第4季

原子操作

本节课主要内容

- 本章主要内容
  - 为什么需要原子操作
  - RISC-V中的LR/SC指令
  - 独占访问工作原理
  - 原子内存操作指令
  - CAS指令与无锁操作

技术手册：

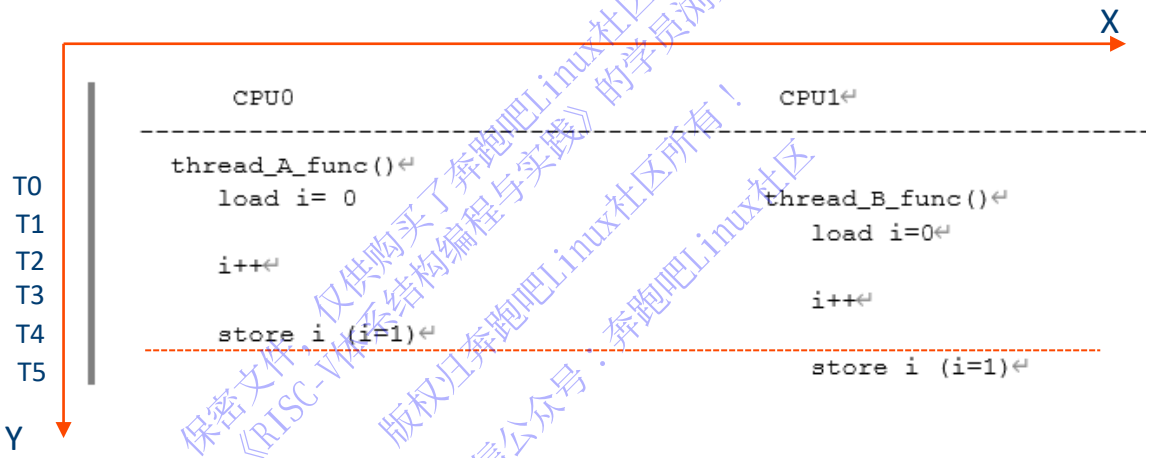
1. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA， Document Version 20191213
2. SiFive U74-MC Core Complex Manual, 21G2.01.00

本节课主要讲解书上第14章内容

为什么需要原子操作?

- 假设thread_A_func和thread_B_func都尝试进行i++操作，
  请问thread-A-func和thread-B-func执行完后，i的值是多少?

有的读者可能认为是2，但也可能不是2，例如下面的步骤中i的结果为1



什么是原子操作?

- 原子操作：指一组相关联的操作要么都不间断的执行，要么都不执行，即保证不会被打断
- 要保证操作的完整性和原子性，通常需要“原子地”（不间断地）完成“读-修改-回写”机制，中间不能被打断。
  - 读取原子变量的值，从内存中读取原子变量的值到寄存器。
  - 修改原子变量的值，在寄存器中修改原子变量的值。
  - 把新值写回内存中，把寄存器中的新值写回内存中。
- 处理器须提供原子操作的指令
  - RISC-V提供保留加载（Load-Reserved，LR）与条件存储（Store-Conditional，SC）指令
  - 原子内存访问指令。

LL/SC机制

- LL/SC机制： Load-Link/Store-Conditional
  - LL：从指定内存地址读取一个值，处理器会监控这个内存地址
  - SC：如果这段时间内其他处理器没有修改该内存地址，则把新值写入该地址。
- LL/SC指令常常用于实现无锁算法与“读-修改-回写”原子操作。
- RISC-V的A扩展指令集里实现了LR和SC指令。

RISC-V中的LR/SC指令

- LR（load-reserved）指令：

```
lr.w rd, (rs1)
lr.d rd, (rs1)
```

LR指令会注册一个保留集（reservation set），包含所寻址的一组字节

- SC（store-conditional）指令：

```
sc.w rd, rs2, (rs1)
sc.d rd, rs2, (rs1)
```

有条件地把rs2寄存器的值存储到rs1地址中，执行的结果反映到rd寄存器中。

- rd = 0，执行成功
- rd != 0, 说明执行失败

SC执行不管成功还是失败，保留集都会失效

LR和SC指令需要配对使用

例子：atomic_add(i,v)函数

```
1   #include <stdio.h>
2
3   static inline void atomic_add(int i, unsigned long *p)
4   {
5       unsigned long tmp;
6       int result;
7
8       asm volatile("# atomic_add\n"
9       "1: lr.d  %[tmp], (%[p])\n"
10      "   add   %[tmp], %[i], %[tmp]\n"
11      "   sc.d  %[result], %[tmp], (%[p])\n"
12      "   bnez  %[result], 1b\n"
13      : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p)
14      : [i]"r" (i)
15      : "memory");
16  }
17
18  int main(void)
19  {
20      unsigned long p = 0;
21
22      atomic_add(5, &p);
23
24      printf("atomic add: %ld\n", p);
25  }
```

