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： 卢晓雅 **学 号：2020080904026** **指导教师：王华**

实验地点： 清水河校区主楼 A2 区 **实验时间：5 月 19 日下午**

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**（需修改流水线 CPU 架构图）。**

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：

一）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：

```
22  module SCCPU(Clock, Resetn, PC,
23     id_ra, id_rb, id_z, pcsource,
24     if_inst_org, if_inst, id_inst,
25     exe_Alu_Result
26      );
27      input Clock, Resetn;
28      output [31:0] PC, if_inst, exe_Alu_Result, if_inst_org, id_ra, id_rb, id_inst;
29      output id_z;
30      output [1:0] pcsource;
31
32      wire [31:0] mem_Alu_Result, wb_Alu_Result;
33
34      wire [31:0] mem_mo, wb_mo; // 各阶段的inst,Alu_Result, mo
35      wire [31:0] m5, r9;          // rom5和reg9的内容，检验指令是否正确执行
36
37      wire [31:0] bpc, jpc, if_pc4, id_pc4;
```

```
38
39      wire [31:0] wdi;
40      wire [31:0] exe_ra, exe_rb, mem_rb;
41      wire [31:0] id_imm, exe_imm;
42
43      wire [4:0] id_rn, exe_rn, mem_rn, wb_rn;
44
45      wire id_wreg, exe_wreg, mem_wreg, wb_wreg;
46      wire id_m2reg, exe_m2reg, mem_m2reg, wb_m2reg;
47      wire id_wmem, exe_wmem, mem_wmem, id_aluimm, exe_aluimm;
48      wire [2:0] id_aluc, exe_aluc;
49      wire id_shift, exe_shift;
50
51      // 将stall信号传入PC和IR
52      IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, if_pc4, if_inst_org, PC);
53
54      // 如果转移，则废除分支或转移指令之后的指令，换成32'h0
55      assign if_inst = (pcsource ==2'b01 || pcsource ==2'b10) ? 32'h0 : if_inst_org;
56
57      assign id_z = (id_ra == id_rb)?1'b1:1'b0;  // 将操作数的比较提前到ID 阶段
58      // 或者 id_z = ~|( id_a ^ id_b );
59
60      IF_ID IR (if_pc4, if_inst, Clock, Resetn, id_pc4, id_inst);
61
62      ID_STAGE stage2 (id_pc4, id_inst, id_rn, wdi, Clock, Resetn, bpc, jpc, pcsource,
63                  id_m2reg, id_wmem, id_aluc, id_aluimm, id_ra, id_rb, id_imm, r9, id_shift, id_wreg,
64                  id_z, wb_wreg, wb_rn);
```

```
65
66      ID_EXE id_exe_reg (Clock, Resetn,
67                  id_m2reg,id_wmem,id_aluc,id_aluimm,id_shift,id_wreg,
68                  id_ra,id_rb,id_imm,id_rn,
69                  exe_m2reg,exe_wmem,exe_aluc,exe_aluimm,exe_shift,exe_wreg,
70                  exe_ra,exe_rb,exe_imm,exe_rn
71                  );
72
73      //z信号直接连到pc模块，不使用nop的情况下不会正确转移
74      EXE_STAGE stage3 (exe_aluc, exe_aluimm, exe_ra, exe_rb, exe_imm, exe_shift, exe_Alu_Result, z);
75
76      EXE_MEM exe_mem_reg (Clock, Resetn,
77              exe_Alu_Result, exe_rb, exe_wmem, exe_m2reg, exe_wreg, exe_rn,
78              mem_Alu_Result, mem_rb, mem_wmem, mem_m2reg, mem_wreg, mem_rn
79      );
80
81      // Alu_Result[6:2]报错：Cannot index into non-array mem_Alu_Result
82      // 将Alu_Result[6:2]改为直接传入完整的Alu_Result信号
83      MEM_STAGE stage4 (mem_wmem, mem_Alu_Result, mem_rb, Clock, mem_mo, m5);
84
85      MEM_WB mem_wb_reg (Clock, Resetn,
86                  mem_Alu_Result,mem_m2reg,mem_wreg,mem_rn,mem_mo,
87                  wb_Alu_Result,wb_m2reg,wb_wreg,wb_rn,wb_mo);
88
89      WB_STAGE stage5 (wb_Alu_Result, wb_mo, wb_m2reg, wdi); //wdi写入寄存器堆的数据直连到ID_STAGE即可
90
91  endmodule
```

将操作数的比较提前到 ID 阶段，产生 zero 标志 id_z，Control_Unit 不需要修改，只要将 id_z 传入，代替之前代码中的 rsrtequ 即可。

当转移发生时，废除转移指令的后一条指令，即将该指令改为 0，所以将 IF_STAGE 生成的信号改成 if_inst_org，if_inst = (pcsource ==2'b01 || pcsource ==2'b10) ? 32'h0 : if_inst_org; 将 if_inst 传入 IR 寄存器。

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
6'h01	addi r1,r1,0x0004	32'h14001021	
6'h02	or r4,r5,r6	32'h042010a6	
6'h03	ori r6,r2,0x00cc	32'h28033046	无冒险 3 条
6'h04	beq r1,r1,6'h03	32'h3c000c21	分支指令，跳转到地址 0x08
6'h05	addi r1,r1,0x0005	32'h14001421	
6'h08	addi r1,r1,0x0004	32'h14001021	
6'h09	beq r3,r4,6'h06	32'h3c001883	分支指令，不相等则不跳转
6'h0a	add r5,r3,r4	32'h00101464	
6'h0b	jump 0x0000001	32'h48000001	跳转指令
6'h0c	and r2,r2,r1	32'h04100841	

