《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	实验三简易单周期 CPU 实验		
实验时间	2024 年 5 月 11 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)掌握不同类型指令在数据通路中的执行路径。

(2)掌握 Vivado 仿真方式。

报告完成时间: 2024 年 5 月 12 日

1 实验内容

阅读实验原理实现以下模块：

(1) Datapath，其中主要包含 alu(实验一已完成)，PC(实验二已完成)，adder、mux2、signext、sl2(其中 adder、mux2 数字逻辑课程已实现，signext、sl2 参见实验原理)，
(2) Controller(实验二已完成)，其中包含两部分，分别为 main_decoder，alu_decoder。
(3) 指令存储器 inst_mem(Single Port Ram)，数据存储器 data_mem(Single Port Ram)；使用 Block Memory Generator IP 构造指令，注意考虑 PC 地址位数统一。(参考实验二)
(4) 参照实验原理，将上述模块依指令执行顺序连接。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

2.1.1 功能描述

datapath 模块含许多模块，主要可以分为 PC 和 Data 两个部分 PC->PC+4-> 判断 branch(instr[15:0]«2+PC+4)-> 判断 jump({PCPlus4[31:28],instr[25:0]«2})-> 更新 PC

IR-寄存器堆->RD1、RD2-ALU->carry_result、WriteData-DataMem->ReadData、carry_result->Result 具体的连线图如下图所示：

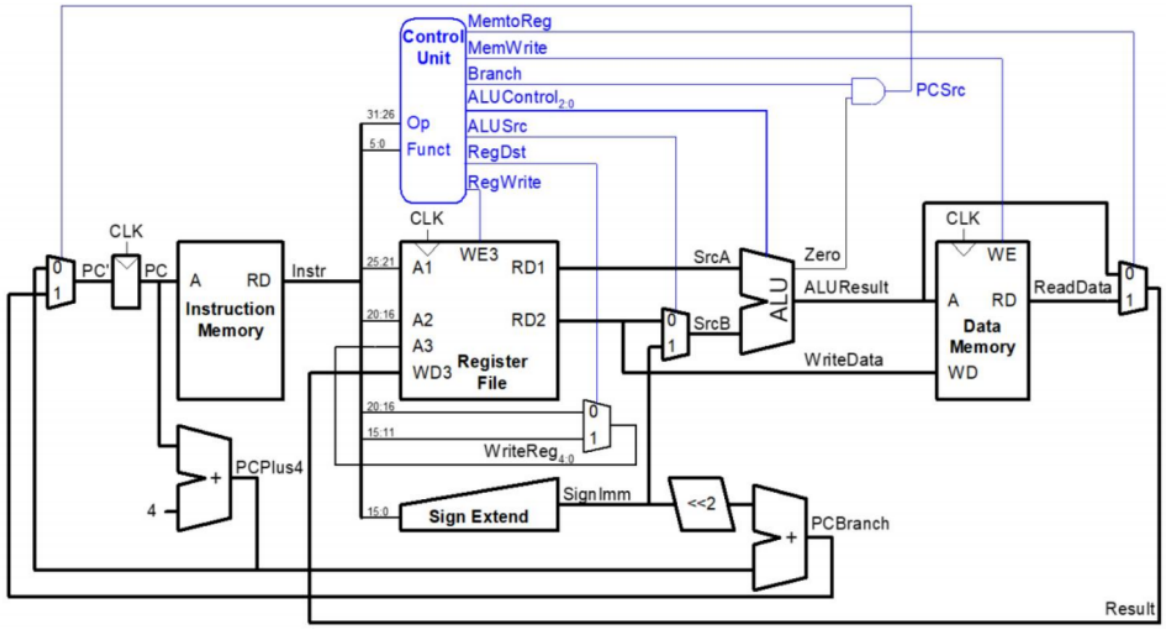

图 1: 具体连线图

2.2 存储器

2.2.1 指令存储器

指令存储器类型选择 Single Port ROM，端口宽度选择 32bits，深度选择 1024，如图 1。使能端设置为 Always Enabled，不设复位信号。具体端口参数选项如图 2。其他选项需加载 coe 文件，如图 3。参数选择完成后生成 ROM 如图 4。

