中国科学院大学计算机组成原理实验课

实 验 报 告

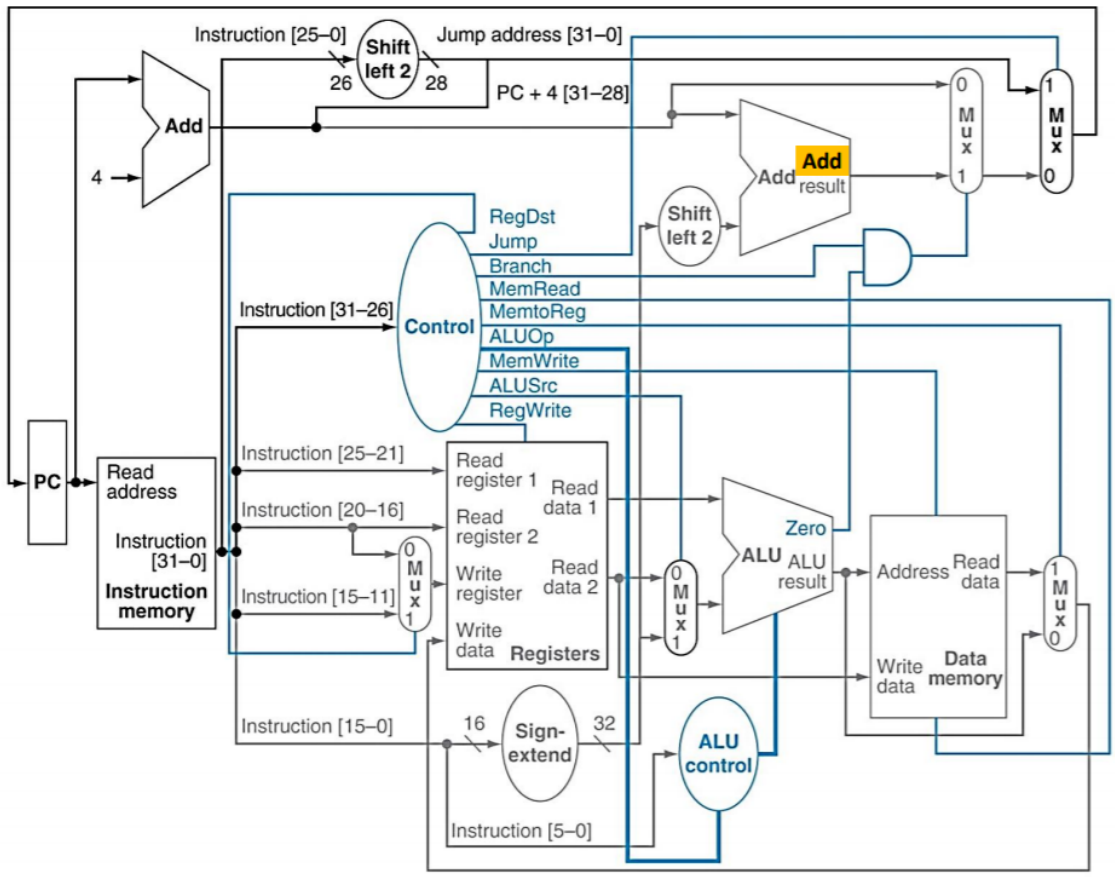
学号：2019K8009915022 姓名：栗祺菁 专业：计算机科学与技术

实验序号：prj2 实验名称：MIPS单周期处理器设计

一、 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

仿真波形：由平台自动判断相关变量值的对错，不在此赘述。

下面是逻辑电路结构，信号名有些差异：



下面介绍几个主要的模块

为了方便阅读，采取伪码形式，省略了部分信号的声明语句：wire语句

0.extend模块

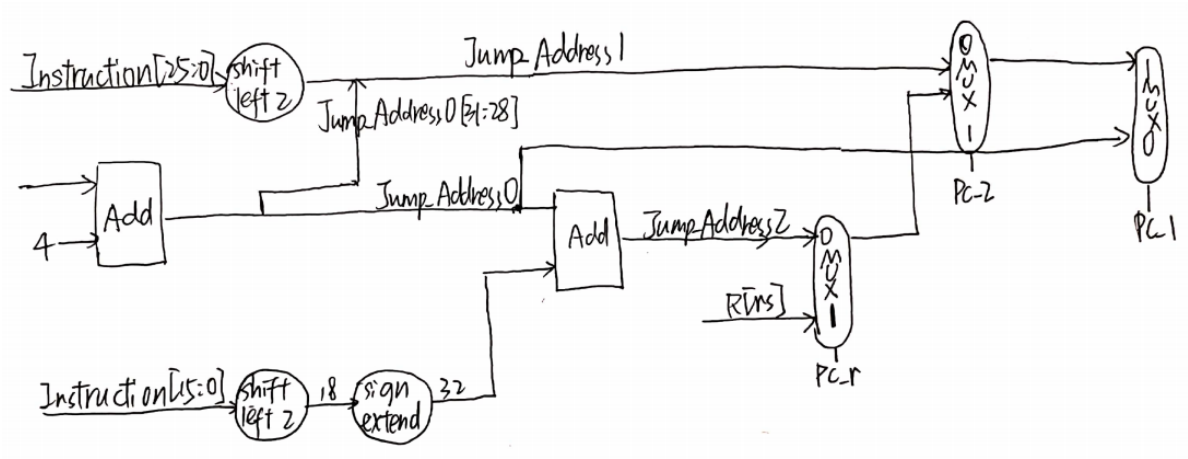
为了后续使用方便，先将立即数的有符号位扩展和无符号扩展都给出来

此处的imm指的是Instruction的后十六位

```
    assign sign_extend_imm={{16{imm[15]}},imm};
    assign zero_extend_imm={16'b0,imm};
```

1.PC模块

原理图，与 上图中PC部分类似，信号名有部分差异，最后增加一个MUX选择器，多了一个选择即R[rs]

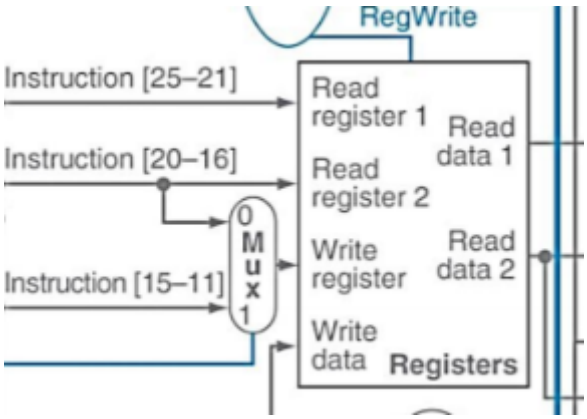



```
//根据不同指令的内容，下一个PC值有四种可能：
//R[rs]
//PC+4（Jump_address0)
//{PC+4高四位，指令低26位左移两位}(Jump_address1)
//PC+4+{imm左移两位后进行符号位扩展}(Jump_address2)

    //probable PC
    assign Jump_address0=PC+4;//fit for most Instructions
    assign Jump_address1={Jump_address0[31:28],Instruction[25:0],2'b00}; //j,jal
    assign Jump_address2=Jump_address0+{{14{imm[15]}},imm,2'b00};
        //branch Instruction:bltz,bgez,beq,bne,blez,bgtz

