_selector)
					// Switch between anodes, outputting with no flashes
			end
				endcase
			end
			else begin
				// Output 1's for a1-a4 indicating all segments off
			end
		end
		DEC_NORM: begin
			// Switch between anodes, outputting with no flashes
		end
	endcase
end
endmodule
```

With all other logic complete, the Seven_Segment_Encoder is implemented as a single case statement where the outputs are updated in the format of A, B, C, D, E, F, G.

```
// Uses active low (0 -> segment on)
module Seven_Segment_Encoder (
                input [3:0] bcd,
                output reg [6:0] segment
        );
        parameter ZERO = 4'b0000;
        parameter ONE = 4'b0001;
        parameter TWO = 4'b0010;
        parameter THREE = 4'b0011;
        parameter FOUR = 4'b0100;
        parameter FIVE = 4'b0101;
        parameter SIX = 4'b0110;
        parameter SEVEN = 4'b0111;
        parameter EIGHT = 4'b1000;
        parameter NINE = 4'b1001;

        // Segments are outputted ABCDEFG
        always @(bcd) begin
                case (bcd)
                ZERO: // Segments A, B, C, D, E, F
                        segment = 7'b0000001;
                ONE: // Segments B and C
                        segment = 7'b1001111;
                TWO: // Segments A, B, D, E, G,
                        segment = 7'b0010010;
                THREE: // Segments A, B, C, D, G
                        segment = 7'b0000110;
                FOUR: // Segments B, C, F, G,
                        segment = 7'b1001100;
                FIVE: // Segments A, C, D, F, G
                        segment = 7'b0100100;
                SIX: // Segments A, C, D, E, F, G
                        segment = 7'b0100000;
                SEVEN: // Segments A, B, C
                        segment = 7'b0001111;
                EIGHT: // Segments A, B, C, D, E, F, G
                        segment = 7'b0000000;
                NINE: // Segments A, B, C, D, F, G
                        segment = 7'b0000100;
                endcase
        end
        endmodule
```

Finally, the BCD_Encoder is implemented using the double dabble algorithm which shifts a BCD input of n bytes n times, adding 3 to each partitioned nibble of trailing 0's to the left which exceeds the number 4. Description of logic can be found at https://en.wikipedia.org/wiki/Double_dabble

```
// Double dabble algorithm
module BCD_Encoder (
                input [13:0] count,
                output reg [3:0] val1,
                output reg [3:0] val2,
                output reg [3:0] val3,
                output reg [3:0] val4
        );
        reg [89:0] store;
        integer i;
        always @(count) begin
                store = count;
                for (i=0; i < 14; i = i + 1) begin
                        store = store << 1;
                        if (i < 13) begin
                                if (store[17:14] > 4'b0100)
                                        store[17:14] = store[17:14] + 2'b11;
                                if (store[21:18] > 4'b0100)
                                        store[21:18] = store[21:18] + 2'b11;
                                if (store[25:22] > 4'b0100)
                                        store[25:22] = store[25:22] + 2'b11;
                                if (store[29:26] > 4'b0100)
                                        store[29:26] = store[29:26] + 2'b11;
                        end

                end
                val1 = store[17:14];
                val2 = store[21:18];
                val3 = store[25:22];
                val4 = store[29:26];
        end

