《计算机组成原理》实验报告

年级、专业、班级	2022 级信息安全 01 班,2022 级信息安全 02 班	**姓名**	杨小艺,姚凡
实验题目	实验四简单五级流水线 CPU		
实验时间	2024 年 5 月 26 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: 谭玉娟

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 6 月 3 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

在流水线 CPU 中，并不是能够完全实现并行执行。在单周期中由于每条指令执行完毕才会执行下一条指令，并不会遇到冒险问题，而在流水线处理器中，由于当前指令可能取决于前一条指令的结果，但此时前一条指令并未执行到产生结果的阶段，这时候，就产生了冒险。

冒险分为：1. 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；2. 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 (由分支指令引起)。

2.1.2 数据冒险

如图 1 所示，and、or、sub 指令均需要使用 $s0 中的数据，然而 add 指令在回写阶段才能写入寄存器堆，此时后续三条指令均已经过或正在执行译码阶段，得到的结果均为错误值。以上就是数据冒险的特点.

数据冒险有以下解决方式：

1. 在编译时插入空指令；

2. 在编译时对指令执行顺序进行重排；

3. 在执行时进行数据前推；

4. 在执行时,暂停处理器当前阶段的执行,等待结果。

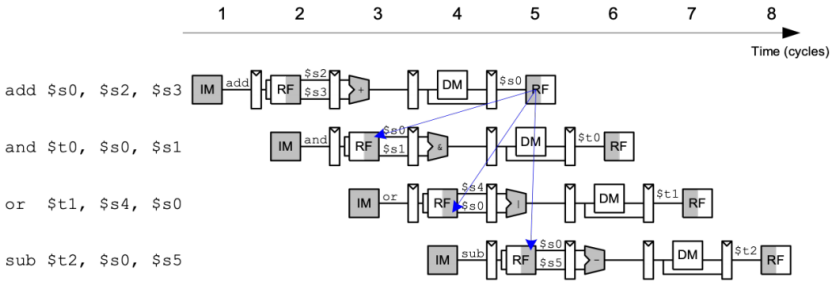

图 1: 数据冒险示例

由于我们未进行编译层的处理,需要在运行时 (run time) 进行解决,故采用 3、4 解决方案。

2.1.3 数据前推

如图 2 所示,add 指令的结果在 execute 阶段已经由 ALU 计算得到,此时可以将 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。

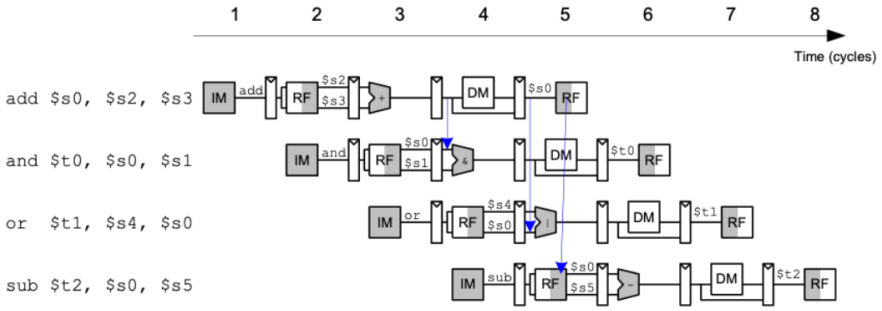

图 2: 数据前推示例

观察图 3, 在 execute 阶段需要判断当前输入 ALU 的地址是否与其他指令在此时执行的阶段要写入寄存器堆的地址相同,如果相同,就需要将其他指令的结果直接通过多路选择器输入到 ALU 中。

此处需要:

1. 增加 rs,rt 的地址传递到 execute 阶段,并与冒险模块连接;

2. Memory 阶段和 writeback 阶段要写入寄存堆的地址与冒险模块连接;

3. Memory 阶段和 writeback 阶段的寄存器堆写使能信号 regwrite 与冒险模块连接;

4. 根据实现逻辑,将生成的 forward 信号输出,控制 mux3 选择器。

数据通路结构如下:

