1 实验内容

在实验三中已介绍过 MIPS 传统的五级流水线阶段，分别是取指、译码、执行、访存、写回 (Instruction Fetch, Decode, Execution, Memory Request, Write Back)，五阶段。单周期 CPU 虽然 CPI 为 1，但由于时钟周期取决于时间最长的指令 (如 lw、sw 存取耗时巨大)，因此没有很好的性能，而多周期虽然能提升性能但仍旧无法满足当今处理器的需求。流水线能够很好地解决效率问题，通过分阶段，达到指令的并行执行。同时，在单周期的基础上，能够很容易地使用触发器做阶段分隔，实现流水线。

本次实验将从实验三单周期处理器过渡至五级流水，并将解决冒险 (hazard) 问题。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 实验原理

实验三已完成单周期 CPU，数据通路如图 1，实验四待完成的流水线 CPU 如图 2。可以看到，流水线 CPU 的主要改动是在各级流水线之间加入流水线寄存器（触发器），使得每个周期执行一条指令的一个阶段，得到的结果传入和下一级之间的流水线寄存器，在下个周期由下一级流水线取出数据并执行，这样能够使指令各阶段并行执行，提升效率。

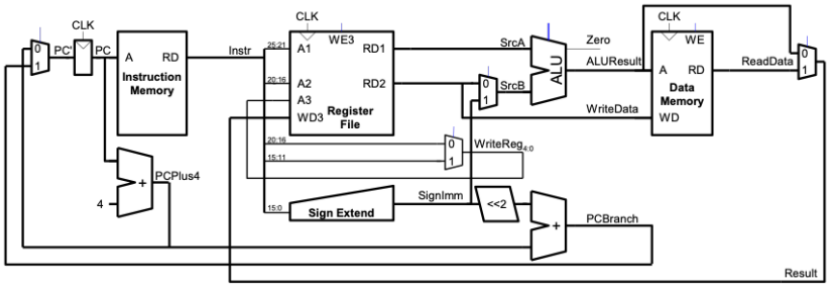

图 1: 单周期 CPU

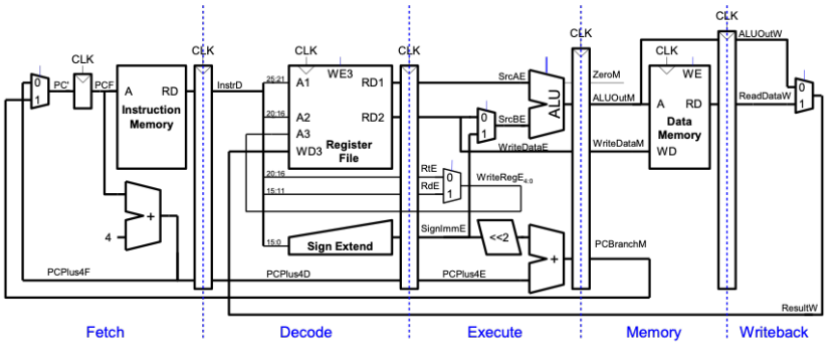

图 2: 流水线 CPU

图 2 中流水线 CPU 有一个问题，因为 R 指令的 ALUOutW 和 lw 指令的 ReadDataW 都是在第五阶段读出，因此 writereg 信号无法直接回到寄存器堆，需要不断传入下一级流水线寄存器，直到第五级流水线读出数据一起传入寄存器堆。修改后的数据通路如图 3 所示：

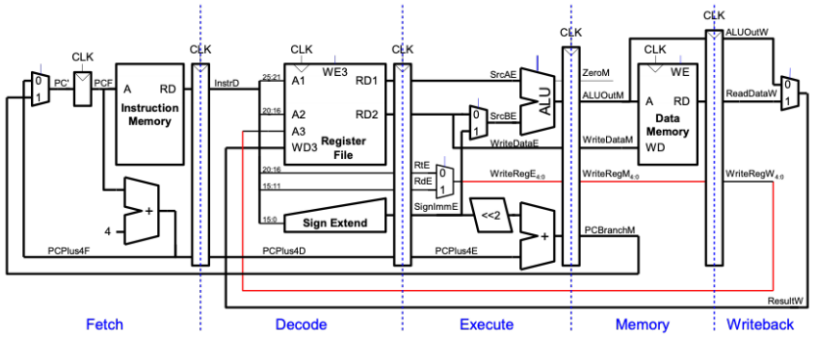

图 3: 修改 writereg 信号

在此基础上，datapath 的基本通路已经形成，下面加入控制器部分，由 Main Decoder 和 ALU Decoder 构成。但由于改为五级流水线后，每一个阶段所用到的的控制信号仅为部分，控制器产生信号的阶段为译码阶段，产生控制信号后，依次通过流水线寄存器传到下一阶段，若当前阶段需要的信号，则不需要继续传递到下一阶段。加入控制信号的数据通路如图 4：

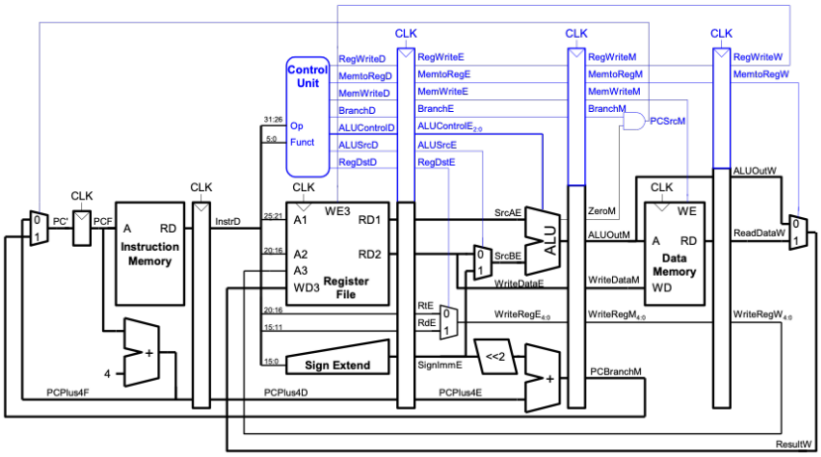

图 4: 加入控制信号的 CPU

2.2 冒险处理模块

在流水线 CPU 中，并不是能够完全实现并行执行，常见的冲突有竞争和冒险。竞争是门电路两个输入信号同时向相反方向的逻辑电平跳变而导致的现象，本实验不作处理。而对于冒险，在单周期中由于每条指令执行完毕才会执行下一条指令，并不会遇到冒险问题，而在流水线处理器中，由于当前指令可能取决于前一条指令的结果，但此时前一条指令并未执行到产生结果的阶段，这时候，就产生了冒险。本实验设计 hazard 模块处理冒险。

