ECE 563
Programming Parallel Machines

What is our goal in this class?

- To learn how to write programs that run *in parallel*
- This requires partitioning, or breaking up the program, so that different parts of it run on different cores or nodes
  - *different parts* may be different iterations of a loop
  - *different parts* can be different textual parts of the program
  - *different parts* can be both of the above

Normally code runs sequentially

All modern processors are *multi-core* processors
Our programs normally run on a single *core* within a processor
If we can efficiently run our code on multiple processors, we can make it g

```
for (i=1; i<n; i++) {
  a[i] = b[i] + c[i];
  c[i] = a[i-1]
}
```

```
a[1] = b[1] + c[1];
c[1] = a[0]
a[2] = b[2] + c[2];
c[2] = a[1]
a[3] = b[3] + c[3];
c[3] = a[3]
. . .
a[n] = b[n] + c[n];
c[n] = a[n]
```

We'd like to run it in parallel

```
for (i=1; i<n; i++) {
  a[i] = b[i] + c[i];
  c[i] = a[i-1]
}
```

Core 0

```
a[1] = b[1] + c[1];
c[1] = a[0]
a[2] = b[2] + c[2];
c[2] = a[1]
. . .
a[n/2] = b[n/2] + c[n/2];
c[n/2] = a[n/2]
```

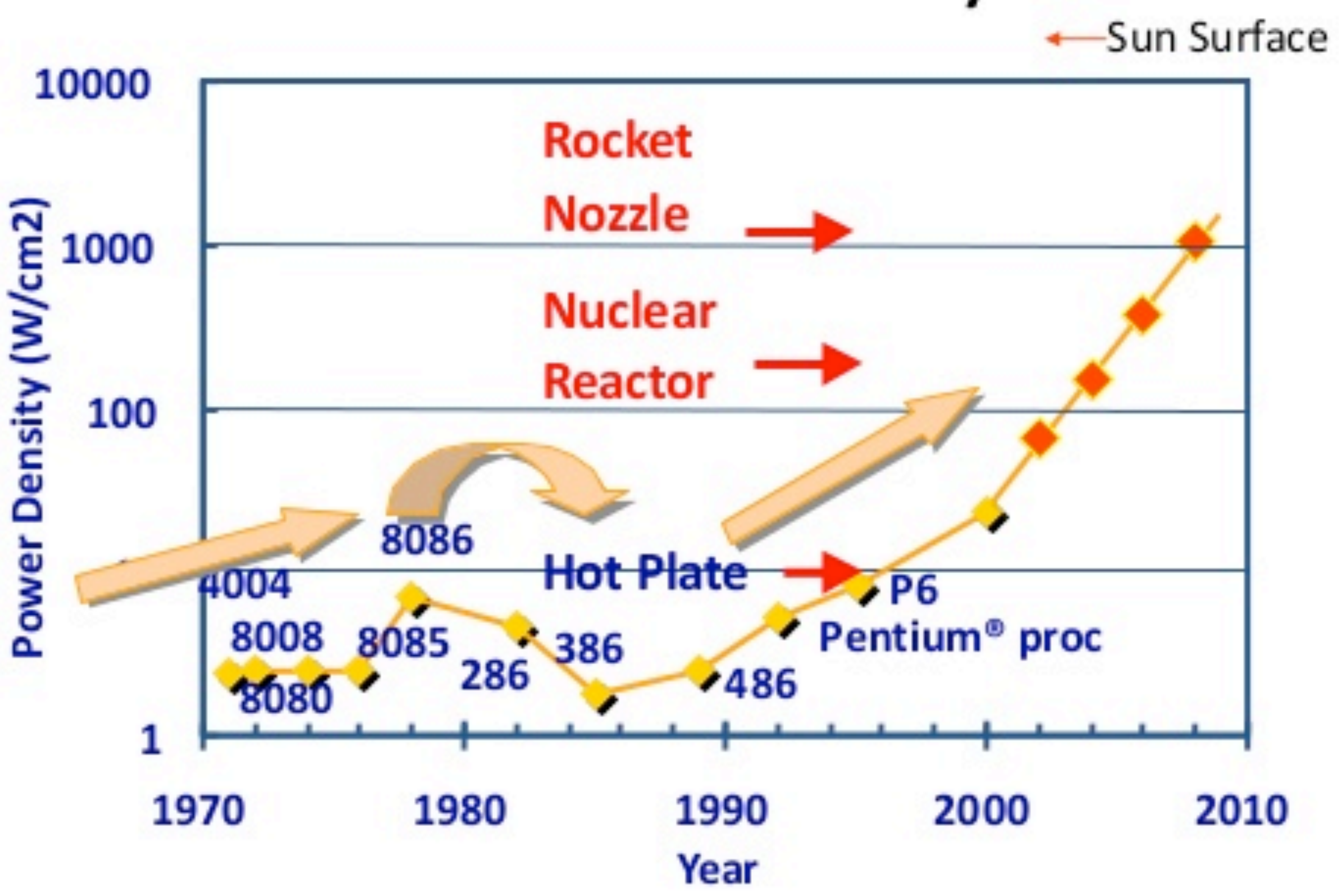
Core 1

```
a[n/2+1] = b[n/2+1] + c[n/2+1];
c[n/2+1] = a[n/2+1]
a[n/2+2] = b[n/2+2] + c[n/2+2];
c[n/2+2] = a[n/2+2]
. . .
a[n] = b[n] + c[n];
c[n] = a[n]
```

Why do we want to run in parallel

- Life was simpler when processor clock rates doubled every couple of years or so
- Processors got faster, enabling more complicated software, when motivated faster processors (and buying a new machine) which motivated even more complicated software . . .
- *If something cannot go on forever, it will stop.* --Stein's Law, first pronounced in the 1980s
  - Always true of exponentials
  - $E = 1/2*C*V^2$, where E is energy, C is capacitance and V is voltage.
  - Higher frequencies require higher voltages
  - More cores increase C, which increases energy linearly

Power density



Power density too high to keep junctions at low temp

Courtesy, Intel

The solution

- Processors don’t get faster, but have more cores
- By splitting up our program and running it on multiple cores, it runs faster.
- Unfortunately, it is up to us to split it up and make it run faster. Which is why we’re here for the next two weeks.

And it's not always legal to run code in parallel

- And sometimes the hardware cannot support what we want to do
  - Instead of 2 cores, what if we want to run our code on 400 cores? 4,000?
- And sometimes we run our code in parallel and it runs slower, not faster

In this class we'll learn

- Computer Architecture:
  - What the computer looks like that runs our code,
  - How computer features can affect our performance
- OpenMP and Pthreads: how to run programs in parallel using the cores on a single chip, or on two chips
- MPI: how to run programs in parallel across multiple *nodes* in a large computer
- Cuda: how to run programs in parallel on a GPU
- Speedup theory: measuring program performance

And we will do programming

- OpenMP and Pthreads: how to run programs in parallel using the cores on a single chip, or on two chips
- MPI: how to run programs in parallel across multiple *nodes* in a large computer

A short architectural overview

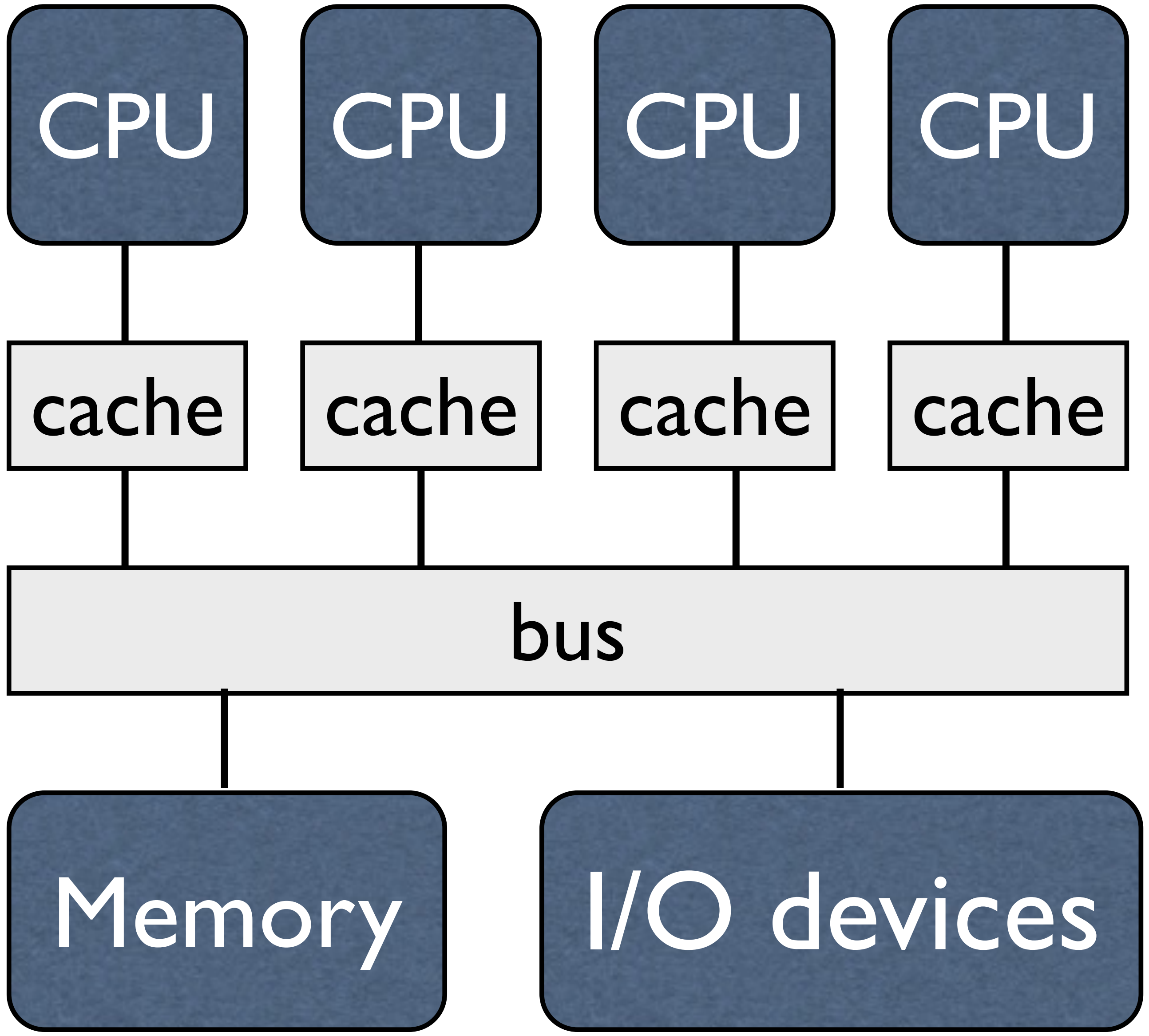
Warning: gross simplifications to follow

Multiprocessor (shared memory multiprocessor)

- Multiple CPUs with a shared memory (or multiple cores in the same CPU)
- The same address on two different processors points to the same memory location
- Multicores are a version of this
- If multiple processors are used, they are connected to a *shared bus* which allows them to communicate with one another via the shared memory
- Two variants:
  - *Uniform memory access*: all processors access all memory in the same amount of time
  - *Non-uniform memory access*: different processors may see different times to access some memory.

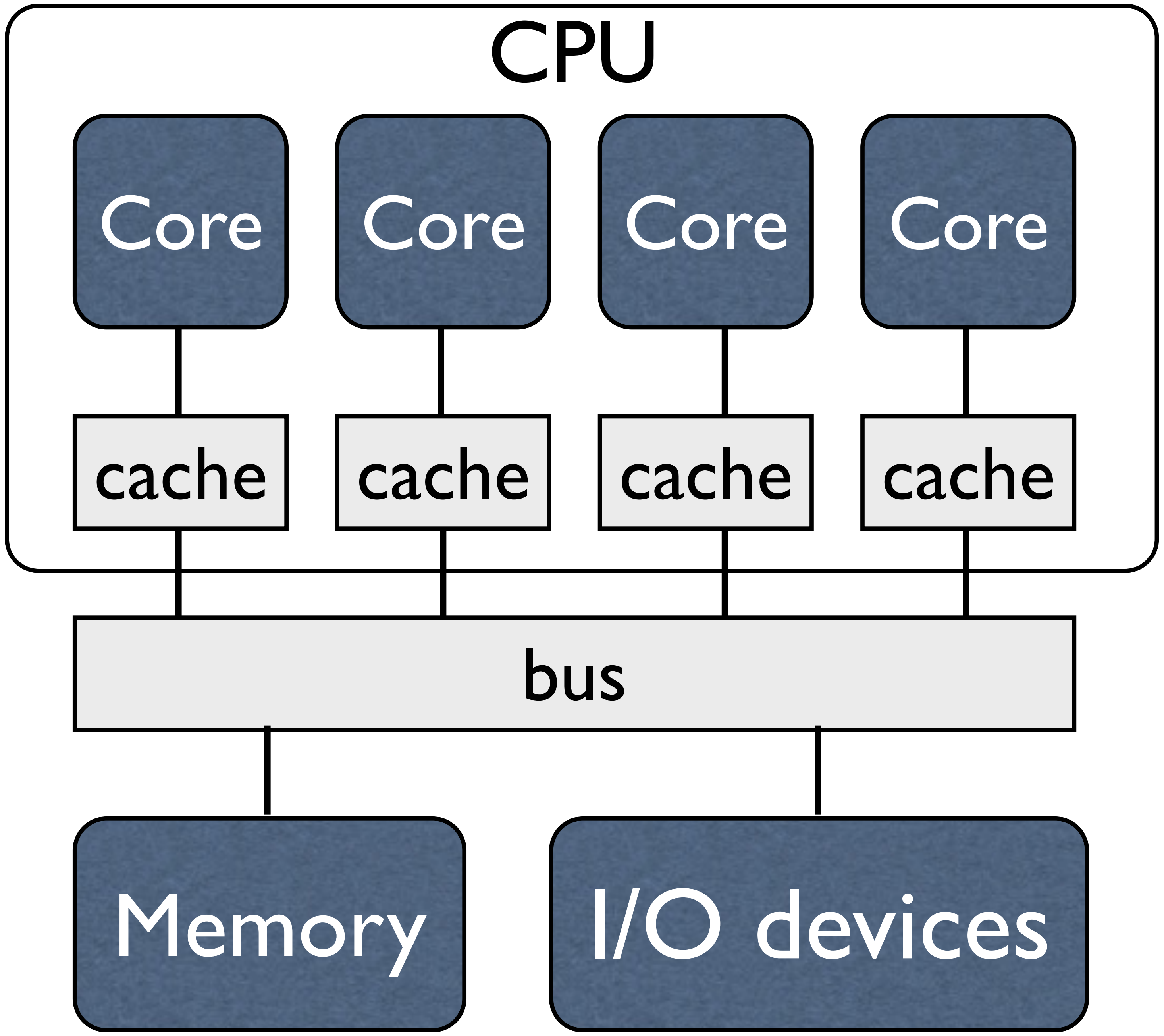
A Uniform Memory Access shared memory machine

All processors access *global memory* at the same speed



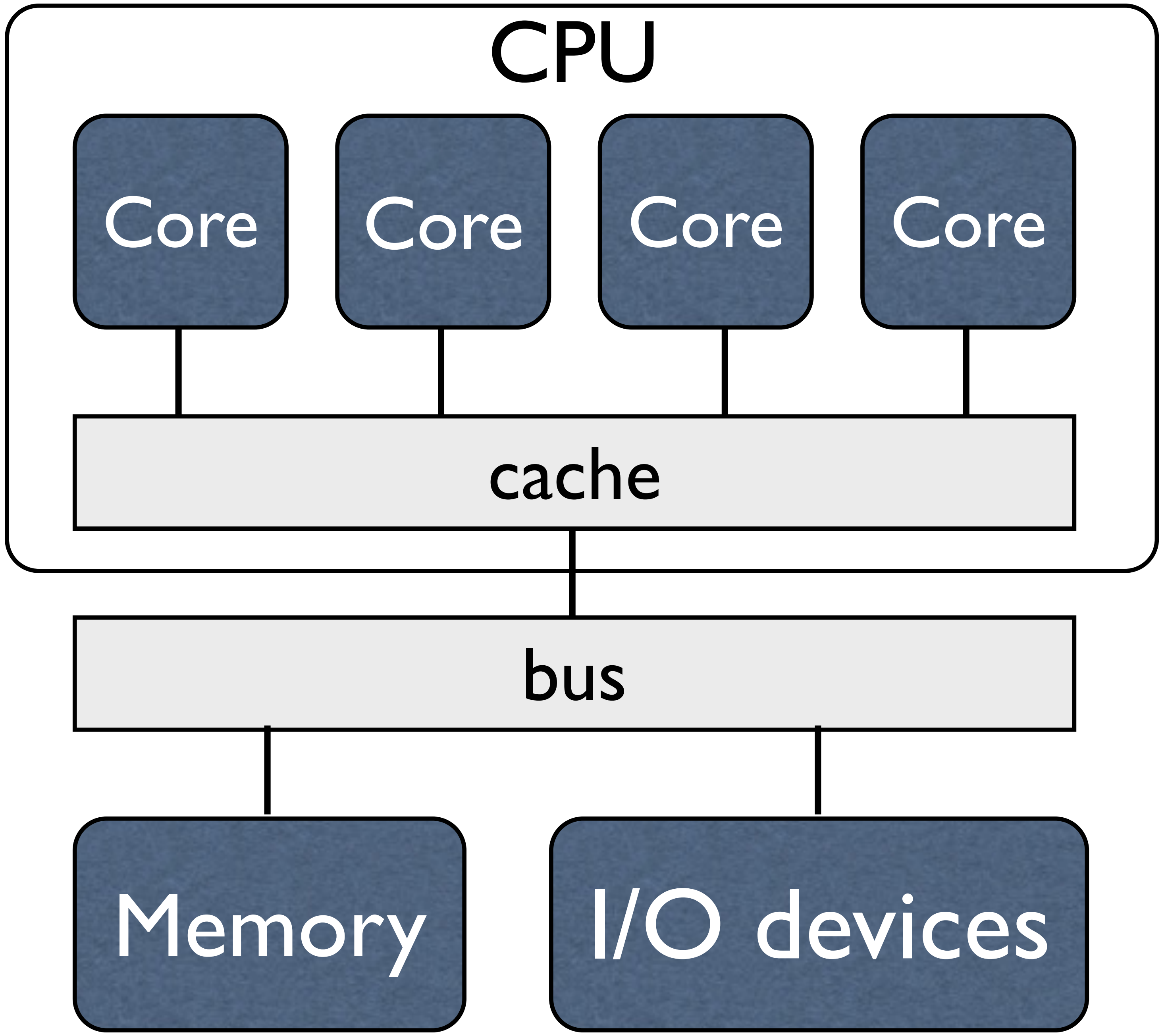
Multicore machines usually have uniform memory access