//根据不同的指令，选择合适的next PC值
//以PC_r代表next PC来自寄存器，
//以PC_1代表next PC是Jump_address1，
//以PC_2代表next PC是Jump_address2，
//剩下的情况中，next PC均为PC+4，即Jump_address0

    //next PC
    assign PC_r=(opcode==6'b000000 && {func[5:3],func[1]}==4'b0010);//jr,jalr
    assign PC_1=(opcode[5:1]==5'b00001);//j,jal
    assign PC_2=(opcode==6'b000001 && ((!rt[0] && !zero)||(rt[0] && zero)))
        || (opcode==6'b000100 && zero)
        || (opcode==6'b000101 && !zero)
        || (opcode==6'b000110 && (!zero || reg_read_data1==32'b0))
        || (opcode==6'b000111 && (zero && reg_read_data1!=32'b0));
    //bltz,bgez,beq,bne,blez,bgtz

    //时序逻辑,下一拍上升沿到达的时候改变PC的值
    always@(posedge clk or posedge rst)
        begin
            if(rst) PC<=32'b0;//复位时，PC值变为0
            else if(PC_r)
                PC<=reg_read_data1;//R[rs]
            else if(PC_1)
                PC<=Jump_address1;
            else if(PC_2)
                PC<=Jump_address2;
            else
                PC<=Jump_address0;
```

```
        end
```

