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　　29 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： Lab 3: Design of CPU and setting up SOC test environment

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 29 日

1 – Purpose of the Experiments and Tasks

i. Implement and design a SOC based on the given components
ii. Design and implement the ROM and RAM
iii. Understand the usage of each component and the circuitry the SOC

2 – Instructions and Experiment Contents

This experiment is built on top of the previous experiment where we had a ROM and a RAM and different modes defined, which we could control them by Switches 5 to 7. This experiment is a step closer to designing a proper CPU and SoC. This experiment has 3 new components, MIO_BUS, Counter and SCPU and different contents for the RAM and ROM. The purpose of existing are not explained since they were explained in the previous lab report but rather the purpose of each new module is explained.

Let's start off with the ROM and RAM. They both have the same sizes, 1024x32bits. But for this experiment they are initialized differently. For the rom we have the following initializer:

```
;cpu_IO_test1.asm
memory_initialization_radix=16;
memory_initialization_vector=
08000008, 00000020, 00000020, 00000020, 00000020, 00000020, 00000020, 00000020, 00000827, 00211820,
00631820, 00631820, 00631820, 00631820, 00631820, 0060a027, 00631820, 00631820, 00631820, 00631820,
00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00631820,
00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00633020, 00c61820, 00632020, 00846820,
01ad4020, 0001102a, 00427020, 01ce7020, 00005027, 014a5020, ac660004, 8c650000, 00a52820, 00a52820,
ac650000, 01224820, ac890000, 8c0d0014, 8c650000, 00a52820, 00a52820, ac650000, 8c650000, 00a85824,
01a26820, 11a00017, 8c650000, 01ce9020, 0252b020, 02569020, 00b25824, 11600005, 1172000a, 01ce9020,
1172000b, ac890000, 08000036, 11410001, 0800004d, 00005027, 014a5020, ac8a0000, 08000036, 8e290060,
ac890000, 08000036, 8e290020, ac890000, 08000036, 8c0d0014, 014a5020, 01425025, 022e8820, 02348824,
01224820, 11210001, 0800005f, 000e4820, 01224820, 8c650000, 00a55820, 016b5820, ac6b0000, ac660004,
8c650000, 00a85824, 0800003e;
```

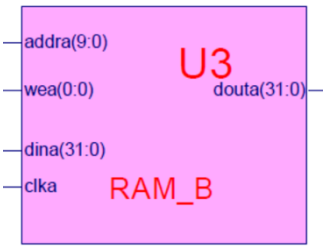
The first line is just a comment indicating that this file is going to be used by CPU_IO assembly module(?) and the second line indicates that vector is going to be in hexadecimal form, and the third line is just an array which is going to be written in the ROM. As mentioned, this block is 1024x32 bits meaning that there are 1024 fields with each having the size of 32 bits. Given an address we can output its corresponding memory content. We don't necessary need 1024 field, in this case 128 fields are sufficient enough, meaning that we only need an output of 7 bits to access the contents of the ROM. Since we are generating a 1024x32bits ROM we need an input of 10bits and an output of 32bits, which is the content of the given memory address. The contents of the ROM correspond to the value to be displayed on each segment, hence the size of 8x4bits (8 hexadecimal values).

Similarly, we can initialize the RAM. The COE of RAM is also given and is as follow:

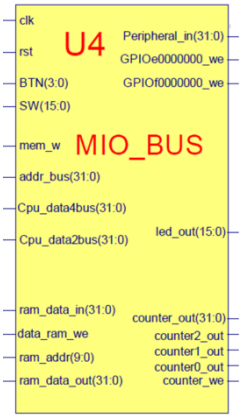
```
memory_initialization_radix=16;
memory_initialization_vector=
f0000000, 000002AB, 80000000, 0000003F, 00000001, FFF70000, 0000FFFF, 80000000, 00000000, 11111111,
22222222, 33333333, 44444444, 55555555, 66666666, 77777777, 88888888, 99999999, aaaaaaaa, bbbbbbbb,
cccccccc, dddddddd, eeeeeeee, ffffffff, 557EF7E0, D7BDFBD9, D7DBFDB9, DFCFFCFB, DFCFBFFF, F7F3DFFF,
FFFFDF3D, FFFF9DB9, FFFFBCFB, DFCFFCFB, DFCFBFFF, D7DB9FFF, D7DBFDB9, D7BDFBD9, FFFF07E0, 007E0FFF,
03bdf020, 03def820, 08002300;
```

