《计算机组成原理》实验报告

年级、专业、班级	2019 级计算机科学与技术(卓越)02 班	**姓名**	李燕琴
实验题目	实验四简单五级流水线 CPU		
实验时间	2021 年 06 月 09 日	**实验地点**	DS1410
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: 2021 年 6 月 8 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

通过数据前推、阻塞处理器当前阶段以等待结果计算或写回这两种方式，来解决五级流水线中同一寄存器在邻近周期内被多次使用导致的数据冒险与分支跳转指令导致的控制冒险这两类冒险问题。

2.1.2 接口定义

如表1所示。

3 实验过程记录

3.1 实验记录

本实验中，按照 MIPS 传统五级流水线阶段（取指 Fetch，译码 Decode，执行 Execute，访存 Memory，写回 Writeback）的原理，对 Controller 和 DataPath 增加触发器，分割不同流水阶段，并连接不同阶段传递的信号；针对同一寄存器在邻近周期被多次使用的数据冒险、分支跳转指令带来的控制冒险问题，设计冒险控制模块；最后通过实验给定的仿真程序，进行仿真验证。

表 1: 冒险模块接口定义

信号名	方向	位宽	功能描述
regwriteE	Input	1-bit	(Execute)是否写寄存器
regwriteM	Input	1-bit	(Memory)是否写寄存器
regwriteW	Input	1-bit	(Writeback)是否写寄存器
memtoRegE	Input	1-bit	(Execute) 寄存器回写数据选择 (ALU 计算的结果 or 存储器读取的数据)
memtoRegM	Input	1-bit	(Memory) 寄存器回写数据选择 (ALU 计算的结果 or 存储器读取的数据)
branchD	Input	1-bit	(Decode)是否为跳转指令
rsD	Input	5-bit	(Decode)第一个源操作数寄存器地址
rtD	Input	5-bit	(Decode)第二个源操作数寄存器地址
rsE	Input	5-bit	(Execute)第一个源操作数寄存器地址
rtE	Input	5-bit	(Execute)第二个源操作数寄存器地址
reg_waddrM	Input	5-bit	(Memory)写回数据的寄存器地址
reg_waddrW	Input	5-bit	(Writeback)写回数据的寄存器地址
reg_waddrE	Input	5-bit	(Execute)写回数据的寄存器地址
stallF	Output	1-bit	是否阻塞 Fetch 阶段
stallD	Output	1-bit	是否阻塞 Decode 阶段
flushE	Output	1-bit	是否清空 Execute 阶段
forwardAD	Output	1-bit	是否前推 Decode 阶段的 A 操作数
forwardBD	Output	1-bit	是否前推 Decode 阶段的 B 操作数
forwardAE	Output	2-bit	是否前推 Execute 阶段的 A 操作数
forwardBE	Output	2-bit	是否前推 Execute 阶段的 B 操作数

3.2 问题记录

问题描述:当一个数据在某时钟上升沿被写回寄存器或存储器时,该时钟上升沿其他地方是获取不到这个正在被写回的数据,此时如果需要读取该数据,则会出现读“XXXXXXXX”的错误。

分析解决:该问题,源自于冒险控制阶段:指导书中,对 forwardAD 和 forwardBD 的赋值定义,没有考虑到 writeback 阶段,故会出现读“XXXXXXXX”错误。这种情况下,如果不考虑 writeback 阶段可能存在的数据冒险,则需要将预写回寄存器或存储器的数据提前写回。而限于流水线分割的触发器均在时钟上升沿触发,故解决上述问题的方案一般是将 regfile 和 data_ram 中的时钟换成 ~clk,即在时钟下降沿写回数据。注意!!! 这里更改时钟沿触发时既要包括 regfile,也要包括 data_ram(ps 就因为忘记了更改 regfile 的时钟沿信号,结果就花了自己很长一段时间来 debug!)

