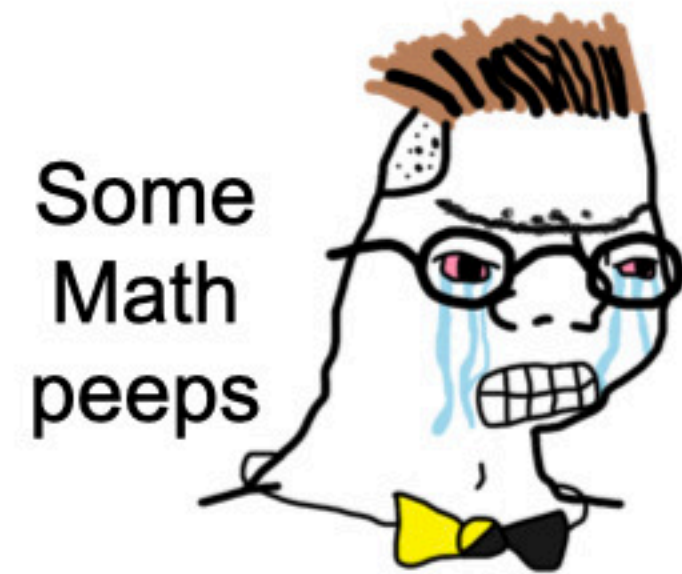


THEN (1980s)



Noooo... CS is just applied math!



Noooo... CS is just applied EE!

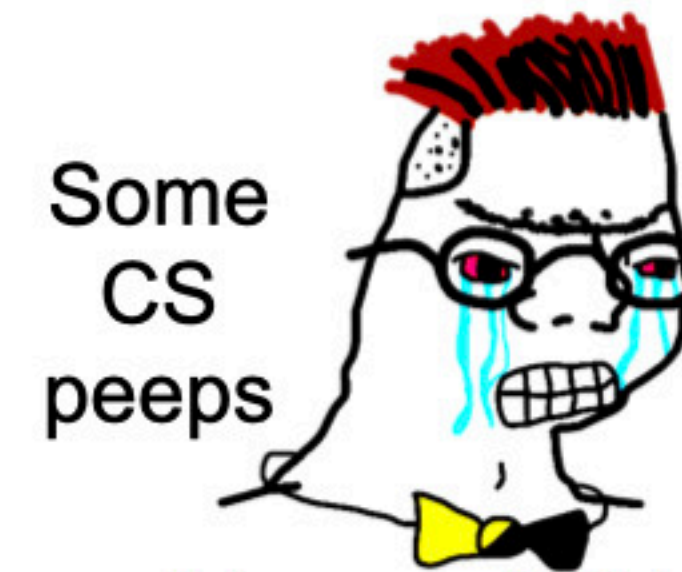


Haha. Computer Science go brrr

NOW (2020s)



Noooo... DS is just applied math!



Noooo... DS is just applied CS!



Haha. Data Science go brrr

DSC 102

Systems for Scalable Analytics

- Haojian Jin

# Review Questions

- Why do computers use binary digits?
- How many integers can you represent with 5 bits?
- How many bits do you need to represent 5 integers?
- What is the hexadecimal for  $20_{10}$ ?
- Why do we need a float standard?
- Why should a data scientist know about float formats?
- What does “lower precision” mean for a float weight in DL?

# Where are we in the class?

- ❖ Basics of Computer Organization
  - ❖ Digital Representation of Data
  - ❖ **Processors and Memory Hierarchy**
- ❖ Basics of Operating Systems
  - ❖ Process Management: Virtualization; Concurrency
  - ❖ Filesystem and Data Files
  - ❖ Main Memory Management
- ❖ Persistent Data Storage

# Today

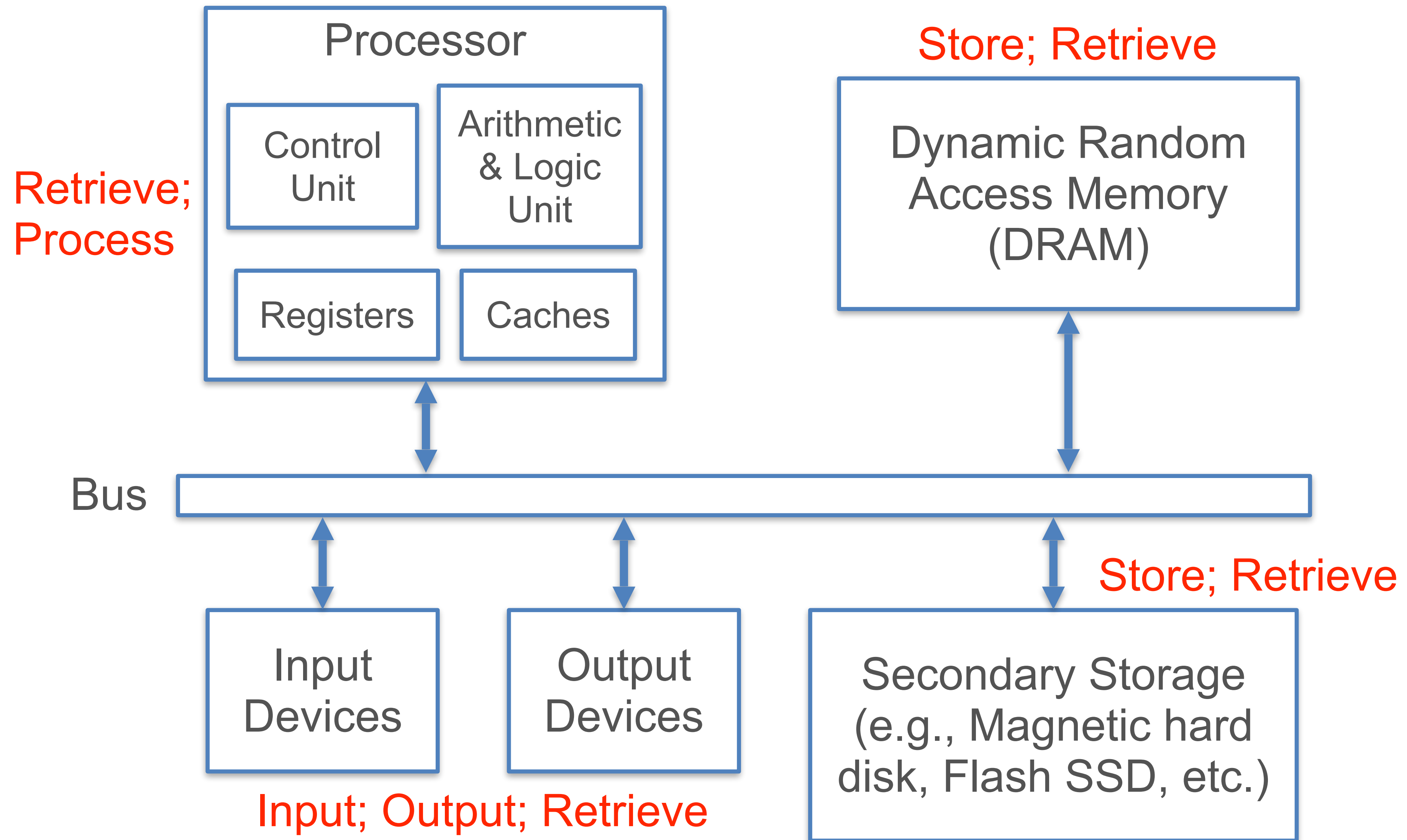
**The memory abstraction**

Locality of reference

The memory hierarchy

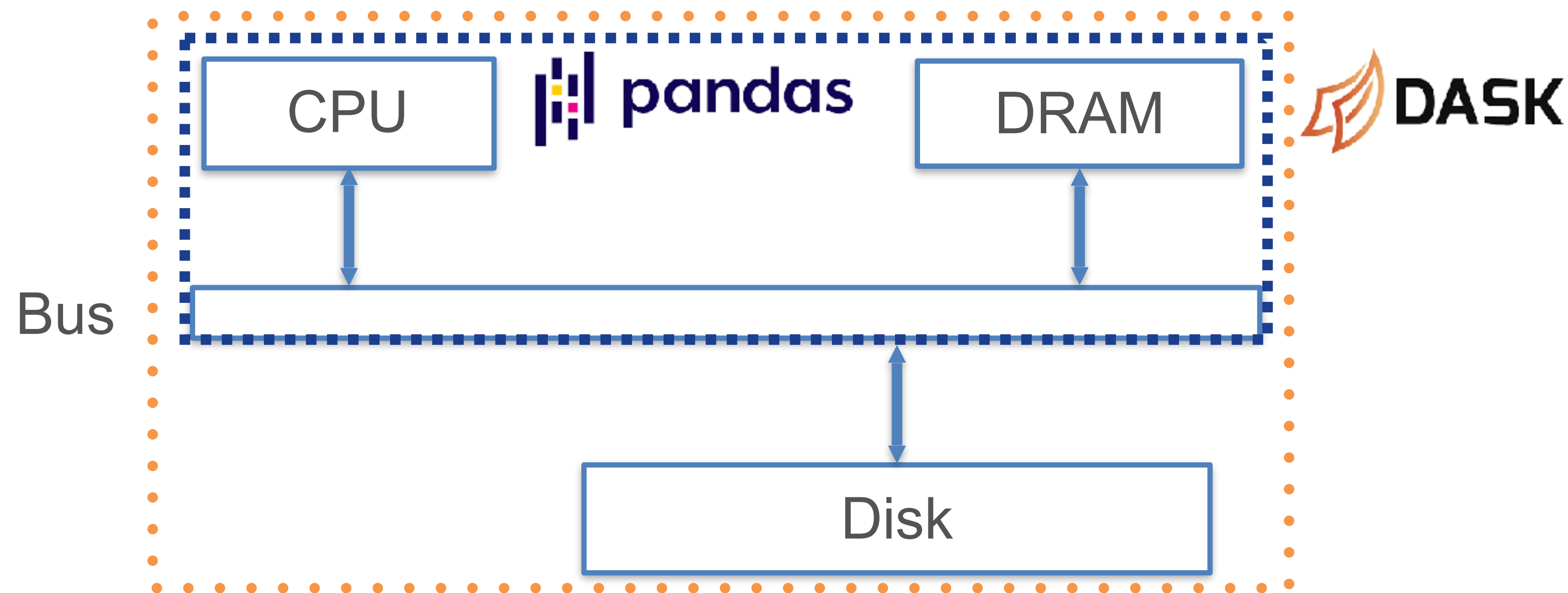
Storage technologies and trends

# Abstract Computer Parts and Data



# Memory Hierarchy in PA0

- Pandas DataFrame needs data to fit entirely in DRAM
- **Dask DataFrame** automatically manages Disk vs DRAM for you
  - Full data sits on Disk, brought to DRAM upon compute()
  - Dask stages out computations using Pandas



- **Tradeoff:** Dask may throw memory configuration issues. :)



# Instruction

Register names



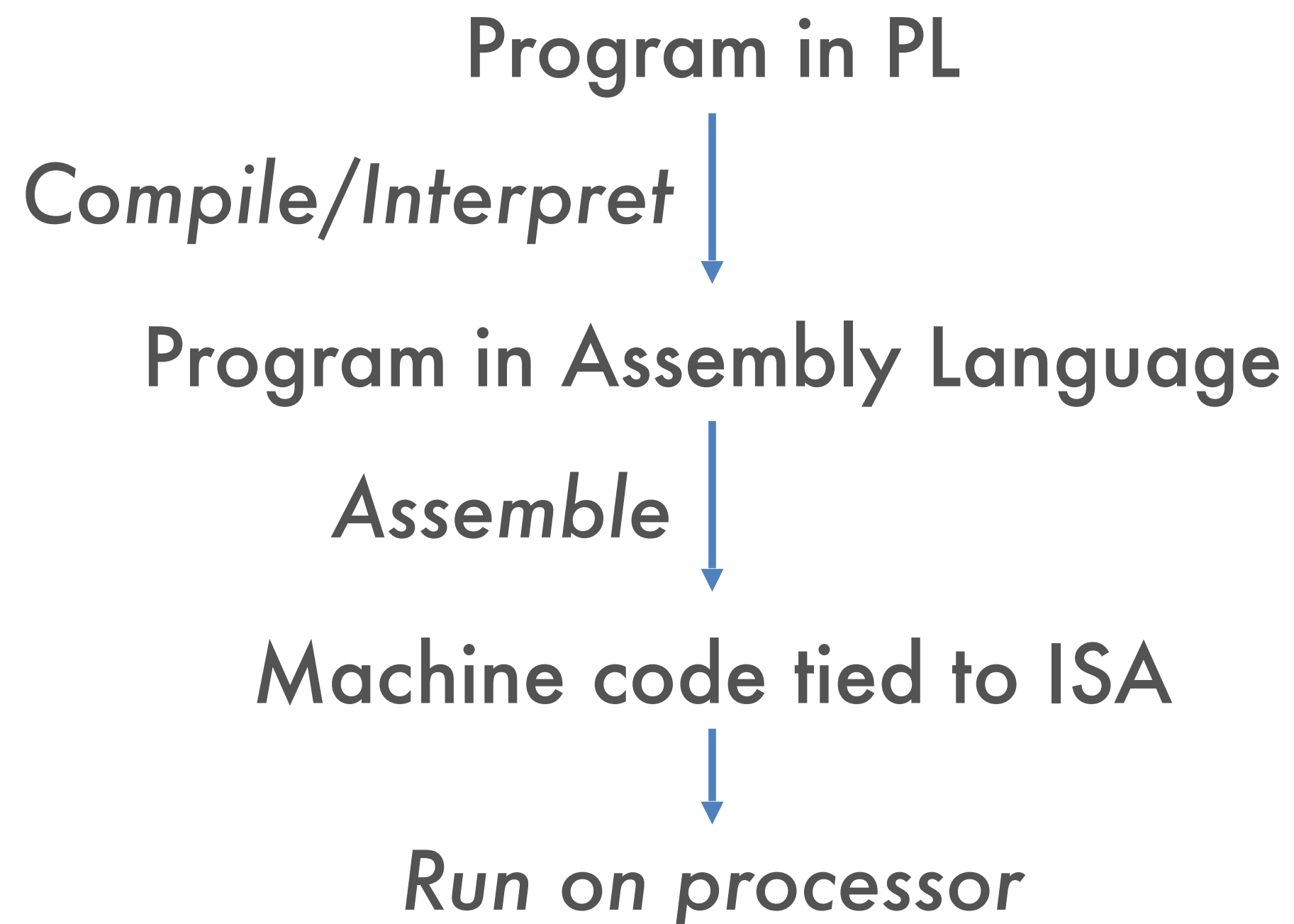
```
addq %rbx, %rax
```

is

```
rax += rbx
```

# Basics of Processors

- **Processor:** Hardware to orchestrate and execute *instructions* to *manipulate data* as specified by a program
  - Examples: CPU, GPU, FPGA, TPU, embedded, etc.
- **ISA (Instruction Set Architecture):**
  - The vocabulary of commands of a processor



```
80483b4: 55          push    %ebp
80483b5: 89 e5       mov     %esp,%ebp
80483b7: 83 e4 f0    and     $0xffffffff0,%esp
80483ba: 83 ec 20    sub     $0x20,%esp
80483bd: c7 44 24 1c 00 00 00 movl    $0x0,0x1c(%esp)
80483c4: 00
80483c5: eb 11       jmp     80483d8 <main+0x24>
80483c7: c7 04 24 b0 84 04 08 movl    $0x80484b0, (%esp)
80483ce: e8 1d ff ff ff call    80482f0 <puts@plt>
80483d3: 83 44 24 1c 01 addl    $0x1,0x1c(%esp)
80483d8: 83 7c 24 1c 09 cmpl    $0x9,0x1c(%esp)
80483dd: 7e e8       jle     80483c7 <main+0x13>
80483df: b8 00 00 00 00 mov     $0x0,%eax
80483e4: c9         leave
80483e5: c3         ret
80483e6: 90         nop
80483e7: 90         nop
80483e8: 90         nop
80483e9: 90         nop
80483ea: 90         nop
```



