The Memory Hierarchy

10/25/16

Transition

- First half of course: hardware focus
  - How the hardware is constructed
  - How the hardware works
  - How to interact with hardware
- Second half: performance and software systems
  - Memory performance
  - Operating systems
  - Standard libraries
  - Parallel programming

Making programs efficient

- Algorithms matter
  - CS35
  - CS41
- Hardware matters
  - Engineering
- Using the hardware properly matters
  - CPU vs GPU
  - Parallel programming
  - Memory hierarchy

Memory so far: array abstraction

- Memory is a big array of bytes.
- Every address is an index into this array.

This is the level of abstraction at which an assembly programmer thinks.

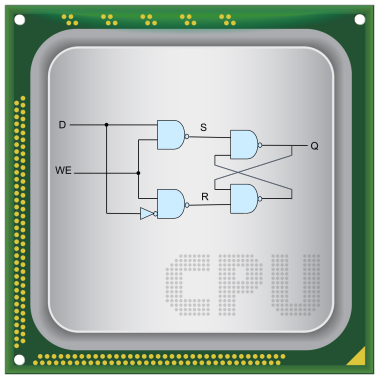
C programmers can think even more abstractly with variables.

Memory Technologies

Volatile

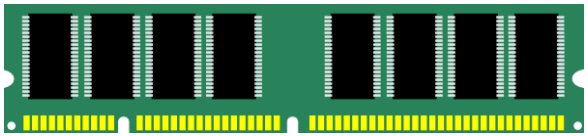
(loses data without power)

Latches (registers, cache)



$$

$$



Capacitors (DRAM)

$$$

Magnetic (hard drives)

$

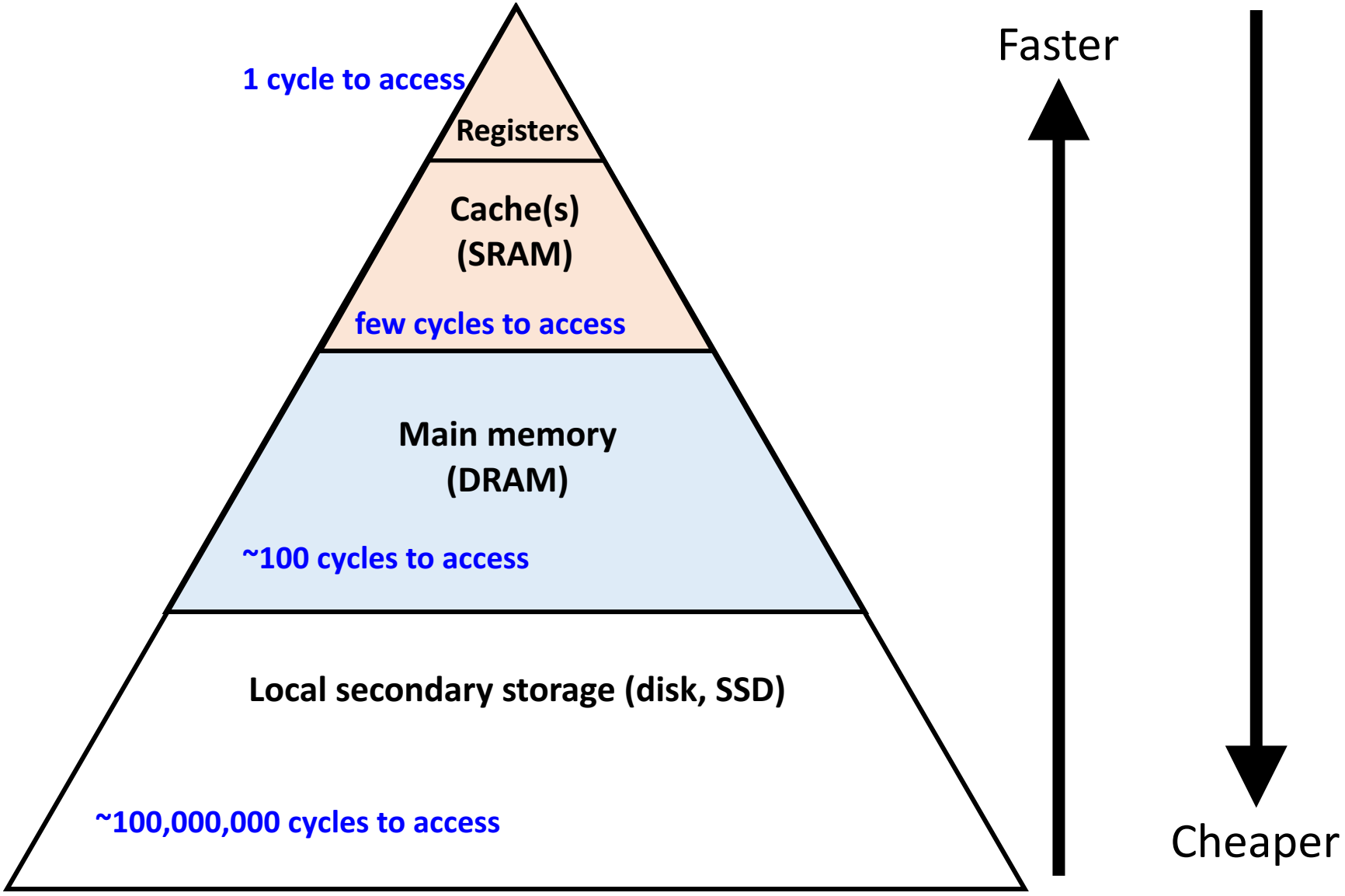
$$

Flash (SSDs)

