**Lab 9报告**

学号：2020K8009929034、2020K8009929019、2020K8009929037

姓名：胡康、李子恒、吕星宇

箱子号：05

一、实验任务（10%）

Cache访存速度和容量都介于寄存器和内存之间，可以有效提升访存速度，缓解访存速度与运算速度之间形成的“剪刀差”。本次实验的主要任务就是设计一个Cache模块。

二、实验设计（40%）

（一）总体设计思路

1. **Cache的组织方式**：两路组相联。

2. **Cache的访问类型**：可以从顶层上划分为四个部分，分别是Look Up, Hit Write, Replace, Refill。Look Up阶段，要根据Tag, Index和Offset，从Cache中查找数据，判断是否命中。对于命中的读操作，直接返回从Cache中读出的数据；对于命中的写操作，进入Hit Write阶段，将数据写入Cache并设置D位。对于未命中的读操作和写操作，进入Replace和Refill阶段，将数据从内存中取出来，替换掉一个Cache行，然后根据读操作还是写操作来设置D位和data信息。

3. **Cache表的组织结构**：每个Cache行含有Tag域, V位, D位, 以及Data域，因为Cache行占16字节，即4个字，所以Data域中保存了四个地址上的信息，可以进一步拆分为4个子表：Bank0, Bank1, Bank2和Bank3。Tag域和V位表现相同，合并为一个{Tag, V}表。D位单独组织成一个表。

3. **Cache的数据通路**：分为五个核心部分，分别是Request Buffer, Tag Compare, Data Select, Miss Buffer和Write Buffer。这一部分会在后面详细展开。

4. **Cache的控制逻辑**：为Cache设计了两个状态机。所有读写操作都要经过Look Up阶段。对于读命中，它会直接将数据返回；对于读不命中和写不命中，它们会复用相同的数据通路来进行Cache行的替换；对于写命中，它需要另有通路来专门向Cache行中写入数据。因此为主要的通路设计了主状态机来进行控制，有五个状态：IDLE, LOOKUP, MISS, REPLACE, REFILL，而对写命中设计了Write Buffer状态机来进行控制，有两个状态：IDLE和WRITE。

（二）重要模块1设计：Cache数据通路

1、工作原理

通过状态机控制，完成Cache的读写通路，实现Cache的基本功能。

2、功能描述

下面结合代码介绍一下Cache数据通路上的五个核心部分，Request Buffer, Tag Compare, Data Select, Miss Buffer和Write Buffer。

（1）Request Buffer：在IDLE阶段，如果发现了Cache请求，主状态机会进入LOOKUP阶段。这个时候要向Request Buffer中填入请求信息，包括op, index, tag, offset等，同时向Cache RAM发起读访问。代码如下图所示：

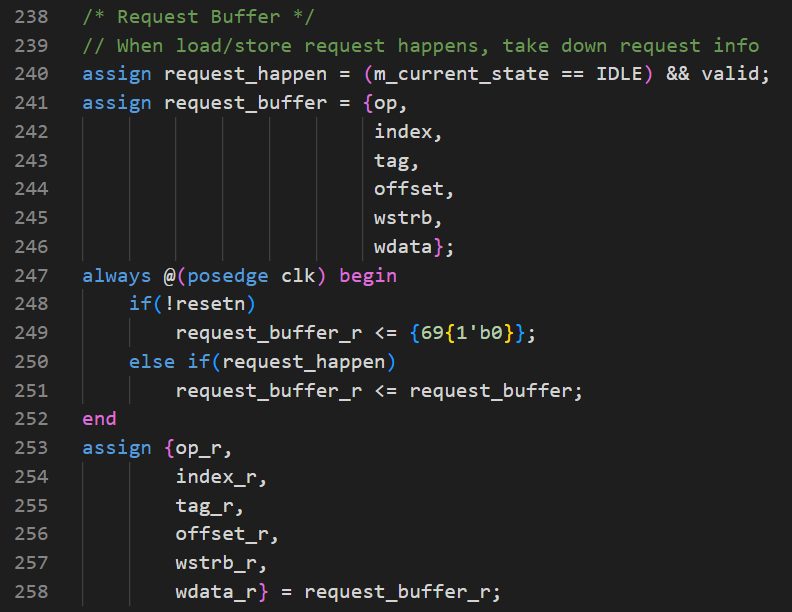


图1：Request Buffer设计代码

（2）Tag Compare：用于比较tag，判断是否命中。命中的条件为：请求的tag和读出的两路Cache行中的某一行相等，且该行V位为1，即该行有效。代码如下图所示：