Unlike ROM, RAM requires a clock input, along that it requires an input which indicates whether we want to read or write data, and if we are going to write to the memory what is our

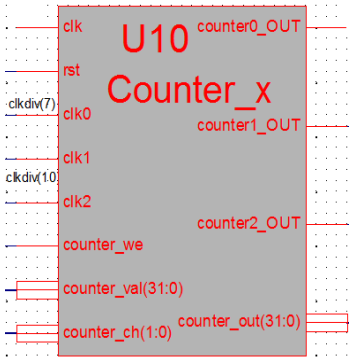
data. The rest is similar to ROM. While generating the RAM we can specify whether it's "write first" or "read first" this means if wea = 1 is whether it's going to write to the memory or read from the memory. In this case, we have it set to write first.



Next, we have BUS_MIO. It is a multiplexed I/O component which routes the inputs to their corresponding outputs based on the selector. MIO are usually used to redirect or route the I/O, peripherals and bus pins without them going to the processors or FPGA. In this case we are using this component to send our inputs to the RAM, ROM, GPIO, SCPU and the counter. The selectors are BTN and SW which are outputs of our AntiJitter module which takes the inputs from on-board switches and keypads as inputs and makes sure that there aren't any noises. As can be seen from the MIO_BUS_IO module we have clk, rst, mem_w, Cpu_data2bus, addr_bus, ram_data_out, led_out, counter_out, counter0_out, counter1_out and counter2_out as inputs. Clk is the input clock of the module, rst is reset input of the module, mem_w is an indication whether we want to write to memory or not, Cpu_data2bus is the data received from CPU, addr_bus is the address of the memory, ram_data_out is the data we get to from the ram, led_out is the LEDs we want to turn on, counter_out, counter1_out, counter0_out and counter2_out are the inputs from the counter module. Then we have Cpu_data4bus, ram_data_in, ram_addr, data_ram_we, GPIOf000000_we, GPIOe000000_we, counter_we and peripheral_in as outputs. Cpu_data4bus is our output data to CPU, ram_data_in is the data we want to write to ram, ram_addr is the address of the memory we want to access/write to, data_ram_we is whether we want to write to ram or not, GPIOf000000_we is the enabler switch for multiplexer and GPIOe000000_we is the enabler switch for GPIO, counter_we is a mode switch for the counter module, and peripheral_in is our mode 0 for the displays and our LED controller.



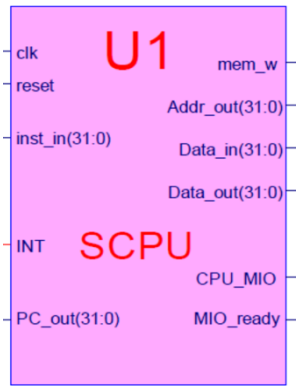
Next, we have our counter module. This module is in charge of counting. There are different modes which are activated by the input signals. The input signals are clk, rst, clk0, counter_we, counter_val, counter_ch. Clk is the clock of our module, rst is the reseter of the module, clk0 is used for the counting, counter_we is our input from MIO_BUS which is one of our mode selectors(or counter write mode?), counter_val is the value of counter which is our input from MIO_BUS which is the CPU's output to the I/O and also our first mode on the multiplexer and counter_ch is our channel counter controller. Our outputs are counter0_out, counter1_out, counter2_out and counter_out. As the name suggests our counter_out is the output of our counter and counter0_out, counter1_out, counter2_out are channel 0, channel 1 and channel 2 counter outputs respectfully.




```
module Counter(input clk,
           input rst,
           input clk0,
           input clk1,
           input clk2,
           input counter_we,
           input [31:0] counter_val,
           input [1:0] counter_ch,       //Counter channel set

           output counter0_OUT,
           output counter1_OUT,
           output counter2_OUT,
           output [31:0] counter_out

           );
reg [32:0] counter0,counter1,counter2;
reg [31:0] counter0_Lock,counter1_Lock,counter2_Lock;
reg [23:0] counter_Ctrl;
reg sq0,sq1,sq2,M0,M1,M2,clr0,clr1,clr2;

//Counter read or write & set   counter_ch=SC1 SC0; counter_Ctrl=XX M2 M1 M0 X

  always @ (posedge clk or posedge rst) begin
    if (rst )
      begin counter0_Lock <=0;counter1_Lock <=0;counter2_Lock <=0;counter_Ctrl<=0; end
    else
      if (counter_we) begin
        case(counter_ch)
          2'h0: begin counter0_Lock <= counter_val;  M0<=1; end //f0000000: bit1 bit0 =00
          2'h1: begin counter1_Lock <= counter_val;  M1<=1; end //f0000000: bit1 bit0 =01
          2'h2: begin counter2_Lock <= counter_val;  M2<=1; end //f0000000: bit1 bit0 =10
          2'h3: begin counter_Ctrl <= counter_val[23:0];    end //counter_Ctrl=XX M2 M1 M0 X
        endcase
```