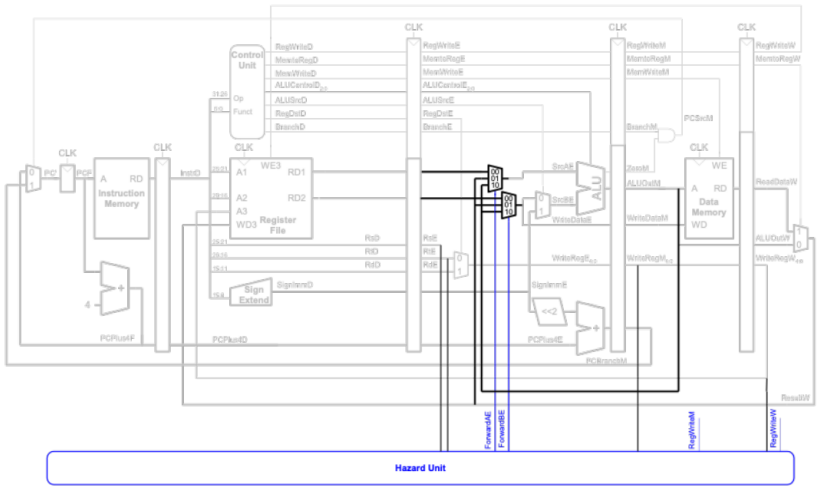

图 3: 加入前推的数据通路

2.1.4 流水线暂停

多数情况下，数据前推能解决很大一部分数据冒险的问题，然而在图中 4，lw 指令在 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。如图 5 在这种情况下，必须使流水线暂停，等待数据读取后，再前推到 execute 阶段。

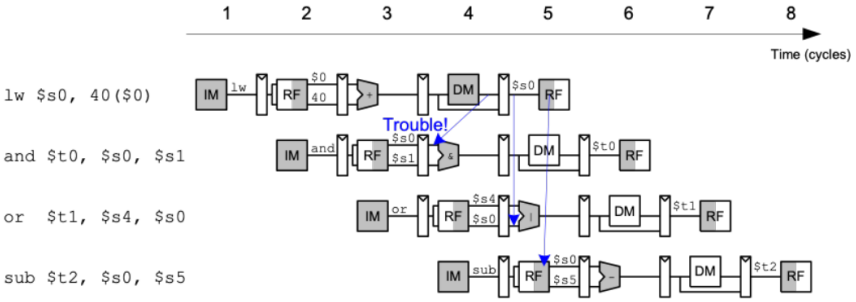

图 4: 数据无法前推的情况

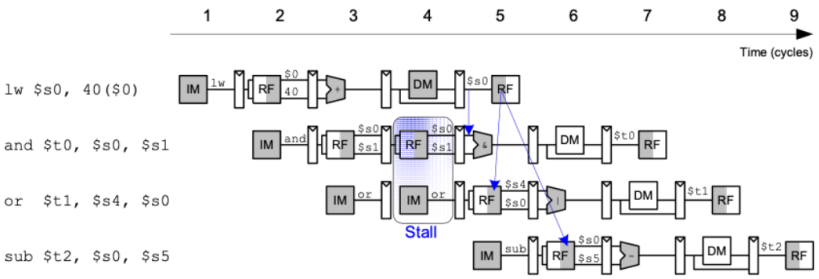

图 5: 流水线暂停示例

结合实现逻辑，需要完成下列功能:

