组成原理实验课程第六次实验报告

实验名称	单周期 CPU 的实现			班级	张金
学生姓名	冯思程	学号	2112213	指导老师	董前琨
实验地点	实验楼 A 306		实验时间	2023.6.5 14：00	

1、 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。

2、实验内容说明

针对组成原理第六次的单周期 CPU 实验，要求：

1）原始代码实验验证使用实验箱验证，可以不进行仿真，验证时在运行一系列指令之后，实验箱拍照，对比说明各个寄存器中的数据是否是执行正确的结果即可。

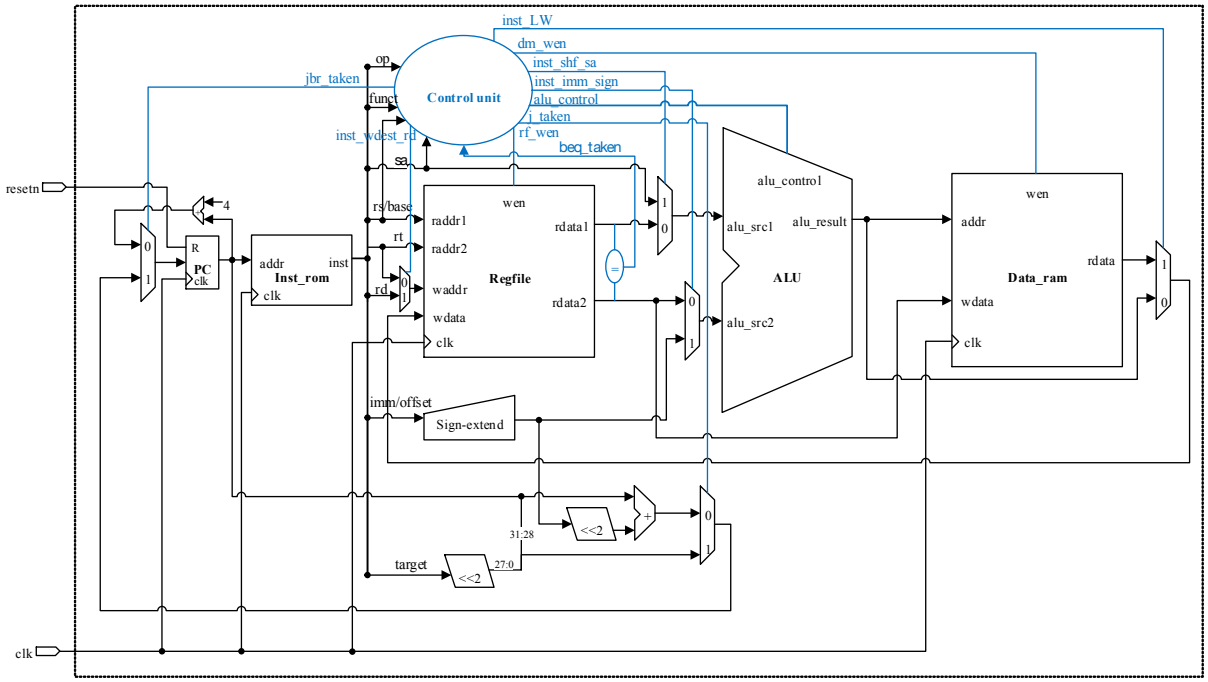
2）改进要求，针对目前 CPU 可运行的 R 型和 I 型 MIPS 指令，各补充一条新的指令，需要修改的 ALU 模块可参照实验四当时的 ALU 改进。改进时注意以下几点：1）MIPS 指令格式要使用规范格式；2）指令执行验证需要修改 inst_rom 中预存储的 16 进制指令数据；3）注意代码中单周期 CPU 模块（single_cycle_cpu)中实现主要功能使用的都是组合逻辑，改进过程中避免使用 alway(clk)这样的时序逻辑。

