《计算机组成原理》实验报告

年级、专业、班级	2020 级计算机科学与技术 01 班 2020 级计算机科学与技术 02 班	**姓名**	陈鹏宇 徐小龙
实验题目	实验四简单五级流水线 CPU		
实验时间	2019 年 11 月 8 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: 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: 2022 年 5 月 28 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

五级流水线 MIPS 处理器，支持简单的算术逻辑运算指令与跳转指令

2.1.2 接口定义

表 1: 接口定义

信号名	方向	位宽	功能描述
instrF	Input	32-bit	Ferch 阶段 A MIPS 32-bit Instruction read from ram.
pcF	Output	32-bit	Fetch 阶段程序计数器，每个时钟上升沿自增 4，指向下一条指令.
jumpD	Output	1-bit	Decoder 阶段当 J 型无条件分支指令为 1.
branchD	Output	1-bit	Decoder 阶段条件分支指令执行后，条件成立则执行 PC 相对寻址，信号为 1 否则为 0.
pcsrcD	Output	1-bit	Decoder 阶段高电平时 pc 由分支目标地址取代，低电平时 pc+4.
regdstE	Output	1-bit	Execute 阶段高电平时写寄存器的目标寄存器号来自 rd(15:11)，低电平时来自 rt(20:16).
alu_controlE	Output	3-bit	Execute 阶段 ALU 控制信号， 由 funct(5:0) 和 op-code(31:26) 共同决定.
alusrcE	Output	1-bit	Execute 阶段高电平时 ALU 第二个操作数来自指令低 16 位符号扩展，低电平时来自寄存器堆第二个输出.
overflowE	Output	1-bit	Execute 阶段 ALU 算数溢出.
readdataM	Intput	32-bit	Memory 阶段从数据存储器读出，输入 datapath 读入 regfile.
writedataM	Output	32-bit	Memory 阶段 regfile 取出的 RD2 操作数，写入数据存储器。只用于 sw 指令.
regwriteM	Output	1-bit	Memory 阶段高电平时指令写入寄存器堆.
aluoutM	Output	32-bit	Memory 阶段数值为 ALU 的结果，当 memwrite 有效时为写入存储器地址.
memwriteM	Output	1-bit	Memory 阶段高电平时将存数指令的 rt 寄存器值写入 ALU 计算结果指向的地址.
memtoregM	Output	1-bit	Memory 阶段高电平时将取数指令的取数地址对应的存储器数据写入到寄存器堆 rt 寄存器.

2.2 冒险处理模块

2.2.1 功能描述

解决分支跳转指令控制冒险，load word 数据冒险以及 alu 数据冒险

2.2.2 接口定义

表 2: 接口定义

信号名	方向	位宽	功能描述
stallF	Output	1-bit	Ferch 阶段暂停信号.
stallD	Output	1-bit	Decode 阶段暂停信号.
flushE	Output	1-bit	Executer 阶段刷新信号.
branchD	Input	1-bit	Decoder 阶段条件分支指令执行后，条件成立则执行 PC 相对寻址，信号为 1 否则为 0.
forwardAD	Output	1-bit	Decoder 阶段如果 lw 指令与后面的指令发生数据冒险，且为 rs 寄存器则为 1.
forwardBD	Output	1-bit	Decoder 阶段如果 lw 指令与后面的指令发生数据冒险，且为 rt 寄存器则为 1.
forwardAE	Output	1-bit	Executer 阶段如果 alu 的 rs 操作数来自前面的指令，若在 M 阶段则为 2，M 阶段为 1 否则默认为 0.
forwardBE	Output	1-bit	Executer 阶段如果 alu 的 rt 操作数来自前面的指令，若在 M 阶段则为 2，M 阶段为 1 否则默认为 0.

3 实验过程记录

3.1 datapath，controller 流水线化

1. 在实验三 datapath 的基础上，添加中间寄存器使整个流水线分为取指、译码、执行、访存和写回五个部分
2. 在此基础上，datapath 的基本通路已经形成，下面加入控制器部分。控制器部分与单周期相同，仍然由 main decoder 和 alu decoder 构成，但由于改为五级流水线后，每一个阶段所需要的控制信号仅为一部分，控制器产生信号的阶段为译码阶段，产生控制信号后，依次通过触发器传到下一阶段，若当前阶段需要的信号，则不需要继续传递到下一阶段

