组成原理实验课程第_6_次实验报告

实验名称	单周期 CPU 实现			班级	李涛老师
学生姓名	孙蕗	学号	2112060	指导老师	董前琨
实验地点	A308		实验时间	2023.6.6	

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、 实验设备

（1） 装有 Xilinx Vivado 的计算机一台。

（2） LS-CPU-EXB-002 教学系统实验箱一套。

3、 实验任务

（1） 学习 MIPS 指令集，深入理解常用指令的功能和编码，并进行归纳确定处理器各部件的控制码，比如使用何种 ALU 运算，是否写寄存器堆等。

（2） 确定自己本次实验中的准备实现的 MIPS 指令，要求至少实现一条 load 指令、一条 store 指令、10 条基础运算指令、一条跳转指令。其中基础运算指令最好包含多种类型的操作，必须包含一条加法和一条减法指令。不考虑指令可能产生异常的情况。单周期 CPU 的实验重点是搭建出一个 CPU 架构，为避免被繁琐的指令所困惑，建议在单周期 CPU 实验中只实现十几条指令。

（3） 对准备实现的指令进行分析，完成表 7.1 的填写。

表 7.1 mips 基础指令特性归纳表

指令类型	汇编指令	指令码	源操作数 1	源操作数 2	目的寄存器	功能描述
R 型指令	addu rd，rs，rt	000000|rs|rt|rd|00000|100001	[rs]	[rt]	rd	GPR[rd]= GPR [rs]+ GPR [rt]
I 型指令	addiu rt,rs,imm	001001|rs|rt|imm	[rs]	sign_ext(imm)	rt	GPR [rt]= GPR [rs]+sign_ext(imm)
J 型指令	j target	000010|target	PC	target		跳转，PC={PC[31:28],target,2'b00}

注：GPR 表示通用寄存器，[rs]表示寄存器 rs 里存储的值，PC 表示程序计数器；imm 为 16 位立即数，sign_ext(imm)表示对其进行符号扩展；target 为 26 位立即数。

（4） 自行设计本次实验的方案，画出结构框图，大致结构框图如图 7.1。图 7.1 中粗线表示接口位数和种类不定，需要在自己的结构框图中详细给出。从图 7.1 中可以看出，本

次实验是需要用到之前实验的结果的，比如 ALU 模块、寄存器堆模块、指令 ROM 模块和数据 RAM 模块，其中 ROM 和 RAM 要使用自行搭建的异步存储器。

单周期 CPU 是指一条指令的所有操作在一个时钟周期内执行完。设计中所有寄存器和存储器都是异步读同步写的，即读出数据不需要时钟控制，但写入数据需时钟控制。

故单周期 CPU 的运作即：在一个时钟周期内，根据 PC 值从指令 ROM 中读出相应的指令，将指令译码后从寄存器堆中读出需要的操作数，送往 ALU 模块，ALU 模块运算得到结果。

如果是 store 指令，则 ALU 运算结果为数据存储的地址，就向数据 RAM 发出写请求，在下一个时钟上升沿真正写入到数据存储器。

如果是 load 指令，则 ALU 运算结果为数据存储的地址，根据该值从数据存 RAM 中读出数据，送往寄存器堆根据目的寄存器发出写请求，在下一个时钟上升沿真正写入到寄存器堆中。

如果非 load/store 操作，若有写寄存器堆的操作，则直接将 ALU 运算结果送往寄存器堆根据目的寄存器发出写请求，在下一个时钟上升沿真正写入到寄存器堆中。

如果是分支跳转指令，则是需要将结果写入到 pc 寄存器中的。

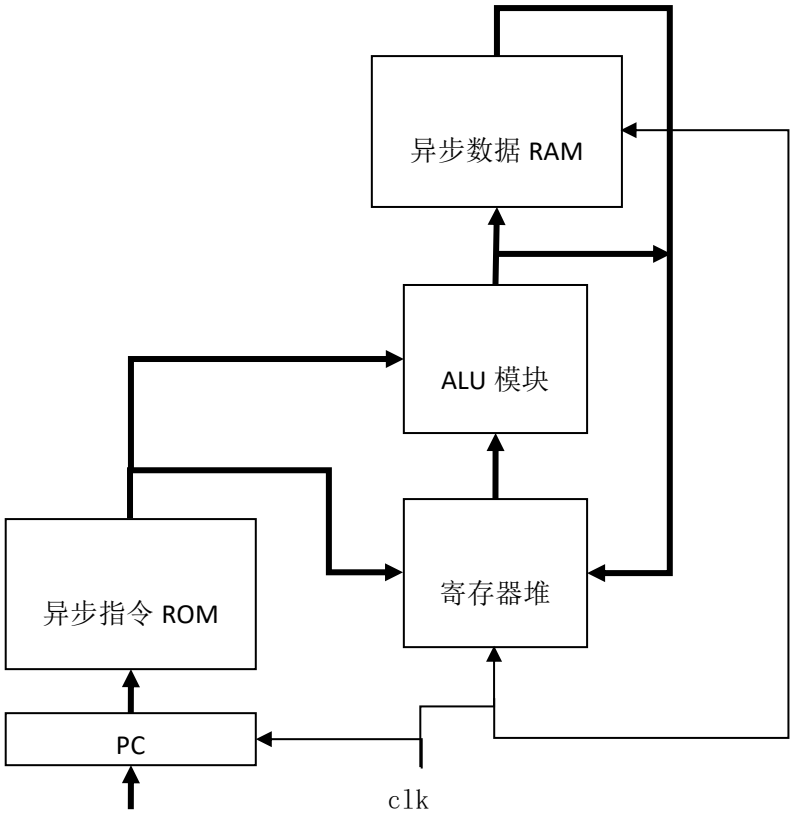


图 7.1　单周期 CPU 的大致框图

（5） 根据设计的实验方案，使用 verilog 编写相应代码。

（6） 依据自己设计中实现的指令，编写一段不少于 20 行的汇编程序，力求验证到所有实现的指令。该段汇编程序是需要内嵌到自行搭建的异步指令 ROM 中的。完成表 7.2 的填写。

