CalPolyPomona
College of Engineering

Lab 6 Report

by

Jonathan Huynh #016137719

Adam Godfrey #015981472

Instructor: Dr. Mohamed Aly

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

July 28th, 2025

Code with Explanation:

BCD Up/Down Counter on 7-Segment Display Top Module Code

```
`timescale 1ns / 1ps
module top_lab6 (
    input wire CLK,
    input wire BTN0,
    input wire [8:0] SW,
    output wire [6:0] SEG,
    output wire [7:0] AN,
    output wire [7:0] LED
);
    wire [31:0] tick_bits;
    wire clk_div;
    wire [3:0] count_a;
    wire [3:0] count_b;
    wire [3:0] ctrl_nibble;
    wire [7:0] alu_output;

    clock_divider clk_div_inst (
        .base_clk(CLK),
        .reset_n(BTN0),
        .sel(SW[4:0]),
        .clk_div(clk_div),
        .tick_bits(tick_bits)
    );

    control_decoder ctrl_decoder_inst (
        .control_switches(SW[8:5]),
        .ctrl_nibble(ctrl_nibble)
    );

    bcd_counter counter_a (
        .clk_div(clk_div),
        .reset_n(BTN0),
        .count_up(SW[7]),
        .bcd_out(count_a)
    );

    bcd_counter counter_b (
        .clk_div(clk_div),
        .reset_n(BTN0),
        .count_up(SW[8]),
        .bcd_out(count_b)
    );

    alu alu_inst (
        .A_val(count_a),
        .B_val(count_b),
```

We wrote this top-level Verilog module, *top_lab6*, to integrate several submodules into a cohesive digital system that runs on the Nexys A7 100T. The module accepts a clock input (**CLK**), a reset button (**BTN0**), and a set of switches (**SW**) to control the functionality. We coded this to demonstrate the interaction of clock division, control decoding, BCD counting, ALU operations, and 7-segment display scanning. Each submodule plays a specific role, and this top module wires them all together.

We coded the *clock_divider* to scale down the input clock frequency based on user-selected settings via **SW**[4:0], producing a slower clock (**clk_div**) and a 32-bit tick pattern (**tick_bits**). This slower clock is used to drive two instances of *bcd_counter*, which we wrote to count up based on two different switch bits (**SW**[7] and **SW**[8]). Meanwhile, the *control_decoder* interprets bits **SW**[8:5] to generate a nibble (**ctrl_nibble**) for controlling the display and ALU behavior. We coded this to allow the user to dynamically adjust counting and display logic using switches.

The alu module takes the outputs from the two BCD counters and performs a logic operation determined by **SW**[6:5], producing an 8-bit result that we send to the LEDs and the display. We wrote the *seg7_scan* module to show the high and low nibbles of this result on a two-digit 7-segment display, while

.control_bits(SW[6:5]),
.result_out(alu_output)
);
 	
seg7_scan display_inst (
.clk_fast(CLK),
.rst_n(BTN0),
.lower_digit(alu_output[3:0]),
.upper_digit(alu_output[7:4]),
.ctrl_nibble(ctrl_nibble),
.seg_pins(SEG),
.dp(),
.anodes(AN)
);
 	
assign LED = alu_output;
assign AN[7:3] = 5'b11111;
 	
endmodule
	keeping the other five digits disabled using **AN**[7:3] = 5'b11111. We coded this setup to give clear visual feedback on both the ALU computation and its control logic, making the system easy to test and demonstrate.

ALU	
`timescale 1ns / 1ps module alu(input wire [3:0] A_val, input wire [3:0] B_val, input wire [1:0] control_bits, output reg [7:0] result_out); always @(*) begin case (control_bits) 2'b00: result_out = A_val + B_val; 2'b01: result_out = A_val - B_val; default: result_out = 8'd0; endcase end endmodule	We coded this *alu* module to perform basic arithmetic between two 4-bit inputs, **A_val** and **B_val**, based on a 2-bit control signal. We wrote this because we needed a simple way to calculate either the sum or difference of two counters in our top-level design. The **always @(*)** block ensures the logic is purely combinational, updating the 8-bit **result_out** instantly based on the selected operation. Addition for 00, subtraction for 01, and zero for all other cases to keep the behavior predictable.

