《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 05 班,04 班	**姓名**	冯宇馨,胡鑫
实验题目	实验四简单五级流水线 CPU		
实验时间	2023 年 5 月 7 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: 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: 2023 年 5 月 23 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

冒险处理模块主要用于解决数据冒险和控制冒险

其中数据冒险指在多个指令之间，有数据依赖的情况，我们主要用数据前推和流水线暂停的方法解决，一般的数据冒险问题可以用数据前推解决，但是在 lw 指令中，，lw 指令在 M 阶段才能够从数据存储器读取数据，此时下一条指令已经完成 ALU 计算，无法进行数据前推，于是需要加入流水线暂停信号一起解决。判断 decode 阶段 rs 或 rt 的地址是否是 lw 指令要写入的地址，设置 PC、fetch->decode 阶段触发器的暂停信号，Decode->exexcute 阶段触发器清除。

控制冒险是分支指令引起的冒险。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，我们也主要用数据前推和流水线暂停的方法解决。控制冒险指分支指令的跳转结果需要在第四周期才知道，然而此时已经又读进来了 3 条指令。我们需要将这三条指令产生的影响全部清除，解决办法是将分支指令的判断提前 2 个周期，在 regfile 输出后添加一个判断相等的模块，即可提前判断 beq，然而还有一条指令不能被减少，于是需要流水线暂停，又出现数据冲突问题，可能存在上一条指令需要改变 beq 判断的两个寄存器的值，需要增加数据前推和流水线暂停模块，等到 M 阶段算出 aluoutM 后利用 forwardAD，forwardBD 来将 aluoutM 的数据前推

2.1.2 接口定义

表 1: 接口定义模版

信号名	方向	位宽	功能描述
rsD	input	5-bit	ID(译码)阶段的 rs 寄存器编号
rtD	input	5-bit	ID(译码)阶段的 rt 寄存器编号
branchD	input	1-bit	ID（译码）阶段的有条件分支跳转信号
rsE	input	5-bit	EX(执行)阶段的 rs 寄存器编号
rtE	input	5-bit	EX(执行)阶段的 rt 寄存器编号
writeregE	input	5-bit	EX（执行）阶段的需要写回的寄存器号
regwriteE	input	1-bit	EX（执行）阶段的是否需要写寄存器堆的控制信号
memtoregE	input	1-bit	EX（执行）阶段的回写寄存器堆数据的控制信号,0 表示数据来自于 ALU 计算的结果, 1 表示存储器读取的数据
writeregM	input	5-bit	MEM（访存）阶段的需要写回的寄存器号
regwriteM	input	1-bit	MEM（访存）阶段的是否需要写寄存器堆的控制信号
memtoregM	input	1-bit	MEM（访存）阶段的回写寄存器堆数据的控制信号, 0 表示数据来自于 ALU 计算的结果,1 表示存储器读取的数据
writeregW	input	5-bit	WB（回写）阶段的需要写回的寄存器号
regwriteW	input	1-bit	WB（回写）阶段的是否需要写寄存器堆的控制信号
stallF	output	1-bit	F 阶段流水线暂停信号
stallD	output	1-bit	D 阶段流水线暂停信号
flushE	output	1-bit	E 阶段流水线刷新信号
forwardAD	output	1-bit	分支跳转时数据前推 A 端控制信号, 0 为 rd1,1 为 aluoutM
forwardBD	output	1-bit	分支跳转时数据前推 B 端控制信号, 0 为 rd1,1 为 aluoutM
forwardAE	output	2-bit	ALU 模块数据前推 A 端控制信号, 00 为 rd1,01 为 resultW,10 为 aluoutM
forwardBE	output	2-bit	ALU 模块数据前推 B 端控制信号, 00 为 rd1,01 为 resultW,10 为 aluoutM

3 实验过程记录

3.1 下板如何显示指定地址寄存器内容

刚开始我们组是在 top 文件中又调用了一个寄存器，将我们要的地址传入，发现没有值输出，原因是新寄存器和 top 里面操作的寄存器是两个不同的寄存器，因此不能运行后直接读出，只能在过程中实现，于是我们在寄存器模块中增加一个 5 位宽地址信号的输入 addr 和内容的输出 rd3，因此就能实现在运行过程中看到指定地址的内容的变化，如下图所示

```
module regfile(
    input wire clk,
    input wire we3,
    input wire[4:0] ra1,ra2,wa3,
    input wire[31:0] wd3,
    input wire[4:0]addr,
    output wire[31:0] rd3,
    output wire[31:0] rd1,rd2
    );

    reg [31:0] rf[31:0];

    always @(negedge clk) begin
        if(we3) begin
            rf[wa3] <= wd3;
        end
    end

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
    assign rd3 = (addr!= 0) ? rf[addr]: 0;
endmodule
```