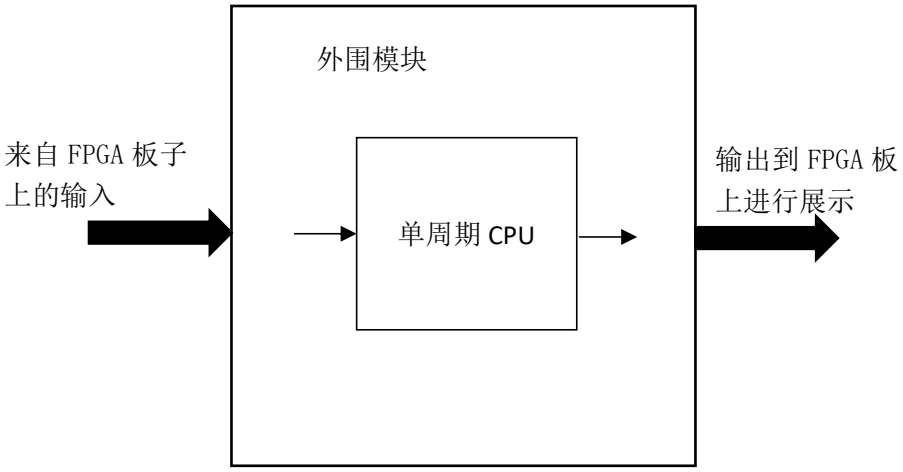
表 7.2　测试所用汇编程序详述

指令地址	汇编指令	结 果描 述	机器指令的机器码	
			16 进制	二进制
00H	addiu ,$1, $0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001__0000_0000_0000_0001

（7） 对编写的代码进行仿真，得到正确的波形图。

（8） 将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，见图 7.2。外围模块中需调用封装好的 LCD 触摸屏模块，观察单周期 CPU 的内部状态，比如 32 个寄存器的值，PC 的值等。并且需要利用触摸功能输入特定数据 RAM 地址，从该 RAM 的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果。通过这些手段，可以在板上充分验证 CPU 的正确性。

图 7.2　单周期 CPU 设计实验的顶层模块大致框图



（9） 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。

注意：

① ：MIPS 架构中有延迟槽的设定，其本意是加快流水 CPU 的执行速度，故在单周期 CPU 中该设定毫无意义，反而带来了 CPU 实现上的麻烦，故建议在单周期 CPU 中不考虑延迟槽技术。

② ：MIPS 架构中分支和跳转指令参与计算的 PC 值均为延迟槽指令对应的 PC(即分支跳转指令的 PC+4),而由于单周期不考虑延迟槽，故在实验中分支跳转指令参与计算使用本指令的 PC 值，故跳转的偏移量设置需要注意下。比如一条指令“beq,r0,r0,#2”在不考虑延迟槽的单周期 CPU 中，其跳转的目标地址为 beq 指令后面的第 2 条。而在考虑延迟槽的流水 CPU 中，其跳转的目标地址为 beq 指令后面的第 3 条（即延迟槽指令后面的第 2 条）。这里需要理解清楚。

4、 实验内容说明

（1） 做好预习：

1） 熟知 MIPS 指令类型，深入理解常用指令的功能和编码；

2） 归纳常用的 MIPS 指令，确定自己准备实现的 MIPS 指令；

3） 对准备实现的指令进行分析，完成表 7.1 的填写；

4） 设计本次实验的方案，画出实验方案的设计框图，即补充完善图 7.1；

5） 如果对 FPGA 板了解的话，可确定设计中与 FPGA 板上交互的接口，画出包含外围模块的整体设计框图，即补充完善图 7.2；

6） 依据自己设计中实现的指令，编写一段不少于 20 行的汇编程序，要求包含所有实现的指令，完成表 7.2 的填写。

（2） 实验实施：

1） 确认单周期 CPU 的设计框图的正确性；

2） 编写 verilog 代码，将表 7.2 中自己编写的汇编程序翻译为二进制，内嵌到指令 ROM 中；

3） 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；

4） 完成调用单周期 CPU 的外围模块的设计，并编写代码；

5） 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。

（3） 实验检查：

1） 完成上板验证后，让指导老师或助教进行检查，进行现场演示。先解读表 7.2 中自己编写的汇编程序，然后采用手动输入时钟，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。

