Design Project - Discrete ADC

Hamza Amar, Santiago Fuentes, Rhenaldi Wijaya; Group 14

Contents

I. Introduction

The goal of the design project was to test all the FPGA design topics seen in class, such as, Analog-to-Digital converters (ADCs), PWM systems, as well, as R2R circuits with hierarchical design in mind using the Digilent Basys3 FPGA board.

A. Objectives

The main objectives of the final project which entail around a 95% tier in the project:

- Create a menu and display system to change the values printed in the 7-segment display in the board.
- Reference the XADC subsystem that displays average and scaled values.
- Use a discrete PWM ramp and R2R ADC subsystem.
- Design and create a Successive Approximation Algorithm (SAR) for both the PWM and R2R circuits as an FSM.
- Final RTL design with a max to clock frequency of 200 MHZ while meeting timing.

II. System architecture

Our system is based on the required hierarchical organization common and mostly necessary in FPGA design.

We divided our Top Module into multiple submodules which are the XADC, PWM, R2R subsystems each of these encompass the required components, combinational logic, and connections, that make these subsystems work. Furthermore, we have the 7-segment display modules, as well as, the MUX4 and DP MUX, which let the user select between the different outputs each of the subsystems print in the display, see Figure 1.

The circuit design is divided into the PWM comparator circuit and the R2R ladder circuit which its main requirement is to compare the data signals coming from the Basys3 board that get displayed into values displayed based on the changes of the voltage due to the potentiometer see Figure 2.

The design allows the user to select into the following different outputs values.

III. Detail design

A. XADC subsystem

This module serves as a measurement tool in the XADC IP code using channel 15 in the PMOD pins of N2/N1 in this design, it operates in continuous sampling mode.

The raw output undergoes an additional averaging using a custom module extending the resolution of the output to 16 bits. Furthermore, we decided to use the following formula

$$\text{mV} = (\text{ave\_data} \cdot 3300) \gg 16 \tag{1}$$

with scaling factor of 16 to scale the RAW value to decimal volts value. A rising edge signal is used to enable the averager correctly. It has hexadecimal, scaled and averaged outputs thanks to the multiplexer.

B. PWM Ramp ADC

The PWM ramp ADC adds an 8-bit analog-to-digital conversion using a pulse-with modulation module in complementation with the Sawtooth generator that creates a ramping duty cycle from 0 to 255 shown in the oscilloscope. The external comparator compares the input voltage versus the ramp voltage using an edge detector to indicate the moment the ramp crosses the input, allowing the capture of the raw ADC result.

With averaging, the effective update rate is approximately 1.56 updates per second.

C. R2R Ladder ADC

Working similarly to the PWM module, the R2R as it name suggests it uses a resistor ladder for the digital-to-analog conversion. The same sawtooth module generates the 8-bit counter values that driver the external resistor. Compared to the PWM it has faster settling times, and it can work without the low pass filter.

The conversion principle remains identical to the PWM implementation: the comparator falling edge triggers capture of the current counter value.

D. Successive Approximation Algorithm (SAR)

The SAR ADC algorithm is implemented as a replacement of the ramp method by performing a binary search across the 8-bit range generated using fewer comparisons to complete a full conversion.

The seven-state FSM, see Figure 4, is used to represent the algorithm with the following states (IDLE, INIT, WAIT_SETTLE, COMPARE, DECIDE, NEXT_BIT, DONE).

Starting from the MSB, each bit is set to 1, and comparing the results after waiting for the PWM and the R2R circuits. If the voltage input is larger than the voltage of the DAC the bit is clear if not it remains. After all the bit testing is done, the final result is output with a conversion pulse.

The SAR, also has a switch detection logic to compare the two modes, with and without the SAR, as well as having auto-restart logic for continuous sampling.

IV. Implementation

The design created had a successful implementation with a positive WNS of 3.159, see Figure 3, of in the final synthesis with the higher input frequency setting than normal of 200 MHz.

Given a user clock period of $T_{\text{user}} = 10.00\,\text{ns}$ and the worst negative slack of $\text{WNS} = 3.159\,\text{ns}$ as previously mentioned, the achievable period is:

$$T_{\text{achievable}} = T_{\text{user}} - \text{WNS} = 10.00 - 3.159 = 6.841\ \text{ns}. \tag{2}$$

Thus, the maximum clock frequency is:

$$f_{\max}(\text{MHz}) = \frac{1000}{T_{\text{achievable}}(\text{ns})} = \frac{1000}{6.841} \approx 146.18\ \text{MHz}. \tag{3}$$

V. Testing and Verification

For the testing and validation of the FPGA design and comparator circuit. Mostly in terms of creating the final breadboard circuit we tested the connections by using the oscilloscope and the multimeter by analyzing graph patterns, reading voltage signals and measuring current.

In contrast, the way we ended up testing the RTL code against, we used a more crude organic approach, instead of using test bench

files or something similar we used real time testing by making changes in the RTL code and synthesizing right way this was possible to do to the great performance of Vivado in a Fedora Linux machine, as well as, performance settings, this greatly reduced the generation to less than 10 minutes.

This type of testing was only possible due to the specific circumstances of the design, this approach should be reevaluated in future work closer to actual industry practices.

VI. AI Use

The AI tools in the project used were Claude and ChatGPT.

For the design portion of the RTL code and the circuit AI was used in help with debugging, clarifying questions, and creating some structure of the modules not provided. All code generated was tested, fixed and verified it functionality before committing to it.

Although, it was really useful to explain and clarifying topics, in terms of implementations of RTL code it was more lackluster, requiring reviewing each code line manually making sure its functionality.

Furthermore, in terms of writing it was used for outline and explanations, as well as, fixing grammar, all writing was creating by hard and verified.

VII. Lessons learned

The main things to take about this project is the importance of the hierarchical design, that used in different coding practices, is fundamental for HDLs.

Our approach of starting the RTL design from the top module down into other modules was the correct design since it made the code more organized, easier to debug, and maintain.

In the other hand, something to think retrospectively is our incorrect approach in testing by utilizing actual proper industry practices by using test benches and other tools, our development times, errors and mistakes could have been mitigated.

VIII. Conclusion

In summary, our team was able to complete the final project design up to the 100% eligible bracket by finishing all the required modules of R2R ladder and PWM including the SAR algorithm, with the correct output values in the R2R and PWM ramp modes, as well as, correct comparisons using the SAR algorithm. Furthermore, our design optimization and efficiency was able to allow higher than normal speeds with still a positive WNS.

Appendix A
Team contributions

A. Amar, Hamza

This team member worked mostly on the design & organization of the project in Vivado, RTL code creation and verification, as well as, code testing, debugging.

B. Fuentes, Santiago

This team member did RTL code testing and verification, as well as, writing, formatting and finishing the final report of the project.

C. Wijaya, Rhenaldi

This team member did RTL code creation and verification, code testing and debugging, as well as, design and testing the breadboard prototype circuit.

Appendix B
Figures & Full RTL Code

A. Figures

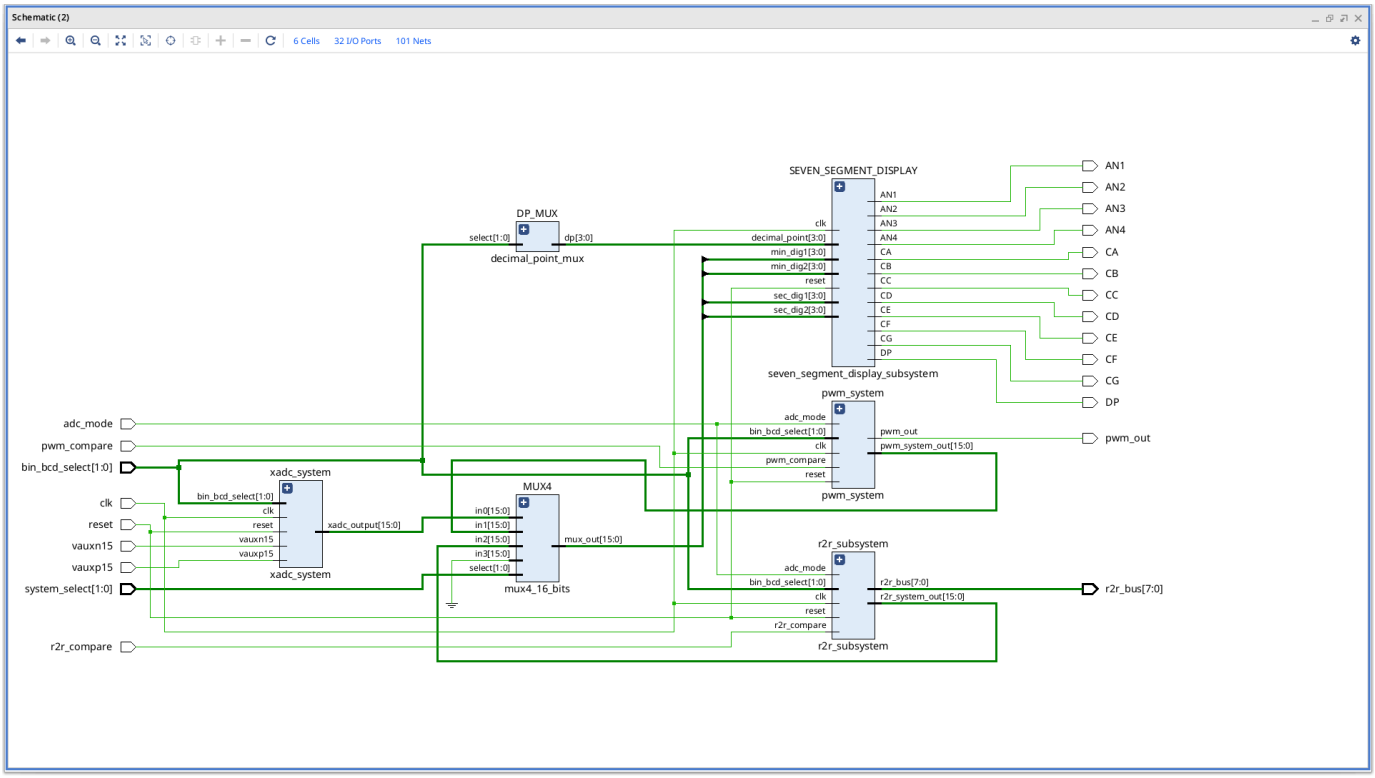
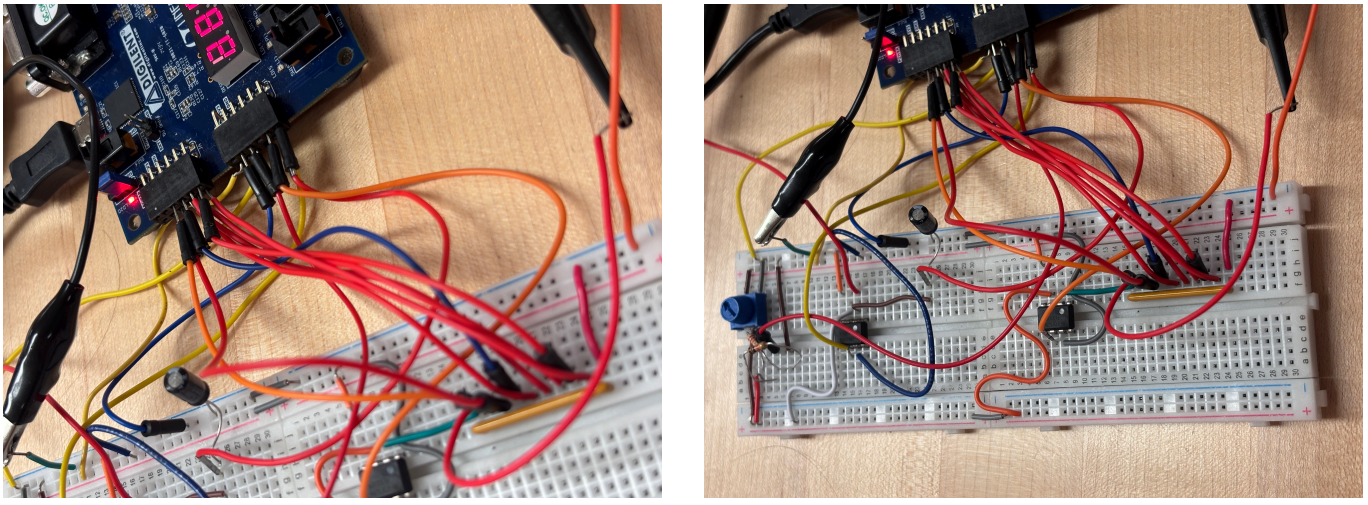


