CalPolyPomona
College of Engineering

Lab 7 Report

by

Jonathan Huynh #016137719

Adam Godfrey #015981472

Instructor: Dr. Mohamed Aly

Class: ECE 3300L .E02-OU - Verilog Design

August 6th, 2025

Code with Explanation:

Barrel Shifter Top Module Code

```
module top_lab7(
    input wire clk,
    input wire btnc,
    input wire [15:0] sw,
    input wire [4:1] btn,
    output wire [6:0] seg,
    output wire [7:0] an,
    output wire [7:0] led
);
    wire rst = 1'b0;
    wire clk1k, clk2;
    clock_divider_fixed div(
        .clk(clk),
        .rst(rst),
        .clk_1kHz(clk1k),
        .clk_2Hz(clk2)
    );

    wire dir, rot;
    wire [1:0] shamt_low;
    wire [1:0] shamt_high;
    wire btnc_toggle;

    debounce_toggle d_dir (.clk(clk1k),
.btn_raw(btn[1]), .btn_toggle(dir));
    debounce_toggle d_rot (.clk(clk1k),
.btn_raw(btn[4]), .btn_toggle(rot));
    debounce_toggle d_s0 (.clk(clk1k),
.btn_raw(btn[2]), .btn_toggle(shamt_low[0]));
    debounce_toggle d_s1 (.clk(clk1k),
.btn_raw(btn[3]), .btn_toggle(shamt_low[1]));
    debounce_toggle d_btnc(.clk(clk1k),
.btn_raw(btnc), .btn_toggle(btnc_toggle));

    shamt_counter counter(
        .clk(clk1k),
        .rst(rst),
        .btnc(btnc_toggle),
        .shamt_high(shamt_high)
    );

    wire [3:0] shamt = {shamt_high,
shamt_low};

    wire [15:0] barrel_out;
    barrel_shifter16 shifter(
```

This module, top_lab7, implements a 16-bit barrel shifter on an FPGA. The user controls the amount and direction of shift using pushbuttons and switches. A clock divider generates 1kHz and 2Hz clocks for stable button detection using debounce toggles. Two buttons select the lower 2 bits of the shift amount, while a counter controlled by the center button (btnc) sets the upper 2 bits. The full 4-bit shift amount (shamt) and control flags are passed to a barrel_shifter16, which shifts the input from switches and displays the result on a 7-segment display. The shift configuration is also shown via onboard LEDs.

```
        .data_in(sw),
        .shamt(shamt),
        .dir(dir),
        .rotate(rot),
        .data_out(barrel_out)
    );

    seg7_scan8 scanner(
        .clk(clk1k),
        .rst(rst),
        .data(barrel_out),
        .seg(seg),
        .an(an)
    );

    assign led = {shamt, dir, rot};
endmodule
```

Barrel Shifter

```
module barrel_shifter16(
    input wire [15:0] data_in,
    input wire [3:0] shamt,
    input wire dir,
    input wire rotate,
    output wire [15:0] data_out
);
    wire [15:0] s_stage1, s_stage2, s_stage3,
s_stage4;

    assign s_stage1 = (shamt[0]) ?
        (rotate ? (dir ? {data_in[0], data_in[15:1]}
: {data_in[14:0], data_in[15]}) :
                  (dir ? {1'b0, data_in[15:1]}       :
{data_in[14:0], 1'b0})) :
        data_in;

    assign s_stage2 = (shamt[1]) ?
        (rotate ? (dir ? {s_stage1[1:0],
s_stage1[15:2]} : {s_stage1[13:0],
s_stage1[15:14]}) :
                  (dir ? {2'b00, s_stage1[15:2]}
: {s_stage1[13:0], 2'b00})) :
        s_stage1;

    assign s_stage3 = (shamt[2]) ?
        (rotate ? (dir ? {s_stage2[3:0],
s_stage2[15:4]} : {s_stage2[11:0],
s_stage2[15:12]}) :
```

