

组成原理实验课程第 6 次实验报告

实验名称	单周期 CPU 改进			班级	李涛
学生姓名	阿斯雅	学号	2210737	指导老师	董前琨
实验地点	实验楼 A 区 308		实验时间	5 月 30 日 中午	

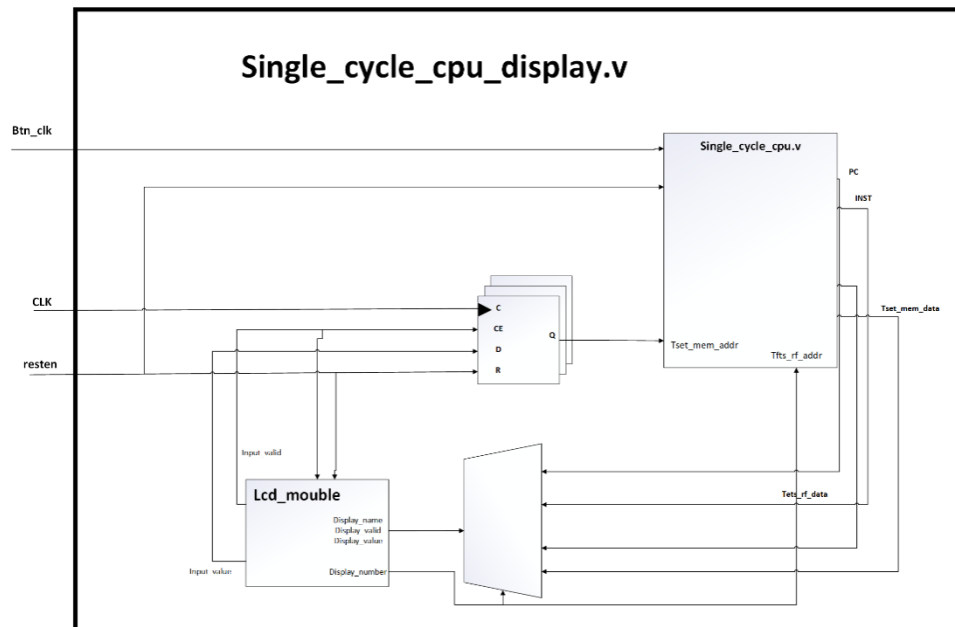
一、实验目的

1. 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。
2. 了解熟悉 MIPS 体系的处理器结构,如延迟槽,哈佛结构的概念。
3. 熟悉并掌握单周期 CPU 的原理和设计。
4. 进一步加强运用 verilog 语言进行电路设计的能力。
5. 为后续设计多周期 cpu 的实验打下基础。

二、实验内容说明

结合实验指导手册中的实验六（单周期 CPU 实验）完成功能改进，在原有 CPU 基础上，扩充 CPU 可运行的 MIPS 指令，要求扩充的指令应为一个时钟周期内能够执行完的指令，要求至少一个 R 型，一个 I 型，另外一个自选。

三、实验原理图



四、实验过程

4.1、分析指令执行原理

在 single_cycle_cpu.v 模块中，我们分析代码可知该程序把指令执行划分成了取指令，指令译码，取操作数执行，访问存储器，写回寄存器等五个步骤，接下来我将一一讲解他们。

● 取指令

在取指令阶段，程序一开始的时候把 PC 设置为了 0，然后又定义了两个变量 jbr_target 和 seq_pc，分别对应着跳转指令后的新 PC 和无跳转指令后的新 PC，其中 jbr_target 需要在指令译码阶段才能确定，而 seq_pc 是在原先的 PC 上加 4 后得到的新的地址，又因为是字节编地址，所以我们只需要在 PC 前 30 位上加 1 就可。之后通过控制信号 jbr_taken 决定 PC，最后根据新 PC 从指令存储器里面进行取指令。

```
// 下一指令地址: seq_pc=pc+4
assign seq_pc[31:2] = pc[31:2] + 1'b1;
assign seq_pc[1:0] = pc[1:0];
// 新指令: 若指令跳转, 为跳转地址; 否则为下一指令
assign next_pc = jbr_taken ? jbr_target : seq_pc;
always @ (posedge clk) // PC程序计数器
begin
    if (!resetn) begin
        pc <= 32'd0 ; // 复位, 取程序起始地址
    end
    else begin
        pc <= next_pc; // 不复位, 取新指令
    end
end

wire [31:0] inst_addr;
wire [31:0] inst;
assign inst_addr = pc; // 指令地址: 指令长度32位
inst_rom inst_rom_module( // 指令存储器
    .addr (inst_addr[6:2]), // I, 5, 指令地址
    .inst (inst) // 0, 32, 指令
);
assign cpu_pc = pc; //display pc
assign cpu_inst = inst;
```

