实验 9 报告

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

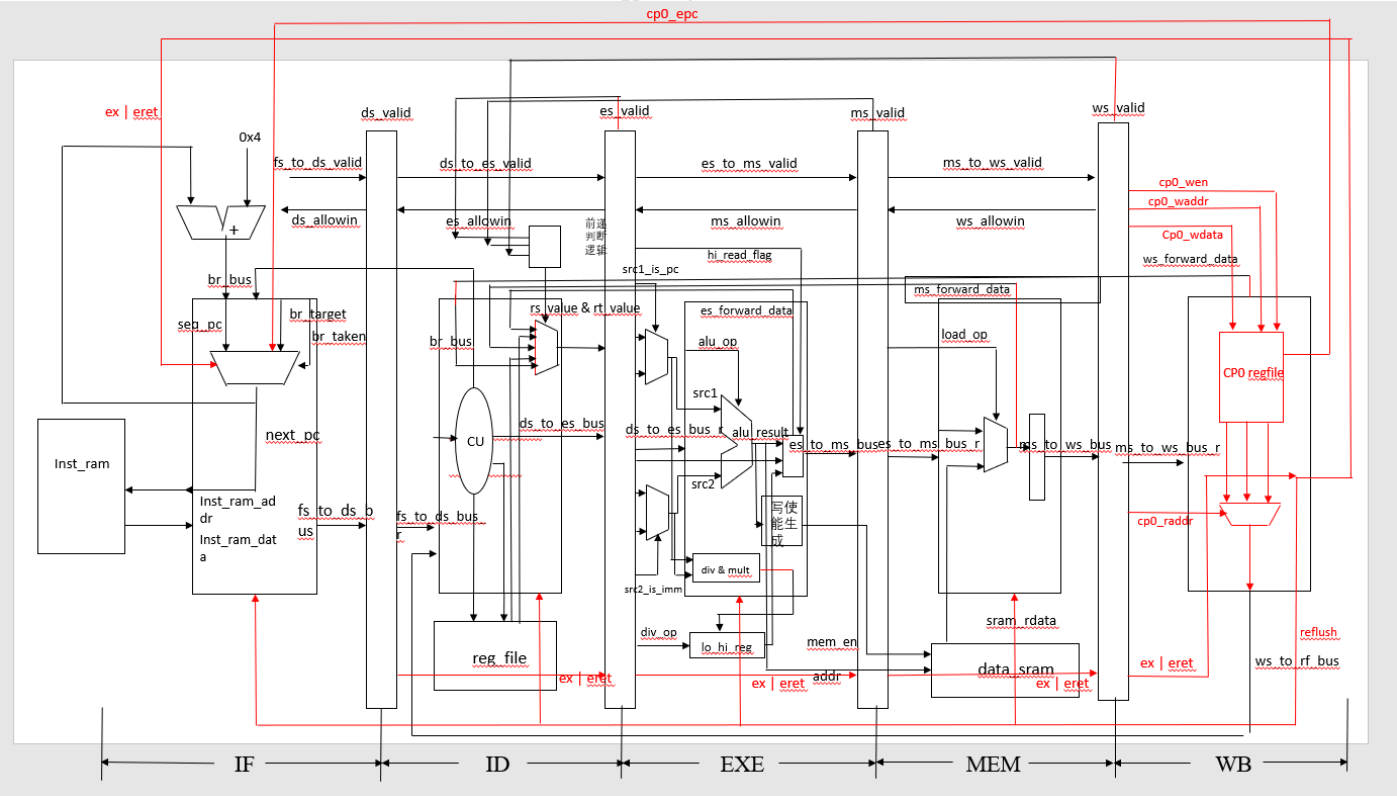
一、实验任务（10%）

增加对地址错例外，整型运算溢出例外以及对时钟中断和软件中断的支持。要求最终能顺利通过仿真并成功上板。

二、实验设计（40%）

（一）总体设计思路

在上周的基础上，添加了其他数据通路用于处理其他例外，将所有例外信号传至 wb 级进行处理，并将结果返回流水线前面各级，新增 compare，count，badvaddr 三个 cp0 寄存器。



