

ECE 563

Programming Parallel Machines

- The syllabus:

<https://engineering.purdue.edu/~smidkiff/ece563/files/syllabus.pdf>



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.](#)

© 2010 [Purdue University](#). [An equal access, equal opportunity university.](#)

"The contents of this web application were developed under a grant from the Department of Education. However, those contents do not necessarily represent the policy of the Department of Education, and you should not assume endorsement by the Federal Government"

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

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

Building information:

https://www.purdue.edu/ehps/emergency_preparedness/bep/WANG--bep.html

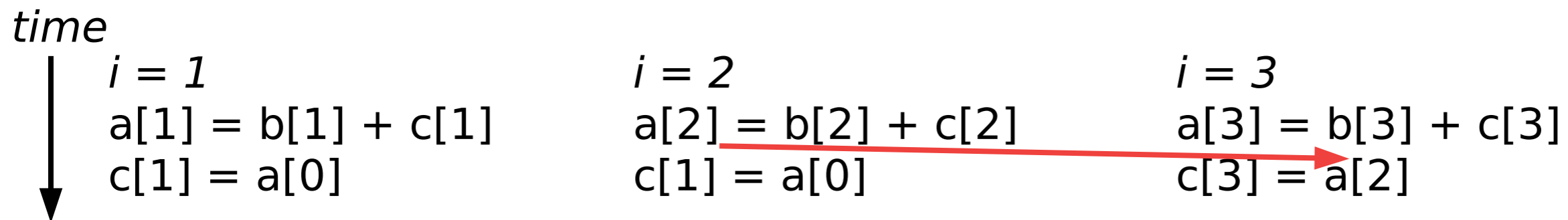
What is our goal in this class?

- 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

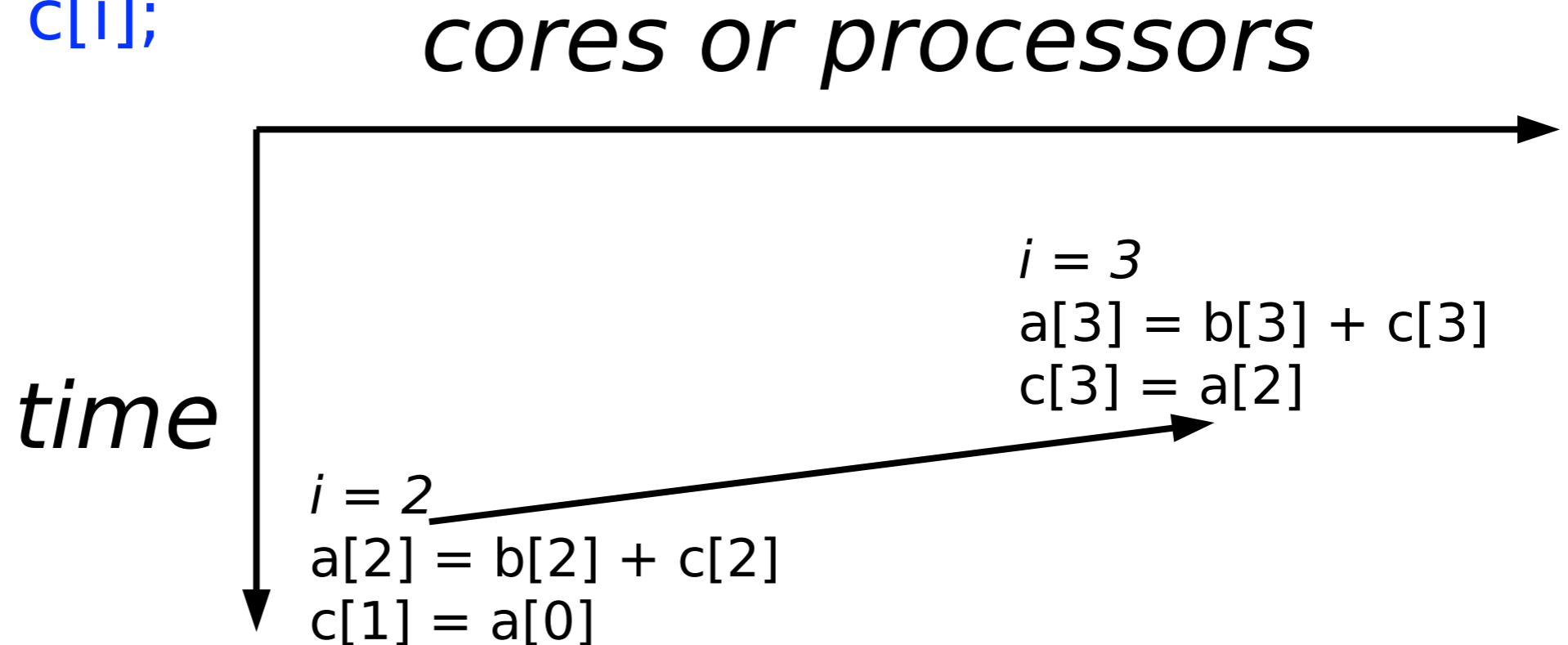


Note that data is produced in one iteration
and consumed in another.

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]  
}
```

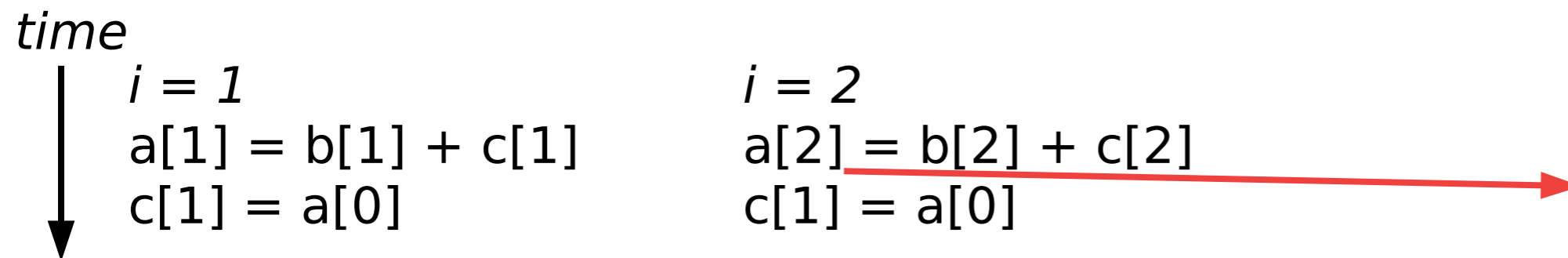


What if the processor executing iteration $i=2$ is delayed for some reason? *Disaster* - the value of $a[2]$ to be read by iteration $i=3$ is not ready when the read occurs!

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.

time



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

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

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

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?

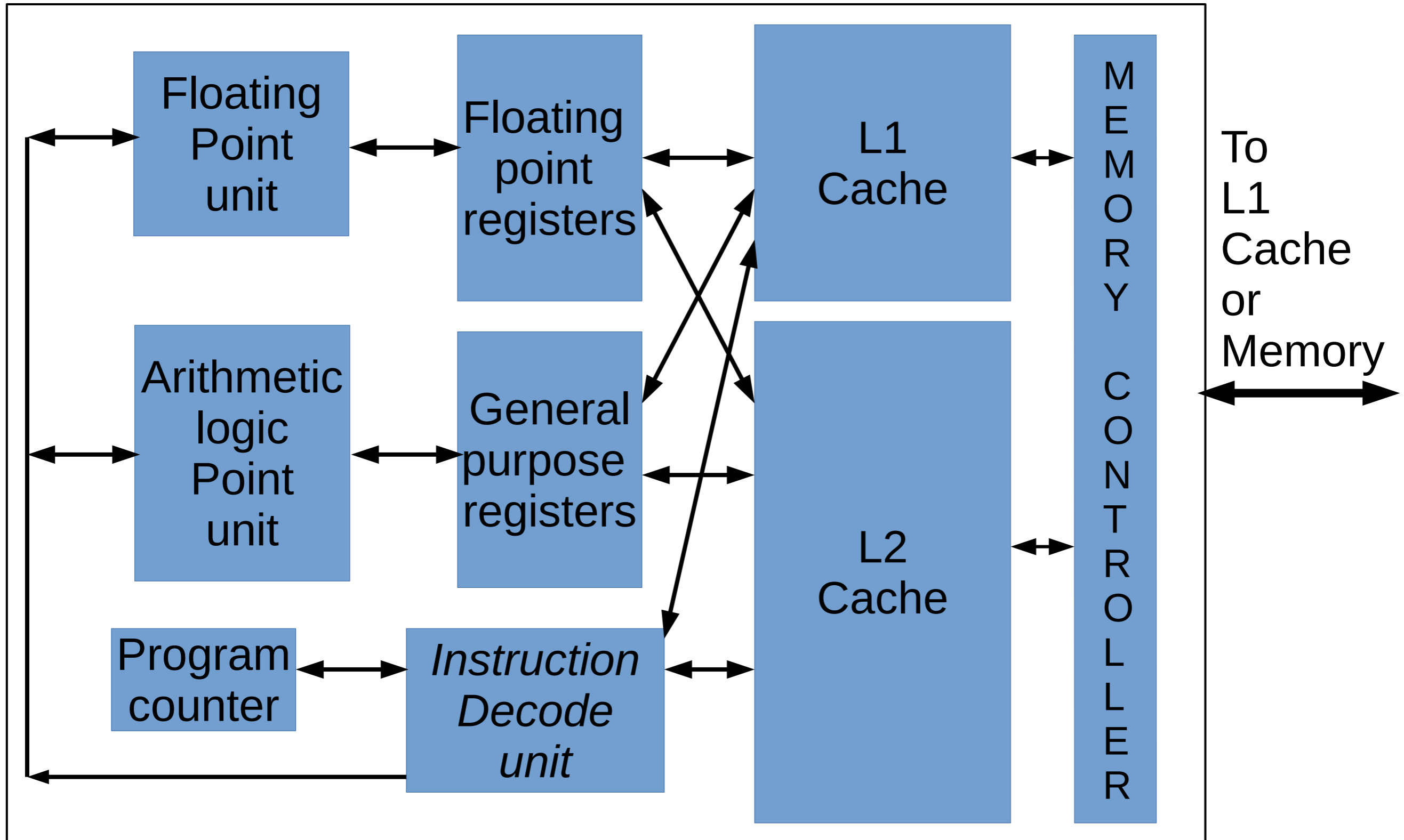
- Most work in most programs, especially numerical programs, is in a loop
- Thus effective parallelization generally requires parallelizing loops
- *Amdahl's law* (discussed in detail later in the semester) says that, e.g., if we parallelize 90% of a program we will get, at most, a *speedup* of 10X, 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*

A simple core/processor



Registers

- Registers are usually directly referenced and accessed by machine code instructions
- On a RISC (*Reduced Instruction Set Computer*) almost all instructions are register-to-register or register to memory

Addi r1, r2, r3 // $r3 = r1 + r2$, i.e., r3 gets the value of the sum of the contents of r2 and r3

ld r1 (r2) // load register 1 with the value in the memory location whose address is in r2

Registers can be accessed in a single cycle and are the fastest storage.

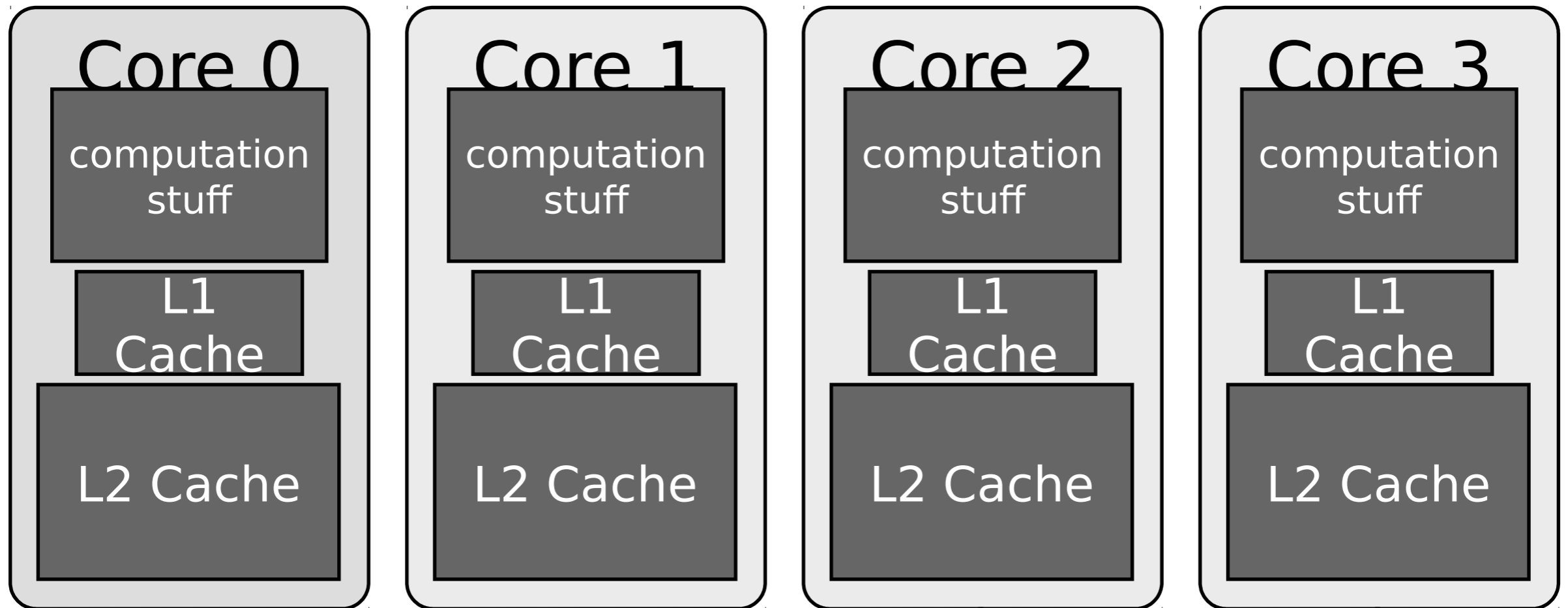
Typically in a RISC machine there are ~64 registers, with 32 general purpose, 32 floating point, plus some others

Intel IA86 has many fewer registers and can do memory to memory operations

Caches

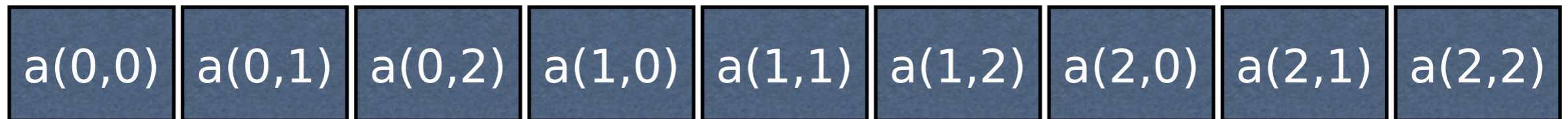
- Processors are much faster than memory
- Core i7 Xeon 5500 (from https://software.intel.com/sites/products/collateral/hpc/tune/performance_analysis_guide.pdf)
 - fastest (L1) cache ~4 cycles
 - next fastest (L2) cache ~10 cycles
 - next fastest (L3) cache ~40 cycles
 - DRAM 100ns or about 300 cycles

Caches

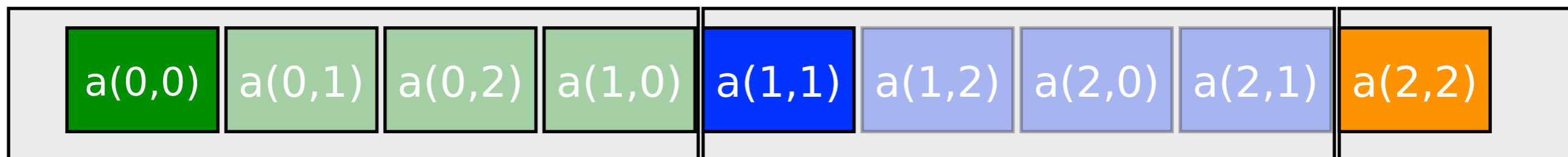


How is memory laid out?

a 2D array in memory looks like:

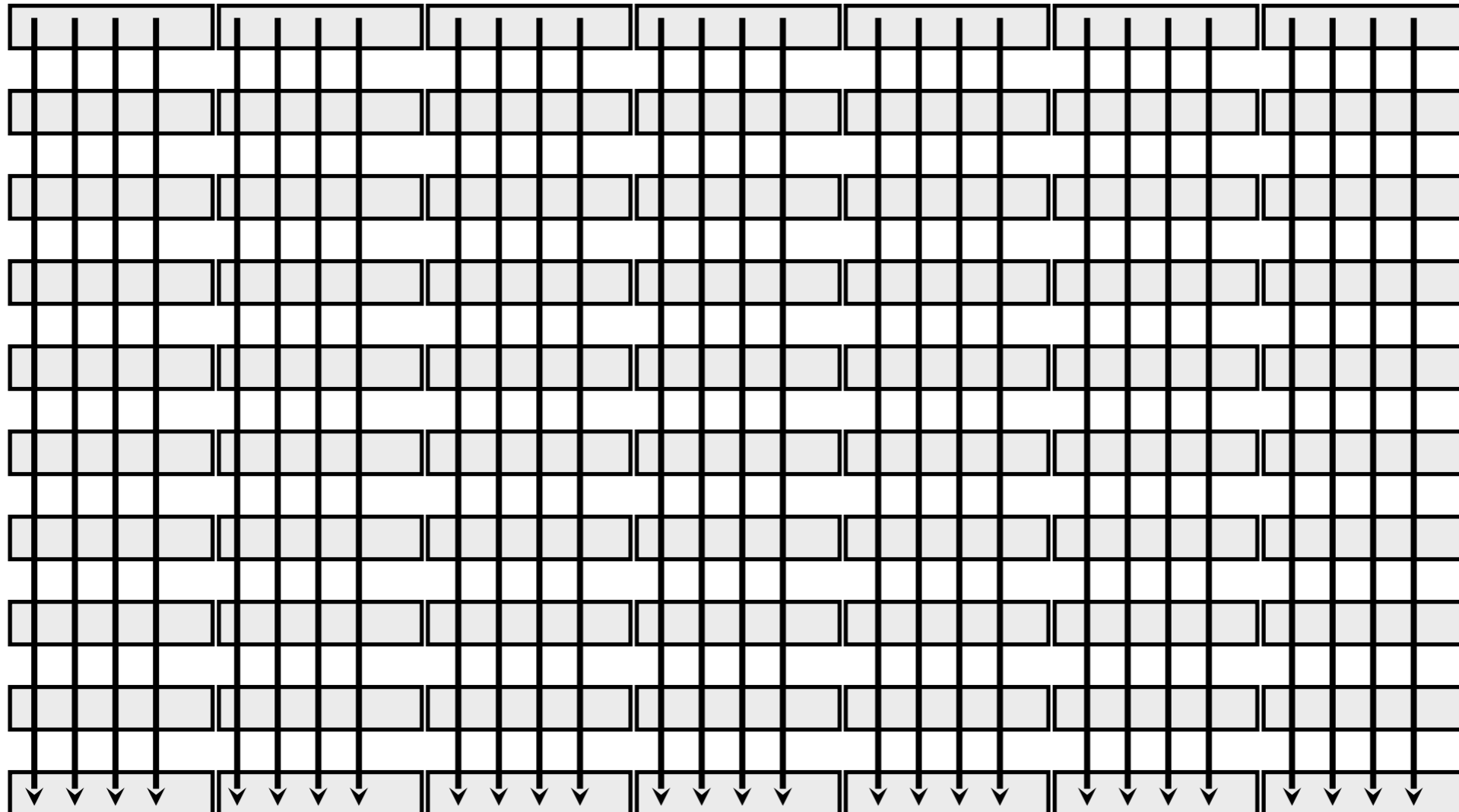


When you read one word, several words are brought into cache



Accessing Arrays

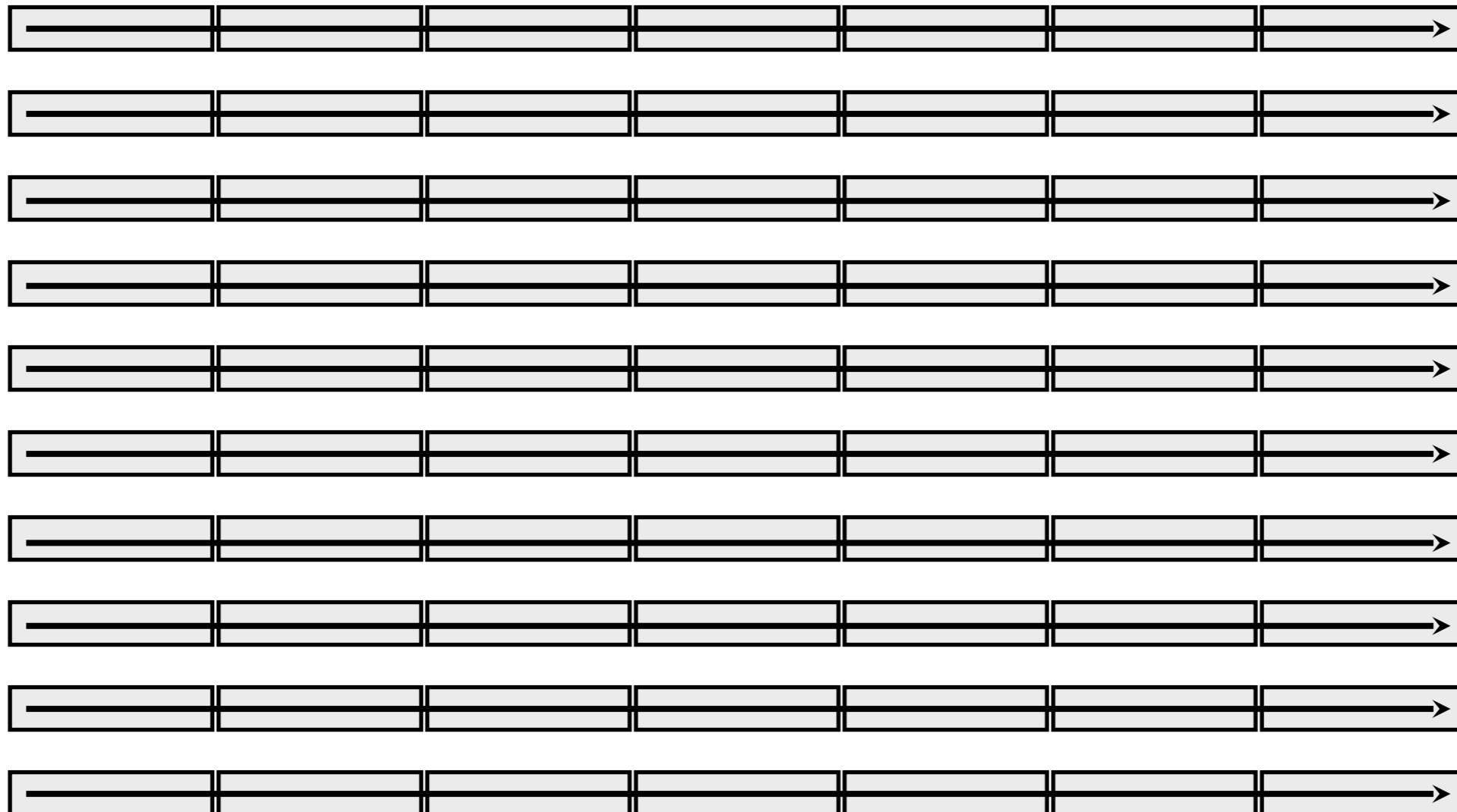
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        . . . = a[j,i]  
    }  
}
```



