ECEN 340
Lab #6

Clock Generation and Module Instantiation

Purpose:

1. To learn how to design and simulate clock dividers
2. To learn how to port signals to connectors to view with an oscilloscope
3. To learn how to instantiate multiple modules into one project
4. To learn how to trust simulation for sub modules

Overview:

The objective of this lab is to drive all 4 of the seven segment LEDs on the Basys 3 board so that each of them displays a different number. All sixteen DIP switches will be used to generate four hex numbers.

This lab will require enabling each of the 4 seven-segment LED columns one at a time at a very high rate so that it appears as though they are all on at once. If the update rate is faster than 30Hz, the flicker is not detectable to the human eye (Figures 1a and 1b).

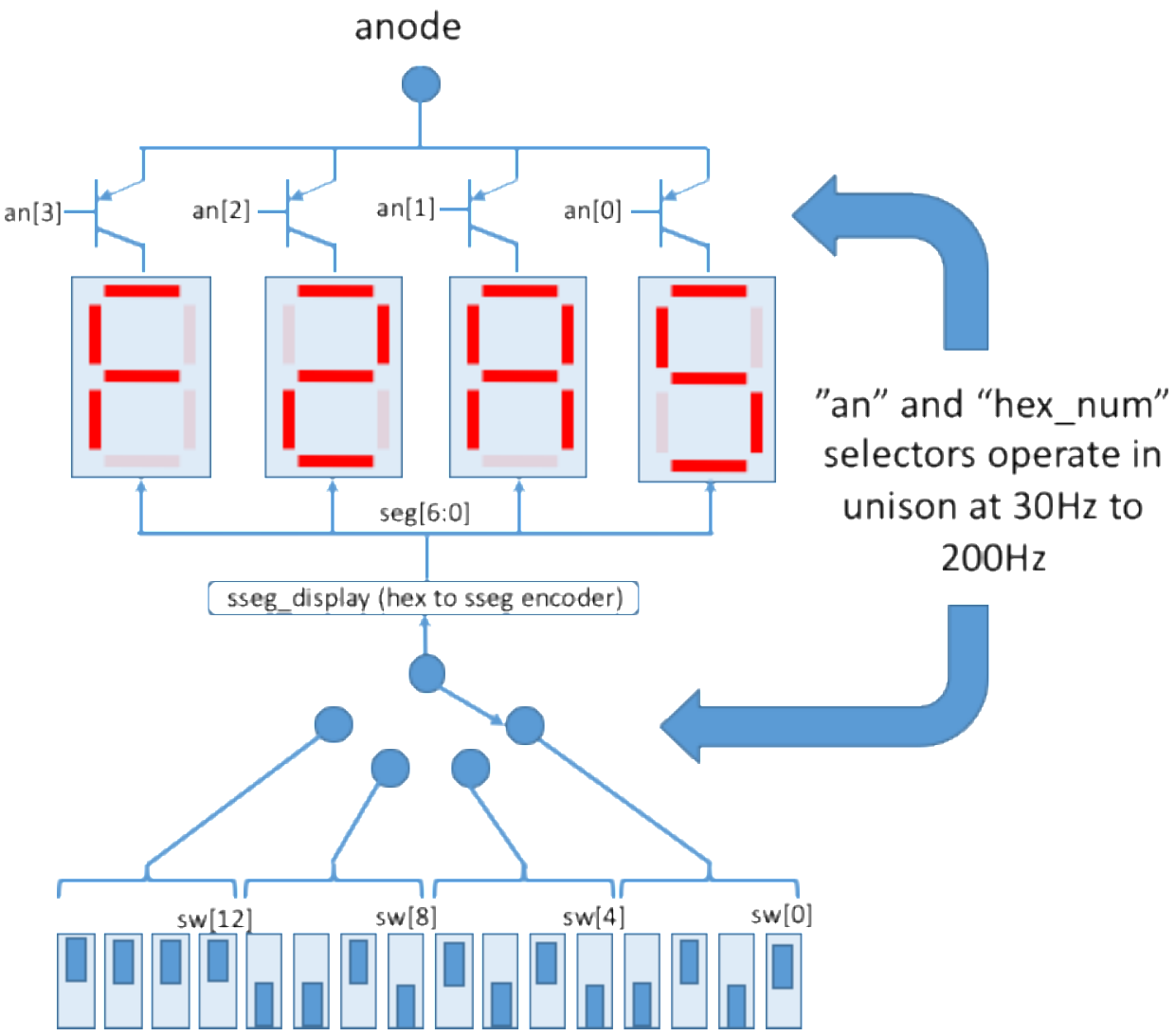

Figure 1a – Seven Segment Functional Overview

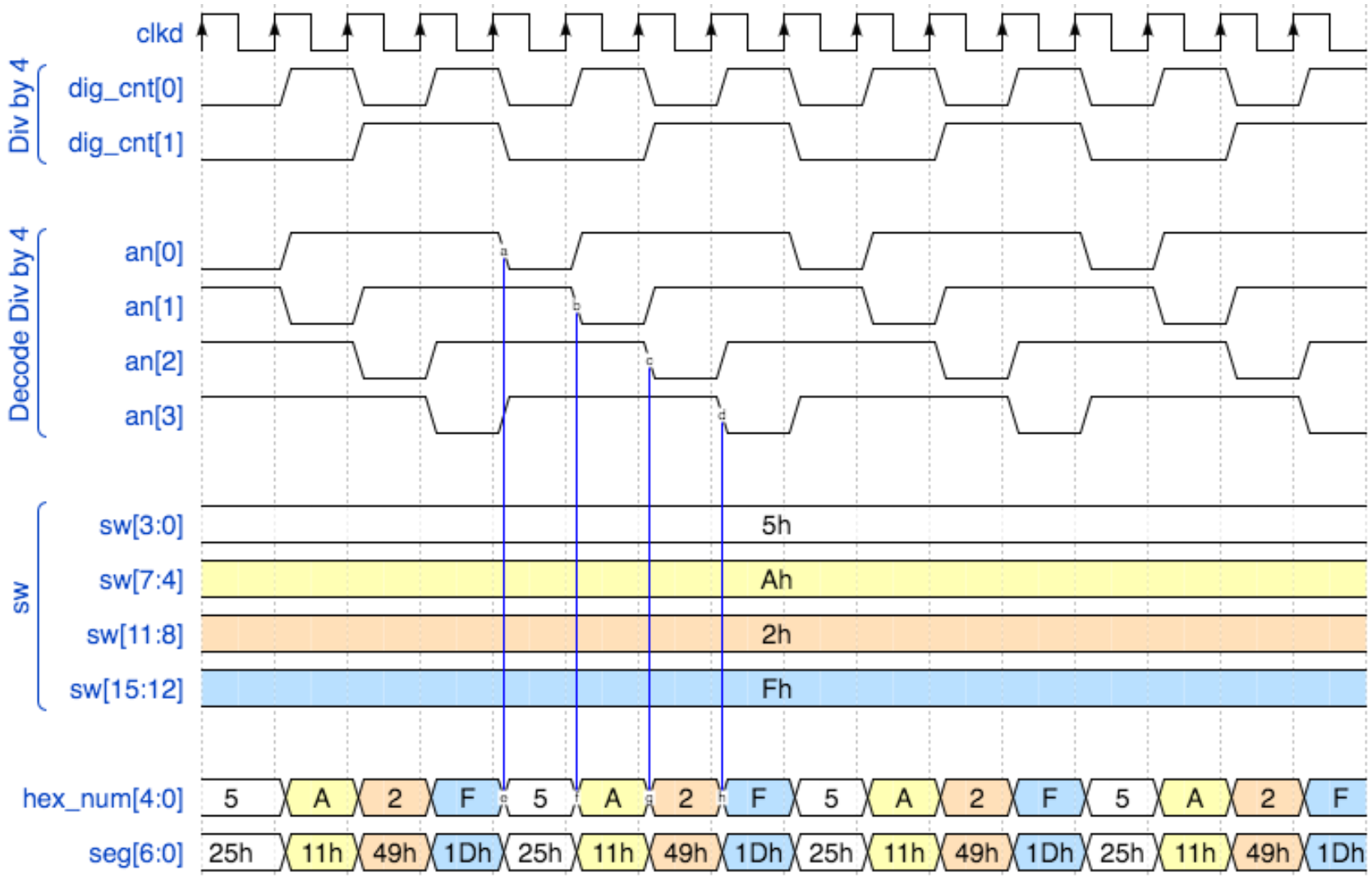

Figure 1b – Seven Segment Timing Overview

Procedure:

Part 1a. Create a new project called “sseg_x4 _top”. The ports will be the 16 dip switches (sw), the center pushbutton as a divider reset (btnC), the 7-seg LEDs (seg), all 4 column selectors (an), the decimal point (dp), five of the PMOD JA pins (JA) and the clock input (clk). The flip-flops in the project will all be reset by the middle push-button on the Basys board (btnC). This reset signal will need to be passed to all modules that use flip-flops (Figure 2).

