组成原理课程第 五 次实验报告

实验名称：ROM 存储器实验和单周期 CPU 实验

学号：2312141 姓名： 张德民 班次：李涛老师班

一、 实验目的

1. 了解只读存储器 ROM 和随机存取存储器 RAM 的原理。
2. 理解 ROM 读取数据及 RAM 读取、写入数据的过程。
3. 理解计算机中存储器地址编址和数据索引方法。
4. 理解同步 RAM 和异步 RAM 的区别。
5. 掌握调用 xilinx 库 IP 实例化 RAM 的设计方法。
6. 熟悉并运用 verilog 语言进行电路设计。
7. 为后续设计 cpu 的实验打下基础。
8. 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。
9. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
10. 熟悉并掌握单周期 CPU 的原理和设计。
11. 进一步加强运用 verilog 语言进行电路设计的能力。
12. 为后续设计多周期 cpu 的实验打下基础。

二、 实验内容说明

1、ROM 存储器实验请完成验证，总结收获，并对比跟之前的同步异步 RAM 实验有何不同，分析总结原因。

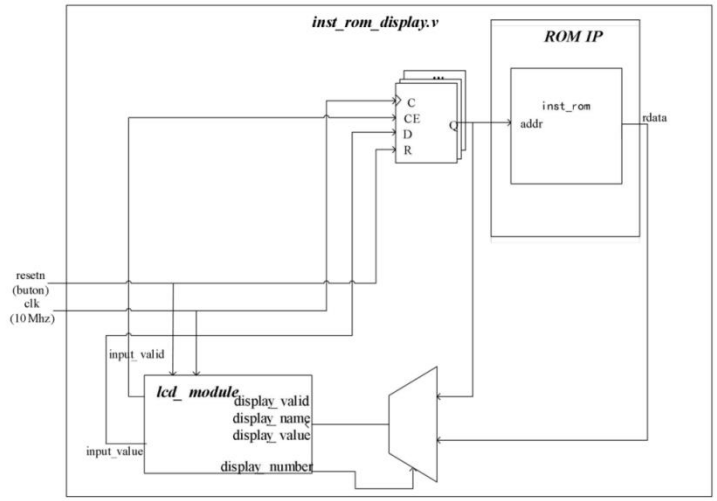
2、单周期 CPU 实验，原理图应基于实验指导手册中的图 7.1，在分析表 7.4 中指令执行过程时，可以在图 7.1 基础上辅助画线表示执行过程。

3、R 型指令和 I 型指令挑两条分析总结执行过程，J 型指令就 1 条，请直接分析总结执行过程。注意，这些指令已经在 inst_rom.v 里面写好，所以请找到对应的指令，逐个分析。从指令的二进制编码开始，分析介绍代码是如何一步一步完成运算并执行的。

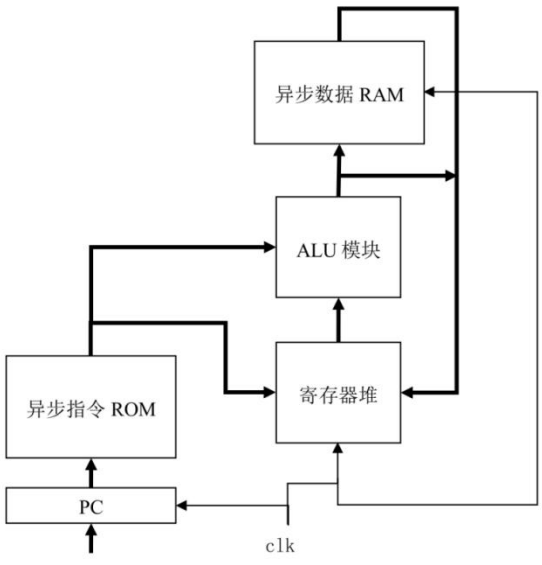
4、（提高要求，不强求做出来）把 ALU 实验中添加的三个指令，自行添加到这个单周期 CPU 中，注意指令码只要跟现有的不冲突就行，不必限制在标准的 MIPS 指令格式。

三、 实验原理图

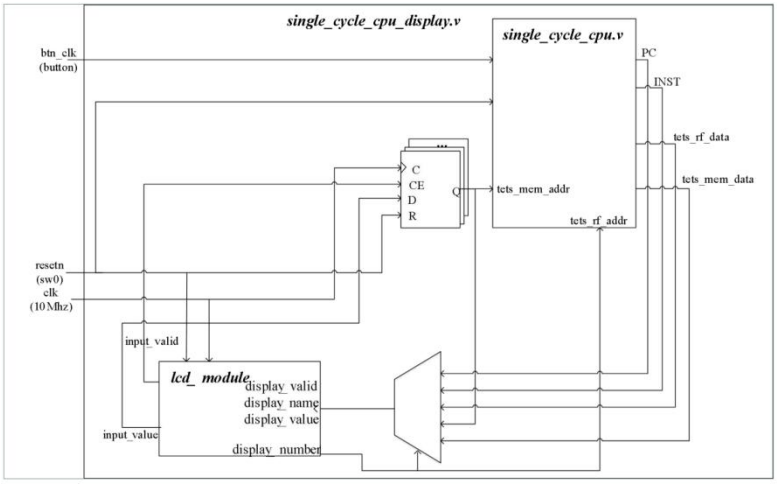
同步 ROM 的顶层模块



单周期 CPU 的大致框图



单周期 CPU 顶层框图



四、 实验步骤

同步 ROM 实验

首先我们需要按照教程创建一个同步 ROM 的 IP 核。

然后自行写入 coe 文件。使用 16 进制的数据。这里只写入两条，作为测试。

e:/computer reason/inst_rom.coe

```
1  memory_initialization_radix = 16;
2  memory_initialization_vector =
3  24010001
4  00011100
```

然后在 display 文件里，我们可以在数码屏上输入指令地址 addr，然后读取到相应地址上的指令。

```
reg  [31:0] addr;
wire [31:0] inst;
```

异步 ROM 实验

ROM 模块实现了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。

```
module inst_rom(
    input       [4 :0] addr, // 指令地址
    output reg [31:0] inst       // 指令
    );
```

这里定义了输入的 5 位的读指令地址，以及 32 位的指令 inst。

```
wire [31:0] inst_rom[19:0];  // 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) | Mem[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'h08000000; // 4CH: j     00H         | 跳转指令00H
```

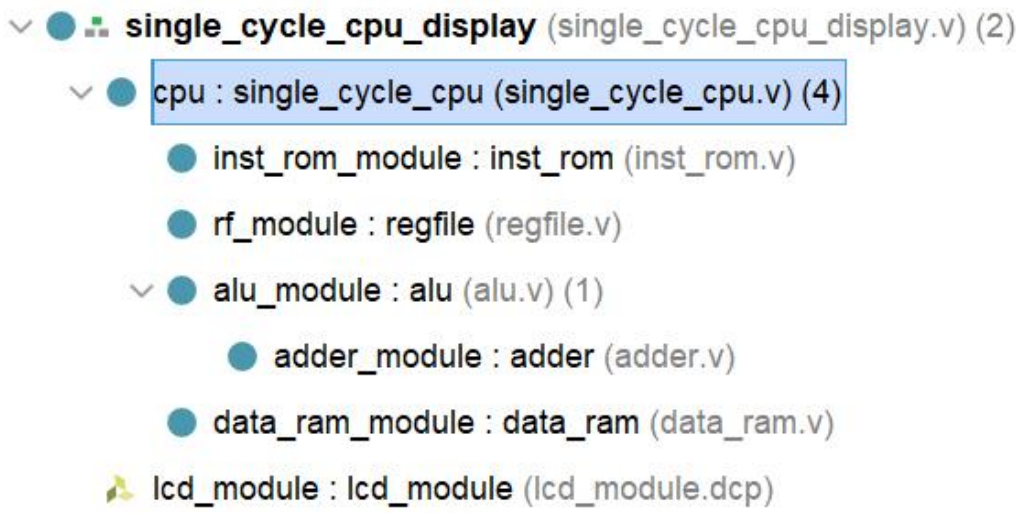
然后用 20 个 32 位的 wire 型数据来存储提前确定好的指令。

```
    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];
            default: inst <= 32'd0;
        endcase
    end
endmodule
```

然后根据 addr 的值去取相应地址的值。

单周期 CPU 实验

首先我来解释一下单周期 cpu 实验所需模块的相关内容。关于 R 型指令，I 型指令，J 型指令的解释分析，会在本节后边进行解释。下图是模块的结构。



ALU 模块

这个模块和上个实验实现的 ALU 模块是一样的，实现了下图的种运算。

```
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
```

并且还有一个加法器的子模块。

```
// 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 :
                    32'd0;
dmodule
```

最后会根据你输入的指令编码来输出运算结果。

data_ram 模块

这个模块实现了一个同步写，异步读的存储器模块。

```
module data_ram(
    input         clk,          // 时钟
    input  [3:0]  wen,          // 字节写使能
    input  [4:0] addr,          // 地址
    input  [31:0] wdata,        // 写数据
    output reg [31:0] rdata,        // 读数据

    //调试端口，用于读出数据显示
    input  [4 :0] test_addr,
    output reg [31:0] test_data
);
    reg [31:0] DM[31:0];  //数据存储器，字节地址7'b000_0000~7'b111_1111
```

如图，这个模块的 wen 输入有 4 位，意思是把 32 位的输入数据分成 4 段，每段 8 位进行处理。addr 是 5 位，可以寻址 32 个 32 位的数据。wdata 和 rdata 都是 32 位的。

test_addr 和 test_data 是用来进行调试的输入地址和输出数据。

DM 是 32 个 32 位的寄存器，这也就是全部的内存空间了。

```
always @(posedge clk)    // 当写控制信号为1，数
begin
    if (wen[3])
    begin
        DM[addr][31:24] <= wdata[31:24];
    end
end
always @(posedge clk)
begin
    if (wen[2])
    begin
        DM[addr][23:16] <= wdata[23:16];
    end
end
always @(posedge clk)
begin
    if (wen[1])
    begin
        DM[addr][15: 8] <= wdata[15: 8];
    end
end
always @(posedge clk)
begin
    if (wen[0])
    begin
        DM[addr][7 : 0] <= wdata[7 : 0];
    end
```

接下来在 always 语句里定义，当时钟上升沿的时候，根据写使能端的信号，来写入四段数据的信息。因为四位的 wen 信号把 32 位的数据均分为 4 分。

```
//读数据,取4字节
always @(*)
begin
    case (addr)
        5'd0 : rdata <= DM[0 ];
        5'd1 : rdata <= DM[1 ];
        5'd2 : rdata <= DM[2 ];
        5'd3 : rdata <= DM[3 ];
        5'd4 : rdata <= DM[4 ];
        5'd5 : rdata <= DM[5 ];
        5'd6 : rdata <= DM[6 ];
        5'd7 : rdata <= DM[7 ];
        5'd8 : rdata <= DM[8 ];
        5'd9 : rdata <= DM[9 ];
        5'd10: rdata <= DM[10];
        5'd11: rdata <= DM[11];
        5'd12: rdata <= DM[12];
        5'd13: rdata <= DM[13];
        5'd14: rdata <= DM[14];
        5'd15: rdata <= DM[15];
        5'd16: rdata <= DM[16];
        5'd17: rdata <= DM[17];
        5'd18: rdata <= DM[18];
        5'd19: rdata <= DM[19];
        5'd20: rdata <= DM[20];
        5'd21: rdata <= DM[21];
        5'd22: rdata <= DM[22];
        5'd23: rdata <= DM[23];
```

