ECE 563
Programming Parallel Machines

- The syllabus: https://engineering.purdue.edu/~smidkiff/ece563/files/syllabus.pdf

READY Purdue

READY Purdue
PREPARE, PLAN, STAY INFORMED
www.purdue.edu/emergency_preparedness/
2011-2012

In case of emergency call 911

- Emergency / Non-Emergency Resources
- Purdue Alert - Emergency Warning Notification System
- Fire / Evacuation Procedures
- Evacuation - Persons With Disabilities
- Medical Emergency
- Mental Health Emergency
- Criminal Activity
- Shelter-In-Place
- Active Shooter/Active Threat
- Severe Weather / Tornado Warning
- Hazardous Materials - Spills, Gas Leaks, Odors
- Earthquake
- Utility Failure / Elevator Malfunction / Flooding
- Bomb Threat / Suspicious Package

Download this application on your computer (336kb .air), you must also have Adobe Air to run it.

http://www.purdue.edu/emergency_preparedness/flipchart/

counseling available at http://www.purdue.edu/caps/

Tornados

From the AAE website

Armstrong Procedure - The **Shelter-in-Place** for Armstrong Hall is the **Lower Level** of the building. When the All-Hazards Emergency Warning Sirens sound **or** if notified by the **Purdue ALERT** system, calmly proceed to the basement level of Armstrong Hall. **Lecture halls B061, B071, and 1010 are preferred for occupants to gather**, in order to communicate among the group. Please be prepared to kneel facing the wall, covering your head to protect yourself from any potential debris. Stay in these sheltered places until local emergency officials have given the all clear announcement.

Note: **The All-Hazards siren cannot be heard from most interior rooms in Armstrong Hall.** AAE safety monitors will notify AAE areas in Armstrong if the All-Hazards siren sounds or if a tornado warning is issued. Safety monitors or other officials will also give an all clear when the emergency has ended.

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

What can run in parallel?

Consider the loop:

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

Let each iteration execute in parallel with all other iterations on its own processor

time

i = 1
a[1] = b[1] + c[1]
c[1] = a[0]

i = 2
a[2] = b[2] + c[2]
c[1] = a[0]

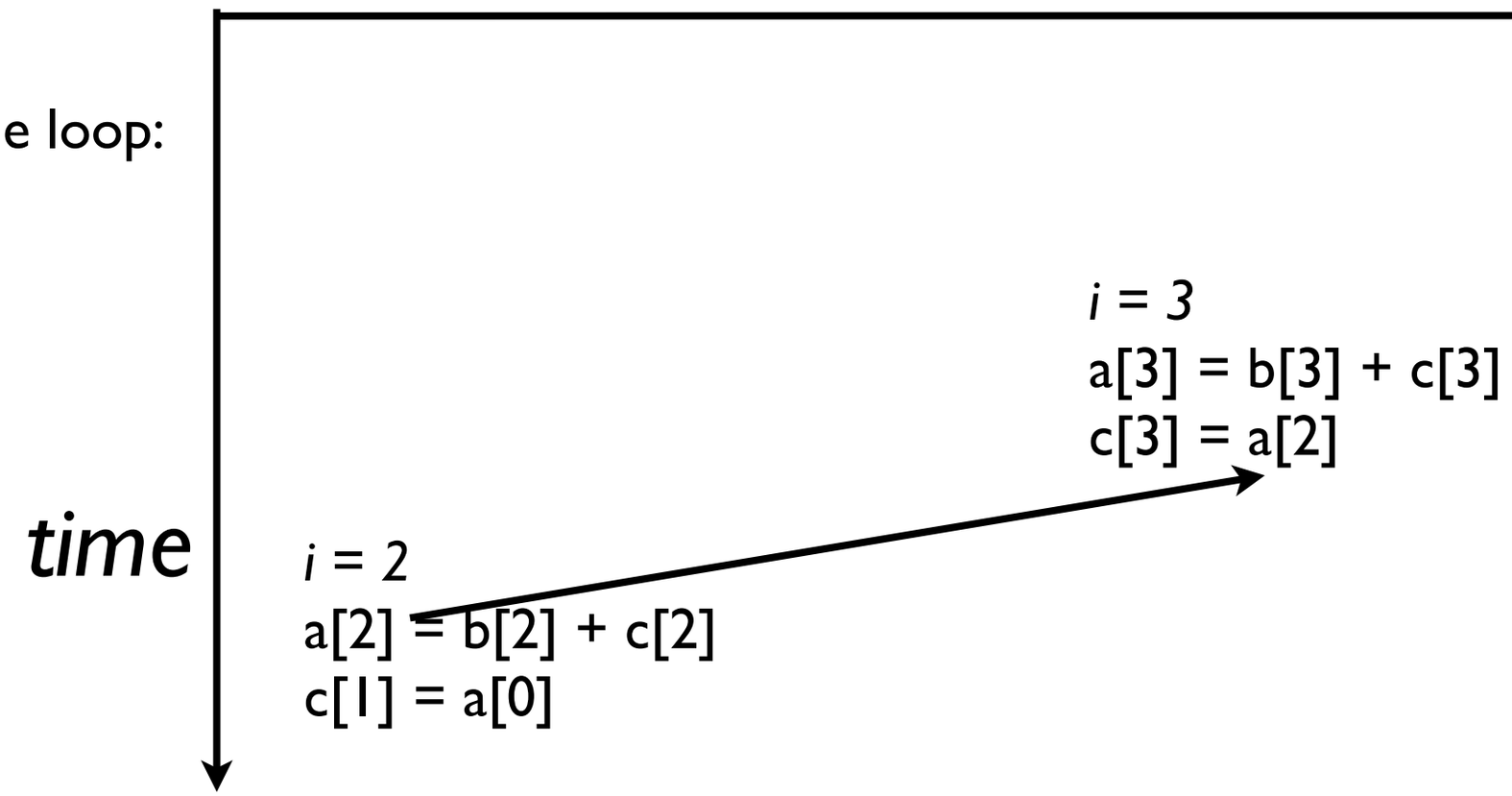
i = 3
a[3] = b[3] + c[3]
c[3] = a[2]

Note that data is produced in one iteration and consumed in another.

What can run in parallel?

Consider the loop:

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



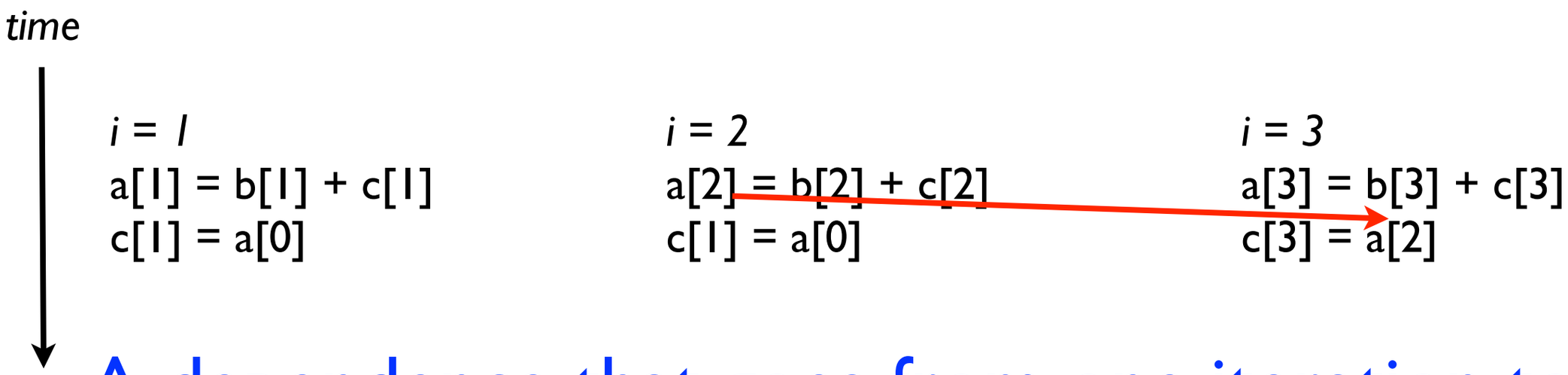
What if the processor executing iteration i=2 is delayed for some reason? *Disaster* - the value of a[2] to be read by iteration i=3 is not ready when the read occurs!

Cross-iteration dependences

Consider the loop:

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

Orderings that must be enforced to ensure the correct order of reads and writes are called *dependences.*



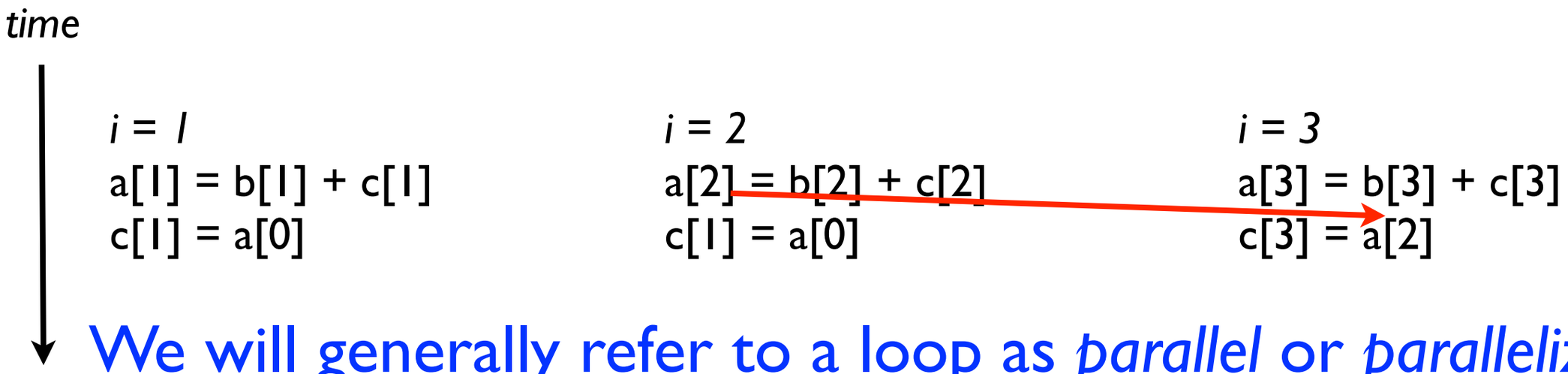
A dependence that goes from one iteration to another is a *cross iteration,* or *loop carried* dependence

Cross-iteration dependences

Consider the loop:

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

Loops with cross iteration dependences cannot be executed in parallel unless mechanisms are in place to ensure dependences are honored.



We will generally refer to a loop as *parallel* or *parallelizable* if dependences do not span the code that is to be run in parallel.

Where is parallelism found?

- Most work in most programs, especially numerical programs, is in a loop
- Thus effective parallelization generally requires parallelizing loops
- *Amdahls law* (discussed in detail later in the semester) says that, e.g., if we parallelize 90% of a program we will only get a *speedup* of 10X, 99% a speedup of 100X. To effectively utilize 1000s of processors, we need to parallelize 99.9% or more of a program!

