# 实验 1-2 报告

段江飞
2016K8009929011

## 一、实验任务

### （一）设计

1、仔细阅读讲义，理解实验要求和各个信号的意思，理清要实现的目标，并试图把握有坑的地方。

2、分析指令，思考各个指令执行所需要的数据通路。

3、在计算机组成原理所讲的多周期 CPU 基础上设计一个符合本实验要求多周期 CPU，实现的是各条指令执行经历相同的阶段，即 IF（取之）-ID（译码）-EX（执行）-WB（写回）。

4、对设计的 CPU 进行模块划分，共划分为 3 个模块，寄存器堆（regfile）模块、运算器（alu）模块和译码（ID）模块。

5、从头检查一遍，看看设计是否合理，是否能实现目标。

### （二）实现

1、修改寄存器堆模块和 alu 模块，其中 alu 模块添加了一些运算。

2、按照讲义中的接口修改组成原理课上的多周期 CPU，并遵照 verilog 编程规范。

3、仿真调试修改过的 CPU 发现和 refer 对比，PC 总是慢了一拍，仔细对照 SRAM 的特性和波形图，发现本实验中的 SRAM 和之前的不一样，读数据和读命令都会慢一拍，思考之后发现不适合将原来的多周期进行修改，觉得重新设计一个简单的支持 19 条指令的 CPU，每条指令的执行经历一个相同的阶段，并且便于之后流水线的修改。