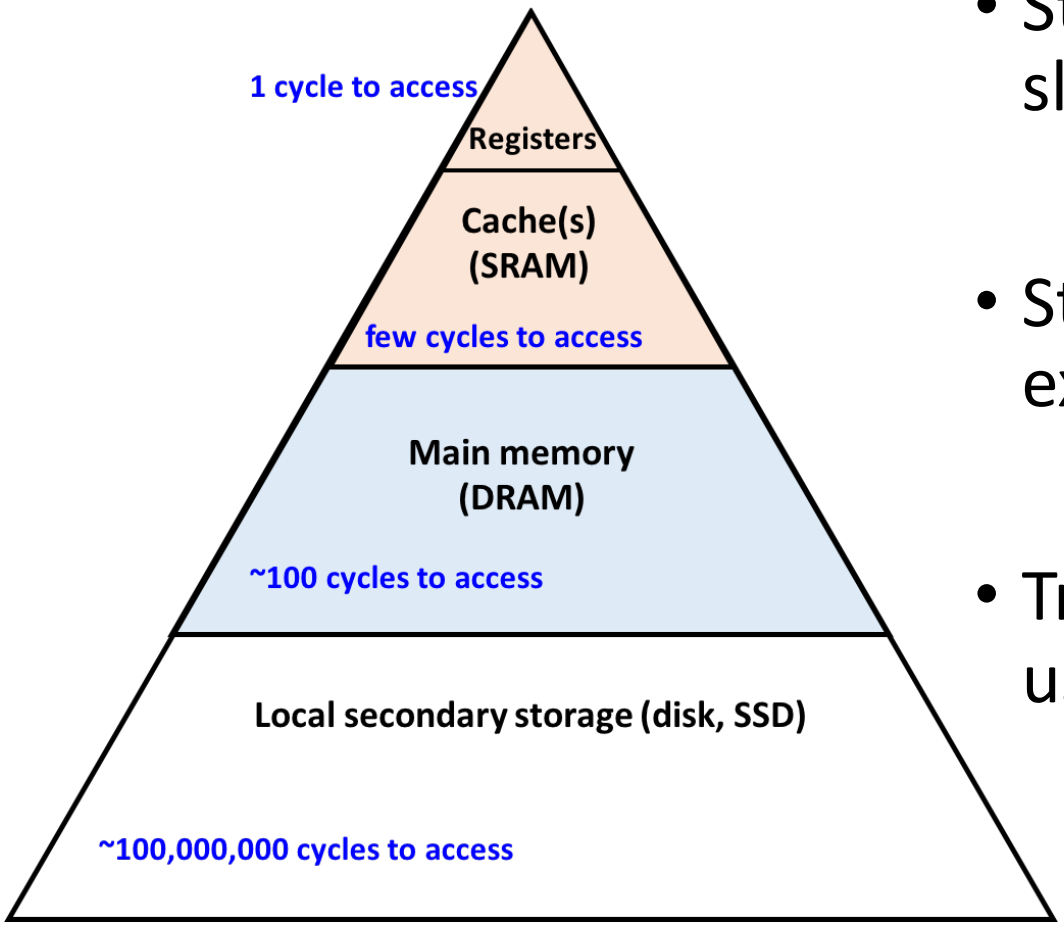
Non-Volatile

(maintains data when computer is turned off)

The Memory Hierarchy



Key idea this week: caching



- Store everything in cheap, slow storage.
- Store a subset in fast, expensive storage.
- Try to guess the most useful subset to cache.

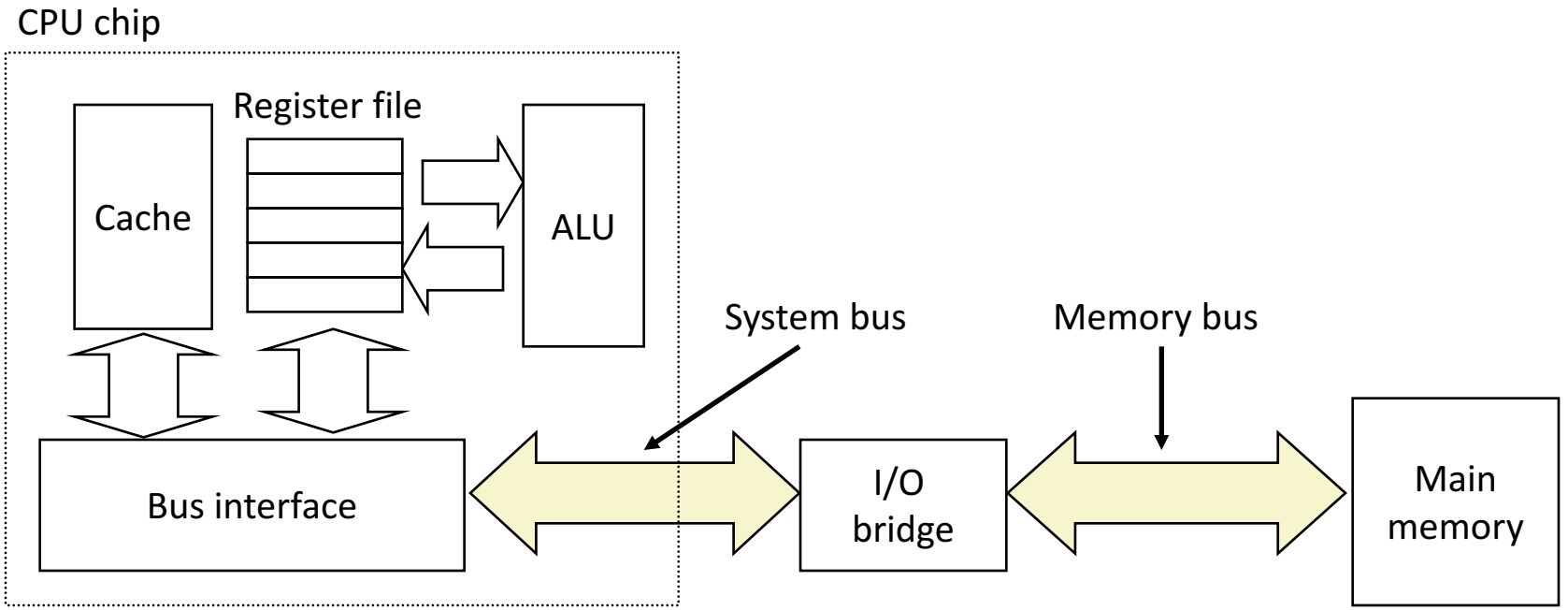
A note on terminology

- Caching: the general principle of holding a small subset of your data in fast-access storage.

- The cache: SRAM memory inside the CPU.

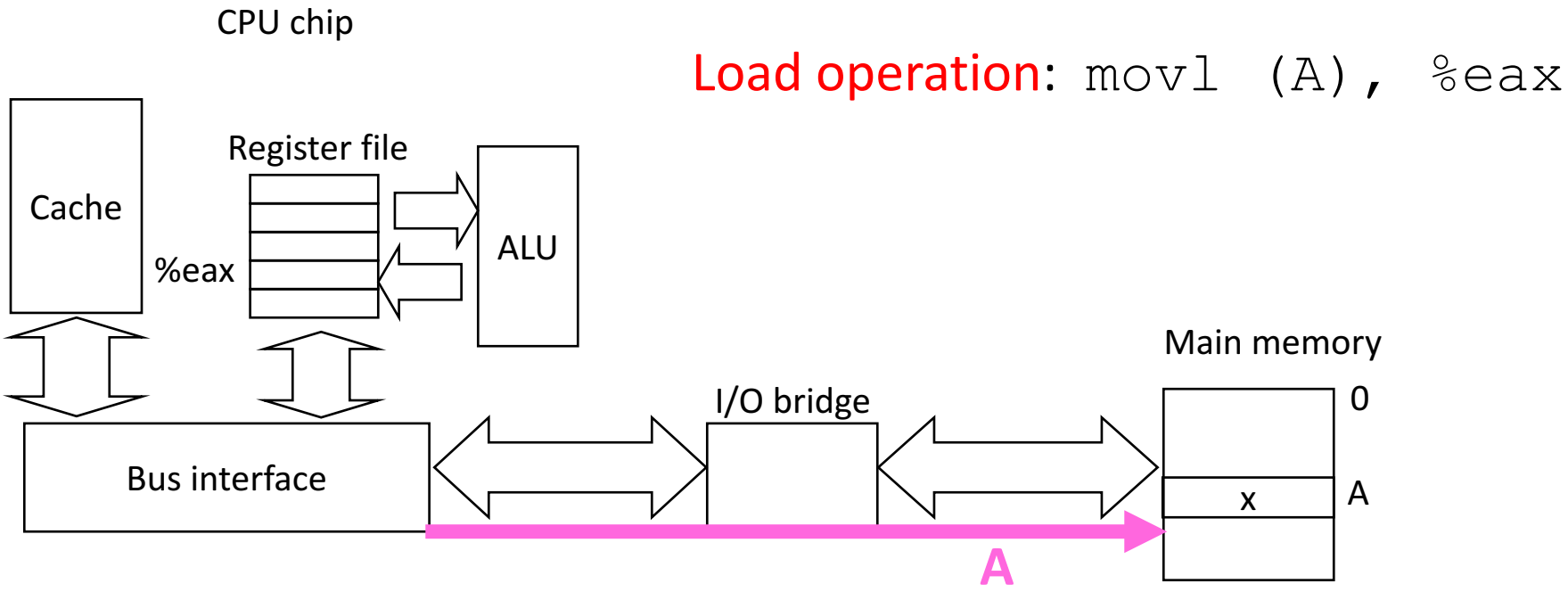
Connecting CPU and Memory

- Components are connected by a bus:
  - A bus is a bundle of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.