A short architectural overview

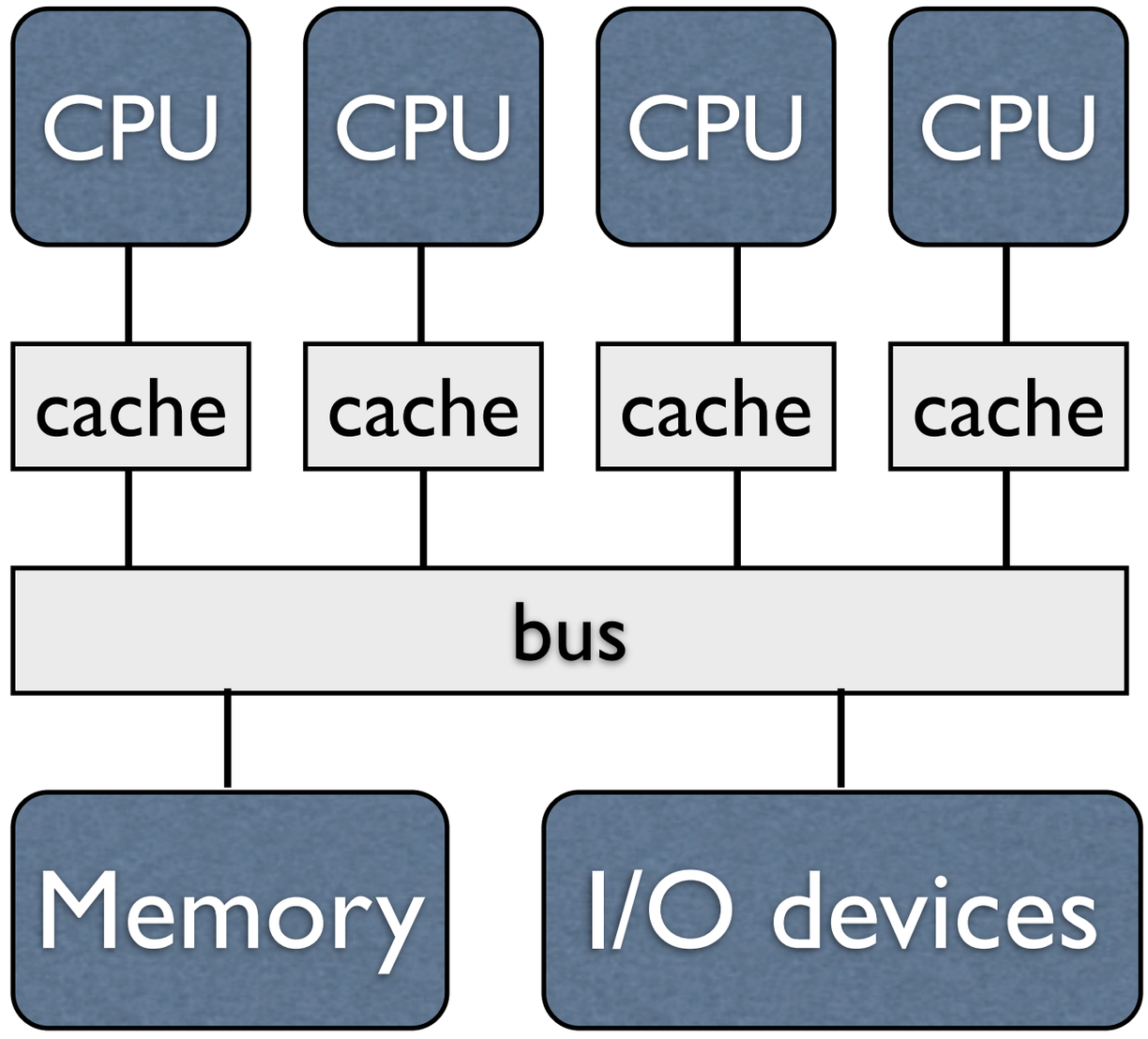
Warning: gross simplifications to follow

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.

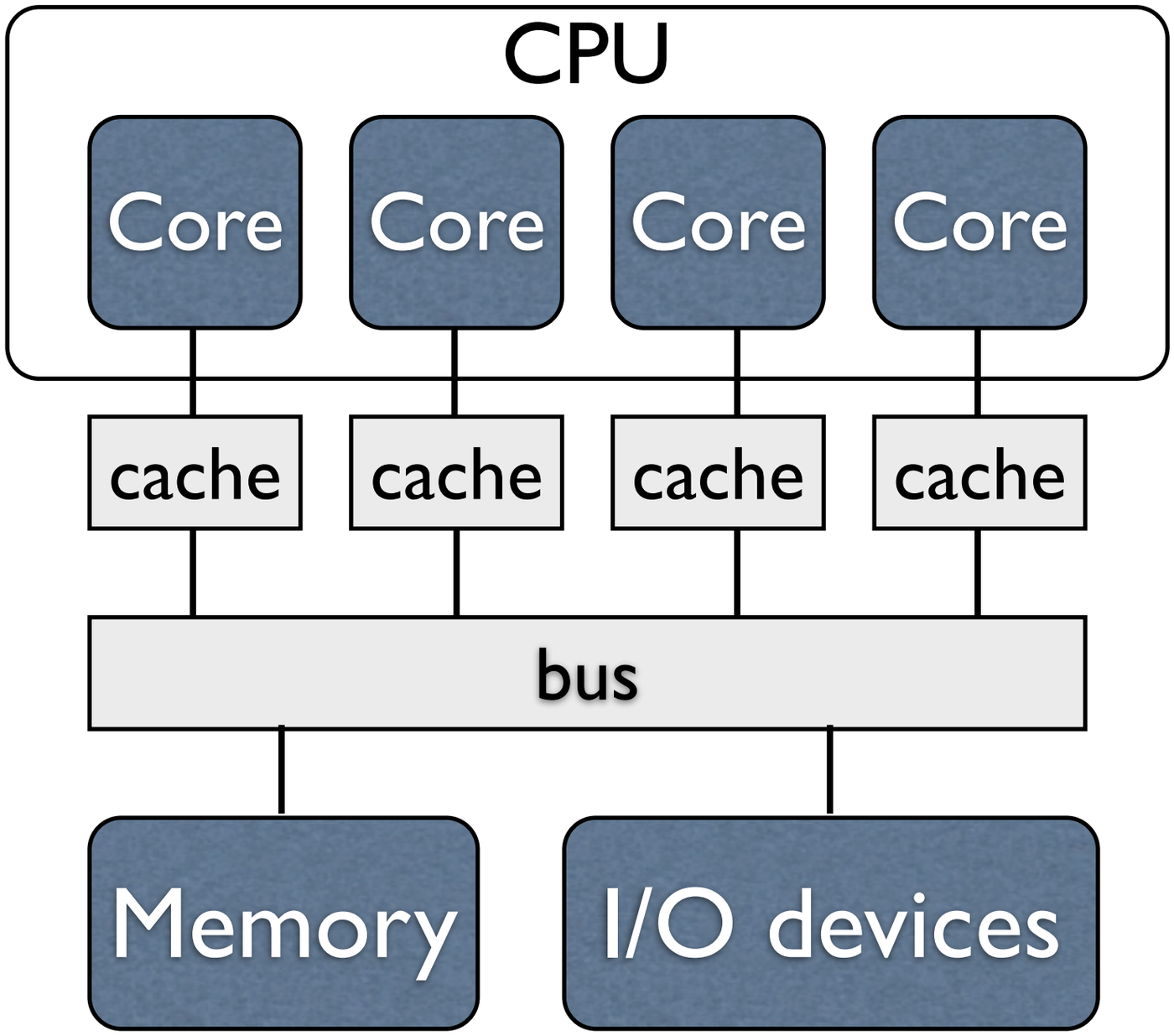
A Uniform Memory Access shared memory machine

All processors access *global memory* at the same speed



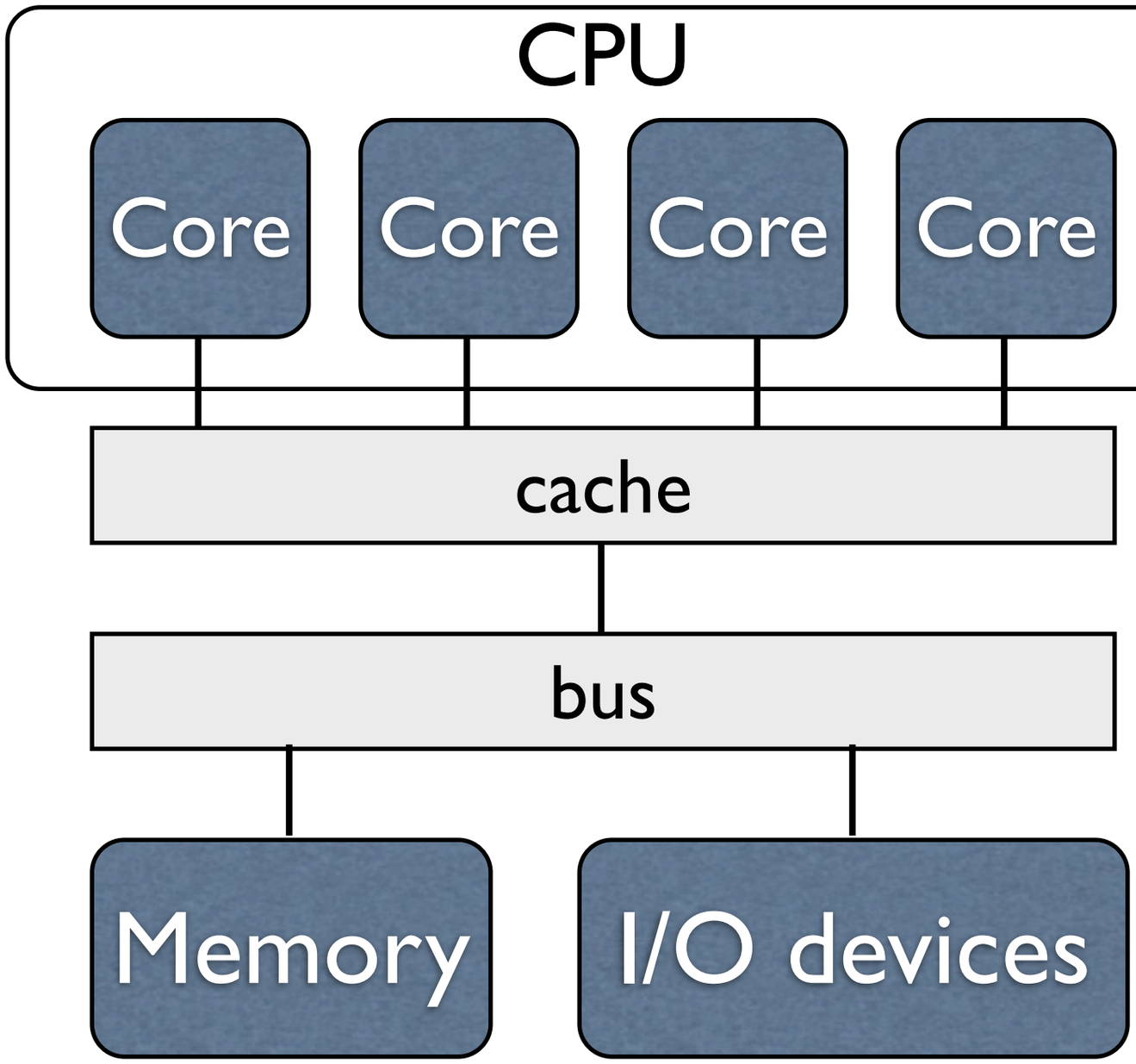
Multicore machines are usually have uniform memory access

