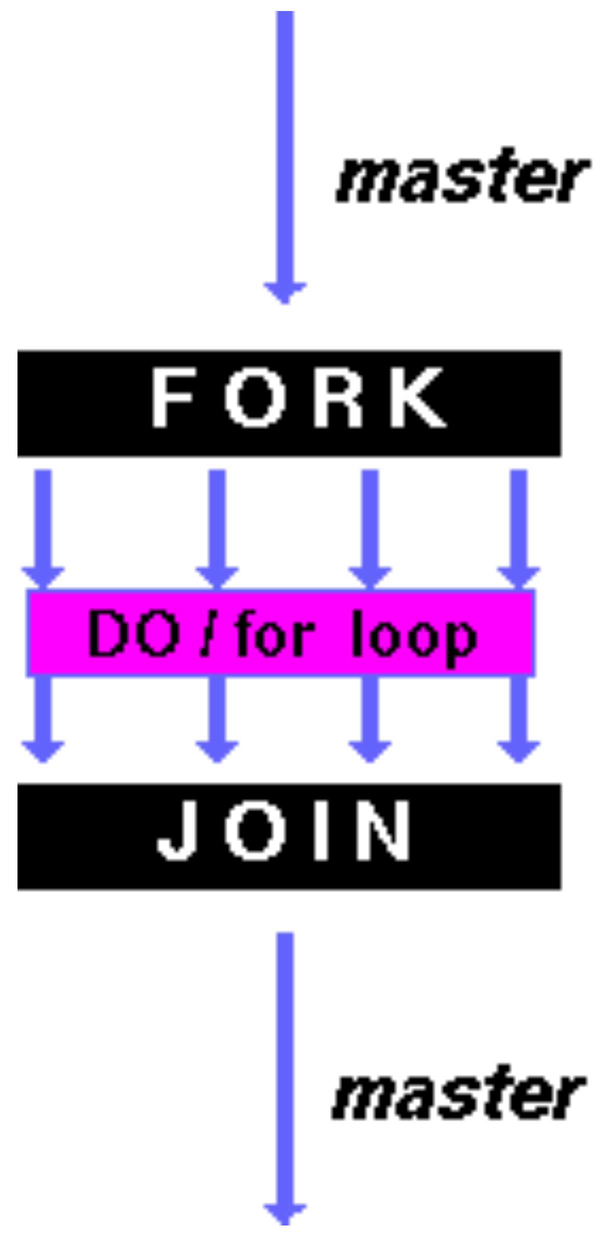
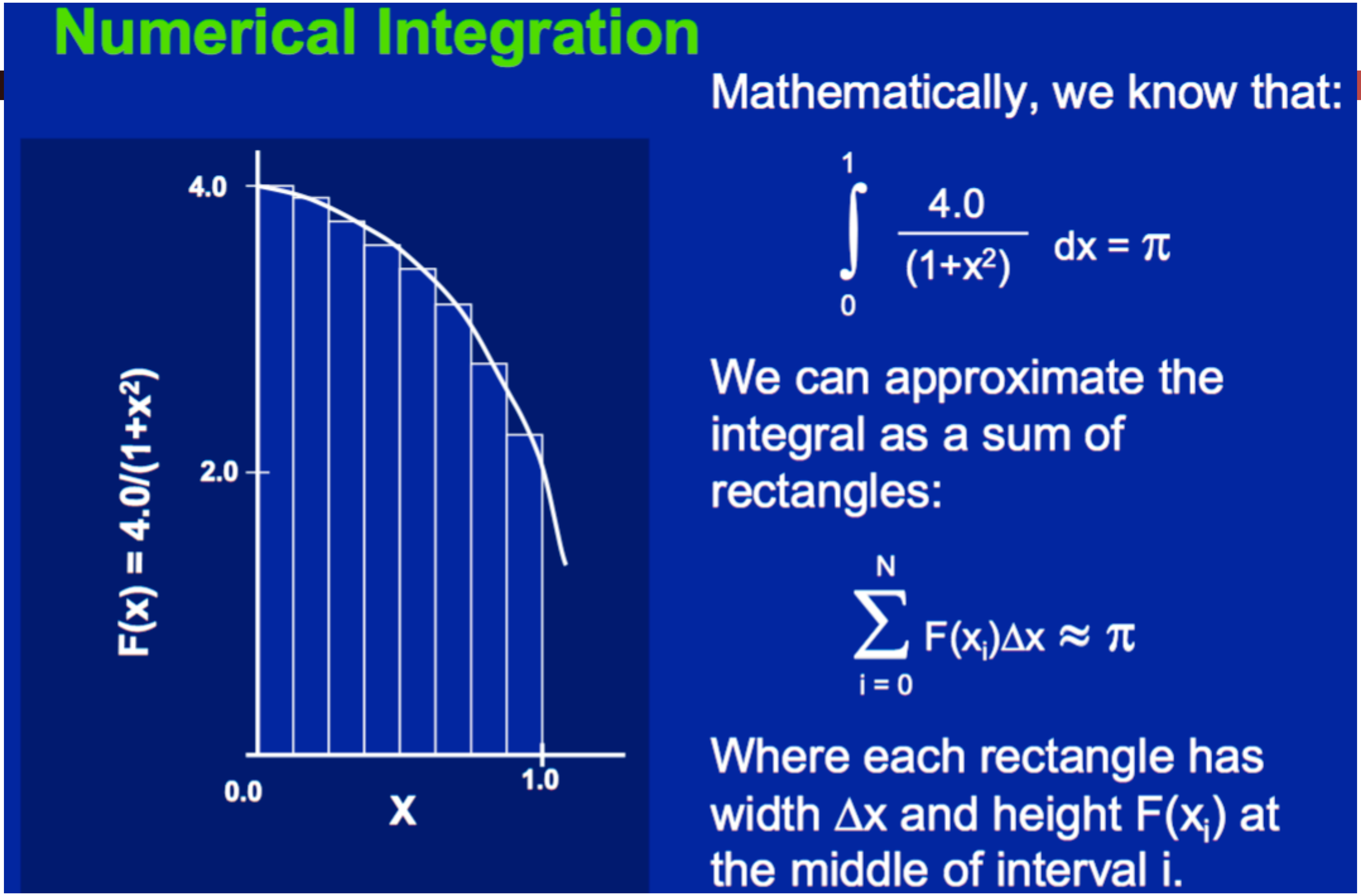
More On Synchronization And Threading

OpenMP `parallel for` pragma

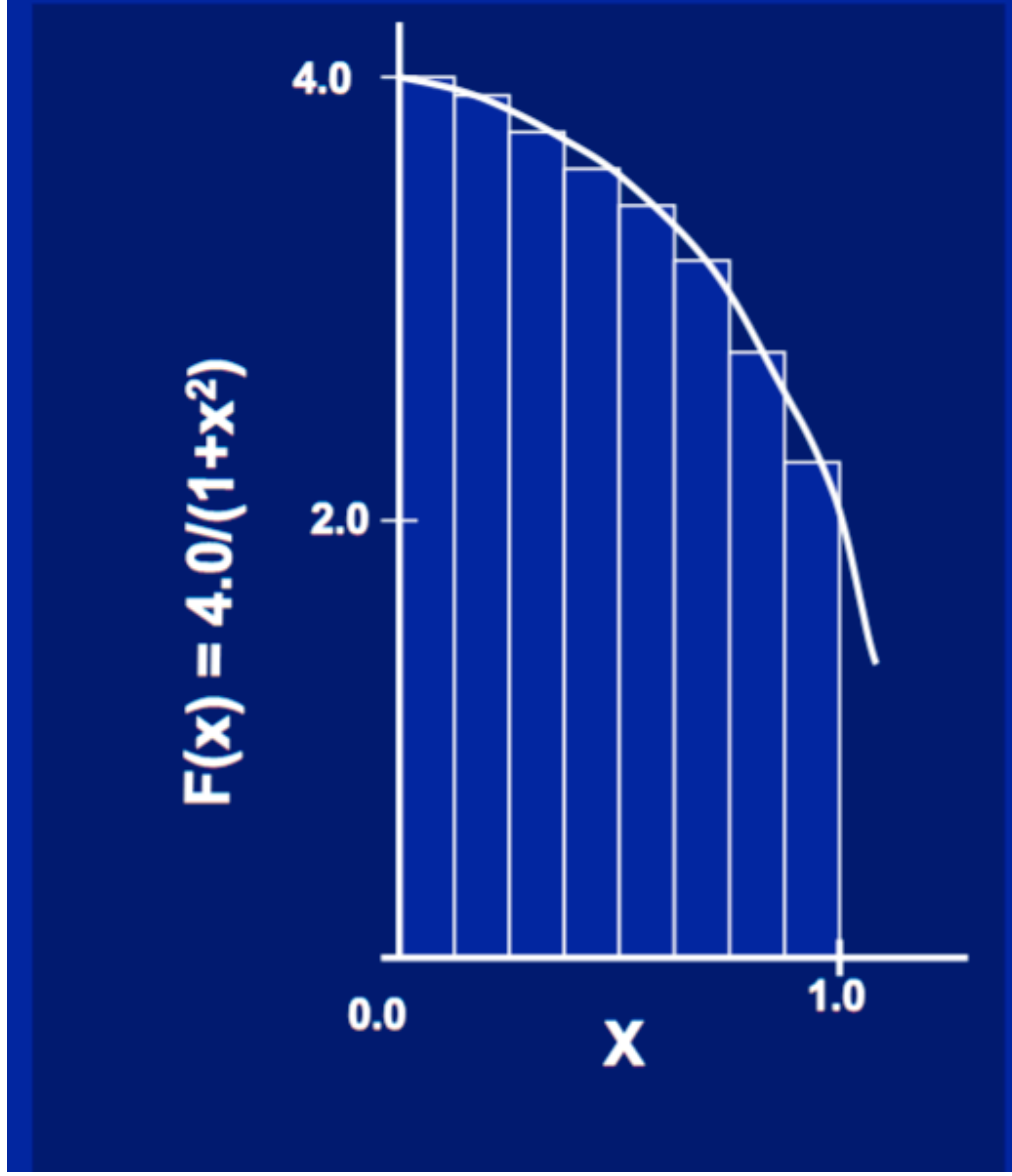
-
  ```
  #pragma omp parallel for
  for (i=0; i<max; i++) zero[i] = 0;
  ```
- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is ***implicitly*** private per thread
- Implicit "barrier" synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, …, (max/n)-1;
  - Thread 1 gets max/n, max/n+1, …, 2*(max/n)-1



Example 2: Computing π



Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

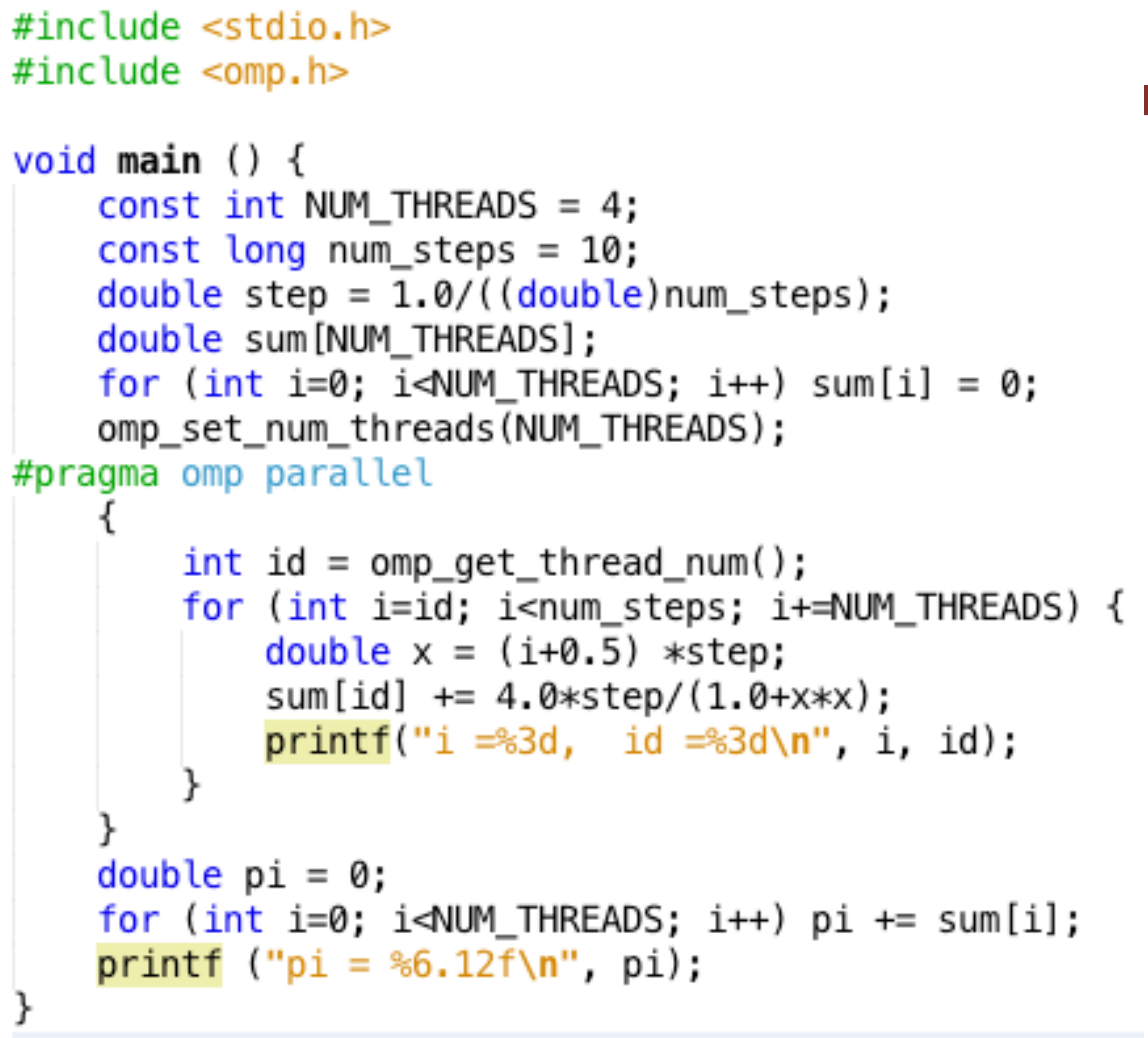
We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

