Lecture 12
Memory Hierarchy

CS213 – Intro to Computer Systems

Branden Ghena – Spring 2021

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Reminder: office hours
  - Course staff is amazing!!
  - Can help with topics as well as assignments
- Get started on Homework 3
- And on Attack Lab

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	
9:00 AM	Neil		Neil + Spencer				9:00 AM
9:30 AM							9:30 AM
10:00 AM							10:00 AM
10:30 AM							10:30 AM
11:00 AM		Anthony + Atishay					11:00 AM
11:30 AM							11:30 AM
12:00 PM					Drake		12:00 PM
12:30 PM							12:30 PM
1:00 PM							1:00 PM
1:30 PM							1:30 PM
2:00 PM		Class		Class	Branden	Jacob	2:00 PM
2:30 PM							2:30 PM
3:00 PM							3:00 PM
3:30 PM				Branden			3:30 PM
4:00 PM		Drake + Peter			Huaxuan + Atishay		4:00 PM
4:30 PM							4:30 PM
5:00 PM							5:00 PM
5:30 PM							5:30 PM
6:00 PM	Anthony + Seth	Seth + Spencer					6:00 PM
6:30 PM							6:30 PM
7:00 PM							7:00 PM
7:30 PM							7:30 PM
8:00 PM				Anthony + Peter			8:00 PM
8:30 PM							8:30 PM
9:00 PM			Huaxuan + Jacob	Atishay + Peter			9:00 PM
9:30 PM							9:30 PM
10:00 PM							10:00 PM
10:30 PM							10:30 PM
11:00 PM							11:00 PM

Changing the focus of CS213

- So far in class we've focused on *how* computers do things
  - Represent data
  - Run instructions

- Now we're focusing on *how to improve* those things
  - Secure (last lecture plus some stuff in two weeks)
  - Efficient (today and next week

- As we'll show today, the most important thing to speed up is memory
  - It is possible and hardware does so already
  - Software can be designed to take advantage of this

Today's Goals

- Explore the memory systems available in modern computers
  - Understand capabilities and limitations of each
- Discuss the memory hierarchy
  - How it improves performance through *caching*
- Describe software patterns that caching is designed to support

Outline

- **Technologies and Trends**
- Memory Hierarchy
- Locality of Reference

Storage in a Computer System

- Data can be stored in different places
  - Registers, caches, memory, disk, etc.
  - Caches are temporary storage in processor used to speed up memory access

- Hugely different characteristics
  - Storage size
    - x86-64: 16 registers. 128 B total (not including FP registers, etc.)
    - Memory is measured in GB, disks in TB
  - Latency (i.e., time to access data)
    - Registers are really fast, memory less so, disk is incredibly slow
  - Cost per byte
    - Registers are really expensive, disks are really cheap, memory somewhere in the middle

- Each serves a different purpose in a system
  - Can design systems to get the best of all worlds (in most cases)

Computing timescales

- Assuming 4 GHz processor, Instruction (with registers): 0.25 ns

Jeff Dean (Google AI): "Numbers Everyone Should Know"

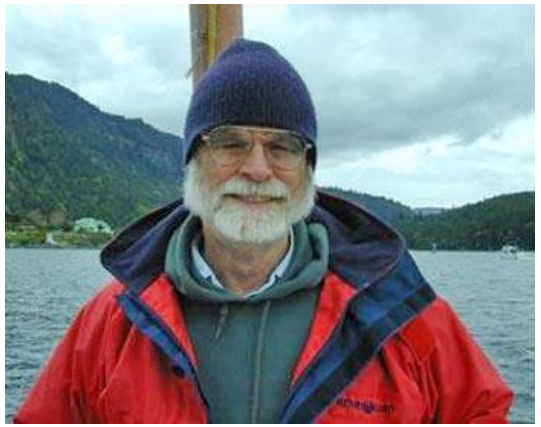
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Jim Gray's storage latency analogy

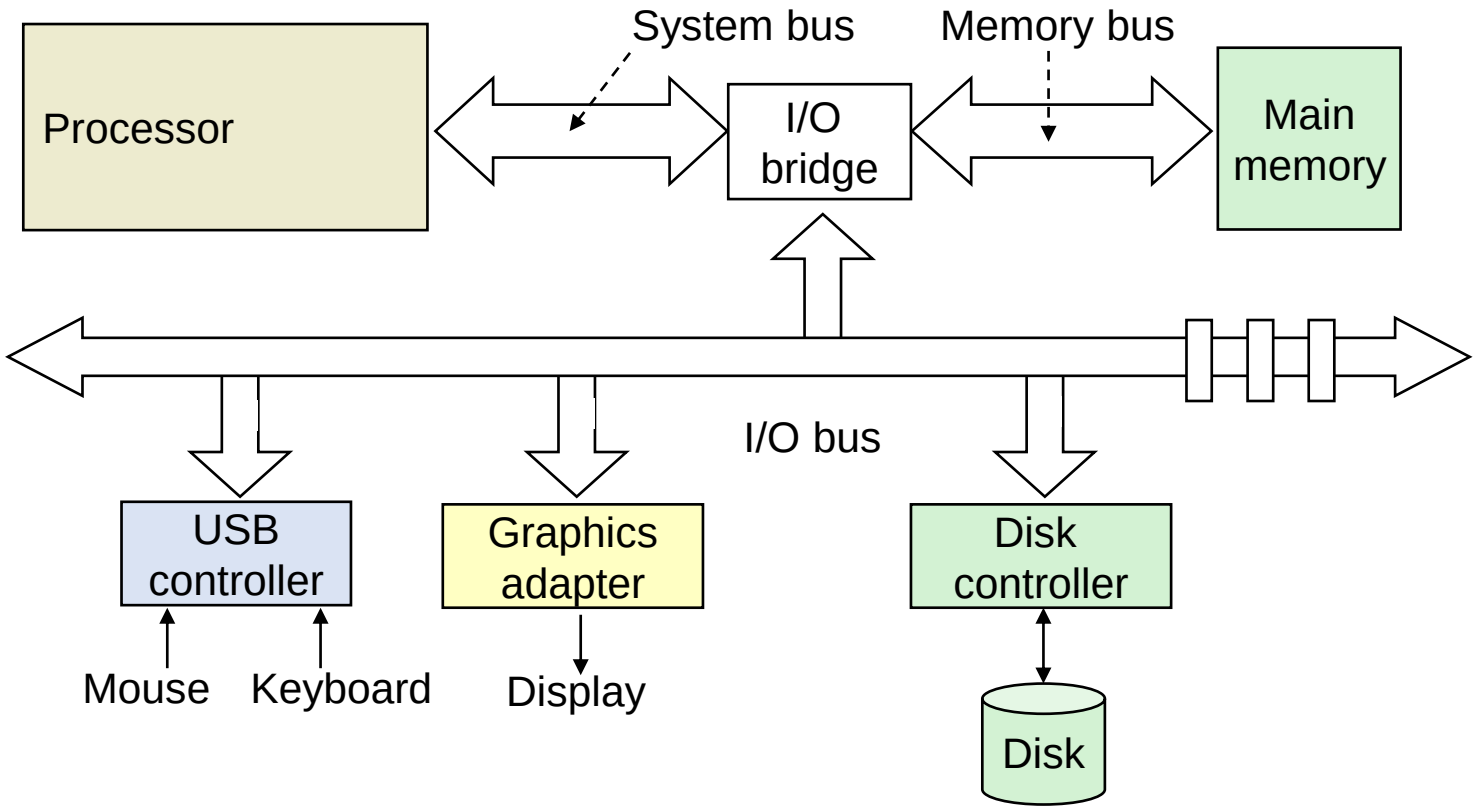
- How "far" is data for the CPU, converted to human scale

Storage	Distance	Time
Registers	In my apartment	1 minute
On-chip cache	Across the street	2-4 minutes
On-board cache	A few blocks away	10 minutes
Main memory	In Milwaukee	1.5 hours
Disk	On Mars	2 years
Tape	On Kepler-76b	2000 years (at speed of light)

