ACE (AXI Coherency Extensions)

目录

[ACE (AXI Coherency Extensions) 1](#_Toc88125562)

[1. Memory Model 4](#_Toc88125563)

[1.1. Memory consistency 4](#_Toc88125564)

[1.2. Memory attributes 4](#_Toc88125565)

[1.3. Memory types 6](#_Toc88125566)

[1.3.1. Normal memory 6](#_Toc88125567)

[1.3.2. Device memory 6](#_Toc88125568)

[2. Coherence Protocol 8](#_Toc88125569)

[2.1. MOESI Protocol 8](#_Toc88125570)

[2.2. ACE Cache State Model 9](#_Toc88125571)

[3. Transaction Overview 10](#_Toc88125572)

[3.1. Non-snooping transactions 10](#_Toc88125573)

[3.2. Coherent transactions 10](#_Toc88125574)

[3.2.1. Performing load operations from Shareable locations 10](#_Toc88125575)

[3.2.2. Performing store operations to Shareable locations 10](#_Toc88125576)

[3.2.3. Accessing Shareable locations when no cached copy is required 11](#_Toc88125577)

[3.3. Memory update transactions 11](#_Toc88125578)

[3.4. Cache maintenance transactions 11](#_Toc88125579)

[3.5. Snoop transactions 12](#_Toc88125580)

[3.6. Barrier transactions 12](#_Toc88125581)

[3.7. Distributed virtual memory transactions 13](#_Toc88125582)

[4. Signal Descriptions 15](#_Toc88125583)

[4.1. Changes to existing AXI channels 15](#_Toc88125584)

[4.1.1. Read address channel (AR) signals 15](#_Toc88125585)

[4.1.2. Write address channel (AW) signals 16](#_Toc88125586)

[4.1.3. Read data channel (R) signals 17](#_Toc88125587)

[4.2. Additional channels defined by ACE 18](#_Toc88125588)

[4.2.1. Snoop address channel (AC) signals 19](#_Toc88125589)

[4.2.2. Snoop response channel (CR) signals 21](#_Toc88125590)

[4.2.3. Snoop data channel (CD) signals 23](#_Toc88125591)

[4.3. Additional response signals and signaling requirements defined by ACE 24](#_Toc88125592)

[4.3.1. Read acknowledge signal 24](#_Toc88125593)

[4.3.2. Write acknowledge signal 24](#_Toc88125594)

[4.3.3. Reset requirements 24](#_Toc88125595)

[5. Transaction processing 25](#_Toc88125596)

[5.1. Snoop channel dependencies 25](#_Toc88125597)

[5.2. Channel usage example 26](#_Toc88125598)

[5.2.1. Performing load operations from Shareable locations 26](#_Toc88125599)

[5.2.2. Performing store operations to Shareable locations 26](#_Toc88125600)

[5.2.3. Write transaction 28](#_Toc88125601)

[5.3. State change 29](#_Toc88125602)

[5.3.1. ReadShared 29](#_Toc88125603)

[5.3.2. ReadUnique 29](#_Toc88125604)

[5.3.3. CleanUnique 30](#_Toc88125605)

[5.3.4. MakeUnique 30](#_Toc88125606)

[5.3.5. WriteBack 30](#_Toc88125607)

[5.3.6. WriteClean 30](#_Toc88125608)

[5.4. Handling overlapping write transactions 31](#_Toc88125609)

[5.4.1. Overlapping ReadUnique 31](#_Toc88125610)

[5.4.2. Overlapping MakeUnique 31](#_Toc88125611)

[5.4.3. Overlapping CleanUnique 32](#_Toc88125612)

[5.5. Memory update in progress 33](#_Toc88125613)

[6. Interconnect Requirements 34](#_Toc88125618)

[6.1. About the interconnect requirements 34](#_Toc88125619)

[6.2. Sequencing transactions 35](#_Toc88125620)

[6.3. Interactions with main memory 37](#_Toc88125621)

[6.3.1. Interconnect read from main memory or peripheral device 37](#_Toc88125622)

[6.3.2. Main memory update that is generated by the interconnect 38](#_Toc88125623)

[6.4. Other requirements 39](#_Toc88125624)

[6.4.1. Non-blocking requirements 39](#_Toc88125625)

[6.4.2. Permitted transaction modifications 39](#_Toc88125626)

[7. ACE-Lite 40](#_Toc88125627)

[8. Cache Coherent Interconnect 41](#_Toc88125628)

[8.1. CCI-400 41](#_Toc88125629)

[8.2. Snoop Filter 42](#_Toc88125630)

[8.3. Snoop Control Unit 44](#_Toc88125631)

[9. Comparison of ACE and TileLink 45](#_Toc88125632)

# Memory Model

## Memory consistency

ARMv8-A采用的是Weakly Ordered Consistency。也就是说内存真实的访问顺序和程序的load/store操作顺序并不完全一致。WO（Weakly Ordered）比SC（Sequential Consistency）松弛得多，编译器可以自由地改变load和store指令的顺序，只要它们不跨越同步边界。在执行过程中，只要同步访问之间的次序得到保证和，load和store指令的执行就可以改变次序或相互叠加，无需原子性地执行这些指令。WO比SC能提供更好的性能，但是其代价是必须向硬件适当地标识同步访问。

## Memory attributes

* cacheability：对于cacheable的区域，允许放入cache，non-cacheable则不允许。
* shareability：对于Shareable的区域，代表此区域可以被多个agent共享，也就是说需要有硬件保证访问这块区域的agent之间保持一致性。

**Shareability domain**是一组Manager (an agent that initiates transactions)，它使Manager在发起coherency or barrier transactions时能够确定需要包含哪些其它的Manager。

对于coherent transactions，Manager使用shareability domain来确定哪些其它的Manager在它们的local cache中可能拥有这个地址空间的备份。Interconnect使用这个信息来确定必须监听哪些Manager以完成transaction。

ACE协议定义了以下几个等级的shareability domain：

* **Non-Shareable**：The domain contains a single Manager component.
* **Inner Shareable**：The domain can include additional Manager components.
* **Outer Shareable**：The domain contains at least all Manager components in the Inner domain, but can include additional Manager components.
* **System**：The domain includes all Manager components in the system.

ARM引入了一个概念，共享域（shareability domains.），包含以下几种：

* Inner shareable，意味着它适用于整个内部可共享域。这表示该domain内的处理器之间可以相互share数据。一个系统可以有多个inner shareable domains，并且当某个操作影响到其中一个inner shareable domain时，它并不会影响到其它的inner shareable domain。
* Outer shareable，意味着它适用于内部可共享和外部可共享域。一个outer shareable domain可以由一个或多个inner shareable domain组成，并且当一个操作影响到outer shareable domain时，也会影响到其下所有的inner shareable domain。
* Non-shareable，表示相关区域只能给指定的处理器访问。
* Full System，包含全部处理器。

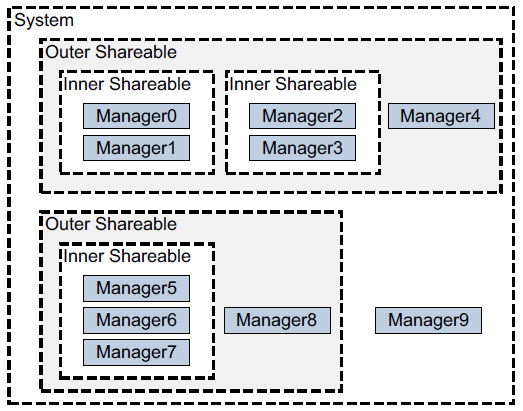


图1 Shareability domains

## Memory types

### Normal memory

这种类型通常用于大容量的内存操作，包括读/写（RAM）和只读操作（ROM）。Normal memory可以通过weakly ordered来提高性能，并且编译器也能做更深层次的优化工作。为了提高性能，应用程序和数据总是被标记为Normal类型。但如果需要严格的memory访问顺序，需要用barrier操作来保证。

Normal memory具有以下属性：

* 允许执行推测性读取（speculative read）。
* 对相同位置的两次write，若其中至少有一次为Normal memory类型，可以在它们到达endpoint之前被merge，除非两次write之间存在排序关系。
* 支持非对齐访问。

### Device memory

Device访问，即AxCACHE[1] = 0时，设备事务必须只使用域级别System。

ARM架构禁止对Device memory进行任何的推测性读取（speculative read）。Device memory适用于读取敏感（read-sensitive）的位置，通常用于外设区域。

ARMv8中Device memory的属性有：

* **Gathering (G)**：如果具有Gathering属性，那么：
  + 同一类型对同一位置的多个访问（读或写），可以merge成一个transaction。
  + 同一类型对不同位置的多个访问（读或写），可以merge成interconnect上的一个transaction。
* **Reordering (R)**：如果具有Reordering属性，那么对该区域的访问可以重新排序，排序规则与访问Normal Non-Cacheable memory一致。
* **Early Write Acknowledgement (E)**：如果具有E属性，对于写操作可以提前回复ack，而不必等到真正完成写入操作。而对于nE属性，则必须等待endpoint返回的write acknowledgement。

Device memory分为以下4种：

* **Device-nGnRnE (strongest)**：Device non-Gathering, non-Reordering, No Early Write Acknowledgement。
* **Device-nGnRE**：Device non-Gathering, non-Reordering, Early Write Acknowledgement。
* **Device-nGRE**：Device non-Gathering, Reordering, Early Write Acknowledgement。
* **Device-GRE （weakest）**：Device Gathering, Reordering, Early Write Acknowledgement。此时行为已经接近Normal memory，除了不能执行speculative的访问。

**Note**：对于G属性，可以在G和nG中选择，R，E同理。软件通常选择weakest type。

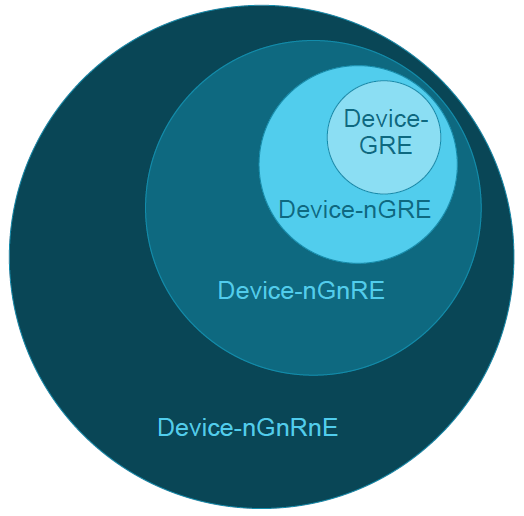


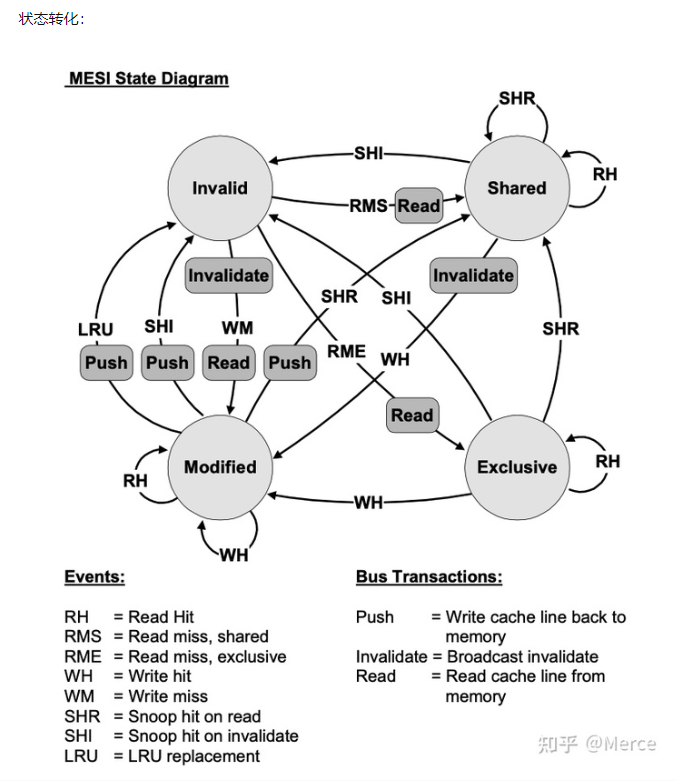
图2 Device memory

# Coherence Protocol

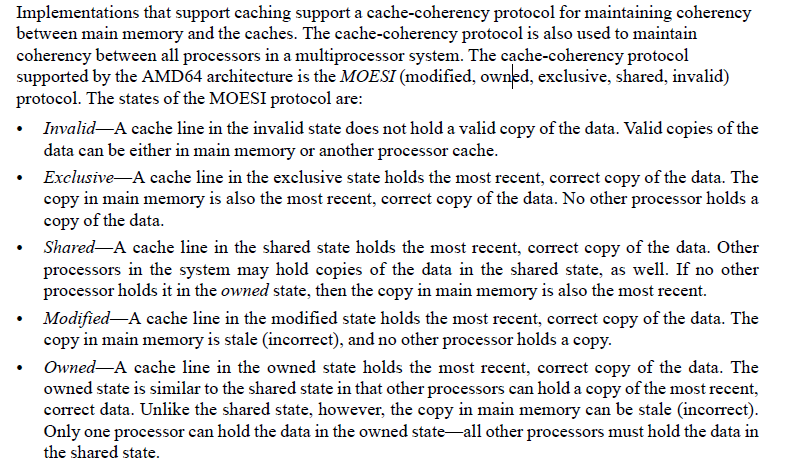
https://zhuanlan.zhihu.com/p/468636398

<https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESIHelp.htm> 动画模拟





MOESI 是cache的一中协议机制，用于确保不同cache 缓存的相同位置的数据的一致性。



Invali: 当前cache line的数据无效；  
exclusive: cache line 数据最新有效，且与memory数据一致，其他cache这种没有相同数据；  
share: 当前cacheline 的数据在其他cache中也存在，分两种情况（1）有且只有一个cache中数据为owned，其他cache也有copy的话， 状态也为shared，memory数据不是最新的；（2）memory数据最新，其他cache的copy都是shared；  
modify: memory 数据不是最新，有且只有当前cache有最新数据；其余cache可能有旧数据（invalid）  
owned：memory 数据不是最新，只有当前cache的数据是最新并且为owned状态，其他cache有一个当前的copy，并且是shared状态。

**MOESI协议引入了一个O(Owned)状态，并在MESI协议的基础上，进行了重新定义了S状态，而E、M和I状态和MESI协议的对应状态相同。同时操作相对于MESI多了几种(MESI操作有4种，本地读写、远程读写)，MOESI操作有8种(数据拷贝读写、本地副本读写、本地副本读写无数据、从远程CPU获取副本读写)。**

