**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。

1. **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。

1. **实验原理图**



1. **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 :

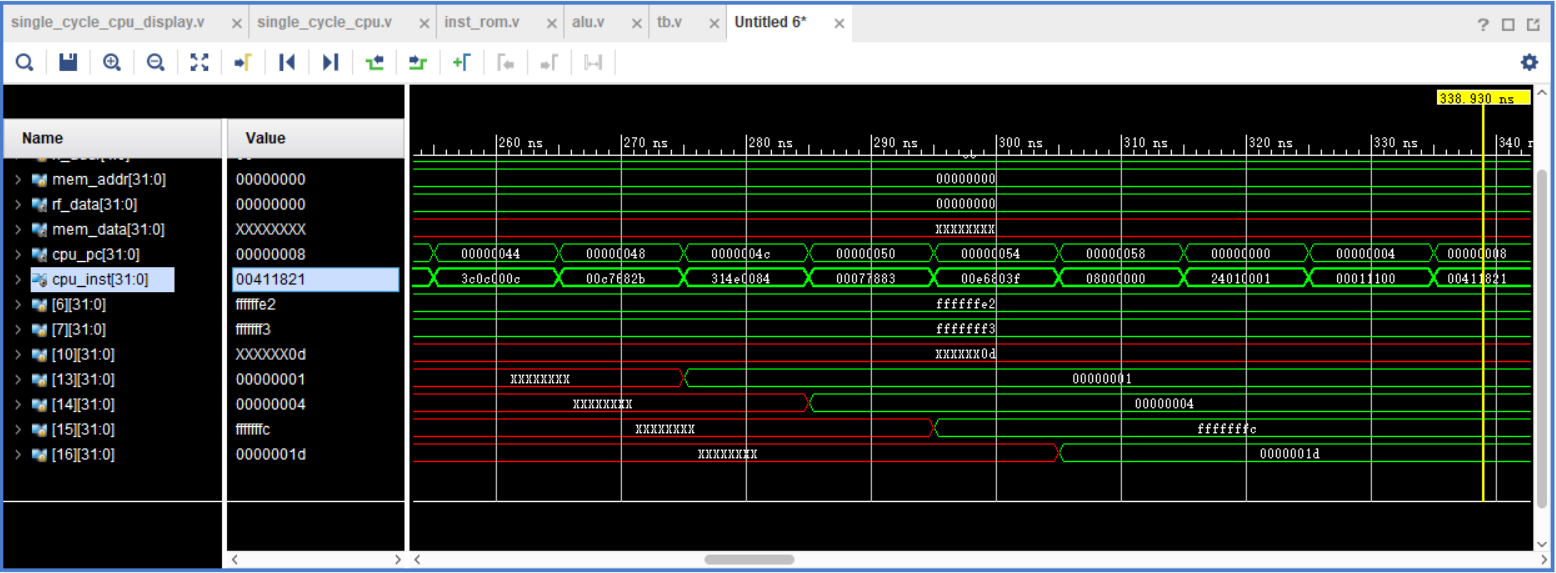
alu\_nand ? nand\_result://输出

32'd0;