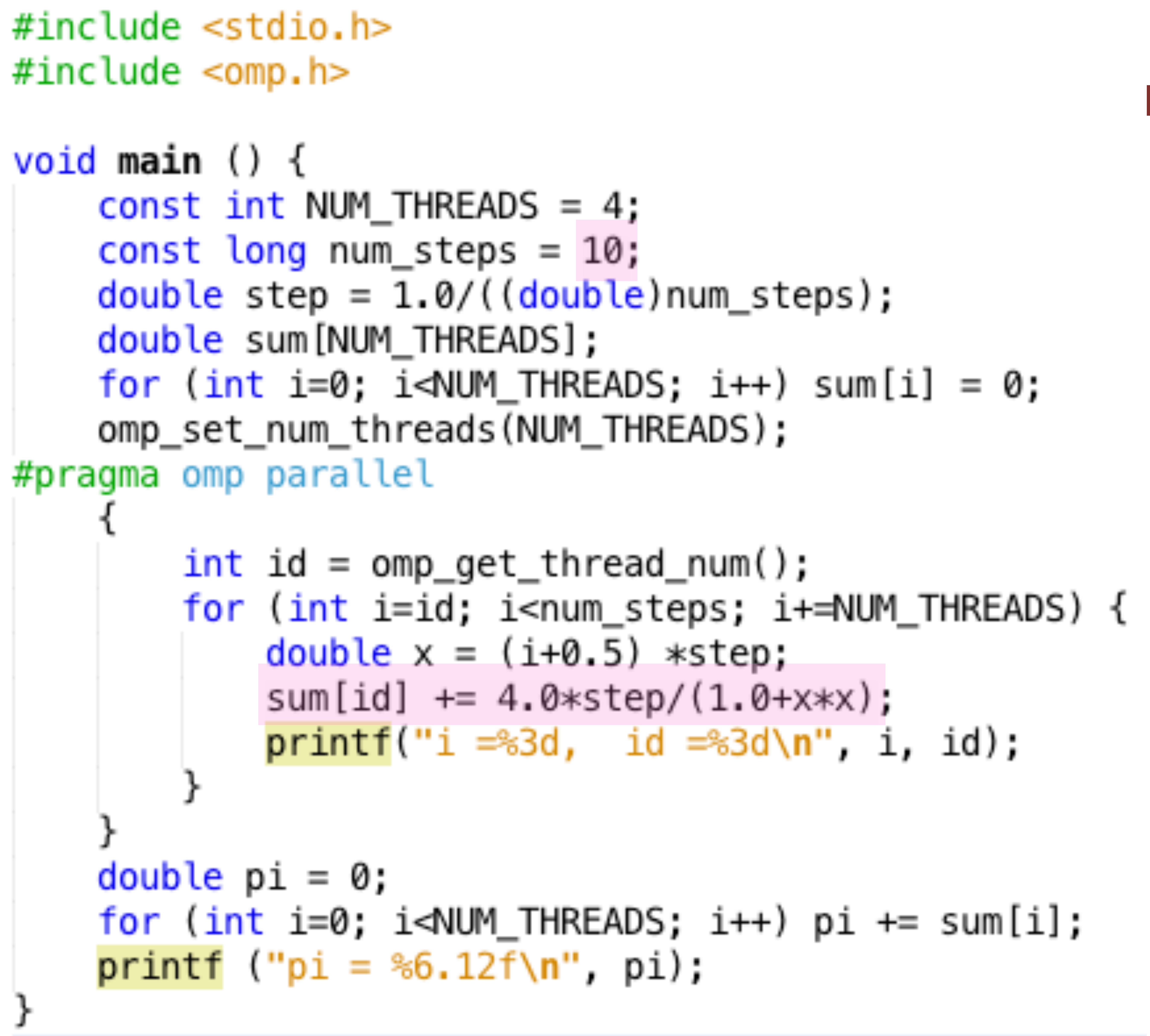
http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Working Parallel π without a for loop



```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

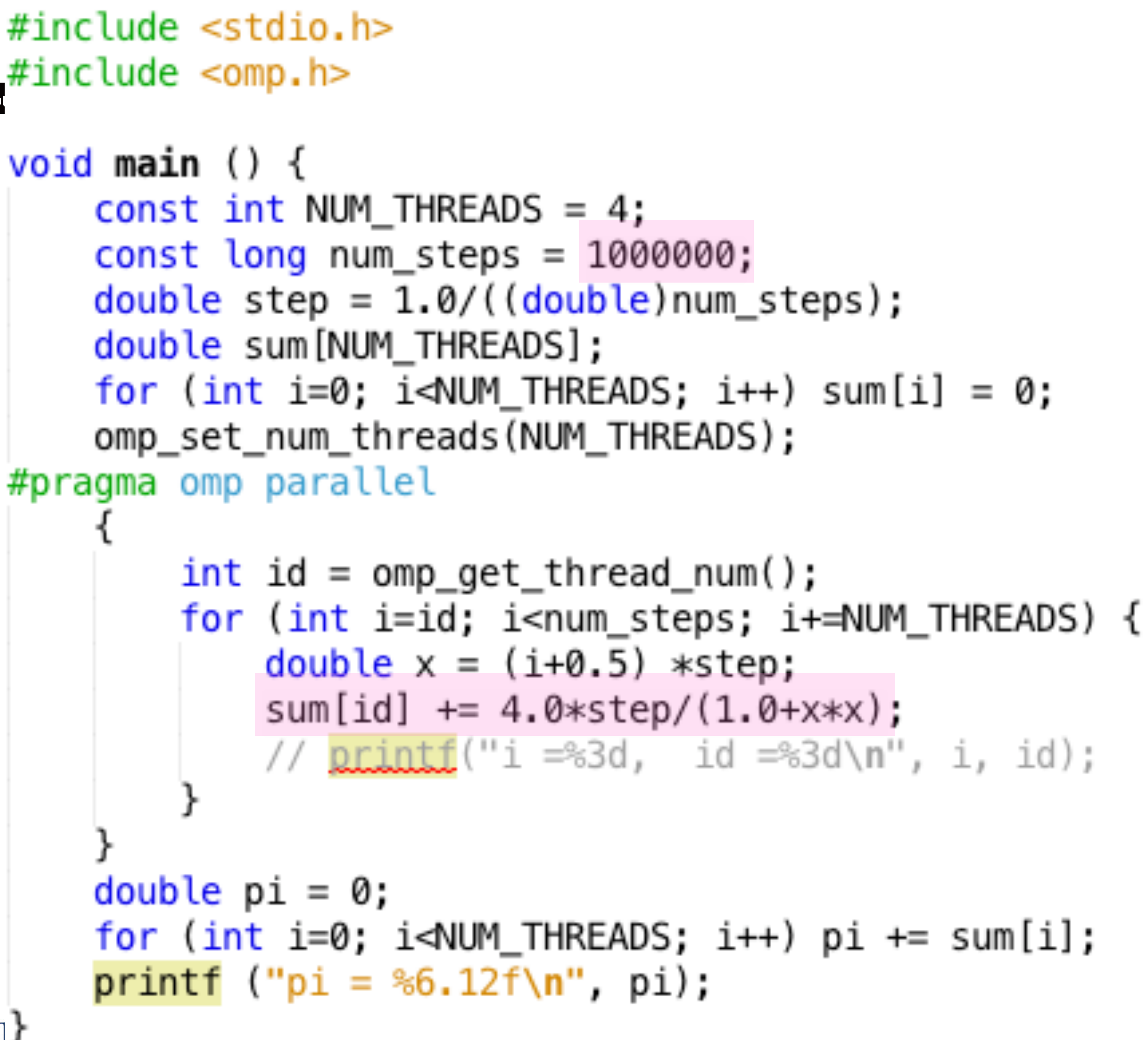
Trial Run



```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.142425985001
```

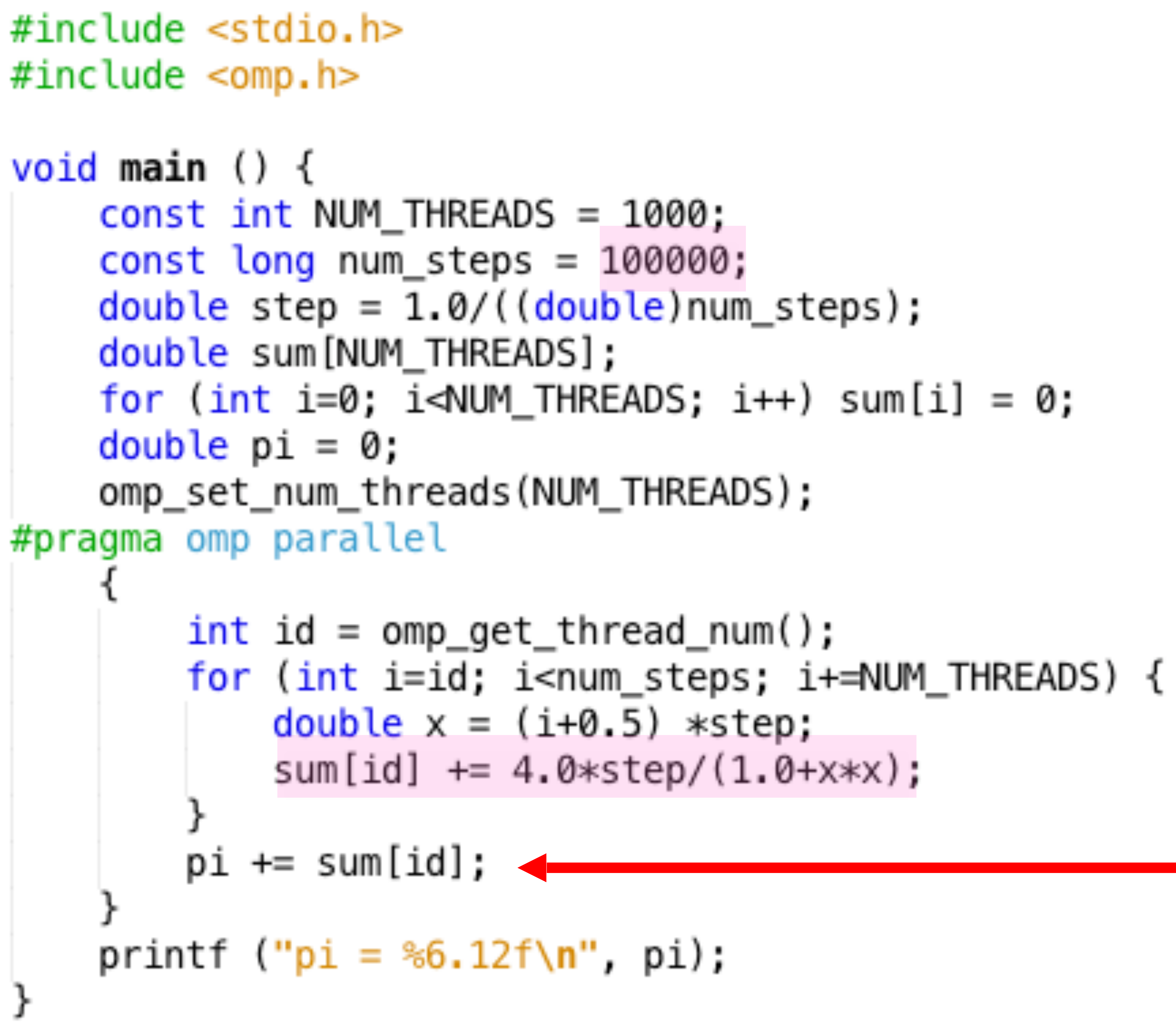
Scale up: num_steps = 10^6



```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
pi = 3.141592653590
```

Can We Parallelize Computing **`sum`**?



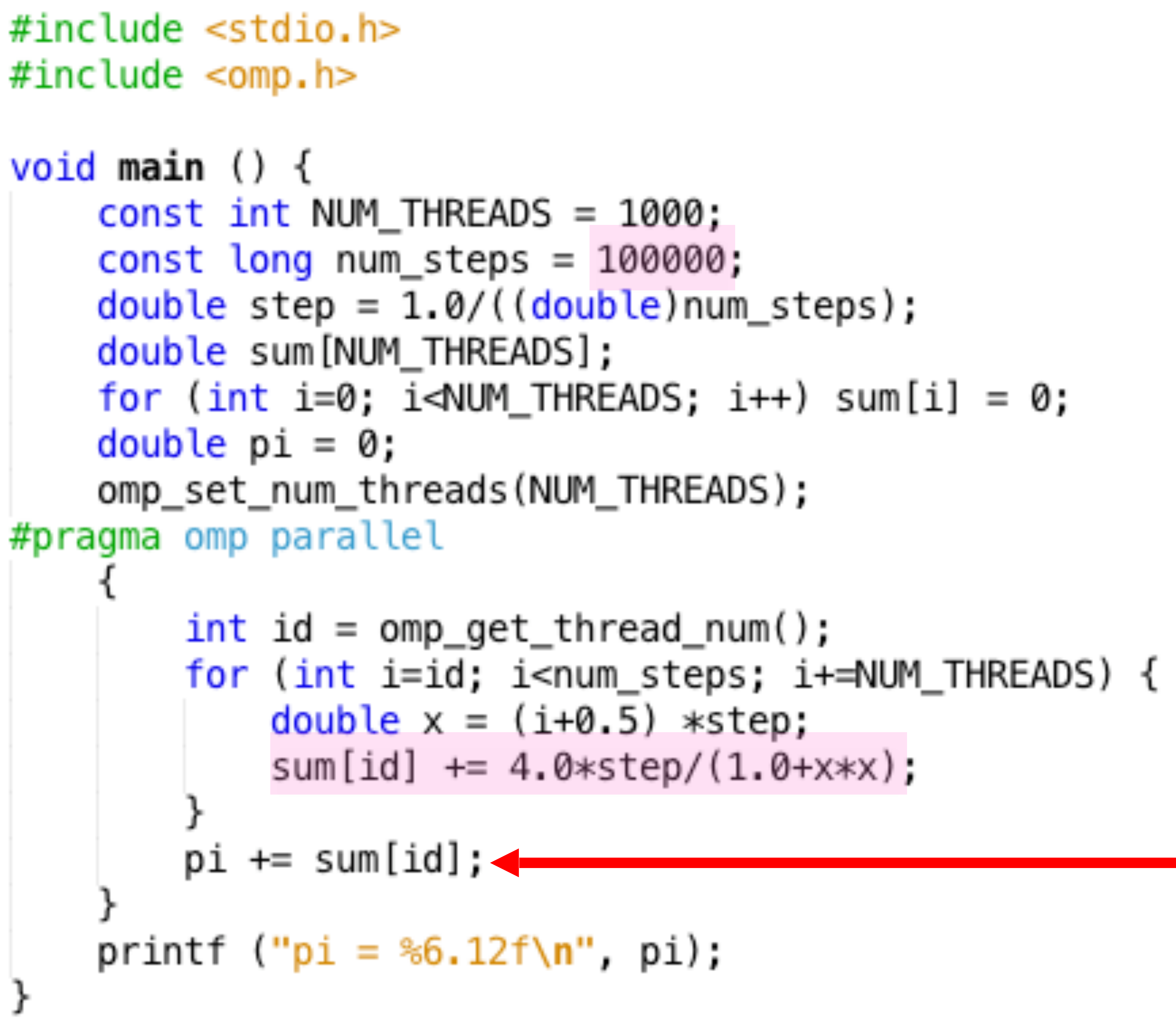
```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section

- Insignificant speedup in this example, but …
- **`pi = 3.138450662641`**
- Wrong! And value changes between runs?!
- What's going on?

What’s Going On?



```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5)*step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

- Operation is really
  `pi = pi + sum[id]`
- What if >1 threads reads current (same) value of **`pi`**, computes the sum, stores the result back to **`pi`**?
- Each processor reads same intermediate value of **`pi`**!
- Result depends on who gets there when
  - A “race” → result is **not deterministic but if we locked this we'd lose almost all speedup**

OpenMP Reduction

-
  ```
  double avg, sum=0.0, A[MAX]; int i;
  #pragma omp parallel for private ( sum )
  for (i = 0; i <= MAX ; i++) {sum += A[i];}
  avg = sum/MAX;  // bug, we only get the master thread's sum
  ```
- Problem is that we really want sum over all threads!
- ***Reduction***: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region:
  reduction(operation:var) where
  - Operation: operator to perform on the variables (var) at the end of the parallel region
  - Var: One or more variables on which to perform scalar reduction: private than combined

-
  ```
  double avg, sum=0.0, A[MAX]; int i;
  #pragma omp for reduction(+ : sum)
  for (i = 0; i <= MAX ; i++) {sum += A[i];}
  avg = sum/MAX;
  ```

Calculating π Simple Version

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
  int i;     double  x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  #pragma omp parallel private ( i, x )
  {
    int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i<num_steps; i=i+NUM_THREADS)
    {
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for(i=1; i<NUM_THREADS; i++)
    sum[0] += sum[i];  pi = sum[0];
  printf ("pi = %6.12f\n", pi);
}
```

Version 2: parallel for, reduction

```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{    int i;      double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
     for (i=1; i<= num_steps; i++){
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
     }
     pi = sum;
     printf ("pi = %6.8f\n", pi);
}
```

Reduction Options...

- Arithmetic
  - `+ * -`
- Comparison
  - **`min max`**
- Logical
  - `& && | || ^`
- And now you know why RISC-V has the atomic memory operations:
  - **`amoadd`**, **`amoand`**, **`amoor`**, **`amoxor`**, **`amomax`**, **`amomin`**
    - All but - and * as the reduction can be implemented as single instructions

OpenMP Timing

- Elapsed wall clock time:

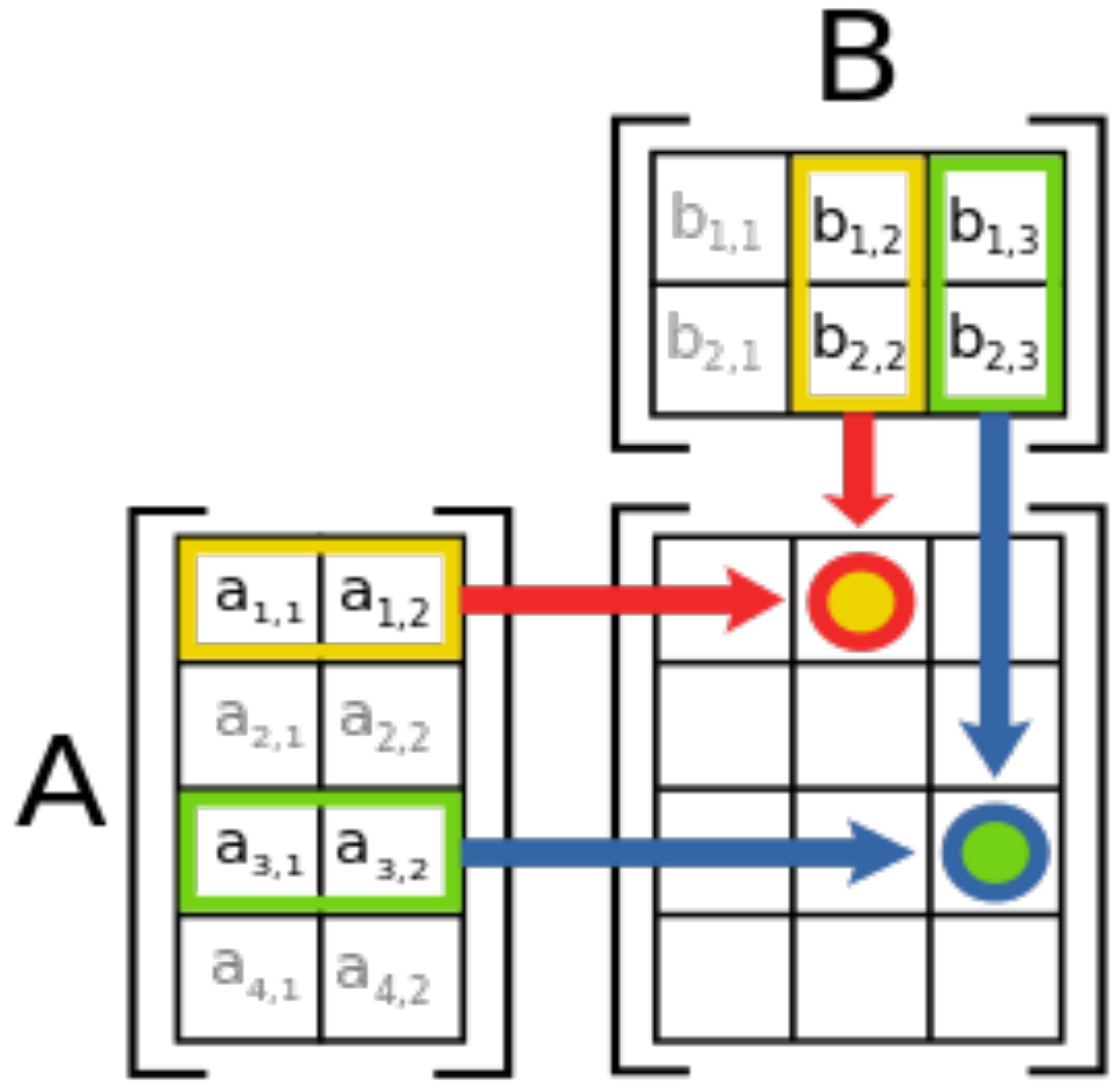
  ```
  double omp_get_wtime(void);
  ```

