**组成原理实验课程第 六 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 | 张洋 | 学号 | 2111460 | 指导老师 | 董前琨老师 |
| 实验地点 | 津南实验楼A306 | | 实验时间 | 5月30日 18：30 – 20：30 | |

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。

1. **实验内容说明**

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. 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。

4. 完成单周期CPU实验的复现，并进行实验箱验证。

5. 分别添加一种R型指令和I型指令，并进行实验箱验证。

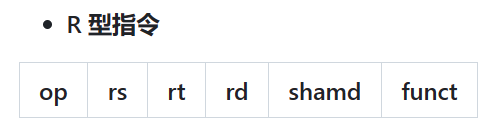
1. **实验原理图**



图1 单周期CPU的实现框图

1. **单周期CPU的指令执行过程**

单周期CPU能够执行MIPS指令集系统的一个子集，共16条指令，包括存储访问指令、运算指令、跳转指令。

****

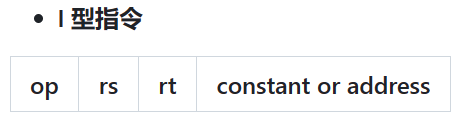
（1） 从指令存储器中取指令，更新 PC 。

（2）ALU 根据 funct 字段确定 ALU 的功能。

（3）从寄存器堆中读出寄存器 rs 和 rt。

（4）ALU 根据 2 中确定的功能，对从寄存器堆读出的数据进行操作。

（5）将运算结果写入到 rd 字段对应的目标寄存器。

****

（1）存取指令：

a. 从指令存储器中取指令，更新 PC 。

b. ALU 根据 op 字段确定 ALU 的功能。

c. 从寄存器堆中读出寄存器 rs 的值，并将其与符号扩展后的指令低16位立即数的值相加。

d. 若为存储指令，则将 rt 寄存器中的值存到上步相加得到的存储器地址；

e. 若为取数指令，则将 上步所得存储器地址里所存的数据放到 rt 目标寄存器中。

(2) 分支指令：

a. 从指令存储器中取指令，更新 PC 。

b. 从寄存器堆中读出寄存器 rs 和 rt 的值。

c. 将所读寄存器的两值相减。

d. 根据上步的结果是否为0，将 PC+4 的值或 address 字段所对应地址存入PC中。

****

(1) 从指令存储器中取指令，更新 PC 。

(2) 取出 address 字段，作为目标跳转地址。

(3) 将目标跳转地址存入PC中。

**5、 实验步骤**

（1）复现单周期CPU

按照指导手册操作，复现单周期CPU，理解单周期CPU的指令执行过程。

（2）添加一条R型指令和一条I型指令

选择按位同或操作作为添加的R型指令，低位加载操作作为添加的I型指令。

**修改后的代码（修改处标红）**

**1.single\_cycle\_cpu.v**

（1）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;

//将添加的两个操作添加到指令列表中

//inst\_NXOR 表示按位同或操作

//inst\_HUI 表示低位加载操作

wire inst\_NXOR, inst\_HUI;

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\_NXOR = (op == 6'b110001); // 逻辑同或运算

assign inst\_HUI = (op == 6'b110000); // 立即数装载低半字节

（2）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,inst\_hui;

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\_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

assign inst\_nxor= inst\_NXOR; // 逻辑同或

assign inst\_hui = inst\_HUI; // 立即数装载低位

（3）修改立即数拓展指令

assign inst\_imm\_sign = inst\_ADDIU | inst\_LUI | inst\_LW | inst\_SW | inst\_HUI;

（4）添加独热编码

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\_nxor,

inst\_hui};

（5）修改寄存器的写回值

assign inst\_wdest\_rd = inst\_ADDU | inst\_SUBU | inst\_SLT | inst\_AND | inst\_NOR

| inst\_OR | inst\_XOR | inst\_SLL | inst\_SRL | inst\_NXOR;

**2.alu.v**

（1）拓展alu控制信号

input [13:0] alu\_control, // ALU控制信号

（2）添加独热码

// 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\_nxor; //按位同或

wire alu\_hui; //低位加载

assign alu\_hui = alu\_control[13];

assign alu\_nxor = 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];

