第4季

内存屏障

本节课主要内容

- 本章主要内容
  - 为什么会产生内存乱序
  - 几种常见的内存模型
  - RVWMO：RISC-V弱一致性内存模型
  - RVWMO中的13条约束规范
  - RISC-V内存屏障指令
  - 案例分析

技术手册：

1. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA， Document Version 20191213

本节课主要讲解书上第15章内容

内存乱序产生的原因与内存一致性模型

为什么会有内存乱序?

- 内存乱序：程序在执行过程中 实际内存访问的顺序 （MO）与 程序代码约定的访存顺序（PO） 不一致
  - 程序次序（Program Order，PO）：程序代码里编写的内存访问序列。
  - 内存次序（Memory Order，MO）：站在内存角度看到的内存访问序列，也是系统所有处理器达成一致的内存操作总序列。

- 内存乱序产生的两个最基本原因：
  - 编译阶段。编译器优化导致内存乱序访问。
  - 执行阶段。
    - 单处理器系统：**CPU内部的访存子系统允许乱序执行。**
    - 多处理系统：**多个CPU访存的交互 引起内存乱序。**

示例1：CPU是先执行哪一条语句呢?

```
a = 1
b = 1
```

示例2：

CPU 0	CPU1
a=1	while(b==0) continue;
b=1	assert(a==1)

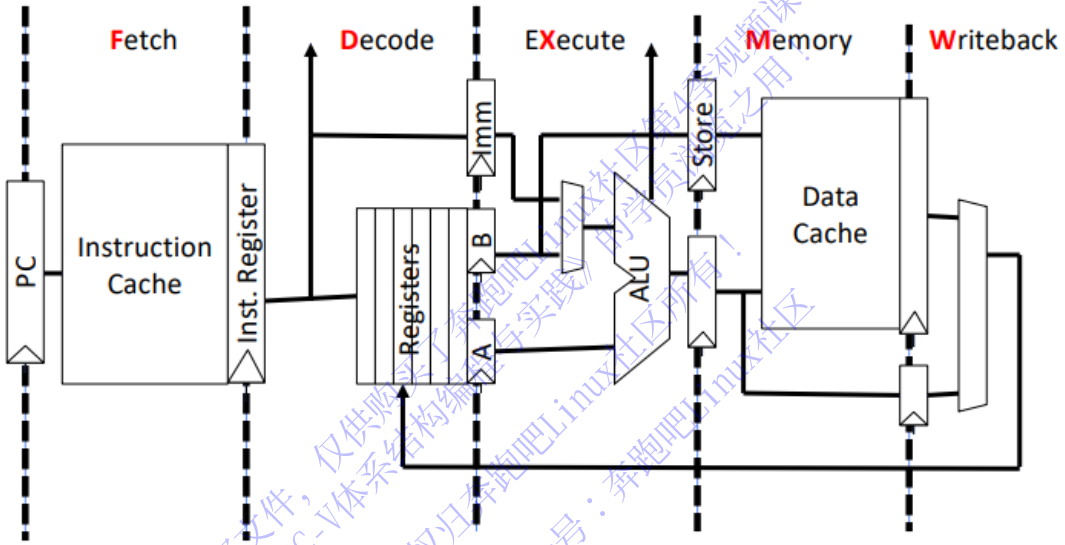
编译器导致的乱序

- 编译器会在翻译成汇编指令时对其进行优化，如内存访问指令的重新排序可以提高指令级并行效率。
- 这些优化可能会与程序员原始的代码逻辑不符，导致一些错误发生。

```
#define barrier() __asm__ __volatile__ ("" ::: "memory")
```