All cores access global memory at the same speed

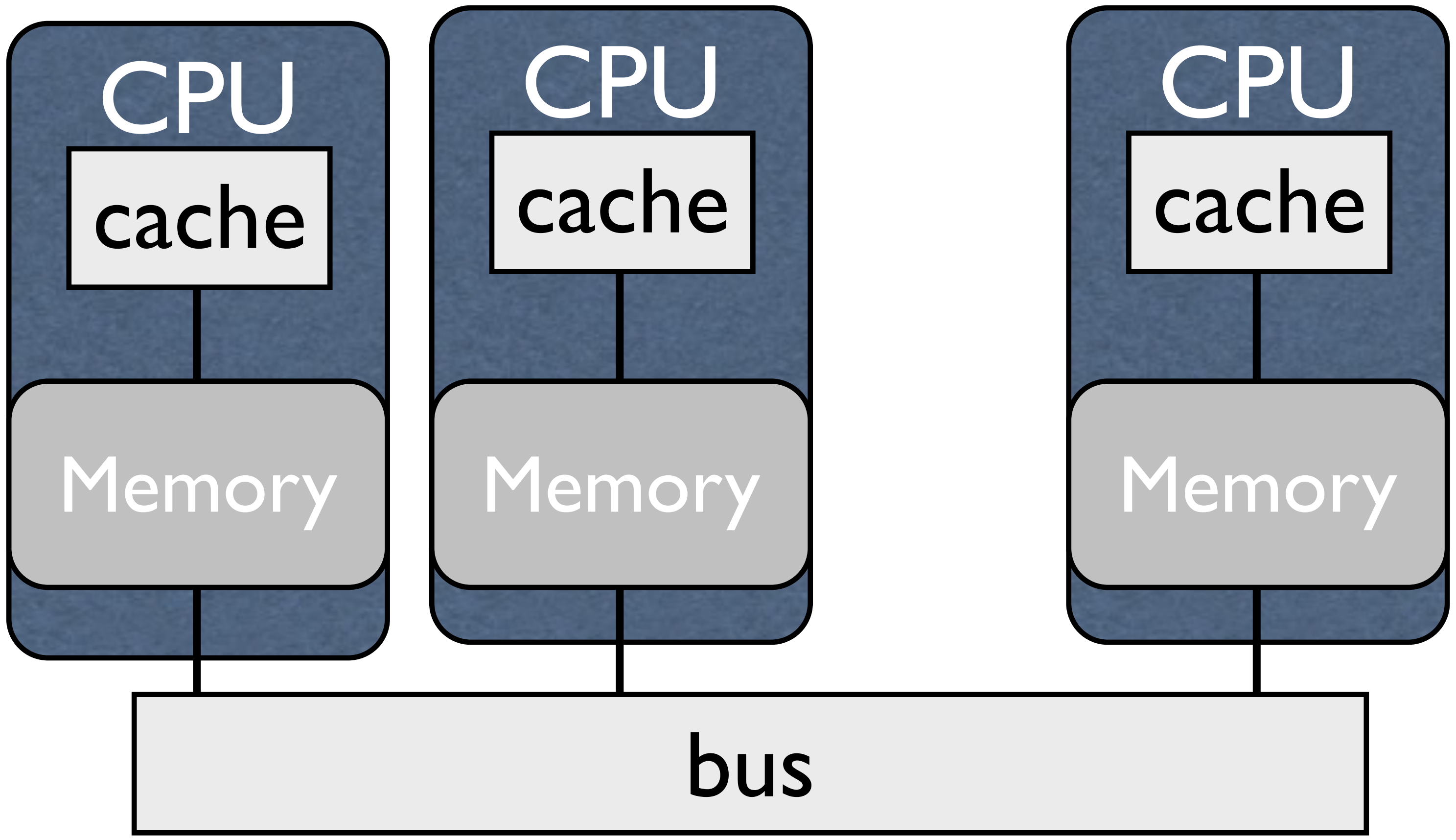


Multicore machines usually share at least one *level* of cache

All cores access global memory at the same speed



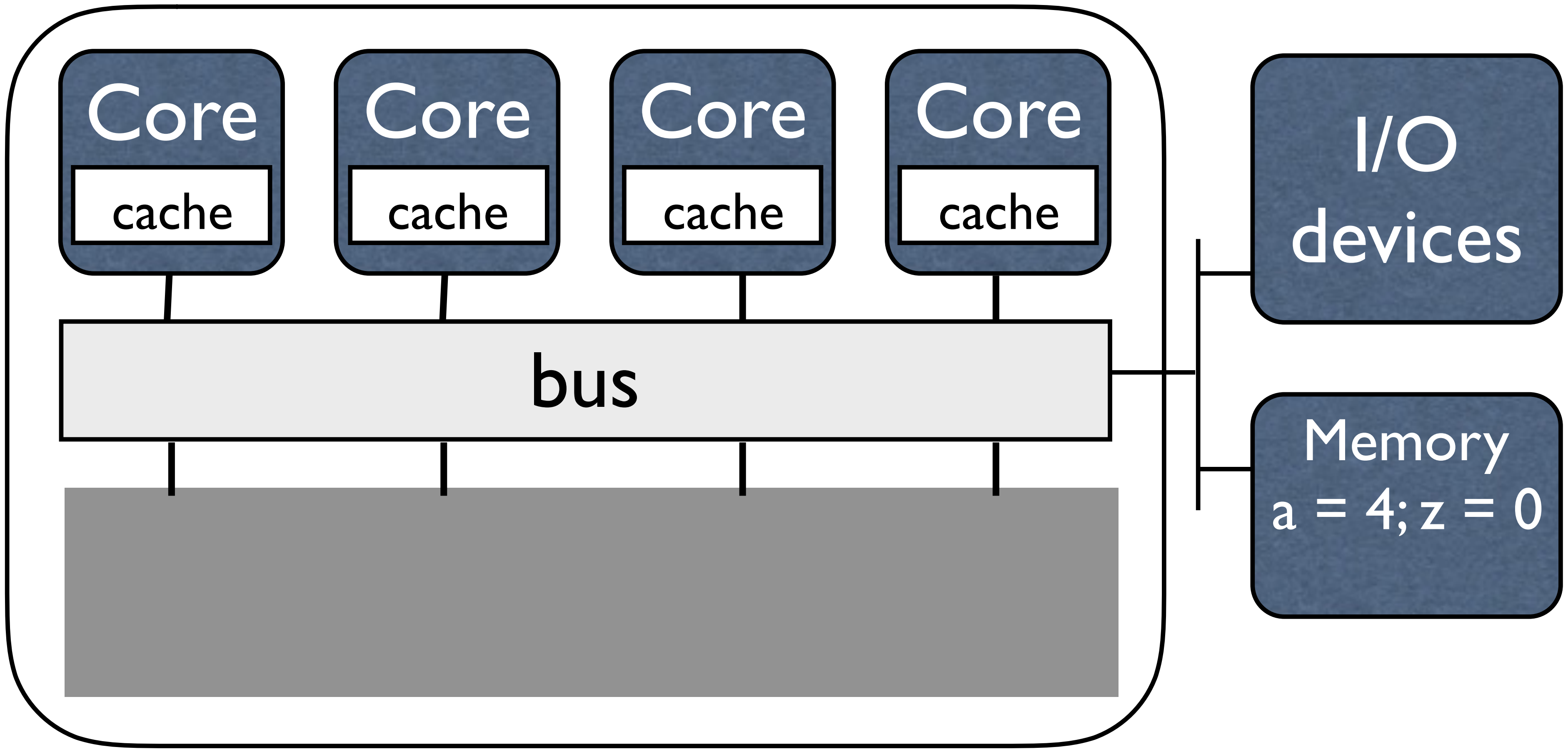
A NUMA (non-uniform memory access) shared memory machine



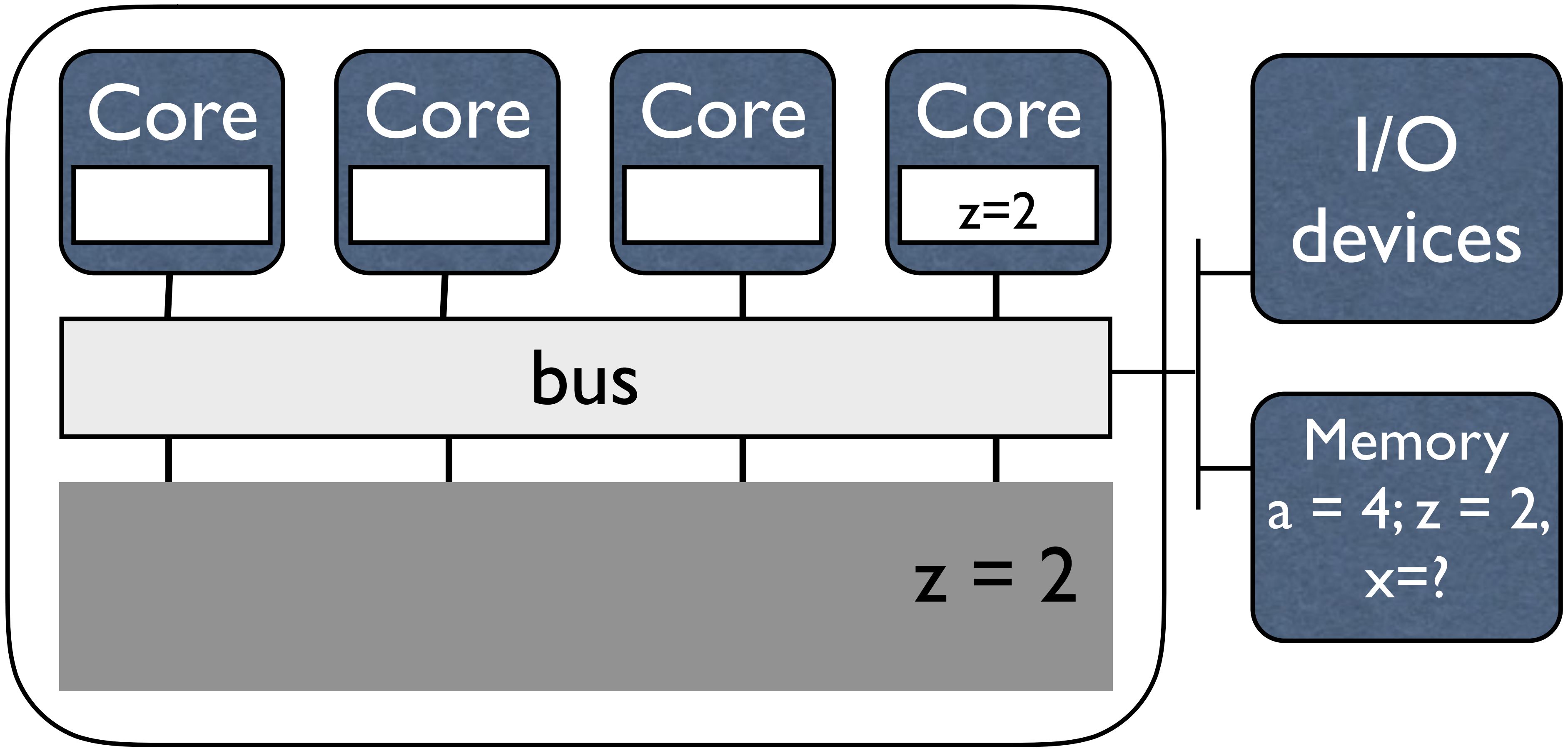
Global memory is spread across, and held in, the local memories of the different nodes of the machine

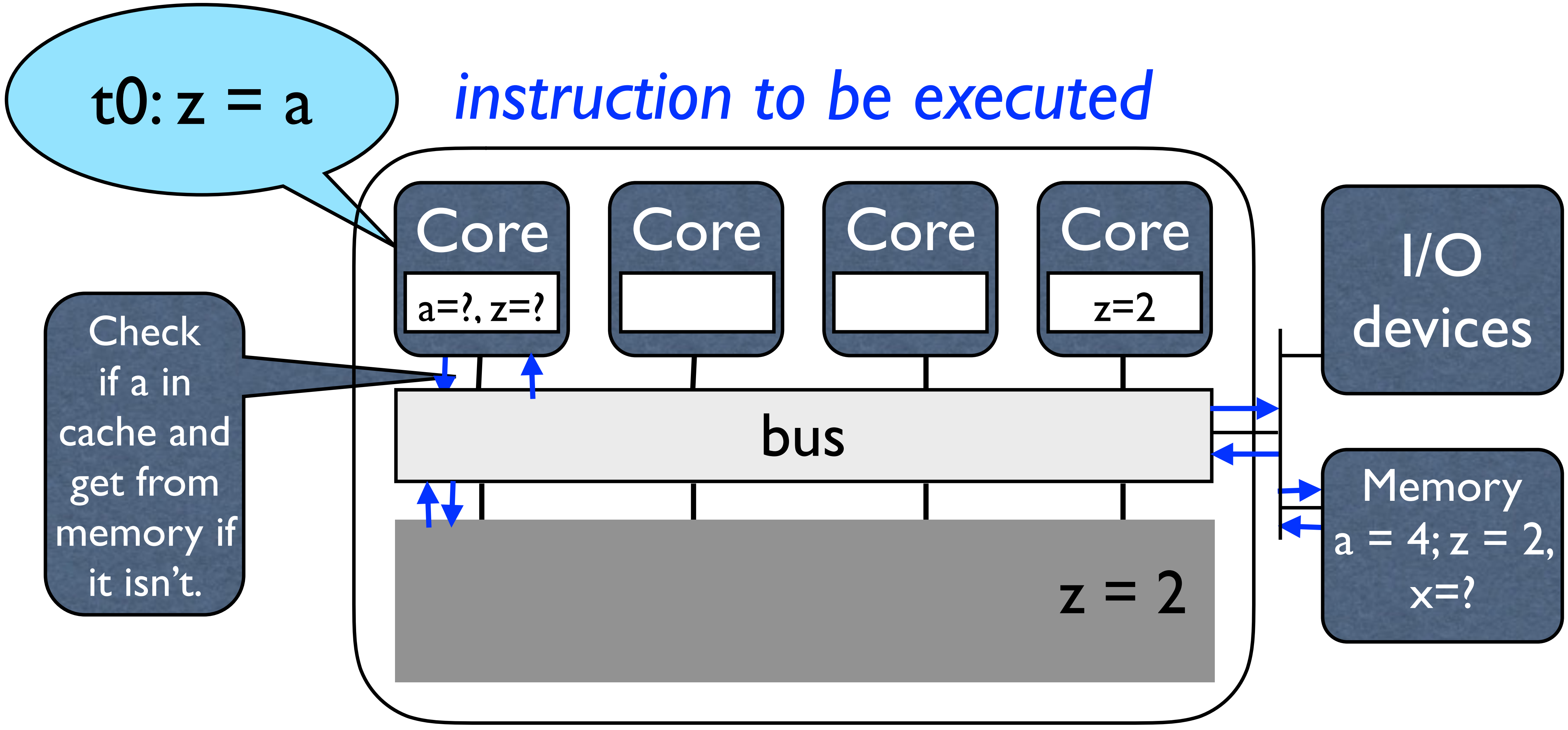
Processors will access their memory faster than their neighbors memory

