电子科技大学计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 计算机系统结构综合实验

电子科技大学教务处制表

实 验 报 告

学生姓名：王涛　　学 号：2020080902001　　指导教师：王华

实验地点：清水河校区主楼 A2 区 413-1　　**实验时间：2023.5.18**

一、实验室名称：国家级计算机实验教学示范中心

二、实验项目名称：解决控制冒险问题

三、实验学时：4

四、实验原理：

一）控制冒险：当程序中遇到分支和跳转指令的时候，可能会改变流水线的执行顺序，依次进入流水线的指令不一定会被执行，因而产生冒险。所以必须进行检测分析以保证流水线执行结果的正确性。

二）如何检测出控制冒险？

1. 译码级检测

1）分支指令

本系统指令集中包含两条分支指令：

- beq　rs,rt,label　//if(rs==rt)　PC←label
- bne　rs,rt,label　//if(rs！=rt)　PC←label

分支指令的重要判定条件是两个寄存器值是否相等，正常情况下这个 exe_z 信号在 EXE 级产生，届时后面有两条指令已经进入流水线。如果在 ID 级增加比较电路，提前获取 exe_z 信号，就让后面进入流水线的指令变为一条，这是方案的第一步。即，将 exe_z 信号变为 id_z 信号，表达式为：

id_z =(rs== rt)?1'b1 : 1'b0;　或者

id_z = ~|(rs ^rt);

接下来，如果分支条件满足，已经进入取指阶段的指令对流水线来说就不应该执行，所以应该考虑如何让该指令对流水线不造成任何影响。这是第二步。

具体操作方案是：让取指阶段的 IF_Inst 信号受控。而跟分支指令密切相关的是 pcsource 信号。修改 IF_Inst 的表达式如下：

IF_Inst = (pcsource ==2'b01) ? 32'h0 : IF_Inst_org;

这里增加一个 IF_Inst_org 信号，表示按照流水线顺序从 inst_ROM 中取出的指令。32'h0 是一条无效指令，即如果分支成立，进入流水线的就是一条无效指令，否则就一切照旧。

2）跳转指令

本系统包含一条跳转指令

jump　target　//PC←target

该指令是无条件跳转，生命周期只有 IF 和 ID 两个阶段，所以只需要考虑对进入取指阶段的指令进行处理即可，修改 IF_Inst 为：

IF_Inst = (pcsource ==2'b10) ? 32'h0 : IF_Inst_org;

信号代表的含义同上所述。

综合考虑分支和跳转指令，IF_Inst 信号的表达式最终为：

IF_Inst = (pcsource==2'b01||pcsource==2'b10) ? 32'h0 : IF_Inst_org;

2. 执行级检测

这里只涉及分支指令。在执行级检测就不需要额外添加比较电路了。此时要考虑的问题有：

➢ 分支条件成立时，已经进入流水线 IF 和 ID 级的指令需要作废

操作方案：需修改 IF_Inst 和 ID_Inst：

IF_Inst = (pcsource==2'b01||pcsource==2'b10) ? 32'h0 : IF_Inst_org;

ID_Inst = (pcsource ==2'b01) ? 32'h0 : ID_Inst_org;

➢ 分支条件成立时，在原来 ID 级生成的 bpc 信号是否还有效？

答案：无效，因为在 ID 级生成的 bpc 信号与当前 exe 级的分支指令无关。

操作方案：

（1）重新计算 bpc 信号，其表达式如下：

bpc=branch?(exe_pc4+{ {14{EXE_Inst[25]}} ,EXE_Inst[25:10],2'b00})
: id_bpc;

branch 信号表达式如下：

branch = ((EXE_Inst[31:26]==6'b001111 && exe_z == 1'b1) || (EXE_Inst[31:26]==6'b010000 && exe_z == 1'b0)) ? 1'b1 : 1'b0;

branch 信号代表分支是否成功（高电平有效），由于分支在 exe 级进行检测，所以需要通过 EXE_Inst 和 exe_z 信号状态进行判定。还需要通过 ID/EXE 流水线寄存器多传递一个 EXE_Inst 信号和 exe_pc4 信号。