  - Returns elapsed wall clock time in seconds
  - Time is measured per thread, no guarantee can be made that two distinct threads measure the same time
  - Time is measured from "some time in the past," so subtract results of two calls to **`omp_get_wtime`** to get elapsed time

Matrix Multiply in OpenMP

```
// C[M][N] = A[M][P] × B[P][N]
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, j, k)
  for (i=0; i<M; i++){
    for (j=0; j<N; j++){
      tmp = 0.0;
      for (k=0; k<P; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += A[i][k] * B[k][j];
      }
      C[i][j] = tmp; // doing this to tmp prevents
       // potential cache issues from multiple
       // writers & "false sharing"
    }
  }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across N threads;
inner loops inside a single thread



Matrix Multiply in Open MP

- More performance optimizations available:
  - Higher compiler optimization (-O2, -O3) to reduce number of instructions executed
  - Cache blocking to improve memory performance
  - Using SIMD AVX instructions to raise floating point computation rate (DLP)
- Gives a guide for Project 4:
  - 1: Get it working period
  - 2: OpenMP Parallel `for` the outer loop
    - Make sure the different threads are writing to different parts of memory (CRITICAL!)
  - 3: Make the inner loop SIMD vectorized
  - 4: Make the inner loop unrolled x4
  - 5: Add blocking (~32 or 64 sounds like a good number)

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads access same location, at least one is a write, and they occur one after another
- If there is a data race, result of program varies depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get *deterministic* behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions

Reminder: Locks

- Computers use locks to control access to shared resources
  - Serves purpose of microphone in example
  - Also referred to as "semaphore"
    - Although "semaphores" have slightly different semantics
- Usually implemented with a variable
  - In most object-oriented languages its an object that you can call
  - Under the hood its usually just an integer that has atomic updates:
    0 == unlocked, 1 == locked
- Two operations:
  - lock.acquire() // blocks this thread until the lock is unlocked
  - lock.release() // Unlocks the lock
- OpenMP also has a lock

  -
    ```
    #pragma omp critical
    {
    ... Only one thread is running at a time here
    }
    ```

Deadlock

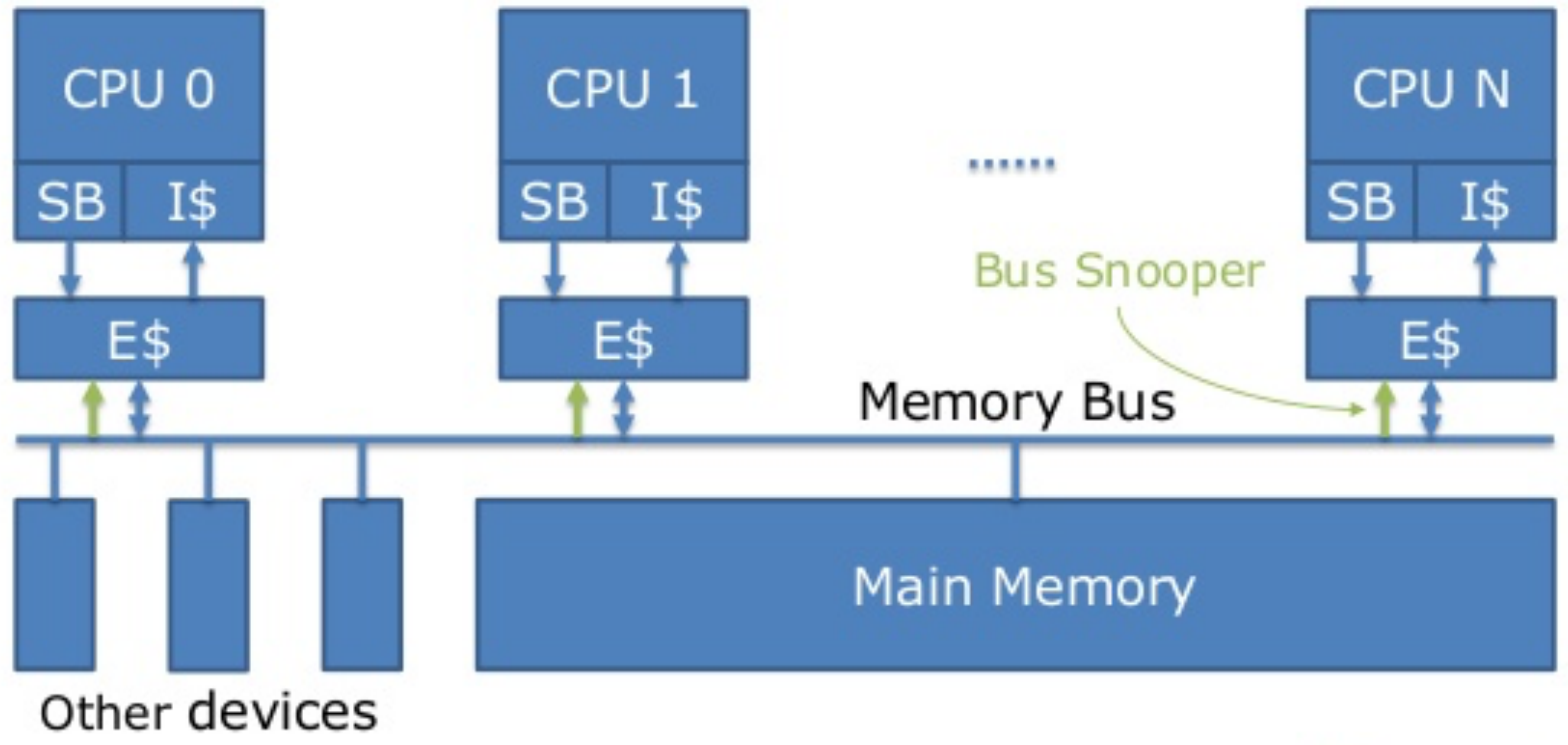
- Deadlock: a system state in which no progress is possible because everything is locked waiting for something else
- Dining ~~Philosopher's~~ Lawyers Problem:
    - Pontificate until the left fork is available; when it is, pick it up
    - Pontificate until the right fork is available; when it is, pick it up
    - When both forks are held, eat for a fixed amount of time
    - Then, put the right fork down
    - Then, put the left fork down
    - Repeat from the beginning
- Solution?

Limiting Parallelism

- Locks act to inhibit parallelism
  - Innately sequential regions -> Amdahl's law problem...
  - Python is "threaded" but not really:
    A global interpreter lock means other threads can be waiting on I/O, but (mostly) only one compute thread at a time
- Can try to limit the locks
  - Rather than locking everything have a different lock for each region...
    - But be careful, then its much easier to get into deadlock situations
- Or try to eliminate locks altogether
  - Thus, e.g. why Go's parallelism is not focused around locks, and RISC-V has the **`lr/sc`** instructions

(Chip) Multicore Multiprocessor

- SMP: (Shared Memory) Symmetric Multiprocessor
  - Two or more identical CPUs/Cores
  - Single shared coherent memory



Multiprocessor Key Questions

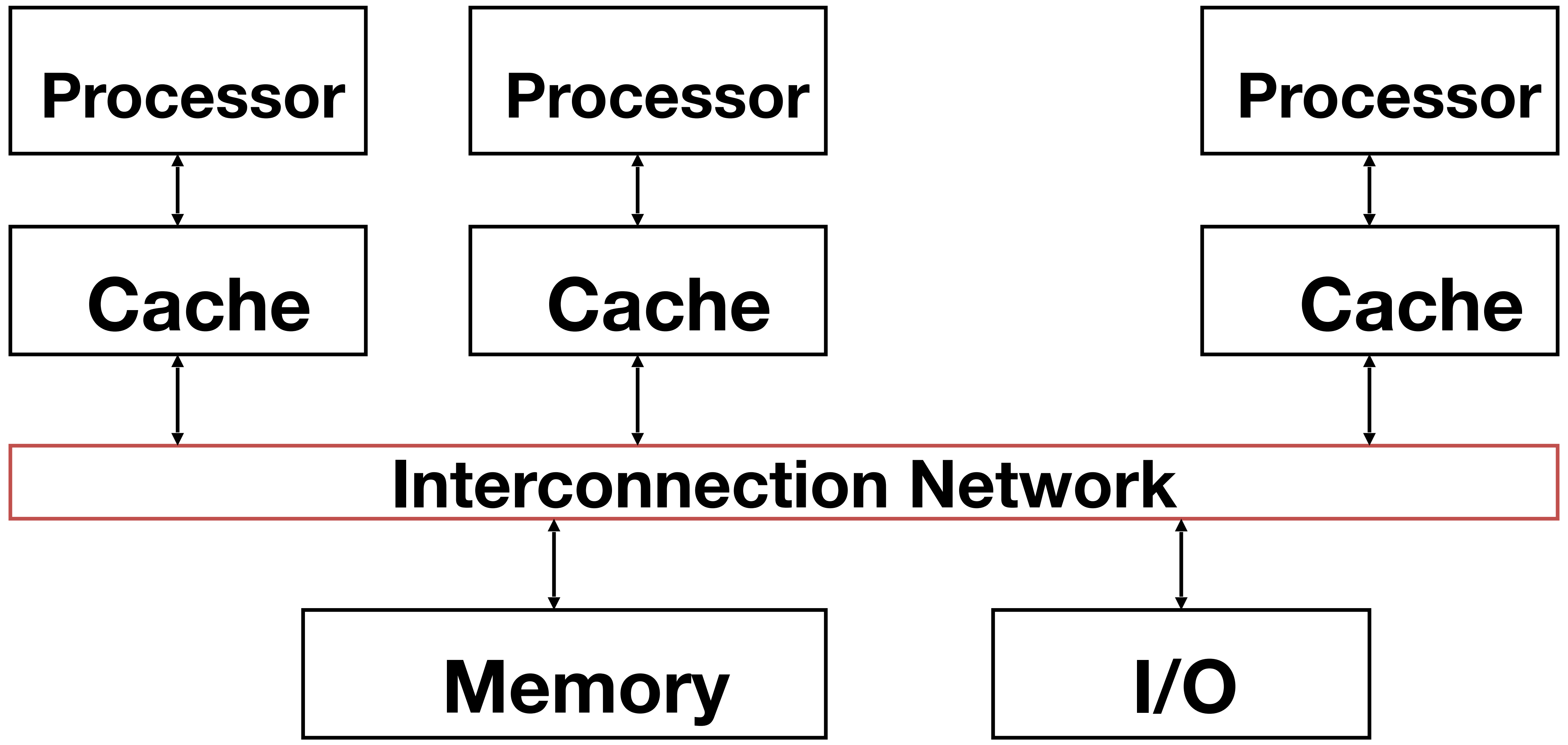
- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

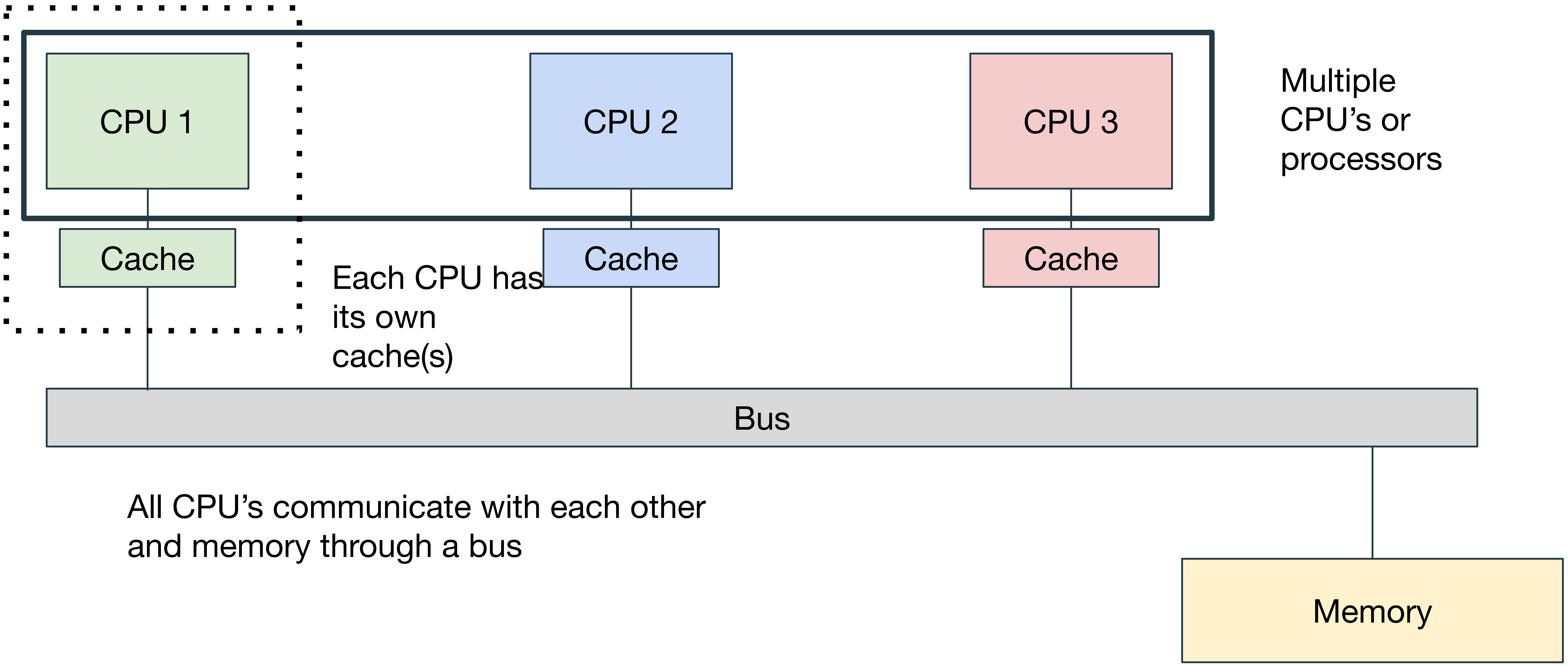
Shared Memory Multiprocessor (SMP)

- Q1 – Single physical address space shared by all processors/cores
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
  - Effectively all multicore computers today are SMP:
    Only difference is modern SMPs will use something that looks more like a network to build the shared "bus" (and shared L3$)
- Q3 - Depends on the workload!
  - Most systems go with "Best available single core within constraints, duplicate that"
  - Power-critical systems (e.g. phones) go "Some of the best available single cores, some of the most power efficient single cores"

Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory





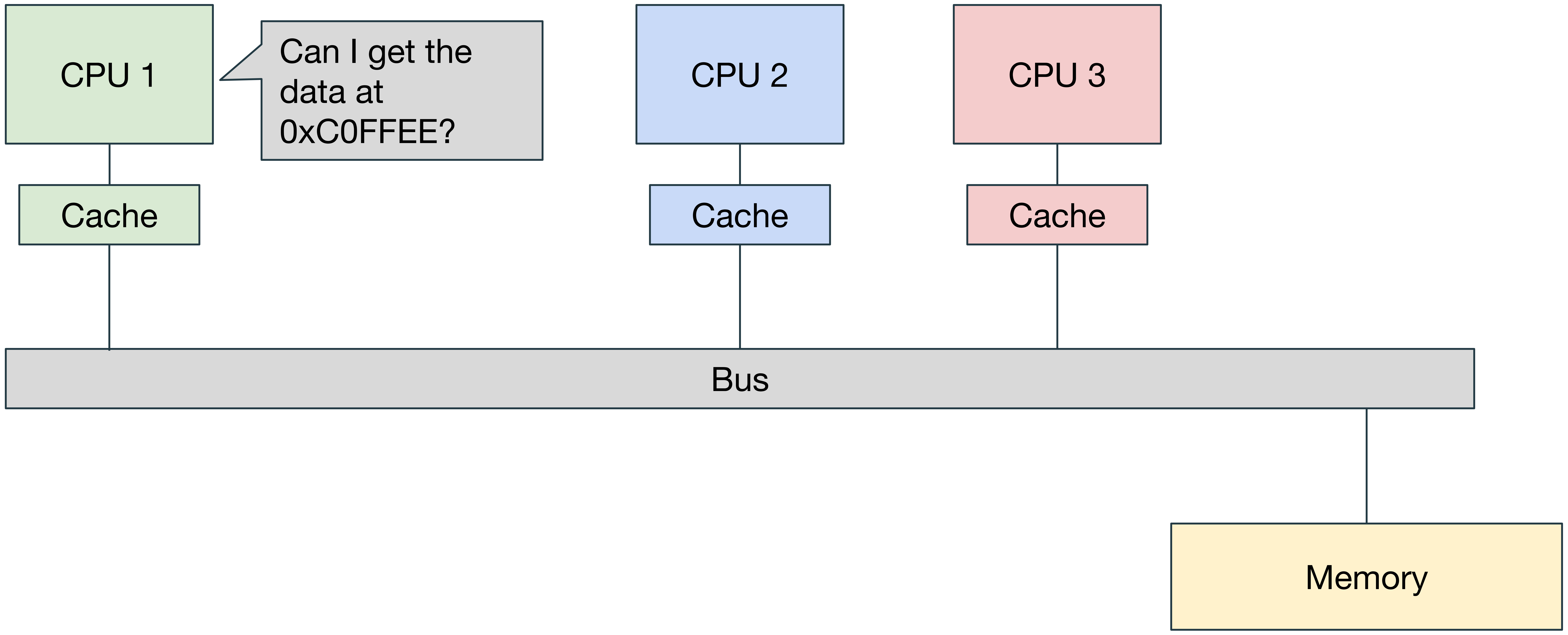
Multiple CPU's or processors

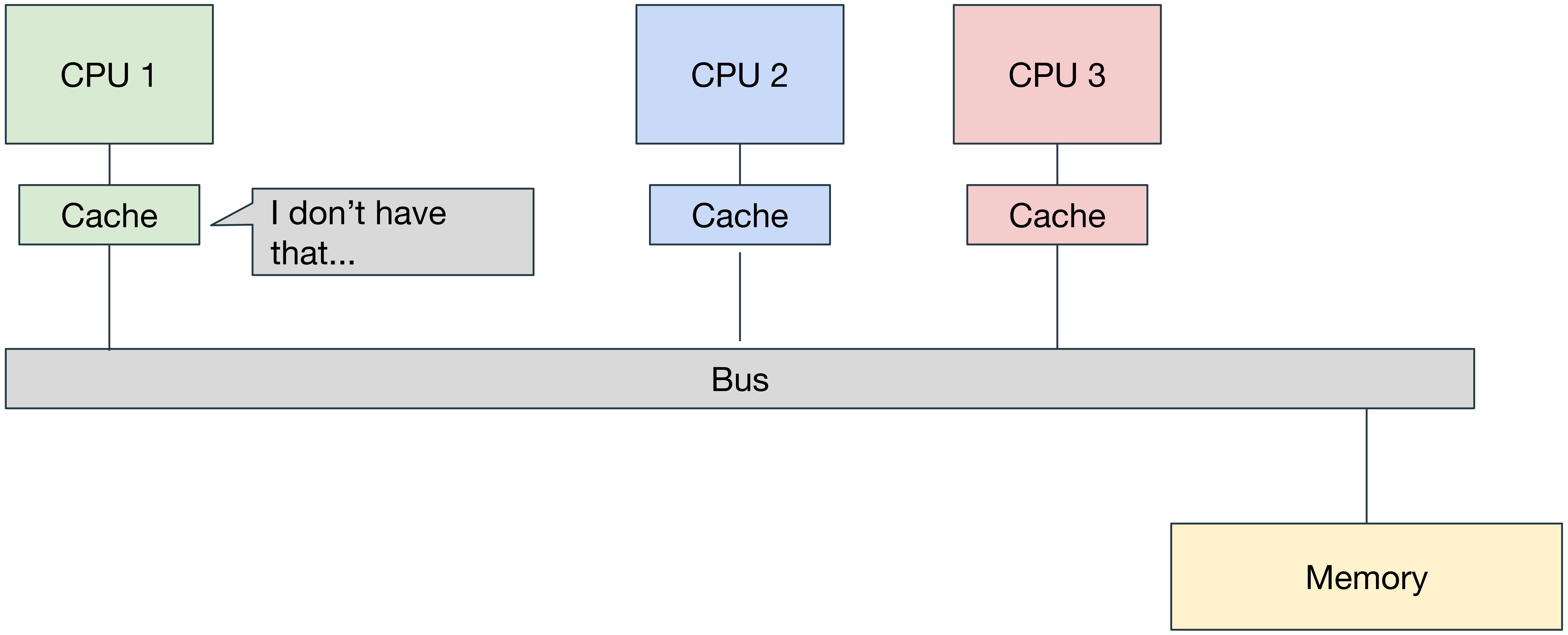
Each CPU has its own cache(s)

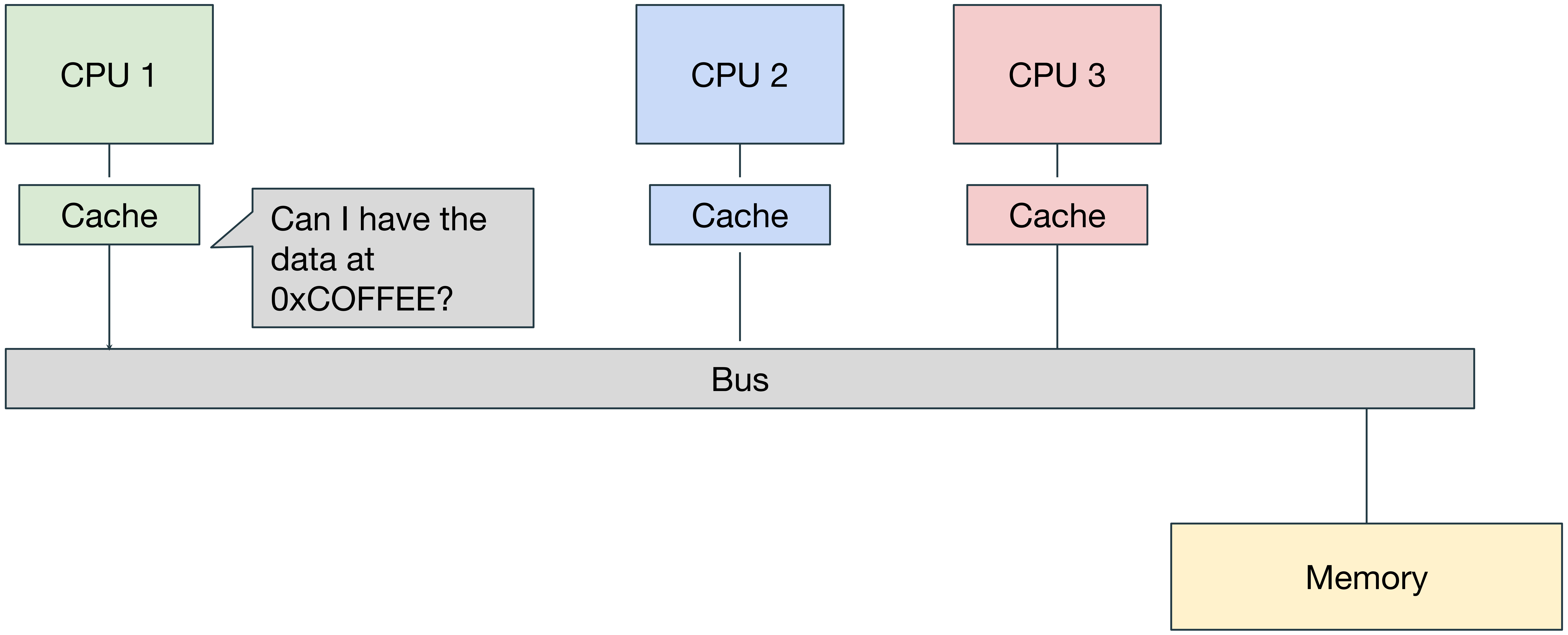
All CPU's communicate with each other and memory through a bus

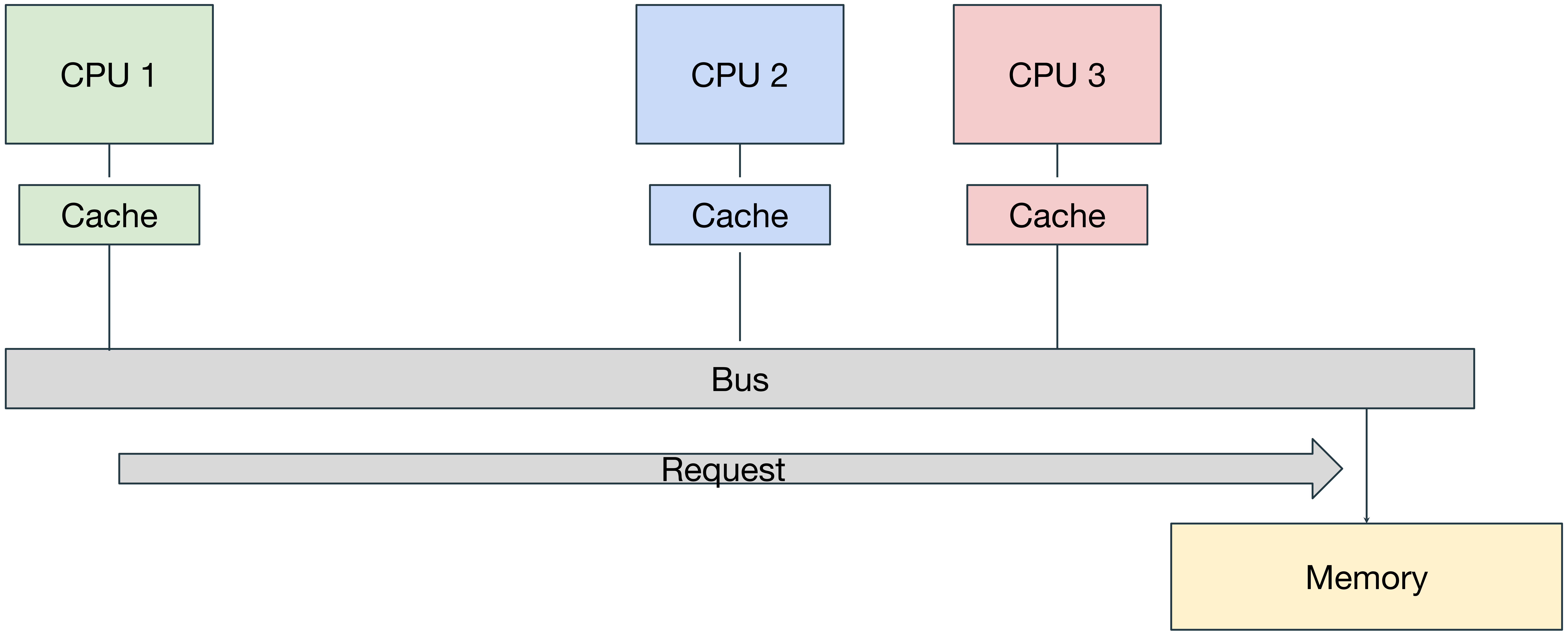
One bank of memory is shared by all CPU's

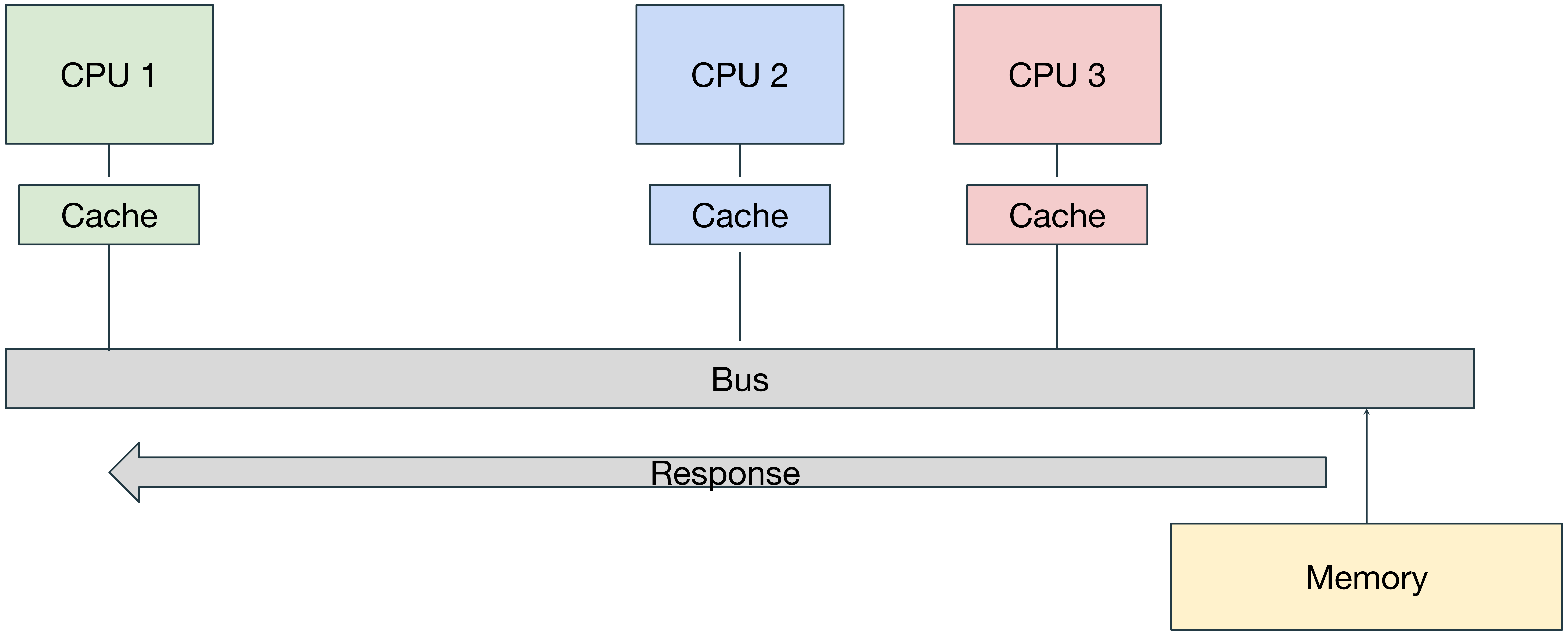
Let's go through an example to motivate why we need cache coherence!

