《计算机组成原理》实验报告

年级、专业、班级	2021 级计算机科学与技术(卓越)01 班	**姓名**	韩昊辰
实验题目	实验四简单五级流水线 CPU		
实验时间	2023 年 5 月 22 日	**实验地点**	A 主 404
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: 2023 年 5 月 22 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

冒险处理模块的功能是解决数据冒险和控制冒险。

数据冒险在五级流水线 CPU 中体现在寄存器“先写后读”，某指令 D 阶段需要读出当前正在执行 E 阶段指令的 ALU 计算结果，或当前正在执行 M 阶段的 Data Memory 读出的数据。可以通过数据前推解决，冒险模块中 ForwardAE 和 ForwardBE 会判断当前输入 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 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]writeback 阶段。若相同，Foward 信号将其他指令结果通过多路选择器输入到 ALU 中。

数据前推可以解决 R 型指令上一个指令是 R 型或上第二个指令是 lw 的数据冒险，但若 lw 指令后紧跟着有数据相关的 R 型指令，R 指令在 E 阶段时 lw 指令还没有在 M 阶段从数据存储器中读出值(因为读值在时钟下降沿，而 R 指令 E 阶段 ALU 是组合逻辑)，因此需要对流水线暂停一周期。

暂停流水线通过判断前一条指令需要对寄存器堆写入 (memtoregE=1) 并且当前写入地址 rtE 被当前指令用 (rsD==rtE 或 rtD==rtE)，输出暂停信号作用于 F-D，D-E 寄存器，暂停 F，D 级，并刷新 E 级数据。

控制冒险是分支指令引起的，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，非常浪费资源和时间。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,此操作能够减少两条指令的执行。

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 d 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。

最后,冒险处理模块和 dapapath 接口对应电路图如图所示。

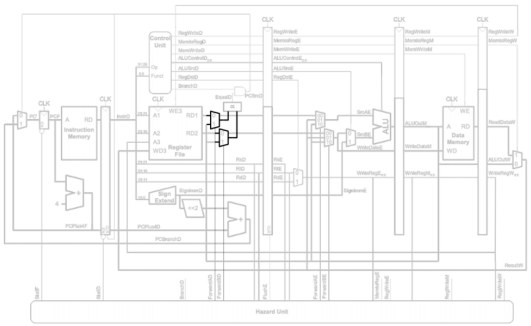

图 1: 冒险模块解决数据冒险电路图

2.1.2 接口定义

3 实验过程记录

3.1 最大问题

本实验历时三天，遇到了种种坎坷，尤其是频频出现的不符合预期的高阻和未知信号，但一一克服，发现最大的问题还是**细心！细心！细心！**不同级别的执行过程需要对应不同级别的信号,例如在实验过程中将 pCBranch（跳转目标 PC）设置成 1 位信号,以为 PCBranch 在 E 级计算，writedata（写入数据寄存器数据）对应的是 M 级的信号，这经常搞错，还有模块接口没接，接错也是一直在发生,这些错误的 debug 和排查占据了大部分实验时间,这些错误在细节上不再赘述，只是得到的经验是,在设计 CPU 时,必须对各个信号的作用非常了解,整个通路,信号传递的过程,在连线,debug 过程中必须非常清晰,否则只能一个一个试错,这严重影响了效率。

3.2 问题 1:controller 输入的指令不是 D 级的问题

问题描述:controller 输出信号始终比 datapath 执行指令快一周期。

解决方法: 指令存储器输入 mips 的指令是 F 级，这个指令同时传入子模块 datapath 和 controller 中,但指令从 F 级到 D 级的触发器处于 datapath 模块中,conteroller 没有相应触发器,因此需要将 datapath 的 D 级指令输出到 controller 中。

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
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

3.3 问题 2:仿真查看子模块信号

问题描述:仿真时想看子模块信号,将其接到顶层模块查看波形,十分麻烦。

解决方法: 直接利用 vivado 提供的功能, 在仿真功能框-scope 选择想查看信号的子模块, scope-object 将信号拖入仿真栏,保存仿真配置。然后 Add simulation 文件,将保存的配置加载进来,重新仿真,就可以看到子模块的波形了。

