**LAB6报告**

学号： 2021K8009929010

姓名： 贾城昊

学号： 2021K8009929016

姓名： 李金明

学号： 2021K8009929007

姓名： 牛浩宇

箱子号： 13

1. 实验任务

Lab7的要求为CPU增加AXI总线接口，令CPU能够通过总线与系统中的内存、外设进行交互。主要分为3个阶段：

Exp14：将CPU原有的访问SRAM的接口改为类SRAM总线接口，增加握手信号、随机延迟。

Exp15：通过转接桥将类SRAM接口转接为AXI接口，并实现固定延迟的验证。

Exp16：随机改变种子，实现随机延迟的验证。

1. 实验设计
   1. 总体设计思路
      1. Exp 14

本实验将CPU访存的接口改为类SRAM总线接口，为此需要调整原有的访问inst\_sram和data\_sram的逻辑。

在原有的设计中，向RAM发送请求的下一拍就能拿到数据，或是完成写入；而现在，发起请求需要等待addr\_ok信号返回，表明请求被成功接收，之后又需要等待若干拍，data\_ok才会拉高，表明写入完成，或是返回读数据。为此，需要把发送请求与接收响应拆分到两个流水级中，并在流水级没有完成自己的任务时将其阻塞。

* + - 1. 指令sram

对于取指而言，pre-IF级负责发送请求，IF级负责接收返回的指令。pre-IF级会计算nextpc，并请求从nextpc取指。IF级负责从inst\_sram中接收读出的指令，由于读完不一定能马上流入ID级，因此需要一个寄存器if\_inst\_reg来保存读出的指令，另外还要一个if\_inst\_reg\_valid来记录if\_inst\_reg中的指令是否有效。在它们没有完成各自的任务时，通过将ready\_go置为0来阻塞对应的流水级。

在设计req的逻辑时，起初为了提高CPU的性能，我遵循讲义中的建议，在if\_allowin为1时才发送取指请求。这可以保证在发完请求后pre-IF级一定能流入IF级，至多也只会有一条需要取消的指令。取消指令是因为可能请求握手成功后，数据返回前发现指令无效，此时需要忽视后面返回的指令。而如果在数据返回后发现指令无效的逻辑，在前面的实验中则已经进行了处理。

类SRAM总线还会引入一个新的问题：如果ID级是转移指令，而它的前一条指令是load指令，又恰好存在数据相关，那么ID级由于延迟会无法通过前递获得数据。为此在ID级加入一个br\_stall信号并传给pre-IF级，使其在发生这种相关时将req拉低，不要发送取指请求，直到相关解除后才发送。

* + - 1. 数据sram

对数据sram的访问逻辑的修改与指令sram类似，也是将请求和接收分配到两个流水级中完成，EXE级负责发送请求，MEM级负责接收返回的信号。这里涉及到的指令只有load和store，并且也不存在指令sram中需要取消请求那样的情形，因此设计相对简单一些。与指令RAM的处理类似，EXE级也只在mem\_allowin为1时发请求，这样可以保证发完请求后可以直接进入MEM级，不需要考虑数据在EXE级就返回的情况。对于ready\_go信号，如果需要访存，则要等待请求握手成功后才能为1，否则可以直接沿用之前实验的逻辑，若为除法指令，等除法计算完后置为1，否则直接置1即可。MEM级也考虑这两种情况，在需要访存时等待data\_ok，否则直接流入下一级。由于在目前的设计中，WB级的ready\_go始终为1，这导致wb\_allowin也始终为1，因此MEM级拿到数据后一定能流入下一级，不需要像IF级一样用寄存器存放读出的数据。

* + 1. Exp 15

在本次实验中，需要在理解AXI总线协议和SRAM总线协议的基础上，设计一个类SRAM-AXI的转接桥，从而实现类SRAM接口和AXI接口的交互。在之前的实验中，我们的CPU提供的接口是类SRAM接口。经过本次实验，我们设计了转接桥AXI\_bridge并在mycpu\_top里进行例化，并将mycpu\_top对外提供的接口改为AXI接口。

当AXI\_bridge的**读请求通道**或**写请求写数据通道空闲**时，会接受来自CPU的请求信号（inst\_sram\_req或data\_inst\_req）和请求地址（inst\_sram\_addr或data\_sram\_addr），根据写信号（inst\_sram\_wr或data\_sram\_wr）来判断是读请求还是写请求。

若是读请求，AXI\_bridge会拉高arvalid，表示请求地址有效并等待握手成功，当返回的arready信号拉高时说明握手成功，拉高inst\_sram\_addr\_ok或data\_sram\_addr\_ok，表示请求已被接受。在此过程中，还需要判断现在的读请求是否与**写请求写数据通道已握手成功但还没有收到写响应**的写操作发生地址相关。需要考虑这种情况是因为读请求通道和写请求写数据通道在CPU中是在EX模块进行握手，而写响应通道是在MEM模块握手，因此当store指令后紧跟一个存在地址相关的load指令时，若load指令在store指令向内存写回数据前发送请求，由于读写是不同的通道，返回的读数据可能是旧数据，从而出错。判断如上地址相关情况的代码如下：



图1 读数据时需要考虑的地址相关情况

读请求握手成功后，读响应通道会拉高rready信号，并等待数据返回。当rvalid拉高时，读响应通道完成握手，此时收到的数据可以传递给CPU（inst\_sram\_rdata或data\_sram\_rdata），并拉高inst\_sram\_data\_ok或data\_sram\_data\_ok表示返回的数据有效。

另外，对于不同id的读请求（在本题中即inst\_sram和data\_sram发出的读请求），可以在读请求通道握手成功后便进行下一次读请求，而不必等到读响应通道握手成功，因此我们使用了一个计数器read\_wait\_counter来表示请求以握手成功但未响应的读操作数量，其相关代码如下：

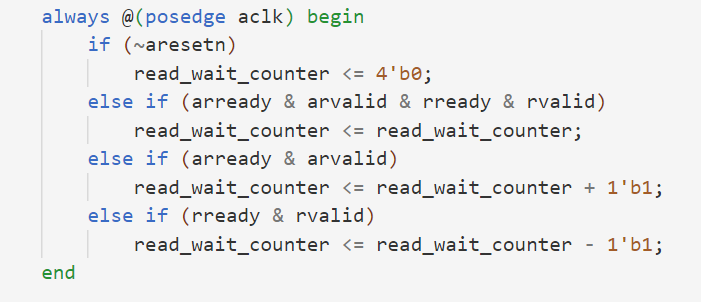


图2 read\_wait\_data信号赋值逻辑

同时由于id是4位宽的，虽然在本次实验中我们只会用到0和1两个值，但是为了扩展方便，我们还设置了有16元素32位宽的寄存器数组rdata\_buffer。在读相应通道握手成功时，它会根据rid向对应的rdata\_buffer填充数据，等到向CPU返回数据时，也是根据rid来判断返回哪一个rdata\_buffer中的数据。寄存器rdata\_buffer的赋值逻辑如下：

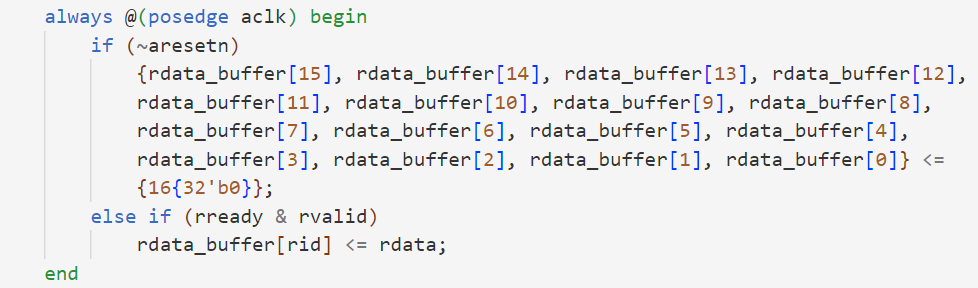


图3 rdata\_buffer信号赋值逻辑

若是写请求，AXI\_bridge会拉高awvalid和wvalid信号，表示写请求和写地址有效并等待握手成功，当返回的awready和wready信号拉高时说明握手成功（这两个信号可能不同时拉高），拉高inst\_sram\_addr\_ok或data\_sram\_addr\_ok，表示请求已被接受。而后写响应通道会拉高bready信号，并等待bvalid信号拉高。当bvalid拉高时，写响应通道完成握手，此时以成功向内存写回数据，拉高inst\_sram\_data\_ok或data\_sram\_data\_ok表示写操作完成。需要指出的是，在CPU的类sram接口的赋值中，指令sram不会发出写请求，因此实际上inst\_sram\_addr\_ok等信号在写操作时不会拉高。

* + 1. Exp 16

exp16完善了exp15中处理错误取指请求的逻辑。在exp15中，本组成员并没有把pre-IF的取消指令与IF的取消指令的操作区分的很清楚，对其对应的cancel信号的复位也并未考虑很周全。同时exp15中，很多地方也没有对br\_stall信号进行合理的考虑，所以仅仅在无延迟种子仿真能过，其它的种子均无法通过。对于preif\_cancel信号，其设置的理由是由于类SRAM总线允许中途更改请求，而AXI总线不允许中途更改。因此哪怕错误的请求还没被接收，我们只能阻塞pre-IF级，拉高preif\_cancel信号，然后忽视后续返回的指令，再重新发正确的请求。可以看到，这里涉及的情形不但有异常和跳转，还有br\_stall，因为在指令进入ID级、拉高br\_stall前，它的后一条指令可能已经开始发送取指请求，因此不能仅仅通过在br\_stall为1时拉低req来处理br\_stall，而是需要先发完再取消。对于if\_cancel信号，则是由于请求握手成功，但是数据还未返回时，发现指令无效，此时需要拉高if\_cancel信号，并忽视其对应返回的指令。所以两个cancel信号都拉高时，需要保证一次data\_ok信号只能复位其中一个信号。

* 1. 重要模块1设计：类SRAM-AXI转接桥AXI\_bridge
     1. 工作原理

类SRAM-AXI转接桥与CPU流水级的类SRAM信号相连，根据收到的请求信号将其转化为对应的AXI接口请求信号，发送给AXI总线，等待外部响应，并向CPU流水级传回相应的响应信号。

* + 1. 接口定义

表1 32位AXI接口信号

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| aclk | IN | 1 | 时钟信号 |
| aresetn | IN | 1 | 复位信号 |
| **读请求通道** | | | |
| arid | OUT | 4 | 读请求ID号（取指置为0，取数置为1） |
| araddr | OUT | 32 | 读请求地址 |
| arlen | OUT | 8 | 请求传输的长度（数据传输所需的拍数）（固定为0） |
| arsize | OUT | 3 | 请求传输的大小（数据传输每拍的字节数） |
| arburst | OUT | 2 | 传输类型（固定为2’b01） |
| arlock | OUT | 2 | 原子锁（固定为0） |
| arcache | OUT | 4 | CACHE属性（固定为0） |
| arprot | OUT | 3 | 保护属性（固定为0） |
| arvalid | OUT | 1 | 读请求地址握手信号，读请求地址有效 |
| arready | IN | 1 | 读请求地址握手信号，slave端准备好接收地址传输 |
| **读响应通道** | | | |
| rid | IN | 4 | 读请求ID号，同一请求的arid和rid一致 |
| rdata | IN | 32 | 读回的数据 |
| rresp | IN | 2 | 读请求是否成功完成（可忽略） |
| rlast | IN | 1 | 读请求最后一拍数据的指示信号（可忽略） |
| rvalid | IN | 1 | 读请求数据握手信号，数据有效 |
| rready | OUT | 1 | 读请求数据握手信号，CPU准备好接收数据 |
| **写请求通道** | | | |
| awid | OUT | 4 | 写请求ID号（固定为1） |
| awaddr | OUT | 32 | 写请求地址 |
| awlen | OUT | 8 | 请求传输的长度（数据传输所需的拍数）（固定为0） |
| awsize | OUT | 3 | 请求传输的大小（数据传输每拍的字节数） |
| awburst | OUT | 2 | 传输类型（固定为2’b01） |
| awlock | OUT | 2 | 原子锁（固定为0） |
| awcache | OUT | 4 | CACHE属性（固定为0） |
| awprot | OUT | 3 | 保护属性（固定为0） |
| awvalid | OUT | 1 | 写请求地址握手信号，写请求地址有效 |
| awready | IN | 1 | 写请求地址握手信号，内存准备好接收地址 |
| **写数据通道** | | | |
| wid | OUT | 4 | 写请求ID号（固定为1） |
| wdata | OUT | 32 | 写数据 |
| wstrb | OUT | 4 | 字节选通位 |
| wlast | OUT | 1 | 最后一拍指示信号（固定为1） |
| wvalid | OUT | 1 | 写请求数据握手信号，写请求数据有效 |
| wready | IN | 1 | 写请求数据握手信号，slave端准备好接收数据传输 |
| **写响应通道** | | | |
| bid | IN | 4 | 写请求ID号，同一请求的awid、rid和bi一致（可忽略） |
| bresp | IN | 2 | 写请求是否成功完成（可忽略） |
| bvalid | IN | 1 | 写请求响应握手信号，写请求响应有效 |
| bready | OUT | 1 | 写请求响应握手信号，CPU准备好接收响应 |
| **类SRAM接口** | | | |
| inst\_sram\_req | IN | 1 | 指令sram请求信号 |
| inst\_sram\_wr | IN | 1 | 指令sram读写控制信号 |
| inst\_sram\_size | IN | 2 | 指令sram该次请求传输的字节数 |
| inst\_sram\_wstrb | IN | 4 | 指令sram该次请求的字节写使能信号 |
| inst\_sram\_addr | IN | 32 | 指令sram请求地址 |
| inst\_sram\_wdata | IN | 32 | 指令sram该次写请求的写数据 |
| inst\_sram\_addr\_ok | OUT | 1 | 指令sram该次请求的地址传输OK |
| inst\_sram\_data\_ok | OUT | 1 | 指令sram该次请求的数据传输OK |
| inst\_sram\_rdata | OUT | 32 | 指令sram该次请求返回的读数据 |
| data sram 类SRAM接口（数据） | | | |
| data\_sram\_req | IN | 1 | 数据sram请求信号 |
| data\_sram\_wr | IN | 1 | 数据sram读写控制信号 |
| data\_sram\_size | IN | 2 | 数据sram该次请求传输的字节数 |
| data\_sram\_wstrb | IN | 4 | 数据sram该次请求的字节写使能信号 |
| data\_sram\_addr | IN | 32 | 数据sram请求地址 |
| data\_sram\_wdata | IN | 32 | 数据sram该次写请求的写数据 |
| data\_sram\_addr\_ok | OUT | 1 | 数据sram该次请求的地址传输OK |
| data\_sram\_data\_ok | OUT | 1 | 数据sram该次请求的数据传输OK |
| data\_sram\_rdata | OUT | 32 | 数据sram该次请求返回的读数据 |

* + 1. 功能描述

在AXI\_bridge中，设置了4个状态机，分别控制读请求通道、读相应通道、写请求和写数据通道以及写响应通道，4个状态机的初始状态均定义为IDLE，它们的工作流程如下：

* + - 1. 读请求通道

读请求通道状态机的状态转移图如下图所示

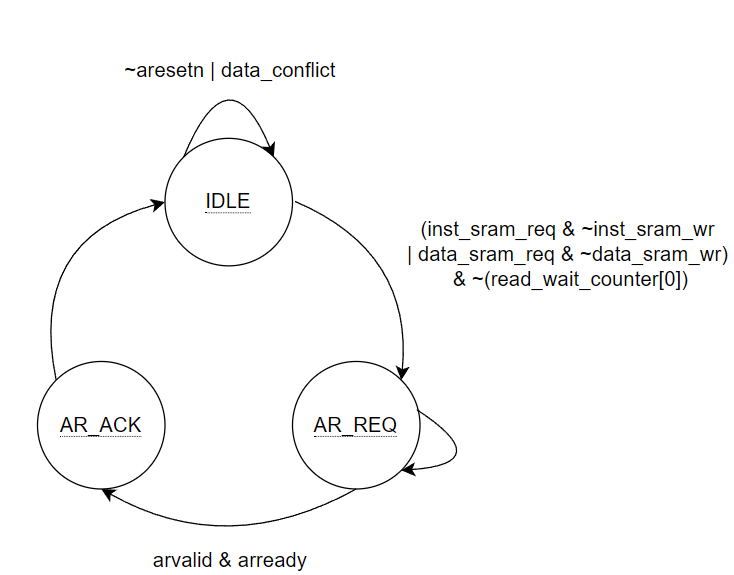


图4 读请求通道状态机状态转移图

在复位时，读请求状态机会被复位到IDLE状态，清零arid、araddr和arsize中存储的值，为arlen、arburst、arlock、arcache和arprot这几个值是固定的寄存器赋值。

复位后，当读请求状态机位于IDLE状态时，若收到类SRAM接口的读请求信号，会根据传入的信号更新arid、araddr和arsize的值，特别的，若指令sram和数据sram同时有未响应的请求，会优先响应数据sram的请求。若此时与正在执行的写请求不存在地址相关，会跳转到AR\_REQ状态，否则会阻塞在IDLE状态。在本次实验中，arid值只有可能为0或1，因此设置“已成功发送请求但还没有响应”的读操作的数量为2。若此时已有2个这样的读操作，又收到新的读请求，状态机也会被阻塞在IDLE状态。

在AR\_REQ状态，读请求通道会拉高arvalid信号，表示读请求地址有效并等待AXI总线返回arready信号。当arready信号拉高时，读请求通道会拉高inst\_sram\_addr\_ok或data\_sram\_addr\_ok信号，表示该次读请求的地址传输OK，状态机会跳转到AR\_ACK状态表示完成握手并在下一拍跳回IDLE阶段，等待下一轮的请求。

* + - 1. 读响应通道

读响应通道状态机的状态转移图如下图所示：

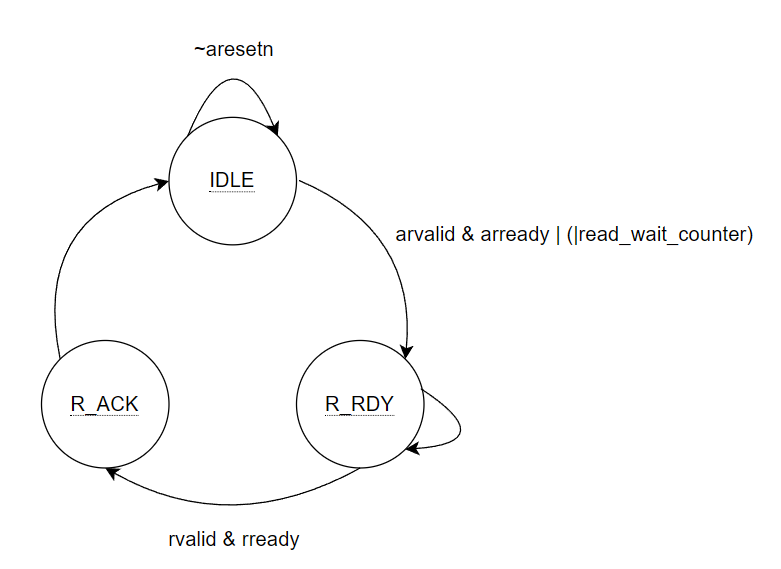


图5 读响应通道状态机状态转移图

在复位时，读响应状态机会被复位到IDLE状态。复位后，读响应状态机位于IDLE状态。当读请求被接受或有读地址握手成功但未得到相应的读操作，会跳转到R\_RDY状态。