Accessing Arrays

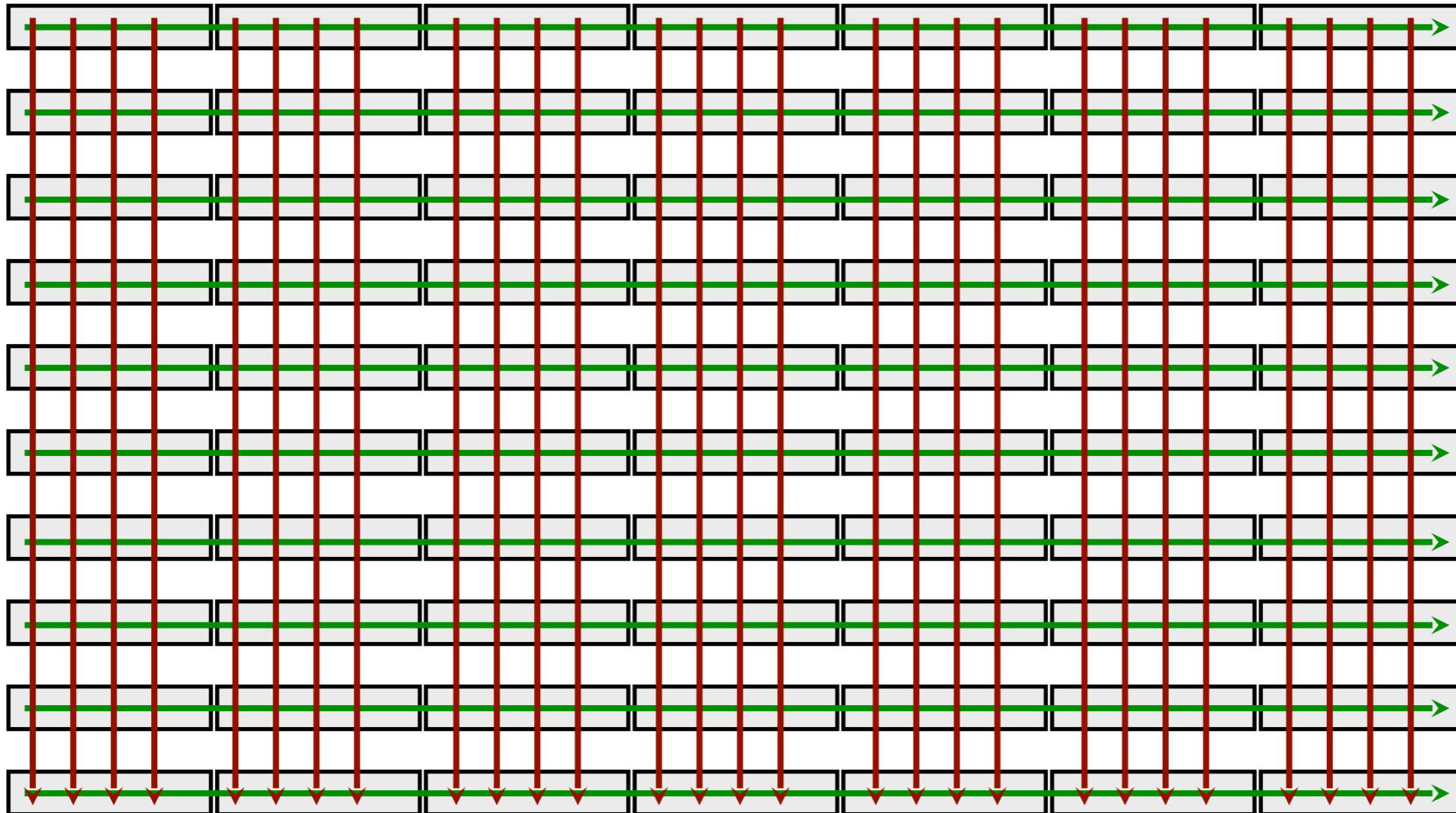
```
for (int j = 0; j < n; j++) {  
    for (int i = 0; i < n; i++) {  
        ... = a[j,i]  
    }  
}
```

*loop
interchange*



Accessing Arrays

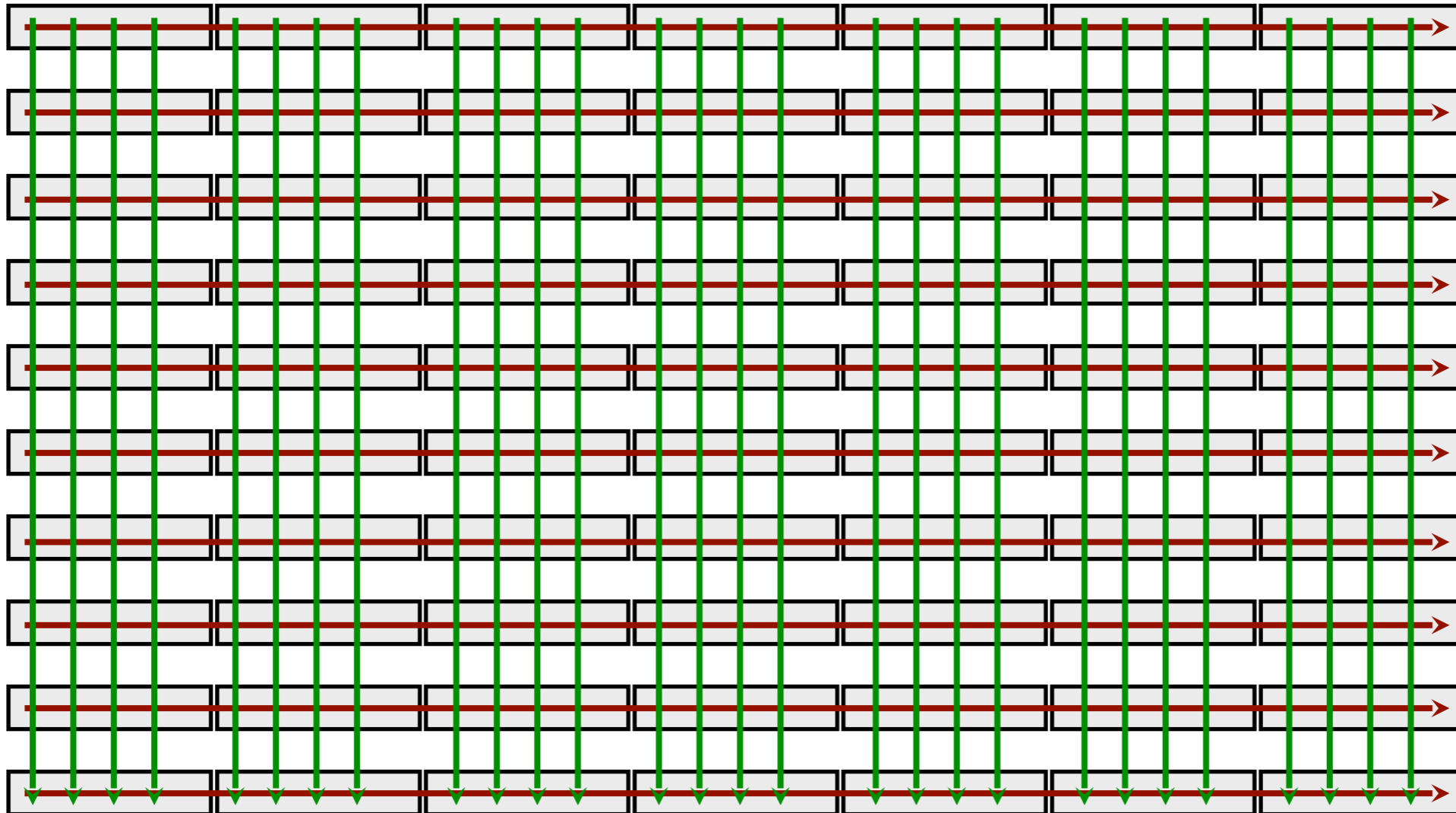
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        a[i,j] = a[j,i]  
    }  
}
```



Accessing Arrays

```
for (int j = 0; j < n; j++) {  
  for (int i = 0; i < n; i++) {  
    a[i,j] = a[j,i]  
  }  
}
```

*loop
interchange
doesn't help*



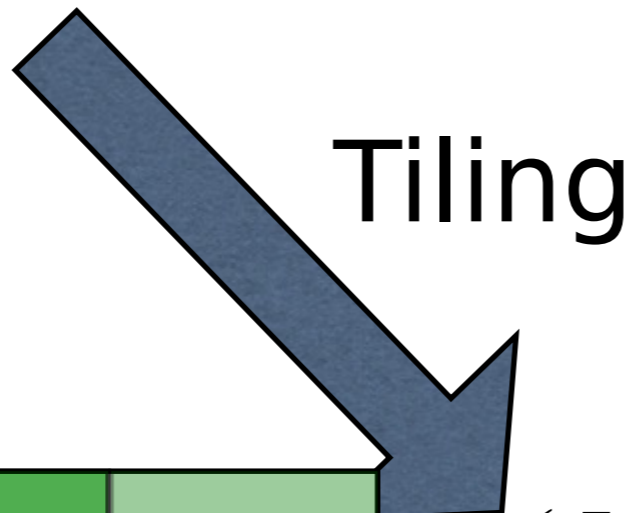
Tiling solves this problem

- This is discussed in detail in ECE 468/573, compilers
- Basically, extra loops are added to the code to allow blocks, or *tiles*, of the array *that fit into cache* to be accessed
 - As much work as possible is done on a tile before moving to the next tile
 - Accesses within a tile are done within the cache
- Because tiling changes the order elements are accessed it is not always legal to do

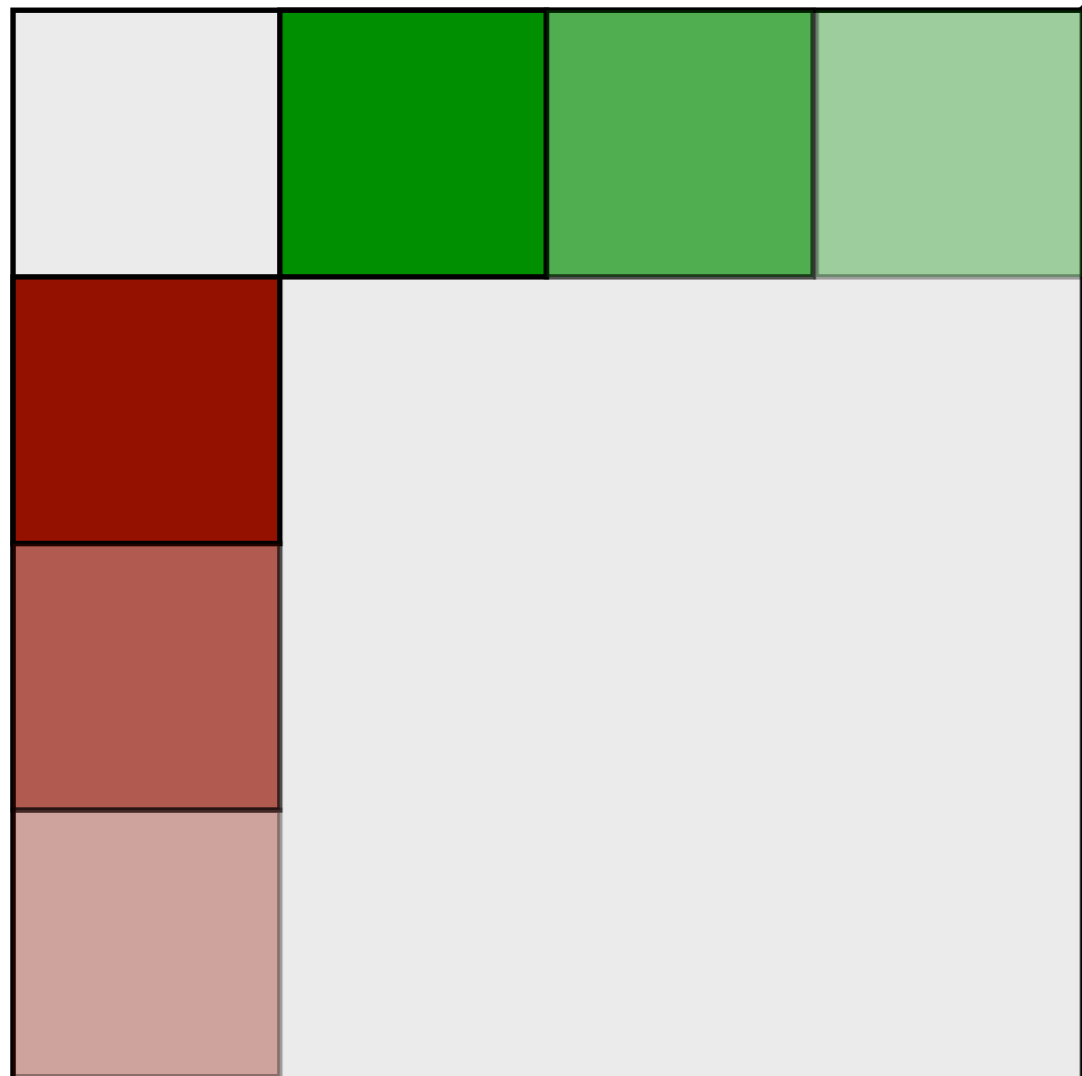
```

