实验 9 报告

学号：2017K8009922027 2017K8009929011
姓名：张磊、郭豪
箱子号：39

一、实验任务（10%）

任务：在实验 8 的基础上，增加 BREAK 指令，增加 CP0 寄存器 COUNT，COMPARE，BADVADDR，同时增加 break，地址错，整数溢出，保留指令的例外支持，运行功能测试通过，运行 func_lab8，要求通过仿真和上板验证。

附加题：运行记忆游戏的测试并顺利通过。

二、实验设计（40%）

（一）总体设计思路

1. 对于 CP0 寄存器的设计，根据讲义在 CP0 模块中添加 COUNT, COMPARE, BADVADDR 寄存器的相关逻辑。中断的判断放置在 ID 阶段，中断的处理是当中断标记的指令执行到 WB 阶段后进行统一处理。
2. Break 类型例外通过在 ID 阶段判断指令码是否是 Break 判断。
3. 保留指令例外通过设置一个判断信号，如果 ID 阶段取回的指令不符合当前译码模块已知的所有指令时，判断为保留指令例外，并标记在当前处于 ID 阶段的指令上。
4. 地址错包括两方面，一方面是判断 IF 阶段取指的地址是否对齐，不对齐的话，标记在 IF 阶段地址错例外。另一方面是在 EXE 阶段，判断发送给 data_ram 的地址是否对齐，若没有对齐，在 EXE 阶段标记地址错，在 WB 阶段汇总后进行处理。
5. 整数溢出的判断也是在 EXE 阶段，溢出判断包括两部分，一方面，在 alu 中判断加减法是否产生溢出错误，另一方面，判断此时的 es_pc 对应的指令是否是否是 add, sub, 或 addi 指令。
6. 如果出现与 MFC0 和 MTC0 有关的寄存器读相关，则将该流水级阻塞；
7. 写回级报例外时（ws_ex == 1'b1），也需要将各流水级刷新为无效状态，而且，如果写回级和访存级出现例外时，在执行阶段的写操作，如写 HI，LO 寄存器的操作，内存的写操作也将不能进行；

