实验报告格式说明：

1. 标题层次建议不超过四级，从第一级开始标号格式依次采用：一、二、三；（一）、（二）、（三）；1、2、3；（1）、（2）、（3）。
   1. 一级标题用小三号字，中文为黑体，英文为Times New Roman，单倍行距，段前段后各0.5行。
   2. 二级标题用四号字，中文为黑体，英文为Times New Roman，单倍行距，段前段后各0.25行。
   3. 三级标题用小四号字，中文为黑体，英文为Times New Roman，单倍行距。
   4. 四级标题用五号字，中文为黑体，英文为Times New Roman，单倍行距。
2. 正文用五号字，中文为宋体，英文为Times New Roman，1.5倍行距。
3. 所有图统一顺序标号，图标题紧挨在图的下方，居中，用小五号字，中文为宋体，英文为Times New Roman，单倍行距，段后0.5行。
4. 所有表统一顺序标号，图标题放在表的上方，居中，用小五号字，中文为宋体，英文为Times New Roman，单倍行距，段前0.5行。
5. **报告形成后删除本模板中所有红色文字！**

**Lab 6报告**

学号

姓名

箱子号 5

一、实验任务（10%）

该实验要理解异常和中断的软硬件协调处理机制。理解精确异常的概念和处理方法，掌握在流水线CPU中添加异常和中断支持的方法，在已完成的CPU的基础上添加系统调用异常和其他异常与终端支持。

实践12需要增加 csr 控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY 和 SAVE0~3，实现对 csr 的三个读写命令 csrrd、csrwr 和 csrxchg，实现中断返回指令ertn 和系统调用指令 syscall。

实验 13 要求在实验 12 的基础上增加对取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不

存在（INE）这几种异常处理的支持，增加对 2 个软中断、8 个硬件中断和定时器中断的支持；在 csr 中增加控制

状态寄存器 ECFG、BADV、TID、TCFG、TVAL 和 TICLR。

该实验的正确性同样由基于 trace 比对的调试辅助手段进行验证，通过将 myCPU 中的 WB 阶段的 PC 值与 golden\_trace 的 PC 值进行比对，如果不一致则立即报错并且停止仿真。在仿真正确后进行上板验证。

二、实验设计（40%）

（一）总体设计思路

CPU发生中断和异常时状态控制信息放置在csr寄存器中，因此新增一个csr寄存器，包含了所有对csr寄存器进行读写的控制电路，并且例化在WB级中。因此当发生异常或中断后，会将信息附着在指令上沿流水线一路携带下去，直至写回级才真正报出异常，此时才会根据携带的信息更新控制状态寄存器。

csrrd、csrwr、csrschg指令用于软件访问CSR，需要使用到rd寄存器，因此产生数据相关时要阻塞.

对于syscall和ertn指令，处理思路类似，在写回级报出异常后，需要清空当前所有流水级的状态，即根据wb\_ex和ertn\_flush前递，将每级valid信号置0并将nextPC设置为异常入口地址。同时，考虑到对数据RAM的写入在mem阶段就已经完成，因此要将mem和wb阶段的异常判断信号传至exe级，使数据RAM的写使能置0。

（二）重要模块1设计：csr模块

1. 工作原理

在写回级处理异常，将相关信号传入CSR模块。若为csr读写指令，则对csr\_num对于的寄存器进行操作。若异常，则将异常码和出错的地址写入对应寄存器，并在异常处理完成后将异常处理程序的入口传给pre-IF级。

1. 接口定义

表格 1：csr模块接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| csr\_value | OUT | 32 | csr寄存器读数据 |
| ex\_entry | OUT | 32 | 例外跳转入口地址 |
| ertn\_entry | OUT | 32 | 例外处理返回指令 |
| ertn\_flush | IN | 1 | ertn指令信号 |
| wb\_ex | IN | 1 | 例外发生信号 |
| ws2csr\_bus | IN | 200 | 包括csr\_re（读使能）, csr\_we（写使能）, csr\_num（写地址）, csr\_wmask（写掩码）, csr\_wvalue（写数据）, wb\_pc（例外发生时的pc）, wb\_ecode, wb\_esubcode, ipi\_int\_in（核间中断输入）, coreid\_in（核编号，TID初值）, hw\_int\_in（硬件中断输入）, wb\_vaddr信号 |
| has\_int | OUT | 1 | 发生中断信号 |

1. 功能描述

csr 模块包含控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3、ECFG、BADV、TID、TCFG、TVAL、TICLR。每个模块的每个域分别用一个寄存器实现，在读出时再拼接为完整的寄存器，通过多路选择器赋值给 csr\_rvalue。每个域在 wmask 中对应的 bit，以及每个寄存器的 csr\_num，都用宏在 csr.h 中定义。

CRMD（当前模式信息），用于决定处理器核当前所处的特权等级、全局中断使能和地址翻译模式。其中对于PLV域，当触发例外时，该域的值置为 0，以确保陷入后处于最高特权等级。对于IE域（当前全局中断使能）。当触发例外时，该域的值置为 0，以确保陷入后屏蔽中断。例外处理程序决定重新开启中断响应时，需显式地将该位置 1。 当执行 ERTN 指令从例外处理程序返回时，需要将 PRMD 寄存器中的 PPLV 和 PIE 域分别写入 PLV 和 IE 域。

PRMD（例外前模式信息），当触发例外时，硬件会将此时处理器核的特权等级和全局中断使能位保存至例外前模式信息寄存器中，用于例外返回时恢复处理器核的现场。触发例外时，PRMD 寄存器中的 PPLV 和 PIE 域被赋值成 CRMD 寄存器的PLV 和 IE 域中的值。

ESTAT（例外状态），该寄存器记录例外的状态信息，包括所触发例外的一二级编码，以及各中断的状态。按照讲义定义对IS域、Ecode域和EsubCode域进行赋值即可。

ERA（例外返回地址），当触发例外时，触发例外的指令的 PC 将被记录在该寄存器。

EEENTRY（例外入口地址），该寄存器用于配置除 TLB 重填例外之外的例外和中断的入口地址，只有VA域会被CSR指令更新。

SAVE0~3（数据保存），提供给特权软件临时存放数据。

ECFG（例外控制），ECFG 的第 0~9 位和第 11~12 位均为局部中断使能位 LIE，每位控制一个中断源，中断源与 ESTAT 的对应位一一对应，其余位为保留位。对于 LIE 域，其读写由 CSR 指令实现，因此只需对此添加相关支持。其余保留位不允许写入，读出的值为 0，因此直接用 0 填充即可。

BADV（出错虚地址），存储触发地址错误相关例外时出错的虚地址。本次实验只涉及到ADEF和ALE。ADEF将wb\_pc写入BADV，ALE把访存地址传到wb级并写入。

TID（定时器编号），软件配置定时器的接口，从低到高依次为定时器使能位、循环模式控制位、初始值、只读位。需要支持 CSR 指令的读写，并用对应位控制定时器 TVAL。

TVAL（定时器值），TVAL 存储定时器的值，初值为TCFG中的 InitVal 域，在 TCFG 的 En 域为1时递减。 减到 0 后，如果 TCFG 的 Periodic 域为 1，则从初始值开始重新倒计时，否则保持不变。

