**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实现对内存读写操作的缓存，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 cache模块接口信号

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| **CPU interface** | | | |
| valid | IN | 10 | 请求有效信号 |
| op | OUT | 1 | 请求指令码（1写0读） |
| index | OUT | 4 | 地址的index域（addr[11:4]） |
| tag | OUT | 20 | 经虚实地址转换后的物理地址形成的tag |
| offset | OUT | 6 | 地址的offset域 |
| wstrb | OUT | 2 | 写字节使能信号 |
| wdata | OUT | 2 | 写数据 |
| addr\_ok | OUT | 1 | 地址传输OK握手信号 |
| rdata | OUT | 1 | 数据传输OK握手信号 |
| **AXI interface**  **Read port** | | | |
| rd\_req | OUT | 1 | 读请求有效信号 |
| rd\_type | OUT | 3 | 读请求类型 |
| rd\_addr | OUT | 32 | 读请求起始地址 |
| rd\_rdy | IN | 1 | 读请求能否被接收的握手信号 |
| ret\_valid | IN | 1 | 返回数据有效信号 |
| ret\_last | IN | 2 | 返回数据是一次读请求对应的最后一个返回数据 |
| ret\_data | IN | 32 | 读返回数据 |
| **Write port** | | | |
| wr\_req | OUT | 1 | 写请求有效信号 |
| wr\_type | OUT | 3 | 写请求类型 |
| wr\_addr | OUT | 32 | 写请求起始地址 |
| wr\_wstrb | OUT | 4 | 写操作字节掩码 |
| wr\_data | OUT | 128 | 写数据 |
| wr\_rdy | IN | 1 | 写请求能否被接收的握手信号 |

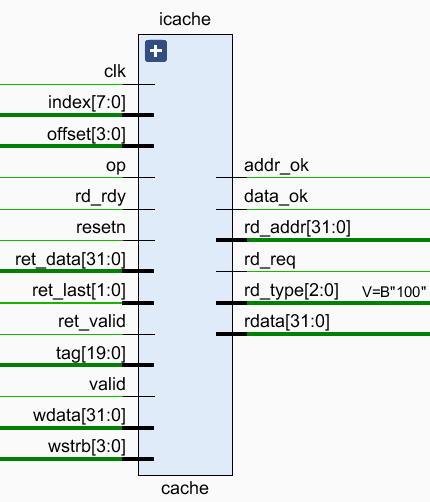


图1 icache模块接口

* + 1. 功能描述

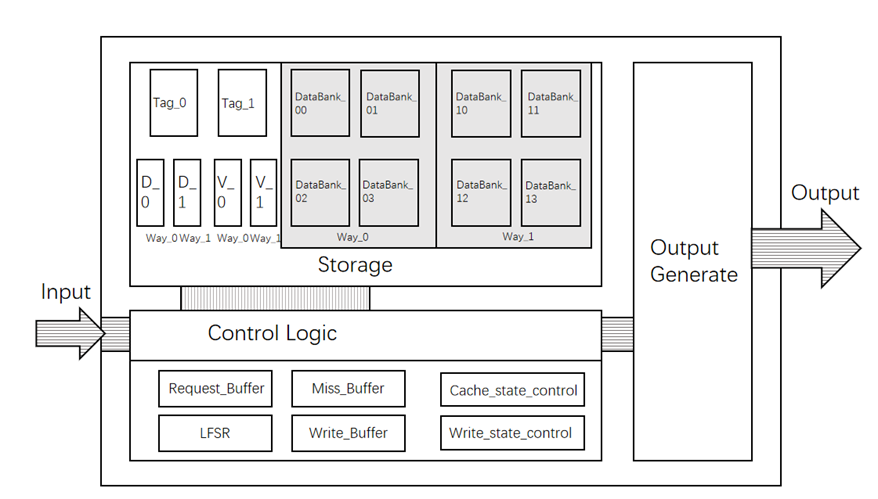


图2 cache结构示意图

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IP核中的Block RAM实现Tag与Valid表（合并在同一块RAM中）和Data表，通过寄存器组实现Dirty表。

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。此外还包含两个暂时存放数据的buffer：Request buffer和Write 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×21 bit的TagV表、两张256项×1 bit的D表以及两张256项×128 bit的data表。每张表都有256项，对应256个Cache行。各表的实现方式分别为：

TagV表由两块深度为256，宽度为21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，分为4个bank，拼起来构成128位的数据。

D表由两个位宽为256的寄存器实现。

三张表都需要初始化，由寄存器实现的D表在复位信号拉高初始化，由同步RAM实现的TagV表与Data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**

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ret\_last拉高，表示数据全部从AXI返回后切换回IDLE状态。

1. **Write buffer 状态机**

Write buffer 状态机负责在写命中时将write buffer中的数据写入cache。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，且查找命中时，hit\_write拉高，状态机从IDLE状态进入WRITE状态。此时将同步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。

hit_write

图4 hit\_write逻辑

* + 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. **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，记录替换路与来自request buffer的index， offset， wstrb和wdata。

* + 1. 替换路算法

采用LRU算法计算应被替换的路。由于只有两路，因此在一路命中时将对应行的替换路设为另一路即可。

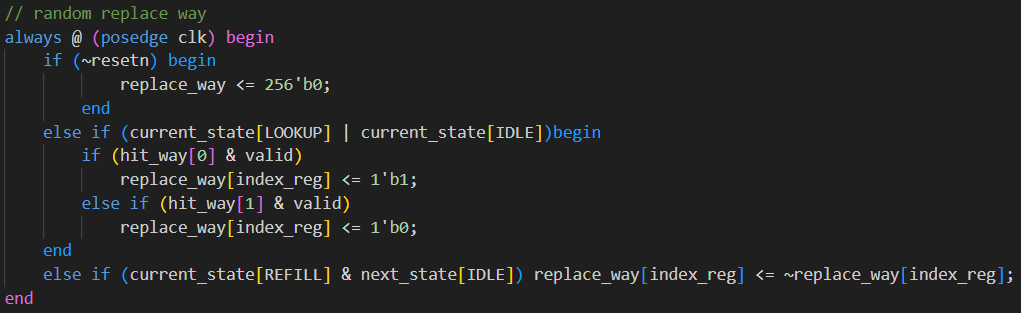


图5 LRU替换算法

* + 1. RAM控制信号

TagV RAM的we信号在ret\_valid=1，ret\_last=1时拉高，即收到了AXI返回数据的最后一个字。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。TagV RAM的写数据tag来源为request buffer中记录的tag，valid必为1。

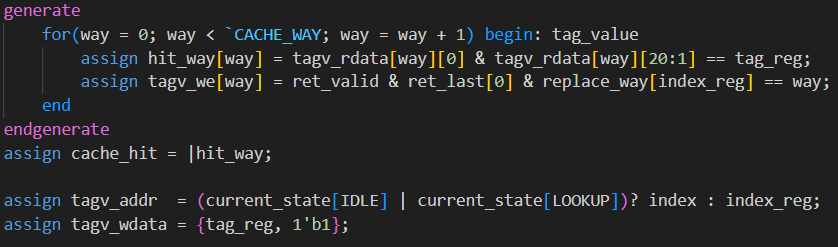


图6 TagV RAM控制信号

Data Bank RAM在内存每返回一个字时将它写入对应的bank中。We信号在写状态机为WRITE状态且bank与way对应时根据写字节掩码拉高，在收到AXI返回数据有效时根据bank和way对应拉高。addr信号来自于CPU输入的（当拍）或request buffer中缓存的index域。wdata在写状态机写时为write buffer的写数据，在AXI返回数据时根据写字节掩码选择是否保存CPU传来的写数据。