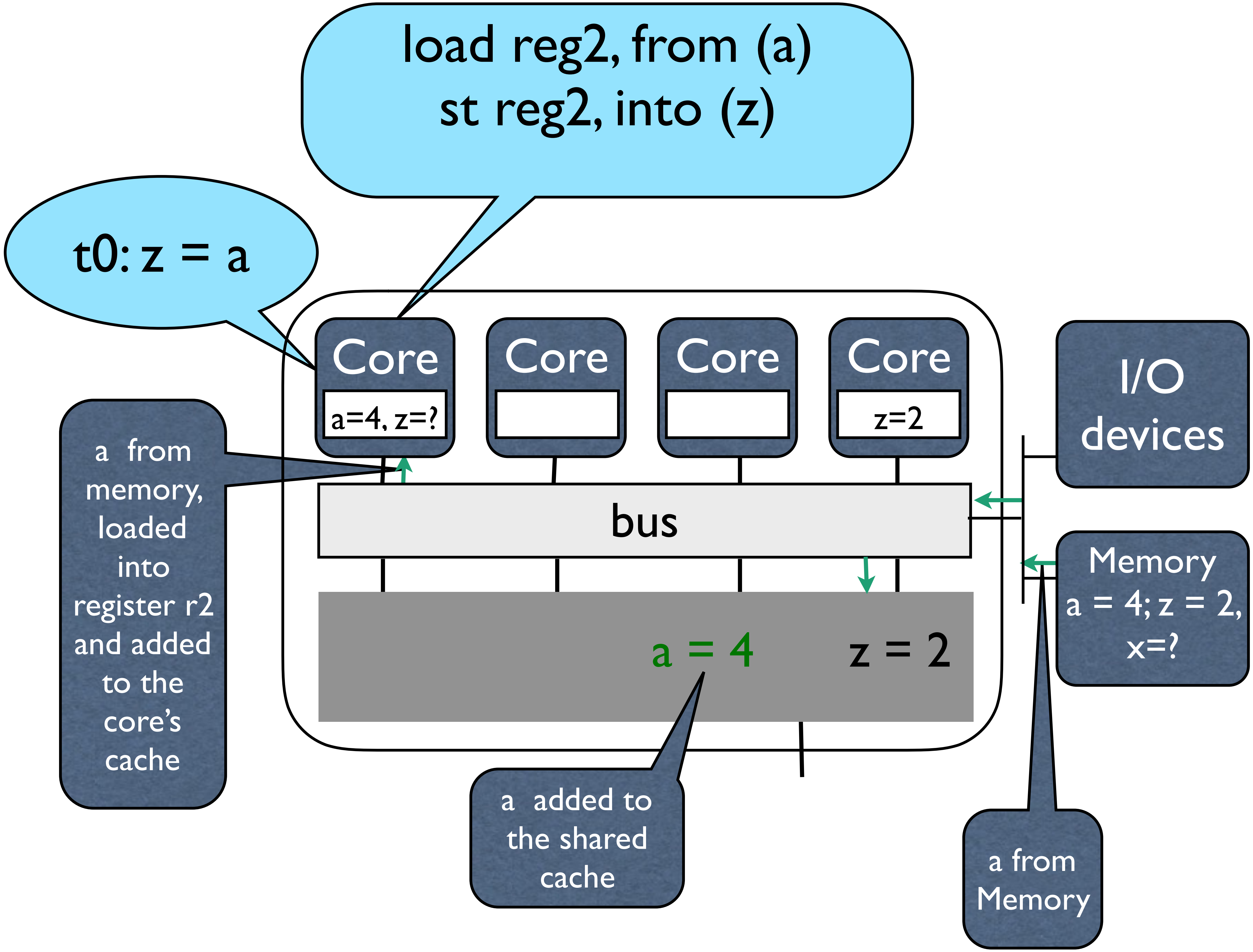
Coherence is needed

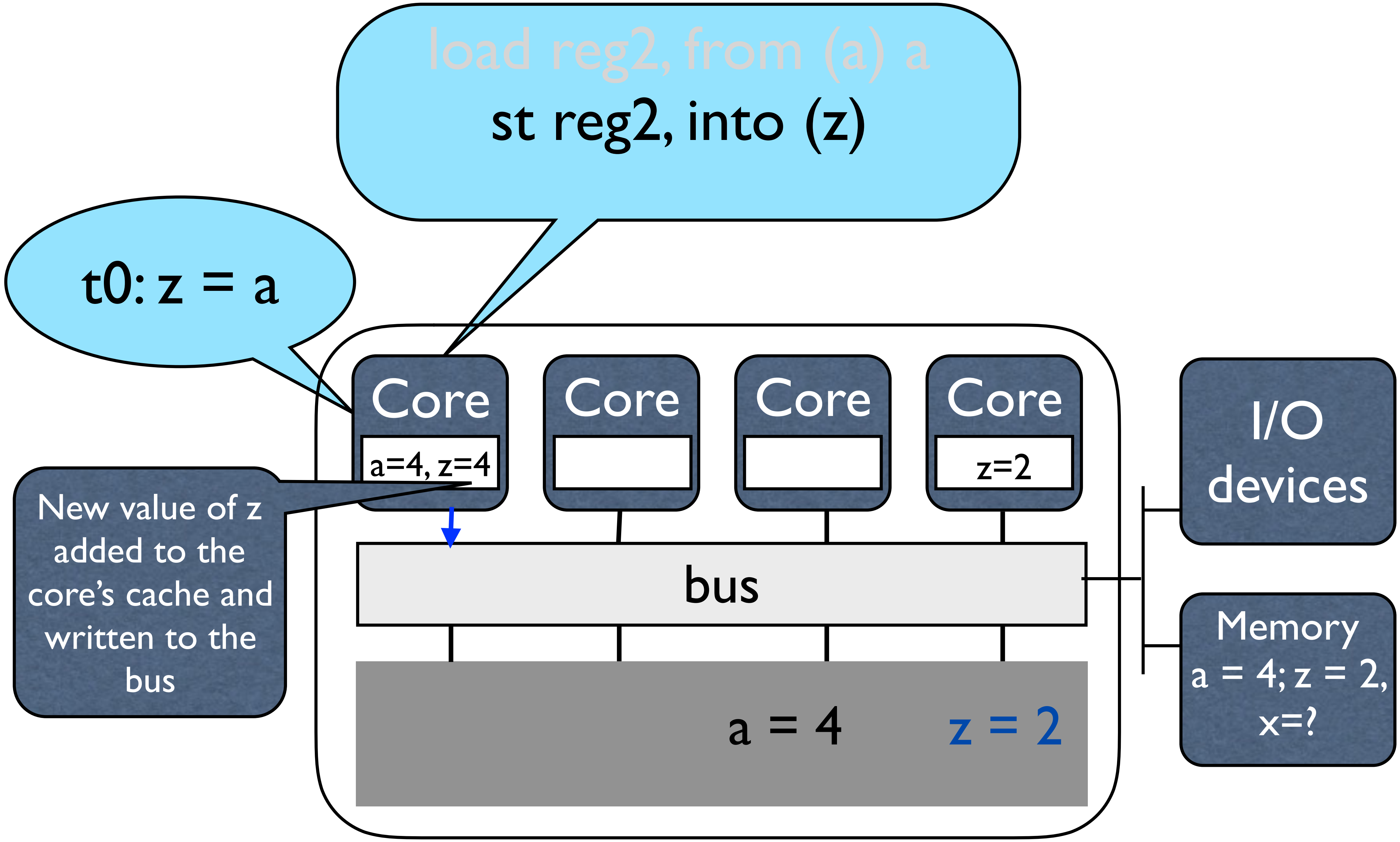


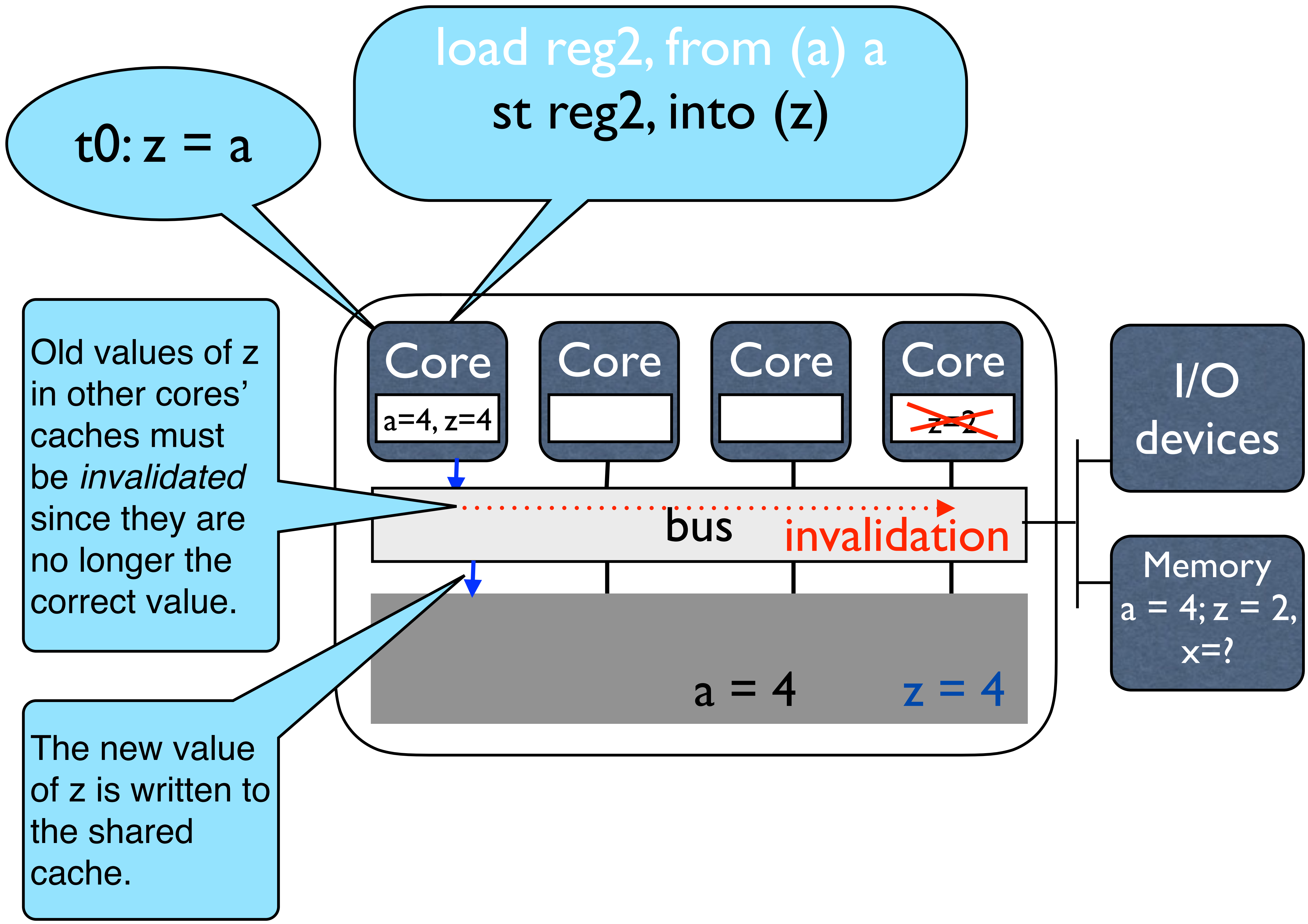
Coherence is needed

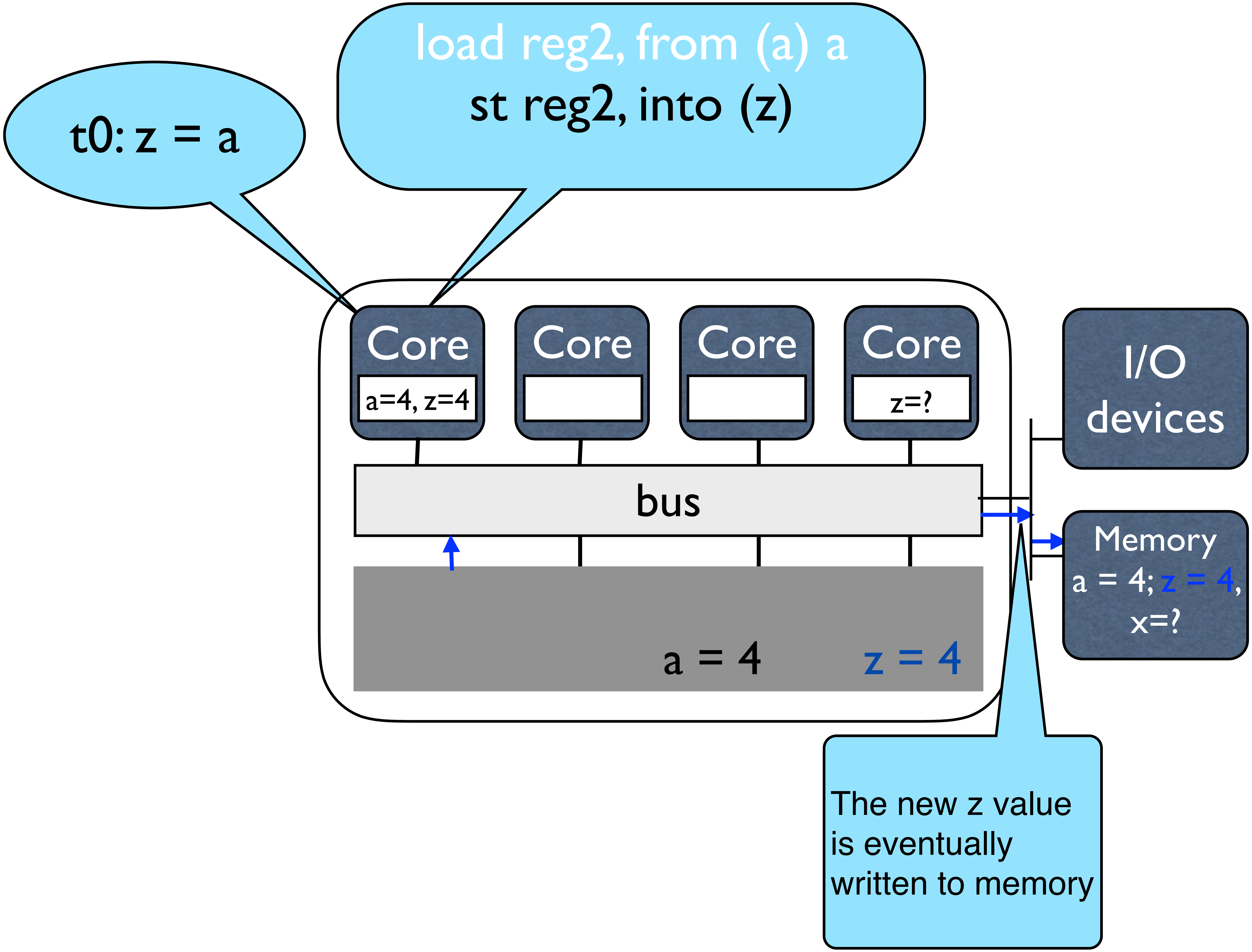


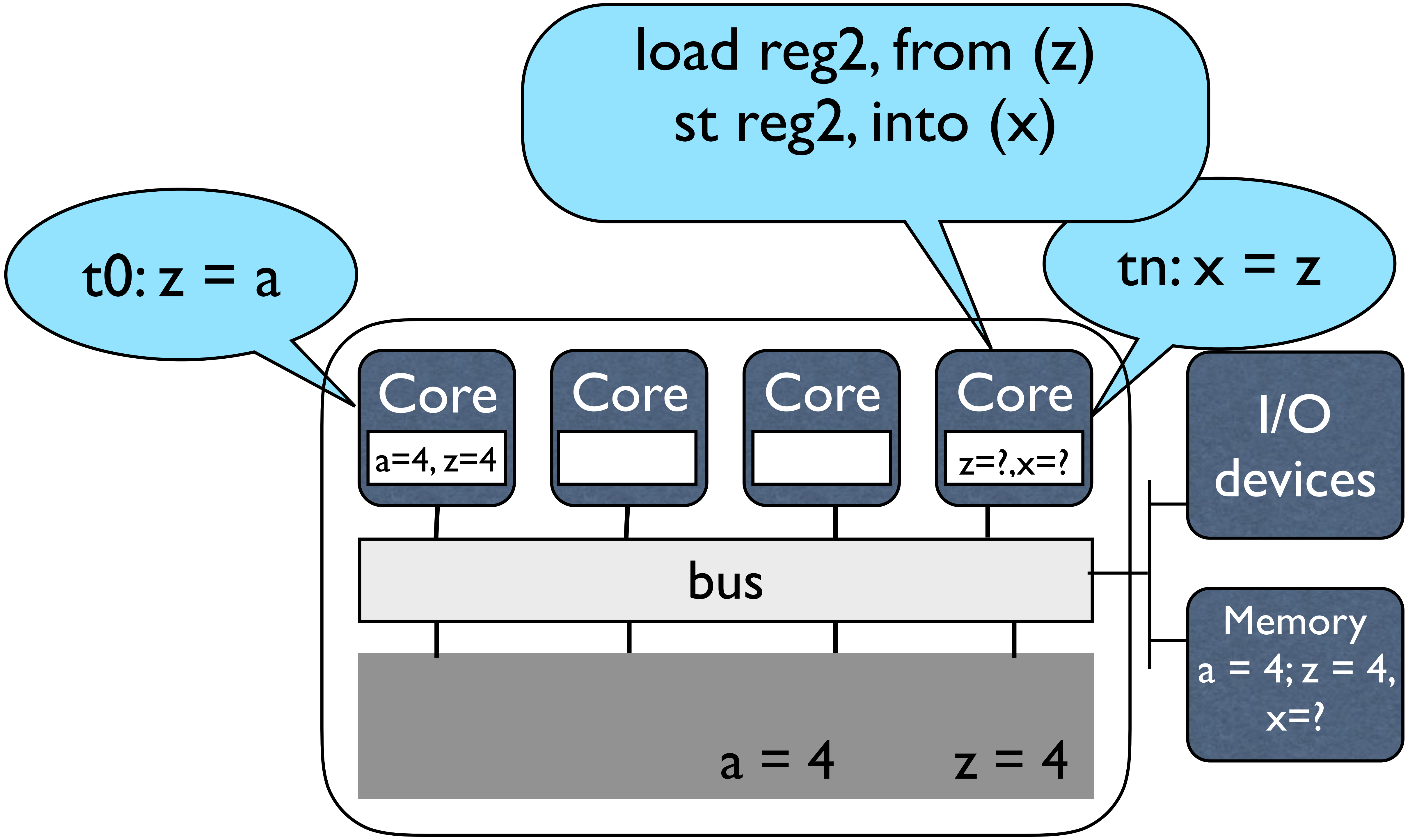

t0: z = a
instruction to be executed
Check if a in cache and get from memory if it isn't.
Core
a=?, z=?
Core
Core
Core
z=2
bus
z = 2
I/O devices
Memory a = 4; z = 2, x=?

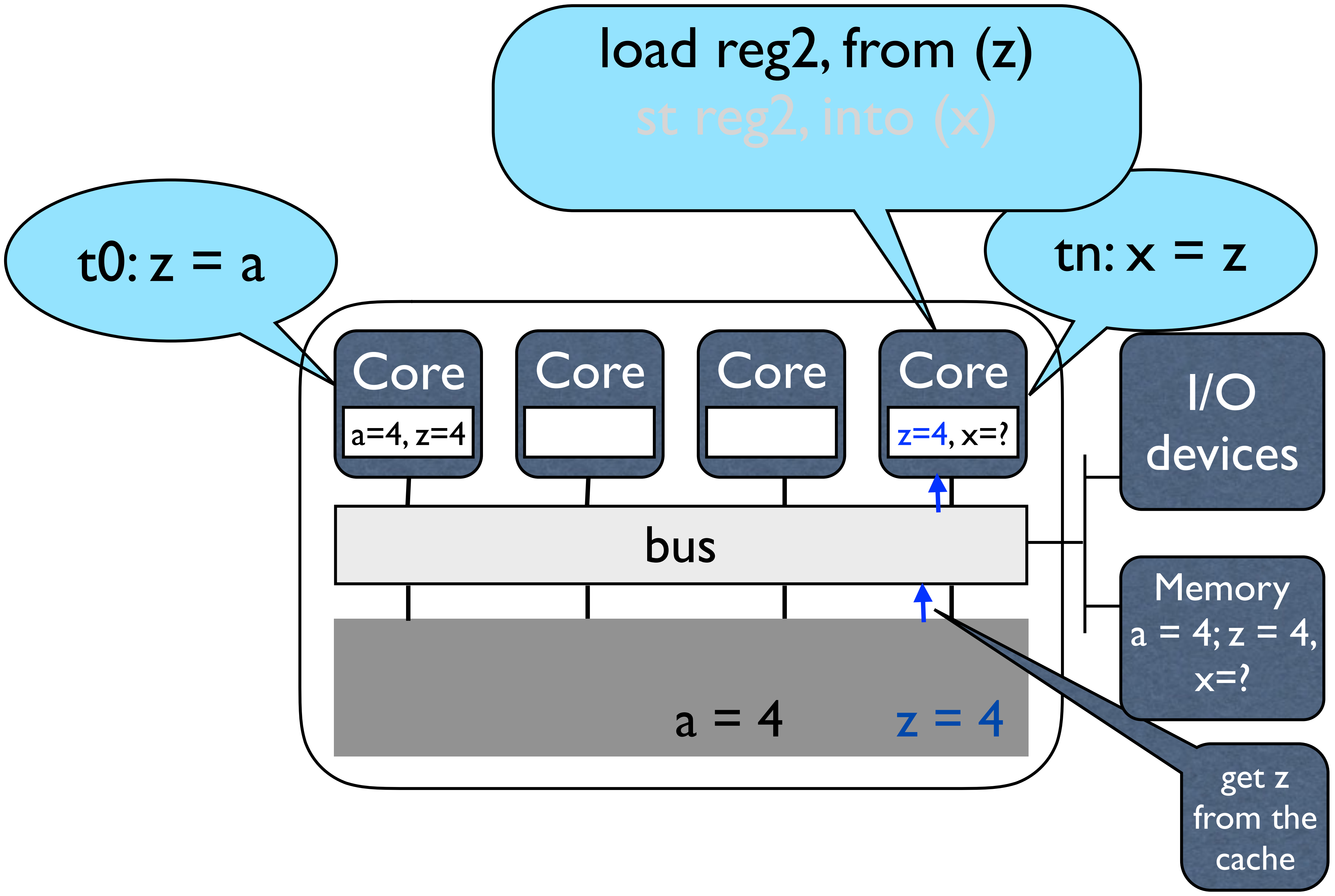

load reg2, from (a)
st reg2, into (z)
t0: z = a
Core
a=4, z=?
Core
Core
Core
z=2
I/O devices
bus
Memory a = 4; z = 2, x=?
a = 4
z = 2
a from memory, loaded into register r2 and added to the core's cache
a added to the shared cache
a from Memory