TICLR（定时中断清除），从该寄存器中读出的值始终为 0，通过对该寄存器最低位写 1 的动作来清除时钟中 断标记。因此只需要用一个 1bit 信号 csr\_ticlr\_clr 来实现该寄存器，并始终赋值为 0，如果要读则在高位拼接上 31bit 的 0。至于清除时钟中断，只需要捕捉对寄存器写 1 的动作，不需要真的写入。

（三）重要模块2设计：IF模块

1、工作原理

1. 功能描述

内部具体是怎么设计的，描述要简洁明了，直中要害。

（四）重要模块3设计：XXX模块

进行重要设计的具体描述。

1. 工作原理

为什么要这么设计，其基本工作机制是否合理

1. 接口定义

每部分的接口是什么。如果写报告的时间充裕，可以以表格形式列出；如果时间仓促，该节可以一笔带过。

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| …… | IN |  |  |
| …… | OUT |  |  |

1. 功能描述

内部具体是怎么设计的，描述要简洁明了，直中要害。

（二）重要模块4设计：XXX模块

进行重要设计的具体描述。

1. 工作原理

为什么要这么设计，其基本工作机制是否合理

1. 接口定义

每部分的接口是什么。如果写报告的时间充裕，可以以表格形式列出；如果时间仓促，该节可以一笔带过。

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| …… | IN |  |  |
| …… | OUT |  |  |

1. 功能描述

内部具体是怎么设计的，描述要简洁明了，直中要害。

三、实验过程（50%）

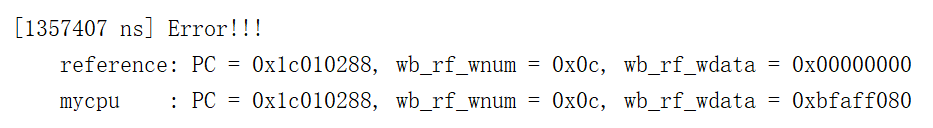
（一）实验流水账

记录哪一天，几点到几点，做了什么事，结果如何。事情不要展开来写。

（二）错误记录

1、错误1：第二个操作数从rk和rd里选择错误

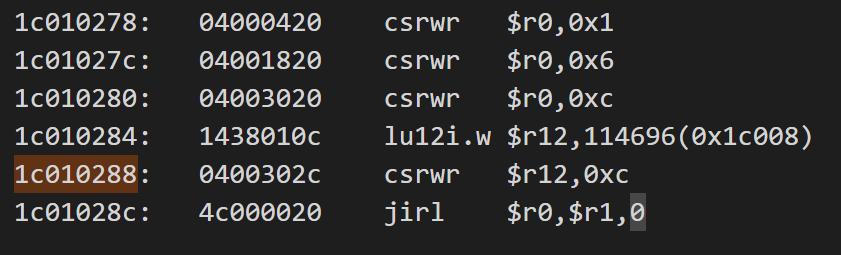
（1）错误现象



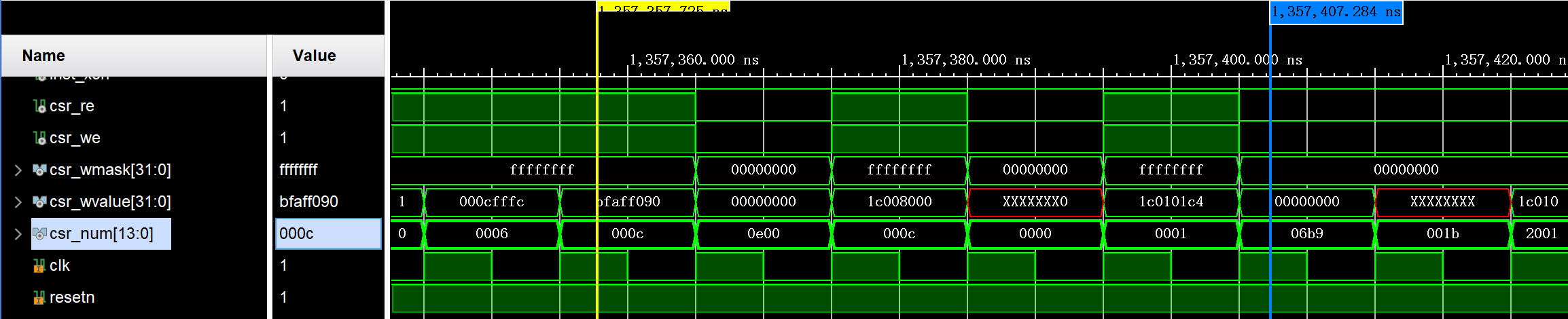
写回寄存器的数据错误。

（2）分析定位过程

经查找，出错的指令是SCRWR，该指令是将指定的CSR寄存器的旧值写回rd寄存器。在排除了指令译码出错的可能性后，就只可能是在上一个SCR写指令时，将错误的值写入了这个CSR寄存器（或者将正确的值写入了错误的其他的CSR寄存器，导致这个CSR寄存器里的值没有更新从而出错）。



查看反汇编代码，发现1c010288的上一个CSR写指令是1c010280。从汇编指令可以看出来，它的CSR寄存器字段与1c010288相同，都是0xc，也就是EENTRY寄存器。



从波形中也可以印证这一点，黄色光标是上一个csrwr的ID，往后两拍是这个csrwr的ID，二者的CSR地址一样，都是0xc。

由于之前出现过类似的情况，因此很容易就想到，CSR读写指令的第二个操作数是rd中取来的，但没有更新src\_reg\_is\_rd。于是在src\_reg\_is\_rd中添加了csr\_re的情况，也就是CSR读写指令。



（3）错误原因

CSR读写指令的第二个操作数是rd中取来的，但没有在src\_reg\_is\_rd添加CSR读写指令的情况。导致在出错指令的上一个指令执行时，错误地把rk而不是rd寄存器里的数据写入了EENTRY寄存器。该指令执行时，就从EENTRY寄存器里读回了错误的值。