该模块完整代码：

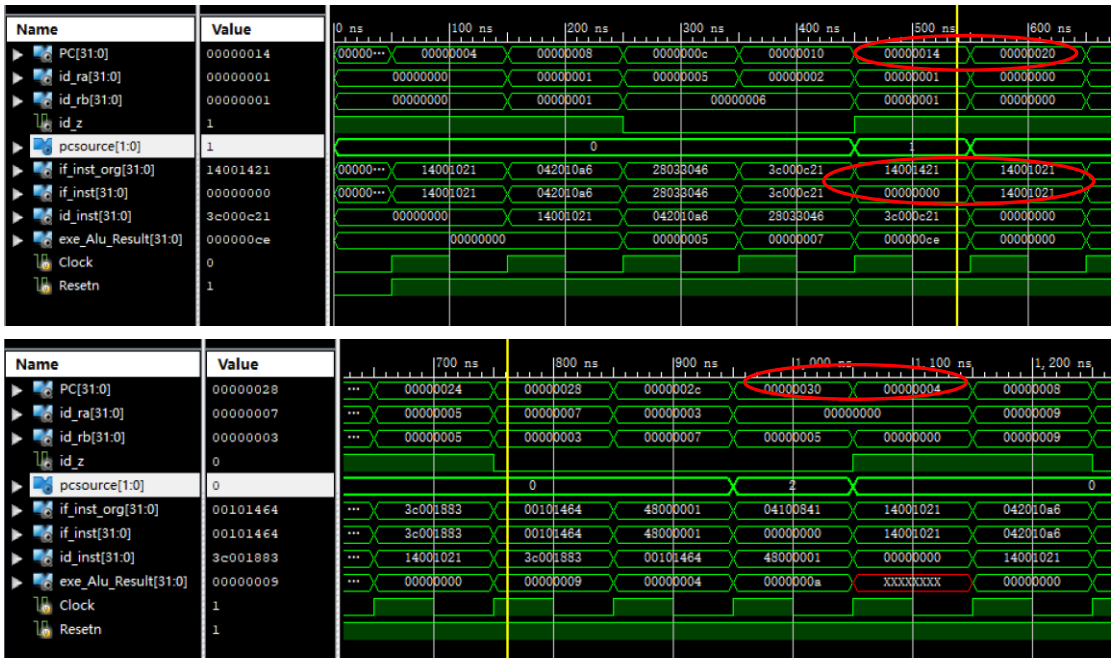
```
module Inst_ROM(a,inst
    );
    input [5:0] a;
    output [31:0] inst;
    wire [31:0] rom [0:63];

    assign rom[6'h00]=32'h00000000;

    assign rom[6'h01]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000005
    assign rom[6'h02]=32'h042010a6;    //or r4,r5,r6  r4=0x00000007
    assign rom[6'h03]=32'h28033046;    //ori r6,r2,0x00cc  r6=0x000000ce
    assign rom[6'h04]=32'h3c000c21;    //beq r1,r1,6'h03  offset=0x0003, 相等则转移到5+3=0x08

    assign rom[6'h05]=32'h14001421;    //addi r1,r1,0x0005  r1=0x0000000a

    assign rom[6'h08]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000009
    assign rom[6'h09]=32'h3c001883;    //beq r3,r4,6'h06  offset=6, 不相等则不跳转

    assign rom[6'h0a]=32'h00101464;    //add r5,r3,r4  r5=0x00000007
    assign rom[6'h0b]=32'h48000001;    //jump 0x0000001  无条件转移到01h处

    assign rom[6'h0c]=32'h04100841;    //and r2,r2,r1  r2=0x00000000

    assign inst=rom[a];
endmodule
```

仿真测试代码如下：

```
module SSCPU_test;

    // Inputs
    reg Clock;
    reg Resetn;

    // Outputs
    wire [31:0] PC, id_ra, id_rb;
    wire id_z;
    wire [1:0] pcsource;
    wire [31:0] if_inst_org;
    wire [31:0] if_inst, id_inst;
    wire [31:0] exe_Alu_Result;

    // Instantiate the Unit Under Test (UUT)
    SCCPU uut (
        .Clock(Clock),
        .Resetn(Resetn),
        .PC(PC),
        .id_ra(id_ra),
        .id_rb(id_rb),
        .id_z(id_z),
        .pcsource(pcsource),
        .if_inst_org(if_inst_org),
        .if_inst(if_inst),
        .id_inst(id_inst),
        .exe_Alu_Result(exe_Alu_Result)
    );
```

```
53
54      initial begin
55          // Initialize Inputs
56          Clock = 0;
57          Resetn = 0;
58
59          #50;
60          Resetn = 1;
61
62      end
63      always #50 Clock=~Clock;
64
65  endmodule
```

