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：5.12 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：解决数据冒险问题

三、实验学时：4

四、实验原理：

一）数据冒险的概念：一条指令必须等到前面的指令把结果写回才能执行，从而导致流水线暂停，称为数据冒险。

二）如何检测出数据冒险？

假设有三条连续的指令 L1/L2/L3（L1 最先执行），L1 指令写目的寄存器 rd，L2 和 L3 的源操作数假设分别在寄存器 rs1 或 rs2 中，L2、L3 的 rs1 或 rs2 与 L1 的目的寄存器号 rd 相等，就可能发生数据冒险。

三）解决数据冒险的方法：

1. 暂停流水线

1）封锁当前译码后和写回数据操作相关的控制信号；

假设 stall 是暂停信号，考虑到暂停，需要对在 ID 级产生的控制信号进行封锁，因此，涉及到的信号有：

- id_wreg：id 级传递到 exe 级的写寄存器堆的使能信号；
- id_wreg_org：id 级译码后直接产生的写寄存器使能信号；
- id_wmem：id 级传递到 exe 级的写数据存储器的使能信号；

- id_wmem_org：id 级译码后直接产生的写数据存储器使能信号；

由于暂停信号的影响，得到上述信号之间的关系（stall 为高电平有效）：

- id_wreg = ~stall & id_wreg_org
- id_wmem = ~stall & id_wmem_org

2）IR 中不接收新的指令；

需要暂停流水线的时候，IR 不能接收新的指令。因此，stall 信号也要对 IR 模块进行控制，只有当 stall=0（无效）时，才正常接收新的指令，否则，IR 值不变。

操作方式：在 IR 模块中增加 stall 信号的影响

3）PC 不接收新指令地址。

需要暂停流水线的时候，PC 不能接收新的指令地址。因此，stall 信号也要对 PC 模块进行控制，只有当 stall=0（无效）时，才正常接收新的指令地址，否则，PC 值不变。

操作方式：在 PC 模块中增加 stall 信号的影响。

4）如何产生 stall 信号？

相邻两条指令产生数据冒险时，结合 EXE 级控制信号进行判断是否有冒险，从而得出暂停信号的生成表达式为：

stall1 = ((Rs == E_Rn) | (Rt == E_Rn)&~regrt)&(E_Rn !=0)&E_Wreg

间隔一条指令的两条指令之间产生数据冒险时，结合 MEM 级控制信号进行判断是否有冒险，从而得出暂停信号的生成表达式为：

stall2 = ((Rs == M_Rn) | (Rt ==M_Rn)&~regrt)&(M_Rn !=0)&M_Wreg

综合上述两种情况，暂停信号 stall 的表达式为：

stall = stall1 | stall2

【说明】在 I 型指令中，rt 是目标操作数，所以对 rt 的判断应该针对非 I 型指令，即只有当 regrt=0 时才判断 rt，因为：

regrt=i_addi|i_andi|i_ori|i_xori|i_lw;

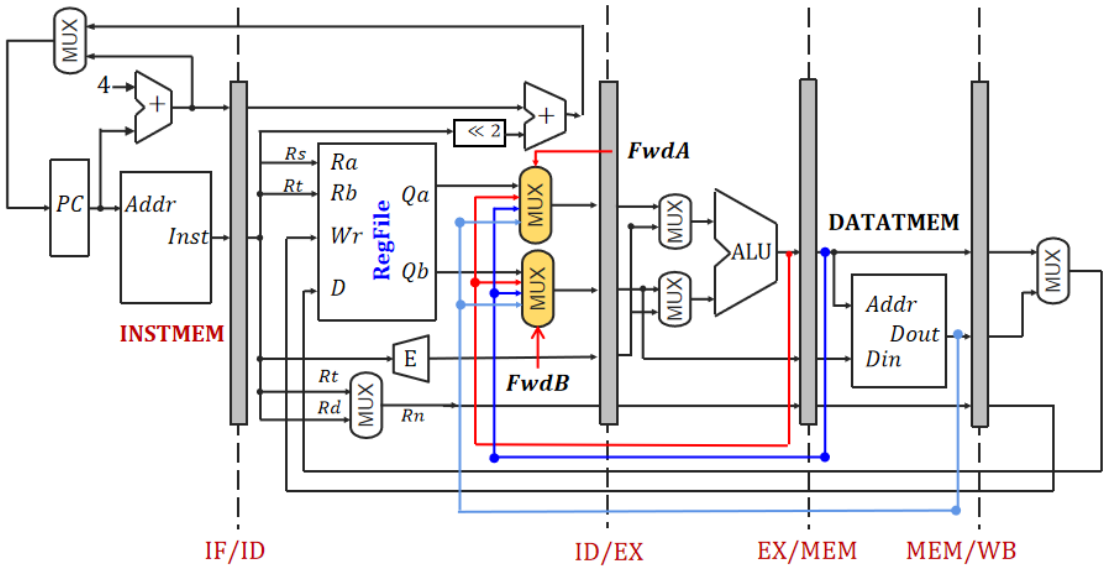
上面的判断式中出现的信号说明分别是：

- E_Wreg：表示 EXE 级寄存器堆写信号；
- M_Wreg：表示 MEM 级寄存器堆写信号；
- E_Rn：表示 EXE 级要写的寄存器号；
- M_Rn：表示 MEM 级要写的寄存器号；
- E_M2reg：表示 EXE 级写入数据来源信号；
- M_M2reg：表示 MEM 级写入数据来源信号；（=1 取 Mo，=0 取 ALU_result）
- Rs/Rt：表示当前译码指令的 2 个源操作数；

2. 数据前推（含暂停）

数据前推的思想就是：在 EXE 级需要的数据，通过增加传输通道的方式，从 MEM 级或 WB 级中获取，也就是把数据提前送到 EXE 级，不必等到前面的指令执行完毕。

一共存在三种情况的数据前推，如下图所示。图中增加的数据传输通道通过四路选择器进行选择输出，作为 ALU 的操作数来源，其中一路黑色线条表示寄存器直接输出数据（即正常传输数据，没有产生冒险）。



1）红色线条代表的数据前推是相邻两条指令产生数据冒险的情况；

2）相隔一条指令产生数据冒险的情况又细分为两种情况，分别是：

（1）M_M2reg=1，需要取 MO 的值，该数据前推用浅蓝色线条表示；

（2）M_M2reg=0，需要取 ALU_result 的值，用深蓝色线条表示。

从图中可见，数据前推的设计方案中，除了增加三路数据传输通道，还需要增加两个四选一多路选择器，假设分别命名为 FwdA 和 FwdB，其中，FwdA 的选项不涉及移位类指令且只涉及对 rs 产生的数据冒险进行检测，FwdB 的选项不涉及 I 型指令，所以均不需要对 regrt 进行检测。

经过分析，FwdA 的表达式如下：

FwdA = ((E_Rn != 5'b0) & E_Wreg & (E_Rn == rs) & ~E_m2reg) ? 2'b01 : // 选 E_Alu

((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & ~M_m2reg) ? 2'b10 : // 选 M_Alu

((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & M_m2reg) ?