（4）修正效果

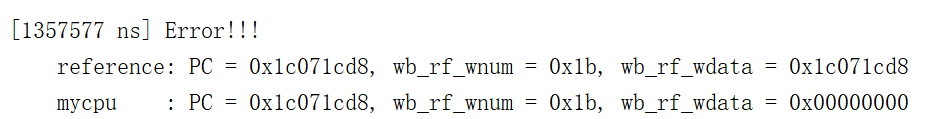
在src\_reg\_is\_rd中添加CSR读写指令的情况后，不再出错。

（5）归纳总结（可选）

在新增指令时，除了新增功能之外，也一定要考虑到是否需要修改之前的逻辑。

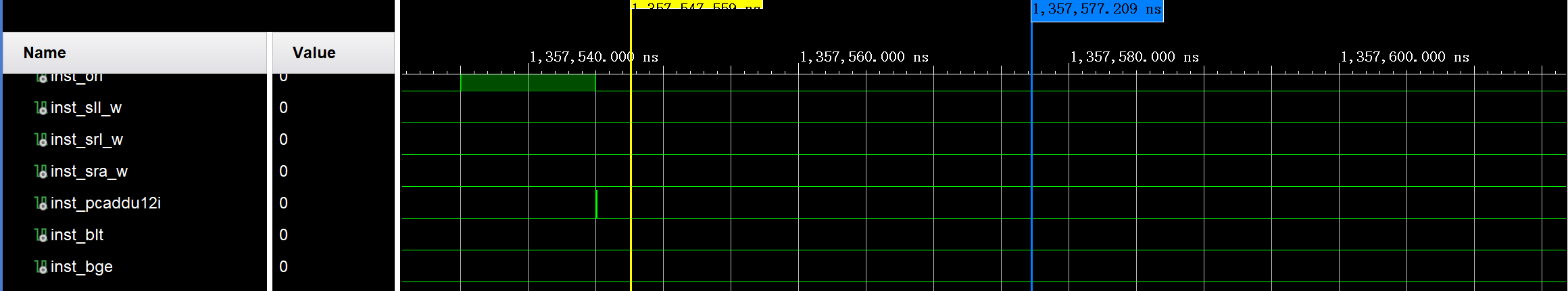
2、错误2：历史遗留错误之LIU12I.W指令

（1）错误现象



写回寄存器的数据错误。

（2）分析定位过程



查看反汇编代码，得知出错的指令是LIU12.W，但在波形上回到第二拍，却发现没有任何一个指令拉高，inst\_lu12i\_w也没有拉高。翻看inst\_lu12i\_w的代码，发现它使用的是inst而不是inst\_reg，也就是错误地使用了此时出于IF阶段的指令去译码。将inst改成inst\_reg修改如下。



反思一下之前为什么没出错，猜测是因为测试文件里所有的当拍的inst[25]恰好和inst\_reg[25]一样，导致这个错误一直隐藏。

（3）错误原因

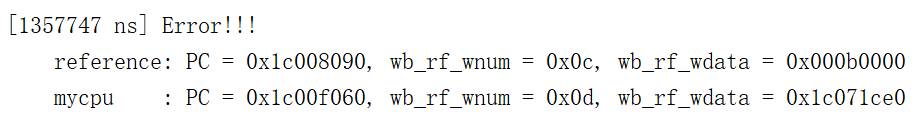
inst\_lu12i\_w使用的是inst而不是inst\_reg，也就是错误地使用了此时出于IF阶段的指令去译码。这导致这一拍没有任何指令执行，所有的控制信号拉低，因此与或逻辑的final\_result为全零。最后选择写回的数据时使用的是三目运算符，同样由于控制信号拉低，最后选中了全零的final\_result\_reg，导致写回的数据全零。

（4）修正效果

将inst改成inst\_reg后，不再出错。

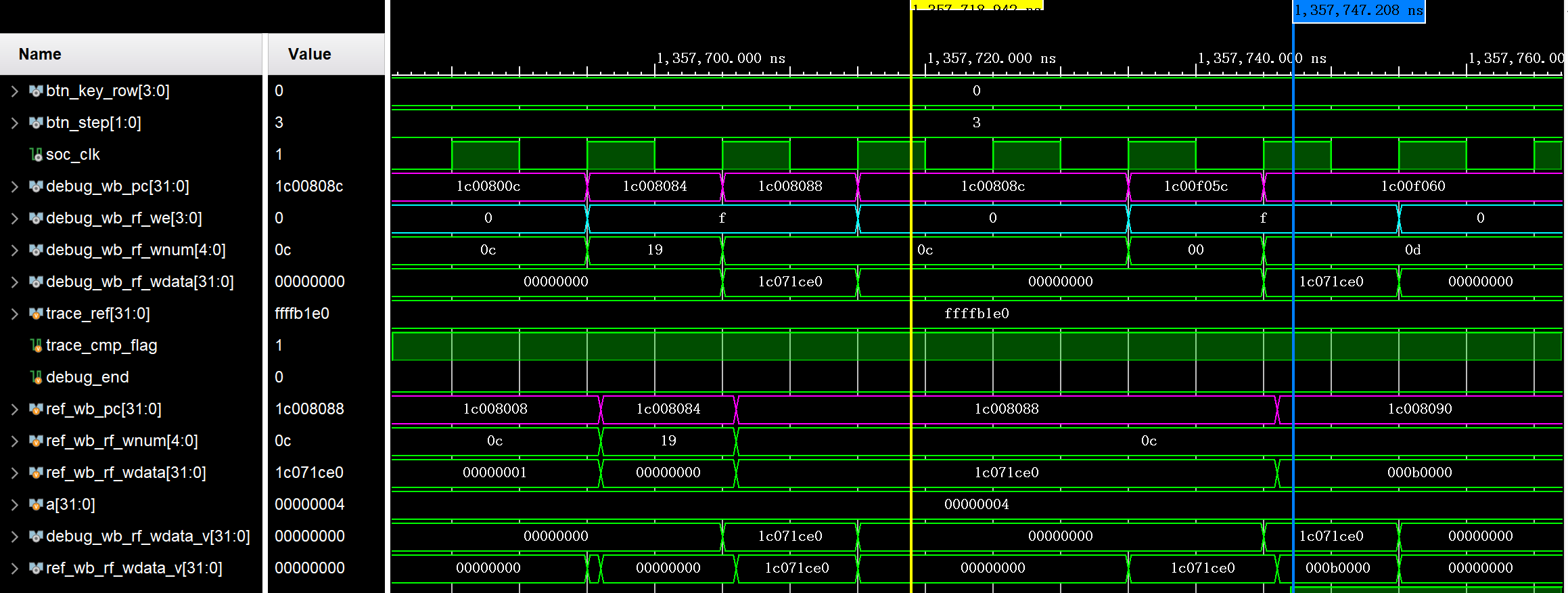
3、错误3：冲突时的阻塞逻辑考虑不全

（1）错误现象



PC错误，根据经验一般是跳转指令出错。

（2）分析定位过程

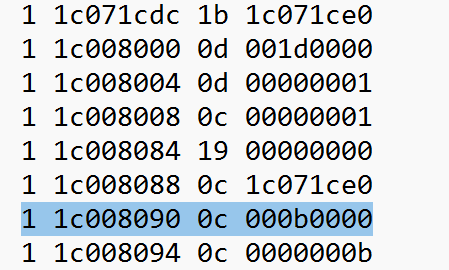


蓝色光标为报错的位置，报错原因是从PC开始就和金标准不相符。可以看到蓝色光标的上一拍，debug\_wb\_rf\_we为f，是拉高的，也就是说这一拍会和金标准做对比。这一拍的PC是1c00f050，金标准这一拍的PC是1c008088，发现在这一拍就已经出错了，不知道出于什么原因没有报错。

