NTNU – Trondheim
Norwegian University of
Science and Technology

Cache memory

A bare minimum of logic

- The smallest common basis for programming languages is that they can evaluate
  - Expressions: carry out some set of operations on given values
  - Variables: give names to values, and recall them later
  - Conditionals: do something only when an expression is valid
  - Jumps: fetch next expression from a different place in the program

- That is, we've got memory and repetitions

  (without these, the length of a program must be proportional to the time required to run it...)

Memory locality

- The vast majority of loops
  - re-evaluate the same variables every iteration, and/or
  - go through an ordered list of elements
- This means that when we read a variable,
  - we'll often need it again soon, and/or
  - we'll soon need some values that are stored close to it
- Those two are called the *temporal* and *spatial locality* of a program.

Capacity and speed

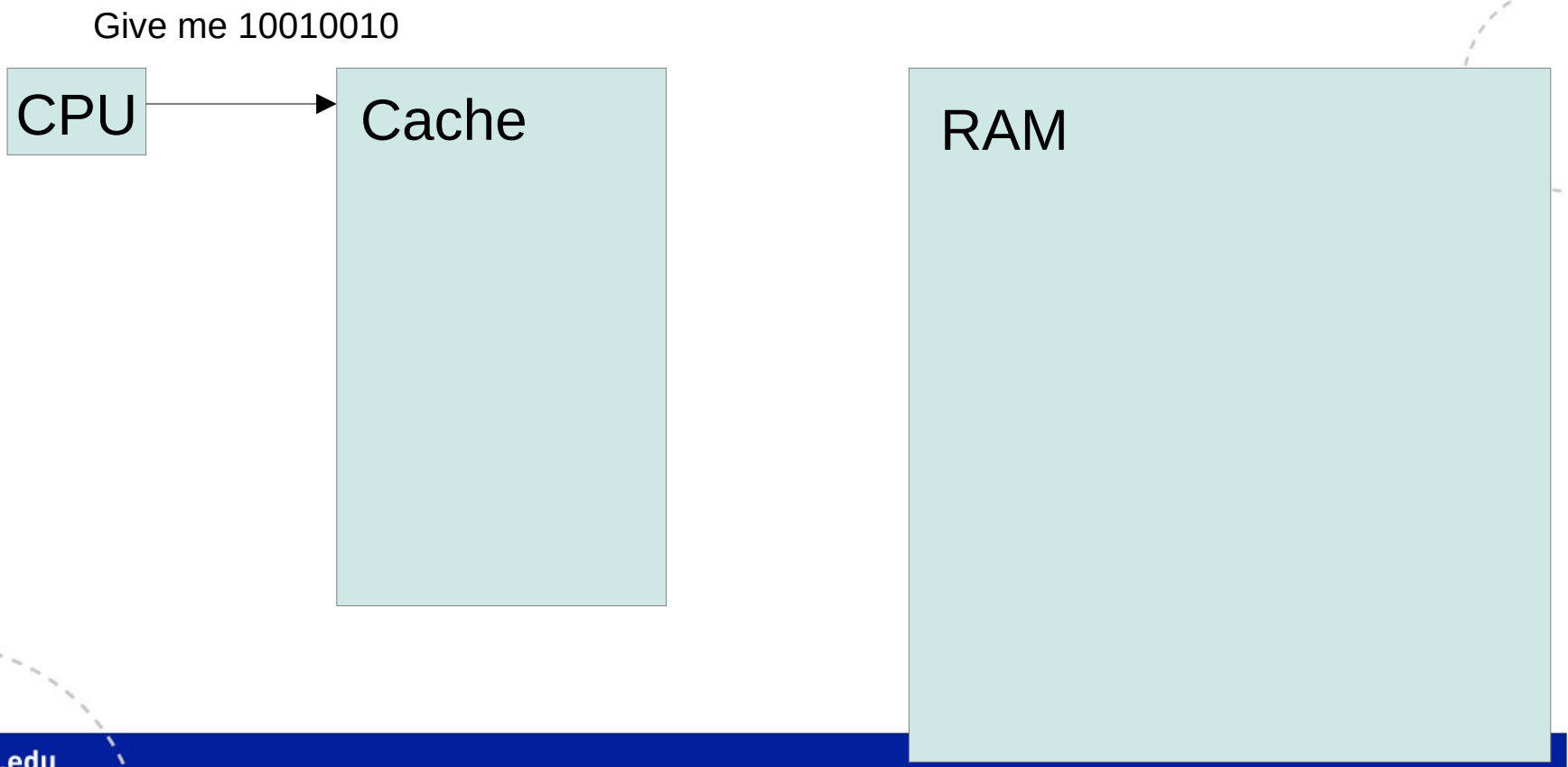
- Big memory systems are slow, small ones are fast
- An inaccurate rule of thumb:
  - Registers: 10-100s of bytes, ~1 CPU cycle access time
  - L1 cache: 16-64kB, 5-10 cycle access time
  - L2 cache: 256kB-32MB, 50-100 cycle access time
  - ...
  - RAM: 4GB+, 500-1000 cycle access time
- The exact numbers change with the season, but their ratios remain comparable

Using what we've got

- Bandwidth is easier to improve than latency
- When we have to spend the time to fetch things from memory, we might as well fetch as much as we can
- Since we expect some spatial and temporal locality out of the program, it's a good guess to fetch
  - the variable itself (because we'll need it again soon)
  - a bunch of its neighbors (because we'll need them soon enough)

