**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**
2. 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。
3. 了解熟悉MIPS体系的处理器结构，如延迟槽，哈佛结构的概念。
4. 熟悉并掌握单周期CPU的原理和设计。
5. 进一步加强运用verilog语言进行电路设计的能力。
6. 为后续设计多周期cpu的实验打下基础。
7. **实验设备**
8. 装有Xilinx Vivado的计算机一台。
9. LS-CPU-EXB-002教学系统实验箱一套。
10. **实验任务**
11. 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。
12. 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。
13. 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。

1. 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。

clk

异步指令ROM

异步数据RAM

寄存器堆

ALU模块

PC

图 7.1 单周期CPU的大致框图

1. 根据设计的实验方案，使用verilog编写相应代码。
2. 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 |
|  |  |  |  |  |
|  |  |  |  |  |

1. 对编写的代码进行仿真，得到正确的波形图。
2. 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

单周期CPU

外围模块

来自FPGA板子上的输入

输出到FPGA板上进行展示

1. 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。
3. **实验内容说明**
4. 做好预习：
5. 熟知MIPS指令类型，深入理解常用指令的功能和编码；
6. 归纳常用的MIPS指令，确定自己准备实现的MIPS指令；
7. 对准备实现的指令进行分析，完成表7.1的填写；
8. 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；
9. 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；
10. 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。
11. 实验实施：
12. 确认单周期CPU的设计框图的正确性；
13. 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；
14. 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；
15. 完成调用单周期CPU的外围模块的设计，并编写代码；
16. 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。
17. 实验检查：
18. 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。
19. 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。

1. **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：
3. MIPS指令格式要使用规范格式；
4. 指令执行验证需要修改inst\_rom中预存储的16进制指令数据；
5. 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。
6. 实验原理图使用实验指导书的图7.3即可，无需修改。
7. **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

1. **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

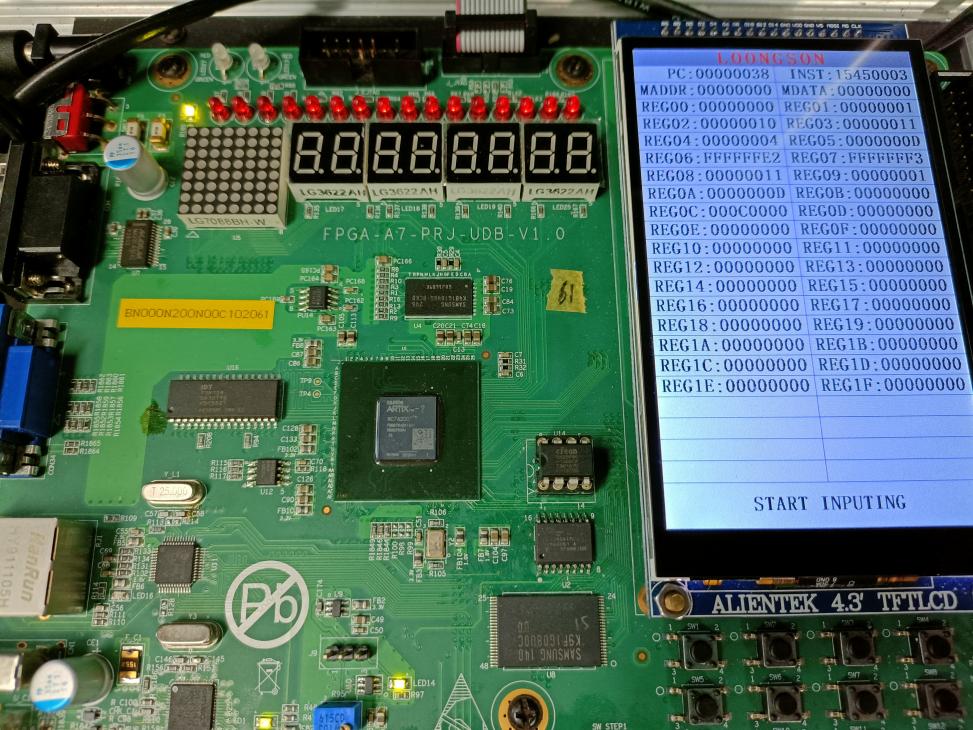
1. 修改代码single\_cycle\_cpu.v

|  |  |
| --- | --- |
|  | `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 ) // O, 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-------------------------------------// |