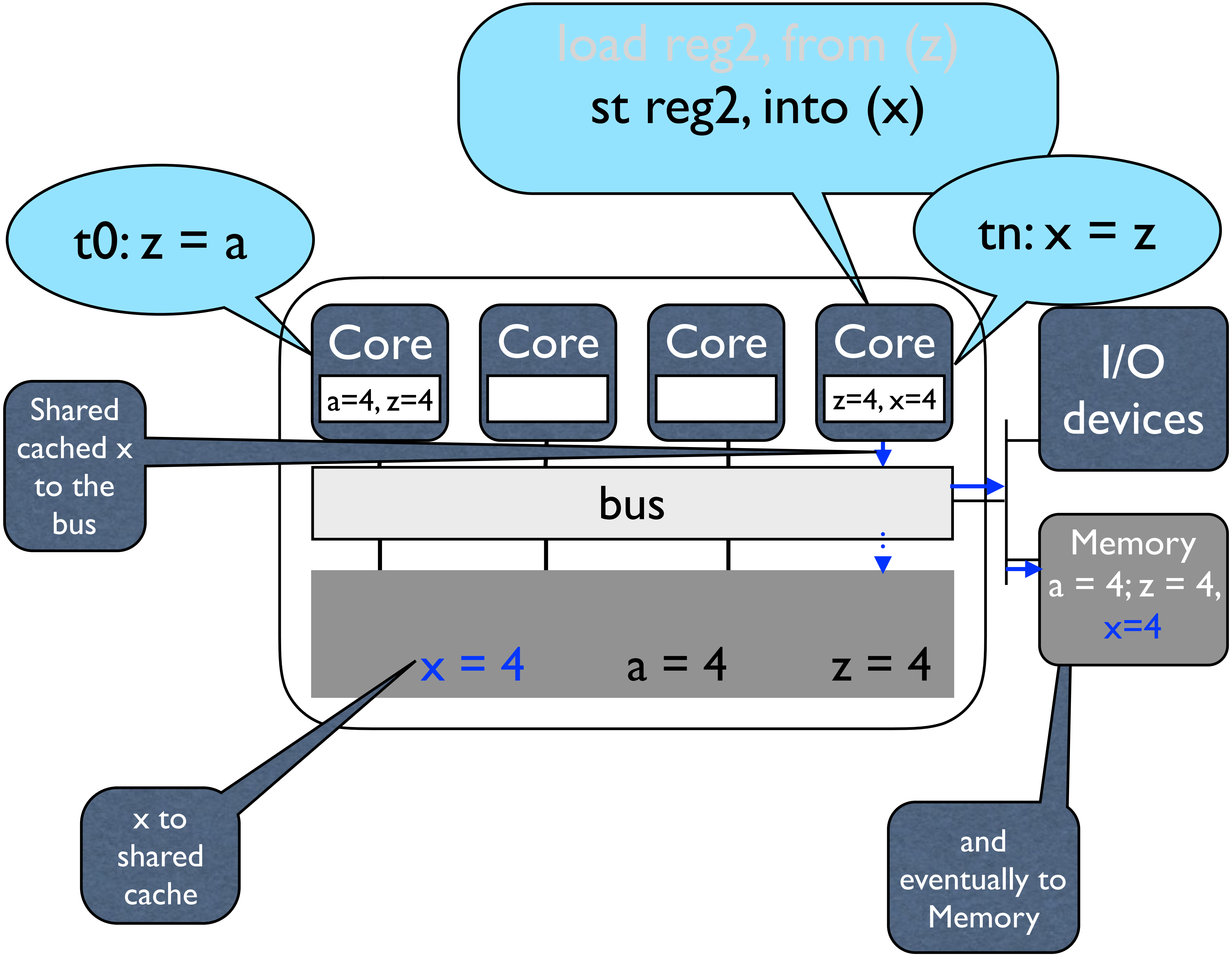

load reg2, from (a) a
st reg2, into (z)
t0: z = a
Core
a=4, z=4
Core
Core
Core
z=2
New value of z added to the core's cache and written to the bus
bus
a = 4
z = 2
I/O devices
Memory a = 4; z = 2, x=?

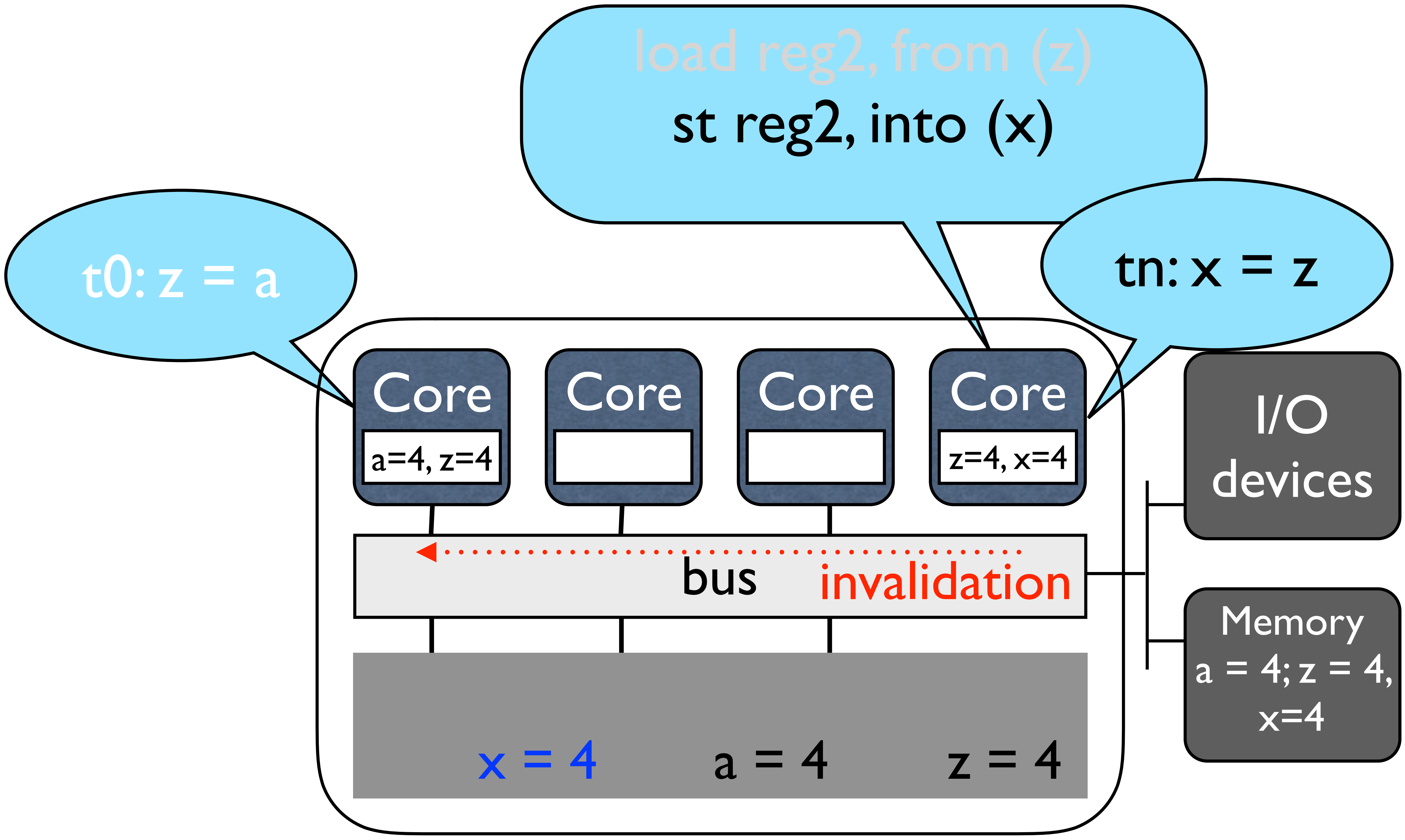

t0: z = a
load reg2, from (a) a
st reg2, into (z)
Old values of z in other cores' caches must be *invalidated* since they are no longer the correct value.
The new value of z is written to the shared cache.
Core
a=4, z=4
Core
Core
Core
~~z=2~~
bus
invalidation
a = 4
z = 4
I/O devices
Memory
a = 4; z = 2, x=?


t0: z = a
load reg2, from (a) a
st reg2, into (z)
Core
a=4, z=4
Core
Core
Core
z=?
I/O devices
bus
Memory
a = 4; z = 4, x=?
a = 4
z = 4
The new z value is eventually written to memory

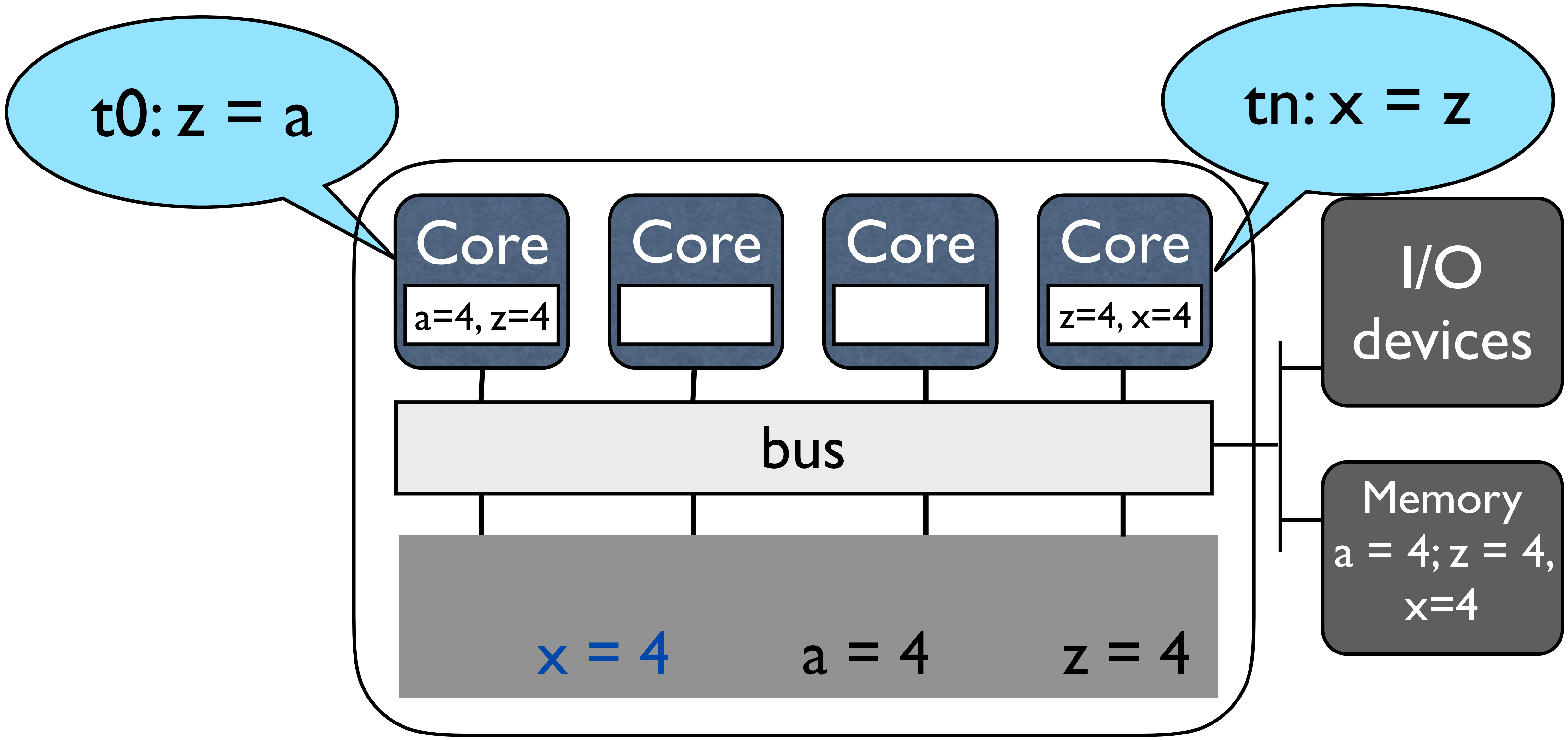

t0: z = a
load reg2, from (z)
st reg2, into (x)
tn: x = z
Core
a=4, z=4
Core
Core
Core
z=?,x=?
I/O devices
bus
Memory
a = 4; z = 4,
x=?
a = 4
z = 4


load reg2, from (z)
st reg2, into (x)
t0: z = a
tn: x = z
Core
a=4, z=4
Core
Core
Core
z=4, x=?
I/O devices
bus
Memory a = 4; z = 4, x=?
a = 4
z = 4
get z from the cache

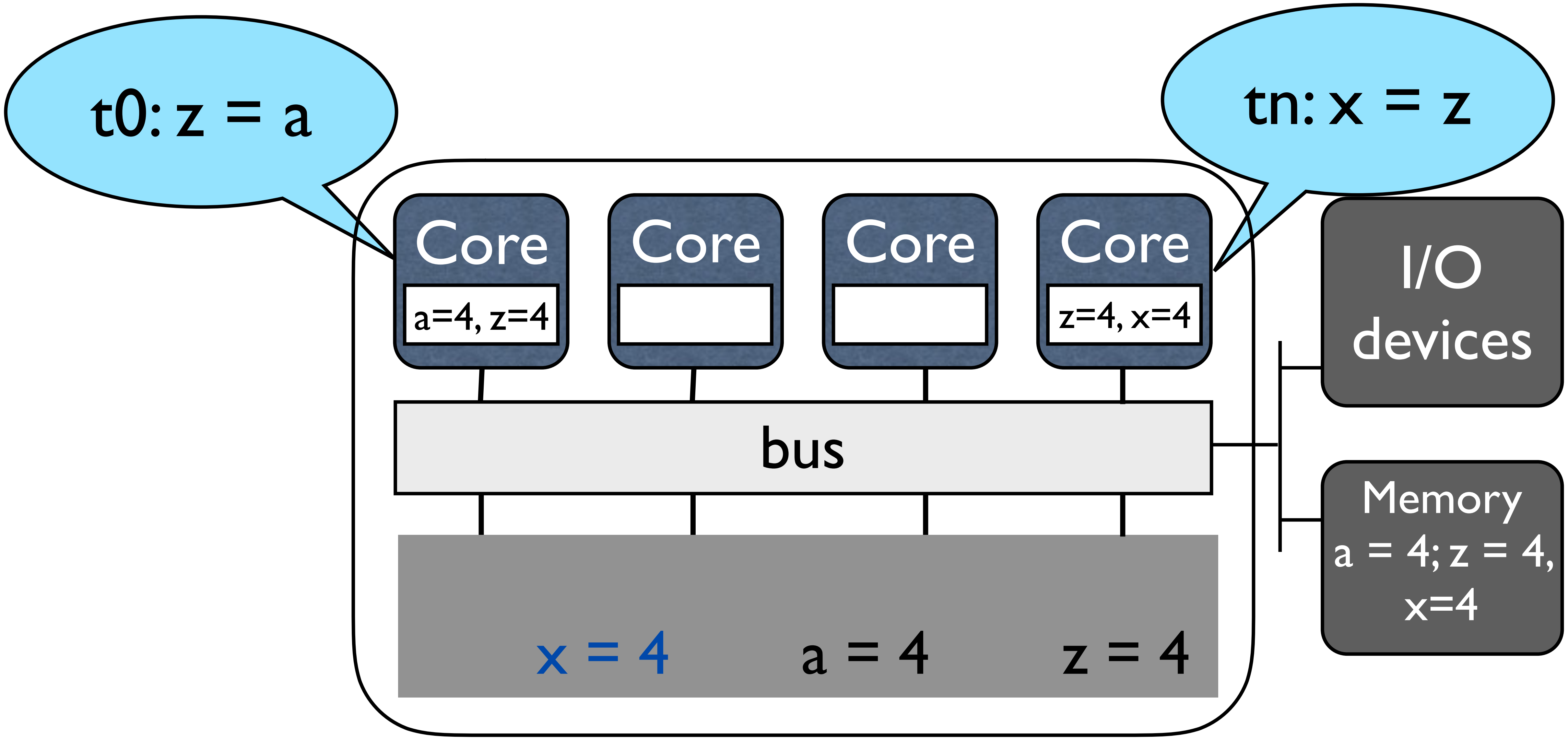

load reg2, from (z)
st reg2, into (x)
t0: z = a
tn: x = z
Core
a=4, z=4
Core
Core
Core
z=4, x=4
I/O devices
Shared cached x to the bus
bus
Memory a = 4; z = 4, x=4
x = 4
a = 4
z = 4
x to shared cache
and eventually to Memory



Whenever a write is made to memory by a core, an invalidation signal is sent to all of the other cores' caches. This is true even if the other cores don't have a copy of the written variable, since only the core knows what is in its cache.