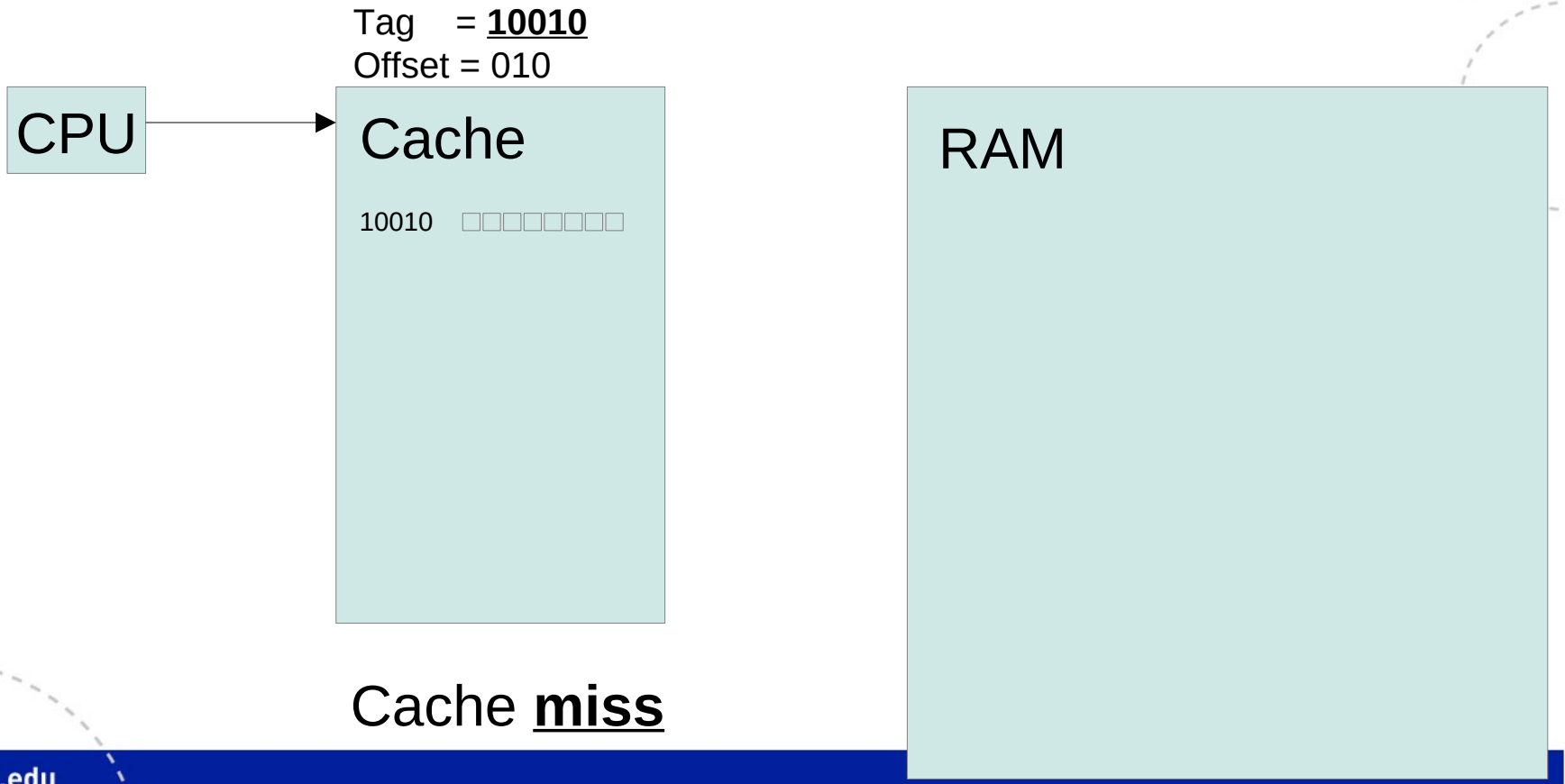
What a simple cache does

- Suppose we have an 8-bit CPU, and it wants address 10010010...



What a simple cache does

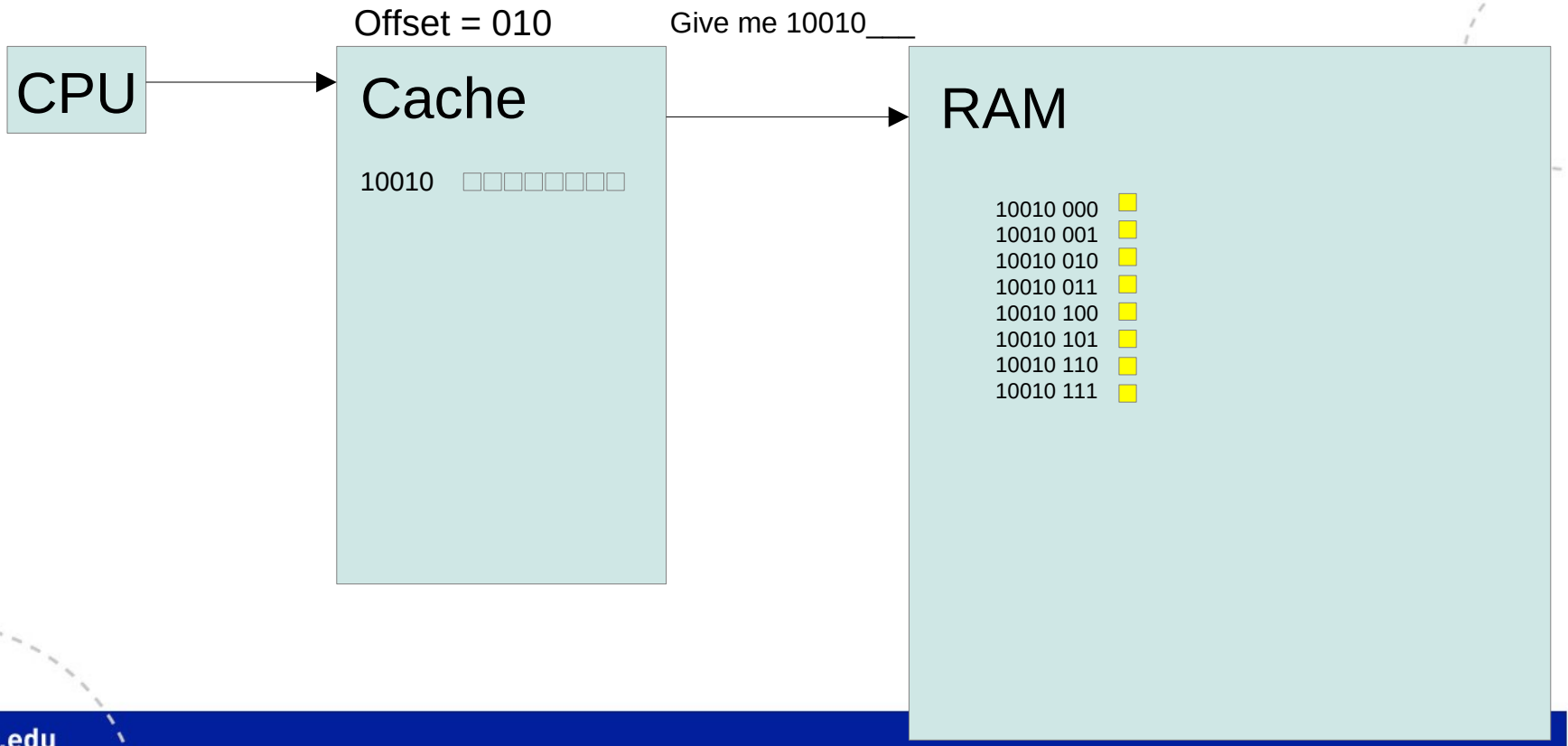
- Suppose we have an 8-bit CPU, and it wants address **10010**010...