在R\_RDY状态，读响应通道会拉高rready信号，表示准备好接收数据传输，等待rvalid信号拉高。当rvalid信号拉高时，读响应通道会存储rid到寄存器中并根据rid将AXI总线返回的数据存储到响应的寄存器中，状态机会跳转到R\_ACK状态。

在R\_ACK状态，读响应通道会根据本次读响应id值为inst\_sram\_rdata或data\_sram\_rdata信号赋值，并拉高inst\_sram\_data\_ok或data\_sram\_data\_ok信号，完成本次读操作。状态机会跳转到IDLE状态等待下次读响应。

* + - 1. 写请求和写数据通道

写请求和写数据通道状态机的状态转移图如下图所示：

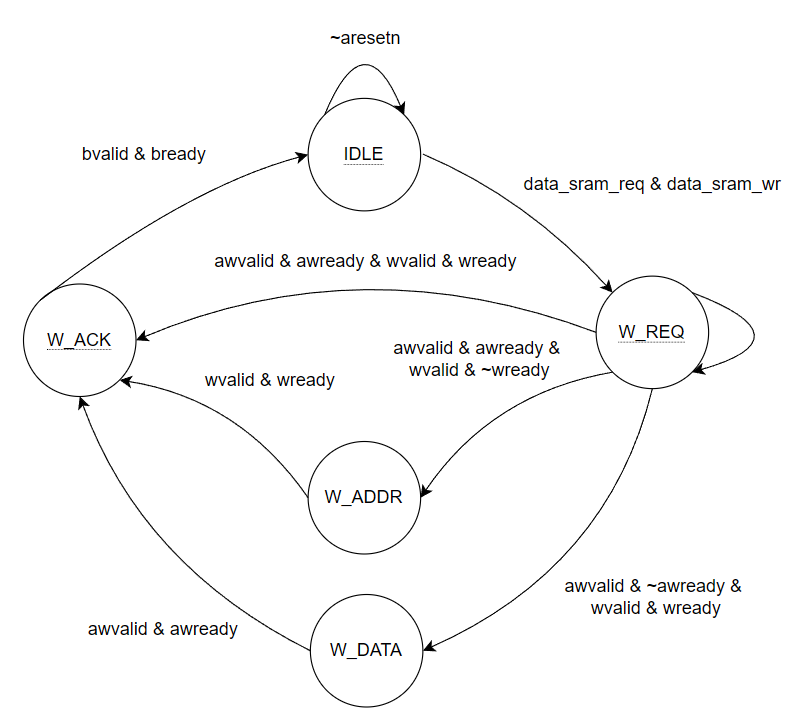


图6 读响应通道状态机状态转移图

在复位时，写请求和写数据状态机会被复位到IDLE状态，写请求和写数据通道会清零awaddr、awsize、wdata和wstrb中存储的值，为awid、awlen、awburst、awlock、 awcache、 awprot、wid和wlast这几个值是固定的寄存器赋值。复位后，当CPU发送写请求时，会根据传入的请求信号为awaddr、awsize、wdata和wstrb赋值，状态机会跳转到W\_REQ状态。

在W\_REQ状态，写请求和写数据通道会拉高awvalid信号和wvalid信号，表示写请求地址和写请求数据有效，并等待awready和wready信号拉高，若这两个信号同时拉高，状态机会跳转到W\_ACK状态；若只有awready信号拉高，状态机会跳转到W\_ADDR状态；若只有wready信号拉高，状态机会跳转到W\_DATA状态。

在W\_ADDR状态，wvalid会被拉高并等待wready拉高，wready信号拉高后状态机会跳转到W\_ACK状态。

在W\_DATA状态，awvalid会被拉高并等待awready拉高，awready信号拉高后状态机会跳转到W\_ACK状态。

在上述情况中，当awvalid和awready信号同时拉高时，会拉高data\_sram\_data\_ok信号，传递给CPU。

在W\_ACK状态，bready信号会被拉高，等待写响应通道工作，当bvalid信号拉高时，说明此次写操作完成，写请求和写数据状态机回到IDLE状态，等待处理下一次写操作。

* + - 1. 写响应通道

写响应通道状态机的状态转移图如下图所示：

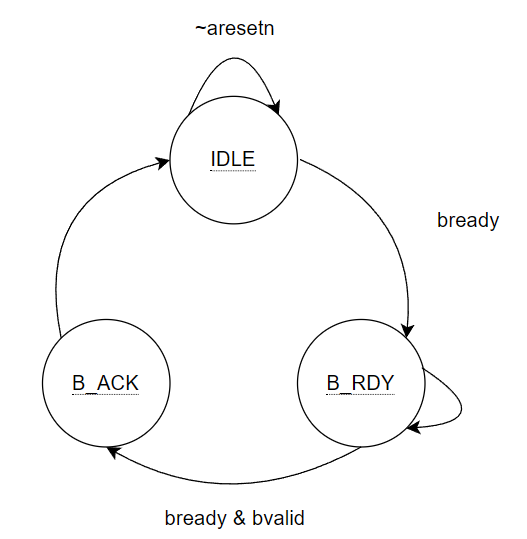


图7 写响应通道状态机状态转移图

在复位时，写响应状态机会被复位到IDLE状态。复位后，当写请求地址和写请求数据都完成握手后，即前文所述写请求和写数据通道状态机处于W\_ACK状态时，会拉高bready信号，此时写响应状态机会跳转到B\_RDY状态。

在B\_RDY状态，写响应通道会等待bvalid信号拉高。当bvalid信号拉高时，写响应通道会根据返回的bid值拉高inst\_sram\_data\_ok或data\_sram\_data\_ok信号（事实上指令sram不会发送写请求），表示本次写操作完成。状态机会跳转到B\_ACK状态并在再下一个周期跳转到IDLE状态等待处理下一次写响应。

* 1. 重要模块2设计：IF流水级
     1. 工作原理

IF模块内包含一个伪流水级pre-IF，pre-IF级负责发送请求并等待地址握手，IF级负责接收指令。如果需要取消指令，则将IF级或pre-IF级的对于cancel信号拉高，并把if\_inst\_reg\_valid信号置为0，从而完成对后面返回指令的取消

* + 1. 接口定义

表2 IF流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| inst\_sram\_req | OUT | 1 | 指令sram请求信号 |
| inst\_sram\_wr | OUT | 1 | 指令sram读写控制信号 |
| inst\_sram\_size | OUT | 2 | 指令sram该次请求传输的字节数 |
| inst\_sram\_wstrb | OUT | 4 | 指令sram该次请求的字节写使能信号 |
| inst\_sram\_addr | OUT | 32 | 指令sram请求地址 |
| inst\_sram\_wdata | OUT | 32 | 指令sram该次写请求的写数据 |
| inst\_sram\_addr\_ok | IN | 1 | 指令sram该次请求的地址传输OK |
| inst\_sram\_data\_ok | IN | 1 | 指令sram该次请求的数据传输OK |
| inst\_sram\_rdata | IN | 32 | 指令sram该次请求返回的读数据 |
| axi\_arid | IN | 4 | AXI转接桥的读请求ID号 |
| id\_allowin | IN | 1 | ID流水级是否允许IF流水级传入数据 |
| br\_taken | IN | 1 | ID流水级传来的跳转信号 |
| br\_stall | IN | 1 | ID流水级传来的跳转阻塞信号 |
| br\_target | IN | 32 | ID流水级传来的跳转地址 |
| if\_to\_id\_valid | OUT | 1 | 标记IF流水级向ID流水级传递的数据是否有效 |
| if\_to\_id\_data | OUT | 64 | IF模块向ID模块传递的数据 |
| if\_to\_id\_excep | OUT | 1 | IF模块向ID模块传递的异常信息 |
| wb\_to\_if\_csr\_data | IN | 66 | WB模块传给IF模块的csr数据 |
| if\_flush | IN | 1 | 传给IF模块的清空流水线信号 |

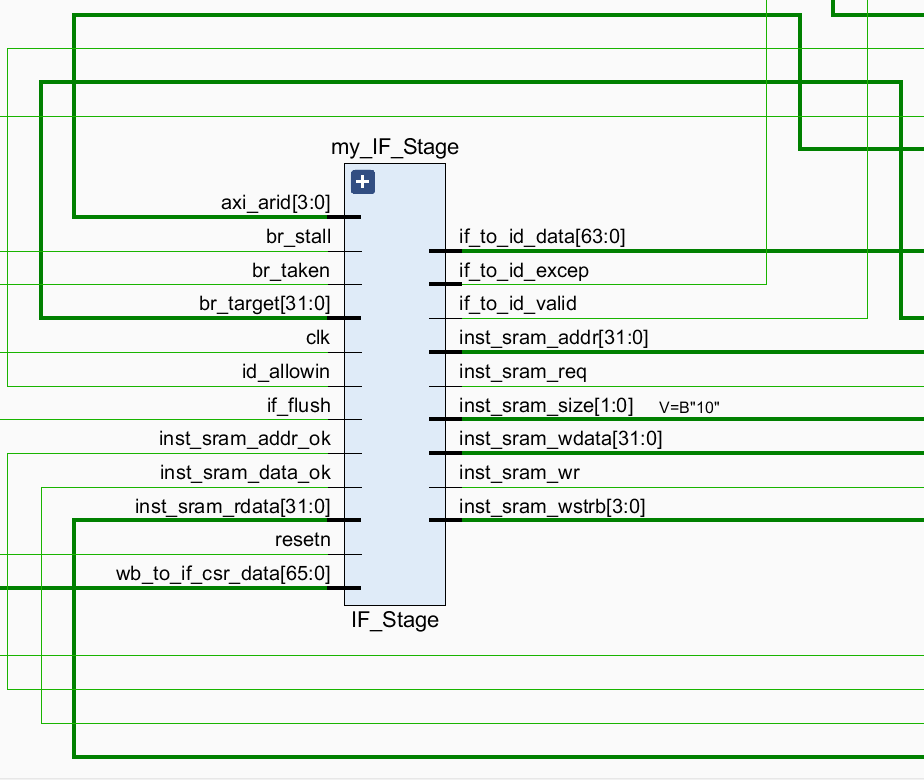


图8 IF模块接口

* + 1. 功能描述
       1. 接口赋值

面向指令RAM的类SRAM接口赋值如下：

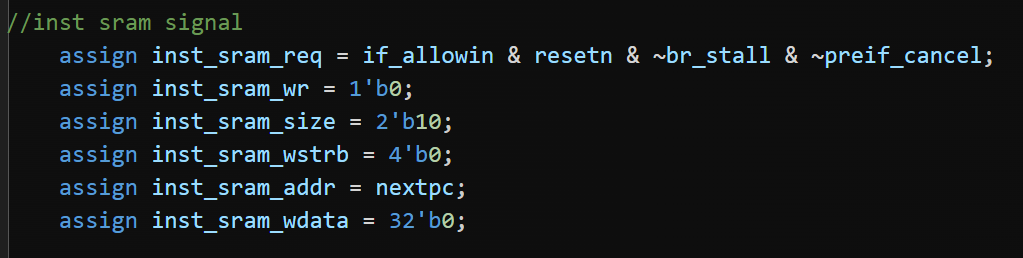


图9 面向指令RAM的类SRAM接口

由于只进行读访问，wr、wstrb和wdata恒置0。指令长度始终为32位，size恒为2’b10。访问地址为预测的nextpc。访问请求在IF流水级allowin，ID阶段没有被阻塞的条件分支指令，且pre-IF取消信号未拉高的情况下发出。

if\_allowin为1是为了保证发完请求后流入下一级；加入~preif\_cancel则是为了控制在等待错误的指令返回的过程中不要再发请求。等错误指令返回之后再重新发送正确的取指请求；加入~br\_stall则是需要等待ID流水级获得正确的前递数据来判断跳转分支。其中preif\_cancel的具体的更新逻辑在后面进行进一步阐述。

另外，本实验中由于wb\_ertn\_flush\_valid、wb\_csr\_ex\_valid、br\_taken等信号都可能在请求被发出前发生变化，因此要为这些信号设置对应的寄存器，在对应信号为1时存入1，在流水级从pre-IF流向IF时还原成0。并用寄存器存储其对应的nextpc值。最终nextpc当前赋值逻辑如下：

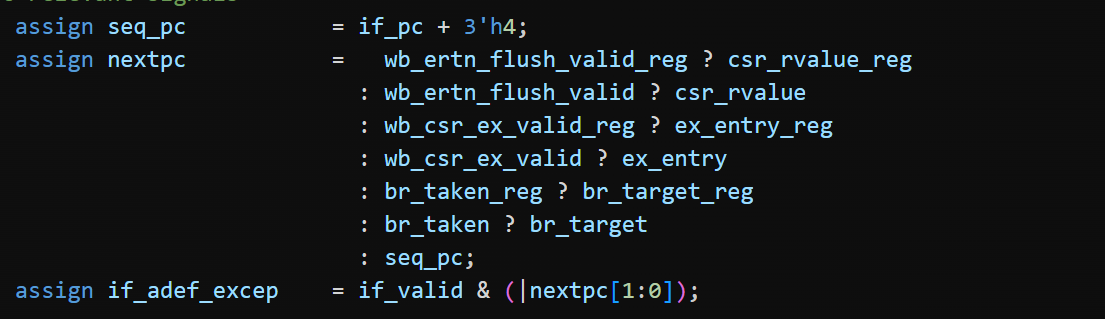


图10 nextpc赋值逻辑

* + - 1. 流水级控制信号设计

**Pre-IF流水级：**

ready\_go信号：req & addr\_ok握手成功。

to\_if\_valid信号：pre-IF的ready\_go与IF的allowin握手成功，且没有被取消或刷新。

**IF流水级：**

ready\_go信号：收到指令（data\_ok或指令缓存有效）且没有被取消。

allowin信号：在IF级内容无效或已经与ID级握手完成时允许进入。

valid信号：在allowin时根据to\_if\_valid赋值，在分支成立或刷新时无效。

if\_to\_id\_valid信号：IF级内容有效，ready\_go拉高，且没有被刷新。

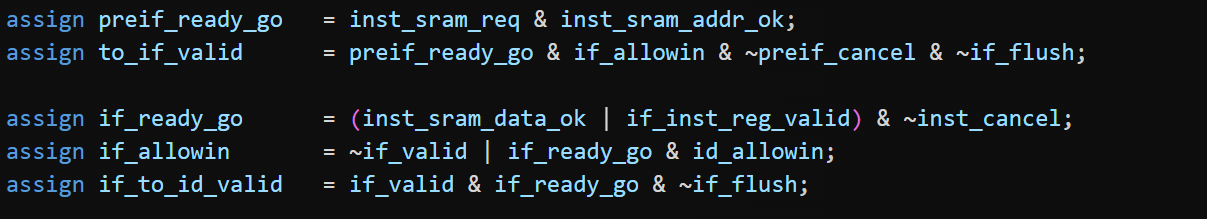


图11 IF流水级的握手信号

**指令寄存器与其对应的valid寄存器：**

IF级负责接收指令，为此设置了一个寄存器if\_inst\_reg和对应的if\_inst\_reg\_valid寄存器。如果发现跳转或者发现中断异常信号以及中断返回信号，或是id\_allowin为1，指令流入ID级，则把valid信号变成0。否则，如果i指令返回，data\_ok拉高，并且不存在取消指令的情况，并且if\_inst\_reg中之前没有有效数据，则将指令存入if\_inst\_reg，并将valid信号变成1。在IF级拿到指令，或是if\_inst\_reg中已经有有效指令时，if\_ready\_go为1，否则为0

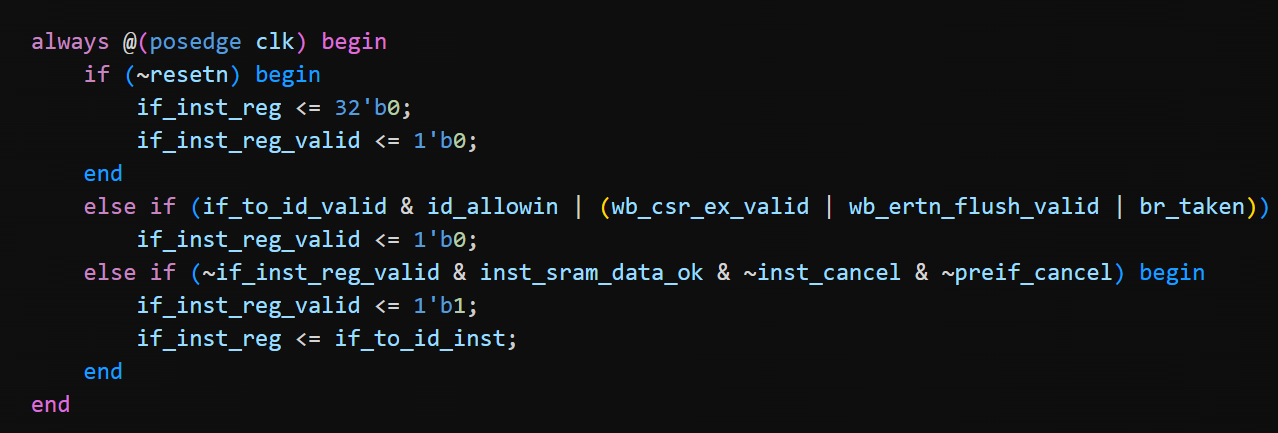


图12 指令寄存器与对应的valid寄存器更新逻辑

IF级负责接收指令，为此设置了一个寄存器if\_inst\_reg和对应的if\_inst\_reg\_valid寄存器。如果发现跳转或者发现中断异常信号以及中断返回信号，或是id\_allowin为1，指令流入ID级，则把valid信号变成0。否则，如果i指令返回，data\_ok拉高，并且不存在取消指令的情况，并且if\_inst\_reg中之前没有有效数据，则将指令存入if\_inst\_reg，并将valid信号变成1。在IF级拿到指令，或是if\_inst\_reg中已经有有效指令时，if\_ready\_go为1，否则为0。

**指令取消逻辑：**

本次实验需要设置preif\_cancel和inst\_cancel两个取消指令的信号。对于preif\_cancel信号，其设置的理由是由于类SRAM总线允许中途更改请求，而AXI总线不允许中途更改。因此哪怕错误的请求还没被接收，我们只能阻塞pre-IF级，拉高preif\_cancel信号，然后忽视后续返回的指令，再重新发正确的请求。对于if\_cancel信号，则是由于请求握手成功，但是数据还未返回时，发现指令无效，此时需要拉高if\_cancel信号，并忽视其对应返回的指令。所以两个cancel信号都拉高时，需要保证一次data\_ok信号只能复位其中一个信号。

preif\_cancel的具体更新逻辑为，在握手信号inst\_sram\_req或者br\_stall拉高的情况下（br\_stall拉高时，inst\_sram\_req拉低，但在之前已经发送过req请求了），发现此时对应的取指地址有误，需要取消时向其中存入1，这里涉及的情形不但有异常和跳转，还有br\_stall，因为在指令进入ID级、拉高br\_stall前，它的后一条指令可能已经开始发送取指请求，因此不能仅仅通过在br\_stall为1时拉低req来处理br\_stall，而是需要先发完再取消。等到错误的数据返回的同时data\_ok拉高，且此时IF的cancel信号（即inst\_cancel）未拉高(防止一次data\_ok复位两个cancel信号)时再将preif\_cancel变为0。具体代码如下：