3.2 冒险的解决

1. 冒险分为：数据冒险 (寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据) 和控制冒险 (下一条要执行的指令还未确定，就按照 PC 自增顺序执行了本不该执行的指令), 结构冒险已用两个存储器解决
2. R 型指令中，将 alu 得到的结果直接推送到下一条指令的 execute 阶段，同理，后续所有的阶段均已有结果，可以向对应的阶段推送，而不需要等到回写后再进行读取，达到数据前推的目的

3. lw 指令中，在 memory 阶段才能够从数据存储器读取数据，此时 and 指令已经完成 ALU 计算，无法进行数据前推。必须使流水线暂停，等待数据读取后，再前推到 execute 阶段
4. 根据以上实现逻辑连线即可解决数据冒险问题
5. 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。在 regfile 输出后添加一个判断相等的模块，即可提前判断 beq
6. 再上一步的基础上再增加数据前推和流水线暂停模块，即解决了控制冒险问题

3.3 问题

3.3.1 问题描述

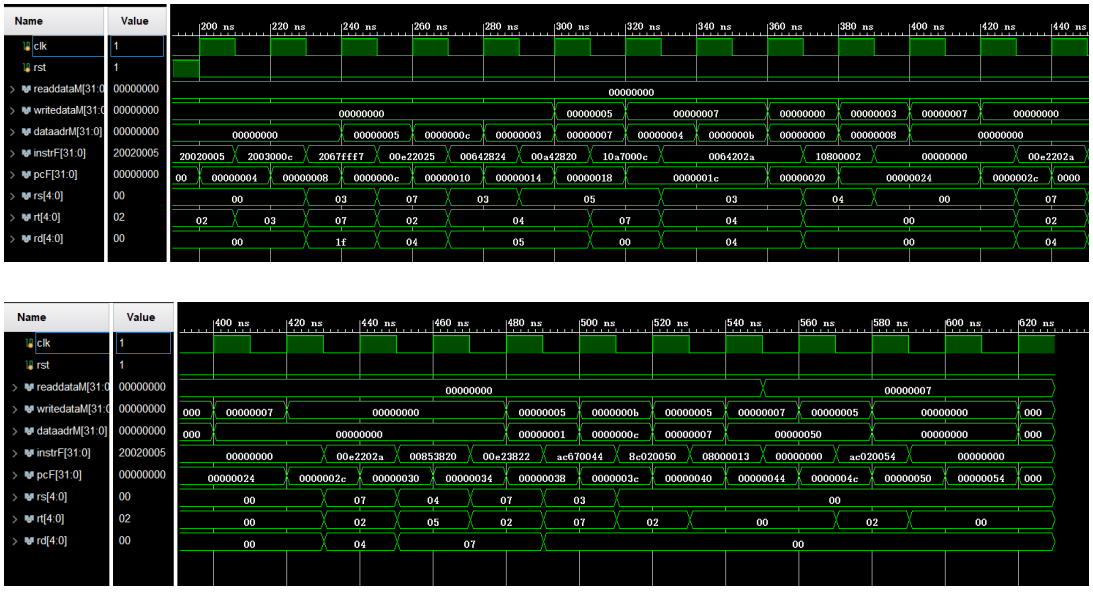
仿真图不正确，部分接口的值为高阻态

3.3.2 问题解决

由于整个 datapath、hazard 模块的值变量名不统一，代码编写时结构和数据的书写都不太有逻辑，不方便 debug，所以重新改写代码，按照取指，译码，执行，访存，回写五部进行分区改写。再改写过程中发现部分连线混淆，导致仿真出错。重新改写后正常。

4 实验结果及分析

4.1 仿真图



4.2 控制台输出

```
Time resolution is 1 ps
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.inst_ram.inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will not precisely model mem
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench.data_ram.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral model for simulation which will not precisely model memory col
 ==============>Simulation succeeded
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:06 . Memory (MB): peak = 957.863 ; gain = 0.000
```

4.3 结果分析

1. R 型数据冒险：指令为 00e22025 时，4 =7 or 2006428245 = 3*and*4，会用到上一条指令的四号寄存器，但由于写回是在第五阶段发生，所以在 alu 计算完后，就应该将结果前推到下一条指令的 execute 阶段。所以如果当前使用的 rsE 地址与 writeregM 相等，则将 memory 阶段的 aluoutM 前推；如果与 writereW 相等，则将 writeback 阶段的 resultW 前推。当前情况是将上一条指令的 aluoutM 前推。
2. lw 数据冒险：指令为 8c020050 时，2 = [80], *ac*0200542。首先判断 decode 阶段 rs 或 rt 的地址是否是 lw 指令要写入的地址，若相等则使 stallD 和 stallF 信号有效。并且使 E 阶段寄存器刷新。
3. beq 控制冒险：指令为 10800001 是，beq 指令发生，但发生时，该指令的后三条指令已经进入流水线，这时需要将这三条指令产生的影响全部清除。将分支指令的判断提前至 decode 阶段，此时能够减少两条指令的执行，只需再刷新他的后一条指令即可。在 regfile 输出后添加一个判断相等的模块，并解决新出现的数据冒险，即可提前判断 beq

