Amirkabir University of Technology
(Tehran Polytechnic)

Operating Systems

Synchronization Tools-Part2

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2024

Hardware Support for Synchronization

Memory model

- **Memory model** are the memory guarantees a computer architecture makes to application programs.

- IMC: Integrated Memory Controller
- The QPI is a point-to-point connection protocol developed by Intel to replace the front-side-bus (FSB).

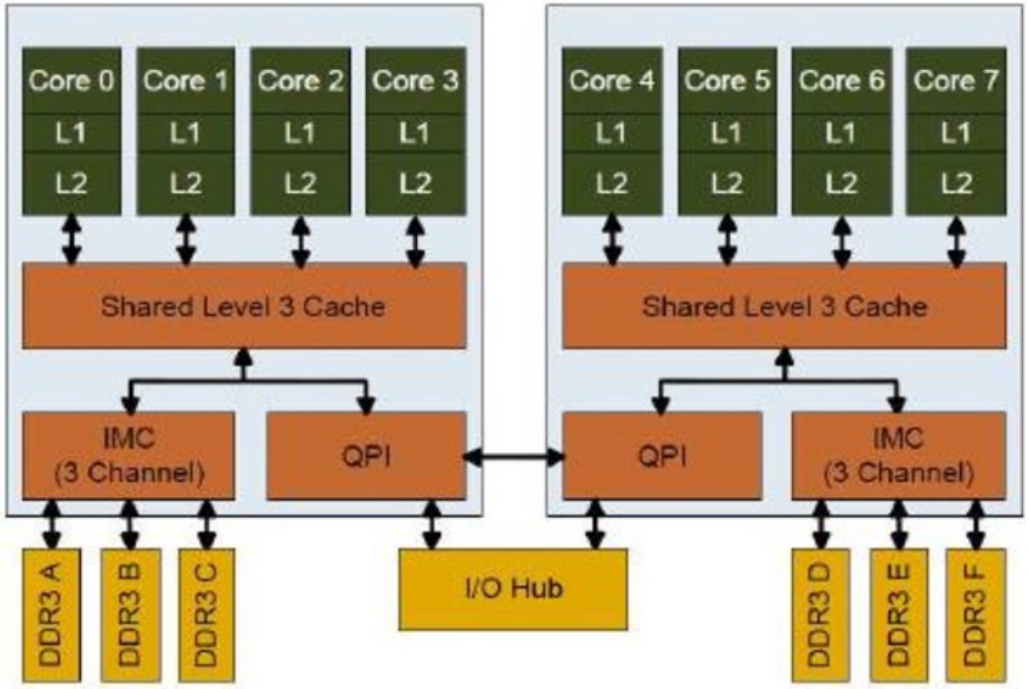


Fig. 2-1. Eight-core Processor cache hierarchy[1]

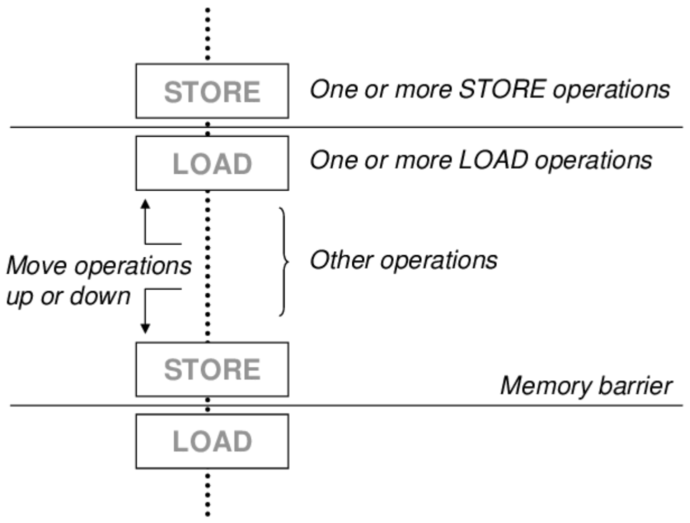
https://www.semanticscholar.org/paper/Multi-Core-Processor-Cache-Hierarchy-Design-Kumari-Charles/2d16258444e9935c8d56c855b17a2c8d2c212745

Memory model (cont.)

- Memory models may be either:
  - **Strongly ordered**
    - Memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered**
    - Memory modification of one processor may not be immediately visible to all other processors.

Memory Barrier (cont.)

- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other ***processors***.



STORE
One or more STORE operations
LOAD
One or more LOAD operations
Move operations up or down
Other operations
STORE
Memory barrier
LOAD

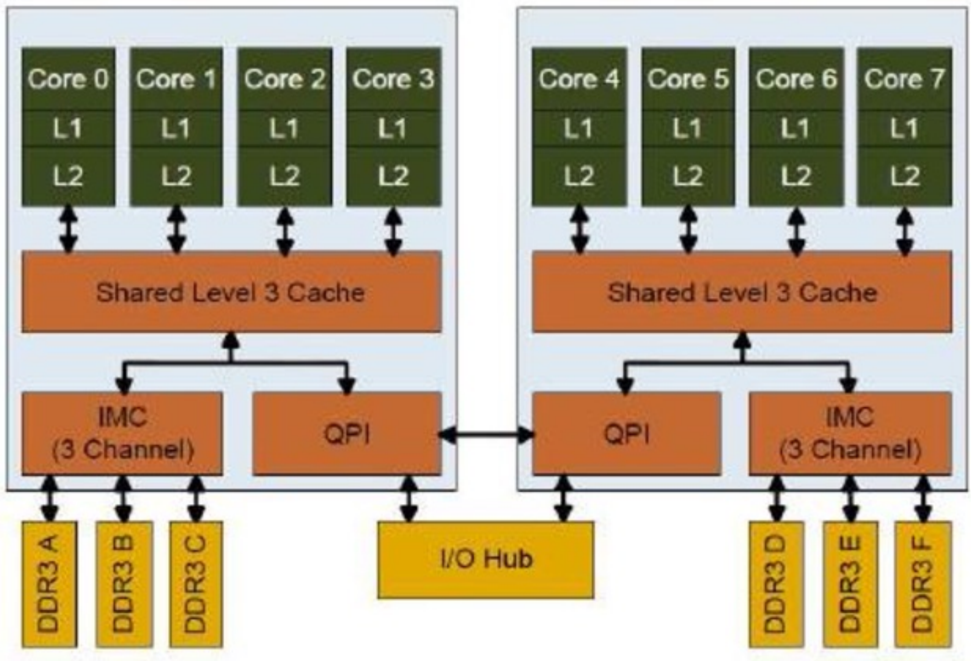


Fig. 2-1. Eight-core Processor cache hierarchy[1]

Memory Barrier Instructions

- **When a memory barrier instruction is performed**, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.

- Therefore, **even if instructions were reordered**, the memory barrier ensures that the store operations are completed in memory and visible to other ***processors*** before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 5-6

- We could add a memory barrier (as follows) to ensure Thread 1 outputs 100.

Memory Barrier Example (cont.)

- Thread 1 now performs

```
while (!flag)
memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 → the value of flag is loaded before the value of x.

- For Thread 2 → the assignment to x occurs **before the assignment flag**.

Memory Barrier for Peterson's solution

Where should we add memory barrier?

```
//P0
1. while (true){
2.   flag[0] = true;
3.   turn = 1;
4.    while (flag[1] && turn == 1);
    /* critical section */
5.    flag[0] = false;
    /* remainder section */
}
```