The barrel_shifter16 module performs a 16-bit shift or rotate on the input data (data_in) by an amount specified by a 4-bit shift amount (shamt). The shifting behavior is controlled by two inputs. The input dir determines the direction (1 = right, 0 = left), and input rotate selects between rotation (wrap-around) or logical shift (zeros shifted in). The shift is implemented in four stages (1, 2, 4, and 8-bit steps), where each stage conditionally applies a shift if the corresponding bit in shamt is set. The final output after all stages is assigned to data_out, giving a fast and flexible way to shift or rotate the data in either direction by 0 - 15 bits.

```
                (dir ? {4'b0000, s_stage2[15:4]}
: {s_stage2[11:0], 4'b0000})) :
        s_stage2;

    assign s_stage4 = (shamt[3]) ?
        (rotate ? (dir ? {s_stage3[7:0],
s_stage3[15:8]} : {s_stage3[7:0],
s_stage3[15:8]}) :
                (dir ? {8'b00000000,
s_stage3[15:8]}   : {s_stage3[7:0],
8'b00000000})) :
        s_stage3;

    assign data_out = s_stage4;
endmodule
```

Clock Divider Code

```
module clock_divider_fixed(
    input wire clk,
    input wire rst,
    output wire clk_1kHz,
    output wire clk_2Hz
);

    parameter DIV_1KHZ = 50000;
    parameter DIV_2HZ = 25000000;

    reg [15:0] cnt1k = 0;
    reg clk1k = 0;

    reg [25:0] cnt2 = 0;
    reg clk2 = 0;

    always @(posedge clk or posedge rst)
begin
        if (rst) begin
            cnt1k <= 0;
            clk1k <= 0;
        end else if (cnt1k == DIV_1KHZ-1) begin
            cnt1k <= 0;
            clk1k <= ~clk1k;
        end else begin
            cnt1k <= cnt1k + 1;
        end
    end

    always @(posedge clk or posedge rst)
begin
```

The clock_divider_fixed module takes a fast input clock (clk) and generates two slower clocks. A 1kHz clock (clk_1kHz) and a 2Hz clock (clk_2Hz). It uses two counters. One (cnt1k) counts up to 49,999 to toggle the 1kHz output, and the other (cnt2) counts up to 24,999,999 to toggle the 2Hz output. Each time the respective count is reached, the output clock is flipped (toggled), effectively dividing the input clock frequency. The module also includes a reset (rst) input to reset both counters and clocks.

```
        if (rst) begin
            cnt2 <= 0;
            clk2 <= 0;
        end else if (cnt2 == DIV_2HZ-1) begin
            cnt2 <= 0;
            clk2 <= ~clk2;
        end else begin
            cnt2 <= cnt2 + 1;
        end
    end

    assign clk_1kHz = clk1k;
    assign clk_2Hz = clk2;
endmodule
```

Debounce Toggle

```
module debounce_toggle(
    input wire clk,
    input wire btn_raw,
    output reg btn_toggle = 0
);
    reg [3:0] shift = 0;
    reg state = 0;

    always @(posedge clk) begin
        shift <= {shift[2:0], btn_raw};
        if (shift == 4'b1111 && !state) begin
            state <= 1;
            btn_toggle <= ~btn_toggle;
// Toggle the output
        end else if (shift == 4'b0000 && state)
begin
            state <= 0;
        end
    end
endmodule
```

The debounce_toggle module cleans up a noisy button signal (btn_raw) using a simple debounce filter and generates a toggle output (btn_toggle) on each clean button press. It samples the button input every clock cycle and stores the last 4 samples in a shift register. When all 4 samples are high (4'b1111) and the internal state is low, it means the button was pressed stably, so it flips (toggles) the output and sets the state to high. When all 4 samples are low (4'b0000), it resets the state, allowing a new toggle on the next press. This ensures each press only toggles once, ignoring bouncing.

ShAmt Counter

```
module shamt_counter(
    input wire clk,
    input wire rst,
    input wire btnc,
    output reg [1:0] shamt_high
);
    reg btnc_prev;
    always @(posedge clk or posedge rst)
begin
        if (rst) begin
            shamt_high <= 2'b00;
            btnc_prev <= 1'b0;
        end else begin
            btnc_prev <= btnc;
            if (~btnc_prev & btnc) begin
                shamt_high <= (shamt_high ==
2'd3) ? 2'd0 : shamt_high + 1;
            end
    end
endmodule
```

The shamt_counter module generates the upper 2 bits (shamt_high) of a 4-bit shift amount, cycling through values 0 to 3. It detects rising edges of a debounced center button (btnc) using the btnc_prev register. On each rising edge (when the button changes from 0 to 1), it increments shamt_high by 1, wrapping back to 0 after reaching 3. The rst input resets both the counter and the previous button state. This allows a user to control how much to shift (in steps of 4 bits) with repeated button presses.