仿真结果如下（截图）：



仿真结果分析：

如图所示，前三条指令正常运行，第四条指令转移条件成立，pcsource=01，发生跳转，已进入 IF 阶段的下一条指令被废除（变成 0 组成的空指令）。

成功跳转后指令继续运行，之后遇到转移条件不成立的分支指令，则不发生跳转，pc=0x30 时，jump 指令移动到 ID 阶段，pcsource=10，

直接跳转到第一条指令，已进入 IF 阶段的下一条指令也被废除。

时钟信号的设置同实验三，传入 RegFile 的时钟信号与全局的时钟信号相反，即在时钟的下降沿完成写入寄存器，使得后半个周期能读到最新数据。

结论：

整体运行符合预期。

二）跳转用 ID 级检测，分支用 EXE 级检测

1.修改后的模块介绍

1) 顶层模块

完整代码如下：

```
22  module SCCPU(Clock, Resetn, PC, bpc,
23     if_inst_org, if_inst,
24     id_inst_org, id_inst,
25     exe_inst,
26     pcsource, branch,
27     exe_Alu_Result
28      );
29      input Clock, Resetn;
30      output [31:0] PC, if_inst_org, if_inst, id_inst, id_inst_org, exe_inst;
31      output [31:0] bpc, exe_Alu_Result;
32      output branch;
33      output [1:0] pcsource;
34
35      wire [31:0] mem_mo, wb_mo; // 输出pc, 各阶段的inst,Alu_Result, mo
36      wire [31:0] m5, r9;          // 输出rom5和reg9的内容，检验指令是否正确执行
37
38      wire [1:0] pcsource_org;
39      wire [31:0] jpc, if_pc4, id_pc4, id_bpc, exe_bpc;
40
41      wire [31:0] wdi, mem_Alu_Result, wb_Alu_Result;
42      wire [31:0] id_ra, exe_ra, id_rb, exe_rb, mem_rb;
43      wire [31:0] id_imm, exe_imm;
44
45      wire [4:0] id_rn, exe_rn, mem_rn, wb_rn;
46
47      wire id_wreg, exe_wreg, mem_wreg, wb_wreg;
48      wire id_m2reg, exe_m2reg, mem_m2reg, wb_m2reg;
49      wire id_wmem, exe_wmem, mem_wmem, id_aluimm, exe_aluimm;
```

```
50      wire [2:0] id_aluc, exe_aluc;
51      wire id_shift, exe_shift, z;
52
53      IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, if_pc4, if_inst_org, PC);
54
55      assign if_inst = (branch||id_inst[31:26] == 6'b010010) ? 32'h0 : if_inst_org;
56      IF_ID IR (if_pc4, if_inst, Clock, Resetn, id_pc4, id_inst_org);
57
58      ID_STAGE stage2 (id_pc4, id_inst_org, id_rn, wdi, Clock, Resetn, id_bpc, jpc, pcsource_org,
59                id_m2reg, id_wmem, id_aluc, id_aluimm, id_ra, id_rb, id_imm, r9, id_shift, id_wreg,
60                z, wb_wreg, wb_rn);
61
62      assign pcsource = (exe_inst[31:26] == 6'b001111 &&exe_z)
63        || (exe_inst[31:26] == 6'b010000 && ~exe_z) ? 2'b01: (id_inst[31:26] == 6'b010010) ? 2'b10 : 2'b00;
64
65      assign branch = ((exe_inst[31:26]==6'b001111 && exe_z == 1'b1)
66        || (exe_inst[31:26]==6'b010000 && exe_z == 1'b0)) ? 1'b1 : 1'b0;
67
68      assign id_inst = branch ? 32'h0 : id_inst_org;
69
70      assign bpc =  branch ? exe_bpc: id_bpc;
71
72      ID_EXE id_exe_reg (Clock, Resetn,
73                id_m2reg,id_wmem,id_aluc,id_aluimm,id_shift,id_wreg,
74                id_ra,id_rb,id_imm,id_rn, id_inst, id_bpc,
75                exe_m2reg,exe_wmem,exe_aluc,exe_aluimm,exe_shift,exe_wreg,
76                exe_ra,exe_rb,exe_imm,exe_rn, exe_inst, exe_bpc
77                );
```

```
79      // 产生exe_z信号
80      EXE_STAGE stage3 (exe_aluc, exe_aluimm, exe_ra, exe_rb, exe_imm, exe_shift, exe_Alu_Result, exe_z);
81
82      EXE_MEM exe_mem_reg (Clock, Resetn,
83              exe_Alu_Result, exe_rb, exe_wmem, exe_m2reg, exe_wreg, exe_rn,
84              mem_Alu_Result, mem_rb, mem_wmem, mem_m2reg, mem_wreg, mem_rn
85      );
86
87      // Alu_Result[6:2]报错: Cannot index into non-array mem_Alu_Result
88      MEM_STAGE stage4 (mem_wmem, mem_Alu_Result[4:0], mem_rb, Clock, mem_mo, m5);
89
90      MEM_WB mem_wb_reg (Clock, Resetn,
91                mem_Alu_Result,mem_m2reg,mem_wreg,mem_rn,mem_mo,
92                wb_Alu_Result,wb_m2reg,wb_wreg,wb_rn,wb_mo);
93
94      WB_STAGE stage5 (wb_Alu_Result, wb_mo, wb_m2reg, wdi); //wdi写入寄存器堆的数据直连到ID_STAGE即可
95
96  endmodule
```

分支指令在 EXE 阶段处理，跳转指令在 ID 阶段处理。

使用 EXE 阶段产生的 zero 标志，结合指令的 op 码产生 pcsource，之前代码中的 pcsource 不再起作用，为了方便修改，直接将在顶层代码中修改为 pcsource_org，并不再使用。

