《计算机组成原理》实验报告

年级、专业、班级	2019 级计算机科学与技术(卓越)02 班	**姓名**	李燕琴
实验题目	实验三简易单周期 CPU 实验		
实验时间	2021 年 05 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: 钟将

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: 2021 年 5 月 26 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

将 pc,pc_adder,regfile,alu,mux2,signext,sl2 等简单器件按照图 1 所示结构图，根据三类指令（R，I，J 指令）执行路径进行连接，换言之，进行大型的多功能组合逻辑（含部分时序逻辑）设计实现。

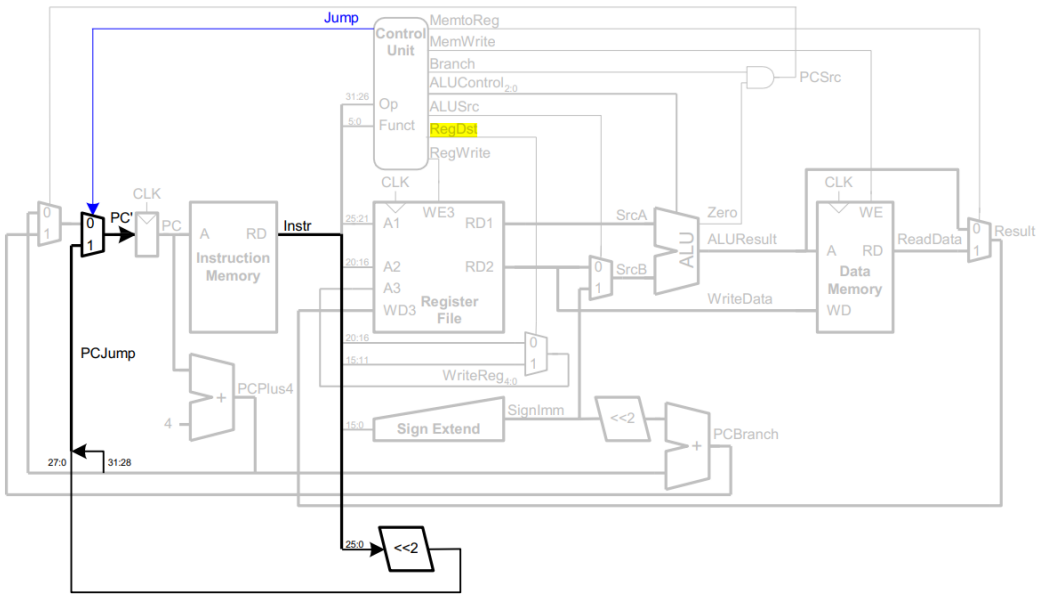

图 1: datapath RTL 设计图

2.1.2 接口定义

如表1所示。

表 1: datapath 接口定义

信号名	方向	位宽	功能描述
clk	Input	1-bit	内置时钟信号
rst	Input	1-bit	重置信号
regwrite	Output	1-bit	是否需要写寄存器堆
regdst	Output	1-bit	写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd
alusrc	Output	1-bit	送入 ALU B 端口的值是立即数的 32 位扩展 / 寄存器堆读取的值
branch	Output	1-bit	是否为 branch 指令
memWrite	Output	1-bit	是否需要写数据存储器
memtoReg	Output	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
jump	Output	1-bit	是否为 jump 指令
alucontrol	Output	3-bit	ALU 控制信号,代表不同的运算类型
instr	Input	32-bit	32 位的指令
data_ram_rdata	Input	32-bit	data_ram 中读取的 32 位数据
pc_now	Output	32-bit	当前 PC 位置
data_ram_waddr	Output	32-bit	data_ram 写入数据对应的地址
data_ram_wdata	Output	32-bit	data_ram 写入的数据值

2.2 存储器

2.2.1 功能描述

指令存储器 instr_ram: 在每一个时钟下降沿根据传入的 pc[9:2]（由于 PC 是自增 4,故按照 pc 取指时,需要从第 2 位看起)地址读取指令。