上图是读数据的部分，根据 addr 的值从 DM 里读取相应的数据。

regfile 模块

这部分是实现了寄存器堆的模块。同步写，异步读。

```
module regfile(
    input              clk,
    input              wen,
    input       [4 :0] raddr1,
    input       [4 :0] raddr2,
    input       [4 :0] waddr,
    input       [31:0] wdata,
    output reg  [31:0] rdata1,
    output reg  [31:0] rdata2,
    input       [4 :0] test_addr,
    output reg  [31:0] test_data
    );
    reg [31:0] rf[31:0];
```

如上图，设定输入 wen 写使能端，raddr1 和 raddr2 是两个读数据的地址，waddr 是写如数据的寄存器的地址，wdata 是写入的数据，输出 rdata1 和 rdata2 是两个读取的数据。test_addr 和 test_data 是用来进行调试的输入地址和输出数据。rf 是 32 个 32 位的寄存器。

```
always @(posedge clk)
begin
    if (wen)
    begin
        rf[waddr] <= wdata;
    end
end
```

然后在时钟上升沿根据写使能信号和写入地址进行写入。

```
always @(*)
begin
    case (raddr1)
        5'd1 : rdata1 <= rf[1 ];
        5'd2 : rdata1 <= rf[2 ];
        5'd3 : rdata1 <= rf[3 ];
        5'd4 : rdata1 <= rf[4 ];
        5'd5 : rdata1 <= rf[5 ];
        5'd6 : rdata1 <= rf[6 ];
        5'd7 : rdata1 <= rf[7 ];
```

然后根据两个读地址去相应寄存器里进行读取数据。

rom 模块

这个模块实现逻辑和上边的异步 ROM 实现的逻辑是一样的，就不多介绍了。

Single_cycle_cpu 模块

这个模块实现了单周期 cpu 的模块。每条指令在单周期内进行。

```
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
    );
```

首先定义这个模块，输入 rf_addr 是寄存器堆的地址，用来读寄存器里的值，输入 mem_addr 是数据存储器的地址，输出 rf_data 是存储从寄存器堆里读取的值，输出 mem_data 用来存储从数据存储器里读取的值。输出 cpu_pc 存当前程序计数器 PC 的值，输出 cpu_inst 用来存当前执行的指令。

接下来开始取指部分。

```
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];
```

接下来定义一个 32 位的寄存器 pc，用于存储当前的程序计数器的值。定义一个 32 位的线网 next_pc，用于存储下一个程序计数器的值。seq_pc：定义一个 32 位的线网 seq_pc，表示顺序执行的下一条指令的地址（pc + 4）。然后定义一个 32 位的线网 jbr_target，表示跳转指令的目标地址。定义一个线网 jbr_taken，表示是否发生跳转。

然后让 seq_pc 里存入当前的 pc 值加 4。

```
assign next_pc = jbr_taken ? jbr_target : seq_pc;
```

接下来进行一个判断，如果没有跳转指令，那么下一条指令地址就是当前的 PC 值加 4，否则下一条指令就是跳转的目标地址。

```
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
```

然后用一个 always 语句更新 PC 程序计数器的值，每当时钟上升沿的时候，如果有复位信号，就取起始地址，否则就更新为下一个地址。

```
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;
```

定义一个 32 位的 wire 型数据 inst_addr,用于存储指令地址。定义一个 32 位的 wire 型数据 inst，用于存储从指令存储器读取的指令

将指令地址 inst_addr 赋值为当前的程序计数器 pc 的值。

实例化一个名为 inst_rom_module 的指令存储器模块,传入指令地址 inst_addr 的低 5 位（[6:2]），并获取读取的指令 inst。

将当前的程序计数器pc的值输出到cpu_pc,用于显示。将从指令存储器读取的指令 inst 输出到 cpu_inst，用于显示。然后取指部分结束。

接下来开始译码阶段。

```
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
```

首先定义一系列 wire 型数据，用于从指令 inst 中提取不同的字段：

op是操作码,从inst的[31:26]位提取。rs是源操作数1的寄存器编号,从inst的[25:21]位提取。rt 是源操作数 2 的寄存器编号，从 inst 的[20:16]位提取。rd 是目标操作数的寄存器编号，从 inst 的[15:11]位提取。sa 是特殊域，可能存放偏移量，从 inst 的[10:6]位提取。funct 是功能码，从 inst 的[5:0]位提取。imm 是立即数，从 inst 的[15:0]位提取。offset 是地址偏移量，从 inst 的[15:0]位提取。target 是目标地址，从 inst 的[25:0]位提取。

这些字段并不是同时都要用的，要根据具体指令来使用。

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

op_zero 和 sa_zero 两个 wire 型数据，分别用于判断操作码 op 和特殊域 sa 是否全为 0。使用按位或操作 (|op) 判断 op 是否全为 0，取反后得到 op_zero。

同理，判断 sa 是否全为 0 并取反得到 sa_zero。

```
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;

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
```

首先定义一系列 wire 型数据，用于表示不同的指令是否被执行，这些 wire 型数据将根据操作码和功能码等字段进行判断。然后根据操作码 op、特殊域 sa 和功能码 funct 等字段判断当前指令是否为相应的指令，将结果赋值给对应的线网。

```
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};
```

j_taken 和 j_target 这两个 wire 型数据，分别表示是否发生无条件跳转和无条件跳转的目标地址。如果当前指令是无条件跳转指令，则 j_taken 为 1。计算无条件跳转的目标地

址，将 pc 的高 4 位、指令中的目标地址 target 和低 2 位 0 拼接起来。

```
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;
```

wire beq_taken、wire bne_taken 和 br_target 这三个 wire 型数据，分别表示 BEQ 指令是否跳转、BNE 指令是否跳转以及分支跳转的目标地址。

如果是 BEQ 指令且源操作数 rs_value 和 rt_value 相等，则 beq_taken 为 1。BNE 指令的跳转条件是 BEQ 跳转条件的取反。

然后计算分支跳转目标地址的高 30 位，将 pc 的高 30 位加上符号扩展后的偏移量 offset。分支跳转目标地址的低 2 位保持 pc 的低 2 位不变。

然后综合判断是否发生跳转，包括无条件跳转 j_taken、BEQ 指令满足条件和 BNE 指令满足条件的情况。

最后根据是否是无条件跳转决定最终的跳转目标地址 jbr_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)
);
```

接下来实现寄存器堆，首先用 wire 型数据 rf_wen 来存是否要写入的信号，rf_wdata 存要写入的数据，rf_waddr 用来存写入的地址。rs_value 和 rt_value 用来存两个源操作数的值。然后实例化一个名为 rf_moudle 的寄存器堆模块。

```
// 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;
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
```

然后定义一系列 wire 型数据，用于表示不同的指令是否被执行，这些线网将根据之前设定好的值进行赋值。

```
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;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;
```

定义一个 32 比特的 wire 型数据 sext_imm，用于存储经过符号扩展后的立即数。定义一个单比特的 wire 型数据 inst_shf_sa,用于标识当前指令是否需要使用 sa 域作为偏移量。定义一个单比特的 wire 型数据 inst_imm_sign，用于标识当前指令是否需要对立即数进行符号扩展。

然后对 16 位的立即数 imm 进行符号扩展,将其扩展为 32 位。具体做法是将 imm 的最高位（符号位）复制 16 次，然后与 imm 拼接在一起。存在 sext_imm 里。

当指令是逻辑左移 inst_SLL 或逻辑右移 inst_SRL 时，inst_shf_sa 信号为高电平，表示需要使用 sa 域作为偏移量。

当指令是立即数无符号加法 inst_ADDIU、立即数装载高半字节 inst_LUI、从内存装载 inst_LW 或向内存存储 inst_SW 时，inst_imm_sign 信号为高电平，表示需要对立即数进行符号扩展。

```
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};
```

然后，定义 wire 型数据 alu_operand1 和 alu_operand2，用来存 alu 的两个操作数，用一个 12 位的 wire 型数据 alu_control 来存操作码，确定要执行什么运算。

到这里译码阶段就结束了，接下来要开始执行阶段。

```
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
);
```

执行阶段比较简单，实例化一个 alu 模块，用一个 32 位的线型数据 alu_result 来存运算后的结果。

然后开始访存阶段。

```
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     )
);
```

定义一个 4 比特的 wire 型数据 dm_wen，作为的写使能信号。当非置为且 inst_SW 信号为 1 的时候，dm_wen 设为 1111，否则 0000。

定义一个 32 比特的 wire 型数据 dm_addr，用于要访问的数据存储器地址。存的是 alu 的结果。

定义一个 32 比特的 wire 型数据 dm_wdata，用于存储要写入数据存储器的数据。存的是 rt 寄存器的值。

定义一个 32 比特的 wire 型数据 dm_rdata，用于存储从数据存储器中读取的数据。

访存结束，最后开始写回过程。

```
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;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                     | inst_OR   | inst_XOR  | inst_SLL | inst_SRL;
// 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
```

定义一个单比特的 wire 型数据 inst_wdest_rt，用于标识当前指令是否需要将结果写入寄存器堆的 rt 寄存器。同理定义一个单比特的 wire 型数据 inst_wdest_rd，用于标识当前指令是否需要将结果写入寄存器堆的 rd 寄存器。

当指令是立即数无符号加法 inst_ADDIU、从内存装载 inst_LW 或立即数装载高半字节 inst_LUI 时，inst_wdest_rt 信号为高电平，表示需要将结果写入寄存器堆的 rt 寄存器。

当指令是无符号加法 inst_ADDU、无符号减法 inst_SUBU、小于置位 inst_SLT、逻辑与 inst_AND、逻辑或非 inst_NOR、逻辑或 inst_OR、逻辑异或 inst_XOR、逻辑左移 inst_SLL 或逻辑右移 inst_SRL 时，inst_wdest_rd 信号为高电平，表示需要将结果写入寄存器堆的 rd 寄存器。

Single_cycle_cpu_display 模块

这里仅分析核心代码。

```
single_cycle_cpu cpu(
    .clk      (cpu_clk   ),
    .resetn   (resetn    ),