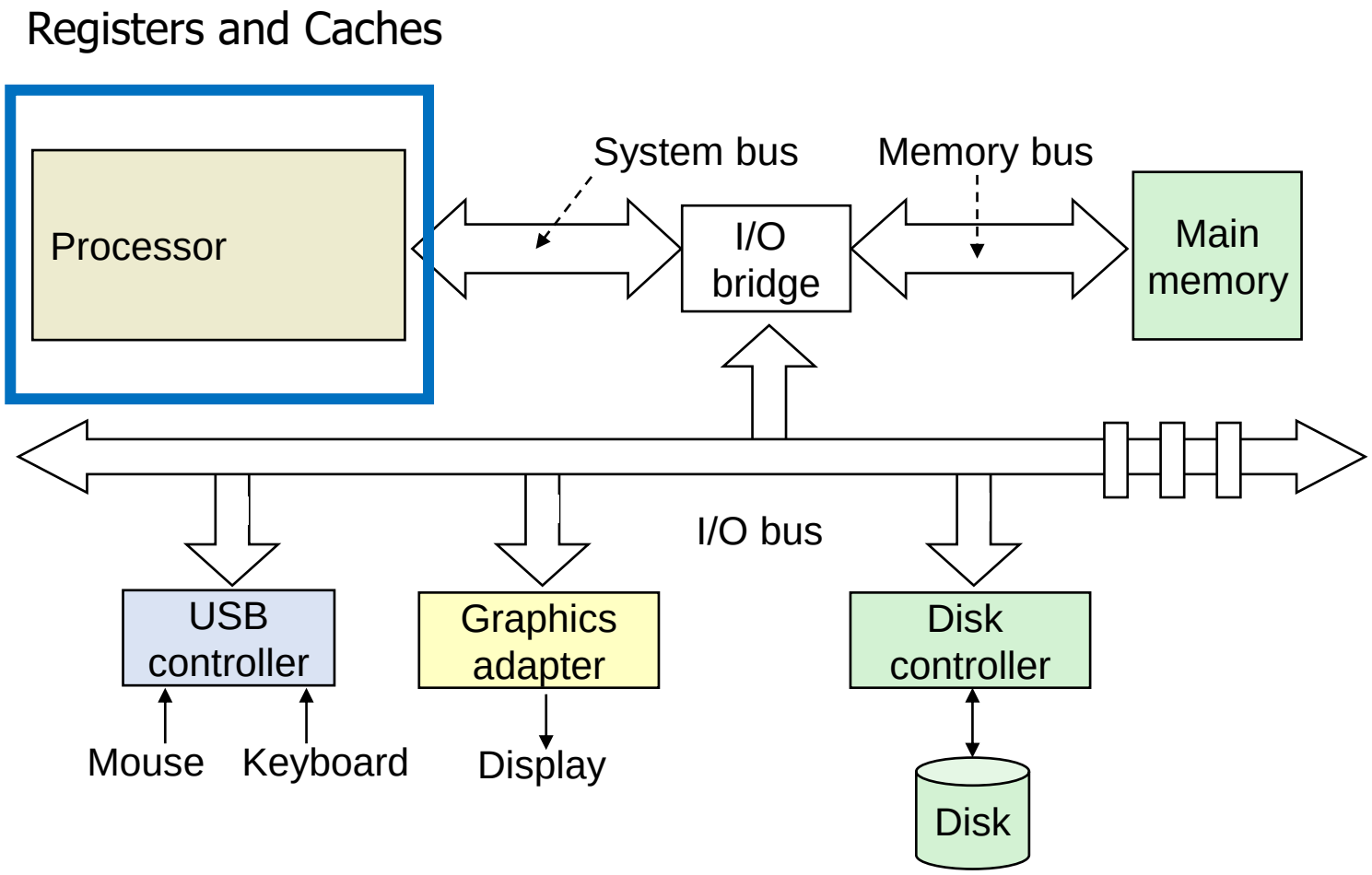
Jim Gray
Turing Award 1998



Tour of computer memory



Tour of computer memory



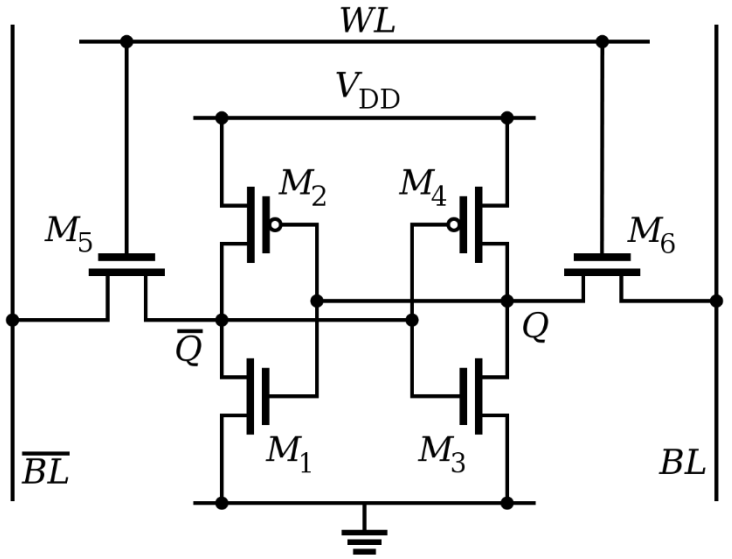
Register storage

- x86-64
  - 16 registers (general-purpose) with 8 bytes each
  - 32 registers (special-purpose) with up to 64 bytes each
  - Plus some other odds and ends (`%rip`, flags, segments, etc.)

- 128 bytes for general purpose registers
- Order 1-2 kB for everything

- Accesses are very fast. Within a single cycle

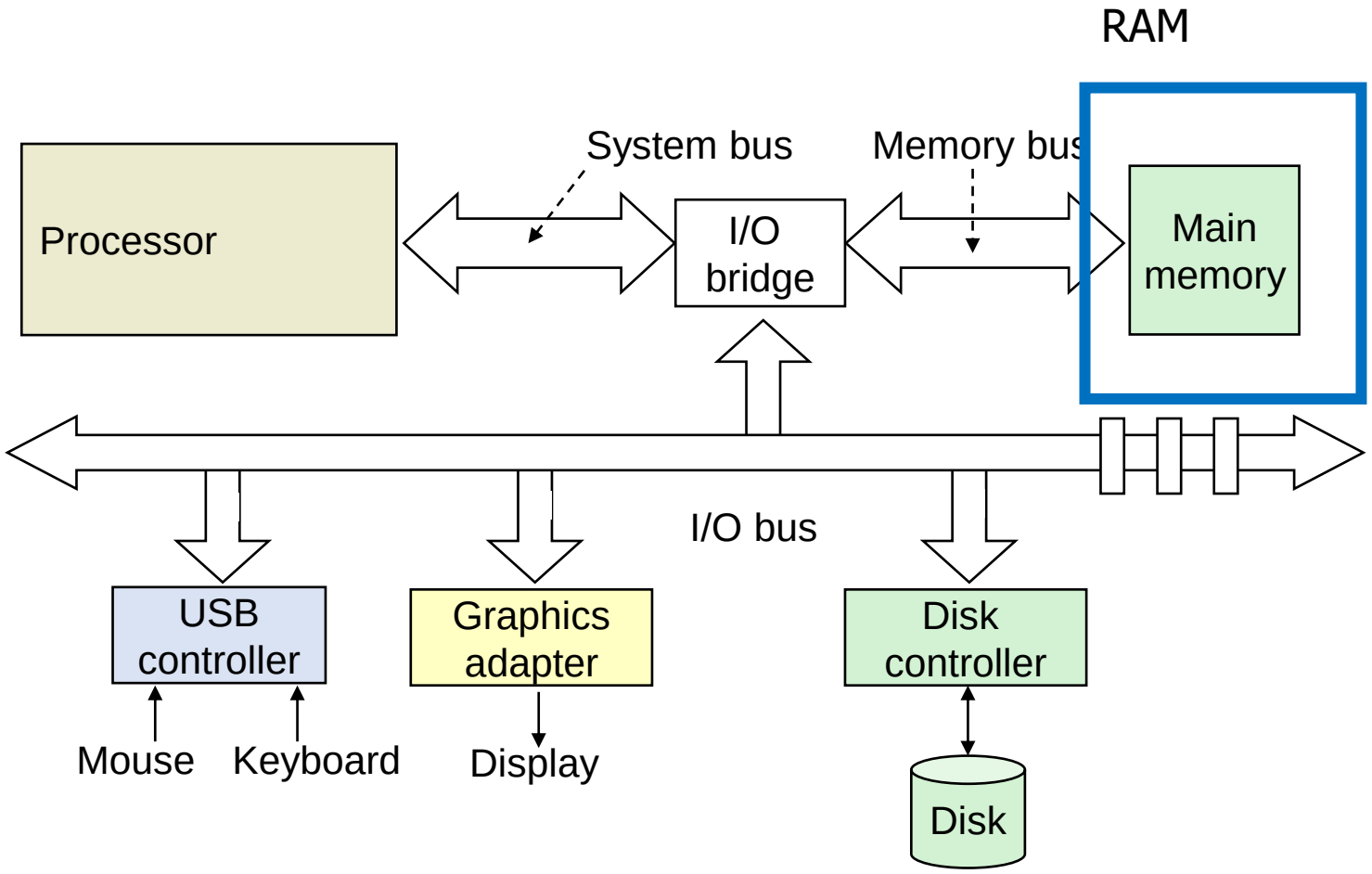
Register technology: SRAM



- Static RAM (SRAM)
  - Each cell stores a bit in a bi-stable circuit, typically a six-transistor circuit
  - Static – no need for periodic refreshing; keeps data while powered
  - Relatively insensitive to disturbances such as electrical noise
    - Energetic particles (alpha particles, cosmic rays) can flip stored bits

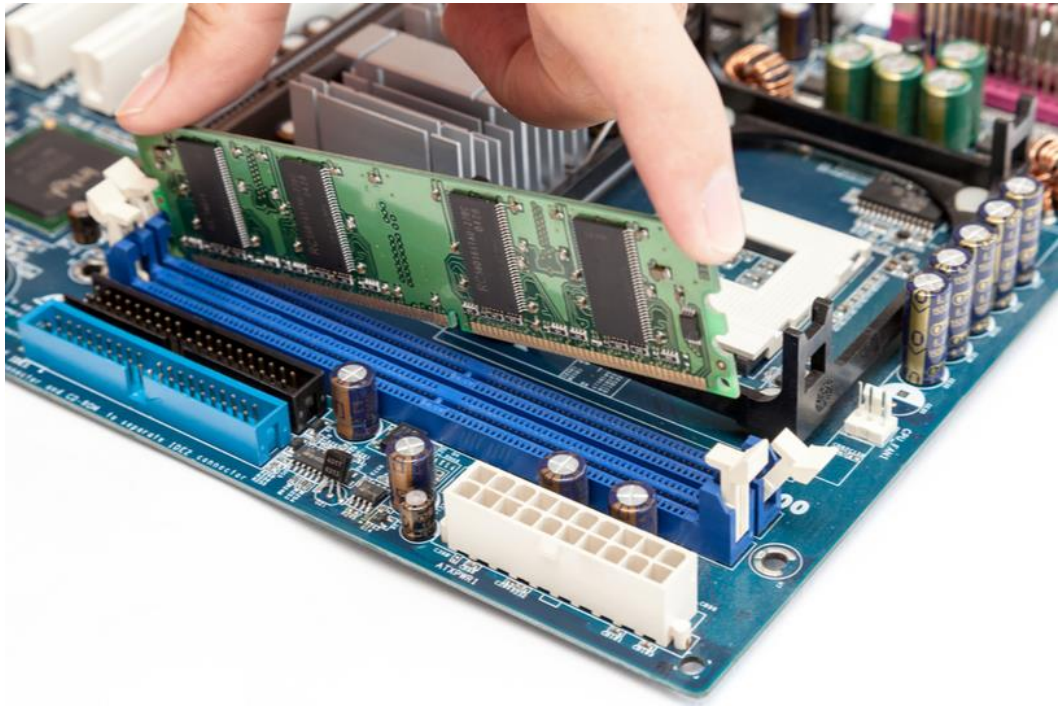
- Fastest memory on computer
  - Also most expensive and takes up most space per bit
  - Typically used for registers and cache memories

