LAB 6

Group C

Edward Enriquez, eenriquez@cpp.edu, ID 013242619
Lorenzo Antonio Fabian Lopez, lfabianlopez@cpp.edu, ID 014903070
Rafael Gaeta, rgaeta@cpp.edu, ID 014522794

Contribution:
Edward Enriquez
50% Code creation and analysis

Lorenzo Antonio Fabian Lopez
25% Analysis of results and conclusion

Rafael Gaeta
25% Report and graphics

Abstract

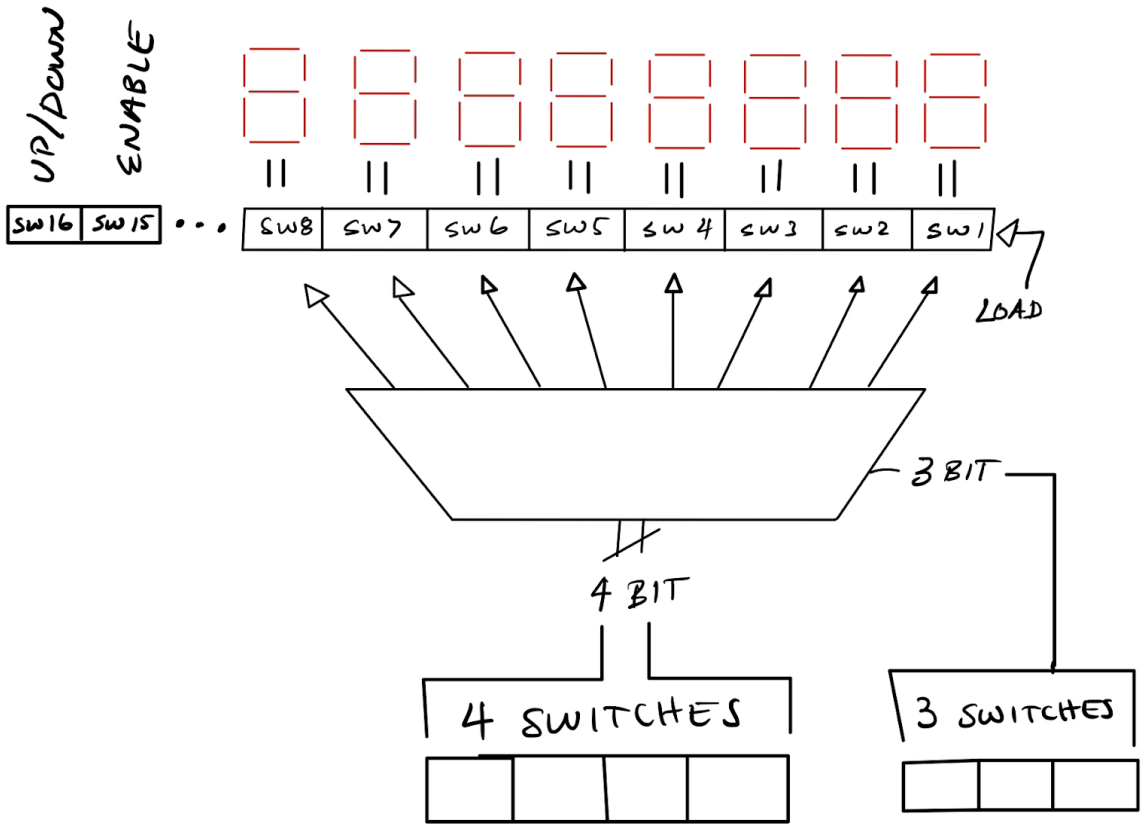
The objective of this lab is to construct a BCD up-counter & down-counter that counts from 0-99999999 & 99999999-0 respectively. The velocity of the counter will depend on the clock divider and will be controlled by the first five switches on the board. The numbers will be displayed on the seven segment display. Additionally a load button will be implemented to inject a number into any of the seven segment displays. When the load button is active, the counter will stop. Once the load button is disabled, the counter will resume counting.