4、按照对多周期 CPU 的模块划分编写各个模块，首先完成 regfile 和 alu 模块，然后完成译码模块，最后完成顶层的 mycpu_top 模块。

### （三）验证

1、用 vivado 的语法检查，检查语法错误并修改。

2、根据讲义上的验证步骤进行验证，跳过第一二步，将 mycpu 的源文件加入项目，然后仿真。

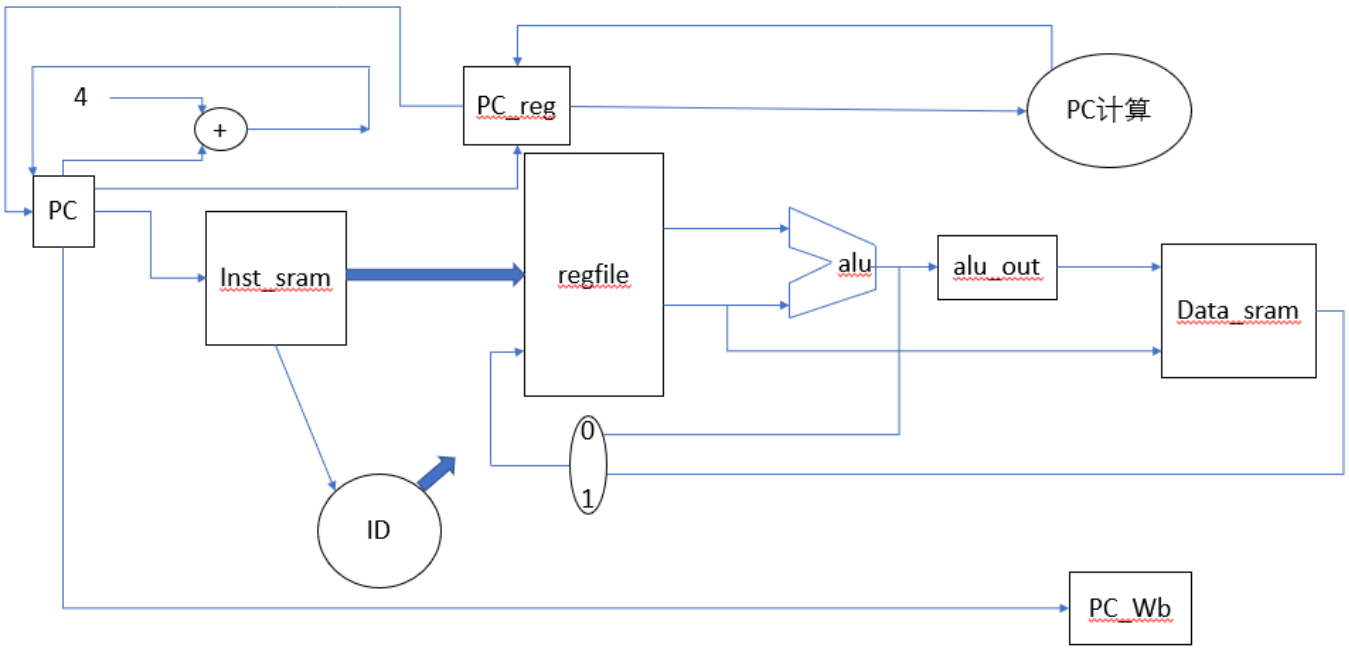
3、根据 tcl 控制台的打印结果定位错误，找出程序的 bug 和逻辑错误。

4、上板验证判断是否结果正确。

## 二、实验设计

### （一）设计方案

## 1、总体设计思路




图一

很简化的 CPU 结构如图一所示，各阶段工作机制为

（1） 取指阶段

从 inst_sram 中取出指令，并且将 PC 存于 PC_Wb 中带到写回级，并且将 PC 加 4 写回 PC 中，如果跳转指令，则需要考虑分支延迟槽，用 PC_reg 计算分支跳转后的 PC，并用一个寄存器存储是否满足条件，若满足，则在执行分支延迟槽中的指令时写回。

（2） 译码阶段

指令在译码模块中进行译码，并且取出寄存器堆中的值。译码模块输入是当前状态和指令，输出是各个控制信号。寄存器堆模块的输入时读地址、写地址、写数据和使能信号，输出时读数据。

（3） 执行阶段

通过 alu 计算并将结果存在 alu_out 中，如果需要写回寄存器，则进行写回。同时，根据指令计算下一个 PC 值。ALU 模块输入时两个值和控制信号，输出计算结果及溢出等异常判断。

（4） 写回阶段

将下一个 PC 写回 PC，如果需要读或者写入 data_sram，则 alu_out 中存的是地址，然后进行读出或写入。

## 2、寄存器堆设计

### （1）工作原理

寄存器堆是 CPU 中临时存储数据的器件，有 32 个寄存器组成，每个 32 位，所以用 32×32 个触发器构成。寄存器堆写入数据，要写使能有效，数据会在后一拍写入；读数据时，在同一拍就能返回数据。寄存器堆是有触发器组成的，写入是要有时钟信号触发，读出不需要。0 号寄存器值始为 0。

（2）接口定义

| 名称 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| clk | IN | 1 | 时钟信号，触发功能 |
| raddr1 | IN | 5 | 读寄存器的编号 1 |
| raddr2 | IN | 5 | 读寄存器的编号 2 |
| rdata1 | OUT | 32 | 读出寄存器的值 1 |
| rdata2 | OUT | 32 | 读出寄存器的值 2 |
| wen | IN | 1 | 写使能信号 |
| waddr | IN | 5 | 写入数据的寄存器编号 |
| wdata | IN | 32 | 写入寄存器的值 |

（3）功能描述

寄存器堆由 32×32 个触发器构成，共 32 个寄存器，每个寄存器位宽为 32 位，写入数据时，当写使能信号有效，写地址对应的寄存器在下一个时钟周期刷新，写数据写入相应的寄存器；读出数据时，将读地址和 0 做按位或运算，若结果非 0，则将该地址对应寄存器的值读出，若结果为 0，则读出 0。

## 3、ALU 模块设计

（1）工作原理

CPU 工作时，要进行地址计算、数据计算等，ALU 是是 CPU 内的运算部件，进行加减、比较、与或非、移位等运算，同时 ALU 是组合逻辑实现的二元运算器，会把所有可进行的运算的结果计算出来，然后根据控制信号输出结果，同时进行溢出和零状态标记。

（2）接口定义

| 名称 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| alu_control | IN | 12 | 控制信号，标记 ALU 要进行的运算 |
| alu_src1 | IN | 32 | 输入运算数据 1 |
| alu_src2 | IN | 32 | 输入运算数据 2 |
| alu_result | OUT | 32 | 运算结果 |
| Overflow | OUT | 1 | 有符号数溢出标记 |
| CarryOut | OUT | 1 | 无符号数溢出标记 |
| Zero | OUT | 1 | 结果为 0 标记 |

（3）功能描述

ALU 控制信号采用独热码，便于解码。ALU 的比较是利用减法判断，而 ALU 的减法利用加上减一个数等于加上其补码，和加法运算利用同一数据通路，先根据是否做减法对运算数求补码，然后相加，并记录进位和借位，根据进位和借位判断是否溢出和比较的结果，与或非和移位就是直接利用逻辑运算符运算，最后，根据控制信号输出结果，零标志位根据输出结果或规约取反设置。