**O位。O位为1表示在当前Cache 行中包含的数据是当前处理器系统最新的数据拷贝，而且在其他CPU中一定具有该Cache行的副本，其他CPU的Cache行状态为S。如果主存储器的数据在多个CPU的Cache中都具有副本时，有且仅有一个CPU的Cache行状态为O，其他CPU的Cache行状态只能为S。与MESI协议中的S状态不同，状态为O的Cache行中的数据与存储器中的数据并不一致。**

**S位。在MOESI协议中，S状态的定义发生了细微的变化。当一个Cache行状态为S时，其包含的数据并不一定与存储器一致。如果在其他CPU的Cache中不存在状态为O的副本时，该Cache行中的数据与存储器一致；如果在其他CPU的Cache中存在状态为O的副本时，Cache行中的数据与存储器不一致。**



## MOESI Protocol

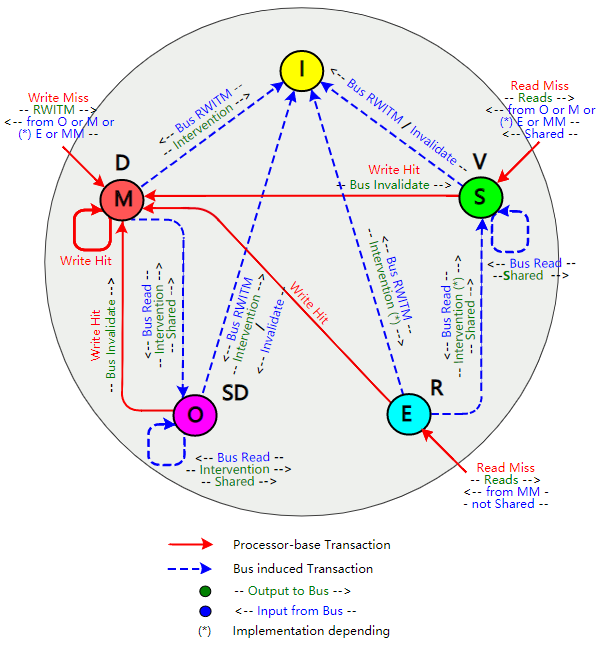


图3 MOESI Protocol State Transaction Diagram

* 当cache line处于M (Modified) 状态时，若接收到bus read，直接将dirty cache line传递给interconnect，并跳转至Owned状态。
* 当cache line处于O (Owned) 状态时，若接收到bus read，同样将dirty cache line传递给interconnect，状态不变。若自己要写，跳转至Modified状态，并invalidate其它cache。若接收到invalidate请求，则跳转至Invalid状态。

当*CPU*对一段存储器进行写操作时，如果这些数据在本地*Cache*中命中时，其状态可能为*E*、*S*、*M*或者*O*。

* 状态为*E*或者*M*时，数据将直接写入到*Cache*中，并将状态改为*M*。
* 状态为*S*时，数据将直接写入到*Cache*中，并将状态改为*M*，同时其他*CPU*保存该数据副本的*Cache*行状态将从*S*或者*O*迁移到*I(Probe Write Hit)*。
* 状态为*O*时，数据将直接写入到*Cache*中，并将状态改为*M*，同时其他*CPU*保存该数据副本的*Cache*行状态将从*S*迁移到*I(Probe Write Hit)*。

当*CPU A*对一段存储器进行写操作时，如果这些数据没有在本地*Cache*中命中时，而在其他*CPU*，如*CPU B*的*Cache*中命中时，其状态可能为*E*、*S*、*M*或者*O*。其中*CPU A*使用*CPU B*在同一个*Cache*共享域中。

* *Cache*行状态为*E*时，*CPU B*将该*Cache*行状态改为*I*；而*CPU A*将从本地申请一新的个*Cache*行，将数据写入，并该*Cache*行状态更新为*M*。
* *Cache*行状态为*S*时，*CPU B*将该*Cache*行状态改为*I*，而且具有同样副本的其他*CPU*的*Cache*行也需要将状态改为*I*；而*CPU A*将从本地申请一个*Cache*行，将数据写入，并该*Cache*行状态更新为*M*。
* *Cache*行状态为*M*时，*CPU B*将原*Cache*行中的数据回写到主存储器，并将该*Cache*行状态改为*I*；而*CPU A*将从本地申请一个*Cache*行，将数据写入，并该*Cache*行状态更新为*M*。
* *Cache*行状态为*O*时，*CPU B*将原*Cache*行中的数据回写到主存储器，并将该*Cache*行状态改为*I*，具有同样数据副本的其他*CPU*的*Cache*行也需要将状态从*S*更改为*I*；*CPU A*将从本地申请一个*Cache*行，将数据写入，并该*Cache*行状态更新为*M*。

*Cache*行状态可以从*M*迁移到*O*。例如当*CPU A*读取的数据从*CPU B*中命中时，如果在*CPU B*中*Cache*行的状态为*M*时，将迁移到*O*，同时*CPU B*将数据传送给*CPU A*新申请的*Cache*行中，而且*CPU A*的*Cache*行状态将被更改为*S*。

当*CPU*读取的数据在本地*Cache*中命中，而且*Cache*行状态为*O*时，数据将从本地*Cache*获得，并不会改变*Cache*行状态。如果*CPU A*读取的数据在其他*Cache*中命中，如在*CPU B*的*Cache*中命中而且其状态为*O*时，*CPU B*将该*Cache*行状态保持为*O*，同时*CPU B*将数据传送给*CPU A*新申请的*Cache*行中，而且*CPU A*的*Cache*行状态将被更改为*S*。

在某些应用场合，使用*MOESI*协议将极大提高*Cache*的利用率，因为该协议引入了*O*状态，从而在发送*Read Hit*的情况时，不必将状态为*M*的*Cache*回写到主存储器，而是直接从一个*CPU*的*Cache*将数据传递到另外一个*CPU*。目前*MOESI*协议在*AMD*和*RMI*公司的处理器中得到了广泛的应用。

## ACE Cache State Model

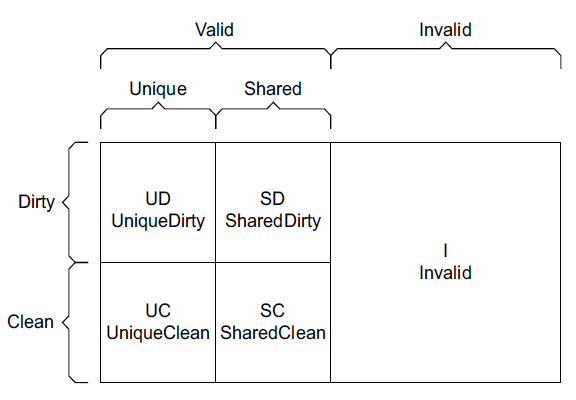
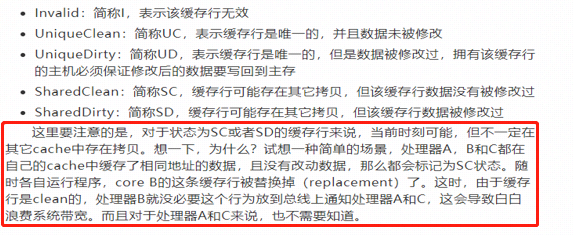


图4 ACE cache state model

* Invalid：对应MOESI的Invalid
* SharedClean：（**It is not known if the cache line is modified with respect to main memory, but this component is not responsible for updating main memory.**）

对应MOESI的Shared

* SharedDirty：对应MOESI的Owned
* UniqueClean：对应MOESI的Exclusive
* UniqueDirty：对应MOESI的Modified



# Transaction Overview

https://mp.weixin.qq.com/s/mAa9efm6trZhNOOerKygnA

## Non-snooping transactions

用于访问不在其他处理器单元cache中的内存位置，因此不需要snoop transactions。主要用于non-shareable和device memory。

* **ReadNoSnoop**
* **WriteNoSnoop**

## Coherent transactions

此类transaction通常用于访问shareable的地址空间，因此必须维护一致性。

### Read Clean （load operations）

* 该事务表示请求读取的处理器只能接受clean的缓存行，不能接受dirty的。该事务用于不能接受dirty缓存行，或者是写通（write through）缓存的处理器。

### Read NotSharedDirty（load operations）

* 该事务表示请求读的处理器可以接受除SharedDirty状态外的缓存行

### Read Shared（load operations）

* 该事务表示请求读的处理器可以接受任何状态的缓存行

如果接收到上述transaction的Cache包含相同的备份，协议推荐该Cache提供数据以完成snoop transaction。Interconnect需要传递数据给发起请求的处理器。若提供数据的Cache开始处于Unique状态，需要转移到Shared状态。

### 3.2.4 Read Unique（store operations）

该事务用于存储（store）部分缓存行，且此处理器没有缓存拷贝的情况下。ReadUnique事务获得数据拷贝，并确保没有其它拷贝在别的处理器。

### 3.2.5 Clean Unique（store operations）

* 该事务用于存储（store）部分缓存行，且此处理器拥有拷贝的情况下。CleanUnique清除其它的拷贝，如果其它拷备是dirty的，要保证dirty的缓存行写回到主存

### 3.2.6 Make Unique（store operations）

* 该transaction用于执行full cache line store的场景。此transaction会invalidate其它所有的备份，即使有dirty cache line也无需写回。

### 3.2.7 Read Once（no cached copy is required）

* 该transaction表示得到数据的snapshot（快照），但是不要求复制到cache中。如果提供数据的cache line为Unique状态，则无需跳转至Shared状态。

### 3.2.8 Write Unique（no cached copy is required）

该transaction可以在发起write transaction前清除该cache line的所有备份，确保Dirty cache line被写回memory。此transaction可以用于write partial/full cache line。

### 3.2.9 Write Line Unique（no cached copy is required）

与**WriteUnique**类似，但只能用于full cache line。

**Read Once Write Unique Write Line Unique**以上transaction主要用于访问Shareable memory location但是Manager并不保留cache line的备份（可能因为不想allocate cache line或者根本没有cache）

## Memory update transactions

此类事务更新内存，但是不需要监听其它处理器

### 3.3.1 WriteBack

将把dirty的缓存行写回到主存，并释放这条缓存行

### 3.3.2 WriteClean

把dirty的缓存行写回到主存，但是保持这条缓存行

### 3.3.3 WriteEvict

逐出一条clean的缓存行，可以用来把缓存行写到下层的cache中，比如L3。

WriteEvict并不需要更新主存

### 3.3.4 Evict

该用来指示一条缓存行的地址被逐出，可以用来构建snoop filter。该事务没有数据传输

## Cache maintenance transactions

处理器广播此类transaction来访问和维护其他处理器的cache。特别地，该类transaction能够使处理器查看load和store操作对无法以其他形式访问的system cache的影响，此操作叫做Software Cache Maintenance。Cache maintenance transactions也可以传播到下游cache，允许维护系统中所有的cache。

**Note**：在处理器发起此类transaction时也需对自己的local cache执行合适的操作。

### 3.4.1 clean Shared

一个处理器可以通过这个事务对其它处理器的缓存进行clean操作。有dirty缓存行的snooped cache接收到CleanShared，必须写回到主存，该snooped cache可以保存本地的副本。

### 3.4.2 Clean Invalid

一个处理器可以通过这个事务对其它处理器的缓存进行clean和invalidate操作。如果snooped cache拥有该条缓存行，并且是clean状态，那么要移除本地拷贝；如果是dirty状态，要提供该条缓存行以写回到主存，并移除本地拷贝。CleanInvalid事务用于确保主存已更新，且没有主存位置的缓存副本

### 3.4.3 Make Invalid

处理器可通过该transaction对系统中其他处理器的cache发起invalidate操作。Snooped cache在接收到此transaction之后，remove相应的cache line，即使有Dirty cache line，也无需写回main memory。

## Snoop transactions

使用snoop address，snoop response，snoop data channels的transactions属于snoop transactions，是coherent transactions和cache maintenance transactions的子集。

## Barrier transactions

Barrier transactions are **not** supported in ACE5 and ACE5-Lite variant interfaces.  
链接：<https://www.zhihu.com/question/20228202/answer/184765470>

来源：知乎

可以看看<A PRIMER ON MEMORY CONSISTENCY AND CACHE COHERENCE>这本书。

先来个某SPEC的定义下 memory barrier， “[memory barrier](https://www.zhihu.com/search?q=memory+barrier&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A184765470%7D)(DMB) is issued by a master to guarantee that if another master in the appropriate domain can observe any transaaction after the barrier it must be able to observe every transaction prior to the barrier.”

