**LAB 9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

Exp20需要理解并掌握cache的组织结构和工作机理，完成cache模块的数据通路与控制逻辑设计，实现一个两路组相连， 写回写分配、采用伪随机替换算法的cache模块。并将设计的cache模块集成到实验提供的模块级验证环境中，通过仿真和上板验证。

Exp21需要将Cache模块作为ICache集成到流水线CPU中。并且需要修改CPU中的AXI转换桥，以支持Burst传输。然后在采用AXI总线的SoC验证环境里通过仿真和上板验证。

1. 实验设计
   1. 总体设计思路
      1. Exp20

Exp20实现了一个两路组相连， 写回写分配、采用伪随机替换算法的cache模块，下面将大致介绍cache的地址划分和cache模块的工作流程。

* + - 1. **规格参数**

本次实验的cache模块为两路组相连，每个cache行大小为16B，总容量8KB，经过计算可以知道cache地址的offset域位宽为4，index域位宽为8，tag域位宽为20。结构如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  | 12 | 11 |  | 4 | 3 |  | 0 |
| Tag | | | Index | | | Offset | | |

本实验的cache既需要与CPU进行交互，也需要与内存进行交互，与内存的交互通过AXI转接桥完成。Cache向内存中读数据时，每次都发送读整个cache行，也就是读16字节（4个word）的突发请求，然后AXI转接桥向cache逐个word返回读数据。Cache向内存中写数据时，每次发送整个cache行，也就是16字节的数据出去，随后真正将数据落实到内存中的过程由AXI转接桥完成，在本实验则无需考虑。

* + - 1. **工作流程**

Cache可以接收CPU流水级发来的读写请求，并在请求不命中时向内存发出读请求，将缺失的数据缓存下来。如果有cache行需要被替换出去，则需要根据脏位决定是否把这一行写回内存。无论读写，cache都需要先依据index域在自己的缓存中读出两路数据，然后比较tag位来判断是否命中。命中时，对于读操作，可以直接返回读数据；对于写操作，还需要将CPU发来的写请求写入自己的缓存，并标记上对应的D（Dirty）位。

如果缓存不命中，cache就需要向内存发读请求并把读数据缓存。存储新数据就意味着相同index位置存放的旧数据需要被替换出，因此此时需要先判断cache读出的旧数据是否Dirty，如果D位是0，代表该行缓存的数据与内存中的数据一致，不需要写回内存，否则需要向内存发送写请求，更新内存中的数据。在保证旧数据的安全之后，便可以将内存发来的读数据写入cache缓存了。

* + 1. Exp21

本实验将exp20实现的cache模块实例化，作为ICache集成到流水线cpu中。本实验中，ICache例化在IF流水级与AXI转接桥之间，所以为了实现对ICache的集成，本实验需要修改AXI转接桥的逻辑，使其能够处理burst传输。此外，由于流水线IF直接向cache发送请求，所以req拉高之后的addr可以进行改变，相应的IF流水级的部分取消指令的逻辑（与preif\_cancel相关的逻辑）也会进行删除。

* 1. 重要模块1设计：cache模块
     1. 工作原理

Cache需要支持读写操作，根据CPU发送的读写请求的tag和index在Cache内部查找，如果命中则直接读写，并返回数据或ok信号；如果没命中则从内存中读出缺失的数据，根据随机算法替换一个Cache行。

* + 1. 接口定义

表1 TLB模块接口信号

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| **Search port 0** | | | |
| s0\_vppn | IN | 19 | 查找虚双页号 |
| s0\_va\_bit12 | IN | 1 | 访存虚地址12位 |
| s0\_asid | IN | 10 | 地址空间标识 |
| s0\_found | OUT | 1 | 查找到信号 |
| s0\_index | OUT | 4 | 页表项索引 |
| s0\_ppn | OUT | 20 | 物理页号 |
| s0\_ps | OUT | 6 | 页大小 |
| s0\_plv | OUT | 2 | 权限等级 |
| s0\_mat | OUT | 2 | 访存种类 |
| s0\_d | OUT | 1 | 脏页信号 |
| s0\_v | OUT | 1 | 页有效信号 |
| **Search port 1** | | | |
| s1\_vppn | IN | 19 | 查找虚双页号 |
| s1\_va\_bit12 | IN | 1 | 访存虚地址12位 |
| s1\_asid | IN | 10 | 地址空间标识 |
| s1\_found | OUT | 1 | 查找到信号 |
| s1\_index | OUT | 4 | 页表项索引 |
| s1\_ppn | OUT | 20 | 物理页号 |
| s1\_ps | OUT | 6 | 页大小 |
| s1\_plv | OUT | 2 | 权限等级 |
| s1\_mat | OUT | 2 | 访存种类 |
| s1\_d | OUT | 1 | 脏页信号 |
| s1\_v | OUT | 1 | 页有效信号 |
| **INVTLB opcode** | | | |
| invtlb\_valid | IN | 1 | 无效指令有效信号 |
| invtlb\_op | IN | 5 | 无效指令操作码 |
| **Write port** | | | |
| we | IN | 1 | 写使能信号 |
| w\_index | IN | 4 | 写页表项索引 |
| w\_e | IN | 0 | 写页表项有效信号 |
| w\_vppn | IN | 19 | 写虚拟页号 |
| w\_ps | IN | 6 | 写页大小 |
| w\_asid | IN | 10 | 写进程号 |
| w\_g | IN | 1 | 写全局信号 |
| w\_ppn0 | IN | 20 | 写物理页号0 |
| w\_plv0 | IN | 2 | 写权限等级0 |
| w\_mat0 | IN | 2 | 写访存种类0 |
| w\_d0 | IN | 1 | 写脏页信号0 |
| w\_v0 | IN | 1 | 写页有效信号0 |
| w\_ppn1 | IN | 20 | 写物理页号1 |
| w\_plv1 | IN | 2 | 写权限等级1 |
| w\_mat1 | IN | 2 | 写访存种类1 |
| w\_d1 | IN | 1 | 写脏页信号1 |
| w\_v1 | IN | 1 | 写页有效信号1 |
| **Read port** | | | |
| r\_index | IN | 4 | 读页表项索引 |
| r\_e | OUT | 0 | 读页表项有效信号 |
| r\_vppn | OUT | 19 | 读虚拟页号 |
| r\_ps | OUT | 6 | 读页大小 |
| r\_asid | OUT | 10 | 读进程号 |
| r\_g | OUT | 1 | 读全局信号 |
| r\_ppn0 | OUT | 20 | 读物理页号0 |
| r\_plv0 | OUT | 2 | 读权限等级0 |
| r\_mat0 | OUT | 2 | 读访存种类0 |
| r\_d0 | OUT | 1 | 读脏页信号0 |
| r\_v0 | OUT | 1 | 读页有效信号0 |
| r\_ppn1 | OUT | 20 | 读物理页号1 |
| r\_plv1 | OUT | 2 | 读权限等级1 |
| r\_mat1 | OUT | 2 | 读访存种类1 |
| r\_d1 | OUT | 1 | 读脏页信号1 |
| r\_v1 | OUT | 1 | 读页有效信号1 |

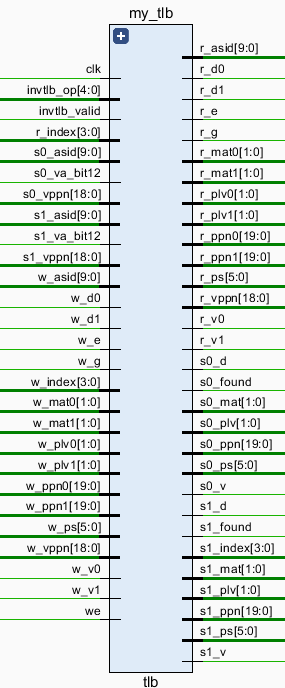


图8 TLB模块接口

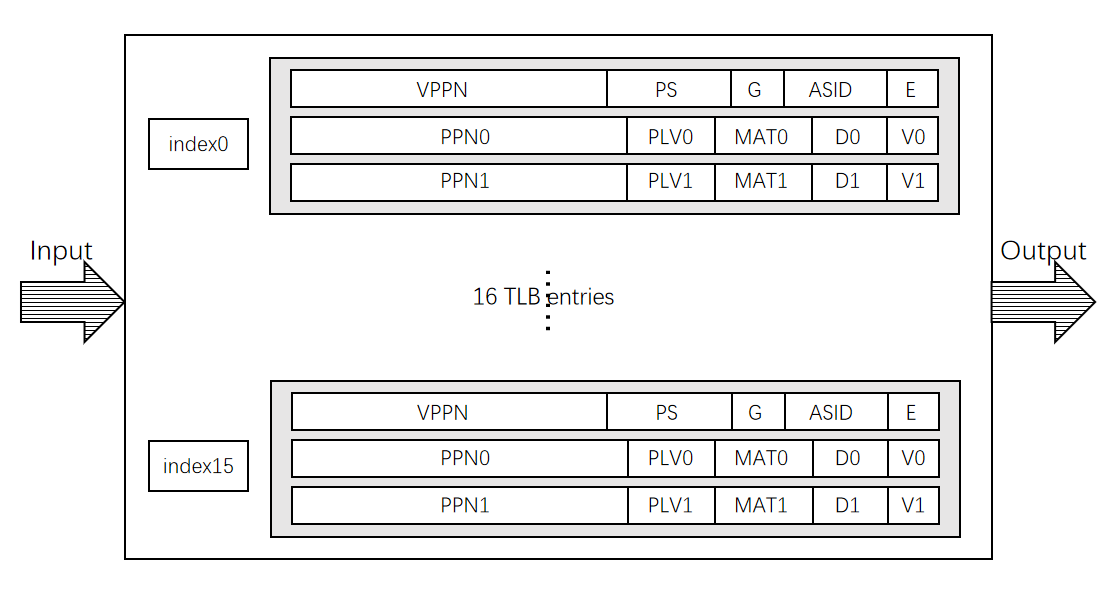


图9 TLB模块示意图

* + 1. 功能描述

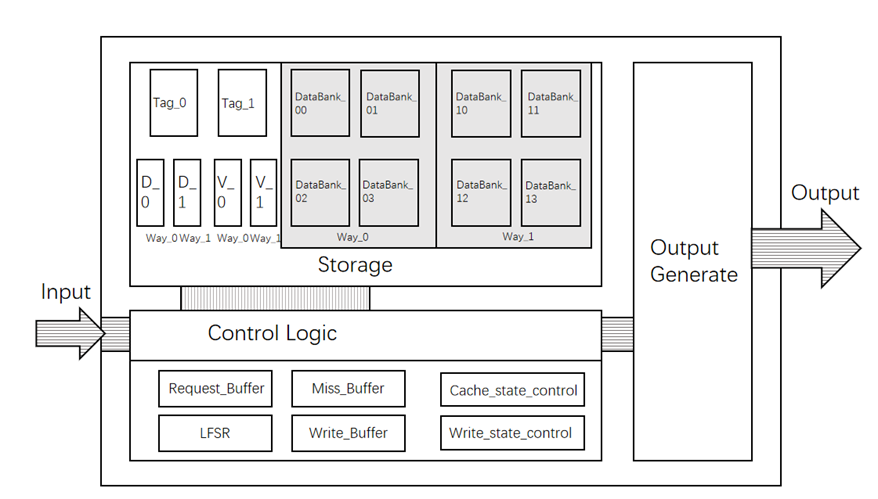


图 1 Cache结构示意图

Cache内部具体设计如上图所示。共分为三大部分，分别是存储模块、控制模块与输出模块。

在存储模块中，通过实例化Xilinx IP核中的Block RAM实现Tag表和Data表，通过寄存器实现Dirty和Valid标志位等信息。

在控制模块中包含两个状态机，用来控制Cache接收并处理请求的整个过程。此外还包含三个暂时存放数据的buffer：Request buffer，Write buffer和Miss buffer。在写命中时，数据会先存到Write buffer里，再写到cache里。而在读/写缺失时，内存返回的数据被直接写到cache里，不会使用Write buffer。

输出模块则由一些组合逻辑与少部分的时序逻辑构成，它们在状态机的控制下生成输出到CPU和内存的信号。

下面逐一详细介绍这三个模块。

* 1. 存储模块

Cache按各行中的信息组织成多张表，比较信息位Tag、有效位V、脏位D、数据Data各自成为两张表，一张对应Cache的一路。Cache中包含两张256项×20 bit的tag表、两张256项×1 bit的V表、两张256项×1 bit的D表以及两张156项×128 bit的data表。每张表都有256项，对应265个Cache行。与讲义中的实现方式略有不同的是，我们没有把Tag表和V合并在一起，而是和实现D表一样，用寄存器实现V表。于是，各表的实现方式分别为，Tag表由两块深度为256，宽度为20的同步RAM实现；Data表由四块深度为256，宽度为32的同步Ram实现，四块拼起来共同构成128位的数据；V表和D表各由两个位宽为256的寄存器实现。

上面提到的表都需要初始化，由寄存器实现的表在复位信号拉高初始化，由同步RAM实现的表则在定制IP核时通过Fill Remaining Memory Locations来初始化成0。