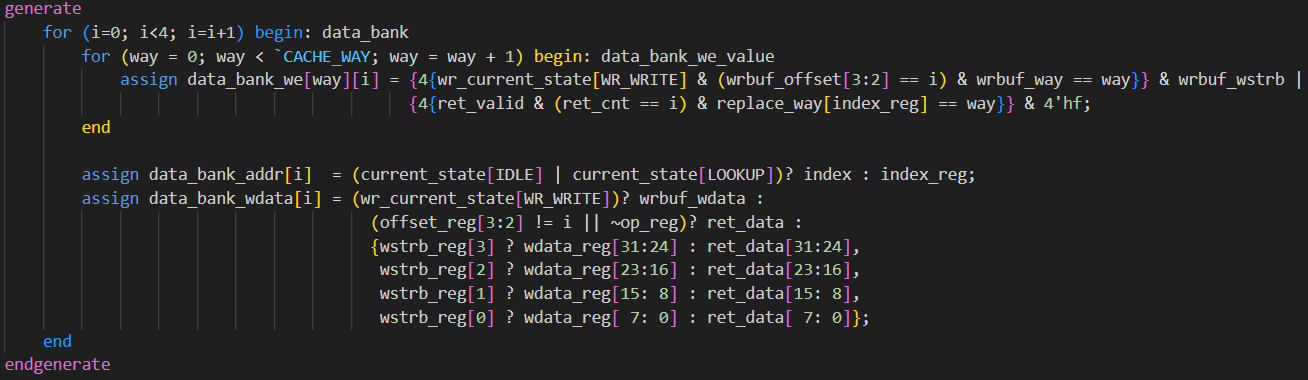


图7 Data Bank RAM控制信号

* 1. 输出模块

cache输出给cpu的信号包括addr\_ok、data\_ok和rdata。生成逻辑如下：

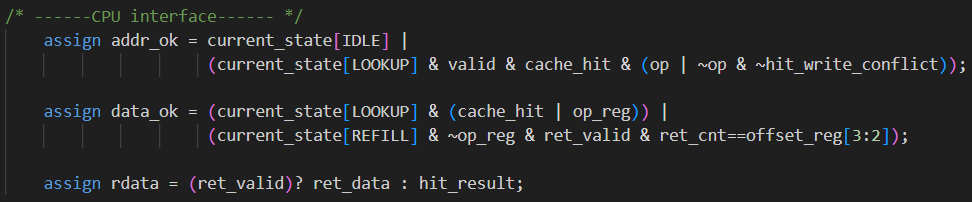


图8 cache向CPU输出信号的逻辑

addr\_ok在主状态机IDLE状态一直拉高，以及在LOOKUP状态命中且为写指令或读指令且没有冲突的时候拉高。

Data\_ok在LOOKUP状态且查找命中或为写指令时拉高，或者在REFILL状态，为读指令，AXI返回数据有效，且为offset为对应bank时有效。

Rdata在未命中时（进入REFILL并收到返回值有效）为AXI返回的数据，否则为命中查找到的数据。

在输出到axi的信号中，rd\_type与wr\_type恒为3’b100，对应cache行。rd\_addr来自request buffer。rd\_req在主状态机处于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。wr\_addr和wr\_data来自需要被替换的cache行。wr\_wstrb恒为4’hf，写字节全部有效。

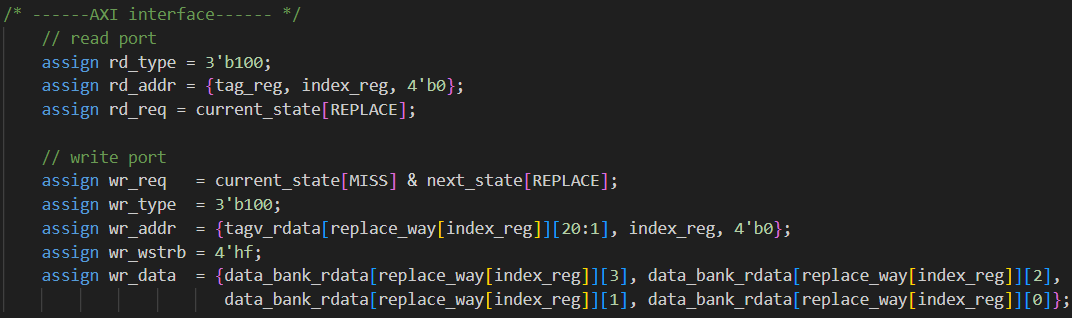


图9 cache向AXI输出信号的逻辑

* 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2 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 |

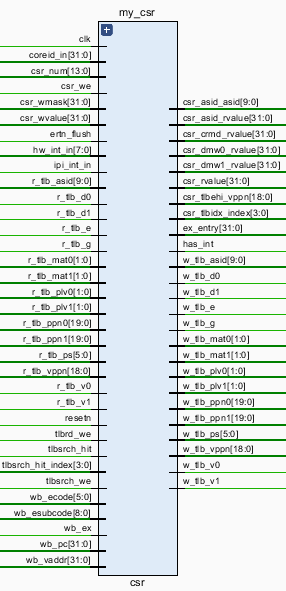


图10 TLB模块接口

* + 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4设计：类SRAM-AXI转接桥AXI\_bridge
     1. 工作原理

类SRAM-AXI转接桥中，数据部分与CPU流水级的类SRAM信号相连，指令部分与icache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3 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** | | | |
| icache\_req | IN | 1 | icache请求信号 |
| icache\_type | IN | 3 | icache读请求类型 |
| icache\_addr | IN | 32 | icache请求地址 |
| icache\_rd\_rdy | OUT | 1 | icache当前的请求已握手成功 |
| icache\_ret\_valid, | OUT | 1 | icache该次请求的数据传输有效 |
| icache\_ret\_last | OUT | 1 | icache本次burst传输完毕 |
| inst\_sram\_rdata | OUT | 32 | icache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，主要修改的是读请求通道和读响应通道。

首先是读请求通道，AXI转接桥要实现支持burst传输。在转接桥向AXI总线传输的信号中，需要更改的是arlen。信号arlen+1代表的是一次读取的数据包个数，因此当读请求是icache发来的时候，arlen要置为8’b11；当读请求是CPU的EX级发来的时候，arlen仍然要为0，其余信号与以前基本一致。

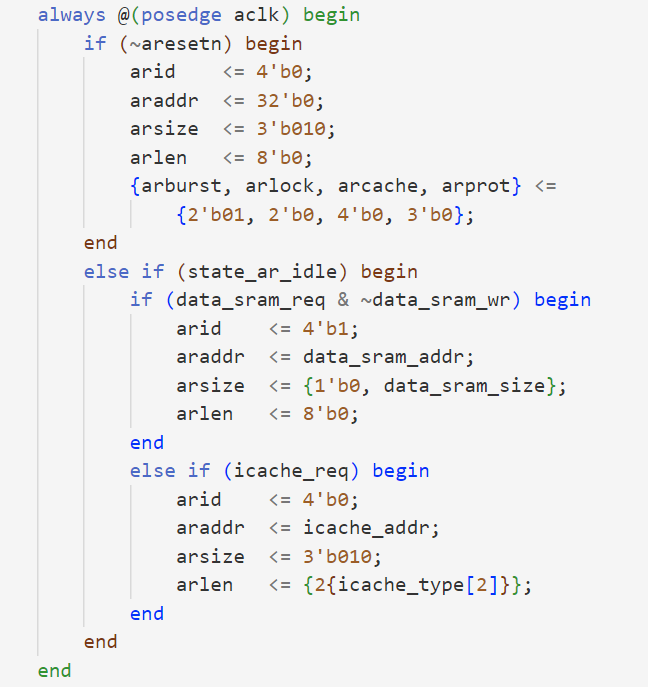


图11 读请求通道连接到AXI总线的部分信号

还需要进行更改的是icache\_rd\_ready信号和icache\_ret\_last信号，它们分别与以前的inst\_sram\_addr\_ok和inst\_sram\_data\_ok比较像：

