DSC 204a Scalable Data Systems

- Haojian Jin

Company's 1000-table database on data lake with 100k attributes

DataFrame API

EVER

Meme idea credit: https://datasystemsfun.tumblr.com/
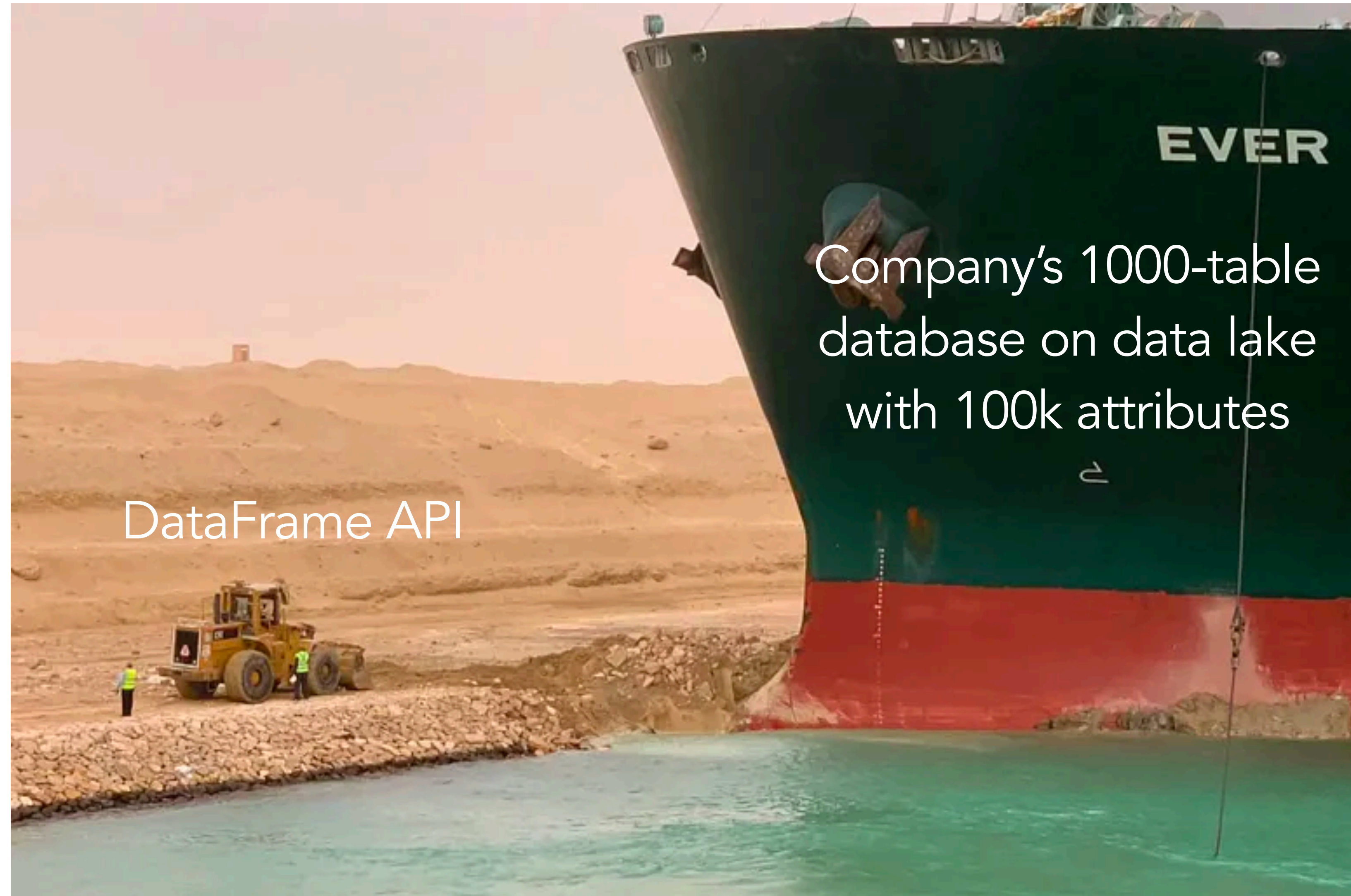
Logistics (Apr. 10)

- We have released the first assignment.
- Be aware about your AWS credits.
- Canvas for important links.
  - Slack for communications with less latency.

Recap: Peer Instruction Activity (About 1min per 1pt)

Here is a custom float6 representation and interpretation.

sign (1 bit)	exponent (2 bits)		fraction (3 bits)		
0	0	1	0	1	0

Bit index: 5   4   3   2   1   0
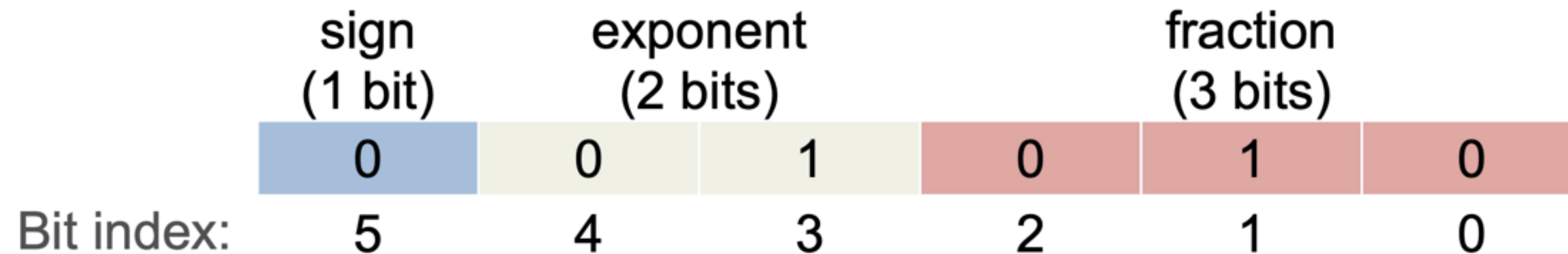
$$(-1)^{sign} \times 2^{exponent-2} \times \left(1 + \sum_{i=1}^{3} b_{3-i} 2^{-i}\right)$$

3. [3 x 3pts] What is the decimal real number representation of these float6 bit sequences:

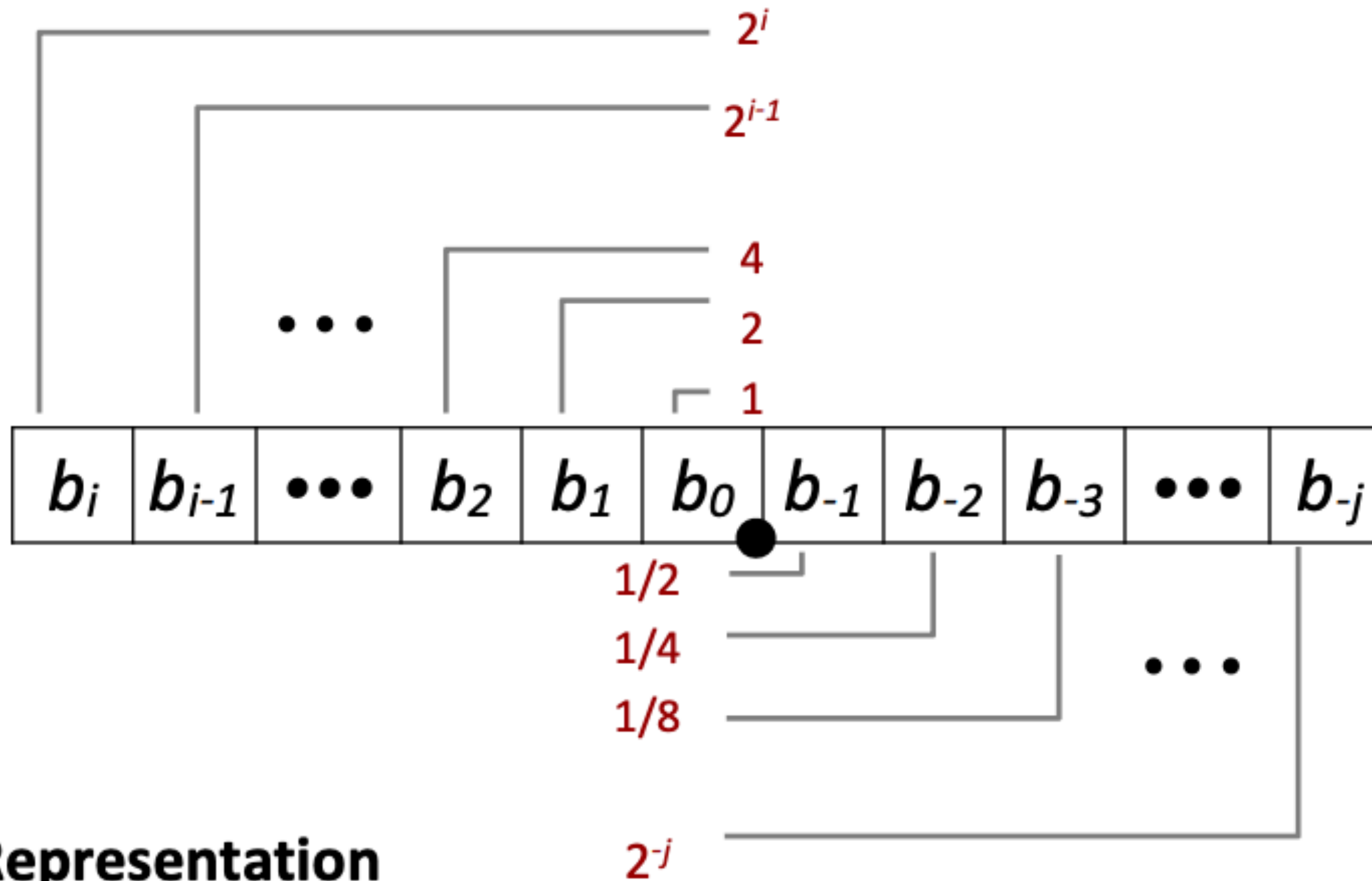A. 001010

B. 111101

A. 5/8, i.e., 0.625

B. -13/4, i.e., -3.25

3

Two-complement: Simple Example

		-16	8	4	2	1		
10	=	0	1	0	1	0		8+2 = 10

		-16	8	4	2	1		
-10	=	1	0	1	1	0		-16+4+2 = -10

Fractional Binary Numbers

- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:
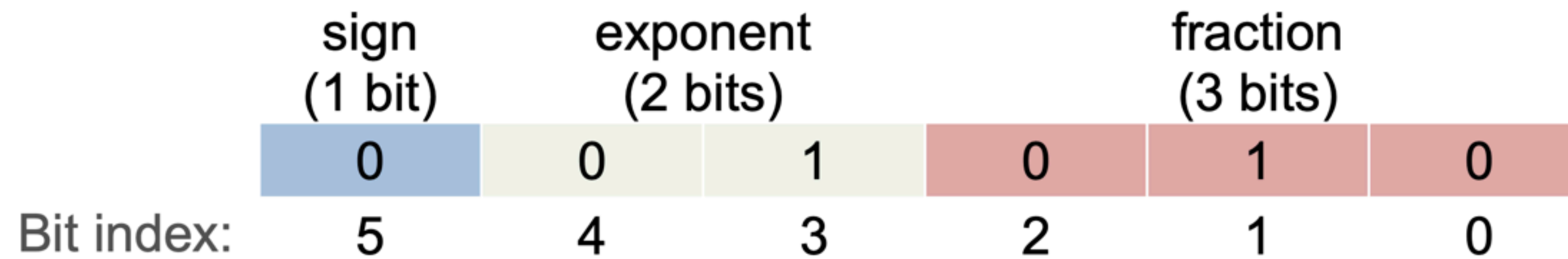
$$\sum_{k=-j}^{i} b_k \times 2^k$$

An alternative representation?

	Sign	Integer		Fraction		
	0	0	1	0	1	0
Bit index:	5	4	3	2	1	0
	-4	2	1	1/2	1/4	1/8

~~1+1/4= 1.25~~   A. 5/8, i.e., 0.625

Can only represent numbers from -4 (100000) to 3.875 (011111).

Revisit the representation (scientific notation)

sign (1 bit)	exponent (2 bits)		fraction (3 bits)		
0	0	1	0	1	0

Bit index: 5   4   3   2   1   0

$$(-1)^{sign} \times 2^{exponent-2} \times (1 + \sum_{i=1}^{3} b_{3-i} 2^{-i})$$

$$(-1)^0 \cdot 2^{(1-2)} \cdot \left(1 + 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8}\right)$$
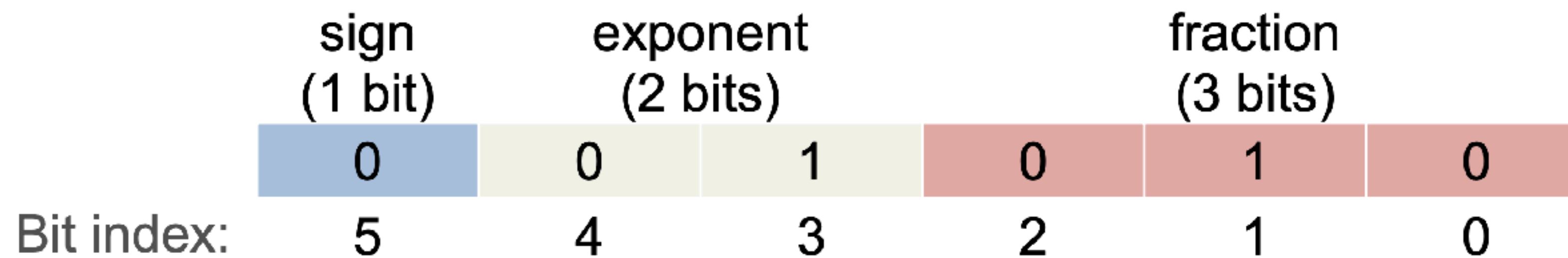
A. 5/8, i.e., 0.625

7

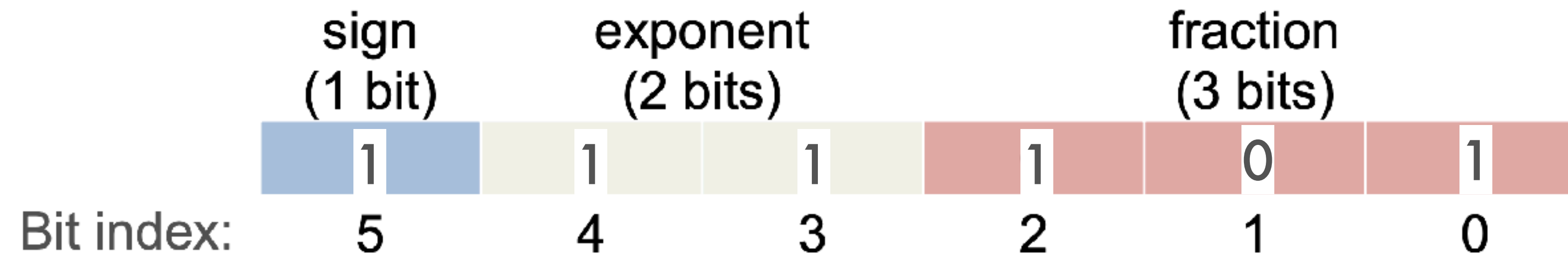# Reverse the representation (scientific notation)

$$0.625_{10} =$$

$$0.625_{10} = 0.101_2$$

$$0.625_{10} = 0.101_2 = 1.01 \cdot 2^{-1}$$

sign (1 bit)	exponent (2 bits)		fraction (3 bits)		
0	0	1	0	1	0

Bit index: 5, 4, 3, 2, 1, 0

$$(-1)^{sign} \times 2^{exponent-2} \times \left(1 + \sum_{i=1}^{3} b_{3-i} 2^{-i}\right)$$

