Final Project

Finite State Machine

Cow Drinks

Emma Burkett

07/23/2024

Description of the Project:

This project has five states that indicate when a ‘cow’ should drink water and expel the water. The drinking is controlled by the potentiometer and the current state is indicated by the offboard red, yellow, and green LEDs. The state is also indicated by the seven-segment display. The analog input is displayed by the on-board LEDs for testing as sometimes the potentiometer is a little finicky.

First Peer Review:

Jared suggested taking our original reminder to drink water for a human and turn it into a cow. The project was much more interesting this way, so we changed it.

Michael suggested that we make our timer module accommodate all our timers into one always block. This added an interesting addition to our finite state machine.

Second Peer Review:

We didn’t remember to do an official second peer review, but we did have quite a few peers look over our project, like Val and Jerad. And they were very positive and supportive.

Conclusion Statement:

The finite state machine, the seven segment display, and the off-board LEDs had 100% functionality. Incorporating Brother Jack’s modified ADC code proved difficult. We got it to run but it would occasionally have errors due to how our timers were set up. The seven-segment display would sometimes display values that we did not write. We also should have included a default value for our seven-segment display, but we ran out of time.

Other than that one error our code functioned as expected so in total we had about 99% functionality. Our Video demonstration does not include the potentiometer in it because we ran out of time, and we had already demonstrated it to Brother Jack.

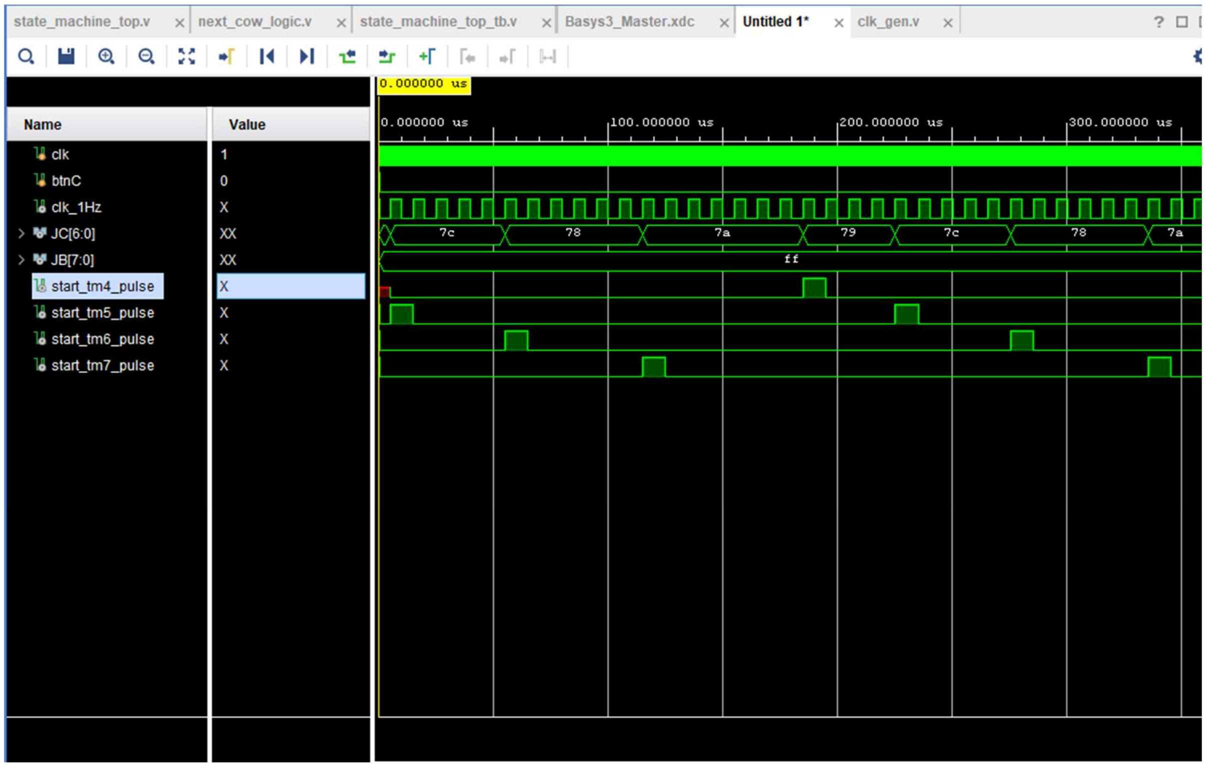
Because our output logic relies on the next state instead of the current state the finite state machine is Mealy than Moore because the outputs do not wait for the clock cycle. This was an error born from misunderstanding lab 7 and the fact that the pulses happened a clock cycle too late, and the states would change unexpectedly on us. Brother Jack told us we could simulate it and identify the error and that would probably fix the seven-segment display but unfortunately due to time constraints we never got to it.

Please see the PDF for Finite State Machine details.

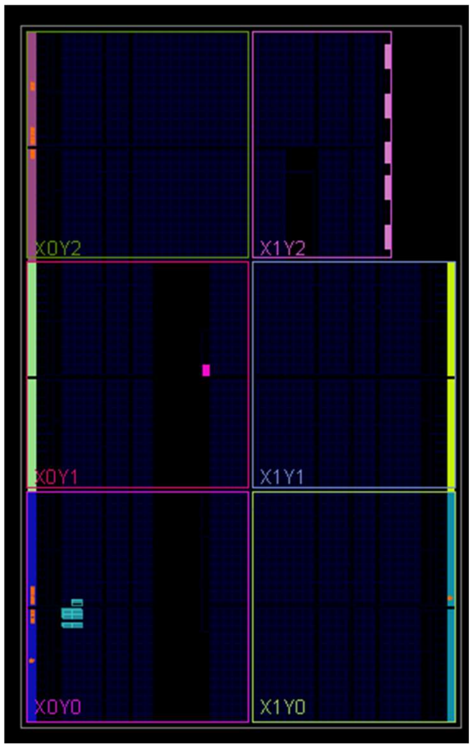
Code for test bench:

```
module state_machine_top_tb(

    );
    // This is a basic testbench for the finite state machine
    reg clk;
    reg btnC;
    wire [6:0] JC;
    wire [7:0] JB;

     // Instance of the state machine.
    state_machine_top state_inst(.clk(clk), .btnC(btnC),
                                .JC(JC), .JB(JB));
    // Run the clock at 10MHz
    always #5 clk = !clk;

    initial
        begin
             // Instantiate values like clk and reset(BTNC)
             clk = 1;
             btnC = 0;
             #20 btnC = 1;
             #20 btnC = 0;
             #1000 $finish;
        end
endmodule
```

