Foundation and the cost of Synchronization

Jinyang Li

Based on the slides of Tiger Wang

What you've learnt

- Challenge of Multi-threaded programming
  - Races, deadlocks
- Pthread library's synchronization primitives:
  - mutex, conditional variable

Today

- How does pthread's mutex work?
- What's the cost of synchronization?

Implement a lock: a naive attempt

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

void lock_init(mutex_t *mu) {
  mu->busy = 0;
}

void lock(mutex_t *mu) {
   while(mu->busy) {}

   mu->busy = 1;
}

void unlock(mutex_t *mu) {
  mu->busy = 0;
}
```

Busy wait
This style of locking is called “Spin Lock”

correct?

Is the naive implementation correct?

Thread 1 | Thread 2 | Thread 3

```
mu->busy = 1
```

```
while (mu->busy)
//mu->busy=1
```

```
while (mu->busy)
//mu->busy=1
```

```
mu->busy = 0
```

```
while (mu->busy)
//mu->busy=0
```

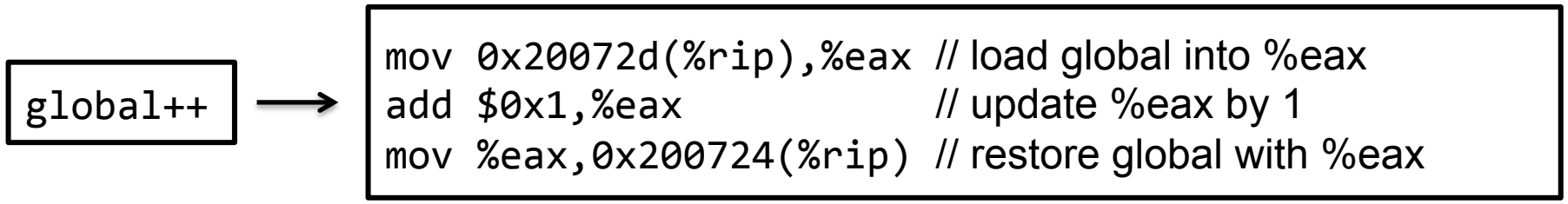
```
while (mu->busy)
//mu->busy=0
```

```
mu->busy = 1
```

```
mu->busy = 1
```

Both threads grabbed lock

are x86 instructions atomic?



Thread 1

```
mov 0x20072d(%rip),%eax

add $0x1,%eax

mov %eax,0x200724(%rip)
```

Thread 2

```
mov 0x20072d(%rip),%eax

add $0x1,%eax

mov %eax,0x200724(%rip)
```

x86 atomic instructions

- We need hardware support to implement locks
- x86 provide atomic instructions:
  - An atomic instruction performs its reads/writes on one or more memory locations atomically.

Multiple instructions' memory access
do NOT interleave (but execute one after another)

A conceptual model for how CPU executes atomic instruction:

freeze other CPUs' activities for the memory address

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

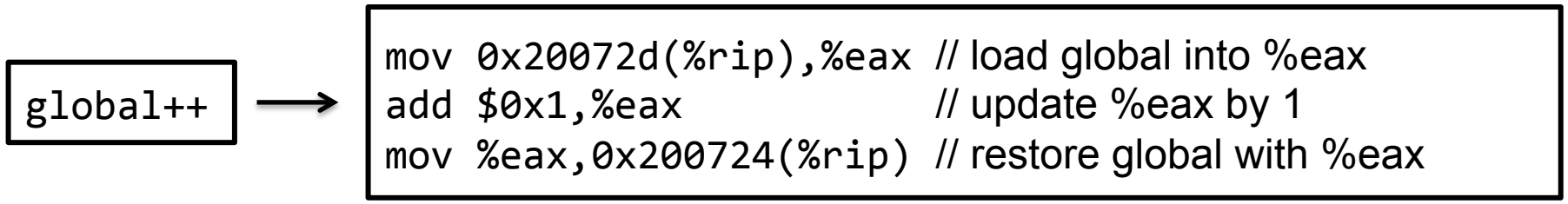
Unlock the memory address

2 types of `atomic` instructions

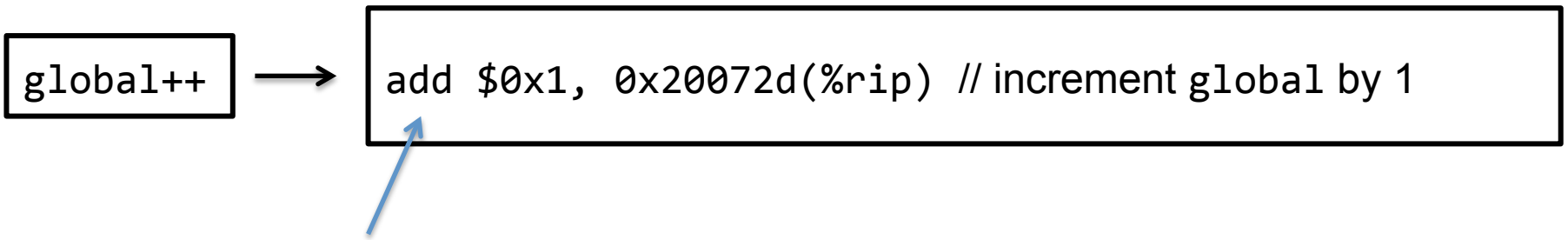
atomic with lock prefix
add, sub
inc, dec
and, or, xor
cmpxchg
...

atomic instructions
mov
xchg
...

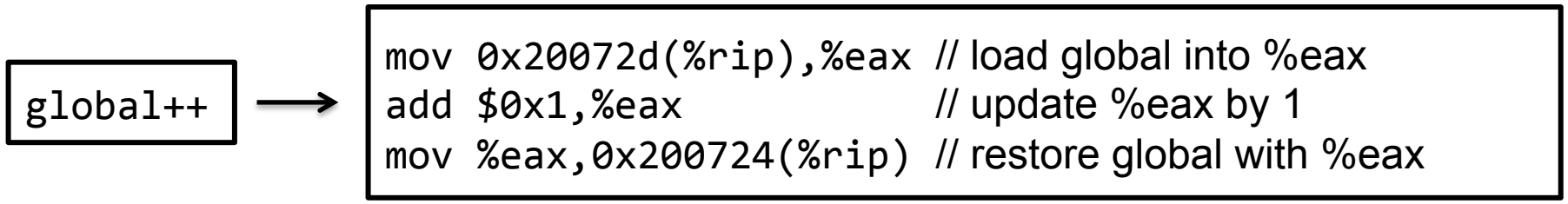
Atomic add (using lock prefix)



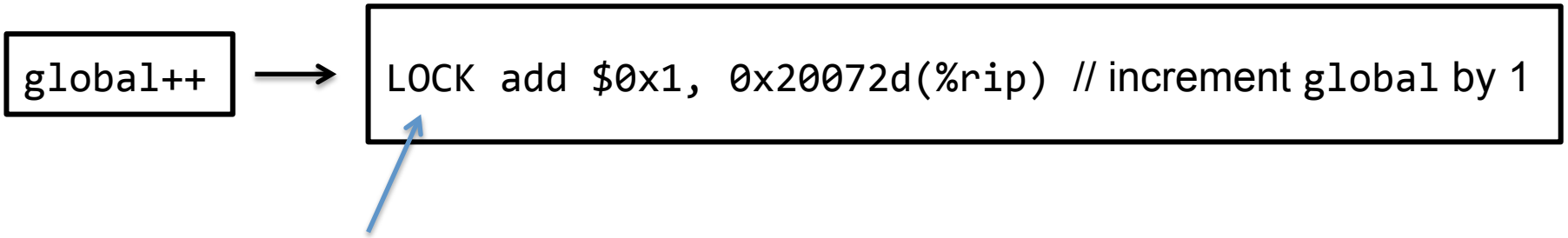
How about “directly” adding to memory?



Atomic add (using lock prefix)



Using atomic add to increment global



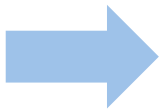
xchg instruction

`xchg op1, op2`

- Swap op1 with op2

`xchg` *`reg, reg`*
`xchg` *`reg, mem`*
`xchg` *`mem, reg`*

```
xchgq %rax, (%rdi)
```



```
movq %rax, %r10
movq (%rdi), %rax
movq %r10, (%rdi)
```

executes atomically

Wrap xchg in a C function

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

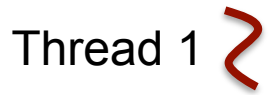
Atomically store x in the memory pointed by `ptr`,
Return the old value stored at `ptr`.

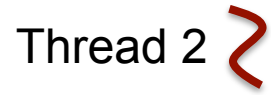
Implement a lock using xchg

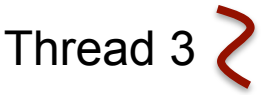
```
typedef struct {
  int busy;
} mutex_t;

void lock_init(mutex_t *mu) {
  mu->busy = 0;
}
```

```
void lock(mutex_t *mu) {
   while (xchg(&mu->busy, 1) !=0) {}

}

void unlock(int *mu) {
   xchg(&mu->busy, 0);
}
```

Spin lock based on xchg





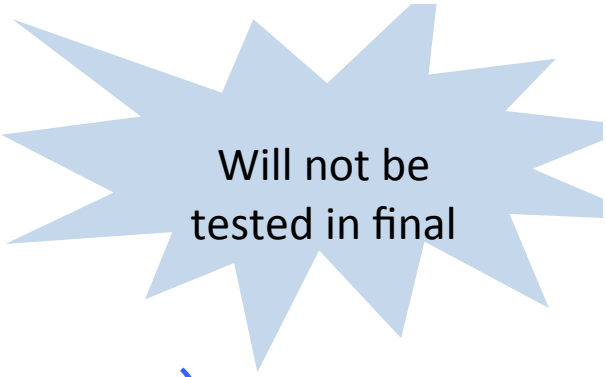