（2）直接在 ID_EXE 流水线寄存器多传递一个 bpc 信号，ID 级产生的 bpc 为 id_bpc，传递到 EXE 级的为 exe_bpc

bpc=branch?exe_bpc : id_bpc;

➢ 分支条件成立时，ID 级生成的 pcsource 信号是否有效？

答案：无效，因为原来的 pcsource 信号的产生条件之一 op 均是 ID 级的 op，而分支指令已经在 EXE 级，ID 级指令是另外一条指令，所以需要分开进行判断。具体的表达式为：

pcsource = branch ? 2'b01:ID_Inst[31:26] == 6'b010010 ?2'b10:2'b00;

综上，必须定义 branch 信号。

➢ 分支条件成立时，ID 级生成的写信号是否有效？

答案：分支条件成立，意味着其后进入流水线的指令无效，因此，在 ID 级生成的写信号无效，应该重新进行描述。具体的操作方案是：

id_wmem = branch? 1'b0 : id_wmem_org;

id_wreg = branch? 1'b0 : id_wreg_org;

各个信号代表的含义同前所述。

五、实验目的：

1. 进一步掌握流水线 CPU 和单周期 CPU 的区别；
2. 进一步熟悉 Verilog HDL 硬件设计语言；
3. 熟悉和掌握开发平台 Xilinx ISE Design Suite 14.7 集成开发系统的操作方法；
4. 进一步理解和掌握流水线控制冒险的概念和解决方法。

六、实验内容：

1. 掌握控制冒险问题的成因，充分理解采用 ID 级检测和 EXE 级检测两种方式解决分支指令控制冒险问题的原理，以及如何解决跳转指令的控制冒险问题；
2. 完成在 ID 级检测解决控制冒险问题，分析具体代码并给出仿真结果，结合 inst_ROM 中的指令序列进行详尽分析说明
3. 完成在 EXE 级检测解决控制冒险问题，对于跳转指令仍然只能在 ID 级进行检测。分析具体代码并给出仿真结果，结合 inst_ROM 中的指令序列进行详尽分析说明
4. 结合数据冒险，设计一个能处理两种冒险的系统。