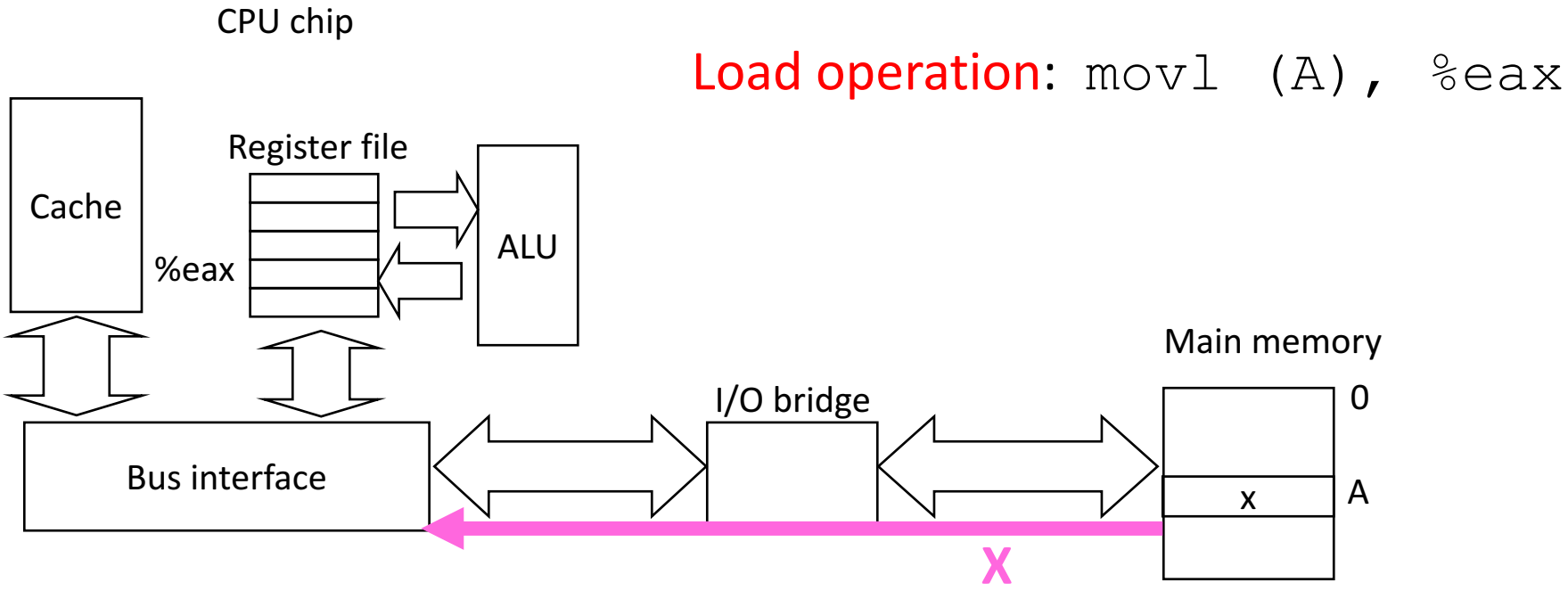
How a Memory Read Works

(1) CPU places address **A** on the memory bus.



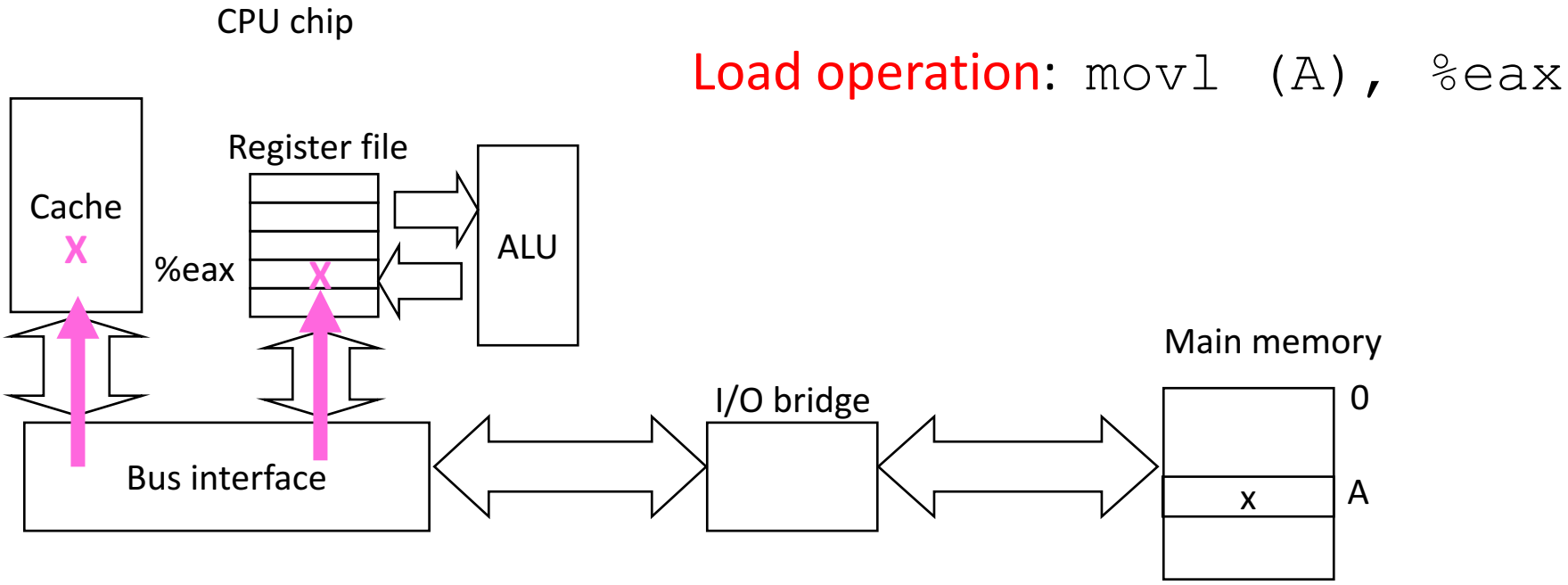
How a Memory Read Works

(2) Main Memory reads Address A from Memory Bus, fetches data **X** at that address and puts it on the bus