Test Bench images for finite state machine only!:

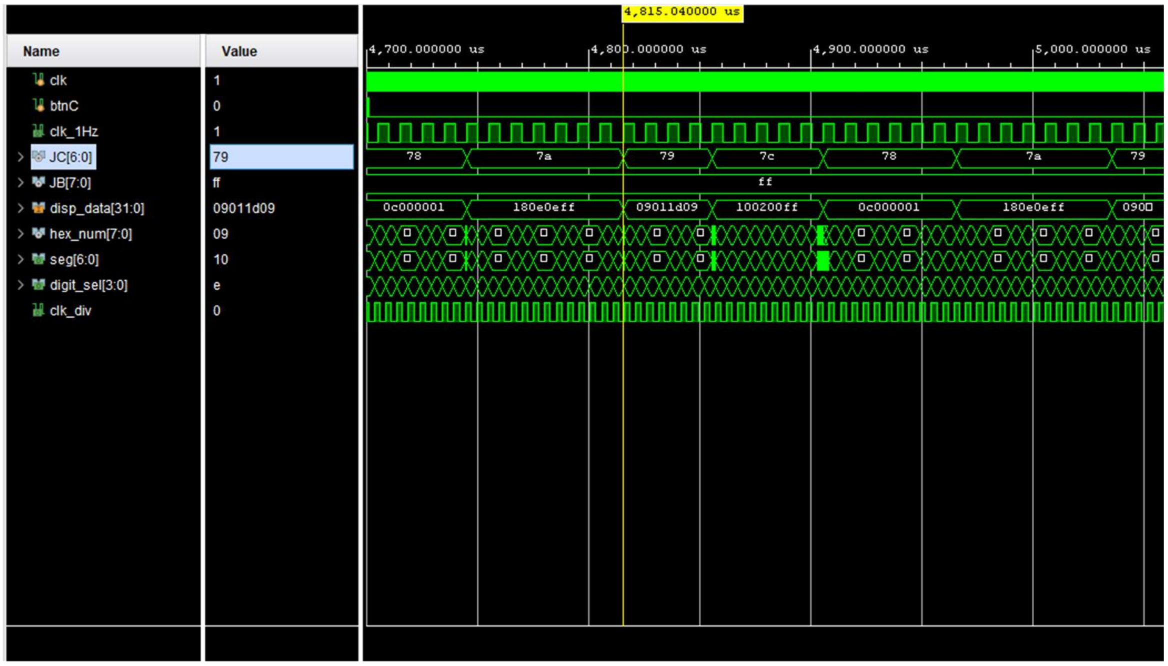


Simulation 1: Finite state Machine Test Bench



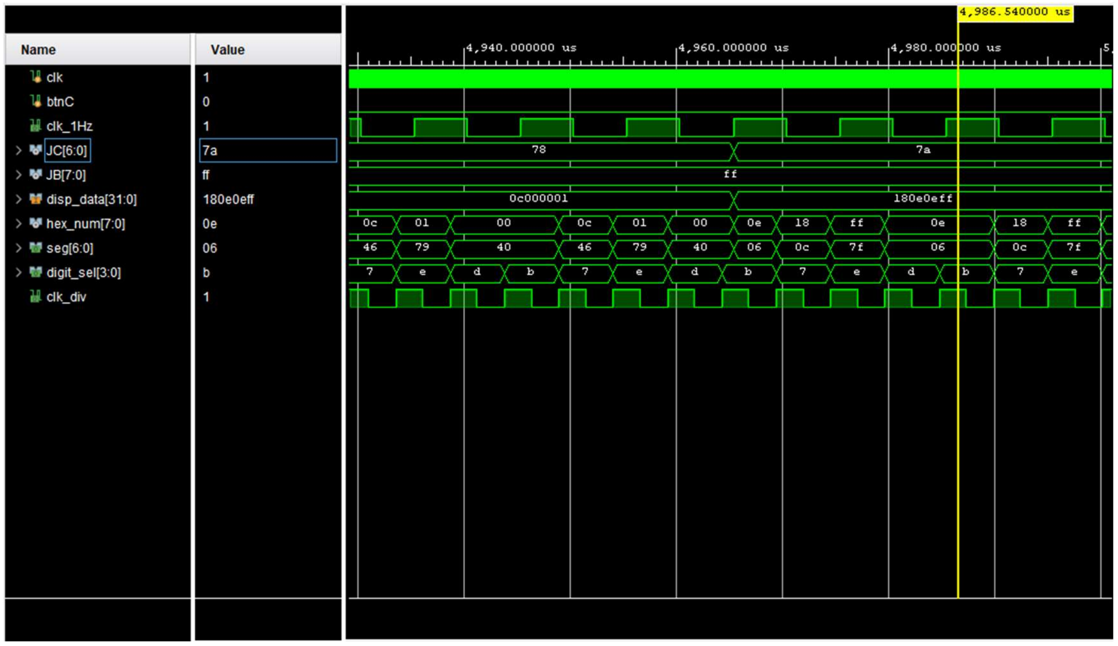
Implementation 1: Finite State Machine Only