```
//P1
1. while (true){
2. flag[1] = true;
3. turn = 0;
4.    while (flag[0] && turn == 0);
    /* critical section */
5.    flag[1] = false;
    /* remainder section */
}
```

Memory Barrier for Peterson’s solution (Cont.)

We could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in the previous slide.

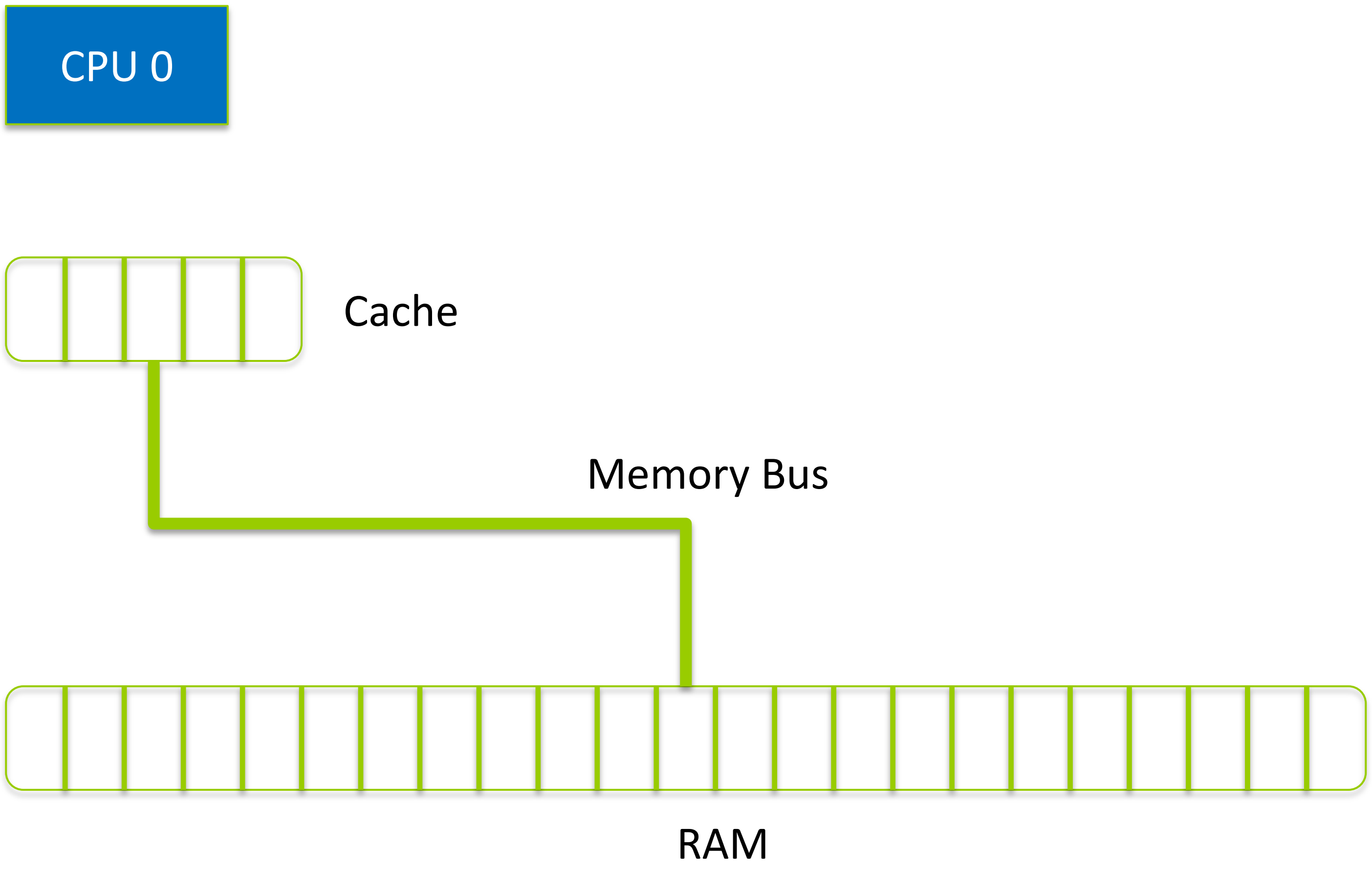
Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

Slides from Don Porter’s Course

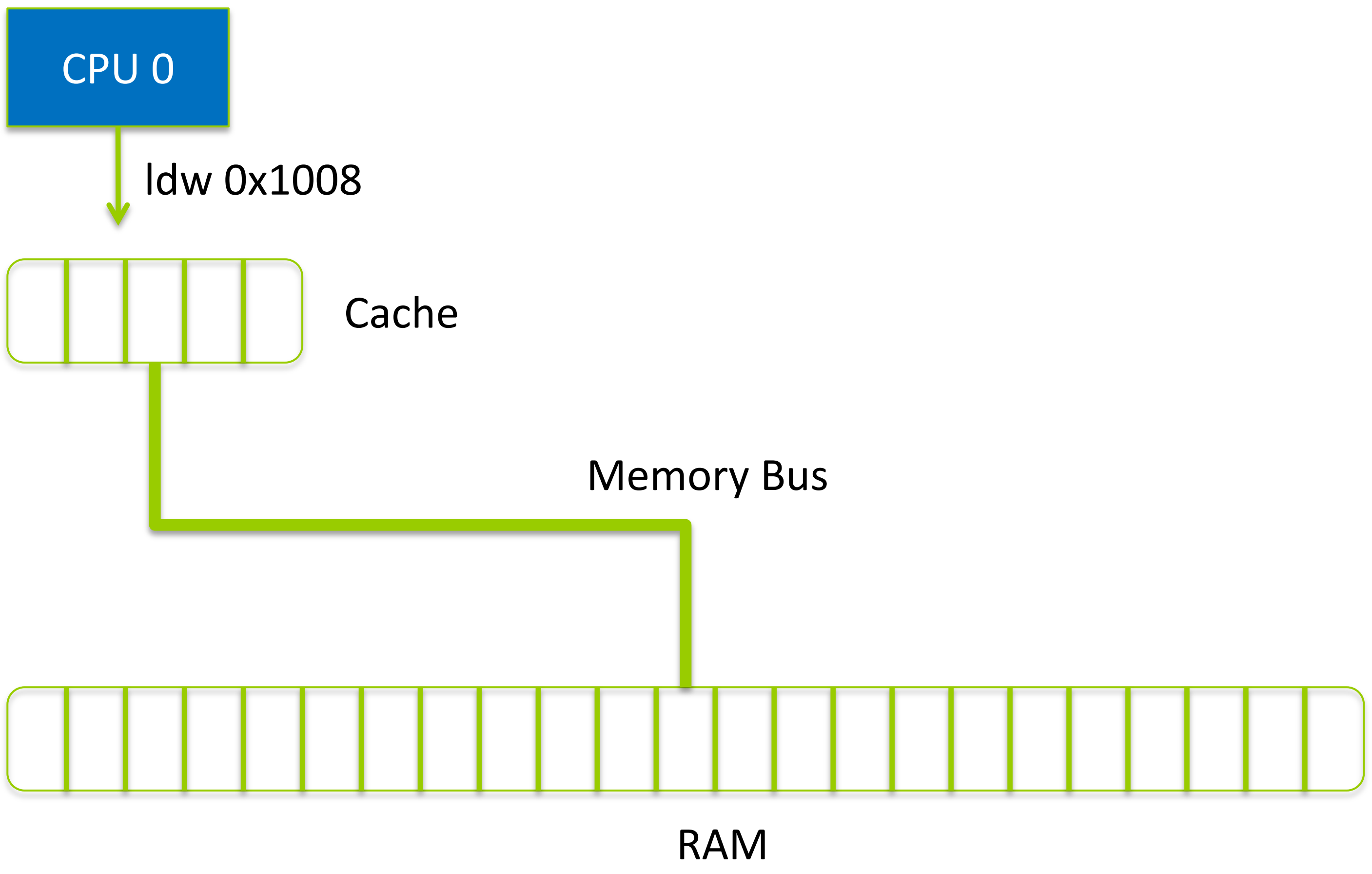
Cache line alignment

- Lines are the basic unit at which memory is cached