8

Another representation (scientific notation)

sign (1 bit)	exponent (2 bits)		fraction (3 bits)		
1	1	1	1	0	1

Bit index: 5   4   3   2   1   0

$$(-1)^{sign} \times 2^{exponent-2} \times (1 + \sum_{i=1}^{3} b_{3-i} 2^{-i})$$

111101 => -3.25

$$(-1)^1 \cdot 2^{3-2} \cdot (1 + \frac{1}{2} + \frac{1}{8})$$

Reverse the representation (scientific notation)

$$3.25_{10} = 11.01_2$$

8	4	2	1	1/2	1/4	1/8
0	0	1	1	0	1	0

$$3.25_{10} = 11.01_2 = 1.101 \cdot 2^1$$

$$-3.25_{10} =$$

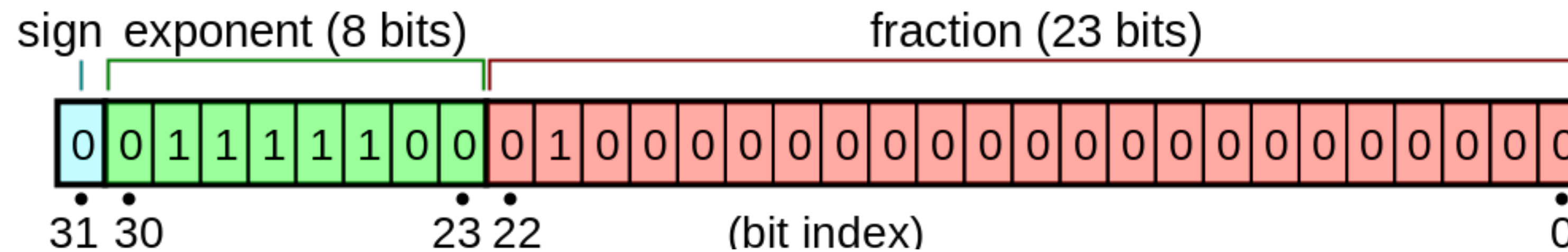| sign (1 bit)	exponent (2 bits)		fraction (3 bits)		
1	1	1	1	0	1

Bit index: 5   4   3   2   1   0

$$(-1)^{sign} \times 2^{exponent-2} \times (1 + \sum_{i=1}^{3} b_{3-i} 2^{-i})$$

Digital Representation of Data
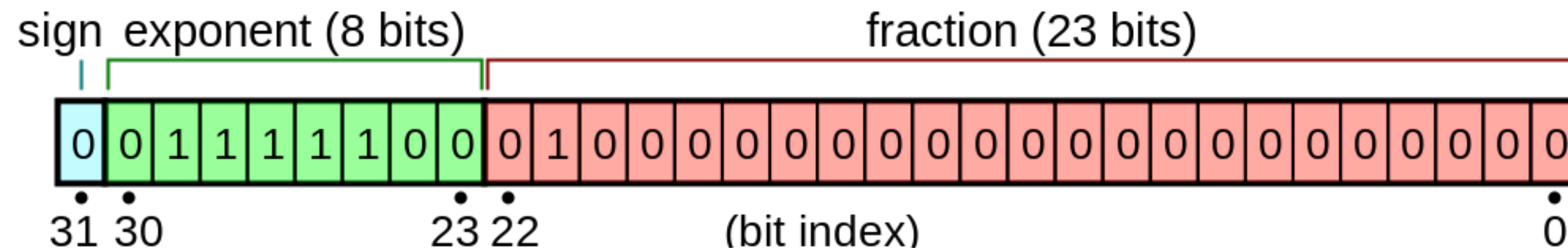
- **Float**:
  - Standard IEEE format for single (aka binary32):

$$(-1)^{sign} \times 2^{exponent-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

$$(-1)^0 \times 2^{124-127} \times (1 + 1 \cdot 2^{-2}) = (1/8) \times (1 + (1/4)) = 0.15625$$

Digital Representation of Data.

$$(-1)^{sign} \times 2^{exponent-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

- Special encodings recognized:
  - Exponent 0x00 and fraction 0 is "+/- 0".
    - 0 00000000 00000000000000000000000 = +0
  - Exponent 0xFF and fraction 0 is +/- "Infinity"
    - 0 11111111 00000000000000000000000 = +INF
  - Not a number/Undefined (Dividing 0 by 0)
    - 0 11111111 10000000000000000000000 = NaN

Digital Representation of Data. Why?
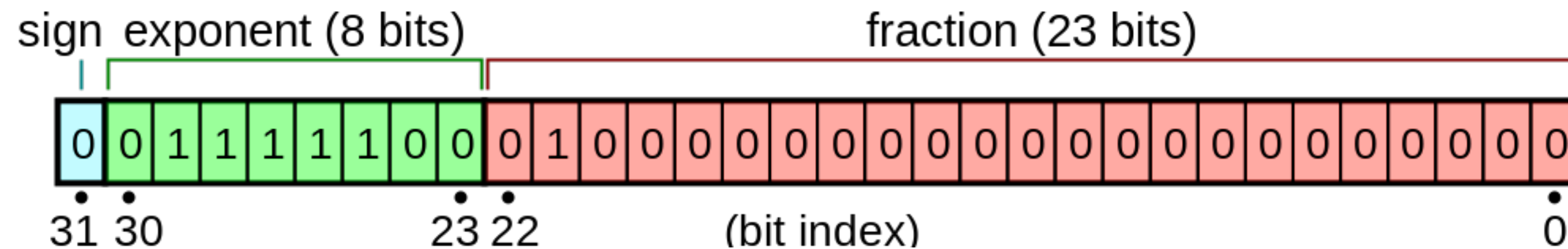
- Exponent for Range, Fraction (significant) for Accuracy.
  - Max: 0 11111110 11111111111111111111111 =>
  - $2^{(254-127)}*(1.111111\ldots)_2 = 2^{(254-127)}*\left(1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{8388608}\right)$
  - Max = 2*2^127 = 2^128 = 3.4 x 10^{38}
  - Why not 0 11111111 11111111111111111111111?
- Scientific notation.
- Dynamic accuracy.

Examples in the final exam?
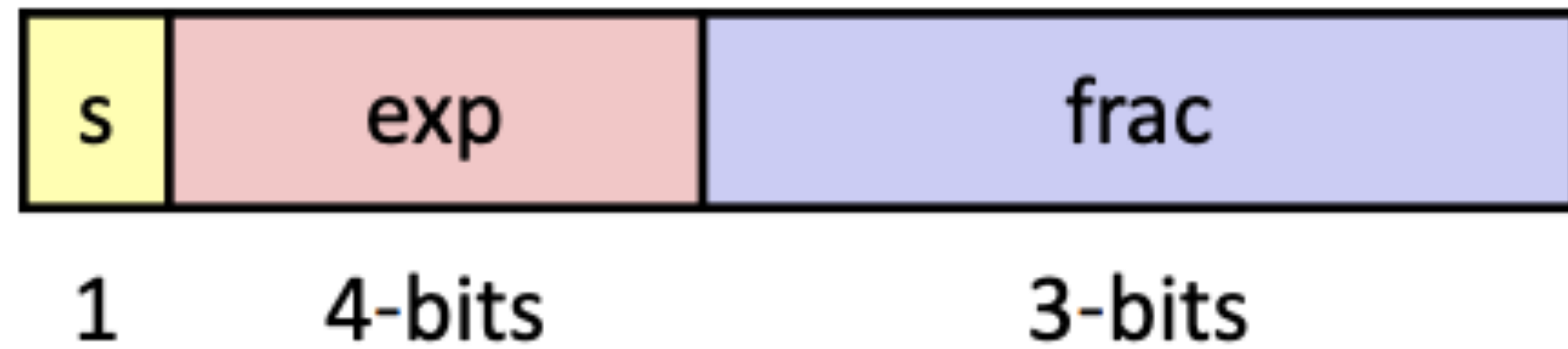
Where are we in the class?

Foundations of Data Systems (2 weeks)

- Digital representation of Data → Computer Organization → **Memory hierarchy** → Process → Storage

Scaling Distributed Systems (3 weeks)

- **Cloud** → Distributed storage → Partition and replication (HDFS) → Distributed computation

Data Processing and Programming model (5 weeks)

- Data Models evolution → Data encoding evolution → → IO & Unix Pipes → Batch processing (MapReduce) → Stream processing (Spark)

Today

The memory abstraction
Locality of reference
The memory hierarchy
Storage technologies and trends

Abstract Computer Parts and Data

Processor

Control Unit

Arithmetic & Logic Unit

Registers

Caches

Retrieve; Process

Store; Retrieve

Dynamic Random Access Memory (DRAM)

Bus

Input Devices

Output Devices

Secondary Storage (e.g., Magnetic hard disk, Flash SSD, etc.)

Input; Output; Retrieve

Store; Retrieve

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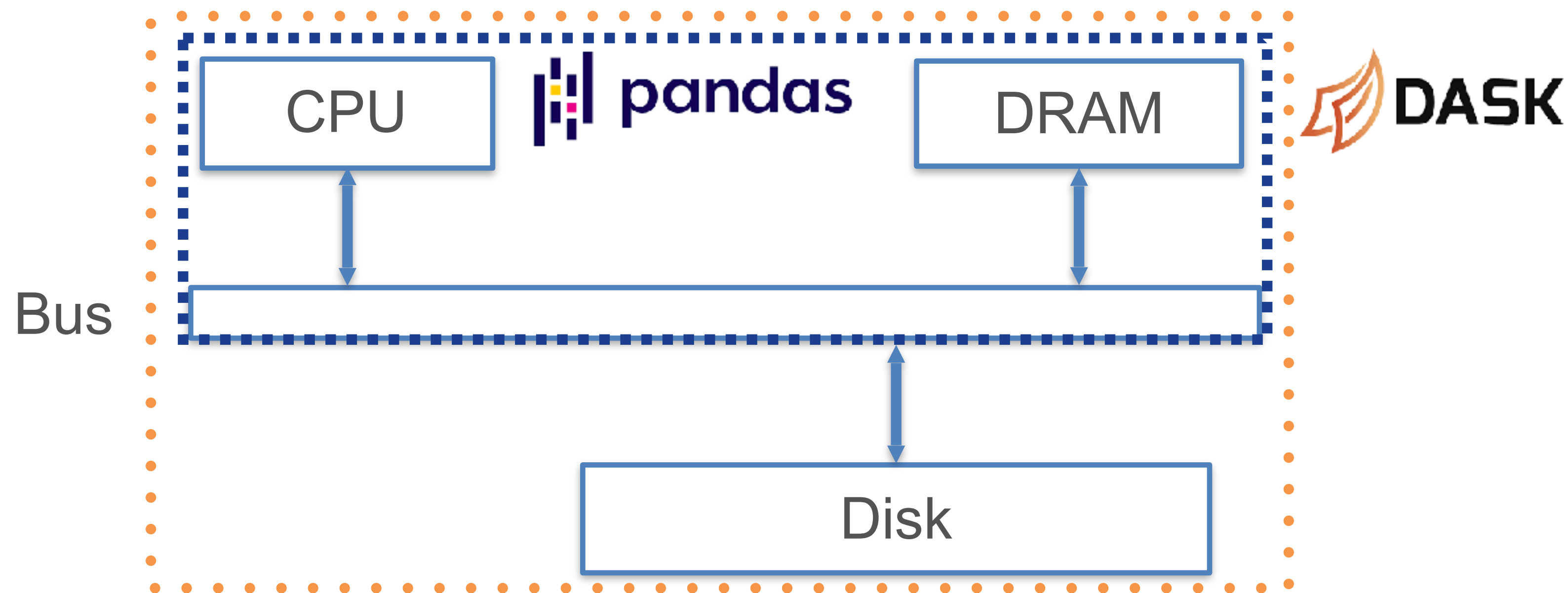
- **Tradeoff:** Dask may throw memory configuration issues. :)
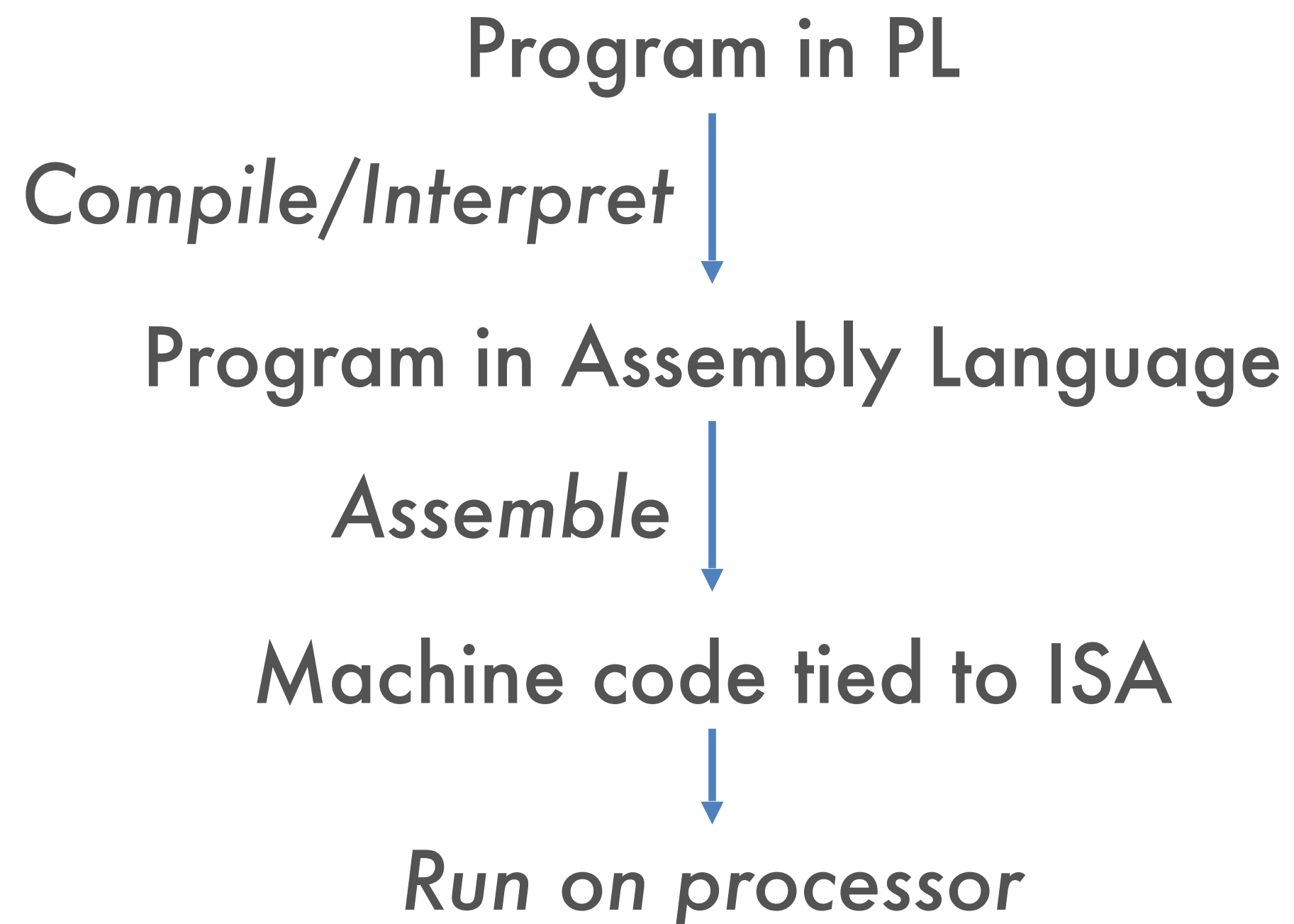
Instruction