2'b11 : // 2'b11 选 M_mo(load)

2'b00; // 2b'00 直接选 regfile 输出

FwdB 的表达式如下：

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

还有一种特殊情况需要处理：如果相邻两条指令产生数据冒险，而第一条指令是 lw 指令，按照规则，lw 指令需要运行到 MEM 级才能产生浅蓝色线条的信号前推，不能直接用前述的解决方案进行处理。

解决方案是：该情况需要紧随其后的指令在 ID 级暂停一次。假设暂停信号为 stall，根据分析，可以得到 stall 的判定表达式为：

stall= ((Rs == E_Rn) | (Rt == E_Rn)&~regrt) & (E_Rn !=0) &（E_Wreg & E_M2reg）

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数据冒险的概念和解决方法。

六、实验内容：

1. 掌握数据冒险问题的成因，充分理解采用暂停流水线与内部前推两种方式解决数据冒险问题的原理；

2. 至少完成一种方式解决数据冒险问题，自行设计指令序列，并给出仿真结果，对结果进行详尽分析说明

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） 用暂停的方式解决数据冒险

1) 修改后的 PC 模块

完整代码如下：

```
21  module dff32(d,clk,clrn,stall,q
22      );
23      input [31:0] d;
24      input clk,clrn;
25      input stall;
26
27      output [31:0] q;
28      reg [31:0] q;
29      always @ (negedge clrn or posedge clk)
30      if(clrn==0)
31          begin
32              q<=0;
33          end
34      else if(~stall)
35          begin
36              q<=d;
37          end
38  endmodule
```

2) 修改后的 IR 模块

完整代码如下：

```
module IF_ID( if_pc4, if_inst, clk, clrn, stall, id_pc4, id_inst
    );

    input[31:0] if_pc4, if_inst;
    input clk, clrn;
    input stall;
    output[31:0] id_pc4, id_inst;

    // if_pc4, if_inst是引线传入的信号
    // 传入寄存器id_pc4, id_inst中保存
    reg[31:0] id_pc4, id_inst;

    // 时钟上升沿，clrn=0，则寄存器信号清零
    // 否则，将if阶段的信号存入寄存器中，作为id阶段的输入
    always @ (posedge clk or negedge clrn)
      if(clrn==0)
         begin
            id_pc4<=0;
            id_inst<=0;
         end
      else if(~stall)
         begin
            id_pc4<=if_pc4;
            id_inst<=if_inst;
         end

endmodule
```

3）修改后的 ID 模块

完整代码如下：

```
module ID_STAGE(pc4,inst,id_rn,
                wdi,clk,clrn,bpc,jpc,pcsource,
                m2reg,wmem,aluc,aluimm,a,b,imm,r9,
                shift,wreg,
                rsrtequ,
                wb_wreg,wb_rn, //WB级回传
                rs, rt, regrt   // 判断load-use冒险
    );

    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 rsrtequ;      //branch控制信号
    input wb_wreg;      // WB阶段回传的wreg信号
    input [4:0] wb_rn;

    output [31:0] bpc,jpc,a,b,imm,r9;     //bpc-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 [4:0] id_rn;    //写回寄存器号
    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] rs, rt;
    output regrt;

    wire [5:0] op,func;
    wire [4:0] rd;
    wire [31:0] qa,qb,br_offset;

    wire [15:0] ext16;
    wire 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(rsrtequ,func,                          //控制部件
                op,wreg,m2reg,wmem,aluc,regrt,aluimm,
                sext,pcsource,shift);

    Regfile rf (rs,rt,wdi,wb_rn,wb_wreg,clk,clrn,qa,qb,r9);//寄存器堆，有32个32位的寄存器，0号寄存器恒为0
    mux5_2_1 des_reg_num (rd,rt,regrt,id_rn); //选择目的寄存器是来自于rd,还是rt

    assign a=qa;
    assign b=qb;

    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};      //jump指令的目标地址的计算

endmodule
```

4）修改后的 RegFile 模块

将时钟信号取反，改成在时钟下降沿写入，避免结构冒险

```
module Regfile(rna,rnb,d,wn,we,clk,clrn,qa,qb,r6
    );
    input [4:0] rna,rnb,wn;
    input [31:0] d;
    input we,clk,clrn;
    output [31:0] qa,qb,r6;
    reg [31:0] register [1:31];
    assign qa=(rna==0)?0:register[rna];
    assign qb=(rnb==0)?0:register[rnb];
    integer i;
    always @(posedge ~clk or negedge clrn) // 改成在时钟下降沿写入，避免结构冒险
        if(clrn==0)      //如果复位信号有效，则进行寄存器初始化。
        begin//:init

        for(i=1;i<32;i=i+1)       //所有寄存器清零
            register[i]<=0;

         register[5'h01]<=32'h00000001; //对指定寄存器的指定地址初始化值
         register[5'h02]<=32'h00000002;
         register[5'h03]<=32'h00000003;
         register[5'h04]<=32'h00000004;
         register[5'h05]<=32'h00000005;
         register[5'h06]<=32'h00000006;
         register[5'h07]<=32'h00000007;
         register[5'h08]<=32'h00000008;
         register[5'h09]<=32'h00000009;

        end
        else if((wn!=0)&&we)
           register[wn]<=d;
    assign r6=register[5'h06];
endmodule
```

5) 修改后的顶层模块

完整代码如下：

```
module SCCPU(Clock, Resetn, PC,
   if_inst, id_inst,
   exe_Alu_Result, mem_Alu_Result, wb_Alu_Result,
   mem_mo, wb_mo,
   m5, r9, stall
    );
    input Clock, Resetn;
    output [31:0] PC, if_inst, id_inst, exe_Alu_Result, mem_Alu_Result, wb_Alu_Result;
    output [31:0] mem_mo, wb_mo; // 输出pc, 各阶段的inst,Alu_Result, mo
    output [31:0] m5, r9;        // 输出rom5和reg9的内容，检验指令是否正确执行
    output stall;

    wire [1:0] pcsource;
    wire [31:0] bpc, jpc, if_pc4, id_pc4;

    wire [31:0] wdi;
    wire [31:0] id_ra, exe_ra, id_rb, exe_rb, mem_rb;
    wire [31:0] id_imm, exe_imm;

    wire [4:0] id_rn, exe_rn, mem_rn, wb_rn;

    wire id_wreg, exe_wreg, mem_wreg, wb_wreg;
    wire id_m2reg, exe_m2reg, mem_m2reg, wb_m2reg;
    wire id_wmem, exe_wmem, mem_wmem, id_aluimm, exe_aluimm;
    wire [2:0] id_aluc, exe_aluc;
    wire id_shift, exe_shift, z;
```

```
    // 为冒险检测引出或增加的信号
    wire [4:0] rs, rt;
    wire regrt;
    wire id_wmem_org, id_wreg_org;
    wire stall1, stall2;

    IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, stall, if_pc4, if_inst, PC);

    IF_ID IR (if_pc4, if_inst, Clock, Resetn, stall, id_pc4, id_inst);

    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, id_rb, id_imm, r9, id_shift, id_wreg_org,
              z, wb_wreg, wb_rn, rs, rt, regrt);

    // 检测冒险，输出暂停信号
    assign stall1 = ((rs == exe_rn) | (rt == exe_rn)&~regrt)&(exe_rn !=0)&exe_wreg;
    assign stall2 = ((rs == mem_rn) | (rt == mem_rn)&~regrt)&(mem_rn !=0)&mem_wreg;
    assign stall = stall1 | stall2 ;

    assign id_wmem = id_wmem_org & ~stall;
    assign id_wreg = id_wreg_org & ~stall;

    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, 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); //wdi写入寄存器堆的数据直连到ID_STAGE即可

endmodule
```