Fig. 1. Top-level RTL schematic.



(a) Connections close-up

(b) Breadboard overview

Fig. 2. Prototype wiring and breadboard implementation.

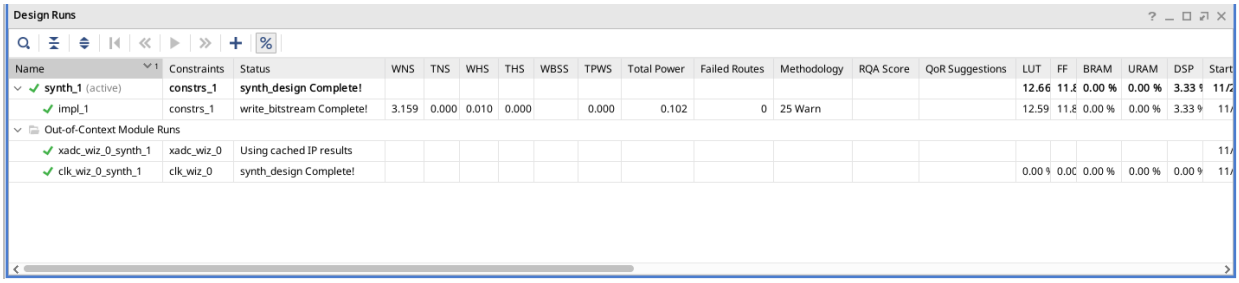


Fig. 3. Vivado timing summary (WNS) table.

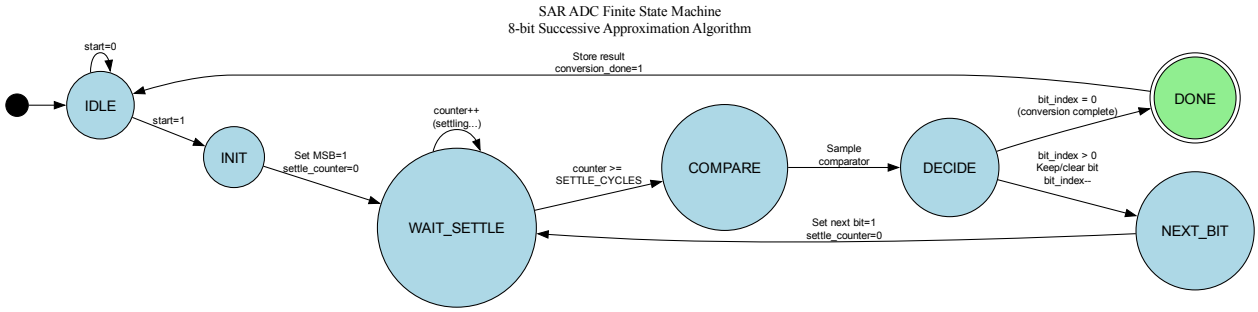


Fig. 4. SAR ADC finite state machine (FSM) diagram.

B. SystemVerilog (RTL)

Listing 1. abc_subsystem.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/15/2025 03:26:06 PM
// Design Name:
// Module Name: adc_subsystem


module adc_subsystem (
    input logic clk,
    input logic reset,
    input vauxp15, // Analog input (positive) - connect to JXAC4:N2 PMOD pin (XADC4)
    input vauxn15, // Analog input (negative) - connect to JXAC10:N1 PMOD pin (XADC4)
    output logic [15:0] data_out,
    output logic [15:0] scaled_adc_data,
    output logic [15:0] ave_data
    );
    logic ready; // Data ready from XADC
    logic ready_r, ready_pulse;
    logic [31:0] scaled_adc_data_temp; // Scaled ADC data for display
    logic [6:0] daddr_in; // XADC address
    logic enable; // XADC enable
```

```
    logic eos_out; // End of sequence
    logic busy_out; // XADC busy signal

    // Constants
    localparam CHANNEL_ADDR = 7'h1f; // XA4/AD15 (for XADC4)
    // XADC Instantiation
    xadc_wiz_0 XADC_INST (
        .di_in(16'h0000), // Not used for reading
        .daddr_in(CHANNEL_ADDR), // Channel address
        .den_in(enable), // Enable signal
        .dwe_in(1'b0), // Not writing, so set to 0
        .drdy_out(ready), // Data ready signal (when high, ADC data is valid)
        .do_out(data_out), // ADC data output
        .dclk_in(clk), // Use system clock
        .reset_in(reset), // Active-high reset
        .vp_in(1'b0), // Not used, leave disconnected
        .vn_in(1'b0), // Not used, leave disconnected
        .vauxp15(vauxp15), // Auxiliary analog input (positive)
        .vauxn15(vauxn15), // Auxiliary analog input (negative)
        .channel_out(), // Current channel being converted
        .eoc_out(enable), // End of conversion
        .alarm_out(), // Not used
        .eos_out(eos_out), // End of sequence
        .busy_out(busy_out) // XADC busy signal
    );
   averager
   #( .power(12), //2**N samples, default is 2**8 = 256 samples
      .N(16) // # of bits to take the average of
    )
   AVERAGER
    ( .reset(reset),
      .clk(clk),
      .EN(ready_pulse),
      .Din(data_out),
      .Q(ave_data)
    );

    always_ff @(posedge clk) begin
        if (reset) begin
            scaled_adc_data <= 0;
            scaled_adc_data_temp <= 0;
        end
        else if (ready_pulse) begin
            // Calculation: This scales FFFFh to 270Fh (i.e. 9999d)
            // mVolts = ave_data/(2^16 - 1) * 9999 = ave_data * 0.152575
            // mVolts ~ ave_data * 1250/2^13 = (ave_data) * 1250 >> 13
            // NOTE: The 7-seg display will display in millivolts,
            // i.e. 9999 is 0.9999 V or 999.9 mV
            // place the decimal point in the correct place!
// scaled_adc_data <= (ave_data*9999) >> 16; // was scaled_adc_data_temp
            scaled_adc_data_temp <= {16'd0, ave_data} * 32'd3300;
            scaled_adc_data <= scaled_adc_data_temp[31:16];
            //scaled_adc_data <= scaled_adc_data_temp; // additional register faciliates pipelining
        end // for higher clock frequencies
    end
        // added these 3 lines for the pulser
      always_ff@(posedge clk)
        if (reset)
            ready_r <= 0;
        else
            ready_r <= ready;

      assign ready_pulse = ~ready_r & ready; // generate 1-clk pulse when ready goes high
```

```
88
89 endmodule
```

Listing 2. averager.sv

```
module averager
    #(parameter int
        power = 8, // 2**N samples, default is 2**8 = 256 samples
        N = 12) // # of bits to take the average of
    (
        input logic clk,
        reset,
        EN,
        input logic [N-1:0] Din, // input to averager
        output logic [N-1:0] Q // N-bit moving average
    );

    logic [N-1:0] REG_ARRAY [2**power:1];
    logic [power+N-1:0] sum;
    assign Q = sum[power+N-1:power];

    always_ff @(posedge clk) begin
        if (reset) begin
            sum <= 0;
            for (int j = 1; j <= 2**power; j++) begin
                REG_ARRAY[j] <= 0;
            end
        end
        else if (EN) begin
            sum <= sum + Din - REG_ARRAY[2**power];
            for (int j = 2**power; j > 1; j--) begin
                REG_ARRAY[j] <= REG_ARRAY[j-1];
            end
            REG_ARRAY[1] <= Din;
        end
    end
endmodule
```

Listing 3. bin_to_bcd.sv

```
// This module was written by Claude 3.5 Sonnet, through several debugging
// iterations with Denis Onen.
// For ENEL 453, you do not have to know the double dabble algorithm used in
// this module. However, you should know how to manually convert binary to
// BCD, and BCD to binary (i.e. on paper).
//////////////////////////////////////////////////////////////////////////////////