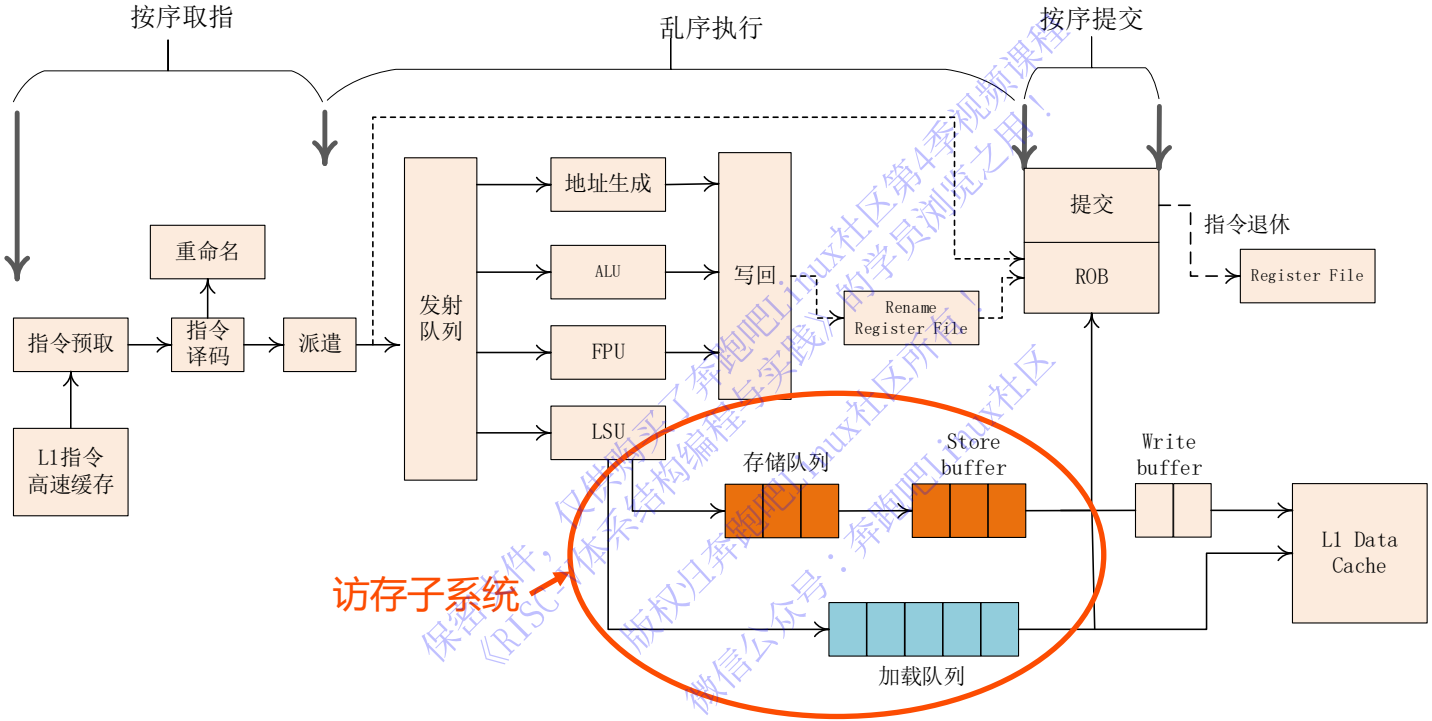
barrier()函数告诉编译器，不要为了性能优化而对这些代码重排序。

CPU体系结构1：顺序执行处理器



- 指令是完全按照程序次序执行的
- 不存在内存乱序
- 系统吞吐量低，性能低

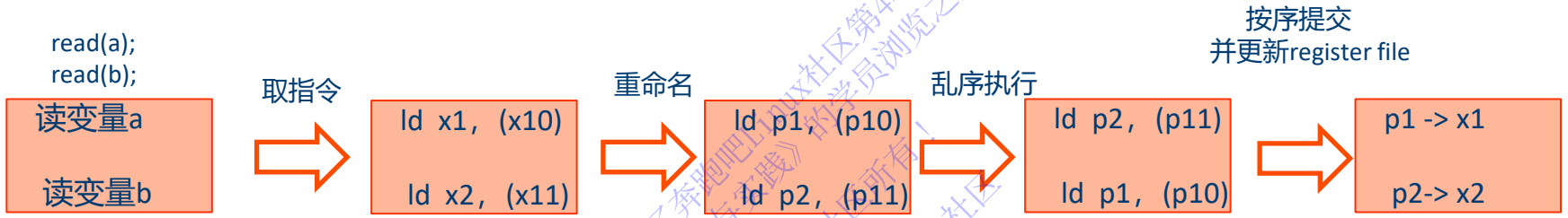
CPU体系结构2：超标量处理器（单核）

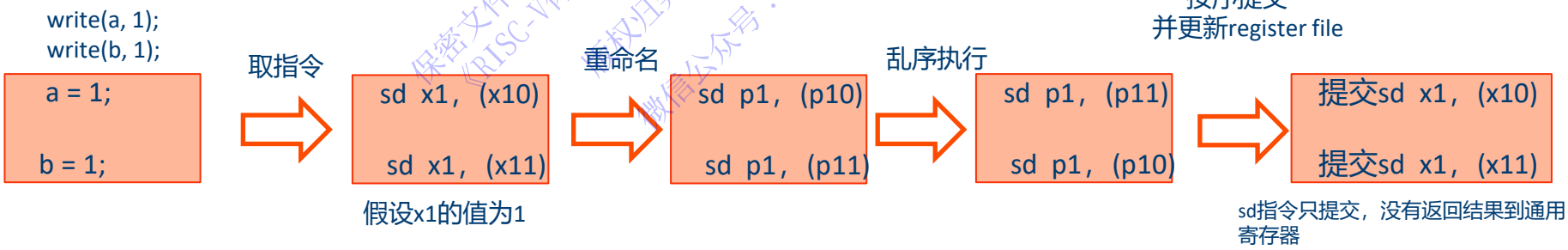


- 按序取指，乱序执行，按序提交
- 为了提高性能：访存子系统支持乱序执行

例子：两条不相干的load指令怎么乱序？

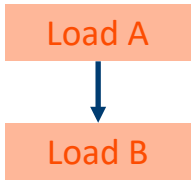
老笨，你刚才说“乱序执行，有序提交”，还说CPU保证程序最终的正确性，怎么理解？



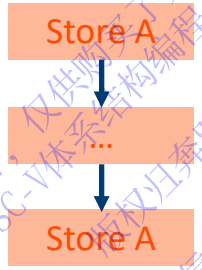


小结：访存子系统的乱序执行

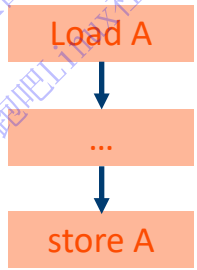
- 访存子系统，load和store指令的乱序为了提高系统访存的效率和性能
- 在单核处理器中，访存子系统的乱序执行 导致 内存乱序的发生
- 不同的内存一致性模型和处理器架构，对访存子系统有不同的约定条件，例如RVWMO约束条件



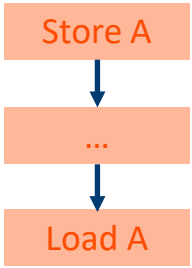
加载两个不相干的地址，可以乱序执行



先写后写WAW



先读后写RAW



先写后读WAR

内存一致性模型（memory consistency model）

- 在多处理器系统中，内存一致性用来保证：处理器对存储子系统访问的正确性
- 内存一致性模型：在一个系统中有n个处理器P1 ~ Pn，假设每个处理器中有Si个存储器操作，那么从全局来看，可能的存储器访问序列有多种组合。为了保证内存访问的一致性，需要按照某种规则来选出合适的组合

```
x = 1;
y = 1;
flag = true;

while (flag) {
        test(x, y);
        break;
}
```

单核处理器系统

Store：写1到x
Store：写1到y
Store：写true到flag
Load：读flag

CPU0

```
x = 1;
y = 1;
flag = true;
```

CPU1

```
while (flag) {
        test(x, y);
        break;
}
```

多核处理器系统

Store：写1到x
Store：写1到y
Store：写true到flag

Load：读flag

常见的内存一致性模型 1

- 原子一致性内存模型：使用一个全局时间尺度（global time scale）部件来决定存储器访问时序。
- 顺序一致性（Sequential Consistency，SC）内存模型：每一个处理器的局部时间尺度（local time scale）部件来确定最新数据
  - 从单处理器角度看，存储访问的执行次序以程序次序为准。
  - 从多处理器角度看，所有的内存访问都是原子性的，其执行顺序不必严格遵循时间顺序。

CPU0	CPU1
a = 1	x = b
b = 1	y = a

当CPU1读出x=1时，我们不可能读出y=0。

- 顺序一致性模型：保证了“读→读”“读→写”“写→写”以及“写→读”4种情况的次序。

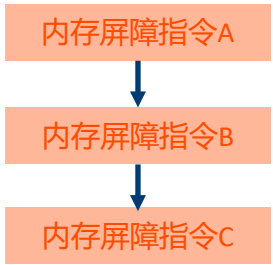
常见的内存一致性模型 2

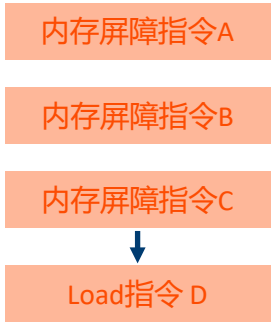
- 处理器一致性（Processor Consistency，PC）内存模型是顺序一致性内存模型的进一步弱化，放宽了“写→读”操作的次序要求。
- 允许一条Load指令从存储缓冲区（store buffer）中读取一条还没有执行的store指令的值，而且这个值还没有被写入高速缓存中
- x86-64处理器实现的全序写（Total Store Ordering，TSO）模型就属于处理器一致性内存模型的一种

常见的内存一致性模型 3：弱一致性内存模型

- 进一步放宽：放宽对“读→读”“读→写”“写→写”以及“写→读”4种情况的执行次序要求
- 程序要得到正确的预期结果，需要添加适当的同步操作（内存屏障指令）
- 对内存的访问可以分成如下几种方式。
  - 共享访问：多个处理器同时访问同一个变量，都执行读操作。
  - 竞争访问：多个处理器同时访问同一个变量，其中至少有一个执行写操作，因此存在竞争访问。

- 弱一致性内存模型（Weak Consistency，WC）定义：
  - 对全局同步变量（内存屏障指令）的访问是顺序一致的。
  - 在一个同步访问（例如，发出内存屏障指令）可以执行之前，以前的所有数据访问必须完成。
  - 在一个正常的数据访问（如数据访问指令）可以执行之前，以前的所有同步访问（内存屏障指令）必须执行完。





常见的内存一致性模型 3：弱一致性内存模型

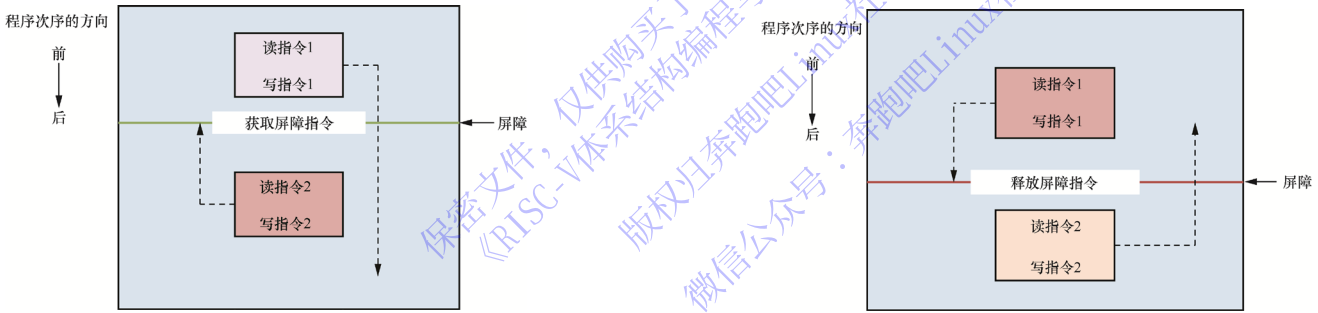
- 弱一致性内存模型实质上对同步访问和普通内存访问进行区分，然后通过同步访问（如发出内存屏障指令）解决共享数据的竞争问题，保证多处理器对共享数据的访问是顺序一致性的。