数据存储器 data_ram: 在每一个时钟上升沿,根据使能信号和地址信号,读取数据或写入数据。

2.2.2 参数设置

指令存储器参数设置如图2;其中,数据存储器参数设置和指令存储器类似,但无本地数据导入。最终设计的存储器接口如图3所示。

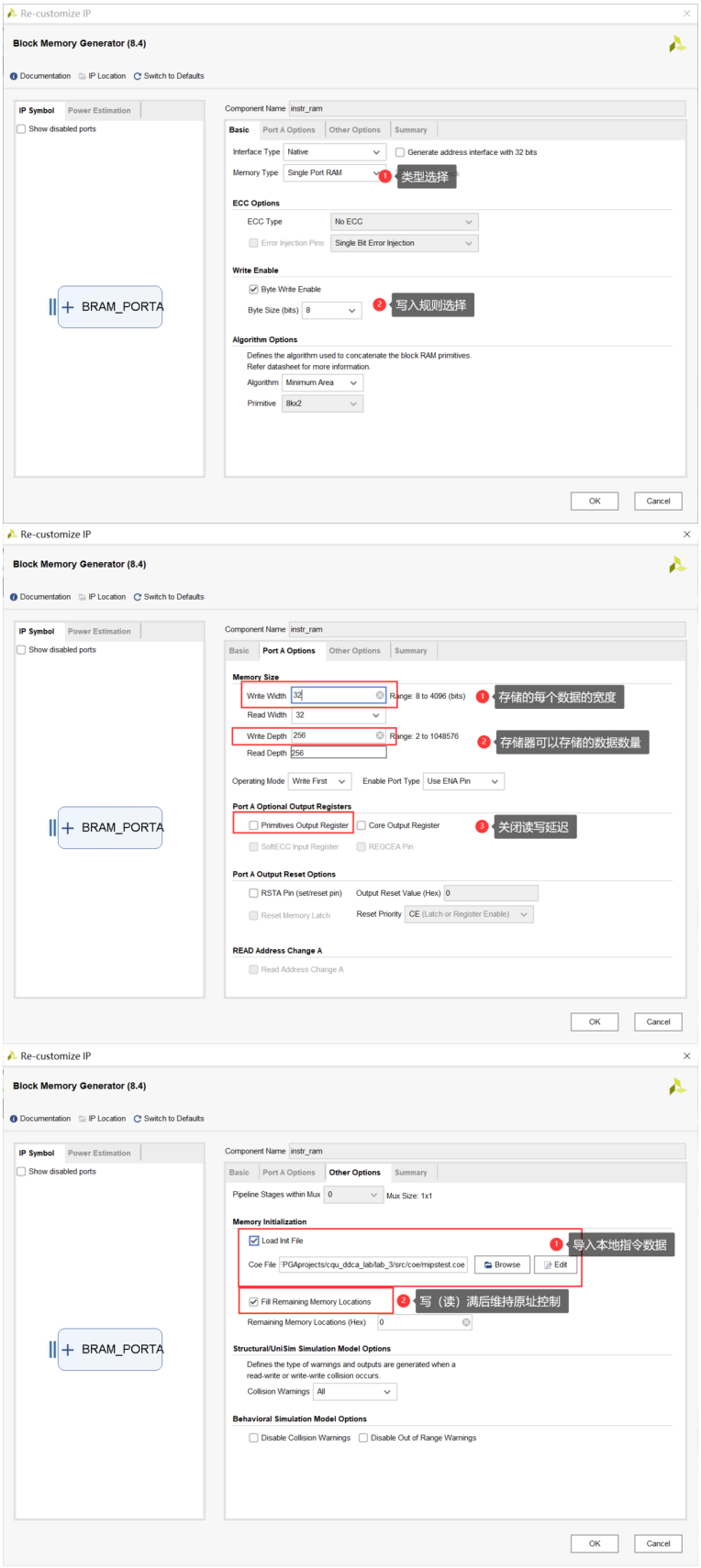
Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
Component Name
instr_ram
Basic
Port A Options
Other Options
Summary
Interface Type
Native
Generate address interface with 32 bits
Memory Type
Single Port RAM
1
类型选择
ECC Options
ECC Type
No ECC
Error Injection Pins
Single Bit Error Injection
Write Enable
Byte Write Enable
Byte Size (bits)
8
2
写入规则选择
Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm
Minimum Area
Primitive
8kx2
OK
Cancel
Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
Component Name
instr_ram
Basic
Port A Options
Other Options
Summary
Memory Size
Write Width
32
Range: 8 to 4096 (bits)
1
存储的每个数据的宽度
Read Width
32
Write Depth
256
Range: 2 to 1048576
2
存储器可以存储的数据数量
Read Depth
256
Operating Mode
Write First
Enable Port Type
Use ENA Pin
Port A Optional Output Registers
Primitives Output Register
Core Output Register
3
关闭读写延迟
SoftECC Input Register
REGCEA Pin
Port A Output Reset Options
RSTA Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change A
Read Address Change A
OK
Cancel
Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
BRAM_PORTA
Component Name
instr_ram
Basic
Port A Options
Other Options
Summary
Pipeline Stages within Mux
0
Mux Size: 1x1
Memory Initialization
Load Init File
1
导入本地指令数据
Coe File
PGAprojects/cqu_ddca_lab/lab_3/src/coe/mipstest.coe
Browse
Edit
Fill Remaining Memory Locations
2
写（读）满后维持原址控制
Remaining Memory Locations (Hex)
0
Structural/UniSim Simulation Model Options
Defines the type of warnings and outputs are generated when a
read-write or write-write collision occurs.
Collision Warnings
All
Behavioral Simulation Model Options
Disable Collision Warnings
Disable Out of Range Warnings
OK
Cancel

