

Foundation and Cost of Synchronization

Jinyang Li

Based on the slides of
Tiger Wang

Example

global++



```
mov 0x20072d(%rip),%eax // load global into %eax  
add $0x1,%eax          // update %eax by 1  
mov %eax,0x200724(%rip) // restore global with %eax
```

Example

global++



```
mov 0x20072d(%rip),%eax // load global into %eax  
add $0x1,%eax          // update %eax by 1  
mov %eax,0x200724(%rip) // restore global with %eax
```

But how about?

global++



```
add $0x1, 0x20072d(%rip) // directly update global by 1
```

add is not atomic

global++



```
add $0x1, 0x20072d(%rip) // directly update global by 1
```

CPU

1. Load the data to CPU's local buffer (invisible)
2. Calculate the result
3. Store the data back to memory

atomic instruction

Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.

atomic instruction

Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.

CPU

Lock the memory address

1. Load the data to CPU's local buffer (invisible)
2. Calculate the result
3. Store the data back to memory

Unlock the memory address

atomic instruction

Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.

CPU

Lock the memory address

1. Load the data to CPU's local buffer (invisible)
2. Calculate the result
3. Store the data back to memory

Unlock the memory address

Add **lock** prefix to make the instruction be atomic

atomic instructions

atomic instructions
mov
xchg
...

atomic with lock prefix
add / sub
inc / dec
and / or / xorl
cmpxchg
...

reading a memory operand,
performing some operation on it,
write it back to the memory

How to implement a lock?

```
// 1: busy, 0: free  
int mutex = 0;
```

```
void lock(int *mu) {  
    while(*mu == 1) {}  
    *mu = 1;  
}
```



Busy wait
This style of locking is called "Spin Lock"

```
void unlock(int *mu) {  
    *mu = 0;  
}
```

How to implement a lock?

```
int mutex = 0;
```

thread-1

```
void lock(int *mu) {  
    while(*mu == 1) {}  
  
    *mu = 1;  
}
```

thread-2

```
void lock(int *mu) {  
  
    while(*mu == 1) {}  
    *mu = 1;  
}
```

xchg instruction

xchg op1, op2

- Swap the op1 operand with the op2 operand

xchg *reg*, *reg*

xchg *reg*, *mem*

xchg *mem*, *reg*

xchg instruction

xchg op1, op2

- Swap the op1 operand with the op2 operand

xchg *reg*, *reg*

xchg *reg*, *mem*

xchg *mem*, *reg*

```
int xchg(int *ptr, int x)
{
    asm volatile("xchgl %0,%1"
                 : "=r" (x)
                 : "m" (ptr), "0" (x)
                 : "memory");
    return x;
}
```

Atomically store the memory pointed by ptr with x, then return the old value stored at ptr.

Spin Lock based on xchg

```
// 1: busy, 0: free  
int mutex = 0;
```

```
void lock(int *mu) {  
    while(xchg(mu, 1) != 0) {}  
}
```

```
void unlock(int *mu) {  
    xchg(mu, 0);  
}
```

Thread-1

Thread-2

xchg(mu,1)=0

xchg(mu,1)=?

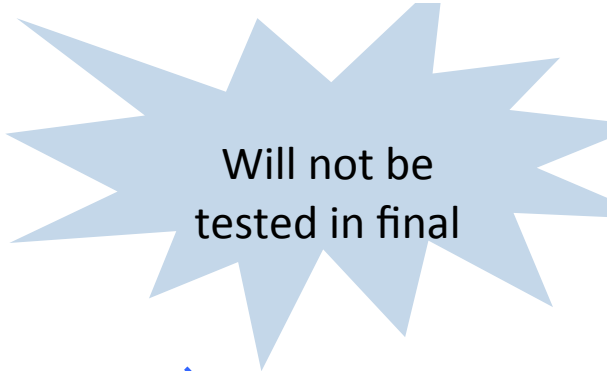


continue to busy wait

Why not always use spin locks?

- If spin lock is not available, thread busy waits (spins)
- Not efficient if critical section is long.
- Better alternative: if one thread blocks, execute another thread that can make progress
 - Need help from OS kernel to put one thread on hold and schedule another.

Futex syscall



Will not be
tested in final

- `futex(int *addr, FUTEX_WAIT, val, ...)`
 - atomically checks `*addr == val` and puts calling thread on OS' wait queue for `addr` if equality holds.
- `futex(int *addr, FUTEX_WAKE, n, ...)`
 - wakes `n` threads on OS' wait queue for `addr`.

A simple pthread_mutex impl.

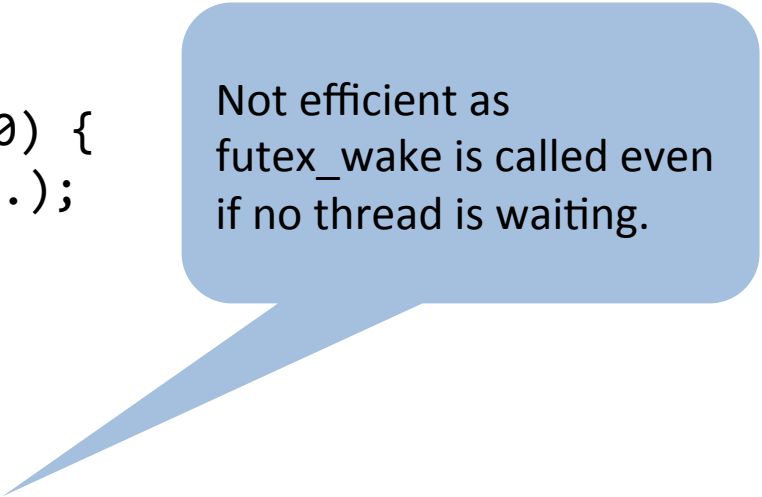
```
typedef struct {
    int locked;
} mutex_t;

void mutex_init(mutex_t *mu) {
    mu->locked = 0;
}

void mutex_lock(mutex_t *mu) {
    while(xchg(&mu->locked, 1) != 0) {
        futex_wait(&mu->locked, 1, ..);
    }
}

void mutex_unlock(mutex_t *mu) {
    xchg(&mu->locked, 0);
    futex_wake(&mu->locked, 1, ..);
}
```

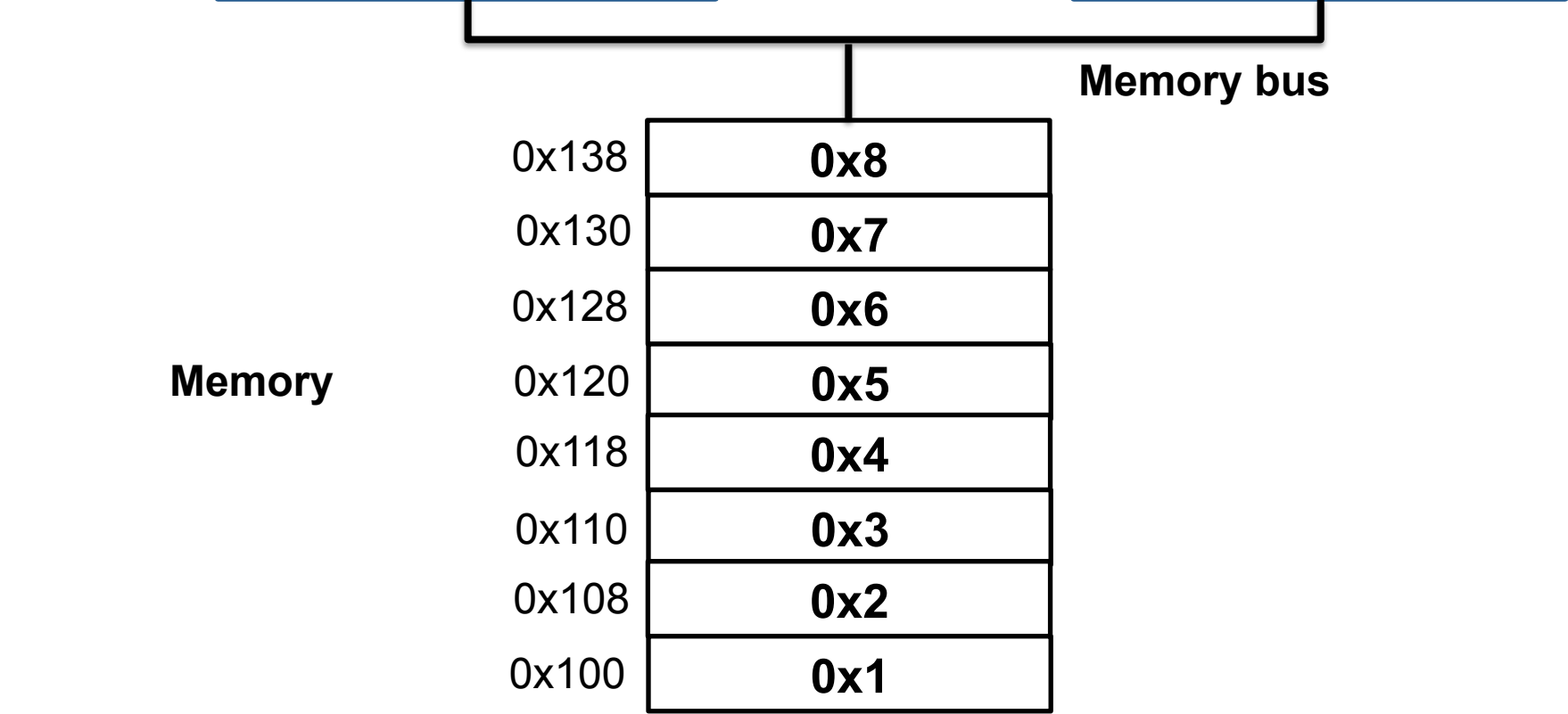
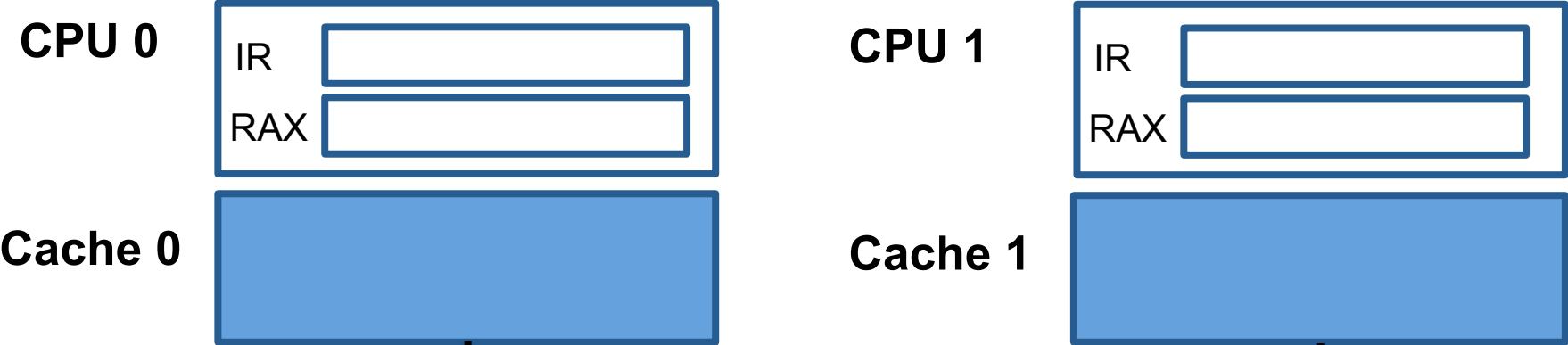
- Actual pthread mutex and conditional variable are much more complex for better performance.
- For more information, google “futexes are tricky” by Ulrich Drepper