// Claude 3rd attempt, works!!
//////////////////////////////////////////////////////////////////////////////////
// Module Name: bin_to_bcd
//
// Description:
// This module converts a 16-bit binary input to a 16-bit BCD (Binary-Coded Decimal) output.
// It uses the Double Dabble algorithm to perform the conversion over 17 clock cycles.
// The module can handle binary inputs up to 9999 (decimal). For inputs greater than 9999,
// it outputs an error code (0xEEEE) to indicate overflow.
//
// Inputs:
// - bin_in : 16-bit binary input to be converted
// - clk : System clock
// - reset : Active-high synchronous reset
//
// Outputs:
// - bcd_out : 16-bit BCD output (4 digits, 4 bits each)
//
// Internal Signals:
// - scratch : 32-bit register used for the Double Dabble algorithm
// - clkcnt : 5-bit counter to track the conversion process (0-17)
// - ready : Indicates when the conversion is complete and output is valid
// - overflow_error: Indicates when the input exceeds 9999
//
// Operation:
// 1. On reset, all registers are cleared and the module is set to ready state.
// 2. When a new binary input is received, the conversion process begins:
// - The input is loaded into the least significant 16 bits of the scratch register.
// - Over the next 16 clock cycles, the Double Dabble algorithm is applied.
// - On the 17th cycle, the final BCD result is latched to the output.
// 3. If the input exceeds 9999, an error code (0xEEEE) is immediately output.
//
// Note: This module requires 17 clock cycles to complete a conversion for valid inputs.
// The 'ready' signal can be used to determine when the output is valid.
//////////////////////////////////////////////////////////////////////////////////
module bin_to_bcd(
    input logic [15:0] bin_in,
    output logic [15:0] bcd_out,
    input logic clk,
    input logic reset
);

    logic [31:0] scratch, next_scratch;
    logic [4:0] clkcnt, next_clkcnt;
    logic ready, next_ready,overflow_error;
    logic [15:0] next_bcd_out;

    always_ff @(posedge clk) begin
        if (reset) begin
            scratch <= '0;
            bcd_out <= '0;
            ready <= 1'b1;
            clkcnt <= '0;
        end else begin
            scratch <= next_scratch;
```

```
            bcd_out <= next_bcd_out;
            ready <= next_ready;
            clkcnt <= next_clkcnt;
        end
    end

    always_comb begin
        next_scratch = scratch;
        next_bcd_out = bcd_out;
        next_ready = ready;
        next_clkcnt = clkcnt;

        if (bin_in > 9999) begin
            next_bcd_out = 16'hEEEE;
            overflow_error = 1;
            next_ready = 1'b0;
            next_clkcnt = '0;
            next_scratch = '0;
        end else begin
            overflow_error = 0;
            case (clkcnt)
                5'd0: begin
                    next_scratch = {16'b0, bin_in};
                    next_clkcnt = clkcnt + 1;
                    next_ready = 1'b0;
                end
                 5'd1, 5'd2, 5'd3, 5'd4, 5'd5, 5'd6, 5'd7, 5'd8, 5'd9, 5'd10, 5'd11, 5'd12, 5'd13, 5'd14, 5'd15, 5'd16: begin
                    // Add 3 to columns >= 5
                    if (next_scratch[31:28] >= 5) next_scratch[31:28] = next_scratch[31:28] + 3;
                    if (next_scratch[27:24] >= 5) next_scratch[27:24] = next_scratch[27:24] + 3;
                    if (next_scratch[23:20] >= 5) next_scratch[23:20] = next_scratch[23:20] + 3;
                    if (next_scratch[19:16] >= 5) next_scratch[19:16] = next_scratch[19:16] + 3;

                    // Shift left
                    next_scratch = {next_scratch[30:0], 1'b0};
                    next_clkcnt = clkcnt + 1;
                end
                5'd17: begin
                    next_bcd_out = next_scratch[31:16];
                    next_ready = 1'b1;
                    next_clkcnt = '0;
                end
                default: begin
                    next_clkcnt = '0;
                end
            endcase
        end
    end

endmodule
```

Listing 4. decimal_point_mux.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name: decimal_point_mux
//
//////////////////////////////////////////////////////////////////////////////////


module decimal_point_mux( input logic [1:0] select, output logic [3:0] dp
    );

    always_comb begin
        case (select)
            2'b00: dp = 4'b0000; // averaged ADC with extra 4 bits
            2'b01: dp = 4'b0010; // averaged & scaled voltage (show DP)
            2'b10: dp = 4'b0000; // raw ADC (12 bits)
            2'b11: dp = 4'b0000;
            default: dp = 16'h0000; // safe default (note: 4 bits, not 16)
        endcase
    end
endmodule
```

Listing 5. digit_multiplexor.sv

```
//----------------------------------------------------------------------------
// Module Name: digit_multiplexor
// Description: This module is a 4-to-1 multiplexer designed to select and
// output one of four 4-bit digit inputs based on a 4-bit selector
// signal. It is typically used in applications where multiple
// digits (such as those representing seconds and minutes) need to
// be displayed sequentially on a single display, such as in a
// timekeeping or stopwatch circuit.
//
// The module accepts four 4-bit inputs corresponding to individual
// digits of time (seconds and minutes), and a 4-bit selector input
// that determines which digit is routed to the output.
//
// The selector input uses a "one-hot" encoding, meaning that only
// one of its bits should be high ('1') at any time to select the
// corresponding digit:
//
// - If `selector` is 4'b0001, `sec_dig1` (seconds digit) is selected.
// - If `selector` is 4'b0010, `sec_dig2` (tens of seconds) is selected.
// - If `selector` is 4'b0100, `min_dig1` (minutes digit) is selected.
// - If `selector` is 4'b1000, `min_dig2` (tens of minutes) is selected.
// - In all other cases, the output `time_digit` is set to 4'b0000.
//
// Inputs:
// - sec_dig1 : 4-bit input representing the seconds digit.
// - sec_dig2 : 4-bit input representing the tens of seconds digit.
// - min_dig1 : 4-bit input representing the minutes digit.
// - min_dig2 : 4-bit input representing the tens of minutes digit.
// - selector : 4-bit one-hot encoded input used to select the digit to output.
//
// Output:
// - time_digit : 4-bit output that carries the selected digit.
//
// Note: This module assumes that the `selector` signal is one-hot encoded. If
// multiple bits in `selector` are high simultaneously, the behavior is
// undefined and will default to outputting 4'b0000.
//----------------------------------------------------------------------------

module digit_multiplexor (
    input logic [3:0] sec_dig1,
    input logic [3:0] sec_dig2,
    input logic [3:0] min_dig1,
    input logic [3:0] min_dig2,
    input logic [3:0] selector,
    input logic [3:0] decimal_point,
    output logic [3:0] time_digit,
    output logic dp_in
);

    always_comb begin
        case (selector)
            4'b0001: time_digit = sec_dig1; // display seconds digit
            4'b0010: time_digit = sec_dig2; // display tens of seconds digit
            4'b0100: time_digit = min_dig1; // display minutes digit
            4'b1000: time_digit = min_dig2; // display tens of minutes digit
            default: time_digit = 4'b0000; // default case
        endcase
    end

    always_comb begin
        case (selector)
            4'b0001: dp_in = decimal_point[0]; // DP right of seconds digit
```

```
            4'b0010: dp_in = decimal_point[1]; // DP right of tens of seconds digit
            4'b0100: dp_in = decimal_point[2]; // DP right of minutes digit
            4'b1000: dp_in = decimal_point[3]; // DP right of tens of minutes digit
            default: dp_in = 0; // default case (all DP are OFF)
        endcase
    end

endmodule
```

Listing 6. edge_detector.sv

```
//*****************************************************************************
// Module: edge_detector
//
// Description:
// Detects rising and falling edges of an input signal. The input signal is
// synchronized through two flip-flops to prevent metastability issues when
// crossing clock domains.
//
// Ports:
// - clk: System clock
// - reset: Active-high synchronous reset
// - signal_in: Input signal to detect edges on (typically from external source)
// - rising_edge: Output pulse (high for one clock cycle) on rising edge →(01)
// - falling_edge: Output pulse (high for one clock cycle) on falling edge →(10)
//
// Operation:
// 1. Input signal is synchronized through two flip-flops (sync1, sync2)
// 2. Synchronized signal is delayed by one clock (signal_delayed)
// 3. Rising edge detected when: previous=0 AND current=1
// 4. Falling edge detected when: previous=1 AND current=0
//
// Example:
// signal_in:
//
//
// rising_edge:
//
//
// falling_edge:
//
//
//*****************************************************************************

module edge_detector (
    input logic clk,
    input logic reset,
    input logic signal_in,
    output logic rising_edge,
    output logic falling_edge
);

    // Two-stage synchronizer to prevent metastability
    logic sync1, sync2;

    // Delayed version for edge comparison
    logic signal_delayed;

    // Synchronization chain
    always_ff @(posedge clk) begin
        if (reset) begin
            sync1 <= 1'b0;
            sync2 <= 1'b0;
            signal_delayed <= 1'b0;
        end else begin
            sync1 <= signal_in; // First stage synchronization
            sync2 <= sync1; // Second stage synchronization
            signal_delayed <= sync2; // Delay for edge comparison
        end
    end

    // Edge detection combinational logic
    assign rising_edge = !signal_delayed && sync2; // →01 transition
```

```
63      assign falling_edge = signal_delayed && !sync2; // →10 transition
64
65  endmodule
```