发起请求的master(可以理解一个CPU) 用DMB来保证另一个master(另一个CPU)在看到BARRIER后面的transaction前必须看到所有的BARRIER之前的transaction。

什么意思？举个简单的列子

MASTER 0做了以下步骤的指令：

1.read A

2.read B

3.write C

4.memory barrier

5.write D

如果另一个MASTER1“看到 "write D"这个操作了”，说明barrier之前的那些操作它一定都看到了。（你肯定觉得这简直不是人话，它到底想干嘛？其实这里有个snoop的概念，所谓MASTER1看到，其实是write D这个命令经由interconnect所发起的SNOOP命令到了MASTER1，通过barrier，我一定可以保证master1先看到前三条指令，再看到第四条指令。为了讲清楚这个问题，用这个master与master之间的例子实在太过复杂和冗长，所以简单期间下面会用一个类似的例子来讲清楚这个东西到底想要干啥，下面都用完成代替看到，用memory代替master1）

就变成了：

**memory 完成 write D这个操作的时候，barrier之前的那些操作它一定都完成了。**

这样是不是好理解很多，那它到底想干一件怎么样的事呢？

那好我们把barrier去掉，并且再来看一下会发生什么：

如果MASTER0 做了以下步骤的指令

1.read A

2.read B

3.write C

4.write D

在这情形下，memory完成write D的时候可不一定read A,read B,write C都完成咯，什么意思？就是write D已经被完成了，可能还没轮到read A read B write C。这是为什么呢？**因为在一个relax-ordering的系统当中，程序的顺序并不一定是机器执行的顺序**，就算同样的顺序，由于路径不同等原因也会造成真正的执行过程不是按照程序执行顺序来的。

那么问题来了，如果我一定要前面三个指令做完了，才去write D,怎么办呢？ 所以就用到了memory barrier，**在硬件实现角度，master在看到barrier的时候，就会把后面的write D先挡住，然后发出这个barrier命令，硬件会保证在前面三条指令完成后，barrier命令才能成功完成，那么如果前面三条指令没有完成，write D这个指令就一直被挡着没有发出去直到barrier成功完成，也就保证了它和前面三个指令的次序关系。**

《A PRIMER ON MEMORY CONSISTENCY AND CACHE COHERENCE》在relax ordering那一章节中，有多个经典的多CORE应用barrier协同工作的例子，在此就不描述，有兴趣去看一下。

### 3.6.1 为什么需要内存屏障

<https://blog.csdn.net/Roland_Sun/article/details/106895899>（详见此博客）

内存屏障的引入，本质上是由于CPU重排序指令引起的。主要源自以下几种场景：

1. 编译器编译时的优化；
2. 处理器执行时的多发射和乱序优化；
3. 读取和存储指令的优化；
4. 缓存同步顺序（导致可见性问题）。

**处理器执行时的多发射和乱序优化**

现代处理器基本上都是支持多发射的，也就是在一个指令周期内可以同时执行多条指令。但是，处理器的资源就那么多，可能不能同时满足处理这些指令的要求。比如，处理器就只有一个加法器，如果同时有两条指令都需要算加法，那么有一条指令必须等待。如果这时候再下一条指令是读取指令，并且和前两条指令无关，那么这条指令将在前面某条加法指令之前完成。还有一种可能，就是前后指令之间具有相关性，比如对同一个地址先读取再写入，后面的写入操作必须等待前面的读取操作完成后才能执行。但是如果这时候第三条指令是写入一个无关的地址，那它可以在前面的写入操作之前被执行，执行顺序再次被打乱了。

所以，一般情况下指令乱序并不是CPU在执行指令之前刻意去调整顺序。CPU总是顺序的去内存里面取指令，然后将其顺序的放入指令流水线。但是指令执行时的各种条件，指令与指令之间的相互影响，可能导致顺序放入流水线的指令，最终不是按照放入的顺序执行完成，在外边看起来仿佛是“乱序”一样，这就是所谓的“顺序流入，乱序流出”。

为了维护这个MOESI（MESI）状态机，需要各个CPU之间进行通信，会引入下面几种类型的消息：

读消息：该消息包含要读取的缓存行的物理地址。

读响应消息：该消息包含较早前的读消息所请求的数据，这个读响应消息要么由物理内存提供，要么由某一个其它CPU上的缓存提供。例如，如果某一个CPU上的缓存拥有处于“Modified”状态的目标数据，那么该CPU上的缓存必须提供读响应消息。

使无效消息：该消息包含要使无效的缓存行的物理地址，所有其它CPU上的缓存必须移除相应的数据并且响应此消息。

使无效应答消息：一个接收到使无效消息的CPU必须在移除指定数据后响应一个使无效应答消息。

读使无效消息：该消息包含要被读取的缓存行的物理地址，同时指示其它CPU上的缓存移除对应的数据。因此，正如名字所示，它将读消息和使无效消息合并成了一条消息。读使无效消息同时需要一个读响应消息及一组使无效应答消息进行应答。

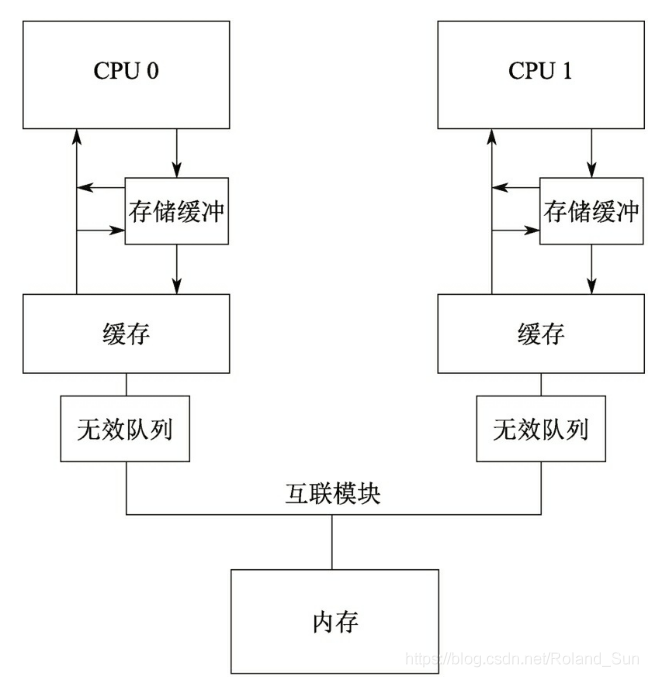
写回消息：该包含要回写到物理内存的地址和数据。这个消息允许缓存在必要时换出处于“Modified”状态的数据，以便为其它数据腾出空间。

通过上面的介绍可以看到，MESI缓存一致性协议可以保证系统中的各个CPU核上的缓存都是一致的。但是也带来了一个很大的问题，由于所有的操作都是“同步”的，必须要等待远端CPU完成指定操作后收到响应消息才能真正执行对应的存储或加载操作，这样会极大降低系统的性能。比如说，如果CPU0和CPU1上同时缓存了同一段数据，如果CPU0想对其进行更改，那么必须先发送使无效消息给CPU1，等到CPU1真的将该缓存的数据段标记成“Invalid”状态后，会向CPU0发送使无效应答消息，理论上只有CPU0收到这个消息后，才可以真的更改数据。但是，从要更改到真的能更改已经经过了好几个阶段了，这时CPU0只能等在那里。

鱼和熊掌都兼得是不可能的，想提高性能，只能稍微放松一下对缓存一致性的要求。具体的，会引入如下两个模块：

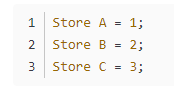
存储缓冲：前面提到过，在写数据之前我们先要得到缓存段的独占权，如果当前CPU没有独占权，要先让系统中别的CPU上缓存的同一段数据都变成无效状态。为了提高性能，可以引入一个叫做存储缓冲（Store Buffer）的模块，将其放置在每个CPU和它的缓存之间。当前CPU发起写操作，如果发现没有独占权，可以先将要写入的数据放在存储缓冲中，并继续运行，仿佛独占权瞬间就得到了一样。当然，存储缓冲中的数据最后还是会被同步到缓存中的，但就相当于是异步执行了，不会让CPU等了。并且，当前CPU在读取数据的时候应该首先检查其是否存在于存储缓冲中。

无效队列：如果当前CPU上收到一条消息，要使某个缓存段失效，但是此时缓存正在处理其它事情，那这个消息可能无法在当前的指令周期中得到处理，而会将其放入所谓的无效队列（Invalidation Queue）中，同时立即发送使无效应答消息。那个待处理的使无效消息将保存在队列中，直到缓存有空为止。



加入了这两个模块之后，CPU的性能是提高了，但缓存一致性就遭到了一定程度的破坏。假设变量X所在内存同时被两个CPU都缓存了，但是这时候CPU0对变量X的值做出了修改，这之后CPU1如果试图读取变量X的值时，有可能读到的是老的值，当然也有可能读到的是新的值。但是，在经过一段不确定的时间后，CPU1一定是可以读到变量X新的值，可以理解为满足所谓的最终一致性。

但这只是对单个变量来说的，如果程序中有多个变量，那么在其它CPU看来，它们之间的读写次序将完全无法得到保证。假设有CPU0上要执行对三个变量的写操作：



但是，这三个变量在缓存中的状态不一样，假设A变量和B变量在CPU0和CPU1中的缓存都存在，也就是处于“Shared”状态，而C变量是CPU0独占的，也就是处于“Exclusive”状态。假设系统经历了如下几个步骤：

在对变量A和B赋值时，CPU0发现其实别的CPU也缓存了，因此会将它们临时放到存储缓冲中。

在对变量C赋值时，CPU0发现是独占的，那么可以直接修改缓存的值，该缓存行的状态被切换成了“Modified”。注意，这个时候，如果在CPU1上执行了读取变量C的操作，其实已经可以读到变量C的最新值了，CPU1发送读消息，CPU0发送读响应消息，包含最新的数据，同时将缓存行的状态都切换成“Shared”。但是，如果这个时候如果CPU1尝试读取变量A或者变量B的数据，将会获得老的数据，应为CPU1上对应变量A和B的缓存行的状态仍然是“Shared”。

CPU0开始处理对应变量A和B的存储缓冲，将它们更新进缓存，但之前必须要向CPU1发送使无效消息。这里再次假设变量A的缓存正忙，而变量B的可以立即处理。那么变量A的使无效消息将存放在CPU1的无效队列中，而变量B的缓存行已经失效。这时，如果CPU1尝试获得变量B，是可以获得最新的数据的，而变量A还是不行。

CPU1对应变量A的缓存已经空闲了，可以处理当前无效队列的请求，因此变量A对应的缓存行将失效。直到这时CPU1才可以真正的读到变量A的最新值。

通过以上的步骤可以看到，虽然在CPU0上是先对变量A赋值，接着对B赋值，最后对C赋值，但是在CPU1上“看到”的顺序刚好是相反的，先“看到”C，接着“看到”B，最后看到“C”。在CPU1上会产生一种错觉，方式CPU0是先对C赋值，再对B赋值，最后对A赋值一样。这种由于缓存同步顺序的问题，让程序看起来好像指令被重排序了的情况，称作“伪”重排序。

### 3.6.2 读写内存屏障

写内存屏障仅仅限制了CPU对写操作的排序，对load操作没有任何效果，对其它的指令也没有作用。而且，写内存屏障只是保证在写内存屏障之后的写入操作一定是在写内存屏障之前的写入操作之后被系统其它组件感知，它并不能保证在写内存屏障之前的所有写入操作的顺序，也不能保证在写内存屏障之后的所有写入操作的顺序。

读内存屏障仅仅限制了CPU对加载操作的排序，对存储操作没有任何效果，对其它指令也没有任何作用。而且，读内存屏障只是保证在读内存屏障之后的读取操作一定是在读内存屏障之前的读取操作之后才去感知内存数据变化的，它并不能保证读内存屏障之前的所有读取操作顺序，也不能保证读内存屏障之后的所有读取操作的顺序。

写内存屏障只管自己CPU上的写入操作能够按照一定次序被系统中其它部件感知，但是如果其它部件有缓存将旧数据缓存下来了，这它管不着。这个是读内存屏障要管的事，读内存屏障只管自己CPU上的读取操作能够按照一定次序去感知系统内存中的值，但是对于其它CPU写入系统内存的次序没有任何约束。这个是写内存屏障要管的事，因此一般读内存屏障要和写内存屏障配对使用。

通用内存屏障（读写内存屏障）

一个通用内存屏障可以提供这样的保证，站在系统中其它组件的角度来看，通用内存屏障之前的加载、存储操作都将在通用内存屏障之后的加载、存储操作之前发生。

首先，通用内存屏障也会对处理器指令重排做出一些限制，也就是在通用内存屏障之前的写入和读取指令一定不会被重排序到通用内存屏障之后的写入和读取指令之后。其次，在执行通用内存屏障之后的任何写入和读取取指令之前，一定要保证清空当前CPU存储缓冲中的所有写操作，并且还要处理完当前CPU的无效队列。

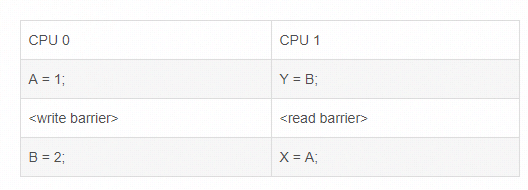
通用内存屏障等同于同时包含了读和写内存屏障的功能，因此也可以替换它们中的任何一个，只不过可能会一定程度上影响性能。

通用内存屏障同时限制了CPU对加载操作和存储操作的排序，但是对其它指令没有任何作用。而且，通用内存屏障只是保证在通用内存屏障之后的所有写入和读取操作一定是在通用内存屏障之前的写入和读取操作之后才执行，它并不能保证通用内存屏障之前的所有读取和写入操作的顺序，也不能保证通用内存屏障之后的所有读取和写入操作的顺序。

原文链接：<https://blog.csdn.net/Roland_Sun/article/details/106895899>

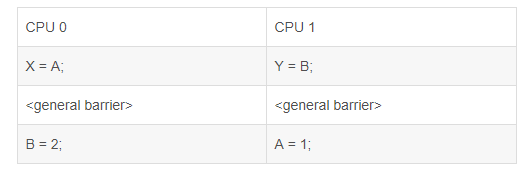
### 3.6.3 内存屏障的使用规则（配对使用场景）

首先，来看最常用的组合，一个CPU上执行两个写入操作，中间用写内存屏障分割，另一个CPU上执行两个读取操作，中间用读内存屏障分割：



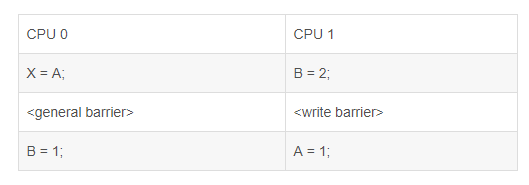
注意，在这种场景下写入变量的顺序和读取变量的顺序刚好要是相反的。加了这一对读、写内存屏障后，可以保证，在两个CPU都执行完上面的代码后，如果Y的值等于2，那么X的值一定等于1。Y的值等于2，意味着在CPU0上对B赋值2的语句已经执行过了，由于有写内存屏障的存在，也就意味着对A赋值1的语句在之前肯定也被执行过了。在CPU1上，由于有读内存屏障的存在，表示读取变量A值的语句一定在读取变量B值的语句之后被执行，也就可以保证，这时候变量A的值一定已经被赋值成了1。

第二种场景，两个CPU上都执行一个读取操作，接着一个写入操作，中间用通用内存屏障分割：



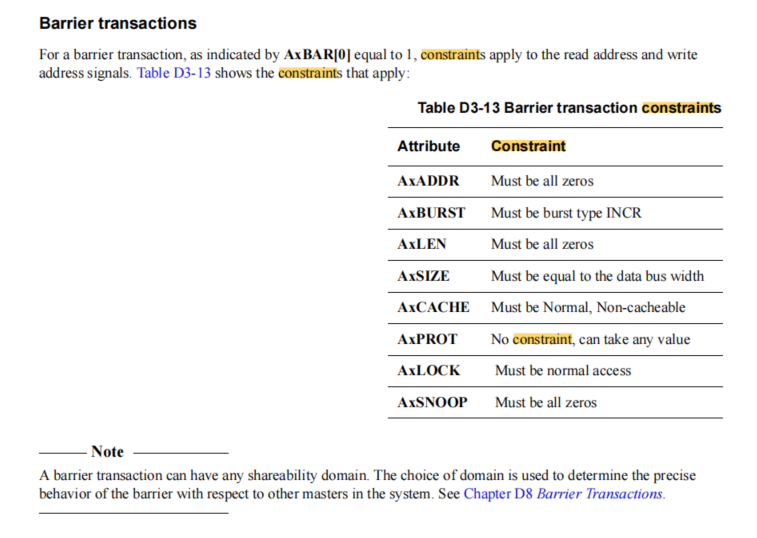
假设变量A和B的初始值都是0，当两个CPU都执行完上面的代码后，如果有Y等于2，那么X一定等于0。如果Y等于2，也就意味着在CPU1上对A赋值1的语句，一定在CPU0上读取变量A值的语句之后被执行。同时，由于对称性，如果有X等于1，那么Y一定等于0。