Cache **miss**

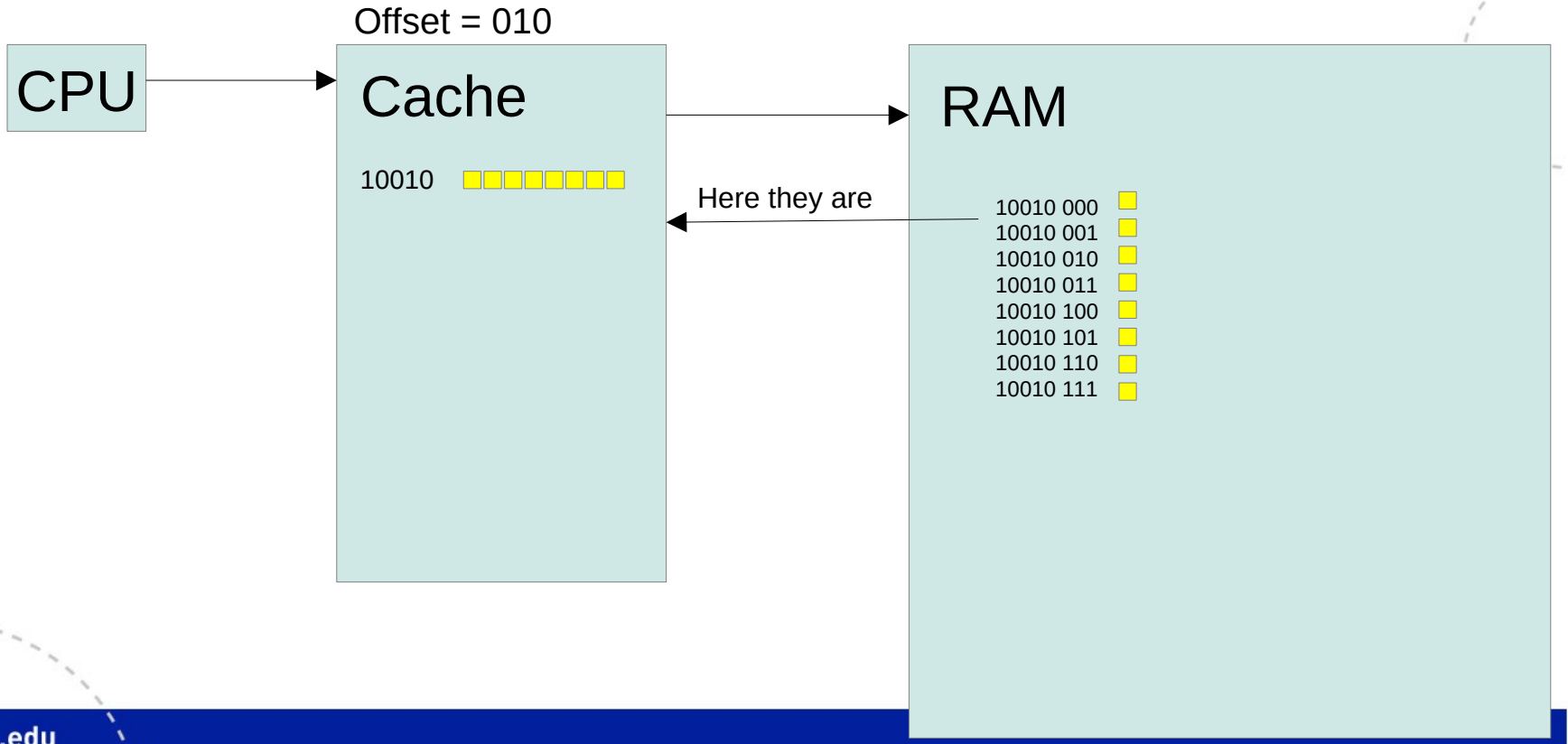
What a simple cache does

- Suppose we have an 8-bit CPU, and it wants address 10010010...



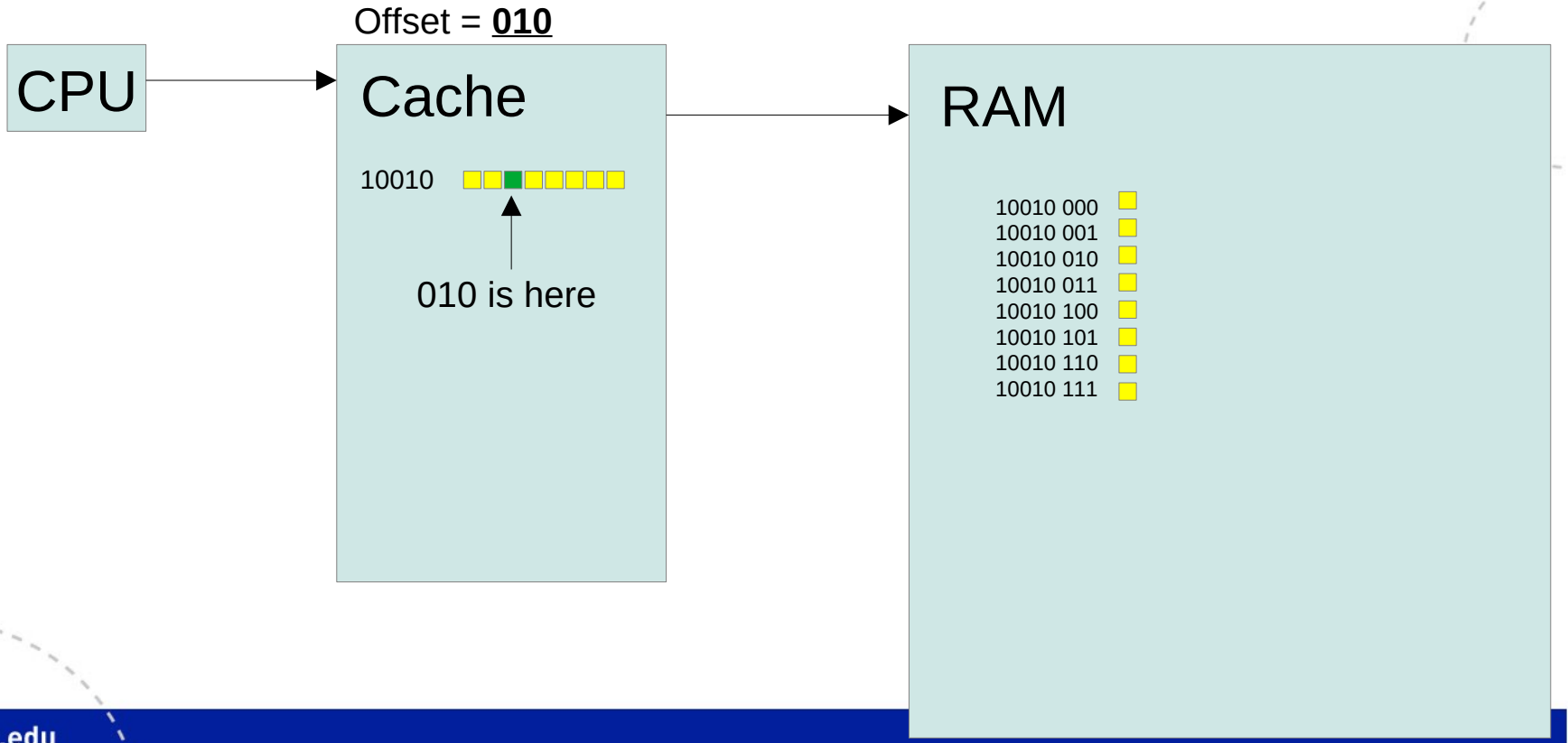
What a simple cache does

- Suppose we have an 8-bit CPU, and it wants address 10010010...