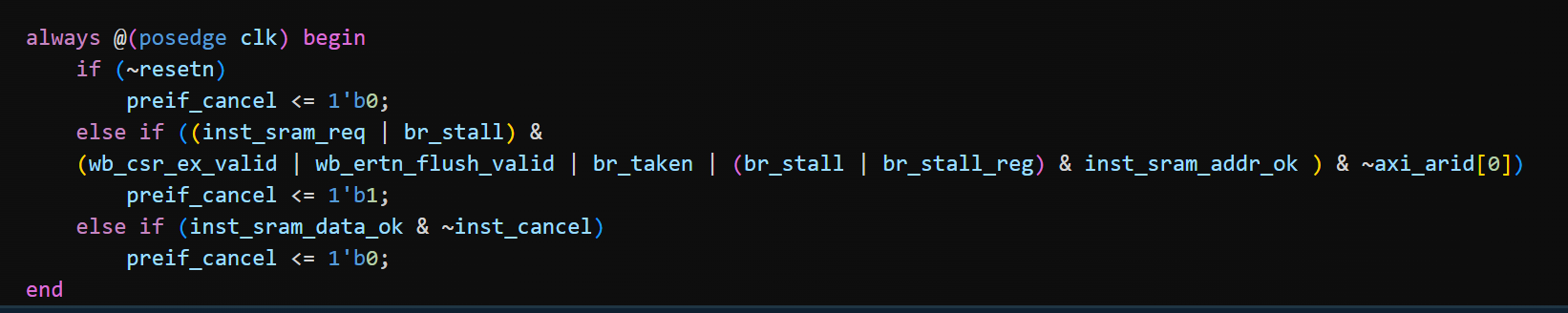


图13 preif\_cancel信号更新逻辑

inst\_cancel则需要使用一个计数器inst\_cancel\_num来存储需要取消的指令数，这是因为可能会发生有两条请求握手成功的取值操作对应的取值地址均是错误的情况，具体而言则是当异常前是跳转指令时，可能两次握手成功的取值操作都是无效的。inst\_cancel\_num更新逻辑则是：如果在请求握手成功后，数据返回前，发现是跳转分支或者异常中断，则把值加一，在数据返回后，把值减一。Inst\_cancel\_num的更新不需要考虑br\_stall，因为br\_stall会把req拉低，不会出现请求握手成功的情况。而对于请求未握手成功的取消操作，在preif\_cancel中已经进行了处理。具体代码如下：

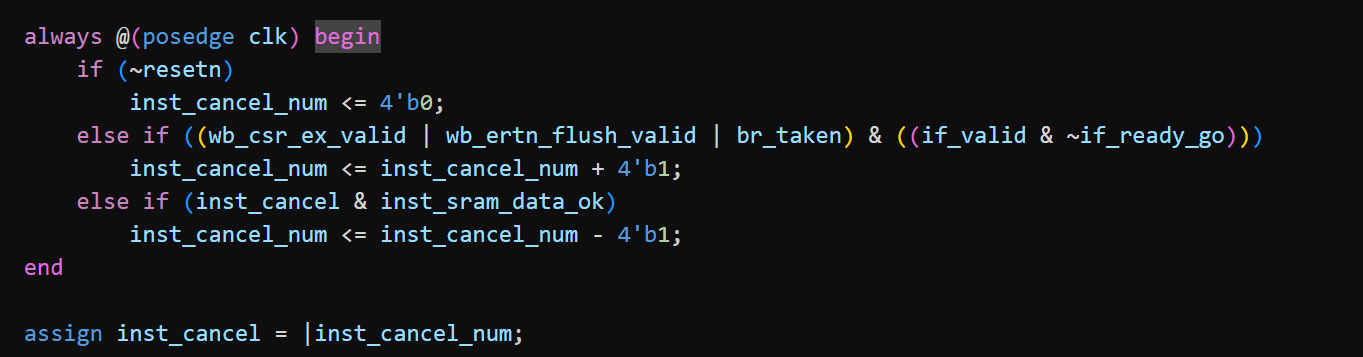


图14 inst\_cancel\_num与inst\_cancel赋值逻辑

而对于请求握手成功且数据已经返回，发现取指地址出错的处理，则在前面的实验中最初设计流水线时已经实现了，具体而言就是把if\_valid和id\_valid信号进行清0，使得其流水级对应的指令无效。

* 1. 重要模块3设计：ID流水级
     1. 工作原理

增加br\_stall信号，发送给IF流水级。用途为在ID级条件分支指令因为写后读冲突被阻塞时，暂停IF级向指令RAM发送取指请求。其余不变。

* + 1. 接口定义

表3 ID流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| id\_allowin | OUT | 1 | ID流水级允许IF流水级传入数据 |
| br\_taken | OUT | 1 | 传给IF流水级的跳转信号 |
| br\_stall | OUT | 1 | 传给IF流水级的跳转阻塞信号 |
| br\_target | OUT | 32 | 传给IF流水级的跳转地址 |
| if\_to\_id\_valid | IN | 1 | 标记IF流水级传入ID流水级的数据是否有效 |
| if\_to\_id\_data | IN | 64 | IF流水级传给ID流水级的数据 |
| if\_to\_id\_excep | IN | 1 | IF流水级传给ID流水级的异常信息 |
| ex\_allowin | IN | 1 | EX流水级允许ID流水级传入数据 |
| id\_to\_ex\_data | OUT | 163 | ID流水级传入EX流水级的数据 |
| id\_to\_ex\_excep | OUT | 86 | ID流水级传入EX流水级的异常信息 |
| id\_to\_ex\_valid | OUT | 1 | 标记ID流水级传入EX流水级的数据是否有效 |
| wb\_rf\_zip | IN | 38 | WB流水级向ID流水级传递的regfile的写回信息和前递数据 |
| mem\_rf\_zip | IN | 39 | MEM流水级前递到ID流水级的数据 |
| ex\_rf\_zip | IN | 40 | EX流水级前递到ID流水级的数据 |
| id\_flush | IN | 1 | ID模块收到的清空流水线信号 |
| has\_int | IN | 1 | 若WB模块判断有中断，将ID阶段的指令进行标记 |

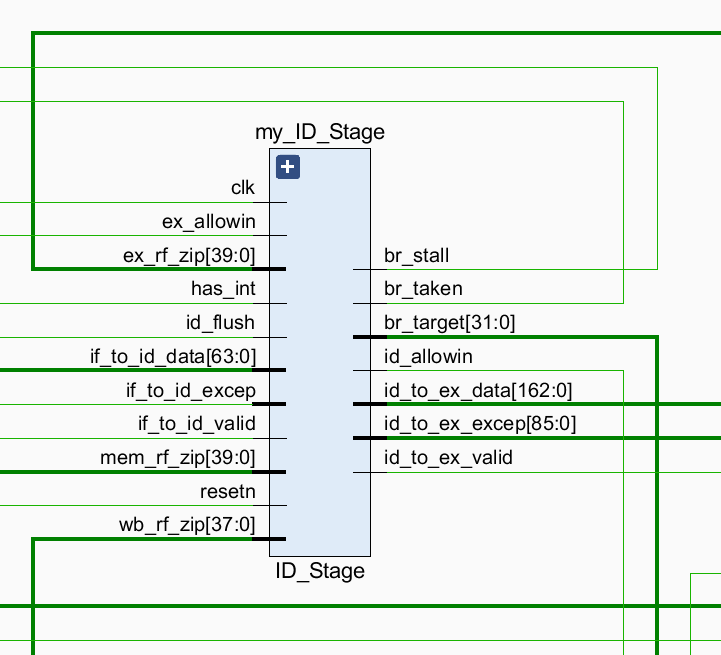


图15 ID流水级接口

* + 1. 功能描述

在此次实验中，如果ID级是转移指令，而它的前一条指令是load指令，又恰好存在数据相关，那么ID级由于延迟会无法通过前递获得数据。为了解决这个问题，添加了br\_stall信号逻辑如下，在条件分支指令因为写后读冲突被阻塞时拉高：

br_stall

图16 br\_stall赋值逻辑

该信号会传递给IF流水级，功能为暂停取值请求的发送，等待得到条件分支指令的判断结果后再决定是否取消当前已发送取值请求进入IF级的指令（具体处理已经在IF流水级进行了给出）。

* 1. 重要模块4设计：EX流水级
     1. 工作原理

EX阶段访问数据RAM的接口改变为类SRAM接口。对于访存指令，在EX阶段发送访存请求并等待地址握手成功，地址握手成功后（addr\_ok & req）进入MEM阶段。其余功能不变。

* + 1. 接口定义

表4 EX流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| ex\_allowin | OUT | 1 | EX模块允许ID模块传入数据 |
| id\_to\_ex\_data | IN | 163 | ID模块传入EX模块的数据 |
| id\_to\_ex\_excep | IN | 86 | ID模块传入EX模块的异常信息 |
| id\_to\_ex\_valid | IN | 1 | 标记ID模块传入EX模块的数据是否有效 |
| mem\_allowin | IN | 1 | MEM模块允许EX模块传入数据 |
| ex\_to\_mem\_data | OUT | 78 | EX模块传入MEM模块的数据 |
| ex\_to\_mem\_excep | OUT | 87 | EX模块传入MEM模块的异常信息 |
| ex\_to\_mem\_valid | OUT | 1 | 标记EX模块传入MEM模块的数据是否有效 |
| ex\_rf\_zip | OUT | 39 | EX模块前递到ID模块的数据 |
| mul\_result | OUT | 64 | 乘法器得到的结果 |
| data\_sram\_req | OUT | 1 | 数据ram请求信号 |
| data\_sram\_wr | OUT | 1 | 数据ram读写控制信号 |
| data\_sram\_size | OUT | 2 | 数据ram请求字节数 |
| data\_sram\_wstrb | OUT | 4 | 数据ram写使能信号 |
| data\_sram\_addr | OUT | 32 | 数据ram请求地址 |
| data\_sram\_wdata | OUT | 32 | 数据ram写数据 |
| data\_sram\_addr\_ok | IN | 1 | 数据ram地址握手信号 |
| ex\_flush | IN | 1 | 清空流水线信号 |
| mem\_to\_ex\_excep | IN | 1 | MEM模块向EX前传的异常信息 |

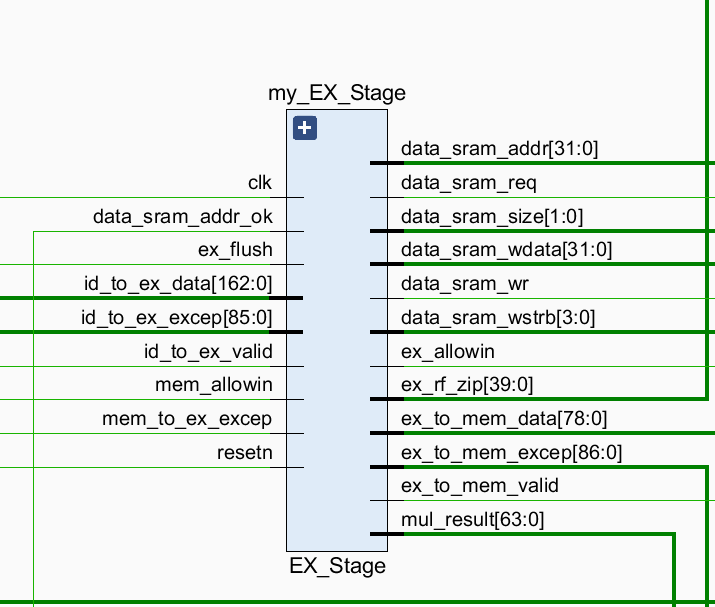


图17 EX流水级接口

* + 1. 功能描述
       1. 接口赋值

面向数据RAM的类SRAM接口赋值如下：

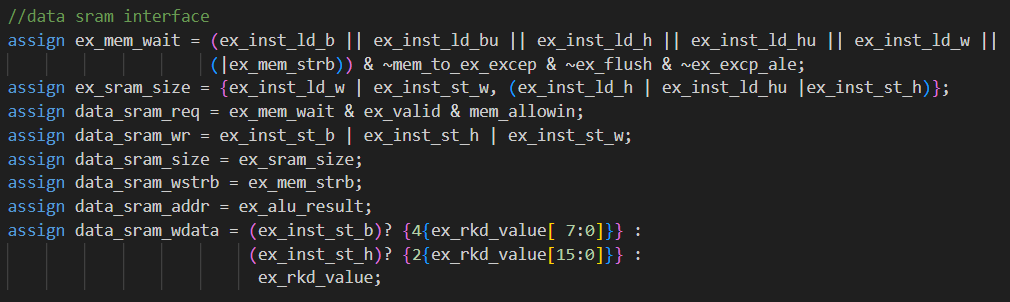


图18 面向数据RAM的类SRAM接口

wait信号表示当前存在访存请求且有效，赋值逻辑为存在访存指令且未发生例外，未被刷新。该信号会传递给MEM流水级，用于MEM流水级判断是否需要等待数据返回而阻塞流水级。

req信号在存在访存请求且有效，EX流水级有效，且MEM阶段允许进入时拉高。这样做同样可以保证在发请求后能直接进入MEM流水级。而且可以保证不需要进行取消的操作，因为必定会等到上一条data\_ok返回后才能发送下一跳的req请求。

wr信号在当前为store指令时拉高，其拉高代表对数据SRAM的写操作，拉低代表对数据SRAM的读操作。

size信号根据访存指令的访问位数进行赋值。

strb信号不变，根据写的位数与地址最低两位赋值。

addr信号不变，为alu的计算结果。

wdata逻辑不变，将8位与16位的写数据进行重复扩展到32位。

* + - 1. 流水级控制信号改变

当前控制信号（ready\_go、allowin、valid）逻辑如下图所示：

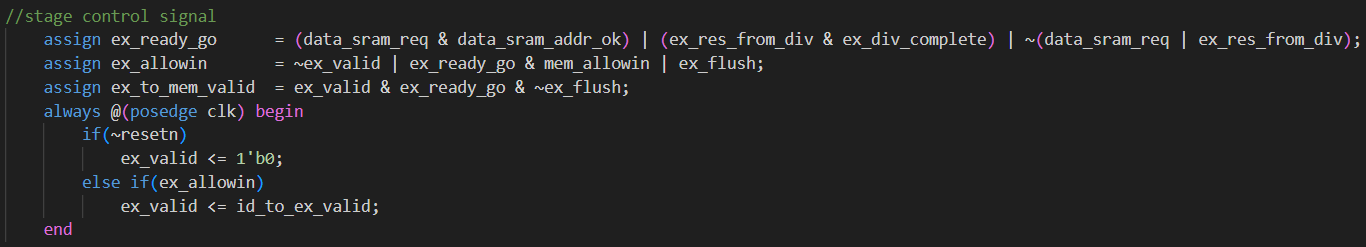


图19 EX流水级阻塞逻辑

对于ready\_go信号，本次实验进行了修改，如果需要访存，则要等待请求握手成功后才能为1，否则沿用之前实验的逻辑，在有除法时等除法计算完后置为1，其余情况恒为1。

* 1. 重要模块5设计：MEM流水级
     1. 工作原理

对于访存指令，MEM阶段在地址握手成功后进入，等待数据握手成功（data\_ok）后才能进入WB阶段。其余功能不变。

* + 1. 接口定义

表5 MEM流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| mem\_allowin | OUT | 1 | MEM模块允许EX模块传入数据 |
| ex\_to\_mem\_data | IN | 78 | EX模块传入MEM模块的数据 |
| ex\_to\_mem\_excep | IN | 87 | EX模块传入MEM模块的异常或中断数据 |
| ex\_to\_mem\_valid | IN | 1 | 标记EX模块传入MEM模块的数据是否有效 |
| wb\_allowin | IN | 1 | WB模块允许MEM模块传入数据 |
| mem\_to\_wb\_data | OUT | 70 | MEM模块传入WB模块的数据 |
| mem\_to\_wb\_excep | OUT | 119 | MEM模块传入WB模块的异常或中断数据 |
| mem\_to\_wb\_valid | OUT | 1 | 标记MEM模块传入WB模块的数据是否有效 |
| data\_sram\_data\_ok | IN | 1 | 数据ram数据握手信号 |
| data\_sram\_rdata | IN | 32 | 数据ram读数据 |
| mul\_result | IN | 64 | 乘法器运算结果 |
| mem\_rf\_zip | OUT | 38 | MEM模块前递到ID模块的数据 |
| mem\_reflush | IN | 1 | 清空流水级信号 |
| mem\_to\_ex\_excep | OUT | 1 | MEM级向EX级传递异常或中断信号 |

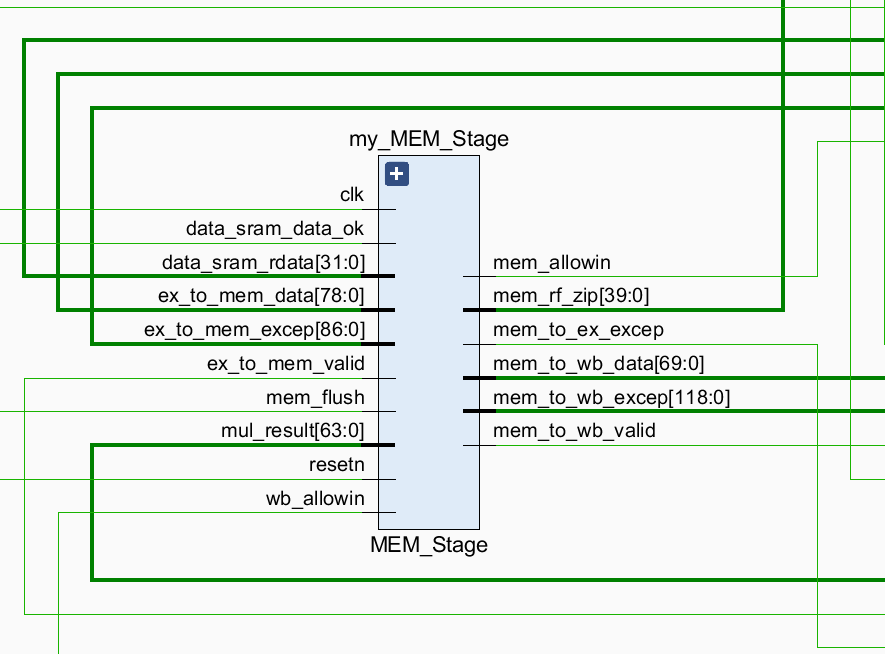


图20 MEM流水级接口

* + 1. 功能描述
       1. 流水级控制信号改变

当前控制信号（ready\_go、allowin、valid）逻辑如下图所示：

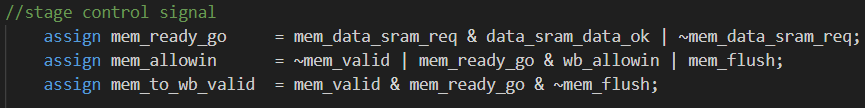


图21 MEM流水级阻塞逻辑

由于在本次实验中增加了总线接口，所以对于ready\_go信号，如果当前为访存指令时，在没有等到类SRAM接口返回的data\_sram\_data\_ok时，将流水线阻塞在MEM级直到获得有效的读数据。其它情况则与之前一样，一直设置为1即可。