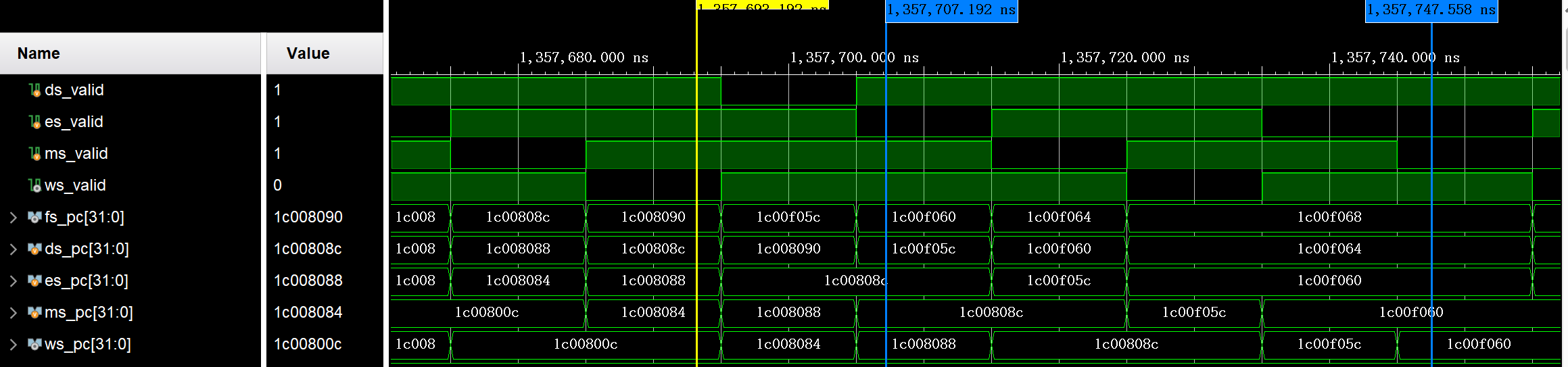
再往前追溯。

再上拍1c008088是CSRRD，紧跟在跳转指令后面被取消了。

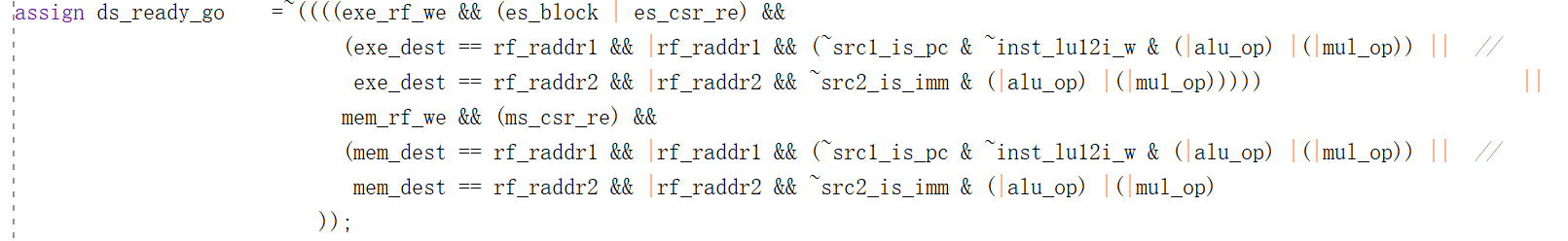
再上拍1c00808c是BNE，看来是跳转错了。这里从波形来看，br\_taken拉高，实际执行了跳转，且经计算跳转地址看样子是对的。那么有可能是这里实际上不该执行跳转，判断条件错误，但通过检查rj和rd寄存器数值的大小，也发现没有错误。



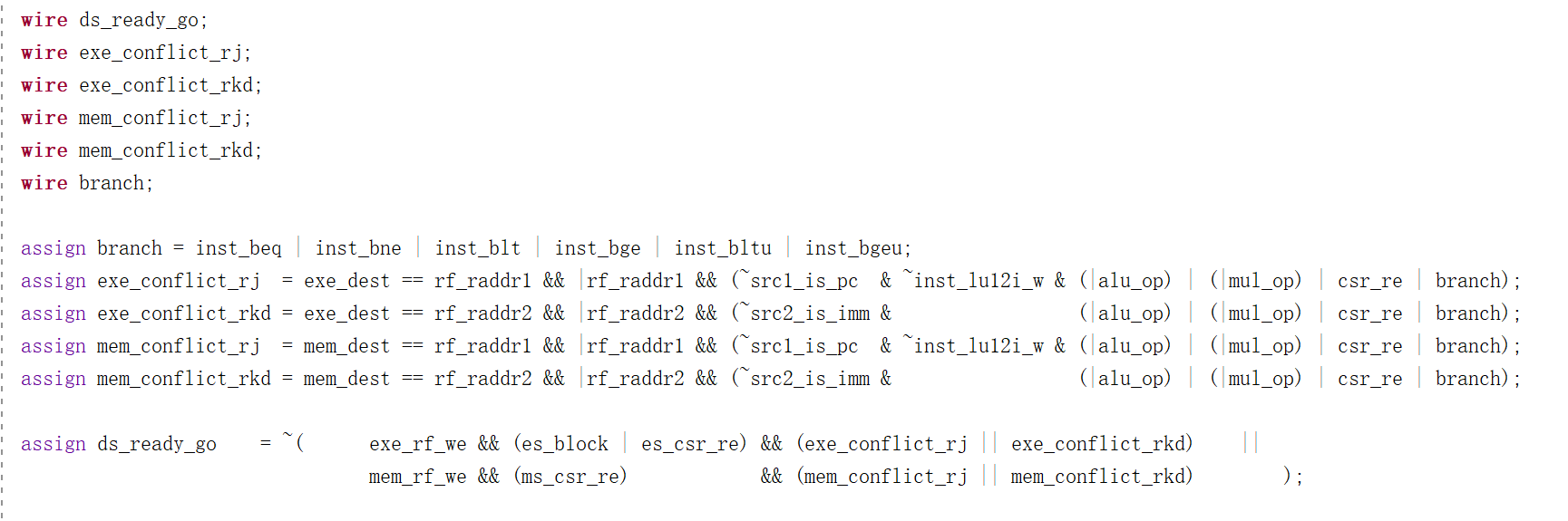
查看金标准文件，按照金标准的意思不应该跳转，那么一定是判断条件错误。



再查看波形，黄色光标时ID的BNE跳转指令和EXE的CSRRD指令冲突了，理论上是要阻塞BNE，但没阻塞，而是直接前递了。



查看代码中的阻塞逻辑，发现是由于与上了(|alu\_op)，因此没考虑到BNE等六条条件跳转指令。



于是修改代码，定义变量branch，表示那六条条件跳转指令，并将其加入阻塞逻辑。

同时还注意到，阻塞逻辑中没有考虑到CSR读写指令，于是将csr\_re一并加入阻塞逻辑。

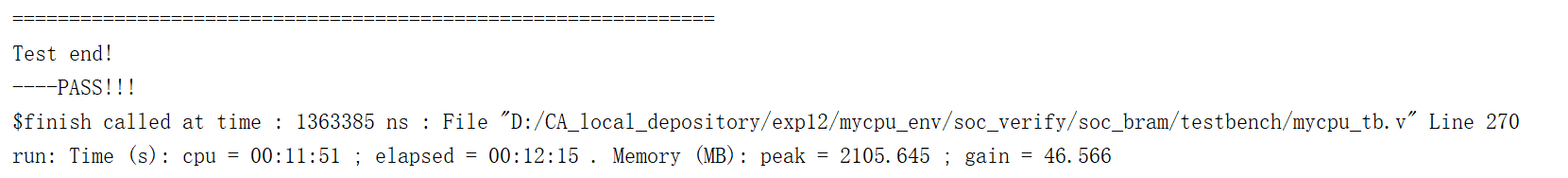
之前之所以没出错，猜测是因为之前的测试文件里全都没有ld指令后面紧跟branch的情况。现在新出现了CSR读写指令后面紧跟branch的情况，暴露了之前的问题。

