浙江大学

本科实验报告

课程名称：计算机组成

姓　　名：Parsa 帕萨

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：中加班

学　　号：3170300180

指导教师：姜晓红

2019 年　　03 月　　15 日

浙江大学实验报告

课程名称：计算机组成 实验类型：综合

实验项目名称：Lab1 and 2: Controlling 8 seven segments

学生姓名：Parsa 帕萨 专业：中加班 学号：3170300180

同组学生姓名：None 指导老师：姜晓红

实验地点：东 4-510 实验日期：2019 年 03 月 15 日

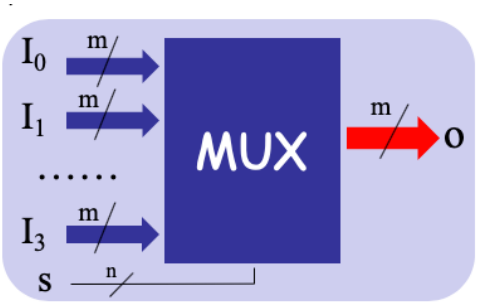
1 – Purpose of the Experiments and Tasks

i. Design and implement an 8-1 multiplexer with a switch input
ii. Design and implement the Multi_8CH32 module
iii. Design and implement the clk_div module
iv. Design and implement the Display module
v. Design and implement SPIO module
vi. Design and implement a RAM and ROM module
vii. Draw or implement the top module given

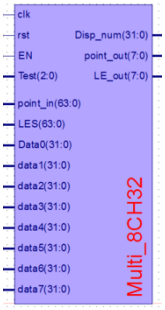
2 – Instructions and Experiment Contents

Multiplexers

Multiplexers are modules/components that given inputs and outputs with it will set the outputs to be different inputs depending on the state of the selector input. In this lab we are expected to design two multiplexers. First multiplexer is a simple 1-to-8 mux which is a multiplexer that takes 8 inputs, 3 inputs for the selector and 8 outputs.



The second is a more complicated 192-to-48 mux (Multi_8CH32) with a clock and an enabler switch. There is a 3-bit switch which selects 8-bits from point_in data, 8-bits from LES and the appropriate 32-bits data (data0-data7) according to the selector switch, the outputs are directed to point_out, LE_out and Disp_num accordingly.

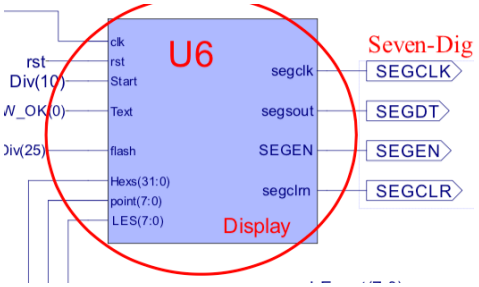


Clock Module

We are also expected to design and implement a clock module based on the 100mhz on-board clock. The clock is expected to have an input straight from the on-board clock, rst switch and a switch from the board that sets the CPU clock. The outputs are an array of clk_div and CPU clock. If rst switch is on, then the module should reset the clock otherwise should add one to the clkdiv wire based on the clock cycle of the on-board clock. If the other switch is on then the CPU clk should have a frequency of 50MHz or 25MHz, otherwise it should correspond to clkdiv[24].

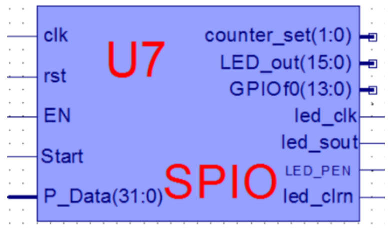
Display Module

There are going to be two modules similar to this, we are expected to design one (U6) that takes clk, rst, Start, a switch, flash, Hexs, point and LES as inputs and it should output segclk, segout, segpen and segclrn. The one we are expected to design is then connected to eight seven segments which would display the inputs. The other module is just used for debugging purposes which is connected to Arduino's 4 seven segments. The second module is called Seg7_dev and has Scan, SW, flash, Hexs, point and LES as inputs Segment and AN as outputs. The second module (U61) controls the seven segments parallelly, while the first module (U6) uses a module called P2S which converts the parallel inputs into series.



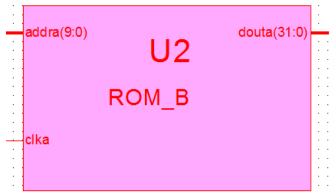
SPIO module

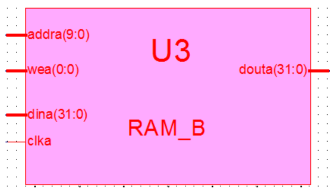
SPIO is a parallel LED controller module. It takes clk, rst, EN, start, P_data as inputs and outputs led_clk, led_sout, LED_PEN and led_clrn.