将load指令进行阻塞的原因为需要等待读数据返回。将store指令进行阻塞的原因为避免load指令紧接store指令，load指令收到store指令的data\_ok并误认为是自己的data\_ok的情况。

* + - 1. 读数据处理

虽然load指令发送申请时的size信号按照load指令的字节数赋值，地址也不一定4字节对齐，但经过仿真发现，数据RAM返回的读数据的结尾依然是4字节对齐的，只是在读1或2字节的情况下，未被读取到的部分被掩盖掉，为全0。因此继续使用之前对rdata的处理逻辑，如下：

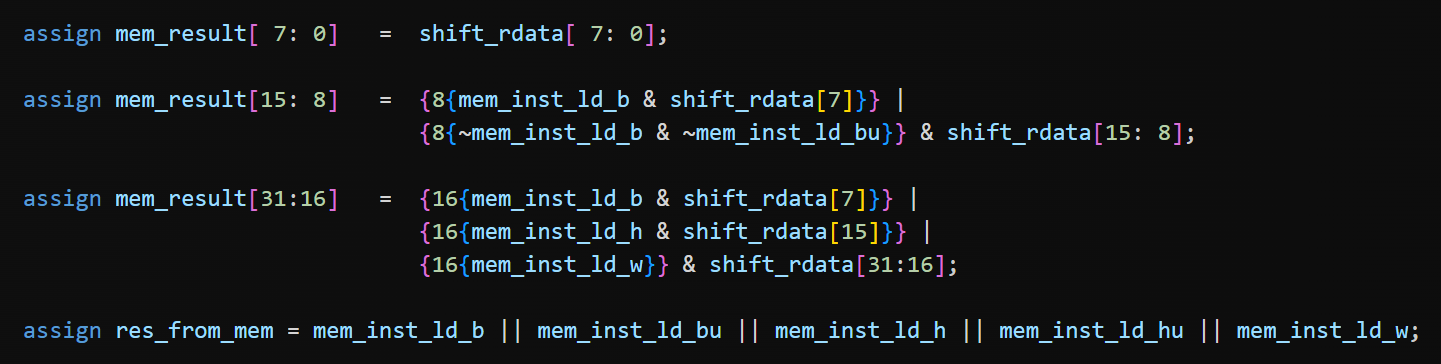


图22 MEM流水级读数据处理

1. 实验过程
   1. 实验流水账

李金明

2023.11.12 21：00-22：00 阅读讲义

2023.11.13 8：00-12：00 完善exp14代码

2023.11.13 12：20-15：00 18：00-21：00 完成exp14的debug

2023.11.17 14:00-15:00 阅读讲义

2023.11.17 15:00-22:00 完成exp15代码

2023.11.17 22:00-次日00:30 exp15的debug

2023.11.18 8:30-11:40 完成exp15的debug

贾城昊

2023.11.13 8:00-9:00 阅读讲义

2023.11.13 10:00-12:00 完善exp14代码

2023.11.13 15:00-21:00 完成exp14debug

2023.11.18 16:00-17:00 阅读讲义

2023.11.18 17：00-23：00 进行exp16的debug

2023.11.19 8：00-12：00 完成exp16的debug

牛浩宇

2023.11.9晚 阅读讲义&增加exp14接口并完成大致逻辑

2023.11.16晚 阅读讲义&增加exp15接口与转接桥。

* 1. 错误记录
     1. 错误1：exp14中跳转指令目标地址指令可能被跳过
        1. 错误现象

Console报错如下：

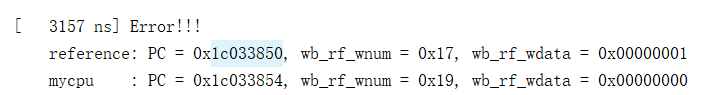


图23 错误1对应的Console报错

* + - 1. 分析定位过程

查看波形图及相关的汇编代码，可知错误发生在跳转指令目的地址的指令，其指令值没有被取出来就被跳过了，转而执行了目的地址的下一条指令。



图24 错误1对应的波形图



图25 错误1对应的汇编代码（1）



图26 错误1对应的汇编代码（2）

* + - 1. 错误原因

这里会出错是因为当有一条跳转指令时，跳转指令的下一条指令也会被送到inst\_sram并发出请求。虽然跳转指令此时已经在ID阶段，可以判断出存在跳转，需要取消下一条指令，但是需要被取消的那条指令以及发出请求并且可能在当拍返回了addr\_ok，此时当inst\_sram\_raddr切换到跳转指令目的地址后，可能会返回之前被取消指令地址读出的数据，且data\_ok拉高会让跳转指令目的地址的那条指令误以为取到了对应的指令而跳过。因此需要修改next\_pc的赋值。

* + - 1. 修正效果

增加代码：

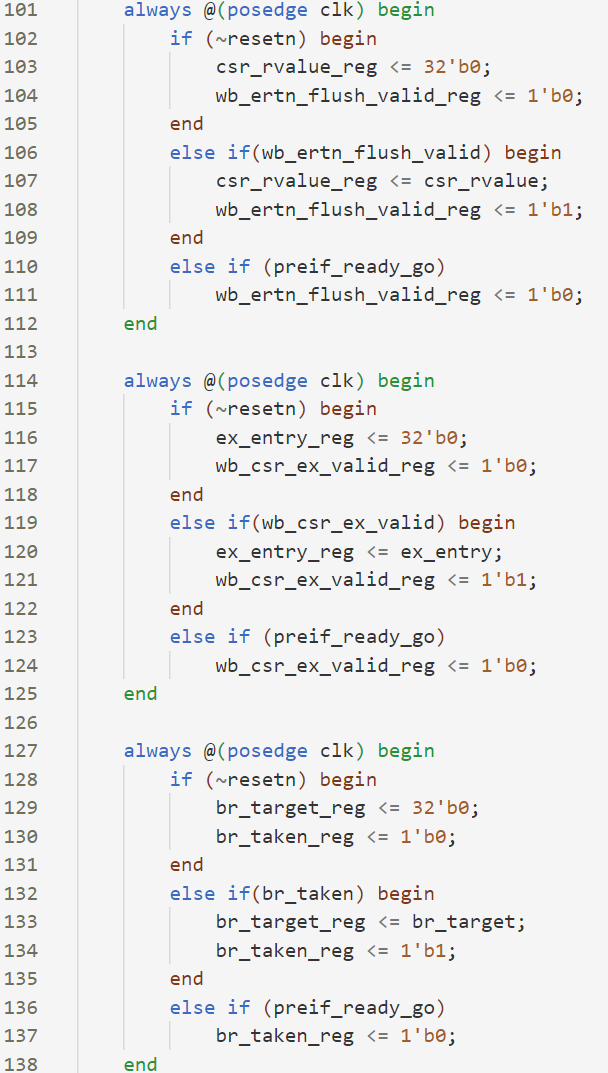


图27 错误1对应的修改代码（1）

修改next\_pc的赋值逻辑为：

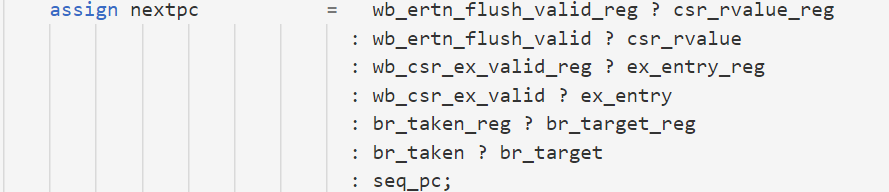


图28 错误1对应的修改代码（2）

此处逻辑为当有例外、ertn指令或跳转时，则将跳转地址存入一个寄存器并将有效位置为1，保证在addr\_ok前请求地址都是csr\_rvalue、ex\_entry或是br\_target，避免在指令地址握手成功前next\_pc就又被更新了。

该方法有效，来到下一个bug。

* + 1. 错误2：exp14中当其他流水级阻塞拍数过多时除法操作没有及时停止
       1. 错误现象

Console报错如下

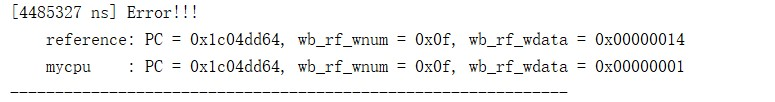


图29 错误2对应的Console报错

* + - 1. 分析定位过程

查阅汇编代码可知，出错的是一条除法指令。查看执行该除法指令的过程，波形图如下：

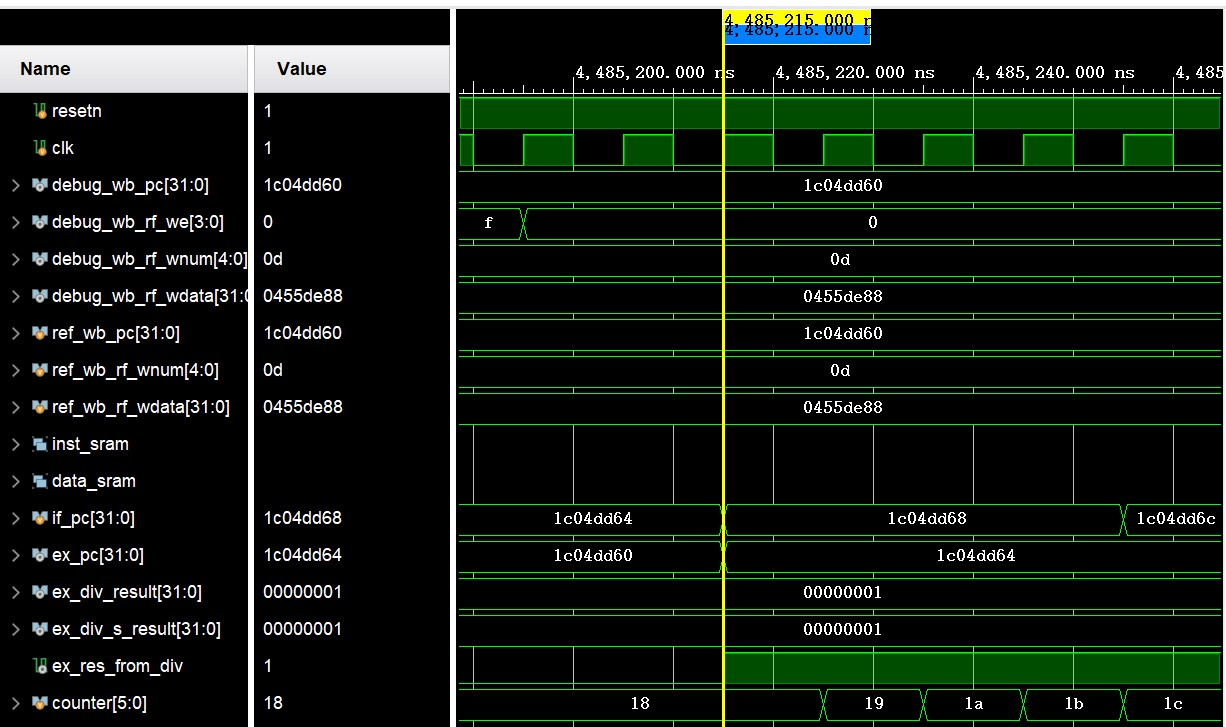


图30 错误2对应的波形图（1）

可以发现除法器对执行时钟周期计数的计数器初值不为0，导致除法器错误。查看上一次执行除法指令相关的波形图：

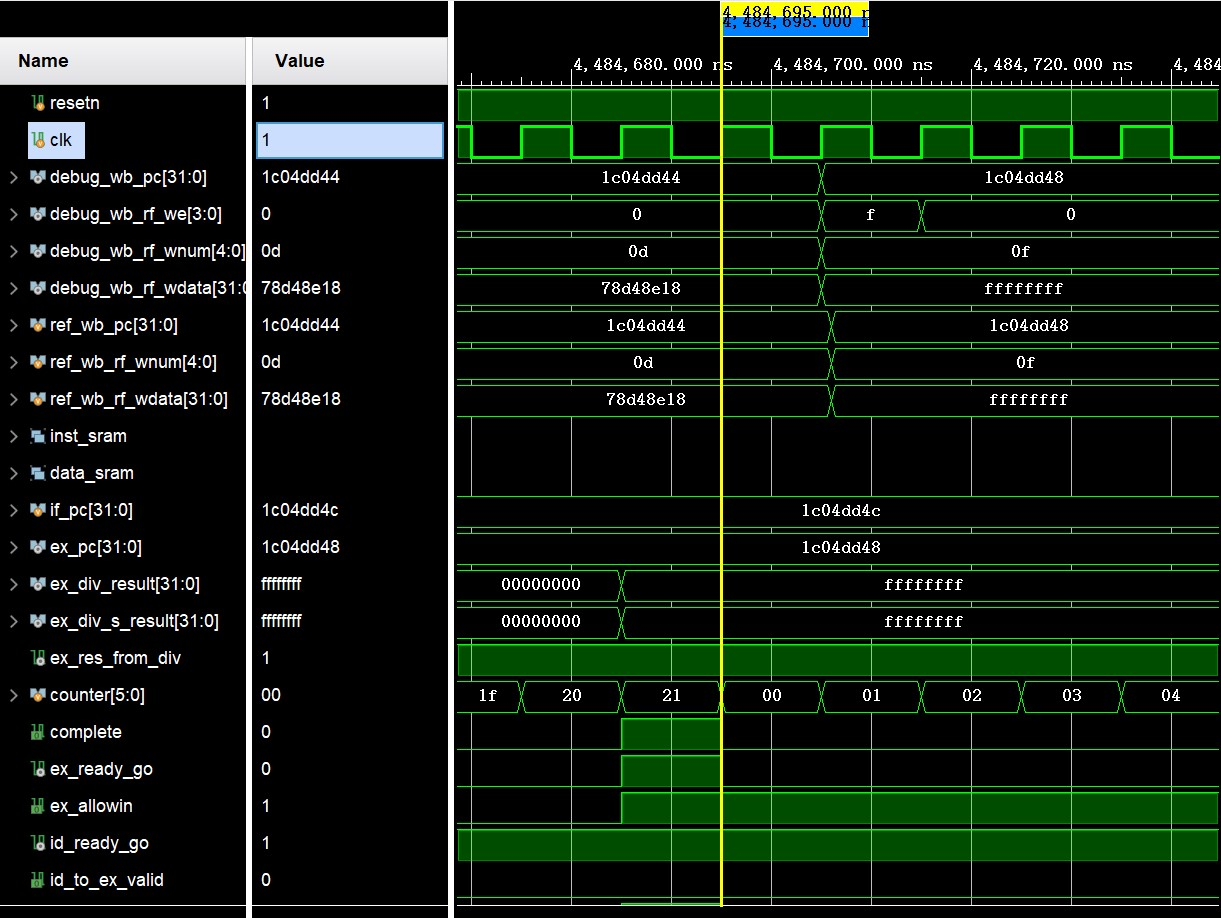


图31 错误2对应的波形图（2）

可以发现，当counter达到最大值0x21时，本应说明此时除法已经执行完毕，但是counter又从0开始加，没有结束。

* + - 1. 错误原因

除法器没有正常结束工作是因为某个流水级出现了多于33拍的阻塞，此时由于被阻塞，EX模块中传入除法器的div信号还是拉高的，导致除法器继续工作。

* + - 1. 修正效果

使用一个寄存器作为除法指令已经完成的标志，当除法器工作完成后，通过该标志将传入除法器的div信号拉低，避免除法器过度工作。修改代码如下：

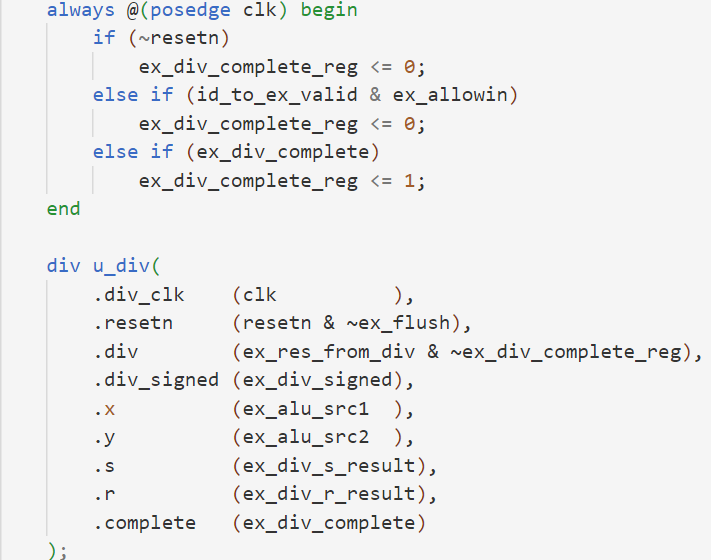


图32 错误2修改代码

该方法有效，来到下一个bug。

* + 1. 错误3：exp14中data\_sram\_size传入位宽出错
       1. 错误现象

Console报错如下

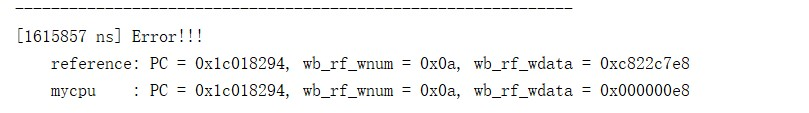


图33 错误3对应的Console报错

* + - 1. 分析定位过程

通过查看汇编代码，可以知道出错的是一条load指令，因此查看该条指令在EX阶段向data\_sram传送的请求信号，相关波形图如下：



图34 错误3对应的波形图

可以发现，出错的是一条ld.w指令，向sram传输的data\_sram\_size的值本应为2，但是此处为0。查看data\_sram\_size赋值相关代码：

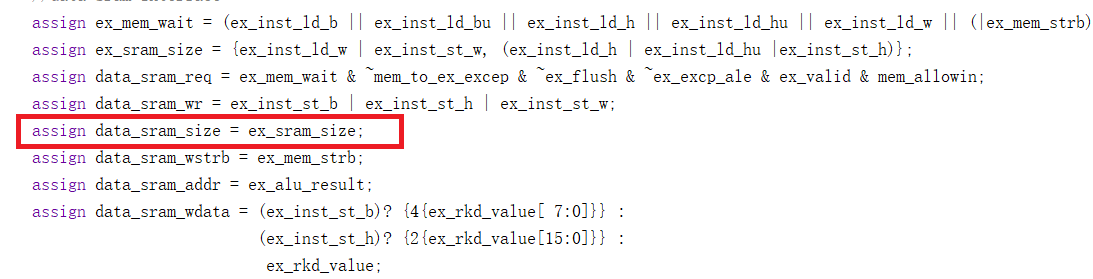


图35 错误3对应的错误代码（1）

可知出错的是EX模块中定义的ex\_sram\_size。查看这个信号赋值及定义，发现其定义位宽出错：

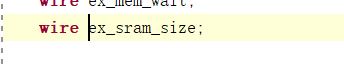


图36 错误3对应的错误代码（2）

* + - 1. 错误原因

信号data\_sram\_size的位宽为2，但是新定义了一个为其赋值的信号ex\_sram\_size的位宽为1，导致data\_sram\_size本应为2，却被赋值成了0

