《计算机组成原理》实验报告

年级、专业、班级	2022级计算机科学与技术03班/01班/06班	**姓名**	叶旭航，解吴雪，李佳玲
实验题目	实验四简单五级流水线CPU		
实验时间	2024 年 4 月 30 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: 冯永

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: 2024年 5月 18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 数据通路模块

2.1.1 功能描述

该datapath模块实现了MIPS微处理器的数据通路，包括取指、译码、执行、访存和写回等五个阶段的功能。它能够根据指令执行算术逻辑操作、控制指令的分支跳转、处理数据的前递和冒险以及控制指令的流水线执行。

2.1.2 接口定义

2.1.3 逻辑控制

1. **写控制：**当写使能信号 `we3` 为高时，在时钟信号 `clk` 的上升沿，将输入数据 `wd3` 写入由写地址 `wa3` 指定的寄存器。
2. **读控制：**读操作是组合逻辑，不受时钟控制。当读地址 `ra1` 或 `ra2` 不为 0 时，分别从寄存器文件 `rf` 中读取对应地址的寄存器值输出到 `rd1` 或 `rd2`；如果读地址为 0，则输出 0，这是因为寄存器 0（`$zero`）在 MIPS 架构中是硬连线为 0 的寄存器。

2.2 触发器(flopenrc)模块

表 1: 数据通路接口定义

信号名	方向	宽度	含义
clk	输入	1	时钟信号
rst	输入	1	异步复位信号
pcF	输出	32	取指阶段的程序计数器输出
instrD	输入	32	译码阶段的指令输入
regwriteD	输入	1	译码阶段的寄存器写使能信号
memtoregD	输入	1	译码阶段的内存到寄存器选择信号
branchD	输入	1	译码阶段的分支控制信号
memwriteD	输入	1	译码阶段的内存写使能信号
jumpD	输入	1	译码阶段的跳转控制信号
regdstD	输入	1	译码阶段的寄存器目标选择信号
alusrcD	输入	1	译码阶段的 ALU 源选择信号
pcsrcD	输入	1	译码阶段的程序计数器源选择信号
alucontrolID	输入	3	译码阶段的 ALU 控制信号
equalID	输出	1	译码阶段的比较结果输出
op	输出	6	译码阶段的指令操作码输出
funct	输出	6	译码阶段的指令功能码输出
stallID	输出	1	译码阶段的流水线暂存信号
readdataM	输入	32	访存阶段的内存读数据输入
writedataM	输出	32	访存阶段的内存写数据输出
aluoutM	输出	32	访存阶段的 ALU 输出
memwriteM	输出	1	访存阶段的内存写使能信号

2.2.1 功能描述

一个具有异步复位和清除功能的寄存器，能够根据时钟信号、异步复位信号、清除信号和使能信号来更新其输出值。

2.2.2 接口定义

表 2: 触发器接口定义

信号名	方向	宽度	含义
clk	输入	1	时钟信号
rst	输入	1	异步复位信号
en	输入	1	使能信号
clear	输入	1	清除信号
d	输入	WIDTH-1:0	数据输入
q	输出	WIDTH-1:0	寄存器输出

2.2.3 逻辑控制

1. **复位逻辑：**当 `rst` 信号为高电平时，无论 `en` 和 `clear` 信号的状态如何，寄存器的输出 `q` 将被重置为 0。
2. **清除逻辑：**当 `clear` 信号为高电平时，无论 `en` 信号的状态如何，寄存器的输出 `q` 将被重置为 0。
3. **使能逻辑：**当 `en` 信号为高电平时，寄存器的输出 `q` 将更新为输入 `d` 的值。

优化逻辑：包括使用异步复位和清除信号来快速初始化寄存器状态，以及确保在时钟上升沿时执行更新操作，以避免亚稳态问题。

2.3 冒险处理模块

2.3.1 功能描述

用于检测和处理流水线中的数据冒险和控制冒险。

2.3.2 接口定义

2.3.3 逻辑控制

1. **数据前推逻辑：**

表 3: 冒险接口定义

信号名	方向	宽度	含义
rsE	输入	5	EX阶段寄存器读取源E
rtE	输入	5	EX阶段寄存器读取源E
writeregM	输入	5	MEM阶段写寄存器地址
writeregW	输入	5	WB阶段写寄存器地址
rsD	输入	5	DEC阶段寄存器读取源D
rtD	输入	5	DEC阶段寄存器读取源D
regwriteM	输入	1	MEM阶段寄存器写使能
regwriteW	输入	1	WB阶段寄存器写使能
memtoregE	输入	1	EX阶段内存到寄存器选择
forwardAE	输出	2	EX阶段ALU数据前推
forwardBE	输出	2	EX阶段ALU数据前推
stallF	输出	1	FETCH阶段流水线暂停
stallID	输出	1	DECODE阶段流水线暂停