（3）错误原因

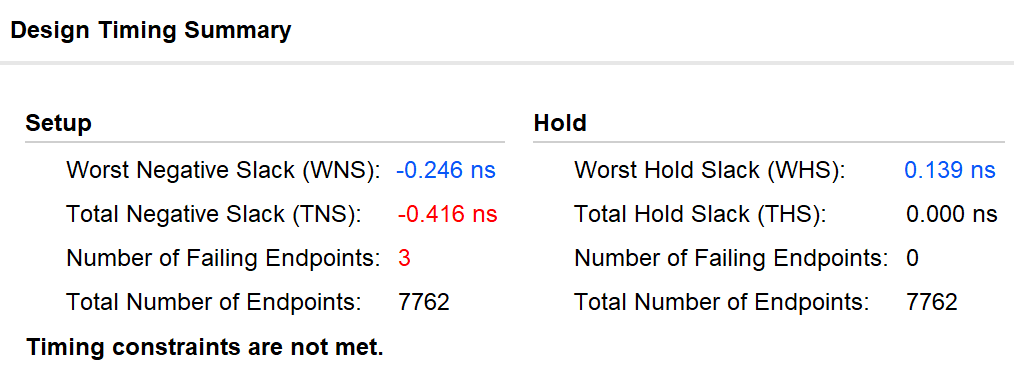
阻塞逻辑（ds\_ready\_go）中没考虑到BNE等六条条件跳转指令，导致BNE跳转指令和EXE的CSRRD指令冲突时，理论上是要阻塞BNE，但实际上没阻塞，而是直接前递了。

（4）修正效果

将branch加入阻塞逻辑之后，不再出错。并最终通过exp12的仿真。



下图是综合实现的时序结果。



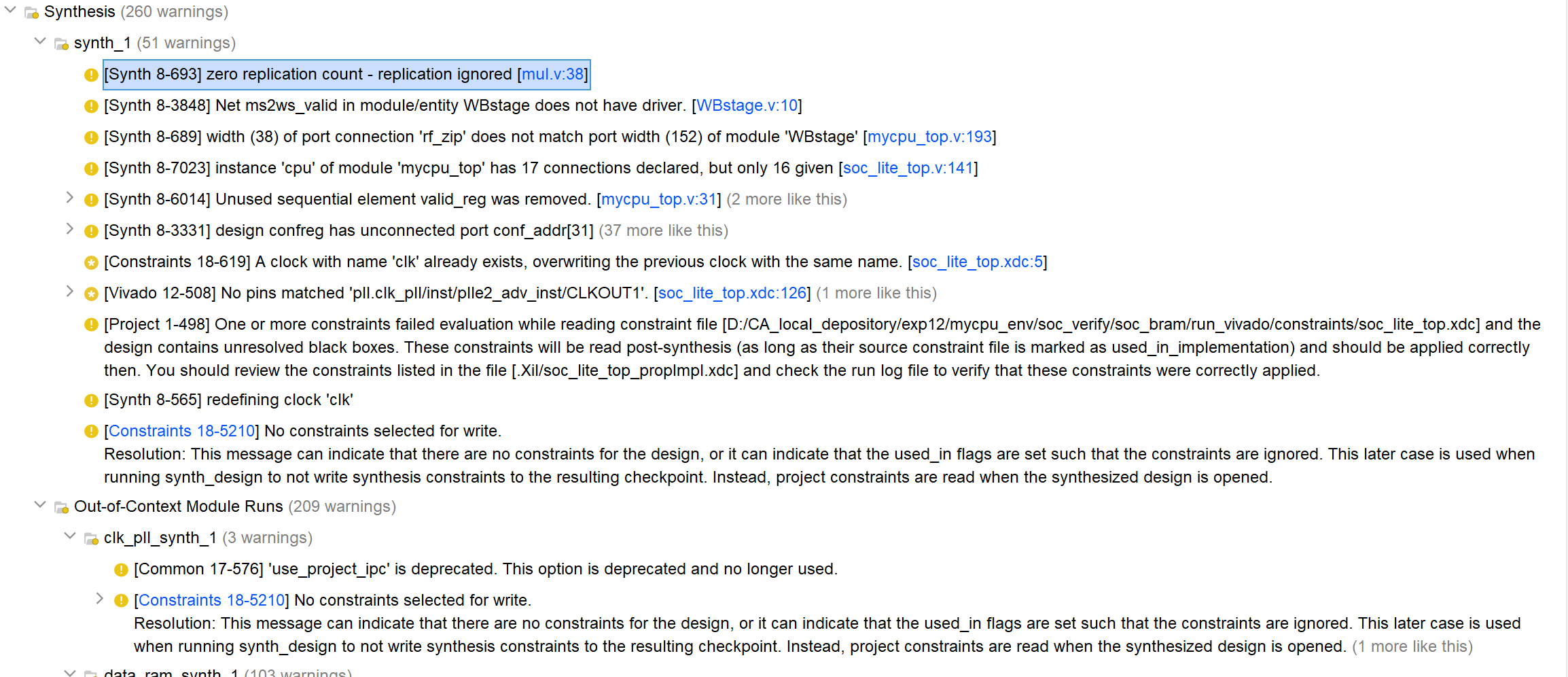
4、错误4：仿真通过，上板不过

（1）错误现象



红灯全亮。

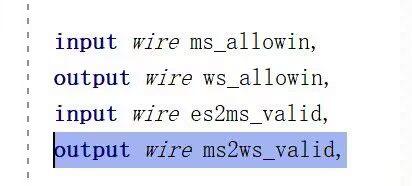
（2）分析定位过程



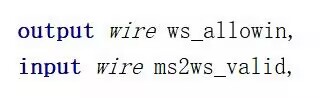
打开综合报告，逐一排查错误。再次生成比特流后，上板通过。虽然修改了若干个错误，但怀疑导致上板失败的是因为其中的一个无驱动错误。



报错显示WB模块里的ms2ws\_valid信号无驱动。



发现是因为误将input 的信号ms2ws\_valid写成了output。同时还有一些其他错误，将这一段修改如下。



（3）错误原因

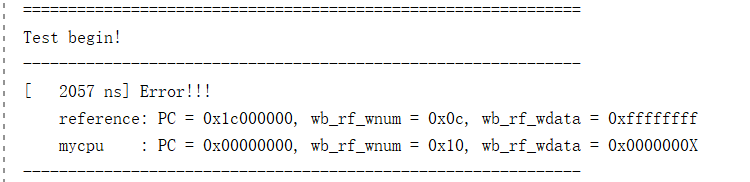
猜测是因为误将WB模块中的input 的信号ms2ws\_valid写成了output。

（4）修正效果

修改过后，重新生成比特流，上板通过。

5、错误5：赋值位数错误

（1）错误现象

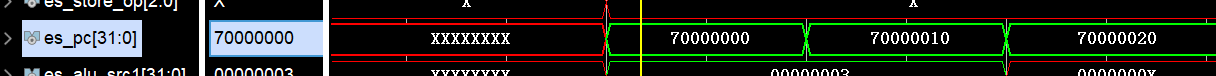


（2）分析定位过程

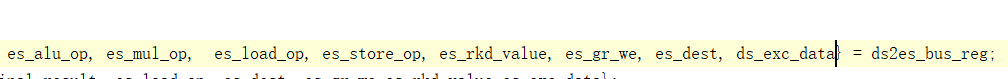
Nextpc出错，检查第一条指令的数据通路，发现是ds2es\_bus 的长度出现错误，导致正常数据未被传输过去。