最后一种场景，一个 CPU 执行一个读取操作，后面跟一个通用内存屏障，再后面是一个写入操作；而另外一个CPU执行一对由写内存屏障分开的写入操作：



这种情况下，如果X的值等于1，那么必然有B的值等于1。如果X的值等于1，就意味着在这之前CPU1上已经执行过了对A赋值1的语句，由于写内存屏障的存在，也就能够保证在CPU1上已经执行过了对B赋值2的语句，而在CPU0上由于有通用内存屏障的存在，那么对B赋值1的语句一定会在对X赋值的语句之后执行。也就是说，可以保证在CPU0上对B赋值1的语句，一定会在CPU1上对B赋值2的语句之后被执行。

### 3.6.4 ACE中的Barrier transactions



Barrier transactions保证系统中transaction的ordering和observation。有两种类型的barrier transaction：

* **Memory barriers：**

Manager发起memory barrier来保证如果合适domain中的另外一个Manager能看到barrier后面的transaction，那么一定能看到barrier前的所有transaction。

* **Synchronization barriers：**

Manager发起synchronization barrier来确定在什么时候合适domain中每个Manager能够看到barrier前的所有transaction。对于System domain synchronization barrier，所有的transaction必须在barrier完成之前到达目标component。

屏障事务可以具有任何可共享性域。域的选择用于确定屏障相对于系统中其他主屏障的精确行为。

**Read barrier transactions**

主组件在读地址通道上发出读屏障事务，并在读数据通道上返回响应。没有数据传输。

**Write barrier transactions**

主组件在写地址通道上发出写障碍，并在写响应通道上返回响应。没有数据传输。

#### 3.6.4.1 Barrier transaction signaling 详见spec p286

AxBAR signaling

AxBAR signaling

Response signaling



#### 3.6.4.2 Master requirements

•barrier对中的两个事务必须具有相同的AxID、AxBAR、AxDOMAIN和AxPROT值。

•如果ARID和AWID信号有不同的宽度，较窄的版本必须进行零扩展以匹配较宽的版本。•在读地址和写地址通道上必须以相同的顺序发出屏障对。

•主接口不需要在同一周期内对读地址和写地址通道发出barrier事务。一个主接口允许发出多个未完成的barrier，这意味着在接收到之前的barrier事务响应之前，可以发出额外的barrier事务。

然而:

一个ACE主接口不能发出超过256个未完成的barrier事务

ACE-Lite主接口可以不受限制地发出未完成的障碍事务。

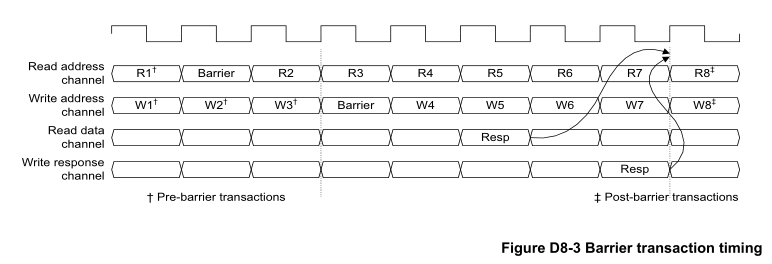
Note：读写响应握手是独立的事件，可以以任何顺序发生。因此，barrier被定义为从周期中第一个读或写barrier生效到周期中读写响应握手均发生为止。

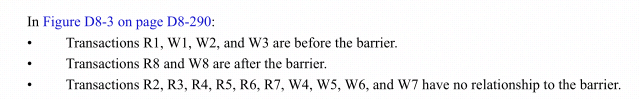
•与非Barrier事务相比，Barrier事务需要使用不同的ID值。允许屏障事务和非屏障事务使用相同的AXI ID值，前提是一个事务在另一个事务发出之前已经完成。

一个主组件必须以barrier对的形式发出barrier事务，在读地址通道和写地址通道上都有一个barrier事务。对于每个地址通道，在barrier事务之前在通道上发出的任何事务都被定义为在barrier之前，即使它是在另一个地址通道上的相应barrier之后发出的。一个事务被定义为在barrier之后，如果它是在读barrier响应和写barrier响应都被接收之后发出的。

在每个地址通道上，任何在该通道上的barrier之前发出的事务被定义为在barrier之前，即使它是在另一个地址通道上相应的barrier之后发出的。

D8-3显示障碍前和障碍后的交易，关于在地址通道上发出的障碍和收到的响应:





#### 3.6.4.3 Slave requirements

下面的规则适用于处理barrier事务的从组件：

1、在收到barrier transaction时，ACE-Lite从接口被允许:

.暂停读地址通道，直到它在写地址通道上收到相应的barrier事务。

. 暂停写地址通道，直到它在读地址通道上收到相应的barrier事务。

2、ACE从接口必须能够在写地址通道上接受256个barrier事务，而不会阻塞后续事务的进程。它要求写地址通道可用，并且写事务可以进展。

3、在接收到读屏障时，ACE从接口允许(但不是必需)停止读地址通道。

#### 3.6.4.4 Interconnect requirements

## Distributed virtual memory transactions

*Distributed Virtual Memory* (DVM) transactions用于虚拟内存系统维护，通常在DVM系统内的组件之间传递消息。



图5 Virtual memory system

Virtual memory system典型的处理过程：

1. Manager使用虚拟地址发起transaction。
2. SMMU收到虚拟地址后，准备转换为物理地址：

* 如果TLB中有该地址转换的缓存，那么直接得到物理地址。
* 否则，SMMU将执行translation table walk，访问memory中的translation table，从而得到对应的物理地址。

1. SMMU使用物理地址为Manager发起transaction。

有两种类型的DVM transaction：DVM message和DVM Complete。

DVM message支持以下操作：

* TLB Invalidate
* Branch Predictor Invalidate
* Physical Instruction Cache Invalidate
* Virtual Instruction Cache Invalidate
* Synchronization
* Hint

DVM transaction只能操作于一些只读的结构，比如Instruction Cache，Branch Predictor，TLB等，因此也只需要invalidate操作。

DVM Complete作为DVM Synchronization message的response，来指示所有要求的操作和相关的transaction都已完成。

# Signal Descriptions

https://mp.weixin.qq.com/s/7P8PabaKS1dub2rSB4WL3Q

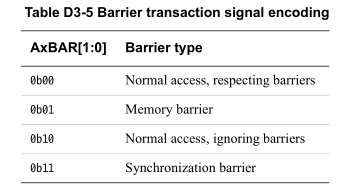
## Changes to existing AXI channels

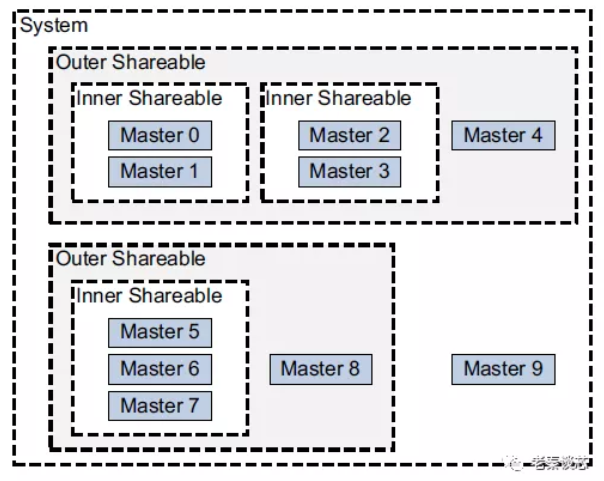
**Note**：**Write data channel (W)** 和**Write response channel (B)** 没有新增信号。

### (AR) signals added

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **ARSNOOP[3:0]** | Manager | 可共享读事务的事务类型 |
| **ARDOMAIN[1:0]** | Manager | ARDOMAIN[1:0] manager读事务的共享域 |
| **ARBAR[1:0]** | Manager | ARBAR[1:0] manager该事务是否为读障碍 barrier |

|  |  |
| --- | --- |
| **AxDOMAIN[1:0]** | **Domain** |
| 00 | Non-Shareable |
| 01 | Inner Shareable |
| 10 | Outer Shareable |
| 11 | System |



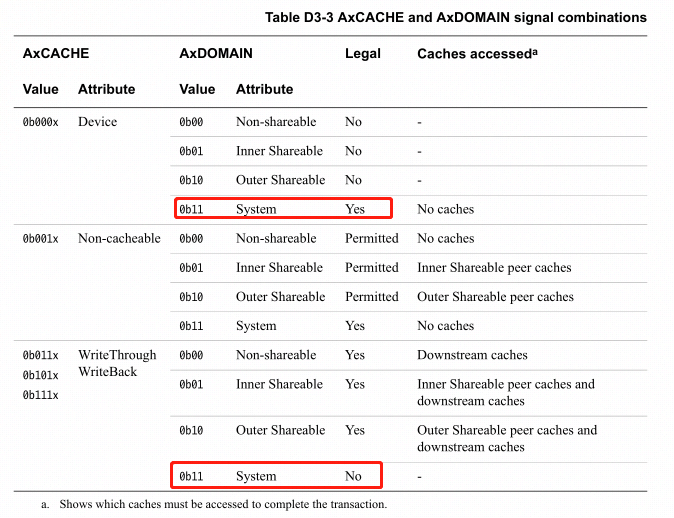


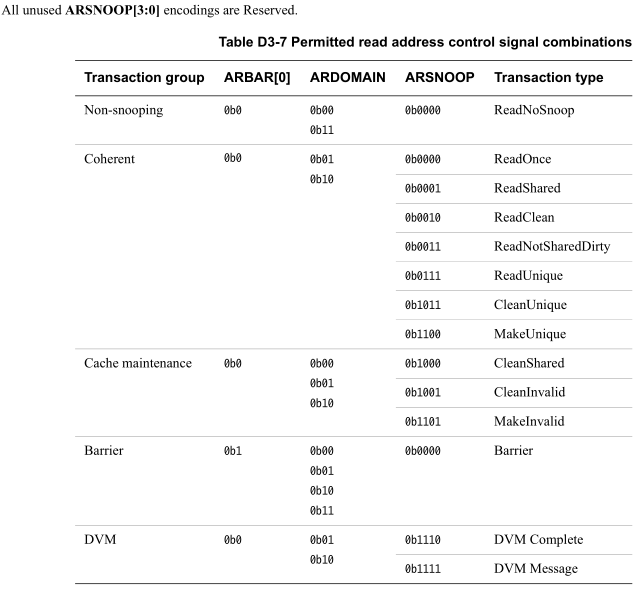
Non-shareable好理解，就是不与其它处理器共享；对于inner域的所有处理器来说，一个处理器的操作会被这个域内其它处理器看到，但是不会影响别的inner域；对于outer域，可以由多个inner域组成，如果一个处理器的操作影响到outer域，那么这个操作会被outer域包含的所有inner域看到；system也比较简单了，就是整个系统。定义这么多域的作用，其实就是要确定某一个操作，需要广播到哪一个层次。

Restrictions:

* Device transaction (AxCACHE[1] == 0)，必须只能使用domain level System。
* Cacheable transaction (AxCACHE[3:2] != 0)，不能使用domain level System。

表D3-3显示了所有AxCACHE和AxDOMAIN组合。有关AxCACHE编码的详细信息，请参见第A4-64页的AX4对内存属性信令的更改。

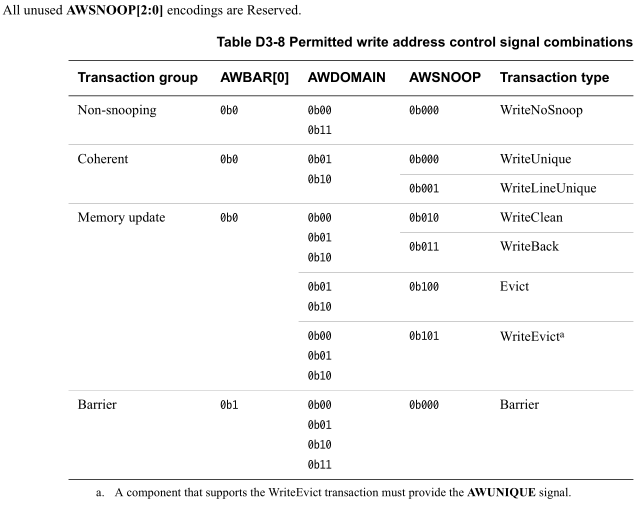


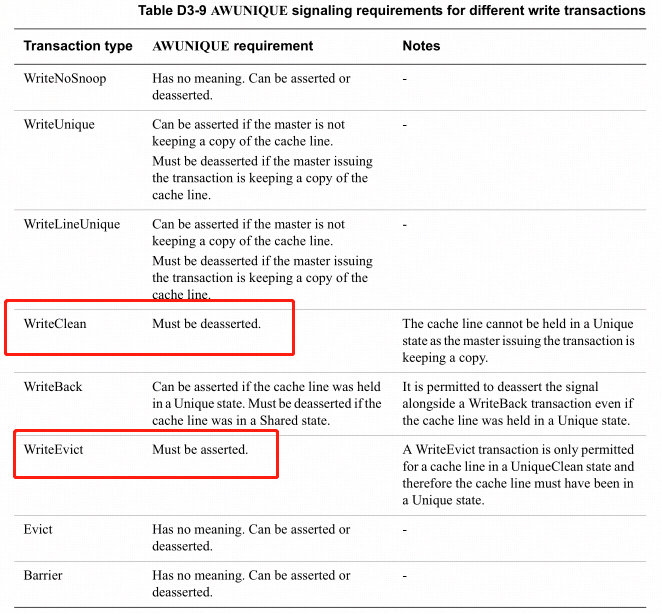


### (AW) signals added

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **AWSNOOP[2:0]** | Manager | 可共享写事务的事务类型 |
| **AWDOMAIN[1:0]** | Manager | 写事务的共享域 |
| **AWBAR[1:0]** | Manager | 指示事务是否为写障碍barrier |
| **AWUNIQUEa** | Manager | 指示允许此事务中的数据以Unique缓存状态保存 |