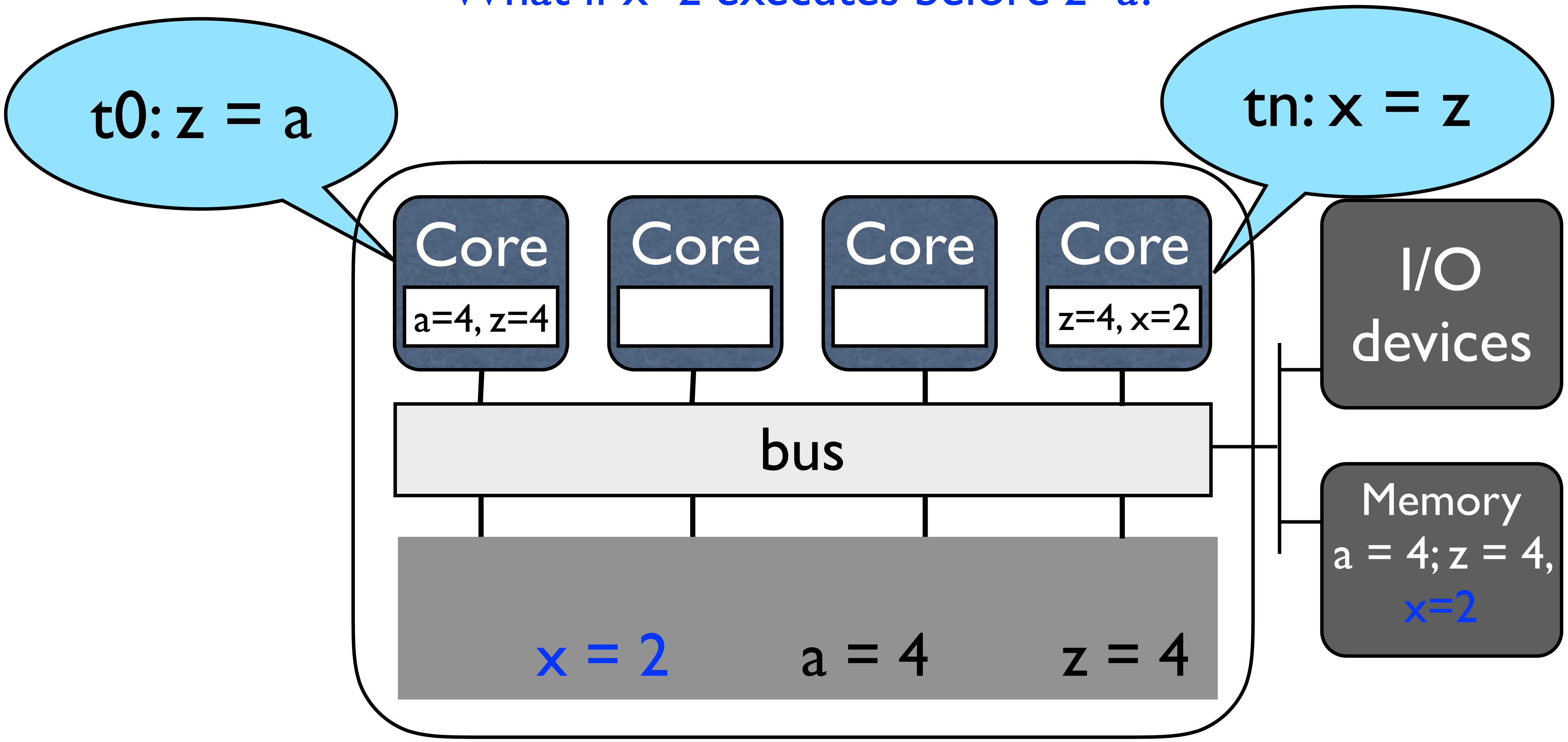

t0: z = a
tn: x = z
Core
a=4, z=4
Core
Core
Core
z=4, x=4
I/O devices
bus
Memory a = 4; z = 4, x=4
x = 4
a = 4
z = 4

Hardware makes sure a core/processor reads the latest value assigned to memory (cache coherence)



Software has to make sure operations occur in the right order across threads/processors

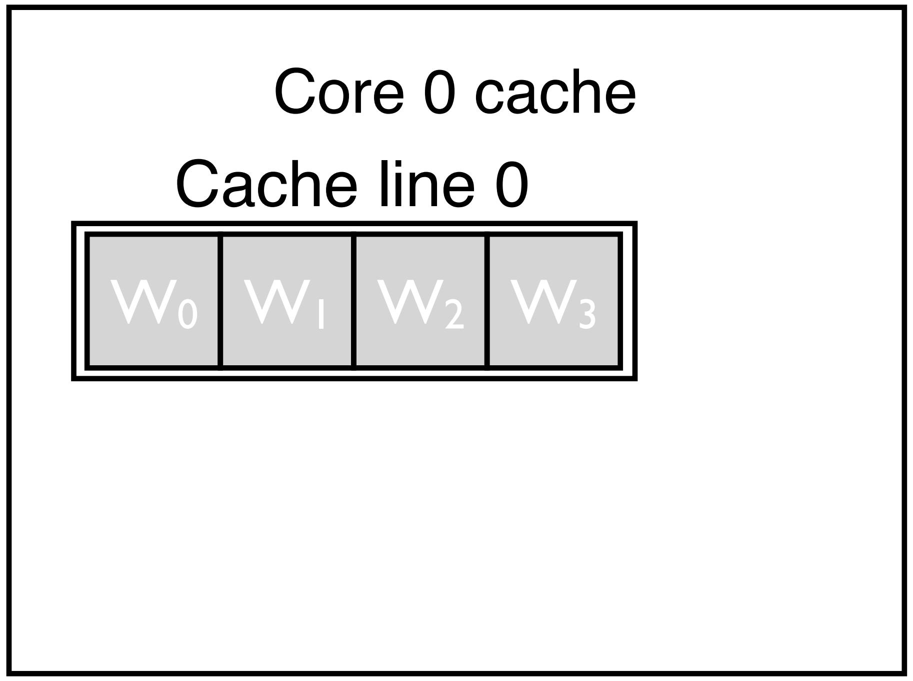
What if x=z executes before z=a?



A little more detail on caches

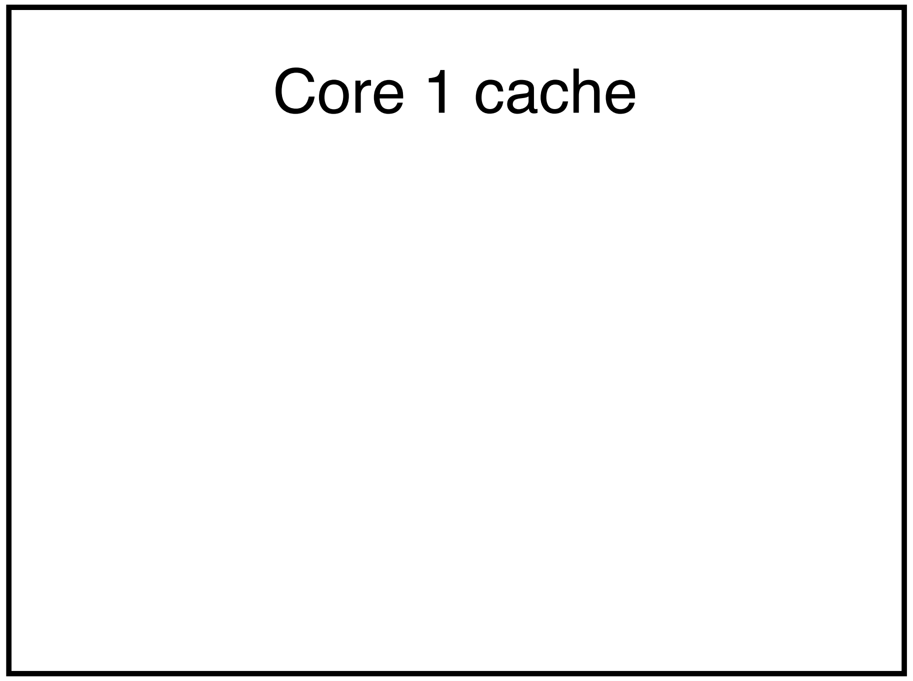
- When a value is brought into the cache, the processor doesn't just bring that value in
- It brings in an entire *cache line* of values
- Cache lines are typically 4 to 8 words long, or 16 to 32 bytes long
- This can help and hurt performance

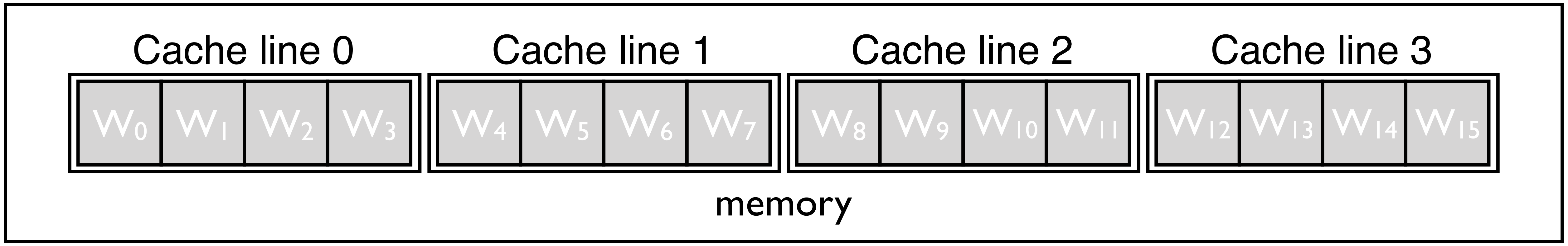
Memory brought into cache by lines



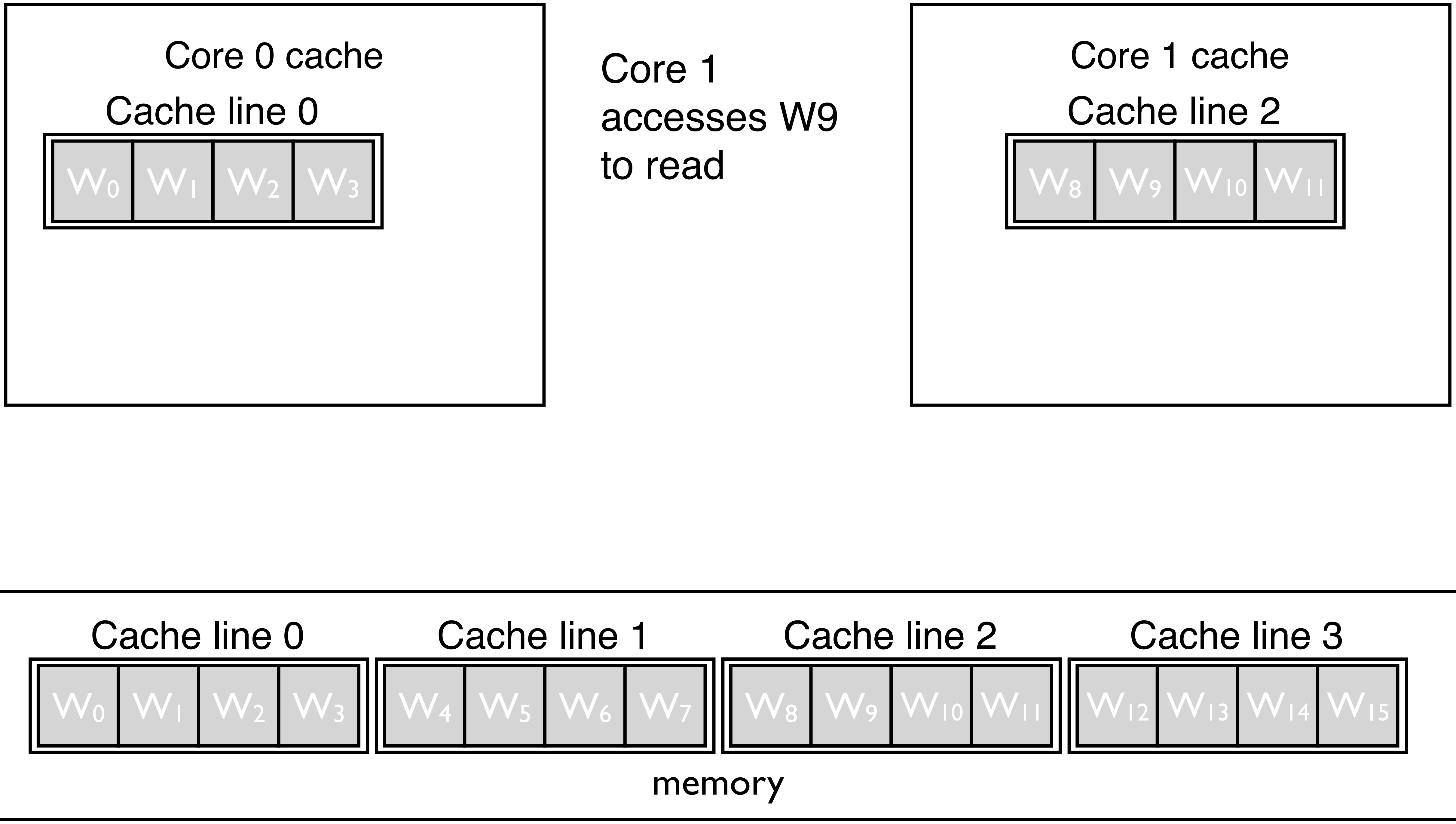
Core 0 accesses W_0 to read

If Core 0 accesses W_0, W_1, W_2, and W_3, only one memory access needed!