## 4、译码模块设计

（1）工作原理

CPU 的很多接口需要控制信号进行控制，这些控制信号有译码产生，将译码单独作为一个模块有助于简化设计，并且便于修改。译码模块是一个组合逻辑的模块，输入是指令、当前状态、ALU 结果和 Zero 标志位，输出是各个控制信号，译码模块先对输入指令对应到具体指令集的操作码，然后根据控制信号的需要进行连线。

（2）接口定义

| 名称 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| inst_sram_data | IN | 32 | 当前的指令 |

| 名称 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| cur_state | IN | 3 | 当前所处的状态 |
| alu_result | IN | 32 | ALU 运算结果 |
| Zero | IN | 1 | ALU 运算结果的零标志位 |
| rf_wen | OUT | 1 | 寄存器堆的读使能控制信号 |
| rf_waddr_sel | OUT | 2 | 寄存器堆的写地址选择信号 |
| rf_wdata_sel | OUT | 1 | 寄存器堆的写数据选择信号 |
| alu_src1_sel | OUT | 1 | ALU 的 alu_src1 的选择信号 |
| alu_src2_sel | OUT | 1 | ALU 的 alu_src2 的选择信号 |
| alu_src_op_jal | OUT | 1 | ALU 的 alu_src2 的 jal 指令时的选择信号 |
| alu_control | OUT | 12 | ALU 的执行的运算的控制信号 |
| data_sram_en | OUT | 1 | 数据 SRAM 的使能信号 |
| data_sram_wen | OUT | 4 | 数据 SRAM 的写使能信号 |
| branch_cond | OUT | 1 | 条件分支的条件为真信号 |
| inst_sram_addr_dirw | OUT | 1 | 控制 PC 直接跳转的信号 |
| inst_sram_addr_condw | OUT | 1 | 控制 PC 条件跳转的信号 |
| inst_sram_addr_src | OUT | 2 | 跳转后的 PC 的来源选择信号 |
| data_sram_addr_src | OUT | 1 | 写入数据 SRAM 的来源选择信号 |

（3）功能描述

译码部分首先将指令和对应的操作码做异或，再对结果做或规约取反，类似于译码器，得到各个指令的 op_*信号，然后根据控制信号为真的条件和当前状态，用 op_*和状态信号进行逻辑运算，多位的控制信号进行一位一位的赋值，alu_result 和 Zero 用于判断条件分支。

## 3、mycpu_top 设计

（1）工作原理

CPU 设计的顶层模块，包括控制状态转移的状态机，调用其他三个模块的数据通路和 PC 跳转的数据通路。由图一的 CPU 结构设计图可以看出 CPU 的核心组成就是这些部分，CPU 依据状态转移执行命令。

（2）接口定义

| 名称 | 方向 | 位宽 | 功能描述 |
|---|---|---|---|
| clk | IN | 1 | 时钟信号，触发功能 |
| resetn | IN | 1 | 复位信号，低电平有效 |
| inst_sram_en | OUT | 1 | 指令 SRAM 使能信号，进行读写指令 SRAM 时有效 |
| inst_sram_wen | OUT | 4 | 指令 SRAM 写使能信号，进行写指令 SRAM 时有效 |
| inst_sram_addr | OUT | 32 | 从指令 SRAM 读出命令的地址 |
| inst_sram_wdata | OUT | 32 | 往指令 SRAM 写入数据的值 |
| inst_sram_rdata | IN | 32 | 从指令 SRAM 读出的命令 |
| data_sram_en | OUT | 1 | 数据 SRAM 使能信号，进行读写数据 SRAM 时有效 |
| data_sram_wen | OUT | 4 | 数据 SRAM 写使能信号，进行写数据 SRAM 时有效 |
| data_sram_addr | OUT | 32 | 从数据 SRAM 读出数据的地址 |
| data_sram_wdata | OUT | 32 | 往数据 SRAM 写入数据的值 |
| data_sram_rdata | IN | 32 | 从数据 SRAM 读出的值 |
| debug_wb_pc | OUT | 32 | debug 信号，连接写回级的 PC |
| debug_wb_rf_wen | OUT | 4 | debug 信号，连接寄存器堆写使能信号 |
| debug_wb_rf_wnum | OUT | 5 | debug 信号，连接写入寄存器的编号 |
| debug_wb_rf_wdata | OUT | 32 | debug 信号，连接写入寄存器的数据 |