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    a[i,j] = a[j,i]
  }
}

```



a array



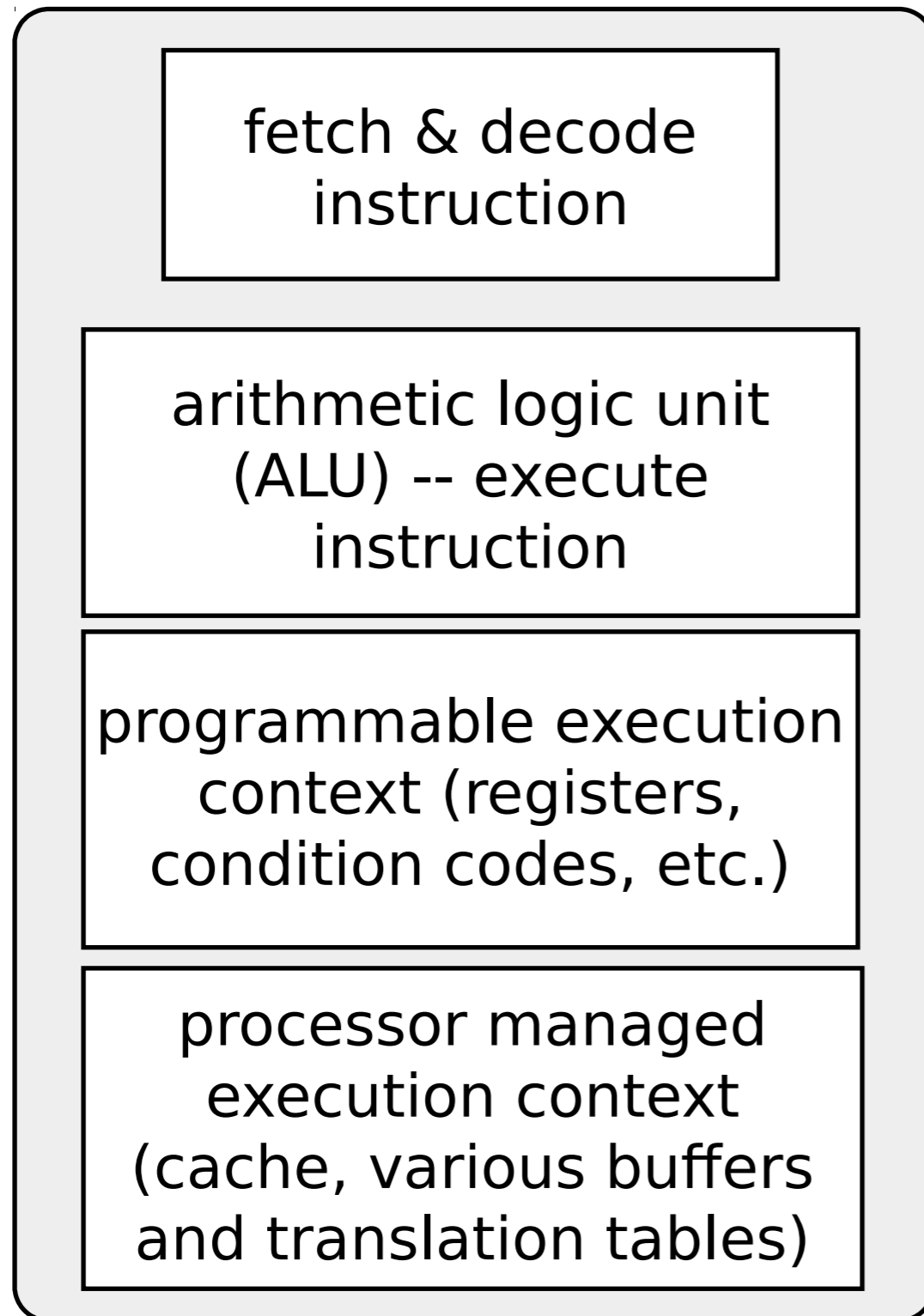
```