4 实验结果及分析

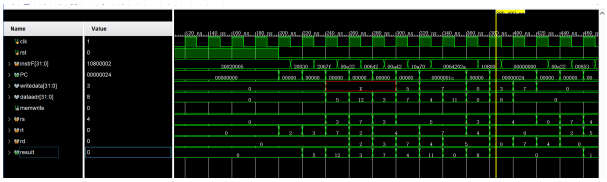

图 2: 仿真波形图(前半段)

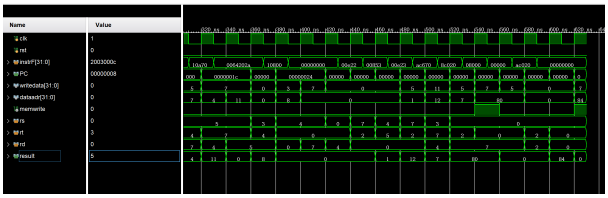

图 3: 仿真波形图(后半段)

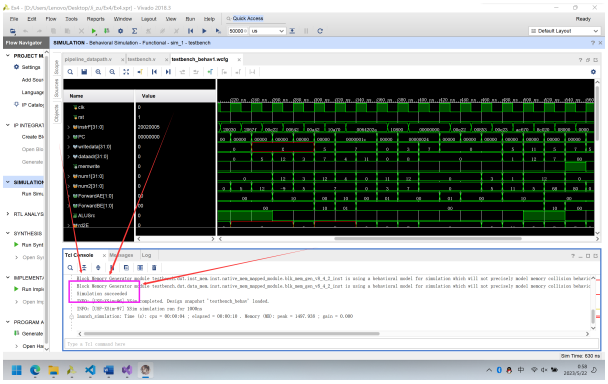

图 4: 控制台输出

A Controller 代码

```
module controller(
    input wire clka, rst,
    input wire [5:0] op,
    input wire [5:0] funct,
    output wire [2:0] ALUControl,
    output wire RegDst, Branch, MemtoReg, MemWrite, ALUSrc, RegWrite, Jump,
```

```
    output wire RegWriteM, RegWriteW, MemtoRegE, RegWriteE, MemtoRegM,

    //
    output wire ALUSrcD_sim,
    output wire [5:0] sigsD_sim, sigsE_sim
    );

assign ALUSrcD_sim = ALUSrcD, sigsD_sim = sigsD, sigsE_sim = sigsE;
//
wire [1:0] ALUOp;
// D
wire RegDstD, BranchD, MemtoRegD, MemWriteD, ALUSrcD, RegWriteD;
wire [2:0] ALUControlD;
main_decoder M_dec(
    .op(op),
    .ALUOp(ALUOp),
    .RegDst(RegDstD),
    .Branch(BranchD),
    .MemtoReg(MemtoRegD),
    .MemWrite(MemWriteD),
    .ALUSrc(ALUSrcD),
    .RegWrite(RegWriteD),
    .Jump(Jump)
    );
alu_decoder A_dec(
    .ALUOp(ALUOp),
    .funct(funct),
    .ALUControl(ALUControlD)
);
assign Branch = BranchD;

// D-E
wire [5:0] sigsD, sigsE;     // RegDstD, BranchD, MemtoRegD, MemWriteD, ALUSrcD,
    RegWriteD;
wire [2:0] ALUControlE;
assign sigsD = {RegDstD, BranchD, MemtoRegD,MemWriteD, ALUSrcD, RegWriteD};
floprc  #(6) rDE1(.clk(clka), .rst(rst), .clear(0), .en(1),  .din(sigsD), .q(sigsE)
    );
floprc  #(3) rDE2(.clk(clka), .rst(rst), .clear(0), .en(1),  .din(ALUControlD), .q(
    ALUControlE));

// E
assign ALUControl = ALUControlE;
assign MemtoRegE = sigsE[3];
assign ALUSrc = sigsE[1];
assign RegDst = sigsE[5];
assign RegWriteE = sigsE[0];
// E-M
wire [3:0] sigsM;    // {BranchD, MemtoRegD, MemWriteD, RegWriteD;}
```

```
floprc  #(4) rEM1(.clk(clka), .rst(rst), .clear(0), .en(1),  .din({sigsE[4:2],sigsE
    [0]}), .q(sigsM));

// M
assign MemWrite = sigsM[1];
assign RegWriteM = sigsM[0];
assign MemtoRegM = sigsM[2];
// M-W
wire [1:0] sigsW;  // {MemtoRegD,RegWriteD}
floprc  #(2) rMW1(.clk(clka), .rst(rst), .clear(0), .en(1),  .din({sigsM[2],sigsM
    [0]}), .q(sigsW));

// W
assign RegWrite = sigsW[0];
assign RegWriteW = sigsW[0];
assign MemtoReg = sigsW[1];

endmodule
```

