上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

数字部件实验二

基于 Verilog 和 FPGA 的基本单周期 CPU 设计

姓 名：　　王鑫伟

学 院：　　电子信息与电气工程学院

学 号：　　516030910041

班 级：　　F1603701

2018 年 12 月 6 日

一、实验目的

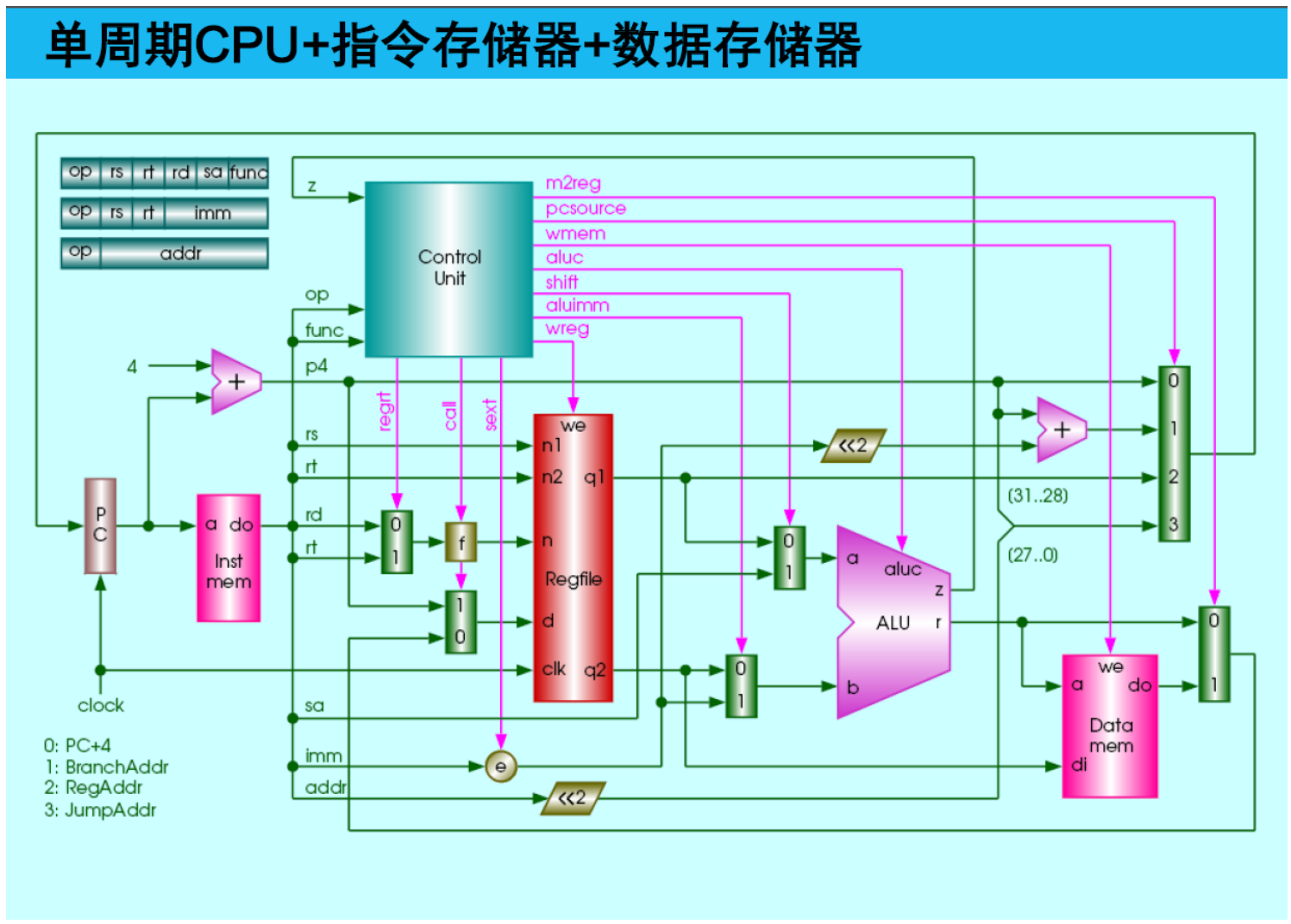
1. 理解计算机 5 大组成部分的协调工作原理，理解存储程序自动执行的原理。
2. 掌握运算器、存储器、控制器的设计和实现原理。重点掌握控制器设计原理和实现方法。
3. 掌握 I/O 端口的设计方法，理解 I/O 地址空间的设计方法。
4. 会通过设计 I/O 端口与外部设备进行信息交互。

二、实验内容

1. 采用 Verilog HDL 在 quartus II 中实现基本的具有 20 条 MIPS 指令的单周期 CPU 设计。
2. 利用实验提供的标准测试程序代码，完成仿真测试。
3. 采用 I/O 统一编址方式，即将输入输出的 I/O 地址空间，作为数据存取空间的一部分，实现 CPU 与外部设备的输入输出端口设计。实验中可采用高端地址。
4. 利用设计的 I/O 端口，通过 lw 指令，输入 DE2 实验板上的按键等输入设备信息。即将外部设备状态，读到 CPU 内部寄存器。
5. 利用设计的 I/O 端口，通过 sw 指令，输出对 DE2 实验板上的 LED 灯等输出设备的控制信号(或数据信息)。即将对外部设备的控制数据，从 CPU 内部的寄存器，写入到外部设备的相应控制寄存器 (或可直接连接至外部设备的控制输入信号)。
6. 利用自己编写的程序代码，在自己设计的 CPU 上，实现对板载输入开关或按键的状态输入，并将判别或处理结果，利用板载 LED 灯或 7 段 LED 数码管显示出来。
7. 例如，将一路 4bit 二进制输入与另一路 4bit 二进制输入相加，利用两组分别 2 个 LED 数码管以 10 进制形式显示“被加数”和“加数”，另外一组 LED 数码管以 10 进制形式显示“和”等。(具体任务形式不做严格规定，同学可自由创意）。
8. 在实现 MIPS 基本 20 条指令的基础上，实现 Y86 相应的基本指令。
9. 在实验报告中，汇报自己的设计思想和方法；并以汇编语言的形式，提供以上两种指令集(MIPS 和 Y86) 的应用功能的程序设计代码，并提供程序主要流程图。

