Introduction

In this lab, an MMCM was used to generate a 10 MHz clock with a synchronized locked output to create an active low reset_n signal. Then, an ALS light sensor interface module was created to implement a state machine that controls the light sensor sampling. This module also implements an SPI interface to interface with the light sensor, using a shift register to read and store the sensor value. Finally, the sensor value was displayed in hexadecimal on the seven segment display.

Solution Overview

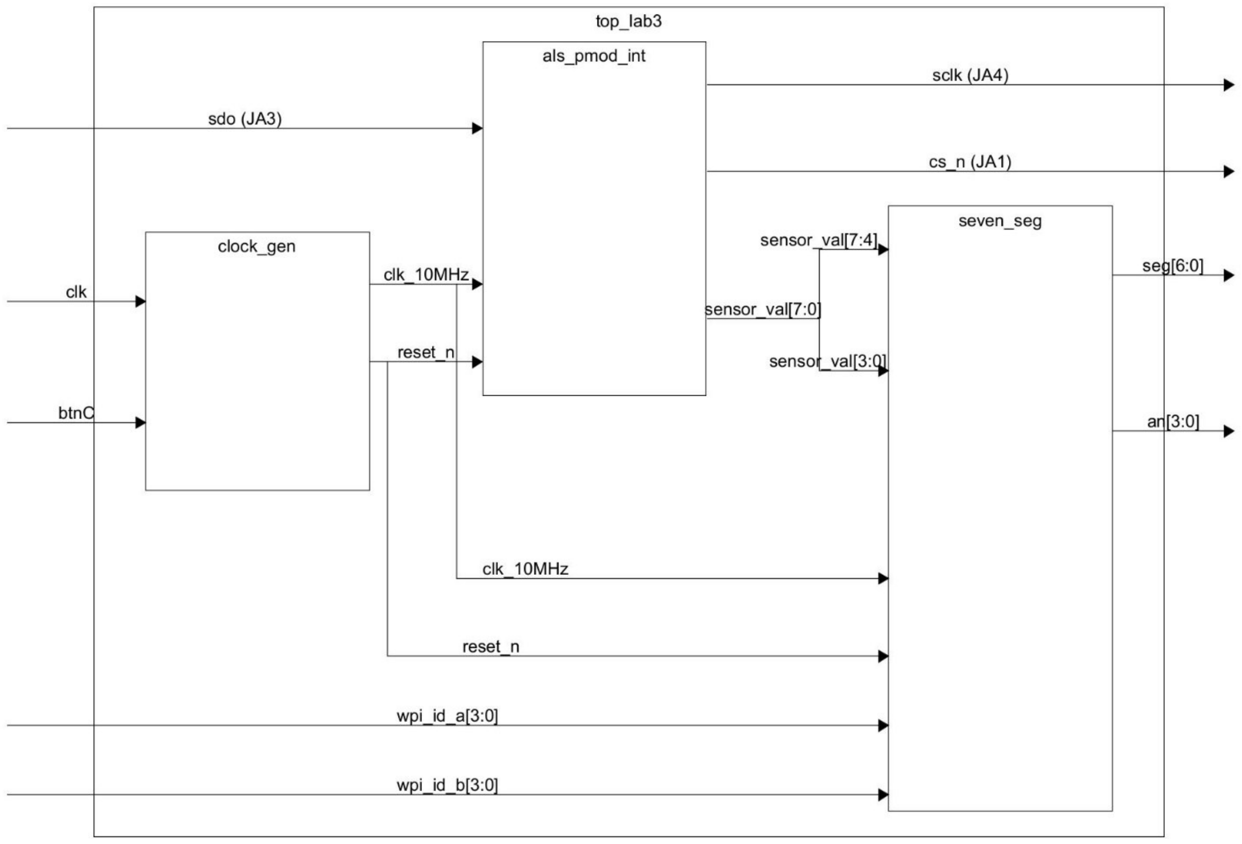
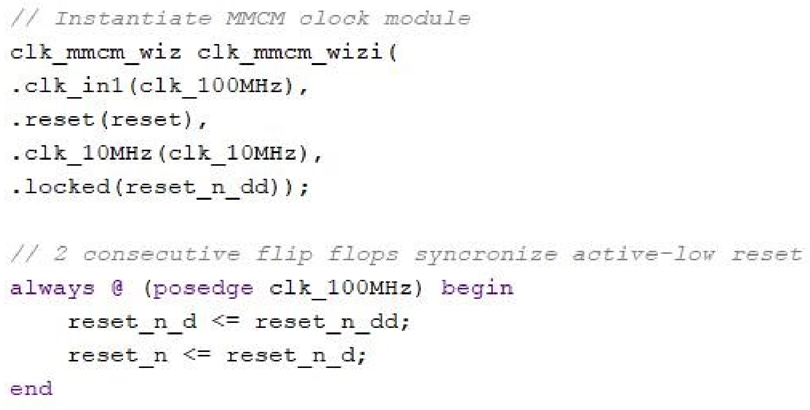


Figure 1: Block diagram of lab 3 with labeled module and signal names

The clock_gen module instantiates the MMCM clock module which produces a 10 MHz clock signal and synchronization logic for the system's reset_n signal. The als_pmod_int module captures a new light sensor value once per second by generating its inputs, sclk and cs_n, to read its output, sdo. Then, the seven segment display shows the compiled light sensor value on the 2 right most displays and the last 2 digits of my WPI ID on displays A and B.