B Hazard 代码

```
module Hazard_Unit(
    input wire [4:0] rsD, rtD, rsE, rtE, WriteRegM, WriteRegW, WriteRegE,
    input wire RegWriteM, RegWriteW, MemtoRegE,
    input wire BranchD, RegWriteE, MemtoRegM,
    output wire [1:0] ForwardAE, ForwardBE,
    output wire StallF, StallD, FlushE, ForwardAD, ForwardBD

    );

// 判断当前输入ALU的地址是否和其他指令（正在M, W阶段）在此时执行的阶段要写入寄存器
    堆的地址相同。
// 如果相同，就需要将其他指令结果直接通过多路选择器输入到ALU中
assign ForwardAE = ((rsE != 5'b0) & (rsE == WriteRegM) & RegWriteM)? 2'b10 :
                   ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)? 2'b01 : 2'b00;


assign ForwardBE = ((rtE != 0) & (rtE == WriteRegM) & RegWriteM)? 2'b10 :
                   ((rtE != 0) & (rtE == WriteRegW) & RegWriteW)? 2'b01 : 2'b00;

// lw流水线暂停
wire lwstall;
assign lwstall = (((rsD == rtE) | (rtD == rtE)) & MemtoRegE);

// 解决控制冒险
assign ForwardAD = ((rsD != 0) & (rsD == WriteRegM) & RegWriteM);
assign ForwardBD = ((rtD != 0) & (rtD == WriteRegM) & RegWriteM);
```

```
wire branchstall;
assign branchstall = (BranchD & RegWriteE & (WriteRegE == rsD | WriteRegE == rtD))
    |
                         (BranchD & MemtoRegM & (WriteRegM == rsD | WriteRegM == rtD
                             ));

assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FlushE = lwstall | branchstall;
endmodule
```