（3）功能描述

状态转移部分，采用独热码表示状态，共 4 个状态，复位时当前状态处于取指状态，下一个状态 next_state 利用组合逻辑记录当前的下一个状态，每一拍当前状态更新为下一个状态。

PC 跳转部分，见设计的取指阶段机制。

调用译码模块，对控制信号进行译码，并根据译码后的控制信号，连接寄存器堆的读地址和写地址以及读使能信号，读出数据。

根据译码后的控制信号，连接 ALU 的两个输入，然后调用 ALU 模块进行运算，并将计算结果存在 alu_out 里，如果需要读取内存，同时也会直接将结果连接到读取数据 SRAM 的地址。

根据写回控制信号，连接写回数据 SRAM 的值，data_sram_addr 为 alu_result 或 alu_out，data_sram_wdata 则是从寄存器堆中读取的数据，写使能信号在单元中赋值。

在 CPU 中不对指令 SRAM 进行写入，所以 inst_sram_wen 和 inst_sram_wdata 一直置 0，使能信号置高。

debug 信号连接对应的信号，带到写回级的 PC，寄存器堆写使能、写地址和写数据信号。

## （二）验证方案

### 1、总体验证思路

#### （1）仿真验证

在 vivado 下仿真验证逻辑是否正确，预期结果是在控制台打印 PASS，如果出现 Error 后者一直不终止，则去定位 bug，修改逻辑。

#### （2）FPGA 系统验证

仿真验证完成之后，上板验证，观察结果是否和讲义上描述一致，若不一致，则看时序等是否符合。

### 2、验证环境

#### （1）仿真验证

软件部分已经准备好，可以跳过。

Vivado 仿真运行 mycpu，inst_ram 加载 func，加载指令，激励文件通过顶层调用 mycpu 模块，执行 inst_ram 中的指令，同时，mycpu 中的 debug 信号抓取 mycpu 的中间运行结果，激励文件中读取龙芯 CPU 运行过程中的信号生成的比对文件，和 mycpu 的 debug 信号进行对比，如果相同则继续运行，不相同，会停止运行，并且输出错误的时间点等信息。

#### （2）FPGA 系统验证

仿真验证通过，且综合和生成 bit 文件成功之后，进行 FPGA 系统验证，通过激励文件，控制 FPGA 板上 led 灯的亮灭，若 FPGA 板上的灯亮灭、颜色变化过程的过程和数码管变化过程和讲义一致，则验证成功。

### 3、验证计划

#### （1）仿真验证

仿真验证看 19 个功能点是否能 PASS 实验。

#### （2）FPGA 系统验证

上板测试，看时序是否满足，能正确显示。

| 编号 | 功能点描述 | 考核标准 |
|---|---|---|
| 1 | lui 指令，高位加载 | 控制台打印 PASS，数码管高低 8 位数值相同，为 1 |
| 2 | addu 指令,无符号数相加 | 控制台打印 PASS，数码管高低 8 位数值相同，为 2 |
| 3 | addiu 指令，无符号数相加 | 控制台打印 PASS，数码管高低 8 位数值相同，为 3 |
| 4 | subu 指令，无符号数减法 | 控制台打印 PASS，数码管高低 8 位数值相同，为 4 |

