组成原理实验课程第六次实验报告

实验名称	单周期 CPU 改进			班级	张金老师班
学生姓名	姚知言	学号	2211290	指导老师	董前琨
实验地点	实验楼 A308		实验时间	2024 年 5 月 31 日	

1、 实验目的

理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。

了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。

熟悉并掌握单周期 CPU 的原理和设计。

进一步加强运用 verilog 语言进行电路设计的能力。

为后续设计多周期 cpu 的实验打下基础。

2、 实验内容说明

熟知 MIPS 指令类型，深入理解常用指令的功能和编码；

归纳常用的 MIPS 指令，确定自己准备实现的 MIPS 指令；

对准备实现的指令进行分析，完成表 7.1 的填写；

确认单周期 CPU 的设计框图的正确性；

编写 verilog 代码，将自己编写的汇编程序翻译为二进制，内嵌到指令 ROM 中；

对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料；

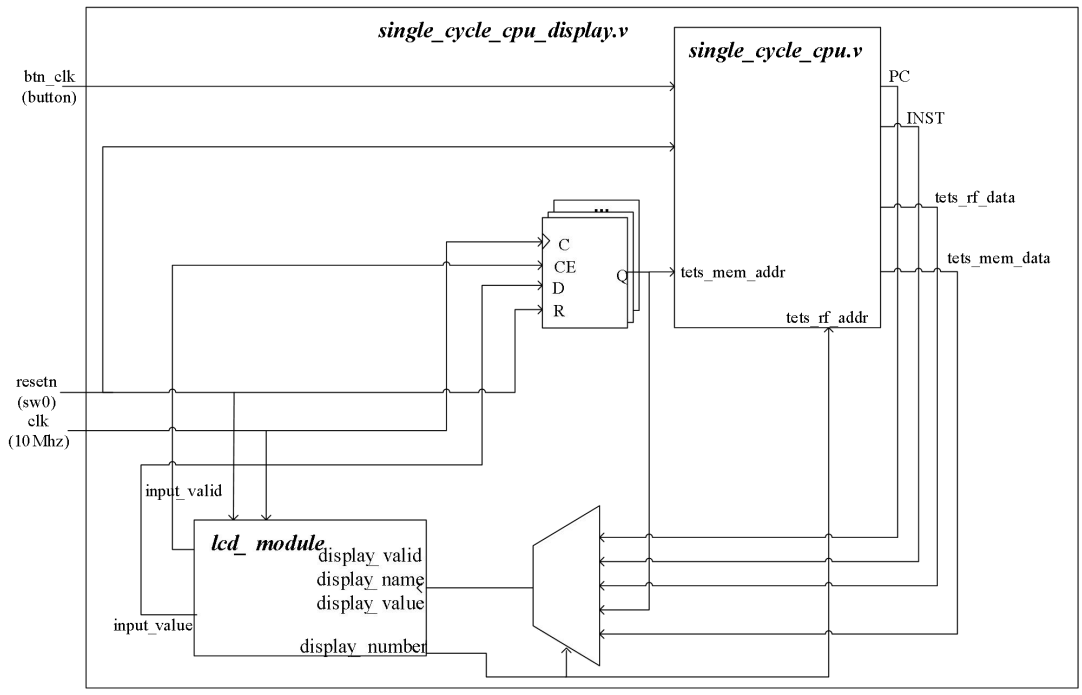
完成调用单周期 CPU 的外围模块的设计，并编写代码；

对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。

采用手动输入时钟，每个周期查看 CPU 状态，按照检查人员的要求进行演示，检查指令运行结果的正确性，可对演示结果进行拍照作为实验报告结果一项的材料。

实验结束后，需按照规定的格式完成实验报告的撰写。

3、 实验原理图



4、 实验步骤

（1）说明

在本实验中实现了以下四个指令。

a.MIPS sltu，R 型指令比较运算，sltu rd,rs,rt：if(rs<rt) rd=1 else rd=0

Bit	31-26	25-21	20-16	15-11	10-6	5-0
R-type	op	rs	rt	rd	shamt	func
sltu	000000	rs	rt	rd	00000	101011

b.MIPS andi，I 型指令位运算，andi rt,rs,immediate：rt=rs&immediate

Bit	31-26	25-21	20-16	15-0
I-type	op	rs	rt	immediate
andi	001100	rs	rt	immediate

c.MIPS sra，R 型指令移位运算，sra rd,rt,shamt：rd=rt>>shamt(符号位保留)

