# 《计算机组成原理》实验报告

| | | | |
|---|---|---|---|
| **年级、专业、班级** | | **姓名** | |
| **实验题目** | 实验四简单五级流水线 CPU | | |
| **实验时间** | | **实验地点** | |
| **实验成绩** | 优秀/良好/中等 | **实验性质** | □验证性<br>☑设计性<br>□综合性 |
| **教师评价：**<br>□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；<br>□实验结果正确； □语法、语义正确； □报告规范；<br>其他:<br>评价教师: | | | |
| **实验目的**<br>(1)掌握流水线 (Pipelined) 处理器的思想。<br>(2)掌握单周期处理中执行阶段的划分。<br>(3)了解流水线处理器遇到的冒险。<br>(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 7 月 21 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 (Hazard Unit) 接受来自控制单元 (Controller) 的和数据通路 (Datapath) 的信号，据此判断并输出流水线旁路，阻塞和刷新的控制信号。

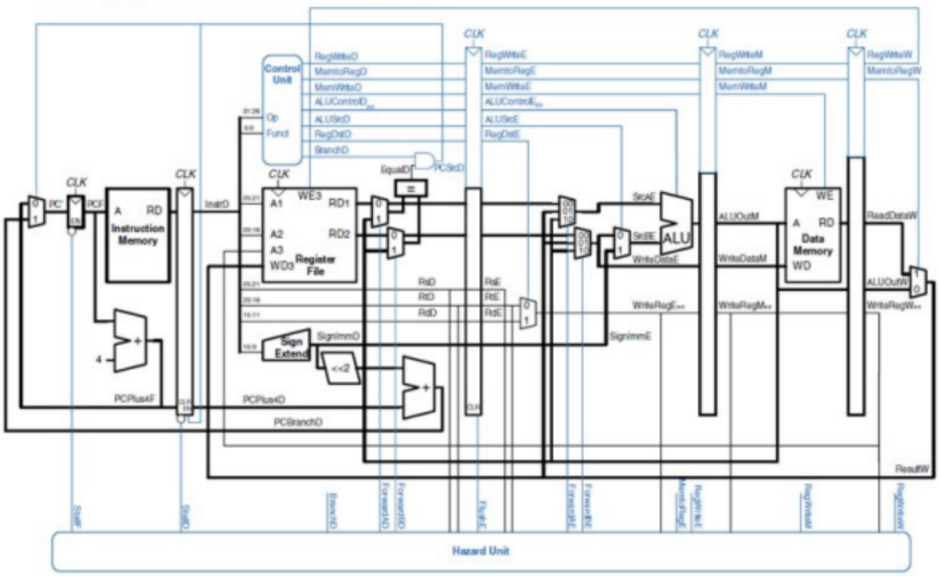


**图 1:** 冒险处理模块 (Hazard Unit)

### 2.1.2 接口定义

**表 1:** Hazard Unit 接口定义

| 信号名 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| BranchD | Input | 1-bit | ID 级指令分支控制信号 |
| MemReadE | Input | 1-bit | EX 级指令数据存储器 (DataMem) 读使能信号 |
| RegWriteE | Input | 1-bit | EX 级指令寄存器堆 (Regfile) 写使能信号 |
| RegWriteM | Input | 1-bit | MEM 级指令寄存器堆 (Regfile) 写使能信号 |
| RegWriteW | Input | 1-bit | WB 级指令寄存器堆 (Regfile) 写使能信号 |
| RsD | Input | 5-bit | ID 级指令 Rs 寄存器号 |
| RtD | Input | 5-bit | ID 级指令 Rt 寄存器号 |
| RsE | Input | 5-bit | EX 级指令 Rs 寄存器号 |
| RtE | Input | 5-bit | EX 级指令 Rt 寄存器号 |
| WriteRegE | Input | 5-bit | EX 级指令写目标寄存器号 |
| WriteRegM | Input | 5-bit | MEM 级指令写目标寄存器号 |
| WriteRegW | Input | 5-bit | WB 级指令写目标寄存器号 |
| StallF | Output | 1-bit | IF 级 PC 寄存器阻塞信号 |
| StallD | Output | 1-bit | IF/ID 流水线寄存器阻塞信号 (Datapath only) |
| ForwardAD | Output | 1-bit | ID 级” 相等则跳转 (Beq)” 指令第一操作数源前推控制信号 |
| ForwardBD | Output | 1-bit | ID 级” 相等则跳转 (Beq)” 指令第二操作数源前推控制信号 |
| FlushE | Output | 1-bit | ID/EX 流水线寄存器阻塞信号 (Both Datapath and Controller) |
| ForwardAE | Output | 2-bit | EX 级 ALU 第一操作数源前推控制信号 |
| ForwardBE | Output | 2-bit | EX 级 ALU 第二操作数源前推控制信号 |