- `forwardAE` 和 `forwardBE`：根据 EX 阶段的寄存器读取源和 MEM/WB 阶段的写寄存器地址，确定 ALU 数据的前推方向。
- `forwardAD` 和 `forwardBD`：根据 DEC 阶段的寄存器读取源和 EX 阶段的写寄存器地址，确定 ALU 数据的前推方向。

2. **流水线暂停逻辑：**
   - `stallF`：当 EX 阶段的读取源与 MEM/WB 阶段的写寄存器地址匹配时，暂停 FETCH 阶段的流水线。
   - `stallD`：当 DECODE 阶段的读取源与 EX 阶段的写寄存器地址匹配时，暂停 DECODE 阶段的流水线。
3. **清除流水线逻辑：**
   - `flushE`：当检测到数据冒险或控制冒险时，清除流水线，防止错误的数据流过。

优化逻辑：包括使用更复杂的检测机制来处理更多的冒险情况，以及确保在检测到冒险时及时清除流水线。

3 实验过程记录

记录实验的过程，完成了什么样的工作，存在的问题包括哪些，解决方案如何等。subsubsection名称自行设定。

4 实验结果及分析

4.1 仿真图

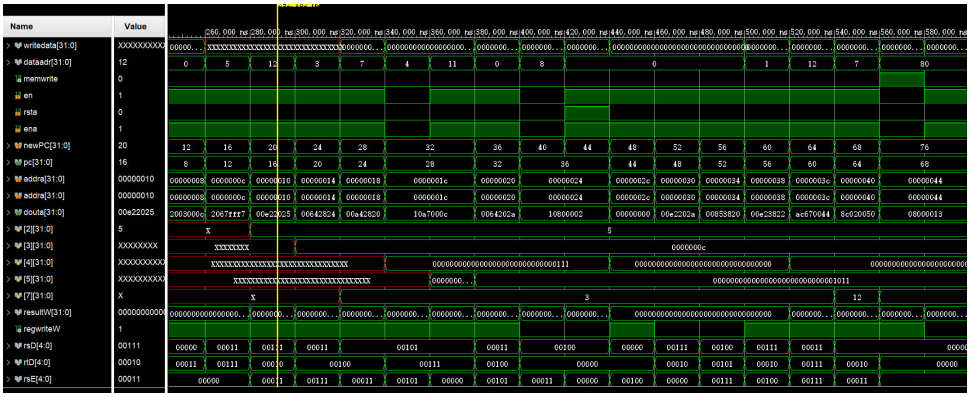

图 1: 仿真1

```
        rst <= 0;
    end

    always begin
        clk <= 1;
        #10;
        clk <= 0;
        #10;

    end

    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
endmodule
```

