《计算机组成原理》实验报告

年级、专业、班级	2020 级计算机科学与技术(卓越)01 班	**姓名**	徐聪
实验题目	实验四简单五级流水线 CPU		
实验时间	2022 年 5 月 14 日	**实验地点**	DS1404
实验成绩	优秀/良好/中等	**实验性质**	□验证性 ☑设计性 □综合性

教师评价：

□算法/实验过程正确； □源程序/实验内容提交； □程序结构/实验步骤合理;

□实验结果正确； □语法、语义正确； □报告规范;

其他:

评价教师: 钟将

实验目的

(1)掌握流水线 (Pipelined) 处理器的思想。

(2)掌握单周期处理中执行阶段的划分。

(3)了解流水线处理器遇到的冒险。

(4)掌握数据前推、流水线暂停等冒险解决方式。

报告完成时间: 2022 年 5 月 19 日

1 实验内容

阅读实验原理实现以下模块：

(1) Datapath，所有模块均可由实验三复用，需根据不同阶段，修改 mux2 为 mux3(三选一选择器)，以及带有 enable(使能)、clear(清除流水线) 等信号的触发器，
(2) Controller，其中 main decoder 与 alu decoder 可直接复用，另需增加触发器在不同阶段进行信号传递
(3) 指令存储器 inst_mem(Single Port Ram)，数据存储器 data_mem(Single Port Ram)；同实验三一致，无需改动，
(4) 参照实验原理，在单周期基础上加入每个阶段所需要的触发器，重新连接部分信号。实验给出 top 文件，需兼容 top 文件端口设定。
(5) 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

2 实验设计

2.1 冒险处理模块

2.1.1 功能描述

实现包含 fetch、decode、execute、memory、writeback 五个阶段的 5 级流水线 cpu，并能够支持 R-type、beq、jump 等 mips 指令的执行。

2.1.2 接口定义

3 实验过程记录

3.1 实验完成工作

1. 分析 mips 指令执行原理，分析数据在数据通路中的流动，分析控制信号对各个模块的控制。
2. 完成 controller、hazard、pc 等模块的代码编写
3. 实现数据通路中的各个模块连接，控制信号的连接，以及数据之间的传送连接。
4. 利用 datapath 将各个模块以及流水线寄存器连接到一起，实现流水线 cpu 的 fetch、decode、execute、memory、writeback 五个阶段功能。
5. 对 R-type、lw、sw、jump、beq 等指令的执行结果验证以及信号分析。
6. 各种 bug 的调试、仿真时序图的分析、计算结果的验证。

模块层次：流水线 cpu 的顶层模块为 datapath，其中包含 pc、pc、寄存器堆、alu、controller、hazard、D 触发寄存器等主要的模块。寄存器堆负责寄存器数据的存储以及读取；alu 负责数据的计算；controller 模块中包含 main_decoder 以及 alu_control 模块，分别负责主要控制信号和 alu 控制信号

表 1: Controller 模块接口定义

信号名	方向	位宽	功能描述
memtoreg	input	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
memwrite	input	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
alusrc	input	1-bit	送入 ALU B 端口的值是立即数的 32 位扩展/寄存器堆读取的值
regdst	input	1-bit	送入 ALU B 端口的值是立即数的 32 位扩展/寄存器堆读取的值
regwrite	input	1-bit	是否需要写寄存器堆
branch	input	1-bit	是否需要写寄存器堆
jump	input	1-bit	是否为 jump 指令
alucontrol	input	3-bit	ALU 控制信号，代表不同的运算类型

表 2: Datapath 模块接口定义

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟信号
rst	input	1-bit	重置信号
instr	input	32-bit	32 位的 mips 指令
readdata	input	32-bit	内存数据读取接口
aluout	output	32-bit	连接内存的地址接口
pc	output	32-bit	pc 值用于获取指令 memory 的数据地址
writedata	output	32-bit	写入内存的数据接口
memwrite	output	1-bit	内存的写使能信号

的译码工作；hazard 模块负责解决数据控制冒险冲突；四个寄存器负责流水线的数据前推。此外还有两个单端口 RAM 的 ip 即 inst_mem 和 data_mem。inst_mem 和 data_mem 分别负责指令和数据的存储。