三、实验设计

3.1 复习单周期 CPU 原理图



[图一] 单周期 CPU 逻辑示意图

3.2 通过真值表，补全 alu.v 和 sc_cu.v 文件

由于比较简单，后面也没怎么用到这两个文件，就不多赘述

```
module alu (a,b,aluc,s,z);
   input [31:0] a,b;
   input [3:0] aluc;
   output [31:0] s;
   output        z;
   reg [31:0] s;
   reg        z;
   always @ (a or b or aluc)
      begin                                  // event
         casex (aluc)
             4'bx000: s <= a + b;            //x000 ADD
             4'bx100: s <= a - b;            //x100 SUB
             4'bx001: s <= a & b;            //x001 AND
             4'bx101: s <= a | b;            //x101 OR
             4'bx010: s <= a ^ b;            //x010 XOR
             4'bx110: s <= (b) << 32'd16;    //x110 LUI: imm << 16bit
             4'b0011: s <= (b) << (a);       //0011 SLL: rd <- (rt << sa)
             4'b0111: s <= (b) >> (a);       //0111 SRL: rd <- (rt >> sa) (logical)
             4'b1111: s <= $signed(b) >>> a; //1111 SRA: rd <- (rt >> sa) (arithmetic)
             default: s <= 0;
         endcase
         if (s == 0 )  z = 1;
            else z = 0;
      end
endmodule
```

［图二］alu.v 补全代码

```
    //  please complete the deleted code.

    wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
                 func[2] & ~func[1] & ~func[0];          //100100
    wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
                 func[2] & ~func[1] & func[0];           //100101

    wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
                 func[2] & func[1] & ~func[0];           //100110
    wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
                 ~func[2] & ~func[1] & ~func[0];         //000000
    wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
                 ~func[2] & func[1] & ~func[0];          //000010
    wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
                 ~func[2] & func[1] & func[0];           //000011
    wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
                 ~func[2] & ~func[1] & ~func[0];         //001000

                      // complete by yourself.
    wire i_ori  = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] &  op[0]; //001101
    wire i_xori = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] & ~op[0]; //001110
    wire i_lw   =  op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //100011
    wire i_sw   =  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0]; //101011
    wire i_beq  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] & ~op[0]; //000100
    wire i_bne  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] &  op[0]; //000101
    wire i_lui  = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] &  op[0]; //001111
    wire i_j    = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] & ~op[0]; //000010
    wire i_jal  = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //000011

                      // complete by yourself.
    assign aluc[3] = i_sra;
    assign aluc[2] = i_sub | i_or | i_srl | i_sra | i_ori | i_lui;
    assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq | i_bne | i_lui;
    assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
    assign shift   = i_sll | i_srl | i_sra ;

                      // complete by yourself.
    assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui | i_sw;
    assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne;
    assign wmem    = i_sw;
    assign m2reg   = i_lw;
    assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui;
    assign jal     = i_jal;
```

［图三］sc_cu.v 补全代码

3.3 修改顶层设计 sc_computer.v 文件

按照我的想法，我希望这次的输出仍然用 7 段数码管，3 个都要用上，所以在 sc_computer .v 中添加了 output 端口 out_port2，相应的 sc_datamem.v 也做适应性修改

```
    sc_cpu cpu (clock,resetn,inst,memout,pc,wmem,aluout,data);          // CPU module.
    sc_instmem  imem (pc,inst,clock,mem_clk,imem_clk);                  // instruction memory.
    sc_datamem  dmem (aluout,data,memout,wmem,clock,mem_clk,dmem_clk, clrn,
          out_port0, out_port1, out_port2, in_port0, in_port1, mem_dataout, io_read_data); // data memory.
```

［图四］sc_computer.v 函数调用参数修改

3.4 设计 IO 输入输出接口文件

由于实验 2.3 中提供的 io_input_reg 和 io_output_reg 不太会用，就写了跟 DE1-SoC 板子对接的两个模块，从 sw 接收输入，从数字管输出。秒表实验的 sevenseg 就直接用了

```v
module io_to_in_port (sw4,sw3,sw2,sw1,sw0,in_port);
   input sw0,sw1,sw2,sw3,sw4;
   output [31:0] in_port;

   assign in_port[0] = sw0;
   assign in_port[1] = sw1;
   assign in_port[2] = sw2;
   assign in_port[3] = sw3;
   assign in_port[4] = sw4;

endmodule
```

```v
module out_seg(in,out1,out0);
   input [31:0] in;
   output [6:0] out1,out0;

   reg [3:0] num1,num0;

   sevenseg display_1( num1, out1 );
   sevenseg display_0( num0, out0 );

   always @ (in)
   begin
      num1 = ( in / 10 ) % 10;    //十位
      num0 = in % 10;             //个位
   end
endmodule
```

```v
module sevenseg ( data, ledsegments);
   input [3:0] data;

   output ledsegments;
   reg [6:0] ledsegments;

   always @ (*)
      case(data)
            //gfe_dcba  // 7 段 LED数码管的位段编号
            //654_3210  // DE1-SOC板上的信号位编号
         0: ledsegments = 7'b100_0000;
         1: ledsegments = 7'b111_1001;
         2: ledsegments = 7'b010_0100;
         3: ledsegments = 7'b011_0000;
         4: ledsegments = 7'b001_1001;
         5: ledsegments = 7'b001_0010;
         6: ledsegments = 7'b000_0010;
         7: ledsegments = 7'b111_1000;
         8: ledsegments = 7'b000_0000;
         9: ledsegments = 7'b001_0000;
        default: ledsegments = 7'b111_1111; //其他值时全灭。
      endcase
endmodule
```