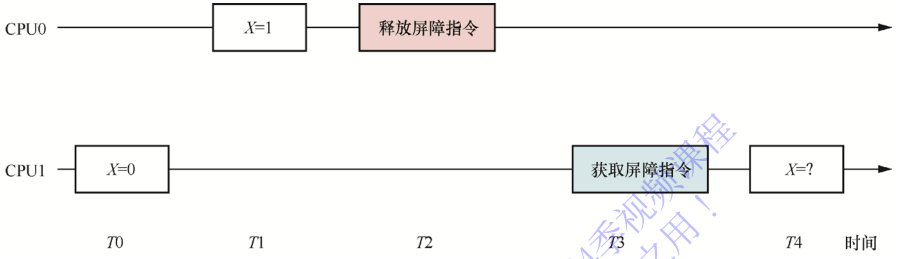
把一致性难题扔给了程序员来解决

- 支持弱一致性内存模型的处理器架构都提供内存屏障指令，例如RISC-V中fence指令

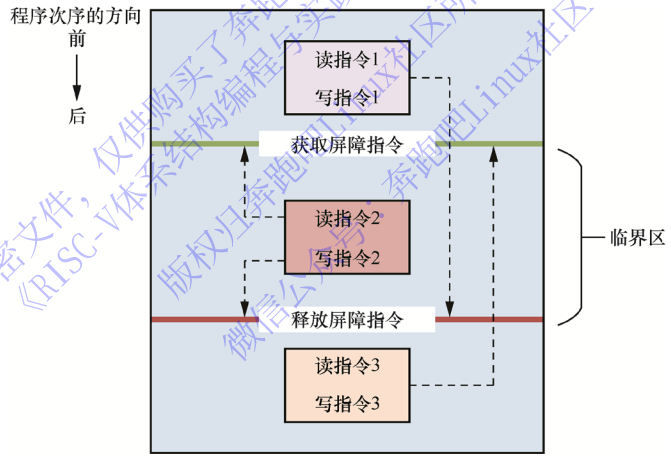
常见的内存一致性模型 4：释放一致性内存模型

- 在弱一致性内存模型的基础上新增了“获取”（acquire）和“释放”（release）屏障原语，用于简化共享数据的互斥访问。
  - 获取屏障原语之后的读写操作不能重排到该屏障原语前面
  - 释放屏障原语之前的读写操作不能重排到该屏障原语后面
  - 获取屏障原语与释放屏障原语之间是顺序执行的。





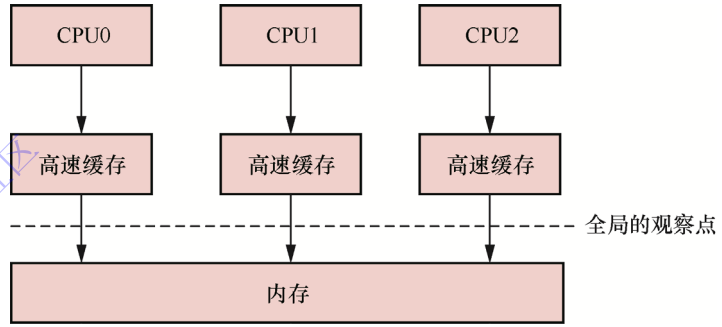
书上例15-2



书上例15-3

常见的内存一致性模型 5： MCA

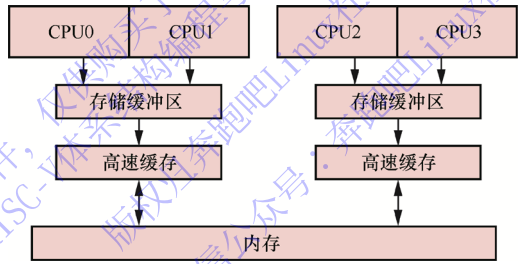
- Multi-Copy Atomicity模型（MCA）：在多核处理器系统中，如果一个本地写操作被其他任意一个观察者（如其他CPU）观察到写入完成，那么系统中所有的观察者都能观察到写入完成。
- 如果对相同地址的写入是串行的，对于所有的观察者来说，它们能观察到相同的写入序列。
- 当所有观察者都观察到写入完成之后，读操作才能读出最新的值。
- Non-MCA模型不能保证处理器的写操作立即被其他观察者观察到，这会导致系统变得复杂。



例15-4：在Non-MCA模型里，假设x和y初始值为0，如果CPU2读到的x1的值为1，那么有没有可能CPU2读到的x2的值为0呢？

如果X和Y地址的初始值为0

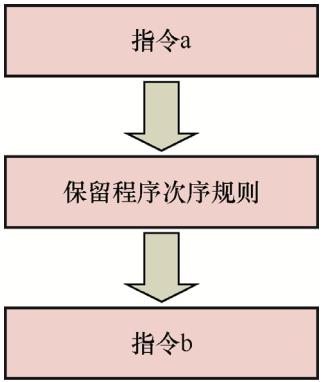
CPU0:	CPU1:	CPU2:
li x1, 1	lw x1,(X)	lw x1,(Y)
sw x1,(X)	sw x1,(Y)	fence r,r
		lw x2,(X)



因为CPU0向X地址写入时，Non-MCA模型没有办法保证CPU2能观察到这个写入操作已经完成。在Non-MCA模型中，需要添加更多的内存屏障（全局的）来确保所有的写操作都是可观察的。

RVWMO的几个概念

- 全局内存次序（global memory order，GMO）：站在内存角度看到的读和写操作的次序。
- 保留程序次序（preserved program order，PPO）：在全局内存次序中必须遵守的一些与内存次序相关的规范和约束。
- TSO的PPO：除放宽“写→读”操作的次序要求外，总的执行次序要遵从程序次序
- RVWMO: 除遵守RVWMO约定的13条规则之外，其他情况下可以乱序执行。



内存模型对比1：顺序一致性

- 公理（Axiomatic）：定义一些理论化的标准
- 可实操（operational）：定义一些可实操的标准或者规则

公理	可实操的规则
GMO：所有内存操作都有一个总顺序	每个CPU核心并行执行指令
PPO：全局内存次序 遵循 程序次序	每个CPU核心按程序次序执行
读数据返回的是从最近的写入相同地址的值	读数据返回的是从最近的写入相同地址的值

内存模型对比2： TSO, RVTSO

公理	可实操的规则
GMO：所有内存操作都有一个总顺序	每个CPU核心并行执行指令
PPO：全局内存次序 遵循 程序次序，放宽“写->读”的次序	每个CPU核心按程序次序执行
读数据返回的是，从最近的写入相同地址的值	写操作分成两步：1. 写入store buffer，2. 从store buffer写入内存
	读数据，先从store buffer中查找数据并返回数据。否则返回从最近的写入相同地址的值

内存模型对比3： RVWMO

公理	可实操的规则
GMO：所有内存操作都有一个总顺序	每个CPU核心并行执行指令
PPO：除RVWMO约定的13条规则之外，其他情况下可以乱序执行	每个CPU核心按程序次序执行
读数据返回的是，从最近的写入相同地址的值	遵循RVWMO约定的13条规则
	读数据，先从store buffer中查找数据并返回数据。否则返回从最近的写入相同地址的值

表 15.1 不同内存一致性模型中保留程序次序约定的规则

内存模型	全局内存次序	保留程序次序
顺序一致性	顺序不确定	严格按照程序次序来执行
x86 体系结构的 TSO	顺序不确定	放宽了“写→读”操作的次序要求，其他情况下需要严格按照程序次序来执行
RISC-V 的 RVWMO	顺序不确定	除 RVWMO 约定的 13 条规则之外，其他情况下可以乱序执行

