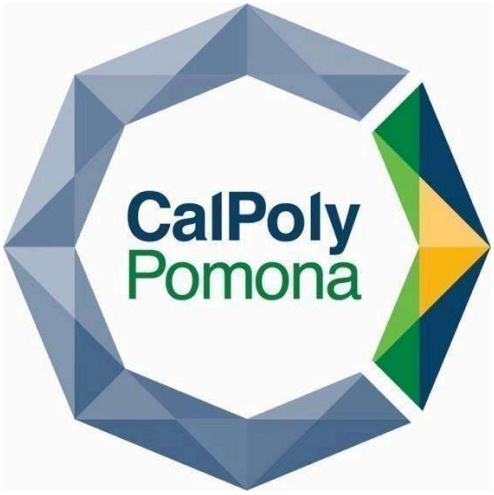


California Polytechnic State University Pomona

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Digital Circuit Design Verilog

ECE 3300L

Report #7

Prepared by

Heba Hafez 017353323

Sean Wygant 017376658

Group Y

Mohamed Aly

August 6, 2025

Objective: Design a 16-bit barrel shifter/rotator on the Nexys A7 FPGA. The barrel shifter must support both logical and rotational shifting operations in either direction (left or right), with the shift amount controlled by debounced push-buttons and an up-counter. Display the 16-bit result across the four hexadecimal digits on the 7-segment display.

Code and Explanation:

top_lab7.v

```
module top_lab7(
    input clk,
    input rst,
    input [15:0] SW,
    input BtnC, BtnU, BtnL, BtnR, BtnD,
    output [15:0] LED,
    output [6:0] SEG,
    output [7:0] AN
    );

    wire BtnCToggle, BtnUToggle, BtnLToggle, BtnRToggle, BtnDToggle, clk_1khz, clk_2hz;
    wire [15:0] result;
    wire [3:0] ShAmt;
    wire togU, pulse;

    toggle Center(.clk(clk), .rst(rst), .btn_raw(BtnC), .state(BtnCToggle));
    toggle Left(.clk(clk), .rst(rst), .btn_raw(BtnL), .state(BtnLToggle));
    toggle Right(.clk(clk), .rst(rst), .btn_raw(BtnR), .state(BtnRToggle));
    toggle Down(.clk(clk), .rst(rst), .btn_raw(BtnD), .state(BtnDToggle));
    ShAmt_decoder Up(.clk(clk), .rst(rst), .ShAmt(ShAmt), .BtnC(BtnCToggle), .BtnD(BtnDToggle), .BtnU(BtnU));
    clock_divider_fixed clock(.clk(clk), .rst(rst), .clk_2hz(clk_2hz), .clk_1khz(clk_1khz));

    barrel_shifter16(.control({BtnLToggle,BtnRToggle}), .SW(SW), .ShAmt(ShAmt), .out(result));

    seg7_scan8 seg7(.clk(clk), .rst(rst), .bits(result), .AN(AN), .SEG(SEG));

    assign LED[3:0] = ShAmt;
    assign LED[6:5] = {BtnLToggle, BtnRToggle};

endmodule
```

The top_lab7 module is the top-level design that wires the user input, display, and processing. It instantiates toggles for various buttons, a clock divider, a shift amount decoder, a 16-bit barrel shifter, and a 7-segment display scanner. The shift direction is determined by the left and right button toggles, and the shift amount is derived from decoded button inputs.

toggle.v

```
module toggle(
    input clk,
    input rst,
    input btn_raw,
    output reg state
);
    wire btn_clean;
    reg btn_prev;
    debounce db (.clk(clk), .btn_in(btn_raw), .btn_clean(btn_clean));
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            state <= 0;
            btn_prev <= 0;
        end
        else begin
            if (btn_clean && !btn_prev)
                state <= ~state;
            btn_prev <= btn_clean;
        end
    end
endmodule
```

The toggle module creates a one-bit toggling state that flips each time a button is pressed, using clocked logic. To prevent glitches caused by button bouncing, it first cleans the input with a debounce module. On each clock cycle, it compares the current debounced signal with the previous state (btn_prev), and if a new rising edge is detected, it toggles the output state. The module also supports an asynchronous reset to initialize the toggle state.

debounce.v

The debounce module removes mechanical bouncing noise from a raw button input by sampling it over three clock cycles using a 3-bit shift register. If all three samples are high, it sets the cleaned output btn_clean to 1; if all three are low, it sets it to 0. This effectively filters out short glitches and ensures the button state only changes when stable.

