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：2023.5.11**

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;

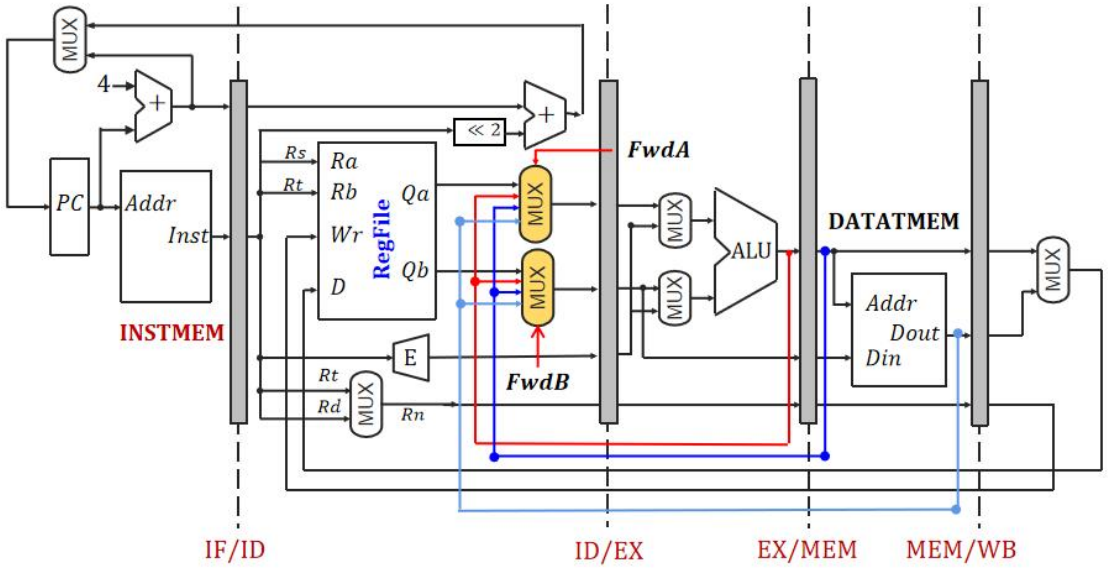
上面的判断式中出现的信号说明分别是：

- 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

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：

用纯暂停的方式解决数据冒险

1.修改后的模块介绍

1) 修改后的顶层模块

完整代码如下：

```
module SCCPU(Clock, Resetn, PC, if_Inst,
exe_Alu_Result,mem_mo,stall,exe_wreg,mem_wreg,exe_rn,
mem_rn,exe_m2reg,mem_m2reg,rs,rt,regrt
    );
    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 stall; //暂停信号输出
    output exe_wreg,mem_wreg;    //写寄存器堆信号
    output [4:0] exe_rn, mem_rn;   //要写的寄存器
    output exe_m2reg,mem_m2reg;   //写寄存器堆数据来源
    output [4:0] rs,rt;   //源寄存器
    output regrt;   //是否用 rt

    wire [31:0] mem_Alu_Result, wb_Alu_Result;//ALU 结果
    wire [1:0] pcsource;   //PC 选择器
    wire [4:0] rs,rt;   //源寄存器传递
    wire regrt;   //是否用 rt
    wire stall;   //暂停信号

    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, 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 id_wreg_org,id_wmem_org; //译码阶段的写寄存器堆和写 DM 的信号

    //取指阶段加入 stall
    IF_STAGE stage1 (Clock, Resetn, pcsource, bpc, jpc, if_pc4, if_Inst, PC,stall);

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

    //译码阶段增加 rs,rt,regrt 引脚输出
    ID_STAGE stage2 (id_pc4, id_Inst, wdi, Clock, Resetn, bpc, jpc, pcsource,
```

```
                    id_m2reg, id_wmem_org, id_aluc, id_aluimm, id_ra, id_rb, id_imm,
                    id_shift, z,id_wreg_org,id_rn,rs, rt, regrt);

    //加入 stall 后的 wreg 和 wmem 信号(译码后的信号更新)
    assign id_wreg = ~stall & id_wreg_org;
    assign id_wmem = ~stall & id_wmem_org;

     //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);

    /stall 信号产生
     STALL STALL_UNIT(exe_wreg,exe_rn,rs,rt,regrt,mem_wreg,mem_rn,stall);

endmodule
```