for (tI = 0, tI < n; tI += 64) {
  for (tJ = 0, tJ < n; tJ += 64) {
    for (i = tI; i < min(tI+63, n); i++) {
      for (j = tJ; j < min(tJ+63, n); j++) {
        a[i][j] = b[j][i]
      }
    }
  }
}

```

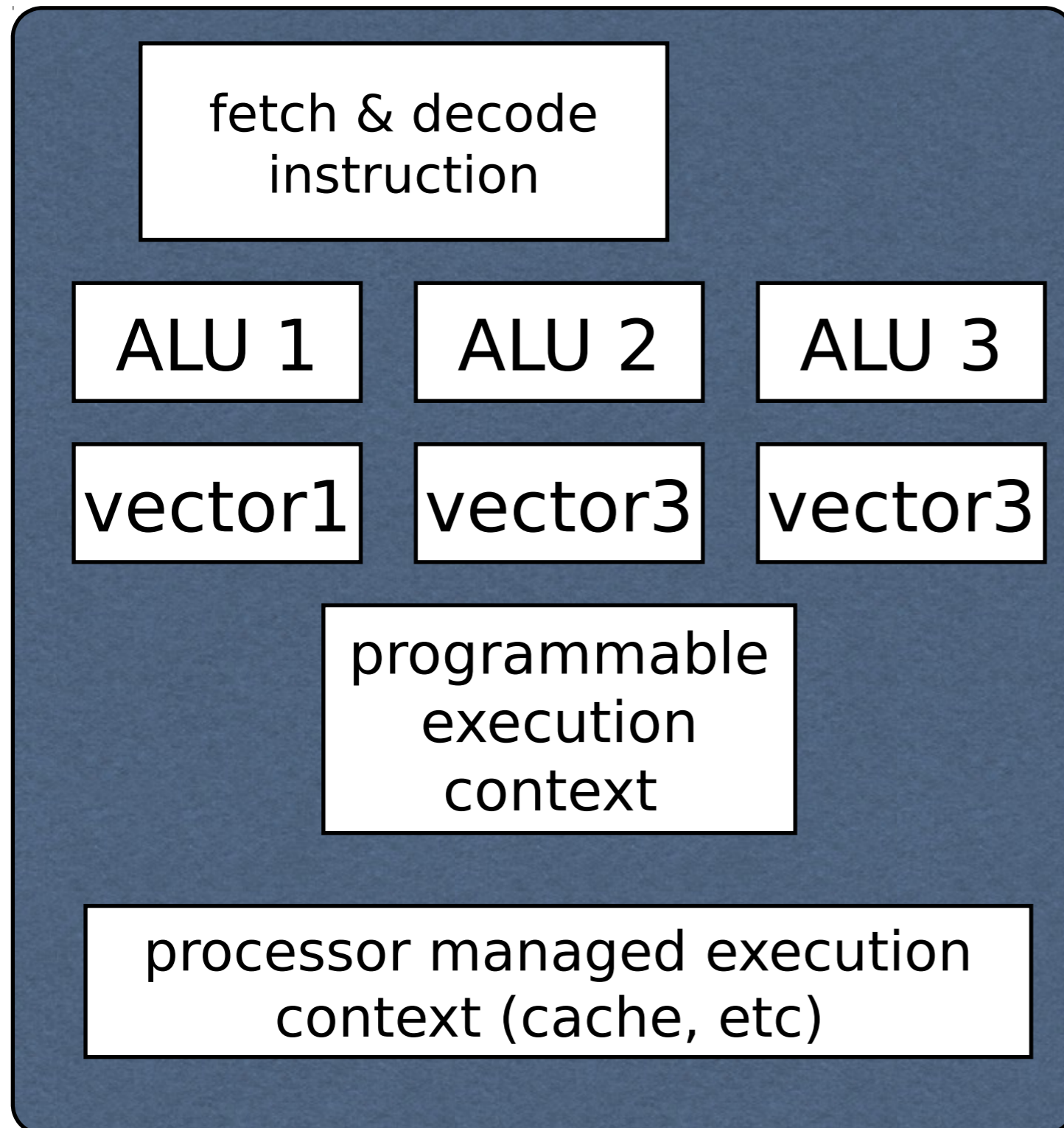
- For matrix multiply, you have $O(N^2)$ data and $O(N^3)$ operations
- Ideally, you would bring $O(N^2)$ data into cache
- Without tiling, you bring $\sim O(N^3)$ data into cache, as array elements get bounced from cache and brought back in
- Tiling reduces cache missing by a factor of N

A simple core



Consider a
simple core

A more realistic processor



Consider a more realistic core

Because programs often have multiple instructions that can execute at the same time, put in multiple ALUs to allow *instruction level parallelism (ILP)*

Average # instructions per cycle < 2 , depends on the application and the architecture.

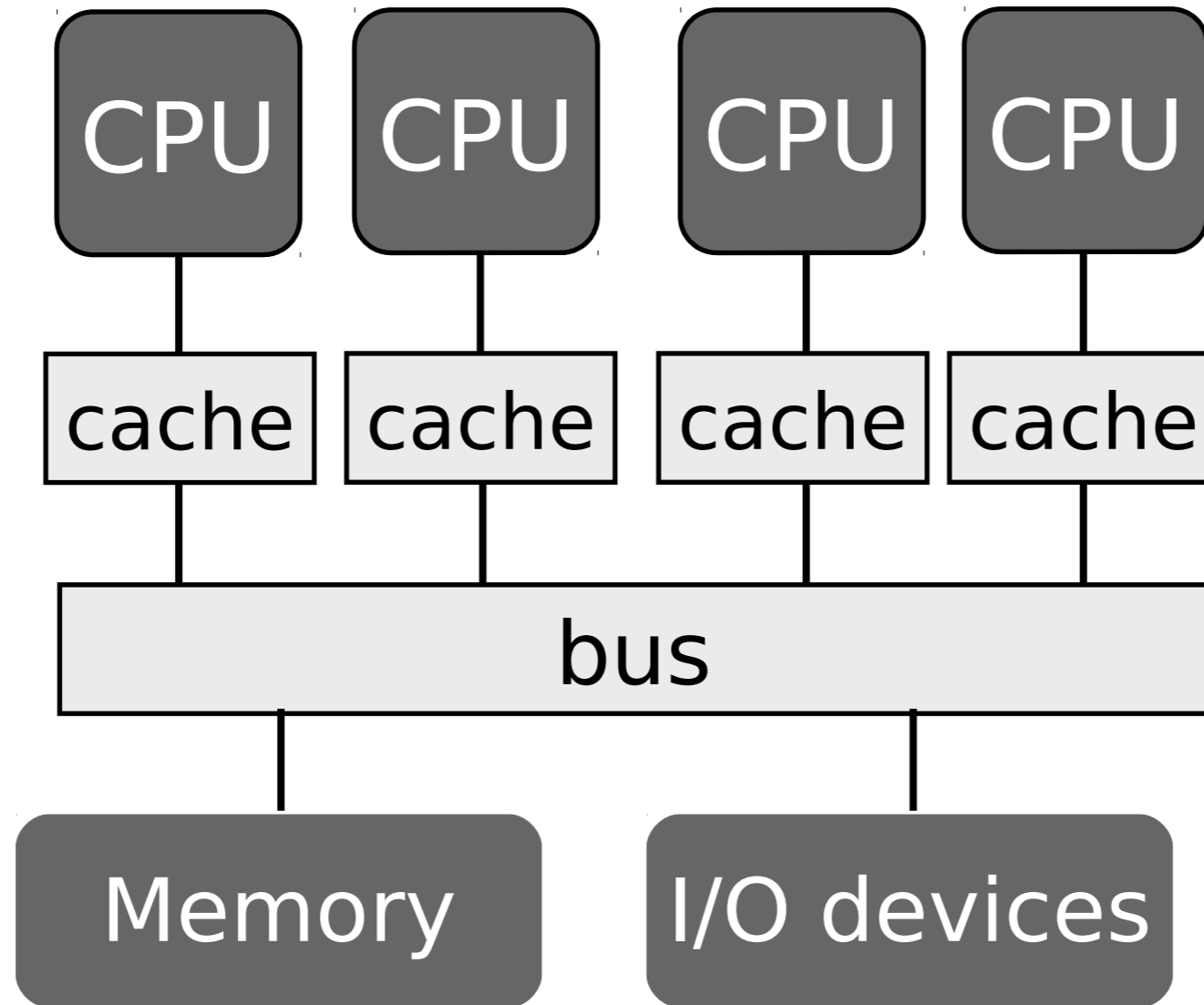
Multiprocessor

(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.

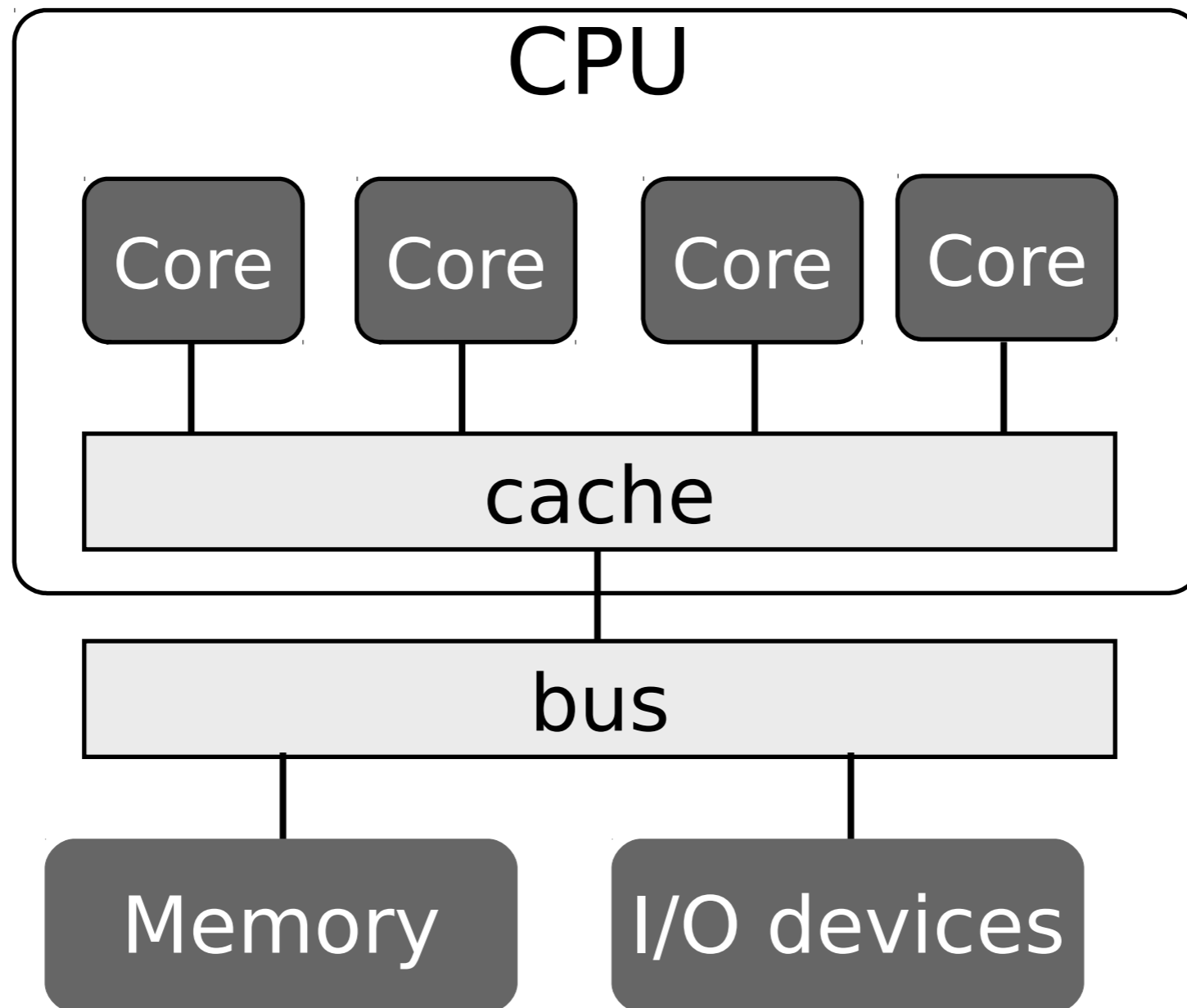
A Uniform Memory Access shared memory machine

All processors access global memory at the same speed



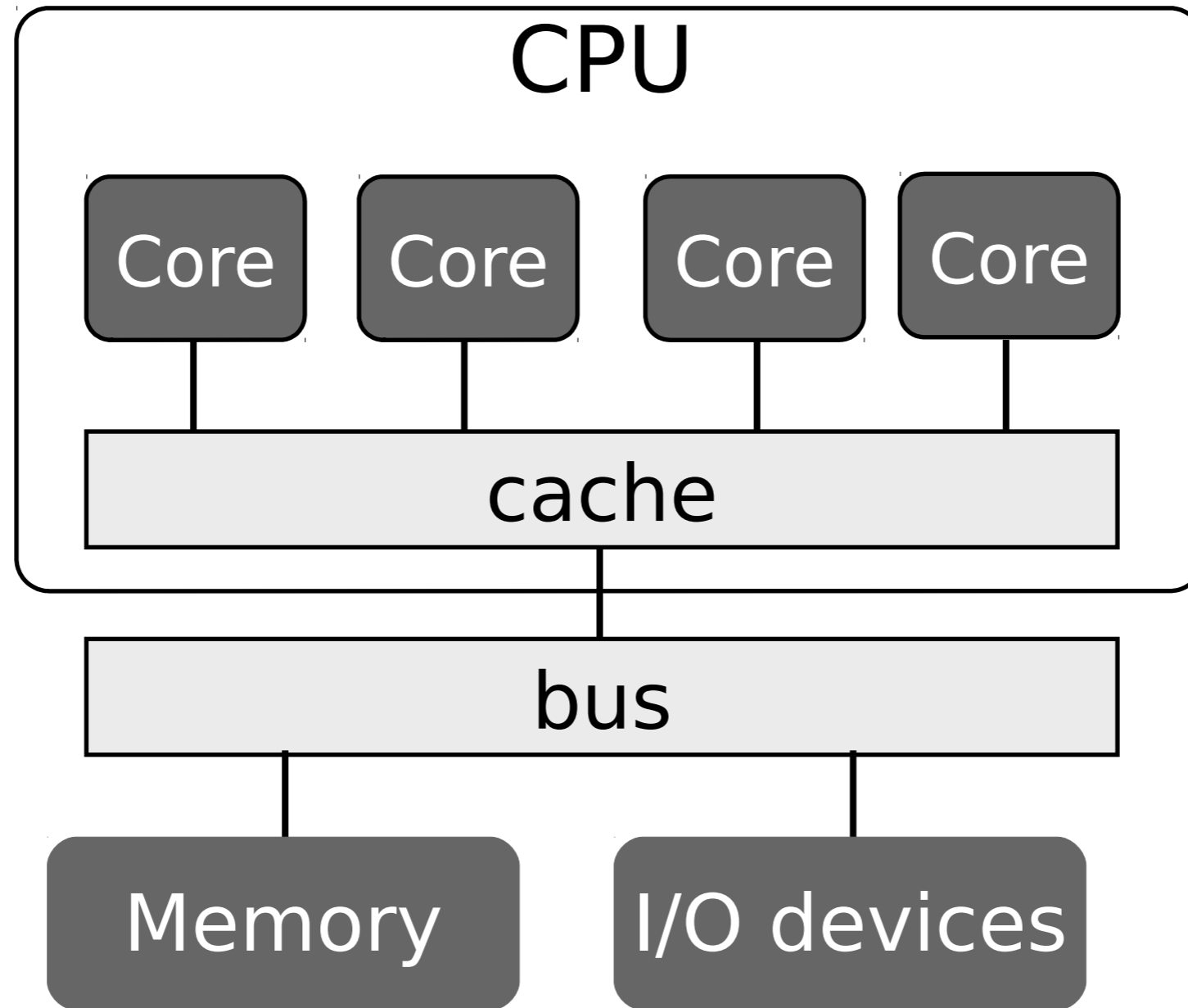
Multicore machines usually have uniform memory access

All cores access global memory at the same speed

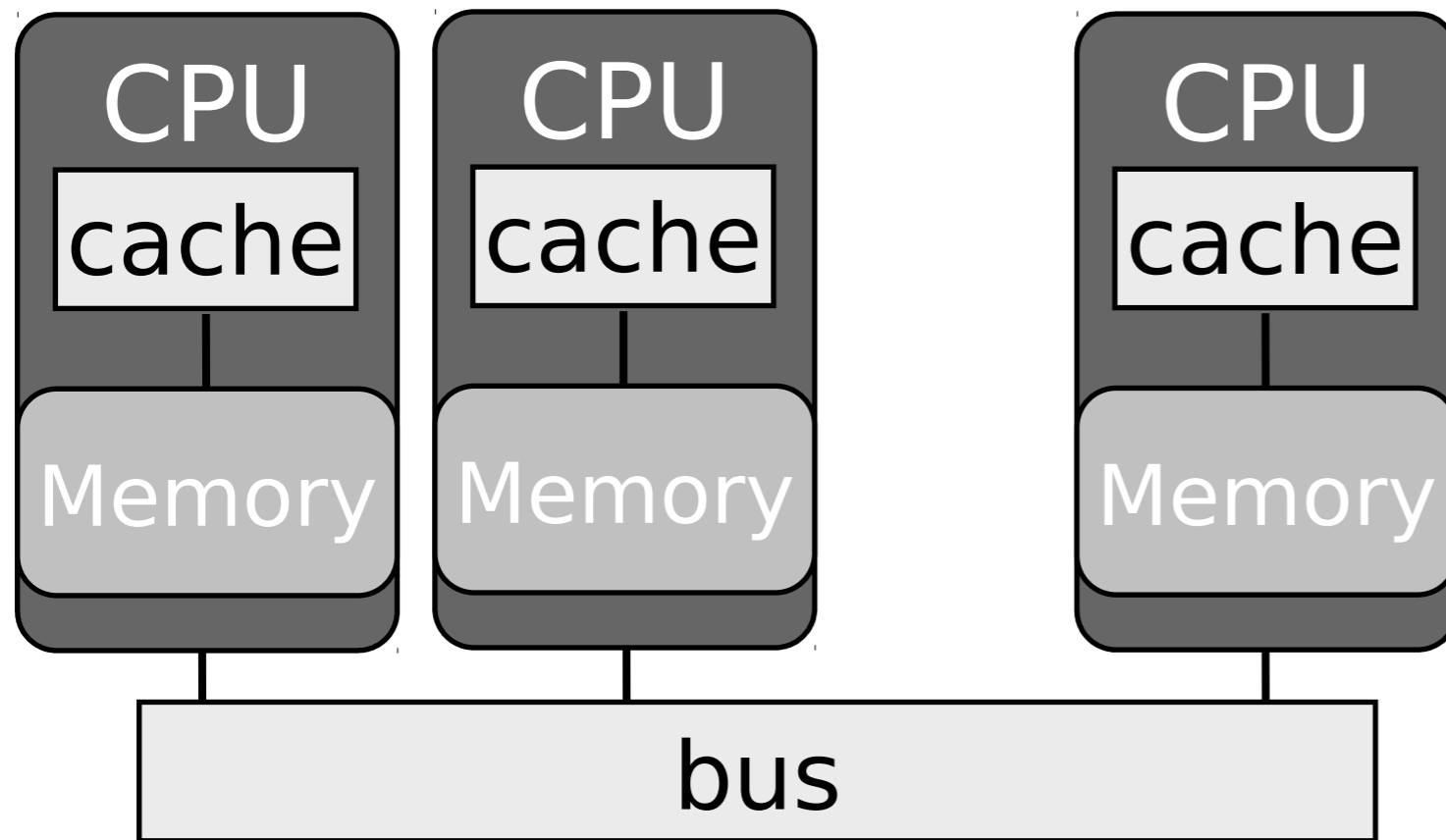


Multicore machines usually share at least one *level* of cache

All cores access global memory at the same speed



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