冒险分为：

1. 数据冒险：寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据；

2. 控制冒险：下一条要执行的指令还未确定，就按照 PC 自增顺序执行了本不该执行的指令(由分支指令引起)。

2.2.1 功能描述

1. 数据冒险：

如图 5 中指令，and、or、sub 指令均需要使用 $s0 中的数据，然而 add 指令在回写阶段才能写入寄存器堆，此时后续三条指令均已经过或正在执行译码阶段，得到的结果均为错误值。以上就是数据冒险的特点，数据冒险有以下解决方式：

(1)在编译时插入空指令；

(2)在编译时对指令执行顺序进行重排；

(3)在执行时进行数据前推；

(4)在执行时，暂停处理器当前阶段的执行，等待结果。

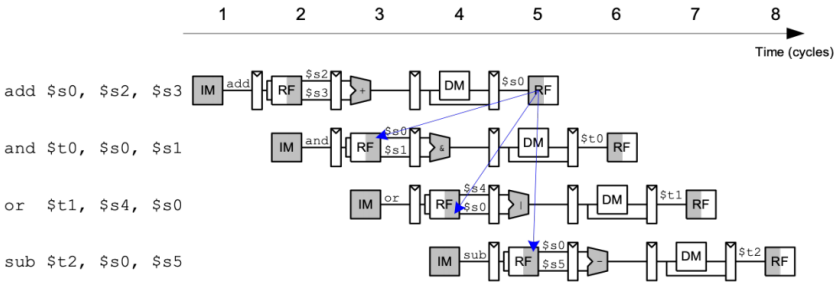

图 5: 数据冒险

但由于我们未进行指令编译层的处理，因此只需要在运行时 (run time) 进行解决，故采用数据前推和暂停处理器两种解决方案。

1.1 数据前推

从图 6 所示，add 指令的结果在 execute 阶段已经由 ALU 计算得到，此时可以将 alu 得到的结果直接推送到下一条指令的 execute 阶段，同理，后续所有的阶段均已有结果，可以向对应的阶段推送，而不需要等到回写后再进行读取，达到数据前推的目的。

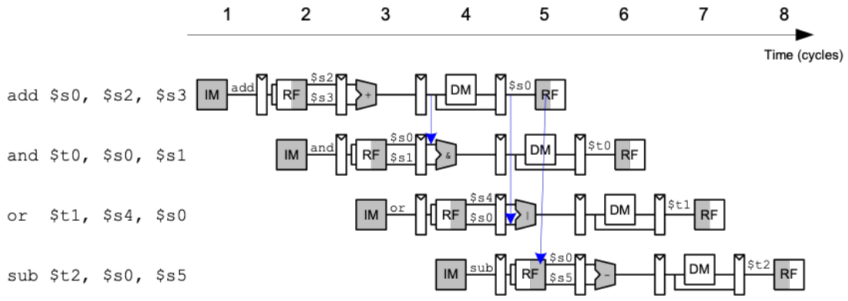

图 6: 数据前推

在 execute 阶段需要判断当前输入 ALU 的地址是否与**前两条指令**在此时执行的阶段要写入寄存器堆的地址相同(lw 的 excute 阶段)，或者当前输入 ALU 的地址是否与**前一条指令**在此时执行的阶段要写入寄存器堆的地址相同(R 的 excute[空] 或 writeback 阶段)。如果相同，就需要将其他指令的结果直接通过多路选择器输入到 ALU 中。由此可以看出，数据前推可以解决 R 型指令以及前两条 lw 指令带来的数据冒险(若 lw 后紧跟指令需要在 EX 阶段使用寄存器内的数据，数据前推无法解决)。此处需要：

(1) 增加 rs,rt 的地址传递到 execute 阶段，并与冒险模块连接 (RsD、RtD -> 流水线寄存器 -> RsE、RtE -> hazard 模块)；

(2)Memory 阶段和 writeback 阶段要写入寄存堆的地址 writereg、写使能信号 regwrite 与冒险模块连接；

(3)根据实现逻辑，将生成的 forward 信号输出，控制 mux3 选择器。

数据通路结构如下：

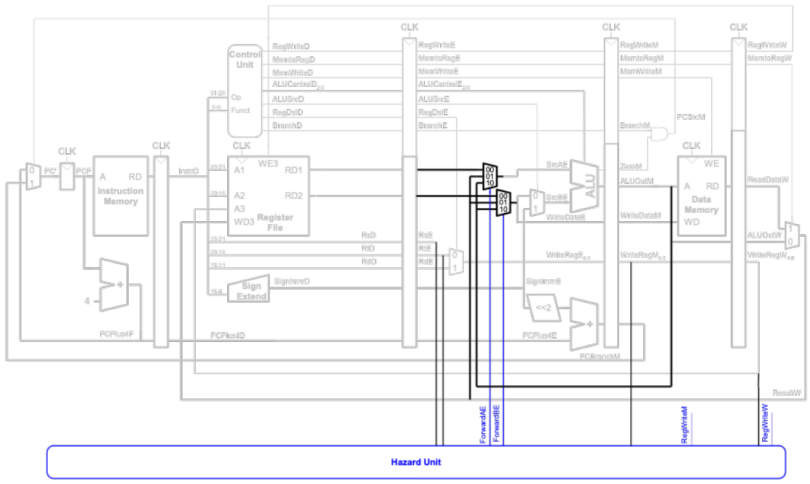

图 7: 数据前推数据通路

逻辑控制如下：

```
//数据前推解决R指令和前两条lw指令的数据冒险
   /*ALU端口SrcAE的数据可能来自：（注意判断reE!=0，否则读保留寄存器直接输出）
       寄存器堆（无冒险情况下）：forwordAE=00、SrcAE=RsD
       数据存储器(lw的数据冒险)：forwordAE=01、SrcAE=ResultW（lw指令写回寄存器堆在MEM阶段，其后第二条指令如需要该数据会受影响）
       ALUOut（ALU运算的数据冒险）：forwordAE=10、SrcAE=ALUOutM（R型指令写回寄存器堆在WB阶段，其后一条指令如需要该数据都会受影响）
   */
   assign forwordAE = ((rsE != 5'b0) & (rsE == writeregM) & regwriteM) ? 2'b10:    //前一条指令是R型，直接将ALUOut传回
                      ((rsE != 5'b0) & (rsE == writeregW) & regwriteW) ? 2'b01:    //前两条指令是lw
                       2'b00;
   assign forwordBE = ((rtE != 5'b0) & (rtE == writeregM) & regwriteM) ? 2'b10:    //SrcBE同SrcAE
                       ((rtE != 5'b0) & (rtE == writeregW) & regwriteW) ? 2'b01:
                       2'b00;
```

图 8: 数据前推逻辑控制

1.2 流水线暂停

数据前推并不是总是奏效，如图 8，lw 指令在 memory 阶段才能够从数据存储器读取数据，此时 and 指令已经完成 ALU 计算，无法进行数据前推。为了解决 lw 指令后紧跟需要在 EX 阶段使用寄存器内的数据的指令带来的数据冒险，必须使流水线暂停，等待数据读取后，再前推到 execute 阶段（如图 9）。