# 3 实验过程记录

## 3.1 Datapath

### 3.1.1 问题 1: 程序停留在第一条指令

**问题描述:** Stall 信号和对应寄存器的 en 信号含义相反。当 Stall 被初始化为 0 时且与流水线寄存器正接时, 整条流水线都被阻塞。

**解决方案:** 将所有 Stall 系列信号反接。

### 3.1.2 问题 2: 寄存器堆 (Regfile), 指令存储器 (InstMem) 和数据存储器 (DataMem) 工作不正常

**问题描述:** 由于五级流水线对应的所有组合逻辑运算都应当在本周期内完成 (避免组合逻辑控制信号的变化, 并确保运算结果在下一次流水线寄存器状态刷新前到达输入端口), 因此上述模块的时钟信号应该与流水线时钟信号相反。

**解决方案:** 将寄存器堆, 指令存储器和数据存储器的时钟信号反接。

## 3.2 Flopenrc:PipelineReg_IF_ID

### 3.2.1 问题 1:j 指令丢失并导致后续指令被执行

**问题描述:** 当 j 型指令进入 ID 级时, 将计算跳转地址, 并对即将存入 IF 级指令的 IF_ID 流水线寄存器发出流水线刷新信号, 以消除该指令的后续影响。但对于连续的两条 lw 指令和 j 指令, 前者将阻塞 j 指令于 ID 级。此时, 由于 IF_ID 流水线寄存器被阻塞, 如果被刷新, 则 j 型指令将既不能进入 EX 阶段, 又未能在 IF _ID 流水线寄存器中保留副本, 由此丢失。特别注意, 此处的阻塞并非是因为 j 指令需要 lw 指令的结果, 而是因为 lwstall 的判断条件不充分导致的误判, 更强的 lwstall 判断条件本次实验不实现, 因此 j 指令作被阻塞情况执行。

**解决方案:** 修改”Flopenrc” 模块中的”else if(clear) Dataout<=0” 为”else if(clear & en) Dataout<=0”。值得一提的是, 如果编译器预先在 j 型指令后插入 nop 指令, 则可以不进行刷新操作 (将 Datapath 模块中的”FlushD=(PCSrcD|JumpD)” 修改为”FlushD=PCSrcD” 即可), 以绕过上述非法刷新操作导致的指令丢失问题。