2.Register 模块

原理图与下图类似，在Write register端口再增加一个MUX选择器，增加一个可能，即31。

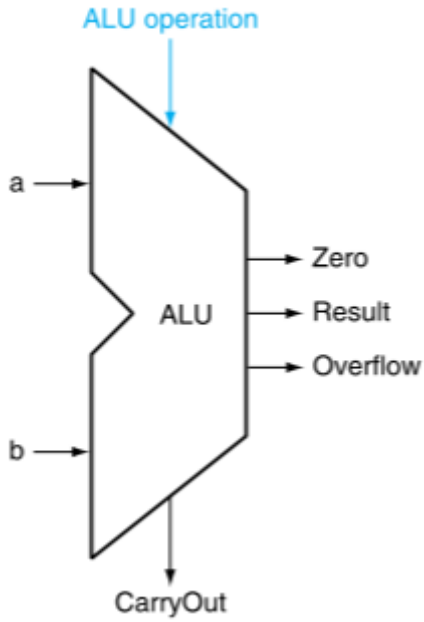



```
//根据不同的指令，寄存器的写地址有三种可能：
//rd:所有的R-type型指令，除了jr（jr:PC=R[rs]，只改变PC值，但不写寄存器，与之类似的
jalr:R[rd]=PC+8 PC=R[rs],不仅改变PC值，还写寄存器）
//rt：load指令（R[rt]=modified_read_data)或者I-type计算指令(R[rt]=R[rs] op
extend_imm)
//31：jal指令（jal：R[31]=PC+8 PC=R[rs]）

    assign rd_write=(opcode==6'b000000 && func!=6'b001000);
    assign rt_write=(opcode[5:3]==3'b001 || (opcode[5]&(~opcode[3])) );

    assign RF_waddr=rd_write?rd:
                    rt_write?rt:
                    (opcode==6'b000011)?5'b11111:
                    5'b00000;//rd or rt or 31(jal)

//写使能有效时，有三种情况：
//除去jr的R-type指令，并且如果是move类型的话，要求比较的结果zero相应地为0或者1，才能够拉高写
使能信号
//load指令或I-type计算指令，即rt_write信号有效时
//jal指令
    assign RF_wen=
        (rd_write && (({func[5:3],func[1]}!=4'b0011) || (func[0]==0 &&
zero==1)|| (func[0]==1 && zero==0)))
        || (rt_write)
        || (opcode==6'b000011);

//寄存器写的内容有六种情况：
//alu的输出结果：R-type运算指令和I-type计算指令（除去lui，属于移位指令单独考虑）
//PC+8:jalr或者jal指令
//R[rs]：R-type mov指令：movez,moven
//shifter输出的结果：R-type移位指令
//{imm，16‘b0}：lui指令
//modified_read_data：load指令

    assign alu_result_write=(opcode==6'b000000 && func[5]==1'b1)
        || (opcode[5:3]==3'b001 && opcode!=6'b001111);
    assign PC_write=(opcode==6'b000000 && func==6'b001001)
        || (opcode==6'b000011);
```

```
    assign reg_data_write=(opcode==6'b000000 && {func[5:3],func[1]}==4'b0011);
    assign shift_write=(opcode==6'b000000 && func[5:3]==3'b000);

    assign RF_wdata=alu_result_write?alu_result:
    PC_write?PC+8:
    reg_data_write?reg_read_data1:
    shift_write?shift_result:
    (opcode==6'b001111)?{imm,16'b0}:
    MemRead?modified_read_data:32'b0;
    //alu_result or PC+8 or R[rs] or shift_result or lui or modified_read_data

//reg_file模块的实例化
//读地址始终为rs和rt,并将读出来的值分别定义为reg_read_data1和reg_read_data2
reg_file
my_reg_file(.clk(clk),.rst(rst),.waddr(RF_waddr),.raddr1(rs),.raddr2(rt),
.wen(RF_wen),.wdata(RF_wdata),.rdata1(reg_read_data1),.rdata2(reg_read_data2));
```