● 指令译码

取到指令后，程序要根据指令中各个字段的值来决定该指令到底是什么样的指令，因为要考虑到所有情况，所以程序根据指令来获得每个情况中的字段相应的值，得到字段的值后接下来就要判断当前指令要实现什么功能，在这里列举一些情况:

比如当 op 字段和 shamt 字段为零的时候，该指令就是 R 型指令，所以要根据最后的 funct 字段来确定该指令要实现的功能，如果是 32 的话就执行两个操作数加法，如果是 34 的话就执行两个操作数减法以此类推。

再比如，如果 op 字段是 35 的话，该指令就是从内存加载数据的 I 型指令，要根据最后的 16 位数据和 rs 字段中的值来计算出内存地址，然后在 data_ram 模

块里面取出相应的值。存储指令亦是如此。

最后是跳转指令，这里分为无条件跳转指令和有条件跳转指令，其中无条件跳转指令实现相对比较简单，直接把指令中最后 26 位数据左移两位然后跟 PC 最前面 4 位拼接就可以得到新的 PC 值，而有条件跳转指令则需要先判断 rs,rt 中的值是否相等或不相等，如果满足条件就需要把最后 16 位中的值左移两位然后加到当前 PC 上得到新的 PC 值。

```
// 实现指令列表
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;
wire inst_NAND, inst_Load, inst_Bltui;

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_NAND = op_zero & sa_zero & (funct == 6'b100111); // 逻辑与非运算
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_Load = op_zero & sa_zero & (funct == 6'b111111); // 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);
```

● 取操作数执行

在这一阶段，程序首先要通过调用寄存器堆模块来获得相应的寄存器值，然后在根据第二阶段中指令译码决定 ALU 要做什么操作，也就是给 ALU 输入对应的控制信号，之后通过控制信号来决定 ALU 的操作数是从寄存器堆提供的还是直接从指令来提供的。最后通过 ALU 来计算出执行结果。

```

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;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW | inst_bltui;

wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
wire [14:0] 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_load,
                      inst_nand,
                      inst_bltui,
                      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};

```

- 访问内存

程序该阶段主要执行的功能是写内存功能，因为在第二阶段程序就把读内存操作完成了。在该阶段程序通过控制信号来决定是否需要写内存，如果需要写内存就调用数据存储器模块把 rt 寄存器的值写到 ALU 计算出的地址里面。

```

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

    //display mem
    .test_addr(mem_addr[6:2]),
    .test_data(mem_data)
);

```

● 写回

最后的阶段是写回阶段，在该阶段程序通过控制信号来决定是否需要写回内存，如果需要的话通过调用寄存器堆模块把 ALU 计算结果写回到 rd 寄存器中，至此一条指令的全部执行阶段就完成了。

```

//-----{写回}begin-----//
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;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                    | inst_OR | inst_XOR | inst_SLL | inst_SRL;
// 寄存器堆写使能信号, 非复位状态下有效
assign rf_wen = (inst_wdest_rt | inst_wdest_rd) & resetn;
assign rf_waddr = inst_wdest_rd ? rd : rt; // 寄存器堆写地址rd或rt
assign rf_wdata = inst_LW ? dm_rdata : alu_result; // 写回结果, 为load结果或ALU结果
assign result_data = rf_wdata;
//-----{写回}end-----//
endmodule