［图五］输入/输出接口辅助代码

3.5　设计时钟输入 clock.v

实验书上解释的挺清楚的，这个实验需要两个时钟信号 clk 和 mem_clk，通过简单的逻辑~运算实现

【问题 2】 单周期 CPU 设计实验中除了一个基本的 clock 时钟信号外，为什么还设计有一个 2 倍频于 clock 信号、标记为 mem_clk 的时钟信号?

【解答】这是每个指令周期内，对各逻辑功能部件的工作时间顺序进行控制的需要。

从单周期 CPU 的设计原理上，如果指令 ROM 和数据 RAM 都为异步器件，即只要收到地址就开始送出数据，则在实现中只需要一个 clock 时钟信号控制 PC 的变化、以及整条指令的执行时间（机器周期）就够了。

但在 DE1-SOC 实验板上 Altera 的 Cyclone Ⅴ系列 FPGA 器件中，只提供同步 ROM 和同步 RAM 宏模块。在【问题 1】的解答中所建立的同步 ROM 和同步 RAM，其读写时序如图 1-13 和图 1-14 所示。

考察图 2-1-13 所示同步读时序，当同步 ROM 或 RAM 接收到数据地址时，不同于异步器件在经过一段组合电路的反应延迟后送出数据，同步器件需要等到同步时钟信号 clock 上升沿时才真正送出数据（这就可以使延迟时间不尽相同的多个器件在输出级达成一致）。