（二）总体设计图

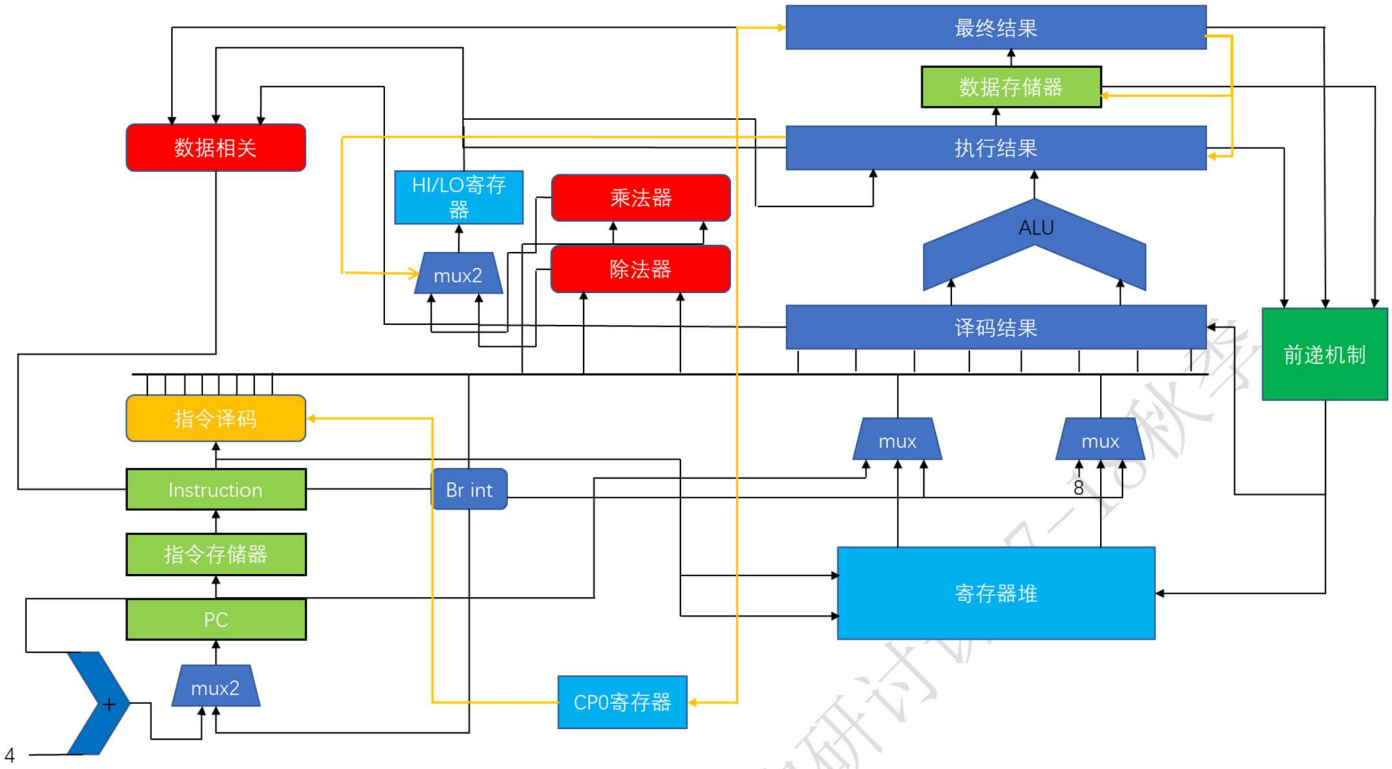

图 1 CPU 结构图(黄色为 lab9 添加)

（三）重要模块 1 设计：IF 阶段例外判断（地址错例外和延迟槽判断）

1、工作原理

如图，一方面，根据 fs_pc 判断是否发生地址错例外，（由于 fs_pc 和 next_pc 错着一个周期，因此若根据 nextpc 判断，需要使用一个寄存器保存判断结果下一拍使用），同时将 fs_badvaddr 设置为 fs_pc 的值。

另一方面，fs_delay_slot 是来自当前的 id 阶段的信号，说明当前 id 阶段的指令为跳转类指令，因此 fs_bd 设置为 1，说明此时 IF 阶段的指令位于延迟槽中。

```
reg fs_addr_error_r;
always@(posedge clk)begin
    if(reset)begin
        fs_addr_error_r <= 1'b0;
    end
    else
        fs_addr_error_r <= (nextpc[1:0] == 2'b0) ? 1'b0 : 1'b1;
end
assign fs_addr_error = fs_addr_error_r;
//assign fs_addr_error = (nextpc[1:0] == 2'b0) ? 1'b0 : 1'b1;
//if exception
assign fs_ex = fs_addr_error && fs_valid;
//the instruction is after branch
//assign fs_bd = br_taken;
assign fs_bd = fs_delay_slot;
assign fs_excode = `EX_ADEL;

//assign fs_badvaddr = nextpc;
assign fs_badvaddr = fs_pc;
endmodule
```

图 2 IF 阶段例外判断

2、功能描述

判断 IF 阶段指令是否有地址错例外并判断是否处于延迟槽中。

（四）重要模块 2 设计：ID 阶段例外判断

1、工作原理

1) has_int 信号通过当前的 cp0 寄存器的 cause 和 status 判断是否有中断发生，并通过 status 的 ie 和 exl 位判断此时是否允许中断。
2) inst_sys 用于判断当前指令是否是系统调用，inst_break 用于判断 break 指令，remd_inst 通过对所有指令取非的方式判断保留指令例外。
3) ds_excode 通过多路选择器的方式，实现不同例外的优先级，优先级顺序是中断＞取指的地址错例外＞保留指令例外＞系统调用 = break 指令

```
assign has_int = (((cause_ip[7:0] & status_im[7:0]) != 8'h00) && status_ie == 1'b1 && status_exl == 1'b0);

assign ds_ex = (has_int | inst_sys | inst_break | remd_inst | fs_to_ds_ex) & ds_valid;

assign ds_bd = fs_to_ds_bd;
assign ds_badvaddr = fs_to_ds_badvaddr;

assign ds_excode = ({5{has_int}} & `EX_INT)
                 |({5{fs_to_ds_ex}} & `EX_ADEL)
                 | ({5{remd_inst}} & `EX_RI)
                 | ({5{inst_sys}} & `EX_SYS)
                 | ({5{inst_break}} & `EX_BP);
```