书上例14-2

LR/SC指令需要注意的地方

- LR和SC指令是配对使用
- SC指令执行成功的条件：
  - 当前保留集有效
  - 保留集中包含的数据被成功更新或者写入

- 无论SC指令执行成功与否，当前CPU包含的保留集都被无效掉。

- SC指令可能会失败的情况：
  - 如果SC写入的地址，不在与之配对的LR指令组成的保留集范围内。
  - 如果在LR和SC指令范围内，执行了另外一条SC指令（这条SC指令写任何地址）
  - 如果在LR和SC指令范围内，执行了另外一条store指令并且是对LR的加载地址进行store的
  - 如果另外一个CPU对当前LR和SC的保留集地址进行写入操作。
  - 如果另外一个外设（不是CPU）对LR加载的数据进行写入

LR/SC指令的约束规范

- RISC-V架构对LR/SC序列做了一些约束，不符合这些约束的LR/SC序列不能保证在所有RISC-V处理器中都能成功。

1. LR/SC的循环loop中最多包含16条指令
2. LR/SC序列包括RV64I指令或者压缩指令，但是不包括加载、存储、向后跳转、向后分支、JALR、FENCE、以及SYSTEM指令。
3. LR/SC序列可以包含向后跳转的retry重试。
4. SC指令的地址必须与同一个CPU执行的最新LR的有效地址和数据大小相同。

LR/SC指令错误使用案例1

```
3 static inline int atomic_add(int i, unsigned long *p, unsigned long *p1)
4 {
5         unsigned long tmp, tmp1 =1;
6         int result;
7
8         asm volatile("# atomic_add\n"
9 "1:      lr.w    %[tmp], (%[p])\n"
10 "       add     %[tmp], %[i], %[tmp]\n"
11 "       sc.w    %[result], %[tmp], (%[p1])\n"
12 //"     bnez    %[result], 1b\n"
13        : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p), [tmp1] "=&r" (tmp1), [p1]"+r"(p1)
14        : [i]"r" (i)
15        : "memory");
16
17        return result;
18 }
```

SC写入的地址，不在与之配对的LR指令组成的保留集范围内

SC写入的是指针p1的地址，而LR加载的是指针p对应的数据

LR/SC指令错误使用案例2

```
3  static inline int atomic_add(int i, unsigned long *p, unsigned long *p1)
4  {
5          unsigned long tmp, tmp1 =1;
6          int result;
7
8          asm volatile("# atomic_add\n"
9  "1:     lr.w    %[tmp], (%[p])\n"
10 "       add     %[tmp], %[i], %[tmp]\n"
11 "       sc.w    %[result], %[tmp], (%[p1])\n"
12 "       sc.w    %[result], %[tmp], (%[p])\n"
13 //"     bnez    %[result], 1b\n"
14         : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p), [tmp1] "=&r" (tmp1), [p1]"+r"(p1)
15         : [i]"r" (i)
16         : "memory");
17
18         return result;
19 }
```

在LR和SC指令范围内，执行了另外一条SC指令，这里SC指令写入的是P1地址

LR/SC指令错误使用案例3

```
3 static inline int atomic_add(int i, unsigned long *p, unsigned long *p1)
4 {
5         unsigned long tmp, tmp1 =1;
6         int result;
7         unsigned char *p2 = (unsigned char *)p;
8         p2 = p2 + 4;
9
10        printf("%llx %llx\n", (unsigned long)p, (unsigned long)p2);
11
12        asm volatile("# atomic_add\n"
13 "1:     lr.w    %[tmp], (%[p])\n"
14 "       add     %[tmp], %[i], %[tmp]\n"
15 //"     sw      %[tmp], (%[p2])\n"
16 "       sw      %[tmp], (%[p])\n"
17 "       sc.w    %[result], %[tmp], (%[p])\n"
18 //"     bnez    %[result], 1b\n"
19        : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p), [tmp1] "=&r" (tmp1), [p2]"+r"(p2)
20        : [i]"r" (i)
21        : "memory");
22
23        return result;
24 }
```

