Cache设计

# Cache应该有怎样的接口？



从宏观上来看，cache和处理器以及RAM之间有上面这些需要交流的内容。我们需要为这些“交流”设计信号。

比如，cache为了给CPU提供读取指令的接口，应该有这样一些信号：

|  |  |
| --- | --- |
| 信号 | 方向 |
| instruction read request | CPU -> cache |
| instruction address | CPU -> cache |
| instruction | cache -> CPU  返回指令的内容 |
| instruction read response | cache -> CPU  告诉CPU，指令准备好了 |

类似的，为支持CPU与cache、cache与RAM的其它“交流”，你也应该设计这样的信号。

# Cache内部应该是什么样？

为了决定cache的宏观结构，我们需要思考以下2问题：

* I$与D$分离还是合并？

我们上面描述CPU与cache的交流时，是将来自CPU的指令与数据的请求分开描述的。在实现cache的时候，我们可以考虑将I$与D$分离，也可以合并。现在的体系结构中，一般采取分离的cache。分离有以下好处：

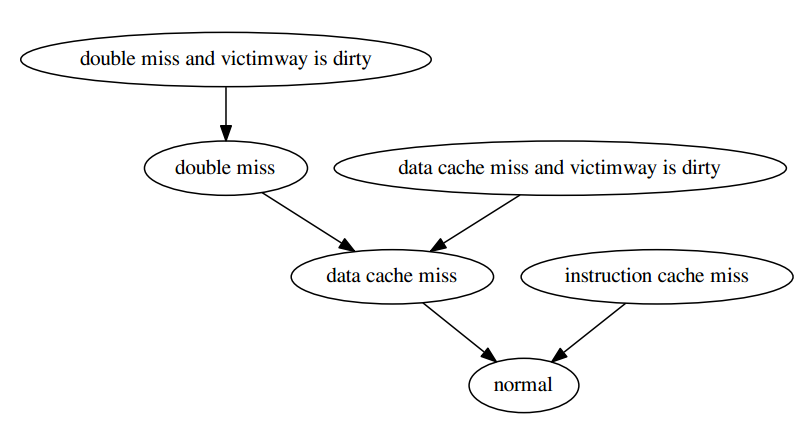
* 减少读口的数量：假设I$需要2个读口，D$需要2个读口，如果合并的话，整个cache需要4个读口；在体系结构课程中会学习到：4KB 4读口的SRAM的latency比2KB 2读口的SRAM大。
* 避免data cache miss影响取值：某些程序的data cache miss rate很高，如果合并的话，会让instruction cache miss rate也上升，进一步降低性能。

当然，我们自己实现的时候，是可以自由选择的，合并的难度更低，而且不会有一致性问题。我自己实现的是分离的。

* 采用write through还是write back？

一般来讲，write back性能更高，占用总线的时间更少，功耗也更低；而write through实现更简单，而且配合vivado提供的DDR2 的mig，性能也不差，因此二者都可以采用。我自己实现的是write back。

以上2个问题决定了你的cache的状态机，下面是采用I$与D$分离+写回机制的cache在cache miss处理时的状态转移的示意图（先处理D$还是I$可以自行调整）：



# 状态转移的条件

由上图引出的问题是，cache的状态应该如何转移。

1. **tag compare**: 当正常的处于空闲状态的cache接收到来自CPU的读写请求时，需要将请求的地址拆分为3段：| tag | index | offset |，cache根据index找到该请求需要访问的set，将该set中现有的line的tag与请求的tag字段一一比较，决定是否命中。
2. **state decision:** 如果I$和D$都没有出现miss，那么cache可以直接取出请求的内容，并响应CPU；如果I$与D$发生了miss，那么就需要根据miss的类型将cache的状态转移到上图中除normal外的5种状态中的一种。
3. **abnormal state handling:** 这里以D$ miss + dirty为例：
4. 因为要被替换的line为脏，必须先写回脏块。需要向DDR控制模块发送一个写入请求，并提供该脏块的内容，当DDR模块接受该请求后转移到下一状态；
5. 此时的状态从D$ miss + dirty转移到了D$ miss。需要向DDR控制模块发送一个载入请求，当DDR控制模块相应该请求时，从DDR控制模块的buffer中读入要载入的行的内容，并写入到data cache对应的victim line中。
6. 回到正常状态