1. 判断 decode 阶段 rs 或 rt 的地址是否是 lw 指令要写入的地址；

2. 设置 PC、fetch->decode 阶段触发器的暂停信号 (触发器使能端 disable)；

3. Decode->exexcute 阶段触发器清除 (避免后续阶段的执行，等待完成后方可继续执行后续阶段)。

数据通路结构如下：

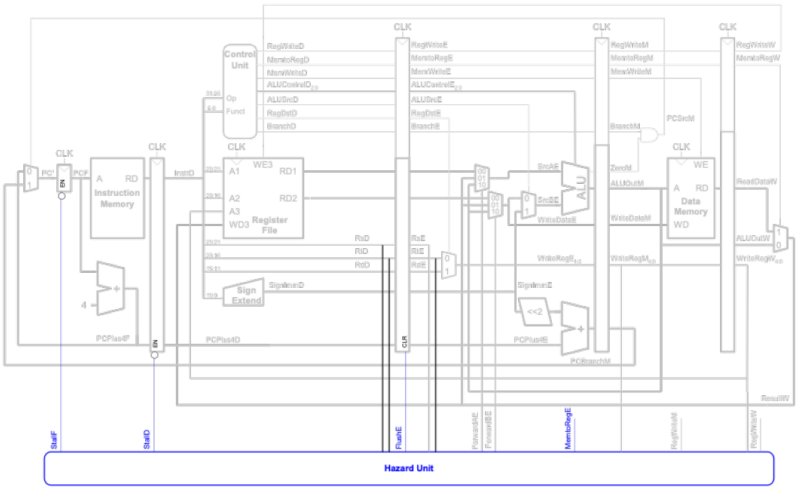

图 6: 加入暂停的数据通路结构

2.1.5 控制冒险

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；而此时，前三个阶段已经导致三条指令进入流水线开始执行，这时需要将这三条指令产生的影响全部清除。