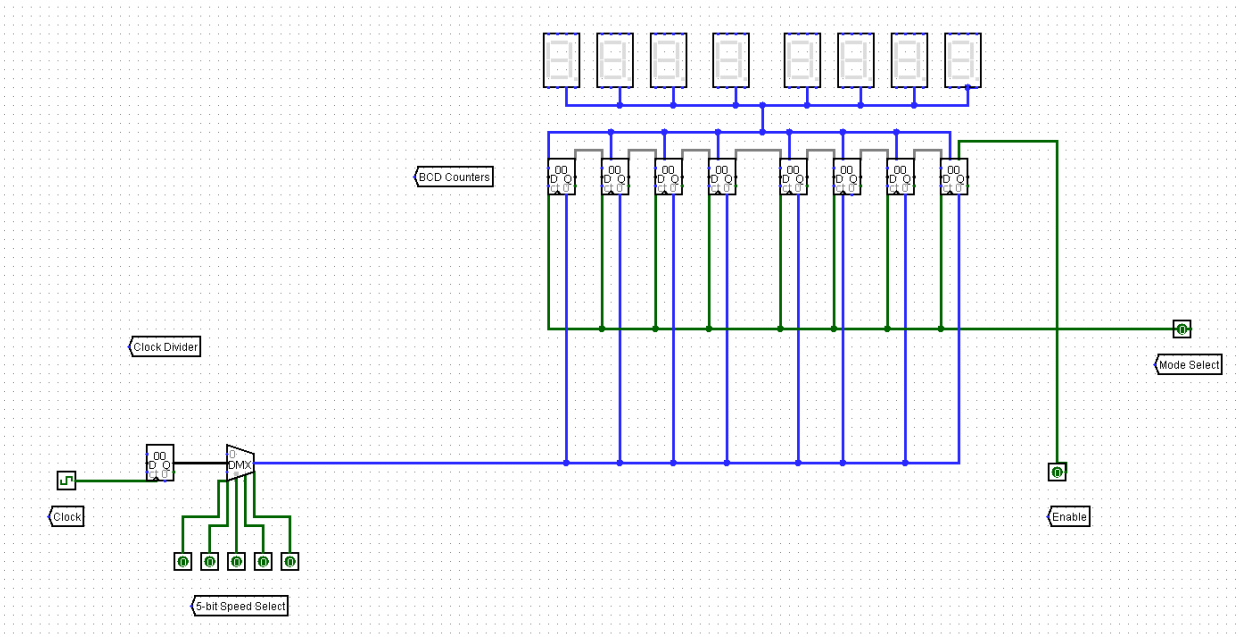
Specifications,

- Define Circuit Schematic/Design
- Total power ≤ 0.114 W
- Number of LUT's ≤ 84
- Number of FF ≤ 101