Tour of computer memory



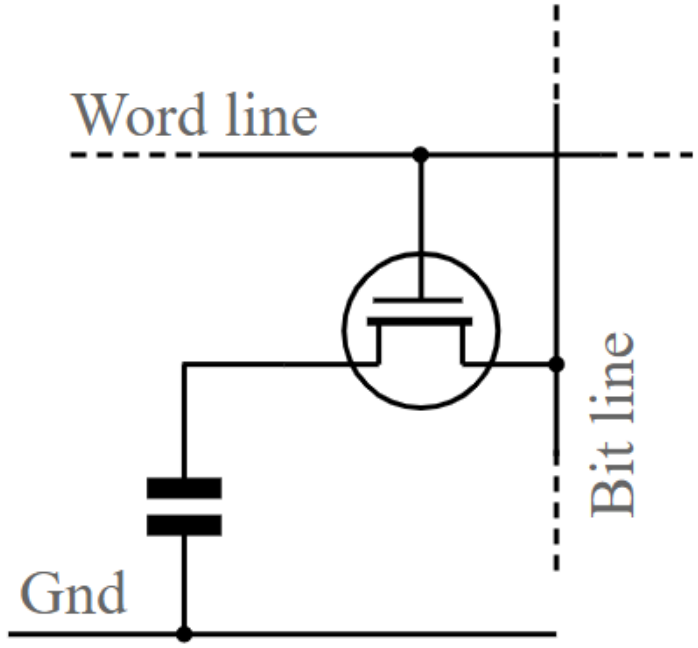
Main memory

- The "RAM" in your computer
  - Random Access Memory
  - Can access any byte you want in "random" order
- Typically measured in GB
  - 1-128 GB
  - Some special purpose systems may have MUCH less
- This is the "array of bytes" we've been using in assembly



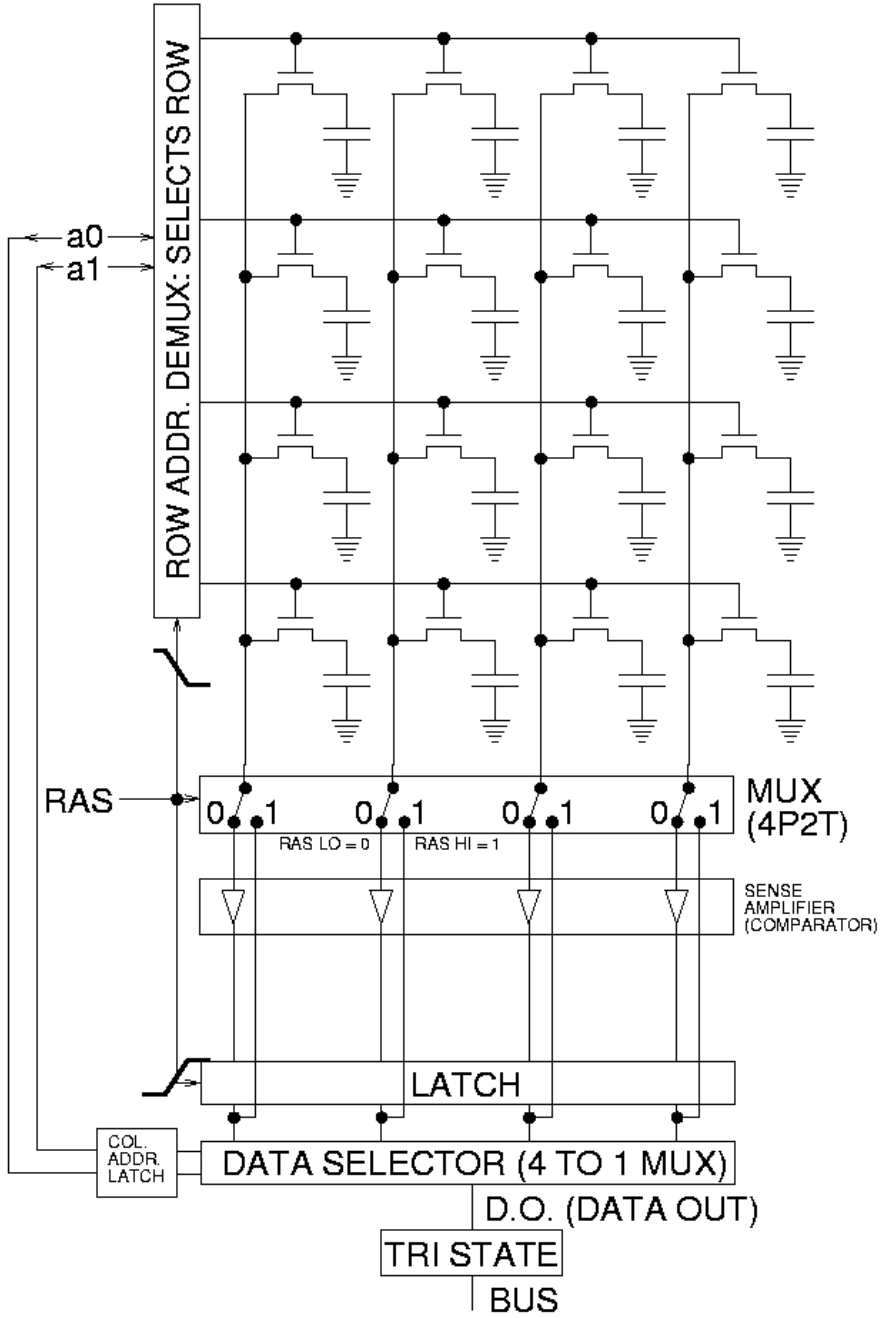
Main memory technology: DRAM

- Dynamic RAM (DRAM)
  - Each cell stores a bit as a charge in a capacitor
  - Capacitors lose charge; each cell must be refreshed every 10-100 ms
  - More sensitive to disturbances (EMI, radiation, …) than SRAM
- Slower than SRAM, but cheaper and denser
  - ~100x slower than registers
- Typically used for main memory



Accessing DRAM

- Read entire row of data at a time
  - Large in practice, kilobytes
- Select actual bytes that are wanted
  - Possibly modifying those bits
- Write row back to memory
  - Must always happen!
  - Reading is destructive



Connecting main memory and the processor

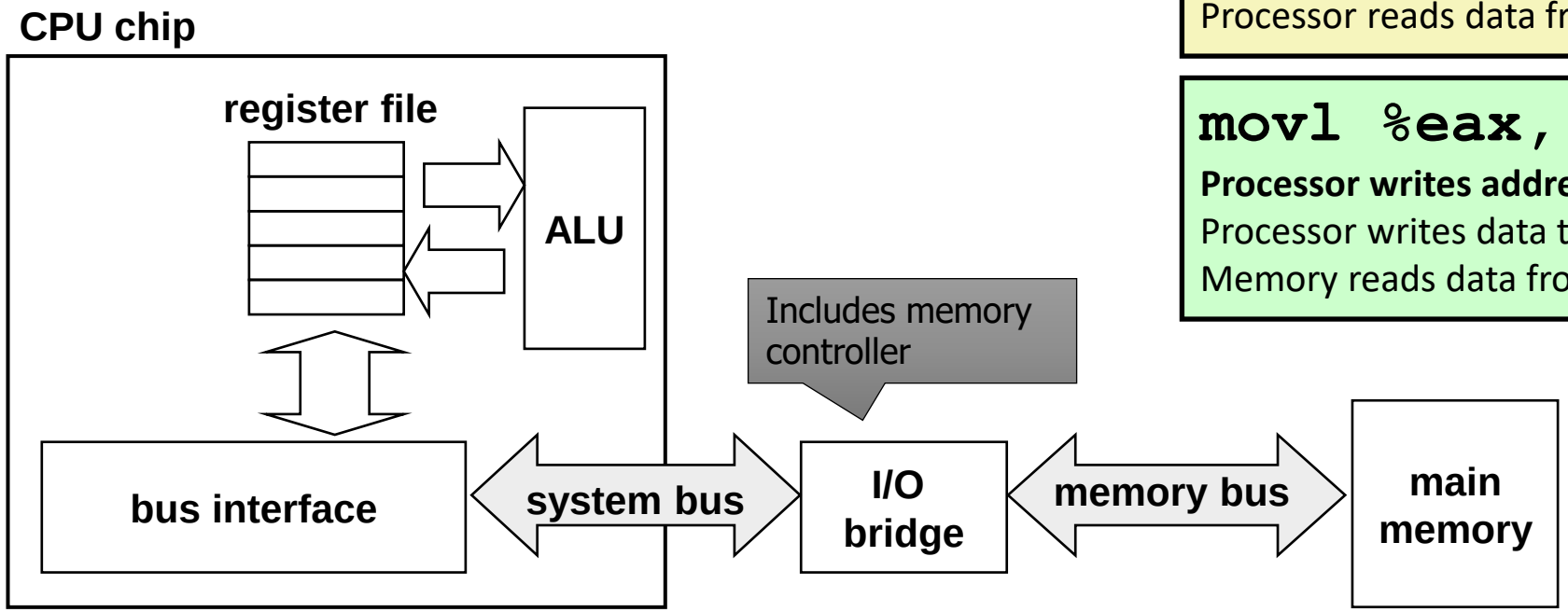
- Data flows between main memory and processor over buses
  - A collection of parallel wires that carry address, data, and control signals
  - Typically shared by multiple devices

`movl A, %eax` (load)

Processor writes address A to the bus
Memory writes data to the bus
Processor reads data from the bus

`movl %eax, A` (store)