    .rf_addr  (rf_addr ),
    .mem_addr(mem_addr),
    .rf_data  (rf_data ),
    .mem_data(mem_data),
    .cpu_pc   (cpu_pc  ),
    .cpu_inst(cpu_inst)
);
```

首先实例化一个 single_cycle_cpu 模块。

然后在数码屏上显示相关内容即可。

分析 R 型指令

R 型指令主要用于执行寄存器之间的算术和逻辑运算，操作数和结果一般都存于寄存器

中。R 型指令一般有加法，减法，逻辑与，逻辑或，逻辑异或，移位指令和比较置位指令。

我这里选取 inst_rom[2] = 32'h00411821，inst_rom[8] = 32'h00E64026 这两个 R 型指令进行分析。

分析 inst_rom[2] = 32'h00411821：

首先从 PC 寄存器里读取地址 08h，然后读取指令 inst_rom[2] = 32'h00411821。转为 2 进制就是 0000_0000_0100_0001_0001_1000_0010_0001,从右向左数，它的第 26 到 31 位是 op 码 000000，第 21 到 25 位是 00010，代表源操作数 1 是 2 号寄存器的值，第 16 到 20 位是 00001，代表源操作数 2 是 1 号寄存器，第 11 到 15 位是 00011，代表目的寄存器是 3 号寄存器，第 6 到 10 位是 00000，即 sa 是全零，第 0 到 5 位是 100001，这是 funct 码。

```
assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b100001);// 无符号加法
```

如上图，根据 single_cycle_cpu 模块里边的设置，op 码为全零，sa 也是全零，并且 funct 码是 100001，代表这一条指令进行的操作是无符号加法。

所以这条指令的意思就是 addu $3,$2,$1,即把 2 号寄存器和 1 号寄存器的值相加，存到 3 号寄存器里边。由于不是跳转指令，所以 PC 计数器的下一个更新值是 0Ch。

然后到了寄存器堆这里，根据之前提取到的 rs，rt 寄存器的编号，读取到 rs_value 和 rt_value 这俩操作数。

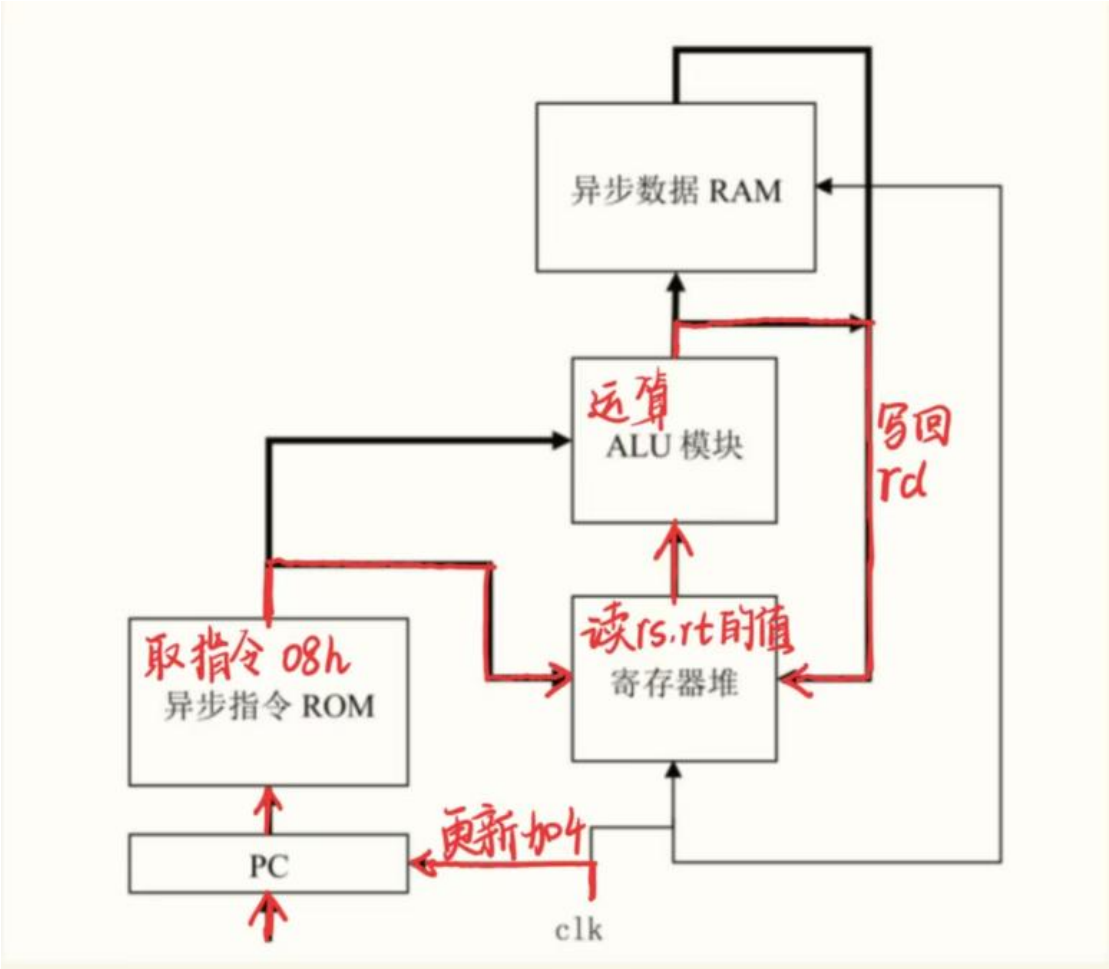
```
regfile rf_module(
    .clk    (clk      ),   // I, 1
    .wen    (rf_wen   ),   // I, 1
    .raddr1 (rs       ),   // I, 5
    .raddr2 (rt       ),   // I, 5
    .waddr  (rf_waddr ),   // I, 5
    .wdata  (rf_wdata ),   // I, 32
    .rdata1 (rs_value ),   // O, 32
    .rdata2 (rt_value ),    // O, 32
```

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
```

读完以后到了 ALU 部分，首先根据之前得到的指令 inst_addu 来设置 alu 要进行的运算是什么。这里无符号加法对应的也是加法指令。然后设置两个操作数的值，传入 ALU 中，得到 ALU 运算的最后结果。也就是无符号加法得到的结果。

最后进行写回操作，把 ALU 计算得到的结果写回到 rd 寄存器里。本周期的这一条指令就结束了。

可以用下图来表示执行过程：



分析 inst_rom[8] = 32'h00E64026：

首先从 PC 寄存器里读取地址 20h，然后把指令转化为二进制：0000_0000_1110_0110_0100_0000_0010_0110。

它的第 26 到 31 位是 op 码 000000，第 21 到 25 位是 00111，代表源操作数 1 是 7 号寄存器的值，第 16 到 20 位是 00110，代表源操作数 2 是 6 号寄存器，第 11 到 15 位是 01000，代表目的寄存器是 8 号寄存器，第 6 到 10 位是 00000，即 sa 是全零，第 0 到 5 位是 100110，这是 funct 码。

```
assign inst_XOR   = op_zero & sa_zero    & (funct == 6'b100110);// 逻辑异或运算
```

根据 op 码为全 0，sa 为全零，funct 码是 100110，可以得知要运行的是异或运算。

所以这一条指令的意思是 xor，$78,$7,$6,由于不是跳转指令，所以 PC 计数器的下一个更新值是当前值加 4，即 24h。

然后到了寄存器堆这里，根据之前提取到的 rs，rt 寄存器的编号，读取到 rs_value 和 rt_value 这俩操作数。

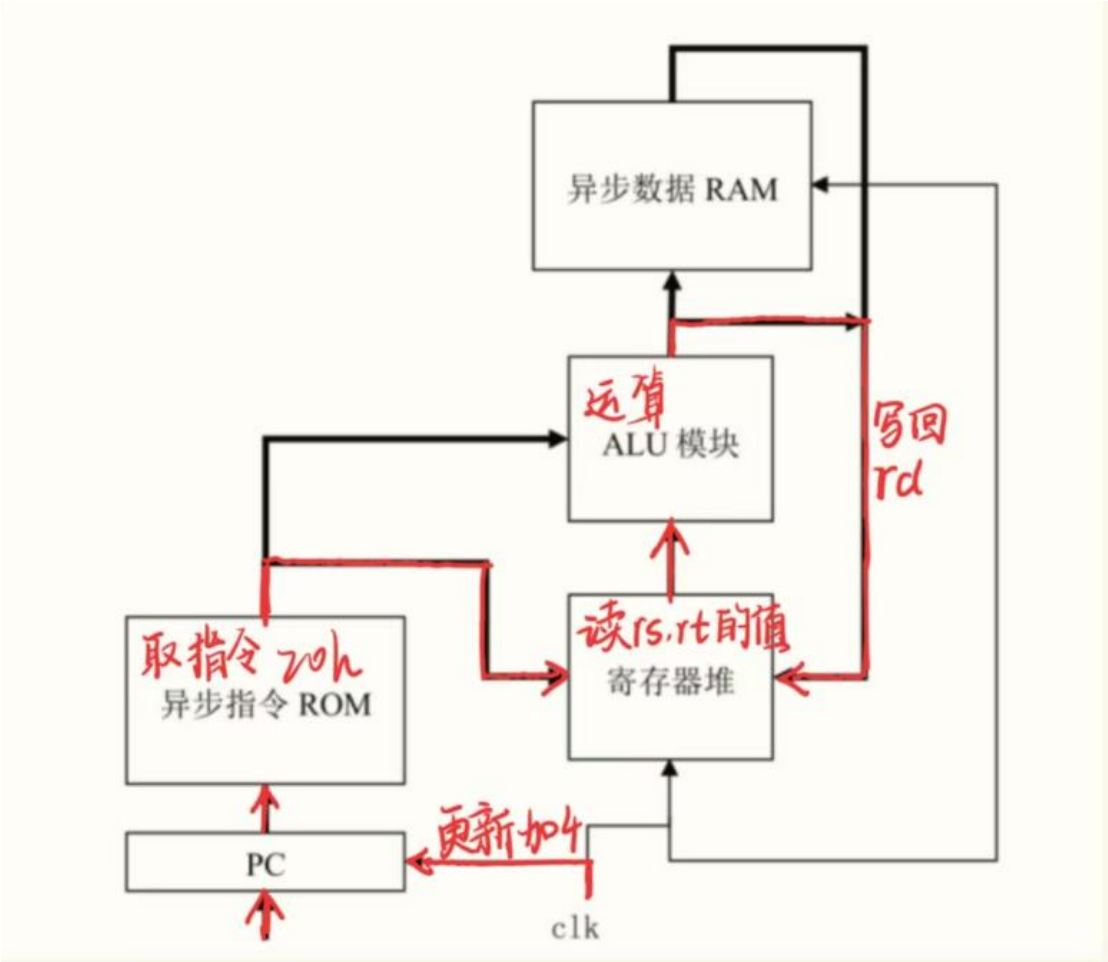
```
regfile rf_module(
    .clk    (clk      ),   // I, 1
    .wen    (rf_wen   ),   // I, 1
    .raddr1 (rs       ),   // I, 5
    .raddr2 (rt       ),   // I, 5
    .waddr  (rf_waddr ),   // I, 5
    .wdata  (rf_wdata ),   // I, 32
    .rdata1 (rs_value ),   // O, 32
    .rdata2 (rt_value ),    // O, 32
```

```
assign inst_xor = inst_XOR;   // 逻辑异或
```