2) 修改后的 IF_STAGE 模块（如 IR 或其他）
完整代码如下：

```
module IF_STAGE(clk,clrn,pcsource,bpc,jpc,pc4,inst, PC,stall
```

```
    );
     input clk, clrn;
     input stall;
     input [31:0] bpc,jpc;
     input [1:0] pcsource;

     output [31:0] pc4,inst;
     output [31:0] PC;

     wire [31:0] pc;
     wire [31:0] npc;      //下一条指令地址

     dff32 program_counter(npc,clk,clrn,pc,stall);     //利用 32 位的 D 触发器实现 PC，PC 加入 stall
     add32 pc_plus4(pc,32'h4,pc4);//32 位加法器，用来计算 PC+4
     mux32_4_1 next_pc(pc4,bpc,jpc,32'b0,pcsource,npc);//根据 pcsource 信号选择下一条指令的地址
     Inst_ROM inst_mem(pc[7:2],inst); //指令存储器

     assign PC=pc;

endmodule
```

3) 修改后的 PC 模块

完整代码如下：

```
module dff32(d,clk,clrn,q,stall
    );
     input [31:0] d;
     input clk,clrn;
     input stall;
     output [31:0] q;
    reg [31:0] q;
    always @ (negedge clrn or posedge clk)
     if(clrn==0)          //复位信号为 0 时，PC 值为 0
          begin
               q<=0;
           end
   else if(stall==1) //stall 信号为 1 时，PC 值保持不变
          begin
               q<=q;
           end
    else                    //否则，PC 值加 4
         begin
```

```
                q<=d;
            end
endmodule
```

4) 修改后的 IF_ID 模块

完整代码如下：

```
module IF_IDreg(clk,clrn,if_pc4,if_inst,id_pc4,id_inst,stall
    );
    input [31:0] if_pc4,if_inst;
    input clk,clrn;
    input stall;
    output [31:0] id_pc4,id_inst;

    reg [31:0] id_pc4,id_inst;

    always @ (posedge clk or negedge clrn)
        if(clrn ==0 )
            begin
                id_pc4<=0;
                id_inst<=0;
            end
        else if(stall==1)   //stall 信号为 1 时，IF/ID 寄存器的值保持不变
            begin
                id_pc4<=id_pc4;
                id_inst<=id_inst;
            end
        else
            begin
                id_pc4<=if_pc4;
                id_inst <=if_inst;
            end
endmodule
```

5）新增 STALL 处理单元（用于产生 STALL）

完整代码如下：

```
module STALL(
    E_Wreg, E_Rn,
    Rs, Rt, regrt,
    M_Wreg, M_Rn, stall
    );

    input E_Wreg, M_Wreg, regrt;   //输入写信号以及 rn 选择信号
    input [4:0] E_Rn, M_Rn, Rs, Rt;   //前面指令的目的寄存器以及后面指令的
```

```
源寄存器
    output stall;   //输出暂停信号

    wire stall1, stall2;

    assign stall1 = ((Rs == E_Rn) | (Rt == E_Rn) & ~regrt) & (E_Rn != 0) & E_Wreg; //ID 级与 EXE 级的数据冒险，需要两个个 stall
    assign stall2 = ((Rs == M_Rn) | (Rt == M_Rn) & ~regrt) & (M_Rn != 0) & M_Wreg;//ID 级与 MEM 级的数据冒险，需要一个 stall
    assign stall = stall1 | stall2;   //输出暂停信号



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）：