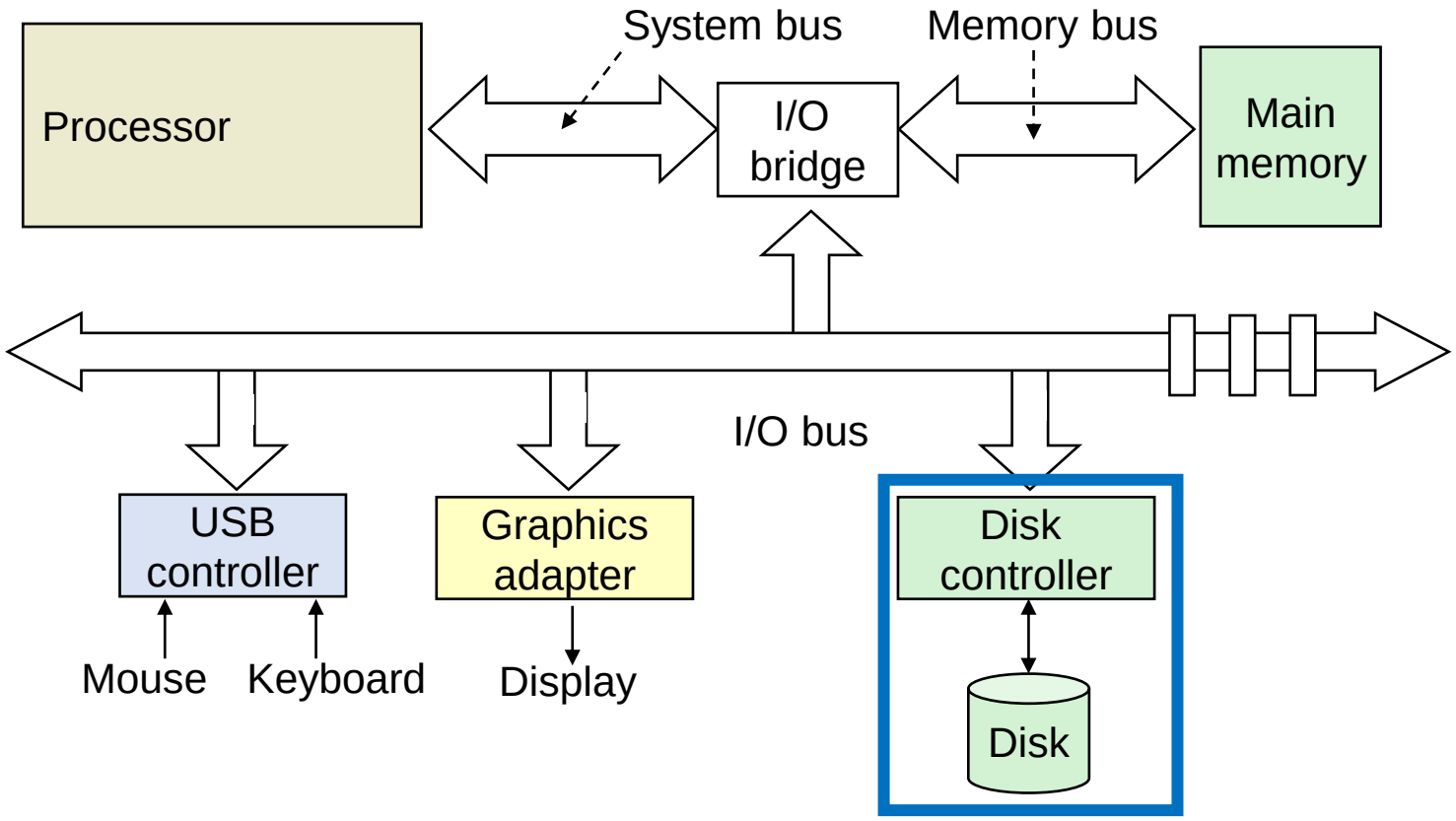
Processor writes address A to the bus
Processor writes data to the bus
Memory reads data from the bus



Sidebar: memory security is important

- Data in RAM disappears without power
  - But the rate depends on temperature, minutes to decay if frozen
  - Cold Boot Attack: freeze RAM to remove from computer and steal contents

Tour of computer memory



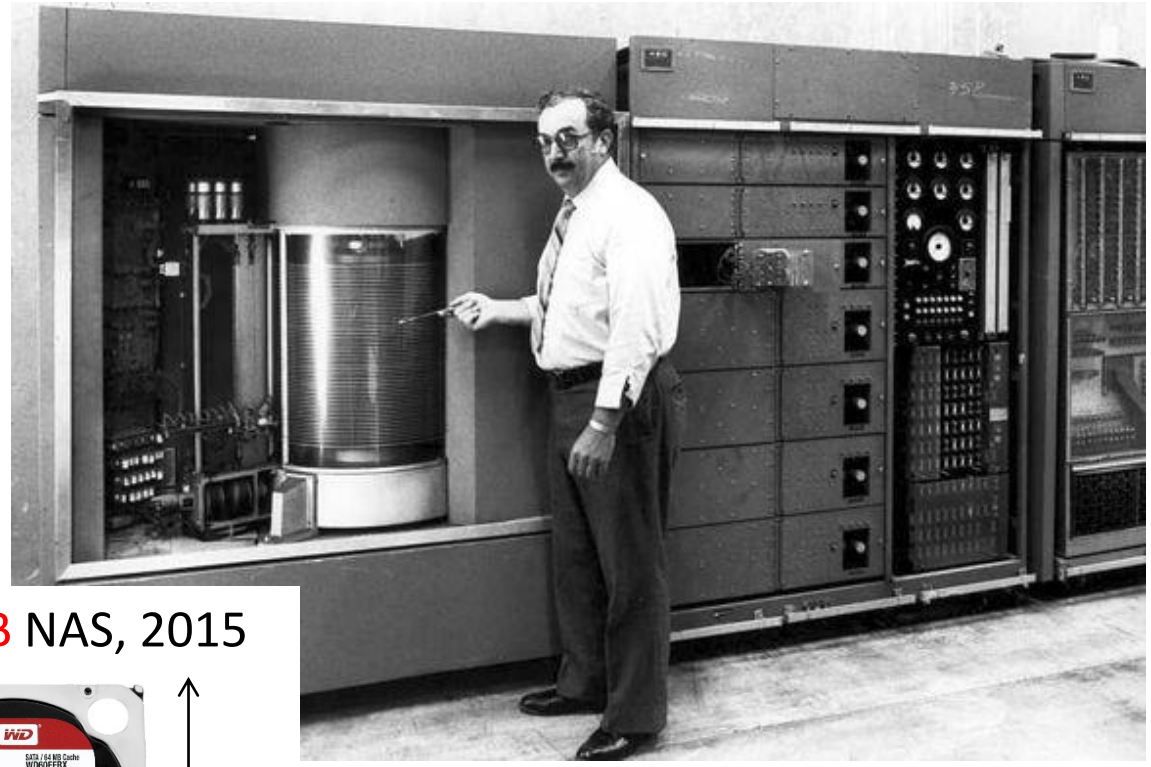
Disk:
Hard drive or SSD

Disk storage

- Workhorse storage devices
  - Terabytes in size in modern computers
  - Milliseconds to read
    - 100,000x longer than reading from DRAM
    - 10,000,000x longer than reading from SRAM

IBM 350 Disk Storage Unit
First disk drive, 1956

Storage?
5MB!



WD Red 6TB NAS, 2015

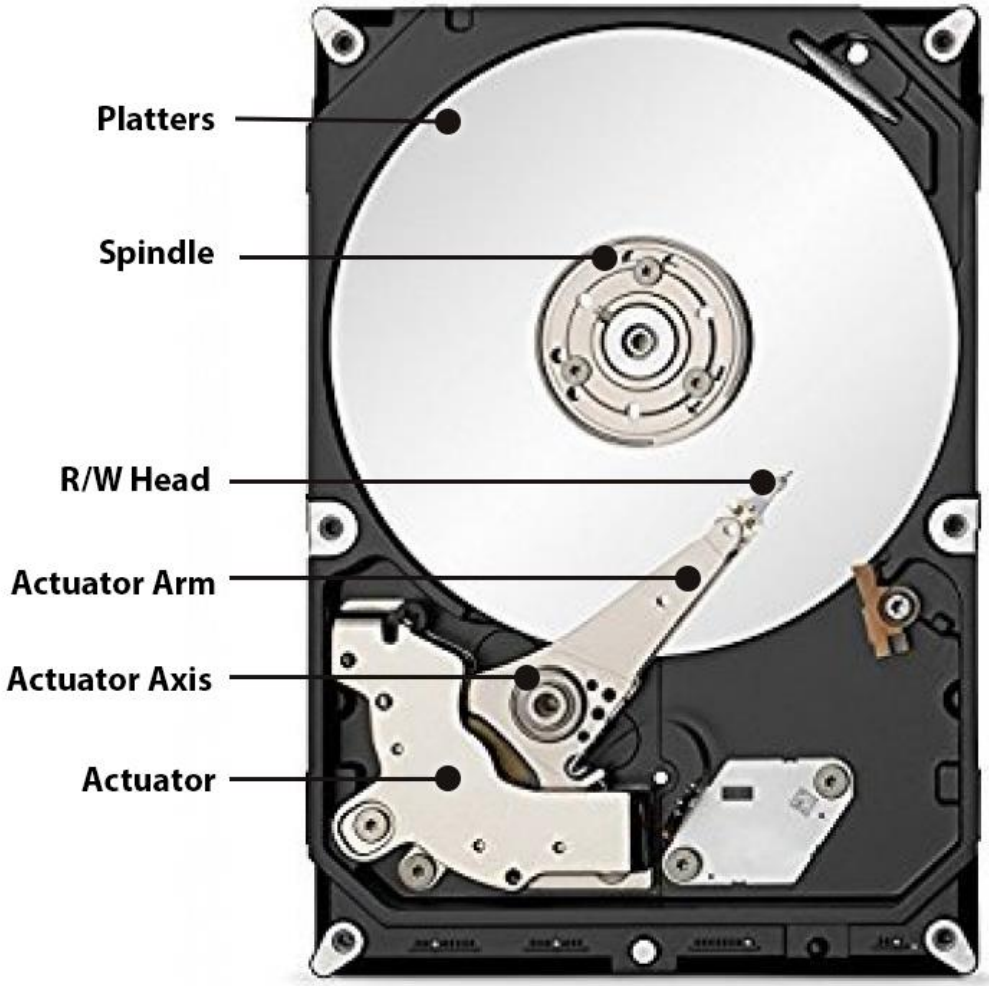
146.99mm

101.6mm

Basic mechanisms the same!