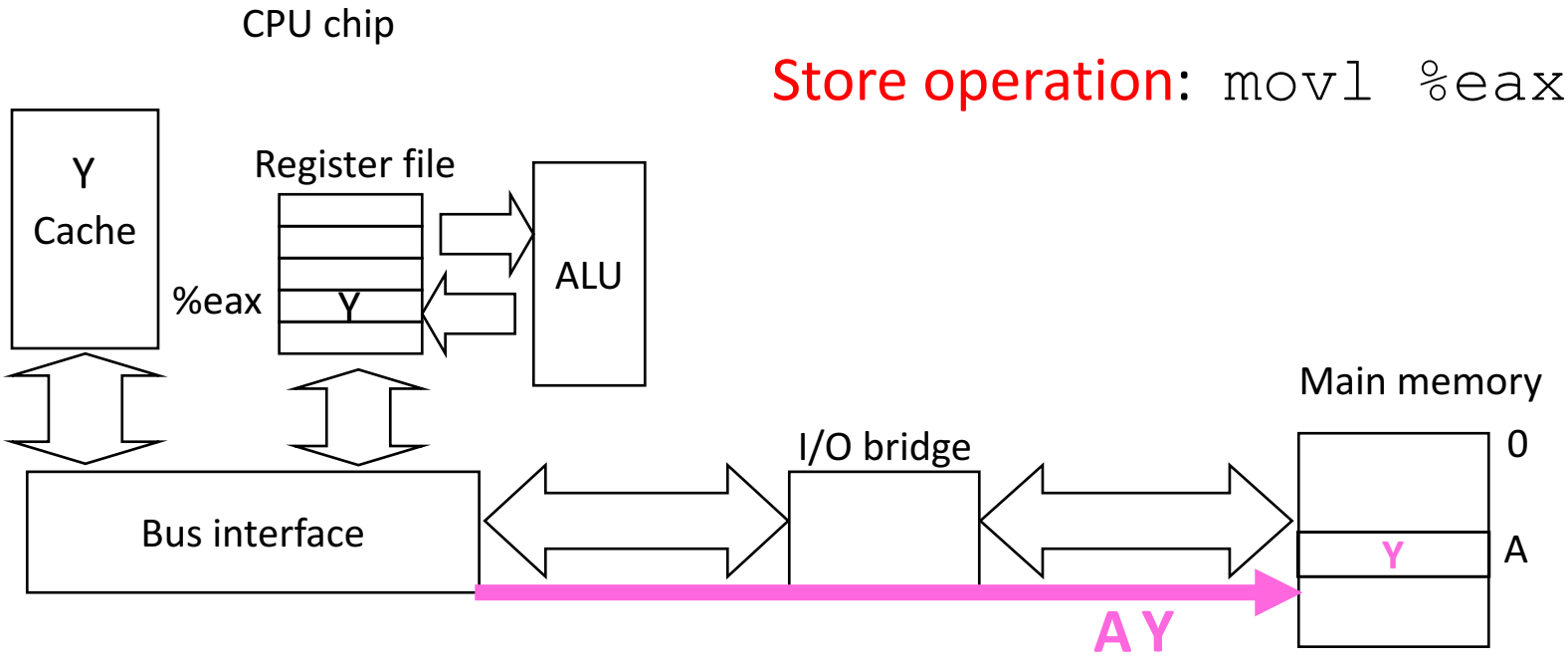
How a Memory Read Works

(3) CPU reads X from the bus, and copies it into register %eax. A copy also goes into the on-chip cache memory