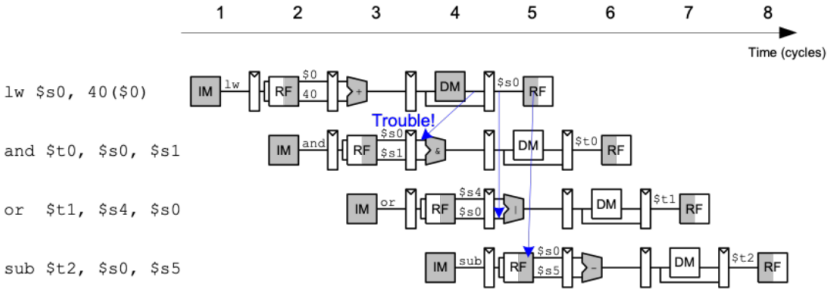

图 9: 数据前推失效

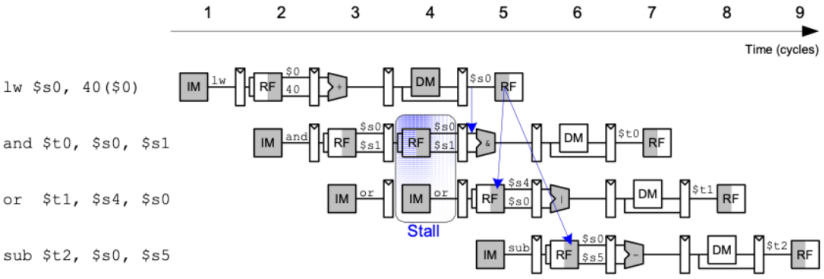

图 10: 流水线暂停

流水线暂停的条件是：前一条指令需要对寄存器堆写入 (memturegE=1) 并且写入地址 rtE 与被当前指令用 (rsD==rtE 或 rtD==rtE)。并且该暂停信号会将后几级流水线全部暂停。注意，流水线暂停 (decoder 级) 还需要一个附加操作，就是清空 excute 级的信号。因为 lw 下一条指令已经按照预期接受了流水寄存器 DE 的输出并进行工作。到下面介绍的控制冒险也会用到流水线暂停，因此将流水线暂停的数据通路和逻辑控制放在控制冒险后介绍。

2. 控制冒险：

控制冒险是分支指令引起的冒险。如图 11，在五级流水线当中，分支指令在第 4 阶段才能够决定是否跳转；而此时，前三个阶段已经导致三条指令进入流水线开始执行，这时需要将这三条

指令产生的影响全部消除。

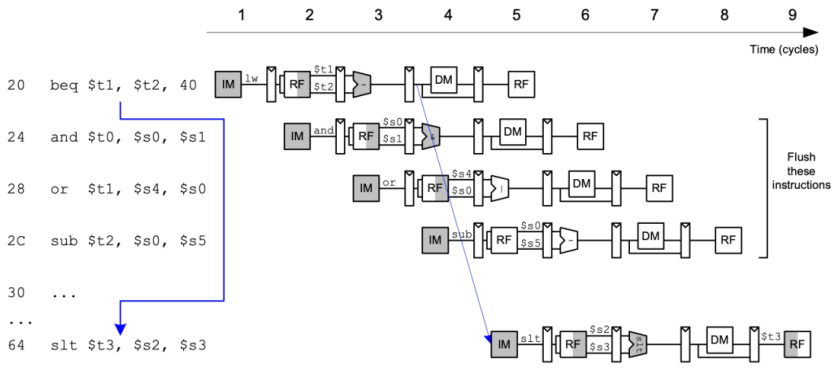

图 11: 控制冒险

一个解决办法是将分支指令的判断提前至 decode 阶段，即在 regfile 输出后添加一个判断相等的模块，即可提前判断 beq，如图 12 所示。此操作能够减少两条指令的执行，图 13 为提前判断分支的实现。

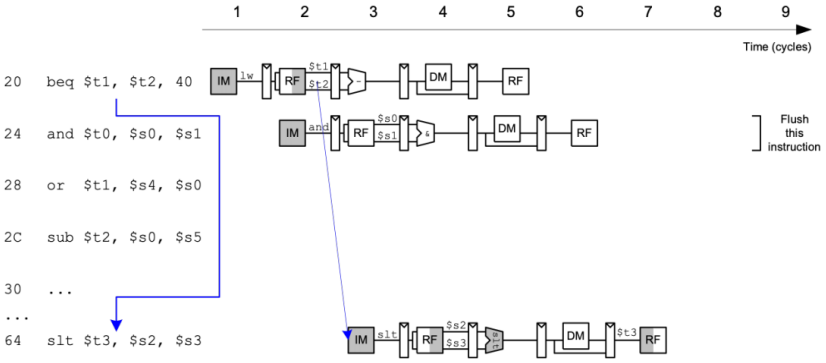

图 12: 提前判断分支

（注：提前判断分支不在 hazard 模块中，而是在 datapath 模块中。）

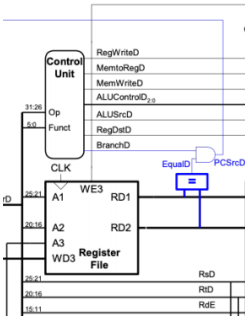

图 13: 提前判断分支实现

但是提前判断分支可能导致所需的数据还没有传入对应的寄存器，而且提前判断分支在 decoder 级操作，上一条指令的 ALU 或 lw 还没有操作，因此必须进行流水线暂停等待上一条指令执行，为了将执行出的结果立即传入当前指令，还需要数据前推。

