《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 01 班、01 班、06 班	**姓名**	谢双骏、林辰洁、王谦铭
实验题目	实验三简易单周期 CPU 实验		
实验时间	2021 年 5 月 23 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: 肖春华

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 6 月 28 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 就像一个传送带，在控制信号的影响下，实现数据的输入和输出。

2.1.2 接口定义

表 1: Datapath 接口定义表

信号名	方向	位宽	功能描述
clk	input	1-bit	时钟信号,上升沿触发
rst	input	1-bit	重置信号,上升沿触发
jump	input	1-bit	是否为 jump 指令
pcsrc	input	1-bit	下一个 PC 值是 PC+4/跳转的新地址号
alusrc	input	1-bit	送入 ALU B 端口的值是立即数的 32 位扩展/寄存器堆读取的值的
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
regwrite	input	1-bit	是否需要写寄存器堆
regdst	input	1-bit	写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd
alucontrol	input	3-bit	ALU 控制信号，代表不同的运算类型
inst	input	32-bit	输入数据通路的指令信号
$reg_{WriteData}$	input	32-bit	写入寄存器的数据信号
zero	output	1-bit	用于 branch 语句的比较
pc	output	32-bit	输出给 instruction ram,以便取址
aluout	output	32-bit	输出给 data ram,作为写入地址
$mem_{WriteData}$	output	32-bit	输出给 data ram,作为写入数据

3 实验过程记录

3.1 完成工作

首先创建 datapath.v 的 rtl 文件根据实验指导书,一共需要以下部件来构建数据通路:

|——pc.v PC 模块,使用实验二代码

|——alu.v ALU 模块,使用实验一代码

|——sl2.v 移位模块,参考《其他组件实现.pdf》

|——signext.v 有符号扩展模块,参考《其他组件实现.pdf》

|——mux2.v 二选一选择器,自行实现

|——regfile.v 寄存器堆,已提供

|——adder.v 加法器,已提供

3.1.1 LW 指令

LW 指令除了需要寄存器堆,之外,还需要 pc 触发器,pc+4 加法器,符号拓展器,ALU 这几个部件,依次在 datapath.v 引用相应的模块,并用中间变量连接,连接后图示如下:

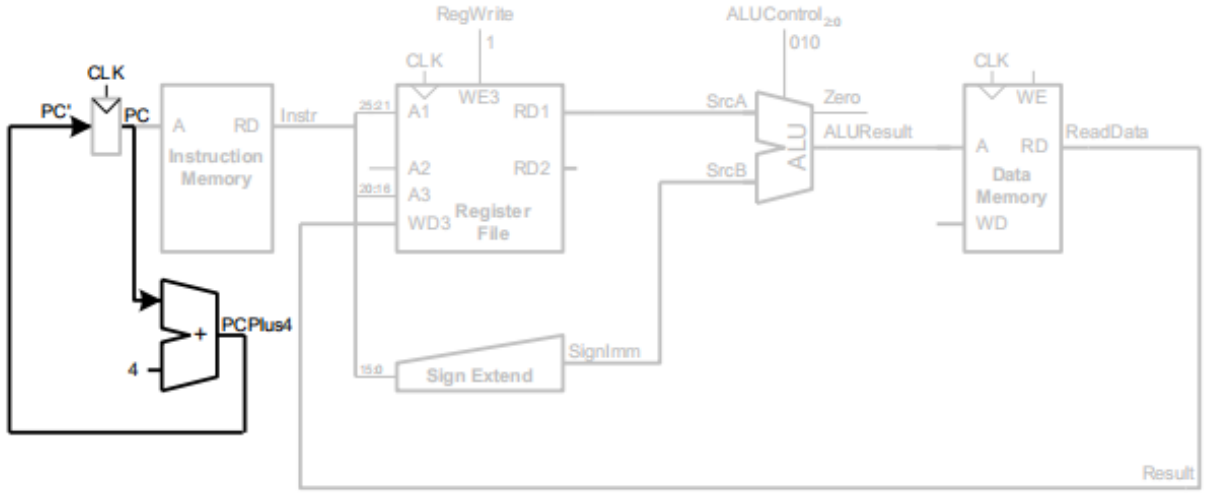

图 1: LW 指令连接图示

3.1.2 SW 指令

在完成 LW 指令的数据通路后,SW 指令仅需进行些许改动即可。只需用中间变量将 RT 连接至 A2,RD2 连接至 WD 即可,不需要增加器件,连接后图示如下:

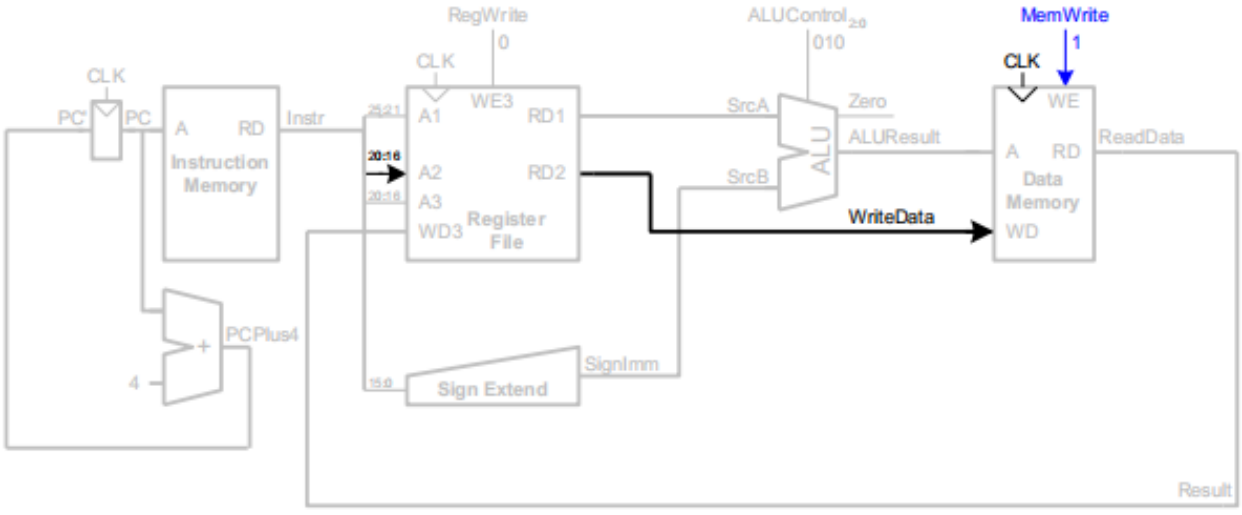

图 2: SW 指令连接图示

3.1.3 R-type 指令

R-Type 指令。除了指令进行的运算类型不同外，其操作均为将 rs,rt 所在寄存器的值进行相应运算后，存入 rd 中。因而，对已经满足 lw、sw 的数据通路进行一定改造即可。只需要加入多路选择器和相应的控制信号即可。lw 指令写入 regfile 的地址为 rt，而 R-Type 为 rd，在此处加入一个多路选择器，输入分别为指令的 [20:16] 和 [15:11]，使用 RegDst(register destination) 信号控制。ALU 输入为 rs,rt，分别对应 srcA,srcB，因此需要在 srcB 处加入多路选择器，选择来源为 RD2 或立即数 SignImm，控制信号为 ALUSrc(ALU source)。最后写回到 regfile 的值应该为 ALU 计算得到的值，为 ALUResult，加入多路选择器控制 9 + ALU Result 来源为 ALU 或数据存储器，控制信号为 MemtoReg(Memory to regfile)。连接后图示如下：

