COMP201
Computer Systems & Programming

Lecture #21 –Cache Memories

Aykut Erdem // Koç University // Fall 2024

KOÇ UNIVERSITY

Recap

- Floating Point
- Memory Layout
- Buffer Overflow

Recap: Programming with SSE3

XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers

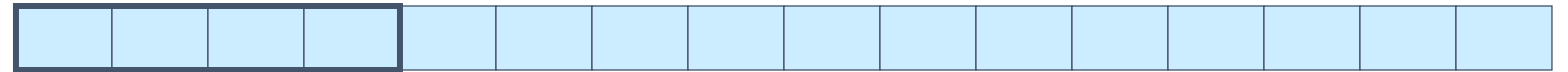- 8 16-bit integers

- 4 32-bit integers

- 4 single-precision floats

- 2 double-precision floats
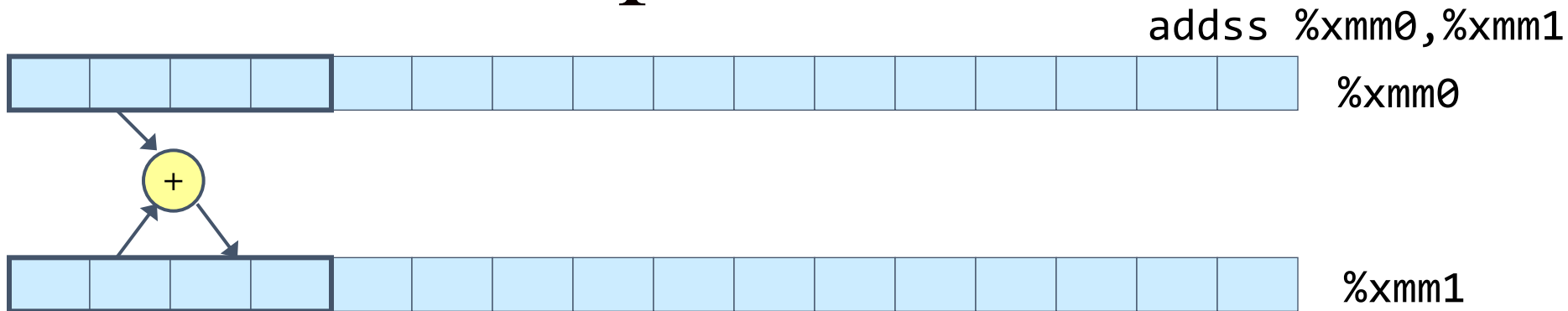
- 1 single-precision float

- 1 double-precision float

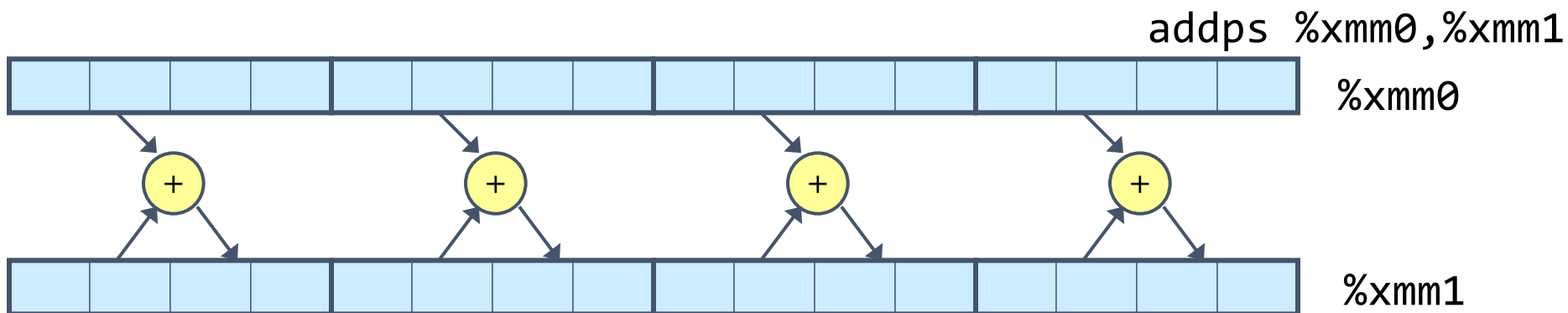# Recap: Scalar & SIMD Operations
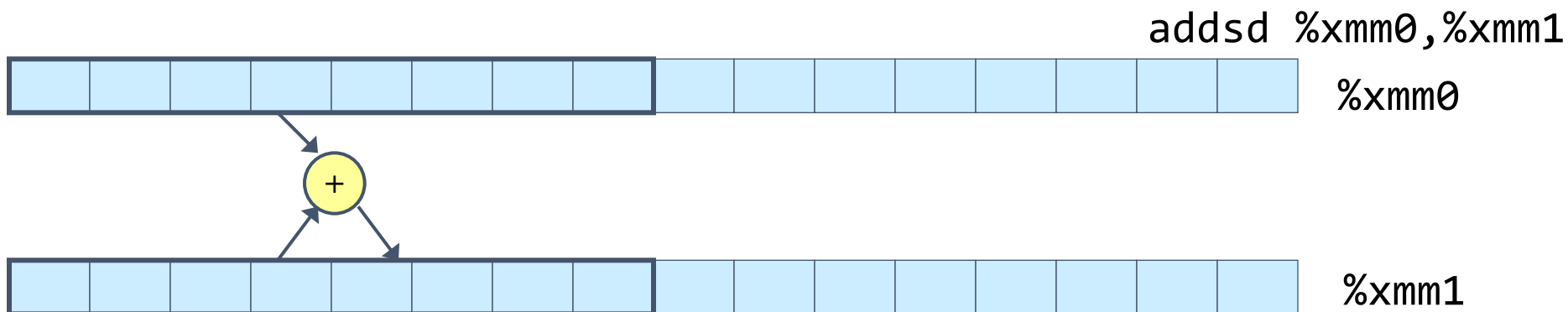
- Scalar Operations: Single Precision