AWUNIQUE信号只有支持WriteEvict事务的组件才需要。





若transaction为**WriteClean**时，**AWUNIQUE**必须为0，因为Manager还留有备份，此cache line后续不能被持有为Unique状态。

若transaction为**WriteEvict**时，**AWUNIQUE**必须为1，因为此时cache line为UniqueClean，evict之后不会再有其它cache持有。

### (R) signals added

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **RRESP[3:2]** | Interconnect | 读响应，表示读传输的状态 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal** | **Source** | **Name** | **Meaning** |  |
| **RRESP[2]** | Interconnect | PassDirty | HIGH | Cache line相对main memory是dirty 的，**且**之后Initiating Manager需要负责将dirty cache line写回。 |
| LOW | Initiating Manager无需负责将该cache line写回main memory |
| **RRESP[3]** | Interconnect | IsShared | HIGH | 其它cache也拥有此备份，cache line为Shared状态。 |
| LOW | 相应的数据为唯一备份，cache line可以为Unique状态。 |

在一次burst中，**RRESP[3:2]**的值应该都相同。

IsShared和PassDirty响应具有以下限制：

•IsShared和PassDirty响应对于突发内的所有数据传输都必须是常量。

•对于需要删除所有其他缓存副本的事务，IsShared响应必须为低。需要删除所有其他缓存副本的事务有：ReadUnique、cleanuque 、MakeUnique CleanInvalid、MakeInvalid

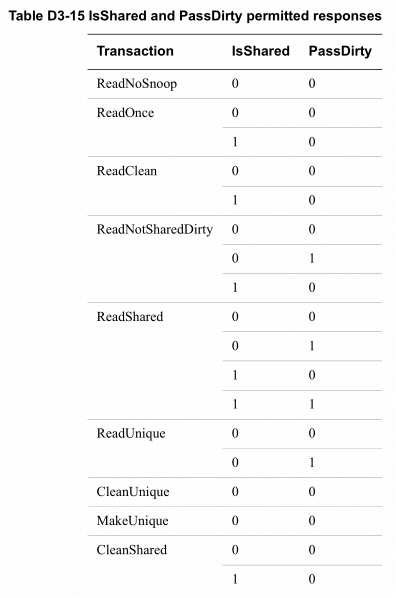
•对于不允许传递脏数据的事务，PassDirty响应必须为低。不允许传递脏数据的事务有：ReadOnce、ReadClean、cleanuque、MakeUnique、CleanShared、CleanInvalid MakeInvalid

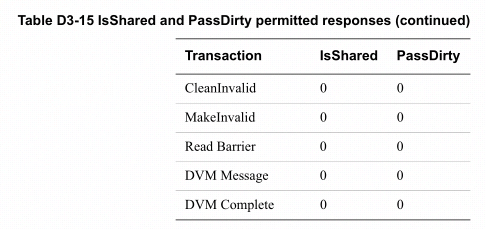
对于的以下事务，IsShared和PassDirty响应必须为低，因为它们对这些响应没有意义：ReadNoSnoop、Barrier transactions、DVM transactions

以下事务具有单读取数据通道传输：CleanUnique、MakeUnique、CleanShared、CleanInvalid、MakeInvalid、Barrier、DVM这些事务在单读取数据通道传输中完成，并且必须断言RLAST。RDATA可以有任何值，必须忽略。

EXOKAY响应仅允许用于ReadNoSnoop、ReadClean、ReadShared或CleanUnique事务。

每个事务允许的IsShared和PassDirty响应：





### 4.1.4 Cache line size restrictions(限制)

每个ACE主机可以支持的缓存线大小在设计时确定。

限制适用于可支持的最小和最大缓存线大小。

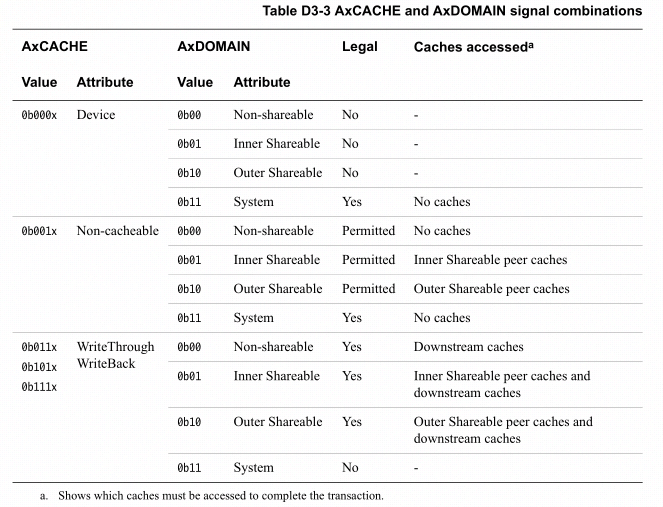
最小缓存线大小为16字节。

最大缓存线大小为以下较小值：

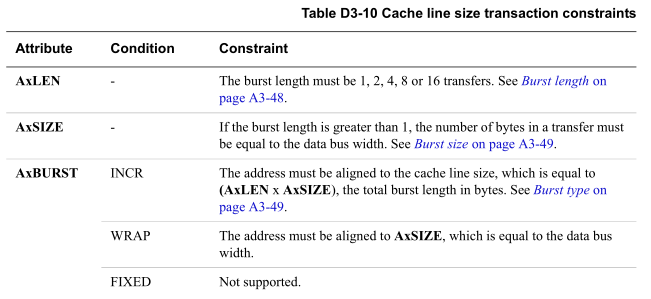
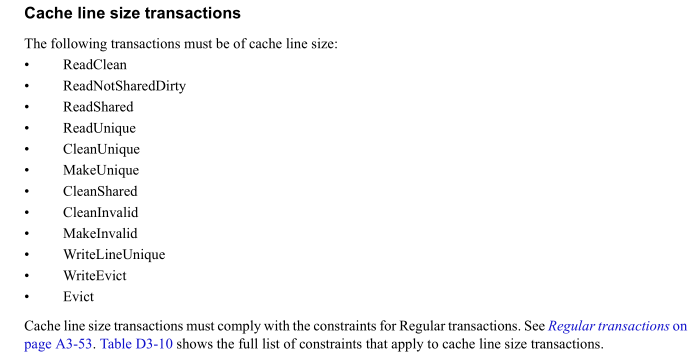
•2048字节

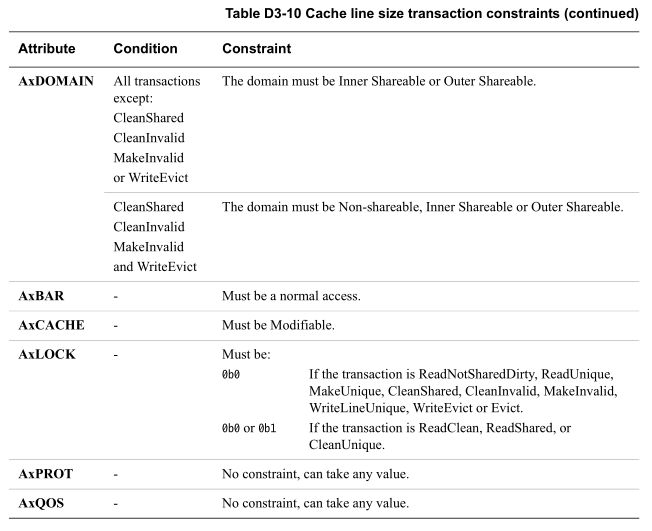
•最大突发长度16与数据总线宽度（字节）的乘积

### 4.1.5 Transaction constraints



#### 4.1.5.1 Cache line size transactions

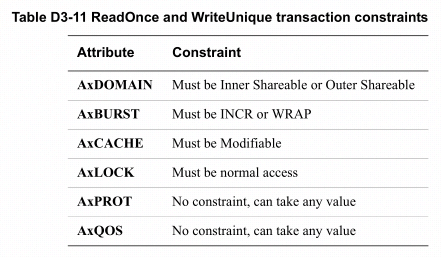




WriteLineUnique和WriteEvict事务需要断言每个写入数据选通（wstrb），也就是说，不允许使用稀疏写入数据选通

#### 4.1.5.2 ReadOnce and WriteUnique transactions

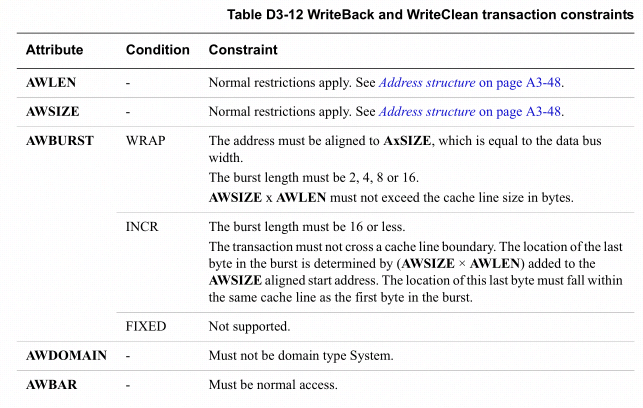
ReadOnce和WriteUnique事务不受缓存线大小的限制。

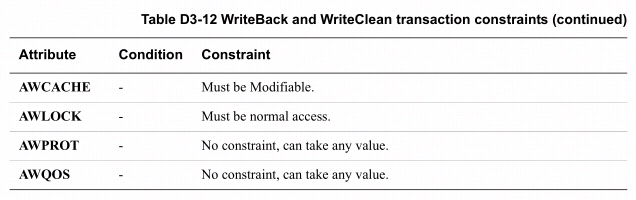


WriteUnique事务不需要断言每个写入数据选通，也就是说，允许稀疏写入数据选通

#### 4.1.5.3 WriteBack and WriteClean transactions

写回和写清除事务不受缓存线大小的限制。允许对缓存线进行部分更新。但是，写回和写清除事务仅限于单个缓存线内的更新。

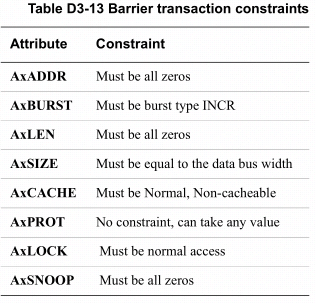




写回和写清除事务允许使用稀疏写数据选通。

#### 4.1.5.4 Barrier transactions

对于屏障事务，如AxBAR[0]等于1所示，约束适用于读地址和写地址信号。表D3-13显示了适用的约束条件：



## Additional channels defined by ACE

* **Snoop Address (AC) channel**是caching Manager的输入通道，提供了snoop transaction的地址和控制信息。
* **Snoop Response (CR) channel**是caching Manager的输出通道，为snoop transaction提供了response。每一个snoop transaction都会有一个response与之对应。
* **Snoop Data (CD) channel** (optional) 是Manager传输snoop data的输出通道。特别地，此通道的输出主要发生在read or clean transaction时，且Manager拥有需要返回的有效备份。



图6 ACE channels

### Snoop address channel (AC) signals

snoop地址通道（AC通道）对于以下主机是必需的：

•保存共享数据的缓存副本

•支持DVM事务snoop地址通道是缓存主机的输入通道。

snoop地址通道将其他组件的snoop事务传递给缓存的主机，以便主机可以确定它必须采取的操作。主机可以以不同的方式响应snoop事务，其响应决定互连必须采取什么操作来完成snoop过程。

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **ACVALID** | Interconnect | 表示snoop地址通道信号有效 |
| **ACREADY** | Manager | 表示可以接受snoop地址通道上的传输 |
| **ACADDR[ac-1:0]a** | Interconnect | 在Snoop transaction中第一次传送的地址 |
| **ACSNOOP[3:0]** | Interconnect | Snoop transaction 类型 |
| **ACPROT[2:0]b** | Interconnect | snoop transaction窥探事务的保护属性 |

1. ac是snoop地址总线的宽度。
2. The ACE specification only assigns meaning to **ACPROT[1]**.

* 适用标准的AXI **VALID/READY**握手规则。
* 在**ACVALID**置位时，地址（**ACADDR**）和控制（**ACSNOOP**, **ACPROT**）信号必须保持不变，直到**ACREADY**信号拉高。**ACVALID**置位之后必须保持直到**ACREADY**拉高。
* 允许**ACREADY**在**ACVALID**置位之前或同拍拉高，若还未发生握手，**ACREADY**可以再拉低。
* 对于coherency transactions，**ACPROT[1]**可以认为是定义了两个地址空间，一个Secure地址空间，一个Non-secure地址空间，可以被视为额外的地址位。硬件不需维护Secure和Non-secure地址空间之间的一致性。

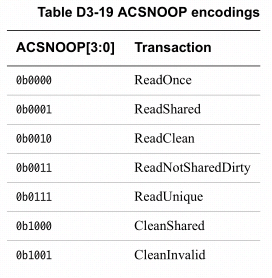
Snoop address channel (**AC**)上没有提供以下的控制信息：

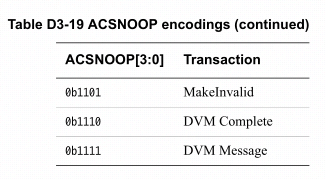
* Burst type
* Burst length
* Transaction size
* Modifiable or Shareable nature of transaction
* Transaction ID