3）实验原理图使用实验指导书的图 7.3 即可，无需修改。

4）按实验报告模板要求完成实验报告，以附件形式提交。注意实验报告中要有介绍分析的内容，针对实验箱照片，要解释图中信息，是否验证成功。

3、实验原理图

单周期 CPU 的实现框图：



对实现框图进行简单的说明：

在这个图中，我们可以看到 CPU 的主要组成部分，包括指令内存、寄存器文件、算术逻辑单元（ALU）、数据内存等。这些部分通过各种控制信号和数据路径相互连接。

然后对单周期 CPU 的设计原理进行简单说明：

1.在每个周期开始时，CPU 从指令内存中的 PC（程序计数器）地址处获取指令。然后，PC 会自动增加 4（因为每个指令的大小是 4 字节），以便下一个周期获取下一条指令。

2 上一步获取到的指令被送入指令解码器进行解码，解码器会根据指令的类型和格式，生成相应的控制信号，以控制 CPU 的其他部分。

3 根据解码得到的控制信号，CPU 可能会从寄存器文件中读取数据，进行算术或逻辑运算，或者进行数据的读写操作。

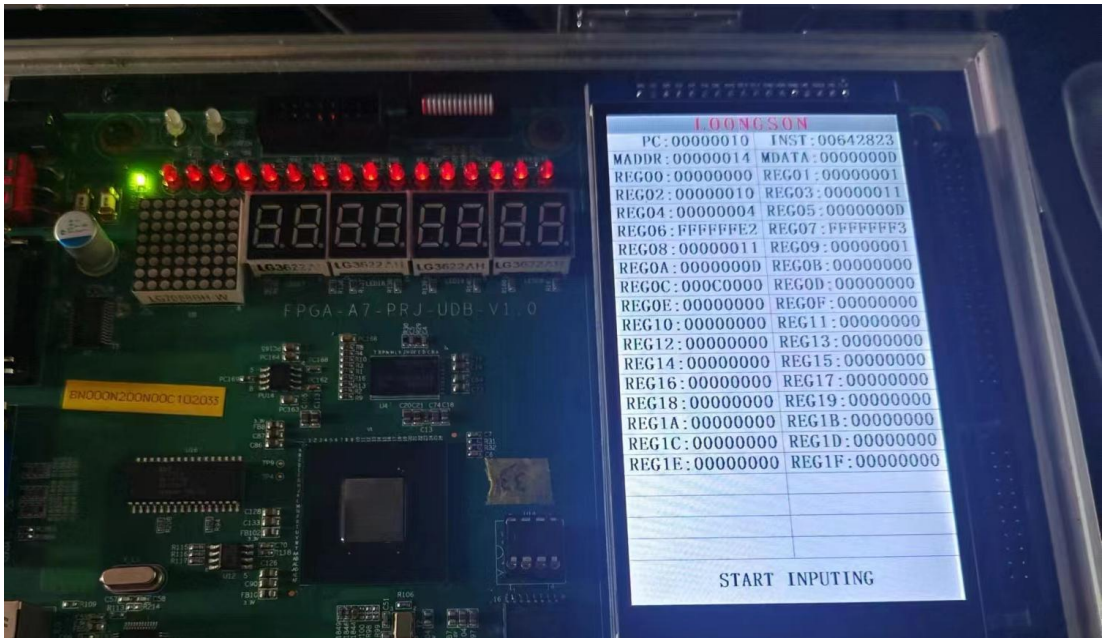
4 如果指令是加载或存储指令，CPU 会在这个阶段访问数据内存。加载指令会从内存中读取数据并写入寄存器，而存储指令会将寄存器中的数据写入内存。