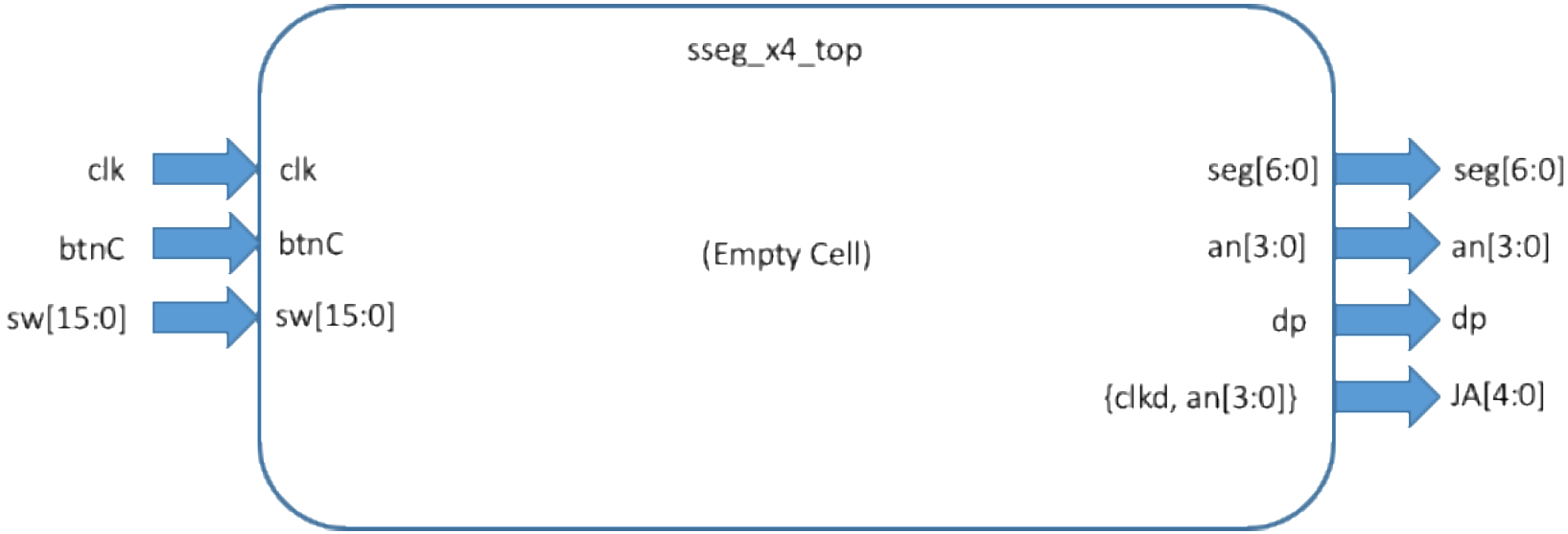

Figure 2 – Block Diagram of Top Level Module creation

Part 1b. As a separate file, write a Verilog module called “clk_gen”. This clock generator will be implemented as a counter. It will count from 0 to (2^26)-1 (how many bits will this take?). There will be a flip-flop for each bit. Each bit of this counter may be used as a divided clock of a different frequency.

You will use the 100MHz clock as an input to this module, so you will need to pass this clock to this module from your top-level module.

If the input clock is 100MHz, what is the frequency of the counter's lsb? msb?

If you want to update the displays faster than 30Hz but slower than 200Hz which outputs could be used to generate a proper clock frequency? Pick an appropriate output and designate it as your clock output port. It will be used to drive other modules (remember that the frequency per display should be between 30Hz - 200Hz, but there are 4 displays, so multiply the desired frequency by 4).

Instantiate this module in your top-level module. At this point, you will only have this one module instantiated in your top-level design (Figure 3). Use the "named port" instantiation method described in Figure 3.33 of the textbook (see "addern" instantiation), and in appendix, A.12.

