《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)02 班	**姓名**	文红兵
实验题目	实验四简单五级流水线 CPU		
实验时间	2023 年 5 月 11 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: 2023 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 冒险处理模块

2.1.1 功能描述

1、实现数据前向（Data Forward）功能，其中 ForwardA_E 和 Forward_E 用于识别 EX 段的寄存器数据是否需要被前向到当前指令的 ID 段，Forward_D 和 Forward_D 用于识别 MEM 段的寄存器数据是否需要被前向到当前指令的 ID 段。

2、实现流水线暂停（Pipeline Stall）功能，其中 LwStall 用于检测当前指令是加载指令且在 EX 段中的目标寄存器与 ID 段中的源寄存器重复，BranchStall 用于检测当前指令是分支指令且在 EX 或 MEM 段中写入的目标寄存器与 ID 段中的源寄存器重复。如果出现这些情况，就需要暂停流水线，等待之前的指令执行完成。

3、实现流水线清空（Pipeline Flush）功能，当流水线暂停时，为了保证程序正确性，需要将之前已经进入流水线的指令清空，重新开始执行。Flush_E 用于清空 EX 段的指令。

2.1.2 接口定义

表 1: harzard 模块的接口定义

信号名	方向	位宽	功能描述
Rs_E	Input	5-bit	EX 段中源寄存器 1 的编号
Rt_E	Input	5-bit	EX 段中源寄存器 2 的编号
Rt_D	Input	5-bit	ID 段中目标寄存器的编号
Rs_D	Input	5-bit	ID 段中源寄存器 1 的编号
WriteReg_M	Input	5-bit	MEM 段写入寄存器的编号
RegWrite_M	Input	1-bit	MEM 段是否需要写入寄存器
WriteReg_W	Input	5-bit	WB 段写入寄存器的编号
RegWrite_W	Input	1-bit	WB 段是否需要写入寄存器
MemtoReg_E	Input	1-bit	EX 段是否需要从内存读取数据
branch_D	Input	1-bit	ID 段是否为分支指令
MemtoReg_M	Input	1-bit	MEM 段是否需要从内存读取数据
RegWrite_E	Input	1-bit	EX 段是否需要写入寄存器
WriteReg_E	Input	5-bit	EX 段写入寄存器的编号
ForwardA_E	Output	2-bit	EX 段源寄存器 1 数据前向给 ID 段的方式
ForwardB_E	Output	2-bit	EX 段源寄存器 2 数据前向给 ID 段的方式
Stall_F	Output	1-bit	流水线是否需要暂停在 IF 段
Stall_D	Output	1-bit	流水线是否需要暂停在 ID 段
Flush_E	Output	1-bit	是否需要清空 EX 段指令
ForwardA_D	Output	1-bit	MEM 段源寄存器 1 数据前向给 ID 段的方式
ForwardB_D	Output	1-bit	MEM 段源寄存器 2 数据前向给 ID 段的方式