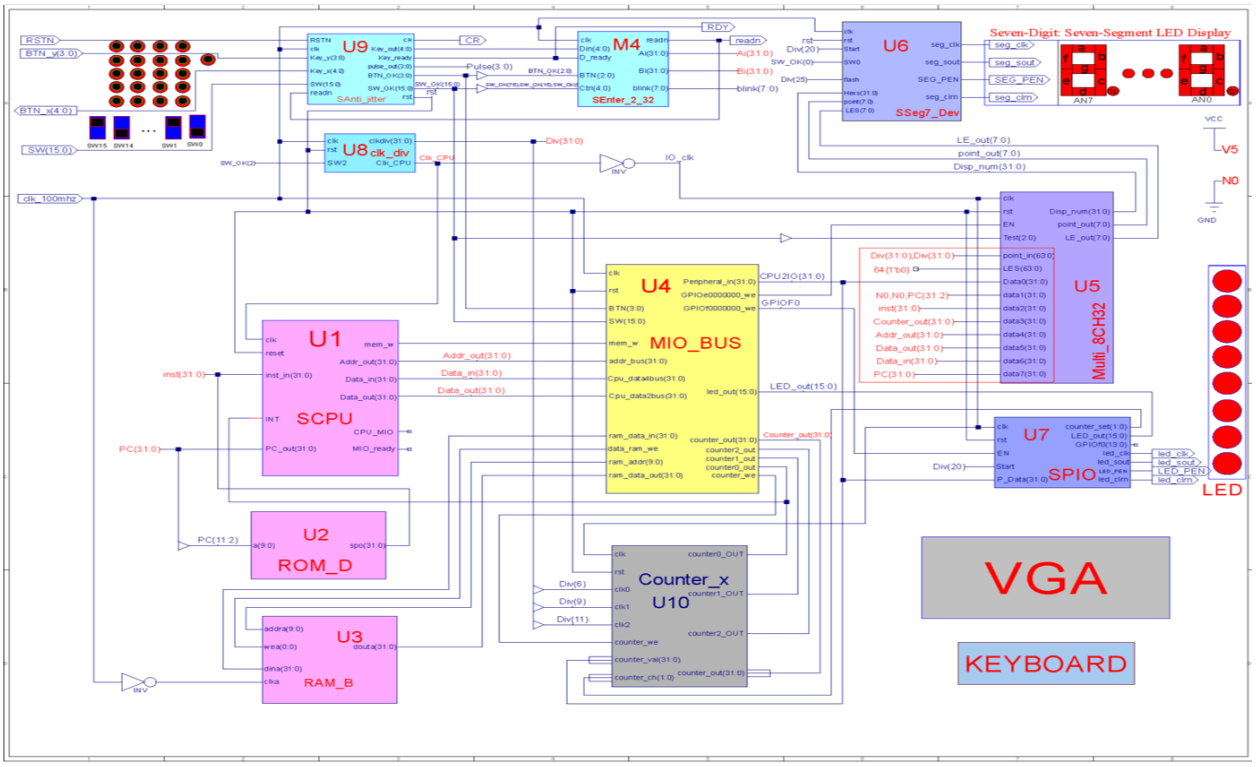
```
        end
      else begin counter0_Lock <=counter0_Lock;
             counter1_Lock <=counter1_Lock;
             counter2_Lock <=counter2_Lock;
             counter_Ctrl<=counter_Ctrl;
             if(clr0) M0<=0;
             if(clr1) M1<=0;
             if(clr2) M2<=0;
           end
  end

// Counter channel 0
  always @ (posedge clk0 or posedge rst) begin
    if (rst )
      begin counter0<=0; sq0<=0; end
    else
      case(counter_Ctrl[2:1])
        2'b00: begin if (M0) begin counter0 <= {1'b0,counter0_Lock}; clr0<=1; end
                     else if  (counter0[32]==0)begin counter0 <= counter0 - 1'b1; clr0<=0; end
               end
        2'b01: begin if (counter0[32]==0) counter0 <= counter0 - 1'b1;
                     else counter0 <={1'b0,counter0_Lock}; end
        2'b10: begin sq0<=counter0[32];
               if (sq0!=counter0[32]) counter0[31:0] <= {1'b0,counter0_Lock[31:1]};
               else counter0 <= counter0 - 1'b1;end
        2'b11: counter0 <= counter0 - 1'b1;
      endcase
  end

  assign counter0_OUT=counter0[32];
  assign counter1_OUT=counter1[32];
  assign counter2_OUT=counter2[32];
  assign counter_out = counter0[31:0];
endmodule
```