7 Segment Hex Digit

```
module hex_to_7seg(
    input wire [3:0] hex,
    output reg [6:0] seg
    always @(*) begin
        case (hex)
            4'h0: seg = 7'b1000000;
            4'h1: seg = 7'b1111001;
            4'h2: seg = 7'b0100100;
            4'h3: seg = 7'b0110000;
            4'h4: seg = 7'b0011001;
            4'h5: seg = 7'b0010010;
            4'h6: seg = 7'b0000010;
            4'h7: seg = 7'b1111000;
            4'h8: seg = 7'b0000000;
            4'h9: seg = 7'b0011000;
            4'hA: seg = 7'b0001000;
            4'hB: seg = 7'b0000011;
            4'hC: seg = 7'b1000110;
            4'hD: seg = 7'b0100001;
            4'hE: seg = 7'b0000110;
            4'hF: seg = 7'b0001110;
        endcase
    end
endmodule
```

The hex_to_7seg module converts a 4-bit hexadecimal input (hex) into the corresponding 7-segment display pattern (seg). Each case in the case statement maps a hex value (0–F) to a 7-bit code that turns on the appropriate segments to display that digit or letter on a common anode 7-segment display.

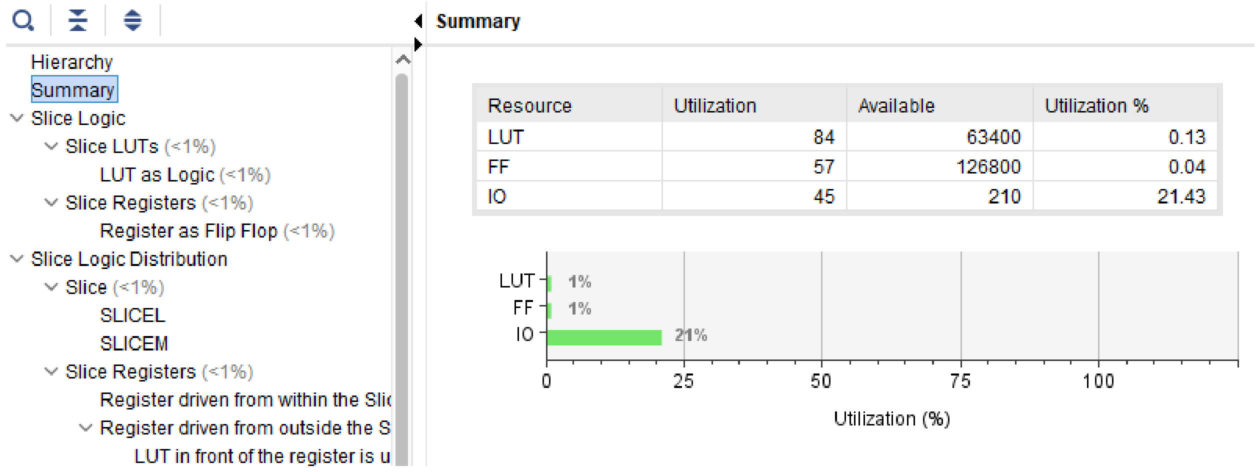
7 Segment 8 Bit Scanner

```
module seg7_scan8(
    input wire clk,
    input wire rst,
    input wire [15:0] data,
    output wire [6:0] seg,
    output reg [7:0] an
);
    reg [2:0] scan;
    wire [3:0] nibble;

    assign nibble = (scan == 3'd0) ? data[3:0] :
                    (scan == 3'd1) ? data[7:4] :
                    (scan == 3'd2) ? data[11:8] :
                    (scan == 3'd3) ? data[15:12] :
                    4'b0000;

    hex_to_7seg h2s(.hex(nibble), .seg(seg));

    always @(posedge clk or posedge rst)
begin
        if (rst) begin
            scan <= 3'd0;
            an <= 8'b11111111;
        end else begin
            scan <= (scan == 3'd3) ? 3'd0 : scan +
1;
            case (scan)
                3'd0: an <= 8'b11111101;
                3'd1: an <= 8'b11111011;
                3'd2: an <= 8'b11110111;
                3'd3: an <= 8'b11111110;
                default: an <= 8'b11111111;
            endcase
        end
    end
endmodule
```

The seg7_scan8 module displays a 16-bit value (data) across 4 digits of an 8-digit 7-segment display using time-multiplexing. It cycles through the lower 4 digits (scan = 0 to 3), extracting 4-bit nibbles and converting each to a 7-segment code using the hex_to_7seg module. The scan counter increments on each clock pulse (clk), enabling one digit at a time via the active-low an output. This creates the illusion of all digits being lit simultaneously by refreshing them rapidly. On reset (rst), scanning is reset and all digits are turned off. Only digits 0 to 3 are used, while digits 4 to 7 remain off.