Snoop transaction的burst length如果大于1，burst type必须为WRAP。突发长度为1的嗅探事务必须为突发类型INCR。

Snoop transaction的长度必须为full cache line。

snoop事务必须与snoop数据通道的宽度相同





### Snoop response channel (CR) signals

具体见spec D3.7部分

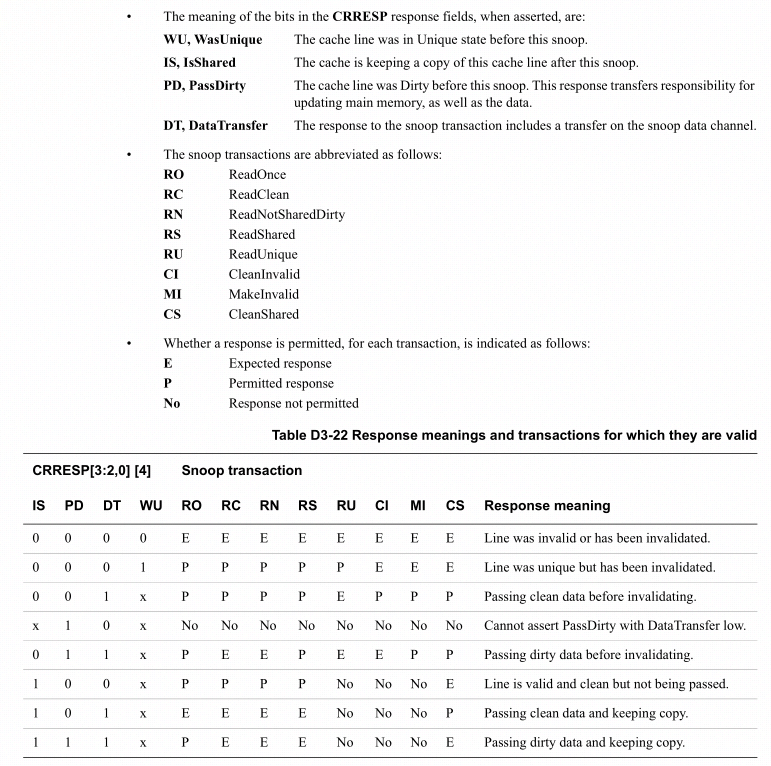
|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **CRVALID** | Manager | 表示snoop响应通道信号有效 |
| **CRREADY** | Interconnect | 指示可以接受snoop响应通道上的传输 |
| **CRRESP[4:0]** | Manager | 窥探响应，表示窥探传输的状态 |

* 适用标准的AXI **VALID/READY**握手规则。
* 在**CRVALID**置位时，**CRRESP**必须保持不变，直到**CRREADY**信号拉高。**CRVALID**置位之后必须保持直到**CRREADY**拉高。
* 允许**CRREADY**在**CRVALID**置位之前或同拍拉高，若还未发生握手，**CRREADY**可以再拉低。
* 所有的snoop transactions都是有序的。CR channel上的response顺序与AC channel上的address顺序保持一致。

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Name | Meaning |  |
| **CRRESP[0]** | DataTransfer | HIGH | 表示对这次transaction将会在CD channel上提供整个cache line的数据。 |
| LOW | 表示这次transaction在CD channel上没有数据传输。 |
| **CRRESP[1]** | Error | HIGH | 表示snooped cache line出错了，通常是因为ECC系统的检测。 |
| LOW | 表示未检测到错误。 |
| **CRRESP[2]** | PassDirty | HIGH | 表示snoop前cache line处于Dirty状态，且要把将cache line写回main memory的责任移交给Initiating Manager或者Interconnect。  对于除MakeInvalid以外的所有transactions，如果cache line为Dirty且cache line不再保留，该bit必须为高。 |
| LOW | 并不传递将cache line写回的责任。 |
| **CRRESP[3]** | IsShared | HIGH | 表示snooped cache在snoop transaction完成之后将继续保留cache line备份。 |
| LOW | 表示不再保留cache line的备份。 |
| **CRRESP[4]** | WasUnique | HIGH | 表示snoop前cache line处于Unique状态。只在已知其它cache没有备份的情况下为高。 |
| LOW | 在snoop前没有信息能确定cache line是否为Unique状态。 |

Illegal responses：

* IsShared，**CRRESP[3]** = 1 for:
  + ReadUnique
  + CleanInvalid
  + MakeInvalid
* PassDirty, **CRRESP[2]** = 1, and DataTransfer, **CRRESP[0]** = 0, for any transaction.



### Snoop data channel (CD) signals

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **CDVALID** | Manager | 表示snoop数据通道信号有效 |
| **CDREADY** | Interconnect | 这个信号表明在当前周期中可以接受snoop数据 |
| **CDDATA[cd-1:0]a** | Manager | Snoop data |
| **CDLAST** | Manager | 指示这是否是窥探事务中的最后一次数据传输 |

1. cd is width of snoop data bus

* 适用标准的AXI **VALID/READY**握手规则。
* 在**CDVALID**置位时，**CDDATA**必须保持不变，直到**CDREADY**信号拉高。**CDVALID**置位之后必须保持直到**CDREADY**拉高。
* 允许**CDREADY**在**CDVALID**置位之前或同拍拉高，若还未发生握手，**CDREADY**可以再拉低。
* Snoop data bus的位宽（**CDDATA**）并不要求与读写数据总线位宽相同。位宽可以为32，64，128，256，512，1024 bits。
* 当**CDVALID**置位时，**CDDATA**的所有byte lanes必须都为valid，也就是说不支持byte strobes。
* 当DataTransfer (**CRRESP[0]**)为1时，snoop data channel (**CD**) 需要被使用。Snoop data的顺序要与snoop address channel (**AC**) 一致。
* Burst length大于1的所有snoop transactions的burst type应定义为WRAP。Snoop burst数据传输的顺序与标准的wrapping burst相同。
* **CDLAST**必须在最后一次数据传输时拉高。

## Additional response signals and signaling requirements defined by ACE

### Read acknowledge signal（RACK）

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **RACK** | Manager | Read acknowledge signal |

**RACK**信号在最后一笔读数据传输（**RLAST**）的**RVALID/RREADY**握手完成之后置位。

### Write acknowledge signal(WACK)

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| **WACK** | Manager | Write acknowledge signal |

**WACK**信号在**BVALID/BREADY**握手完成之后置位。

### Reset requirements

ACE协议使用AXI的**ARESETn**信号。在复位期间：

* Manager需要驱动**RACK**, **WACK**, **CRVALID**, **CDVALID**为低。
* Interconnect需要驱动**ACVALID**为低。

# Transaction processing（事务处理）

1. Initiating Manager发起transaction。
2. 决定于是否需要coherency的支持，transaction：
   * 根据所使用的地址译码方案，直接传递给Subordinate（从属）。
   * 传递给interconnect内的coherency support logic。
3. 和后续来自其它Manager的transaction做检查，确保正确的处理顺序。
4. Interconnect确定是否需要发起snoop transaction。
5. 每个收到snoop transaction的caching Manager必须提供snoop response，可能还需提供snoop data。
6. Interconnect确定是否需要访问main memory。
7. Interconnect核对snoop response和所需的数据。
8. Initiating Manager完成transaction。

## Snoop channel dependencies

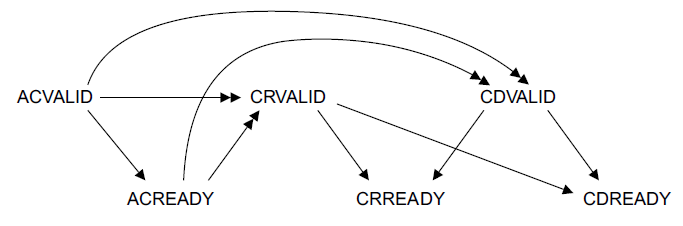


图7 Snoop channel dependencies

Manager发起的**CRVALID**和**CDVALID**必须在**ACVALID**和**ACREADY**握手之后才能拉高。

## Channel usage example

### Performing load operations from Shareable locations

场景：Manager的local cache中没有备份。

1. Initiating Manager通过read address channel (**AR**) 发起read transaction。
2. Interconnect通过snoop address channel (**AC**) 向其它caching Manager传递transaction。
3. Snooped Managers通过snoop response channel (**CR**) 回复response，同时，可选择地在snoop data channel (**CD**) 上提供数据。
4. 如果其中的Manager有提供数据，interconnect将通过read data channel (**R**) 将数据回复给initiating Manager。
5. 如果没有Manager提供数据：
   1. Interconnect向main memory发起transaction。
   2. 读数据从read data channel (**R**) 返回Manager。
6. Manager通过**RACK**信号指示transaction已完成。



图8 transaction过程示意图

### Performing store operations to Shareable locations

当Manager要对一个处于Shareable location的cache line发起store操作时，需要remove该cache line的所有其它备份，以确保执行store时Manager拥有唯一备份。

#### Store operations for a partial cache line

场景：Manager的local cache中没有备份。

1. Initiating Manager通过read address channel (**AR**) 发起**ReadUnique** transaction，请求remove其它备份。
2. Interconnect通过snoop address channel (**AC**) 向其它caching Manager传递transaction。
3. Snooped Managers通过snoop response channel (**CR**) 回复response，同时，可选择地在snoop data channel (**CD**) 上提供数据。
4. 如果其中的Manager有提供数据，interconnect将通过read data channel (**R**) 将数据回复给initiating Manager。
5. 如果没有Manager提供数据：
   1. Interconnect向main memory发起transaction。
   2. 读数据从read data channel (**R**) 返回Manager。
6. Manager执行store同时通过**RACK**信号指示transaction已完成。

#### Store operations for an entire cache line

**Note**：Store整个cache line并不需要先获取cache line。

1. Initiating Manager通过read address channel (**AR**) 发起**MakeUnique** transaction，请求remove其它备份。
2. Interconnect通过snoop address channel (**AC**) 向其它caching Manager传递transaction。
3. Snooped Managers通过snoop response channel (**CR**) 回复response，指示cache line已经被remove。无需在snoop data channel (**CD**) 上传输数据。
4. Interconnect通过read data channel (**R**) 将response传递给initiating Manager。
5. Manager执行store同时通过**RACK**信号指示transaction已完成。

#### Store operations where the cache line is already cached

场景：Manager已经拥有该cache line处于Shared状态的备份。

1. Initiating Manager通过read address channel (**AR**) 发起**CleanUnique** transaction，请求remove其它备份。
2. Interconnect通过snoop address channel (**AC**) 向其它caching Manager传递transaction。
3. Snooped Managers通过snoop response channel (**CR**) 回复response，指示：

* cache line已经成功被remove。
* 是否有dirty cache line需要被interconnect写回main memory。

1. 如果有dirty cache line需要写回，则通过snoop data channel (**CD**) 提供数据。Interconnect之后会发起transaction将dirty cache line写回main memory。
2. Interconnect通过read data channel (**R**) 将response传递给initiating Manager。
3. Manager执行store同时通过**RACK**信号指示transaction已完成。

### Write transaction

The write transaction group is:

* WriteNoSnoop
* WriteUnique
* WriteLineUnique
* WriteBack
* WriteClean
* WriteEvict

Write transaction的过程如下：

1. 地址在write address channel (**AW**) 上发送。
2. 数据在write data channel (**W**) 上传送。
3. Response在write response channel (**B**) 上返回。
4. Manager通过**WACK**信号来指示write transaction已完成。

## Coherency Transactions on the Read Address and Write Address Channels

### 5.3.1 About an initiating master

本节描述发起主机的行为。通常，发起主机会发出一个事务，以执行内部操作，如load加载或store存储操作。

内部操作要求：

•对于load，主机必须从以下任一位置获取数据：

-相应缓存线的有效副本

-返回有效读取数据的事务

•对于store，主机需要从以下任一位置存储缓存线的权限：

-处于unique状态的相应缓存线的副本

-授予主机存储缓存线权限的事务类型

#### 5.3.1.1 Transaction groups

下面几节描述了事务组的预期通道活动:

**Read transactions**

The read transaction group is:

• ReadNoSnoop

• ReadOnce

• ReadClean

• ReadNotSharedDirty

• ReadShared

• ReadUnique

读取事务的进程如下：

1。地址在读地址（AR）通道上发出。

2.数据和响应在读取数据（R）通道上返回。所需的数据拍数由ARLEN确定。

3.主机架发出读取事务完成信号的断言。

**Clean transactions**

The clean transaction group is:

• CleanUnique

• CleanShared

• CleanInvalid

Clean事务流程如下:

1. 地址在AR通道上发布。

2. R通道上的单个传输返回响应。Clean事务不返回任何数据。

3.Clean事务的完成由主机断言RACK发出信号。

**Make transactions**

The make transaction group is:

• MakeUnique

• MakeInvalid

对于发起主机，Make事务的进程如下：

1.地址在AR通道上发出。

2.R通道上的单个传输返回响应。Make事务不会返回任何数据。

3.主机发出Make事务完成的信号断言

**Write transactions**

The write transaction group is:

• WriteNoSnoop

• WriteUnique

• WriteLineUnique

• WriteBack

• WriteClean

• WriteEvict

对于发起主机，写入事务的进程如下：

1．该地址在AW通道上发布。

2.数据在W通道上传输。

3.在B通道上返回响应。

4.写入事务的完成由主断言W ACK发出信号。

**Evict transactions**

The evict transaction group is, Evict.

对于启动主机，逐出事务的进程如下：

1．该地址在AW通道上发布。

2.在B通道上返回响应。没有为逐出事务传输数据。

3. 逐出事务的完成由主断言W ACK发出信号。

**Read barrier transactions**

对于发起事务的主机，读屏障事务的进程如下：

1．该事务在AR通道上发布。

2. R通道上的单个传输返回响应。读取障碍事务不会返回任何数据。

3.主机发出读屏障事务完成的信号断言。

**Write barrier transactions**

对于发起事务的主机，写屏障事务的进程如下：

1．交易在AW通道上发布。

2.在B通道上返回响应。写入障碍事务不传输任何数据。

3.写屏障事务的完成由主机断言W ACK发出信号。

**DVM transactions**

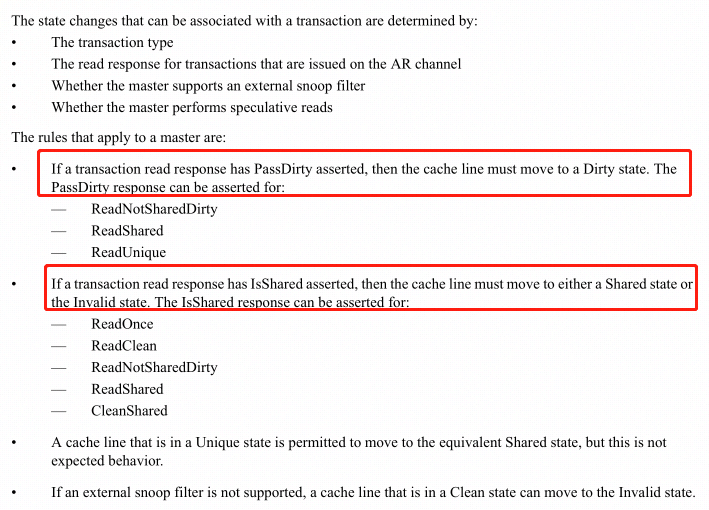
对于发起事务的主机，DVM事务的进程如下：

1．该事务在AR通道上发布。

2.R通道上的单个传输返回响应。DVM事务不返回任何数据。

3.主断言机架发出DVM事务完成的信号。

## State changes on different transactions



### 与load相关的状态更改（State changes associated with a load）

load的内部操作不需要改变缓存线状态。

### 与一致store相关联的状态变化（State changes associated with a coherent store）

在对共享内存中的高速缓存线执行内部存储操作之前，master必须确保它有存储权限。如果缓存线处于UniqueClean或UniqueDirty状态，则master具有存储权限。