（二）重要模块 1 设计：cp0_compare，count，badvaddr

1、工作原理

对三个 cp0 进行赋值并接入 cp0 模块的输出。

2、功能描述

```
reg [31:0] cp0_badvaddr;

always @(posedge clk)
begin
    if(ws_ex && (ws_excode == 5'h04 | ws_excode == 5'h05 ) )
    //if(ws_ex && ws_excode == 5'h04  )
    begin
        cp0_badvaddr <= ws_badvaddr;
    end
end

always @(posedge clk)
begin
    if(reset)
    begin
       cp0_compare <= 1'b0;
    end
    else if (mtc0_we && cp0_addr == cr_compare)
    begin
        cp0_compare <= cp0_wdata;
    end
end

always @(posedge clk)
begin
    if (reset) tick <= 1'b0;
    else tick <= ~tick;

    if(reset)
    begin
       cp0_count <= 1'b0;
    end
    else if (mtc0_we && cp0_addr == cr_count)
    begin
        cp0_count <= cp0_wdata;
    end
    else if (tick)
    begin
        cp0_count <= cp0_count + 1'b1;
    end
end
```

（三）重要模块 2 设计：在各流水级接入的例外信号。

```
assign fs_pc_adel = (fs_pc[1:0]==2'b0)? 1'b0 :1'b1;
```

```
assign ds_reserve = ~(inst_addu | inst_subu | inst_slt | inst_sltu | inst_and | inst_or | inst_xor | inst_nor | inst_sll | inst_srl |
                    inst_sra | inst_addiu | inst_lui | inst_lw | inst_sw | inst_beq | inst_bne | inst_jal | inst_jr | inst_add | inst_adc
                    inst_sub | inst_slti | inst_sltiu | inst_andi | inst_ori | inst_xori | inst_sllv | inst_srav | inst_srlv |
                    inst_mult | inst_multu | inst_div | inst_divu | inst_mfhi | inst_mflo | inst_mthi | inst_mtlo | inst_bgez | inst_bgtz
                    inst_blez | inst_bltz | inst_j | inst_bltzal | inst_bgezal | inst_jalr | inst_lb | inst_lbu | inst_lh | inst_lhu | ins
                    inst_lwr | inst_sb | inst_sh | inst_swl | inst_swr | inst_mtc0 | inst_mfc0 | inst_eret | inst_syscall | inst_break );
```

```
assign es_ades = (es_sw & address_low != 2'b0 ) | ( es_sh & address_low[0] != 1'b0);
```

```
assign ms_load_adel = ( ms_lw & addr_low != 2'b0 ) | ( (ms_lh |ms_lhu) & addr_low[0] != 1'b0) ;
```

```
assign clock_interrupt = (count_eq_compare | ((cp0_cause_ip & cp0_status_im) != 8'b0) )& ~cp0_status_exl & cp0_status_ie;
```

在各级计算出例外信号后随流水线传入 wb 级处理。

（三）重要模块 3 设计：关于中断的设计。

中断分为两种：时钟中断和软件中断，不管哪种中断，发生的必要条件：cp0_status_ie=1,cp0_status_exl=0 以及某个中断源产生中断且该中断源未被屏蔽，在我们的代码中中断信号分成三部分，第一部分为判断是否产生了时钟中断（compare 寄存器的值等于 count 寄存器的值）或软件中断（cause 寄存 ip 的某一位和 status 寄存器 im 的某一位恰好同时为 1）。判断完中断信号后我们将中断信号标记在译码级的指令上并通过流水级传递下去，在写回级进行例外的处理。对于各类中断，我们统一跳转到例外入口进行处理。


```
assign clock_interrupt = (count_eq_compare | ((cp0_cause_ip & cp0_status_im) != 8'b0) )& ~cp0_status_exl & cp0_status_ie;
```