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

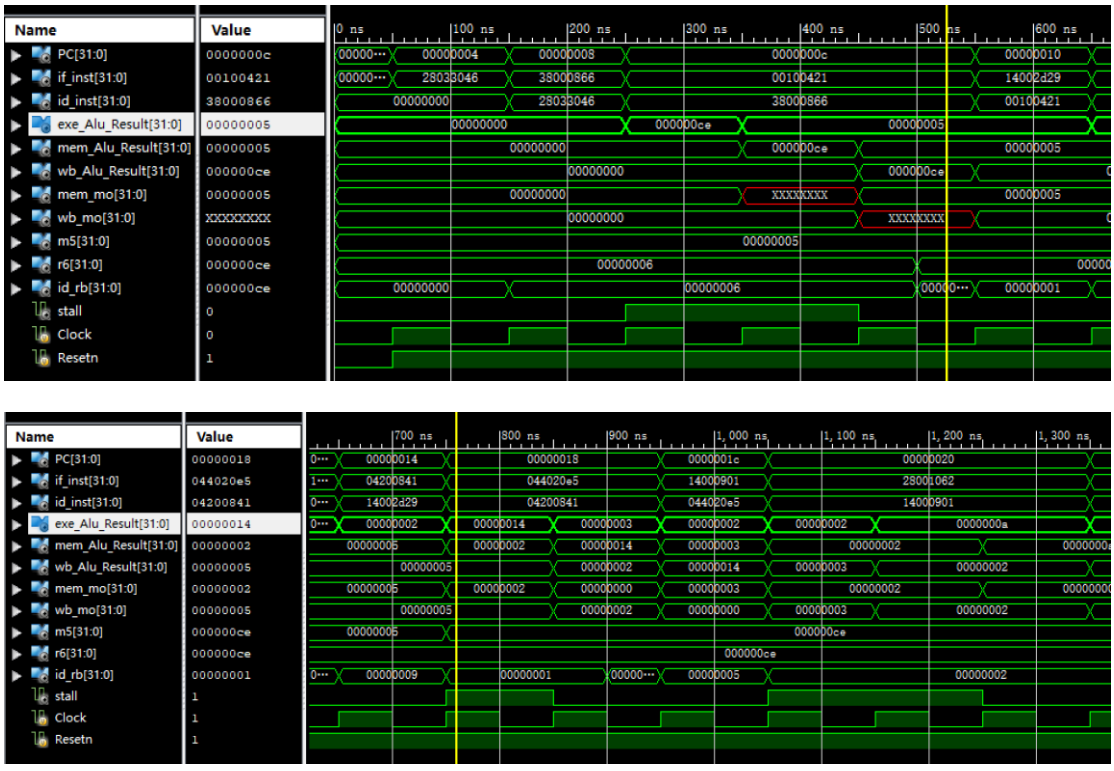
3）指令存储器初始化：

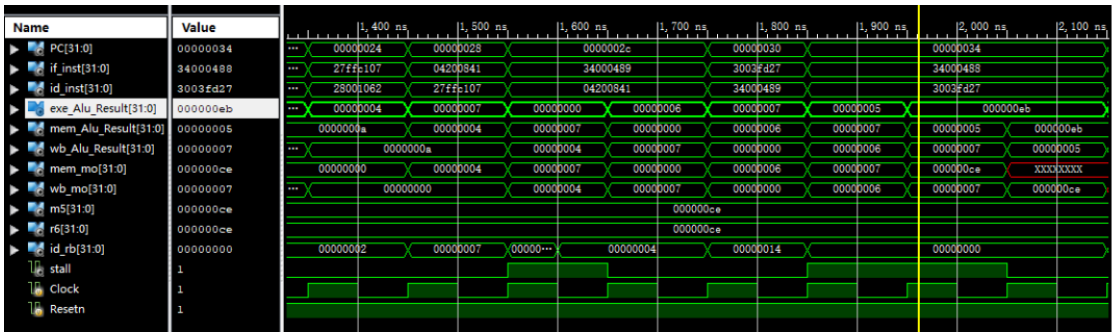
该模块完整代码（含注释，要标注助记符指令及其相关的冒险情况）：

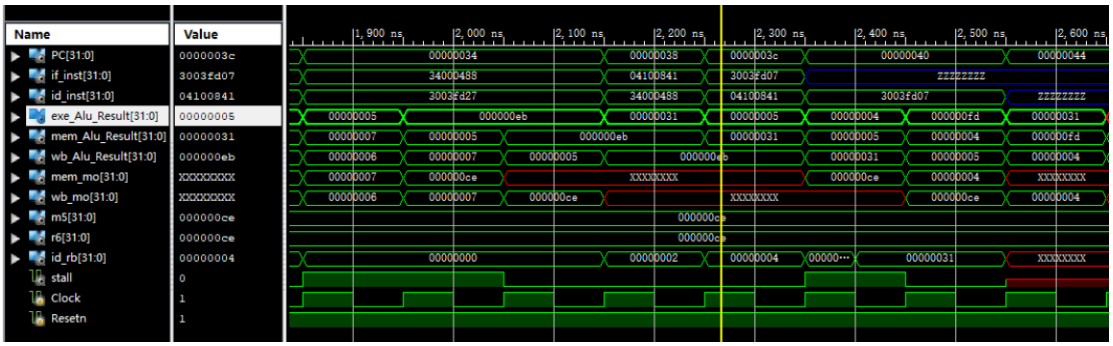
```
// 相邻指令关于rt的数据冒险，如果不解决冒险，m5=0x00000006
assign rom[6'h01]=32'h28033046;    //ori r6,r2,0x00cc  r6=0x000000ce
assign rom[6'h02]=32'h38000866;    //store r6,0x0002(r3)  m5=0x000000ce

// 间隔一条指令关于rt的数据冒险，如果不解决冒险，r2=0x00000003
assign rom[6'h03]=32'h00100421;    //add r1,r1,r1  r1=0x00000002
assign rom[6'h04]=32'h14002d29;    //addi r9,r9,0x000b  r9=0x00000014
assign rom[6'h05]=32'h04200841;    //or r2,r2,r1  r2=0x00000002

// 相邻指令关于rs的数据冒险，如果不解决冒险，r5=8+2=10=0x0000000a
assign rom[6'h06]=32'h044020e5;    //xor r8,r7,r5  r8=0x00000002
assign rom[6'h07]=32'h14000901;    //addi r1,r8,0x02  r1=0x00000004

// 间隔一条指令关于rs的数据冒险，如果不解决冒险，r8=2|4=0x00000006
assign rom[6'h08]=32'h28001062;    //ori r2,r3,0x04  r2=0x00000007
assign rom[6'h09]=32'h27ffc107;    //andi r7,r8,0xfff0  r7=0x00000000
assign rom[6'h0a]=32'h04200841;    //or r8,r2,r1  r8=0x00000007

// load-use 冒险，如果不能解决冒险，r7=0x000000eb
assign rom[6'h0b]=32'h34000489;    //load r9,0x0001(r4)  r9=m5=0x000000ce
assign rom[6'h0c]=32'h3003fd27;    //xori r7,r9,0x00ff  r7=0x00000031

// 相隔一条指令的load冒险，不用暂停直接前推，如果不能解决冒险，r7=0x000000eb
assign rom[6'h0d]=32'h34000488;    //load r8,0x0001(r4)  r8=m5=0x000000ce
assign rom[6'h0e]=32'h04100841;    //and r2,r2,r1  r2=0x00000004
assign rom[6'h0f]=32'h3003fd07;    //xori r7,r8,0x00ff  r7=0x00000031
```