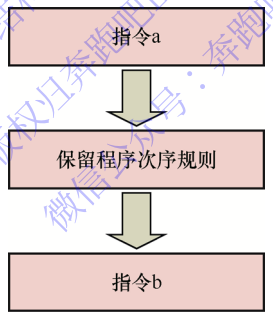
RVWMO的13条PPO约束规则

➢ **约定：**在如下指令序列中

{指令a，指令b}

假设指令a和指令b都是内存访问指令，如果指令a和指令b之间符合RVWMO约定的13条保留程序次序规则任意1条，那么在全局内存次序中指令a先执行，然后执行指令b。

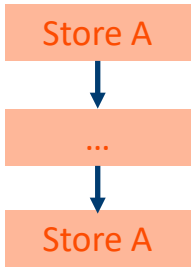
如果它们都不符合任意一条保留程序次序规则，则指令b可以比指令a先执行。



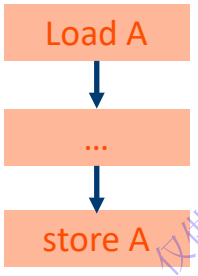
RVWMO的13个约束规范

- **规则1**：如果指令a和指令b访问相同或者重叠的内存地址，指令b执行存储操作，那么指令a必须先于指令b执行。

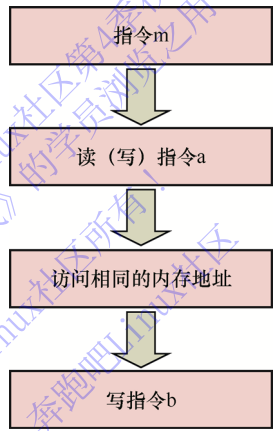
- 类似：先写后写WAW，先读后写RAW



先写后写WAW



先读后写RAW



- 例子15-5：

```
1    lw a1, 0(s1)
2    lw a2, 0(s0)
3    sw t1, 0(s0)
```

从规则1可知，如果两条连续的store指令或者load+store指令组合，它们访问的地址不重叠，那么这两条指令可以乱序执行。

- **规则2**：CoRR（Coherence for Read-Read pairs）：对同一个地址（或者重叠地址）的加载-加载操作。基本要求是新加载操作返回的值 不能 比旧加载操作返回的值更老。
- 这是一个“load-load” 问题。
- 类似先写后读WAR

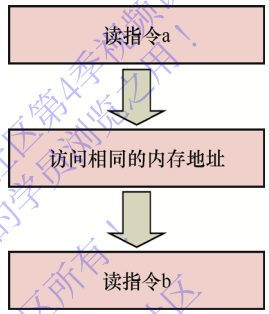
- 例子15-6：

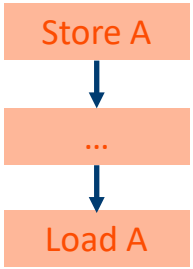
```
1    li t2, 2
2    lw a0, 0(s1)
3    sw t2, 0(s1)
4    lw a1, 0(s1)
```

Load A

Store A

Load A





先写后读WAR

- 例子15-7：

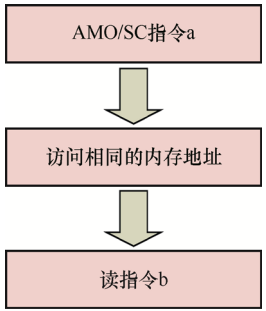
```
1    li t2, 2
2    sw t2, 0(s1)
3    lw a0, 0(s1)
4    lw a1, 0(s1)
```

从规则2可知，如果两条连续的load指令访问的地址不重叠，那么这两条指令可以乱序执行。

- **规则3**：假设指令a和指令b访问相同的地址，如果加载指令b返回的值是AMO或者SC指令a写入的值，那么在指令a执行完之前不能返回值给指令b，也就是说AMO和SC指令的执行结果还没有变成全局可见之前，不能把结果给后面的load指令。

- 例子15-8：

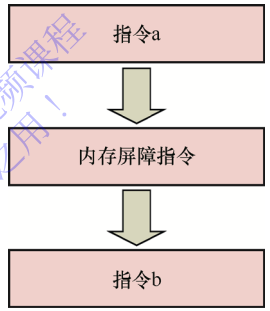
```
1    li t2, 2
2    amoadd.d a0, t2, 0(s1)
3    ld a1, 0(s1)
```



- **规则4**：如果指令a和指令b中间有内存屏障指令，a和b之间的执行次序需要遵循内存屏障指令的规则

- 例子15-9：

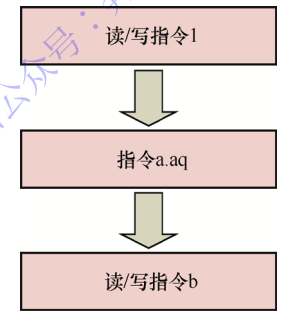
```
1    li t1, 1
2    sw t1, 0(s0)
3    fence w, w
4    sw t1, 0(s1)
```

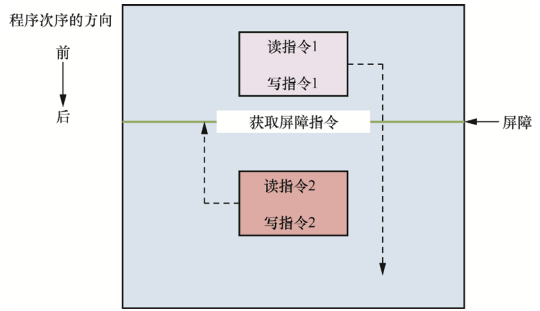


- **规则5**：指令a内置了获取内存屏障原语，如获取屏障指令。如果指令a.aq表示内置了获取屏障原语，那么指令b不能重排到指令a.aq前面，如位于指令1处

- 例子15-10：

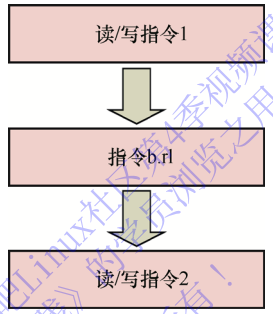
```
1        sd x1, (a1)
2        ld x2, (a2)
3        li t0, 1
4    again:
5        amoswap.w.aq t0, t0, (a0)  #获取锁
6        bnez t0, again
7        #临界区
8        sd x3, (a3)
9        ld x4, (a4)
10       ...
```

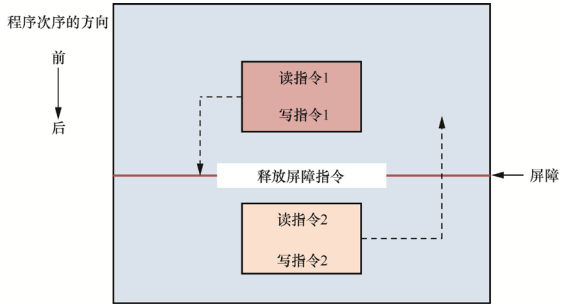




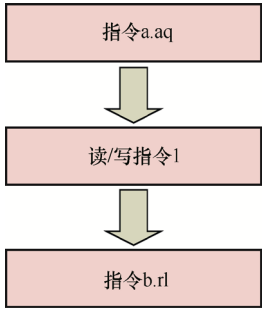
➢ **规则6**：指令b.rl表示内置了释放屏障原语，指令1不能重排到指令b.rl的后面，如位于指令2处。