三、实验过程（50%）

（一）实验流水账

2019/10/31　15：00 ~ 21：00　写代码并顺利通过仿真并成功上板

2019/11/ 2　14：00 ~ 15：00　顺利运行通过记忆游戏

（二）错误记录

1、错误 1：寄存器写回错误

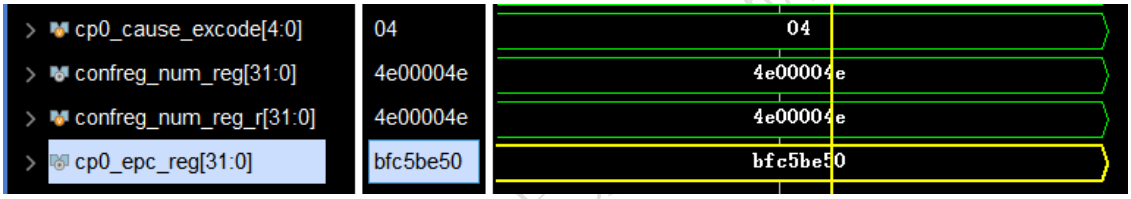
（1）错误现象

在取指地址例外的测试样例中，某处寄存器写回错误，如下图所示：

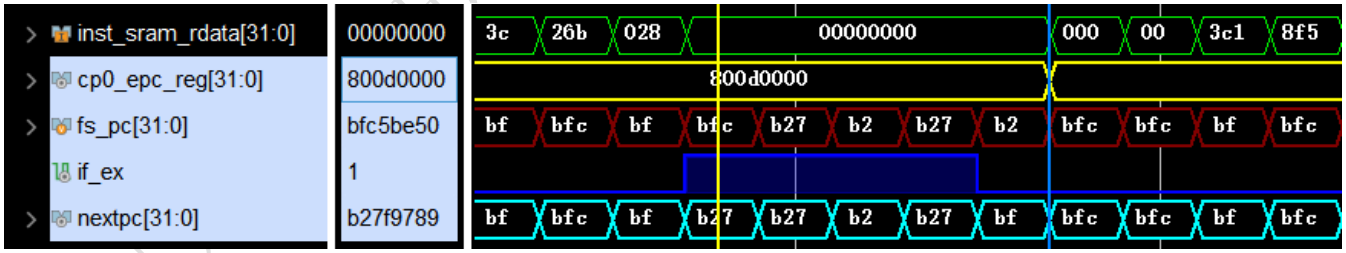
```
[1725457 ns] Error!!!
    reference: PC = 0xbfc00544, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xb27f9789
    mycpu    : PC = 0xbfc00544, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc5be50
```

（2）分析定位过程

报错的 PC 地址是 0xbfc00544, 通过查询 test.s 文件可以发现，此条指令是一条 mfc0　k0, c0_epc 指令。通过在 WB 阶段查询 epc 寄存器的值，如下所示，发现 与报错时 myCPU 的值相同，因此可以排除是 mfc0 译码错误的原因。



因此发生错误的原因是上一次 cp0_epc 寄存器的写入有误。需要找到上一次 epc 写入的位置。由于例外发生时是在 WB 级写入了 CP0 寄存器，因此需要向前推进 5 个时钟周期才能发现该指令取值时是否发生了错误。如下图所示：



可以看出，在五个周期前，取值地址是 0xbfc5be50，但是这条指令却在取值级被标上了 if_ex, 即取指例外。因此，这条指令的 fs_pc 在流向 cp0 寄存器时就被写入了 cp0_epc。这显然是错误的。观察 if_ex 的生成逻辑，如下所示：

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

发现我的最初设想是用 nextpc 来判断取值地址是否有效，但是结合提供的 reference 来看，还是应该使用地址无效的那条指令的 PC 来存入 epc 寄存器。

（3）错误原因

在判断是否有取指地址错误例外时，应该使用本条指令的 PC，即 fs_pc 而不是 nextpc 的值来判断取指地址是否有效。最初的设计中采用了用 nextpc 来判断的做法，因为我们认为发出一个非法的取值地址根本无法得到 inst_sram 的有效回应，需要将上一条正确的指令来重新运行一次以生成正确的取值 PC。

（4）修正效果

将判断取值地址是否错误的逻辑中的 nextpc 修改为 fs_pc。修改后就能成功通过该点测试。

（5）归纳总结（可选）

这个错误是由于没有考虑清楚例外是如何处理而导致的。对于 inst_sram 的行为的不了解也错误地让我们以为在例外处理中需要通过上一条正确指令的运行来得到下一条指令的正确取指地址。