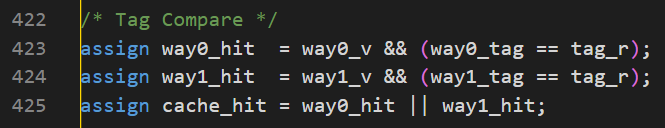


图2：Tag Compare设计代码

（3）Data Select：选出前面读出的两路Cache行的Data信息，产生读或写操作的结果。如果是读命中，只需要把命中的Cache行中，offset所指示的那个字选择出来。如果是读写不命中，需要把LFSR所指示的那一路Cache，也就是要被替换的Cache中的全部数据读出来。这里写命中没有考虑，因为它有专门的数据通路，由Write Buffer状态机进行管理，用Write Buffer存储内容。代码如下图所示：

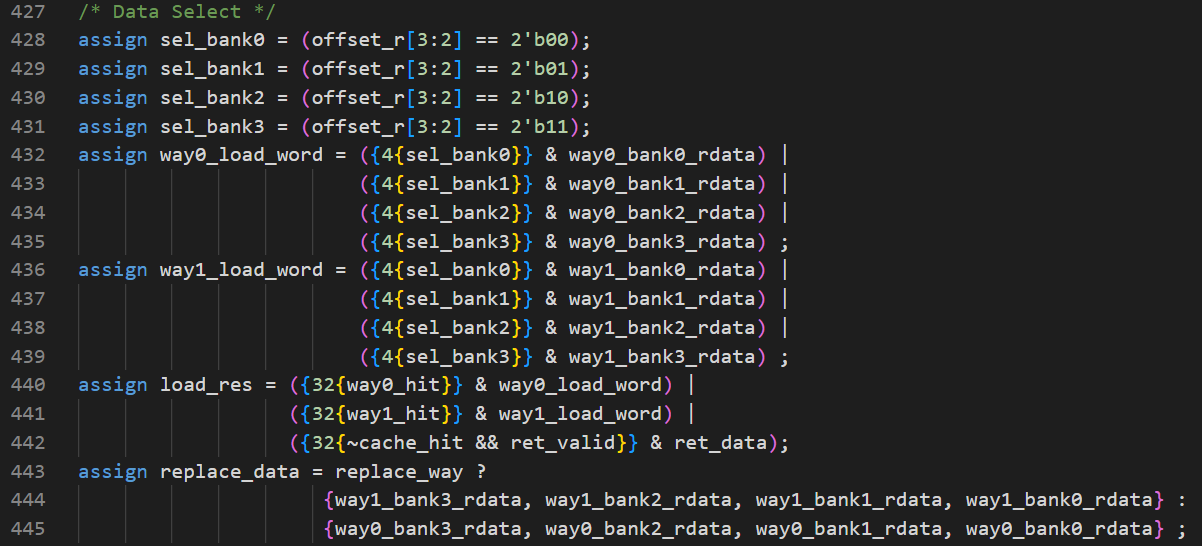


图3：Data Select设计代码

（4）Miss Buffer：读写不命中时要替换，Miss Buffer记录要替换的路的信息， 包括要替换哪一路（replace\_way），从AXI总线上返回的数据个数（ret\_data\_num），以及要替换的数据（miss\_bank0/1/2/3）。其中，replace\_way直接由LFSR生成，ret\_data\_num由AXI上返回的有效数据来计数，miss\_bank中存储某个bank要写回Cache的数据，它的来源有两个，一是来自于AXI返回的数据，也就是从内存中读出来的字；另一个是要向Cache中写的数据，这是不命中的store操作要写的数据。代码如下图所示（图中只展示了miss\_bank0，其余三个miss\_bank代码类似）：