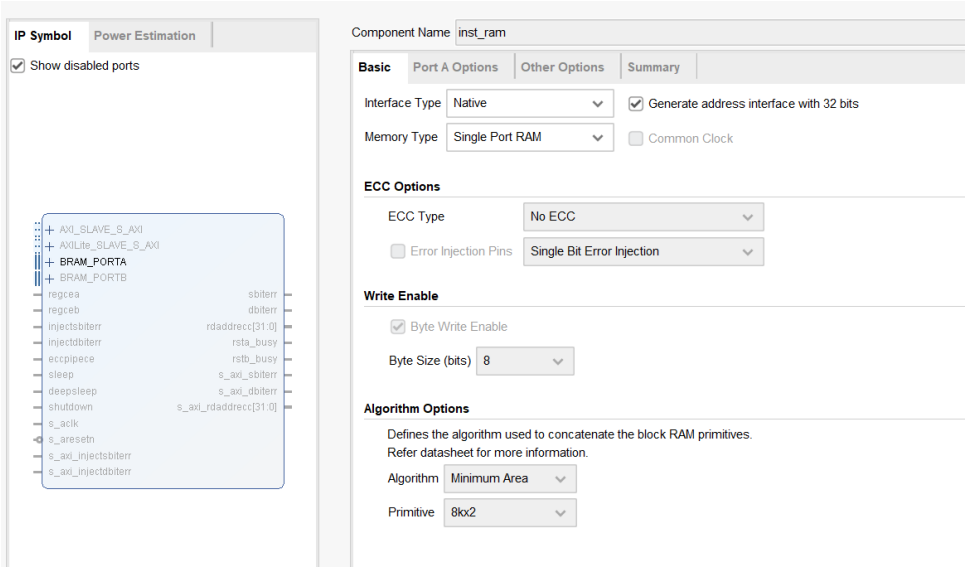

图 2: 指令存储器类型选择

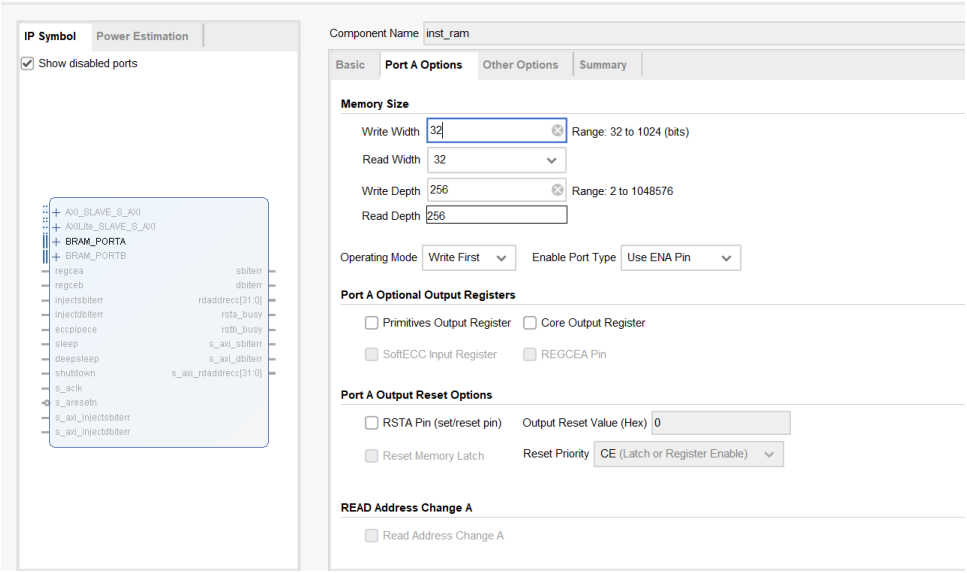

图 3: 指令存储器设置

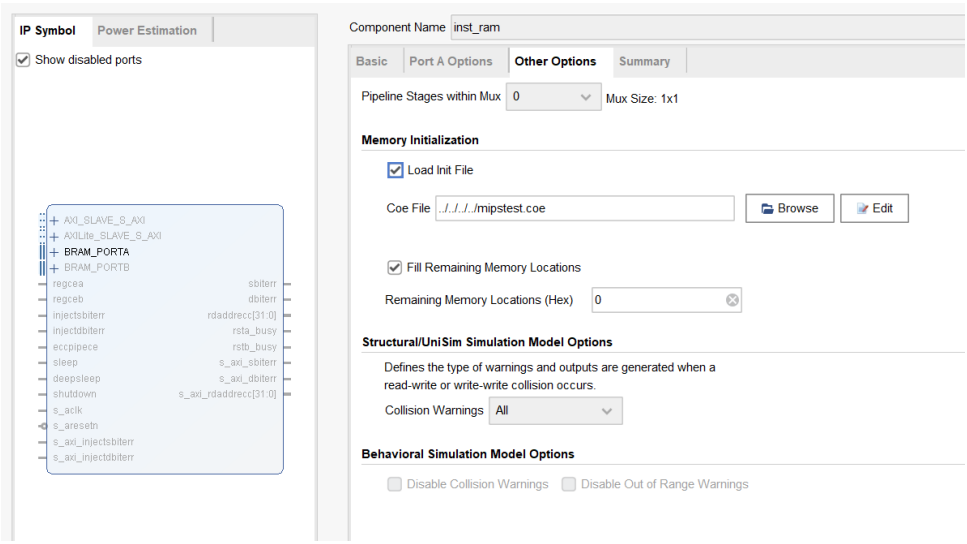

图 4: 其他设置