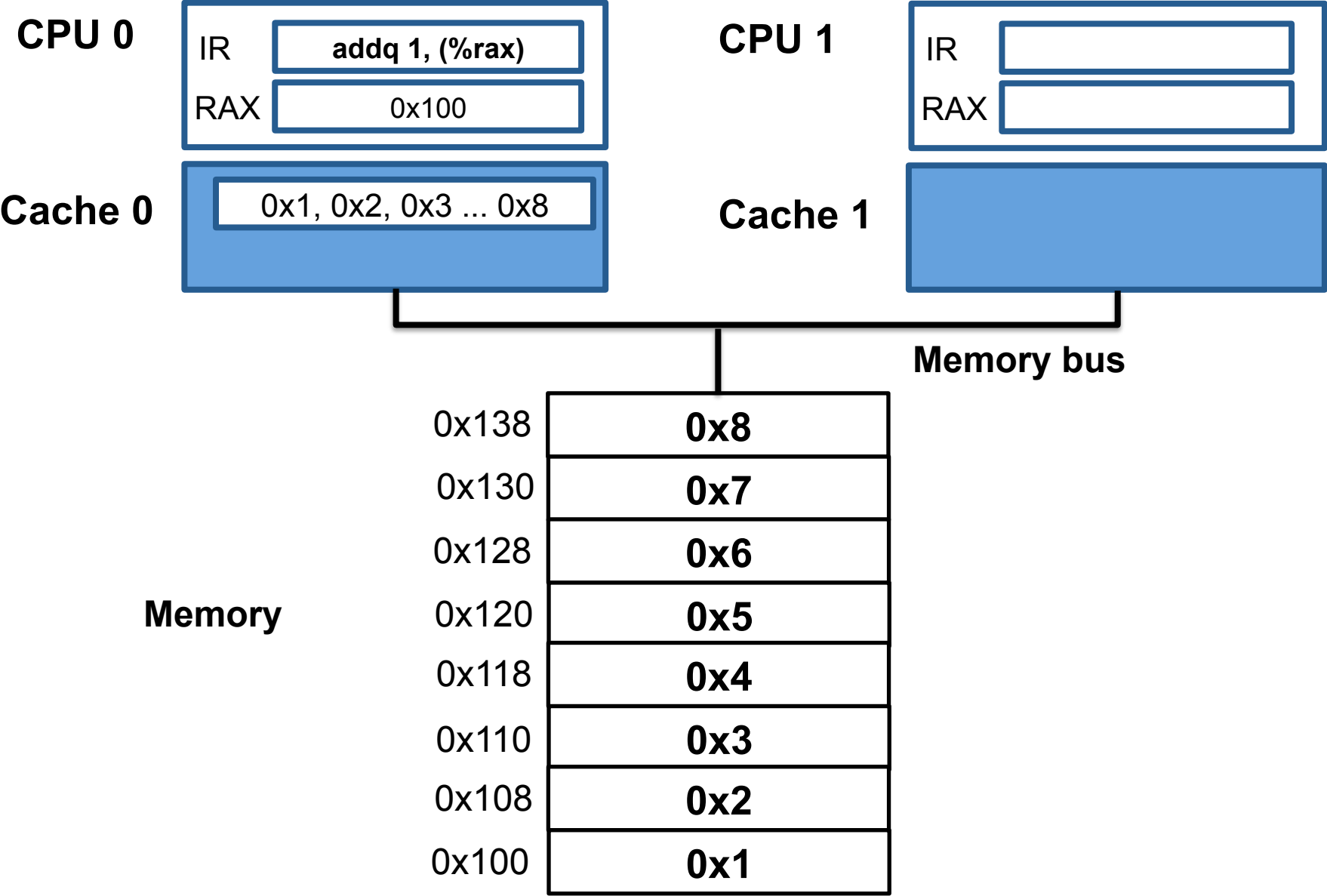
Not efficient as futex_wake is called even if no thread is waiting.