* 1. 控制模块
     1. 状态机

控制模块内一共有两个状态机，分别是主状态机和write buffer状态机。

1. **主状态机**

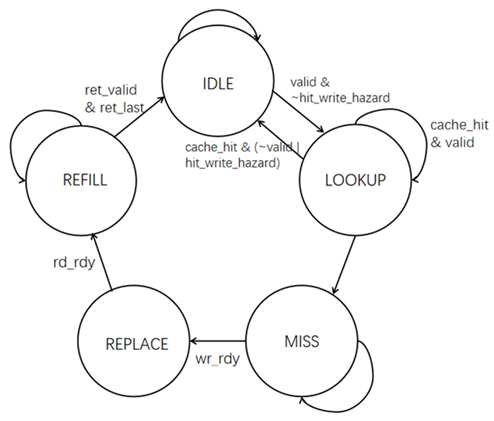


图 2 主状态机状态转移图

主状态机IDLE 状态表示 Cache 空闲，收到新的请求后转换到LOOKUP状态。如果查询结果为命中，则根据有无新请求来决定留在LOOKUP还是回到IDLE。如果没有命中，则进入MISS状态。MISS状态用来在当前处理的操作没有命中时，等待AXI总线的wr\_rdy信号。等到wr\_rdy为1后会发起进入REPLACE状态，在REPLACE状态的第一拍向内存发送写请求，写数据来自从Cache中读出的要被替换的Cache行。在REPLACE状态下，还需要等待AXI总线的rd\_rdy信号拉高，拉高后进入REFILL状态，并发出读请求，读取缺失的数据。等数据全部从AXI返回后切换回IDLE状态。

1. **Write buffer 状态机**

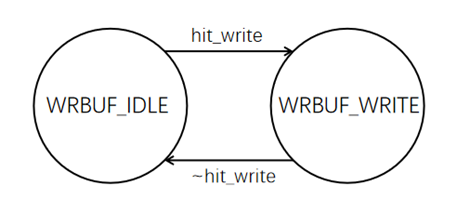


图 3 Write buffer 状态机状态转移图

Write buffer 状态机负责在写命中时将write buffer中的数据写入cache。它一共有两个状态，IDLE状态为空闲状态，WRITE状态负责写入cache。一旦写命中，也就是当主状态机处于LOOKUP状态、op\_reg为写操作对应的1时，查找命中，hit\_write拉高，状态机就会从IDLE状态进入WRITE状态，并将同步RAM对应的写使能置1，将输入写入RAM，直到没有待写的数据时才转换到IDLE状态。



* + 1. buffer

1. **request buffer**

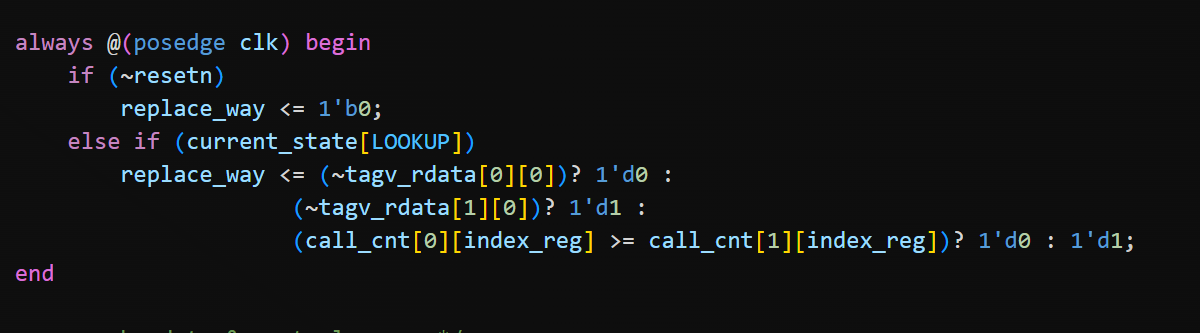
CPU发来的请求中包含的op， wstrb， wdata， index， tag， offset都会被存到request buffer中。后面生成控制信号、查找Cache、向内存发读写请求时都需要用到暂存在里面的数据。

1. **miss buffer**

在我们的实现中，miss buffer由两个寄存器构成：replace\_way和ret\_cnt。

replace\_way会在需要替换时更新，记录当前由随机数决定的要替换的路，以此保证被写回到内存的数据和重新填回的新数据位于同一路上。这里的随机数是伪随机数，由寄存器lfsr控制，我们取lfsr的最后一位为替换路。

ret\_cnt记录AXI总线返回的字的个数，从而能够将返回的数据写回到Cache中正确的位置。每返回一个字，ret\_cnt都会加1。当ret\_last和ret\_valid同时拉高时，说明当拍传输的是最后一个字，于是使用非阻塞赋值将计数器清零，这样计数器就会在下一拍变成0。



1. **write buffer**

write buffer记录了在写命中时要写入Cache的信息。它在hit\_write拉高时更新，记录替换路，并记录来自req\_buf的index， offset， wstrb和wdata。hit\_write拉高也会让write buffer状态机进入WRITE状态，控制write buffer中的数据写入同步RAM。

* + 1. RAM控制信号

tag ram的写使能只需要在ret\_valid=1且ret\_last=1时拉高，并且只要拉高对应一路的使能，因为tag只会在读/写miss，要从内存中读取缺失数据时才会被更新。tag ram的写数据来源也只有req\_buf中记录的tag这一种，无需更多的信号来控制。

data bank ram则需要考虑更多情况。在因为读/写miss而需要将从内存中读出的数据写入Cache时，它和tag ram一样要更新，但它并不会一下子完成更新，而是在内存每返回一个字时将它写入对应的bank中。另外，在写命中时无需更新tag，但需要将write buffer中的数据写入data bank ram，这时也需要拉高写使能。基于这两种情况，需要用多路选择器生成data\_bank\_wdata，并且需要将wstrb纳入考虑范围。

* 1. 输出模块

cache输出给cpu的信号包括addr\_ok、data\_ok、rdata，addr\_ok和data\_ok的生成逻辑讲义上已经给出，这里不再赘述。rdata在cache缺失时是从内存中读出的数据，在cache命中时是从cache中读出的数据。

在输出到axi的信号中，rd\_req在主状态处于REPLACE状态时拉高。wr\_req由一个寄存器赋值，在从MISS状态进入REPLACE状态时拉高一拍。因为从MISS状态进入REPLACE状态的条件是wr\_rdy=1，因此wr\_req一定能在下一拍被接受。rd\_addr来自req\_buf，wr\_addr和wr\_data则来自需要被替换的cache行。

* 1. 重要模块2设计：CSR模块
     1. 工作原理

新增用于虚实地址转换及TLB相关例外处理的寄存器

* + 1. 接口定义

表3 CSR模块接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟输入 |
| resetn | IN | 1 | 复位信号 |
| csr\_num | IN | 14 | 控制状态寄存器读写地址 |
| csr\_we | IN | 1 | 控制状态寄存器写使能 |
| csr\_wmask | IN | 32 | 控制状态寄存器写掩码 |
| csr\_wvalue | IN | 32 | 控制状态寄存器写数据 |
| ertn\_flush | IN | 1 | ertn指令刷新信号 |
| wb\_ex | IN | 1 | 例外发生信号 |
| wb\_ecode | IN | 6 | 例外发生ecode代码 |
| wb\_esubcode | IN | 9 | 例外发生esubcode代码 |
| wb\_pc | IN | 32 | WB级PC值，用于记录例外发生的PC |
| csr\_rvalue | OUT | 32 | 控制状态寄存器读数据 |
| ex\_entry | OUT | 32 | 例外跳转入口地址 |
| tlb\_ex\_entry | OUT | 32 | TLB例外跳转入口地址 |
| has\_int | OUT | 1 | 发生中断信号 |
| ipi\_int\_in | IN | 1 | 核间中断输入 |
| coreid\_in | IN | 32 | 核编号，也是TID初值 |
| hw\_int\_in | IN | 8 | 硬件中断输入 |
| wb\_vaddr | IN | 32 | 出错地址 |
| csr\_tlbehi\_vppn | OUT | 19 | CSR.TLBEHI.VPPN，TLB表项的虚页号 |
| csr\_tlbidx\_index | OUT | 4 | CSR.TLBIDX.INDEX，TLB表项索引 |
| tlbsrch\_we | IN | 1 | TLBSRCH指令的写使能 |
| tlbsrch\_hit | IN | 1 | TLBSRCH查询命中 |
| tlbrd\_we | IN | 1 | TLBRD指令的写使能 |
| tlbsrch\_hit\_index | IN | 4 | TLBSRCH查询命中的索引 |
| csr\_crmd\_rvalue | OUT | 32 | CSR.CRMD的值 |
| csr\_asid\_rvalue | OUT | 32 | CSR.ASID的值 |
| csr\_dmw0\_rvalue | OUT | 32 | CSR.DMW0的值 |
| csr\_dmw1\_rvalue | OUT | 32 | CSR.DMW1的值 |
| r\_tlb\_e | IN | 1 | 从TLB读出的e域信息 |
| r\_ tlb\_vppn | IN | 19 | 从TLB读出的vppn域信息 |
| r\_ tlb\_ps | IN | 6 | 从TLB读出的ps域信息 |
| r\_ tlb\_asid | IN | 10 | 从TLB读出的asid域信息 |
| r\_ tlb\_g | IN | 1 | 从TLB读出的g域信息 |
| r\_ tlb\_ppn0 | IN | 20 | 从TLB读出的偶页ppn域信息 |
| r\_ tlb\_plv0 | IN | 2 | 从TLB读出的偶页plv域信息 |
| r\_ tlb\_mat0 | IN | 2 | 从TLB读出的偶页mat域信息 |
| r\_ tlb\_d0 | IN | 1 | 从TLB读出的偶页d域信息 |
| r\_ tlb\_v0 | IN | 1 | 从TLB读出的偶页v域信息 |
| r\_ tlb\_ppn1 | IN | 20 | 从TLB读出的奇页ppn域信息 |
| r\_ tlb\_plv1 | IN | 2 | 从TLB读出的奇页plv域信息 |
| r\_ tlb\_mat1 | IN | 2 | 从TLB读出的奇页mat域信息 |
| r\_ tlb\_d1 | IN | 1 | 从TLB读出的奇页d域信息 |
| r\_ tlb\_v1 | IN | 1 | 从TLB读出的奇页v域信息 |
| w\_ tlb\_e | OUT | 1 | TLB写行为的e域信息 |
| w\_ tlb\_vppn | OUT | 19 | TLB写行为的vppn域信息 |
| w\_ tlb\_ps | OUT | 6 | TLB写行为的ps域信息 |
| w\_ tlb\_asid | OUT | 10 | TLB写行为的asid域信息 |
| w\_ tlb\_g | OUT | 1 | TLB写行为的g域信息 |
| w\_ tlb\_ppn0 | OUT | 20 | TLB写入偶页的ppn域信息 |
| w\_ tlb\_plv0 | OUT | 2 | TLB写入偶页的plv域信息 |
| w\_ tlb\_mat0 | OUT | 2 | TLB写入偶页的mat域信息 |
| w\_ tlb\_d0 | OUT | 1 | TLB写入偶页的d域信息 |
| w\_ tlb\_v0 | OUT | 1 | TLB写入偶页的v域信息 |
| w\_ tlb\_ppn1 | OUT | 20 | TLB写入奇页的ppn域信息 |
| w\_ tlb\_plv1 | OUT | 2 | TLB写入奇页的plv域信息 |
| w\_ tlb\_mat1 | OUT | 2 | TLB写入奇页的mat域信息 |
| w\_ tlb\_d1 | OUT | 1 | TLB写入奇页的d域信息 |
| w\_ tlb\_v1 | OUT | 1 | TLB写入奇页的v域信息 |

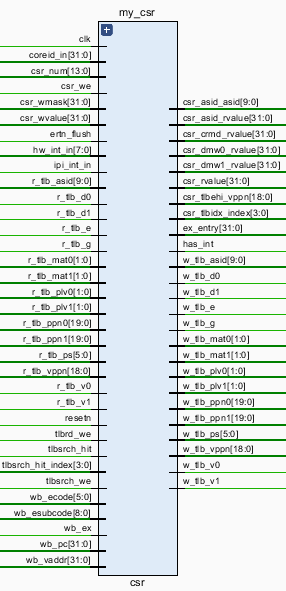


图20 CSR模块接口

* + 1. 功能描述

在虚实地址转换时，地址翻译模式由CRMD中的DA和PG域控制，为此需要让这两个域能被csr指令更新。当DA=0，PG=1时，处理器核的MMU处于直接地址翻译模式。当DA=0，PG=1时，MMU处于映射地址翻译模式，这种模式又可分为直接映射地址翻译模式和页表映射地址翻译模式。

对于直接映射地址翻译模式，相关配置信息存储在DMW0、DMW1寄存器中，每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。存储访问类型由CRMD的DATF和DATM域控制。这些域都只需支持csr指令的读写。

对于没有落在DMW0、DMW1设置的直接映射配置窗口中的地址，需要采用页表映射地址翻译模式，为此也需要在CSR模块中增加TLB相关的控制状态寄存器。TLBIDX包含TLB指令操作TLB时的索引值、PS以及是否有效等信息。TLBEHI包含TLB 表项高位的虚页号，TLBELO0、TLBELO1则包含表项低位的物页号、有效、脏等信息。ASID寄存器包含地址空间标识符等信息。TLBSRCH、TLBRD、TLBWR、TLBFILL指令会对这些寄存器进行读写。于是，这些寄存器更新的逻辑需要考虑两种情况：被TLB维护指令更新，或是被之前实验实现的csr写指令更新。TLBEHI在此基础上还需多考虑一种情况，即在发生例外时将触发例外的虚地址记录到其VPPN域中。根据这些TLB相关的控制状态寄存器，便可以产生传递给TLB模块的各种信号，其中需要注意的是，传递给TLB的w\_e信号在TLB充填例外时恒为1而不受TLBIDX的NE位的影响。