```

4.2、增加指令

我第一个增加的指令是 R 型与非指令-nand，要实现这个并不复杂，只要在 ALU 模块里面给控制信号增加一位然后把原先的与指令的结果取个反就行，修改完 ALU 模块接着要修改 CPU 模块，我们要给这个新 R 型指令定义它专属的 funct 字段，在本次实验中我设置 funct 字段为 100111，也要在调用 ALU 模块之前计算好 ALU 的控制信号。最后要在 inst_rom 模块中增加相应的指令，因为 op 字段是 0，shamt 字段是 0，funct 字段是 101111，所以我们只要定义好

两个源操作数和目标寄存器就可，在本次实验中，我三个寄存器都设置为一号寄存器，所以最后的指令为：

00000000001000010000100000101111-0021082fh

第二个新增的指令为 I 型无符号大于则置位指令-bltui，也就是实现寄存器和无符号常数之间的比较，如果寄存器里面的数大于常数，就把目标寄存器里的值设为 1，否则为 0，因为是 I 型指令，所以要在计算控制信号 inst_imm_sign 中也要把 inst_bltui 加进去。其他的操作跟上面指令的步骤大差不差，在 ALU 中新增一位控制信号，然后基于小于则置位指令定义大于则置位运算，然后在 CPU 模块中给该指令定义专属操作码，在该实验中我定义的操作码为 000011，接着计算好源操作数和目标寄存器，最后在 inst_rom 模块中新增指令，而在本次实验中我选择的目标寄存器，源寄存器和常数分别为一号寄存器，一号寄存器和常数 ffffh。对应的指令为：

00001100001000011111111111111111- 0C21ffffh

最后自定义的一个指令也是 R 型指令 load，所执行的操作是把源寄存器 1 的高 16 位和源寄存器 2 的低 16 位拼接起来放到目标寄存器里面。我给它定义的专属 funct 字段值为 111111，其他的操作跟上面两个没太大区别。最好我选择的目标寄存器，源寄存器分别是二号寄存器，一号寄存器和二号寄存器，对应的指令为：

00000000001000100001000000111111- 0022103fh

最后也不能忘了在控制写回 rd 寄存器的控制信号和控制写回 rt 寄存器的控制信号中加入所加的指令。而因为我加的三个指令都需要写回寄存器，所以为了在仿真结果中更好的看出指令的执行是否正确，我也在仿真输出中多加了一个输出-result，用来输出最终写回寄存器的值。

```

assign alu_load=alu_control[14];
assign alu_nand=alu_control[13];
assign alu_bltui=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];

```

```

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;
assign load_result={alu_src1[31:16], alu_src2[15:0]}; //拼接
assign bltui_result = {31'd0, adder_cout}; //大于则置位
assign nand_result = ~and_result; //与非 // 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

```



```

Untitled 1 x inst_rom.v x single_cpu.v* x
? ⓘ

85 wire inst_NOR, inst_OR, inst_XOR, inst_SLL;
86 wire inst_SRL, inst_ADDIU, inst_BEQ, inst_BNE;
87 wire inst_LW, inst_SW, inst_LUI, inst_J;
88 wire inst_MAND, inst_Load, inst_Bltui;
89
90 assign inst_ADDU = op_zero & sa_zero & (funct == 6'b100001); // 无符号加法
91 assign inst_SUBU = op_zero & sa_zero & (funct == 6'b100011); // 无符号减法
92 assign inst_SLT = op_zero & sa_zero & (funct == 6'b101010); // 小于则置位
93 assign inst_AND = op_zero & sa_zero & (funct == 6'b100100); // 逻辑与运算
94 assign inst_MAND = op_zero & sa_zero & (funct == 6'b101111); // 逻辑与非运算
95 assign inst_NOR = op_zero & sa_zero & (funct == 6'b100111); // 逻辑或非运算
96 assign inst_OR = op_zero & sa_zero & (funct == 6'b100101); // 逻辑或运算
97 assign inst_XOR = op_zero & sa_zero & (funct == 6'b100110); // 逻辑异或运算
98 assign inst_SLL = op_zero & (rs==5'd0) & (funct == 6'b000000); // 逻辑左移
99 assign inst_SRL = op_zero & (rs==5'd0) & (funct == 6'b000010); // 逻辑右移
100
101 assign inst_Load = op_zero & sa_zero & (funct == 6'b111111); // 加载
102 assign inst_MAND = op_zero & sa_zero & (funct == 6'b101111); // 逻辑与非运算
103 assign inst_Bltui = (op == 6'b000011); // 大于则置位 (I型)
104 assign inst_ADDIU = (op == 6'b001001); // 立即数无符号加法
105 assign inst_BEQ = (op == 6'b000100); // 判断相等跳转
106 assign inst_BNE = (op == 6'b000101); // 判断不等跳转
107 assign inst_LW = (op == 6'b100011); // 从内存装载
108 assign inst_SW = (op == 6'b101011); // 向内存存储
109 assign inst_LUI = (op == 6'b001111); // 立即数装载高半字节
110 assign inst_J = (op == 6'b000010);
111
112 // 无条件跳转判断
113