- Cache lines are bigger than words
  - Word: 32-bits or 64-bits
  - Cache line – 64—128 bytes on most CPUs

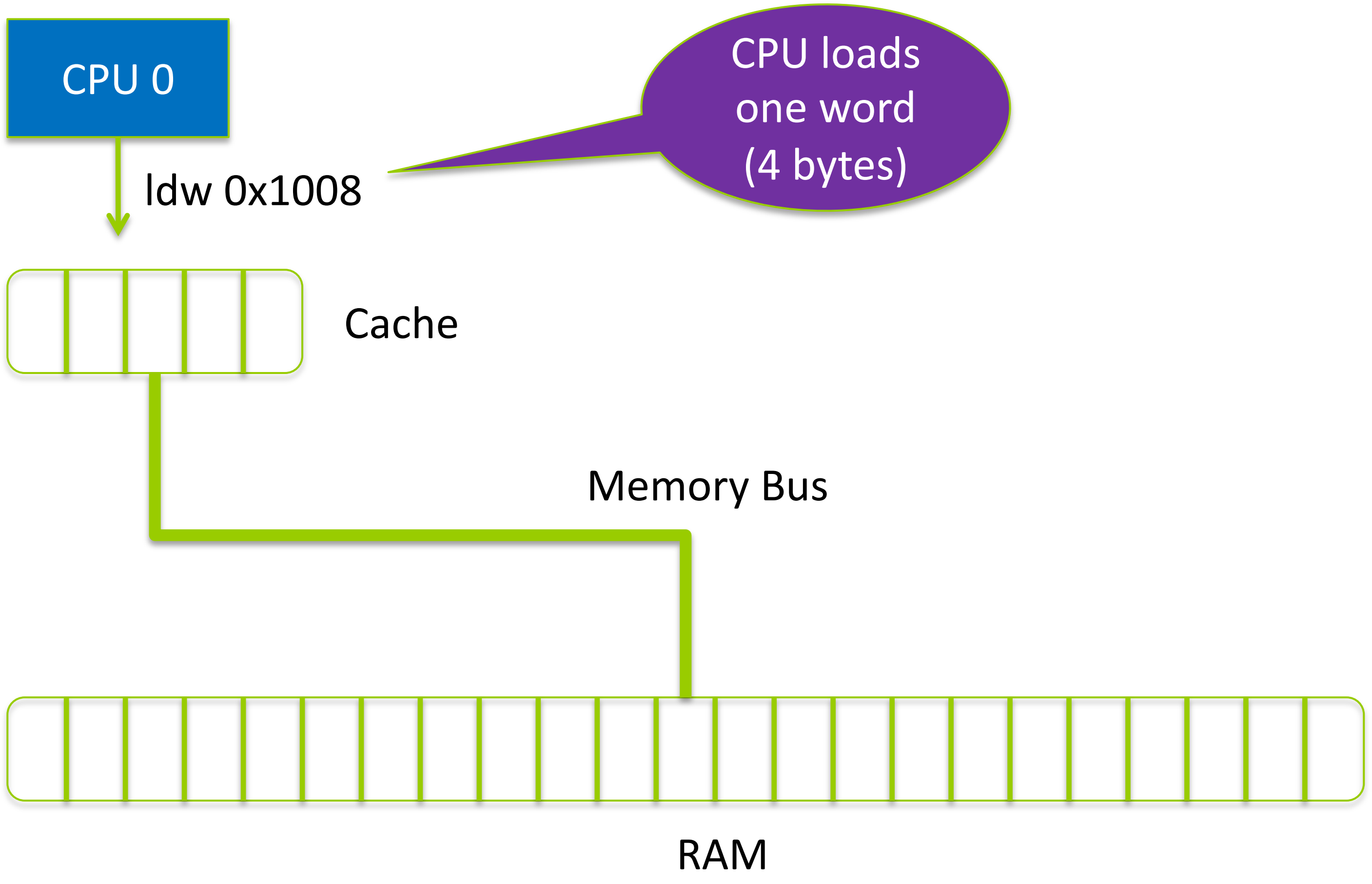
Undergrad Architecture Review



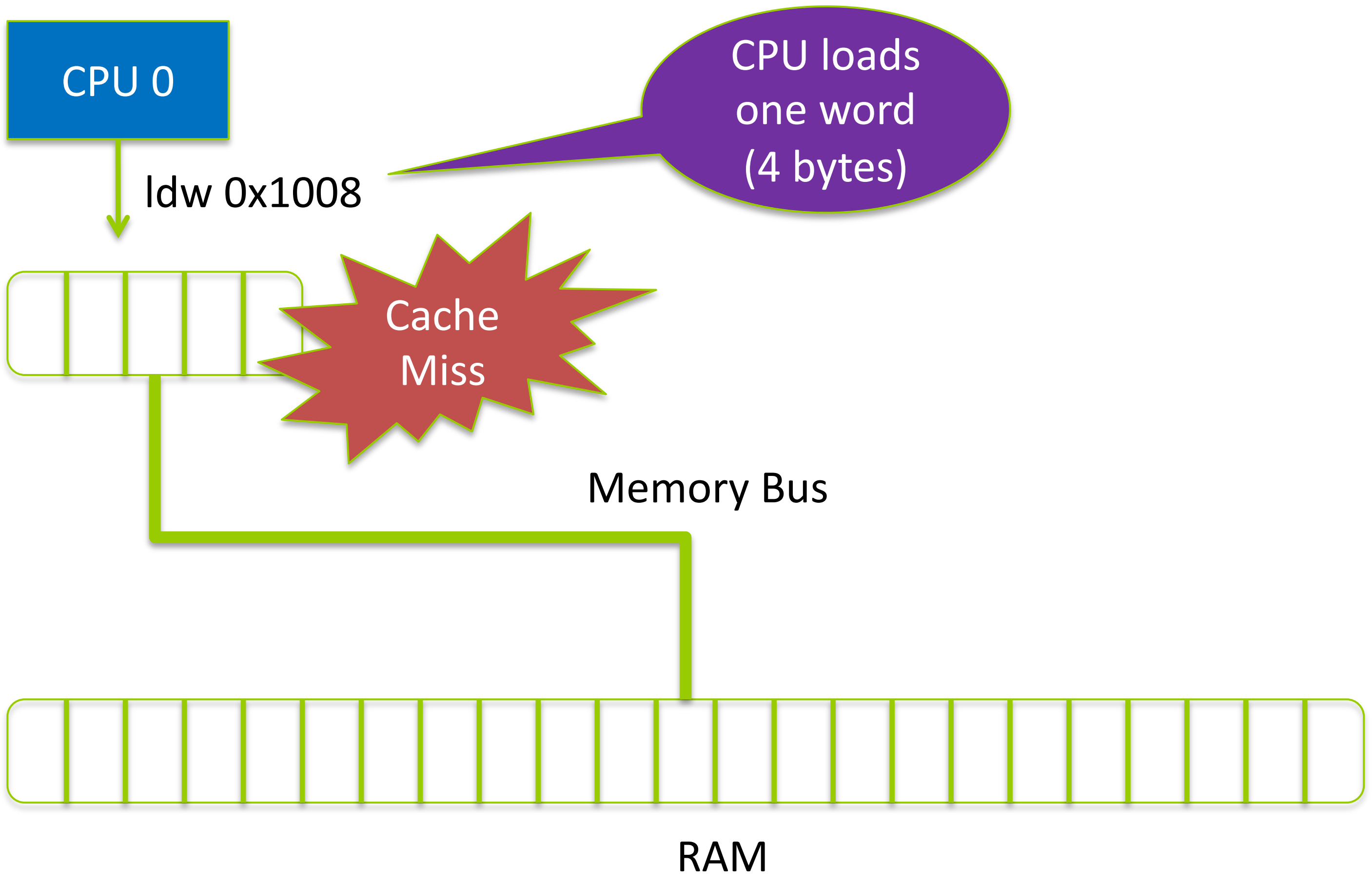
Undergrad Architecture Review



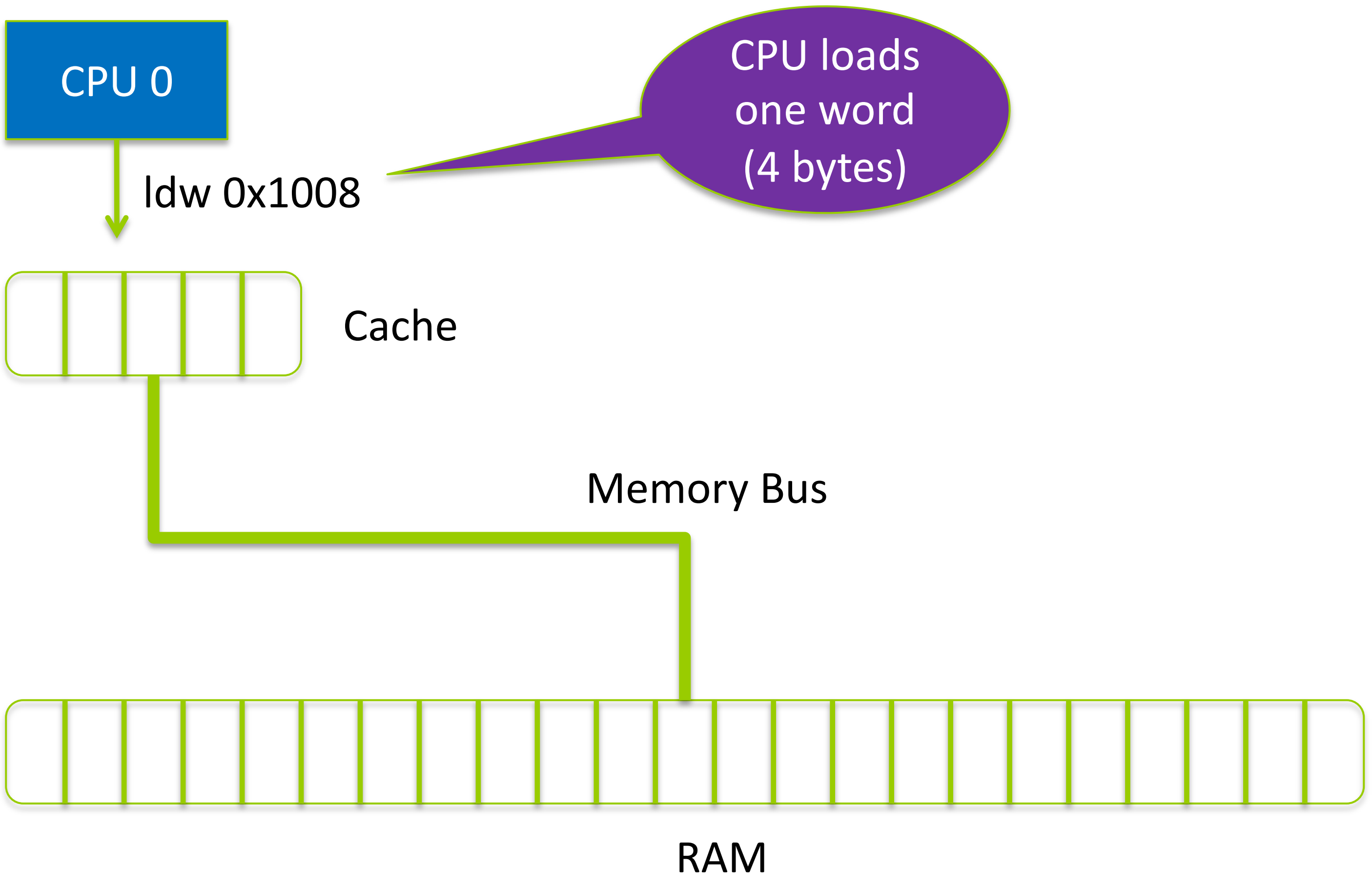
Undergrad Architecture Review



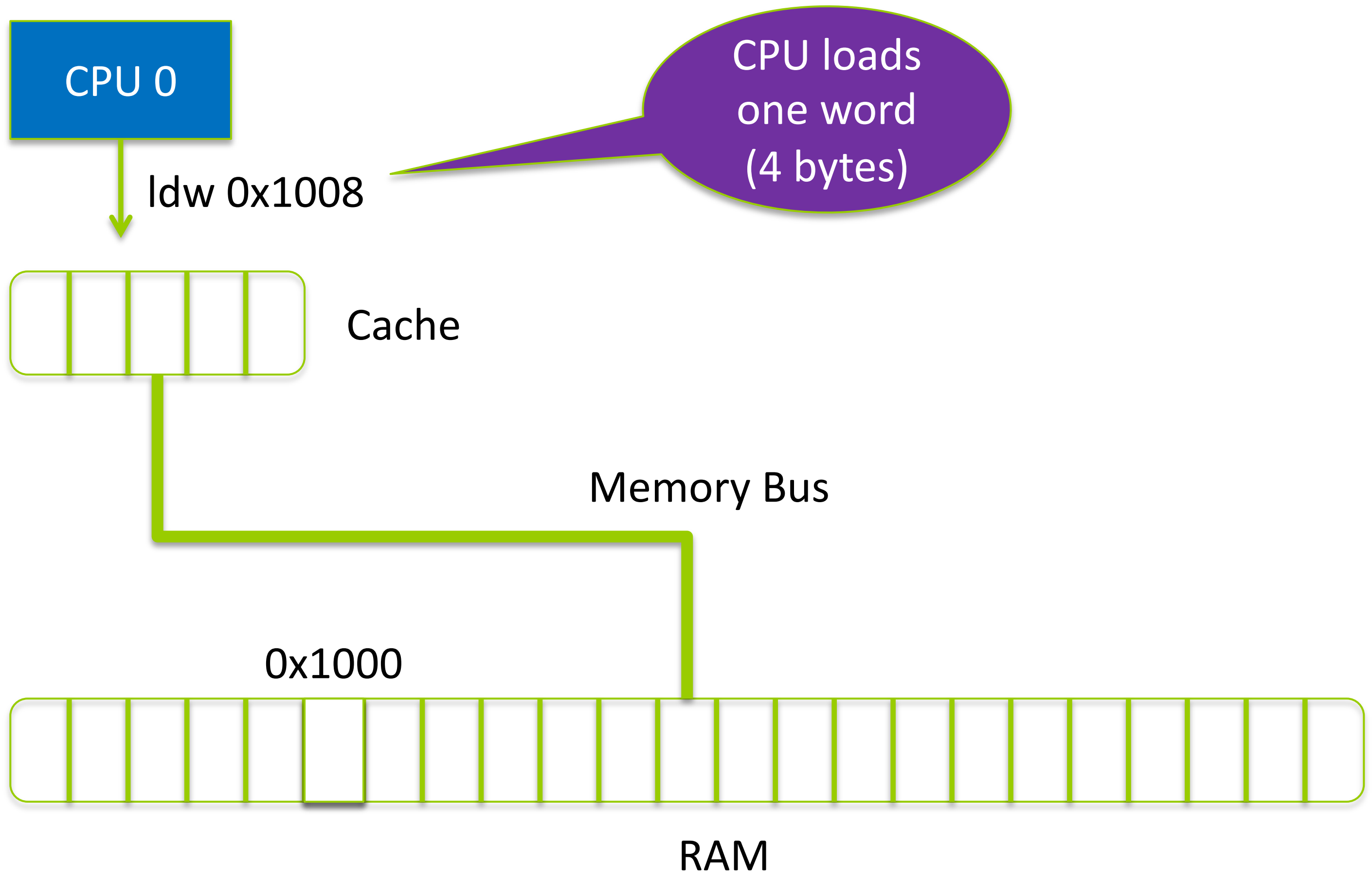
Undergrad Architecture Review



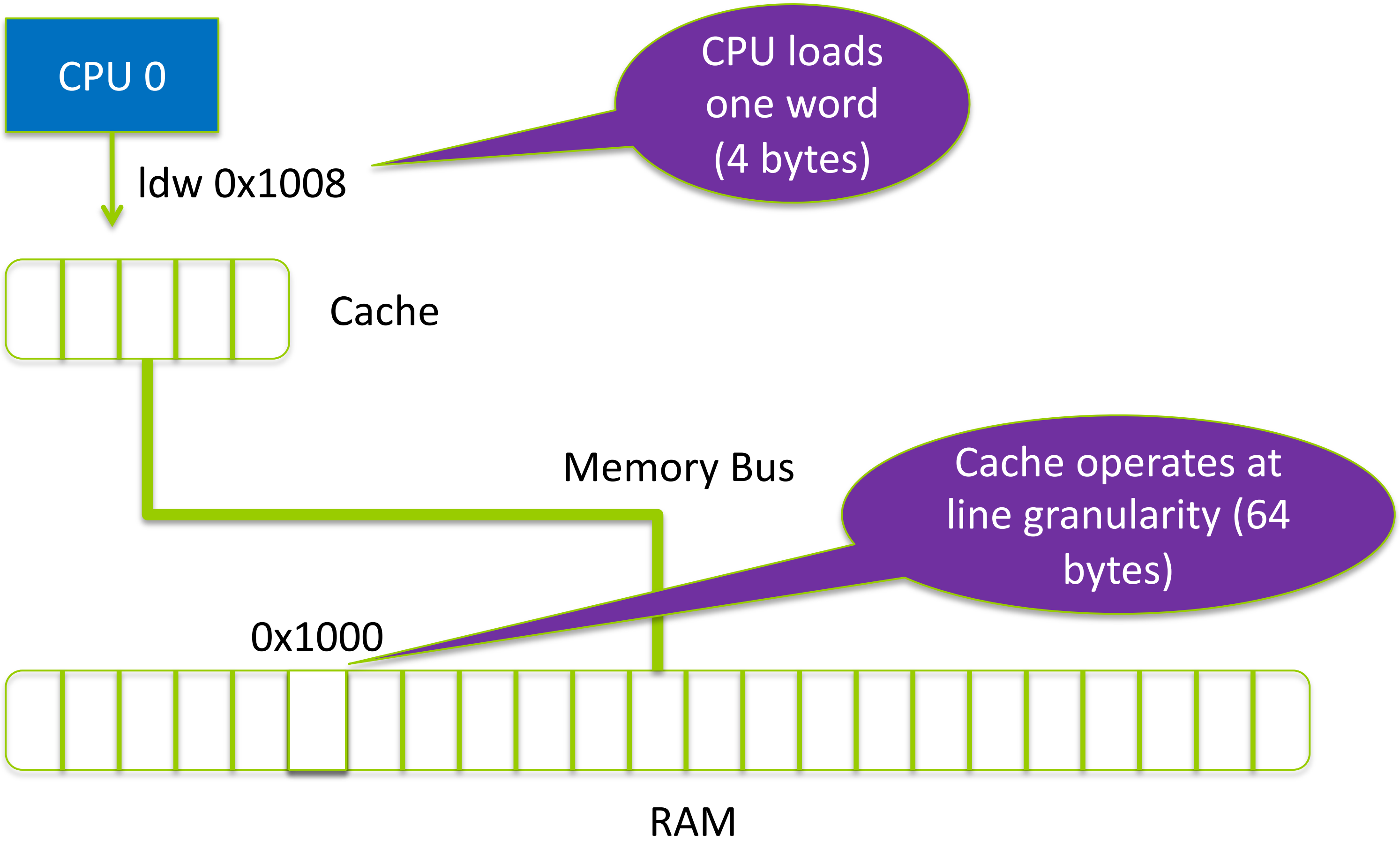
Undergrad Architecture Review



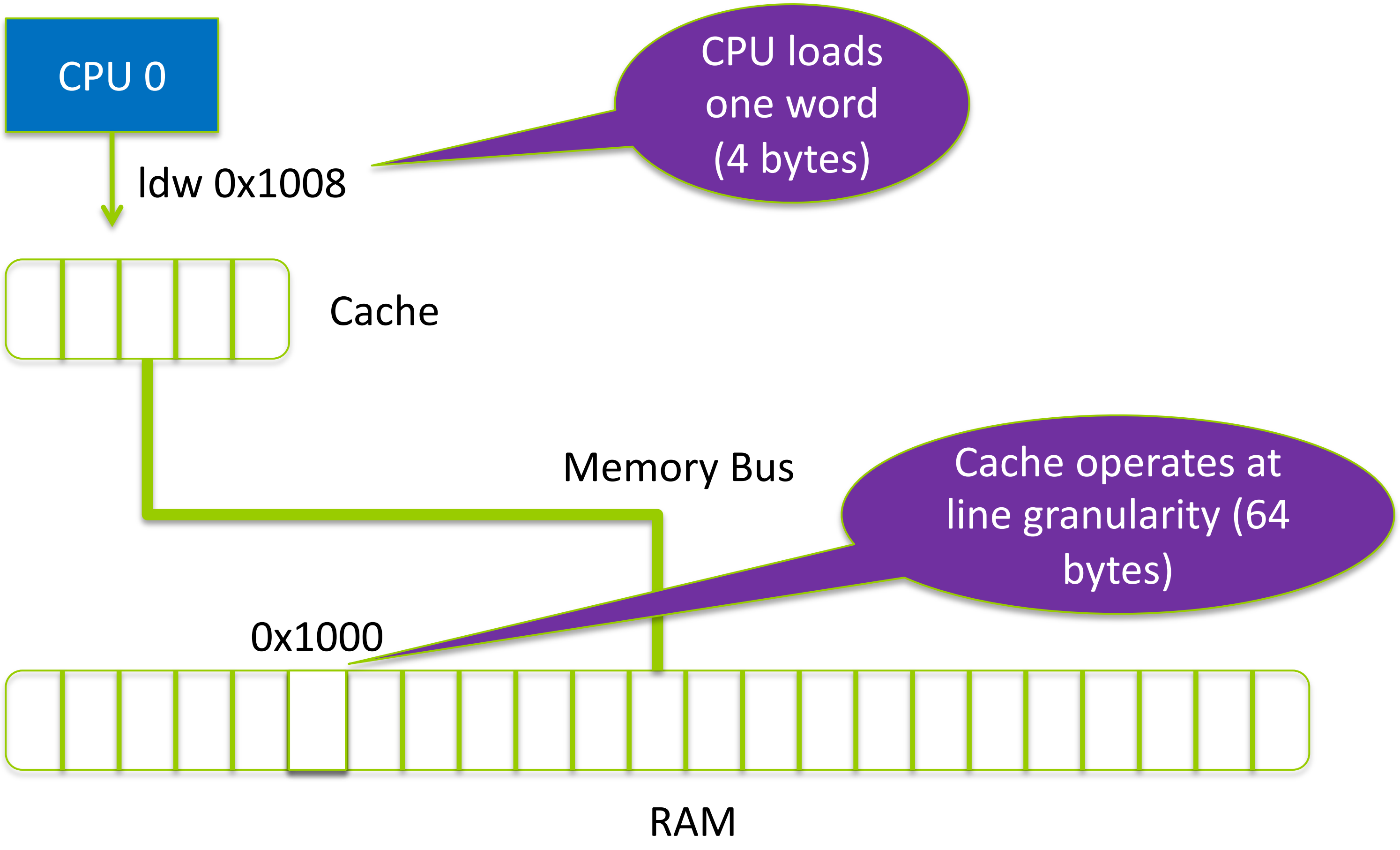
Undergrad Architecture Review



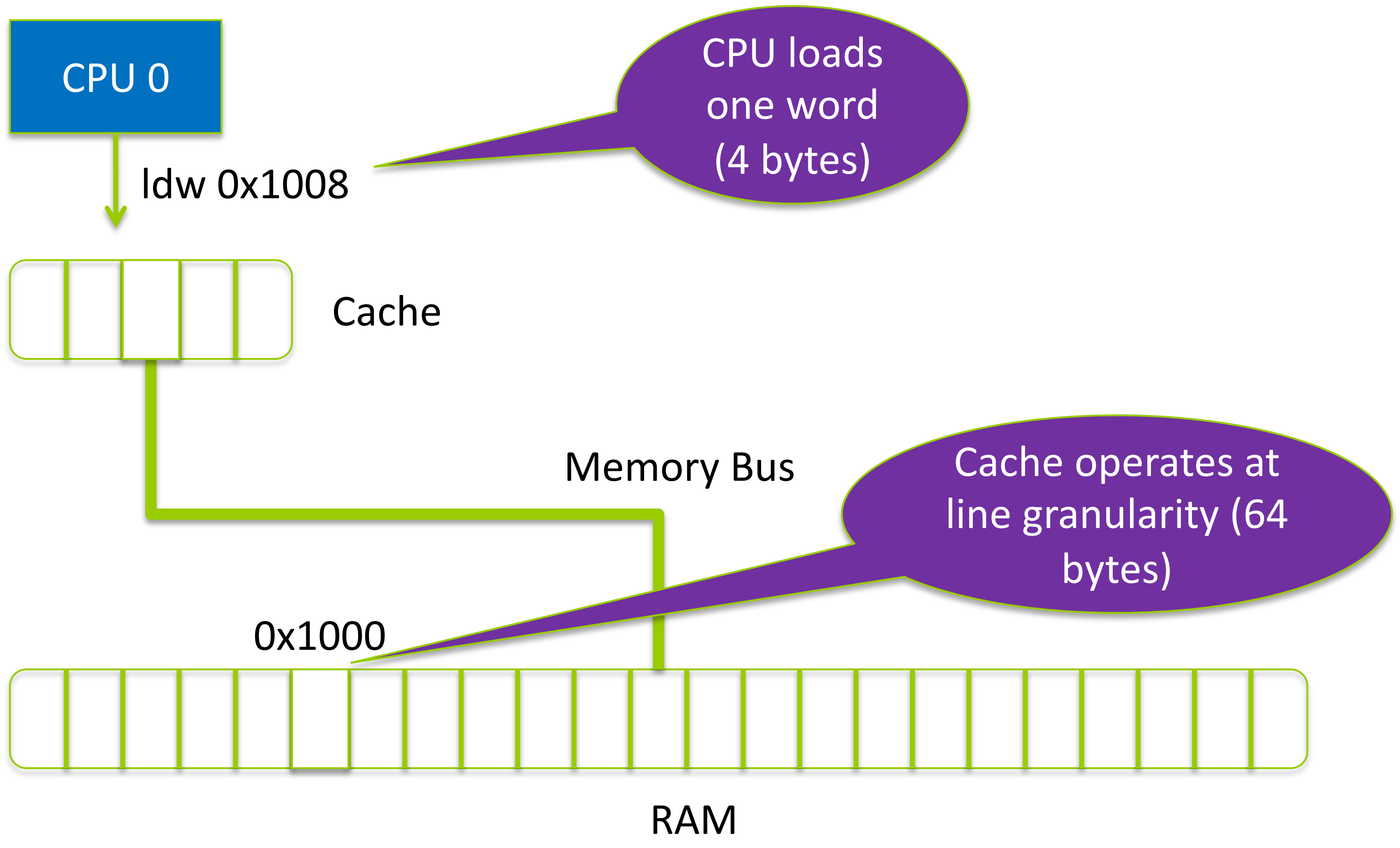