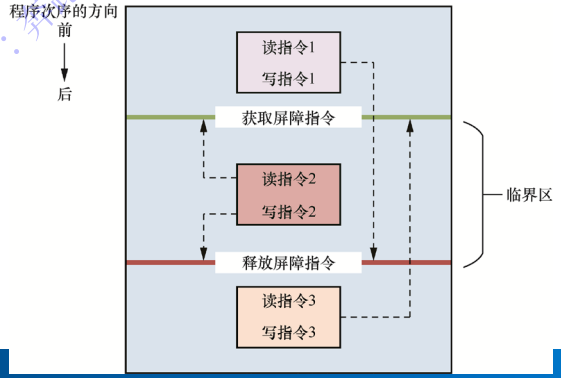
```
1        sd x1, (a1)
2        ld x2, (a2)
3        li t0, 1
4    again:
5        amoswap.w.aq t0, t0, (a0) #获取锁
6        bnez t0, again
7        #临界区
8        sd x3, (a3)
9        ld x4, (a4)
10       ...
11       amoswap.w.rl x0, x0, (a0) #释放锁
12       sd x1, (a1)
13       ld x2, (a2)
```





➢ **规则7**：指令a与b分别内置了获取与释放内存屏障原语，指令a.aq和指令b.rl形成了一个临界区，临界区内的指令不能向前或者向后越过临界区

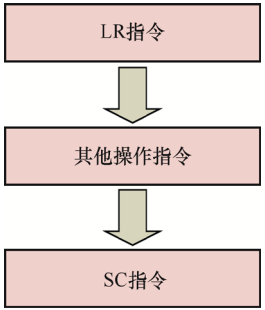




➢ **规则8**：在LR和SC指令组成的原子操作中，LR和SC指令的执行次序必须保持一致

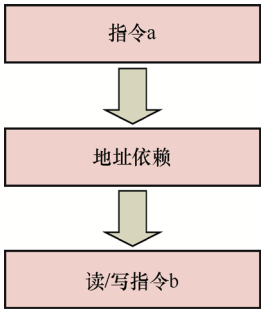
➢ 例子15-12：

```
1         li t2, 2
2     1:  lr.d  a1, 0(a2)
3         or  a1, a1, a5
4         sc.d  a3, a1, 0(a2)
5         bnez  a3, 1b
```



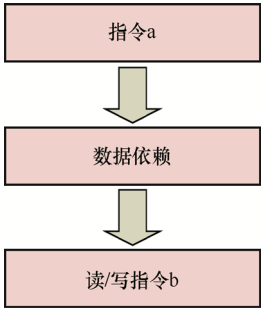
- **规则9**：指令b和指令a存在地址依赖（address dependency）。地址依赖指的是指令b寻址的内存地址源自指令a的计算结果
- 例子15-13：

```
1    ld a1, 0(s0)
2    xor a2, a1, a1
3    add s1, s1, a2
4    ld  a5, 0(s1)
```



- **规则10**：指令b和指令a存在数据依赖（data dependency）。数据依赖指的是指令b的参数来自指令a的结果
- 例子15-14：

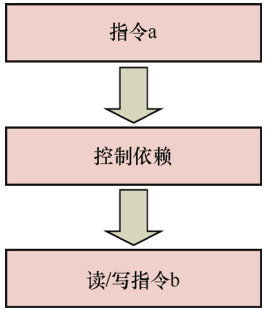
```
1  ld a1, 0(s0)
2  sd a1, 0(s1)
```



- **规则11**：指令b和指令a存在控制依赖（control dependency）。控制依赖指的是根据指令a的结果选择不同的分支，从而控制指令b是否运行

- 例子15-15：

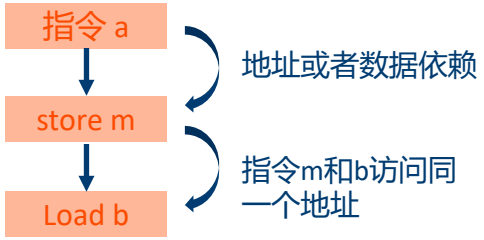
```
1        lw x1, 0(x2)
2        bne x1, x0, next
3   next:
4        sw x3, 0(x4)
```



➢ **规则12**：连环依赖。写指令m 依赖指令a，读指令b返回写指令m的值

➢ 例子15-16：

```
1    lw a0, 0(s1)
2    sw a0, 0(s2)
3    lw a1, 0(s2)
```



➢ **规则13**：连环依赖。指令a和指令b之间有一条指令m，指令b执行存储操作，指令m和指令a之间存在地址依赖

➢ 例子15-17：

```
1    lw a1, 0(s1)
2    lw a2, 0(a1)
3    sw t1, 0(s0)
```

sw指令 不能在 地址依赖解决之前 提前执行，因为有可能a1=s0

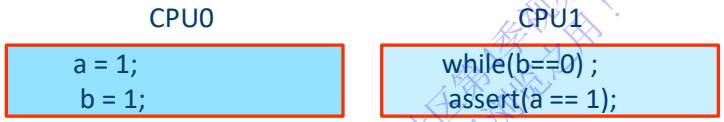
使用内存屏障指令的场景

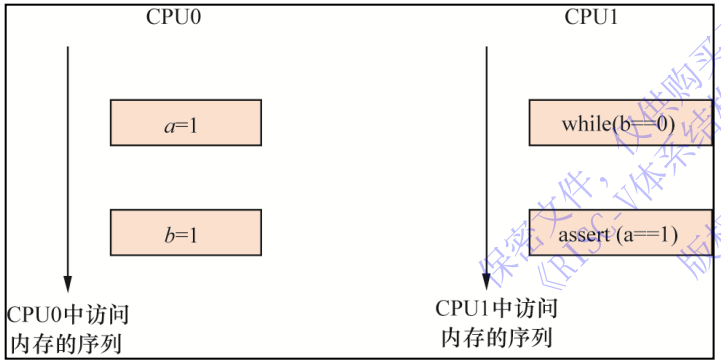
- **单核处理器系统**：CPU会保证最终执行结果符合程序员的要求。
  - CPU内部可以乱序执行，但是最终还是按序提交
  - CPU内部有内存一致性模型的约束规则，如RVWMO中约定的13条规则。
  - 例外：CPU与外设（DMA）交互，修改内存管理的策略等

- **多核处理器系统**：多核并发编程。
  - 在多个不同CPU内核之间共享数据。在弱一致性内存模型下，某个CPU的内存访问次序可能会产生竞争访问。
  - 执行和外设相关的操作，如DMA操作。
  - 修改内存管理的策略，如上下文切换、请求缺页以及修改页表等。
  - 修改存储指令的内存区域，如自修改代码的场景。

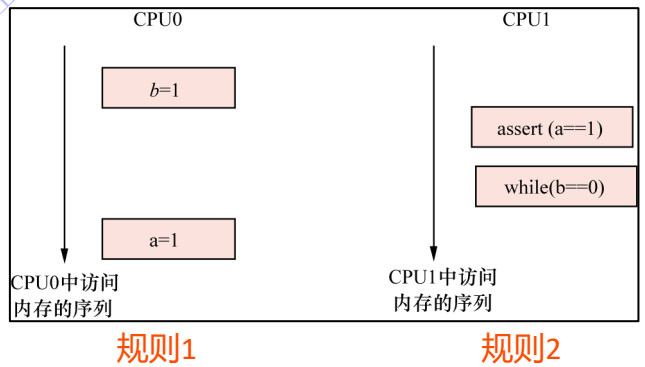
使用内存屏障指令的目的：让CPU按照程序次序（program order）来执行，而不是被CPU乱序执行破坏程序原本的功能。

重点关注：多核并发编程引发的内存乱序





理想的全局内存次序



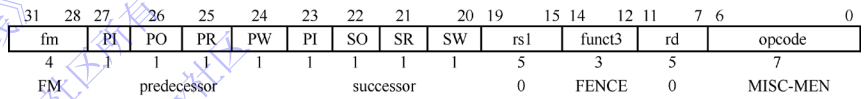
现实可能的全局内存次序

RISC-V架构中内存屏障指令1：fence指令

- fence指令可以对I/O设备和普通内存的访问进行排序。

fence iorw, iorw

- fence指令参数：表示要约束的前后指令的类型，
  - i：设备输入类型的指令
  - o：设备输出类型的指令
  - r：内存读类型的指令
  - w：内存写类型的指令