All cores access global memory at the same speed

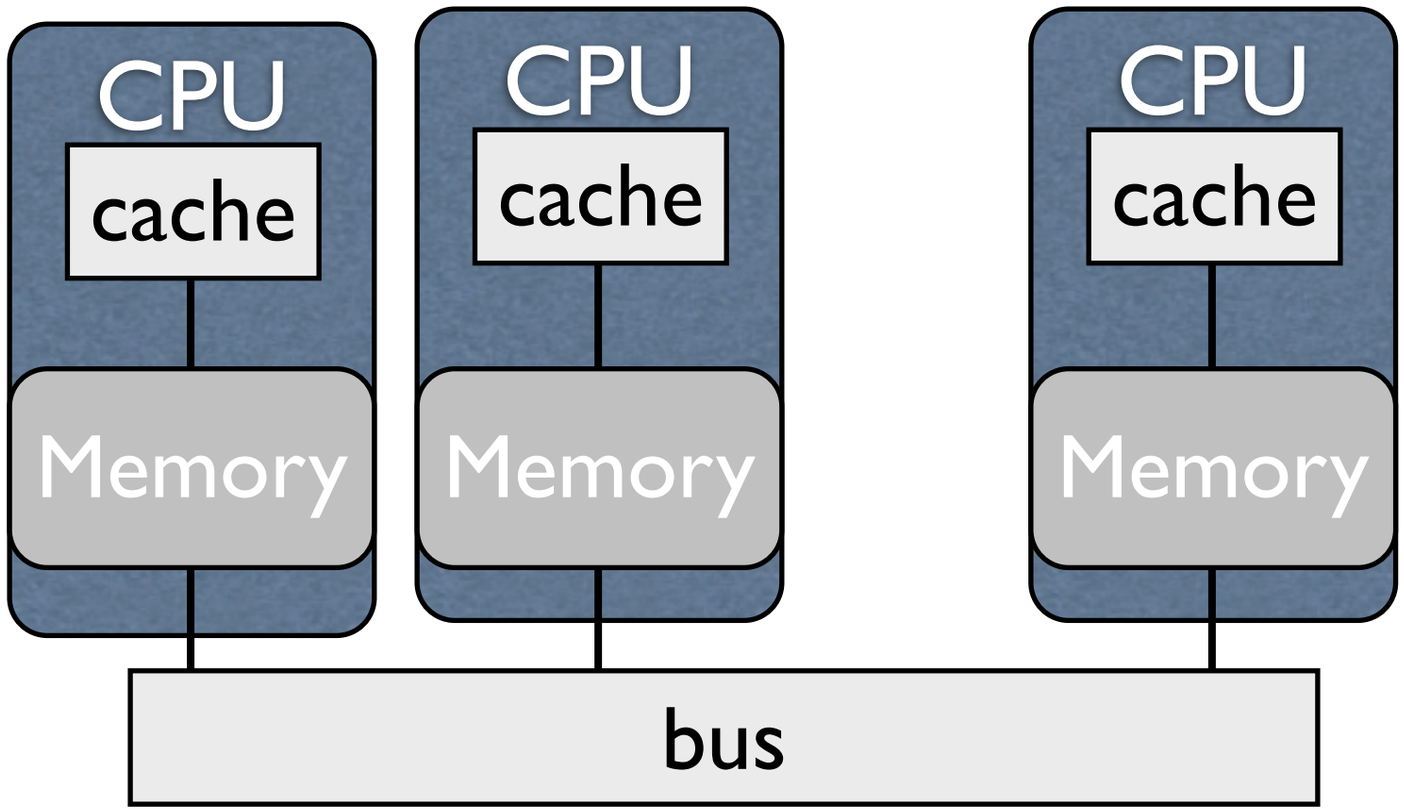


Multicore machines usually share at least one *level* of cache

All cores access global memory at the same speed



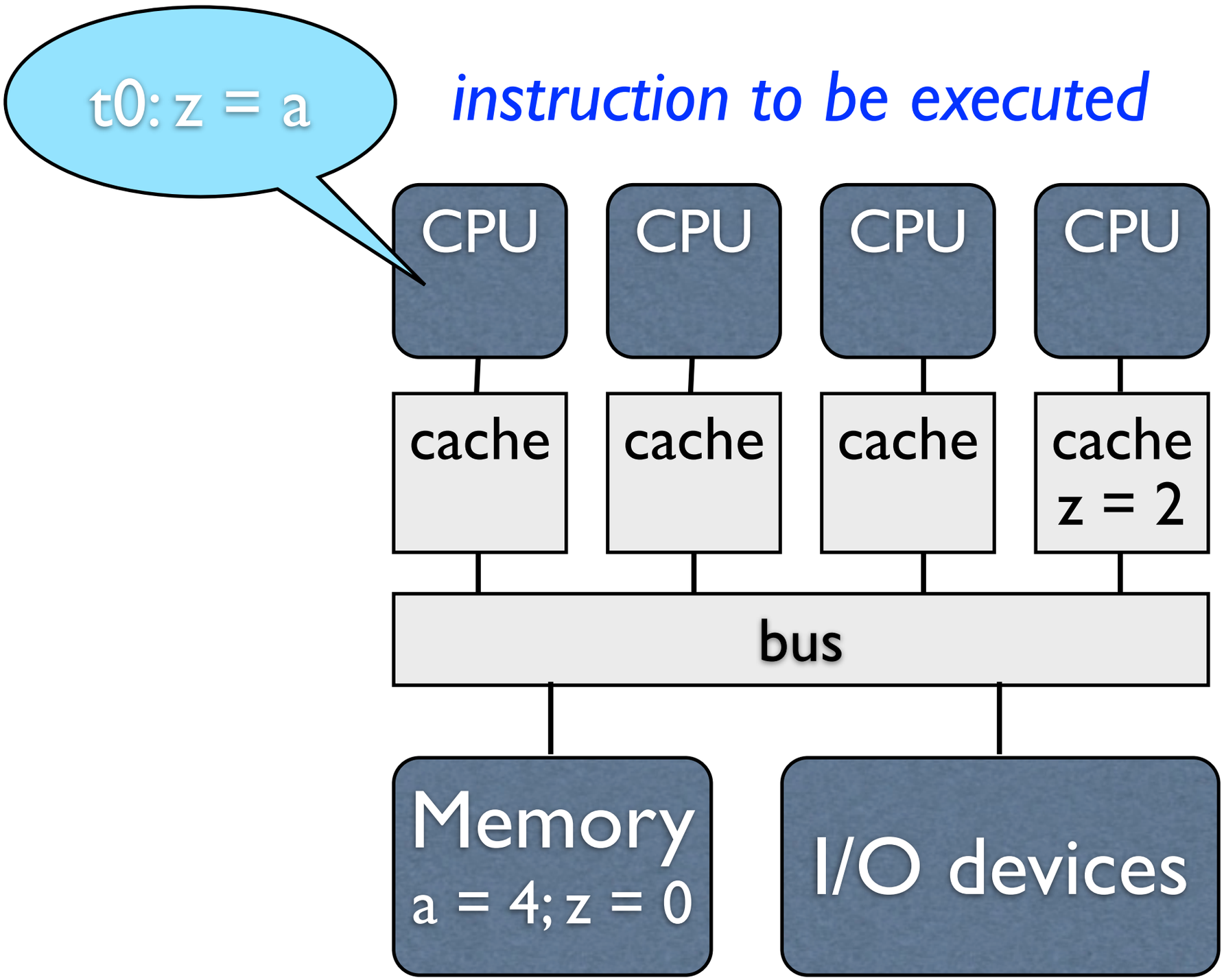
A NUMA shared memory machine

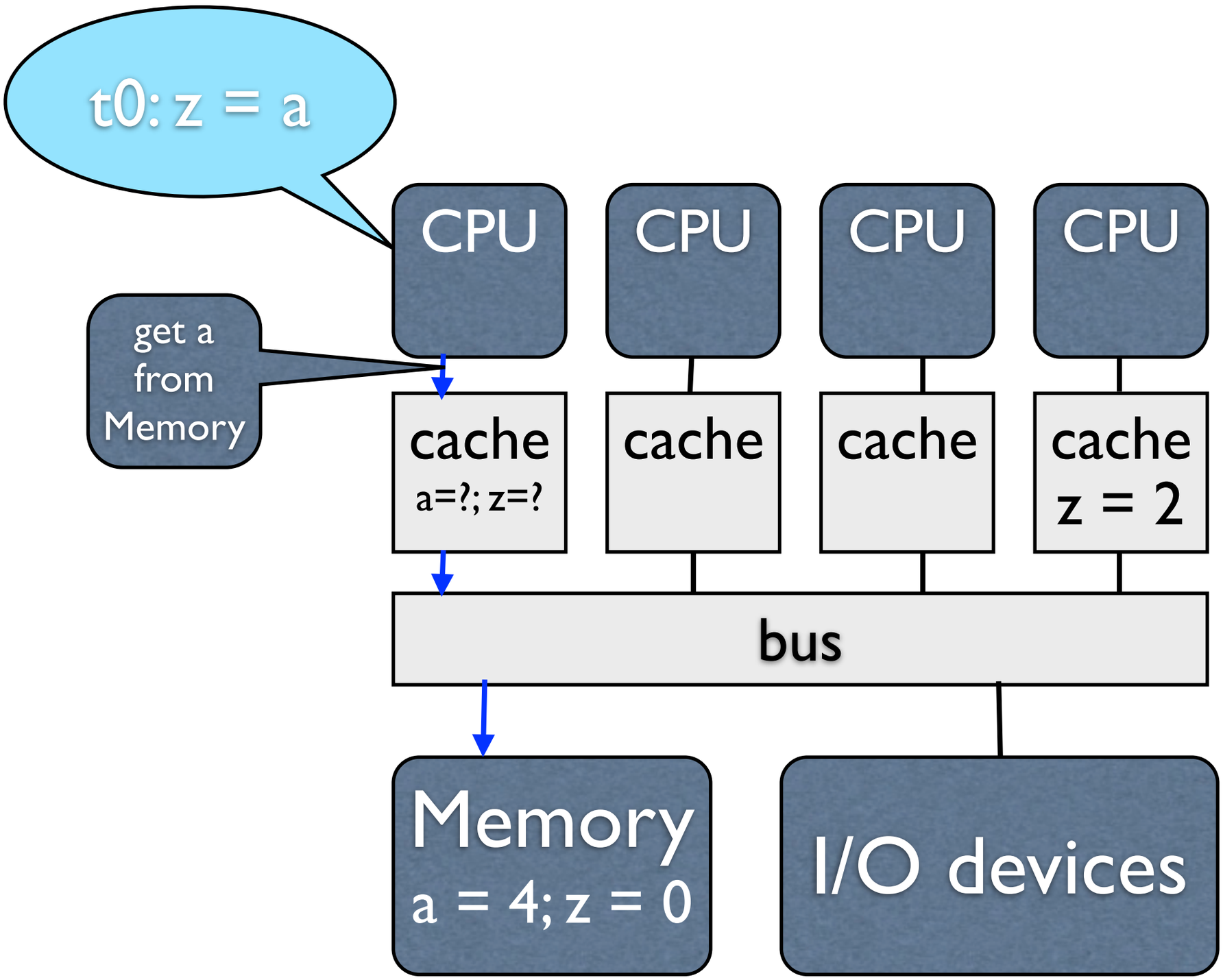


Global memory is spread across, and held in, the local memories of the different nodes of the machine

Processors will access their memory faster than their neighbors memory

Coherence is needed




t0: z = a
get a from Memory
CPU
CPU
CPU
CPU
cache
a=?; z=?
cache
cache
cache
z = 2
bus
Memory
a = 4; z = 0
I/O devices

load reg2, from (a)
st reg2, into (z)

t0: z = a

CPU CPU CPU CPU

a from Memory

cache
a=4; z=?