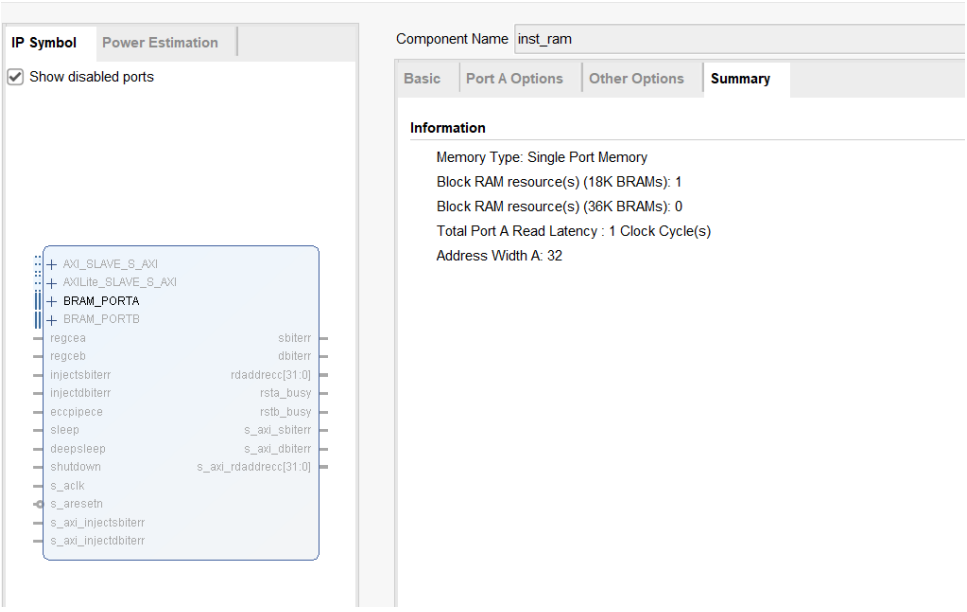

图 5: summary

2.2.2 数据存储器

数据存储器类型选择 Single Port RAM，端口宽度选择 32bits，深度选择 1024，如图 5。使能端设置为 Always Enabled，不设复位信号。具体端口参数选项如图 6。其他选项不必加载 coe 文件，如图 7。参数选择完成后生成 RAM 如图 8。