4 实验结果及分析

仿真结果如图1所示,控制台结果输出如图2所示(**Simulation succeeded**)。通过对仿真图中每条指令 instrF[31:0] 对应的数据结果 alu_resultE[31:0] 进行比对分析,结果均符合预期,实验完结撒花!

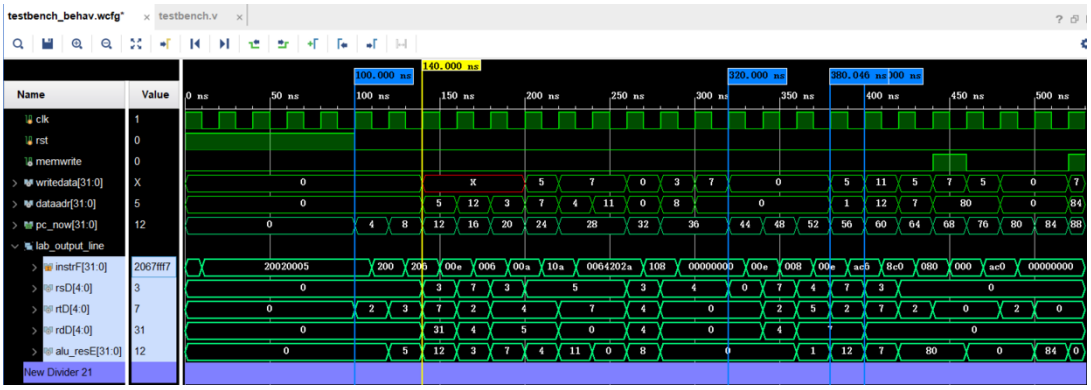

图 1: 仿真结果图

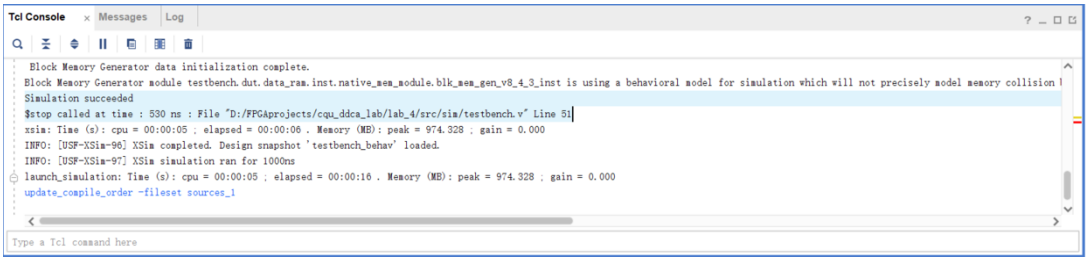

图 2: 控制台输出结果

A Datapath 代码

```
‘timescale 1ns / 1ps
module datapath (
    input wire clk,rst,
    input wire regwriteW,regdstE,alusrcE,branchD,memWriteM,memtoRegW,jumpD,
    input wire [2:0]alucontrolE,
    // 数据冒险添加信号
    input wire regwriteE,regwriteM,memtoRegE,memtoRegM,
    input wire [31:0]instrF,data_ram_rdataM,
    output wire [31:0]instrD,pc_now,data_ram_waddr,data_ram_wdataM
);

// ==================== 变量定义区，避免重复定义，还是集中在一起吧
    ========================
wire pcsrcD,clear,ena,equalD; // wire zero ==> branch跳转控制（已经升级到*控制
    冒险*）
wire stallF,stallD,flushD,flushE,forwardAD,forwardBD;
```

```
wire [1:0]forwardAE,forwardBE;
wire [4:0]rtD,rdD,rsD,rtE,rdE,rsE;
wire [4:0]reg_waddrE,reg_waddrM,reg_waddrW;
wire [31:0]pc_plus4F,pc_plus4D,pc_branchD,pc_next,pc_next_jump;
wire [31:0]rd1D,rd2D,rd1E,rd2E,wd3W,rd1D_branch,rd2D_branch,sel_rd1E,sel_rd2E;
wire [31:0]instrD_sl2,sign_immD,sign_immE,sign_immD_sl2;
wire [31:0]srcB,alu_resE,alu_resM,alu_resW,data_ram_rdataW;

assign clear = 1'b0;
assign ena = 1'b1;
assign flushD = pcsrcD | jumpD;
// ====================================== Fetch
    ======================================
mux2 mux2_branch(.a(pc_plus4F),.b(pc_branchD),.sel(pcsrcD),.y(pc_next)); // 注
    意，这里是PC_next是沿用的pc_plus4F

mux2 mux2_jump(
    .a(pc_next),
    .b({pc_plus4D[31:28],instrD_sl2[27:0]}), // 注意，这里是D阶段执行的
        pc_plus4D
    .sel(jumpD),
    .y(pc_next_jump)
);

pc pc(
    .clk(clk),
    .rst(rst),
    .ena(~stallF),
    .din(pc_next_jump),
    .dout(pc_now)
);

adder adder(
    .a(pc_now),
    .b(32'd4),
    .y(pc_plus4F)
);

// ====================================== Decoder
    ======================================
// 注意：这里要不要flushD都没问题，因为跳转指令后面都是一个nop，所以没关系
flopenrc DFF_instrD(clk,rst,flushD,~stallD,instrF,instrD);
flopenrc DFF_pc_plus4D(clk,rst,clear,~stallD,pc_plus4F,pc_plus4D);

assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];

regfile regfile(
```

```
  .clk(clk),
  .we3(regwriteW),
  .ra1(instrD[25:21]),
    .ra2(instrD[20:16]),
    .wa3(reg_waddrW), // 前in后out
  .wd3(wd3W),
  .rd1(rd1D),
    .rd2(rd2D)
);

// jump指令拓展
sl2 sl2_instr(
    .a(instrD),
    .y(instrD_sl2)
);

signext sign_extend(
    .a(instrD[15:0]), // input wire [15:0]a
    .y(sign_immD) // output wire [31:0]y
);

sl2 sl2_signImm(
    .a(sign_immD),