Debounce_toggle_tb.v

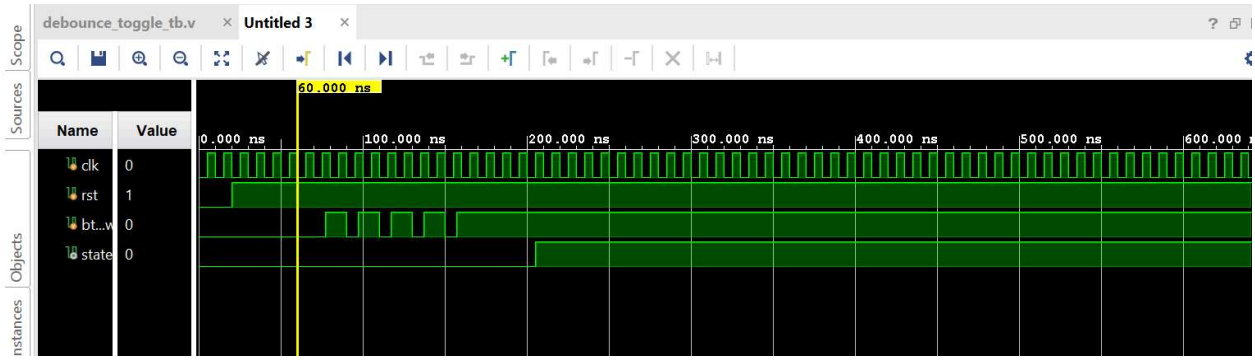
```
module debounce_toggle_tb;
    reg clk;
    reg rst;
    reg btn_raw;
    wire state;
    toggle uut (
        .clk(clk),
        .rst(rst),
        .btn_raw(btn_raw),
        .state(state)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    task simulate_bounce;
        input level;           // desired steady state after bounce (0 or 1)
        input [3:0] count;     // glitch amt
        integer i;
        begin
            for (i = 0; i < count; i = i + 1) begin
                btn_raw = ~level;
                #7;
                btn_raw = level;
                #13;
            end
        end
```

```
    initial begin
        rst = 0;
        btn_raw = 0;
        #20;
        rst = 1;

        $display("Time\t\tbtn_raw\tstate");
        $monitor("%0t\t%b\t%b", $time, btn_raw, state);

        #50;

        // Simulate a noisy button press (with bounce)
        $display("Press with bounce...");
        simulate_bounce(1, 5);  // 5 rising glitches
        btn_raw = 1;
        #500;
        btn_raw = 0;
        #500;

        $display("Press again with bounce...");
        simulate_bounce(1, 3);
        btn_raw = 1;
        #500;
        btn_raw = 0;
        #500;

        $display("Clean press...");
        btn_raw = 1;
```

```
        #500;

        $stop;
    end

endmodule
```

The debounce_toggle_tb module is a testbench that verifies the behavior of the toggle module under realistic button-press scenarios, including bounce. It generates a clock, controls reset and raw button inputs, and uses a task (simulate_bounce) to simulate noisy button presses with temporary glitches. It observes the toggle output (state) and prints changes over time to ensure

that only clean rising edges cause the state to flip, validating the effectiveness of the debounce and toggle logic.




```
Time resolution is 1 ps
Time        btn_raw state
20000   0   0
Press with bounce...
77000   1   0
90000   0   0
97000   1   0
110000  0   0
117000  1   0
130000  0   0
137000  1   0
150000  0   0
157000  1   0
205000  1   1
670000  0   1
```

shamt_decoder.v

```
module ShAmt_decoder(
    input BtnC, BtnU, BtnD, rst, clk,
    output reg [3:0] ShAmt
    );
    reg [1:0] upper;
    wire state;

    debounce btnU(.clk(clk), .btn_in(BtnU), .btn_clean(state));

    always@(posedge state or negedge rst) begin
        if (!rst) begin
            upper <= 0;
        end
        else begin
            upper = upper + 1;
            #2;
        end
    end

    always@(*) begin
        ShAmt[0] <= BtnD;
        ShAmt[1] <= BtnC;
        ShAmt[3:2] <= upper;
    end

endmodule
```