        endmodule
```

Simulation Output

The testbench is designed to be comprehensive and includes testing of all resets, all states, a countdown to 0, countdowns greater than 0, attempts at exceeding the maximum display value, idle in the initial state. Its logic is as follows:

// Set to 16s

// Waits 20s. Time flashes during countdown with 2s period. flashes during init state (0000) with 1s period.

// Set to 150s

// Wait 1s to decrement to 149s

// Add 60s to make count equal 209s

// Add 120s to make count equal 329s

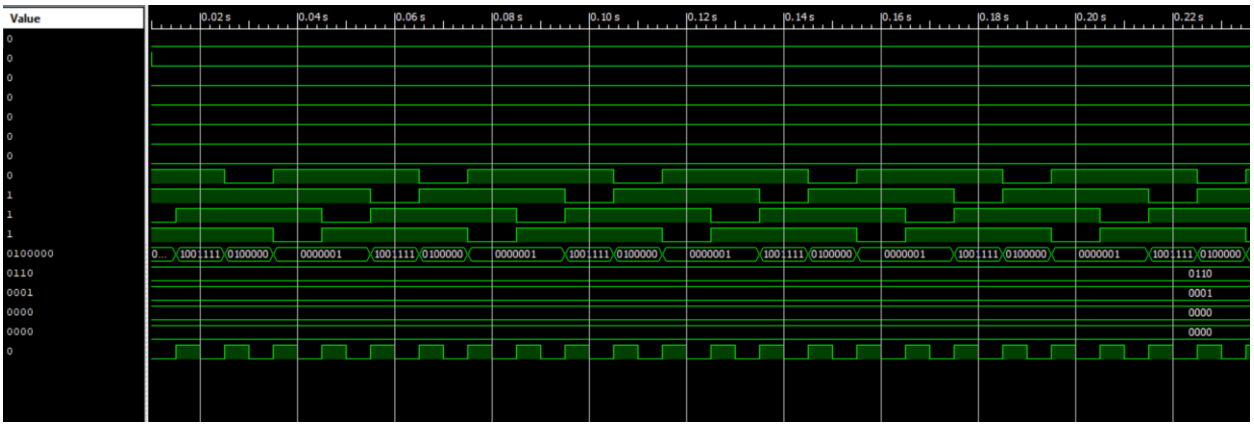
// Add 180s to make count equal 509s

// Add 300s to make count equal 809s

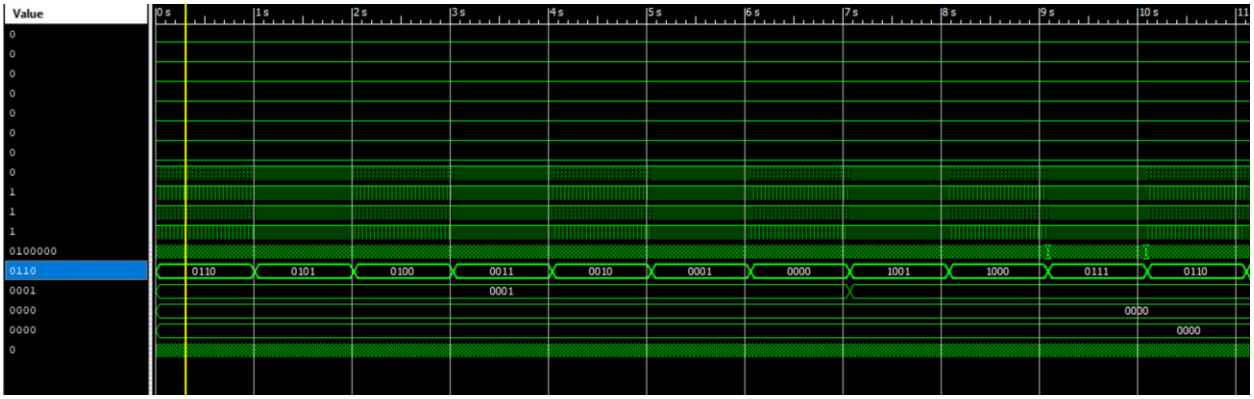
// Add 300s repeatedly to test overflow and set to 9999s

// Let decrement for 10 seconds down to 9989

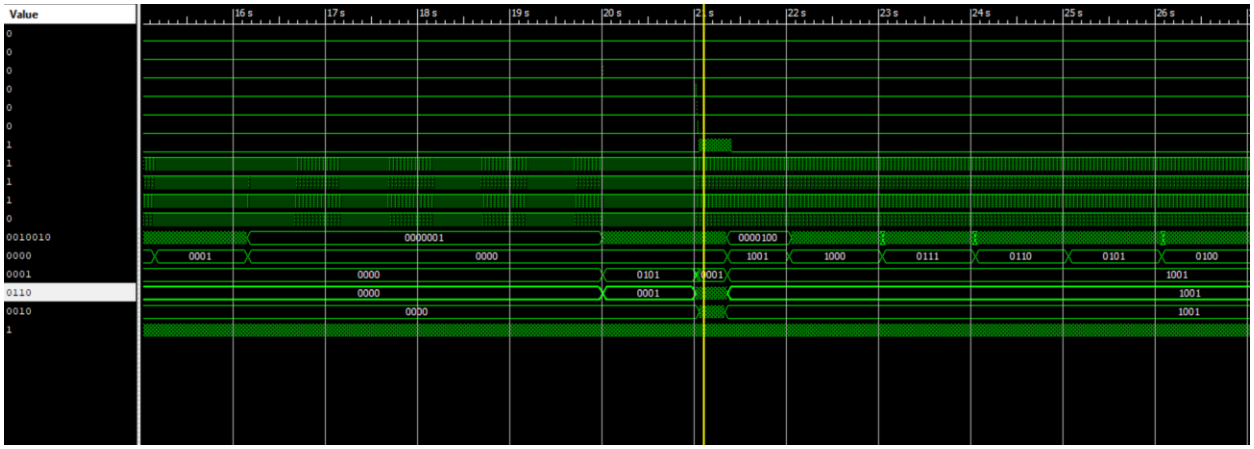
// Reset to 0s and watch flash



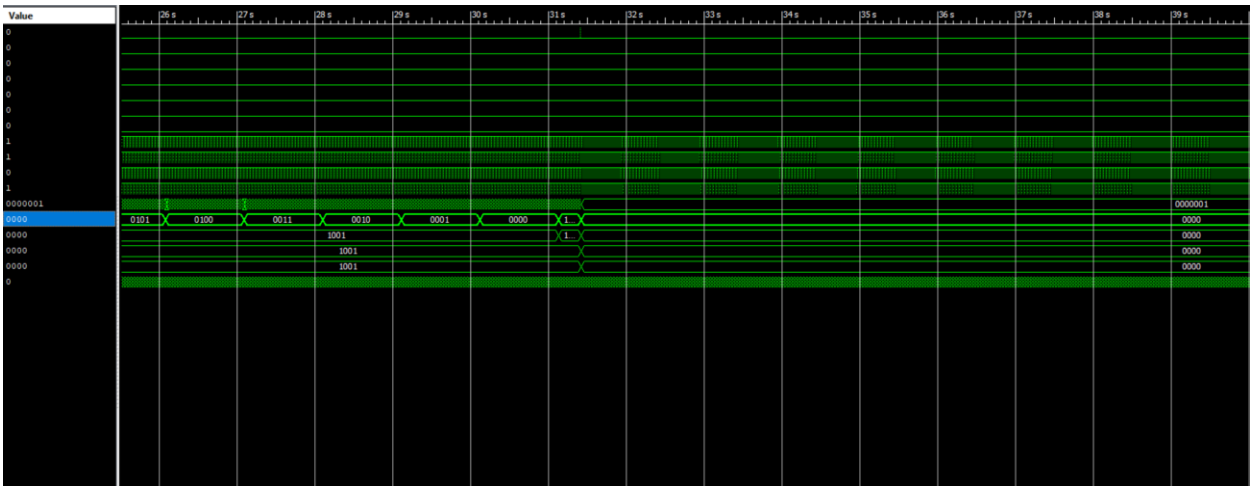
Simulation output immediately following reset to 16s. Output is correct because it shows the current BCD values in val1-val4, continuous updating and changing of the anode selection bits, and seven segment display vectors which match the correct BCD values.



Simulation output showing decrements after reset. Output is correct because the counter is decrementing every second and the display is flashing with a 50% duty cycle and 2 second period.



Simulation output showing attempts at exceeding the maximum value. Output is correct because the counter remains at 9999 and then begins to decrement from 9999 once additions have stopped.



Simulation output displaying behavior once the counter has reached 0. The output is correct because the counter ceases to decrement and instead flashes all 0's with a 50% duty cycle and 1 second period.

Post-Project Evaluation

This project was my favorite of the four projects because I felt like it was less logically intensive than the vending machine and more challenging than the first two projects. I liked how behavior was both simple and clearly defined and that the project had multiple possible methods of solving the given task including some without the use of states. I felt like this project further developed my Verilog and logic design skills and believe that it is an excellent project to assign during finals due to its reasonable difficulty and moderate time requirements. To improve this project, I would give at least two specific examples of how the seven segment display works that explain step-by-step how each set of segments updates. Because the display is not intuitive, it was difficult to understand it at first and examples would have immediately removed all confusion. Aside from this minor shortcoming, the directions were well phrased and easy to understand.