C Datapath 代码

```
module pipeline_datapath(
    input wire clka, rst,
    input wire memtoreg,alusrc,regdst,regwrite,jump,branch,
    input wire [31:0] instr, mem_read_data,
    input wire [2:0] alucontrol,
    input wire RegWriteM, RegWriteW, MemtoRegE, RegWriteE, MemtoRegM,
    output wire overflow,
    output wire [31:0] pc, ALUResult, writedataM, ALUResultM,//pc(F), ALUResult(E),
        writedatda(E)
    output wire [31:0] instrD,

    //用于debug
    output wire [31:0] alu_srca_sim, alu_srcb_sim, SignImmE_sim, WriteRegM_sim,
        WriteRegE_sim,
    output wire [1:0] ForwardAE_sim,
    output wire [4:0] rtE_sim, rsE_sim

    );
assign alu_srca_sim = alu_srca, alu_srcb_sim = alu_srcb, SignImmE_sim = SignImmE,
    WriteRegM_sim = wa3M;
assign ForwardAE_sim = ForwardAE;
assign rtE_sim = rtE, WriteRegE_sim = wa3, rsE_sim = rsE;
////////
wire StallF, StallD, FlushE;
wire [1:0] ForwardAE, ForwardBE;
wire ForwardAD, ForwardBD;
wire [4:0] wa3M;
wire [4:0] wa3W;

wire [4:0] rtE, rsE;
wire [31:0] rd1, rd2;    // read_data
wire [31:0] SignImm, writedata;
wire [31:0] PCBranch, PCBranchM;
```

```
wire [31:0] SignImm_sl2, Jump_adr;
wire [31:0] pc_add_4, alu_srcb, pc_next, pc_next_tmp, wd3;
wire [4:0] wa3;
wire pcsrc;
wire [31:0] pc_add_4D;
wire [4:0] rtD, rdD, rsD, rdE;
wire [31:0] SignImmE, pc_add_4E;
wire [31:0] alu_srca, alu_srcb_tmp, rd1E, rd2E, wd3W;
// M信号
wire zeroM;

// wire [4:0] wa3M;
// W信号
wire [31:0] ALUResultW, mem_read_dataW;

// F  ////////////////////////////////////////////////////////////////////
// pc
D_flip_flop PC(
    .clk(clka),
    .rst(rst),
    .en(~StallF),
    .din(pc_next),
    .q(pc)
    );

// pc + 4
Add_4 pc_add_4_adder(
    .a(pc),
    .res(pc_add_4)
    );

// F-D
floprc  #(32) r11(.clk(clka), .rst(rst), .clear(pcsrc), .en(~StallD),  .din(instr),
    .q(instrD));
floprc  #(32) r12(.clk(clka), .rst(rst), .clear(0), .en(~StallD),  .din(pc_add_4),
   .q(pc_add_4D));

// D  ////////////////////////////////////////////////////////////////////
wire [31:0] eq_input1, eq_input2;
// wire [4:0] rsD
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];
// reg_file
regfile regfile(
  .clk(~clka),
  .we3(regwrite),
  .ra1(instrD[25:21]), // base
    .ra2(instrD[20:16]), // sw, load from rt
```

```
    .wa3(wa3W), // lw, store to rt
  .wd3(wd3W),
  .rd1(rd1),
    .rd2(rd2)
    );

// mux2 for eq_input1 & eq_input2
wire EqualD;
Mux2 #(32) mux2_eq_input1(.a(ALUResultM), .b(rd1), .s(ForwardAD), .y(eq_input1));
Mux2 #(32) mux2_eq_input2(.a(ALUResultM), .b(rd2), .s(ForwardBD), .y(eq_input2));

assign EqualD = (eq_input1 == eq_input2);

// sign_extend
sign_extend sign_extend(
    .a(instrD[15:0]),
    .y(SignImm)
    );

// D-E
floprc  #(32) r21(.clk(clka), .rst(rst), .clear(FlushE), .en(1),  .din(rd1), .q(
    rd1E));
floprc  #(32) r22(.clk(clka), .rst(rst),.clear(FlushE), .en(1),  .din(rd2), .q(rd2E
    ));
floprc  #(5) r23(.clk(clka), .rst(rst), .clear(FlushE), .en(1),.din(rtD), .q(rtE));
floprc  #(5) r24(.clk(clka), .rst(rst), .clear(FlushE), .en(1),.din(rdD), .q(rdE));