Test bench images for finite state Machine and seven seg display:



Name	Value
clk	1
btnC	0
clk_1Hz	1
JC[6:0]	79
JB[7:0]	ff
disp_data[31:0]	09011d09
hex_num[7:0]	09
seg[6:0]	10
digit_sel[3:0]	e
clk_div	0

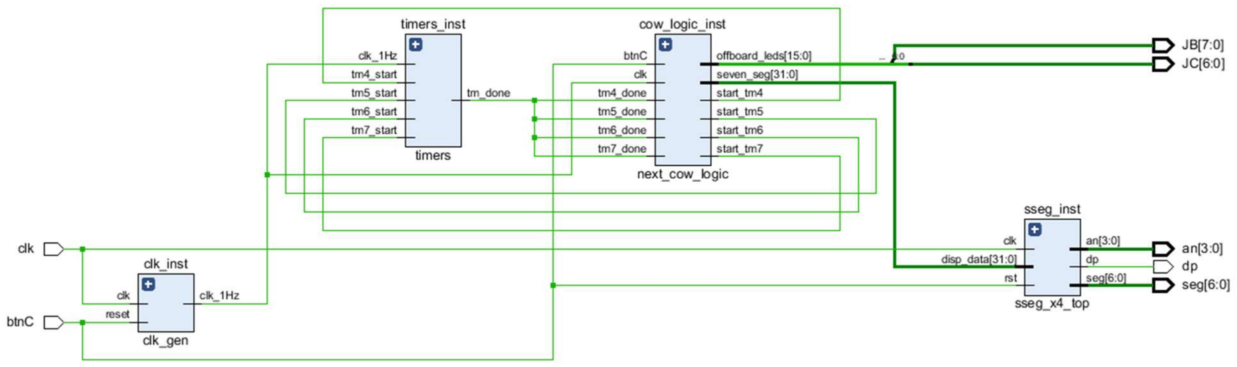
Simulation 2: After instantiating the seven segment display code we ran the same test bench again. It probably should not have worked because we didn't pass in an, seg, and dp into our state machine instantiation but it did.



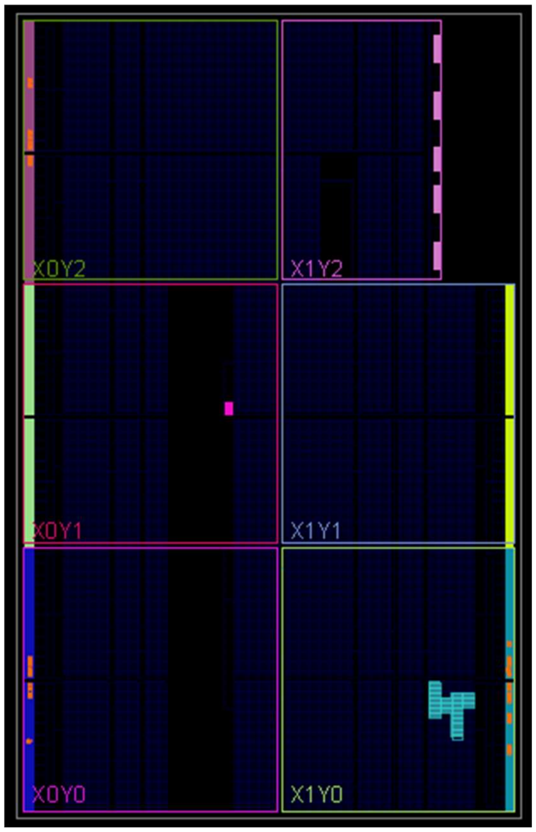
Name	Value
clk	1
btnC	0
clk_1Hz	1
JC[6:0]	7a
JB[7:0]	ff
disp_data[31:0]	180e0eff
hex_num[7:0]	0e
seg[6:0]	06
digit_sel[3:0]	b
clk_div	1

4,986.540000 us
4,940.000000 us
4,960.000000 us
4,980.000000 us
78
7a
ff
0c000001
180e0eff
0c 01 00 0c 01 00 0e 18 ff 0e 18 ff
46 79 40 46 79 40 06 0c 7f 06 0c 7f
7 e d b 7 e d b 7 e d b 7 e

Simulation 2b: Zoomed in to see the an[3:0] value changing and the seg[6:0] value.



Schematic 1: This schematic is cool because it is so visible <u>it is only the state machine and the segment display.</u>



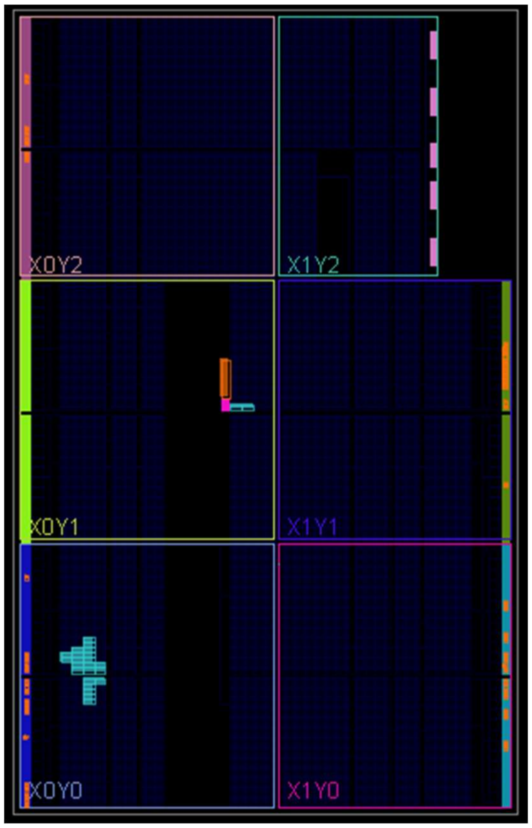
Implementation 2: Finite state machine and seven segment display

Additional Project Description and Video:

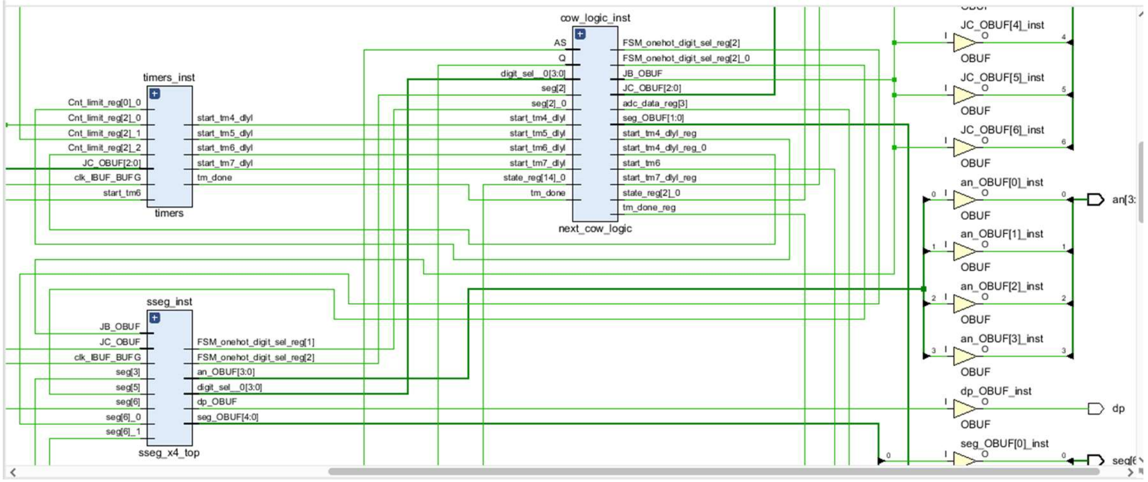
Here is our video demonstration on YouTube. It does not include the potentiometer; however we did demonstrate that functionality to Brother Jack on Monday or Friday. It does include the 7-segment display and the off board LEDs. The off board LEDs form the word cow, and they have three LEDs that turn on corresponding to what state you are in. You can refer to the PDF submitted with this document that outlines how long each state lasts and which red, yellow, or green LED lights up. The only state not demonstrated here is the ‘cow dead’ state as the cow dead state needs the potentiometer to have functionality.

https://youtu.be/75OV4b0kyMk

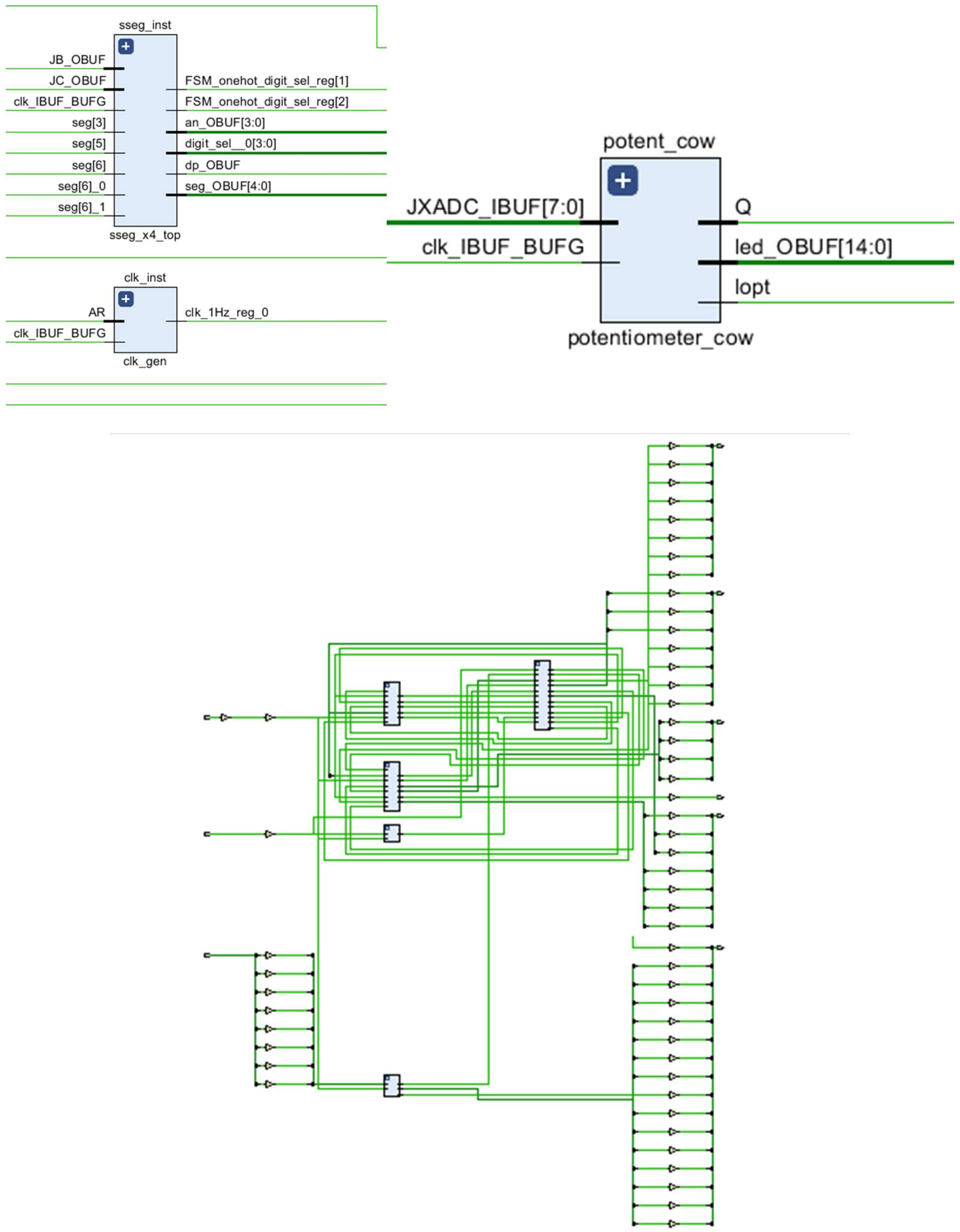
Images for the whole project: Note no simulation for the ADC data due to time constraints.