增加表示分支是否成立的信号 branch，从而判断转移指令之后的两条进入流水线的指令是否需要废除。

原来的 bpc 是在 ID 阶段产生的，现在需要使用流水线寄存器将其传递到 EXE 阶段，如果分支成立的话，就将 exe_bpc 传入 PC 部件。

1) ID_EXE 流水段寄存器模块

完整代码如下：

```
module ID_EXE(clk, clrn,
            id_m2reg,id_wmem,id_aluc,id_aluimm,id_shift,id_wreg,
            id_qa,id_qb,id_imm,id_rn,id_inst,id_bpc,
            exe_m2reg,exe_wmem,exe_aluc,exe_aluimm,exe_shift,exe_wreg,
            exe_qa,exe_qb,exe_imm,exe_rn,exe_inst,exe_bpc
    );

    input clk, clrn;
    input id_m2reg,id_wmem,id_aluimm,id_shift,id_wreg;
    input[2:0] id_aluc;
    input[31:0] id_inst,id_qa,id_qb,id_imm,id_bpc;
    input[4:0] id_rn;

    output exe_m2reg,exe_wmem,exe_aluimm,exe_shift,exe_wreg;
    output[2:0] exe_aluc;
    output[31:0] exe_inst,exe_qa,exe_qb,exe_imm,exe_bpc;
    output[4:0] exe_rn;

    reg exe_m2reg,exe_wmem,exe_aluimm,exe_shift,exe_wreg;
    reg[2:0] exe_aluc;
    reg[31:0] exe_qa,exe_qb,exe_imm,exe_inst,exe_bpc;
    reg[4:0] exe_rn;

    always @ (posedge clk or negedge clrn)
      if(clrn==0)
         begin
            exe_m2reg<=0;
            exe_wmem<=0;
            exe_aluc<=0;
            exe_aluimm<=0;
            exe_shift<=0;
            exe_wreg<=0;
            exe_qa<=0;
            exe_qb<=0;
            exe_imm<=0;
            exe_rn<=0;
            exe_inst<=0;
            exe_bpc<=0;
         end
      else
         begin
            exe_m2reg<=id_m2reg;
            exe_wmem<=id_wmem;
            exe_aluc<=id_aluc;
            exe_aluimm<=id_aluimm;
            exe_shift<=id_shift;
            exe_wreg<=id_wreg;
            exe_qa<=id_qa;
            exe_qb<=id_qb;
            exe_imm<=id_imm;

             exe_rn<=id_rn;
             exe_inst<=id_inst;
             exe_bpc<=id_bpc;
          end
endmodule
```

由于 EXE 阶段需要用到 inst 产生 pcsource 和 branch，并将 bpc 传递到 EXE 阶段，该流水段寄存器需要传入 id_inst 和 id_bpc，写到 exe_inst 和 exe_bpc。

3. 初始化部分

1）寄存器堆初始化（该模块完整代码）：

同实验一

4）数据存储器初始化（该模块完整代码）：

同实验一

5）指令存储器初始化：

指令地址	汇编指令	机器指令	备注
6'h01	addi r1,r1,0x0004	32'h14001021	
6'h02	or r4,r5,r6	32'h042010a6	
6'h03	ori r6,r2,0x00cc	32'h28033046	无冒险 3 条
6'h04	beq r1,r1,6'h03	32'h3c000c21	分支指令，跳转到地址 0x08
6'h05	addi r1,r1,0x0005	32'h14001421	
6'h06	add r1,r1,r1	32'h00100421	
6'h07	and r2,r2,r1	32'h04100841	
6'h08	addi r1,r1,0x0004	32'h14001021	
6'h09	beq r3,r4,6'h06	32'h3c001883	分支指令，不相等则不跳转
6'h0a	add r5,r3,r4	32'h00101464	
6'h0b	jump 0x0000001	32'h48000001	跳转指令
6'h0c	and r2,r2,r1	32'h04100841	

该模块完整代码：

```
module Inst_ROM(a,inst
    );
    input [5:0] a;
    output [31:0] inst;
    wire [31:0] rom [0:63];

    assign rom[6'h00]=32'h00000000;

    assign rom[6'h01]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000005
    assign rom[6'h02]=32'h042010a6;    //or r4,r5,r6  r4=0x00000007
    assign rom[6'h03]=32'h28033046;    //ori r6,r2,0x00cc  r6=0x000000ce
    assign rom[6'h04]=32'h3c000c21;    //beq r1,r1,6'h03  offset=0x0003, 相等则转移到5+3=0x08

    assign rom[6'h05]=32'h14001421;    //addi r1,r1,0x0005  r1=0x0000000a
    assign rom[6'h06]=32'h00100421;    //add r1,r1,r1
    assign rom[6'h07]=32'h04100841;    //and r2,r2,r1

    assign rom[6'h08]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000009
    assign rom[6'h09]=32'h3c001883;    //beq r3,r4,6'h06  offset=6, 不相等则不跳转
    assign rom[6'h0a]=32'h00101464;    //add r5,r3,r4  r5=0x00000007
    assign rom[6'h0b]=32'h48000001;    //jump 0x0000001  无条件转移到01h处

    assign rom[6'h0c]=32'h04100841;    //and r2,r2,r1  r2=0x00000000
    assign inst=rom[a];
endmodule
```