floprc  #(32) r25(.clk(clka), .rst(rst),.clear(FlushE), .en(1),  .din(SignImm), .q(
    SignImmE));
floprc  #(32) r26(.clk(clka), .rst(rst),.clear(FlushE), .en(1),  .din(pc_add_4D), .
    q(pc_add_4E));
floprc  #(5) r27(.clk(clka), .rst(rst), .clear(FlushE), .en(1),.din(rsD), .q(rsE));

// E  ////////////////////////////////////////////////////////////////
assign writedata = alu_srcb_tmp;

// mux3 for ForwardAE & ForwardBE
mux3 #(32) mux3_ALU_num1(.a(rd1E), .b(wd3W), .c(ALUResultM), .s(ForwardAE), .y(
    alu_srca));
mux3 #(32) mux3_ALU_num2_tmp(.a(rd2E), .b(wd3W), .c(ALUResultM), .s(ForwardBE), .y(
    alu_srcb_tmp));

// ALU
ALU alu(
    .num1(alu_srca),
    .num2(alu_srcb),
    .op(alucontrol),     // 操作码
    .res(ALUResult),    // 计算结果
    .overflow(),
    .Zero(zero)
```

```
    );

// mux2 for alu_srcb
Mux2 mux_alu_srcB(
    .a(SignImmE),
    .b(alu_srcb_tmp),
    .s(alusrc),
    .y(alu_srcb)
    );

// mux for reg_file_a3
Mux2 #(5) mux_regfile_a3(
    .a(rdE),
    .b(rtE),
    .s(regdst),
    .y(wa3)
    );

// 得到PCBranch(D)
assign SignImm_sl2 = {SignImm[29:0],2'b00}; // shift left 2
assign PCBranch = SignImm_sl2 + pc_add_4D;

// E-M
floprc  #(1) r31(.clk(clka), .rst(rst), .clear(0), .en(1), .din(zero), .q(zeroM));
floprc  #(32) r32(.clk(clka), .rst(rst), .clear(0), .en(1), .din(ALUResult), .q(
    ALUResultM));
floprc  #(32) r33(.clk(clka), .rst(rst), .clear(0), .en(1), .din(writedata), .q(
    writedataM));
floprc  #(5) r34(.clk(clka), .rst(rst), .clear(0), .en(1), .din(wa3), .q(wa3M));
floprc  #(32) r35(.clk(clka), .rst(rst), .clear(0), .en(1), .din(PCBranch), .q(
    PCBranchM));

// M ////////////////////////////////////////////////////////////////

// M-W
floprc  #(32) r41(.clk(clka), .rst(rst), .clear(0), .en(1),  .din(ALUResultM), .q(
    ALUResultW));
floprc  #(32) r42(.clk(clka), .rst(rst), .clear(0), .en(1),  .din(mem_read_data), .
    q(mem_read_dataW));
floprc  #(32) r43(.clk(clka), .rst(rst), .clear(0), .en(1),  .din(wa3M), .q(wa3W));

// W ////////////////////////////////////////////////////////////////
// mux2 for reg_file_wd3
Mux2 mux_regfile_wd3(
    .a(mem_read_dataW),
    .b(ALUResultW),
    .s(memtoreg),
    .y(wd3W)
    );
```

```
// mux for pc_next_tmp (branch?)
assign pcsrc = EqualD & branch;
Mux2 mux_pc_next_tmp(
    .a(PCBranch),
    .b(pc_add_4),
    .s(pcsrc),
    .y(pc_next_tmp)
    );

// mux for pc_next (jump?)
assign Jump_adr[27:0] = {instrD[25:0],2'b00}; // shift left 2
assign Jump_adr[31:28] = pc_add_4[31:28];
Mux2 mux_pc_next(
    .a(Jump_adr),
    .b(pc_next_tmp),
    .s(jump),
    .y(pc_next)
);

// HAZARD
Hazard_Unit hazard(
    .rsE(rsE),
    .rtE(rtE),
    .rsD(rsD),
    .rtD(rtD),
    .WriteRegM(wa3M),
    .WriteRegW(wa3W),
    .RegWriteM(RegWriteM),
    .RegWriteW(RegWriteW),
    .ForwardAE(ForwardAE),
    .ForwardBE(ForwardBE),
    .MemtoRegE(MemtoRegE),
    .StallF(StallF),
    .StallD(StallD),
    .FlushE(FlushE),
    .BranchD(branch),
    .RegWriteE(RegWriteE),
    .MemtoRegM(MemtoRegM),
    .ForwardAD(ForwardAD),
    .ForwardBD(ForwardBD),
    .WriteRegE(wa3)
    );


endmodule
```