《计算机组成原理》实验报告

年级、专业、班级	2020 级计算机科学与技术(卓越)01 班	**姓名**	徐聪
实验题目	实验三简易单周期 CPU 实验		
实验时间	2022 年 05 月 12 日	**实验地点**	DS1404
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: 2022 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

实现传统 MIPS 架构单周期 CPU 的取指、译码、执行、访存、回写五阶段的功能。

2.1.2 接口定义

3 实验过程记录

3.1 实验完成工作

1. 分析 mips 指令执行原理，分析数据在数据通路中的流动，分析控制信号对各个模块的控制。
2. 实现控制器译码功能，其中包含 alu 控制器以及 main 控制器的代码编写工作。
3. 实现数据通路中的各个模块连接，控制信号的连接，以及数据之间的传送连接。
4. 对 R-type、lw、sw、jump、beq 等指令的执行结果验证以及信号分析。
5. 各种 bug 的调试、仿真时序图的分析、计算结果的验证。

模块层次：单周期 cpu 的顶层模块为 mips，其中包含两个主要的模块 datapath、controller 以及两个单端口 RAM 的 ip 即 inst_mem 和 data_mem。inst_mem 和 data_mem 分别负责指令和数据的存储。datapath 其中包含 pc、寄存器堆、alu 等主要的模块。寄存器堆负责寄存器数据的存储以及读取，alu 负责数据的计算。controller 模块中包含 main_decoder 以及 alu_control 模块，分别负责主要控制信号和 alu 控制信号的译码工作。

表 1: Datapath 模块接口定义

信号名	方向	位宽	功能描述
inst	input	32-bit	读取来至指令寄存器中的指令
clk	input	1-bit	时钟信号
rst	input	1-bit	刷新信号
instr	input	32-bit	32 位指令
memtoreg	input	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
memwrite	input	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
pcsrc	input	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
alusrc	input	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
regdst	input	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
regwrite	input	1-bit	是否需要写寄存器堆
branch	input	1-bit	是否需要写寄存器堆
jump	input	1-bit	是否为 jump 指令
alucontrol	input	3-bit	ALU 控制信号，代表不同的运算类型
zero	output	1-bit	alu 运算结果是否等于 0
pc	output	32-bit	pc 值用于后面仿真观察 pc 变化
aluout	output	32-bit	alu 运算结果
writedata	output	32-bit	写入内存的数据

3.2 实验中出现的问题

3.2.1 没有时钟降频导致指令延后

问题描述：通过观察仿真时序图，发现 pc 和 inst 之间延迟了两个周期，即 pc 对应的 inst 指令出现在了 pc 指令后第二个周期。

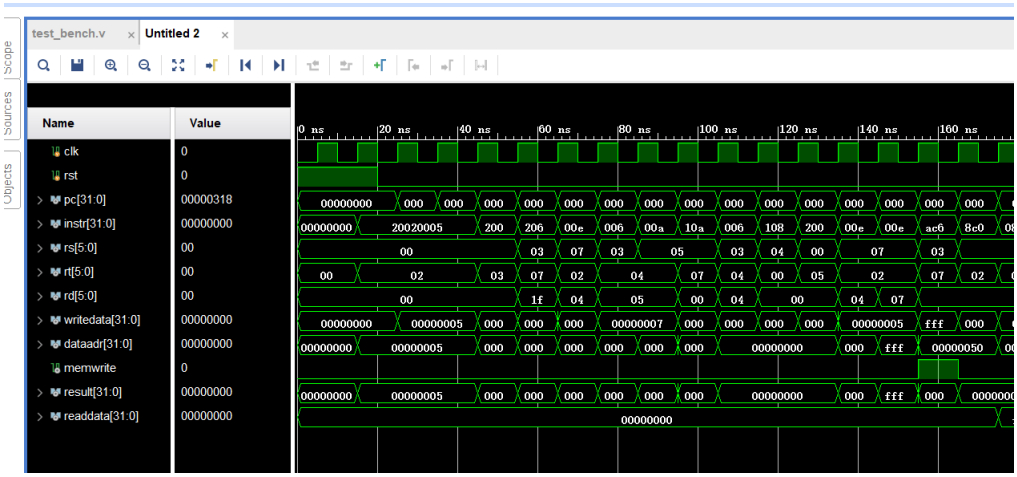


图 1: pc 和对应的 inst 指令延迟两个周期

问题分析：通过分析，问题应该出现在了 pc 和 inst 之间的信号连接模块上。通过检查顶层 top 模块中 pc 和 inst 的信号连接，发现 datapath 和 inst_mem 模块之间的 clk 信号是相同的。这样导致了在获取 pc 的上升沿时刻并不能同时将 inst_mem 中 pc 对应的 mips 指令读出。导致 inst 落后于 pc 两个周期。