The ShAmt_decoder module takes input from three buttons (BtnU, BtnC, BtnD) and a clock and reset signal to generate a 4-bit shift amount (ShAmt). It uses a debounced version of BtnU to increment the upper two bits (ShAmt[3:2]) on each press, and directly maps BtnD and BtnC to the lower two bits. The shift amount is user-controlled via button presses and updated on the rising edge of the cleaned BtnU signal or asynchronously cleared by reset.

clock_divider_fixed.v

```
module clock_divider_fixed(
    input clk,
    input rst,
    output reg clk_2hz,
    output reg clk_1khz
    );
    parameter DIV_VALUE = 26'd50000000;
    reg [25:0] Hz;
    reg [25:0] Khz;

    always@(posedge clk or negedge rst) begin
        if (rst == 0) begin
            Hz <= 0;
            Khz <=0;
        end
        else begin
            Hz = Hz + 1'b1;
            Khz = Khz + 1'b1;
            if (Hz == DIV_VALUE) begin
                Hz = 0;
                clk_2hz = ~clk_2hz;
            end
            if (Khz == DIV_VALUE/500) begin
                Khz = 0;
                clk_1khz = ~clk_1khz;
            end
        end
    end
endmodule
```

The clock_divider_fixed module creates two slower clock signals from a high-frequency system clock. It uses two counters (Hz and Khz) to count up to predefined thresholds (DIV_VALUE and DIV_VALUE/500), toggling the output clocks when the thresholds are reached. The clock division and toggling occur synchronously with the system clock and can be reset asynchronously.

barrel_shifter16.v

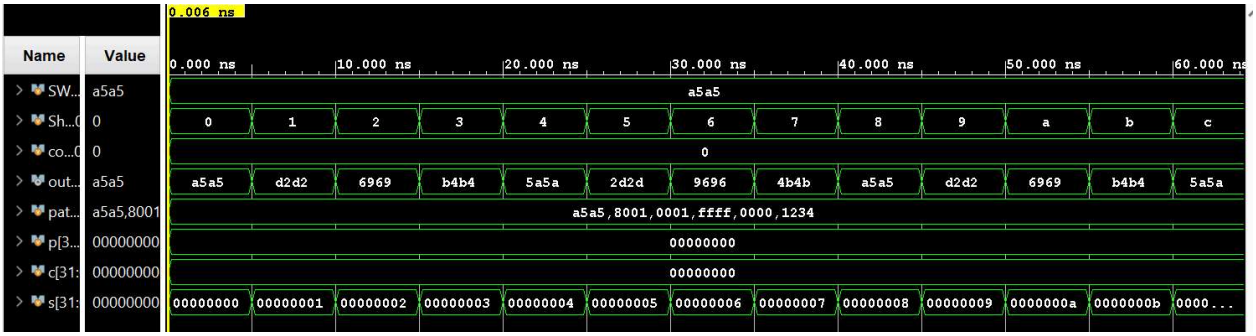
```
module barrel_shifter16(
    input [1:0] control,
    input [15:0] SW,
    input [3:0] ShAmt,
    output reg [15:0] out
    );
    wire [15:0] out_layer0, out_layer1, out_layer2, out_layer3;
    reg [15:0] calc;

    genvar i;
    generate
        for (i=0;i<16;i=i+1) begin : layer_0
            mux_2x1 mux_m0 (.a(SW[(i+(ShAmt[3]*4'd8))%16]), .b(SW[(i+(ShAmt[3]*4'd8))%16]), .sel(control[0]), .out(out_layer0[i]));
        end
        for (i=0;i<16;i=i+1) begin : layer_1
            mux_2x1 mux_n0 (.a(out_layer0[(i+(ShAmt[2]*4'd4))%16]), .b(out_layer0[(i+(16-(ShAmt[2]*4'd4)))%16]), .sel(control[0]), .out(out_layer1[i]));
        end
        for (i=0;i<16;i=i+1) begin : layer_2
            mux_2x1 mux_x0 (.a(out_layer1[(i+(ShAmt[1]*4'd2))%16]), .b(out_layer1[(i+(16-(ShAmt[1]*4'd2)))%16]), .sel(control[0]), .out(out_layer2[i]));
        end
        for (i=0;i<16;i=i+1) begin : layer_3
            mux_2x1 mux_y0 (.a(out_layer2[(i+(ShAmt[0]*4'd1))%16]), .b(out_layer2[(i+(16-(ShAmt[0]*4'd1)))%16]), .sel(control[0]), .out(out_layer3[i]));
        end
    endgenerate

    always@(*) begin
        calc = 16'b1111111111111111;
        case (control)
            2'b10: out = out_layer3 & (calc << ShAmt);
            2'b11: out = out_layer3 & (calc >> ShAmt);
            default: out = out_layer3;
        endcase
    end

endmodule
```