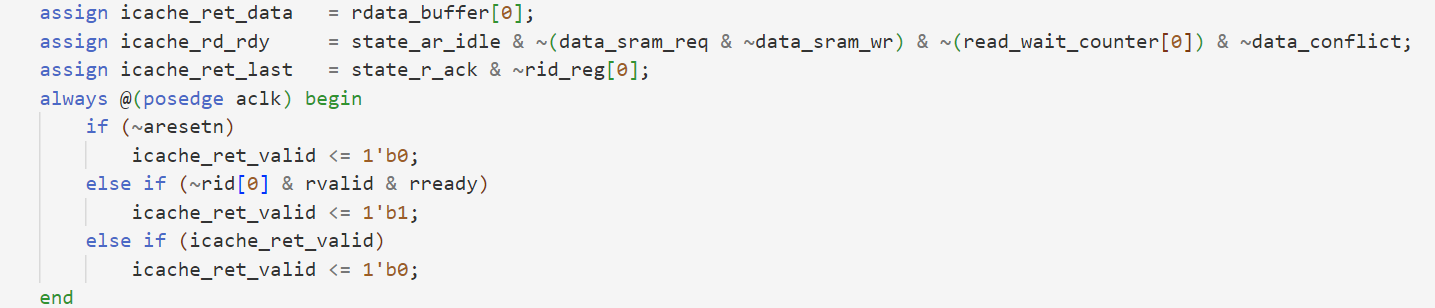


图12 icache\_rd\_ready、 icache\_ret\_valid和icache\_ret\_last信号

不过有一点小区别的是，作为握手信号，icache\_rd\_rdy不是在处理一个读请求后才拉高，而实在读请求状态机空闲的时候拉高，与以前更多作为一个返回信号的inst\_sram\_addr\_ok相比，icache\_rd\_rdy更像一个握手信号了。

为了支持burst传输，还需要增加icache\_ret\_valid信号。每当读取到一个数据，就在下一拍将这个信号拉高。等到数据传输完毕后，再拉高icache\_ret\_last信号。

* 1. 重要模块5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4 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 |

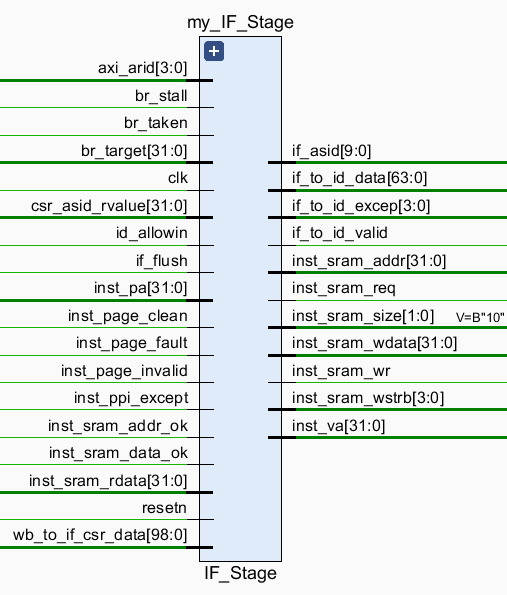


图13 IF模块接口

* + 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

2023.12.14 10：00-12：00，15：00-18：00 完成exp20的debug

2023.12.24 23：00-次日2：00 完成exp21的设计

2023.12.25 8：00-12：00进行exp21的debug

贾城昊

2023.12.24 23：00-次日2：00 完成exp21的设计

2023.12.25 10：00-12：00进行exp21的debug

2023.12.26 18：00-23：00完成exp21的debug

牛浩宇

2023.12.12 10：00-16：00 进行exp20的设计

2023.12..13 19：00-22：00完成exp20的设计

* 1. 错误记录
     1. 错误1：exp21中AXI转接桥的read\_wait\_counter信号赋值有误
        1. 错误现象

程序无法正常停下，console界面如下：

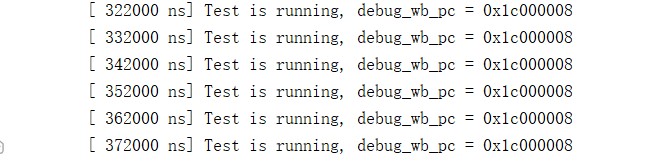


图14 错误1对应的Console界面

* + - 1. 分析定位过程

查看波形图，其中映入眼帘中的有一个为Z的信号ret\_last，但是这不是问题所在。在讲义上，cache模块中的ret\_last是两位，但事实上它应该是一位。这个地方我们本以为会有后续的作用所以未修改，后来发现没有别的用处，因此更改其为了1位。

不过这不会导致错误，从r\_current\_state可以看出，卡死的原因是AXI转接桥中的读响应通道卡在了RDY阶段。

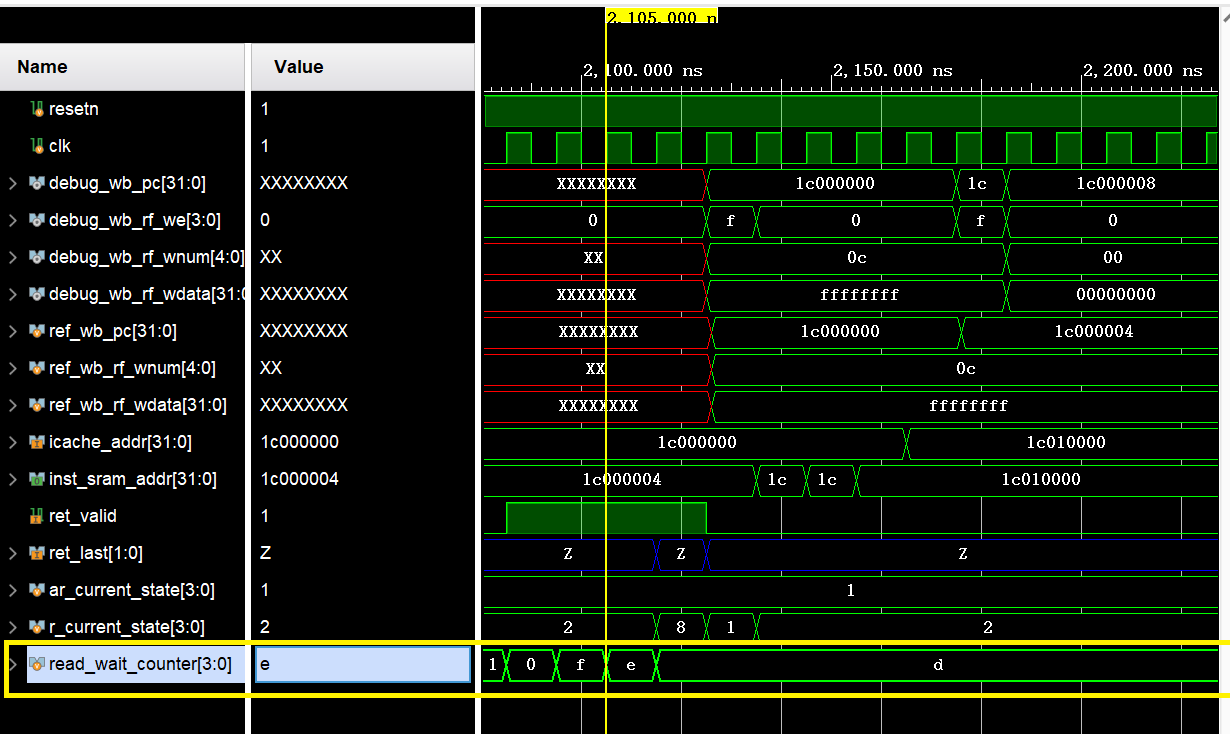


图15 错误1对应的波形图

我们查看读相应通道状态机的转换逻辑：

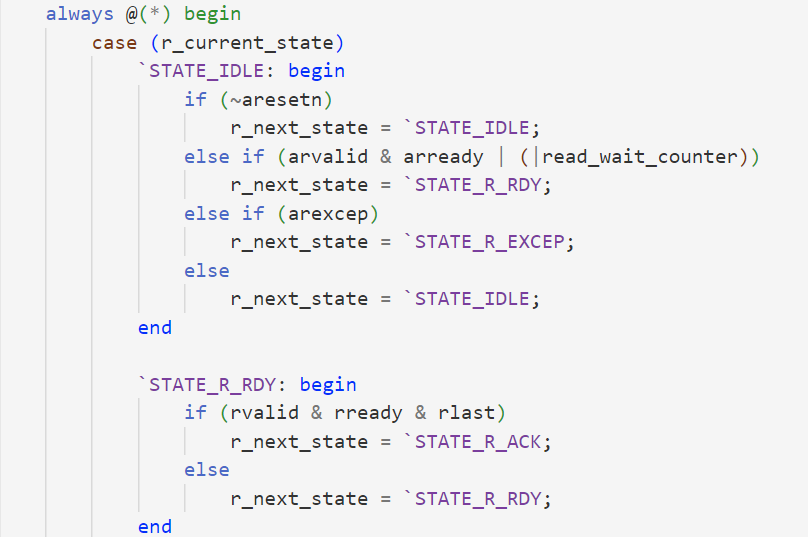


图16 AXI转接桥读响应通道状态机

查看相关信号，已经能看出一些端倪：读响应通道状态机之所以会从IDLE状态转为RDY阶段是因为read\_wait\_counter信号不为0而为0xd，而这是不可能的，因为在课程的设计中，rid只会有两种取值，也就说明同时最多只会有2个读操作未得到响应。虽然我们将read\_wait\_counter设置为了4位，但这只是为了和rid的位数匹配以便后续扩展，实际上它的值不应该超过2