BCD Up/Down Counter Code	

`timescale 1ns / 1ps
module bcd_counter(
input wire clk_div,
input wire reset_n,
input wire count_up,
output reg [3:0] bcd_out
);
always @(posedge clk_div or negedge reset_n) begin
if (!reset_n)
bcd_out <= 0;
else if (count_up)
bcd_out <= (bcd_out == 9) ? 0 : bcd_out + 1;
else
bcd_out <= (bcd_out == 0) ? 9 : bcd_out - 1;
end
endmodule
	We coded this *bcd_counter* module to produce a 4-bit decimal counter that counts up or down between 0 and 9 based on the **count_up** input. The counter updates on the rising edge of a divided clock and resets to zero when **reset_n** is low. We designed it to wrap from 9 to 0 when counting up, and from 0 to 9 when counting down, ensuring the output always stays within valid BCD range.

Clock Divider Code	

`timescale 1ns / 1ps
module clock_divider(
input wire base_clk,
input wire reset_n,
reset signal
input wire [4:0] sel,
output wire clk_div, /
output reg [31:0] tick_bits
);
always @(posedge base_clk or negedge reset_n) begin
if (!reset_n)
tick_bits <= 32'b0;
else
tick_bits <= tick_bits + 1;
end
assign clk_div = tick_bits[sel];
endmodule
	We coded this *clock_divider* module to generate a slower, selectable clock signal (**clk_div**) by incrementing a 32-bit counter (**tick_bits**) on each pulse of the base clock. We wrote this because other modules in the system, like the counters and display, require a slower clock for proper operation and visibility. The **sel** input lets us choose which bit of the counter to output as the divided clock, allowing easy adjustment of speed. When **reset_n** is low, the counter resets to zero, ensuring consistent behavior on startup.

Control Decoder	

`timescale 1ns / 1ps
module control_decoder(
  input wire [3:0] control_switches,
  output wire [3:0] ctrl_nibble
);
  assign ctrl_nibble = control_switches;
endmodule
	We coded this *control_decoder* module to pass the 4-bit **control_switches** input directly to the **ctrl_nibble** output without modification. We wrote this because we needed a clean and modular way to route control signals, such as those for display formatting or operation selection, from external switches into other parts of the system.

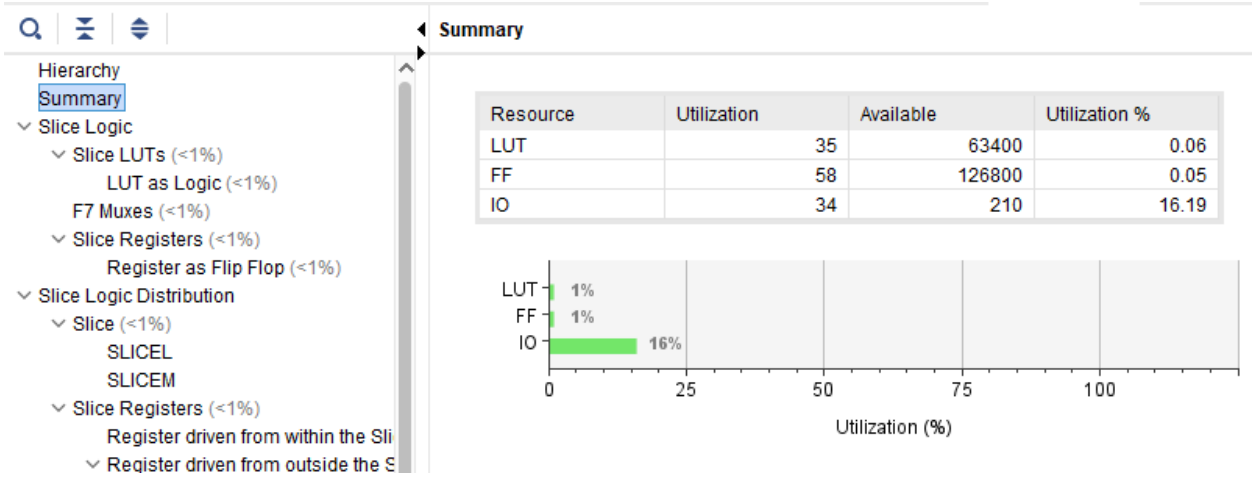
7 Segment Code	

`timescale 1ns / 1ps
module seg7_scan (
  input wire clk_fast,
  input wire rst_n,
  input wire [3:0] lower_digit,
  input wire [3:0] upper_digit,
  input wire [3:0] ctrl_nibble,
  output reg [6:0] seg_pins,
  output wire dp,
  output reg [2:0] anodes
);
 	