Listing 7. lab_5_top_level_students.sv

```
/*
This design uses the XADC from the IP Catalog. The specific channel is XADC4.
The Auxiliary Analog Inputs are VAUXP[15] and VAUXN[15].
These map to the FPGA pins of N2 and N1, respecitively (also in .XDC).
These map to the JXADC PMOD and the specific PMOD inputs are
JXADC4:N2 and JXAC10:N1, respectively. These pin are right beside the PMOD GND
on JXAC11:GND and JXAC5:GND.

The ADC is set to single-ended, continuous sampling, 1 MSps, 256 averaging.
Additional averaging is done using the averager module below.
*/
module lab_5_top_level_students (
    input logic clk,
    input logic reset,
    input logic [1:0] bin_bcd_select,
    input logic [1:0] system_select,
    input logic pwm_compare,
    input logic r2r_compare,
    input logic adc_mode,

   // input logic [15:0] switches_inputs,
    input vauxp15, // Analog input (positive) - connect to JXAC4:N2 PMOD pin (XADC4)
    input vauxn15, // Analog input (negative) - connect to JXAC10:N1 PMOD pin (XADC4)
    output logic CA, CB, CC, CD, CE, CF, CG, DP,
    output logic AN1, AN2, AN3, AN4,
    output logic pwm_out,
    output logic [7:0] r2r_bus
);
    // Internal signal declarations

    // Tie analog inputs to high-impedance to prevent I/O buffer inference
    //assign vauxp5 = 1'bZ;
    //assign vauxn5 = 1'bZ;

    logic [15:0] mux_out;
    logic [15:0] xadc_out, pwm_system_out, r2r_system_out;
    logic [3:0] decimal_point;




    xadc_system xadc_system(
    .clk (clk),
    .reset (reset),
    .bin_bcd_select (bin_bcd_select),
    .vauxp15(vauxp15),
    .vauxn15 (vauxn15),
    .xadc_output (xadc_out)
    );

    pwm_system pwm_system(
    .clk(clk),
    .reset(reset),
    .bin_bcd_select(bin_bcd_select),
    .pwm_compare(pwm_compare),
    .adc_mode(adc_mode),
    .pwm_system_out(pwm_system_out),
    .pwm_out(pwm_out)
    );


    r2r_subsystem r2r_subsystem(
```

```
    .clk(clk),
    .reset(reset),
    .bin_bcd_select(bin_bcd_select),
    .r2r_compare(r2r_compare),
    .adc_mode(adc_mode),
    .r2r_bus(r2r_bus),
    .r2r_system_out(r2r_system_out)
    );

     mux4_16_bits MUX4 (
        .in0(xadc_out), // hexadecimal, scaled and averaged
        .in1(pwm_system_out), // decimal, scaled and averaged
        .in2(r2r_system_out), // raw 12-bit ADC hexadecimal
        .in3(16'h0000), // averaged and before scaling 16-bit ADC (extra 4-bits from averaging) hexadecimal
        .select(system_select),
        .mux_out(mux_out)
    );

    // Decimal point control based on display mode
    decimal_point_mux DP_MUX (
        .select(bin_bcd_select),
        .dp(decimal_point)
    );



    // Seven Segment Display Subsystem
    seven_segment_display_subsystem SEVEN_SEGMENT_DISPLAY (
        .clk(clk),
        .reset(reset),
        .sec_dig1(mux_out[3:0]), // Lowest digit
        .sec_dig2(mux_out[7:4]), // Second digit
        .min_dig1(mux_out[11:8]), // Third digit
        .min_dig2(mux_out[15:12]), // Highest digit
        .decimal_point(decimal_point),
        .CA(CA), .CB(CB), .CC(CC), .CD(CD),
        .CE(CE), .CF(CF), .CG(CG), .DP(DP),
        .AN1(AN1), .AN2(AN2), .AN3(AN3), .AN4(AN4)
    );



endmodule
```

Listing 8. mux4_16_bits.sv

```
module mux4_16_bits(
    input logic [15:0] in0,
    input logic [15:0] in1,
    input logic [15:0] in2,
    input logic [15:0] in3,
    input logic [1:0] select,
    output logic [15:0] mux_out,
    output logic [3:0] decimal_point
    );

    always_comb begin
        case(select)
            2'b00: mux_out = in0;
            2'b01: mux_out = in1;
            2'b10: mux_out = in2;
            2'b11: mux_out = in3;
            default: mux_out = 16'h0000; // Default case: output all zeros
        endcase
    end

   always_comb begin
     case(select)
         2'b00: decimal_point = 4'b0000; // averaged ADC with extra 4 bits
         2'b01: decimal_point = 4'b0010; // averaged and scaled voltage
         2'b10: decimal_point = 4'b0000; // raw ADC (12-bits)
         2'b11: decimal_point = 4'b0000;
         default: decimal_point = 16'h0000; // Default case: output all zeros
     endcase
   end
   //assign decimal_pt = 4'b0010; // vector to control the decimal point, 1 = DP on, 0 = DP off
                                  // [0001] DP right of seconds digit
                                  // [0010] DP right of tens of seconds digit
                                  // [0100] DP right of minutes digit
                                  // [1000] DP right of tens of minutes digit

endmodule
```

Listing 9. pwm_system.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name: pwm_system
// Description: PWM-based ADC subsystem with selectable algorithm
// - Ramp ADC: Counts 0->255, captures at comparator edge
// - SAR ADC: Binary search, exactly 8 comparisons
//////////////////////////////////////////////////////////////////////////////////

module pwm_system (
    input logic clk,
    input logic reset,
    input logic [1:0] bin_bcd_select,
    input logic pwm_compare,
    input logic adc_mode,
    output logic [15:0] pwm_system_out,
    output logic pwm_out
);
    //
        ================================================================================

    // RAMP ADC Signals
    //
        ================================================================================

    logic [7:0] ramp_duty_cycle;
    logic ramp_rising_edge;
    logic ramp_falling_edge;
    logic ramp_enable;
    logic [7:0] ramp_raw_adc;
    logic [7:0] ramp_avg_data;
    logic ramp_comparator_was_high;
    logic ramp_pwm_out;

    assign ramp_enable = 1'b1;

    // Sawtooth generator for Ramp ADC (generates both PWM and ramp)
    saw_generator #(
        .WIDTH(8),
        .CLOCK_FREQ(200_000_000),
        .WAVE_FREQ_HZ(100)
    ) ramp_saw_gen (
        .clk(clk),
        .reset(reset),
        .enable(ramp_enable),
        .pwm_out(ramp_pwm_out),
        .R2R_out(ramp_duty_cycle)
    );

    // Edge detector for Ramp ADC
    edge_detector ramp_edge_det (
        .clk(clk),
        .reset(reset),
        .signal_in(pwm_compare),
        .rising_edge(ramp_rising_edge),
        .falling_edge(ramp_falling_edge)
    );

    // Ramp ADC capture with saturation handling
    always_ff @(posedge clk) begin
        if (reset) begin
            ramp_raw_adc <= 8'h00;
        end else if (ramp_falling_edge) begin
```

```
59              ramp_raw_adc <= ramp_duty_cycle;
60          end else if (ramp_duty_cycle >= 8'hF9) begin
61              if (pwm_compare) begin
62                  ramp_raw_adc <= 8'hFF;
63              end
64          end
65      end
66
67      // Averager for Ramp ADC
68      averager #(
69          .power(6),
70          .N(8)
71      ) ramp_averager (
72          .reset(reset),
73          .clk(clk),
74          .EN(ramp_falling_edge),
75          .Din(ramp_raw_adc),
76          .Q(ramp_avg_data)
77      );
78
79      //
            ==========================================================================
80      // SAR ADC Signals
81      //
            ==========================================================================
82      logic [7:0] sar_dac_out;
83      logic [7:0] sar_adc_result;
84      logic sar_conversion_done;
85      logic sar_start;
86      logic sar_pwm_out;
87      logic [7:0] sar_avg_data;
88      logic sar_enable;
89
90      assign sar_enable = (adc_mode == 1'b1);
91
92      // SAR ADC core
93      sar_adc #(
94          .WIDTH(8),
95          .SETTLE_CYCLES(100000)
96      ) sar_adc_inst (
97          .clk(clk),
98          .reset(reset),
99          .start(sar_start),
100         .compare_in(pwm_compare),
101         .dac_out(sar_dac_out),
102         .adc_result(sar_adc_result),
103         .conversion_done(sar_conversion_done)
104     );
105
106     // PWM generator for SAR DAC output
107     pwm #(
108         .WIDTH(8)
109     ) sar_pwm_inst (
110         .clk(clk),
111         .reset(reset),
112         .enable(sar_enable),
113         .duty_cycle(sar_dac_out),
114         .pwm_out(sar_pwm_out)
115     );
116
117     // Auto-start logic and mode switch detection (combined)
```

```
    logic [15:0] sar_start_delay;
    logic adc_mode_prev;

    always_ff @(posedge clk) begin
        if (reset) begin
            sar_start <= 1'b0;
            sar_start_delay <= 16'd0;
            adc_mode_prev <= 1'b0;
        end else begin
            adc_mode_prev <= adc_mode;

            if (adc_mode == 1'b1) begin
                // Check for mode switch (rising edge of adc_mode)
                if (adc_mode && !adc_mode_prev) begin
                    sar_start_delay <= 16'd1;
                    sar_start <= 1'b0;
                end
                // Normal auto-restart logic
                else if (sar_conversion_done) begin
                    sar_start_delay <= 16'd1;
                    sar_start <= 1'b0;
                end else if (sar_start_delay > 0) begin
                    sar_start_delay <= sar_start_delay + 1;
                    if (sar_start_delay >= 16'd100) begin
                        sar_start <= 1'b1;
                        sar_start_delay <= 16'd0;
                    end
                end else if (sar_start) begin
                    sar_start <= 1'b0;
                end
            end else begin
                sar_start <= 1'b0;
                sar_start_delay <= 16'd0;
            end
        end
    end

    // Averager for SAR ADC
    averager #(
        .power(6),
        .N(8)
    ) sar_averager (
        .reset(reset),
        .clk(clk),
        .EN(sar_conversion_done),
        .Din(sar_adc_result),
        .Q(sar_avg_data)
    );

    //
        ============================================================================

    // Mode Selection Multiplexing
    //
        ============================================================================

    logic [7:0] selected_raw_adc;
    logic [7:0] selected_avg_data;
    logic selected_update_flag;

    always_comb begin
        if (adc_mode == 1'b0) begin
            // Ramp ADC mode
```