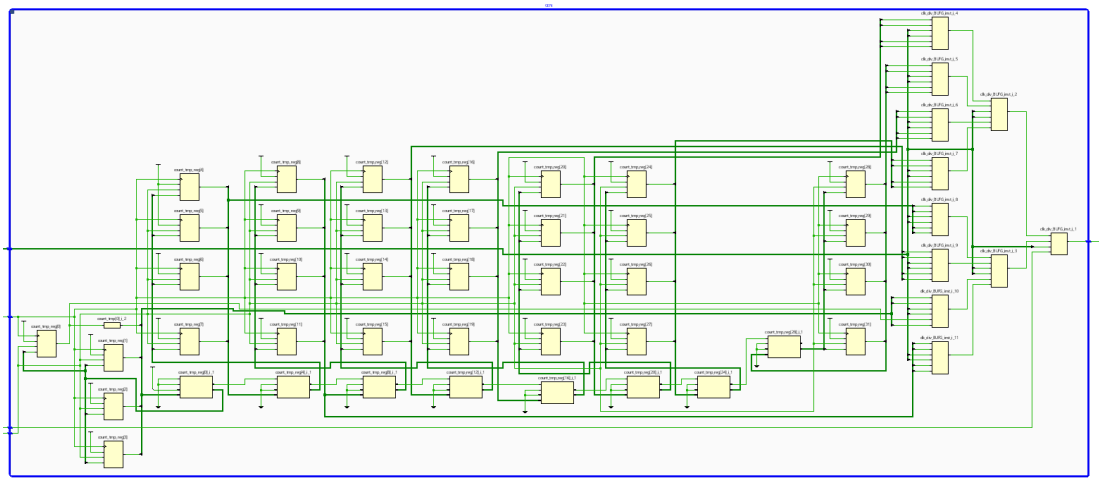
Circuit Description

Diagram

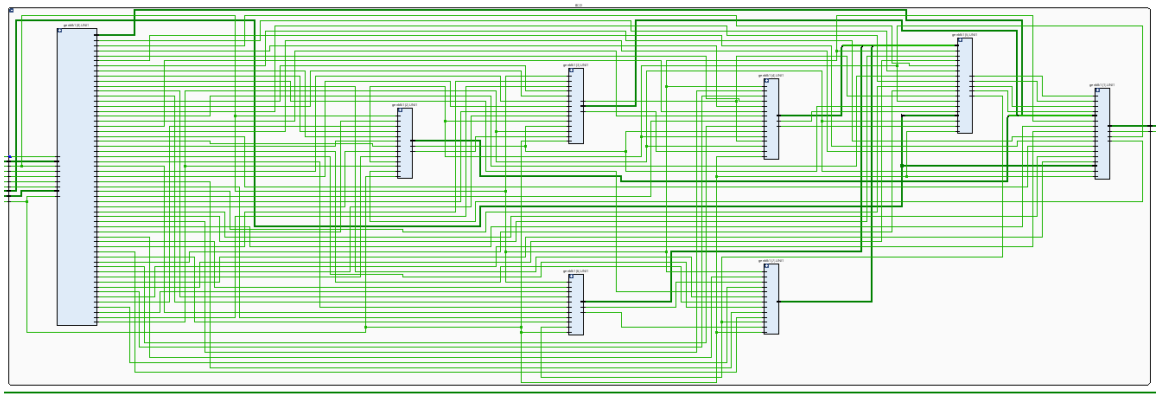




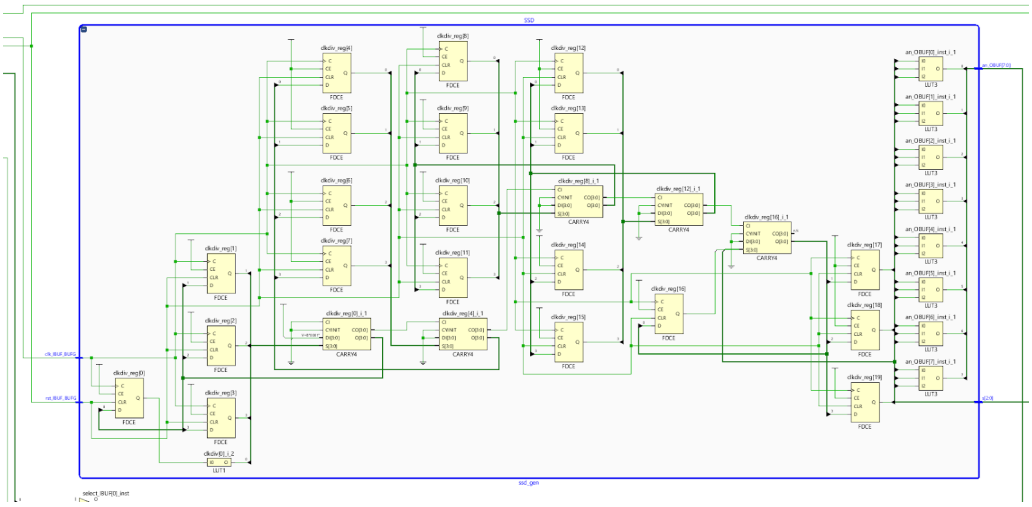
Verilog Generated Schematics with Instantiation