Clock Generation Module



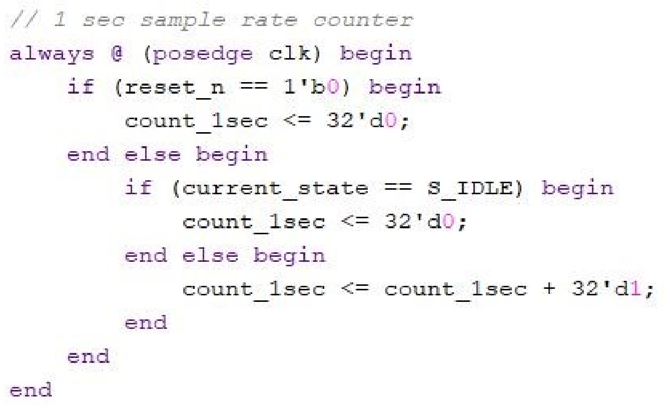
```
// Instantiate MMCM clock module
clk_mmcm_wiz clk_mmcm_wizi(
.clk_in1(clk_100MHz),
.reset(reset),
.clk_10MHz(clk_10MHz),
.locked(reset_n_dd));

// 2 consecutive flip flops syncronize active-low reset
always @ (posedge clk_100MHz) begin
    reset_n_d <= reset_n_dd;
    reset_n <= reset_n_d;
end
```

Figure 2: MCMM clock module generation and locked output synchronization

The counter tutorial was used to generate an MMCM clock module that outputs a 10 MHz clock and a locked output signal. This module instantiates the MMCM clock module and adds synchronization logic for the systems reset_n signal. The locked output is synchronized by putting it through 2 consecutive flip-flops without a reset, making the chances of metastability very small.

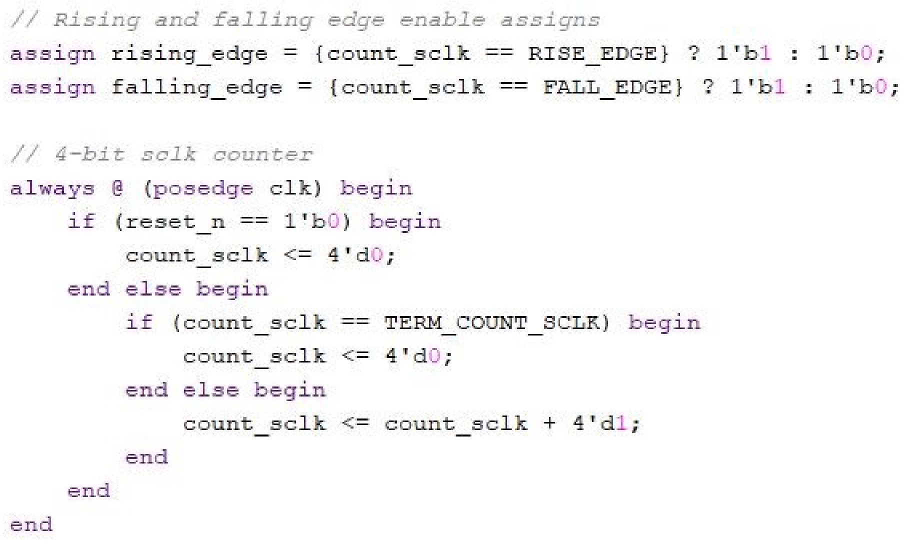
Sensor Interface and Control Module



```
// 1 sec sample rate counter
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        count_1sec <= 32'd0;
    end else begin
        if (current_state == S_IDLE) begin
            count_1sec <= 32'd0;
        end else begin
            count_1sec <= count_1sec + 32'd1;
        end
    end
end
```

Figure 2: 1 second counter for sample rate emulation

In the ALS interface and control module, a counter is used to emulate the sampling rate of 1 second while in the wait state. This is accomplished by setting a 32-bit wide port parameter called SAMPLE_RATE equal to 10000000, since a 10 MHz clock needs to count this many times to reach 1 second.

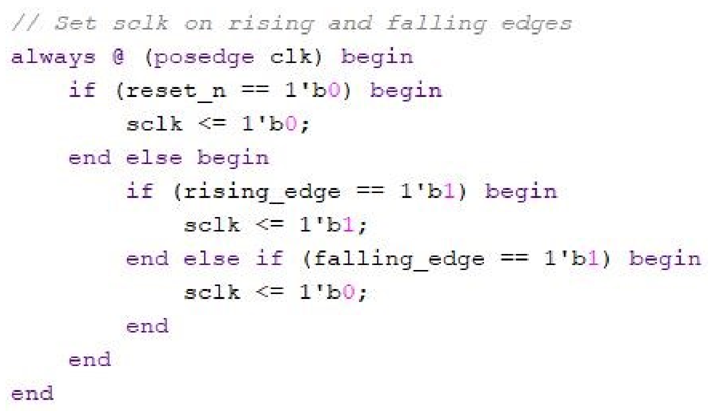


```
// Rising and falling edge enable assigns
assign rising_edge = {count_sclk == RISE_EDGE} ? 1'b1 : 1'b0;
assign falling_edge = {count_sclk == FALL_EDGE} ? 1'b1 : 1'b0;

// 4-bit sclk counter
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        count_sclk <= 4'd0;
    end else begin
        if (count_sclk == TERM_COUNT_SCLK) begin
            count_sclk <= 4'd0;
        end else begin
            count_sclk <= count_sclk + 4'd1;
        end
    end
end
```