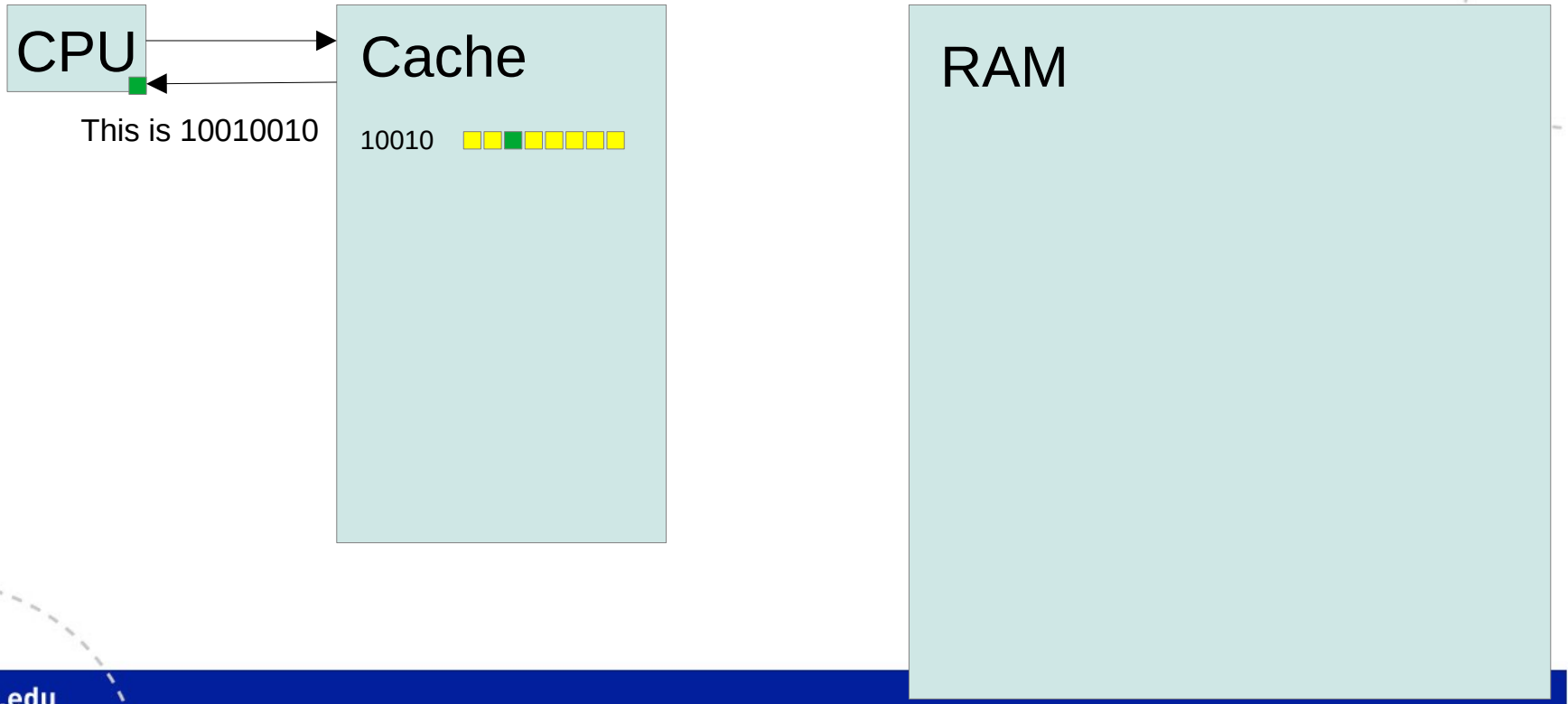
What a simple cache does

- Suppose we have an 8-bit CPU, and it wants address 10010**010**...