```

```

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;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW | inst_bltui;

wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
wire [14:0] alu_control;

```



```

module single_cycle_cpu(
    input clk,      // 时钟
    input resetn,   // 复位信号，低电平有效

    //display data
    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,
    output [31:0] result_data
);

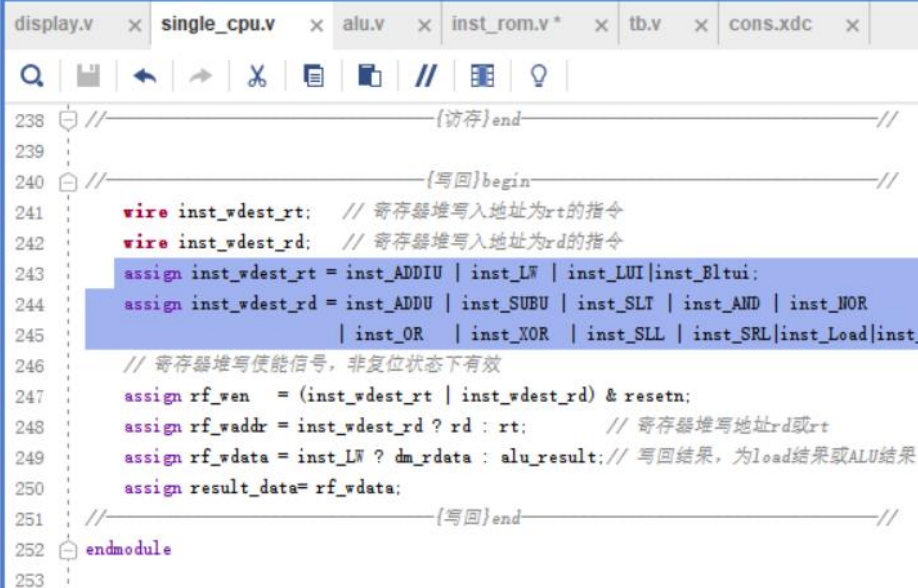
```

```

    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;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                          | inst_OR   | inst_XOR | inst_SLL | inst_SRL;
    // 寄存器堆写使能信号，非复位状态下有效
    assign rf_wen  = (inst_wdest_rt | inst_wdest_rd) & resetn;
    assign rf_waddr = inst_wdest_rd ? rd : rt;      // 寄存器堆写地址rd或rt
    assign rf_wdata = inst_LW ? dm_rdata : alu_result; // 写回结果，为load结果或ALU结果
    assign result_data = rf_wdata;

    //-----{写回}end-----//
endmodule

```



```

display.v x single_cpu.v x alu.v x inst_rom.v* x tb.v x cons.xdc x
238 //-----{访存}end-----//
239
240 //-----{写回}begin-----//
241     wire inst_wdest_rt; // 寄存器堆写入地址为rt的指令
242     wire inst_wdest_rd; // 寄存器堆写入地址为rd的指令
243     assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI|inst_Extui;
244     assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
245                           | inst_OR   | inst_XOR | inst_SLL | inst_SRL|inst_Load|inst_HAND;
246     // 寄存器堆写使能信号，非复位状态下有效
247     assign rf_wen  = (inst_wdest_rt | inst_wdest_rd) & resetn;
248     assign rf_waddr = inst_wdest_rd ? rd : rt;      // 寄存器堆写地址rd或rt
249     assign rf_wdata = inst_LW ? dm_rdata : alu_result; // 写回结果，为load结果或ALU结果
250     assign result_data = rf_wdata;
251     //-----{写回}end-----//
252 endmodule
253