ROM and RAM modules

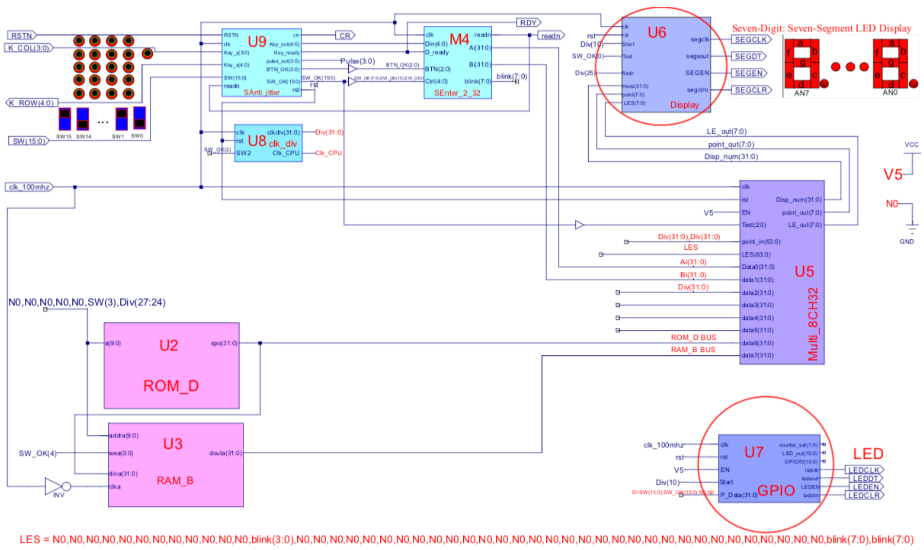
We are expected to generate a ROM module which as its name suggests a fixed pattern for displaying on the 7-segments and a RAM to store the state of the displays. Both Ram and ROM are required to have a size of 1024x32bits. The contents of the ROM are given and are used to display graphics using the 7-segments. RAM is used to store the contents of the displays as they're getting modified for future uses.





Top Module

Once the mentioned modules are designed, we are expected to replicate the following diagram. All modules are provided as 'black boxes' for point of reference. We are encouraged to complete the top module using the provided modules to see the expected results and then design the aforementioned components. In the next section each module will be discussed in detail.



3 – Implementations and results

Multiplexers

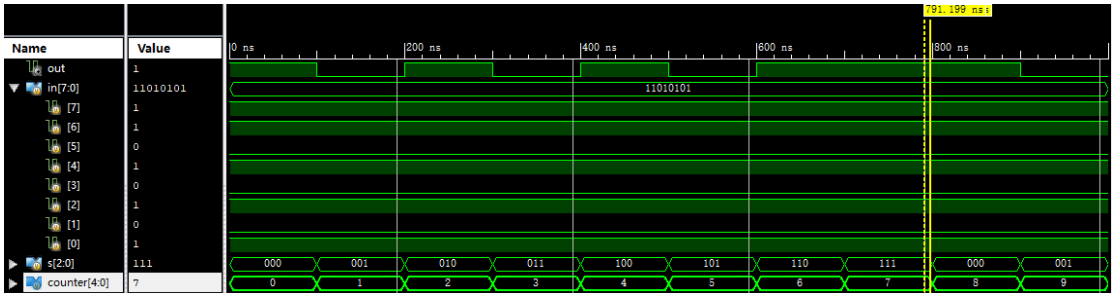
8-to-1 Multiplexer

According to this table, given that we have 8 inputs, we need 3 selectors to select from these input and direct them to our only output. A simple implementation of this module is that given the value of selector in decimal, output is $In_{selector}$. The implementation of this module is as follow:

Input	Selector	Output
$In_7In_6In_5In_4In_3In_2In_1In_0$	000	In_0
$In_7In_6In_5In_4In_3In_2In_1In_0$	001	In_1
$In_7In_6In_5In_4In_3In_2In_1In_0$	010	In_2
$In_7In_6In_5In_4In_3In_2In_1In_0$	011	In_3
$In_7In_6In_5In_4In_3In_2In_1In_0$	100	In_4
$In_7In_6In_5In_4In_3In_2In_1In_0$	101	In_5
$In_7In_6In_5In_4In_3In_2In_1In_0$	110	In_6
$In_7In_6In_5In_4In_3In_2In_1In_0$	111	In_7

```
module mux81(input wire [7:0]in,
             input wire [2:0]s,
             output wire out
    );
   assign out = in[s];
endmodule
```

As it can be seen in the simulation, the multiplexer works as expected. We have 11010101 as our input and every 100ms we add one bit to our selector to select the next input and the output corresponds to the right input.