# Basics of Processors

**Q:** *How does a processor execute machine code?*

- Most common approach: **load-store architecture**
- **Registers:** Tiny local memory (“scratch space”) on proc. into which instructions and data are copied
- ISA specifies bit length/format of machine code commands
- ISA has several commands to manipulate register contents

# Writing & Reading Memory

## Write

Transfer data from memory to CPU

**movq %rax, %rsp**

“Store” operation

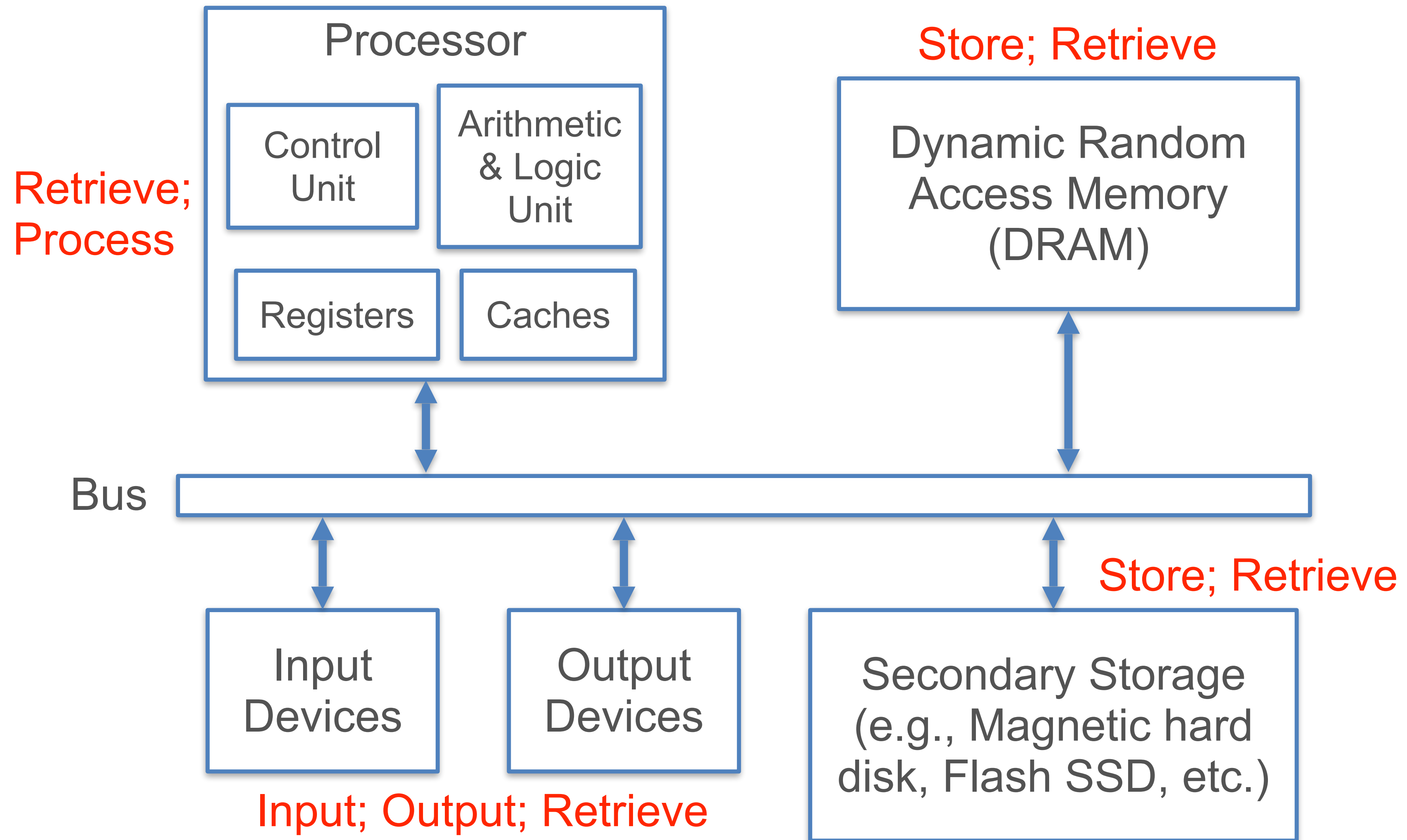
## Read

Transfer data from CPU to memory

**movq %rsp, %rax**

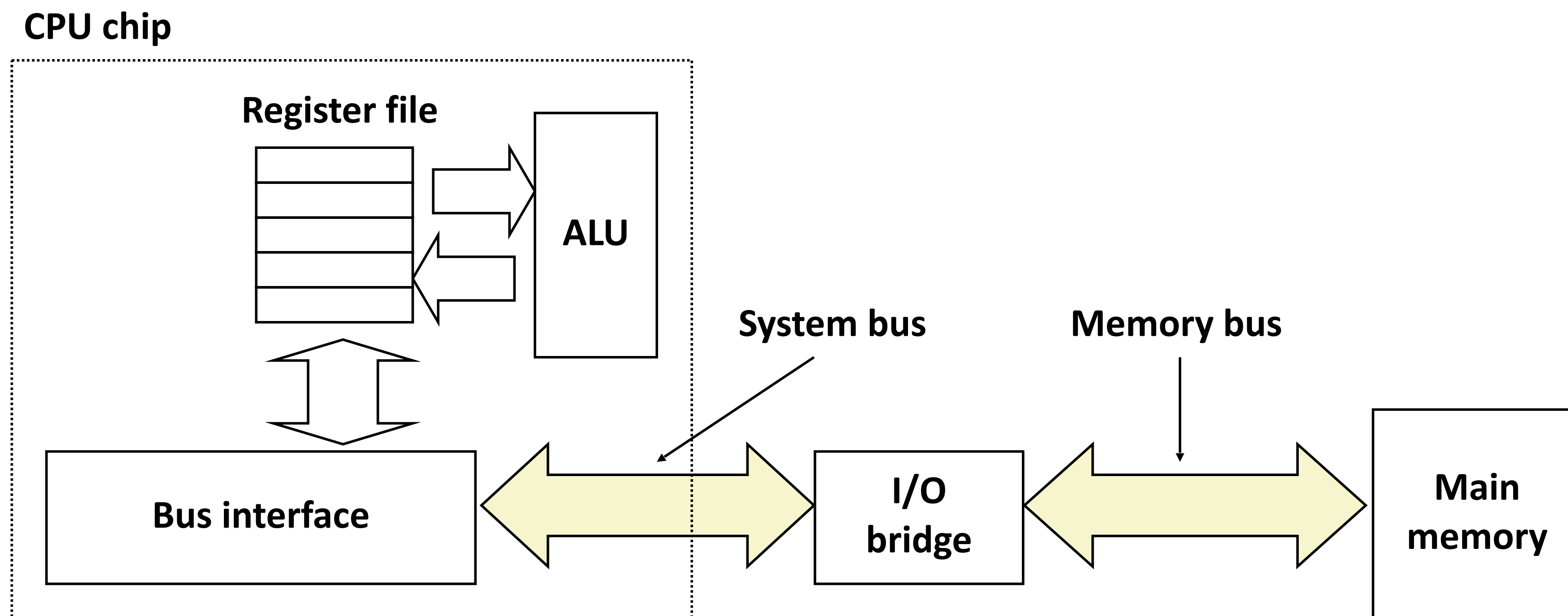
“Load” operation

# Abstract Computer Parts and Data

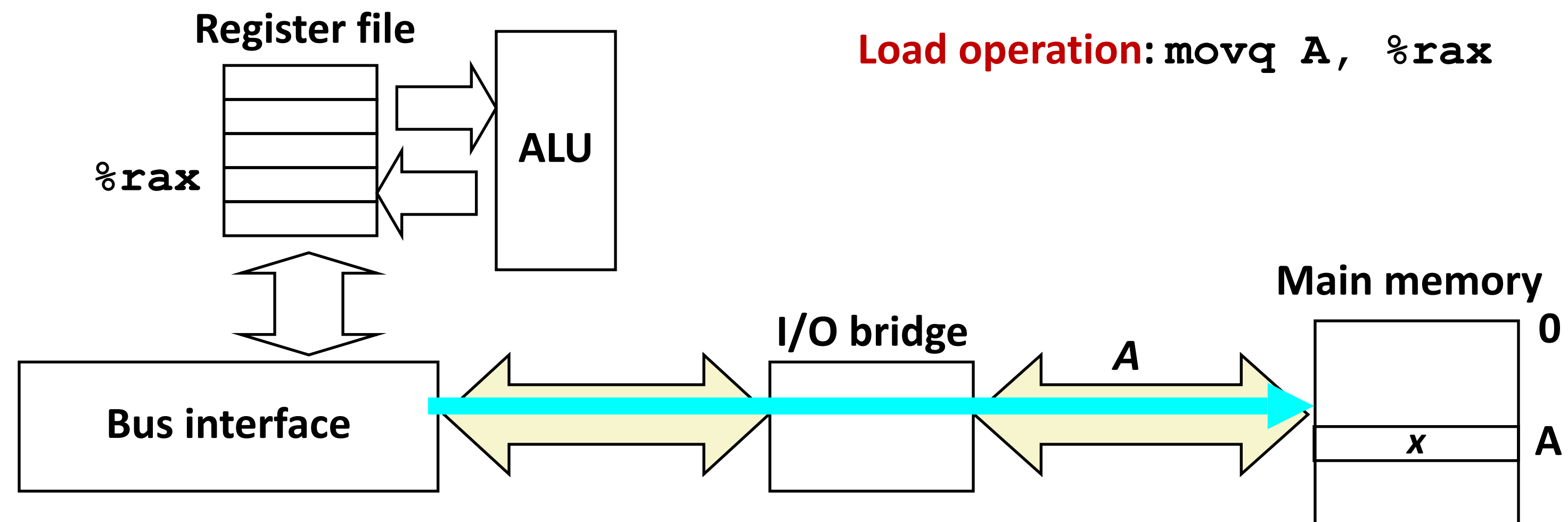


# Bus Structure Connecting CPU and Memory

A **bus** is a collection of parallel wires that carry address, data, and control signals. Buses are typically shared by multiple devices.

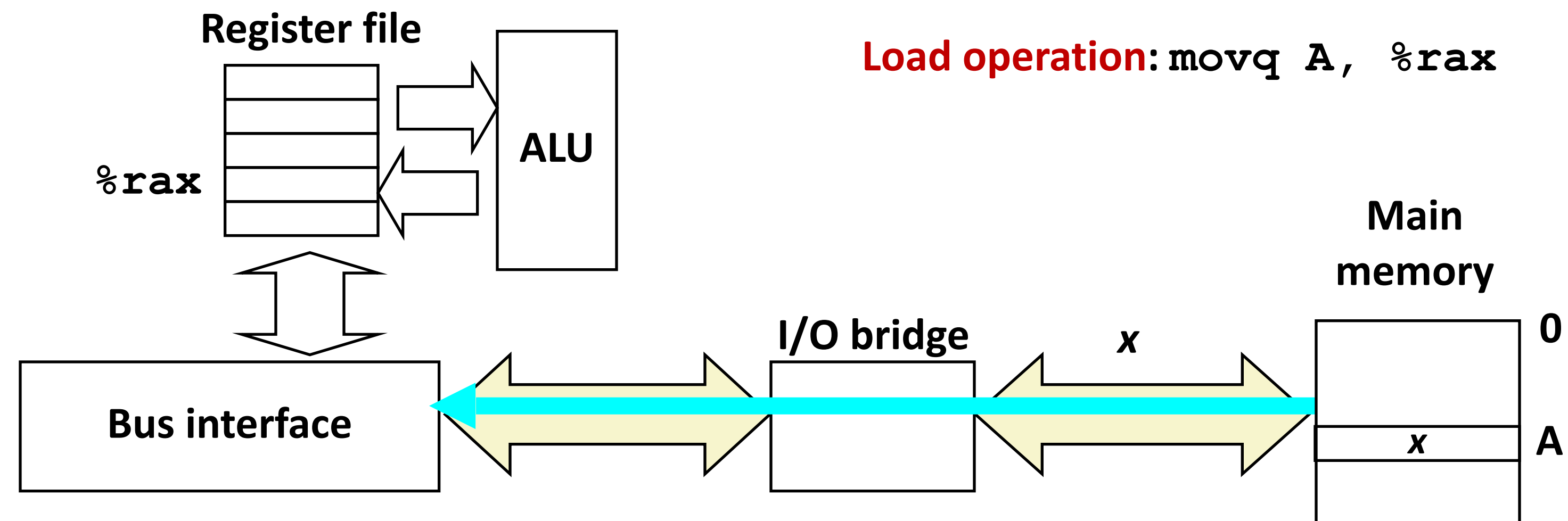


# Memory Read Transaction (1)



CPU places address *A* on the memory bus.