Finally, we have SCPU. This module is MIPS based which supports three types of instructions: R-Type, I-Type and J-Type. It's the brain of this experiment. It has 6 inputs, clk, reset, inst_in, Data_in and INT. Clk is the input clock, reset is used to reset the module, inst_in is the instruction set, Data_in is the input data and INT is the interrupt input. As a result, we have the following outputs, PC_out, Addr_out, Data_out, and mem_w. PC_out is program space access pointer which part of it goes to the ROM, Addr_out is the ram address, Data_out is the data to be written to the memory, mem_w is whether we want to write to ram or not.



3 – Implementation and results

The top module should look like this:



As discussed in chapter 1, this experiment shares a lot of similar components and connections to the previous lab. There are few new components in this experiment which were introduced and explained in chapter 2. Other components that exist in the previous serve the same purposes and used in a similar way. Similar to our previous lab we have a main multiplexer called Multi_8CH32, which routes the inputs to our display module. The inputs though, differ compared to the previous experiment. Here we have CPU2IO as our first input which is the peripheral output from the MIO_BUS or CPU's output to the I/O, then we have PC as our second input which is the program space access pointer that comes from SCPU, next one is ROM's output which is also the instructions, then we have the address of the memory which is coming from the SCPU, and Data_out, Data_in which are the output data and input data from SCPU. The outputs are stayed untouched. Here is the implementation of the multiplexer in the top module:

```
Multi_8CH32 U5(.clk(~CLK_CPU), .rst(rst), .EN(EN), .Test(SW_OK[7:5]), .point_in({Div, Div}), .LES(64'b0),
               .Data0(CPU2IO), .data1({2'b0, PC[31:2]}), .data2(inst), .data3(counter_out), .data4(Addr_out),
               .data5(Data_out), .data6(Data_in), .data7(PC), .Disp_num(Disp_num), .LE_out(LE_out), .point_out(point_out));
```

Another difference between this experiment and the previous is the input of the SPIO (GPIO) which in this case is replaced by the output of the MIO_BUS (peripheral_in) instead of switches. And as mentioned in chapter 2 the EN switch is now coming from GPIOf00000000_we. This signal is produced by MIO_BUS, similarly GPIOe00000000_we is used for the enabler switch of Multiplexer (Multi_8CH32). The other inputs and outputs are stayed intact. Here is the implementation of this module in the top module:

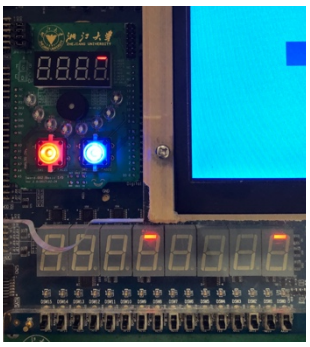
```
GPIO U7(.clk(~CLK_CPU), .rst(rst), .EN(GPIOF), .Start(Div[20]), .P_Data(CPU2IO), .counter_set(counter_ch), .LED_out(LED_out),
        .ledclk(led_clk), .ledsout(led_sout), .LEDEN(LED_PEN), .ledclrn(led_clrn));
```