`addss %xmm0,%xmm1`

%xmm0

%xmm1

- SIMD Operations: Single Precision

`addps %xmm0,%xmm1`

%xmm0

%xmm1

- Scalar Operations: Double Precision

`addsd %xmm0,%xmm1`

%xmm0

%xmm1

4

Recap: FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers

- FP values passed in XMM registers

- Different `mov` instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1    # Copy v
movsd   (%rdi), %xmm0  # x = *p
addsd   %xmm0, %xmm1    # t = x + v
movsd   %xmm1, (%rdi)  # *p = t
ret
```

Recap: x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables

- Heap
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`

- Data
  - Statically allocated data
  - E.g., global vars, static vars, string constants

- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

```
00007FFFFFFFFFFF
```

Stack

8MB

Shared Libraries

Heap

Data

Text

Hex Address → `400000`
`000000`
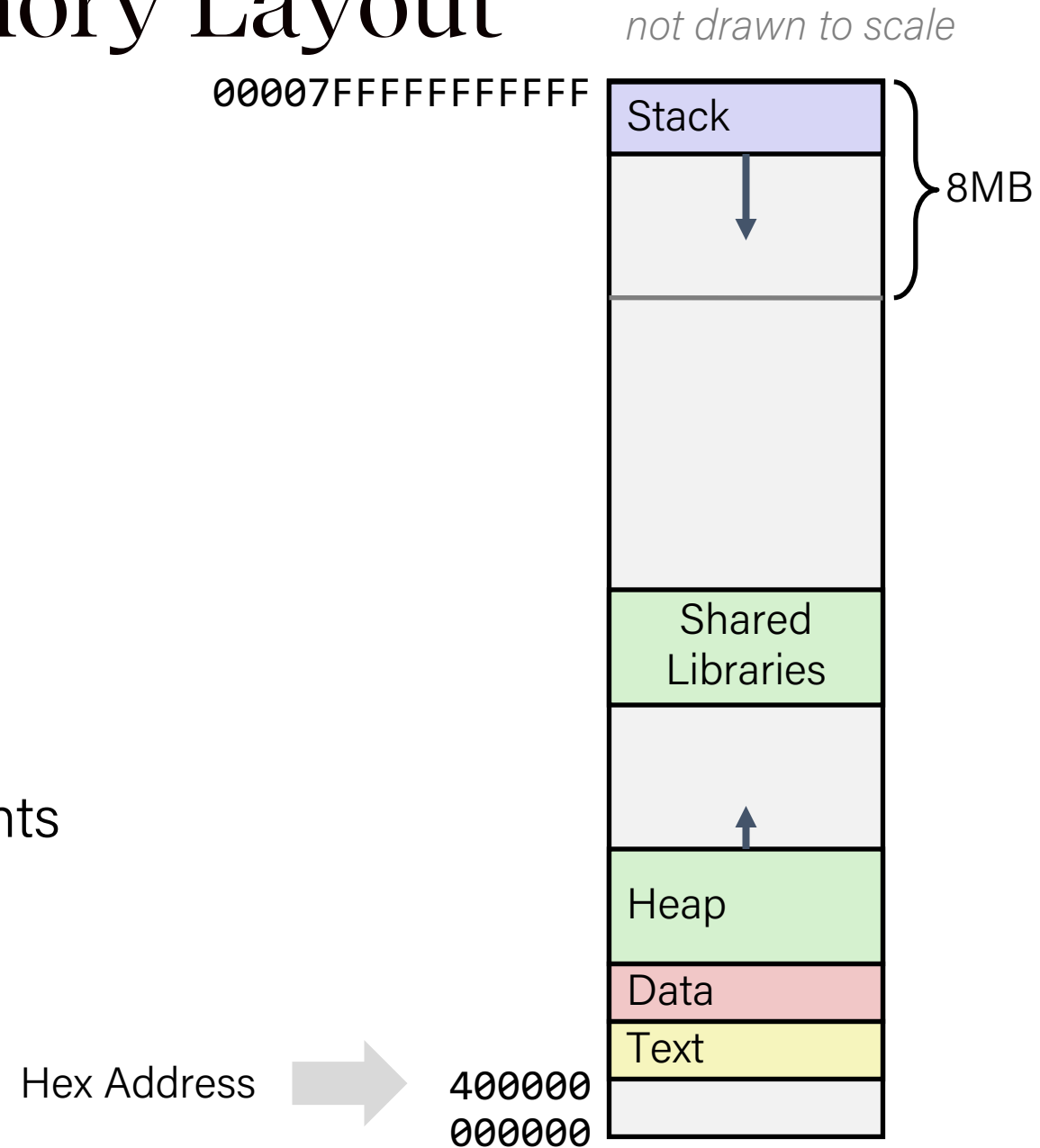
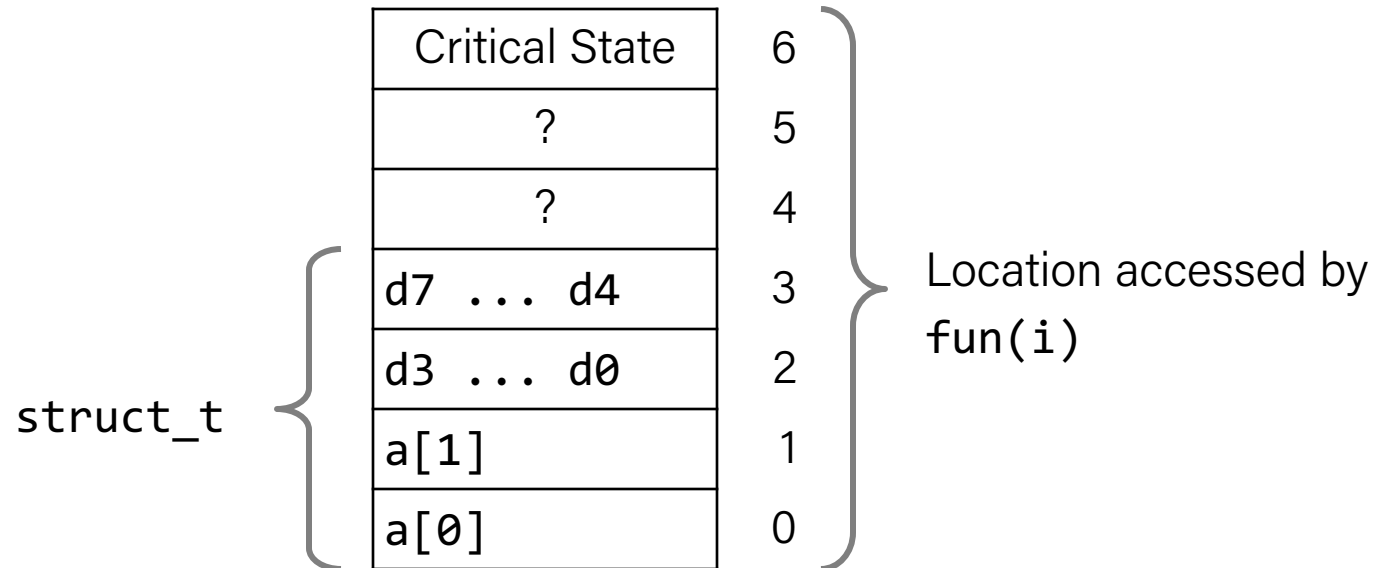# Recap: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

fun(0) → 3.14

fun(1) → 3.14

fun(2) → 3.1399998664856

fun(3) → 2.0000061035156

fun(4) → 3.14

fun(6) → Segmentation fault