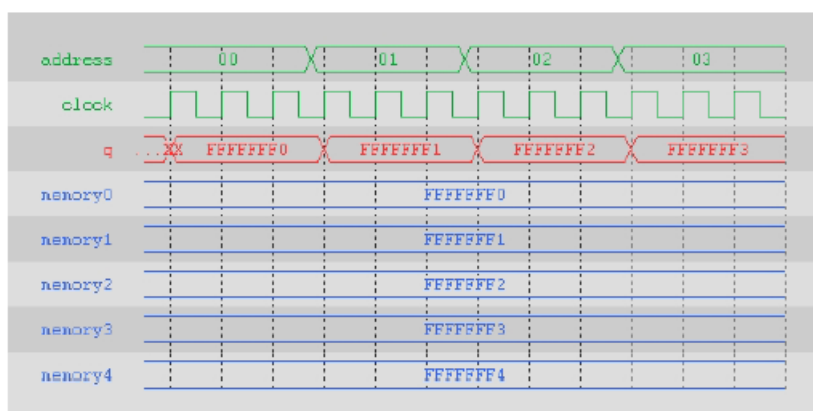


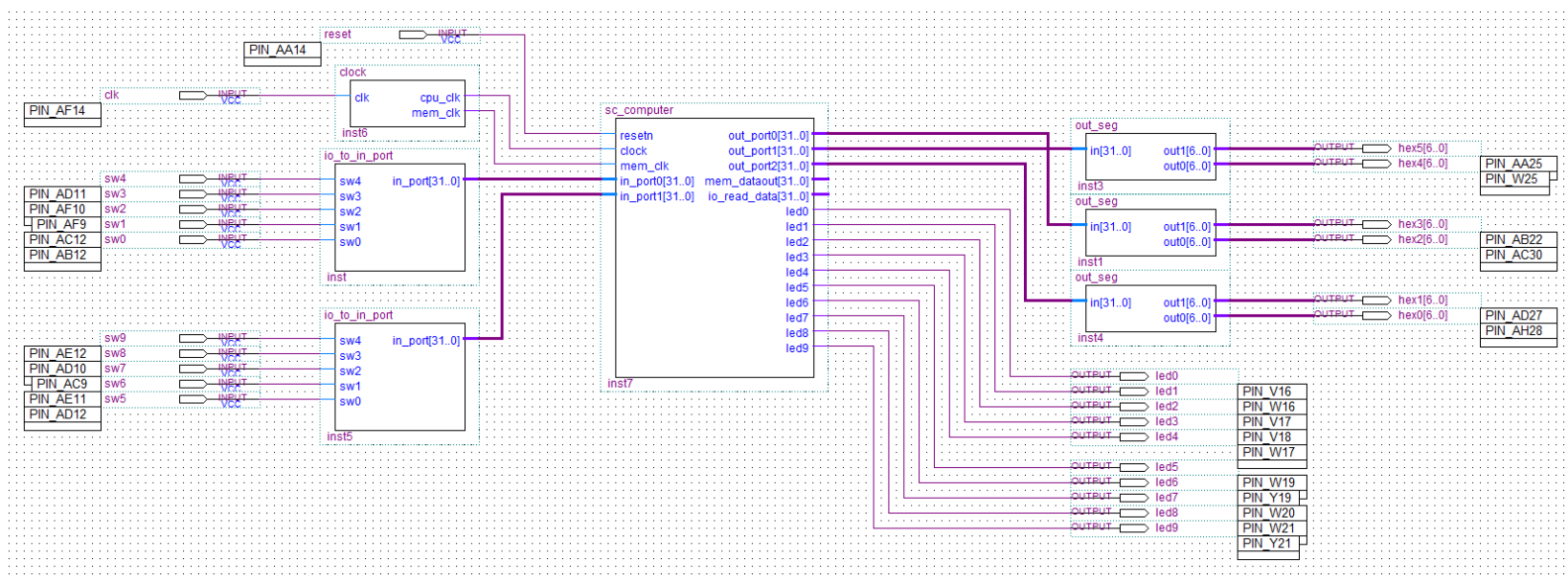
图 2-1-13. 同步 ROM 和同步 RAM 的“读”操作工作时序

［图六］实验指导书关于两个时钟的解释

```
1   module clock(clk, cpu_clk, mem_clk);
2
3      input       clk;
4      output      cpu_clk, mem_clk;
5      reg         cpu_clk;
6
7      assign      mem_clk = clk;
8      initial
9         begin
10           cpu_clk <= 0;
11        end
12
13     always @(posedge clk)
14        begin
15           cpu_clk <= ~cpu_clk;
16        end
17
18  endmodule
```

［图七］clock.v 代码

3.6 绘制 sc_computer.bdf 顶层模块接线图，在 pin planner 中完成 pin 脚分配



［图八］sc_computer.bdf 接线图

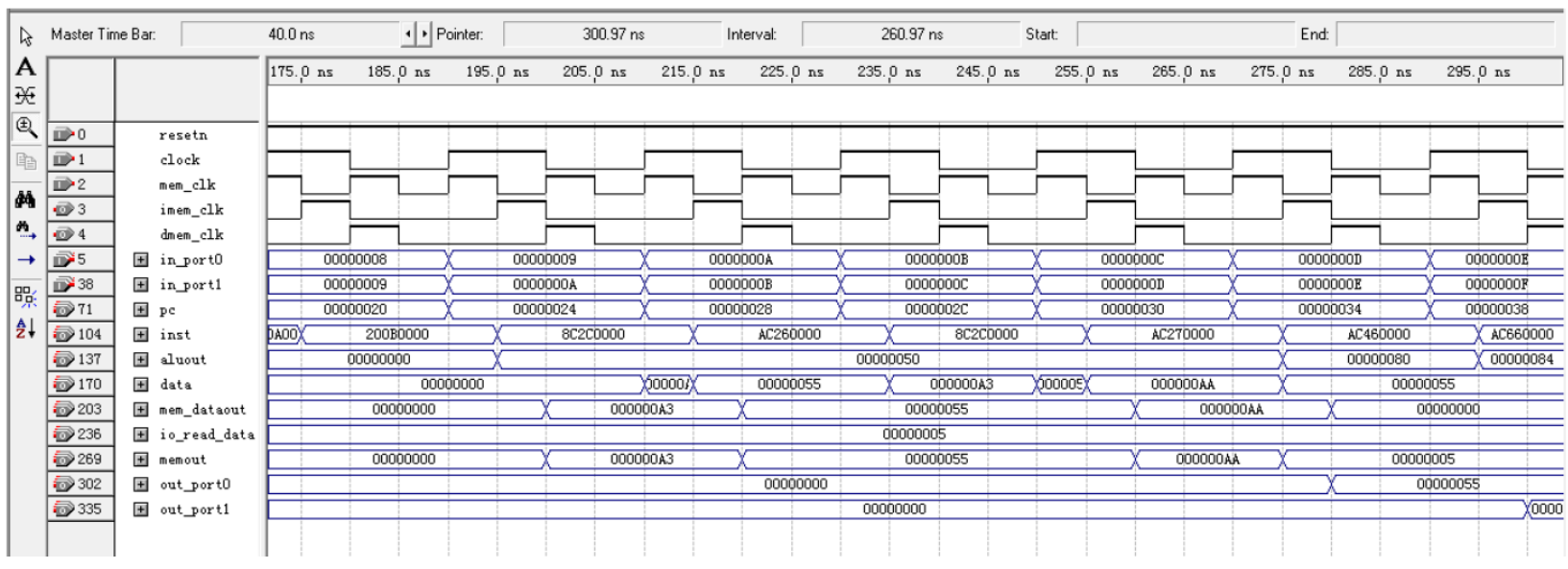
3.7 Modelsim 仿真

为了能够验证 modelsim 仿真结果的正确性，用了实验 2.3 中给的 mif 文件，并把自己的仿真结果和实验书中给出的结果对比，截取的是 pc 从 0x24 到 0x34 的部分，完全一致