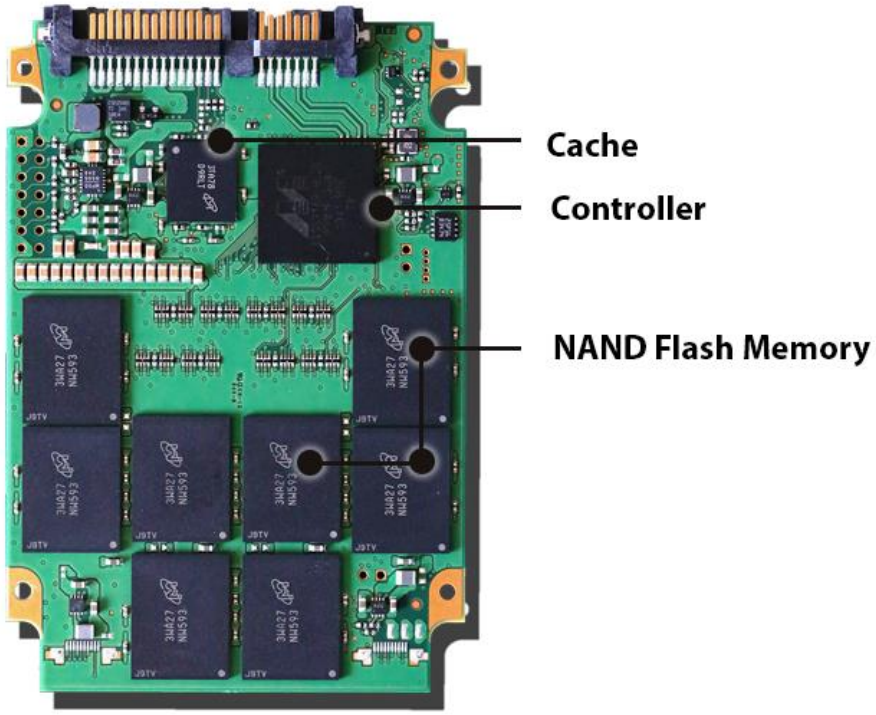
Disk types

HDD
3.5″



Shock resistant up to 55g (operating)
Shock resistant up to 350g (non-operating)

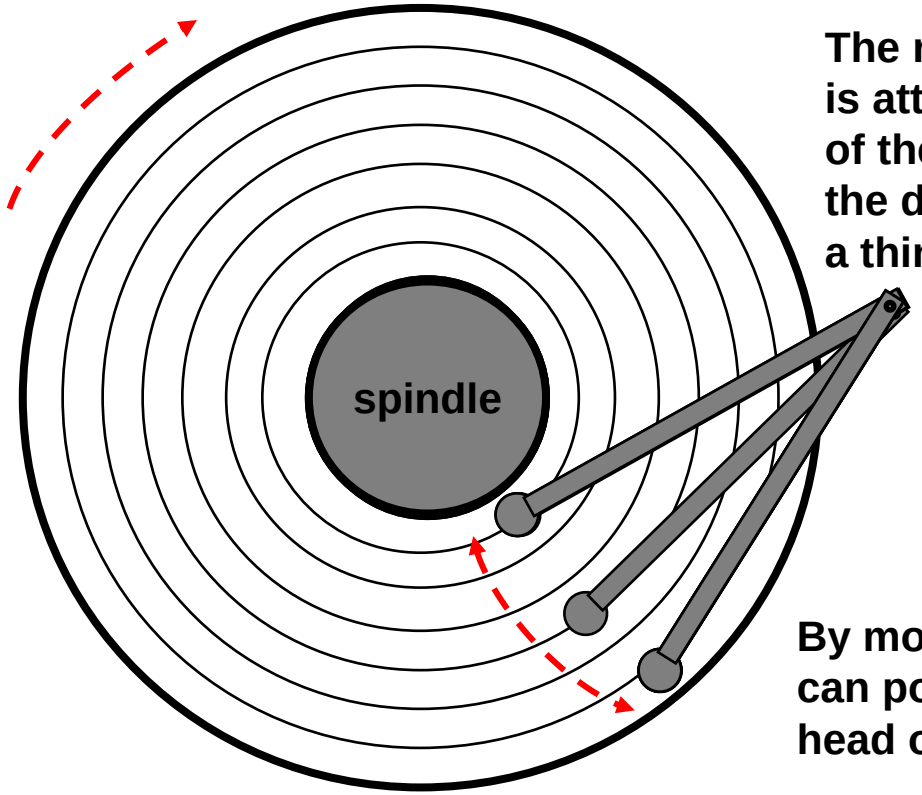
SSD
2.5″



Shock resistant up to 1500g
(operating and non-operating)

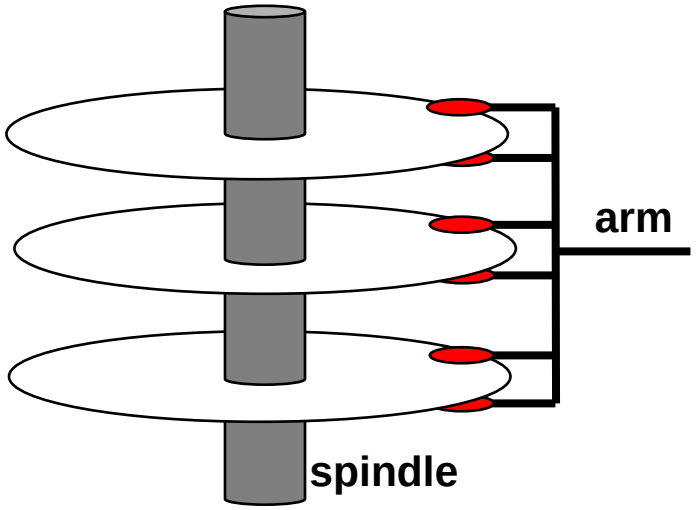
Hard disk drive operation

The disk surface spins at a fixed rotational rate (5400-15000 RPM)



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.



read/write heads move in unison from cylinder to cylinder

Accessing data on disk

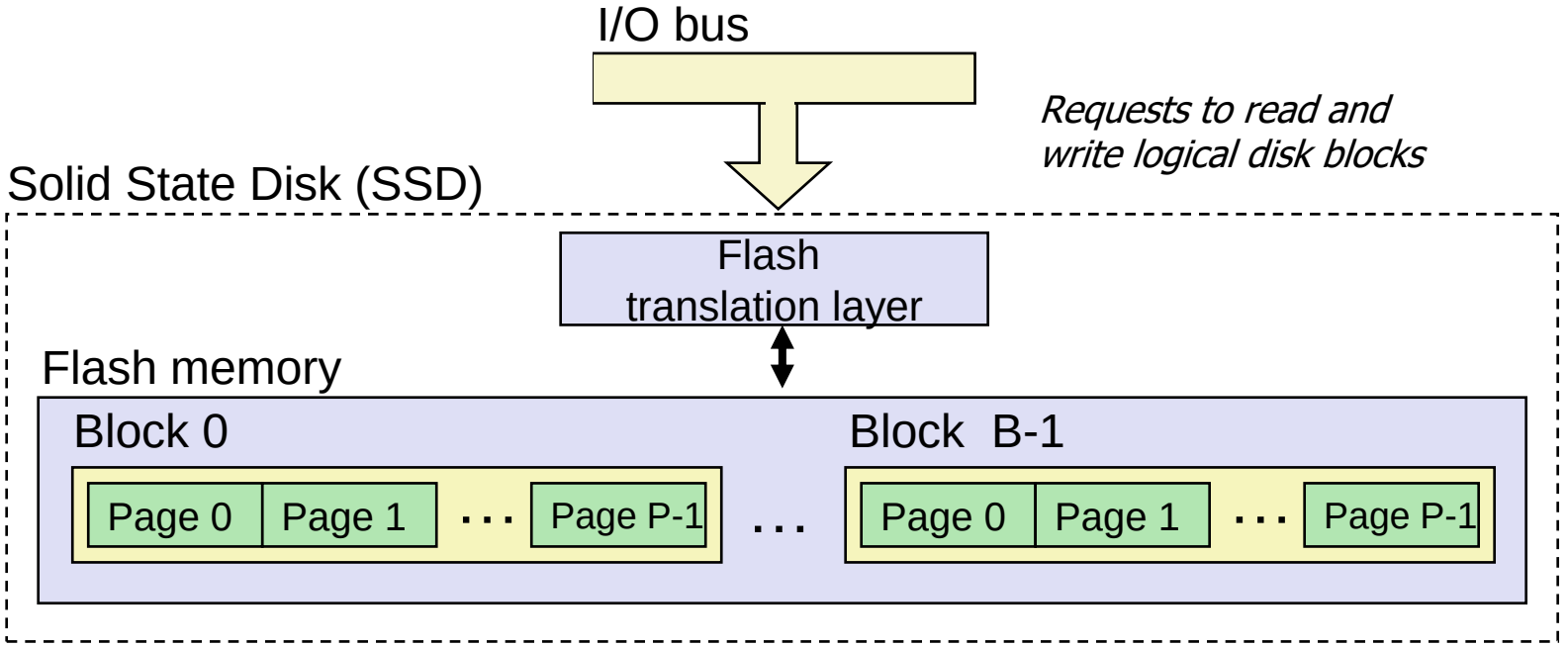
- Three-step process
  - Read head needs to move to the right cylinder: ***seek time***
  - Platter turns until start of sector under the read head: ***rotational latency***
  - Sector passes under read head and data is read: ***transfer time***

- Time cost dominated by seek time and rotational latency
  - **Consequence:** reading first bit of a sector is expensive
  - But reading the rest of the sector is basically free!

- When using disks, best to favor large sequential reads/writes
  - Terrible for random access! (reading a little bit here, a little bit there)
  - But overall still really slow compared to main memory or registers

Solid State Drives (SSDs)

