第4季

缓存一致性

保密文件，仅供奔跑吧Linux社区视频课程
《RISC-V体系结构编程与实践》的学员阅读之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

本节课主要内容

- 本章主要内容
  - 缓存一致性介绍
  - 一致性解决方案
  - MESI协议
  - 系统缓存一致性方案介绍
  - 缓存一致性案例分析

技术手册：

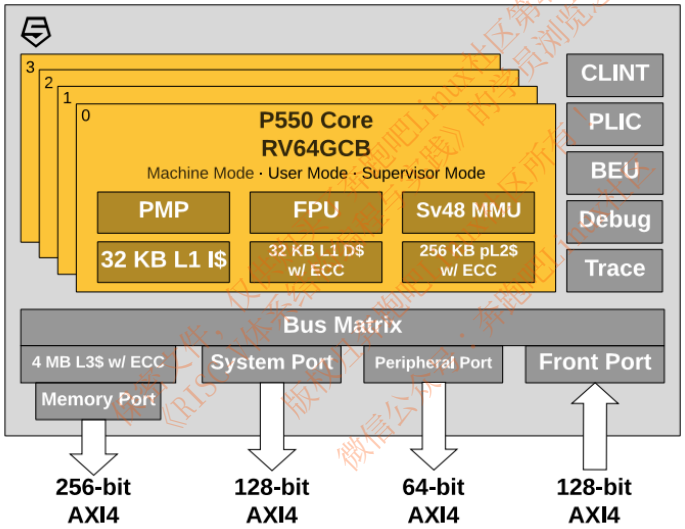
1. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203
2. SiFive U74-MC Core Complex Manual, 21G2.01.00

本节课主要讲解书上第12章内容

保密协议：仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

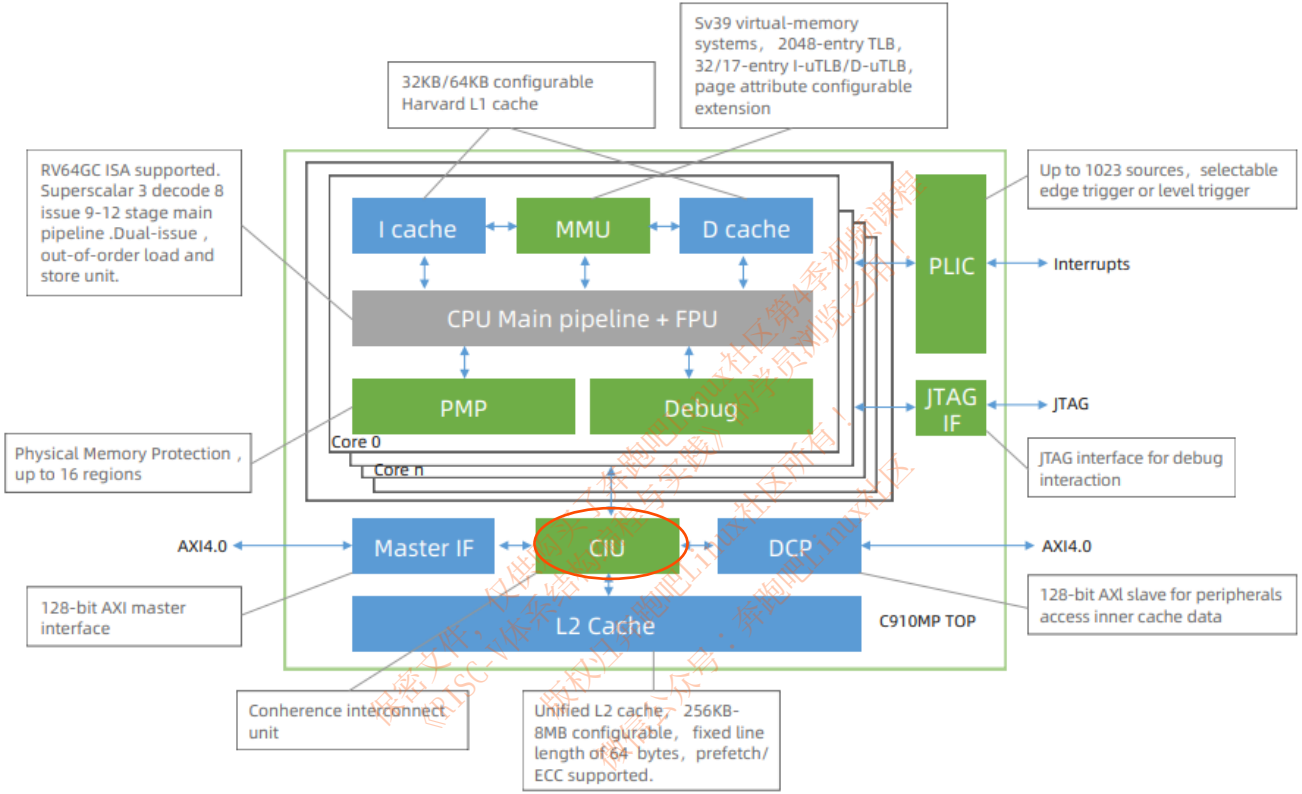
RISC-V架构与缓存一致性

- RISC-V架构手册里没有约定缓存一致性采用何种方案。
- 目前大部分RISC-V芯片采用Arm的AMBA总线以及缓存一致性的IP，如SCU，ACE总线，CCI等。



SiFive的P550多核处理器

《RISC-V体系结构与编程实践》的学员浏览之用！
微信公众号：奔跑吧Linux社区



平头哥C910多核处理器框图，内部使用CIU管理多核的缓存一致性（类似SCU）

国内某款多核RISC-V处理器框图，多核缓存一致性采用Arm的SCU

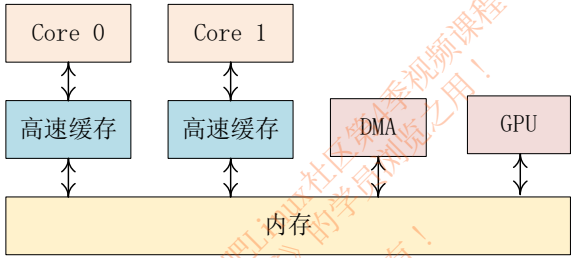
保密文件，仅供购买《RISC-V体系结构编程与实践》的学员浏览之用！
版权所有，奔跑吧Linux社区
微信公众号：奔跑吧Linux社区

Part1： cache一致性介绍

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程
《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区

为什么要cache一致性（cache coherency）？

- 系统中各级cache都有不同的数据备份，例如每个CPU核心都有L1 cache。



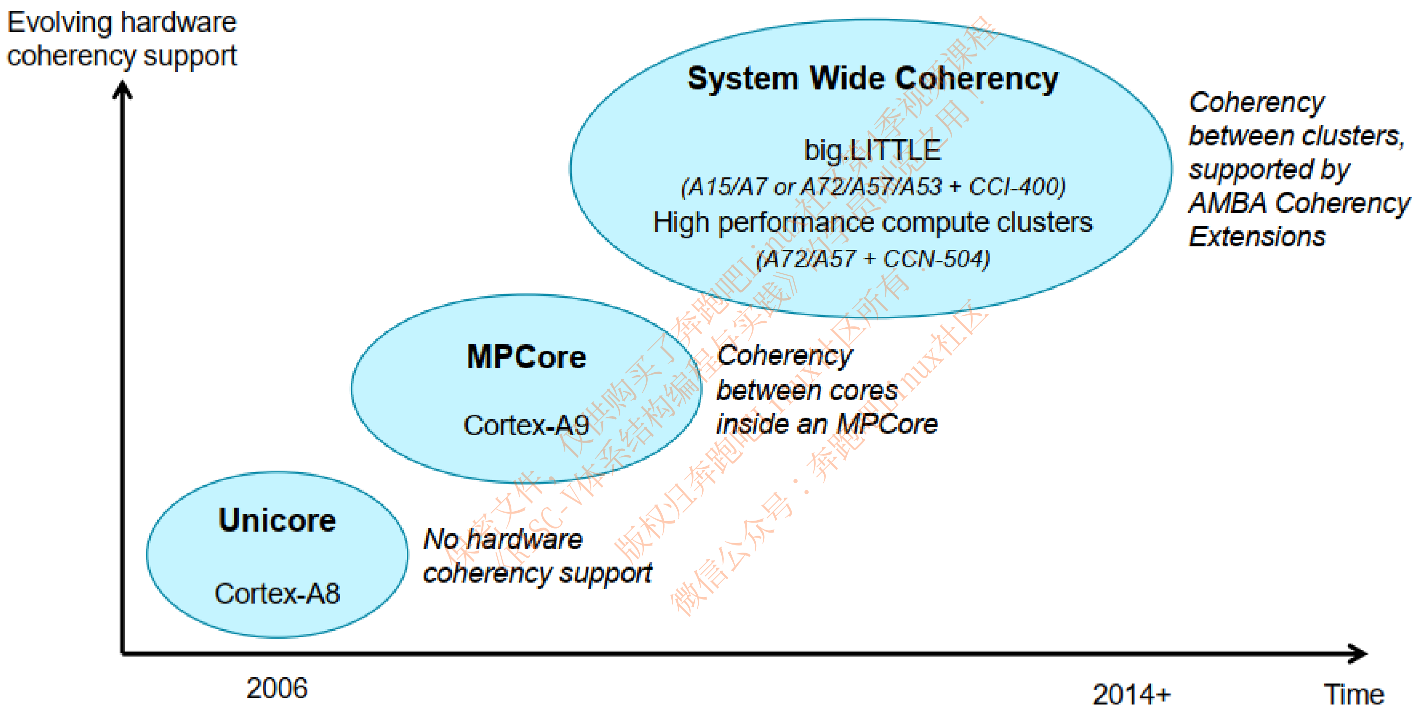
- cache一致性关注的是同一个数据在多个高速缓存和内存中的一致性问题，解决高速缓存一致性的方法主要是总线监听协议，例如MESI协议等。

- 需要关注cache一致性的例子：
  - ✓ 驱动中使用DMA（数据cache和内存不一致）
  - ✓ Self-modifying code（数据cache的数据可能比指令cache新）
  - ✓ 修改了页表（TLB里保存的数据可能过时）

- 大多时候硬件来维护cache一致性，不过有些场景下需要软件来手动保证cache一致性

奔跑吧Linux内核旗舰篇视频课程

ARM的cache一致性的演进



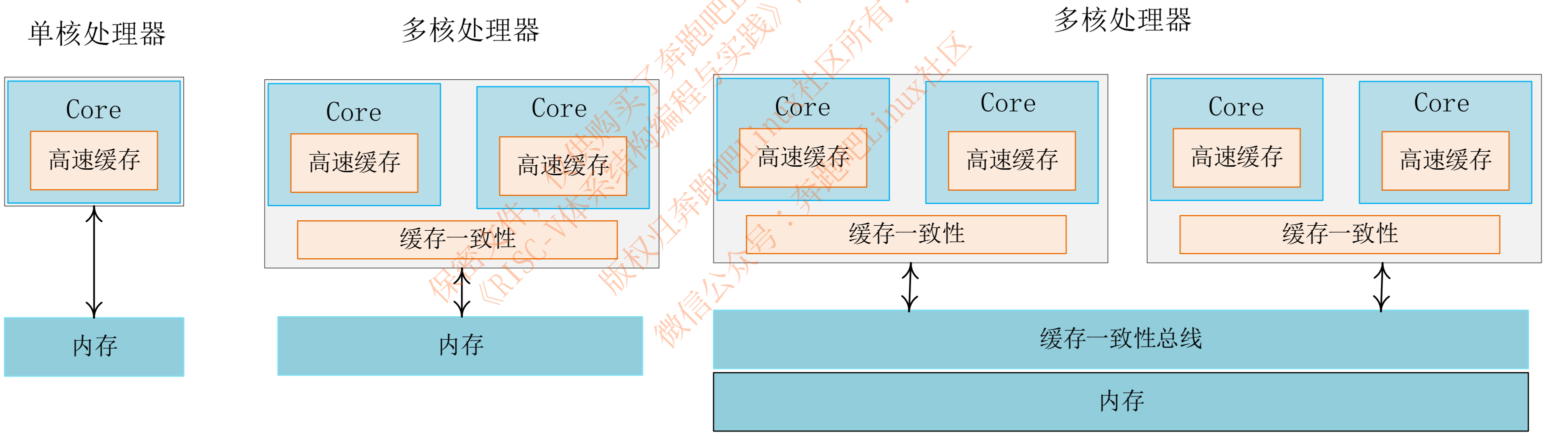
保密文件，仅供购买了奔跑吧Linux社区《RISC-V体系结构编程与实践》奔视频课程
版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区