    .y(sign_immD_sl2)
);

adder adder_branch(
    .a(sign_immD_sl2),
    .b(pc_plus4D),
    .y(pc_branchD)
);

// ******************* 控制冒险 *****************
// 在 regfile 输出后添加一个判断相等的模块，即可提前判断 beq，以将分支指令提前
    到Decode阶段
mux2 mux2_forwardAD(rd1D,alu_resM,forwardAD,rd1D_branch);
mux2 mux2_forwardBD(rd2D,alu_resM,forwardBD,rd2D_branch);
// assign equalD = (rd1D_branch == rd2D_branch);
assign equalD = (rd1D_branch == rd2D_branch) ? 1:0;
assign pcsrcD = equalD & branchD;

// ===================================== Execute
    =====================================

flopenrc DFF_rd1E(clk,rst,flushE,ena,rd1D,rd1E);
flopenrc DFF_rd2E(clk,rst,flushE,ena,rd2D,rd2E);
flopenrc DFF_sign_immE(clk,rst,flushE,ena,sign_immD,sign_immE);
flopenrc #(5) DFF_rtE(clk,rst,flushE,ena,rtD,rtE);
flopenrc #(5) DFF_rdE(clk,rst,flushE,ena,rdD,rdE);
```

```
flopenrc #(5) DFF_rsE(clk,rst,flushE,ena,rsD,rsE);

mux2 #(5) mux2_regDst(.a(rtE),.b(rdE),.sel(regdstE),.y(reg_waddrE));

// ****************** 数据冒险 *****************
// 00原结果, 01写回结果_W, 10计算结果_M
mux3 #(32) mux3_forwardAE(rd1E,wd3W,alu_resM,forwardAE,sel_rd1E);
mux3 #(32) mux3_forwardBE(rd2E,wd3W,alu_resM,forwardBE,sel_rd2E);
mux2 mux2_aluSrc(.a(sel_rd2E),.b(sign_immE),.sel(alusrcE),.y(srcB));

alu alu(
    .a(sel_rd1E),
    .b(srcB),
    .f(alucontrolE),
    .y(alu_resE),
    .overflow(),
    .zero() // wire zero ==> branch跳转控制 (已经升级到*控制冒险*)
);

// ====================================== Memory
    ======================================
flopenrc DFF_alu_resM(clk,rst,clear,ena,alu_resE,alu_resM);
flopenrc DFF_data_ram_wdataM(clk,rst,clear,ena,sel_rd2E,data_ram_wdataM);
flopenrc #(5) DFF_reg_waddrM(clk,rst,clear,ena,reg_waddrE,reg_waddrM);
// flopenrc #(1) DFF_zeroM(clk,rst,clear,ena,zero,zeroM);  ==> 控制冒险, 已将分
    支指令提前到Decode阶段

assign data_ram_waddr = alu_resM;
// assign pcsrcM = zeroM & branchM;  ==> 控制冒险, 已将分支指令提前到Decode阶段

// ====================================== WriteBack
    ======================================
flopenrc DFF_alu_resW(clk,rst,clear,ena,alu_resM,alu_resW);
flopenrc DFF_data_ram_rdataW(clk,rst,clear,ena,data_ram_rdataM,data_ram_rdataW)
    ;
flopenrc #(5) DFF_reg_waddrW(clk,rst,clear,ena,reg_waddrM,reg_waddrW);

mux2 mux2_memtoReg(.a(alu_resW),.b(data_ram_rdataW),.sel(memtoRegW),.y(wd3W));

// ****************** 冒险信号总控制 *****************
hazard hazard(
    regwriteE,regwriteM,regwriteW,memtoRegE,memtoRegM,branchD,
    rsD,rtD,rsE,rtE,reg_waddrM,reg_waddrW,reg_waddrE,
    stallF,stallD,flushE,forwardAD,forwardBD,
    forwardAE, forwardBE
);

endmodule
```