这里我对实验指导书中的代码进行了部分的改写。去除了 mips 模块，将 controller 模块放到 datapath 模块中成为其子模块，避免了数据之间的交互麻烦，同时将数据通路中的数据和指令同时进行流水线的前推尽量避免了数据交互导致的问题。

3.2 实验出现的问题

3.2.1 没有时钟降频导致指令延后

问题描述：通过观察仿真时序图，发现 pc 和 inst 之间延迟了两个周期，即 pc 对应的 inst 指令出现在了 pc 指令后第二个周期。

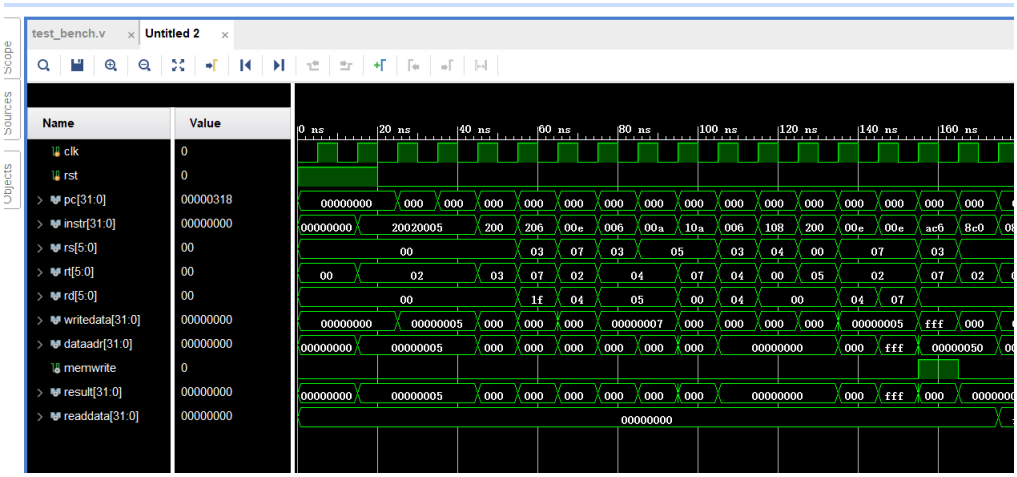

图 1: pc 和对应的 inst 指令延迟两个周期

问题分析：通过分析，问题应该出现在了 pc 和 inst 之间的信号连接模块上。通过检查顶层 top 模块中 pc 和 inst 的信号连接，发现 datapath 和 inst_mem 模块之间的 clk 信号是相同的。这样导致了在获取 pc 的上升沿时刻并不能同时将 inst_mem 中 pc 对应的 mips 指令读出。导致 inst 落后于 pc 两个周期。

解决方案：在 top 文件中，设置一条 lclk 信号连接降频 10 倍的 clk，datapath 连接降频后的 lclk 信号，inst_mem 以及 data_mem 连接高频 clk 信号。这样可以降低 pc 与 inst_mem 以及 data_mem 之间的信号延迟到一个周期以内。但是也发现 inst 与 pc 之间会有一定的信号延迟，根据降频信号的倍数而改变。

3.2.2 RsD、RtD 信号未定义

问题描述：在 DataPath 中，RSD、RtD 信号用于数据冒险以及控制冒险的状态的判定，但是在该模块中并没有定义这两个 5 位信号，导致仿真到需要判断控制冒险的 mips 代码时仿真失败。