图 3 ID 阶段例外判断

2、功能描述

判断 ID 阶段可能产生的例外，并根据优先级确定例外对应的操作码

（五）重要模块 3 设计：EXE 阶段例外判断

3、工作原理

1) Overflow 信号是 alu 模块传出的判断加减法是否溢出的信号，通过判断当前指令是否是 add,addi 和 sub 指令来判断是否存在例外。
2) Mem_addr_ex 是判断当前 EXE 阶段是否存在访存地址例外的信号，若访存地址出现例外，用 es_badvaddr 保存错误的访存地址。

```
//OverFlow
wire OverFlow_es;
assign OverFlow_es = overflow & OverFlow_inst;
assign es_ex = OverFlow_es| mem_addr_ex | ds_to_es_ex;
//assign es_ex = ds_to_es_ex;
assign es_bd = ds_to_es_bd;
assign es_badvaddr = mem_addr_ex ? data_sram_addr : ds_to_es_badvaddr;

assign es_excode = OverFlow_es ? (!ds_to_es_ex) ? `EX_OV : ds_to_es_excode:
                                 mem_addr_ex ? mem_excode : ds_to_es_excode;
```

图 4 EXE 阶段例外判断

4、功能描述

EXE 阶段判断加减法溢出例外和访存指令地址例外部分逻辑的

（六）重要模块 4 设计: CP0 寄存器中 COUNT , COMPARE, 和 BadVAddr 的实现

5、工作原理

1) Count 寄存器实现，通过一个 tick 信号每隔一周期取反一次，来实现 COUNT 寄存器每两拍加一的操作。

```
//CP0_COUNT++++++++++++++++++++++++++++++++++++++++++++++
reg tick;
always @(posedge clk)
begin
    if(rst)
        tick <= 1'b0;
    else
        tick <= ~tick;
end

reg [31:0] cp0_count_r;
always @(posedge clk)
begin
    if(rst)
        cp0_count_r <= 32'b0;
    else if(mtc0_we && cp0_addr == `CP0_COUNT_ADDR)
        cp0_count_r <= cp0_wdata;
    else if(tick)
        cp0_count_r <= cp0_count_r + 1'b1;
end

assign cp0_count = cp0_count_r;
```

图 5 CP0_COUNT 寄存器

2） COMPARE 寄存器，主要用于实现 mtc0 对该寄存器的写操作

```
//CP0_COMPARE+++++++++++++++++++++++++++++++++++++++++++++
reg [31:0]cp0_compare_r;
always @(posedge clk)
begin
    if(mtc0_we && cp0_addr == `CP0_COMP_ADDR)
        cp0_compare_r <= cp0_wdata;
end

assign cp0_compare = cp0_compare_r;
```

图 6 CP0_COMPARE 寄存器

3） BadVaddr 的实现，发成地址错例外时，会将错误地址在 WB 阶段例外汇总后保存到 BadVaddr 寄存器中。

```
//CP0_BADVADDR++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
reg [31:0] cp0_badvaddr_r;
always @(posedge clk)
begin
    //if(wb_ex && wb_excode == `EX_ADEL)
    if(wb_ex && (wb_excode == `EX_ADEL || wb_excode == `EX_ADES))
        cp0_badvaddr_r <= wb_badvaddr;
end

assign cp0_badvaddr = cp0_badvaddr_r;
```

图 7 CP0_badvaddr 寄存器

6、功能描述

CO0 寄存器的完善，实现了 COUNT, COMPARE 和 BadVaddr 寄存器。

三、实验过程（50%）

（一）实验流水账

1. 2019 年 11 月 2 日 12:00-14:00 完成 rtl 代码编写；
2. 2019 年 11 月 2 日 14:00-01：00，完成 debug，通过测试
3. 2019 年 11 月 4 日 19：00- 21：00，完成报告撰写

（二）错误记录

1、错误 1：仿真报错

（1）错误现象

测试 26 时报出错误。

（2）分析定位过程

测试 26 报错，理论上早就应该通过的，应该是修改的部分逻辑导致运行错误，查看发现测试减法指令 sub，说明应该是溢出的问题，调处相关波形图如下，可以看出，alu 在没有发生溢出的时候判断成了溢出并跳转到例外处理，造成了错误。