单核处理器（Cortex-A8）

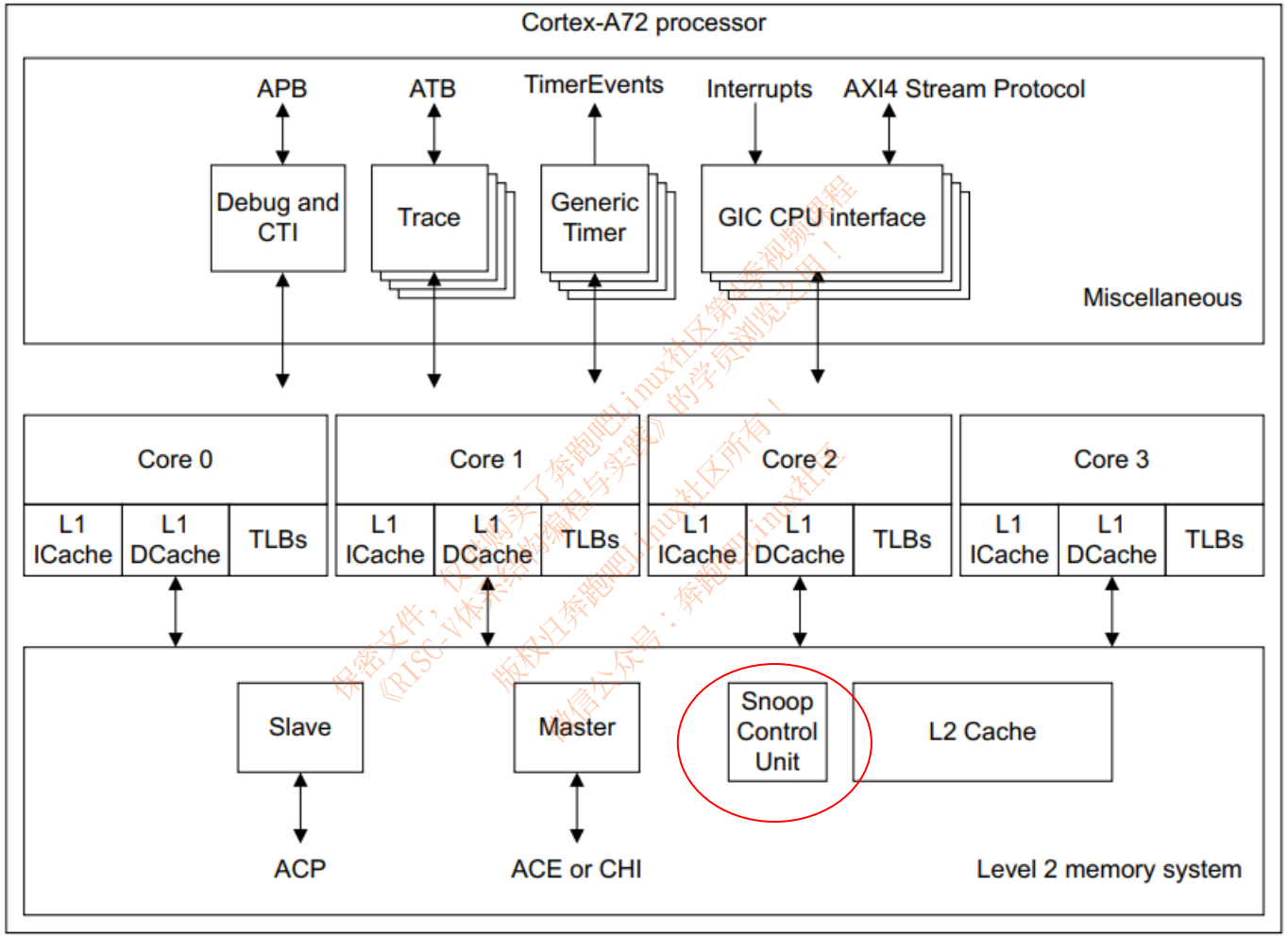
- 单核，没有cache一致性问题
- Cache管理指令仅仅作用于单核

多核处理器（Cortex-A9 MP以及之后的处理器）

- 硬件上支持cache一致性
- Cache管理指令会广播到其他CPU核心



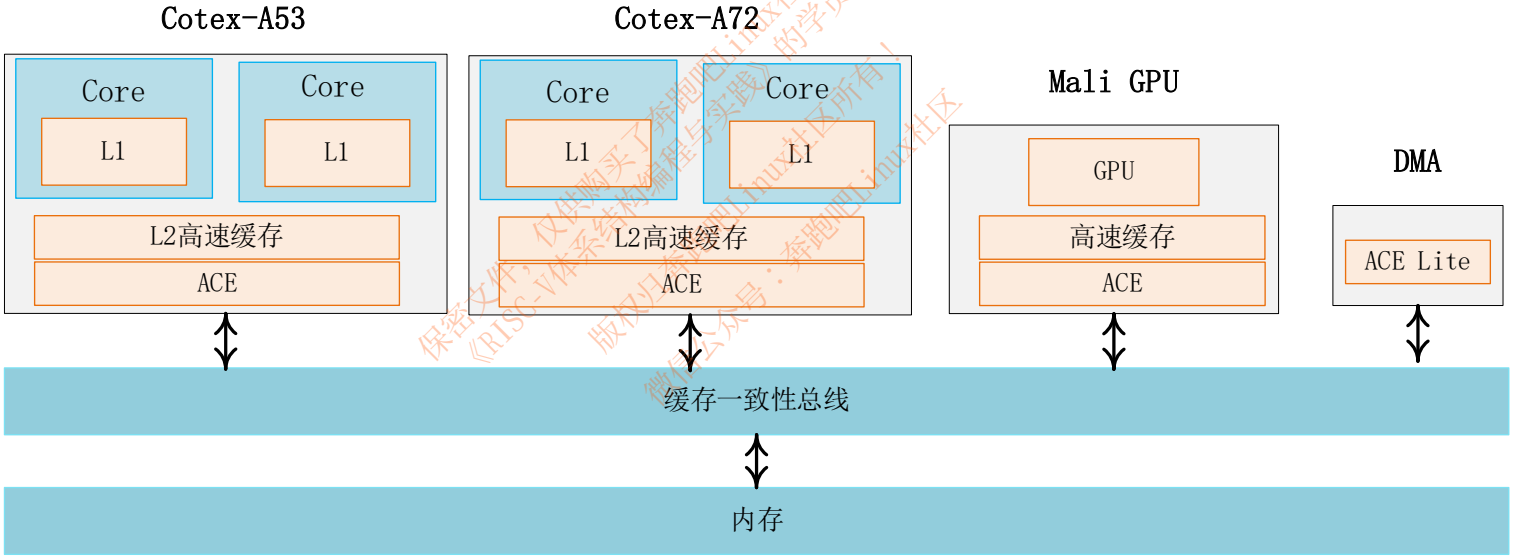
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



保密文件，仅供购买了奔跑吧Linux社区第1季视频课程《RISC-V体系结构编程与实现》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

系统级别的cache一致性

- 系统cache一致性需要cache一致性内部总线（cache coherent interconnect）
  - ✓ AMBA 4 协议有ACE (AXI Coherency Extensions)
  - ✓ AMBA 5协议有CHI



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

Cache一致性的解决方案

1. 关闭cache。
   - ✓ 优点：简单
   - ✓ 缺点：性能低下，功耗增加

2. 软件维护cache一致性。
   - ➢ 优点：硬件RTL实现简单
   - ➢ 缺点：
     - ✓ 软件复杂度增加。软件需要手动clean/flush cache或者invalidate cache
     - ✓ 增加调试难度
     - ✓ 降低性能，和增加功耗

3. 硬件维护cache一致性。

   MESI协议来维护多核cache一致性。ACE接口来实现系统级别的cache一致性
   - ➢ 优点：对软件透明
   - ➢ 缺点：增加了硬件RTL实现难度和复杂度

本文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实战》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

Part2：多核之间的cache一致性

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实战》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

cache一致性协议

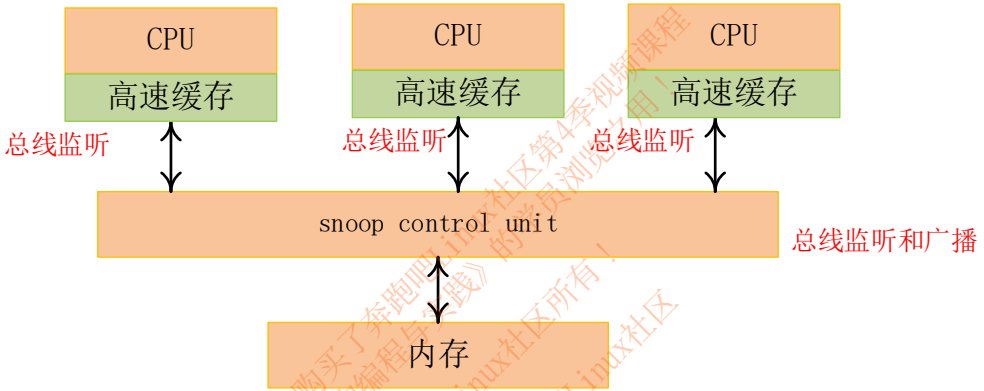
- 多核CPU产生cache一致性的原因：同一个内存数据在多个CPU核心的L1 cache中存在多个不同的副本，导致数据不一致。

- 维护cache一致性的关键是跟踪每一个cache line的状态，并根据处理器的读写操作和总线上的相应传输来更新cache line在不同CPU内核上的cache的状态，从而维护cache一致性。

- Cache一致性协议：
  - ✓ 监听协议（snooping protocol），每个高速缓存都要被监听或者监听其他高速缓存的总线活动；
  - ✓ 目录协议（directory protocol），全局统一管理高速缓存状态。

- MESI协议：
  - ✓ 1983年，James Goodman提出Write-Once总线监听协议，后来演变成目前最流行的MESI协议。
  - ✓ 所有的总线传输事务对于系统内所有的其他单元是可见的，因为总线是一个基于广播通信的介质，因而可以由每个处理器的高速缓存来进行监听。

《RISC-V体系结构编程与实践》的学员测试使用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



Snoop control unit单元实现总线监听和广播

每个CPU的L1 cache也实现了总线监听功能

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览使用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

MESI协议

- 每个cache line有4个状态
  - ✓ 修改（Modified）
  - ✓ 独占（Exclusive）
  - ✓ 共享（Shared）
  - ✓ 失效（Invalid）

表 1.2 MESI 协议定义

状　态	描　述
M	这行数据有效，数据被修改，和内存中的数据不一致，数据只存在本高速缓存中
E	这行数据有效，数据和内存中数据一致，数据只存在于本高速缓存中
S	这行数据有效，数据和内存中数据一致，多个高速缓存有这个数据副本
I	这行数据无效

保密文件，仅供参加了奔跑吧Linux社区《RISC-V体系结构编程与实践》的学员使用
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

- ✓ 修改和独占状态的cache line，数据都是独有的，不同点在于修改状态的数据是脏的，和内存不一致，而独占态的数据是干净的和内存一致。脏的cache line会被回写到内存，其后的状态变成共享态。
- ✓ 共享状态的cache line，数据和其他cache共享，只有干净的数据才能被多个cache共享。
- ✓ I的状态表示这个cache line无效。