Implementation 3: Finite State Machine, Seven Seg Display, and Potentiometer.



Schematic 2: Finite State Machine, Seven Seg Display, and Potentiometer



Schematic 2b: Closer Images and Overview

Top Module

```
module state_machine_top(
    input clk, btnC,
    input  [7:0] JXADC,
    output [6:0] JC,
    output [7:0] JB,
    output [15:0] led,
    output [6:0]  seg,
    output [3:0]  an,
    output        dp
    );
    wire [11:0] adc_data;
    wire [14:0] offboard_leds;
    assign JB = offboard_leds[14:7];
    assign JC = offboard_leds[6:0];
    wire [31:0] seven_seg;

    clk_gen clk_inst(.clk(clk), .reset(btnC),
.clk_1Hz(clk_1Hz));

    next_cow_logic cow_logic_inst(
        .tm4_done(tm_done),
        .tm5_done(tm_done),
        .tm6_done(tm_done),
        .tm7_done(tm_done),
        .clk(clk_1Hz),
        .btnC(btnC),
        .start_tm5(start_tm5),
    .start_tm4(start_tm4),
    .start_tm6(start_tm6),
    .start_tm7(start_tm7),
    .next_state(next_state),
    .seven_seg(seven_seg),
    .offboard_led(offboard_leds),
    .water_content(adc_data)
);

timers timers_inst(
    .tm4_start(start_tm4),
    .tm5_start(start_tm5),
    .tm6_start(start_tm6),
    .tm7_start(start_tm7),
    .clk_1Hz(clk),
    .tm_done(tm_done)
);

potentiometer_cow potent_cow(
    .clk(clk),
    .JXADC(JXADC),
    .led(led),
    .adc_data(adc_data)
    );

sseg_x4_top sseg_inst(
    .seg(seg),
        .an(an),
        .dp(dp),
        .clk(clk),
        .rst(btnC),
        .disp_data(seven_seg)
    );

endmodule
```

1 Hz Clock:

```
module clk_gen(
    input clk, reset,
    output reg clk_1Hz
    );
    //reg [15:0] Q;
    reg [26:0] Q;
    always @(posedge clk, posedge reset)
        if (reset)
            begin
                Q <= 0;
                clk_1Hz <= 0;
            end
        else if (Q < 50000000)
             Q <= Q +1;
        else
            begin
                Q <= 0;
                clk_1Hz <= !clk_1Hz;
            end

endmodule
```

Next State Logic:

```
module next_cow_logic(
    input tm6_done, tm7_done, tm5_done, tm4_done, clk, btnC,
[11:0] water_content,
    output reg start_tm5, start_tm4, start_tm7, start_tm6,
    output reg [14:0] next_state,
    output reg [31:0] seven_seg,
    output reg [14:0] offboard_led
    );

    wire [14:0] cow_drinks;
    wire [14:0] cow_happy;
    wire [14:0] cow_potty;
    wire [14:0] cow_thirst;
    wire [14:0] cow_dead;

    assign cow_drinks [14:0] = 15'b111111111111100;
    assign cow_happy  [14:0] = 15'b111111111111000;
    assign cow_potty  [14:0] = 15'b111111111111010;
    assign cow_thirst [14:0] = 15'b111111111111001;
    assign cow_dead   [14:0] = 15'b000000000000000;

    reg [14:0] state;
    //Flip flops for state
    always @(posedge clk, posedge btnC)
        if (btnC)
            state <= cow_thirst;
        else
        state <= next_state;


    always @ *
        case(state)
            cow_drinks:
                if(water_content[3] == 1'b1 && tm5_done)
                    next_state = cow_happy;
                else if (water_content[3] == 1'b0 && tm5_done)
                    next_state = cow_dead;
                else
                    next_state = cow_drinks;
            cow_happy:
                if (tm6_done)
                    next_state = cow_potty;
                else
                    next_state = cow_happy;
            cow_potty:
                if (water_content[3] == 0'b0 && tm7_done)
                    next_state = cow_thirst;
                else if (water_content[3] == 1'b1 && tm7_done)
                    next_state = cow_dead;
                else
                    next_state = cow_potty;
            cow_thirst:
                if (tm4_done)
                    next_state = cow_drinks;
                else
                next_state = cow_thirst;
        cow_dead:
            next_state = cow_dead;
        default:
            next_state <= cow_thirst;
    endcase
// **Output logic**
always @(next_state)
    case(next_state)
        cow_drinks: //0h20
            begin
                start_tm4 = 0;
                start_tm5 = 1;
                start_tm6 = 0;
                start_tm7 = 0;
                offboard_led = cow_drinks;
                seven_seg = 32'h100200ff;
            end
        cow_happy: //c001
            begin
                start_tm4 = 0;
                start_tm5 = 0;
                start_tm6 = 1;
                start_tm7 = 0;
                offboard_led = cow_happy;
                seven_seg = 32'h0c000001;
            end
```

```
cow_potty: //pee
    begin
        start_tm4 = 0;
        start_tm5 = 0;
        start_tm6 = 0;
        start_tm7 = 1;
        offboard_led = cow_potty;
        seven_seg = 32'h180e0eff;
    end
cow_thirst: //glug
    begin
        start_tm4 = 1;
        start_tm5 = 0;
        start_tm6 = 0;
        start_tm7 = 0;
        offboard_led = cow_thirst;
        seven_seg = 32'h09011d09;
    end
cow_dead: //beef
    begin
        start_tm4 = 0;
        start_tm5 = 0;
        start_tm6 = 0;
        start_tm7 = 0;
        offboard_led = cow_dead;
        seven_seg = 32'h0b0e0e0f;
    end
```

```
            default:
                begin //potty
                    start_tm4 = 0;
                    start_tm5 = 0;
                    start_tm6 = 0;
                    start_tm7 = 1;
                    offboard_led = cow_potty;
                    seven_seg = 32'h190e0eff;
                end
        endcase

endmodule
```