A Datapath 代码

```
module datapath(
    input clk, rst,
    ////////////////////fetch//////////////////
    output [31:0] pcF,
    input  [31:0] instrF,
    ////////////////////decode//////////////////
    input pcsrcD, branchD, jumpD,
    output equalD,
    output [5:0] opD, functD,
    ////////////////////execute//////////////////////
    input memtoregE, alusrcE, regdstE, regwriteE,
    input [2:0] alucontrolE,
    output flushE,
    ////////////////////memory////////////////////
    input memtoregM, regwriteM,
    output [31:0] aluoutM,writedataM,
    input  [31:0] readdataM,
    ////////////////////write back//////////////////
    input memtoregW, regwriteW
);
    ////////////////////fetch//////////////////
    wire stallF;
```

```
wire [31:0] pcnextFD,pcnextbrFD,pc4F,pcbranchD;

pc #(32) pcreg(clk, rst, ~stallF, pcnextFD, pcF);
adder pcadder1(pcF, 32'b100, pc4F);
//////////////////decode//////////////////
wire [31:0] pc4D,instrD;
wire forwardAD,forwardBD;
wire [4:0] rsD,rtD,rdD;
wire flushD,stallD;
wire [31:0] sign_immD,sign_imm_sl2D;
wire [31:0] srcAD,srcA2D,srcBD,srcB2D;

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];
assign equalD = (srcA2D == srcB2D)? 1:0;

signext signext(instrD[15:0], sign_immD);
sl2     sl2(sign_immD, sign_imm_sl2D);
adder   pcadder2(pc4D, sign_imm_sl2D, pcbranchD);

mux2 #(32) pcmux1(pc4F, pcbranchD, pcsrcD, pcnextbrFD);
mux2 #(32) pcmux2(pcnextbrFD, {pc4D[31:28],instrD[25:0],2'b00}, jumpD,
    pcnextFD);
mux2 #(32) forwardamux(srcAD,aluoutM,forwardAD,srcA2D);
mux2 #(32) forwardbmux(srcBD,aluoutM,forwardBD,srcB2D);
flopenr #(32)  r1D(clk, rst, ~stallD, pc4F, pc4D);
flopenrc #(32) r2D(clk, rst, ~stallD, flushD, instrF, instrD);
//////////////////execute//////////////////
wire [1:0] forwardAE,forwardBE;
wire [4:0] rsE,rtE,rdE,writeregE;
wire [31:0] sign_immE,srcAE,srcA2E,srcBE,srcB2E,srcB3E,aluoutE;

mux2 #(32) srcb(srcB2E, sign_immE, alusrcE, srcB3E);
mux2 #(5)  wa(rtE, rdE, regdstE, writeregE);
alu        alu(srcA2E, srcB3E, alucontrolE, aluoutE);

floprc #(32) r1E(clk,rst,flushE,srcAD,srcAE);
floprc #(32) r2E(clk,rst,flushE,srcBD,srcBE);
floprc #(32) r3E(clk,rst,flushE,sign_immD,sign_immE);
floprc #(5)  r4E(clk,rst,flushE,rsD,rsE);
floprc #(5)  r5E(clk,rst,flushE,rtD,rtE);
floprc #(5)  r6E(clk,rst,flushE,rdD,rdE);
//////////////////memory//////////////////
wire [4:0] writeregM;

flopr #(32) r1M(clk,rst,srcB2E,writedataM);
```

```
    flopr #(32) r2M(clk,rst,aluoutE,aluoutM);
    flopr #(5)  r3M(clk,rst,writeregE,writeregM);
    ////////////////write back////////////////
    wire [4:0] writeregW;
    wire [31:0] aluoutW,readdataW,resultW;

    mux2 #(32) resmux(aluoutW,readdataW,memtoregW,resultW);
    mux3 #(32) formux1(srcAE,resultW,aluoutM,forwardAE,srcA2E);
    mux3 #(32) formux2(srcBE,resultW,aluoutM,forwardBE,srcB2E);
    flopr #(32) r1W(clk,rst,aluoutM,aluoutW);
    flopr #(32) r2W(clk,rst,readdataM,readdataW);
    flopr #(5)  r3W(clk,rst,writeregM,writeregW);

    regfile rf(clk,regwriteW,rsD,rtD,writeregW,resultW,srcAD,srcBD);
    ////////////////hazard////////////////
    hazard h(
        //////////////////fetch//////////////////
        stallF,
        //////////////////decode//////////////////
        rsD,rtD,branchD,forwardAD,forwardBD,stallD,
        //////////////////execute//////////////////
        rsE,rtE,writeregE,regwriteE,memtoregE,forwardAE,forwardBE,flushE,
        //////////////////memory//////////////////
        writeregM,regwriteM,memtoregM,
        //////////////////write back//////////////
        writeregW,regwriteW
    );
endmodule
```