在发生TLB重填例外时，处理器核将通过更新DA和PG来进入直接地址翻译模式。TLB重填例外的入口地址储存在TLBENTRY中，该寄存器也只需支持csr指令的读写。在处理完例外后，处理器执行ETRN指令，此时需要让DA=0，PG=1，回到映射地址翻译模式。因此，DA和PG除了要支持csr指令的更新，还需考虑在发生TLB重填例外和执行ERTN时的更新。

* 1. 重要模块3设计：MMU模块
     1. 工作原理

MMU模块完成虚实地址的转换，包括判断映射模式以及查找TLB等操作，同时MMU模块还会输出一系列用于判断TLB相关异常的信号。

* + 1. 接口定义

表4 MMU模块接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| **Search port** | | | |
| s\_vppn | OUT | 19 | 查找虚双页号 |
| s\_va\_bit12 | OUT | 1 | 访存虚地址12位 |
| s\_asid | OUT | 10 | 地址空间标识 |
| s\_found | IN | 1 | 查找到信号 |
| s\_ppn | IN | 20 | 物理页号 |
| s\_ps | IN | 6 | 页大小 |
| s\_plv | IN | 2 | 权限等级 |
| s\_mat | IN | 2 | 访存种类 |
| s\_d | IN | 1 | 脏页信号 |
| s\_v | IN | 1 | 页有效信号 |
| **virtual addr and physical addr** | | | |
| va | IN | 32 | 虚拟地址 |
| pa | OUT | 32 | 物理地址 |
| asid\_input | IN | 10 | 输入地址空间标识 |
| **Data from csr** | | | |
| csr\_crmd\_rvalue | IN | 32 | 读CRMD寄存器的值 |
| csr\_dmw0\_rvalue | IN | 32 | 读DMW0寄存器的值 |
| csr\_dmw1\_rvalue | IN | 32 | 读DMW1寄存器的值 |
| **exception** | | | |
| page\_invalid | OUT | 1 | 页无效 |
| ppi\_except | OUT | 1 | 页特权等级不合规例外 |
| page\_fault | OUT | 1 | 页不匹配 |
| page\_clean | OUT | 1 | 页非脏 |

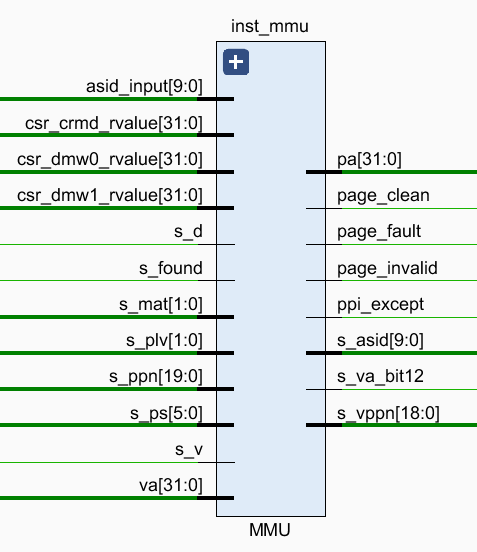


图21 MMU模块接口（取指MMU为例）

* + 1. 功能描述

MMU模块完成虚实地址的转换，地址翻译模式由CRMD中的DA和PG域控制。当DA=0，PG=1时，处理器核的MMU处于直接地址翻译模式。当DA=0，PG=1时，MMU处于映射地址翻译模式，这种模式又可分为直接映射地址翻译模式和页表映射地址翻译模式。

对于直接映射地址翻译模式，相关配置信息存储在DMW0、DMW1寄存器中。每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。

当虚地址命中某个有效的直接映射配置窗口时，其物理地址直接等于虚地址的[28：0]位拼接上该映射窗口所配置的物理地址高位。命中的判断方式是：虚地址的最高 3 位（[31：29]位）与配置窗口寄存器中的[31：29]相等，且当前特权等级在该配置窗口中被允许。映射地址翻译模式下，除了落在直接映射配置窗口中的地址之外，其余所有合法地址都必须通过页表映射完成虚实地址转换。

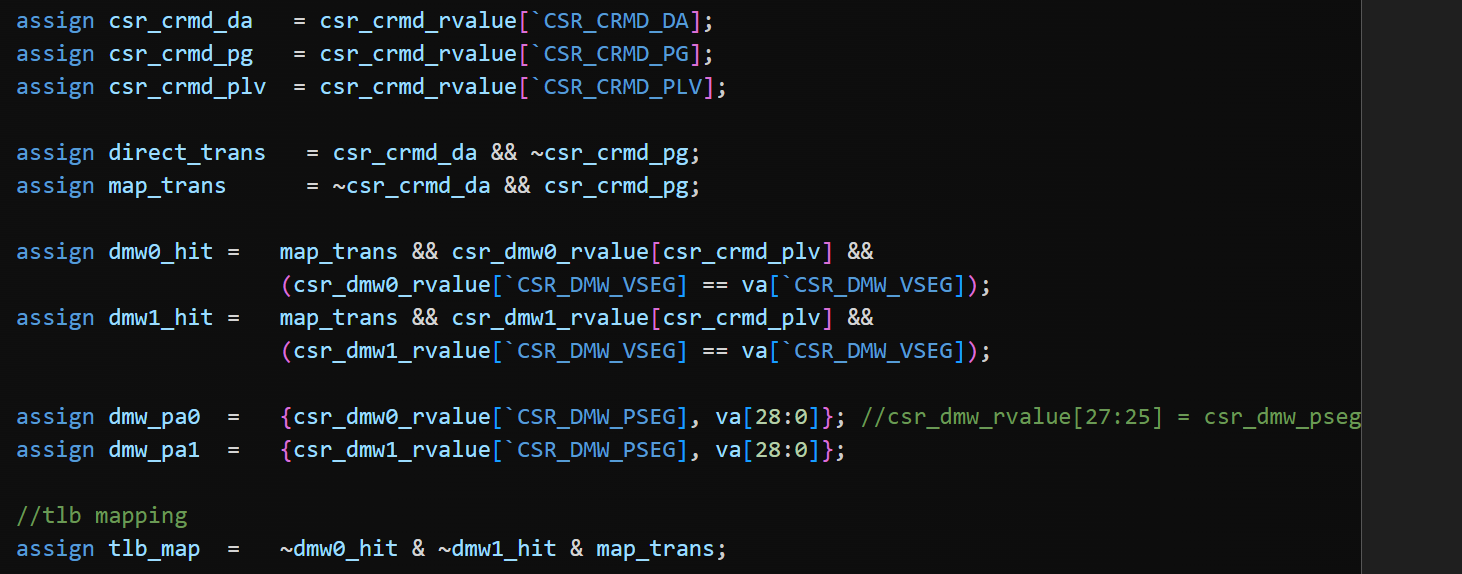


图22 MMU模块对于地址翻译模式的判断

对于页表映射地址翻译模式，TLB 作为处理器中存放操作系统页表信息的一个临时缓存，用于加速映射地址翻译模式下的取指和 load/store 操作的虚实地址转换过程。所以只需要把虚拟地址的相关信息传给TLB模块，如果能匹配到页表项，则根据TLB模块返回的PS字段的值，把物理页表项s\_ppn拼接上对应的offset，便可以得到页表映射的物理地址。

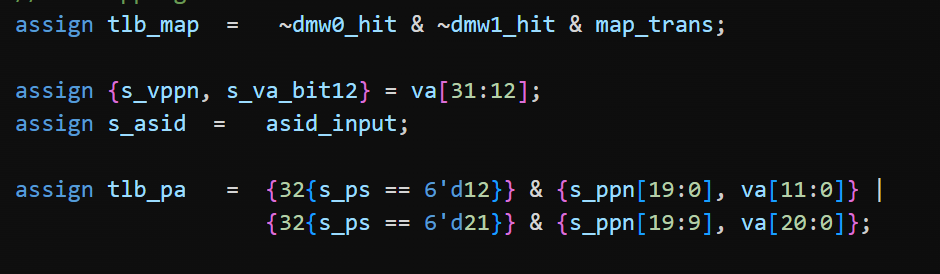


图23 MMU模块对于页表映射地址翻译模式下的物理地址的产生

如果没有匹配到页表项，则把输出信号page\_fault拉高，方便TLB重填例外的判断。除此之外，MMU模块还会根据TLB模块返回的页表项的V，D，PLV字段的值，对输出信号进行赋值，方便进行load/store取指操作页无效例外、页修改例外、页特权等级不合规例外的判断。

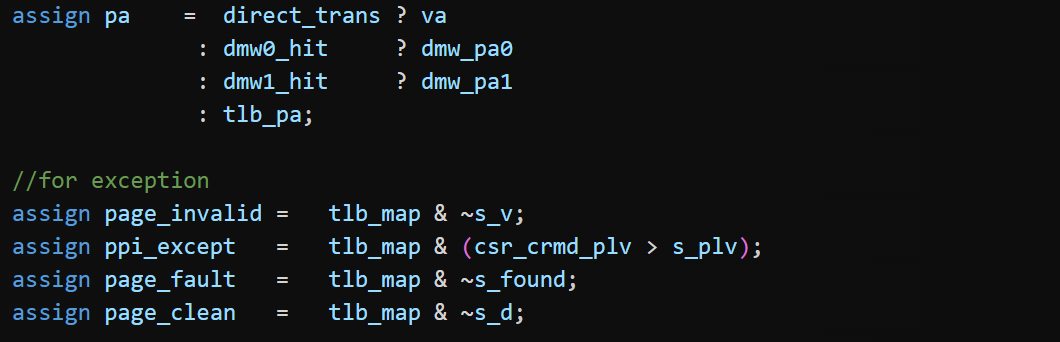
****

图24 MMU模块物理地址的产生以及用于例外判断的信号

* 1. 重要模块4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

本实验中，由于req拉高后握手成功前addr可以发生改变，所以preif\_cancel相关的逻辑都变为冗余，需要进行删除。其它逻辑与之前实验相同。

* + 1. 接口定义

表5 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 interface** | | | |
| 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 interface** | | | |
| axi\_arid | IN | 4 | AXI转接桥的读请求ID号 |
| **ID interface** | | | |
| 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 interface** | | | |
| 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 |
| **MMU interface** | | | |
| csr\_asid\_rvalue | IN | 32 | 从csr的ASID寄存器中读出地址空间标识 |
| inst\_va | OUT | 32 | 指令虚拟地址 |
| inst\_pa | IN | 32 | 指令物理地址 |
| if\_asid | OUT | 10 | 地址空间标识 |
| inst\_page\_invalid | IN | 1 | 页无效 |
| inst\_ppi\_except | IN | 1 | 页特权等级不合规例外 |
| inst\_page\_fault | IN | 1 | 页不匹配 |
| inst\_page\_clean | IN | 1 | 页非脏 |

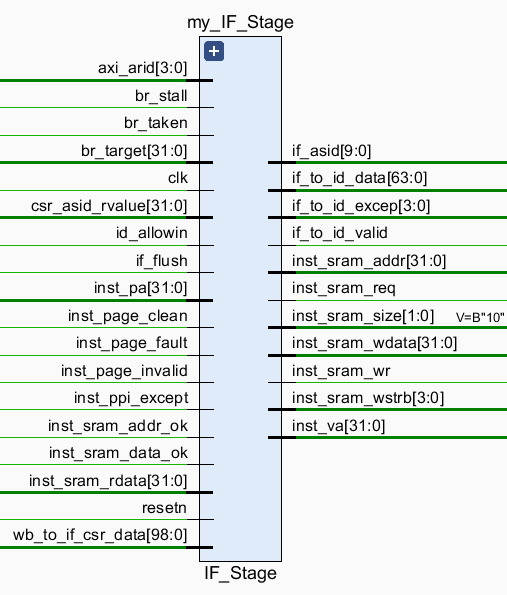


图25 IF模块接口

* + 1. 功能描述

本实验中，由于req拉高后握手成功前addr可以发生改变，所以preif\_cancel相关的逻辑都变为冗余，需要进行删除。

另外，添加cache后，发现了一些之前被隐藏的bug，本次实验也全部进行了修正。在本次实验也发现了IF流水级的许多设计上的bug，这个在bug记录部分时再具体介绍，这里就不展开进行叙述了。