```

```

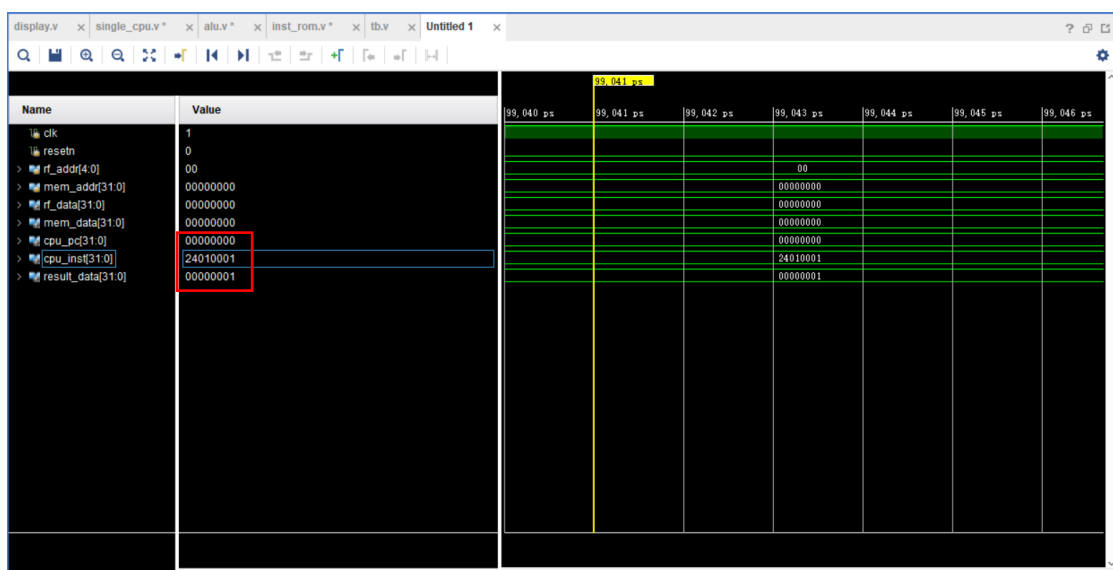
// Outputs
wire [31:0] rf_data;
wire [31:0] mem_data;
wire [31:0] cpu_pc;
wire [31:0] cpu_inst;
wire [31:0] result_data;

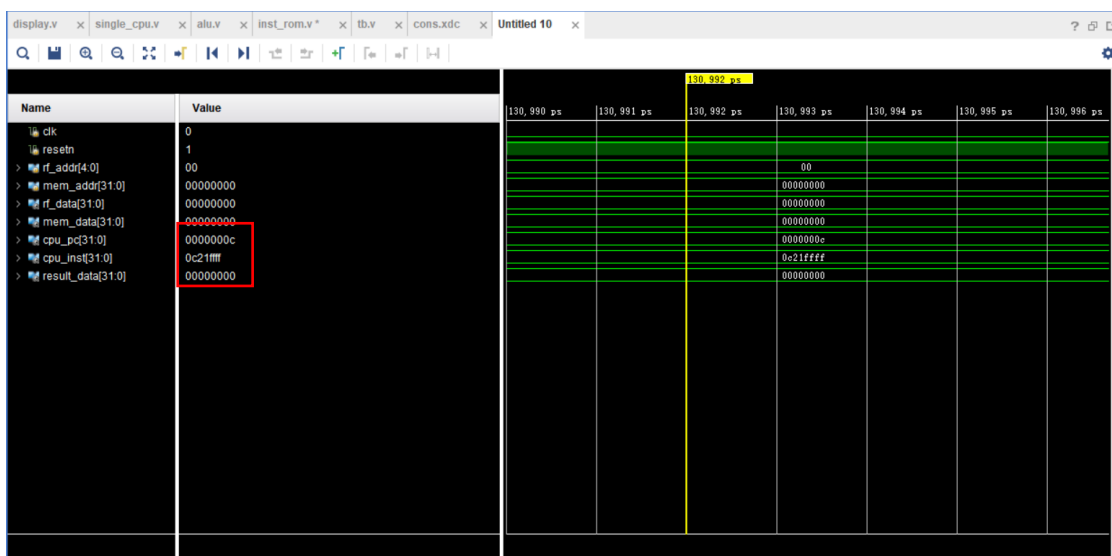
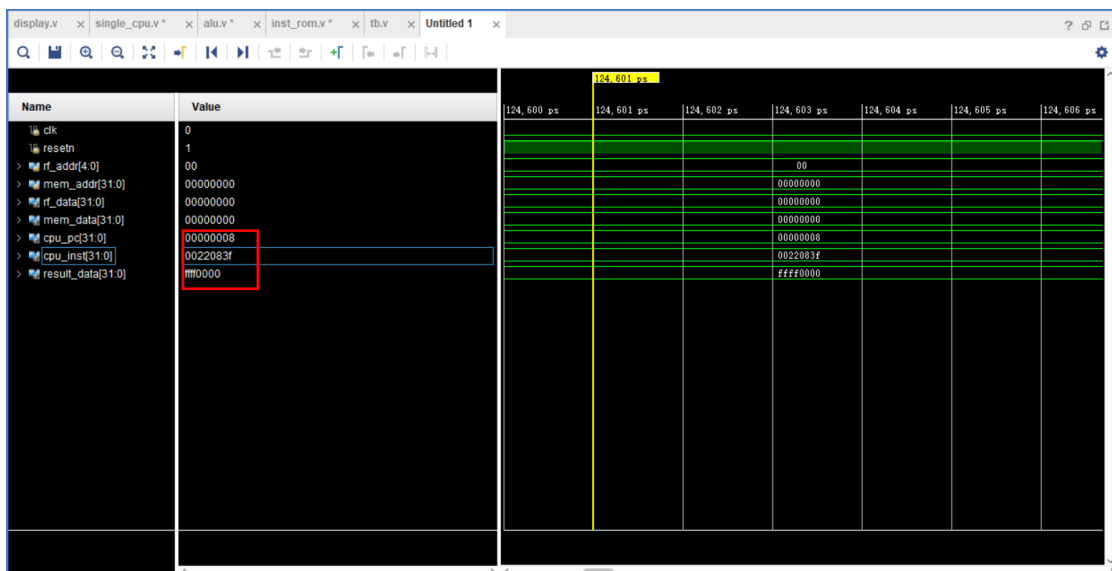
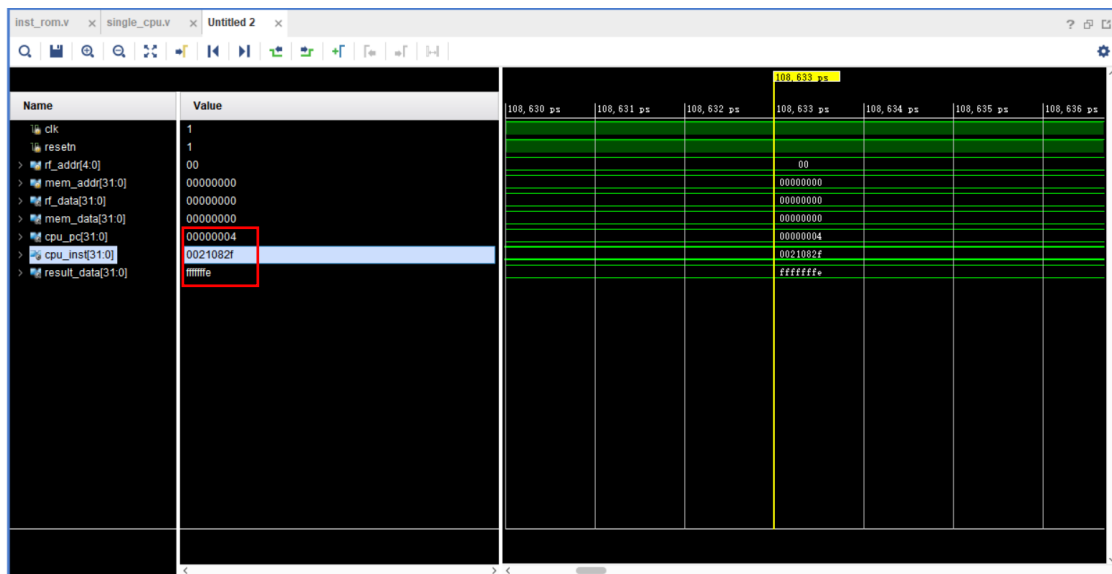
// Instantiate the Unit Under Test (UUT)
single_cycle_cpu uut (
    .clk(clk),
    .resetn(resetn),
    .rf_addr(rf_addr),
    .mem_addr(mem_addr),
    .rf_data(rf_data),
    .mem_data(mem_data),
    .cpu_pc(cpu_pc),
    .cpu_inst(cpu_inst),
    .result_data(result_data)
);