cache

cache

cache
z = 2

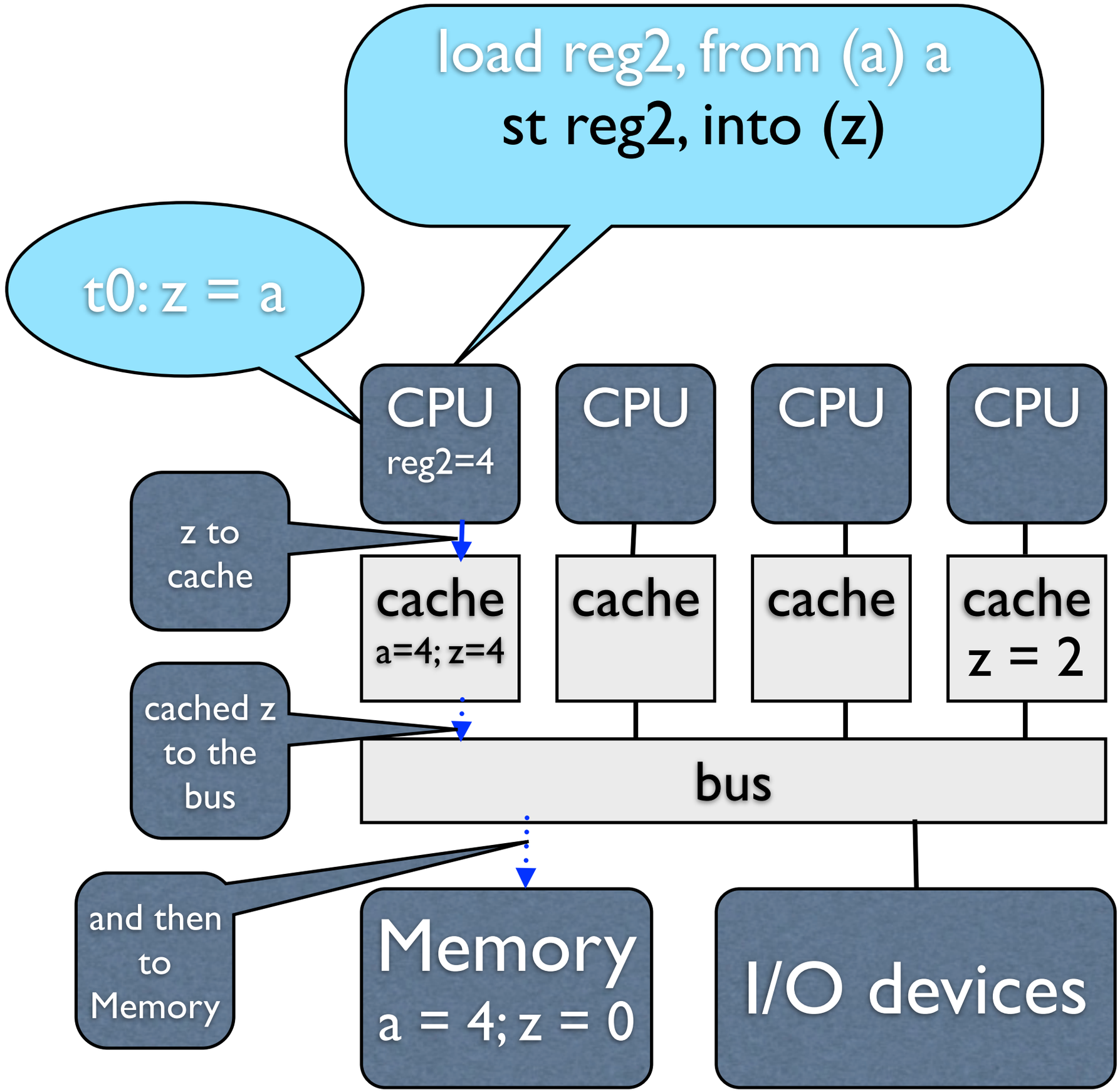
a from Memory

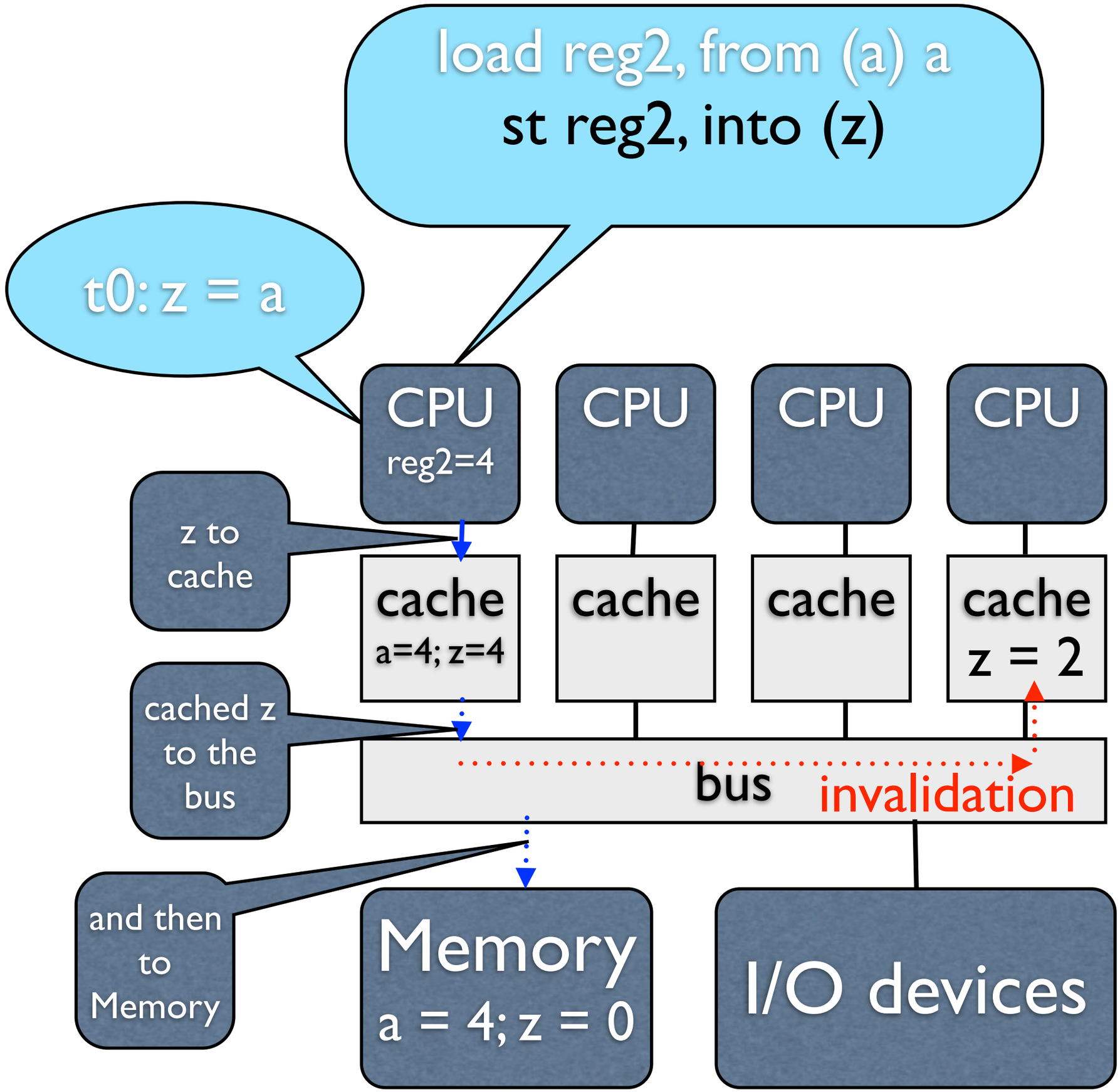
bus

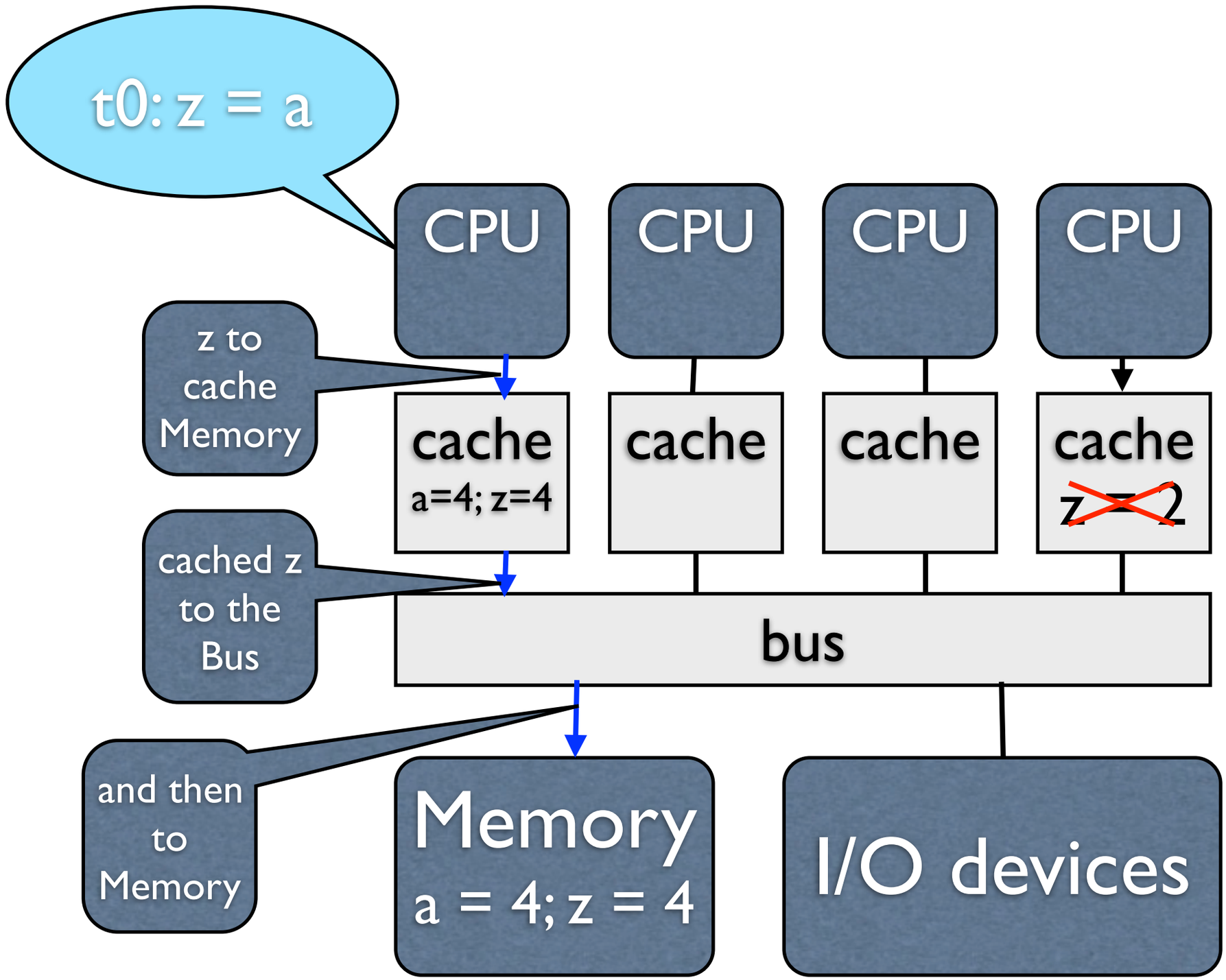
a from Memory

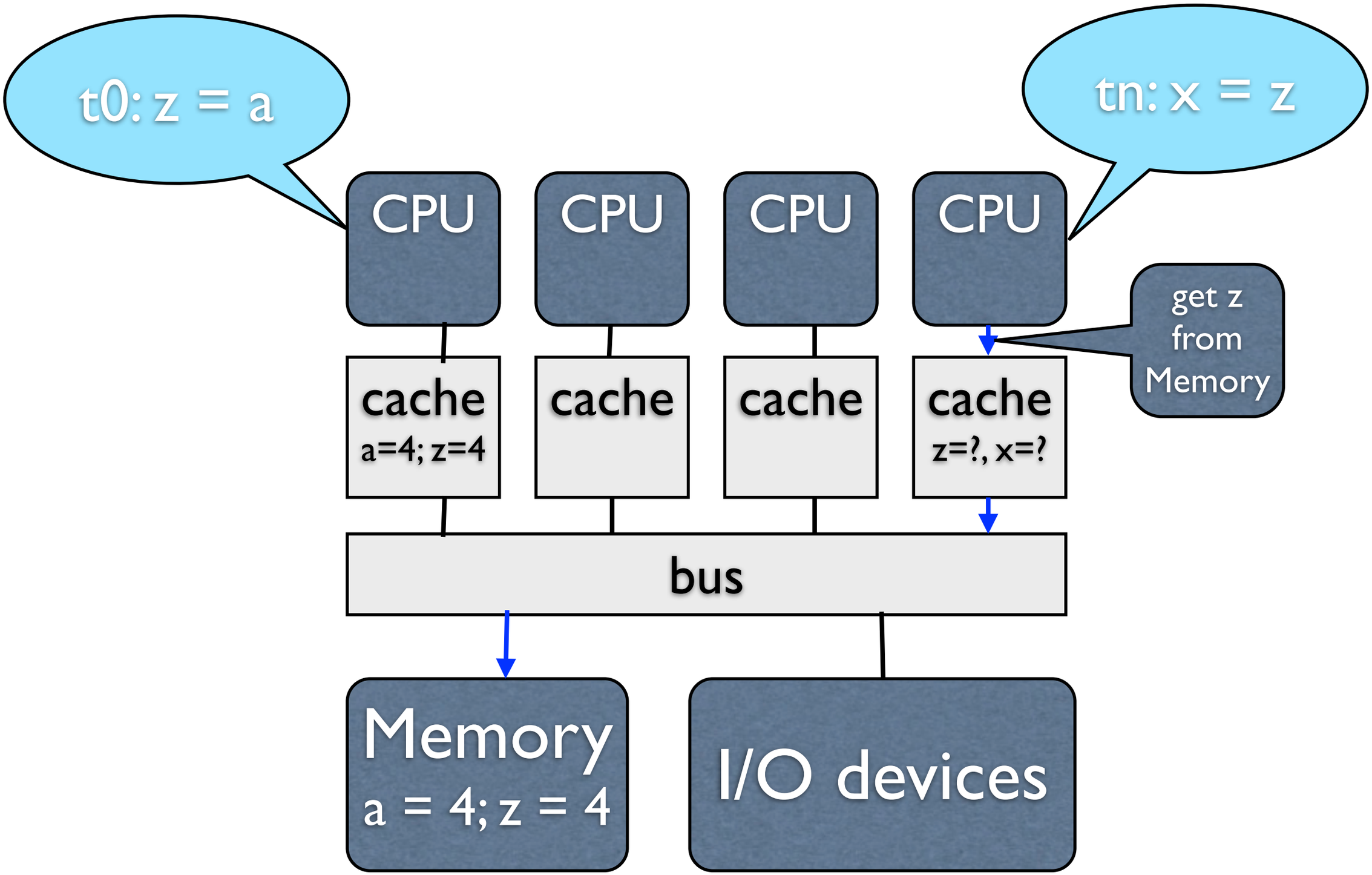
Memory
a = 4; z = 0

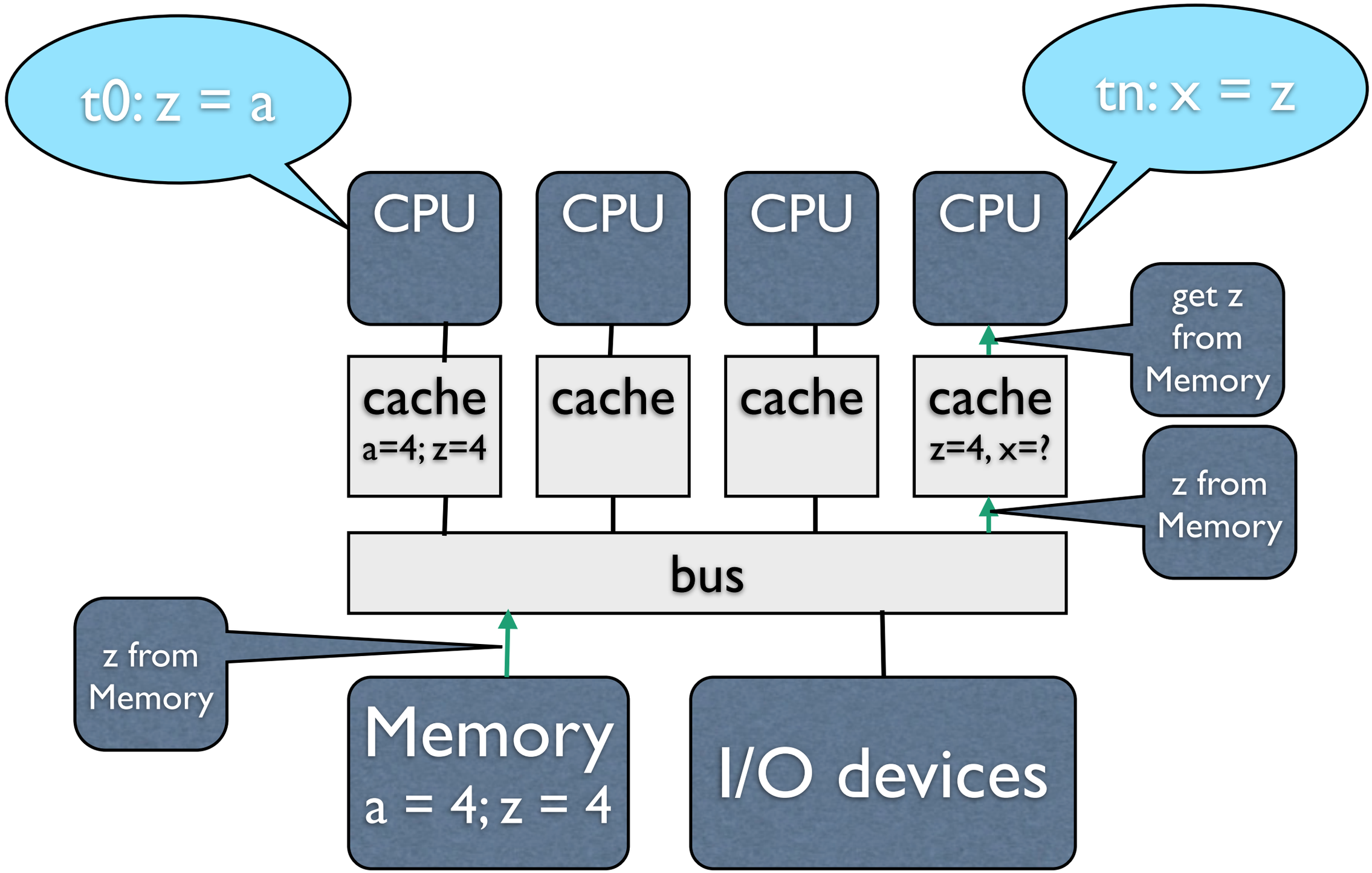
I/O devices

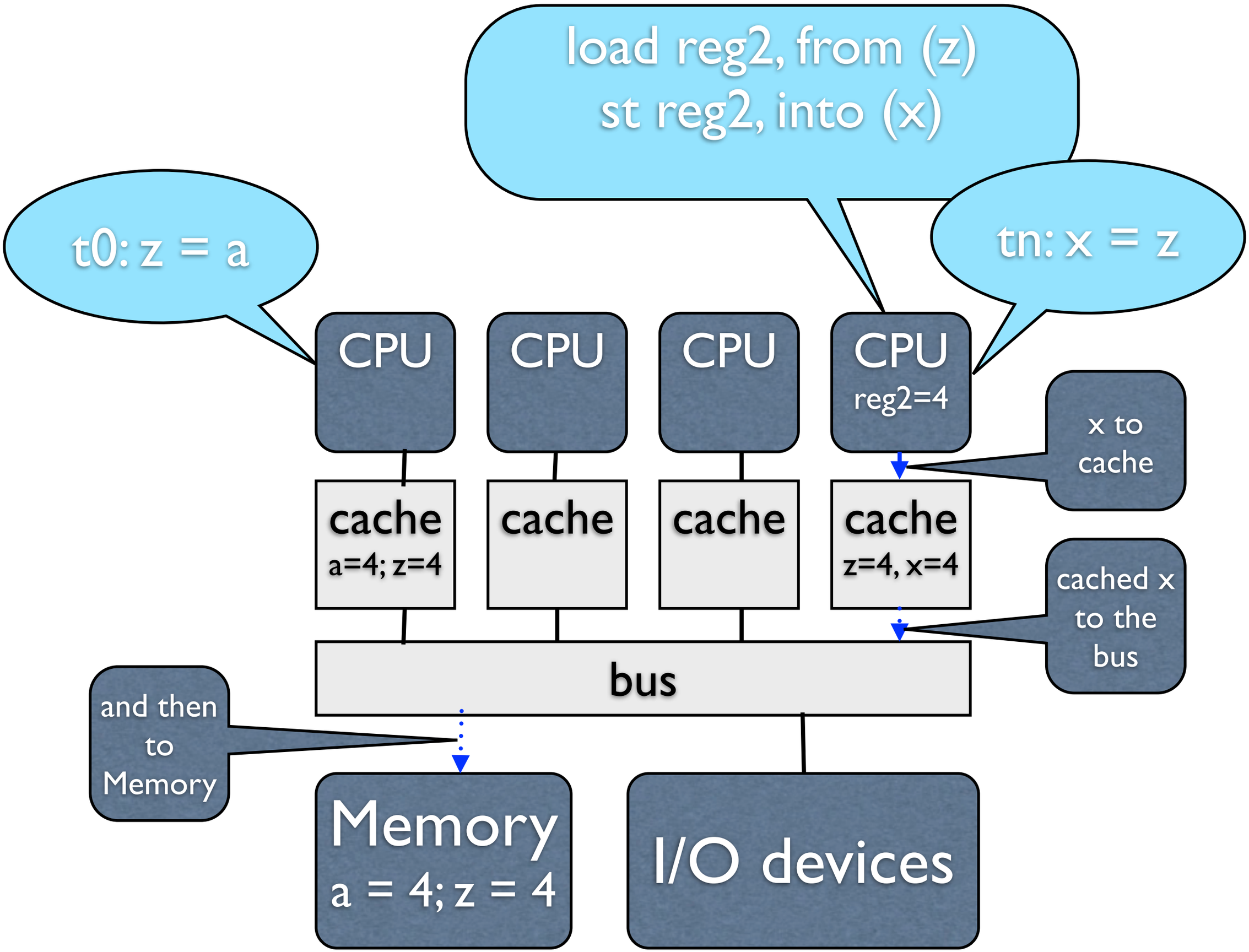

load reg2, from (a) a
st reg2, into (z)
t0: z = a
CPU
reg2=4
CPU
CPU
CPU
z to cache
cache
a=4; z=4
cache
cache
cache
z = 2
cached z to the bus
bus
and then to Memory
Memory
a = 4; z = 0
I/O devices


load reg2, from (a) a
st reg2, into (z)
t0: z = a
CPU
reg2=4
CPU
CPU
CPU
z to cache
cache
a=4; z=4
cache
cache
cache
z = 2
cached z to the bus
bus
invalidation
and then to Memory
Memory
a = 4; z = 0
I/O devices


t0: z = a
CPU
CPU
CPU
CPU
z to cache Memory
cache
a=4; z=4
cache
cache
cache
z = 2
cached z to the Bus
bus
and then to Memory
Memory
a = 4; z = 4
I/O devices


t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
cache
a=4; z=4
cache
cache
cache
z=?, x=?
get z from Memory
bus
Memory
a = 4; z = 4
I/O devices


t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
cache
a=4; z=4
cache
cache
cache
z=4, x=?
get z from Memory
z from Memory
bus
z from Memory
Memory
a = 4; z = 4
I/O devices