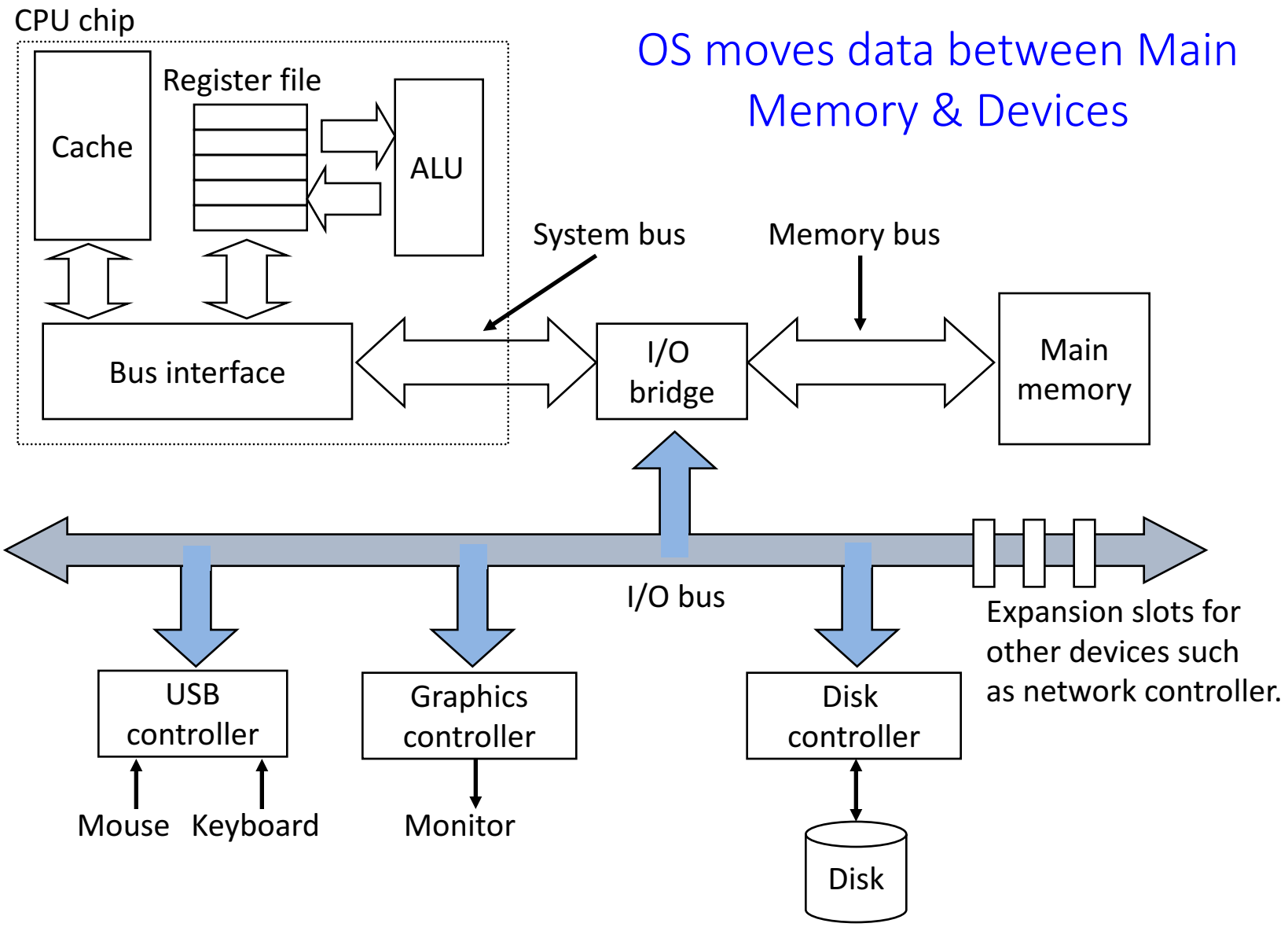


Write

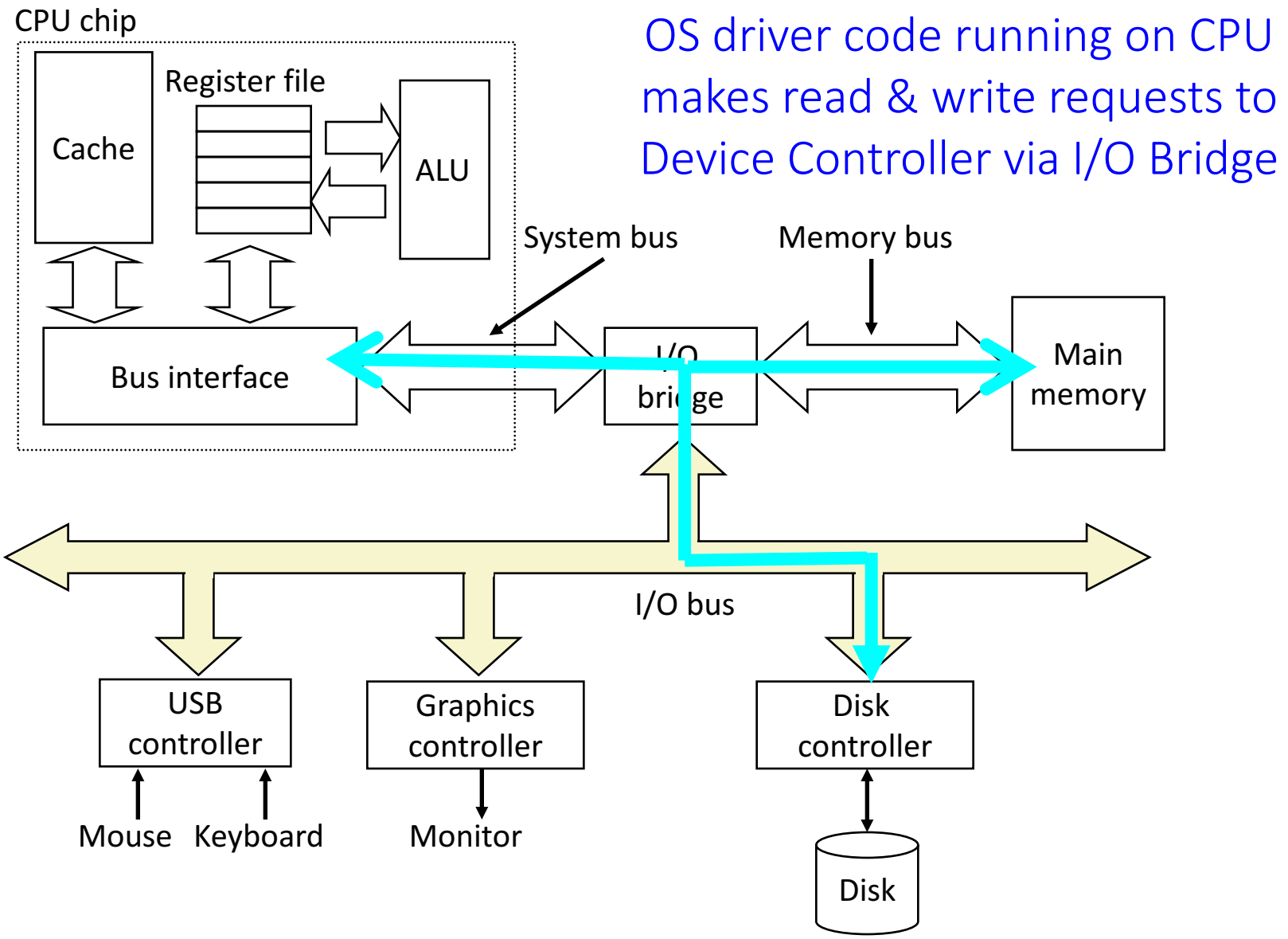
1. CPU writes A to bus, Memory Reads it
2. CPU writes Y to bus, Memory Reads it
3. Memory stores read value, y, at address A



I/O Bus: connects Devices & Memory



Device Driver: OS device-specific code



OS driver code running on CPU makes read & write requests to Device Controller via I/O Bridge

Abstraction Goal

- Reality: There is no one type of memory to rule them all!
- Abstraction: hide the complex/undesirable details of reality.
- Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.

What's Inside A Disk Drive?

