《计算机组成原理实验》

实验报告

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	计算机类 教务一班
学 生 姓 名 ：	关雅雯
学　　号 ：	18340045
时　　间 ：	2019 年 12 月 4 日

成 绩 ：

单周期CPU设计与实现

一. 实验目的

1. 理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现
2. 通过对单周期CPU 的运行状况进行观察和分析，进一步加深理解

二. 实验内容

1. CPU的设计

该部分是处理器的总体设计，确定处理器由哪些部分组成：运算器(ALU)，寄存器(Reg)，控制单元(CU)；定义各个期间的控制信号以及控制方法。处理器的设计要结合存储器(Mem)单元进行设计，确定控制单元对存储器的控制信号及控制方法。

2. 仿真与测试

对各个功能部件进行仿真测试，进行 FPGA 测试，数码管显示运算结果，至少反应 4 种指令的运行结果（R 型、条件跳转、lw、ori 等）。

具体实验任务如下：

利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，用 Verilog HDL 语言或 VHDL 语言来编写，实现单周期 CPU 的设计，这个单周期 CPU 至少能够完成 20 条 MIPS 指令，至少包含以下指令：

- 支持基本的内存操作如 lw，sw 指令
- 支持基本的算术逻辑运算如 add，sub，and，ori，slt，addi 指令
- 支持基本的程序控制如 beq，j 指令

将其中的 alu 运算结果在开发板数码管上显示出来。

另外可拓展添加其他指令。

MIPS 的指令格式为 32 位。图 1 给出了 MIPS 指令的 3 种格式。

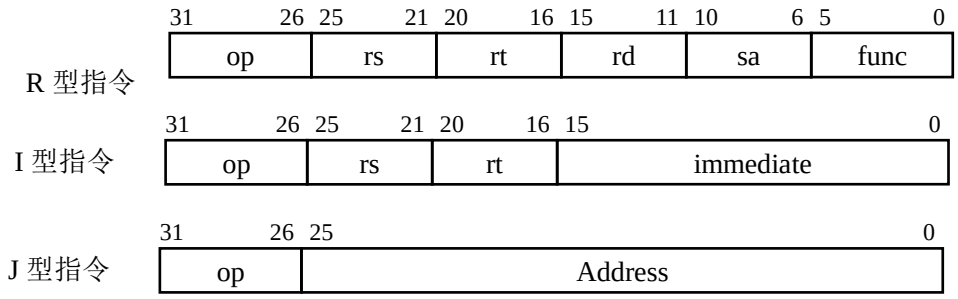


图 1 MIPS32™ 指令格式

其中Rs和Rt为两个源操作数寄存器，Rd为目的操作数寄存器。shamt为移位操作时的移位运算值，是一个立即数。func为R型指令的功能码。imme为I型指令的立即数。dest为J型指令的跳转地址。

表 1 MIPS 的 31 种指令

助记符	指	令	格	式			示 例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	**op**	**rs**	**rt**	**rd**	**shamt**	**func**			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2&S3	(rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2|S3	(rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2^S3	(rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1= ~($2 | S3)	(rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3)$1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs=$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs=$2，rt=$3, rd=$1, 无符号数
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (算术右移，注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	(rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)←(rs)
I-类型	**op**	**rs**	**rt**	**immediate**					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2&10	(rt)←(rs)&(zero-extend)immediate,rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2|10	(rt)←(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2^10	(rt)←(rs)^(zero-extend)immediate,rt=$1,rs=$2

lui	001111	00000	rt	immediate	lui $1,10	$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset	lw $1,10($2)	$1=Memory [$2+10]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset	sw $1,10($2)	Memory [$2+10] =$1	Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset	beq $1,$2,40	if($1=$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1, rt=$2
bne	000101	rs	rt	offset	bne $1,$2,40	if($1≠$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=$1, rt=$2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
J-类型	**op**	**address**					
j	000010	address			j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4
jal	000011	address			jal 10000	$31=PC+4 goto 10000	($31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4

补充指令（共8条）：

R-类型	**op**	**rs**	**rt**	**rd**	**shamt**	**func**			
mult	000000	rs	rt	00000	00000	011000	mult $1, $2	$1 * $2	$hi = $1 * $2 高 16 位，$lo = $1 * $2 低 16 位
div	000000	rs	rt	00000	00000	011010	div $1, $2	$1 / $2	$hi = $1 / $2，$lo = $1 % $2
mfhi	000000	00000	00000	rd	00000	010000	mfhi $1	$1 = $hi	$1 = $hi
mflo	000000	00000	00000	rd	00000	010010	mflo $1	$1 = $lo	$1 = $lo
I-类型	**op**	**rs**	**rt**	**immediate**					
sh	101001	rs	rt	offset			sh $1, 8($2)	Memory [$2+8] = $1	Memory[(rs)+(sign_extend(offset))←(rt) rt=$1,rs=$2 (半字)
sb	101000	rs	rt	offset			sb $1, 8($2)	Memory [$2+8] = $1	Memory[(rs)+(sign_extend(offset))←(rt) rt=$1,rs=$2 (字节)
lh	100001	rs	rt	offset			lh $1, 8($2)	$1=Memory [$2+8]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2 (半字)
lb	100000	rs	rt	offset			lb $1, 8($2)	$1=Memory [$2+8]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2 (字节)

本实验共实现了以上39条不同指令。

三. 实验原理

单周期 CPU 在每个 CLK 上升沿时更新 PC，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。其时序示意如图 2。

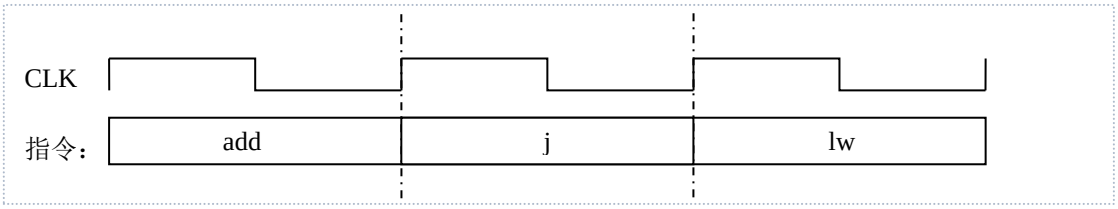


图 2 单时钟周期 CPU 时序示意图

下图是一个单周期 CPU 的顶层结构实现。主要器件有程序计数器 PC、程序存储器、寄存器堆、ALU、数据存储器和控制部件等。所有的控制信号简单地说明如下：

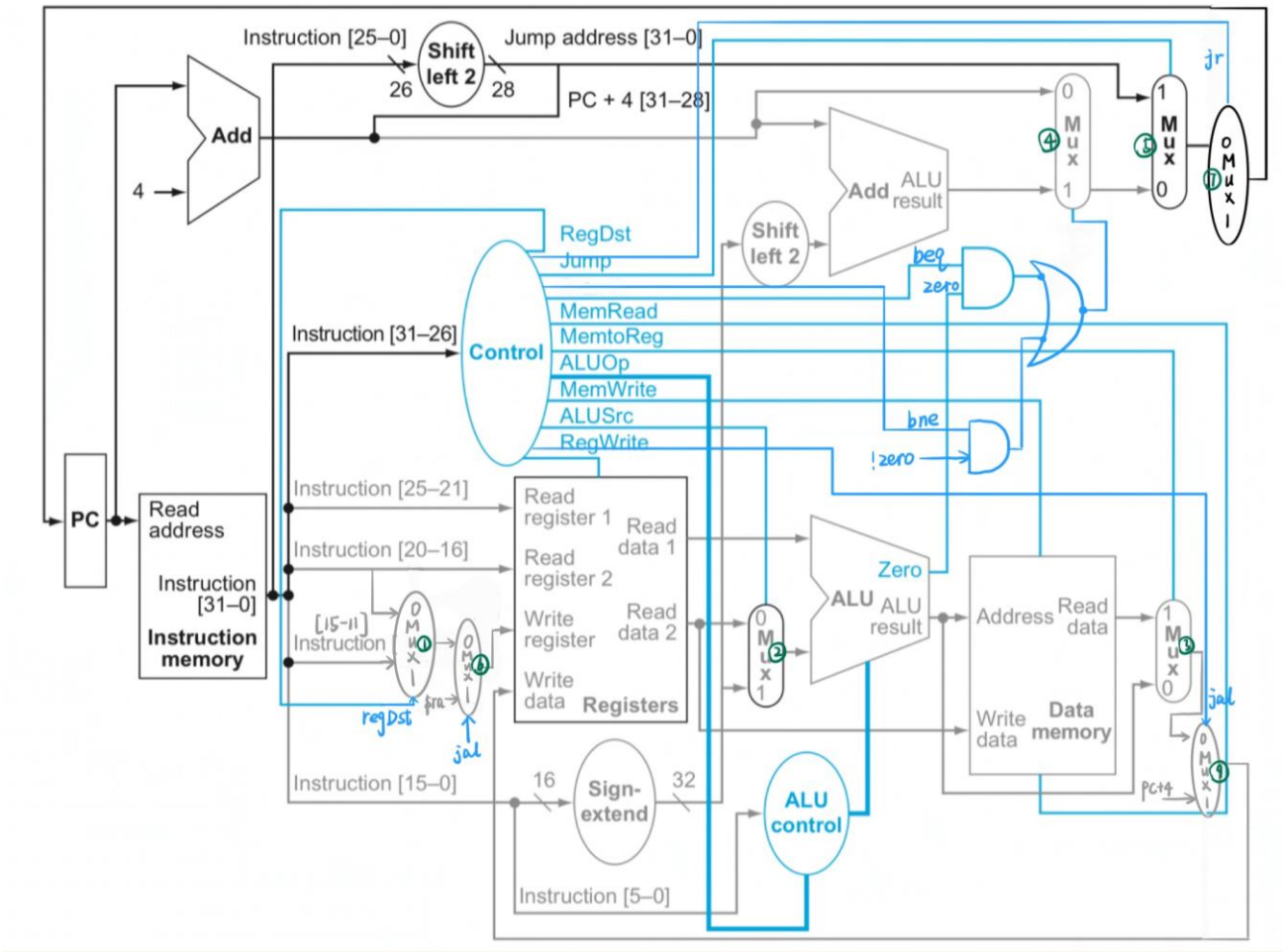


图 3 单时钟周期 CPU 详细逻辑设计图

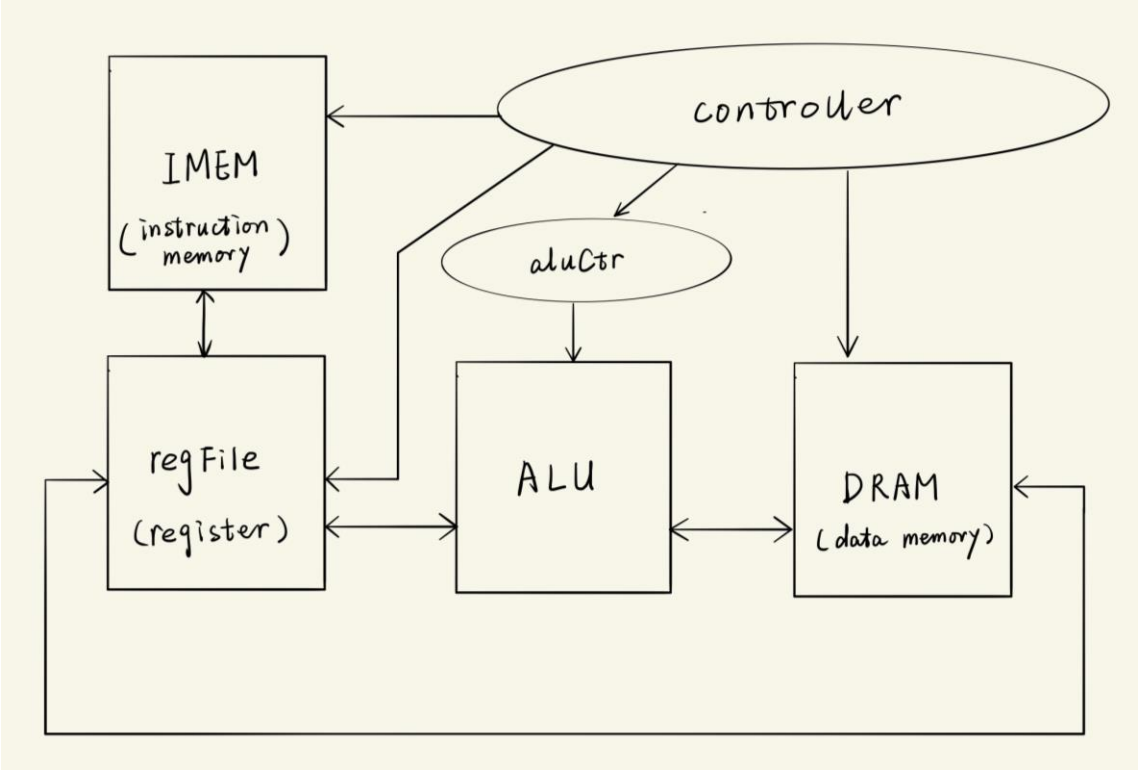
四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

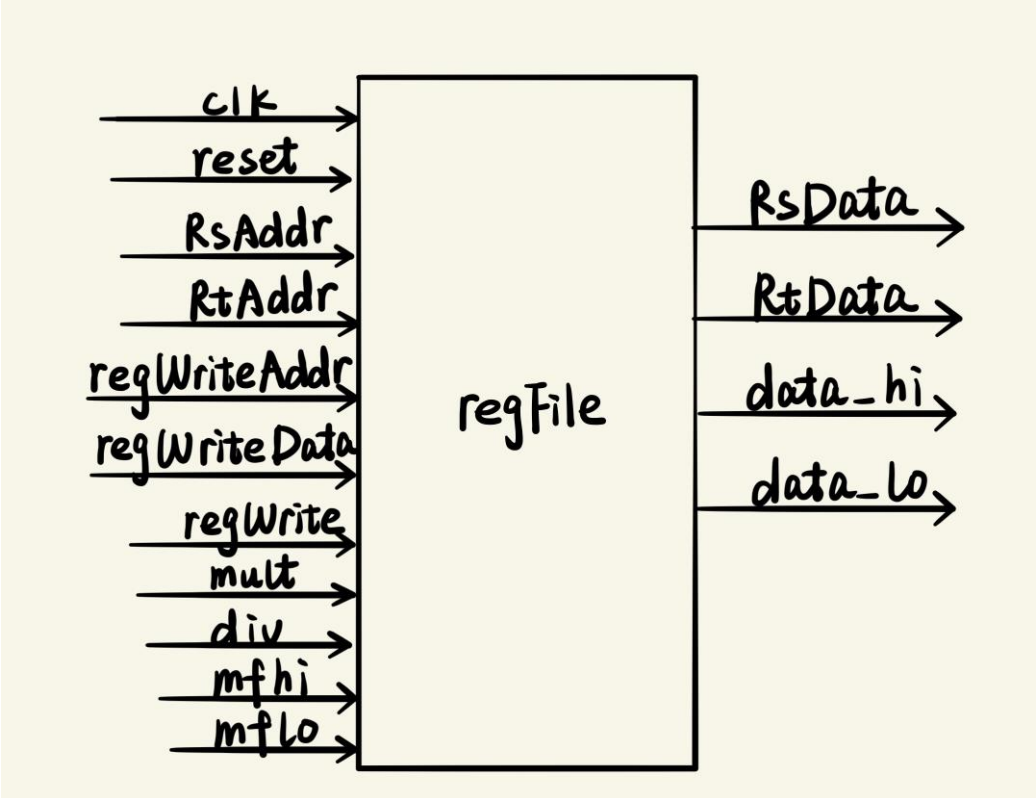
1. CPU设计

CPU总设计：



另外有top模块总管上图中所有模块。

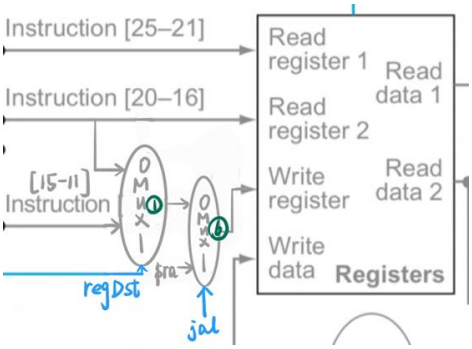
（1）regFile（寄存器组）模块设计



寄存器组是指令操作的主要对象，MIPS处理器里一共有32个32位的寄存器，故可以声明一个包含32个32位的寄存器数组。读寄存器时需要Rs，Rd的地址（RsAddr，RtAddr），得到其数据（RsData，RtData）。写寄存器Rd时需要所写地址(regWriteAddr)，所写数据（regWriteData），同时需要写使能（regWrite）。以上所有操作需要在时钟（clk）和复位信号（reset）控制下操作。