  reg [15:0] refresh_counter = 0;
  reg [1:0] scan = 0;
  wire [3:0] current_digit;
 	
  always @(posedge clk_fast) begin
  refresh_counter <= refresh_counter + 1;
  end
 	
  always @(posedge refresh_counter[15])
begin
  scan <= scan + 1;
  end
 	
  assign current_digit = (scan == 2'd0) ?
lower_digit :
  (scan == 2'd1) ? upper_digit
: ctrl_nibble;
 	
  always @(*) begin
	We coded this *seg7_scan* module to control a 3-digit 7-segment display by rapidly cycling through and displaying **lower_digit**, **upper_digit**, and **ctrl_nibble**. We wrote this because we needed a compact way to multiplex multiple digits on limited hardware, using time-division to refresh one digit at a time. A refresh counter and 2-bit scan register determine which digit to display and which anode to activate, while a case block maps each 4-bit value to its corresponding 7-segment LED pattern. This design lets us show three hexadecimal digits clearly with minimal wiring and efficient timing.

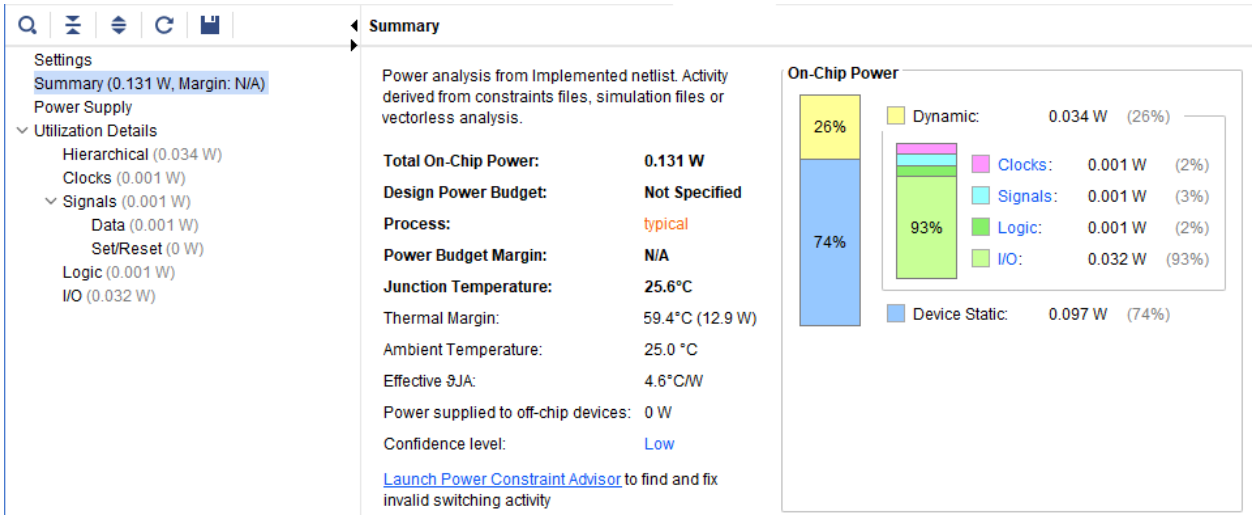
```
        case (scan)
            2'd0: anodes = 3'b110;
            2'd1: anodes = 3'b101;
            2'd2: anodes = 3'b011;
            default: anodes = 3'b111;
        endcase

        case (current_digit)
            4'h0: seg_pins = 7'b1000000;
            4'h1: seg_pins = 7'b1111001;
            4'h2: seg_pins = 7'b0100100;
            4'h3: seg_pins = 7'b0110000;
            4'h4: seg_pins = 7'b0011001;
            4'h5: seg_pins = 7'b0010010;
            4'h6: seg_pins = 7'b0000010;
            4'h7: seg_pins = 7'b1111000;
            4'h8: seg_pins = 7'b0000000;
            4'h9: seg_pins = 7'b0010000;
            4'hA: seg_pins = 7'b0001000;
            4'hb: seg_pins = 7'b0000011;
            4'hC: seg_pins = 7'b1000110;
            4'hd: seg_pins = 7'b0100001;
            4'hE: seg_pins = 7'b0000110;
            4'hF: seg_pins = 7'b0001110;
            default: seg_pins = 7'b1111111;
        endcase
    end
endmodule
```