endmodule

1. **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。

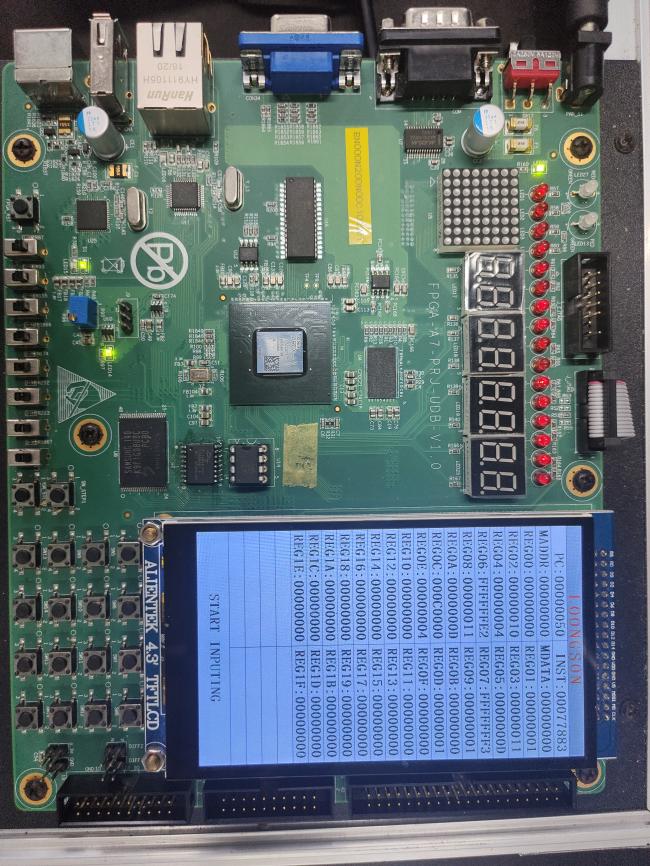
b.执行0x48的语句0x00C7682B



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。

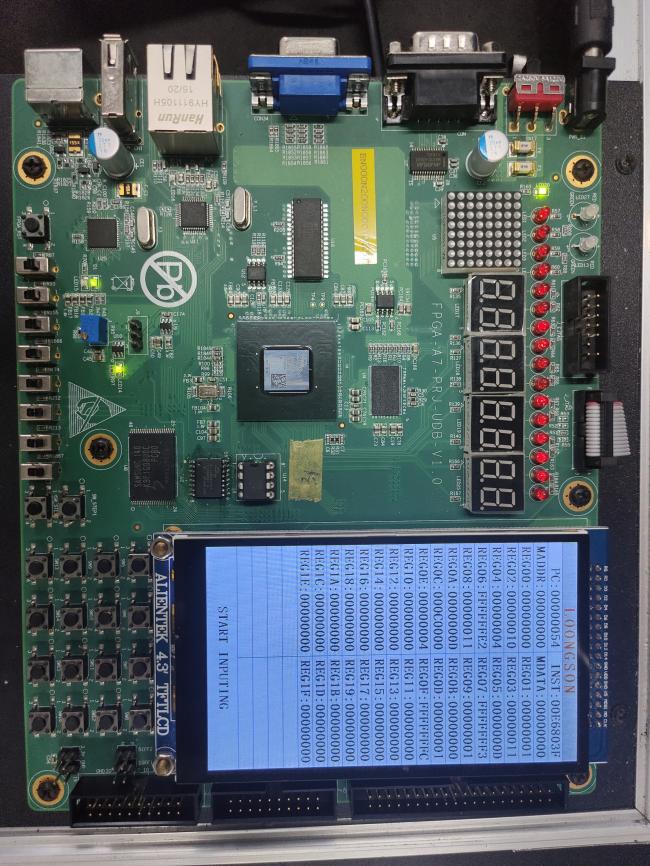
c.执行0x4C的语句0x314e0084



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。

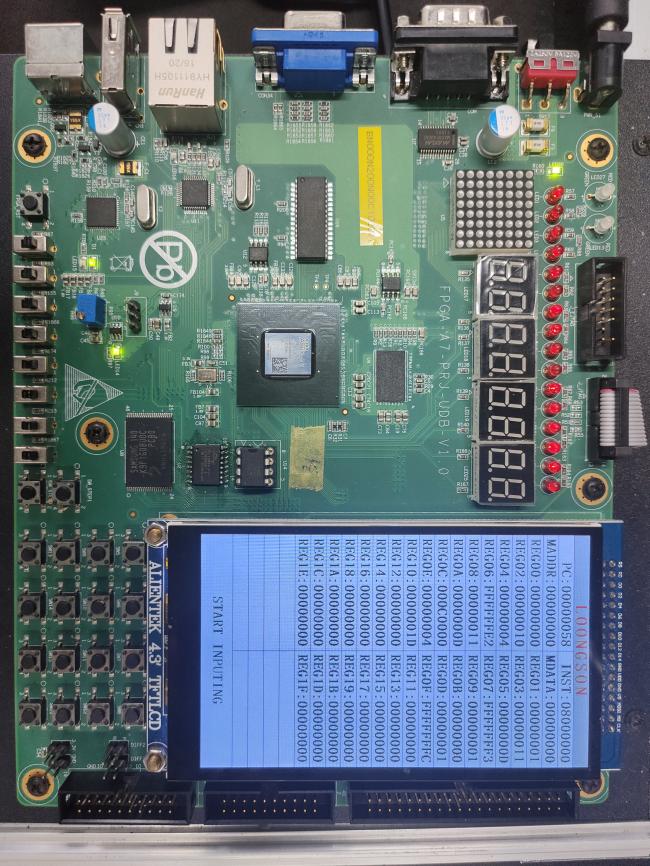
d.执行0x50的语句0x00077883



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。

1. **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！