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

load reg2, from (z)
st reg2, into (x)

t0: z = a

tn: x = z

CPU

CPU

CPU

CPU
reg2=4

x to cache

cache
a=4; z=4

cache

cache

cache
z=4, x=4

cached x to the bus

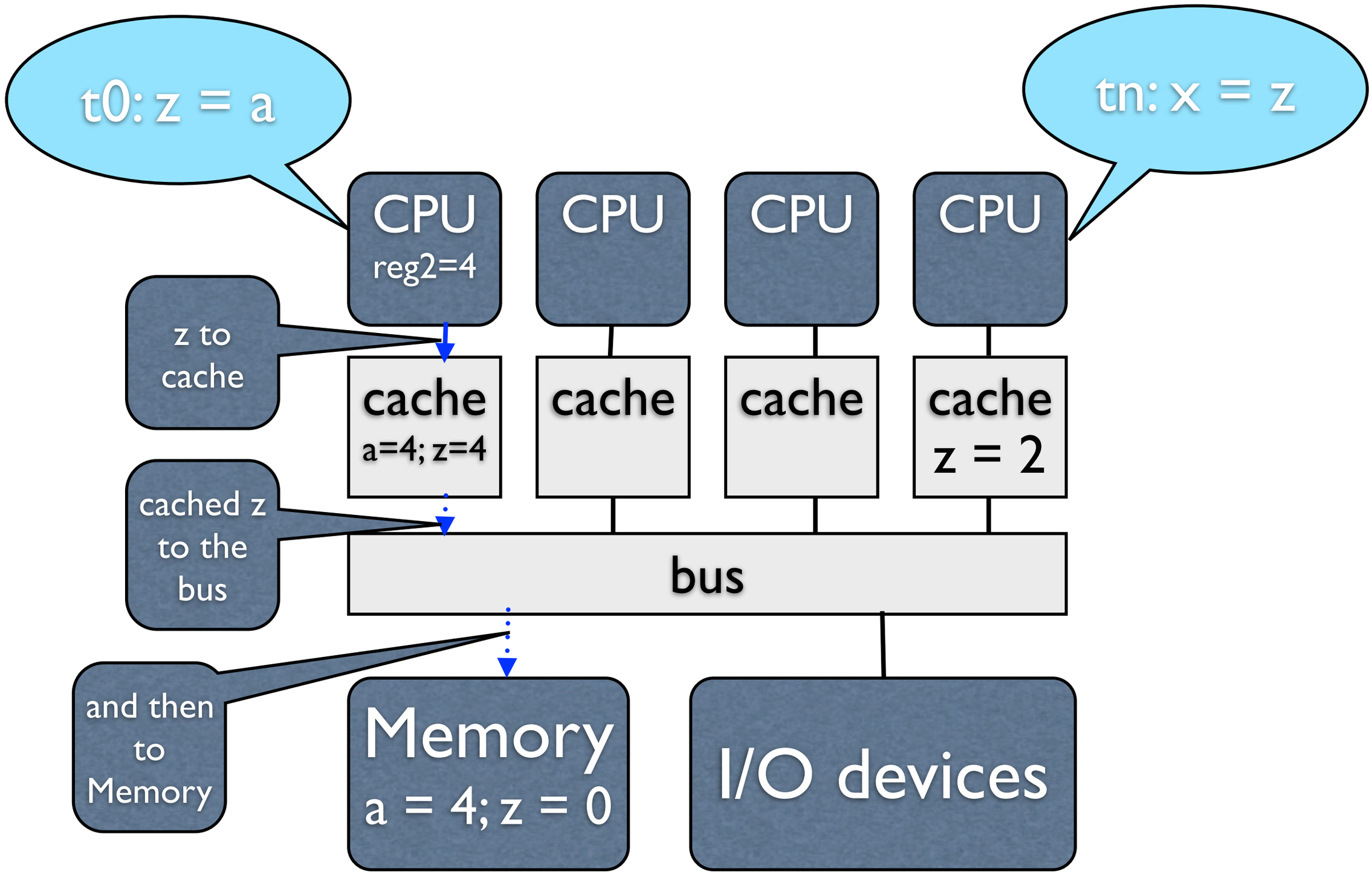
bus invalidation

and then to Memory

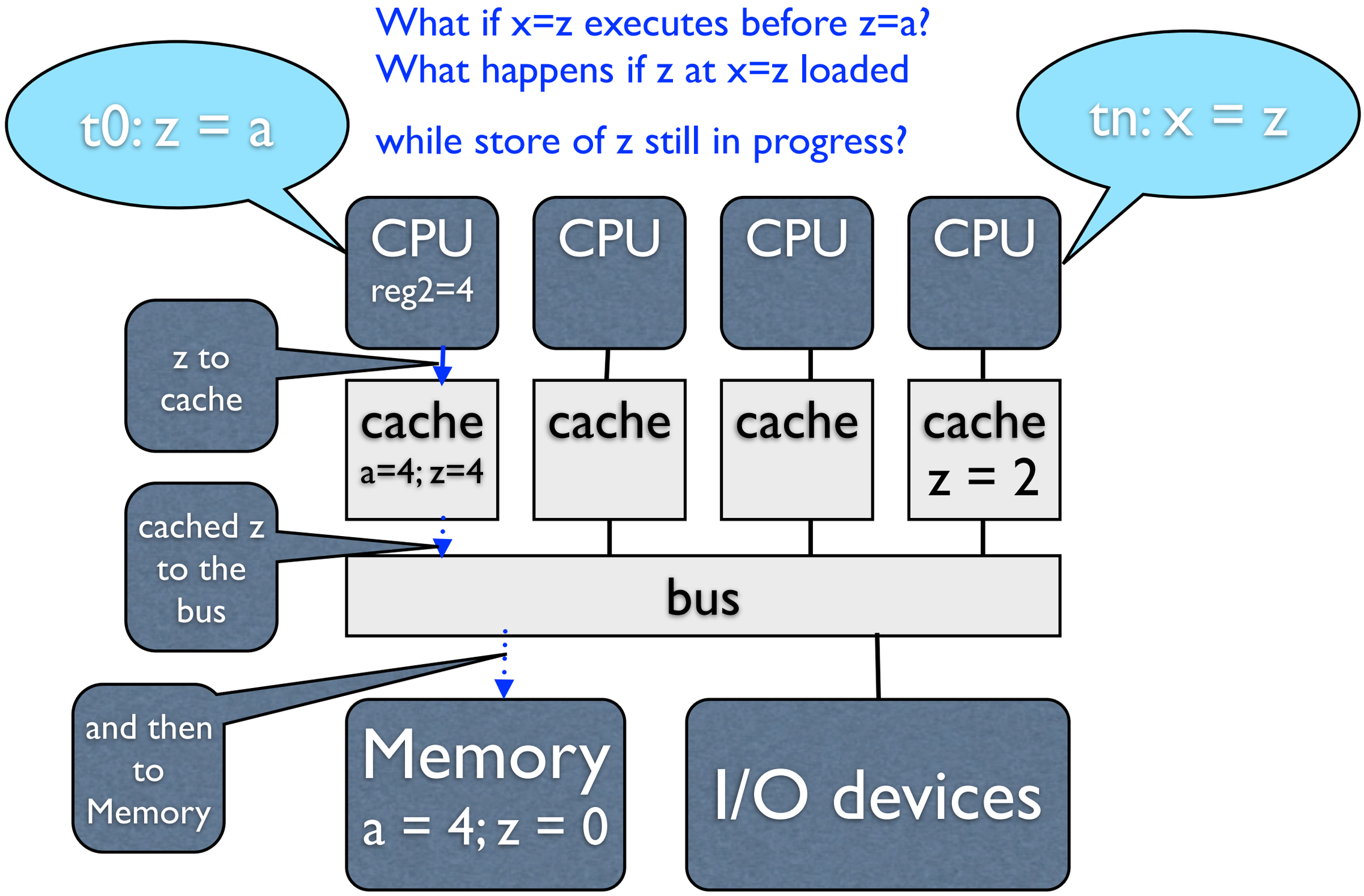
Memory
a, x, z = 4

I/O devices

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



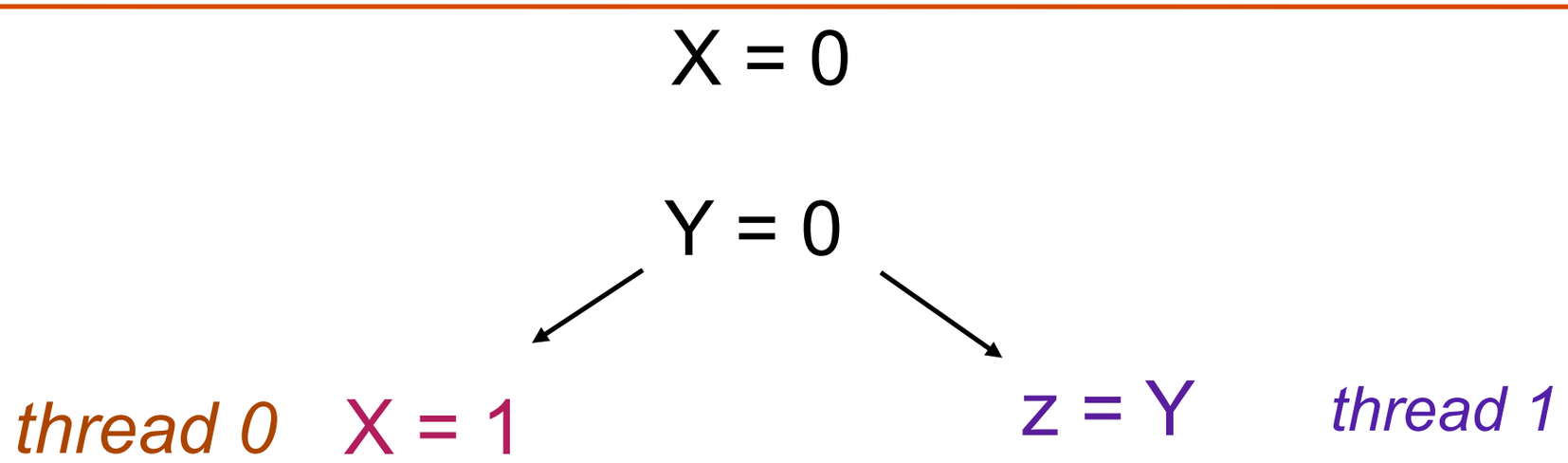
Software has to make sure operations occur in the right order across threads/processors



Sequential Consistency (SC)

- Coherence says that a read will get the last value written for a variable
- Consistency is concerned with the interactions between writes to different variables
- Sequential consistency (see Lamport paper) is when ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

SC Example

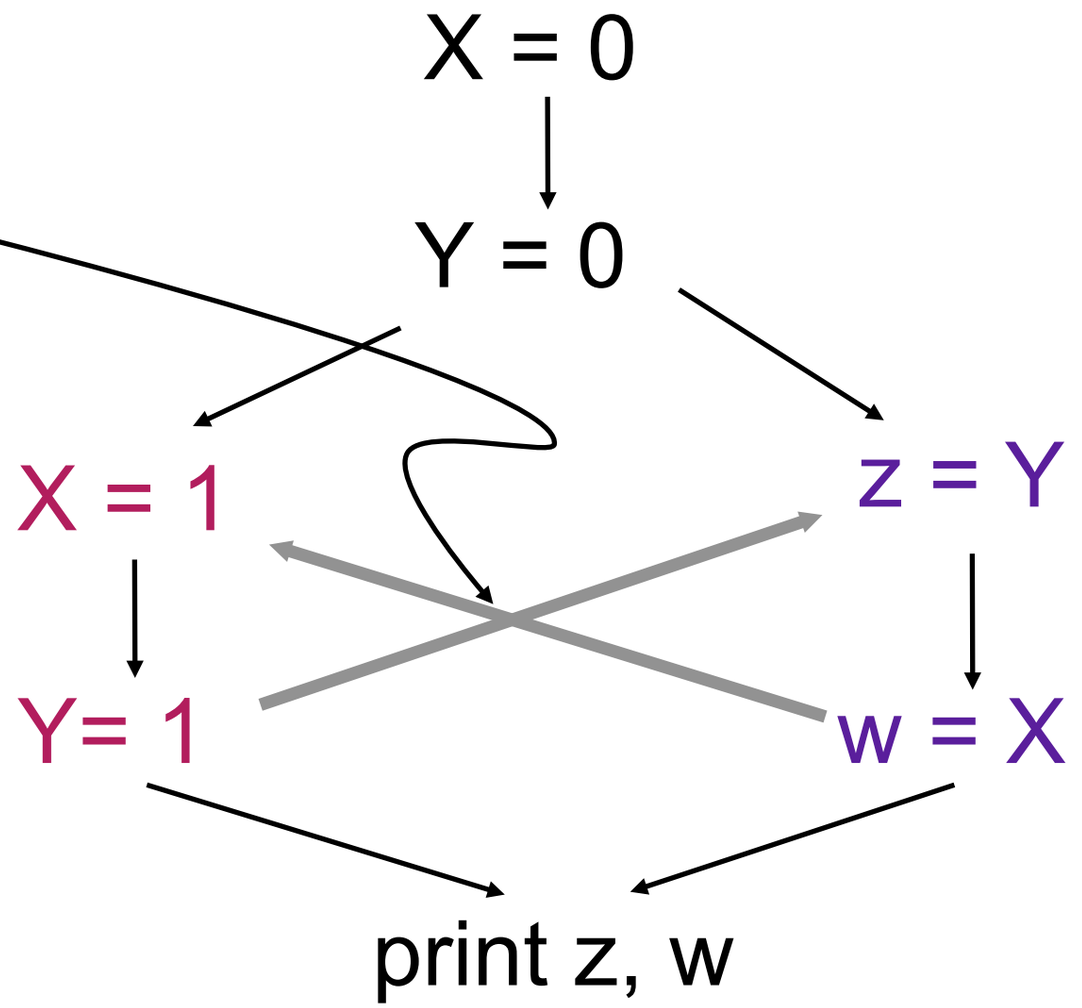


Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: Not with sequential consistency

