实验 8 报告

学号：2017K8009922026，2017K8009922032
姓名：康齐瀚，赵鑫浩
箱子号：63

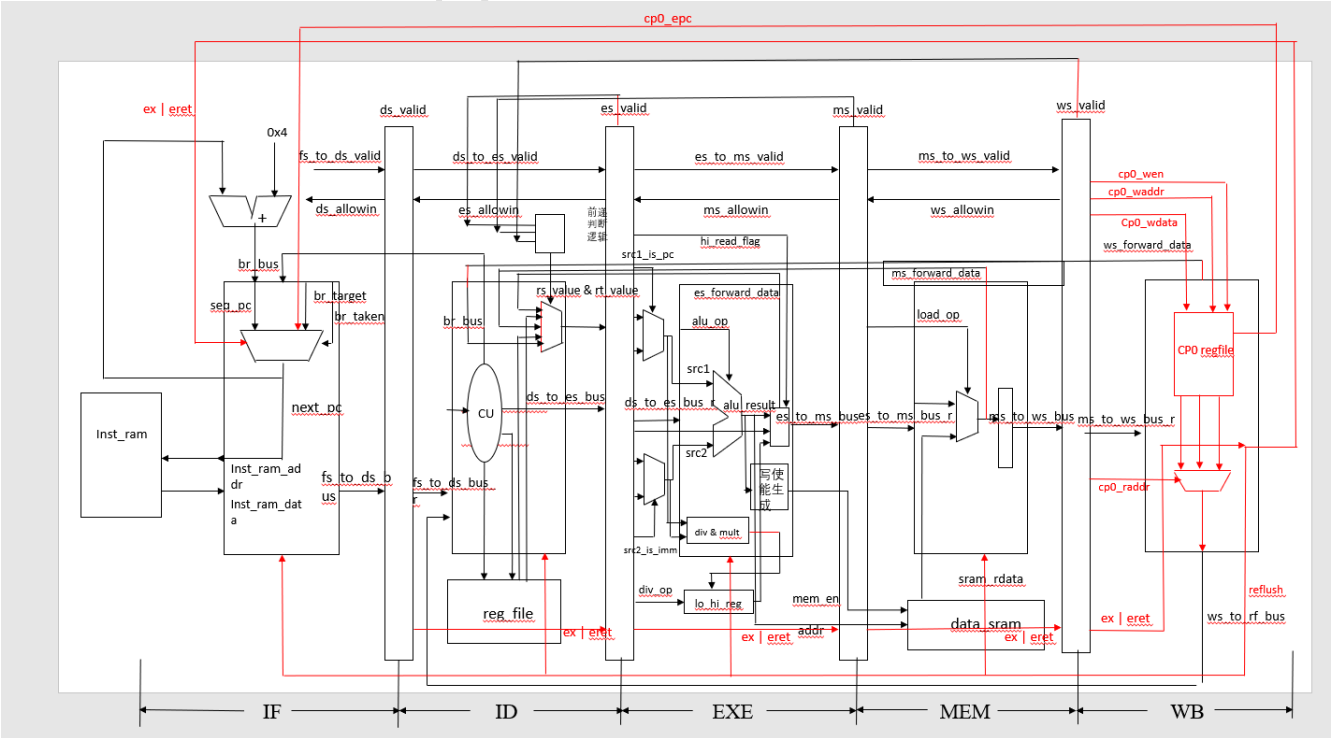
一、实验任务（10%）

向流水线中添加例外处理支持。本次实验需要添加 syscall 指令以及与例外处理相关的 mfc0，mtc0 指令，新增加 cp0 寄存器模块，并能进行合适的赋值的读取数据。最终要求能通过仿真并顺利上板通过测试。

二、实验设计（40%）

（一）总体设计思路

为了增加例外支持，需要在各级增加检测例外是否发生的相关信号。并且当某一流水级发生例外时，需要将例外信息沿着流水线传递下去，直到最后达到写回级进行例外处理。达到写回级时，需要将流水线进行刷新，我们采用的刷新方法是将每两个流水级之间的 bus 信号全赋值为 0，并且将所有流水级的 readygo 信号赋值为 1。这样实际上相当于在发生例外的指令与例外处理指令之间添加了若干的 nop 指令。同理，在 eret 返回刷新流水线时也需要进行同样的刷新操作。mfc0 与 mtc0 指令则需要从 ID 级开始在流水线中传递需要读或写的 CP0 寄存器地址，直到最后在 WB 级写回对应的寄存器。