```
xchg(&mu->busy,1)=0
                        while (xchg(&mu->busy,1)!=0)
                        //xchg(..) = 1
                                                    while(xchg(&mu->busy,1)!=0)
                                                    //xchg(..) = 1
  xchg(&mu->busy,0)
                         while (xchg(&mu->busy,1)!=0)
                         //xchg(..) = 0
                                                     while(xchg(&mu->busy,1)!=0
                                                     //xchg(..) = 1
```

Why not always use spin locks?

- If 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



- `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 busy;
} mutex_t;

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

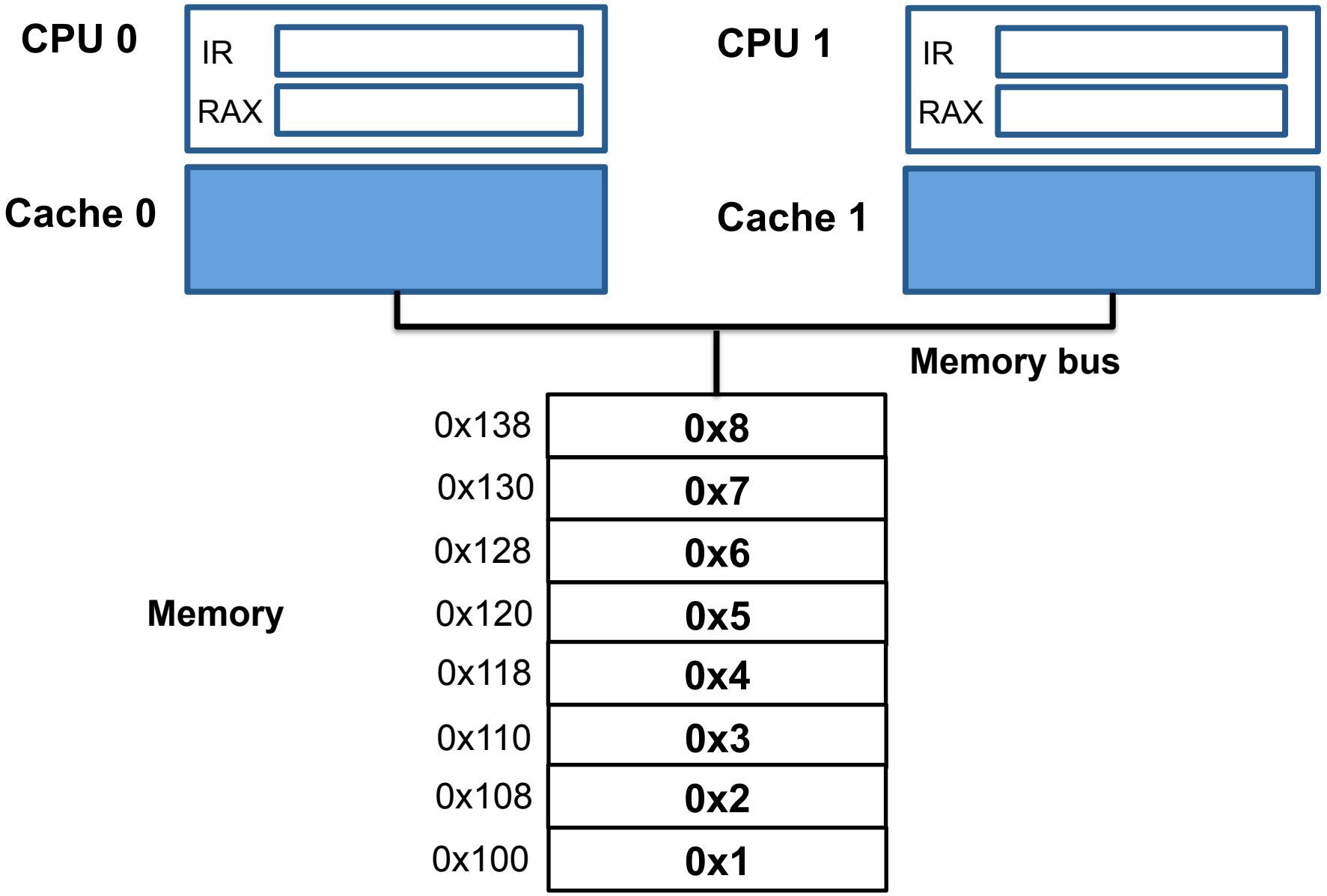
```
void mutex_lock(mutex_t *mu) {
    while(xchg(&mu->busy, 1) != 0) {
                                ???
    }
}
void mutex_unlock(mutex_t *mu) {
    xchg(&mu->busy, 0);
                                ???
}
```