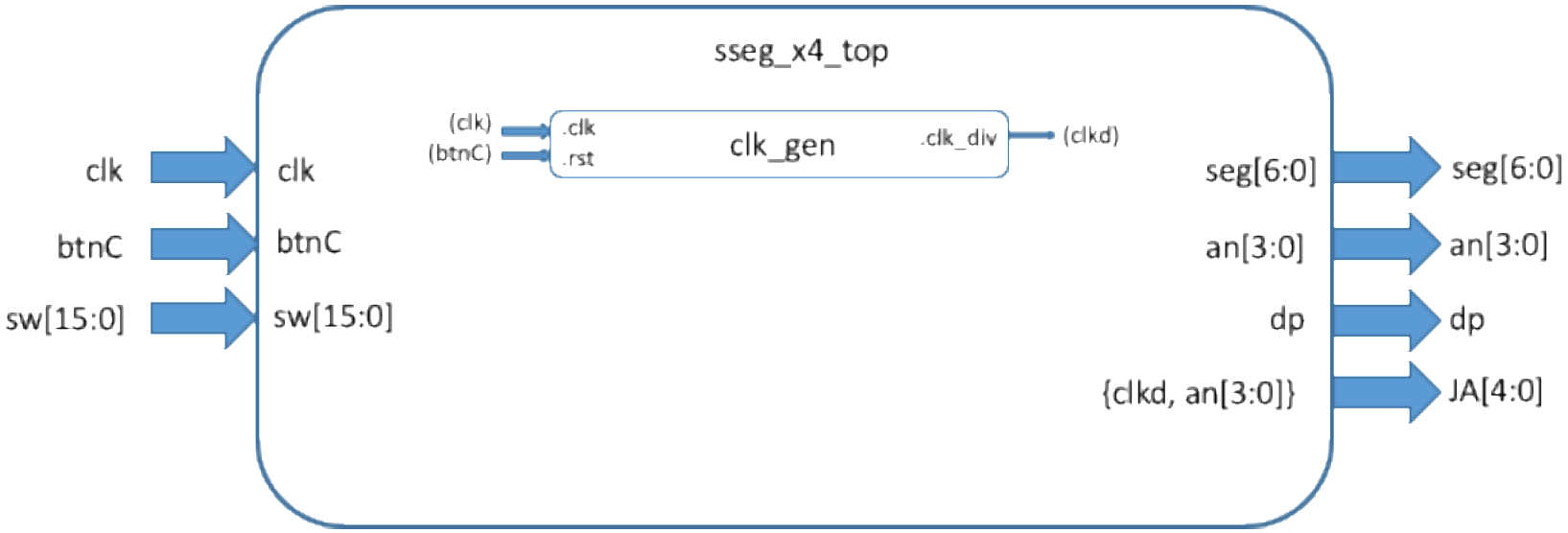

Figure 3 – Block Diagram of Instantiation of "clk_gen" module

For example, to instantiate "clk_gen" in "sseg_24_top", use the following port format:

clk_gen U1 (.clk(clk), .rst(btnC), .clk_div(clkd));

Part 1c. Create a testbench to simulate this module and verify its functionality. At a minimum, the testbench should include signals to drive “clk”, “btnC”, and "sw[15:0]” (Figure 4). Include a screenshot of the simulation results in the report.

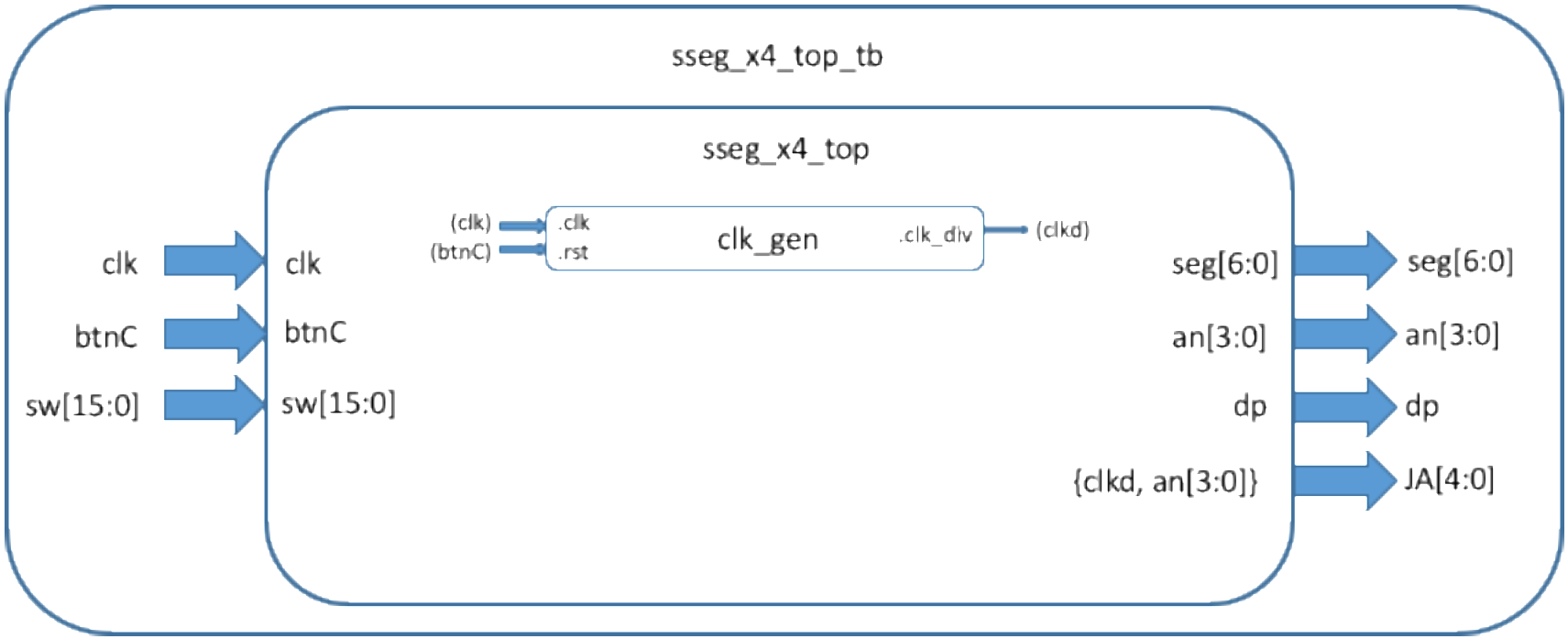

Figure 4 – Block Diagram of Test Bench With Top-Level Module Instantiated

Part 1d. As a separate file, write a module called “digit_selector.” The digit selector will receive the clock from the “clk_gen” cell, and will output a four-bit signal called “digit_sel”. This digit selector will decode the output of a 2-bit counter so that only one of four outputs is enabled at a time.

Instantiate the digit_selector module in your top-level module, and connect it up to your clock_gen module with port connections (Figure 5).

Simulate with the same test bench to verify that digit selector is functioning correctly.