Screenshot Proofs:

Resource Utilization Table:

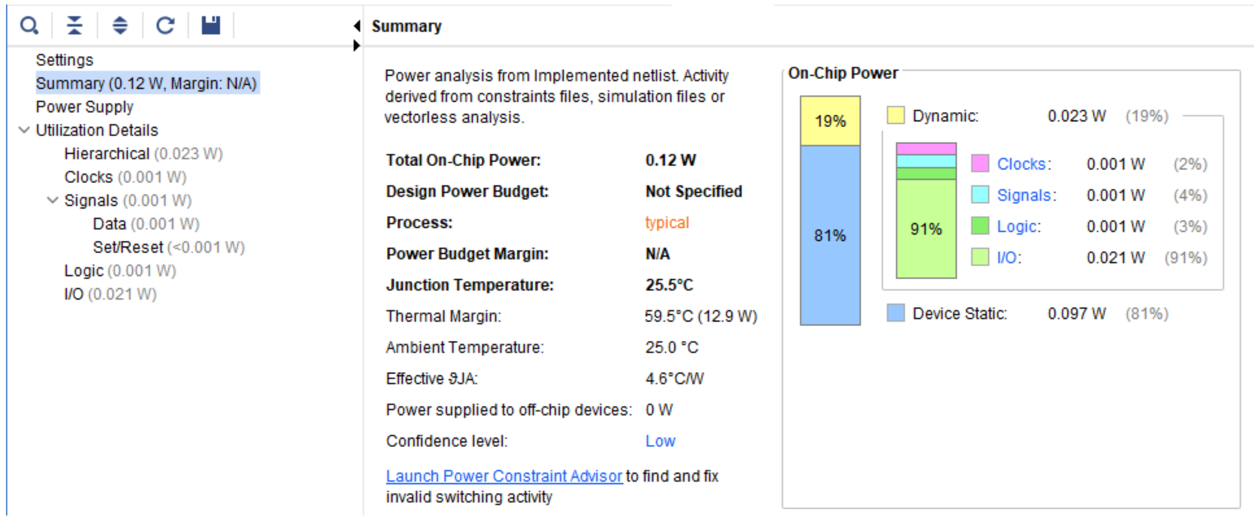


Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Logic (<1%)
Slice Registers (<1%)
Register as Flip Flop (<1%)
Slice Logic Distribution
Slice (<1%)
SLICEL
SLICEM
Slice Registers (<1%)
Register driven from within the Slic
Register driven from outside the S
LUT in front of the register is u

Summary

Resource	Utilization	Available	Utilization %
LUT	84	63400	0.13
FF	57	126800	0.04
IO	45	210	21.43

Power Utilization:



Settings
Summary (0.12 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (0.023 W)
Clocks (0.001 W)
Signals (0.001 W)
Data (0.001 W)
Set/Reset (<0.001 W)
Logic (0.001 W)
I/O (0.021 W)

Summary

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

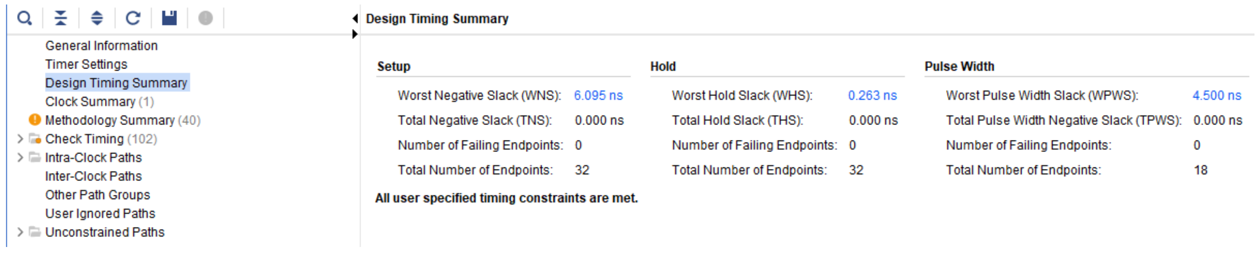
Total On-Chip Power:	**0.12 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (12.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		0.023 W	(19%)
	Clocks:	0.001 W	(2%)
	Signals:	0.001 W	(4%)
	Logic:	0.001 W	(3%)
	I/O:	0.021 W	(91%)
Device Static:		0.097 W	(81%)