Undergrad Architecture Review


CPU 0
ldw 0x1008
CPU loads one word (4 bytes)
Cache
Memory Bus
Cache operates at line granularity (64 bytes)
0x1000
RAM

Undergrad Architecture Review

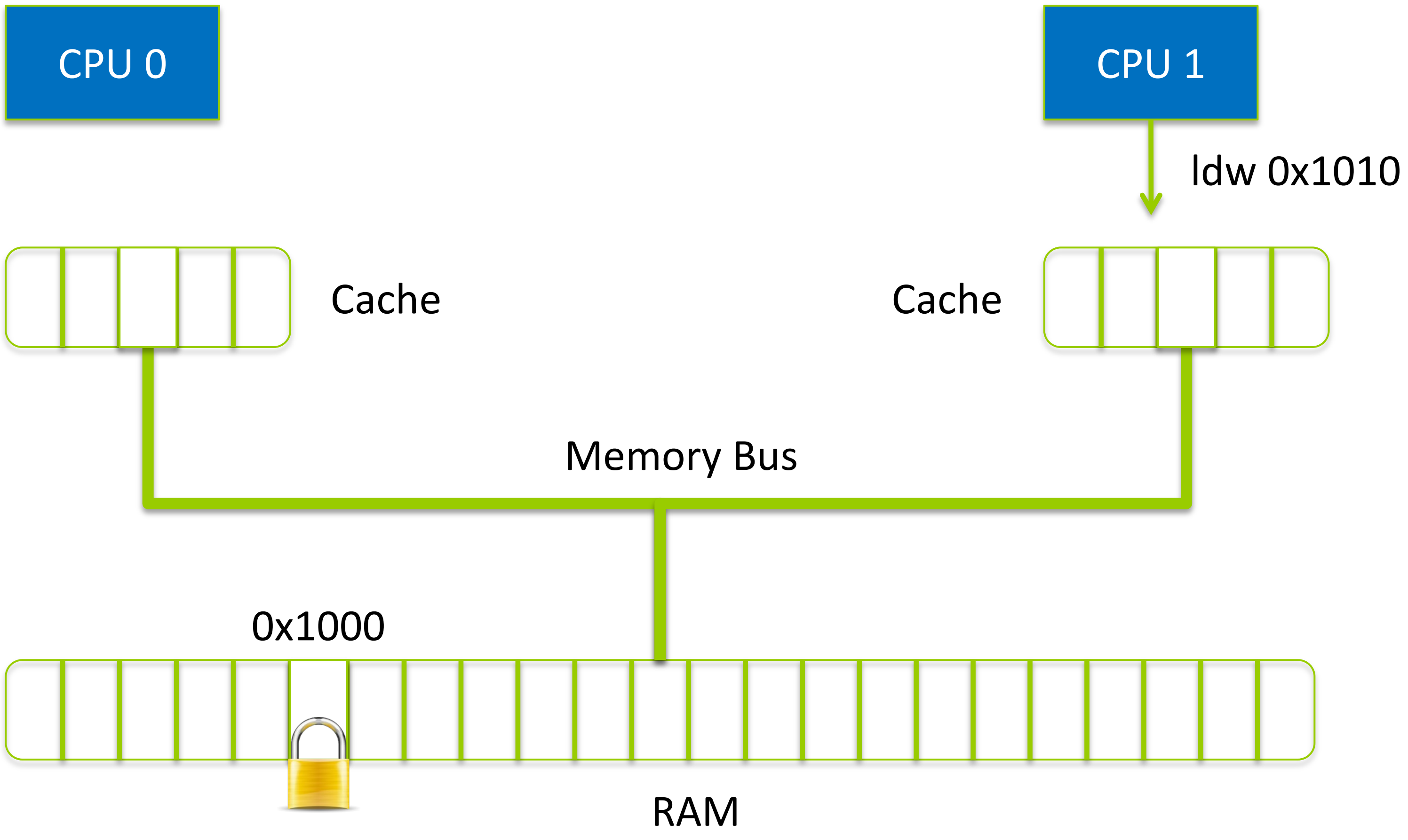

CPU 0
ldw 0x1008
CPU loads one word (4 bytes)
Cache
Memory Bus
Cache operates at line granularity (64 bytes)
0x1000
RAM

Undergrad Architecture Review


CPU 0
ldw 0x1008
CPU loads one word (4 bytes)
Cache
Memory Bus
Cache operates at line granularity (64 bytes)
0x1000
RAM

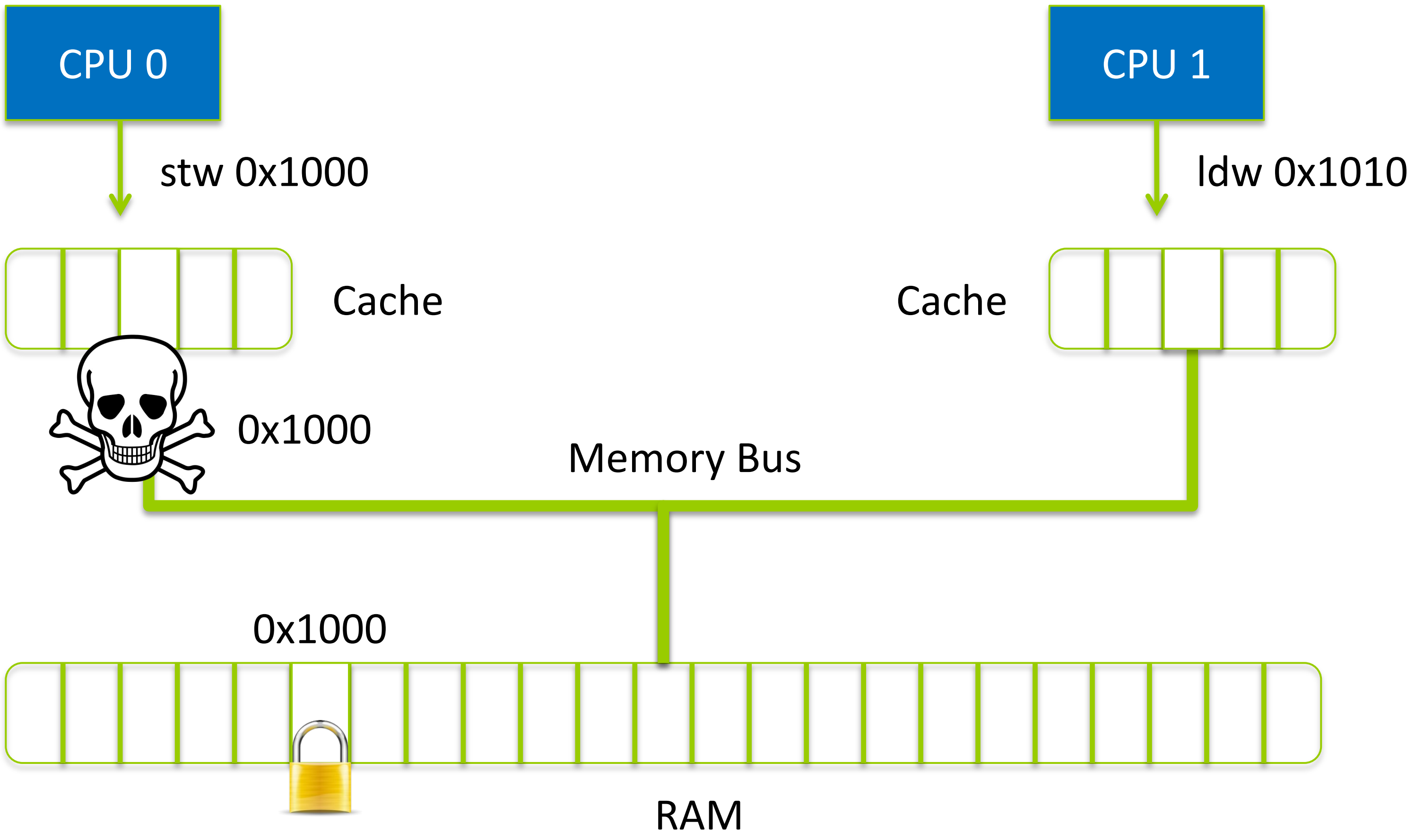
Cache Coherence (1)