B Hazard 代码

```
‘timescale 1ns/1ps
module hazard (
    input wire regwriteE,regwriteM,regwriteW,memtoRegE,memtoRegM,branchD,
    input wire [4:0]rsD,rtD,rsE,rtE,reg_waddrM,reg_waddrW,reg_waddrE,
    output wire stallF,stallD,flushE,forwardAD,forwardBD,
    output wire[1:0] forwardAE, forwardBE
);

    // 数据冒险
    assign forwardAE =  ((rsE != 5’b0) & (rsE == reg_waddrM) & regwriteM) ? 2’
        b10: // 前推计算结果
                        ((rsE != 5’b0) & (rsE == reg_waddrW) & regwriteW) ? 2’
                            b01: // 前推写回结果
                        2’b00; // 原结果
    assign forwardBE =  ((rtE != 5’b0) & (rtE == reg_waddrM) & regwriteM) ? 2’
        b10: // 前推计算结果
                        ((rtE != 5’b0) & (rtE == reg_waddrW) & regwriteW) ? 2’
                            b01: // 前推写回结果
                        2’b00; // 原结果

    // 控制冒险产生的写冲突
    // 0 原结果， 1 写回结果
    assign forwardAD = (rsD != 5’b0) & (rsD == reg_waddrM) & regwriteM;
    assign forwardBD = (rtD != 5’b0) & (rtD == reg_waddrM) & regwriteM;

    // 判断 decode 阶段 rs 或 rt 的地址是否是上一个lw 指令要写入的地址rtE;
    wire lwstall,branch_stall; // 指令阻塞
    assign lwstall = ((rsD == rtE) | (rtD == rtE)) & memtoRegE;
    assign branch_stall =   (branchD & regwriteE & ((rsD == reg_waddrE)|(rtD ==
         reg_waddrE))) | // 执行阶段阻塞，前面有写入的数据
                            (branchD & memtoRegM & ((rsD == reg_waddrM)|(rtD ==
                                reg_waddrM))); // 写回阶段阻塞

    assign stallF = lwstall | branch_stall;
    assign stallD = lwstall | branch_stall;
    assign flushE = lwstall | branch_stall;
endmodule
```