The cost of synchronization

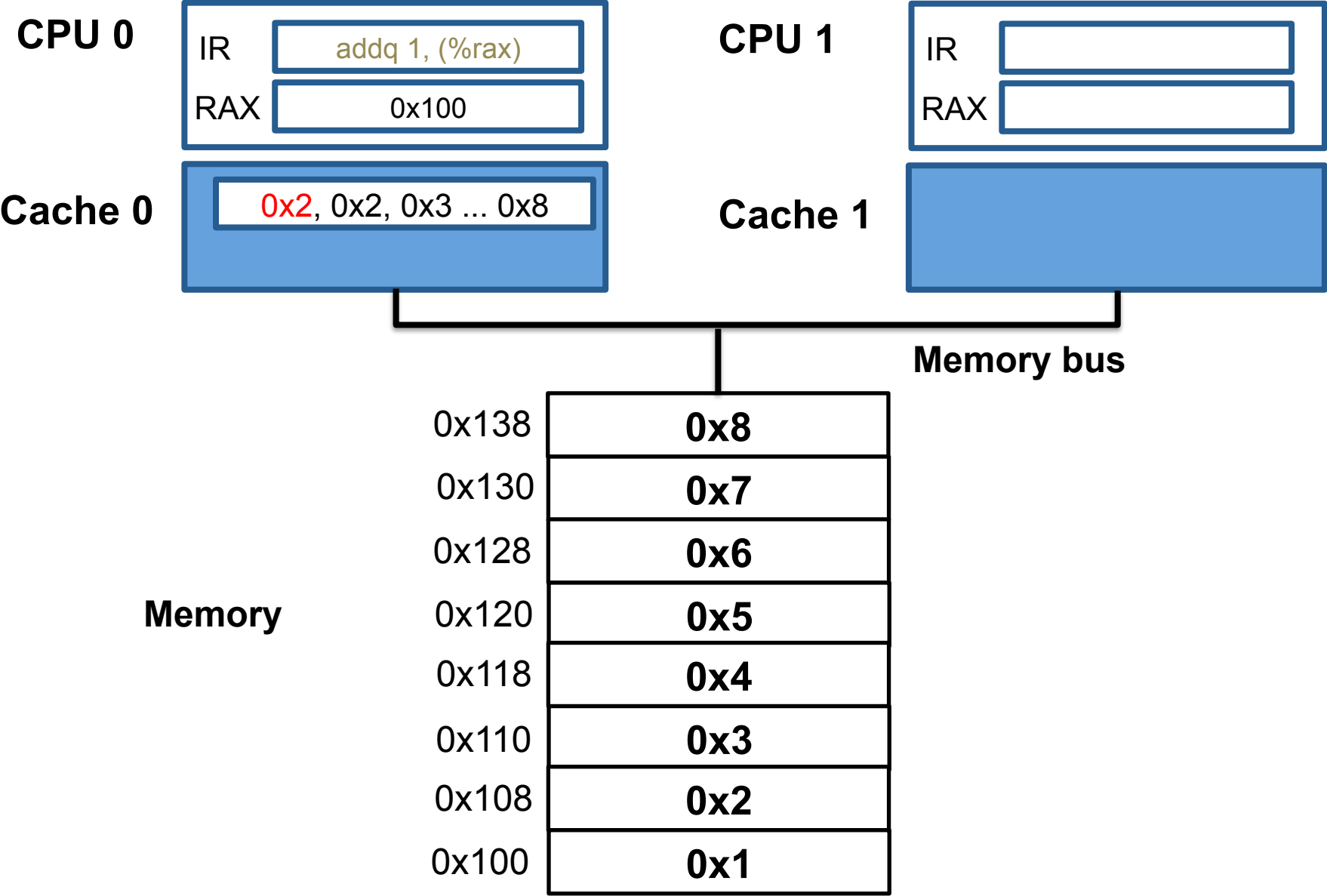
Basic Idea of Cache Coherence



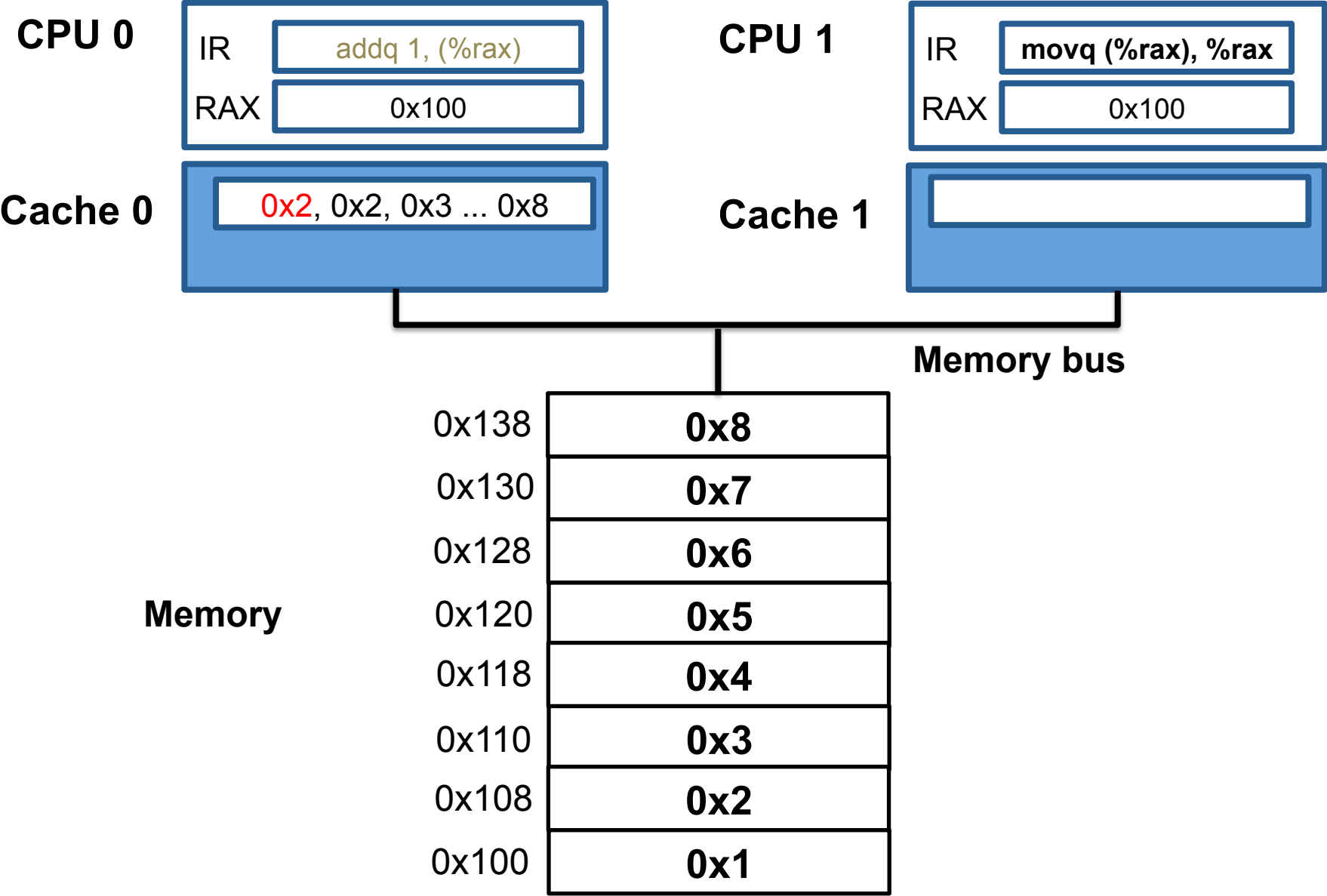
Basic Idea of Cache Coherence



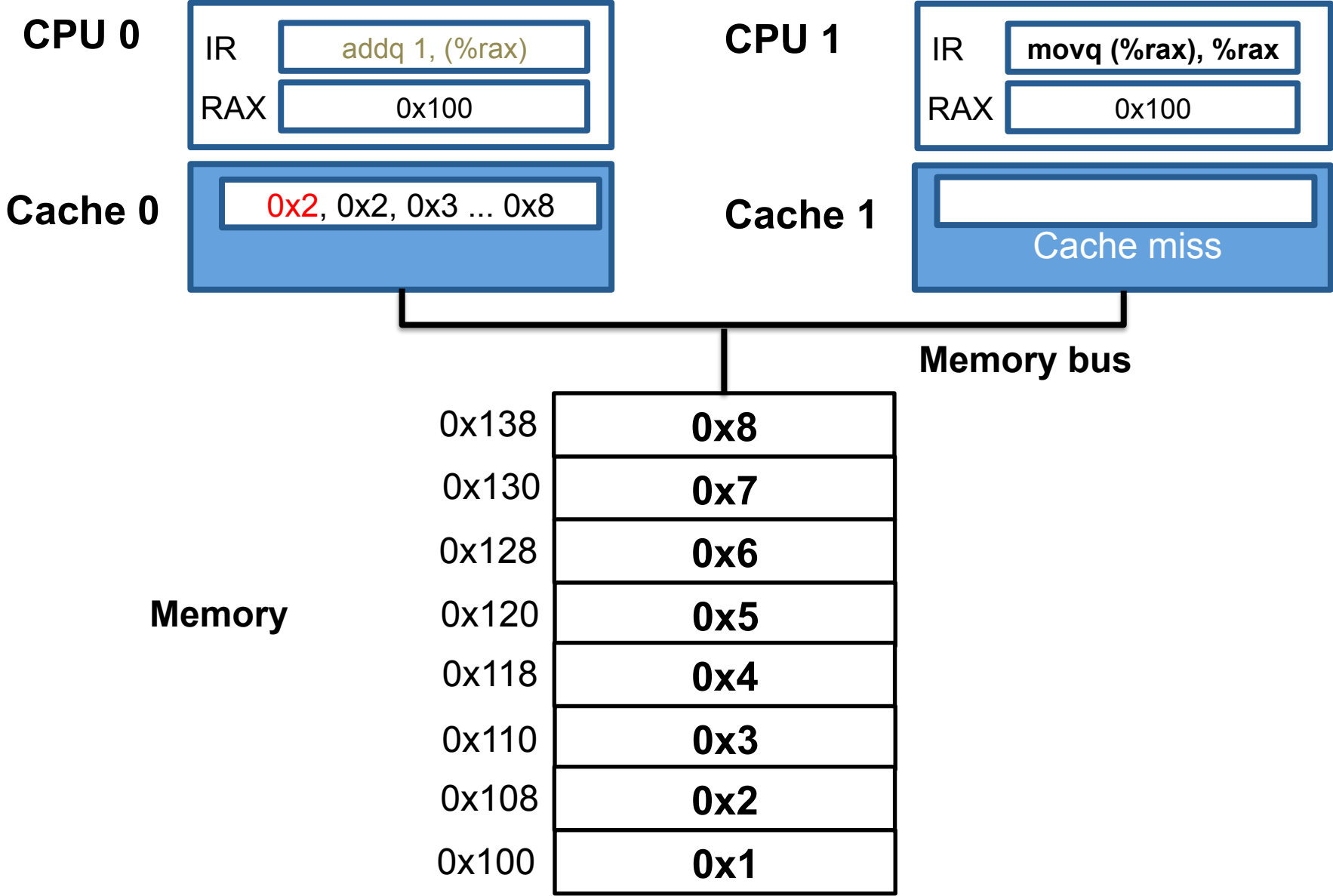
Basic Idea of Cache Coherence



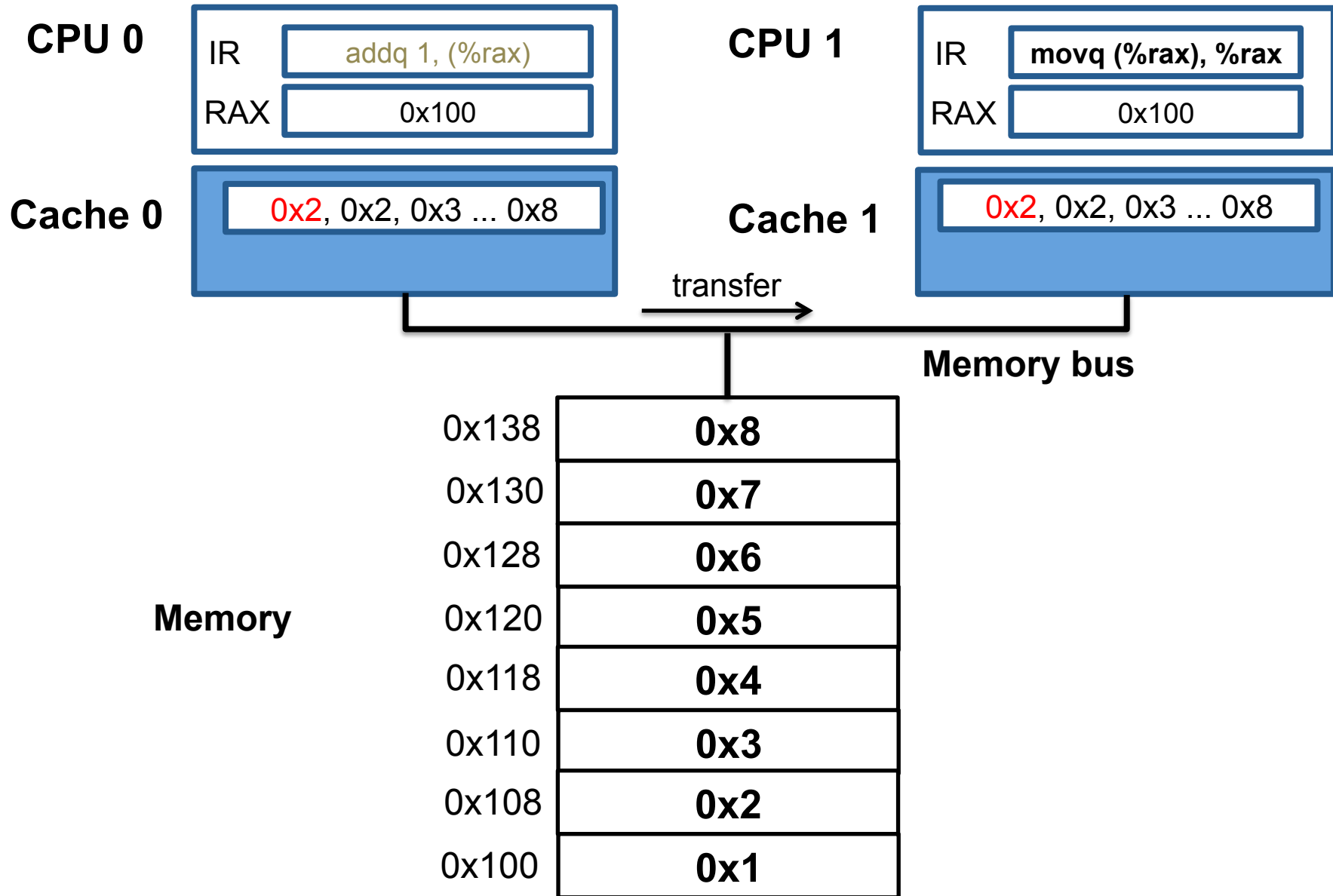
Basic Idea of Cache Coherence



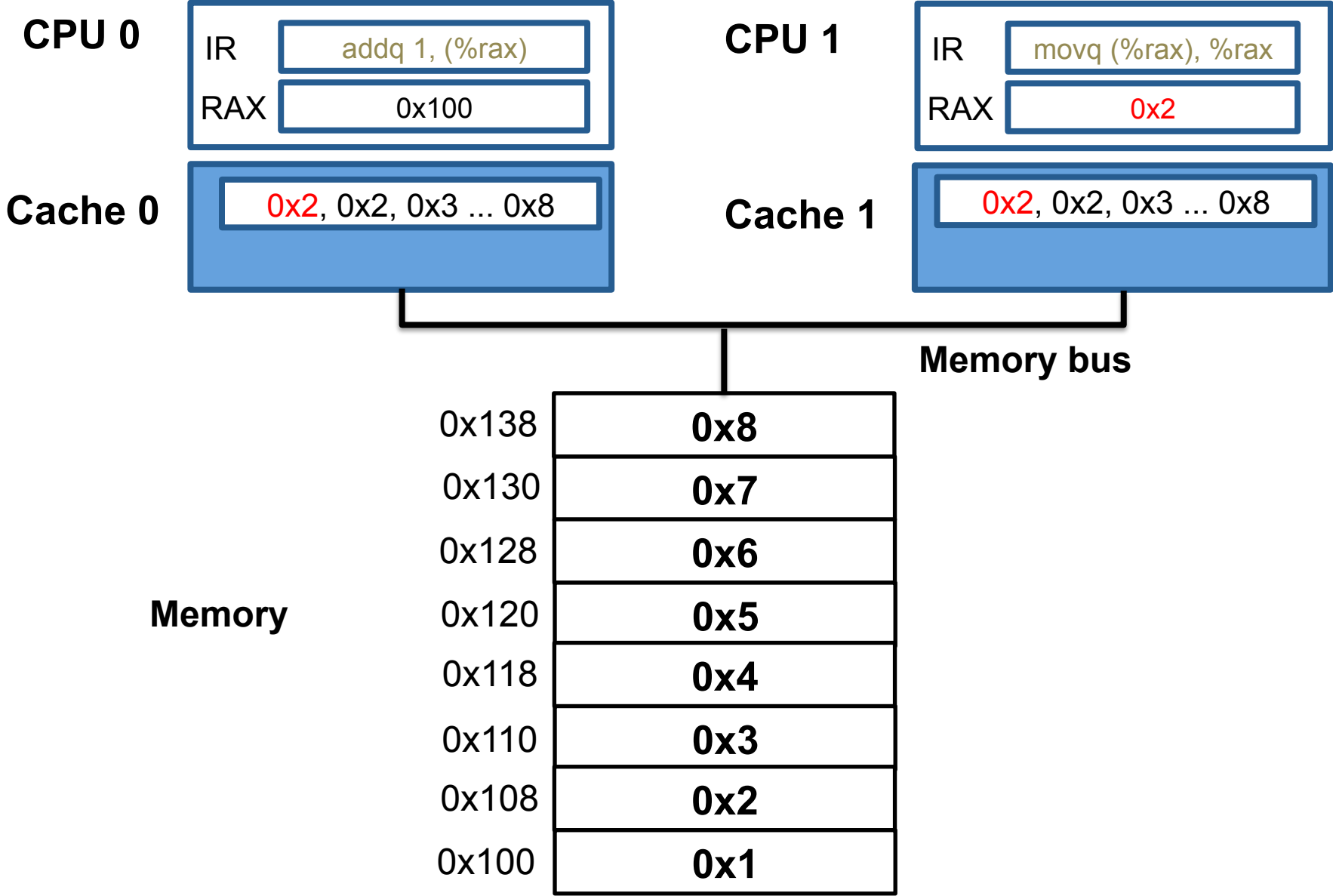