1. 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 |

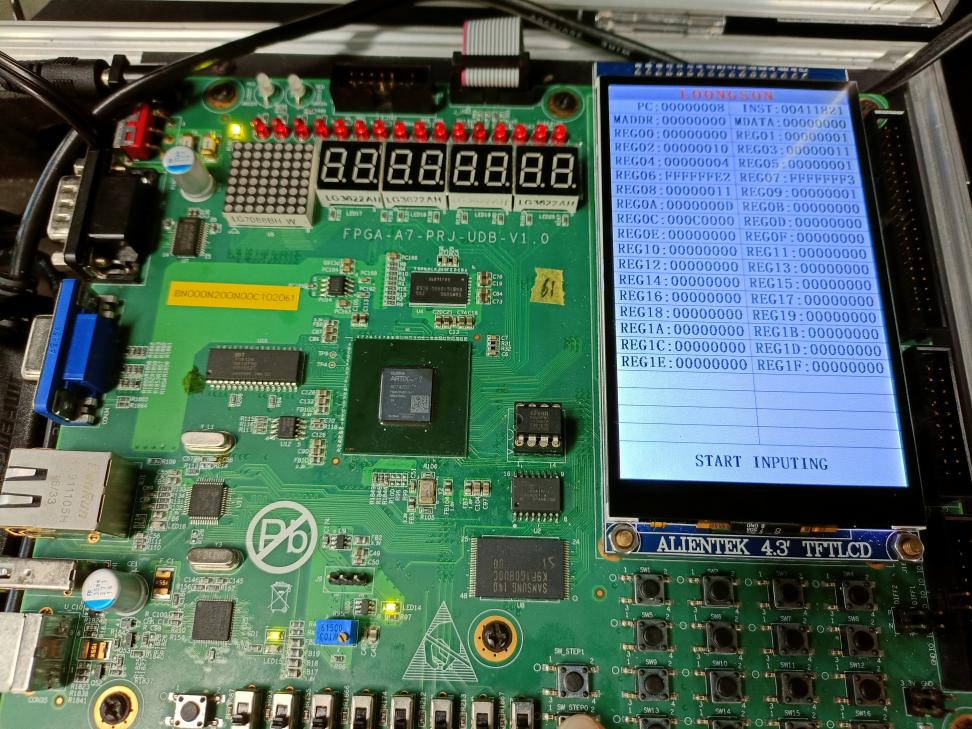
1. **实验结果分析**
2. 验证修改前

****

1. 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

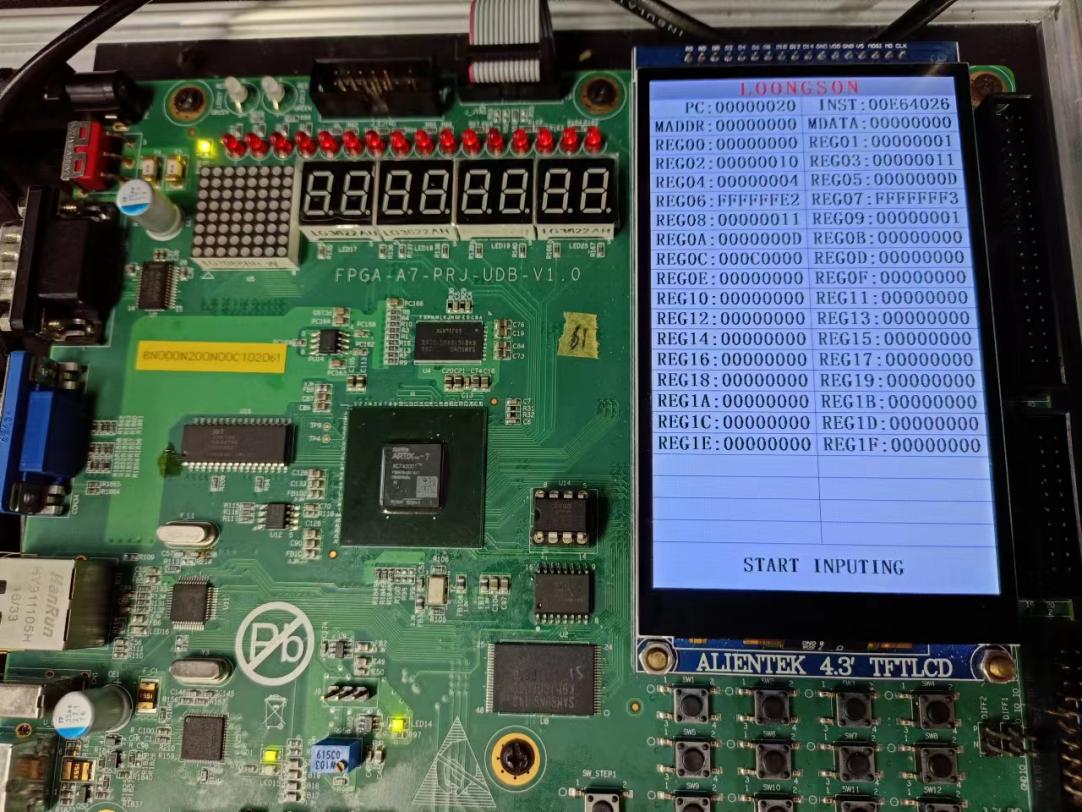
寄存器10，也就是REG0A应该显示的结果是0000000D，结果是符合的。



1. 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