3.2 下板如何显示指定地址存储器内容

同样我们犯了和寄存器一样的错误，我们又重新实例化一个 datamem 想要从中读取值，然而是不能实现的，但是存储器不像寄存器一样可以修改，他是调用的 IP，我们只好人工创造了一个存储器 a，当我们指定一个地址时，便在过程中把 datamem 里面该地址的变动传入 a，我们再通过 a 来显示存储器内容，如下图所示

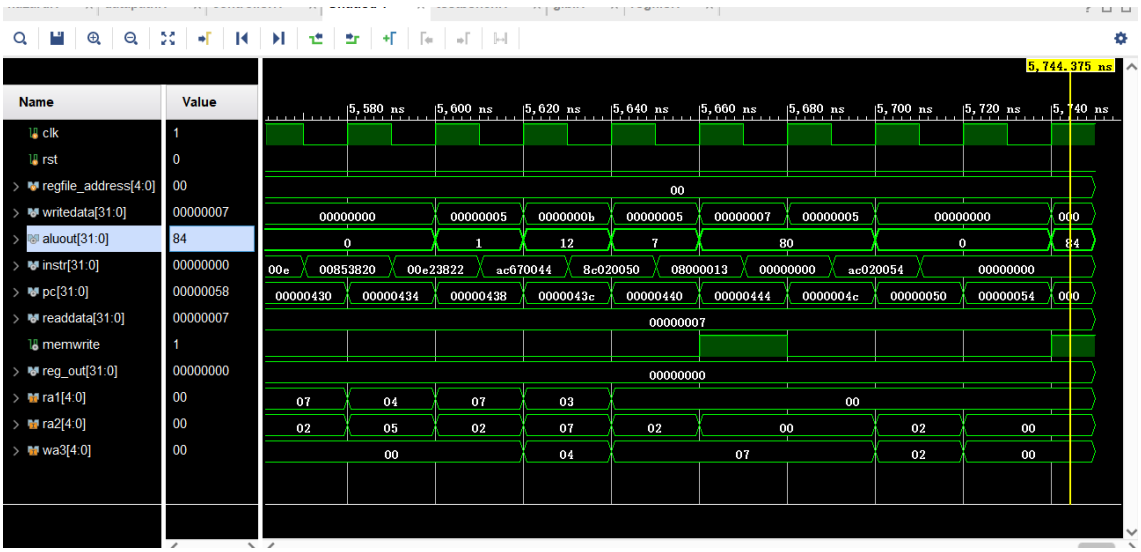
```
reg [31:0]dataram[0:100];
always@(*)
begin
if((seat==aluout)&memwrite)
begin
dataram[seat]<=readdata;
end
end
```

3.3 回写存储器的值出错

我们一开始 aluout 能正确运行到 84，但是 writedata 却不能运行到 7，原因是存储器的 clk 要沿下降沿触发，不然就会落后

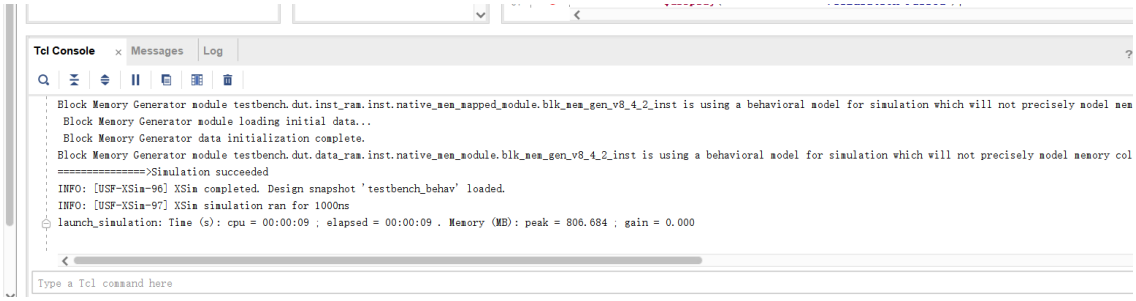
4 实验结果及分析

4.1 仿真图



最后一条指令是 sw 2,84(0)；aluout 是写回 datamem 的地址偏移量为 84，writedata 是回写 datamem 的值为 7，ra1 代表 rs，ra2 代表 rt，可见在 D 阶段（前两个周期），rs 为 0，rt 为 2，运行正确