5 在执行完毕后，结果会被写回到寄存器文件中。

4、实验步骤

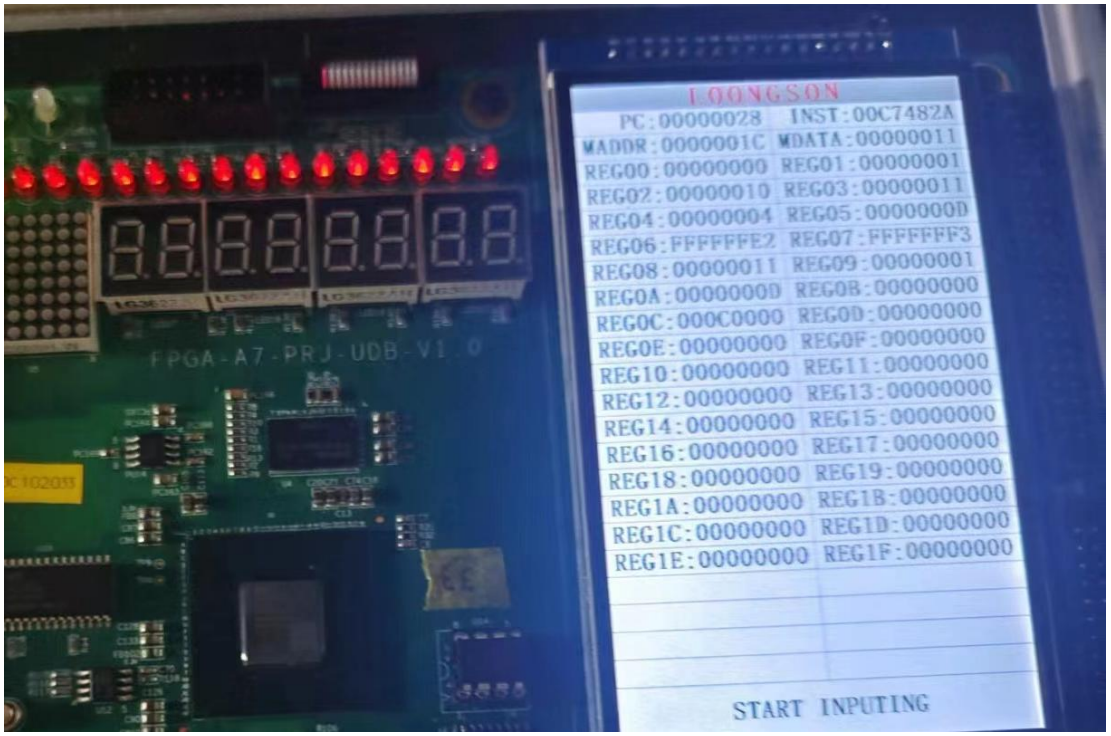
1）首先对原始的单周期 CPU 进行上箱验证这里的源码来自于 source_code 文件，编写好文件，导入 lcd 屏模块，然后分别依次跑综合、增强后确认无误，将电脑连接到实验箱后生成流文件到实验箱上，然后在实验箱上进行调试观察，验证是否完整的实现了目标功能，下列按照 PC 递增的顺序进行部分操作（关键结点）的结果验证：

I.PC 为 10H 时，对照手册上的功能表可知此时执行的功能时 subu 指令，命令执行的结果应该是$5 寄存器的值是 0000_000DH，功能正确执行，如下图：