Figure 3: Rising and falling edge enable logic

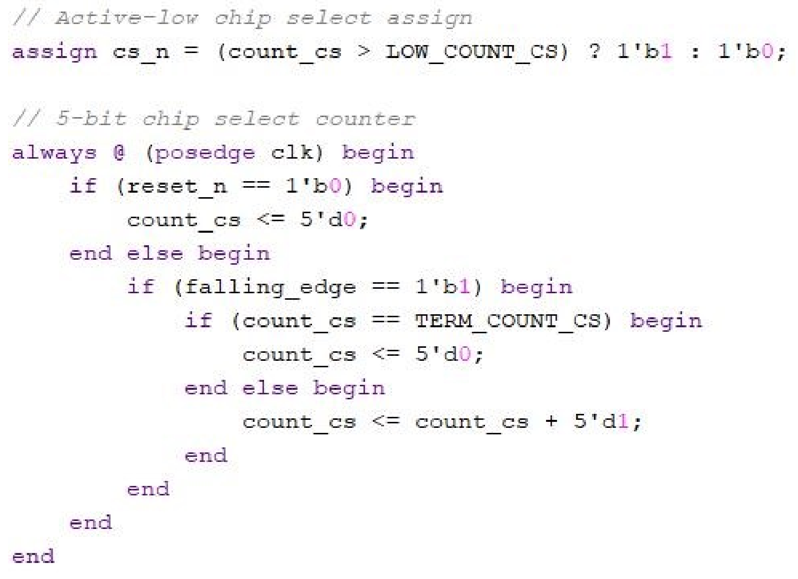
Next, rising edge and falling edge enable signals are generated for use in emulating sclk, cs_n, and reading sdo. This is accomplished by creating a counter to generate a 1 MHz clock, using 9 as the terminal count since a 10 MHz clock input is used. Then, RISE_EDGE and FALL_EDGE parameters are defined as 0 and 5 to track when sclk should turn high and when it should turn low.



```
// Set sclk on rising and falling edges
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        sclk <= 1'b0;
    end else begin
        if (rising_edge == 1'b1) begin
            sclk <= 1'b1;
        end else if (falling_edge == 1'b1) begin
            sclk <= 1'b0;
        end
    end
end
```

Figure 4: Sensor clock signal generation

Then, a 1 MHz sclk signal is generated by checking the rising and falling edge signals described previously. As shown in the figure above, sclk is set to 1 on the rising edge and 0 on the falling edge before being sent to the ALS.

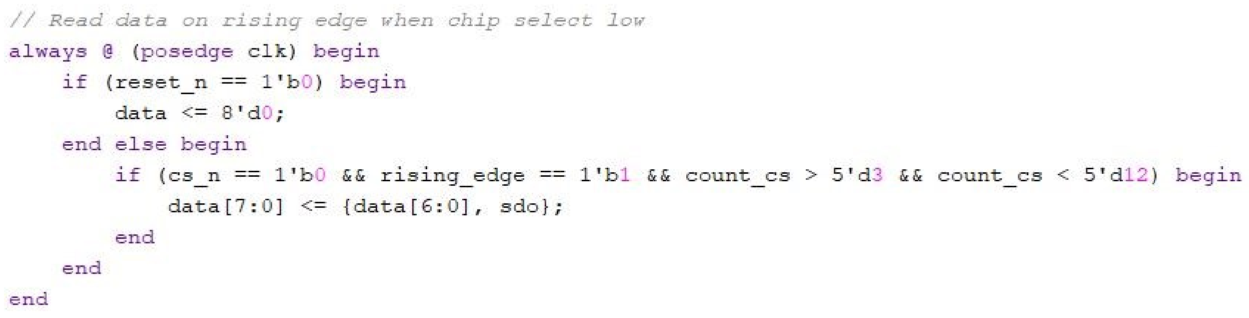


```
// Active-low chip select assign
assign cs_n = (count_cs > LOW_COUNT_CS) ? 1'b1 : 1'b0;

// 5-bit chip select counter
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        count_cs <= 5'd0;
    end else begin
        if (falling_edge == 1'b1) begin
            if (count_cs == TERM_COUNT_CS) begin
                count_cs <= 5'd0;
            end else begin
                count_cs <= count_cs + 5'd1;
            end
        end
    end
end
```