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

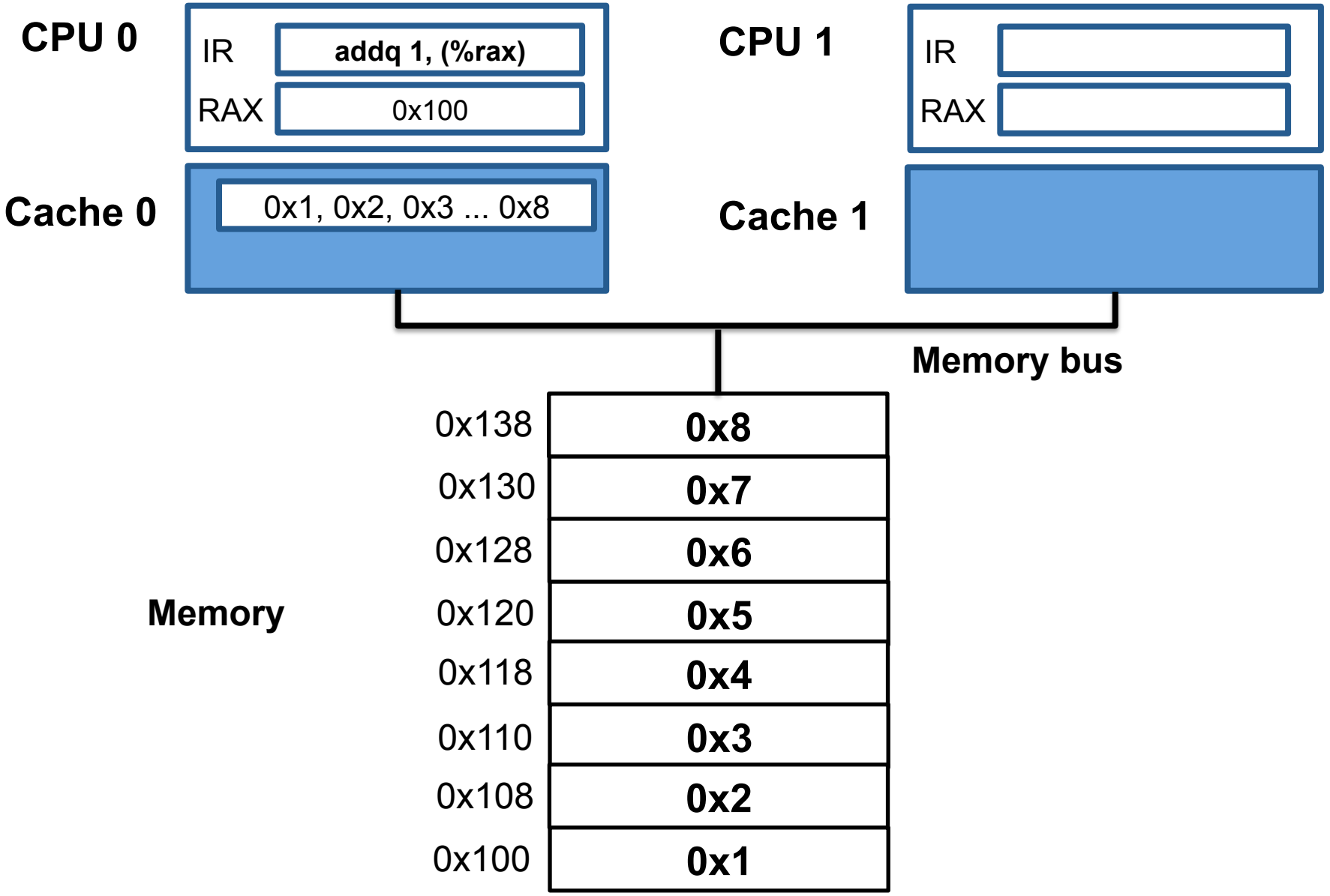
Not efficient as futex_wake is called even if no thread is waiting.

The cost of synchronization

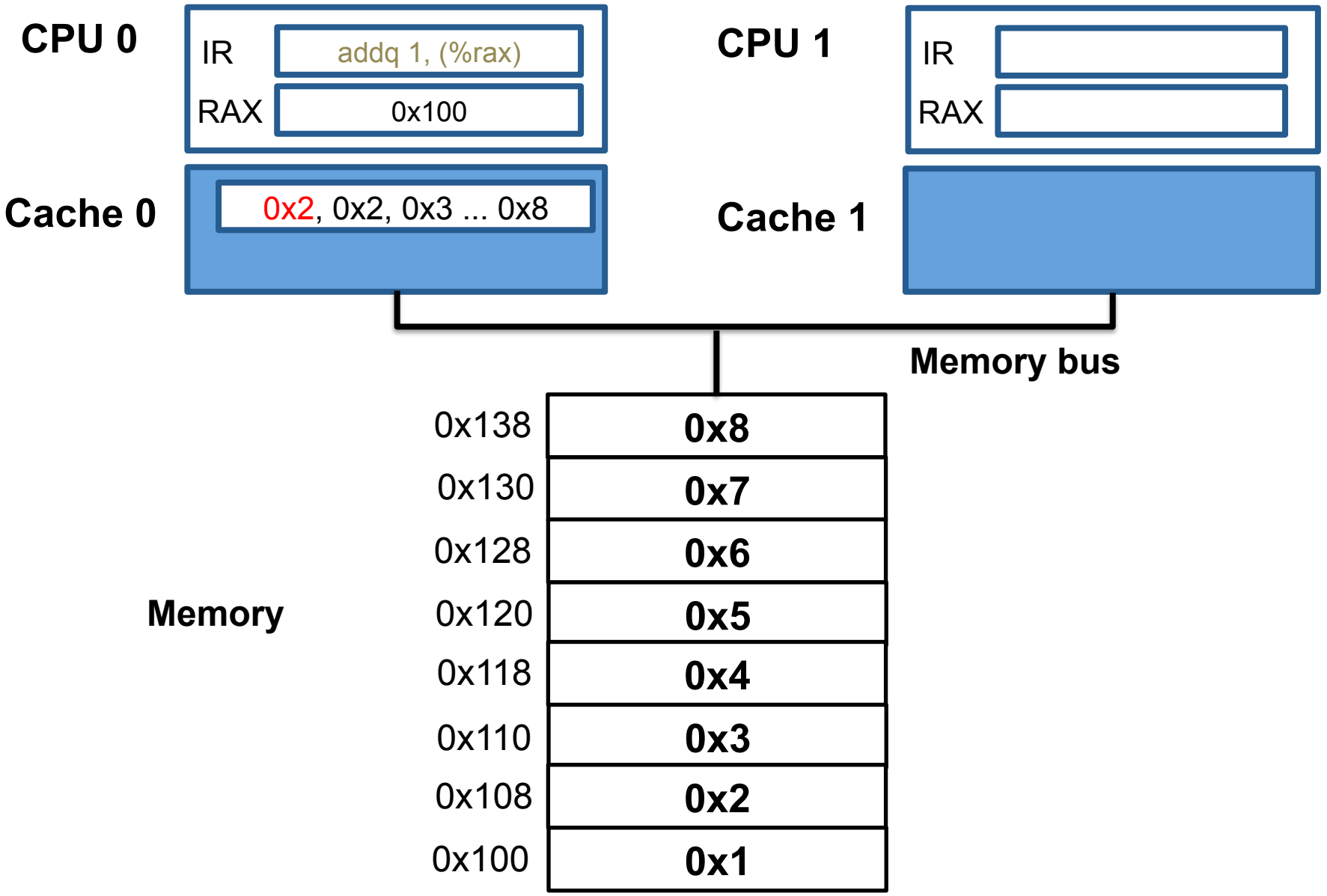
Basic Idea of Cache Coherence



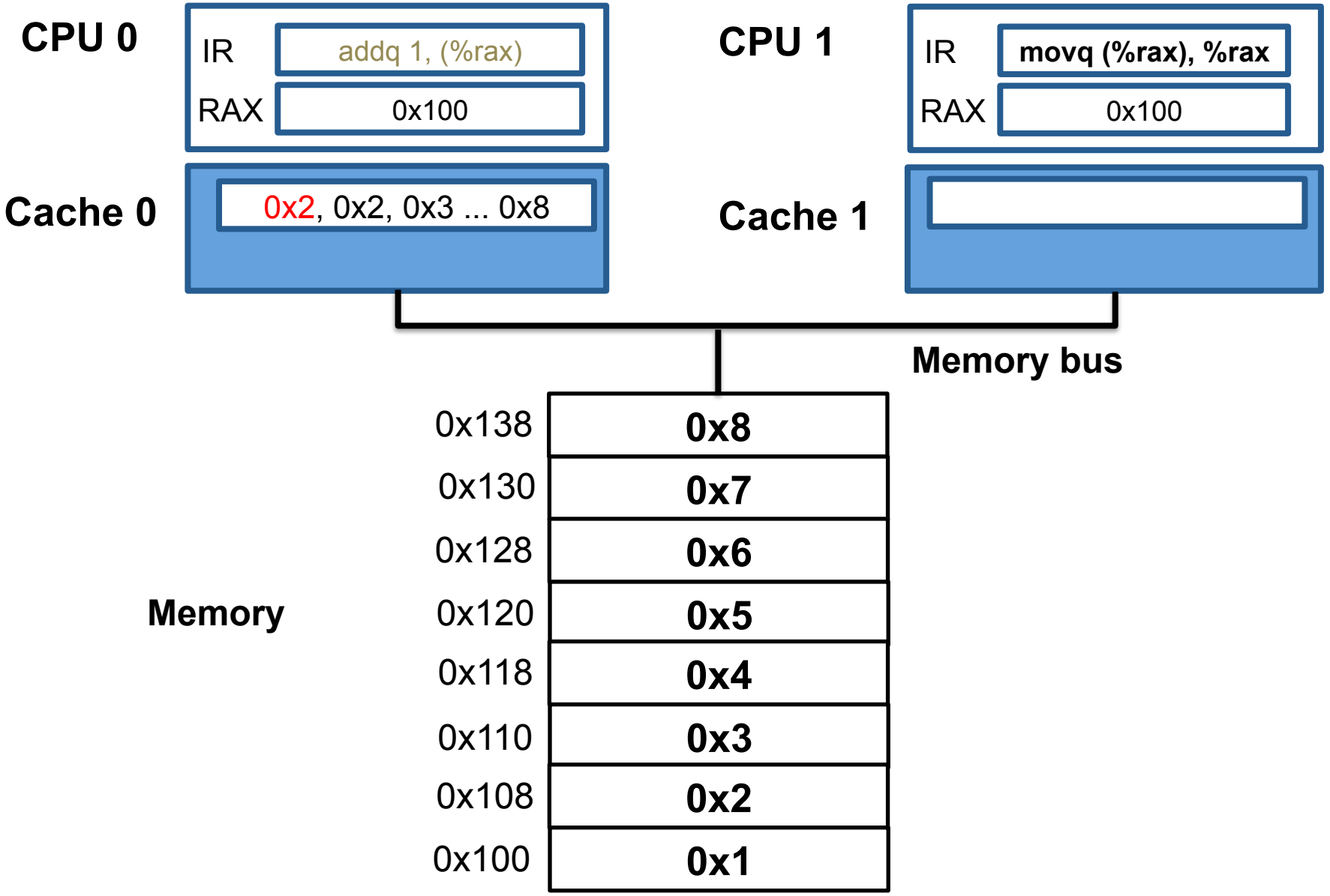
Basic Idea of Cache Coherence



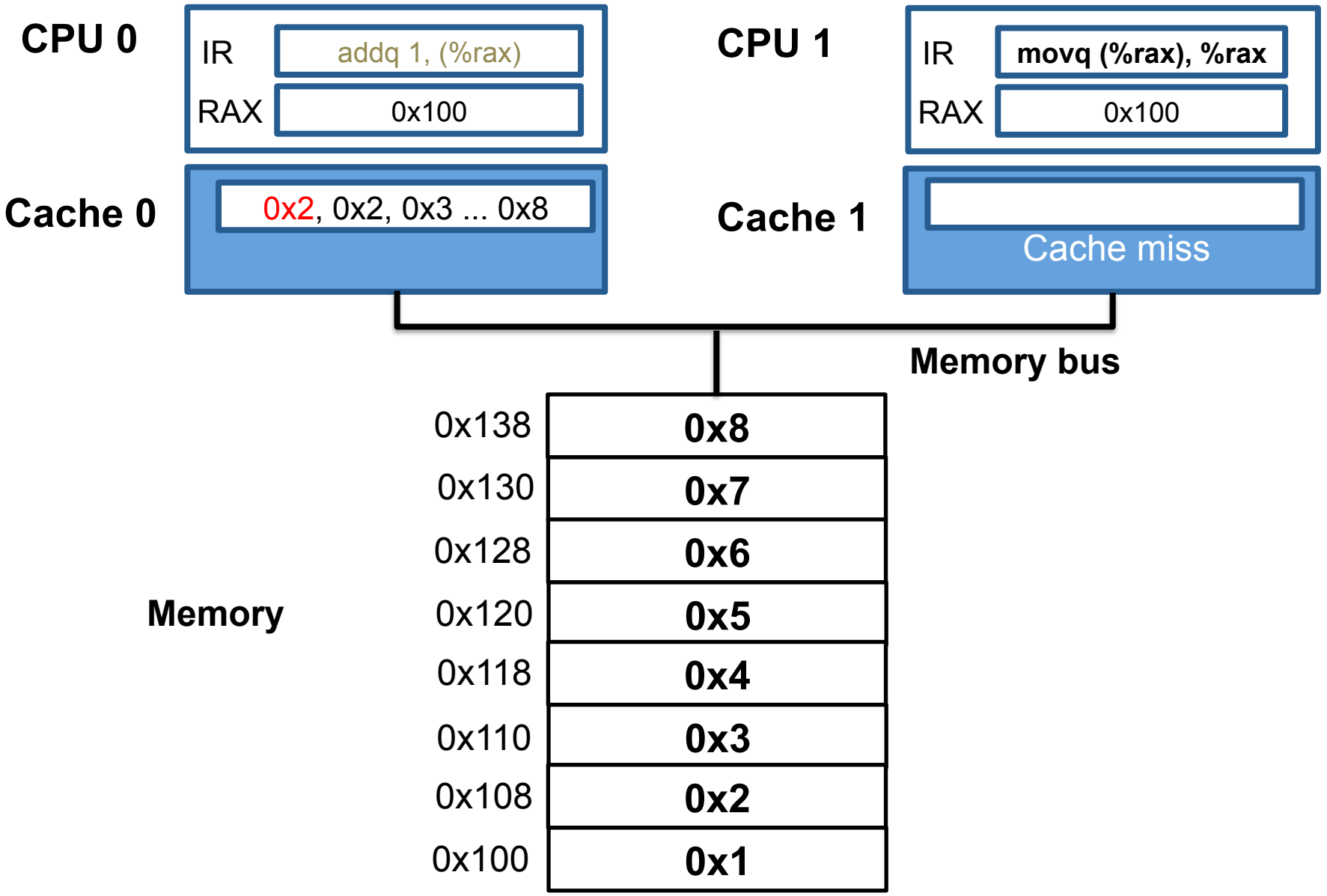
Basic Idea of Cache Coherence



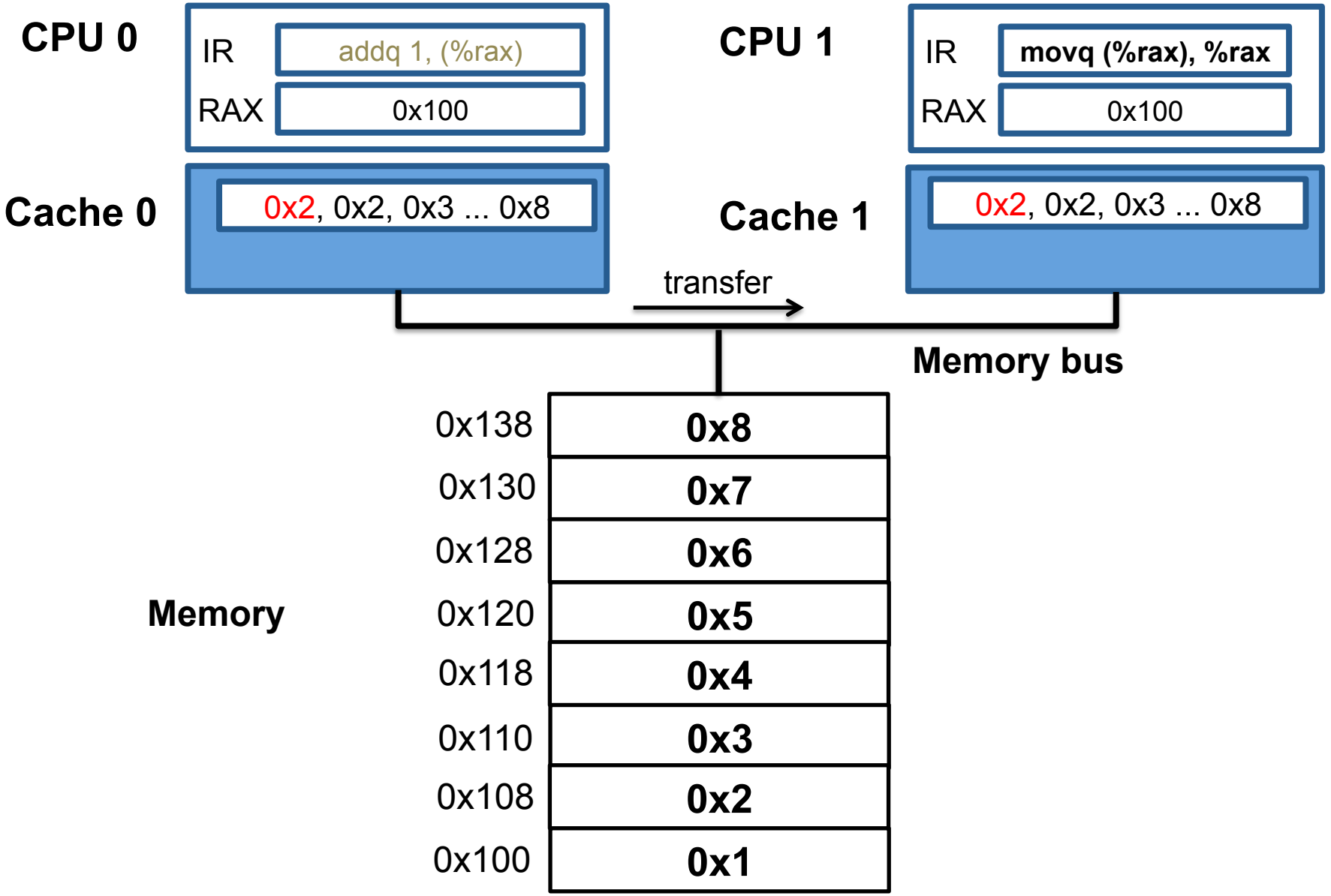
Basic Idea of Cache Coherence



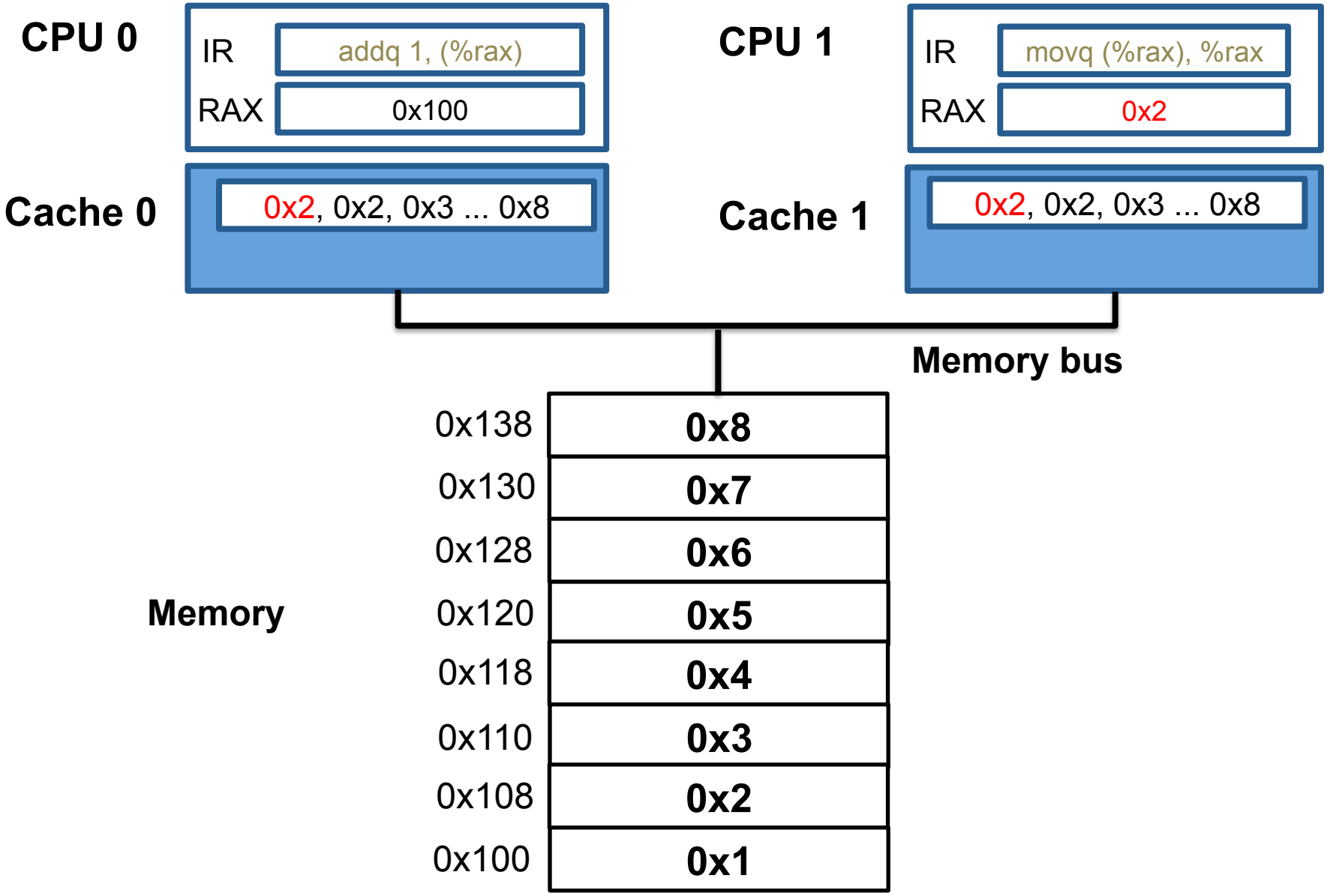