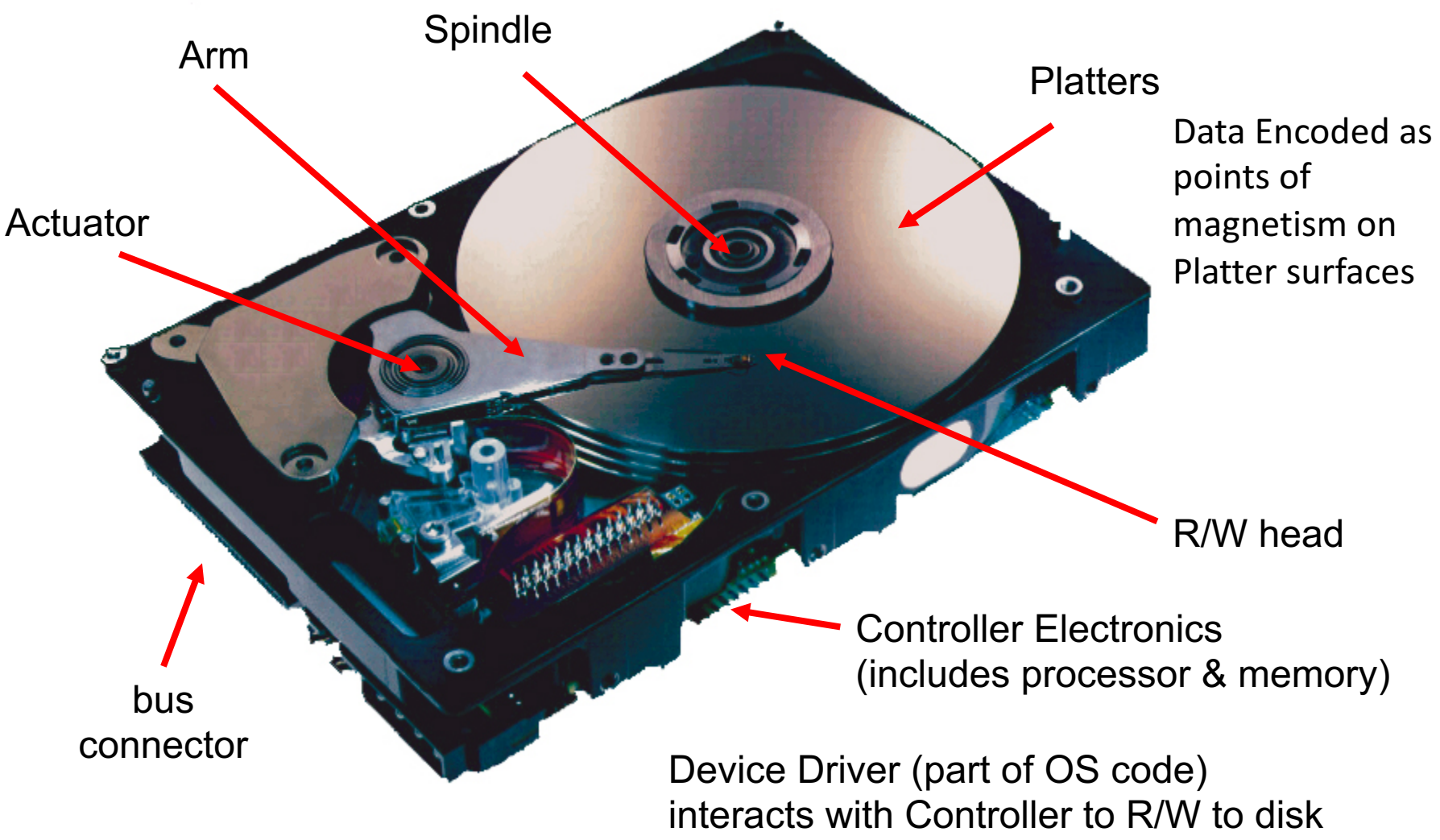
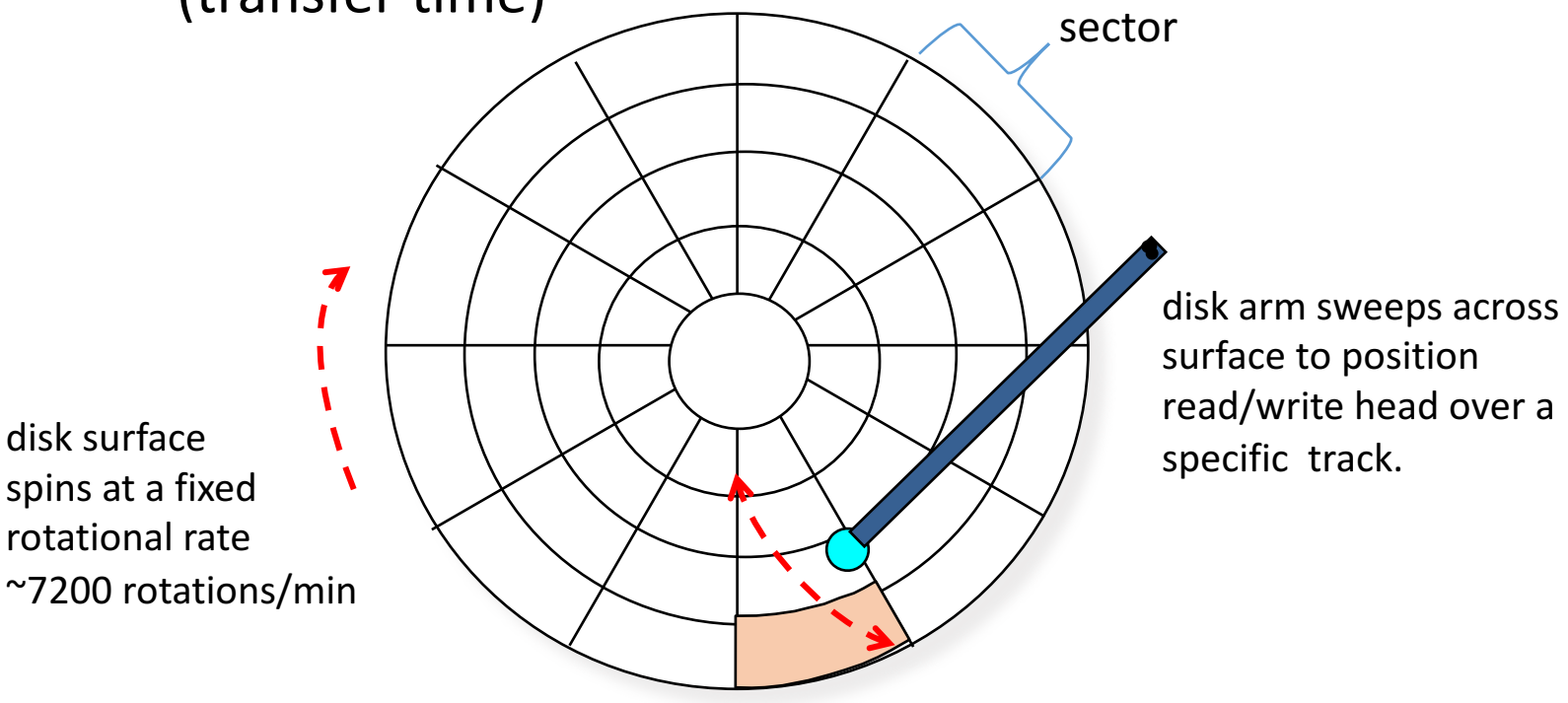


Image from Seagate Technology

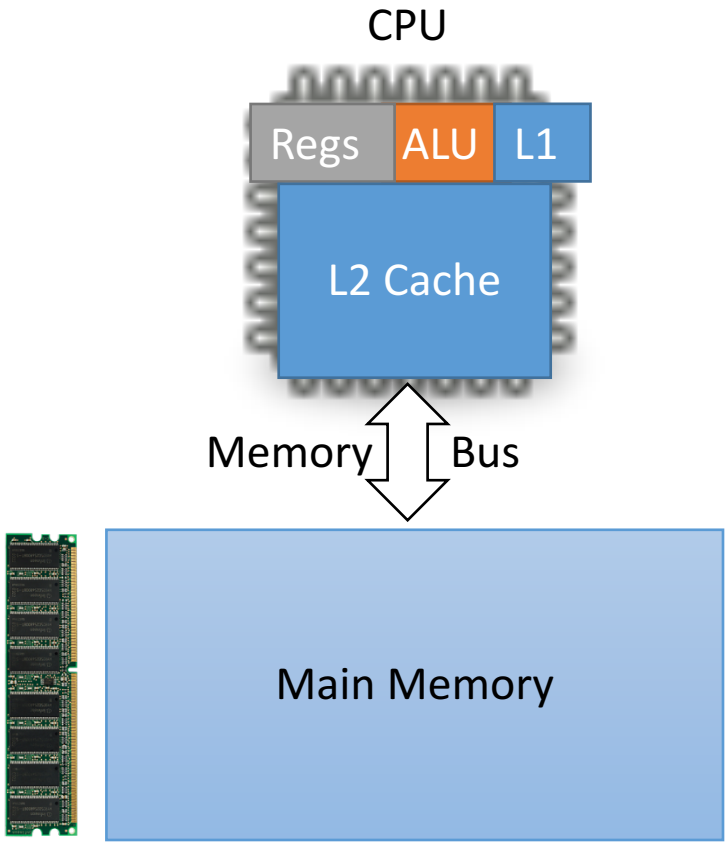
Reading and Writing to Disk

Data blocks located in some Sector of some Track on some Surface

1. Disk Arm moves to correct track (seek time)
2. Wait for sector spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)

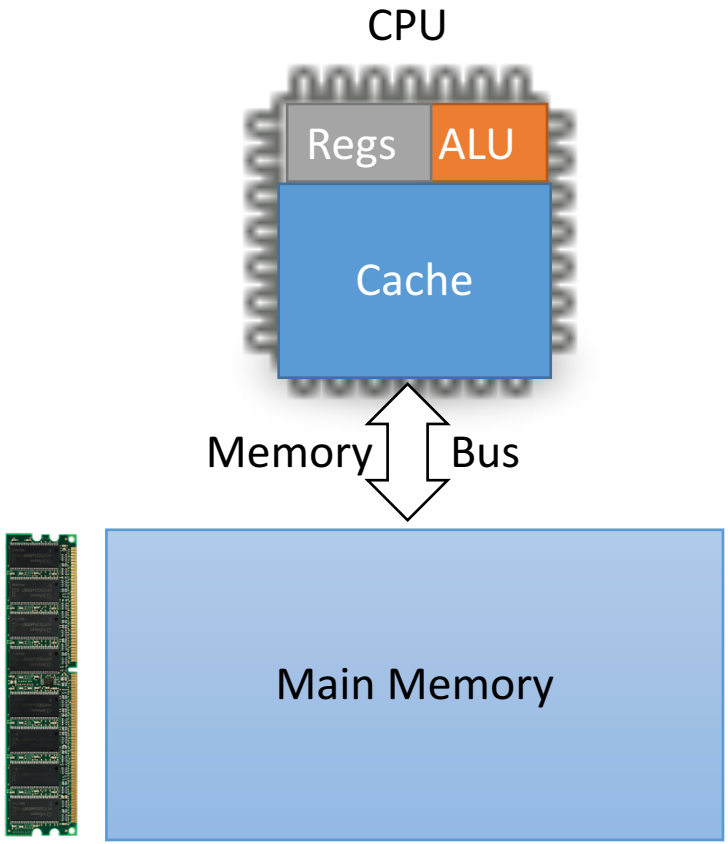


Cache Basics



- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- Same rules apply:
  - L1 subset of L2