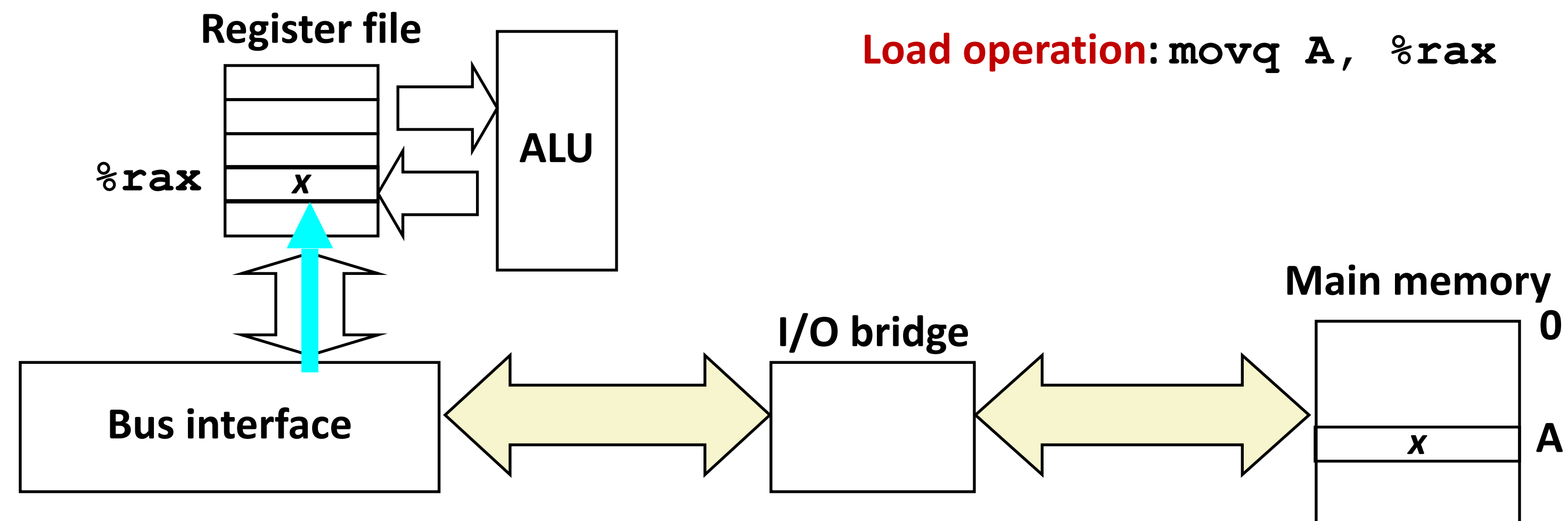
# Memory Read Transaction (2)



Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

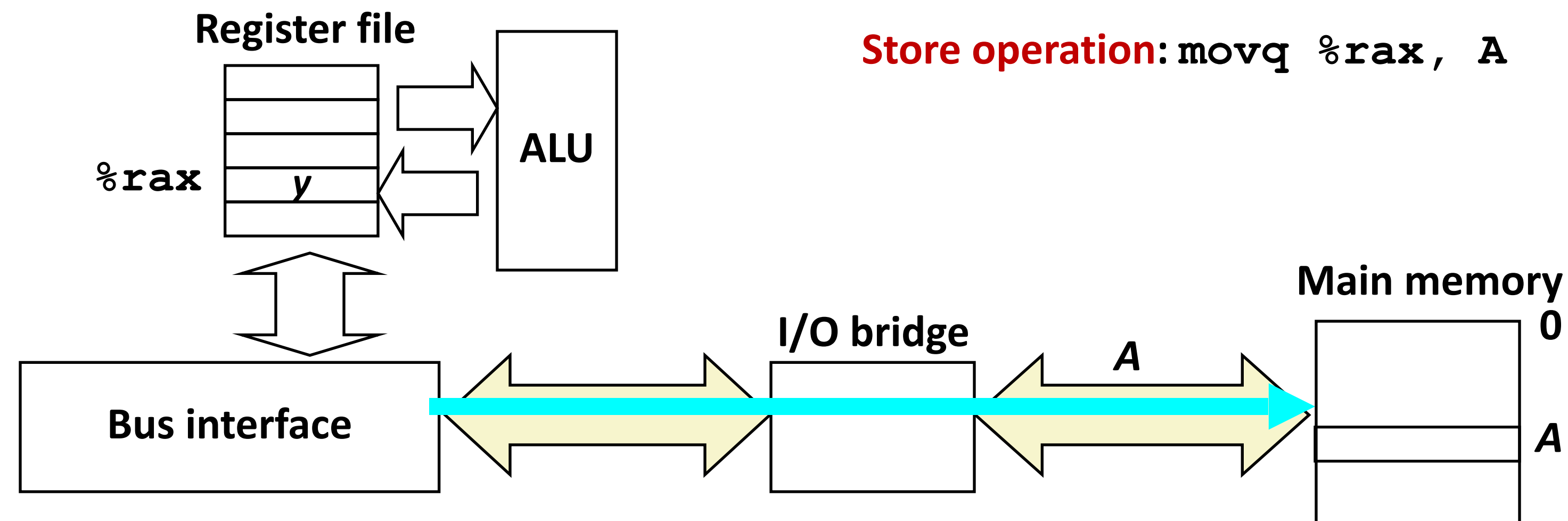


# Memory Read Transaction (3)



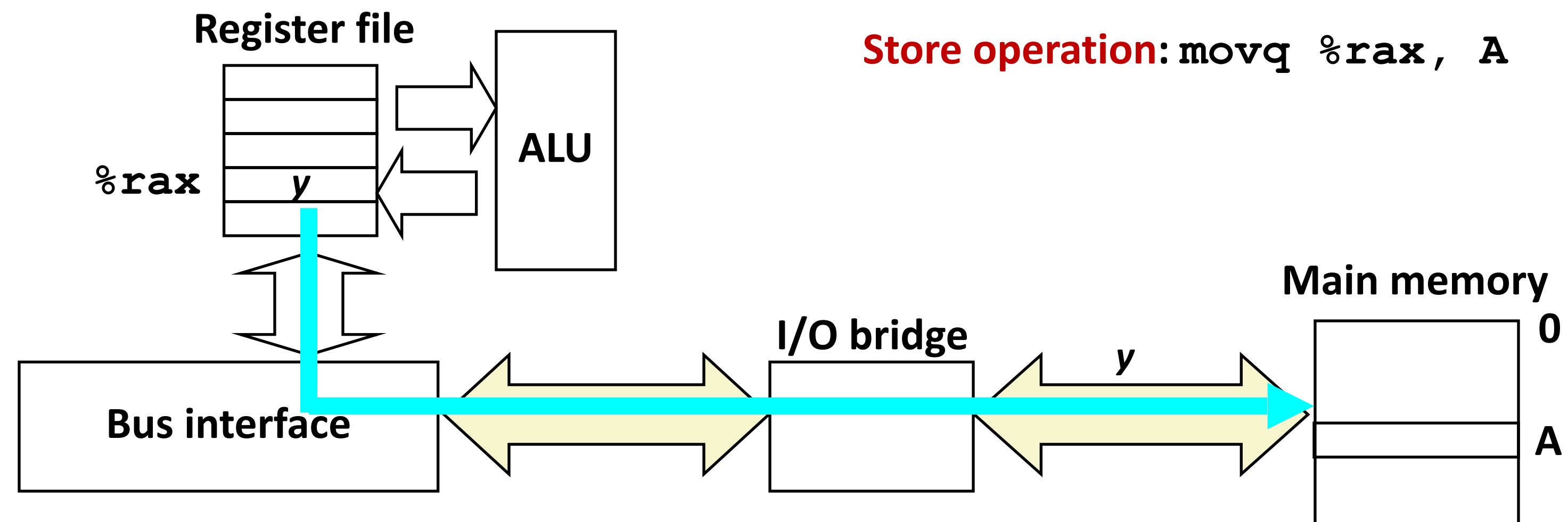
CPU reads word `x` from the bus and copies it into register `%rax`.

# Memory Write Transaction (1)



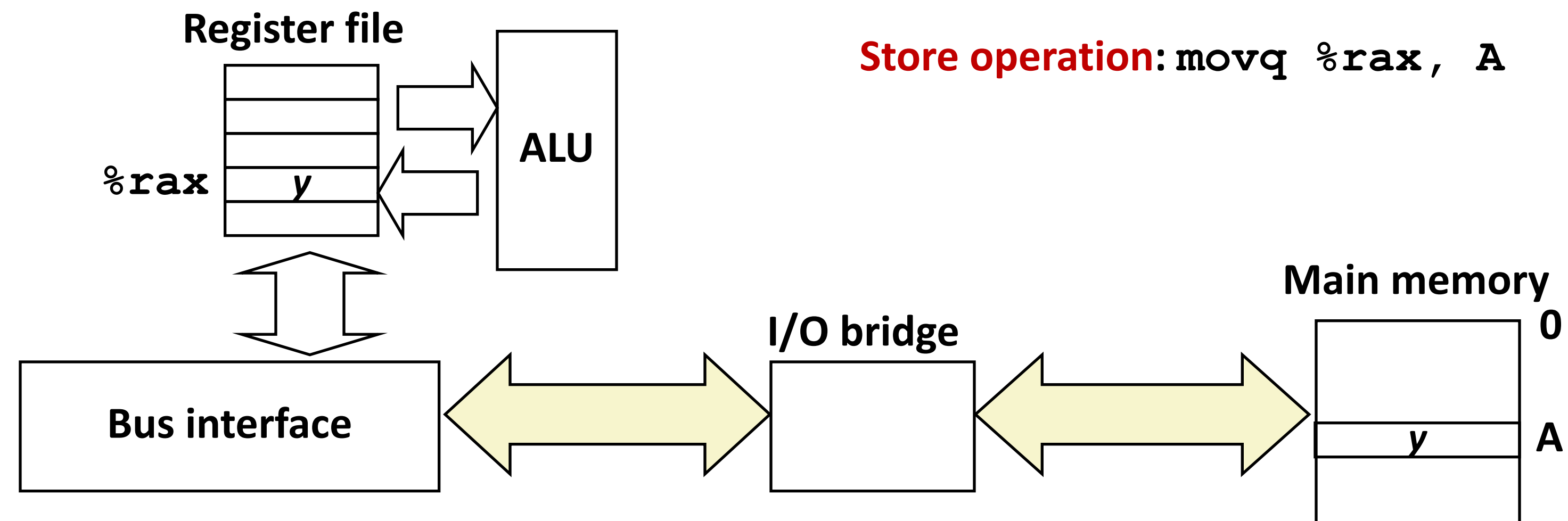
CPU places address *A* on bus. Main memory reads it and waits for the corresponding data word to arrive.

# Memory Write Transaction (2)



CPU places data word `y` on the bus.

# Memory Write Transaction (3)



Main memory reads data word `y` from the bus and stores it at address `A`.

# Basics of Processors

**Q:** *How does a processor execute machine code?*

- Types of ISA commands to manipulate register contents:
  - **Memory access: load** (copy bytes from a DRAM address to register); **store** (reverse); put constant
  - **Arithmetic & logic** on data items in registers: add/multiply/etc.; bitwise ops; compare, etc.; handled by ALU
  - **Control flow** (branch, call, etc.); handled by CU
- **Caches:** Small local memory to buffer instructions/data



80483b5:	89 e5	mov	%esp,%ebp
80483b7:	83 e4 f0	and	\$0xfffffffff0,%esp
80483ba:	83 ec 20	sub	\$0x20,%esp
80483bd:	c7 44 24 1c 00 00 00	movl	\$0x0,0x1c(%esp)
80483c4:	00		
80483c5:	eb 11	jmp	80483d8 <main+0x24>
80483c7:	c7 04 24 b0 84 04 08	movl	\$0x80484b0, (%esp)
80483ce:	e8 1d ff ff ff	call	80482f0 <puts@plt>
80483d3:	83 44 24 1c 01	addl	\$0x1,0x1c(%esp)
80483d8:	83 7c 24 1c 09	cmpl	\$0x9,0x1c(%esp)
80483dd:	7e e8	jle	80483c7 <main+0x13>
80483df:	b8 00 00 00 00	mov	\$0x0,%eax
80483e4:	c9	leave	
80483e5:	c3	ret	
80483e6:	90	nop	
80483e7:	90	nop	
80483e8:	90	nop	
80483e9:	90	nop	



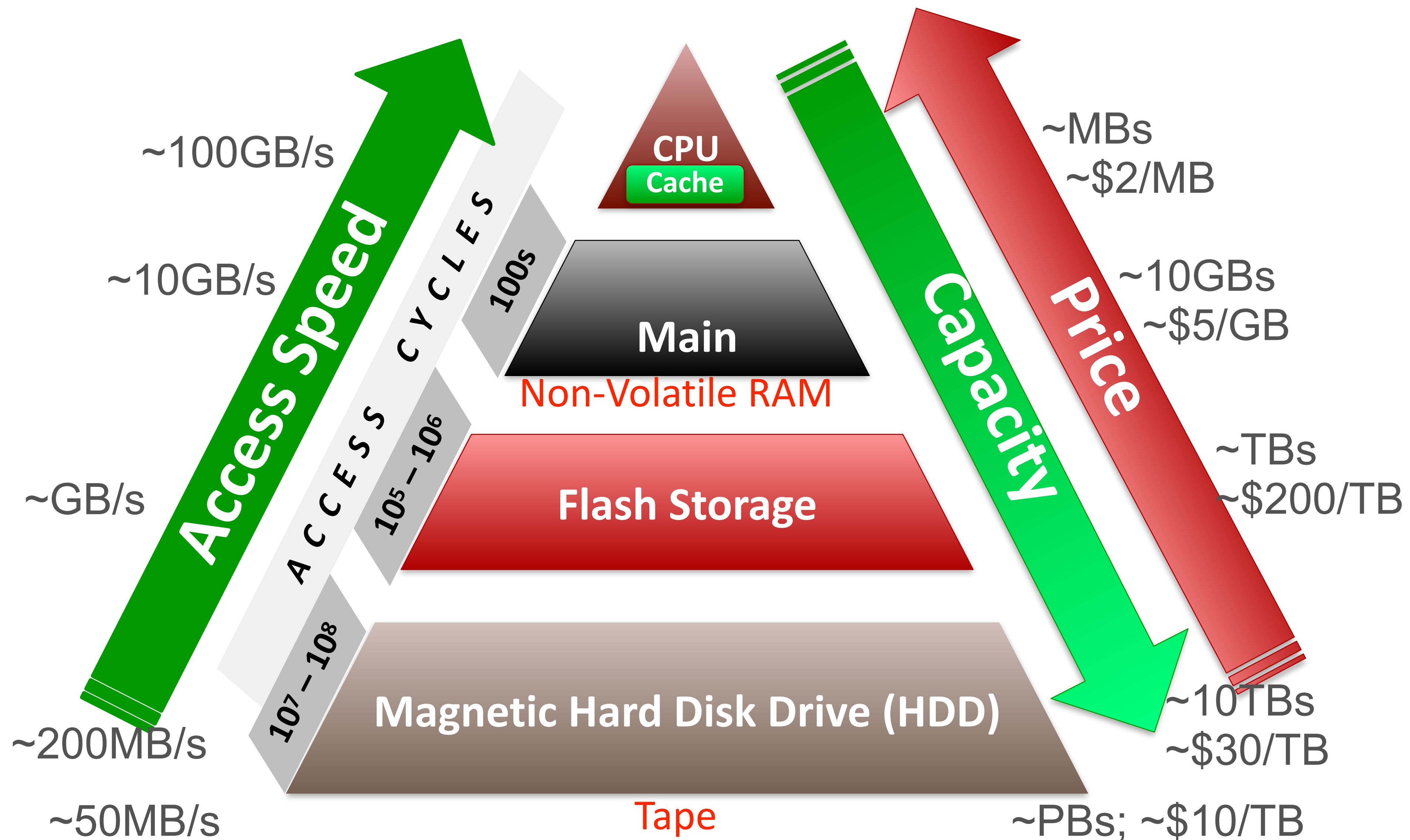
# Processor Performance

*Q: How fast can a processor process a program?*

- ❖ Modern CPUs can run millions of instructions per second!
- ❖ ISA tells **#clock cycles** per instruction
- ❖ CPU's **clock rate** helps map that to runtime (ns)
- ❖ Alas, most programs do not keep CPU always busy:
  - ❖ Memory access commands **stall** the processor; ALU and CU are *idle* during DRAM-register transfer
  - ❖ Worse, data may not be in DRAM—wait for (disk) I/O!
  - ❖ So, actual *runtime* of program may be OOM higher than what clock rate calculation suggests