此外，为避免 branch 下一条指令在 branch 跳转前执行，加入清空该级流水线的操作。于是解决控制冲突的数据通路如下所示：

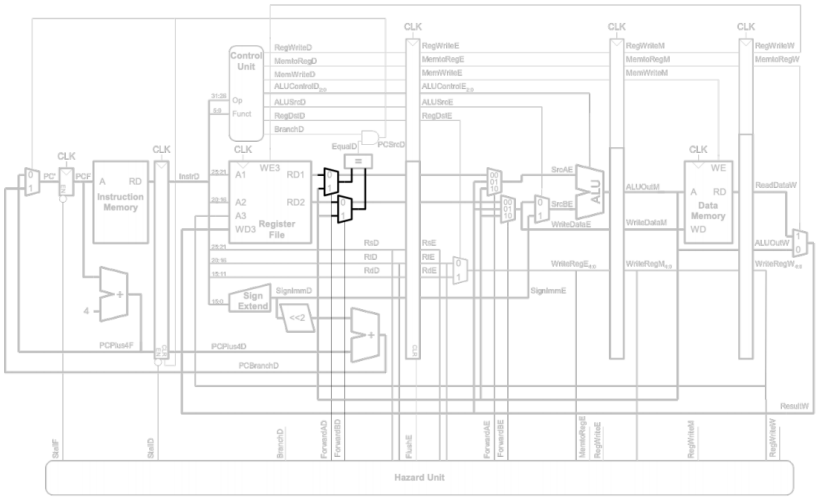

图 14: 提前判断分支数据通路

控制冒险的逻辑控制如下：

```
//提前判断分支解决beq后2、3条指令的控制冒险时出现的数据冒险
    assign forwordAD = ((rsD != 5'b0) & (rsD == writeregE) & regwriteE);            //前一条指令要写回寄存器堆且该数据被beq指令所用
    assign forwordBD = ((rtD != 5'b0) & (rtD == writeregE) & regwriteE);
//流水线暂停解决lw后一条指令需要用寄存器堆数据带来的数据冒险、beq需要用前一条指令写回寄存器堆的数据
    /*lw指令写入寄存器堆的地址为rt，因此下一条指令若要用到rt则需要暂停，即rsD==rtE或rtD==rtE
      beq指令需要rs、rt号寄存器的数据，若上一条指令要写到该寄存器，则需要暂停流水线
    */
    wire lwstall,branch_stall;                                          //流水线暂停信号
    assign lwstall = (((rsD == rtE) | (rtD == rtE)) & memtoregE);   //判断前一条指令需要对寄存器堆写入(memturegE)并且写入地址rtE与被当前指令用到
    assign branch_stall=( branchD&regwriteE&((writeregE==rsD)|(writeregE==rtD)) )   //当前指令为branch、上一条指令要写寄存器堆且写的数据当前要用
                        |( branchD&memtoregM&((writeregM==rsD)|(writeregM==rtD)) );  //当前指令为branch、上2条指令要写寄存器堆且写的数据当前要用
    always @(*)begin
        stallF = rst? 1'b0 : (lwstall | branch_stall);      //若被重置则全部清零
        stallD = rst? 1'b0 : (lwstall | branch_stall);      //lw带来的数据冒险或提前判断分支带来的数据冒险都可以暂停流水线
        flushE = rst? 1'b0 : (lwstall | branch_stall);      //lw/beq下一条已经执行的需要清空
    end
```

图 15: 控制冒险逻辑控制

2.2.2 接口定义

表 1: 接口定义模版

信号名	方向	位宽	功能描述
rst	Input	1-bit	重置信号
rsD	Input	5-bit	decoder 级的 rs 信号
rtD	Input	5-bit	decoder 级的 rt 信号
rsE	Input	5-bit	excute 级的 rs 信号
rtE	Input	5-bit	excute 级的 rt 信号
regwriteE	Input	1-bit	excute 级的 rs 信号
regwriteM	Input	1-bit	memory access 级的 rs 信号
regwriteW	Input	1-bit	writeback 级的 rs 信号
memtoregE	Input	1-bit	excute 级判断写寄存器堆的数据来源
memtoregM	Input	1-bit	memory 级判断写寄存器堆的数据来源
branchD	Input	1-bit	decoder 级提前判断分支信号
writeregE	Input	5-bit	excute 级的寄存器堆写地址
writeregM	Input	5-bit	memory 级的寄存器堆写地址
writeregW	Input	5-bit	writeback 级的寄存器堆写地址
forwardAE	Output	2-bit	excute 级控制 mux3 选择 SrcA
forwardBE	Output	2-bit	excute 级控制 mux3 选择 SrcB
forwardAD	Output	2-bit	decoder 级控制 mux3 选择 SrcA
forwardBD	Output	2-bit	decoder 级控制 mux3 选择 SrcB
stallF	Output	1-bit	fetch 级暂停
stallD	Output	1-bit	decoder 级暂停
flushE	Output	1-bit	清空 excute 流水线

3 实验过程记录

3.1 问题 1:Main Decoder 解码信号传递混乱

问题描述: Main Decoder 模块解码后输出信号 jump, branch, alusrc, memwrite, memetoreg, regwrite, regdst, data_ram_ena, alucontrol, 加上中间一些连线或寄存器, 导致在 Controller 模块中信号过多太过混乱,实例化模块时非常容易出错或漏掉信号。导致仿真多次大面积出现 X 或 Z 信号。

解决方案:吸取实验三漏传信号的经验,将 Main Decoder 输出的 9 个信号连接成 9bit 的 sigs 信号输出。此处注意 Controller 模块需要输出完整的 8 个信号。

3.2 问题 2: 流水线寄存器的 rst 和 clear 信号混淆

问题描述:流水线寄存器的 rst 和 clear 看似作用相同,都是清空流水线实则不同

解决方案:。rst 相当于整个电路的开关,置 1 时整个电路归零,有可能是异常或流水线冲突;clear 是较为简单的流水线清空,有可能是人为清空流水线结束 CPU 或者流水线地址跳转却已经执行跳转前代码。因此需将 rst 和 clear 分开。

3.3 问题 3:Controller 输出 jump 和 branch 的时间

问题描述:原本准备按照图 4 将 jump 和 branch 在流水线寄存器 EM 后输出,但流水线 CPU 容易出现控制冒险,即下一条指令的跳转地址还没传出,PC 已经按照先前继续 +4 进入下一条指令。

解决方案:将 jump 和 branch 解码后立即输出,即在流水线寄存器 DE 前直接输出,防止地址进入错误位置。

3.4 问题 4: 触发器的使能端问题

问题描述:实验中 Controller 模块用到的流水线寄存器不含有使能端,datapath 模块使用的流水线寄存器含有使能端,刚开始实验时混淆。

解决方案: Controller 模块使用的流水线寄存器为 floprc,即含有 rst 和 clear 信号的触发器,因为此处流水线寄存器的功能只是在时钟信号到来时将数据传入或接受输入的控制信号,随时有效,无需使能端;datapath 使用的流水线寄存器存储的是数据信号,只有在使能端有效的情况下才输入输出。

3.5 问题 5:0 号寄存器的读数据问题

问题描述:一开始写寄存器堆时忽略了 0 号寄存器为保留寄存器,用来存储 0。

解决方案:在读数据时增加判断是否为 0,如图 5 所示:

```
//读是组合逻辑
assign rd1 = (ra1 != 0) ? rf[ra1] : 0;  //0号寄存器留给0
assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
```

图 16: 保留寄存器

3.6 问题 6: 同一信号在流水寄存器中的输入输出问题

问题描述:在设计冒险模块时误认为 rsD=rsE,rtD=rtE,想将其合并为两个信号。

解决方案：由于是流水线设计，同一时刻的 rsD 和 rsE 对应先后两条指令的 rs 字段，其数值并不相同，不能合并。

3.7 问题 7: 控制冒险设计

问题描述：看指导书中提前判断分支解决 beq 后 2 3 条指令的控制冒险，便将 EqualID 放入 hazard，后来发现线路混乱。

解决方案：虽然提前判断分支是用来解决控制冒险的，但并不将其放入 hazard 模块，而是将其放入 datapath 中，但 hazard 中需要放入提前判断分支导致的数据冒险，用暂停流水线和数据前推解决。