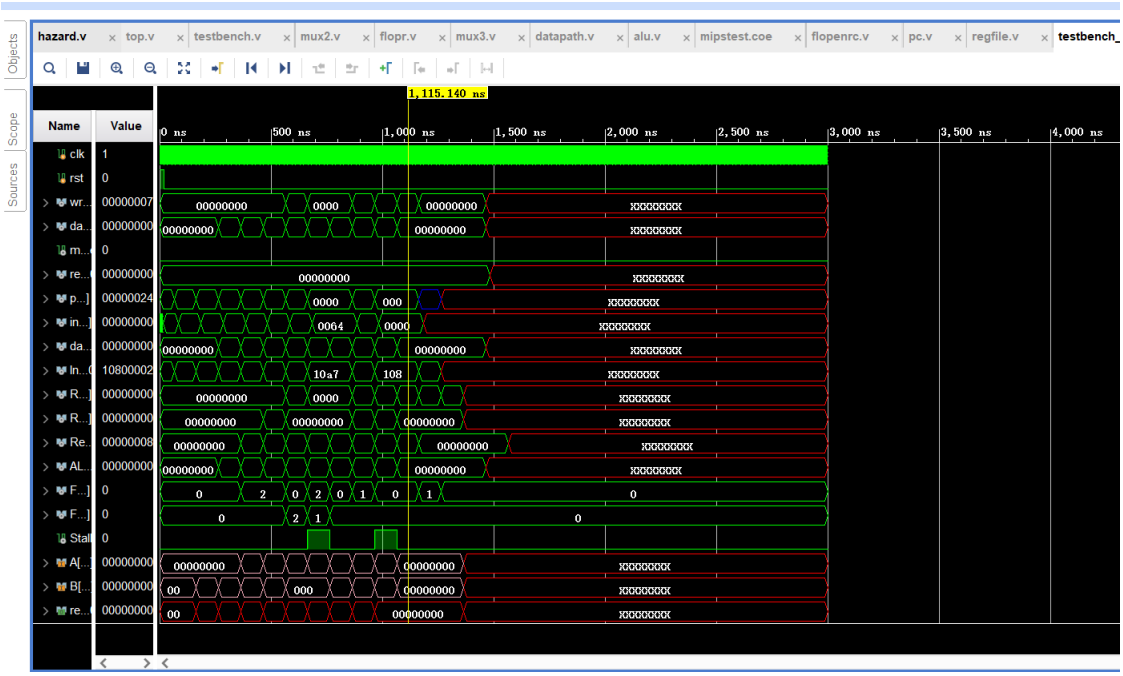

图 2: rsd、rtd 信号未定义导致仿真失败

解决方案:添加 RsD、RtD 信号的定义即可解决仿真问题。

3.2.3 寄存器缺失 posedge 敏感信号触发

问题描述:分析实验中给出的 mips 代码中第 4 条指令以及第 5 条指令,如下图所示。经过分析,这两条指令 RsD 以及 RtD 均要用到前前前一条指令向寄存器堆中写入的数据,因此我分析是寄存器的读写在同一周期导致的数据冲突问题,即当在同一个周期里面实现读出刚写入的数据。