修改后仍然错误，继续往后查看EXE阶段，es\_pc出现奇怪的值，说明赋值出现错误



（3）错误原因



检查后发现EXE阶段漏给es\_time\_op赋值,遗漏了两位。

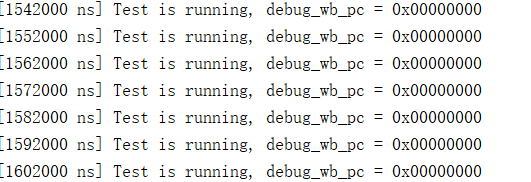
（4）修正效果

assign {es\_pc,es\_res\_from\_mul, es\_alu\_src1, es\_alu\_src2, es\_alu\_op, es\_mul\_op,  es\_load\_op, es\_store\_op, es\_rkd\_value, es\_gr\_we, es\_dest, ds\_exc\_data,es\_time\_op} = ds2es\_bus\_reg;

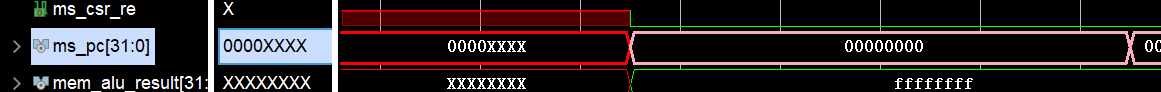
修正后该时间仿真测试通过。

6、错误6：寄存器间数值传递位宽错误

（1）错误现象



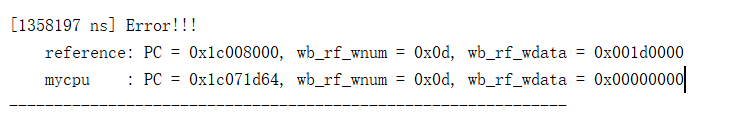
（2）分析定位过程



同样是bus赋值出错，ES2MS\_BUS\_LEN应为208位，在原来的基础上补充例外信号多余的位数。修改后，仿真开始运行。

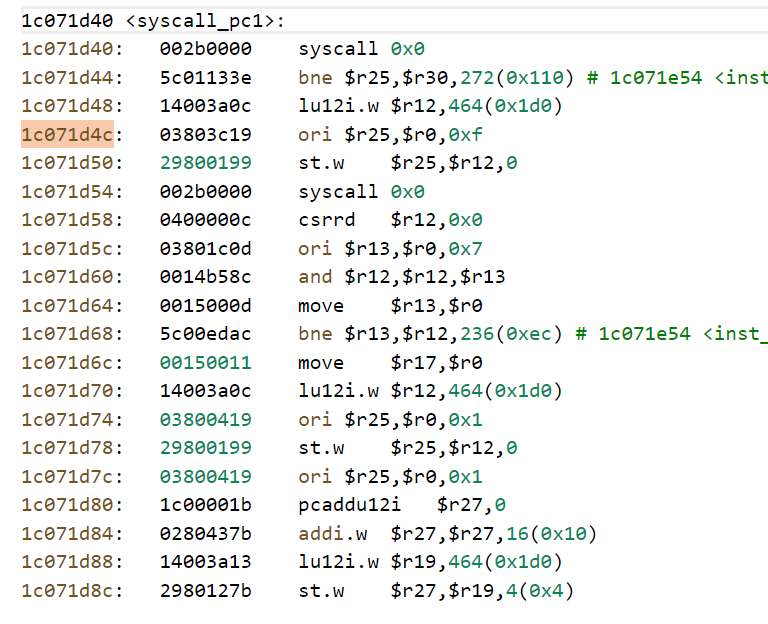
7、错误7：ready\_go信号错误

（1）错误现象

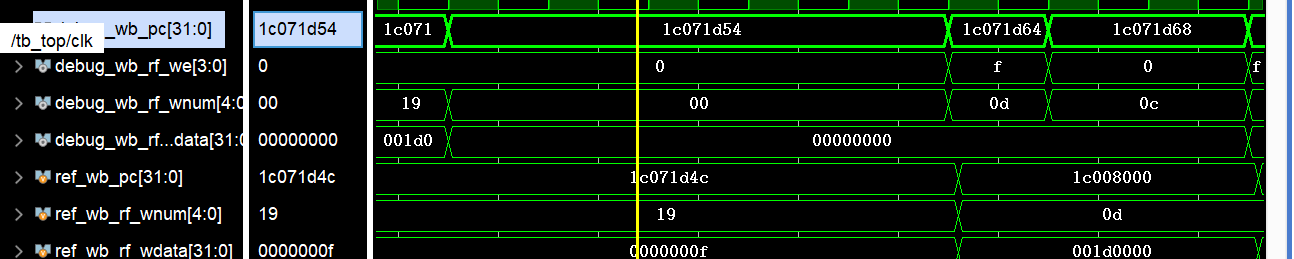


（2）分析定位过程

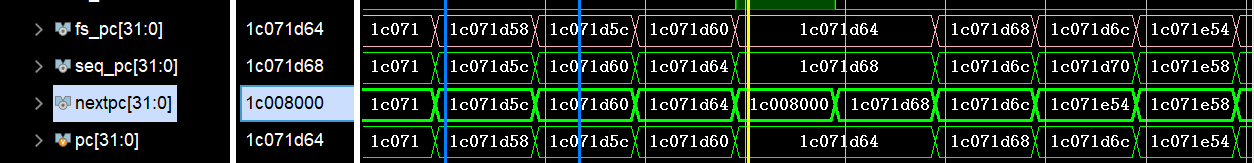
这一部分是sys\_call测试点



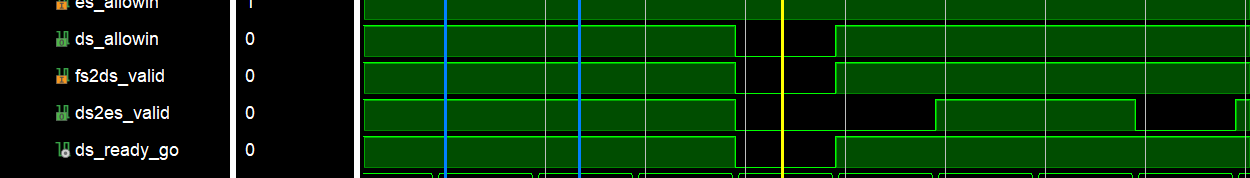
报错点之前执行的是一条syscall指令，猜测syscall出现问题



检查例外信息的数据通路发现到WB阶段都没有问题，查看该拍对应的IF模块中的例外入口地址，发现nextpc对应正常的1c008000 ，但fs\_pc却是seq\_pc的值，说明此时赋值逻辑存在问题。