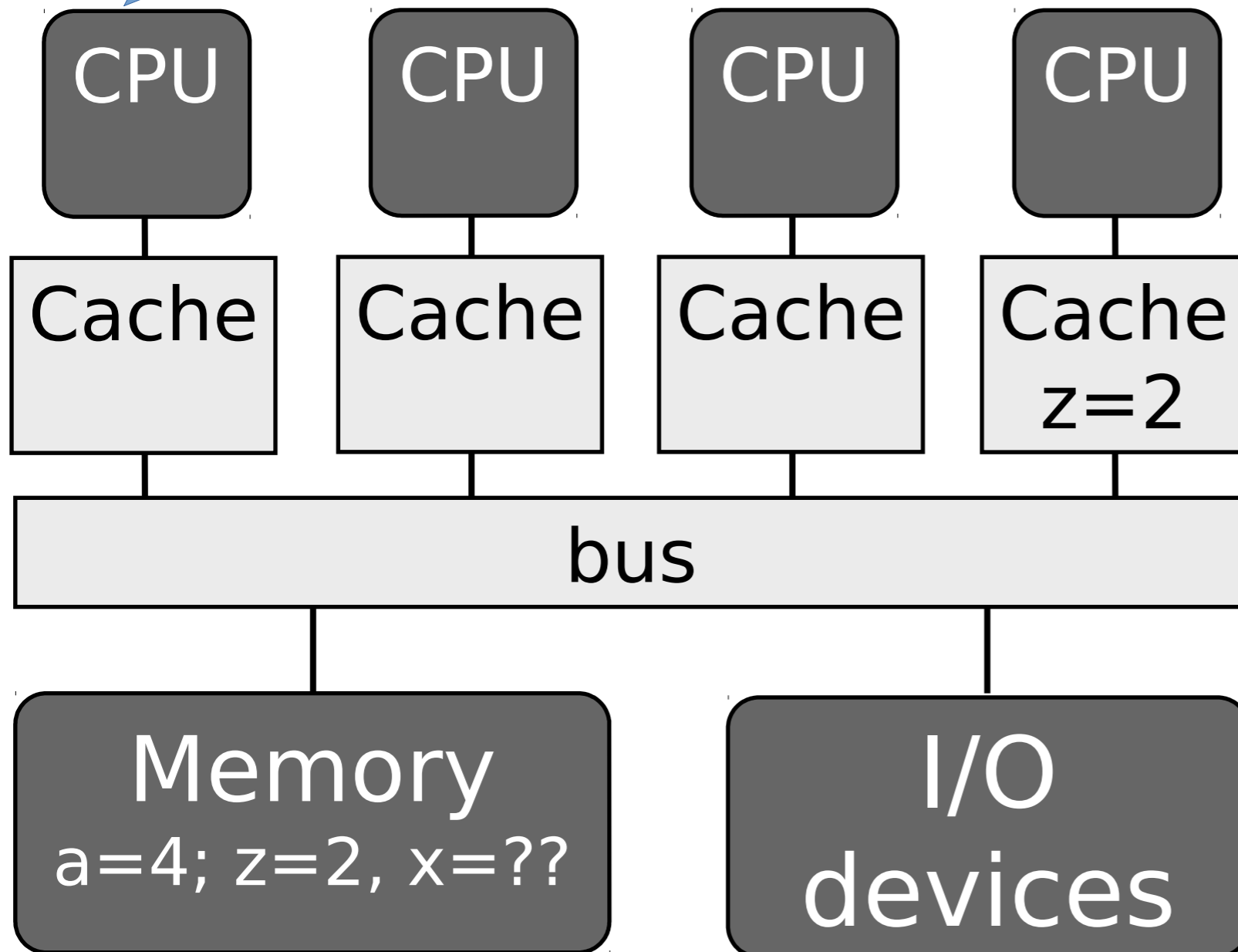


Global memory is spread across, and held in, the local memories of the different nodes of the machine

Processors will access their memory faster than their neighbors memory

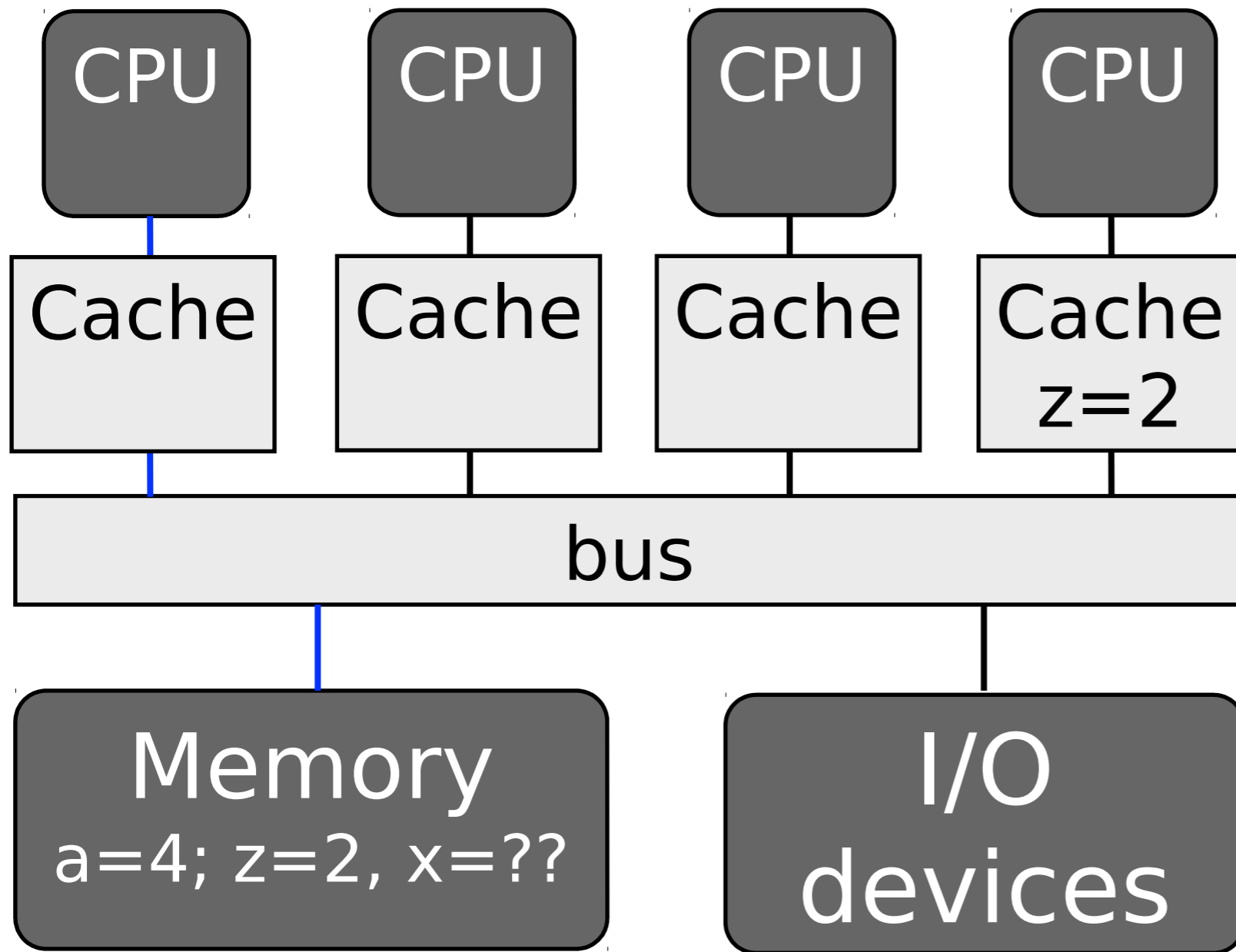
Coherence is needed

T0: $z = a$
(instruction to be executed)



T0: $z = a$

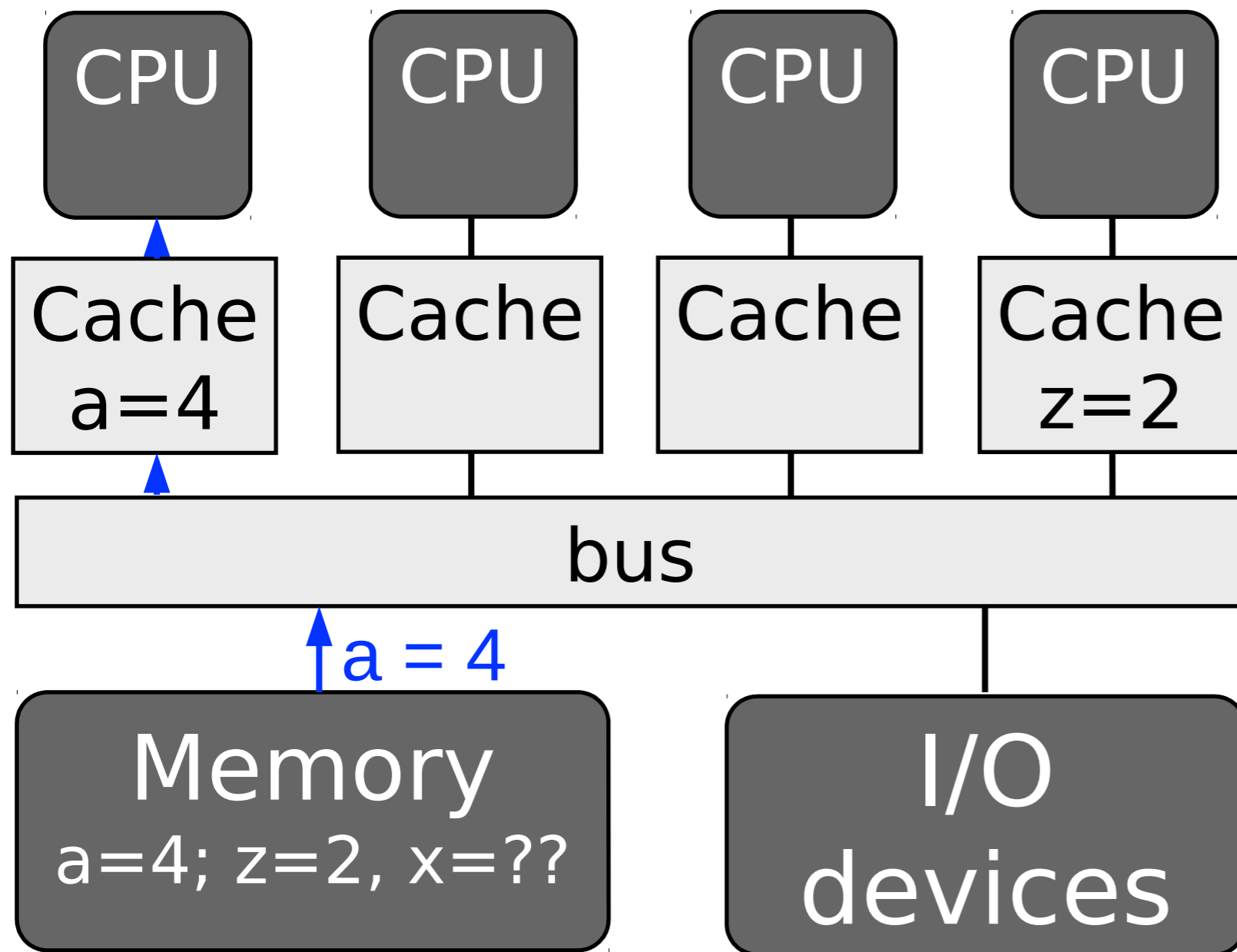
Load a from memory



T0: $z = a$

load reg2, from (a) // load a from memory

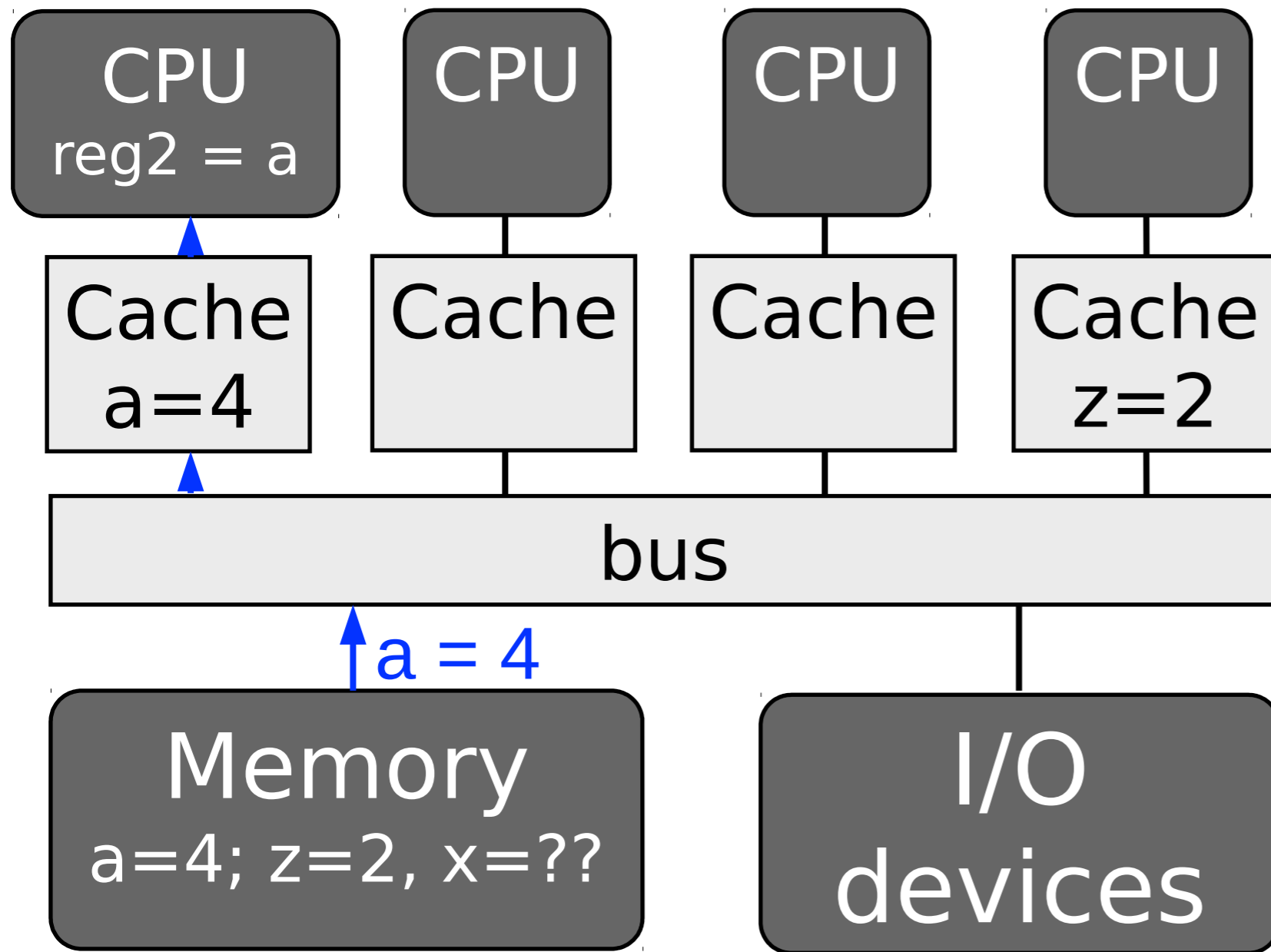
st reg2, into (z)



T0: z = a

load reg2, from (a) // load a from memory

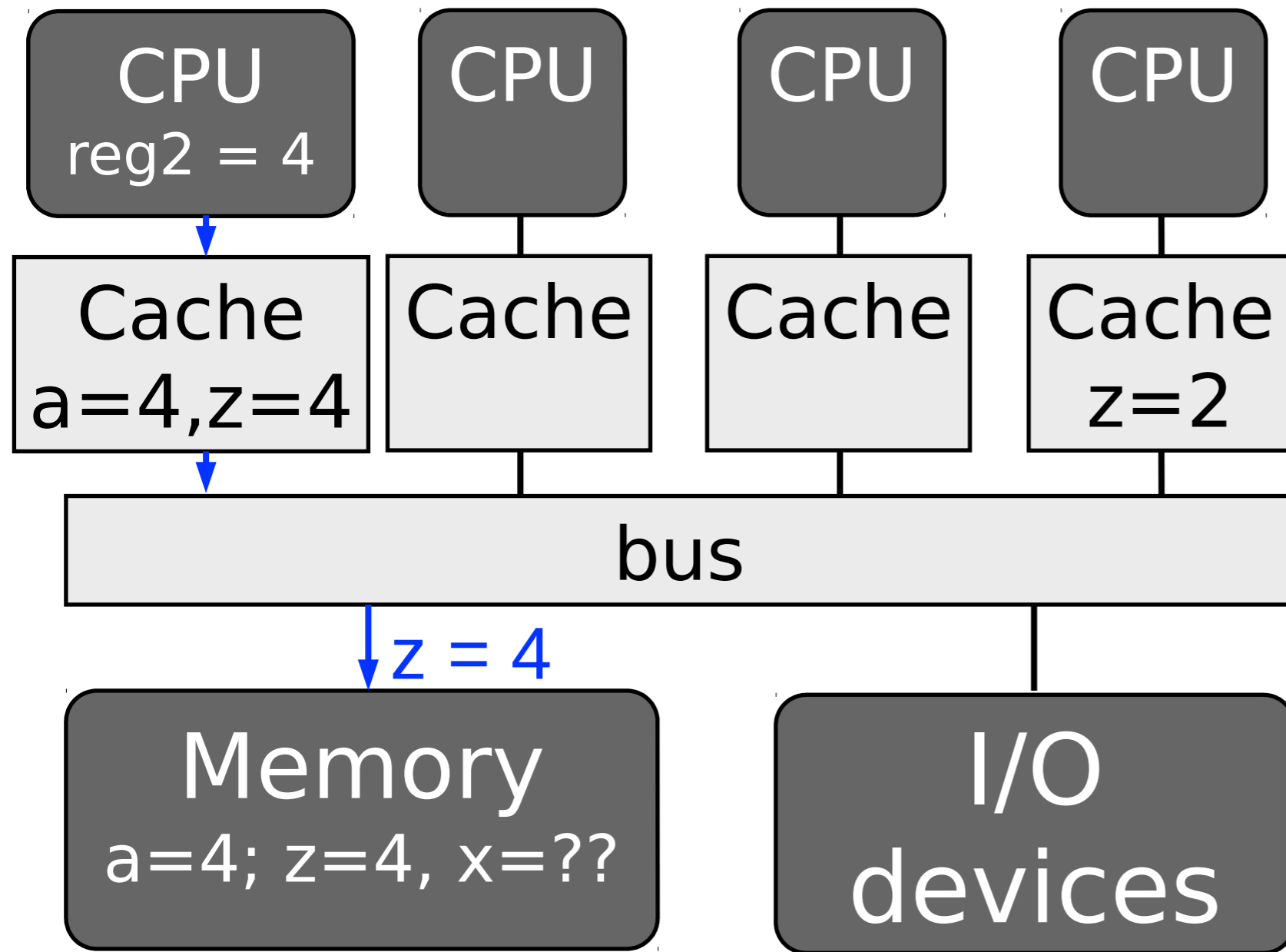
st reg2, into (z)



T0: $z = a$

load reg2, from (a) // load a from memory

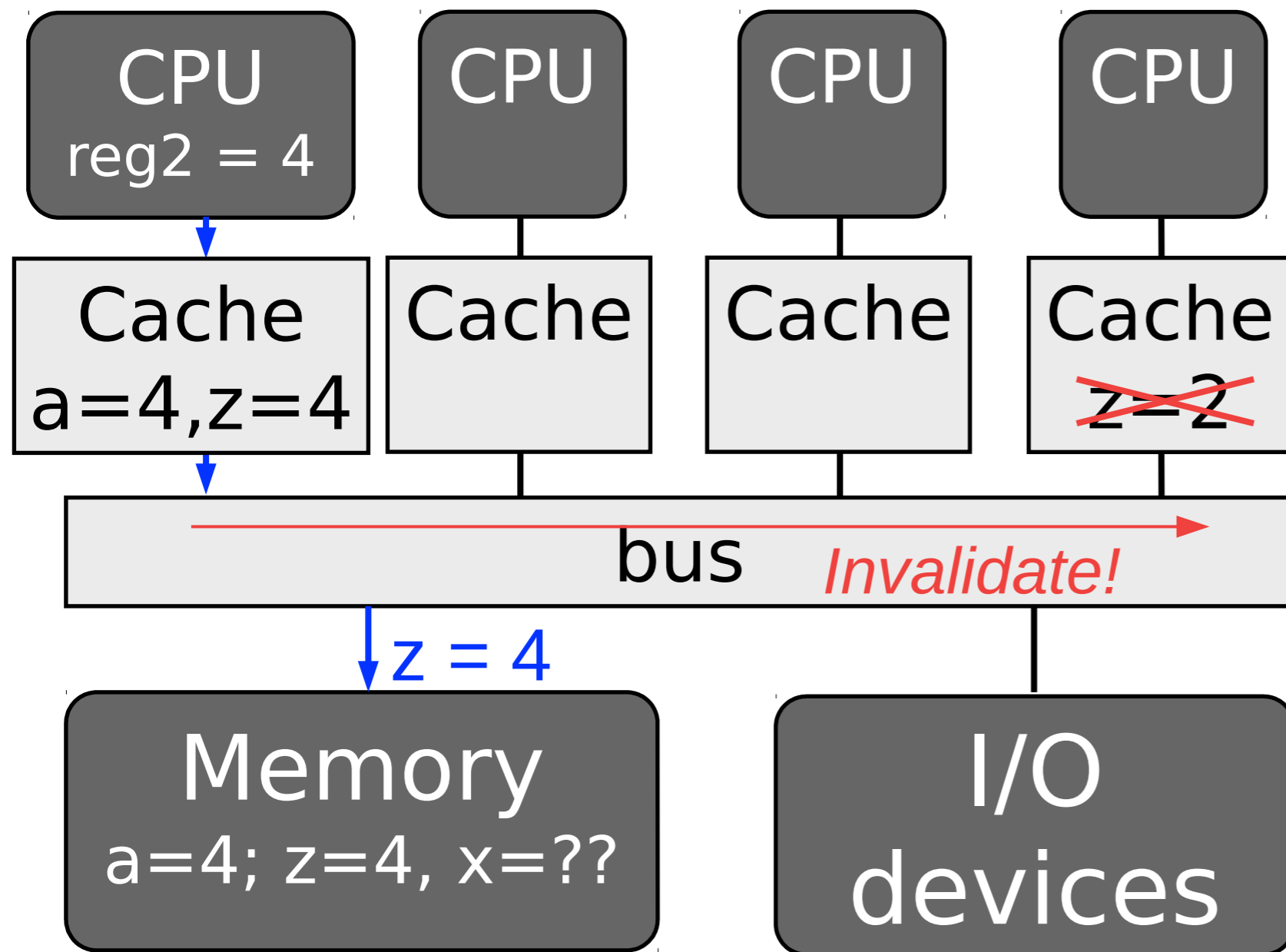
st reg2, into (z) // reg2



T0: $z = a$

load reg2, from (a) // load a from memory

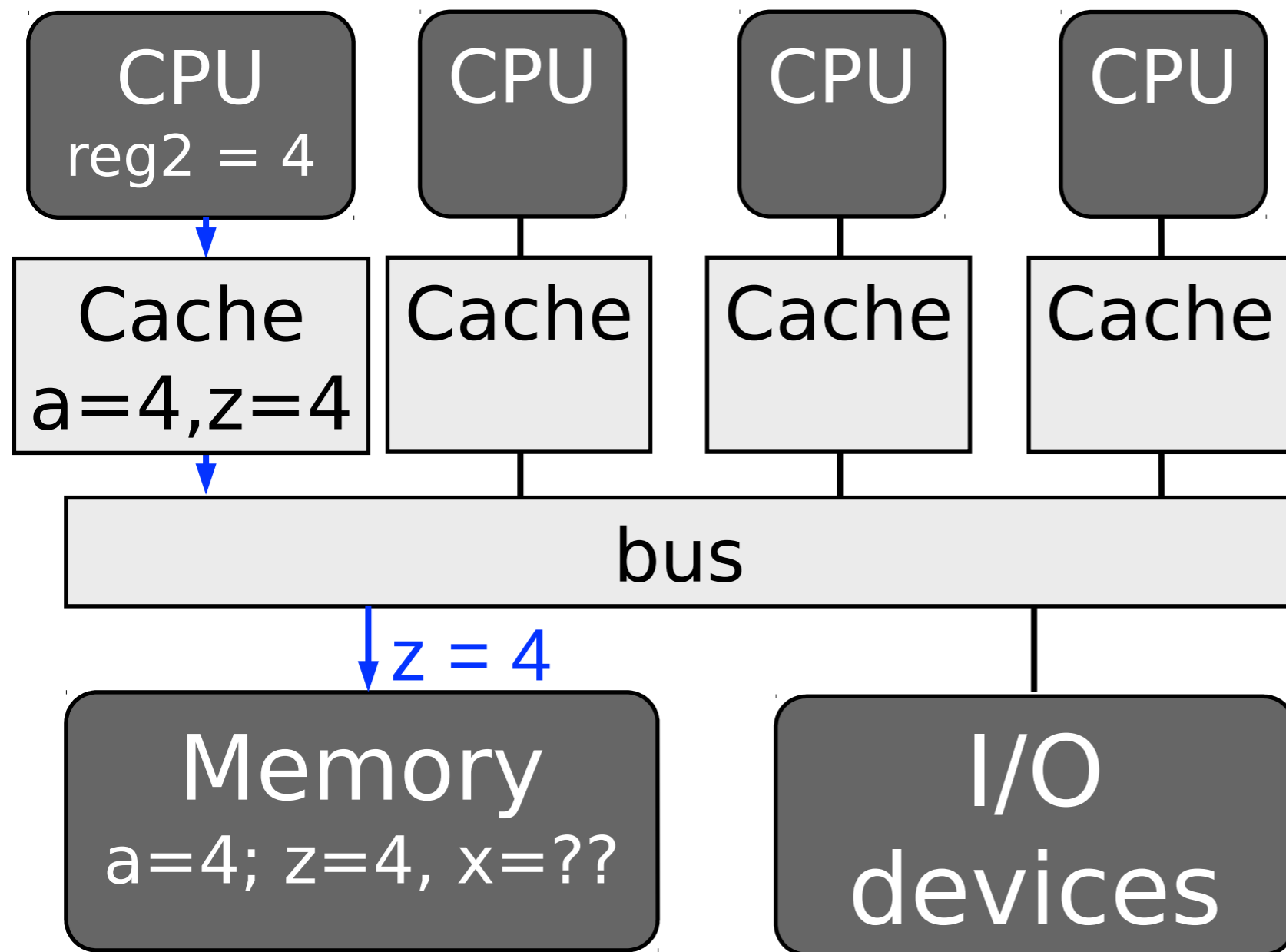
st reg2, into (z) // reg2



T0: $z = a$

load reg2, from (a) // load a from memory

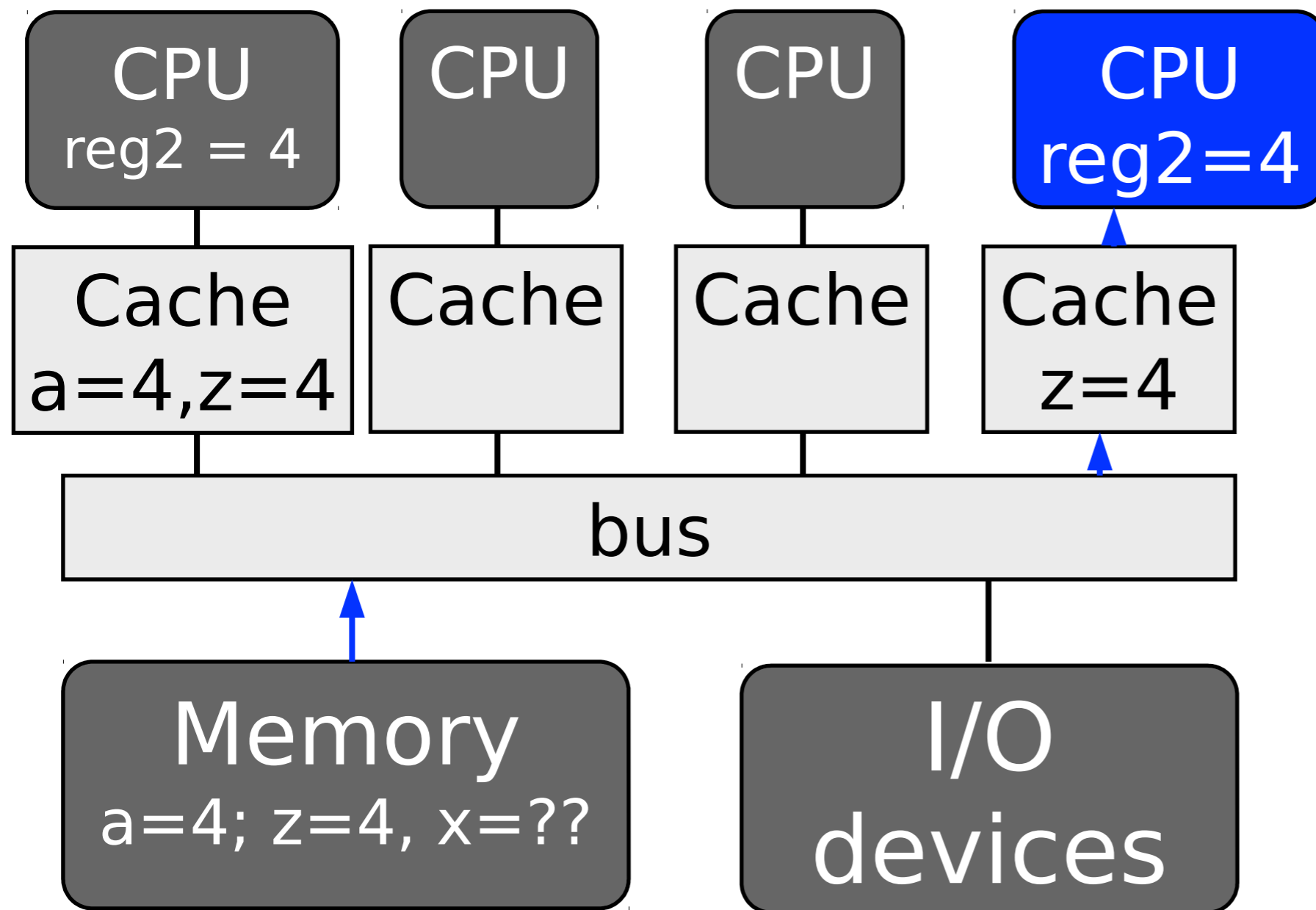
st reg2, into (z) // reg2



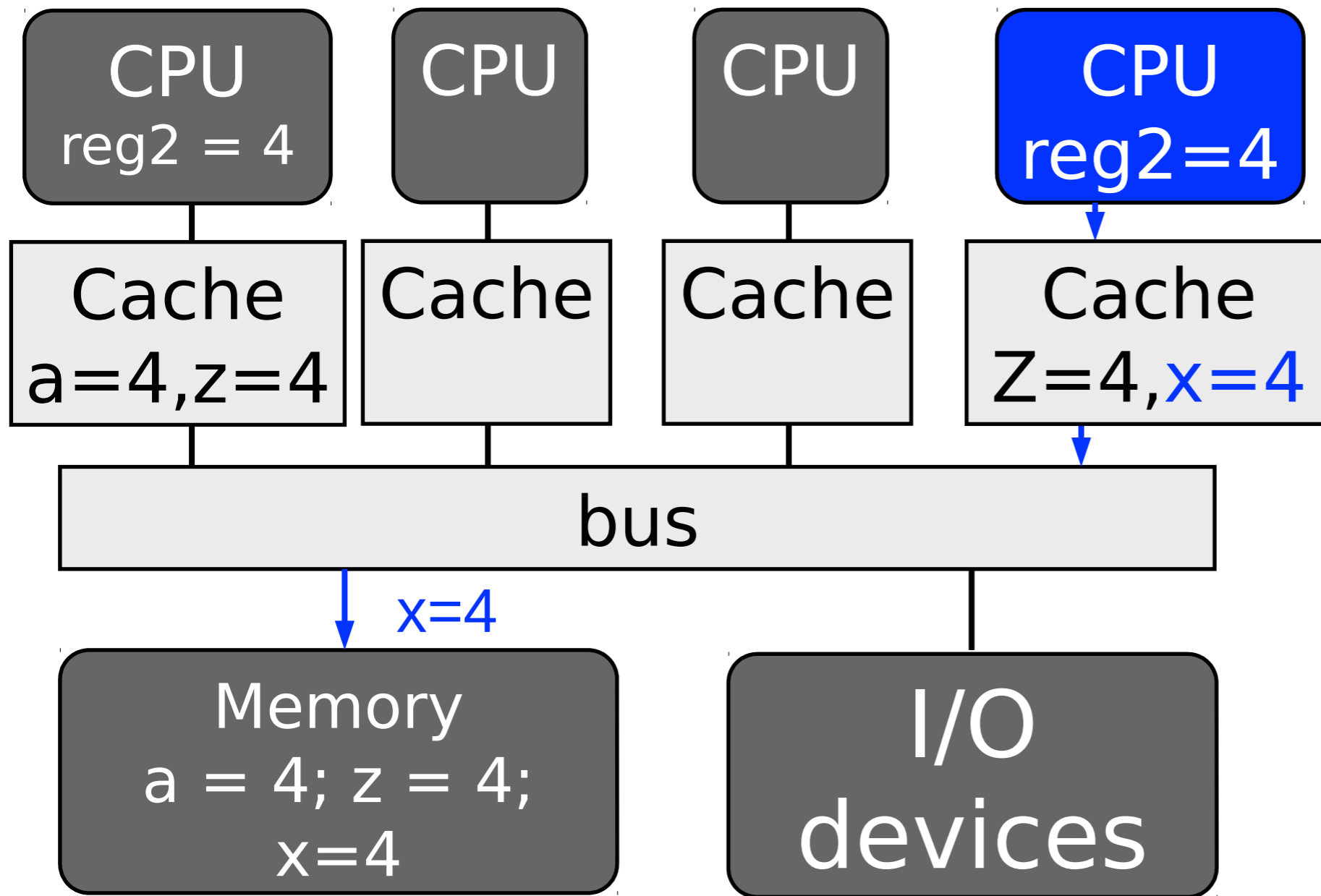
Tn: $x = z$ // instruction in rightmost CPU

load reg2, from (z)

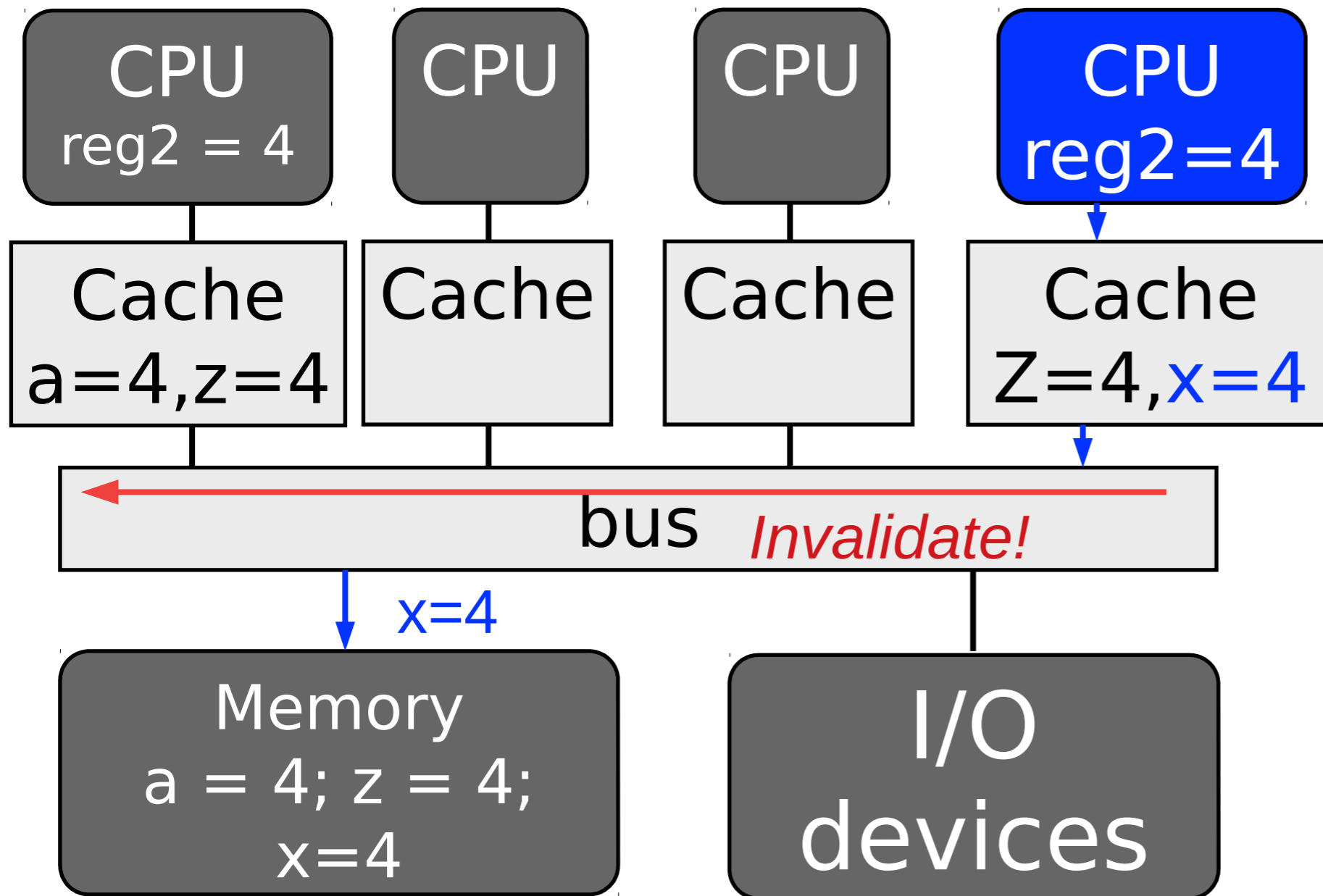
st reg2, into (x)



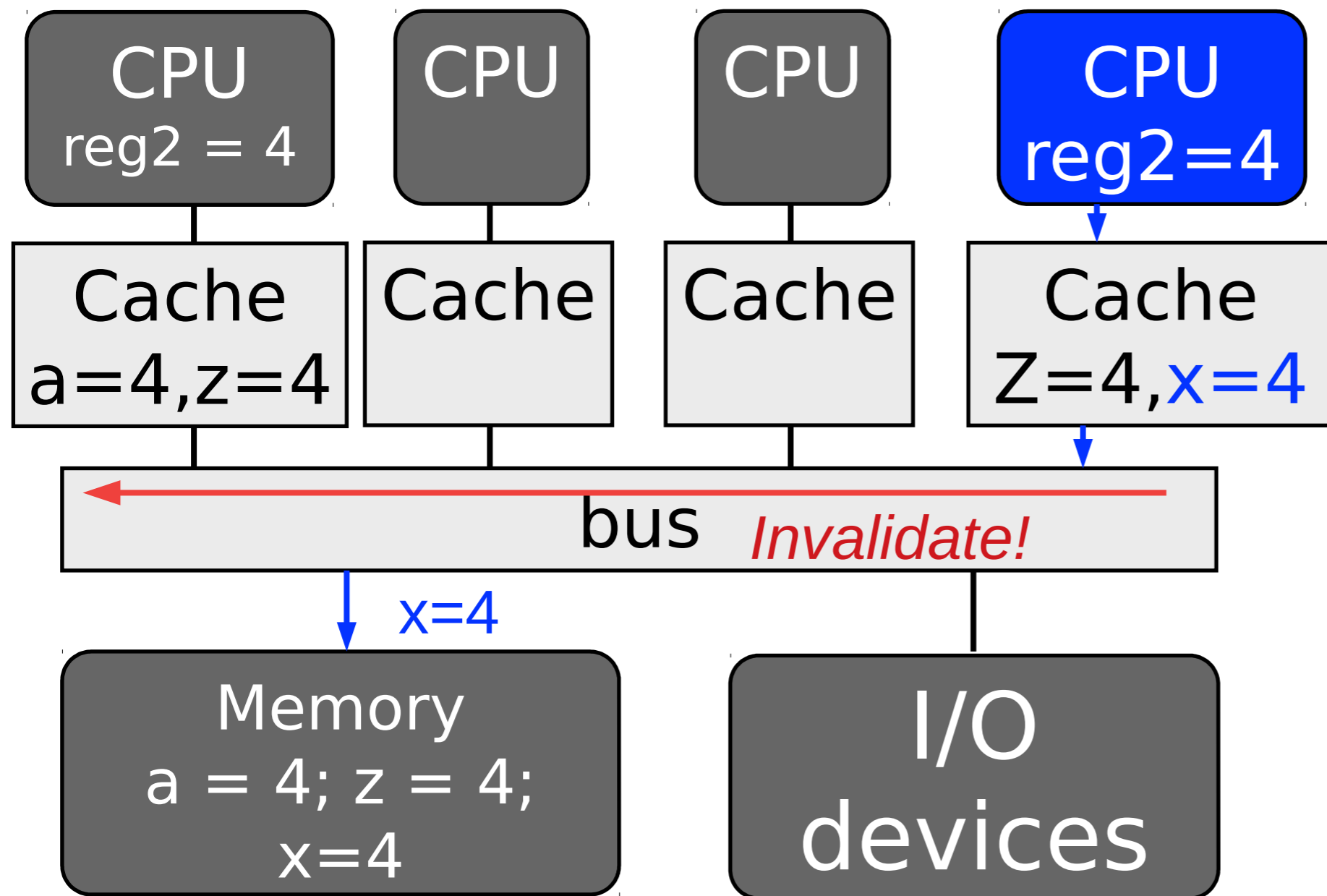
Tn: `x = z` // instruction in rightmost CPU
load reg2, from (z)
`st reg2, into (x)`



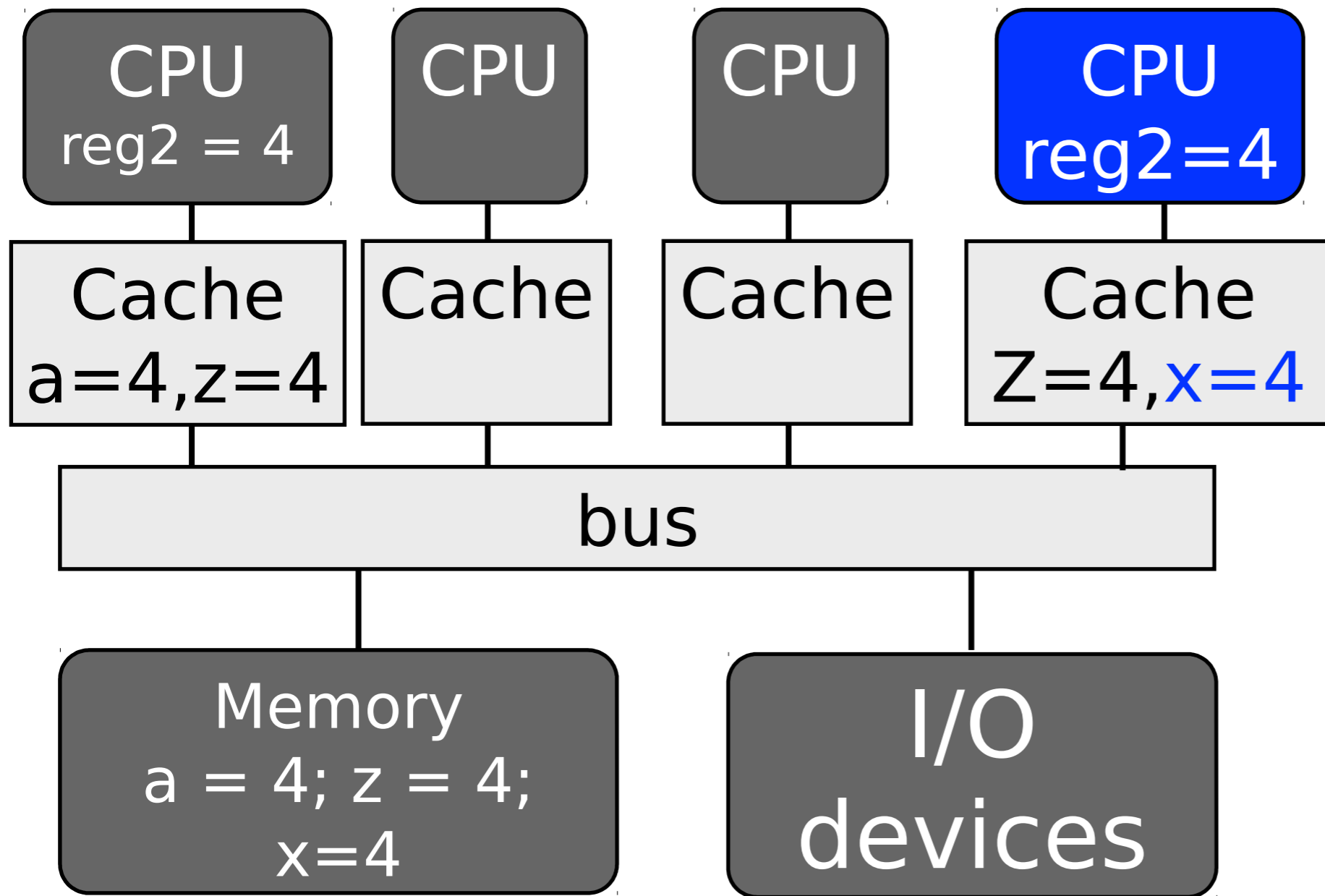
Tn: `x = z` // instruction in rightmost CPU
load reg2, from (z)
`st reg2, into (x)`



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