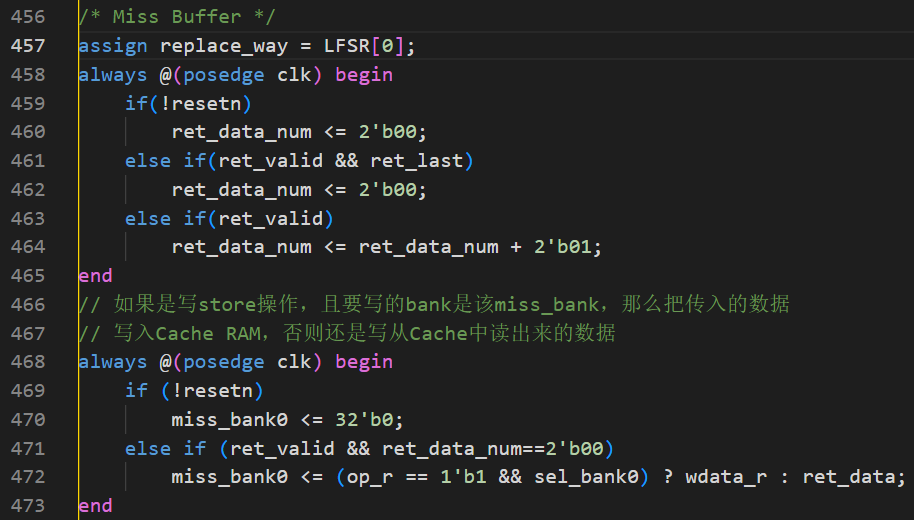


图4：Miss Buffer设计代码

（5）Write Buffer：这个是为写命中Hit Write专门准备的，前面已提到过多次。当Look Up阶段发现写命中的时候，会把store操作要写入的Cache块的相关信息先寄存起来，Write Buffer状态机转为WRITE状态，再写回。代码如下图所示：

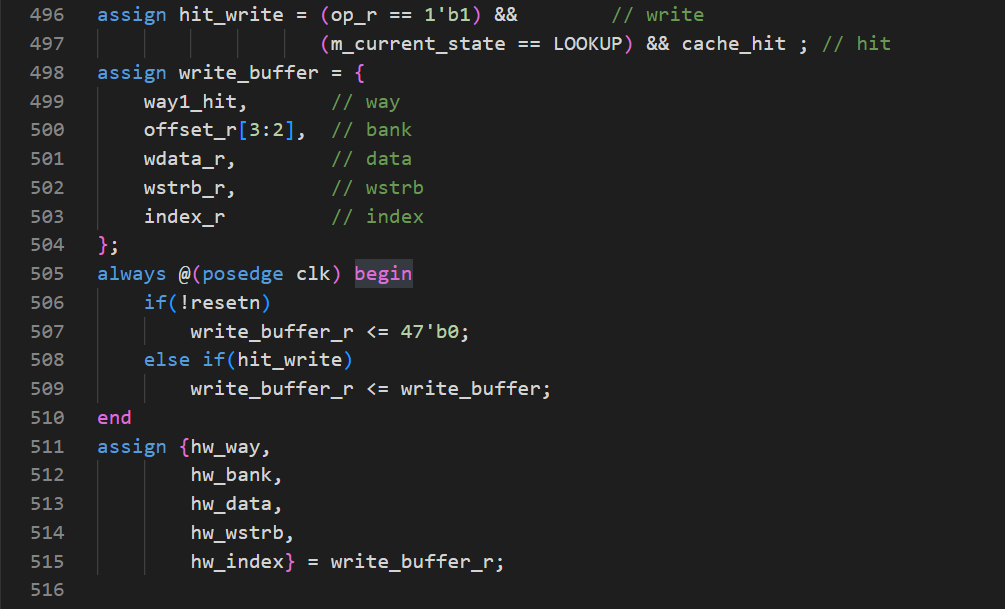


图5：Write Buffer设计代码

（三）重要模块2设计：Cache表读写信号生成

1、工作原理

Cache表分为七部分，分别是{Tag, V}表，D表和四个bank表。又因为是两路组相联结构，因此有两个{Tag, V}表，八个bank表和两个D表，其中{Tag, V}表使用21×256的同步RAM，bank表使用32×256的同步RAM，D表使用两个寄存器堆。这些RAM和寄存器堆需要有专门的读写信号来修改Cache的内容，从而实现Cache的具体行为。

2、功能描述

（1）D表：写命中时， Cache中写入了未被保存到内存的新值，需要将D位置为1。写不命中时，从内存取出来的值在被放入Cache之前，要用待写入的字换掉原有的那个字，因此也需要将D位置为1。读不命中时，从内存取出来的值直接被放入Cache，并且回传到CPU中，没有置新值，因此D位置0。综上，实现D位读写的代码逻辑如下（仅展示零路Cache的D表行为，一路同理）：