* + - 1. 修正效果

将ex\_sram\_size的位宽改为2，该方法有效，来到下一处bug。

* + 1. 错误4：exp14中inst\_cancel的更新逻辑有误
       1. 错误现象

运行仿真后，console报错如下：

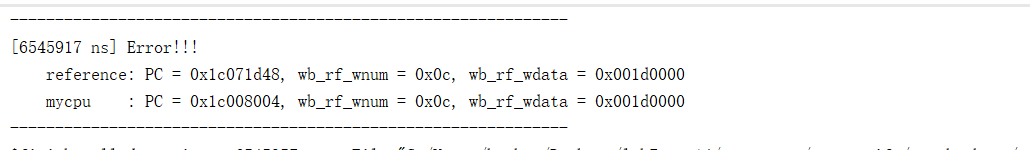


图37 错误4对应的Console报错

* + - 1. 分析定位过程

可以看出，PC发生了错误，很容易想到可能是inst\_cancel的问题，但是首先查看波形图，如下所示：



图38 错误4中断异常到来时inst\_cancel未正常拉高

可以看到，此时if\_flush信号拉高，代表进入了中断异常，此时请求握手成功但是数据还未返回，需要对取指进行取消，但是此时inst\_cancel却没有正常拉高，导致后面传给ID流水级的PC与返回的指令不对应而出错。查看inst\_cancel的的更新逻辑(此时本组还没有加inst\_cancel\_num信号)，如下所示：

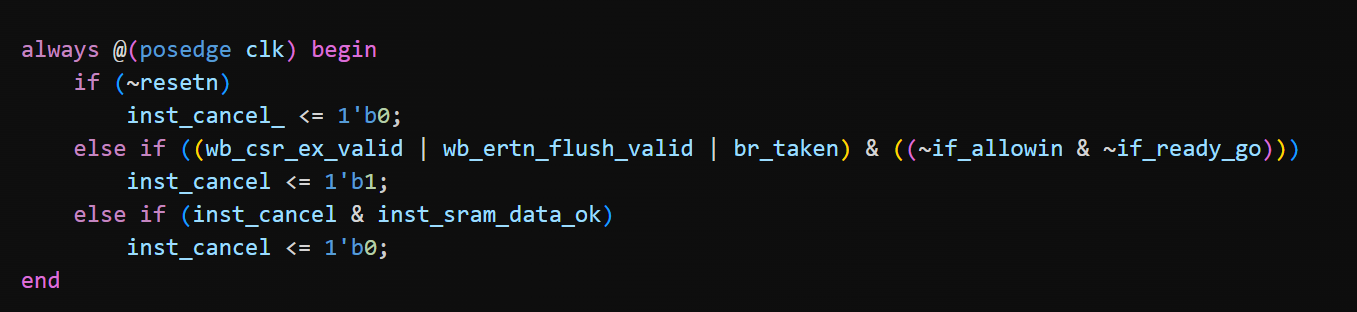


图39 错误4对应的错误代码（inst\_cancel更新逻辑）

可见此时inst\_cancel未拉高的原因是if\_allowin拉高，这是因为本组设计的是if\_allowin在if\_flush为1时拉高，代表此时IF流水级的工作已经无效，允许preIF把nextpc送过来。所以if\_allowin的逻辑是相对正确的，这时候，可以考虑到inst\_cancel之所以加上~if\_allowin，最初的本意是为了兼容if\_valid拉低的情况，所以直接把~if\_allowin改为if\_valid，便可以解决这个问题。

* + - 1. 错误原因

inst\_cancel的更新逻辑有误，判断逻辑之间存在相互冲突的地方，导致进入中断异常时，错误的取指的返回值没有被取消掉，进而导致传递给ID的PC与返回的指令不对应，出现问题。

* + - 1. 修正效果

修改inst\_cancel的更新逻辑，用if\_valid信号替换~if\_allowin，使得进入中断异常且需要取消取指的情况下，if\_cancel能正常拉高，如下所示：

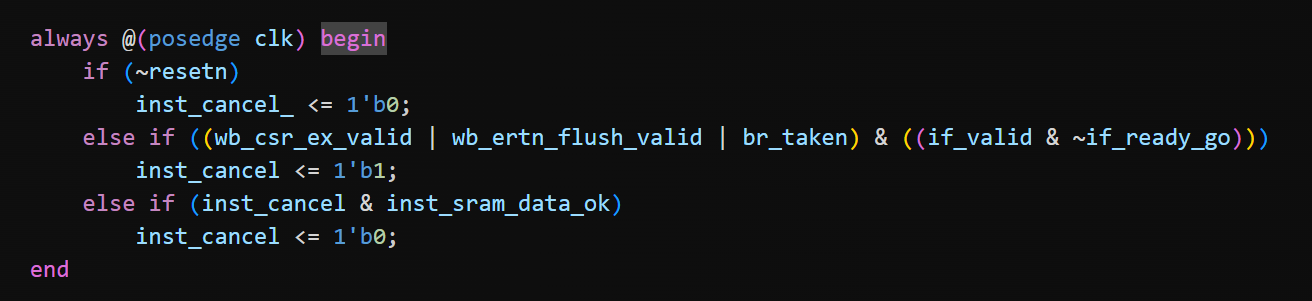


图40 错误4修改代码

该方法有效，来到下一个bug。

* + 1. 错误5：exp14中mem\_ready\_go信号逻辑出错
       1. 错误现象

运行仿真后，console报错如下：

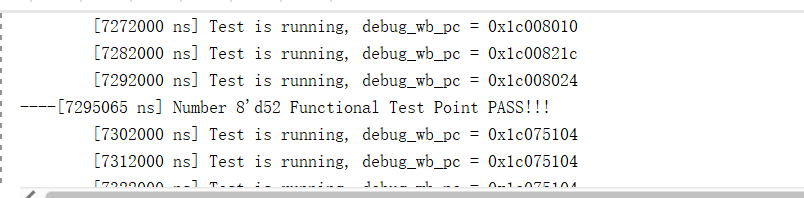


图41 错误5对应的Console报错

* + - 1. 分析定位过程

可见此时PC一直没有得到更新，导致CPU进入死循环，只能查看波形图观察是哪一个流水级的阻塞出现了问题，如下所示：

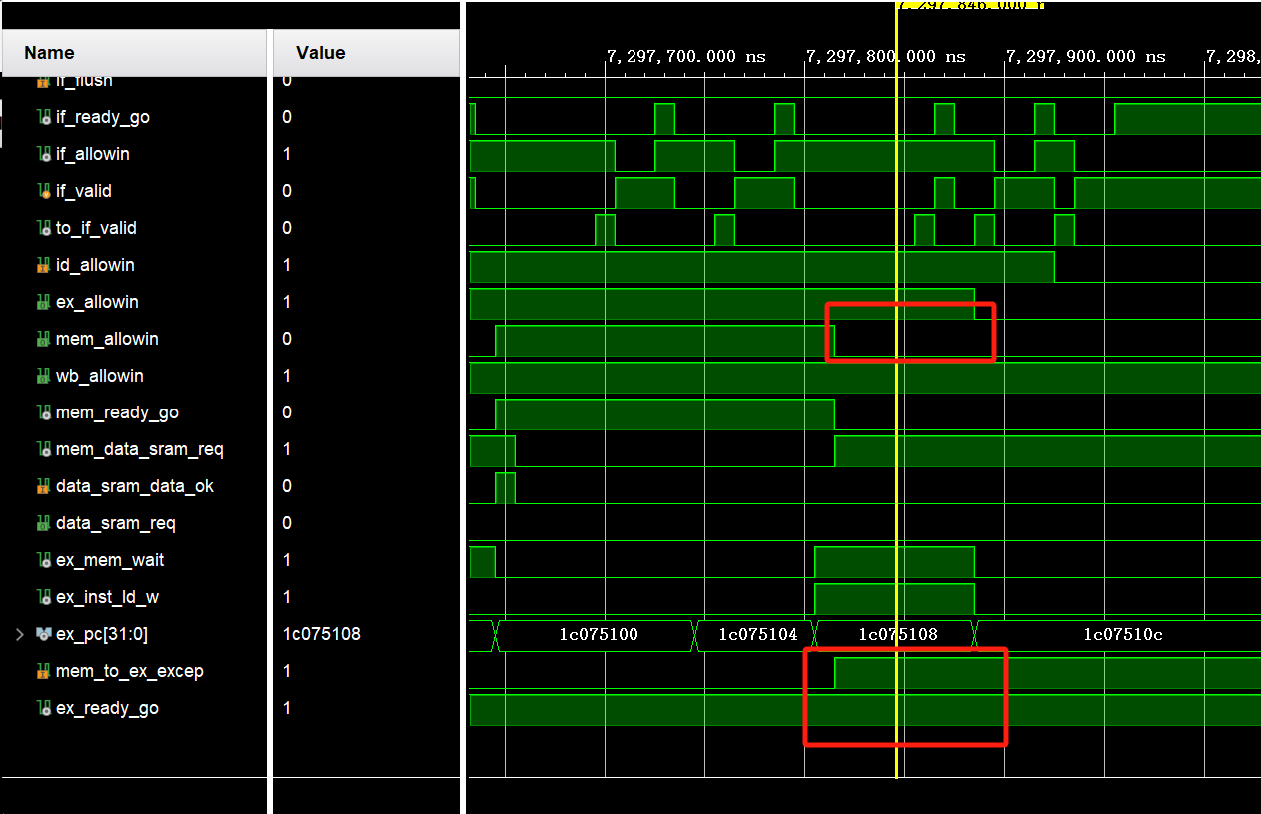


图42 错误5中MEM流水级错误阻塞

可以看到，mem\_allowin一直拉低，但ex\_ready\_go拉高，说明是MEM流水级的阻塞逻辑出现了问题，接着观察波形图，查看为啥阻塞出现问题，如下所示：

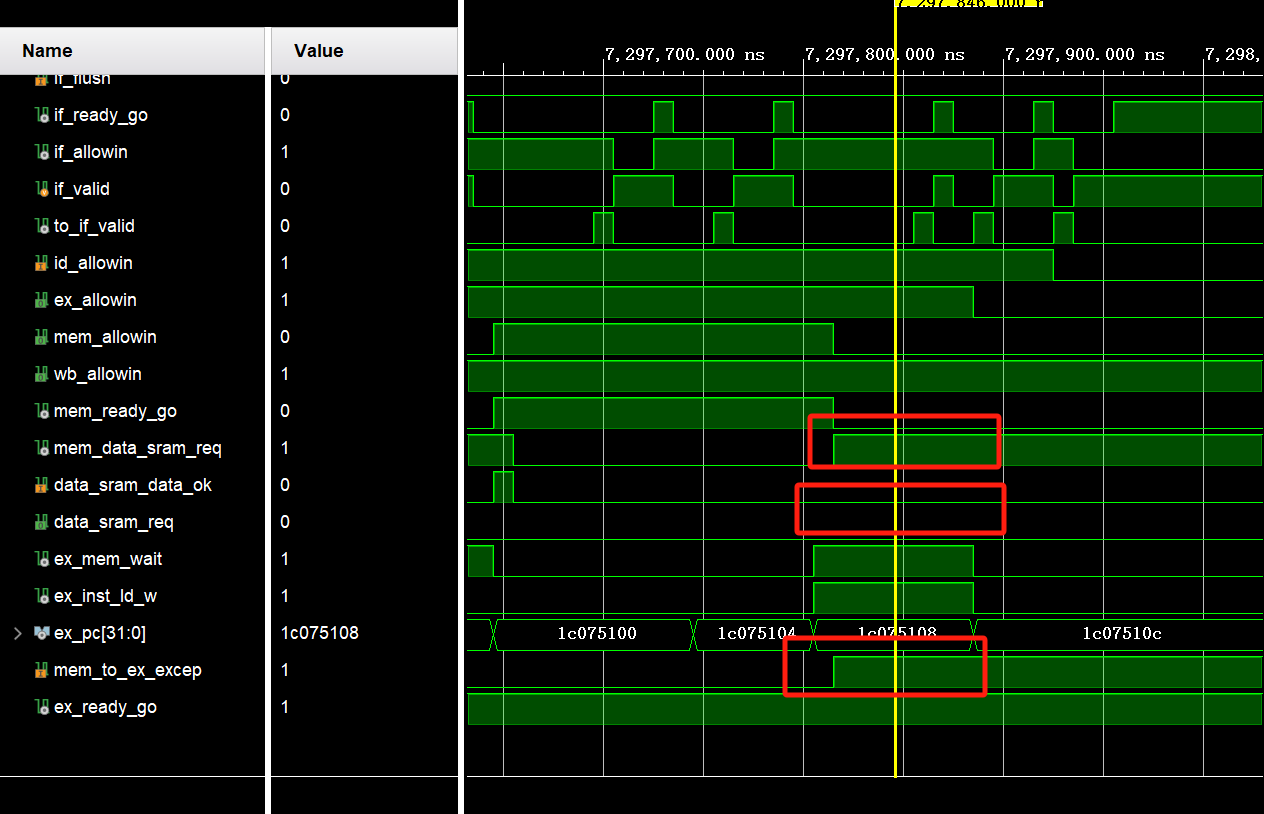


图43 异常到来EX未发请求，但MEM流水级错误阻塞等待数据返回

我们可以观察到，此时mem\_to\_ex\_excep信号拉高，代表着上一条指令发生了中断异常，所以此时EX流水级的data\_sram\_req不会发送请求，没有拉高，但是MEM从EX接受到的mem\_data\_sram\_req却拉高了，查看代码，这是因为mem\_data\_sram\_req对应的信号是EX的ex\_mem\_wait，其赋值逻辑如下：

（注：此时，本小组的信号命名还有些不规范，这也是debug环节对信号含义不断修改所导致的）

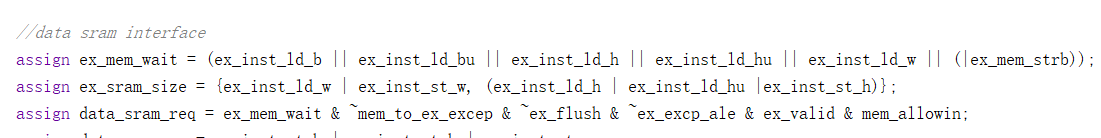


图44 错误5中ex\_mem\_wait信号未考虑中断异常

可以看到，ex\_mem\_wait没有考虑到异常中断或是流水级的刷新，所以此时EX流水级没有发req信号，但MEM错误的认为需要阻塞等待数据返回。

* + - 1. 错误原因

MEM流水级错误的阻塞，这是因为EX流水级传递给MEM流水级用于阻塞的ex\_mem\_wait信号没有考虑到异常中断或是流水级的刷新，所以此时EX流水级没有发req信号，但MEM错误的认为需要阻塞等待数据返回。

* + - 1. 修正效果

把EX流水级传递给MEM流水级用于阻塞的ex\_mem\_wait信号加上对异常中断或是流水级的刷新的考虑即可，如下所示：

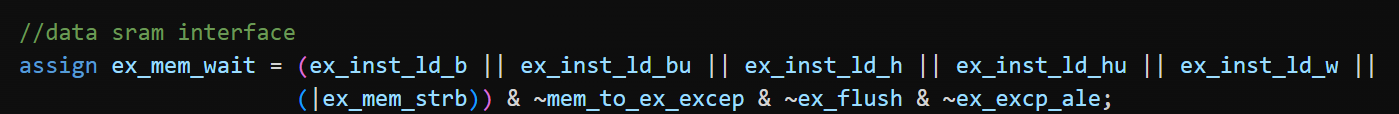


图45 错误5修改代码

该方法有效，来到下一个bug。

* + 1. 错误6：exp14中未设置inst\_cancel\_num而无法取消多条指令
       1. 错误现象

在解决前面的错误后，此时仿真已经能够通过了，但是上板后，修改种子进行测试时，发现有的种子无法通过，于是修改为对应的种子，再次进行仿真，console报错信息如下：

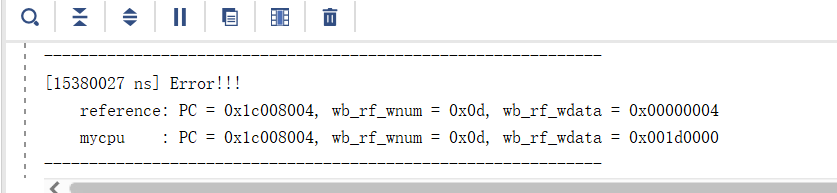


图46 错误6对应的Console报错

* + - 1. 分析定位过程

可以看到此时rf\_wdata出错，鉴于之前的经验，可能是指令取消出错，导致传递给ID的PC与指令不对应，查看波形图如下所示：

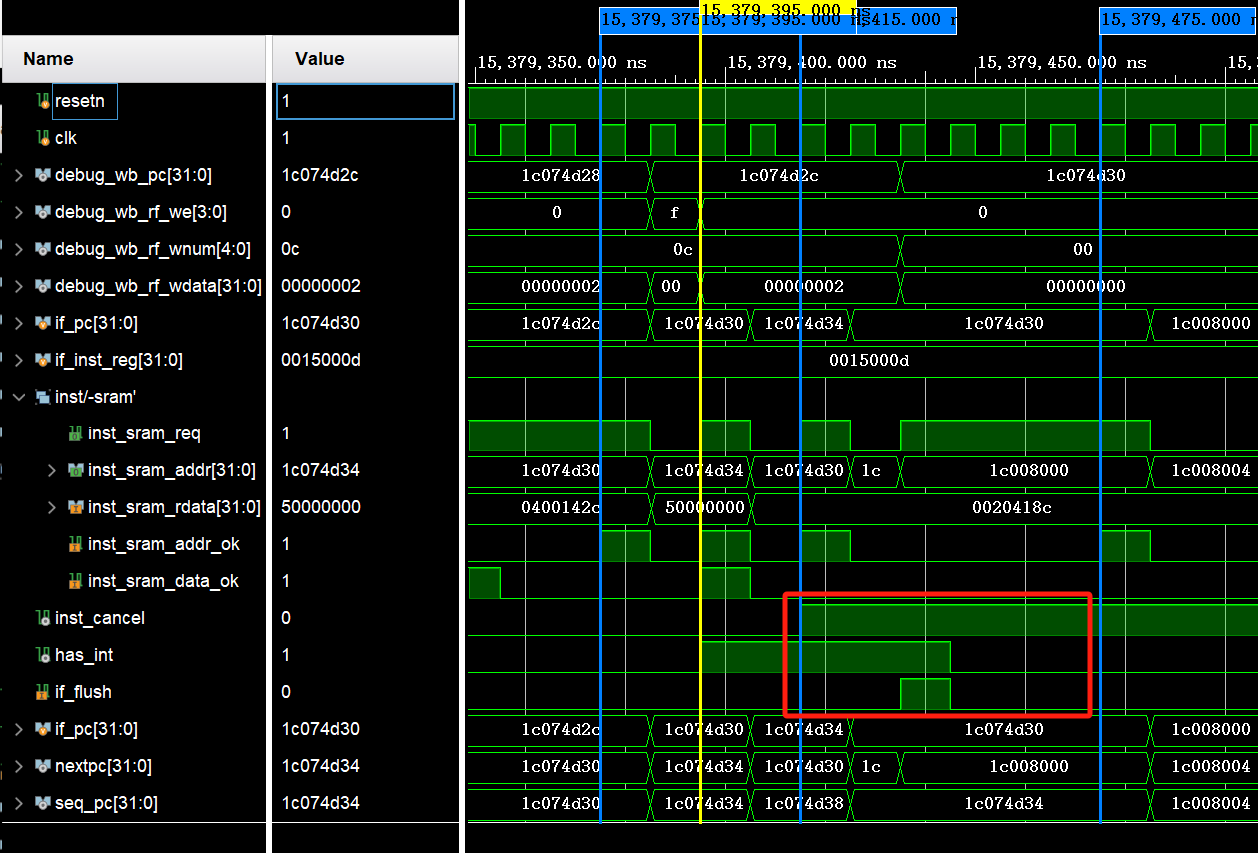


图47 错误6 中inst\_cancel无法取消多个指令返回

在仔细对应data\_ok与addr\_ok的关系后，发现果然是指令取消出现了问题，如上图所示，在if\_flush信号拉高时，inst\_cancel已经拉高了。这时inst\_cancel拉高的原因是前面出现了br\_taken。所以从逻辑上来说，此时有两个指令返回需要取消，但由于inst\_cancel只是一个1位寄存器信号，其一次最多只能取消一条指令的返回，所以才会导致后面指令没有被正确取消，进而导致传递给ID的PC和指令不对应。