（4） 实验报告的撰写：

实验结束后，需按照规定的格式完成实验报告的撰写。

5、 实验内容改进说明

请根据实验指导书实验六相关部分完成单周期 CPU 设计实验，在原始实验基础上进行改进，请按照如下要求完成实验报告：

（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。

6、 实验原理图

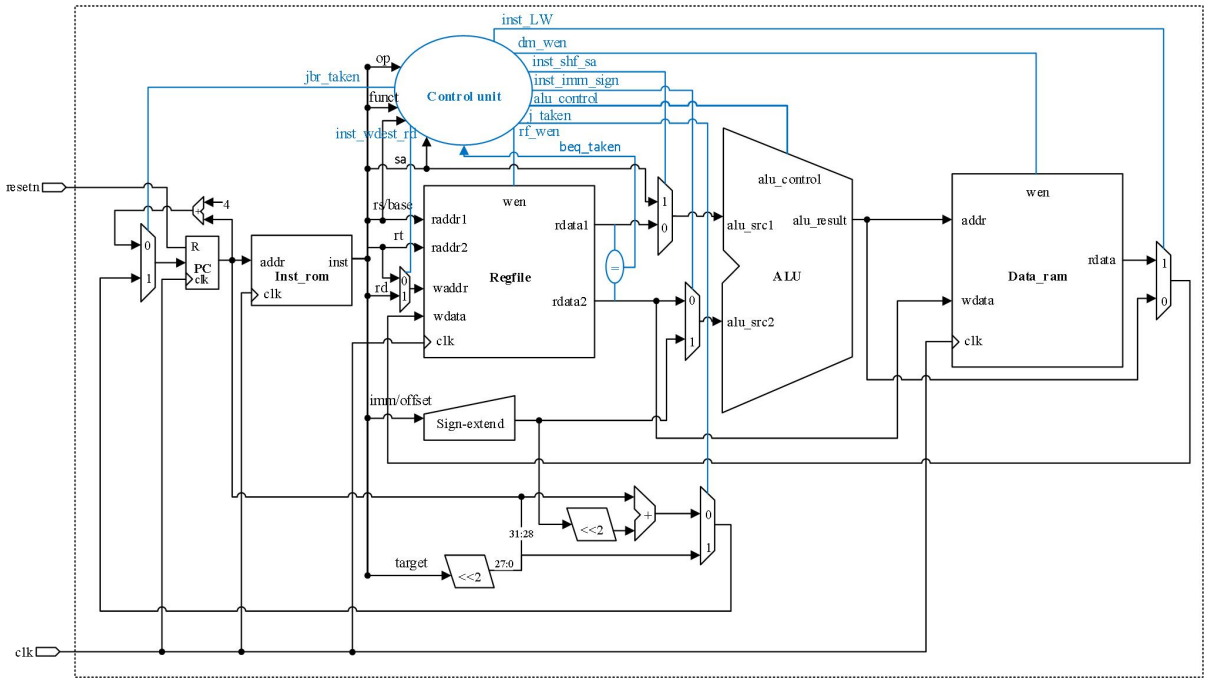


图 7.3　单周期 CPU 的实现框图

表 7.3 为单周期 CPU 实现的指令集。表 7.4 为这些指令的解析。

表 7.3　单周期 CPU 实现的指令

指令名称	汇编指令	功能描述
无符号加法	addu rd,rs,rt	GPR[rd] = GPR[rs] + GPR[rt]
无符号减法	subu rd,rs,rt	GPR[rd] = GPR[rs] - GPR[rt]
有符号比较，小于置位	slt rd,rs,rt	GPR[rd] = (sign(GPR[rs]) < sign(GPR[rt]))
按位与	and rd,rs,rt	GPR[rd] = GPR[rs] & GPR[rt]
按位或非	nor rd,rs,rt	GPR[rd] = ~(GPR[rs] | GPR[rt])
按位或	or rd,rs,rt	GPR[rd] = GPR[rs] | GPR[rt]
按位异或	xor rd,rs,rt	GPR[rd] = GPR[rs] ^ GPR[rt]
逻辑左移	sll rd,rt,shf	GPR[rd] = zero(GPR[rt]) << shf
逻辑右移	srl rd,rt,shf	GPR[rd] = zero(GPR[rt]) >> shf
立即数无符号加法	addiu rt,rs,imm	GPR[rt] = GPR[rs] + sign_ext(imm)
立即数装载高位	lui rt,imm	GPR[rt] = {imm, 16'd0}
从内存装载字	lw rt,offset(base)	GPR[rt] = Mem[GPR[base] + sign_ext (offset)]
向内存存储字	sw rt,offset(base)	Mem[GPR[base]+sign_ext (offset)] = GPR[rt]
判断相等跳转	beq rs,rt,offset	if GPR[rs] = GPR[rt] then PC=PC+ sign_ext (offset)<<2
判断不等跳转	bne rs,rt,offset	if GPR[rs] ≠ GPR[rt] then PC=PC+ sign_ext (offset)<<2
直接跳转	j target	PC = {PC[31:28], target<<2}