- Flash memory
  - Just like Flash Drives



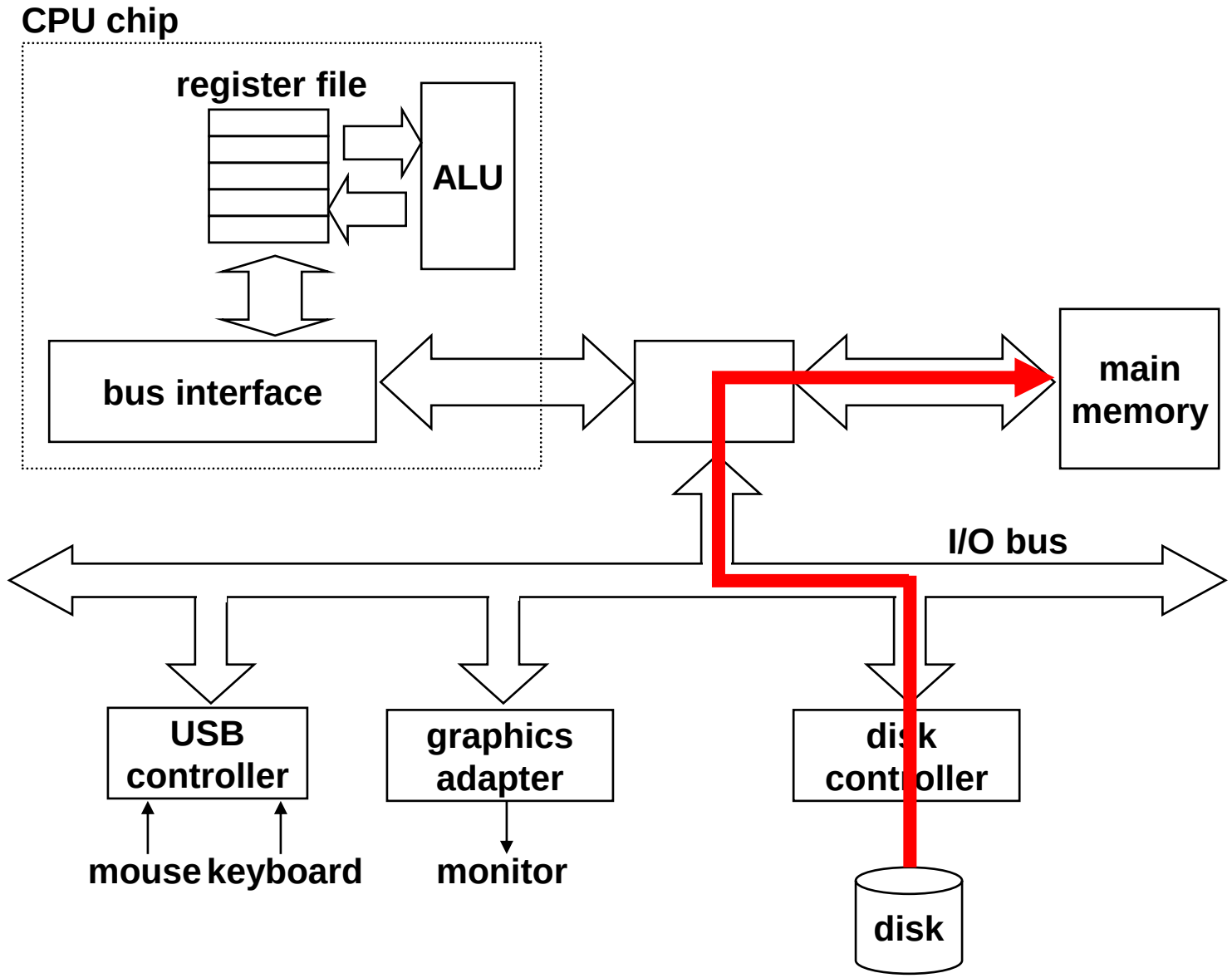
- Pages: 2KB to 512KB, Blocks: 32 to 128 pages
- Data read/written in units of pages
- Page can be written only after its block has been erased
  - Need to copy rest of the data to not lose it. Expensive!
- A block wears out after repeated writes (10k – 100k writes) and can no longer be used

SSDs vs Rotating Disks

- Advantages
  - No moving parts → faster, less power, more rugged
- Disadvantages
  - Have the potential to wear out
    - Mitigated by "wear leveling logic" in flash translation layer
    - Order petabyte (10^{15} bytes) of random writes before they wear out
  - More expensive per byte (but getting cheaper)
    - 2021: HDD $0.02 per GB, SSD $0.09 per GB
- Applications
  - Portable electronics (phones, tablets, etc.)
  - Began to appear in desktops and servers circa 2007
  - Now common on laptops as well

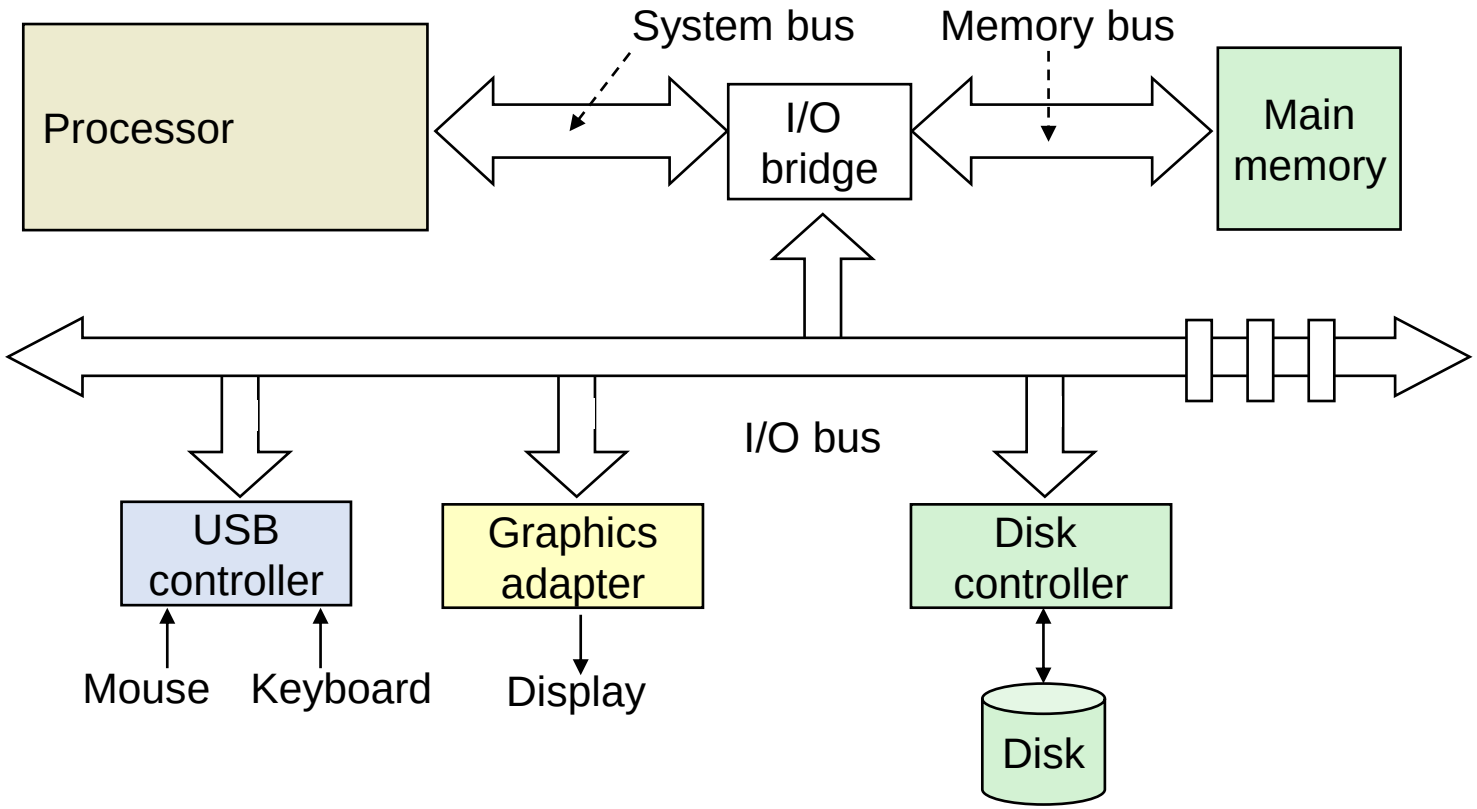
Biggest speed improvement to your computer:
- Get an SSD

Reading memory from disk



- Data from disk is always read into Main Memory
- Direct Memory Access (DMA)
  - Processor starts a read and then returns to programs
  - Disk performs the read, transfers data, then notifies processor when done

Tour of computer memory



Break + Question

- How do you make an SSD with a longer lifetime (more writes)?
  - Without changing any of the physics of how it works

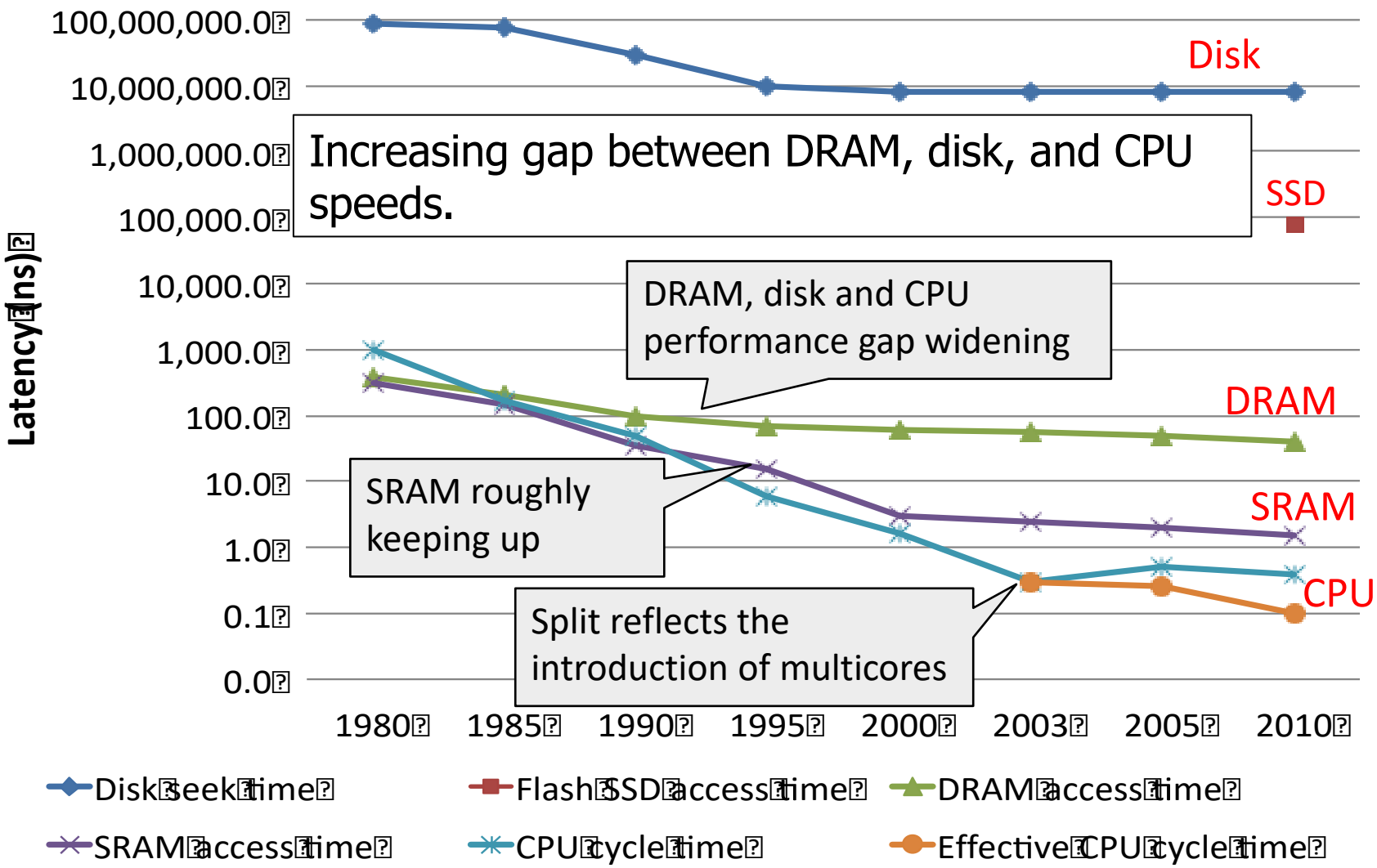
Break + Question

- How do you make an SSD with a longer lifetime (more writes)?
  - Without changing any of the physics of how it works
  - Secretly make it larger than it claims to be
    - e.g. 200 GB when it claims to be 100 GB
  - Behind the scenes move around memory as necessary so the device can still hold 100 GB even if half of the flash is
    - Maintain a mapping of which data is located where

Outline

- Technologies and Trends
- **Memory Hierarchy**
- Locality of Reference

The CPU-Memory gap



The CPU-Memory Gap

- CPUs outspeed memory
  - But they can't compute on data they don't have!
  - If the CPU has to wait for data to reach it, it just sits idle!
    - All these GHz don't look so useful anymore, do they?

