MESI

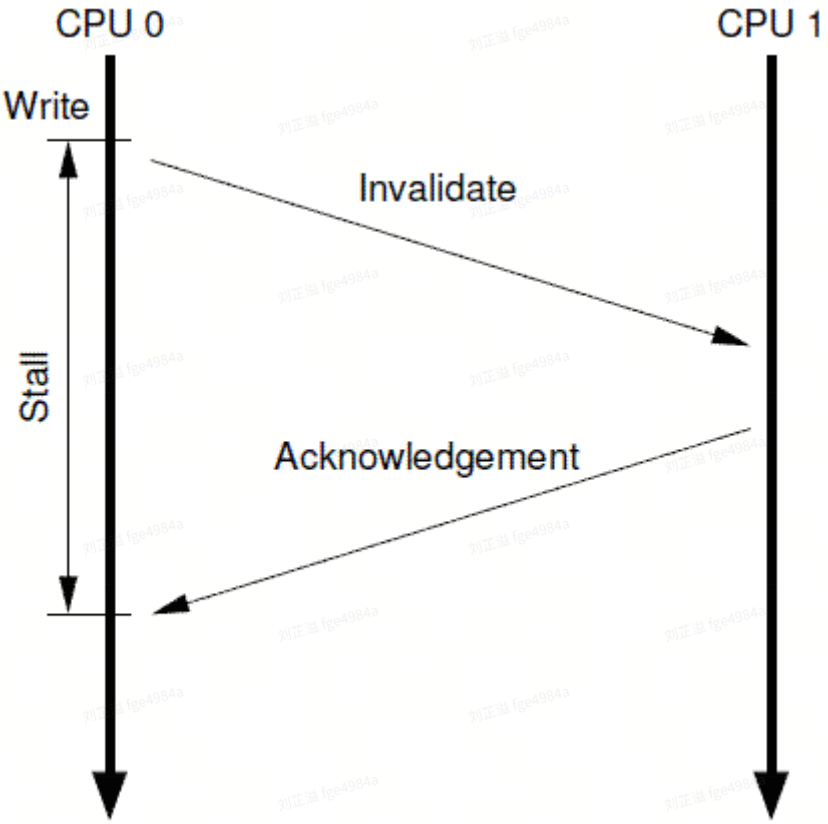
Why we need memory barrier?

MESI protocol state

状态	描述	监听任务
M 修改 (Modified)	该Cache line有效，数据被修改了，和内存中的数据不一致，数据只存在于本Cache中。	缓存行必须时刻监听所有试图读该缓存行相对就主存的操作，这种操作必须在缓存将该缓存行写回主存并将状态变成S（共享）状态之前被延迟执行。
E 独享、互斥 (Exclusive)	该Cache line有效，数据和内存中的数据一致，数据只存在于本Cache中。	缓存行也必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成S（共享）状态。
S 共享 (Shared)	该Cache line有效，数据和内存中的数据一致，数据存在于很多Cache中。	缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
I 无效 (Invalid)	该Cache line无效。	无