Basic Idea of Cache Coherence



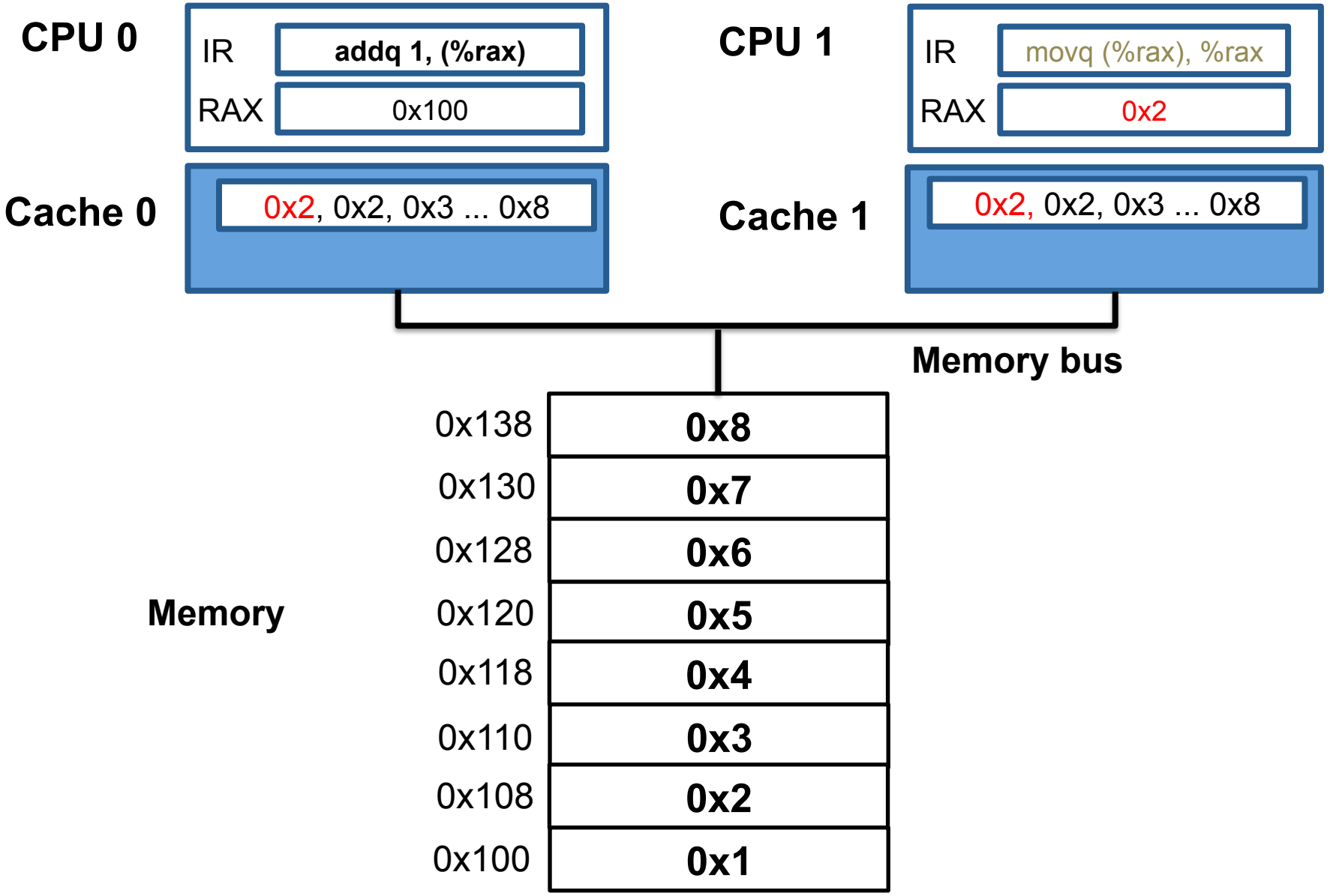
Basic Idea of Cache Coherence



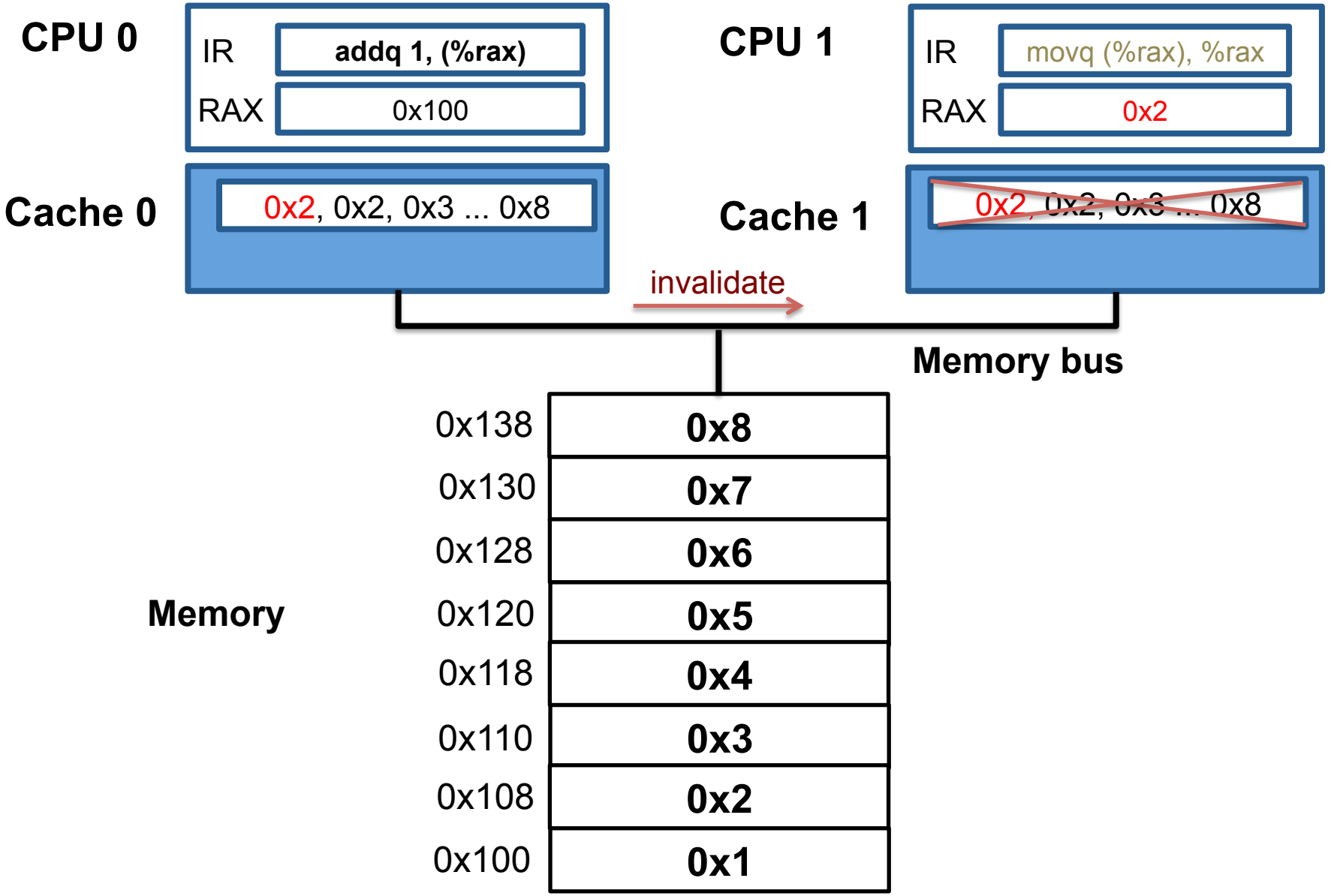
Basic Idea of Cache Coherence



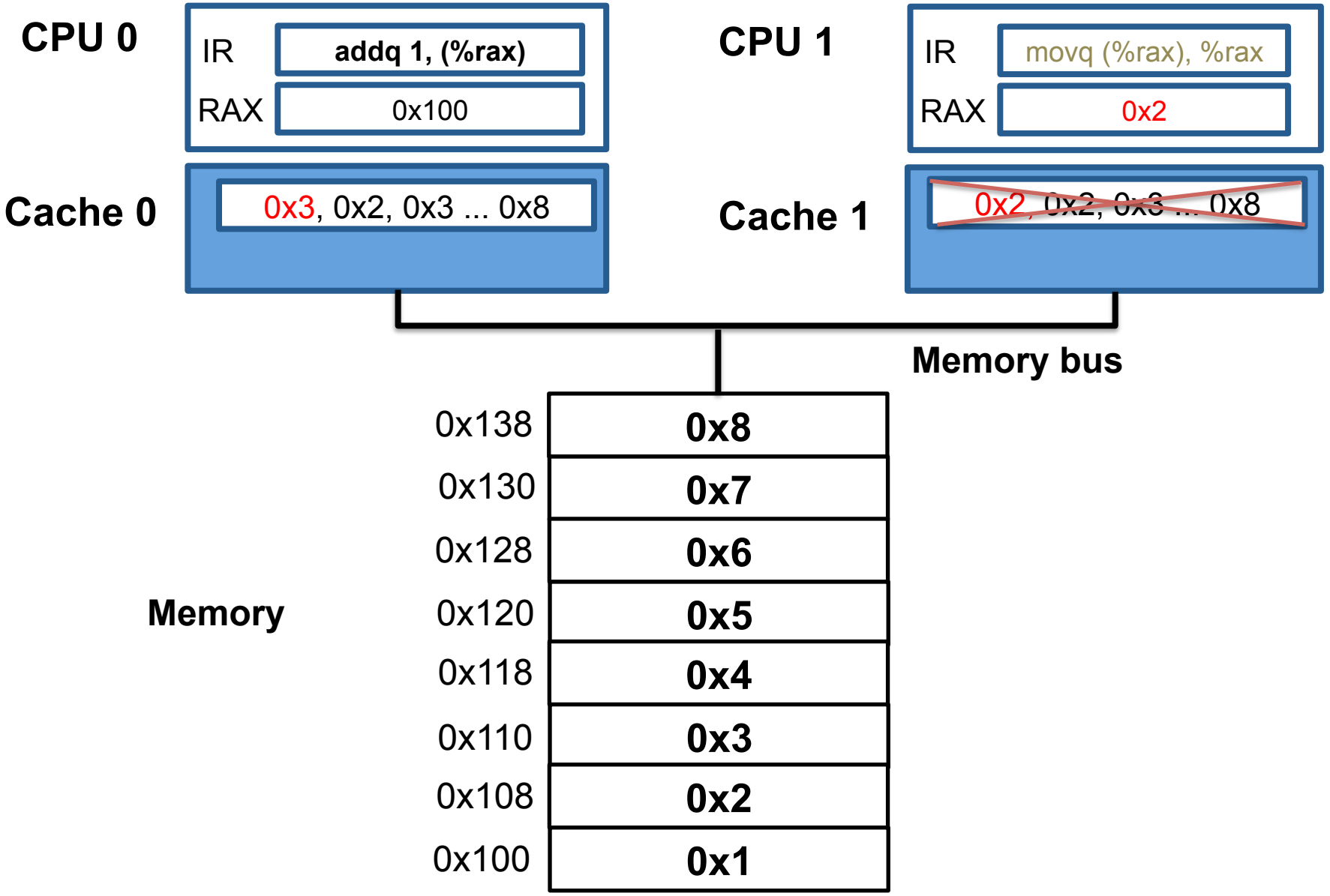
Basic Idea of Cache Coherence



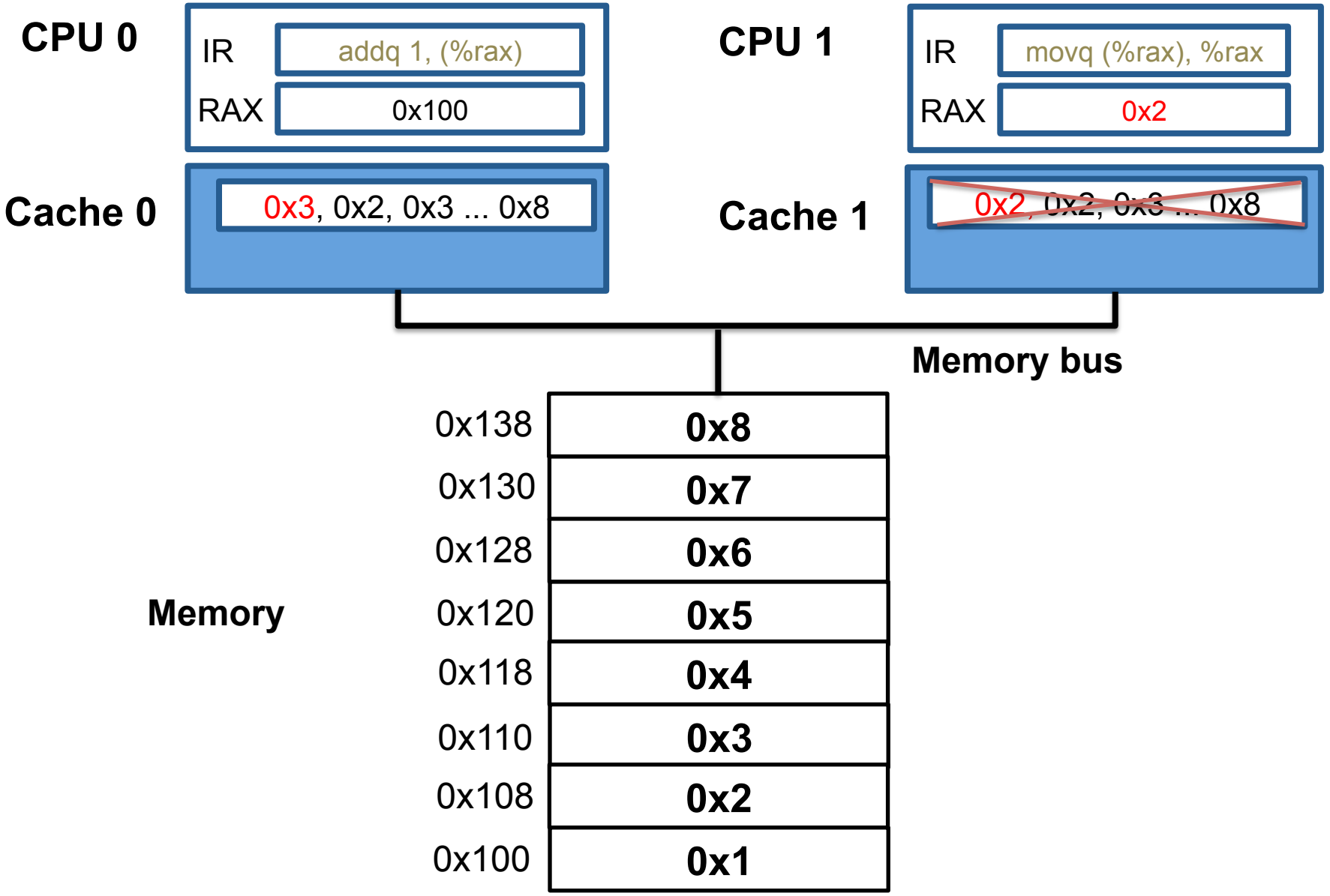
Basic Idea of Cache Coherence



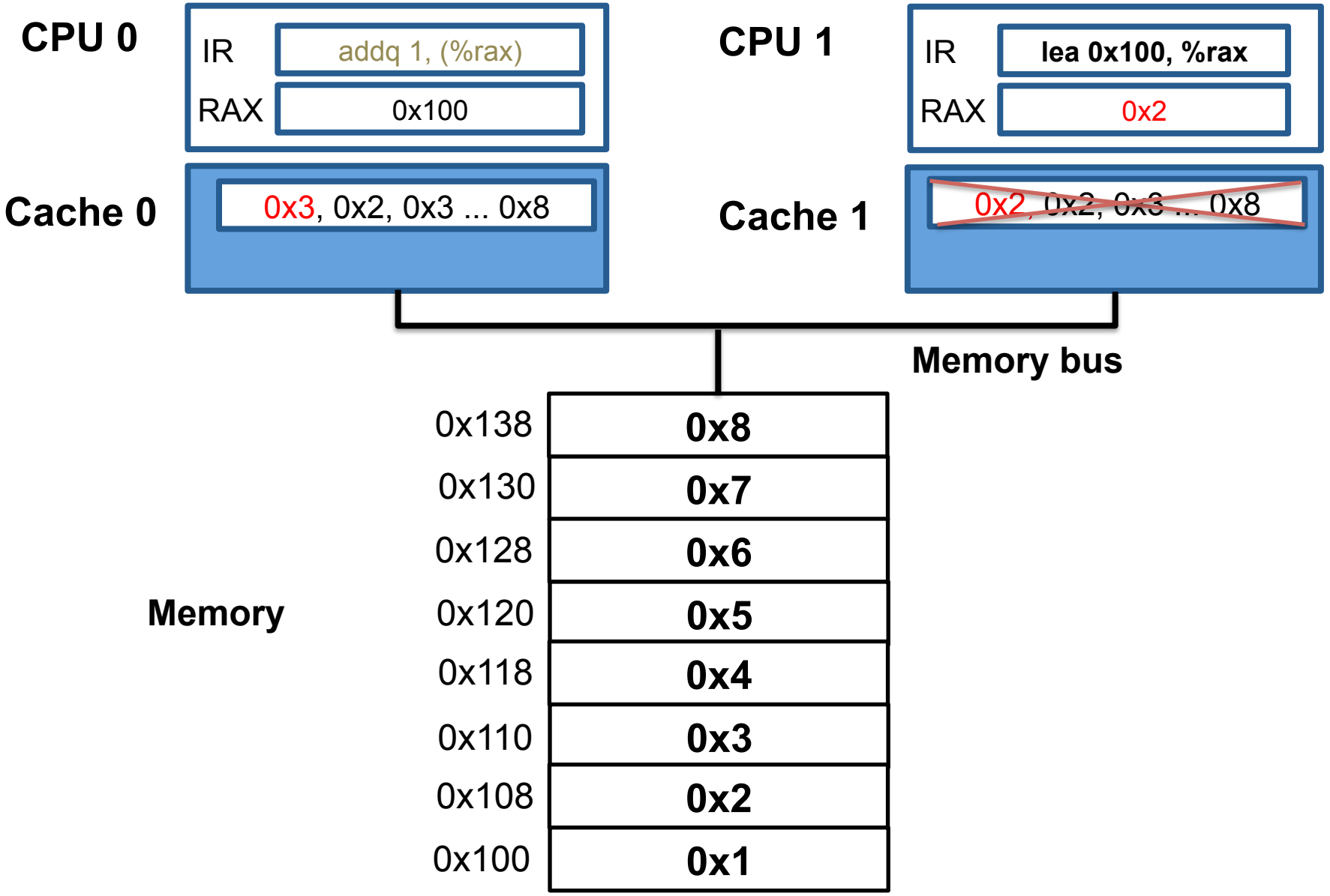
Basic Idea of Cache Coherence



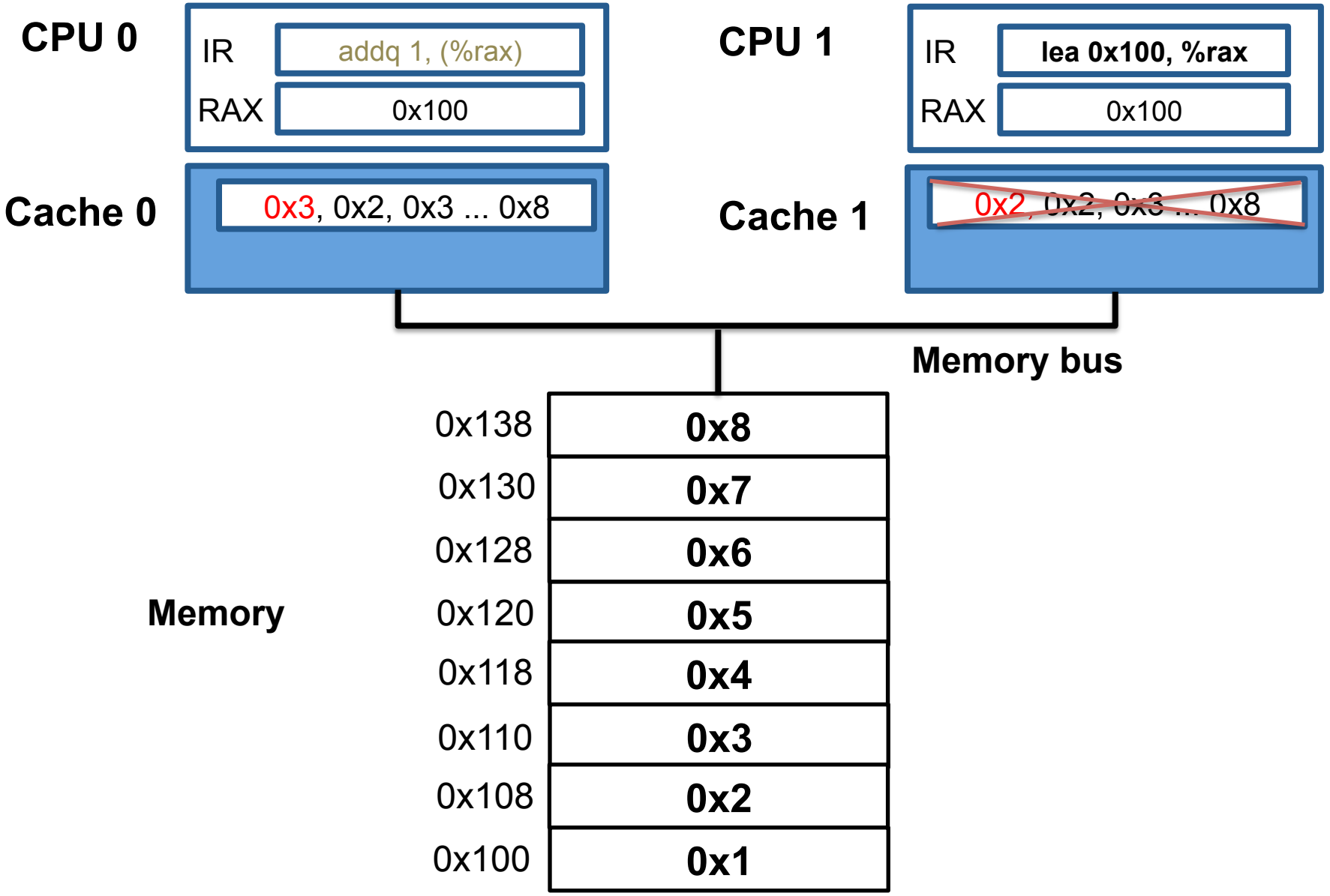
Basic Idea of Cache Coherence



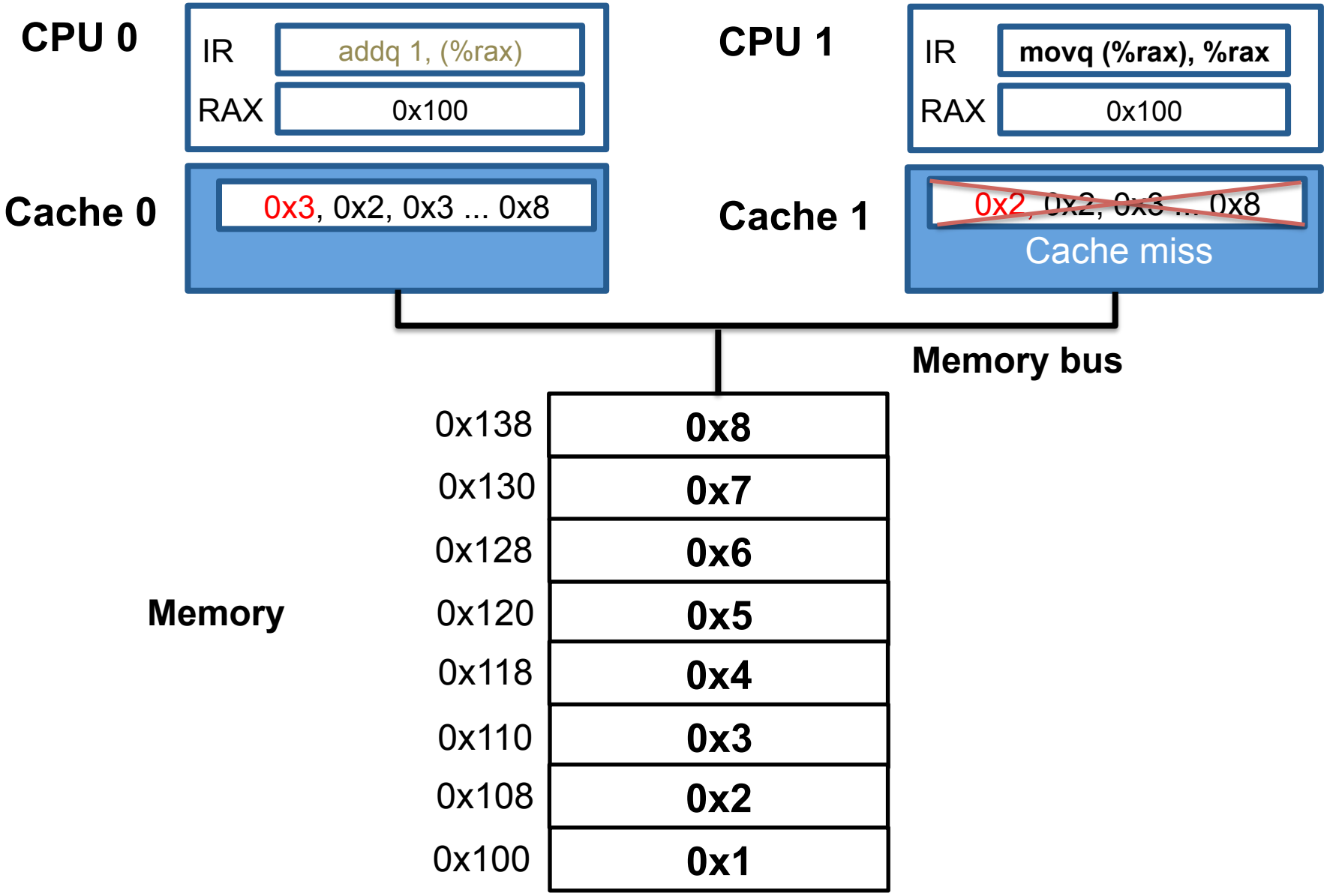