Bit	31-26	25-21	20-16	15-11	10-6	5-0
R-type	op	rs	rt	rd	shamt	func
sra	000000	00000	rt	rd	shamt	000011

d.结合 alu 实验改编指令，nand，R 型指令位运算，实现与非功能，主要仿照 and 指令结构进行创作，修改 func 为 111111 以防止与其他指令冲突。

nand rd,rs,rt：rd=^(rs&rt)

Bit	31-26	25-21	20-16	15-11	10-6	5-0
R-type	op	rs	rt	rd	shamt	func
nand	000000	rs	rt	rd	00000	111111

增加的验证语句如下。

```
assign inst_rom[18] = 32'h00C7682B;//sltu $13,$6,$7
assign inst_rom[19] = 32'h314E0084;//andi $14,$10,0x84
assign inst_rom[20] = 32'h00077883;//sra $15,$7,0x10
assign inst_rom[21] = 32'h00E6803F;//nand $16,$6,$7
```

实验过程中因为原有语句 bne 在仿真的时候有一些 bug，尝试解决未果，所以注释，其他保留，以上语句在最后一条 j 型语句前面添加，结果和详细说明见实验结果分析。

（2）inst_rom.v

```
//在本部分中，修改测试指令以验证功能
`timescale 1ns / 1ps
module inst_rom(
    input       [4 :0] addr,
    output reg [31:0] inst
    );
    wire [31:0] inst_rom[22:0];   //修改 rom 长度
    assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1     | $1 = 0000_0001H
    assign inst_rom[ 1] = 32'h00011100; // 04H: sll     $2 ,$1,#4     | $2 = 0000_0010H
    assign inst_rom[ 2] = 32'h00411821; // 08H: addu    $3 ,$2,$1     | $3 = 0000_0011H
    assign inst_rom[ 3] = 32'h00022082; // 0CH: srl     $4 ,$2,#2     | $4 = 0000_0004H
    assign inst_rom[ 4] = 32'h00642823; // 10H: subu    $5 ,$3,$4     | $5 = 0000_000DH
    assign inst_rom[ 5] = 32'hAC250013; // 14H: sw      $5 ,#19($1) | Mem[0000_0014H] =
0000_000DH
    assign inst_rom[ 6] = 32'h00A23027; // 18H: nor     $6 ,$5,$2     | $6 = FFFF_FFE2H
```

```
        assign inst_rom[ 7] = 32'h00C33825; // 1CH: or       $7 ,$6,$3      | $7 = FFFF_FFF3H
        assign inst_rom[ 8] = 32'h00E64026; // 20H: xor      $8 ,$7,$6      | $8 = 0000_0011H
        assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw       $8 ,#28($0) | Mem[0000_001CH] =
0000_0011H
        assign inst_rom[10] = 32'h00C7482A; // 28H: slt      $9 ,$6,$7      | $9 = 0000_0001H
        assign inst_rom[11] = 32'h11210002; // 2CH: beq      $9 ,$1,#2      |  跳转到指令 34H
        assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4      |  不执行
        assign inst_rom[13] = 32'h8C2A0013; // 34H: lw       $10,#19($1) | $10 = 0000_000DH
        assign inst_rom[14] = 32'h00415824; // 38H: and      $11,$2,$1      | $11 = 0000_0000H
        assign inst_rom[15] = 32'hAC0B001C; // 3CH: sw       $11,#28($0) | Men[0000_001CH]
= 0000_0000H
        assign inst_rom[16] = 32'hAC040010; // 40H: sw       $4 ,#16($0) | Mem[0000_0010H]
= 0000_0004H
        assign inst_rom[17] = 32'h3C0C000C; // 44H: lui      $12,#12        | [R12] =
000C_0000H
        assign inst_rom[18] = 32'h00C7682B;//48H
        assign inst_rom[19] = 32'h314E0084;//4CH
        assign inst_rom[20] = 32'h00077883;//50H
        assign inst_rom[21] = 32'h00E6803F;//54H
        assign inst_rom[22] = 32'h08000000;//58H: j          00H              |  跳转指令 00H
        always @(*)
        begin
            case (addr)
                5'd0 : inst <= inst_rom[0 ];
                5'd1 : inst <= inst_rom[1 ];
                5'd2 : inst <= inst_rom[2 ];
                5'd3 : inst <= inst_rom[3 ];
                5'd4 : inst <= inst_rom[4 ];
                5'd5 : inst <= inst_rom[5 ];
                5'd6 : inst <= inst_rom[6 ];
                5'd7 : inst <= inst_rom[7 ];
                5'd8 : inst <= inst_rom[8 ];
                5'd9 : inst <= inst_rom[9 ];
                5'd10: inst <= inst_rom[10];
                5'd11: inst <= inst_rom[11];
                5'd12: inst <= inst_rom[12];
                5'd13: inst <= inst_rom[13];
                5'd14: inst <= inst_rom[14];
                5'd15: inst <= inst_rom[15];
                5'd16: inst <= inst_rom[16];
                5'd17: inst <= inst_rom[17];
                5'd18: inst <= inst_rom[18];
                5'd19: inst <= inst_rom[19];
                5'd20: inst <= inst_rom[20];
```

```
                5'd21: inst <= inst_rom[21];
                5'd22: inst <= inst_rom[22];
                default: inst <= 32'd0;
            endcase
        end
endmodule
```