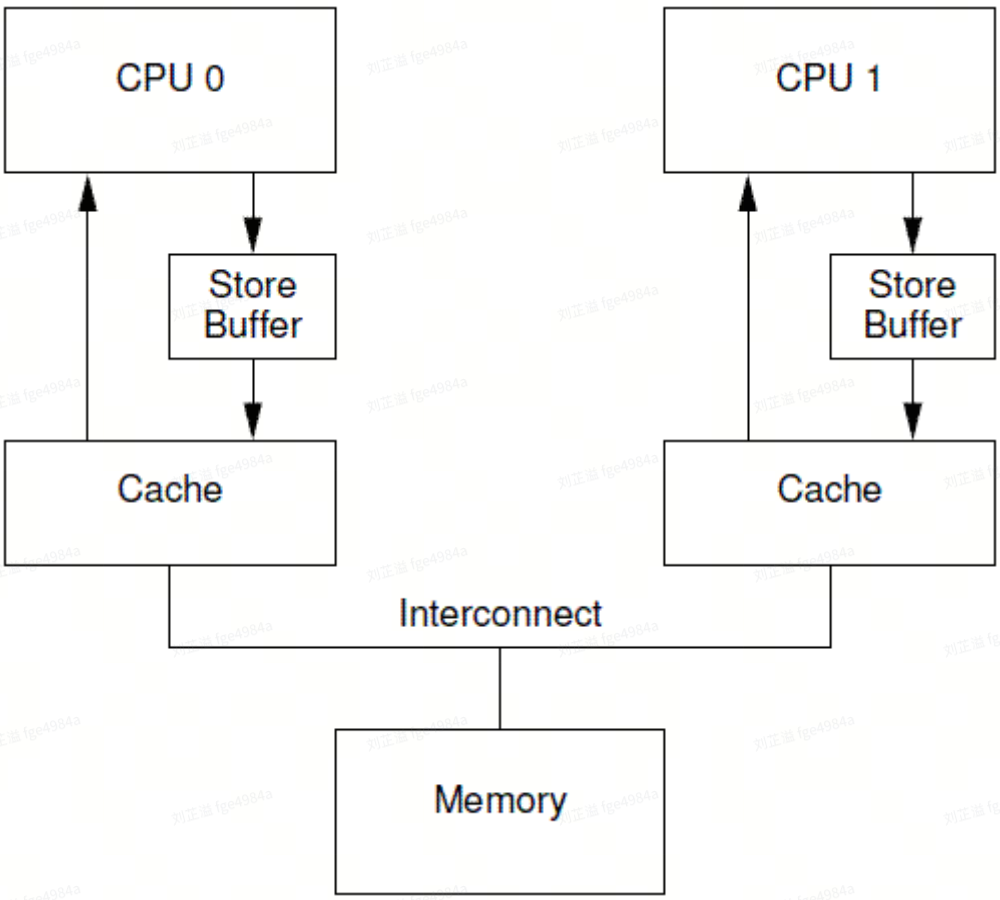
Stores Result in Unnecessary Stalls

- 针对某些特定地址的数据（在一个cacheline中）重复的进行读写，仅仅使用 `cache` 可以获得很好的性能，不过，对于第一次写，其性能非常差。
- CPU 0发起一次对某个地址的写操作，但是 `local cache` 没有数据，该数据在CPU 1的 `local cache` 中，因此，为了完成写操作，CPU 0发出invalidate的命令，invalidate其他CPU的cache数据。只有完成了这些总线上的transaction之后，CPU 0才能正在发起写的操作，这是一个漫长的等待过程。



store buffer

- 有一种可以阻止cpu进入无聊等待状态的方法就是在CPU和cache之间增加 `store buffer` 这个 `HW block`



- 但是执行下列代码时，可能发生失败。

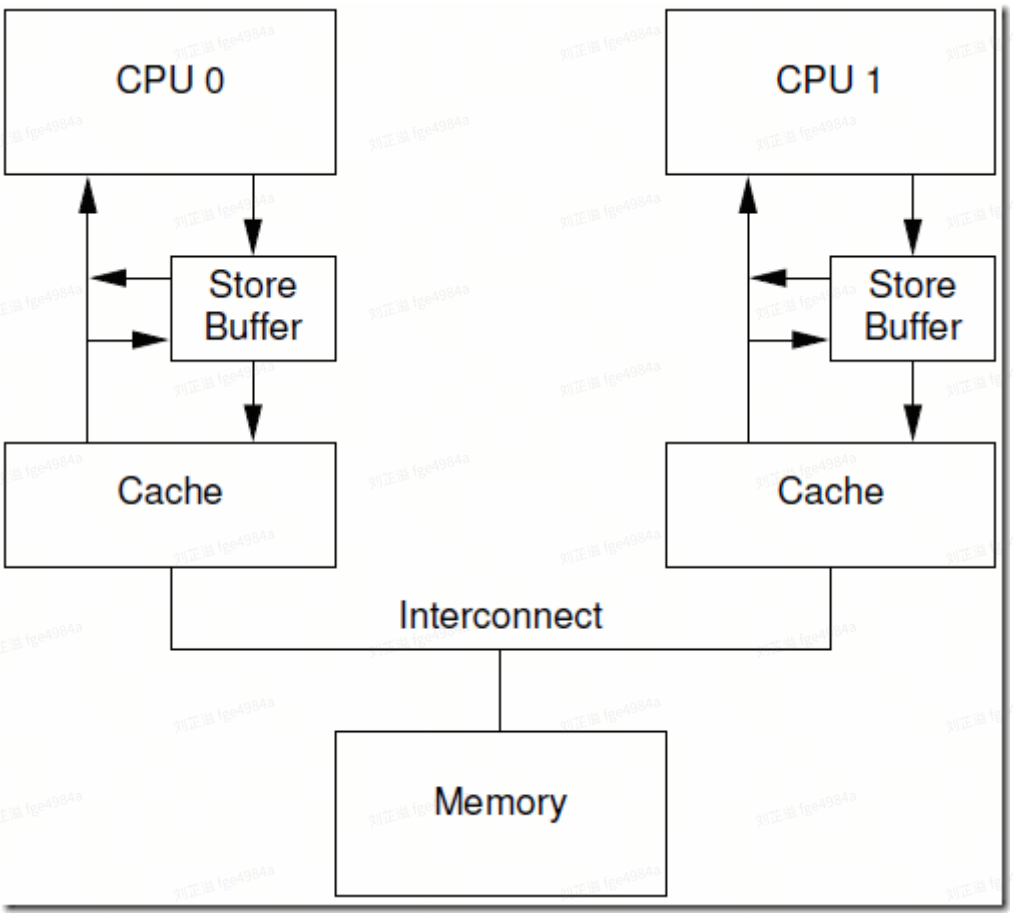
- 初始情况假定a和b都为0，且a在CPU1的cache line中，而b在CPU0的cache line中。

```
1   a = 1;
2   b = a + 1;
3   assert(b == 2);
```