（3）添加结果信号

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] hui\_result;

（4）添加运算逻辑

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; // 同或结果为异或结果按位取反

assign hui\_result = {16'd0, alu\_src2[15:0]}; // 立即数装载结果为立即数移位至低半字节

（5）选择相应结果输出

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\_hui ? hui\_result :

32'd0;

**3.inst\_rom.v**

添加指令

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

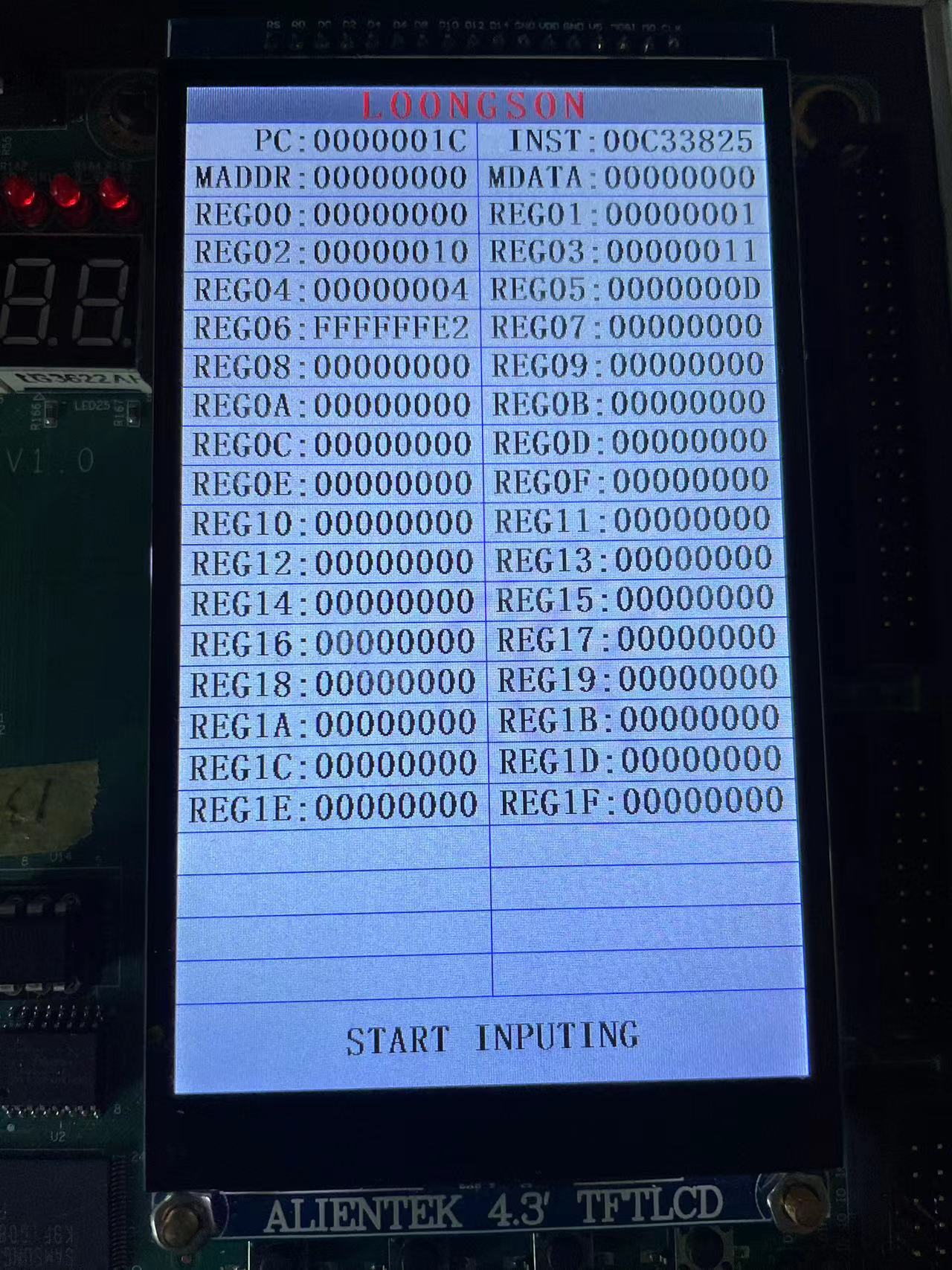
assign inst\_rom[11] = 32'h00C78631; // 2CH: nxor $13,$6,$7 | $13 = FFFF\_FFEEH