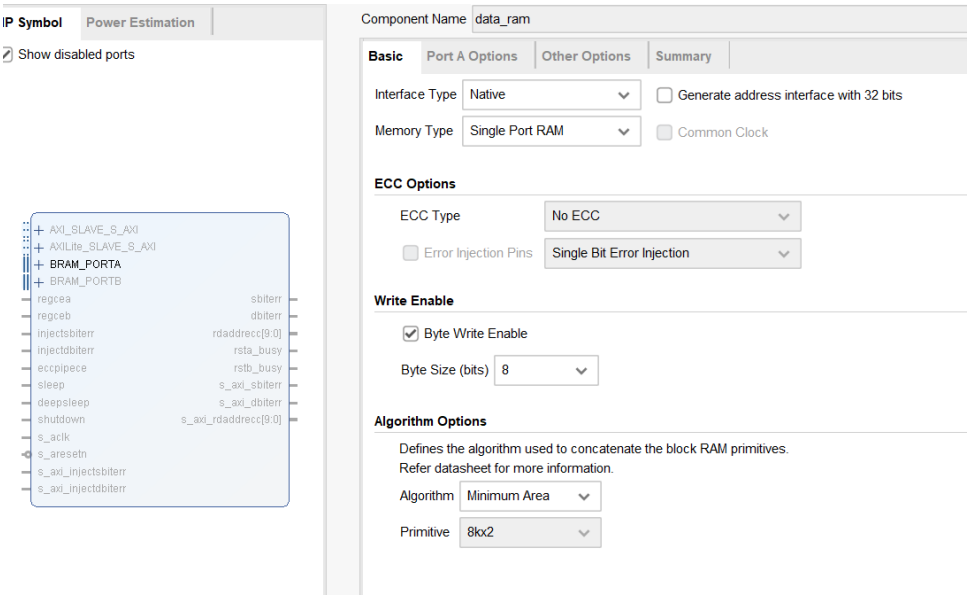

图 6: 数据存储类型选择

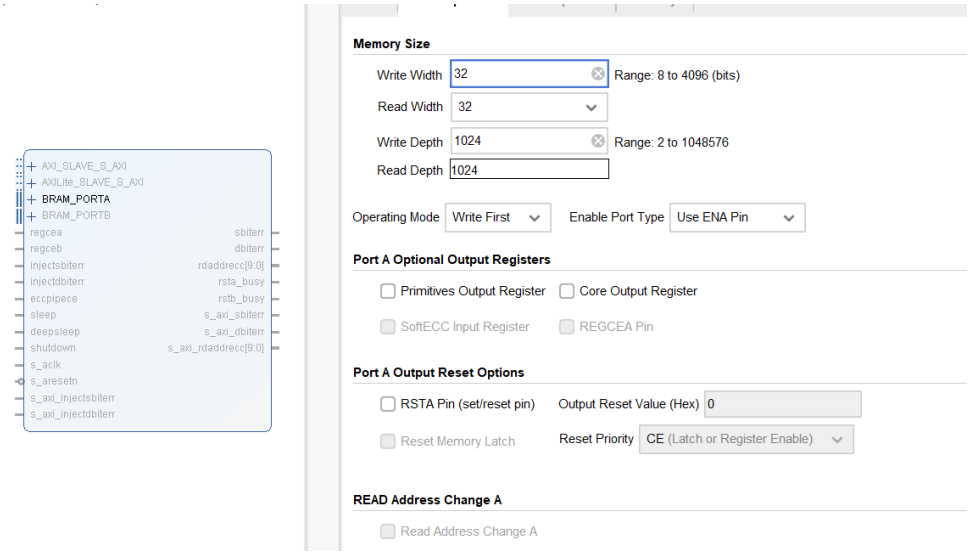

图 7: 数据存储使能设置

2.2.3 MIPS 处理

将 controller 模块和 datapath 模块连通。controller 为纯组合逻辑，不包含任何存储功能，datapath 为时序逻辑，需要时钟信号控制。因为本实验是单周期 CPU，将 datapath 视为一个整体 (由同一个时钟信号控制)，指令寄存器取指令、datapath 数据传输、data memory 存取数据都需要在同一个时钟周期内完成。因此 instruction memory 和 data memory 均采用下降沿触发，datapath 采