七、实验器材：

同实验一

八、实验步骤：

同实验一

九、实验数据及结果分析：

ID 级检测解决控制冒险

1.修改后的模块介绍

1) 顶层模块

完整代码如下：

```
module SCCPU(Clock, Resetn, PC, if_Inst, exe_Alu_Result,mem_mo,pcsource,id_z,id_ra,id_rb
    );
    input Clock, Resetn;//输入的时钟及复位信号
    output [31:0] PC, if_Inst, exe_Alu_Result,mem_mo; //输出当前 PC 值，指令，alu 计算结果，DM 取出的数据
    output [1:0] pcsource; //输出当前 pcsource
    output id_z; //ID 级的操作数是否相等
    output [31:0] id_ra,id_rb; //两个操作数

    wire [31:0] mem_Alu_Result, wb_Alu_Result;//ALU 结 t 果
    wire [1:0] pcsource;  //PC 选择器

    wire [31:0] bpc, jpc, if_pc4, id_pc4, id_Inst,if_Inst; //pc 及指令

    wire [31:0] wdi, id_ra, exe_ra, id_rb, exe_rb, mem_rb, id_imm, exe_imm;
    wire id_m2reg, exe_m2reg,mem_m2reg,wb_m2reg;//写回数据选择器

    wire id_wmem, exe_wmem, id_aluimm, exe_aluimm, id_shift, exe_shift, id_z;
```

```
        wire [2:0] id_aluc, exe_aluc; //alu 操作码

        wire id_wreg, exe_wreg, mem_wreg, wb_wreg; //写寄存器堆信号传递
        wire [4:0] id_rn, exe_rn, mem_rn, wb_rn; //rn 传递

        wire [31:0] mem_mo, wb_mo;//DM 取出的数据

        wire [31:0] if_Inst_org; //初始的指令序列

        //取指阶段
        IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, if_pc4, if_Inst_org, PC);

      //如果是跳转指令，指令为 0，否则为原指令
      assign if_Inst = (pcsource == 2'b01 || pcsource == 2'b10) ? 32'h0 : if_Inst_org;

        //锁存并传递 pc4，Inst
        IF_IDreg    IF_ID (Clock,Resetn,if_pc4,if_Inst,id_pc4,id_Inst);

        //译码阶段
        ID_STAGE stage2 (id_pc4, id_Inst, wdi, Clock, Resetn, bpc, jpc, pcsource,
                         id_m2reg, id_wmem, id_aluc, id_aluimm, id_ra, id_rb, id_imm,
                          id_shift, id_z,id_wreg,id_rn);       //将原本的 Z 信号使用 id_z 代替，
id_z 表示 branch 指令的两个操作数是否相等

        //锁存并传递   m2reg, wmem, aluc, aluimm, ra, rb, imm, shift，wreg，rn
        ID_EXEreg ID_EXE (Clock,Resetn,id_m2reg,id_wmem,id_aluc,id_aluimm,
                id_ra,id_rb,id_imm,id_shift,id_wreg,id_rn,exe_m2reg,exe_wmem,
                exe_aluc,exe_aluimm,exe_ra,exe_rb,exe_imm,exe_shift,exe_wreg,exe_rn);

        //执行阶段
         EXE_STAGE stage3 (exe_aluc, exe_aluimm, exe_ra, exe_rb, exe_imm, exe_shift,
exe_Alu_Result,
                z,exe_m2reg,exe_wmem,exe_wreg,exe_rn);

        //锁存并传递 Alu_Result, rb，wmem，m2reg，wreg，rn
                                                                    EXE_MEMreg
EXE_MEM(Clock,Resetn,exe_Alu_Result,exe_rb,exe_wmem,exe_m2reg,exe_wreg,exe_rn,
                mem_Alu_Result,mem_rb,mem_wmem,mem_m2reg,mem_wreg,mem_rn);

        //访存阶段
          MEM_STAGE   stage4   (mem_wmem,   mem_Alu_Result[4:0],   mem_rb,   Clock,
mem_mo,
                mem_wreg,mem_m2reg,mem_rn);
```

```
    //锁存并传递 Alu_Result，m2reg，wreg，rn，mo
                                                                MEM_WBreg
MEM_WB(Clock,Resetn,mem_Alu_Result,mem_mo,mem_m2reg,mem_wreg,mem_rn,
            wb_Alu_Result,wb_mo,wb_m2reg,wb_wreg,wb_rn);

    //写回寄存器堆
    WB_STAGE stage5 (wb_Alu_Result, wb_mo, wb_m2reg, wdi,wb_wreg,wb_rn);


endmodule
```

2) 涉及到修改的子模块（ID 级模块）
完整代码如下：

```
module ID_STAGE(pc4,inst,
                wdi,clk,clrn,bpc,jpc,pcsource,
                   m2reg,wmem,aluc,aluimm,a,b,imm,
                   shift,id_z,wreg,rn
    );
    input [31:0] pc4,inst,wdi;          //pc4-PC 值用于计算 jpc；inst-读取的指令；
wdi-向寄存器写入的数据
    input clk,clrn;       //clk-时钟信号；clrn-复位信号；
    input id_z;      //判断 qa 和 qb 是否相等
    output [31:0] bpc,jpc,a,b,imm;          /c-branch_pc；jpc-jump_pc；a-寄存器操
作数 a；b-寄存器操作数 b；imm-立即数操作数
    output [2:0] aluc;          //ALU 控制信号
    output [1:0] pcsource;          //下一条指令地址选择
    output m2reg,wmem,aluimm,shift,wreg;
    output [4:0] rn;

    wire wreg;
    wire [4:0] rn;          //写回寄存器号
    wire [5:0] op,func;
    wire [4:0] rs,rt,rd;
    wire [31:0] qa,qb,br_offset;
    wire [15:0] ext16;
    wire regrt,sext,e;

    assign func=inst[25:20];
    assign op=inst[31:26];
    assign rs=inst[9:5];
    assign rt=inst[4:0];
    assign rd=inst[14:10];
    Control_Unit cu(id_z,func,                                  //控制部件
```

```
                    op,wreg,m2reg,wmem,aluc,regrt,aluimm,
                      sext,pcsource,shift);

      Regfile rf (rs,rt,wdi,rn,wreg,~clk,clrn,qa,qb);//寄存器堆，有 32 个 32 位的寄存器，0 号寄存器恒为 0，对时钟去反，实现前半周期写入，后半周期读取
      mux5_2_1 des_reg_num (rd,rt,regrt,rn); //选择目的寄存器是来自于 rd,还是 rt

      assign a=qa;
      assign b=qb;
      assign id_z = (a == b) ? 1'b1 : 1'b0;//判断 qa 和 qb 是否相等，用于 beq,bne 指令

      assign e=sext&inst[25];//符号拓展或 0 拓展
      assign ext16={16{e}};//符号拓展
      assign imm={ext16,inst[25:10]};      //将立即数进行符号拓展

      assign br_offset={imm[29:0],2'b00};      //计算偏移地址
      add32 br_addr (pc4,br_offset,bpc);      //beq,bne 指令的目标地址的计算
      assign jpc={pc4[31:28],inst[25:0],2'b00};      /指令的目标地址的计算

endmodule
```