```
            pwm_out = ramp_pwm_out;
            selected_raw_adc = ramp_raw_adc;
            selected_avg_data = ramp_avg_data;
            selected_update_flag = ramp_falling_edge;
        end else begin
            // SAR ADC mode
            pwm_out = sar_pwm_out;
            selected_raw_adc = sar_adc_result;
            selected_avg_data = sar_avg_data;
            selected_update_flag = sar_conversion_done;
        end
    end

    //
        ================================================================================

    // Scaling and Display Logic (Common to both modes)
    //
        ================================================================================

    logic [15:0] scaled_pwm_data;
    logic [31:0] scaled_pwm_temp;
    logic [15:0] bcd_value;

    always_ff @(posedge clk) begin
        if (reset) begin
            scaled_pwm_data <= 0;
            scaled_pwm_temp <= 0;
        end
        else if (selected_update_flag) begin
            scaled_pwm_temp <= {24'd0, selected_avg_data[7:0]} * 32'd13246;
            scaled_pwm_data <= scaled_pwm_temp[25:10];
        end
    end

    // Binary to BCD converter for decimal display
    bin_to_bcd bin2bcd_inst (
        .clk(clk),
        .reset(reset),
        .bin_in(scaled_pwm_data),
        .bcd_out(bcd_value)
    );

    // Output multiplexer
    always_comb begin
        case (bin_bcd_select)
            2'b00: pwm_system_out = scaled_pwm_data;
            2'b01: pwm_system_out = bcd_value;
            2'b10: pwm_system_out = {8'h00, selected_raw_adc};
            2'b11: pwm_system_out = {8'h00, selected_avg_data};
            default: pwm_system_out = 16'h0000;
        endcase
    end
endmodule
```

Listing 10. pwm.sv

```
module pwm #(
    parameter int WIDTH = 8
) (
    input logic clk,
    input logic reset,
    input logic enable,
    input logic [WIDTH-1:0] duty_cycle,
    output logic pwm_out
);

    logic [WIDTH-1:0] counter;

    always_ff @(posedge clk) begin
        if (reset)
            counter <= 0;
        else if (enable)
            counter <= counter + 1;
    end

    always_comb begin
        if (!enable)
            pwm_out = 1'b0; // Output low when not enabled
        else if (duty_cycle == {WIDTH{1'b1}})
            pwm_out = 1'b1;
        else if (counter < duty_cycle)
            pwm_out = 1'b1;
        else
            pwm_out = 1'b0;
    end

endmodule
```

Listing 11. r2r_subsystem.sv

```
module r2r_subsystem (
    input logic clk,
    input logic reset,
    input logic [1:0] bin_bcd_select,
    input logic r2r_compare,
    input logic adc_mode, // 0 = Ramp, 1 = SAR
    output logic [7:0] r2r_bus,
    output logic [15:0] r2r_system_out
);
    //
        ============================================================================

    // RAMP ADC Signals
    //
        ============================================================================

    logic [7:0] ramp_duty_cycle;
    logic ramp_rising_edge;
    logic ramp_falling_edge;
    logic ramp_enable;
    logic [7:0] ramp_raw_adc;
    logic [7:0] ramp_avg_data;

    assign ramp_enable = 1'b1;

    // Sawtooth generator for Ramp ADC
    saw_generator #(
        .WIDTH(8),
        .CLOCK_FREQ(200_000_000),
        .WAVE_FREQ_HZ(100)
    ) ramp_saw_gen (
        .clk(clk),
        .reset(reset),
        .enable(ramp_enable),
        .R2R_out(ramp_duty_cycle)
    );

    // Edge detector for Ramp ADC
    edge_detector ramp_edge_det (
        .clk(clk),
        .reset(reset),
        .signal_in(r2r_compare),
        .rising_edge(ramp_rising_edge),
        .falling_edge(ramp_falling_edge)
    );

    // Ramp ADC capture with saturation handling
    always_ff @(posedge clk) begin
        if (reset) begin
            ramp_raw_adc <= 8'h00;
        end else if (ramp_falling_edge) begin
            ramp_raw_adc <= ramp_duty_cycle;
        end else if (ramp_duty_cycle == 8'hFE) begin
            if (r2r_compare) begin
                ramp_raw_adc <= 8'hFF;
            end
        end
    end

    // Averager for Ramp ADC
    averager #(
        .power(6),
```

```
        .N(8)
    ) ramp_averager (
        .reset(reset),
        .clk(clk),
        .EN(ramp_falling_edge),
        .Din(ramp_raw_adc),
        .Q(ramp_avg_data)
    );

    //
        ================================================================================

    // SAR ADC Signals
    //
        ================================================================================

    logic [7:0] sar_dac_out;
    logic [7:0] sar_adc_result;
    logic sar_conversion_done;
    logic sar_start;
    logic [7:0] sar_avg_data;

    // SAR ADC core (R2R settles much faster than PWM, so fewer settle cycles)
    sar_adc #(
        .WIDTH(8),
        .SETTLE_CYCLES(250000) // R2R settles quickly, ~1us at 100MHz
    ) sar_adc_inst (
        .clk(clk),
        .reset(reset),
        .start(sar_start),
        .compare_in(r2r_compare),
        .dac_out(sar_dac_out),
        .adc_result(sar_adc_result),
        .conversion_done(sar_conversion_done)
    );

    // Auto-start logic and mode switch detection (combined)
    logic [15:0] sar_start_delay;
    logic adc_mode_prev;

    always_ff @(posedge clk) begin
        if (reset) begin
            sar_start <= 1'b0;
            sar_start_delay <= 16'd0;
            adc_mode_prev <= 1'b0;
        end else begin
            adc_mode_prev <= adc_mode;

            if (adc_mode == 1'b1) begin
                // Check for mode switch (rising edge of adc_mode)
                if (adc_mode && !adc_mode_prev) begin
                    sar_start_delay <= 16'd1;
                    sar_start <= 1'b0;
                end
                // Normal auto-restart logic
                else if (sar_conversion_done) begin
                    sar_start_delay <= 16'd1;
                    sar_start <= 1'b0;
                end else if (sar_start_delay > 0) begin
                    sar_start_delay <= sar_start_delay + 1;
                    if (sar_start_delay >= 16'd100) begin
                        sar_start <= 1'b1;
                        sar_start_delay <= 16'd0;
```

```
                    end
                end else if (sar_start) begin
                    sar_start <= 1'b0;
                end
            end else begin
                sar_start <= 1'b0;
                sar_start_delay <= 16'd0;
            end
        end
    end

    // Averager for SAR ADC
    averager #(
        .power(6),
        .N(8)
    ) sar_averager (
        .reset(reset),
        .clk(clk),
        .EN(sar_conversion_done),
        .Din(sar_adc_result),
        .Q(sar_avg_data)
    );

    //
        ============================================================================

    // Mode Selection Multiplexing
    //
        ============================================================================

    logic [7:0] selected_raw_adc;
    logic [7:0] selected_avg_data;
    logic selected_update_flag;

    always_comb begin
        if (adc_mode == 1'b0) begin
            // Ramp ADC mode
            r2r_bus = ramp_duty_cycle;
            selected_raw_adc = ramp_raw_adc;
            selected_avg_data = ramp_avg_data;
            selected_update_flag = ramp_falling_edge;
        end else begin
            // SAR ADC mode
            r2r_bus = sar_dac_out;
            selected_raw_adc = sar_adc_result;
            selected_avg_data = sar_avg_data;
            selected_update_flag = sar_conversion_done;
        end
    end

    //
        ============================================================================

    // Scaling and Display Logic (Common to both modes)
    //
        ============================================================================

    logic [15:0] scaled_r2r_data;
    logic [31:0] scaled_r2r_temp;
    logic [15:0] bcd_value;

    always_ff @(posedge clk) begin
        if (reset) begin
```

```
            scaled_r2r_data <= 0;
            scaled_r2r_temp <= 0;
        end
        else if (selected_update_flag) begin
            scaled_r2r_temp <= {24'd0, selected_avg_data[7:0]} * 32'd13246;
            scaled_r2r_data <= scaled_r2r_temp[25:10];
        end
    end

    // Binary to BCD converter
    bin_to_bcd bin2bcd_inst (
        .clk(clk),
        .reset(reset),
        .bin_in(scaled_r2r_data),
        .bcd_out(bcd_value)
    );

    // Output multiplexer
    always_comb begin
        case (bin_bcd_select)
            2'b00: r2r_system_out = scaled_r2r_data;
            2'b01: r2r_system_out = bcd_value;
            2'b10: r2r_system_out = {8'h00, selected_raw_adc};
            2'b11: r2r_system_out = {8'h00, selected_avg_data};
            default: r2r_system_out = 16'h0000;
        endcase
    end