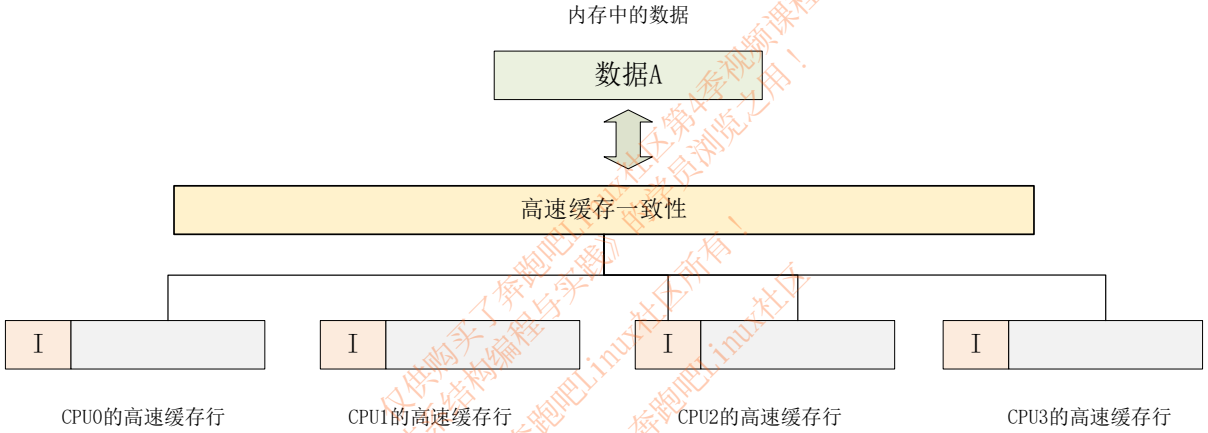
MESI操作

- 本地读写：本地CPU读写自己私有的cache line，这是一个私有的操作。
- 总线读写：读写远端CPU的cache line，CPU可以发送请求到总线上，所有的CPU都可以收到这个请求。

表 1.3　本地读写和总线操作

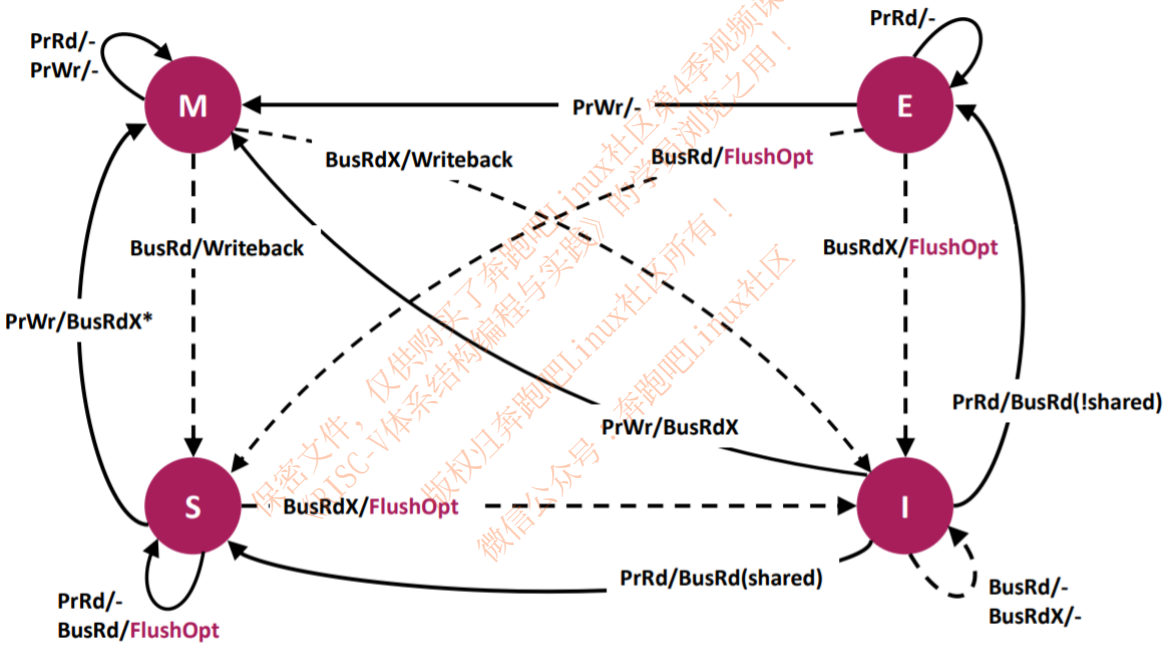
类　型	描　述
初始状态（Initial）	缓存行还没加载任何数据时，状态为 I
本地读（Local Read/PrRd）	表示本地 CPU 读取缓存行数据
本地写（Local Write/PrWr）	表示本地 CPU 更新缓存行数据
总线读（Bus Read, BusRd）	总线侦听到一个来自其他 CPU 的读缓存请求。收到信号的 CPU 先检查自己的高速缓存中是否有缓存该数据，然后广播应答信号
总线写（Bus Write/BusRdX）	总线侦听到一个来自其他 CPU 的写缓存请求。收到信号的 CPU 先检查自己的高速缓存中是否有缓存该数据，然后广播应答信号
总线更新（BusUpgr）	总线侦听到更新请求，请求其他 CPU 做一些额外事情。其他 CPU 收到请求后，若 CPU 上有缓存副本，则需要做额外的一些更新操作，比如无效本地的高速缓存行等
刷新（Flush）	总线侦听到刷新请求。收到请求的 CPU 把自己的高速缓存行的内容写回到主内存中
刷新到总线（FlushOpt）	收到该请求的 CPU 会把高速缓存行内容发送到总线上，这样发送请求的 CPU 就可以获取到这个高速缓存行的内容

保密文件，仅供奔跑吧Linux社区第4季视频课的学员浏览之用！

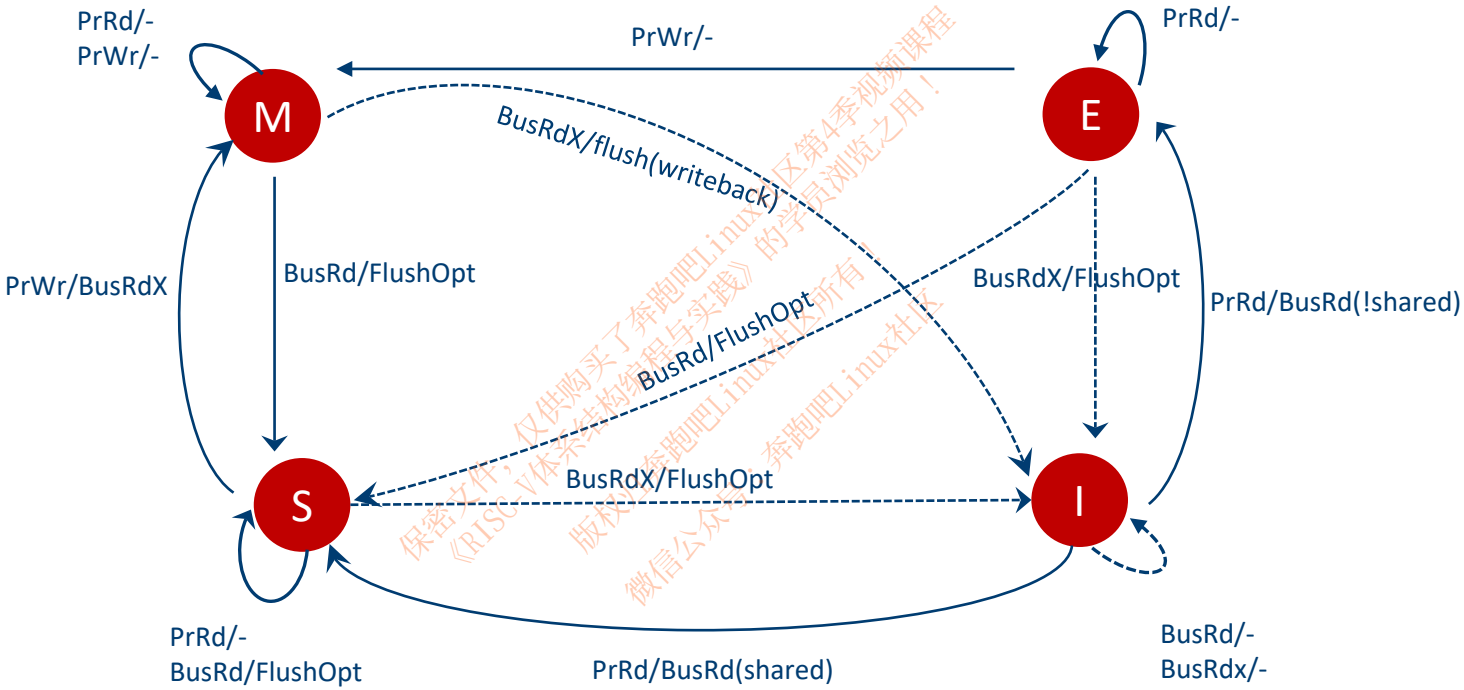


保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的会员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

MESI状态图

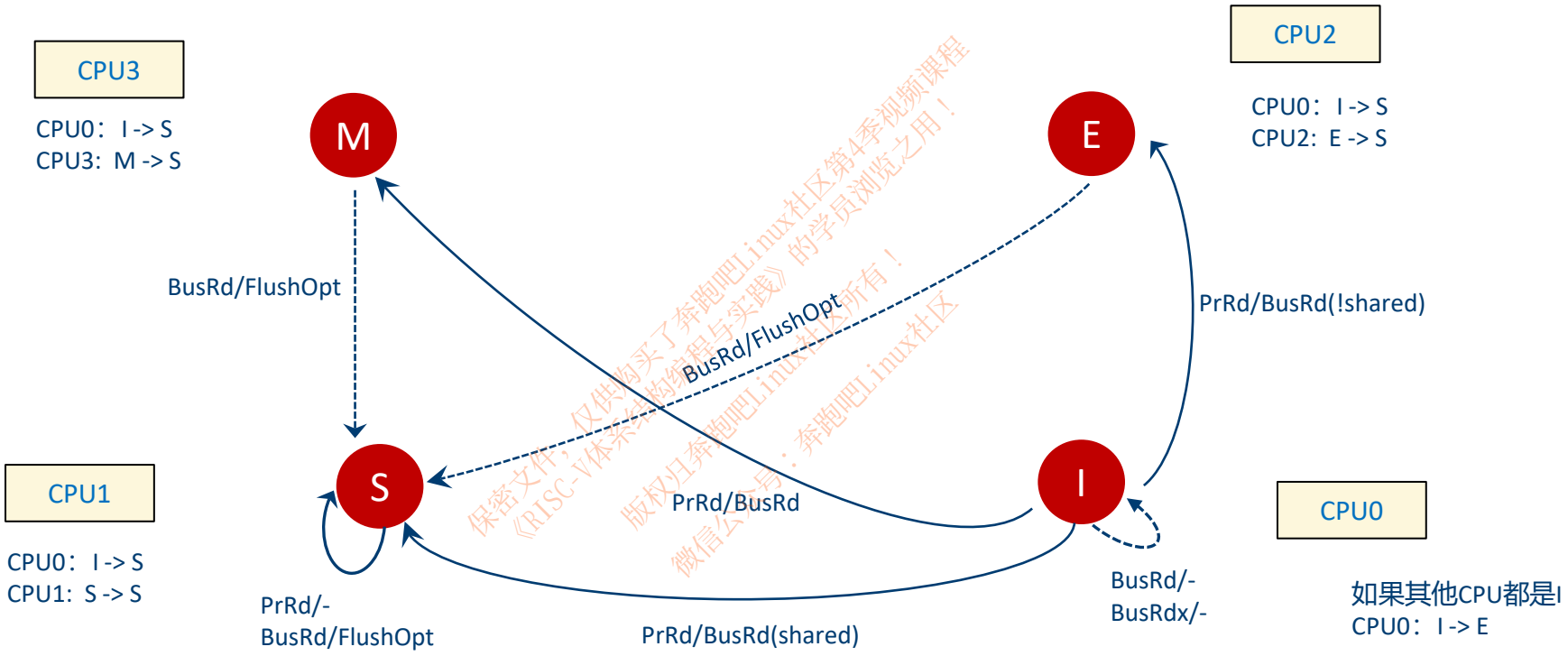


保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《基于RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权所有 奔跑吧Linux社区
微信公众号：奔跑吧Linux社区