* + - 1. 错误原因

未设置inst\_cancel\_num信号来取消多条指令，此时inst\_cancel只是一个1位寄存器信号，其一次最多只能取消一条指令的返回，导致后面指令没有被正确取消，传递给ID的PC和指令不对应。

* + - 1. 修正效果

新增inst\_cancel\_num寄存器，用来存储需要取消指令的数量，inst\_cancel信号则根据inst\_cancel\_num生成，代码修改如下：

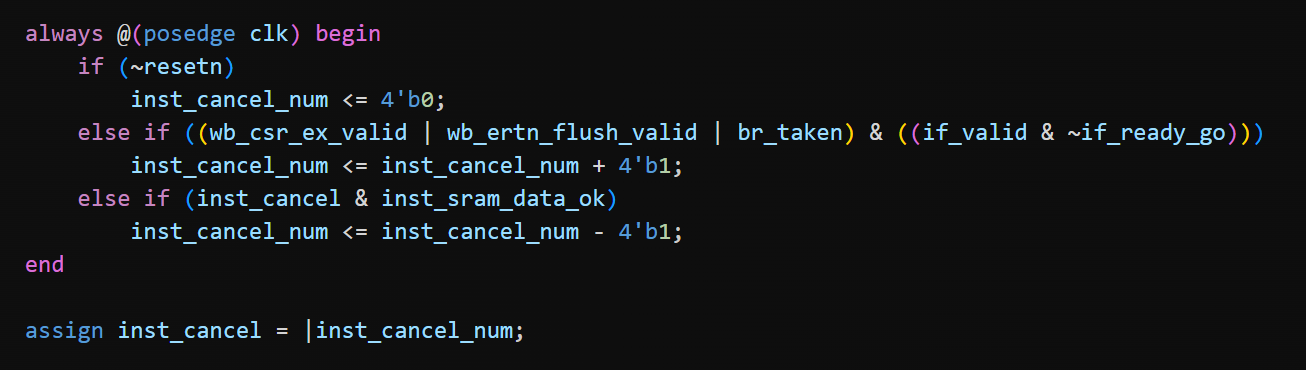


图48 错误6修改后代码(新增inst\_cancel\_num)

该方法有效，exp14仿真和上板均通过。

* + 1. 错误7：exp15中读响应状态机状态转移条件出错
       1. 错误现象

传递的PC值一直是0xxxxx\_xxxx

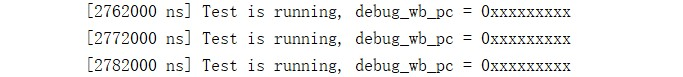


图49 错误7对应的Console界面

* + - 1. 分析定位过程

查看波形图可知第一条指令一直卡在了IF阶段，因此查看AXI转接桥的工作状态：

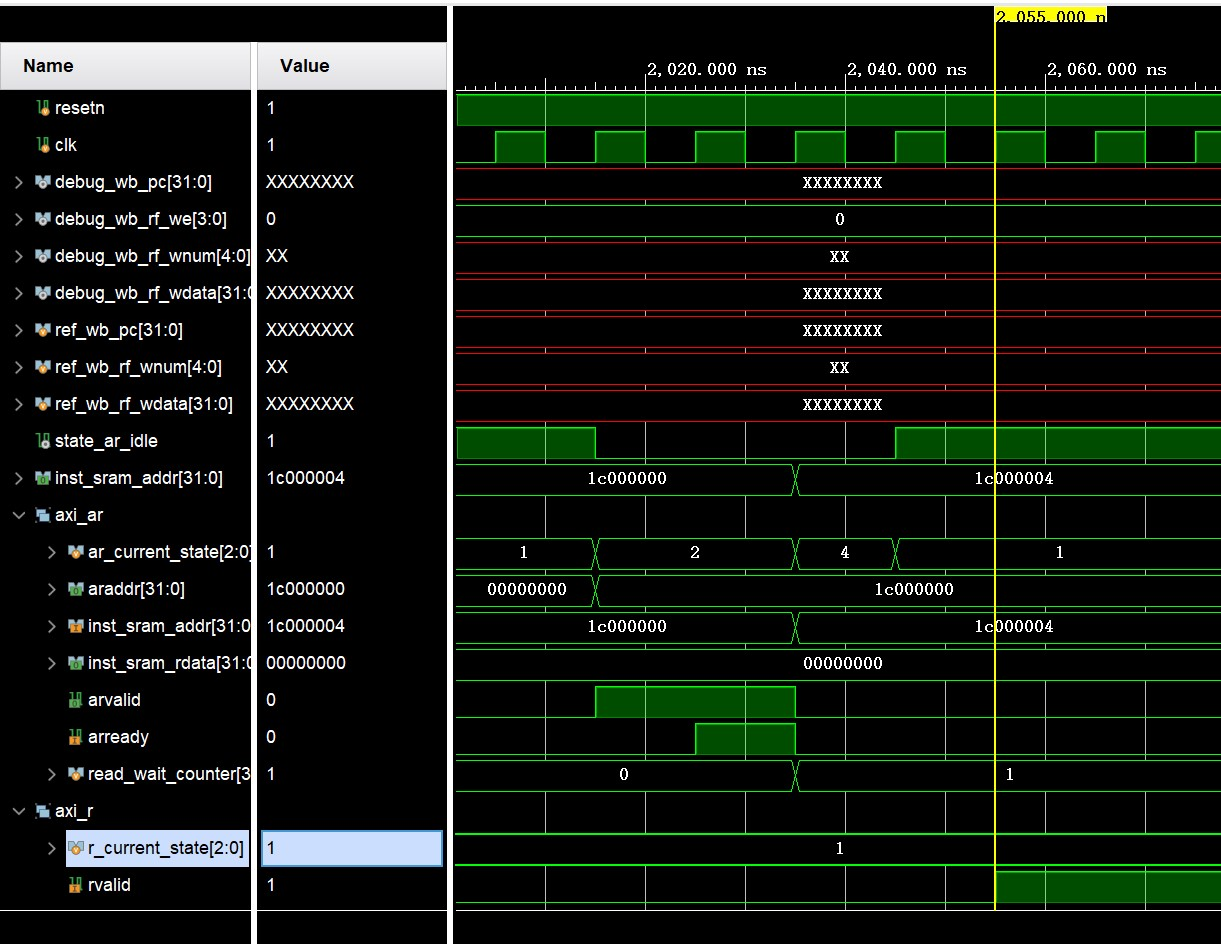


图50 错误7对应读请求通道和读响应通道的波形图

通过波形图可以看出读请求状态机正常工作了，但是读响应状态机一直停在IDLE状态。因此查看读响应状态机的状态转移逻辑：

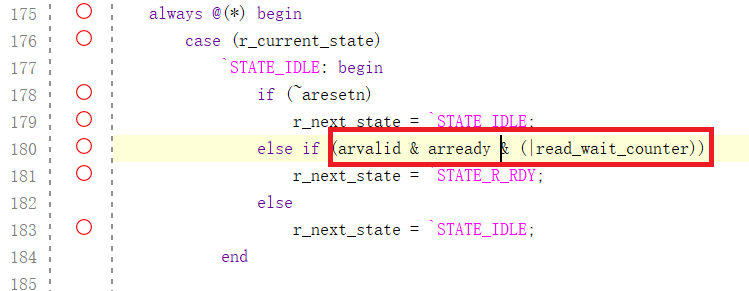


图51 错误7读响应状态机状态转移条件

* + - 1. 错误原因

读响应状态机从IDLE状态转移到R\_RDY状态的条件应是读请求地址握手成功**或**有读请求地址已握手成功但还未响应的读操作。因此状态转移条件应是arvalid & arready | (|read\_wait\_counter)。另外，原代码不能正常工作的原因还有read\_wait\_counter是一个寄存器，其会在arvalid和arready同时拉高的下一个周期加1，因此这个条件一直不会满足。

* + - 1. 修正效果

将读响应状态机在IDLE状态的状态转移机制修改如下：

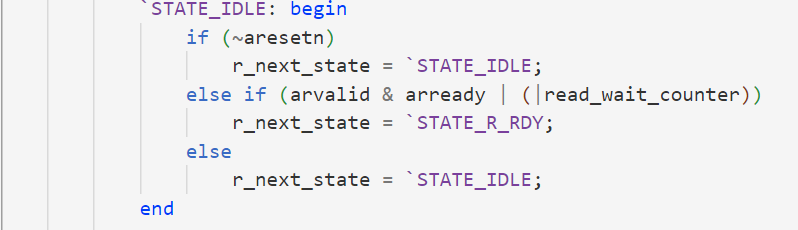


图52 错误7修改代码

该方法有效，来到下一个bug。

* + 1. 错误8：exp15中跳转指令后一条指令未取消
       1. 错误现象

Console报错如下：

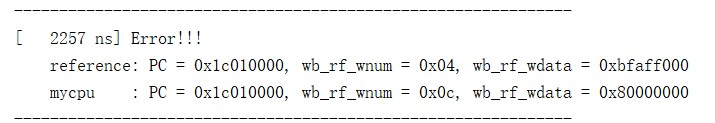


图53 错误8对应的Console报错

* + - 1. 分析定位过程

查看相关汇编代码得知错误出现在一条跳转指令后，相关波形图如下：

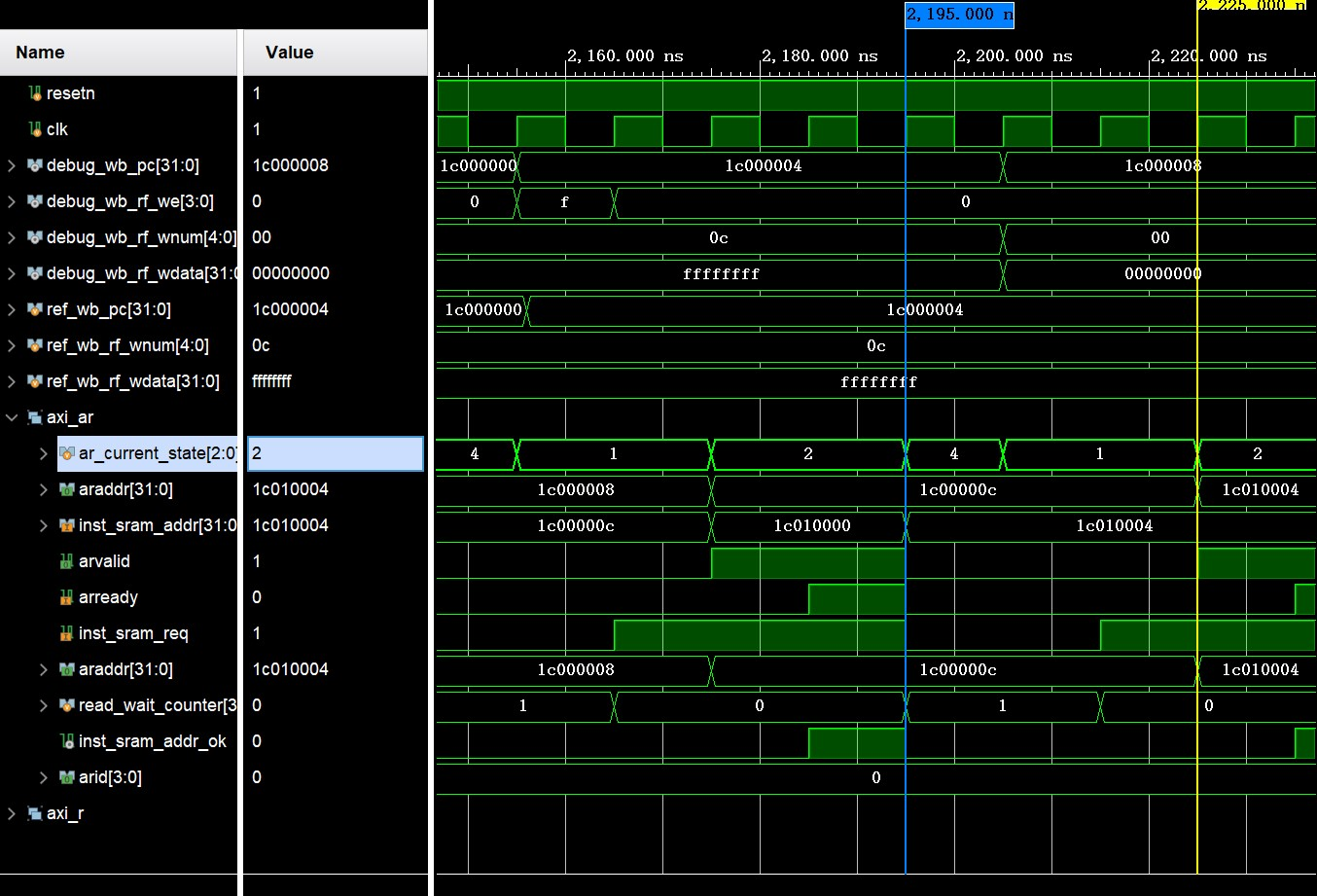


图54 错误8对应的波形图

可以发现CPU提交了对跳转指令下一条指令对应地址的请求，且此时读请求通道处于空闲状态，其接受了请求。由于跳转指令，inst\_sram\_addr发生了改变，但没有提交给读请求通道。

* + - 1. 错误原因

在程序中，0x1c000008是一个跳转到0x1c010000的指令，在这条指令译码时，它的下一条指令会发送对指令地址的请求，由于读请求通道此时处于空闲状态，会接受请求。而由于AXI总线“发送请求后不能更改请求地址”的特性，由于跳转变化的inst\_sram\_addr没有发送请求。当AXI总线返回addr\_ok时，实质上是对0x1c00000c这个地址的确认，而CPU误认为是对0x1c010000这个地址的确认，导致出错。

* + - 1. 修正效果

对跳转等情况加以讨论。若此时出现跳转、异常、ertn指令，且此时有指令已经发出了地址请求，则需要取消这条指令。在exp14中可以在发送请求的过程中更改请求地址，因此没有考虑指令在pre-IF阶段的取消，现在需要加上这种情况，修改代码如下：