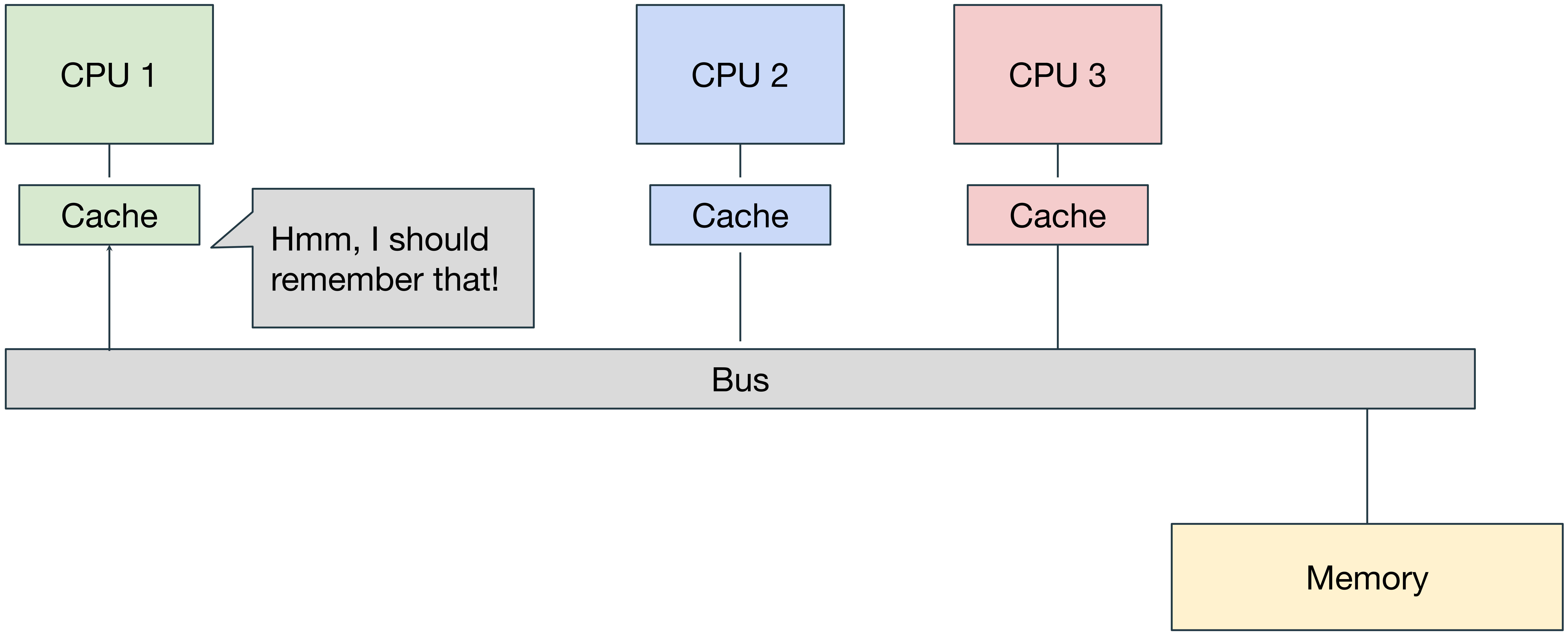

CPU 1
Can I get the data at 0xC0FFEE?
CPU 2
CPU 3
Cache
Cache
Cache
Bus
Memory


CPU 1
Cache
I don't have that...
CPU 2
Cache
CPU 3
Cache
Bus
Memory

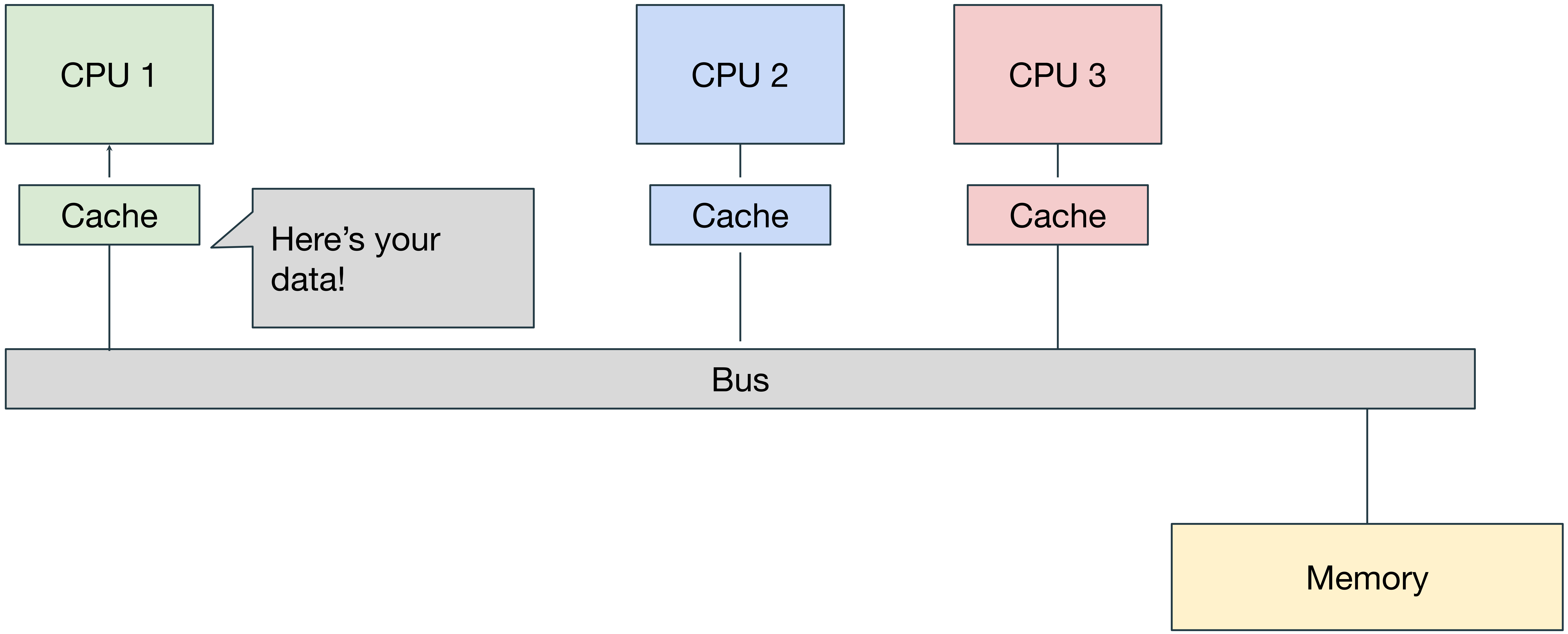

CPU 1
CPU 2
CPU 3
Cache
Cache
Cache
Can I have the data at 0xCOFFEE?
Bus
Memory


CPU 1
CPU 2
CPU 3
Cache
Cache
Cache
Bus
Request
Memory

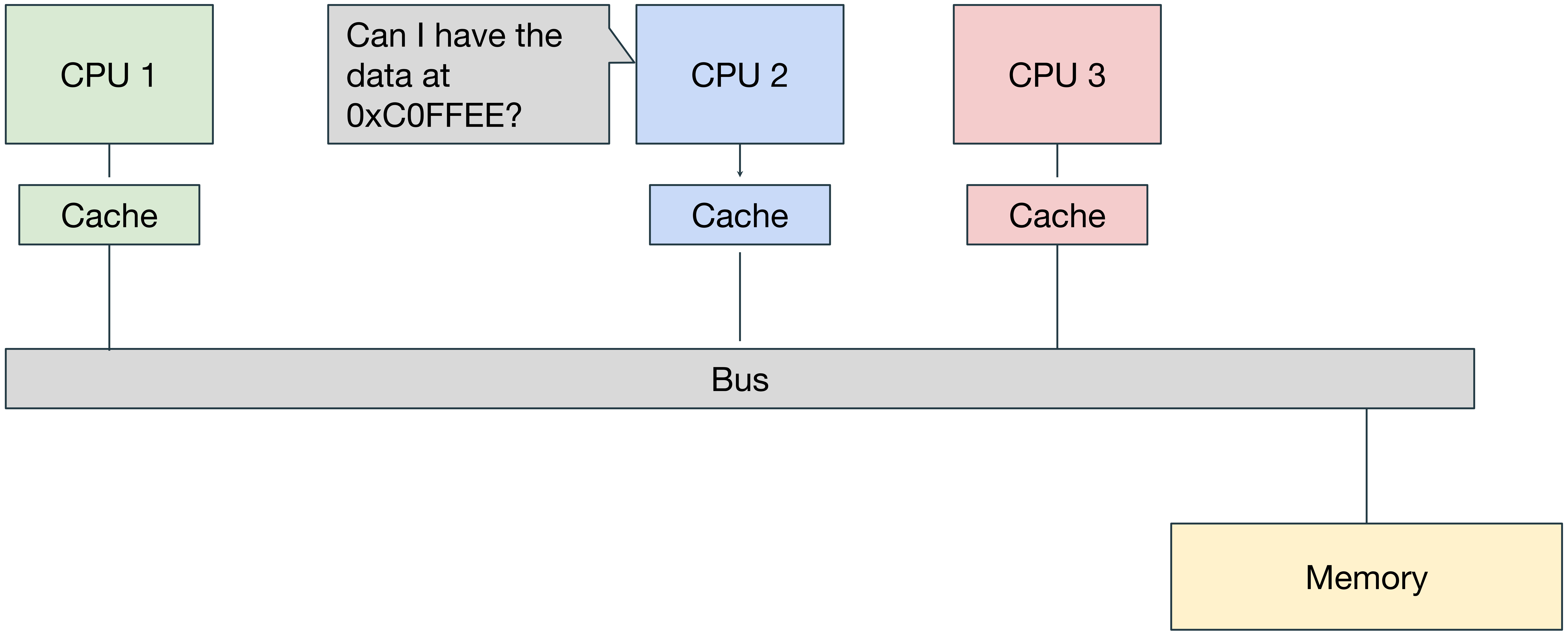

CPU 1
CPU 2
CPU 3
Cache
Cache
Cache
Bus
Response
Memory



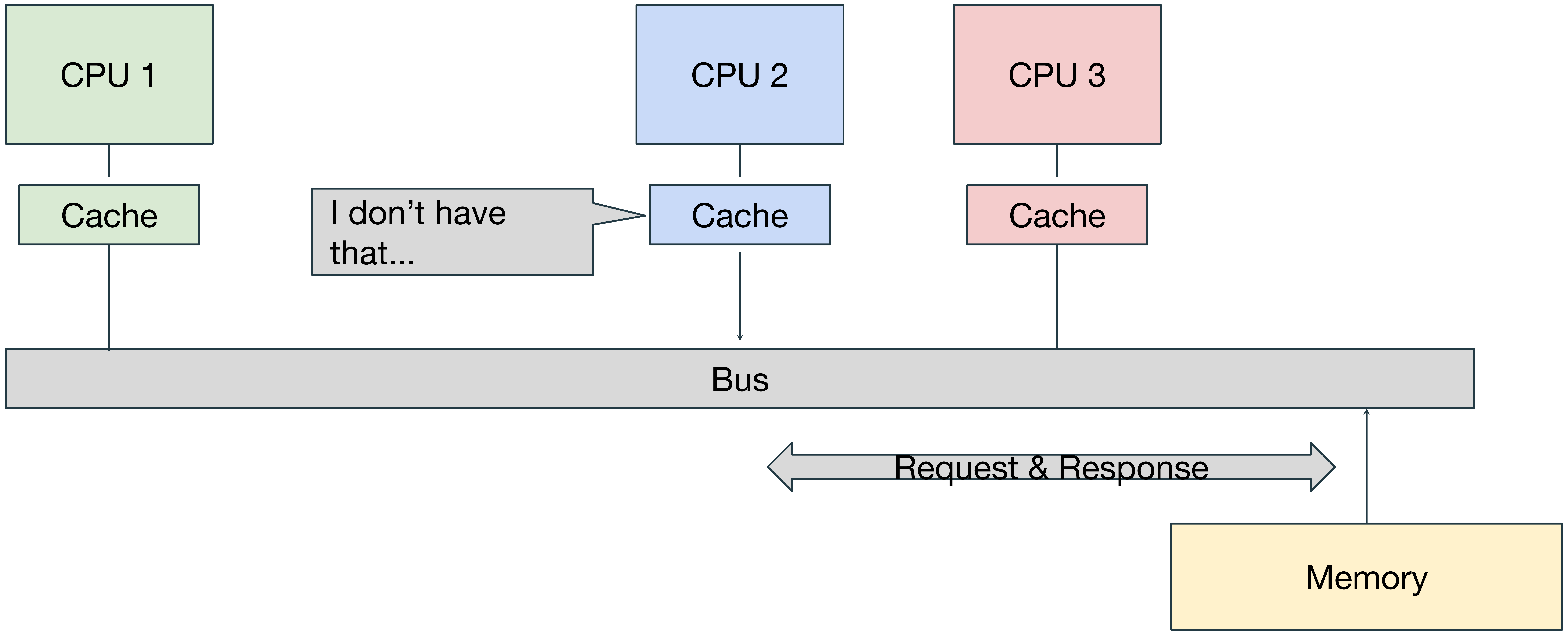
0xC0FFEE = False



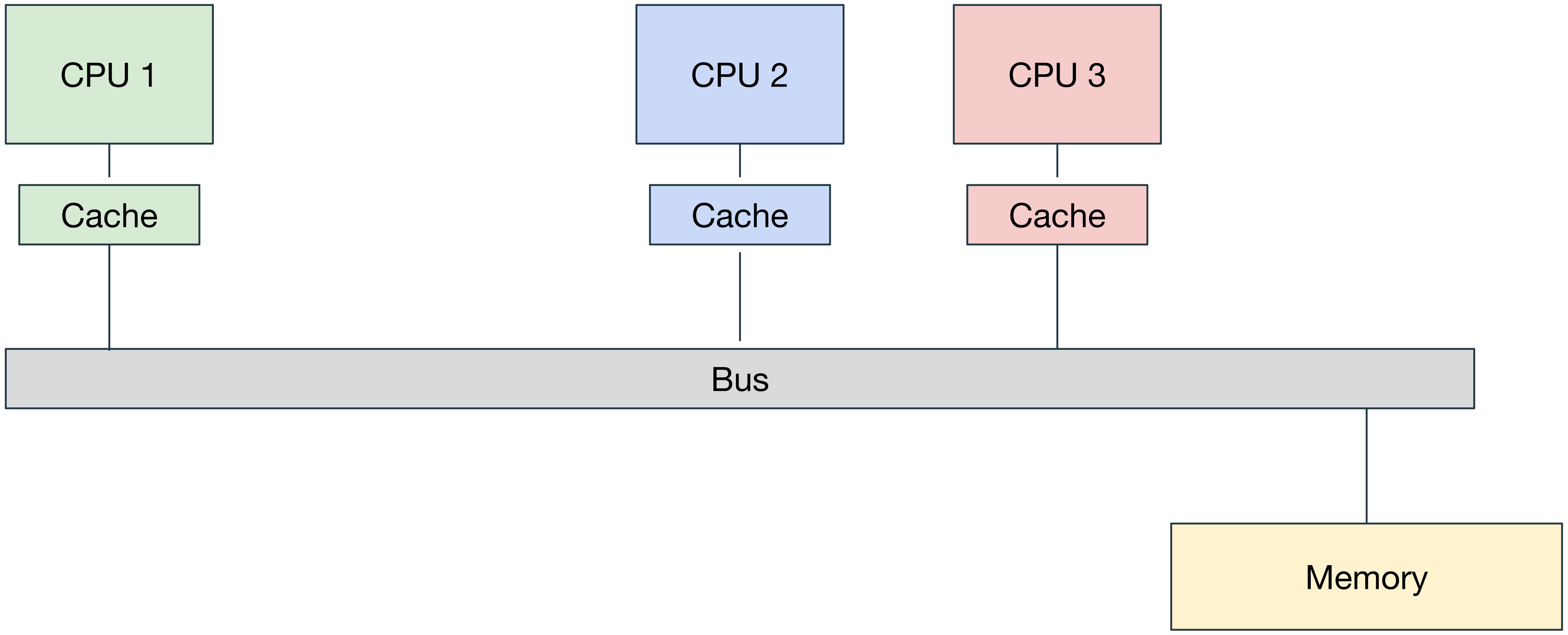
0xC0FFEE = False



0xC0FFEE = False

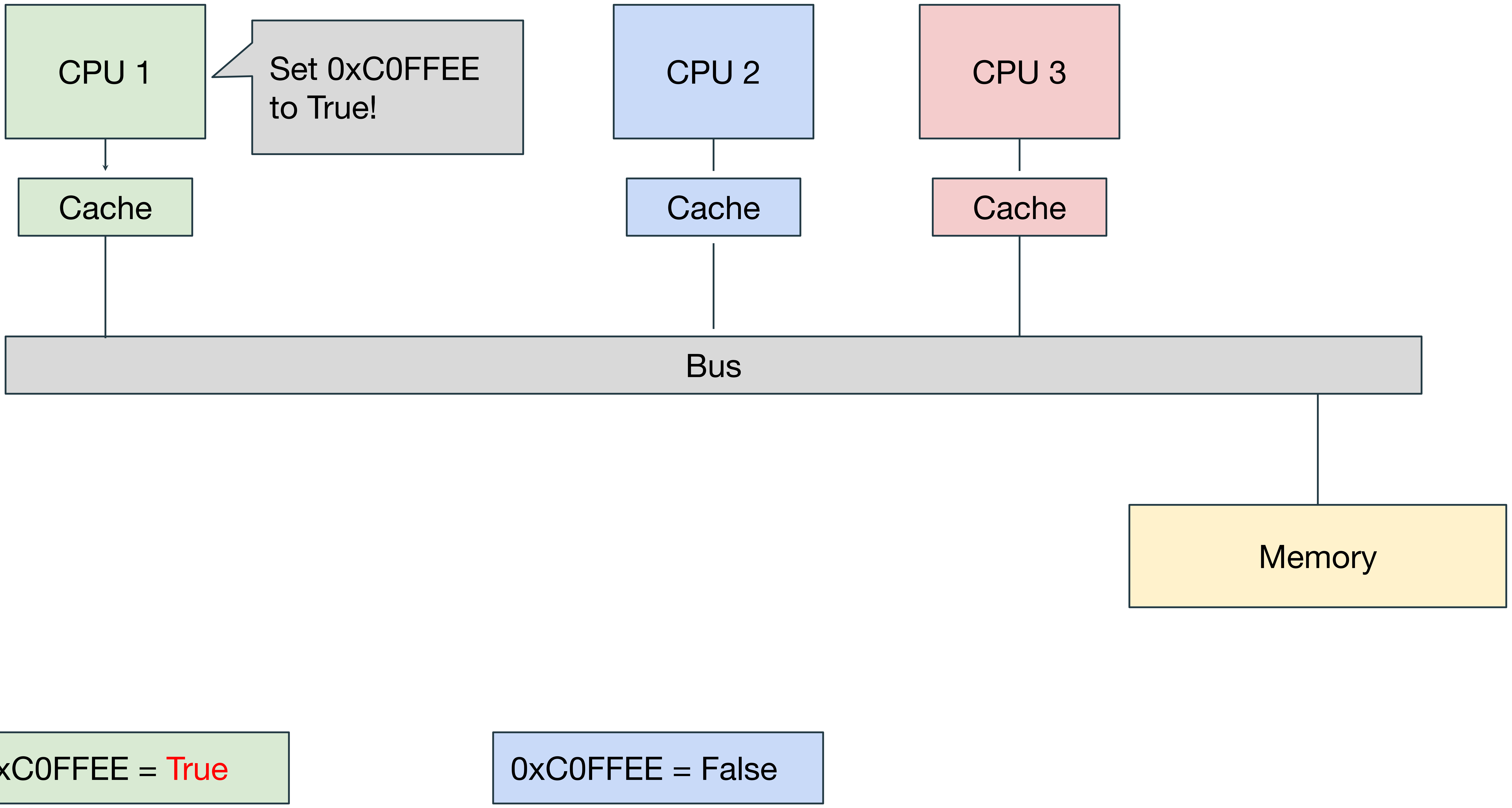


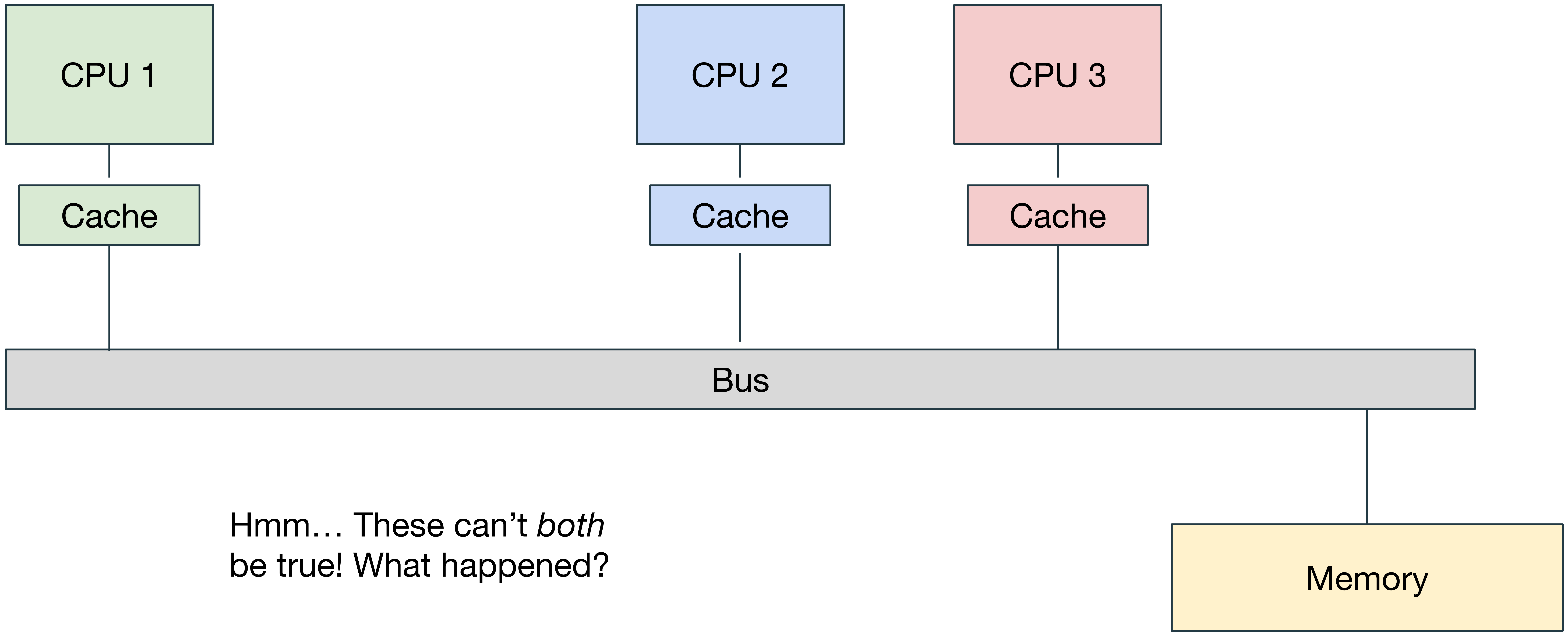
0xC0FFEE = False



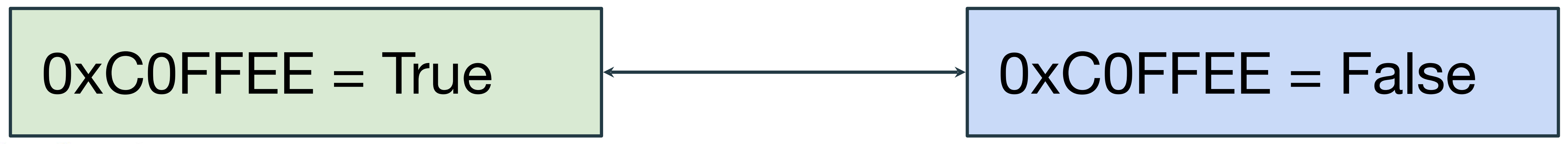
0xC0FFEE = False

0xC0FFEE = False


CPU 1
Set 0xC0FFEE to True!
CPU 2
CPU 3
Cache
Cache
Cache
Bus
Memory
0xC0FFEE = True
0xC0FFEE = False



Hmm… These can't *both* be true! What happened?



What happened?!

- Two CPU's on a bus (communication device!) both read the same data; each processor got a copy of the data and stored it in their cache.
- Then, one CPU performed a write; their cache is up to date, but the other processor has stale data (and it doesn't know its data is stale!)
- Things to think about:
  - How do we communicate when one processor changes the state of shared data?
    - Does every processor action cause data to change state?
  - Who should be responsible for providing the updated data?
  - What happens to memory while all of this is happening?

Cat Break!

Keeping Multiple Caches Coherent

- Architect's job: shared memory
  => keep cache values ***coherent***
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache

Basic Requirements: Keeping state & MSI

- Because we have multiple caches, we might have multiple copies of a piece of data floating around in our system (eg. the value of 0xC0FFEE in our previous example).
- We'll need some way to track the status of these copies; do they match what we have in memory? (dirty, or "modified"), do other processors have a copy? Is that copy up to date or stale?
- We'll keep state on a block-by-block basis; remember caches pull data in "blocksize" chunks, so all data within one block should have the same state.
  - We already have "Valid" and "Dirty", so lets add an additional bit for state: shared

Additional Enhancements:

- We want to use write-***back*** caches
  - A write-through cache design uses significantly more memory bandwidth
- We want to minimize writes overall