Figure 5: Active-low chip select signal generation

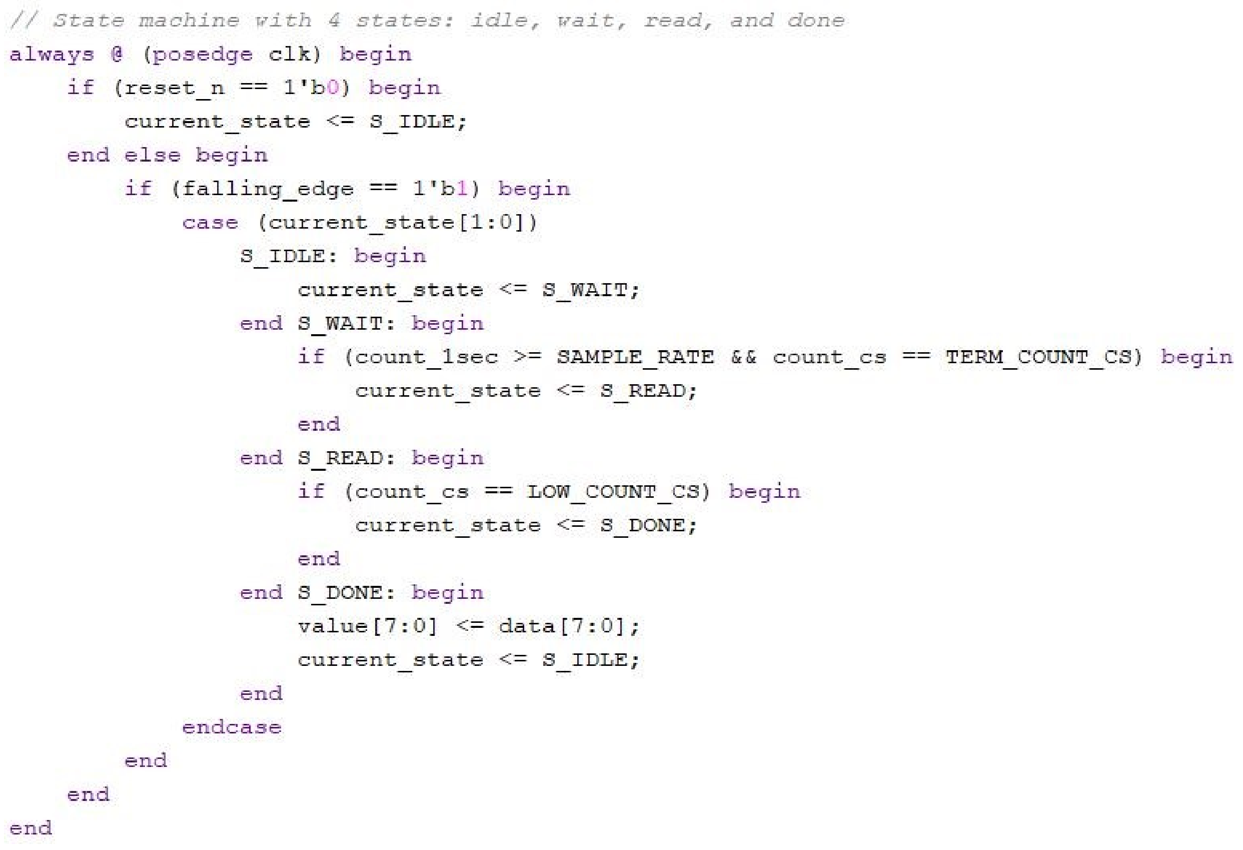
Next, an active-low chip select signal is generated by checking the rising and falling edge enables as well as a cs_count signal. The above assign logic sets cs_n to 1 when cs_count is above 15, the LOW_COUNT_CS parameter, and 0 otherwise. The cs_count signal is only incremented on the falling edge of sclk and has a terminal count of 19, the TERM_COUNT_CS parameter, which means that cs_n is high for 4 sclk cycles and low for 16 sclk cycles.



```
// Read data on rising edge when chip select low
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        data <= 8'd0;
    end else begin
        if (cs_n == 1'b0 && rising_edge == 1'b1 && count_cs > 5'd3 && count_cs < 5'd12) begin
            data[7:0] <= {data[6:0], sdo};
        end
    end
end
```

Figure 6: Logic for reading sdo output of ALS

Then, when data is ready to read, the new read data value is shifted in on the rising edge enable flag. This is accomplished by checking if cs_n is 0, since reading should only be done when the ALS is outputting. Also, data is read on the rising edge by checking that rising_edge is high because sdo is set when falling_edge is high, so this reading ensures that data is not lost. Lastly, data is only read when cs_count is greater than 3 and less than 12 to account for the 3 leading zeros and 4 lagging zeros output by the light sensor.



```
// State machine with 4 states: idle, wait, read, and done
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        current_state <= S_IDLE;
    end else begin
        if (falling_edge == 1'b1) begin
            case (current_state[1:0])
                S_IDLE: begin
                    current_state <= S_WAIT;
                end S_WAIT: begin
                    if (count_1sec >= SAMPLE_RATE && count_cs == TERM_COUNT_CS) begin
                        current_state <= S_READ;
                    end
                end S_READ: begin
                    if (count_cs == LOW_COUNT_CS) begin
                        current_state <= S_DONE;
                    end
                end S_DONE: begin
                    value[7:0] <= data[7:0];
                    current_state <= S_IDLE;
                end
            endcase
        end
    end
end
```