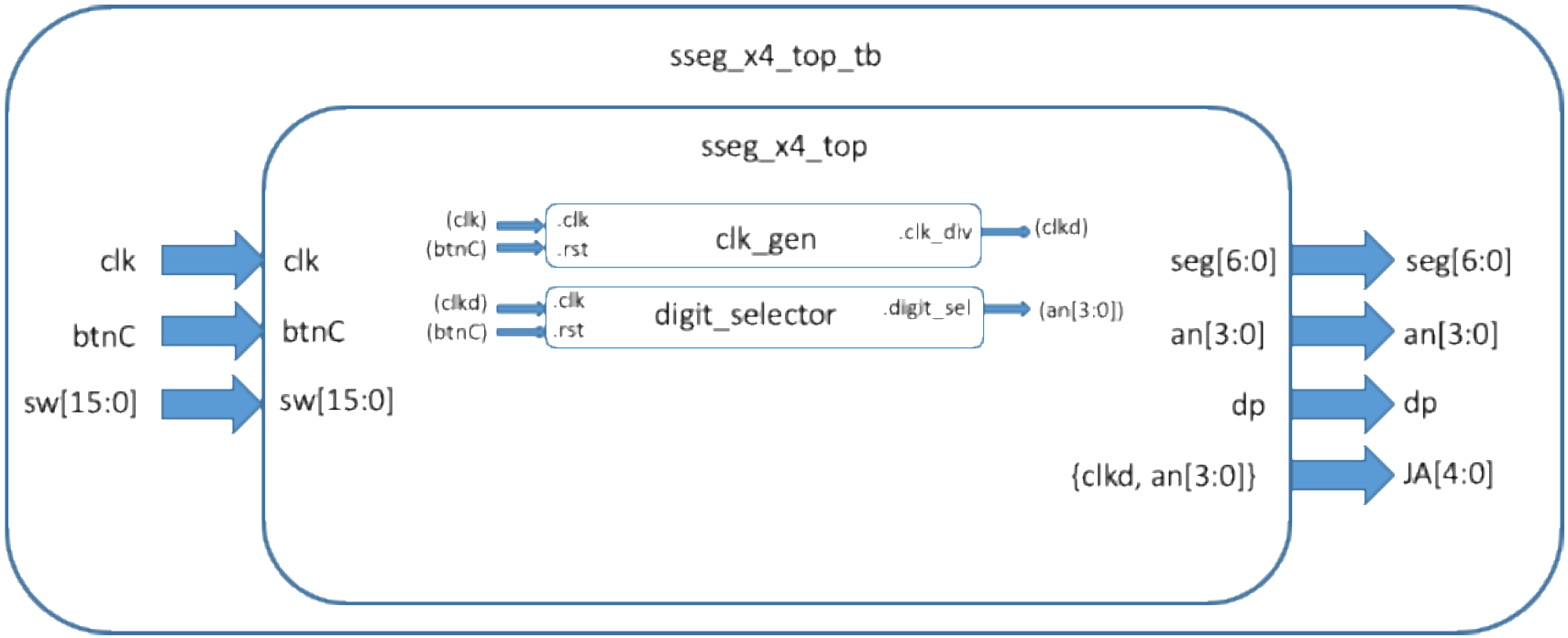

Figure 5 – Block Diagram of “digit_selector” instantiation

Part 2a. In your Project Manager, add the Verilog source file you're your previous seven-segment lab and instantiate your seven-segment module into sseg_x4_top (Figure 6). YOU DON'T NEED TO MODIFY THIS! If the name of your seven-segment lab were "sseg", it would look something like this:

```
wire [3:0] not_used;
reg   [3:0] hex_num;
                        // seg and dp are already declared as ports

sseg inst1 (
        .seg    (seg),          // drives the seven LEDs for the display
        .an     (not_used),  //  assigned to 4'b1110, so I am not using it
        .dp     (dp),           //  decimal place assigned permanently to 1'b1
        .sw     (hex_num)   //  can't be called sw any more so I'll call it hex_num
        );
```

Notes:

1. This instantiation uses an improved naming convention for the ports. The port names are no longer positional. Instead, we reference the port with the original name preceded with a ".", and we map the original name to a new name found in the (). The names in the () must be declared if they are not already declared as a port.

2. The LED column selector "an" is a four-bit word. Each bit enables one column. We were only using one column in the last lab, so we permanently tied this to 4'b1110, where the "0" enables the left column. We now want to drive the "an" port with "digit_sel" from part 1c of this lab.

3. The decimal place "dp" was not used in the last lab so it was set to 1'b1. We are still not using it in this lab, so we can keep the same name.

4. The variable “sw” was only 4 bits wide in the last lab, and was used to generate a hex number with four switches. We are now going to generate the four-bit hex number by rotating through all 16 switches, four at a time. To avoid confusion, we must change the port name to something other than “sw”, like “hex_num”. Remember that sw will be used on the top level as a 16-bit input.