1. 实验过程
   1. 实验流水账

李金明

2023.11.30 20：00-20：30 完成exp17的debug

2023.12.5 21：00-23：00 完成exp18的设计

2023.12.5 23：00-次日1：00进行exp18的debug

2023.12.6 10：00-11：00完成exp18的debug

2023.12.6 23：30-次日3：00 完成exp19设计

2023.12.7 10：00-12：30，14：00-16：30完成exp19的debug

2023.12.15 17：30-22：10 撰写实验报告

贾城昊

2023.11.30 14：30-15：00，20：00-20：30 完成exp17的debug

2023.12.4 13：00-15：00进行exp18的设计

2023.12.5 12：00-15：00，21：00-23：00 完成exp18的设计

2023.12.5 23：00-次日1：00进行exp18的debug

2023.12.6 11：00-14：00进行exp19的设计

2023.12.6 22：30-次日3：00完成exp19的设计

2023.12.15 13：30-17：00，22：30-23：00 撰写实验报告

牛浩宇

2023.11.28-29 完成exp17的设计

2023.12.18晚补充实验报告

* 1. 错误记录
     1. 错误1：exp17中PS字段赋值出错
        1. 错误现象

Console报错如下：

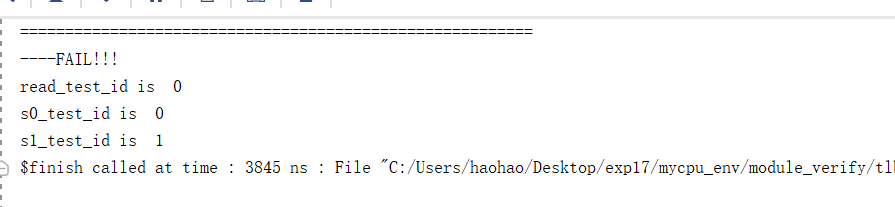


图38 错误1对应的Console报错

* + - 1. 分析定位过程

查看波形图，查看TLB相关信号的值，发现r\_ps的值出现明显的错误，然后查看r\_ps的赋值逻辑，很快便定位到了问题所在。

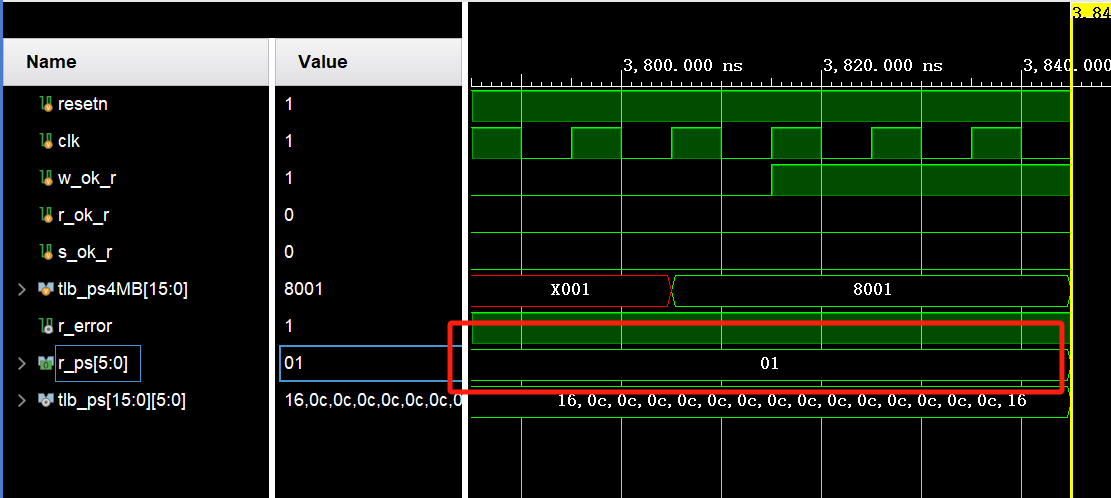


图39 错误1对应的波形图

* + - 1. 错误原因

TLB模块中对r\_ps信号的赋值逻辑出错，其实本质原因还是因为本组写exp17的成员在写r\_ps的赋值逻辑时出现笔误，所以才导致了如下代码的产生：

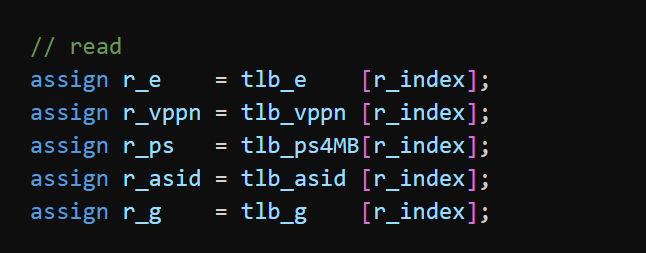


图40 错误1中r\_ps赋值逻辑出错

* + - 1. 修正效果

这是很明显的笔误，所以把TLB模块中对r\_ps信号的赋值逻辑进行修改即可，如下所示：

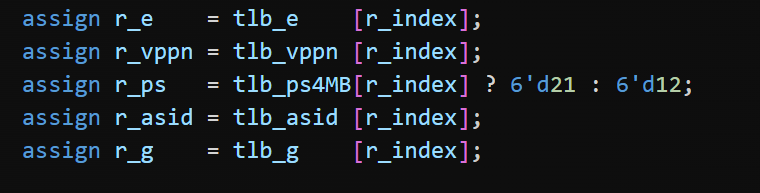


图41 错误1对应的修改代码

该方法有效，exp17通过。

* + 1. 错误2：exp18流水线回滚时PC更新出错
       1. 错误现象

Console报错如下

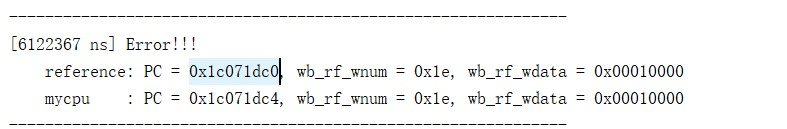


图42 错误2对应的Console报错

* + - 1. 分析定位过程

从console报错来看，PC出错但寄存器写是对的，按照以往的经验，很可能是PC的更新或者指令的取消的问题，但不管怎么说，都得首先查看波形图，如下所示：

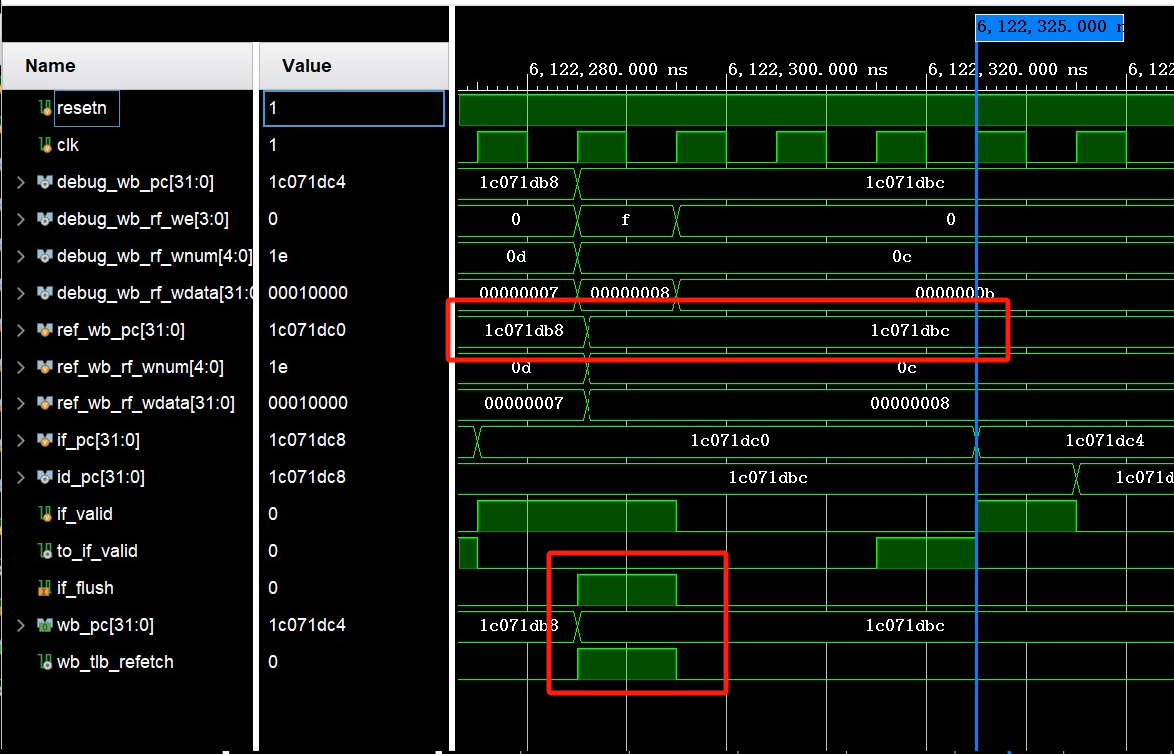


图43 错误2对应的波形图

通过一波向前溯源的操作，发现出现问题的地方在于，当tlb\_refetch信号拉高时，本来应该清空流水线，然后重新进行取值，这个时候从理论上来说，应该PC会变为WB对应指令的下一条指令的PC，但通过波形图可以看出，此时PC仅仅进行了加4的操作，而没有回滚到对应的指令的PC值，从而导致出错。

其实分析到这后，已经可以确定是设计的问题了（其实写之前本组成员设计时，已经考虑了这个问题，但最终由于工作量比较多，导致debug前这个地方忘记编写相关代码了）于是，本组成员查看了IF流水级的PC更新的代码，果然发现WB没有把其PC值传过来，而且也没有对应的选择逻辑，所以PC进行了默认的加4操作，从而导致出错。

* + - 1. 错误原因

对TLB指令进行回滚操作时，需要将WB的PC传递给IF流水级，并把相关控制信号传递给IF，方便IF流水级nextpc信号进行选择，但本组成员由于疏忽，debug前忘记编写与之相关的代码，所以导致出错，出错前IF的nextpc更新逻辑如下所示：。

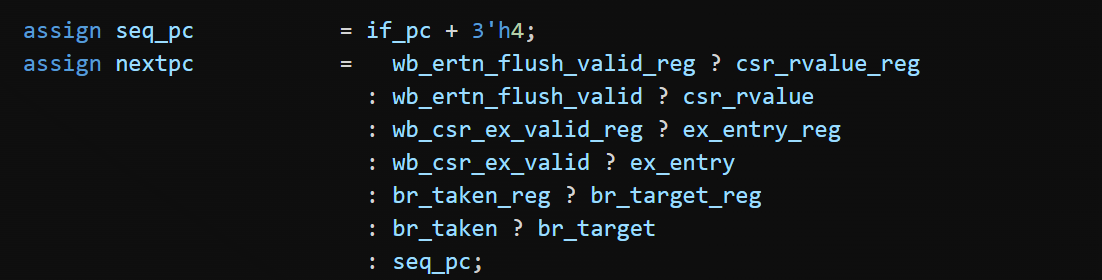


图44 错误2修改前代码(nextpc赋值逻辑)

* + - 1. 修正效果

首先，新增将WB的PC传递给IF流水级的数据通路，并把相关控制信号传递给IF，方便IF流水级nextpc信号进行选择。然后，在IF流水级新增寄存器存储WB的PC值和其控制信号，方便nextpc进行选择。最后，更改nextpc更新逻辑，如下所示：

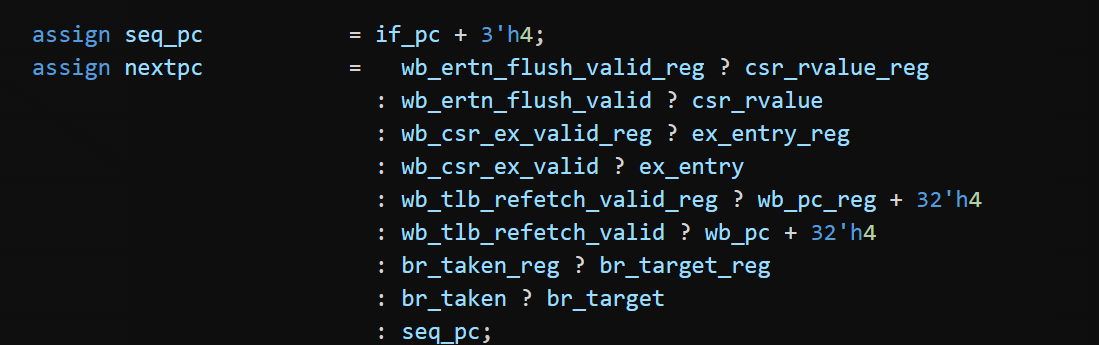


图45 错误2修改后代码(nextpc赋值逻辑)

该方法有效，来到下一个bug。

* + 1. 错误3：exp18中WB级对MEM传递来的TLB信号数据通路解码出错
       1. 错误现象

Console报错如下

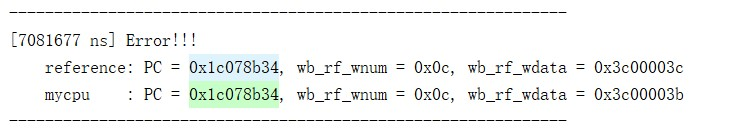


图46 错误3对应的Console报错

* + - 1. 分析定位过程

这个错误的定位过程花费较久，因为最终出错位置相比于报错位置较远。起初本组成员根据汇编代码，对汇编的预期行为进行判断，最后花费了较长时间仍然没有找到错误所在。于是，本组成员转换思路，考虑到出错在第60个测试点，其对应汇编文件如下：