如果主机没有存储权限，则必须：

1. 在AR通道上发出获得存储权限的事务，然后对缓存线执行存储。在存储到缓存线之后，主机必须处于UniqueDirty状态。获得存储权限的事务包括：ReadUnique、CleanUnique、MakeUnique
2. 在AW通道上发出一个事务，该事务获得存储和更新主内存的权限。获得存储数据和更新主存权限的事务包括：WriteUnique、WriteLineUnique

### 与主存更新关联的状态更改

当缓存线处于脏状态时，可以执行对主内存的更新。

当主机被授予更新主存的权限时，相关写事务最早可能发生的周期是RVALID/RREADY握手后的周期，在该握手中，RLAST为授予更新主存权限的事务断言。

使用write back或write clean执行对主内存的更新

主内存更新后，缓存线必须处于干净或无效状态。

如果支持外部snoop筛选器，则适用以下限制：

•写回事务后，缓存线必须处于无效状态。

•WriteClean事务后，缓存线必须处于干净状态。

### 与缓存维护操作关联的状态更改

缓存维护事务是CleanShared、CleanInvalid、MakeInvalid

在发出缓存维护事务之前，主机必须确保：

* 1. 对于CleanShared，缓存线必须处于clean或Invalid状态
  2. 对于CleanInvalid和MakeInvalid，缓存线必须处于Invalid状态。

缓存维护事务不会更改缓存线状态。

### State change descriptions

参考specD4.4

### ReadShared

|  |  |  |  |
| --- | --- | --- | --- |
| **Transaction** | **Start state** | **RRESP[3:2]**  **IsShared/PassDirty** | **Expected end state** |
| ReadShared | I (Invalid) | 00 | UC (Exclusive) |
| 01 | UD (Modified) |
| 10 | SC (Shared) |
| 11 | SD (Owned) |

* 如果snooped Manager可能是下一个写该cache line的Manager，协议推荐snooped Manager保留Dirty状态，传递Clean给initiating Manager。
* 如果initiating Manager可能是下一个写该cache line的Manager，协议推荐传递Dirty给initiating Manager。
  + 如果initiating Manager可能对该cache line的store早于snooped Manager的load，协议推荐snooped Manager直接不保留该cache line的备份。
  + 如果snooped Manager可能对该cache line的load早于initiating Manager的store，协议推荐snooped Manager保留该cache line的备份。
* 如果不知道谁可能是下一个写该cache line的Manager，协议推荐最小可能被evict的cache line中为Dirty。那这通常是initiating Manager，因为它是最近被访问的。

### ReadUnique

|  |  |  |  |
| --- | --- | --- | --- |
| **Transaction** | **Start state** | **RRESP[3:2]**  **IsShared/PassDirty** | **Expected end state** |
| ReadUnique | I (Invalid) | 00 | UC (Exclusive) |
| 01 | UD (Modified) |

### CleanUnique

|  |  |  |  |
| --- | --- | --- | --- |
| **Transaction** | **Start state** | **RRESP[3:2]**  **IsShared/PassDirty** | **Expected end state** |
| CleanUnique | SC (Shared) | 00 | UC (Exclusive) |
| SD (Owned) | 00 | UD (Modified) |

### MakeUnique

|  |  |  |  |
| --- | --- | --- | --- |
| **Transaction** | **Start state** | **RRESP[3:2]**  **IsShared/PassDirty** | **Expected end state** |
| MakeUnique with full cache line store | I (Invalid) | 00 | UD (Modified) |
| SC (Shared) | 00 | UD (Modified) |
| SD (Owned) | 00 | UD (Modified) |

### WriteBack

|  |  |  |
| --- | --- | --- |
| **Transaction** | **Start state** | **Expected end state** |
| WriteBack | UD (Modified) | I (Invalid) |
| SD (Owned) | I (Invalid) |

### WriteClean

|  |  |  |
| --- | --- | --- |
| **Transaction** | **Start state** | **Expected end state** |
| WriteClean | UD (Modified) | UC (Exclusive) |
| SD (Owned) | SC (Shared) |

## Handling overlapping write transactions

当2个Manager对处于Shareable存储空间的同一cache line同时发起store操作，此时interconnect需要对这2个请求进行排序。为了简单表示，下面把排在前面的叫Manager1，后面的叫Manager2。

### Overlapping ReadUnique

如果Manager2因为需要数据副本而发出ReadUnique事务，则会发生以下情况：

1. Manager2发出一个ReadUnique事务。
2. Manager2会在snoop port上收到来自Manager1的以下transaction之一试图写入同一行：

* ReadUnique
* CleanInvalid
* MakeInvalid

此时Manager2的local cache中如果有对应的备份，则需要invalidate，没有则无需操作。

1. 当Manager2发起的**ReadUnique**完成时，会返回已经被Manager1更新过的cache line。
2. Manager2执行store操作。

### Overlapping MakeUnique

1. Manager2要写整个cache line，发起**MakeUnique** transaction。
2. Manager2会在snoop port上收到来自Manager1的以下transaction之一试图写入同一行：

* ReadUnique
* CleanInvalid
* MakeInvalid

此时Manager2的local cache中如果有对应的备份，则需要invalidate，没有则无需操作。

1. 当Manager2发起的**MakeUnique** transaction完成时，执行full cache line store。

### Overlapping CleanUnique

1. Manager2中已经有cache line的备份，执行store时发起**CleanUnique** transaction。
2. Manager2会在snoop port上收到来自Manager1的以下transaction之一试图写入同一行：

* ReadUnique
* CleanInvalid
* MakeInvalid

此时Manager2的local cache中如果有对应的备份，则需要invalidate，没有则无需操作。

1. 当CleanUnique完成时，Manager2无法执行其本地存储，因为它丢失了缓存线的本地副本。
2. Manager2可以重新发起一个新的**ReadUnique** transaction去获取一个cache line的备份。
3. Manager2执行store。

主服务器可以通过最初发出ReadUnique事务而不是CleanUnique事务来消除在事务中发出新的ReadUnique的需要，如CleanUnique案例中所述。但是，这有时会导致在不需要时从主内存获取数据。

还有一种做法就是，不再重新发一个新的**ReadUnique** transaction，而是在**CleanUnique**完成，有写权限之后，直接写到main memory，只用更新需要的几个byte。Manager2的local cache中将不再保留对应的cache line。

## Memory update in progress

协议规定，2个component不能同时更新main memory的同一区域。

如果snooped Manager在它使用**WriteBack**或**WriteClean**更新main memory的时候，接收到一个snoop transaction，那么这时snooped Manager应当确保没有其它的Manager能同时更新main memory的同一位置。要想实现这一点，有以下两种方法：

* 给出PassDirty为0且IsShared为1的snoop response，这样就不会给出该cache line的写权限，同时也不会把更新memory的责任移交出去。
* Delay snoop response直到snooped Manager更新main memory完成。

当snooped Manager要传递写权限，即对应的snoop response给到CR channel之前，所有要更新main memory的write transactions应当都已完成。

## 5.7 Snoop Transaction

本章描述在snoop地址通道上看到的snoop事务。描述了所需的和协议推荐的snoop事务行为

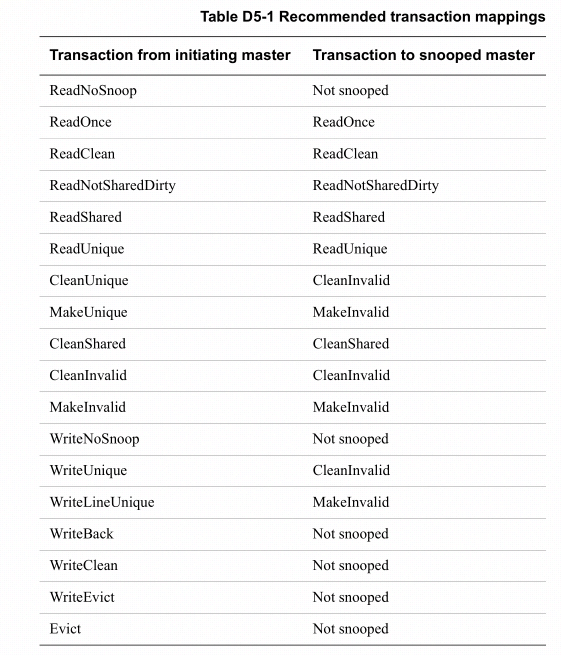
### 5.7.1 将一致性操作映射到snoop操作（Mapping coherency operations to snoop operations）

#### 5.7.1.1 支持的映射

本节描述由发起主机监视的缓存主机在snoop地址通道上看到的snoop事务

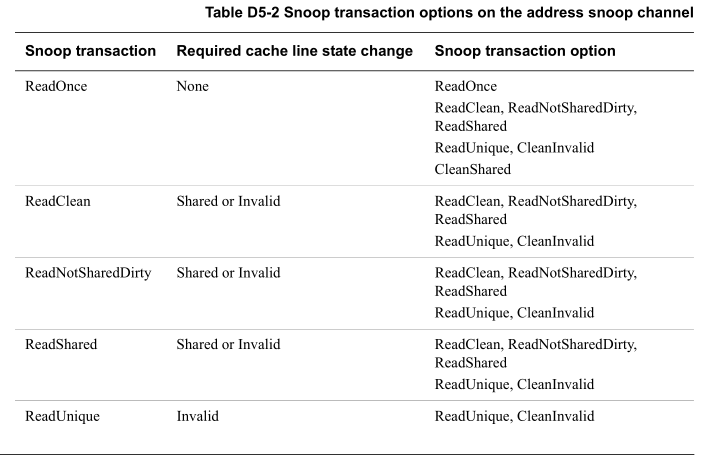
当发起主机发出事务时，互连负责执行完成原始事务所需的任何snoop事务。

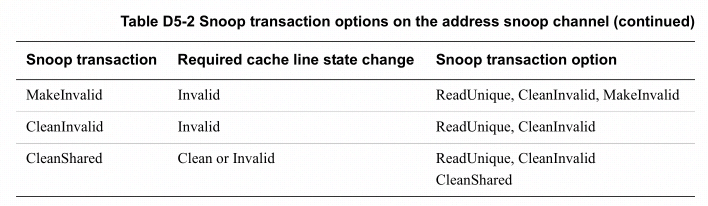
并非发起主机发出的所有事务都允许在snoop地址通道上进行。表D5-1显示了发起主机发出的事务与缓存主机在snoop地址通道上看到的snoop事务之间的协议建议映射。



互连可以使用其他映射来强制窥探主机中相同的缓存线状态更改，见D5-2

D5-2显示snoop地址通道上每个允许的snoop事务、该事务所需的缓存线状态更改以及可使用的备用snoop事务。为完整起见，snoop事务选项列包括原始snoop事务。

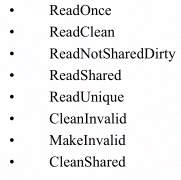




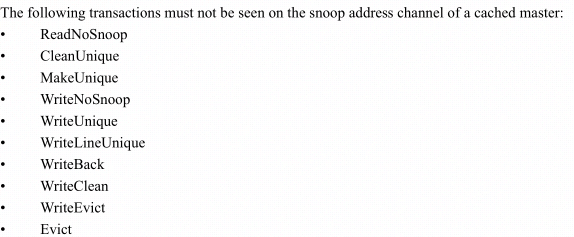
映射到不同的嗅探事务可以简化嗅探主机的设计。例如，窥探主机可以以与ReadUnique事务相同的方式处理所有窥探事务。这是允许的，因为ReadUnique事务（如第D5-233页的表D5-2所示）是所有其他snoop事务的替代snoop事务

#### 5.7.2 允许的窥探交易

尽管该协议不需要一组固定的事务映射，但该协议确实要求在缓存主机的snoop地址通道上只能看到以下定义的事务子集



#### 5.7.3 不允许作为窥探事务的事务



### 5.7.2 snoop事务的一般要求（General requirements for snoop transactions）

对于每个snoop事务，协议都指定了所需和建议的行为：

见spec D5.2

### 5.7.3 snoop事务

本节介绍每个snoop事务，并提供有关存在选项的建议行为的信息。

见spec D5.3



# Interconnect Requirements

## About the interconnect requirements

It is the responsibility of interconnect to:

* 接收initiating Manager 发起的transaction。
* 当同时接收到多个transactions时，决定它们的顺序。
* 根据需要，发起snoop transactions。
* 接收snoop response和data。
* 产生回给initiating Manger的response。
* 必要时访问main memory。

## Sequencing transactions

多个Manager可能会同时发起transaction，因此interconnect有责任确保对同一cache line发生的transaction有一个定好的顺序，并且这个顺序对于所有的component都是一样的。在2个Manager同时对同一cache line发起transaction时，interconnect需要决定先后顺序。仲裁的方法协议没有定义。

Interconnect通过对transaction response和snoop transaction进行排序来指示对同一cache line的transaction的顺序。Ordering rules：

* 如果Manager对一个cache line发起Coherent or Cache Maintenance transaction，并且在它得到response前接收到了对相同cache line的snoop transaction，此时snoop transaction优先。
* 如果Manager对一个cache line发起Coherent or Cache Maintenance transaction，并且在它接收对相同cache line的snoop transaction前已经得到了response，此时Manager发起的transaction优先。

**Note**：以上的顺序仅针对于对同一cache line的transaction。



图9 Sequencing transaction

要满足这种规则，那么interconnect必须确保：

* 如果interconnect已经在向Manager提供Coherent or Cache Maintenance transaction的response了，那么就不应再向Manager发起同一cache line的snoop transaction，直到收到来自Manager的**RACK**或**WACK**。
* 如果interconnect给Manager发起snoop transaction，那么就不应再向Manager提供对同一cache line的Coherent or Cache Maintenance transaction的response了，直到收到来自Manager的**CRRESP**。