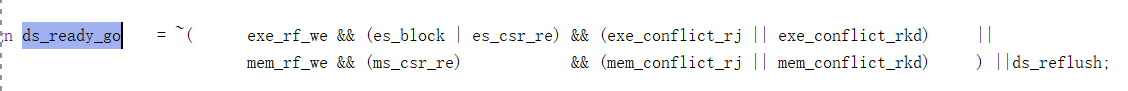


检查后发现，fs\_allowin未拉高，导致next\_pc未赋值进去，



ID模块中的ds\_ready\_go信号此时未拉高，而它理应拉高，因为此时是例外产生信号，应当把bus中的数据刷掉，但这里并未刷掉。

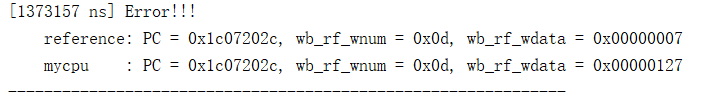
（4）修正效果

****

应当在ds\_ready\_go信号加上ds\_reflush信号。修改后该位置仿真不报错

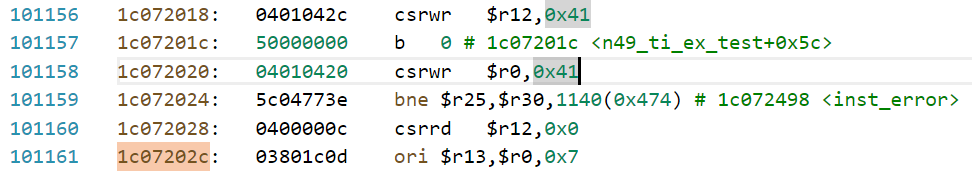
8、错误8：寄存器0地址不需要赋值

（1）错误现象

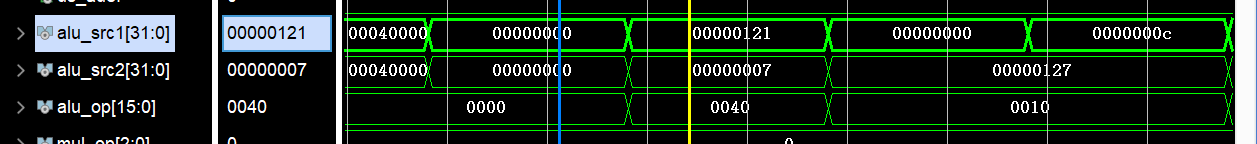


（2）分析定位过程

查看汇编文件，发现这是一条ori指令，

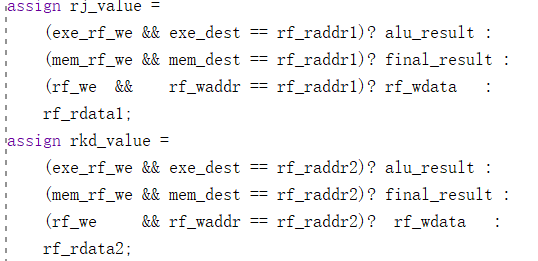


查看对应的译码阶段，发现是操作数选取有错。继续追踪发现是rj\_value的赋值是WB阶段的前递结果，而事实上并不需要前递。虽然出现写后读，但是这是一条写地址为0的指令，所以并不需要前递。



（3）错误原因

零地址不需要前递。



（4）修正效果

assign rj\_value =

    (exe\_rf\_we && exe\_dest == rf\_raddr1 && |rf\_raddr1)? alu\_result :

    (mem\_rf\_we && mem\_dest == rf\_raddr1 && |rf\_raddr1)? final\_result :

    (rf\_we     && rf\_waddr == rf\_raddr1 && |rf\_raddr1)? rf\_wdata   :

    rf\_rdata1;

assign rkd\_value =

    (exe\_rf\_we && exe\_dest == rf\_raddr2 && |rf\_raddr2)? alu\_result :

    (mem\_rf\_we && mem\_dest == rf\_raddr2 && |rf\_raddr2)? final\_result :

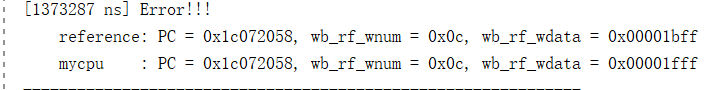
    (rf\_we     && rf\_waddr == rf\_raddr2 && |rf\_raddr2)?  rf\_wdata   :

rf\_rdata2;

修改后该时间点仿真测试通过。

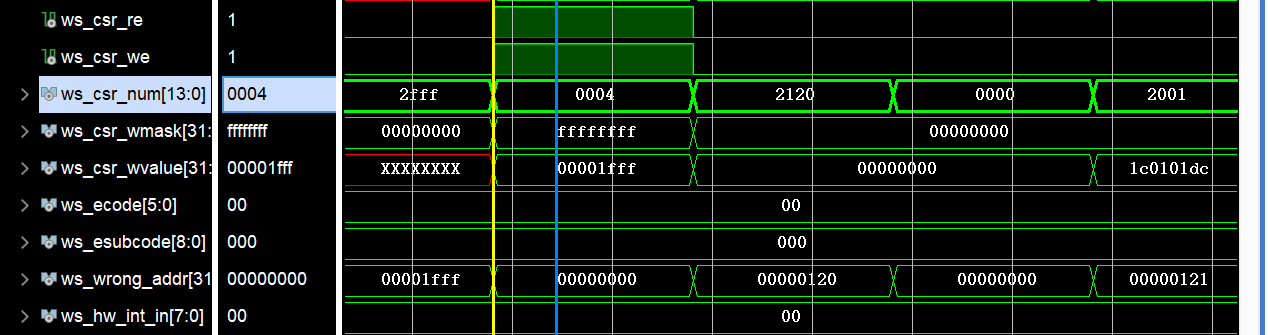
9、错误9：ECFG寄存器不允许修改值被修改

（1）错误现象

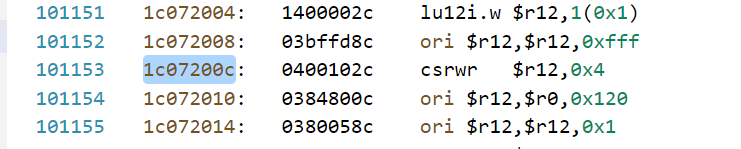


（2）分析定位过程

这是一条csrwr指令，其读出的旧值有错，检查其数据通路。发现之前的一条指令将其值修改为了0x00001fff，但实际上应该是0x00001bff



这条指令的pc是1c07200c，同样是一条csrwr指令，这说明从r12寄存器的旧值存在问题，但是写入的值肯定会通过debug信号报错，未报错表明不是那条指令执行出错。因此猜测，那条csrwr指令没有正确将r12指令的旧值写入04CSR寄存器。但检查后发现，该指令正确地将值写入了ECFG寄存器，那说明ECFG之后理应有某条指令更新其值为0x00001bff.