说明：在 ID 级处理控制冒险，只需要在 ID 级将原本 ALU 返回的 Z 信号替换为 id_z，即不在需要 ALU 的计算结果标志位来判断，只需要在 ID 级判断 qa 与 qb 是否相等就能得到 id_z 信号，从而替代原本得 Z 信号存入控制单元，进行译码得到 pcsource，然后再主模块根据 pcsource 的值，如果为 01 或 10（选择 bpc 或 jpc），则表示有控制冒险发生（需要跳转），因此将 IM 输出的指令赋为 0 传给下一阶段（相当于插入一个 nop 指令），然后下一周期的 PC 就是跳转后的正常地址，就消除了控制冒险。

2. 初始化部分

1）寄存器堆初始化（该模块完整代码）：

同实验一

2）数据存储器初始化（该模块完整代码）：

同实验一

3）指令存储器初始化：

指令地址	汇编指令	机器指令	备注
其余	/	/	无冒险 3 条
03 06	load r9,0x0001(r4) add r1,r1,r1	32'h34000489 32'h00100421	结构冒险 2 条
02 04	bne r1,r2,6'h04 beq r1,r7,6'h08	32'h40000422 32'h3c000c27	分支指令 2 条
05	jump 0x0000001	32'h48000001	跳转指令 1 条

4）
该模块完整代码：

```
      assign rom[6'h00]=32'h00000000;      //0 地址为空，从 1 地址开始执行；
       assign rom[6'h01]=32'h00101464;          //add r5,r3,r4    r5=0x00000007
       assign rom[6'h02]=32'h40000422;            //bne r1,r2,6'h04 offset=0x0001
      //bne  中 r1 与 r2 不相等，满足转移条件，发生转移出现控制冒险，将下条指令清 0

      assign rom[6'h03]=32'h34000489;          //load r9,0x0001(r4)    该语句实际不会执行
      assign rom[6'h04]=32'h3c000c27;          //beq r1,r7,6'h08    相等转移到 08H 处  offset=0x0003
      //beq 不满足转移条件，不发生转移，继续执行下一条语句

       assign rom[6'h05]=32'h48000001;           //jump 0x0000001    无条件转移到 01h 处
      //jump 语句转移，出现控制冒险，将下条指令清 0
      //jump 语句转向第一条语句，因此后三条实际不执行
      assign rom[6'h06]=32'h00100421;           //add r1,r1,r1
      assign rom[6'h07]=32'h04200823;           //or r2,r1,r3
      assign rom[6'h08]=32'h0821a408;           //sr1 r9,r8,5'h03
```

3. 仿真
仿真测试代码如下：

```
module PICPU_tb;

    // Inputs
    reg Clock;
    reg Resetn;

    // Outputs
    wire [31:0] PC;
    wire [31:0] if_Inst;
    wire [31:0] exe_Alu_Result;
    wire [31:0] mem_mo;
    wire [1:0] pcsource;
    wire id_z;
    wire [31:0] id_ra;
    wire [31:0] id_rb;

    // Instantiate the Unit Under Test (UUT)
    SCCPU uut (
```