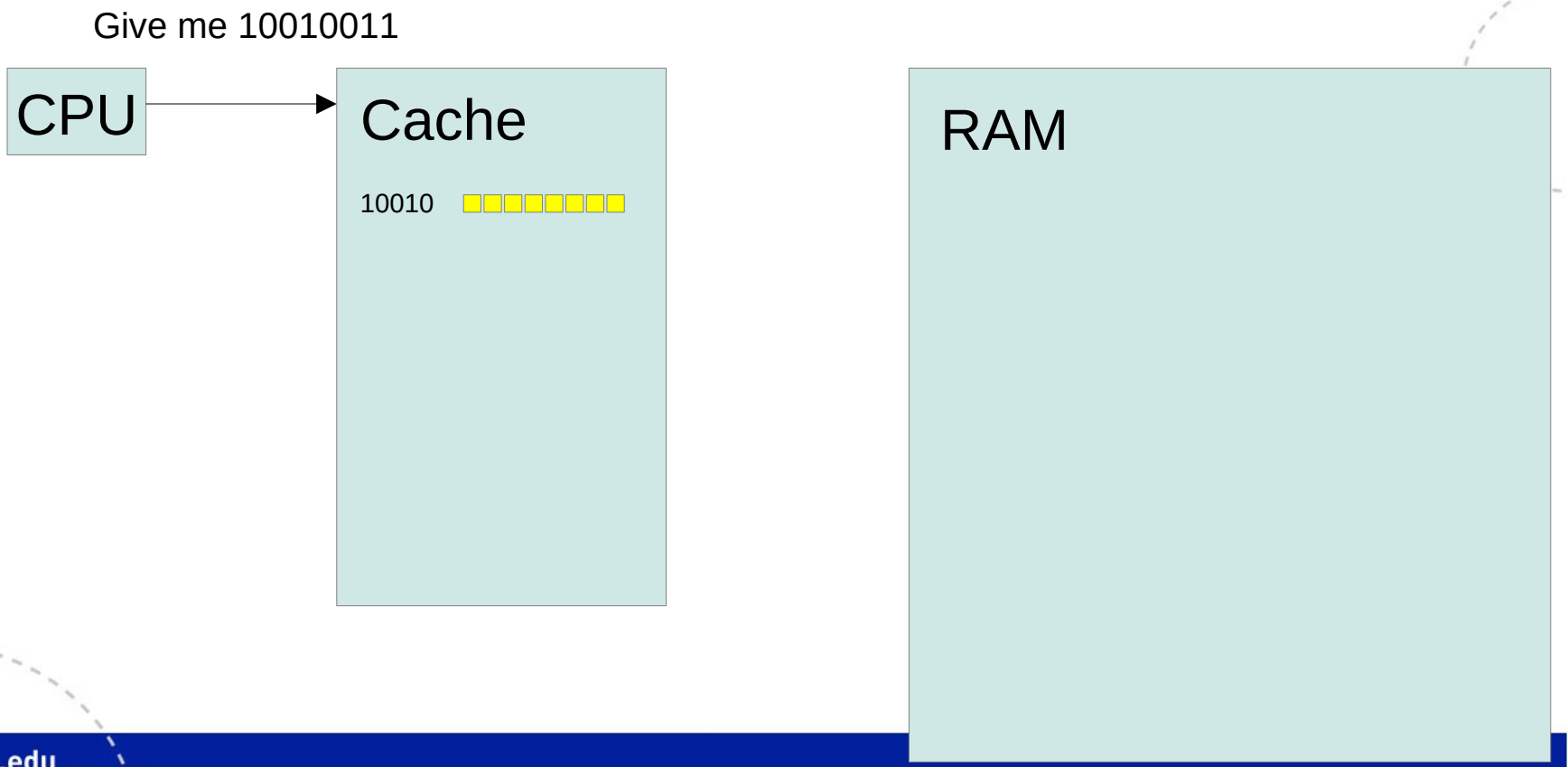
What a simple cache does

- Suppose we have an 8-bit CPU, and it wants address 10010010...