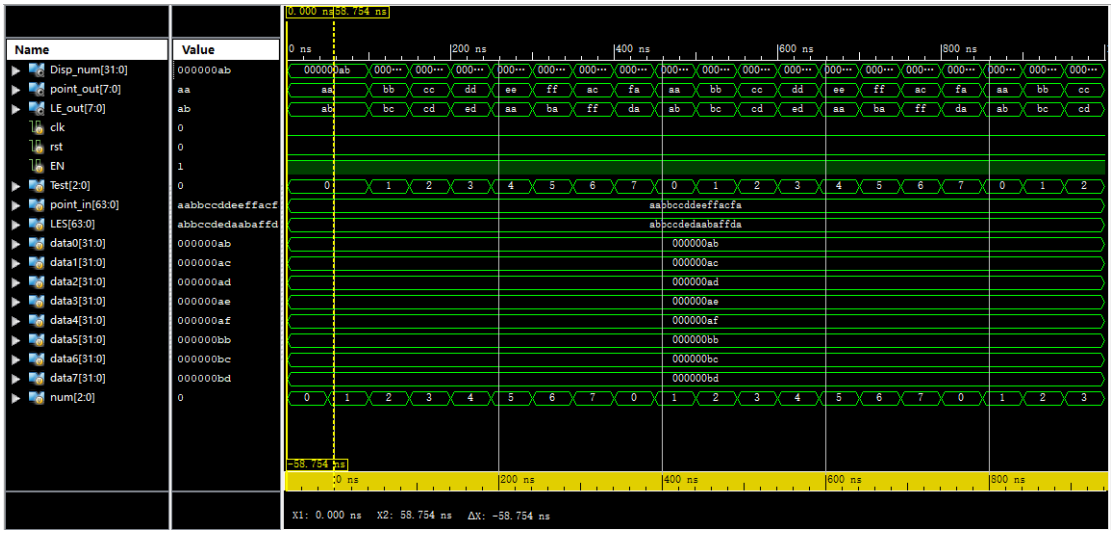
Multi_8CH32

The second multiplexer we are asked to design/implement is the Multi_8CH32 multiplexer. This multiplexer is a combination of 3 multiplexers, though it can be implemented easily without using the other 3 multiplexers. This multiplexer takes a clock, reset switch, enabler switch, Test (our selector), point_in, LES and 8 entry data, then depending on our selector, it outputs display_num, point_out and LE_out. As mentioned before, this multiplexer can be broken down into 3 different multiplexers, one for point_in (64-to-8 mux), one for LES (64-to-8 mux) and one for data inputs(256-to-32) mux. The selector (Test input) then must go to each one of them and the outputs are chosen accordingly. Another alternative to this is implementing

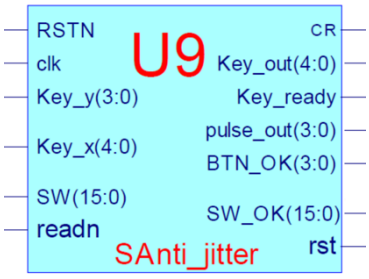
an all-in-one multiplexer. This can be done by selecting 8-bits from both point_in and LES and then appropriate data based on test. The implementation of this part is as follow:

```
module Multi_8CH32(input wire clk,
            input wire rst,
            input wire EN,
            input wire [2:0] Test,
            input wire [0:63] point_in,
            input wire [0:63] LES,
            input wire [0:31] data0,
            input wire [0:31] data1,
            input wire [0:31] data2,
            input wire [0:31] data3,
            input wire [0:31] data4,
            input wire [0:31] data5,
            input wire [0:31] data6,
            input wire [0:31] data7,
            output reg [0:31] Disp_num,
            output reg [0:7] point_out,
            output reg [0:7] LE_out
    );
always @* begin
  if (EN)
    begin
      if (Test == 0)begin
        Disp_num <= data0;
        LE_out <= LES[0:7];
        point_out <= point_in[0:7];
      end
      else if (Test == 1)begin
        Disp_num <= data1;
        LE_out <= LES[8:15];
        point_out <= point_in[8:15];
      end
      else if (Test == 2)begin
        Disp_num <= data2;
        LE_out <= LES[16:23];
        point_out <= point_in[16:23];
      end
      else if (Test == 3)begin
        Disp_num <= data3;
        LE_out <= LES[24:31];
        point_out <= point_in[24:31];
      end
      else if (Test == 4)begin
        Disp_num <= data4;
        LE_out <= LES[32:39];
        point_out <= point_in[32:39];
      end
      else if (Test == 5)begin
        Disp_num <= data5;
        LE_out <= LES[40:47];
        point_out <= point_in[40:47];
      end
      else if (Test == 6)begin
        Disp_num <= data6;
        LE_out <= LES[48:55];
        point_out <= point_in[48:55];
      end
      else if (Test == 7)begin
        Disp_num <= data7;
        LE_out <= LES[56:63];
        point_out <= point_in[56:63];
      end
    end
  end

endmodule
```

Here is the simulation of this module:



Santi Jitter Module



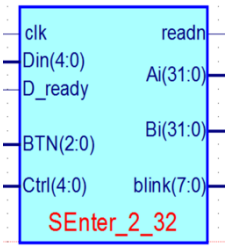
The input buttons usually have a lot of noise when they're pressed. Using this module, we can reduce the input noise received from the keypad. The module takes inputs from the switches and keypad and produces signals corresponding to the input signals. This module takes RSTN, clk, Key_y, SW and readn as inputs and outputs Key_x, CR, Key_out, pulse_out, BTN_Ok, SW_ok and rst. This module is referred to as U9 in the top module and is provided.