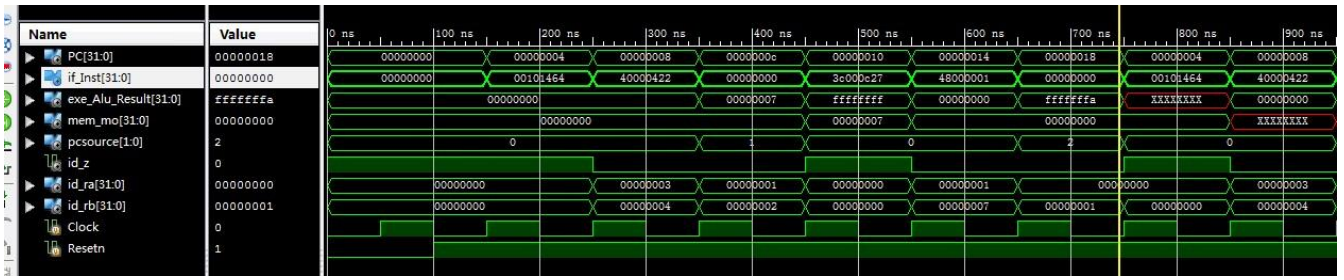
```
        .Clock(Clock),
        .Resetn(Resetn),
        .PC(PC),
        .if_Inst(if_Inst),
        .exe_Alu_Result(exe_Alu_Result),
        .mem_mo(mem_mo),
        .pcsource(pcsource),
        .id_z(id_z),
        .id_ra(id_ra),
        .id_rb(id_rb)
    );

    initial begin
        // Initialize Inputs
        Clock = 0;
        Resetn = 0;

        // Wait 100 ns for global reset to finish
        #100;
        Resetn=1;
        // Add stimulus here

    end
     always#50 Clock=~Clock;
endmodule
```

仿真结果如下（截图）：



仿真结果分析：

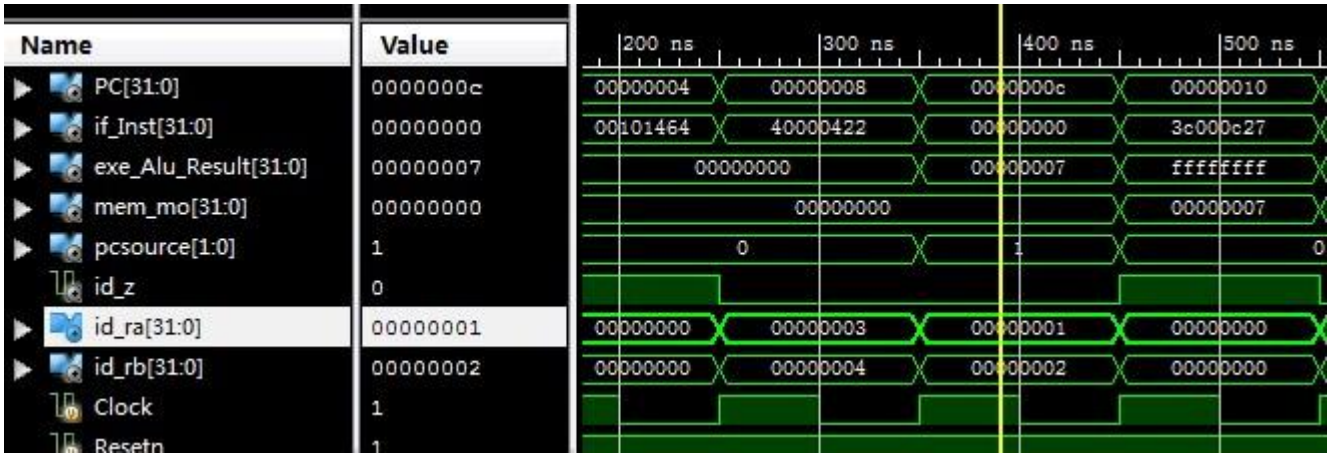
根据设计的指令集，可以得到以下时空图，从中可以看出存在两处控制冒险。第一处是在第二条 bne 指令，bne 的两个操作数不相等跳转条件成立，需要跳转到第四条指令，但是在 IF 级第三条指令已经进入，因此产生了控制冒险，同理第五条 jump 指令同样存在控制冒险。