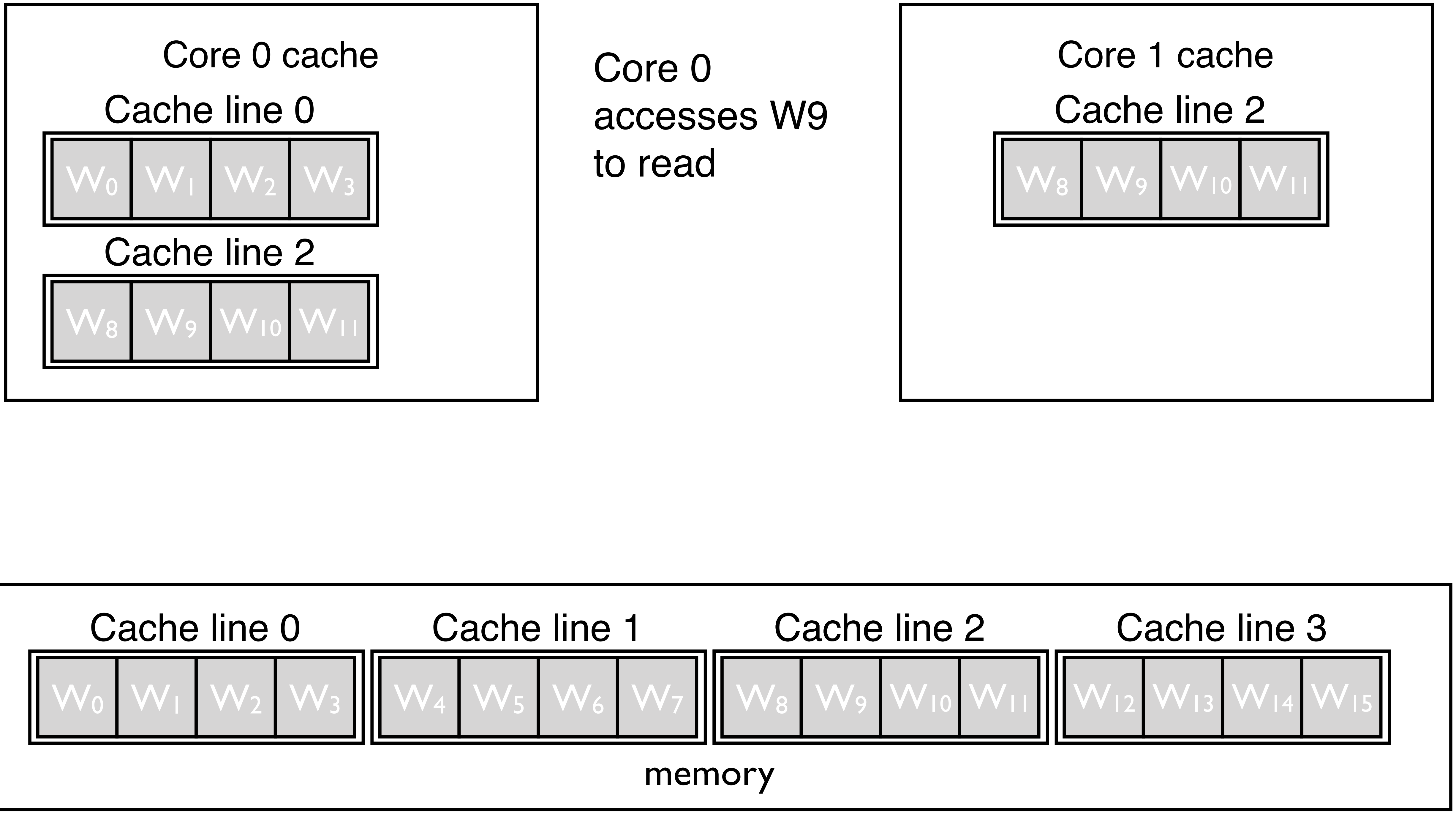


Memory brought into cache by lines

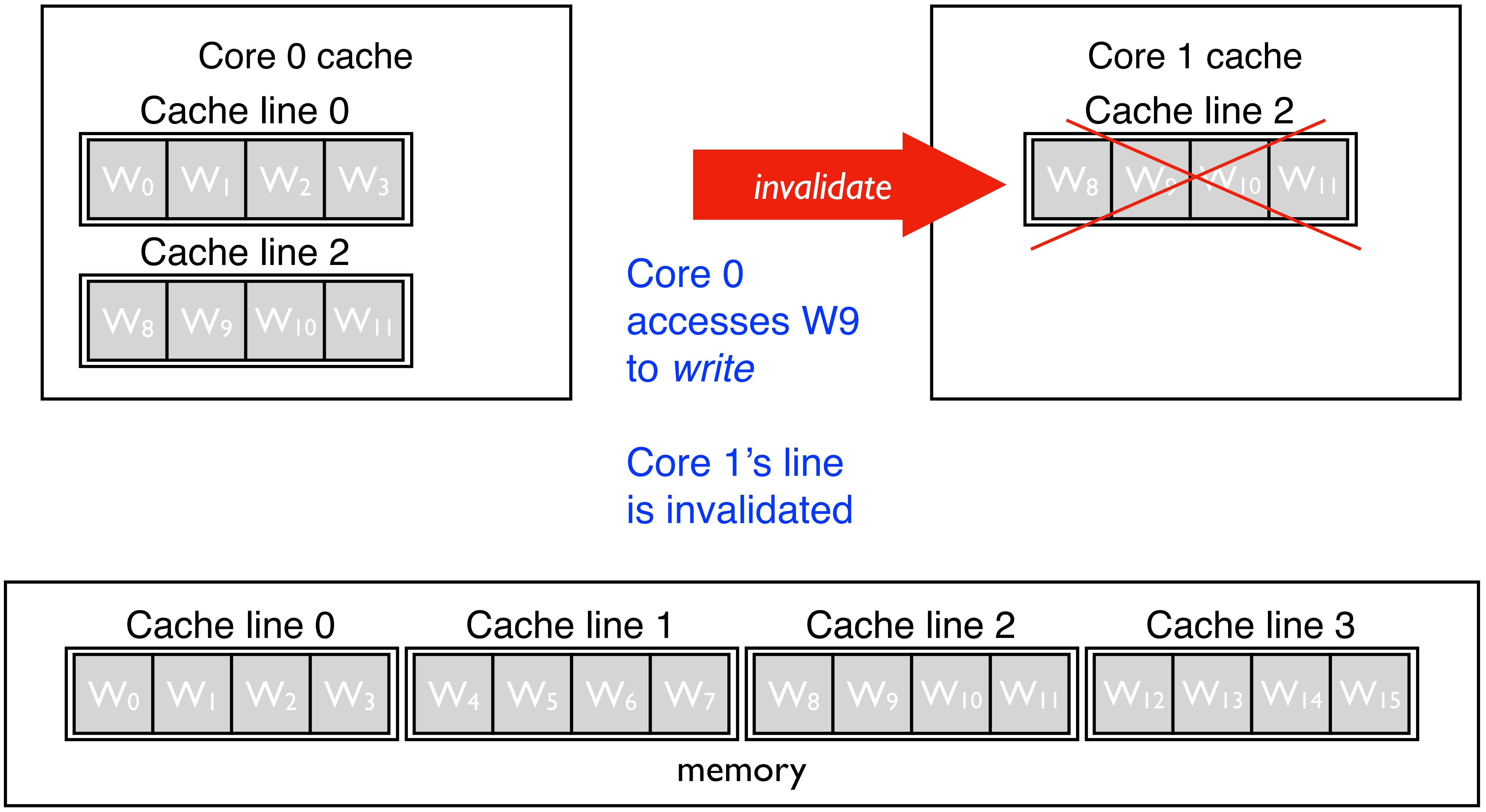


Core 1 accesses W9 to read

Many caches can hold the same line for *reading*



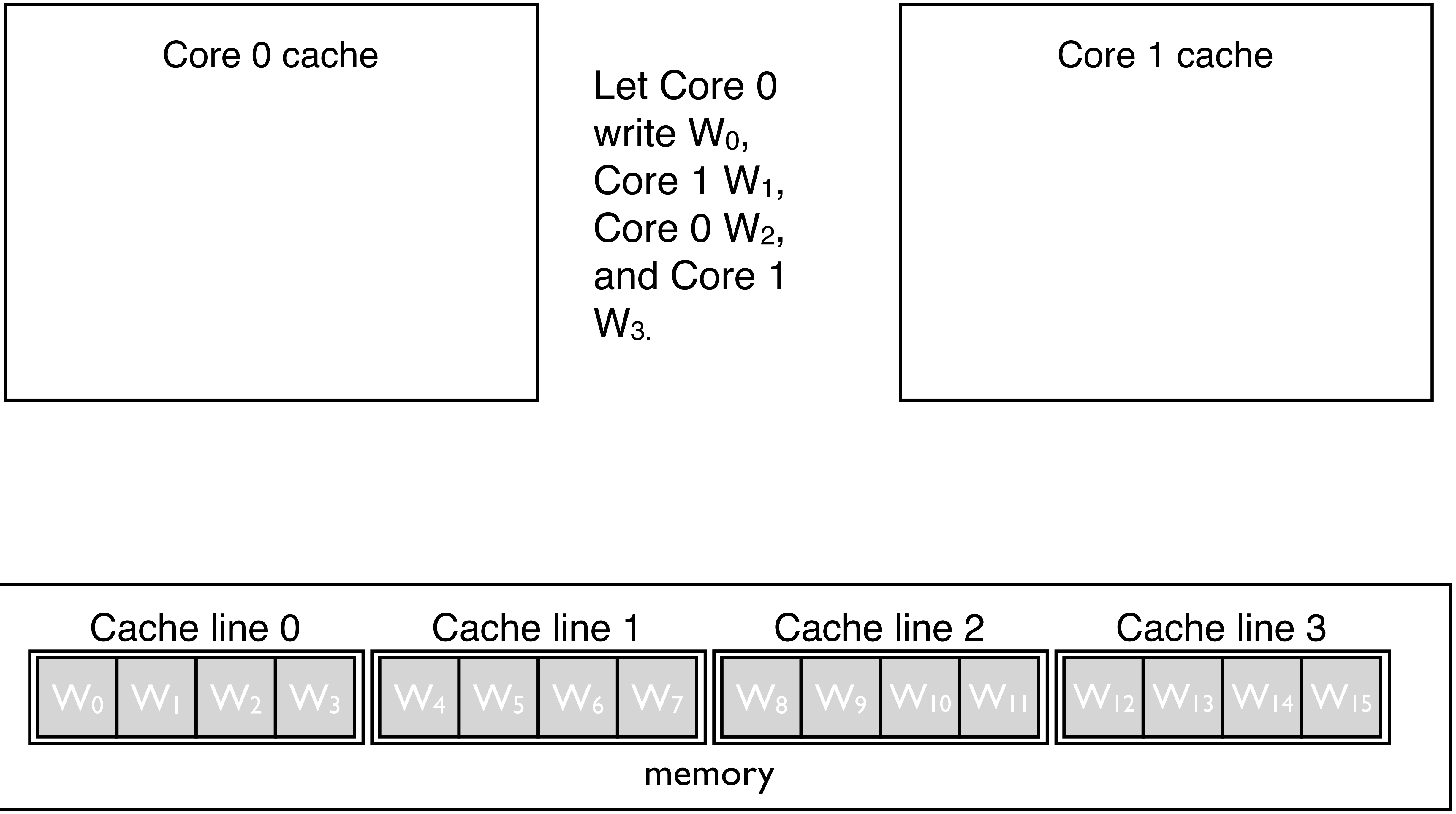
Only one cache can hold a line for *writing*



Core 0 accesses W9 to *write*

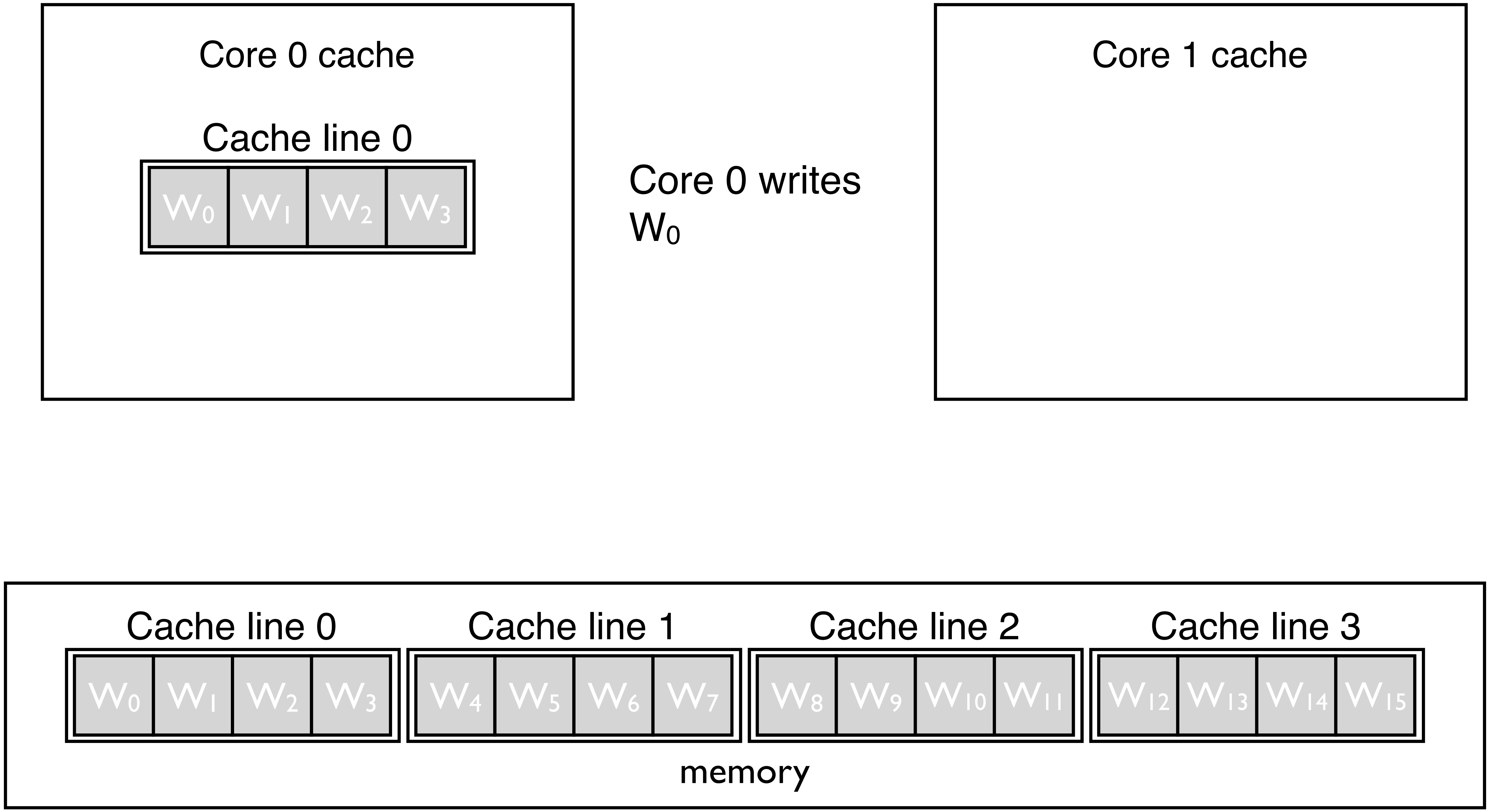
Core 1's line is invalidated

Only one cache can hold the same line for *writing*

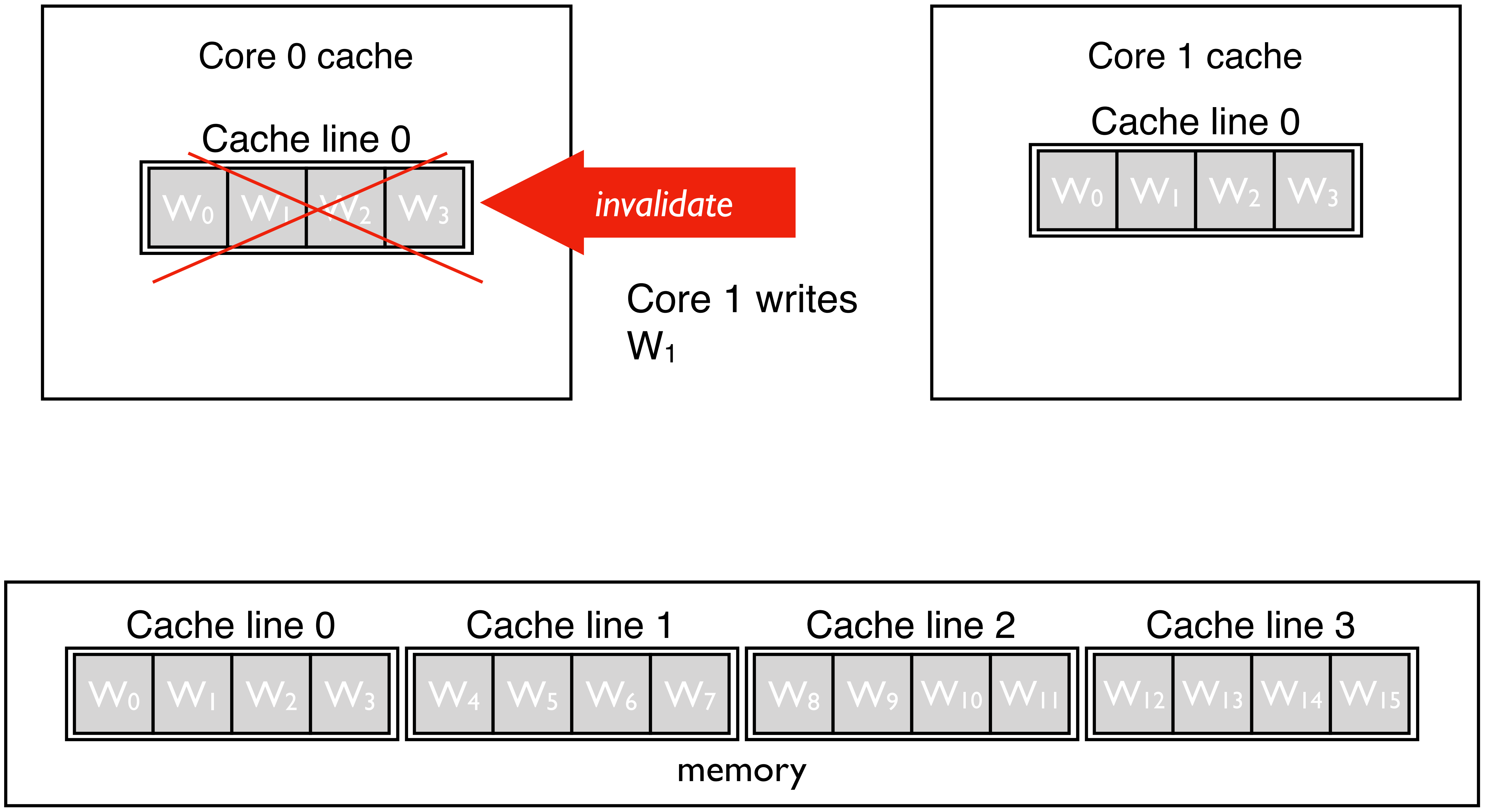


Let Core 0 write W_0, Core 1 W_1, Core 0 W_2, and Core 1 W_3.

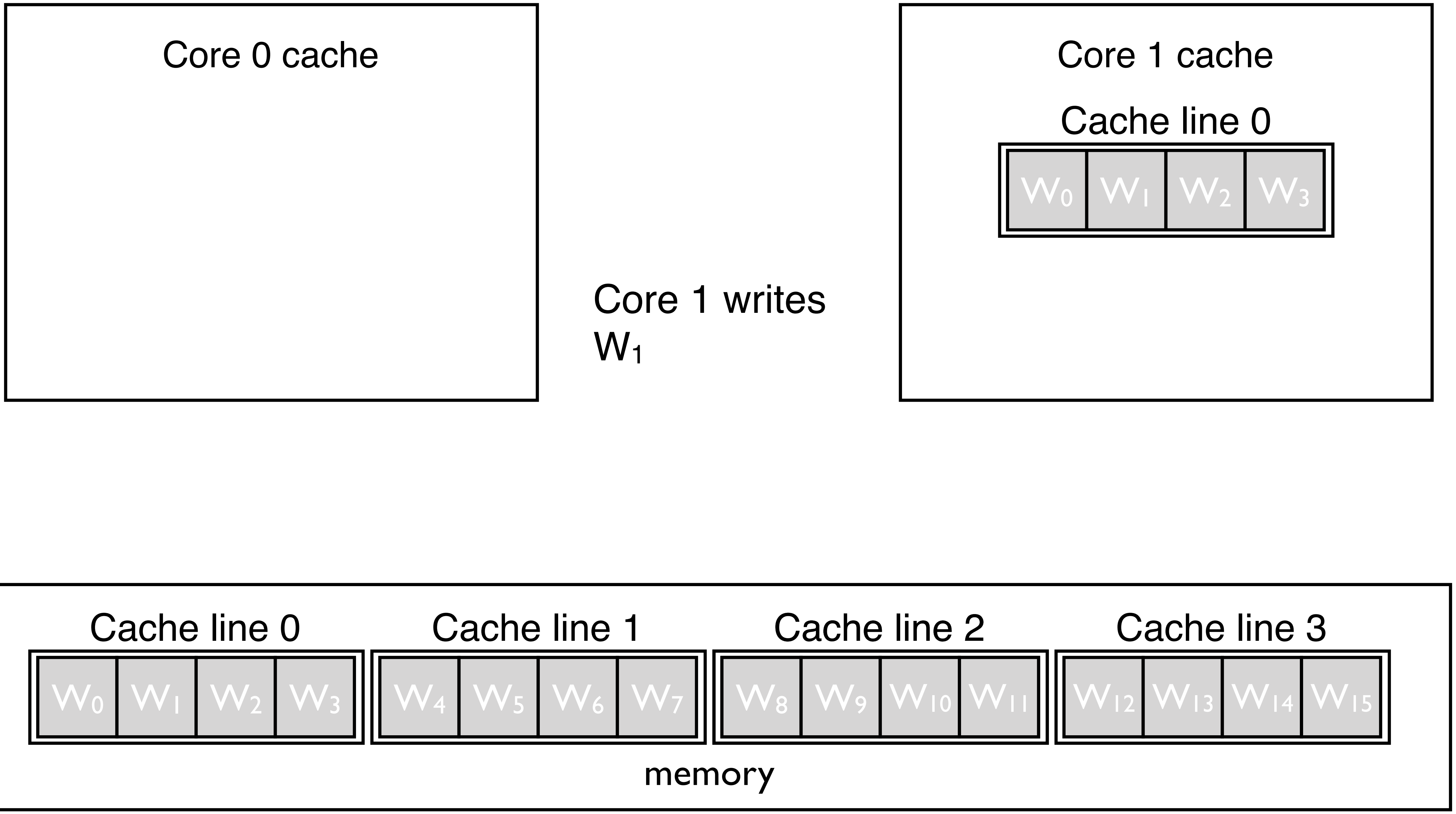
Only one cache can hold the same line for *writing*



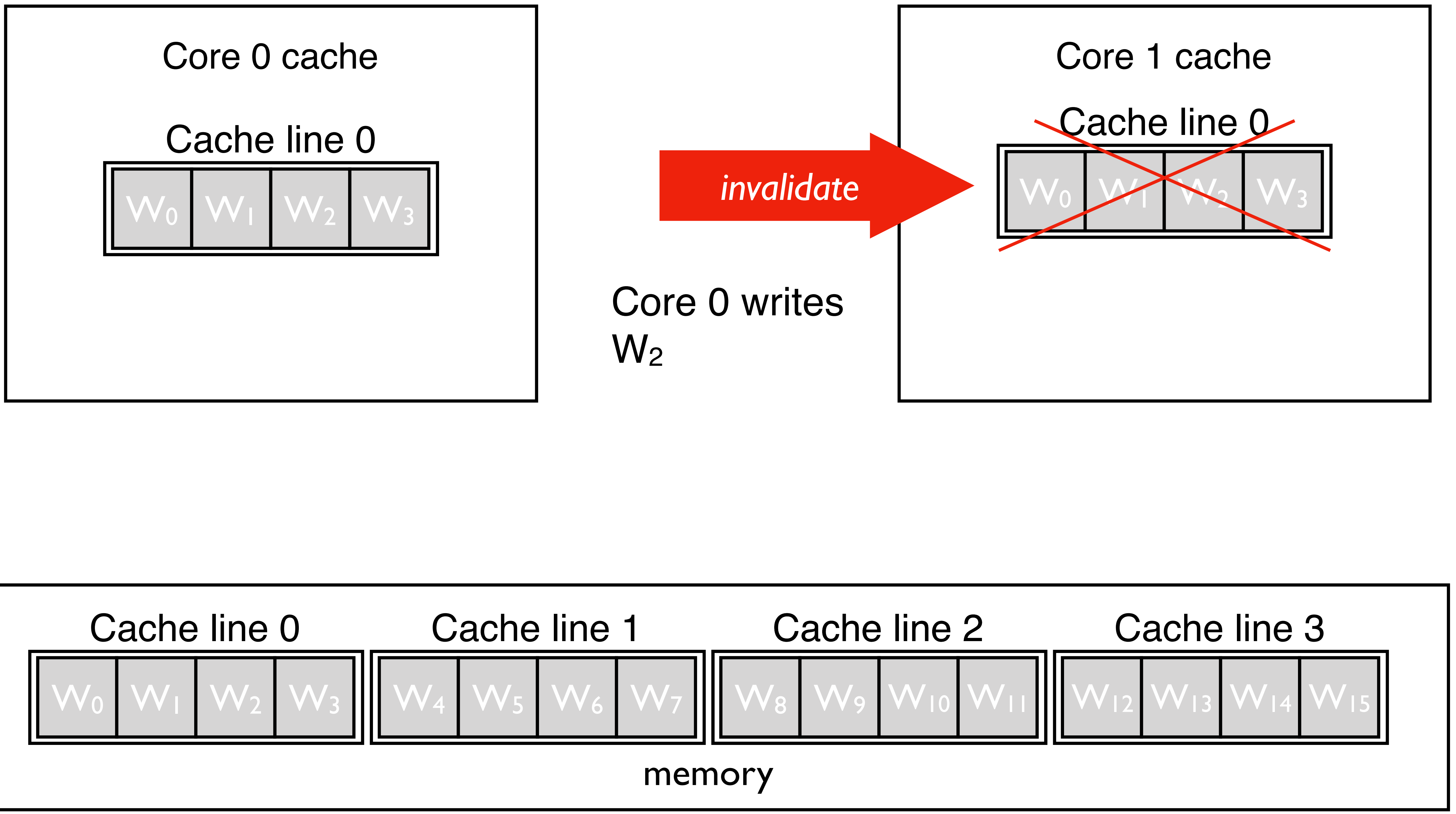
Only one cache can hold the same line for *writing*