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

Program in PL

Compile/Interpret

Program in Assembly Language

Assemble

Machine code tied to ISA

Run on processor

```
80483b4:    55                          push    %ebp
80483b5:    89 e5                       mov     %esp,%ebp
80483b7:    83 e4 f0                    and     $0xfffffff0,%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
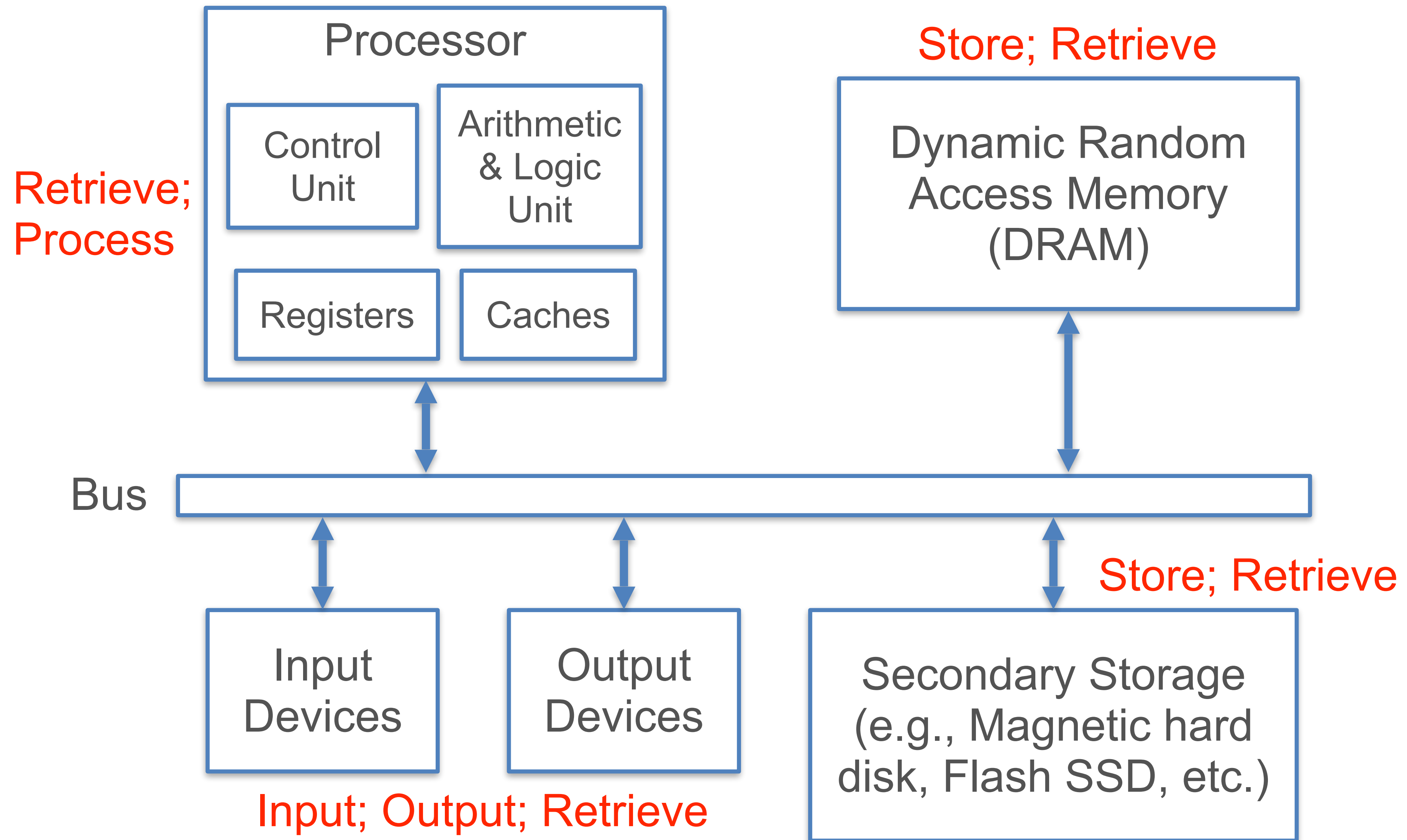**movq %rax, %rsp**

"Store" operation

Read

Transfer data from CPU to memory
movq %rsp, %rax

"Load" operation

Abstract Computer Parts and Data

Processor

Control Unit

Arithmetic & Logic Unit

Registers

Caches

Retrieve; Process

Store; Retrieve

Dynamic Random Access Memory (DRAM)

Bus

Store; Retrieve

Input Devices

Output Devices

Secondary Storage (e.g., Magnetic hard disk, Flash SSD, etc.)
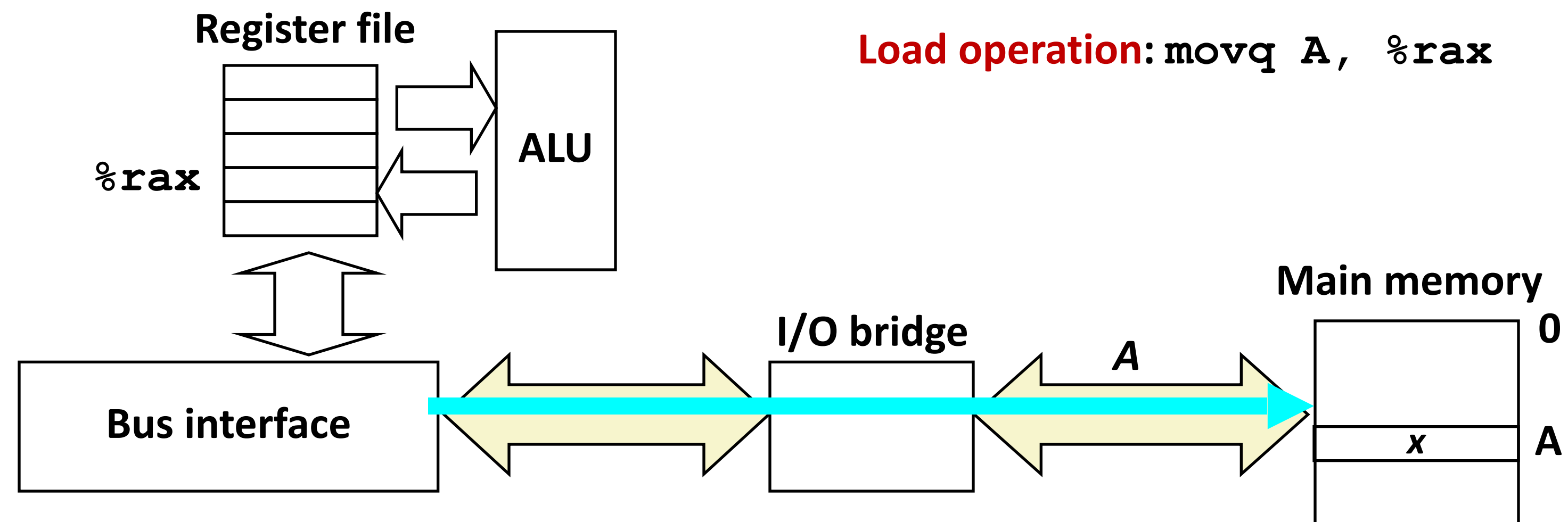
Input; Output; Retrieve

23

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.

CPU chip

Register file

ALU

System bus

Memory bus

Bus interface

I/O bridge

Main memory

Memory Read Transaction (1)

Register file

Load operation: `movq A, %rax`

ALU

%rax

Main memory

I/O bridge

A

0

Bus interface

x   **A**
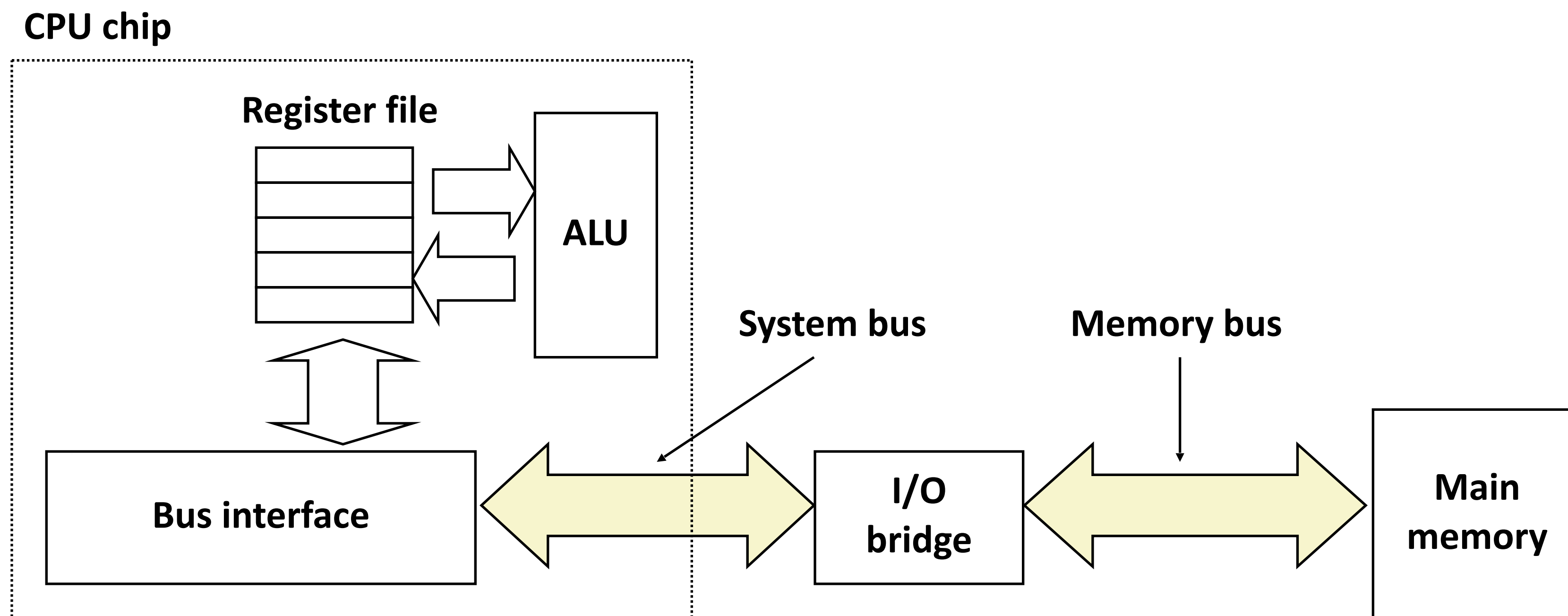
CPU places address A on the memory bus.

25

Memory Read Transaction (2)

Register file

%rax

ALU

Load operation: `movq A, %rax`

Main
memory

I/O bridge

x

0

Bus interface

x

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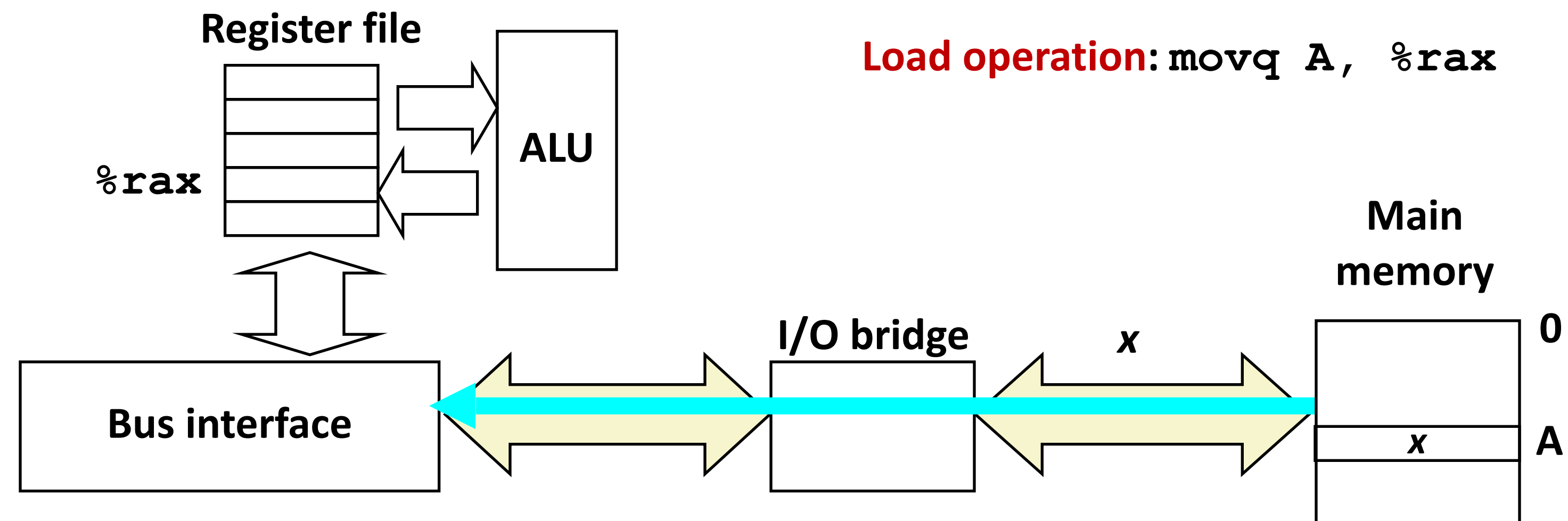
Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Memory Read Transaction (3)

Register file

Load operation: `movq A, %rax`

%rax | *x*

ALU

Bus interface

I/O bridge

Main memory

0

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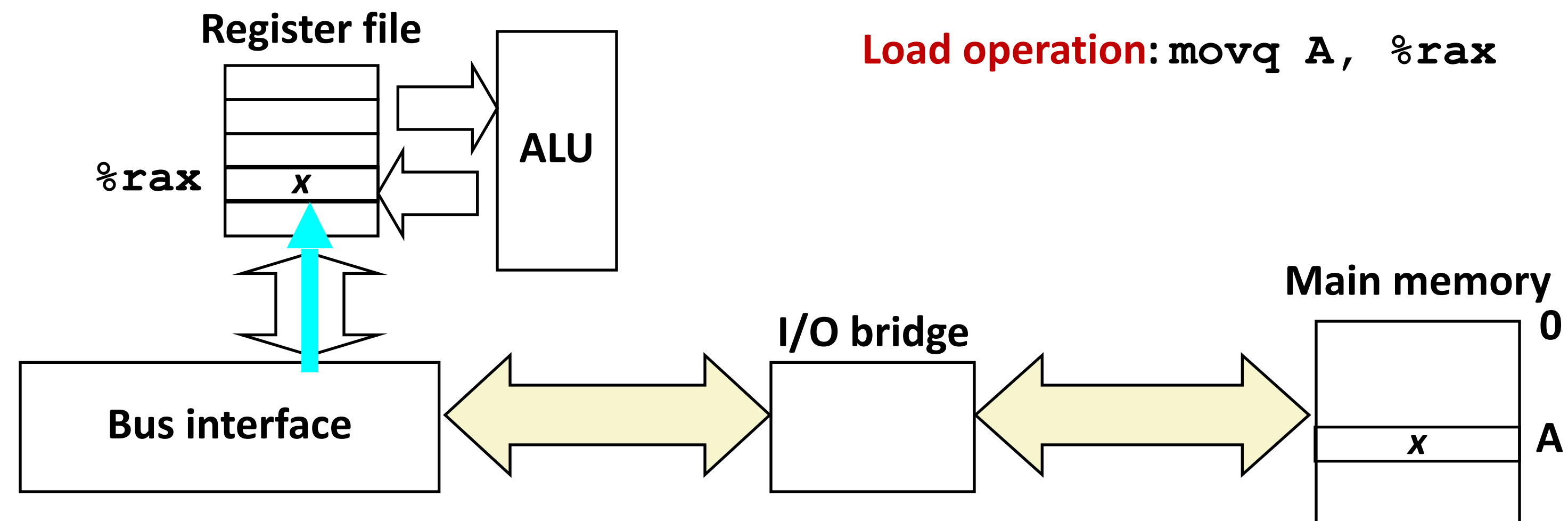
CPU reads word x from the bus and copies it into register `%rax`.

Memory Write Transaction (1)

Store operation: `movq %rax, A`

Register file

%rax   *y*

ALU

Bus interface

I/O bridge

Main memory
0

A

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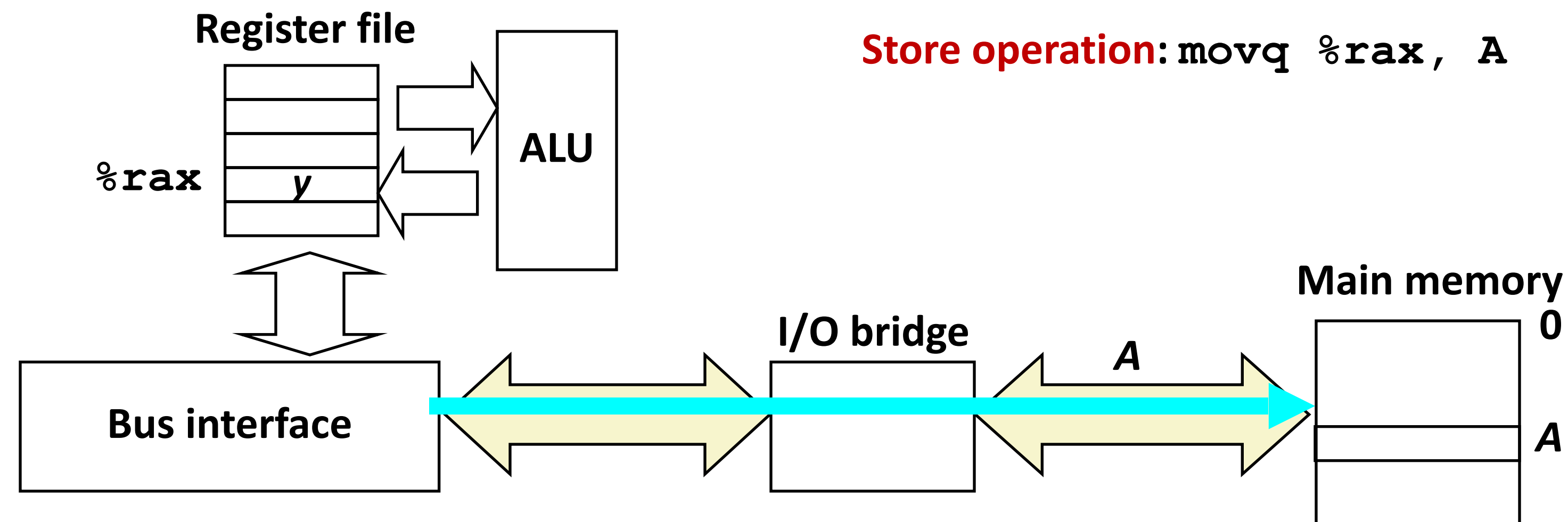
CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Memory Write Transaction (2)

Register file

Store operation: `movq %rax, A`

ALU

`%rax`   *y*

Main memory

I/O bridge   *y*

0

Bus interface

A
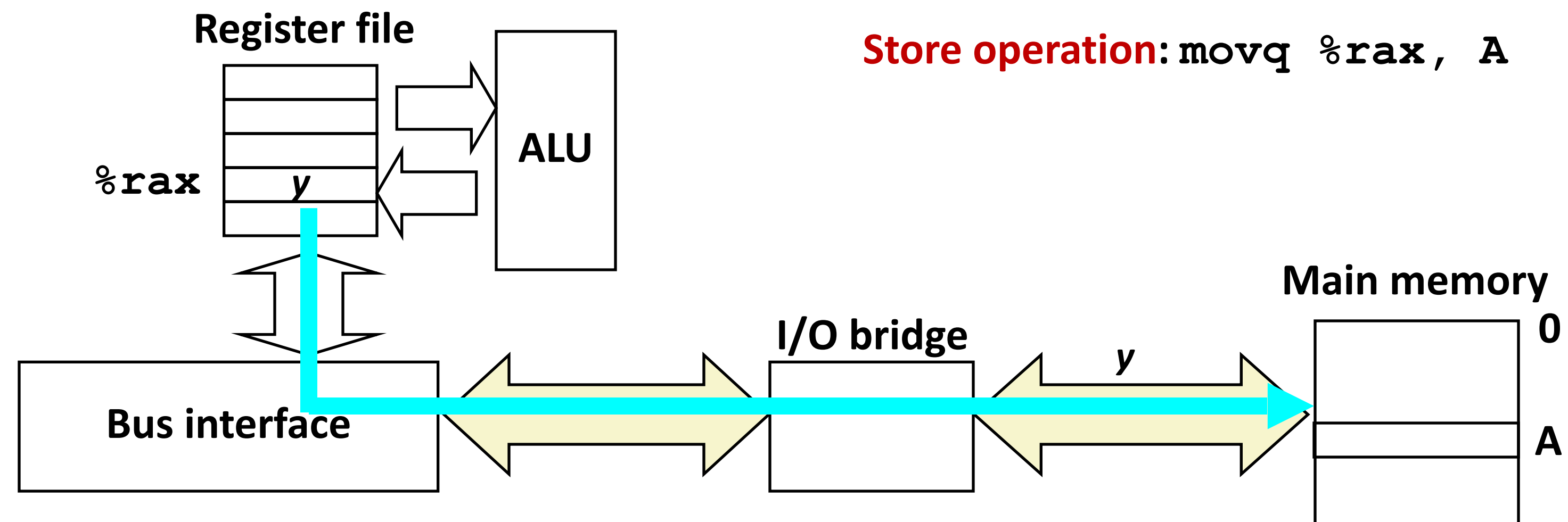
CPU places data word y on the bus.

Memory Write Transaction (3)

Store operation: `movq %rax, A`
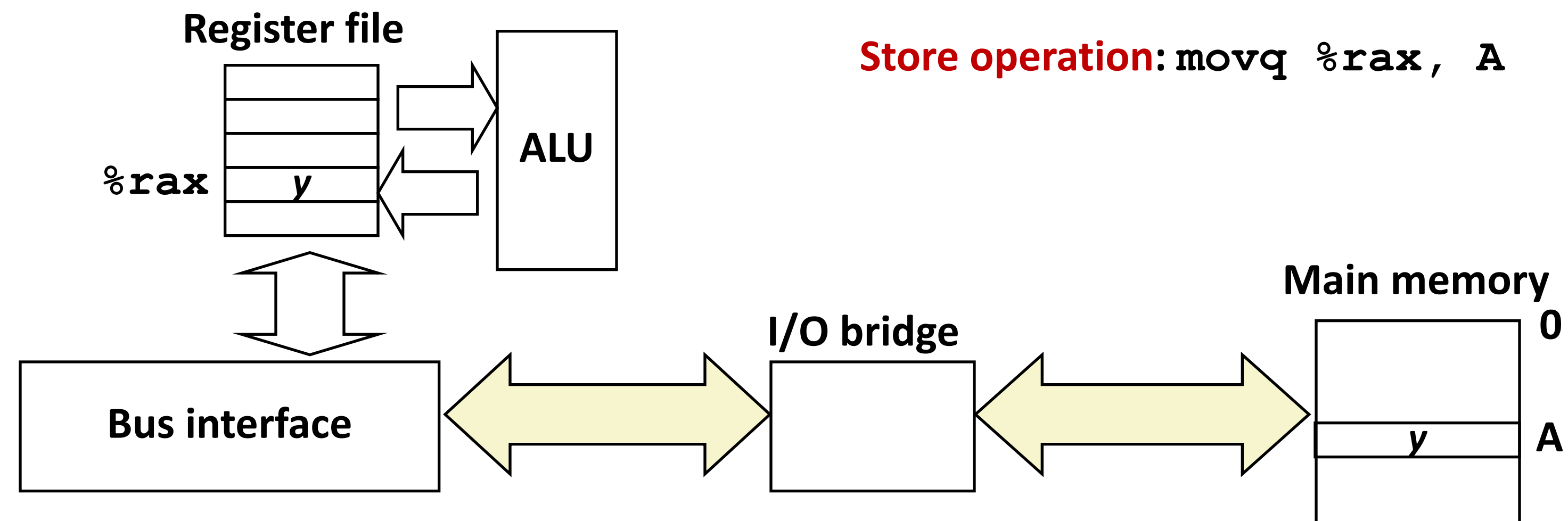
Main memory reads data word y from the bus and stores it at address A.

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

If interested in more details: https://www.youtube.com/watch?v=cNN_tTXABUA