Figure 7: State machine conditions and output code

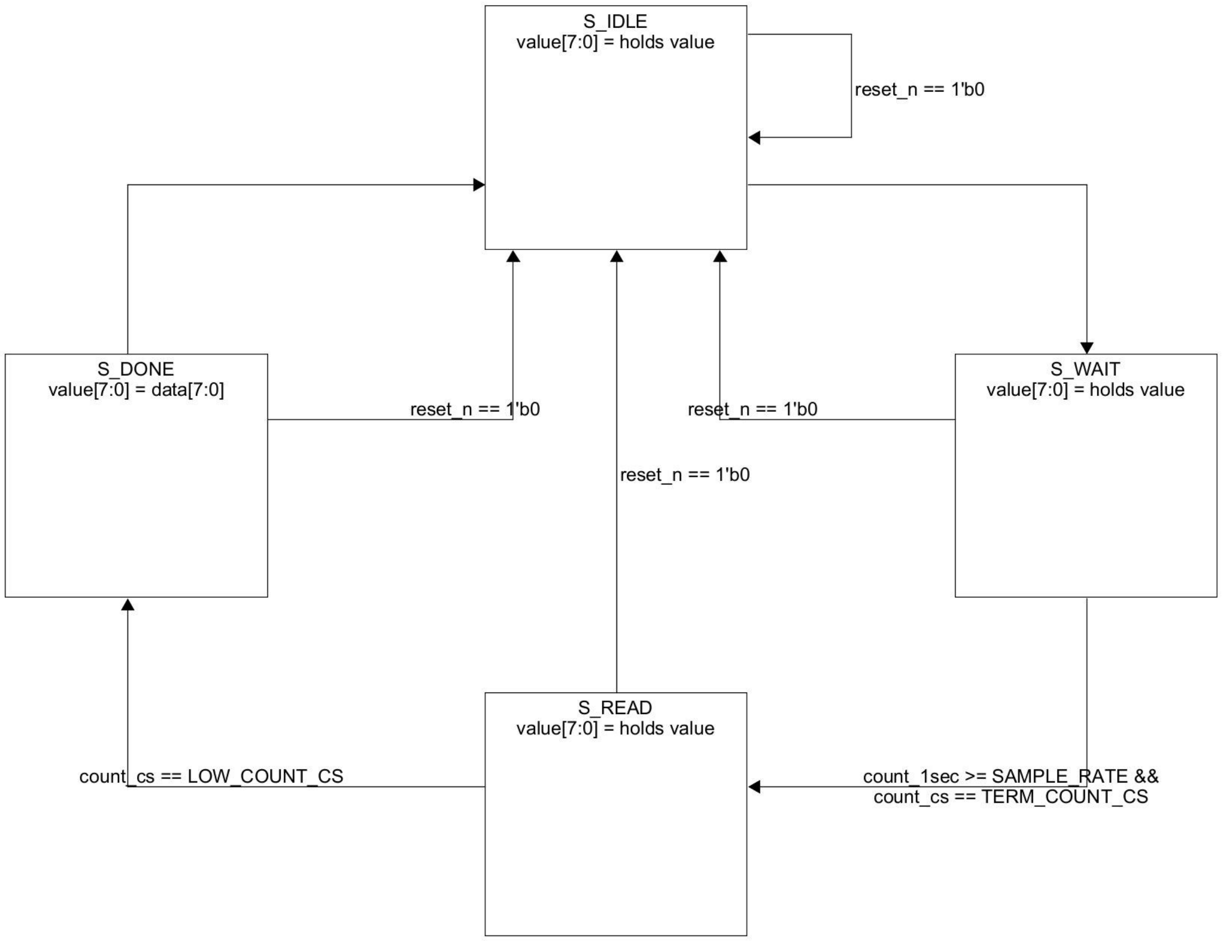
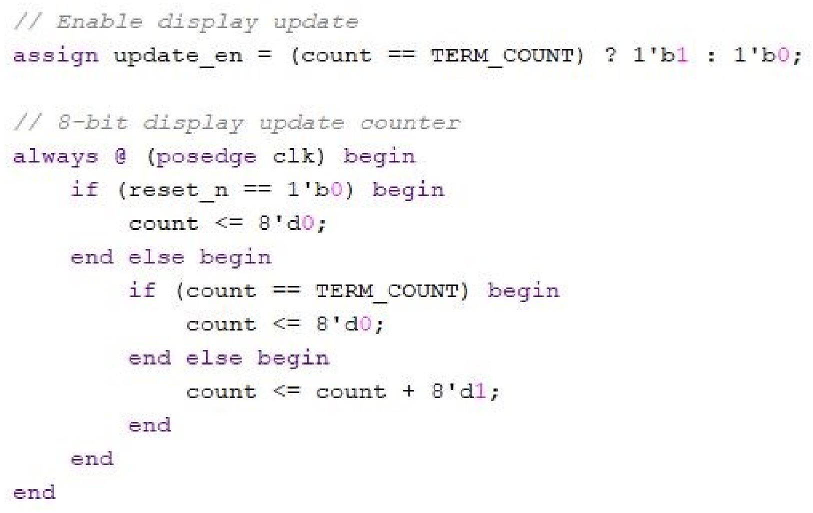


Figure 7: State machine conditions and output diagram

As shown in the figure above, the state machine logic for this system consists of four states: an idle state to initialize the interface on power up, a wait state for 1 sec to control the sampling time, a read light sensor state that triggers a read on the SPI interface, and a done state that drives out the sampled data on value before returning to idle. Regardless of the current state, when reset_n is low, the state is set to idle to make sure that the state is known on reset. When in the idle state, on the next falling edge of sclk, the state is set to wait. In the wait state the 1 second counter described previously will count until it reaches its terminal count, at which point the state will be set to read. In the read state, the chip select counter reaches its low count to make sure all of the data to be read is sampled. When in the done state, the data is stored as the module's output value for the seven segment display before returning to idle when cs_n is set high.