分析了如何进行状态转移之后，我们可以将cache分割为这几部分：

* 控制逻辑
* D$
* I$

在**tag compare**阶段，cache向I$和D$送入地址，I$和D$则返回命中和脏位等信息；控制逻辑需要根据这些信息决定下一状态。为此I$与D$需要向顶层cache提供这些接口：

|  |  |
| --- | --- |
| enable | cache -> I$ & D$  因为仅当顶层cache收到来自CPU的请求时才需要让I$和D$比较tag，其它时候可以不进行比较 |
| tag | index | offset | cache -> I$ & D$  I$和D$需要index来定位set，需要tag来确定是否命中，以及是否为脏；需要offset来确定返回的word是一个块中的哪一个word |
| hit | I$ & D$ -> cache |
| dirty | D$ -> cache |

在**state decision**阶段，我们需要在顶层cache中设计逻辑，根据上述信息确定下一周期cache的状态。在实际设计中，如果cache hit，那么你既可以选择在当周期完成读写，也可以选择在下一周期读写。二者的区别是，如果要在当周期完成读写，当周期的组合逻辑链可能过长，有可能成为系统的critical path，降低性能，且该方案功耗较高；如果在下一周期完成，则cache读写命中的延迟将会变为2个周期。在现在的实际系统中，L1$采用第一种方案，因为速度要求高，且L1$的体积小，延迟低；而在L2$和LLC中，则采取先比较，后读写的策略。

在**abnormal state handling**阶段，需要进行2种操作：载入和写回（write back）。

1. 载入时，cache需要向RAM提供地址，这个地址需要从I$或者D$中拼接得到。cache需要告诉RAM应该工作了，告诉RAM应该进行读取 工作。然后RAM准备好数据之后，RAM需要告诉cache数据准备好了，并且将数据送给cache。最后cache需要将载入的数据写入到I$或者D$中，需要注意的是这时候的写入和来自CPU的写入是不同的：不需要进行tag比较，直接写入到victim line。根据上述需求，我们需要在cache与RAM之间增加如下接口：

|  |  |
| --- | --- |
| request address | cache -> memory |
| enable | cache -> memory |
| loading block | memory -> cache  即需要载入的行?? |
| loading ready | memory -> cache  表示cache请求的行已经准备好 |

在I$与D$之间需要增加如下接口：

|  |  |
| --- | --- |
| loading block | cache -> I$ & D$ |
| write enable | cache -> I$ & D$  用于写入载入的行 |
| if compare | cache -> I$ & D$  I$和D$对载入时的写操作和来自CPU的写操作应该做出不同的反应，而此信号就是用来告诉I$和D$他们是否应该进行tag 比较的。 |

1. 请自行分析D$的脏块写回需要注意哪些问题，并为之增加接口。

# I$与D$内部的设计

这部分内容实际上是如何实现一个组相联的cache的实体，而此内容在UWsic的cs552中有现成的讲义，所以请参考次链接：

<http://pages.cs.wisc.edu/~david/courses/cs552/S12/includes/cache-mod.html>

该讲义为学生提供了设计一个直接映射的cache的基本示例，设计风格值得参考。该实验的目的是让学生实现一个组相联的cache，你可以考虑直接使用网站上提供的直接映射的代码拼接成组相联的cache，也可以考虑自己从头实现。

请注意，其中dump cache内容的代码是不可综合的，我们可以不使用。

# 实验建议

1. 我们的讲义是自顶向下的，而你实现的时候应该自底向上地实现。首先你需要实现UWisc的cs552中的cache的功能，然后再将实现的组相联的cache作为I$和D$拼接成为最终的顶层cache。
2. 关于测试，建议使用一个Block RAM作为cache的后端，并且自己将Block RAM封装，以模拟DDR的burst read功能。以后我们的处理器使用的是一块DDR2芯片，是具有burst read 功能的。Traffic generator将会由助教提供。