在LR和SC指令范围内，执行了另外一条store指令，这条store指令写入的地址为LR加载的地址

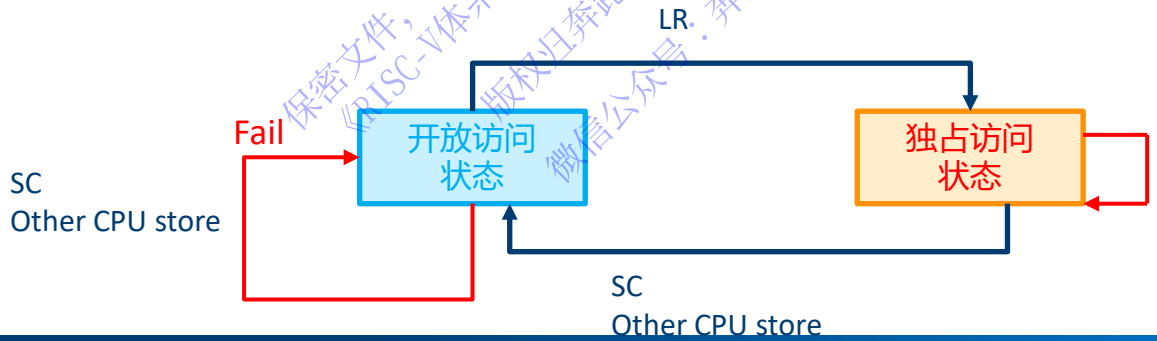
独占访问工作原理

- RISC-V指令手册中并没有约定LR/SC指令如何实现
- 本节课介绍一种基于独占监视器（exclusive monitor）来监控内存访问的方法。

- 独占监视器一共有两个状态：开放访问状态（Open Access state）和独占访问状态（Exclusive Access state）

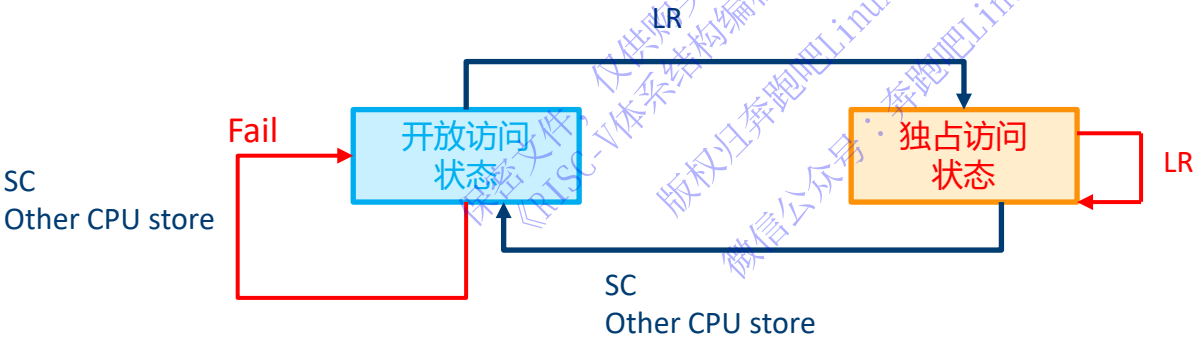
- LR指令从内存加载数据时，CPU会把这个内存地址标记为独占访问状态

- 当CPU执行SC指令的时候，需要根据独占监视器的状态来做决定
  - 如果独占监视器的状态为独占访问状态，那么SC指令存储成功，SC指令返回0，独占监视器的状态变成了开放访问状态。
  - 如果独占监视器的状态为开发访问状态，那么SC指令存储失败，stxr指令返回1



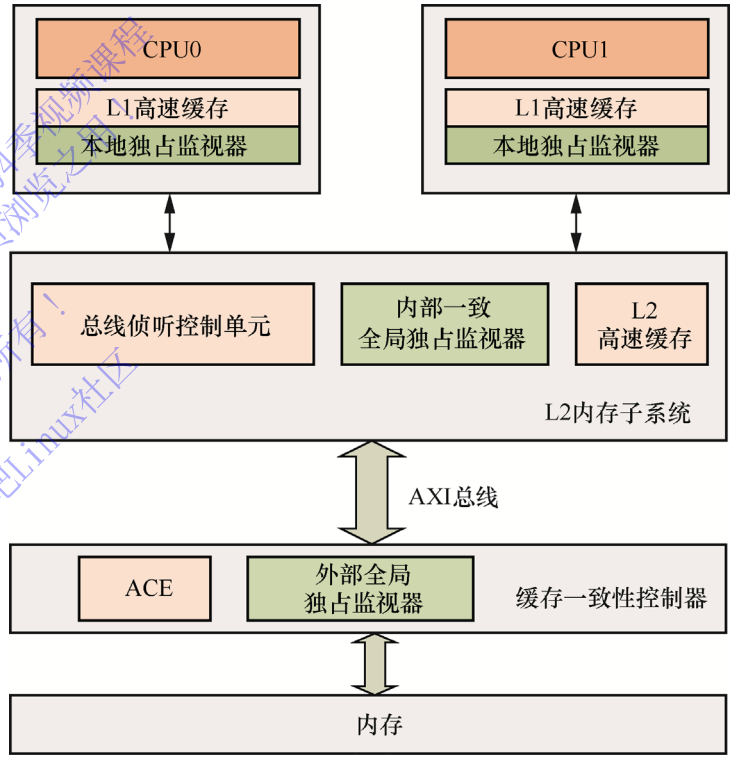
独占监视器注意事项

- 独占监视器本身不是用来阻止CPU核心来访问被标记的内存，不会lock总线
- 独占监视器 仅仅是起到监视的作用，监视状态的变化
- 不能把 独占监视器看成是一个硬件的锁



独占监视器的组成架构

- 通常一个系统由多级独占监视器组成（由芯片设计时定义）
  - ✓ 本地独占监视器（Local monitor）
  - ✓ 缓存一致性的全局独占监视器（Internal coherent global monitor）
  - ✓ 外部的全局独占监视器（External global monitor）