查看read\_wait\_counter信号复制逻辑：

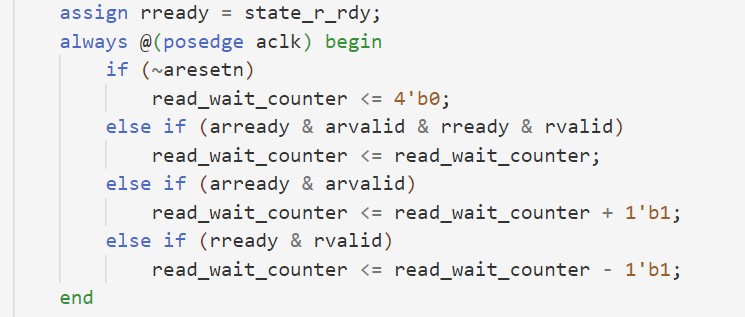


图17 read\_wait\_counter信号赋值逻辑

* + - 1. 错误原因

由此可以看出错误原因：只有当rvalid和rlast信号同时拉高，才能说明一次读操作结束，这与以前不同，这是因为这次需要支持burst传输，一次传输可能会有多次rvalid的拉高

* + - 1. 修正效果

由此可以看出错误原因：只有当rvalid和rlast信号同时拉高，才能说明一次读操作结束，这与以前不同，这是因为这次需要支持burst传输，一次传输可能会有多次rvalid的拉高

* + 1. 错误2：exp21中AXI转接桥的rd\_rdy信号赋值不正确
       1. 错误现象

程序无法正常停下，console界面如下：

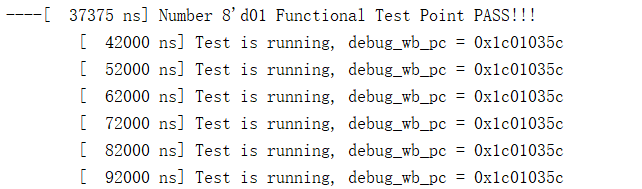


图19 错误2对应的Console界面

* + - 1. 分析定位过程

首先查看波形图，如下所示：

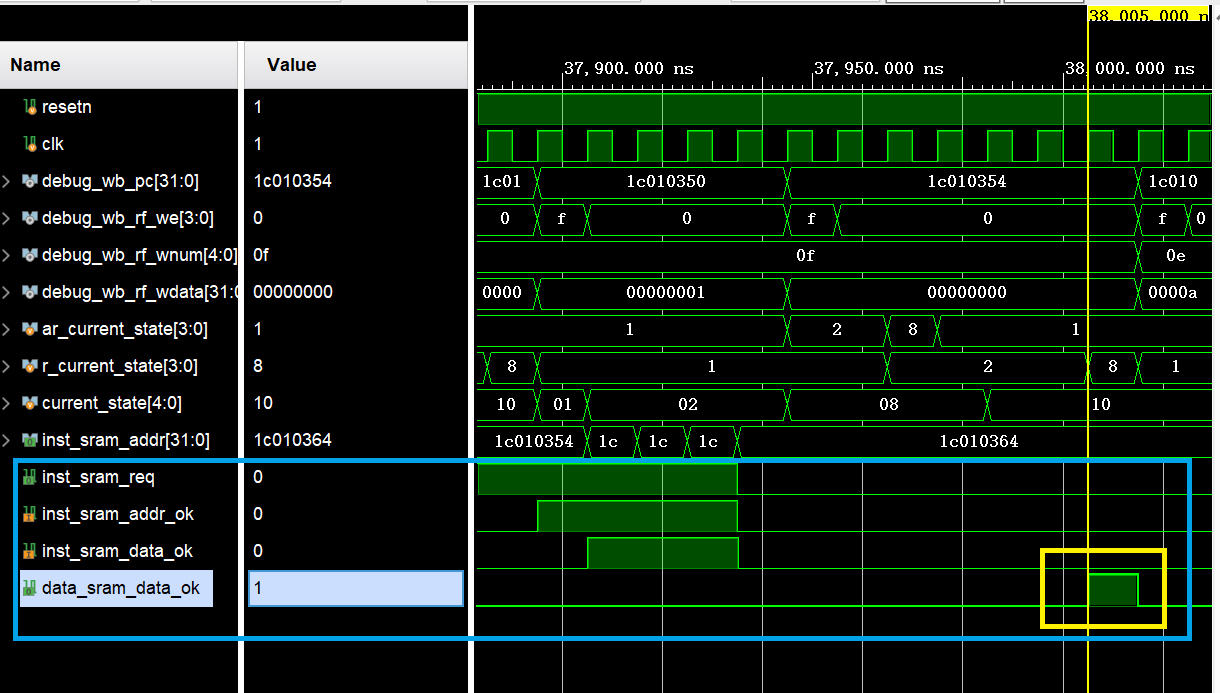


图20 错误2对应的波形图

通过波形图，我们意识到这个错误出现在icache不命中时，icache一直处于REFILL状态，等待AXI转接桥返回相应的ret\_valid和ret\_last信号。在出现这个错误时，我们还没将AXI转接桥的各种信号命名成与icache统一的，因此波形图中的inst\_sram\_addr\_ok即是连接到icache上的ret\_valid信号。可以看到，icache和AXI转接桥成功握手，但是迟迟没有返回inst\_sram\_data\_ok信号。

不久后返回的data\_sram\_data\_ok信号引起了我们的注意，我们发现导致这一错误的原因可能是当前AXI转接桥同时收到了指令和数据的读请求，在我们inst\_sram\_addr\_ok信号导致出错。

* + - 1. 错误原因

查看相关代码：

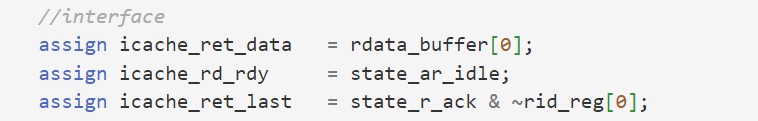


图21 错误2中icache\_rd\_rdy修改前赋值

可以看到，我们icache\_rd\_rdy的设计过于简单，没有考虑到各种情况，认为只要AXI转接桥的读请求状态机处于IDLE阶段就拉高icache\_rd\_rdy信号。而事实上，如果此时还有一个数据读请求，则不能拉高icache\_rd\_rdy信号。

* + - 1. 修正效果

当收到数据读请求时，不拉高rd\_rdy信号，修改后代码如下：

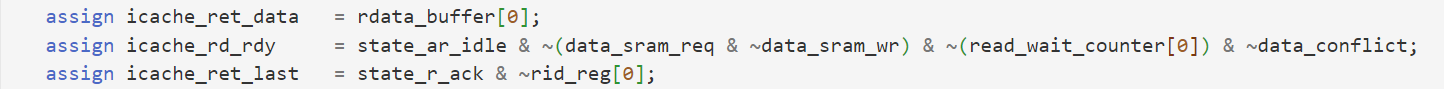


图22 错误2修改后代码(nextpc赋值逻辑)