endmodule
```

Listing 12. sar.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name: sar_adc
// Description: Successive Approximation Register (SAR) ADC implementation
// FIXED: Added input synchronization and robust counters
//////////////////////////////////////////////////////////////////////////////////

module sar_adc #(
    parameter WIDTH = 8,
    parameter SETTLE_CYCLES = 50000 // Default for PWM usage
)(
    input logic clk,
    input logic reset,
    input logic start,
    input logic compare_in, //Different comparators signals for r2r and pwm
    output logic [WIDTH-1:0] dac_out,
    output logic [WIDTH-1:0] adc_result,
    output logic conversion_done
);

    // FSM State Definition
    typedef enum logic [2:0] {
        IDLE, INIT, WAIT_SETTLE, COMPARE, DECIDE, NEXT_BIT, DONE
    } state_t;

    state_t current_state, next_state;

    // Internal registers
    logic [WIDTH-1:0] sar_reg;
    logic [3:0] bit_index;
    logic [31:0] settle_counter; // 32-bit to support long delays
    logic comp_result;

    // Synchronizer signals
    logic comp_sync_1, comp_sync_2;

    // Output the current SAR register value to the DAC
    assign dac_out = sar_reg;

    // Double-flop synchronizer for asynchronous comparator input
    // This is critical for reliable state machine operation
    always_ff @(posedge clk) begin
        if (reset) begin
            comp_sync_1 <= 0;
            comp_sync_2 <= 0;
        end else begin
            comp_sync_1 <= compare_in;
            comp_sync_2 <= comp_sync_1;
        end
    end

    // State Register
    always_ff @(posedge clk) begin
        if (reset) current_state <= IDLE;
        else current_state <= next_state;
    end

    // Next State Logic
    always_comb begin
        next_state = current_state;

        case (current_state)
```

```
            IDLE: if (start) next_state = INIT;
            INIT: next_state = WAIT_SETTLE;
            WAIT_SETTLE: if (settle_counter >= (SETTLE_CYCLES - 1)) next_state = COMPARE;
            COMPARE: next_state = DECIDE;
            DECIDE: if (bit_index > 0) next_state = NEXT_BIT;
                          else next_state = DONE;
            NEXT_BIT: next_state = WAIT_SETTLE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    // Datapath
    always_ff @(posedge clk) begin
        if (reset) begin
            sar_reg <= '0;
            bit_index <= WIDTH - 1;
            settle_counter <= '0;
            comp_result <= '0;
            adc_result <= '0;
            conversion_done <= '0;
        end else begin
            conversion_done <= '0;

            case (current_state)
                IDLE: begin
                    settle_counter <= '0;
                    if (start) begin
                        sar_reg <= '0;
                        bit_index <= WIDTH - 1;
                    end
                end

                INIT: begin
                    sar_reg[WIDTH-1] <= 1'b1; // Set MSB
                    settle_counter <= '0;
                end

                WAIT_SETTLE: begin
                    settle_counter <= settle_counter + 1;
                end

                COMPARE: begin
                    // Use synchronized input
                    comp_result <= comp_sync_2;
                end

                DECIDE: begin
                    // If Vin < Vdac (Comparator Low), Clear the bit
                    if (!comp_result) begin
                        sar_reg[bit_index] <= 1'b0;
                    end
                    // Move index down
                    if (bit_index > 0) bit_index <= bit_index - 1'b1;
                end

                NEXT_BIT: begin
                    // Set next bit to 1
                    sar_reg[bit_index] <= 1'b1;
                    settle_counter <= '0;
                end

                DONE: begin
```

```
                        adc_result <= sar_reg;
                        conversion_done <= 1'b1;
                    end
                endcase
            end
        end
endmodule
```

Listing 13. saw_generator.sv

```
    module saw_generator #(
    parameter int WIDTH = 8,
    parameter int CLOCK_FREQ = 100_000_000,
    parameter int WAVE_FREQ_HZ = 1 // integer Hz (use 1 for the lab)
) (
    input logic clk,
    input logic reset,
    input logic enable,
    output logic pwm_out,
    output logic [WIDTH-1:0] R2R_out
);
    // 2^WIDTH steps per period →for WAVE_FREQ_HZ=1 Hz, we need 256 steps/sec
    localparam int STEPS = (1 << WIDTH); // 256
    // clocks per step (rounded, never 0)
    localparam int DIV = (CLOCK_FREQ + (STEPS*WAVE_FREQ_HZ)/2) / (STEPS*WAVE_FREQ_HZ);
    localparam int DIV_W = (DIV <= 1) ? 1 : $clog2(DIV);

    logic [DIV_W-1:0] divcnt;
    logic tick;

    // Step tick @ CLOCK_FREQ / DIV
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            divcnt <= '0;
            tick <= 1'b0;
        end else if (!enable) begin
            divcnt <= '0;
            tick <= 1'b0;
        end else if (divcnt == DIV-1) begin
            divcnt <= '0;
            tick <= 1'b1;
        end else begin
            divcnt <= divcnt + 1'b1;
            tick <= 1'b0;
        end
    end

    // Saw counter: 0..255 then wrap to 0
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            R2R_out <= '0;
        end else if (!enable) begin
            // OPTION A (recommended): restart from 0 next time
            R2R_out <= '0;
            // OPTION B: comment the line above to "pause" and resume where it left off
        end else if (tick) begin
            R2R_out <= R2R_out + 1'b1; // wraps naturally at 255->0
        end
    end

    // 8-bit PWM whose duty follows the saw value
    pwm #(.WIDTH(WIDTH)) u_pwm (
        .clk(clk), .reset(reset),
        .enable(enable),
        .duty_cycle(R2R_out),
        .pwm_out(pwm_out)
    );
endmodule
```

Listing 14. seven_segment_decoder.sv

```
//*************************************************************************
// Module: seven_segment_decoder
//
// Description:
// This module decodes a 4-bit input into the appropriate signals to drive a
// 7-segment LED display. It supports displaying hexadecimal digits (0-F),
// with additional support for a decimal point.
//
// Inputs:
// - data[3:0]: 4-bit input representing the digit to be displayed (0-F)
// - dp_in: Input for the decimal point (active-low)
//
// Outputs:
// - CA, CB, CC, CD, CE, CF, CG: Individual segment controls (active-low)
// - DP: Decimal point control (active-low)
//
// Behavior:
// - The module uses a combinational logic block to decode the input into
// a 7-bit pattern representing the segments to be lit.
// - The decoded pattern is then inverted and assigned to the respective
// segment outputs (CA-CG), as the 7-segment display is active-low.
// - The decimal point (DP) is passed through directly from dp_in.
//
// Note:
// - The module supports all hexadecimal digits (0-F), but digits A-F are
// noted as "Not used in stopwatch" in the original comments.
// - The segment mapping is as follows:
// A
// F B
// G
// E C
// D DP
//
//*************************************************************************/


module seven_segment_decoder (
    output logic CA,
    output logic CB,
    output logic CC,
    output logic CD,
    output logic CE,
    output logic CF,
    output logic CG,
    output logic DP,
    input logic dp_in,
    input logic [3:0] data
);

    logic [6:0] decoded_bits;

    always_comb begin
        // Decode the input data into 7-segment display pattern
                                            // ABCDEFG 7-segment LED pattern for reference (1 is on)
        case (data) // 6543210
            4'b0000: decoded_bits = 7'b1111110; // 0 A-6
            4'b0001: decoded_bits = 7'b0110000; // 1 F-1 B-5
            4'b0010: decoded_bits = 7'b1101101; // 2 G-0
            4'b0011: decoded_bits = 7'b1111001; // 3 E-2 C-4
            4'b0100: decoded_bits = 7'b0110011; // 4 D-3 DP
            4'b0101: decoded_bits = 7'b1011011; // 5
            4'b0110: decoded_bits = 7'b1011111; // 6
```

```
            4'b0111: decoded_bits = 7'b1110000; // 7
            4'b1000: decoded_bits = 7'b1111111; // 8
            4'b1001: decoded_bits = 7'b1111011; // 9
            4'b1010: decoded_bits = 7'b1110111; // A (Not used in stopwatch)
            4'b1011: decoded_bits = 7'b1111111; // B (Not used in stopwatch)
            4'b1100: decoded_bits = 7'b1001110; // C (Not used in stopwatch)
            4'b1101: decoded_bits = 7'b1111110; // D (Not used in stopwatch)
            4'b1110: decoded_bits = 7'b1001111; // E (Not used in stopwatch)
            4'b1111: decoded_bits = 7'b1000111; // F (Not used in stopwatch)
// Students: fill in the remaining rows for this case statement,
// to account for the hexademcial digits A, B, C, D, E, and F

            default: decoded_bits = 7'b0000000; // All LEDs off
        endcase // ABCDEFG
    end // 6543210

    // Assign the decoded bits to the 7-segment display outputs (active-low on Basys3, i.e. 0 is ON)
    // Invert LED signals that were active-high for convenience
    assign DP = ~dp_in; // Passes through the decimal point signal (from top_level)
    assign CA = ~decoded_bits[6];
    assign CB = ~decoded_bits[5];
    assign CC = ~decoded_bits[4];
    assign CD = ~decoded_bits[3];
    assign CE = ~decoded_bits[2];
    assign CF = ~decoded_bits[1];
    assign CG = ~decoded_bits[0];

endmodule
```

Listing 15. seven_segment_digit_selector.sv

```
//////////////////////////////////////////////////////////////////////////////////
// Module Name: seven_segment_digit_selector
//
// Description:
// This module implements a digit selector for a 4-digit 7-segment display.
// It generates a rotating selection signal at approximately 763 Hz,
// allowing for time-multiplexed control of the 4 digits.
//
// Inputs:
// - clk : System clock (assumed to be 100 MHz)
// - reset : Active-high synchronous reset
//
// Outputs:
// - digit_select : 4-bit output indicating the currently selected digit (one-hot encoded)
// - an_outputs : 4-bit active-low output for directly driving 7-segment display anodes
//
// Internal Signals:
// - count : 17-bit counter used to generate a ~763 Hz clock signal
// - q : 4-bit register storing the current digit selection state
// - d : Next state for the digit selection register
//
// Operation:
// 1. A 17-bit counter divides the 100 MHz clock to create a ~763 Hz signal.
// 2. On each ~763 Hz clock tick, the digit selection rotates:
// 1000 -> 0100 -> 0010 -> 0001 -> 1000 (repeat)
// 3. The digit_select output directly reflects the current selection state.
// 4. The an_outputs is the inverted digit_select, suitable for driving active-low anodes.
//
// Reset Behavior:
// On reset, the counter is cleared and the digit selection is set to 1111.
// This ensures a known state and allows the module to start its rotation from a defined point.
//
// Note: The actual update frequency may vary slightly from 763 Hz due to the binary division.
//////////////////////////////////////////////////////////////////////////////////

module seven_segment_digit_selector (
    input logic clk,
    input logic reset,
    output logic [3:0] digit_select,
    output logic [3:0] an_outputs
);

    logic [3:0] d, q;
    logic [16:0] count;

    // 1 kHz clock process (100 MHz / 2^17 = 762.9 Hz)
    always_ff @(posedge clk) begin
        if (reset) begin
            count <= 17'b0;
        end else begin
            count <= count + 1;
        end
    end

    // DFFs process
    always_ff @(posedge clk) begin
        if (reset) begin
            // Reset state values for q
            q <= 4'b1111;
        end else if (count == 17'b0) begin
            // Propagate signals through the DFF
            if (q[0] && q[1]) begin
```

```
                q <= 4'b1000;
            end else begin
                q <= d;
            end
        end
    end

    // Connect the DFFs into a chain/loop
    assign d[0] = q[3];
    assign d[1] = q[0];
    assign d[2] = q[1];
    assign d[3] = q[2];

    // Output assignments
    assign digit_select = q;

    // Copying q to the LED anodes, invert because active low
    assign an_outputs = ~q;

endmodule
```