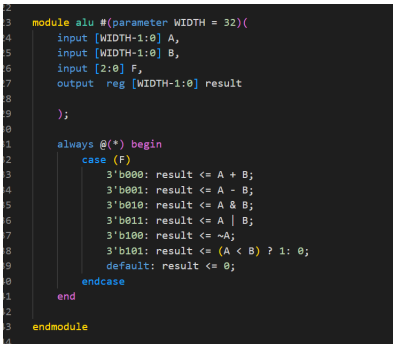
解决方案：在 top 文件中，设置一条 lclk 信号连接降频 10 倍的 clk，datapath 连接降频后的 lclk 信号，inst_mem 以及 data_mem 连接高频 clk 信号。这样可以降低 pc 与 inst_mem 以及 data_mem 之间的信号延迟到一个周期以内。但是也发现 inst 与 pc 之间会有一定的信号延迟，根据降频信号的倍数而改变。

3.2.2 使用实验一 alu 模块导致计算错误

问题描述： alu 计算功能和指令功能不对应，导致 alu 计算结果错误，导致传入 data_mem 模块的数据地址不对，最后导致仿真结果错误。

问题分析：观察实验一中的 alu 模块，发现该模块和实验三 alu 的 aluop 和计算功能不对应。如下图所示，导致 alu 计算结果错误。

解决方案：将实验一中的 alu 模块进行改写，使得其符合单周期 cpu 的 mips 指令计算功能，如下图所示。



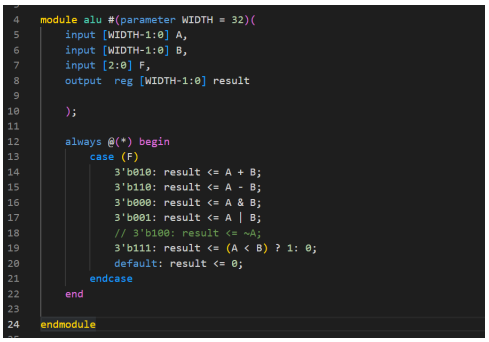
```
module alu #(parameter WIDTH = 32)(
    input [WIDTH-1:0] A,
    input [WIDTH-1:0] B,
    input [2:0] F,
    output  reg [WIDTH-1:0] result

    );

    always @(*) begin
        case (F)
            3'b000: result <= A + B;
            3'b001: result <= A - B;
            3'b010: result <= A & B;
            3'b011: result <= A | B;
            3'b100: result <= ~A;
            3'b101: result <= (A < B) ? 1: 0;
            default: result <= 0;
        endcase
    end

endmodule
```

图 2: 实验一中的 alu 模块



```
module alu #(parameter WIDTH = 32)(
    input [WIDTH-1:0] A,
    input [WIDTH-1:0] B,
    input [2:0] F,
    output  reg [WIDTH-1:0] result

    );

    always @(*) begin
        case (F)
            3'b010: result <= A + B;
            3'b110: result <= A - B;
            3'b000: result <= A & B;
            3'b001: result <= A | B;
            // 3'b100: result <= ~A;
            3'b111: result <= (A < B) ? 1: 0;
            default: result <= 0;
        endcase
    end

endmodule
```

图 3: 实验三中的 alu 模块

4 实验结果及分析

4.1 实验结果展示

4.2 实验结果分析

通过分析各条指令的执行情况，以及 alu 计算结果，根据实验指导文件中给出来的汇编语言表，以及 mips 指令文件。设计的单周期 cpu 正确高效的完成了 fetch、decode、execute、memory、writeback 各个阶段的功能，最后成功地输出了计算结果！

A Datapath 代码

```
`timescale 1ns / 1ps


module datapath(
    input clk,              // 时钟信号
    input rst,              //
    input [31:0] instr,     //32位指令
    input memtoreg,         //回写的数据来自于 ALU 计算的结果/存储器读取的数据
    input pcsrc,            //回写的数据来自于 ALU 计算的结果/存储器读取的数据
    input alusrc,           //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
```