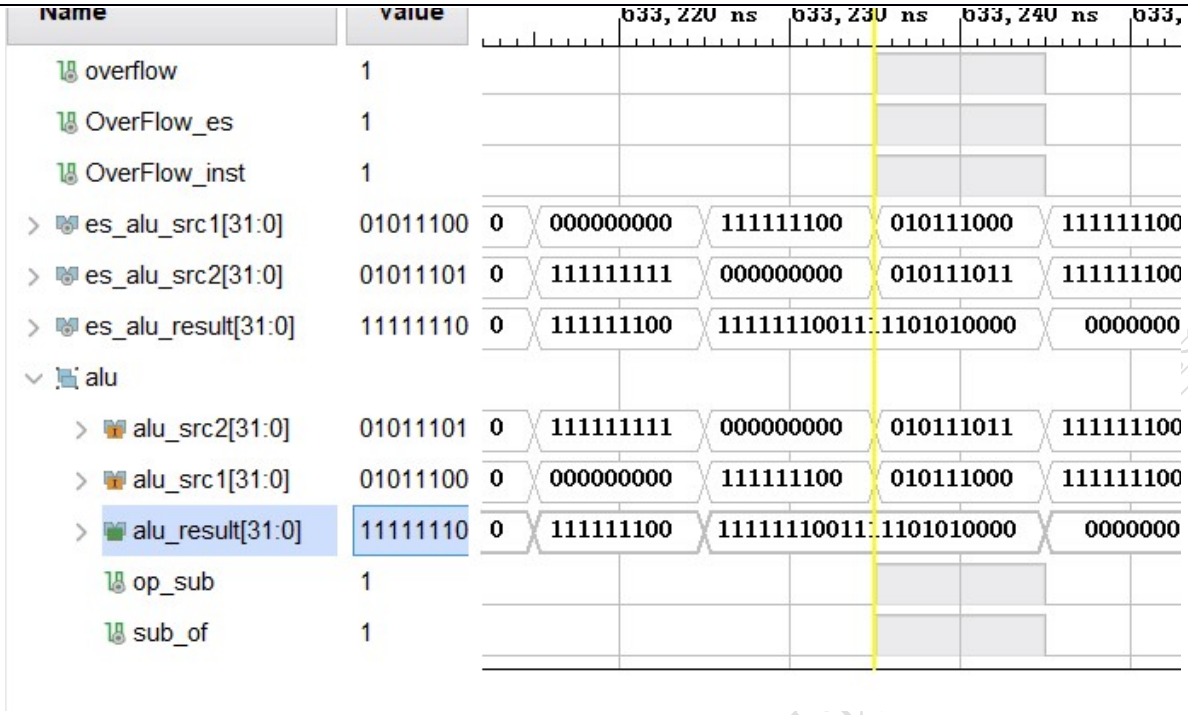

图 8 错误 1 波形图

（3）错误原因

减法判断溢出时没有用 alu 输入的 alu_src1 和 alu_src2，而是使用了已经按位取反加一的 adder_a 和 adder_b，造成减法时产生误判。

```
assign add_of = op_add & ((alu_src1[31] == 1'b0 && alu_src2[31] == 1'b0 && adder_result[31] == 1'b1)
                        | (alu_src1[31] == 1'b1 && alu_src2[31] == 1'b1 && adder_result[31] == 1'b0));

/*assign sub_of = op_sub & ( (adder_a[31] == 1'b0 && adder_b[31] == 1'b1 && adder_result[31] == 1'b1)
                        | (adder_a[31] == 1'b1 && adder_b[31] == 1'b0 && adder_result[31] == 1'b0));*/
assign sub_of = op_sub & ((~alu_src1[31] && alu_src2[31] && adder_result[31]) ||
                        (alu_src1[31] && ~alu_src2[31] && ~adder_result[31]));
```

图 9 错误 1 部分代码

（4）修正效果

修正后，顺利通过该测试。

2、错误 2：加法溢出处理的错误

（1）错误现象

```
[1674537 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x0001001a
    mycpu    : PC = 0xbfc3ba4c, wb_rf_wnum = 0x02, wb_rf_wdata = 0x64c76d7c
```

图 10 错误 2 报错

（2）分析定位过程

根据波形判断，该处加法 add 指令发生了例外，之后需要跳转到 0xbfc00380，观察波形没有什么错误，但是和对比而言就是差了两拍，在 piazza 上提问后，得到答案是发生溢出例外的计算指令不能写入寄存器，寄存器的写

使能信号需要置零，即不能让其修改处理器状态。但是增加了这点后，依旧会错在原处。再查看波形，寄存器的写使能信号还是被拉高了，一直找不出原因，后来发现有一个语法错误，这个模块没有 endmodule，添加之后就顺利通过该测试了。