Listing 16. seven_segment_display_subsystem.sv

```
//*********************************************************************************
// Module: seven_segment_display_subsystem
//
// Description:
// This module integrates the digit_multiplexor, seven_segment_digit_selector,
// and seven_segment_decoder into a single subsystem to drive a 4-digit
// 7-segment display. It is designed to interface with a top-level module like
// lab_1b_top_level and enables hierarchical design.
//
// Inputs:
// - clk: Clock input
// - reset: Active-high synchronous reset
// - sec_dig1, sec_dig2, min_dig1, min_dig2: 4-bit BCD digit inputs
//
// Outputs:
// - CA, CB, CC, CD, CE, CF, CG, DP: Individual segment controls (active-low)
// - AN1, AN2, AN3, AN4: Anode controls for the 4 digits (active-low)
//
// Internal Signals:
// - digit_select: One-hot encoded output for digit selection
// - digit_to_display: 4-bit BCD value to display on the current digit
// - in_DP: Control signal for the decimal point
//
//*********************************************************************************

module seven_segment_display_subsystem (
    input logic clk,
    input logic reset,
    input logic [3:0] sec_dig1, // seconds digit (units)
    input logic [3:0] sec_dig2, // tens of seconds
    input logic [3:0] min_dig1, // minutes digit (units)
    input logic [3:0] min_dig2, // tens of minutes
    input logic [3:0] decimal_point,
    output logic CA, CB, CC, CD, CE, CF, CG, DP, // segment outputs (active-low)
    output logic AN1, AN2, AN3, AN4 // anode outputs for digit selection (active-low)
);

    // Internal signals
    logic [3:0] digit_to_display;
    logic [3:0] digit_select;
    logic [3:0] an_outputs;
    logic in_DP, out_DP;

    // Instantiate digit multiplexor
    digit_multiplexor DIGIT_MUX (
        .sec_dig1( sec_dig1), // input for seconds digit (units)
        .sec_dig2( sec_dig2), // input for tens of seconds digit
        .min_dig1( min_dig1), // input for minutes digit (units)
        .min_dig2( min_dig2), // input for tens of minutes digit
        .selector( digit_select), // one-hot selector for the digit
        .decimal_point(decimal_point),
        .time_digit(digit_to_display), // 4-bit digit output to display
        .dp_in(in_DP) // output
    );

    // Instantiate digit selector
    seven_segment_digit_selector DIGIT_SELECTOR (
        .clk( clk), // Clock input
        .reset( reset), // Reset input (active-high)
        .digit_select(digit_select), // Output: one-hot encoded digit select
        .an_outputs( an_outputs) // Output: active-low anode controls
    );
```

```
63
64      // Instantiate seven segment decoder
65      seven_segment_decoder SEG_DECODER (
66          .data( digit_to_display), // Input: 4-bit BCD digit to display
67          .dp_in( in_DP), // Input: Decimal point control
68          .CA( CA), .CB( CB), .CC( CC), .CD( CD), .CE( CE), .CF( CF), .CG( CG), // Segment outputs (active-low)
69          .DP( out_DP) // Decimal point output (active-low)
70      );
71
72      // Connect anodes
73      assign AN1 = an_outputs[0];
74      assign AN2 = an_outputs[1];
75      assign AN3 = an_outputs[2];
76      assign AN4 = an_outputs[3];
77
78      // Control the decimal point: You can modify `in_DP` assignment as per the design
79      //assign in_DP = 0; // No decimal point by default, modify as needed
80      assign DP = out_DP; // Pass the decimal point signal from the decoder
81
82  endmodule
```

Listing 17. XADC_system.sv

```
/*
This design uses the XADC from the IP Catalog. The specific channel is XADC4.
The Auxiliary Analog Inputs are VAUXP[15] and VAUXN[15].
These map to the FPGA pins of N2 and N1, respecitively (also in .XDC).
These map to the JXADC PMOD and the specific PMOD inputs are
JXADC4:N2 and JXAC10:N1, respectively. These pin are right beside the PMOD GND
on JXAC11:GND and JXAC5:GND.

The ADC is set to single-ended, continuous sampling, 1 MSps, 256 averaging.
Additional averaging is done using the averager module below.
*/
module xadc_system(
    input logic clk,
    input logic reset,
    input logic [1:0] bin_bcd_select,
   // input logic [15:0] switches_inputs,
    input vauxp15, // Analog input (positive) - connect to JXAC4:N2 PMOD pin (XADC4)
    input vauxn15, // Analog input (negative) - connect to JXAC10:N1 PMOD pin (XADC4)
    output logic [15:0] xadc_output,
    output logic [15:0] led
);
    // Internal signal declarations

    // Tie analog inputs to high-impedance to prevent I/O buffer inference
    //assign vauxp5 = 1'bZ;
    //assign vauxn5 = 1'bZ;

    logic [15:0] data, ave_data; // Raw ADC data
    logic [15:0] scaled_adc_data; // Scaled ADC data for display;
    logic [3:0] decimal_pt; // vector to control the decimal point, 1 = DP on, 0 = DP off
                            // [0001] DP right of seconds digit
                            // [0010] DP right of tens of seconds digit
                            // [0100] DP right of minutes digit
                            // [1000] DP right of tens of minutes digit
    logic [15:0] bcd_value, mux_out;

    // Constants
    localparam CHANNEL_ADDR = 7'h1f; // XA4/AD15 (for XADC4)

adc_subsystem ADC (
    .clk(clk),
    .reset(reset),
    .vauxp15(vauxp15), // Analog input (positive) - connect to JXAC4:N2 PMOD pin (XADC4)
    .vauxn15(vauxn15), // Analog input (negative) - connect to JXAC10:N1 PMOD pin (XADC4)
    .data_out(data),
    .scaled_adc_data(scaled_adc_data),
    .ave_data(ave_data)
);

    bin_to_bcd BIN2BCD (
        .clk(clk),
        .reset(reset),
        .bin_in(scaled_adc_data),
        .bcd_out(bcd_value)
    );

    mux4_16_bits MUX4 (
        .in0(scaled_adc_data), // hexadecimal, scaled and averaged
        .in1(bcd_value), // decimal, scaled and averaged
        .in2(data[15:4]), // raw 12-bit ADC hexadecimal
        .in3(ave_data), // averaged and before scaling 16-bit ADC (extra 4-bits from averaging) hexadecimal
        .select(bin_bcd_select),
```

```
63              .mux_out(xadc_output)
64      );
65
66
67
68  endmodule
```

Listing 18. Basys3_Lab_7.xdc

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
## Xilinx part number XC7A35T-1CPG236C (from Reference Manual)
## Xilinx part number xc7a35tcpg236-1 (from Xilinx Vivado)

## Clock signal
set_property -dict { PACKAGE_PIN W5 IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]


## Switches
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports {bin_bcd_select[0]}]
set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports {bin_bcd_select[1]}]
#set_property -dict { PACKAGE_PIN W16 IOSTANDARD LVCMOS33 } [get_ports {system_select[0]}]
#set_property -dict { PACKAGE_PIN W17 IOSTANDARD LVCMOS33 } [get_ports {system_select[1]}]
#set_property -dict { PACKAGE_PIN W15 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[4]}]
#set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[5]}]
#set_property -dict { PACKAGE_PIN W14 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[6]}]
#set_property -dict { PACKAGE_PIN W13 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[7]}]
#set_property -dict { PACKAGE_PIN V2 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[8]}]
#set_property -dict { PACKAGE_PIN T3 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[9]}]
#set_property -dict { PACKAGE_PIN T2 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[10]}]
#set_property -dict { PACKAGE_PIN R3 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[11]}]
#set_property -dict { PACKAGE_PIN W2 IOSTANDARD LVCMOS33 } [get_ports {switches_inputs[12]}]
set_property -dict { PACKAGE_PIN U1 IOSTANDARD LVCMOS33 } [get_ports adc_mode]
set_property -dict { PACKAGE_PIN T1 IOSTANDARD LVCMOS33 } [get_ports {system_select[0]}]
set_property -dict { PACKAGE_PIN R2 IOSTANDARD LVCMOS33 } [get_ports {system_select[1]}]


## LEDs
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports {led[0]}]
set_property -dict { PACKAGE_PIN E19 IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
set_property -dict { PACKAGE_PIN U19 IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
set_property -dict { PACKAGE_PIN V19 IOSTANDARD LVCMOS33 } [get_ports {led[3]}]
set_property -dict { PACKAGE_PIN W18 IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
set_property -dict { PACKAGE_PIN U15 IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports {led[6]}]
set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports {led[7]}]
set_property -dict { PACKAGE_PIN V13 IOSTANDARD LVCMOS33 } [get_ports {led[8]}]
set_property -dict { PACKAGE_PIN V3 IOSTANDARD LVCMOS33 } [get_ports {led[9]}]
set_property -dict { PACKAGE_PIN W3 IOSTANDARD LVCMOS33 } [get_ports {led[10]}]
set_property -dict { PACKAGE_PIN U3 IOSTANDARD LVCMOS33 } [get_ports {led[11]}]
set_property -dict { PACKAGE_PIN P3 IOSTANDARD LVCMOS33 } [get_ports {led[12]}]
set_property -dict { PACKAGE_PIN N3 IOSTANDARD LVCMOS33 } [get_ports {led[13]}]
set_property -dict { PACKAGE_PIN P1 IOSTANDARD LVCMOS33 } [get_ports {led[14]}]
set_property -dict { PACKAGE_PIN L1 IOSTANDARD LVCMOS33 } [get_ports {led[15]}]