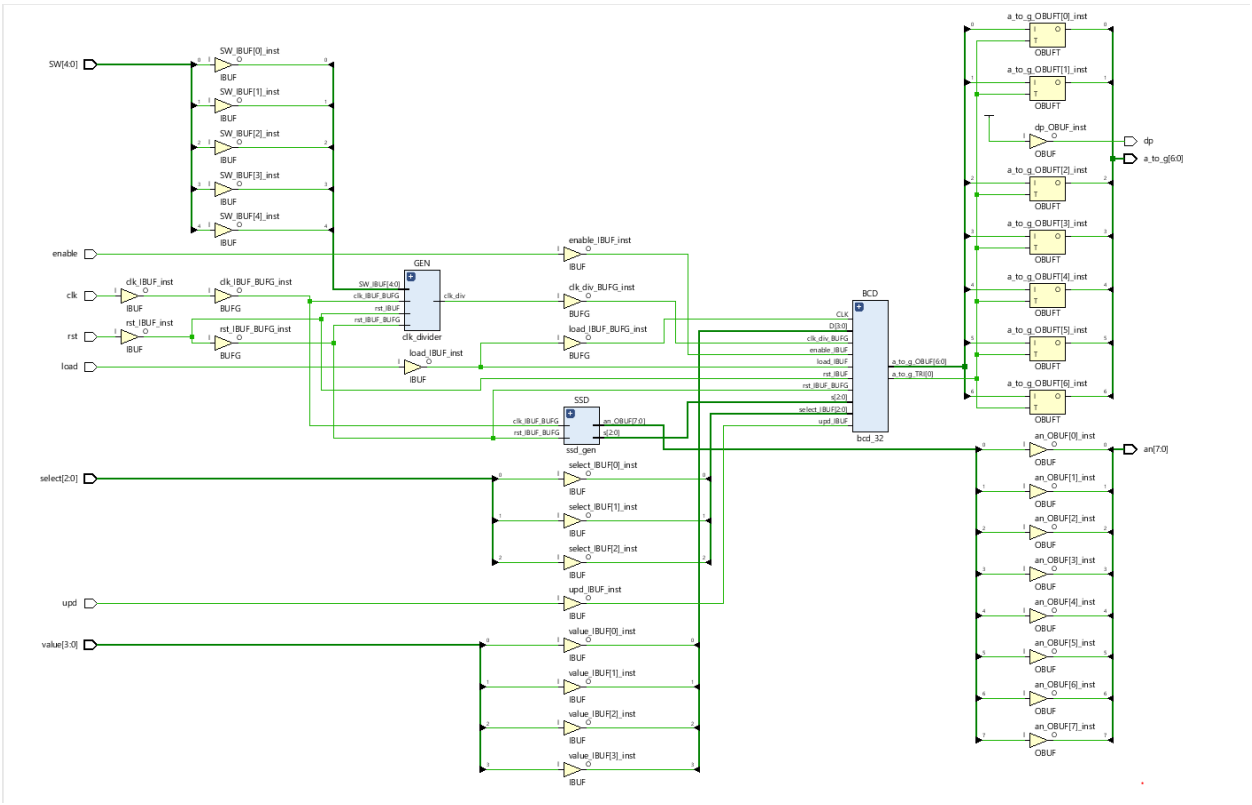
Clock Divider Generated Schematic



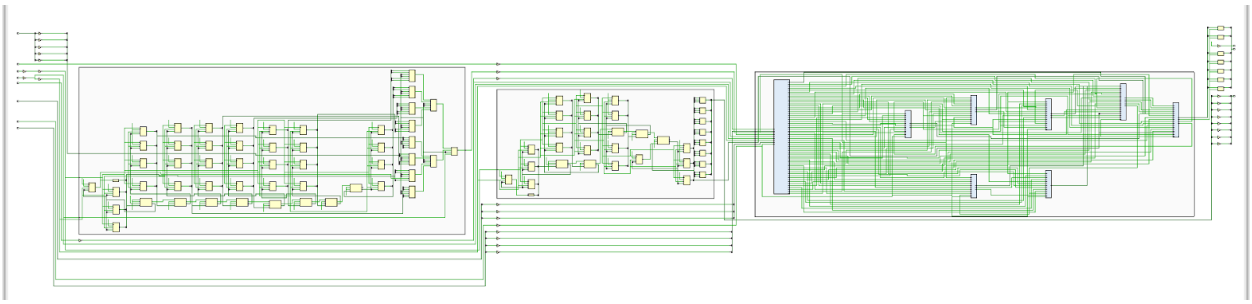
Counter Generated Schematic



Seven Segment Display Generated Schematic



Full Circuit Schematic



Full Circuit Schematic Expanded

Code Detail

The module which contained the major change in the code from last week's lab is the bcd_counter module. Four additional inputs were added to the module which were, load, value, key, and select. The *key* input is a unique number which is assigned to each seven segment display (SSD). The *select* input is a 3 switch input which chooses the SSD which we are loading the value into. The *value* input is a 4 switch input and dictates which number is to be loaded to the chosen SSD.

The first function of the *load* input is to assign the wire named *opwire* the contents of the *value* input. The second function of the *load* input is to assign *opwire* to the output of the module. However, this second function only happens when the *select* input matches the key of the specific module instantiation. All together the *load* button will output the given *value* input only if the *select* matches the given SSD *key.* Otherwise the counter will work as it did in the previous lab.

```
`timescale 1ns / 1ps
module bcd_counter(
                    input clk,
                    input rst,
                    input en_in,
                    input upd,
                    input load,
                    input [3:0] value,
                    input [2:0] key,
                    input [2:0] select,
                    output reg [3:0] op,
                    output wire en_out
                   );

                   reg [3:0] opwire;
                   reg [3:0] reset;

                   always@(posedge load)
                   begin
                     opwire<=value;
                   end

                   always@(posedge rst)
                   begin
                      reset <= upd*(4'd9);
                   end

                   always@(posedge rst, posedge load, posedge clk)
                   begin
                     if(rst)
                     op<=reset;
                     else if(load && (key==select))
                     op<=opwire;
                     else
                     begin
                         if (en_in==1'b1)
                         begin
                             if (upd==1'b1)
                             begin
                                 if (op > 0)
                                 op <= op - 1;

                                  else
                                  op <= 4'd9;
                              end
                              else if (upd==1'b0)
                              begin