Basic Idea of Cache Coherence



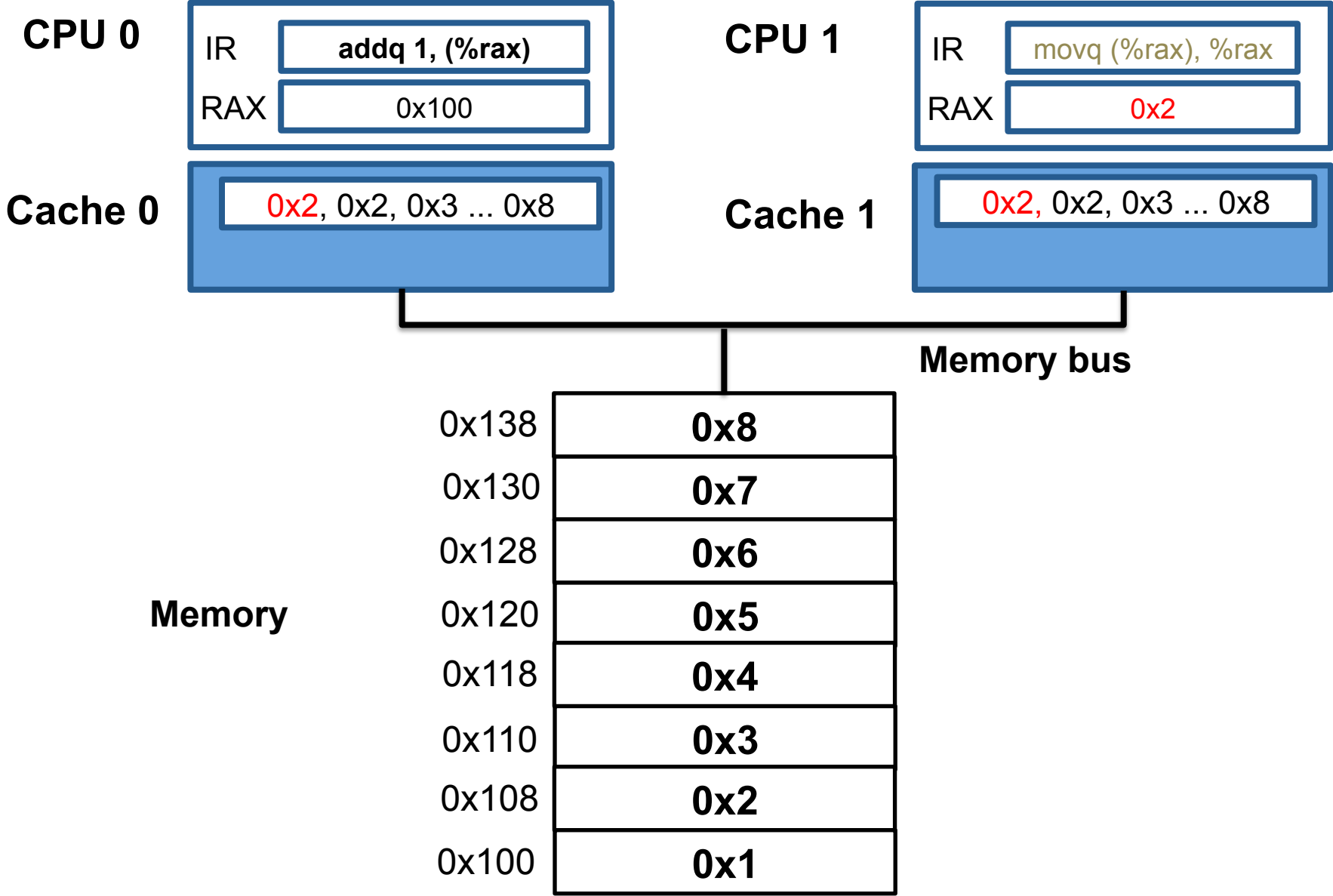
Basic Idea of Cache Coherence



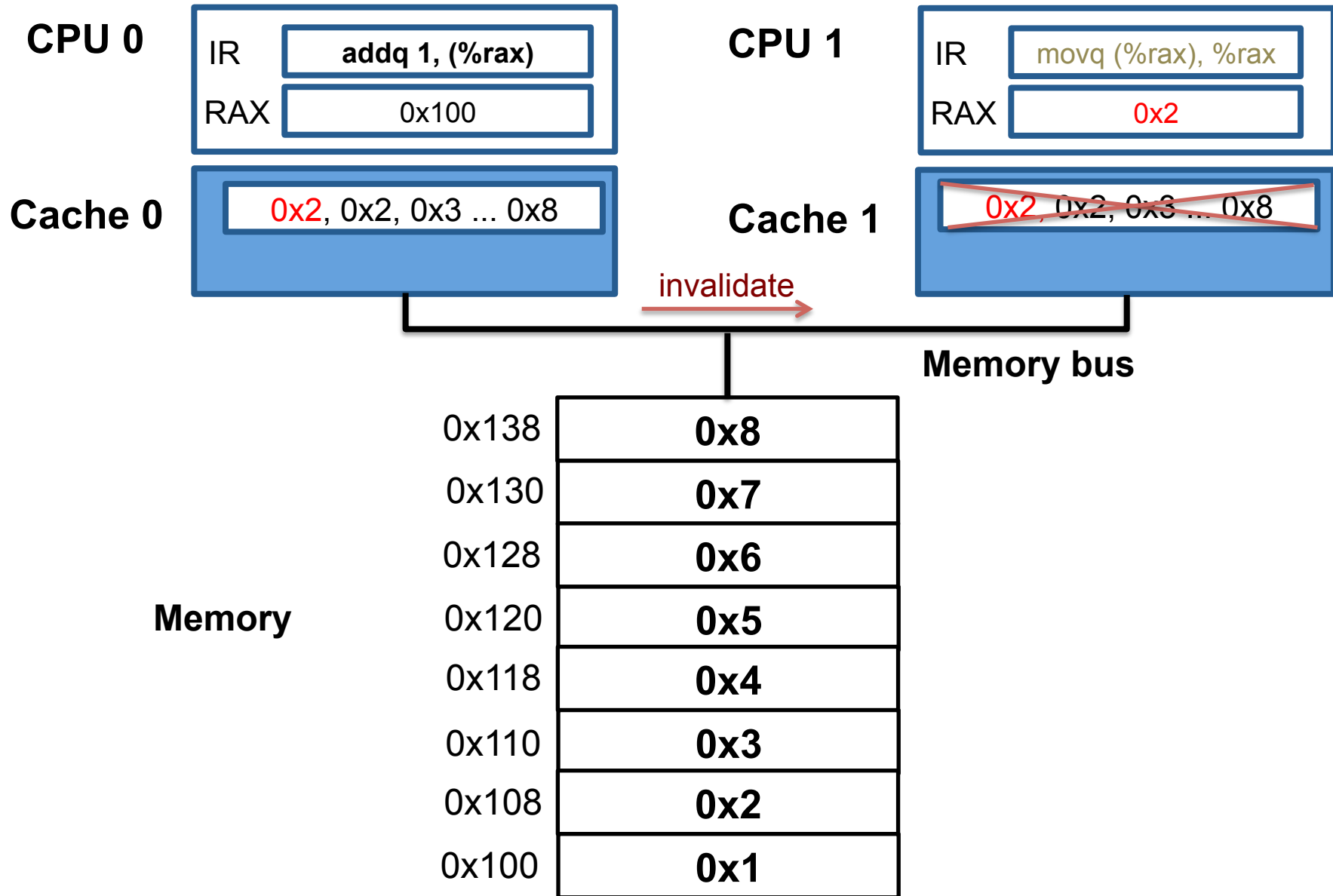
Basic Idea of Cache Coherence



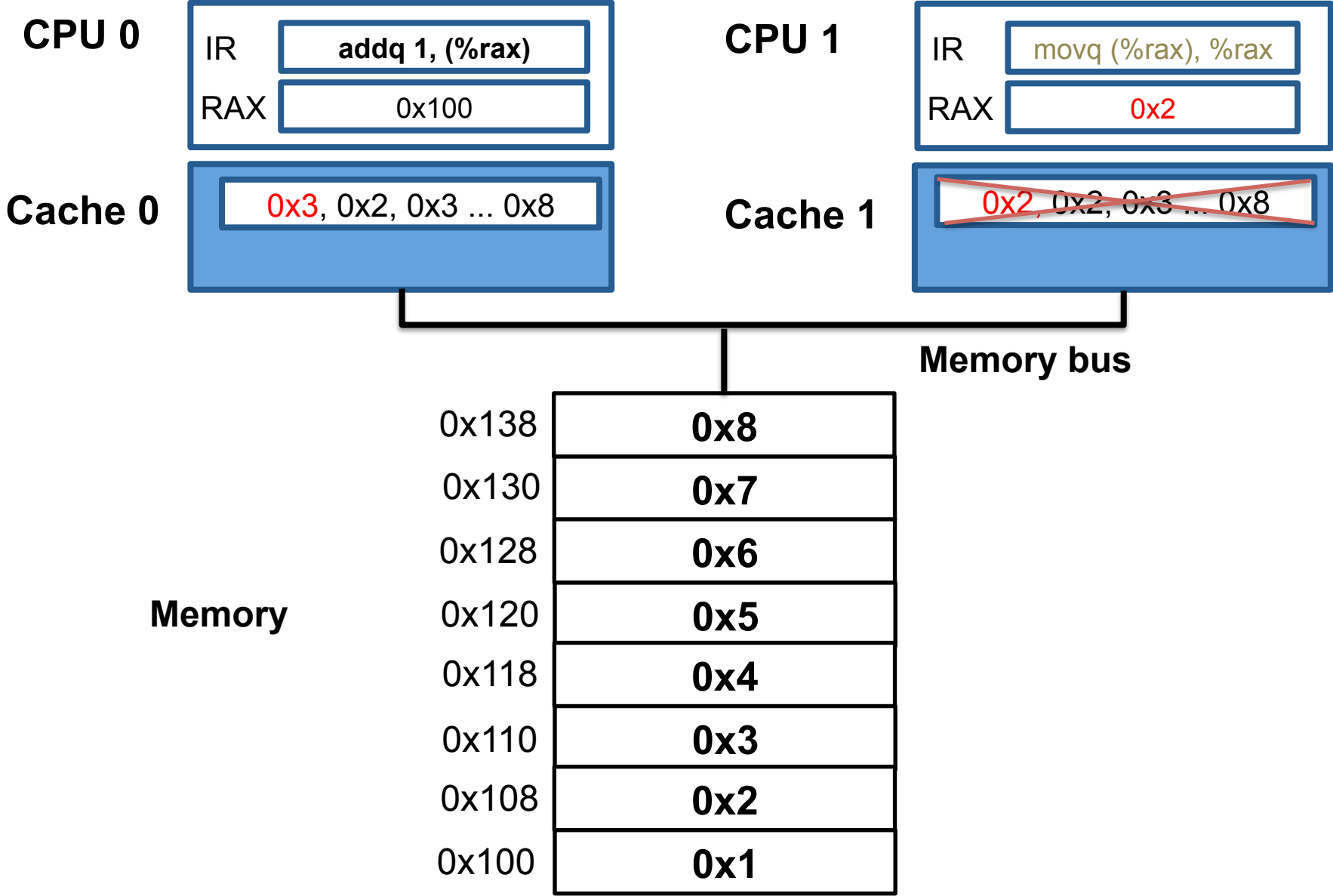
Basic Idea of Cache Coherence



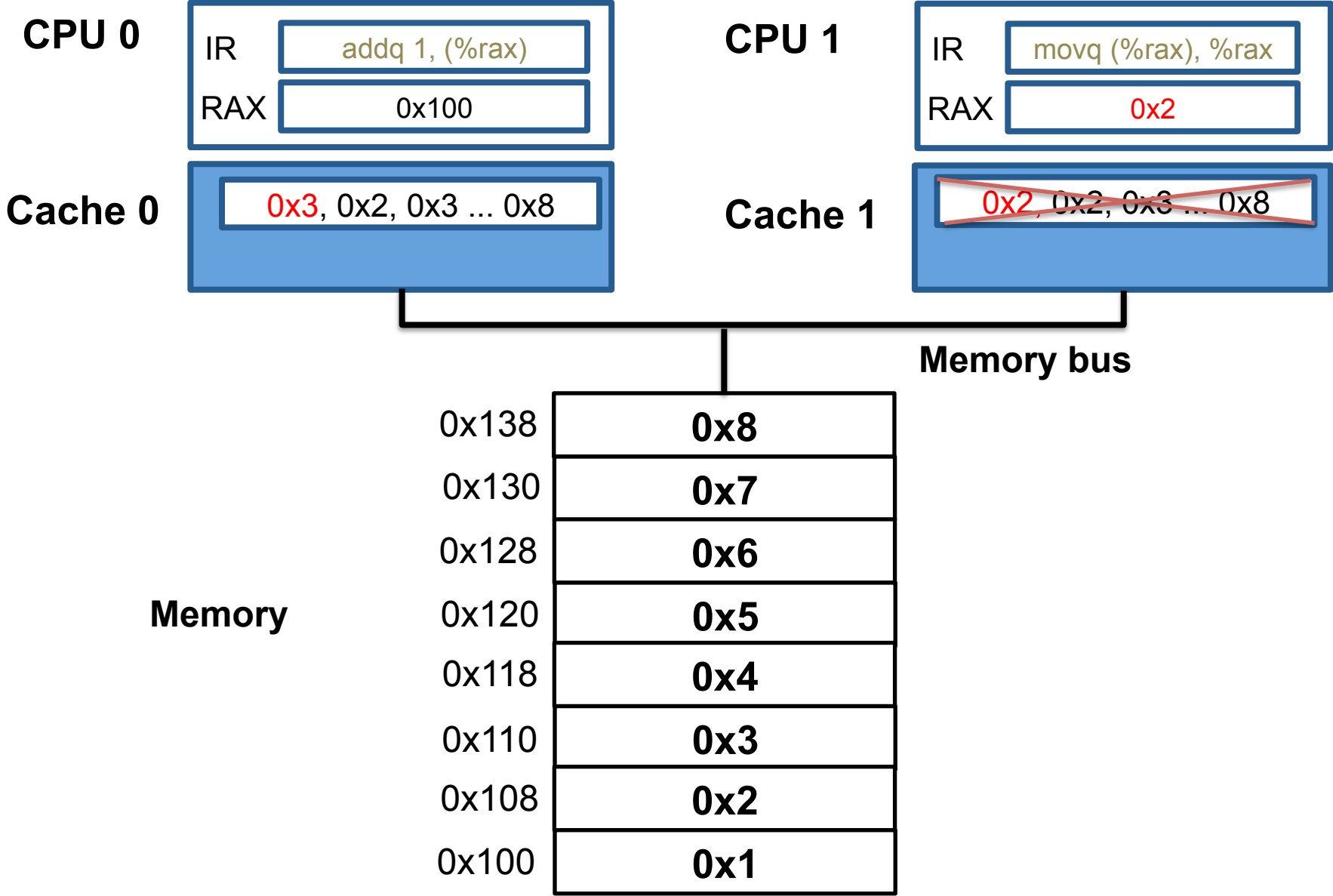
Basic Idea of Cache Coherence