图 2: 仿真2

A Datapath代码

```
module datapath(
input clk, rst,
// FETCH
output [31:0] pcF,
input [31:0] instrD,

// DECODE
```

```
input regwriteD, memtoregD, branchD, memwriteD, jumpD, regdstD, alusrcD, pcsrcD
    ,
input [2:0] alucontrolD,
output equalD,
output wire [5:0] op, funct,
output stallD,

// EXECUTE
// no i/o signal

// MEMORY
input [31:0] readdataM,
output [31:0] writedataM, aluoutM,
output wire memwriteM

// WRITEBACK
// no i/o signal
);

    // FETCH
    wire [31:0] nextpc, pcplus4F, pcbranchD, pcjumpD, pctemp;
    wire stallF;

    // DECODE
    wire [31:0] pcplus4D;
    wire [31:0] rd1D, rd2D;
    wire [31:0] branchnum1D, branchnum2D;
    wire [4:0] rsD, rtD, rdD;
    wire [31:0] sign_extendD;
    wire forwardAD, forwardBD;

    // Execute
    wire regwriteE, memtoregE, memwriteE, regdstE, alusrcE;
    wire [2:0] alucontrolE;
    wire [31:0] srcAE, srcBE, aluoutE;
    wire [4:0] rsE, rtE, rdE;
    wire [31:0] sign_extendE;
    wire [31:0] rd1E, rd2E;
    wire [4:0] writeregE;
    wire [31:0] writedataE;
    wire [1:0] forwardAE, forwardBE;
    wire flushE;

    // MEMORY
    wire regwriteM, memtoregM;
    wire [4:0] writeregM;

    // WRITEBACK
    wire regwriteW, memtoregW;
```

```
wire [4:0] writeregW;
wire [31:0] readdataW, aluoutW, resultW;

// pc
assign pcjumpD = {pcplus4F[31:28], instrD[25:0], 2'b00};
pc pc_inst1(.clk(clk), .rst(rst), .en(~stallF), .newPC(nextpc), .pc(pcF));
mux2 sel_PCtemp(.option1(pcplus4F), .option2(pcbranchD), .select(pcsrcD), .
    data(pctemp));
mux2 sel_PCnext(.option1(pctemp), .option2(pcjumpD), .select(jumpD), .data(
    nextpc), .rst(rst));
adder addPC(.a(pcF), .b(32'd4), .y(pcplus4F));

// FETCH ----> DECODE
flopenrc #32 pcplus4FD_inst(.clk(clk), .rst(rst), .en(~stallD), .clear(
    pcsrcD), .d(pcplus4F), .q(pcplus4D));
assign op = instrD[31:26];
assign funct = instrD[5:0];
regfile rf_inst1(.clk(clk), .we3(regwriteW), .ra1(instrD[25:21]), .ra2(
    instrD[20:16]), .wa3(writeregW), .wd3(resultW), .rd1(rd1D), .rd2(rd2D));
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];
sign_extend extend(.sign(instrD[15:0]), .sign_extend(sign_extendD));
adder addbranchD(.a({sign_extendD[29:0], 2'b00}), .b(pcplus4D), .y(
    pcbranchD));
mux2 sel_brnum1(.option1(rd1D), .option2(aluoutM), .select(forwardAD), .
    data(branchnum1D));
mux2 sel_brnum2(.option1(rd2D), .option2(aluoutM), .select(forwardBD), .
    data(branchnum2D));
assign equalD = branchnum1D == branchnum2D ? 1'b1 : 1'b0;

// DECODE ----> EXECUTE
floprc #1 regwriteDE(.clk(clk), .rst(rst), .clear(flushE), .d(regwriteD), .
    q(regwriteE));
floprc #1 memtoregDE(.clk(clk), .rst(rst), .clear(flushE), .d(memtoregD), .
    q(memtoregE));
floprc #1 memwriteDE(.clk(clk), .rst(rst), .clear(flushE), .d(memwriteD), .
    q(memwriteE));
floprc #1 regdstDE(.clk(clk), .rst(rst), .clear(flushE), .d(regdstD), .q(
    regdstE));
floprc #1 alusrcDE(.clk(clk), .rst(rst), .clear(flushE), .d(alusrcD), .q(
    alusrcE));
floprc #3 alucontrolDE(.clk(clk), .rst(rst), .clear(flushE), .d(alucontrolD
    ), .q(alucontrolE));
floprc #5 rsDE(.clk(clk), .rst(rst), .clear(flushE), .d(rsD), .q(rsE));
floprc #5 rtDE(.clk(clk), .rst(rst), .clear(flushE), .d(rtD), .q(rtE));
floprc #5 rdDE(.clk(clk), .rst(rst), .clear(flushE), .d(rdD), .q(rdE));
floprc #32 sign_extendDE(.clk(clk), .rst(rst), .clear(flushE), .d(
    sign_extendD), .q(sign_extendE));
```

```
    floprc #32 rd1DE(.clk(clk), .rst(rst), .clear(flushE), .d(rd1D), .q(rd1E));
    floprc #32 rd2DE(.clk(clk), .rst(rst), .clear(flushE), .d(rd2D), .q(rd2E));
    mux3 srcAE_inst(.option1(rd1E), .option2(resultW), .option3(aluoutM), .
        select(forwardAE), .data(srcAE));
    mux3 srcBE_temp_inst(.option1(rd2E), .option2(resultW), .option3(aluoutM),
        .select(forwardBE), .data(writedataE));
    mux2 srcDE_inst(.option1(writedataE), .option2(sign_extendE), .select(
        alusrcE), .data(srcBE));
    mux2 #5 writeregE_sel(.option1(rtE), .option2(rdE), .select(regdstE), .data
        (writeregE));
    ALU mainalu(.num1(srcAE), .num2(srcBE), .op(alucontrolE), .ans(aluoutE));

    // EXECUTE ----> MEMORY
    flopr #1 regwriteEM(.clk(clk), .rst(rst), .d(regwriteE), .q(regwriteM));
    flopr #1 memtoregEM(.clk(clk), .rst(rst), .d(memtoregE), .q(memtoregM));
    flopr #1 memwriteEM(.clk(clk), .rst(rst), .d(memwriteE), .q(memwriteM));
    flopr #5 writeregEM(.clk(clk), .rst(rst), .d(writeregE), .q(writeregM));
    flopr #32 aluoutEM(.clk(clk), .rst(rst), .d(aluoutE), .q(aluoutM));
    flopr #32 writedataEM(.clk(clk), .rst(rst), .d(writedataE), .q(writedataM))
        ;

    // MEMORY ----> WRITEBACK
    flopr #1 regwriteMW(.clk(clk), .rst(rst), .d(regwriteM), .q(regwriteW));
    flopr #1 memtoregMW(.clk(clk), .rst(rst), .d(memtoregM), .q(memtoregW));
    flopr #5 writeregMW(.clk(clk), .rst(rst), .d(writeregM), .q(writeregW));
    flopr #32 readdataMW(.clk(clk), .rst(rst), .d(readdataM), .q(readdataW));
    flopr #32 aluoutMW(.clk(clk), .rst(rst), .d(aluoutM), .q(aluoutW));
    mux2 resultW_inst(.option1(aluoutW), .option2(readdataW), .select(memtoregW
        ), .data(resultW));

    // Hazard Control
    hazard hazard_control(.stallF(stallF), .stallD(stallD), .branchD(branchD),
    .forwardAD(forwardAD), .forwardBD(forwardBD), .rsD(rsD), .rtD(rtD), .flushE
        (flushE), .rsE(rsE), .rtE(rtE),
    .forwardAE(forwardAE), .forwardBE(forwardBE), .memtoregE(memtoregE), .
        regwriteE(regwriteE), .writeregE(writeregE),
    .writeregW(writeregW), .regwriteW(regwriteW), .memtoregM(memtoregM), .
        regwriteM(regwriteM), .writeregM(writeregM));
endmodule
```