```
80483b5:        89 e5                       mov     %esp,%ebp
80483b7:        83 e4 f0                    and     $0xfffffff0,%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
```

Example

Q: What does this program do when run with 'python'?

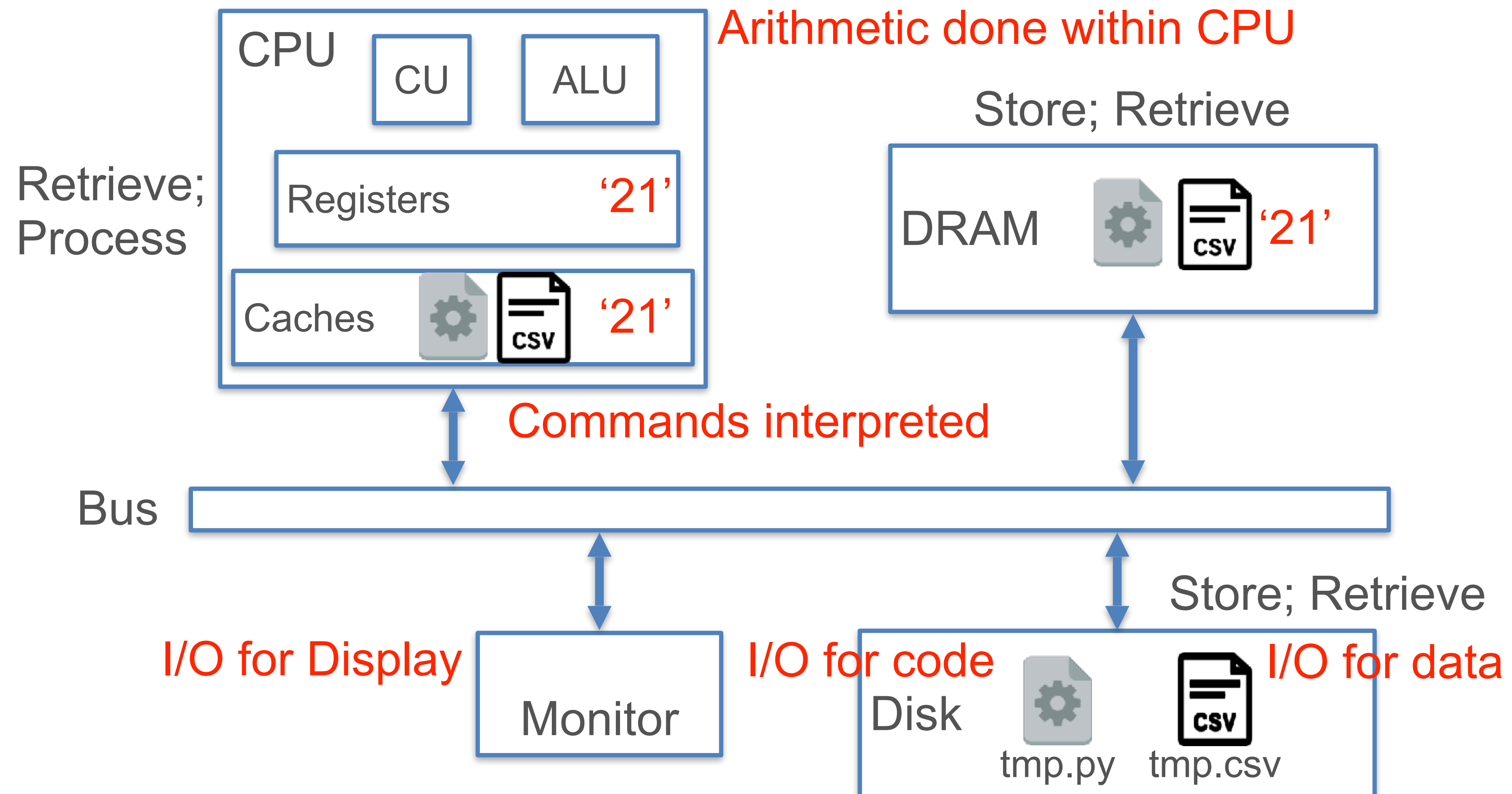*(Assume tmp.csv is in current working directory)*

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

CPU

CU   ALU

Arithmetic done within CPU

Registers   '21'

Store; Retrieve

DRAM   '21'

Retrieve;
Process

Caches   '21'

Commands interpreted

Bus

I/O for Display

Monitor

I/O for code

Disk

I/O for data

Store; Retrieve

tmp.py   tmp.csv

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.

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

```c
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];
}
```
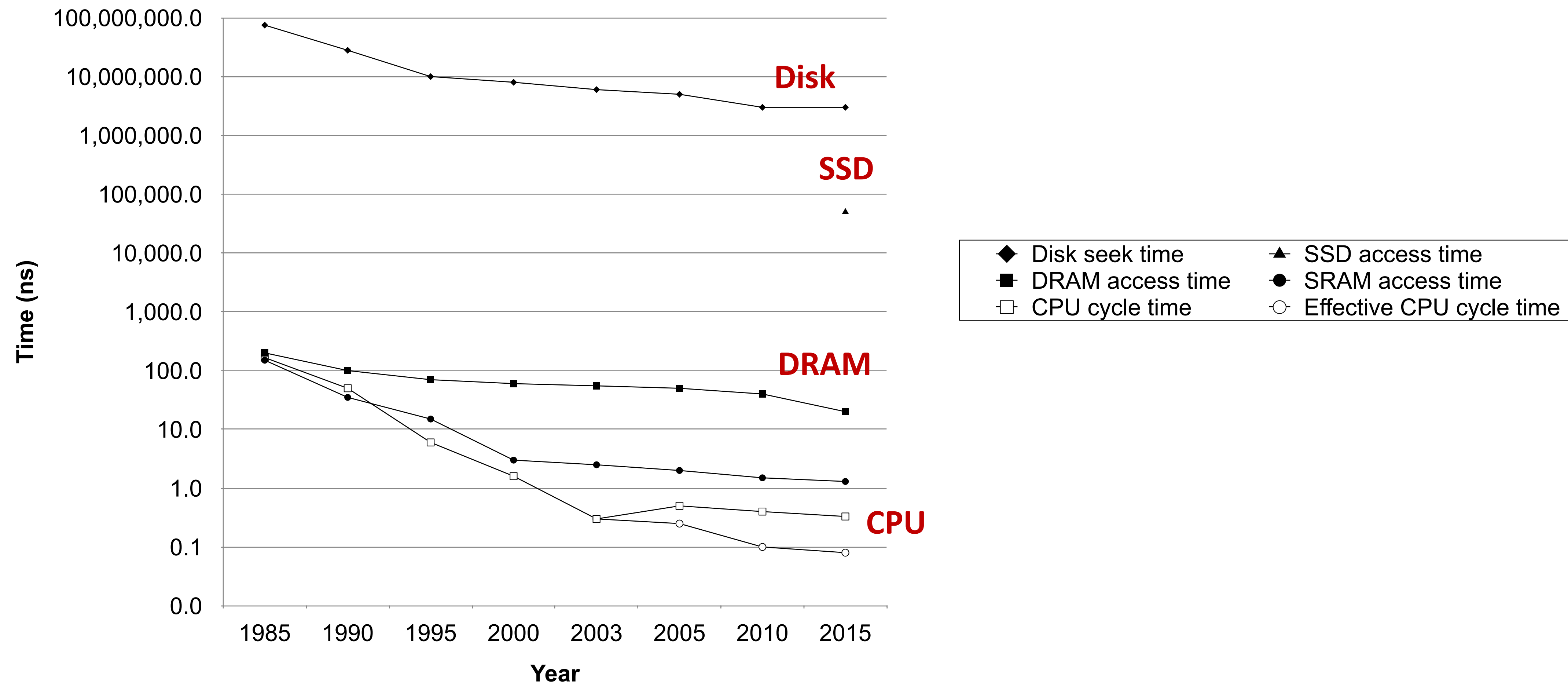
4.3 milliseconds

```c
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.

Locality to the Rescue!

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

Locality

Principle of Locality: Many Programs tend to use data and instructions with addresses near or equal to those they have used recently.
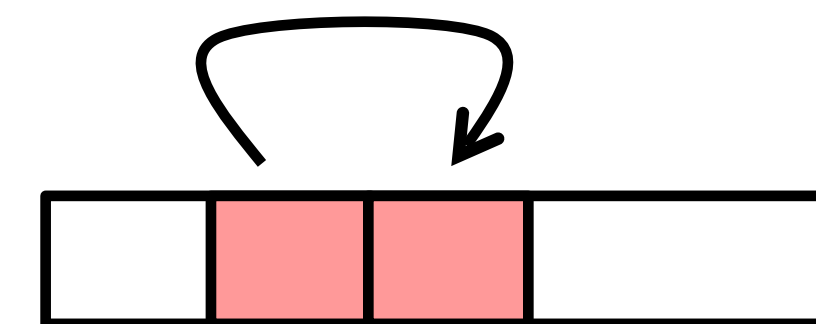
Temporal locality:
   Recently referenced items are likely
   to be referenced again in the near future
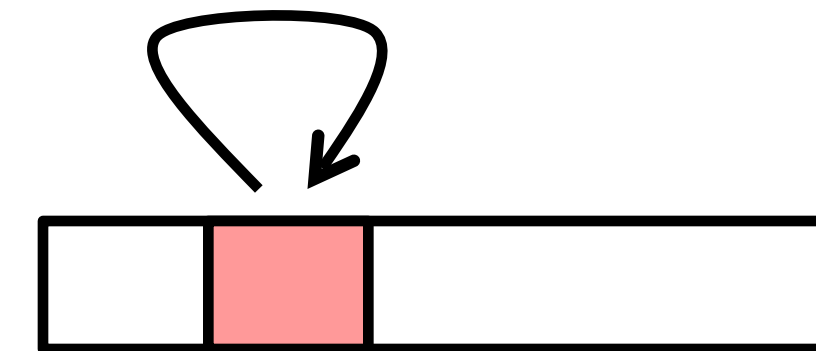
Spatial locality:
   Items with nearby addresses tend
   to be referenced close together in time

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).   **spatial**

Reference variable **sum** each iteration.   **temporal**

Instruction references

Reference instructions in sequence.   **spatial**

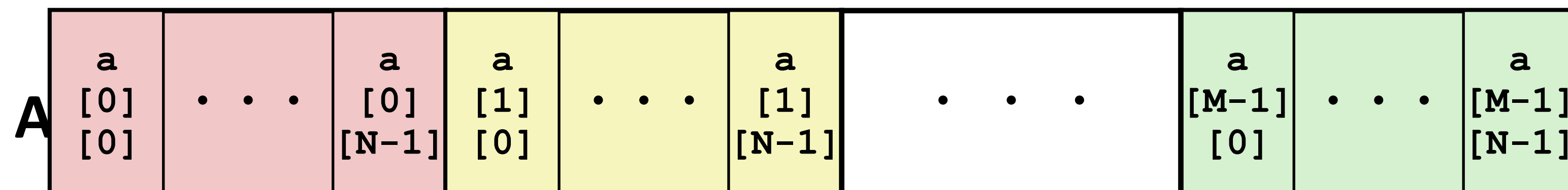Cycle through loop repeatedly.   **temporal**

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;
}
```

A
| a[0][0] | • • • | a[0][N-1] | a[1][0] | • • • | a[1][N-1] | • • • | a[M-1][0] | • • • | a[M-1][N-1] |

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;
}
```
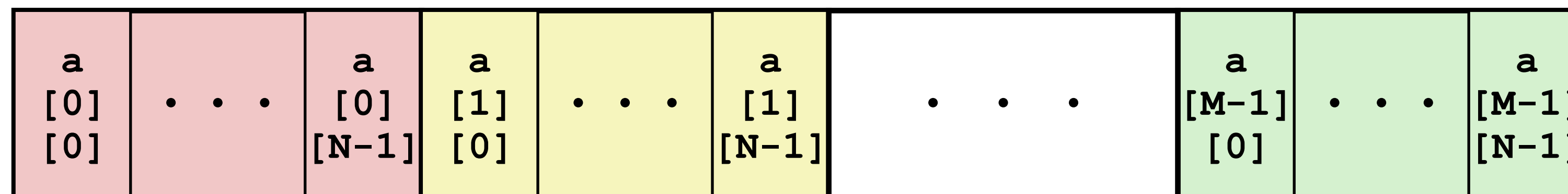
Answer: no, unless...