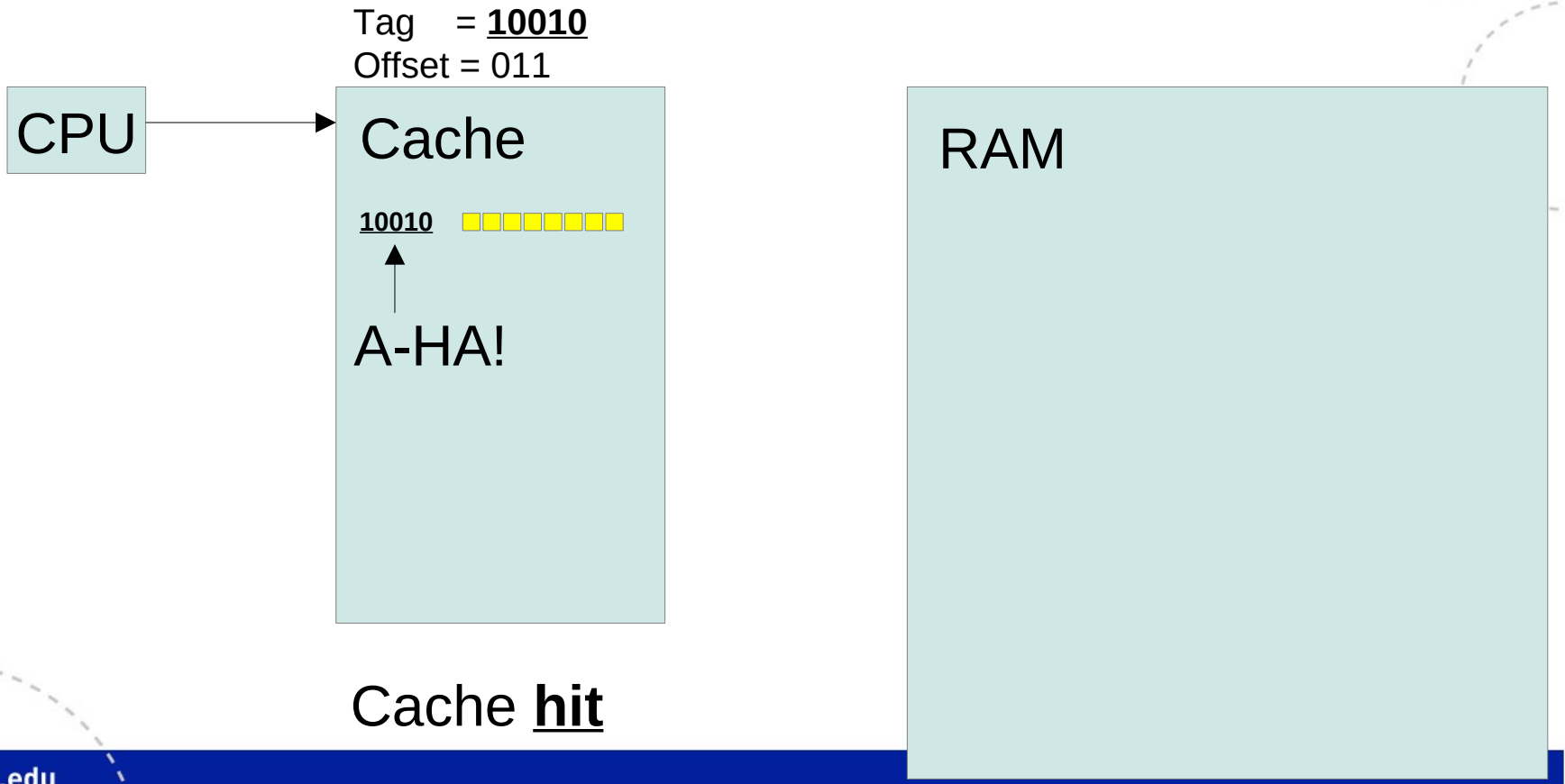
Next iteration

- Now the CPU wants address 10010011



Next iteration

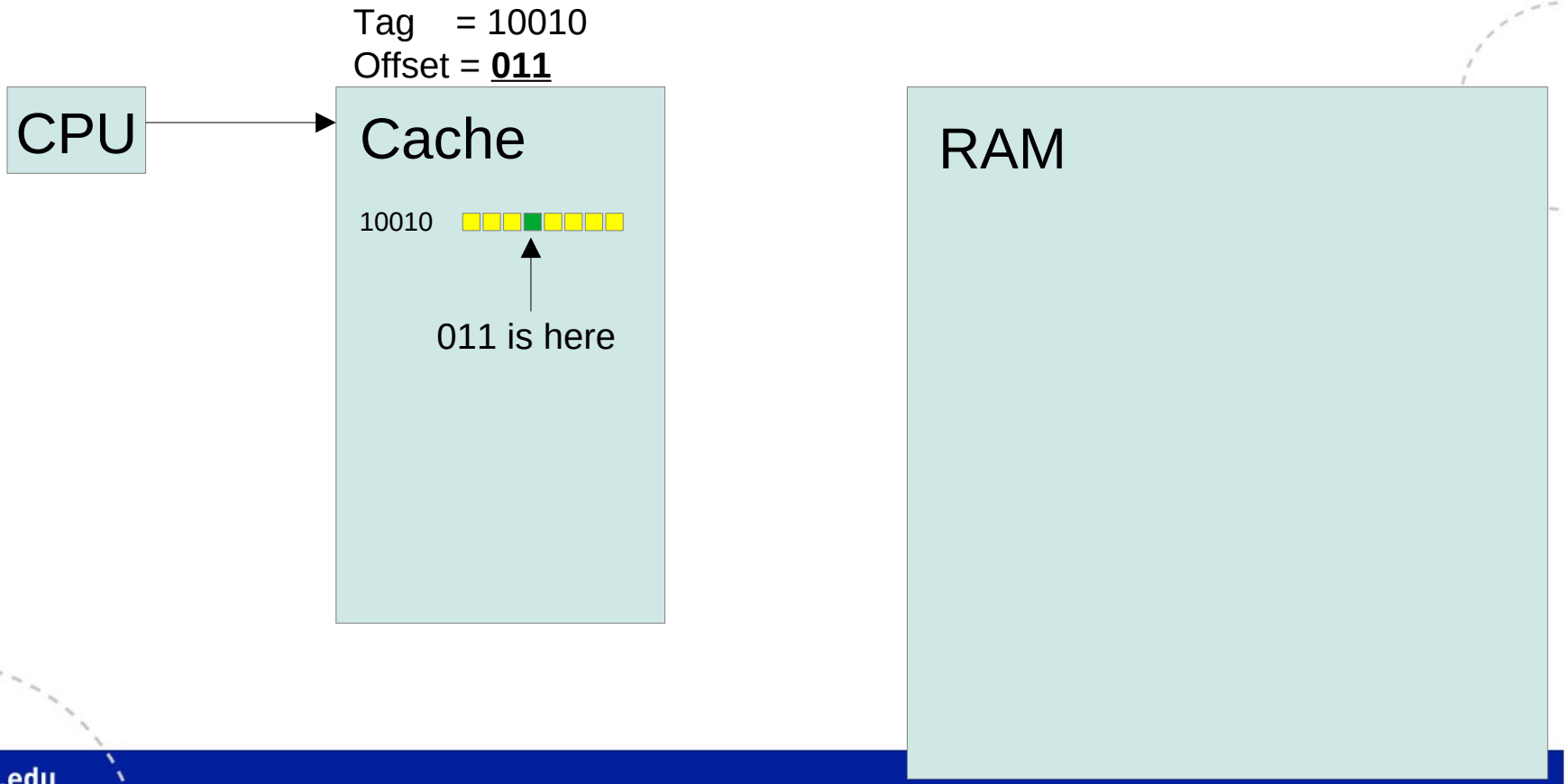
- Now the CPU wants address **<u>10010</u>**011



Cache **<u>hit</u>**

Next iteration

- Now the CPU wants address 10010**011**



Next iteration

- Now the CPU wants address 10010011

As you can see...

- Cache will not save any time on the first fetch
  - It is limited by the latency of RAM
  - By using more bandwidth, we can predict a future saving
- Cache saves time when the next fetch is from the same location, or one of its neighbors
- Each little neighborhood of values is called a *cache line,* there's room for several in the cache
- Every cache access requires a search through the tags that are already associated with lines

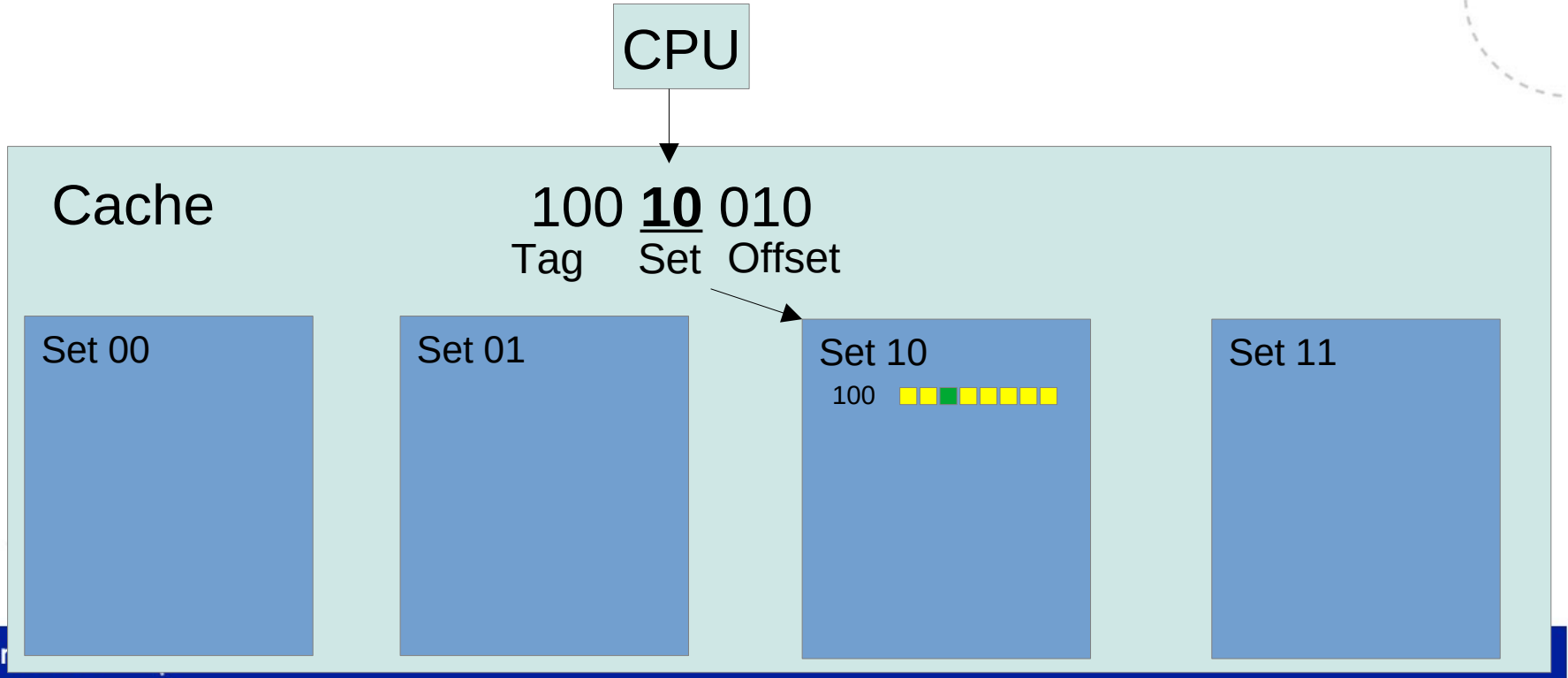
  ...so bigger caches become slower

A common speed improvement

- What we have seen is a *fully associative* cache
- Bigger specimens cut down the search time by a constant factor
- Use some bits to separate the entire address space into distinct parts
- Use fully associative caches for each part

Set-associative cache

- Divide the space into independent parts called *sets*
- Statically partition the whole address space between them



Points of order

- The # of sets are called the “ways of associativity”
  - What we just drew was a *4-way associative* cache
- Line lengths are powers of 2
  - We need them to cover all combinations for a group of bits
- Ways of associativity are powers of 2
  - We need them to cover all combinations for a group of bits also
- The # of lines in each set can be any nice integer
  - This only affects how many tags we must search through before declaring a cache miss