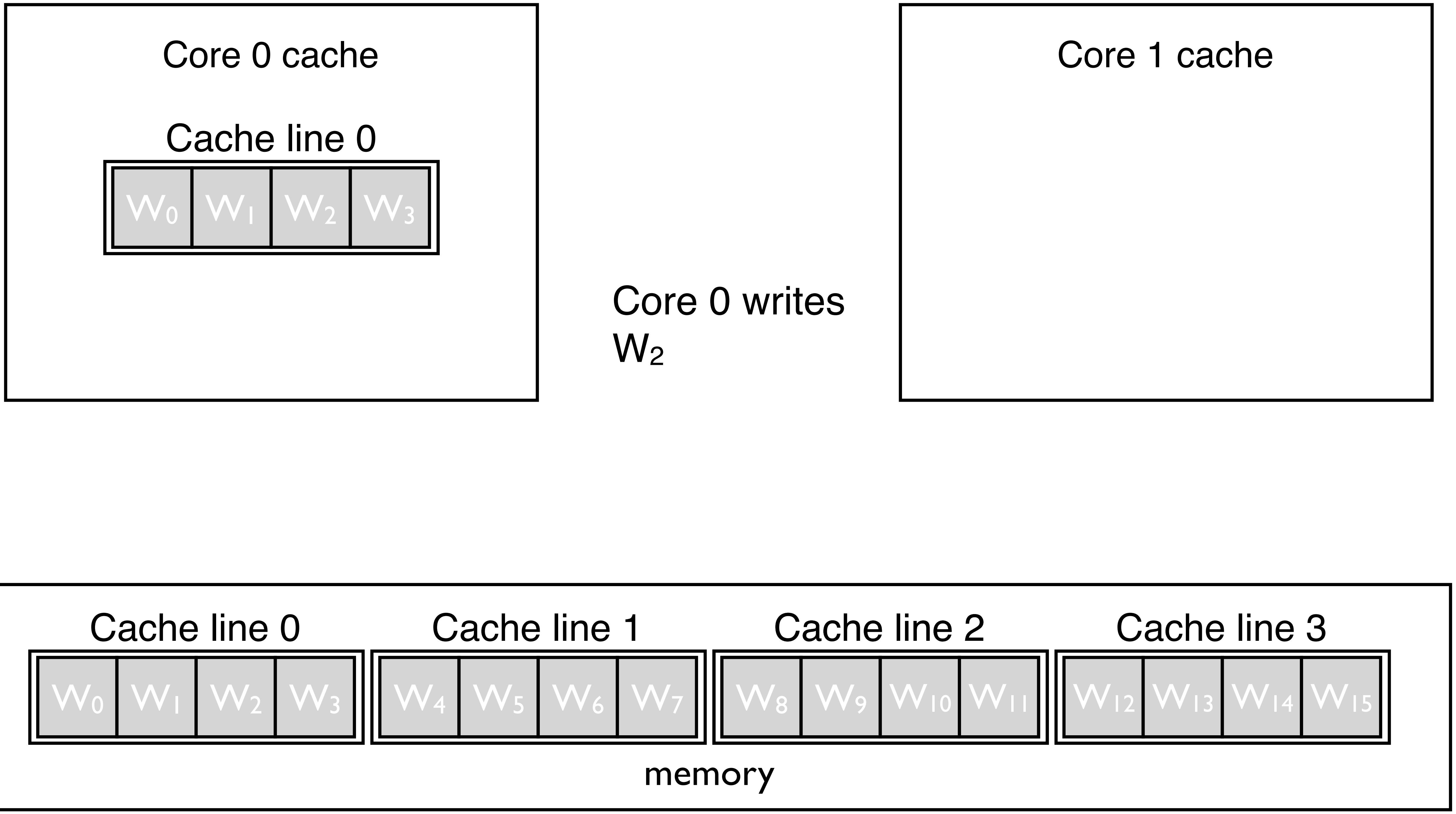
Only one cache can hold the same line for *writing*



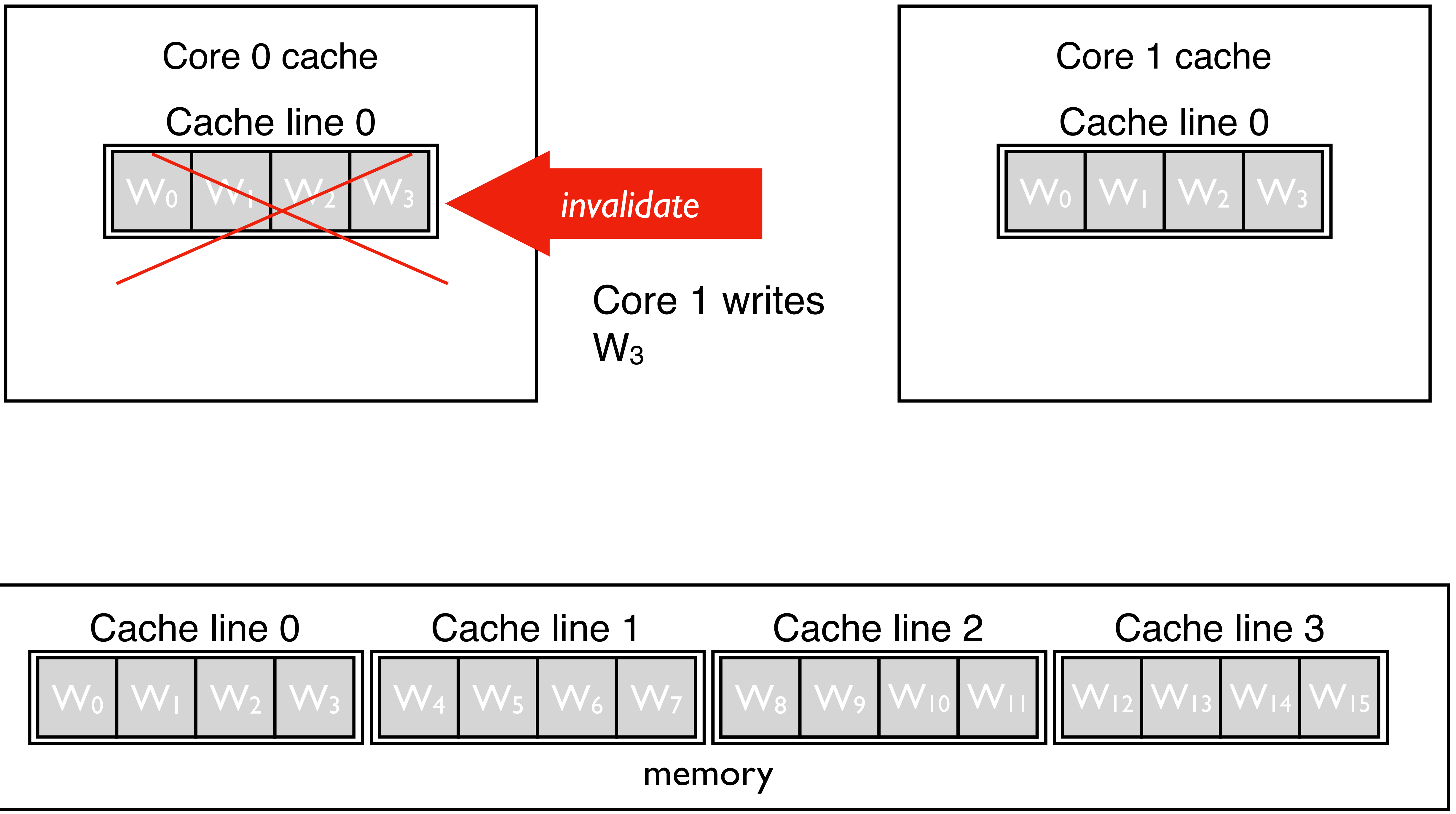
Only one cache can hold the same line for *writing*



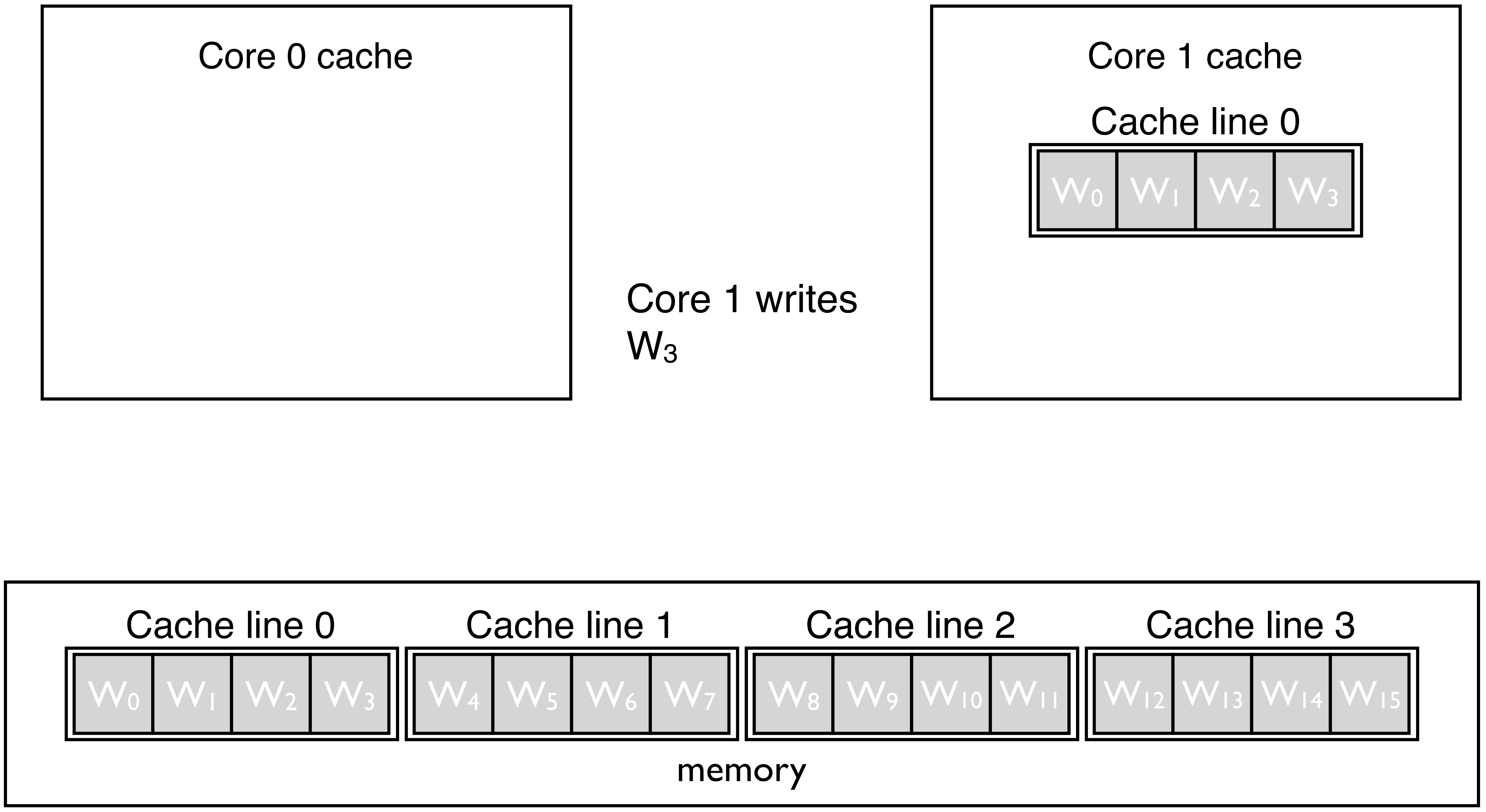
Many caches can hold the same line for *writing*



Only one cache can hold the same line for *writing*



Only one cache can hold the same line for *writing*



The problem with this

- All the invalidation and re-accessing the cache line takes time
- This makes program execution slower
- This can happen when, for example, the even iterations of a loop execute on one core and the odd iterations execute on another
- We'll talk more later about how to avoid this.

Other cache facts

- Most general purpose processors have 3 levels of cache (numbers for Intel Haswell, are different for different processors and models.
  - *Level 1* is the fastest, ~4 cycles to access, ~32K bytes. Level 1 cache is in a core
  - *Level 2* is the next fastest, ~12 cycles to access, ~256K bytes. Level 2 cache is in a core
  - *Level 3* is the next fastest, ~36 cycles to access, ~256K bytes. Level 3 is shared among cores

Other cache facts (2)

- When a cache is full, and another line *x* needs to be accessed
  - A line *y* in the cache is selected to be *evicted*
  - The line *y* is *evicted*
  - The line *x* is put where line *y* used to be

How to make caches work for us

- Accesses of code running on a core should access words close to the last word accessed. This reduces the number of lines accessed, and the number of lines that need to be held.
- Accessing the same line repeatedly before the cache is full also reduces the number of lines that need to be brought in
- More details on how to do this when we actually start talking about programming
- Using caches well can make programs run 10X faster!

A programming model must provide a way of specifying

- what parts of the program execute in parallel with one another
- how the work is distributed across different cores
- the order that reads and writes to memory will take place
- that a sequence of accesses to a variable will occur *atomically* or without interference from other threads.
- **And,** ideally, it will do this while giving *good performance* and allowing *maintainable programs* to be written.

OpenMP

- Open *M*ulti-*P*rocessor
  - targets multicores and multi-processor shared memory machines
  - An open standard, not controlled by any manufacturer
- Allows loop-by-loop & region-by-region parallelization of sequential programs.

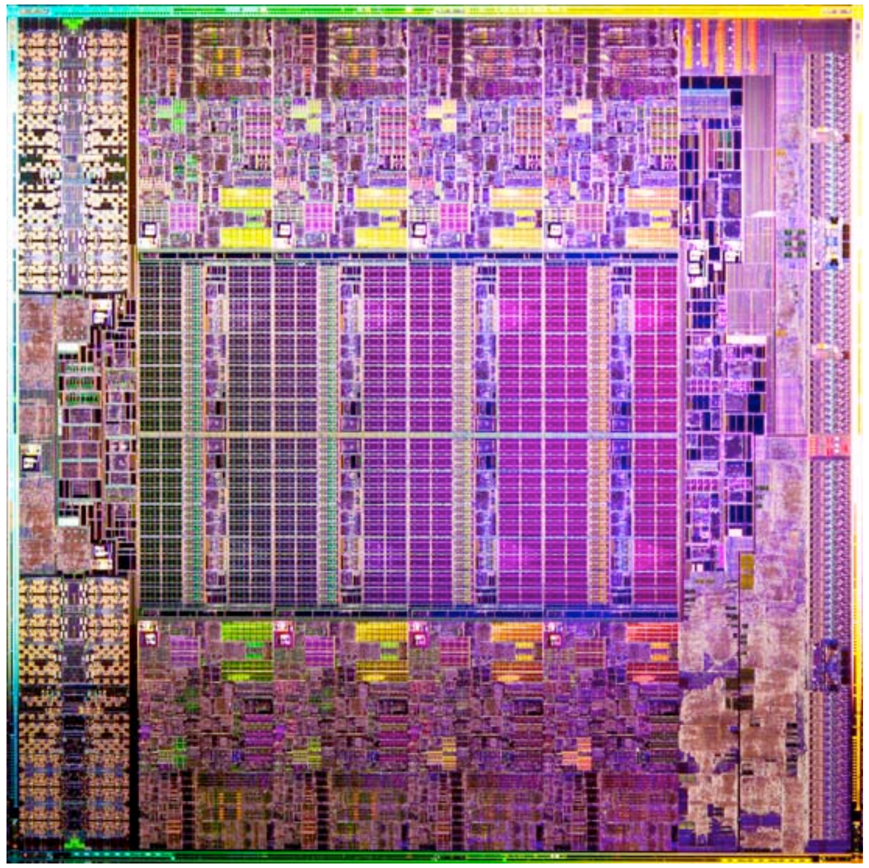
What executes in parallel?

```
c = 57.0;
for (i=0; i < n; i++) {
  a[i] = c[i] + a[i]*b[i]
}
```

```
c = 57.0
#pragma omp parallel for
for (i=0; i < n; i++) {
  a[i] = c[i] + a[i]*b[i]
}
```