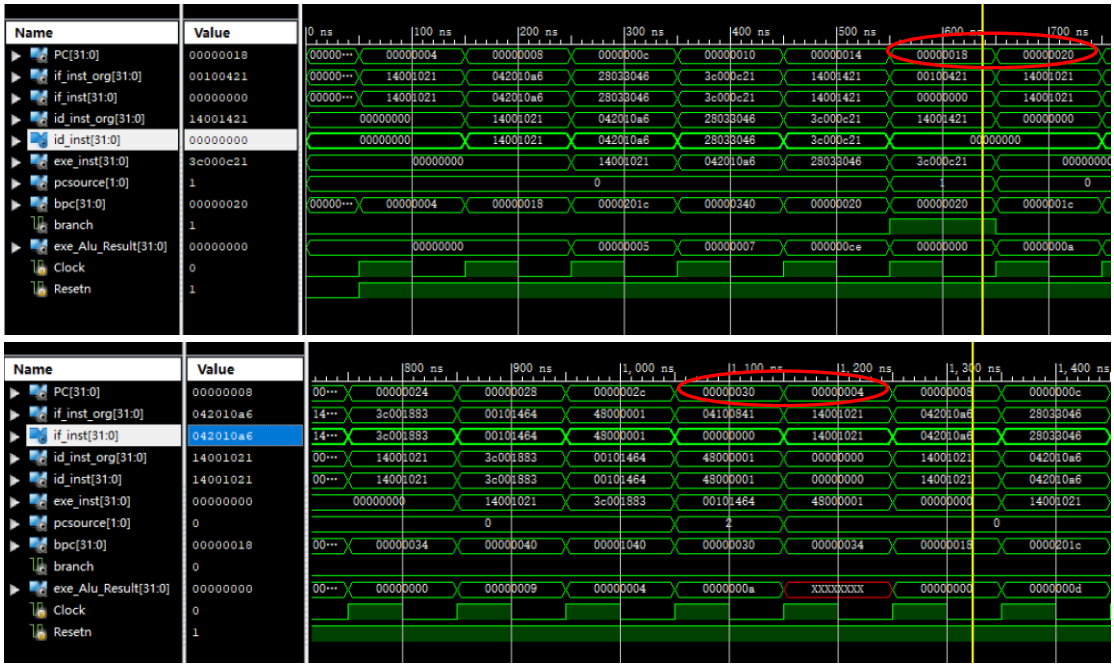
4. 仿真

仿真测试代码如下：

```
module SSCPU_test;

    // Inputs
    reg Clock;
    reg Resetn;

    // Outputs
    wire [31:0] PC;
    wire [31:0] if_inst_org;
    wire [31:0] if_inst;
    wire [31:0] id_inst_org;
    wire [31:0] id_inst;
    wire [31:0] exe_inst;
    wire [1:0] pcsource;
    wire [31:0] bpc;
    wire branch;
    wire [31:0] exe_Alu_Result;

    // Instantiate the Unit Under Test (UUT)
    SCCPU uut (
        .Clock(Clock),
        .Resetn(Resetn),
        .PC(PC),
        .bpc(bpc),
        .if_inst_org(if_inst_org),
        .if_inst(if_inst),
        .id_inst_org(id_inst_org),
        .id_inst(id_inst),
```

```
53          .exe_inst(exe_inst),
54          .pcsource(pcsource),
55          .branch(branch),
56          .exe_Alu_Result(exe_Alu_Result)
57      );
58
59      initial begin
60          // Initialize Inputs
61          Clock = 0;
62          Resetn = 0;
63
64          #50;
65          Resetn = 1;
66
67      end
68      always #50 Clock=~Clock;
69
70  endmodule
```

仿真截图如下：



仿真结果分析：

如图所示，分支指令运行到 EXE 阶段且跳转条件成立时，branch=1，pcsource=01，后面进入流水线的两条指令被废除，运行到跳转指令时，pcsource=10，成功跳转，后面进入流水线的一条指令被废除。