1. CPU0 Cache miss a，发送read invalidate
2. CPU0 store a to store buffer
3. CPU1 erase cacheline(a) , send read response and invalidate acknowledge
4. CPU0 execute b=a+1
5. CPU0 load a from cacheline(from CPU1), now a = 0
6. CPU0 calculate b = 0 + 1 = 1, assert failed

- 导致这个问题的根本原因是我们有两个a值，一个在 `cacheline` 中，一个在 `store buffer` 中。

store forwarding



- 初始情况假定a和b都为0，且a在CPU1的cache line中，而b在CPU0的cache line中。CPU0执行foo()，CPU1执行bar()

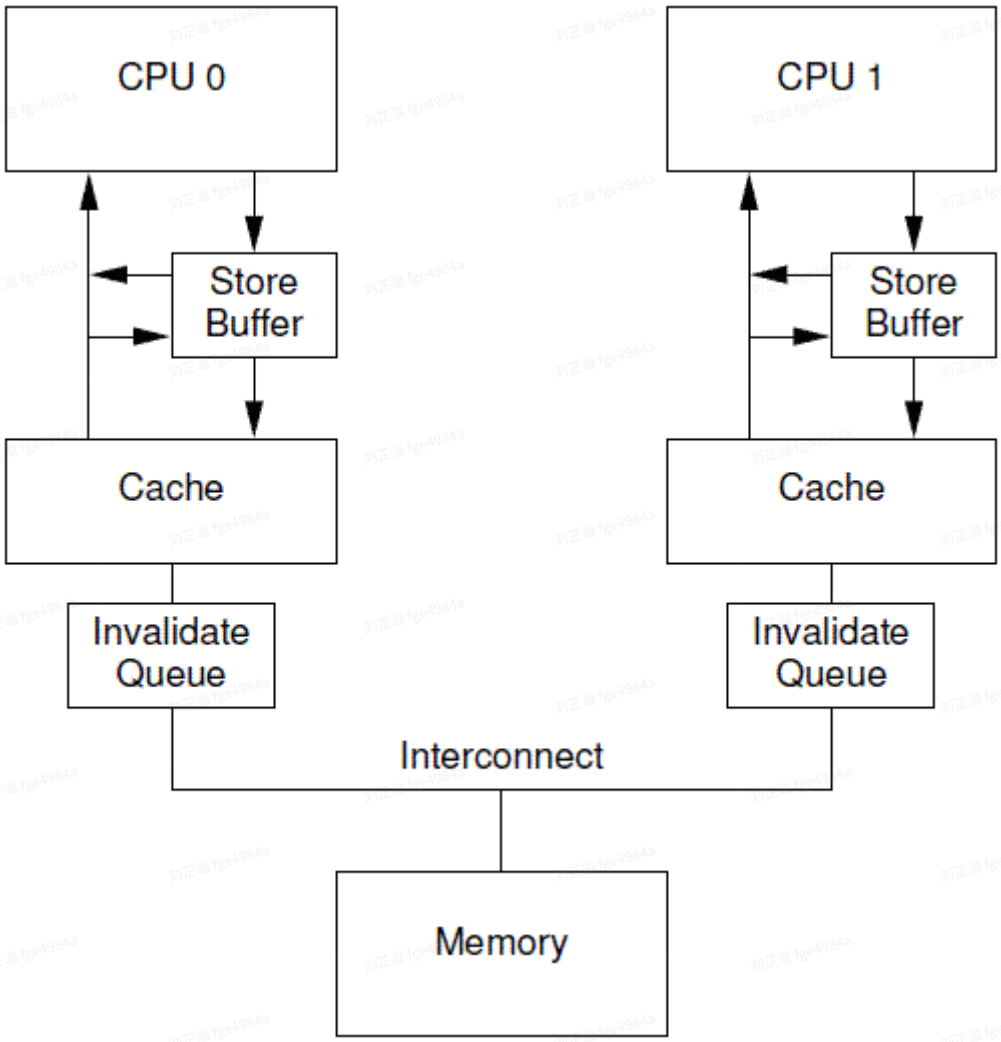
```
void foo()
 {
    a = 1;
    b = 1;
}
void bar()
{
while(b == 0) continue;
    assert(a == 1);
}
```