3. 仿真

仿真测试代码如下：







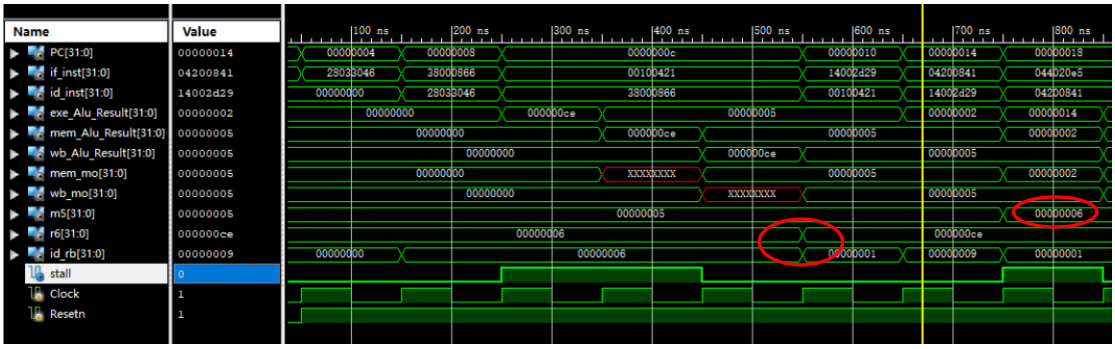
【分析】

从仿真结果中可以看到，相邻指令发生冒险时，stall 信号变为 1，控制流水线停顿两个周期，间隔一条的指令的两条指令发生冒险时，stall 信号变为 1，控制流水线停顿一个周期。

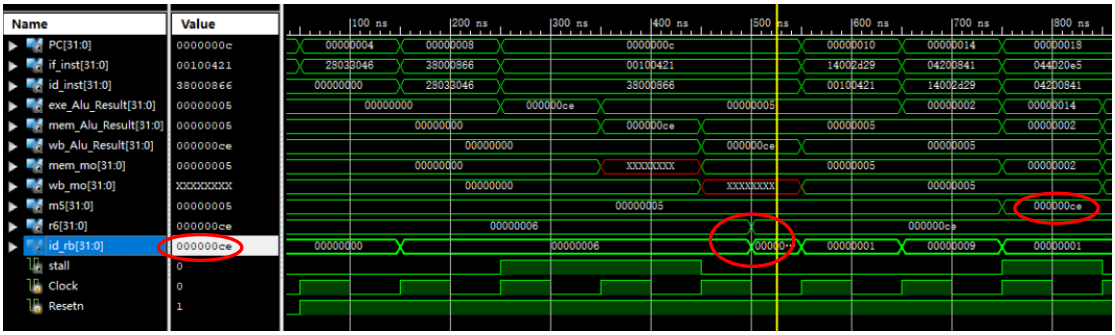
以前两条指令 ori r6,r2,0x00cc（32'h28033046）和 store r6,0x0002(r3)（32'h38000866）为例，第三个时钟周期，第二条指令位于 ID 阶段，电路检测到 rt 与 exe_rn 存在冒险，stall=1，停顿一个周期，此时第一条指令还在 EXE 阶段，未写入 r6，所以 r6 为初始值 6，第四个周期，第一条指令位于 MEM 阶段，电路检测到 rt 与 mem_rn 存在冒险，再停顿一个周期，第五个周期，exe_wreg 和 mem_wreg 均为 0，停顿解除，第一条指令的结果在时钟下降沿写入 r6，由于读寄存器由组合逻辑控制，第二条指令读取到的值也迅速更新，得到正确的值，第七个周期结束时更新后的 r6 内容被写入 mem5，从图中可见，mem5=0x000000ce，冒险被成功解决。后续指令的处理同理，各

阶段的 Alu_Result 信号与预测结果一致（参见指令部分的注释），所有冒险均被解决。

时钟信号会影响时序部件的运行结果。控制写寄存器堆的时钟信号与总时钟信号一致，即在时钟上升沿写入的仿真测试如下图：



控制写寄存器堆的时钟信号与总时钟信号相反，即在时钟下降沿写入的仿真测试如下图：



如图所示，当控制写寄存器堆的时钟信号与总时钟信号相同时，寄存器堆的写入和流水段寄存器的更新都在时钟上升沿（也就是某个周期结束时）完成，因此流水段寄存器被更新的是写入前的旧值，存在冒险。应当使控制写寄存器堆的时钟信号与总时钟信号相反，这样就能提前半个周期，在时钟下降沿写入寄存器，然后由组合逻辑控制的读出内容（id_ra 和 id_rb）也立即更新，在上升沿写入流水段寄存器的就是更新后的信号。