# 4 实验结果及分析

## 4.1 含编译器优化 (nop) 的仿真图和控制台输出

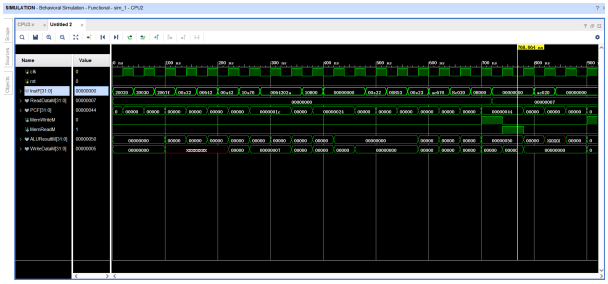

**图 2:** 仿真图 (with nop)

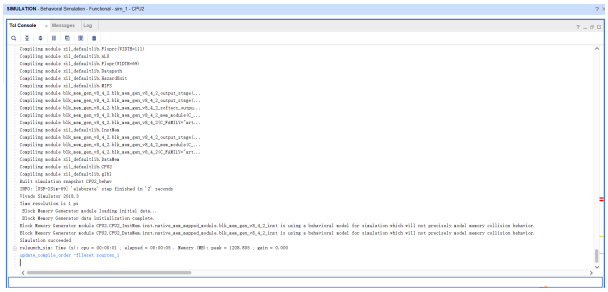

**图 3:** 控制台输出 (with nop)

## 4.2 不含编译器优化 (nop) 的仿真图和控制台输出

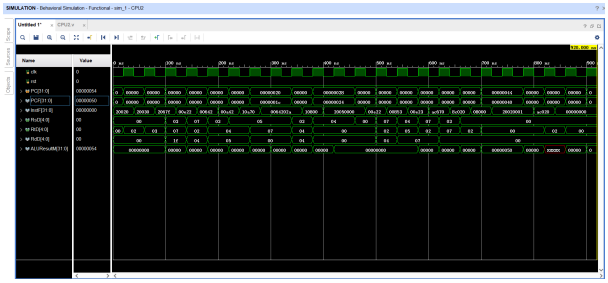

**图 4:** 仿真图 (without nop)

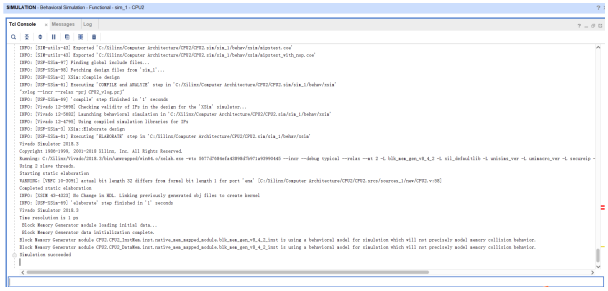

**图 5:** 控制台输出 (without nop)

# A CPU2(Pipeline_CPU) 代码

```
module CPU2();
    reg clk;
    reg rst;

    wire [31:0] InstF;
    wire [31:0] ReadDataM;
    wire [31:0] PCF;
    wire MemWriteM;
    wire MemReadM;
    wire [31:0] ALUResultM;
    wire [31:0] WriteDataM;

    always #20 clk=~clk;
    initial
    begin
        clk<=0;
        rst<=0;
    end

  always @(negedge clk) begin
    if(MemWriteM) begin
      /* code */
      if(ALUResultM=== 84 & WriteDataM === 7) begin
        /* code */
        $display("Simulation succeeded");
        $stop;
      end else if(ALUResultM !== 80) begin
        /* code */
        $display("Simulation Failed");
        $stop;
      end
    end
  end

    MIPS CPU2_MIPS(clk,rst,InstF,ReadDataM,PCF,MemWriteM,MemReadM,ALUResultM,
        WriteDataM);
    InstMem CPU2_InstMem(.clka(~clk),.ena(1),.addra(PCF),.douta(InstF));
    DataMem CPU2_DataMem(.clka(~clk),.ena(MemWriteM|MemReadM),.rsta(rst),.wea({4{
        MemWriteM}}),.addra(ALUResultM),.dina(WriteDataM),.douta(ReadDataM));
endmodule
```