**Key Principle:** Optimizing access to DRAM and use of processor caches is critical for processor performance!

# Memory/Storage Hierarchy



# Example

**Q:** *What does this program do when run with 'python'?  
(Assume tmp.csv is in current working directory)*

tmp.py

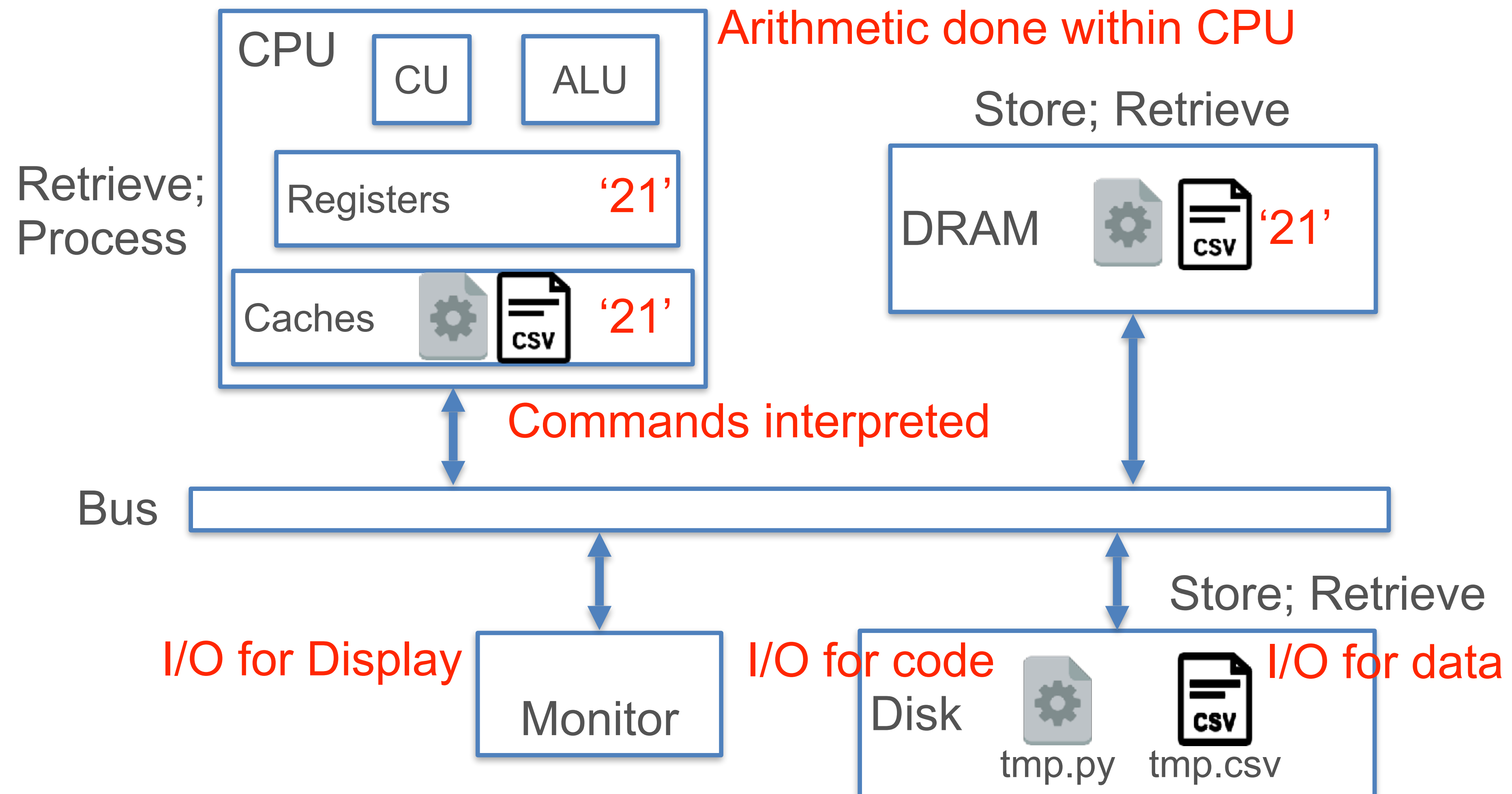
```
import pandas as p
m = p.read_csv('tmp.csv',header=None)
s = m.sum().sum()
print(s)
```

tmp.csv

```
1,2,3
4,5,6
```

# Example

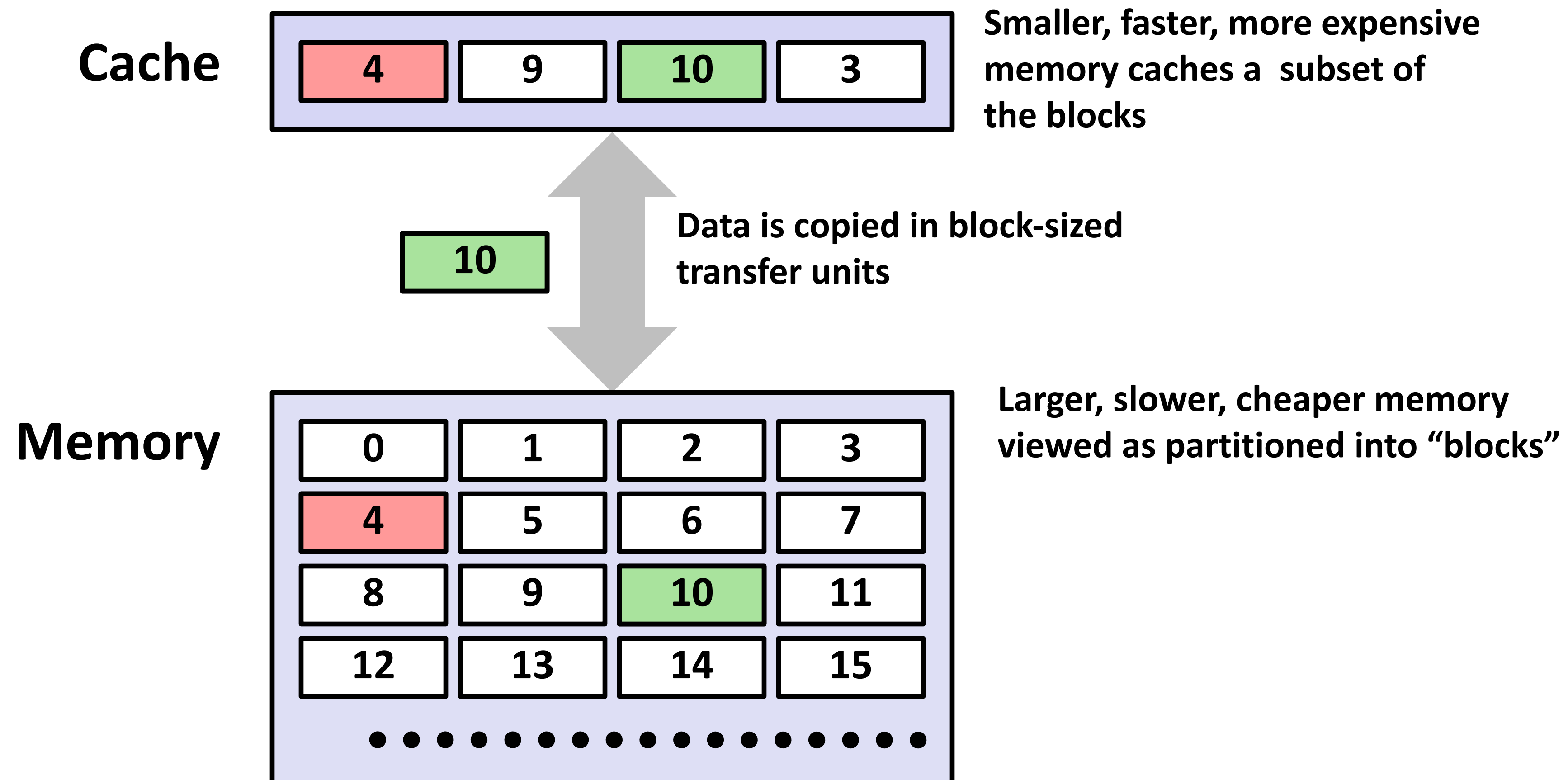
Rough sequence of events when program is executed



# Concepts of Memory Management

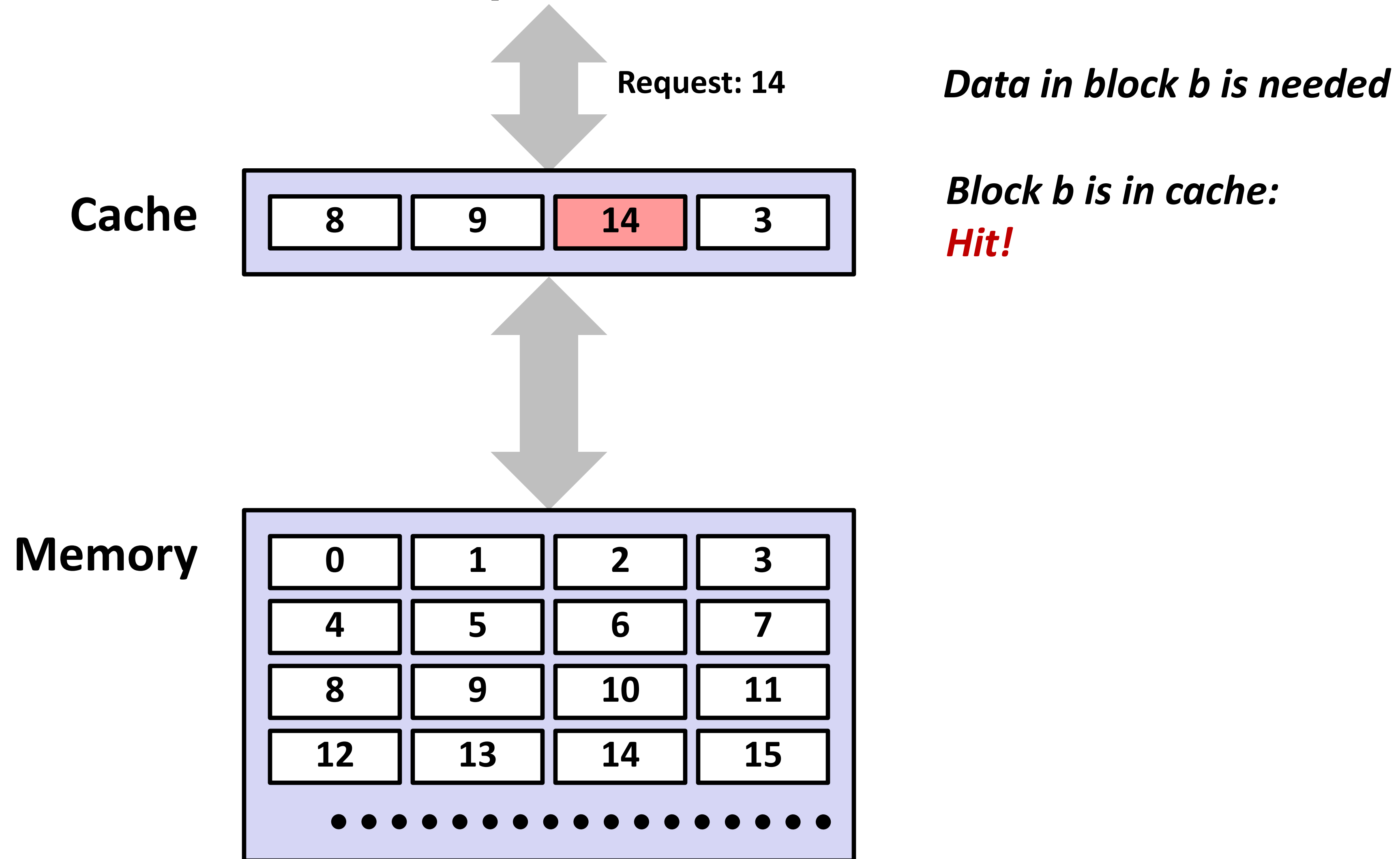
- ❖ **Caching:** Buffering a copy of bytes (instructions and/or data) from a lower level at a higher level to exploit locality
- ❖ **Prefetching:** Preemptively retrieving bytes (typically data) from addresses not explicitly asked yet by program
- ❖ **Spill/Miss/Fault:** Data needed for program is not yet available at a higher level; need to get it from lower level
  - ❖ **Register Spill** (register to cache); **Cache Miss** (cache to main memory); **“Page” Fault** (main memory to disk)
- ❖ **Hit:** Data needed is already available at higher level
- ❖ **Cache Replacement Policy:** When new data needs to be loaded to higher level, which old data to evict to make room?  
Many policies exist with different properties

# General Cache Concepts

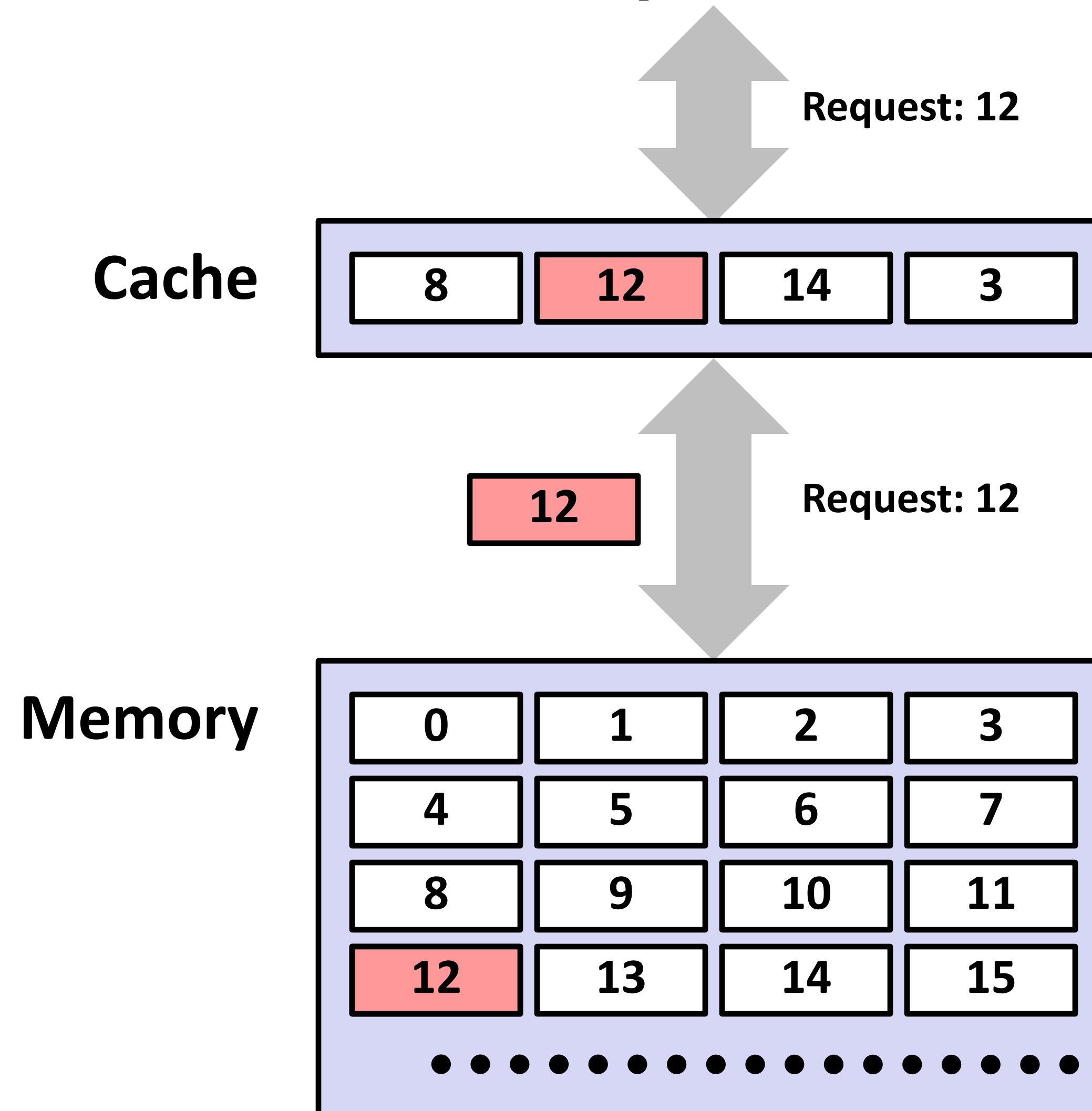




# General Cache Concepts: Hit



# General Cache Concepts: Miss



*Data in block b is needed*

*Block b is not in cache:*  
**Miss!**

*Block b is fetched from*  
*memory*

*Block b is stored in cache*

- **Placement policy:**  
determines where b goes
- **Replacement policy:**  
determines which block  
gets evicted (victim)

# x86 v.s. x64

- x86 is the name of an ISA (80X86). 32-bit CPU.
- x64 is the name of an ISA(x86-64). 64-bit CPU
- 32-bit CPU can handle 32 bits information in each instruction.
- When we represent the memory address in bits, 32-bit CPU can support at most  $2^{32}$  bytes = 4 GB.
- You can install a 32-bit OS on a 64-bit CPU. But not a 64-bit OS on a 32-bit CPU.