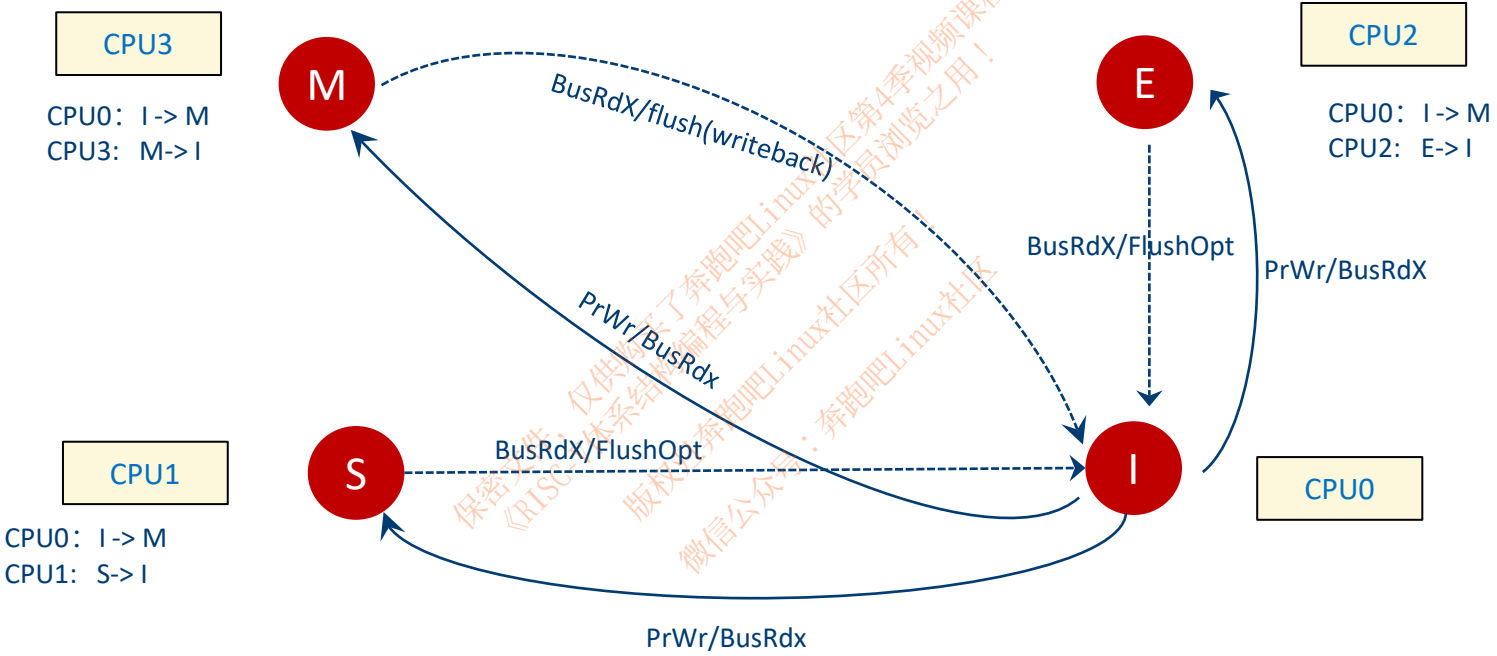
当初始化状态为I时，并且发出本地读操作



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

最终发出读操作的这个cache line状态变成E或者S

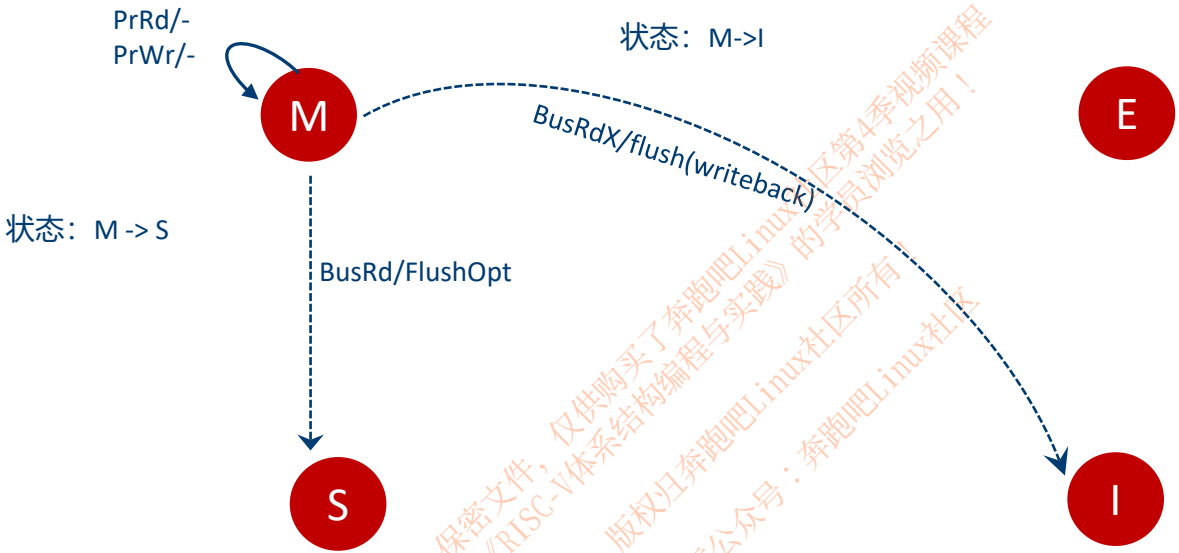
当初始化状态为I时，并且发出本地写操作



最终发出写操作的这个cache line状态变成M

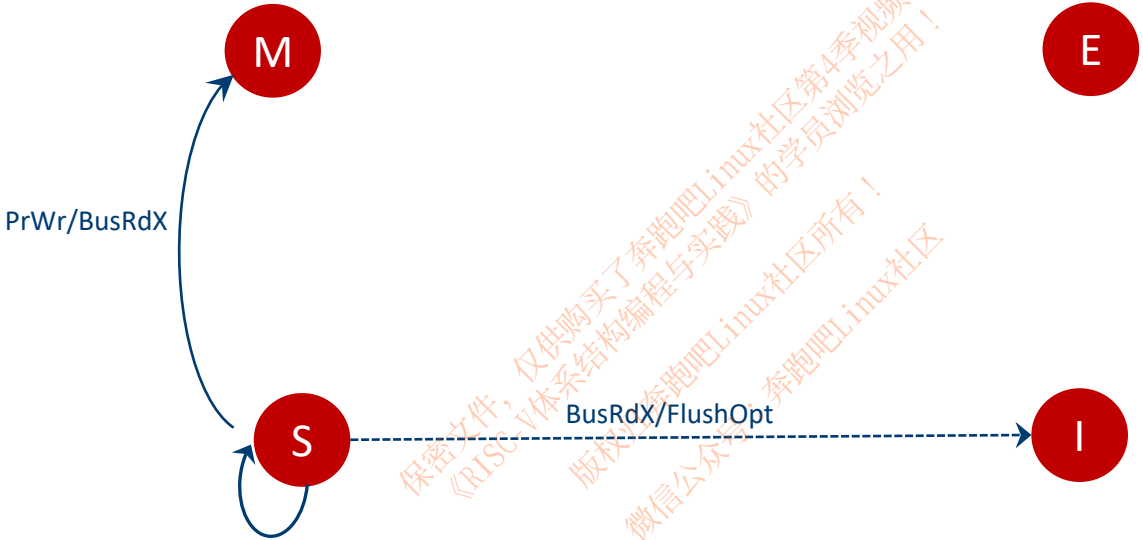
保密协议：仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权所有：奔跑吧Linux社区
微信公众号：奔跑吧Linux社区

当初始化状态为M时



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

当初始化状态为S时



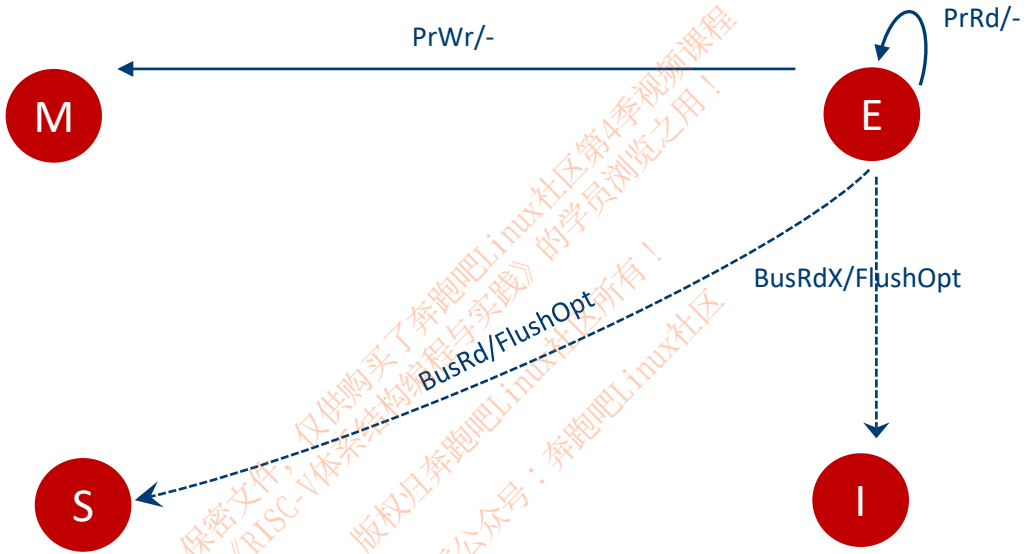
本地写的CPU： S -> M

收到BusRdX信号：S -> I

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权所有 奔跑吧Linux社区
微信公众号：奔跑吧Linux社区

当初始化状态为E时

本地写：E -> M



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

收到BusRd信号的CPU： E -> S

收到BusRdX信号的CPU： E -> I

MESI协议各个状态的转换关系

表 12.4 MESI 协议中各个状态的转换关系

当前状态	本地读	本地写	本地换出①	总线读②	总线写	总线更新
I	发出总线读信号。如果没有共享者，则状态 I 变成 E。如果有共享者，状态 I 变成 S	发出总线写信号，状态 I 变成 M	状态不变	状态不变，忽略总线上的信号	状态不变，忽略总线上的信号	状态不变，忽略总线上的信号
S	状态不变	发出总线更新信号，状态 S 变成 M	S 变成 I	状态不变，回应 FlushOpt 信号并且把内容发送到总线上	状态 S 变成 I	状态 S 变成 I
E	状态不变	E 变成 M	E 变成 I	回应 FlushOpt 信号并把内容发送到总线上，状态 E 变成 S	状态 E 变成 I	错误状态
M	状态不变	状态不变	写回数据到内存，M 变成 I	回应 FlushOpt 信号并把内容发送到总线上和内存中，状态 M 变成 S	回应 FlushOpt 信号并把内容发送到总线上和内存中，状态 M 变成 I	错误状态