3.8 问题 8:beq 指令跳转错误

问题描述：连接 datapath 时，判断需要分支时直接将当前 PC 值与 branch 字节地址相加。但实际上此时的 PC 已经执行了 PC+4，存在偏移。

解决方案：在计算 branch 分支地址时将 PC-4 再传入。

3.9 问题 9: 控制信号与当前指令的解码信号不同

问题描述：仿真时比较同一时刻的指令与解码信号，发现与指令并不匹配，怀疑传入信号出错。

解决方案：实际上并没有问题，因为当前指令是给 F 级流水线使用的，D、E、M、W 级的控制信号应该去寻找对应的 instD、instE、instM、instW，发现可以对应。

4 实验结果及分析

根据 testbench 对 top 模块进行仿真，得到结果如图，与预期一致。（图中开始的 X 为初始情况未产生结果，中间的某一时刻无写入操作，writedata 为 X）

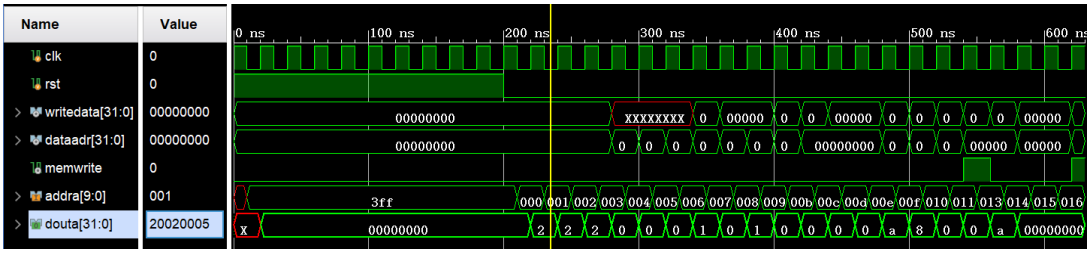

图 17: 仿真波形图

控制台输出、仿真文件中途成功跳出以及仿真过程各模块信号的波形图如下所示：

```
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.instr_ram.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will no
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.data_ram.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will not
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

图 18: 仿真控制台输出

```
always @(negedge clk) begin
    if(memwrite) begin
        /* code */
        if(dataadr === 84 & writedata === 7) begin
            /* code */
            $display("Simulation succeeded");
            $stop;
        end else if(dataadr !== 80) begin
            /* code */
            $display("Simulation Failed");
            $stop;
        end
    end
end
```

图 19: 仿真结束

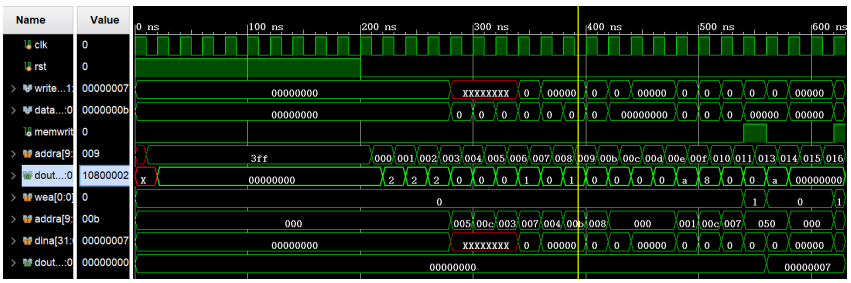

图 20: 仿真 memory 波形图

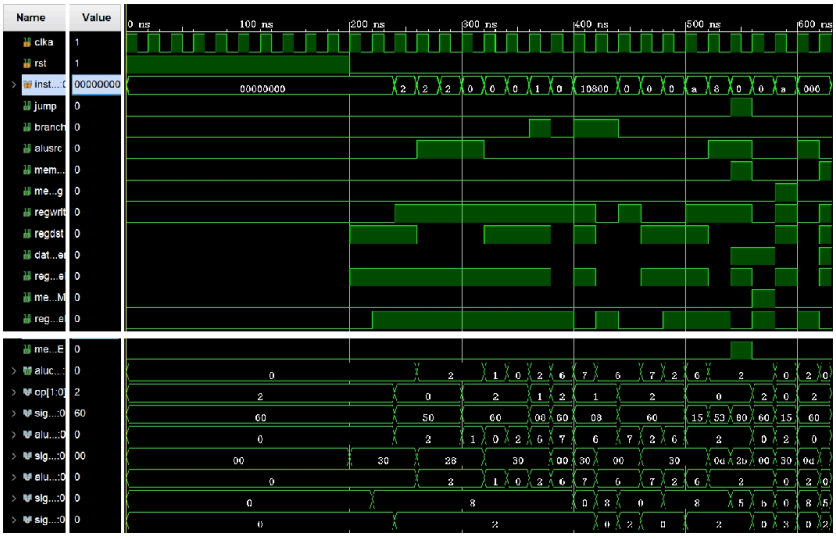

图 21: 仿真 controller 波形图

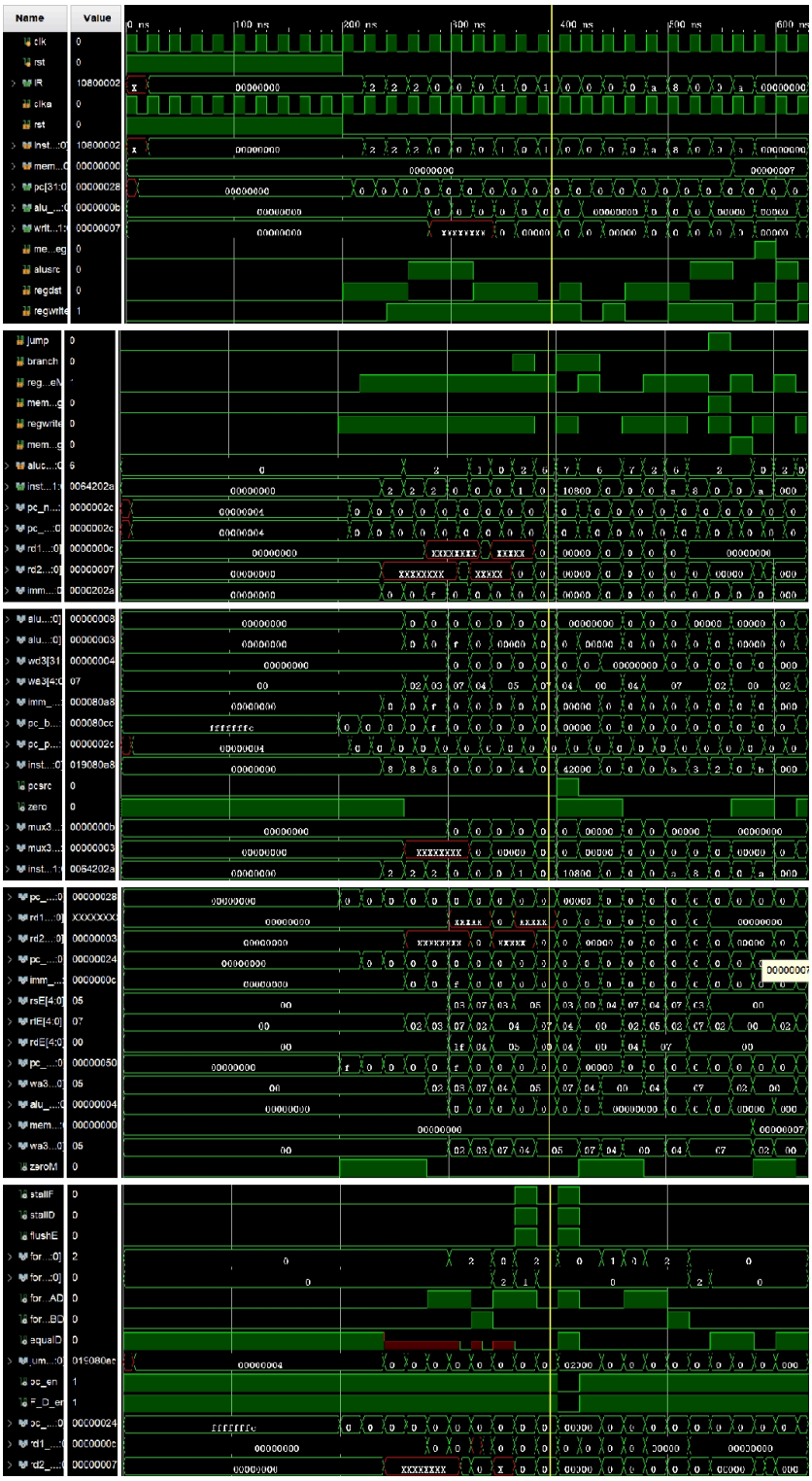

图 22: 仿真 datapath 波形图

A Datapath 代码