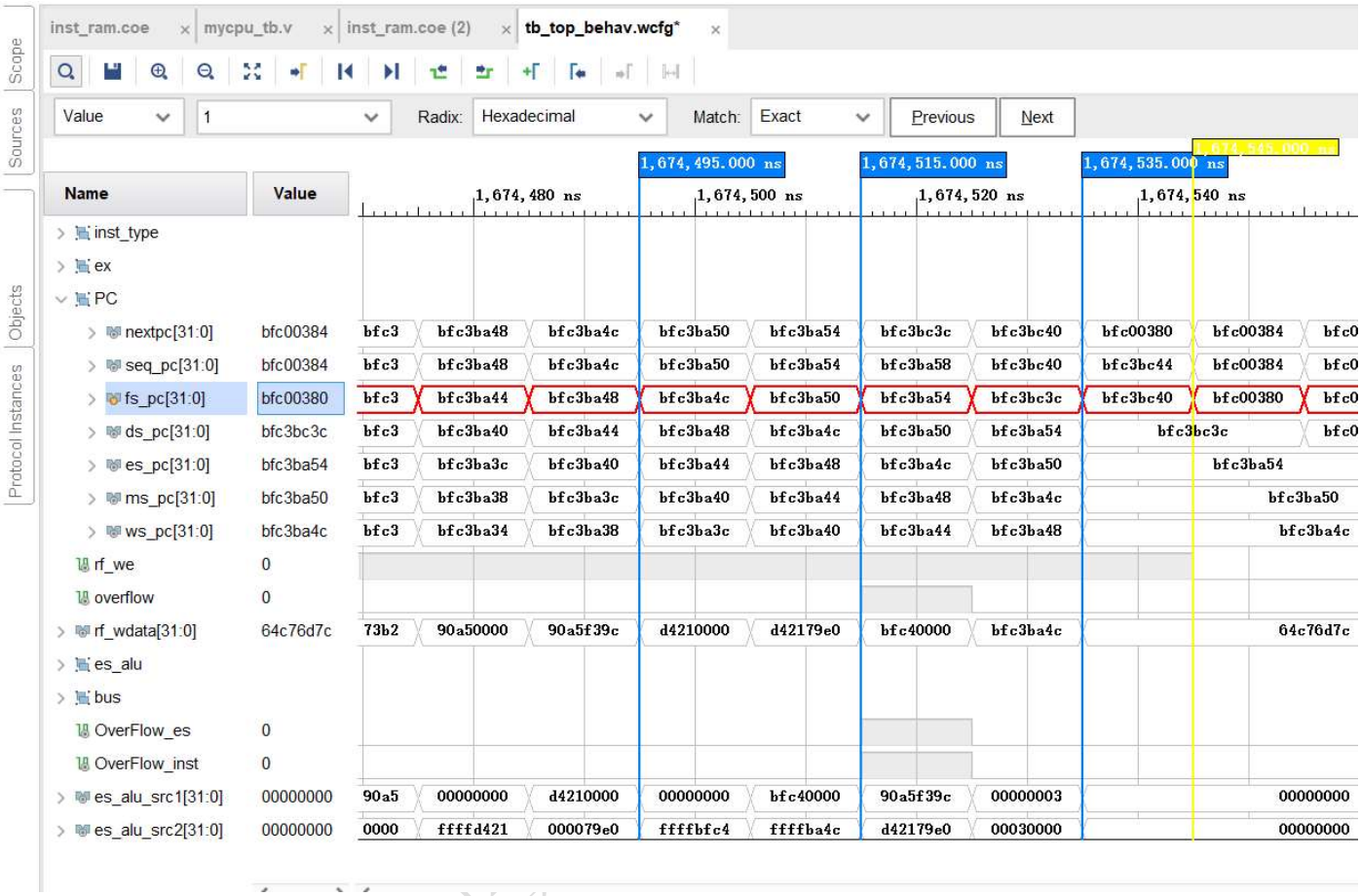

图 11 错误 2 波形图

（3）错误原因

溢出例外没有将寄存器写使能信号置零，EXE 模块缺少 endmodule

（4）修正效果

修正后，顺利通过测试点。

3、错误 3：延迟槽判断错误

（1）错误现象

```
1776747 ns] Error!!!
   reference: PC = 0xbfc003f4, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc97f58
   mycpu    : PC = 0xbfc003f4, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc97f5c
```