  - So "write-back" even in the case of shared cache blocks!
    Helps reduce the cost of coherency misses
- We can communicate by broadcasting requests
  - Shout to all other processors
- The neighboring processor's cache is faster to get to than main memory!

How Does HW Keep $ Coherent?

- We already saw how to do this with just ***valid*** and ***dirty***, but we can also think of it this way...
- Each cache tracks state of each *block* in cache:

1. *Shared*: up-to-date data, other caches may have a copy (Valid Bit set, shared bit set)
2. *Modified*: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date (Valid and Dirty bit is set)

Two Optional Performance Optimizations of Cache Coherency via New States

3. *Exclusive*: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
   - If any other cache reads this line the state becomes ***shared***
     - Can provide the line if I'm faster than main memory
   - If I write to this line, state becomes ***modified*** but I don't need to broadcast this when I do the write
   - Only valid is set
4. *Owner*: up-to-date data, other caches may have a copy (they must be in Shared state), memory is ***not up-to-date***
   - But this cache now supplies data on read instead of going to memory:
     Saves the need for a write back when somebody else reads the line
   - And now when you write you once again have to have the other caches invalidate
   - Valid, dirty, and shared is set

Name of This Common Cache Coherency Protocol: MOESI

- Memory access to cache is either

Modified (in cache)

Owned (in cache)

Exclusive (in cache)

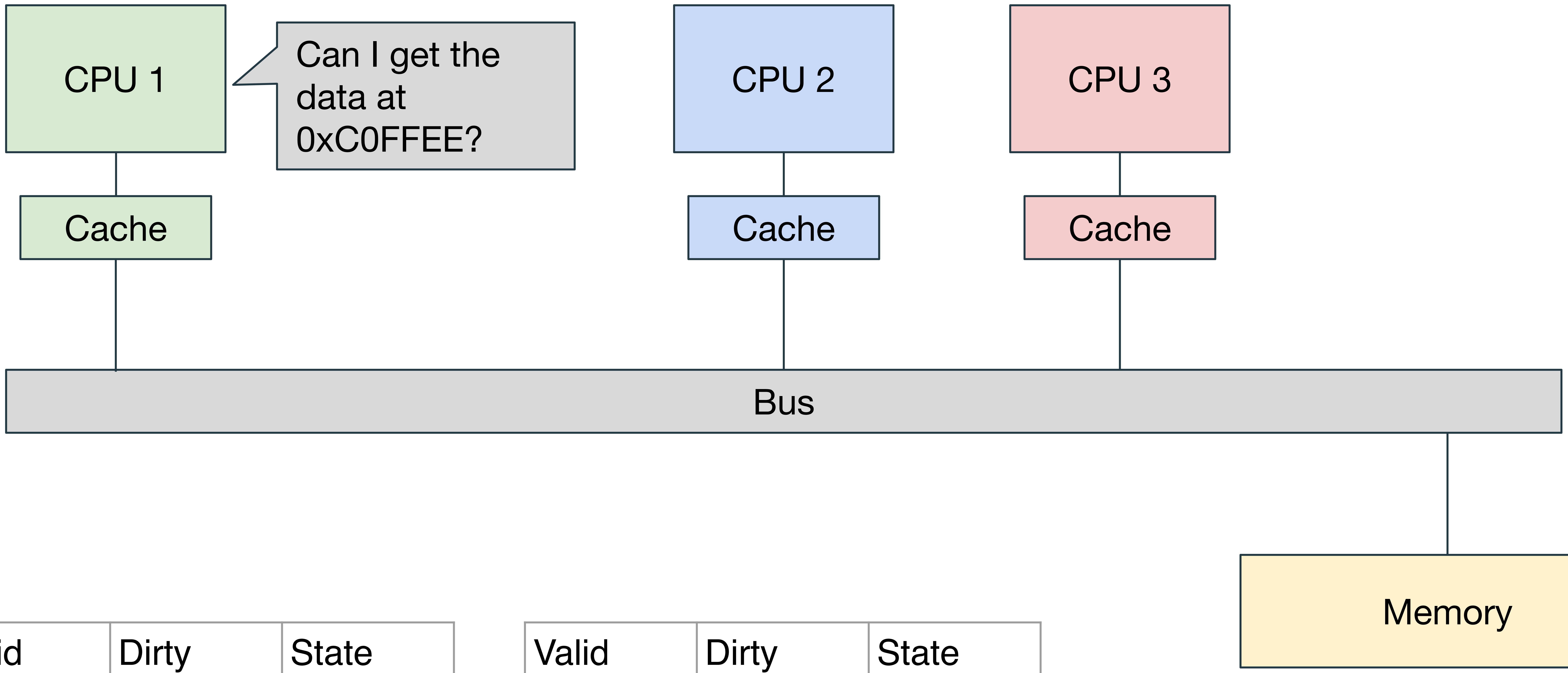
Shared (in cache)

Invalid (not in cache)

Snooping/Snoopy Protocols
e.g., the Berkeley Ownership Protocol
See http://en.wikipedia.org/wiki/Cache_coherence
Berkeley Protocol is a wikipedia stub!

Idea Behind States

- M: Modified
  - I have the only copy, and can write, and its dirty
  - If this gets evicted, I need to flush the entry
- O: Owned
  - I have the ***official*** copy, and can write, and its dirty
  - When writing, I have to tell everyone else I'm writing (and it now turns Modified)
- E: Exclusive
  - I have the ***only*** copy
- S: Shared
  - I have a copy and can read away...
- I: Invalid (duh)

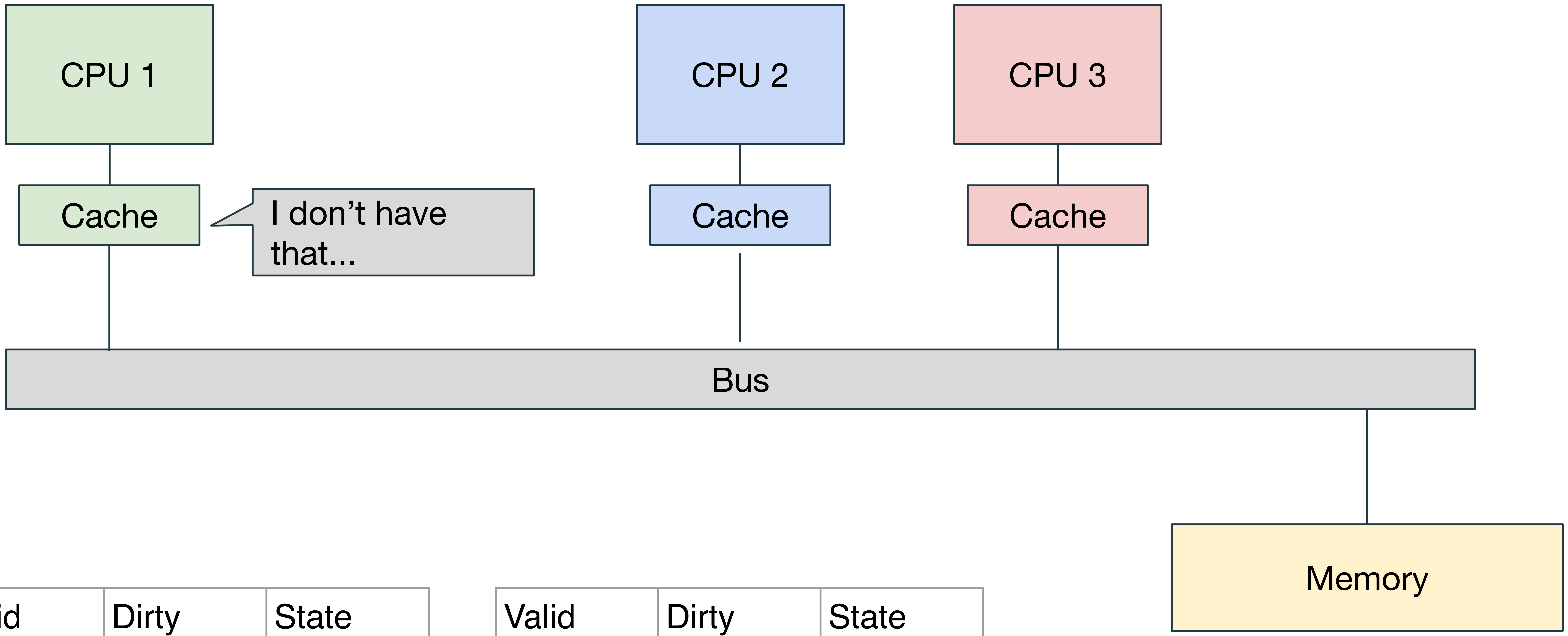


Valid	Dirty	State
0	0	I

Valid	Dirty	State
0	0	I

0xC0FFEE = ???

0xC0FFEE = ???

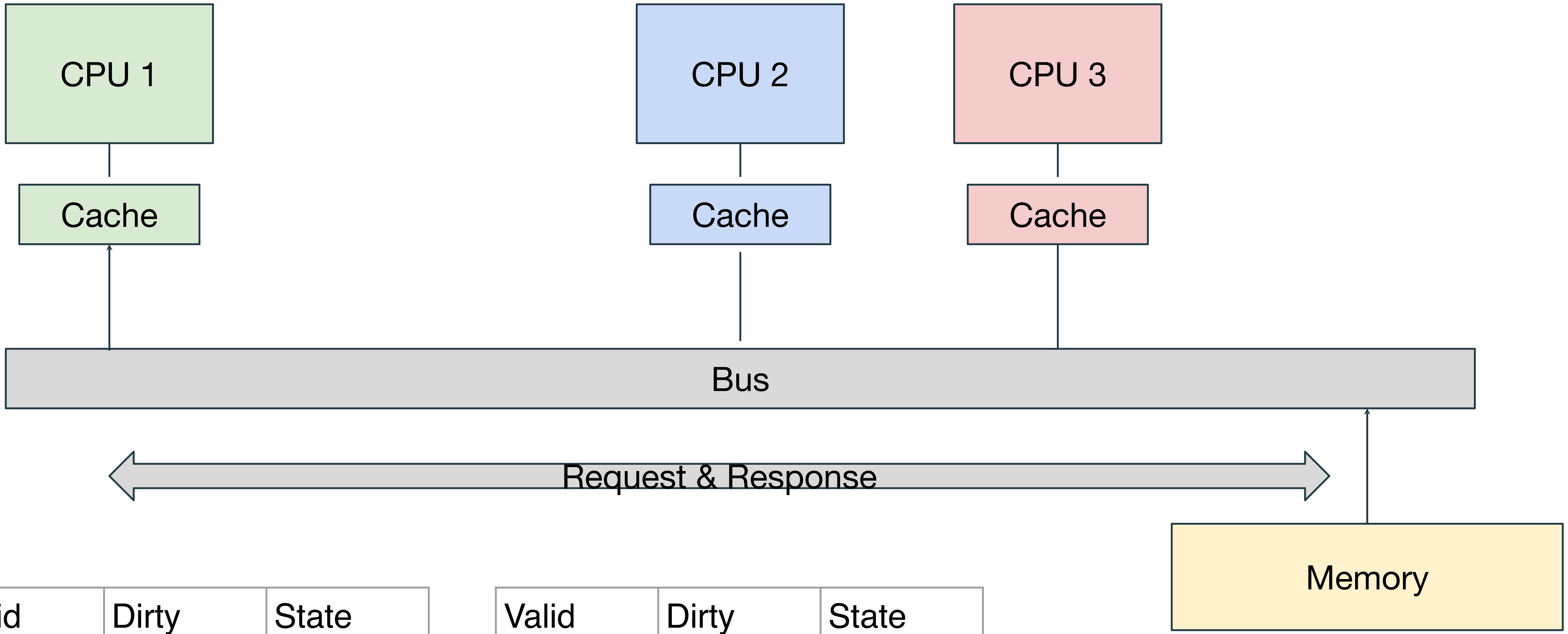


Valid	Dirty	State
0	0	I

Valid	Dirty	State
0	0	I

0xC0FFEE = ???

0xC0FFEE = ???