- fence指令常见组合：

```
fence iorw, iorw
fence ir, ir
fence ow, ow

fence rw, rw
fence r, r
fence w, w
```

```
#define mb()            RISCV_FENCE(iorw,iorw)
#define rmb()           RISCV_FENCE(ir,ir)
#define wmb()           RISCV_FENCE(ow,ow)

#define __smp_mb()      RISCV_FENCE(rw,rw)
#define __smp_rmb()     RISCV_FENCE(r,r)
#define __smp_wmb()     RISCV_FENCE(w,w)
```

RISC-V架构中内存屏障指令2：fence.i指令

- fence.i指令：同一个CPU中高速缓存指令与预取指令之间的同步原语。
- 简单的实现：刷新本地CPU的指令高速缓存和指令流水线，让CPU从指令高速缓存中重新取指
- fence.i指令不需要带参数
- fence.i指令仅对本地CPU有效，如果需要其他CPU也生效，则需要通过IPI通知其他CPU也执行fence.i指令。
- 使用场景：
  - 更改了PTE页表项并且该PTE页表项具有可执行权限
  - 自修改代码

RISC-V架构中内存屏障指令3： sfence.vma指令

- SFENCE.VMA指令用于实施虚拟内存管理的同步操作：
- 内存屏障功能。这条指令保证在屏障之前的存储操作与屏障之后的读写操作的执行次序。这里主要指的是对虚拟内存管理中的相关数据的读写操作，例如，对页表的读写操作等。
- 刷新TLB。这条指令还会刷新本地处理器上与地址转换相关的高速缓存，如TLB等。

- SFENCE.VMA指令的作用范围仅限于本地处理器。

sfence.vma rs1 rs2

rs1：用来指定虚拟地址
rs2：用来指定ASID

- 使用场景：
  - 更改进程ASID
  - 更新stap寄存器
  - 更新了页表，包括子叶PTE和非子叶PTE

RISC-V架构中内存屏障指令4：获取和释放屏障原语

- 在原子操作指令中提供内置的获取和释放屏障原语的变种指令，如LR/SC和AMO指令。
  - .aq：内置了获取内存屏障原语，后续的读写指令都不在该指令之前执行，例如，lr.d.aq指令。
  - .rl：内置了释放内存屏障原语，前面的读写指令都在该指令前执行完，例如，sc.d.rl指令。
  - .aqrl：同时内置了获取和释放内存屏障原语，前面的读写指令在本指令之前执行，后面的指令在本指令之后执行。

- 使用场景：
  - 使用lr/sc指令实现spinlock

案例分析

案例分析之约定

- 假设CPU实现的是RVWMO内存一致性模型。
- RISC-V中的加载指令没有内置获取和释放内存屏障原语，用“fence r , rw”内存屏障指令替代获取屏障原语

```
#define RISCV_ACQUIRE_BARRIER    "\tfence r , rw\n"
#define RISCV_RELEASE_BARRIER    "\tfence rw,  w\n"
```

- WAIT([sn]==1)宏：表示一直在等待sn寄存器的值等于1

```
loop
     ld t2,  (sn)
     li t1, 1
     bne t2, t1 loop
```

- WAIT_ACQ([sn]==1)在WAIT后面加了获取屏障原语

```
loop
     ld t2,  (sn)
     RISCV_ACQUIRE_BARRIER
     li t1, 1
     bne t2, t1 loop
```

- 所有的内存变量都初始化为0。

案例分析1：消息传递问题

```
//CPU1
     sd s5, (s1) ;    //写入新数据
     sd s0, (s2) ;    //设置标志位

//CPU2
     WAIT([s2]==1) ; //等待标志位
     ld s5, (s1) ;    //读取新数据
```

根据规则1，这两条store指令可以乱序执行

CPU2先读s1寄存器，然后等s2寄存器的标志位置位，于是CPU2读取了错误的数据。

```
1     //CPU1
2          sd s5,  (s1) ;          //写入新数据
3          RISCV_RELEASE_BARRIER ;  //释放内存屏障原语
4          sd s0,  (s2) ; //设置标志位
5
6     //CPU2
7          WAIT_ACQ([s2]==1) ;      //等待标志位
8          ld s5,  (s1) ;           //读取新数据
```

使用释放内存屏障原语保证CPU先写完新数据，再写标志位

案例分析2：自旋锁spinlock

- 自旋锁：当lock为0时，表示锁是空闲的；当lock为1时，表示锁已经被CPU持有。

```
1    loop:
2        lr.d.aq s5, (s1)
3        beq a5, s0, loop
4
5        /*锁已经释放，尝试获取锁*/
6        sc.d.rl a1, s0, (s1)
7        bnez a1, loop
8        ;  //成功获取了锁
9        sd x1, (a1)   //临界区里的存储指令
```

内置获取-释放屏障原语组成一个临界区，防止在临界区里的加载/存储指令（如第9行）被乱序重排到临界区外面

```
1    //锁的临界区里的读写操作
2    sd x1, (a1)   //临界区里的存储指令
3    ...
4    RISCV_RELEASE_BARRIER;
5    sd x0, (s1) ; //清除锁
```

使用释放内存屏障指令（见第4行），阻止锁的临界区里的加载/存储指令（如第2行）越出临界区。

案例分析3：邮箱传递消息

- 多核之间可以通过邮箱机制共享数据。
- 例子：两个CPU通过邮箱机制共享数据，其中全局变量SHARE_DATA表示共享的数据，FLAGS表示标志位。

```
1     li s1, SHARE_DATA
2     li s2, FLAGS
3
4     sd s6, (s1)   //写新数据
5     fence w,w
6     sd  x0, (s2) //更新 FLAGS 为 0，通知 CPU1 数据已经准备好
```

```
1     li s1, SHARE_DATA
2     li s2, FLAGS
3
4     //等待 CPU0 更新 FLAGS
5     loop:
6           ld s7, (s2)
7           bnez s7, loop
8
9     fence r,r
10
11    //读取共享数据
12    ld s8, (s1)
```

案例分析4：CPU与DMA交互

- 例子：启动DMA

在写入新数据到DMA缓冲区与启动DMA传输之间需要插入一条I/O设备的写内存屏障指令。

```
sd s5, (s2)   //写入新数据到 DMA 缓冲区
fence wo,wo   //I/O 设备的写内存屏障指令
sd s0, (s4)   //启动 DMA 引擎
```

通过DMA引擎读取数据也需要插入一条I/O设备的读内存屏障指令。

```
WAIT ([s4] == 1)   //等待 DMA 引擎的状态置位，这表示数据已经准备好了
fence ri,ri        //I/O 设备的读内存屏障指令
ld s5, (s2)        //从 DMA 缓冲区中读取新数据
```

RISC-V内存屏障指令移植指南

从RISC-V到x86体系结构

表 15.2　把 x86 体系结构中常见的指令映射到 RISC-V 的 RVWMO 模型

x86 体系结构中的相关指令	映射到 RVWMO 模型
加载指令	l{b|h|w|d}; fence r,rw
存储指令	fence rw,w; s{b|h|w|d}
原子操作指令	有两种方式。 采用 AMO 指令：amo<op>.{w|d}.aqrl 采用 LR/SC 指令组合： loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop
FENCE 指令	fence rw,rw

从RISC-V到ARM体系结构

表 15.3　　ARM 体系结构中的常见指令映射到 RISC-V 的 RVWMO 模型

ARM 体系结构中的相关指令	映射到 RVWMO 模型
加载指令	l{b|h|w|d}
加载-获取指令	fence rw, rw; l{b|h|w|d}; fence r,rw
独占加载指令	lr.{w|d}
内置了获取内存屏障原语的独占加载指令	lr.{w|d}.aqrl
存储指令	s{b|h|w|d}
存储-释放指令	fence rw,w; s{b|h|w|d}
独占存储指令	sc.{w|d}
内置了释放内存屏障原语的独占存储指令	sc.{w|d}.rl
DMB 指令	fence rw,rw
DMB.LD 指令	fence r,rw
DMB.ST 指令	fence w,w
ISB 指令	fence.i; fence r,r