The code implements a 16-bit barrel shifter with rotation and shifting both directions through the control input. Control[0] is for direction , 0 for left, 1 for right, Control[1] is for rotate or shift, 0 for rotate, 1 logical shift . It uses four layers of 2:1 multiplexers to conditionally shift the sw input by bits 1, 2, 4, 8 bits based on shAmt[3:0] .

Barrel_shifter16_tb

```
module barrel_shifter16_tb;
    reg [15:0] SW;
    reg [3:0] ShAmt;
    reg [1:0] control;
    wire [15:0] out;
    barrel_shifter16 uut (
        .control(control),
        .SW(SW),
        .ShAmt(ShAmt),
        .out(out)
    );

    reg [15:0] patterns [0:5];
    integer p, c, s;

    initial begin
        patterns[0] = 16'hA5A5;
        patterns[1] = 16'h8001;
        patterns[2] = 16'h0001;
        patterns[3] = 16'hFFFF;
        patterns[4] = 16'h0000;
        patterns[5] = 16'h1234;
        $display("Start");
        $display("Time\tCtrl\tShAmt\tSW\t\tOut");

        for (p = 0; p < 6; p = p + 1) begin
            SW = patterns[p];

            for (c = 0; c < 4; c = c + 1) begin
                control = c[1:0];

                for (s = 0; s < 16; s = s + 1) begin
                    ShAmt = s[3:0];
                    #5;
                    $display("%4dns\t%2b\t%2d\t%h\t%h", $time, control, ShAmt, SW, out);
                end
            end
```

The testbench exhaustively tests the barrel_shifter16 module by sweeping through combinations of shmAt values 0-15, control values from 00 to 11 by rotating/shifting and going left/right, multiple patterns like 0xA5A5 and 0x8001.



mux_2x1.v

```
module mux_2x1(input a, input b, input sel, output reg out);
    always@(*) begin
        if (!sel) out = a;
        else out = b;
    end
endmodule
```

Standard 2x1 Mux as used in previous labs.