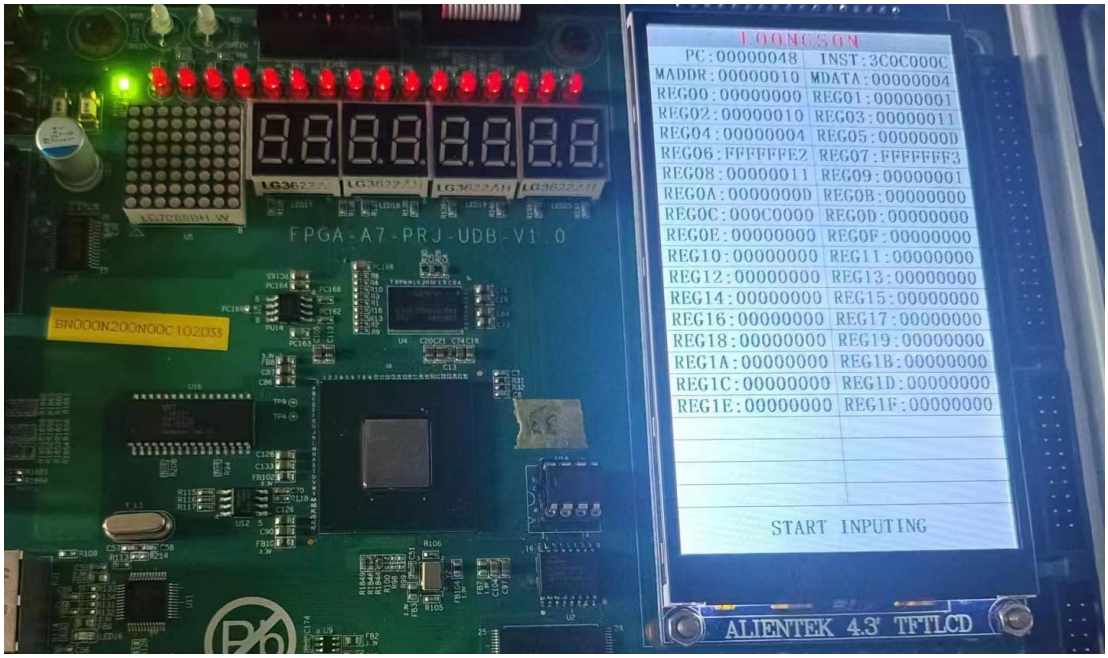
同时我又对前几个指令进行检查，分别发现$1 寄存器的值是 0000_0001H、$2 寄存器的值是 0000_0010H、$3 寄存器的值是 0000_0011H、$4 寄存器的值是 0000_0004H，这些均与手册给出的指令执行结果相同，说明功能正确执行了。

Ii.PC 为 28H，此时执行小于则置位 slt 指令，由于$6 寄存器的数据小于$7,所以应该对$9 寄存器置位，可以看到$9 的数值为 0000_0001H，功能正确执行，如下图：



同时对$6、$7、$8 寄存器进行检查，分别为 FFFF_FFE2H、FFFF_FFF3H、0000_0011H，结果正确，指令正确执行。

Iii.PC 为 48H，此时执行的指令为 lui $12 #12，即将$0C 寄存器的高 16 位置为立即数 12，可以看到下图中该寄存器的值为 000C_0000H，结果正确，功能正确执行，如下图：



同时对$10、$11 寄存器进行检查，分别为 0000_000DH、0000_0000H，结果正确，执行正确执行。

综上初始的单周期 CPU 验证结束，所有功能正确无误可以执行。下面开始进行按要求修改。

2）本次实验修改要求另外添加一条 R 型和一条 I 型 MIPS 指令，我这里选择了 R 型指令：同或(nxor)操作；I 型指令：立即数与(andi)操作。首先要对 ALU 模块进行修改，如下：（其中红色表示修改或者添加部分）

```
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_nxor;   //同或
wire alu_andi;   //立即数与
assign alu_add  = alu_control[13];
assign alu_sub  = alu_control[12];
assign alu_slt  = alu_control[ 11];
assign alu_sltu = alu_control[ 10];
assign alu_and  = alu_control[ 9];
assign alu_nor  = alu_control[ 8];
assign alu_or   = alu_control[ 7];
assign alu_xor  = alu_control[ 6];
assign alu_sll  = alu_control[ 5];
assign alu_srl  = alu_control[ 4];
assign alu_sra  = alu_control[ 3];
assign alu_lui  = alu_control[ 2];
assign alu_andi = alu_control[1];
assign alu_nxor = 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]nxor_result;
wire[31:0]andi_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 nxor_result=~xor_result;   //同或结果为异或取反
    assign andi_result=alu_src1 &alu_src2 ;//与立即数做与操作

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_nxor              ? nxor_result :
                                alu_andi              ? andi_result:
                                32'd0;
endmodule
```