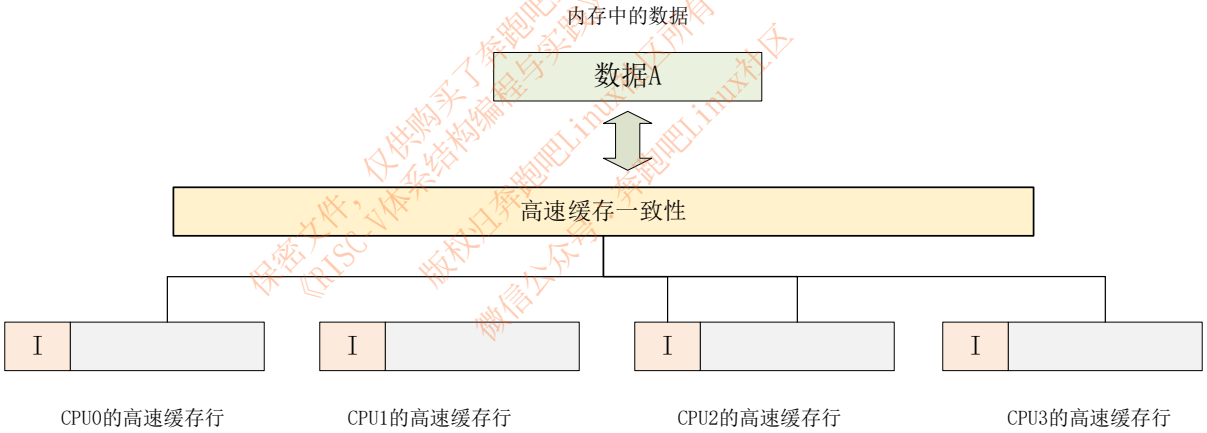
① 指的是本地换出（local eviction）高速缓存行。

② 这里指在当前 MESI 状态下的高速缓存行收到总线读信号。

体验文件，仅供购买了奔跑吧Linux内核第4季视频课程《RISC-V体系结构编程与实战》的学员浏览之用！
微信公众号：奔跑吧Linux社区

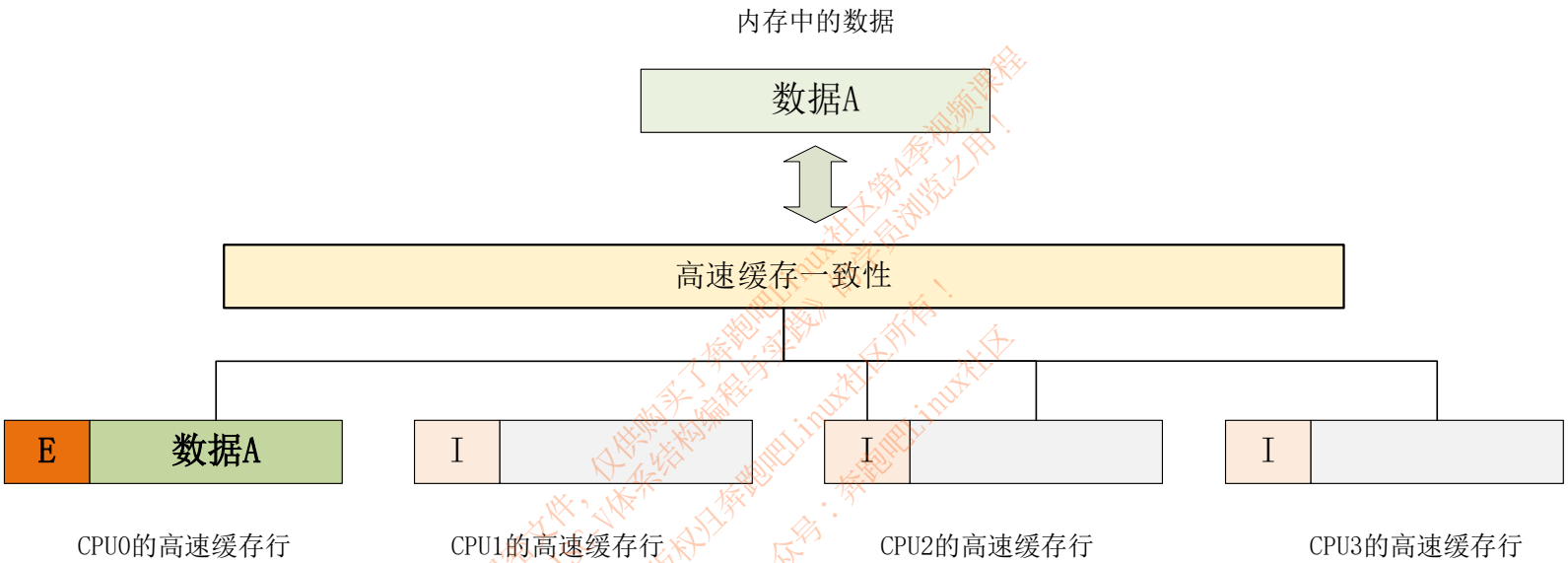
MESI协议分析的一个例子

- 假设系统中有4个CPU，每个CPU都有各自的一级缓存，它们都想访问相同地址的数据A，大小为64字节。
  - ✓ T0时刻：4个CPU的L1 cache都没有缓存数据A，cache line的状态为I (无效的)
  - ✓ T1时刻：CPU0率先发起访问数据A的操作
  - ✓ T2时刻：CPU1也发起读数据操作
  - ✓ T3时刻：CPU2的程序想修改数据A中的数据
- 请分析上述过程中, MESI状态的变化。



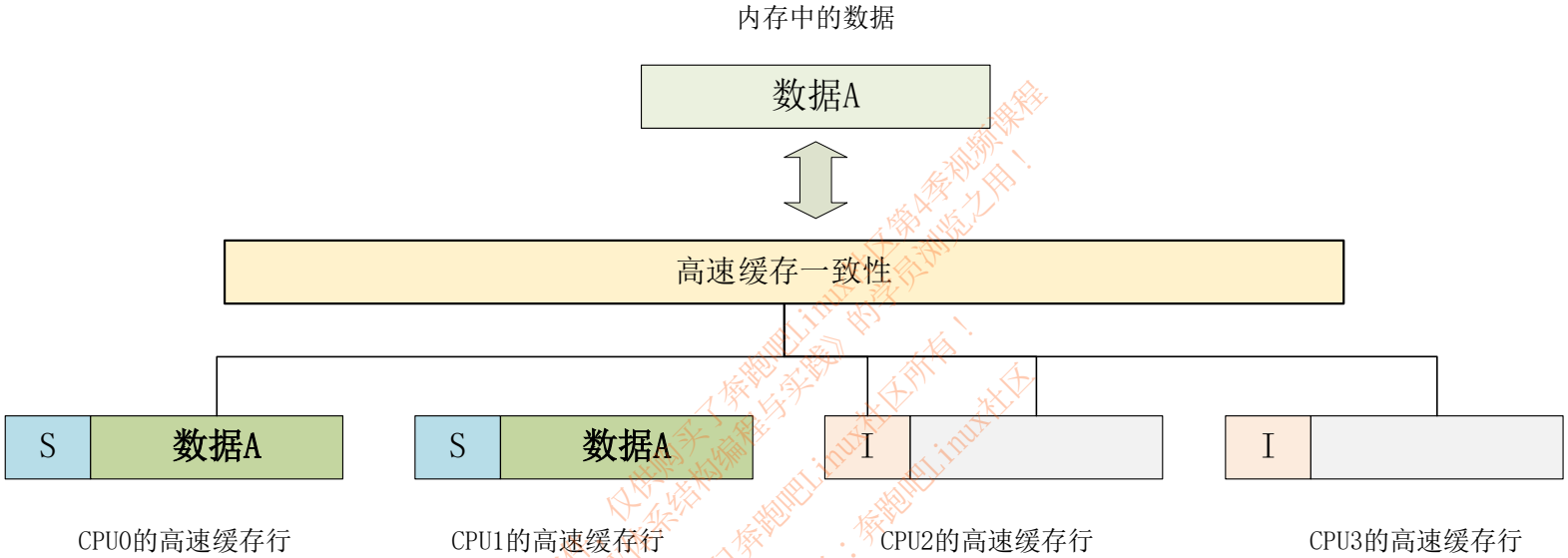
T0时刻4个CPU的cache line的状态

仅供购买了奔跑吧Linux社区第4季视频课《RISC-V体系结构编程与实践》的学员浏览之用！
保密文件，版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



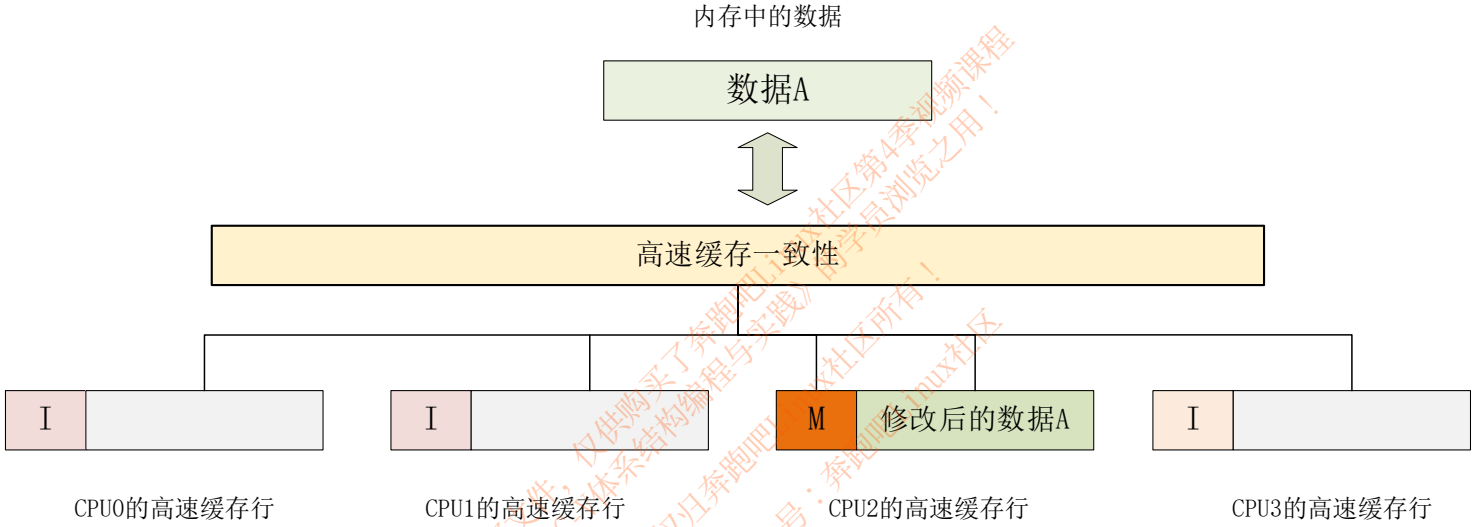
T1时刻4个CPU的cache line的状态

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T2时刻4个CPU的cache line的状态

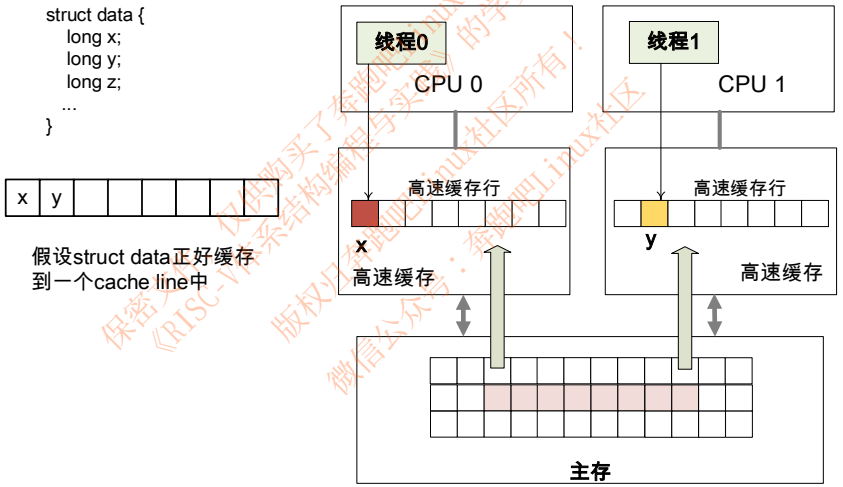


T3时刻4个CPU的cache line的状态

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

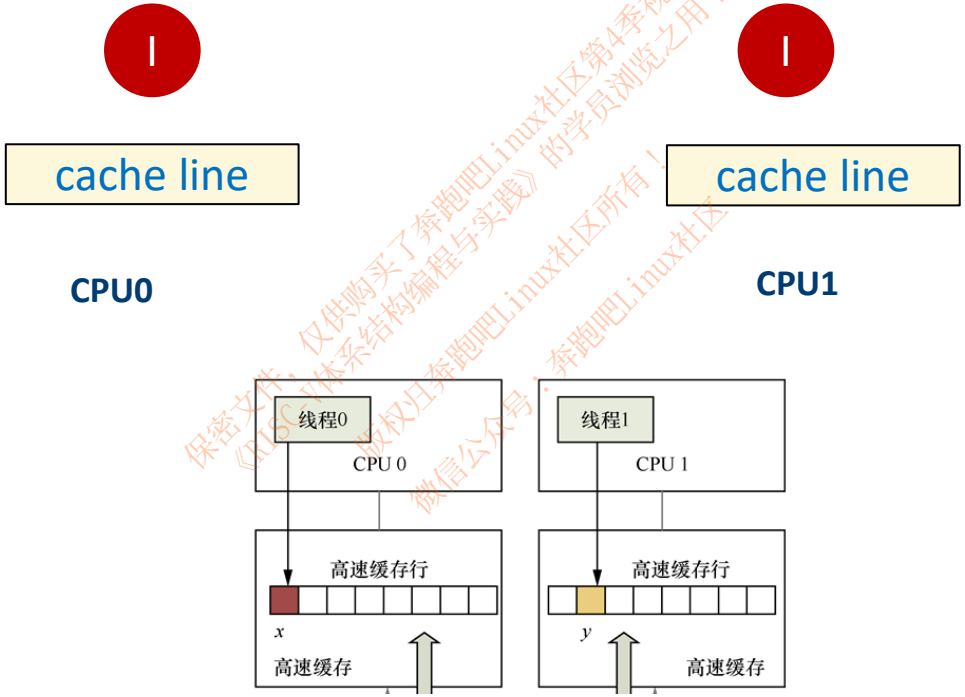
高速缓存伪共享（False Sharing）

- 伪共享：如果多个处理器同时访问一个缓存行中不同的数据时，带来了性能上的问题
- 举个栗子：假设CPU0上的线程0想访问和更新struct data数据结构中的x成员，同理CPU1上的线程1想访问和更新struct data数据结构中的y成员，其中x和y成员都被缓存到同一个缓存行里。