// Timers for 4, 5, 6, & 7 seconds

```
module timers(
    input tm4_start, tm5_start, tm6_start, tm7_start, clk_1Hz,
    output reg tm_done
    );

    wire start_tm4_pulse;
    wire start_tm5_pulse;
    wire start_tm6_pulse;
    wire start_tm7_pulse;

    reg [2:0] Cnt;
    reg [2:0] Cnt_limit;
    reg  start_tm4_dlyl;
    reg  start_tm5_dlyl;
    reg  start_tm6_dlyl;
    reg  start_tm7_dlyl;

    always @ (posedge clk_1Hz)
        begin
            start_tm4_dlyl <= tm4_start;
            start_tm5_dlyl <= tm5_start;
            start_tm6_dlyl <= tm6_start;
            start_tm7_dlyl <= tm7_start;
        end

    assign start_tm4_pulse = tm4_start && !start_tm4_dlyl;
    assign start_tm5_pulse = tm5_start && !start_tm5_dlyl;
    assign start_tm6_pulse = tm6_start && !start_tm6_dlyl;
    assign start_tm7_pulse = tm7_start && !start_tm7_dlyl;
    // Integrated clock signals!
    always @ (posedge clk_1Hz)
        if (start_tm4_pulse)
            begin
                Cnt <= 0;
                tm_done <= 0;
                Cnt_limit <= 2;
            end
        else if (start_tm5_pulse)
            begin
                Cnt <= 0;
                tm_done <= 0;
                Cnt_limit <= 3;
            end
        else if (start_tm6_pulse)
            begin
                Cnt <= 0;
                tm_done <= 0;
                Cnt_limit <= 4;
            end
        else if (start_tm7_pulse)
            begin
                Cnt <= 0;
                tm_done <= 0;
```

```
                Cnt_limit <= 5;
            end
        else if (Cnt < Cnt_limit)
            Cnt <= Cnt +1;
        else
            tm_done <= 1;

endmodule
```

```
// Brother Jacks modified code - we remove the seven segment
display and output adc_data

module potentiometer_cow(

   input clk,

   input [7:0] JXADC,

   output reg [15:0] led,

   output reg [11:0] adc_data

 );


   wire [15:0] data_out; //Dynamic Reconfiguration Port (DRC)
data output

   wire [15:0] data_in; // DRC data input

   wire [15:0] aux_channel_n, aux_channel_p;  // Analog
differential inputs

   wire [6:0] DRP_address_in; //

   wire eoc_pulse; // use "End of Conversion" (EOC) to enable
DRP

   wire dwe_in; // DRC write enable










  assign DRP_address_in = 7'h16;

  // Analog inputs

  assign aux_channel_p = {JXADC[3], JXADC[1], 6'bzz_zzzz,
JXADC[2], JXADC[0], 6'bzz_zzzz};

  assign aux_channel_n = {JXADC[7], JXADC[5], 6'bzz_zzzz,
JXADC[6], JXADC[4], 6'bzz_zzzz};
```

```
  assign dwe_in = 1'b0;


  /* INIT_40 description

  15 14   13    12    11     10          9        8   7  6  5
4    3    2    1    0

CAVG, 0, AVG1, AVG0, MUX, Bip/Uni', Event/Cont', ACQ, 0, 0, 0,
CH4, CH3, CH2, CH1, CH0

CAVG = 1 to disable Calibration Averaging

AVG1, AVG0: 00 No Sample Averaging, 01 Average 16, 10 Average
64, 11 Average 256

MUX = 1 to enable external Multiplexer Mode

ACQ = 1 to increase settling time to 6 clock cycles (single chan
mode)

CH4 CH3 CH2 CH1 CH0

00000 On-Chip temp

00001 VCCINT

00010 VCCAUX

00011 VP, VN

00100 VREFP (1.25V)

00101 VREFN (0V)

00110 VCCBRAM

01101 VCCPINT

01110 VCCPAUX

01111 VCCO_DDR

1XXXX Select Auxiliary Input Channel XXXX (0-15)*/

/* INIT_41 description

 15    14    13    12    11    10     9     8     7     6     5
4     3     2     1   0
```

```
SEQ3, SEQ2, SEQ1, SEQ0, ALM6, ALM5, ALM4, ALM3, CAL3, CAL2,
CAL1, CAL0, ALM2, ALM1, ALM0 OT

SEQ3 SEQ2 SEQ1 SEQ0

0000 Default sequencer (monitors default voltages)

0001 Single pass sequence

0010 Continuous Sequence mode

0011 Single channel mode (Sequencer off)

01xx Simultaneous sampling mode

10xx Independent ADC mode

11xx Default mode

CAL3 = 1 Enables Supply sensor offset and gain correction

CAL2 = 1 Enables Supply sensor offset correction

CAL1 = 1 Enables ADC offet and gain correction

CAL0 = 1 Enables ADC offset correction

ALMx = 1 to disable respective alarm

OT = 1 to disable Over temperature signal */

/* INIT_42 description

 15   14   13   12   11   10    9    8  7  6   5    4   3  2  1
0

CD7, CD6, CD5, CD4, CD3, CD2, CD1, CD0, 0, 0, PD1, PD0, 0, 0, 0,
0

CD7-CD0 Sets the DCLK division (minimum is 2, maximum is 255)

PD1 PD0

00 = ALL ADC Blocks powered up

10 = ADC B powered down

11 = XADC Blocks powered down */


//xadc instantiation connect the eoc_out .den_in to get
continuous conversion
```

```
XADC #(

// Initializing the XADC Control Registers

.INIT_40(16'h8016),//For Aux6 continuous sampling and no
averaging: 16'h8016

.INIT_41(16'h3fff),//For Single channel, all alarms off, all
calibration off: 16'h3fff

.INIT_42(16'h0400),// Set DCLK divider to 4, ADC = 1Msps, DCLK =
100MHz

// not using the following registers

.INIT_48(16'h0000),// not using sequencer

.INIT_49(16'h0000),// not using sequencer

.INIT_4A(16'h0000),// not using averaging

.INIT_4B(16'h0000),// No averaging on external channels

.INIT_4C(16'h0000),// Sequencer Bipolar selection

.INIT_4D(16'h0000),// Sequencer Bipolar selection

.INIT_4E(16'h0000),// Sequencer Acq time selection

.INIT_4F(16'h0000),// Sequencer Acq time selection

.INIT_50(16'hb5ed),// Temp upper alarm trigger 85°C

.INIT_51(16'h5999),// Vccint upper alarm limit 1.05V

.INIT_52(16'hA147),// Vccaux upper alarm limit 1.89V

.INIT_53(16'h0000),// OT upper alarm limit 125°C using automatic
shutdown

.INIT_54(16'ha93a),// Temp lower alarm reset 60°C

.INIT_55(16'h5111),// Vccint lower alarm limit 0.95V

.INIT_56(16'h91Eb),// Vccaux lower alarm limit 1.71V

.INIT_57(16'hae4e),// OT lower alarm reset 70°C

.INIT_58(16'h5999),// VCCBRAM upper alarm limit 1.05V

.INIT_5C(16'h5111)// VCCBRAM lower alarm limit 0.95V

//.SIM_MONITOR_FILE("sensor_input.txt")
```

```
// Analog Stimulus file. Analog input values for simulation

)

XADC_INST ( // Connect up instance IO. See UG480 for port
descriptions

.CONVST(GND_BIT), // not used

.CONVSTCLK(GND_BIT), // not used

.DADDR(DRP_address_in),

.DCLK(clk),

.DEN(eoc_pulse),

.DI(data_in),

.DWE(dwe_in),

.RESET(),// not used

.VAUXN(aux_channel_n[15:0]),

.VAUXP(aux_channel_p[15:0]),

.ALM(),

.BUSY(),

.CHANNEL(),

.DO(data_out),

.DRDY(),

.EOC(eoc_pulse),

.EOS(),

.JTAGBUSY(),// not used

.JTAGLOCKED(),// not used

.JTAGMODIFIED(),// not used

.OT(),

.MUXADDR(),// not used

.VP(VP_IN),

.VN(VN_IN)
```

```
);

  always @ (posedge clk)
    if (eoc_pulse)
    begin
      adc_data <= data_out >> 4;