Once these changes are made, we can implement the top module by adding the new components and slightly modify the existing top module. Here is the final result of the top module:

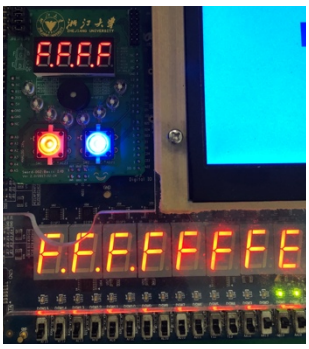
```
module top(input RSTN,
           input [3:0]BTN_y,
          output [4:0] BTN_x,
          input [15:0] SW,
          input clk_100mhz,
          output led_clk,
          output led_sout,
          output LED_PEN,
          output led_clrn,
          output seg_clk,
          output seg_sout,
          output SEG_PEN,
          output seg_clrn,
          output CR,
          output RDY,
          output readn,
          output[7:0]SEGMENT,
          output [3:0]AN);

  wire [3:0] Pulse;
  wire [3:0] BTN_OK;
  wire [15:0] SW_OK;
  wire [4:0] Din;
  wire rst;
  SAnti_jitter U9(.clk(clk_100mhz), .RSTN(RSTN), .Key_y(BTN_y), .Key_x(BTN_x),
             .SW(SW), .readn(readn), .Key_ready(RDY), .CR(CR), .Key_out(Din), .pulse_out(Pulse),
             .BTN_OK(BTN_OK), .SW_OK(SW_OK), .rst(rst));

  wire [31:0] Ai;
  wire [31:0] Bi;
  wire [7:0] blink;

  SEnter_2_32 M4(.clk(clk_100mhz), .Din(Din), .D_ready(RDY), .BTN(BTN_OK[2:0]), .Ctrl({SW_OK[7:5], SW_OK[15], SW_OK[0]})
            ,.readn(readn), .Ai(Ai), .Bi(Bi), .blink(blink));


  wire [31:0] Div;
  clk_div U8(.clk(clk_100mhz), .rst(rst), .SW2(SW_OK[2]), .clkdiv(Div), .Clk_CPU(CLK_CPU));



  wire [31:0] inst, ram_data_in, ram_data_out;
  wire [9:0] ram_addr;
  wire data_ram_we;

  RAM_B U3 (.clka(~clk_100mhz), .addra(ram_addr), .wea(data_ram_we), .dina(ram_data_in), .douta(ram_data_out));
  ROM_B U2 (.a(PC[11:2]), .spo(inst));

  wire [31:0] counter_out, CPU2IO;
  wire [1:0] counter_ch;


  Counter U10(.clk(~CLK_CPU), .rst(rst), .clk0(Div[6]), .clk1(Div[9]), .clk2(Div[11]),
          .counter_we(counter_we), .counter_val(CPU2IO), .counter_ch(counter_ch),
          .counter0_OUT(counter0_out), .counter1_OUT(counter1_out), .counter2_OUT(counter2_out),
          .counter_out(counter_out));
```

```
wire [31:0] PC, Addr_out, Data_in, Data_out;

SCPU U1 (.clk(CLK_CPU), .reset(rst), .inst_in(inst), .INT(counter0_out), .PC_out(PC), .MR(mem_w),
      .Addr_out(Addr_out), .Data_out(Data_out), .Data_in(Data_in));


wire [15:0] LED_out;

MIO_BUS U4(.clk(clk_100mhz), .rst(rst), .BTN(BTN_OK), .SW(SW_OK), .mem_w(mem_w), .addr_bus(Addr_out),
        .Cpu_data4bus(Data_in), .Cpu_data2bus(Data_out), .ram_data_in(ram_data_in), .data_ram_we(data_ram_we),
        .ram_addr(ram_addr),.ram_data_out(ram_data_out), .Peripheral_in(CPU2IO), .GPIOe0000000_we(EN),
        .GPIOf0000000_we(GPIOF),.led_out(LED_out), .counter_out(counter_out), .counter0_out(counter0_out),
        .counter1_out(counter1_out),.counter2_out(counter2_out), .counter_we(counter_we));

wire [7:0] LE_out, point_out;
wire [31:0] Disp_num;


Multi_8CH32 U5(.clk(~CLK_CPU), .rst(rst), .EN(EN), .Test(SW_OK[7:5]), .point_in({Div, Div}), .LES(64'b0),
               .Data0(CPU2IO), .data1({2'b0, PC[31:2]}), .data2(inst), .data3(counter_out), .data4(Addr_out),
               .data5(Data_out), .data6(Data_in), .data7(PC), .Disp_num(Disp_num), .LE_out(LE_out), .point_out(point_out));


GPIO U7(.clk(~CLK_CPU), .rst(rst), .EN(GPIOF), .Start(Div[20]), .P_Data(CPU2IO), .counter_set(counter_ch), .LED_out(LED_out),
        .ledclk(led_clk), .ledsout(led_sout), .LEDEN(LED_PEN), .ledclrn(led_clrn));


Display U6(.clk(clk_100mhz), .rst(rst), .Start(Div[20]), .Text(SW_OK[0]), .flash(Div[25]), .Hexs(Disp_num), .point(point_out),
           .LES(LE_out), .segclk(seg_clk), .segsout(seg_sout), .SEGEN(SEG_PEN), .segclrn(seg_clrn));

Seg7_Dev U61(.Scan({SW_OK[1], Div[19:18]}), .SW0(SW_OK[0]), .flash(Div[25]),
           .Hexs(Disp_num), .point(point_out), .LES(LE_out), .SEGMENT(SEGMENT), .AN(AN));
endmodule
```