（二）重要模块 1 设计：CP0 regfile

1、工作原理

CP0 regfile 模块提供了 CP0 协处理器需要的寄存器。本次实验中需要添加 cp0_status, cp0_epc, cp0_cause 等寄存器。通过三个输出端口

2、接口定义

名称	方向	位宽	功能描述
Clk	In		时钟信号
reset	In		复位信号
ws_ex	In	1	例外信号
ws_bd	In	1	判断例外信号是否处于延时槽中
mtc0_we	In	1	有效的 mtc0 信号
ext_int_in	In	6	外部接入的例外信号
cp0_wdata	In	32	写入 cp0 寄存器的数值
ws_excode	In	5	例外类型
ws_pc	In	32	写回级的 pc 值
cp0_addr	In	8	判断 cp0 寄存器的类型
eret_flush	In	1	判断 Eret 指令
cp0_result	Out	32	存入寄存器的 cp0 寄存器的值
cp0_epc	Out	32	epc 寄存器的值

3、功能描述

当例外发生或 eret 指令出现在 WB 级时，我们将例外的信息：包括种类和例外信号是否处于延时槽等信号接入 cp0 模块来改变 cp0 寄存器的值，并将下一拍的 pc 跳转至中断处理入口，当有 mtc0 和 mfc0 信号时，我们可以取出 cp0 寄存器的值或将寄存器的值存入。

在 CP0 regfile 的模块设计上，并没有采用类似通用寄存器的写法，而是将需要设置的 CP0 寄存器的域单独抽离出来。以 CP0 status 寄存器的 im 域为例，将其声明为一个 reg 型变量。有以下赋值操作：

```
always@(posedge clk) begin
    if(reset)
        cp0_status_im <= 8'b0;
    else if(mtc0_we && cp0_waddr == 5'b01100)
        cp0_status_im <= cp0_wdata[15: 8];
end
```

其他 CP0 寄存器的域也是相同的处理。

本次实验中仅添加了 CP0 status，cause 和 epc 寄存器。输出时需要将它们每一个域与若干个 0 拼接后作为整个模块的一个输出。如下图所示：

```
assign cp0_status_reg = {
                        9'd0            ,       // 31:23
                        cp0_status_bev  ,       // 22
                        6'd0            ,       // 21:16
                        cp0_status_im   ,       // 15:8
                        6'd0            ,       // 7: 2
                        cp0_status_exl  ,       // 1
                        cp0_status_ie           // 0
assign cp0_cause_reg = {
                       cp0_cause_bd             ,   // 31
                       cp0_cause_ti             ,   // 30
                       14'd0                    ,   // 29:16
                       cp0_cause_ip             ,   // 15: 8
                       1'd0                     ,  // 7
                       cp0_cause_excode         , // 6: 2
                       2'd0
                       };
```

再由 WB 流水级的 mfc0_raddr 来选择对应的数据。

（三）重要模块 2 设计：分散在各个流水级的例外检测

每个流水级都有相应的例外检测，它们主要有两个功能，第一是传递来自上一个流水级的例外信息，第二是检测指令执行到当前流水级的时候是否有新的例外产生。同样，在上一级的例外信息和当前流水级例外信息的取舍上，应该是先发生的例外被继续传递下去。

取指级的例外检测：

```
assign invalid_addr_ex = (nextpc[1: 0] != 2'b0);
assign if_ex           = (fs_valid) ? invalid_addr_ex : 1'b0 ;
assign if_excode       = (if_ex) ? 5'h04 : 5'h0;
assign fs_excode       = (fs_valid) ? if_excode : 5'b0;


assign ds_ex           = (ds_valid) ? (fs_ex | id_stage_ex) : 1'b0;
assign ds_excode       = (ds_valid) ? ((fs_ex)        ? fs_excode :
                                       (id_stage_ex)  ? id_stage_excode : 5'b0)
                                     : 5'b0;
```

译码级的例外检测：

```
assign id_stage_ex     = inst_syscall;
assign id_stage_excode = ({5{inst_syscall}} & 5'h08);


assign ds_ex           = (ds_valid) ? (fs_ex | id_stage_ex) : 1'b0;
assign ds_excode       = (ds_valid) ? ((fs_ex)        ? fs_excode :
                                       (id_stage_ex)  ? id_stage_excode : 5'b0)
                                     : 5'b0;
```