Clk_div module

This module takes the on-board clock from and generates a clock cycle and also creates a new clock called CPU clock. This module should have an input that controls the CPU clock, which comes from SW2. When SW2 is on then the module should direct the clkdiv[24] to the CPU clock otherwise it should redirect clkdiv[2] to the Clock CPU. When rst, reset is on then the clock should reset to zero otherwise it should keep adding one to its previous state by each cycle of the on-board clock. The design of this module is as follow:

```
module clk_div(input clk,
               input rst,
               input SW2,
               output reg [31:0]clkdiv,
               output wire Clk_CPU
               );
   always @ (posedge clk or posedge rst) begin
        if (rst) clkdiv <= ~clkdiv ;
        else clkdiv <= clkdiv+1'b1;
   end
   assign Clk_CPU=(SW2) ? clkdiv[24] : clkdiv[2];
endmodule
```

SEnter_2_32

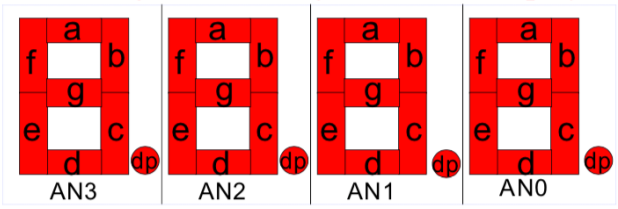


This module takes the outputs of the Santi_Jitter module and updates the numbers on the screen based on the switches. The output of this module is then sent to our multiplexer. This module is denoted as M4 in the top module.

Display

As mentioned, there are two display modules, U6 – Display and U61 – Seg_Dev modules. U61 is used for debugging purposes and is connected to the Arduino's 4 7Segments. Since we can't replicate the same output as our 8 7-segments on our Arduino's module we divide the segments into two sections, the right and left section. Each section has 4 7Segments and using a switch we can display 4 segments at the same time from our 8 segments, and by turning on the switch the other 4 can be displayed. Unlike our U6 module, the U61 sends the signals parallelly. So we have two outputs, an array of 8 for controlling each segment within the segment and AN for

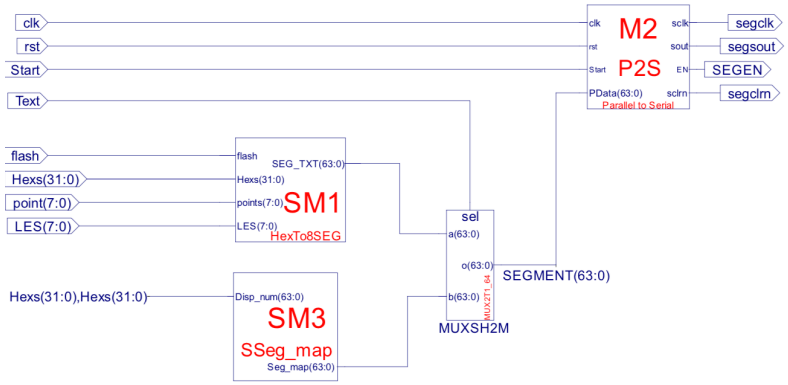
controlling the common anode of segments. Consider the following diagram:



Let's say we want to display number 1 on the first (right-most 7segment and) and 2 on the second one (second right-most one). To do that, we have set the AN_0 and AN_1 to be 1 and for the other ones we should set them to zero. Then by setting segments *a* and *b* to 1 we can display 1 on the first segment, and by setting *a, b, d, e* and *g* to 1 we can display 2 on the second one. LES, point, Hexs, flash, SW0, and Scan are given as inputs. LES and point control the which segments and decimal points to turn on and Hexs is the incoming data from our multiplexer, Multi_8CH32 which we want to display on both 7-segments. This module is given, and we are not expected to design it.

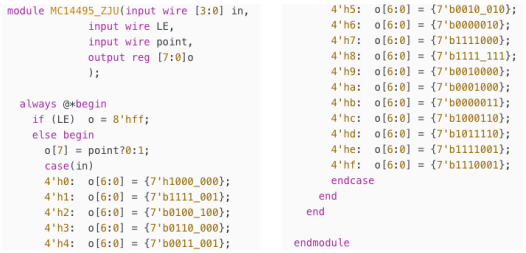
Display

This module consists of 4 modules, HexTo8SEG, SSeg_map, Mux2T1_64 and P2S. This module also has clk, rst, Start, Text, flash, Hexs, point, LES as inputs and segclrn, SEGEN, Segsout, segclk as outputs.



