《计算机组成原理》实验报告

年级、专业、班级	2021 级计算机科学与技术(卓越)01 班	**姓名**	韩昊辰
实验题目	实验三简易单周期 CPU 实验		
实验时间	2023 年 5 月 18 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: 2023 年 5 月 19 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 接受 Controller 解码指令得到的各个信号，接受 Instruction Memory 中读出的指令和 Data Memory 读出的数据，完成各种指令的执行并输出更新后的 PC，ALU 计算结果和 Data Memory 写入数据的地址。

2.2 逻辑实现

在 clk 上升沿进行上一周期更新后 pc 的写回 pc 寄存器，完成 pc+4，在 clk 下降沿根据指令内容从寄存器堆中读出数据，并且得到在指令存储器和数据存储器中得到的指令和数据（用于下一周期），反接 clk 原因在于该 cpu 必须在单周期内执行完成，并且不反接会造成指令冲突。

2.2.1 接口定义

3 实验过程记录

3.1 问题 1：PC 自增模块移除 PC 模块

问题描述：

在实验二中，PC 模块将自增模块放在了 PC 模块内部，clk 每一个上升沿，PC 自增 4。但是在单周期 CPU 中，下一周期 PC 还由 Branch 和 Jump 指令决定。

表 1: DataPath 接口定义

信号名	方向	位宽	功能描述
clka	Input	1-bit	时钟信号
rst	Input	1-bit	重置信号
alucontrol	Input	3-bit	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 读取的数据
alusrc	Input	1-bit	判断 SrcB 是 RD2 还是 SignImm
regdst	Input	1-bit	判断写寄存器号
regwrite	Input	1-bit	写寄存器的使能信号
branch	Input	1-bit	是否是 Branch 指令
jump	Input	1-bit	判断是否执行 jump
mem_read_data	Input	32-bit	lw 从 Data memory 中读取的数据
instr	Input	32-bit	指令
ALUResult	Output	32-bit	ALU 运算结果
wd3	Output	32-bit	写进 Data memory 的数据
pc	Output	32-bit	更新后 PC 值
overflow	Output	1-bit	ALU 计算结果是否溢出

解决方法：

需要将增 4 模块放在 PC 模块外部，PC 模块实质上只是一个 D 触发器，因此直接将 D 触发器模块实例化即可得到 pc 寄存器。下一周期 PC 通过多路选择器进行选择。

3.2 问题 2：存储器与 datapath 分离

问题描述：

本来将 Data Memory 和 Instr Memory 的 IP 核实例化放置在了 Datapath 中，但根据实验指导书的单周期 CPU 框架，两个存储器应该独立于 controller 和 datapath，这实际体现了 cpu 和内存分开的思想，有时存储器容量会非常巨大，不适宜直接嵌入 cpu。

解决方法：

在顶层模块包含两个存储器和 mips 模块，mips 模块包含 datapath 和 controller 模块，模块间用导线连接

3.3 问题 3：ALU 增加 Zero 信号

问题描述：

实验一 ALU 没有 Zero 输出，单周期 cpu 的 branch 指令执行需要用 zero 判断两个操作数是否相

等,以此判断 pc_next 是否为 pc+4。

解决方法:
ALU 增加 Zero 输出信号,输出判断 ALU 结果是否等于 0

3.4 Datapath 不输出 zero 信号

本实验中,对 branch 和 zero 与的处理放在了 datapath 中。Pcsrc 用于判断 pc_next 是否来源于 branch 指令,datapath 接受 controller 的 branch 信号,和内部信号 Zero 相与得到 Pcsrc,datapath 不输出 zero 信号,这样会更符合 datapath 的功能。

3.5 问题 4:无法仿真

问题描述:
synthesis 过了,sim 始终报错。

解决方法:
发现 pc+4 模块实例化名字和数据名字冲突。将实例化名字修改一下就可以跑仿真了。

3.6 问题 5:regfile 写入地址赋值和位宽定义问题

问题描述:
指令第三个 addi 无法正常执行,ALUResult 是高阻态。排查原因发现 ALU 第二个操作数是高阻态,意味着无法从 regfile 中读出数据,进一步排查前两个指令的 regfile 输入输出,发现写入的数据正确,但是地址始终为高阻态,原因在于没有将 regdst 信号作用于 wa3(regfile 写入地址选择,修改后进一步仿真发现 wa3 始终喂不定态(第 6 位为 Z,后 5 位正常)。最后排查 wa3),发现 wa3 本应是 5 位我定义成 6 位。

解决方法:
将 wa3 定义成 5 位,调整各个模块排线,最后解决。

4 实验结果及分析

仿真结果如下:

其中,部分仿真信号作用如下表所示。

和 coe 文件中指令信息和顺序比对,验证正确,仿真报告 Simulation Succeed,截图如下:

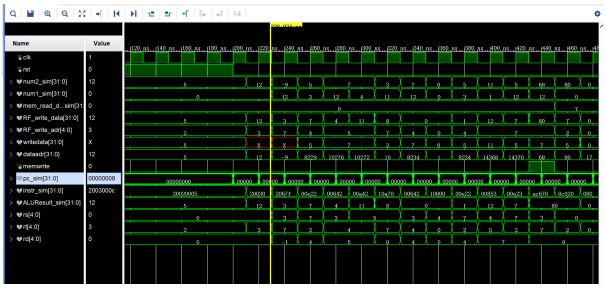

图 1: 仿真结果(前半段)

图 2: 仿真结果(后半段)

表 2: 部分仿真信号定义

信号名	功能描述
num1_sim	ALU 第一操作数
num2_sim	ALU 第二操作数
mem_read_data_sim	数据存储器读出的数据
RF_write_data	寄存器堆写入数据
RF_write_adr	寄存器堆写入地址
writedata	寄存器堆读出的第二个数据
dataadr	指令后 15 位
memwrite	数据存储器写标志
pc_sim	当前 pc
$instr_sim$	当前指令
$ALUResult_sim$	ALU 结果
rs	rs 地址
rt	rt 地址
rd	rd 地址

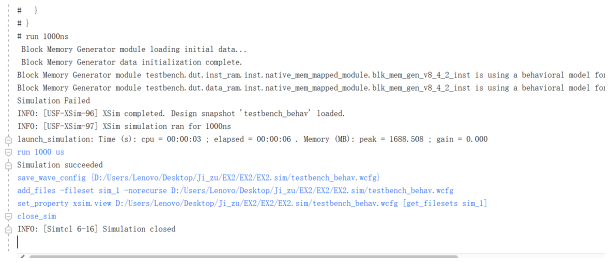

图 3: 仿真报告

A Datapath 代码

```
module datapath(
    input wire clka, rst,
    input wire memtoreg,alusrc,regdst,regwrite,jump,branch,
    input [31:0] instr, mem_read_data,
    input wire [2:0] alucontrol,
    output wire overflow,
    output wire [31:0] pc, ALUResult, writedata,

    // 用于仿真
    output wire [31:0] num2_sim, num1_sim,
    output wire [31:0] RF_write_data,
    output wire [4:0] RF_write_adr
    );

wire [31:0] rd1, rd2;    // read_data
wire [31:0] SignImm;
wire [31:0] PCBranch;
wire [31:0] SignImm_sl2, Jump_adr;
wire [31:0] pc_add_4, alu_srcb, pc_next, pc_next_tmp, wd3;
wire [4:0] wa3;
wire pcsrc;

assign writedata = rd2;
assign RF_write_data = wd3, RF_write_adr = wa3;
// reg_file
regfile regfile(
  .clk(~clka),
  .we3(regwrite),
  .ra1(instr[25:21]), // base
    .ra2(instr[20:16]), // sw, load from rt
    .wa3(wa3), // lw, store to rt
  .wd3(wd3),
  .rd1(rd1),
    .rd2(rd2)
    );

// pc
```

```
D_flip_flop PC(
    .clk(clka),
    .rst(rst),
    .din(pc_next),
    .q(pc)
    );

// pc + 4
Add_4 pc_add_4_adder(
    .a(pc),
    .res(pc_add_4)
    );

// mux2 for reg_file_wd3
Mux2 mux_regfile_wd3(
    .a(mem_read_data),
    .b(ALUResult),
    .s(memtoreg),
    .y(wd3)
    );

// mux for reg_file_a3
Mux2 #(5) mux_regfile_a3(
    .a(instr[15:11]),
    .b(instr[20:16]),
    .s(regdst),
    .y(wa3)
    );

// sign_extend
sign_extend sign_extend(
    .a(instr[15:0]),
    .y(SignImm)
    );

// mux2 for alu_srcb
Mux2 mux_alu_srcB(
    .a(SignImm),
    .b(rd2),
    .s(alusrc),
    .y(alu_srcb)
    );

// ALU
ALU ALU(
    .num1(rd1),
    .num2(alu_srcb),
    .op(alucontrol),     // 操作码
    .res(ALUResult),    // 计算结果
```

```
    .overflow(),
    .Zero(zero),

    .num2_sim(num2_sim),
    .num1_sim(num1_sim)
    );

// 得到PCBranch
assign SignImm_sl2 = {SignImm[29:0],2'b00}; // shift left 2
assign PCBranch = SignImm_sl2 + pc_add_4;

// mux for pc_next_tmp (branch?)
assign pcsrc = zero & branch;
Mux2 mux_pc_next_tmp(
    .a(PCBranch),
    .b(pc_add_4),
    .s(pcsrc),
    .y(pc_next_tmp)
    );

// mux for pc_next (jump?)
assign Jump_adr[27:0] = {instr[25:0],2'b00}; // shift left 2
assign Jump_adr[31:28] = pc_add_4[31:28];
Mux2 mux_pc_next(
    .a(Jump_adr),
    .b(pc_next_tmp),
    .s(jump),
    .y(pc_next)
);


endmodule
```