```
    assign rom[6'h00]=32'h00000000;     //0 地址为空，从 1 地址开始执行；
     assign rom[6'h01]=32'h28033046;          //ori r6,r2,0x00cc
     assign rom[6'h02]=32'h38000866;          //store r6,0x0002(r3)
     //store 中 r6 与 ori 语句中的 r6 形成数据冒险，需停顿两个周期

     assign rom[6'h03]=32'h14002d29;            //addi r9,r9,0x000b
     assign rom[6'h04]=32'h044020e5;            //xor r8,r7,r5
     assign rom[6'h05]=32'h34000489;          //load                    r9,0x0001(r4)
r9=0x000000ce
     assign rom[6'h06]=32'h0821a408;            //sr1 r9,r8,5'h03
     //srl 中 r8 于 xor 语句中的 r8 形成数据冒险，需停顿一个周期

     assign rom[6'h07]=32'h044020e5;      //xor r8,r7,r5
     assign rom[6'h08]=32'h00100421;            //add r1,r1,r1
     assign rom[6'h09]=32'h3003fd27;            //xori r7,r9,0x00ff
     //xori 中 r9 于 sr1 语句中的 r9 形成数据冒险，对 regfile 时钟周期取反，实现前半周期写，后半周期读

     assign rom[6'h0A]=32'h00101464;          //add r5,r3,r4    r5=0x00000007
```

3. 仿真

仿真测试代码如下：

```
module CPU_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 stall;
	wire exe_wreg;
	wire mem_wreg;
	wire [4:0] exe_rn;
	wire [4:0] mem_rn;
	wire exe_m2reg;
	wire mem_m2reg;
	wire [4:0] rs;
	wire [4:0] rt;
	wire regrt;

	// 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),
		.stall(stall),
		.exe_wreg(exe_wreg),
		.mem_wreg(mem_wreg),
		.exe_rn(exe_rn),
		.mem_rn(mem_rn),
		.exe_m2reg(exe_m2reg),
		.mem_m2reg(mem_m2reg),
		.rs(rs),
		.rt(rt),
		.regrt(regrt)
	);

	initial begin
```

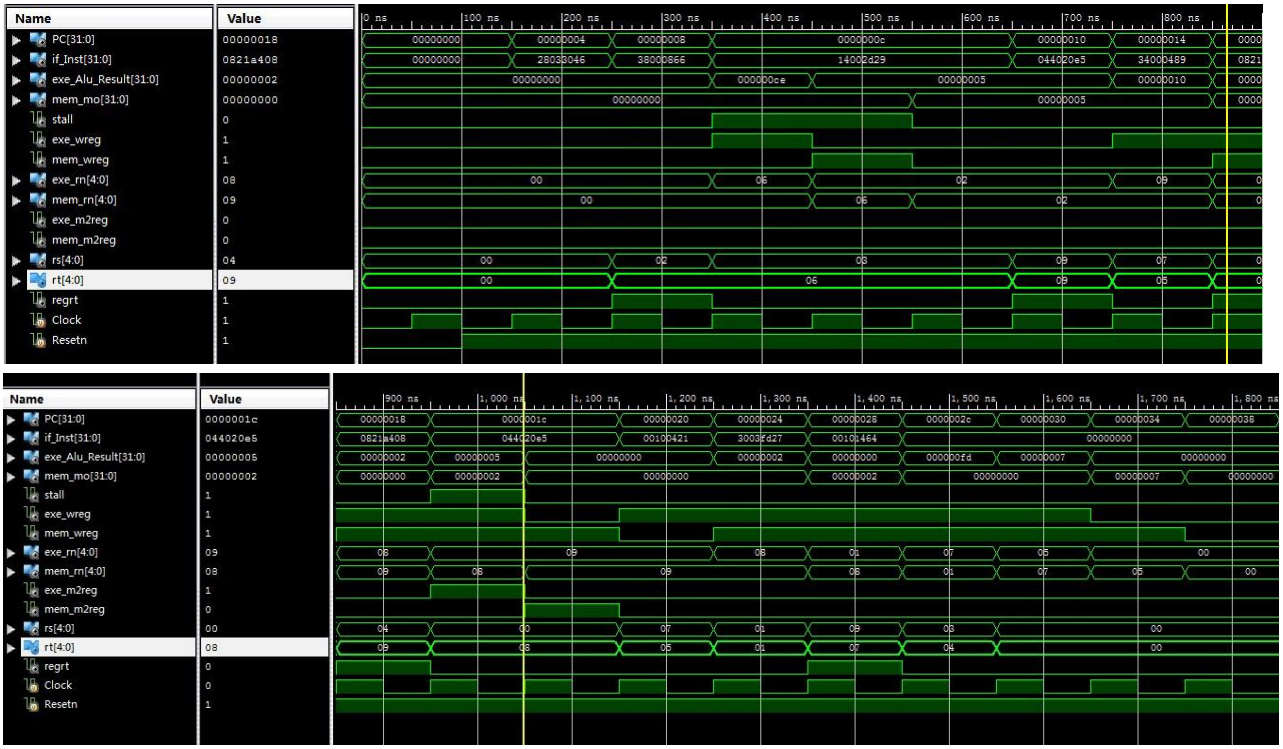
```
        // Initialize Inputs
        Clock = 0;
        Resetn = 0;