- Challenge: get data to the CPU despite "slow" memory
  - So the CPU can work at full speed, without waiting for data

- Two-pronged strategy
  - ***Memory hierarchy:*** keep data we need closer to the CPU
  - ***Locality of reference:*** predict which data we're likely to need

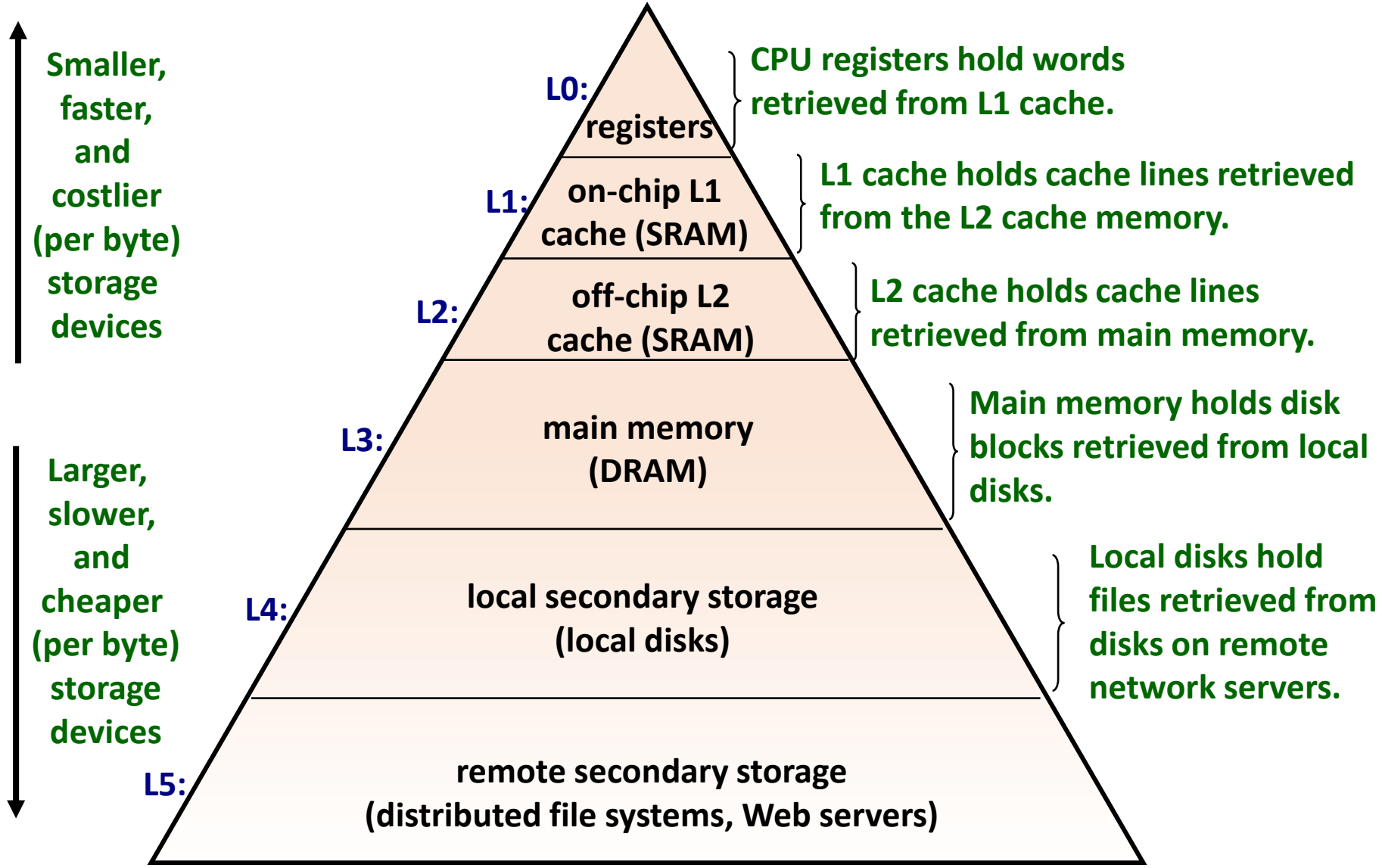
Memory hierarchy

- Some fundamental and enduring properties of systems
  - The faster the storage, the more expensive ($) it is
  - The faster the storage, the smaller (capacity) it is
  - The gap between processor and main memory speed is widening

- Key idea: keep the data you need the most in fast memory!
  - Data you only need from time to time can be in slow memory, no big deal
  - Most used data goes in registers
  - Least used data goes to disk

- Analogy: kitchen ingredients I use
  - Salt, all the time: it sits out on the counter
  - Oregano, frequently: front of the cabinet
  - Onion powder, occasionally: back of the cabinet
  - Brown sugar, sometimes: somewhere in the pantry
  - Saffron, never: I can go buy some if I do need it

Memory hierarchy



Caching, general principle

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
  - Data lives in both places (typically)
  - When the consumer (e.g., CPU) reads the data, it gets it from the smaller, faster storage
  - If the data we want is not in the cache, we pay the full cost of bringing it over from the larger, slower storage into the smaller, faster storage
    - The hope: we don't need to do it too often
- Fundamental idea in systems. Shows up all over!
  - Memory hierarchies
  - Content delivery networks (CDNs) on the Internet (Akamai, Cloudflare, etc.)

Caching in a memory hierarchy

- Fundamental idea of a memory hierarchy
  - For each **k**, the faster, smaller device at level **k** serves as a cache for the larger, slower device at level **k+1**
    - L2 cache memory as a cache for main memory
    - Main memory as a cache for disk, etc.

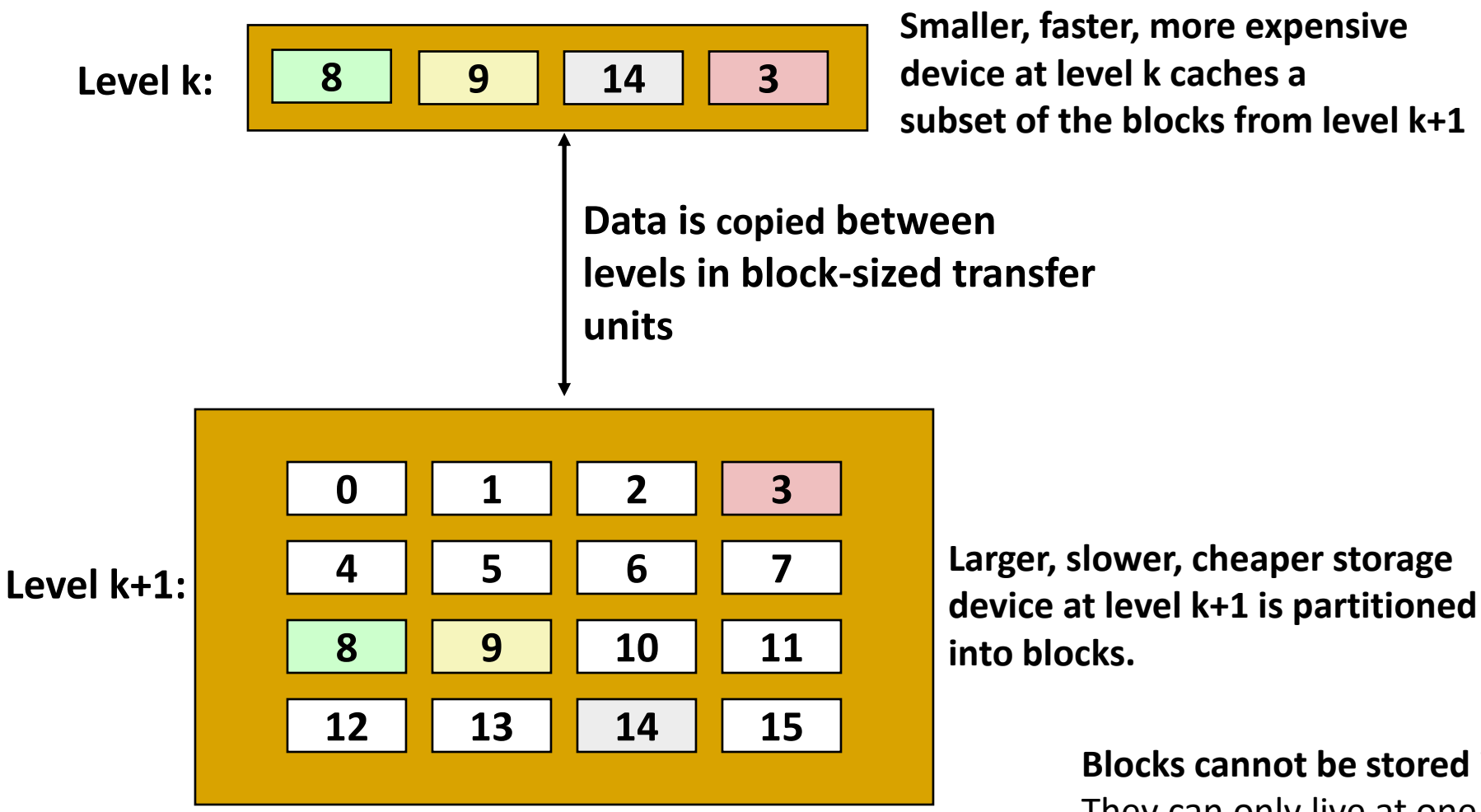
  - Each level stores some of the most frequently accessed data
  - The closer the cache is to the processor, the "hotter" the cached data