Lines shared for reading have a ***shared lock***

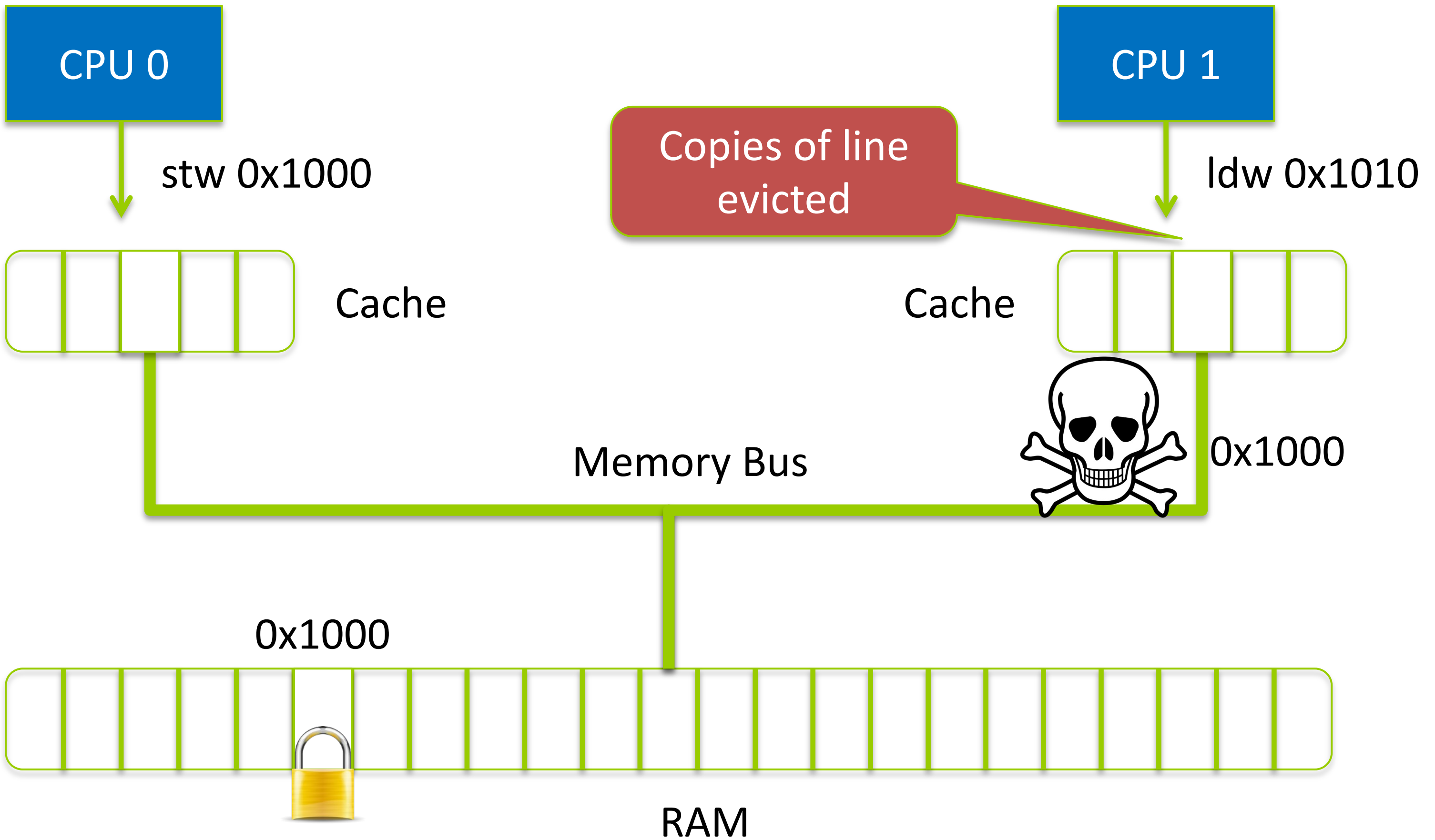


Cache Coherence (2)

Lines to be written have an ***exclusive lock***



Cache Coherence (2)



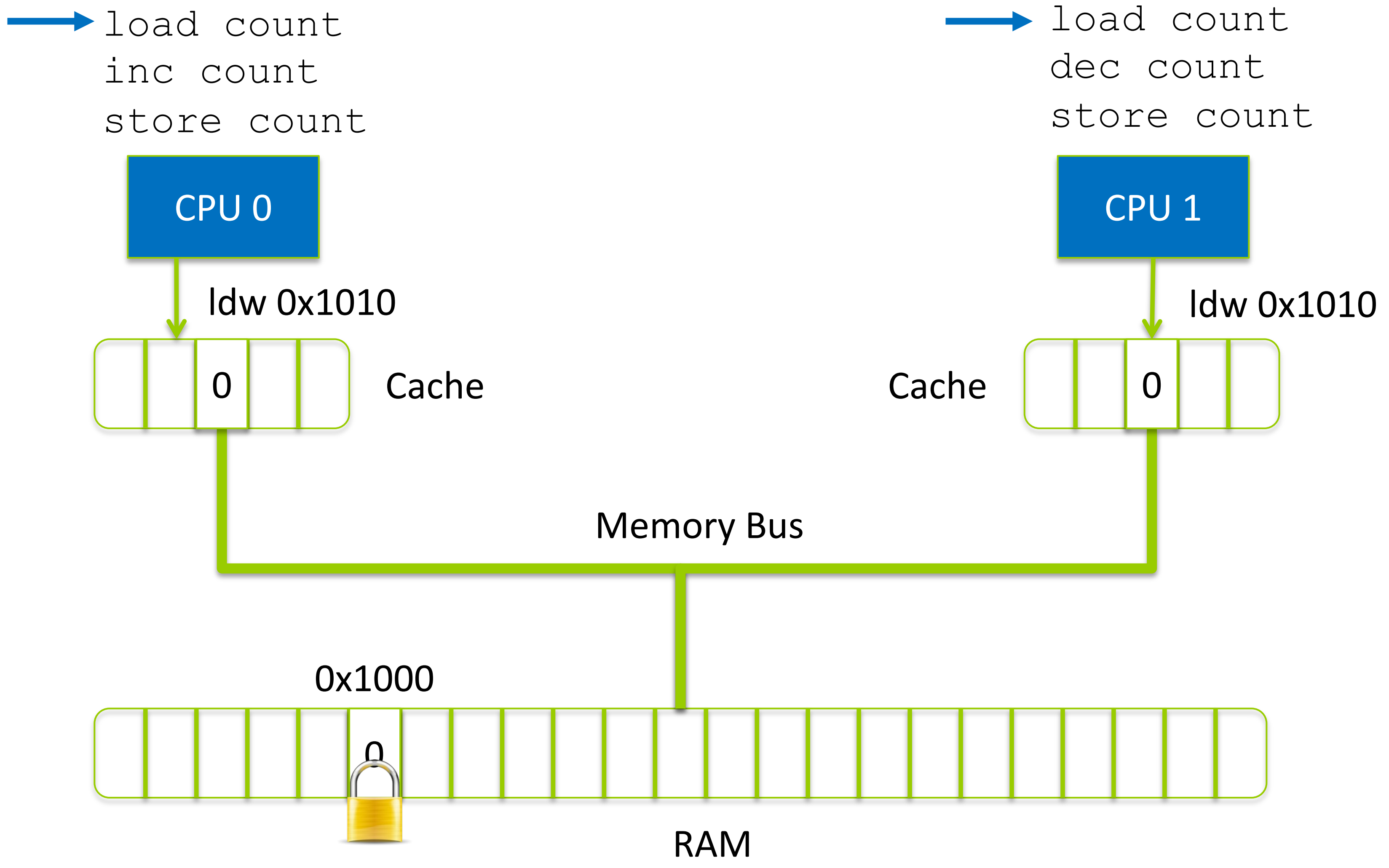
Simple coherence model

- When a memory region is cached, CPU automatically acquires a reader-writer lock on that region
  - Multiple CPUs can share a read lock
  - Write lock is exclusive

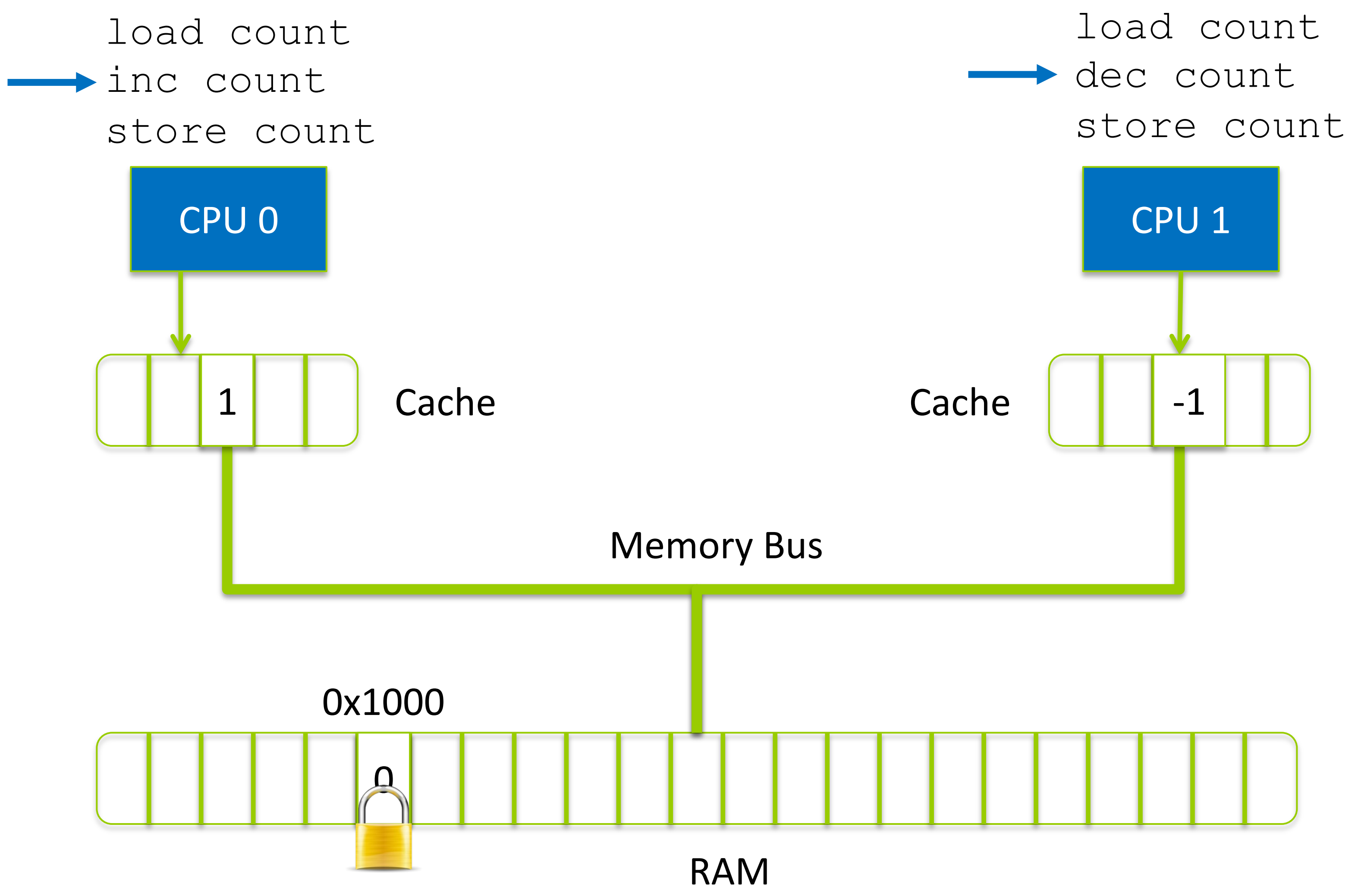
- Programmer can’t control how long these locks are held