为了实现mult、div、mfhi、mflo指令，加入四个标志，并支持读出hi、lo寄存器的值。

为了支持jal指令，设计复用器如下：



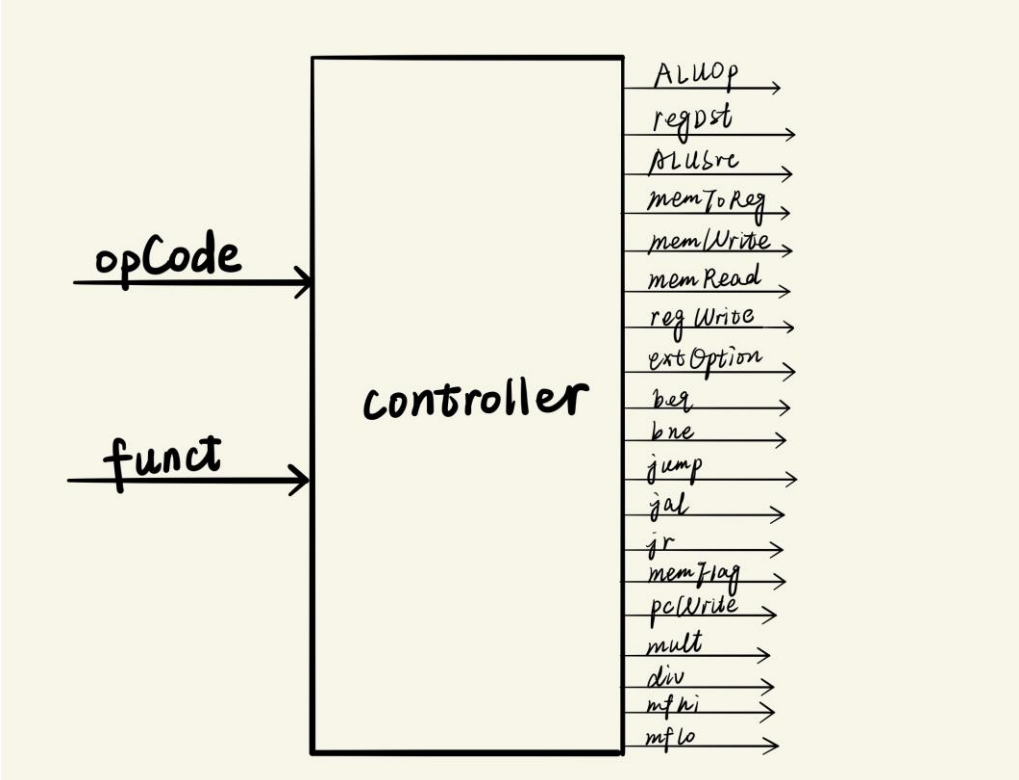
具体代码以及注释如下：

```
`timescale 1ns / 1ps

module regFile(
    input clk,                  //时钟信号
    input reset,                //复位信号
    input [4:0] RsAddr,         //哪个寄存器作为 Rs 寄存器
    input [4:0] RtAddr,         //哪个寄存器作为 Rt 寄存器
    input [4:0]  regWriteAddr,  //写寄存器时寄存器的地址（即写哪个寄存器）
    input [31:0] regWriteData,  //写寄存器的值
    input regWrite,             //写寄存器使能
    input mult,                 //mult 标志
    input div,                  //div 标志
    input mfhi,                 //读 hi 寄存器标志
    input mflo,                 //读 lo 寄存器标志
    input [31:0] data_hi,       //要写入 hi 寄存器的值
    input [31:0] data_lo,       //要写入 lo 寄存器的值
    output [31:0] RsData,       //Rs 寄存器的值
    output [31:0] RtData        //Rt 寄存器的值
);

//寄存器地址都是 5 位二进制数，
//寄存器有 32+2=34 个（加上 hi、lo）
reg[31:0] regs[0:33];                   //寄存器组

//  根据地址读出 Rs、Rt 寄存器数据
assign RsData = (RsAddr == 5'b0) ? 32'b0 : regs[RsAddr];
assign RtData = (RtAddr == 5'b0) ? 32'b0 : regs[RtAddr];

integer i;

always @( posedge clk )                 //时钟上升沿操作
begin
    if(!reset)
    begin
```

```
        if(regWrite == 1)              //写使能信号为 1 时写操作
        begin
            if(mult == 1  || div == 1)//乘除法，写入 hi、lo 寄存器
            begin
                regs[32] = data_hi;
                regs[33] = data_lo;
            end
            else if(mfhi) regs[regWriteAddr] = regs[32];//mfhi
            else if(mflo) regs[regWriteAddr] = regs[33];//mflo
            else regs[regWriteAddr] = regWriteData;     //写入数据
        end

    end
    else
    begin
        for(i = 0; i <= 33; i = i + 1) regs[i] = 0; //重置时所有寄存器赋值为 0，复位
    end
end

endmodule
```

（2）controller（控制器）模块设计

控制器输入指令的操作码opCode段，输出各个复用器、存储器读写等信号，控制数据通路的正常进行。为了支持拓展指令，输入端增加了结合funct段。



具体信号及注释如下：

```
module controller(
    input [5:0] opCode,          //指令的opCode段
    input [5:0] funct,           //指令的funct段
    output reg [3:0] ALUOp,      //ALUOp信号，用于结合funct控制ALUCtr信号
    output reg regDst,           //1：使用rd寄存器；0：使用rt寄存器
    output reg ALUSrc,           //1：存在立即数；0：不存在立即数
    output reg memToReg,         //1：从DataMemory向寄存器写入数据
    output reg memWrite,         //1：写DataMemory使能信号
    output reg memRead,          //1：读DataMemory使能信号
    output reg regWrite,         //1：写寄存器使能信号
    output reg extOption,        //1：符号扩展；0：0扩展
    output reg beq,              //beq指令标志
    output reg bne,              //bne指令标志
    output reg jump,             //j指令标志
    output reg jal,              //jal指令标志
    output reg jr,               //jr指令标志
    output reg[1:0] memFlag,     //11:word; 01: half word; 00: byte
    output pcWrite,              //写pc使能信号
    output reg mult,             //mult指令标志
    output reg div,              //div指令标志
    output reg mfhi,             //mfhi指令标志
    output reg mflo              //mflo指令标志
);
```

对pcWrite（PC写使能信号）的处理：

```
    assign pcWrite = (opCode == 6'b111111) ? 0 : 1;
```

若为停机指令，则pcWrite为0，否则为1；

对R型指令相关信号的处理：

```
always @(opCode,funct)
begin
    case(opCode)
        6'b000000: // R型指令
        begin
            ALUOp = 4'b0111;
            regDst = 1;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;

            if(funct==6'b001000)  jr = 1;  //jr
            else if(funct==6'b011000) // mult
                    mult = 1;
            else if(funct==6'b011010) //div
                    div = 1;
            else if(funct==6'b010000) //mfhi
                    mfhi = 1;
            else if(funct==6'b010010) //mflo
                    mflo = 1;
        end
```

对其它信号的处理：

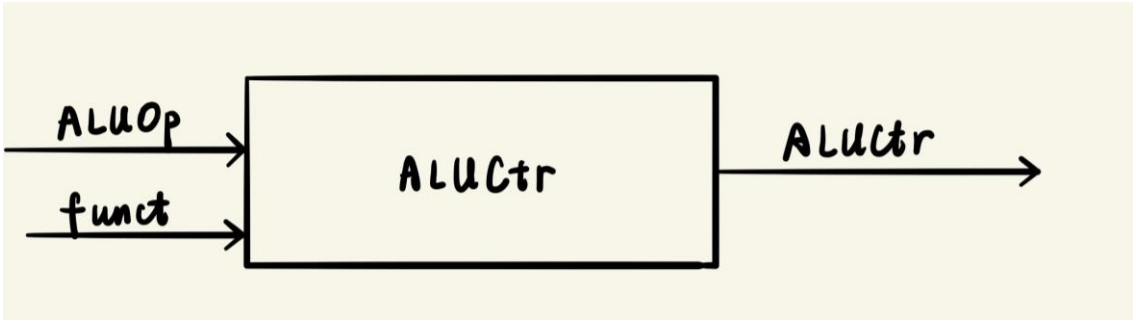
```
always @(opCode,funct)
begin
    case(opCode)
        6'b000000: // R 型指令
        begin
            ALUOp = 4'b0111;
            regDst = 1;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;

            if(funct==6'b001000)  jr = 1;  //jr
            else if(funct==6'b011000) // mult
                    mult = 1;
            else if(funct==6'b011010) //div
                    div = 1;
            else if(funct==6'b010000) //mfhi
                    mfhi = 1;
            else if(funct==6'b010010) //mflo
                    mflo = 1;
        end
        6'b001000: // addi 指令
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001001: // addiu 指令
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001100: // andi 指令
        begin
            ALUOp = 4'b0010;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 0;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
    6'b001101: // ori 指令
        begin
            ALUOp = 4'b0011;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 0;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001110: // xori 指令
        begin
            ALUOp = 4'b0100;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 0;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001111:    //lui
        begin
            ALUOp = 4'b0110;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b100011: // lw
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 1;
            memRead = 1;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b100001: // lh
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 1;
            memRead = 1;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b01;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b100000: // lb
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 1;
            memRead = 1;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b00;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b101011: // sw
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 1;
            regWrite = 0;
            extOption = 1;
            beq = 0;
```

```
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b101001: // sh
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 1;
            regWrite = 0;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b01;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b101000: // sb
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 1;
            regWrite = 0;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b00;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b000100: //beq 指令
        begin
            ALUOp = 4'b0001;
            regDst = 0;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 0;
            extOption = 1;
            beq = 1;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;
```

```
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b000101: //bne 指令
        begin
            ALUOp = 4'b0001;
            regDst = 0;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 0;
            extOption = 1;
            beq = 0;
            bne = 1;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001010: // slti 指
令
        begin
            ALUOp = 4'b0101;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b001011: //sltiu
        begin
            ALUOp = 4'b0101;
            regDst = 0;
            ALUSrc = 1;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
```

```
            mflo = 0;
        end
        6'b000010: // jump
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 0;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 1;
            jal = 0;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        6'b000011: // jal
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 1;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 1;
            jr = 0;
            memFlag <= 2'b11;

            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
        default: // 缺省值
        begin
            ALUOp = 4'b0000;
            regDst = 0;
            ALUSrc = 0;
            memToReg = 0;
            memRead = 0;
            memWrite = 0;
            regWrite = 0;
            extOption = 1;
            beq = 0;
            bne = 0;
            jump = 0;
            jal = 0;
            jr = 0;
            mult = 0;
            div = 0;
            mfhi = 0;
            mflo = 0;
        end
    endcase
end
```

（3）aluCtr（ALU控制器）模块设计



ALU 功能真值表

指令	ALUop	func	ALUCtr	功能描述
R	0111	100000	00000	add
R	0111	100001	00001	addu
R	0111	100010	00010	sub
R	0111	100011	00011	subu
R	0111	100100	00100	and
R	0111	100101	00101	or
R	0111	100110	00110	xor
R	0111	100111	00111	nor
R	0111	101010	01000	slt
R	0111	101011	01001	sltu
R	0111	000000	01011	sll
R	0111	000010	01100	srl
R	0111	000011	01101	sra
R	0111	000100	01111	sllv
R	0111	000110	10000	srlv
R	0111	000111	10001	srav
R	0111	011000	10010	mult
R	0111	011010	10011	div
lw,lh,lb,sw,sh, sb,addi	0000	xxxxxx	00000	add
beq,bne	0001	xxxxxx	00010	sub
andi	0010	xxxxxx	00100	and
ori	0011	xxxxxx	00101	or
xori	0100	xxxxxx	00110	xor

slti	0101	xxxxxx	01001	slt
lui	0110	xxxxxx	01010	lui

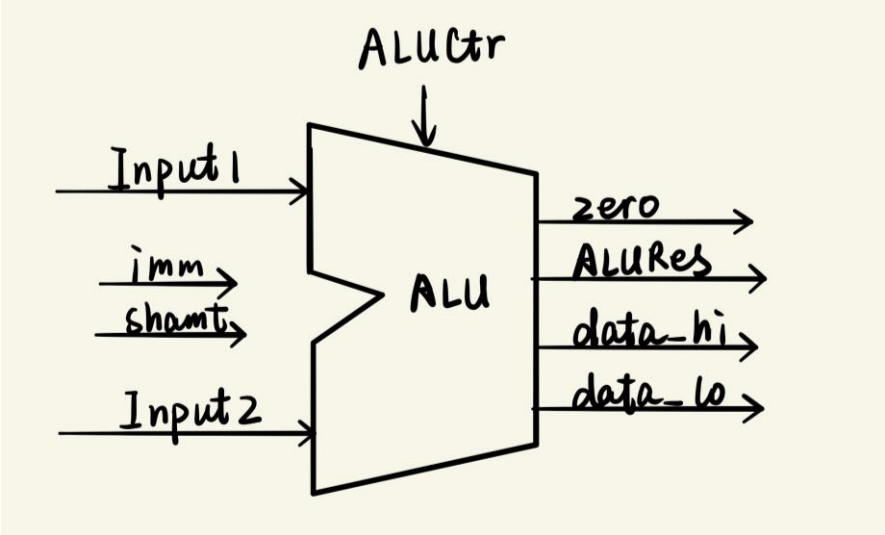
代码及注释如下：

```
`timescale 1ns / 1ps

module aluCtr(
    input [3:0] ALUOp,
    input [5:0] funct,
    output reg [4:0] ALUCtr
);

always @(ALUOp or funct)
    casex({ALUOp, funct})
        //非R型
        10'b0000xxxxxx: ALUCtr = 5'b00000; // add : lw, lh, lb, sw, sh, sb, addi
        10'b0001xxxxxx: ALUCtr = 5'b00010; // sub : beq, bne
        10'b0010xxxxxx: ALUCtr = 5'b00100; // and : andi
        10'b0011xxxxxx: ALUCtr = 5'b00101; // or : ori
        10'b0100xxxxxx: ALUCtr = 5'b00110; // xor : xori
        10'b0101xxxxxx: ALUCtr = 5'b01001; // slt : slti, sltiu
        10'b0110xxxxxx: ALUCtr = 5'b01010; // lui

        //R型  ALUOp = 111
        10'b0111_100000: ALUCtr = 5'b00000; // add
        10'b0111_100001: ALUCtr = 5'b00001; // addu
        10'b0111_100010: ALUCtr = 5'b00010; // sub
        10'b0111_100011: ALUCtr = 5'b00011; // subu
        10'b0111_100100: ALUCtr = 5'b00100; // and
        10'b0111_100101: ALUCtr = 5'b00101; // or
        10'b0111_100110: ALUCtr = 5'b00110; // xor
        10'b0111_100111: ALUCtr = 5'b00111; // nor
        10'b0111_101010: ALUCtr = 5'b01000; // slt
        10'b0111_101011: ALUCtr = 5'b01001; // sltu

        10'b0111_000000: ALUCtr = 5'b01011; // sll
        10'b0111_000010: ALUCtr = 5'b01100; // srl
        10'b0111_000011: ALUCtr = 5'b01101; // sra
        10'b0111_000100: ALUCtr = 5'b01111; // sllv
        10'b0111_000110: ALUCtr = 5'b10000; // srlv
        10'b0111_000111: ALUCtr = 5'b10001; // srav
        10'b0111_011000: ALUCtr = 5'b10010; // mult
        10'b0111_011010: ALUCtr = 5'b10011; // div

        default: ALUCtr = 5'b00000;
    endcase

endmodule
```