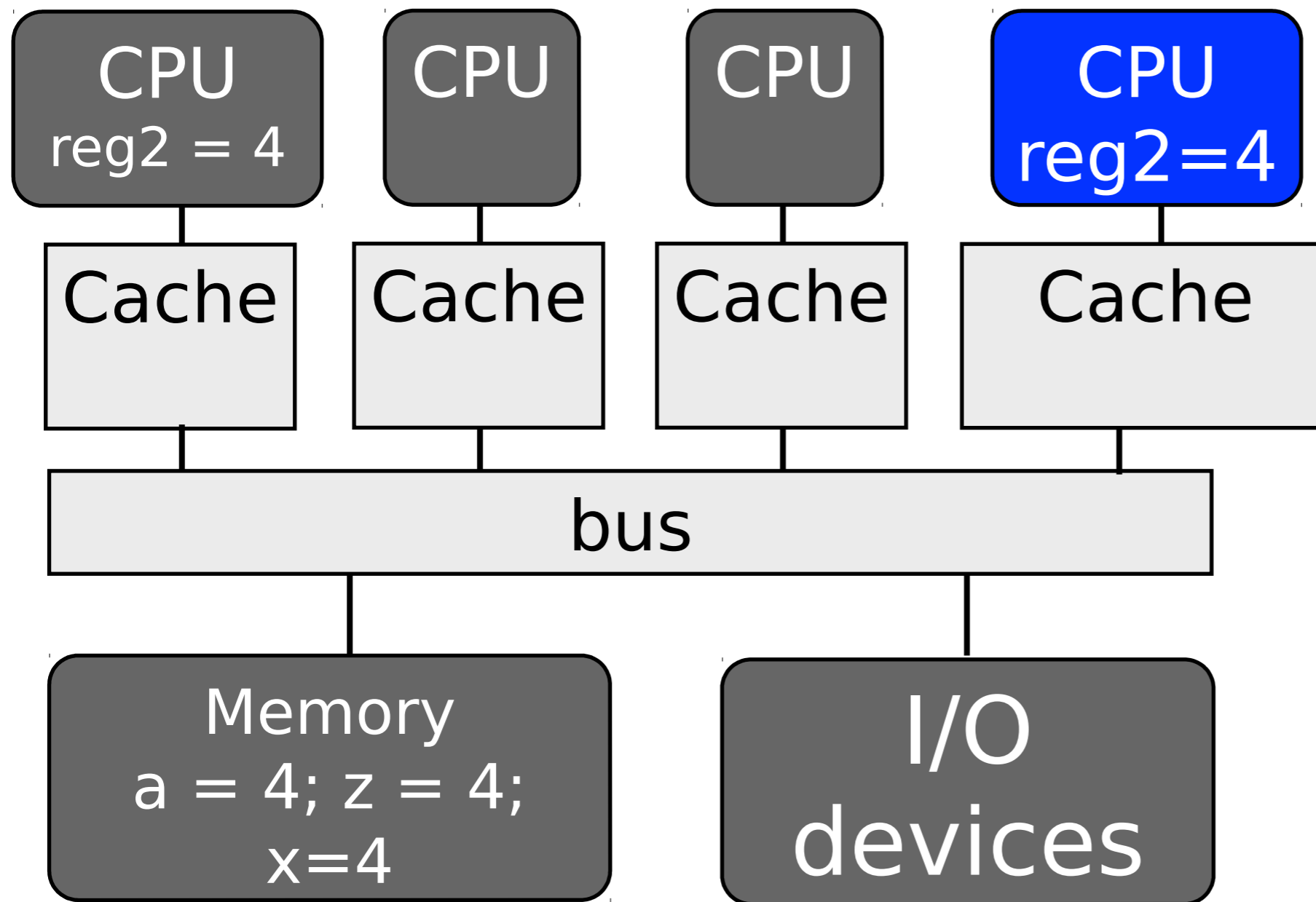
Hardware makes sure a core/processor reads the latest value assigned to memory
(cache coherence)



What if $x=z$
executes before
 $z=a$?

What happens if
 z at $x=z$ is
loaded while
store of z is still
in progress?

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



Does $x=z$ or
 $z=4$ execute
first?

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.*

Sequential Consistent (SC) executions

Parallel stream 0

Instruction 1

Instruction 2

Instruction 3

Parallel stream 1

Instruction 4

Instruction 5

Instruction 6

SC

Instruction 1

Instruction 2

Instruction 3

Instruction 4

Instruction 5

Instruction 6

SC

Instruction 1

Instruction 4

Instruction 2

Instruction 5

Instruction 3

Instruction 6

SC

Instruction 4

Instruction 5

Instruction 6

Instruction 1

Instruction 2

Instruction 3

NOT SC

Instruction 1

Instruction 2

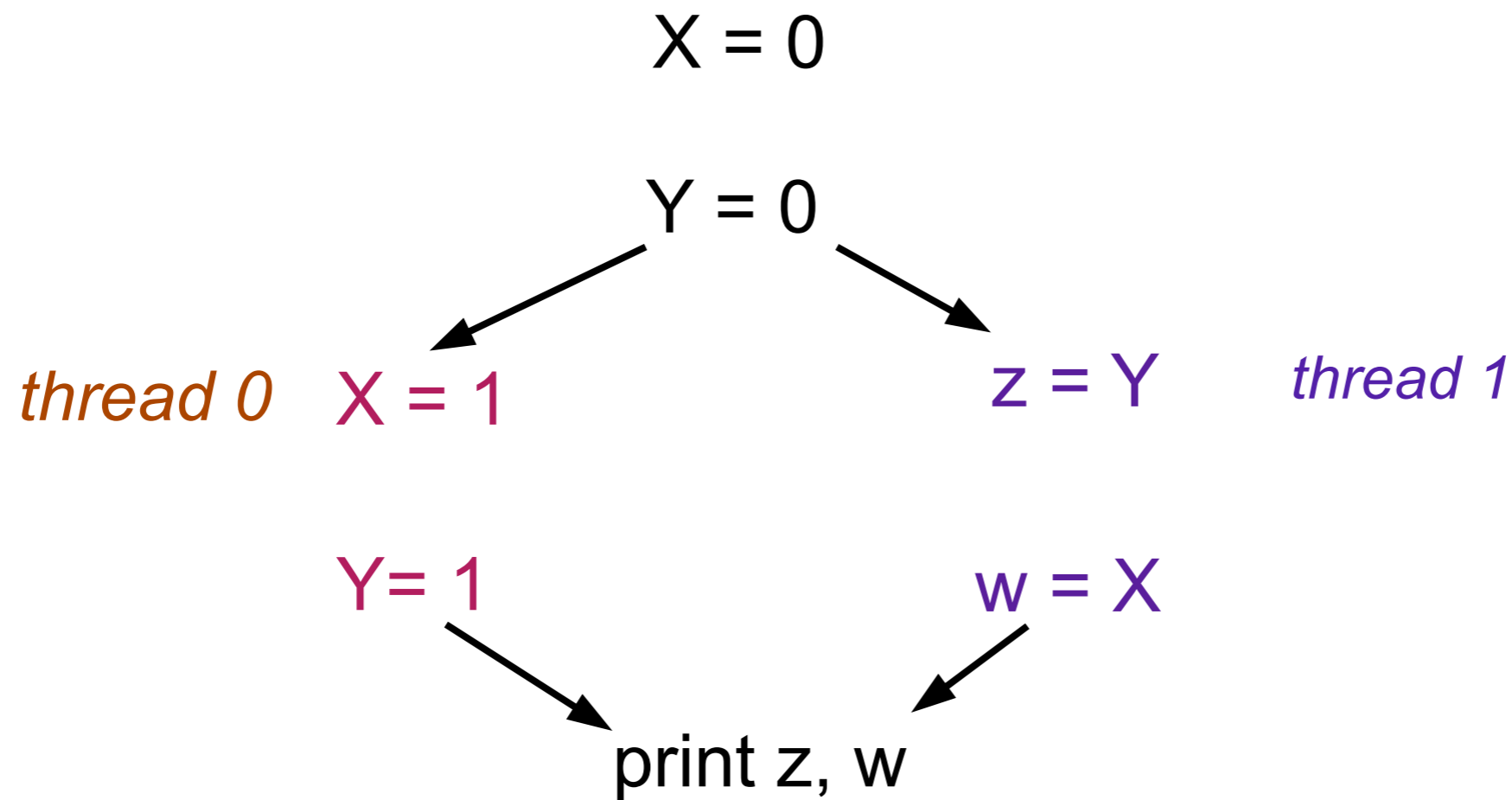
Instruction 5

Instruction 3

Instruction 4

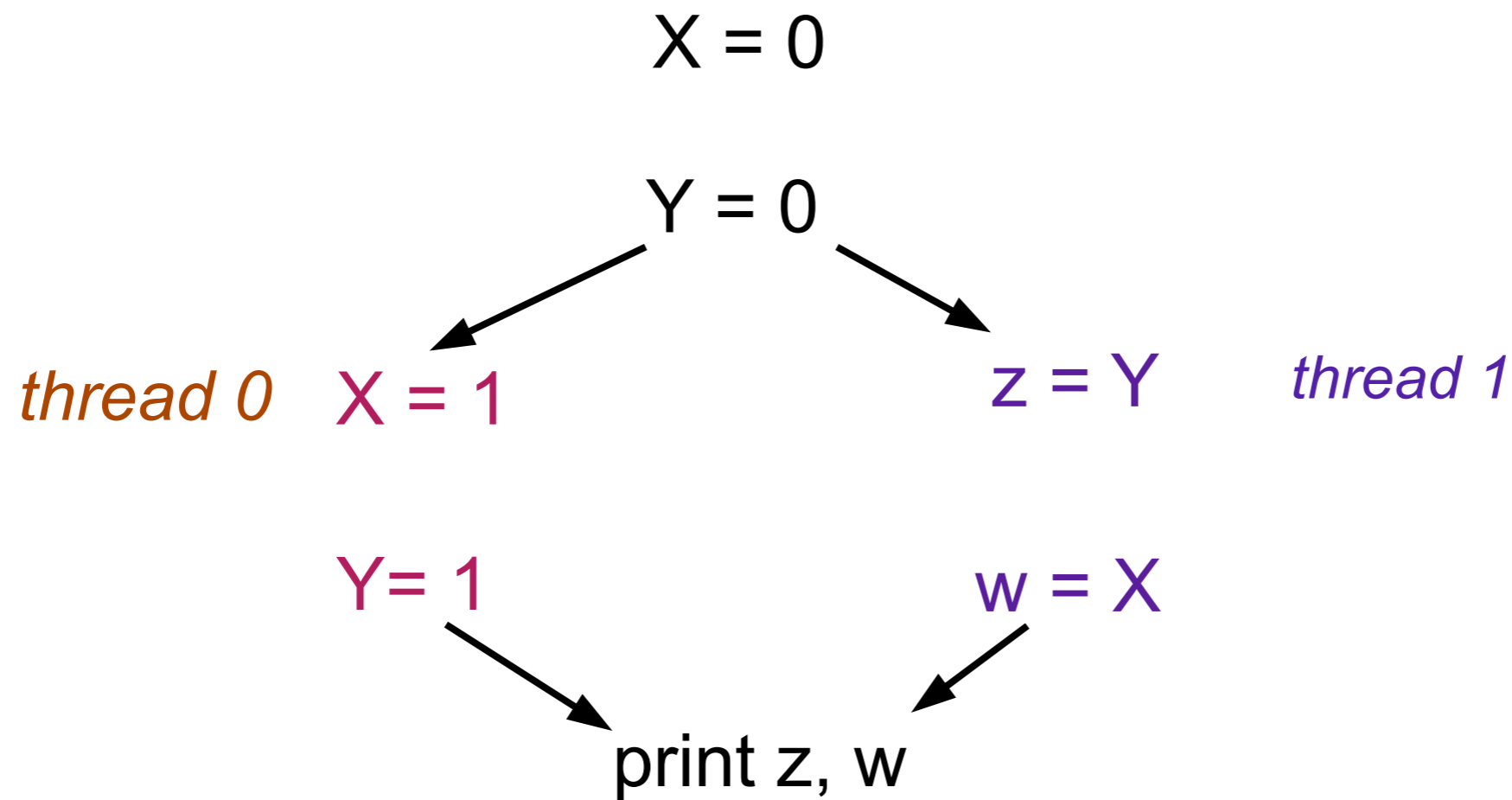
Instruction 6

SC Example



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

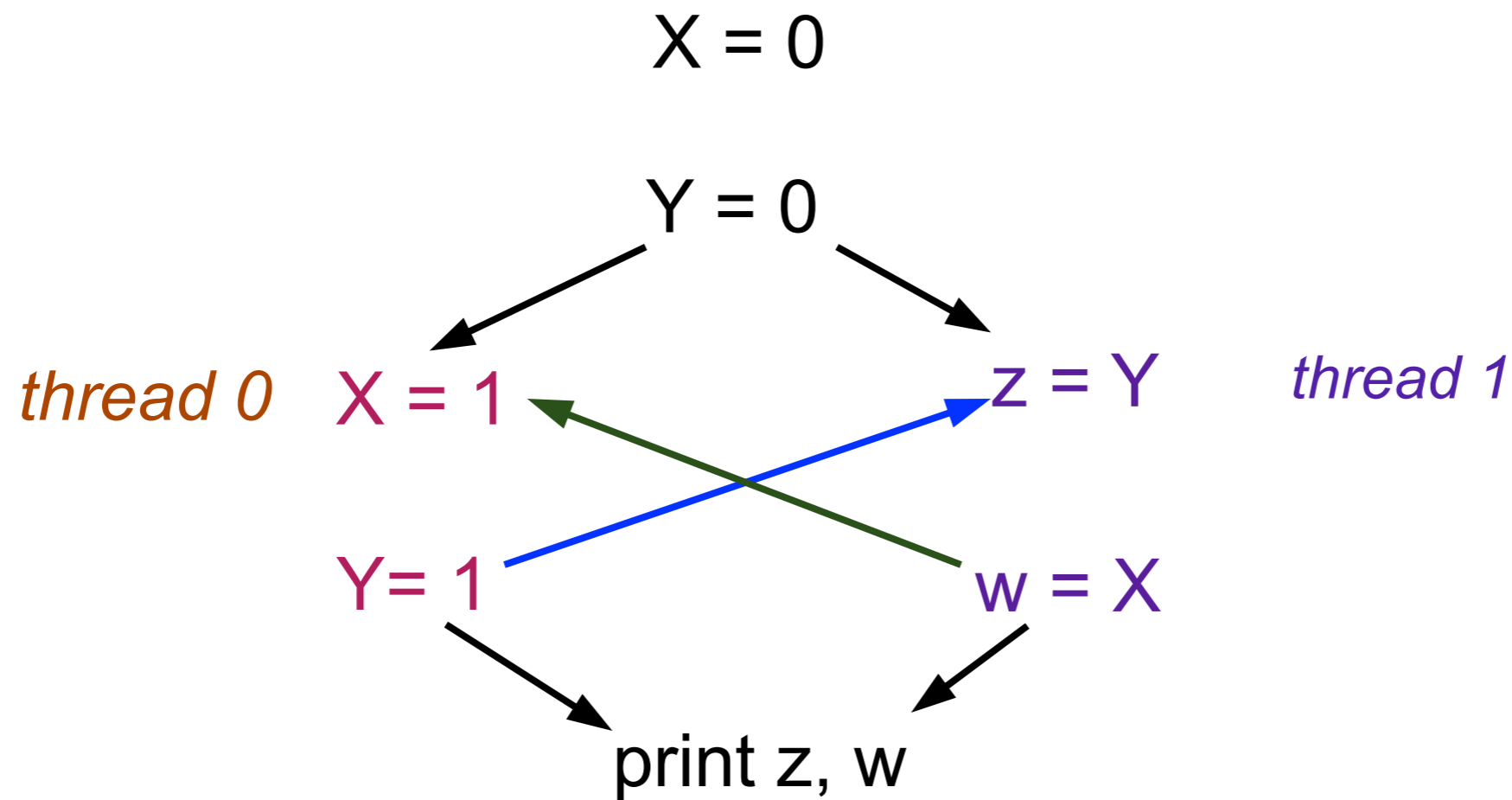
SC Example



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

Answer: Not with sequential consistency

SC Example



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

For $z == 1$, “ $Y=1$ ” must execute before “ $z=Y$ ”

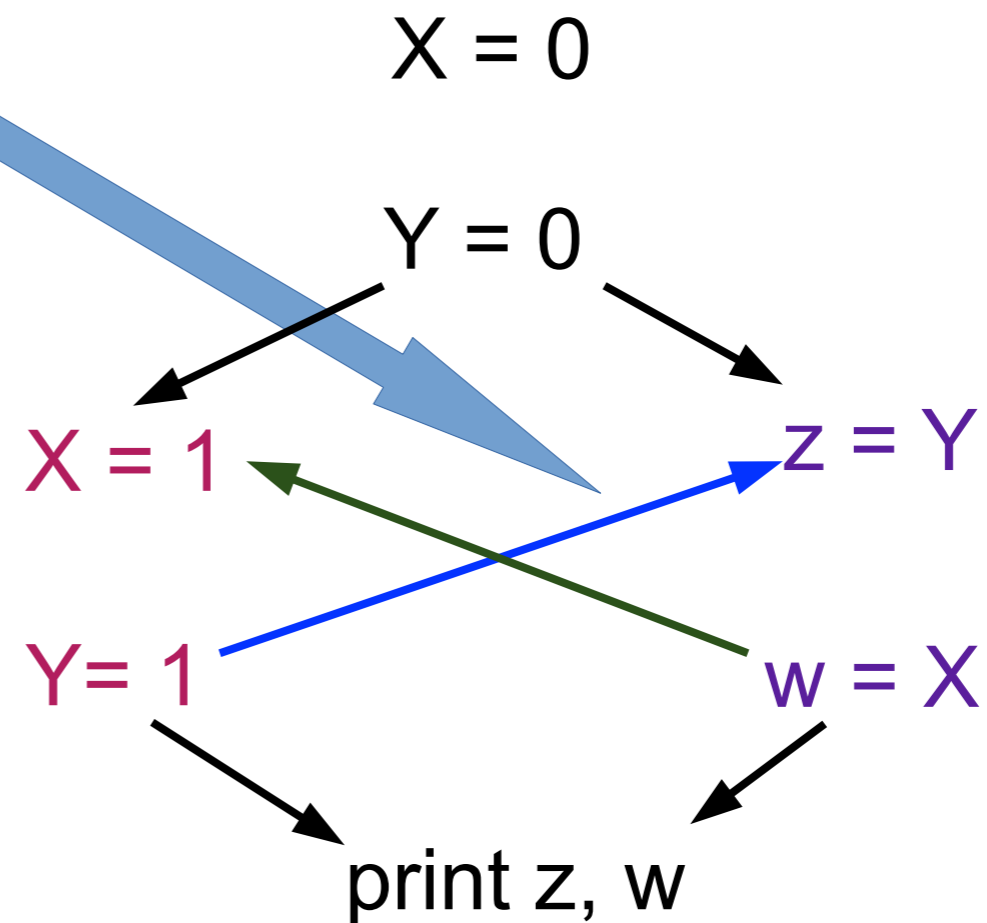
For $w == 0$, “ $w = X$ ” must execute before “ $X=1$ ”

Sequential Consistency