然后根据指令确定要输入 ALU 的控制编码，然后设置两个操作数的值，传入 ALU 中，得到 ALU 运算的最后结果。也就是异或运算得到的结果。

最后进行写回操作，把 ALU 计算得到的结果写回到 rd 寄存器里。本周期的这一条指令就结束了。

流程图如下：



分析 I 型指令

I 型指令是一种常见的指令格式，用于执行带有立即数的运算或进行内存访问。我分析 inst_rom[0] = 32'h24010001 和 inst_rom[5] = 32'hAC250013 这两个指令，一个是立即数运算，另一个是 sw 运算。

分析 inst_rom[0] = 32'h24010001

首先从 PC 寄存器里读取地址 00h，然后把指令转化为二进制：0010_0100_0000_0001_0000_0000_0000_0001。

op 码是 001001，第 21 到 25 位是 rs 源寄存器，编号为 00000，16 到 20 是目的寄存器 rt，编号是 00001，第 0 到 15 位是立即数 0000_0000_0000_0001。扩展后得到的立即数是 1。

```
assign inst_ADDIU = (op == 6'b001001);                    // 立即数无符号加法
```

根据 op 码是 001001，我得知这一条指令进行的运算是立即数无符号加法。

则这一条指令的意思是 addiu，$1,$0,#1。

由于不是跳转指令，所以 PC 计数器的下一个更新值是当前值加 4，即 04h。

然后到了寄存器堆这里，根据之前提取到的 rs，寄存器的编号，读取到 rs_value 这个操作数。

```
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
```

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
```

```
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;
```

然后根据指令确定要输入 ALU 的控制编码,即进行加法，并设置立即数信号。

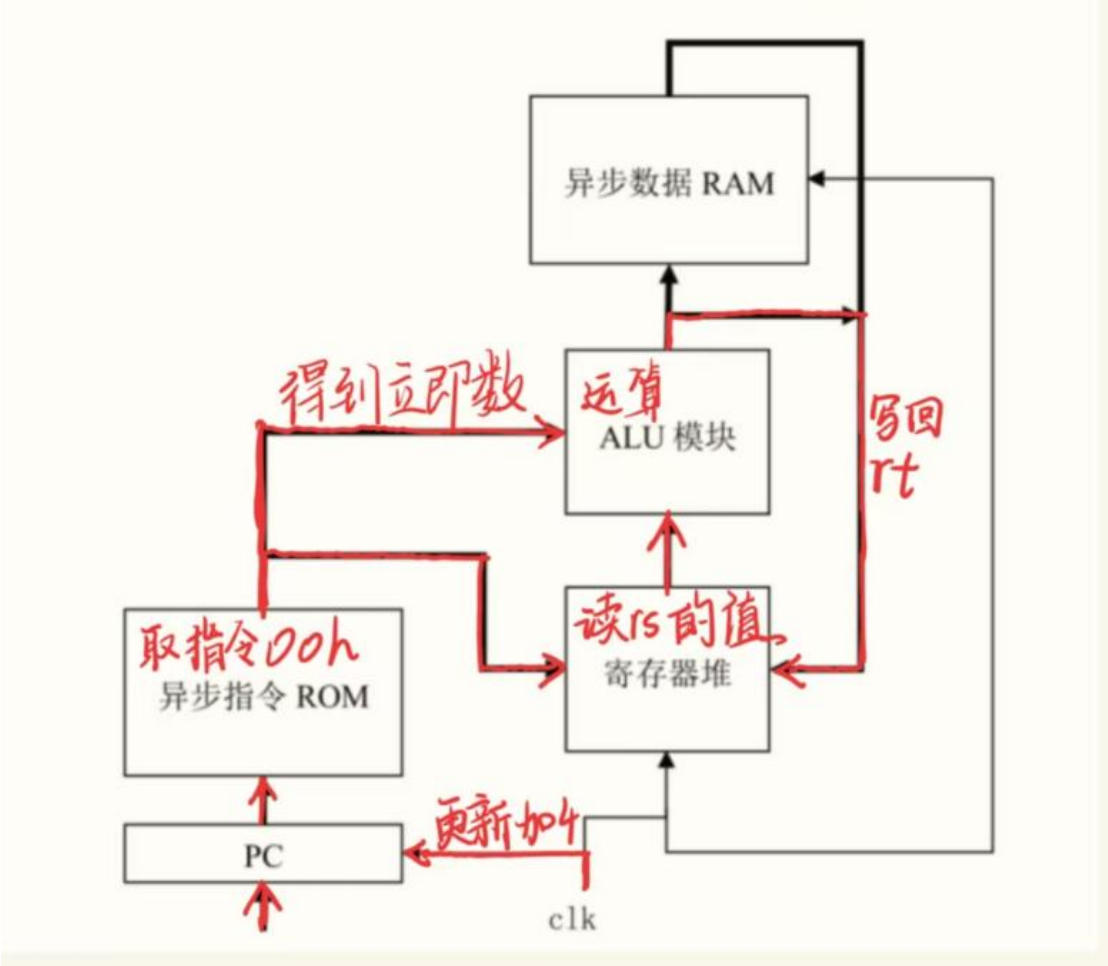
```
assign alu_operand1 = inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_imm_sign ? sext_imm : rt_value;
```

然后根据立即数信号设置两个操作数的值，一个是 rs 寄存器的值，另一个是立即数 1。传入 ALU 中，得到 ALU 运算的最后结果。也就是无符号立即数加法运算得到的结果。

```
assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI;
```

最后根据向 rt 寄存器写的信号，把运算得到的结果写回 rt 寄存器里。本周期的这一条指令就结束了。

流程图如下：



分析 inst_rom[5] = 32'hAC250013：

首先从 PC 寄存器里读取地址 14h，然后把指令转化为二进制：1010_1100_0010_0101_0000_0000_0001_0011。

op 码是 101011，第 21 到 25 位是 rs 源寄存器，编号为 00001，16 到 20 是目的寄存器 rt，编号是 00101，第 0 到 15 位是立即数(地址)0000_0000_0001_0011。即 19。

```
assign inst_SW     = (op == 6'b101011);                    // 向内存存储
```

根据操作码 101011，我可以知道，本条指令进行的操作是向内存存储操作。

那么这一条指令的意思是 sw $5 ,#19($1) 即把$1 寄存器里的地址偏移 19，取值存到￥5 里。

由于不是跳转指令，所以 PC 计数器的下一个更新值是当前值加 4，即 04h。

然后到了寄存器堆这里，根据之前提取到的 rs，rt 寄存器的编号，读取到 rs_value 和 rt_value 这俩操作数。

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
```

```
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;
```

然后根据指令确定要输入 ALU 的控制编码,即进行加法，并设置立即数信号。

```
assign sext_imm      = {{16{imm[15]}}, imm};// 立即数符号扩展
```

并且对立即数继续扩展。

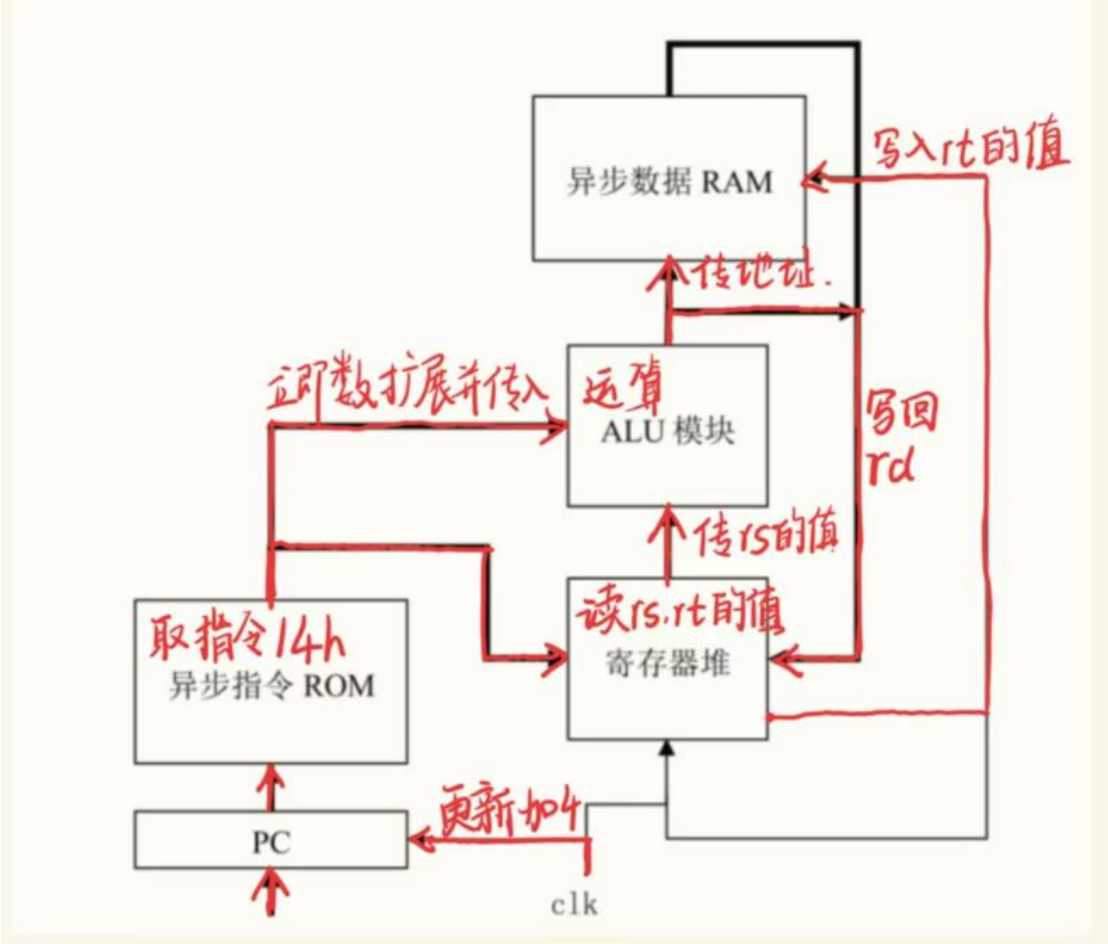
然后根据立即数信号设置两个操作数的值，一个是 rs 寄存器的值，另一个是立即数。传入 ALU 中，得到 ALU 运算的最后结果。这就是要取值的地址。

然后进入访存阶段，根据 ALU 算出来的地址，在数据存储器里写入新值。

```
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
```

写入的数据是 rt 寄存器里的值。本周期的这一条指令就结束了。

流程图如下：



分析 J 型指令

J 型指令即跳转指令。我分析 inst_rom[19] = 32'h08000000。

分析 inst_rom[19] = 32'h08000000:

首先从 PC 寄存器里读取地址 4ch，然后把指令转化为二进制：0000_1000_0000_0000_0000_0000_0000_0000。

操作码 op 是 000010。

```
assign inst_J      = (op == 6'b000010);                    // 直接跳转
```

根据这个 op 码，我直接知道了这是一个直接跳转指令。其跳转地址是 00000000000000000000000000（26 个 0）。

```
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};
```