    end

 //led thermometer dmm
  always @(adc_data)
  begin
    case (adc_data[11:8])
      1:  led <= 16'b0000000000000011;
      2:  led <= 16'b0000000000000111;
      3:  led <= 16'b0000000000001111;
      4:  led <= 16'b0000000000011111;
      5:  led <= 16'b0000000000111111;
      6:  led <= 16'b0000000001111111;
      7:  led <= 16'b0000000011111111;
      8:  led <= 16'b0000000111111111;
      9:  led <= 16'b0000001111111111;
      10: led <= 16'b0000011111111111;
      11: led <= 16'b0000111111111111;
      12: led <= 16'b0001111111111111;
      13: led <= 16'b0011111111111111;
      14: led <= 16'b0111111111111111;
      15: led <= 16'b1111111111111111;
      default: led <= 16'b0000000000000001;
    endcase
  end
endmodule
```

```
module sseg_x4_top(
    output [6:0]  seg,
    output [3:0]  an,
    output        dp,
    input         clk,
    input         rst,
    input  [31:0] disp_data
    );
    wire clkd;
    wire [7:0] hex_num;

    /* Divides the clock */
    clk_gen_SEG clk_gen_SEG_inst(.clk_div(clkd), .clk(clk),
.rst(btnC));

    /* Selects the digit */
    digit_selector digit_selector(.digit_sel(an), .clk(clkd),
.rst(btnC));

    /* Decided which switches to listen to */
    hex_num_gen hex_num_gen(.hex_num(hex_num), .digit_sel(an),
.sw(disp_data));

    /* The display */
    sseg_display sseg(.seg(seg), .dp(), .sw(hex_num));

    assign dp = (an == 4'b0111)?1'b0:1'b1;
```

```
endmodule
```

```
module clk_gen_SEG(
    output clk_div,
    input  clk,
    input  rst
    );
    reg [25:0] cntr;            // Holds the 26-bit count
    assign clk_div = cntr[18];  // bit 18 is 190 Hz, bit 17 is 
380 Hz



    /* Implements the 26-bit counter */
    always @ (posedge clk, posedge rst)
        begin
            if (rst)
                cntr <= 26'b0;          // Resets the counter
            else if (clk)
                cntr <= cntr + 1;  // Increment the counter each 
clock tick
        end

endmodule
```

```
    /* Determines which digit is lit up */
    always @ (posedge clk, posedge rst)
    begin
         if (rst)
             digit_sel <= 4'b1110;  // Resets to the least
significant digit
         else
             case (digit_sel)
                 4'b1110:
                        digit_sel <= 4'b1101;
                 4'b1101:
                        digit_sel <= 4'b1011;
                 4'b1011:
                        digit_sel <= 4'b0111;
                 4'b0111:
                        digit_sel <= 4'b1110;
                 default:
                        digit_sel <= 4'b1110;
             endcase
     end

endmodule
```

```
/* Generates the number that syncs the switches with the correct
digit */

module hex_num_gen(

    output reg [7:0] hex_num,

    input  [3:0] digit_sel,

    input [31:0] sw

    );


    /* Generates the number that syncs the switches with the
correct digit */

    always @ (*)

        begin

            case (digit_sel)

                4'b1110: hex_num = sw[7:0];    // Sets the first
digit

                4'b1101: hex_num = sw[15:8];    // Sets the
second digit

                4'b1011: hex_num = sw[23:16];   // Sets the
third digit

                4'b0111: hex_num = sw[31:24];  // Sets the
fourth digit

                default: hex_num = 8'b00000001;    // Turns it
off

            endcase

        end


endmodule
```

```
module sseg_display(
    output reg [6:0] seg,
    output dp,
    input [7:0] sw
    );
    assign dp = 1'b1;     // Force the decimal point off

    /* Change the number on the display each time a different
       sequence of switches are flipped */
    always @ (sw)
        begin
            if(sw == 8'h00)       seg = 7'b1000000; //0
            else if (sw == 8'h01) seg = 7'b1111001; //1
            else if (sw == 8'h02) seg = 7'b0100100; //2
            else if (sw == 8'h03) seg = 7'b0110000; //3
            else if (sw == 8'h04) seg = 7'b0011001; //4
            else if (sw == 8'h05) seg = 7'b0010010; //5
            else if (sw == 8'h06) seg = 7'b0000010; //6
            else if (sw == 8'h07) seg = 7'b1111000; //7
            else if (sw == 8'h08) seg = 7'b0000000; //8
            else if (sw == 8'h09) seg = 7'b0010000; //9
            else if (sw == 8'h0a) seg = 7'b0001000; //A
            else if (sw == 8'h0b) seg = 7'b0000011; //b
            else if (sw == 8'h0c) seg = 7'b1000110; //C
            else if (sw == 8'h0d) seg = 7'b0100001; //d
            else if (sw == 8'h0e) seg = 7'b0000110; //E
            else if (sw == 8'h0f) seg = 7'b0001110; //F
            else if (sw == 8'h10) seg = 7'b0001001; //H
            else if (sw == 8'h18) seg = 7'b0001100; //P
            else if (sw == 8'h1d) seg = 7'b1000001; //U
            else if (sw == 8'hff) seg = 7'b1111111;// blank
            else if (sw == 8'hf0) seg = 7'b1001100; // first
part m
            else if (sw == 8'hf1) seg = 7'b1011000; // last half
            else seg = 7'b1111111;

//          //H20 //mad // f0f10a0d
//          else if (sw == 8'h10) seg = 7'b0001001;//H
//          else if (sw == 4'h2) seg = 7'b0100100; //2
//          if(sw == 4'h0)       seg = 7'b1000000; //0

//          //COO1
//          else if (sw == 4'hc) seg = 7'b1000110; //C
//          if(sw == 4'h0)       seg = 7'b1000000; //0
//          if(sw == 4'h0)       seg = 7'b1000000; //0
//          else if (sw == 4'h1) seg = 7'b1111001; //1

//          //PEE
//          else if (sw == 8'h18) seg = 7'b0001100;//P 190e0e
//          else if (sw == 4'he) seg = 7'b0000110; //E
//          else if (sw == 4'he) seg = 7'b0000110; //E

//          //bEEF
//          else if (sw == 4'hb) seg = 7'b0000011; //b
//          else if (sw == 4'he) seg = 7'b0000110; //E
//          else if (sw == 4'he) seg = 7'b0000110; //E

//          else if (sw == 4'hf) seg = 7'b0001110; //F //
0b0e0e0f


//          //91U9 glug

//          else if (sw == 8'h09) seg = 7'b0010000; //9

//          else if (sw == 8'h01) seg = 7'b1111001; //1

//          else if (sw == 8'h1d) seg = 7'b1000001; //U

//          else if (sw == 8'h09) seg = 7'b0010000; //9


//          //dEAd

//          else if (sw == 8'h0d) seg = 7'b0100001; //d

//          else if (sw == 8'h0e) seg = 7'b0000110; //E

//          else if (sw == 8'h0a) seg = 7'b0001000; //A

//          else if (sw == 8'h0d) seg = 7'b0100001; //d

        end


    endmodule
```