  - Ex: a store from a register holds the write lock long enough to perform the write; held from there until the next CPU wants it

How Does Race Condition Happen?

Lines shared for reading have a ***shared lock***

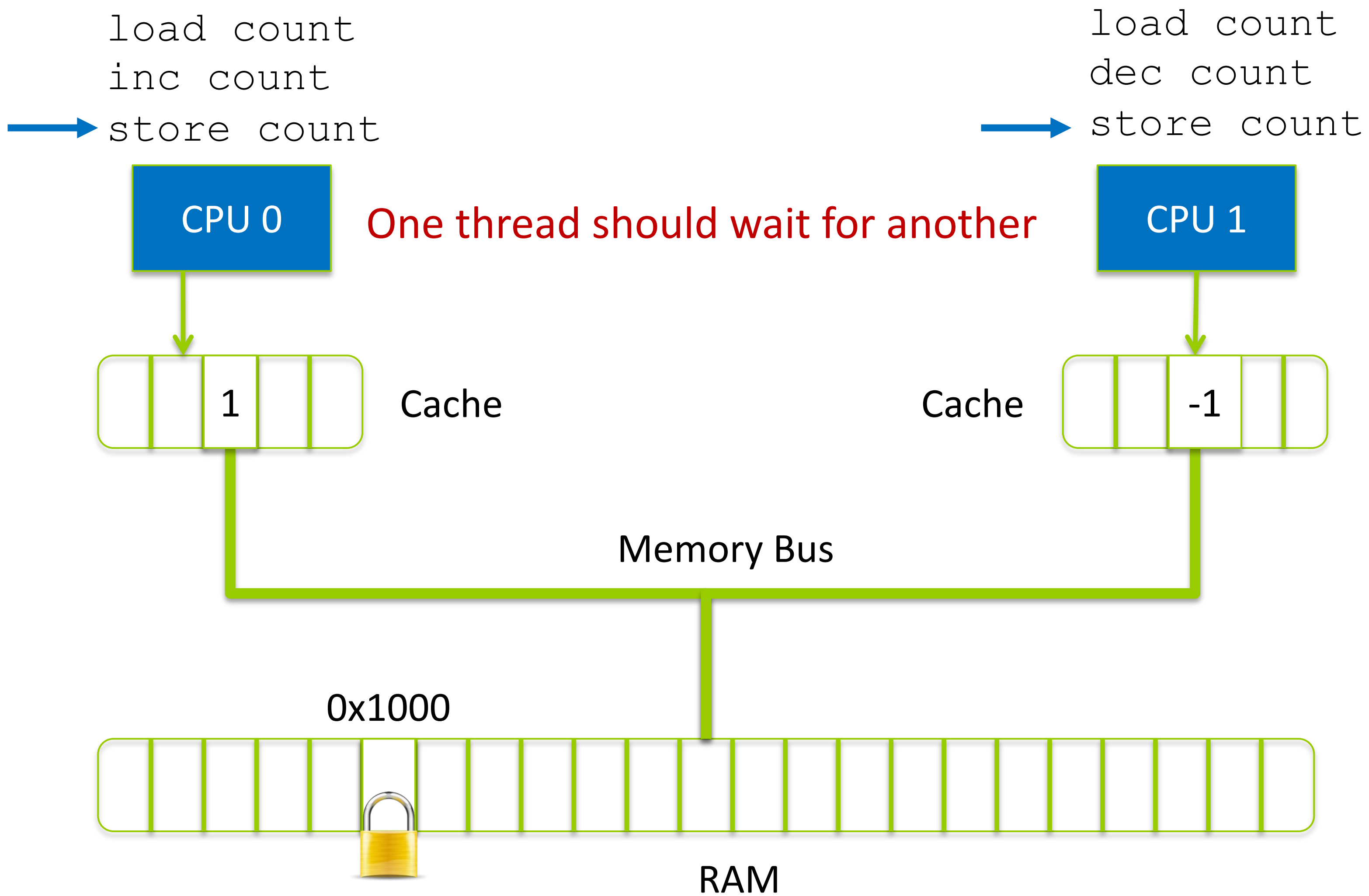


How Does Race Condition Happen?



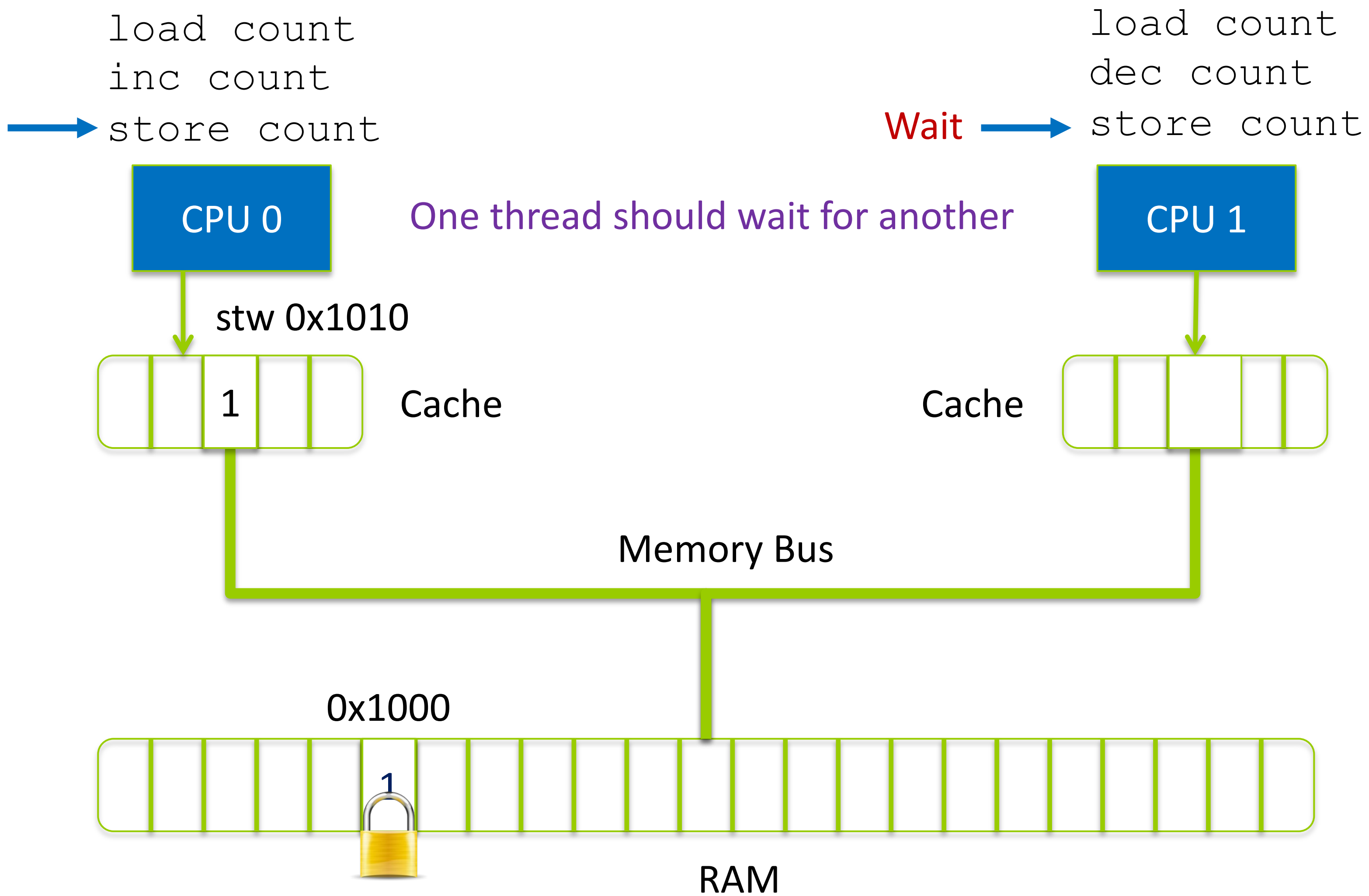
How Does Race Condition Happen?

Lines to be written have an ***exclusive lock***



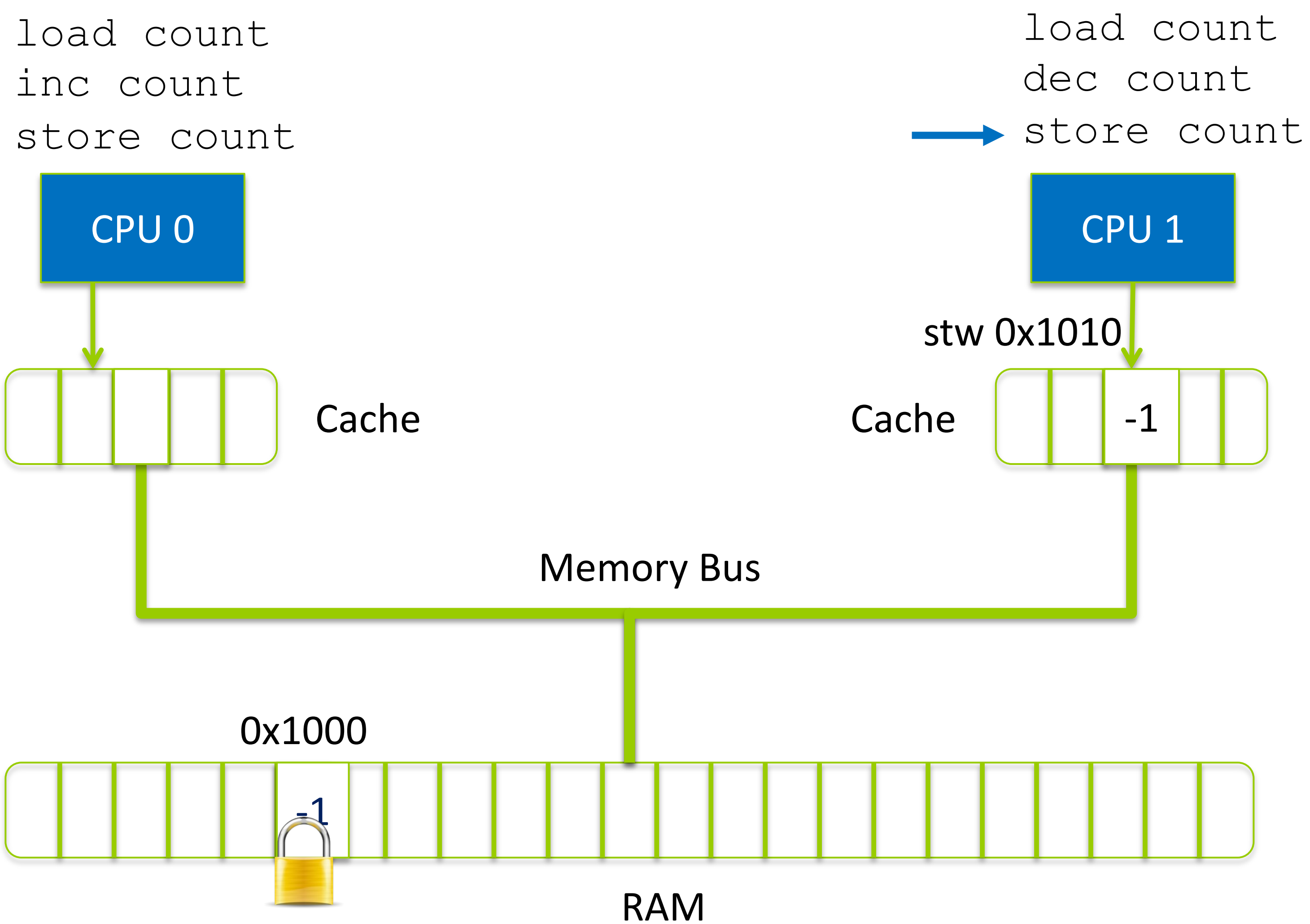
How Does Race Condition Happen?

Lines to be written have an ***exclusive lock***



How Does Race Condition Happen?

Lines to be written have an ***exclusive lock***



Back to HW supports for CS problem

Synchronization Hardware Support

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally, too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable.
- We will look at two forms of hardware support:

1. **Hardware instructions**
2. **Atomic variables**

Hardware Instructions

- Special hardware instructions that allow us to either ***test-and-modify*** the content of a word, or two ***swap*** **the contents of two words atomically** (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;
        return rv:
 }
```