```
`timescale 1ns / 1ps

module datapath(
    input wire clka,rst,
    input wire [31:0] instr,
    input wire [31:0] mem_rdata, //data_ram中读出的数据
    output wire [31:0] pc,
    output wire [31:0] alu_resultM,
    output wire [31:0] writedataM,
    input wire memtoreg,
    input wire alusrc,
    input wire regdst,
    input wire regwrite,
    input wire jump,
    input wire branch,
    input wire regwriteM,
    input wire memtoregE,
    input wire regwriteE,
    input wire memtoregM,
    input wire [2:0] alucontrol,
    output wire [31:0] instrD_to_controller//从datapath中传出，由于instr必须为D
        阶段的才能使controller与其匹配。同时instrD受harzard控制，必须从datapath
        中传出。
    );
    wire [31:0] pc_next;          //pc+4后的下一位pc
    wire [31:0] pc_next_jump;     //选择pc+4?branch后，再次选择是否jump后的PC值
    wire [31:0] rd1D;             //regfile输出的rd1
    wire [31:0] rd2D;             //regfile输出的rd2
    wire [31:0] imm_extend;       //i型指令16位imm有符号扩展成32位
    wire [31:0] alu_result;       //alu计算结果
    wire [31:0] alu_srcB;         //alusec控制得到的alu_srcB
    wire [31:0] wd3;              //写regfile数据(ReadData ? ALUOut)
    wire [4:0] wa3;               //写regfile的寄存器号（rt ? rd)
    wire [31:0] imm_sl2;          //imm_extend左移2位（在branch指令下工作)
    wire [31:0] pc_branch;        //branch分支地址
    wire [31:0] pc_plus_4;        //pc+4
    wire [31:0] instr_jump_offset_sl2; //jump指令中低26位偏移地址左移两位后的结
        果（这里是32位，高四位后面舍弃)
    wire pcsrc;                   //判断pc 在branch指令下能否执行
    wire zero;                    //提前判断分支，比较branch指令中是否相等
    wire [31:0] mux3_A_result;
    wire [31:0] mux3_B_result;
    //F-D间信号
    wire [31:0] instrD;
    wire [31:0] pc_plus_4D;
    //D-E间信号
    wire [31:0] rd1E;
```

```
wire [31:0] rd2E;
wire [31:0] pc_plus_4E;
wire [31:0] imm_extendE;
wire [4:0] rsE;              //instr[25:21], 用于hazard模块
wire [4:0] rtE;              //filereg回写地址时 rt的地址 传入mux wa3
wire [4:0] rdE;              //filereg回写地址时 rd的地址 传入mux wa3
//E-M间信号
wire [31:0] pc_branchM;      //branch指令下pc跳转结果
wire [4:0] wa3M;             //选择rd还是rs写回数据的结果
//M-W间信号
wire [31:0] alu_resultW;     //回写的aluresult, 送到选择器去
wire [31:0] mem_rdataW;      //Data_ram中读出, 送到writeback阶段的选择器
wire [4:0] wa3W;             //选择rd还是rs写回数据的结果
wire zeroM;
//hazard传出的延迟与刷新信号
wire stallF,stallD,flushE;
//数据前推控制器
wire [1:0] forwordAE,forwordBE;
wire forwordAD,forwordBD;
wire equalD;
//pcSrc的判断
assign pcsrc = branch & equalD;
mux2 #(32) mux_pc_next(
    .a(pc_branch),           //branch的跳转
    .b(pc_plus_4),
    .s(pcsrc),               //连接pcSrc
    .y(pc_next)
    );
//j指令的低26位为字地址, 左移两位得到字节地址, 后面与PC高4位拼接
shift_2 sl2_pc_jump(
    .a(instrD),
    .y(instr_jump_offset_sl2)  //得到jump指令中偏移地址左移两位后的结果, 此
        处为32位, 高4位舍弃
    );
//j指令的跳转地址
wire [31:0] jump_addra;
assign jump_addra = {pc_plus_4[31:28],instr_jump_offset_sl2[27:0]} +4;
//PC跳转
mux2 #(32) mux_pc_jump(
    .a(jump_addra), //jump指令下的地址跳转
    .b(pc_next),    //PC+4
    .s(jump),       //jump判断信号
    .y(pc_next_jump)
    );
//判断在branch指令时是否要清空pc (流水线延迟导致branch下一条指令执行)
wire pc_en;
assign pc_en = ~(stallF & pcsrc);
//PC
pc pc_module(.clk(clka),.rst(rst),.en(1'b1),.din(pc_next_jump),.q(pc));
```

```
    //PC+4
    adder pc_plus_4_module(.a(pc),.b(32'h4),.y(pc_plus_4));
//流水线寄存器控制信号
    wire F_D_en;
    assign F_D_en = ~(stallD & pcsrc);
    //F-D数据传输
    flopenrc #(32) r1D(.clk(clka),.rst(rst),.en(F_D_en),.clear(1'b0),.d(instr)
        ,.q(instrD));
    flopenrc #(32) r2D(.clk(clka),.rst(rst),.en(F_D_en),.clear(1'b0),.d(
        pc_plus_4),.q(pc_plus_4D));

    assign instrD_to_controller = instrD;//从datapath中传出，由于instr必须为D阶
        段的才能使controller与其匹配。同时instrD受harzard控制，必须从datapath中
        传出。