M is very small

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

a [0] [0]	· · ·	a [0] [N-1]	a [1] [0]	· · ·	a [1] [N-1]	· · ·	a [M-1] [0]	· · ·	a [M-1] [N-1]

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
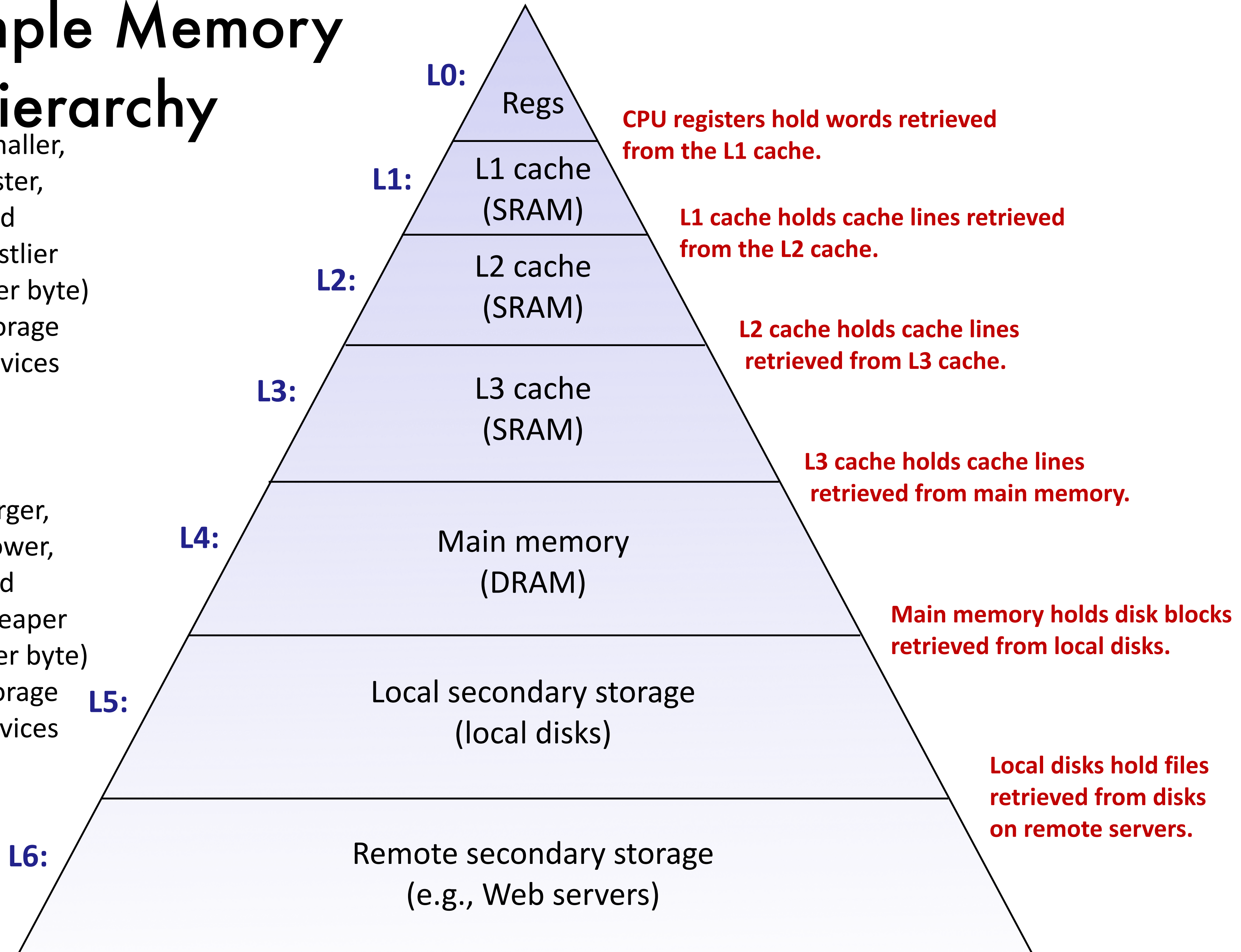Storage technologies and trends

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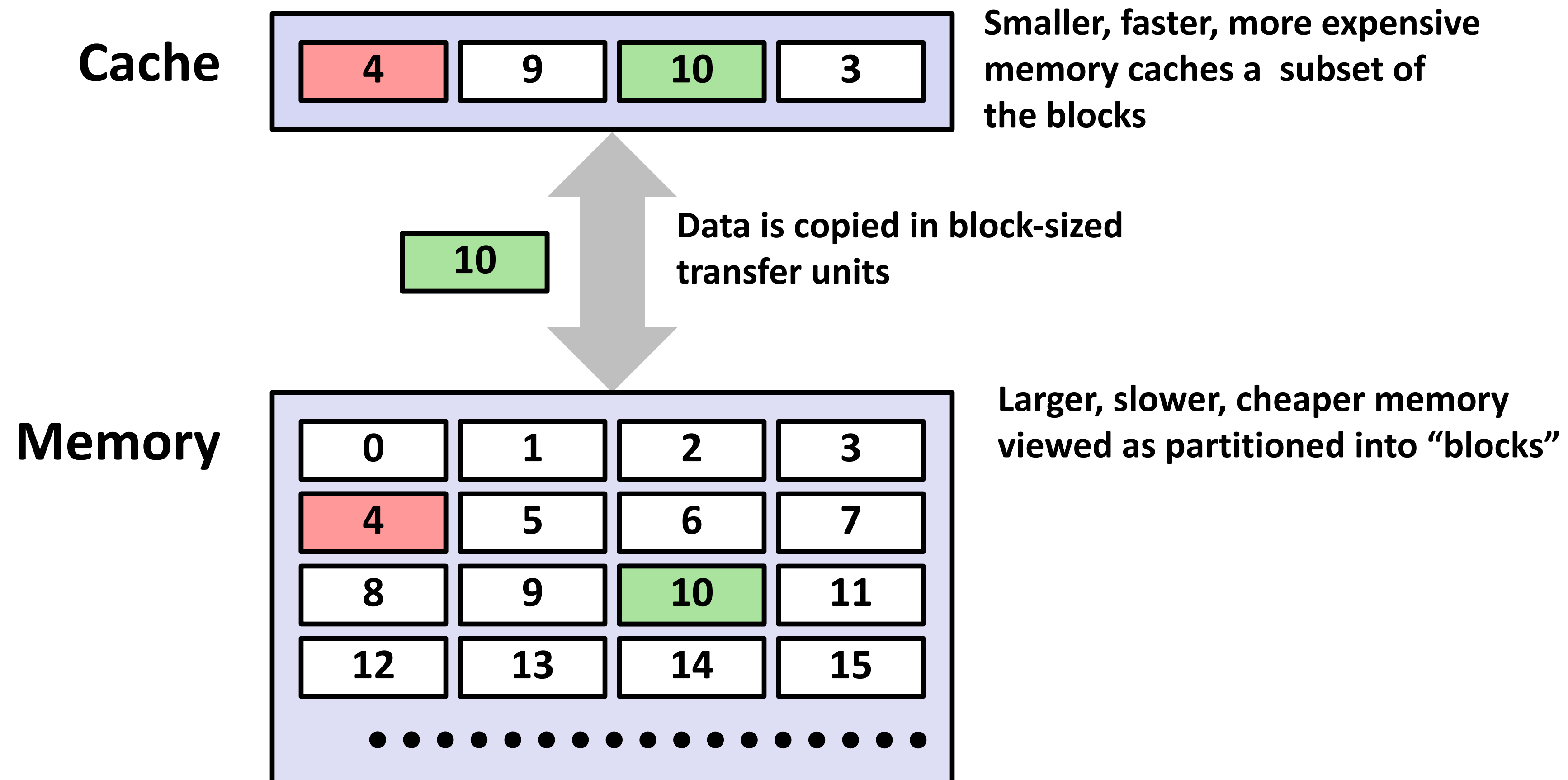
Smaller,
faster,
and
costlier
(per byte)
storage
devices

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

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

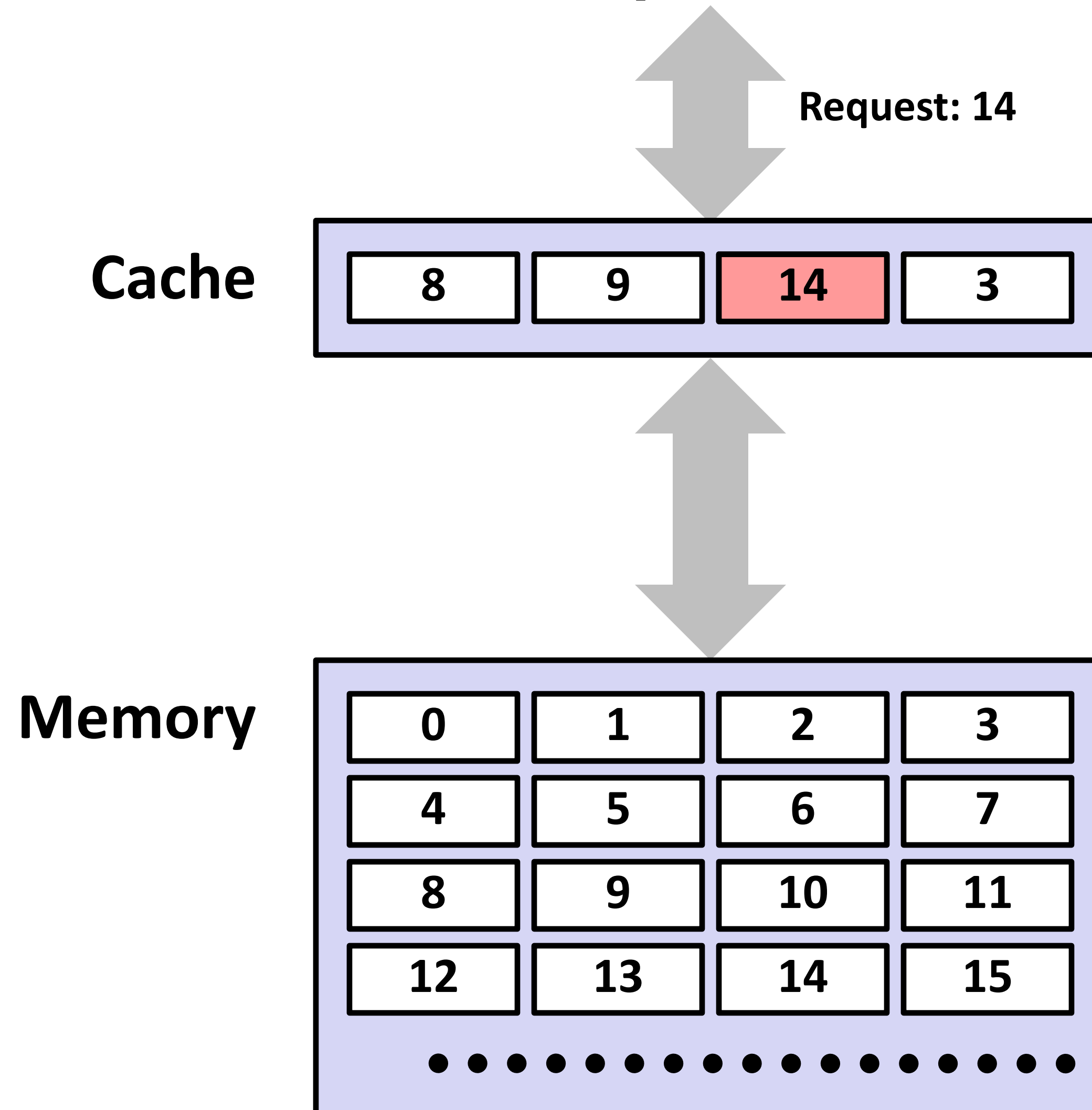Local disks hold files retrieved from disks on remote servers.

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.

General Cache Concepts

Cache

4	9	10	3

Smaller, faster, more expensive memory caches a subset of the blocks

10

Data is copied in block-sized transfer units

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Hit

Request: 14

Data in block b is needed

Cache

8 9 14 3

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Block b is in cache:
Hit!

General Cache Concepts: Miss

Request: 12

Data in block b is needed

Cache

| 8 | 12 | 14 | 3 |
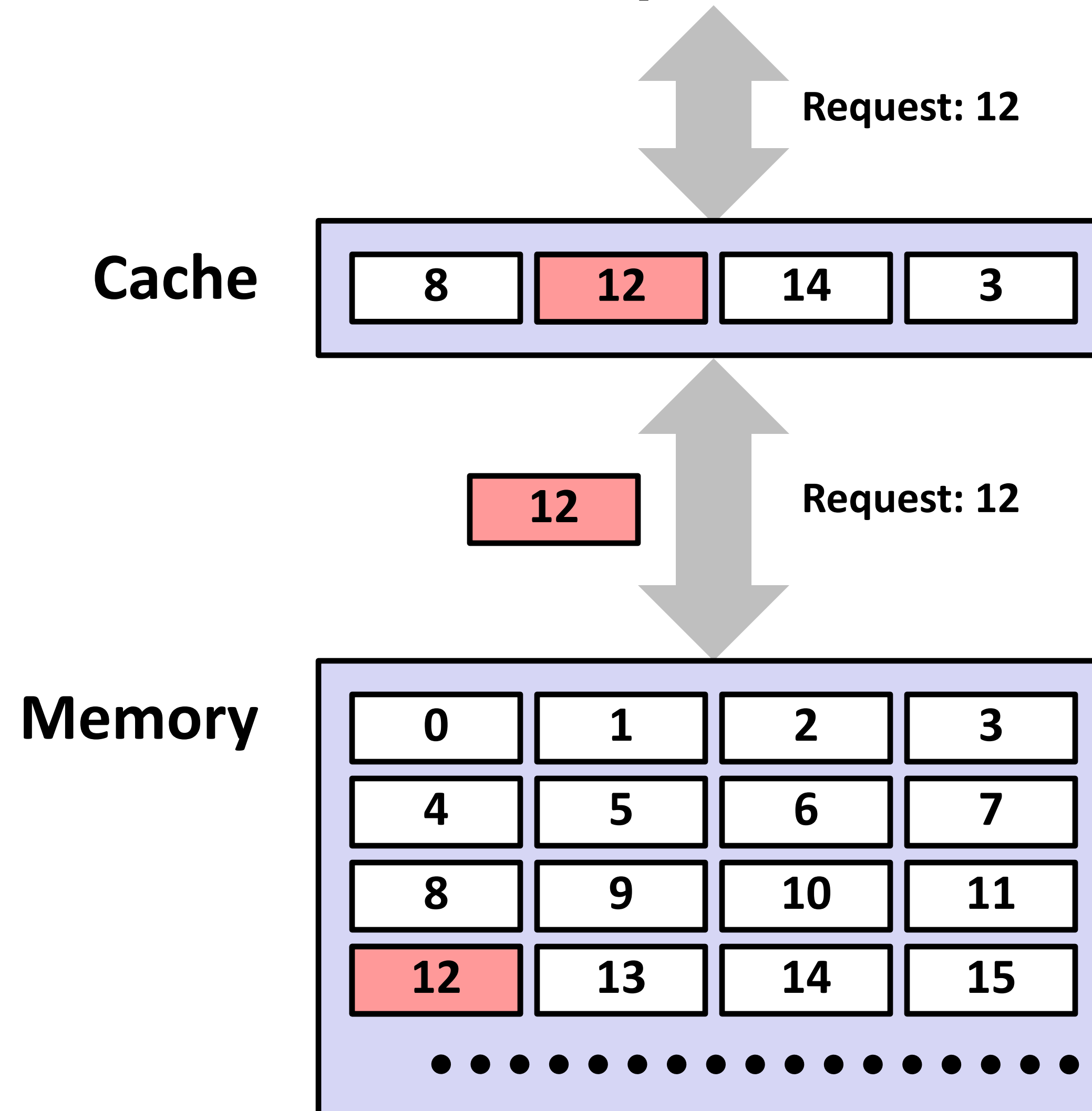
Block b is not in cache:
Miss!