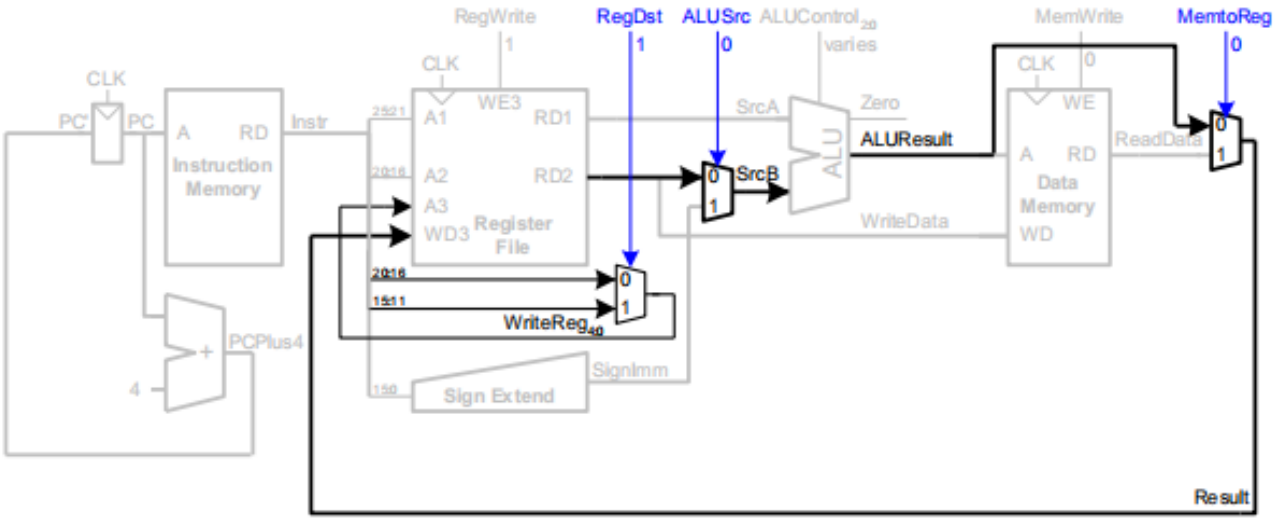

图 3: R-type 指令连接图示

3.1.4 Branch 指令

此处以 beq 指令为例。只需添加一个与门（无需单独写，assign 赋值即可）得到 pcsrc，再用一个两路选择器选择 branch 语句 pc 和 pc+4，控制信号为 pcsrc。连接后图示如下：

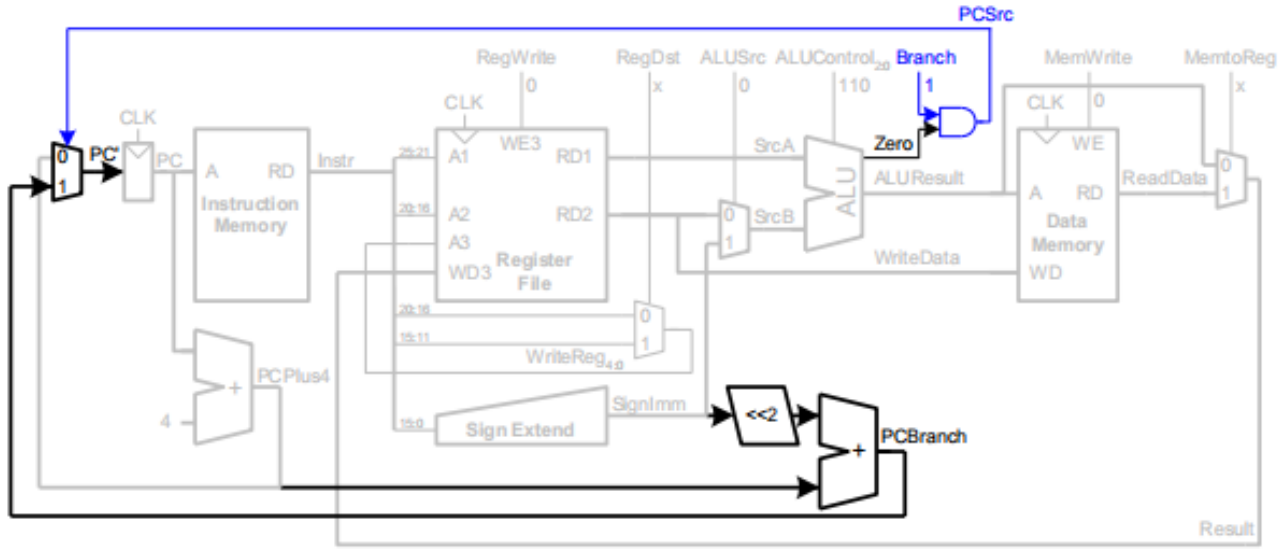

图 4: Branch 指令连接图示

3.1.5 Jump 指令

跳转指令实际为无条件跳转，不需要做任何判断，因此更加简单。只需要计算地址，更改 PC 即可。首先用一个移位器将指令 [25:0] 位左移两位后和 pc 高四位拼接（Verilog 语言实现，无需设计拼接），再用一个两路选择器选择 jump 指令的 pc 和之前已经选择过的 pc，控制信号为 jump。连接后图示如下：在完成上述步骤之后，datapath 基本上就连接完成了，配合 maindec 和 aludec 和