    //instr低16位偏移地址扩展与左移
    sign_extend sign_extend(.a(instrD[15:0]),.y(imm_extend));
    shift_2 sl2(.a(imm_extend),.y(imm_sl2));
    //计算branch时PC已经+4了，要还原
    wire [31:0] pc_plus_4D_for_brach;
    assign pc_plus_4D_for_brach = pc_plus_4D - 4;
    adder pc_branch_module(.a(pc_plus_4D_for_brach),.b(imm_sl2),.y(pc_branch));
    //寄存器堆
    regfile regfile(
        .clk(clka),
        .we3(regwrite),                 //写使能
        .ra1(instrD[25:21]),            //读寄存器号1
        .ra2(instrD[20:16]),            //读寄存器号2
        .wa3(wa3W),                     //写地址
        .wd3(wd3),                      //写数据
        .rd1(rd1D),                     //读数据1
        .rd2(rd2D)                      //读数据2
        );

    wire [31:0] rd1_1sle_branch_A,rd2_1sle_branch_B;
    //rd1_decode_sle_branch_A
    mux2 #(32) mux_rd1_1sle_branch_A(
        .a(alu_result),                 //jump指令下的地址跳转
        .b(rd1D),
        .s(forwordAD),                  //jump
        .y(rd1_1sle_branch_A)
        );
    //rd2_decode_sle_branch_B
    mux2 #(32) mux_rd2_1sle_branch_B(.a(alu_result),.b(rd2D),.s(forwordBD),.y(
        rd2_1sle_branch_B));
    //提前判断分支减少控制冒险
    assign equalD = (rd1_1sle_branch_A == rd2_1sle_branch_B);
    //判断branch时是否要清空excute级流水线
    wire D_E_en;
```

```
assign D_E_en = flushE & pcsrc;

//D-E数据传输
flopenrc #(32) r1E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(rd1D),.
    q(rd1E));
flopenrc #(32) r2E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(rd2D),.
    q(rd2E));
flopenrc #(5) r3E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(instrD
    [20:16]),.q(rtE));
flopenrc #(5) r4E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(instrD
    [15:11]),.q(rdE));
flopenrc #(32) r5E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(
    pc_plus_4D),.q(pc_plus_4E));
flopenrc #(32) r6E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(
    imm_extend),.q(imm_extendE));
flopenrc #(5) r7E(.clk(clka),.rst(rst),.en(1'b1),.clear(D_E_en),.d(instrD
    [25:21]),.q(rsE));

//连接regfile的wa3,选择写入结果的地址是rt（lw）还是rd（r-type）
mux2 #(5) mux_wa3(
    .a(rdE),            //rt的地址
    .b(rtE),            //rd的地址
    .s(regdst),         //regdst
    .y(wa3)
    );

mux3 #(32) srcA_sel3(.d0(rd1E),.d1(wd3),.d2(alu_resultM),.s(forwordAE),.y(
    mux3_A_result));

mux3 #(32) srcB_sel3(.d0(rd2E),.d1(wd3),.d2(alu_resultM),.s(forwordBE),.y(
    mux3_B_result));

//alu_srcB
mux2 #(32) mux_alu_srcb(.a(imm_extendE),.b(mux3_B_result),.s(alusrc),.y(
    alu_srcB));
//ALU
alu alu(.a(mux3_A_result),.b(alu_srcB),.op(alucontrol),.result(alu_result)
    ,.zero(zero));

//E-M数据传输
flopenrc #(32) r1M(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(
    alu_result),.q(alu_resultM));
flopenrc #(1) r2M(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(zero),.q(
    zeroM));
flopenrc #(32) r3M(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(
    mux3_B_result),.q(writedataM));
flopenrc #(32) r4M(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(pc_branch
    ),.q(pc_branchM));
```

```
    flopenrc #(5) r5M(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(wa3),.q(
        wa3M));

    //M-W数据传输
    flopenrc #(32) r1W(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(
        alu_resultM),.q(alu_resultW));
    flopenrc #(32) r2W(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(mem_rdata
        ),.q(mem_rdataW));
    flopenrc #(5) r3W(.clk(clka),.rst(rst),.en(1'b1),.clear(1'b0),.d(wa3M),.q(
        wa3W));

    //wd3
    mux2 #(32) mux_wd3(.a(mem_rdataW),.b(alu_resultW),.s(memtoreg),.y(wd3));

    hazard hazard(.rst(rst), .rsD(instrD[25:21]), .rtD(instrD[20:16]), .rsE(rsE
        ),.rtE(rtE), .regwriteE(regwriteE), .regwriteM(regwriteM), .regwriteW(
        regwrite), .memtoregE(memtoregE), .memtoregM(memtoregM), .branchD(branch
        ), .writeregE(wa3), .writeregM(wa3M), .writeregW(wa3W), .forwordAE(
        forwordAE), .forwordBE(forwordBE), .forwordAD(forwordAD), .forwordBD(
        forwordBD), .stallF(stallF), .stallD(stallD), .flushE(flushE));

endmodule
```

B Hazard 代码

```
`timescale 1ns / 1ps
/*冒险解决：（数据冒险）数据前推+流水线暂停、（控制冒险）提前判断分支导致的数据
    前推、流水线暂停
    其中流水线暂停时需要清空下一级流水线。提前判断分支放在datapath里实现，并不
        在hazard中实现
*/
module hazard(
    input wire rst,
    input wire [4:0] rsD,       //instr2[25:21]（同一时刻rsD和rsE对应先后两条指
        令的rs字段，并不相同）
    input wire [4:0] rtD,       //instr2[20:15]
    input wire [4:0] rsE,       //instr1[25:21]
    input wire [4:0] rtE,       //instr1[20:15]
    input wire regwriteE,       //寄存器堆的写使能信号（E、M、W表示excute、
        memory和writeback阶段）
    input wire regwriteM,
    input wire regwriteW,
    input wire memtoregE,       //判断写回寄存器堆的数据是sw的ReadData(0)还是R
        的ALUOut(1)
    input wire memtoregM,
    input wire branchD,         //提前判断分支
    input wire [4:0] writeregE, //寄存器堆的写地址，连接wa3W
```