Explanation:

Critical State	6
?	5
?	4
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

struct_t

Location accessed by fun(i)
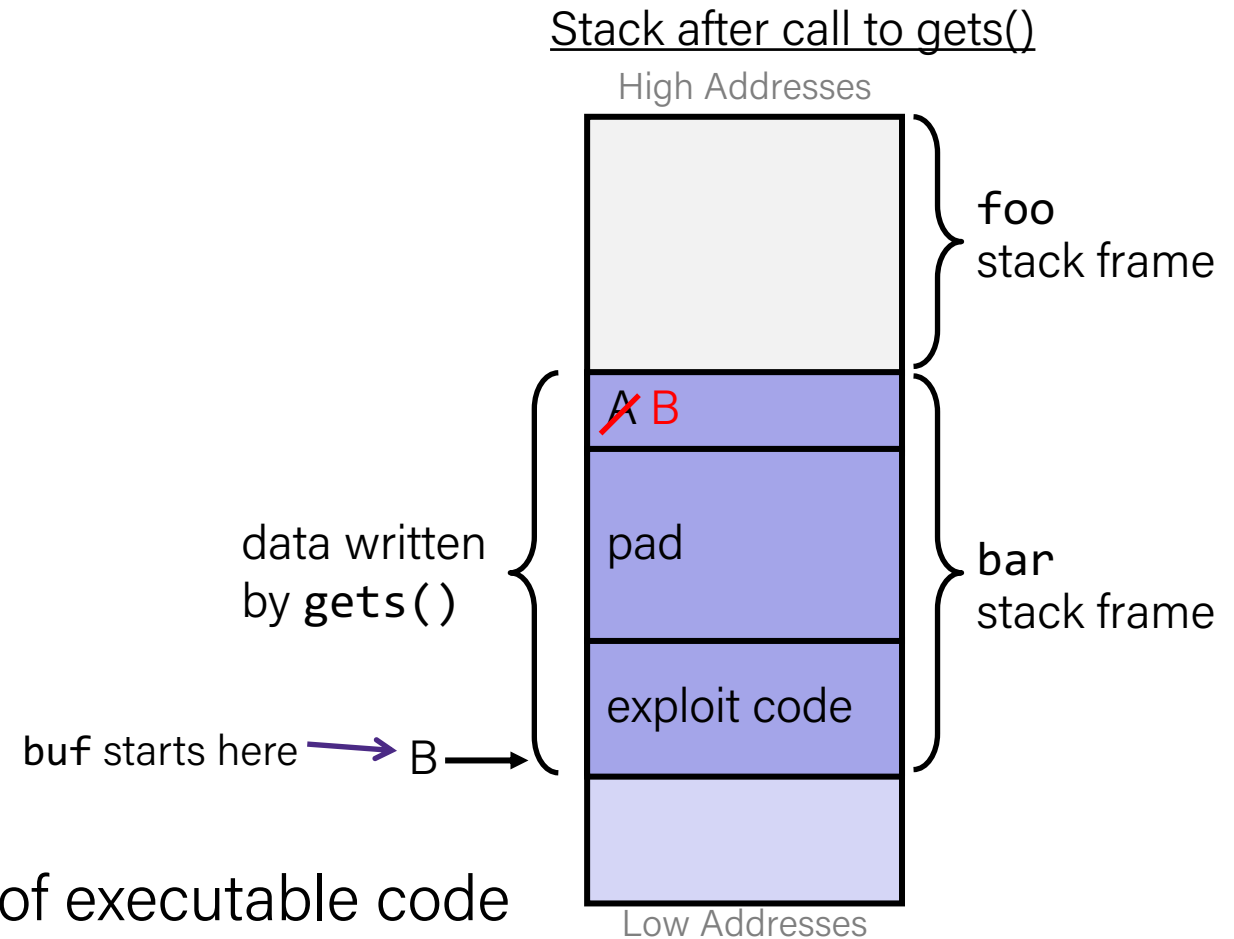
7

Recap: Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**
- Distressingly common in real programs
  - Programmers keep making the same mistakes 🙁
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- You will learn some of the tricks in Assignment 5
  - Hopefully to convince you to never leave such holes in your programs!!

Recap: Malicious Use of Buffer Overflow