C Controller 代码

```
‘timescale 1ns/1ps
module controller (
    input wire clk,rst,
    input wire [31:0]instrD,
    output wire regwriteW,regdstE,alusrcE,branchD,memWriteM,memtoRegW,jumpD,
```

```
    // 数据冒险添加信号
    output wire regwriteE,regwriteM,memtoRegE,memtoRegM, // input wire
    output wire [2:0]alucontrolE
);

    wire [1:0]aluop;
    wire [6:0]signsD,signsE,signsM,signsW;
    wire [2:0]alucontrolD;
    wire ena;
    assign ena = 1'b1;

    // Decoder
    main_dec main_dec(
        .op(instrD[31:26]),
        .signs(signsD),
        .aluop(aluop)
    );

    alu_dec alu_dec(
        .aluop(aluop),
        .funct(instrD[5:0]),
        .alucontrol(alucontrolD)
    );

    // Execute
    flopenr #(3) dff2E(clk,rst,ena,alucontrolD,alucontrolE);
    flopenr #(7) dff1E(clk,rst,ena,signsD,signsE);
    flopenr #(7) dff1M(clk,rst,ena,signsE,signsM);
    flopenr #(7) dff1W(clk,rst,ena,signsM,signsW);

    // assign {regwrite,regdst,alusrc,branch,memWrite,memtoReg,aluop,jump} =
        temp;
    // signsD = {6regwrite,5regdst,4alusrc,3branch,2memWrite,1memtoReg,0jump}
    assign regwriteW = signsW[6];
    assign regwriteE = signsE[6];
    assign regwriteM = signsM[6];
    assign regdstE = signsE[5];
    assign alusrcE = signsE[4];
    assign branchD = signsD[3];
    assign memWriteM = signsM[2];
    assign memtoRegW = signsW[1];
    assign memtoRegE = signsE[1];
    assign memtoRegM = signsM[1];
    assign jumpD = signsD[0];

endmodule


module main_dec (
```

```
    input wire[5:0]op,
    // output wire regwrite,regdst,alusrc,branch,memWrite,memtoReg,jump,
    output wire [6:0]signs,
    output wire [1:0]aluop
);
    wire [8:0]temp;
    assign  temp =  (op==6'b000000) ? 9'b110000_10_0: // R-type 最后一位是jump
                    (op==6'b100011) ? 9'b101001_00_0: // lw
                    (op==6'b101011) ? 9'b001010_00_0: // sw
                    (op==6'b000100) ? 9'b000100_01_0: // beq
                    (op==6'b001000) ? 9'b101000_00_0: // addi
                    (op==6'b000010) ? 9'b000000_00_1: // j
                    9'b000000000; // 注意: 这里的X信号, 全部视为0
    // assign {regwrite,regdst,alusrc,branch,memWrite,memtoReg,aluop,jump} =
        temp;
    assign {signs[6:1],aluop,signs[0]} = temp;
endmodule


module alu_dec (
    input wire[1:0]aluop,
    input wire[5:0]funct,
    output wire[2:0]alucontrol
);
    assign alucontrol = (aluop==2'b00) ? 3'b010: // add
                        (aluop==2'b01) ? 3'b110: // sub
                        (aluop==2'b10) ?
                            (funct==6'b100000) ? 3'b010:
                            (funct==6'b100010) ? 3'b110:
                            (funct==6'b100100) ? 3'b000: // and
                            (funct==6'b100101) ? 3'b001: // or
                            (funct==6'b101010) ? 3'b111: // slt
                            3'b000:
                        3'b000; // default
endmodule
```