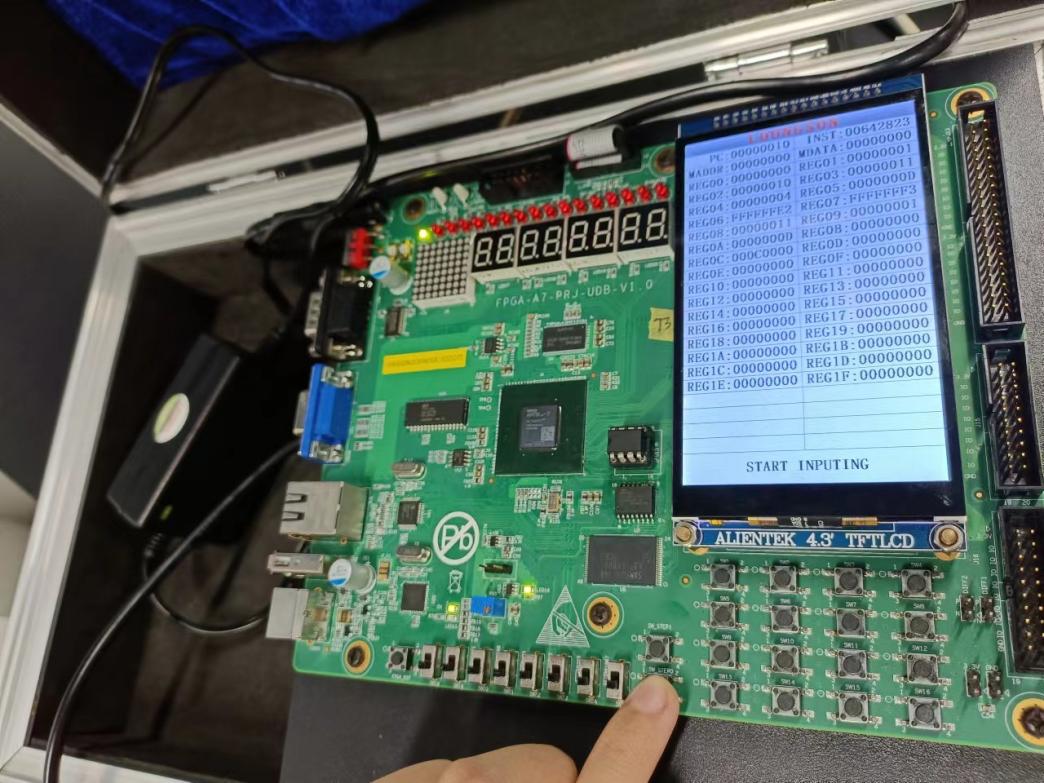
寄存器2，也就是REG02应该显示的结果是00000010，结果是符合的。



1. 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。

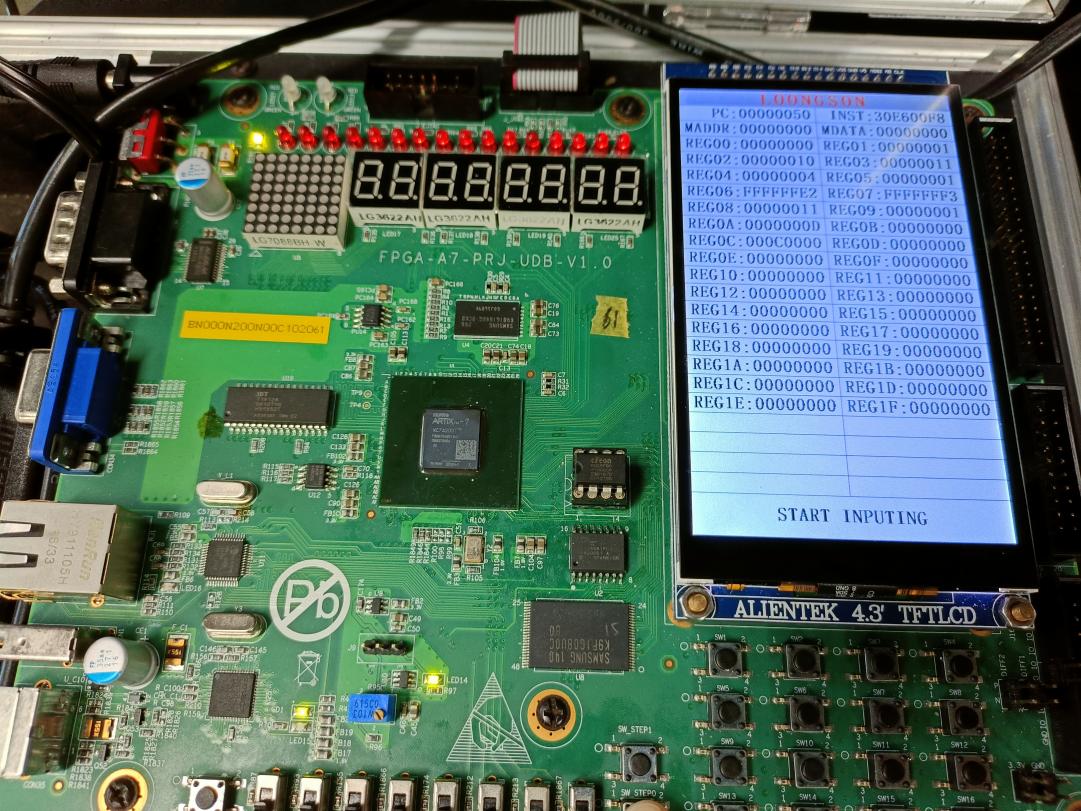


1. 下一条指令PC=10H，则现在程序执行的是

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。

1. 添加算术右移后



1. 下一条指令PC=50H，则现在程序执行的是