- Properties
  - **Executed atomically**
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution Using test_and_set()

- Shared boolean variable **lock**, initialized to **false**

```
do{
 while (test_and_set(&lock)); /*do nothing */
      /* critical section */
      lock = false;

      /* remainder section */

} while (true);
```

- **Does it solve the critical-section problem?**

Requirement	Yes/No
Mutual Exclusion	
Progress	
Bounded waiting	

The compare_and_swap Instruction

- Definition

```
Int compare_and_swap(int *value, int expected, int new_value)
{
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

- Properties
  - Executed **atomically**
  - Returns the original value of passed parameter value
  - Set the variable value the value of the passed parameter new_value but only if *value == expected is true.

Solution using compare_and_swap

- Shared integer `lock` initialized to 0;

```
while (true){
   while(compare_and_swap(&lock, 0, 1)!= 0);/*do nothing*/
    /* critical section */
     lock = 0;
       /* remainder section */
 }
```

- **Does it solve the critical-section problem?**

Requirement	Yes/No
Mutual Exclusion	
Progress	
Bounded waiting	

Second solution using compare-and-swap

- The common data structures are:

```
boolean waiting[n];

int lock;
```

- The elements in the ***waiting*** array are initialized to ***false***
- Variable ***lock*** is initialized to ***0***.

Second solution using compare-and-swap

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;
    /* critical section */

    …

}
```

Second solution using compare-and-swap

```
while (true) {
      …
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Requirement	Yes/No?
Mutual Exclusion	
Progress	
Bounded waiting	

Synchronization Hardware Support

- **Hardware instructions**
  - test_and_set()
  - Compare_and_swap()

- **Atomic variables**
  - We unfortunately do not have enough time to cover this
  - Please read the related section in the reference book