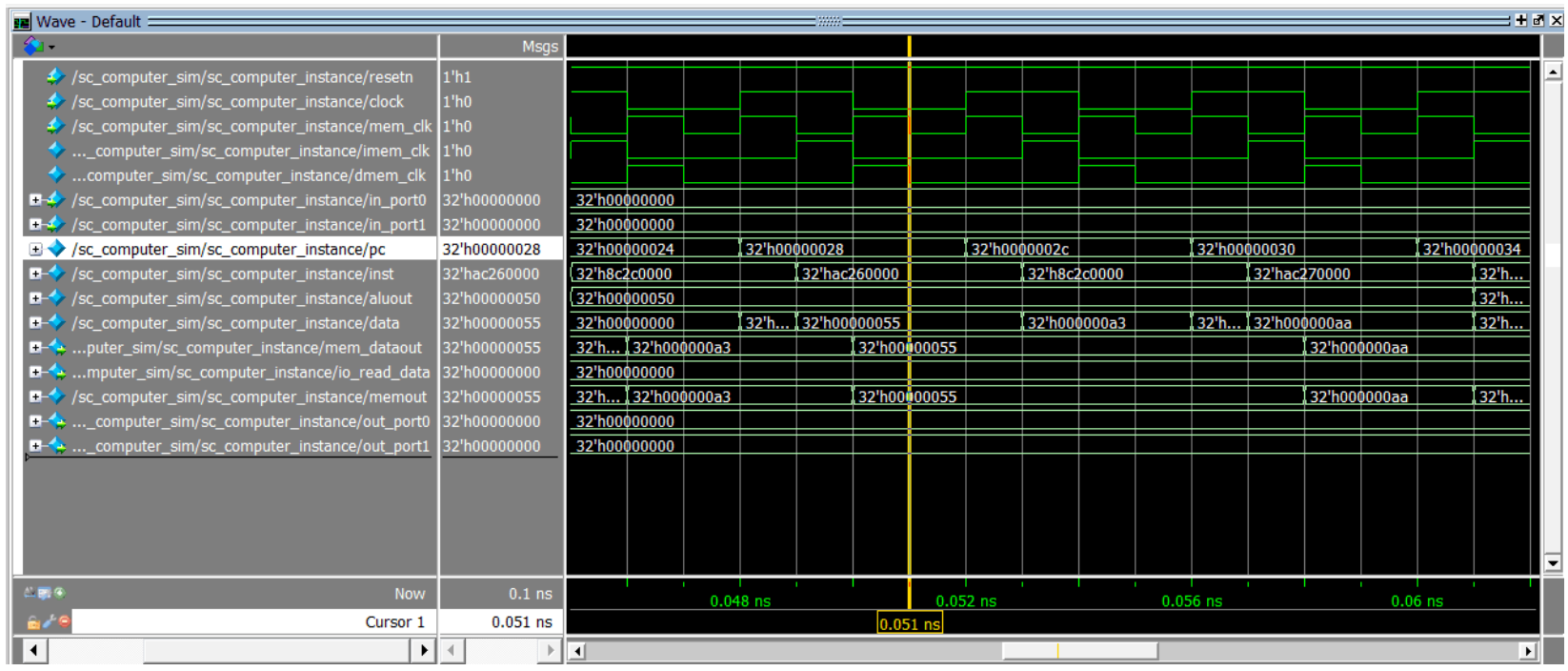
（5）针对 I/O 端口的一段测试例程代码：（sc_instmem_02.mif）

```
DEPTH = 64; % Memory depth and width are required %
WIDTH = 32; % Enter a decimal number %
ADDRESS_RADIX = HEX; % Address and value radixes are optional %
DATA_RADIX = HEX; % Enter BIN, DEC, HEX, or OCT; unless %
% otherwise specified, radixes = HEX %
CONTENT
BEGIN

 0 : 20010050; % (00) main:   addi $1, $0, 01010000b     # address 50h          %
 1 : 20020080; % (04)          addi $2, $0, 10000000b     # address 80h          %
 2 : 20030084; % (08)          addi $3, $0, 10000100b     # address 84h          %
 3 : 200400c0; % (0c)          addi $4, $0, 11000000b     # address c0h          %
 4 : 200500c4; % (10)          addi $5, $0, 11000100b     # address c4h          %
 5 : 20060055; % (14)          addi $6, $0, 01010101b     # data       55h       %
 6 : 200700aa; % (18)          addi $7, $0, 10101010b     # data       aah       %
 7 : 200a0000; % (1c) loop:   addi $10,$0, 0              # r10 = 0              %
 8 : 200b0000; % (20)          addi $11,$0, 0             # r11 =
```



［图九］Modelsim 仿真所用代码（部分）及期望结果



[图十] Modelsim 仿真实际结果

3.8 编写测试代码并用 MIPS_Compiler 中的 exe 编码

测试代码的意思就是从两个输入端口（5 根开关代表一个输入数）读取两个数（10 进制，最大 31），并将加法计算过程显示到数字管上。生成的两个 mif 文件直接替换掉原先的

