**实验三**

datapath

`timescale 1ns / 1ps

module datapath(

input clk,rst,

input [31:0]inst, reg\_WriteData,

input jump,

input pcsrc,

input alusrc,

input memtoreg,

input regwrite,

input regdst,

input [2:0] alucontrol,

output zero,

output [31:0] pc, aluout, mem\_WriteData, wd3

);

//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, 寄存器堆输出1, 寄存器堆输出2, 立即数符号拓展后的结果

wire [31:0] pc\_4, pc\_branched, pc\_realnext, rd1, rd2, extend\_imm;

//ALU计算结果，寄存器堆写入数据，左移2位后的立即数，跳转地址

wire [31:0] ALUsrcB, wd3, sl2\_imm, pc\_Branch, sl2\_inst, pc\_jump;

//写入寄存器堆的地址

wire [4:0] reg\_WriteNumber;

assign mem\_WriteData = rd2;

assign pc\_jump = {pc[31:28], sl2\_inst[27:0]};

//PC+4加法器

adder pc\_4\_adder (

.a(pc),

.b(32'h4),

.y(pc\_4)

);

//mux, PC指向选择, PC+4(0), pc\_src(1)

mux2 #(32) mux\_pcbranch(

.a(pc\_Branch),//来自数据存储器

.b(pc\_4),//来自ALU计算结果

.s(pcsrc),

.y(pc\_branched)

);

//pc

pc\_module pc\_module(

.clk(clk),

.rst(rst),

.d(pc\_realnext),

.pc(pc)

);

//立即数的左移2位

sl2 sl2\_1(

.a(extend\_imm),

.y(sl2\_imm)

);

//jump指令的左移2位

sl2 sl2\_2(

.a({6'b0, inst[25:0]}),

.y(sl2\_inst)

);

//mux, 选择分支之后的pc与pc\_jump

mux2 #(32) mux\_pcnext(

.a(pc\_jump),//来自数据存储器

.b(pc\_branched),//来自ALU计算结果

.s(jump),

.y(pc\_realnext)

);

//branch跳转地址加法器

adder pc\_branch\_adder (

.a(pc\_4),

.b(sl2\_imm),

.y(pc\_Branch)

);

//符号拓展

signext sign\_extend(

.a(inst[15:0]),

.y(extend\_imm)

);

//mux,寄存器堆写入数据来自存储器 or ALU ，memtoReg

mux2 #(32) mux\_WD3(

.a(reg\_WriteData),//来自数据存储器

.b(aluout),//来自ALU计算结果

.s(memtoreg),

.y(wd3)

);

//mux,寄存器堆写入地址rt or rd，RegDst

mux2 #(5) mux\_WA3(

.a(inst[15:11]),//rt

.b(inst[20:16]),//rd

.s(regdst),

.y(reg\_WriteNumber)

);

//寄存器堆

regfile regfile(

.clk(clk),

.we3(regwrite),

.ra1(inst[25:21]),

.ra2(inst[20:16]),

.wa3(reg\_WriteNumber),

.wd3(wd3),

.rd1(rd1),

.rd2(rd2)

);

//mux,ALU B端输入值，rd2(0),imm(1)，alusrc

mux2 #(32) mux\_ALUsrc(

.a(extend\_imm),//立即数

.b(rd2),//寄存器堆

.s(alusrc),

.y(ALUsrcB)

);

//ALU

alu alu(

.a(rd1),

.b(ALUsrcB),

.op(alucontrol),

.res(aluout),

.zero(zero)

);

endmodule

**sl2**

`timescale 1ns / 1ps

module sl2(input [31:0] a,

output [31:0] y);

// shift left by 2

assign y = {a[29:0],2'b00};

endmodule

**signext**

`timescale 1ns / 1ps

module signext (input [15:0] a,

output [31:0] y);

assign y = {{16{a[15]}}, a};

endmodule

**controller**

`timescale 1ns / 1ps

module controller(input [5:0] op,

input [5:0] funct,

output regdst,

output regwrite,

output alusrc,

output memwrite,

output memtoreg,

output branch,

output jump,

output [2:0]alucontrol);

wire [1:0]aluop;

main\_decoder main\_decoder(

.op(op),

.regdst(regdst),

.regwrite(regwrite),

.alusrc(alusrc),

.memwrite(memwrite),

.memtoreg(memtoreg),

.branch(branch),

.jump(jump),

.aluop(aluop)

);

aludec aludec(

.funct(funct),

.aluop(aluop),

.alucontrol(alucontrol)

);

Endmodule

# 实验四

datapath

`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, mem\_WriteData, 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] pc\_branched, pc\_realnext;

//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, sl2\_imm, sl2\_j\_addr, jump\_addr;

// resultW,

// Fetch phase

wire [31:0] pc\_4F;

// 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 pc\_4\_adder (

.a(pcF),

.b(32'h4),

.y(pc\_4F)

);

// ----------------------------------------

// fetech to decode memory flops

// pc\_4

flopenrc #(32) FD\_pc\_4 (

.clk(clk),

.rst(rst),

.en(~stallD),

.clear(pcsrcD),

.d(pc\_4F),

.q(pc\_4D)

);

// ----------------------------------------

// Decode

//jump指令的左移2位

sl2 sl2\_2(

.a({6'b0, instrD[25:0]}),

.y(sl2\_j\_addr)

);

assign jump\_addr = {pc\_4D[31:28], sl2\_j\_addr[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: 用来跟jump地址再选一次

mux2 #(32) mux\_pcbranch(

.a(pcbranchD),//来自数据存储器

.b(pc\_4F),//来自加法器计算结果

.s(pcsrcD),

.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),

.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\_rt\_rd (

.clk(clk),

.rst(rst),

.clear(flushE),

.d({rsD, rtD, rdD}),

.q({rsE, rtE, rdE})

);

// extend\_imm

floprc #(32) DE\_imm (

.clk(clk),

.rst(rst),

.clear(flushE),

.d(extend\_immD),

.q(extend\_immE)

);

// ----------------------------------------

// Exe

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

mux3 #(32) mux\_ALUAsrc(

.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

);

//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

**Hazard**

`timescale 1ns / 1ps

//处理冒险

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

**controller**

`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\_decoder main\_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