The HexTo8Seg gets LES, point and HEXs as inputs and generate a signal based on the hexs for each segment to be turned on. To do so, it requires a MC14495 module which is decoder. This means that given any inputs it will output which segments should turn on. For example, we if want to display number two on the segments; as discussed, we should turn on *a, b, d, e* and *g* elements which can be encoded to 01011011. Our flash

and LES are used as switches to turn on the segment and point is just whether we want to turn on the decimal point of that segment. Here is the implementation of MC14995:

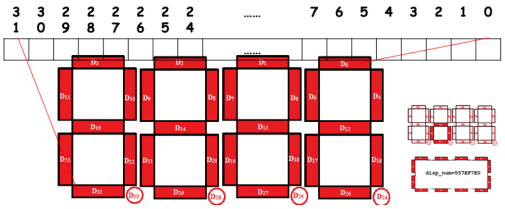


```
module MC14495_ZJU(input wire [3:0] in,
                   input wire LE,
                   input wire point,
                   output reg [7:0]o
                   );

 always @*begin
   if (LE)  o = 8'hff;
   else begin
     o[7] = point?0:1;
     case(in)
     4'h0:  o[6:0] = {7'h1000_000};
     4'h1:  o[6:0] = {7'b1111_001};
     4'h2:  o[6:0] = {7'b0100_100};
     4'h3:  o[6:0] = {7'b0110_000};
     4'h4:  o[6:0] = {7'b0011_001};
     4'h5:  o[6:0] = {7'b0010_010};
     4'h6:  o[6:0] = {7'b0000010};
     4'h7:  o[6:0] = {7'b1111000};
     4'h8:  o[6:0] = {7'b1111_111};
     4'h9:  o[6:0] = {7'b0010000};
     4'ha:  o[6:0] = {7'b0001000};
     4'hb:  o[6:0] = {7'b0000011};
     4'hc:  o[6:0] = {7'b1000110};
     4'hd:  o[6:0] = {7'b1011110};
     4'he:  o[6:0] = {7'b1111001};
     4'hf:  o[6:0] = {7'b1110001};
     endcase
   end
 end

endmodule
```

Once we have MC14995 implemented then we can use it to design our HexTo8Seg, here is its implementation:

```
module HexTo8Seg(input wire flash,
            input wire [7:0]LES,
            input wire [7:0]point,
            input wire [31:0]Hexs,
            output wire [63:0] SEG_TXT);


    wire And0 = flash & ~LES[0];
    wire Point0= ~point[0];
    MC14495_ZJU m0(.in(Hexs[3:0]), .LE(And0), .point(Point0), .o(SEG_TXT[7:0]));


    wire And1 = flash & ~LES[1];
    wire Point1= ~point[1];
    MC14495_ZJU m1(.in(Hexs[7:4]), .LE(And1), .point(Point1), .o(SEG_TXT[15:8]));

    wire And2 = flash & ~LES[2];
    wire Point2= ~point[2];
    MC14495_ZJU m2(.in(Hexs[11:8]), .LE(And2), .point(Point2), .o(SEG_TXT[23:16]));

    wire And3 = flash & ~LES[3];
    wire Point3= ~point[3];
    MC14495_ZJU m3(.in(Hexs[15:12]), .LE(And3), .point(Point3), .o(SEG_TXT[31:24]));

    wire And4 = flash & ~LES[4];
    wire Point4= ~point[4];
    MC14495_ZJU m4(.in(Hexs[19:16]), .LE(And4), .point(Point4), .o(SEG_TXT[39:32]));

    wire And5 = flash & ~LES[5];
    wire Point5= ~point[5];
    MC14495_ZJU m5(.in(Hexs[23:20]), .LE(And5), .point(Point5), .o(SEG_TXT[47:40]));

    wire And6 = flash & ~LES[6];
    wire Point6= ~point[6];
    MC14495_ZJU m6(.in(Hexs[27:24]), .LE(And6), .point(Point6), .o(SEG_TXT[55:48]));

    wire And7 = flash & ~LES[7];
    wire Point7= ~point[7];
    MC14495_ZJU m7(.in(Hexs[31:28]), .LE(And7), .point(Point7), .o(SEG_TXT[63:56]));

endmodule
```

The other module is SSeg_map which takes Hexs as inputs and turns on the corresponding segment based on the Hexs. The locations of segments are hard-coded in this module and are based on the following diagram:



So, the module's implementation is as follow:

```
module SSeg_map(input wire [31:0] Hexs,
          output wire [0:63] Seg_map
    );

   assign Seg_map =
        {Hexs[0], Hexs[4],Hexs[16], Hexs[25], Hexs[17], Hexs[5], Hexs[12], Hexs[24],
        Hexs[1], Hexs[6],Hexs[18], Hexs[27], Hexs[19], Hexs[7], Hexs[13], Hexs[26],
        Hexs[2], Hexs[8],Hexs[20], Hexs[29], Hexs[21], Hexs[9], Hexs[14], Hexs[28],
        Hexs[3], Hexs[10],Hexs[22], Hexs[31], Hexs[23], Hexs[11], Hexs[15], Hexs[30],

        Hexs[0], Hexs[4],Hexs[16], Hexs[25], Hexs[17], Hexs[5], Hexs[12], Hexs[24],
        Hexs[1], Hexs[6],Hexs[18], Hexs[27], Hexs[19], Hexs[7], Hexs[13], Hexs[26],
        Hexs[2], Hexs[8],Hexs[20], Hexs[29], Hexs[21], Hexs[9], Hexs[14], Hexs[28],
        Hexs[3], Hexs[10],Hexs[22], Hexs[31], Hexs[23], Hexs[11], Hexs[15], Hexs[30]};

endmodule
```

Once we have these two modules designed, then we need a multiplexer, MUX2T1_64, to choose the mode given the outputs of these two modules. Here is the implementation of this module:

```
module MUX2T1_64(input wire sel,
            input wire [63:0] a,
            input wire [63:0] b,
            output wire [63:0] o
    );
   assign o = sel? a: b;


endmodule
```

As discussed, this module takes the Text input of the display module and use it as a selector, if it's 1 then it directs the output of SSeg_map to the P2S module, otherwise it directs the output of HexTo8Seg. Once, we have all these modules then top module for Display can be designed based on the given schematic. Here is the implementation of Display module:

```
module Display(input wire clk,
          input wire rst,
          input wire Start,
          input wire Text,
          input wire flash,
          input wire [31:0] Hexs,
          input wire [7:0]point,
          input wire [7:0]LES,
          output wire segclk,
          output wire segsout,
          output wire SEGEN,
          output wire segclrn
    );

  wire [63:0] a, b;
  HexTo8Seg SM1(.flash(flash), .Hexs(Hexs), .point(point), .LES(LES), .SEG_TXT(a));
  SSeg_map SM3(.Hexs(Hexs), .Seg_map(b));
  wire [63:0]SEGMENTS;
  MUX2T1_64 MUXSH2M(.sel(Text), .a(a), .b(b), .o(SEGMENTS));
  P2S M2(.clk(clk), .rst(rst), .Serial(Start),.P_Data(SEGMENTS), .s_clk(segclk),
         .sout(segsout), .EN(SEGEN),.s_clrn(segclrn));

endmodule
```

GPIO

The next module to be designed is GPIO which is a LED driver module. This module is used to control the LEDs on the board. The implementation of this module is as follow:

```
module GPIO(input wire clk,
        input wire rst,
        input wire EN,
        input wire Start,
        input wire [31:0] P_Data,
        output reg [1:0] counter_set,
        output wire [15:0] LED_out,
        output reg [13:0] GPIOf0,
        output wire ledclk,
        output wire ledsout,
        output wire LEDEN,
        output wire ledclrn
    );
   reg [15:0] LED;
   assign LED_out = LED;
   always @(negedge clk or posedge rst)begin
    if (rst)begin
      LED <= 8'h2a;
      counter_set <= 2'b00;
    end
    else if (EN)begin
      {GPIOf0, LED, counter_set} <= P_Data;
    end
    else begin
      LED <= LED;
      counter_set <= counter_set;
    end
  end
  LEDP2S #(.DATA_BITS(16), .DATA_COUNT_BITS(4))
        M1(clk, rst, Start, {LED}, ledclk, ledclrn, ledsout, LEDEN);

endmodule
```

ROM and RAM

The ram is just a regular ram module, but the ROM should be initialized with this data:

```
memory_initialization_radix=16;
memory_initialization_vector=
00000000, 11111111,
22222222, 33333333, 44444444, 55555555, 66666666, 77777777, 88888888, 99999999, aaaaaaaa, bbbbbbbb,
cccccccc, dddddddd, eeeeeeee, ffffffff, 557EF7E0, D7BDFBD9, D7DBFDB9, DFCFFCFB, DFCFBFFF, F7F3DFFF,
FFFFDF3D, FFFF9DB9, FFFFBCFB, DFCFFCFB, DFCFBFFF, D7DB9FFF, D7DBFDB9, D7BDFBD9, FFFF07E0, 007E0FFF,
03bdf020, 03def820, 08002300;
```

These two modules should match the specifications mentioned in the second section.