sc_instmem.mif

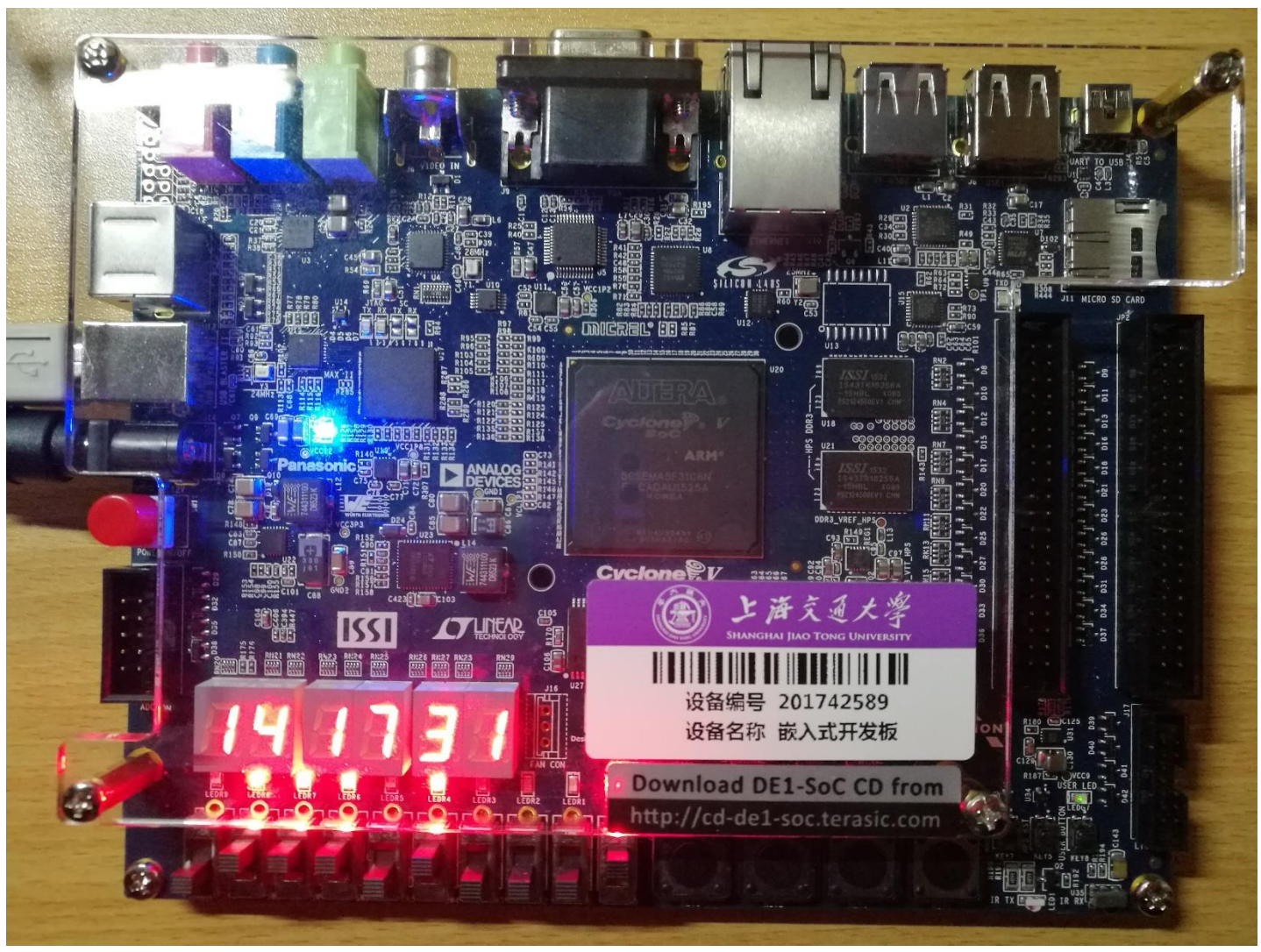
```
DEPTH = 16;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %

1 : 3c000000;        % (04)       lui $0, 0          #  %
2 : 20010080;        % (08)       addi  $1, $0, 0x80 #  %
3 : 20020084;        % (0c)       addi  $2, $0, 0x84 #  %
4 : 20030088;        % (10)       addi $3, $0, 0x88  #  %
5 : 200400c0;        % (14)       addi  $4, $0, 0xc0 #  %
6 : 200500c4;        % (18)       addi $5, $0, 0xc4  #  %
8 : 8c860000;        % (20)       lw $6, 0($4)       #  %
9 : 8ca70000;        % (24)       lw  $7, 0($5)      #  %
A : 00c74020;        % (28)       add $8, $6, $7     #  %
B : ac260000;        % (2c)       sw $6, 0($1)       #  %
C : ac470000;        % (30)       sw  $7, 0($2)      #  %
D : ac680000;        % (34)       sw $8, 0($3)       #  %
E : 08000007;        % (38)       j  body            #  %
END ;
```

[图十一] 经过编译生成的 sc_instmem.mif 测试代码

3.9 将程序烧写到 DE1-SoC FPGA 实验板

前两个数是加数和被加数，第三个数是结果。扳动的开关 LED 灯会亮



[图十二] DE1-SoC 实验板结果示意图

四、实验中遇到的问题

4.1. 编译错误

群文件提供的部分文件是过不了编译的，比较典型的就是 cla32.v 文件，由于在本次实验中并没有用到这个文件，我直接将其注释掉了；另一个是在 regfile.v 中，报了“不能在匿名块声明变量”的错，应该吧 integer i 移到外面去。还有一些自己漏分号之类的小问题。

```
1   module cla32 (pc,x1,x2,p4)
2      input [31:0] pc
3      input [31:0]
4
5      output [31:0] p4
6
7      assign p4 = pc + 32'b4;
8      assign adr = p4 + offset;
9
10  endmodule
```

```
12021 Found 0 design units, including 0 entities, in source file sc_datamem_2.v
10170 Verilog HDL syntax error at cla32.v(2) near text "input";  expecting ";"
12021 Found 0 design units, including 0 entities, in source file cla32.v
12021 Found 1 design units, including 1 entities, in source file sevenseg.v
12021 Found 1 design units, including 1 entities, in source file out_seg.v
12021 Found 1 design units, including 1 entities, in source file io_to_in_port.v
12021 Found 1 design units, including 1 entities, in source file sc_instmem.v
12021 Found 1 design units, including 1 entities, in source file sc_datamem.v
```

```
14     integer i;
15
16     always @(posedge clk or negedge clrn) begin
17     if (clrn == 0) begin
18           //integer i;
19           for (i=1; i<32; i=i+1)
20              register[i] <= 0;
21        end else begin
22           if ((wn != 0) && (we == 1))          // write
23              register[wn] <= d;
24        end
25     end
```

［图十三］cla32.v 和 regfile.v 的编译错误

4.2. pin 脚过度使用导致不够

在最初给的文件中，sc_computer.v 的 input 和 output 端口很多，导致了 pin 脚不够的问题（大约用掉了 320+的 pin 脚，而 DE1-SoC 没有那么多端口），查阅实验指导书发现许多标黄的 port 是可以删除的，这个问题就解决了。最后用了 200 个 pin 脚左右

```
10  module sc_computer (resetn,clock,mem_clk, mem_dataout, io_read_data,
11        out_port0, out_port1, out_port2, in_port0, in_port1,
12        led0, led1, led2, led3, led4, led5, led6, led7, led8, led9);
13
14    /*input          clrn;
15      output [31:0]  pc,inst,aluout,memout;
16      output         imem_clk,dmem_clk; */
17
18      input          resetn,clock,mem_clk;
19      input  [31:0]  in_port0, in_port1;
20      output [31:0]  out_port0, out_port1, out_port2;
21      output [31:0]  mem_dataout;
22      output [31:0]  io_read_data;
```