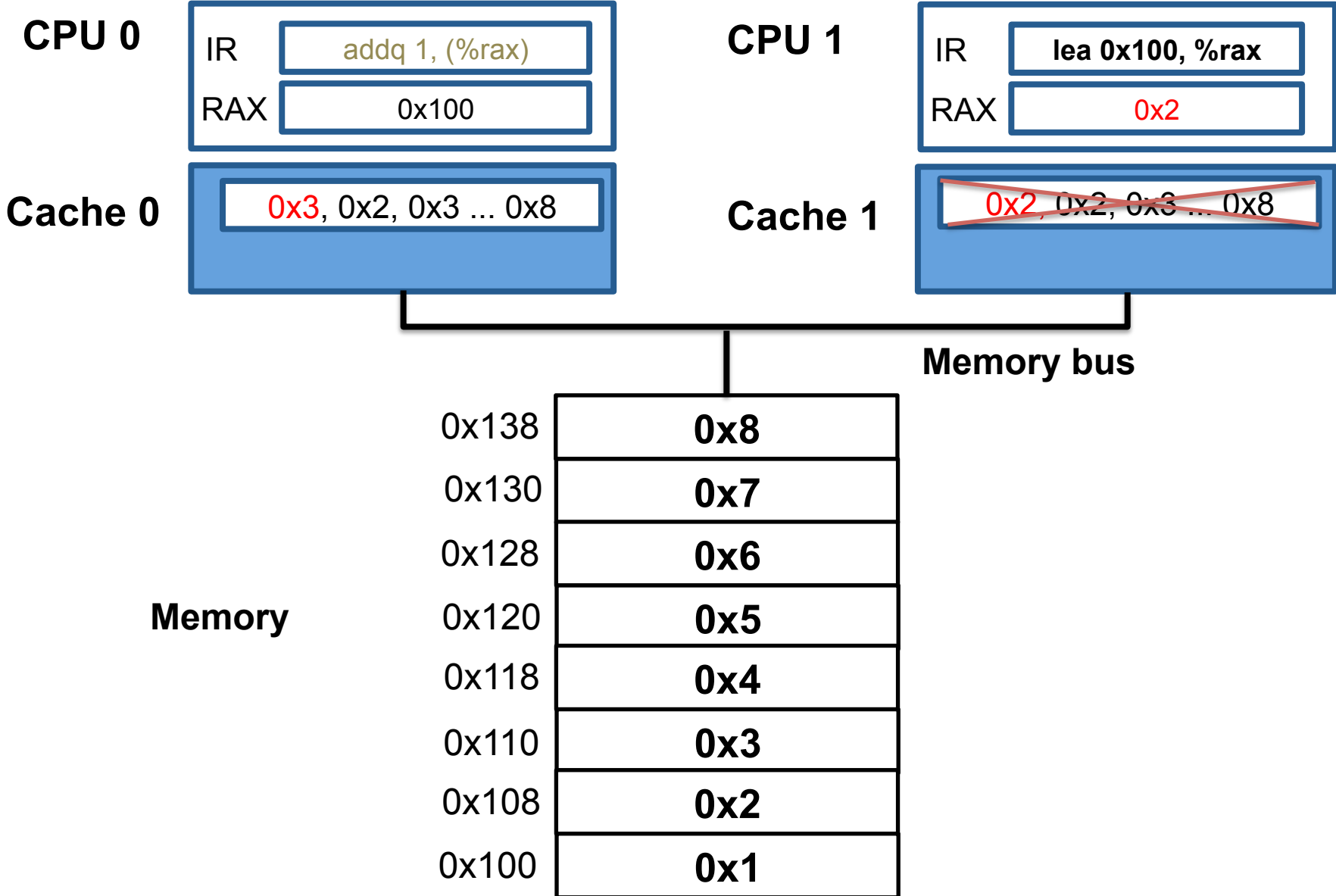
Basic Idea of Cache Coherence



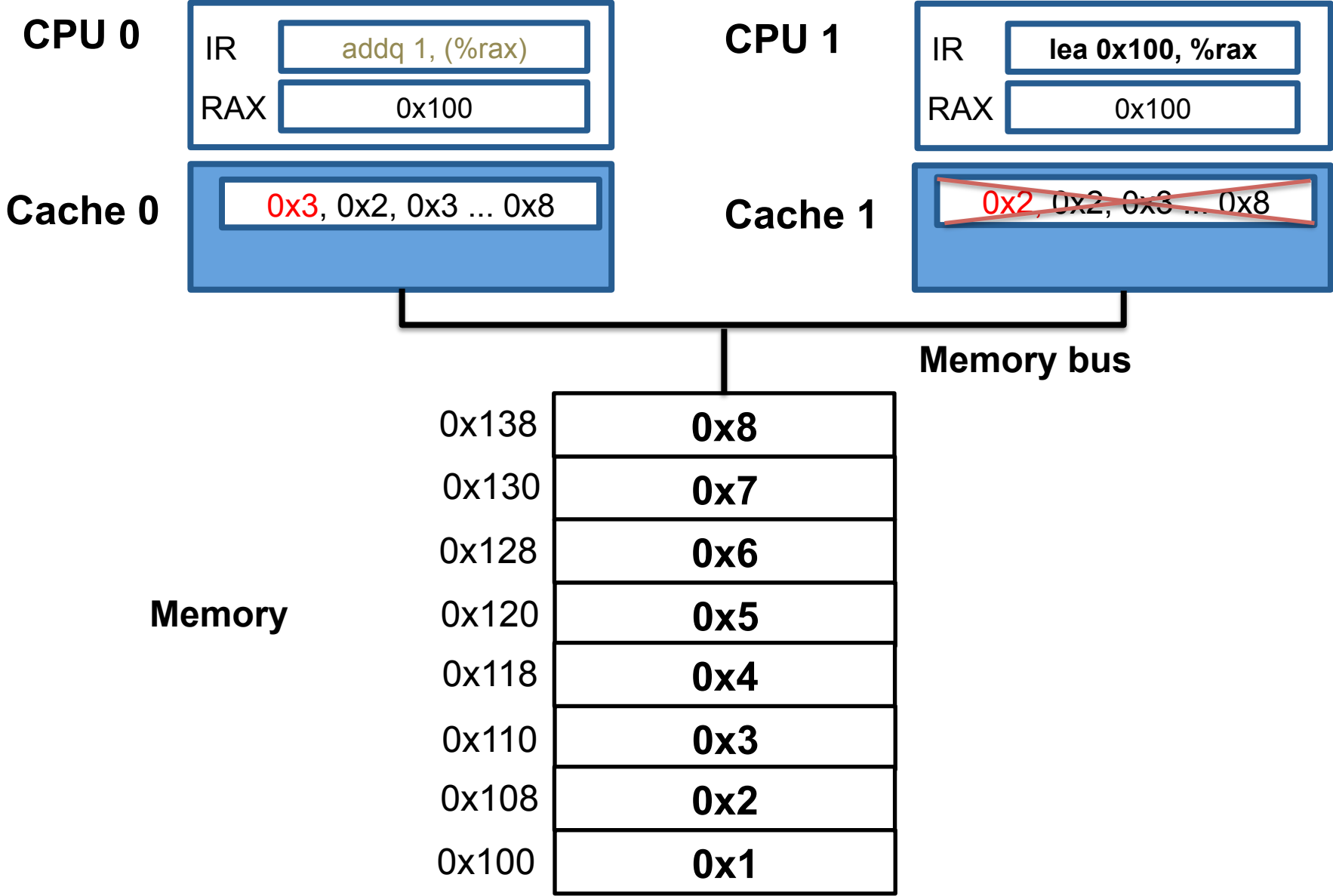
Basic Idea of Cache Coherence



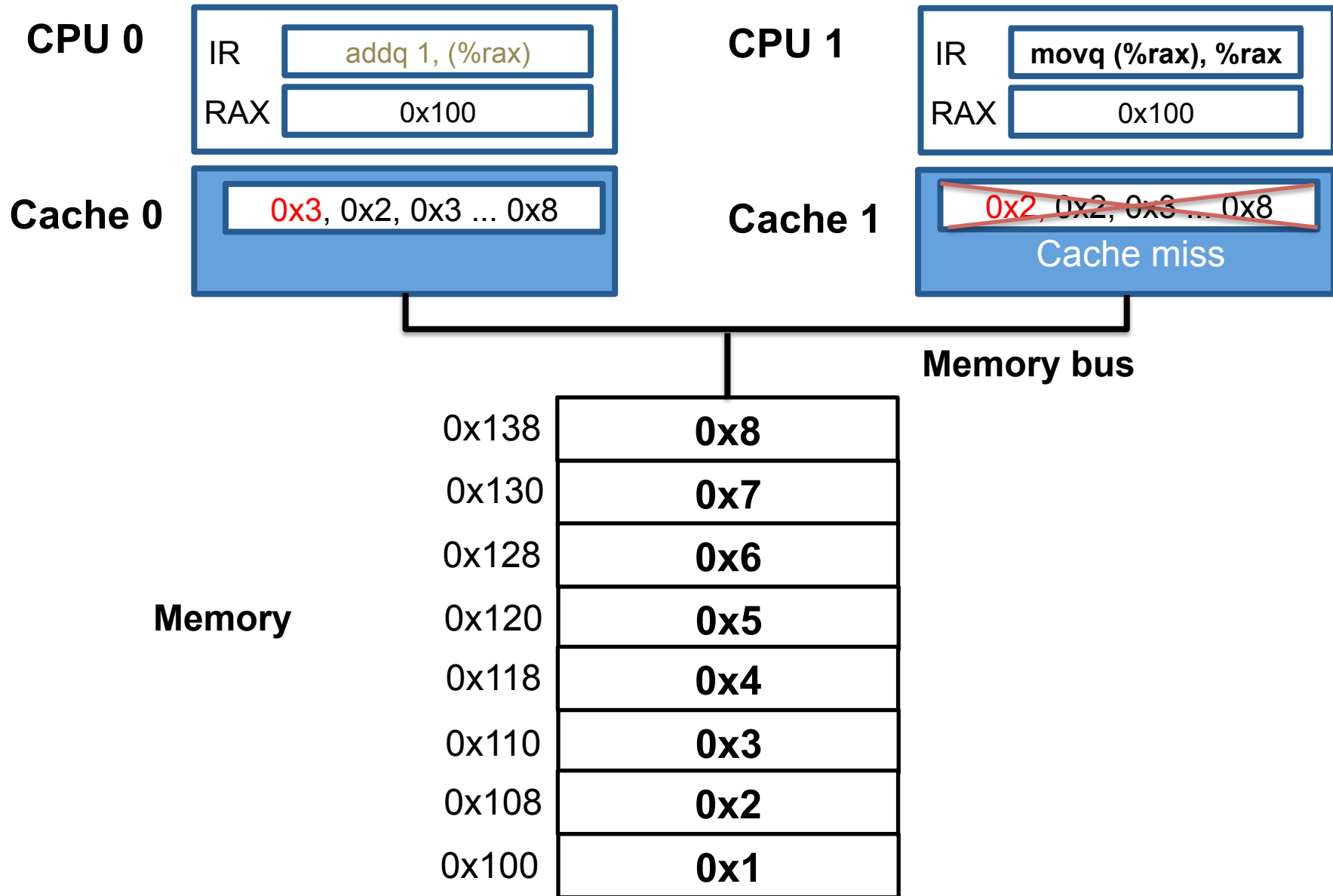
Basic Idea of Cache Coherence



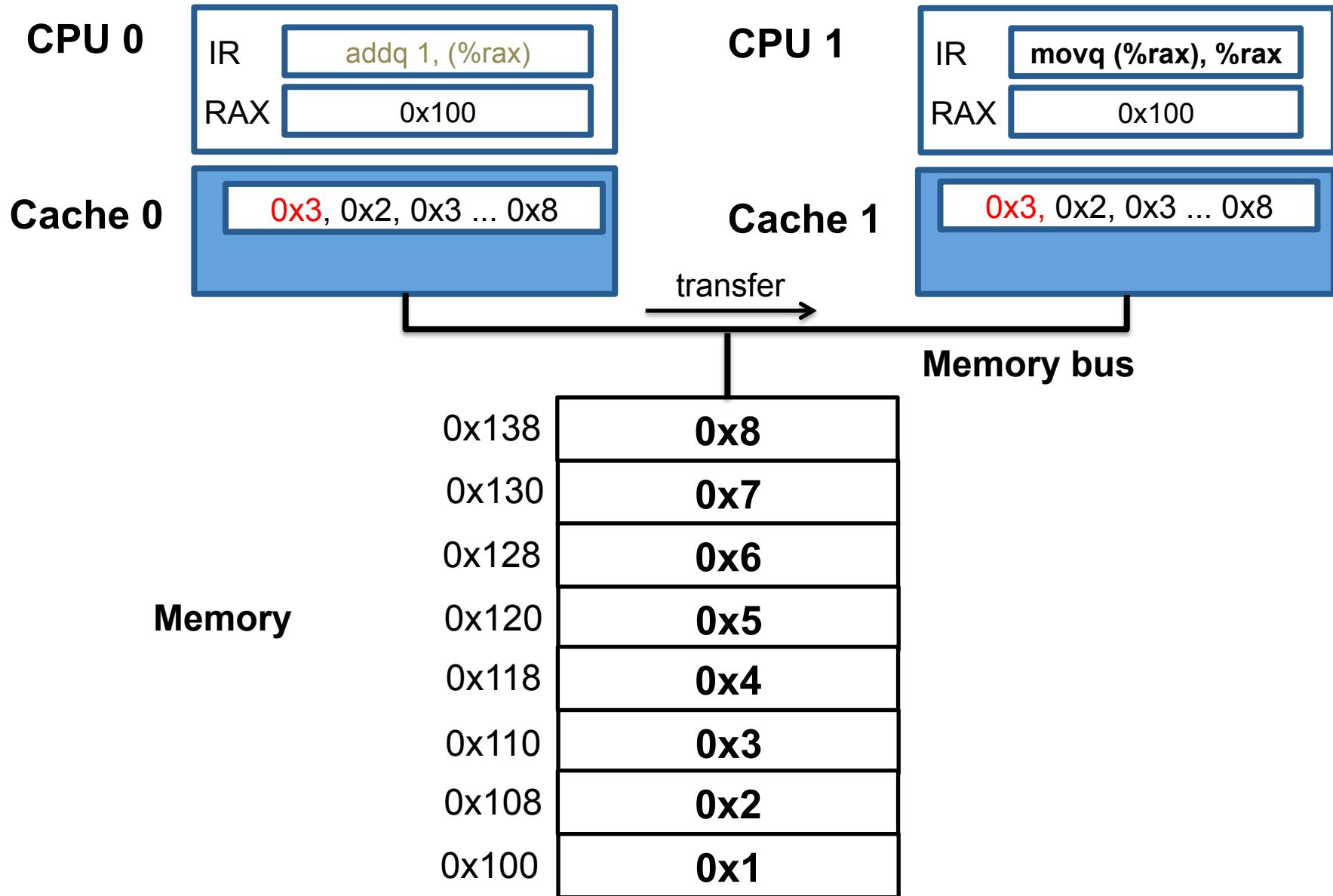
Basic Idea of Cache Coherence



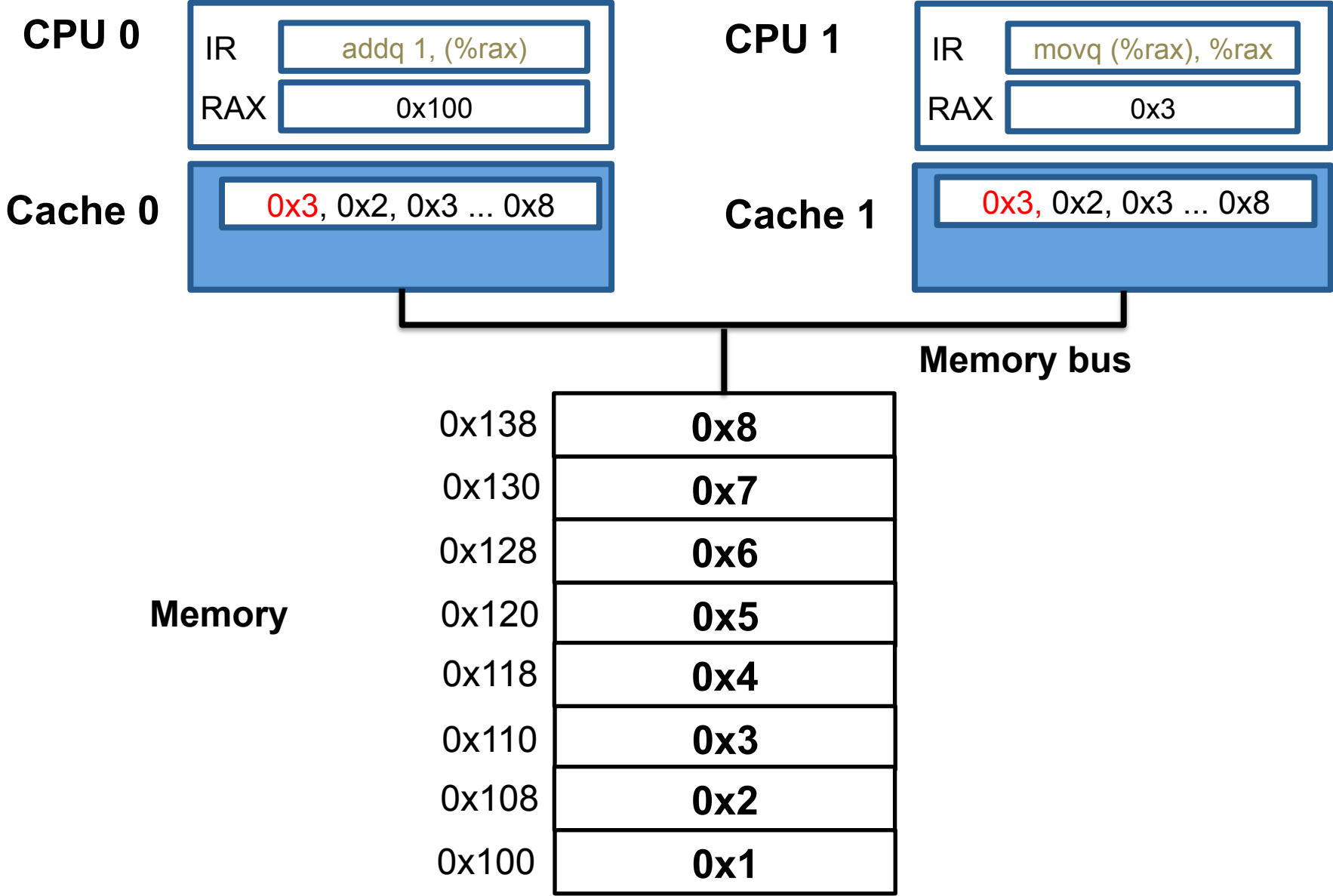
Basic Idea of Cache Coherence



Basic Idea of Cache Coherence



Basic Idea of Cache Coherence



Update global variable

addq 1, (%rax) **CPU 0**

cache miss

addq 1, (%rax) **CPU 1**

cache miss

Update global variable

addq 1, (%rax) **CPU 0**

cache miss



load cacheline from memory

addq 1, (%rax) **CPU 1**

cache miss



load cacheline from memory

Update global variable

addq 1, (%rax)