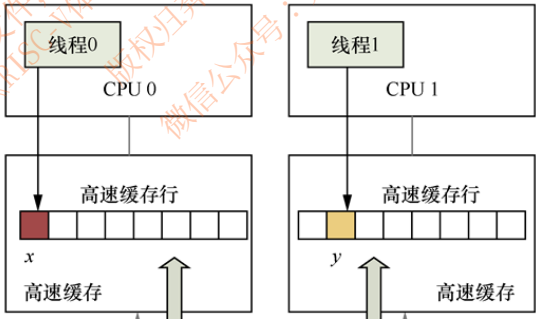
保密公告：仅供购买了奔跑吧Linux社区第4季视频课的学员浏览之用！
版权归奔跑吧Linux社区所有！
《RISC-V体系结构编程与实战》
微信公众号：奔跑吧Linux社区

T0时刻：cpu0和CPU1的cache line都是I



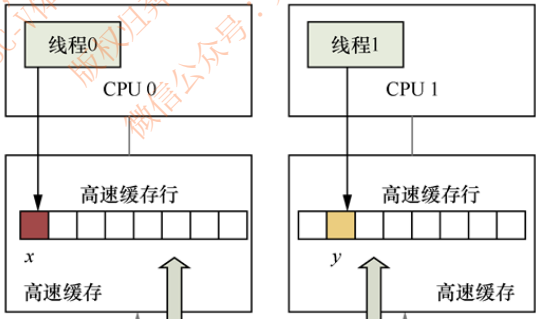
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T1时刻：CPU0第一次访问x成员



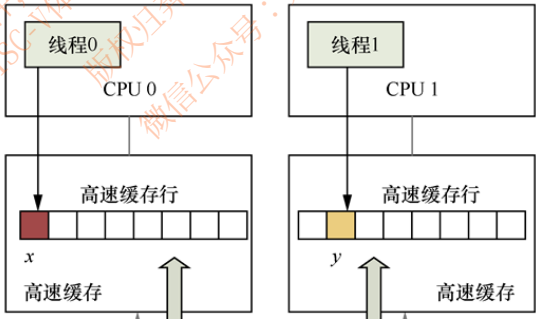
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《ARM64体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T2时刻：CPU1第一次访问y成员



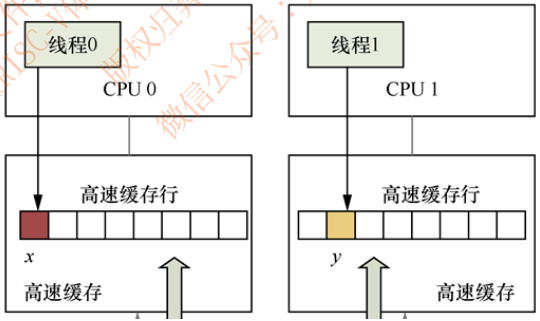
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览与学习！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T3时刻：CPU0写x成员



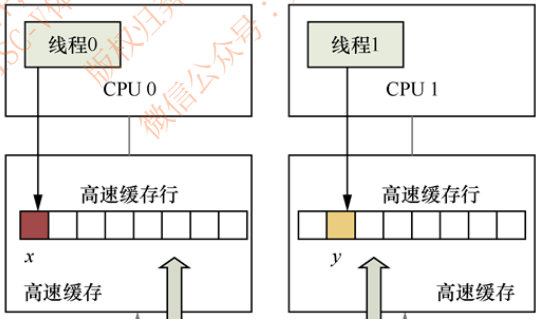
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览与学习！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T4时刻：CPU1写y成员



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览与学习！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

T5时刻：CPU0写X成员



保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览与学习！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

高速缓存伪共享（False Sharing）

- 高速缓存伪共享的解决办法就是让多线程操作的数据处在不同的高速缓存行，通常可以采用高速缓存行填充（padding）技术或者高速缓存行对齐（align）技术，即让数据结构按照高速缓存行对齐，并且尽可能填充满一个高速缓存行大小。

- 下面的代码定义一个counter_s数据结构，它的起始地址按照高速缓存行的大小对齐，数据结构的成员通过pad[4]来填充。这样， counter_s的大小正好 是一个cache line的大小，64个字节，而且它的起始地址也是也cache line对齐

```
typedef struct counter_s
{
    uint64_t packets;
    uint64_t bytes;
    uint64_t failed_packets;
    uint64_t failed_bytes;
    uint64_t pad[4];
}counter_t __attribute__((__aligned__((64)));
```

微信公众号：奔跑吧Linux社区

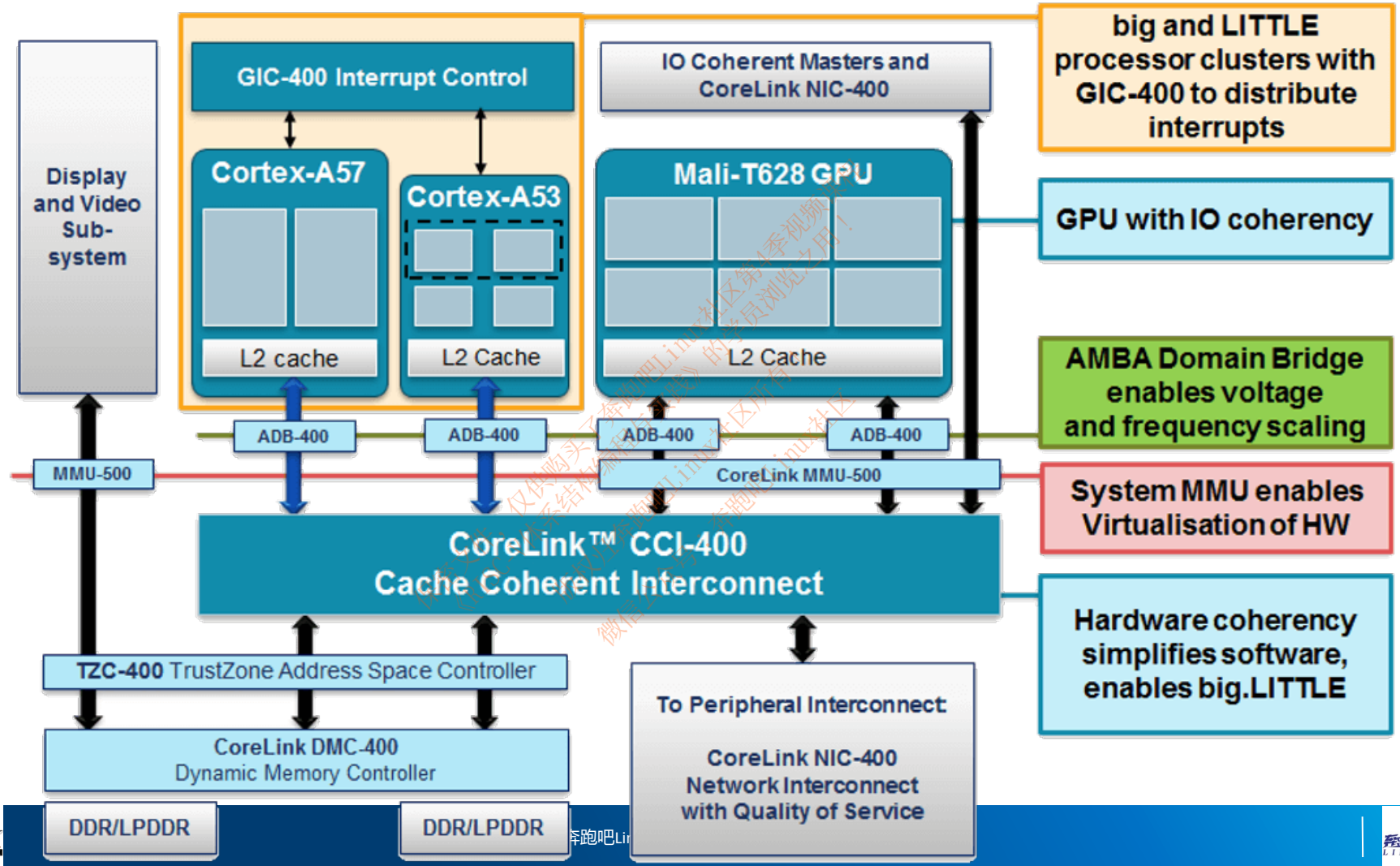
Part3：系统间的cache一致性

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程
《RISC-V体系结构编程与实战》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

CoreLink Cache Coherent Interconnect Family

	CoreLink CCI-550	CoreLink CCI-500	CoreLink CCI-400
Summary	Highly configurable ACE interconnect supporting up to 6 clusters	Highly configurable ACE interconnect supporting up to 4 clusters	Smallest area, 2 cluster coherent interconnect
Fully coherentACE Slave Interfaces	1-6	1-4	2
Processors	Up to 24 cores	Up to 16 cores	Up to 8 cores
IO CoherentACE-Lite Slave Interfaces	0-6 (maximum 7 ACE and ACE-Lite slave ports)	0-6 (maximum 7 ACE and ACE-Lite slave ports)	1-3
System and DMC Master Interfaces	1-6 memory interfaces 1-3 system	1-4 memory interfaces 1-2 system	1-2 memory interfaces 1 system
Memory map	32-48 bit Physical address 40/44/48 bit DVM	32-44 bit Physical address 40/44/48bit DVM	40 bit Physical address 44 bit DVM
Coherency and Snoop Filter	Integrated snoop filter maintains directory of processor cache contents and reduces CPU snoops	Integrated snoop filter maintains directory of processor cache contents and reduces CPU snoops	Broadcast snoop coherency
QoS	Integrated QoS mechanisms		
Power, Area, Frequency	Contact Arm for more information		

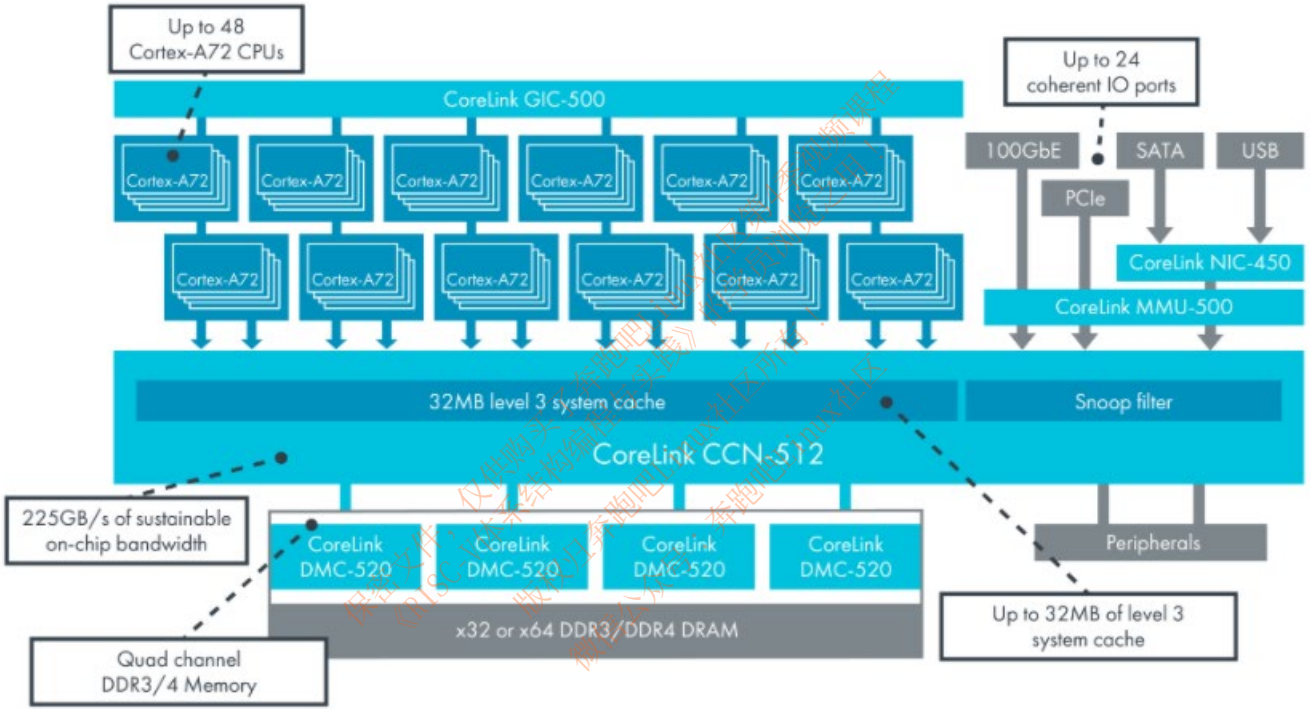
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实现》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区