                                  if(op <9 )
                                  op <= op+ 1;
                                  else
                                  op <= 4'd0;
                              end
                          end
                     end
                   end


  assign en_out = (op[3] & op[0] & (~op[2]) & (~op[1]) & (~upd) & (en_in)) | ( (~op[3]) & (~op[0]) & (~op[2]) & (~op[1]) & (upd) & (en_in)) ;
endmodule

```

bcd_counter.v

The *bcd_32.v* module was modified to pass down the *load, value,* and *select* inputs from the *top* module to the 8 individual *bcd_counter* modules. It also assigned each instantiation of the *bcd_counter* module with a unique *key* value based on the variable *i.* This will allow the user to reference a specific SSD using the 3 switch *select* input. All other values of the module stayed the same.

```
1  `timescale 1ns / 1ps
2  module bcd_32(
3                  input clk,
4                  input rst,
5                  input en,
6                  input upd,
7                  input load,
8                  input [3:0] value,
9                  input [2:0] select,
10                 output wire [31:0] count
11                 );
12                wire [7:0] tmp;
13
14                assign tmp[0] = en;
15
16                genvar i;
17
18
19                generate
20
21                for (i = 0; i<8 ; i = i+1)
22                begin
23                bcd_counter  UNIT  (
24                                     .clk(clk),
25                                     .rst(rst),
26                                     .en_in(tmp[i]),
27                                     .upd(upd),
28                                     .load(load),
29                                     .value(value),
30                                     .select(select),
31                                     .key(i),
32                                     .op(count [4*(i+1)-1 : 4*i]),
33                                     .en_out(tmp[i+1])
34                                    );
35             end
36             endgenerate
37 endmodule
```

bcd_32.v

The clock divider and SSD remained unchanged from the previous lab. The load function focused on assigning new values to the SSDs and did not change how or at what rate they were to be displayed at.

```
`timescale 1ns / 1ps
module clk_divider
      (
         input clk,
         input rst,
         input [4:0] SW,
         output reg clk_div
     );

     reg [31:0] count_tmp;

     initial  #1 count_tmp<=0;

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

     always@(count_tmp[SW] or rst)
     begin
     if(rst)
     clk_div<=1'b0;
     else
     clk_div<=count_tmp[SW];
     end

endmodule
```