# B MIPS 代码

```
module MIPS(clk,rst,InstF,ReadDataM,PCF,MemWriteM,MemReadM,ALUResultM,WriteDataM);
    input clk;
```

```
    input rst;
    input [31:0] InstF;
    input [31:0] ReadDataM;

    output [31:0] PCF;
    output MemWriteM;
    output MemReadM;
    output [31:0] ALUResultM;
    output [31:0] WriteDataM;


    wire [5:0] op,Funct;//Controller
    wire BranchD;
    wire JumpD;
    wire [2:0] ALUControlE;
    wire AluSrcE;
    wire RegDstE;
    wire MemReadE;
    wire RegWriteE;
    wire RegWriteM;
    wire RegWriteW;
    wire MemtoRegW;
    wire [4:0] RsD,RtD;//Datapath
    wire [4:0] RsE,RtE;
    wire [4:0] WriteRegE,WriteRegM,WriteRegW;
    wire StallD;//HazardUnit
    wire StallF;
    wire FlushE;
    wire ForwardAD,ForwardBD;
    wire [1:0] ForwardAE,ForwardBE;

    Controller MIPS_Controller(clk,rst,FlushE,op,Funct,BranchD,JumpD,ALUControlE,
        AluSrcE,RegDstE,MemReadE,RegWriteE,RegWriteM,MemWriteM,MemReadM,RegWriteW,
        MemtoRegW);
    Datapath MIPS_Datapath(clk,rst,InstF,StallF,BranchD,JumpD,StallD,ForwardAD,
        ForwardBD,ALUControlE,AluSrcE,RegDstE,FlushE,ForwardAE,
                                ForwardBE,ReadDataM,RegWriteW,MemtoRegW,PCF,op,Funct,
                                    RsD,RtD,RsE,RtE,WriteRegE,ALUResultM,WriteDataM,
                                    WriteRegM,WriteRegW);
    HazardUnit MIPS_HazardUnit(BranchD,MemReadE,RegWriteE,RegWriteM,RegWriteW,RsD,
        RtD,RsE,RtE,WriteRegE,WriteRegM,WriteRegW,StallF,StallD,
                                ForwardAD,ForwardBD,FlushE,ForwardAE,ForwardBE);
endmodule
```

# C Controller 代码

```
module Controller(clk,rst,clear,op,Funct,BranchD,JumpD,ALUControlE,AluSrcE,RegDstE,
    MemReadE,RegWriteE,RegWriteM,MemWriteM,MemReadM,RegWriteW,MemtoRegW);
    input clk;
    input rst;
    input clear;
    input [5:0] op;//ID
    input [5:0] Funct;
    output BranchD;//ID
    output JumpD;
    output [2:0] ALUControlE;//EX
    output AluSrcE;
    output RegDstE;
    output MemReadE;
    output RegWriteE;
    output RegWriteM;
    output MemWriteM;//MEM
    output MemReadM;
    output RegWriteW;//WB
    output MemtoRegW;
    wire [1:0] ALUOp;

    wire RegWriteD;
    wire MemtoRegD,MemtoRegE,MemtoRegM;
    wire MemWriteD,MemWriteE;
    wire MemReadD;
    wire [2:0] ALUControlD;
    wire AluSrcD;
    wire RegDstD;

    Maindec ControlMaindec(op,RegWriteD,RegDstD,AluSrcD,BranchD,MemWriteD,MemtoRegD
        ,MemReadD,JumpD,ALUOp);
    Aludec ControlAludec(ALUOp,Funct,ALUControlD);

    Floprc #(9) ControlReg_ID_EX(clk,rst,clear,{RegWriteD,RegDstD,AluSrcD,MemWriteD
        ,MemtoRegD,MemReadD,ALUControlD},{RegWriteE,RegDstE,AluSrcE,MemWriteE,
        MemtoRegE,MemReadE,ALUControlE});
    Flopr #(4) ControlReg_EX_MEM(clk,rst,{RegWriteE,MemWriteE,MemtoRegE,MemReadE},{
        RegWriteM,MemWriteM,MemtoRegM,MemReadM});
    Flopr #(2) ControlReg_MEM_WB(clk,rst,{RegWriteM,MemtoRegM},{RegWriteW,MemtoRegW
        });
endmodule
```