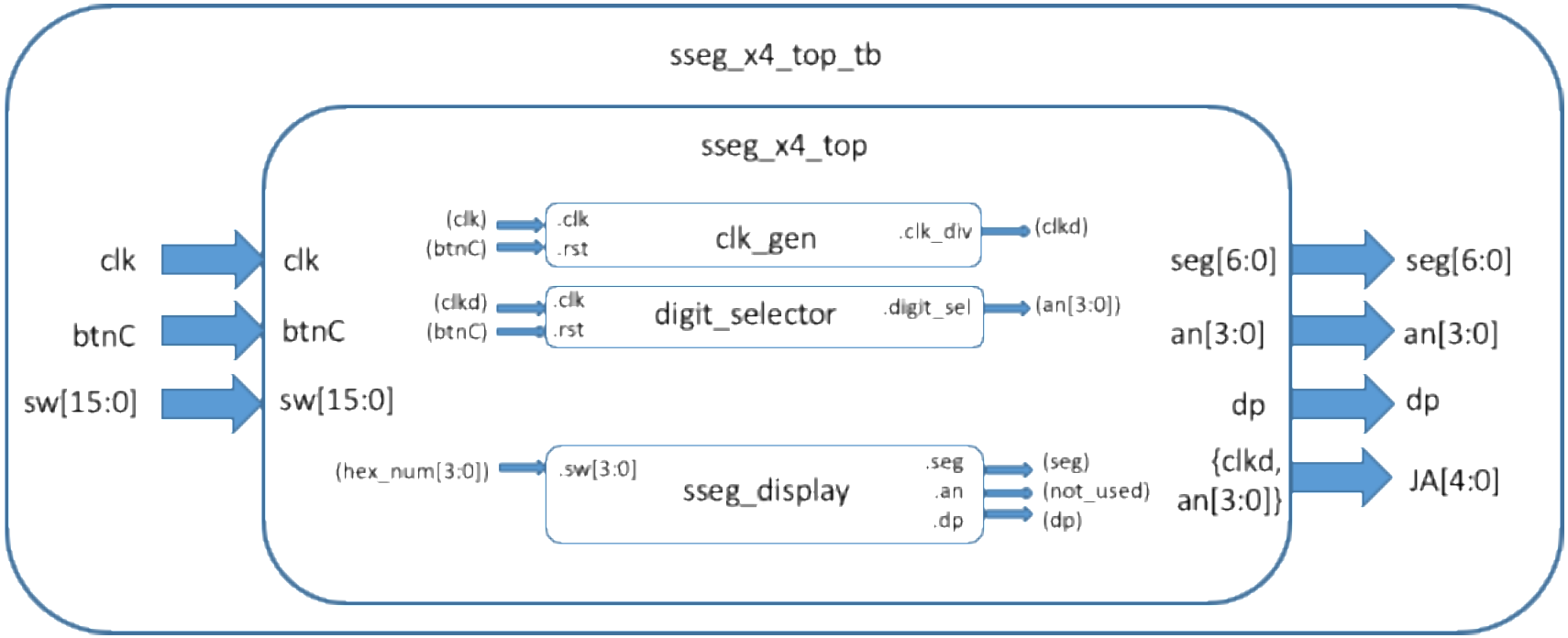

Figure 6 – Block Diagram of “sseg_display” Instantiation

Part 2b. Now you are only missing one element of this design. You have a digit selector, but you will need a way to select the corresponding group of four bits from the 16-switch input and write that hex number to the “hex_num” nibble. It must be in sync with the digit selector. How are you going to do this? Are you going to

write a new module or just place it in your digit selector? This will be your call. Figure 7 shows the creation of a new module called “hex_num_gen” to accomplish this. “hex_num_gen” takes the input from the 16 switches and creates the input (hex_num) to the “sseg_display” module. “hex_num” is four bits wide and it cycles through all 16 switch values four bits at a time.

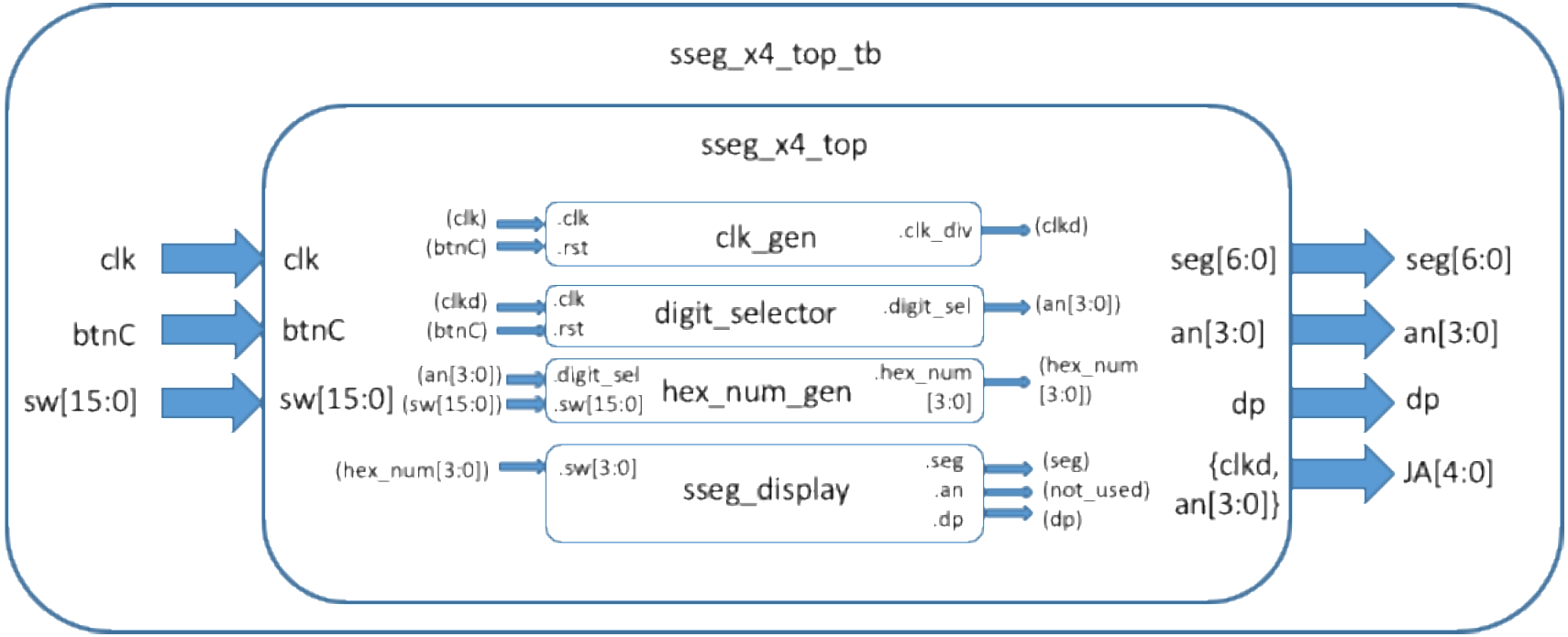

Figure 7 – Block Diagram of the Complete Hierarchy of the Seven Segment Display Driver

You can use the same test bench from part 1, and modify it as needed to be sure everything is working.

Part 2c. Generate the bitstream and test your design! Use the JA[4:0] pins to view critical waveforms on an oscilloscope.

```
// main module
`timescale 1ns / 1ps
module Lab_06(
   input [15:0] sw,
   input btnC,
   input clk,
   output [6:0] seg,
   output [3:0] an,
   output dp,
   output reg [4:0] JA
   );

   wire clkd;
   wire [3:0] hex_number;

   Clk_Gen U1(.clk(clk), .rst(btnC), .clk_div(clkd));

   digit_selector R2(.clkd(clkd), .btnC(btnC), .digit_sel(an), .sw(sw), .hex_number(hex_number));

   decoder W4(.sw4(hex_number), .seg(seg), .dp(dp));

   always @(posedge clk)
      begin

   JA[4] = clkd;
   JA[3:0] = an;
```

```
  end
endmodule
```

// module to divide the clock

```
`timescale 1ns / 1ps
module Clk_Gen(
   input clk,
   input rst,
   output reg clk_div
   );


   // 26-bit counter
   reg [25:0] counter;

   always @(posedge clk or posedge rst) begin
      if (rst)
         counter <= 26'b0; // Reset the counter
      else
         counter <= counter + 1'b1; // Increment the counter
   end

   // Output the counter value
   always  @ (posedge clk or posedge rst)
      begin
         if(rst)
```

```
            clk_div <= 1'b0;
        else
            clk_div <= counter [17];
    end
endmodule
```