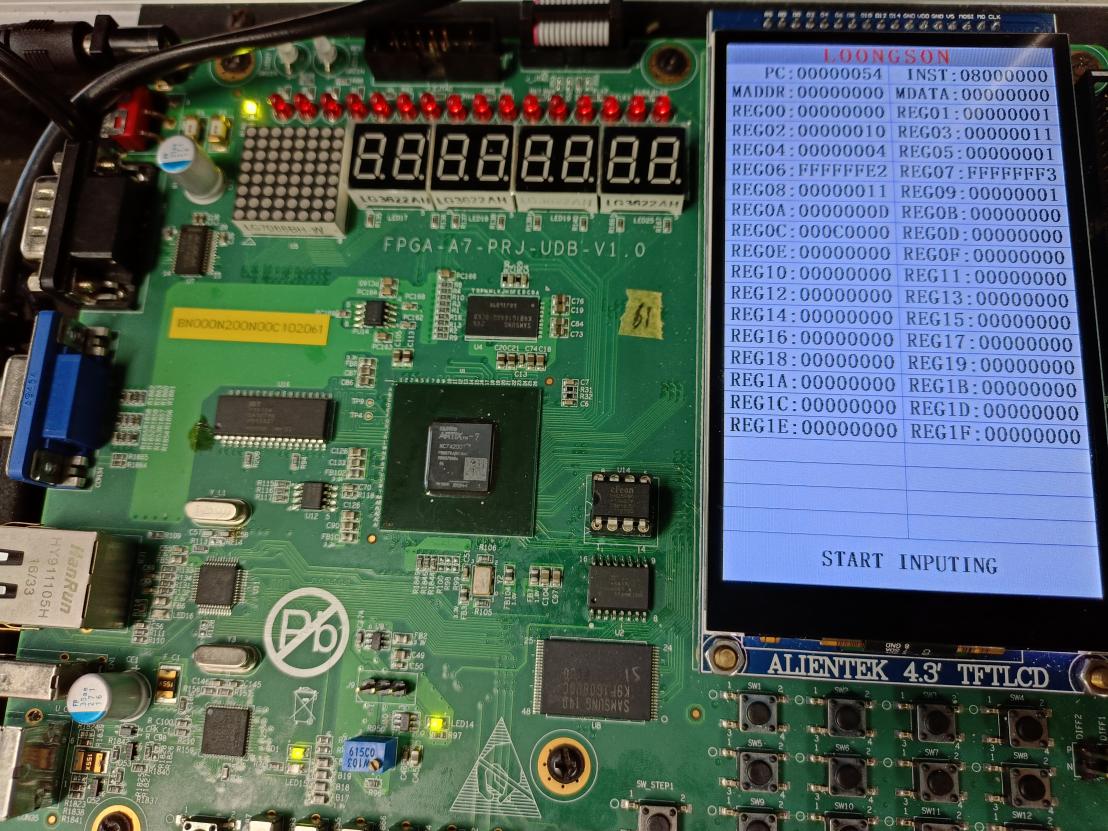
assign inst\_rom[19] = 32'h000A28C3; // 4CH: sra $5 ,$10,#3 | $5 = 0000\_0001H

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。

1. 添加立即数或后



1. 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。

1. **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。