指令类型	汇编指令	指令码	源操作数 1	源操作数 2	源操作数 3	目的寄存器	功能描述
R 型指令	addu rd,rs,rt	000000|rs|rt|rd|00000|100001	[rs]	[rt]		rd	GPR[rd]=GPR[rs]+GPR[rt]
	subu rd,rs,rt	000000|rs|rt|rd|00000|100011	[rs]	[rt]		rd	GPR[rd]=GPR[rs]-GPR[rt]
	slt rd,rs,rt	000000|rs|rt|rd|00000|101010	[rs]	[rt]		rd	GPR[rd]=(sign(GPR[rs])<sign(GPR[rt]))
	and rd,rs,rt	000000|rs|rt|rd|00000|100100	[rs]	[rt]		rd	GPR[rd]=GPR[rs]&GPR[rt]
	nor rd,rs,rt	000000|rs|rt|rd|00000|100111	[rs]	[rt]		rd	GPR[rd]=~(GPR[rs]|GPR[rt])
	or rd,rs,rt	000000|rs|rt|rd|00000|100101	[rs]	[rt]		rd	GPR[rd]=GPR[rs]|GPR[rt]
	xor rd,rs,rt	000000|rs|rt|rd|00000|100110	[rs]	[rt]		rd	GPR[rd]=GPR[rs]^GPR[rt]
	sll rd,rt,shf	000000|00000|rt|rd|shf|000000		[rt]		rd	GPR[rd]=zero(GPR[rt])<<shf
	srl rd,rt,shf	000000|00000|rt|rd|shf|000010		[rt]		rd	GPR[rd]=zero(GPR[rt])>>shf
I 型指令	addiu rt,rs,imm	001001|rs|rt|imm	[rs]	sign_ext(imm)		rt	GPR[rt]=GPR[rs]+ sign_ext (imm)
	beq rs,rt,offset	000100|rs|rt|offset	[rs]	[rt]			if GPR[rs]=GPR[rt] then PC=PC+ sign_ext (offset)<<2
	bne rs,rt,offset	000101|rs|rt|offset	[rs]	[rt]			if GPR[rs]≠GPR[rt] then PC=PC+ sign_ext (offset)<<2
	lw rt,offset(b)	100011|b|rt|offset	[b]	sign_ext (offset		rt	GPR[rt]=Mem[GPR[b]+sign_ext(offset)]

				)			
	sw rt,offset(b)	101011|b|rt|offset	[b]	sign_ext(offset)	[rt]		Mem[GPR[b]+sign_ext(offset)]=GPR[rt]
	lui rt,imm	001111|00000|rt|imm		{imm, 16'd0}		rt	GPR[rt]= {imm, 16'd0}
J型指令	j target	000010|target					PC={PC[31:28],target<<2}