3 实验过程记录

3.1 问题 1: 变量过多

问题描述:变量过多导致写代码,调试逻辑混乱。

解决方案:修改所有变量名字,规范变量命名方法。

3.2 问题 2: 数据前推

问题描述:数据前推信号正确,但是选择出的数据却是错误。

解决方案:发现是由于将 Verilog 的”10“错误地当作 2 进制,改正错位为”2‘b10“后正确。

3.3 问题 3: 指令使用错误

问题描述:指令发现使用错误,使用了实验三的指令 coe。

解决方案:修正成为实验四的指令 coe 后正确。

4 实验结果及分析

4.1 仿真图

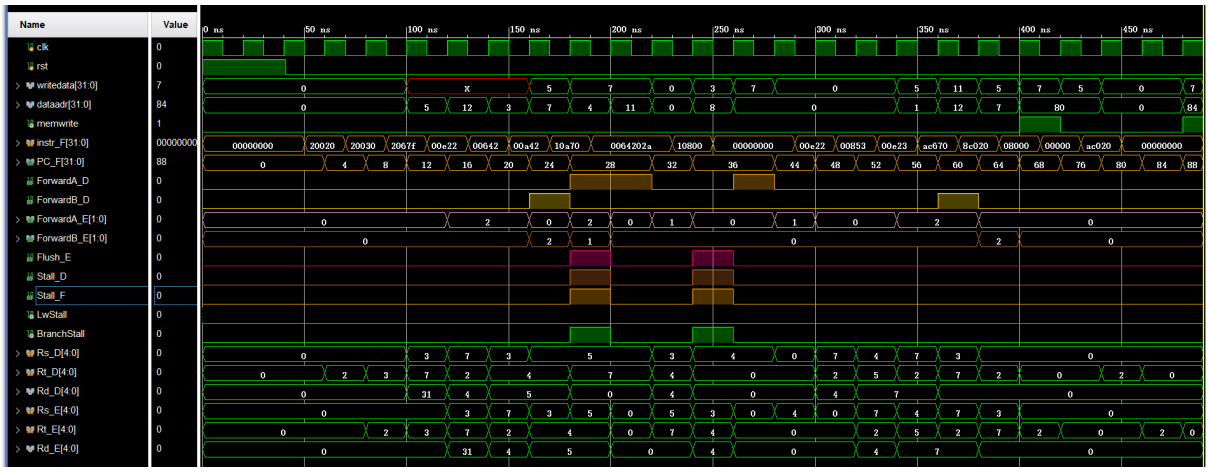

图 1: 仿真图

4.2 控制台输出

```
# }
# run 1000ns
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.inst_mem.inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a b
Block Memory Generator module testbench.dut.data_mem.inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a b
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

图 2: 控制台输出图

A Datapath 代码

```
module datapath(
        input wire CLK, RST,
        input wire[4:0] WriteReg_M, WriteReg_W, WriteReg_E,
        input wire[4:0] Rs_E, Rt_E, Rt_D, Rs_D,
        input wire Stall_F, Stall_D,  ForwardA_D, ForwardB_D, Flush_E,
        input wire[1:0] ForwardA_E, ForwardB_E,
        input wire[31:0] instr_F, // 输入指令
        input wire[31:0] instr_D,
        input wire[31:0] ReadData_M, // data memory的读出数据
        input wire [2:0] ALUControl_E,  // ALU的控制信号
        input wire PCSrc_D , // 启用分支地址信号
        input wire MemtoReg_W , // 写入寄存器堆数据的选择信号, 1选择ReadData
        input wire ALUSrc_E , // 启用立即数信号
        input wire RegDst_E , // 启用rd寄存器信号
        input wire RegWrite_W , // 寄存器堆写使能信号
        input wire Jump_D , // 无条件跳转指令
        input wire MemWrite_M,  // 控制信号, data memory的写使能信号
        input wire MemRead_M,   // 控制信号, data memory的读使能信号
        output wire[31:0] ALUResult_M,  // ALU计算结果
        output wire[31:0] WriteData_M,  // Register File第二个读出的数据
        output wire[31:0] PC_F,  // 地址
        output wire Zero_D,   // 零信号
        output wire Flush_M, Flush_W, Stall_E, Stall_M, Stall_W
    );

    // pipeline wire
    wire[31:0] ALUResult_W, ReadData_W, WriteData_E;
    wire[4:0] Rd_E, Rd_D ;
    wire[31:0] rd1_E, rd2_E, SignImm_E ;
    wire[31:0] PCPLUS4_D ;


    // wire
    wire[31:0] PCPLUS4_F ;// pc + 4
    wire[31:0] ALUResult_E ; // ALU计算结果
    wire[31:0] SignImm_D ; // 立即数符号扩展的数字
    wire[31:0] rd1_D ; // Register File第一个读出的数据
    wire[31:0] rd2_D ; // Register File第一个读出的数据
    wire[31:0] BranchSrcA_D, BranchSrcB_D ; // branch分支指令的两个操作数
    wire Flush_D, Flush_F ;



    // PCSrc_D and Jump_D
    wire[31:0] PCSeclected_F, PCBranch_D, PCJump_D ;
    assign PCBranch_D = PCPLUS4_D + {SignImm_D[29:0],2'b00} ;
    assign PCJump_D = {PCPLUS4_D[31:28],{instr_D[25:0], 2'b00}} ;
```

```
assign PCSeclected_F = (Jump_D == 1)? PCJump_D :
                       (PCSrc_D == 1)? PCBranch_D : PCPLUS4_F ;
// pc
assign Flush_F = 1'b0 ;
flopenrc #(32) pc(
    CLK, RST,~Stall_F, Flush_F,
    PCSeclected_F,
    PC_F
);

// pc + 4
assign PCPLUS4_F = PC_F + 4 ;

// RegDst_E
assign WriteReg_E = (RegDst_E == 1) ? Rd_E : Rt_E ;