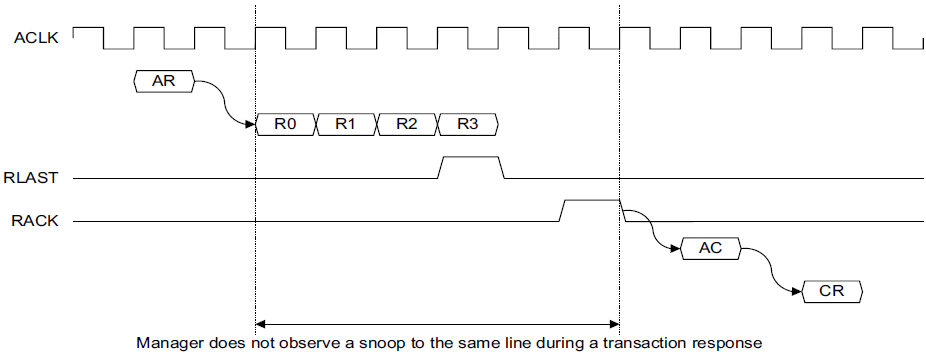


图10 Transaction response before a snoop transaction

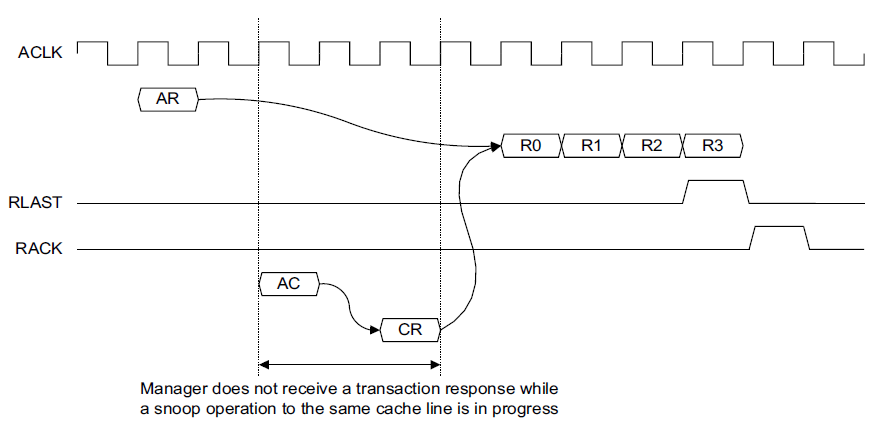


图11 Snoop transaction before a transaction response

## Interactions with main memory

### Interconnect read from main memory or peripheral device

对于**ReadNoSnoop** transaction，interconnect会从main memory或外设读取。

对于下述transaction，如果interconnect没有从snoop transaction得到数据，那么就会从main memory中读数据以完成transaction：

* **ReadOnce**
* **ReadClean**
* **ReadNotSharedDirty**
* **ReadShared**
* **ReadUnique**

Interconnect允许在所有的snoop transaction完成之前从main memory中读数据，但是有以下的规则：

* 当某个处于shareability域的cache中如果有对应cache line的备份是dirty时，从main memory中获得的数据不能被使用。因此，如果有cache line通过snoop transaction被提供时，从main memory中获得的数据不能被使用。

**Note**：Snoop response不会指示cache中是否有dirty的备份，仅仅表示是否要传递更新main memory的责任。

* 从main memory读出的数据若可能与所有相应的snoop transaction完成之后读出的数据不一样，则该数据不能被使用。举个例子，如果对main memory的读请求发起时，有**WriteBack**或**WriteClean**还未完成，那么获得的数据不能被使用，需要发起新的读请求以获得正确的数据。

### Main memory update that is generated by the interconnect

以下transaction通过interconnect直接传递到main memory：

* **WriteNoSnoop**
* **WriteBack**
* **WriteClean**

对于**WriteUnique**和**WriteLineUnique**来说，interconnect还需要执行对应的snoop transaction。这时如果收到了PassDirty为1的snoop response，那么最终memory中的结果应与先写dirty cache line再写**WriteUnique**或**WriteLineUnique**中的数据一致。实现的例子如下：

* 写数据的顺序为：
  1. 从snoop中得到的dirty cache line写回main memory。
  2. **WriteUnique**或**WriteLineUnique**的数据写到main memory。
* **WriteUnique**或**WriteLineUnique**的数据merge到获得的dirty cache line中，**WriteUnique**或**WriteLineUnique**中valid byte覆盖dirty cache line中的相应byte。执行单次的写操作将merge后的数据写入main memory。

## Other requirements

### Non-blocking requirements

* 任何AR channel上的transaction可以被stall，等待AC channel上的transaction。
* 任何AC channel上的snoop transaction可以被stall，等待AW channel上的write transaction (除了**WriteUnique**和**WriteLineUnique**)。

### Permitted transaction modifications

通过Modifiable属性的定义，即**AxCACHE[1]**，允许互联改变transaction：

* 一个transaction可以被拆分为多个transaction。
* 多个transaction可以被merge成一个transaction。
* Read transaction可以获得比原本需求更多的数据。
* Write transaction可以访问比原来需求更大的地址空间，通过Write strobes来确保仅更新所需的内存位置。
* 在每个transaction中，以下信号可以被改变：
  + The transfer address, **AxADDR**
  + The burst size, **AxSIZE**
  + The burst length, **AxLEN**
  + The burst type, **AxBURST**

**Note**：interconnect对transaction的改变，对Manager来说是不可见的。

# ACE-Lite

ACE-Lite主要使用于自身没有硬件一致性cache，但是又有以下要求的组件：

* 指示发起的transaction是否能被带有硬件一致性cache的Manager处理。
* 发起barrier transaction。
* 发起广播的cache maintenance operations (CMO)。

ACE-Lite在AXI4的read address channel (**AR**)和write address channel (**AW**)上添加了一些信号，与ACE在这2个channel上添加的信号一致。

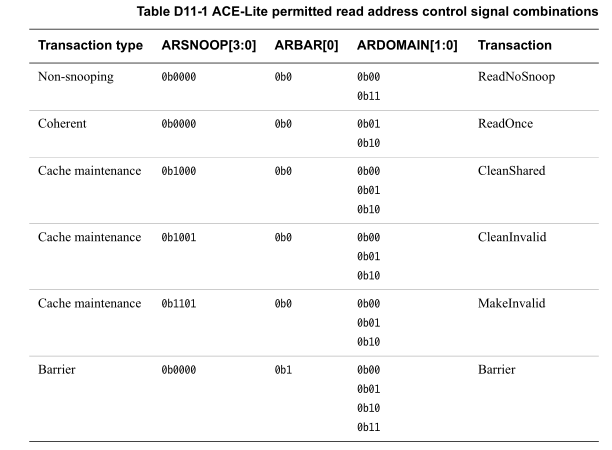
ACE-Lite**不包括**：

* Snoop address channel (**AC**)
* Snoop response channel (**CR**)
* Snoop data channel (**CD**)
* Read acknowledge signal (**RACK**)
* Write acknowledge signal (**WACK**)
* ACE-specific read response bits (**RRESP[3:2]**)

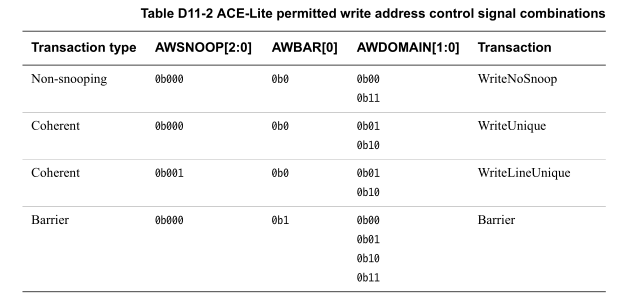
## 7.1 ACE\_Lite signal requirements

ACE-Lite接口可以发出所有非共享事务，但只能使用一组受限制的可共享事务类型。

表D11-1给出了每个可共享读事务允许的ARSNOOP[3:0]、ARBAR[0]和ARDOMAIN[1:0]的组合。



表D11-2显示了允许的AWBAR[0]和AWDOMAIN[1:0]的组合对于每个允许的可共享写事务类别。



# Cache Coherent Interconnect

## CCI-400

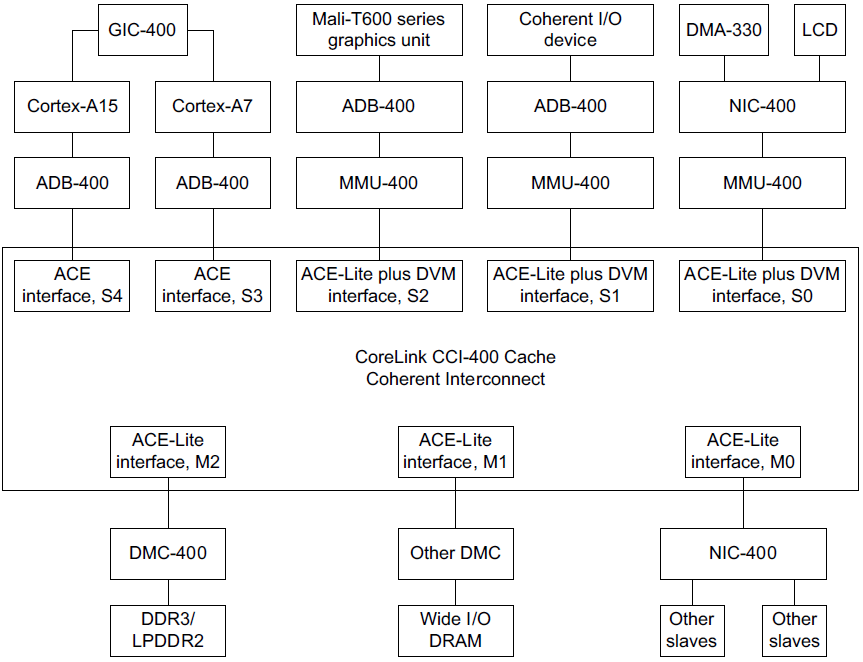


图12 Example system with a CCI-400

想要实现硬件缓存一致性，最简单的方法是广播snoop到所有的cache，根据需要来共享数据。但是对于大多数的workload，由于snoop request带来的tag查找大多数都miss了，这也就意味着由于snoop引起的带宽和功耗都是不必要的。

## Snoop Filter

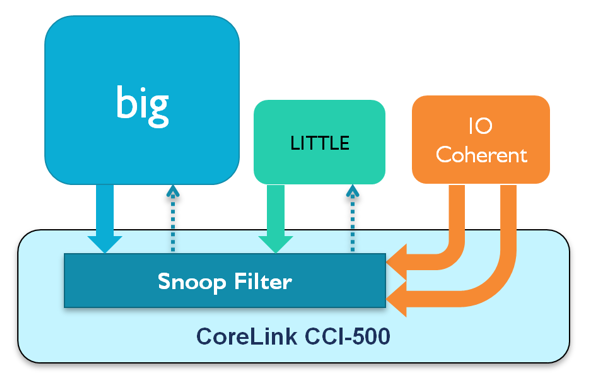


图13 Snoop Filter

通过把Snoop Filter集成到interconnect中，我们可以维护cache的目录，就不需要再去广播snoop。

Snoop Filter的基本原理如下：

* 所有cache的tag会存储到interconnect的目录（Snoop Filter）中。
* 所有Shared的访问都会在Snoop Filter中查找，有两种可能的结果：
  + HIT：说明数据在某个cluster中，并指示对应包含数据的cluster。
  + MISS：去外部memory取数。

为了支持Snoop Filter，caching Manager必须提供足够的信息使得Snoop Filter能够追踪local cache中的allocation和eviction。

对于eviction，主要包括：

* Local cache line evictions
* WriteBack of cache lines to memory
* Snoop transaction that cause an eviction

Cache line是否会被allocate决定于transaction。Snoop Filter可以通过观察Manager发起的transaction来确定cache line期望的allocation state。如果实际的allocation与Snoop Filter期望的不一致，需要执行相应的Evict操作来确保Snoop Filter正确地追踪allocated cache line。

|  |  |
| --- | --- |
| **Transaction** | **Expected cache line allocation** |
| ReadOnce | Allocation does not change |
| ReadClean | Allocated |
| ReadNotSharedDirty | Allocated |
| ReadShared | Allocated |
| ReadUnique | Allocated |
| CleanUnique | Allocated |
| MakeUnique | Allocated |
| CleanShared | Allocation does not change |
| CleanInvalid | Evicted |
| MakeInvalid | Evicted |
| WriteUnique | Allocation does not change |
| WriteLineUnique | Allocation does not change |
| WriteClean | Allocation does not change |
| WriteBack | Evicted |
| WriteEvict | Evicted |
| Evict | Evicted |

## Snoop Control Unit

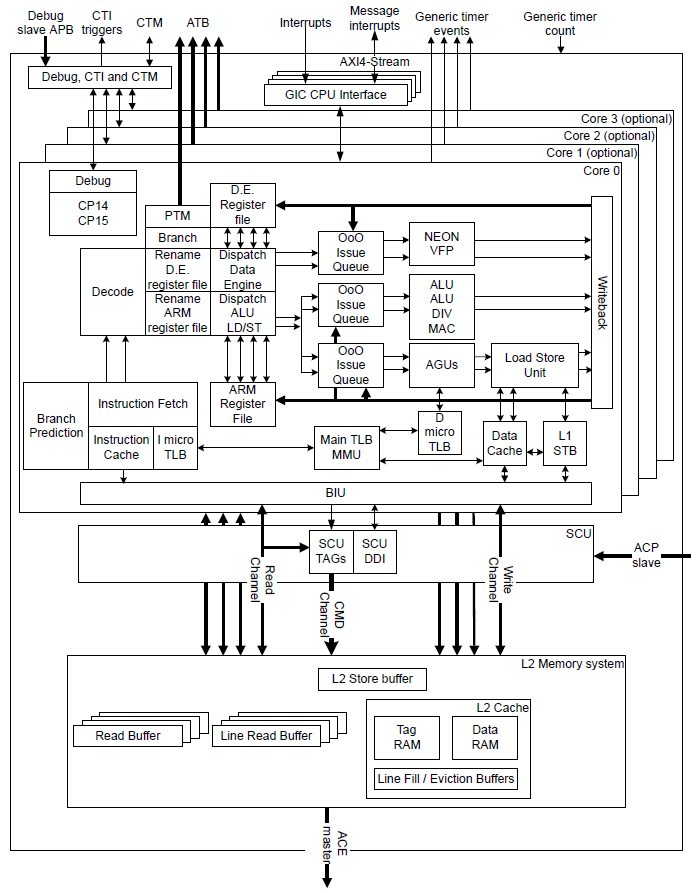


图14 Cortex-A73 processor block diagram

# Comparison of ACE and TileLink

* ACE协议可以支持cluster间的一致性。
* ACE可以支持MOESI一致性协议，TileLink支持MESI协议。
* ACE支持核间forward（ReadUnique，ReadShared）。
* ACE可以支持critical word first（WRAP burst）。
* ACE可以做到更细粒度的控制（PassDirty，IsShared）。
* ACE支持DVM操作。
* ACE支持Exclusive access（ARLOCK）。
* TileLink不支持read interleaving。