表 7.4　单周期 CPU 实现的 mips 指令特性归纳

实验顶层模块框图

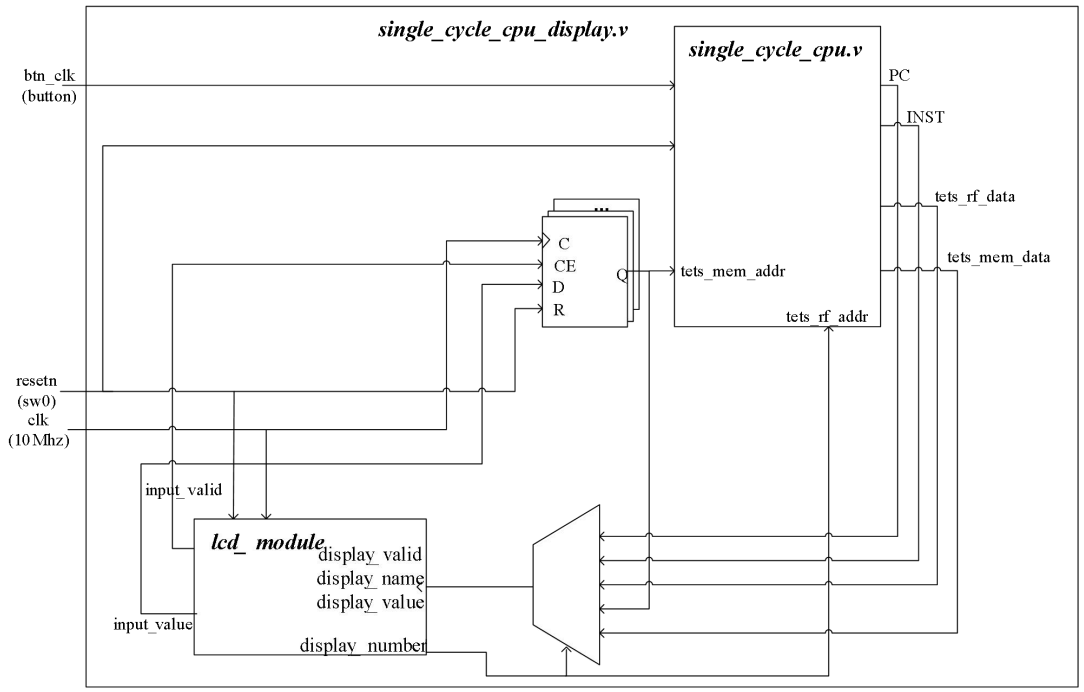


图 7.4　单周期 CPU 参考设计的顶层模块框图

7、 实验步骤

R 型指令添加算术右移功能，I 型指令添加立即数或功能

（1） 修改代码 single_cycle_cpu.v

```
`timescale 1ns / 1ps
//*************************************************************************
//   > 文件名: single_cycle_cpu.v
//   > 描述   :单周期 CPU 模块，共实现 16 条指令
//   >          指令 rom 和数据 ram 均采用异步读数据，以便单周期 CPU 好实现
//   > 作者   : LOONGSON
```

```
//   > 日期  : 2016-04-14
//#################################################################################
`define STARTADDR 32'd0  // 程序起始地址
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
    );

//---------------------------------{                                    取
指}begin-------------------------------------//
    reg  [31:0] pc;
    wire [31:0] next_pc;
    wire [31:0] seq_pc;
    wire [31:0] jbr_target;
    wire jbr_taken;

    // 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 <= `STARTADDR; // 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;
//---------------------------------{                                    取
指}end-------------------------------------//

//---------------------------------{                                    译
码}begin-----------------------------------//
    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];  // 操作码
    assign rs     = inst[25:21];  // 源操作数 1
    assign rt     = inst[20:16];  // 源操作数 2
    assign rd     = inst[15:11];  // 目标操作数
    assign sa     = inst[10:6];   // 特殊域，可能存放偏移量
    assign funct  = inst[5:0];    // 功能码
    assign imm    = inst[15:0];   // 立即数
    assign offset = inst[15:0];   // 地址偏移量
    assign target = inst[25:0];   // 目标地址

    wire op_zero;  // 操作码全 0
    wire sa_zero;  // sa 域全 0
    assign op_zero = ~(|op);
    assign sa_zero = ~(|sa);

    // 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;

    //R 型指令，算术右移
    wire inst_SRA;

    //I 型指令，立即数或
```

```
    wire inst_ORI;

    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

    //R 型指令，算术右移
    assign inst_SRA = op_zero & (rs == 5'd0) & (funct == 6'b000011); // 算术
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

   //I 型指令，立即数或
    assign inst_ORI = (op == 6'b001101);                    //立即数或

    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;

     //R 型指令，算术右移 sra
     wire inst_sra,inst_lui;

    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，立即数与(I)
    assign inst_nor = inst_NOR;  // 逻辑或非
    assign inst_or  = inst_OR|inst_ORI;   // 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

    //R 型指令，算术右移
    assign inst_sra = inst_SRA;       // 算术右移

    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 = (inst_SRA) ? $signed({16'b0, imm[15:0]}) : {{16{imm[15]}},
imm}; // 添加算术右移的符号扩展。使用三元条件运算符来判断是否是算术右移指令
(inst_SRA)。如果是算术右移指令，则将立即数 imm 进行符号扩展，使用
$signed({16'b0, imm[15:0]}) 表示将其转换为有符号数。如果不是算术右移指令，则使
用原来的符号扩展方式 {{16{imm[15]}}, imm}
    assign inst_shf_sa   = inst_SLL | inst_SRL |inst_SRA; // 包括逻辑左移、逻
辑右移和算术右移指令
    assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;//立即数
或不需要符号扩展，是零扩展的

    wire [31:0] alu_operand1;
    wire [31:0] alu_operand2;
    wire [11:0] 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};
    //---------------------------------{                                    译
    码}end-------------------------------------//

    //---------------------------------{                                    执
    行}begin-----------------------------------//
        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
        );
    //---------------------------------{                                    执
    行}end-------------------------------------//

    //---------------------------------{                                    访
    存}begin-----------------------------------//
        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     )
    );
//---------------------------------{                                    访
存}end-------------------------------------//

//---------------------------------{                                    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 | inst_ORI;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                         | inst_OR    | inst_XOR   | inst_SLL | inst_SRL|
inst_SRA;
    // 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
//---------------------------------{                                    写
回}end-------------------------------------//
```

（2） 修改代码 inst_rom.v

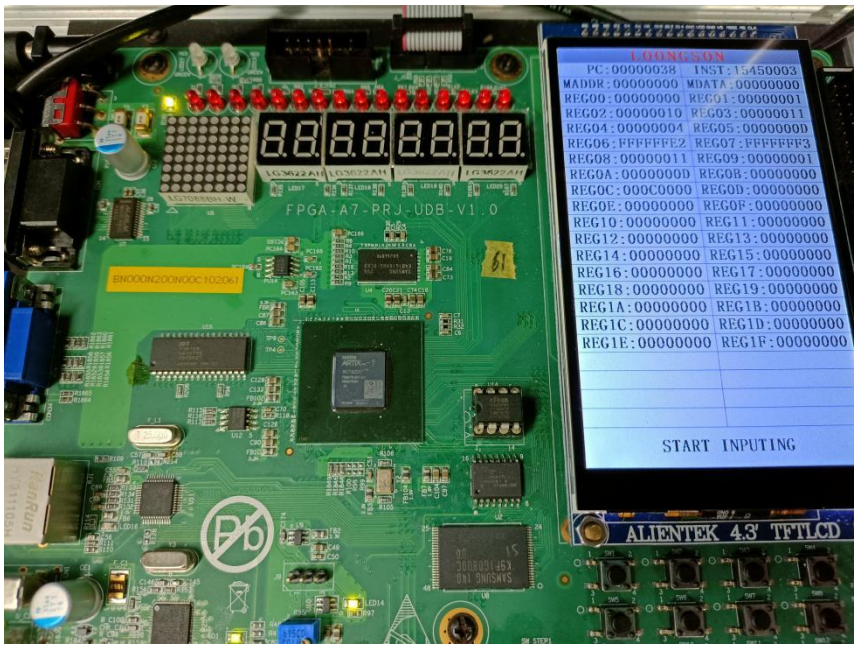
```
`timescale 1ns / 1ps
//*************************************************************************************
//   > 文件名: inst_rom.v
//   > 描述  ：异步指令存储器模块，采用寄存器搭建而成，类似寄存器堆
//   >         内嵌好指令，只读，异步读
//   > 作者  : LOONGSON
//   > 日期  : 2016-04-14
//*************************************************************************************
module inst_rom(
    input      [4 :0] addr, // 指令地址
    output reg [31:0] inst       // 指令
    );

    wire  [31:0]  inst_rom[21:0];     //  指 令 存 储 器 ， 字 节 地 址
7'b000_0000~7'b111_1111
    //------------- 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果
-----//
    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'h15450003; // 38H: bne   $10,$5,#3   | 不跳转
    assign inst_rom[15] = 32'h00415824; // 3CH: and    $11,$2,$1    | $11 =
0000_0000H
    assign inst_rom[16] = 32'hAC0B001C; // 40H: sw      $11,#28($0)  |
Men[0000_001CH] = 0000_0000H
    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'h000A28C3; // 4CH:  sra    $5 ,$10,#3   | $5 =
0000_0001H (算术右移) 将寄存器 $10 中的值右移 3 位，结果存储在寄存器 $5 中
000000 00000 01010 00101 00011 000011。

    assign inst_rom[20] = 32'h34E600F0; // 50H: ori  $6 ,$7,#240 | $6 = FFFF_FFE2H
andi $6, $7, #240，将寄存器 $7 的值与立即数 240 进行位或运算，结果存储在寄存
器 $6 中。001101 00111 00110 0000000011110000

    assign inst_rom[21] = 32'h08000000; // 54H: j     00H          | 跳转指令
00H
    //读指令,取 4 字节
    always @(*)
```