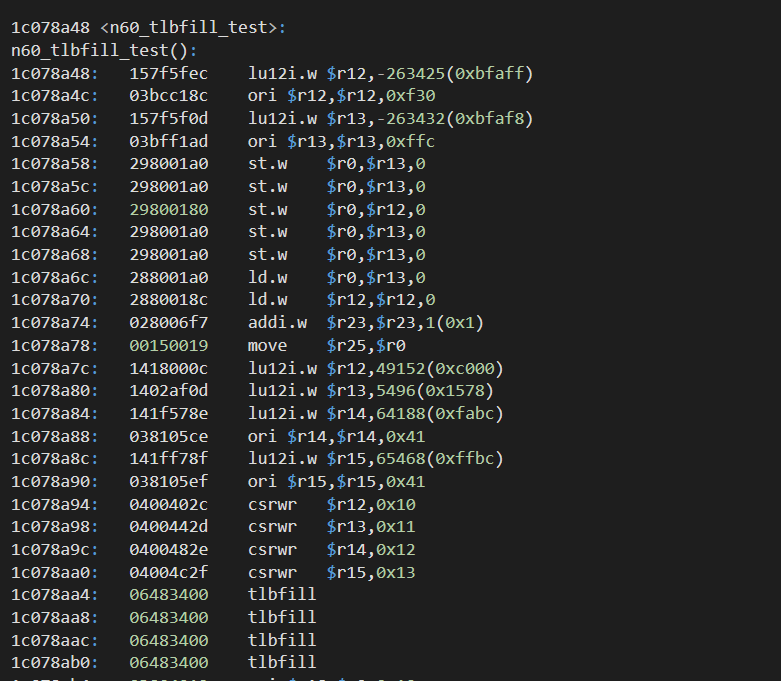


图47 错误3的出错测试点汇编代码

通过汇编代码可知，出错的测试点是为了测试tlbfill指令，而在此测试点中出现的新指令只有tlbfill，其它指令通过之前的测试可知其处理是没有问题的。于是问题的焦点便成为了tlbfill的前后一段指令，查看这段指令对应的波形图，观察相关信号，很快便发现了出现问题的信号，如下所示：

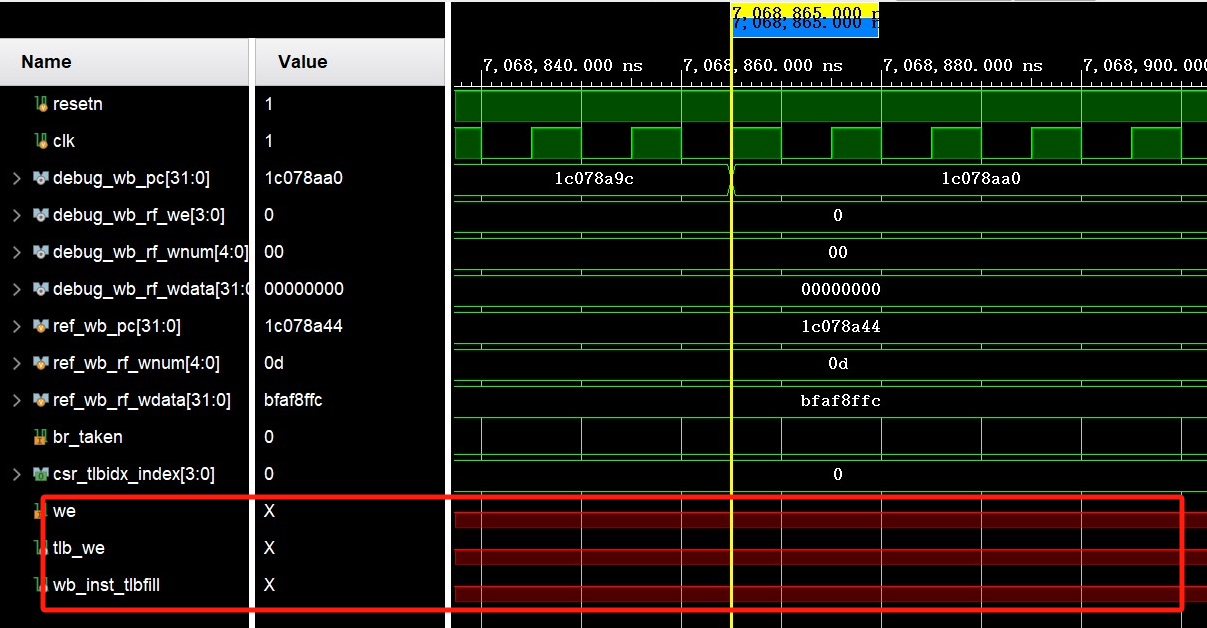


图48 错误3对应的波形图

通过波形图，可以发现，此时传递给TLB模块的we，tlb\_we信号以及传递的inst\_tlbfill信号均为X，这显然是不对的。于是查看MEM到WB的数据通路以及解码逻辑，果然发现了问题所在：

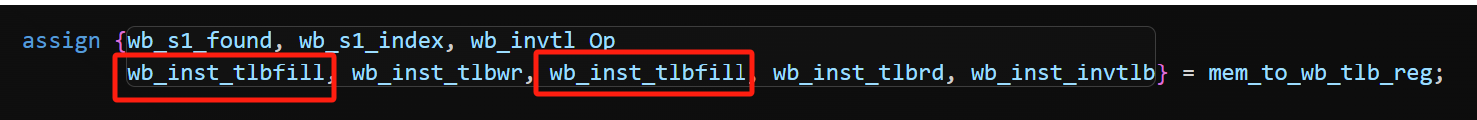


图49 错误3对应的错误代码

* + - 1. 错误原因

WB流水级对MEM流水级传递过来的TLB相关信号的解码出错：由于笔误，导致本应是wb\_inst\_tlbsrch的位置写成了wb\_inst\_tlbfill，所以导致WB解码出的wb\_inst\_tlbfill信号值为X，导致tlbfill的操作出现错误，从而影响到了后面的指令的行为。

* + - 1. 修正效果

将笔误修改过来，即把第一个wb\_inst\_tlbfill改为wb\_inst\_tlbsrch即可，如下所示：

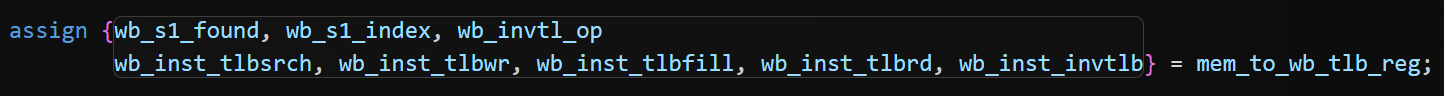


图50 错误3修改后的代码

该方法有效，来到下一处bug。

* + 1. 错误4：exp18中未考虑invtlb指令op值错误的情况
       1. 错误现象

运行仿真后，console报错如下：

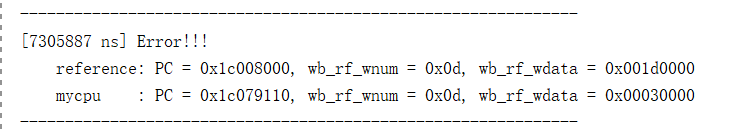


图51 错误4对应的Console报错

* + - 1. 分析定位过程

首先查看报错位置的汇编代码：

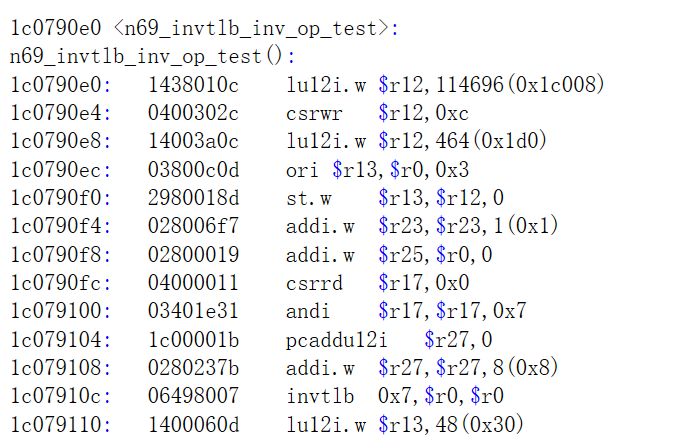


图52 错误4对应的汇编指令

显而易见，地址0x1c07910c处的invtlb指令的op值为0x7，这与要求不符：invtlb指令的op值只能为0x0-0x6，应该进入异常，从错误的PC值也可以看出，我们设计的CPU进行的是invtlb指令的下一条指令，而正常应该进入异常处理程序。

* + - 1. 错误原因

当invtlb指令的op值不合法时，按照指令集手册所述，应当进入保留异常，但我们当时不太确定触发什么异常，因此没有写这个地方。后来经过思考并询问助教老师确认后，决定当invtlb指令的op值不合法时，触发INE指令不存在异常。

* + - 1. 修正效果

在ID模块中更改INE异常的触发条件即可，如下所示：

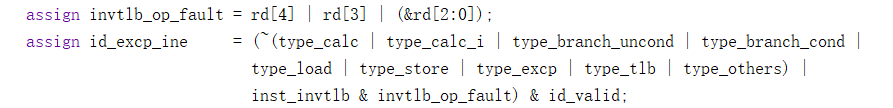


图53 错误10修改后代码

该方法有效，exp18通过。

* + 1. 错误5：exp19中EX模块传到mmu的va出错
       1. 错误现象

运行仿真后，console报错如下：

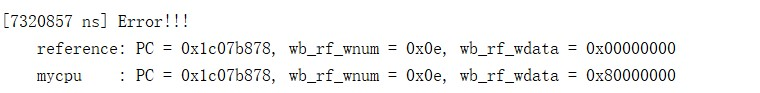


图54 错误5对应的Console报错

* + - 1. 分析定位过程

首先查看报错位置的汇编代码：



图55 错误5对应的汇编指令

可以发现，该部分的功能是首先执行tlbsrch指令，tlbsrch指令会将查找结果写入CSR.TLBIDX，然后通过csrrd指令将该csr寄存器中的值读到寄存器中，从而与golden trace比较，确定tlbsrch执行的结果是否正确。

Golden trace的结果是0x00000000，而我们设计输出的是0x80000000，说明tlbsrch本应有命中项，但实际执行的过程中未命中。由于exp17和exp18已经对TLB模块有了充分验证，我们不认为是TLB模块内部工作出错，而将视角转向参数传递的过程。查看相应的波形图：

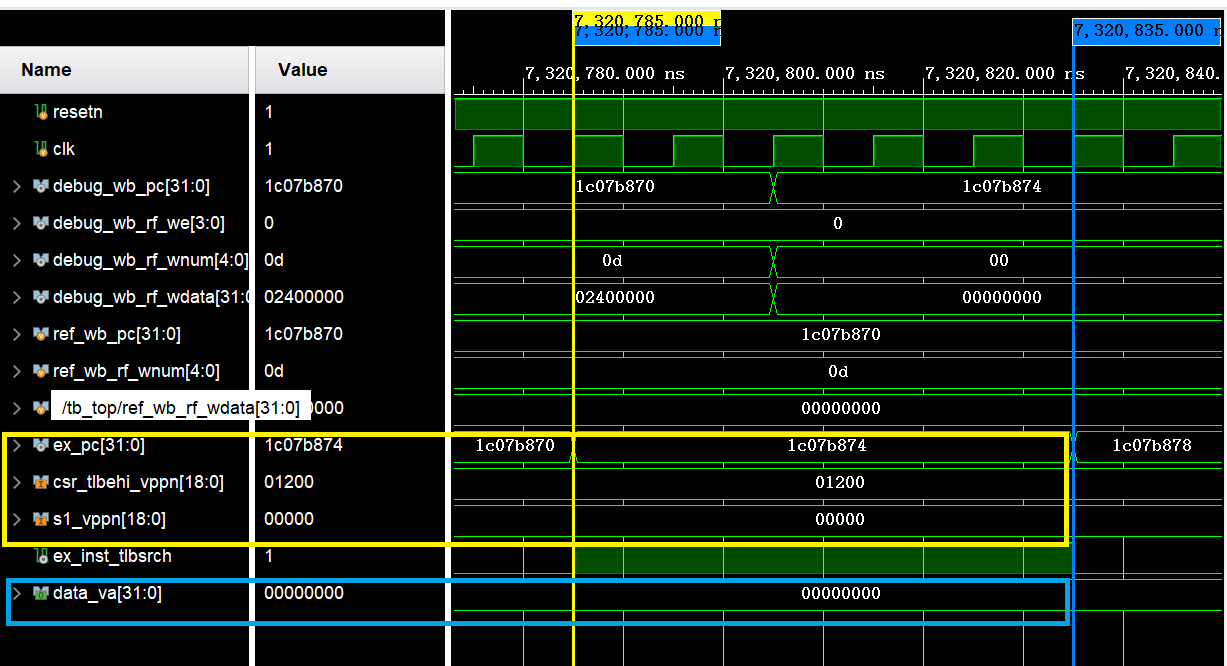


图56 错误5对应的波形图

首先观察黄框中的信号，可以看到，当EX模块处理的指令对应地址为0x1c07b874时，代表当前指令是tlbsrch的信号正常拉高了。但是从CSR.TLBEHI中读出的数据是0x01200，而传递到TLB的vppn值是0x00000，二者不匹配。这一信号的传递过程是：EX模块先得到CSR.TLBEHI中的数据，从而确定传递到MMU的虚拟地址，再由MMU解析，将vppn部分传给TLB模块。我们认为错误大概率出现在EX模块中，调取EX模块传递到MMU的va信号（蓝框部分），可以发现其错误地为0。查看相关代码：