```

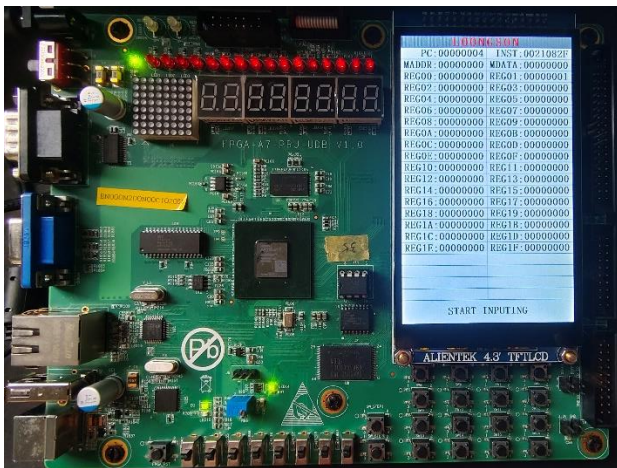
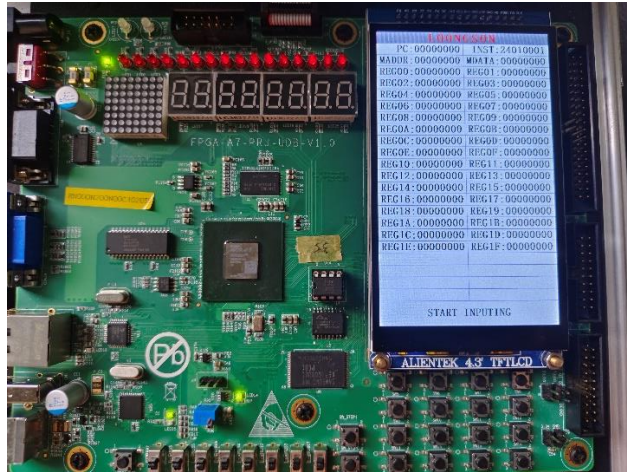
五、实验结果