图 2: 指令存储器设置

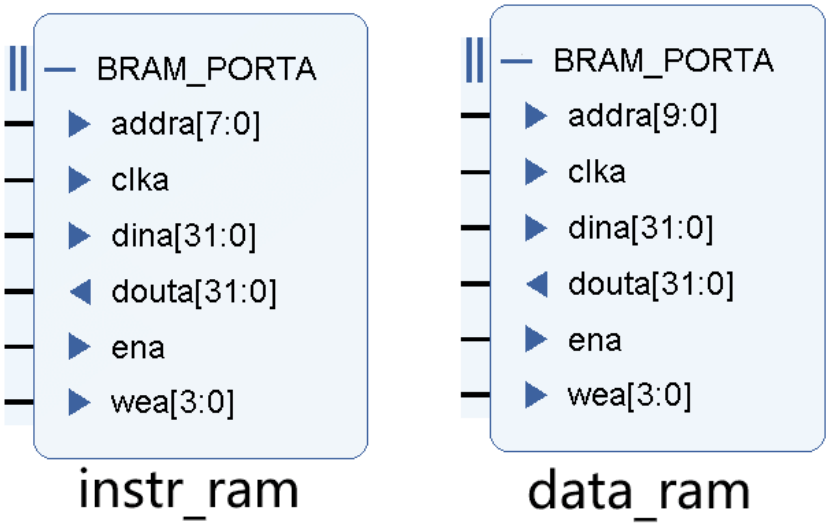

图 3: 存储器接口

2.3 顶层模块

2.3.1 功能描述

将 mips(即集成 controller 和 datapath 的内核模块),data_ram,instr_data 三个模块按照如图 4 所示的结构图,进行连接,打通整个数据通路。

2.3.2 接口定义

如表2所示。

表 2: 顶层模块接口定义

信号名	方向	位宽	功能描述
clk	Input	1-bit	内置时钟信号
rst	Input	1-bit	重置信号
data_ram_waddr	Output	32-bit	data_ram 写入数据对应的地址
data_ram_wdata	Output	32-bit	data_ram 写入的数据值
memWrite	Output	1-bit	是否需要写数据存储器