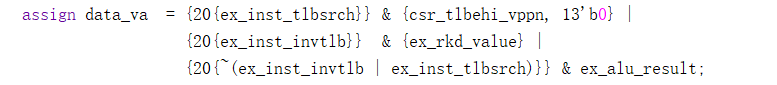


图57 错误5中data\_va的赋值

* + - 1. 错误原因

在我们的设计中，当EX模块处理虚拟地址时，一开始是将地址的31到12位传递给MMU供其解析，后来决定将整个虚拟地址都传递过去，更改逻辑时忘记因为data\_va是32位的，因此要将这一部分代码中的20改为32，从而导致出错。

* + - 1. 修正效果

把EX流水级传递给MMU的data\_va信号赋值逻辑中的20改为32即可，如下所示：

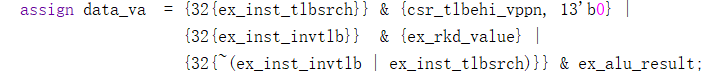


图58 错误5修改代码

该方法有效，来到下一个bug。

* + 1. 错误6：exp19中csr模块修改后读crmd值出错
       1. 错误现象

运行仿真后，console报错如下：

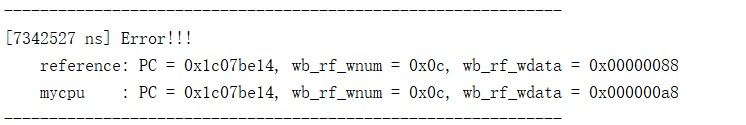


图59 错误6对应的Console报错

* + - 1. 分析定位过程

首先查看报错位置的汇编代码：

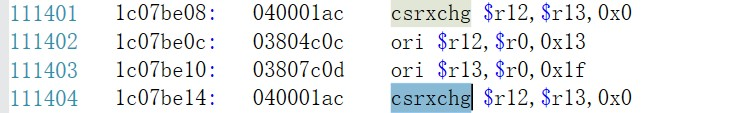


图60 错误6对应的汇编指令

可以发现，该部分的功能是执行csrxchg指令，查看的寄存器是CSR.CRMD中的值。0x1c07be08处的csrxchg指令将寄存器的内容写入 CSR.CRMD，并通过把与golden trace比对判断CRMD中的旧值是否正确，0x1c07be14处的csrxchg指令也是如此。考虑到csrxchg指令的实现应该不会有问题，不会是后一条csrxchg指令读数据错误，同时考虑到该部分汇编代码的工作逻辑，问题大概率出现在0x1c07be08处的csrxchg指令向CRMD写回的过程中。查看相关波形图：

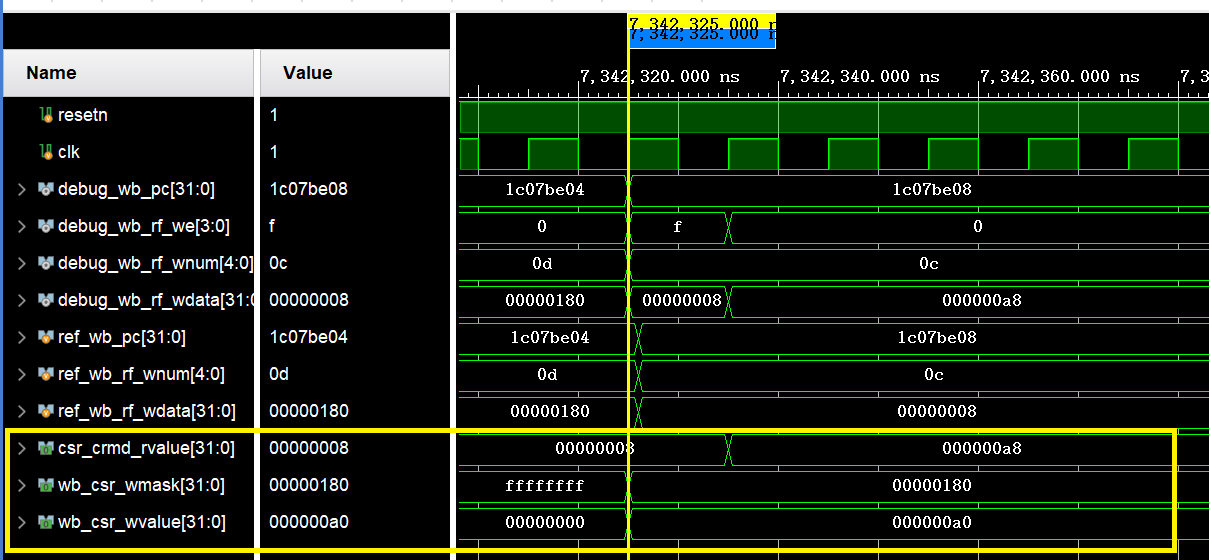


图61 错误6对应的波形图

可以看到，当debug\_wb\_pc为0x1c07be08时，第一条csrxchg指令会向CRMD寄存器写回数据。传递给CSR模块的写数据wb\_csr\_wvalue信号是0x000000a0，掩码信号wb\_csr\_wvalue是0x00000180，而查看CSR模块中的csr\_crmd\_rvalue信号，过了一个周期CRMD中的数据更新后，其值变为了0x000000a8，与预期的0x00000088不符。故在CSR模块中查看CRMD寄存器写数据和读数据赋值的相关代码：

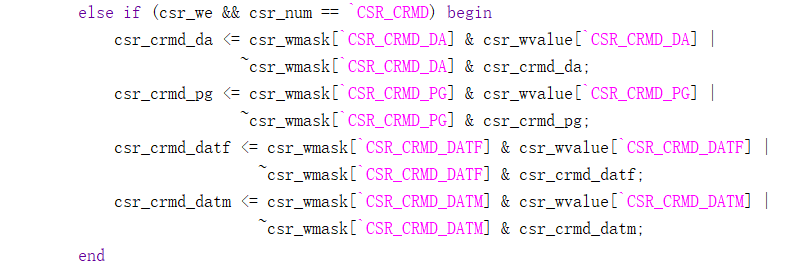


图62 错误6中CSR.CRMD写数据的更新逻辑



图63 错误6中CSR.CRMD读数据的赋值

* + - 1. 错误原因

可以发现，写回的过程没有什么问题，问题出现在读数据的赋值。在CRMD中，第5到6位为DATF，7到8位为DATM，而我们的代码中错误地将第5到6位也赋值为了DATM部分，导致出错。这一部分代码在以前实验中就存在问题，但是之前不涉及地址翻译模式，不论是DATF还是DATM都是0，故不会发生错误。

* + - 1. 修正效果

把csr\_crmd\_rvalue信号赋值逻辑修改正确即可，如下所示：



图64 错误6修改后代码

该方法有效，来到下一个bug。

* + 1. 错误7：exp21中AXI处理异常的逻辑出错
       1. 错误现象

运行仿真后，console报错如下：

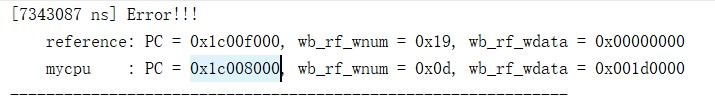


图65 错误7对应的Console报错

* + - 1. 分析定位过程

由console报错可知，PC又没有得到更新了，按照经验，应该是跳转分支，例外处理等地方出现了错误，打开波形图，如下所示：

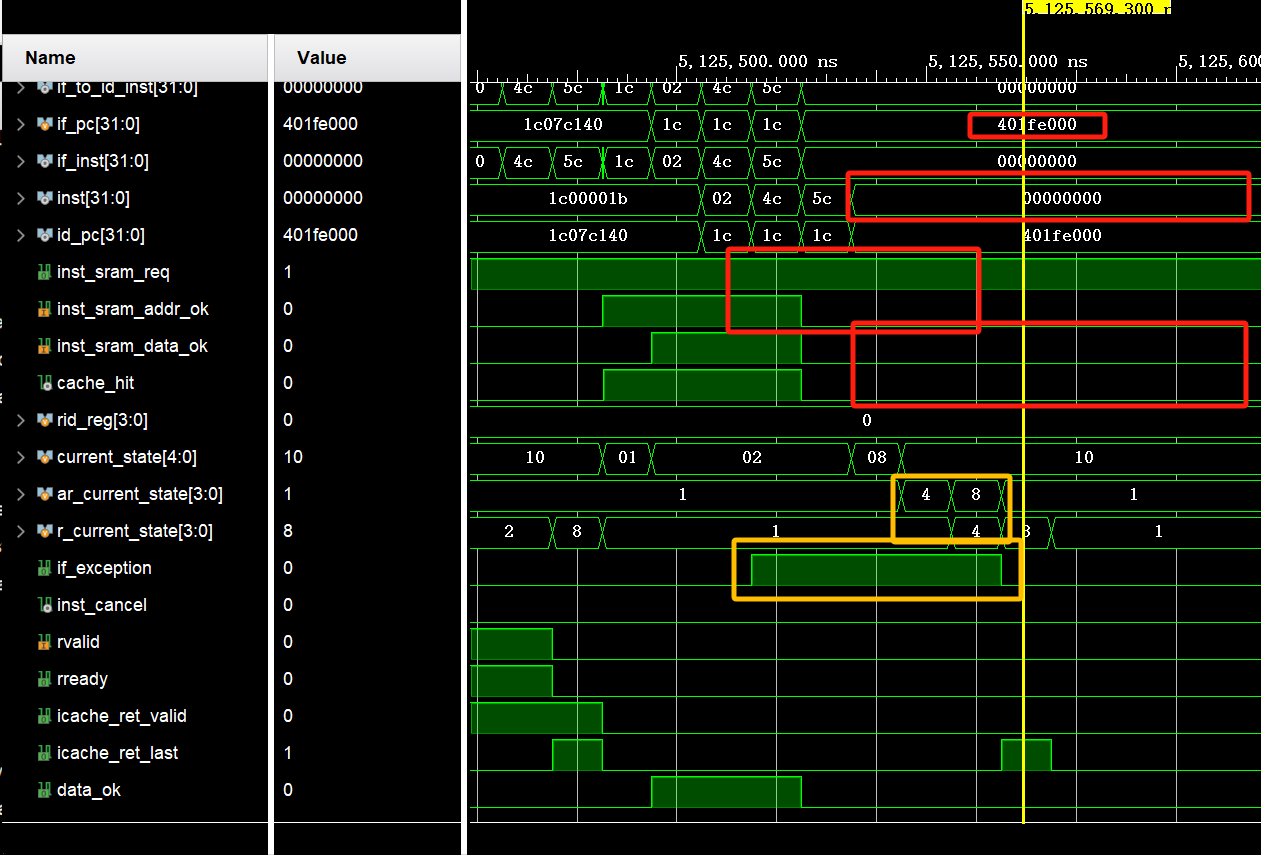


图66 错误7对应的波形图

从波形可以看出，PC没有更新的原因是addr\_ok握手成功后，data\_ok并没有返回。然后继续查看cache发现，cache并没有命中。同时也可以注意到，此时if\_pc不是一个合法的地址，所以if\_exception也拉高了。由于cache没有hit，所以查看AXI的状态转移，很明显发现此时读通道和读响应通道的状态机均进入了“4”这个状态。

于是很快我们便想到了这个状态的含义，代表的是EXCEP，即当收到不合理地址时，AXI转接桥不向总线发送请求。其通过IF流水级传递过来的if\_exception进行处理。但是本次实验中，AXI与cache进行交互，所以IF的例外信号传递给AXI导致其不会进行burst传输，从而也就不会返回给cache信号。所以cache也就不会给IF传递data\_ok信号。其中原先AXI对异常的处理逻辑如下：

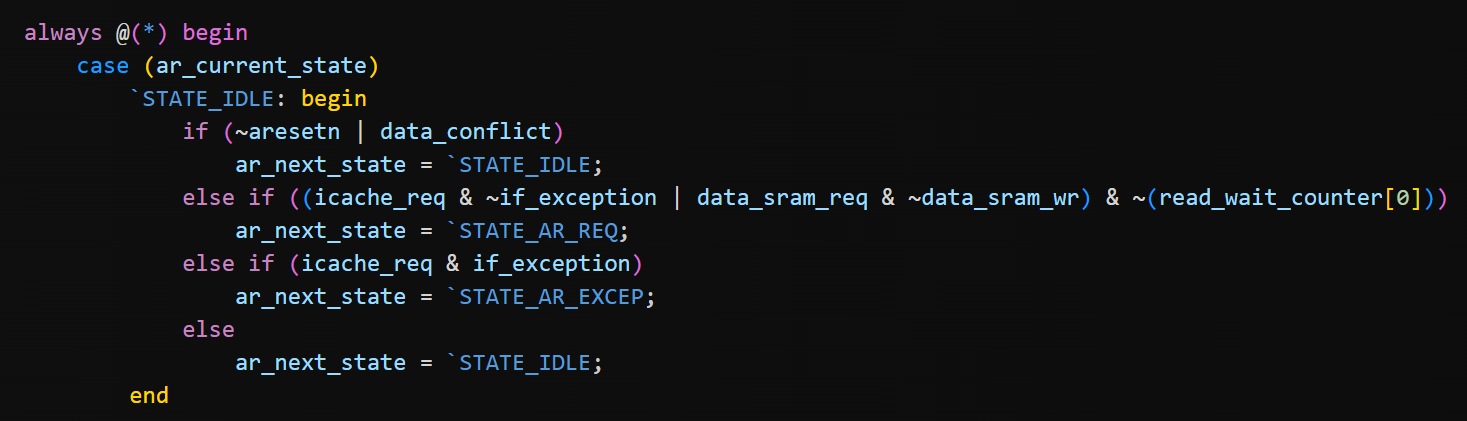


图67 错误7中AXI对异常的处理逻辑(1)