2、错误 2：寄存器写回值错误

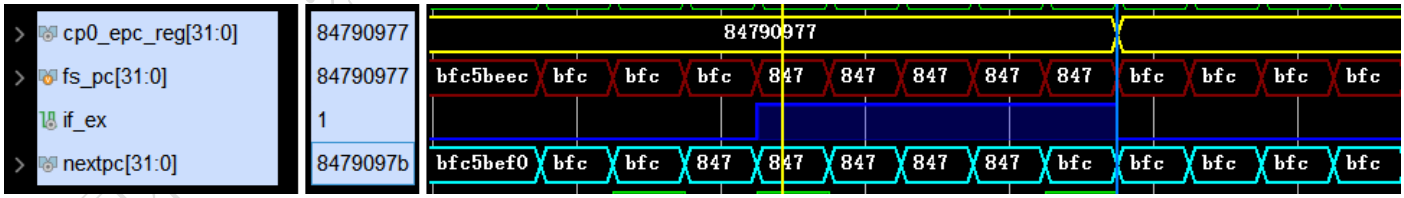
（1）错误现象

在取指地址例外的测试样例中，某处寄存器写回错误，如下图所示：

```
[1727747 ns] Error!!!
    reference: PC = 0xbfc00544, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x84790977
    mycpu    : PC = 0xbfc00544, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x84790973
```

（2）分析定位过程

报错 PC 仍然为 0xbfc00544，还是 epc 寄存器的值错误。和错误一不同，这次发生错误的比对 wdata 和 myCPU 写回的 wdata 只相差了 4，几乎就是两条相邻指令的差别。考虑到例外总是处理最先发生的一条指令，因此 0x8490977 的上一条指令是正确执行的，即它的 PC 应该是 0xbfc……



从上面的波形图可以看出，最早发生取地址错误例外的地方是 0x8490977。

跟踪这条例外处理到达 WB 级，可以发现，错误的 PC 值 0x84790977 确实被传递到了 WB 级。但是下一个时钟周期时，理应被写入 0x84790977 的 epc 寄存器却被写入了 0x84790973：

仔细分析 cp0_epc 的生成信号可以发现，发生这种情况的原因是 bd 信号被拉高了。导致在写入 epc 寄存器时，在实际的错误 PC 的基础上减去了 4。观察这条取指地址错误指令的前一条指令：

0xbfc5bef8 是一条 bne 指令，所以下一条指令被标记为处于延迟槽中，导致最终写回 epc 的值差了 4.

（3）错误原因

对于延迟槽中指令发生取指地址异常例外的情况处理不当，直接按照一般方法处理的话，最终 epc 寄存器中存有的数据是非法 PC 减 4。

（4）修正效果

在对 bd 信号进行赋值的同时检查取指阶段是否发生了 PC 地址非法例外，如果发生了例外，即使该指令处于延迟槽中，也不要把 bd 信号拉高。

（5）归纳总结（可选）

发生此次错误的一个原因在于对于某些特殊情况处理不当。一般来说，对 bd 信号的赋值就按照：在延迟槽中赋值 1，否则赋值 0 就好了。但是如果例外处理程序在一些特殊情况下需要收集错误 PC 的值，那么还是需要根据特殊情况对 bd 赋值。