（3）single_cycle_cpu.v

```
`timescale 1ns / 1ps
`define STARTADDR 32'd0
module single_cycle_cpu(
        input clk,
        input resetn,
        input    [ 4:0] rf_addr,
        input    [31:0] mem_addr,
        output [31:0] rf_data,
        output [31:0] mem_data,
        output [31:0] cpu_pc,
        output [31:0] cpu_inst
        );
        reg    [31:0] pc;
        wire [31:0] next_pc;
        wire [31:0] seq_pc;
        wire [31:0] jbr_target;
        wire jbr_taken;
        assign seq_pc[31:2]       = pc[31:2] + 1'b1;
        assign seq_pc[1:0]        = pc[1:0];
        assign next_pc = jbr_taken ? jbr_target : seq_pc;
        always @ (posedge clk)
        begin
            if (!resetn) begin
                pc <= `STARTADDR;
            end
            else begin
                pc <= next_pc;
            end
        end

        wire [31:0] inst_addr;
        wire [31:0] inst;
        assign inst_addr = pc;
        inst_rom inst_rom_module(
            .addr          (inst_addr[6:2]),
            .inst          (inst                  )
        );
```

```
assign cpu_pc = pc;
assign cpu_inst = inst;
wire [5:0] op;
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [4:0] sa;
wire [5:0] funct;
wire [15:0] imm;
wire [15:0] offset;
wire [25:0] target;
assign op     = inst[31:26];
assign rs     = inst[25:21];
assign rt     = inst[20:16];
assign rd     = inst[15:11];
assign sa     = inst[10:6];
assign funct  = inst[5:0];
assign imm    = inst[15:0];
assign offset = inst[15:0];
assign target = inst[25:0];
wire op_zero;
wire sa_zero;
assign op_zero = ~(|op);
assign sa_zero = ~(|sa);
wire inst_ADDU, inst_SUBU , inst_SLT, inst_AND;
wire inst_NOR , inst_OR   , inst_XOR, inst_SLL;
wire inst_SRL , inst_ADDIU, inst_BEQ, inst_BNE;
wire inst_LW  , inst_SW   , inst_LUI, inst_J;
//增加新的指令
wire inst_SLTU,inst_ANDI,inst_SRA,inst_NAND;
assign inst_ADDU  = op_zero & sa_zero     & (funct == 6'b100001);//  无符号加法
assign inst_SUBU  = op_zero & sa_zero     & (funct == 6'b100011);//  无符号减法
assign inst_SLT   = op_zero & sa_zero     & (funct == 6'b101010);//  小于则置位
assign inst_AND   = op_zero & sa_zero     & (funct == 6'b100100);//  逻辑与运算
assign inst_NOR   = op_zero & sa_zero     & (funct == 6'b100111);//  逻辑或非运算
assign inst_OR    = op_zero & sa_zero     & (funct == 6'b100101);//  逻辑或运算
assign inst_XOR   = op_zero & sa_zero     & (funct == 6'b100110);//  逻辑异或运算
assign inst_SLL   = op_zero & (rs==5'd0) & (funct == 6'b000000);//  逻辑左移
assign inst_SRL   = op_zero & (rs==5'd0) & (funct == 6'b000010);//  逻辑右移
assign inst_ADDIU = (op == 6'b001001);                          //  立即数无符号加法
assign inst_BEQ   = (op == 6'b000100);                          //  判断相等跳转
assign inst_BNE   = (op == 6'b000101);                          //  判断不等跳转
assign inst_LW    = (op == 6'b100011);                          //  从内存装载