// MemtoReg_W
wire[31:0] WriteBackResult_W ; // 写回数据的结果
assign WriteBackResult_W = (MemtoReg_W == 0)? ALUResult_W : ReadData_W ;

// Rs,Rt,Rd
assign Rs_D = instr_D[25:21] ;
assign Rt_D = instr_D[20:16] ;
assign Rd_D = instr_D[15:11] ;

// Zero_D
assign BranchSrcA_D = (ForwardA_D == 1)? ALUResult_M : rd1_D ;
assign BranchSrcB_D = (ForwardB_D == 1)? ALUResult_M : rd2_D ;
assign Zero_D = (BranchSrcA_D == BranchSrcB_D) ;

// regfile
regfile regfile(
    .clk(~CLK), // 时钟
    .we3(RegWrite_W), // 写入端口的使能信号
    .ra1(Rs_D),
    .ra2(Rt_D),
    .wa3(WriteReg_W), // 两个读入端口的地址，一个写入端口的地址
    .wd3(WriteBackResult_W), // 写入的数据
    .rd1(rd1_D),
    .rd2(rd2_D) // 两个端口读出的数据
);


// sign_extend
signExtension signExtension(
    .a(instr_D[15:0]),
    .y(SignImm_D)
);
```

```
    // ALUSrc_E
    wire[31:0] SrcA_E ; // ALU的第一个操作数
    wire[31:0] SrcB_E ; // ALU的第二个操作数
    assign SrcA_E = (ForwardA_E == 2'b00)? rd1_E :
                    (ForwardA_E == 2'b01)? WriteBackResult_W :
                    (ForwardA_E == 2'b10)? ALUResult_M : 32'b0 ;
    assign WriteData_E = (ForwardB_E == 2'b00)? rd2_E :
                         (ForwardB_E == 2'b01)? WriteBackResult_W :
                         (ForwardB_E == 2'b10)? ALUResult_M : 32'b0 ;
    assign SrcB_E = (ALUSrc_E == 1)? SignImm_E : WriteData_E ;

    // ALU
    alu alu(
        .a(SrcA_E),
        .b(SrcB_E),
        .op(ALUControl_E),
        .s(ALUResult_E)
        );

    // ================================================================
    // Fetch-Decode
    assign Flush_D = PCSrc_D ;
    flopenrc #(32) r1F (CLK, RST, ~Stall_D, Flush_D, PCPLUS4_F, PCPLUS4_D) ;
    flopenrc #(32) r2F (CLK, RST, ~Stall_D, Flush_D, instr_F, instr_D) ;

    // Decode-Execute
    assign Stall_E = 1'b0 ;
    flopenrc #(32) r1E (CLK, RST, ~Stall_E, Flush_E, rd1_D, rd1_E) ;
    flopenrc #(32) r2E (CLK, RST, ~Stall_E, Flush_E, rd2_D, rd2_E) ;
    flopenrc #(32) r3E (CLK, RST, ~Stall_E, Flush_E, SignImm_D, SignImm_E) ;
    flopenrc #(5) r5E (CLK, RST, ~Stall_E, Flush_E, Rs_D, Rs_E) ;
    flopenrc #(5) r6E (CLK, RST, ~Stall_E, Flush_E, Rt_D, Rt_E) ;
    flopenrc #(5) r7E (CLK, RST, ~Stall_E, Flush_E, Rd_D, Rd_E) ;

    // Execute-Memory
    assign Flush_M = 1'b0 ;
    assign Stall_M = 1'b0 ;
    flopenrc #(32) r1M (CLK, RST, ~Stall_M, Flush_M, ALUResult_E, ALUResult_M) ;
    flopenrc #(32) r2M (CLK, RST, ~Stall_M, Flush_M, WriteData_E, WriteData_M) ;
    flopenrc #(5) r3M (CLK, RST, ~Stall_M, Flush_M, WriteReg_E, WriteReg_M) ;

    // Memory-Writeback
    assign Flush_W = 1'b0 ;
    assign Stall_W = 1'b0 ;
    flopenrc #(32) r1W (CLK, RST, ~Stall_W, Flush_W, ALUResult_M, ALUResult_W) ;
    flopenrc #(32) r2W (CLK, RST, ~Stall_W, Flush_W, ReadData_M, ReadData_W) ;
    flopenrc #(5) r3W (CLK, RST, ~Stall_W, Flush_W, WriteReg_M, WriteReg_W) ;
endmodule
```

B Hazard 代码