【结论】

控制写寄存器堆的时钟信号应该与总时钟信号相反，即在时钟下降沿写入，这样才能解决结构冒险，再使用停顿方法解决数据冒险后，整体运行符合预期。

二）用暂停+前推的方式解决数据冒险

1. 修改后的模块介绍

1）修改后的 PC 模块

完整代码同上一种方法。

2） 修改后的 IR 模块

完整代码同上一种方法。

3） 修改后的 ID 模块

完整代码同上一种方法。

4) 修改后的顶层模块

完整代码如下：

```
22  module SCCPU(Clock, Resetn, PC,
23     if_inst, id_inst,
24     exe_Alu_Result, mem_Alu_Result, wb_Alu_Result,
25     mem_mo, wb_mo,
26     m5, r9,
27     stall, FwdA, FwdB
28      );
29      input Clock, Resetn;
30      output [31:0] PC, if_inst, id_inst, exe_Alu_Result, mem_Alu_Result, wb_Alu_Result;
31      output [31:0] mem_mo, wb_mo; // 输出pc, 各阶段的inst,Alu_Result, mo
32      output [31:0] m5, r9;        // 输出rom5和reg9的内容，检验指令是否正确执行
33      // 输出与冒险相关的信号，检查冒险处理是否正确
34      output stall;
35      output [1:0] FwdA, FwdB;
36
37      wire [1:0] pcsource;
38      wire [31:0] bpc, jpc, if_pc4, id_pc4;
39
40      wire [31:0] wdi;
41      wire [31:0] id_ra, exe_ra, id_rb, exe_rb, mem_rb;
42      wire [31:0] id_imm, exe_imm;
43
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
58    // 将stall信号传入PC和IR
59    IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, stall, if_pc4, if_inst, PC);
60
61    IF_ID IR (if_pc4, if_inst, Clock, Resetn, stall, id_pc4, id_inst);
62
63    // 从ID_stage引出rs, rt, regrt
64    ID_STAGE stage2 (id_pc4, id_inst, id_rn, wdi, Clock, Resetn, bpc, jpc, pcsource,
65              id_m2reg, id_wmem_org, id_aluc, id_aluimm, id_ra_org, id_rb_org, id_imm, r9,
66              id_shift, id_wreg_org,
67              z, wb_wreg, wb_rn, rs, rt, regrt);
68
69    // 检测load-use冒险，输出暂停信号
70    assign stall= ((rs == exe_rn) | (rt == exe_rn)&~regrt)&(exe_rn!=5'b0)&(exe_wreg&exe_m2reg);
71    assign id_wmem = id_wmem_org & ~stall;
72    assign id_wreg = id_wreg_org & ~stall;
73
74    // 生成信号FwdA和FwdB，并控制MUX4_1
75    // 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
76    assign FwdA = ((exe_rn != 5'b0) & exe_wreg & (exe_rn == rs) &  ~exe_m2reg) ? 2'b01 : // 选 E_Alu
77                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
78                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
79                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rs) &  mem_m2reg) ? 2'b11 :
80                  2'b00;  // 2'b11 选M_mo(load)，2b'00 直接选regfile输出
81
82    assign FwdB = ((exe_rn != 5'b0) & exe_wreg & (exe_rn == rt) &  ~exe_m2reg) ? 2'b01 : // 选 E_Alu
83                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rt) &  ~mem_m2reg) ? 2'b10 : // 选 M_Alu
84                  ((mem_rn != 5'b0) & mem_wreg & (mem_rn == rt) &  mem_m2reg) ? 2'b11 :
85                  2'b00;  // 2'b11 选M_mo(load)，2b'00 直接选regfile输出
86
87    mux32_4_1 id_ina(id_ra_org,exe_Alu_Result,mem_Alu_Result,mem_mo,FwdA,id_ra);
88    mux32_4_1 id_inb(id_rb_org,exe_Alu_Result,mem_Alu_Result,mem_mo,FwdB,id_rb);
89
90    ID_EXE id_exe_reg (Clock, Resetn,
91              id_m2reg,id_wmem,id_aluc,id_aluimm,id_shift,id_wreg,
92              id_ra,id_rb,id_imm,id_rn,
93              exe_m2reg,exe_wmem,exe_aluc,exe_aluimm,exe_shift,exe_wreg,
94              exe_ra,exe_rb,exe_imm,exe_rn
95              );
96
97    //z信号直接连到pc模块，不使用nop的情况下不会正确转移
98    EXE_STAGE stage3 (exe_aluc, exe_aluimm, exe_ra, exe_rb, exe_imm, exe_shift, exe_Alu_Result, z);
```

```
99
100     EXE_MEM exe_mem_reg (Clock, Resetn,
101             exe_Alu_Result, exe_rb, exe_wmem, exe_m2reg, exe_wreg, exe_rn,
102             mem_Alu_Result, mem_rb, mem_wmem, mem_m2reg, mem_wreg, mem_rn
103     );
104
105     // Alu_Result[6:2]报错：Cannot index into non-array mem_Alu_Result
106     // 将Alu_Result[6:2]改为直接传入完整的Alu_Result信号
107     MEM_STAGE stage4 (mem_wmem, mem_Alu_Result, mem_rb, Clock, mem_mo, m5);
108
109     MEM_WB mem_wb_reg (Clock, Resetn,
110                mem_Alu_Result,mem_m2reg,mem_wreg,mem_rn,mem_mo,
111                wb_Alu_Result,wb_m2reg,wb_wreg,wb_rn,wb_mo);
112
113     WB_STAGE stage5 (wb_Alu_Result, wb_mo, wb_m2reg, wdi);
114
115 endmodule
```