Sequential Consistency

Relative execution order implied by assigned value



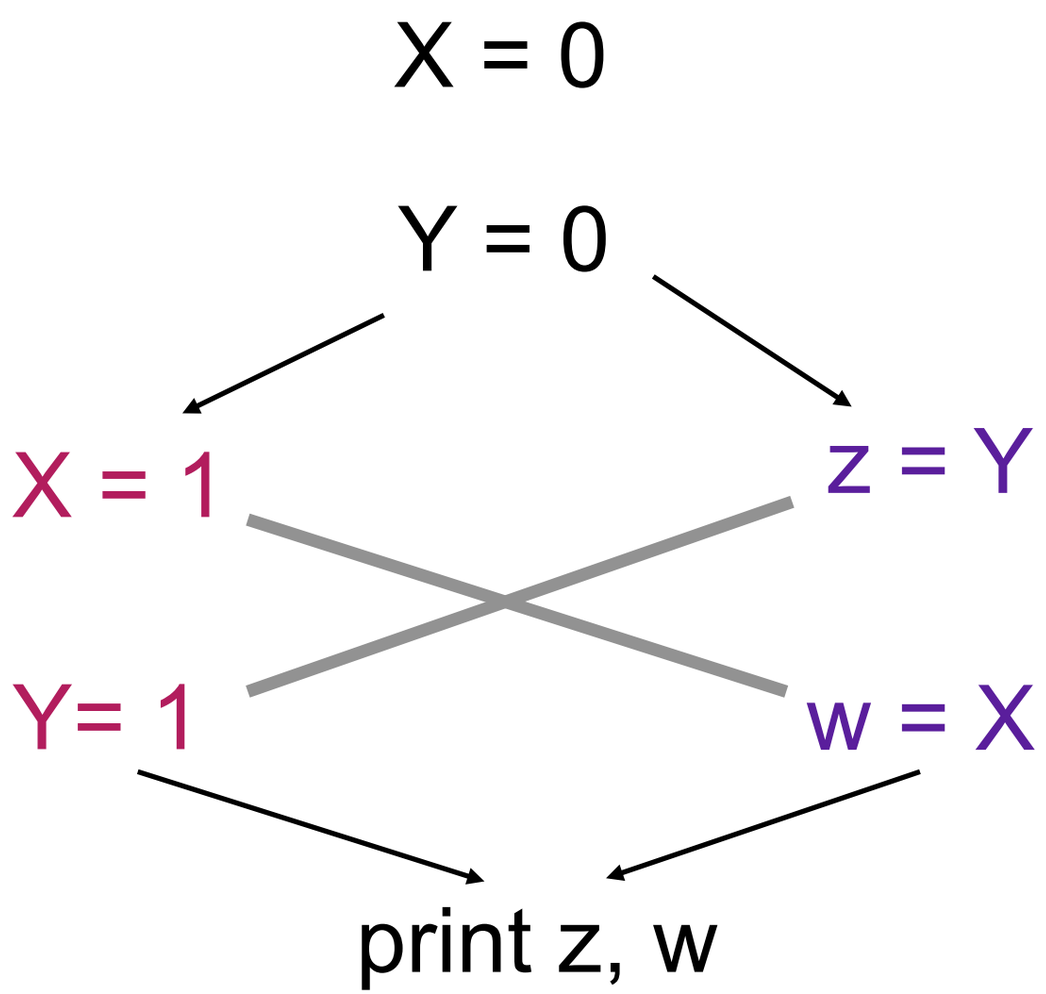
Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: ***NO.*** *For* ***z == 1*** and ***w == 0*** it is necessary for either "X=1" and "Y=1" to execute in a different order, or for "z=Y" or "w=X" to execute in a different order.

Many languages violate SC by default

Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: YES. Java semantics allow "X=1" and "Y=1" to execute in a different order, or for "z=Y" or "w=X" to execute in a different order.



Sequential Consistency (SC)

- Coherence says that a read will get the last value written for a variable
- Consistency is concerned with the interactions between writes to different variables
- Sequential consistency (see Lamport paper) is when ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

We generally want programs to be SC

- After we parallelize the program the executions of the program should all give an answer such that ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

- Moreover, it is often good to have the program give the same answer as a sequential, one node, one core, one thread, etc. implementation of the algorithm

- It will be our responsibility as programmers to ensure this -- the hardware and software will not

Shared memory programming models

- Can either be a language, language extension, library or a combination
  - Java is a language and associated *virtual machine* that provides runtime support
  - OpenMP is a language extension (for C/C++ and Fortran) and an associated library (or *runtime*)
  - Pthreads (or *Posix Threads*) is a library with C/C ++ and Fortran bindings

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.

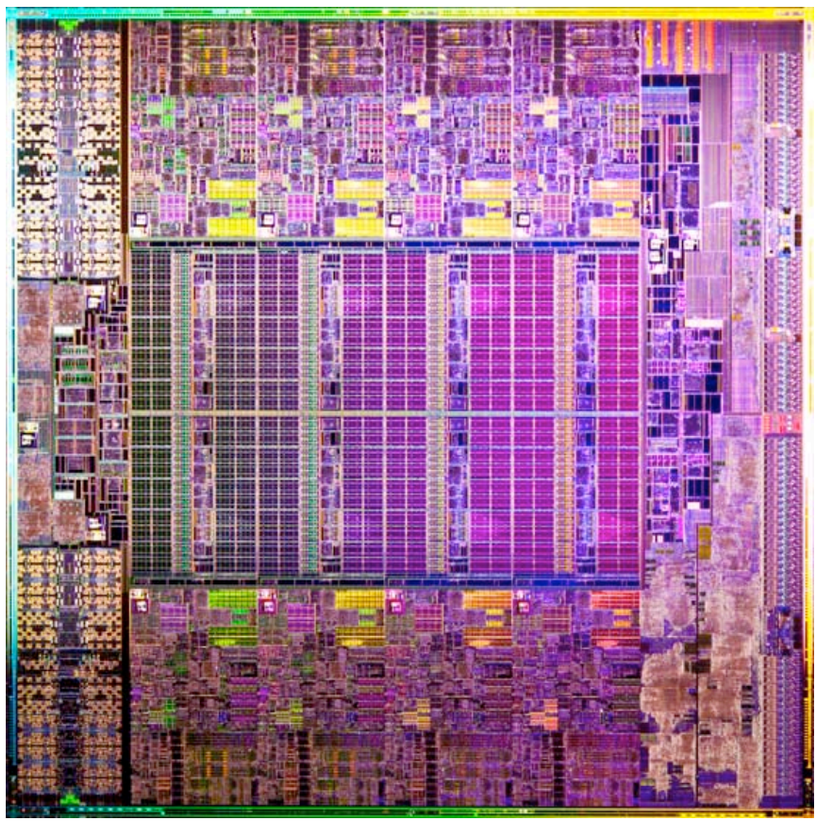
What executes in parallel?

```
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]
}
```

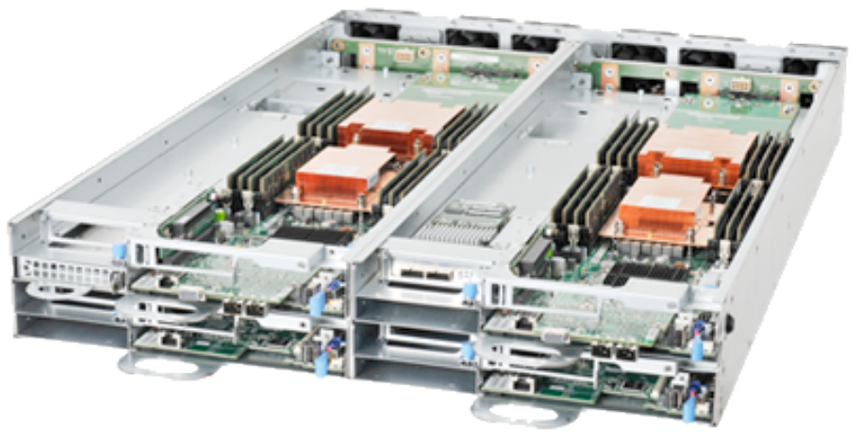
- *pragma* appears like a comment to a non-OpenMP compiler
- pragma requests parallel code to be produced for the following for loop

processors, nodes, processes and threads



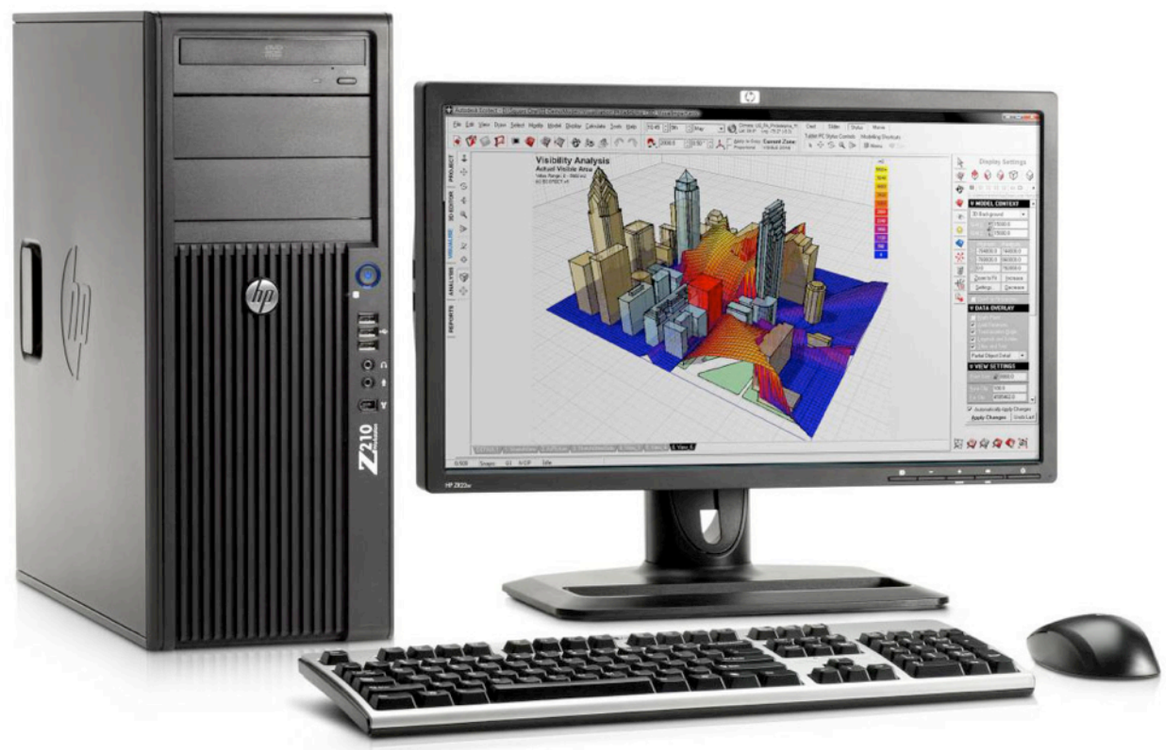
A processor is a physical piece of hardware