        // 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：

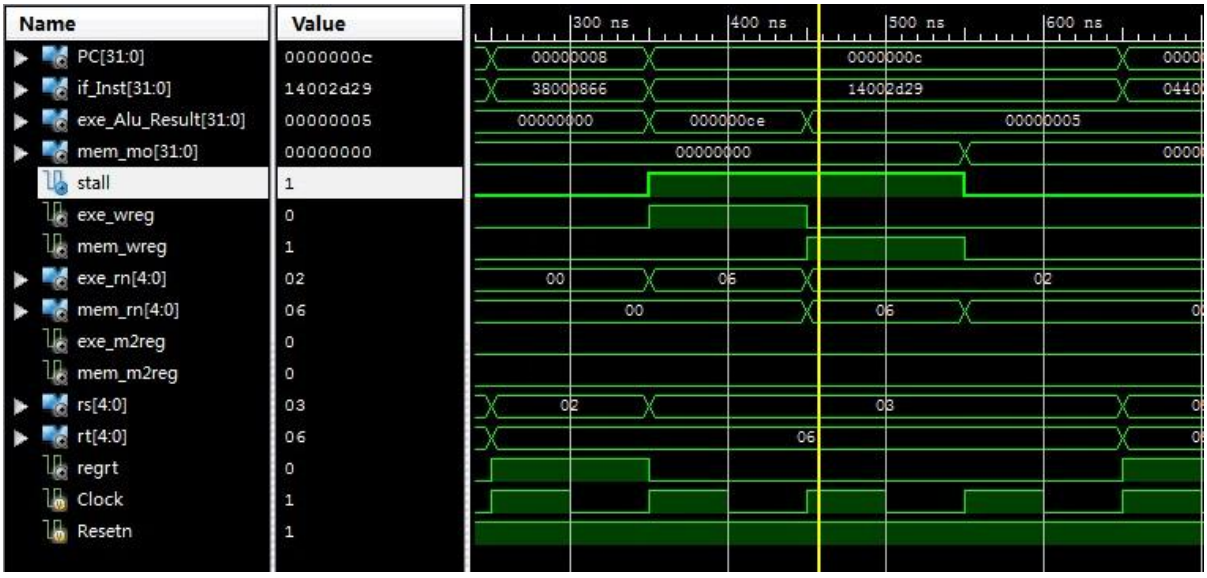
根据设计的指令集，可以发现存在三处数据冒险（如下图红色部分），第一处需要暂停两个周期，第二处需要暂停一个周期，第三处可以通过将 Regfile 的时钟去反来实现前半周期写，后半周期读。

时钟周期数	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ori r6,r2,0x00cc	ori	r2,0x00cc	or		r6									
store r6,0x0002(r3)		store	r3,r6	+	mem									
.addi r9,r9,0x000b			addi	r9	+		r9							
xor r8,r7,r5				xor	r7,r5	xor		r8						
load r9,0x0001(r4)					load	r4	+	mem	r9					
sr1 r9,r8,5'h03						srl	r8	<<		r9				
xor r8,r7,r5							xor	r7,r5	xori		r8			
add r1,r1,r1								add	r1	+		r1		
xori r7,r9,0x00ff									xori	r9	xor		r7	
add r5,r3,r4										add	r3,r4	+		r5