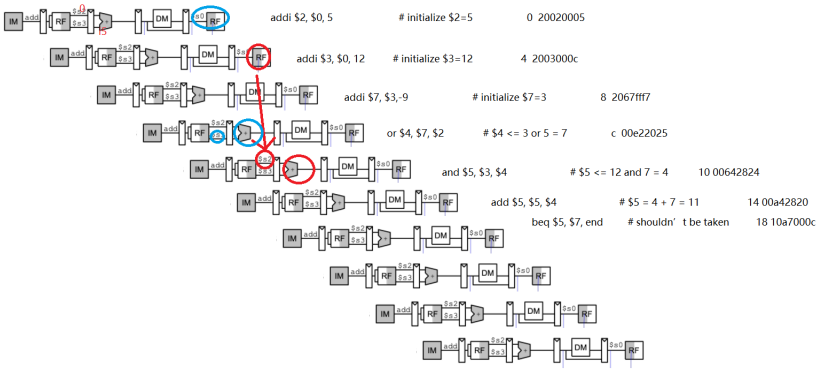

图 3: 4、5 指令的数据冒险问题

解决方案:将寄存器堆的 clk 设置为下降沿触发。

3.2.4 没有添加 jump 指令执行部件

问题描述：文档中给出的 mips 指令倒数第 3 条为 jump 指令，但实验指导书中数据通路图并没有支持 jump 指令的部件，因此导致最后指令的跳转失败。

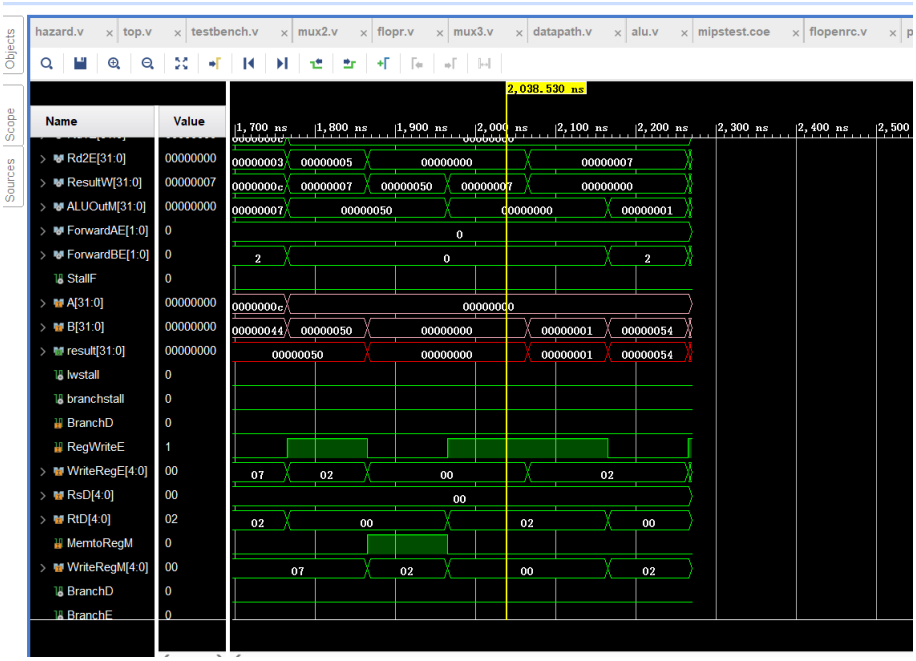

图 4: jump 指令未成功跳转

解决方案：按照实验三中给出的数据通路图添加 jump 指令部件，即可以实现 jump 指令的跳转。

4 实验结果及分析

4.1 实验结果展示

4.2 结果分析

通过分析各条指令的执行情况，以及 alu 计算结果，根据实验指导文件中给出来的汇编语言表，以及 mips 指令文件。设计的 5 级流水线 cpu 正确高效的完成了 fetch、decode、execute、memory、writeback 各个阶段的功能，最后成功地输出了计算结果！