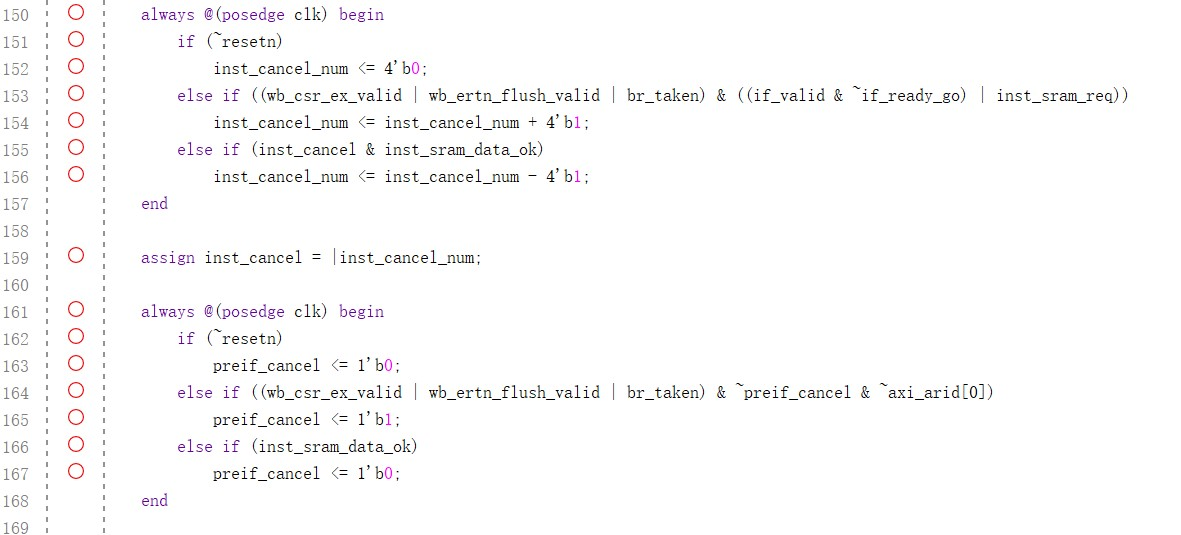


图55 错误8修改代码：增加pre-IF信号



图56 错误8修改代码：修改to\_if\_valid信号复制条件

该方法有效，来到下一处bug。

* + 1. 错误9：exp15中写响应状态机状态转移
       1. 错误现象

程序不会停下，Console界面如下：

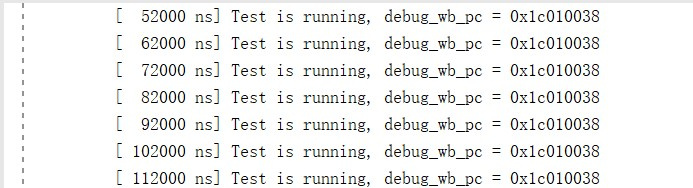


图57 错误9对应的Console界面

* + - 1. 分析定位过程

查看汇编代码，得知出错的指令是store指令。首先查看处理这条store指令时读请求和读数据通道、读响应通道状态机工作的波形图：

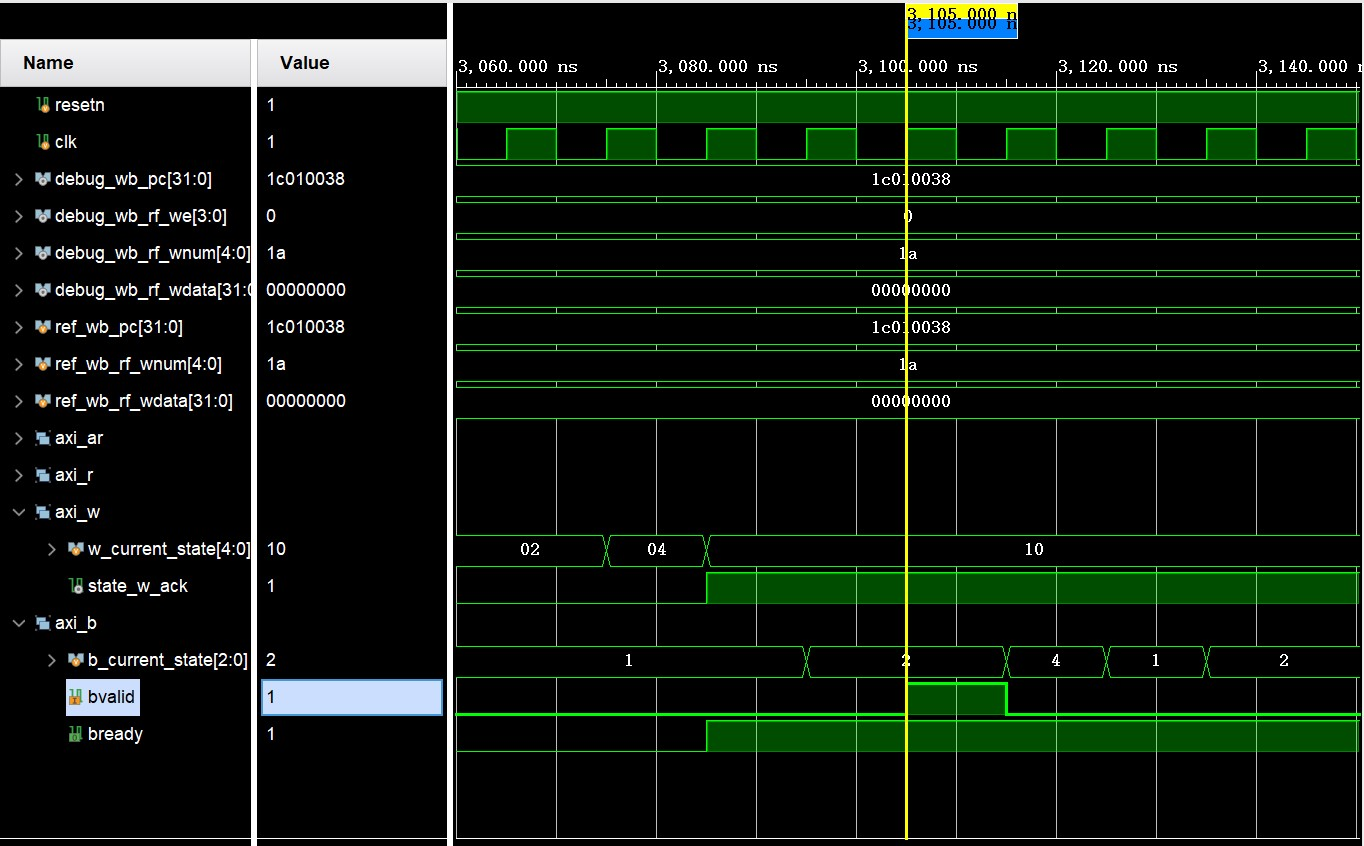


图58 错误9对应写请求和写数据通道、写响应通道状态机的波形图

可以发现写请求和写数据通道状态机一直卡在了W\_ACK状态，查看相关代码：

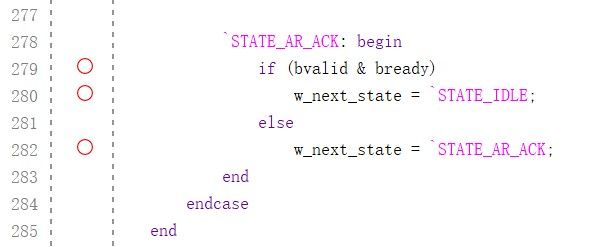


图59 错误9对应写请求和写数据通道状态机状态转移代码

* + - 1. 错误原因

可以发现在写请求和写数据状态机中，我们对状态判断出错：与写请求和写数据状态机对应的应当是W\_ACK，而不是AR\_ACK

* + - 1. 修正效果

将图中代码的AR\_ACK改为W\_ACK，修改如下：

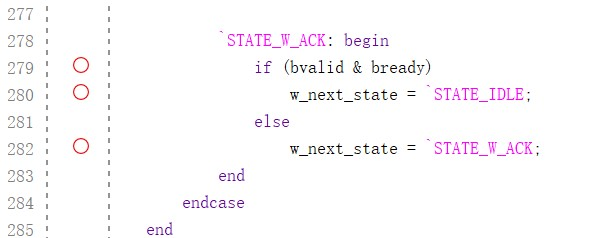


图60 错误9修改后代码

此方法有效，exp15通过。

* + 1. 错误10：exp16中preif\_cancel与inst\_cancel信号的复位逻辑出错
       1. 错误现象

在exp15通过后，上板发现只有无延迟的种子能过通过，其它种子均会出错，于是修改种子，再次运行仿真后，console报错如下所示：

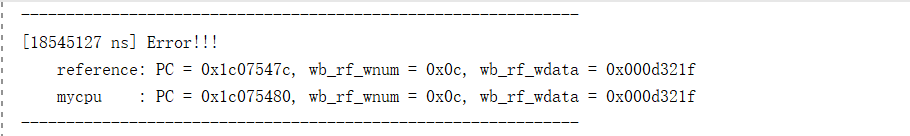


图 61 错误10对应的Console报错

* + - 1. 分析定位过程

可以看到此时PC出错，鉴于之前的经验，可能是指令取消出错，导致传递给ID的PC与指令不对应，查看波形图如下所示，具体波形图如下所示：

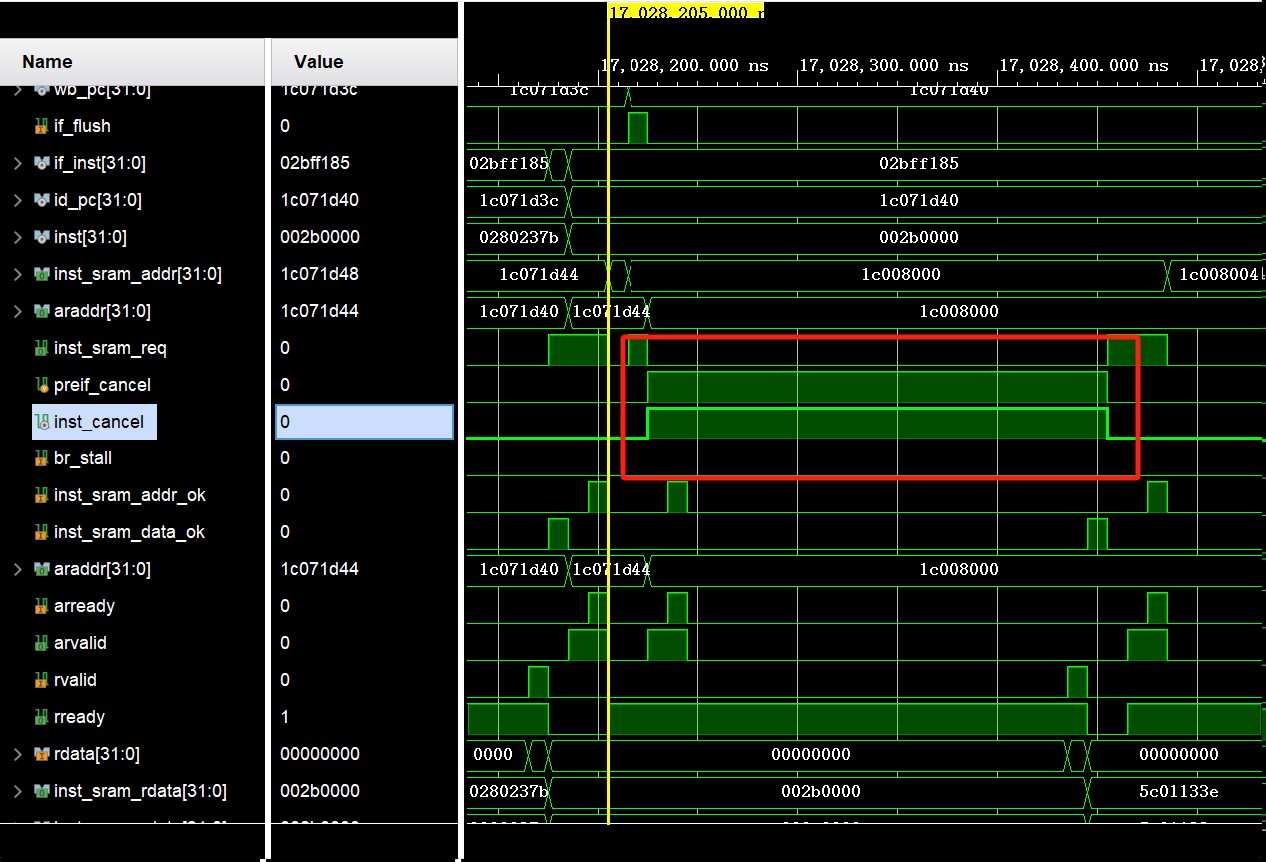


图62 错误10中preif\_cancel与inst\_cancel同时复位

可以看到，这里preif\_cancel与inst\_cancel信号同时复位为0，导致指令没有被正确取消，查看对应的代码，发现其均是在data\_ok返回后更新为0，所以会导致同时复位为0，出错前代码如下：

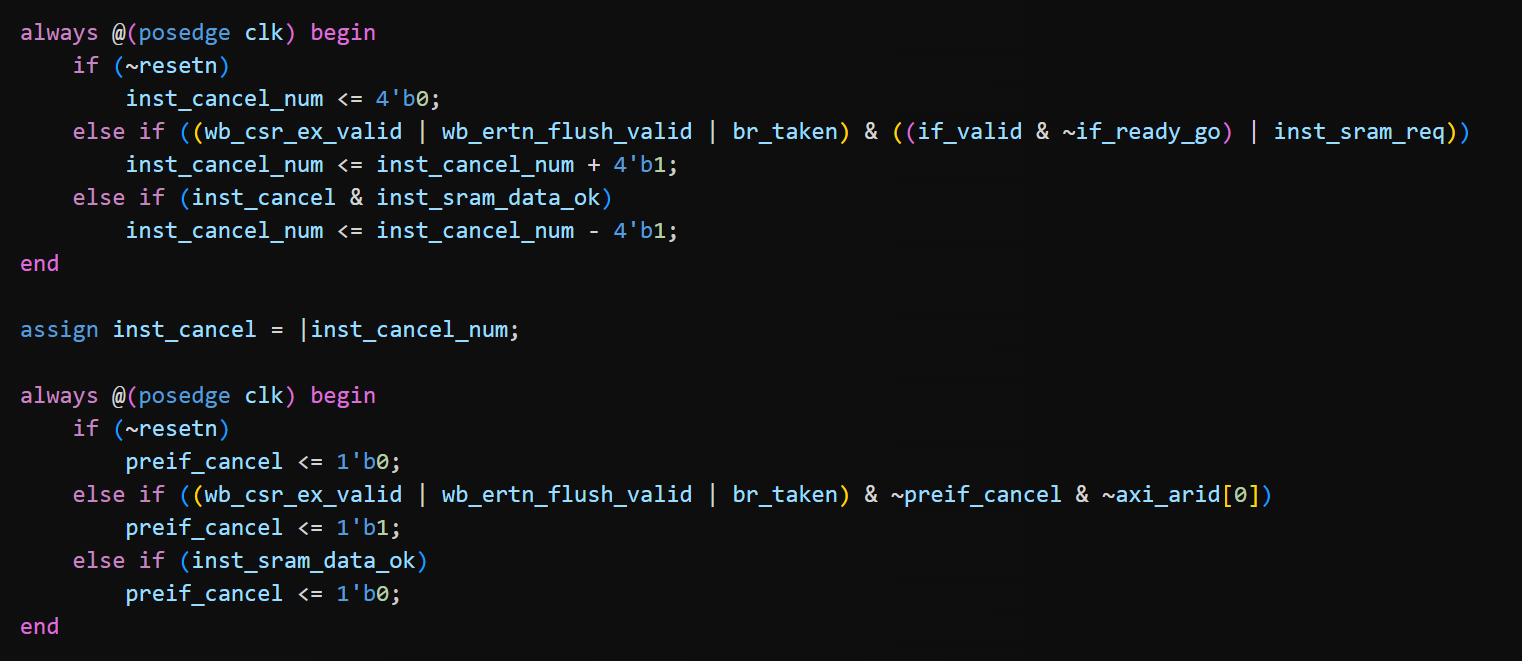


图63 错误10preif\_cancel与inst\_cancel修改前复位逻辑

* + - 1. 错误原因

preif\_cancel与inst\_cancel复位没有考虑到彼此，从而导致data\_ok来的时候，这两个信号同时复位为0，导致指令没有被正确取消。

* + - 1. 修正效果

对preif\_cancel信号的复位逻辑进行修改即可，修改后代码如下：

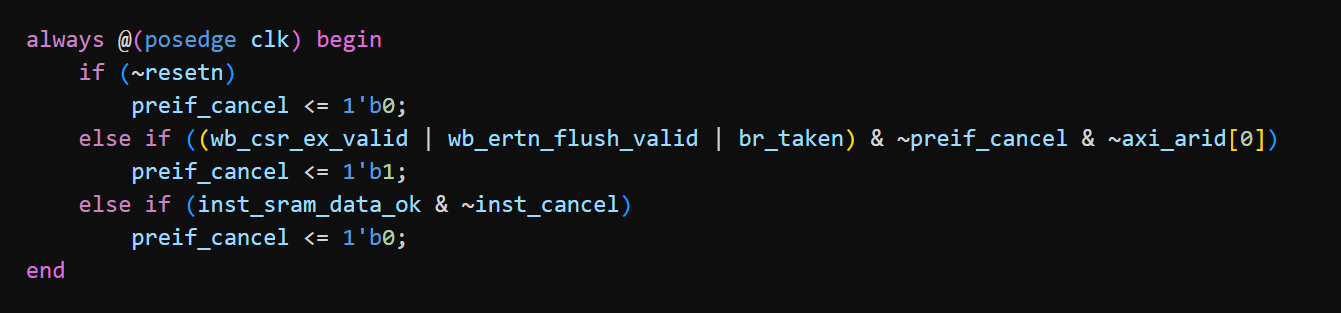


图64 错误4改后代码

这样修改后产生了新的错误，并没有让CPU运行到更后面的测试点，但这是exp15设计中的重大错误之一。

* + 1. 错误11：exp16中preif\_cancel和inst\_cancel的功能区分不清楚
       1. 错误现象

在修改了上一个错误后，console报错如下：

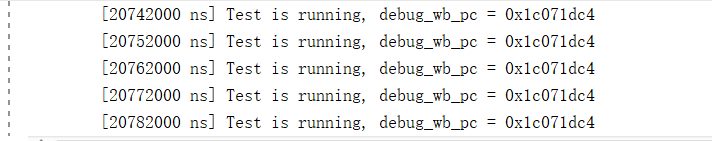


图65 错误11对应的Console报错

* + - 1. 分析定位过程

可见CPU陷入了停滞，于是只能查看波形图，观察是什么地方出现了错误，波形图如下所示：

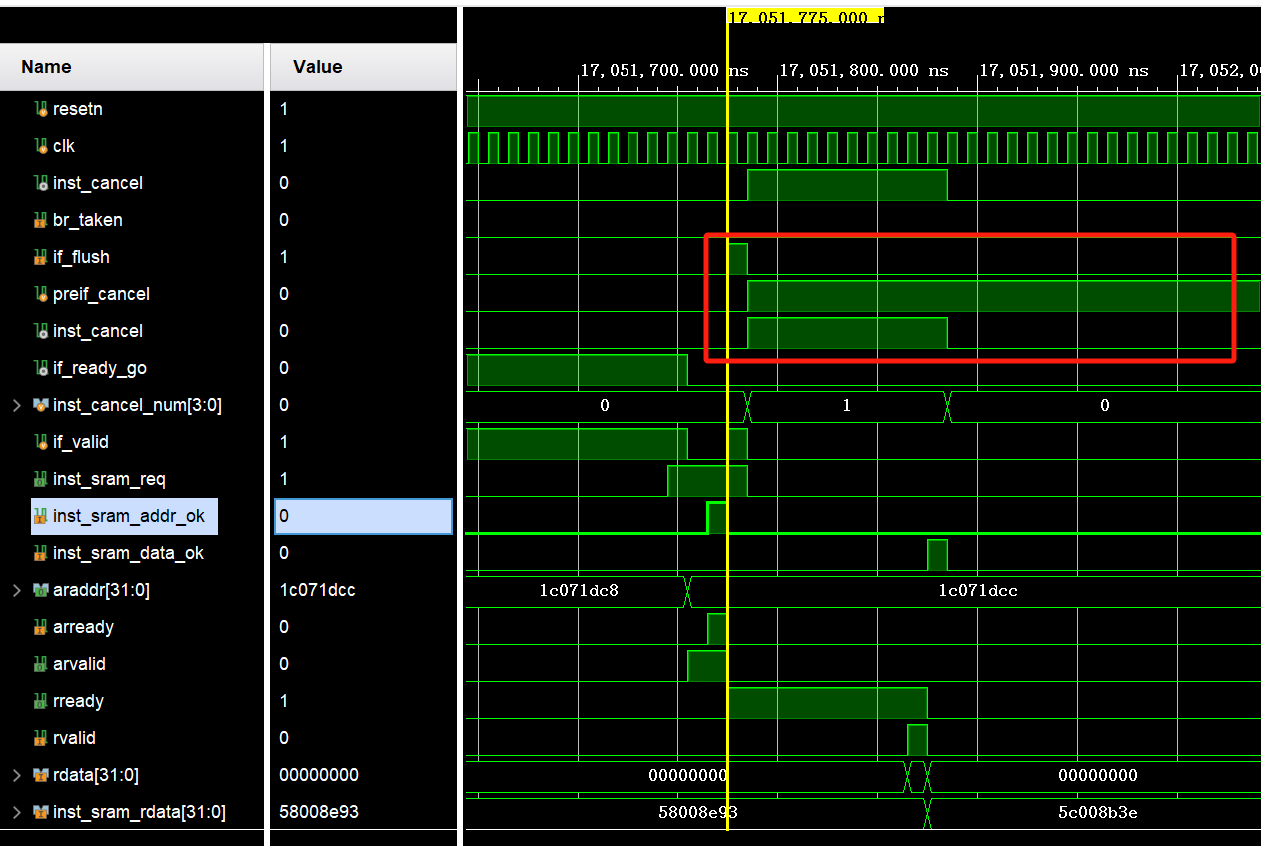


图66 错误11 的preif\_cancel与inst\_cancel拉高异常

可以看到，在if\_flush到来后，两个cancel信号同时拉高，但此时只需要取消一次取指。而由于修改了上一个bug，所以此时只有一个cancel信号拉低，从而导致出错。

到这可以说明，本小组exp15的通过有很大的偶然因素，其中很多地方尤其是cancel信号的更新复位逻辑并未搞的很清楚。查看preif\_cancel和inst\_cancel的逻辑如下：