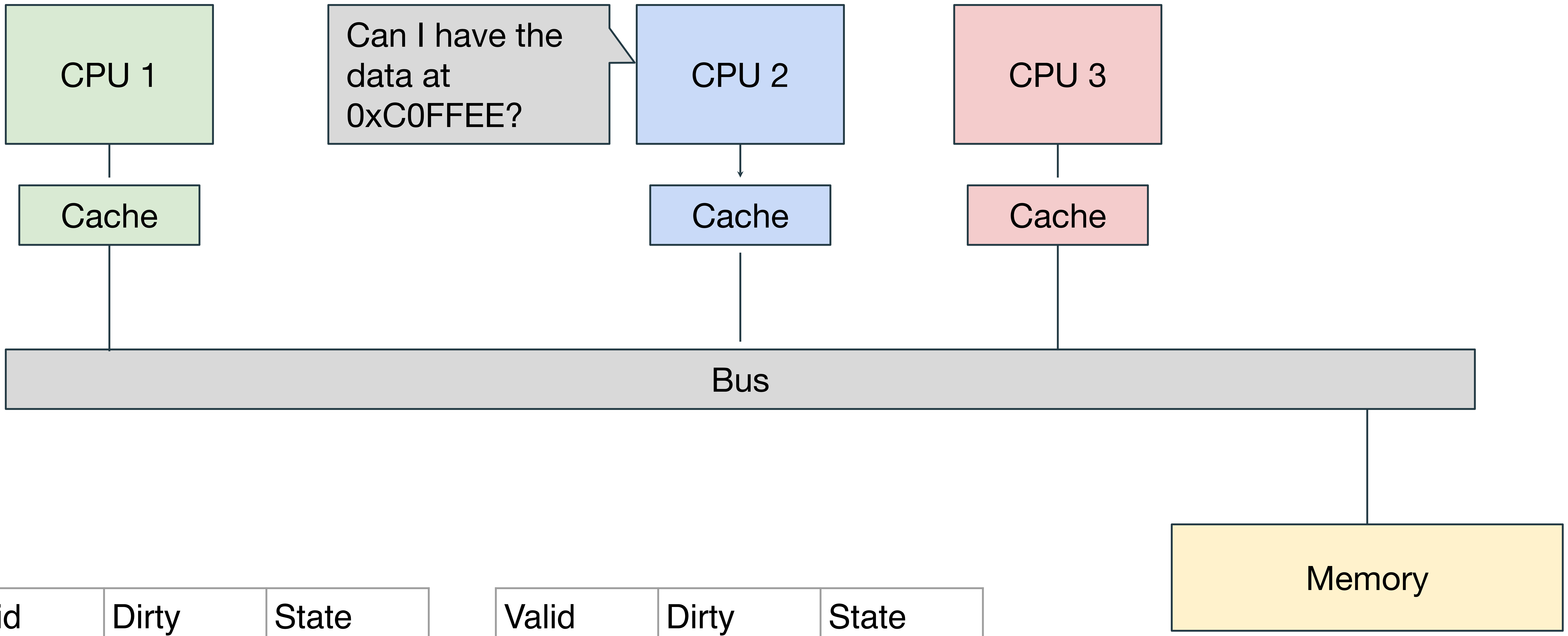


Valid	Dirty	State
1	0	E

Valid	Dirty	State
0	0	I

0xC0FFEE = False

0xC0FFEE = ???

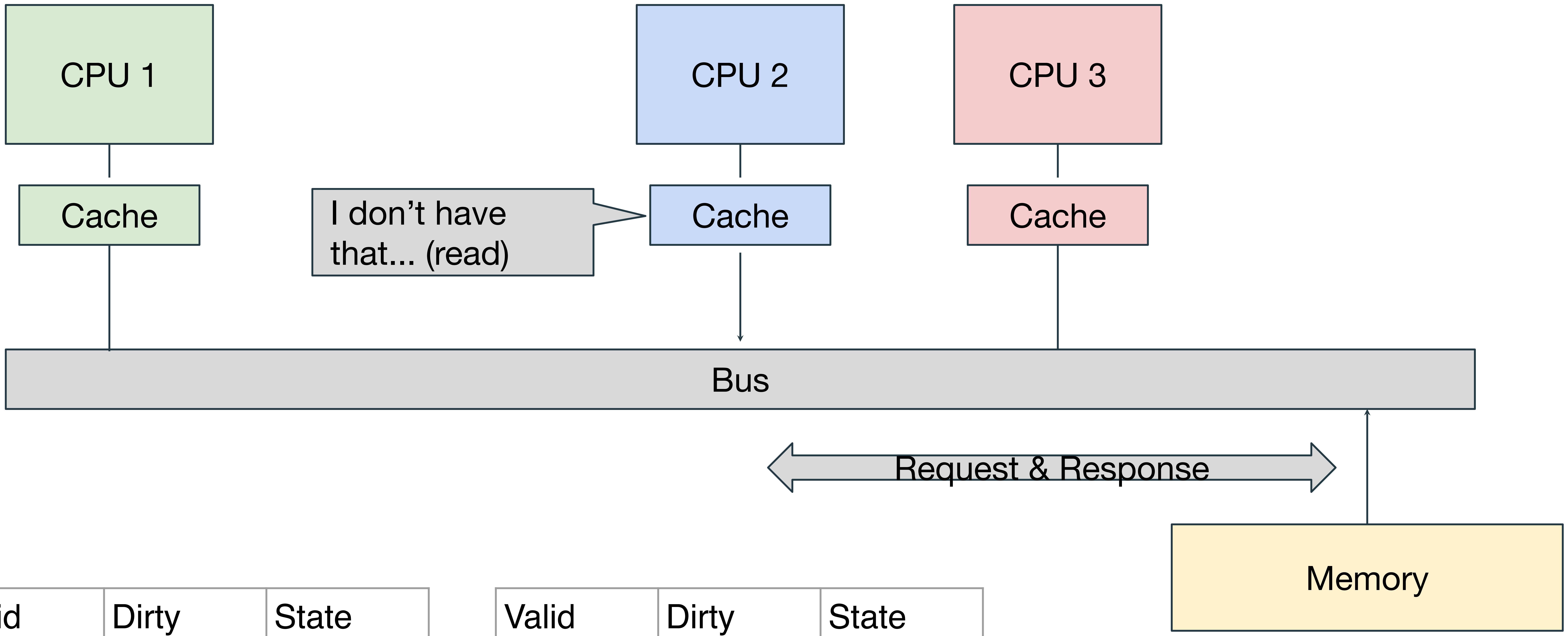


Valid	Dirty	State
1	0	E

Valid	Dirty	State
0	0	I

0xC0FFEE = False

0xC0FFEE = ???

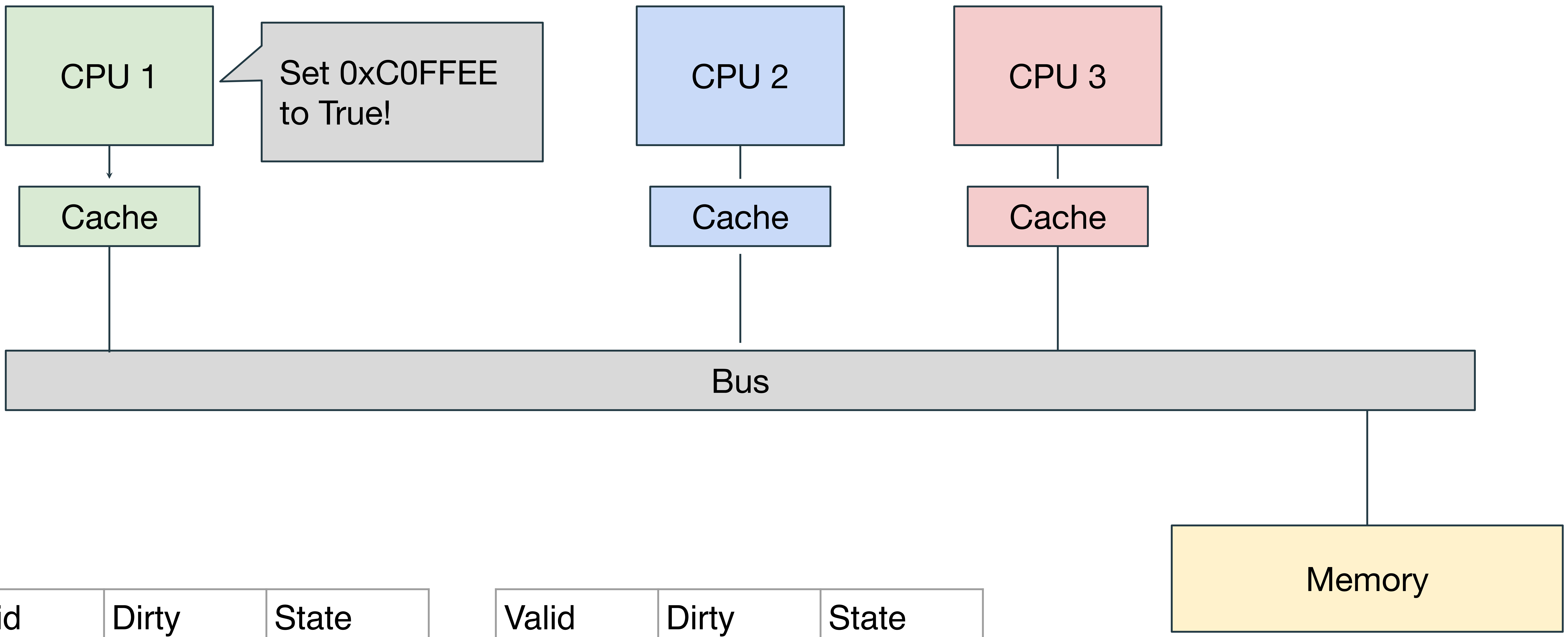


Valid	Dirty	State
1	0	S

Valid	Dirty	State
1	0	S

0xC0FFEE = False

0xC0FFEE = False

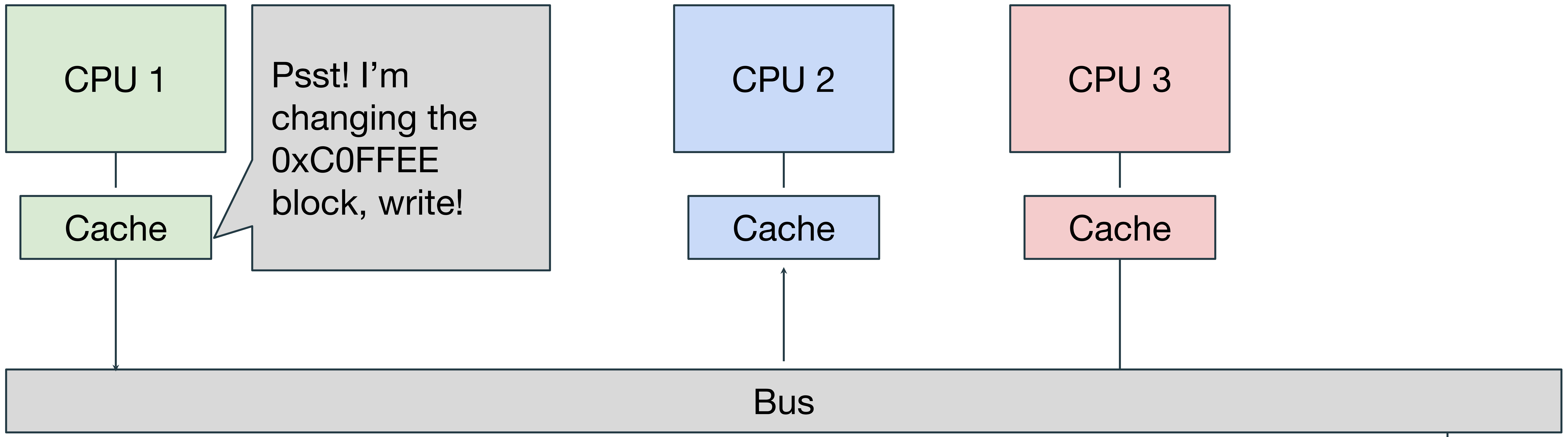


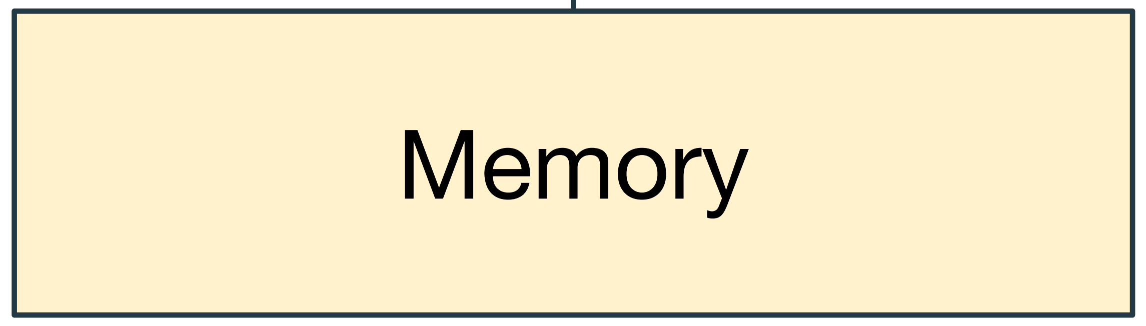
Valid	Dirty	State
1	0	S

Valid	Dirty	State
?	0	?

0xC0FFEE = True

0xC0FFEE = False?



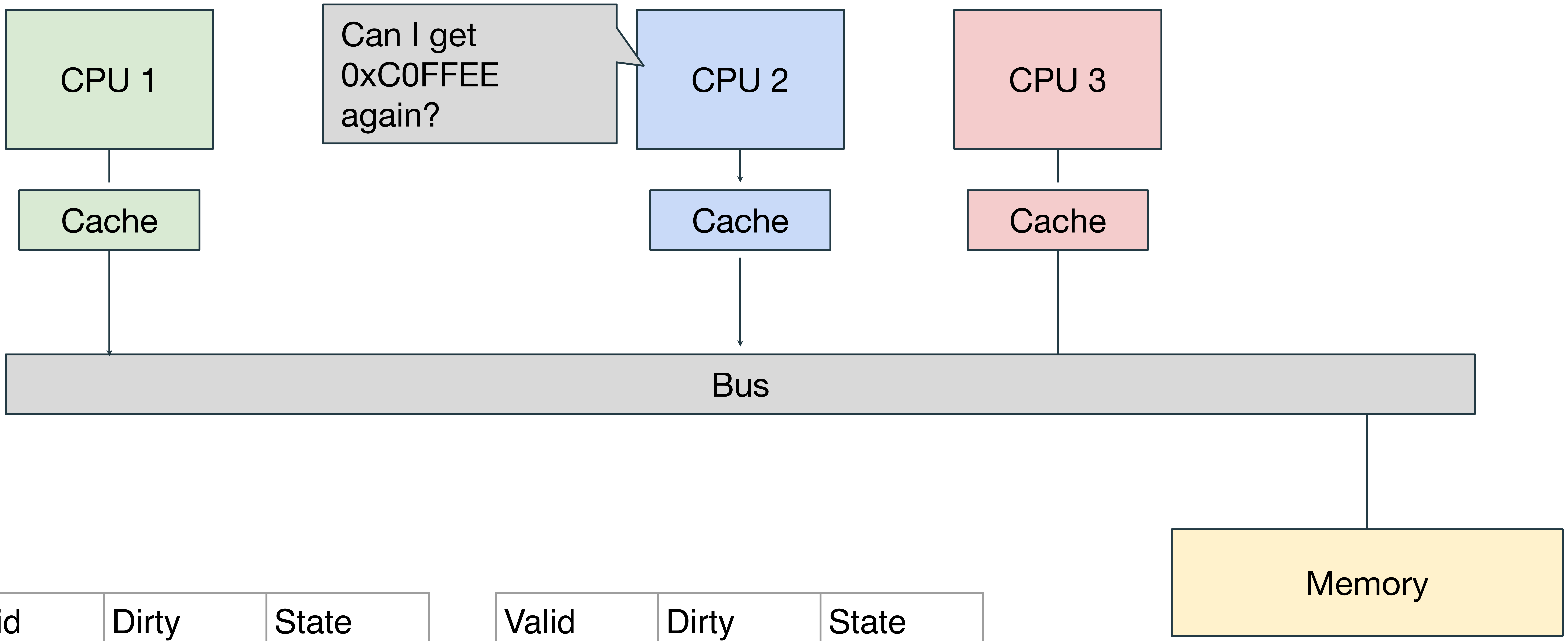


Valid	Dirty	State
1	1	M

Valid	Dirty	State
0	0	I

0xC0FFEE = True

0xC0FFEE = ???