# HIGHLIGHT

**DANGER**  
BE CAREFUL  
WHEN BELT  
IS IN MOTION.



# | Will & Grace



# Today

The memory abstraction

**Locality of reference**

The memory hierarchy

Storage technologies and trends

# copyij v.s copyji: copy a 2048 X 2048 integer array

```
void copyij(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

**4.3 milliseconds**

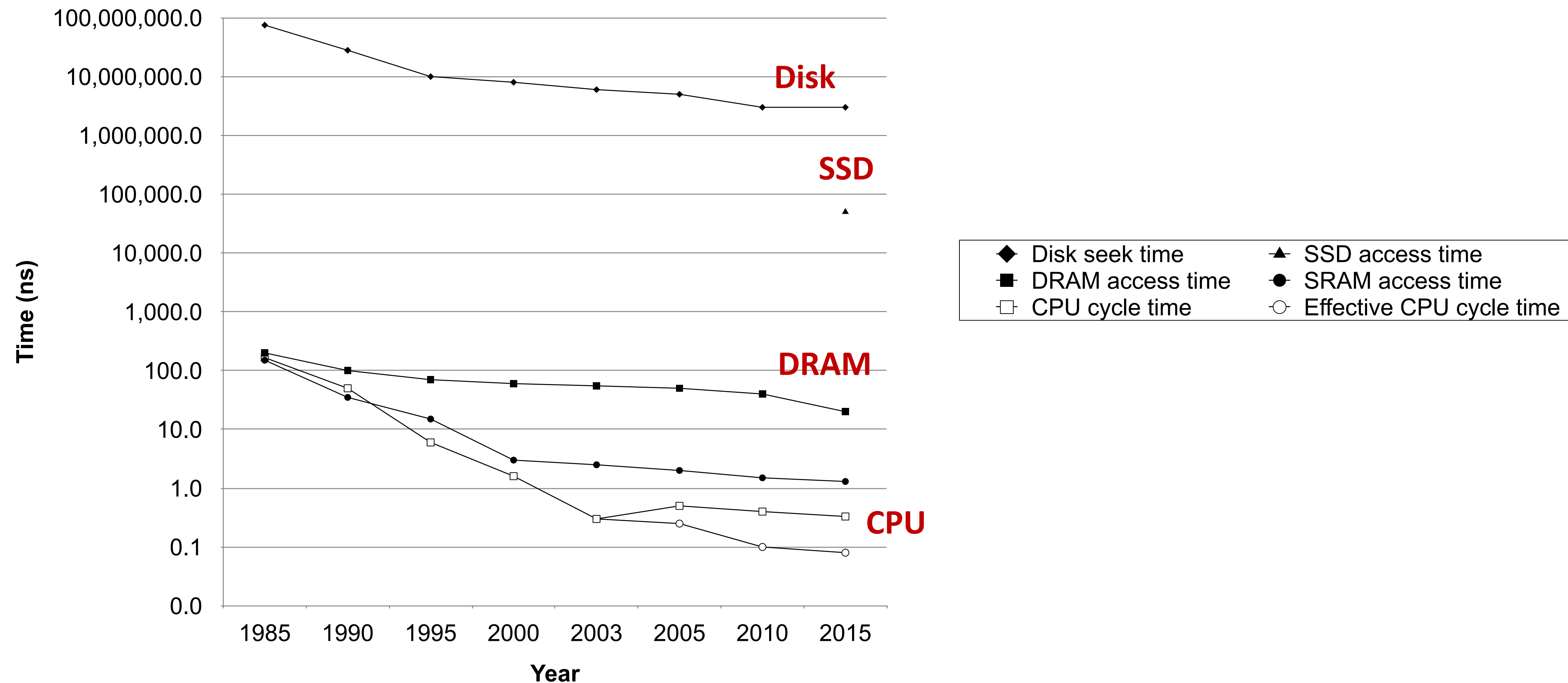
```
void copyji(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

**81.8 milliseconds**



# The CPU-Memory Gap

**The gap *widens* between DRAM, disk, and CPU speeds.**



# Why?

```
void copyij(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```
void copyji(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

1. Not hardware-efficient

2. Data layout in DRAM is in often row-major order.

src[0;0], src[0;1], src[0;2], ..., src[1;0], ...

3. Prefetching + caching means full row based on innermost loop is brought to processor cache.

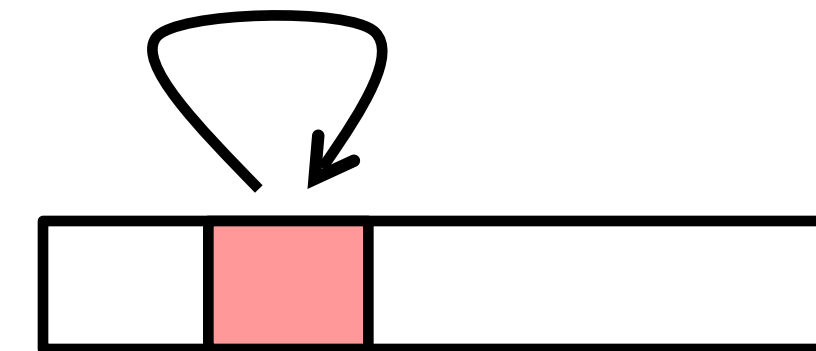
4. Hit v.s. miss.

# Locality

**Principle of Locality:** Many Programs tend to use data and instructions with addresses near or equal to those they have used recently.

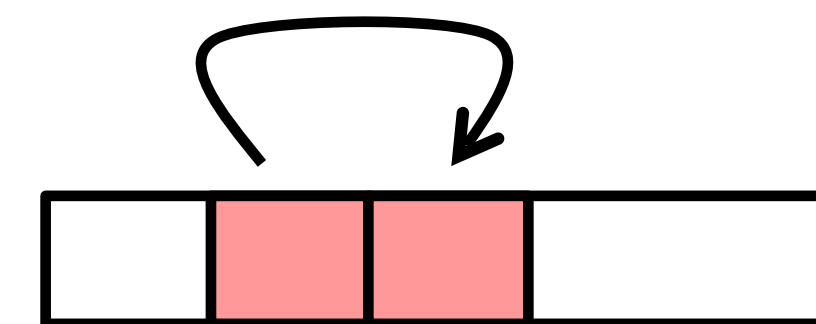
## Temporal locality:

Recently referenced items are likely to be referenced again in the near future



## Spatial locality:

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



# Spatial locality

```
void copyij(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}

void copyji(long int src[2048][2048], long int dst[2048][2048])
{
    long int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

DRAM:

src[0;0], src[0;1], src[0;2], ..., src[1;0],  
...

Caches:

src[0:0], src[0:1], src[0:3],...

Two types of spatial locality:

1. OS prefetching
2. Hardware

# Temporal locality

```
int sum_array_cols(int a[M])
{
    int i = 0;

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

DRAM:

a[0], a[1], a[2], .....

Caches:

sum, a[0], a[1], a[2] ...

Two types of temporal locality:

1. Cache
2. Memory
3. Disk

# Locality to the Rescue!

The key to bridging this CPU-Memory gap is an important property of computer programs known as **locality**.

# Locality Example