clk_divider.v

```
`timescale 1ns / 1ps
//`define an_control 8'b00000000
`define dp_off   1'b1
`define initial_digit 8'b11111111

module ssd_gen(
                input [31:0] SW,
                input clk,
                input rst,
                output reg [6:0] a_to_g,
                output reg [7:0] an,
                output wire dp
              );

assign dp  = `dp_off;

wire [2:0] s;
wire [7:0] aen;
reg [19:0] clkdiv;
reg [3:0] digit;
assign s = clkdiv[19:17];
assign aen = `initial_digit;



// clock divider
    always@(posedge clk or posedge rst)
        begin
            if (rst)
                clkdiv <= 0;
            else
                clkdiv <= clkdiv+1;
        end

// digit select :anode
            always@(aen, s)
            begin
                an = 8'b11111111;
                if(aen[s] == 1)
                    an[s] = 0;
            end
                an[s] = 0;
        end

        // 4-to-1 Mux: mux4x1
        always@(s, SW)
          begin
            case(s)
                0: digit = SW[3 : 0];
                1: digit = SW[7 : 4];
                2: digit = SW[11: 8];
                3: digit = SW[15:12];
                4: digit = SW[19:16];
                5: digit = SW[23:20];
                6: digit = SW[27:24];
                7: digit = SW[31:28];
          default: digit = 4'bZZZZ;
            endcase
          end

// 7-segement decoder : hex7seg

always@(digit)
    begin
        case(digit)
                0: a_to_g = 7'b0000001;
                1: a_to_g = 7'b1001111;
                2: a_to_g = 7'b0010010;
                3: a_to_g = 7'b0000110;
                4: a_to_g = 7'b1001100;
                5: a_to_g = 7'b0100100;
                6: a_to_g = 7'b0100000;
                7: a_to_g = 7'b0001111;
                8: a_to_g = 7'b0000000;
                9: a_to_g = 7'b0000100;
          default: a_to_g = 7'bZZZZZZZ;
        endcase
    end


endmodule
```

ssd_gen.v

The *top.v* module instantiated all modules as it did in the previous lab. The only differences were the 3 added inputs which were *load, select,* and *value.* The inputs were assigned in the *nexys4ddr* file and passed down to the *bcd_32.v* module.

```
1  `timescale 1ns / 1ps
2  module top(
3              input clk,
4              input rst,
5              input [4:0] SW,
6              input enable,
7              input upd,
8              input load,
9              input [3:0] value,
10             input [2:0] select,
11             output wire [6:0] a_to_g,
12             output wire [7:0] an,
13             output wire dp
14          );
15         wire clk_div;
16         clk_divider  GEN    (
17                              .clk(clk),
18                              .rst(rst),
19                              .SW(SW),
20                              .clk_div(clk_div)
21                              );
22         wire [31:0] TMP;
23          bcd_32  BCD    (
24                          .clk(clk_div),
25                          .rst(rst),
26                          .en(enable),
27                          .upd(upd),
28                          .count(TMP),
29                          .load(load),
30                          .value(value),
31                          .select(select)
32                          );
33         ssd_gen   SSD   (
34                           .SW(TMP),
35                           .clk(clk),
36                           .rst(rst),
37                           .a_to_g(a_to_g),
38                           .an(an),
39                           .dp(dp)
40                          );
41 endmodule
```