然后设置直接跳转信号，计算目标地址为 00000000h。

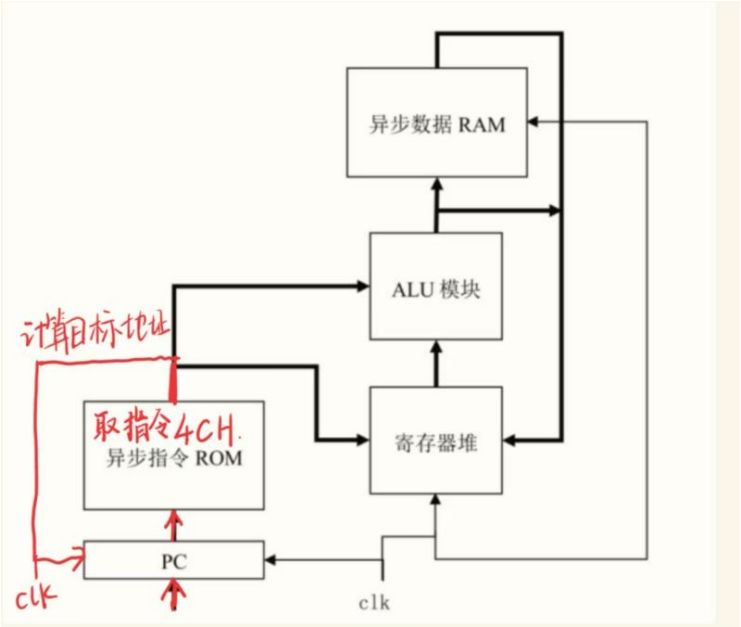
则这一条指令的意思是 j 00000000h

```
assign next_pc = jbr_taken ? jbr_target : seq_pc;
```

然后设置下一条 PC 的值是目标地址。

然后这一个周期这个指令就结束了，当时钟上升沿的时候，PC 会更新为新的 PC 值。

流程图如下：



添加三个指令提高实验

```
assign inst_SGT   = op_zero & sa_zero    & (funct == 6'b101011);//有符号大于置位
assign inst_SGTU   = op_zero & sa_zero    & (funct == 6'b101100);//无符号大于置位
assign inst_ANDOR = op_zero & sa_zero    & (funct == 6'b100101);//按位与非操作
```

如上图，首先在 single_cycle_cpu 模块里添加三个指令的信号。判断的 funct 码是我随机选择的，确保不冲突即可。并设置 ALU 的运算控制信号。

```
always @(*) begin
case (1'b1)
    inst_add:   alu_control = 4'b0001;
    inst_sub:   alu_control = 4'b0010;
    inst_slt:   alu_control = 4'b0011;
    inst_sltu:  alu_control = 4'b0100;
    inst_and:   alu_control = 4'b0101;
    inst_or:    alu_control = 4'b0110;
    inst_xor:   alu_control = 4'b0111;
    inst_nor:   alu_control = 4'b1000;
    inst_sll:   alu_control = 4'b1001;
    inst_srl:   alu_control = 4'b1010;
    inst_sra:   alu_control = 4'b1011;
    inst_lui:   alu_control = 4'b1100;
    inst_sgt:   alu_control = 4'b1101;
    inst_sgtu:  alu_control = 4'b1110;
    inst_andor: alu_control = 4'b1111;
endcase
nd
```

然后如上图，修改一下 alu_control 的判断条件。

```
assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI;
assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR|inst_ANDOR
                     | inst_OR   | inst_XOR  | inst_SLL | inst_SRL|inst_SGT|inst_SGTU;
```

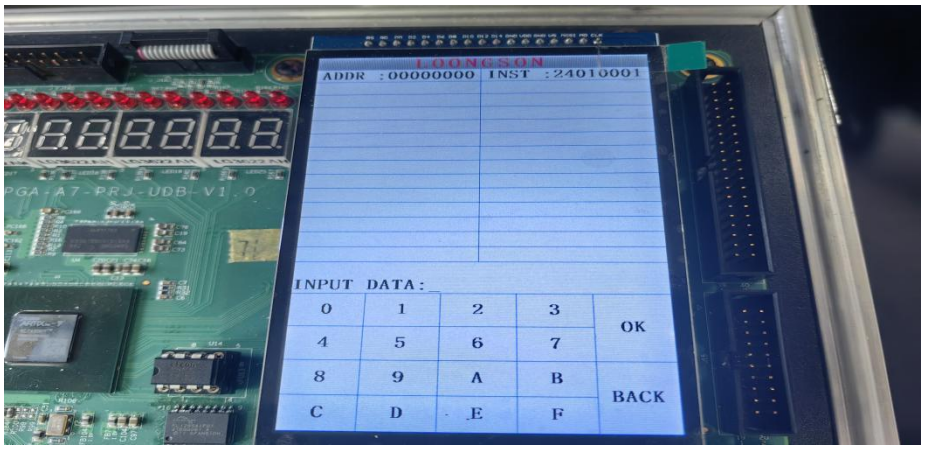
然后修改写入的判断条件，加上新的三个指令。

```
assign inst_rom[19] = 32'h00C7482B;//有符号大于置位sgt    $9 , $6, $7
assign inst_rom[20] = 32'h00C7482C;//无符号大于置位sgtu    $9 , $6, $7
assign inst_rom[21] = 32'h00C3382F;//与非andor     $7 , $6, $3
```

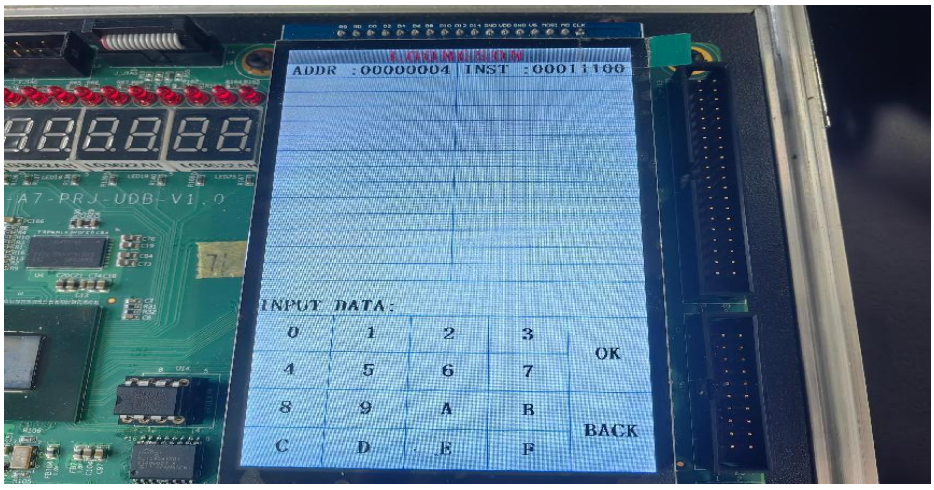
最后，在 ROM 里写入相应的指令即可。

五、 实验结果分析

同步 ROM 实验

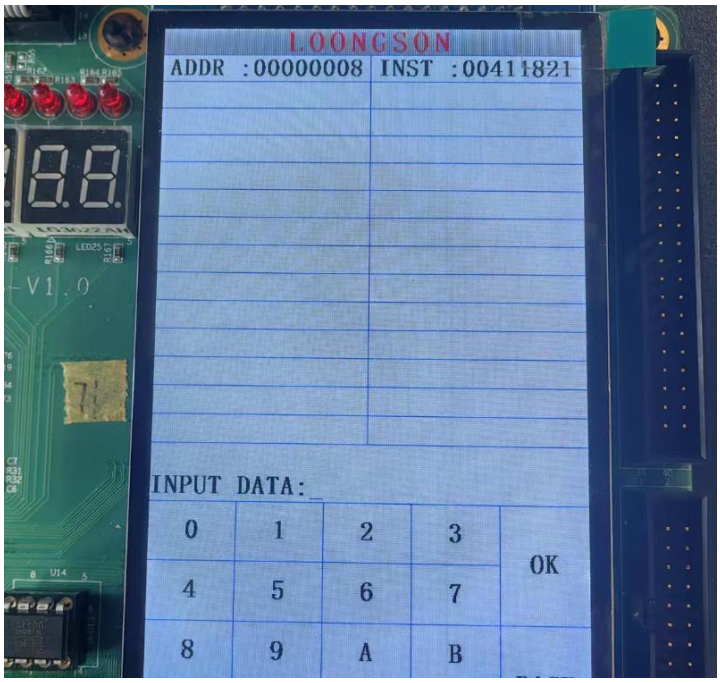


如上图，我输入的地址是 00000000h，这个地址都是字节地址。根据我在 coe 文件里写入的数据，在这个地址上的指令应该是 24010001h，而显示屏上的 inst 的只是 24010001h，答案正确。

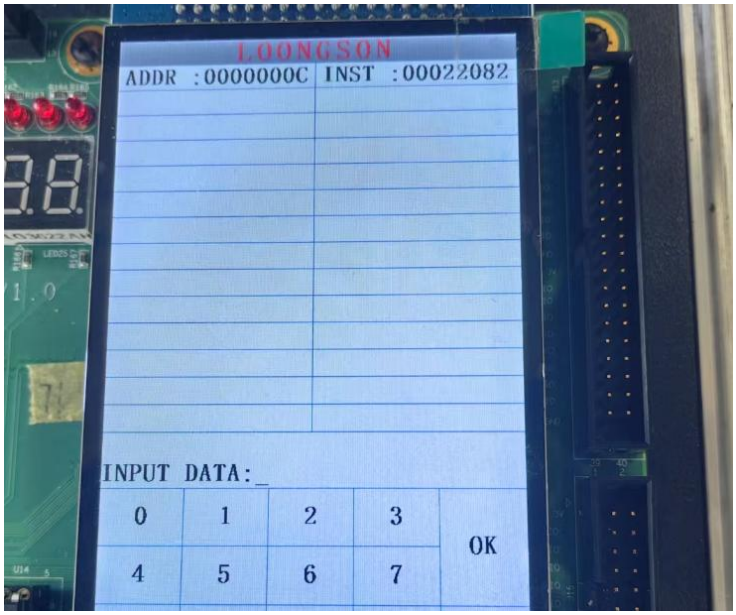


如上图，我输入的地址是 00000004h，根据我在 coe 文件里写入的数据，在这个地址上的指令应该是 00011100h，而显示屏上的 inst 的是 00011100h，答案正确。

异步 ROM 实验



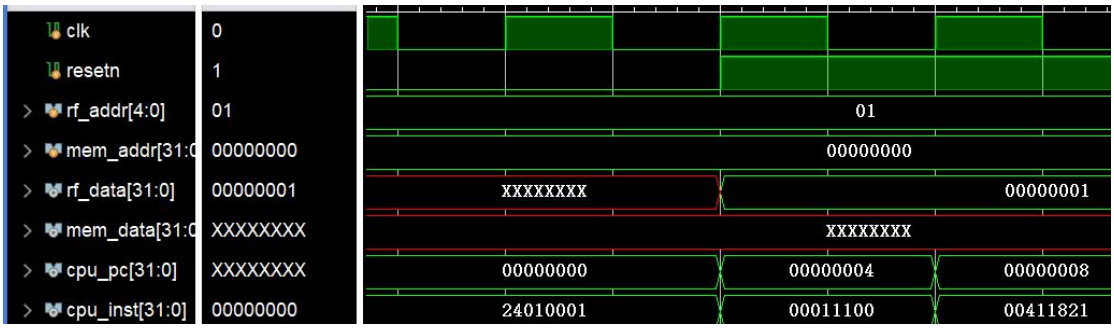
如上图，我输入的地址是 00000008h，根据我在 rom.v 文件里写入的数据，在这个地址上的指令应该是 00411821h，而显示屏上的 inst 的是 00411821h，答案正确。