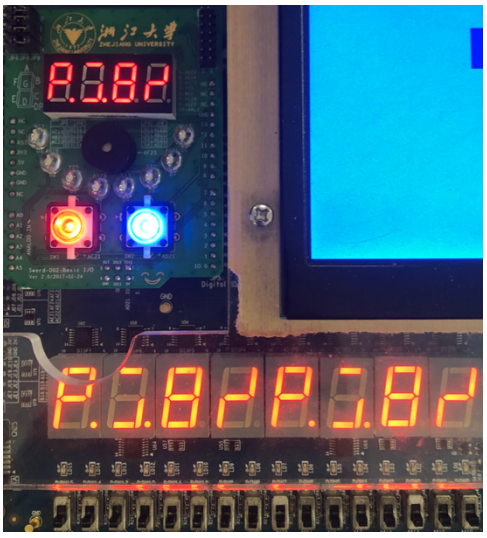
Top Module

Once we have all those modules designed, we can replace the given components with the ones that we designed. The result should look like this:

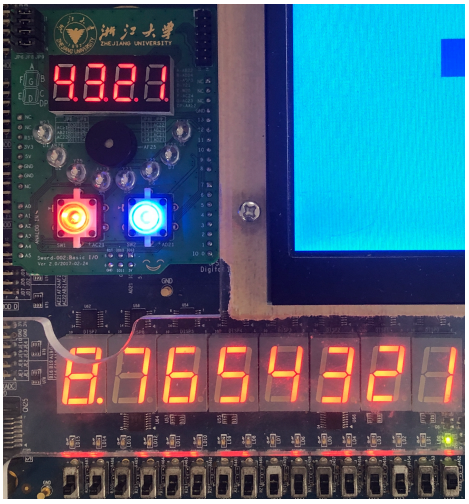
```
module top(
    input wire RSTN,
    input wire [3:0] BTN_y,
    output wire [4:0] BTN_x,
    input wire [15:0] SW,
    input wire clk_100mhz,
    output wire CR,
    output wire RDY,
    output wire readn,
    output wire seg_clk,
    output wire seg_sout,
    output wire SEG_PEN,
    output wire seg_clrn,
    output wire [7:0]SEGMENT,
    output wire [3:0]AN,
    output wire [7:0]LED,
    output wire Buzzer,
    output wire LEDCLK,
    output wire LEDDT,
    output wire LEDEN,
    output wire LEDCLR
  );

  wire [4:0] U9_Keyout;
  wire [3:0] Pulse;
  wire [3:0] U9_BTNOK;
  wire [15:0] U9_SWOK;
  wire rst;
  SAnti_jitter U9(.RSTN(RSTN), .clk(clk_100mhz), .Key_y(BTN_y), .Key_x(BTN_x),
           .SW(SW), .readn(readn), .CR(CR), .Key_out(U9_Keyout), .Key_ready(RDY),
           .pulse_out(Pulse), .BTN_OK(U9_BTNOK), .SW_OK(U9_SWOK), .rst(rst));
  wire [31:0] U8_div;
  wire U8_clkCPU;
  clk_div U8(.clk(clk_100mhz), .rst(rst), .SW2(U9_SWOK[2]), .clkdiv(U8_div), .Clk_CPU(U8_clkCPU));

  wire [31:0] M4_Ai;
  wire [31:0] M4_Bi;
  wire [7:0] M4_blink;
  SEnter_2_32 M4(.clk(clk_100mhz), .Din(U9_Keyout), .D_ready(RDY), .BTN(U9_BTNOK[2:0]),
            .Ctrl({U9_SWOK[7:5], U9_SWOK[15], U9_SWOK[0]}), .readn(readn), .Ai(M4_Ai),
            .Bi(M4_Bi), .blink(M4_blink));
```

```
  wire [7:0] U5_leout;
  wire [7:0] U5_pointout;
  wire [31:0] U5_dispnum;

  Display U6(.clk(clk_100mhz), .rst(rst), .Start(U8_div[20]), .Text(U9_SWOK[0]), .flash(U8_div[25]),
          .Hexs(U5_dispnum), .point(U5_pointout), .LES(U5_leout), .segclk(seg_clk), .segsout(seg_sout),
          .SEGEN(SEG_PEN), .segclrn(seg_clrn));

  wire [31:0]ROM_D_Bus;
  wire [31:0]RAM_B_Bus;
  wire [9:0] nihao_1 = {5'b0, SW[3],U8_div[27:24]};
  wire invClk = ~clk_100mhz;
  ROM_B U2(.a(nihao_1),.spo(ROM_D_Bus));
  RAM_B U3(.clka(invClk), .wea(U9_SWOK[4]),  .addra(nihao_1), .dina(ROM_D_Bus), .douta(RAM_B_Bus));
  wire notLEDDT;
  assign LEDDT = ~notLEDDT;

  GPIO U7 (.clk(clk_100mhz), .rst(rst), .EN(1'b1), .Start(U8_div[10]), .P_Data({SW[13:0], U9_SWOK[15:0], 2'b0}),
  .counter_set(counter_set), .LED_out(led_out_U7), .GPIOf0(GPIOf0), .ledclk(LEDCLK), .ledsout(notLEDDT),
  .LEDEN(LEDEN), .ledclrn(LEDCLR));

  Multi_8CH32 U5(.clk(clk_100mhz), .rst(rst), .EN(1'b1), .Test(U9_SWOK[7:5]), .point_in({U8_div[31:0], U8_div[31:0]}),
            .LES(LES), .Data0(M4_Ai), .data1(M4_Bi), .data2(U8_div), .data3(32'b0), .data4(32'b0), .data5(32'b0),
            .data6(ROM_D_Bus), .data7(RAM_B_Bus), .Disp_num(U5_dispnum), .point_out(U5_pointout), .LE_out(U5_leout));


  Seg7_Dev U61(.Scan({U9_SWOK[1], U8_div[19:18]}), .SW0(U9_SWOK[0]), .flash(U8_div[25]),
    .Hexs(U5_dispnum), .point(U5_pointout), .LES(U5_leout), .SEGMENT(SEGMENT), .AN(AN));

  assign Buzzer = 1'b0;

  PIO U71(.clk(clk_100mhz), .rst(1'b0), .EN(1'b1), .PData_in(M4_Ai), .LED_out(LED));

endmodule
```