```
num_list = [1, 2, 3, 4, 5, 7]
sum = 0;
for (x in num_list)
    sum += x;
return sum;
```

**Spatial or Temporal  
Locality?**

## Data references

Reference array elements in succession (stride-1 reference pattern).

Reference variable **sum** each iteration.

**spatial**

**temporal**

## Instruction references

Reference instructions in sequence.

Cycle through loop repeatedly.

**spatial**

**temporal**

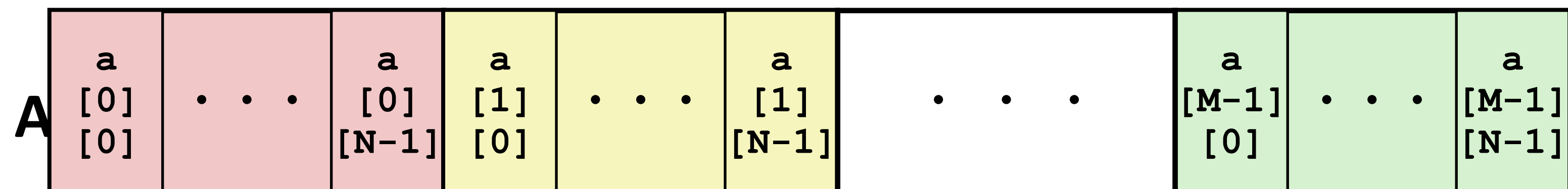
# Qualitative Estimates of Locality

**Hint: array layout  
is row-major order**

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```



**Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

**Question:** Does this function have good locality with respect to array *a*?



# Locality Example

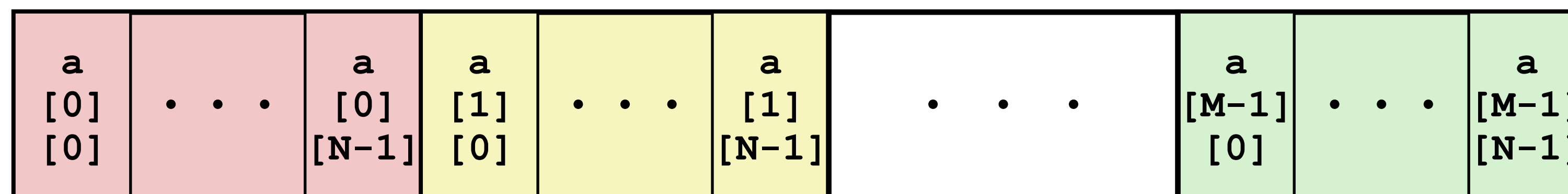
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

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

**Answer: no, unless...**

**M is very small**

**Question:** Does this function have good locality with respect to array a?



# Locality Example

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```

**Question:** Can you permute the loops so that the function scans the 3-d array *a* with a stride-1 reference pattern (and thus has good spatial locality)?

**Answer: make j the inner loop**

# Today

The memory abstraction

Locality of reference

**The memory hierarchy**

Storage technologies and trends

# Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These properties complement each other well for many types of programs.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

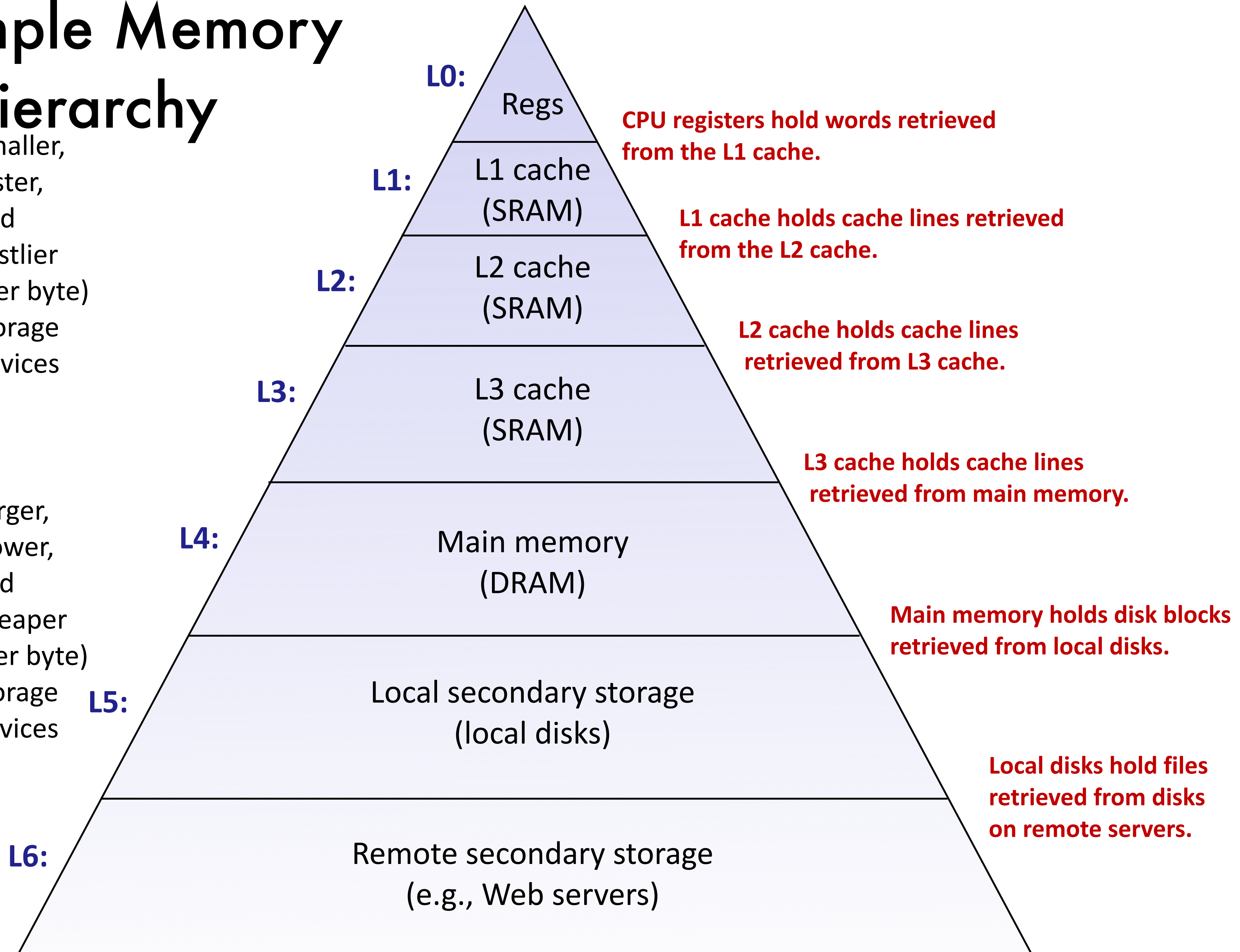
# Example Memory Hierarchy



Smaller,  
faster,  
and  
costlier  
(per byte)  
storage  
devices



Larger,  
slower,  
and  
cheaper  
(per byte)  
storage  
devices



# Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- 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  $k+1$ .
- Why do memory hierarchies work?
  - Because of locality: 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.
- **Big Idea (Ideal):** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

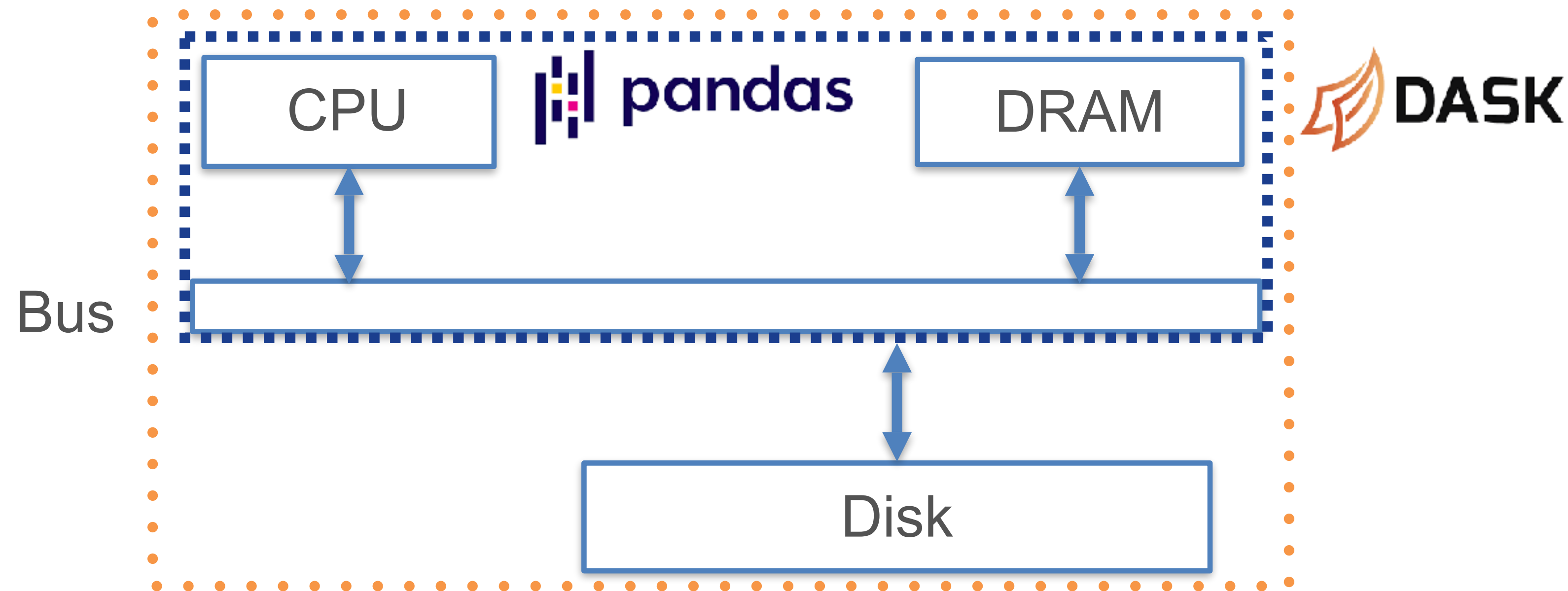


# Memory/Storage Hierarchy

- ❖ Typical desktop computer today (\$700):
  - ❖ 1 TB magnetic hard disk (SATA HDD); 32 GB DRAM
  - ❖ 3.4 GHz CPU; 4 cores; 8MB cache
- ❖ High-end enterprise rack server for RDBMSs (\$8,000):
  - ❖ 12 TB Persistent memory; 6 TB DRAM
  - ❖ 3.8 GHz CPU; 28-core per proc.; 38MB cache
- ❖ Renting on Amazon Web Services (AWS):
  - ❖ EC2 m5.large: 2-core, 8GiB: \$0.096 / hour
  - ❖ EC2 m5.24xlarge: 96-core, 384 GiB, \$4.608 per hour
  - ❖ EBS general SSD: \$0.10 per GB-month
  - ❖ S3 store / read: \$0.023 / 0.013-0.023 per GB-month

# Memory Hierarchy in PA0

- ❖ Pandas DataFrame needs data to fit entirely in DRAM