assign inst_SW    = (op == 6'b101011);                          //  向内存存储
```

```
    assign inst_LUI     = (op == 6'b001111);                    // 立即数装载高半字节
    assign inst_J       = (op == 6'b000010);                    // 直接跳转
    assign inst_SLTU = op_zero & sa_zero & (funct == 6'b101011);
    assign inst_ANDI = (op==6'b001100);
    assign inst_SRA = op_zero & (rs==5'd0) & (funct == 6'b000011);
    assign inst_NAND = op_zero & sa_zero & (funct == 6'b111111);
    // 无条件跳转判断
    wire            j_taken;
    wire [31:0] j_target;
    assign j_taken    = inst_J;
    // 无条件跳转目标地址：PC={PC[31:28],target<<2}
    assign j_target = {pc[31:28], target, 2'b00};
   //为了解决未定义的 warning 更改了寄存器堆和跳转判断的顺序，不知道有没有影响
    wire rf_wen;
    wire [4:0] rf_waddr;
    wire [31:0] rf_wdata;
    wire [31:0] rs_value, rt_value;
     regfile rf_module(
        .clk      (clk          ),    // I, 1
        .wen      (rf_wen       ),    // I, 1
        .raddr1 (rs           ),    // I, 5
        .raddr2 (rt           ),    // I, 5
        .waddr    (rf_waddr ),    // I, 5
        .wdata    (rf_wdata ),    // I, 32
        .rdata1 (rs_value ),    // O, 32
        .rdata2 (rt_value ),     // O, 32
        //display rf
        .test_addr(rf_addr),
        .test_data(rf_data)
    );
    //分支跳转
    wire            beq_taken;
    wire            bne_taken;
    wire [31:0] br_target;
    assign beq_taken = (rs_value == rt_value);          // BEQ 跳转条件：GPR[rs]=GPR[rt]
    assign bne_taken = ~beq_taken;                          // BNE 跳转条件：
GPR[rs]≠GPR[rt]
    assign br_target[31:2] = pc[31:2] + {{14{offset[15]}}, offset};
    assign br_target[1:0]    = pc[1:0];      // 分支跳转目标地址：PC=PC+offset<<2
    //跳转指令的跳转信号和跳转目标地址
    assign jbr_taken = j_taken          // 指令跳转：无条件跳转 或 满足分支跳转条件
                            | inst_BEQ & beq_taken
                            | inst_BNE & bne_taken;
    assign jbr_target = j_taken ? j_target : br_target;
```

```
// 传递到执行模块的 ALU 源操作数和操作码
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;
wire inst_nand;//增加一个 ALU 模块
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
assign inst_sub = inst_SUBU; // 减法
assign inst_slt = inst_SLT;   // 小于置位
assign inst_sltu= inst_SLTU;          // 无符号小于置位
assign inst_and = inst_AND | inst_ANDI;   // 逻辑与
assign inst_nor = inst_NOR;   // 逻辑或非
assign inst_or    = inst_OR;     // 逻辑或
assign inst_xor = inst_XOR;   // 逻辑异或
assign inst_sll = inst_SLL;   // 逻辑左移
assign inst_srl = inst_SRL;   // 逻辑右移
assign inst_sra = inst_SRA;          // 算术右移
assign inst_lui = inst_LUI;   // 立即数装载高位
assign inst_nand = inst_NAND;

wire [31:0] sext_imm;
wire      inst_shf_sa;       //使用 sa 域作为偏移量的指令
wire      inst_imm_sign;    //对立即数作符号扩展的指令
assign sext_imm           = {{16{imm[15]}}, imm};// 立即数符号扩展