seg7_scan8.v

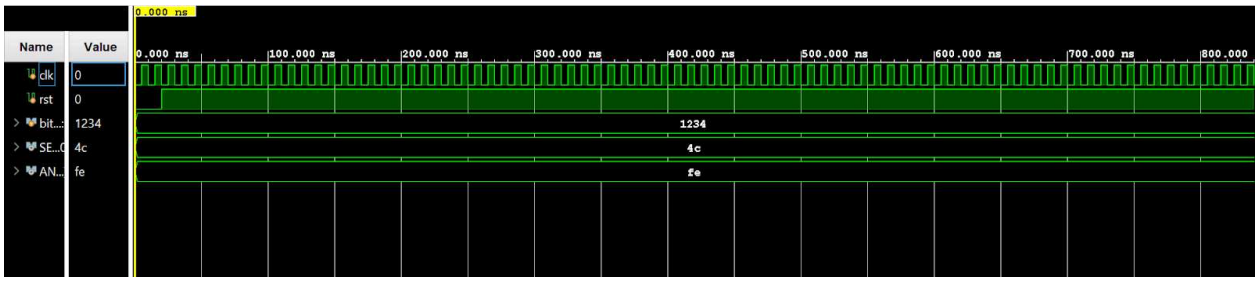
```
module seg7_scan8(
    input clk,
    input rst,
    input [15:0] bits,
    output reg [6:0] SEG,
    output [7:0]AN
    );
    reg [19:0] tmp;
    reg [3:0] digit;

    always@(digit)
        case(digit)
            4'd0:SEG=7'b0000001; 4'd1:SEG=7'b1001111; 4'd2:SEG=7'b0010010;
            4'd3:SEG=7'b0000110; 4'd4:SEG=7'b1001100; 4'd5:SEG=7'b0100100;
            4'd6:SEG=7'b0100000; 4'd7:SEG=7'b0001111; 4'd8:SEG=7'b0000000;
            4'd9:SEG=7'b0001100; 4'd10:SEG=7'b0001000;4'd11:SEG=7'b1100000;
            4'd12:SEG=7'b0110001;4'd13:SEG=7'b1000010;4'd14:SEG=7'b0110000;
            4'd15:SEG=7'b0111000;default:SEG=7'b1111111;
        endcase

    always@(posedge clk or negedge rst) begin
        if(!rst)
            tmp<=0;
        else
            tmp<=tmp+1;
    end

    wire [1:0] s = tmp[19:18];

    always@(s, bits)
        case (s)
            2'd0:digit=bits[3:0];2'd1:digit=bits[7:4];
            2'd2:digit=bits[11:8];2'd3:digit=bits[15:12];
            default:digit=4'b0000;
        endcase

    reg [7:0] AN_tmp;

    always@(s)
        case(s)
            2'd0:AN_tmp=8'b11111110; 2'd1:AN_tmp=8'b11111101;
            2'd2:AN_tmp=8'b11111011; 2'd3:AN_tmp=8'b11110111;
            default:AN_tmp=8'b11111111;
            //7-segment decoder for the first 3 lamps
    endcase

    assign AN=AN_tmp;
```

The code is 4-digit, time-multiplexed 7-segment display driver that uses a 20-bit counter (tmp) to determine which digit to display based on the upper bits tmp[19:18]. The bits input contains 4 hexadecimal digits (16 bits), and every 2 bits of s = tmp[19:18] selects one nibble of bits to show on the display. This selected nibble is decoded into 7-segment format (SEG) using a case statement. Another case statement generates the AN output to enable one of the four digits (active low). The display cycles through the digits rapidly, leveraging persistence of vision to appear as a steady 4-digit display.

Seg7_scan8_tb.v

```
module seg7_scan8_tb;
reg clk;
    reg rst;
    reg [15:0] bits;
    wire [6:0] SEG;
    wire [7:0] AN;

    seg7_scan8 uut (
        .clk(clk),
        .rst(rst),
        .bits(bits),
        .SEG(SEG),
        .AN(AN)
    );
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        rst = 0;
        bits = 16'h1234;

        #20;
        rst = 1;

        #200000;
        bits = 16'hABCD;
        #200000;
        bits = 16'hF00D;
        #200000;
        $stop;
    end
    initial begin
        $display("Time\t\tclk\trst\tbits\t\tSEG\t\tAN");
        $monitor("%0t\t%b\t%b\t%h\t%b\t%b", $time, clk, rst, bits, SEG, AN);
    end
```

The testbench simulates the seg7_scan8 module by generating a 100 MHz clock and applying a reset signal (rst). It sets various 16-bit values to the bits input (0x1234, 0xABCD, 0xF00D) to test different display outputs over time. The simulation runs for long periods to allow the internal counter to drive the digit selector (AN) and segment outputs (SEG). A $monitor displays key signal values at each simulation step, helping verify that each digit and its corresponding 7-segment pattern is selected and encoded correctly over time.



```
20000   0   1   1234    1001100 11111110
25000   1   1   1234    1001100 11111110
30000   0   1   1234    1001100 11111110
35000   1   1   1234    1001100 11111110
40000   0   1   1234    1001100 11111110
45000   1   1   1234    1001100 11111110
50000   0   1   1234    1001100 11111110
55000   1   1   1234    1001100 11111110
60000   0   1   1234    1001100 11111110
65000   1   1   1234    1001100 11111110
70000   0   1   1234    1001100 11111110
75000   1   1   1234    1001100 11111110
80000   0   1   1234    1001100 11111110
85000   1   1   1234    1001100 11111110
90000   0   1   1234    1001100 11111110
95000   1   1   1234    1001100 11111110
100000  0   1   1234    1001100 11111110
105000  1   1   1234    1001100 11111110
110000  0   1   1234    1001100 11111110
115000  1   1   1234    1001100 11111110
120000  0   1   1234    1001100 11111110
125000  1   1   1234    1001100 11111110
```