A Datapath 代码

```
‘timescale 1ns / 1ps


module datapath(
```

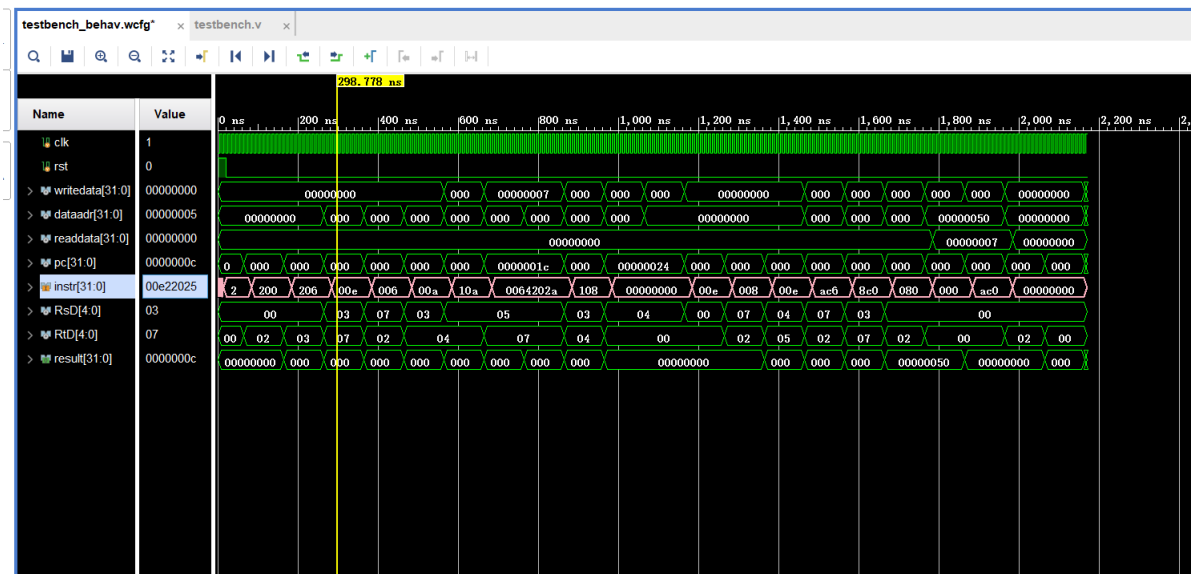

图 5: 实验成功仿真时序图

```
#   }
# }
# log_wave -r /
# run 3000ns
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.imem.inst.native_mem_module.blk_mem_g
Block Memory Generator module testbench.dut.dmem.inst.native_mem_module.blk_mem_g
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 3000ns
launch_simulation: Time (s): cpu = 00:00:08 ; elapsed = 00:00:07 . Memory (MB): p
```