Screenshot Proofs:

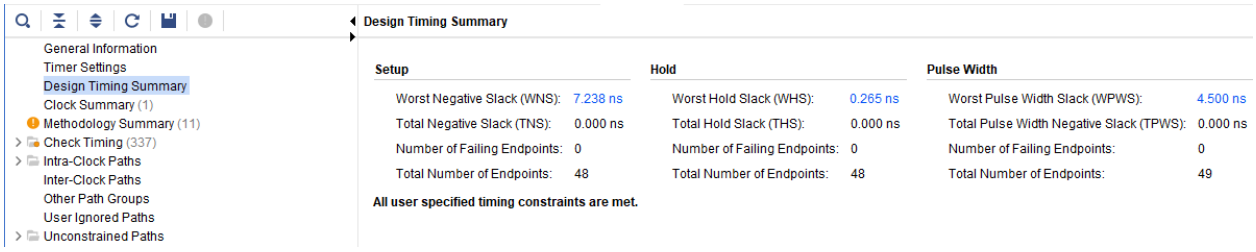
Resource Utilization Table:



Power Utilization:

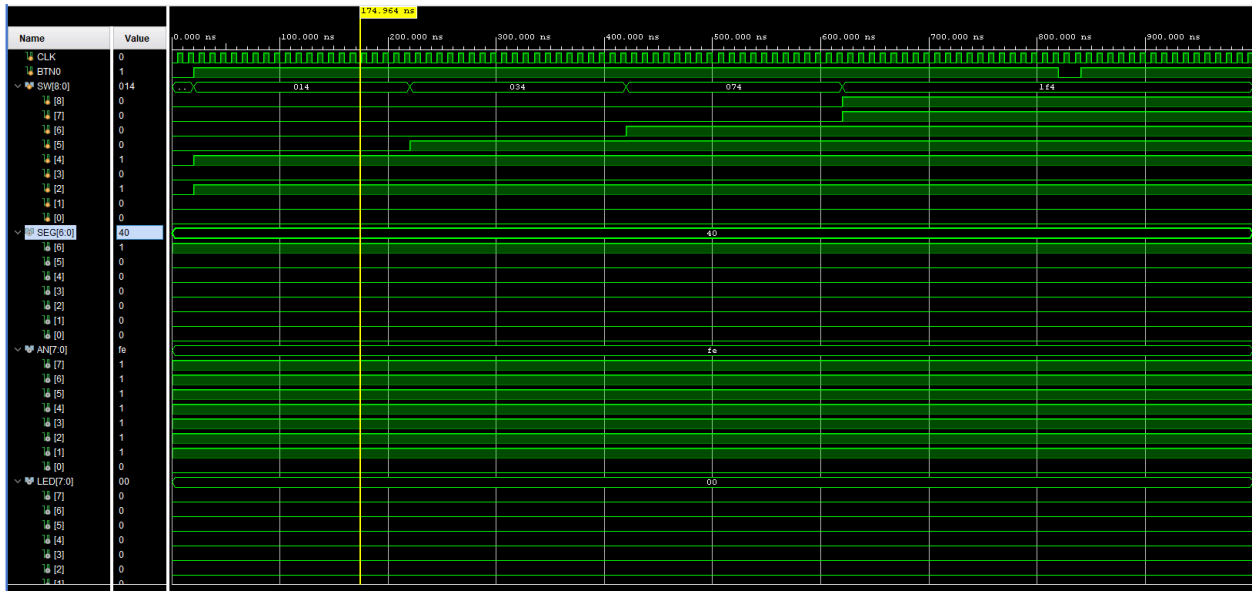


Timing Summary:

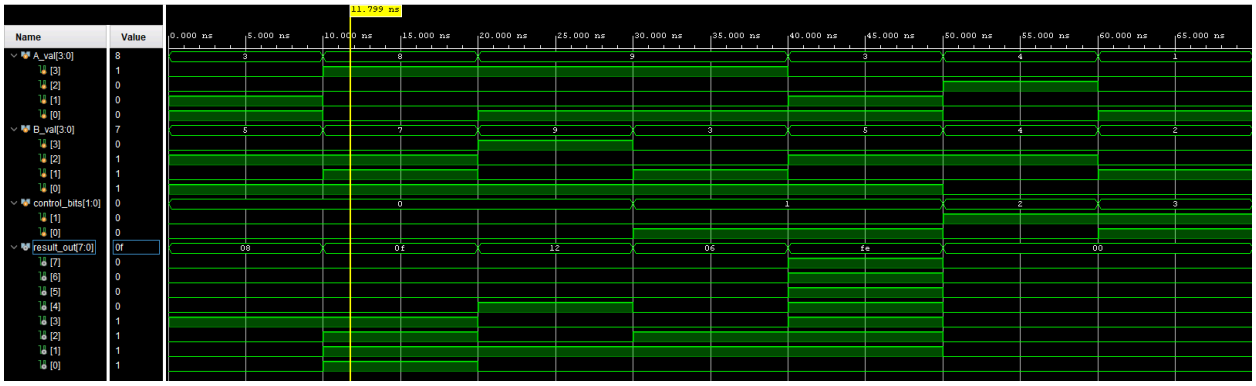


Simulation Waveforms:

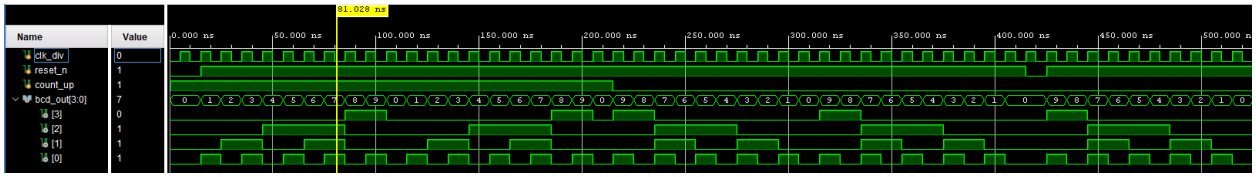
Top Module



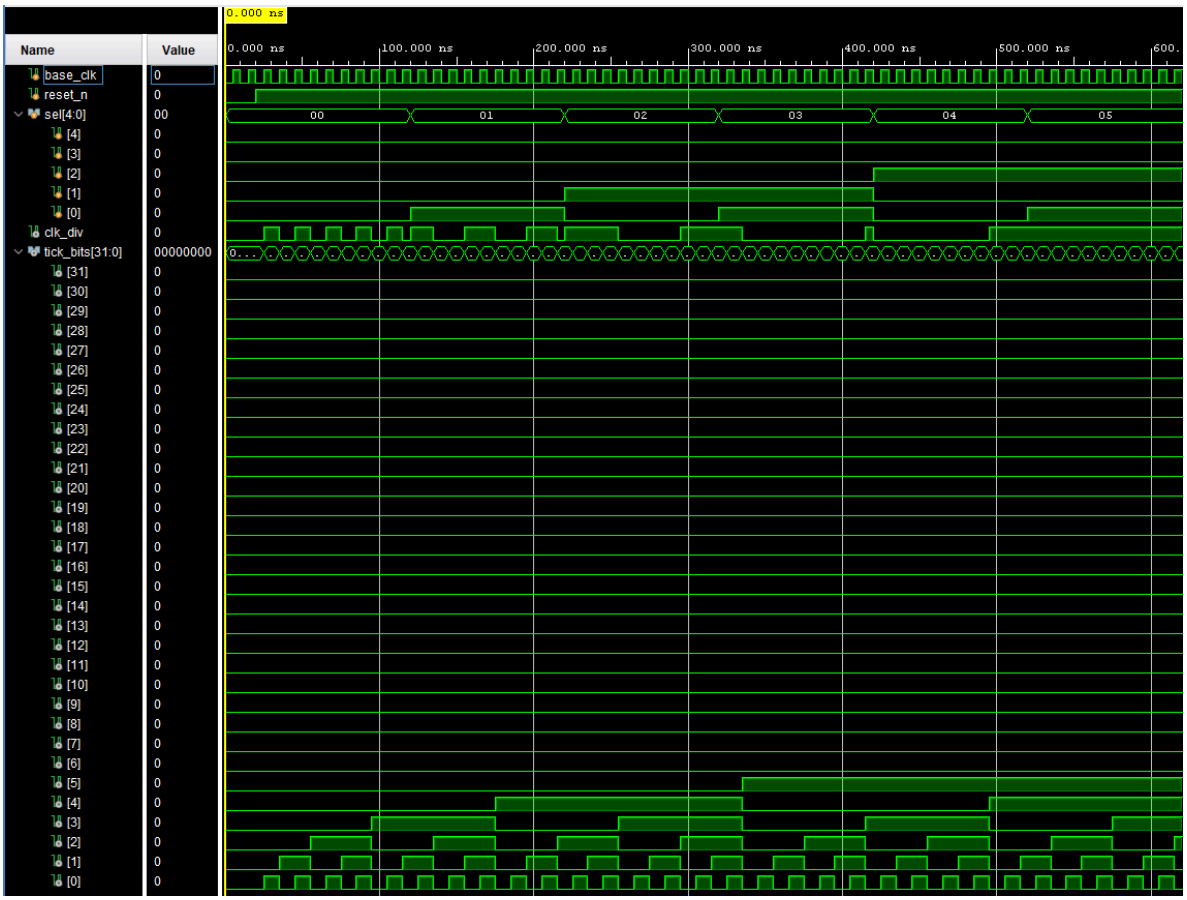
ALU



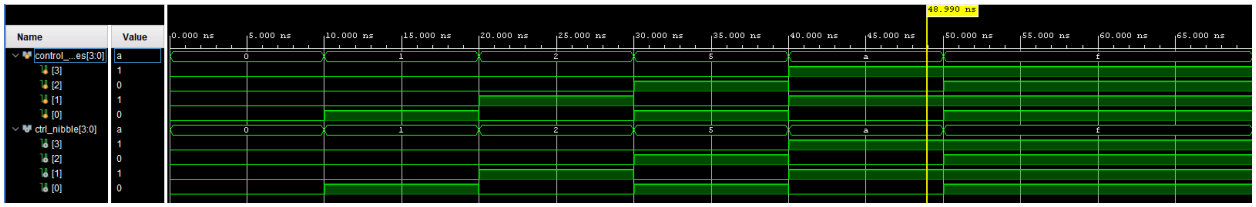
BCD Counter



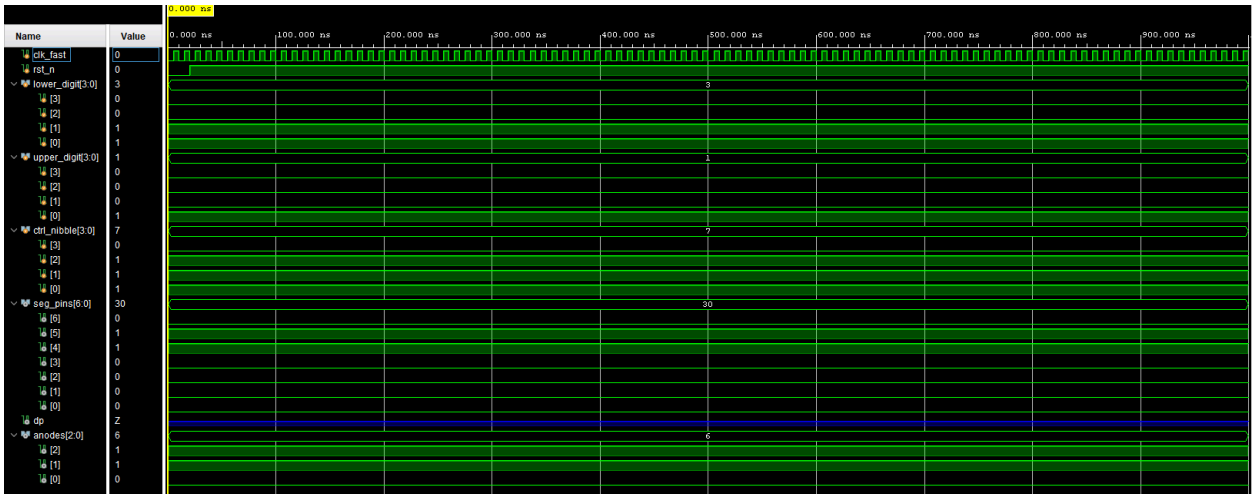
Clock Divider



Control Decoder



7 Segment



Partner Contributions:

Team Member	Contribution	% Effort
Jonathan Huynh	**Code:** [Clock Divider, BCD Up/Down Counter, ALU] **Testbench:** [Top Module, ALU, Control Decoder] **Additional:** Synthesis/Implementation and Demo	50%
Adam Godfrey	**Code:** [7 Segment, Top Module, Control Decoder] **Testbench:** [7 Segment, BCD Up/Down, Clock Divider] **Additional:** Simulation and Lab Report	50%