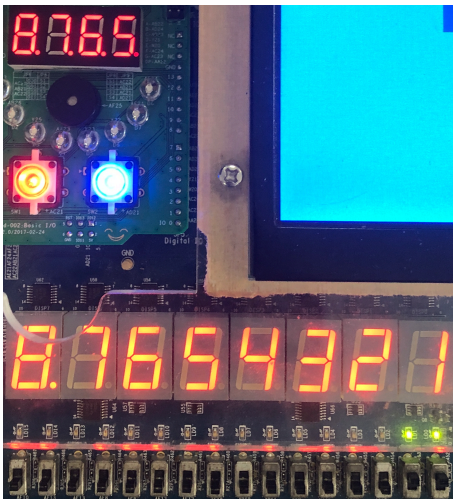
Results



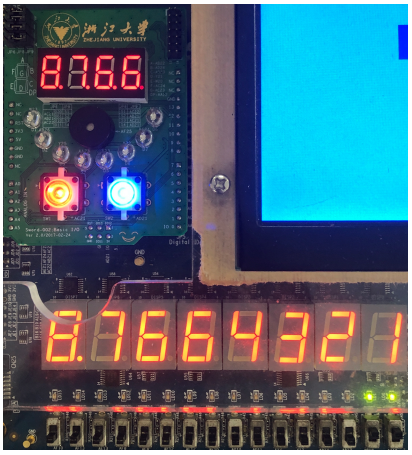
When SW[0] is off



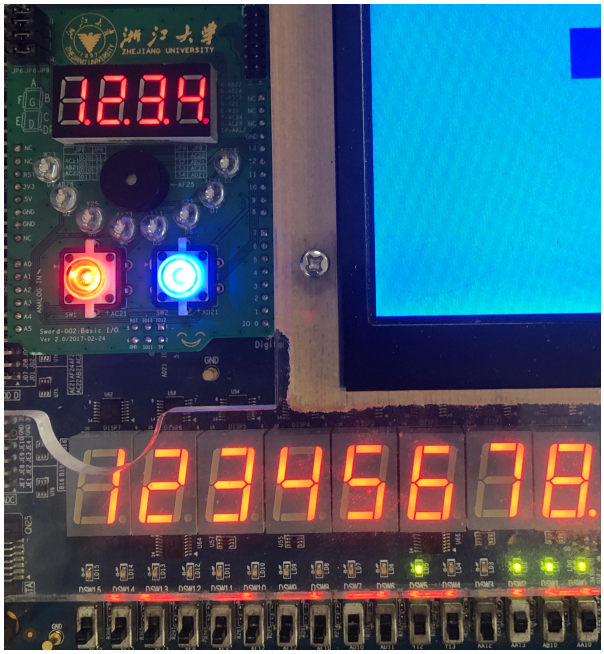
When SW[0] is on and SW[1] is off, right half is shown



When SW[0] is on and SW[1] is on, left half is shown



Displays can be modified by the KeyPad, notice that the fifth segment has changed to 6



When Switches [0], [2] and [5] are on this is shown. Switch[1] is used to change which screen to be displayed on the our debugging screen.

The results of text, clock and synchronised modes are attached as gif files to this report.

4 – Discussion

I encountered few difficulties during designing the display module, one of which was that the given map for SSeg_map was inversed of what I had in mind, so I had to inverse my mappings. After completing the design of Display module, I noticed a lot of mismatching characters and random flashing, after further investigation I realized that the source of all of this was the faulty P2S module, which we were then provided and after replacing the module, everything worked as expected.