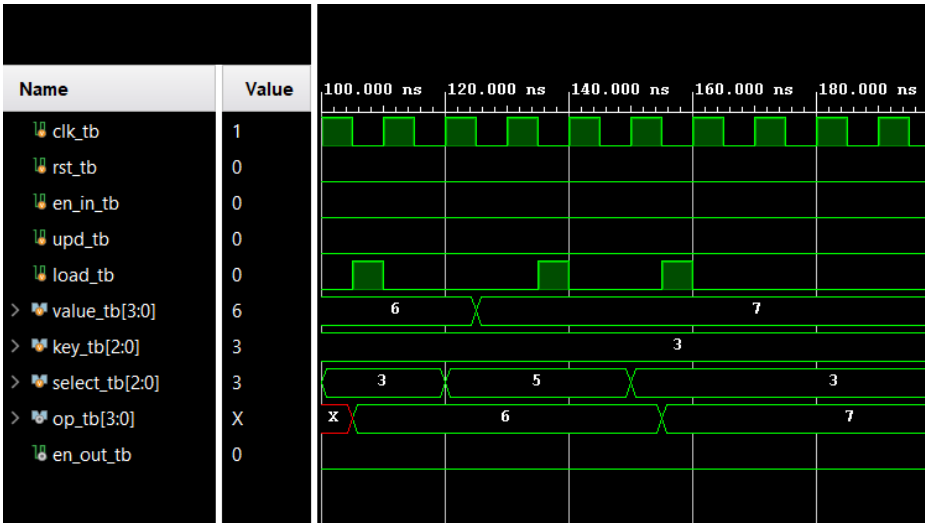
top.v

Test Bench

A testbench for the new *bcd_counter.v* was made to test the load function.

```
`timescale 1ns / 1ps
module bcd_counter_tb(

    );

                    reg clk_tb;
                    reg rst_tb;
                    reg en_in_tb;
                    reg upd_tb;
                    reg load_tb;
                    reg [3:0] value_tb;
                    reg [2:0] key_tb;
                    reg [2:0] select_tb;
                    wire [3:0] op_tb;
                    wire en_out_tb;

                    bcd_counter test
                    (
                    .clk(clk_tb),
                    .rst(rst_tb),
                    .en_in(en_in_tb),
                    .upd(upd_tb),
                    .value(value_tb),
                    .key(key_tb),
                    .select(select_tb),
                    .load(load_tb),
                    .op(op_tb),
                    .en_out(en_out_tb)
                    );

    initial
        begin
            clk_tb<=1'b1;
            forever #5 clk_tb<=~clk_tb;
        end

initial
    begin
    en_in_tb<=0;
    key_tb<=3'd3;
    value_tb<=4'd6;
    load_tb<=0;
    select_tb<=3'd3;
    upd_tb<=1'b0;
    rst_tb<=1'b0;#5
    rst_tb<=1'b1;#5
    rst_tb<=1'b0;#50;
    load_tb<=1'b1;#5
    load_tb<=1'b0;#10
    select_tb<=3'd5;
    value_tb<=4'd6; #10
    load_tb<=1'b1;#5
    load_tb<=1'b0;#10;
    select_tb<=3'd3;#5
     load_tb<=1'b1;#5
    load_tb<=1'b0;#10;
    select_tb<=3'd5;#5
    value_tb<=4'd7; #10
     load_tb<=1'b1;#5
    load_tb<=1'b0;#10;
    select_tb<=3'd3;#5
     load_tb<=1'b1;#5
    load_tb<=1'b0;#10;
    end
endmodule
```

bcd_counter_tb.v



The output waveform shows that the output only changes at every *load_tb* positive edge when *value_tb* is equal to *key_tb*. When they are not equal the output stays constant. This is a model for a single BCD counter.

Corner Cases

Corner cases for this lab include displaying numbers properly instead of random numbers when the counter direction is changed. If the up-down switch counter is enabled, numbers must start at the highest BCD number and reset once the number reaches the smallest value 0. Similarly, when the down-up switch is enabled the BCD numbers should reset when the highest number is reached, in this case, 9. Proper testing is also required to properly display the counter, using a high frequency will cause the seven segment displays to resemble a display with all LEDs turned on. Additionally with the new button added to the environment, unexpected behaviour is possible, such as counter not stopping/continuing when the load button is enabled. This corner case will be tested during the demo section.