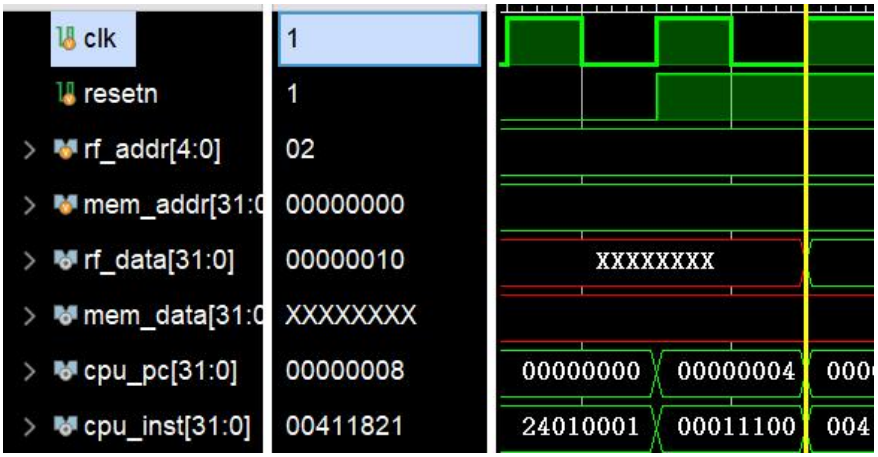
如上图，我输入的地址是 0000000Ch，根据我在 rom.v 文件里写入的数据，在这个地址上的指令应该是 00022082h，而显示屏上的 inst 的是 00022082h，答案正确。

单周期 cpu 实验

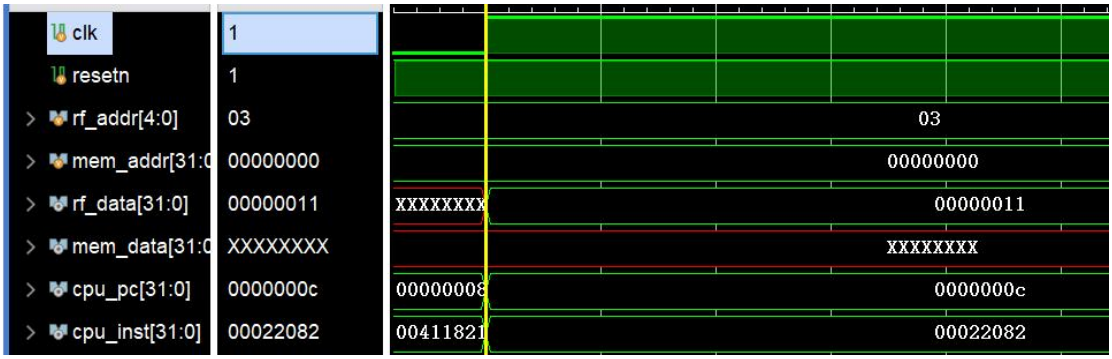
本次验证我使用 vivado 进行仿真实验，分析每一步指令及其运算结果。



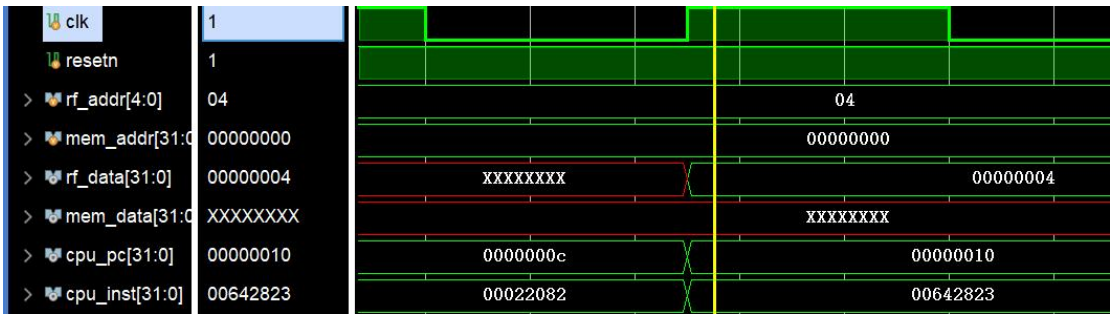
如上图，进行第一条指令 24010001，addiu $1 ,$0,#1 ，我读寄存器 1 的值，可以看见，当取消置位后，在取消置位的下一个时钟上升沿，$1 的值由原来的 X（不定值）变为了 1。即$0 的值 0 加上立即数 1，得到 1。结果正确。



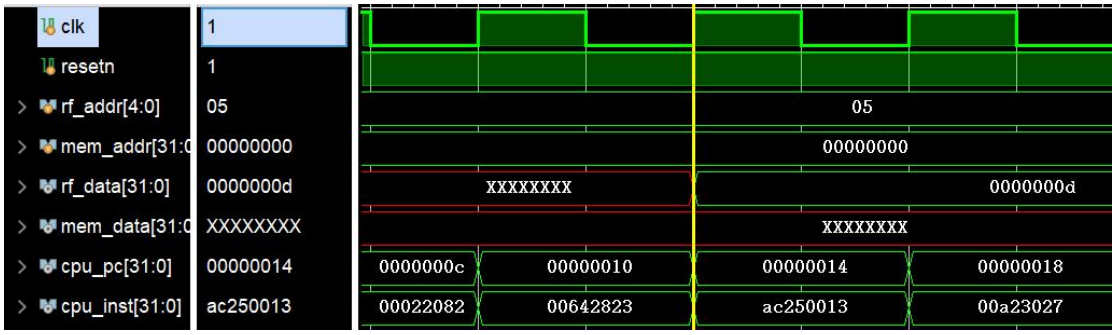
如上图，进行第 2 条指令 00011100，sll $2 ,$1,#4 ，我读寄存器 2 的值，可以看见，在下一个时钟上升沿，$2 的值变为了 00000010h。即$1 的值 00000001 左移 4 位，得到 16(十进制)。结果正确。



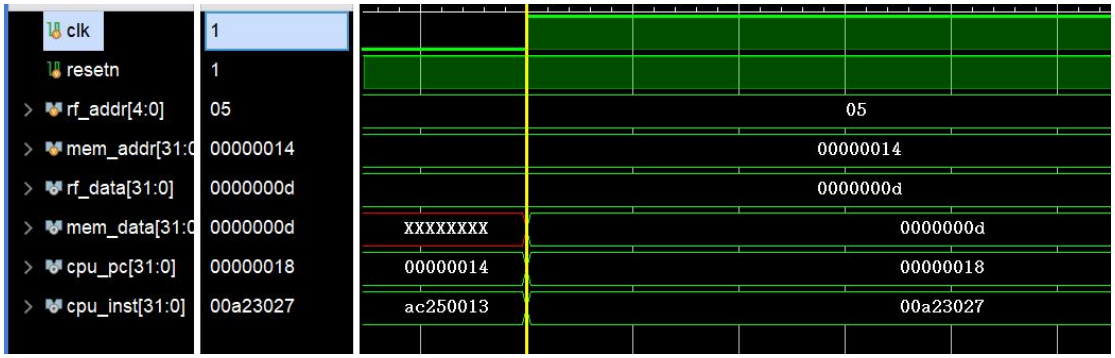
如上图，进行第 3 条指令 00411821，addu $3,$2,$1，我读寄存器 3 的值，可以看见，在下一个时钟上升沿，$3 的值变为了 00000011h。即$1 的值加上$2 的值，结果正确。



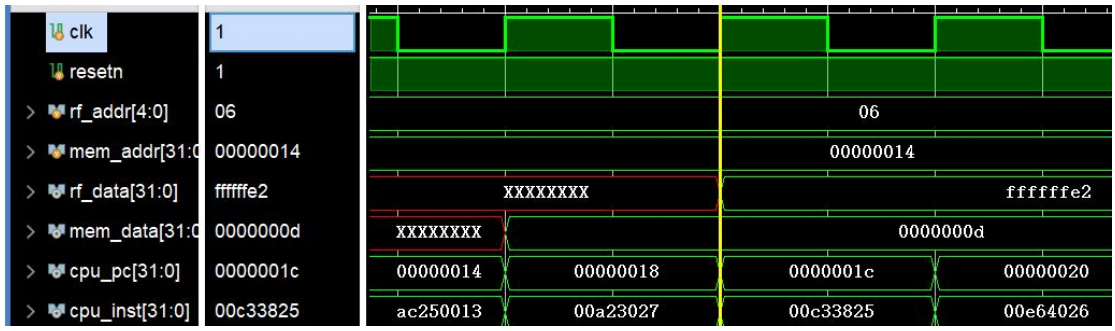
如上图，进行第 4 条指令 00022082，srl　$4 ,$2,#2，我读寄存器 4 的值，可以看见，在下一个时钟上升沿，$4 的值变为了 00000004h。即$2 的值 00000010h 的值右移两位，结果正确。



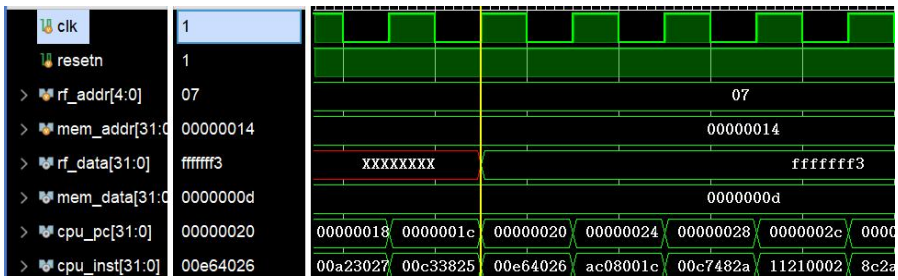
如上图，进行第 5 条指令 00642823，subu　$5 ,$3,$4 我读寄存器 5 的值，可以看见，在下一个时钟上升沿，$5 的值变为了 0000000dh。即$3 的值 00000011h 减去$4 的值 00000004h，结果正确。