| 12 |

Request: 12

Block b is fetched from memory

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Examples of Caching in the Mem. Hierarchy

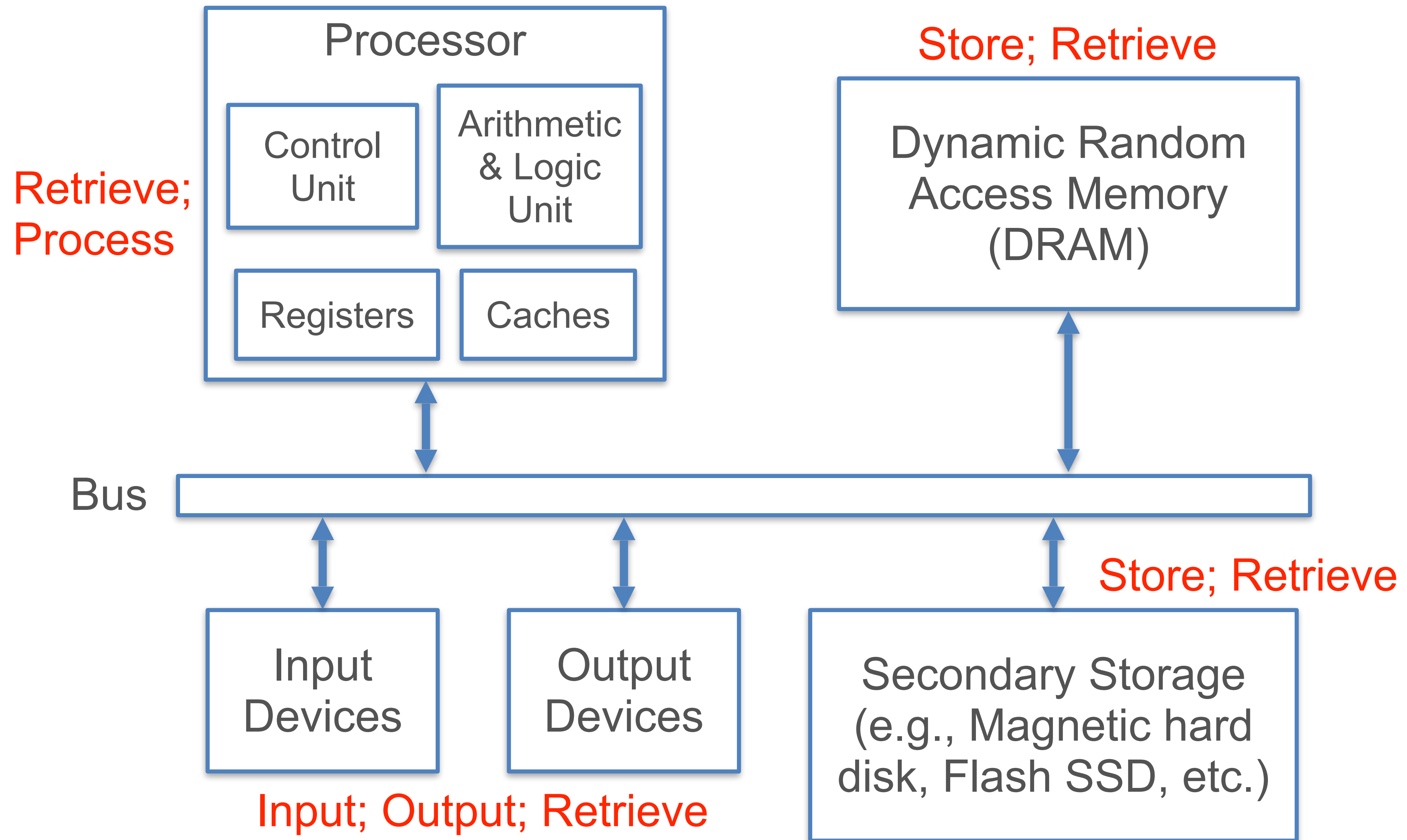| Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	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

Recap

The memory abstraction
Locality of reference
The memory hierarchy
Storage technologies and trends

Abstract Computer Parts and Data

Processor

Control Unit

Arithmetic & Logic Unit

Registers

Caches

Retrieve; Process

Store; Retrieve

Dynamic Random Access Memory (DRAM)

Bus

Input Devices

Output Devices

Secondary Storage (e.g., Magnetic hard disk, Flash SSD, etc.)

Input; Output; Retrieve

Store; Retrieve

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];
}
```

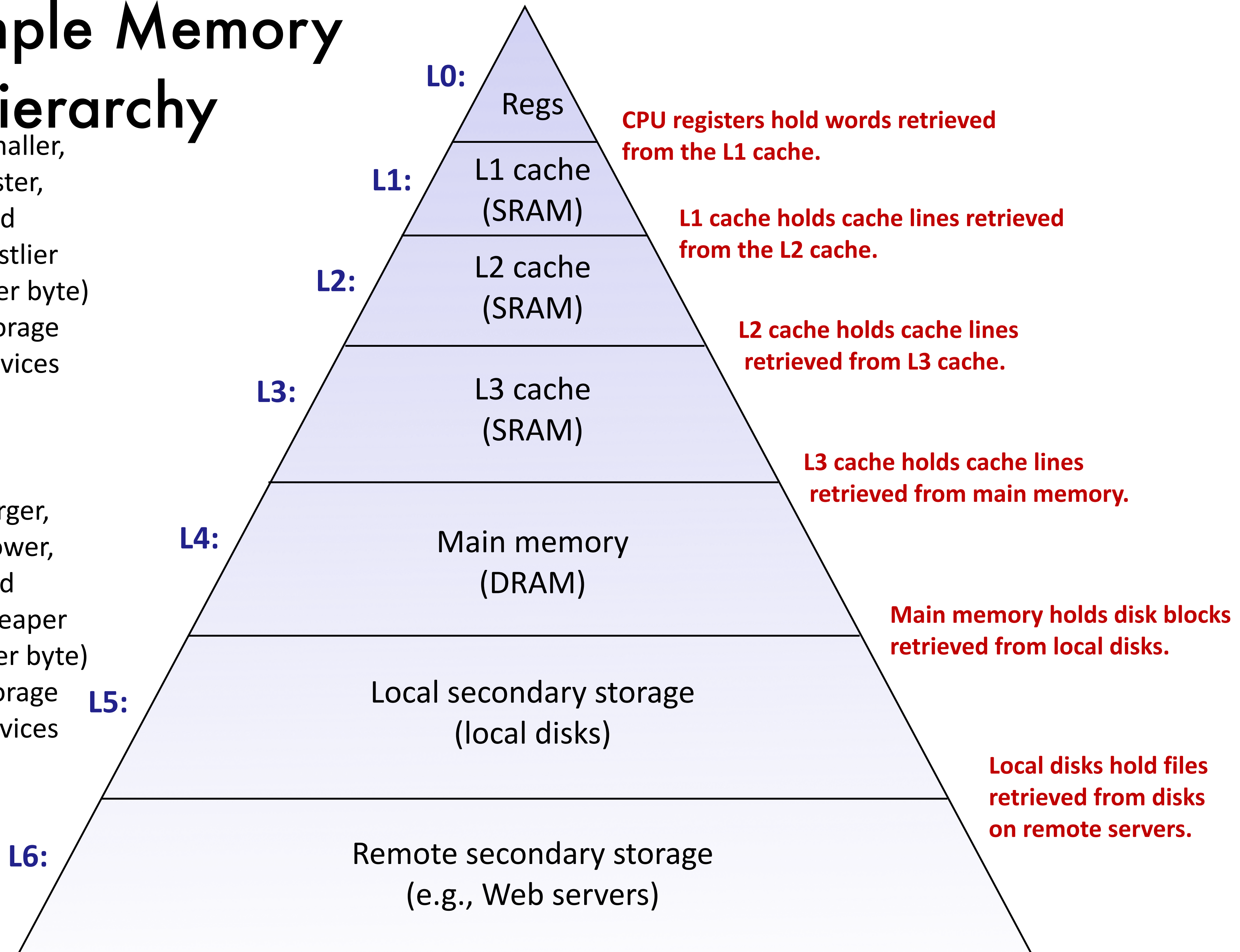**4.3 milliseconds**

```
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

Example Memory Hierarchy

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

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

L0:
Regs

L1:
L1 cache
(SRAM)

L2:
L2 cache
(SRAM)

L3:
L3 cache
(SRAM)

L4:
Main memory
(DRAM)

L5:
Local secondary storage
(local disks)

L6:
Remote secondary storage
(e.g., Web servers)

CPU registers hold words retrieved
from the L1 cache.

L1 cache holds cache lines retrieved
from the L2 cache.

L2 cache holds cache lines
retrieved from L3 cache.

L3 cache holds cache lines
retrieved from main memory.

Main memory holds disk blocks
retrieved from local disks.

Local disks hold files
retrieved from disks
on remote servers.

Today

The memory abstraction
Locality of reference
The memory hierarchy
Storage technologies and trends

What's Inside A Disk Drive?

Arm

Spindle

Platters

Actuator

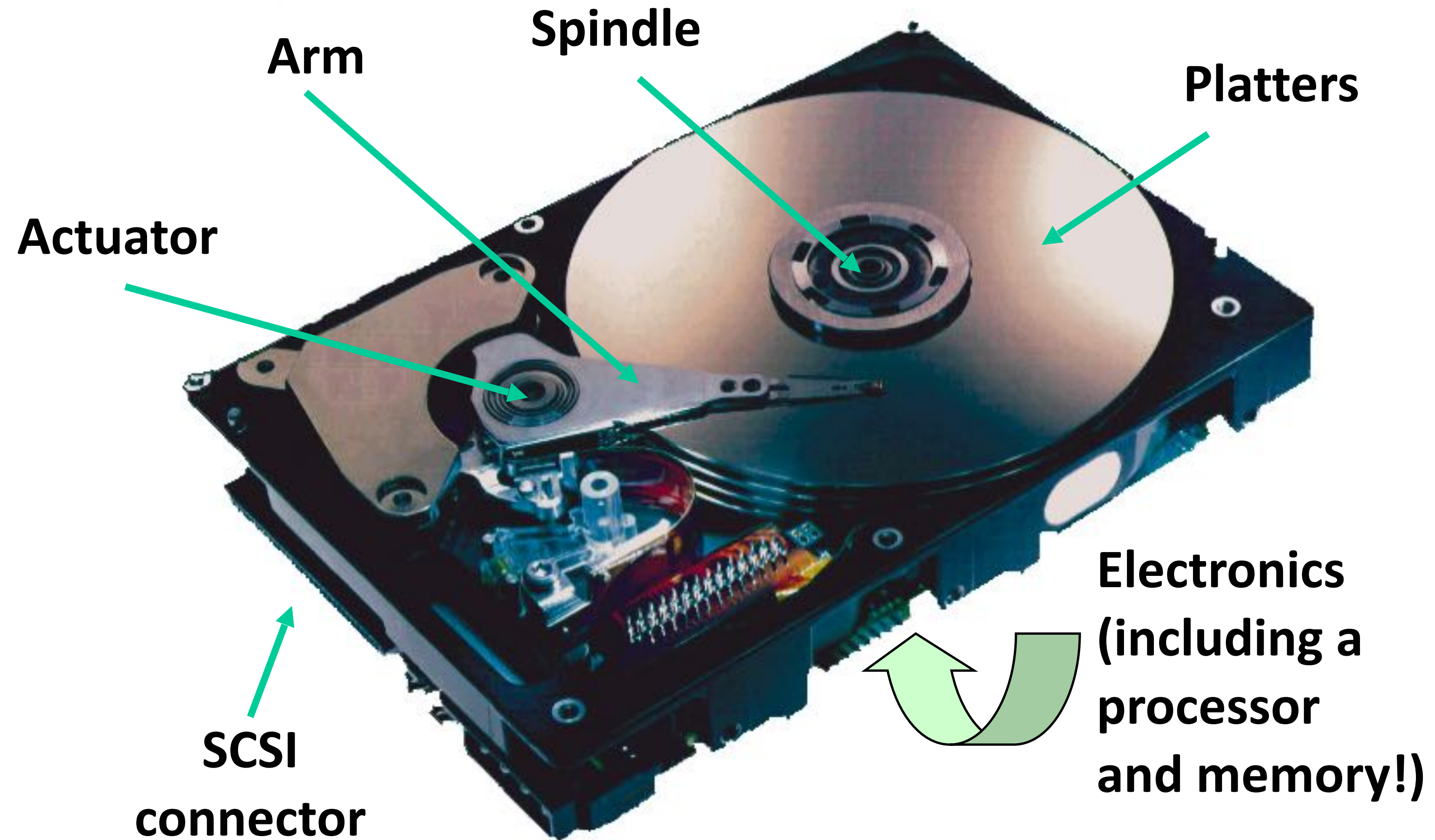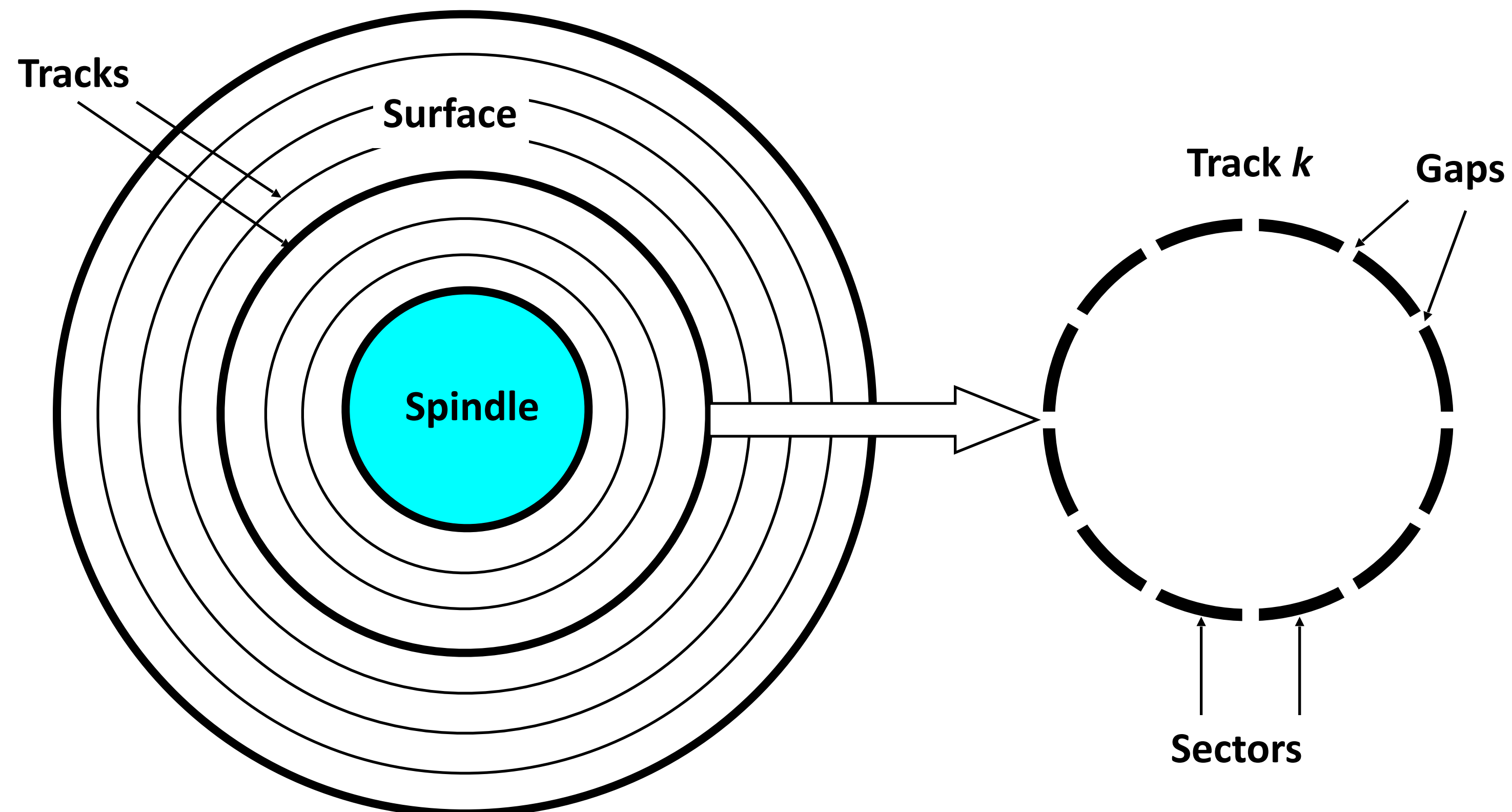**Electronics (including a processor and memory!)**