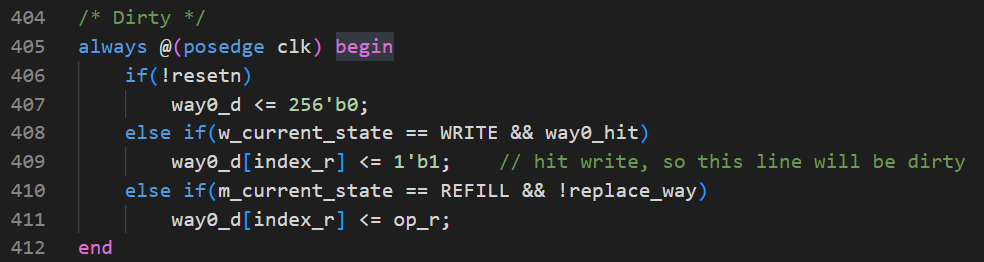


图6：D表读写逻辑代码

（2）{Tag, V}表：重填时要更新tag表，并将V置为1，地址要用寄存的index\_r。非重填时，读数的地址从index中直接取出即可，因为它和填入Request Buffer是在同一拍内进行的。代码如下图所示：

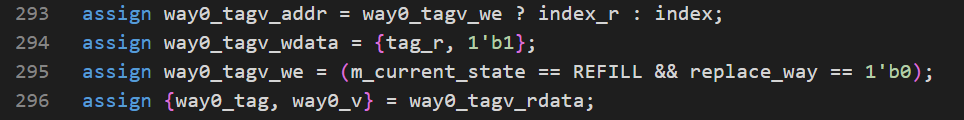


图7：{Tag, V}表读写逻辑代码

（3）bank表：对于we写使能信号，它需要在写命中和重填时拉高。写命中时，用wstrb控制其使能位，重填时，需要全部赋值成1。对于 wdata写入数据，它需要在写命中时赋成CPU上传来的wdata，而重填时需要赋成重填数据，它是CPU上传来的字覆盖AXI总线上传来的Data信息后生成的新信息。这里由于Data被拆分成了四个子表，所以直接用多路选择器和offset位判断各子表的赋值来源是CPU还是AXI送来的数据即可。对于地址addr信号，同样是来自index信号，根据是否命中判断地址来源即可。最终各信号的赋值方法如下图所示：

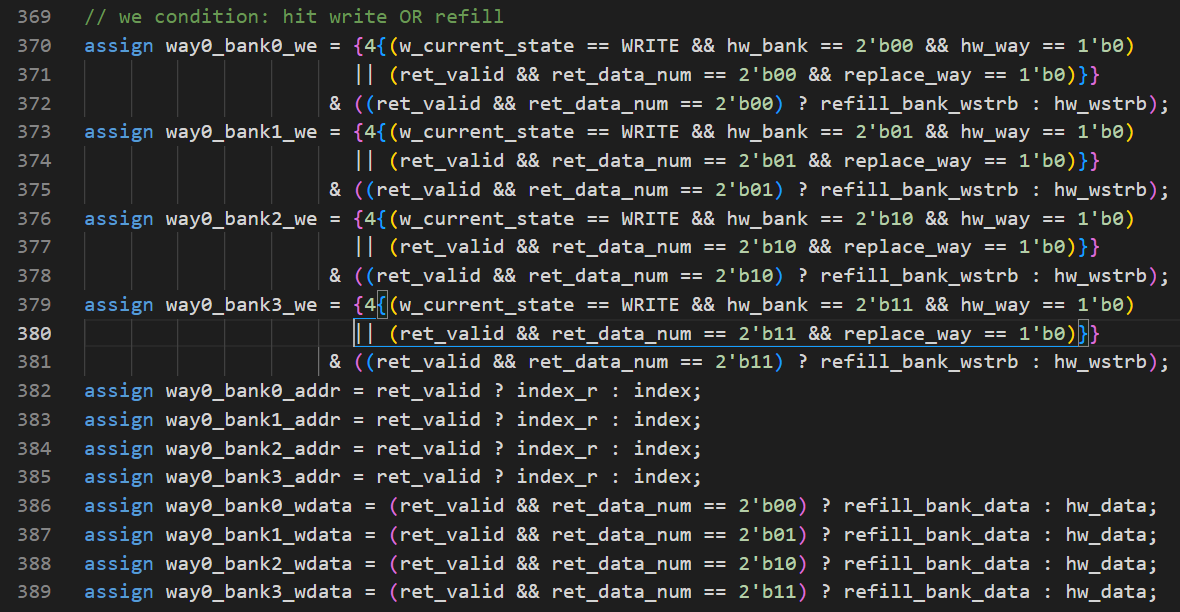


图8：4个bank表读写逻辑代码

三、实验过程（50%）

（一）实验流水账

12月12日 20：00-23：00 阅读讲义，理清Cache结构、读写操作过程、数据通路和控制逻辑，标记一些值得注意的细节。

12月14日 14：00-18：00，20：00-22：00 完成初步代码设计。

12月15日 13：30-15：00 仿真与debug。

12月16日 8：00-12：30 撰写报告，完成上板验证。

（二）错误记录

1、错误1：信号值为“Z”