该方法有效，来到下一个bug。

* + 1. 错误3：exp21中跳转指令后PC值出错
       1. 错误现象

Console报错如下

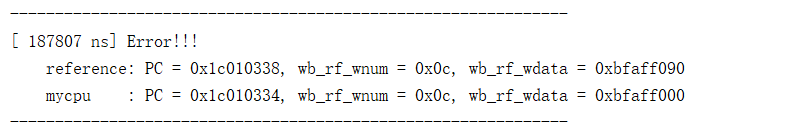


图23 错误3对应的Console报错

* + - 1. 分析定位过程

查看相关波形图，可以看出错误的直接原因是PC为0x1c010334的指令提交了两次，根据前一个PC得知这一个指令是从别处跳转过来的

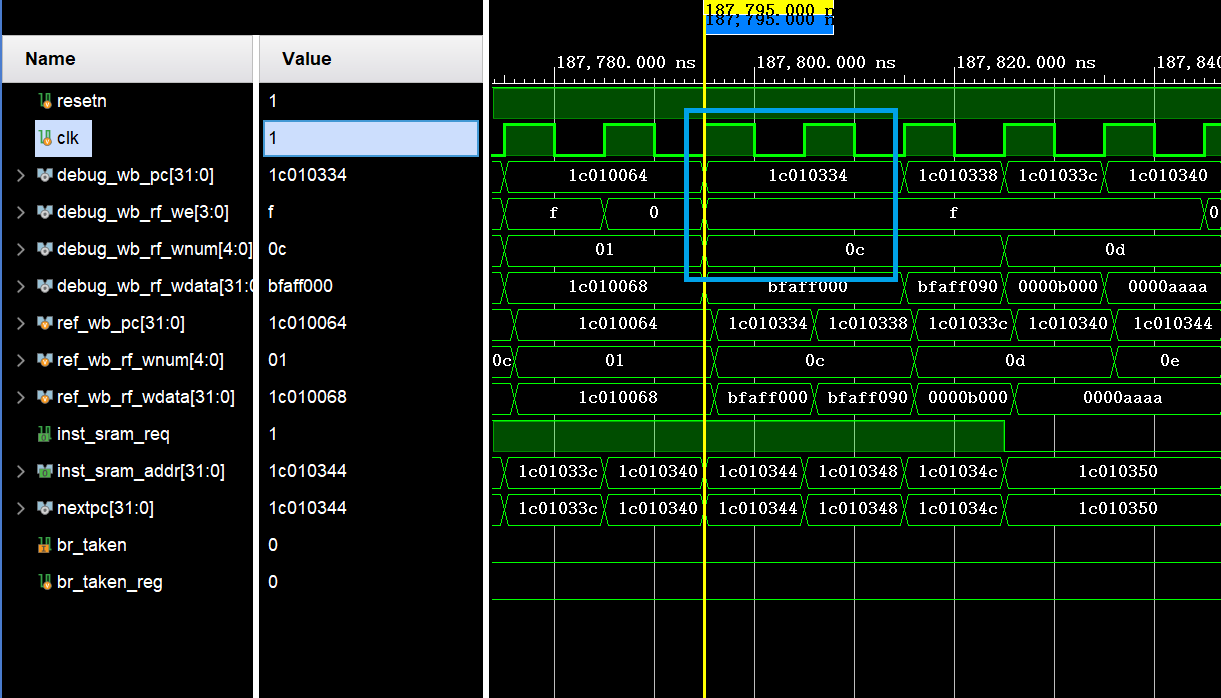


图24 错误3的相关波形图（1）

查看IF模块中对该地址的读指令相关操作：

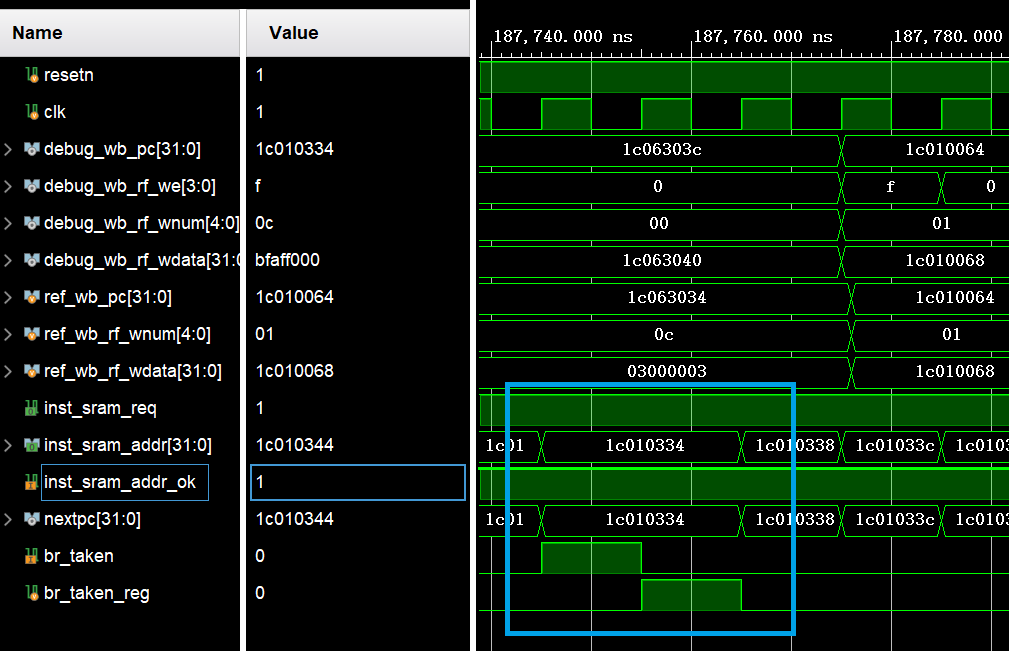


图25 错误3相关波形图（2）

通过波形图，可以看出错误的直接原因是IF模块对同一个地址0x1c010334发了两次请求。

* + - 1. 错误原因

会造成这一现象的原因还是cache的加入。前一条是一个跳转指令，此时br\_taken拉高，inst\_sram\_req和inst\_sram\_addr\_ok都拉高了，也就是请求地址为跳转目标地址的指令读请求已经成功握手，不需要再拉高br\_taken\_reg，使nextpc仍为跳转目标地址了。查看相关代码：

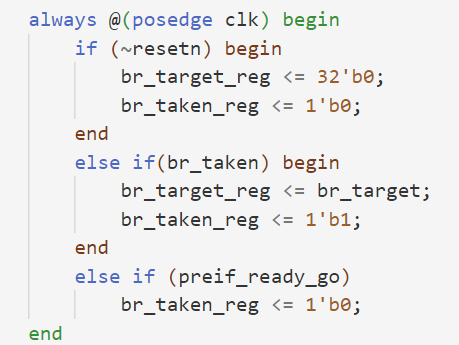


图26 错误3相关的代码

由此可以看出问题所在：当inst\_sram\_req和inst\_sram\_addr\_ok同时拉高时，就说明跳转目标地址已经握手成功，不应该再拉高br\_taken\_reg信号。