// module to display digits on the 7 segment

```
`timescale 1ns / 1ps
module digit_selector(
  input clkd,
  input btnC,
  input [15:0] sw,
  output  reg [3:0] hex_number,  // Changed to reg to use inside always block
  output reg [3:0] digit_sel
);
  reg [3:0] counter = 4'b0001;

  always @(posedge btnC, posedge clkd) begin
    if (btnC)
      counter <= 4'b0001; // Reset the counter
    else
    begin
      counter <= (counter << 1); // Shift left

      if (counter == 4'b1000)
      counter <= 4'b0001;
```

```
      end
      // If counter exceeds 4'b0111, wrap back to 4'b0001

   end

   // Always block to assign digit_sel based on counter value


   // Always block to assign hex_number based on counter value
   always @(*) begin
      case (counter)
         4'b0001: begin  hex_number = sw[3:0]; digit_sel = 4'b1110;end
         4'b0010: begin  hex_number = sw[7:4]; digit_sel = 4'b1101;end
         4'b0100: begin  hex_number = sw[11:8]; digit_sel = 4'b1011;end
         4'b1000: begin  hex_number = sw[15:12]; digit_sel = 4'b0111;end

      endcase
   end
endmodule
```

// 7 segment decoder module

```
`timescale 1ns / 1ps
module decoder(
   input [3:0] sw4,         // 4-bit input for digit
   output reg [6:0] seg,    // 7-segment display output
```

```
  output dp             // Decimal point (set to always off) and Anode control
  );

  assign dp = 1'b1;        // Decimal point off


  always @ (sw4)
    case (sw4)
      4'b0000: seg = 7'b1000000;  // 0
      4'b0001: seg = 7'b1111001;  // 1
      4'b0010: seg = 7'b0100100;  // 2
      4'b0011: seg = 7'b0110000;  // 3
      4'b0100: seg = 7'b0011001;  // 4
      4'b0101: seg = 7'b0010010;  // 5
      4'b0110: seg = 7'b0000010;  // 6
      4'b0111: seg = 7'b1111000;  // 7
      4'b1000: seg = 7'b0000000;  // 8
      4'b1001: seg = 7'b0011000;  // 9
      4'b1010: seg = 7'b0001000;  // A
      4'b1011: seg = 7'b0000011;  // b
      4'b1100: seg = 7'b1000110;  // C
      4'b1101: seg = 7'b0100001;  // d
      4'b1110: seg = 7'b0000110;  // E
      4'b1111: seg = 7'b0001110;  // F
      default: seg = 7'b1111110;  // lol
    endcase
endmodule
```

// Test Bench

```
`timescale 1ns / 1ps
module Lab6_tb(
    );

    reg [15:0] sw;
    reg btnC;
    wire [6:0] seg;
    wire [3:0] an;
    wire dp;
    reg [4:0] JA;
    reg clk;

    wire [3:0] hex_number;
    wire clkd;

   Clk_Gen U1(.clk(clk), .rst(btnC), .clk_div(clkd));

   digit_selector U2(.clkd(clkd), .btnC(btnC), .digit_sel(an), .sw(sw), .hex_number(hex_number));

   decoder W4(.sw4(hex_num), .seg(seg), .dp(dp));


    always begin
       #5 clk = ~clk;  // Generate clock with period of 10 time units
    end
```

```
// Initial block for test stimuli and monitoring
initial begin
    // Initialize inputs
    clk = 0;
    btnC = 0;
    sw = 16'b0001_0001_0001_0001;  // Example switch pattern


    // Add delay for initial state
    #10;

    // Monitor the values
    $monitor("Time = %0t, btnC = %b, sw = %b, clkd = %b, hex_number = %h, an = %b, seg = %b",
            $time, btnC, sw, clkd, hex_number, an, seg);

    // Stimulus
    // Simulate button press reset
    btnC = 1;  // Simulate pressing button
    #10;
    btnC = 0;  // Button released

    // Cycle through different switch patterns
    sw = 16'b1111_0000_1111_0000;
    #20;

    sw = 16'b0000_1111_0000_1111;
    #20;
```

```
        sw = 16'b0101_0101_0101_0101;
        #20;

        // Final states (adding delays to simulate clock cycles)
        #50;

        $finish;  // End the simulation
    end
endmodule
```