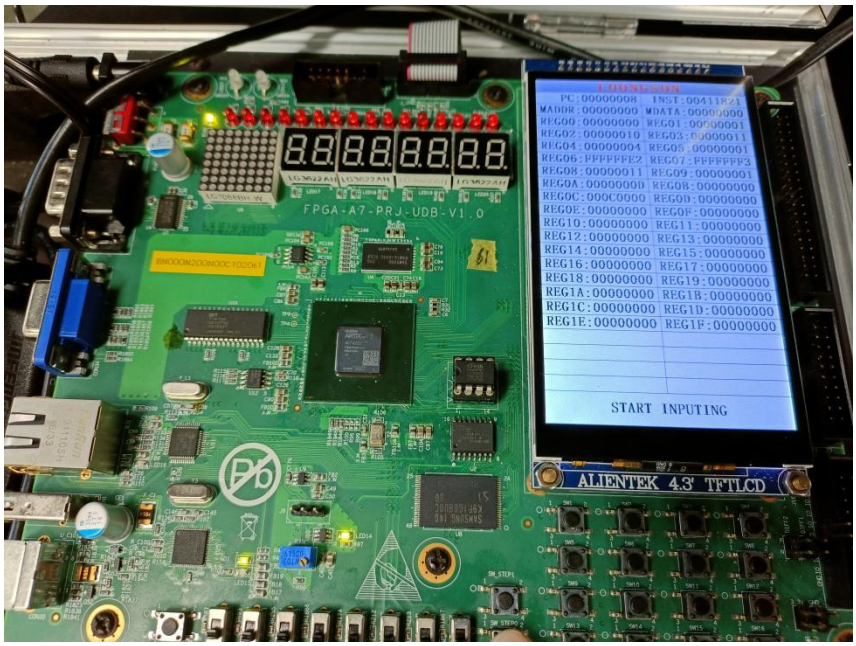
```
    begin
        case (addr)
            5'd0 : inst <= inst_rom[0 ];
            5'd1 : inst <= inst_rom[1 ];
            5'd2 : inst <= inst_rom[2 ];
            5'd3 : inst <= inst_rom[3 ];
            5'd4 : inst <= inst_rom[4 ];
            5'd5 : inst <= inst_rom[5 ];
            5'd6 : inst <= 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
```

8、 实验结果分析

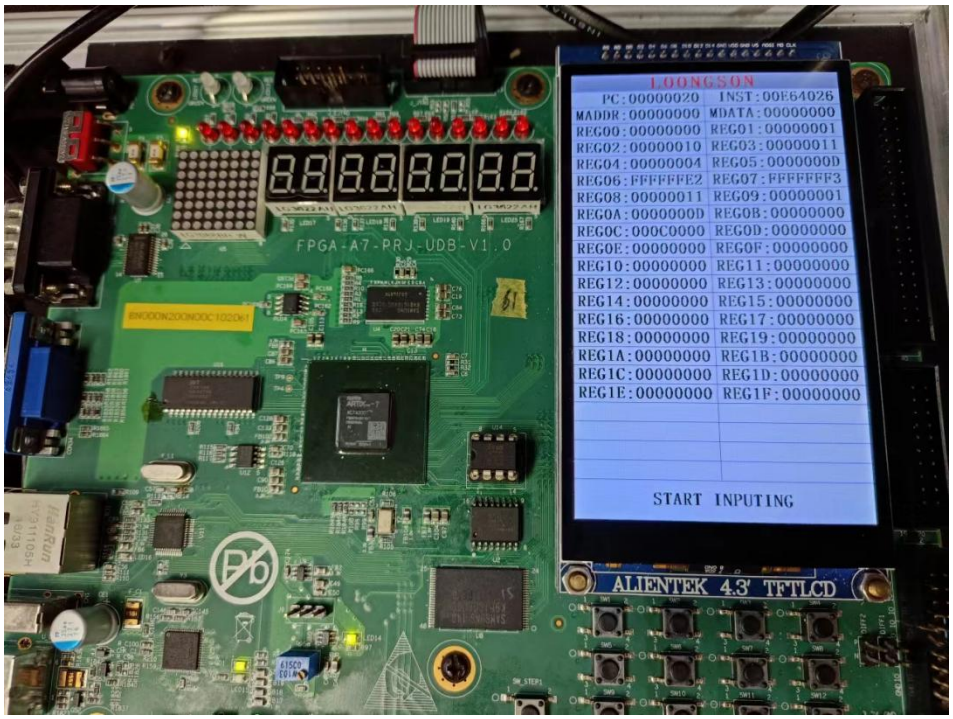
（1） 验证修改前



A） 下一条指令 PC=38H，则现在程序执行的是
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw $10,#19($1) | $10 = 0000_000DH
寄存器 10，也就是 REG0A 应该显示的结果是 0000000D，结果是符合的。