图 12 错误 3 报错

（2）分析定位过程

查看 inst 发现是系统调用的测试，查看反汇编文件知道是读 epc 寄存器读出的数据错误，syscall 恰好处于 beqz

指令之后，差值是 4 很容易判断出是延迟槽的问题，即处于延迟槽内的指令灭有被标记。查看波形图发现延迟槽标记在了 sys 后面的一条指令，而不是 syscall 指令。

图 13 错误 3 波形图

（3）错误原因

延迟槽标记出错，标记到了延迟槽指令之后的指令上。若当前阶段处于 ID 阶段的指令是 B/J 类指令时，此时处于 IF（fs_pc）的指令处于延迟槽中，应该标记给 fs_pc，而不是 next_pc。同时延迟槽的判断应该根据 ID 阶段的指令类型判断，而不是根据 ID 阶段传到 IF 阶段的 br_taken 判断，即不管是否跳转，B/J 类指令的下一条都处于延迟槽中。

（4）修正效果

修正后顺利向前进行。

4、错误 4：mfc0 读取 badaddr 错误

（1）错误现象

这个问题的现象和上一个问题一模一样：

```
----------------------------------------------------------------
[1718847 ns] Error!!!
    reference: PC = 0xbfc7e36c, wb_rf_wnum = 0x16, wb_rf_wdata = 0x800df545
    mycpu    : PC = 0xbfc7e36c, wb_rf_wnum = 0x16, wb_rf_wdata = 0x800dfc89
----------------------------------------------------------------
```

图 14 错误 4 报错

（2）分析定位过程

查看反汇编文件得知是 mfc0 读取 badvaddr 时读出的数据错误，badvaddr 用于保存访存地址例外时的错误地址，说明是访存地址例外。查看波形图 EXE 阶段发生例外时，es_badvaddr 顺利保存了错误地址，但是并没有保存到 CP0 寄存器中。查看 CP0 寄存器中逻辑发现了错误。