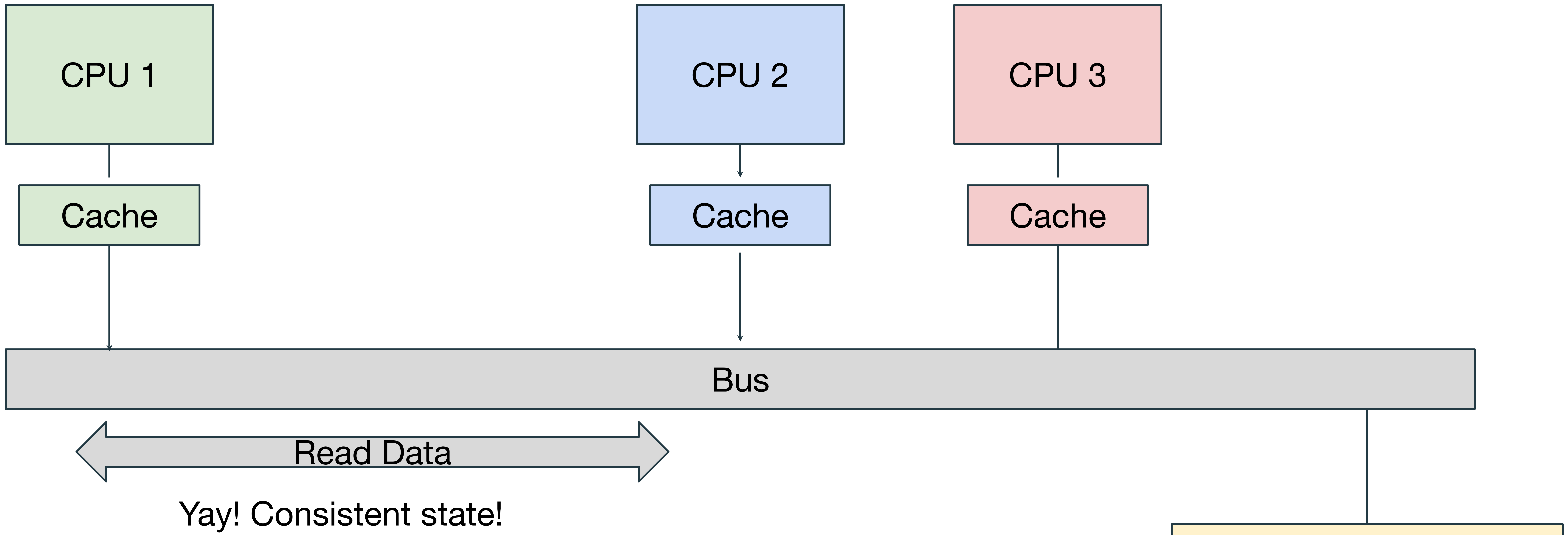


Valid	Dirty	State
1	1	M

Valid	Dirty	State
0	0	I

0xC0FFEE = True

0xC0FFEE = ???

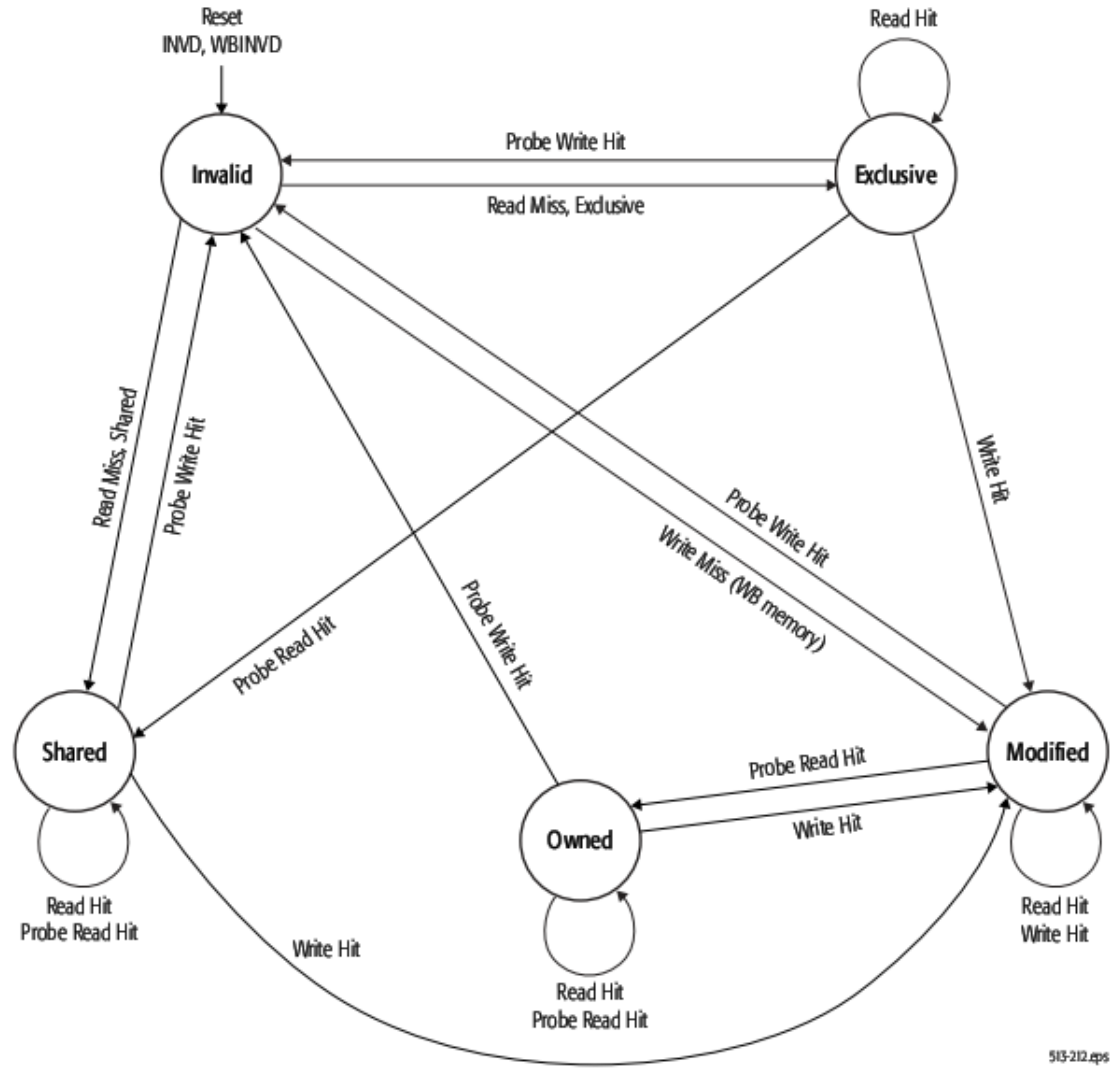


Valid	Dirty	State
1	1	O

Valid	Dirty	State
1	0	S

0xC0FFEE = True

0xC0FFEE = True



Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

The Bits Needed

Valid	Dirty	Shared	State
0	0	0	Invalid
0	0	1	Invalid
0	1	0	Invalid
0	1	1	Invalid
1	0	0	Exclusive
1	0	1	Shared
1	1	0	Modified
1	1	1	Owned

And Implementing `lr/sc`

- `lr`: Just do a load like normal...
  - Now it will be in the cache in E/S/M/O state...
- `sc`: Before storing, make sure it is still in the cache...
  - If E/M: Just write away...
  - If S/O: Broadcast a request to do the write
    - Need to handle the case if two caches do a write in the exact same cycle...
    - If successful, return success
      - If fail, return failure..
    - Which is why caches for multiprocessors are a huge beast to get right in practice
  - If I: Return failure...

Well, that's the simple way...

- RISC-V offers some guarantees on **`lr`/`sc`**
  - Result is actually implementing this is even harder!
- If you do a **`lr`/`sc`** pair with the following properties
  - The **`lr`** is the last **`lr`** issued
  - 16 instructions total, including recovery code
  - No branches except a back branch on failure of the **`sc`**
  - Only arithmetic/logical instructions
- You will make forward progress if multiple cores are doing this...
  - Allows e.g. more complex reduction operations without locks.

The Problem: Sharing and False Sharing...

- If you share data between two processor cores:
  - Every time one does a write the other will take a cache miss on the next read or write...
  - New class of cache miss type: ***coherency***
    - Goes along next to ***compulsory***, ***capacity***, and ***conflict***
- If you share a cache ***line*** between two processor cores:
  - Every time one does a write the other will take a cache miss...
  - Even if you are writing to different ***parts*** of the cache line
- But as long as everyone is reading its all cool
  - And just make sure your writes are to different cache ***lines*** to minimize the cost of writes

The Opposite Happens with Multithreaded cores

- If two hardware threads share the same physical core...
- Both writing to and reading the same data?
  - No problem, doesn't cause Coherence misses
- Both hardware threads using ***different data?***
  - Now we get a bunch of cache misses (?incoherence?)
- Why initial hardware multithreading was often a performance loser
  - OS scheduler needs to be aware

But An Alternate // Programming Paradigm... Communicating Sequential Processes

- OpenMP has ***very restrictive*** parallelism
  - Really only good for parallelizing loops with an optional reduction step
    - And Amdahl's law therefore quickly rears its ugly head
- Raw threads is ***very easy*** to get wrong
  - Deadlock situations
- Enter CSP:
  - A way for different threads to efficiently communicate
- A good CSP language: Go (golang)
  - Really good for shared-memory multiprocessors

What is Go

- Language created at Google starting in 2007
  - Primarily by a bunch of old Unix hands: Robert Griesemer, Rob Pike, and Ken Thompson
  - 1.0 released in March 2012
- Language continues to evolve, but a commitment to backwards compatibility (so far)
  - A correct program written today will still work tomorrow
    - I'm looking at you, ***python 3....***
- Mostly C-ish ***looking*** but...
  - Strong typing, no pointer arithmetic, lambdas, interfaces, garbage collection and...
  - Strong emphasis on concurrent computation

Good Go Resources

- The Go website:
  - https://golang.org/
- Especially useful: Effective Go:
  - A cheatsheet of programming idioms. Several example from this lecture stolen from there
  - https://golang.org/doc/effective_go.html
- When searching Google, ask for ***golang***, not go
  - The language may be Go, but golang refers to the language too
- If I'm starting new code and I need to care about performance, scalability, or maintainability, I use Go
  - Only exception is if the problem can't stand a garbage-collector pause (such as an OS, low level drone flight control, etc...), in that case I'll have to learn Rust

So Think of Go as:

- C's general structure & concepts
  - But with implied ;s and a garbage collector
- A better typing system with interfaces, slices, and maps
  - No class inheritance, however
- Much more symmetric functions
  - Can return multiple values
- Scheme-like lexical scope
  - Lambdas and interior function declarations
- Communicating Synchronous Processes (CSP) concurrency
  - Multiple things at once in the same shared memory space:
    quite suitable for MIMD

Coroutines, err, goroutine

- Conceptually, a goroutine is just a thread...
  - `go fubar() // Executes fubar as a new coroutine`
- But in practice it is designed to be ***much*** lighter weight
  - Threads (e.g. in OpenMP) relatively expensive to create:
    Operating System involvement is never cheap as ***any*** call to the OS is effectively an interrupt
- Go's runtime instead pre-creates a series of threads
  - And then schedules the active coroutines itself to the available threads
- Result is goroutines are ***cheap***
  - It is only slightly more expensive than a plain function call:
    new goroutine just has a small independent stack
    context switching between goroutines is very cheap

Channels

- Channels are the primary synchronization mechanism
  - You have locks but why bother?
  - A typed and (optionally) buffered communication channel

  -
    ```
    c := make(chan int)
    e := make(chan *fubar, 100)
    ```
- Writing data to a channel:
  - `c <- 32 // Writing blocks if unbuffered or full`
- Reading from a channel:
  - `var f = <- e // Reading blocks if no data`

Channels as Synchronization Barriers

- Any writes in the code before writing to the channel complete first

  -
    ```
    globalA = ...
    d <- 1    // The write to globalA must complete just before this
    ```
- Any reads in the code after reading from the channel do not start until channel-read takes place

  -
    ```
    <- d
    fubar(globalA) // Won't read globalA before channel read
    ```
- Otherwise, compiler can reorder ***however it wants*** as long as sequential semantics are preserved for the sequential function
  - Go may have removed a lot of ways to shoot yourself in the foot...
    but unless you use channels you will easily blow it off with race conditions

Without Synchronization Barriers, The Compiler Can Go To Town

- This doesn't work!
  - Compiler can reorder the writes between x and done safely
    - Similar variants also possible
    - Oh, and some processors (notably RISC-V) also allow the processor to recorder the writes to memory too!
- This is one of the two biggest pitfalls of Go:
  - Unless you explicitly synchronize, multiple processes can write in "weird" ways
  - The other is the abysmal error/exception handling mechanism

```
var x : string
var done : bool
func foo(){
   ...
   x = "something"
   done = true
}

func bar(){
   go foo()
   for !done {...}
   y := x
}
```

Using goroutines

- For things which may block or wait
  - EG, on input/output, waiting for stuff to happen, etc
  - Just create as many as you want!
    - Its cheap so why not: let the scheduler do useful work when another one is waiting
    - EG, if building a webserver, you launch a goroutine to handle each communication stream
    - This is what python threads are actually good for:
      And then use python Queues like go's channels (without the cool of select...)
- For performance tasks
  - Theoretically create as many as there are CPU cores
    - Otherwise you are wasting resources: but the wastage is low, so you don't necessarily need to
      Remember, Amdahl's law
  - But it should be much more efficient than OpenMP:
    - Thread creation is vastly more overhead in C/C++ than go

EG a parallel loop in Go

-
  ```
  type Element struct {
    ...
    done chan(bool);
  }

  func bar(data element, index int){
    ...
    // Only if doing fork/join model
    data.done <- true
  }

  func foo(data []element){
     for x, i := range(foo){
        go bar(x,i)
     }
     // Needed only if I want to do fork/join wait
     // until everything is done...
     for x, _ := range(foo){
         <- x.done
     }
  }
  ```

Select

- Select allows you to wait on multiple channels

-
  ```
  select {
     case c <- x:
           x, y = y, x+y
     case d := <- e:
         fmt.Println("Got %v", d)
     default:
         time.Sleep(50 * time.Millisecond)
  }
  ```
- If can write or read to a channel, do so and ***then*** execute the associated case
  - If multiple cases are valid, chose one ***at random***
- If nothing is available, execute default (if any)
  - If no default, just block until you can write or read
  - Default enables non-blocking read & write

Lots of other features for code correctness

- Compiler is, umm, persnickety
  - It is an error to declare but not use a variable or include but not use a package
- Designed to turn comments into documents
  - Including examples
- Libraries for building example and test routines
- Built in package management integrated into git
- "Single workspace" notion
  - Use common modules to prevent code drift

So You are Starting A NEW Project?

- There are two modern languages you should first consider: Go and Rust
  - Both have good static typing, high performance compliers, wide support
- Rust
  - ***NO GARBAGE COLLECTOR***!
    Instead a complex memory model that gives malloc()/free() type performance with safety
    - Makes things better on tight embedded systems or problems where you need deterministic memory performance
  - Better exception handling syntactic sugar
- Go
  - More graceful learning curve
  - Better parallel programming model with select

And Finally... Nick Applies 61C to Build His New Desktop at the start of COVID

- In Spring 2020: being in the COVID-bunker, Nick decided he needed a new desktop system
  - Since he's always in a fixed place, the laptop's portability matters a lot less...
- "Not My Money But Is My Money"
  - So sorta care about the budget, but not as much...
- 61C designed performance improvements
  - >2x improvement in sequential performance
    - That I can get this is ***yuge***, and partially the reason for the expense
  - ~10x improvement in MIMD performance
    - Lots of cores
  - 10x improvement or more in SIMD performance

Previous Desktop: Intel Skull Canyon NUC

- Core i7-6770HQ (6th generation)
  - 4 cores/8 threads
    - 2.6 GHz processor core, 6 uOP/cycle issue rate
    - L1I$ = 32 KiB, 8-way set associative, 64B line size
    - L1D$ = 32 KiB, 8-way set associative, 64B line size
    - L2$ = 256 KiB, 4-way set associative, 64B line size
    - L3$ = 6 MiB, 12-way set associative?, 64B line size
    - ***L4$ = 128 MiB*** (DRAM in package/shared with graphics subsystem)
  - Iris Pro Graphics 580 GPU
    - 72 execution units, 1 Teraflop maximum single-precision theoretical output

Step 1:
Number Of Cores?

- Want >8 cores to get the MIMD improvement
  - But don't want to sacrifice sequential performance!
- Decided on AMD over Intel
  - Brand A v Brand X a long subject of debate...
  - At the time, I liked the Zen-2 AMD cores slightly better
    - Today I'd get an Intel "Alder Lake" if building an x86 unless I needed >8 performance cores...
- Zen 2 core
  - 32 KB, 8-way set associative L1 I and D caches
  - 512 KB, ***8-way*** set associative L2 caches
    - Double associativity is about as good as doubling the cache size!
  - Lots of other microarchitecture changes

Two choices...

- Ryzen 9 3900x
  - 12 cores, 24 threads, 3.8 GHz
  - 64 MB L3 cache!
- Ryzen 9 3950x
  - 16 cores, 32 threads, 3.5 GHz
  - Still a 64 MB L3 cache
  - And more money!
- So go with the faster/fewer thread version
  - Amdahl's law is a @#)(@#()

So Far...

- Somewhere between 2x and 3x sequential performance
  - AMD claims a ~17%+ gain in CPI over the Zen 1 core...
    Plus bigger caches...
    Plus much higher clock rate
    - Higher clock rate largely due to the ability to cool much more aggressively: the Skull Canyon was a small form factor so small fan vs huge-glowing-heat-sink
- Close to 10x on MIMD workloads
  - 3x the cores, with the cores being a lot better
- Now to select the SIMD solution...
  - Options are AMD vs Nvidia...
    Another Brand A/Brand N type situation
  - Decided on Nvidia: Both are pretty much the same for gaming, but Nvidia has an edge in use for compute...
  - Got one using the latest architecture:
    Worse on cost/performance but hey...

Selecting High End Graphics/Compute...

- Nvidia GeForce RTX Super line
  - 2060 Super: 2176 CUDA cores, 6 TFLOP ~$400
  - 2070 Super: 2560 CUDA cores, 8 TFLOP ~$500
  - 2080 Super: 3072 CUDA cores, 10 TFLOP ~$700
- Decided on the 2080...
  - Note: bought before the cryptocurrency space went insane (again), and not upgrading to a 3080...
- Result:
  - 2x-3x sequential
  - close to 10x MIMD
  - easily >10x SIMD
- For each of these:
  - 1/2 the performance from just going to newer technology
  - 1/2 the performance from selecting "High performance" rather than "cost/performance" on the design point