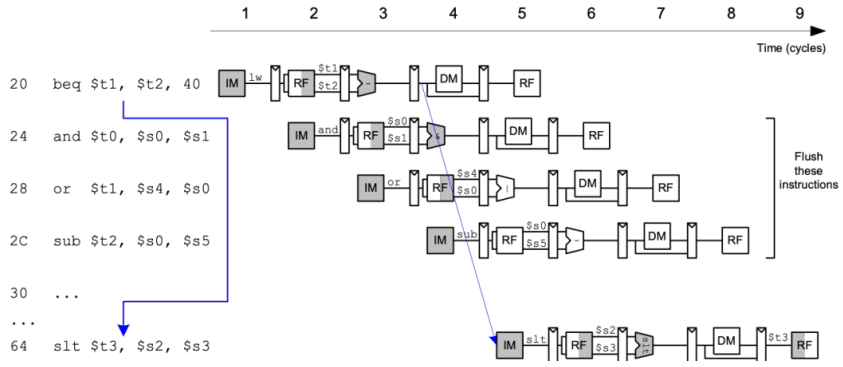

图 7: 控制冒险示例

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，如图 8；

此时又产生了数据冲突问题，需要增加数据前推和流水线暂停模块，如图 9；

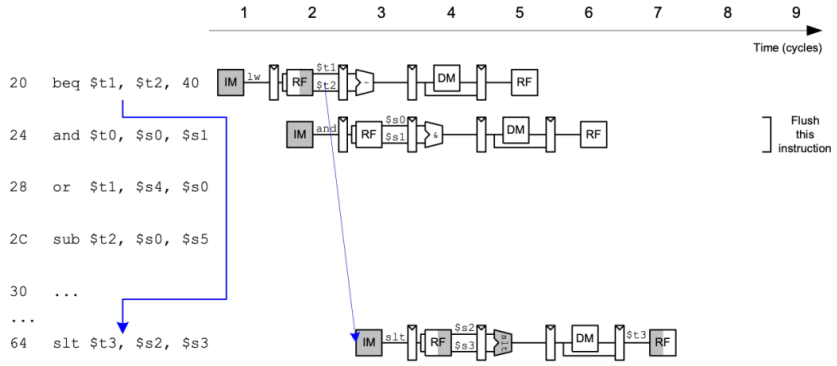

图 8: 提前判断分支

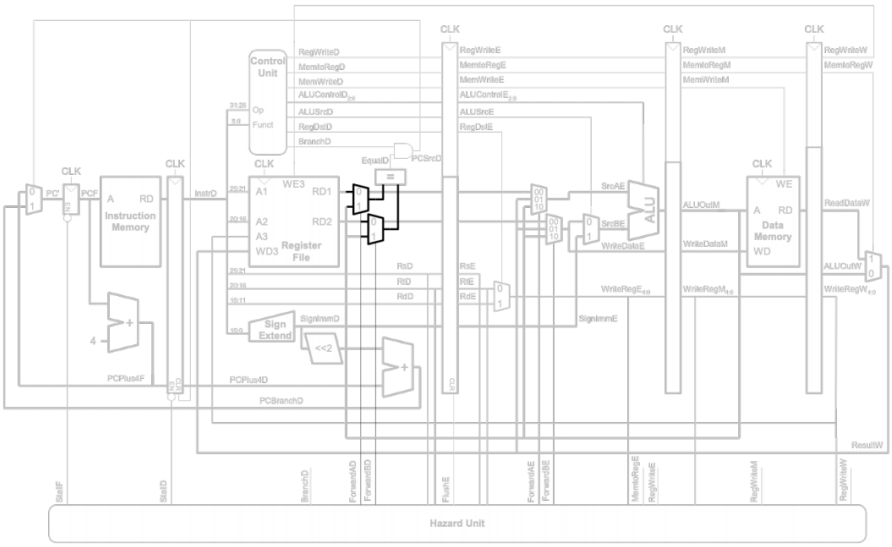

图 9: 解决控制冒险的数据通路

2.1.6 接口定义

表 1: 接口定义模版

信号名	方向	位宽	功能描述
rst	input	1-bit	重置信号
rsD	input	5-bit	decoder 级的 rs 信号
rtD	input	5-bit	decoder 级的 rt 信号
rsE	input	5-bit	excute 级的 rs 信号
rtE	input	5-bit	excute 级的 rt 信号
branchD	input	1-bit	decoder 级提前判断分支信号
regwriteE	input	1-bit	excute 级的 rs 信号
regwriteM	input	1-bit	memory access 级的 rs 信号
regwriteW	input	1-bit	writeback 级的 rs 信号
memtoregE	input	1-bit	excute 级判断写寄存器堆的数据来源
memtoregM	input	1-bit	memory 级判断写寄存器堆的数据来源
writeregE	input	5-bit	excute 级的寄存器堆写地址
writeregM	input	5-bit	memory 级的寄存器堆写地址
writeregW	input	5-bit	writeback 级的寄存器堆写地址
forwardaE	output	2-bit	excute 级控制 mux3 选择 SrcA
forwardbE	output	2-bit	excute 级控制 mux3 选择 SrcB
forwardaD	output	2-bit	decoder 级控制 mux3 选择 SrcA
forwardbD	output	2-bit	decoder 级控制 mux3 选择 SrcB
stallF	output	1-bit	fetch 级暂停
satllD	output	1-bit	decode 级暂停
flushE	output	1-bit	清空 excute 流水线

3 实验过程记录

在整个实验过程中完成了以下工作:

1、设计 MIPS 指令,将其编写为 coe 文件,导入 vivado 使流水线正常工作;

2、实现流水线的五个阶段: 将 CPU 的执行过程划分为五个阶段, 包括指令获取、指令解码、执行操作、访问内存和写回结果。根据指令集架构和控制信号,设计每个阶段的硬件逻辑;

3、处理数据和控制依赖: 在流水线中, 数据和控制依赖可能会导致冲突和竞争条件。为了解

决这些问题,需要使用合适的前推、旁路和数据转发技术,以确保正确的数据和控制流;

4、设计控制逻辑: 根据指令集架构和流水线的特性, 设计控制逻辑来生成适当的控制信号。这些信号用于在不同阶段之间传递数据、控制流和状态信息;

5、进行功能仿真: 使用 Verilog 语言编写 CPU 的模块,并进行功能仿真。在仿真中,使用测试程序和测试向量来验证 CPU 的正确性,并检查输出结果是否符合预期;