Memory hierarchies make memory fast and large

- Why do memory hierarchies work?
  - Programs tend to access the data at level **k** more often than they access the data at level **k+1**
  - Thus, the storage at level **k+1** can be slower, and thus larger and cheaper per bit

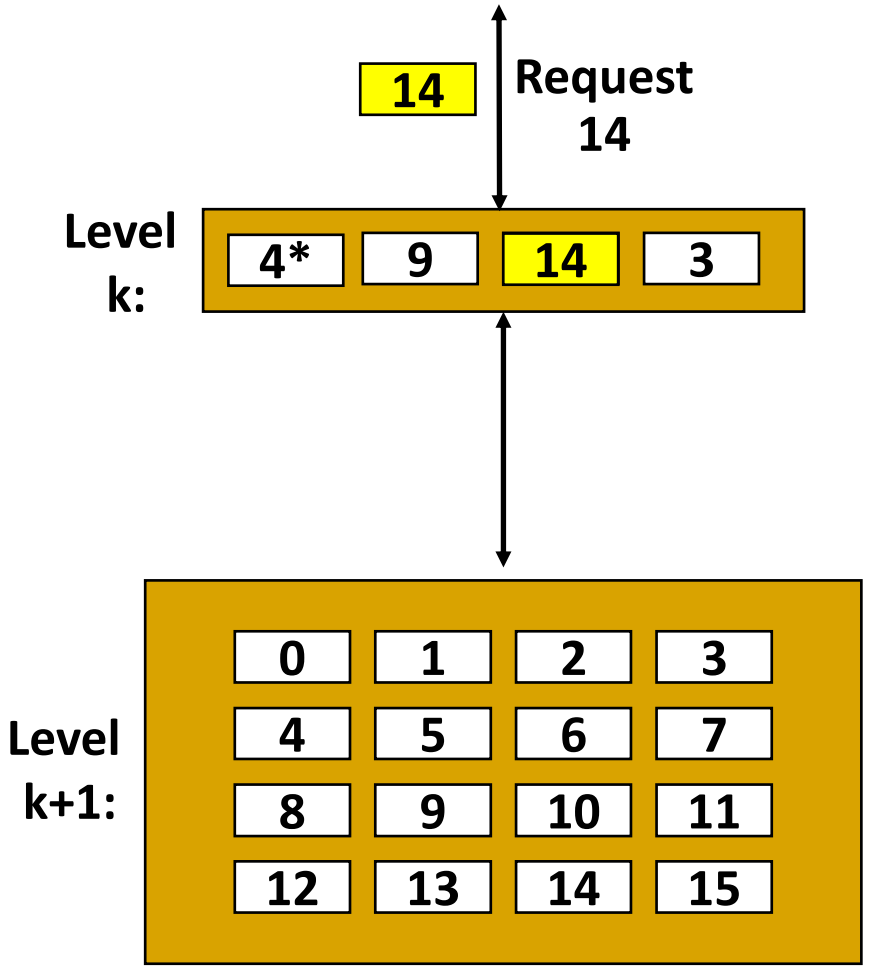
- Net effect:
  - A large pool of memory that costs as little as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top
  - Best of both worlds!

Caching in a memory hierarchy



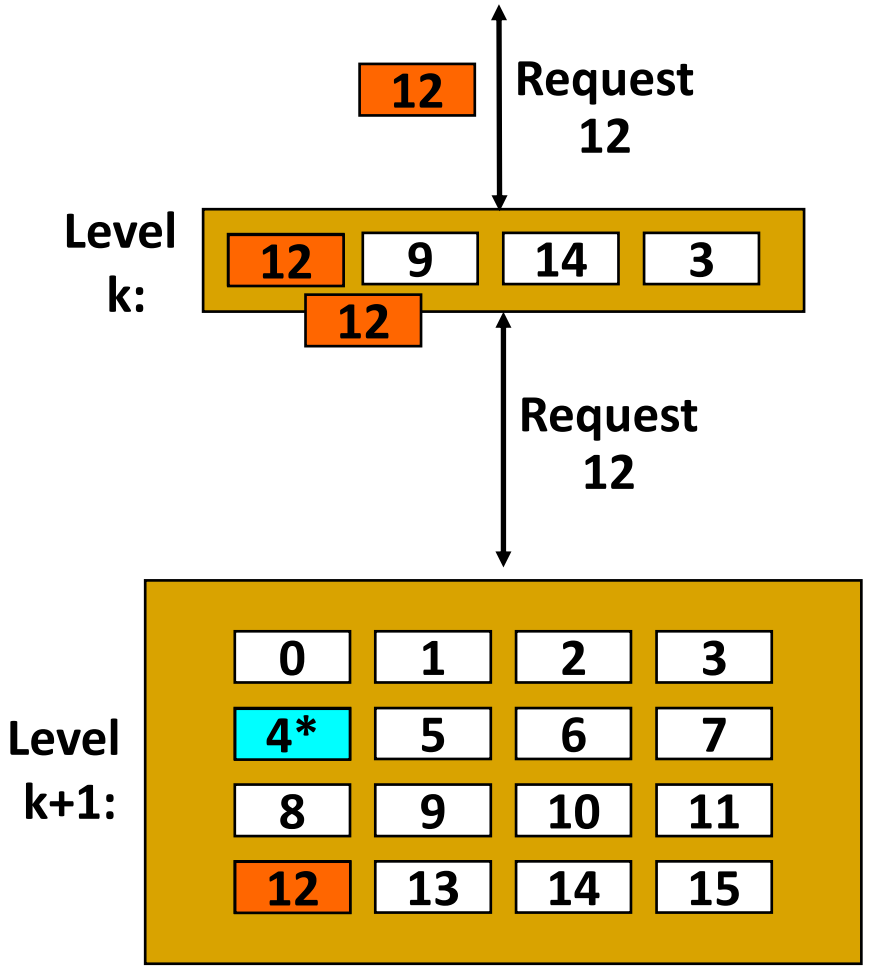
Blocks cannot be stored in an arbitrary location!
They can only live at one of a fixed set of locations.
In this example: they must be in the same "column" for both levels.

General caching concepts



- Program needs object **d**, which is stored in some block **b**
- **Cache hit**
  - Program finds **b** in the cache at level **k**
    e.g., block 14

General caching concepts



" * " means the block is *dirty*
(i.e., it has been modified)

- Program needs object **d**, which is stored in some block **b**
- **Cache hit**
  - Program finds **b** in the cache at level **k**
    e.g., block 14
- **Cache miss**
  - **b** is not at level **k**, so the level **k** cache must fetch it from level **k+1**,
    e.g., block 12
  - If the level-k cache is full, then some current block must be replaced (**evicted**). Which one is the "victim"?
    - Here, we pick 4; same column as 12
    - 4 is "dirty", need to write back to k+1
    - More on this next lecture

Cache Misses Taxonomy

- **Cold (compulsory) miss**
  - Cold misses occur when a block is accessed for the first time
  - No one ever accessed it, so there wasn't any reason to bring it into cache

- **Capacity miss**
  - Occurs when the set of active cache blocks (*working set*) is larger than the cache
  - There's no way the working set can all fit in the cache, so there will be misses

- **Conflict miss**
  - In most caches, blocks cannot be stored in any available slot
  - If two blocks need to go in the same slot, need to evict the old one to store the new!
  - If after that, we need to access the old block, conflict miss!
    - We had a conflict, evicted a block, and now we miss trying to access that block
  - **Note**: can happen even when there is "room" elsewhere in the cache!

Break + Video

- How do you remember which cache miss is which?
  - Mr. Bean can help you tell the difference! (video)

Outline

- Technologies and Trends
- Memory Hierarchy
- **Locality of Reference**

Locality

- Goal: predict which data the CPU will want to access
  - So we can bring it to (and keep it in!) fast memory
  - Problem: memory is huge! (billions of bytes) how do you decide which to save?

- Principle of Locality
  - Programs tend to reuse/use data items recently used or nearby those recently used

- Temporal locality
  - Recently referenced items are likely to be referenced in the near future

- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time

Types of locality practice

- Temporal locality
  - Recently referenced items are likely to be referenced in the near future
- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- Quiz: what kind of locality?
  - Data
    - Reference array elements in succession: **Spatial locality**
    - Reference sum each iteration: **Temporal locality**
  - Instructions
    - Execute instructions in sequence: **Spatial locality**
    - Cycle through loop repeatedly: **Temporal locality**

Locality example

- Can get a sense for whether a function has good locality just by looking at its memory access patterns
- Does this function have good locality?

```
int sumarrayrows(int a[M][N]){
  int sum = 0;
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      sum += a[i][j];
      }
  }
  return sum;
}
```

Temporal or spatial locality?

Spatial: accesses to array
Temporal: accesses to sum

- ***Yes!***
  - Array is accessed in same row-major order in which it is stored in memory
  - **`a`** through **`a+3`** , **`a+4`** through **`a+7`**, **`a+8`** through **`a+11`**, etc.

Locality example

- Does this function have good locality?

```
int sumarraycols(int a[M][N]){
  int sum = 0;
  for (int j = 0; j < N; j++) {
    for (int i = 0; i < M; i++) {
      sum += a[i][j];
    }
  }
  return sum;
}
```

- ***No!***
  - Scans array column-wise instead of row-wise
  - **a** through **a+3**, then **a+4*N** through **a+4*N+3**, etc.
  - Holy jumping around memory Batman!

- More on that in a couple of lectures

Locality to the Rescue!

- How can we exploit locality to bridge the CPU-memory gap?
  - Use it to determine which data to put in a cache!

- Spatial locality
  - When level *k* needs a byte from level *k+1*, don't just bring one byte
  - Bring neighboring bytes as well!
  - Good chances we'll need them too in the near future

- Temporal locality
  - When you bring something into the cache, try to keep it there
  - Good chances we'll need it again in the near future

- Result: most accesses should be cache hits!
  - Memory system: size of largest memory, with speed close to that of fastest memory

- We'll see how that works in detail next time

Outline

- Technologies and Trends
- Memory Hierarchy
- Locality of Reference