GIC-400 Interrupt Control
IO Coherent Masters and CoreLink NIC-400
big and LITTLE processor clusters with GIC-400 to distribute interrupts
Display and Video Sub-system
Cortex-A57
Cortex-A53
Mali-T628 GPU
L2 cache
L2 Cache
L2 Cache
GPU with IO coherency
ADB-400
ADB-400
ADB-400
ADB-400
AMBA Domain Bridge enables voltage and frequency scaling
MMU-500
CoreLink MMU-500
System MMU enables Virtualisation of HW
CoreLink™ CCI-400
Cache Coherent Interconnect
Hardware coherency simplifies software, enables big.LITTLE
TZC-400 TrustZone Address Space Controller
To Peripheral Interconnect:
CoreLink NIC-400
Network Interconnect
with Quality of Service
CoreLink DMC-400
Dynamic Memory Controller
DDR/LPDDR
DDR/LPDDR

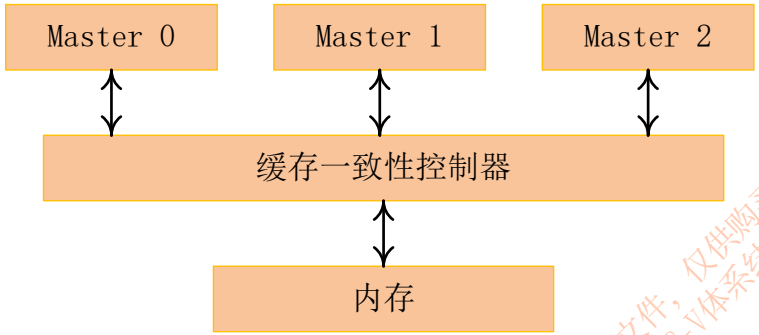
CoreLink Cache Coherent Network Family

	CoreLink CCN-512	CoreLink CCN-508	CoreLink CCN-504	CoreLink CCN-502
Summary	Up to 12 cluster interconnect (max processor bandwidth)	Up to 8 cluster interconnect (high bandwidth)	Up to 4 cluster interconnect (high IO performance)	Up to 4 cluster interconnect (optimized for size)
Performance	Up to 225 GB/s	Up to 200 GB/s	Up to 150 GB/s	Up to 100 GB/s
Processors	Up to 48 cores	Up to 32 cores	Up to 16 cores	Up to 16 cores
DDR	1 to 4 channels	1 to 4 channels	1 to 2 channels	1 to 4 channels
IO	Up to 24 AXI4/ACE-Lite	Up to 24 AXI4/ACE-Lite	Up to 18 AXI4/ACE-Lite	Up to 9 AXI4/ACE-Lite
Level 3 Cache	1-32 MB	1-32 MB	1-16 MB	0-8 MB
System Size	Large	Medium-Large	Medium	Small
Example Applications	Cloud applications, storage array network controller, Macro Base Station, and core networks.	Cloud applications, storage array network controller, Macro Base Station, and core networks.	Small enterprise solutions, cellular cell network, media content.	Home networks, home gateway, small cell base station, DSL modem, and web tier server.

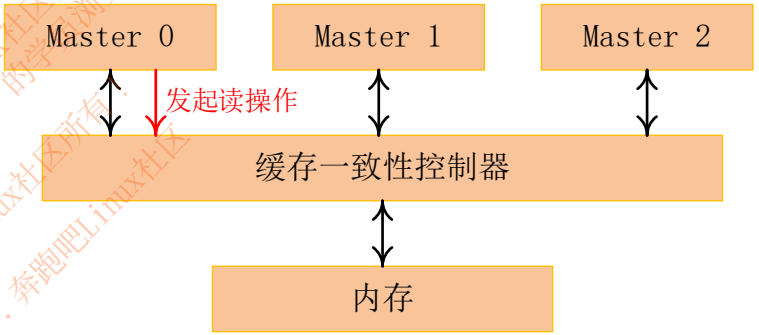
保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览与用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区


Up to 48
Cortex-A72 CPUs
CoreLink GIC-500
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Cortex-A72
Up to 24
coherent IO ports
100GbE
SATA
USB
PCIe
CoreLink NIC-450
CoreLink MMU-500
32MB level 3 system cache
Snoop filter
CoreLink CCN-512
225GB/s of sustainable
on-chip bandwidth
CoreLink
DMC-520
CoreLink
DMC-520
CoreLink
DMC-520
CoreLink
DMC-520
Peripherals
x32 or x64 DDR3/DDR4 DRAM
Up to 32MB of level 3
system cache
Quad channel
DDR3/4 Memory

系统cache一致性：读数据的例子

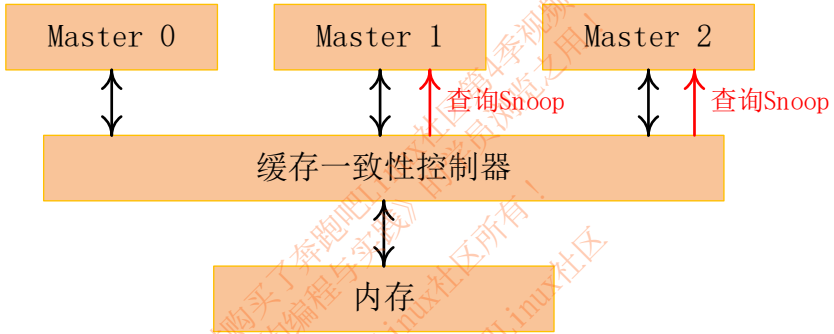


T0时刻：初始化状态



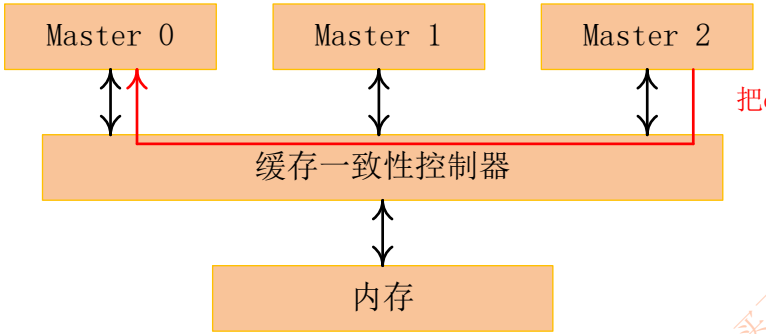
T1时刻：Master 0发起读内存操作，首先发送一个coherent request到缓存一致性控制器

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区



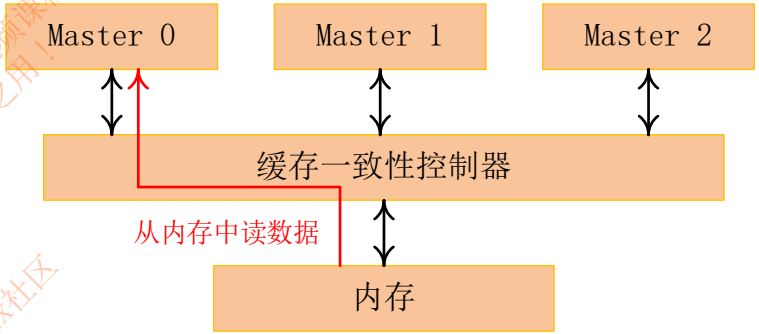
T2时刻：缓存一致性控制器通过系统总线发送查询snoop消息，询问Master1和Master2本地是否有cache副本

保密文件，仅供购买了奔跑吧Linux社区《体系结构编程与实战》课程会员使用！
版权所有：奔跑吧Linux社区



把cache副本发送到Master 0

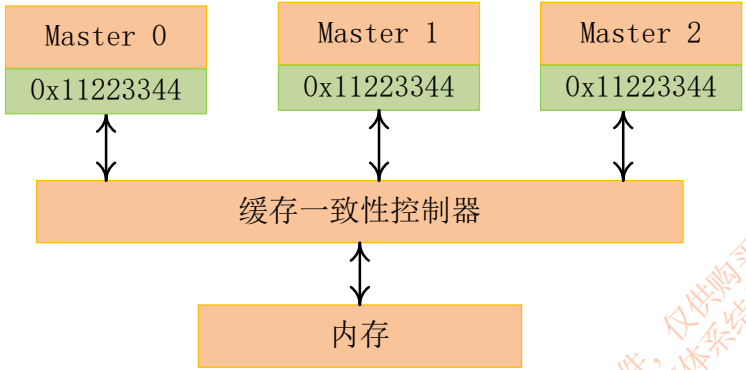
T3时刻：假设Master 2本地有cache副本，那么通过缓存一致性控制器把该cache副本发送给Master 0



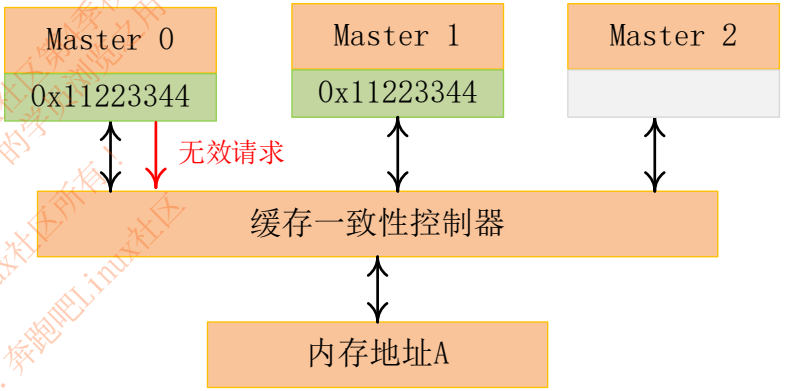
T3时刻：假设Master 1和Master 2都没有cache副本，那么直接从内存读数据到Master 0的cache line中

保密文件，供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

系统cache一致性：写数据的例子

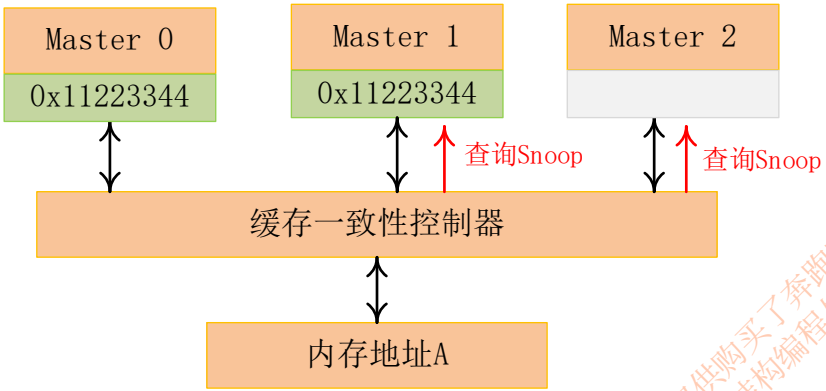


T0时刻：初始化状态，Master 0和Master 1上的cache line缓存了内存地址A的数据

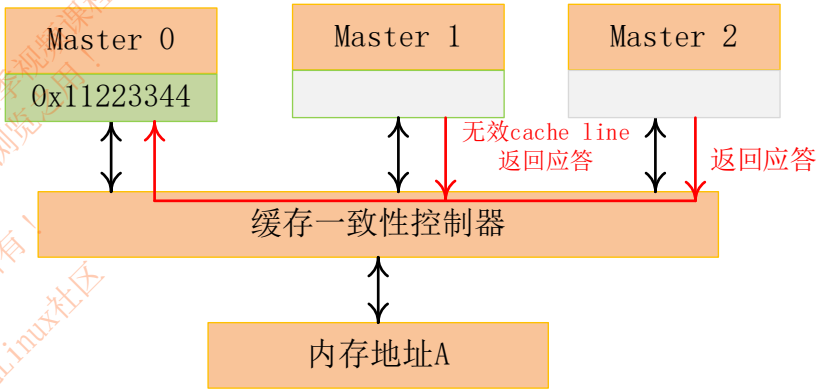


T1时刻：Master 0想写内存地址A。首先它要向缓存一致性控制器发送 一致性的无效请求（coherent invalidate request）。

保密文件，仅供购买了奔跑吧Linux社区xx季视频课程《RISC-V体系结构编程与实践》的学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