顶层代码：

```
module   sc_computer ( resetn, clock, mem_clk, pc, inst, aluout, memout, imem_clk, dmem_clk);
// 定义顶层模块 sc_computer，作为工程文件的顶层入口，如图 1-1 建立工程时指定。
        input   resetn,clock,mem_clk;
// 定义整个计算机 module 和外界交互的输入信号，包括复位信号 resetn、时钟信号 clock、
// 以及一个频率是 clock 两倍的 mem_clk 信号。注：resetn 是低电平（neg）有效信号。
// 这些信号都可以用作仿真验证时的输出观察信号。
        output  [31:0]  pc, inst, aluout, memout;
// 模块用于仿真输出的观察信号。缺省为 wire 型。
        output          imem_clk,dmem_clk;
        //模块用于仿真输出的观察信号，用于观察验证指令 ROM 和数据 RAM 的读写时序。
        wire    [31:0]  data;   // 模块间互联传递数据或控制信息的信号线。
        wire            wmem;   // 模块间互联传递数据或控制信息的信号线。
        sc_cpu  cpu (clock,resetn,inst,memout,pc,wmem,aluout,data);     // CPU module.
// 实例化了一个 CPU 模块，其内部又包含运算器 ALU 模块、控制器 CU 模块等。
```

注：以上 module 中黄色标记的信号，是使用老版本 quartusII 直接进行仿真时需要外引的信号（历史原因），使用 ModelSim 时就不需要了。

［图十四］sc_computer.v 中端口过多导致的 pin 脚不足问题

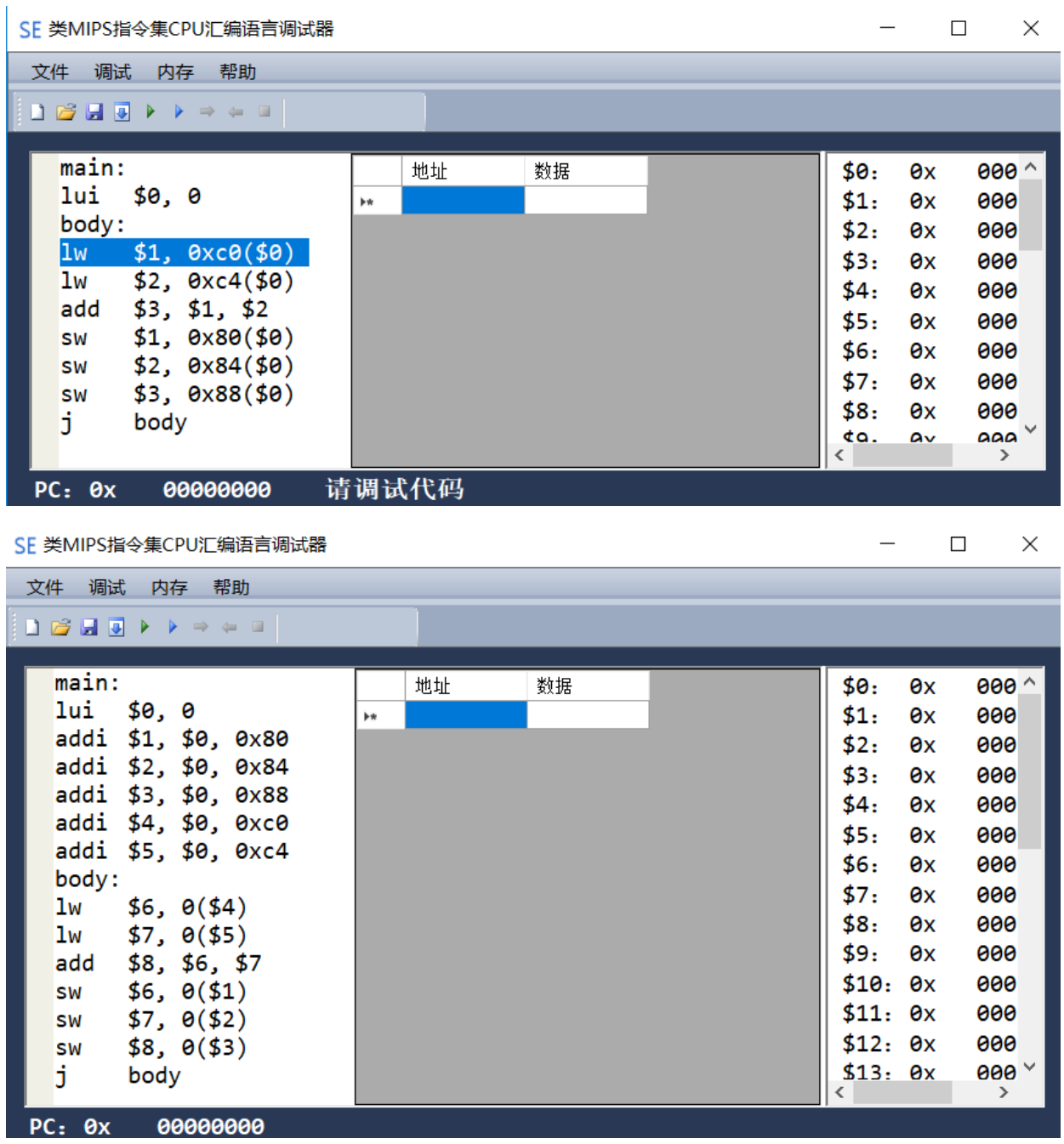
4.3. Modelsim 破解