（4）ALU（算术逻辑单元）模块设计

ALU模块主要接受两个操作数，完成各类运算操作。包括加、减、与、或、小于设置。

R型指令需要ALU进行运算，sw、lw需要ALU进行地址运算，beq需要ALU进行相等比较，以及指令地址运算，所以需要输入ALU控制信号判断ALU进行的操作，再进一步对两个操作数进行操作，操作完输出结果。



各输入输出信号及意义如下：

```
3   module ALU(
4       input [31:0] input1,        //操作数，32位，输入
5       input [31:0] input2,        //操作数，32位，输入
6       input [15:0] imm,           //lui指令的立即数
7       input [4:0] ALUCtr,         //ALUCtr，5位操作码，输入
8       input [4:0] shamt,          //shift的移动距离shamt
9       output reg[31:0] ALURes,    //运算结果，32位，输出
10      output reg[31:0] data_hi,   //要写入hi寄存器的数据
11      output reg[31:0] data_lo,   //要写入lo寄存器的数据
12      output reg zero             //零标志，1位；当运算结果为0时，该位为1，否则为0
13  );
```

ALU代码及注释如下：

```
`timescale 1ns / 1ps

module ALU(
    input [31:0] input1, //操作数，32 位，输入
    input [31:0] input2, //操作数，32 位，输入
    input [15:0] imm, //lui 指令的立即数
    input [4:0] ALUCtr, //ALUCtr，5 位操作码，输入
    input [4:0] shamt, //shift 的移动距离 shamt
    output reg[31:0] ALURes, //运算结果，32 位，输出
    output reg[31:0] data_hi, //要写入 hi 寄存器的数据
    output reg[31:0] data_lo, //要写入 lo 寄存器的数据
    output reg zero//零标志，1 位；当运算结果为 0 时，该位为 1，否则为 0
);

/*
+     00000
+u    00001
-     00010
-u    00011
and   00100
or    00101
xor   00110
nor   00111
slt   01000
sltu  01001
lui   01010
sll   01011
srl   01100
sra   01101
sllv  01111
srlv  10000
srav  10001
```

```
*/
always @(input1 or input2 or ALUCtr) // 运算数或控制码变化时操作
begin
    case(ALUCtr)
    5'b00000: // +
    begin
        ALURes = $signed(input1) + $signed(input2);
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00001: // +u
    begin
        ALURes = input1 + input2;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00010: // -
    begin
        ALURes = $signed(input1) - $signed(input2);
        if(ALURes == 0) zero = 1;
        else zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00011: // -u
        begin
            ALURes = input1 - input2;
            if(ALURes == 0) zero = 1;
            else zero = 0;
            data_hi = 0;
            data_lo = 0;
        end
    5'b00100: // and
    begin
        ALURes = input1 & input2;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00101: // or
    begin
        ALURes = input1 | input2;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00110: // xor
    begin
        ALURes = ((~input1) & input2) | (input1 & (~input2));
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b00111: // nor
    begin
        ALURes = ~(input1 | input2);
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01000: // slt
    begin
        if($signed(input1) < $signed(input2)) ALURes = 1;
        else ALURes = 0;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01001: // sltu
    begin
        if(input1 < input2) ALURes = 1;
        else ALURes = 0;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01010: // lui
    begin
        ALURes = {imm[15:0], 16'h0000};
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01011: //sll
    begin
        ALURes = input2 << shamt;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01100: //srl
    begin
        ALURes = input2 >> shamt;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01101: //sra
    begin
        ALURes = $signed(input2) >>> shamt;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b01111: //sllv
    begin
        ALURes = input2 << input1[4:0];
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b10000: //srlv
    begin
        ALURes = input2 >> input1[4:0];
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b10001: //srav
    begin
        ALURes = $signed(input2) >>> input1[4:0];
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
    5'b10010: //mult
    begin
        {data_hi, data_lo} = input1 * input2;
        ALURes = 0;
        zero = 0;
    end
    5'b10011: //div
    begin
        data_hi = input1 / input2;
        data_lo = input1 % input2;
        ALURes = 0;
        zero = 0;
    end
    default:
    begin
        ALURes = 0;
        zero = 0;
        data_hi = 0;
        data_lo = 0;
    end
```

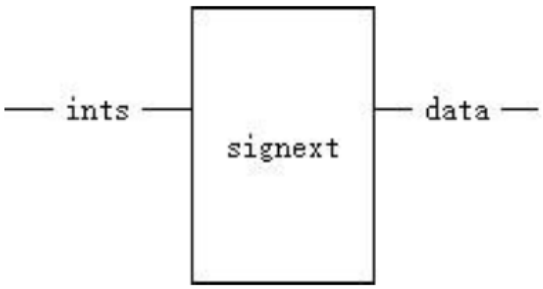
```
    endcase
end
```

```
endmodule
```

（5）signExt（符号拓展）模块设计

将16位有符号数扩展成32位有符号数，只需要在16位数前面补足符号即可。

input	inst[15:0]
output	data[31:0]

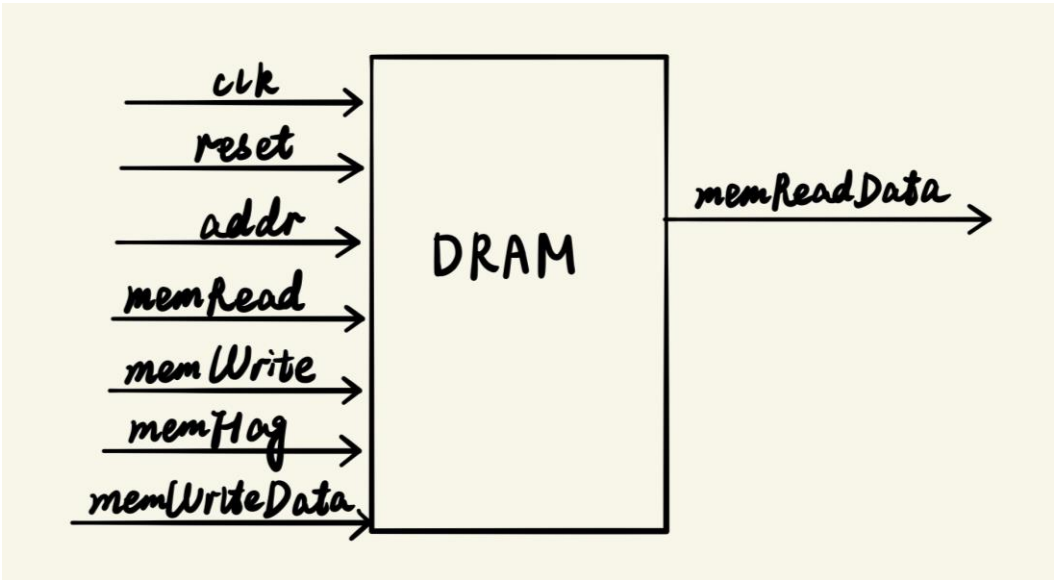


代码及注释如下：

```
`timescale 1ns / 1ps

module signext(
    input [15:0] imm,  // 16 bits input
    input extOption,    // 1: signextend; 0: zeroextend
    output [31:0] data  // 32 bits output
);

// 根据符号补充符号位
// 如果符号位为1，则补充16个1，即16'h ffff
// 如果符号位为0，则补充16个0
assign data = (extOption & imm[15:15]) ? {16'hffff,imm} : {16'h0000,imm};

endmodule
```

（6）DRAM（数据存储器）模块设计

DRAM模块作数据存储器，主要在指令sw、lw中使用，将数值存进相应地址的内存中或根据取值地址将内存数据取出。



各输入输出信号及意义如下：

```
3   module DRAM(
4       input clk,                  //时钟信号
5       input reset,                //复位信号
6       input [7:0] addr,           //地址信号
7       input memRead,              //memory读使能信号
8       input memWrite,             //memory写使能信号
9       input [1:0] memFlag,        //00-> byte 01->half word  11->word
10      input [31:0] memWriteData,  //要写入memory的信号
11      output [31:0] memReadData   //从memory读取的信号
12  );
```

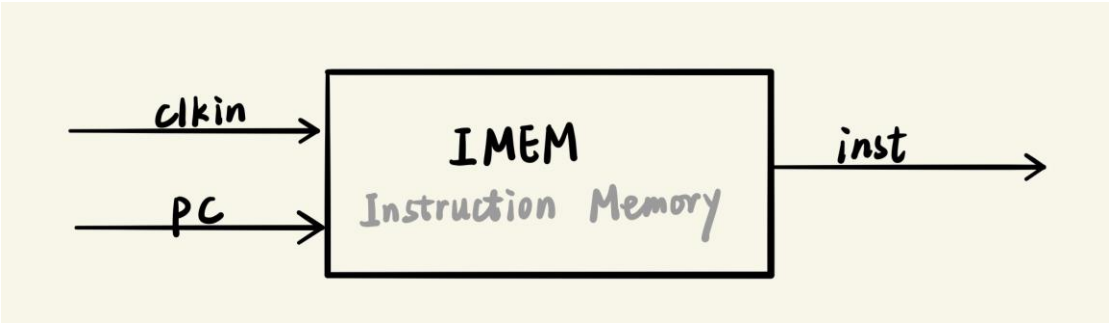
代码及注释如下：

```
`timescale 1ns / 1ps

module DRAM(
    input clk,                  //时钟信号
    input reset,                //复位信号
    input [7:0] addr,           //地址信号
    input memRead,              //memory 读使能信号
    input memWrite,             //memory 写使能信号
    input [1:0] memFlag,        //00-> byte 01->half word  11->word
    input [31:0] memWriteData,  //要写入 memory 的信号
    output [31:0] memReadData   //从 memory 读取的信号
);

reg[7:0] RAM[255:0];

//read
assign memReadData = memRead ? ( memFlag[1] ? { {RAM[addr]}, {RAM[addr+1]}, {RAM[ad
dr+2]}, {RAM[addr+3]}} : ( memFlag[0] ? { {16{RAM[addr][7]}}, {RAM[addr]} , {RAM[ad
dr+1]} } : { {24{RAM[addr][7]}}, RAM[addr]} ) ) : 32'b0;

//write flag 作为字节或字半字操作的标志
integer i;

always @ (posedge clk, posedge reset)
begin
    if(reset)
    begin
        for(i = 0; i < 256; i = i + 1)
            RAM[i]=0;
    end
    else if (memWrite)
    begin
        if(memFlag == 2'b00)
        begin
            RAM[addr] = memWriteData[7:0];
        end
        else if(memFlag == 2'b01)
        begin
            { {RAM[addr]}, {RAM[addr+1]} } = memWriteData[15:0];
        end
        else if(memFlag == 2'b11)
        begin
            { {RAM[addr]}, {RAM[addr+1]}, {RAM[addr+2]}, {RAM[addr+3]}} = memWriteD
ata[31:0];
        end
    end
end

endmodule
```

（7）IMEM（指令存储器）模块设计

IMEM的数据宽度为八位。因为指令为32位，故需要连续取四个字节。

采用编码方式录入指令寄存器。



各输入输出信号及意义如下：

```
3   module IMEM(
4       input clkin,            //时钟信号
5       input [31:0] pc,        //pc信号
6       output [31:0] inst      //输出指令
7   );
```

具体的取指令代码实现：

```
assign addr0 = pc[7:0];
assign addr1 = addr0 + 1'b1;
assign addr2 = addr1 + 1'b1;
assign addr3 = addr2 + 1'b1;
```

```
assign inst = {imem[addr0], imem[addr1], imem[addr2], imem[addr3]};
```

（8）数码管显示模块

（9）top模块设计

CPU封装及与各个模块连接

1、到程序 ROM 中取指令

2、对 PC 值进行＋4 处理

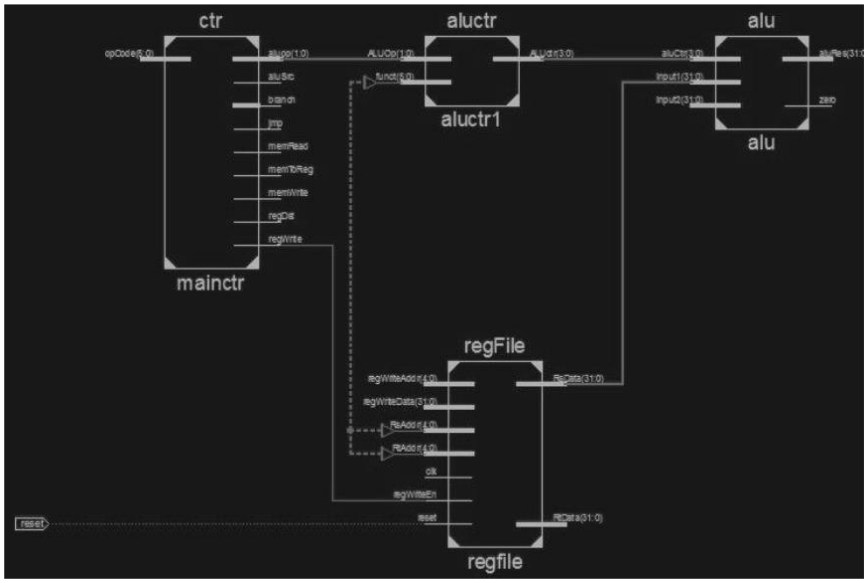
3、完成各种跳转指令的 PC 修改功能

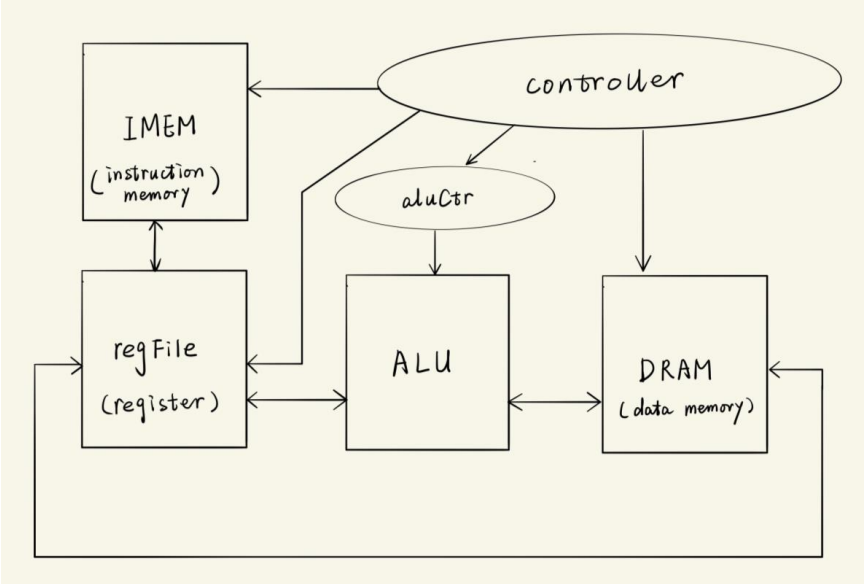
4、在有中断的情况下处理中断到来时的 PC 修改

顶层模块需要将前面的多个模块实例化后，通过导线以及多路复用器将各个部件链接起来，并且在时钟的控制下修改PC的值，PC是一个32位的寄存器，每个时钟沿自动增加4。

多路复用器MUX直接通过三目运算符实现：Assign $OUT=SE\ ?INPUT1：INPUT2$;