* + - 1. 修正效果

将br\_taken\_reg信号的赋值逻辑修改过来，如下所示：

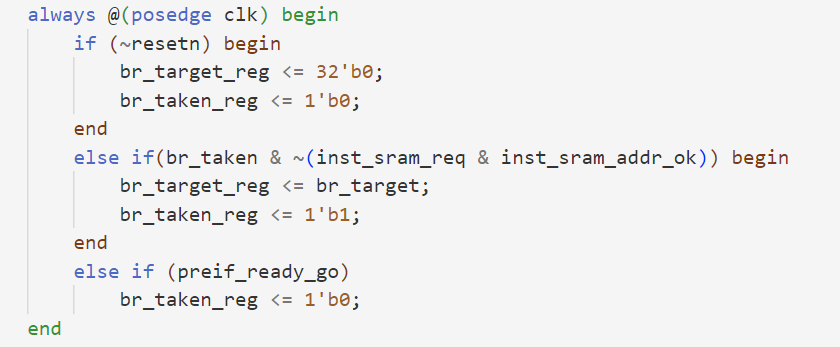


图27 错误3修改后的代码

该方法有效，来到下一处bug。

* + 1. 错误4：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：

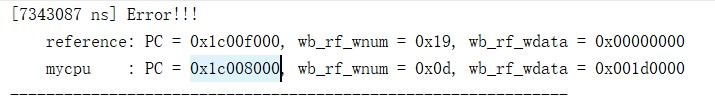


图28 错误4对应的Console报错

* + - 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：

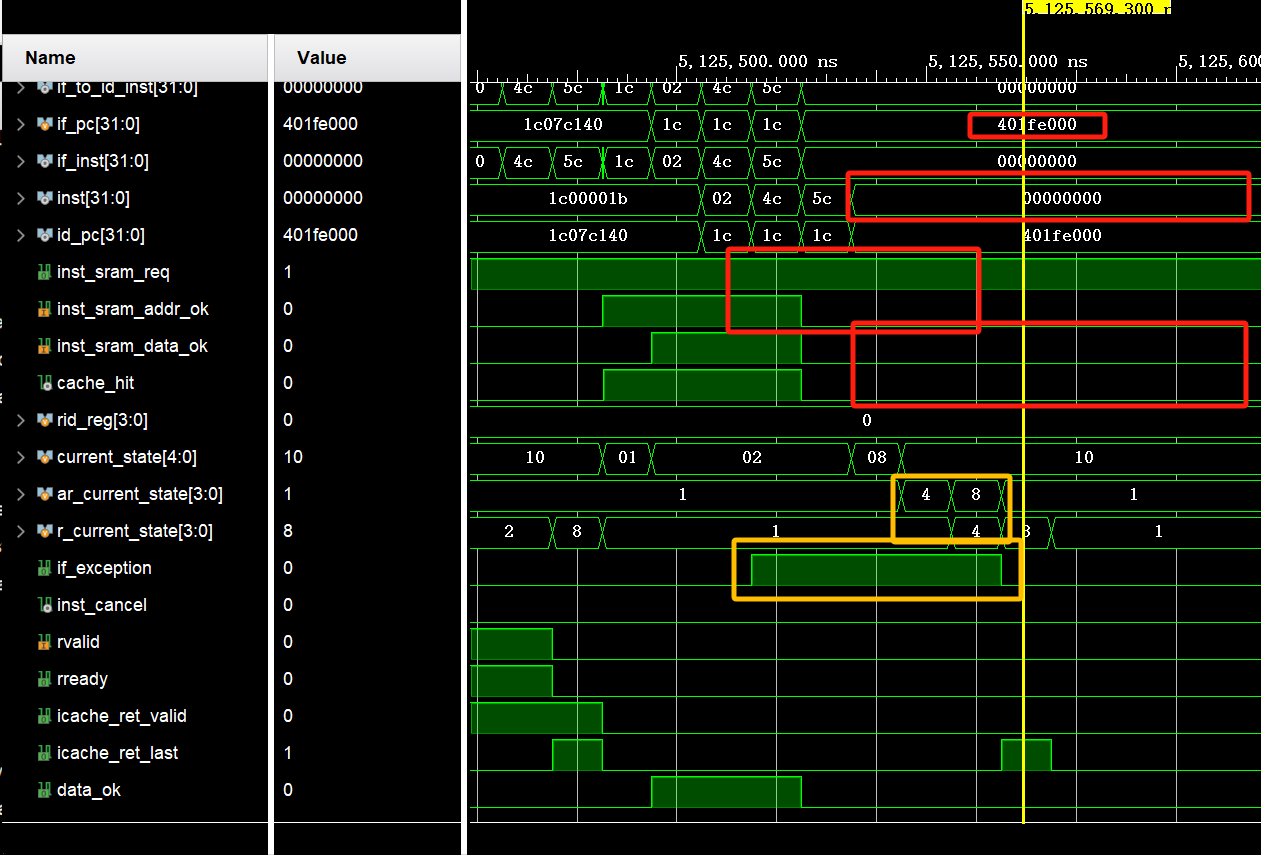


图29 错误4对应的波形图

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：

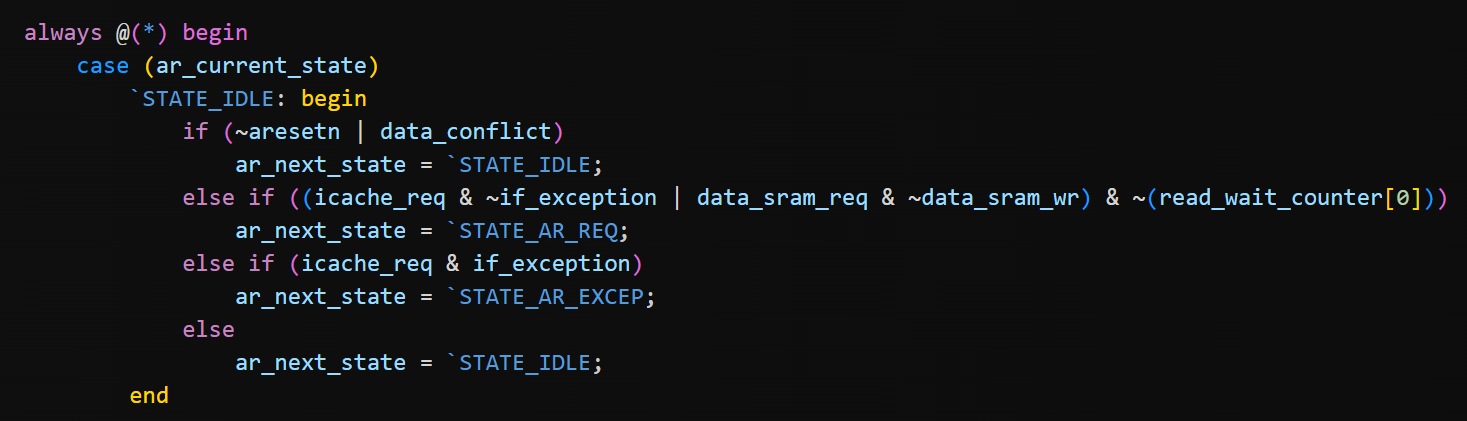


图30 错误4中AXI对异常的处理逻辑(1)

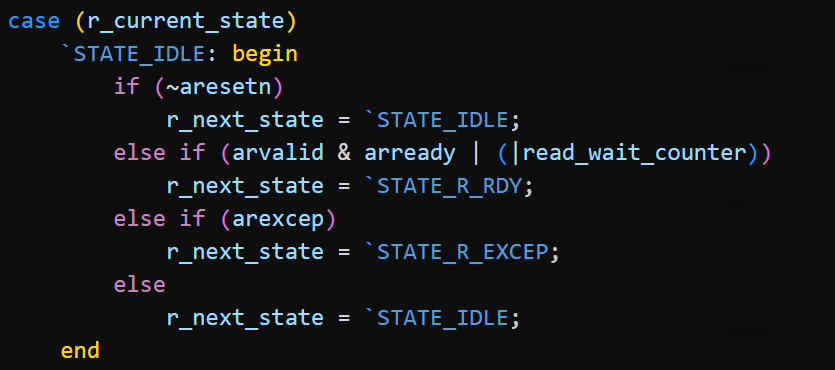


图31 错误4中AXI对异常的处理逻辑(2)