4 实验结果及分析

4.1 仿真结果

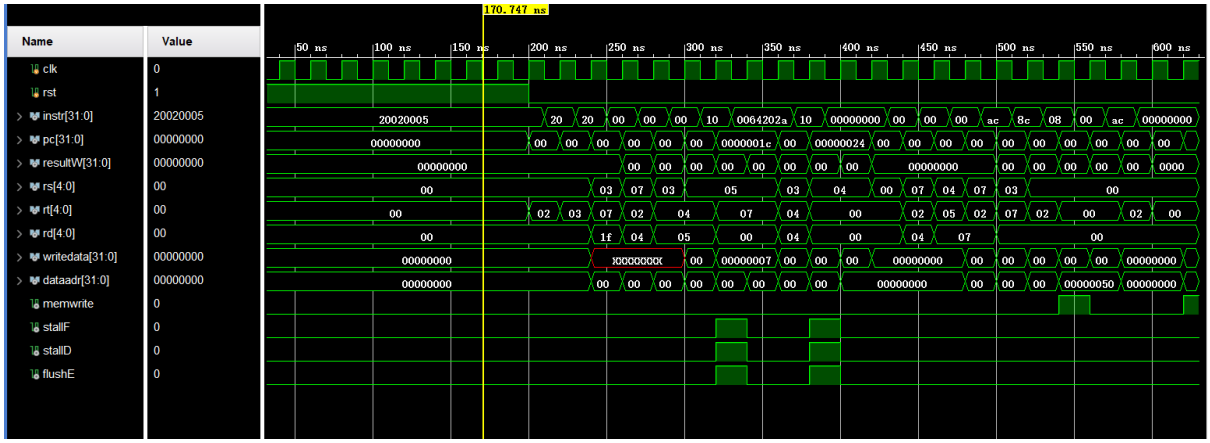

图 10: 仿真图

4.2 控制台输出结果

```
Block Memory Generator module testbench1.dut.inst_ram.inst.native_mem_modul
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module testbench1.dut.data_ram.inst.native_mem_mappe
Simulation succeeded
```

图 11: 控制台输出

A Datapath 代码

‘timescale 1ns / 1ps

```
module datapath(
input clk,rst,
input [31:0]instrD, readdata, // 数据存储器读出的数据
input regwriteE,
input regwriteM,
input regwriteW,
input memtoregW,
input memtoregE,
input memtoregM,
input [2:0]alucontrolE,
input alusrcE,
input regdstE,
input jumpD,
input branchD,
output wire [31:0] pc, aluoutM, memWriteData, resultW,
output pcsrcD,
output wire stallF, stallD, flushE
);
//分别为:pc+4, 多路选择分支之后的 pc, 下一条真正要执行的指令的 pc
wire [31:0] pcbranched, pcrealnext;
//ALU 数据来源 A、B,寄存器堆写入数据,左移 2 位后的立即数,
wire [31:0] ALUsrcA, ALUsrcB1, ALUsrcB2, sl2imm, sl2jaddr,
jumpaddr;
// resultW,
// Fetch phase
wire [31:0] pc4F;
// Decode phase
```

// $pc_4 : pc + 4, pcbranch : pc + 4 + imm << 2$

wire [31:0] pcF, $pc_4D, pcbranchD, rd1D, rd2D, extend_immD$;

wire [4:0] rsD, rtD, rdD;

//wire pcsrcD;

// Execute phase

wire [31:0] $pc_4E, rd1E, rd2E, extend_immE, aluoutE, writedataE$;

wire [4:0] rsE, rtE, rdE, writeregE; // 写入寄存器堆的地址

// Mem phase

wire [31:0] writedataM;

// aluoutM;

wire [4:0] writeregM;

// WB phase

wire [31:0] aluoutW, readdataW;

wire [4:0] writeregW;

// hazard

wire [1:0] forwardAE, forwardBE;

wire forwardAD, forwardBD;

// wire stallF, stallD, flushE;

wire equalD;

wire [31:0] equalsrc1, equalsrc2;