示意图：





top模块代码及注释如下：

```
`timescale 1ns / 1ps

module top(
    // input clk,
    input clkin,
    input reset,
    output wire [31:0] inst,
    output reg [31:0] pc,
    output reg [31:0] pcAdd4
    // output[3:0] which,
    // output[6:0] seg
);
```

```
// integer clk_cnt, clk_cnt_display;
// reg clkin, clkin_display;
// always @(posedge clk)
//     if(clk_cnt == 32'd50_000_000)
//     begin
//         clk_cnt <= 1'd0;
//         clkin <= ~clkin;
//     end
//     else clk_cnt <= clk_cnt + 1'd1;

// always @(posedge clk)
//     if(clk_cnt_display == 32'd12_500_000)
```

```
//     begin
//         clk_cnt_display <= 1'd0;
//         clkin_display <= ~clkin_dis
play;
//     end
//     else clk_cnt_display <= clk_cnt
_display + 1'd1;

wire[31:0] inst;
reg [31:0] pc, pcAdd4;
// 指令寄存器pc
wire pcWrite;
wire branchChoose, jumpChoose;
wire[31:0] branchAddress, jumpAddress;

// 复用器信号线
wire[31:0] mux2, mux3, mux4, mux5, mux
7, mux9;
wire[4:0] mux1, mux6/*, mux8*/;

// CPU 控制信号
wire regDst, ALUSrc, memRead, memWrite
, memToReg, regWrite, jump, jal, jr, b
eq, bne, lui, extOption, mult, div, mf
hi, mflo;
wire[3:0] ALUOp;

// ALUCtr、ALU 信号
wire zero;
wire[31:0] ALURes, data_hi, data_lo;
wire[4:0] ALUCtr;

// 内存信号
wire[31:0] memReadData, memWriteData;
wire[1:0] memFlag;

// 寄存器信号线
wire[31:0] RsData, RtData;

// 扩展信号
wire[31:0] expand, expandShiftLeft2;

// 取指令
always @(negedge clkin) // 时钟下降沿操
作
begin
    if(!reset)
    begin
        if(pcWrite)
        begin
            pc = mux7; // 计算下一条pc,
修改pc
            pcAdd4 = pc + 4;
        end
    end
    else
    begin
        pc = 32'b0; // 复位时pc=0
        pcAdd4 = 32'h4;
    end
end

// 实例化控制器模块
controller mainctr(
    .opCode(inst[31:26]),
    .funct(inst[5:0]),
    .ALUOp(ALUOp),
    .regDst(regDst),
    .ALUSrc(ALUSrc),
    .memToReg(memToReg),
    .memWrite(memWrite),
    .memRead(memRead),
    .regWrite(regWrite),
    .extOption(extOption),
    .beq(beq),
    .bne(bne),
    .jump(jump),
    .jal(jal),
    .jr(jr),
    .memFlag(memFlag),
    .pcWrite(pcWrite),
    .mult(mult),
    .div(div),
    .mfhi(mfhi),
    .mflo(mflo)
);

// 实例化ALU 模块
aluCtr aluctr(
    .ALUOp(ALUOp),
    .funct(inst[5:0]),
    .ALUCtr(ALUCtr)
);

// 实例化寄存器模块
regFile regfile(
    .clk(!clkin),
    .reset(reset),
    .RsAddr(inst[25:21]),
    .RtAddr(inst[20:16]),
    .regWriteAddr(mux6),
    .regWriteData(mux9),
    .regWrite(regWrite),
    .mult(mult),
    .div(div),
    .mfhi(mfhi),
    .mflo(mflo),
    .data_hi(data_hi),
    .data_lo(data_lo),
    .RsData(RsData),
    .RtData(RtData)
);

// 实例化ALU 模块
ALU alu(
    .input1(RsData),
    .input2(mux2),
    .imm(inst[15:0]),
    .shamt(inst[10:6]),
    .ALUCtr(ALUCtr),
    .zero(zero),
    .ALURes(ALURes),
    .data_hi(data_hi),
    .data_lo(data_lo)
);

// 实例化DRAM 模块
DRAM DMEM(
    .clk(!clkin),
    .reset(reset),
    .addr(ALURes[7:0]),
    .memRead(memRead),
    .memWrite(memWrite),
    .memFlag(memFlag),
    .memWriteData(RtData),
    .memReadData(memReadData)
);
```

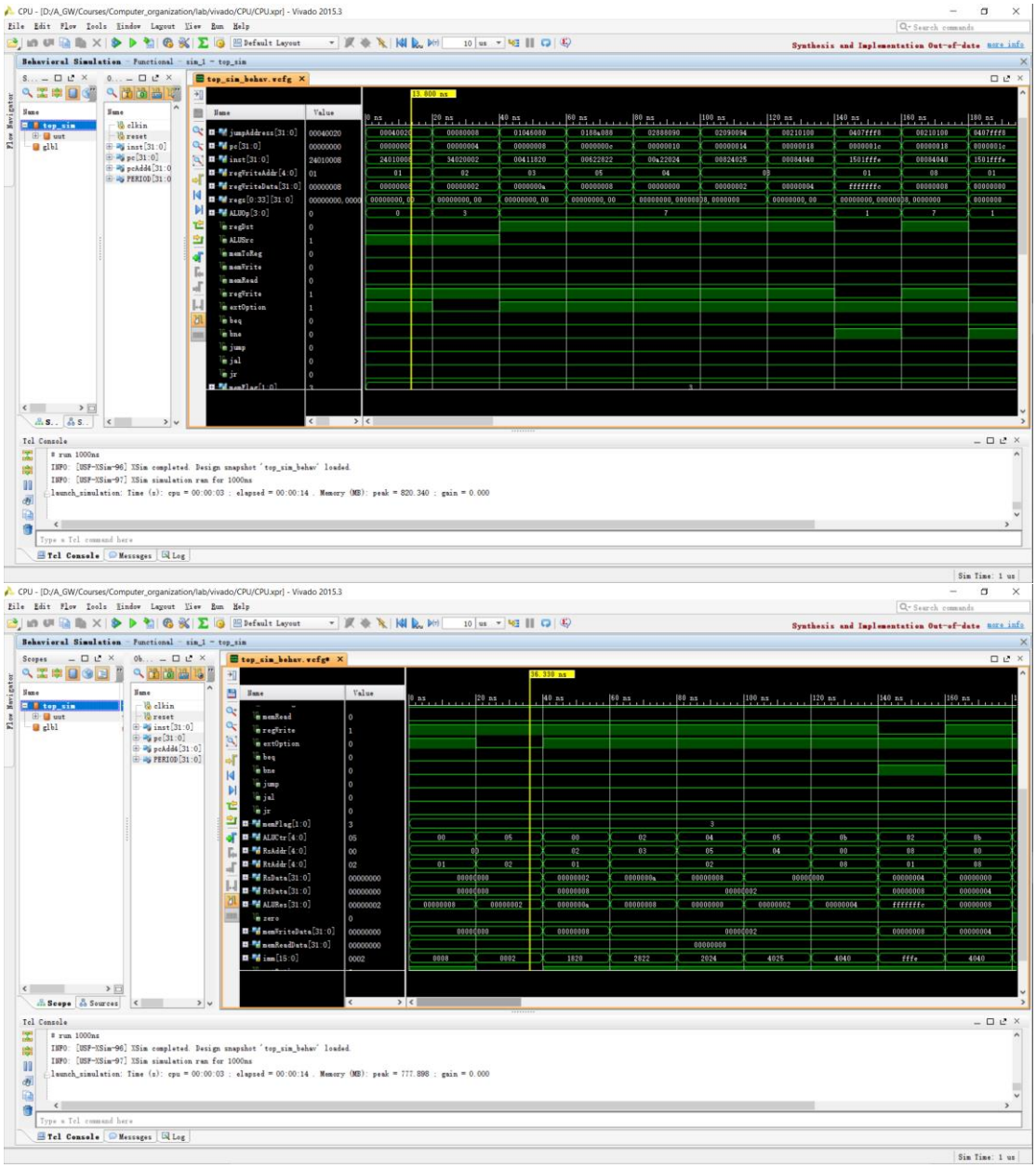
```
// 实例化符号扩展模块
signext signext(
    .imm(inst[15:0]),
    .extOption(extOption),
    .data(expand)
);

IMEM imem(
    .clkin(clkin),
    .pc(pc),
    .inst(inst)
);

// 各个控制信号线，地址，符号扩展
assign mux1 = regDst ? inst[15:11] : i
nst[20:16];
assign mux2 = ALUSrc ? expand : RtData
;
assign mux3 = memToReg ? memReadData :
 ALURes;
assign mux4 = branchChoose ? branchAdd
ress : pcAdd4;
assign mux5 = jumpChoose ? jumpAddress
 : mux4;
assign mux6 = jal ? 5'd31 : mux1;
assign mux7 = jr ? RsData : mux5;
assign mux9 = jal ? pcAdd4 : mux3;
assign branchChoose = (beq & zero) | (
bne & (~zero));
assign jumpChoose = jump | jal;
assign expandShiftLeft2 = expand << 2;

assign jumpAddress = {pcAdd4[31:28], i
nst[25:0], 2'b00};
assign branchAddress = pcAdd4 + expand
ShiftLeft2;

// display dis(
//     .clk(clkin_display),
//     .data(ALURes[15:0]),
//     .which(which),
//     .seg(seg)
// );

endmodule
```

2. 验证CPU的正确性

仿真代码及注释如下：

```
`timescale 1ns / 1ps

module top_sim;

// Inputs
reg clkin;
reg reset;

wire[31:0] inst, pc, pcAdd4;

// Instantiate the Unit Under Test (UUT)
top uut (
    .clkin(clkin),
    .reset(reset),
    .inst(inst),
    .pc(pc),
    .pcAdd4(pcAdd4)
);

initial
begin
    // Initialize Inputs
    clkin = 0;
    reset = 1;
    // Wait 20 ns for global reset to finish
    #20;
    reset = 0;
end

parameter PERIOD = 20;

always
begin
    clkin = 1'b0;
    #(PERIOD / 2);
    clkin = 1'b1;
    #(PERIOD / 2);
end
endmodule
```

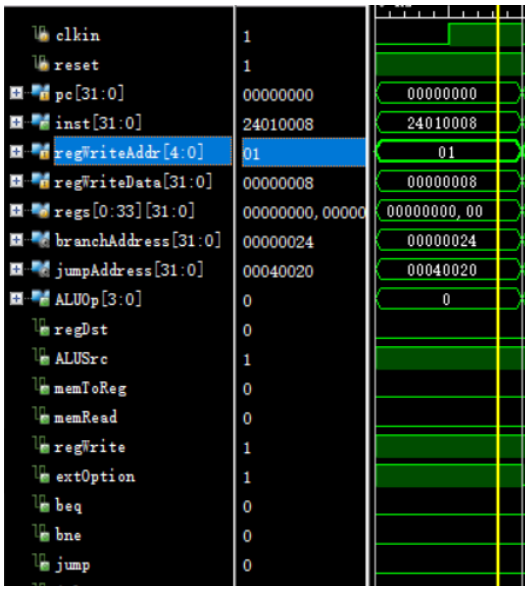
仿真结果：

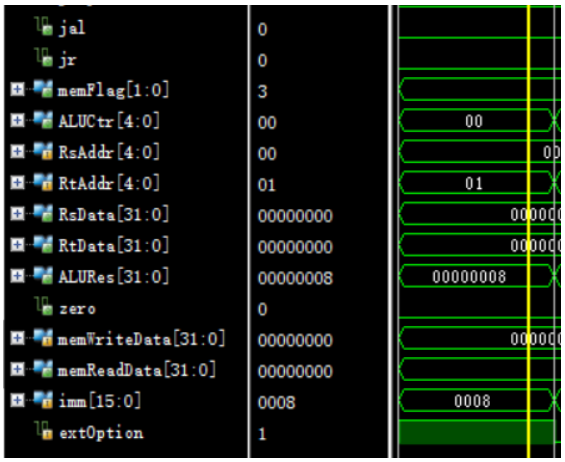


波形分析：

1.

address	instruction	op	rs	rt	immediate	code	result
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	24010008	$1 = 8





该指令正确：$0 = 0, $1 = 0 + 8 = 8

pc = 00000000

regWriteAddr（写入的寄存器地址）=1，regWriteData（向寄存器写入的值）=8

regWrite（寄存器写入使能信号）=1

2.

address	instruction	op	rs	rt	immediate	code	result
0x00000004	ori $2,$0,2	001101	00000	00010	0000 0000 0000 0010	34020002	$2 = 2

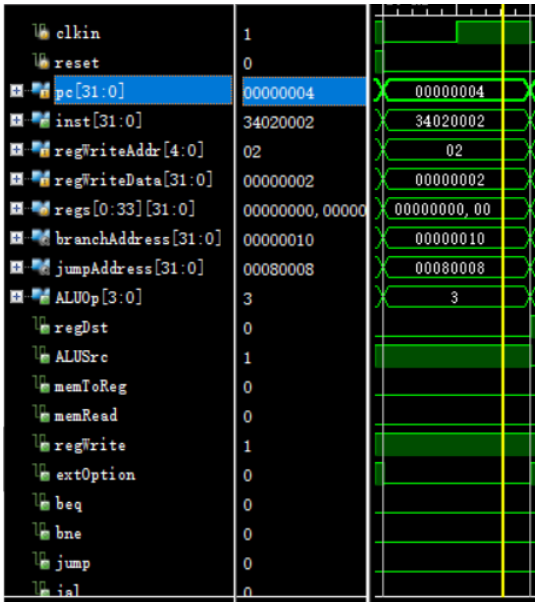
该指令正确：$2 = 0 or 2 = 2

pc = 00000004

regWriteAddr（写入的寄存器地址）=2

regWriteData（向寄存器写入的值）=2

regWrite（寄存器写入使能信号）=1



3.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000008	add $3,$2,$1	000000	00010	00001	00011	00000	100000	00411820	$3 = 10

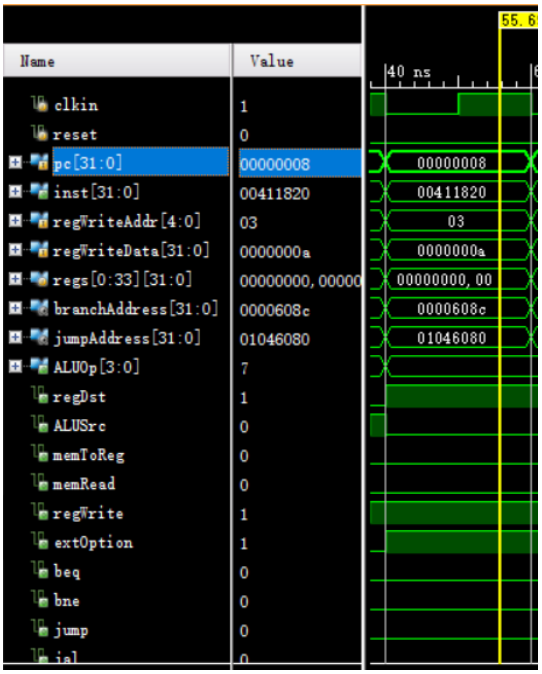
该指令正确：$3 = $2 and $1 = 11B and 1000B = 1010B = 10

pc = 00000008

regWriteAddr（写入的寄存器地址）=3

regWriteData（向寄存器写入的值）=10

regWrite（寄存器写入使能信号）=1



4.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101	00000	100010	00622822	$5 = 8