执行级的例外检测：

```
assign ex_overflow          = overflow_inst & overflow;
assign ex_invalid_store_addr = (op_sh & vaddr[0])                    //sh
                             | (op_sw & (vaddr != 2'b0));            //sw

assign ex_invalid_load_addr  = (es_load_store_op[11] & (vaddr != 2'b00))//lw
                             | (es_load_store_op[ 8] & vaddr[0]       )//lh
                             | (es_load_store_op[ 7] & vaddr[0]       );//lhu

assign exe_stage_ex         = ex_overflow | ex_invalid_load_addr | ex_invalid_store_addr;
assign exe_stage_excode     = ({5{ex_overflow}}          & 5'h0c)
                            | ({5{ex_invalid_load_addr}} & 5'h04)
                            | ({5{ex_invalid_store_addr}} & 5'h05);
assign es_ex                = (es_valid) ? (exe_stage_ex | ds_ex) : 1'b0;
assign es_excode            = (es_valid) ? ((ds_ex)        ? ds_excode :
                                            (exe_stage_ex) ? exe_stage_excode : 5'b0)
                                          : 5'b0;
```

其中，对访存时访存地址的合法性检验(无论是 store 还是 load 型指令)都是放在 exe 级别进行检测的，这样无需在 EXE 级对 lw 型指令进行进一步的地址合法性检验。代码中的*_stage_excode 和*_stage_ex 均表示当前流水级新产生的例外信号。ds_ex, fs_ex 等则表示上一流水级的例外信号。在选择传递至下一级的例外信号时，先优先考虑该指令先发生的例外。