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

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：

该模块完整代码（含注释，要标注助记符指令及其相关的冒险情况）：

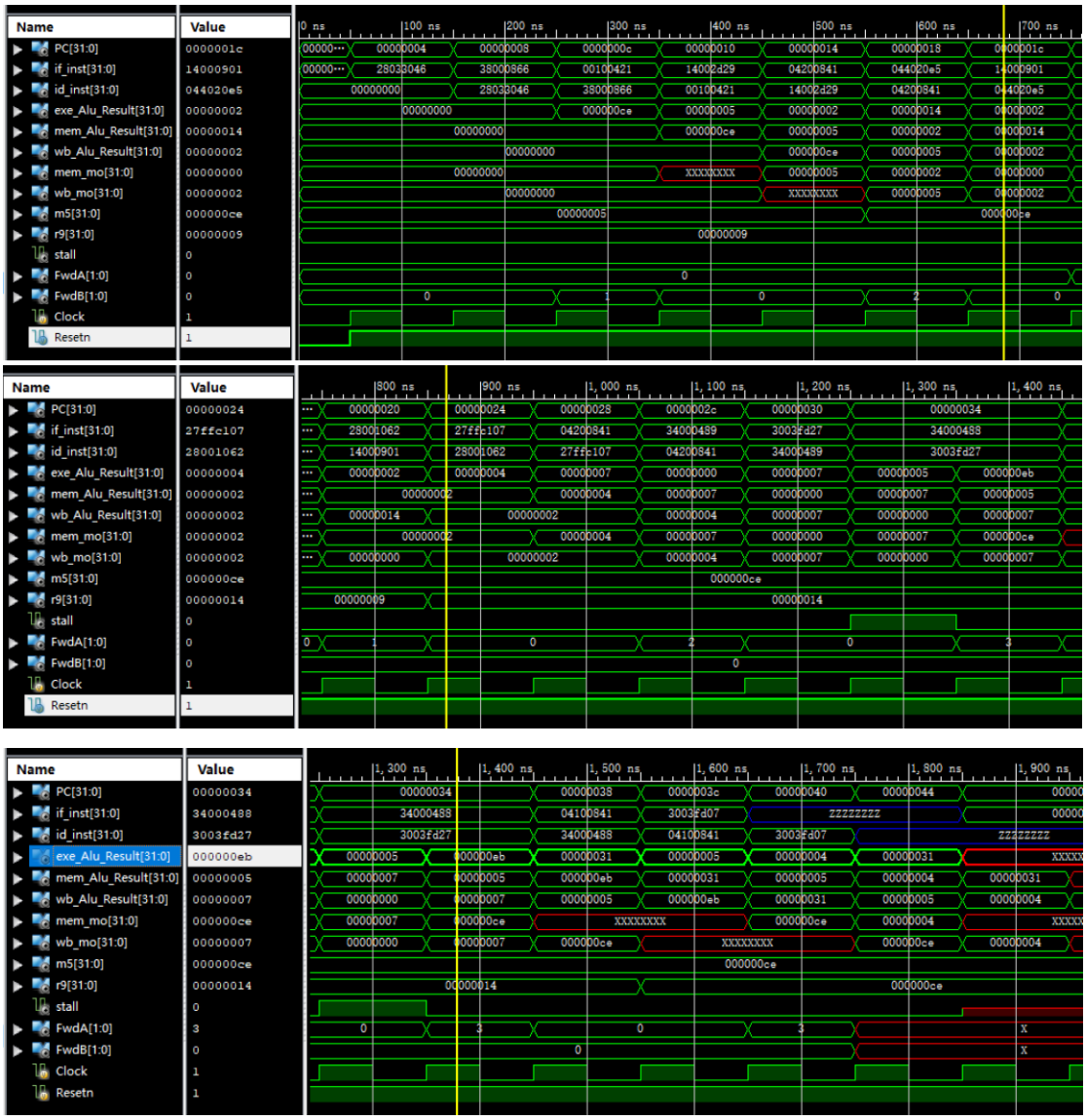
```
assign rom[6'h00]=32'h00000000;     //0地址为空，从1地址开始执行；

// 相邻指令关于rt的数据冒险，如果不解决冒险，m5=0x00000006
assign rom[6'h01]=32'h28033046;     //ori r6,r2,0x00cc   r6=0x000000ce
assign rom[6'h02]=32'h38000866;     //store r6,0x0002(r3)   m5=0x000000ce

// 间隔一条指令关于rt的数据冒险，如果不解决冒险，r2=0x00000003
assign rom[6'h03]=32'h00100421;     //add r1,r1,r1   r1=0x00000002
assign rom[6'h04]=32'h14002d29;     //addi r9,r9,0x000b   r9=0x00000014
assign rom[6'h05]=32'h04200841;     //or r2,r2,r1   r2=0x00000002

// 相邻指令关于rs的数据冒险，如果不解决冒险，r5=8+2=10=0x0000000a
assign rom[6'h06]=32'h044020e5;     //xor r8,r7,r5   r8=0x00000002
assign rom[6'h07]=32'h14000901;     //addi r1,r8,0x02   r1=0x00000004

// 间隔一条指令关于rs的数据冒险，如果不解决冒险，r8=2|4=0x00000006
assign rom[6'h08]=32'h04200823;     //or r2,r1,r3   r2=0x00000007
assign rom[6'h09]=32'h27ffc107;     //andi r7,r8,0xfff0   r7=0x00000000
assign rom[6'h0a]=32'h04200841;     //or r8,r2,r1   r8=0x00000007

// load-use 冒险，如果不能解决冒险，r7=0x000000eb
assign rom[6'h0b]=32'h34000489;     //load r9,0x0001(r4)   r9=m5=0x000000ce
assign rom[6'h0c]=32'h3003fd27;     //xori r7,r9,0x00ff   r7=0x00000031

// 相隔一条指令的load冒险，不用暂停直接前推，如果不能解决冒险，r7=0x000000eb
assign rom[6'h0d]=32'h34000488;     //load r8,0x0001(r4)   r8=m5=0x000000ce
assign rom[6'h0e]=32'h04100841;     //and r2,r2,r1   r2=0x00000004
assign rom[6'h0f]=32'h3003fd07;     //xori r7,r8,0x00ff   r7=0x00000031
```

3. 仿真

仿真测试代码如下：

```
module SCCPU_test;

    // Inputs
    reg Clock;
    reg Resetn;

    // Outputs
    wire [31:0] PC;
    wire [31:0] if_inst;
    wire [31:0] id_inst;
    wire [31:0] exe_Alu_Result;
    wire [31:0] mem_Alu_Result;
    wire [31:0] wb_Alu_Result;
    wire [31:0] mem_mo;
    wire [31:0] wb_mo;
    wire [31:0] m5, r9;
    wire stall;
    wire [1:0] FwdA, FwdB;

    // Instantiate the Unit Under Test (UUT)
    SCCPU uut (
        .Clock(Clock),
        .Resetn(Resetn),
        .PC(PC),
        .if_inst(if_inst),
        .id_inst(id_inst),
        .exe_Alu_Result(exe_Alu_Result),
        .mem_Alu_Result(mem_Alu_Result),
        .wb_Alu_Result(wb_Alu_Result),
        .mem_mo(mem_mo),
        .wb_mo(wb_mo),
        .m5(m5),
        .r9(r9),
        .stall(stall),
        .FwdA(FwdA),
        .FwdB(FwdB)
    );

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

        // Wait 100 ns for global reset to finish
        #50;
        Resetn = 1;
        // Add stimulus here

    end
    always #50 Clock=~Clock;

endmodule
```

仿真结果如下（截图）：



【分析】

第一组冒险指令：

相邻指令关于 rt 的数据冒险，如果不解决冒险，m5=0x00000006

rom[6'h01]=32'h28033046;　　//ori r6,r2,0x00cc　r6=0x000000ce

rom[6'h02]=32'h38000866;　　//store r6,0x0002(r3)　m5=0x000000ce

Rs 不存在冒险，FwdA=2’b00

Rt 存在冒险， ((E_Rn != 5'b0) & E_Wreg & (E_Rn == rt) & ~E_m2reg)=1 ，FwdB= 2'b01 ，选 E_Alu

当第一条指令运行到 EXE 的时候，第二条指令需要数据 r6，因此，可以将第一条指令在 EXE 级的结果进行前推，由图可知，FwdB= 2'b01，m5 被更新为 0xce，仿真结果正确。