# D Datapath 代码

```
module Datapath(clk,rst,InstF,StallF,BranchD,JumpD,StallD,ForwardAD,ForwardBD,
    ALUControlE,AluSrcE,RegDstE,FlushE,ForwardAE,ForwardBE,ReadDataM,RegWriteW,
    MemtoRegW,PCF,op,Funct,RsD,RtD,RsE,RtE,WriteRegE,ALUResultM,WriteDataM,WriteRegM
```

```
,WriteRegW);
input clk;
input rst;
input [31:0] InstF;//IF
input StallF;
input BranchD;//ID
input JumpD;
input StallD;
input ForwardAD,ForwardBD;
input [2:0] ALUControlE;//EX
input AluSrcE;
input RegDstE;
input FlushE;
input [1:0] ForwardAE,ForwardBE;
input [31:0] ReadDataM;//MEM
input RegWriteW;//WB
input MemtoRegW;

output [31:0] PCF;//IF
output [5:0] op,Funct;
output [4:0] RsD,RtD;//ID
output [4:0] RsE,RtE;//EX
output [4:0] WriteRegE;
output [31:0] ALUResultM;//MEM
output [31:0] WriteDataM;
output [4:0] WriteRegM;
output [4:0] WriteRegW;//WB

wire [31:0] PC;//IF
wire [31:0] PCPlus4F;
wire [31:0] PC_ifBranch;
wire [31:0] Branch_Addr;
wire [31:0] Jump_Addr;
wire [31:0] InstD;//ID
wire [31:0] PCPlus4D;
wire [4:0] RdD;
wire PCSrcD;
wire EqualD;
wire [31:0] BeqSrcAD,BeqSrcBD;
wire [31:0] ReadData1D,ReadData2D;
wire [31:0] SignExtendD;
wire [31:0] ifBranch_Addr;
wire [31:0] ReadData1E,ReadData2E;//EX
wire [4:0] RdE;
wire [31:0] SignExtendE;
wire [31:0] WriteDataE;
wire [31:0] SrcAE,SrcBE;
wire [31:0] ALUResultE;
wire [31:0] ReadDataW;//WB
```

```
wire [31:0] ALUResultW;
wire [4:0] WriteRegW;
wire [31:0] WriteData3W;


PC PCRegF(clk,rst,~StallF,PC,PCF);//IF

assign PCPlus4F=PCF+4;//IF
assign PC_ifBranch=(PCSrcD==0)?PCPlus4F:Branch_Addr;
assign PC=(JumpD==0)?PC_ifBranch:Jump_Addr;

Flopenrc #(64) PiplineReg_IF_ID(clk,rst,FlushD,~StallD,{InstF,PCPlus4F},{InstD,
    PCPlus4D});//IF_ID
Regfile RegfileD(~clk,RegWriteW,RsD,RtD,WriteRegW,WriteData3W,ReadData1D,
    ReadData2D);//ID

assign op=InstD[31:26];//ID
assign Funct=InstD[5:0];
assign RsD=InstD[25:21];
assign RtD=InstD[20:16];
assign RdD=InstD[15:11];
assign BeqSrcAD=(ForwardAD==0)?ReadData1D:ALUResultM;
assign BeqSrcBD=(ForwardBD==0)?ReadData2D:ALUResultM;
assign EqualD=(BeqSrcAD==BeqSrcBD);
assign PCSrcD=(BranchD&EqualD);
assign FlushD=(PCSrcD|JumpD);
assign SignExtendD={{16{InstD[15]}},InstD[15:0]};
assign Branch_Addr=PCPlus4D+{SignExtendD[30:0],2'b00};
assign Jump_Addr={PCPlus4D[31:28],InstD[25:0],2'b00};

Floprc #(111) PiplineReg_ID_EX(clk,rst,FlushE,{ReadData1D,ReadData2D,RsD,RtD,
    RdD,SignExtendD},{ReadData1E,ReadData2E,RsE,RtE,RdE,SignExtendE});//ID_EX
ALU ALUE(ALUControlE,SrcAE,SrcBE,ALUResultE);

assign SrcAE=(ForwardAE==2'b00)?ReadData1E:
             (ForwardAE==2'b01)?WriteData3W:
             (ForwardAE==2'b10)?ALUResultM:32'b0;
assign WriteDataE=(ForwardBE==2'b00)?ReadData2E:
                  (ForwardBE==2'b01)?WriteData3W:
                  (ForwardBE==2'b10)?ALUResultM:32'b0;
assign SrcBE=(AluSrcE==0)?WriteDataE:SignExtendE;
assign WriteRegE=(RegDstE==0)?RtE:RdE;

Flopr #(69) PiplineReg_EX_MEM(clk,rst,{ALUResultE,WriteDataE,WriteRegE},{
    ALUResultM,WriteDataM,WriteRegM});//EX_MEM

Flopr #(69) PiplineReg_MEM_WB(clk,rst,{ReadDataM,ALUResultM,WriteRegM},{
    ReadDataW,ALUResultW,WriteRegW});//MEM_WB
```

```
    assign WriteData3W=(MemtoRegW==0)?ALUResultW:ReadDataW;
endmodule
```