4.2 控制台输出



输出 Simulation succeeded,仿真正确

A Datapath 代码

```
module datapath(
  input wire clk,rst,
  input wire [4:0]regfile_address,
  //IF
  output wire[31:0] pcF,
  input wire[31:0] instrF,

  //ID
  input wire branchD,
  input wire jumpD,
  output wire[5:0] opD,functD,
    output wire [31:0]reg_out,//regs输出值
  //EX
  input wire memtoregE,
  input wire alusrcE,
  input wire regdstE,
  input wire regwriteE,
  input wire[2:0] alucontrolE,
  output wire flushE,

  //MEM
  input wire memtoregM,
  input wire regwriteM,
  output wire[31:0] aluoutM,
  output wire[31:0]writedataM,
  input wire[31:0] readdataM,

  //WB
  input wire memtoregW,
  input wire regwriteW
);
```

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


//IF
wire stallF;
//pc也是一个触发器
wire [31:0]pc_next,pcplus4F;

 pc pc0(clk,rst,~stallF,pc_next,pcF);
 adder pcadd(pcF,32'b100,pcplus4F);


//F-D
wire pcsrcD,equalD;
wire [31:0] pcplus4D,instrD;
wire forwardAD,forwardBD;
wire [4:0] rsD,rtD,rdD;
wire flushD,stallD;
flopenr #(32) fd1(clk,rst,~stallD,pcplus4F,pcplus4D);
flopenrc #(32) fd2(clk,rst,~stallD,flushD,instrF,instrD);


//ID
assign opD = instrD[31:26];
assign functD = instrD[5:0];
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];

//符号扩展
wire [31:0] signimmD,signimmD_sl2;
signext signext0(instrD[15:0],signimmD);
sl2 sll(signimmD,signimmD_sl2);
wire [31:0]pcbranchD;

//pcbranch
adder pcadd2(pcplus4D,signimmD_sl2,pcbranchD);

//regfile
wire [31:0]rd1D,rd2D;
regfile rf(clk,regwriteW,rsD,rtD,writeregW,resultW,regfile_address,reg_out,rd1D,
    rd2D);


//equalD
```

```
wire [31:0] srcAD,srcBD;
mux2 #(32) mux1(rd1D,aluoutM,forwardAD,srcAD);
mux2 #(32) mux2(rd2D,aluoutM,forwardBD,srcBD);
eqcmp comp(srcAD,srcBD,equalD);
assign pcsrcD = branchD & equalD;

 //pc_jump&pc_next
wire [31:0]pc_tmp,pc_jump;
mux2 #(32) mux3(pcplus4F,pcbranchD,pcsrcD,pc_tmp);
assign pc_jump={pcplus4D[31:28],instrD[25:0],2'b00};
mux2 #(32) pcmux(pc_tmp,pc_jump,jumpD,pc_next);

//D-E
wire [31:0] rd1E,rd2E;
wire [4:0] rsE,rtE,rdE;
wire [31:0] signimmE;
floprc #(32) de1(clk,rst,flushE,rd1D,rd1E);
floprc #(32) de2(clk,rst,flushE,rd2D,rd2E);
floprc #(32) de3(clk,rst,flushE,signimmD,signimmE);
floprc #(5) de4(clk,rst,flushE,rsD,rsE);
floprc #(5) de5(clk,rst,flushE,rtD,rtE);
floprc #(5) de6(clk,rst,flushE,rdD,rdE);


//EX
wire [31:0]srcAE,srcBE,writedataE;
wire [31:0] aluoutE;
wire [4:0] writeregE;
wire [1:0] forwardAE,forwardBE;
mux3 #(32) mux_3(rd1E,resultW,aluoutM,forwardAE,srcAE);
mux3 #(32) mux4(rd2E,resultW,aluoutM,forwardBE,writedataE);
mux2 #(32) mux5(writedataE,signimmE,alusrcE,srcBE);
mux2 #(5) mux6(rtE,rdE,regdstE,writeregE);
alu alu(srcAE,srcBE,alucontrolE,aluoutE);

//E-M
wire [4:0] writeregM;
flopr #(32) em1(clk,rst,aluoutE,aluoutM);
flopr #(32) em2(clk,rst,writedataE,writedataM);
flopr #(5) em3(clk,rst,writeregE,writeregM);