3、错误 3：寄存器写回值错误

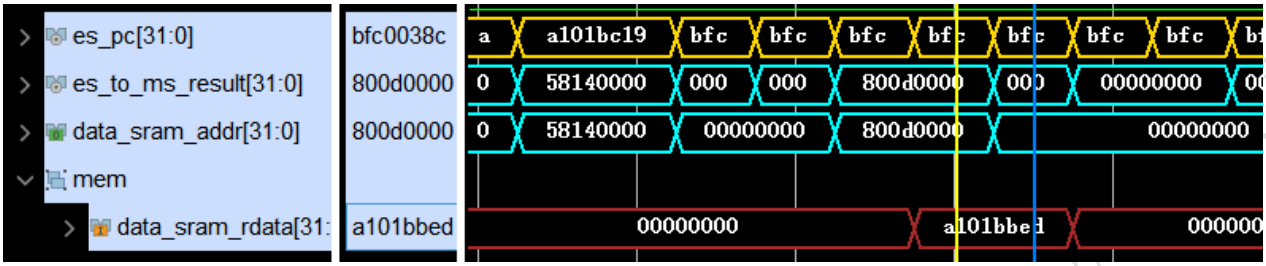
（1）错误现象

在取指地址例外的测试样例中，某处寄存器写回错误，如下图所示：

```
[1726157 ns] Error!!!
    reference: PC = 0xbfc0038c, wb_rf_wnum = 0x1b, wb_rf_wdata = 0x00000006
    mycpu    : PC = 0xbfc0038c, wb_rf_wnum = 0x1b, wb_rf_wdata = 0xa101bbed
------------------------------------------------------------
```

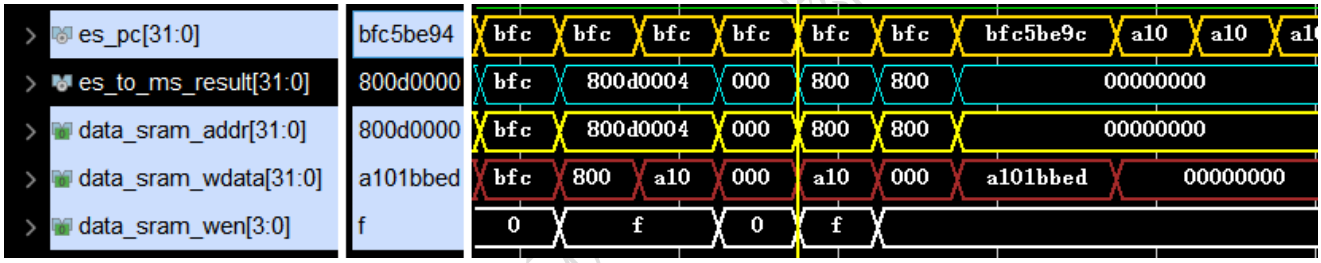
（2）分析定位过程

查询 test.s 文件可知，0xbfc0038c 位置是一条 lw 指令，写回寄存器的值错误，可能是由于最近的一次 sw 指令到对应位置时的写回数据错误。



从波形图可以看出，在 0xbfc0038c 这条指令的位置执行了 lw 指令，计算出了访存地址为 0x800d0000，在下一个时钟周期接收到了来自 sram 的数据 0xa101bbed。

追踪到最近的一次写该内存地址的指令，如下图所示：



此处写内存的指令 PC 为 0xbfc5be94，分析此处指令附近的指令流：

```
bfc5be8c:   ad140004    sw  s4,4(t0)
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n79_ft_adel_e
bfc5be90:   42000018    c0  0x18
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n79_ft_adel_e
bfc5be94:   ad140000    sw  s4,0(t0)
```

这条指令前一条指令恰好为 eret 指令，eret 指令需要到 WB 级才会刷新流水线，当 sw 到达 EXE 时，eret 才到达 MEM 级，因此没有刷新流水线导致发生了写内存错误。

（3）错误原因

对于 eret 后面的需要写内存，进行除法操作，以及写 hi，lo 寄存器等操作的处理不当。使得本来应该在 eret 之后不被执行的指令产生了执行效果。

（4）修正效果

指令在 EXE 级需要检测后面几个流水级是否有 eret 指令，如果有，则不要发出写内存信号，除法操作以及写

hi，lo 寄存器的信号。

（5）归纳总结（可选）

在做实验 8 时没有发现这个问题，是因为在 test.s 中，每条 eret 指令后面都手动加有一条 nop 指令，这样即使有上述需要修改寄存器或内存状态的指令到达了 EXE 级，eret 指令恰好到达了 WB 级，此时发出刷新流水线信号，使得那些对内存和寄存器状态进行修改的指令都无效了。但是在这次实验中，并没有增加 nop 指令作为缓冲，有发生上述问题的风险。