时钟信号的设置同实验三。

结论：

整体运行符合预期。

三）解决数据冒险+控制冒险

1.修改后的模块介绍

1) 顶层模块

完整代码如下：

```
22  module SCCPU(Clock, Resetn, PC, pcsource,
23     if_inst_org, if_inst, id_inst,
24     id_ra, id_rb,
25     exe_Alu_Result,
26     stall, FwdA, FwdB
27      );
28      input Clock, Resetn;
29      output [31:0] PC, if_inst, id_inst, exe_Alu_Result;
30
31      // 输出与冒险相关的信号，检查冒险处理是否正确
32      output stall;
33      output [1:0] FwdA, FwdB;
34      output [1:0] pcsource;
35      output [31:0] if_inst_org;
36      output [31:0] id_ra, id_rb;
37
38      wire [31:0] bpc, jpc, if_pc4, id_pc4;
39
40      wire [31:0] wdi;
41      wire [31:0] exe_ra, exe_rb, mem_rb;
42      wire [31:0] id_imm, exe_imm;
43      wire [31:0] mem_Alu_Result, wb_Alu_Result, mem_mo, wb_mo;
44      wire [4:0] id_rn, exe_rn, mem_rn, wb_rn;
45
46      wire id_wreg, exe_wreg, mem_wreg, wb_wreg;
47      wire id_m2reg, exe_m2reg, mem_m2reg, wb_m2reg;
48      wire id_wmem, exe_wmem, mem_wmem, id_aluimm, exe_aluimm;
49      wire [2:0] id_aluc, exe_aluc;
50      wire id_shift, exe_shift, z;
51      wire id_wmem_org, id_wreg_org;
52
53      // 增加与冒险相关的wire变量
54      wire [4:0] rs, rt;
55      wire regrt;
56      wire [31:0] id_ra_org, id_rb_org;
57      wire [31:0] m5, r9;          // 输出rom5和reg9的内容，检验指令是否正确执行
58
59      // 将stall信号传入PC和IR
60      IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, stall, if_pc4, if_inst_org, PC);
61
62      // 废除分支或转移指令之后的指令，换成32'h0
63      assign if_inst = (pcsource ==2'b01 || pcsource ==2'b10) ? 32'h0 : if_inst_org;
64
65      IF_ID IR (if_pc4, if_inst, Clock, Resetn, stall, id_pc4, id_inst);
66
67      assign id_z = (id_ra == id_rb)?1'b1:1'b0;  // 将操作数的比较提前到ID 阶段
68      // 或者 id_z = ~|( id_a ^ id_b );          // 使用四选一MUX选出的ra和rb，解决数据冒险
69
```

```
    // 从ID_stage引出rs, rt, regrt
    // 将id_z传入ID_STAGE的Control Unit中，产生新的pcsource
    ID_STAGE stage2 (id_pc4, id_inst, id_rn, wdi, Clock, Resetn, bpc, jpc, pcsource,
              id_m2reg, id_wmem_org, id_aluc, id_aluimm, id_ra_org, id_rb_org, id_imm, r9,
              id_shift, id_wreg_org,
              id_z, wb_wreg, wb_rn, rs, rt, regrt);

    // 检测load-use冒险，输出暂停信号
    assign stall= ((rs == exe_rn) | (rt == exe_rn)&~regrt)&(exe_rn!=5'b0)&(exe_wreg&exe_m2reg);
    assign id_wmem = id_wmem_org & ~stall;
    assign id_wreg = id_wreg_org & ~stall;

    // 生成信号FwdA和FwdB，并控制MUX4_1
    // 不需要检测单独排除shift和I型指令，因为后面还有一个多路选择器
    assign FwdA = ((exe_rn != 5'b0) & exe_wreg & (exe_rn == rs) &  ~exe_m2reg) ? 2'b01 : // 选 E_Alu
                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  mem_m2reg) ? 2'b11 :
                  2'b00;  // 2'b11 选M_mo(load)，2b'00 直接选regfile输出

    assign FwdB = ((exe_rn != 5'b0) & exe_wreg & (exe_rn == rt) &  ~exe_m2reg) ? 2'b01 : // 选 E_Alu
                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rt) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rt) &  mem_m2reg) ? 2'b11 :
                  2'b00;  // 2'b11 选M_mo(load)，2b'00 直接选regfile输出

    mux32_4_1 id_ina(id_ra_org,exe_Alu_Result,mem_Alu_Result,mem_mo,FwdA,id_ra);
    mux32_4_1 id_inb(id_rb_org,exe_Alu_Result,mem_Alu_Result,mem_mo,FwdB,id_rb);

    ID_EXE id_exe_reg (Clock, Resetn,
              id_m2reg,id_wmem,id_aluc,id_aluimm,id_shift,id_wreg,
              id_ra,id_rb,id_imm,id_rn,
              exe_m2reg,exe_wmem,exe_aluc,exe_aluimm,exe_shift,exe_wreg,
              exe_ra,exe_rb,exe_imm,exe_rn
              );

    //z信号直接连到pc模块，不使用nop的情况下不会正确转移
    EXE_STAGE stage3 (exe_aluc, exe_aluimm, exe_ra, exe_rb, exe_imm, exe_shift, exe_Alu_Result, z);

    EXE_MEM exe_mem_reg (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
    );

    // Alu_Result[6:2]报错：Cannot index into non-array mem_Alu_Result
    // 将Alu_Result[6:2]改为直接传入完整的Alu_Result信号
    MEM_STAGE stage4 (mem_wmem, mem_Alu_Result[4:0], mem_rb, Clock, mem_mo, m5);

    MEM_WB mem_wb_reg (Clock, Resetn,
              mem_Alu_Result,mem_m2reg,mem_wreg,mem_rn,mem_mo,
              wb_Alu_Result,wb_m2reg,wb_wreg,wb_rn,wb_mo);

    WB_STAGE stage5 (wb_Alu_Result, wb_mo, wb_m2reg, wdi);

endmodule
```

废除分支或跳转指令后一条指令的操作与第一种方案一致。

用于产生零标志的两个操作数使用四选一多路选择器选出的数据，即处理数据冒险之后的数据 id_ra 和 id_rb，而不是 id_ra_org 和 id_rb_org。

2) 涉及到修改的子模块

完整代码如下：

其他模块的代码同使用停顿+前推的方案解决数据冒险的代码一致

5. 初始化部分

1）寄存器堆初始化（该模块完整代码）：

同实验一

6）数据存储器初始化（该模块完整代码）：

同实验一

7）指令存储器初始化：

指令地址	汇编指令	机器指令	备注
6'h01	addi r1,r1,0x0004	32'h14001021	
6'h02	or r4,r5,r6	32'h042010a6	
6'h03	addi r3,r2,0x0005	32'h14001443	无冒险 3 条
6'h04	beq r3,r4,6'h06	32'h3c001883	有数据冒险的分支指令
6'h05	addi r1,r1,0x0005	32'h14001421	
6'h0b	addi r1,r1,0x0004	32'h14001021	
6'h0c	ori r6,r2,0x00cc	32'h28033046	
6'h0d	add r5,r3,r4	32'h00101464	
6'h0e	jump 0x0000001	32'h48000001	跳转指令
6'h0f	and r2,r2,r1	32'h04100841	

该模块完整代码：

```
module Inst_ROM(a,inst
    );
    input [5:0] a;
    output [31:0] inst;
    wire [31:0] rom [0:63];

    assign rom[6'h00]=32'h00000000;    //0地址为空，从1地址开始执行；

    assign rom[6'h01]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000005
    assign rom[6'h02]=32'h042010a6;    //or r4,r5,r6  r4=0x00000007
    assign rom[6'h03]=32'h14001443;    //addi r3,r2,0x0005  r3=0x00000007
    assign rom[6'h04]=32'h3c001883;    //beq r3,r4,6'h0b  offset=6,跳转到6'h0b
    assign rom[6'h05]=32'h14001421;    //addi r1,r1,0x0005  r1=0x0000000a

    assign rom[6'h0b]=32'h14001021;    //addi r1,r1,0x0004  r1=0x00000009
    assign rom[6'h0c]=32'h28033046;    //ori r6,r2,0x00cc  r6=0x000000ce
    assign rom[6'h0d]=32'h00101464;    //add r5,r3,r4  r5=0x00000007
    assign rom[6'h0e]=32'h48000001;    //jump 0x0000001  无条件转移到01h处

    assign rom[6'h0f]=32'h04100841;    //and r2,r2,r1  r2=0x00000000

    assign inst=rom[a];
endmodule
```