//M-W
flopr #(32) mw1(clk,rst,readdataM,readdataW);
flopr #(32) mw2(clk,rst,aluoutM,aluoutW);
flopr #(5) mw3(clk,rst,writeregM,writeregW);
mux2 #(32) mux7(aluoutW,readdataW,memtoregW,resultW);

//冒险
```

```
  hazard hazard0(
    .stallF(stallF),
    .rsD(rsD),
    .rtD(rtD),
    .branchD(branchD),
    .forwardAD(forwardAD),
    .forwardBD(forwardBD),
    .stallD(stallD),
    .rsE(rsE),
    .rtE(rtE),
    .writeregE(writeregE),
    .regwriteE(regwriteE),
    .memtoregE(memtoregE),
    .forwardAE(forwardAE),
    .forwardBE(forwardBE),
    .flushE(flushE),
    .writeregM(writeregM),
    .regwriteM(regwriteM),
    .memtoregM(memtoregM),
    .writeregW(writeregW),
    .regwriteW(regwriteW)
  );
endmodule
```

B Hazard 代码

```
module hazard(


  input wire[4:0] rsD,rtD,
  input wire branchD,
  input wire[4:0] rsE,rtE,
  input wire[4:0] writeregE,
  input wire regwriteE,
  input wire memtoregE,
  input wire[4:0] writeregM,
  input wire regwriteM,
  input wire memtoregM,
  input wire[4:0] writeregW,
  input wire regwriteW,

  output wire stallF,
  output wire forwardAD,forwardBD,
  output wire stallD,
  output reg[1:0] forwardAE,forwardBE,
  output wire flushE
);
```

```
  wire lwstallD,branchstallD;

  //branch_数据前推 ID
  assign forwardAD = (rsD != 0 & rsD == writeregM & regwriteM);
  assign forwardBD = (rtD != 0 & rtD == writeregM & regwriteM);


    //数据前推: 判断读的是否为0号寄存器, 判断寄存器号是否相同, 判断依赖的指令是否在
        回写寄存器堆
    //00为rd1,01为resultW,10为aluoutM
  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
    end
    if(rtE != 0)
    begin
      if(rtE == writeregM & regwriteM)
       begin
        forwardBE = 2'b10;
      end
      else if(rtE == writeregW & regwriteW)
      begin
        forwardBE = 2'b01;
      end
    end
  end

  //stalls
  assign #1 lwstallD = memtoregE & (rtE == rsD | rtE == rtD);
  assign #1 branchstallD=(branchD&regwriteE&((writeregE==rsD)|(writeregM==rtD)))
                         |(branchD&memtoregM&((writeregM==rsD)|(writeregM==rtD)));
  assign #1 stallD = lwstallD | branchstallD;
  assign #1 stallF = lwstallD | branchstallD;
  assign #1 flushE = lwstallD | branchstallD;

endmodule
```

C Controller 代码

```
module controller(
  input wire clk,rst,
  //ID
  input wire[5:0] opD,functD,
  output wire branchD,jumpD,
  //EX
  input wire flushE,
  output wire memtoregE,alusrcE,
  output wire regdstE,regwriteE,
  output wire[2:0] alucontrolE,
  //MEM
  output wire memtoregM,memwriteM,regwriteM,
  //WB
  output wire memtoregW,regwriteW

);
  //ID
  wire[1:0] aluopD;
  wire memtoregD,memwriteD,alusrcD,regdstD,regwriteD;
  wire[2:0] alucontrolD;

  maindec maindec0(
    .op(opD),
    .memtoreg(memtoregD),
    .memwrite(memwriteD),
    .branch(branchD),
    .alusrc(alusrcD),
    .regdst(regdstD),
    .regwrite(regwriteD),
    .jump(jumpD),
    .aluop(aluopD)
  );



  aludec aludec0(
            .funct(functD),
            .aluop(aluopD),
            .alucontrol(alucontrolD)
            );

  //EX
  wire memwriteE;

  //D-E
  floprc #(8) regE(
    clk,
```

```
    rst,
    flushE,
    {memtoregD,memwriteD,alusrcD,regdstD,regwriteD,alucontrolD},
    {memtoregE,memwriteE,alusrcE,regdstE,regwriteE,alucontrolE}
  );

  //E-M
  flopr #(3) regM(
    clk,rst,
    {memtoregE,memwriteE,regwriteE},
    {memtoregM,memwriteM,regwriteM}
  );

  //M-W
  flopr #(2) regW(
    clk,rst,
    {memtoregM,regwriteM},
    {memtoregW,regwriteW}
  );

endmodule
```