1. CPU0 store a=1 in store buffer
2. CPU1 can not find b, send read invalidate
3. CPU0 store b=1 in cacheline
4. CPU0 send read response
5. CPU1 get cacheline(b=1)
6. CPU1 read cacheline(a=0), assert failed
7. CPU1 send read response and invalidate acknowledge
8. CPU0 store a=1 in cache line

- 原因在于两个CPU对于写入的理解（要求）不一致，CPU0认为写入到store buffer就叫写入，而CPU1则认为写入到cache line才叫写入。
- 使用**内存屏障操作**：`smp_mb()`这个内存屏障的操作会在执行后续的store操作之前，首先flush store buffer（也就是将之前的值写入到cacheline中）。

Store Sequences Result in Unnecessary Stalls

- 不幸的是：每个cpu的store buffer不能实现的太大，其entry的数目不会太多。当cpu以中等的频率执行store操作的时候（假设所有的store操作导致了cache miss），store buffer会很快的被填满
- 增加`invalidate`消息的缓存`invalidate queues`
- 当然，如果本CPU想要针对某个cacheline向总线发送invalidate消息的时候，那么CPU必须首先去Invalidate Queue中看看是否有相关的cacheline，如果有，那么不能立刻发送，需要等到Invalidate Queue中的cacheline被处理完之后再发送。



- 我们假设a和b初值是0，并且a在CPU 0和CPU 1都有缓存的副本，即a变量对应的CPU0和CPU 1的cacheline都是shared状态。b处于exclusive或者modified状态，被CPU 0独占。我们假设CPU 0执行foo函数，CPU 1执行bar函数。

```
1   void foo()
2    {
3       a = 1;
4       smp_mb();
5       b = 1;
6   }
7   void bar()
8   {
9   while(b == 0) continue;
10      assert(a == 1);
11  }
```

1. CPU0 execute a=1, store a=1 to store buffer, send invalidate to other CPUS
2. CPU1 not find b, send read
3. CPU1 receive invalidate to invalidate queue, send acknowledge

4. CPU0 receive acknowledge, execute smp_mb(), cacheline(a=1)
5. CPU0 execute b=1, cacheline(b=1)
6. CPU0 receive read, send response and cacheline(b=1)
7. CPU1 get read response
8. CPU1 get b=1 from cacheline
9. CPU1 get cacheline from self, a=0
10. CPU1 process invalidate queue, erase cahceline(a=0)

- 加速Invalidation response导致foo函数中的memory barrier失效了；需要再次使用smp_mb()来让两个CPU可以通信
- CPU1在执行assert前，需要将invalidate queue中请求处理完毕，将a的cacheline移除

```
1   void foo()
2   {
3       a = 1;
4       smp_mb();
5       b = 1;
6   }
7   void bar()
8   {
9       while(b == 0) continue;
10      smp_mb();
11      assert(a == 1);
12  }
```

Read and Write Memory Barriers

- 在我们上面的例子中，`memory barrier`指令对`store buffer`和`invalidate queue`都进行了标注，不过，在实际的代码片段中，foo函数不需要`mark invalidate queue`，bar函数不需要`mark store buffer`
- 因此，许多CPU architecture提供了弱一点的memory barrier指令只mark其中之一。
  - 如果只mark invalidate queue，那么这种memory barrier被称为`read memory barrier`
  - 相应的，`write memory barrier`只mark store buffer。
  - 一个全功能的`memory barrier`会同时mark store buffer和invalidate queue。
- 对于read memory barrier指令，它只是约束执行CPU上的load操作的顺序，具体的效果就是CPU一定是完成read memory barrier之前的load操作之后，才开始执行read memory barrier之后的load操作

- write memory barrier指令，它只是约束执行CPU上的store操作的顺序，具体的效果就是CPU一定是完成write memory barrier之前的store操作之后，才开始执行write memory barrier之后的store操作

```
1  void foo()
2  {
3      a = 1;
4      smp_wmb();
5      b = 1;
6  }
7  void bar()
8  {
9      while(b == 0) continue;
10     smp_rmb();
11     assert(a == 1);
12 }
```