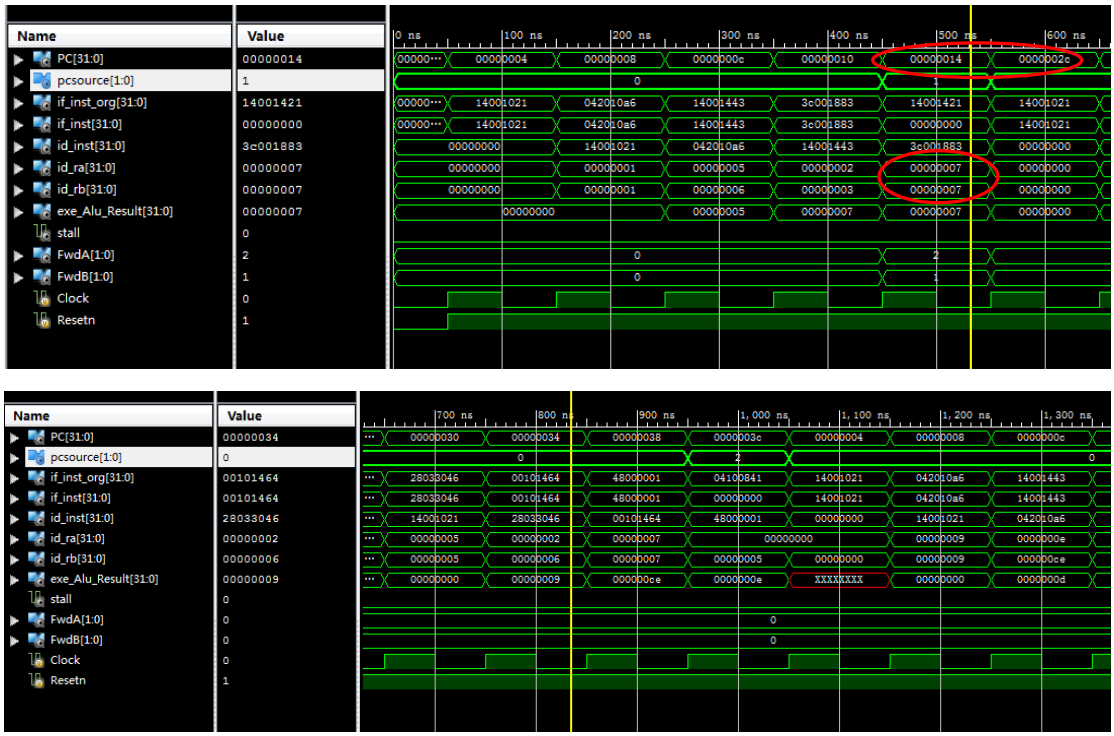
6. 仿真

仿真测试代码如下：

```
25  module SSCPU_test;
26
27      // Inputs
28      reg Clock;
29      reg Resetn;
30
31      // Outputs
32      wire [31:0] PC;
33      wire [1:0] pcsource;
34      wire [31:0] if_inst_org;
35      wire [31:0] if_inst;
36      wire [31:0] id_inst;
37      wire [31:0] id_ra;
38      wire [31:0] id_rb;
39      wire [31:0] exe_Alu_Result;
40      wire stall;
41      wire [1:0] FwdA;
42      wire [1:0] FwdB;
```

```
44      // Instantiate the Unit Under Test (UUT)
45      SCCPU uut (
46          .Clock(Clock),
47          .Resetn(Resetn),
48          .PC(PC),
49          .pcsource(pcsource),
50          .if_inst_org(if_inst_org),
51          .if_inst(if_inst),
52          .id_inst(id_inst),
53          .id_ra(id_ra),
54          .id_rb(id_rb),
55          .exe_Alu_Result(exe_Alu_Result),
56          .stall(stall),
57          .FwdA(FwdA),
58          .FwdB(FwdB)
59      );
60
61      initial begin
62          // Initialize Inputs
63          Clock = 0;
64          Resetn = 0;
65
66          #50;
67          Resetn = 1;
68
69      end
70      always #50 Clock=~Clock;
71
72  endmodule
```

仿真结果如下（截图）：



仿真结果分析：

如图所示，数据冒险被正确处理，分支指令 beq r3,r4,6'h06 在 ID 阶段比较的数分别是 EXE 和 MEM 阶段前推得到的数据，因此两数相等，发生跳转，其余结果的分析同以上两种方案。