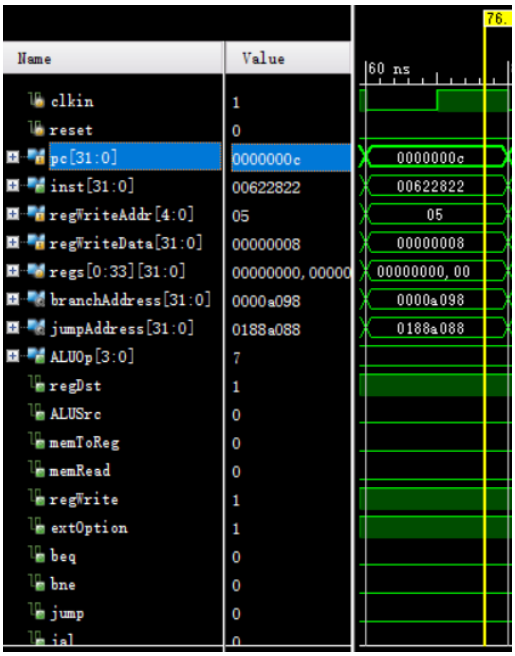
该指令正确：$5 = $3 - $2 = 10 – 2 = 8

pc = 0000000C

regWriteAddr（写入的寄存器地址）=5

regWriteData（向寄存器写入的值）=8

regWrite（寄存器写入使能信号）=1



5.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000010	and $4,$5,$2	000000	00101	00010	00100	00000	100100	00a22024	$4 = 0

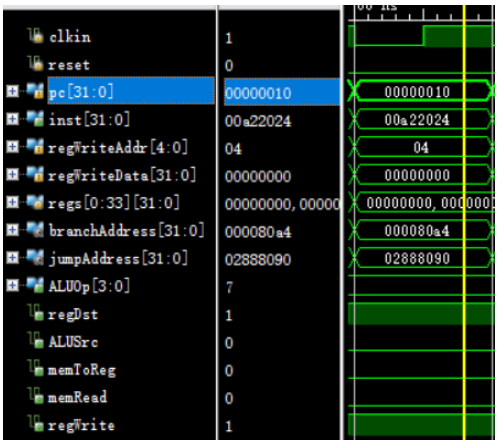
该指令正确：$4 = $5 and $2 = 1000B and 0010B = 0000B = 0

pc = 00000010

regWriteAddr（写入的寄存器地址）=4

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1



6.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000014	or $8,$4,$2	000000	00100	00010	01000	00000	100101	00824025	$8 = 2

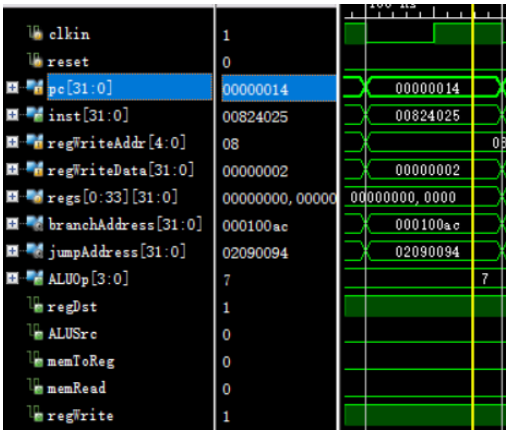
该指令正确：$8 = $4 or $2 = 00B or 10B = 10B = 2

pc = 00000014

regWriteAddr（写入的寄存器地址）=8

regWriteData（向寄存器写入的值）=2

regWrite（寄存器写入使能信号）=1



7.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000018	sll $8,$8,1	000000	00000	01000	01000	00001	000000	00084040	$8 = 4

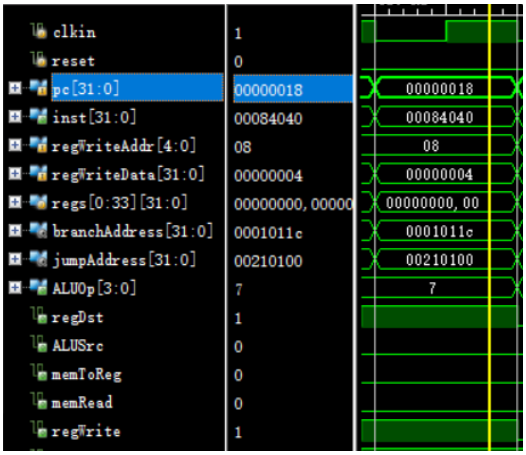
该指令正确：$8 = $8 << 1 = 2 << 1 = 4

pc = 00000018

regWriteAddr（写入的寄存器地址）=8

regWriteData（向寄存器写入的值）=4

regWrite（寄存器写入使能信号）=1



8.

address	instruction	op	rs	rt	immediate	code	result
0x0000001C	bne $8,$1,-2	000101	00001	01000	1111 1111 1111 1110	1501fffe	(≠,转 18)

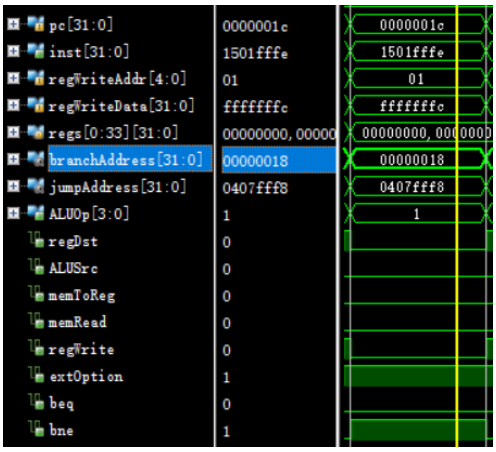
该指令正确：$8 = 4 ≠ $1 = 8，跳转。

pc = 0000001C

branchAddress=18

（条件分支要跳转到的地址）

bne = 1



9.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000018	sll $8,$8,1	000000	00000	01000	01000	00001	000000	00084040	$8 = 8

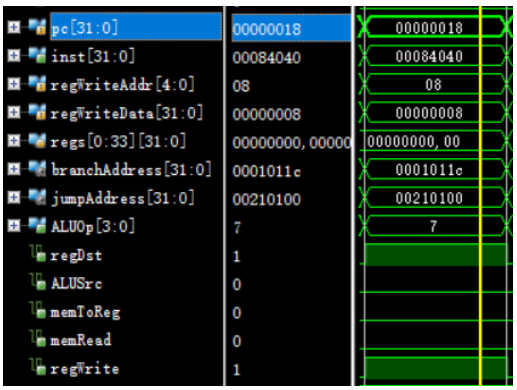
该指令正确：$8 = $8 << 1 = 4 << 1 = 8

pc = 00000018

regWriteAddr（写入的寄存器地址）=8

regWriteData（向寄存器写入的值）=8

regWrite（寄存器写入使能信号）=1



10.

address	instruction	op	rs	rt	immediate	code	result
0x0000001C	bne $8,$1,-2	000101	00001	01000	1111 1111 1111 1110	1501fffe	=，不跳转

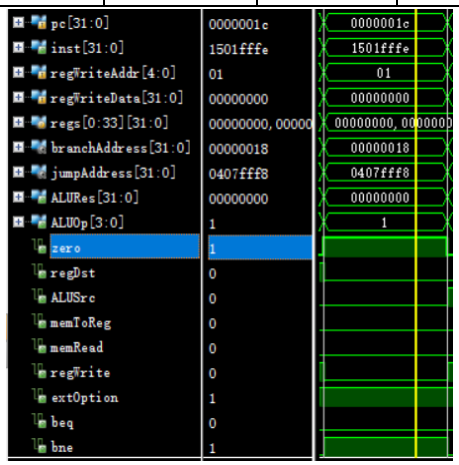
该指令正确：$8 = $1 = 8，不跳转

pc = 0000001C

bne = 1

zero = 0

不跳转



11.

address	instruction	op	rs	rt	immediate	code	result
0x00000020	slti $6,$2,4	001010	00010	00110	0000 0000 0000 0100	28460004	$6 = 1

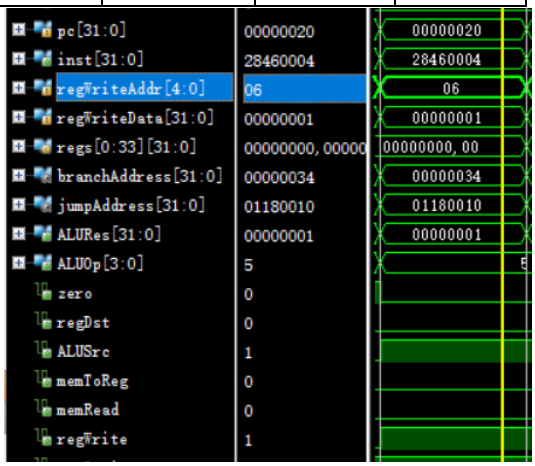
该指令正确：$2 = 2 < 4，故 $6 = 1

pc = 00000020

regWriteAddr（写入的寄存器地址）=6

regWriteData（向寄存器写入的值）=1

regWrite（寄存器写入使能信号）=1



12.

address	instruction	op	rs	rt	immediate	code	result
0x00000024	sltiu $7,$6,0	001011	00110	00111	0000 0000 0000 0000	2cc70000	$7 = 0

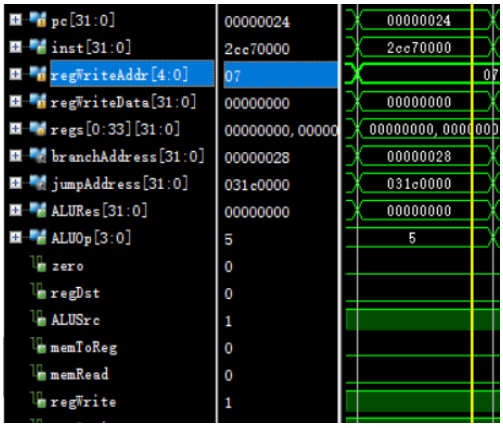
该指令正确：$6 = 1 > 0，故 $7 = 0

pc = 00000024

regWriteAddr（写入的寄存器地址）=7

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1



13.

address	instruction	op	rs	rt	immediate	code	result
0x00000028	addiu $7,$7,8	001000	00111	00111	0000 0000 0000 1000	24e70008	$7 = 8

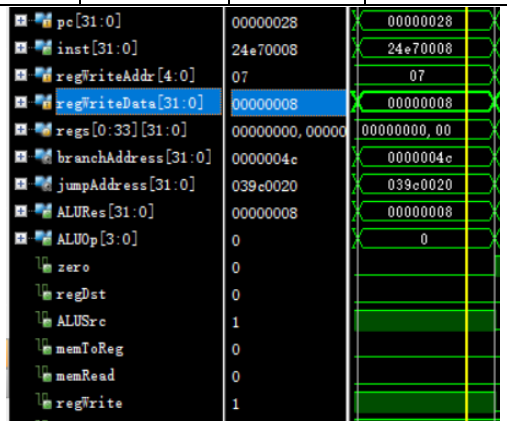
该指令正确：$7 = $7 + 8 = 0 + 8 = 8

pc = 00000028

regWriteAddr（写入的寄存器地址）=7

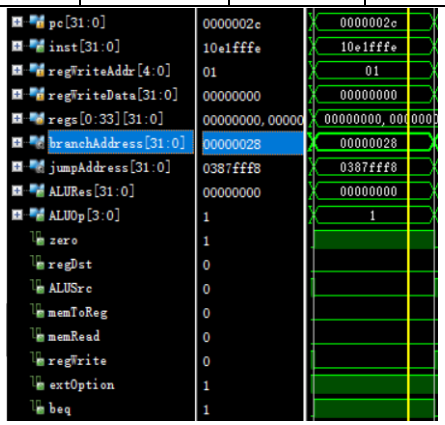
regWriteData（向寄存器写入的值）=8

regWrite（寄存器写入使能信号）=1



14.

address	instruction	op	rs	rt	immediate	code	result
0x0000002C	beq $7,$1,-2	000100	00111	00001	1111 1111 1111 1110	10e1fffe	(=,转 28)



该指令正确:$7 = $1 = 8,跳转到pcAdd4的前两条指令

pc = 0000002C

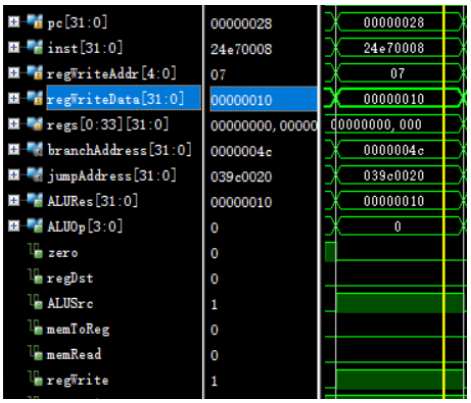
beq = 1

zero = 1

branchAddress = 28

15.

address	instruction	op	rs	rt	immediate	code	result
0x00000028	addiu $7,$7,8	001000	00111	00111	0000 0000 0000 1000	24e70008	$7 = 16



该指令正确：$7 = $7 + 8 = 8 + 8 = 16

pc = 00000028

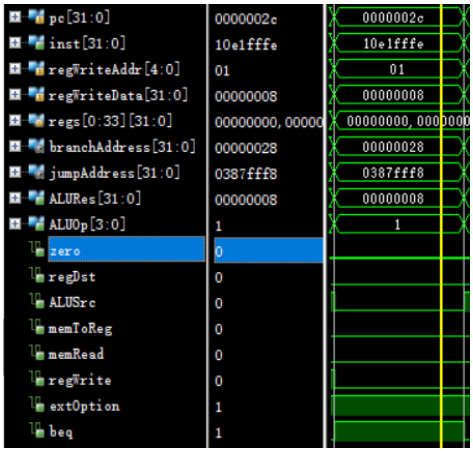
regWriteAddr（写入的寄存器地址）=7

regWriteData（向寄存器写入的值）=0x10

regWrite（寄存器写入使能信号）=1

16.

address	instruction	op	rs	rt	immediate	code	result
0x0000002C	beq $7,$1,-2	000100	00111	00001	1111 1111 1111 1110	10e1fffe	(≠,不转)



该指令正确：$7=16≠$1=8，故不跳转

pc = 0000002C

beq = 1

zero = 0

不跳转

17.

address	instruction	op	rs	rt	immediate	code	result
0x00000030	sw $2,4($1)	101011	00001	00010	0000 0000 0000 0100	ac220004	mem[12:15]<=2

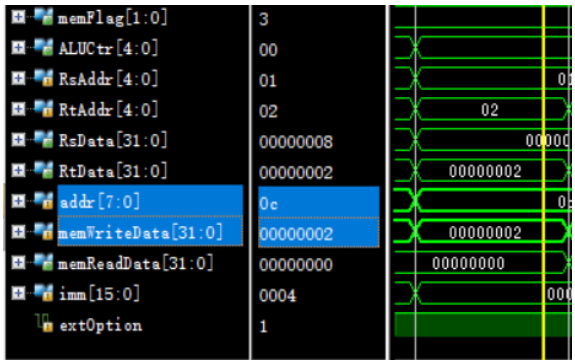
该指令正确：

pc = 00000030

addr(DMEM的写入地址) = 0x0c

memWriteData = 2

此时DMEM的状态为：mem[12:15] = 00000000 00000000 00000000 00000010



18.

address	instruction	op	rs	rt	immediate	code	result
0x00000034	lw $9,4($1)	100011	00001	01001	0000 0000 0000 0100	8c290004	$9 = 2

该指令正确：上条指令在mem的同一位置存入了2，故该指令会读取2并赋值给$9

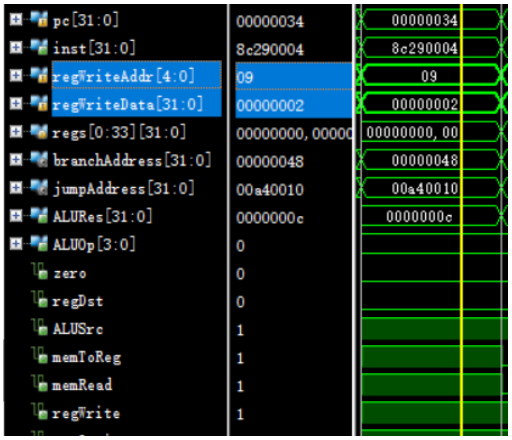
pc = 00000034

regWriteAddr = 9

regWriteData = 2

memRead = 1, memToReg = 1

regWrite = 1



19.