assign inst\_rom[12] = 32'hC00E000E; // 30H: hui $14,#14 | $14 = 0000\_000EH

**6、 实验结果分析**

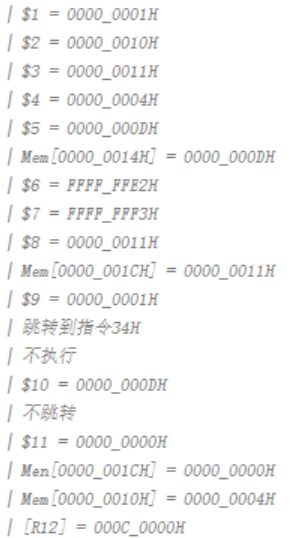
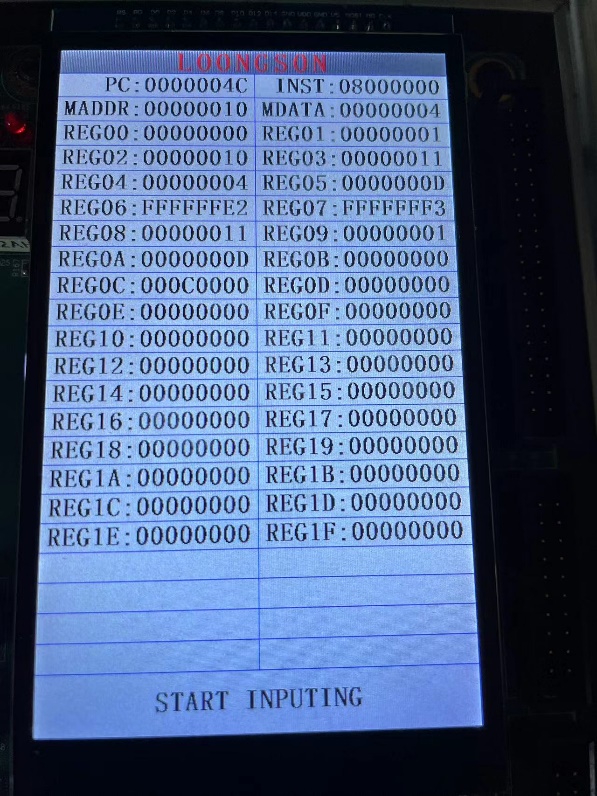
**(1)复现单周期CPU**

**a.执行6条指令**



对应的第六条指令REG06=FFFFFFE2，还没有执行第七条指令，所以REG07=00000000，结果正确。

**b.执行全部指令**

****

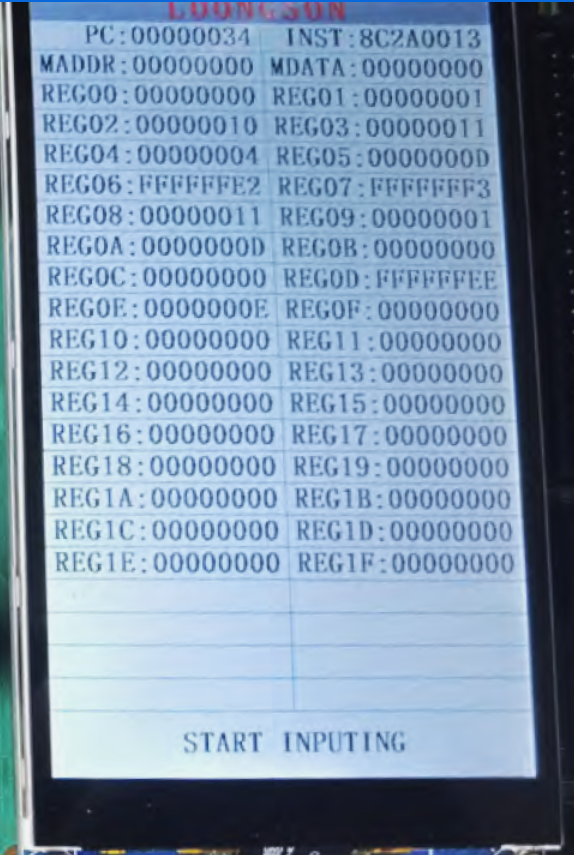
实验箱上的结果与理论结果一直，并且查询Mem[00000010]=00000004，结果正确。