3）然后继续对 single_circle_cpu 模块进行修改，添加两条指令的变量，命名分别为 inst_NXOR 和 INST_ANDI。同时需要两条新指令进行编码，这里要注意不要占用已经用过码序列，这里将同或运算的功能码设置为 011001；将立即数与的 OP 码设置成 111111，代码如下：

```
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_NXOR;//同或
    wire inst_ANDI;//立即数与
    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_NXOR     = op_zero & sa_zero       & (funct == 6'b011001);//  逻辑同或运算
    assign inst_ANDI     = (op == 6'b111111);//立即数与操作
    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
    // 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};
    //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;
    // 寄存器堆
    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)
    );
```

```
// 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_nxor;
wire inst_andi;
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= 1'b0;      // 暂未实现
assign inst_and = inst_AND;  // 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_nxor=inst_NXOR;//  逻辑同或
assign inst_andi=inst_ANDI;//立即数与
assign inst_sll = inst_SLL;  // 逻辑左移
assign inst_srl = inst_SRL;  // 逻辑右移
assign inst_sra = 1'b0;      // 暂未实现
assign inst_lui = inst_LUI;  // 立即数装载高位
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_NXOR;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW|inst_ANDI;
wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
wire [13:0] 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_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,
```

```
                                inst_andi,
                                inst_nxor};
```

4）对 inst_rom.v 模块进行修改，各补充上述的两条指令，这里的运算示例分别是对$6,$7 寄存器进行同或操作，将结果存到$8 寄存器中，和将$1 寄存器和 1 进行立即数与操作，将结果保存到$2 寄存器中，代码如下：

```
assign inst_rom[17] = 32'hAC040010; // 44H: sw       $4 ,#16($0) | Mem[0000_0010H] = 0000_0004H
    assign inst_rom[18] = 32'h3C0C000C; // 48H: lui     $12,#12        | [R12] = 000C_0000H
    assign inst_rom[19] = 32'h00E64019;// 4CH:同或 nxor     $8 ,$7,$6 修改寄存器 8 的数值
    assign inst_rom[20]=   32'hFC220001;// 50H:立即数与 andi $2,$1,#1   |修改寄存器 2 的值，应该还是 1，因为$1 里的数据和立即数 1 做与操作还是 1
    assign inst_rom[21] = 32'h08000000; // 54H: j        00H             | 跳转指令 00H
    //读指令,取 4 字节
    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];
            default: inst <= 32'd0;
        endcase
    end
endmodule
```