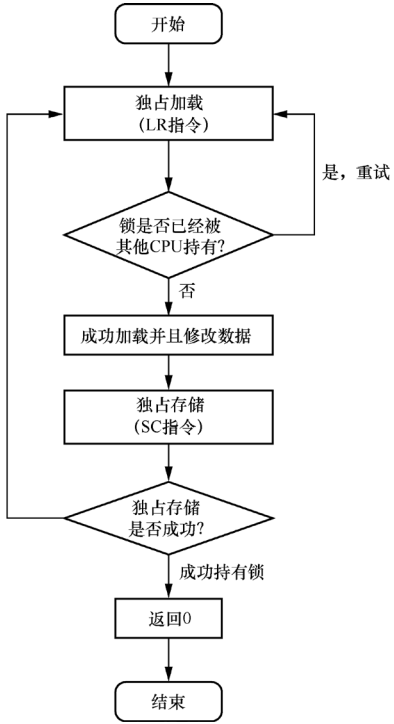


独占监视器与缓存一致性

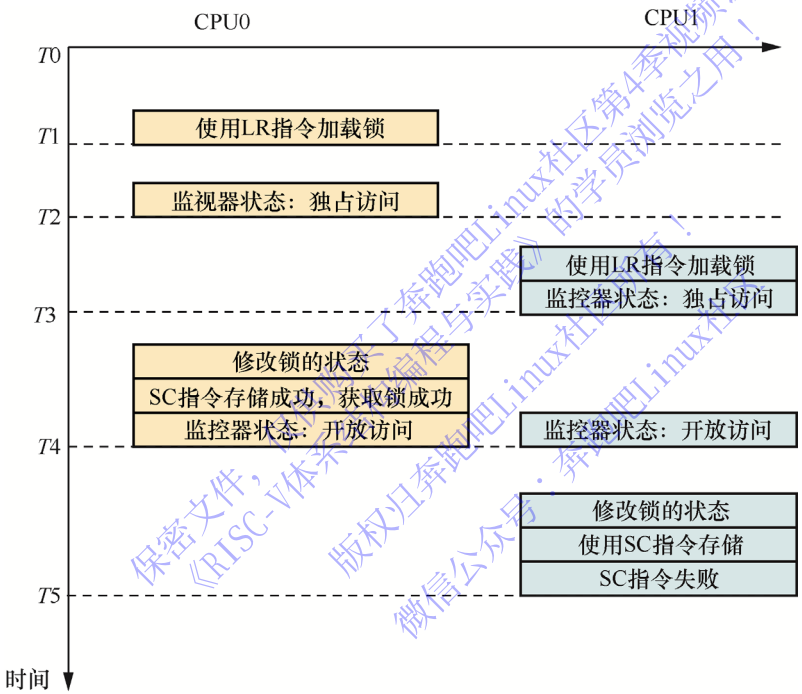
假设CPU0和CPU1同时访问一个锁（lock），这个锁的地址为a0寄存器的值，下面是获取锁的伪代码。

<获取锁的伪代码>

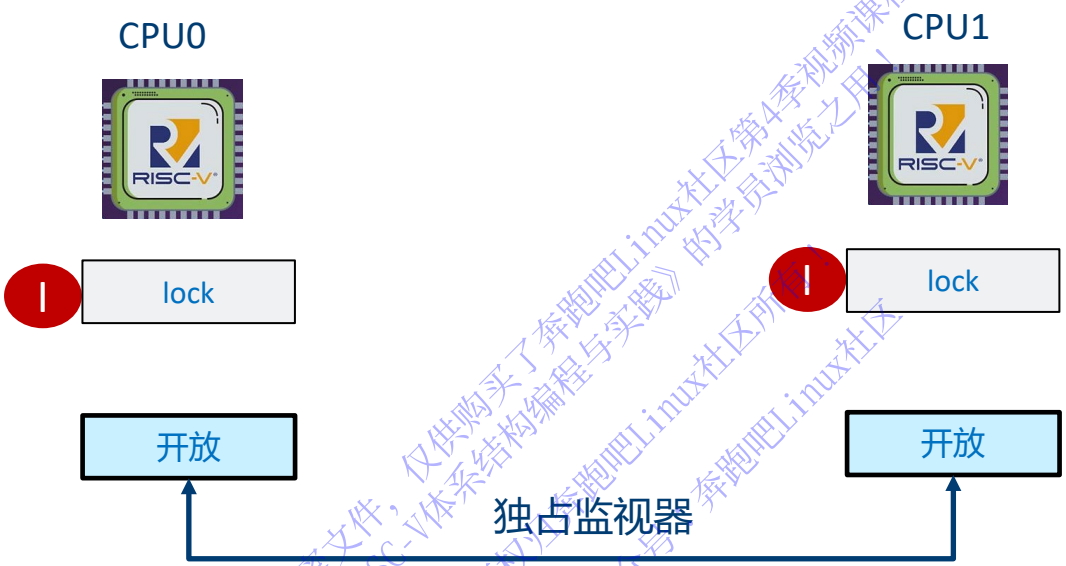
```
1     /*
2        get_lock(lock)
3      */
4
5     .global get_lock
6     get_lock:
7         li a2, 1
8     retry:
9         lr.w a1, (a0)
10        beq a1, a2, retry
11
12        /* 锁已经释放，尝试获取锁 */
13        sc.w a1, a2, (a0)
14        bnez a1, retry
15
16        ret
```