assign inst_shf_sa      = inst_SLL | inst_SRL | inst_SRA;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW |inst_ANDI;
//为需要使用立即数符号扩展操作的 andi 和偏移量移位操作的 sra 更改 wire
wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
wire [12:0] alu_control;
assign alu_operand1 = inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_imm_sign ? sext_imm : rt_value;
assign alu_control = {inst_nand,
                      inst_add,           // ALU 操作码，独热编码
                      inst_sub,
                      inst_slt,
                      inst_sltu,
                      inst_and,
                      inst_nor,
                      inst_or,
                      inst_xor,
                      inst_sll,
                      inst_srl,
                      inst_sra,
                      inst_lui};
```

```
        wire [31:0] alu_result;
        alu alu_module(
            .alu_control    (alu_control ),    // I, 12, ALU 控制信号
            .alu_src1       (alu_operand1),    // I, 32, ALU 操作数 1
            .alu_src2       (alu_operand2),    // I, 32, ALU 操作数 2
            .alu_result     (alu_result    )     // O, 32, ALU 结果
        );
        wire [3 :0] dm_wen;
        wire [31:0] dm_addr;
        wire [31:0] dm_wdata;
        wire [31:0] dm_rdata;
        assign dm_wen      = {4{inst_SW}} & resetn;       // 内存写使能,非 resetn 状态下有效
        assign dm_addr     = alu_result;                  // 内存写地址，为 ALU 结果
        assign dm_wdata = rt_value;                       // 内存写数据，为 rt 寄存器值
        data_ram data_ram_module(
            .clk     (clk             ),    // I, 1,   时钟
            .wen     (dm_wen          ),    // I, 1,   写使能
            .addr    (dm_addr[6:2]),    // I, 32,  读地址
            .wdata (dm_wdata       ),    // I, 32,  写数据
            .rdata (dm_rdata       ),    // O, 32,  读数据
            .test_addr(mem_addr[6:2]),
            .test_data(mem_data        )
        );
        wire inst_wdest_rt;     // 寄存器堆写入地址为 rt 的指令
        wire inst_wdest_rd;     // 寄存器堆写入地址为 rd 的指令
        assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI | inst_ANDI;
        assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                                | inst_OR     | inst_XOR    | inst_SLL | inst_SRL | inst_SLTU |
inst_SRA | inst_NAND;   //指定写入地址
        assign rf_wen      = (inst_wdest_rt | inst_wdest_rd) & resetn;
        assign rf_waddr = inst_wdest_rd ? rd : rt;
        assign rf_wdata = inst_LW ? dm_rdata : alu_result;
    endmodule
    （4）single_cycle_cpu.xdc
    ......
    set_property IOSTANDARD LVCMOS33 [get_ports btn_clk]
    //有一个疑似大小写错误，不知道有没有影响，因为改了就写一下
    ......
    （5）alu.v
    `timescale 1ns / 1ps
    module alu(
        input    [12:0] alu_control,   //调整位宽，增加新控制指令
        input    [31:0] alu_src1,        // ALU 操作数 1,为补码
        input    [31:0] alu_src2,        // ALU 操作数 2，为补码
```