Linux内核常用的内存屏障API函数

表 15.4　Linux 内核提供的与处理器体系结构无关的内存屏障 API 函数

API 函数	说明	RISC-V 中的实现
smp_mb()	用于 SMP 环境下的读写内存屏障指令	`fence rw,rw`
smp_rmb()	用于 SMP 环境下的读内存屏障指令	`fence r,r`
smp_wmb()	用于 SMP 环境下的写内存屏障指令	`fence w,w`
dma_rmb()	用于 I/O 设备的读内存屏障	`fence r,r`
dma_wmb()	用于 I/O 设备的写内存屏障	`fence w,w`
mb()	单处理器系统版本的读写内存屏障指令	`fence iorw,iorw`
rmb()	单处理器系统版本的读内存屏障指令	`fence ri,ri`
wmb()	单处理器系统版本的写内存屏障指令	`fence wo,wo`
smp_load_acquire()	用于 SMP 环境下带获取内存屏障的加载操作	`l{b|h|w|d}; fence r,rw`
smp_store_release()	用于 SMP 环境下带释放内存屏障的存储操作	`fence rw,w; s{b|h|w|d}`

API 函数	说明	RISC-V 中的实现
atomic_<op>_acquire	带获取内存屏障的原子操作	采用 AMO 指令：amo<op>.{w|d}.aq 采用 LR/SC 指令： loop: lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop
atomic_<op>_release	带释放内存屏障的原子操作	采用 AMO 指令：amo<op>.{w|d}.rl 采用 LR/SC 指令： loop: lr.{w|d}; <op>; sc.{w|d}.aqrl; bnez loop
atomic_<op>	带获取-释放内存屏障的原子操作	采用 AMO 指令：amo<op>.{w|d}.aqrl 采用 LR/SC 指令： loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop
atomic_<op>_relaxed	不带内存屏障的原子操作	采用 AMO 指令：amo<op>.{w|d} 采用 LR/SC 指令： loop: lr.{w|d}; <op>; sc.{w|d}; bnez loop

模拟和测试内存屏障故障

使用Litmus测试工具集

- Litmus是一套用于测试内存一致性模型的工具集，常用于模拟程序内存一致性等相关问题，如模拟系统无锁编程、访问共享内存问题、使用内存屏障等。

- 在Ubuntu 20.04主机上安装Litmus。

```
$ sudo apt install opam
$ opam init
$ opam update
$ opam install herdtools7
$ eval $(opam config env)
```

- herdtools7工具包含如下3个小程序。
  - litmus7：在真实硬件上运行的测试程序。
  - herd7：模拟器，可以在主机上模拟RISC-V等处理器体系结构的内存一致性问题。
  - diy7：脚本生成器。

案例分析：使用Litmus测试

<共享数据的伪代码>

```
1    static int data, flag;
2
3    void CPU0()
4    {
5        data = 55;  //写 55 到共享数据
6        flag = 1;   //设置 flag 标志位
7    }
8
9    void CPU1()
10   {
11       while (flag == 0)  //等待标志位置位
12           ;
13
14       assert(data == 55); //断言 data 等于 55
15   }
```

第14行的断言会不会失败呢？请使用Litmus来模拟这个场景

<riscv_test.litmus脚本>

```
1    RISCV test
2    "test shared_data and flag without memory barrier"
3    {
4    0:a0=data; 0:a1=flag;
5    1:a1=flag; 1:a0=data;
6    }
7     P0             | P1              ;
8     li t0, 55      | li t0, 1        ;
9     li t1, 1       | LC00:           ;
10    sd t0, (a0)    | ld t1, (a1)     ;
11    sd t1, (a1)    | bne t0, t1, LC00 ;
12                   | ld a5, (a0)     ;
13   exists (1:a5!=55)
```

触发条件：a5不等于55

符合这样的退出条件，说明有内存乱序问题

```
rlk@master:litmus$ herd7 -cat riscv.cat riscv_test.litmus
Test test Allowed
States 2
1:x15=0;
1:x15=55;
Ok
Witnesses
Positive: 3 Negative: 3
Condition exists (not (1:x15=55))
Observation test Sometimes 3 3
Time test 0.01
Hash=429d02be8843fa3cd80444317f0afecd
```

发现了3次触发条件，说明有3种情况触发了内存乱序

修复

<riscv_test_mb.litmus脚本>

```
1   RISCV test
2   "test shared_data and flag with memory barrier"
3   {
4   0:a0=data; 0:a1=flag;
5   1:a1=flag; 1:a0=data;
6   }
7    P0             | P1              ;
8    li t0, 55      | li t0, 1        ;
9    li t1, 1       | LC00:           ;
10   sd t0, (a0)    | ld t1, (a1)     ;
11   fence w, w     | bne t0, t1, LC00 ;
12   sd t1, (a1)    | fence r, r      ;
13                  | ld a5, (a0)     ;
14  exists (1:a5!=55)
```

```
rlk@master:litmus$ herd7 -cat riscv.cat riscv_test_mb.litmus
Test test Allowed
States 1
1:x15=55;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (not (1:x15=55))
Observation test Never 0 3
Time test 0.01
Hash=fa88f0815cecd4c75e8e608cc32876ee
```

在P0和P1线程中增加正确的内存屏障指令

LKMM（Linux- Kernel Memory Model）Linux内核内存模型

- Linux内核内置了一套基于Linux内核的内存一致性模型测试—LKMM（Linux- Kernel Memory Model，Linux内核内存模型）测试，允许使用C语言来编写Litmus脚本

<c_test.litmus脚本>

```
1   C test
2   "shared_data and flag test without memory barrier"
3   { }
4   P0(int *data, int *flag)
5   {
6       WRITE_ONCE(*data, 55);
7       WRITE_ONCE(*flag, 1);
8   }
9   P1(int *data, int *flag)
10  {
11      int a= READ_ONCE(*flag);
12      int b = READ_ONCE(*data);
13  }
14  exists (1:a=1 /\ 1:b!=55)
```

触发条件：a=1并且b不等于55

符合这样的退出条件，说明有内存乱序问题

```
$ cd runninglinuxkernel_5.15/tools/memory-model
$ herd7 -conf linux-kernel.cfg c_test.litmus
Test test Allowed
States 4
1:a=0; 1:b=0;
1:a=0; 1:b=55;
1:a=1; 1:b=0;
1:a=1; 1:b=55;
Ok
Witnesses
Positive: 1 Negative: 3
Condition exists (1:a=1 /\ not (1:b=55))
Observation test Sometimes 1 3
Time test 0.01
Hash=bead5c0fd70620854941017462b4731
```

发现了1次触发条件，说明有1种情况触发了内存乱序

修复

<c_test_mb.litmus脚本>

```
1   C test
2   "shared_data and flag test with memory barrier"
3   { }
4   P0(int *data, int *flag)
5   {
6       WRITE_ONCE(*data, 55);
7       smp_wmb();
8       WRITE_ONCE(*flag, 1);
9   }
10  P1(int *data, int *flag)
11  {
12      int a= READ_ONCE(*flag);
13      smp_rmb();
14      int b = READ_ONCE(*data);
15  }
16  exists (1:a=1 /\ 1:b!=55)
```

```
rlk@master:tmp$ herd7 -conf linux-kernel.cfg c_test_mb.litmus
Test test Allowed
States 3
1:a=0; 1:b=0;
1:a=0; 1:b=55;
1:a=1; 1:b=55;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:a=1 /\ not (1:b=55))
Observation test Never 0 3
Time test 0.01
Hash=eeb3e191172c6fd47a4bf60cbf81701b
```