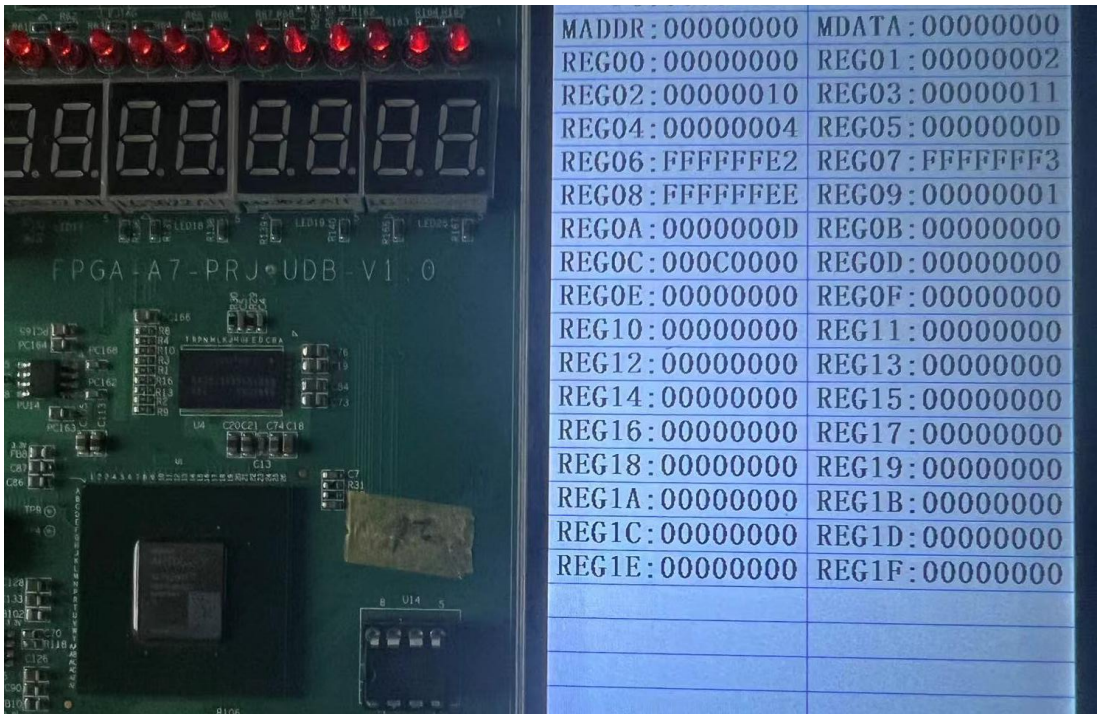
5）然后分别依次跑综合、增强后确认无误，将电脑连接到实验箱后生成流文件到实验箱

上，然后在实验箱上进行调试观察，验证是否完整的实现了目标功能。

5、实验结果分析

上箱结果验证：（要对按钮进行说明，这里只需要点击实验箱中间下面的按钮中的位置更下的按钮实现，PC 的递增即可）然后这里仅进行对新添加指令的功能验证，因为原始的功能已经在上文中进行了验证。

I. 进行同或操作的验证，此时 PC 为 50H，执行的操作是对$6，$7 寄存器进行同或操作，将结果存到$8 寄存器中，此时$6、$7 寄存器分别为 FFFF_FFE2H、FFFF_FFF3H，进行同或运算结果应该是 FFFF_FFEEH，结果正确，如下图；



II. 进行立即数与操作的验证：此时 PC 为 54H，执行的操作是将$1 寄存器和 1 进行立即数与操作，将结果保存到$2 寄存器中，此时$1 寄存器的是 0000_0001H，与 1 与运算之后，结果应该是 0000_0001H，结果正确，指令正确执行，如下图：

综上成功的完成了所有实验要求，并成功添加了一条 R 型指令和一条 I 型指令。

6、总结感想

这次实验可以说是所有实验中最难的一个实验，因为设计的模块很多，而且细节处理会有很多，所以在整个实验过程中开始的几次烧录都失败了，但最后不断检查不断找错，最后成功实现了目标功能。

下面是我的心得总结：

1　这次实验让我对 MIPS 指令的结构和功能有了深入的理解。从以前只知道理论的层面，到现在能够归类和编码各种常用指令，我感觉我对 MIPS 指令集有了新的理解。

2　我也对 MIPS 的处理器结构有了更深入的了解。我研究了它是怎么工作的，这让我能更好理解这个体系。

3　单周期 CPU 的设计和原理我完全理解并且可以进行一些简单应用了。我实现了单独设计一个单周期 CPU，带来了成就感。

4　我也在实验中进一步提升了我的 verilog 编程技能。通过设计电路，我能更好地理解电路的运作方式，这也让我对编程有了新的理解和感触。

总的来说，这次实验让我收获颇丰。我不仅理解了 MIPS 指令集和单周期 CPU 的设计原理，还提升了我在 verilog 编程方面的技能。我很期待未来可以继续深入学习，例如像下一章的多周期 CPU 的实现等等。