三、实验过程（50%）

（一）实验流水账

2019/10/24　14:00 ~ 22: 00　初步完成代码，仿真能够通过 syscall 测试的第一个测试

2019/10/25　9:00 ~ 9:50　修改代码，成功通过所有测试，并上板成功

（二）错误记录

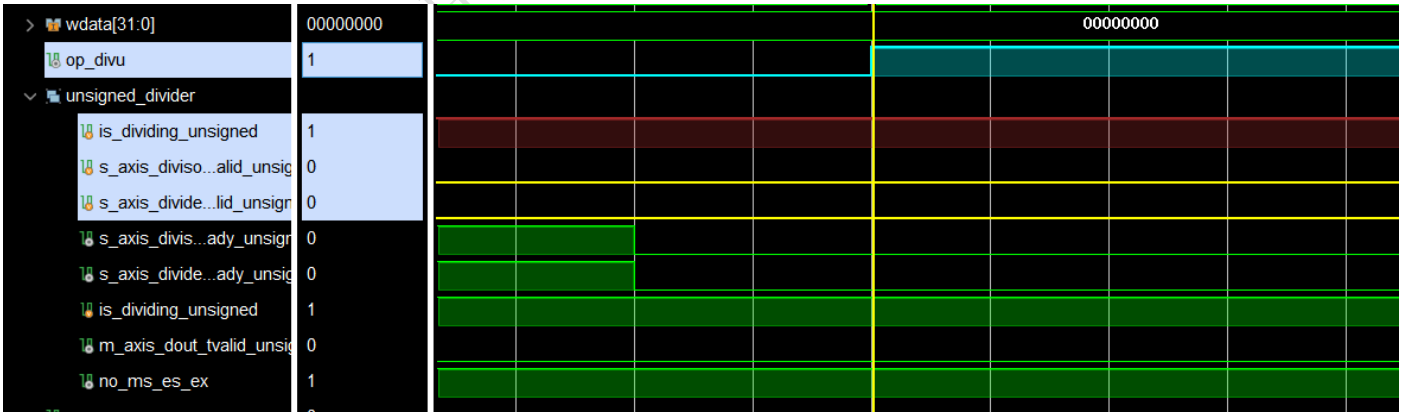
1、错误 1：PC 值一直保持为 0x00000000

（1）错误现象

在流水线运行通过前面 68 个测试点后，在某个位置 debug_PC 一直保持为 0x00000000，流水线无法继续运行下去。

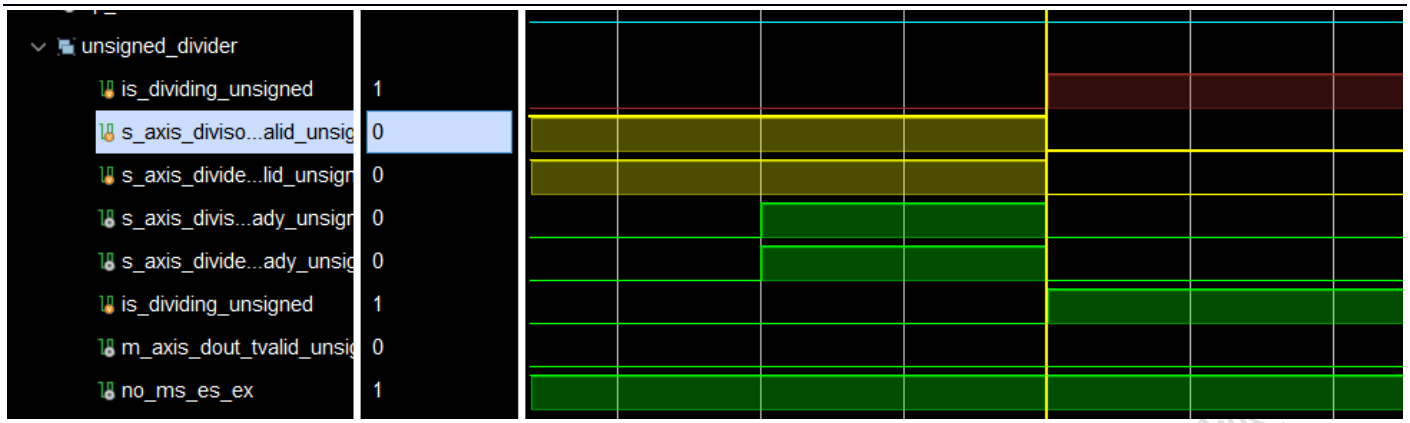
（2）分析定位过程

观察波形可以发现，流水线一直被阻塞在 0xbfc1de80 这个位置，通过 test.s 查询这条指令，发现是一条 bne 指令。再向前可以发现前面是一条 divu 指令，因此该指令被阻塞在 id 级别很有可能是因为 divu 指令导致 es_ready_go 变为 0 而阻塞。

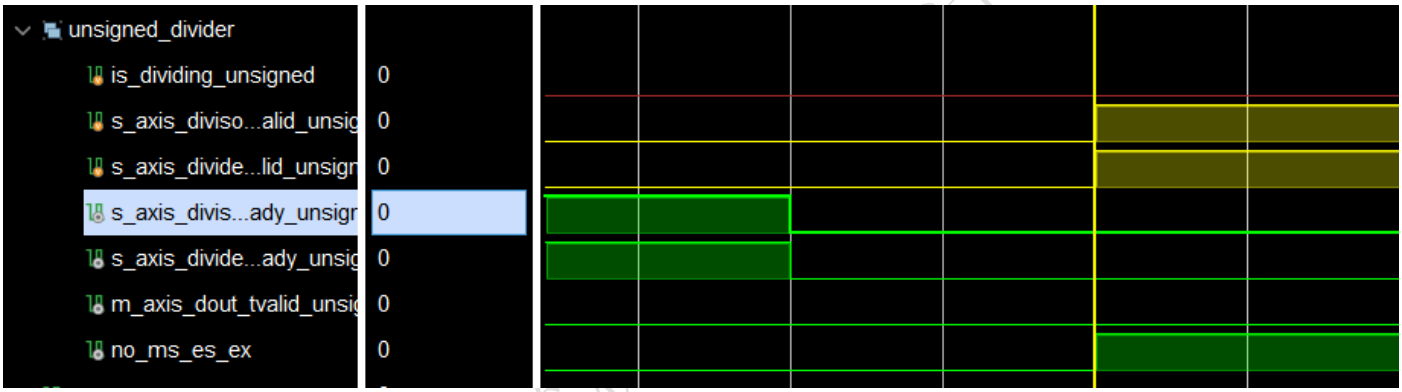


观察这条 divu 指令来到执行级时可以发现，该指令需要的除数和被除数的握手信号 tvalid 始终没有被拉高来发出除法运算请求。仔细观察发现，一个用来标识当前是否正在进行除法运算的 is_dividing_unsigned 信号一直是 1，因此 tvalid 没有拉高。