Timing Summary:



General Information
Timer Settings
Design Timing Summary
Clock Summary (1)
Methodology Summary (40)
Check Timing (102)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths

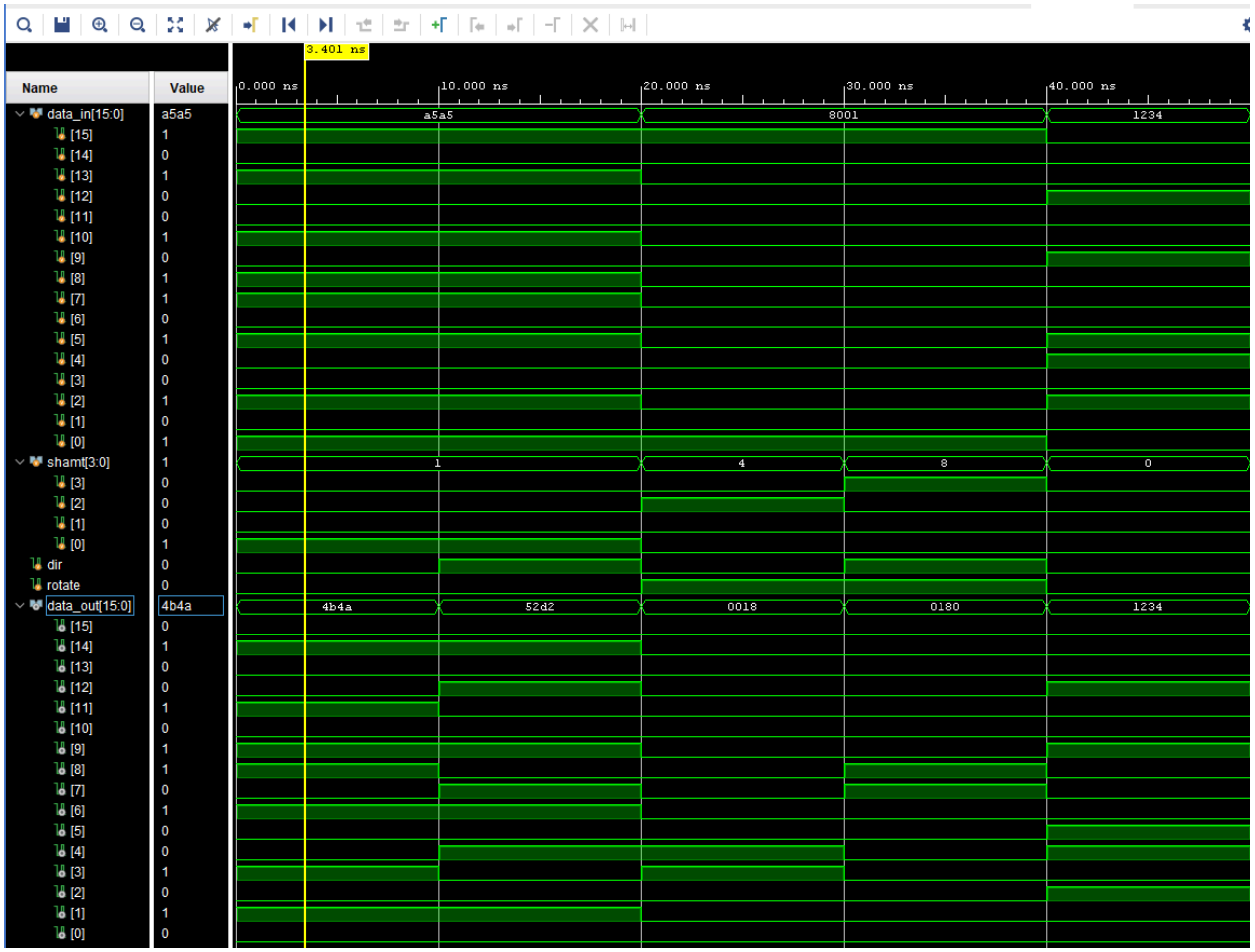
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.095 ns	Worst Hold Slack (WHS):	0.263 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32	Total Number of Endpoints:	18

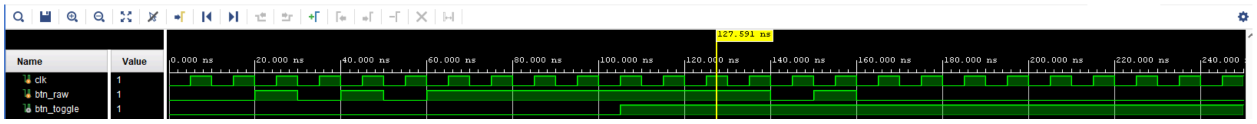
All user specified timing constraints are met.