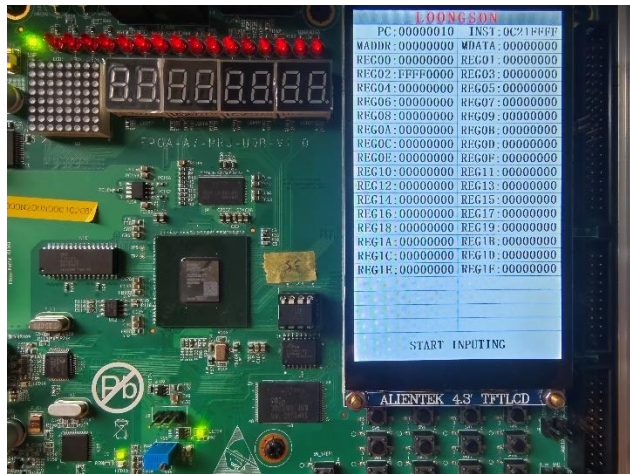
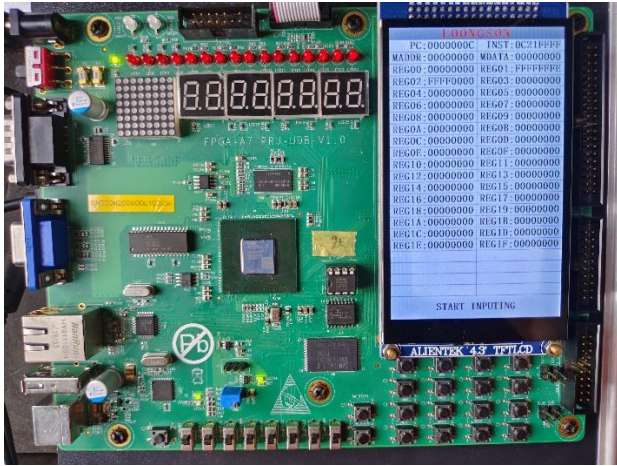
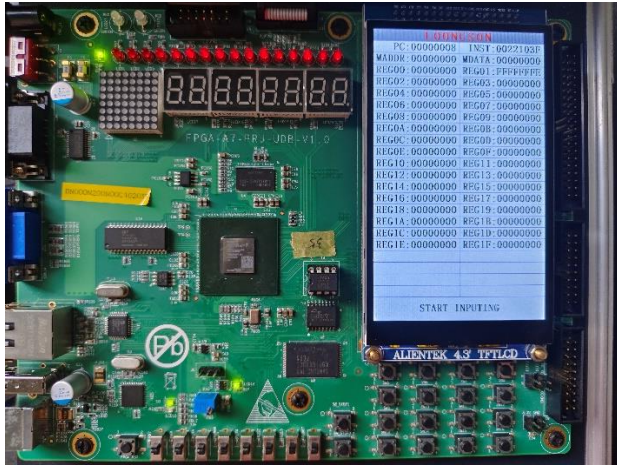
5.1、仿真结果





5.2、上机效果





5.3、结果解释

第一条指令是原来的指令 24010001h，也就是给一号寄存器设置为 1，我们也可以在仿真结果和上机结果中可以看出当 pc=4 的时候一号寄存器的值变成了 1，第二条指令是我自己扩充的与非指令，因为一号寄存器里面的值是 1，所以 1 跟自己与非的结果为：

[illegible]

我们可以从仿真结果和上机结果中看出当第二条指令执行完之后一号寄存器里的值变成了 FFFFFFFE，也是符合我们预期的。相应的第三条指令是我自己加的装载指令，第四条指令是我自己加的 I 型无符号比较指令，我们可以从结果中可以看出当第三条指令执行完二号寄存器里面的值变成了 FFFF0000，这也符合我们的预期，因为这个结果的高 16 位来自一号寄存器的值而低 16 位来自二号寄存器的值，因为一号寄存器和二号寄存器的值为 FFFFFFFEh 和 00000000h，所以二号寄存器里的结果就变成了 FFFF0000。而当第四条指令执行完一号寄存器的值变成了 0，这是因为在执行无符号比较的时候程序首先会执行符号扩展，也就是用最高位填满高 16 位，所以当 16 位无符号常数是 ffffh 的时候会变成 ffffffffh，所以这时候它是比一号寄存器 ffffffffh 大的，所以一号寄存器不会被置位，而是会被复位，变成零，这也可以在我们仿真结果和上机结果中看到。

六、心得体会

通过本次单周期 CPU 实验的实操，我深刻理解了计算机组成原理理论课上所讲授的关于简单单周期处理器运行指令的机理。通过亲手编写和调试代码，我不仅观察到了指令如何在处理器内部流转和处理，还通过增加不同类型的指令来进一步探讨了处理器如何识别并处理这些指令。这一实践过程使我更加清晰地认识到处理器如何区分和执行各种运算，从而加深了我对计算机底层工作原理的认识。