第二组冒险指令：

间隔一条指令关于 rt 的数据冒险，如果不解决冒险，r2=0x00000003

rom[6'h03]=32'h00100421; //add r1,r1,r1 r1=0x00000002

rom[6'h04]=32'h14002d29; //addi r9,r9,0x000b r9=0x00000014

rom[6'h05]=32'h04200841; //or r2,r2,r1 r2=0x00000002

Rs 不存在冒险，FwdA=2’b00

Rt 存在冒险， ((M_Rn != 5'b0) & M_Wreg & (M_Rn == rt) & ~M_m2reg) =1，FwdB = 2'b10 : // 选 M_Alu

当第三条指令运行到 MEM 的时候，第五条指令需要数据 r1，因此，可以将第三条指令在 MEM 级的结果进行前推，由图可知，FwdB= 2'b10，第三条指令产生的 Alu_Result=0x02，仿真结果正确。

第三组冒险指令：

相邻指令关于 rs 的数据冒险，如果不解决冒险，

r5=8+2=10=0x0000000a

rom[6'h06]=32'h044020e5;　　//xor r8,r7,r5　r8=0x00000002

rom[6'h07]=32'h14000901;　　//addi r1,r8,0x02　r1=0x00000004

Rt 不存在冒险，FwdB=2’b00

Rs 存在冒险，((E_Rn != 5'b0) & E_Wreg & (E_Rn == rs) & ~E_m2reg) =1，FwdA = 2'b01，选 E_Alu

当第六条指令运行到 EXE 的时候，第七条指令需要数据 r8，因此，可以将第六条指令在 EXE 级的结果进行前推，由图可知，FwdA= 2'b01，第七条指令产生的 Alu_Result=0x04，仿真结果正确。

第四组冒险指令：

间隔一条指令关于 rs 的数据冒险，如果不解决冒险，r8=2|4=0x00000006

rom[6'h08]=32'h28001062;　　//ori r2,r3,0x04　r2=0x00000007

rom[6'h09]=32'h27ffc107;　　//andi r7,r8,0xfff0　r7=0x00000000

rom[6'h0a]=32'h04200841;　　//or r8,r2,r1　r8=0x00000007

Rt 不存在冒险，FwdB=2’b00

Rs 存在冒险，((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & ~M_m2reg) =1，FwdA = 2'b10 : // 选 M_Alu

当第十条指令运行到 MEM 的时候，第八条指令需要数据 r1，因此，可以将第十条指令在 MEM 级的结果进行前推，由图可知，FwdB= 2'b10，第十条指令产生的 Alu_Result=0x07，仿真结果正确。

第五组冒险指令：

load-use 冒险，如果不能解决冒险，r7=0x000000eb

rom[6'h0b]=32'h34000489; //load r9,0x0001(r4) r9=m5=0x000000ce

rom[6'h0c]=32'h3003fd27; //xori r7,r9,0x00ff r7=0x00000031

第十一条指令在 ID 阶段时，第十条指令在 EXE 阶段，还未从内存中读取 m5，因此必须将第十一条指令停顿一个周期，再前推第十条指令在 MEM 阶段读出的内容。

((rs==exe_rn)|(rt==exe_rn)&~regrt)&(exe_rn!=5'b0)&(exe_wreg&exe_m2reg)=1，stall=1，停顿一个周期后，((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & M_m2reg) =1， FwdA =2'b11 ，选 Mo(load)，由图可知，FwdA= 2'b11，第十二条指令产生的 Alu_Result=0x31，仿真结果正确。

第六组冒险指令：

相隔一条指令的 load 冒险，不用暂停直接前推，如果不能解决冒险，r7=0x000000eb

rom[6'h0d]=32'h34000488; //load r8,0x0001(r4) r8=m5=0x000000ce

rom[6'h0e]=32'h04100841; //and r2,r2,r1 r2=0x00000004

rom[6'h0f]=32'h3003fd07; //xori r7,r8,0x00ff r7=0x00000031

Rt 不存在冒险，FwdB=2’b00

Rs 存在冒险， ((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & ~M_m2reg) =1，FwdA = 2'b10 : // 选 M_Alu

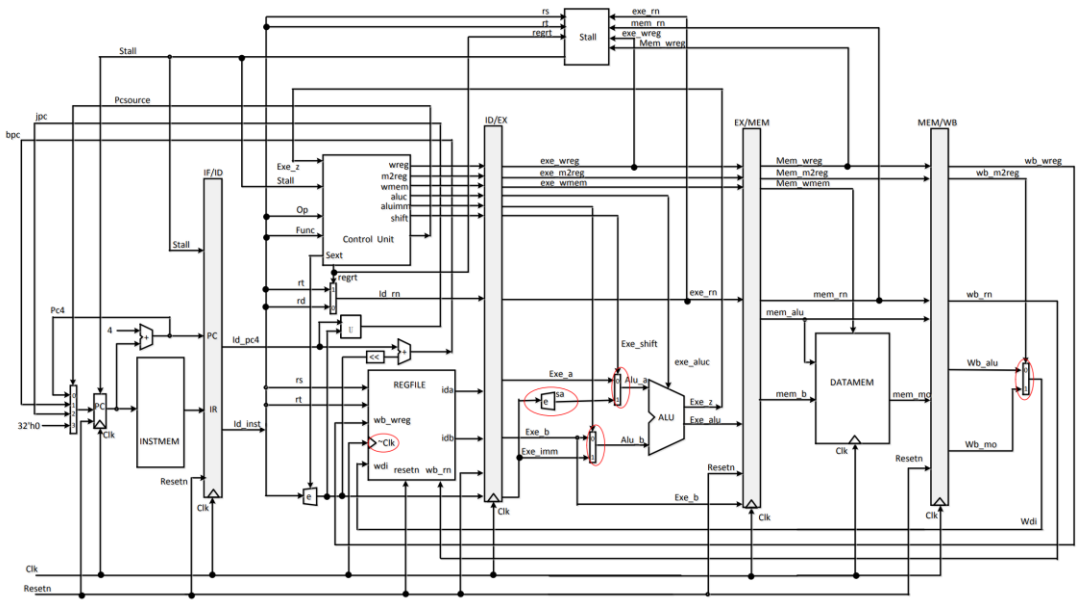
当第十三条指令运行到 MEM 的时候，第十五条指令需要数据 r8，因此，可以将第十三条指令在 MEM 级的结果进行前推，因此这种情况不需要暂停，((M_Rn != 5'b0) & M_Wreg & (M_Rn == rs) & M_m2reg) =1， FwdA =2'b11 ，选 Mo(load)，由图可知，FwdA= 2'b11，第十五条指令产生的 Alu_Result=0x31，仿真结果正确。