processors, nodes, processes and threads



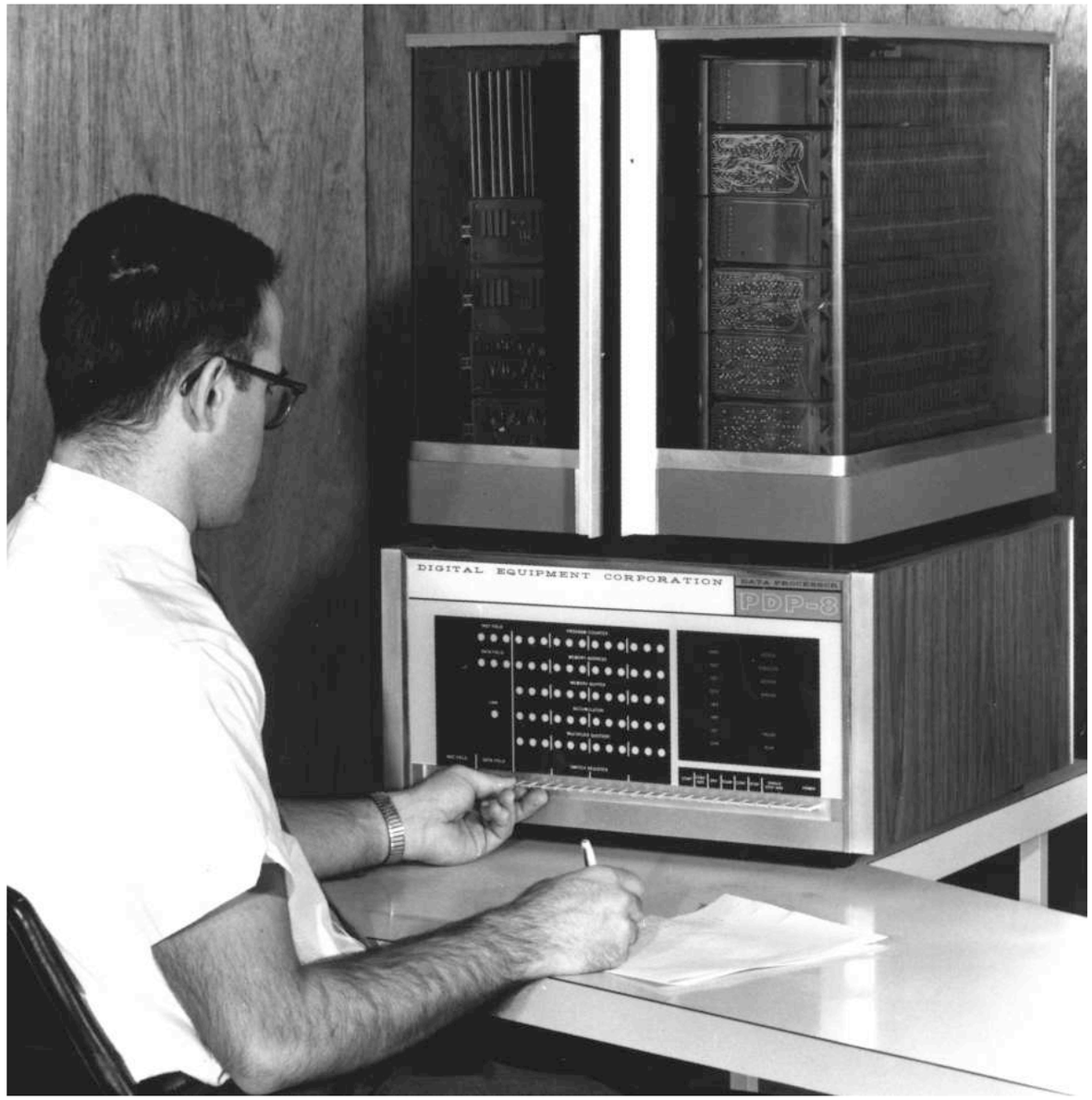
A node is a processor along with associated devices (disk drive, memory, i/o cards, communication cards, etc.

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



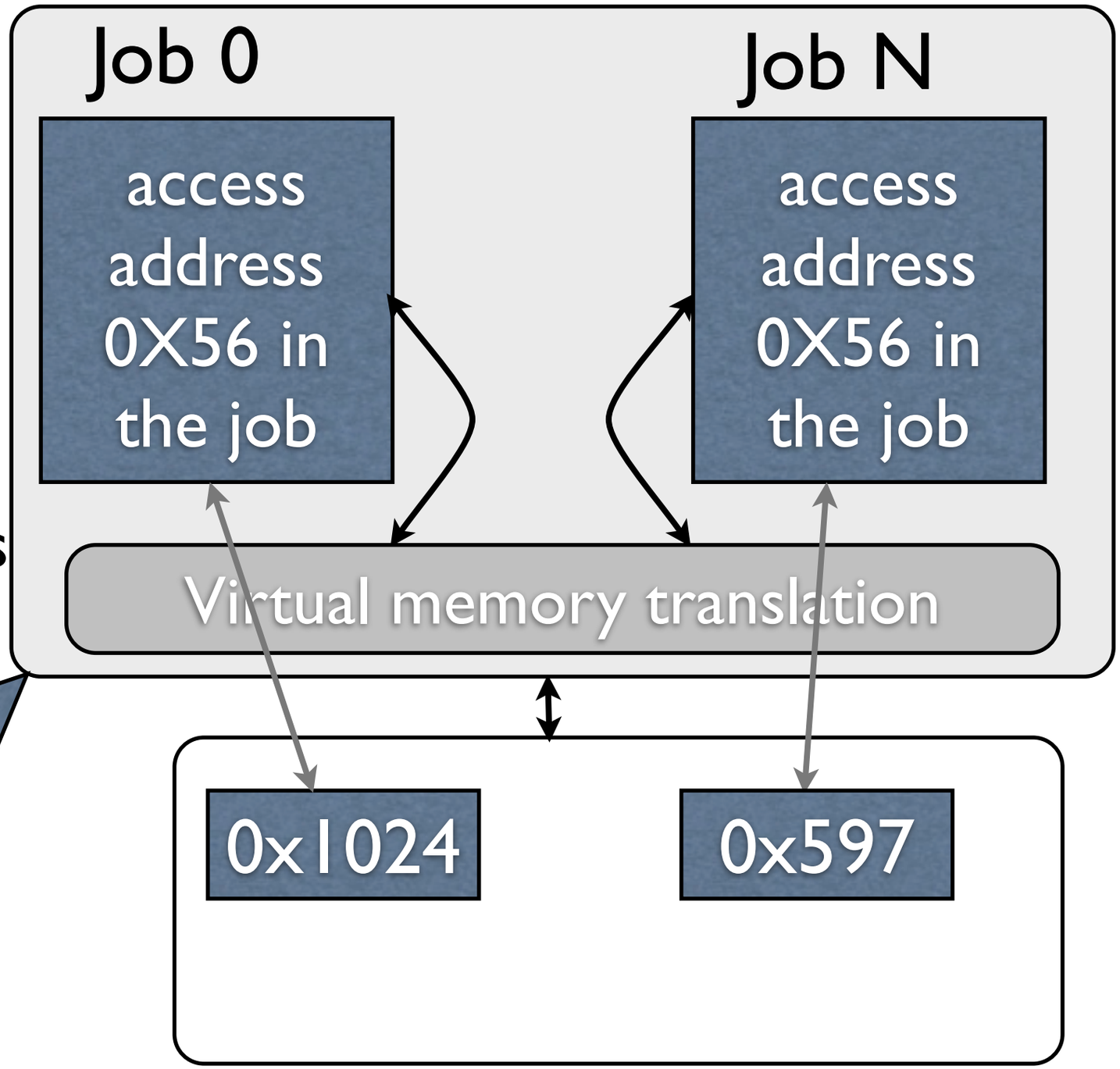
A Digital Equipment Corporation
PDP-8 (programmable data processor)
Early low-cost mass produced computer

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.

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



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

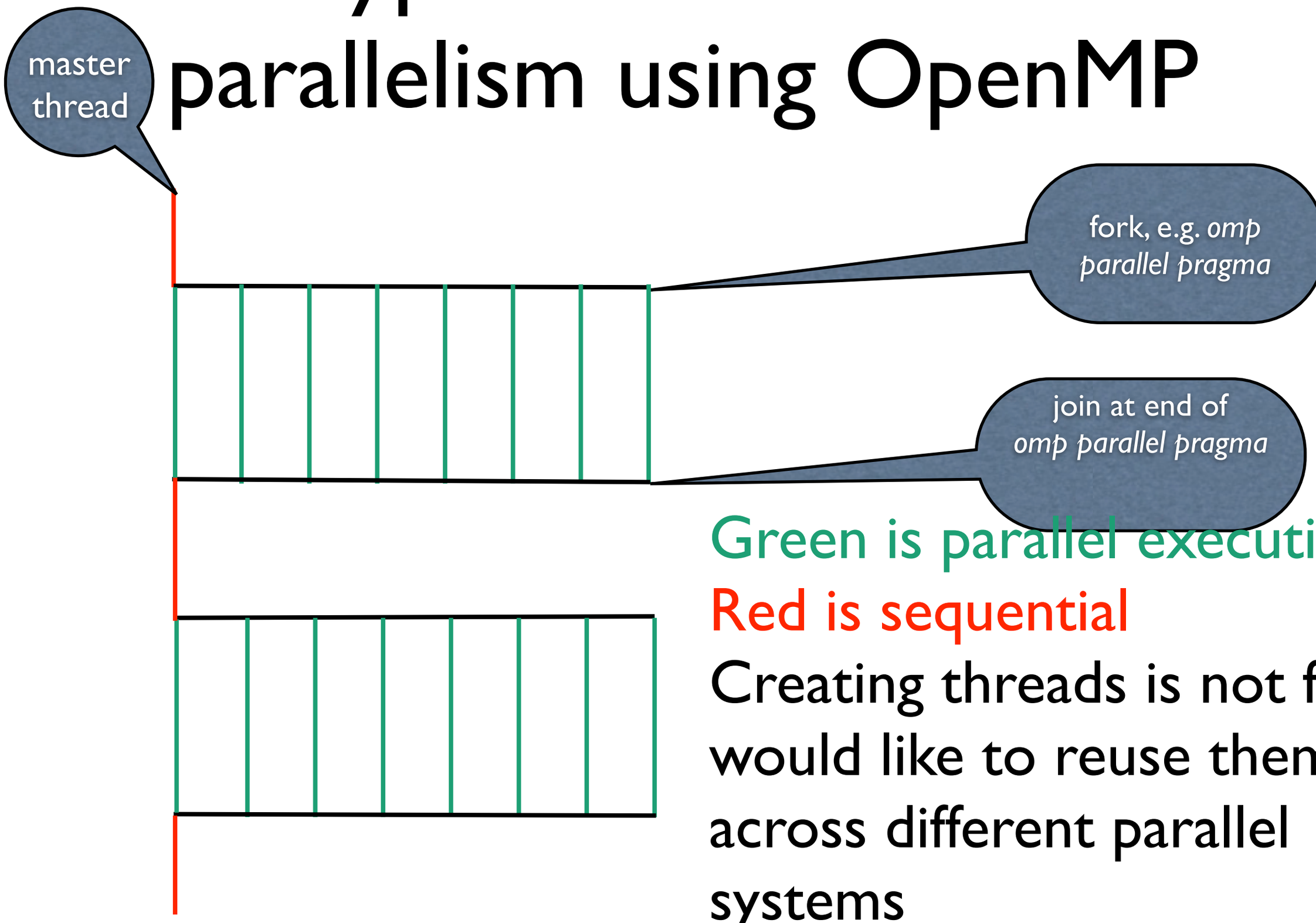
Threads and processes -- summary

- 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



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]
}
```

barrier

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

- 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.

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