| 编号 | 功能点描述 | 考核标准 |
|---|---|---|
| 5 | slt 指令，比较 | 控制台打印 PASS，数码管高低 8 位数值相同，为 5 |
| 6 | sltu 指令，无符号数比较 | 控制台打印 PASS，数码管高低 8 位数值相同，为 6 |
| 7 | and 指令，按位与 | 控制台打印 PASS，数码管高低 8 位数值相同，为 7 |
| 8 | or 指令，按位或 | 控制台打印 PASS，数码管高低 8 位数值相同，为 8 |
| 9 | xor 指令，按位异或 | 控制台打印 PASS，数码管高低 8 位数值相同，为 9 |
| 10 | nor 指令，按位或非 | 控制台打印 PASS，数码管高低 8 位数值相同，为 10 |
| 11 | sll 指令，逻辑左移 | 控制台打印 PASS，数码管高低 8 位数值相同，为 11 |
| 12 | srl 指令，逻辑右移 | 控制台打印 PASS，数码管高低 8 位数值相同，为 12 |
| 13 | sra 指令，算数右移 | 控制台打印 PASS，数码管高低 8 位数值相同，为 13 |
| 14 | lw 指令，加载字 | 控制台打印 PASS，数码管高低 8 位数值相同，为 14 |
| 15 | sw 指令，存储字 | 控制台打印 PASS，数码管高低 8 位数值相同，为 15 |
| 16 | beq 指令，相等则跳转 | 控制台打印 PASS，数码管高低 8 位数值相同，为 16 |
| 17 | bne 指令，不相等则跳转 | 控制台打印 PASS，数码管高低 8 位数值相同，为 17 |
| 18 | jal 指令，跳转并链接 | 控制台打印 PASS，数码管高低 8 位数值相同，为 18 |
| 19 | jr 指令，跳转到寄存器中地址 | 控制台打印 PASS，数码管高低 8 位数值相同，为 19 |

# 三、实验实现

## （一）实现交付说明

|--myCPU/ ： myCPU 目录

| |--myCPU_define.h ：my_CPU 头文件

| |--myCPU_ALU.v ：my_CPU 的 ALU 模块代码

| |--myCPU_regfile.v ：my_CPU 的寄存器堆模块代码

| |--myCPU_ID.v ：my_CPU 的译码模块代码

| |--myCPU_top.v ：my_CPU 的顶层模块代码

## （二）实现说明

具体模块对应关系见上表，myCPU_top.v 顶层模块调用其他三个模块。

# 四、实验测试

## （一）测试过程

在先修改组成原理的多周期 CPU 后，首先测试第一个功能点，发现 PC 总是慢一拍，仔细考虑之后，觉得重新设计 CPU，不在原有的 CPU 上修改。

新设计的 CPU 对功能点依次进行测试，中间遇到了比较多漏写条件，还有跳转错误，通过波形图和打印的错误信息找到 bug 并修改。

生成 bit 文件，进行 FPGA 上板测试。

## （二）测试结果

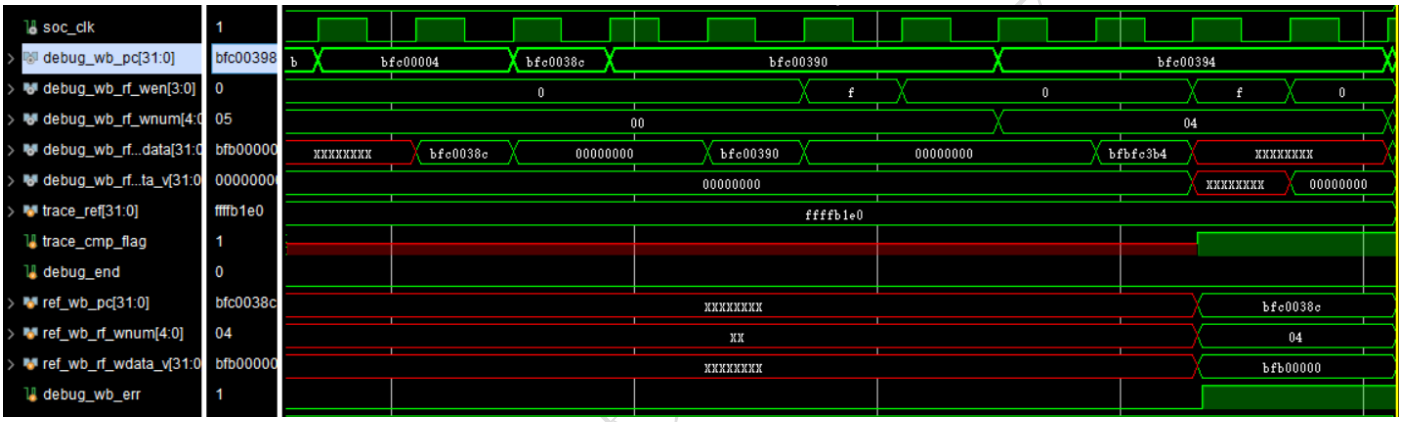
各个测试点均成功通过测试，在控制台打印 PASS，并且，上板测试过程与讲义一致。