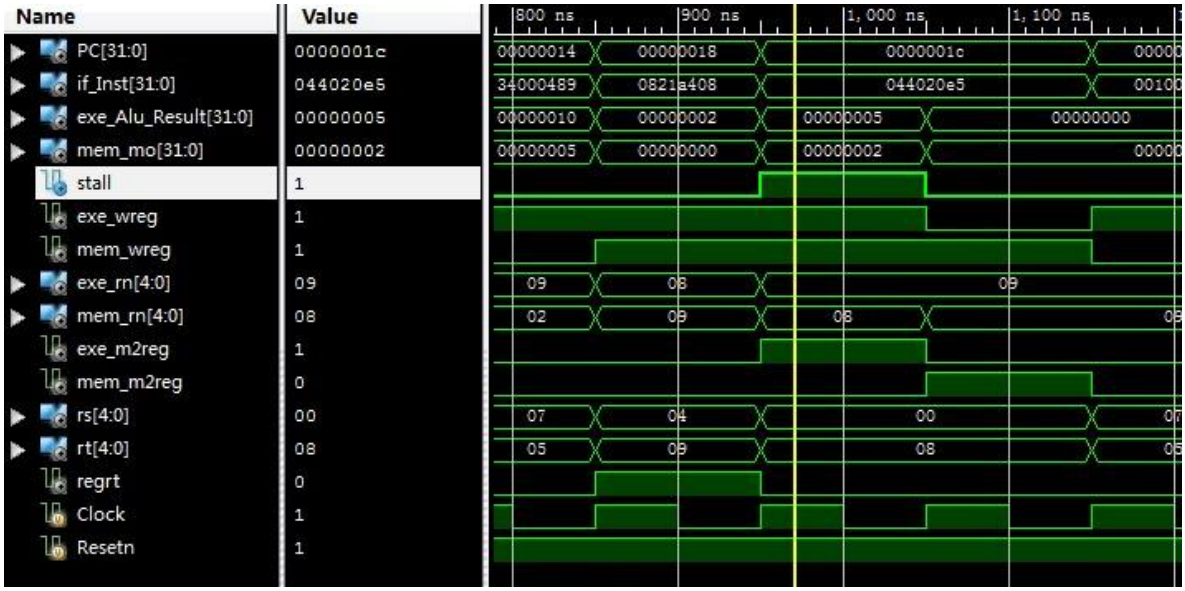
根据修改的方案，解决数据冒险后的预期结果如下图所示，其中，以第一处的数据冒险进行分析，store 指令在 ID 级得知需要读 r3,r6 的数据，但是第一条指令需要写 r6，因此 store 指令暂停执行，但是下个周期 IF 级会将下条指令 addi 取出，但是同样会被暂停，然后在第六个时钟周期，r6 在时钟的前半周期写入 regfile，后半周期 store 语句将数据读出来，流水线恢复正常，共暂停了两个周期。

时钟周期数	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ori r6,r2,0x00cc	ori	r2,0x00cc	or		r6									
store r6,0x0002(r3)(ID后阻塞)		store	r3,r6	stall										
stall			addi	stall										
stall（可继续执行了）				addi	r3,r6	+	mem							
.addi r9,r9,0x000b					addi	r9	+		r9					
xor r8,r7,r5						xor	r7,r5	xor		r8				
load r9,0x0001(r4)							load	r4	+	mem	r9			
sr1 r9,r8,5'h03（ID后阻塞）								srl	r8	stall				
stall（可继续执行了）									xor	r8	<<		r9	
xor r8,r7,r5										xor	r7,r5	xori		r8
add r1,r1,r1											add	r1	+	
xori r7,r9,0x00ff												xori	r9	xor
add r5,r3,r4													add	r3,r4

下图即为 store 发生数据冒险后加入 stall 后的仿真结果，可以看到在 200-300ns 周期，IF 取指 store，然后在 300-400ns 周期，IF 取指 addi,store 指令译码得知将会发生数据冒险，产生 stall 信号，stall 信号维持两个周期，流水线就暂停了两个周期。