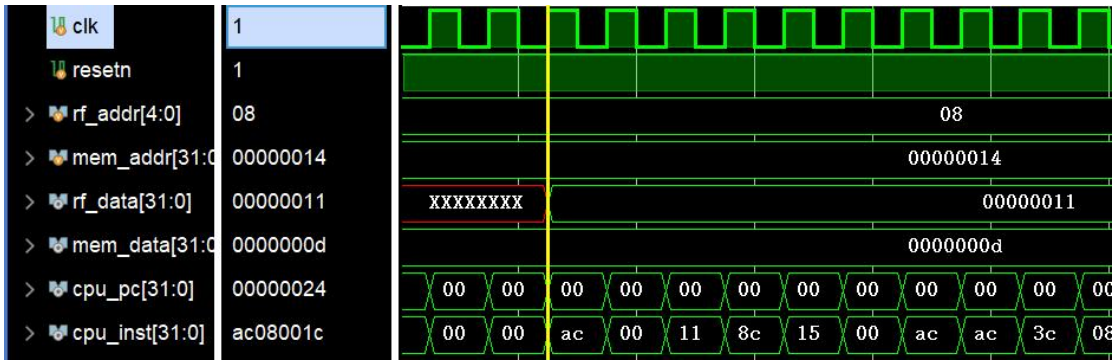
如上图，进行第 6 条指令 ac250013，sw　　$5 ,#19($1)，把$5 的值写入到数据存储器里地址为$1+19 的寄存器，我读这个寄存器的值，可以看见，在下一个时钟上升沿，此处的的值变为了 0000000dh。即$5 的值，结果正确。



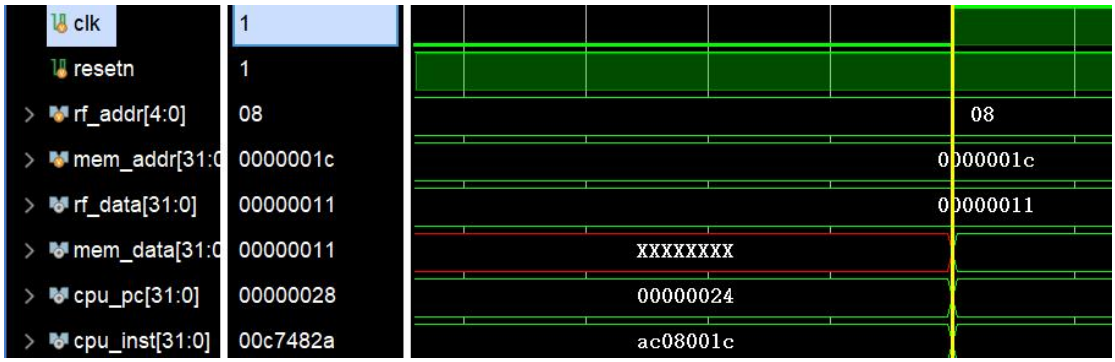
如上图，进行第 7 条指令 00a23027，nor　　$6 ,$5,$2，　我读寄存器 6 的值，可以看见，在下一个时钟上升沿，寄存器 6 的的值变为了 fffffffe2h。即$5 的 0000000d 和$2 的 00000010h 进行按位非或的值，结果正确。



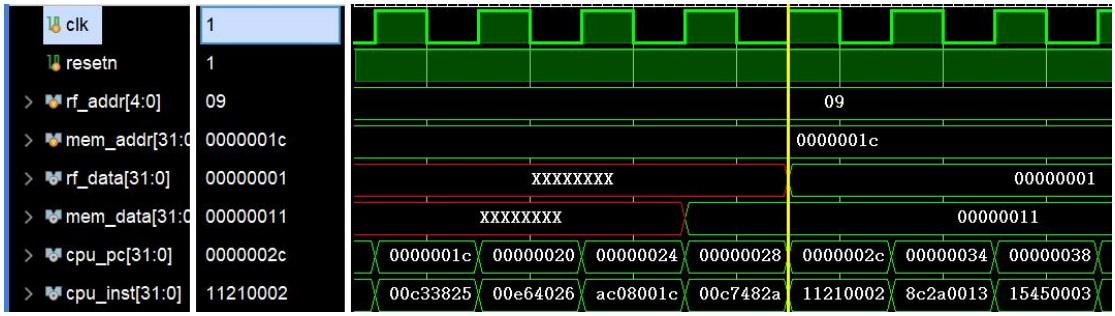
如上图，进行第 8 条指令 00c33825，or　　$7 ,$6,$3，　我读寄存器 7 的值，可以看见，在下一个时钟上升沿，寄存器 7 的的值变为了 fffffff3h。即$6 和$3 进行按位或的值，结果正确。



如上图，进行第 9 条指令 00e64026，xor　　$8 ,$7,$6 ，　我读寄存器 8 的值，可以看见，在下一个时钟上升沿，寄存器 8 的的值变为了 00000011h。即$7 的和$6 的进行按位异或的值，结果正确。



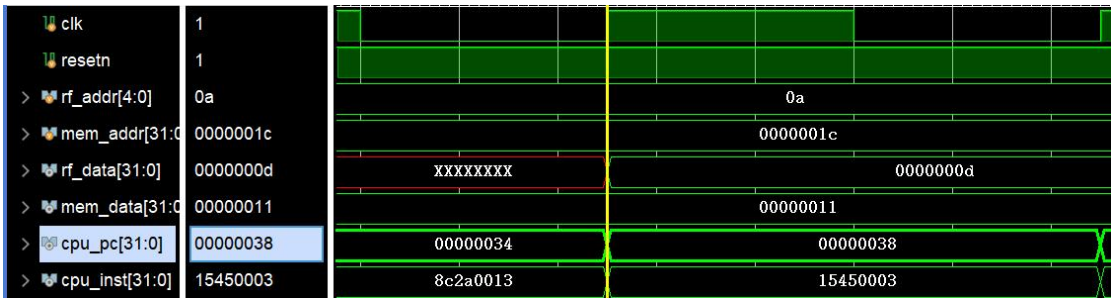
如上图，进行第 10 条指令 ac08001c，$8 ,#28($0)，把$8 的值写入到数据存储器里地址为$0+28 的寄存器，我读这个寄存器的值，可以看见，在下一个时钟上升沿，此处的的值变为了 00000011h。即$8 的值，结果正确。



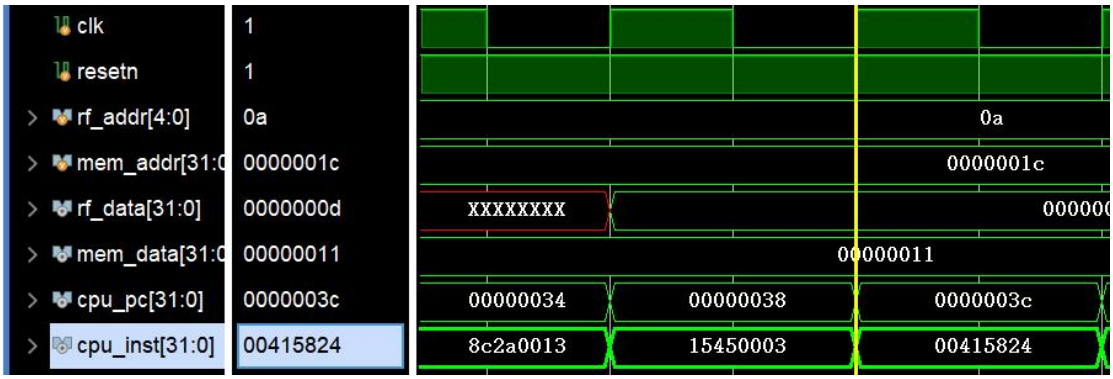
如上图，进行第 11 条指令 00c7482a，slt　　$9 ,$6,$7 ，进行小于置位比较。 我读寄存器 9 的值，可以看见，在下一个时钟上升沿，寄存器 9 的的值变为了 00000001h。即$6 的 fffffffe2h 小于和$7 的 fffffff3h，进行置位，结果正确。

进行第 12 条指令 11210002，beq　　$9 ,$1,#2 ，　寄存器 1 和 9 的值是一样的。说明要进行跳转，跳到第 14 个指令。Cpu_pc 变成了 34h，代表第 14 条指令，结果正确

第 13 条指令不执行。

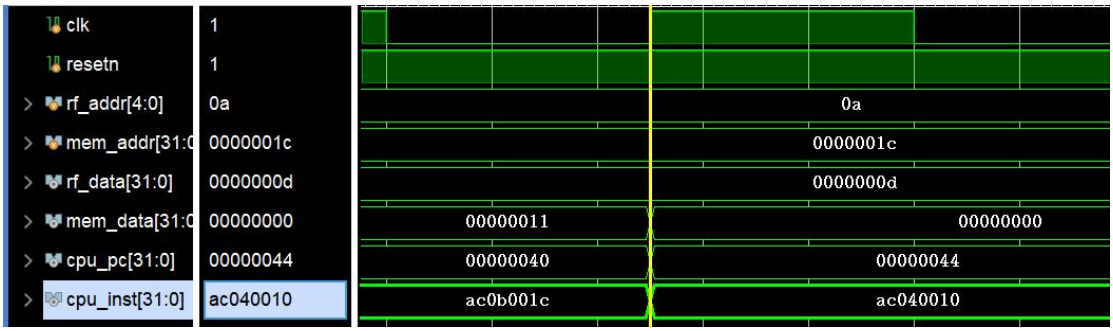


如上图，进行第 14 条指令 8C2A0013，lw　$10,#19($1)。读取地址是$1+19 的数据存储器的值，并存到$10，我读寄存器 10 的值，可以看见，在下一个时钟上升沿，寄存器 10 的的值变为了 0000000dh，取值成功。结果正确。



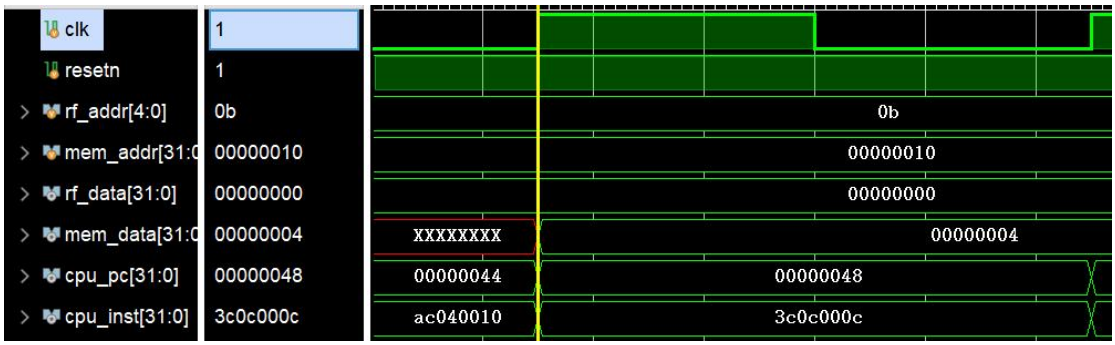
如上图，进行第 15 条指令 15450003，bne $10,$5,#3 进行有条件跳转。$10 和$5 相等，说明不跳转，继续顺序执行，pc 是结果正确的。

如上图，进行第 16 条指令 00415824，and　$11,$2,$1，进行按位与置位比较。 我读寄存器 11 的值，可以看见，在下一个时钟上升沿，寄存器 11 的的值变为了 00000000h。即$2 的 10h 和$1 的 01h 按位与运算，结果正确。