CPU 0

cache miss



load cacheline from memory



load data into cpu buffer

addq 1, (%rax)

CPU 1

cache miss



load cacheline from memory



load data into cpu buffer

Update global variable

addq 1, (%rax)

CPU 0

cache miss



load cacheline from memory



load data into cpu buffer



calculation

addq 1, (%rax)

CPU 1

cache miss



load cacheline from memory



load data into cpu buffer



calculation

Update global variable

addq 1, (%rax)

CPU 0

cache miss



load cacheline from memory



load data into cpu buffer



calculation



invalidate cpu1's cacheline

addq 1, (%rax)

CPU 1

cache miss



load cacheline from memory



load data into cpu buffer



calculation

be invalidated



Update global variable

addq 1, (%rax)

CPU 0

cache miss



load cacheline from memory



load data into cpu buffer



calculation



invalidate cpu1's cacheline



restore data from cpu buffer to cache

addq 1, (%rax)

CPU 1

cache miss



load cacheline from memory



load data into cpu buffer



calculation

be invalidated

cache miss



Update global variable

addq 1, (%rax)

CPU 0

cache miss



load cacheline from memory



load data into cpu buffer



calculation



invalidate cpu1's cacheline



restore data from cpu buffer to cache

addq 1, (%rax)

CPU 1

cache miss



load cacheline from memory



load data into cpu buffer