```
module harzard(
    input wire[4:0] Rs_E, Rt_E, Rt_D,Rs_D,
    input wire[4:0] WriteReg_M,
    input wire RegWrite_M,
    input wire[4:0] WriteReg_W,
    input wire RegWrite_W, MemtoReg_E, branch_D, MemtoReg_M, RegWrite_E,
    input wire[4:0] WriteReg_E,
    output wire[1:0] ForwardA_E,ForwardB_E,
    output wire Stall_F, Stall_D, Flush_E, ForwardA_D, ForwardB_D
    );

    wire LwStall, BranchStall ;

    // Data Forward
    assign ForwardA_E = (( Rs_E != 0) & ( Rs_E == WriteReg_M ) & RegWrite_M ) ? 10
        :
                        (( Rs_E != 0) & ( Rs_E == WriteReg_W ) & RegWrite_W ) ? 01 :
                            00 ;
    assign ForwardB_E = (( Rt_E != 0) & ( Rt_E == WriteReg_M ) & RegWrite_M ) ? 10
        :
                        (( Rt_E != 0) & ( Rt_E == WriteReg_W ) & RegWrite_W ) ? 01 :
                            00 ;

    assign ForwardA_D = ( Rs_D !=0) & ( Rs_D == WriteReg_M ) & RegWrite_M ;
    assign ForwardB_D = ( Rt_D !=0) & ( Rt_D == WriteReg_M ) & RegWrite_M ;
    // Pipeline Stall
    assign LwStall = (( Rs_D == Rt_E ) | ( Rt_D == Rt_E ) ) & MemtoReg_E ;
    assign BranchStall = branch_D & RegWrite_E & ( WriteReg_E == Rs_D | WriteReg_E
        == Rt_D )
                        | branch_D & MemtoReg_M & ( WriteReg_M == Rs_D |
                            WriteReg_M == Rt_D ) ;
    assign Stall_F = LwStall | BranchStall ;
    assign Stall_D = LwStall | BranchStall ;
    assign Flush_E = LwStall | BranchStall ;
endmodule
```

C Controller 代码

```
module controller(
    input wire CLK, RST,
    input wire RegWrite_M, MemtoReg_E, branch_D, MemtoReg_M, RegWrite_E,
    input wire Stall_E, Flush_E, Stall_M, Flush_M, Stall_W, Flush_W,
    input wire[31:0] inst_D,
    input wire Zero_D,
```

```
output wire RegWrite_W, RegDst_E, ALUSrc_E, PCSrc_D, MemWrite_M, MemRead_M,
    MemtoReg_W, Jump_D,
output wire[2:0] ALUControl_E
);

wire RegWrite_D, RegDst_D, ALUSrc_D, MemWrite_D, MemRead_D, MemtoReg_D;
wire[2:0] ALUControl_D ;
wire  MemWrite_E, MemRead_E;

wire[1:0] aluop ;

assign PCSrc_D = branch_D & Zero_D ;

maindec maindec(
    .opcode(inst_D[31:26]),
    .regwrite(RegWrite_D),
    .regdst(RegDst_D),
    .alusrc(ALUSrc_D),
    .branch(branch_D),
    .memWrite(MemWrite_D),
    .memRead(MemRead_D),
    .memtoReg(MemtoReg_D),
    .jump(Jump_D),
    .aluop(aluop)
);

aludec aludec(
    .funct(inst_D[5:0]),
    .aluop(aluop),
    .alucontrol(ALUControl_D)
);

// ==========================================================
// Decode-Execute
flopenrc #(9) r1(
    CLK,RST, ~Stall_E, Flush_E,
    {RegWrite_D, RegDst_D, ALUSrc_D, MemWrite_D, MemRead_D, MemtoReg_D,
        ALUControl_D},
    {RegWrite_E, RegDst_E, ALUSrc_E, MemWrite_E, MemRead_E, MemtoReg_E,
        ALUControl_E}
) ;

// Execute-Memory
flopenrc #(4) r2(
    CLK,RST, ~Stall_M, Flush_M,
    {RegWrite_E, MemtoReg_E, MemWrite_E, MemRead_E},
    {RegWrite_M, MemtoReg_M, MemWrite_M, MemRead_M}
) ;
```

```
    // Memory-WriteBack
    flopenrc #(2) r3(
        CLK,RST,~Stall_W, Flush_W,
        {RegWrite_M, MemtoReg_M},
        {RegWrite_W, MemtoReg_W}
    );
endmodule
```