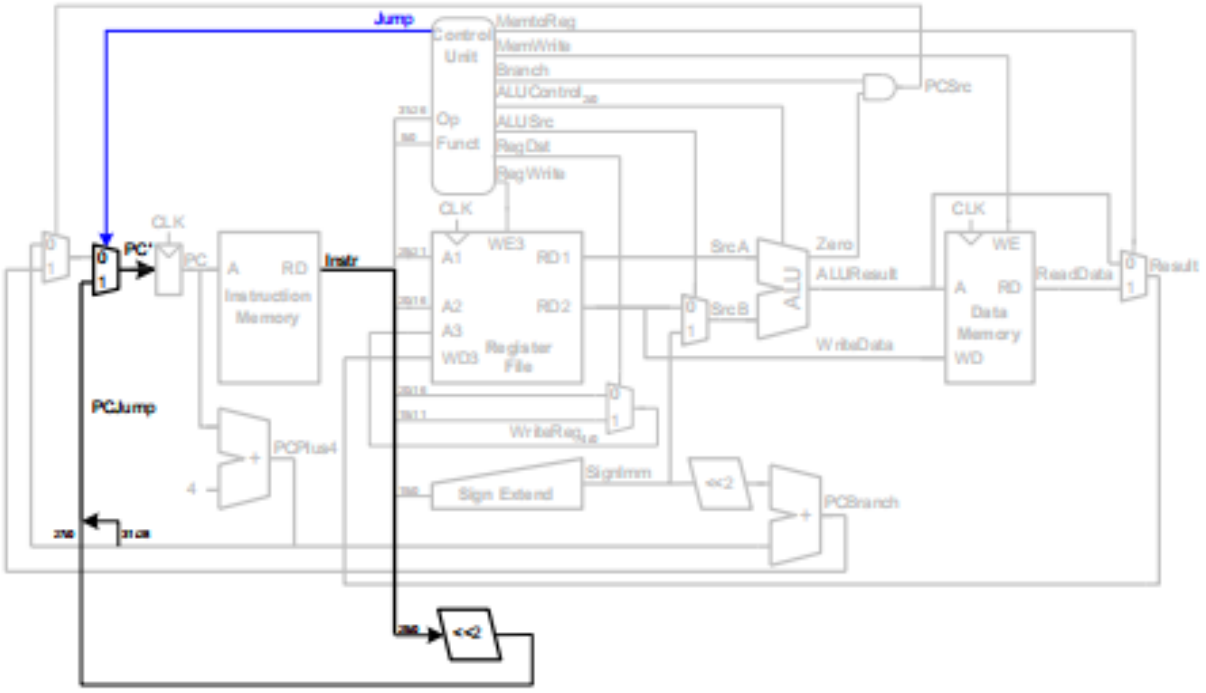

图 5: Jump 指令连接图示

controller 和 mips 文件构成了 MIPS 软核，然后在 top 文件中和 IP 调用创建的 instruction ram 和 data ram 连接，就完成了一个单周期简单 CPU 的设计

3.2 问题

3.2.1 PC 取址问题

在连接完成后，进行 testbench 文件仿真时，发现一旦出现 branch 语句，执行就会出错，是因为当前指令的 pc 和真正的 pc 错开了一个时钟周期。举个简单的例子，假设 pc=20 时，对应取出的指令是 add，pc=24 时取出的指令是 beq，跳转的地址为 80，pc=28 时，对应取出的指令是 sub。假设 pc 原本 =20，在时钟上升沿触发时，pc 更新为 24，指令取出 pc=20 的指令也就是 add，执行完成之后，第二次时钟触发时，pc 更新为 28，指令取出 pc=24 的指令也就是 branch，第三次时钟触发时，pc 更新为 80 也就是 branch 选择的语句，指令取出 pc=28 的指令也就是 sub。我们会发现，执行完 branch 语句之后，并没有跳转执行 pc=80 的语句。而是继续执行 pc+4 的语句，就会出错。

3.2.2 Vivado memory 的调用（1）

vivado 的 block memory 的调用深度达不到 2 的 32 次方，这让读入地址没有 32 位，在和 mips 软核连接时就出现了接口宽度不匹配的问题。