- ❖ **Dask DataFrame** automatically manages Disk vs DRAM for you
  - ❖ Full data sits on Disk, brought to DRAM upon compute()
  - ❖ Dask stages out computations using Pandas



- ❖ **Tradeoff:** Dask may throw memory configuration issues. :)

# Processor-Optimized Math Libraries

*Q: Would you like to write ML code in a cache-aware manner? :)*

- ❖ Matrices/tensors are ubiquitous in statistics/ML/DL programs
- ❖ Processor-optimized libraries for matrix/tensor arithmetic (*linear algebra*) studied for decades
- ❖ They reduce memory stalls and increase parallelism (more on parallelism later) automatically:
  - ❖ **Multi-core CPUs:** BLAS/LAPACK (C), Eigen (C++), la4j (Java), NumPy/SciPy (Python; can wrap BLAS)
  - ❖ **GPUs:** cuBLAS, cuSPARSE, cuDNN, cuDF, cuGraph

If interested, some benchmark empirical comparisons:

<https://medium.com/datathings/benchmarking-blas-libraries-b57fb1c6dc7>

# Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

# Today

The memory abstraction

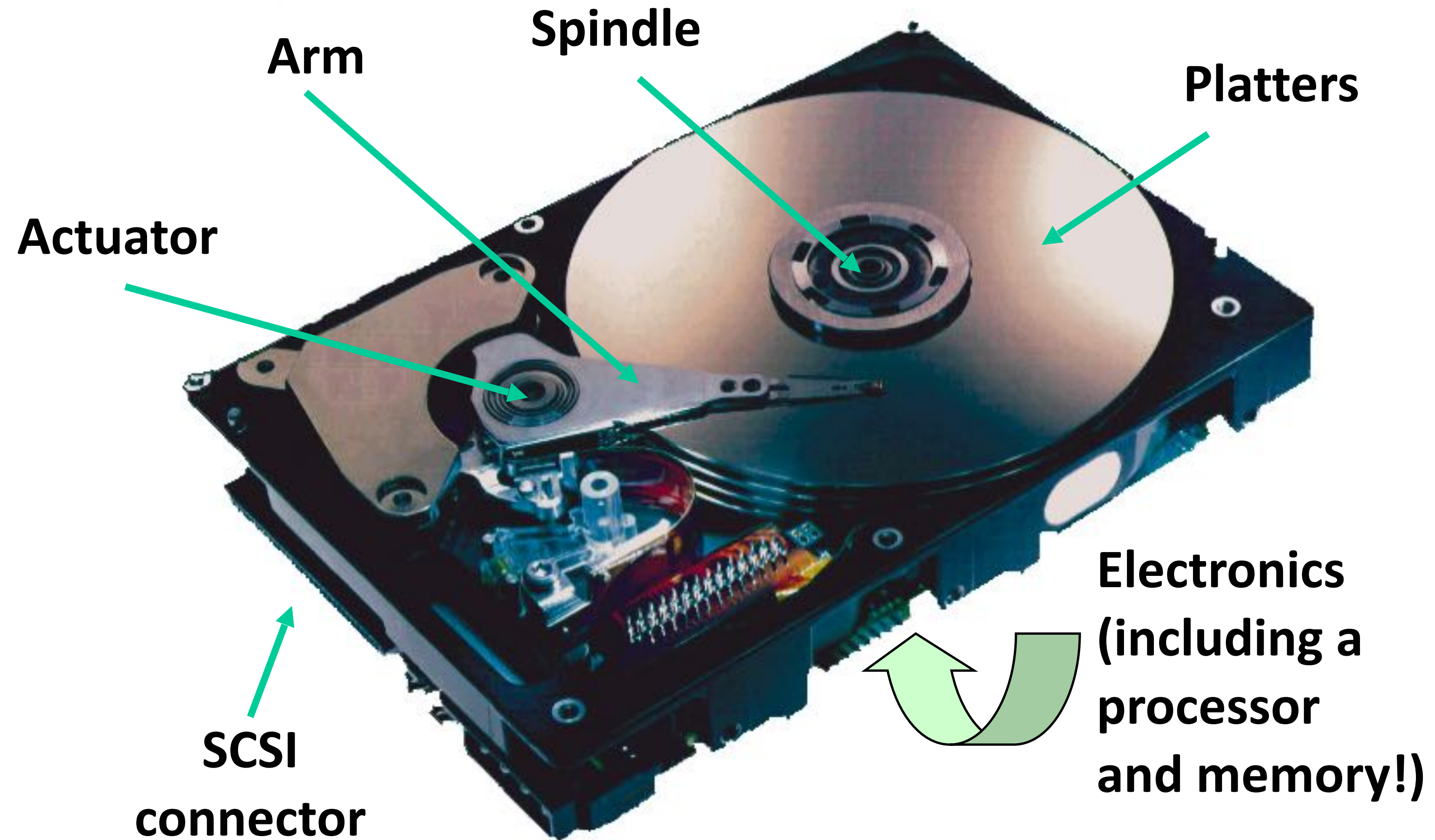
Locality of reference

The memory hierarchy

**Storage technologies and trends**



# What's Inside A Disk Drive?

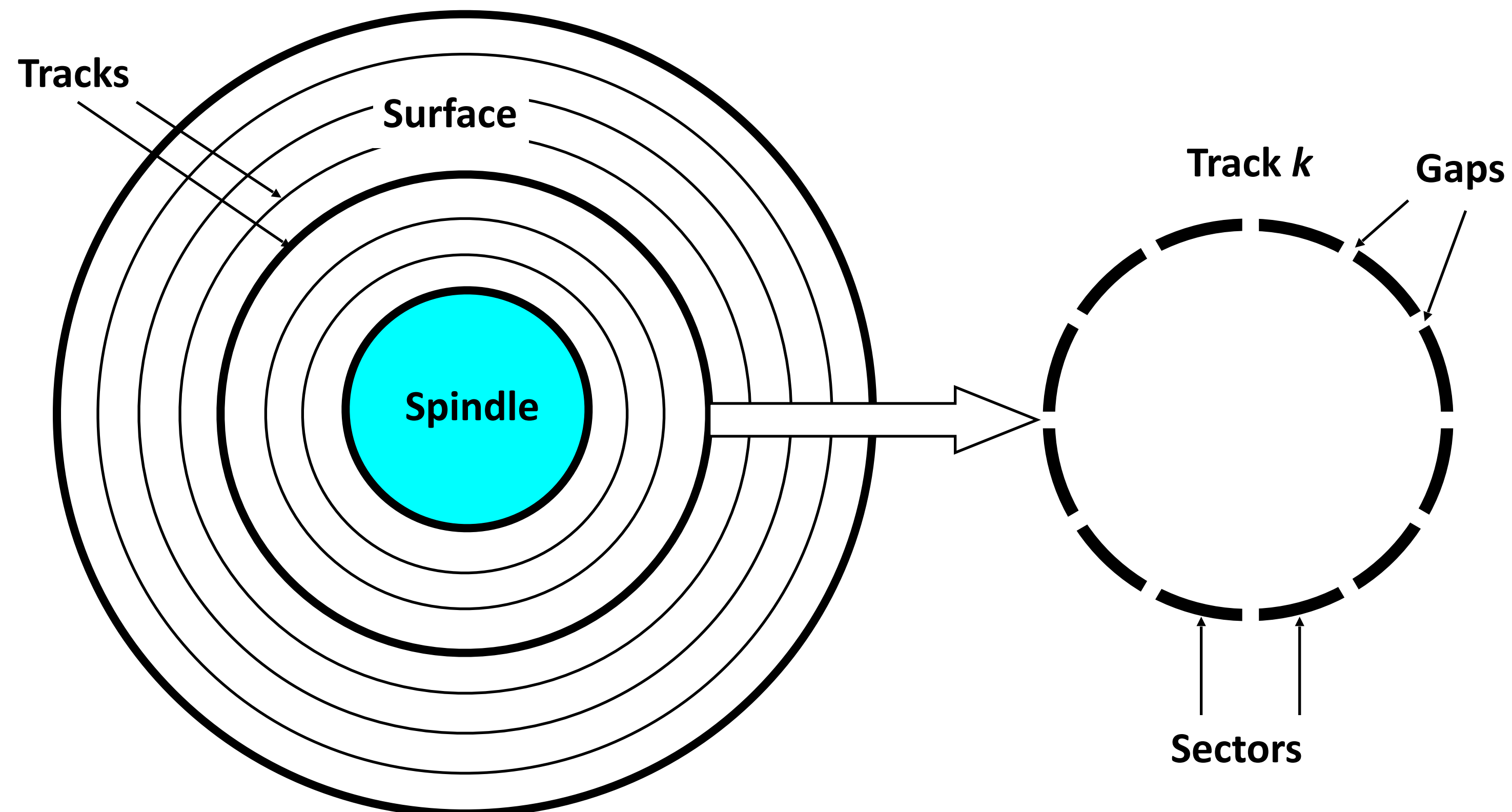


*Image courtesy of Seagate Technology*



# Disk Geometry

- Disks consist of **platters**, each with two **surfaces**.
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by **gaps**.



# Disk Capacity

**Capacity**: maximum number of bits that can be stored.

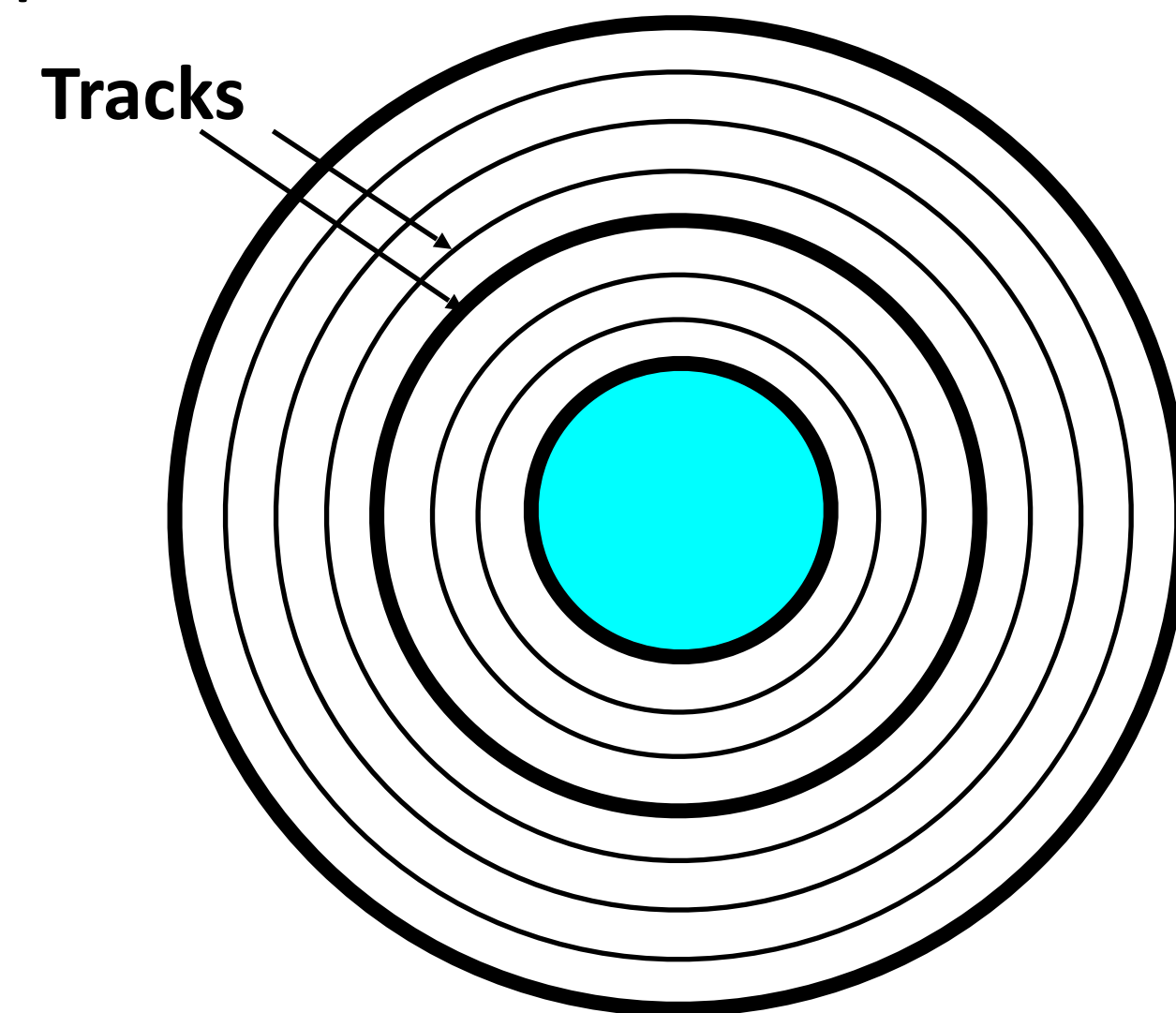
Vendors express capacity in units of gigabytes (GB) or terabytes (TB), where 1 GB =  $10^9$  Bytes and 1 TB =  $10^{12}$  Bytes

Capacity is determined by these technology factors:

**Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.

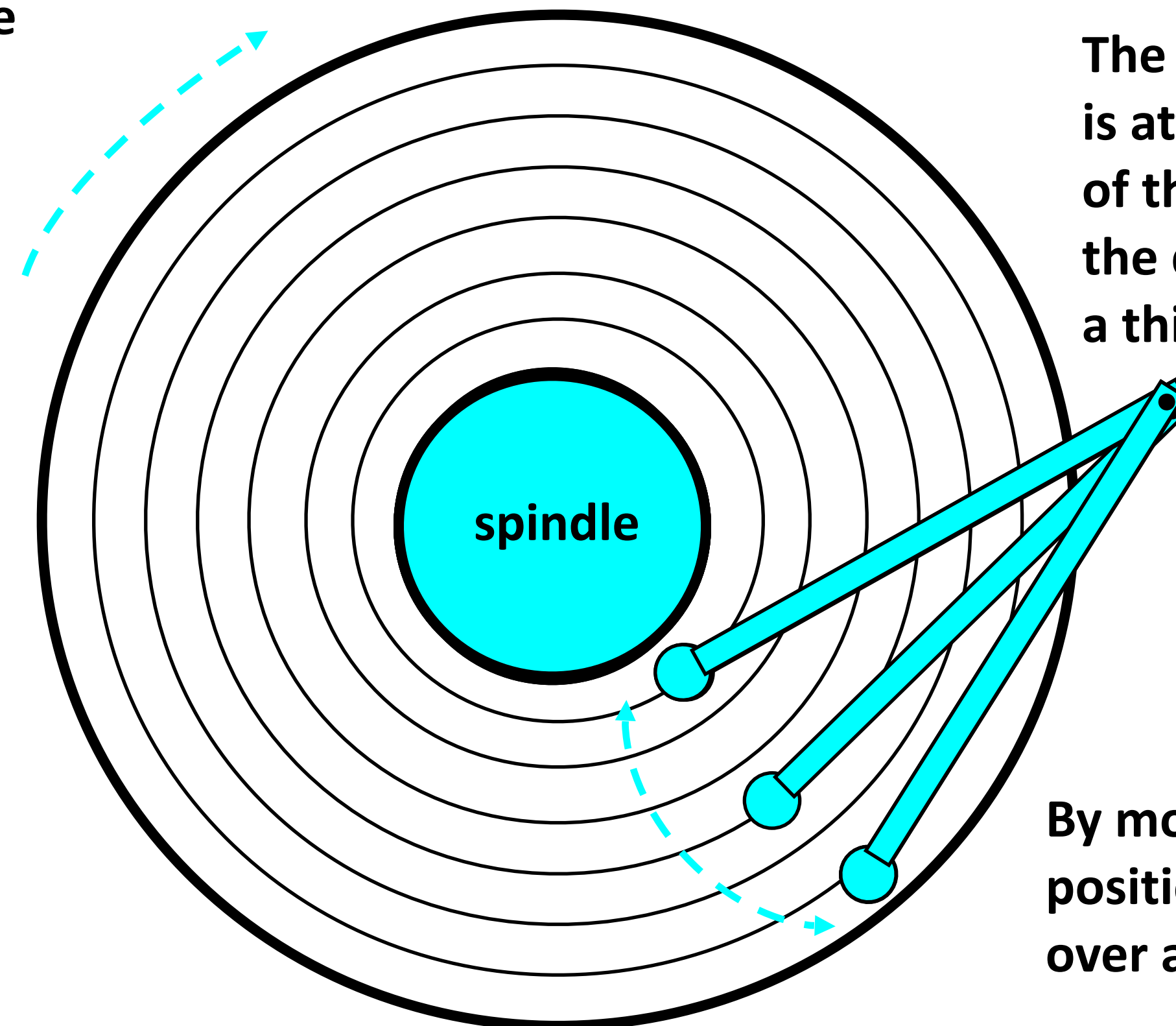
**Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.

**Areal density** (bits/in<sup>2</sup>): product of recording and track density.



# Disk Operation (Single-Platter View)

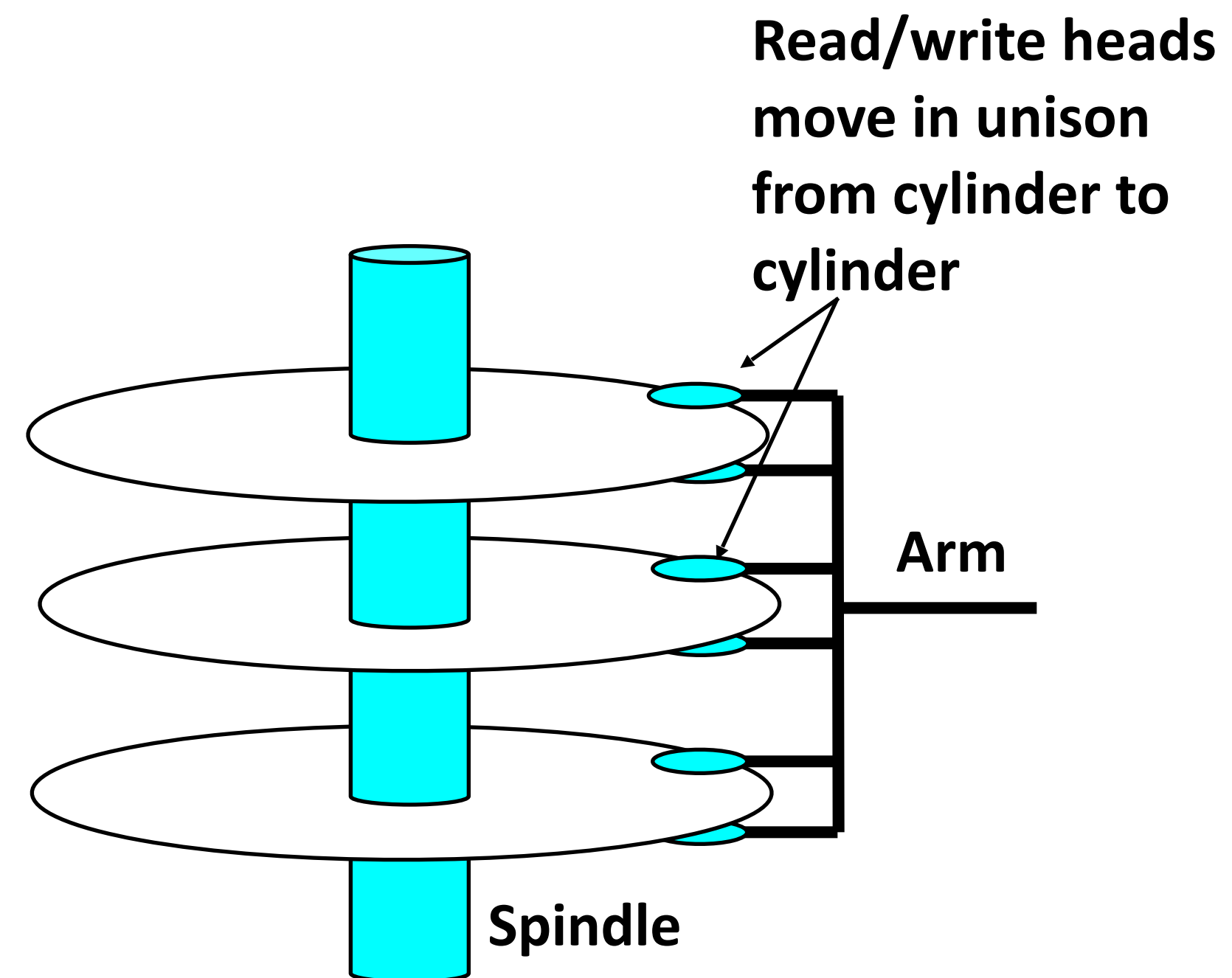
The disk surface spins at a fixed rotational rate



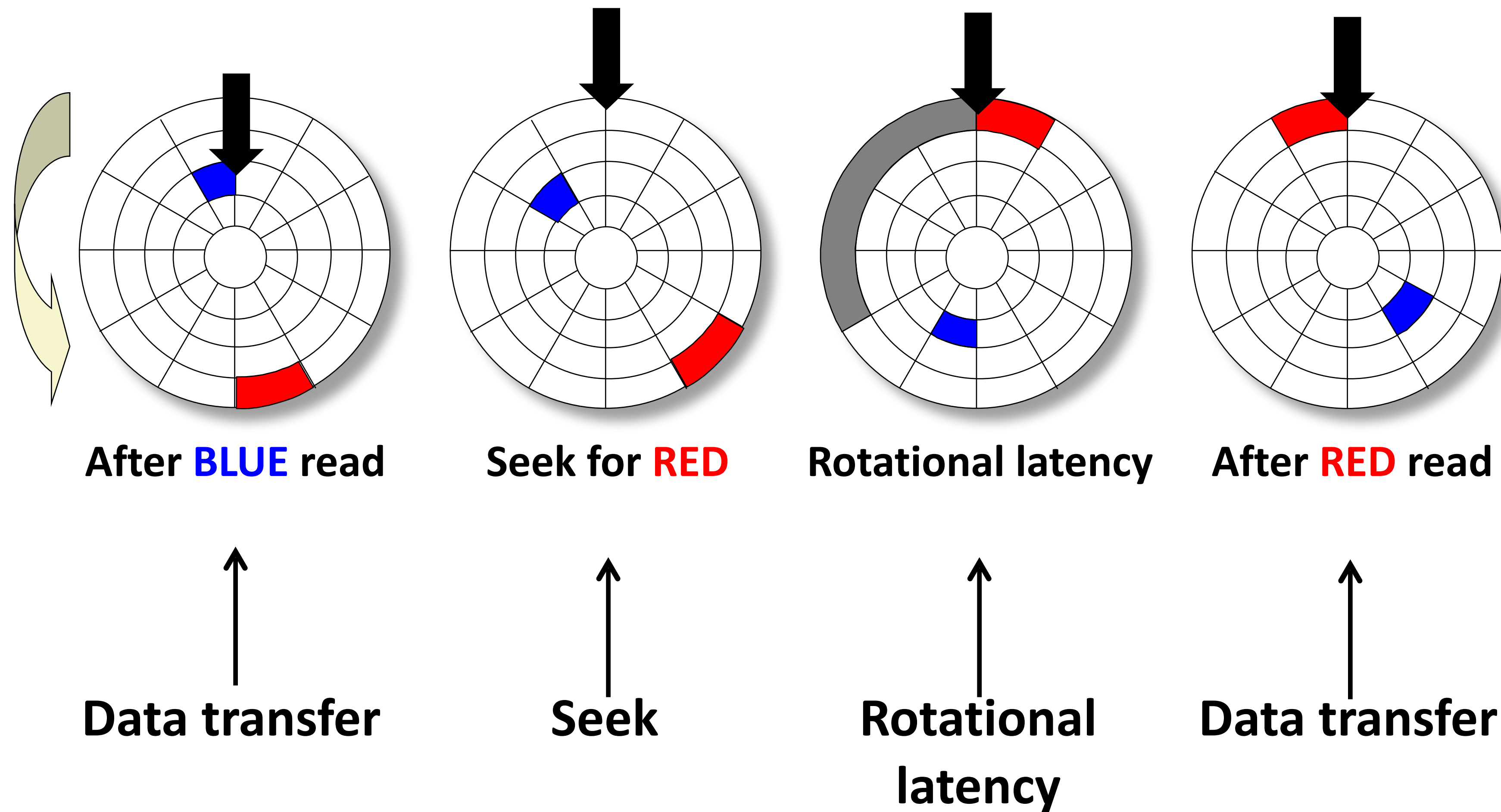
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

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

# Disk Operation (Multi-Platter View)



# Disk Access – Service Time Components



# Disk Access Time

Average time to access some target sector approximated by:

$$T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$$

**Seek time** ( $T_{\text{avg seek}}$ )

Time to position heads over cylinder containing target sector.

Typical  $T_{\text{avg seek}}$  is 3—9 ms

**Rotational latency** ( $T_{\text{avg rotation}}$ )

Time waiting for first bit of target sector to pass under r/w head.

$$T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$$

Typical rotational rate = 7,200 RPMs