// ——————————————————-

// Fetch

//pc

flopenr (32) pc_module(

.clk(clk),

.rst(rst),

.en(stallF),

```
.d(pc_realnext),
.q(pc)
);
assign pcF = pc;
//PC+4 加法器
adder pc4adder(
.a(pcF),
.b(32'h4),
.y(pc4F)
);
// ————————————————————-
// fetech to decode memory flops
// pc4
flopenrc (32) FDpc4(
.clk(clk),
.rst(rst),
.en( stallD),
.clear(pcsrcD),
.d(pc4F),
.q(pc4D)
);
// ————————————————————-
// Decode
//jump 指令的左移 2 位
sl2 sl22(
.a(6'b0, instrD[25:0]),
.y(sl2jaddr)
```

```
);
assign jump_addr = pc4D[31 : 28], sl2_jaddr[27 : 0];
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];
//寄存器堆
regfile regfile(
.clk( clk),
.rst(rst),
.we3(regwriteW),
.ra1(instrD[25:21]),
.ra2(instrD[20:16]),
.wa3(writeregW),
.wd3(resultW),
.rd1(rd1D),
.rd2(rd2D)
);
mux2 (32) mux_equalsrc1(
.a(aluoutM),
.b(rd1D),
.s(forwardAD),
.y(equalsrc1)
);
mux2 (32) mux_equalsrc2(
.a(aluoutM),
.b(rd2D),
.s(forwardBD),
```

```
.y(equalsrc2)
);
assign equalD = (equalsrc1 == equalsrc2);
assign pcsrcD = branchD  equalD;//
//符号拓展
signext sign_extend(
.a(instrD[15:0]),
.y(extend_immD)
);
//立即数的左移 2 位
sl2 sl2_1(
.a(extend_immD),
.y(sl2_imm)
);
//branch 跳转地址加法器
adder pc_branch_adder(
.a(pc_4D),
.b(sl2_imm),
.y(pcbranchD)
);
//mux, PC 指向选择, PC+4(0), pc_src(1)
// pc_branched : jump
mux2 (32) mux_pcbranch(
.a(pcbranchD),//来自数据存储器
.b(pc_4F), //
.s(pcsrcD),//=0 , y=PC+4 ; =1 , y=pcbranchD 分支地址
.y(pc_branched)
```

```
);
//mux, 选择分支之后的 pc 与 jump_addr
mux2 (32) mux_pcnext(
.a(jump_addr),
.b(pc_branched),
.s(jumpD),//=0, y=pcbranchD ;=1, jump_addr
.y(pc_realnext)
);
// ————————————————————-
// decode to execution flops
// rd1
floprc (32) DE_rd1(
.clk(clk),
.rst(rst),
.clear(flushE),
.d(rd1D),
.q(rd1E)
);
// rd2
floprc (32) DE_rd2(
.clk(clk),
.rst(rst),
.clear(flushE),
.d(rd2D),
.q(rd2E)
);
// rs, rt, rd
```

floprc (15) DE$_r t_r d$(

.clk(clk),

.rst(rst),

.clear(flushE),

.d(rsD, rtD, rdD),

.q(rsE, rtE, rdE)

);

// extend$_i mm$

floprc (32) DE$_i mm$(

.clk(clk),

.rst(rst),

.clear(flushE),

.d(extend$_i mmD$),

.q(extend$_i mmE$)

);

// ————————————————-

// Exe

//处理数据冒险

// ALU,A 端输入值，rd1E(00),resultW(01)，aluoutM(10)

mux3 (32) mux$_A LUAsrc$(

.a(rd1E),

.b(resultW),

.c(aluoutM),

.s(forwardAE),

.y(ALUsrcA)

);