时钟周期数	2	3	4	5	6	7	8	9	10	11	12
add r5,r3,r4	add	r3,r4	+		r5						
bne r1,r2,6'h04		bne	r1,r2	-							
load r9,0x0001(r4)			load	r4	+	mem	r9				
beq r1,r7,6'h08				beq	r1,r7	-					
jump 0x0000001					jump	addr					
add r1,r1,r1						add	r1,r1	+		r1	
or r2,r1,r3							add	r3,r4	+		r5
sr1 r9,r8,5'h03											

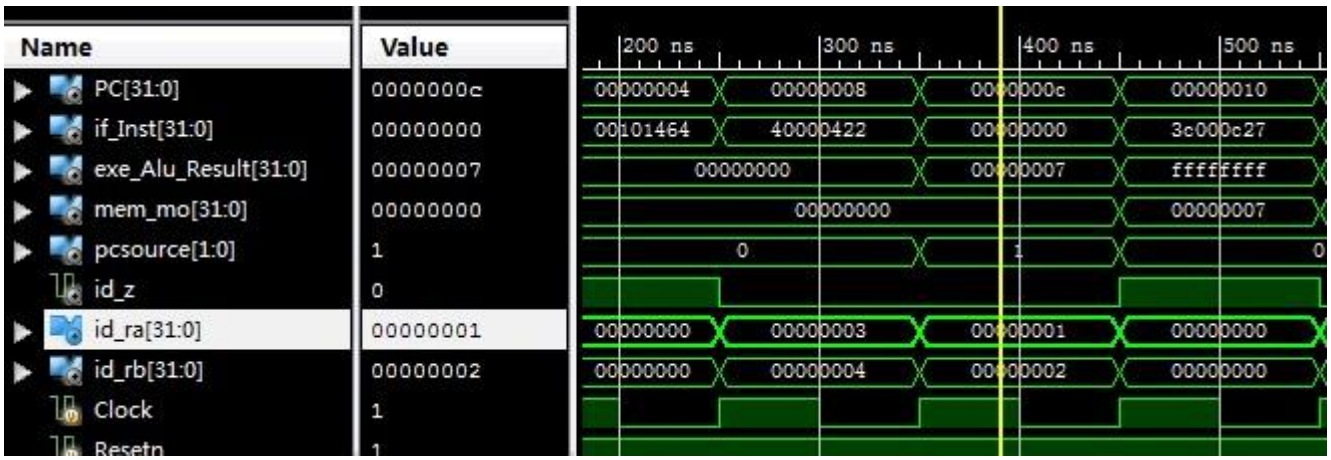
根据预期的结果，解决控制冒险后的时空图应该如下图所示，以第一处控制冒险为例，在第三个时钟周期 ID 级译码得知为 bne 指令，且得到 id_z 信号为 0，满足跳转条件。因此将第四个时钟周期取指的指令赋为 0，相当于插入一条 nop 指令，此时等待第五个时钟周期 IF 取指第四条 beq 指令，完成了控制冒险的消除。对于第二个 jump 指令引发的控制冒险同理。

时钟周期数	2	3	4	5	6	7	8	9	10	11	12
add r5,r3,r4	add	r3,r4	+		r5						
bne r1,r2,6'h04		bne	r1,r2	-							
nop			nop								
beq r1,r7,6'h08				beq	r1,r7	-					
jump 0x0000001					jump	addr					
nop						nop					
add r5,r3,r4							add	r3,r4	+		r5
bne r1,r2,6'h04											

如下图所示为解决第一个控制冒险的仿真结果图，可以看到在在 200-300ns 周期 IF 取指 bne，然后 300-400ns 周期，bne 指令进入 ID 级译码，然后将 IF 级取出的指令赋值为 0，这样插入一个 nop，然后再 400-500ns 周期取出 beq 指令继续执行，控制冒险被消除。