# E HazardUnit 代码

```
module HazardUnit(BranchD,MemReadE,RegWriteE,RegWriteM,RegWriteW,RsD,RtD,RsE,RtE,
    WriteRegE,WriteRegM,WriteRegW,StallF,StallD,ForwardAD,ForwardBD,FlushE,ForwardAE
    ,ForwardBE);
    input BranchD;
    input MemReadE;
    input RegWriteE;
    input RegWriteM;
    input RegWriteW;
    input [4:0] RsD,RtD;
    input [4:0] RsE,RtE;
    input [4:0] WriteRegE;
    input [4:0] WriteRegM;
    input [4:0] WriteRegW;

    output StallF;
    output StallD;
    output ForwardAD,ForwardBD;
    output FlushE;
    output [1:0] ForwardAE,ForwardBE;

    wire lwstall;
    wire branchstall;

    //数据冒险
    assign ForwardAE=(RegWriteM & (WriteRegM!=0) & (WriteRegM==RsE))?10://EX冒险前
        推信号
                    (RegWriteW & (WriteRegW!=0) & (WriteRegW==RsE))?01:00;//(复杂)
                        MEM冒险前推信号
    assign ForwardBE=(RegWriteM & (WriteRegM!=0) & (WriteRegM==RtE))?10://EX冒险前
        推信号
                    (RegWriteW & (WriteRegW!=0) & (WriteRegW==RtE))?01:00;//(复杂)
                        MEM冒险前推信号
    assign lwstall=((RsD==RsE) | (RtD==RtE)) & MemReadE;//lw冒险阻塞信号

    //控制冒险
    assign ForwardAD=(RegWriteM & (WriteRegM!=0) & (WriteRegM==RsD));//MEM冒险前推
        信号,本次实验不实现WB冒险前推信号
    assign ForwardBD=(RegWriteM & (WriteRegM!=0) & (WriteRegM==RtD));
    assign branchstall=(BranchD & RegWriteE & ((WriteRegE==RsD) | (WriteRegE==RtD))
        );//MEM冒险阻塞信号
    // | (BranchD & MemReadE & ((WriteRegM==RsD) | (WriteRegM==RtD)));//WB冒险阻塞
        信号,本次实验不必要
```

```
    //阻塞与刷新
    assign StallF=lwstall | branchstall;//lw或beq指令均需阻塞ID级和IF级指令
    assign StallD=lwstall | branchstall;
    assign FlushE=lwstall | branchstall;//lw或beq均需清空ID级指令影响
    //FlushD=(PCSrcD|JumpD),beq指令分支发生时或j指令执行时才需要清空IF级指令
endmodule
```