**（2）添加指令后验证**

**a.执行按位同或操作**

|  |
| --- |
| assign inst\_rom[11] = 32'h00C78631; // 2CH: nxor $13,$6,$7 | $13 = FFFF\_FFEEH |

当第11条指令执行时，指令编码为0x00C75031，表示将$6和$7进行按位同或运算，将得到的结果存入$13中。根据之前汇编代码得出的结果，$6的值为0xFFFFFFE2，$7的值为0xFFFFFFF3，运算得到的结果为0xFFFFFFEE，所以REG0D中的值应该为0xFFFFFFEE。

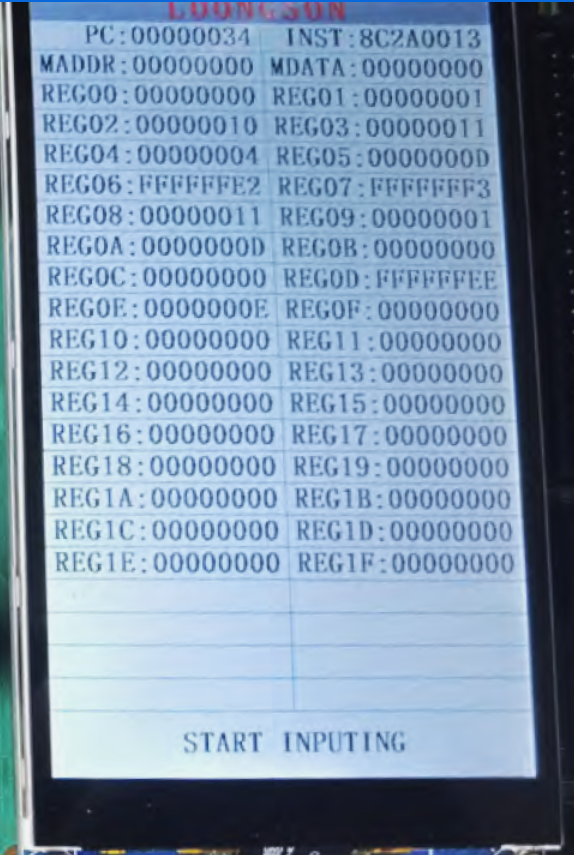


结果正确。

**b. 执行低位加载操作**

|  |
| --- |
| assign inst\_rom[12] = 32'hC00E000E; // 30H: hui $14,#14 | $14 = 0000\_000EH |

当第12条指令执行时，指令编码为0xC00E000E，表示将立即数14按低位加载的模式加载到寄存器$14中。REG0E中的值应为0x0000000E。



结果正确。

**7、 总结感想**

本次实验是复现单周期CPU，并添加一条R型指令和一条I型指令。通过实验，我深刻理解了单周期CPU的指令执行过程，同时也学会了如何添加新的指令。

在实验过程中，我首先按照指导手册操作，复现了单周期CPU。通过对单周期CPU的各个模块的理解，我深入了解了单周期CPU的指令执行过程，包括指令的取指、指令的译码、指令的执行、数据的读写等过程。这些过程相互协作，最终完成了指令的执行。

接着，我添加了一条R型指令和一条I型指令。我选择了按位同或操作作为添加的R型指令，低位加载操作作为添加的I型指令。通过对指令的添加，我深入了解了指令的格式和指令的编码方式。同时，我也学会了如何在单周期CPU中添加新的指令，这对我理解CPU的指令集和指令的执行过程有着重要的意义。

总之，本次实验让我深刻理解了单周期CPU的指令执行过程，同时也学会了如何添加新的指令。这对我今后的学习和研究都有着重要的意义。