address	instruction	op	rs	rt	immediate	code	result
0x00000038	addiu $10,$0,-2	001001	00000	01010	1111 1111 1111 1110	240afffe	$10 = -2

该指令正确：$10 = 0 – 2 = -2

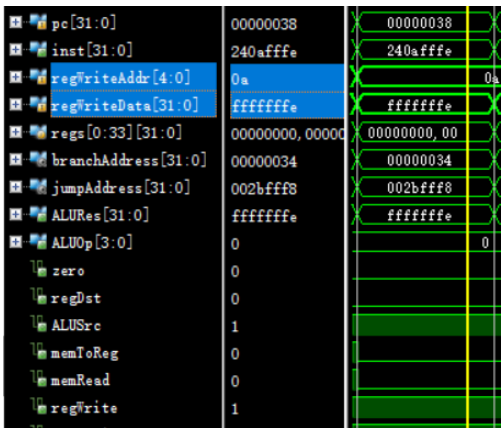
pc = 00000038

regWriteAddr (写入的寄存器地址) =0xa

regWriteData（向寄存器写入的值）=-2

-2的补码为fffffffe

regWrite（寄存器写入使能信号）=1



20.

address	instruction	op	rs	rt	immediate	code	result
0x0000003C	addiu $10,$10,1	001001	01010	01010	0000 0000 0000 0001	254a0001	$10 = -1

该指令正确:$10 = $10 + 1 = -2 + 1 = -1

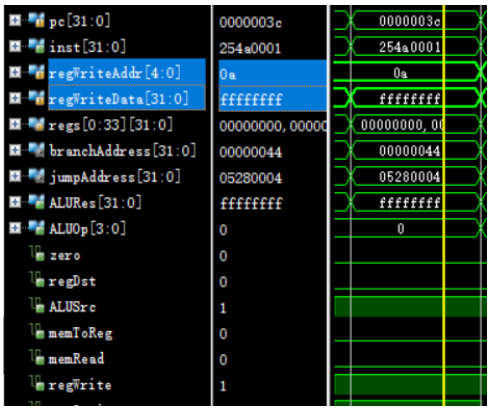
pc = 0000003C

regWriteAddr(写入的寄存器地址)=0xa

regWriteData（向寄存器写入的值）=-1

-1的补码为ffffffff

regWrite（寄存器写入使能信号）=1



21.

address	instruction	op	rs	rt	immediate	code	result
0x00000040	andi $11,$2,2	001100	00010	01011	0000 0000 0000 0010	304b0002	$11 = 2

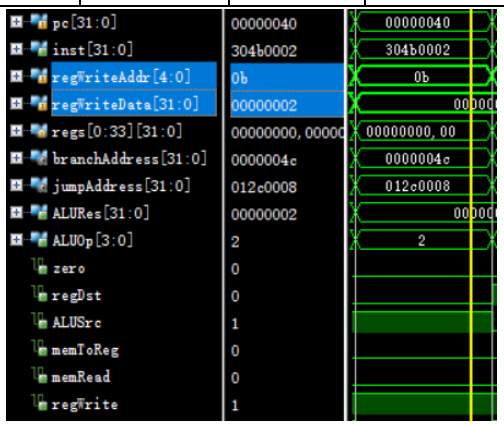
该指令正确：$11 = $2 + 2 = 2 + 2 = 4

pc = 00000040

regWriteAddr(写入的寄存器地址)=0xb

regWriteData（向寄存器写入的值）=2

regWrite（寄存器写入使能信号）=1



22.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000044	addu $10,$0,$2	000000	00000	00010	01010	00000	100001	00025021	$10 = 2

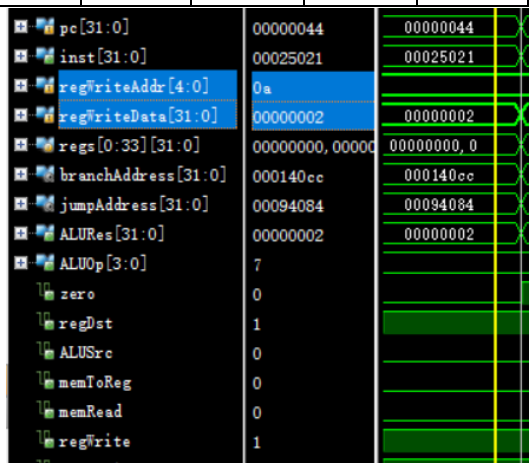
该指令正确：$10 = $0 + $2 = 0 + 2 = 2

pc = 00000044

regWriteAddr(写入的寄存器地址)=0xa

regWriteData（向寄存器写入的值）=2

regWrite（寄存器写入使能信号）=1



23.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000048	subu $10,$10,$2	000000	01010	00010	01010	00000	100011	01425023	$10 = 0

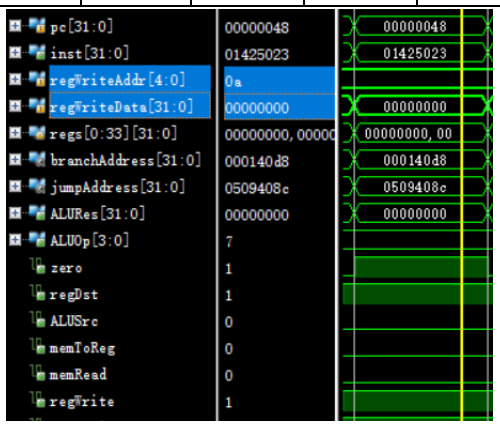
该指令正确:$10 = $10 – 2 = 2 – 2 = 0

pc = 00000048

regWriteAddr (写入的寄存器地址) =0xa

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1



24.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000004C	xor $10,$8,$2	000000	01000	00010	01010	00000	100110	01025026	$10 =10

该指令正确: $10 = 1000B xor 0010B =

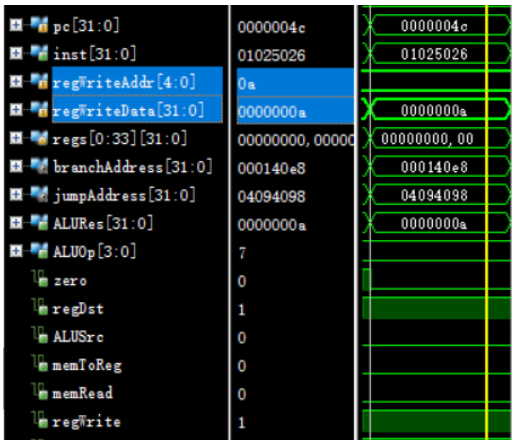
1010 B = 10

pc = 0000004C

regWriteAddr (写入的寄存器地址) =0xa

regWriteData (向寄存器写入的值) =0xa

regWrite（寄存器写入使能信号）=1



25.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000050	nor $10,$8,$2	000000	01000	00010	01010	00000	100111	01025027	$10 = fffffff5

该指令正确: $10 = 1000B nor 0010B =

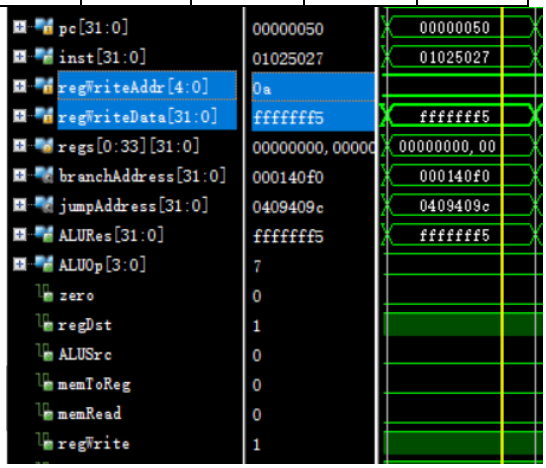
= ffffff5H

pc = 00000050

regWriteAddr (写入的寄存器地址) =0xa

regWriteData（向寄存器写入的值）=fffffff5

regWrite（寄存器写入使能信号）=1



26.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000054	sltu $10,$8,$2	000000	01000	00010	01010	00000	101011	0102502b	$10 = 0

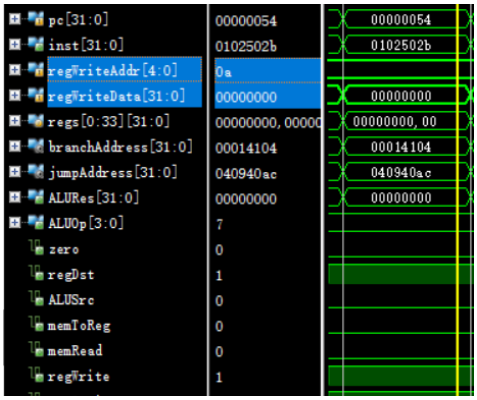
该指令正确：$8 = 8 > $2 = 2，故$10 = 0

pc = 00000054

regWriteAddr（写入的寄存器地址）=0xa

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1



27.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000058	srl $10,$8,1	000000	00000	01000	01010	00001	000010	00085042	$10 = 4

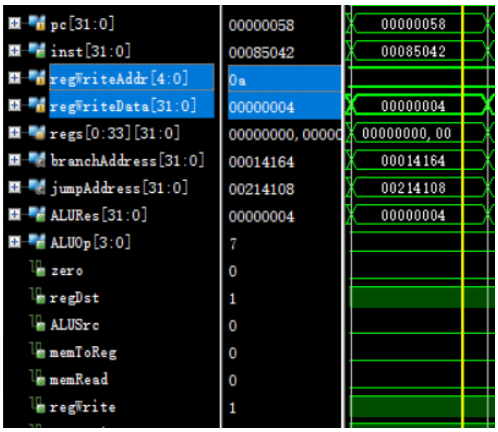
该指令正确：$10 = 8 >> 1 = 4

pc = 00000058

regWriteAddr（写入的寄存器地址）=0xa

regWriteData（向寄存器写入的值）=4

regWrite（寄存器写入使能信号）=1



28.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000005C	sllv $10,$8,$6	000000	01000	00110	01010	00000	000100	00c85004	$10 =16

该指令正确:

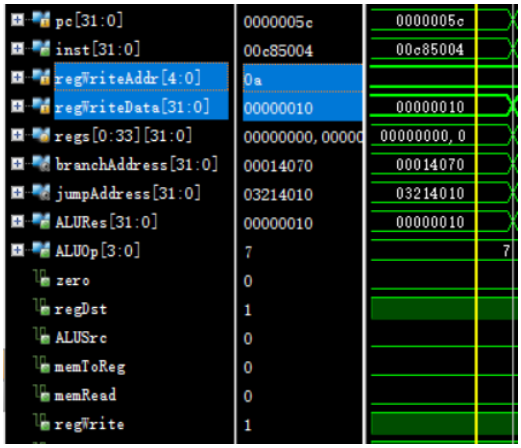
$10 = $8 << $6 = 8 << 1 = 16 = 0x10

pc = 0000005C

regWriteAddr（写入的寄存器地址）=0xa

regWriteData（向寄存器写入的值）=0x10

regWrite（寄存器写入使能信号）=1



29.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000060	srlv $10,$8,$6	000000	01000	00110	01010	00000	000110	00c85006	$10 = 4

该指令正确：

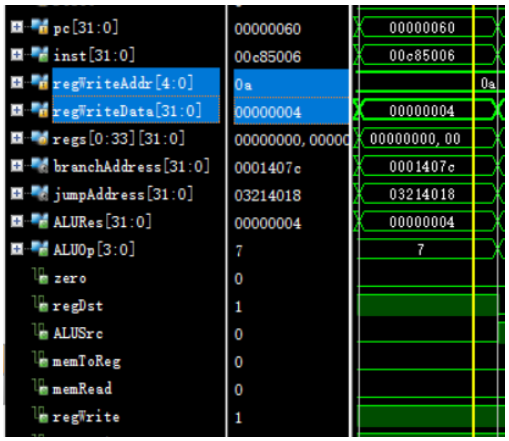
$10 = $8 >> $6 = 8 >> 1 = 4

pc = 00000060

regWriteAddr（写入的寄存器地址）=0xa

regWriteData（向寄存器写入的值）=4

regWrite（寄存器写入使能信号）=1



30.

address	instruction	op	rs	rt	immediate	code	result
0x00000064	addi $10,$0,-1	001000	00000	01010	1111 1111 1111 1111	200affff	$10 = -1

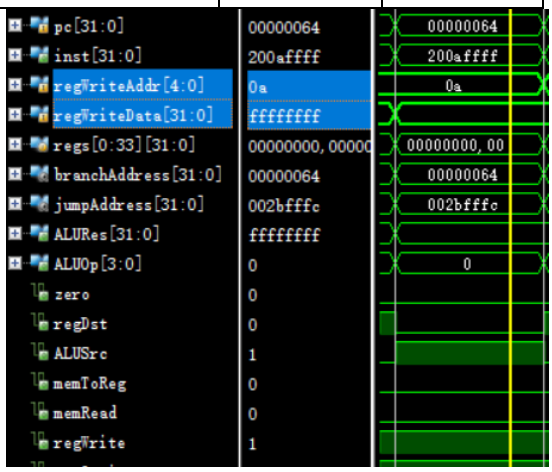
该指令正确：$10 = 0 － 1 = -1

pc = 00000064

regWriteAddr（写入的寄存器地址）=0xa

regWriteData（向寄存器写入的值）=ffffffff

regWrite（寄存器写入使能信号）=1



31.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000068	sra $11,$10,1	000000	00000	01010	01011	00001	000011	000a5843	$11 = -1

该指令正确：$11 = $10 >> 1（逻辑右移）

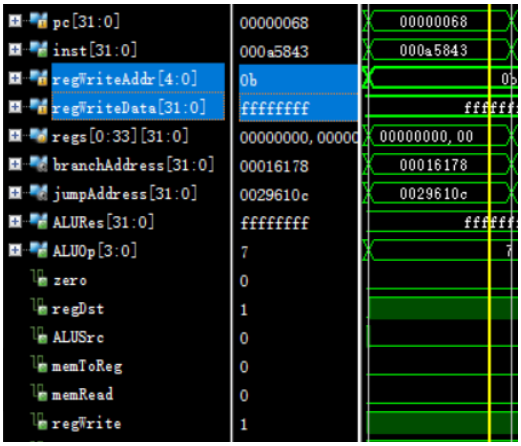
= ffffff >> 1 = ffffff

pc = 00000068

regWriteAddr（写入的寄存器地址）=0xb

regWriteData（向寄存器写入的值）=ffffff

regWrite（寄存器写入使能信号）=1



32.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000006C	srav $11,$10,$6	000000	01010	00110	01011	00000	000111	00ca5807	$11 = -1

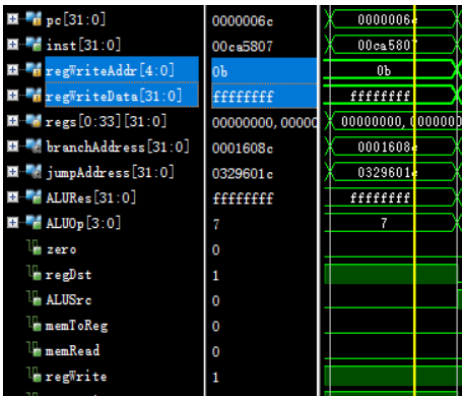
该指令正确：$11 = $10 >> $6（逻辑右移） = ffffffff >> 1 = ffffffff

pc = 0000006C

regWriteAddr(写入的寄存器地址)=0xb

regWriteData（向寄存器写入的值）=ffffffff

regWrite（寄存器写入使能信号）=1



33.