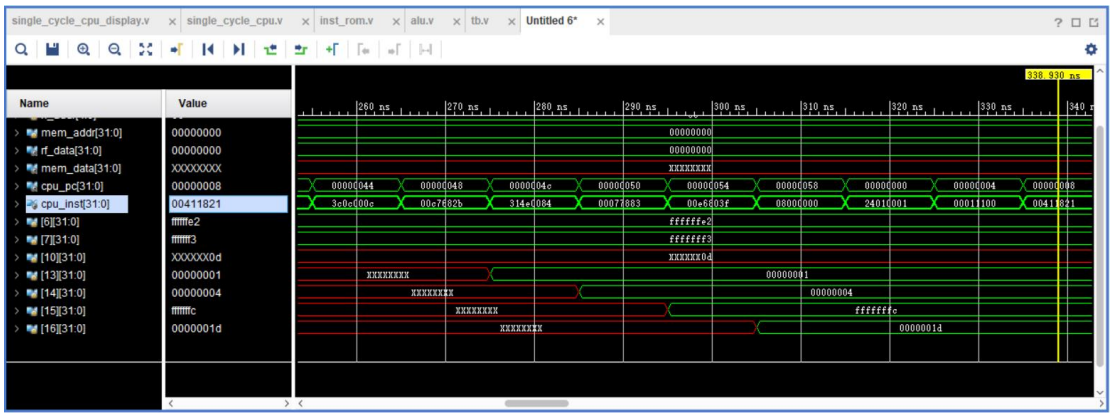
```
output [31:0] alu_result       // ALU 结果
);
// ALU 控制信号，独热码
wire alu_add;     //加法操作
wire alu_sub;     //减法操作
wire alu_slt;     //有符号比较，小于置位，复用加法器做减法
wire alu_sltu;    //无符号比较，小于置位，复用加法器做减法
wire alu_and;     //按位与
wire alu_nor;     //按位或非
wire alu_or;      //按位或
wire alu_xor;     //按位异或
wire alu_sll;     //逻辑左移
wire alu_srl;     //逻辑右移
wire alu_sra;     //算术右移
wire alu_lui;     //高位加载
wire alu_nand;//增加 wire
assign alu_nand = alu_control[12];
assign alu_add    = alu_control[11];
assign alu_sub    = alu_control[10];
assign alu_slt    = alu_control[ 9];
assign alu_sltu = alu_control[ 8];
assign alu_and    = alu_control[ 7];
assign alu_nor    = alu_control[ 6];
assign alu_or     = alu_control[ 5];
assign alu_xor    = alu_control[ 4];
assign alu_sll    = alu_control[ 3];
assign alu_srl    = alu_control[ 2];
assign alu_sra    = alu_control[ 1];
assign alu_lui    = alu_control[ 0];
wire [31:0] add_sub_result;
wire [31:0] slt_result;
wire [31:0] sltu_result;
wire [31:0] and_result;
wire [31:0] nor_result;
wire [31:0] or_result;
wire [31:0] xor_result;
wire [31:0] sll_result;
wire [31:0] srl_result;
wire [31:0] sra_result;
wire [31:0] lui_result;
wire [31:0] nand_result;
assign and_result = alu_src1 & alu_src2;         // 与结果为两数按位与
assign or_result    = alu_src1 | alu_src2;       // 或结果为两数按位或
assign nor_result = ~or_result;                  // 或非结果为或结果按位取反
```

```
assign xor_result = alu_src1 ^ alu_src2;          // 异或结果为两数按位异或
assign lui_result = {alu_src2[15:0], 16'd0};    // 立即数装载结果为立即数移位至高半字节

assign nand_result = ~and_result;//计算新的运算结果
.....//中间其他 result 计算过程无需修改，略去
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt          ? slt_result :
                    alu_sltu         ? sltu_result :
                    alu_and           ? and_result :
                    alu_nor           ? nor_result :
                    alu_or            ? or_result    :
                    alu_xor           ? xor_result :
                    alu_sll          ? sll_result :
                    alu_srl           ? srl_result :
                    alu_sra           ? sra_result :
                    alu_lui           ? lui_result :
                    alu_nand           ? nand_result://输出
                    32'd0;
endmodule
```