【结论】

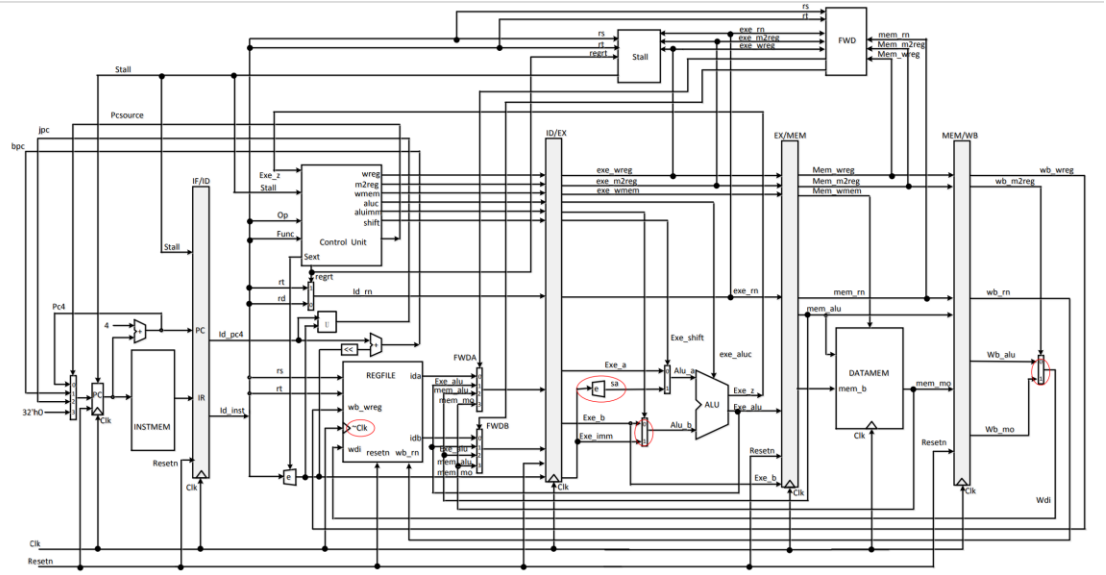
使用停顿+前推的方法解决数据冒险，时钟设计方案同上，得到的整体运行结果符合预期

4. 解决数据冒险的五级流水线 CPU 架构如下：

纯暂停：



前推+暂停：



修改说明：除了之前的两处修改外，为了处理结构冒险，要将控制 RegFile 部件的时钟信号改成与总信号相反，从而在上半周期完成写入。

十、实验结论：（联系理论知识进行说明）

修改后的电路能正确处理流水线冒险，仿真结果符合预期。

实验中需要注意以下几点：

1. 分清组合逻辑和时序逻辑器件，指令存储器、RegFile 和 MEM 的读操作由组合逻辑控制，PC、流水段寄存器的写操作由时序逻辑控制，受时钟信号的影响。
2. 参考代码顶层模块中的 Alu_Result[6:2] 需要修改成 Alu_Result[4:0]，根据指令存储器的设计，一条指令为 32 位，计算 pc 时下一条指令的 pc=pc+4，说明是按字节编址，根据指令存储器的设计，MEM 中的一个地址存放 32 位的内容，即四个字节，所以传入 Alu_Result 的低四位作为取指令的地址即可。

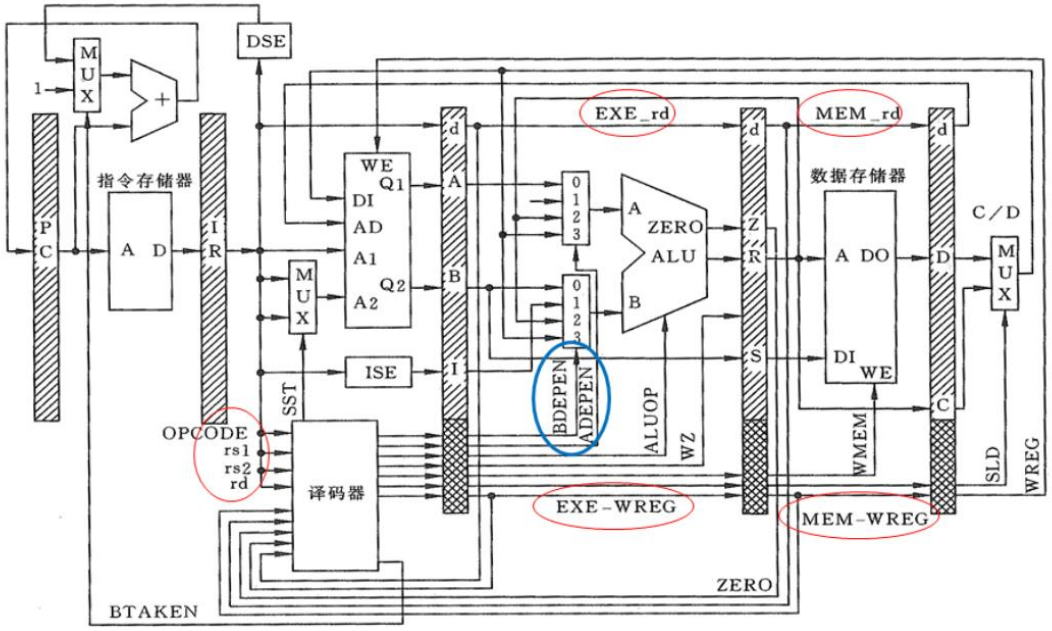
   mem 的一个单元占 4 个字节，所以 load 和 store 指令的地址*4 。
3. 暂停一个周期后，load 指令位于 MEM 阶段，前推的是从内存中读出的数据，FwdA/B=2’b11。
4. 检测冒险是，注意考虑 EXE 和 MEM 阶段的 wreg 和 wmem 信号，写使能为 1 时才存在冒险，这样设计有两方面考虑，一方面，store 指令是不回写寄存器的，所以不会和后面的指令产生冒险，需要排除，另一方面，stall=1 时，ID 阶段产生的 wreg 和 wmem 为 0，并向后面的阶段传递，这样就能解除停顿（否则对于 add r1, r1, r2，这种 rd=rt/rs 的指令，停顿无法解除）。
5. 本实验使用的架构中，移位位数和立即数是用 EXE 阶段单独的多路选择器来选出的，所以设计 FwdA 和 FwdB 时，不用考虑 shift 指令和 I 型指令。
6. 注意信号的宽度，定义变量的时候不要漏写宽度，否则就会被默认是一位的信号。

十一、总结及心得体会：

本次实验我们在上节课写出的流水线 CPU 的基础上实现了处理流水线冒险的电路，对检测冒险和前推数据的原理有了更深的理解。

十二、对本实验过程及方法、手段的改进建议：

体系结构课程上提出的流水线架构可能比实验中选用的架构更简洁、高效，如下图，但这种架构没有考虑 shift 指令，代码的修改也会比实验中的方案更复杂。



该架构是将数据直接前推到 EXE 阶段，扩充原来的二选一多路选择器，从而选出适当的数据输入 ALU，这样可以节省一个二选一的多路选择器（选择移位位数的多路选择器不能节省）。针对间隔一个周期的两条指令发生冒险的情况，该架构直接前推 WB 阶段多路选择器选出的数据，而实验中是分别前推 ALU 的输出和从 Mem 中读

取的数据，没有充分利用多路选择器，造成了一些线路的冗余。

报告评分：

指导教师签字：