```
    input wire [4:0] writeregM,
    input wire [4:0] writeregW,
    output [1:0] forwordAE,      //在excute阶段控制mux3选择SrcA（数据冒险）
    output [1:0] forwordBE,      //在excute阶段控制mux3选择SrcB
    output [1:0] forwordAD,      //在decode阶段控制二选一选择regfile rd1出来的数
        据（控制冒险下的数据冒险）
    output [1:0] forwordBD,      //在decode阶段控制二选一选择regfile rd2出来的数
        据
    output reg stallF,           //instr fetch级暂停
    output reg stallD,           //decoder暂停
    output reg flushE            //excute读到的流水线需要清空
    );
//数据前推解决R指令和前两条lw指令的数据冒险
    /*ALU端口SrcAE的数据可能来自：（注意判断reE!=0，否则读保留寄存器直接输出）
        寄存器堆（无冒险情况下）：forwordAE=00、SrcAE=RsD
        数据存储器(lw的数据冒险)：forwordAE=01、SrcAE=ResultW（lw指令写回寄存器
            堆在MEM阶段，其后第二条指令如需要该数据会受影响）
        ALUOut（ALU运算的数据冒险）：forwordAE=10、SrcAE=ALUOutM（R型指令写回寄
            存器堆在WB阶段，其后一条指令如需要该数据都会受影响）
    */
    assign forwordAE = ((rsE != 5'b0) & (rsE == writeregM) & regwriteM) ? 2'b10
        :    //前一条指令是R型，直接将ALUOut传回
                       ((rsE != 5'b0) & (rsE == writeregW) & regwriteW) ? 2'b01
                           :    //前两条指令是lw
                        2'b00;
    assign forwordBE = ((rtE != 5'b0) & (rtE == writeregM) & regwriteM) ? 2'b10
        :    //SrcBE同SrcAE
                       ((rtE != 5'b0) & (rtE == writeregW) & regwriteW) ? 2'b01
                           :
                        2'b00;
//提前判断分支解决beq后2、3条指令的控制冒险时出现的数据冒险
    assign forwordAD = ((rsD != 5'b0) & (rsD == writeregE) & regwriteE);
                    //前一条指令要写回寄存器堆且该数据被beq指令所用
    assign forwordBD = ((rtD != 5'b0) & (rtD == writeregE) & regwriteE);
//流水线暂停解决lw后一条指令需要用寄存器堆数据带来的数据冒险、beq需要用前一条指
    令写回寄存器堆的数据
    /*lw指令写入寄存器堆的地址为rt，因此下一条指令若要用到rt则需要暂停，即rsD==
        rtE或rtD==rtE
      beq指令需要rs、rt号寄存器的数据，若上一条指令要写到该寄存器，则需要暂停流
          水线
    */
    wire lwstall,branch_stall;       //流水线暂停信号
    assign lwstall = (((rsD == rtE) | (rtD == rtE)) & memtoregE);    //判断前一
        条指令需要对寄存器堆写入(memturegE)并且写入地址rtE与被当前指令用到
    assign branch_stall=(branchD&regwriteE&((writeregE==rsD)|(writeregE==rtD)))
          //当前指令为branch、上一条指令要写寄存器堆且写的数据当前要用
                    |(branchD&memtoregM&((writeregM==rsD)|(writeregM==rtD)));
           //当前指令为branch、上2条指令要写寄存器堆且写的数据当前要用
    always @(*)begin
```

```
        stallF = rst? 1'b0 : (lwstall | branch_stall);  //若被重置则全部清零
        stallD = rst? 1'b0 : (lwstall | branch_stall);  //lw带来的数据冒险或提
            前判断分支带来的数据冒险都可以暂停流水线
        flushE = rst? 1'b0 : (lwstall | branch_stall);  //lw/beq下一条已经执行
            的需要清空
    end

endmodule
```

C Controller 代码

```
`timescale 1ns / 1ps
/*Controller模块：解码部分原理同实验二，此处输出控制信号时不能直接输出8bit sigs
    ，而是单独输出。
    本模块不仅负责解码，还需要操控每一级流水线和流水线寄存器之间的数据进出。下
        为各信号输出位置：
    jump、branch：Main Decoder后
    alucontrol、alusrc、regdst、regwriteE、memtoregE：流水线寄存器DE后
    memwrite、data_ram_ena、memtoregM、regwriteM：流水寄存器EM后
    regwrite、memtoreg：流水寄存器MW后
其中regwriteE,memtoregM,regwriteM,memtoregE均为传入datapath中的hazard模块，处理
    冒险情况
*/
module controller(
    input clka,rst,
    input wire [31:0] instr,
    output wire jump,branch,alusrc,memwrite,memetoreg,regwrite,regdst,
        data_ram_ena,regwriteE,memtoregM,
    output wire regwriteM,  //regwriteE,memtoregM,regwriteM,memtoregE传入
        datapath中的hazard需要
    output wire memtoregE,
    output wire [2:0] alucontrol
    );
//根据instr[31:26]和instr[5:0]解码
    wire [1:0] aluop;       //Main Decode输出的aluop信号，传入ALU Decoder
    wire [7:0] sigsD;       //Main Decode输出的8bit控制信号
    //main_dec 实例化
    main_dec Main_Decoder(.op(instr[31:26]),.sigs(sigsD),.aluop(aluop));
    wire [2:0] alucontrolD; //ALU Decoder输出的ALU控制信号，传入流水线寄存器
    //alu_dec 实例化
    alu_dec ALU_Control(.funct(instr[5:0]),.op(aluop),.alucontrol(alucontrolD))
        ;
    assign jump = sigsD[7]; //jump和branch信号不用继续传输，直接传给下一条指令
        以减少控制冒险
    assign branch = sigsD[3];
```

```
//流水线寄存器DE间的数据进出：{regwrite,regdst,alusrc,memwrite,memetoreg,
    data_ram_ena}和ALUControlD
    wire [5:0] sigsE;       //{regwrite,regdst,alusrc,memwrite,memetoreg,
        data_ram_ena}
    wire [2:0] alucontrolE; //从流水线寄存器DE读出的ALU控制信号
    floprc #(6) r1E(.clk(clka),.rst(rst),.clear(1'b0),.d({sigsD[6:4],sigsD
        [2:0]}),.q(sigsE));
    floprc #(3) r2E(.clk(clka),.rst(rst),.clear(1'b0),.d(alucontrolD),.q(
        alucontrolE));
    assign regdst = sigsE[4];
    assign alusrc = sigsE[3];
    assign alucontrol = alucontrolE;
    assign memtoregE = sigsE[1];
    assign regwriteE = sigsE[5];

//流水线寄存器EM间的数据进出：{regwrite,memwrite,memetoreg,data_ram_ena}
    wire [3:0] sigsM;
    floprc #(4) r1M(.clk(clka),.rst(rst),.clear(1'b0),.d({sigsE[5],sigsE[2:0]})
        ,.q(sigsM));
    assign memwrite = sigsM[2];
    assign data_ram_ena = sigsM[0];
    assign regwriteM = sigsM[3];  //传入datapath中的hazard需要
    assign memtoregM = sigsM[1]; //传入datapath中的hazard需要

//流水线寄存器MW间的数据进出：{regwrite,memwrite,memetoreg}
    wire [1:0] sigsW;
    floprc #(2) r1W(.clk(clka),.rst(rst),.clear(1'b0),.d({sigsM[3],sigsM[1]}),.
        q(sigsW));
    assign regwrite = sigsW[1];
    assign memetoreg = sigsW[0];

endmodule
```