因为下的 Modelsim 不是学生版是 SE 版本的，就找教程破解了一下。

大概就是补足了 LICENSE.TXT 文件，配置了一下环境变量。网上都有。

4.4. MIPS 汇编语法错误

在自己用 MIPS 写简单的汇编生成 instmem.mif 和 datamem.mif 测试文件的时候出过这样的问题。0xc0/0xc4 是输入端口，0x80/0x84/0x88 是输出端口，实际上现在我也不知道为什么不能这样写，而先 addi 后再 lw 就能正确。

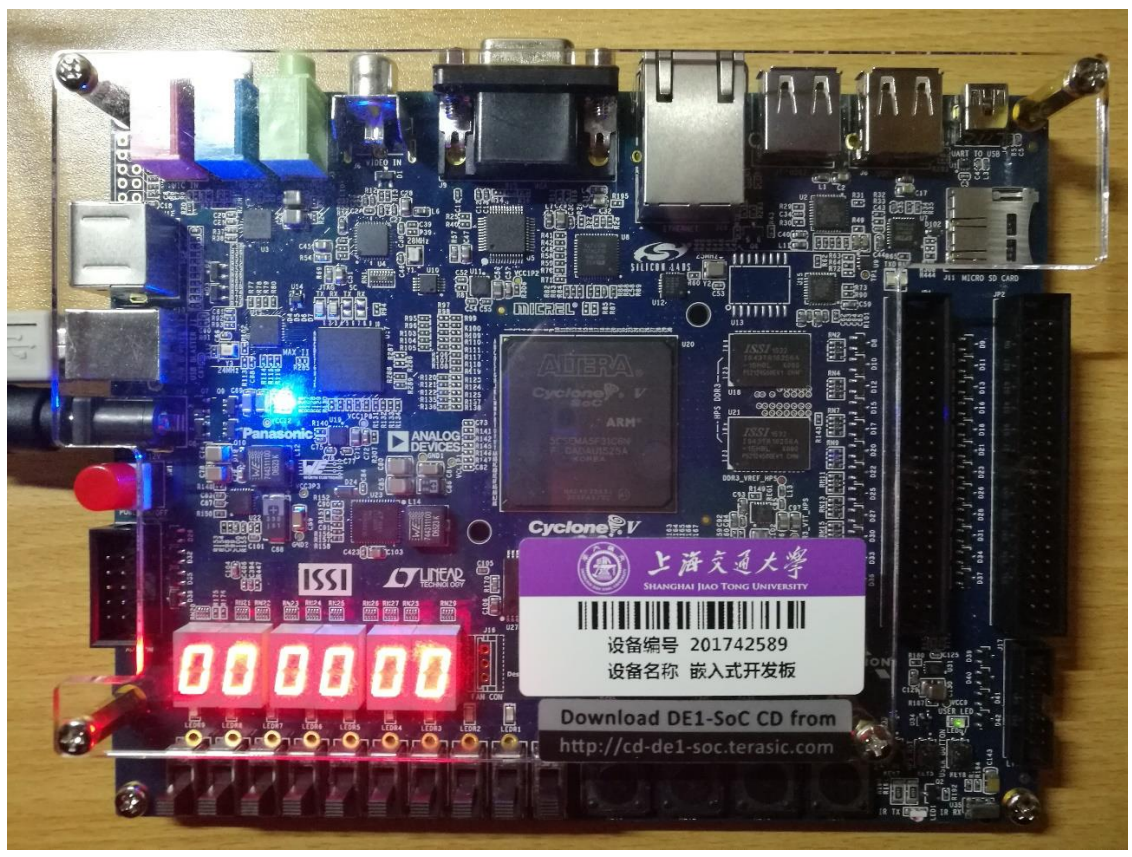


［图十五］MIPS 指令生成汇编错误示意图

4.5. 实验上扳动无反应，数字管一直显示为 0

这个错误出在我的 sc_computer.v 把 clrn 作为 input 输入，却没有实际上的输入。解决办法是将 clrn 声明为 wire，直接将 resetn 值赋给 clrn，反正也是一个时钟信号。现在实验板就可以使用了

```
34
35      assign          clrn = resetn;
36
```



［图十六］未赋值 clrn 引起实验板错误示意图

五、实验感想

这个实验实际上还是挺难的。尽管上课我都听懂了，单周期那张图的逻辑我也能不怎么费力的理顺，尽管代码量很小，算来算去就是对着真值表抄一抄，但是一开始会比较迷茫，不知道该做什么。一开始的很长一段时间我想不清“要呈现出什么样的效果”，秒表实验的时候我很清楚我要用数字管和按钮做出怎么样的效果，但在单周期 CPU 里我花了一段时间来弄明白这个问题，也反映出来 IO 这一块我学的不是很扎实，后来经过同学的帮助和老师上课的讲解总算是弄明白了，做出了可视化的想要的结果。

Modelsim 确实是很好用的，至少在仿真过后对于代码的逻辑都不用再特别担心了，之后都在考虑一些输入输出接口的问题，而有些同学在后期还要返工代码，实际上 verilog 的代码读起来稍微有点头疼，省去了返工代码的时间也是很好的一件事吧。

最近比较忙，这个实验也拖了一段时间才完成，最后能顺利完成，还是非常感谢同学助教和老师的帮助!