Hierarchical memory

- Because the lookup speeds of caches are tied to their sizes, a hierarchy of caches has evolved:
- Level 1 sits close to the registers, there are often separate caches for instructions and data here
  - If something isn't in L1 cache, a bigger L2 is searched (typically contains both instructions and data, may be private to one processor or shared between several)
    - If something isn't in L2 cache, L3 is searched, it's typically shared by every core on a CPU socket
      - There may be an L4 cache which sits outside of the CPU chip
        - *etc. etc.* 3 levels is typical in 2023
- We try not to go to memory unless the last level cache (LLC) declares a miss

Virtual memory

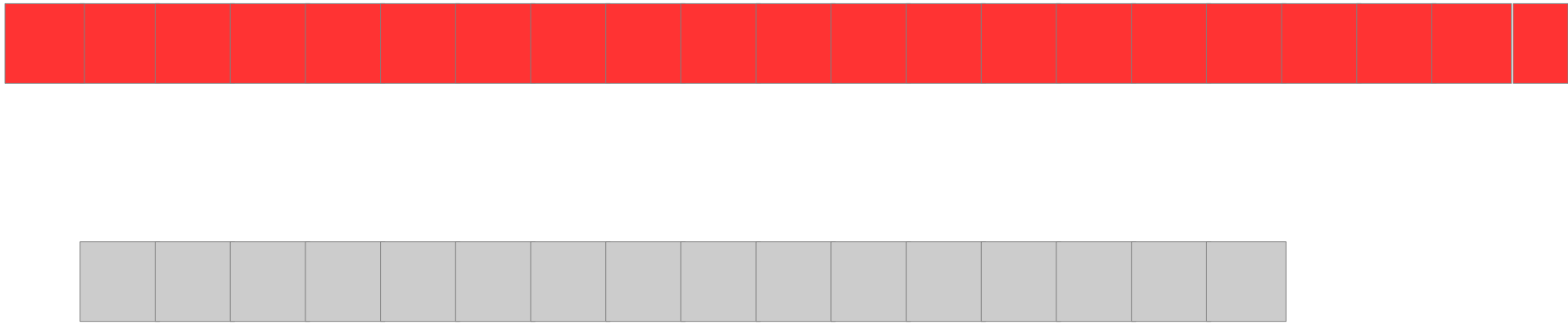
- Modern CPUs also have a memory management unit (MMU) that allows the operating system to do something similar with main memory:

  The process memory image thinks it has the entire processor address range to itself, partitioned in *pages*:

  The actual, physical memory is divided into *frames* with the same size as pages:

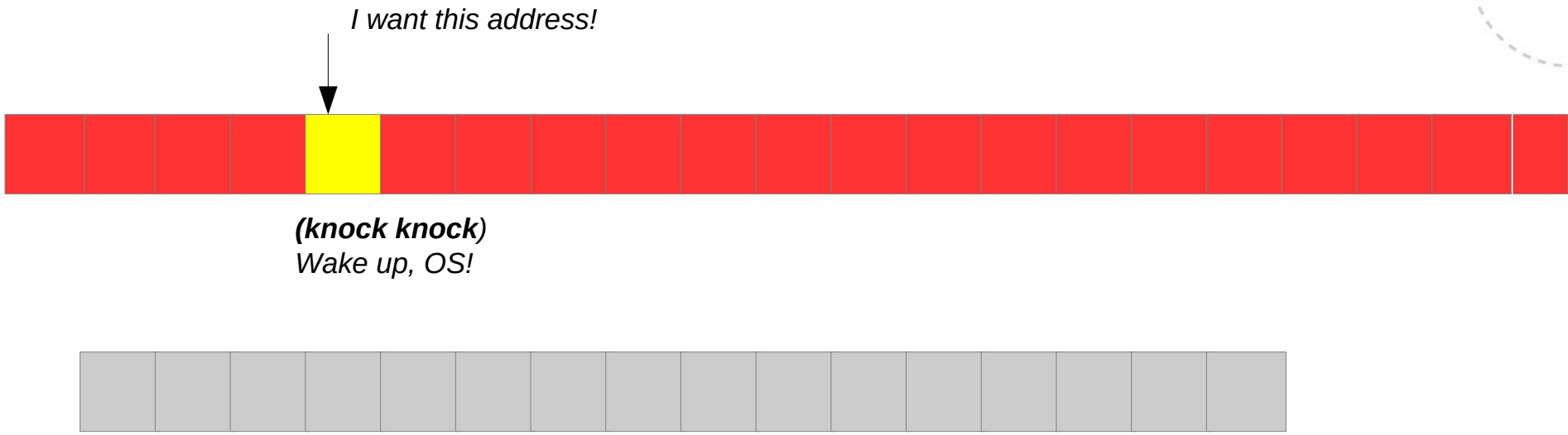
Page protection

- The OS maintains a table that maps pages to frames
- Unused pages have a default setting as *protected*



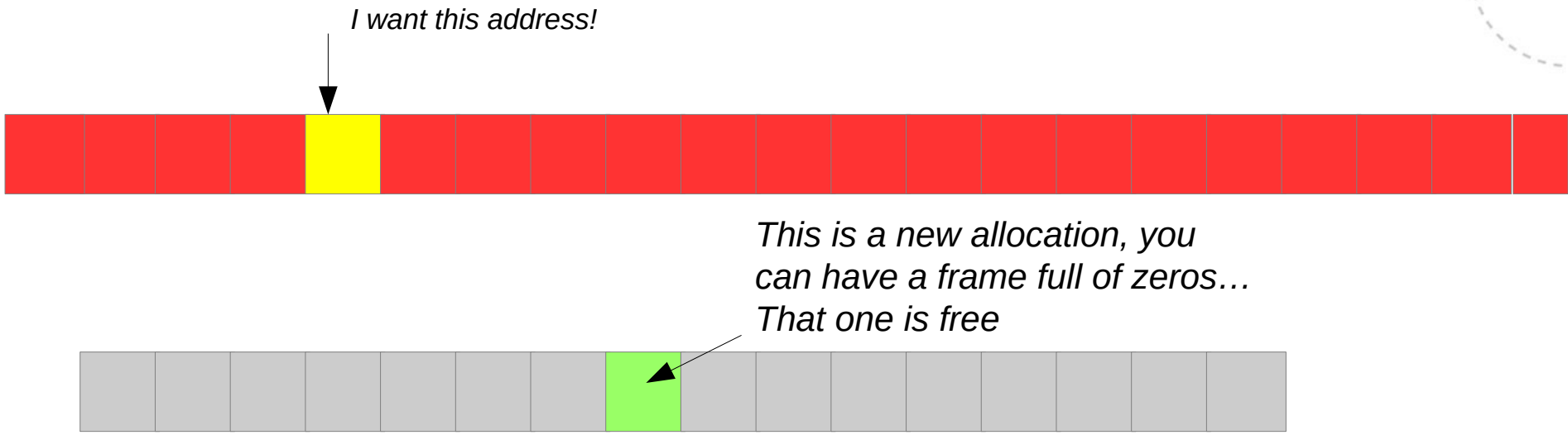
Page fault

- When a protected page is accessed, it creates a *page fault*, which triggers the OS to step in and determine what to do