B Hazard 代码

```
module hazard(
    //////////////fetch////////////////
    output stallF,
    //////////////decode//////////////
    input [4:0] rsD,rtD,
    input branchD,
    output forwardAD,forwardBD,stallD,
    //////////////execute//////////////
    input [4:0] rsE,rtE,writeregE,
    input regwriteE,memtoregE,
    output reg [1:0] forwardAE,forwardBE,
    output flushE,
    //////////////memory//////////////
    input [4:0] writeregM,
    input regwriteM,memtoregM,
    ////////// //write back////////////
```

```
    input [4:0] writeregW,
    input regwriteW
);

    ////////////////control/////////////////
    wire lwstallD,branchstallD;
    assign forwardAD = (rsD != 0 & rsD == writeregM & regwriteM);
    assign forwardBD = (rtD != 0 & rtD == writeregM & regwriteM);
    ////////////////data forward/////////////////
    always @(*)
       begin
            forwardAE = 2'b00;
            forwardBE = 2'b00;
            if(rsE != 0)
                begin
                    if(rsE == writeregM & regwriteM) forwardAE = 2'b10;
                    else if(rsE == writeregW & regwriteW) forwardAE = 2'b01;
                end
            if(rtE != 0)
                begin
                    if(rtE == writeregM & regwriteM) forwardBE = 2'b10;
                    else if(rtE == writeregW & regwriteW) forwardBE = 2'b01;
                end
        end
    ////////////////data stop/////////////////
    assign lwstallD = memtoregE & (rtE == rsD | rtE == rtD);
    assign branchstallD = branchD &
                (regwriteE &
                (writeregE == rsD | writeregE == rtD) |
                memtoregM &
                (writeregM == rsD | writeregM == rtD));
    assign  stallD = lwstallD | branchstallD;
    assign  stallF = stallD;
    assign  flushE = stallD;
endmodule
```

C Controller 代码

```
module controller(
    input clk,rst,
    ////////////////decode/////////////////
    input [5:0] opD,functD,
    output pcsrcD,branchD,equalD,jumpD,
    ////////////////execute///////////////
    input flushE,
    output memtoregE,alusrcE,regdstE,regwriteE,
    output [2:0] alucontrolE,
```

```
    ////////////////memory/////////////////
    output memtoregM,memwriteM,regwriteM,
    ////////////////write back/////////////
    output memtoregW,regwriteW
);

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

    wire memwriteE;
    maindec maindec(opD,memtoregD,memwriteD,branchD,alusrcD,regdstD,regwriteD,
        jumpD,aluopD);
    aludec  aludec(functD,aluopD,alucontrolD);
    assign pcsrcD = branchD & equalD;

    floprc #(1) r1E(clk,rst,flushE,memtoregD,memtoregE);
    floprc #(1) r2E(clk,rst,flushE,memwriteD,memwriteE);
    floprc #(1) r3E(clk,rst,flushE,alusrcD,alusrcE);
    floprc #(1) r4E(clk,rst,flushE,regdstD,regdstE);
    floprc #(1) r5E(clk,rst,flushE,regwriteD,regwriteE);
    floprc #(3) r6E(clk,rst,flushE,alucontrolD,alucontrolE);

    flopr #(1) r1M(clk,rst,memtoregE,memtoregM);
    flopr #(1) r2M(clk,rst,memwriteE,memwriteM);
    flopr #(1) r3M(clk,rst,regwriteE,regwriteM);

    flopr #(1) r1W(clk,rst,memtoregM,memtoregW);
    flopr #(1) r2W(clk,rst,regwriteM,regwriteW);
endmodule
```