Simulation Waveforms:

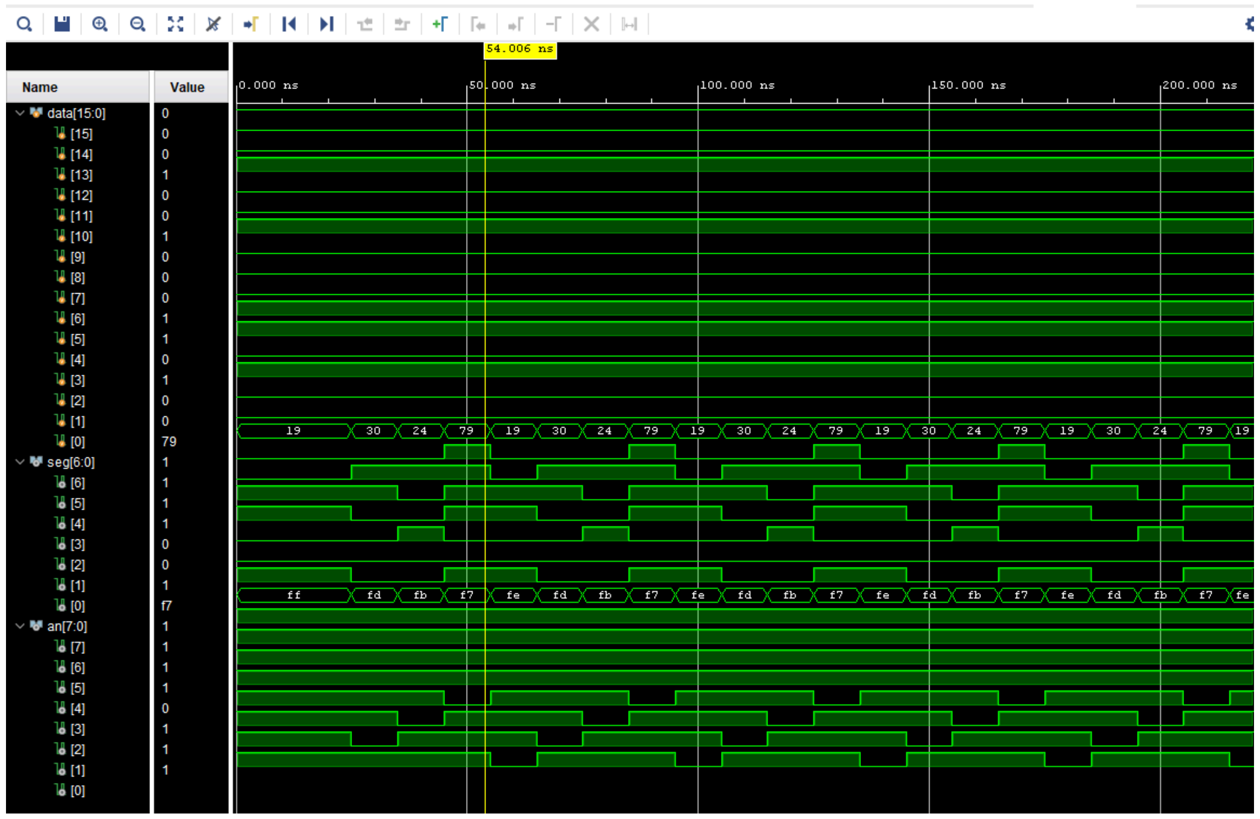
Barrel Shifter



Debounce Module



Seg7_Scan8



Partner Contributions:

Team Member	Contribution	% Effort
Jonathan Huynh	**Code:** [7 Segment Hex/Scanner, ShAmt Counter, Debounce] **Testbench:** [Barrel Shifter] **Additional:** Synthesis/Implementation and Demo	50%
Adam Godfrey	**Code:** [Clock Divider, Top Module, Barrel Shifter] **Testbench:** [Debounce, 7 Segment] **Additional:** Simulation and Lab Report	50%