使用纯C语言来模拟内存屏障

创建了两个线程，然后通过CPU_SET()接口函数让run_thread0()运行在CPU0上，让run_thread1()运行在CPU1

<asm.s 代码片段>

```
1    /*
2       void run_thread0(unsigned long *data, unsigned long *flag,
3             unsigned long value);
4     */
5    .global run_thread0
6    run_thread0:
7        li t0, 1
8
9        sd  a2, (a0) //写新数据
10       sd  t0, (a1) //更新 flag 为 1,通知 CPU1 数据已经准备好
11
12       ret
13
14
15   /*
16      unsigned long run_thread1(unsigned long *data, unsigned long *flag);
17    */
18   .global run_thread1
19   run_thread1:
20      li t1, 1
21
22     //等待 CPU0 更新 flag
23   loop:
24       ld t0, (a1)
25       bne t0, t1, loop
26
27       //读出共享数据
28       ld a0, (a0)
29
30       ret
```

实验1：编写Litmus脚本并测试内存一致性1

1．实验目的

熟悉使用Litmus工具进行内存一致性验证。

2．实验要求

x和y是全局变量，a是CPU0中的局部变量，b是CPU1中的局部变量。

<伪代码片段>

```
1     void CPU0()
2     {
3         x = 1; //写 1 到 x
4         a = y; //读取 y 的值
5     }
6
7     void CPU1()
8     {
9         y = 1; //写 1 到 y
10        b = x; //读取 x 的值
11    }
```

（1）是否存在这样的情况，即CPU0中变量a和CPU1中变量b的值都等于0的情况？请编写Litmus脚本并验证。如果存在上述情况，请思考如何修复。

（2）编写C程序来验证。

实验2：编写Litmus脚本并测试内存一致性2

1．实验目的

熟悉使用Litmus工具进行内存一致性验证。

2．实验要求

x、y和z是全局变量，a是CPU1中的局部变量，b和c是CPU2中的局部变量。。

```
<伪代码片段>

1    void CPU0()
2    {
3        x = 1;  //写 1 到 x
4        y = 1;  //写 1 到 y
5    }
6
7    void CPU1()
8    {
9        a = y;  //读取 y 的值
10       z = 1;  //写 1 到 z
11   }
12
13   void CPU2()
14   {
15       b = z;  //读取 z 的值
16       c = x;  //读取 x 的值
17   }
```

（1）是否存在这样的情况，即CPU1中变量a和CPU2中变量b的值都等于1并且CPU2中变量c的值等于0的情况？请编写Litmus脚本并验证。如果存在上述情况，请思考如何修复。

（2）编写C程序来验证。

总结：该如何去理解内存屏障？

- **建议1：从处理器微架构去理解：单核处理器微架构和多核处理器架构**
- **建议2：从内存一致性模型角度去理解**

- **单核处理器系统**：CPU会保证最终执行结果符合程序员的要求。
  - CPU内部可以乱序执行，但是最终还是按序提交，并程序运行正确
  - CPU内部有内存一致性模型的约束规则，如RVWMO中约定的13条规则。

- **多核处理器系统**：多核并发编程。
  - 在多个不同CPU内核之间共享数据。在弱一致性内存模型下，某个CPU的内存访问次序可能会产生竞争访问。

程序员需要会分析多核并发的内存乱序问题，合理使用内存屏障指令

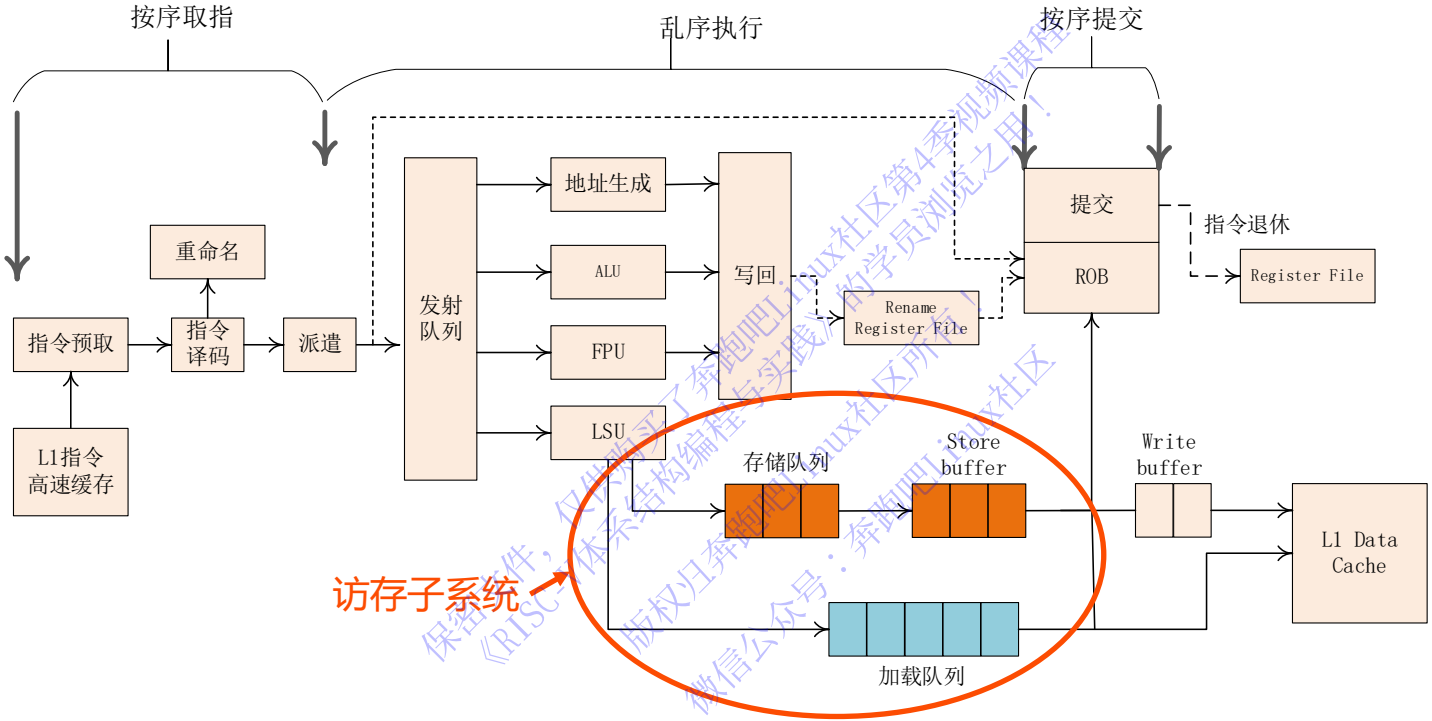
总结：RISC-V规范中关于内存屏障相关的内容

- RISC-V架构中的内存一致性模型支持弱一致性模型（RVWMO）和RVTSO模型
- 目前绝大部分RISC-V处理器采用RVWMO模型
- RVWMO模型 基于弱一致性模型 + MCA模型
- RVWMO提炼了13条约束规则：
  - ✓ 规则1 ~ 规则3是约定load和store指令的
  - ✓ 规则4~7是关于内存屏障指令的约束
  - ✓ 规则8是关于原子操作指令的约束
  - ✓ 规则9~13是关于依赖的约束

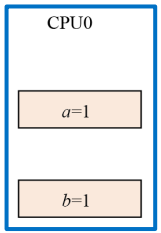
- RISC-V架构提供4种内存屏障指令
  - ✓ fence指令
  - ✓ fence.i指令
  - ✓ sfence.vma指令
  - ✓ 获取与释放内存屏障原语

- 只有原子操作指令可以内置获取和释放内存屏障原语，普通load和store不支持

CPU体系结构：超标量处理器（单核）



- 按序取指，乱序执行，按序提交
- 为了提高性能：访存子系统支持乱序执行



CPU体系结构：多核处理器架构