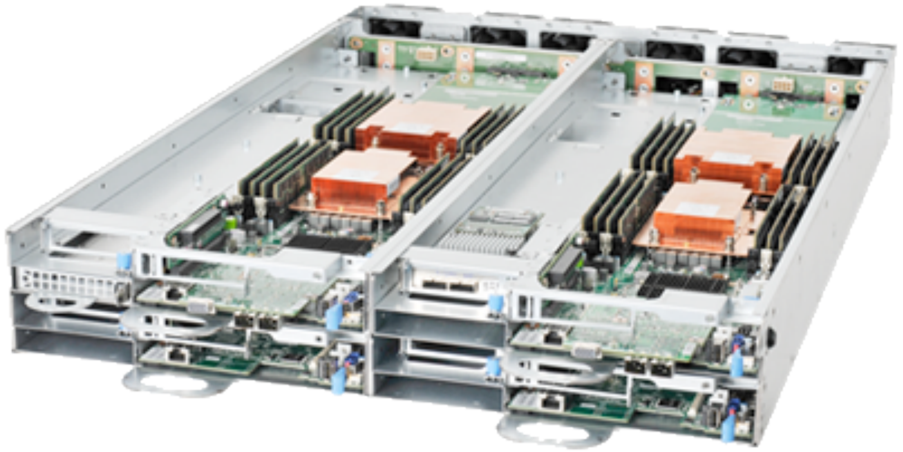
- *pragma* appears like a comment to a non-OpenMP compiler
- pragma requests parallel code to be produced for the following for loop

processors, nodes, processes and threads



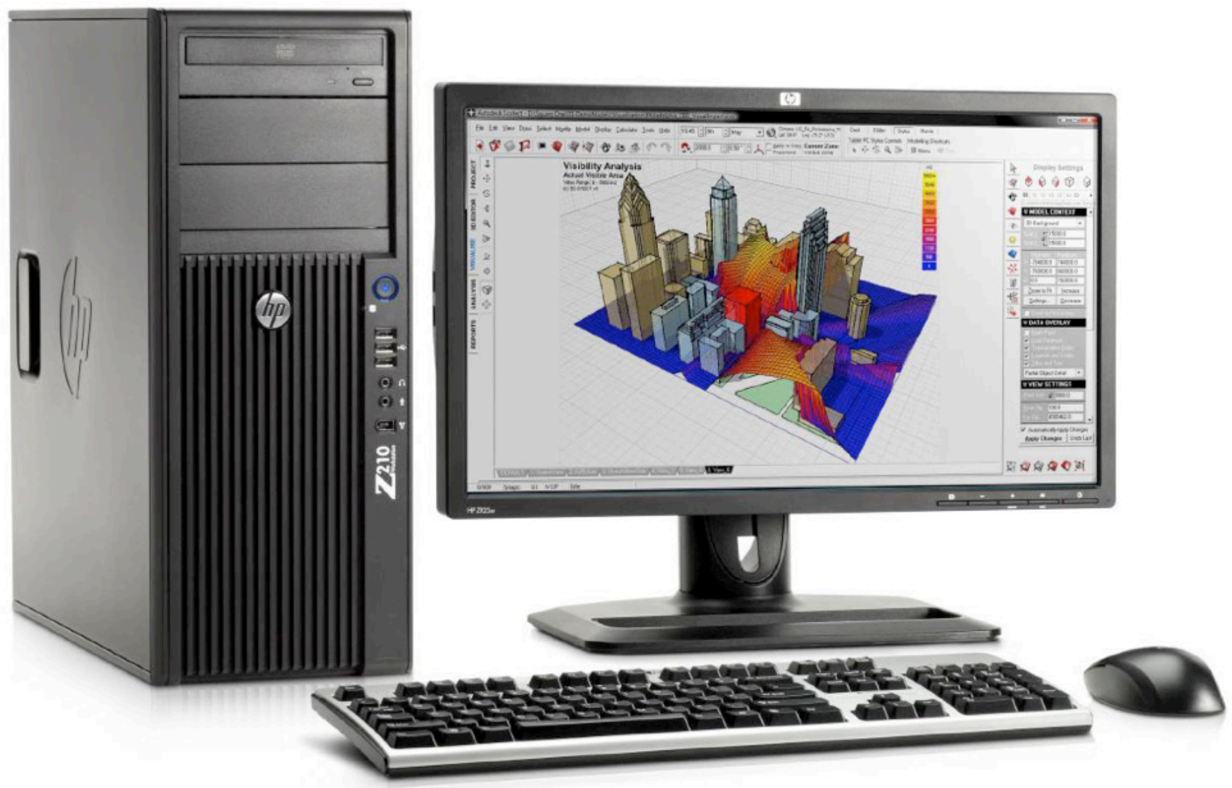
A processor is a physical piece of hardware with one or more cores that executes instructions

processors, **nodes**, processes and threads



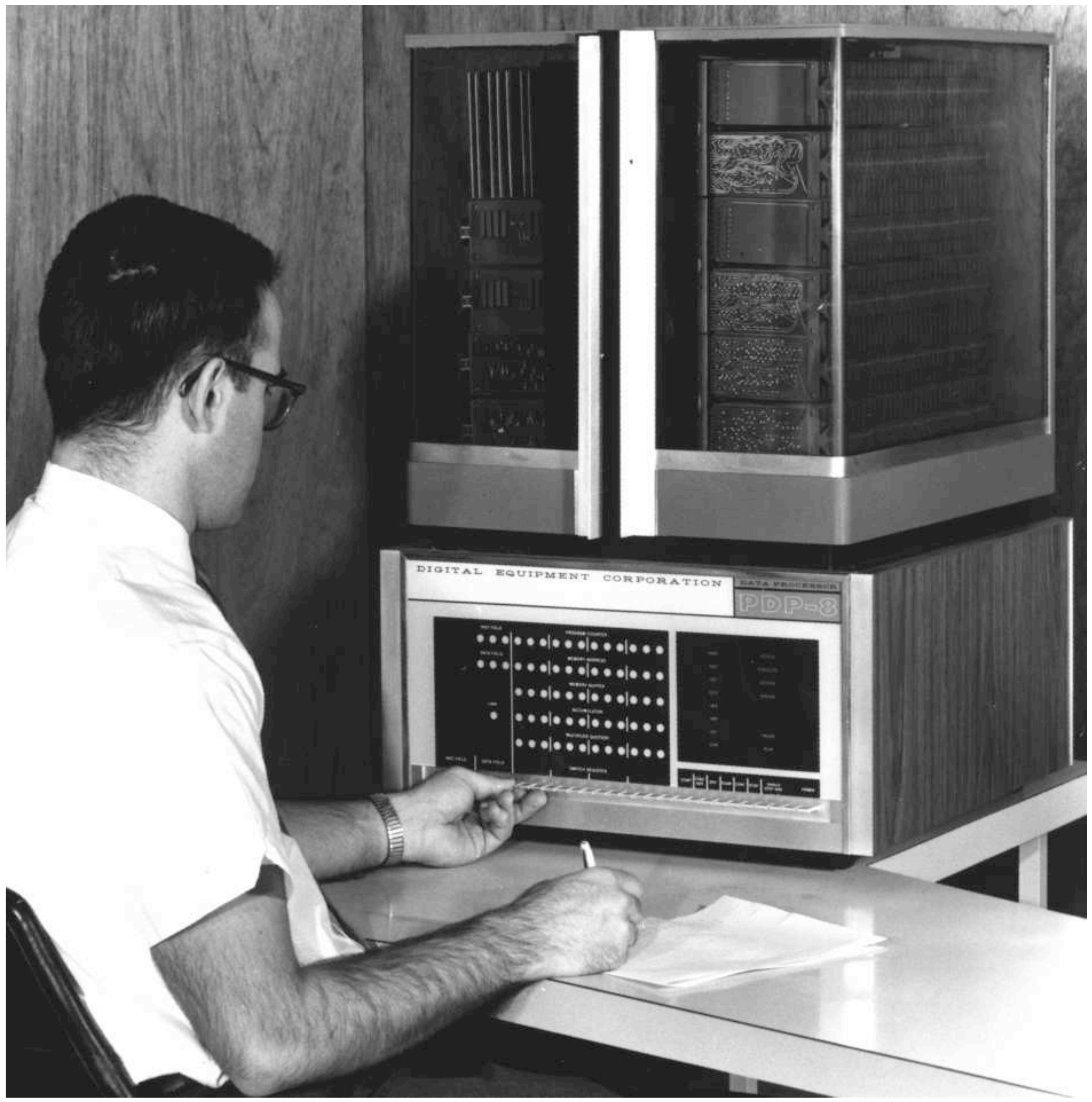
A node is one or more processors along with associated devices (disk drive, memory, i/o cards, communication cards, etc.

One or more nodes form a system



processors, nodes, processes and threads

- In the early days of computing and on specialized machines, one “program” runs on the machine at a time
- It has access to the raw hardware and communicates with the hardware directly
- This is not very useful -- only one person or job can use the machine at a time



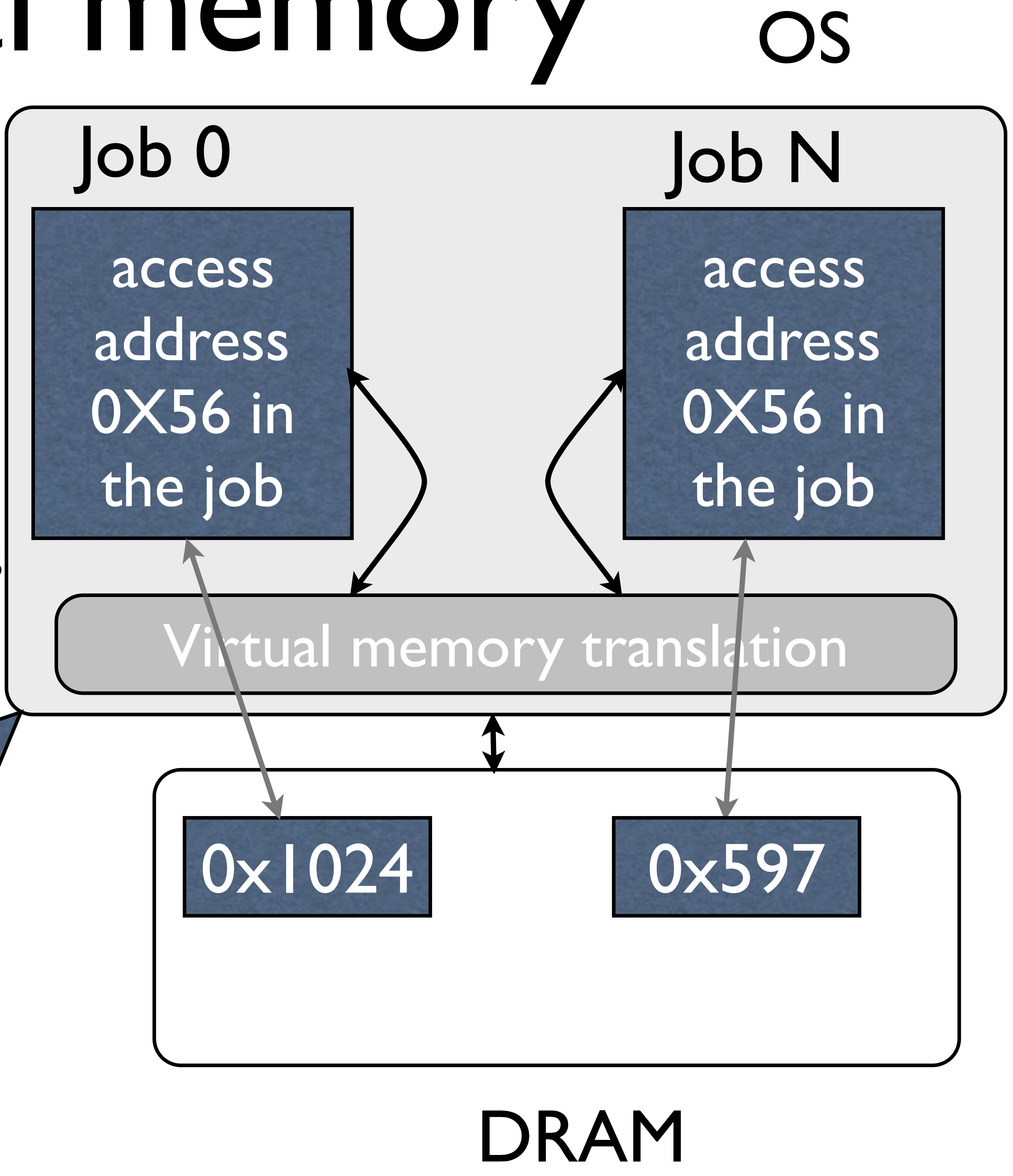
A Digital Equipment Corporation
PDP-8 (programmable data processor)
Early low-cost mass produced computer

processors, nodes, **processes** and threads

- An *operating system* allows multiple jobs and/or users to access the machine at the same time
    - The OS virtualizes the machine -- each job sees the machine as entirely its own
    - The OS protects each job from other jobs
- Virtual memory allows each job to act as if it has access to the entire address space of memory. This is done by having the OS, with help from the hardware, map program addresses into small parts of real DRAM addresses. Physical DRAM serves as a cache and disk as the backing store.

Virtual memory

- An *operating system* allows multiple jobs and/or users to access the machine at the same time
  - The OS virtualizes the machine -- each job sees the machine as entirely its own


OS
Job 0
Job N
access address 0X56 in the job
access address 0X56 in the job
Virtual memory translation
0x1024
0x597
DRAM

processors, nodes, processes and threads

- The keyboard, printers, disk drives, intra-system network, inter-system network (the internet), etc.
- The name for a single job that has a single virtualized image of the system is a *process*
    - Browser, email program, program you have written, Word, VI, emacs, are all processes, and all can be active and sharing the system
- Via time-sharing/multiplexing of processes, all can appear to us to be running simultaneously, even with a single core

processors, nodes, processes and threads

- For our purposes, the most important aspect of a process is that its address space is separate from other processes address spaces
- Cannot communicate directly with other processes
    - this is not entirely accurate as unices and other OSes support shared memory segments among processes
    - Not commonly used by programmers for parallel programming, more commonly used by systems programs
- Communication among processes requires sending *messages* via OS (often *sockets* are used)
- MPI (Message passing interface) is a common way to send messages

processors, nodes, processes and threads

- But sometimes we want multiple “things” running at the same time to be able to communicate and share memory locations, e.g., values of variables
- *Threads* allow this to happen
- Threads are usually managed by the OS, but a given thread is *owned* by a process
- All threads owned by the process share the virtualized resources given to the process by the OS. In particular, all threads owned by a process share the same address space.
- This allows threads to communicate via memory, which is usually faster than communicating via messages
- Threads run on a core
- Every process has a *main* thread that runs the processes’ code

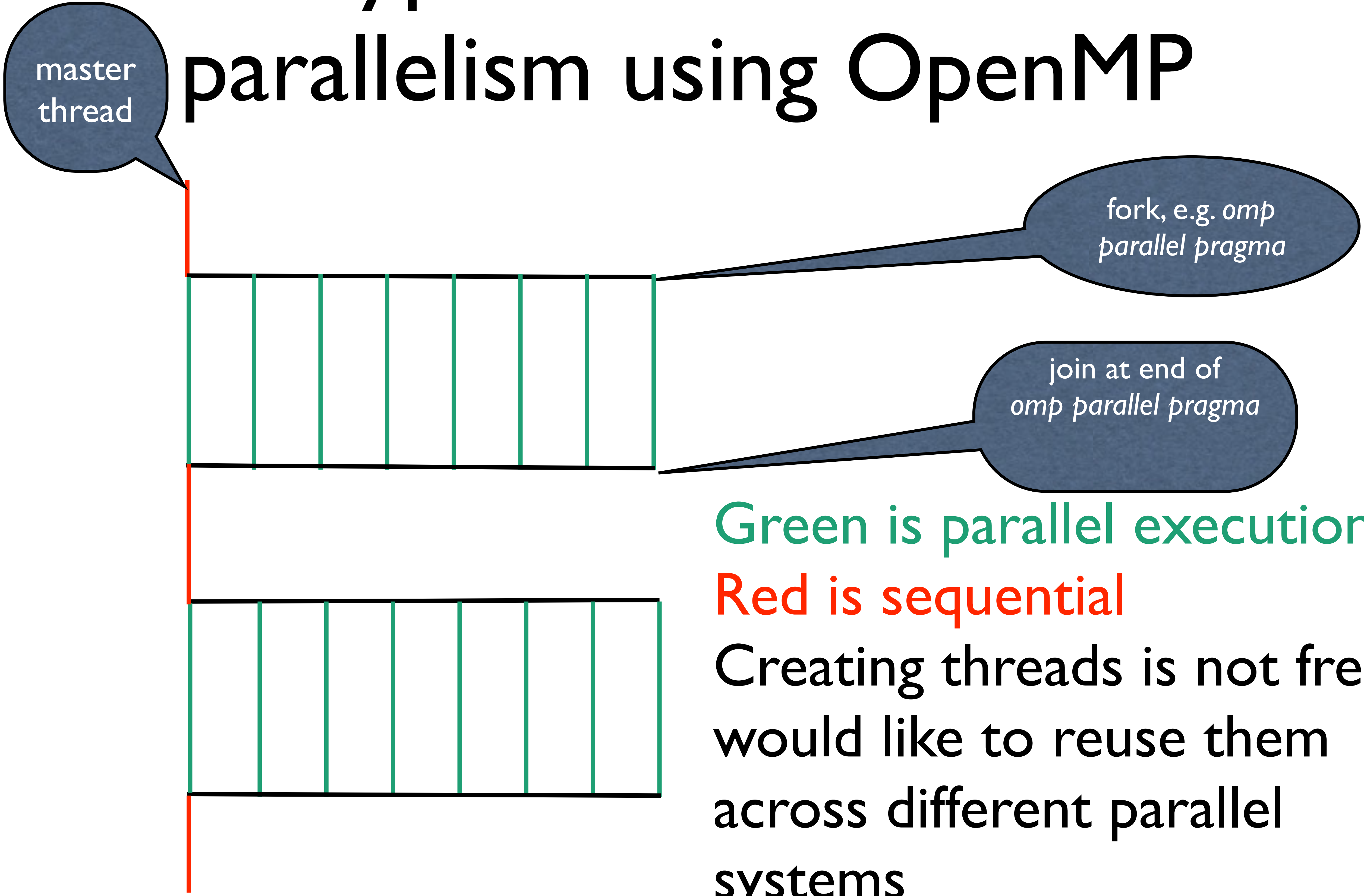
Threads and processes -- summary

- Threads and processes are typically operating system entities and concepts
- A *process* has its own address space and owns a typically *virtualized* copy of the machine when executing
  - processes may own one or more threads
- A *thread* shares its address space with it’s owning process and all other threads owned by the same process
  - each thread has its own copy of registers
  - local variables can be created that are accessible only by the thread
  - threads are the fundamental building block of parallel shared memory programs

Two main levels of parallelism

- Thread level
  - Parallelism is across threads
  - Typically within a node
  - We will look at systems later in the class that support thread level parallelism across nodes
  - We will use *OpenMP* and *Pthreads* to exploit thread level parallelism
- Process level parallelism
  - Parallelism is across processes
  - Typically across nodes
  - We will use *MPI (Message Passing Interface)* to exploit thread level parallelism

Typical thread level parallelism using OpenMP


master thread
fork, e.g. *omp parallel pragma*
join at end of *omp parallel pragma*

Green is parallel execution

Red is sequential

Creating threads is not free -- would like to reuse them across different parallel systems

How is the work distributed across different cores?

```
c = 57.0
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

- Split the loop into chunks of contiguous iterations with approximately *t/n* iterations per chunk
- Thus, if 4 threads and 100 iterations, thread one would get iterations 0:24, thread 2 25:49, and so forth
- Other scheduling strategies supported and will be discussed later.

The order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
#pragma omp parallel for schedule(static)
for (j=0; j < n; j++) {
   a[j] = c[j] + a[j]*b[j]
}
```

barrier

- Within an iteration, access to data appears in-order
- Across iterations, no order is implied. *Races* lead to undefined programs
- Across loops, an implicit *barrier* prevents a loop from starting execution until all iterations and writes (stores) to memory in the previous loop are finished
- The barrier is associated with the green *i* loop
- Parallel constructs execute after preceding sequential constructs finish

Relaxing the order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static) nowait
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
#pragma omp parallel for schedule(static)
for (j=0; j < n; j ++) {
   a[j] = c[j] + a[j]*b[j]
}
```

The *nowait* clause allows a thread that finishes its part of the green *i* loop to begin executing its iterations of the blue *j* loop without waiting for other threads to finish their iterations of the green *i* loop.

Static says how the iterations of the loop are split among different threads.

Accessing variables without interference from other threads

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   a = a + b[i]
}
```

Dangerous -- all iterations are updating *a* at the same time -- a *race* (or *data race*).

```
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   a = a + b[i];
}
```

Not particularly useful, but correct -- *critical* pragma allows only one thread to execute the next statement at a time.   Can be *very inefficient!*

We will learn better ways to do this.

Next -- OpenMP in more detail