CPU0和CPU1同时执行get_lock()操作

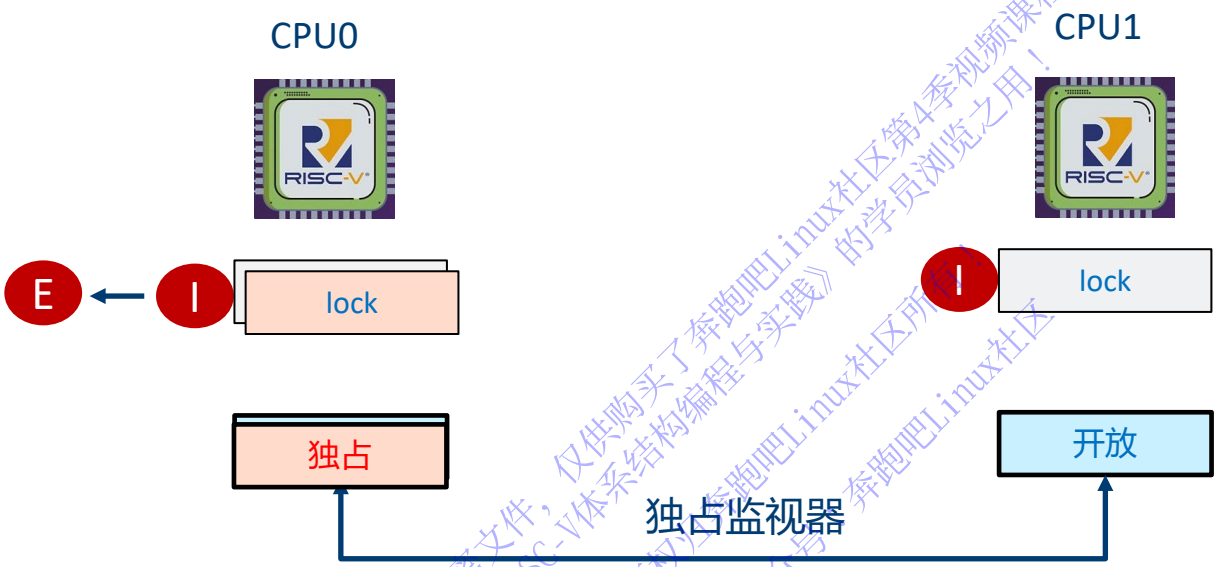


T0时刻：初始化状态



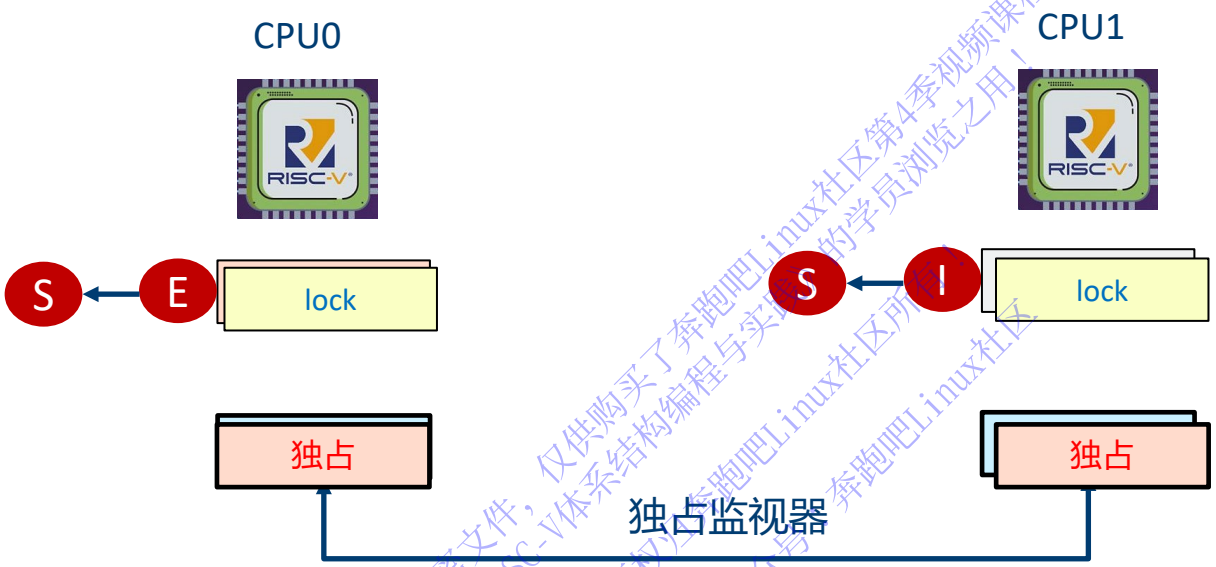
T1 & T2时刻

T1和T2时刻：CPU0执行LR指令



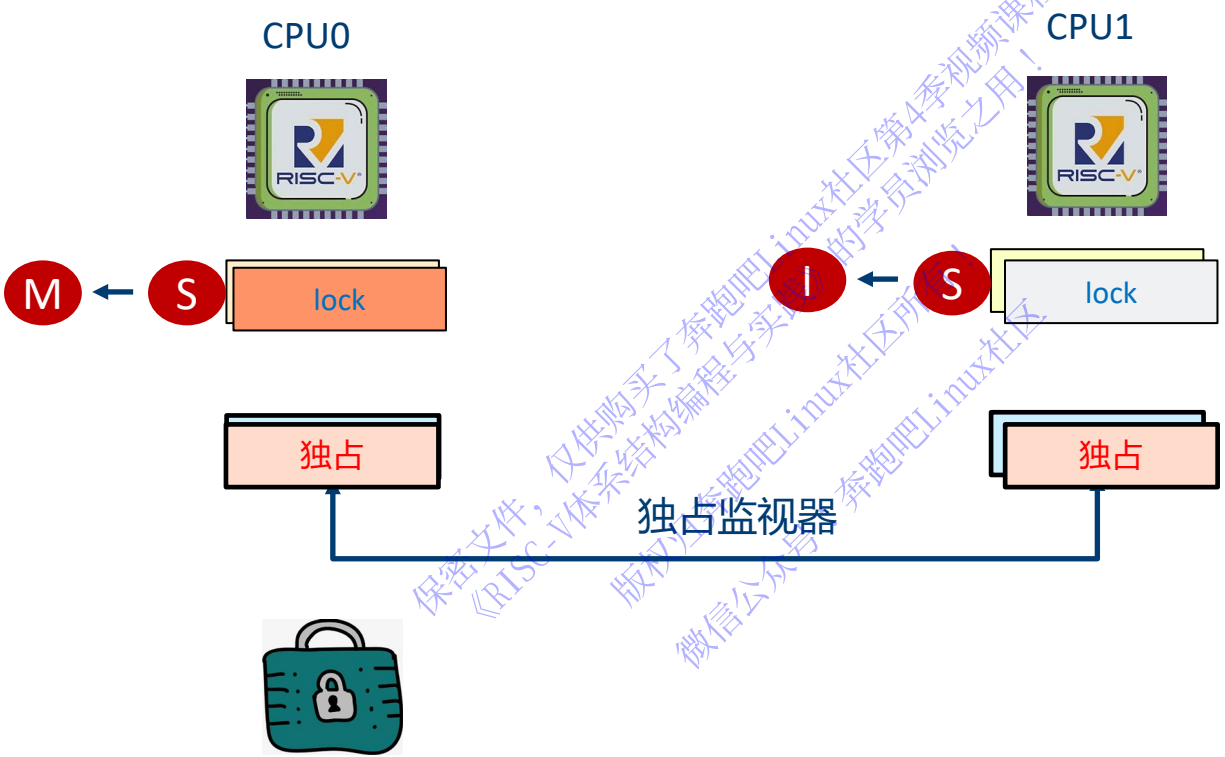
T3时刻

T3时刻：CPU1执行LR指令



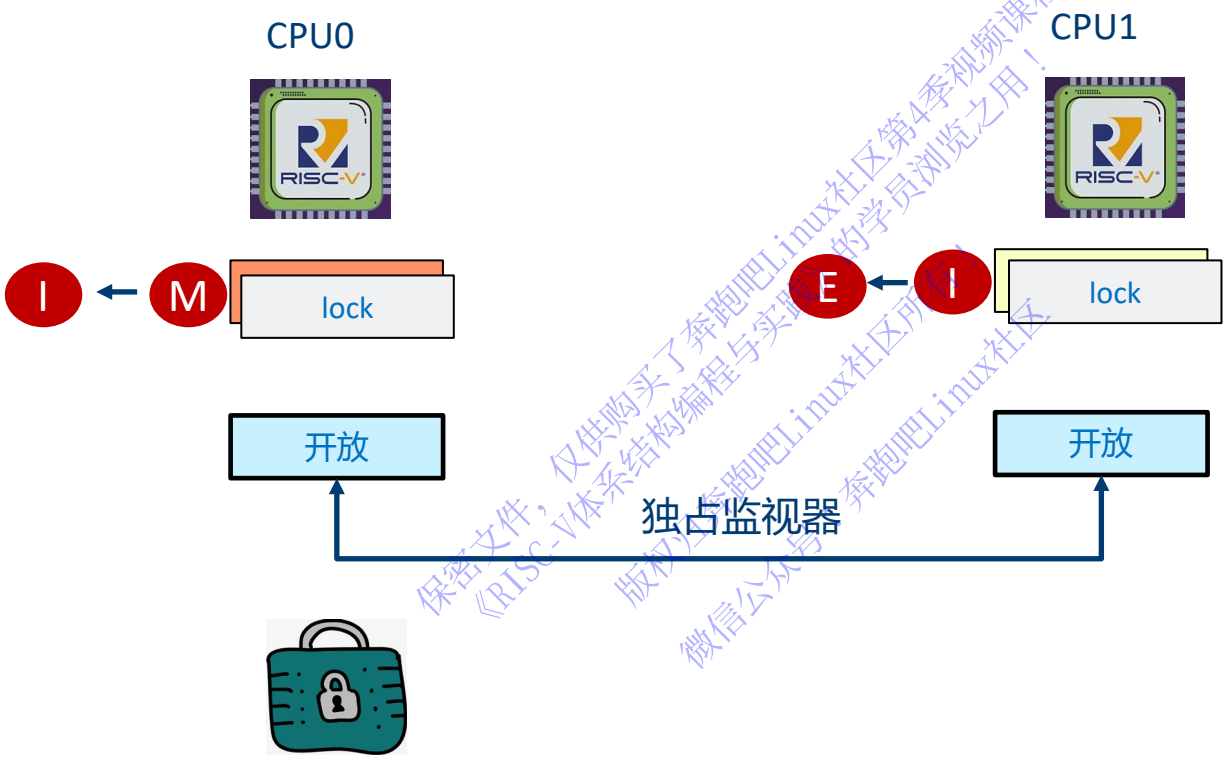