calculation

be invalidated

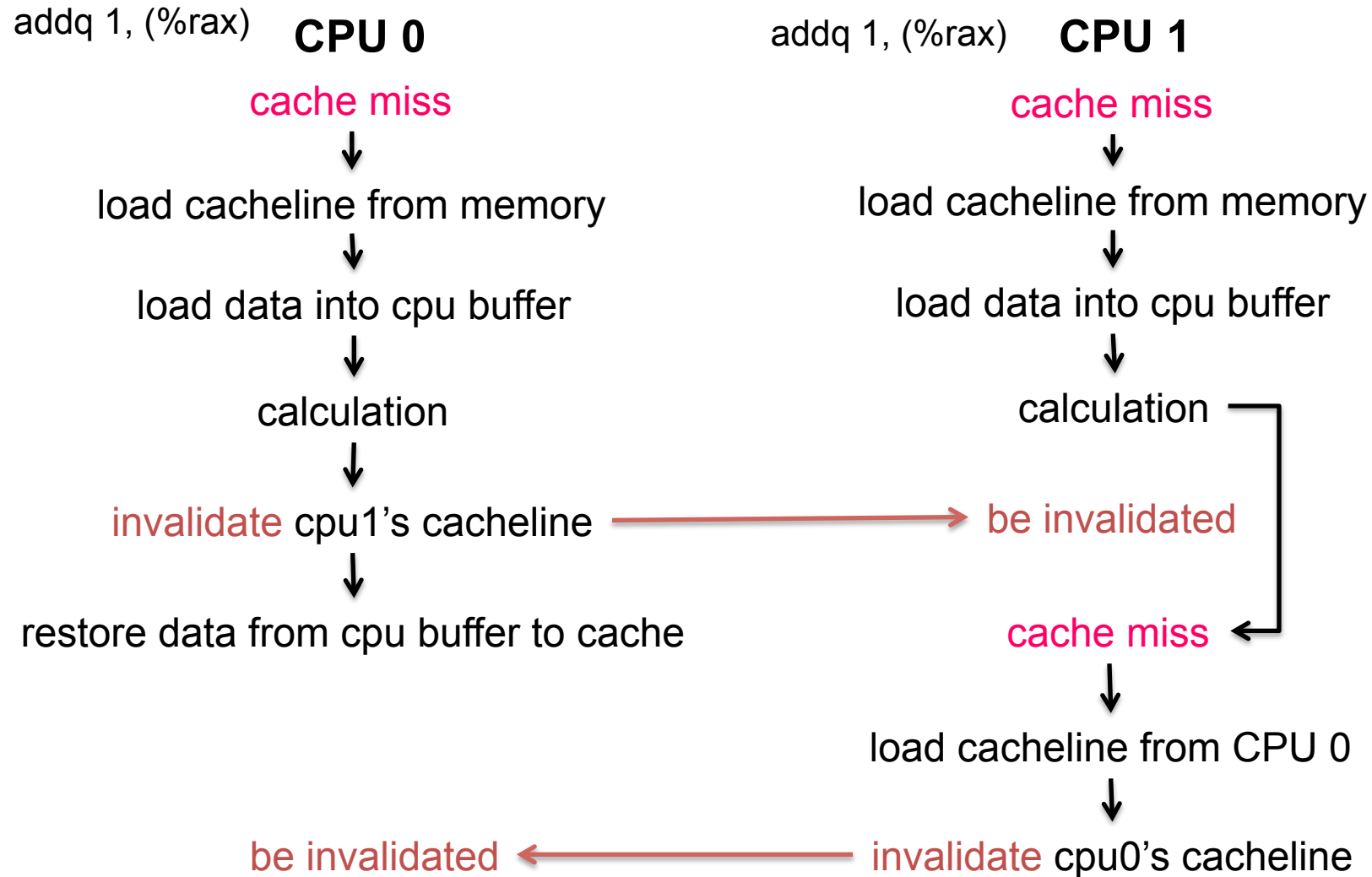
cache miss



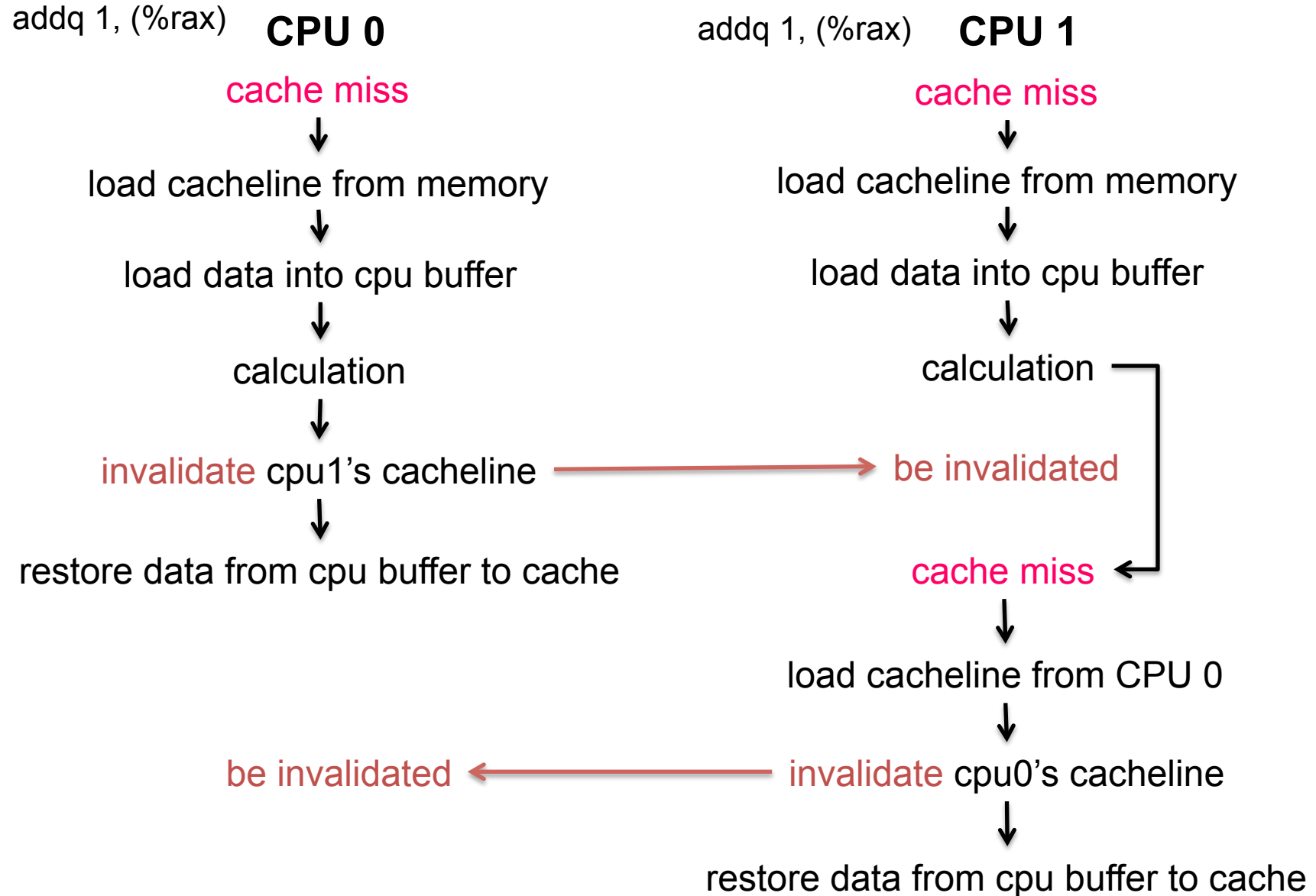
load cacheline from CPU 0



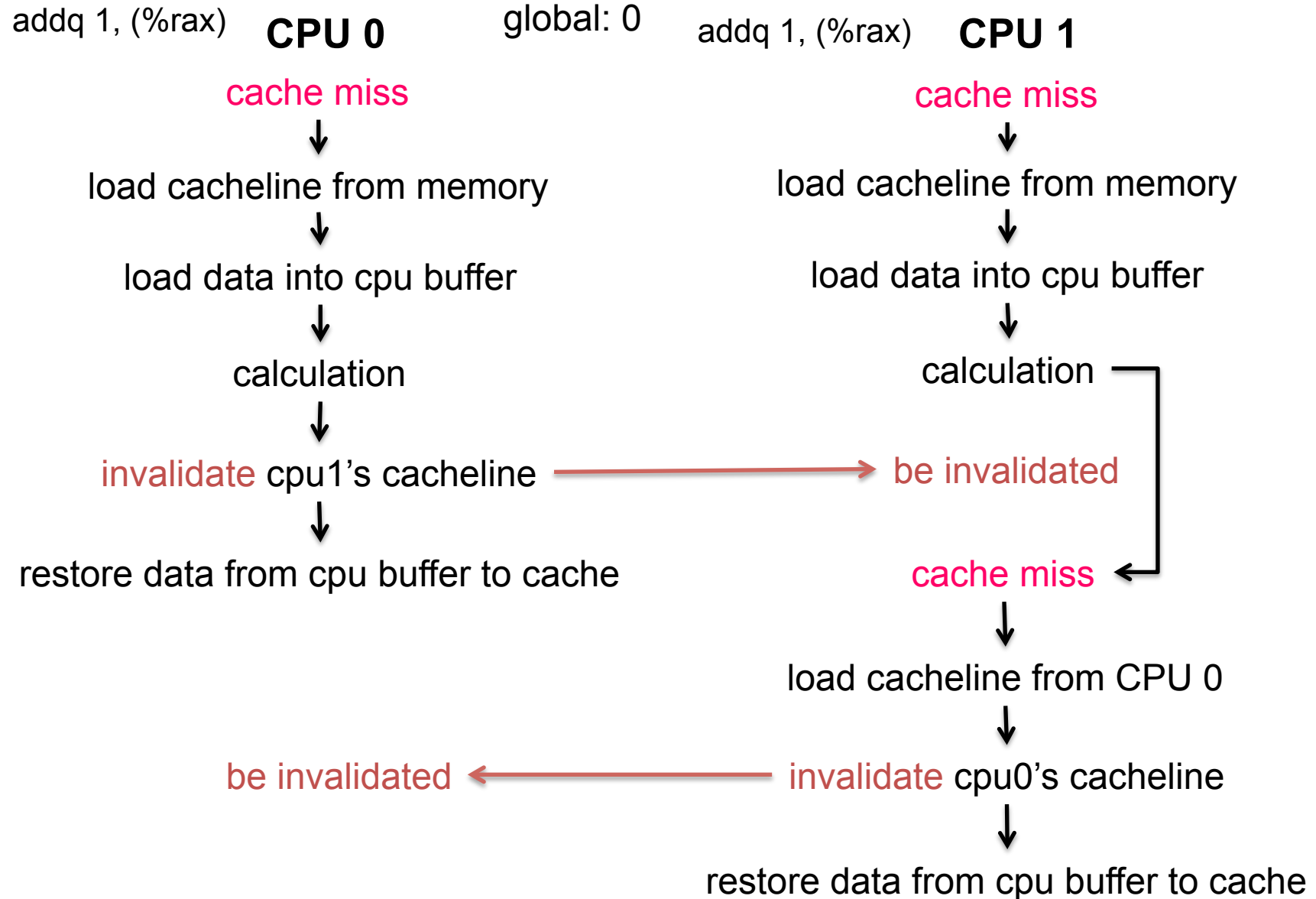
Update global variable



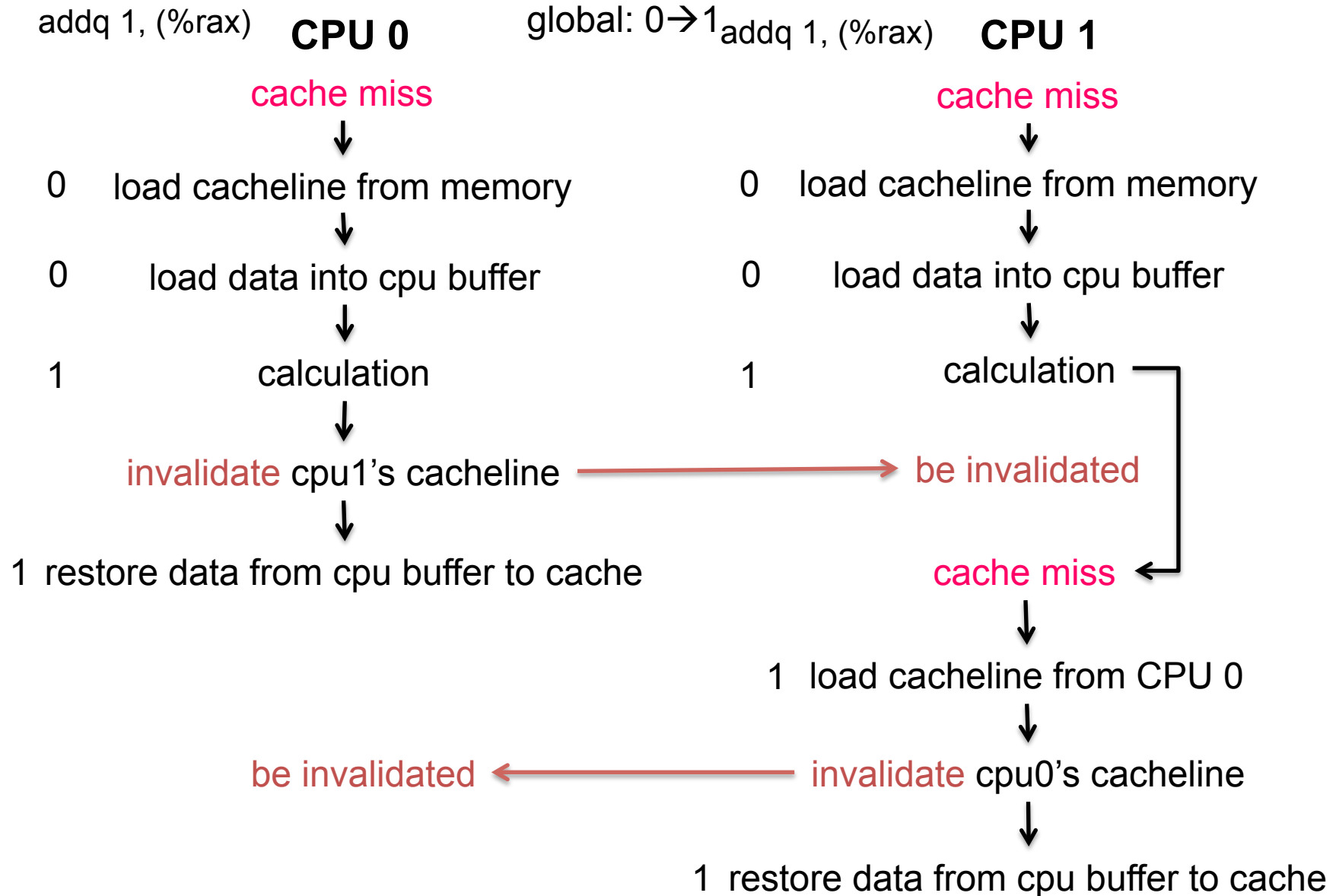
Update global variable



Update global variable



Update global variable



Update global variable with lock

`lock; addq 1, (%rax)` **CPU 0** global: 1

`lock; addq 1, (%rax)` **CPU 1**

lock cacheline/ address /memory bus

lock cacheline/ address /memory bus



cache miss



load cacheline from memory



load into CPU buffer and calculation



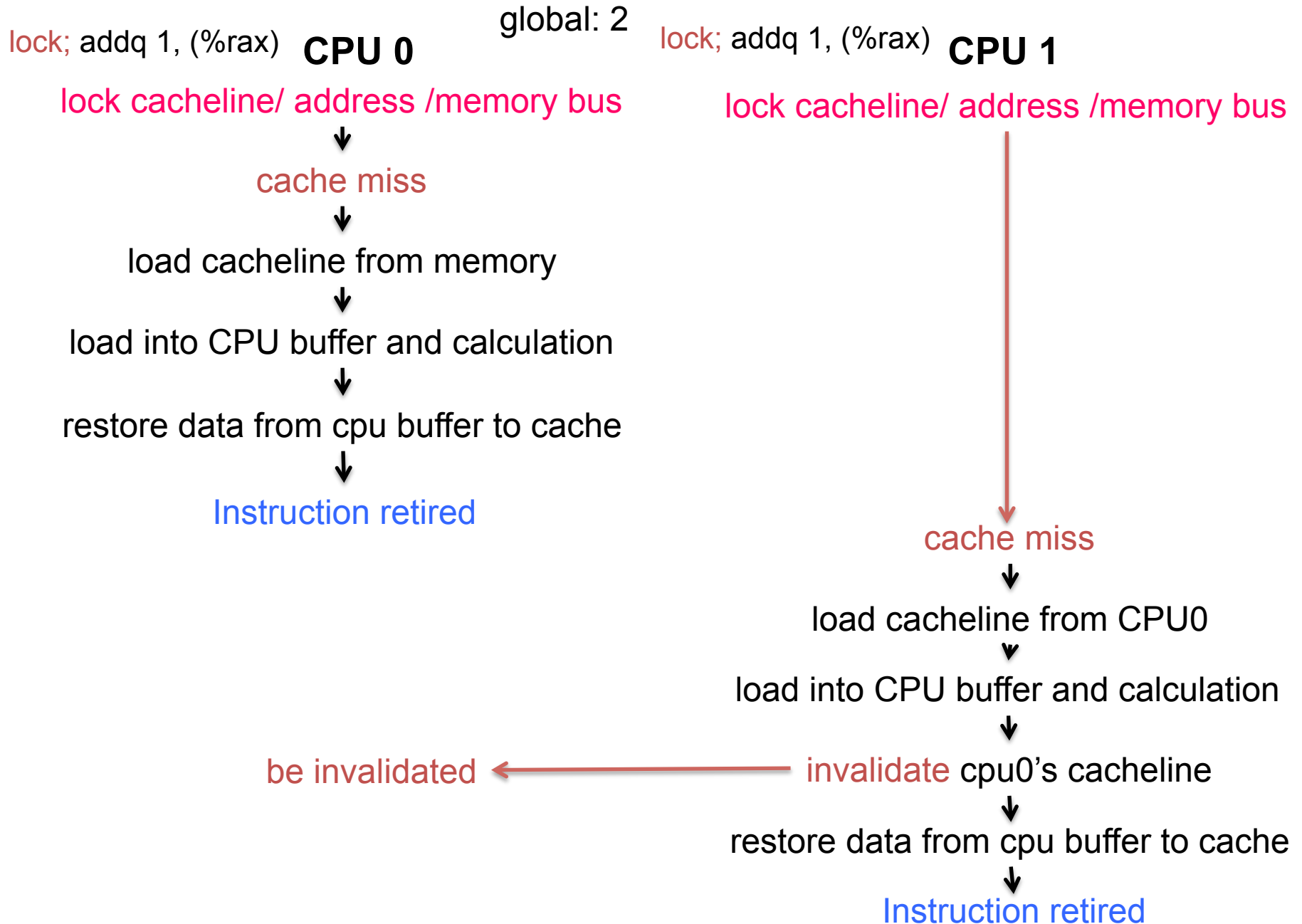
restore data from cpu buffer to cache



Instruction retired



Update global variable with lock



Synchronization Cost

	No Lock	Atomic Instruction	Spin Lock	Pthread Mutex
Single thread	5	19	24	49
Two threads / Same variable		33	127	200
Two threads / Same cacheline		29		
Two threads / Different cachelines		10		

Cycles / Operation

Synchronization magnify the cost of cache coherence

Synchronization Cost

	No Lock	Atomic Instruction	Spin Lock	Pthread Mutex
Single thread	5.5	19.3	24	50.4
Two threads / Same variable	3.0	32.9	124	166.8
Two threads / Same cacheline	3.1	30	63	124
Two threads / Different cachelines	2.9	10	13	25.8

Cycles / Operation

Synchronization magnify the cost of cache coherence