Seven Segment Display Module

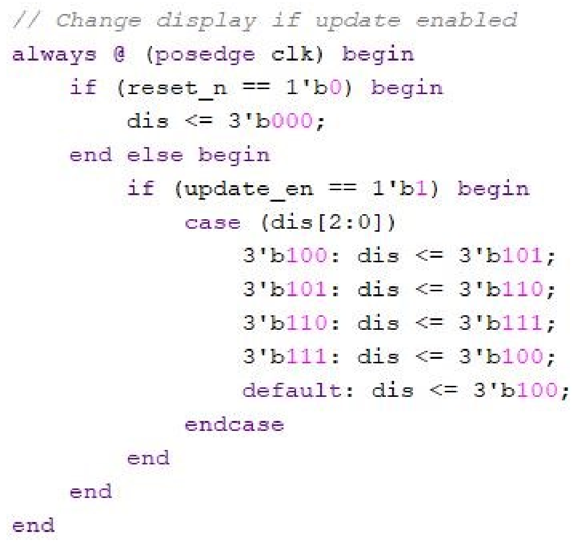


```
// Enable display update
assign update_en = (count == TERM_COUNT) ? 1'b1 : 1'b0;

// 8-bit display update counter
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        count <= 8'd0;
    end else begin
        if (count == TERM_COUNT) begin
            count <= 8'd0;
        end else begin
            count <= count + 8'd1;
        end
    end
end
```

Figure 8: Display update enable and counter

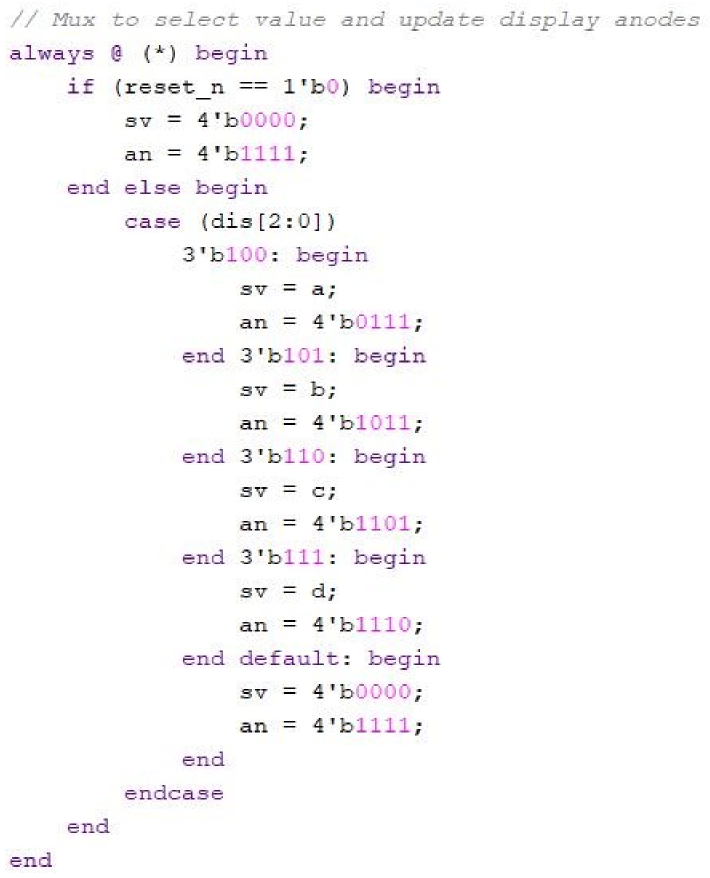
Similar to the previous lab, the seven-segment module counts on the positive edge of the 10 MHz clock input, resetting the count to zero if in reset or if the maximum count has been reached. The update enable signal is set each time the count is equal to the maximum count so that the display can be switched. An 8-bit counter was used to count from 0 to 255 which was shown to light the displays bright enough without flickering.



```
// Change display if update enabled
always @ (posedge clk) begin
    if (reset_n == 1'b0) begin
        dis <= 3'b000;
    end else begin
        if (update_en == 1'b1) begin
            case (dis[2:0])
                3'b100: dis <= 3'b101;
                3'b101: dis <= 3'b110;
                3'b110: dis <= 3'b111;
                3'b111: dis <= 3'b100;
                default: dis <= 3'b100;
            endcase
        end
    end
end
```