5、 实验结果分析

（1）仿真结果



从图中可以看出：

执行到 0x48 的语句 0x00C7682B 的时候，寄存器 13 的值置为 0x1；

执行到 0x4C 的语句 0x314e0084 的时候，寄存器 14 的值置为 0x4；

执行到 0x50 的语句 0x00077883 的时候，寄存器 15 的值置为 0xfffffffc；

执行到 0x54 的语句 0x00e6803f 的时候，寄存器 16 的值置为 0x1d。

（结果分析见下一部分上箱验证，无本质区别）

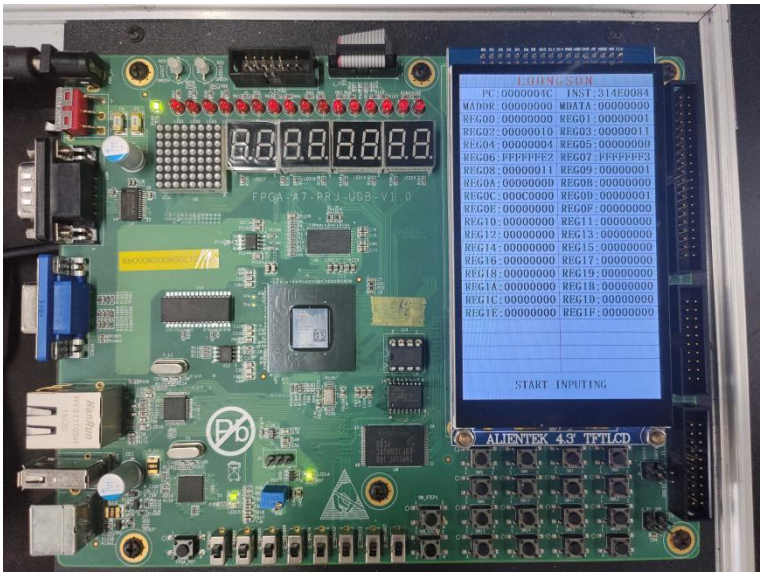
（2）上箱验证

a.初始状态。

在后续上箱验证的过程中，对于内存没有读写操作，故不需要展示。仅有对寄存器 6（06），7（07），10（0A）的读操作以及寄存器 13（0D），14（0E），15（0F），16（10）的写操作。

此时寄存器 13（0D）-16（10）的值均为 0，寄存器 6（06）的值为 0xFFFFFFE2，寄存器 7（07）的值为 0xFFFFFFF3，寄存器 10（0A）的值为 0x0000000D。

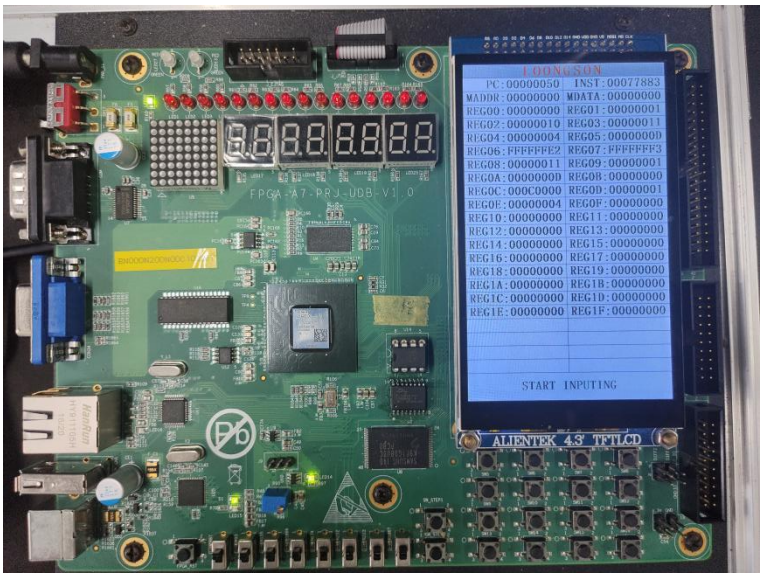
b.执行 0x48 的语句 0x00C7682B



该语句本质上是 MIPS sltu 操作：000000 00111 00110 01101 00000 101011，即 sltu $13,$6,$7。

由于 REF06 的值为 0xFFFFFFE2，REF07 的值为 0xFFFFFFF3，无符号比较中 REF06<REF07，因此将 REF1D 置为 0x00000001。