3 实验过程记录

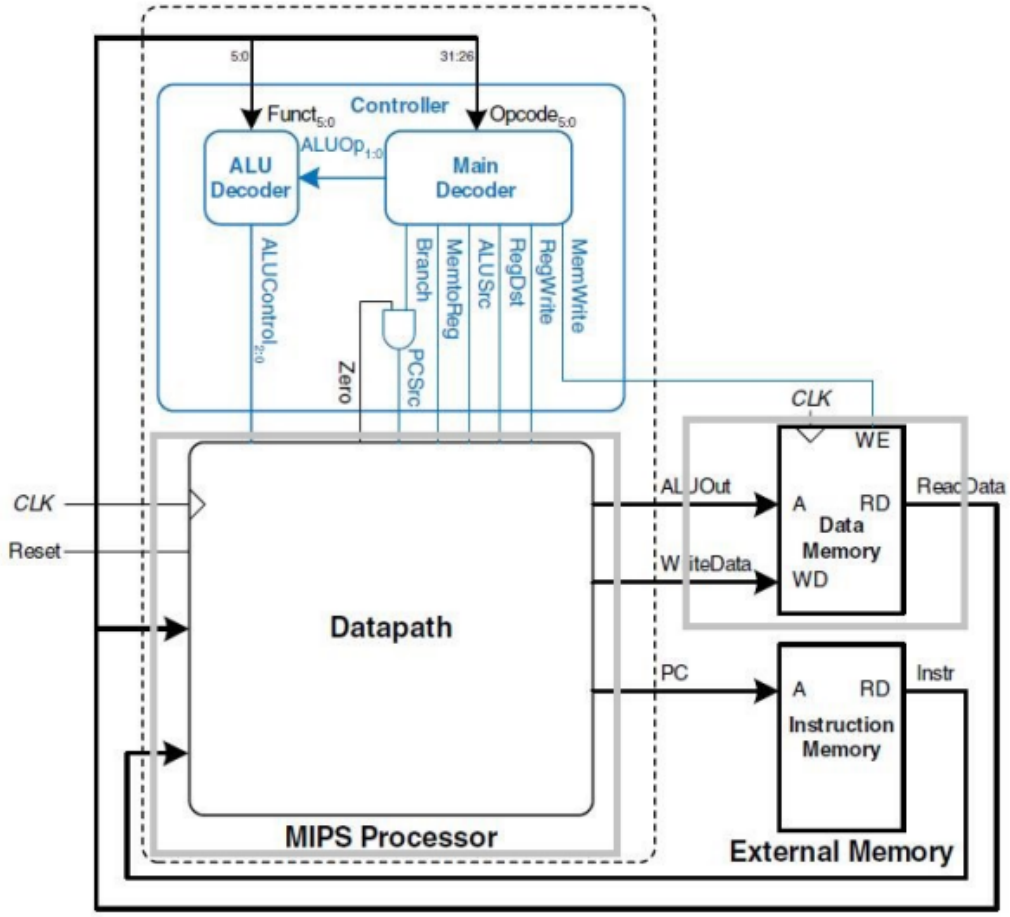

图 4: 顶层模块 RTL 设计图

3.1 实验记录

本实验中，完成了简易单周期 CPU 数据通路中各个小模块的设计，组合 controller、datapath、instr_ram、data_ram 四个模块，并完成不同类型指令（有 R-type,J-Type,I-Type 等指令）的数据通路执行路径的搭建，并编写测试程序，进行行为仿真验证。

3.2 问题记录——论 ALU 输出恒为 0 时如何 Debug？

问题描述：

在仿真 debug 过程，发现写入 data_ram 中的数据对应的地址 data_ram_waddr 恒为 0，于是由此开始了我的福尔摩斯之旅。

调试过程：

由图1可以知道，data_ram_waddr 来源于 datapath 中 alu 模块的 alu_res，当查看 alu_res 的仿真波形图时，发现 alu_res 全程为 0，真的是波“澜”不“惊”。