Figure 9: Changing display signal if update enabled

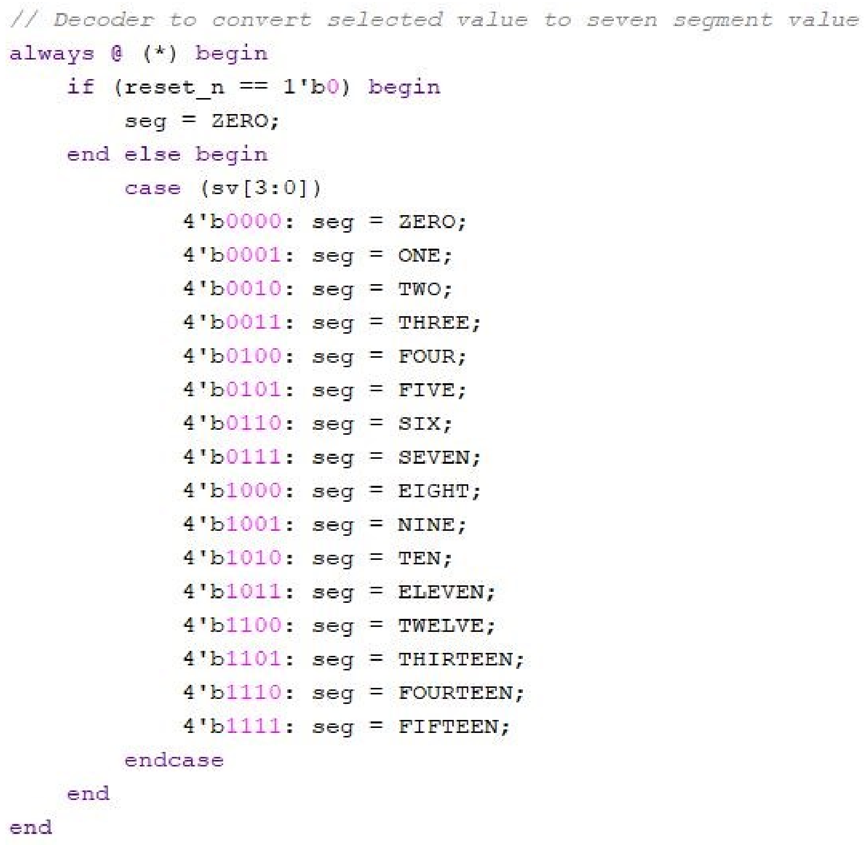
Next, if the update enable signal is set, the display signal will be set to its next state. For example, if the current display signal is 3’b100 for display A, the next display to be shown is B, so the display signal will be set to 3’b101.



```
// Mux to select value and update display anodes
always @ (*) begin
    if (reset_n == 1'b0) begin
        sv = 4'b0000;
        an = 4'b1111;
    end else begin
        case (dis[2:0])
            3'b100: begin
                sv = a;
                an = 4'b0111;
            end 3'b101: begin
                sv = b;
                an = 4'b1011;
            end 3'b110: begin
                sv = c;
                an = 4'b1101;
            end 3'b111: begin
                sv = d;
                an = 4'b1110;
            end default: begin
                sv = 4'b0000;
                an = 4'b1111;
            end
        endcase
    end
end
```

Figure 10: Mux for selecting which display is shown

Next, the display signals are mapped to their respective segment value and anode signals. The segment values for displays A and B are hardcoded the last two digits of my WPI ID while the values for C and D are received from the sensor interface and control module. Also, an active-low asynchronous reset is used to shut the displays off.

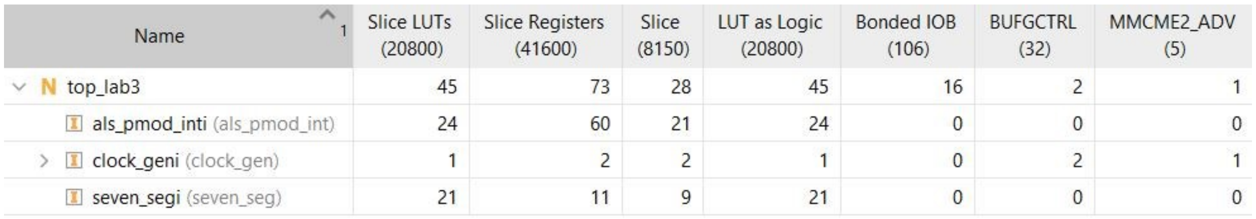


```
// Decoder to convert selected value to seven segment value
always @ (*) begin
    if (reset_n == 1'b0) begin
        seg = ZERO;
    end else begin
        case (sv[3:0])
            4'b0000: seg = ZERO;
            4'b0001: seg = ONE;
            4'b0010: seg = TWO;
            4'b0011: seg = THREE;
            4'b0100: seg = FOUR;
            4'b0101: seg = FIVE;
            4'b0110: seg = SIX;
            4'b0111: seg = SEVEN;
            4'b1000: seg = EIGHT;
            4'b1001: seg = NINE;
            4'b1010: seg = TEN;
            4'b1011: seg = ELEVEN;
            4'b1100: seg = TWELVE;
            4'b1101: seg = THIRTEEN;
            4'b1110: seg = FOURTEEN;
            4'b1111: seg = FIFTEEN;
        endcase
    end
end
```

Figure 11: Decoder to convert hex input into seven segment values

Lastly, a decoder is used to set the real 7-bit segment value output based on the 4-bit value that was set inside of the mux previously. Parameters were used to define the real segment values as ZERO through FIFTEEN.

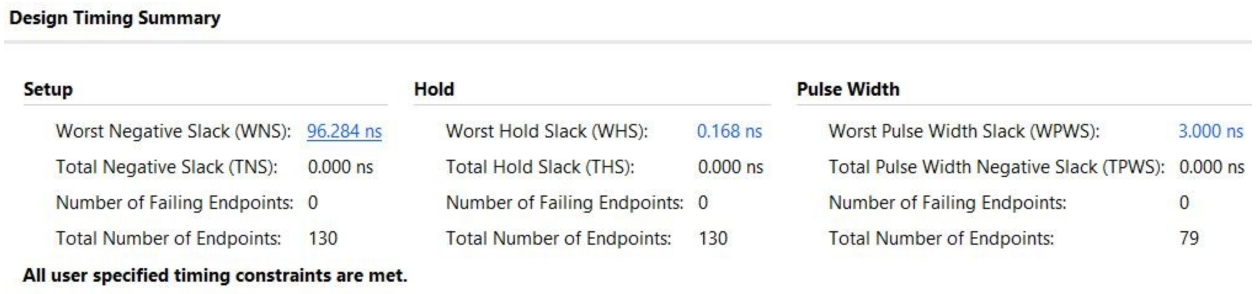
Implementation Summary



Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)	MMCME2_ADV (5)
top_lab3	45	73	28	45	16	2	1
als_pmod_inti (als_pmod_int)	24	60	21	24	0	0	0
clock_geni (clock_gen)	1	2	2	1	0	2	1
seven_segi (seven_seg)	21	11	9	21	0	0	0