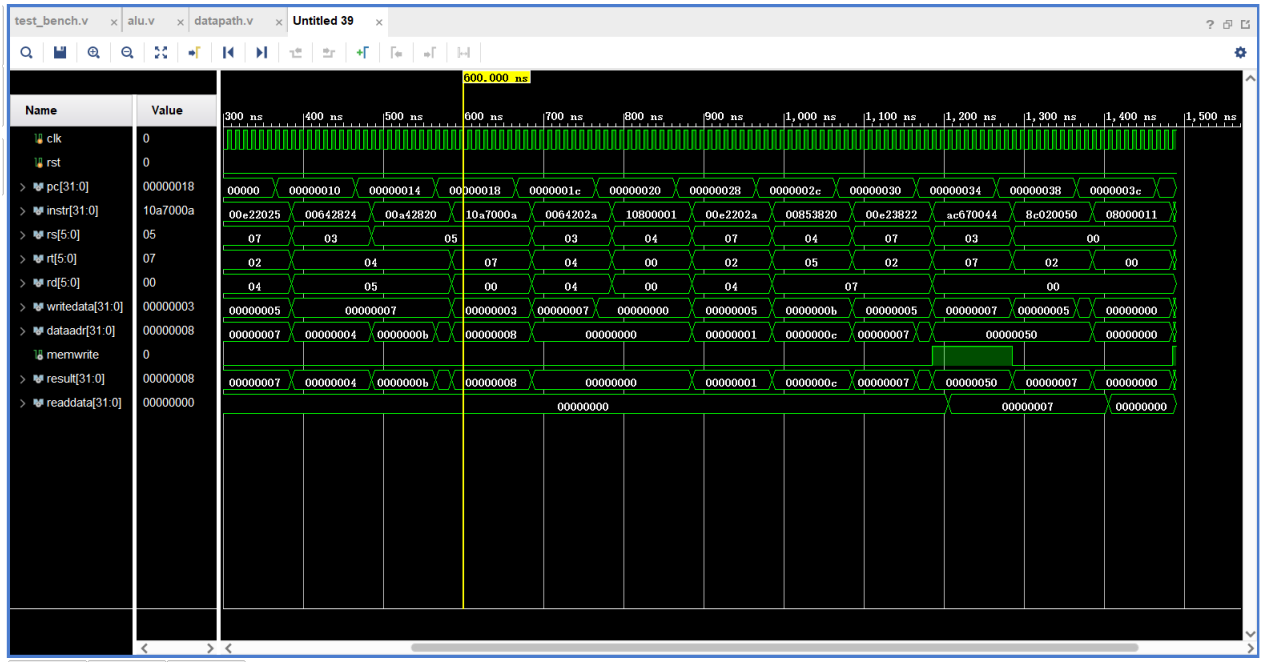


图 4: 仿真时序图

```
# run 2000ns
  Block Memory Generator module loading initial data...
  Block Memory Generator data initialization complete.
Block Memory Generator module test_bench.dut.imem.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral s
Block Memory Generator module test_bench.dut.dmem.inst.native_mem_module.blk_mem_gen_v8_4_2_inst is using a behavioral s
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'test_bench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 2000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:06 . Memory (MB): peak = 2148.461 ; gain = 0.000
```

图 5: 控制台输出

```
input regdst,        //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
input regwrite,      //是否需要写寄存器堆
input jump,          //是否为 jump 指令
input [2:0] alucontrol,      //ALU 控制信号，代表不同的运算类型
input [31:0] readdata,       //内存读取数据
output zero,                 //计算结果是否为零
output [31:0] pc,            //pc
output [31:0] aluout,        //alu计算结果
output [31:0] writedata      //写入内存数据
);

wire [31:0] newpc1, newpc2, pcplus4;
wire [31:0] pcbranch;
wire [31:0] srcA, srcB;
wire [4:0] writereg;
wire [31:0] signimm, signimm_sl2;
wire [31:0] result;

assign pcplus4 = pc + 32'h4;

// 取指令
mux2 #(32) pc_mux1(.a(pcplus4), .b(pcbranch), .f(pcsrc), .c(newpc1));
```

```
    mux2 #(32) pc_mux2(.a(newpc1), .b({pcplus4[31:28], instr[25:0], 2'b00}), .f
        (jump), .c(newpc2));

    pc p(
        .clk(clk),
        .rst(rst),
        .newpc(newpc2),
        .pc(pc)
    );


    // 指令译码
    mux2 #(5) reg_mux(.a(instr[20:16]), .b(instr[15:11]), .f(regdst), .c(
        writereg));

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

    assign signimm = {{16{instr[15]}}, instr[15:0]};
    sl2 sl2(.a(signimm), .y(signimm_sl2));
    adder branch_add(.a(signimm_sl2), .b(pcplus4), .y(pcbranch));


    // 计算执行
    mux2 #(32) src_mux(.a(writedata), .b(signimm), .f(alusrc), .c(srcB));
    alu #(32) alu(.A(srcA), .B(srcB), .F(alucontrol), .result(aluout));
    assign zero = aluout == 32'b0;


    // 内存访问

    // 回写
    mux2 #(32) result_mux(.a(aluout), .b(readdata), .f(memtoreg), .c(result));


endmodule
```