我们找到最近的一次导致该信号被拉高的地方，如下图所示：



一直向后搜索发现表示该 dout_valid 信号一直没有拉高，因此可能是除法器出现了问题。应该是在 tvalid 的赋值时机不对。



在最下面的 no_ms_es_ex 信号表示的是当前 mem 级与 ws 级是否有携带例外的指令，此时 no_ms_es_ex 为 0，表示这两个流水级中有一个带有例外，此时不应该把两个 valid 信号拉高来进行除法运算。否则当例外处理到来时，由于 es_readygo 信号为 0，整个流水线都被一直阻塞。

（3）错误原因

在后面几级流水线中携带有例外标识指令时，开启了除法器的运算。

（4）修正效果

在对除法器的两个运算数的握手信号进行赋值时，需要保证当前后面两个流水级没有携带例外信息，否则就不要将两个握手信号拉高进行握手。

修改如下：

```
always@(posedge clk) begin
    if(reset) begin
        s_axis_divisor_tvalid_unsigned <= 1'b0;
    end
    else if(is_dividing_unsigned) begin
        s_axis_divisor_tvalid_unsigned <= 1'b0;
    end
    else if(s_axis_divisor_tready_unsigned) begin
        s_axis_divisor_tvalid_unsigned <= 1'b0;
    end
    else if(op_divu & ~is_dividing_unsigned & no_ms_es_ex) begin
        s_axis_divisor_tvalid_unsigned <= 1'b1;
    end
end
```

修改之后，能顺利通过该次 syscall 测试点

（5）归纳总结（可选）

出现这样的 bug 主要还是写代码之前情况考虑不清晰导致的。

2、错误 2：寄存器写回错误

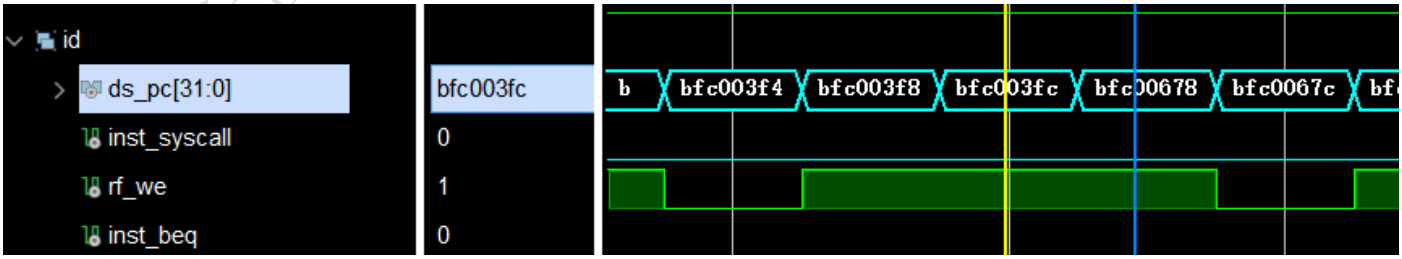
（1）错误现象

执行到某一位置时，debug 信号的 PC 错误，但是写回寄存器号以及写回寄存器的数据都是正确的，如下如所示：

```
[1658677 ns] Error!!!
    reference: PC = 0xbfc00400, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000020
    mycpu    : PC = 0xbfc00678, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000020
```

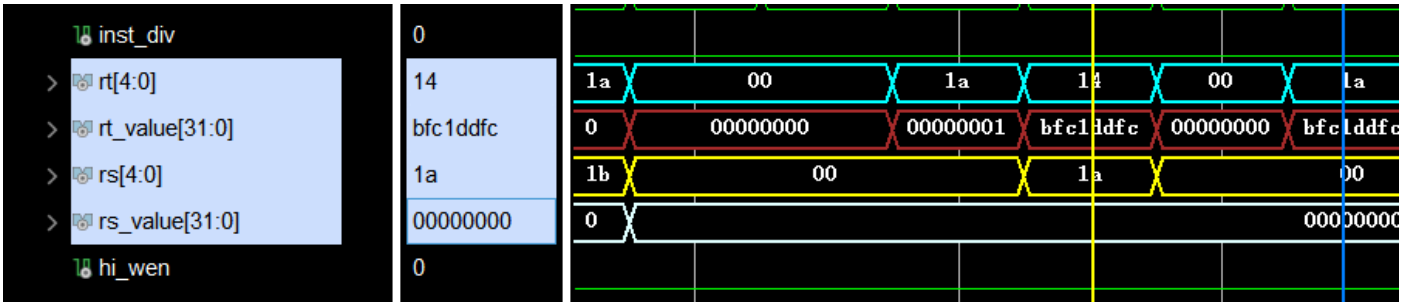
（2）分析定位过程

通过 test.s 文件可以发现 0xbfc00400 和 0xbfc00678 两个位置的指令都是 mfc0, k0, cause。观察波形可以发现，在 0xbfc00678 之前的指令是 0xbfc003fc, 因此很大可能发生了跳转：