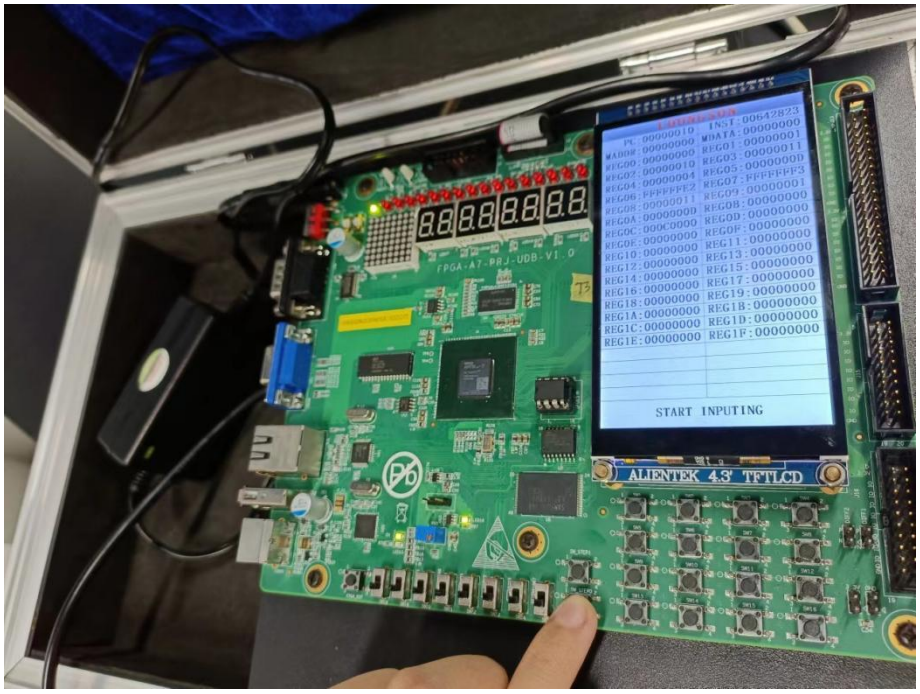
B) 下一条指令 PC=08H，则现在程序执行的是
assign inst_rom[1] = 32'h00011100; // 04H: sll $2 ,$1,#4 | $2 = 0000_0010H
寄存器 2，也就是 REG02 应该显示的结果是 00000010，结果是符合的。