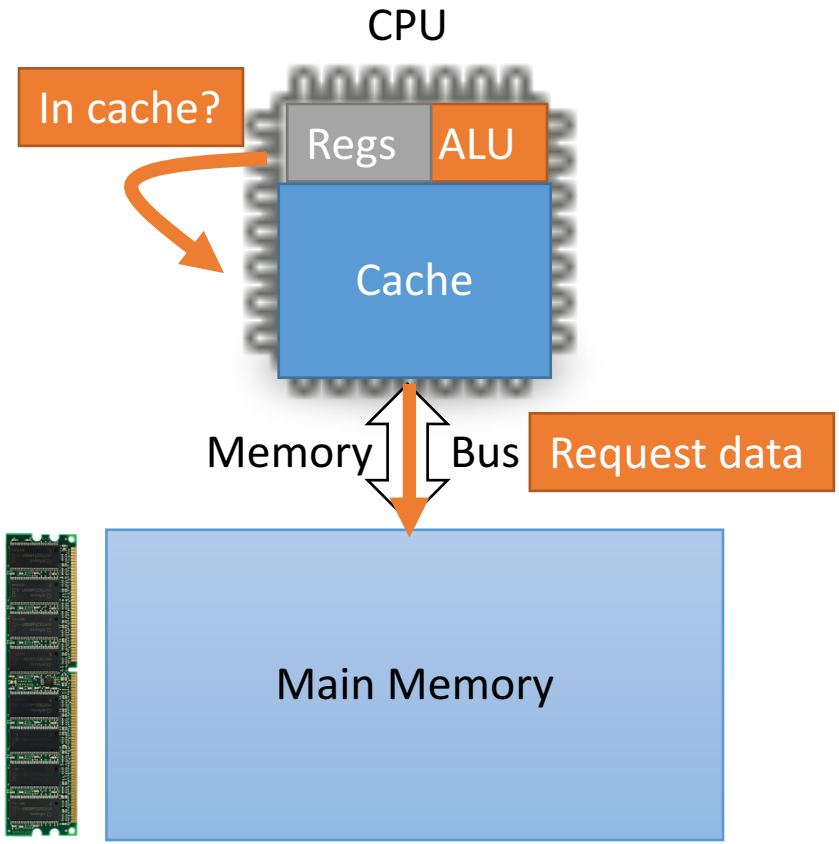
Cache Basics



Cache is a subset of main memory.
(Not to scale, memory much bigger!)

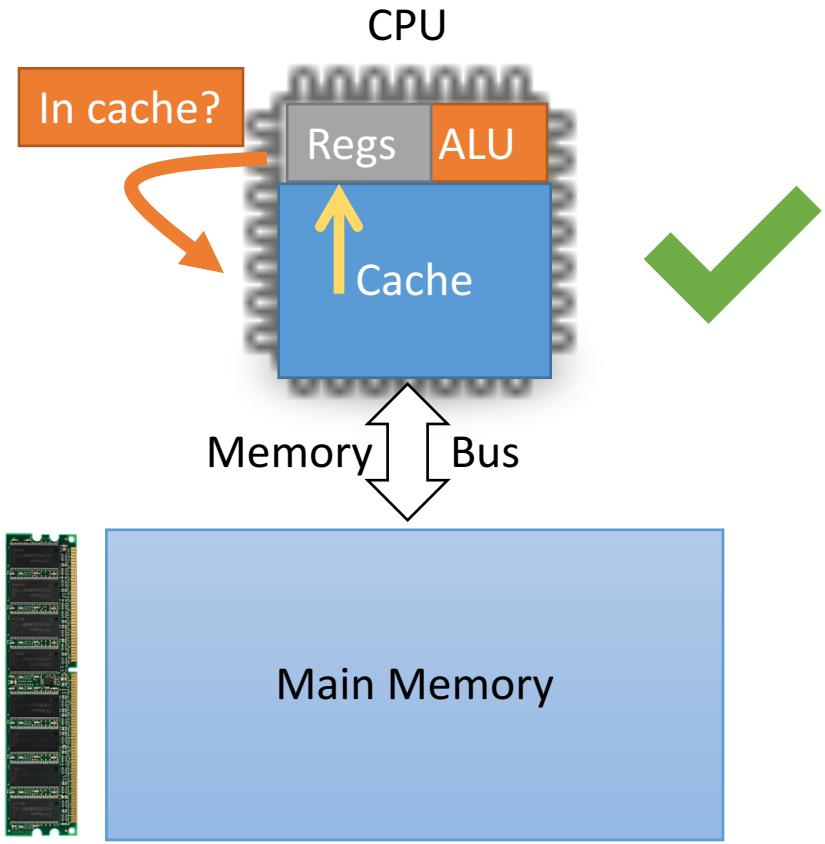
- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- We'll assume one cache (same principles)

Cache Basics: Read from memory



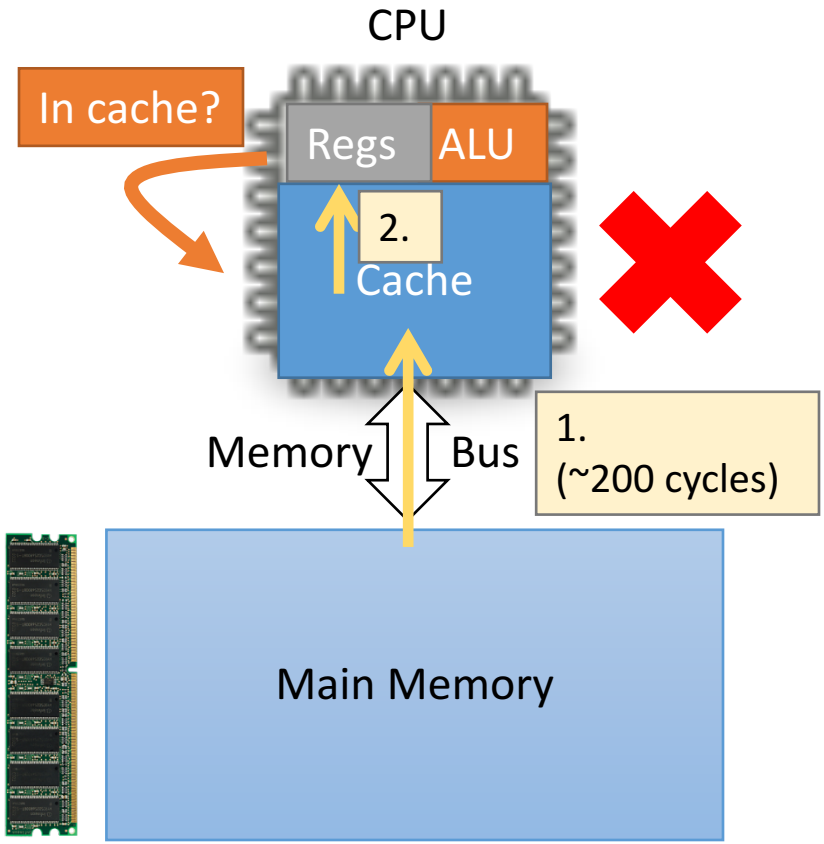
- In parallel:
  - Issue read to memory
  - Check cache