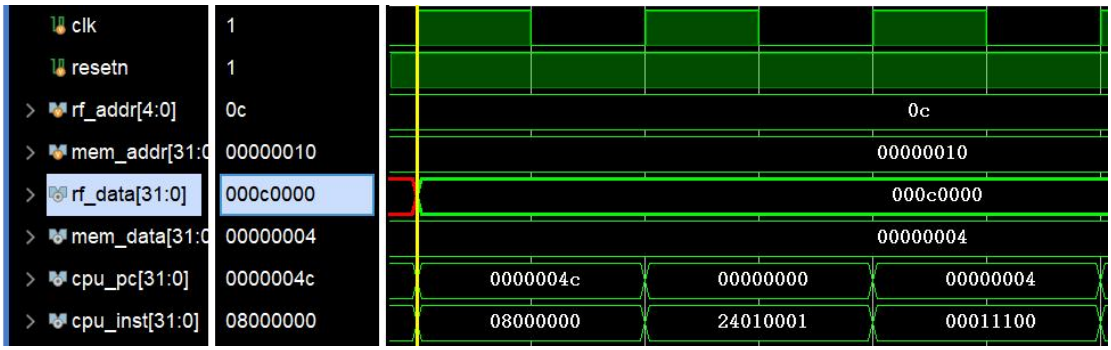


如上图，进行第 17 条指令 AC0B001C，sw　$11,#28($0)，把$11 的值写入到数据存储器

里地址为$0+28 的寄存器，我读这个寄存器的值，可以看见，在下一个时钟上升沿，此处的的值变为了 00000000h。即$11 的值，结果正确。



如上图，进行第 18 条指令 AC040010，sw　$4 ,#16($0)，把$4 的值写入到数据存储器里地址为$0+16 的寄存器，我读这个寄存器的值，可以看见，在下一个时钟上升沿，此处的的值变为了 00000004h。即$4 的值，结果正确。



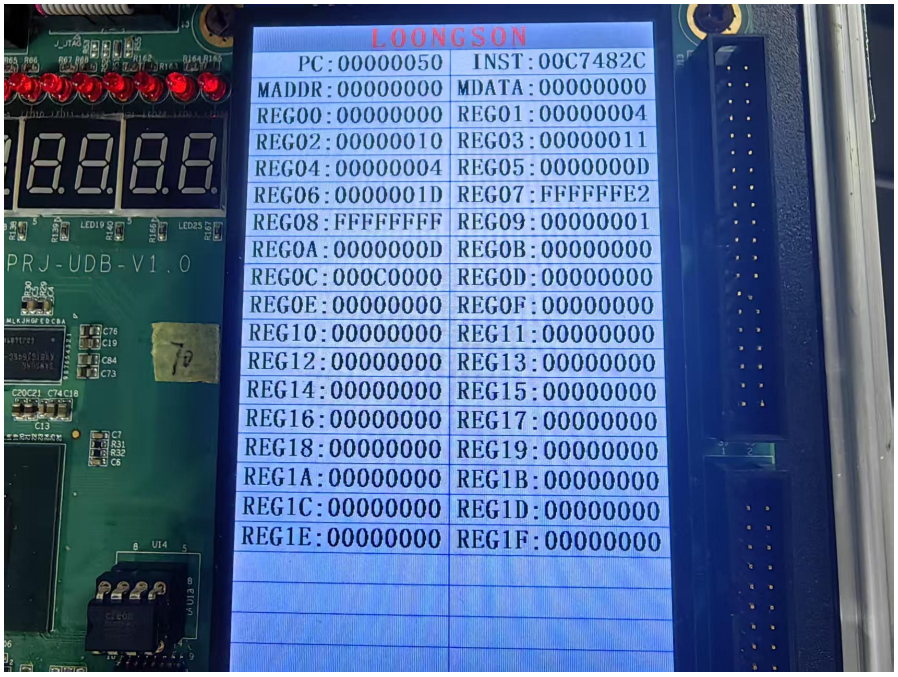
如上图，进行第 19 条指令 3C0C000C，lui　$12,#12，把立即数 12 加载到$12 的高 16 位，低 16 位补 0。我读寄存器 12 的值，可以看见，在下一个时钟上升沿，寄存器 12 的的值变为了 00000001h。结果正确。

如上图，进行第 20 条指令 08000000，j　00H，进行无条件跳转。可以看见，下一个 PC 的值变为了 00000000h，成功进行跳转，结果正确。

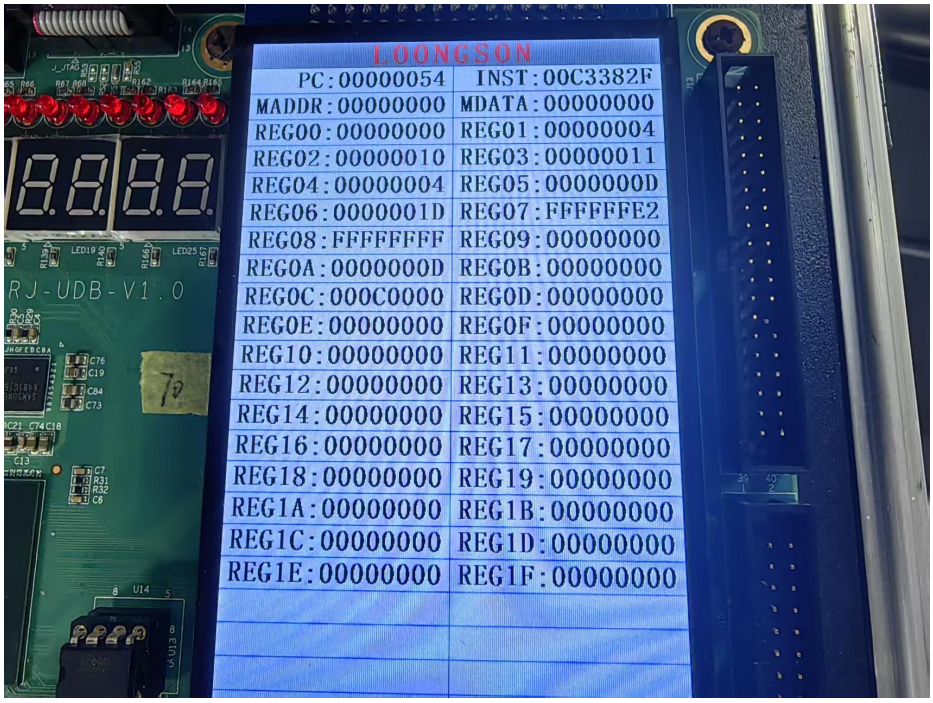
接下来就一直进行上边 20 条指令的循环。仿真结果分析结束。

添加三个指令提高实验

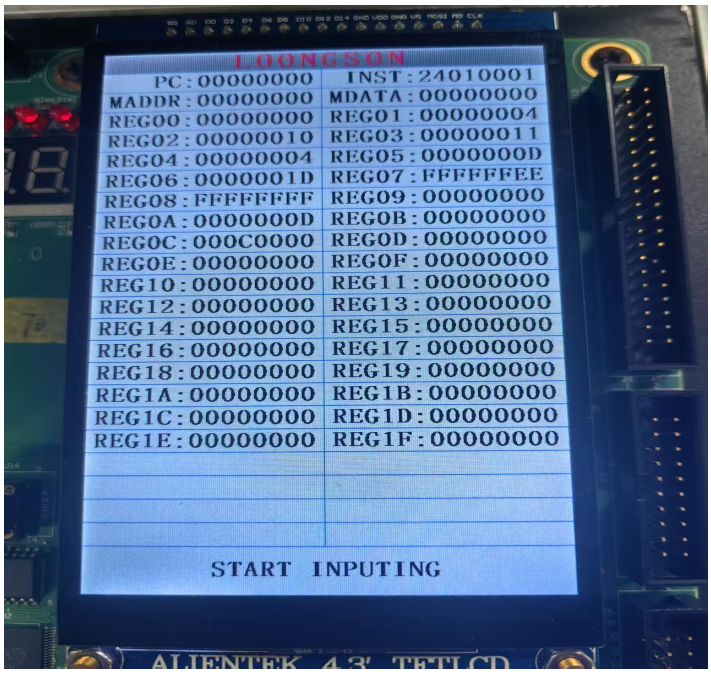
我使用实验箱进行验证，实验结果如下：



如上图，此时进行的是有符号大于置位运算，sgt　$9 ,$6,$7 ，即比较$6,$7，若$6 大于$7，则令$9 为 1，否则为 0。如上图，$6 的值是 1D，$7 的值是 ffffffe2,显然$7 是负数，$6 更大，因此应该置位，我也观察到$9 为 1，结果正确。



如上图，此时进行的是无符号大于置位运算，sgtu　$9 ,$6,$7 ，即比较$6,$7，若$6 大于$7，则令$9 为 1，否则为 0。如上图，$6 的值是 1D，$7 的值是 ffffffe2,显然$7 大于$6，因此不应该置位，我也观察到$9 为 0，结果正确。



如上图是进行了与非运算，andor　　$7 ,$6,$3，计算$6,$3 按位与非的值，然后存到$7 里边，$6 是 0000001D，$3 是 00000011，先与运算，得到 00000011，然后进行非运算，得到的结果是 ffffffee，我观察到$7 此时的值是 ffffffee，答案正确。

综上，添加的三个指令正常运行，答案正确。

六、 总结感想

ROM 实验总结收获

本次实验我进行了同步 ROM 实验和异步 ROM 实验。

通过本次 ROM 实验，我掌握了只读存储器的基本结构、工作方式。实验过程中，我学会了如何使用 Verilog 设计同步和异步 ROM 模块，理解了 ROM 中数据的预定义与不可写特性，并掌握了基于地址访问数据的基本原理。

实验为我今后进一步学习存储系统以及更复杂的开发奠定了坚实的实践基础。这次实验不仅提升了我的理论知识运用能力，也增强了独立分析问题和解决问题的工程思维。

ROM 和 RAM 的对比分析

1. 功能不同

ROM 是只读存储器，即 ROM 只能实现读的功能，而读取的数据必须提前写入好，而在程序运行的时候，是不能向 ROM 存储器里进行写入的。例如在上边的 ROM 实验里，我必须提前在 ROM 里写好指令，后续不能再写入。

RAM 则不同，它既可以读，也可以写，两种功能都可以实现。

2. 时序不同

对于同步 ROM 和同步 RAM 来说，对它们进行的操作必须依赖时钟，我进行的实验里，只用当时钟上升沿的时候，同步 RAM 才会进行读写操作。同步 ROM 才会进行读操作。

而异步 ROM 和异步 RAM 则没有这种依赖，它们执行功能时不依赖时钟，只要地址稳定，那么立即进行读或写操作。

单周期 CPU 实验总结收获

在本次单周期 CPU 实验中，我深入理解了处理器各模块，如指令译码、寄存器堆、ALU、数据存储器的协同工作原理，掌握了 Verilog 硬件描述语言的实际应用，加深了对控制信号生成和数据通路设计的理解。

通过调试单周期 CPU，我加深了对 MIPS 指令执行流程的认识，尤其是在程序计数器更新、分支跳转、立即数扩展等方面的处理逻辑。

此次实验不仅巩固了课程理论知识，也为后续学习多周期、流水线 CPU 等复杂体系结构奠定了坚实的基础。