图 6: 控制台输出 Simulation succeeded

```
input clk,
input rst,
input [31:0] instr,
input [31:0] readdata,

//----------------------------
output [31:0] aluout,
output [31:0] pc,
output [31:0] writedata,
output [31:0] memwrite
);

// 数据信号-----------------------
wire [31:0] PC_, PC_new, PCF, InstrD;
wire [31:0] PCPlus4F, PCPlus4D, PCPlus4E;
wire [31:0] Rd1D, Rd2D, Rd1E, Rd2E;
wire [4:0] RtE, RdE, RsE, RsD, RtD, RdD;
wire [31:0] SignImmD, SignImmE;
wire [31:0] SrcAE, SrcBE, WriteDataE,WriteDataM;
wire [4:0] WriteRegE, WriteRegM, WriteRegW;
wire [31:0] PCBranchE, PCBranchM, PCBranchD;
wire [31:0] ALUOutE, ALUOutM, ALUOutW;
wire ZeroE, ZeroM;
```

```
wire [31:0] ReadDataW, ResultW;
//-------------------------------

// 控制信号-----------------------
wire RegWriteD, RegWriteE, RegWriteM, RegWriteW;
wire MemtoRegD, MemtoRegE, MemtoRegM, MemtoRegW;
wire MemWriteD, MemWriteE, MemWriteM;
wire BranchD, BranchE, BranchM;
wire [2:0] ALUControlD, ALUControlE;
wire ALUSrcD, ALUSrcE;
wire RegDstD, RegDstE;
wire PCSrcM, PCSrcD;
wire JumpD;
//--------------------

// 竞争处理信号------------------
wire [1:0]ForwardAE, ForwardBE;
wire StallF, StallD, FlushE;
wire ForwardAD, ForwardBD;
//---------------------------------


//信号连接-----------------------
assign aluout = ALUOutM;
assign pc = PCF;
assign writedata = WriteDataM;
assign memwrite = MemWriteM;

assign RsD = InstrD[25:21];
assign RtD = InstrD[20:16];
assign RdD = InstrD[15:11];
//---------------------------



// 竞争冒险处理
hazard ha(
    .RsE(RsE),
    .RtE(RtE),
    .RsD(RsD),
    .RtD(RtD),
    .WriteRegM(WriteRegM),
    .WriteRegW(WriteRegW),
    .WriteRegE(WriteRegE),
    .RegWriteM(RegWriteM),
    .RegWriteW(RegWriteW),
    .RegWriteE(RegWriteE),
    .MemtoRegE(MemtoRegE),
    .MemtoRegM(MemtoRegM),
```

```
    .MemtoRegW(MemtoRegW),
    .BranchD(BranchD),

    //----------
    .ForwardAE(ForwardAE),
    .ForwardBE(ForwardBE),
    .ForwardAD(ForwardAD),
    .ForwardBD(ForwardBD),
    .StallF(StallF),
    .StallD(StallD),
    .FlushE(FlushE)
);



// Fetch阶段
mux2 #(32) pc_mux1(.a(PCPlus4F), .b(PCBranchD), .f(PCSrcD), .c(PC_));
mux2 #(32) pc_mux2(.a(PC_), .b({PCPlus4F[31:28], InstrD[25:0], 2'b00}), .f(
    JumpD), .c(PC_new));

pc p(
    .clk(clk),
    .rst(rst),
    .clr(1'b0),
    .en(~StallF),
    .newpc(PC_new),
    .pc(PCF)
);

assign PCPlus4F = PCF + 32'h4;

// 32+32
flopenrc #(200) flop_D(
    .clk(clk),
    .rst(rst),
    .en(~StallD),
    .clear(PCSrcD),
    .d({instr, PCPlus4F}),
    .q({InstrD, PCPlus4D})
);



// Decode阶段
// 控制竞争
wire [31:0]t_rd1, t_rd2;
mux2 rd1_mux(.a(Rd1D), .b(ALUOutM), .f(ForwardAD), .c(t_rd1));
mux2 rd2_mux(.a(Rd2D), .b(ALUOutM), .f(ForwardBD), .c(t_rd2));
```

```
  assign PCSrcD = BranchD & (t_rd1 == t_rd2);

  controller c(
  .inst(InstrD),
  //------------------
  .regwrite(RegWriteD),
  .memtoreg(MemtoRegD),
  .memwrite(MemWriteD),
  .branch(BranchD),
  .alucontrol(ALUControlD),
  .alusrc(ALUSrcD),
  .regdst(RegDstD),
      .jump(JumpD)
);

  regfile rf(
      .clk(clk),
      .we3(RegWriteW),
      .ra1(InstrD[25:21]),
      .ra2(InstrD[20:16]),
      .wa3(WriteRegW),
      .wd3(ResultW),
      .rd1(Rd1D),
      .rd2(Rd2D)
  );


  assign SignImmD = {{16{InstrD[15]}}, InstrD[15:0]};
  adder branch_add(.a({SignImmD[29:0], 2'b00}), .b(PCPlus4D), .y(PCBranchD));

  // (1+1+1+1+1+1+3)+(32+32+5+5+5+32) = 120
  flopenrc #(200) flop_E(
      .clk(clk),
      .en(1'b1),
      .clear(FlushE),
      .rst(rst),
      .d({RegWriteD,MemtoRegD,MemWriteD,BranchD,ALUControlD,ALUSrcD,RegDstD,
          Rd1D, Rd2D, InstrD[25:11], SignImmD}),
      .q({RegWriteE,MemtoRegE,MemWriteE,BranchE,ALUControlE,ALUSrcE,RegDstE,
          Rd1E, Rd2E, RsE, RtE, RdE, SignImmE})
  );



  // Execute阶段
  alu #(32) alu(.A(SrcAE), .B(SrcBE), .F(ALUControlE), .result(ALUOutE));
  mux2 #(32) alu_mux(.a(WriteDataE), .b(SignImmE), .f(ALUSrcE), .c(SrcBE));
  mux2 #(5) reg_mux(.a(RtE), .b(RdE), .f(RegDstE), .c(WriteRegE));
  // assign ZeroE = ALUOutE == 32'b0;
```

```
    //数据前推
    mux3 SrcAE_mux(.a(Rd1E), .b(ResultW), .c(ALUOutM), .f(ForwardAE), .y(SrcAE)
        );
    mux3 SrcBE_mux(.a(Rd2E), .b(ResultW), .c(ALUOutM), .f(ForwardBE), .y(
        WriteDataE));

    // (1+1+1)+(32+32+5) = 72
    flopenrc #(200) flop_M(
        .clk(clk),
        .rst(rst),
        .en(1'b1),
        .clear(1'b0),
        .d({RegWriteE,MemtoRegE,MemWriteE, ALUOutE, WriteDataE,WriteRegE}),
        .q({RegWriteM,MemtoRegM,MemWriteM, ALUOutM, WriteDataM, WriteRegM})
    );



    // Memory阶段
    // assign PCSrcM = BranchM & ZeroM;
    // assign PCSrcM = 1'b0;

    // (1+1)+(32+32+5) = 71
    flopenrc #(200) flop_W(
        .clk(clk),
        .rst(rst),
        .clear(1'b0),
        .en(1'b1),
        .d({RegWriteM,MemtoRegM,ALUOutM, readdata, WriteRegM}),
        .q({RegWriteW,MemtoRegW,ALUOutW, ReadDataW, WriteRegW})
    );

    // Write Back阶段
    mux2 #(32) result_mux(.a(ALUOutW), .b(ReadDataW), .f(MemtoRegW), .c(ResultW
        ));


endmodule
```