B Hazard代码

```
module hazard(
    input [4:0]rsE, rtE, writeregM, writeregW, rsD, rtD,
    input regwriteM, regwriteW, memtoregE,
```

```
    output reg [1:0] forwardAE, forwardBE,
    output wire stallF, stallD, flushE,
    input [4:0] writeregE,
    input branchD, regwriteE, memtoregM,
    output wire [1:0] forwardAD, forwardBD
        );

    always @(*) begin
        forwardAE = 2'b00;
        forwardBE = 2'b00;
        if (rsE != 0) begin
            if ((rsE == writeregM) & regwriteM) begin
                forwardAE = 2'b10;
            end
            else if ((rsE == writeregW) & regwriteW) begin
                forwardAE = 2'b01;
            end
            else begin
                forwardAE = 2'b00;
            end
        end

        if (rtE != 0) begin
            if ((rtE == writeregM) & regwriteM) begin
                forwardBE = 2'b10;
            end
            else if ((rtE == writeregW) & regwriteW) begin
                forwardBE = 2'b01;
            end
            else begin
                forwardBE = 2'b00;
            end
        end
    end

    assign forwardAD = (rsD != 0) & (rsD == writeregM) & regwriteM;
    assign forwardBD = (rtD != 0) & (rtD == writeregM) & regwriteM;

    wire lwstall;
    wire branchstall;
    assign branchstall = branchD &
        (regwriteE & (writeregE == rsD | writeregE == rtD) | memtoregM & (
            writeregM == rsD | writeregM == rtD));
    assign lwstall = memtoregE & (rsE == rsD | rtE == rtD);
    assign stallF = lwstall | branchstall;
    assign stallD = stallF;
    assign flushE = stallF;
endmodule
```

C Controller代码

```
module controller(
    input wire [5:0] op, funct,
    input wire zero,
    output memtoreg, memwrite,
    output pcsrc, alusrc,
    output wire regdst, regwrite,
    output jump, branch,
    output [2:0] alucontrol
);
    wire [1:0] aluop;

    main_decoder md(
        .op(op),
        .memtoreg(memtoreg),
        .memwrite(memwrite),
        .branch(branch),
        .alusrc(alusrc),
        .regdst(regdst),
        .regwrite(regwrite),
        .jump(jump),
        .aluop(aluop)
    );

    alu_decoder ad(
        .funct(funct),
        .aluop(aluop),
        .alucontrol(alucontrol)
    );

    assign pcsrc = branch & zero;

endmodule
```