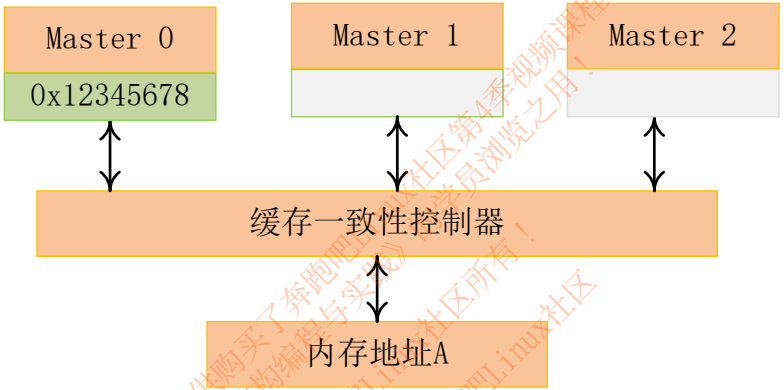
T2时刻：缓存一致性控制器往总线里发送广播消息：如果有内存地址A的cache副本的，请无效本地的cache副本



T3时刻：Master 1上cache副本，无效这个cache line，并且返回应答消息。Master 2本地没有缓存副本，直接返回应答信息。

Master 0收到应答消息。

本文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览使用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区



T4时刻： Master 0修改了内存地址A的本地cache line的数据为0x12345678

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《linux体系结构编程与实战》学员浏览之用！
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区

Part4： cache一致性的案例分析

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程
《RISC-V体系结构编程与实战》学员浏览之用！
版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区

案例1：高速缓存伪共享的避免

- 一些常用的数据结构在定义时就约定数据结构以一级缓存对齐。例如使用如下的宏来让数据结构首地址以L1 cache对齐。

```
#define cacheline_aligned   __attribute__((__aligned__(L1_CACHE_BYTES)))
```

- 数据结构中频繁访问的成员可以单独占用一个高速缓存行，或者相关的成员在高速缓存行中彼此错开，以提高访问效率。*例如struct zone数据结构使用ZONE_PADDING技术（填充字节的方式）来让频繁访问的成员在不同的cache line中。*

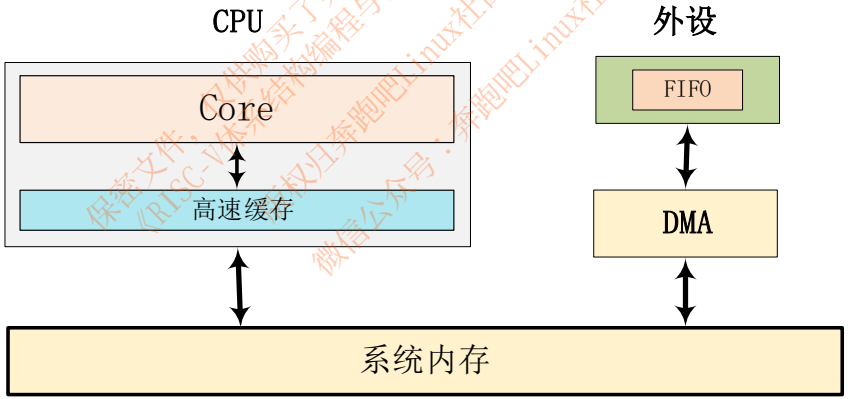
```
struct zone {
        ...
        spinlock_t      lock;
        ZONE_PADDING(_pad2_)
        spinlock_t      lru_lock;
        ...

} ____cacheline_in_smp;
```

保密文档，仅供购买了《RISC-V体系结构编程与实践》的学员浏览！
版权归奔跑吧Linux社区所有！
关注公众号：奔跑吧Linux社区，加入奔跑吧Linux社区第4期视频课程

案例2：DMA的cache一致性

- DMA（Direct Memory Access）直接内存访问，它在传输过程中是不需要CPU干预的，可以直接从内存中读写数据。

- DMA产生cache一致性问题的原因：
  - ✓ *DMA直接操作系统总线来读写内存地址，而CPU并不感知。*
  - ✓ *如果DMA修改的内存地址，在CPU的cache中有缓存，那么CPU并不知道内存数据被修改了，CPU依然去访问cache的旧数据，导致Cache一致性问题。*

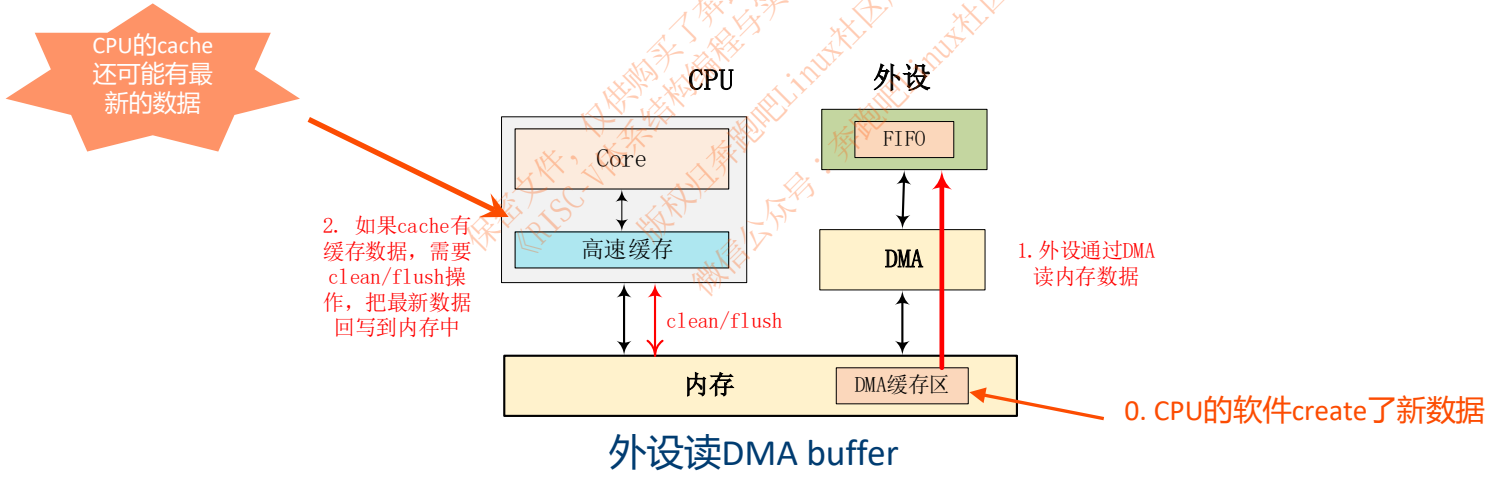


保密文件，仅供购买了奔跑吧Linux社区《RISC-V体系结构编程与实践》第4季视频课程的个人学习使用，版权归奔跑吧Linux社区所有，谢绝一切形式的传播！
微信公众号：奔跑吧Linux社区

DMA的cache一致性解决方案

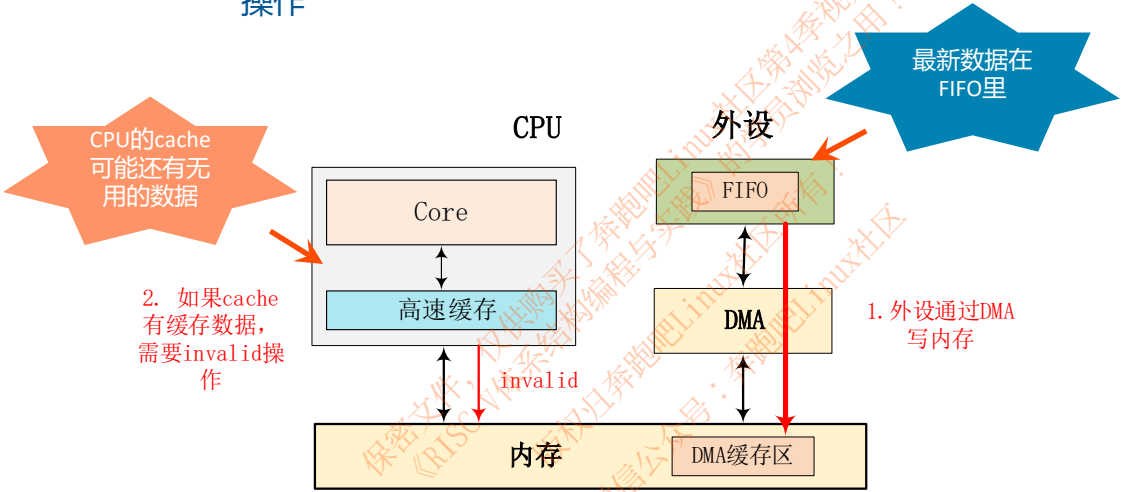
- 硬件解决办法，需要ACE支持（咨询SoC vendor）
- 使用non-cacheable的内存来进行DMA传输。
  - *缺点：在不使用DMA的时候，CPU访问这个buffer会导致性能下降*

- 软件干预cache一致性，根据DMA传输数据的方向，软件来维护cache一致性
  - Case 1：内存 →设备FIFO （设备例如网卡，通过DMA读取内存数据到设备FIFO）
    - ✓ 在DMA传输之前，CPU的cache可能缓存了内存数据，需要调用**cache clean/flush操作**，把cache内容写入到内存中。因为CPU cache里可能缓存了最新的数据。



保密文件，仅供购买了奔跑吧Linux社区《RISC-V系统架构编程与实践》的学员使用！版权归奔跑吧Linux社区所有！微信公众号：奔跑吧Linux社区

- Case 2：设备FIFO →内存 （设备把数据 写入到内存中）
  - ✓ 在DMA传输之前，需要把**cache 做invalid操作**。因为此时最新数据在设备FIFO中，CPU缓存的cache数据是过时的，一会要写入新数据，所以做invalid操作



外设写DMA buffer

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程的个人浏览之用！
《RISC-V架构与系统结构编程与实践》版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区

案例3：self-modifying code

- 指令cache和数据cache是分开的。指令cache一般只读。
- 指令cache和数据cache的一致性问题。指令通常不能修改，但是某些特殊情况指令存在被修改的情况。
- self-modifying code，在执行过程中修改自己的指令，过程如下。
  - ✓ 把要修改的指令，加载到数据cache里。
  - ✓ 程序（CPU）修改新指令，数据cache里缓存了最新的指令。

问题：

- ✓ 指令cache依然缓存了 旧的指令。
- ✓ 新指令还在数据cache里。

保密文件，仅供购买了奔跑吧Linux社区《RISC-V体系结构编程与实践》第4季视频课程学员使用之用
版权归奔跑吧Linux社区所有
微信公众号：奔跑吧Linux社区

self-modifying code解决办法

- 解决思路：使用cache管理指令和内存屏障指令来保证数据cache和指令cache的一致性
- 例子：假设t0寄存器存储了代码段的地址，通过sd指令把新的指令数据t1写入t0寄存器中，实现修改代码的功能。

```
1    sd t1, (t0)
2    cbo.flush t0
3    fence rw,rw
4    fence.i
```

保密文件，仅供购买了奔跑吧Linux社区第4季视频课程《RISC-V体系结构编程与实践》的学员浏览之用
版权归奔跑吧Linux社区所有！
微信公众号：奔跑吧Linux社区