Cache Basics: Read from memory



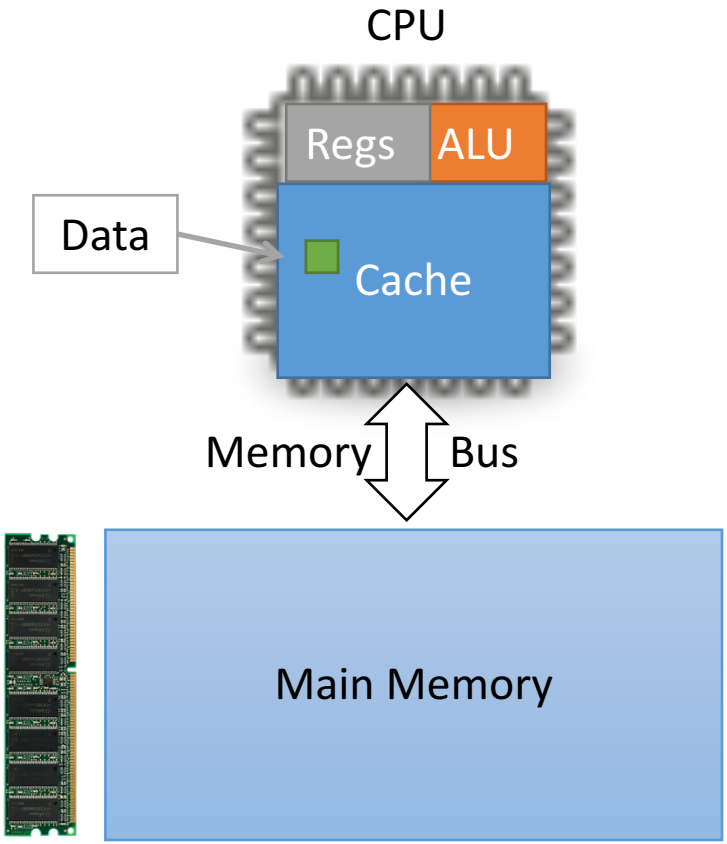
- In parallel:
  - Issue read to memory
  - Check cache
- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory

Cache Basics: Read from memory



- In parallel:
  - Issue read to memory
  - Check cache
- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory
- Data not in cache (miss):
  1. Load cache from memory (might need to evict data)
  2. Send to register

Cache Basics: Write to memory



- Assume data already cached
  - Otherwise, bring it in like read

1. Update cached copy.

2. Update memory?

When should we copy the written data from cache to memory? Why?

A. Immediately update the data in memory when we update the cache.

B. Update the data in memory when we evict the data from the cache.

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)

When should we copy the written data from cache to memory? Why?

A. Immediately update the data in memory when we update the cache. ("Write-through")

B. Update the data in memory when we evict the data from the cache. ("Write-back")

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

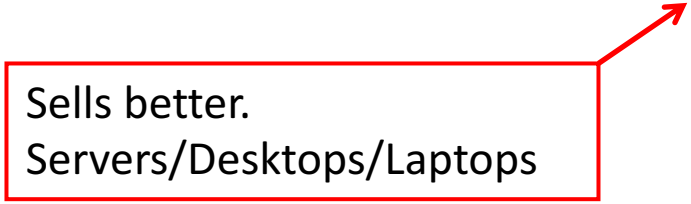
D. Update the data in memory at some other time. (When?)

Cache Basics: Write to memory

- Both options (write-through, write-back) viable

- write-though: write to memory immediately
  - simpler, accesses memory more often (slower)

- write-back: only write to memory on eviction
  - complex (cache inconsistent with memory)
  - potentially reduces memory accesses (faster)



Discussion Question

What data should we keep in the cache?

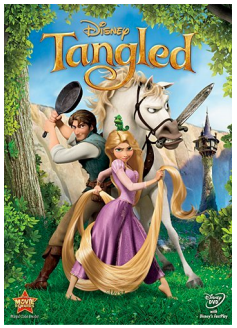
What principles can we use to make a decent guess?

Problem: Prediction

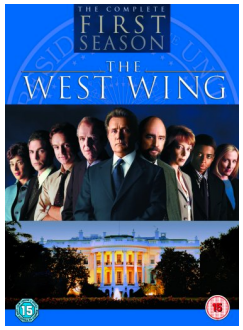
- We can’t know the future…
- So… are we out of luck?
  What might we look at to help us decide?
- The past is often a pretty good predictor…

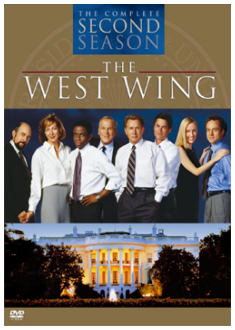
Analogy: two types of Netflix users

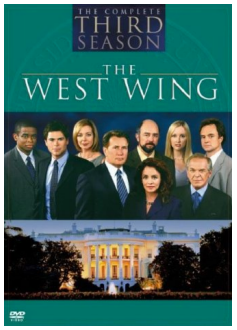
1:



2:







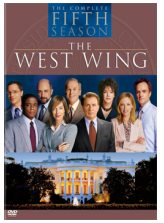
What should be next in each user's queue?

Critical Concept: Locality

- Locality: we tend to repeatedly access recently accessed items, or those that are nearby.

- Temporal locality: An item accessed recently is likely to be accessed again soon.

- Spatial locality: We're likely to access an item that's nearby others we just accessed.



In the following code, how many examples are there of temporal / spatial locality?
Where are they?

```
void print_array(int *array, int num) {
  int i;
  for (i = 0; i < num; i++) {
    printf("%d : %d", i, array[i]);
  }
}
```

A. 1 temporal, 1 spatial
B. 1 temporal, 2 spatial
C. 2 temporal, 1 spatial
D. 2 temporal, 2 spatial
E. Some other number

Example

```
void print_array(int *array, int num) {
  int i;
  for (i = 0; i < num; i++){
    printf("%d : %d", i, array[i]);
  }
}
```

Temporal Locality?

`array`, `num` and `i` used over and over again in each iteration

Spatial Locality?

`array` bucket access
program instructions

Programs with loops tend to have a lot of locality

and most programs have loops:

it's hard to write a long-running program w/o a loop

Use Locality to Speed-up Memory Access

Caching Key idea: keep copy of "likely to be accessed soon" data in higher levels of Memory Hierarchy to make their future accesses faster:

- recently accessed data (temporal locality)
- data nearby recently accessed data (spatial locality)

If program has high degree of locality, next data access is likely to be in cache

- if little/no locality, then caching won't help

+ luckily most programs have a high degree of locality

Discussion Question

What data should we evict from the cache?

What principles can we use to make a decent guess?