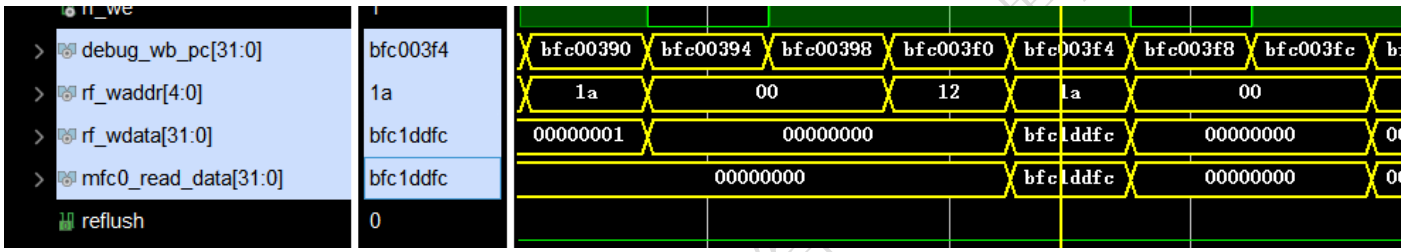
对比 test.s 中的指令序列可以发现，0xbfc003fc 之前是一条 bne 指令，结合 referrence 给出的 PC 值可以猜测，可能是在不应该发生跳转的情况下发生了跳转。

截取 bfc003f8 位置的波形图如下：



该指令需要读取 k0 和 S4 寄存器的值并进行比较，从这里的波形图可以看出二者并不相等，因此发生了跳转。但是从 rs 寄存器中读出了 0x00000000 十分值得怀疑，通过 test.s 可以发现最近的一次更新 0x1a 号寄存器正是上一条指令 mfc0　k0, c0_epc. 去读上一个周期时的 c0_epc 寄存器数据发现，该数据正好是 bfc1ddfc。因此应该是寄存器写回时的问题。

观察这个周期 rs_value 的所有来源：所有的 forward_data， 以及 rf_rdata 发现均不为 0xbfc11dfc. 因此可以确定是阻塞的问题。



在这个周期发现写回数据，写回寄存器号以及 PC 均完全符合。但此时的译码级 PC 却是 0xbfc00678,因此是 id 级的阻塞信号设置有误。

（3）错误原因

Id 级流水线 ds_readygo 信号设置有误，没有恰当地进行阻塞。

（4）修正效果

对 ds 级的 readygo 信号进行恰当的赋值，当检测到 es 或 ms 级有 mfc0 的操作并且该操作的写回地址与当前要读取的寄存器号相同，以及写使能有效时，就暂时阻塞直到指令到达 WS 流水级完成数据的读取。如下图所示：

```
assign ds_ready_go  =  (reflush) ? 1'b1 :
                       (is_lw == 1'b1 && es_is_valid && ((rf_raddr1 == es_waddr && es_wen) || (rf_raddr2 == es_waddr && es_wen))) ? 1'b0 :
                       (es_is_valid && es_mfc0_read && es_waddr != 5'b0 && (es_waddr == rf_raddr1 | es_waddr == rf_raddr2) && es_wen) ? 1'b0 :
                       (ms_is_valid && ms_mfc0_read && ms_waddr != 5'b0 && (ms_waddr == rf_raddr1 | ms_waddr == rf_raddr2) && ms_wen) ? 1'b0 : 1'b1;
```