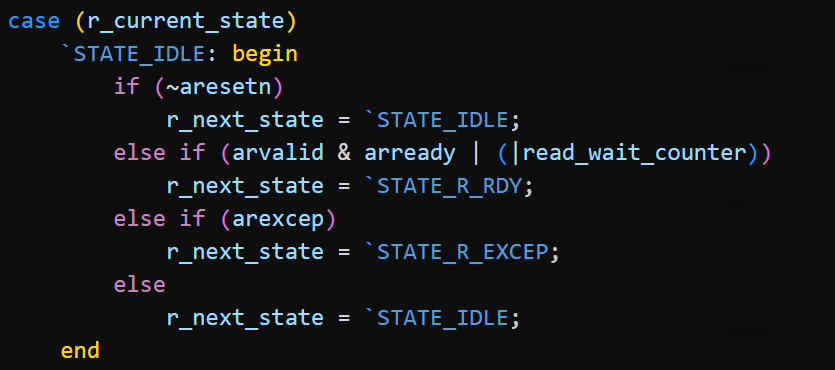


图67 错误7中AXI对异常的处理逻辑(2)

* + - 1. 错误原因

当收到不合理地址时，AXI转接桥不向总线发送请求。其通过IF流水级传递过来的if\_exception进行处理。但是本次实验中，AXI与cache进行交互，所以IF的例外信号传递给AXI导致其不会进行burst传输，从而也就不会返回给cache信号。所以cache也就不会给IF传递data\_ok信号。

* + - 1. 修正效果

对上面处理异常的逻辑进行修改，AXI不再根据IF传来的if\_exception进行处理，取消EXCEPT这个状态。异常处理改为由流水线和cache模块进行处理：

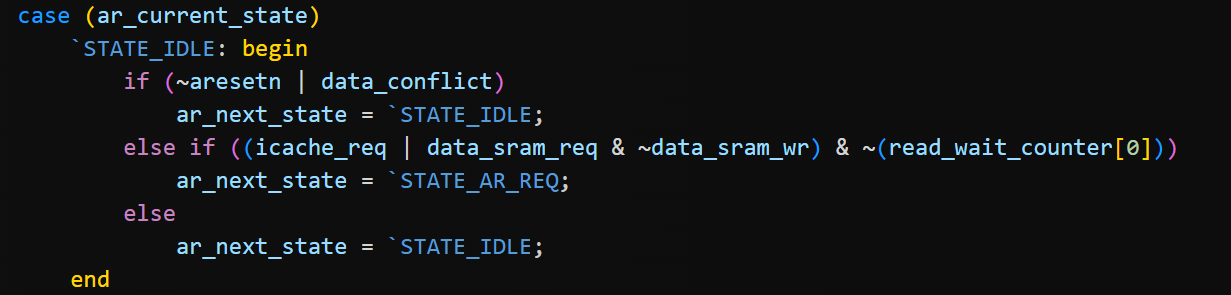


图68 错误7修改后代码（1）

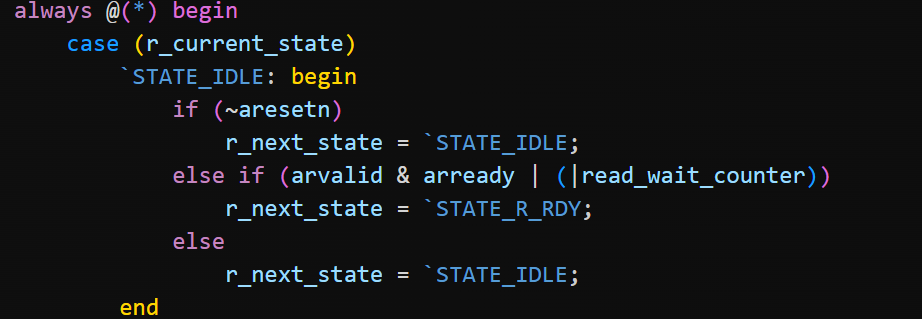


图68 错误7修改后代码（2）

该方法有效，来到下一个bug。

* + 1. 错误8：exp21中IF流水指令寄存器更新逻辑出错
       1. 错误现象

运行仿真后，console报错如下：

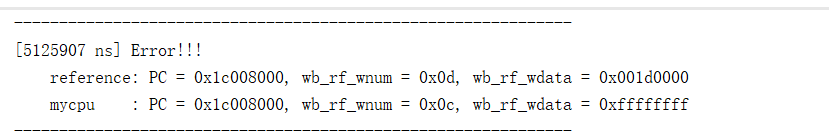


图69 错误8对应的Console报错

* + - 1. 分析定位过程

从console报错来看，PC是对的，但是写回寄存器的数据不对。于是按照经验，很明显这个IF和ID的控制逻辑出错，导致指令和PC不对应，于是查看相关波形图：

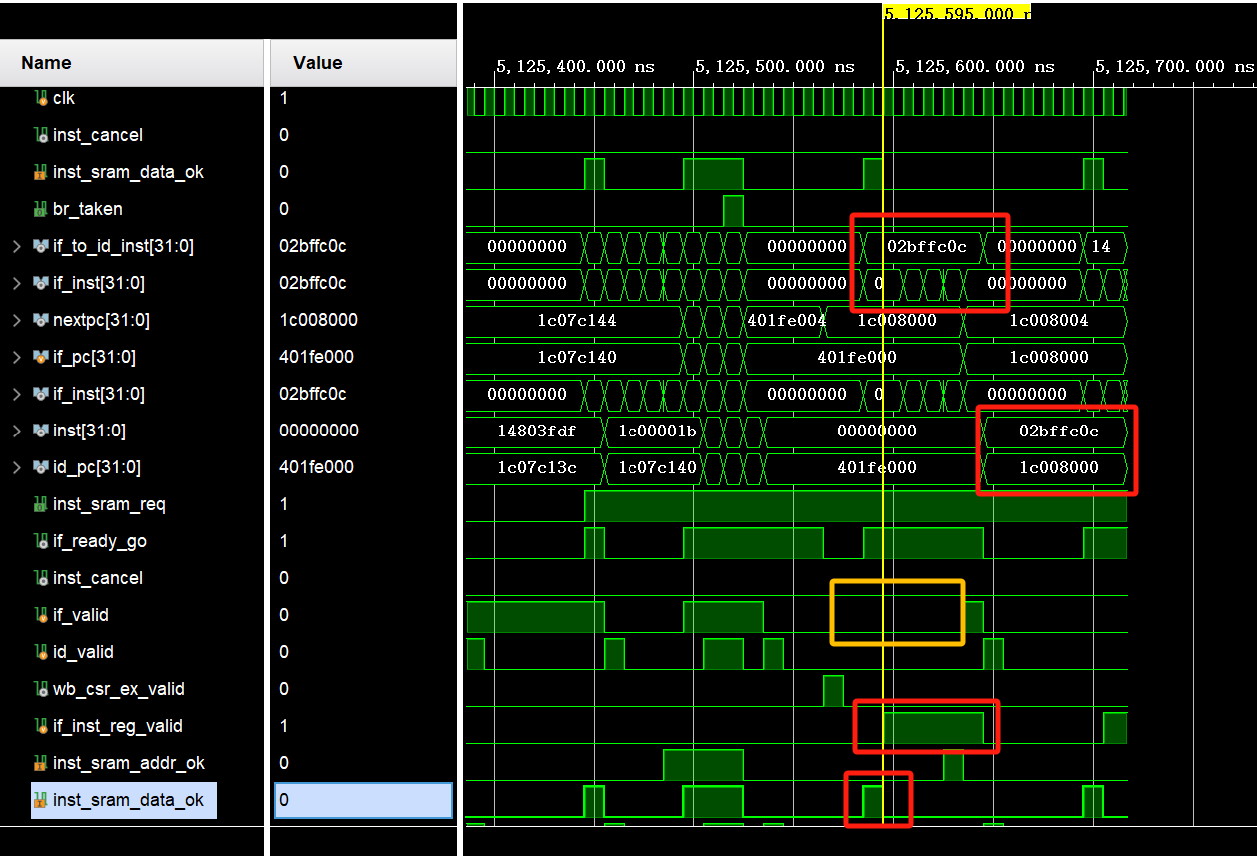


图72 错误8对应的波形图

可以看到，很明显，ID流水级收到的指令与PC不对应。进一步查看原因，发现传递给ID流水级的指令来自if\_inst\_reg寄存器。于是进一步查看相关信号，发现inst\_sram\_data\_ok返回后，if\_inst\_reg\_valid拉高了。if\_inst\_reg\_valid和if\_inst\_reg寄存器一起使用，if\_inst\_reg设置的目的是为了处理data\_ok返回后，但是ID流水级没有空闲出来，所以用一个寄存器来存储获得指令。

所以理论上，只有当返回的指令是有用的才需要我们。但是通过波形图可以看出，此时if\_valid拉低了，这表明此时返回给IF的指令是无效的，但if\_inst\_reg\_valid却错误地拉高，if\_inst\_reg也错误地更新，导致传递给ID的指令出错，于是查看if\_inst\_reg\_valid和if\_inst\_reg的代码，发现了错误，代码如下：

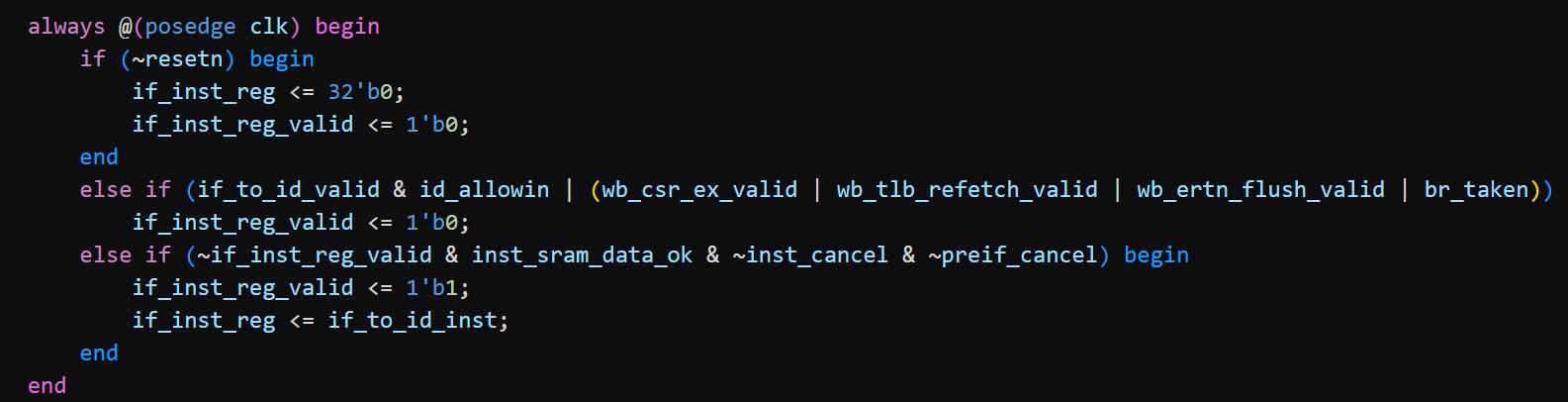


图73 错误8中if\_inst\_reg\_valid和if\_inst\_reg更新逻辑

* + - 1. 错误原因

IF流水级的if\_inst\_reg设置的目的是为了处理data\_ok返回后，但是ID流水级没有空闲出来，用一个寄存器来存储获得指令。但其更新逻辑没有考虑if\_valid，导致if\_vaild拉低时，虽然此时返回给IF的指令是无效的，但if\_inst\_reg\_valid却错误地拉高，if\_inst\_reg也错误地更新，进而导致传递给ID的指令出错。