Figure 12: Utilization report per module

After generating the above utilization report by following the counter tutorial, it can be seen that this design utilizes 45 LUTs and 73 registers in total. The sensor interface and control module uses only 24 LUTs but 60 registers, many of which store the sampling rate counter. The other registers in this module are used to store the counts for chip select and sclk, as well as the current state and data registers. Also, this module uses 24 LUTs which go towards the functions for generating each type of count, rising and falling edge signals, chip select signal, and current state signal. The clock generation module uses only 1 LUT for the flip-flop function and only 2 registers for storing the reset signal and the reset signal delayed by 1 clock cycle. The seven segment module uses 21 LUTs and 11 registers, some of which store the selected display, its value, and the update count. The 21 LUTs used by this module are mostly to drive the functions for setting the anode and segment signals, since these have the widest output signals.



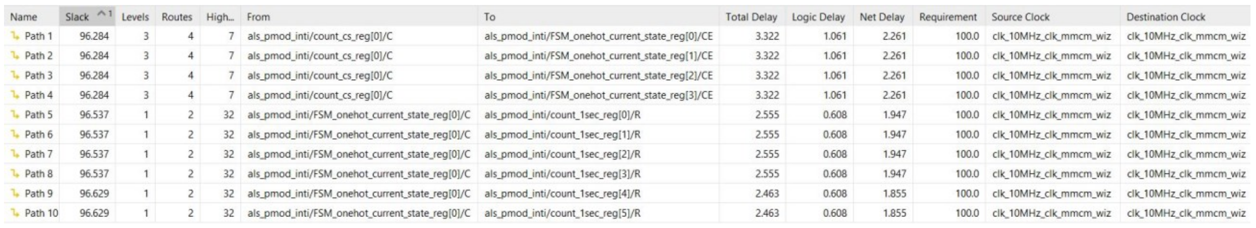
Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	96.284 ns	Worst Hold Slack (WHS):	0.168 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	130	Total Number of Endpoints:	130	Total Number of Endpoints:	79

All user specified timing constraints are met.

Figure 13: Timing summary report

As shown above in the timing summary for this design, in terms of worst case slack, the net that has the least margin, the worst negative slack was 96.284 ns, the worst hold slack was 0.168 ns, and the worst pulse width slack was 3.000 ns. In terms of total slack, the sum of all the failing nets in this design, the total negative slack, total hold slack, and total pulse width negative slack were all 0.000 ns.



Name	Slack ^1	Levels	Routes	High...	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	96.284	3	4	7	als_pmod_inti/count_cs_reg[0]/C	als_pmod_inti/FSM_onehot_current_state_reg[0]/CE	3.322	1.061	2.261	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 2	96.284	3	4	7	als_pmod_inti/count_cs_reg[0]/C	als_pmod_inti/FSM_onehot_current_state_reg[1]/CE	3.322	1.061	2.261	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 3	96.284	3	4	7	als_pmod_inti/count_cs_reg[0]/C	als_pmod_inti/FSM_onehot_current_state_reg[2]/CE	3.322	1.061	2.261	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 4	96.284	3	4	7	als_pmod_inti/count_cs_reg[0]/C	als_pmod_inti/FSM_onehot_current_state_reg[3]/CE	3.322	1.061	2.261	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 5	96.537	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[0]/R	2.555	0.608	1.947	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 6	96.537	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[1]/R	2.555	0.608	1.947	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 7	96.537	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[2]/R	2.555	0.608	1.947	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 8	96.537	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[3]/R	2.555	0.608	1.947	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 9	96.629	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[4]/R	2.463	0.608	1.855	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz
Path 10	96.629	1	2	32	als_pmod_inti/FSM_onehot_current_state_reg[0]/C	als_pmod_inti/count_1sec_reg[5]/R	2.463	0.608	1.855	100.0	clk_10MHz_clk_mmcm_wiz	clk_10MHz_clk_mmcm_wiz

Figure 14: Intra-clock paths report

As shown in the intra-clock paths report, the signal paths that took the most time were the count_cs to current_state paths which accumulated slack of 96.284 ns. Timing can be improved on these nets by using one-hot encoding for the states, using a synthesis state machine coding tool, reducing the number in input signals to functions, registering input and output signals, or pre-decoding counter values.

Conclusion

Challenges faced in implementation included missing the locked output of the MMCM clock module inside of the flip-flops in clock_gen. Since the reset button stops the 10 MHz MMCM clock output from oscillating, it cannot be used in the sensitivity list of the flip-flop block, or else the reset signal will not be properly. Also, generating the chip select signal was difficult for me to wrap my head around at first, but I learned that it can be generated by using a simple counter alongside the rising and falling edge signals for sclk. Similarly, I learned that reading can be enabled by using the cs_n directly and checking for the rising edge signal to go high. For further improvement of this lab experience, I believe it should be made clear that the top row of pins in JA correspond to the correct pin connections on the ALS. I enjoyed writing the state machine for this lab, as I was able to step through the actual control of the system.