**Transfer time** ( $T_{\text{avg transfer}}$ )

Time to read the bits in the target sector.

$$T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg \# sectors/track}) \times 60 \text{ secs}/1 \text{ min}$$

time for one rotation (in minutes)      fraction of a rotation to be read



# Disk Access Time Example

Given:

Rotational rate = 7,200 RPM

Average seek time = 9 ms

Avg # sectors/track = 400

Derived:

$$T_{\text{avg rotation}} = 1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms}$$

$$T_{\text{avg transfer}} = 60/7200 \times 1/400 \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$$

$$T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$$

Important points:

Access time dominated by seek time and rotational latency.

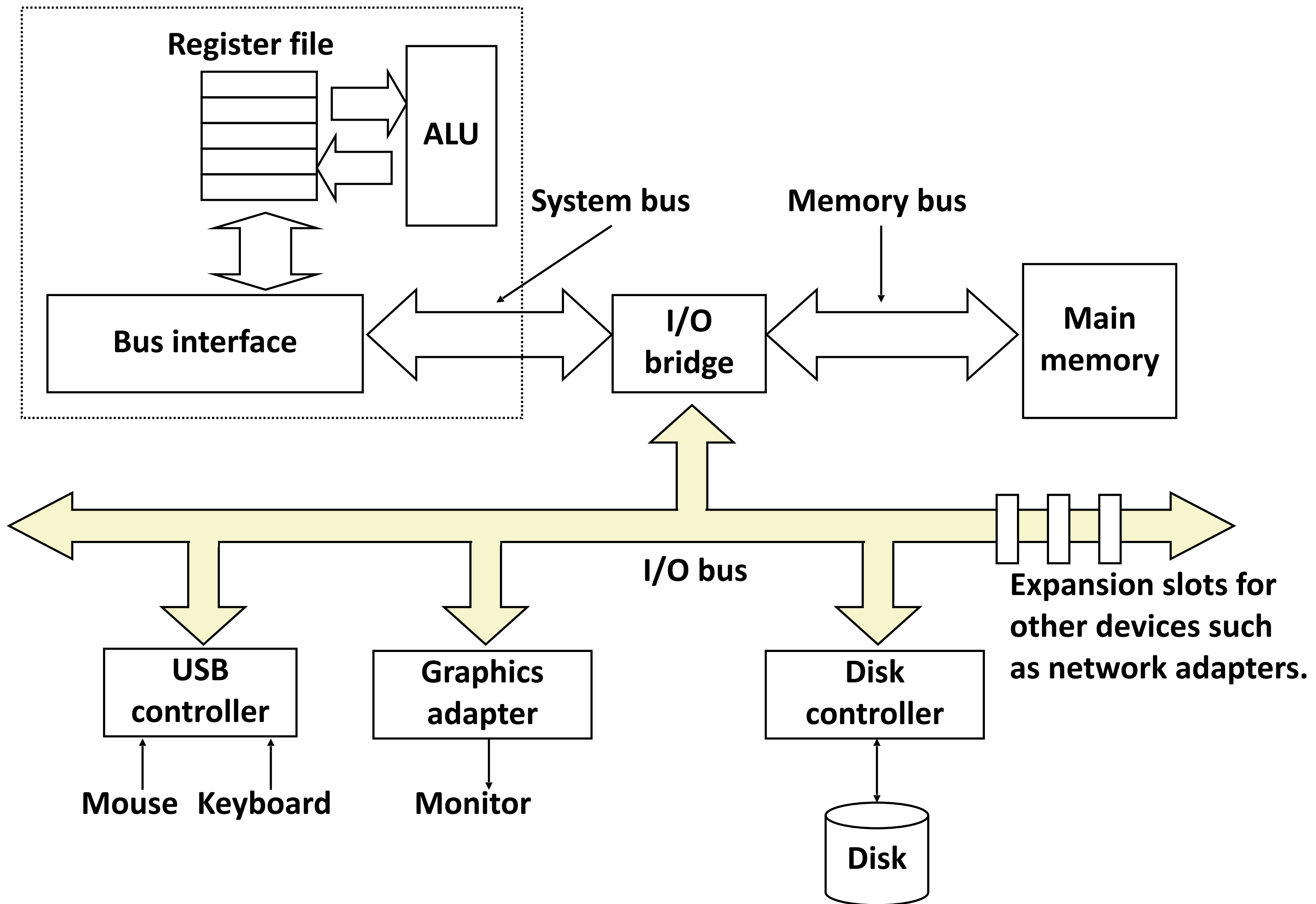
First bit in a sector is the most expensive, the rest are free.

*SRAM access time is about 4 ns/doubleword, DRAM about 60 ns*

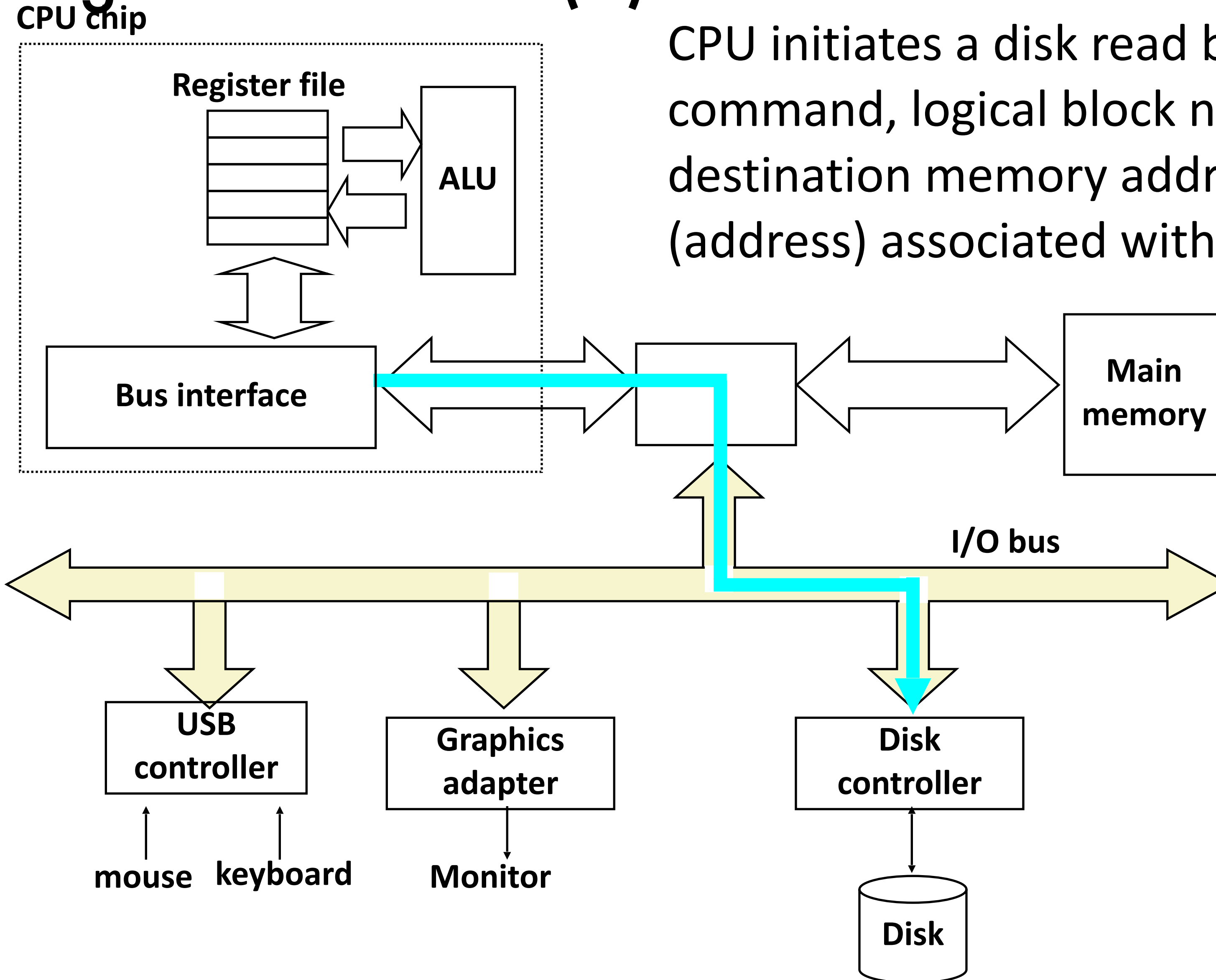
*Disk is about 40,000 times slower than SRAM,*

*2,500 times slower than DRAM.*

# I/O Bus

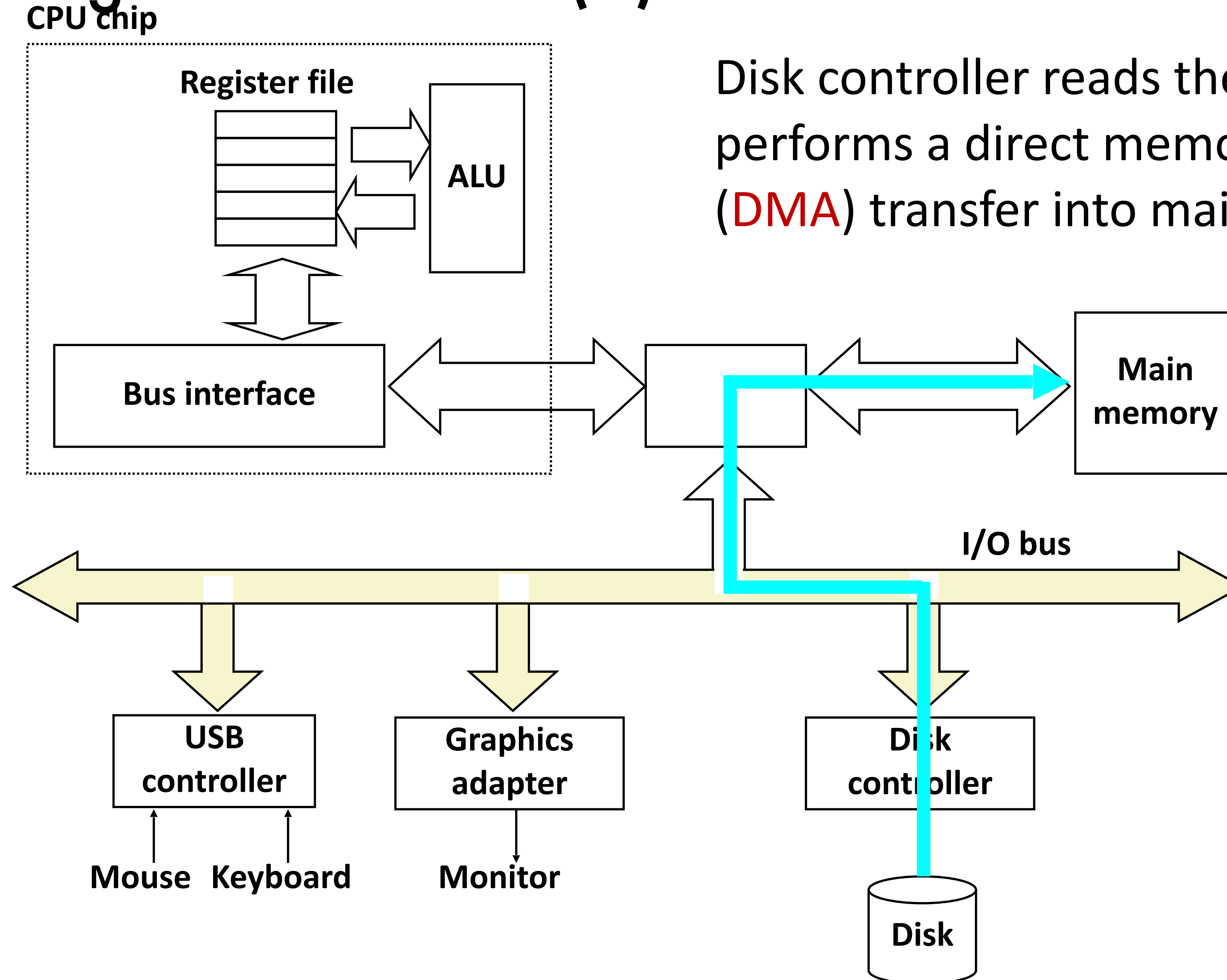


# Reading a Disk Sector (1)



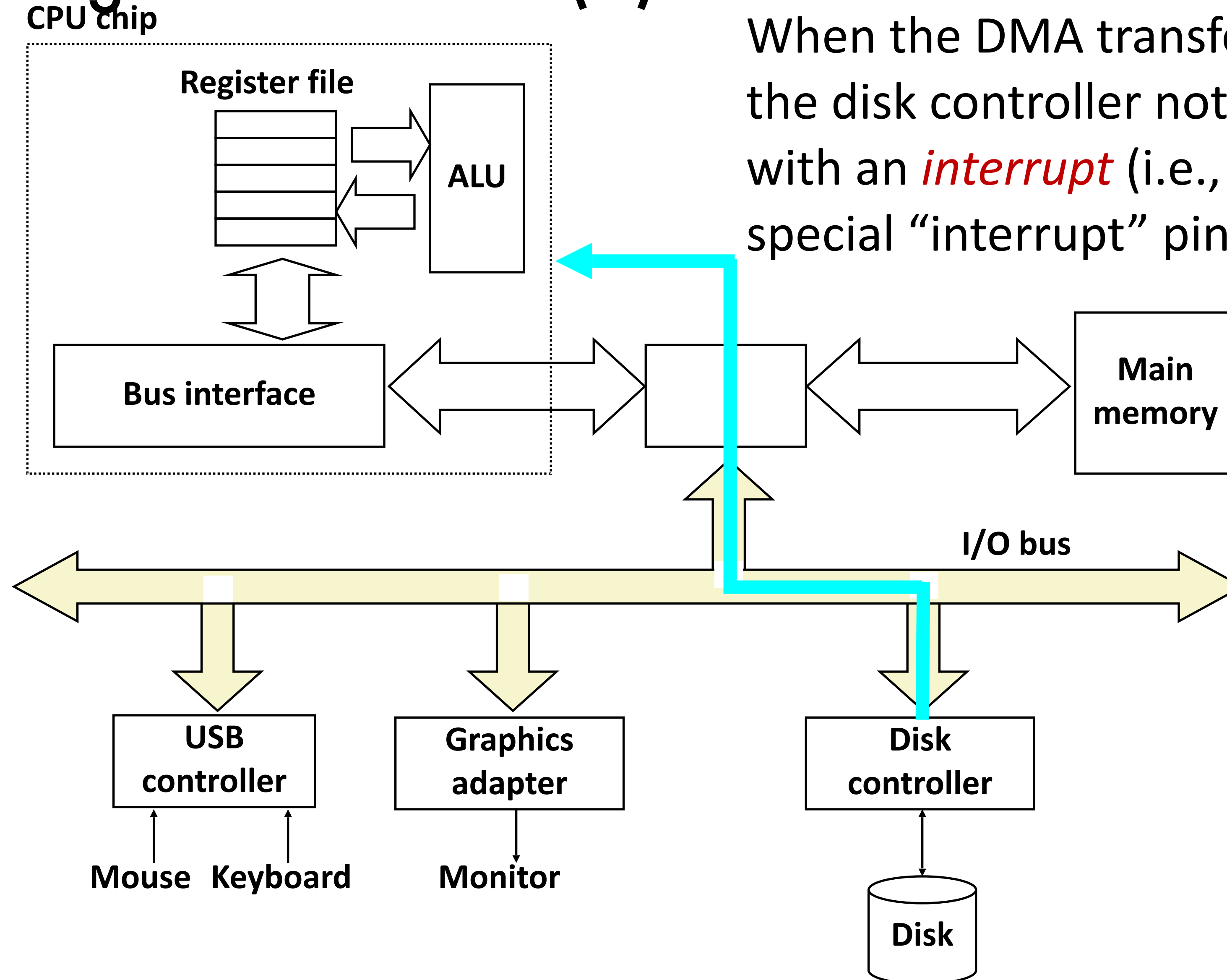
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a **port** (address) associated with disk controller.

# Reading a Disk Sector (2)



Disk controller reads the sector and performs a direct memory access (**DMA**) transfer into main memory.

# Reading a Disk Sector (3)



When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on the CPU).

# Processor Performance

**Q:** *How fast can a processor process a program?*

- Modern CPUs can run millions of instructions per second!
  - ISA tells #**clock cycles** per instruction
  - CPU's **clock rate** helps map that to runtime (ns)
- Alas, most programs do not keep CPU always busy:
  - Memory access commands **stall** the processor; ALU and CU are *idle* during DRAM-register transfer
  - Worse, data may not be in DRAM—wait for (disk) I/O!
  - So, actual *runtime* of program may be OOM higher than what clock rate calculation suggests

**Key Principle:** Optimizing access to DRAM and use of processor caches is critical for processor performance!



# Summary

- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called *locality*.
- Memory hierarchies based on *caching* close the gap by exploiting locality.

# Review Questions

- What is an ISA?
- What are the 3 main kinds of commands in an ISA?
- Why do CPUs have both registers and caches?
- Why is it typically impossible for data processing programs to achieve 100% processor utilization?
- Which of these memory hierarchy layers is the most expensive: CPU cache, DRAM, flash disks, or magnetic hard disks?
- Which of the above layers is the slowest for data access?
- Which library helps ML users avoid need for writing GPU cache-aware computations?