Vivado (LUTs, FFs, Power):

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	109	0	0	63400	0.17
LUT as Logic	109	0	0	63400	0.17
LUT as Memory	0	0	0	19000	0.00
Slice Registers	50	0	0	126800	0.04
Register as Flip Flop	50	0	0	126800	0.04
Register as Latch	0	0	0	126800	0.00
F7 Muxes	16	0	0	31700	0.05
F8 Muxes	0	0	0	15850	0.00

* Warning! LUT value is adjusted to account for LUT combining.

1.1 Summary of Registers by Type

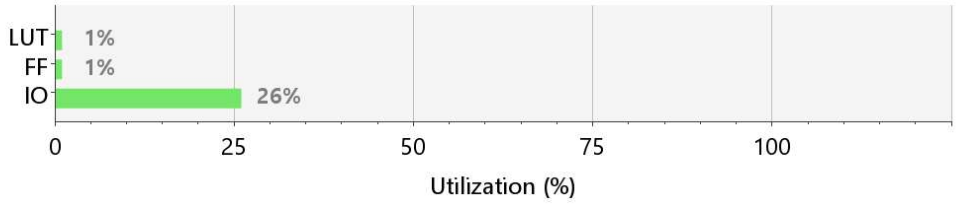
Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
30	Yes	-	Reset
0	Yes	Set	-
20	Yes	Reset	-

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	42	0	0	15850	0.26
SLICEL	32	0			
SLICEM	10	0			
LUT as Logic	109	0	0	63400	0.17
using O5 output only	0				
using O6 output only	99				
using O5 and O6	10				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
LUT as Shift Register	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	50	0	0	126800	0.04
Register driven from within the Slice	33				
Register driven from outside the Slice	17				
LUT in front of the register is unused	8				
LUT in front of the register is used	9				
Unique Control Sets	3		0	15850	0.02

* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more information regarding control sets.

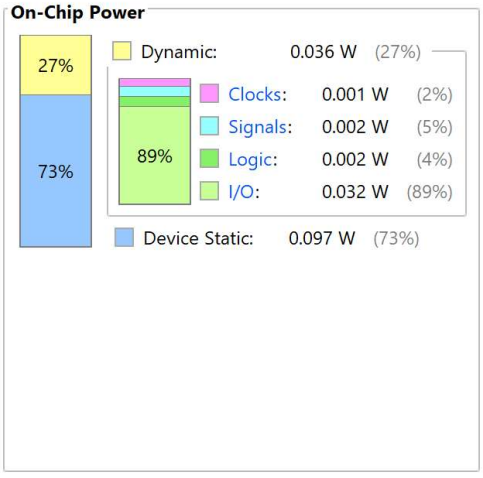
Resource	Utilization	Available	Utilization %
LUT	110	63400	0.17
FF	50	126800	0.04
IO	54	210	25.71



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.133 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (12.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



```
1. Summary
----------

+--------------------------+--------------+
| Total On-Chip Power (W)  | 0.133        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 0.036        |
| Device Static (W)        | 0.097        |
| Effective TJA (C/W)      | 4.6          |
| Max Ambient (C)          | 84.4         |
| Junction Temperature (C) | 25.6         |
| Confidence Level         | Low          |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
* Specify Design Power Budget using, set_operating_conditions -design_power_budget <value in Watts>
```

Video Link:
https://youtu.be/qzq43ESMjGI

Reflections:
Getting the barrel shifter to work was interesting to get going because the implemented blocks and arithmetic had to be tweaked multiple times to work. Initially, the concept of the barrel shifter had to be revisited in order to get the foundation of the code going. Overall, however, the lab went smoothly and the skills developed over the previous lab have become evident while writing this lab.

Partner Contributions:
The lab was 50/50. Heba initially started the code, working on some parts while Sean worked on his parts then each others work was crossed checked and edited. Lab report was collaborated on as well each taking different sections .