address	instruction	op	rs	rt	immediate	code	result
0x00000070	xori $10,$3,7	001110	00011	01010	0000 0000 0000 0111	386a0007	$10 = 13

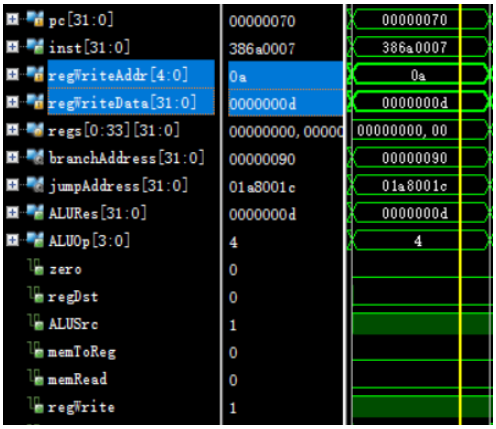
该指令正确：$10 = $3 xor 7 = 1010B xor 0111 = 1101B = 13

pc = 00000070

regWriteAddr(写入的寄存器地址)=0xa

regWriteData(向寄存器写入的值)=0xd

regWrite（寄存器写入使能信号）=1



34.

address	instruction	op	rs	rt	immediate	code	result
0x00000074	lui $5, 10	001111	00000	00101	0000 0000 0000 1010	3c05000a	$5=0xA0000

该指令正确：$5 = 10 << 16 = 0xA0000

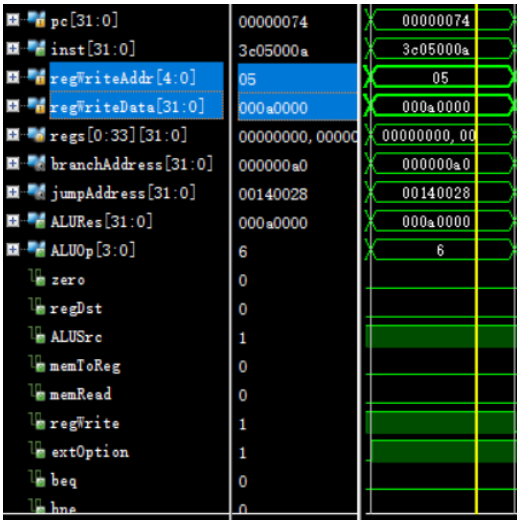
pc = 00000074

regWriteAddr（写入的寄存器地址）=5

regWriteData（向寄存器写入的值）=0xA0000

regWrite（寄存器写入使能信号）=1

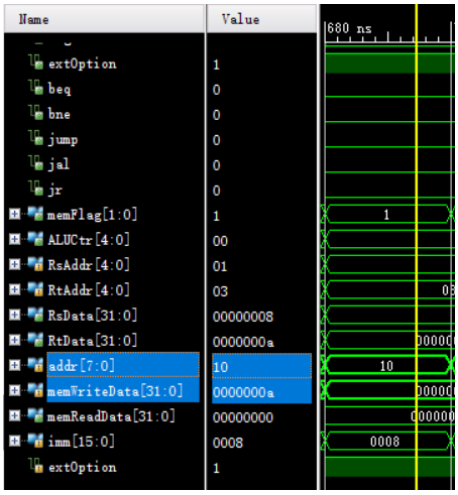
lui = 1



35.

address	instruction	op	rs	rt	immediate	code	result
0x00000078	sh $3, 8($1)	101001	00001	00011	0000 0000 0000 1000	a4230008	mem[16:17]=10

该指令正确：



mem[16:17] = 00000000 00001010

pc = 00000078

memWrite = 1

addr = 0x10

memWriteData=0xa

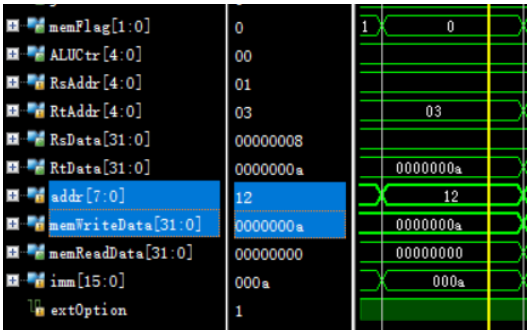
此时DMEM的状态为：

mem[12:17] = 00000000 00000000 00000000 00000010 00000000 00001010

36.

address	instruction	op	rs	rt	immediate	code	result
0x0000007C	sb $3, 10($1)	101000	00001	00011	0000 0000 0000 1010	a023000a	mem[18]=10

该指令正确：mem[18] = 00001010



pc = 0000007C

memWrite = 1

addr = 0x1

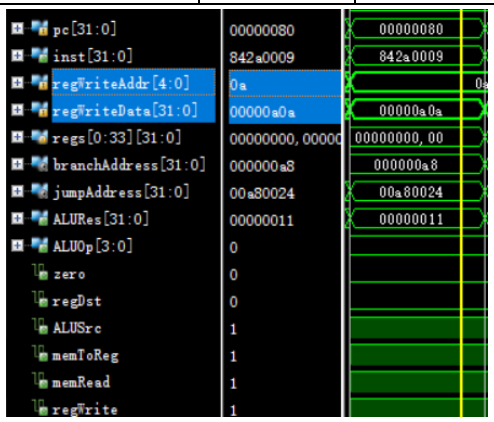
memWriteData=0xa

此时DMEM的状态为：mem[12:18] = 00000000 00000000 00000000 00000010 00000000 00001010 00001010

37.

address	instruction	op	rs	rt	immediate	code	result
0x00000080	lh $10, 9($1)	100001	00001	01010	0000 0000 0000 1001	842a0009	$10=0x00000a0a

该指令正确：mem[17:18] = 00001010 00001010



pc = 00000080

regWriteAddr=0xa

regWriteData=0x00000a0a

regWrite（寄存器写入使能信号）=1

38.

address	instruction	op	rs	rt	immediate	code	result
0x00000084	lb $10, 4($1)	100000	00001	01010	0000 0000 0000 0100	802a0004	$10 = 0

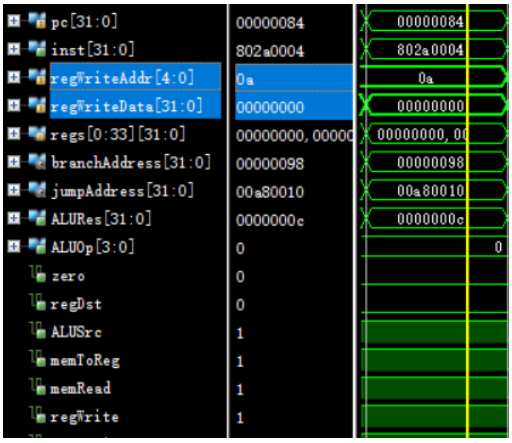
该指令正确：mem[12] = 00000000

pc = 00000084

regWriteAddr (写入的寄存器地址) =0xa

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1



39.

address	instruction	op	address	code	result
0x00000088	jal 0000008C	000011	0000008c	0c000023	

该指令正确：将pcAdd4存入$31并跳转

pc = 00000088

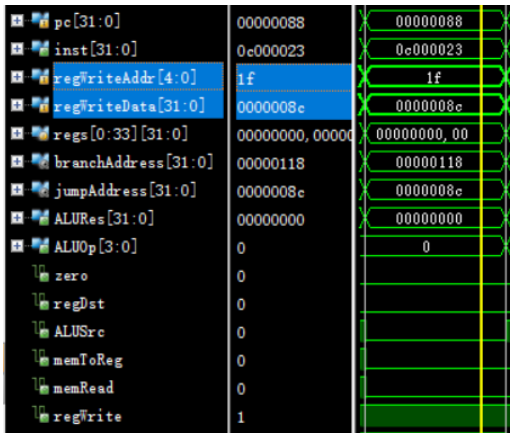
jal = 1

jumpAddress = 0x8C

$ra = $31 = pcAdd4 = 0x8C

故：regWriteAddr = 0x1f($31)

regWriteData = 0x8C



40.

address	instruction	op	rs	rt	immediate	code	result
0x0000008C	ori $1,$0,0x00000098	001101	00000	00001	0000 0000 0110 0010	34010098	$1=0x00000098

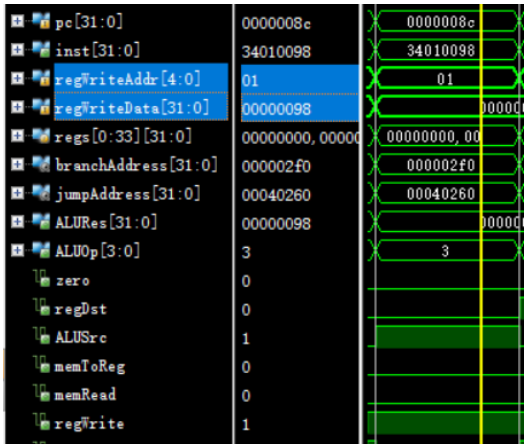
该指令正确：$1 = 0 or 0x98 = 0x98

pc = 0000008C

regWriteAddr（写入的寄存器地址）=1

regWriteData（向寄存器写入的值）=0x98

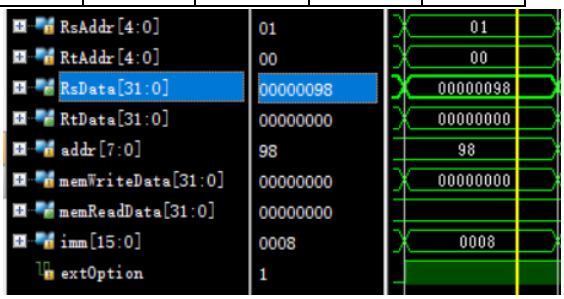
regWrite（寄存器写入使能信号）=1



41.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000090	jr $1	000000	00001	00000	00000	00000	001000	00200008	

该指令正确：$1=0x98，跳转到9x98



pc = 00000090

jr = 1

RsAddr = 1

RsData = 0x98

42.

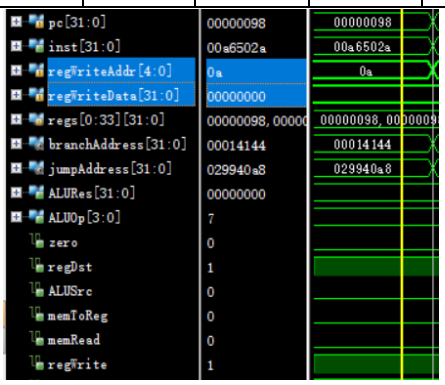
address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000094	add $10,$1,$2	000000	00001	00010	01010	00000	000010	00225020	

该指令会被跳过，正确。

43.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000098	slt $10,$5,$6	000000	00101	00110	01010	00000	101010	00a6502a	$10 = 0

该指令正确：



$5 = A0000 > $6 = 1， 故$10 = 0

pc = 00000098

regWriteAddr(写入的寄存器地址)=0xa

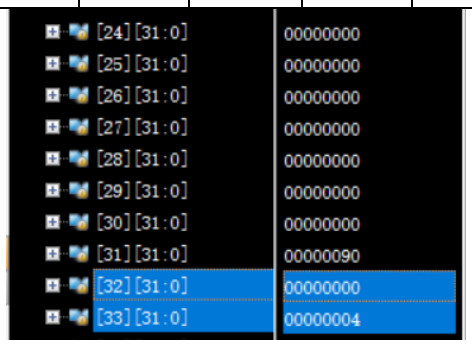
regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1

44.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000009C	mult $9,$2	000000	01001	00010	00000	00000	011000	01220018	hi = 0 lo = 4

该指令正确：$9 * $2 = 2*2 = 4



pc = 0000009C

hi = 0($32 = 0)

lo = 4($33 = 4)

regWrite（寄存器写入使能信号）=1

45.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x000000A0	mfhi $10	000000	00000	00000	01010	00000	010000	00005010	$10 = 0

该指令正确：$10 = $hi = 0

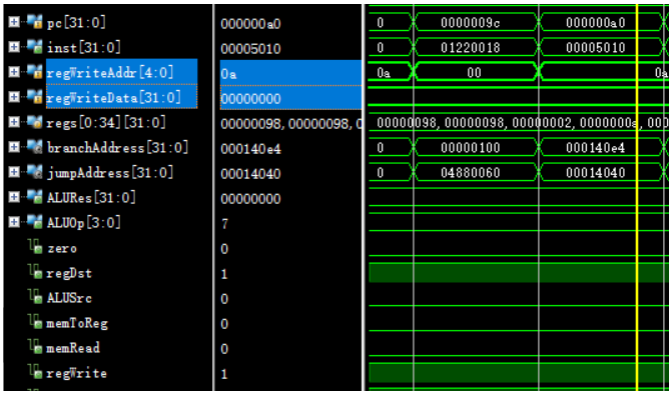
pc = 000000A0

regWriteAddr (写入的寄存器地址) =0xa

regWriteData（向寄存器写入的值）=0

regWrite（寄存器写入使能信号）=1

mfhi = 1



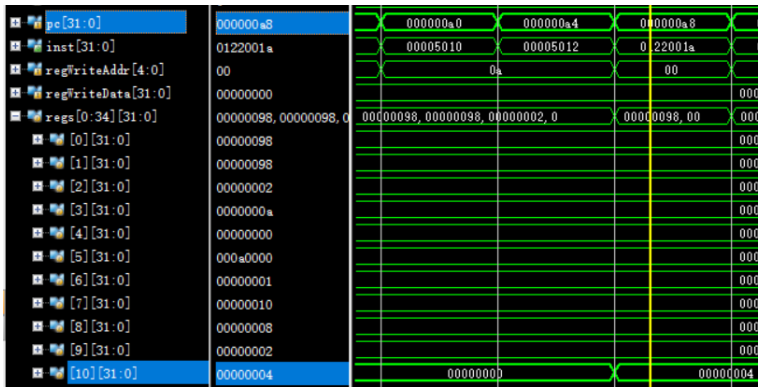
46.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x000000A4	mflo $10	000000	00000	00000	01010	00000	010010	00005012	$10 = 4

该指令正确：$10 = $lo = 4

pc = 000000A4

regs[10] = 4, mflo = 1



47.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x000000A8	div $9, $2	000000	01001	00010	00000	00000	011010	0122001a	hi=1.lo=0

该指令正确：$hi = $9 / $2 = 2 / 2 = 1

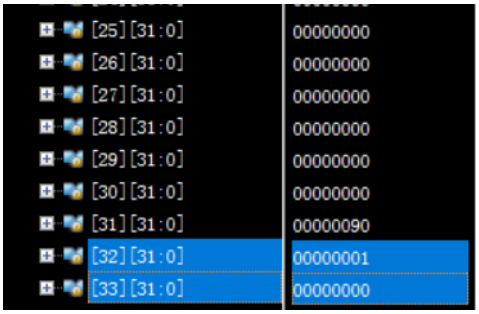
$lo = $9 % $2 = 2 % 2 = 0

pc = 000000A8

hi = 1($32 = 1)

lo = 0($33 = 0)

div = 1



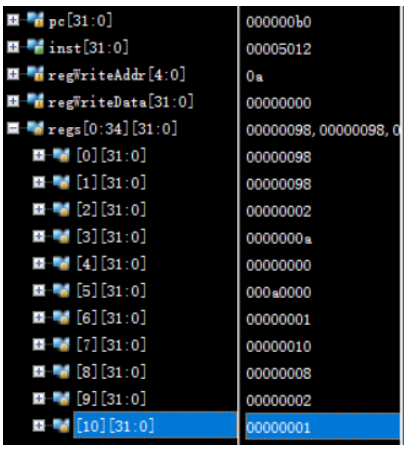
48.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x000000AC	mfhi $10	000000	00000	00000	01010	00000	010000	00005010	$10 = 1

该指令正确：$10 = $hi = 1

pc = 000000AC

regs[10] = 1



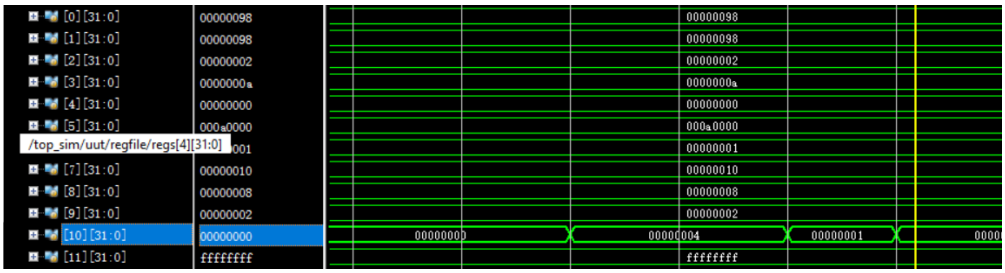
49.