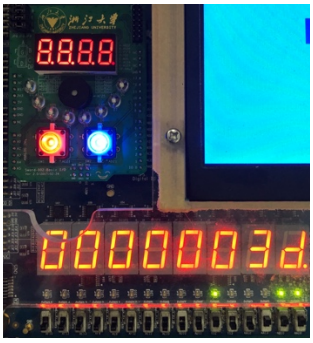
Results:



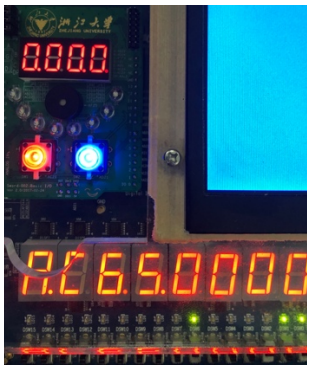
Mode 0: peripherals (~SW0)



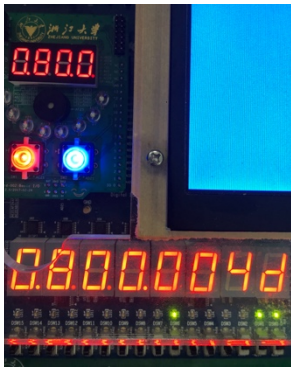
Mode 0: peripherals (SW0)



Mode 1: PC(SW0)

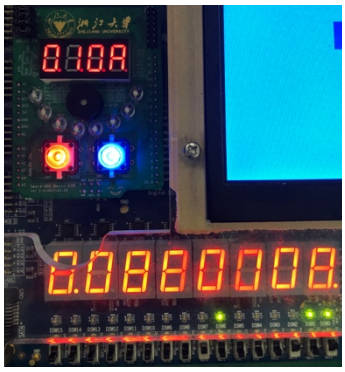


Mode 2: instructions

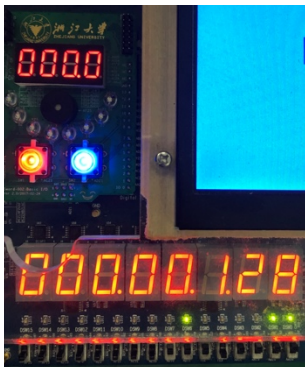


Mode 3: Counter

Mode 4: Address



Mode 5: Data out



Mode 7: PC

There are more pictures in the images folder.

4 – Discussion

For debugging purposes, I used the Seg7_Dev to see if what's being displayed on the 8 segments matches the 4 segments one, and sure it did but the problem with that clock didn't match so there was a blinking effect, and when SW[2] was turned off the display would be a bit unreadable and sometimes didn't match the 8 segments one due to the clock syncing problem.

The second problem I encountered was that whenever SW[2] was turned on to slow down the counters, it state of the LEDs wouldn't change until it was turned off.

The programming files and top files are all available in the Lab2 Content folder.