Technical Report

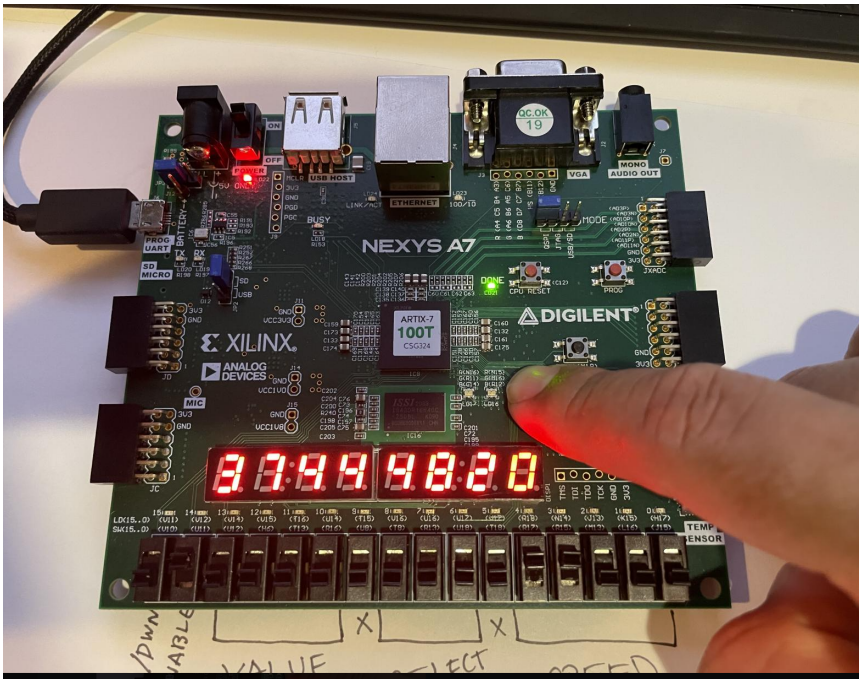
After the simulation was performed. The results obtained from the report are as follows. The total power consumption is 0.109W, 179 LUT's and 121 Flip-Flops. The power specifications for this lab were met. The number of LUT's and Flip-Flops increased from the previous lab as an extra function was added to each BCD counter.

Synthesis results for design

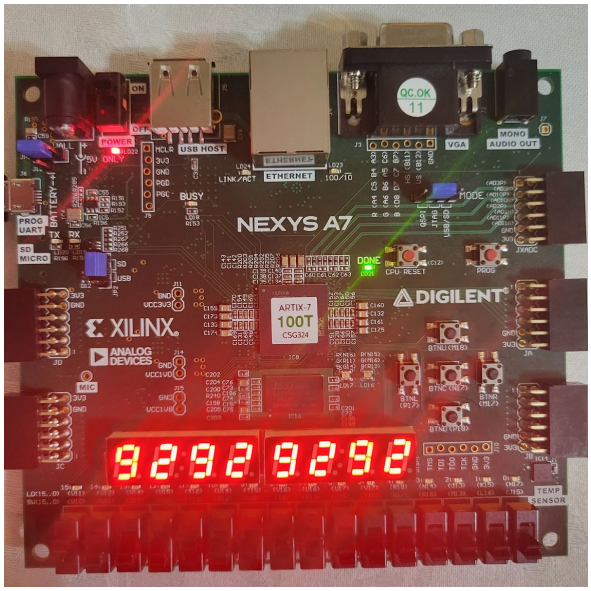
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Synthesis Out-of-date								181	121	0.0	0	0	10/18/21, 6:03 PM	00:00:30	Vivado Synthesis De
impl_1	constrs_1	Implementation Out-of-date	7.095	0.000	0.319	0.000	0.000	0.109	0	179	121	0.0	0	0	10/18/21, 6:04 PM	00:01:09	Vivado Implementa

After generating bitstream, the file was transferred to the Nexys 7. The behavior of the programmed board was identical to that of the required specifications and the switches, buttons & seven-segment display worked as expected. The counter displayed numbers properly under normal circumstances & when any number was injected into any of the seven segments displays.

Programmed Nexys 7



Number 0 was injected into the first (from right to left) seven-segment display



Pattern loaded in Nexys A7

Conclusion

After programming the board the functionality was the same as in the previous lab with the addition of the 3 selector switches for the seven-segment display, the four switches controlling the value to be injected and the additional button that injects the new value into any of the seven-segment displays. Corner cases were tested and the counter stopped when the injection button was pressed. Similarly, when the injection button was released, the counter continued from the number that was injected.