由 alu_res 又往前一层查看，即整个 alu 模块，与之相关的数据有 rd1，srcB，alucontrol 三个数

据。分别查看每个数据的仿真图，rd1 恒为 0，srcB 正常，alucontrol**“感觉”**正常。

于是可以把目标锁定在 regfile 和 alu 两个模块。经过一系列检查（此处省略描述自己 debug 的辛酸过程的一万字），终于发现是 alu 的问题了（虽然早就怀疑他有问题，但是我之前没有证据）。在 alucontrol 信号从 controller 传播到 datapath 中的 alu 过程中，我的算术逻辑单元 alu 的 alucontrol（执行指令信号）是源自实验 1 的定义，但是我在信号控制器 controller 中 alucontrol（控制这个执行指令信号）是源自实验 2 的定义。二者在 add、subtract、and、or、slt 等运算信号控制上存在一定差别，纠正并统一信号解释后，**Simulation succeeded**，于是我的福尔摩斯之旅结束。

问题解决：

将 alu 中的 alucontrol 和 controller 中的 alucontrol 设计一一对应即可，结果如表3所示。

表 3: alucontrol 信号表

算术操作	**alucontrol[2:0]**
and	000
or	001
add	010
not	100
sub	110
slt	111

4 实验结果及分析

仿真结果如图5所示，控制台结果输出如图6所示（**Simulation succeeded**）。通过对仿真图中最后一行即 alu_result[31:0] 的数据，比照.coe 文件中的 18 行指令结果进行分析，结果正确无误。

A Datapath 代码

```
‘timescale 1ns / 1ps
// done_TODO 实验重点 13点20分
module datapath (
    input wire clk,rst,
    input wire regwrite,regdst,alusrc,branch,memWrite,memtoReg,jump,
    input wire [2:0]alucontrol,
    input wire [31:0]instr,data_ram_rdata,
    output wire [31:0]pc_now,data_ram_waddr,data_ram_wdata
);

// pc
```