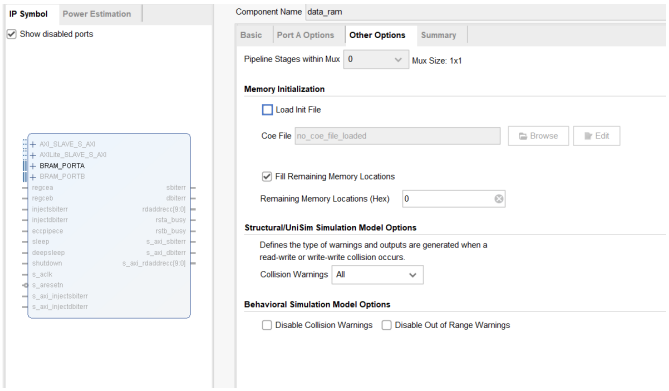

图 8: 数据存储其他设置

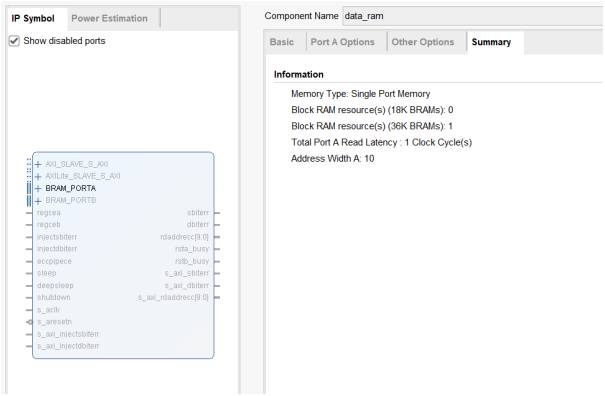

图 9: 数据存储 summary

用上升沿触发。当 clk 上升沿到来时, datapath 将传入的 instr 和各控制信号进行运算, 输出 pc、writedata 和 aluout; clk 下降沿来临时, data memory 将 writedata 写回 (sw)/aluout 取出 (lw)/aluout 传出 (R), 同时 Instruction memory 取指令传出; 等到下一个 clk 上升沿来临, datapath 接收传回的 readdata(lw)/aluout(R) 写回寄存器堆。

具体连接方式见下图:

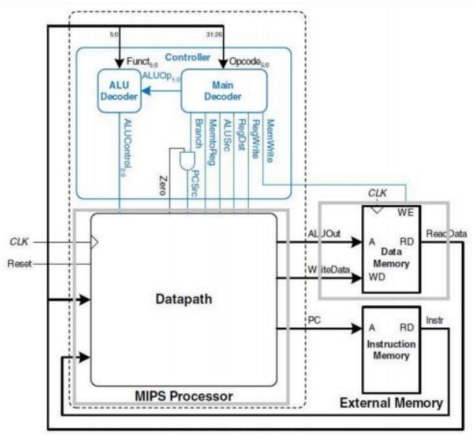

图 10: 连接图

2.3 接口定义

见表 1。

表 1: 接口定义模版

信号名	方向	位宽	功能描述
clk	Input	1-bit	时钟信号
rst	Input	1-bit	重置信号
alucontrol	Input	3-bits	ALU 控制信号
memtoreg	Input	1-bit	判断写回寄存器堆的是 ALU 的计算结果还是 lw 读取数据
alusrc	Input	1-bit	判断 SrcB 是 RD2 还是 SignImm
regdst	Input	1-bit	判断写寄存器
memwrite	Input	1-bit	数据寄存器的使能信号
pcsrc	Input	1-bit	判断是否执行 branch
jump	Input	1-bit	判断是否执行 jump
readdata	Input	32-bits	从数据存储器读出的数据
instr	Input	32-bits	指令
aluout	output	32-bits	ALU 运算结果
pc	output	32-bits	当前 PC 值
writedata	output	32-bits	sw 写进 Data memory 的数据
zero	output	1-bit	ALU 计算结果是否为 0
WA3	output	5-bit	写入 dataram 的地址

3 实验过程记录

3.1 实验过程

1. datapath 主要由 pc, adder, alu, regfile 等模块构成, 均已完成或以提供, 仅需要添加 mux 将各个端口连接即可;

2. controller 模块已在实验二实现, 将各个控制信号输入 datapath, 为 mux, regfile 提供控制信号;

3. inst_ram, data_ram 直接调用 ip 核, 由 datapath 输入 pc, dataadr 和 writedata 后, 将 readdata 和 inst 输入 datapath ;

4. 在 datapath 内部, 根据单周期的完整通路图, 一一连接 add、and、sub、or、slt、beq、j、lw、sw、

addi 等指令的数据通路即可;

5. 采用提供的 testbench.v, 把需要观察的信号从左侧添加到波形图中, 然后重新进行仿真 6. 调试完后无误则会执行最后一条 sw 指令, 将 $2 的 7 存到数据存储器的 [84], 并在控制台得到’ Simulation succeeded’。

3.1.1 pc 模块

基于实验二, 修改 pc 模块使 pc 信号输入可以实现 branch、jump 等功能。并根据 clk 输出 pc 的时序逻辑, 而将原本 pc 中所包含的 pc+4 功能转移到 adder 中, 如下图:

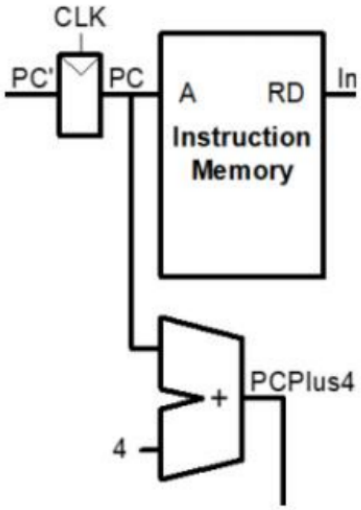

图 11: pc 原理图

3.1.2 alu 模块

将实验一中 ALU 模块其中的 8 位操作数改成 32 位操作数。

3.1.3 controller 模块

实验二原本的 controller 是没有关于接收 zero 信号的处理的, 这导致 controller 板块没有办法和 datapath 板块直接连接。在传入 datapath 之前设置一个 wire 型信号做 zero&branch 再传入 datapath。