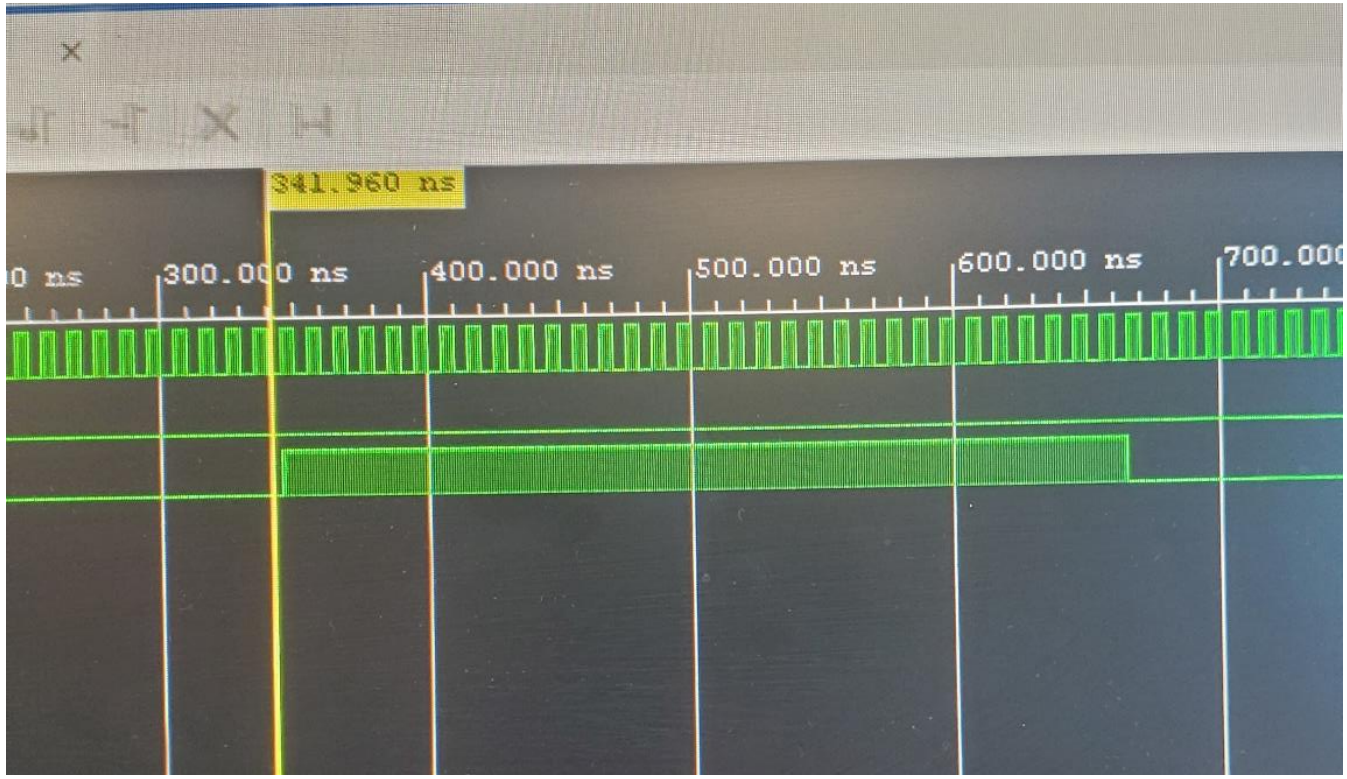

Figure A - Clk_Gen test simulation

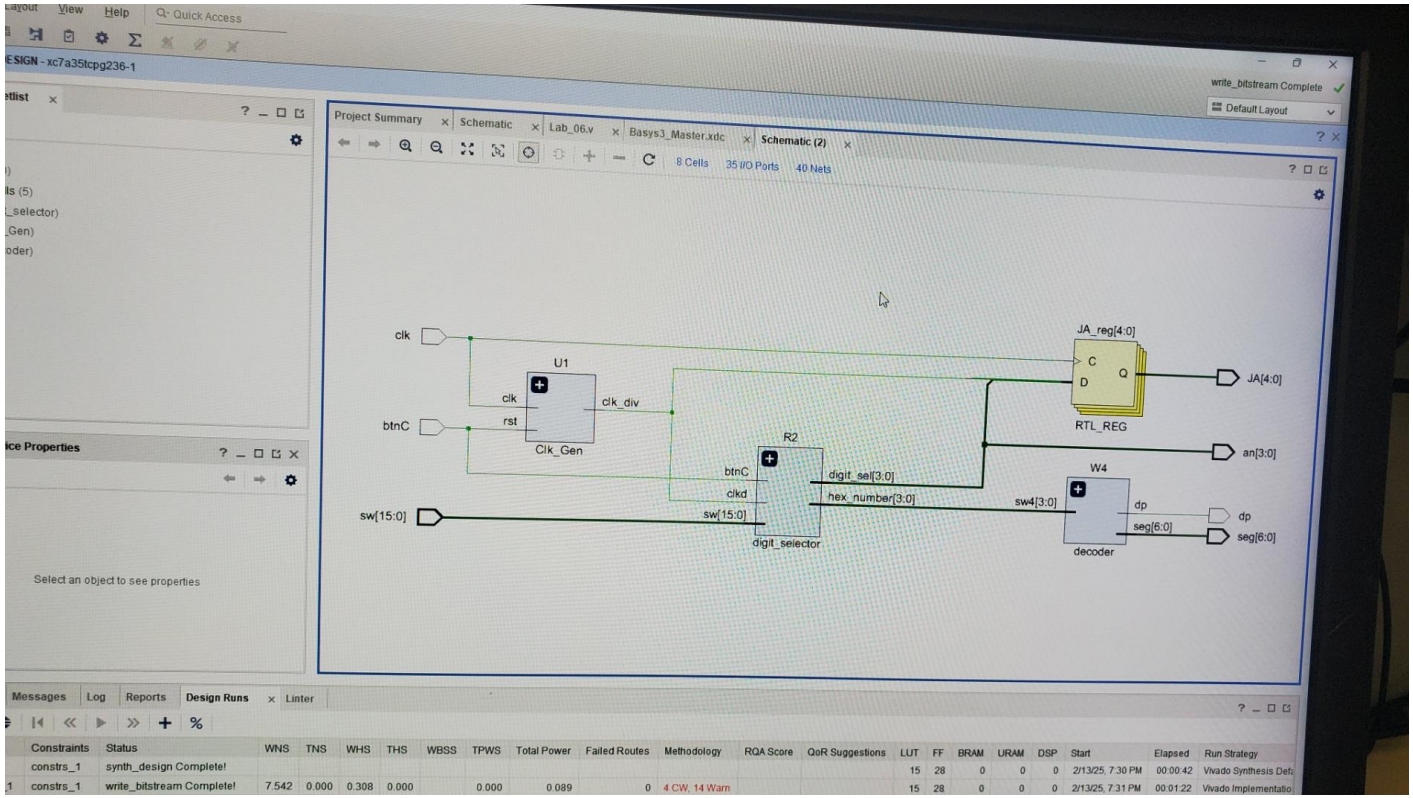

Figure B - Final Schematic

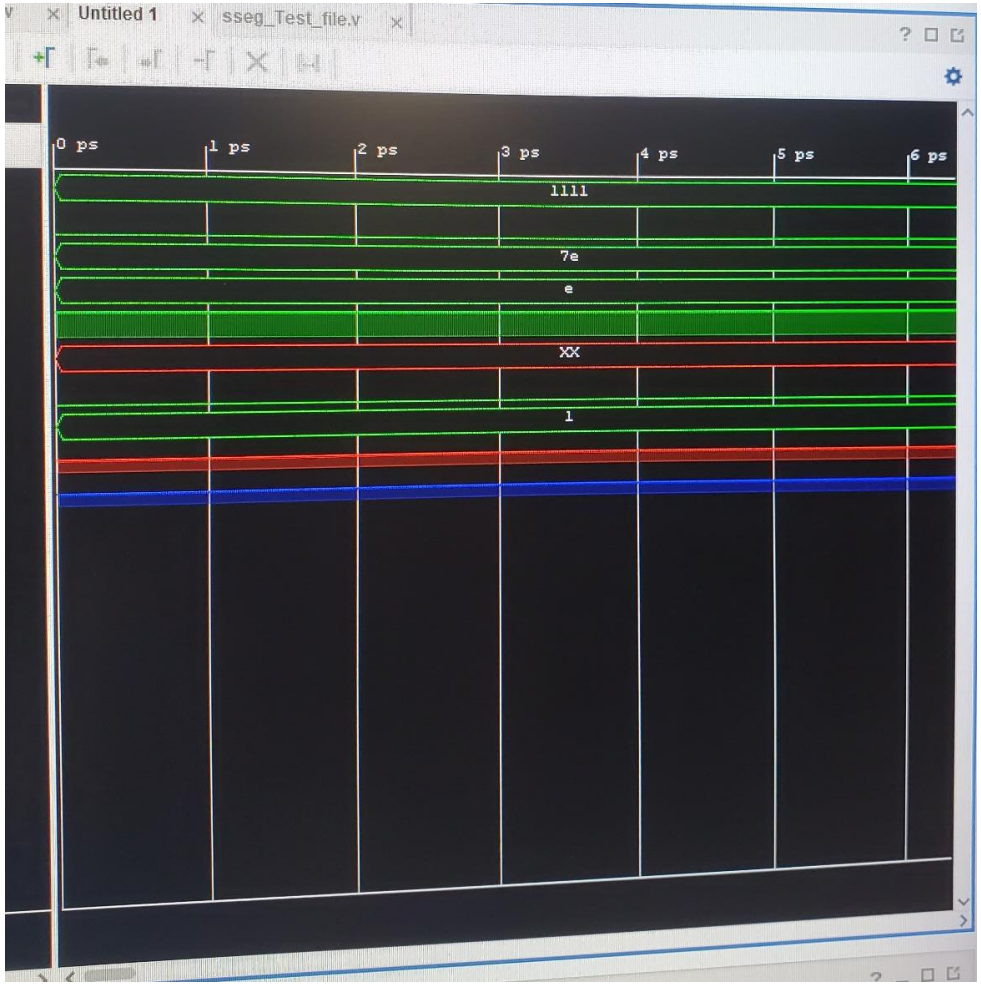

Figure C - Final Simulation

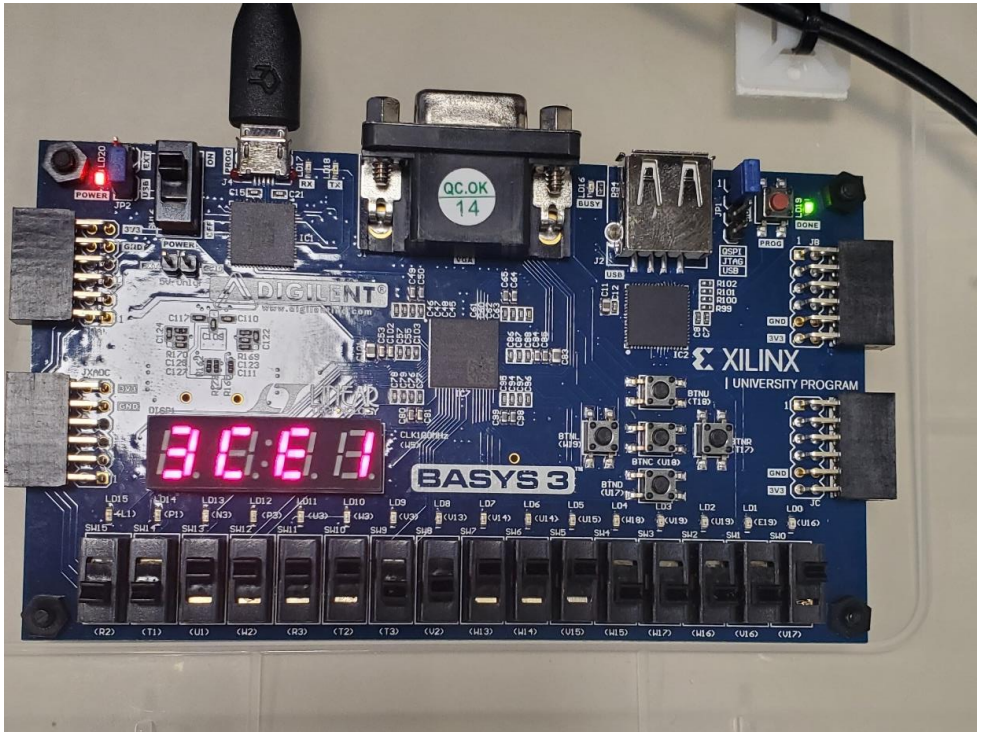

Figure D - Final Implementation

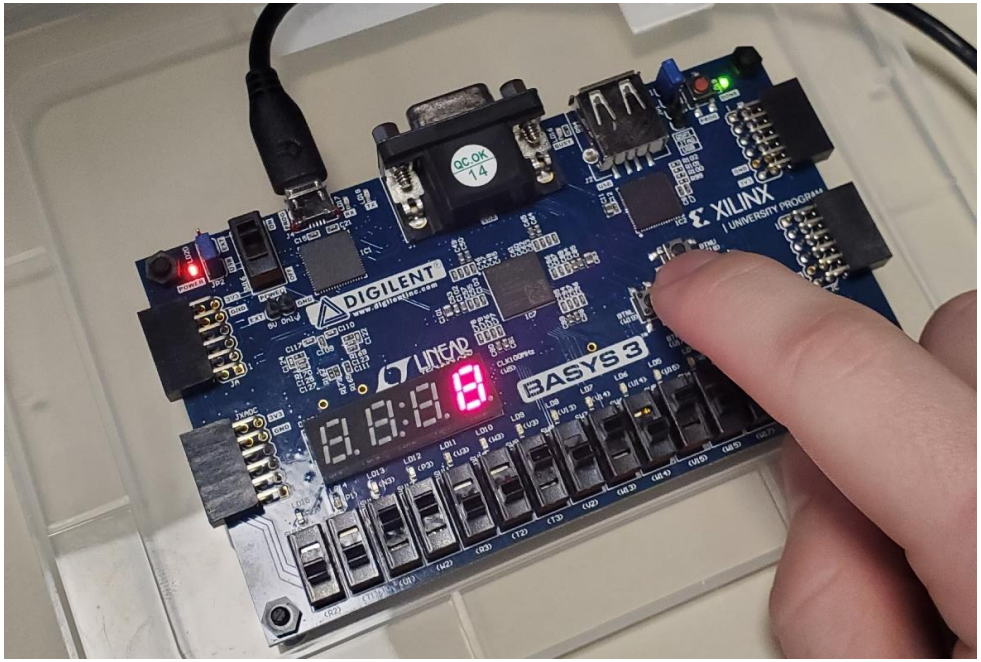

Figure E - Final Implementation (reset)

Conclusion:

In this lab, we learned how to design and simulate clock dividers to get the frequency we want and we used this divided clock to implement 4 digits on the 7 segment display. We used several different modules to implement this design so we had to learn how to get the modules to work together even if they're in separate files. And then we learned a lot about writing test benches to simulate our modules. So our new tools and methods we used in this lab were creating our own clock divider, using several different modules, and making test benches for simulation. Our results of this were eventually we got our intended design to work how we wanted. We tested our modules and final designs in simulation and we also programmed the board and physically tested different inputs and verified correct outputs. We struggled for a while with different errors and in creating portions of the code but in the end our design was 100% functional.