T4时刻

T4时刻：CPU0执行SC指令来获取了锁



T5时刻

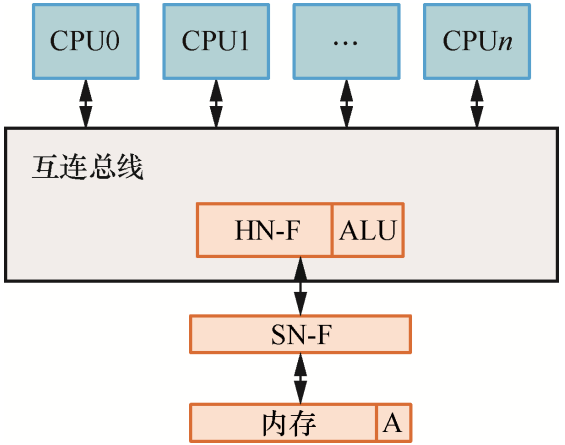
T5时刻：CPU1执行SC指令尝试获取了锁



由于现在lock对应的内存区域的状态为：开放访问状态，所以SC指令执行失败

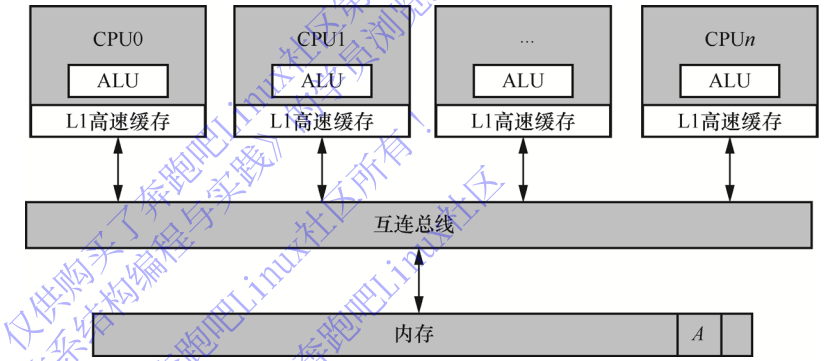
原子内存操作

- 原子内存操作指令，它允许在靠近数据的地方原子地实现“读-修改-写回”操作。
  - 近端原子（near atomic）操作
  - 远端原子（far atomic）操作：需要CHI总线或者TiLink总线支持
- AMBA 5总线引入了原子事务（atomic transaction），允许将原子操作发送到数据端，并且允许原子操作在靠近数据的地方执行
- 例子：内存中的地址A存储了一个计数值，CPU0执行一条AMOADD指令使计数值加1
  1. 发出一个原子存储事务（atomic store transaction）请求到互连总线上。
  2. HN-F会协同SN-F以及ALU来完成加法原子操作。
  3. CPU发送完该事务就认为AMOADD指令已经执行完