B Hazard 代码

```
`timescale 1ns / 1ps


module hazard(
    input [4:0] RsE,RtE, RsD, RtD,
```

```
    input [4:0] WriteRegM, WriteRegW, WriteRegE,
    input RegWriteM, RegWriteW, RegWriteE,MemtoRegE,MemtoRegM, MemtoRegW,
    input BranchD,

    output reg [1:0] ForwardAE, ForwardBE,
    output wire ForwardAD, ForwardBD,
    output StallF, StallD, FlushE
    );

    always@(RsE or RtE or WriteRegM or WriteRegW or RegWriteM or RegWriteW)
        begin
        ForwardAE=2'b00;
        ForwardBE=2'b00;
        if(RsE!=0) begin
            if(RsE==WriteRegM&&RegWriteM)
                ForwardAE=2'b10;
            else if(RsE==WriteRegW&&RegWriteW)
                ForwardAE=2'b01;
        end
        if(RtE!=0) begin
            if(RtE==WriteRegM&&RegWriteM)
                ForwardBE=2'b10;
            else if(RtE==WriteRegW&&RegWriteW)
                ForwardBE=2'b01;
        end
    end

    wire branchstall;
    assign branchstall = BranchD &
                    (RegWriteE &
                    (WriteRegE == RsD | WriteRegE == RtD) |
                    MemtoRegM &
                    (WriteRegM == RsD | WriteRegM == RtD));

    assign ForwardAD = (RsD != 0 & RsD == WriteRegM & RegWriteM);
    assign ForwardBD = (RtD != 0 & RtD == WriteRegM & RegWriteM);

    wire lwstall;
    assign lwstall = MemtoRegE & (RtE == RsD | RtE == RtD);
    assign StallD = lwstall | branchstall;
    assign StallF = StallD;
    assign FlushE = StallD;



endmodule
```

C Controller 代码

```
‘timescale 1ns / 1ps


module hazard(
    input [4:0] RsE,RtE, RsD, RtD,
    input [4:0] WriteRegM, WriteRegW, WriteRegE,
    input RegWriteM, RegWriteW, RegWriteE,MemtoRegE,MemtoRegM, MemtoRegW,
    input BranchD,

    output reg [1:0] ForwardAE, ForwardBE,
    output wire ForwardAD, ForwardBD,
    output StallF, StallD, FlushE
    );

    // 数据前推
    always@(RsE or RtE or WriteRegM or WriteRegW or RegWriteM or RegWriteW)
        begin
        ForwardAE=2'b00;
        ForwardBE=2'b00;
        if(RsE!=0) begin
            if(RsE==WriteRegM&&RegWriteM)
                ForwardAE=2'b10;
            else if(RsE==WriteRegW&&RegWriteW)
                ForwardAE=2'b01;
        end
        if(RtE!=0) begin
            if(RtE==WriteRegM&&RegWriteM)
                ForwardBE=2'b10;
            else if(RtE==WriteRegW&&RegWriteW)
                ForwardBE=2'b01;
        end
    end

    wire branchstall;
    assign branchstall = BranchD &
                    (RegWriteE &
                    (WriteRegE == RsD | WriteRegE == RtD) |
                    MemtoRegM &
                    (WriteRegM == RsD | WriteRegM == RtD));

    assign ForwardAD = (RsD != 0 & RsD == WriteRegM & RegWriteM);
    assign ForwardBD = (RtD != 0 & RtD == WriteRegM & RegWriteM);

    wire lwstall;
    assign lwstall = MemtoRegE & (RtE == RsD | RtE == RtD);
    assign StallD = lwstall | branchstall;
    assign StallF = StallD;
```

```
    assign FlushE = StallD;


endmodule
```