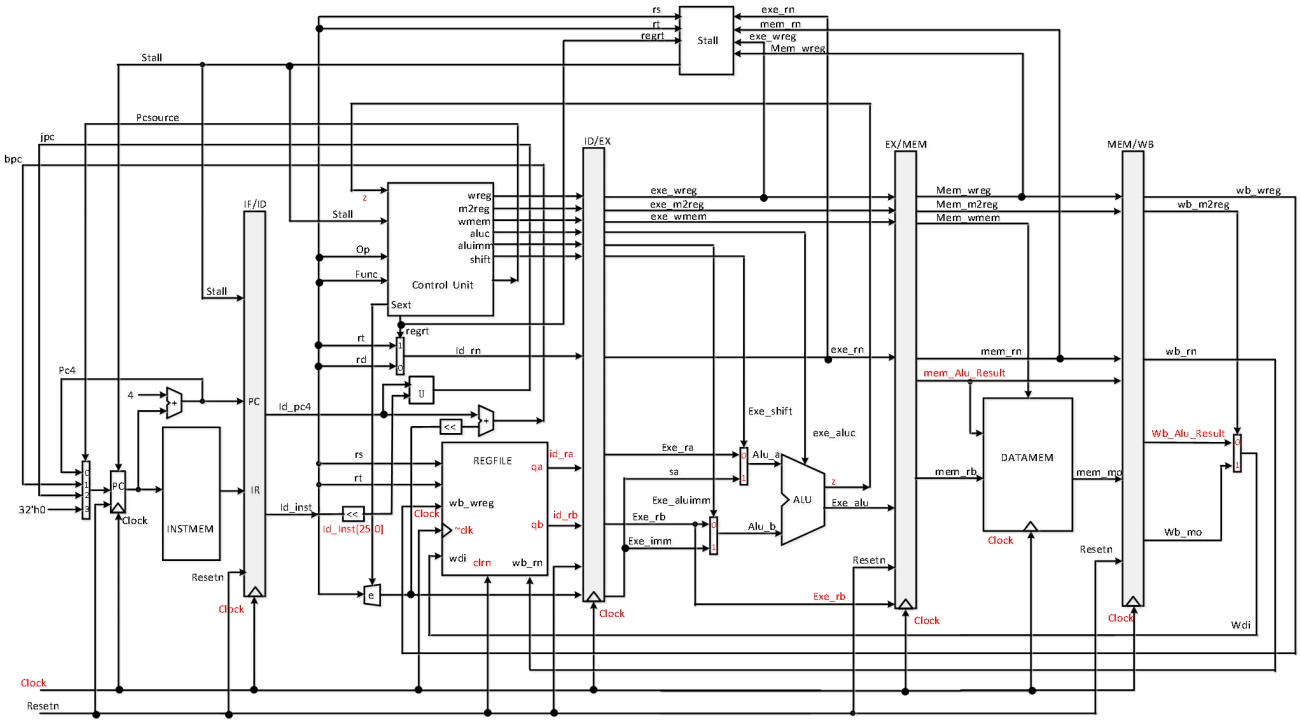


如下图所示为第二处冒险的处理仿真图，其原理与第一处相似，此处产生一个时钟周期的 stall，因此不再赘述。



结论：**整体符合预期。**

4. 解决数据冒险的五级流水线 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，在解决数据冒险的过程中，指令中加入了 branch 指令，并在将要 stall 的语句后，因为没有处理控制冒险，预期使用一条 beq 指令但不跳转，但是在 stall 之后 ALU 结果被清 0，因此导致产生了控制冒险，后续将该指令替换出去来解决这个问题。同时设计指令集也应该覆盖到所有情况不是不能引入其

他类型的冒险。

十一、总结及心得体会：

通过本次实验，接触到了处理冒险的方法，也将理论上的冒险处理实现到了电路之上，在实现电路过程中又发现了很多的细节，加深了对数据冒险的理解，在解决数据冒险中，也看到了实际仿真的结果，同时引出多个数据也方便查到逻辑错误。

十二、对本实验过程及方法、手段的改进建议：

实验可以设计带有多种冒险的指令集，在流水线中实现多种冒险的处理电路，同时可以采用更复杂的电路来处理数据冒险，以便流水线的工作效率最大化。

报告评分：

指导教师签字：