* + - 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：

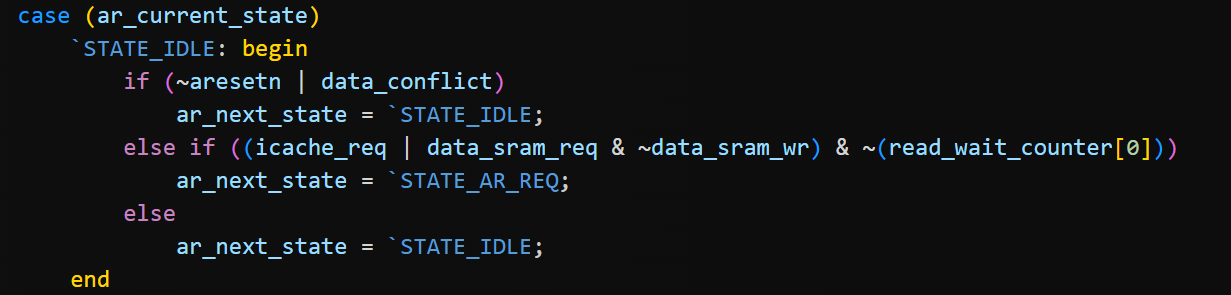


图32 错误4修改后代码（1）

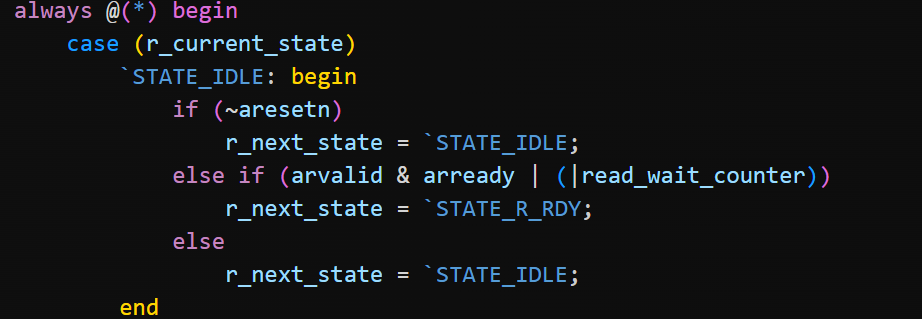


图33 错误4修改后代码（2）

该方法有效，来到下一个bug。

* + 1. 错误5：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：

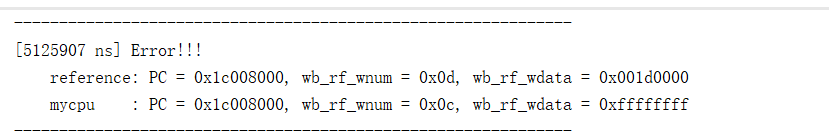


图34 错误5对应的Console报错

* + - 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：mym

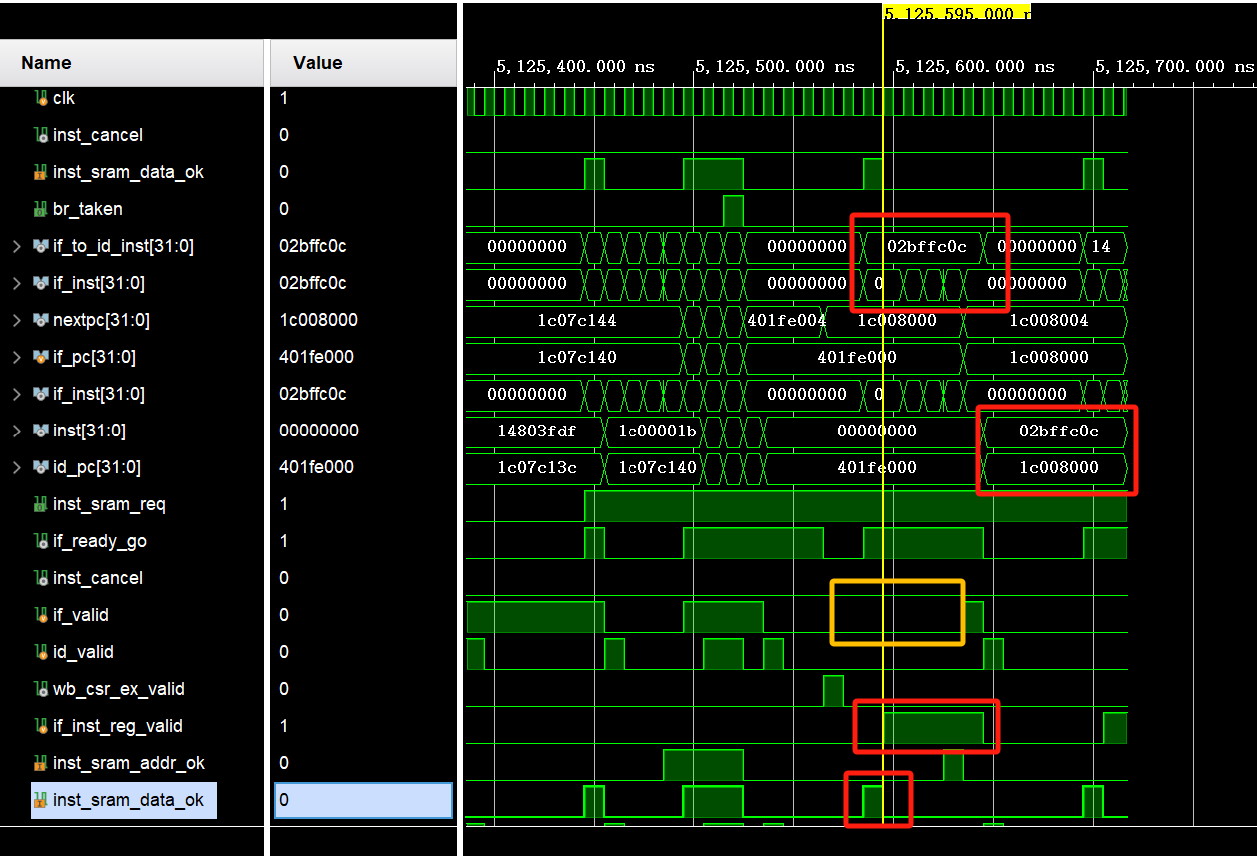


图35 错误5对应的波形图

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：

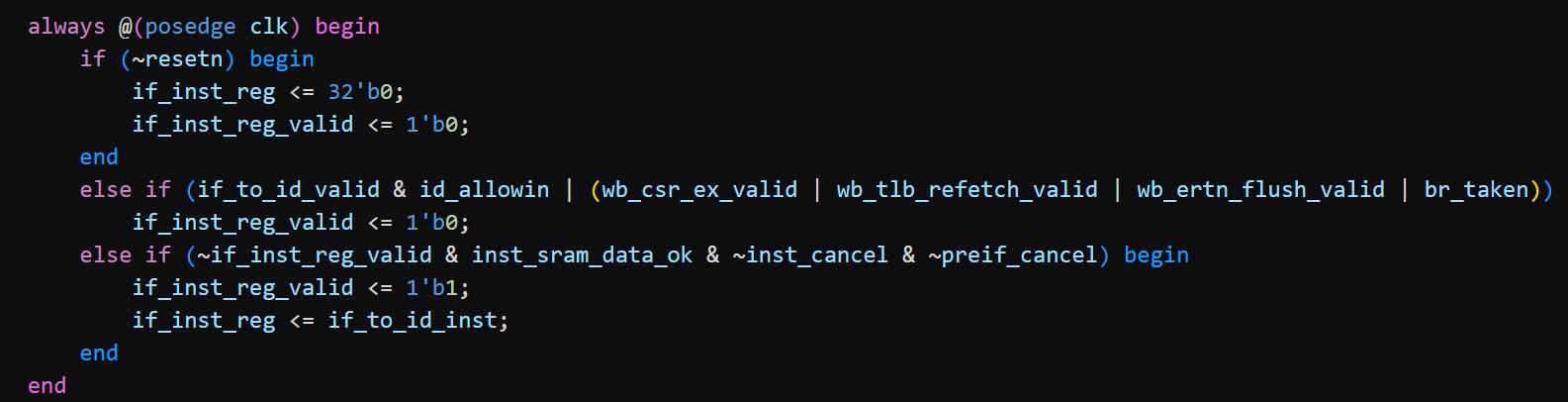


图36 错误5中if\_inst\_reg\_valid和if\_inst\_reg更新逻辑

* + - 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：

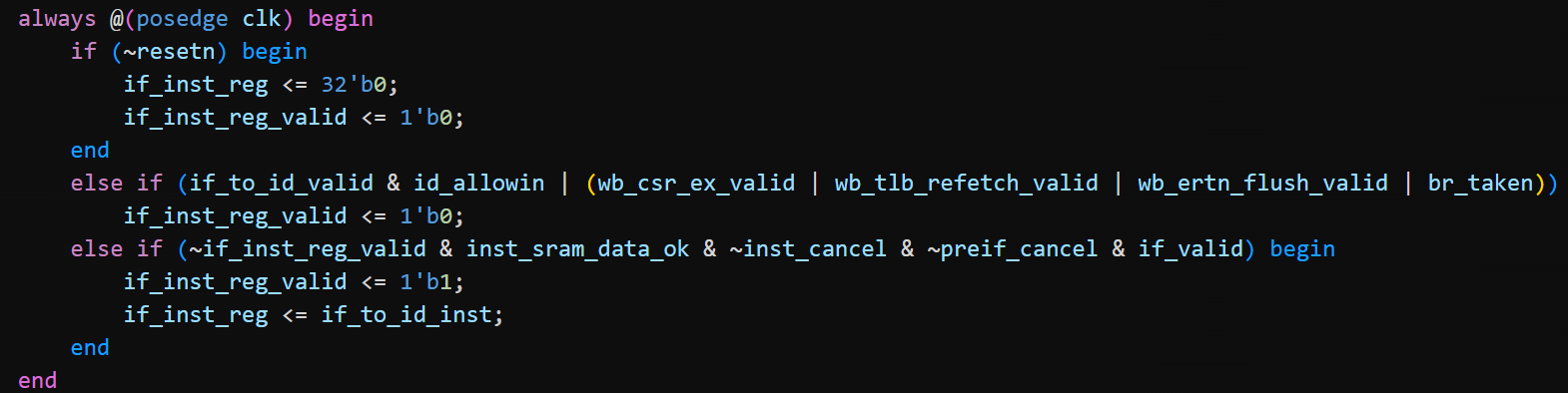


图37 错误5修改后代码

该方法有效，来到下一个bug。

* + 1. 错误6：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：

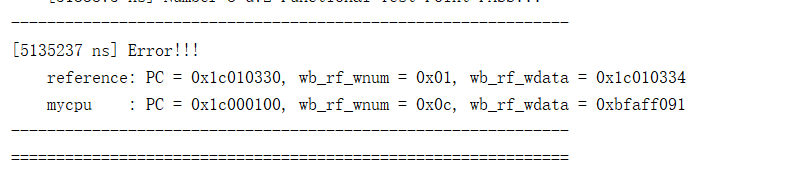


图38 错误6对应的Console报错

* + - 1. 分析定位过程

首先查看报错位置的汇编代码：

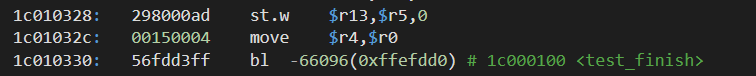


图39 错误6对应的汇编指令