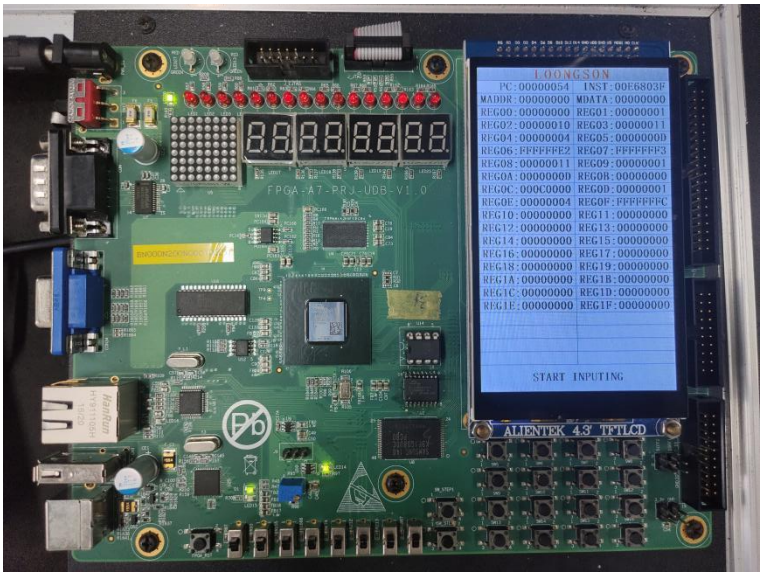
c.执行 0x4C 的语句 0x314e0084



该语句本质上是 MIPS andi 操作：001100 01010 01110 0000000010000100，即 andi $14,$10,0x84。

REF0A 的值为 0x0000000D（即 0000 1101）与 0x84（即 1000 0100）按位与的结果为 0x4（即 0000 0100），所以将 REF0E 的值置为 0x00000004。

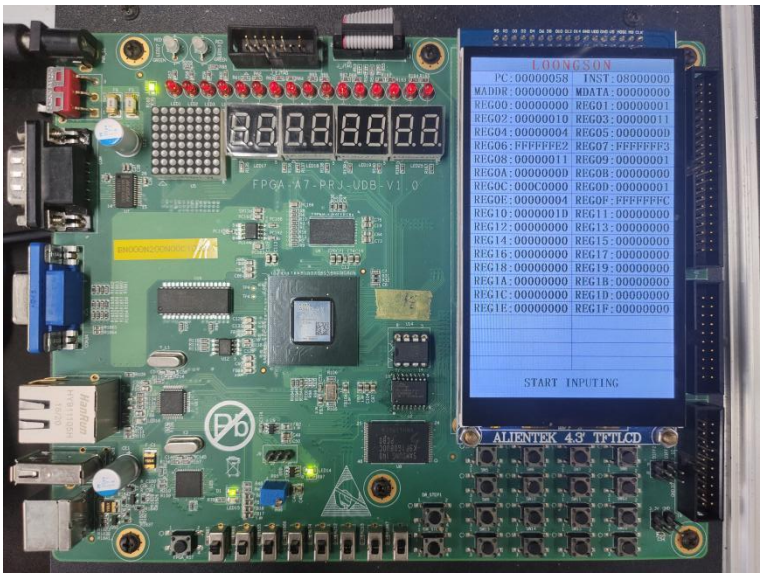
d.执行 0x50 的语句 0x00077883



该语句本质上是 MIPS sra 操作：000000 00000 00111 01111 00010 000011，即 sra $15,$7,0x10。

REF07 的值为 0xFFFFFFF3，即 1111 1111 1111 1111 1111 1111 1111 0011，将其算术右移两位，需要保留符号位，即为 1111 1111 1111 1111 1111 1111 1111 1100，0xFFFFFFFC。将 REF0F 的值置为 0xFFFFFFFC。

e.执行 0x54 的语句 0x00e6803



该语句本质上是我自己实现的 nand 运算 000000 00111 00110 10000 00000 111111（指令格式见实验步骤开始的说明部分），即为 nand $16,$6,$7。

REF06 的值为 0xFFFFFFE2（1111 1111 1111 1111 1111 1111 1110 0010），REF07 的值为 0xFFFFFFF3（1111 1111 1111 1111 1111 1111 1111 0011），与非结果为 0x1D（0000 0000 0000 0000 0000 0000 0001 1101），将其赋值给 REF10。

6、 总结感想

在本次实验中，我针对单周期 CPU 进行了改进，实现了 MIPS 指令集中的 SLTU，ANDI，SRA，并自己实现了 NAND 指令。以这四个指令的实现为例，基本上尽可能多的体会了实现指令的类型。

计算机组成原理的实验到此告一段落，该实验课夯实我的理论基础，也为我后续的计算机体系机构等课程的学习打好了基础，很有收获！