##7 Segment Display
set_property -dict { PACKAGE_PIN W7 IOSTANDARD LVCMOS33 } [get_ports {CA}]
set_property -dict { PACKAGE_PIN W6 IOSTANDARD LVCMOS33 } [get_ports {CB}]
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS33 } [get_ports {CC}]
set_property -dict { PACKAGE_PIN V8 IOSTANDARD LVCMOS33 } [get_ports {CD}]
set_property -dict { PACKAGE_PIN U5 IOSTANDARD LVCMOS33 } [get_ports {CE}]
set_property -dict { PACKAGE_PIN V5 IOSTANDARD LVCMOS33 } [get_ports {CF}]
set_property -dict { PACKAGE_PIN U7 IOSTANDARD LVCMOS33 } [get_ports {CG}]

set_property -dict { PACKAGE_PIN V7 IOSTANDARD LVCMOS33 } [get_ports DP]

set_property -dict { PACKAGE_PIN U2 IOSTANDARD LVCMOS33 } [get_ports {AN1}]
```

```
set_property -dict { PACKAGE_PIN U4 IOSTANDARD LVCMOS33 } [get_ports {AN2}]
set_property -dict { PACKAGE_PIN V4 IOSTANDARD LVCMOS33 } [get_ports {AN3}]
set_property -dict { PACKAGE_PIN W4 IOSTANDARD LVCMOS33 } [get_ports {AN4}]


##Buttons
# Basys3 pushbuttons are normally 0, and 1 when pushed down
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports reset]; #set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports btnC]
#set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports startSAR]
#set_property -dict { PACKAGE_PIN W19 IOSTANDARD LVCMOS33 } [get_ports btnL]
#set_property -dict { PACKAGE_PIN T17 IOSTANDARD LVCMOS33 } [get_ports btnR]
#set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports btnD]


##Pmod Header JA
set_property -dict { PACKAGE_PIN H1 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[0]}];#Sch name = JA7
set_property -dict { PACKAGE_PIN J1 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[1]}];#Sch name = JA1
set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[2]}];#Sch name = JA8
set_property -dict { PACKAGE_PIN L2 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[3]}];#Sch name = JA2
set_property -dict { PACKAGE_PIN H2 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[4]}];#Sch name = JA9
set_property -dict { PACKAGE_PIN J2 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[5]}];#Sch name = JA3
set_property -dict { PACKAGE_PIN G3 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[6]}];#Sch name = JA10
set_property -dict { PACKAGE_PIN G2 IOSTANDARD LVCMOS33 } [get_ports {r2r_bus[7]}];#Sch name = JA4

##Pmod Header JB
#set_property -dict { PACKAGE_PIN A14 IOSTANDARD LVCMOS33 } [get_ports {JB[0]}];#Sch name = JB1
#set_property -dict { PACKAGE_PIN A16 IOSTANDARD LVCMOS33 } [get_ports {JB[1]}];#Sch name = JB2
#set_property -dict { PACKAGE_PIN B15 IOSTANDARD LVCMOS33 } [get_ports {JB[2]}];#Sch name = JB3
#set_property -dict { PACKAGE_PIN B16 IOSTANDARD LVCMOS33 } [get_ports {JB[3]}];#Sch name = JB4
#set_property -dict { PACKAGE_PIN A15 IOSTANDARD LVCMOS33 } [get_ports {JB[4]}];#Sch name = JB7
#set_property -dict { PACKAGE_PIN A17 IOSTANDARD LVCMOS33 } [get_ports {JB[5]}];#Sch name = JB8
#set_property -dict { PACKAGE_PIN C15 IOSTANDARD LVCMOS33 } [get_ports {JB[6]}];#Sch name = JB9
#set_property -dict { PACKAGE_PIN C16 IOSTANDARD LVCMOS33 } [get_ports {JB[7]}];#Sch name = JB10

##Pmod Header JC
#set_property -dict { PACKAGE_PIN K17 IOSTANDARD LVCMOS33 } [get_ports {JC[0]}];#Sch name = JC1
#set_property -dict { PACKAGE_PIN M18 IOSTANDARD LVCMOS33 } [get_ports {JC[1]}];#Sch name = JC2
#set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports {JC[2]}];#Sch name = JC3
#set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33 } [get_ports {JC[3]}];#Sch name = JC4
#set_property -dict { PACKAGE_PIN L17 IOSTANDARD LVCMOS33 } [get_ports {JC[4]}];#Sch name = JC7
#set_property -dict { PACKAGE_PIN M19 IOSTANDARD LVCMOS33 } [get_ports {JC[5]}];#Sch name = JC8
#set_property -dict { PACKAGE_PIN P17 IOSTANDARD LVCMOS33 } [get_ports {JC[6]}];#Sch name = JC9
#set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports {JC[7]}];#Sch name = JC10

## ORIGINAL
##Pmod Header JXADC
##set_property PACKAGE_PIN J3 [get_ports vauxp5]; #set_property -dict { PACKAGE_PIN J3 IOSTANDARD LVCMOS33 } [get_ports {JXADC[0]}];#Sch name = XA1_P
##set_property -dict { PACKAGE_PIN L3 IOSTANDARD LVCMOS33 } [get_ports {JXADC[1]}];#Sch name = XA2_P
#set_property -dict { PACKAGE_PIN M2 IOSTANDARD LVCMOS33 } [get_ports {JXADC[2]}];#Sch name = XA3_P
#set_property PACKAGE_PIN N2 [get_ports vauxp15]; #set_property -dict { PACKAGE_PIN N2 IOSTANDARD LVCMOS33 } [get_ports {JXADC[3]}];#Sch name = XA4_P
##set_property PACKAGE_PIN K3 [get_ports vauxn5]; #set_property -dict { PACKAGE_PIN K3 IOSTANDARD LVCMOS33 } [get_ports {JXADC[4]}];#Sch name = XA1_N
##set_property -dict { PACKAGE_PIN M3 IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}];#Sch name = XA2_N
##set_property -dict { PACKAGE_PIN M1 IOSTANDARD LVCMOS33 } [get_ports {JXADC[6]}];#Sch name = XA3_N
#set_property PACKAGE_PIN N1 [get_ports vauxn15]; #set_property -dict { PACKAGE_PIN N1 IOSTANDARD LVCMOS33 } [get_ports {JXADC[7]}];#Sch name = XA4_N

##############################################################
##Pmod Header JXADC
#set_property PACKAGE_PIN J3 [get_ports pwm_out]; #set_property -dict { PACKAGE_PIN J3 IOSTANDARD
```

```
      LVCMOS33 } [get_ports {JXADC[0]}];#Sch name = XA1_P
set_property -dict { PACKAGE_PIN L3 IOSTANDARD LVCMOS33 } [get_ports r2r_compare];#Sch name = XA2_P
#set_property -dict { PACKAGE_PIN M2 IOSTANDARD LVCMOS33 } [get_ports {JXADC[2]}];#Sch name = XA3_P
set_property -dict { PACKAGE_PIN N2 IOSTANDARD LVCMOS33 } [get_ports {vauxp15}];#Sch name = XA4_P
#set_property PACKAGE_PIN K3 [get_ports pwm_compare]; #set_property -dict { PACKAGE_PIN K3 IOSTANDARD
      LVCMOS33 } [get_ports {JXADC[4]}];#Sch name = XA1_N
#set_property -dict { PACKAGE_PIN M3 IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}];#Sch name = XA2_N
#set_property -dict { PACKAGE_PIN M1 IOSTANDARD LVCMOS33 } [get_ports {JXADC[6]}];#Sch name = XA3_N
set_property -dict { PACKAGE_PIN N1 IOSTANDARD LVCMOS33 } [get_ports {vauxn15}];#Sch name = XA4_N
set_property -dict { PACKAGE_PIN J3 IOSTANDARD LVCMOS33 } [get_ports pwm_out];#Sch name = XA4_N
set_property -dict { PACKAGE_PIN K3 IOSTANDARD LVCMOS33 } [get_ports pwm_compare];#Sch name = XA4_N

##VGA Connector
#set_property -dict { PACKAGE_PIN G19 IOSTANDARD LVCMOS33 } [get_ports {vgaRed[0]}]
#set_property -dict { PACKAGE_PIN H19 IOSTANDARD LVCMOS33 } [get_ports {vgaRed[1]}]
#set_property -dict { PACKAGE_PIN J19 IOSTANDARD LVCMOS33 } [get_ports {vgaRed[2]}]
#set_property -dict { PACKAGE_PIN N19 IOSTANDARD LVCMOS33 } [get_ports {vgaRed[3]}]
#set_property -dict { PACKAGE_PIN N18 IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[0]}]
#set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[1]}]
#set_property -dict { PACKAGE_PIN K18 IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[2]}]
#set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[3]}]
#set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[0]}]
#set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[1]}]
#set_property -dict { PACKAGE_PIN G17 IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[2]}]
#set_property -dict { PACKAGE_PIN D17 IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[3]}]
#set_property -dict { PACKAGE_PIN P19 IOSTANDARD LVCMOS33 } [get_ports Hsync]
#set_property -dict { PACKAGE_PIN R19 IOSTANDARD LVCMOS33 } [get_ports Vsync]


##USB-RS232 Interface
#set_property -dict { PACKAGE_PIN B18 IOSTANDARD LVCMOS33 } [get_ports RsRx]
#set_property -dict { PACKAGE_PIN A18 IOSTANDARD LVCMOS33 } [get_ports RsTx]


##USB HID (PS/2)
#set_property -dict { PACKAGE_PIN C17 IOSTANDARD LVCMOS33 PULLUP true } [get_ports PS2Clk]
#set_property -dict { PACKAGE_PIN B17 IOSTANDARD LVCMOS33 PULLUP true } [get_ports PS2Data]


##Quad SPI Flash
##Note that CCLK_0 cannot be placed in 7 series devices. You can access it using the
##STARTUPE2 primitive.
#set_property -dict { PACKAGE_PIN D18 IOSTANDARD LVCMOS33 } [get_ports {QspiDB[0]}]
#set_property -dict { PACKAGE_PIN D19 IOSTANDARD LVCMOS33 } [get_ports {QspiDB[1]}]
#set_property -dict { PACKAGE_PIN G18 IOSTANDARD LVCMOS33 } [get_ports {QspiDB[2]}]
#set_property -dict { PACKAGE_PIN F18 IOSTANDARD LVCMOS33 } [get_ports {QspiDB[3]}]
#set_property -dict { PACKAGE_PIN K19 IOSTANDARD LVCMOS33 } [get_ports QspiCSn]


### Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```