* + - 1. 修正效果

把if\_inst\_reg\_valid和if\_inst\_reg的更新逻辑考虑上if\_valid即可，如下所示：

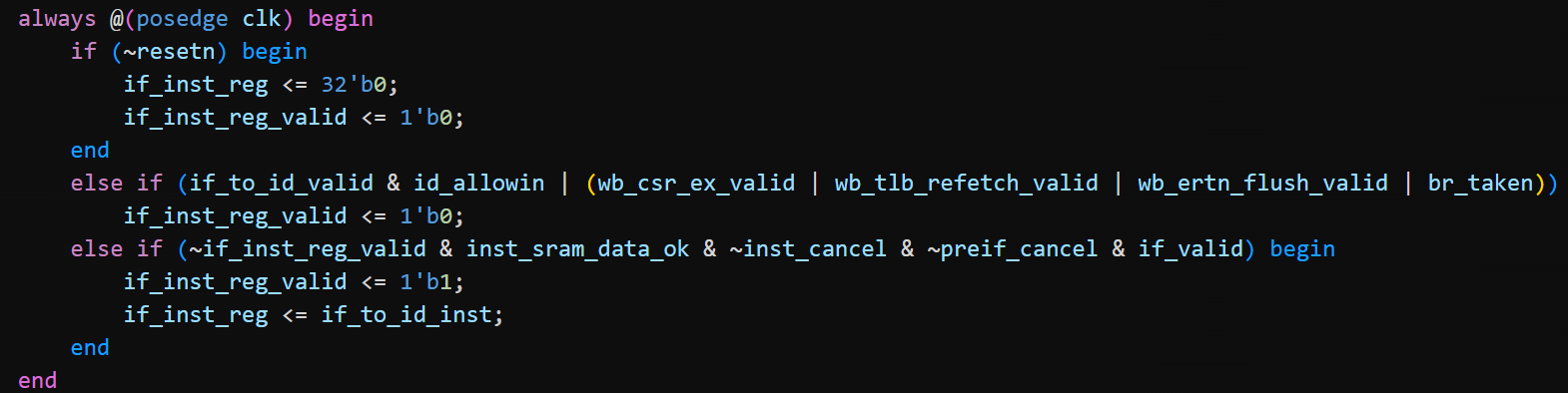


图74 错误8修改后代码

该方法有效，来到下一个bug。

* + 1. 错误9：exp21中ID流水级的valid信号对br\_taken处理出错
       1. 错误现象

运行仿真后，console报错如下：

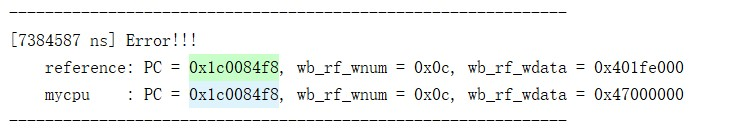


图75 错误9对应的Console报错

* + - 1. 分析定位过程

首先查看报错位置的汇编代码：

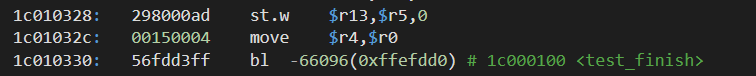


图76 错误9对应的汇编指令

可以发现，此时出错的是一个bl指令，而且这已经到了测试的最后几个指令了，所以此时bl居然能够出现报错，也是让我们很是吃惊，但不管怎么说，先查看波形图再说，如下所示：

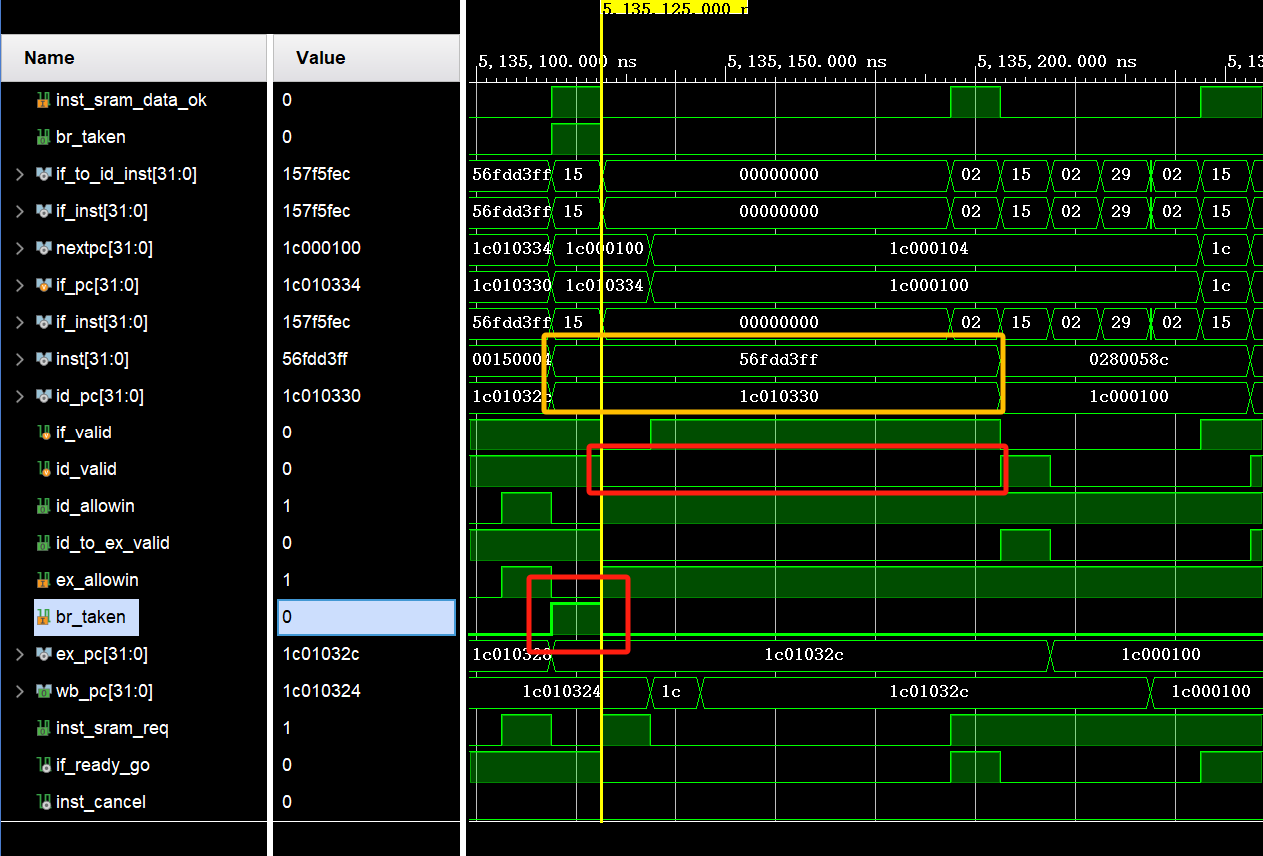


图77 错误9对应的波形图

可以发现，出错的原因很明显是这个bl指令根本没有进行处理。然后进一步进行查看，可以发现，原因是因为此时br\_taken拉高了，导致id\_valid被拉低，指令没有传入EX流水级进行后续处理。

很明显，这是id\_valid更新逻辑的问题，在很早以前的实验中，由于每个流水级均会1拍做完，此时对br\_taken的处理是其拉高了便在下拍把id\_valid拉低，这样刚好下一条指令来到ID流水级时，被作为无效指令处理。后面的实验中，流水线的阻塞越来越多，IF的时间也越来越长，这种方法已经不适合，但我们把br\_taken放在IF进行进一步处理，对于ID的这个逻辑并没有修改。

所以这就导致，当ID流水级停留不止1拍时，本来应该进行处理的bl和jirl指令会被无效掉而不被处理，导致出错。原先的id\_valid更新逻辑如下：

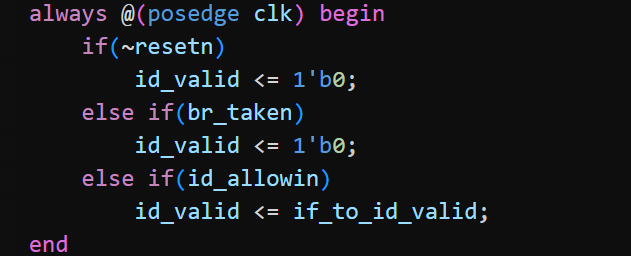


图78 错误9中id\_valid更新逻辑

这个错误也让我们十分震惊。因为这是一个很容易触发的bug，只有bl和jirl指令在ID停留超过一拍就会出现问题。以往的测试中有无数个bl和jirl指令，均没有出现对应的情况，所以这引起了我们的思考。

但其实稍微想想也很能理解，在类SRAM和AXI转接桥实验后，IF的取值延迟便特别大，每次成功取值要经历漫长的握手等待。虽然ID，EX和MEM会有很多的阻塞逻辑，但由于IF的取值延迟，流水线十分稀疏，往往上一条指令在WB流水级处理完了，IF的下一条指令还没握手成功，这也是为什么之前这么多次实验都没有出现问题的原因。

但其实本组成员在这想提一点测试程序的改进，比如在这些容易出现问题的指令前面加上div指令，由于div的执行拍数比较长，这样就可以看到阻塞下流水线有没有出错，上面本组出现的问题也会提前暴露。

* + - 1. 错误原因

id\_valid更新逻辑存在错误，在很早以前的实验中，由于每个流水级均会1拍做完，此时对br\_taken的处理是其拉高了便在下拍把id\_valid拉低，这样刚好下一条指令来到ID流水级时，被作为无效指令处理。在本次实验中，这样的逻辑会导致，当ID流水级停留不止1拍时（即ID流水级被阻塞时），本来应该进行处理的bl和jirl指令会被无效掉而不被处理，导致出错。

* + - 1. 修正效果

把id\_valid更新逻辑修改正确即可，在下一拍IF内容就会传递给ID，且br\_taken的时候，才把id\_valid赋值为0，如下所示：

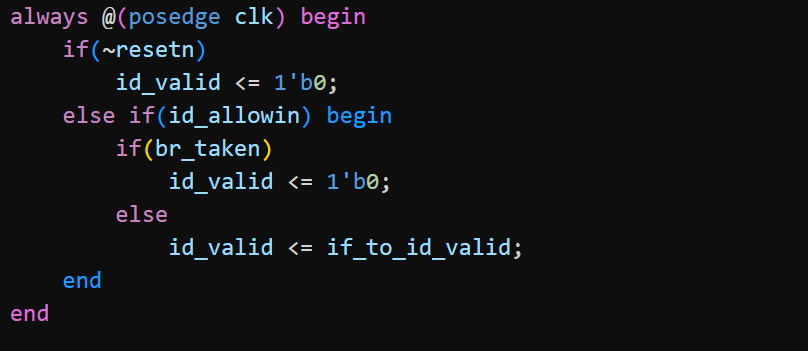


图79 错误9修改后代码

该方法有效，exp21通过。

1. 实验总结

本实验实现了一个二路组相连的Cache模块，并在五级流水线中集成了ICache。使我们对理论课上介绍的Cache的结构与工作方式有了更好的理解，也学习了Cache的具体实现方式。讲义上对Cache模块的设计给出了比较详细的介绍，对如何集成ICache的介绍也很多，因此读懂讲义后实现起来难度并不大。

在集成ICache的时候，发现了一些在之前实验中流水线中存在的潜在bug。由于AXI总线实验后，IF流水级的取指延迟较高，导致很多时候上一条指令进入到了WB级，下一条指令的取指还没有握手成功，所以在以往的实验中埋下了部分流水线的bug。在添加ICache后，取指的效率大大提高，因此流水线中的指令也更加密集，所以以前一些潜在的问题都进行了解决。

在添加Cache后，由于取指需要经历的路径变的更长了（以前AXI转接桥通过寄存器把组合路径断开了），所以本次实验后时序有一定的下降，但也不影响上板的结果，经过我们的多次测试，每个种子都能100%通过。

1. 课程结语

本学期的体系结构研讨课实验差不多也就要结束了。回顾整个学期，我们实现了五级流水，完成了对中断异常的支持，实现了AXI转接桥，添加了对页表TLB和对虚实地址转换的支持，最后实现了cache。至此，我们基本完成了在设计的CPU上运行操作系统的绝大部分要求，这是我们在计算机组成原理研讨课中想都不敢想的工程。

通过整个学期的实验，我们对于体系结构有了更加深刻的认识。特别是小组完成的机制，让大家能有更多的讨论，虽然身边很多组都是一个人做一次实验，但对于我们组来说，由于我们组的成员都是一个寝室的，交流特别方便，所以每次实验从设计到代码撰写再到debug的环节，基本都是大家一起做的。这让我们能想出更多好的解决方法，尤其是在设计时，往往会提出很多的想法，然后从中选择一种最合理的进行实现，我们觉得这也正是分组实验的目的与意义。以往的单人实验，由于大家的代码框架并不相同，所以相互之间的交流并不算多，很多时候的讨论也没能有很高的参考价值，所以总的来说我们觉得分组是一个挺好的机制。

然而，身边也有很多组并不如我们顺利，可能他们的某位组员干的事情比较少，导致其他人压力过大；又或者由于人员比较分散，很难面对面交流，只能把每个任务交由单独的人员完成，然后需要花很长的时间理解队友的代码，别人的部分出现的bug也难以解决。所以对于分组机制，可能需要老师多一点建议，也可能需要多一点约束，让每个人工作量尽可能相同。

对于测试程序，可能希望能检查的更加全面一点吧。很多流水线设计上的缺陷在添加cache前都没有暴露。记得我们组也给其它组提出过不少潜在的bug，而我们自己也尽可能保证每次实验中的设计尽可能完善，但在最终的cache集成实验中，还是出现了几个以前的潜在bug。通过我们的讨论，我们觉得其中很多都可以在特定指令前加上div指令来检测出来（虽然这样仿真时间可能就很长了），所以也算是给一点测试程序上的小建议吧。然后针对讲义，其实也存在或多或少的小错误，我们组发现的小错误也都在piazza上提出来过。

最后，真心感谢授课老师和每一位尽职尽业的助教老师们。在每周的研讨课上，对于我们组的代码提出过一些很有价值的改进办法，也对我们提出的问题和疑问给出了很认真的回答，整个过程也是比较轻松愉快的，既学到了很多知识，也没有感受到很紧张的氛围。也感谢每一个对我们实验提出建议和对我们的设计给与帮助的同学。

完结撒花，祝老师们和同学们新的一年事事如意！