4、错误 4：寄存器写回值错误

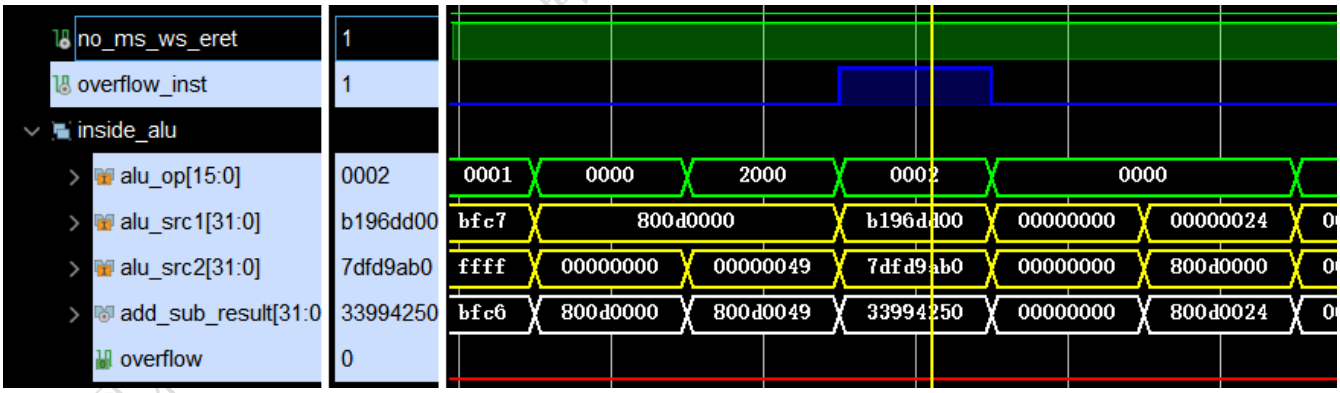
（1）错误现象

在溢出例外的测试样例中，某处寄存器写回错误，如下图所示：

```
[1685987 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000024
    mycpu    : PC = 0xbfc6957c, wb_rf_wnum = 0x02, wb_rf_wdata = 0x33994250
--------------------------------------------------------------
```

（2）分析定位过程

从对比的两个 PC 可以看出，正确的指令 PC 应该是 0xbfc00380，进入例外处理。而 myCPU 的 PC 更像是沿着以前的指令流继续执行，因此可能是在判断溢出指令时出现了问题。根据 test.s 可以看出，0xbfc6957c 位置是一条 sub 指令。因此很有可能触发溢出例外。观察该指令在 EXE 级中 ALU 的执行情况，如下图所示：



overflow_inst 表示该指令确实是一条会触发溢出例外的指令。但是观察运算结果可以看出，这是一个负数减去一个正数的运算，结果确实一个正数，因此发生了溢出。需要对 overflow 位设置为 1。代码中对 overflow 的设置如下：

```
assign overflow   = (op_add & ~(alu_src1[31]) & ~(alu_src2[31]) &  add_sub_result[31])
                  | (op_add &  (alu_src1[31]) &  (alu_src2[31]) & ~add_sub_result[31])
                  | (op_sub & ~(alu_src1[31]) &  (alu_src2[31]) &  add_sub_result[31]);
```

可以看出设计中没有采用 33 位数据判断溢出的方法，容易造成溢出情况考虑不完善的问题，这里就省略了负数减去正数变为正数的情形，从而导致了错误。

（3）错误原因

判断溢出时省略了负数减正数等于正数这种溢出的情况，从而导致了 overflow 位设置异常。

（4）修正效果

在溢出例外异常中添加此判断，添加后能顺利通过溢出例外的测试。

5、错误 5：中断信号赋值错误

（1）错误现象

在将软件中断赋值信号写好接到 wb 级时，wb 级信号一直为 0

（2）分析定位过程

```
assign clock_interrupt = (count_eq_compare | (cp0_cause_ip & cp0_status_im != 8'b0) )& ~cp0_status_exl & cp0_status_ie;
```

在将所有信号都拖出观察波形后发现信号一直为 0 但输入是正确的，最后拆分该信号后发现，对于按位与和不等于两种运算的优先级判断错误，应该先与后进行判断，在按位与两侧加入括号号信号正常。

（3）错误原因

对于按位与和不等于两种运算的优先级判断错误。

（4）修正效果

在按位与两侧加入括号号信号正常。