Relative execution order implied by assigned value

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

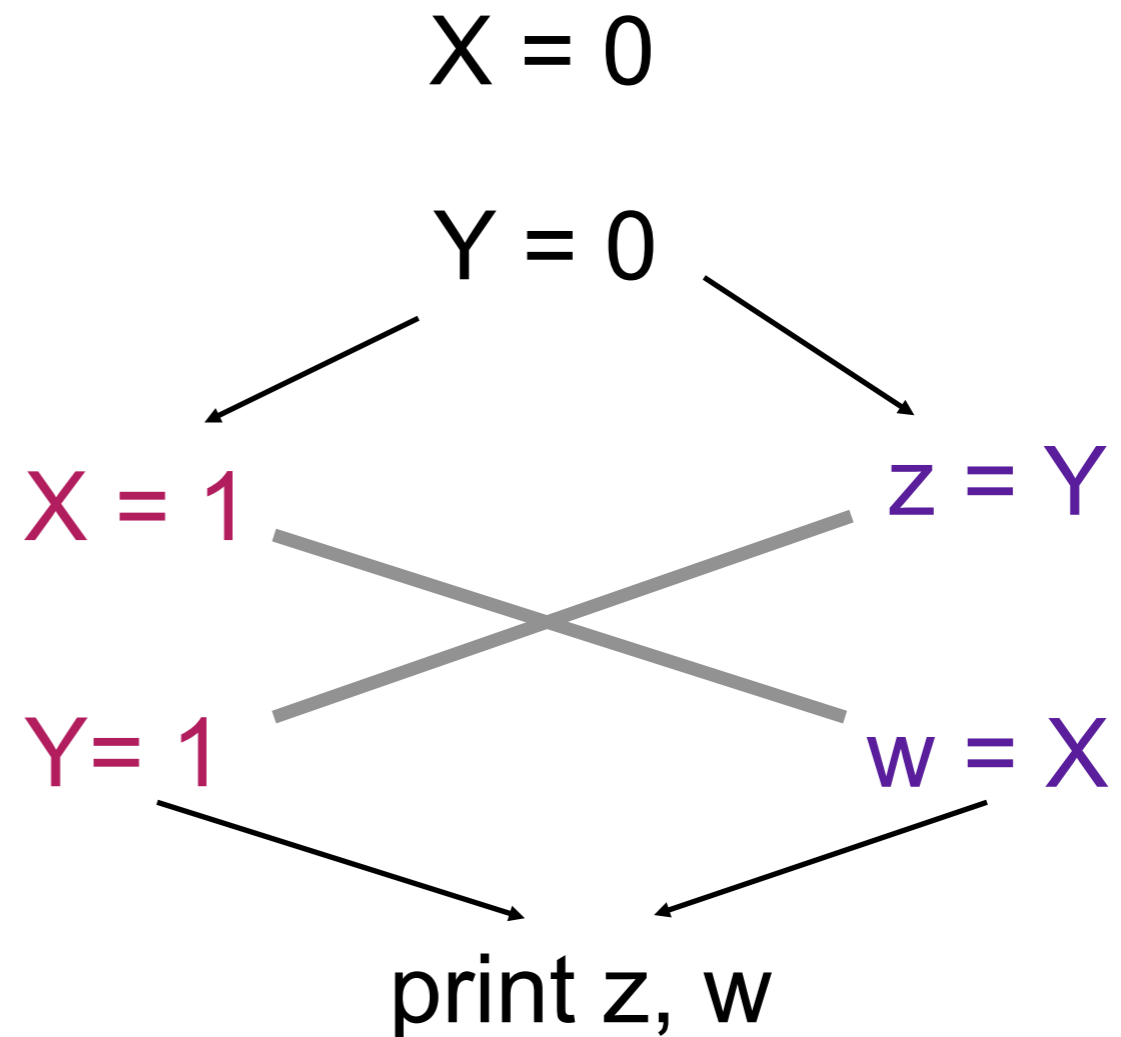
Answer: **NO**. For $z == 1$ and $w == 0$, ordering in previous slide requires either “ $X=1$ ” and “ $Y=1$ ” to execute in a different order, or for “ $z=Y$ ” or “ $w=X$ ” to execute in a different order.



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. This will be an illegal program in C/C++/Fortran. We’ll discuss the reasons for this later.



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, i.e., execution orders as seen in different threads are consistent with some definition of how orders should occur
- 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

We generally want programs to be SC

- It will be our responsibility as programmers to ensure this -- the hardware and software will not
- Hardware maintains coherence -- values read from a cache or memory will be the last value written
- Hardware typically maintains relaxed consistency -- within code running on a single thread, read orders with respect to writes *for a single variable* are maintained, write orders with respect to writes, *for a single variable*, are maintained.
- Instructions are provided to prevent re-orderings of other operations

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 *Multi-Processor*
- 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.

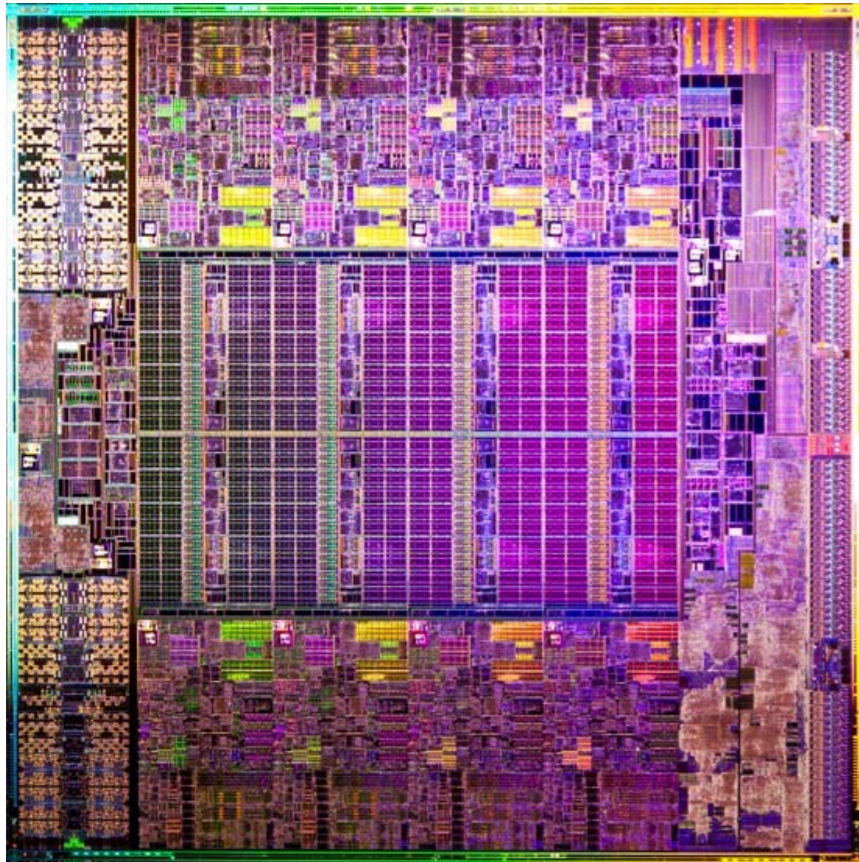
What executes in parallel?