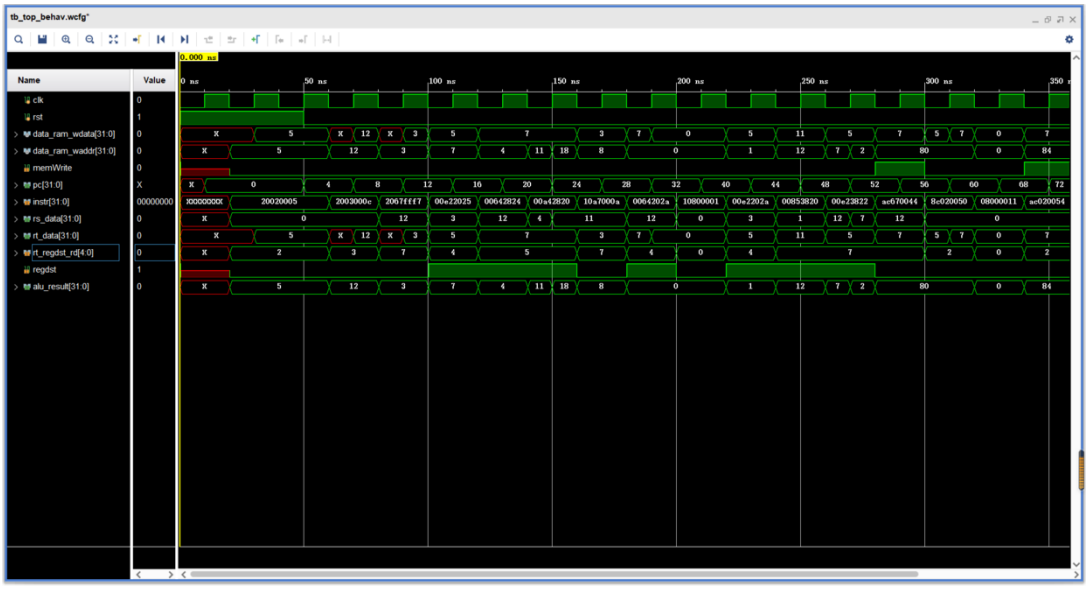

图 5: 仿真结果图

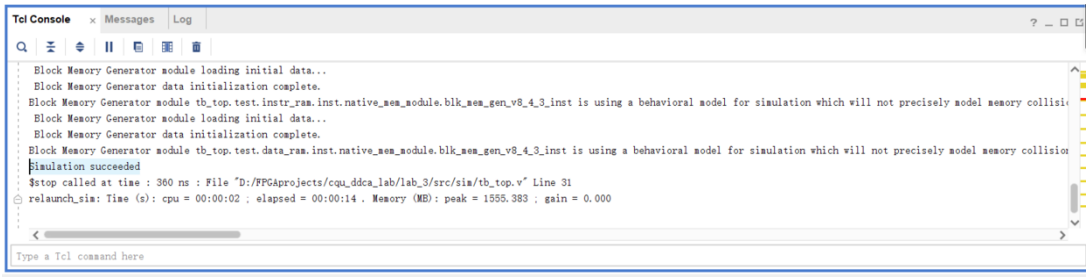

图 6: 控制台输出结果

```
wire [31:0]pc_plus4,pc_branch,pc_next,instr_sl2,pc_next_jump;

// sign extend
wire [31:0]sign_imm,sign_imm_sl2;

// regfile
wire[5:0] wa3;
wire[31:0]rd1,rd2,wd3;

// alu
wire [31:0]srcB,alu_res;

// control
wire pcsrc,zero;

assign pcsrc = zero & branch;
assign data_ram_wdata = rd2;
mux2 mux2_branch(
    .a(pc_plus4),
    .b(pc_branch),
    .sel(pcsrc),
```

```
    .y(pc_next)
);

mux2 mux2_jump(
    .a(pc_next),
    .b({pc_plus4[31:28],instr_sl2[27:0]}),
    .sel(jump),
    .y(pc_next_jump)
);

pc pc(
    .clk(clk),
    .rst(rst),
    .din(pc_next_jump),
    .dout(pc_now)
);

adder adder(
    .a(pc_now),
    .b(32'd4),
    .y(pc_plus4)
);

mux2 #(5) mux2_regDst(
    .a(instr[20:16]),
    .b(instr[15:11]),
    .sel(regdst),
    .y(wa3)
);

regfile regfile(
  .clk(clk),
  .we3(regwrite),
  .ra1(instr[25:21]),
    .ra2(instr[20:16]),
    .wa3(wa3),
  .wd3(wd3),
  .rd1(rd1),
    .rd2(rd2)
);

signext sign_extend(
    .a(instr[15:0]), // input wire [15:0]a
    .y(sign_imm) // output wire [31:0]y
);

sl2 sl2_signImm(
    .a(sign_imm),
    .y(sign_imm_sl2)
```

```
);

sl2 sl2_instr(
    .a(instr),
    .y(instr_sl2)
);

adder adder_branch(
    .a(sign_imm_sl2),
    .b(pc_plus4),
    .y(pc_branch)
);

mux2 mux2_aluSrc(
    .a(rd2),
    .b(sign_imm),
    .sel(alusrc),
    .y(srcB)
);

alu alu(
    .a(rd1),
    .b(srcB),
    .f(alucontrol),
    .y(alu_res),
    .overflow(),
    .zero(zero)
);

mux2 mux2_memtoReg(
    .a(alu_res),
    .b(data_ram_rdata),
    .sel(memtoReg),
    .y(wd3)
);

assign data_ram_waddr = alu_res;

endmodule
```