原子内存访问指令与LR/SC指令的效率对比

- 在独占内存访问体系结构下，ALU位于每个CPU内核内部，如果多个CPU同时争用一个内存数据，导致cache颠簸。



- 原子内存操作指令在互连总线中的HN-F节点中对所有发起访问的CPU请求进行全局仲裁，并且在HN-F节点内部完成算术运算，从而避免高速缓存颠簸消耗的总线带宽。

RISC-V中的原子内存访问指令

- 原子内存访问指令格式：

```
amo<op>.w/d  rd, rs2, (rs1)
```

表 14.1 操作后缀

操作后缀	说明
swap	交换
add	加法运算
and	与操作
or	或操作
xor	异或操作
max	求有符号数的最大值
maxu	求无符号数的最大值
mix	求有符号数的最小值
mixu	求无符号数的最小值

- 例子：AMOADD指令，执行过程的伪代码：

```
rd = *rs1;
*rs1 = rd <op> rs2;
return rd
```

例子：用AMOADD指令实现atomic_add()函数

```
1  static inline void atomic_add(int i, unsigned long *p)
2  {
3          unsigned long result;
4
5          asm volatile("# atomic_add\n"
6  "       amoadd.d %[result], %[i], (%[p])\n"
7          : [result]"=&r"(result) , [p]"+r" (p)
8          : [i]"r" (i)
9          : "memory");
10 }
```

书上例14-5

例子：使用AMOMAX指令来实现经典的自旋锁

```
/*
    get_lock(lock)
*/

.global get_lock
get_lock:
        li a2, 1
retry:
        amomax.w a1, a2, (a0)
        bnez a1, retry

        ret

/*
    free_lock(lock)
*/
.global free_lock
free_lock:
        sw x0, (a0)
```

书上例14-6

比较并交换操作

- CAS操作：检查ptr指向的值与expected是否相等。若相等，则把new的值赋给ptr；否则，什么也不做。

```
int compare_swap(int *ptr, int expected, int new)
{
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;

    return actual;
}
```

- ARMv8和x86体系结构提供了专用的CAS指令，但是RISC-V中的没有，而推荐使用LR/SC指令来实现
  - CAS指令会有ABA问题，但是LR/SC指令可以避免这个问题
  - CAS还需要一种新的整数指令格式来支持3种源操作数（地址、比较值、交换值）以及一种不同的内存系统消息格式，这将使处理器设计变得复杂化。

例子：使用LR/SC指令实现cmpxchg()函数

```
 3 static inline unsigned long cmpxchg(volatile void *ptr, unsigned long old, unsigned long new)
 4 {
 5         unsigned long tmp;
 6         unsigned long result;
 7
 8         asm volatile(
 9 "1:     lr.d    %[result], (%[ptr])\n"
10 "       bne  %[result], %[old], 2f\n"
11 "       sc.d  %[tmp], %[new], (%[ptr])\n"
12 "       bnez    %[tmp], 1b\n"
13 "       2:\n"
14         : [result]"+r" (result), [tmp]"+r" (tmp), [ptr]"+r" (ptr)
15         : [new]"r" (new), [old]"r"(old)
16         : "memory");
17
18         return result;
19 }
```

书上例14-7

例子：使用LR/SC指令实现xchg()函数

xchg(new, v)函数：把new赋给原子变量v，然后返回原子变量v的旧值。

```
 3 static inline unsigned long xchg(volatile void *ptr, unsigned long new)
 4 {
 5         unsigned long tmp;
 6         unsigned long result;
 7
 8         asm volatile(
 9 "1:     lr.d    %[result], (%[ptr])\n"
10 "       sc.d  %[tmp], %[new], (%[ptr])\n"
11 "       bnez    %[tmp], 1b\n"
12         : [result]"+r" (result), [tmp]"+r" (tmp), [ptr]"+r" (ptr)
13         : [new]"r" (new)
14         : "memory");
15
16         return result;
17 }
18
```

书上例14-8

无锁（lock-free）操作

- 在链表并发操作过程中，不需要额外使用操作系统提供的锁机制，使用CAS指令来实现
- Linux内核的MCS锁，qspinlock锁的实现使用了无锁操作

思考题：如果在LR和SC序列中发生了进程切换，那么LR/SC还能成功吗?

```
3 static inline void atomic_add(int i, unsigned long *p)
4 {
5         unsigned long tmp;
6         int result;
7
8         asm volatile("# atomic_add\n"
9 "1:      lr.d    %[tmp], (%[p])\n"
10 "       add     %[tmp], %[i], %[tmp]\n"
11 "       sc.d    %[result], %[tmp], (%[p])\n"
12 "       bnez    %[result], 1b\n"
13        : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p)
14        : [i]"r" (i)
15        : "memory");
16 }
```

如果在第10行的时候发生了进程切换，会怎么样?

总结

- RISC-V架构的原子操作通过A扩展指令集来支持
- RISC-V架构中的LR/SC指令采用LL/SC机制
- RISC-V架构手册没有约定LR/SC指令在微架构中如何实现
- LR/SC指令会注册保留集，手册中约定了几种情况会导致SC指令执行失败
- RISC-V架构手册对LR/SC指令序列做了约束
- RISC-V架构支持原子内存操作指令
- RISC-V架构没有单独的CAS指令