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x000000B0	mflo $10	000000	00000	00000	01010	00000	010010	00005012	$10 = 0

该指令正确：$10 = $lo = 0

pc = 000000B0

regs[10] = 0



50.

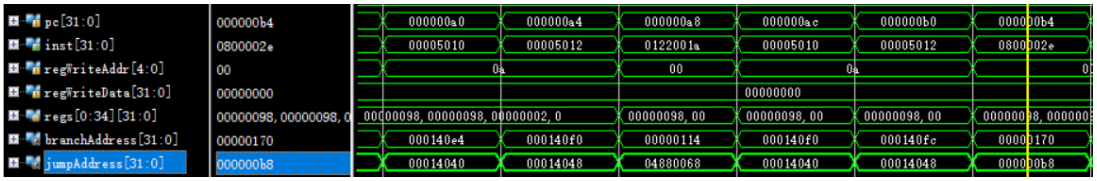
address	instruction	op	address	code	result
0x000000B4	j 0x000000B8	000010	000000b8	0800002e	

该指令正确：跳转到0xB8处

pc = 000000B4

jump = 1

jumpAddress = 0xB8



51.

address	instruction	op	rs	rt	immediate	code	result
0x000000B8	haul	111111	00000	00000	0000000000000000	fc000000	haul

该指令正确：停机，pc不再改变，结束

pc = 000000B0

3. 实现（在Basys3板上运行所设计的CPU）

（1）display模块设计

设计思路：sm_wei控制数码管的哪一位亮，sm_duan控制数码管的段码。首先对时钟信号clk进行分频。由于数码管只能显示16位，通过hi_lo信号控制显示高16位或低16位

代码及注释如下：

```
`timescale 1ns / 1ps

module display(clk,data,hi_lo,sm_wei,sm_duan);

input clk;
input [31:0] data;
input hi_lo;//1: 高 16 位; 0:低 16 位
output [3:0] sm_wei;
output [6:0] sm_duan;

reg [3:0]wei_ctrl=4'b1110;
reg clkdiv;

integer cnt;

always @(posedge clk)
    if(cnt==32'd100_000)
    begin
    cnt <= 1'b0;
    clkdiv <= ~clkdiv;
    end
    else cnt <= cnt + 1'b1;

reg [3:0]duan_ctrl;

always @(posedge clkdiv)
wei_ctrl <= {wei_ctrl[2:0],wei_ctrl[3]};

always @(wei_ctrl)

    case(hi_lo)
    1'b0:
        case(wei_ctrl)
        4'b1110:duan_ctrl=data[3:0];
        4'b1101:duan_ctrl=data[7:4];
        4'b1011:duan_ctrl=data[11:8];
        4'b0111:duan_ctrl=data[15:12];
        default:duan_ctrl=4'hf;
```

```
        endcase
    1'b1:
        case(wei_ctrl)
        4'b1110:duan_ctrl=data[19:16];
        4'b1101:duan_ctrl=data[23:20];
        4'b1011:duan_ctrl=data[27:24];
        4'b0111:duan_ctrl=data[31:28];
        default:duan_ctrl=4'hf;
        endcase
    endcase

reg [6:0]duan;

always @(duan_ctrl)
    case(duan_ctrl)
    4'h0:duan=7'b100_0000;//0
    4'h1:duan=7'b111_1001;//1
    4'h2:duan=7'b010_0100;//2
    4'h3:duan=7'b011_0000;//3
    4'h4:duan=7'b001_1001;//4
    4'h5:duan=7'b001_0010;//5
    4'h6:duan=7'b000_0010;//6
    4'h7:duan=7'b111_1000;//7
    4'h8:duan=7'b000_0000;//8
    4'h9:duan=7'b001_0000;//9
    4'ha:duan=7'b000_1000;//a
    4'hb:duan=7'b000_0011;//b
    4'hc:duan=7'b100_0110;//c
    4'hd:duan=7'b010_0001;//d
    4'he:duan=7'b000_0111;//e
    4'hf:duan=7'b000_1110;//f
    // 4'hf:duan=7'b111_1111;//不显示
    default : duan = 8'b1100_0000;
    endcase

assign sm_wei = wei_ctrl;
assign sm_duan = duan;

endmodule
```

（2）修改top文件

在top文件中添加display模块的实例化：

```
display dis(
    .clk(clk),
    .data(ALURes),
    .hi_lo(hi_lo),
    .sm_wei(sm_wei),
    .sm_duan(seg)
);
```

修改输入输出：

```
3   module top(
4       input clk,//clk for display
5       input clkin,//clkin for instruction
6       input reset,
7       input hi_lo,
8       // output wire [31:0] inst,
9       // output reg [31:0] pc,
10      // output reg [31:0] pcAdd4
11      output[3:0] sm_wei,
12      output[6:0] seg
13  );
```

（3）引脚连接（约束文件设计）

引脚设计：七段数码管显示数据，一个拨号开关用于hi_lo信号，一个按钮用于reset信号，一个按钮用于clk信号（用于控制top中时钟），板子上的时钟信号用于display模块的时钟信号。

代码如下：

CPU > CPU.srcs > constrs_1 > new > display_pin.xdc

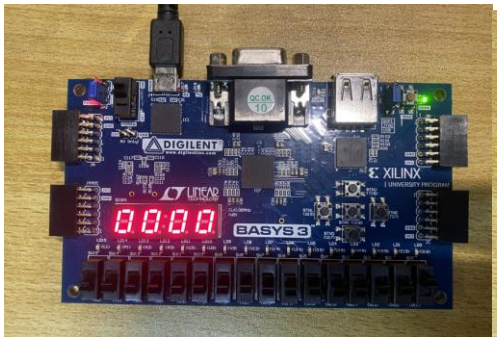
```
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
set_property PACKAGE_PIN U2 [get_ports {sm_wei[0]}]
set_property PACKAGE_PIN U4 [get_ports {sm_wei[1]}]
set_property PACKAGE_PIN V4 [get_ports {sm_wei[2]}]
set_property PACKAGE_PIN W4 [get_ports {sm_wei[3]}]
set_property PACKAGE_PIN U18 [get_ports clkin]
set_property PACKAGE_PIN W5 [get_ports clk]
set_property PACKAGE_PIN V17 [get_ports hi_lo]
set_property PACKAGE_PIN T18 [get_ports reset]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clkin]


set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports clkin]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports hi_lo]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
```

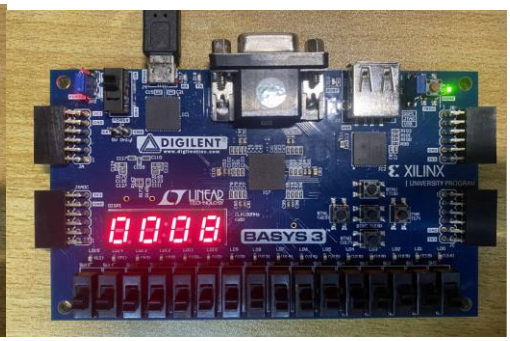
（4）烧写到basy3板上后的运行结果

第一条指令

address	instruction	op	rs	rt	immediate	code	result
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	24010008	$1 = 8

高16位、低16位：

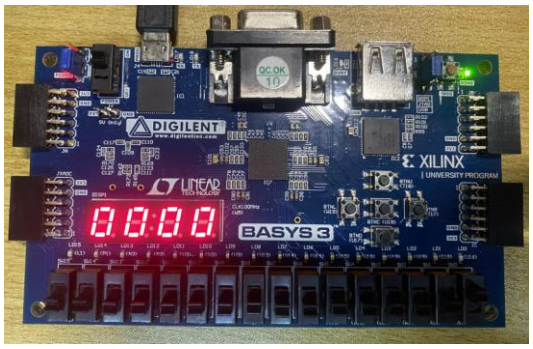


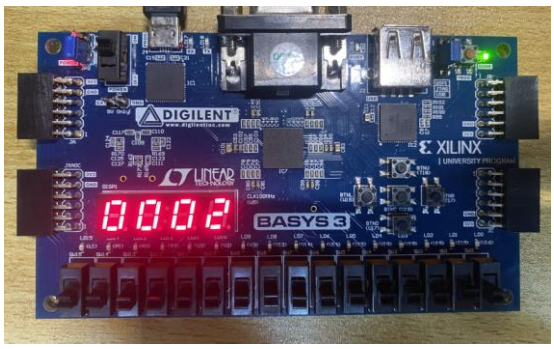


第二条指令：

address	instruction	op	rs	rt	immediate	code	result
0x00000004	ori $2,$0,2	001101	00000	00010	0000 0000 0000 0010	34020002	$2 = 2

高16位、低16位：

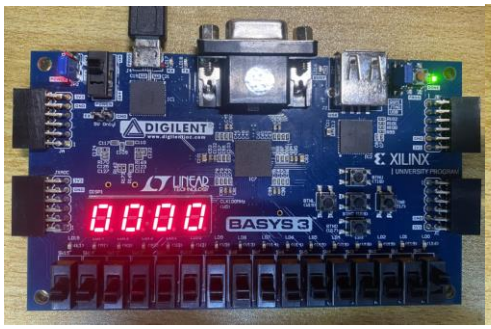


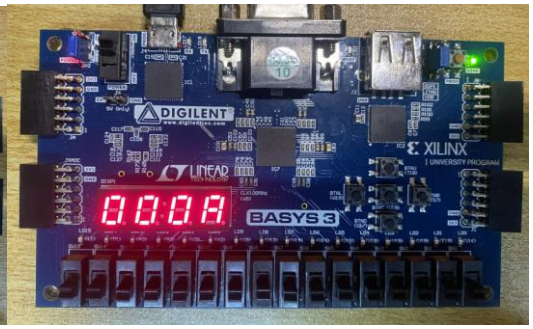


第三条指令：

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000008	add $3,$2,$1	000000	00010	00001	00011	00000	100000	00411820	$3 = 10

高16位、低16位：

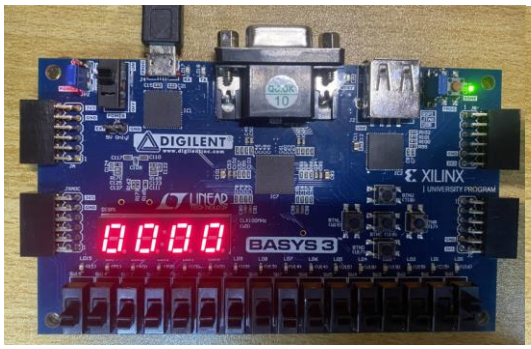


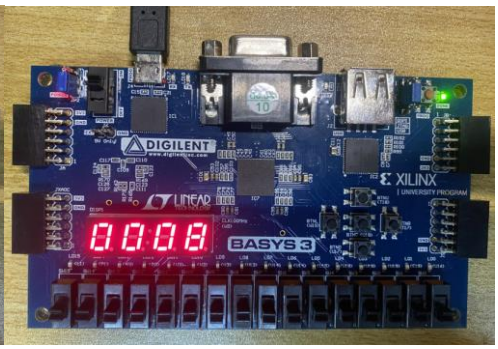


第四条指令：

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101	00000	100010	00622822	$5 = 8

高16位、低16位：

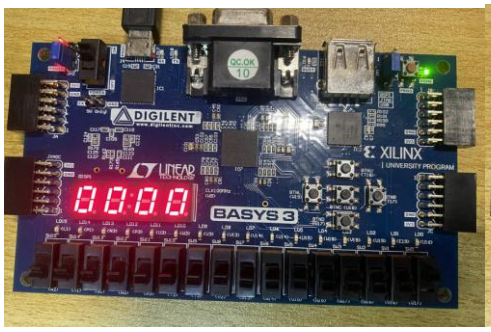
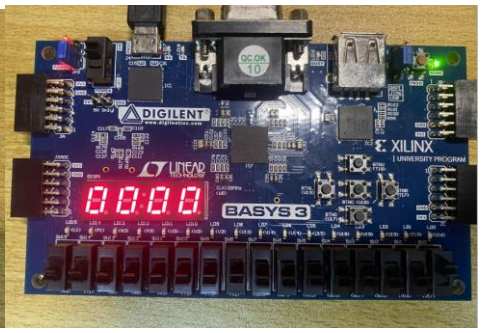




第五条指令：

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000010	and $4,$5,$2	000000	00101	00010	00100	00000	100100	00a22024	$4 = 0

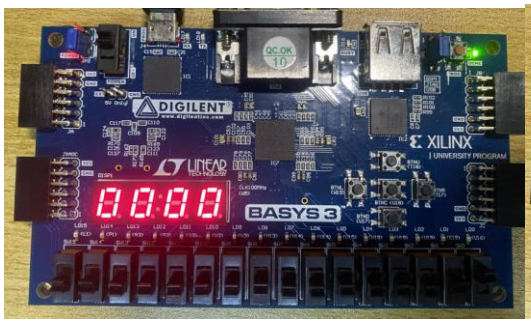
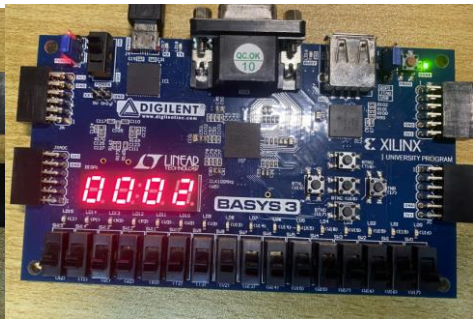
高16位、低16位：

第六条指令：

address	instruction	op	rs	rt	rd	shamt	funct	code	result
0x00000014	or $8,$4,$2	000000	00100	00010	01000	00000	100101	00824025	$8 = 2

高16位、低16位：

六. 实验心得

本次实验前前后后花了一周的时间，在这整个过程中学到了很多东西，也是第一次写这种复杂度的一个实验。首先是要从高抽象上设计各个模块，把每个模块的功能划分好，尽量解耦，让功能尽可能地分离并且容易修改。然后是要理清各个模块之间的联系，不可漏掉任何要传输的信号（数据信号、使能信号），并且要考虑冒险等问题。最后是具体的实现，编写代码的时候比较考验耐心，特别是对照着指令表一条条地对照每个情况的时候，重复中又带有不同，十分容易出错。写好代码之后，要设计测试指令集，让所有指令都能体现它们的功能，可以利用mars内置的mips汇编器来转换成机器代码，此后的仿真、综合、实现、烧写到板上都要花费一定的时间。

一开始我遇到了指令的读取与pc之间存在延时的问题，通过对IMEM进行硬编码存储解决了这个问题。在拓展指令集的过程中，遇到了一些verilog语法上的问题，通过查阅资料解决了这个问题。整个实验锻炼了我的能力，我认为上述实验也还有改进的地方，就是模块的设计可以更加完善，让功能更加分离，比如说我采取了shifer内置在ALU内的实现，但shifter的设计可以独立出来，与ALU共享一些信号，让整个设计更好维护；以及一些信号可以设计成接收参数的函数。