3.2 问题和解决方案

问题描述 1:RAM 接受传入的 addr, 输出对应的数据, 使用了组合逻辑。

问题解决：组合逻辑只有逻辑门，即根据输入给出输出，没有记忆和存储功能。任意时刻的状态由该时刻的输入信号决定，无输入信号时则输出为空。储存器改用时序逻辑，添加 clk 控制触发器，信号边沿触发时改变状态。

问题描述 2：在仿真 datapath 时发现多处信号为 X 和 Z，连线也没问题，从 jump 信号向后就都出了问题。

问题解决：由于 datapath 模块信号较多，实例化模块时会出现遗漏参数、顺序错误等问题，采用实参对应形参的方式，并且保证实例化端口数量与设计文件一致后解决，

问题描述 3：对于 J 指令，总会跳转到错误地址。

问题解决：代码逻辑出问题，J 指令是 PC 高四位拼接 J 的 26 位地址，再左移 2 位得到。

4 实验结果及分析

控制台输出和仿真结果如图 12、图 13 所示，仿真结果与预期结果相同，控制台输出结尾显示 Simulation succeed。

4.1 控制台输出

```
# run 1000ns
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module sim.dut.imem.inst.native_mem_mapped_module.blk_mem_gen_v8_4_2_inst is using a behav
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module sim.dut.dmem.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral m
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sim_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:08 . Memory (MB): peak = 833.488 ; gain = 17.332
```

图 12: 控制台输出

4.2 仿真图输出结果

图 13: 仿真图

4.3 A Datapath 代码

```
module datapath(
input clk,
input rst,
input [31:0]instr,
input memtoreg,
input pcsrc,
input alusrc,
input branch,
input regdst,
input regwrite,
input jump,
input [2:0]alucontrol,
input [31:0]readdata,//dataram 的输出
output ena,
output overflow,
output [31:0]pc,//program counter
output [31:0]aluout,
output [31:0]writedata,//写入 dataram 的数据
output [31:0]dataaddr,
output [4:0]WA3//写入 dataram 的地址
);
// pc
wire [31:0] pc,PCPlus4;
adder pcplus(.a(pc),.b(32'h00000004),.c(PCPlus4));
pc programounter(.clk(clk), .RST(rst), .pc(pc), .insaddr(pc), .inste(ena));
// reg
```

```
//wire [4:0]WA3;
wire [31:0]RD1, RD2, WD3;
// mux to reg
defparam mux_regwrite.WIDTH = 5;
mux2 mux_regwrite(.zero(instr[20 : 16]), .one(instr[15 : 11]), .mux(regdst), .c(WA3));
regfile register(.clk(clk),.we3(regwrite),.ra1(instr[25:21]),.ra2(instr[20:16]),.wa3(WA3),.wd3(WD3),.rd1(RD1),.rd2
// SrcB
wire [31:0]Signlmm;
signext SignExtend(.constant(instr[15:0]),.sign_extend(Signlmm));
wire [31:0]SrcB;
mux2 mux_ALUsrc(.zero(RD2), .one(Signlmm), .mux(alusrc), .c(SrcB));
// ALU
wire zero;
alu ALU(.num1(RD1),.num2(SrcB),.alu_control(alucontrol), .result(aluout), .overflow(overflow), .zero(zer
//pcsrc
assign pcsrc = zero  branch;
// dataram
assign writedata = RD2;
assign dataaddr = aluout;
mux2 mux_regwriteData(.zero(aluout), .one(readdata), .mux(memtoreg), .c(WD3));
// PCbranch
wire [31:0]PCBranch, PCJump,PCSrc;
adder pcbranch(.a(Signlmm«2),.b(PCPlus4),.c(PCBranch));
assign PCJump = PCPlus4[31:28],instr[25:0]«2;
// mux pcsrc
mux2 mux_PCSrc(.zero(PCPlus4), .one(PCBranch), .mux(pcsrc), .c(PCSrc));
// mux jump
```

mux2 mux$_P CJump(.zero(PCSrc), .one(PCJump), .mux(jump), .c(pc_)$);

endmodule