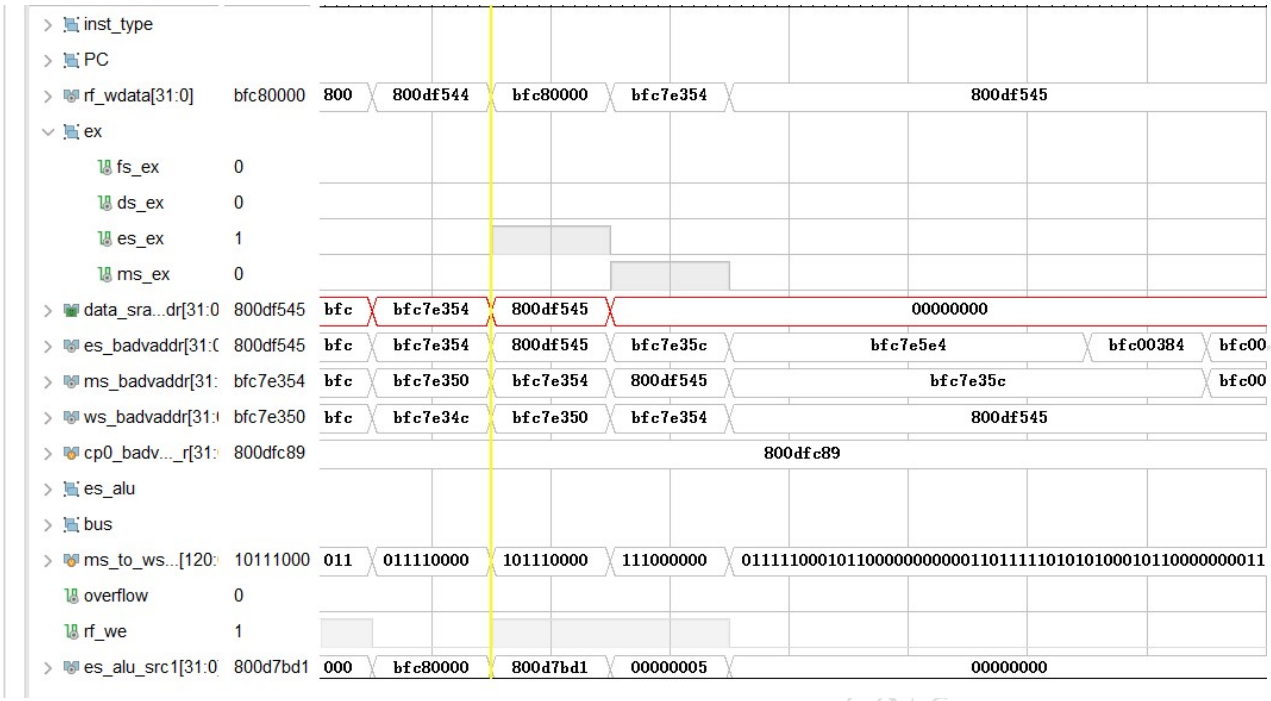

图 15 错误 4 波形图

（3）错误原因：

CP0_badvaddr 的例外只考虑了 ADEL 的地址例外码，而忽略了 S 类指令访存地址错误的例外码。

```
reg [31:0] cp0_badvaddr_r;
always @(posedge clk)
begin
    //if(wb_ex && wb_excode == `EX_ADEL)
    if(wb_ex && (wb_excode == `EX_ADEL || wb_excode == `EX_ADES))
        cp0_badvaddr_r <= wb_badvaddr;
end
```

图 16 错误 4 部分代码

（4）修正效果

修正后，顺利通过测试。

5、错误五： 死循环错误

（1）错误现象

程序进入死循环。

（2）分析定位过程

查看测试文件了解到是时钟中断的测试，那么死循环的原因就只可能是没有触发中断，查看波形图发现 CP0_COUNT = COMPARE 时，cause 的 ti 位被拉高了，但是 cause_ip 没有发生变化，而 has_int 是根据 cause_ip 判断的，所以出错。

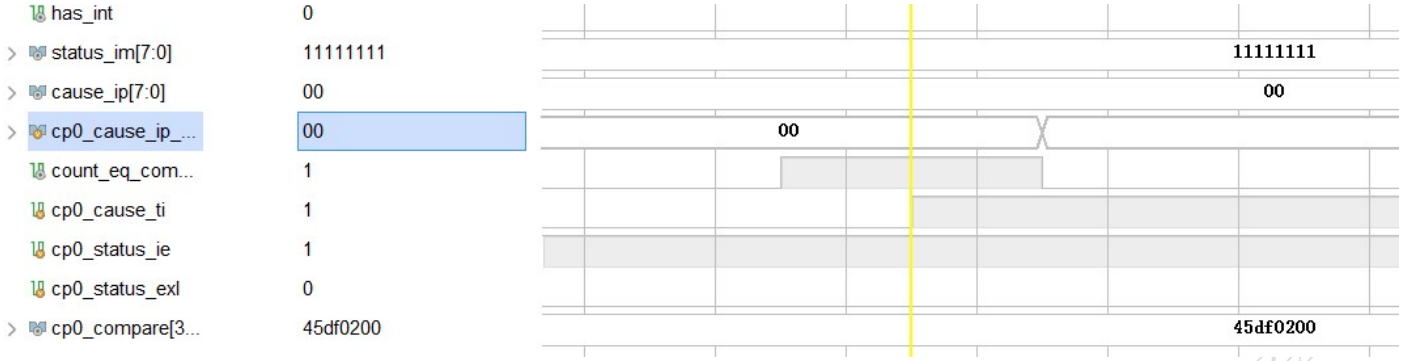

图 17 错误 5 波形图

（3）**错误原因：**

Cause 寄存器赋值错误，直接赋值的 32’b0。

（4）**修正效果**

修正后，顺利通过所有测试。

四、实验总结（可选）

1. 写实验前一定要先看讲义!!!不然可能出现像时钟中断这种只是标记差了一拍但是并不影响结果的错误。
2. 由于代码部分主要是队友在上周完成，这周只需要 debug。但是发现这种模式不太合理，给别人写的错误 debug 真的有一点费力，尤其时集中例外的实现上都有偏差和错误的时候。
3. 通过流水线清空处的 debug，加深了对代码中原有一些信号的理解，更清楚的能了解到当我想实现某个功能的时候为什么用这个信号而不用另一个。
4. 信号命名上还是避免重复最为方便，不然很可能因为信号名相同查看波形图时拿出了相同的信号而陷入找不到问题的困局。
5. 感觉把每个阶段的所有信号拉出来分组不太理性。相比而言，波形图中只留有一些常用的信号，当对自己代码的执行流程的数据通路时，debug 的流程可以转换为先看是哪个测试的错误，在具体看错在哪一项，通过搜索方式找出相关信号拉出波形图更为快捷。