3.2.3 Vivado memory 的调用（2）

且在第一次初始化时没有添加 coe 文件，导致指令永远是 32’b0。

3.2.4 Vivado memory 的调用（3）

发现读取的指令老是慢一个时钟周期。

3.3 解决方案

3.3.1 PC 取址问题

为了解决这个问题，尝试了很多办法，比如添加第二个时钟，最后发现只需要利用时钟的上升沿更新 pc，然后在下降沿取址，就可以保证执行的永远是更新完毕的正确的 pc

3.3.2 Vivado memory 的调用（1）

在调用时有一个选项如图示，只需选择后即可将地址位宽设置为 32 位。

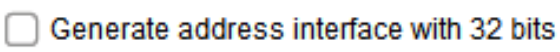

图 6: memory 的调用（1）

3.3.3 Vivado memory 的调用(2)

在图示中,选择所给 coe 文件即可。

图 7: memory 的调用(2)

3.3.4 Vivado memory 的调用(3)

在图示中,不选择所给选项即可。

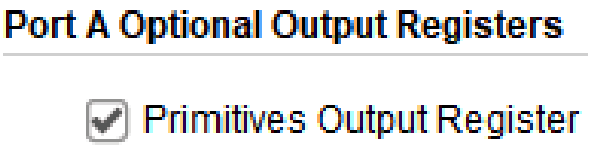

图 8: memory 的调用(3)