# 五、实验 bug 说明

## （一）bug-1： debug 信号慢一拍

### 1、错误现象

看到图二中波形图可以发现 debug_wb_pc 信号的变化复杂，而且对不上比对的 pc 的值，期间我还改过一次，是 debug_wb_pc 和比对的 pc 慢一拍，因为调试过程中没有保存仿真波形截图，代码也没有保存，只好看这一个了。




图二

### 2、分析原因

首先，我查看 testbench，看仿真过程是如何比对的，然后我对应到波形图，发现前面没有触发比对是因为写地址为 0，但根据比对，写地址不应该是 0，我怀疑是我代码取指令取错了，然后我去找反汇编代码，发现是读出的指令慢了一拍，然后，回到讲义和 piazza，发现有提醒这一段，读指令会慢一拍才能读出，和之前设计的 CPU 不太一样，而且 pc 比对的是带到写回级的 pc，我的 pc 并没有带到写回级。

```
always@(posedge clk or negedge resetn)
begin
    if(~resetn)
        inst_sram_addr <= `myCPU_INIT_INST_ADDR;
    else begin
        if(inst_sram_addr_dirtw | (inst_sram_addr_condw & branch_cond))
            case(inst_sram_addr_src)
                2'b00: //jr, jalr
                    inst_sram_addr <= {alu_result[31:1], 1'b0};
                2'b01: //branch
                    inst_sram_addr <= alu_out;
                2'b10: //j,jal
                    inst_sram_addr <= {inst_sram_addr[31:28], inst_reg[25:0], 2'b00};
                default:
                    inst_sram_addr <= inst_sram_addr;
            endcase
    end
end
```

图三

### 3、解决方案

我首先尝试在原有电路上修改。我在电路中间添加一个 pc 的寄存器，保存 pc 的运算的值，仿真运行之后，pc 刚开始能够对上，但是写入的值不对，不太确定修改下去能不能完成一个正确的 CPU。我仔细思考之后，加上 piazza 上关于写回级的讨论，我发现上学期组成原理的课上的多周期 CPU 和实验课上要实现的多周期 CPU 的内在思想有些不同，实验课上的多周期 CPU 是为了之后的流水线 CPU 做准备的，这个把 pc 带到写回级机制和 CPU 的取指机制很适合一个各条指令经历一个相同的执行过程，而组成原理实验课的多周期CPU不同类型的指令执行经过一个不同的过程，不太好改成多周期 CPU，所以我觉得重新设计一个和组成原理的多周期 CPU 执行过程不同的 CPU 更好一些，于是，我重新写了一个 CPU，之前花费了一天多的修改全部白费，主要还是没理解这个过程。

## （二）bug-2：永不终止

### 1、错误现象

程序运行过程中一直不停止，也不打印出错，而且打印的 debug_wb_pc 很奇怪，不在比对范围内。

```
[9072000 ns] Test is running, debug_wb_pc = 0xbfc9d7d4
[9082000 ns] Test is running, debug_wb_pc = 0xbfc9d9c8
[9092000 ns] Test is running, debug_wb_pc = 0xbfc9dbbc
[9102000 ns] Test is running, debug_wb_pc = 0xbfc9ddb0
[9112000 ns] Test is running, debug_wb_pc = 0xbfc9dfa4
[9122000 ns] Test is running, debug_wb_pc = 0xbfc9e198
[9132000 ns] Test is running, debug_wb_pc = 0xbfc9e38c
[9142000 ns] Test is running, debug_wb_pc = 0xbfc9e580
[9152000 ns] Test is running, debug_wb_pc = 0xbfc9e774
[9162000 ns] Test is running, debug_wb_pc = 0xbfc9e968
[9172000 ns] Test is running, debug_wb_pc = 0xbfc9eb5c
[9182000 ns] Test is running, debug_wb_pc = 0xbfc9ed50
[9192000 ns] Test is running, debug_wb_pc = 0xbfc9ef44
[9202000 ns] Test is running, debug_wb_pc = 0xbfc9f138
[9212000 ns] Test is running, debug_wb_pc = 0xbfc9f32c
[9222000 ns] Test is running, debug_wb_pc = 0xbfc9f520
[9232000 ns] Test is running, debug_wb_pc = 0xbfc9f714
[9242000 ns] Test is running, debug_wb_pc = 0xbfc9f908
[9252000 ns] Test is running, debug_wb_pc = 0xbfc9fafc
[9262000 ns] Test is running, debug_wb_pc = 0xbfc9fcf0
[9272000 ns] Test is running, debug_wb_pc = 0xbfc9fee4
run: Time (s): cpu = 00:00:59 ; elapsed = 00:04:45 . Memory (MB): peak = 958.004 ; gain = 10.313
close_sim
INFO: [Simtcl 6-16] Simulation closed
```