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：

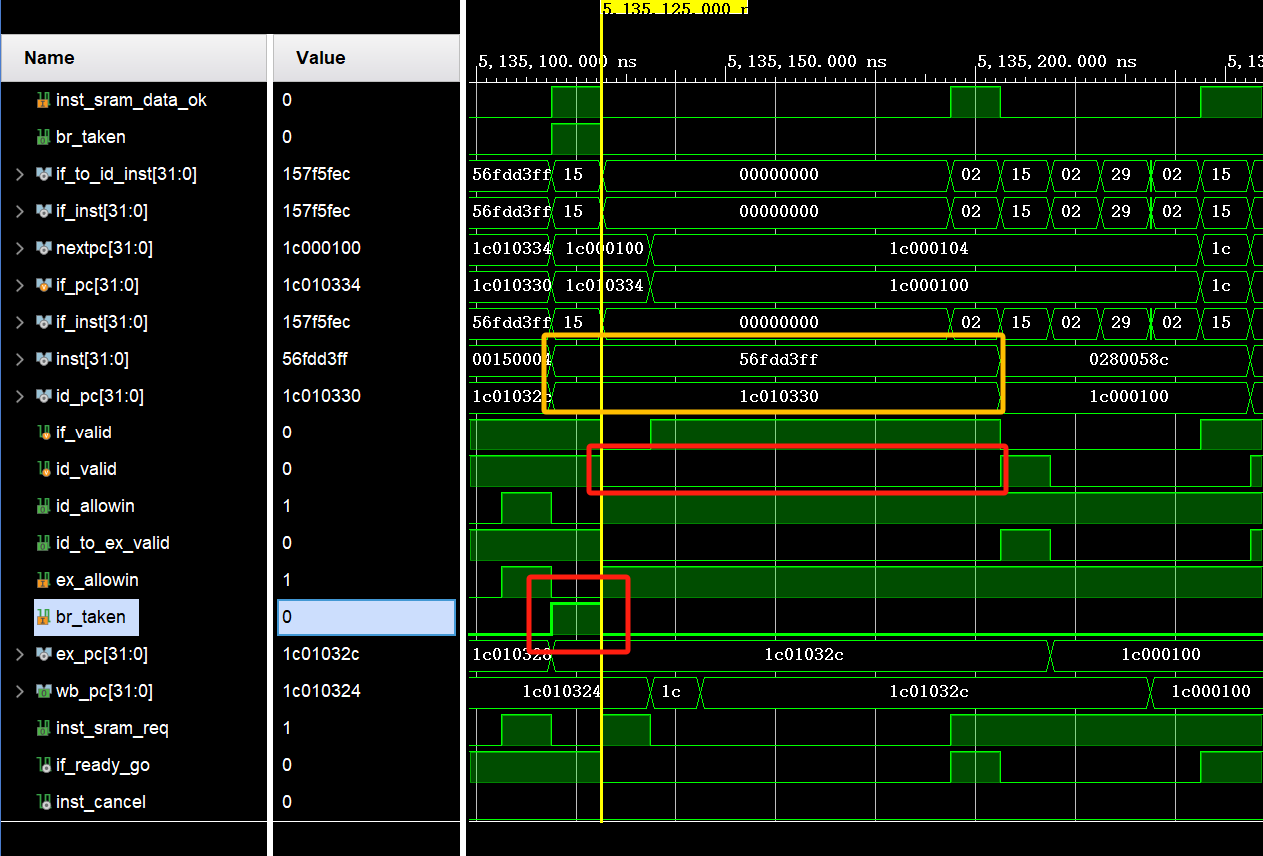


图40 错误6对应的波形图

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：

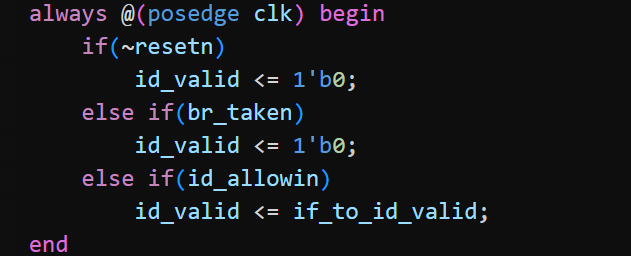


图41 错误6中id\_valid更新逻辑

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：

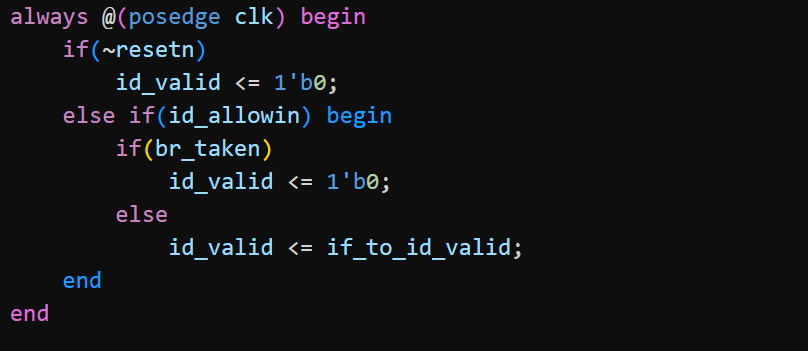


图42 错误6修改后代码

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. 课程总结与建议

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。

在设计时，我们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，这让我们能考虑更多好的解决方法，避免一些繁琐低效的设计。在代码编写环节，我们往往是两个人一起写或者是一个人写完，由另一个人检查，这个过程既让我们理解了代码，也在仿真前减少了许多简单的bug，提高了debug的效率。而在debug方面，由于我们都对代码有很清晰的理解，每个人都能进行debug，所以当某位组员其它课程任务繁忙时，能够相互进行帮助。我们觉得这些过程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与和谐，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（身边不少人考试周因为这些破防的）。所以对于分组机制，可能需要老师多一点建议，让组员之间能有更多的交流，也可能需要多一点约束，让每个人工作量尽可能相同（当然对于那些能单刷的大佬，可能也觉得无所谓吧）。

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心想事成，事事如意！