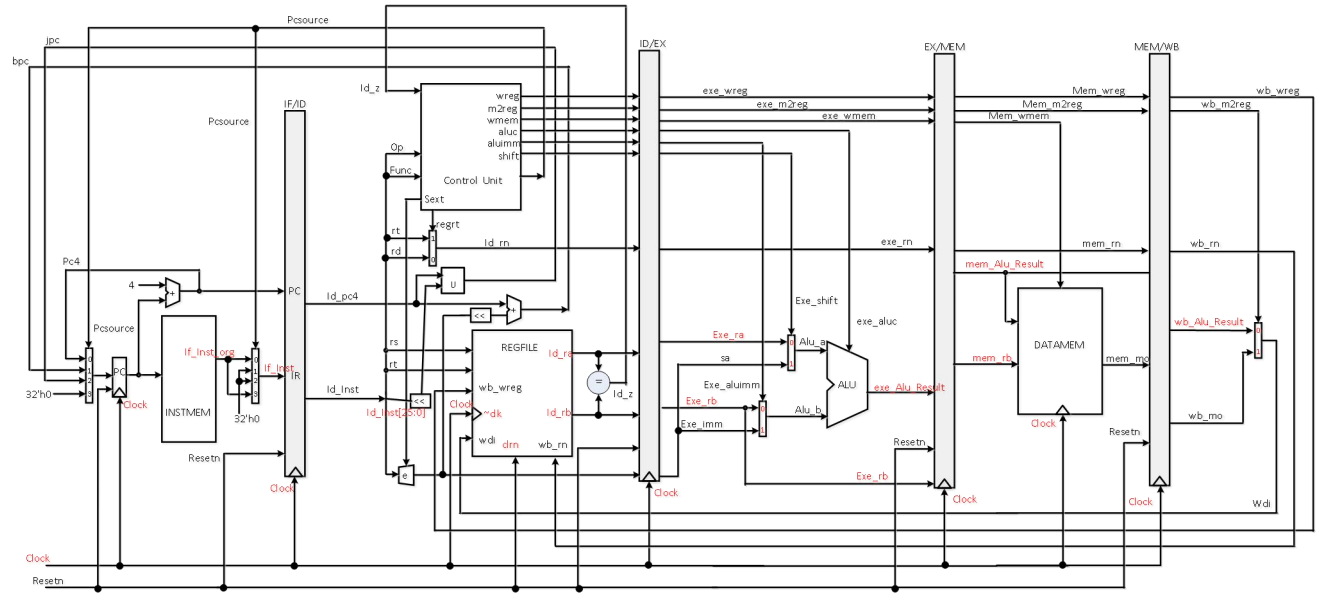


同理，下图为 jump 指令引发的控制冒险解决后的仿真图，其效果与预期效果，原理也与第一个控制冒险的消除相同，因此不再赘述。



结论：**结果符合预期。**

4. 解决控制冒险的五级流水线 CPU 架构如下：



十、实验结论：（联系理论知识进行说明）

在本次实验中，在 ID 级解决控制冒险所需要更改的代码 1 并不多，但是更改是需要注意连锁反应，在加入 id_z 信号和 if_Inst 后，还需要将传入控制单元的 Z 信号更改为 id_z 信号，以便获取正确的 pcsource，同样在指令的设计中不应该包含其他类型的冒险，因为代码中只有处理控制冒险的逻辑，加入其他冒险会得到错误的结果。

十一、总结及心得体会：

谈谈本次实验给你带来的收获和思考。

通过本次实验，学习了控制冒险的处理方法，了解到了解决控制冒险的多个方法以及控制冒险和数据冒险的一起处理方法，实现用到的方法在理论知识中没

有讲解，但是通过实验也逐渐的理解了，同样发现在 ID 级来处理控制冒险的简易性，也了解了每种方法各自的优缺点，加深了对冒险问题的理解和解决冒险的认识。

十二、对本实验过程及方法、手段的改进建议：

实验可以将几种冒险都结合起来，然后设计电路来进行处理，可以使用多种方法来进行电路设计，分析其中的优缺点以及流水线的性能。

报告评分：

指导教师签字：