Basic Idea of Cache Coherence



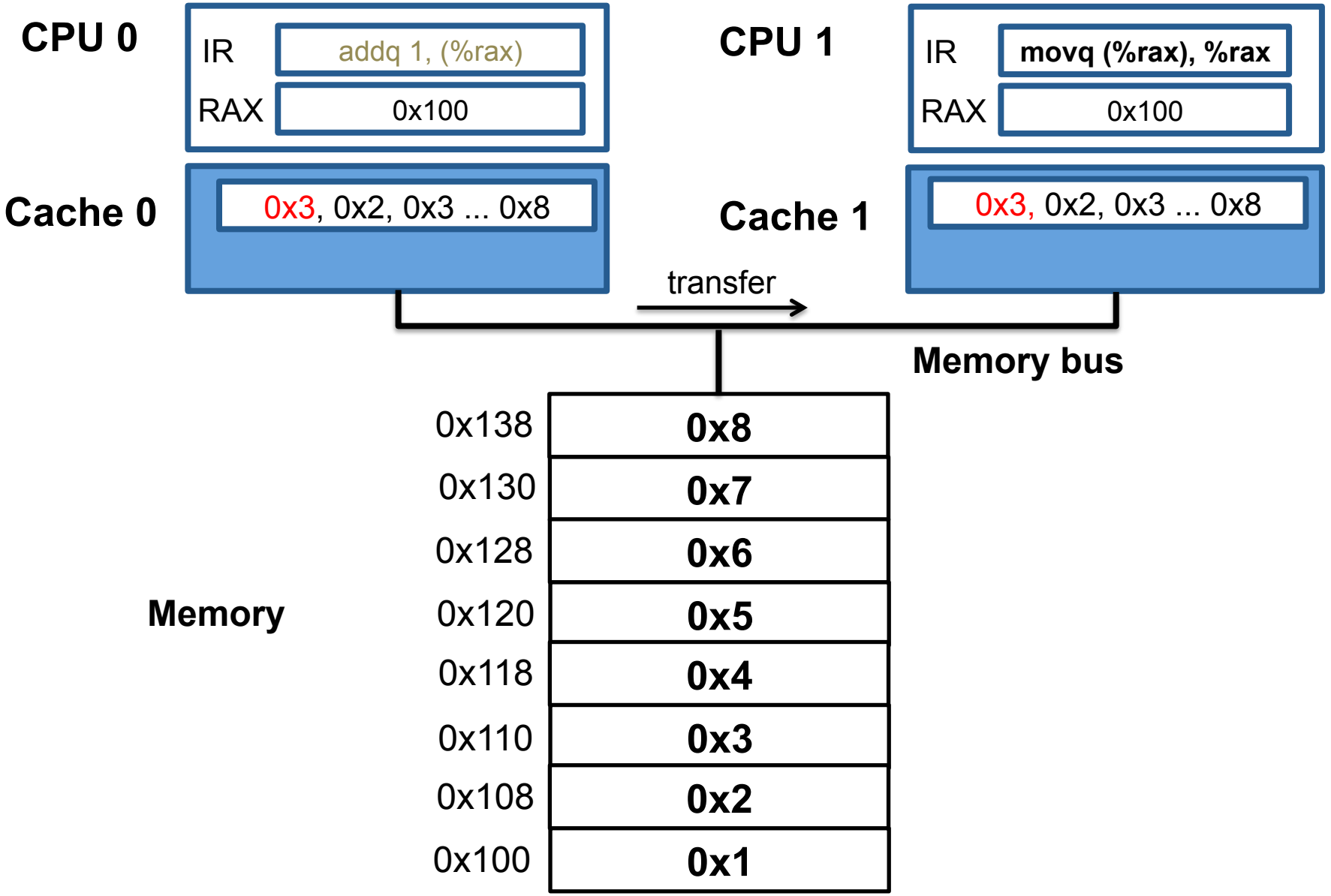
Basic Idea of Cache Coherence



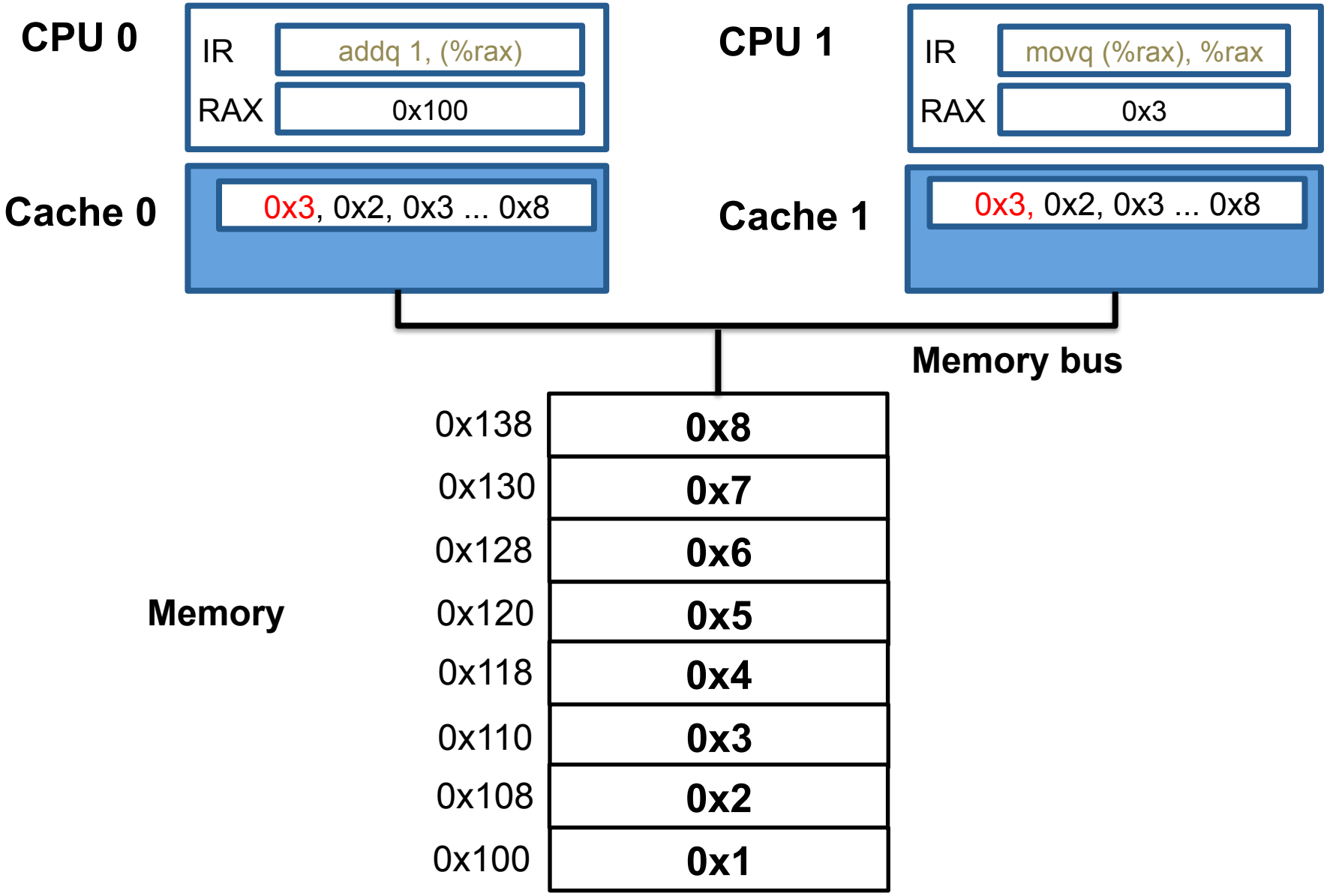
Basic Idea of Cache Coherence



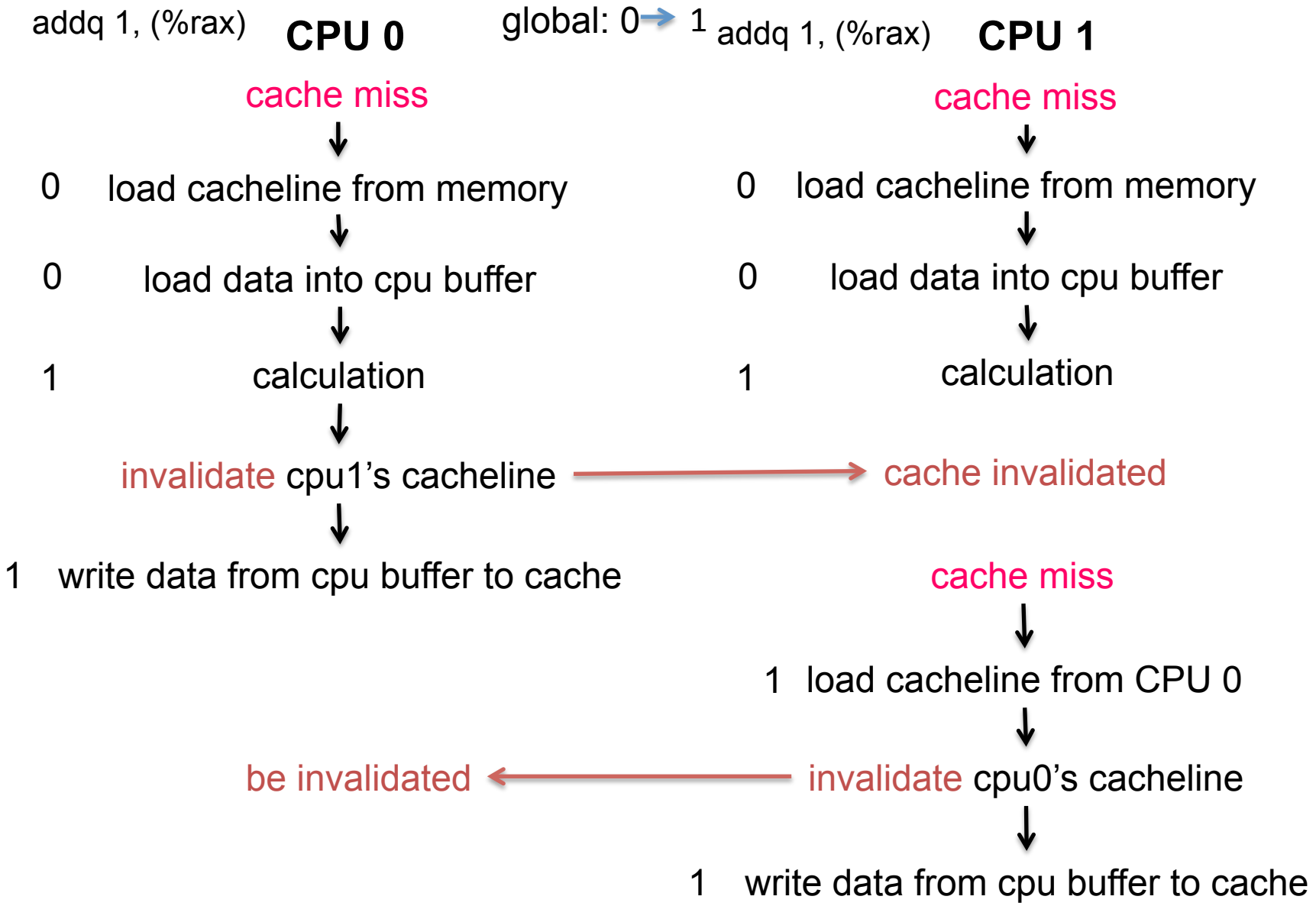
Basic Idea of Cache Coherence



Basic Idea of Cache Coherence



Update global variable



Update global variable with lock

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

global: 2

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 calculate

↓

write data from cpu buffer to cache

↓

Instruction retired

cache miss

↓

load cacheline from CPU0

↓

load into CPU buffer and calculate

↓

invalidate cpu0's cacheline

←

cache invalidated

↓

write data from cpu buffer to cache

↓

Instruction retired

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

Total Cycles / Total Operations

- Synchronizing per add makes multi-threading slower than single thread
- Synchronization magnifies the cost of cache coherence

A brief note about lab 5

- How to implement a read-write lock?