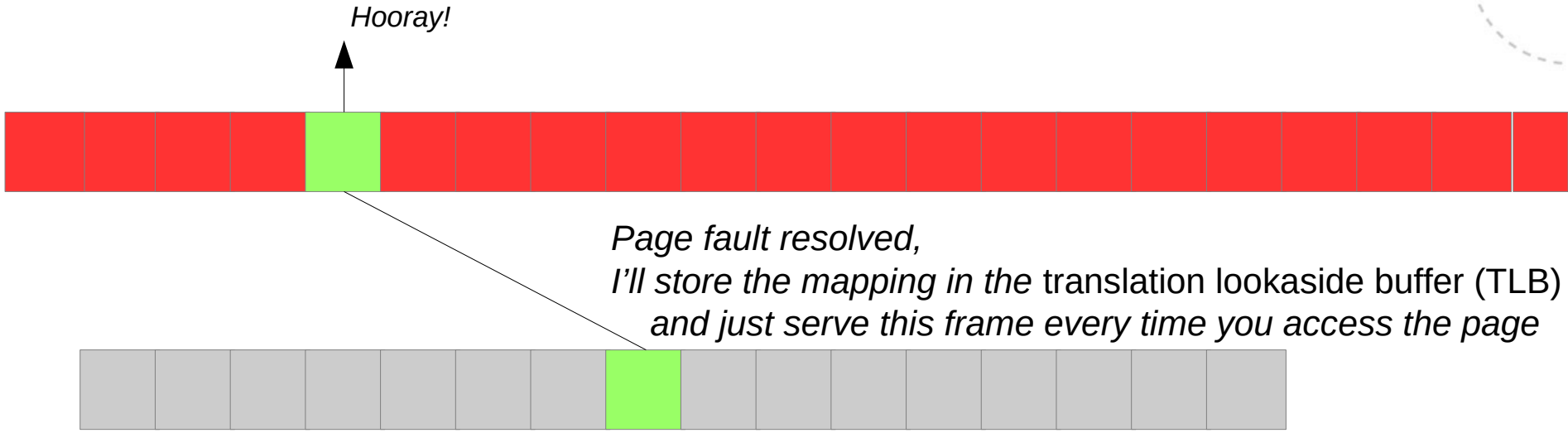
Mapping to a physical frame

- Within the process image, pages can lie within a memory-mapped file, or be empty
- Contents are loaded from disk, or initialized as appropriate



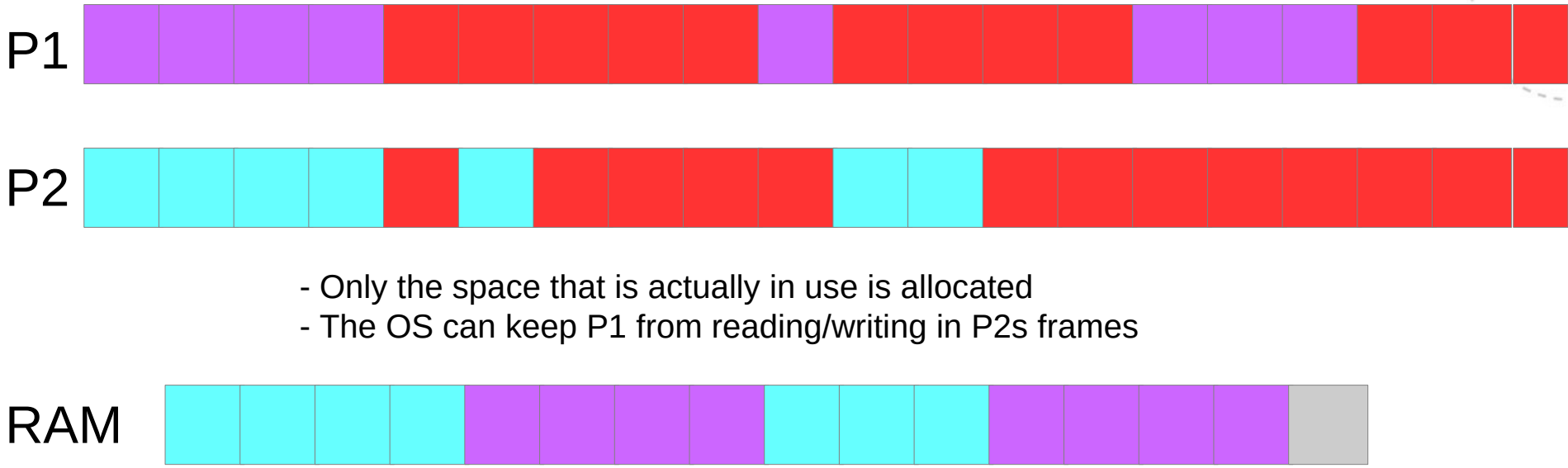
The program continues

- As the process completes, the program can continue to deal with its private linear address space
- The OS+MMU take care of the fact that virtual adr. 0x00001000 may actually be physical adr. 0xCAFE1000 or 0xBEEF1000 or something



The **main** benefit of this

- Several programs can share the same physical memory (as long as none of them take all of it)



The commonly cited 2nd benefit

- If a frame isn't used for a long time, the OS can
  - Protect the page again
  - Save its contents in *swap space* (a file or dedicated disk partition)
  - Put it back in a frame when the program needs it again
- This lets us run a set of programs that collectively use more memory than there is physical backing for
- The transfer to/from disk is quite time consuming, though
  - If you try to run a set of programs that frequently overflow into swap space so that they end up constantly fighting for frames, your computer will probably freeze up until you cut its power

Page sizes

- There’s a tradeoff between size and speed here
  - Large pages rarely give page faults, but a 10-byte allocation can end up getting an entire big page to itself, reducing the utilization of available physical memory
  - Small pages waste less space, but cause page faults more frequently
- Pages are usually 4096 bytes by default, as a frequently reasonable compromise
- Page size is often configurable if you recompile the kernel of your OS

Why are *we* talking about it?

- Several independent programs on one computer can collaborate on the same problem
  - It's a way to get parallel computing
- Page faults affect the performance of your program
  - If you allocate a gigantic array, you'll be assigned all the virtual memory almost immediately, but access will be measurably slower at every 4kB boundary the first time you use the contents
- There are a few more advantages to virtual memory, but they don't concern us much in this class

Fun fact

- Via a system call, your program can set the protection bits of its own pages, and register callback functions for when the OS traps access attempts
- This mechanism can be exploited to write completely unreadable programs that implement some very creative tricks
- We won't need to do that either, but you can look into it on your spare time if you enjoy such things