3.ALU

ALU原理图:



ALUop对应的各项操作如下:

```
wire op_and=ALUop==3'b000;
wire op_or=ALUop==3'b001;
wire op_add=ALUop==3'b010;
wire op_sltu=ALUop==3'b011;
wire op_xor=ALUop==3'b100;
wire op_nor=ALUop==3'b101;
wire op_sub=ALUop==3'b110;
wire op_slt=ALUop==3'b111;
```

对于新增加的无符号数比较sltu操作，可以将两个操作数前面都再拼接一个0，再进行减法，则实现了无符号数比较大小。

```
//ALUop1对应R-type的运算指令
```

```
//ALUop2对应I-type计算指令
//REGIMM指令和I-type分支指令中的blez和bgtz，需要用到slt比较操作
//R-type的mov指令和I-type分支指令中的beq和bne指令，需要用到sub操作
//load或store指令，需要用到add操作
    assign ALUop1=(func[3:2]==2'b00)?{func[1],2'b10}:
        (func[3:2]==2'b01)?{func[1],1'b0,func[0]}:
        (func[3:2]==2'b10)?{~func[0],2'b11}:3'b000;//R-type-calculate code

    assign ALUop2=(opcode[2:1]==2'b00)?{opcode[1],2'b10}:
        (opcode[2]==1'b1)?{opcode[1],1'b0,opcode[0]}:
        (opcode[2:1]==2'b01)?{~opcode[0],2'b11}:3'b000;//I-type-calculate code

    assign ALUop=(opcode==6'd000000 && func[5]==1'b1)?ALUop1:
        (opcode[5:3]==3'b001)?ALUop2:
        (opcode==6'b000001 || opcode[5:1]==5'b00011)?3'b111:
        (opcode[5:1]==5'b00010 || (opcode==6'd000000 &&
{func[5:3],func[1]}==4'b0011))?3'b110:
        (opcode[5])?3'b010:3'b000;


    //sign_in表示操作数B是符号位扩展，zero_in表示操作数B是无符号扩展
    assign sign_in=(opcode[5] || opcode[5:2]==4'b0010);//store or load or
slti,sltiu,addiu
    assign zero_in=opcode[5:2]==4'b0011; //andi,ori,xori

    //move 信号表示是否是R-type的move操作，如果是的话，操作数B变为R[rt]
    assign move=(opcode==6'b000000 && {func[5:3],func[1]}==4'b0011);//R-type-
move

    assign A=move?reg_read_data2:reg_read_data1;//if move A=R[rt] else A=R[rs]
    assign B=sign_in?sign_extend_imm:
        zero_in?zero_extend_imm:
        move?5'b0:reg_read_data2;

    //实例化ALU部件
    alu my_alu(.A(A),.B(B),.ALUop(ALUop),.Overflow(Overflow),
.CarryOut(CarryOut),.Zero(zero),.Result(alu_result));
```

4.Data memory

代码部分不在此赘述，简要说明思路如下：

首先用n作为有效地址（alu_result)的最后两位，将地址改为Address={alu_result[31:2],2'b00}

对于load操作：

对于lw,lwr,lwl指令：

按照偏移量n和指令类型将Read_data做适当的偏移，作为shift_read_data

用mask作为掩码，表示shift_read_data的哪些字节是有效的，即R[rt]的哪些字节需要被替换掉

对于lb,lbu,lh,lhu指令：

用select_byte和select_half选出Read_data有效的一个字节或者两个字节，之后按照指令类型分别与0或者符号位进行拼接

最后综合所有情况得到modified_read_data，即可以直接写到register的数

下面是mask与目标字节值的关系：

```
    wire[7:0] byte_0;
    wire[7:0] byte_1;
    wire[7:0] byte_2;
    wire[7:0] byte_3;
    assign byte_0=mask[0]?shift_read_data[7:0]:reg_read_data2[7:0];
    assign byte_1=mask[1]?shift_read_data[15:8]:reg_read_data2[15:8];
    assign byte_2=mask[2]?shift_read_data[23:16]:reg_read_data2[23:16];
    assign byte_3=mask[3]?shift_read_data[31:24]:reg_read_data2[31:24];
```