图四

### 2、分析原因

通过最后一次在控制台打印的数字附近的 PC 定位错误，发现是跳转的时候，PC 计算出错，跳到了一个莫名其妙的地方，然后就一直运行，脱离了正常轨道。

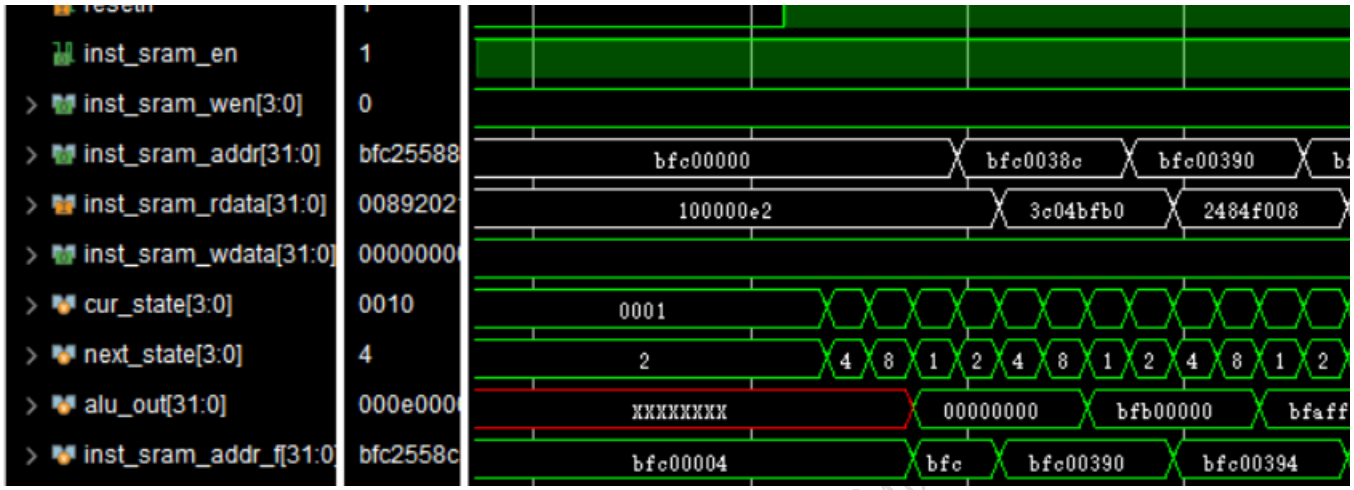
### 3、解决方案

定位到了错误，对照控制信号，进行更改，然后重新仿真运行，看是否解决问题。

## （三）bug-2：永不终止

### 1、错误现象

没有分支延迟槽。




图五

### 2、分析原因

PC 写回的时候，是最后计算好下一条指令的地址，在写回阶段直接写回，没有考虑分支延迟槽的情况。

### 3、解决方案

修改代码，PC 在取指阶段完成后就加 4，然后在中间加上一级 PC 的寄存器，暂存 PC 分支跳转的目的地址，然后根据条件，在执行分支延迟槽指令时写回 PC。

# 六、成员分工

在原有的多周期上修改，因为实现的指令比较多，而且不太熟悉，花费了一天左右的时间，否定了直接修改的方案，然后重新考虑设计，花费大概七八个小时，全部实现，但是实验报告就写了超级久。

修改没有分支延迟槽的设计错误，花费 3 小时，无论怎么想，都感觉实现的很不自然。

# 七、实验总结

通过这次实验，我对硬件开发有了更深的理解，一定要想好具体如何实现，之后再去动手写代码，这样事半功倍，而且代码的可行性也有保证，避免了盲目修改，造成时间的大量浪费。还要看好实验任务，不要漏了分支延迟槽！

实验之前一定要仔细的阅读讲义和老师的提醒，这些问题是实验的大坑，如果不注意，会造成很大的问题。