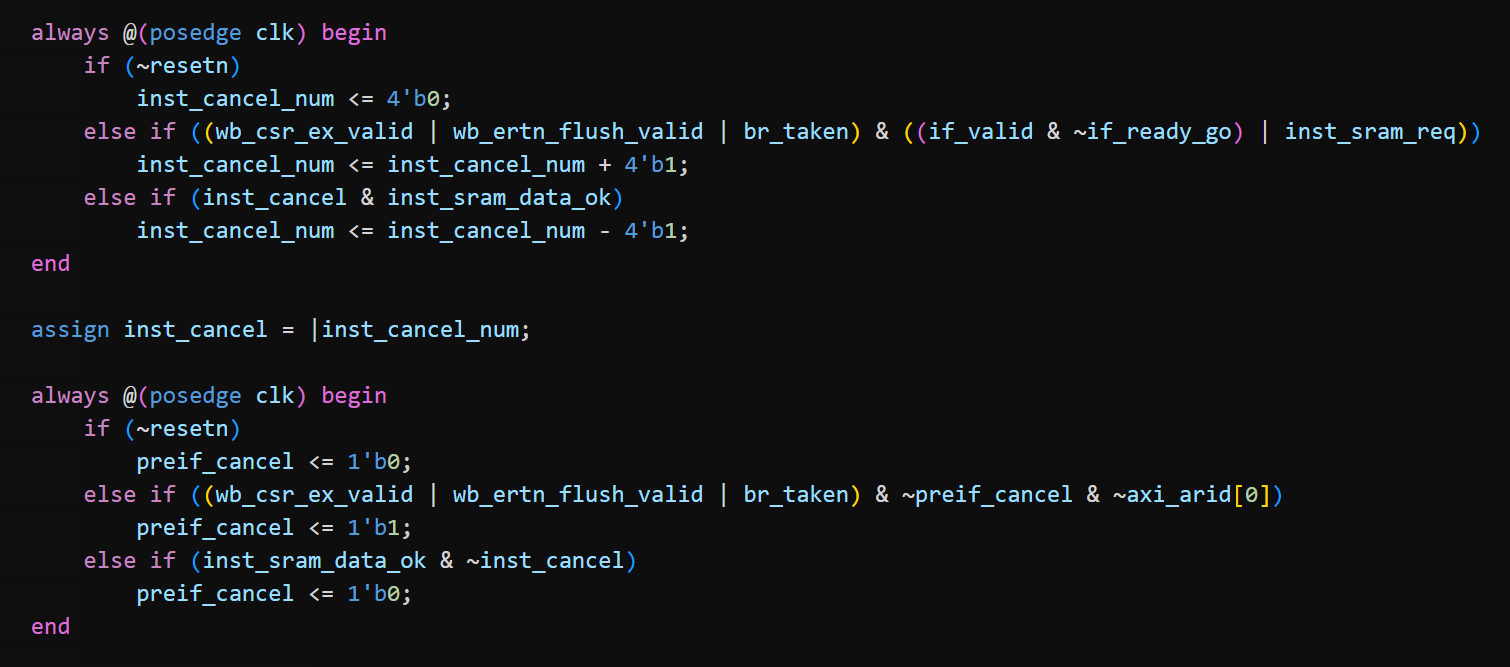


图67 错误11 的inst\_cancel与preif\_cancel修改前更新逻辑

可以看到，设计这两个信号的本意是区分不同情况下的取消指令操作。但是在该逻辑下，inst\_cancel\_num加一的逻辑中加上了“| inst\_sram\_req”，这个本来是preif\_cancel拉高的判断逻辑的一部分，所以这样写导致了在很多本应该只用preif\_cancel拉高的情况下inst\_cancel也拉高了。除此之外，preif\_cancel信号的更新逻辑也极其不完善，首先其根本没有考虑inst\_sram\_req，没有区分当跳转分支异常，中断等信号到来时，请求是否已经握手完成，导致其与inst\_cancel也混淆在一起。

出现上面这种错误的原因是本小组在定好每个信号的含义后，写exp15的同学没有清楚的区分这两个信号的作用，而仅仅根据exp15仿真中出现的错误对相关的逻辑进行强硬的修改，导致最终偶然条件下才能通过无延迟的种子，其它种子均无法通过。

* + - 1. 错误原因

exp16中preif\_cancel和inst\_cancel的功能区分不清楚，导致这两个信号均会存在本只应该对方拉高，但自己也拉高的情况，从而导致指令取消出错，CPU陷入停滞。

* + - 1. 修正效果

最后本组中写exp16的同学对这两个信号的更新逻辑进行大幅度的修改，其中也肯定不可避免的出现了许多的bug，但是在实验报告中就不全部进行展示了。对于这两个信号的修改，其中一个阶段性的结果如下：

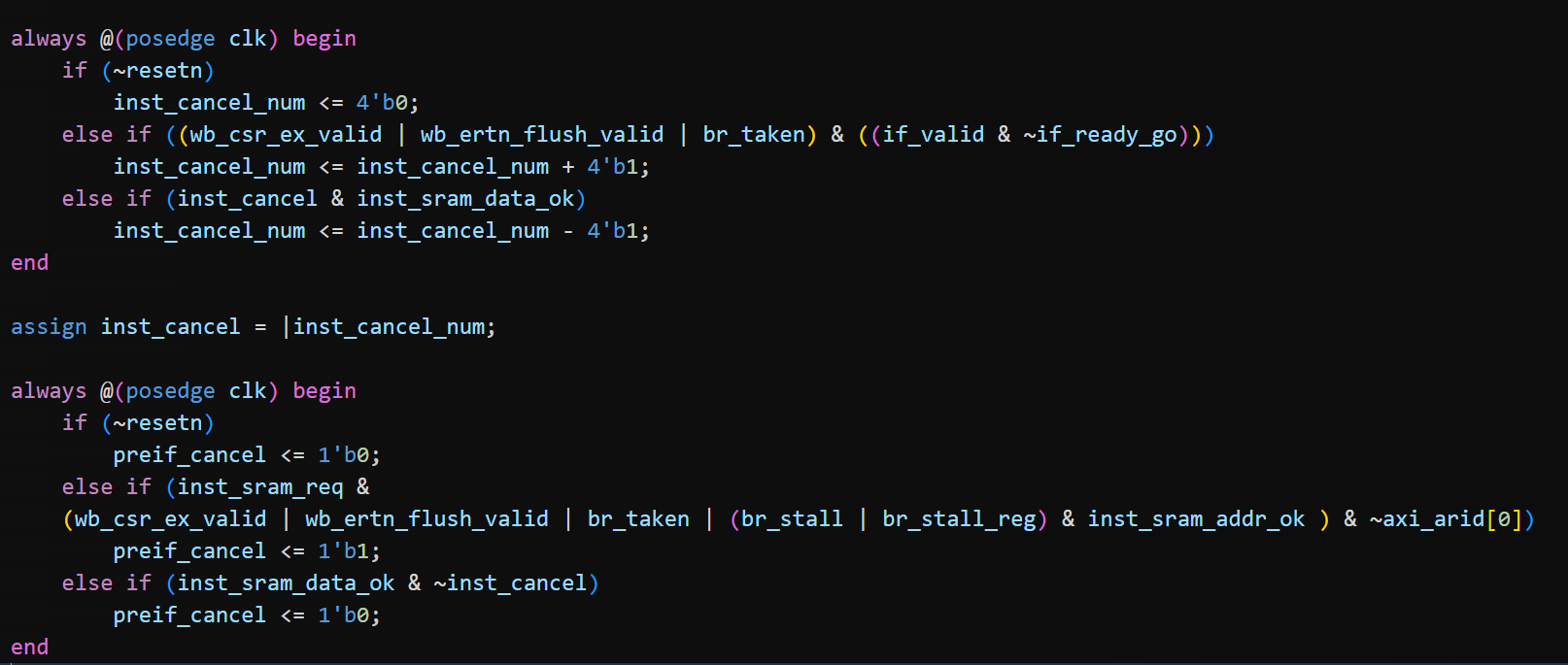


图68 错误11修改后代码

* + 1. 错误12：exp16中preif\_cancel信号未对br\_stall进行充分考虑出错
       1. 错误现象

在修改完上面的bug后，运行仿真，Console报错如下：

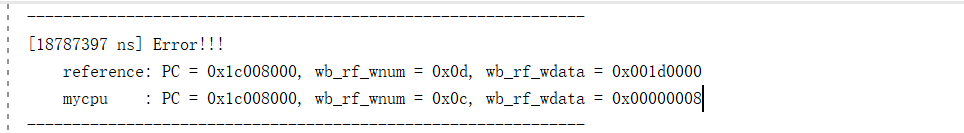


图69 错误12对应的Console报错

* + - 1. 分析定位过程

可以看到此rf\_wada出错，鉴于之前的经验，可能是指令取消出错，导致传递给ID的PC与指令不对应，查看波形图如下所示，具体波形图如下所示：：

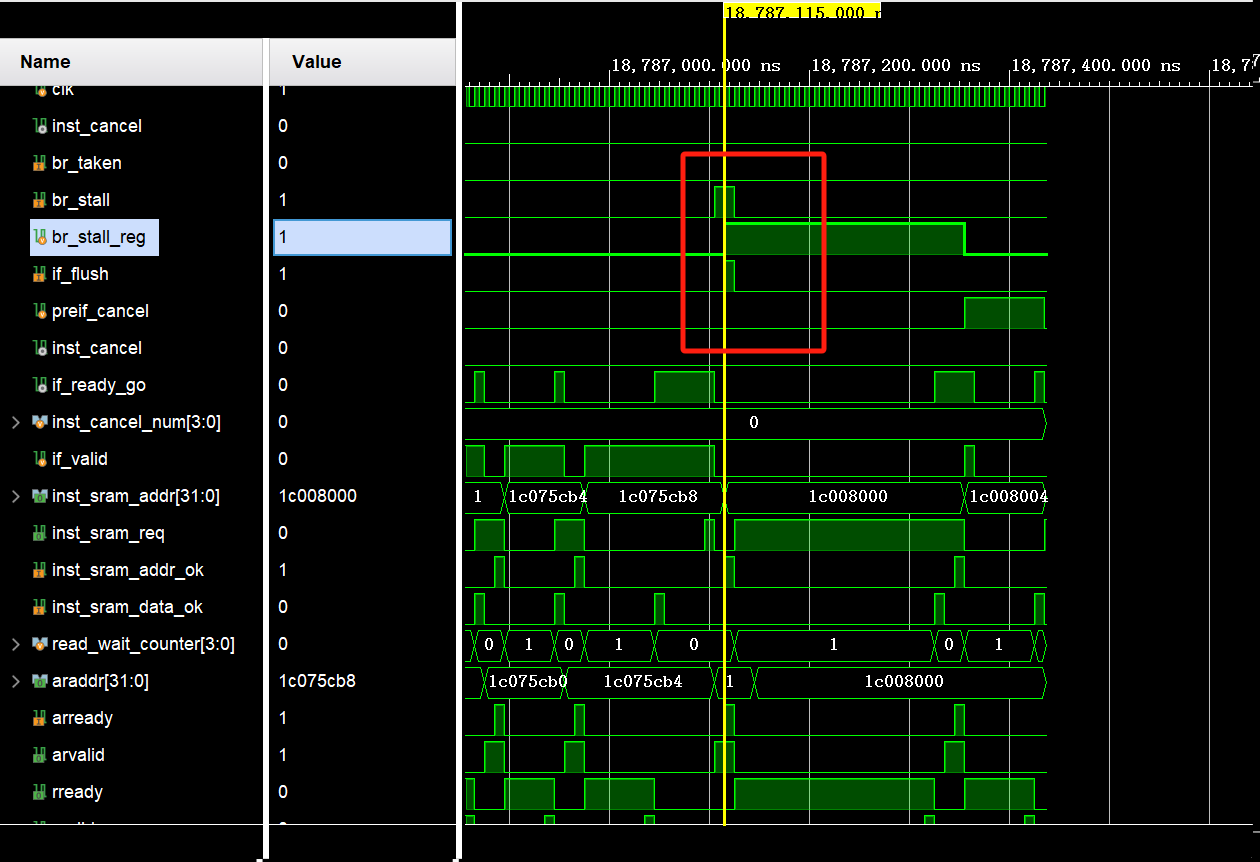


图70 错误12中因为br\_stall导致preif\_cancel未正常拉高

可以看到，此时if\_flush信号到来，且此时inst\_sram\_req发送了但addr\_ok还未返回，理论上应该preif\_cancel信号需要拉高，但是此时它没有正常拉高，查看preif\_cancel信号的更新逻辑，如下：

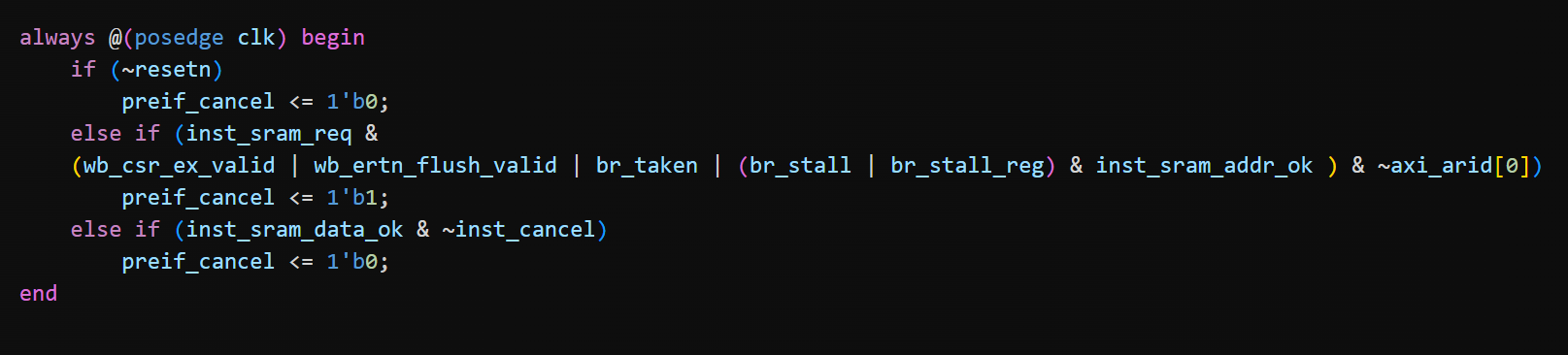
****

图71 错误12修改前preif\_cancel赋值代码(未对br\_stall充分考虑)

可以看到，此时preif\_cancel没有正常拉高的原因是在if\_flush到来时，br\_stall信号拉高了，所以此时inst\_sram\_req信号不会拉高（br\_stall时不发请求，等待ID数据冲突解决了再进行请求的发送），但是在指令进入ID级、拉高br\_stall前，它的后一条指令已经开始发送取指请求，因此不能仅仅通过在br\_stall为1时拉低req来处理br\_stall，还需要拉高pre\_cancel信号，等待先发完再取消。

* + - 1. 错误原因

preif\_cancel信号未对br\_stall进行充分考虑，导致取指没有被正确取消。在指令进入ID级、拉高br\_stall前，它的后一条指令已经开始发送取指请求，因此不能仅仅通过在br\_stall为1时拉低req来处理br\_stall，还需要拉高pre\_cancel信号，等待先发完再取消。

* + - 1. 修正效果

修改preif\_cancel的逻辑，添加对br\_stall的考虑，最终代码如下所示：

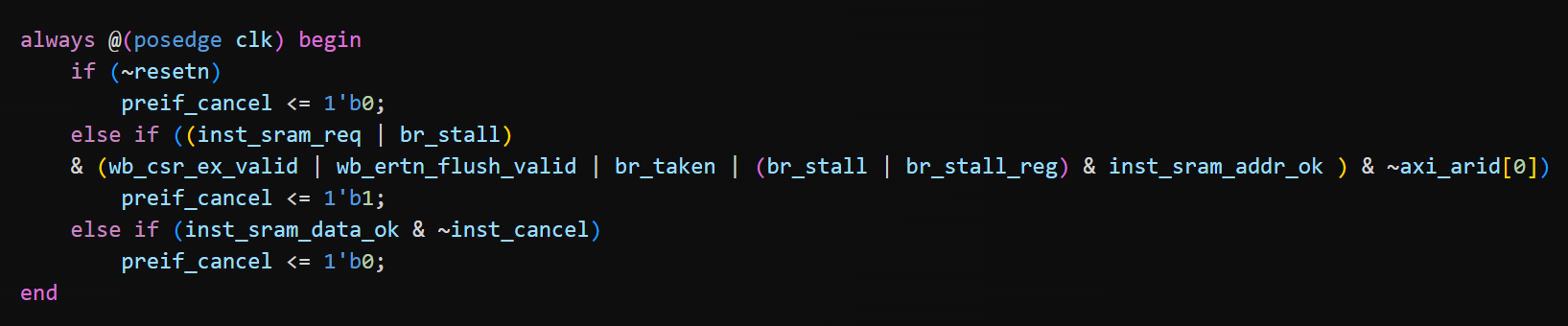


图72 错误6修改后代码

修改完这个bug后，在该种子下的仿真便能正常通过了。同时上板后，发现其它种子也能正常通过。至此，对exp16终于结束。

1. 实验总结

本实验为CPU增加了AXI总线接口，令CPU能够通过总线与系统中的内存、外设进行交互。。

类SARM总线和AXI总线的引入使得流水级变化的情形更加复杂，尤其是IF流水级，逻辑较之前的实验复杂了很多，同时随机种子也为debug增加了一些难度，exp14和exp16均出现了对应种子能过，但其它种子不能通过的情况。在debug的过程中，我们需要遍历所有可能的情况，认真分析CPU的设计，这提高了我们对流水线CPU、总线协议等各方面的认识，也积累了不少debug的经验，并且通过本次实验，本小组成员还进一步熟悉了git的使用，对如何使用git进行合作有了更深的理解。

在这三个实验中，我们的时序有了很大的改善，exp16的时序见下图。本组成员认为这是主要因为在这三次实验中，CPU对外连接的变成了AXI总线而不是原来的SRAM接口，这样可能导致组合逻辑链变得比原来更加简单。除此之外，原先的组合逻辑有不少都被换成了时序逻辑，比如原先的nextpc是由一个多路选择器在多个组合逻辑信号的基础上生成的，而现在这些信号中的一部分被换成了时序逻辑，组合逻辑链变得更加简单，时序也有了很大的提升。

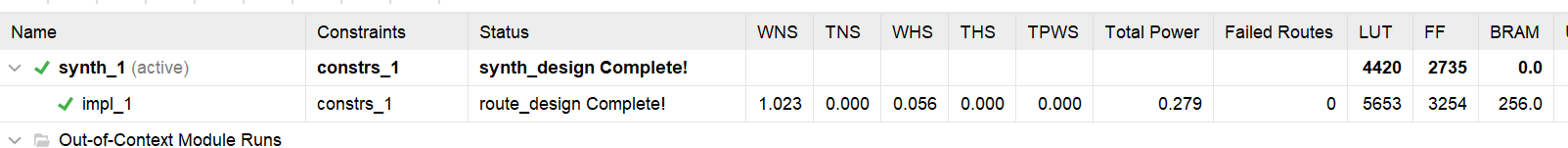


图73 exp16的时序结果