（1）错误现象

如图所示，ret\_last信号的值恒为Z：

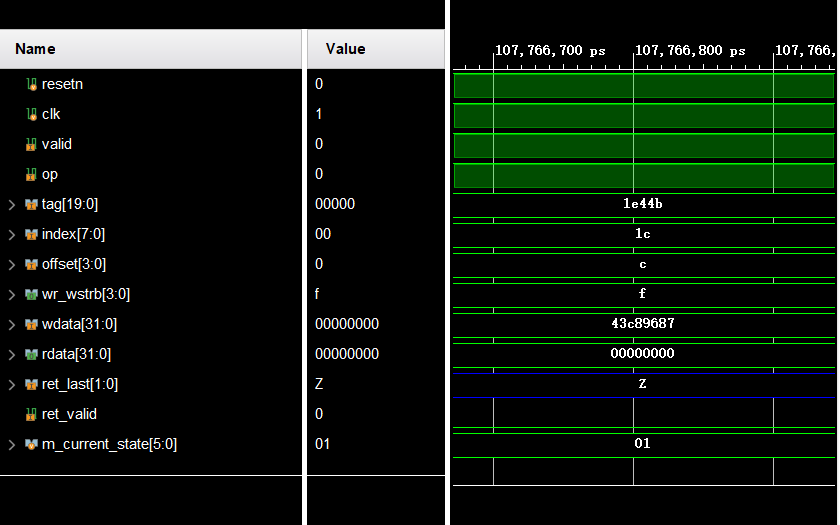


图9：ret\_last信号值为“Z”错误波形图

（2）分析定位过程

展开这个信号，发现它的低位是有效值：

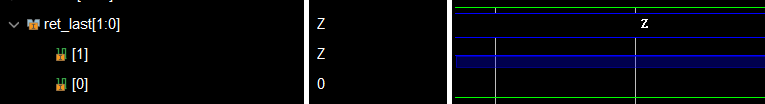


图10：ret\_last各比特位信号值展开图

想到，可能是因为信号位的赋值不对，或者位宽搞错了。于是我找到了cache\_top.v文件，看看顶层是怎么定义的，发现顶层文件中这个信号只有一位：

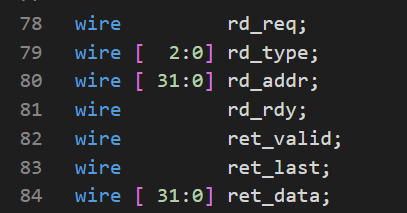


图11：ret\_last在cache\_top.v顶层文件中位宽定义为1位

果然是位宽搞错了，这里是讲义的位宽给错了，我直接照搬了讲义。改成1位之后就没有这个问题了。

（3）错误原因

信号位宽定义错误。

（4）修正效果

修改位宽为1位，很快便不再有Z值了。

（5）归纳总结

不能完全照搬讲义。当然这有时候也难以避免，最好的办法就是，不能假定讲义一定是对的，出错误的时候也要确认一下讲义是不是一定没出现手误搞错的地方。

2、错误2：仿真通过，上板不过。

（1）错误现象

仿真虽然通过了，但是上板时数码管一直显示为00，不递增到FF。

（2）分析定位过程

刚开始的两天毫无头绪，后来实在没办法了，找到了一位同学，想比照一下他的代码设计，看看我的问题可能出在哪里。对照他的设计，修改了几处可能的“错误”，但是依然是仿真通过，上板不过的问题。

后来去问另一位同学，他说问题可能出在某些信号的值为X或者Z上，这样的信号在仿真阶段是X或Z，但是在上板的时候一定是一个确定的信号，0或者1，这就有可能导致条件判断上仿真和上板结果不一样。但是我把出现信号为X的全部修正掉之后，也没有解决这个问题。

最后，我去和一个组员一起debug，这时他发现我和我一开始比照的另一个同学的代码，有些端口的输入输出方向不一样。一看讲义，真的是我搞错了。我把写错的input和output方向修改过来，上板一验证，果然通过了。

（3）错误原因

部分端口信号输入输出方向定义错误。

（4）修正效果

把写错的input和output方向修改过来，仿真和上板都能通过。

（5）归纳总结

一定要细心，不能犯这种低级错误。可以写完代码之后对照讲义看一下，看看自己是否把讲义中提到的核心设计理念和设计要求都给实现了。