对于store操作:

相比起load更加简单，用Write_strb表示mem中应该更新的字节

用Write_data表示修改从register中取出来的R[rt]，按要求进行拼接操作，以保证需要用到的字节在Write_strb有效的位置上

5.shifter模块

比较简单。以func[1:0]作为Shiftop。操作数A1是R[rt]，操作数B1可能是shame或者R[rs]

对于算数右移的情况，将变量在前面拼接32个符号位，然后再移动，最后取低32位就可以了。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如RTL代码中出现的逻辑bug，仿真、云平台调试过程中的难点等）

实验过程中有一些细节得注意，比如rs==5'b11011这样的语句中，5'b是不能省略的，否则右边的值将被自动扩展为32位，左右永远不相等，这个问题在写代码的时候没有注意，有的地方没有忘记，有的地方忘记了，导致多次调试波形的时候，发现都是这里出了问题，当然最好不用这样的表达式。还有一些诸如将：写成；的粗心大意犯的错误，应该养成好的习惯，及时查看修改，而不是等到编译的时候报错，再回头查看。还有就是ALU无符号数比较操作以及shifter的算数右移操作，最开始都没有想明白逻辑，之后才靠拼接一个0和拼接32个符号位解决。

这个实验目前的版本还有一些小问题没有更改，考试周时间繁忙，只能说下次写代码的时候注意了。例如，在PC模块，可以用组合逻辑定义一个next_pc，下一拍时直接将p赋值为next_pc就行。还有就是原文中用了大量的？：表达式，这样在运行时就产生了层次结构，实际上若条件互斥，直接用与或逻辑就可以表达（将条件信号扩展为32位或5位再和目标赋值相与），可以提高运行效率。此外，ALUop可以整理总结出相应的真值表，将ALUop的各位用与op和func相关的与或非逻辑表达式表示。在最后的load操作部分，我的原代码使用了mask,shift_read_data,select_byte,select_half等变量来构造modified_read_data,我觉得还可简化，在位移或者拼接部分应该统一起来，不过那样可读性应该没有现在强。

三、 对讲义中思考题（如有）的理解和回答

四、 在课后，你花费了大约12小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

上完课之后觉得还挺简单，老师讲得如此通俗易懂了，应该上手很容易。但开始看45条指令时，就开始迷茫了，我都不清楚每一条指令的含义是啥，费劲周章我将所有指令的简要含义写在了一个表里（这个表后续有多次修改），然后就开始一个个模块看了。第一个模块是PC模块，next PC有四种可能的值，然后确定next PC取这四个值的分别是是哪些指令，指令比较多的话就将一组指令综合起来产生一个控制信号（例如PC_2，指代的是所有跳转地址为PC+4+{{14{imm[15]}},imm,2'b00}的指令），若PC_2为高电平，就将PC赋值为PC+4+{{14{imm[15]}},imm,2'b00}，然后写extend，Register，Alu，Memory，shifter模块，在整理总结指令共性操作的部分，都是在草稿纸上完成的，过于潦草，也不放上来了。其中Memory 模块花了大量的时间调试，主要是对于store和load变式指令的含义不清楚，对于偏移量和原始数据的拼接问题开始没想明白，后来经过刘泽昊同学的提醒，以及助教王嵩岳学长的帮助，终于弄明白了含义，完成了实验。

这个实验因为是单周期的，涉及时序的只有PC模块和register模块，大部分内容都是组合逻辑，一旦想清楚了主要框架就可以开始写了，还比较简单。事实证明，万事开头难，看PPT看了将近一天，感觉没有啥头绪，决定先写写看，写着写着就越来越明白它要干嘛了。多周期MIPS指令实验，听完老师的课感觉也不是很难（狗头），只能说考试完有时间再看吧，考试周确实是有点赶。