```
typedef struct {
  ...
} rwl;
```

```
void rwl_init(rwl *l);
int rwl_nwaiters(rwl *l);
int rwl_rlock(rwl *l, struct timespec *expire);
int rwl_runlock(rwl *l);
int rwl_wlock(rwl *l, struct timespec *expire);
int rwl_wunlock(rwl *l);
```

Implementing a read-write lock

- Need to track mode of the grabbed lock?
  - read (“shared”) vs. write (“exclusive”)
- Need to track how many readers have the lock?
  - Multiple readers can grab lock in “read” mode
- Need to track waiting threads (waiters)?
  - If there are waiters, we should wake them up upon lock release
  - Shall we track waiting writers and readers separately?
    - Lab requires you to prioritize writer, i.e. a waiting writer should get the lock over waiting readers

A brief note about lab 5

- How to implement a read-write lock? An example.

```
typedef struct {

  int n_waiting_readers;
  int n_waiting_writers;
  int n_writers;
  int n_readers;
  pthread_mutex_t mu;
  pthread_mutex_t cond;

} rwl;
```


What's the state of the lock if it's locked on "exclusive" mode?

What's the state of the lock if it's locked on "shared" mode?

An example rwl_wlock: it prioritizes readers instead of writers

```
int rwl_wlock(rwl *l, struct timespec *expire) {
    pthread_mutex_lock(&l->mu);
    l->n_waiting_writers++;
    //if lock has been locked, block
    //if lock has waiting readers, also block
    while (l->n_waiting_readers > 0
               || l->n_writers > 0
               || l->n_readers > 0) {

        pthread_cond_wait(&l->cond, &l->mu);

    }
    //update lock state
     l->n_waiting_writers--;
     l->n_writers++;
    pthread_mutex_unlock(&l->mu);
}
```

An example rwl_wlock: it prioritizes readers instead of writers

```
int rwl_wlock(rwl *l, struct timespec *expire) {
    pthread_mutex_lock(&l->mu);
    l->n_waiting_writers++;
    while (l->n_waiting_readers > 0
           || l->n_writers > 0
           || l->n_readers > 0) {

       if (cond_timedwait(&l->cond,&l->mu, expire)==ETIMEDOUT){
           l->n_waiting_writers--;
           pthread_mutex_unlock(&l->m);
           return ETIMEDOUT;
       }

    }
    //update lock state
    l->n_waiting_writers--;
    l->n_writers++;
    pthread_mutex_unlock(&l->mu);
}
```