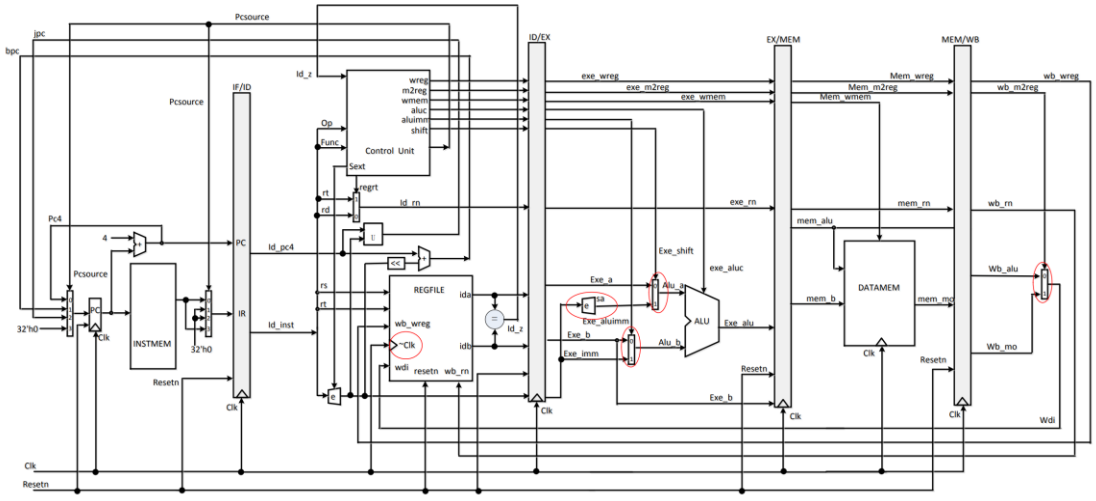
时钟信号的设置同实验三。

结论：

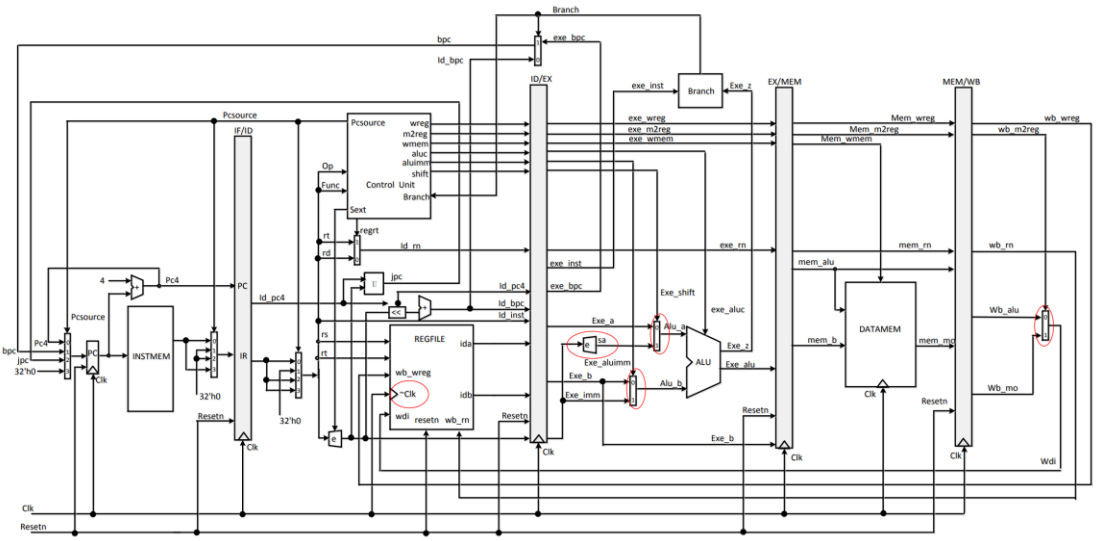
整体运行符合预期。

7. 解决控制冒险的五级流水线 CPU 架构如下：

解决控制冒险的五级流水线 CPU 架构图（ID）：



解决控制冒险的五级流水线 CPU 架构图（EXE）（选择第二种方案，直接传递 bpc）：



十、实验结论：（联系理论知识进行说明）

修改后的流水线 CPU 能处理控制冒险，正确完成转移，废除无意义的指令。

实验中需要注意以下两点：

1）EXE 级检测的方案中，branch=1 时，不是必须需修改 id_wreg 和 id_wmem，因为此时分支指令之后的两条指令已经被修改成 32’h0 了，之后产生的写使能信号本来就会是 0。

2）编写选择语句的时候需要注意书写顺序导致的优先级，比如下面表达式“Assign pcsource = branch ? 2'b01: id_inst[31:26] == 6'b010010? 2'b10: 2'b00;”，branch 会比 id_inst 更先被判断。当分支指令后面跟着跳转指令时，分支指令执行到 EXE 级时产生 branch 信号，跳转指令运行到 ID 级，分支和跳转之间就会产生竞争，此时要优先执行前面的分支指令，所以书写选择语句时要先判断 branch，不能更改顺序。

十一、总结及心得体会：

控制冒险通常会带来整条指令的废除，代价很大，我们需要选择适当的处理方式，尽可能减少控制冒险带来的损失。在 ID 阶段处理控制冒险的方法需要增加一个比较器，但能提前一个周期判断是否转移，也就能少产生一条需要被废除的指令，对流水线的效率有很大的提高。如果结合预测是否转移选中的方法，还能进一步提高效率。

十二、对本实验过程及方法、手段的改进建议：

对于 EXE 级检测控制冒险的方案，PPT 中好像只考虑了分支成立时要废除当前 IF 阶段的指令，没有考虑跳转时也要废除 IF 阶段的指令，所以代码要改成 assign if_inst = (branch||id_inst[31:26] == 6'b010010) ? 32'h0 : if_inst_org; 才能正确运行。

扩展：执行级检测（只有分支的情况）

```
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;
IF_Inst = branch ? 32'h0 : IF_Inst_org;
ID_Inst = branch ? 32'h0 : ID_Inst_org;
```

报告评分：

指导教师签字：