Description of the Project:

This project has five states that indicate when a ‘cow’ should drink water and expel the water. The drinking is controlled by the potentiometer and the current state is indicated by the offboard red, yellow, and green LEDs. The state is also indicated by the seven-segment display. The analog input is displayed by the on-board LEDs for testing as sometimes the potentiometer is a little finicky.

First Peer Review:

Jared suggested taking our original reminder to drink water for a human and turn it into a cow. The project was much more interesting this way, so we changed it.

Michael suggested that we make our timer module accommodate all our timers into one always block. This added an interesting addition to our finite state machine.

Second Peer Review:

We didn’t remember to do an official second peer review, but we did have quite a few peers look over our project, like Val and Jerad. And they were very positive and supportive.

Conclusion Statement:

The finite state machine, the seven segment display, and the off-board LEDs had 100% functionality. Incorporating Brother Jack’s modified ADC code proved difficult. We got it to run but it would occasionally have errors due to how our timers were set up. The seven-segment display would sometimes display values that we did not write. We also should have included a default value for our seven-segment display, but we ran out of time.

Other than that one error our code functioned as expected so in total we had about 99% functionality. Our Video demonstration does not include the potentiometer in it because we ran out of time and we had already demonstrated it to Brother Jack.

Because our output logic relies on the next state instead of the current state the finite state machine is Mealy than Moore because the outputs do not wait for the clock cycle. This was an error born from misunderstanding lab 7 and the fact that the pulses happened a clock cycle too late, and the states would change unexpectedly on us. Brother Jack told us we could simulate it and identify the error and that would probably fix the seven-segment display but unfortunately due to time constraints we never got to it.