```
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



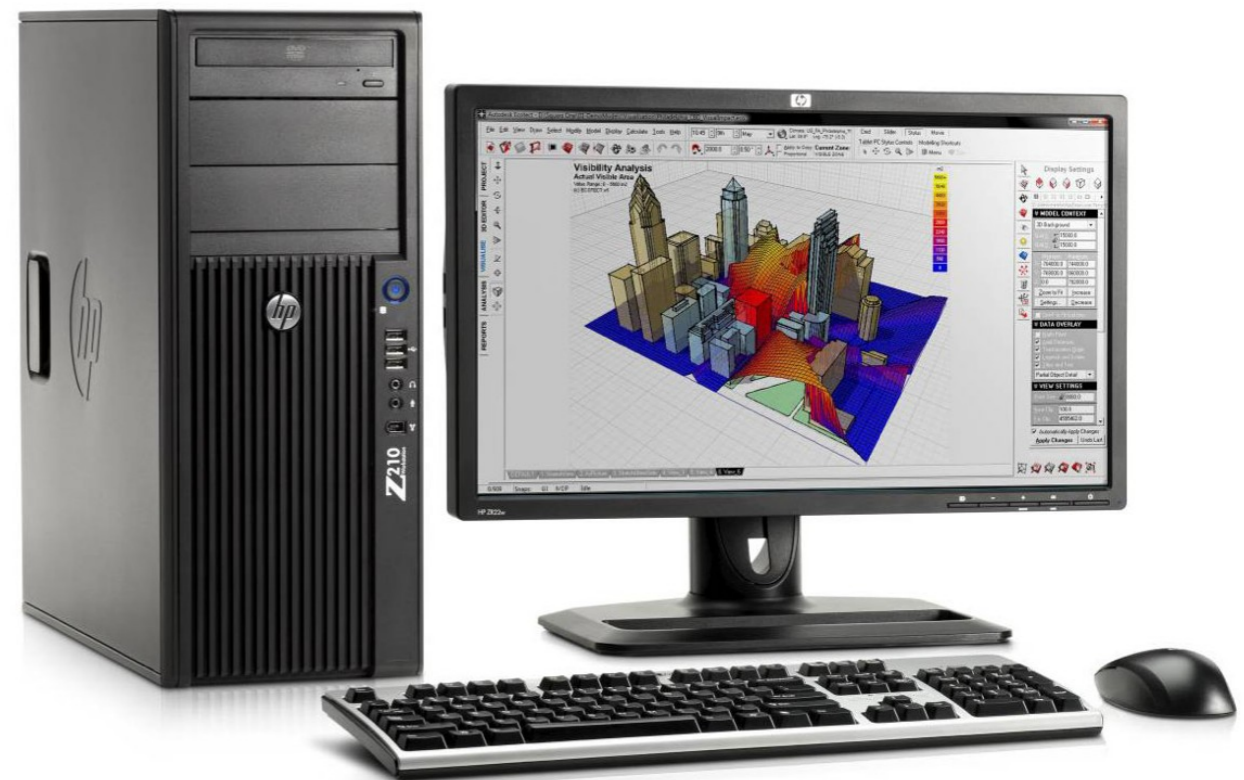
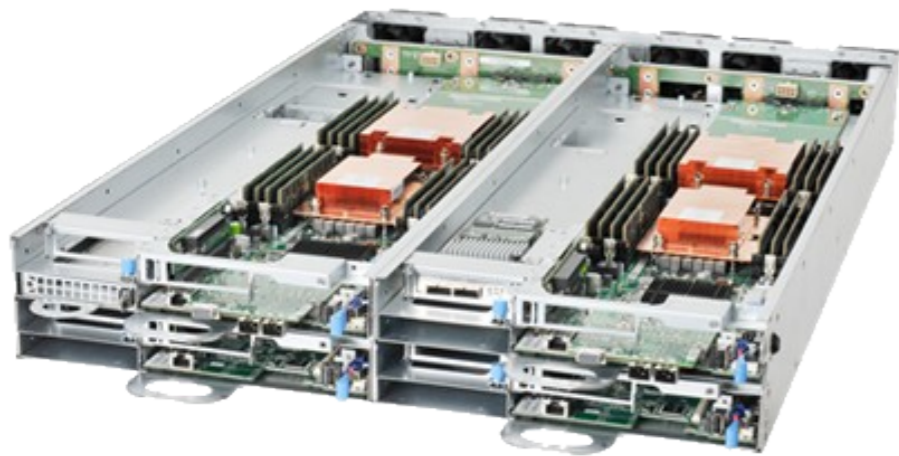
A processor is a physical piece of hardware with one or more cores that executes instructions



processors, nodes, processes and threads

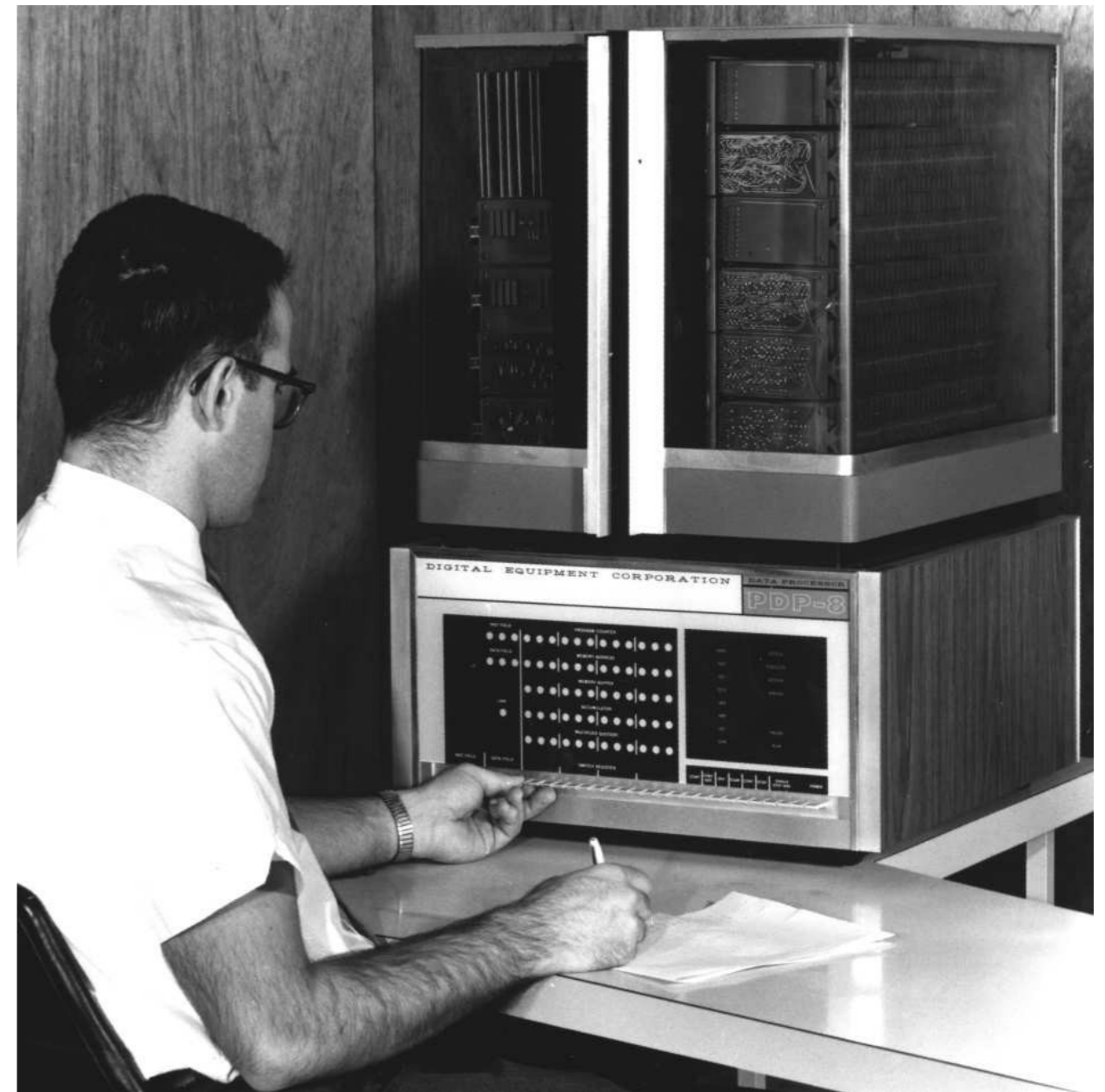
A node is one or more processors along with associated devices (disk drive, memory, i/o cards, communication cards, etc.

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.

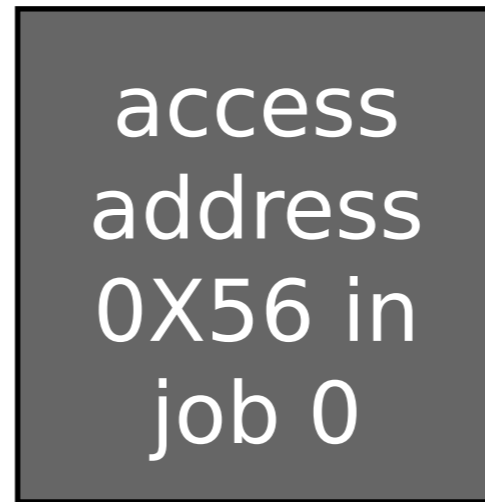
Virtual memory

OS

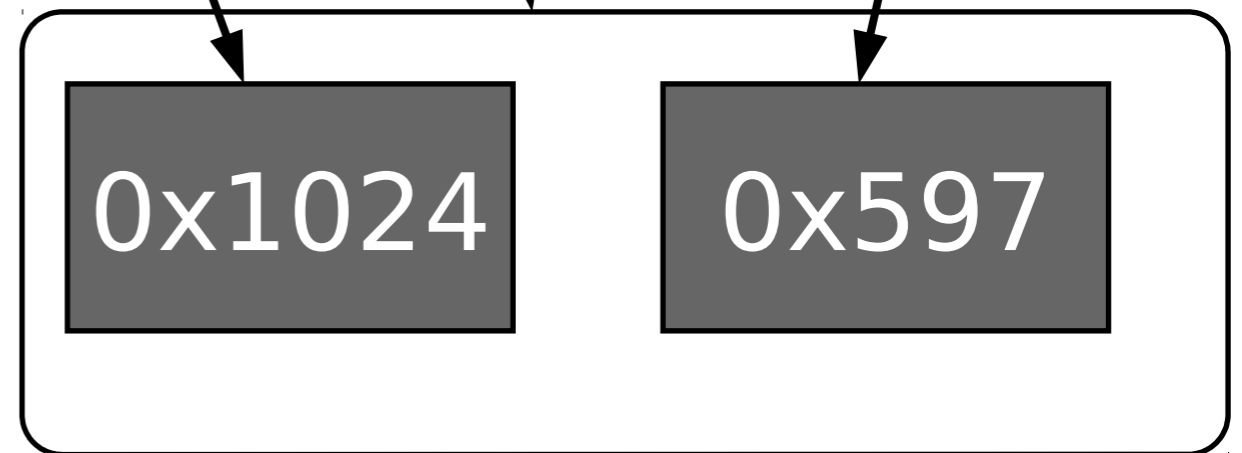
- An *operating system* allows multiple jobs and/or users to access the machine at the same time

- The OS virtualizes the machine -- each job sees the machine as entirely its own

Job 0



Job N



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

processors, nodes, processes and threads

- 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

Threads and processes -- summary

- *A thread* shares its address space with its 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

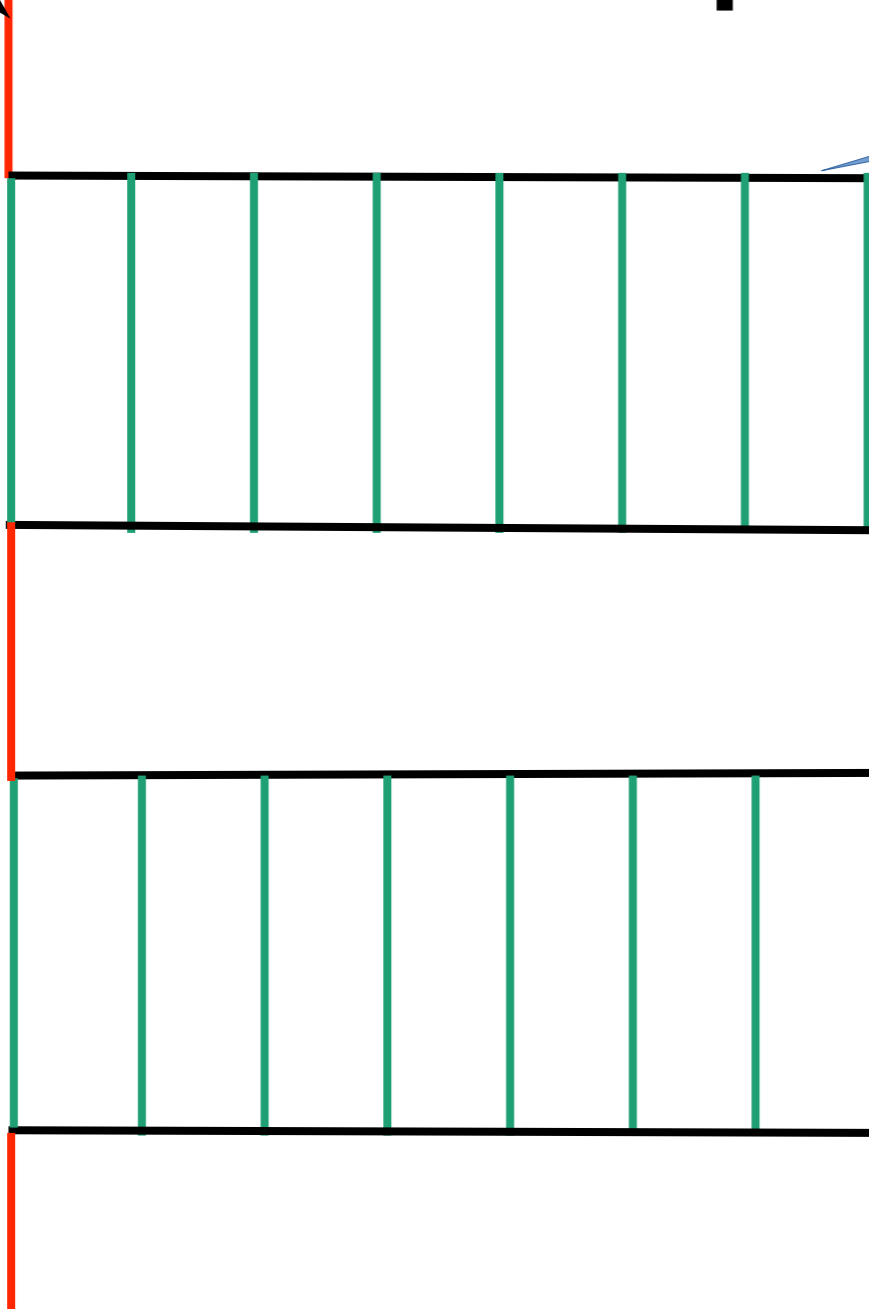
Typical thread level parallelism using OpenMP

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 regions



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.

Control over 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

Control over 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]
```

```
}
```

- 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]
}
No Barrier!
#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!*

Next -- OpenMP in
more detail