C) 下一条指令 PC=20H，则现在程序执行的是

assign inst_rom[7] = 32'h00C33825; // 1CH: or $7 ,$6,$3 | $7 = FFFF_FFF3H

寄存器 7，也就是 REG07 应该显示的结果是 FFFFFFF3，结果是符合的。

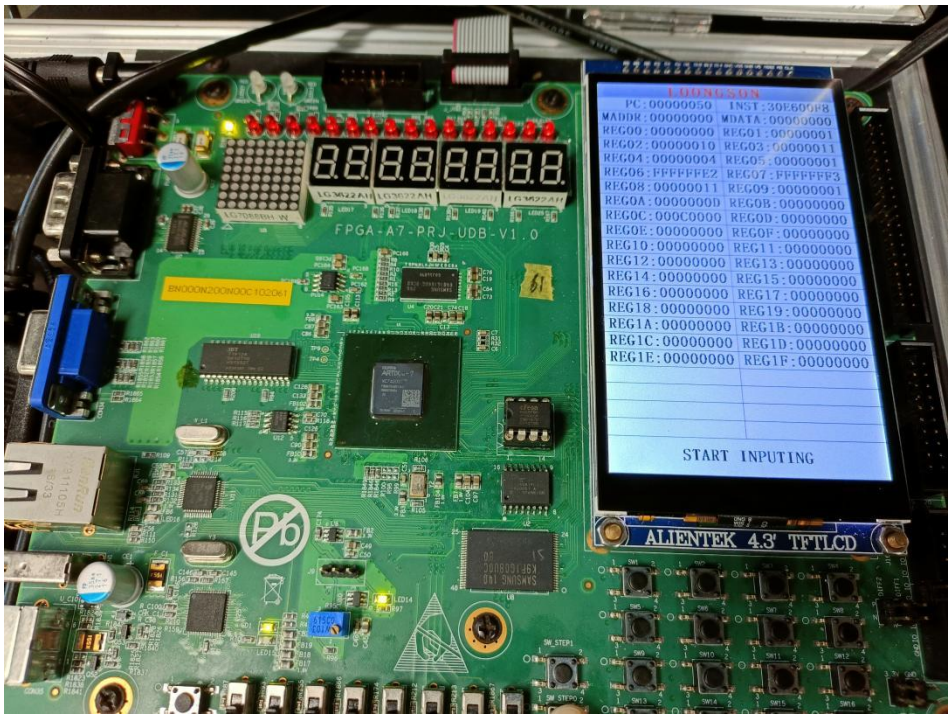


D) 下一条指令 PC=10H，则现在程序执行的是

assign inst_rom[2] = 32'h00411821; // 08H: addu $3 ,$2,$1 | $3 = 0000_0011H

寄存器 3，也就是 REG03 应该显示的结果是 00000011，结果是符合的。

（2） 添加算术右移后



A） 下一条指令 PC=50H，则现在程序执行的是

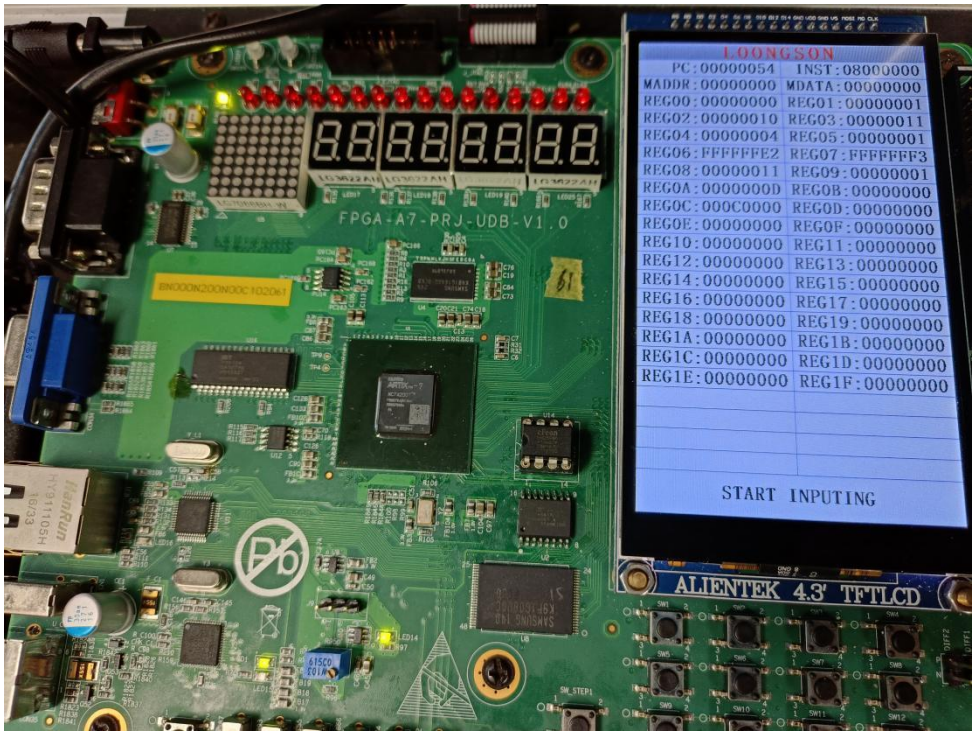
assign inst_rom[19] = 32'h000A28C3; // 4CH: sra $5 ,$10,#3 | $5 = 0000_0001H

将寄存器 $10 中的值右移 3 位，结果存储在寄存器 $5 中 000000 00000 01010 00101 00011 000011。

$10 = 0000_000DH，转换成二进制后为 0000 0000 0000 0000 0000 0000 0000 1011。向右算术右移 3 位，变成 0000 0000 0000 0000 0000 0000 0000 0001。转换成 16 进制后为 00000001。

寄存器 5，也就是 REG05 应该显示的结果是 00000001，结果是符合的。

（3） 添加立即数或后



B） 下一条指令 PC=54H，则现在程序执行的是

assign inst_rom[20] = 32'h34E600F0; // 50H: ori $6 ,$7,#240 | $6 = FFFF_FFE2H

将寄存器 $7 的值与立即数 240 进行位或运算，结果存储在寄存器 $6 中。001101 00111 00110 0000000011110000

$7 的值为 1111 1111 1111 1111 1111 1111 1111 0011，240 转换成 0000 0000 0000 0000 0000 0000 1111 0000，位或的结果是 1111 1111 1111 1111 1111 1111 1111 0011，转换 16 进制为 FFFFFFE2。

寄存器 6，也就是 REG06 应该显示的结果是 FFFFFFE2，结果是符合的。

9、 总结感想

在完成单周期 CPU 设计实验的过程中，我对计算机体系结构和指令执行原理有了更深入的了解。通过对原始实验的改进，我不仅实现了基本的 R 型和 I 型 MIPS 指令的执行，还各自添加了一条新的指令。对于 R 型指令，添加了算术右移功能，对于 I 型指令，我添加了立即数或功能，使得 CPU 能够执行更多种类的指令。

在修改指令执行验证时，我根据新指令的格式修改了 inst_rom 中预存储的 16 进制指令数据。这样，在验证时就能够运行一系列包含新指令的程序，并通过实验箱拍照对比各个寄存器中的数据是否正确，从而验证指令的执行结果。

通过这次实验，我不仅加深了对单周期 CPU 设计的理解，还学会了如何修改和扩展现有的 CPU 设计以支持新的指令。我认识到计算机体系结构的设计需要考虑到指令的种类和功能，以及硬件实现的复杂性和效率。同时，我也体会到了实验过程中的调试和验证的重要性，只有通过实际验证，才能确保设计的正确性和可靠性。