3、错误 3：写回寄存器时 PC 错误

（1）错误现象

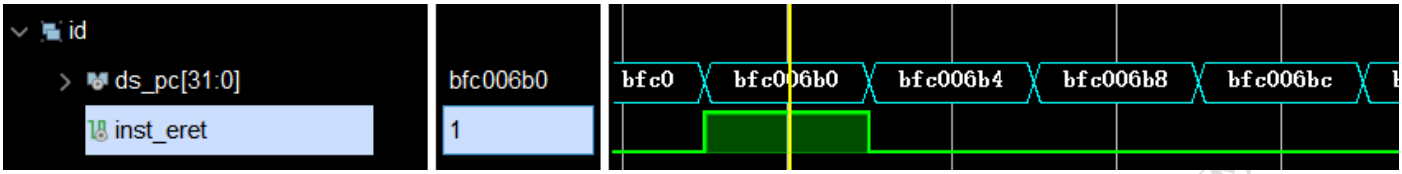
在运行过程中发生写回阶段写回 PC 错误。如下图所示：

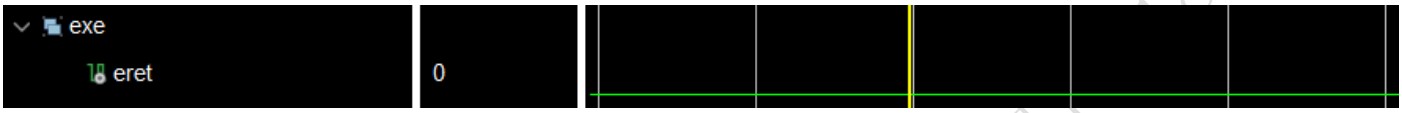
```
[1658997 ns] Error!!!
    reference: PC = 0xbfc1de08, wb_rf_wnum = 0x12, wb_rf_wdata = 0x00000001
    mycpu    : PC = 0xbfc006b8, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
```

（2）分析定位过程

查询 test.s 发现 0xbfc006b8 位置是一条 lui 指令。观察波形：

该指令序列前面是一条 nop 指令以及 eret 指令，然而从后面的指令执行序列来看，eret 指令并没有执行。





观察上面的波形可以发现，在 bfc006b0 位置的这条指令确实被译码为了 eret 指令。然而传递到下一流水级时的 eret 信号却变为了 0。考虑是在两个流水级之间传递的 ds_to_es_bus 信号除了问题。

观察代码：

```
`define DS_TO_ES_BUS_WD 241
assign ds_to_es_bus = (reflush) ? 228'b0: { inst_eret    , // 241
                                            fs_bd        , // 240
                                            mfc0_read    , // 239
                                            mtc0_we      , // 238
                                            cp0_waddr    , //237:233
                                            cp0_raddr    , //232:228
```

发现是 ds_to_es_bus 信号的长度有问题，少了的这一位恰好是 eret 信号，因此导致后面流水级接收到的 eret 信号一直为 0.

（3）错误原因

宏定义中的信号位宽定义与实际代码中使用到的位宽定义不一致

（4）修正效果

修改宏定义中的 ds_to_es_bus 的位宽为 242 位，最终能成功通过测试。

4、错误 4：流水线清空时出现错误

（1）错误现象

从 wb 级接回 eret_flush 后，将流水线清空，但发现无法通过任何测试 PC 值为 X

（2）分析定位过程

观察最初波形后发现，由于 eret_flush 还没产生，此时导致流水线的 bus 无法成功传递。所以我在初始化 eret_flush 时与上 ws_valid 信号保证该信号最开始就有值，确保流水线正常运作。

（3）修正效果

修改 eret_flush 后流水线正常运作。

5、错误 5：ready_go 信号出现问题

（1）错误现象

执行 syscall 后发现之后几拍的一条指令写回值发生错误。观察波形后发现，与准确值相差了一拍，

（2）分析定位过程

观察波形后发现，与准确值相差了一拍，之后观察各级 ready_go 信号发现 exe 级发生了阻塞。

（3）错误原因

Syscall 后因为流水线清空的原因，必须立即将前面各级的 ready_go 信号赋成 1, 避免阻塞。

（4）修正效果

修改 ready_go 信号后流水线正常运作。

四、实验总结

通过这次实验，我们学会了 CPU 对例外和中断处理的具体流程，更加深入地理解了精确例外的概念和处理方法。尽管这次实验仅仅只是添加了 syscall 例外处理以及 mfc0 和 mtc0 这几条指令，但整个 CPU 例外处理的框架已经大致完成了，后面的需要继续添加的例外相对来说应该简单一些。不过这次实验大概可以算得上是近几次实验中最困难的一次了，很多细节处理，比如在 mfc0 后的指令如果需要使用到 mfc0 指令的结果时需要阻塞，以及 EXE 级的写内存请求，除法器握手信号，对 hi，lo 寄存器的写使能信号，在发生了例外时都需要进行相应的设置。这些问题虽然在讲义中提到了，但实际写代码时难免有遗漏。不过，老师提供的 CP0 寄存器按域划分的写法相对于通用寄存器的写法确实简单了很多。