// ALU, B 端输入值，rd1E(00),resultW(01)，aluoutM(10)

```
mux3 (32) mux_ALUBsrc1(
.a(rd2E),
.b(resultW),
.c(aluoutM),
.s(forwardBE),
.y(ALUsrcB1)
);
mux2 (32) mux_ALUBsrc2(
.a(extend_immE),
.b(ALUsrcB1),
.s(alusrcE),
.y(ALUsrcB2) // B 输入第二个选择器之后的结果 //是否是立即数指令来控制，如果是立即数指令，那么就选择扩展立即数。
);
//ALU
alu alu(
.a(ALUsrcA),
.b(ALUsrcB2),
.op(alucontrolE),
.res(aluoutE)
);
assign writedataE = ALUsrcB1; // B 输入第一个选择器之后的结果
// 寄存器堆写入地址 writereg
mux2 (5) mux_WA3(
.a(rdE), //instr[15:11]
.b(rtE), //instr[20:16]
.s(regdstE),
```

```
.y(writeregE)
);
// ———————————————————-
// execution to Mem flops
// aluout
flopenr (32) EM_aluout(
.clk(clk),
.rst(rst),
.en(1'b1),
.d(aluoutE),
.q(aluoutM)
);
// writedata
flopenr (32) EM_writedata(
.clk(clk),
.rst(rst),
.en(1'b1),
.d(writedataE),
.q(writedataM)
);
// writereg
flopenr (5) EM_writereg(
.clk(clk),
.rst(rst),
.en(1'b1),
.d(writeregE),
.q(writeregM)
```

```
);
// ———————————————————-
// Mem
assign mem_writeData = writedataM;
// ———————————————————-
// Mem to wb flops
// aluout
flopenr (32) MF_aluout(
.clk(clk),
.rst(rst),
.en(1’b1),
.d(aluoutM),
.q(aluoutW)
);
// readdata from data memory
flopenr (32) MF_readdata(
.clk(clk),
.rst(rst),
.en(1’b1),
.d(readdata),
.q(readdataW)
);
// writereg
flopenr (5) MW_writereg(
.clk(clk),
.rst(rst),
.en(1’b1),
```

```
.d(writeregM),
.q(writeregW)
);
// ——————————————————-
// Write Back
//mux, 寄存器堆写入数据来自存储器 or ALU , memtoReg
mux2 (32) mux_WD3(
.a(readdataW),//来自数据存储器
.b(aluoutW),//来自 ALU 计算结果
.s(memtoregW),
.y(resultW)
);
// ——————————————————-
// hazard
hazard hazard(
.rsD(rsD),
.rtD(rtD),
.rsE(rsE),
.rtE(rtE),
.writeregE(writeregE),
.writeregM(writeregM),
.writeregW(writeregW),
.regwriteE(regwriteE),
.regwriteM(regwriteM),
.regwriteW(regwriteW),
.memtoregE(memtoregE),
.memtoregM(memtoregM),
```

```
.branchD(branchD),
.forwardAE(forwardAE),
.forwardBE(forwardBE),
.forwardAD(forwardAD),
.forwardBD(forwardBD),
.stallF(stallF),
.stallD(stallD),
.flushE(flushE)
);
endmodule
```

B Hazard 代码

```
‘timescale 1ns / 1ps
module hazard(input [4:0] rsD,
input [4:0] rtD,
input [4:0] rsE,
input [4:0] rtE,
input [4:0] writeregE,
input [4:0] writeregM,
input [4:0] writeregW,
input regwriteE,
input regwriteM,
input regwriteW,
input memtoregE,
input memtoregM,
input branchD,
```

```
output [1:0] forwardAE,
output [1:0] forwardBE,
output forwardAD,
output forwardBD,
output wire stallF, stallD, flushE
);
// ————————————————
// 数据冒险
// forward
assign forwardAE = ((rsE != 0)  (rsE == writeregM)  regwriteM) ? 2'b10 :
((rsE != 0)  (rsE == writeregW)  regwriteW) ? 2'b01 : 2'b00;
assign forwardBE = ((rtE != 0)  (rtE == writeregM)  regwriteM) ? 2'b10 :
((rtE != 0)  (rtE == writeregW)  regwriteW) ? 2'b01 : 2'b00;
assign forwardAD = ((rsD != 0)  (rsD == writeregM)  regwriteM);
assign forwardBD = ((rtD != 0)  (rtD == writeregM)  regwriteM);
// ————————————————
// stall
wire lwstall;
//stallF, stallD, flushE;
wire branchstall;
assign lwstall = ((rsD == rtE) || (rtD == rtE))  memtoregE; // . 判断 decode 阶段 rs 或 rt 的地址是否是 lw 指令要写入的地址;
assign branchstall = branchD  regwriteE
(writeregE == rsD || writeregE == rtD) ||
branchD  memtoregM
(writeregM == rsD || writeregM == rtD);
assign stallF = lwstall || branchstall;
```

```
assign stallD = lwstall || branchstall;
assign flushE = lwstall || branchstall;
endmodule
```

C Controller 代码

```
‘timescale 1ns / 1ps
module controller(input [5:0] op,
input [5:0] funct,
output regwriteD,
output memtoregD,
output memwriteD,
output branchD,
output [2:0]alucontrolD,
output alusrcD,
output regdstD,
output jumpD);
wire [1:0]aluop;
main_decodermain_decoder(
.op(op),
.regdst(regdstD),
.regwrite(regwriteD),
.alusrc(alusrcD),
.memwrite(memwriteD),
.memtoreg(memtoregD),
.branch(branchD),
.jump(jumpD),
```

```
.aluop(aluop)
);
aludec aludec(
.funct(funct),
.aluop(aluop),
.alucontrol(alucontrolD)
);
endmodule
```