SCSI connector

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:

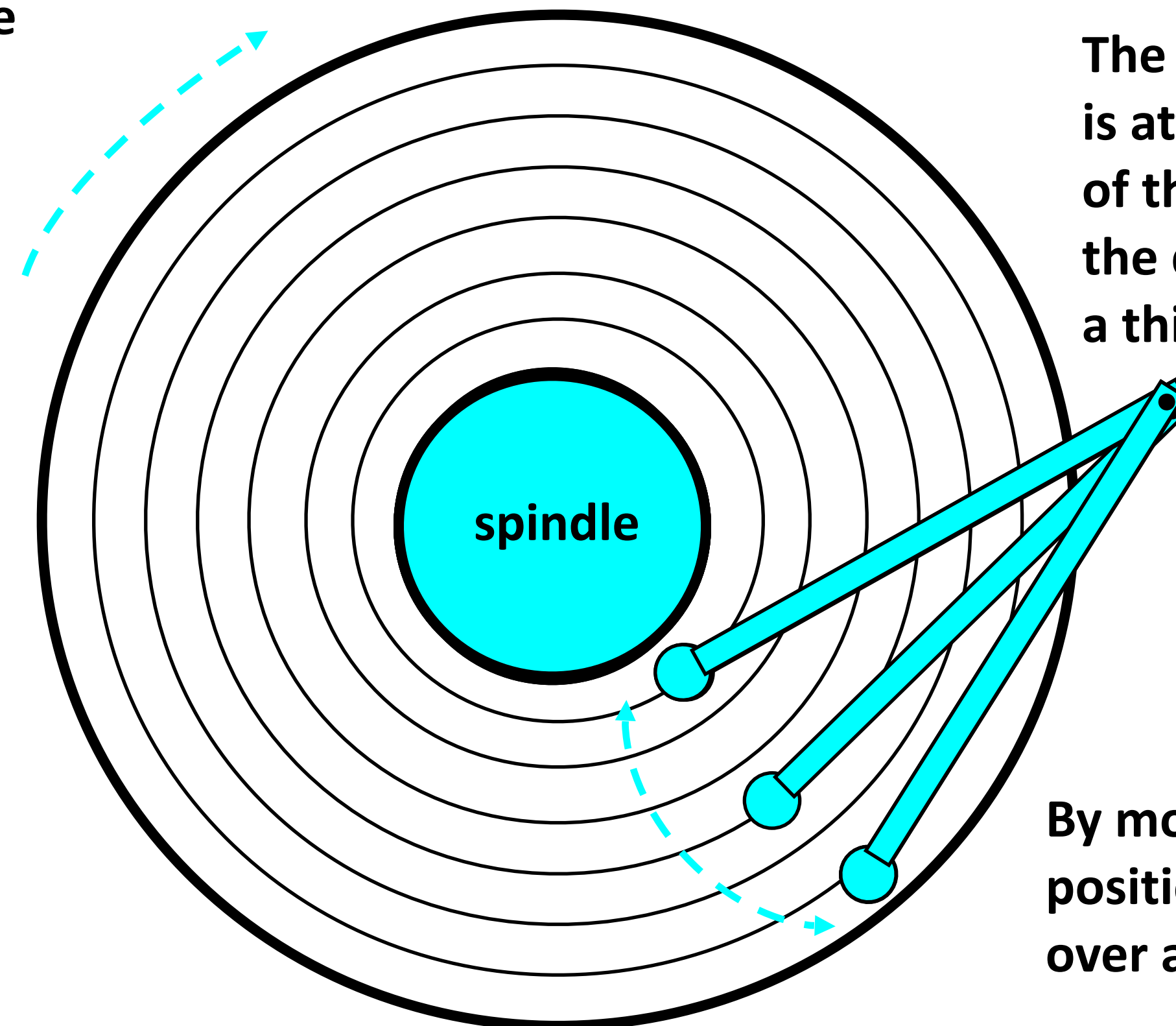Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.

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

Areal density (bits/in^2): product of recording and track density.

Tracks
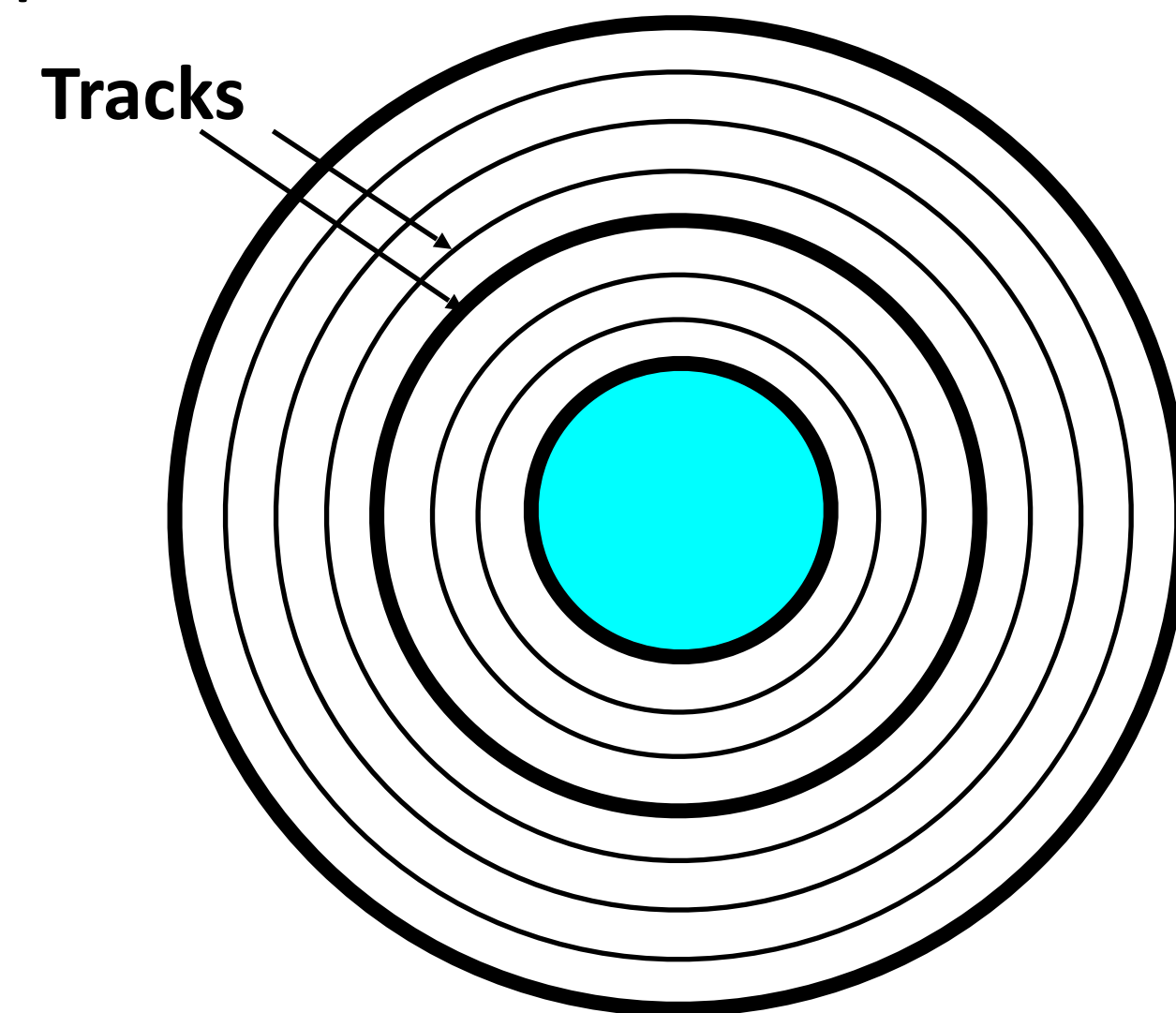
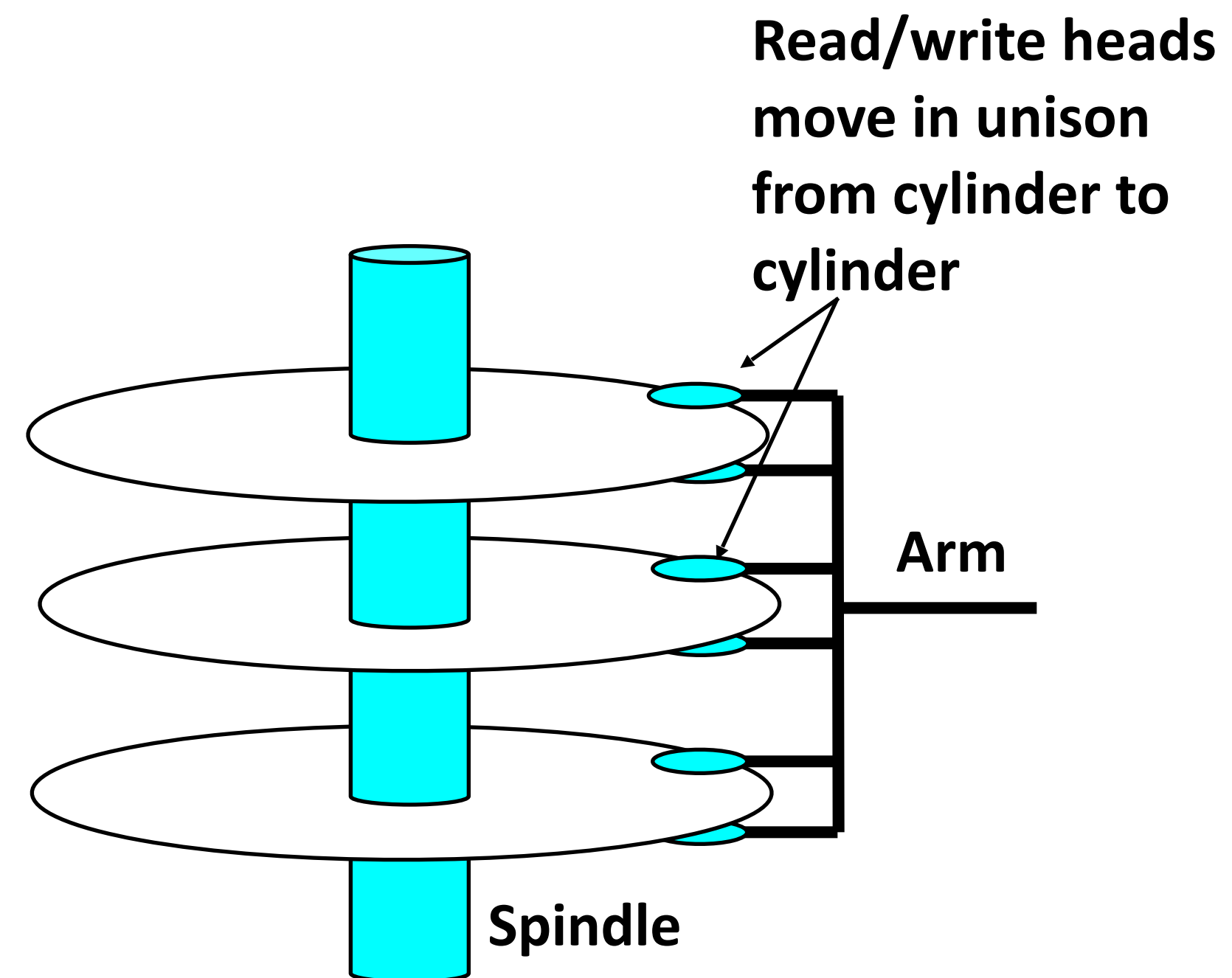# Disk Operation (Single-Platter View)

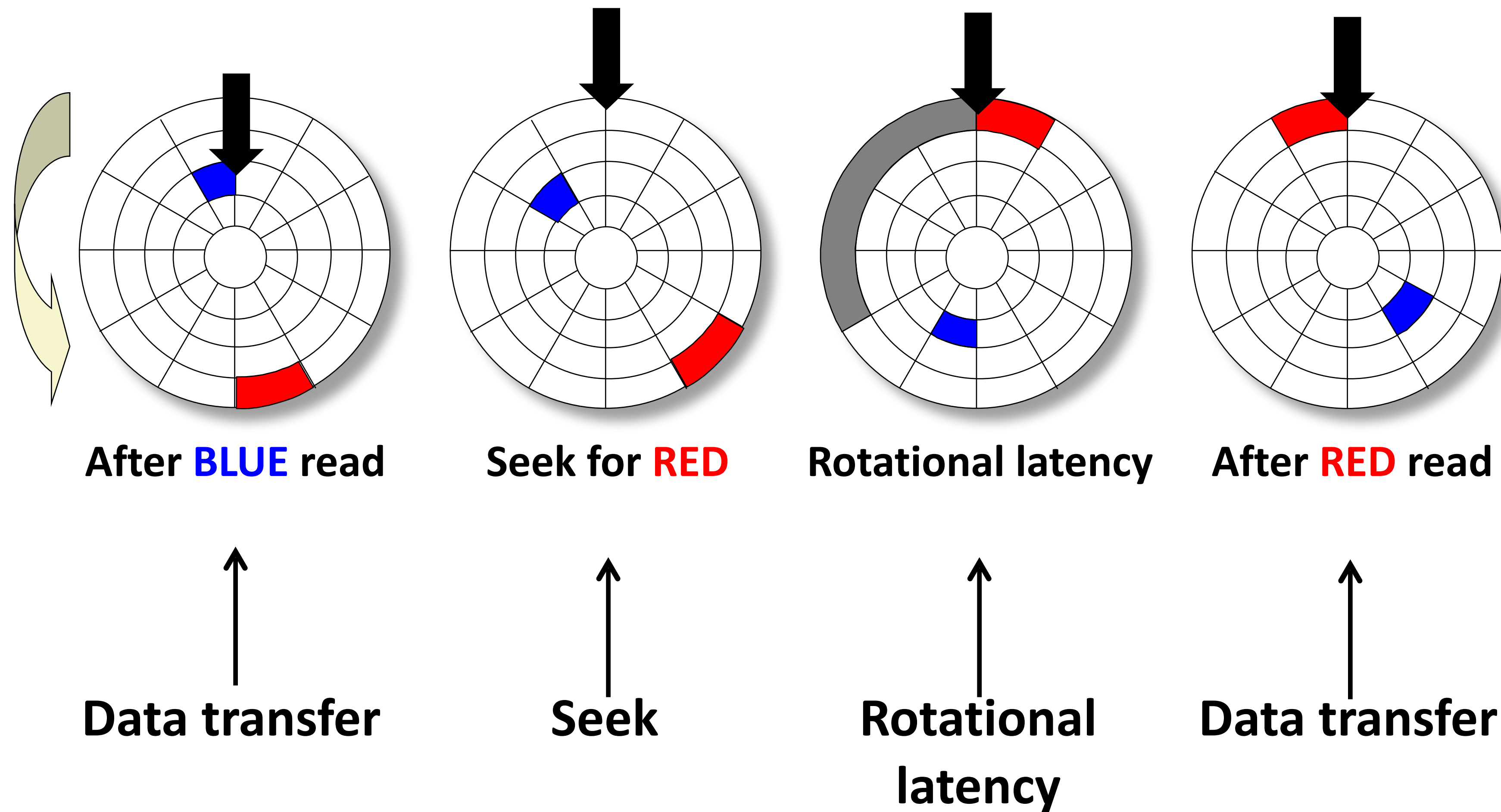**The disk surface spins at a fixed rotational rate**

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

spindle

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

Disk Operation (Multi-Platter View)

Read/write heads move in unison from cylinder to cylinder

Arm

Spindle

Disk Access – Service Time Components

After **BLUE** read	Seek for **RED**	Rotational latency	After **RED** read
↑	↑	↑	↑
Data transfer	**Seek**	**Rotational latency**	**Data transfer**

Disk Access Time

Average time to access some target sector approximated by:

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

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

Time to position heads over cylinder containing target sector.