但是仔细检查了之后发现也没有在这之后发现任何一条写ECFG寄存器的指令，debug一度陷入了僵局，完全不知道为什么会这样。按理说之前写进去一个值，后面读出来的旧值肯定就应该是该值，为什么会这样呢？？？

直到我突然意识到1fff和1bff最大的不同在于第10位的区别。查看ECFG状态寄存器的说明，赫然写着软件不允许修改其值！！！案件就此侦破，



（4）修正效果

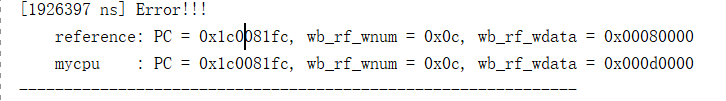
将csr模块中的CFG读出信号修改为：



修改后该时间点仿真测试通过。

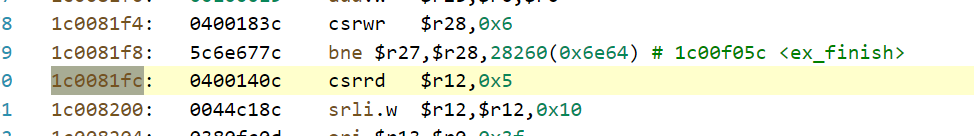
10、错误10：带宽错误

（1）错误现象

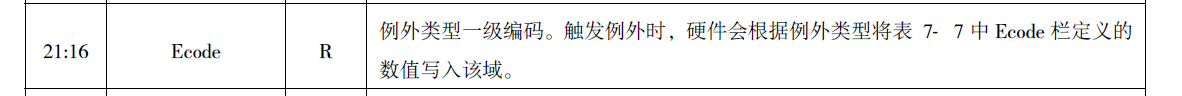


（2）分析定位过程

查看汇编文件，这是一条csrrd指令，其读出的值有错，说明可能之前写错或者和上个bug一样考虑到了特殊位.



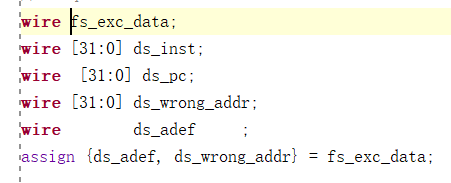
不同的位是ESTAT寄存器的Ecode域，参考值位8，但实现的CPU是d，分别对应ADE错误和INE错误。



找到CPU中上次给ECODE写入d的情况，发现对应的指令是一个取值错误pc，理应是ine而不是ade。

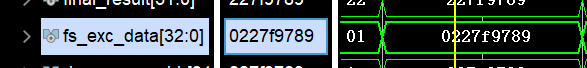


这是在EXE产生新异常的时候未考虑这可能是错误pc导致的异常，因此ecode应为取指地址错误的情况而非访存地址错误的情况。

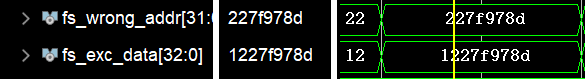


原因是源代码中，传给ID阶段的fs\_exc\_data位宽不对，而因为之前并没有取指地址错误的情况，所以并没有报错。但修改后继续仿真发现还是在同样的位置报错，检查后发现从IF传过来的例外信息与预期不符。

ID中：

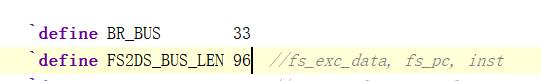


IF中：



立刻想到例外信息的位宽应该是97位，而不是96位，所以最高位被切掉出现错误。

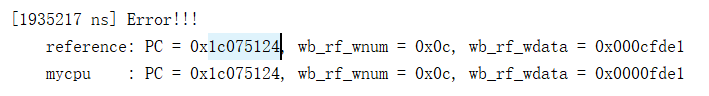
（3）修正效果



重新修改后进行仿真，该测试点通过。

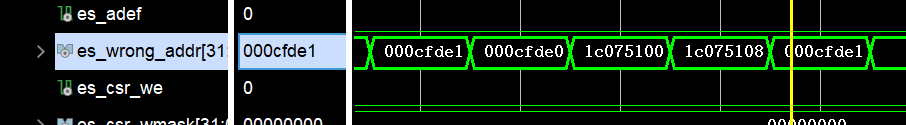
11、错误11：位宽错误

（1）错误现象

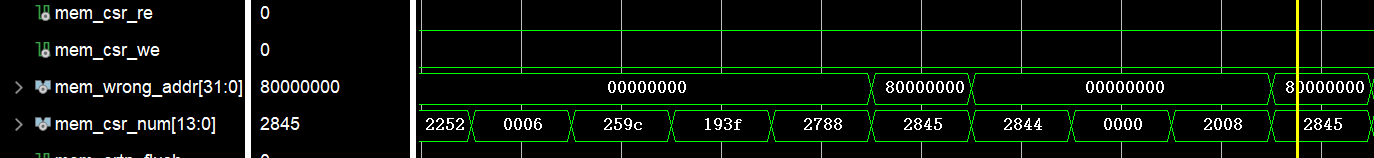


（2）分析定位过程

这同样是一条csrrd指令，读出BADV寄存器中的值，因此立刻联想到访存地址或取指地址出错产生的异常，查看上一次给BADV寄存器写入0x0000fde1的情况，看到其EXE阶段给出的地址正确，但最后写入的地址却是错误的



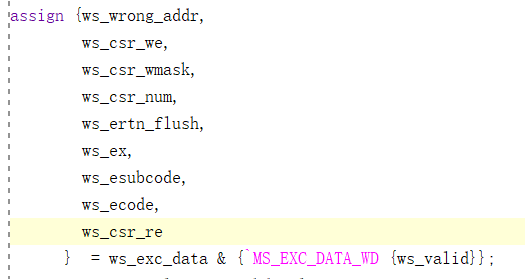
想破脑袋都想不明白，为什么mem\_wrong\_addr会变成0x80000000。肯定不会是位宽的问题，因为如果位宽错了，早应该在之前某个地方就报错了，我本来是这么想 的，但是因为在MEM的译码只是为了生成一些简单位的控制逻辑，所以这里不应该有影响才对，mem的异常信息是直接传给了WB阶段。而这里之所以会是0x80000000是因为未声明wmask信号，导致其隐式为1出现wrong\_addr变为奇怪的值。



但是这并不能解释为什么直接传给WB阶段的异常信息的译码会出现两位位的差别。

最终检查WB阶段的译码时发现了问题，之前该位置是一个魔数，我在写代码时未修改其值，其只有83位，但是加入新的内容后，当前应有97位，导致ws\_wrong\_addr的一部分被抹去，从而出现了错误。

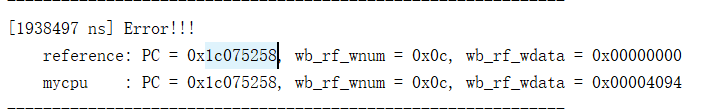
（3）修正效果



修改后该时间点测试通过。

12、错误12：错误简介命名

（1）错误现象



（2）分析定位过程

该位置是一条除法指令，写数据与预期不符

居然是因为当前指令是0操作数除以一个非0值，但是之前的除法器未考虑到被除数为0，除数非0的情况，最终导致照常恢复余数产生错误的结果。

（3）修正效果

对最后的result进行选取即可：



修改后，仿真通过。

四、实验总结（可选）

本次实验中比较难的部分在于捋清每个异常指令是如何运行的，比如ertn和syscall指令应该放在哪部分处理。决定放WB阶段后，哪些指令对应的数据会变成无效的以及如何置为无效。由于老师已经将大部分csr寄存器部分的代码给出，因此本次实验主要是对控制通路进行修改。