4 实验结果及分析

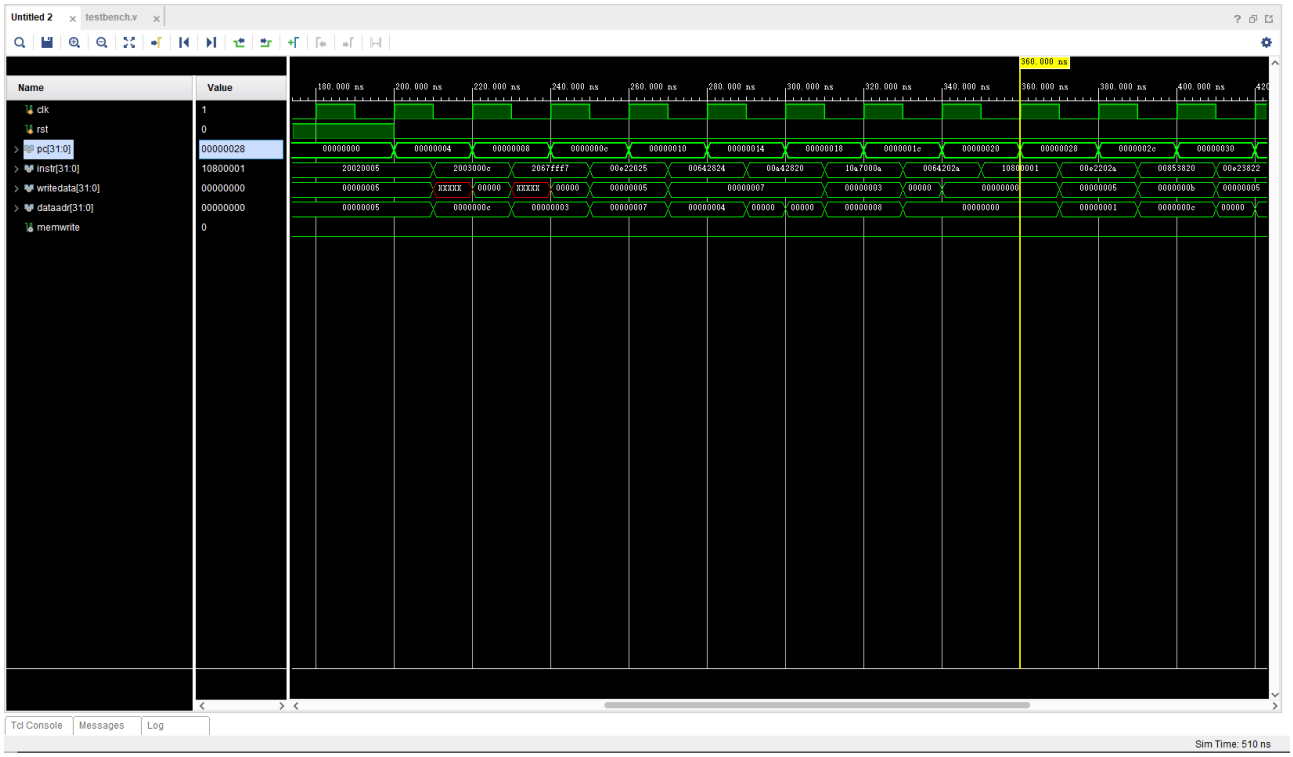

图 9: 仿真结果图

图 10: 仿真文件图

```
Block Memory Generator module testbench.dut.data_ram.inst.\native_mem_mapped_module.blk_mem_gen_v8_4_4_inst  is using a
Simulation succeeded
$stop called at time : 510 ns : File "D:/Xilinx/Xilinx-project/exp3/exp3.srcs/sim_1/new/testbench.v" Line 52
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 1013.824 ; gain = 0.000
```

图 11: 控制台结果图

#	Assembly	Description	Address	Machine	
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	c	00e22025	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	10	00642824	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a	0064202a
	beq $4, $0, around	# should be taken	20	10800001	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050	8c020050
	j end	# should be taken	3c	08000011	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	44	ac020054	ac020054

图 12: coe 指令说明

在仿真结果图中可以看到,在 memwrite=1 时,dataadr=84 并且 writedata=7,仿真结束了,在仿真文件中,断点指向了 stop,同时在控制台中出现了"Simulation succeeded", 说明设计的 CPU 通过了实验。通过的关键在于 coe 文件中的 beq 指令有没有跳过后面的 addi 指令。根绝仿真结果图我们可以看到,pc 在执行完 beq 指令之后,从二十变成了 28,也就是完成了跳转。执行的是 slt 指令。

A Datapath 代码

```
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
);
//声明所有需要的中间变量

//pc+4, branch和pc+4选之后的值, pc的真实值, rd1, rd2, 拓展立即数
wire [31:0] pc_4, pc_branched, pc_realnext, rd1, rd2, extend_imm;


//ALU结果, 写回寄存器堆的数据, 左移后的立即数, jump的地址
wire [31:0] ALUsrcB, wd3, sl2_imm, pc_Branch, sl2_inst, pc_jump;

//寄存去堆写入地址
wire [4:0] reg_WriteNumber;

//内存写入数据
assign mem_WriteData = rd2;

//拼接jump指令
assign pc_jump = {pc[31:28], sl2_inst[27:0]};

//符号拓展
signext sign_extend(
    .a(inst[15:0]),
    .y(extend_imm)
);

//pc的部分, 也就是执行什么指令

//pc
pc_module pc_module(
.clk(clk),
.rst(rst),
.d(pc_realnext),
.pc(pc)
);
//pc+4加法器
```

```
adder pc_4_adder (
.a(pc),
.b(32'h4),
.y(pc_4)
);

//PC指向选择，PC+4和branch语句
mux2 #(32) mux_pcbranch(
.a(pc_Branch),
.b(pc_4),
.s(pcsrc),
.y(pc_branched)
);
//选择分支之后的pc与jump的pc
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)
);

//立即数左移
sl2 sl2_1(
.a(extend_imm),
.y(sl2_imm)
);

//jump指令左移
sl2 sl2_2(
.a({6'b0, inst[25:0]}),
.y(sl2_inst)
);

//branch地址加法器
adder pc_branch_adder (
.a(pc_4),
.b(sl2_imm),
.y(pc_Branch)
);

//pc的部分结束

//寄存器部分

//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)
);

//寄存器堆写入数据来源
mux2 #(32) mux_WD3(
  .a(reg_WriteData),
  .b(aluout),
  .s(memtoreg),
  .y(wd3)
);

//寄存器堆写入地址
mux2 #(5) mux_WA3(
  .a(inst[15:11]),
  .b(inst[20:16]),
  .s(regdst),
  .y(reg_WriteNumber)
);

//ALU部分

//ALU
alu alu(
    .a(rd1),
    .b(ALUsrcB),
    .op(alucontrol),

    .res(aluout),
    .zero(zero)
);
//ALUB端的输入值
mux2 #(32) mux_ALUsrc(
    .a(extend_imm),
    .b(rd2),
    .s(alusrc),
    .y(ALUsrcB)
);
endmodule
```