Typical $T_{avg\ seek}$ is 3—9 ms

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

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

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

Typical rotational rate = 7,200 RPMs

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

Time to read the bits in the target sector.

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

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:

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

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

T_{access} = 9 ms + 4 ms + 0.02 ms

Important points:

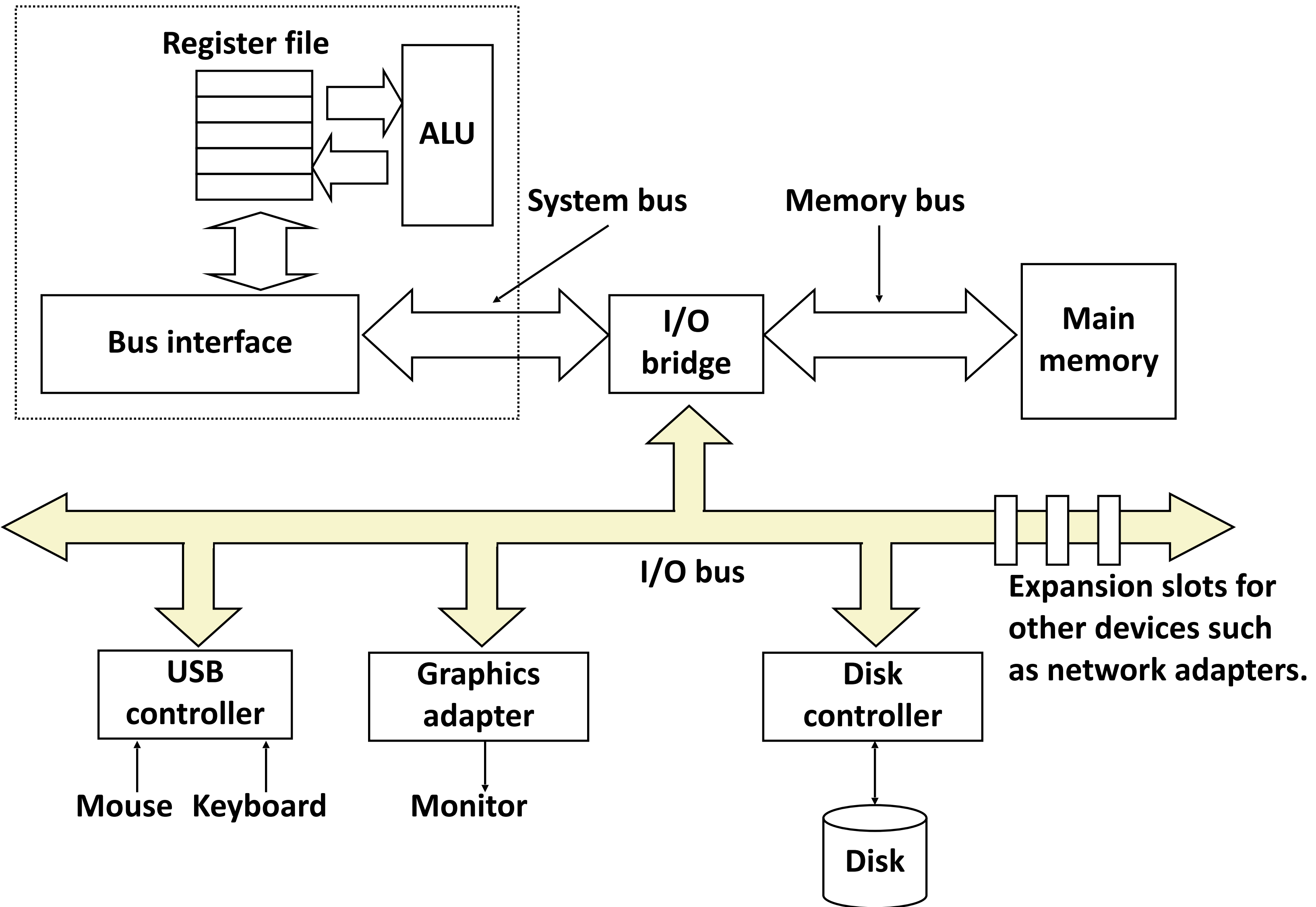Access time dominated by seek time and rotational latency.

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

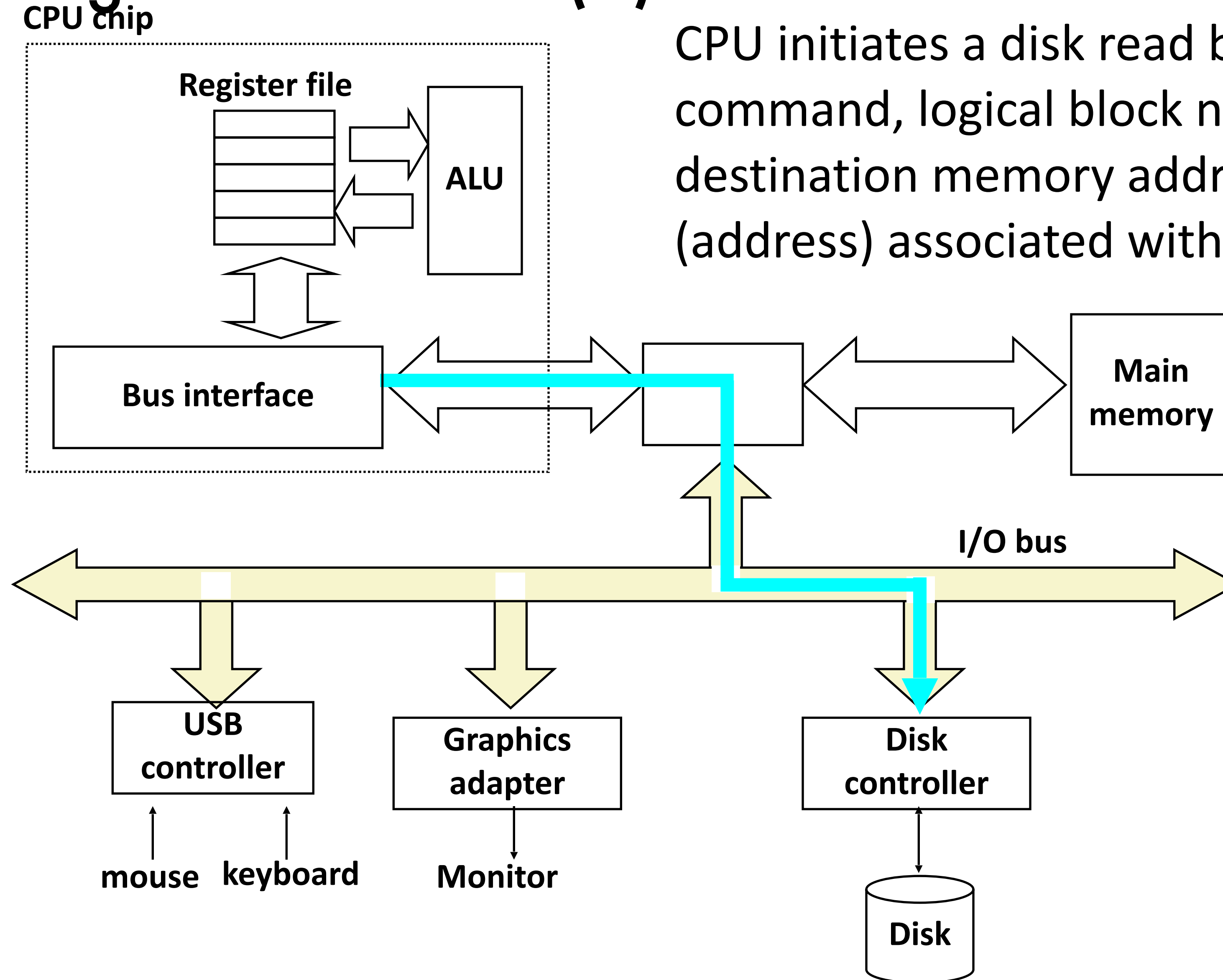*SRAM access time is about 4 ns/doubleword, DRAM about 60 ns*

Disk is about 40,000 times slower than SRAM,

2,500 times slower than DRAM.

I/O Bus

CPU chip

Register file

ALU

System bus

Memory bus

Bus interface

I/O bridge

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Expansion slots for other devices such as network adapters.

Mouse **Keyboard**

Monitor

Disk

Reading a Disk Sector (1)

CPU chip

Register file

ALU

Bus interface

Main memory

I/O bus

USB controller

mouse   keyboard

Graphics adapter

Monitor

Disk controller

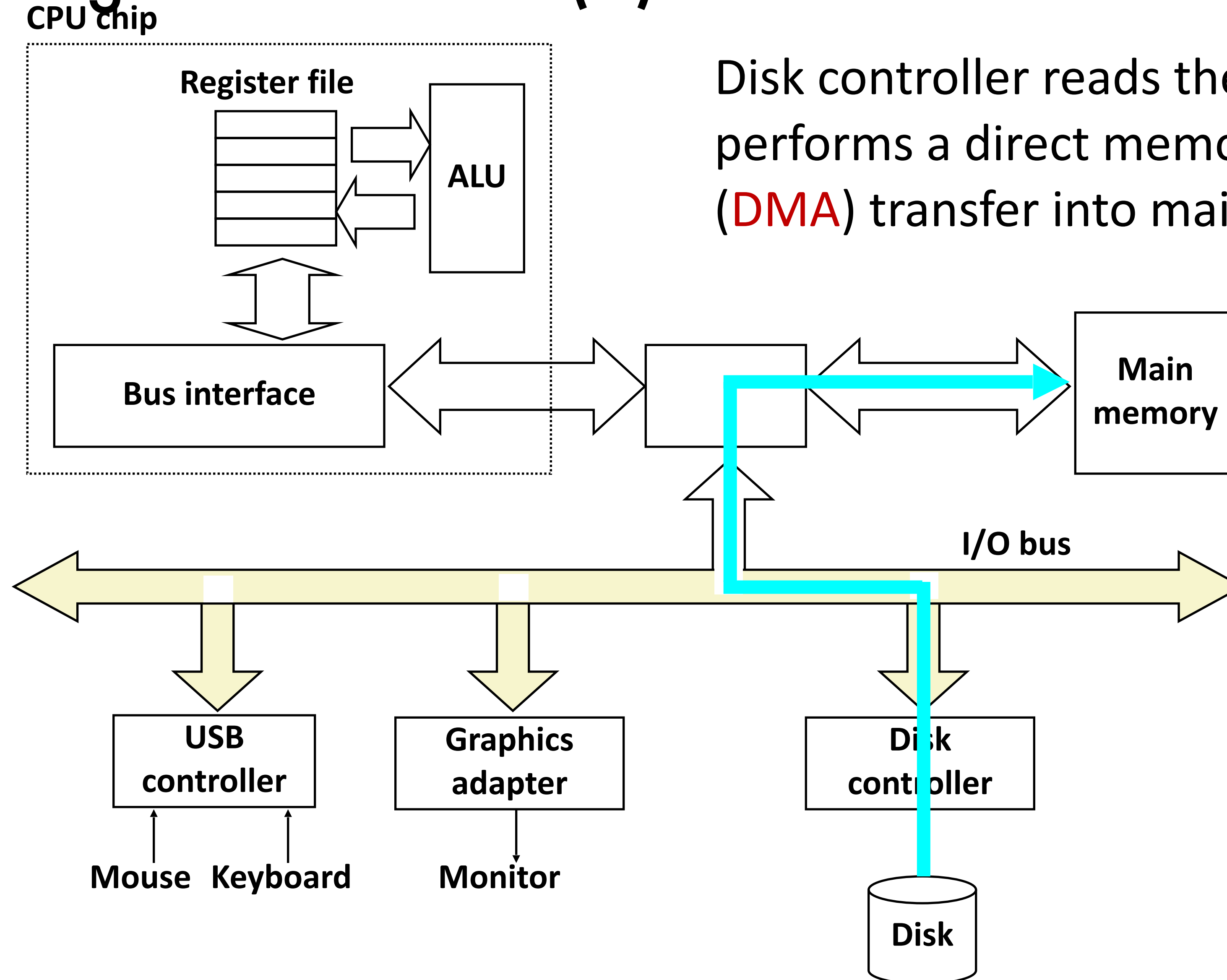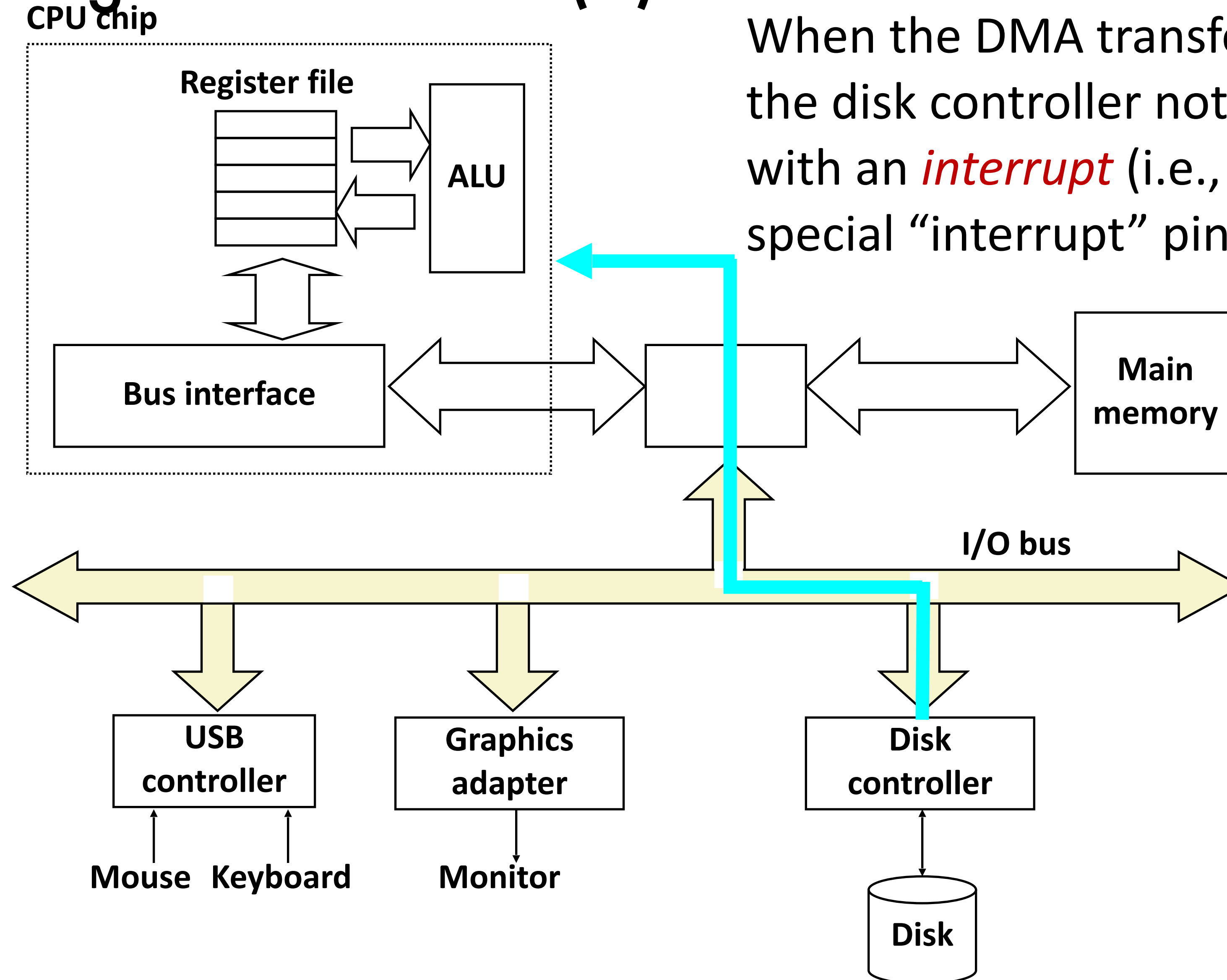**Disk**

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)

CPU chip

Register file

ALU

Bus interface

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

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Mouse **Keyboard**

Monitor

Disk

Reading a Disk Sector (3)

CPU chip

Register file

ALU

Bus interface

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Mouse   Keyboard

Monitor

Disk

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?