```
void foo(){
  bar();
A:...
}
```

← return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack after call to gets()

High Addresses

foo stack frame

A̶ B

data written by gets()

pad

bar stack frame

exploit code

buf starts here → B

Low Addresses

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code

9

COMP201 Topic 7: How does the memory system is organized as a hierarchy of different storage devices with unique capacities?

Plan for Today

- The memory abstraction

- Storage technologies and trends

- Locality of reference

- The memory hierarchy

- Cache basics

- Cache organization

Disclaimer: Slides for this lecture were borrowed from
—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class
—Porter Jones' UW CSE 351 class

Lecture Plan

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy
- Cache basics
- Cache organization

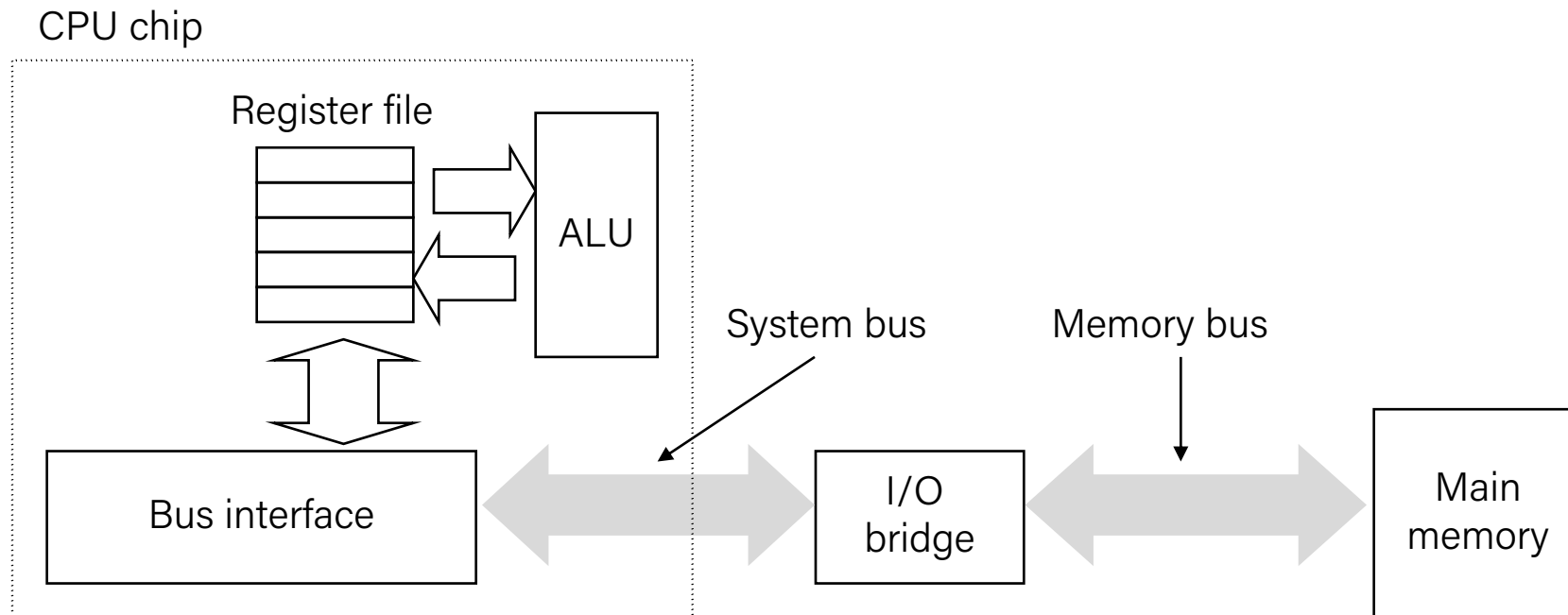# Writing & Reading Memory

- **Write**
  - Transfer data from CPU to memory
    `movq %rax, 8(%rsp)`
  - "Store" operation

- **Read**
  - Transfer data from memory to CPU
    `movq 8(%rsp), %rax`
  - "Load" operation

Traditional 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)

- CPU places address A on the memory bus.

Register file

%rax

ALU

Load operation: `movq A, %rax`

Bus interface

I/O bridge

A

Main memory

0

x

A
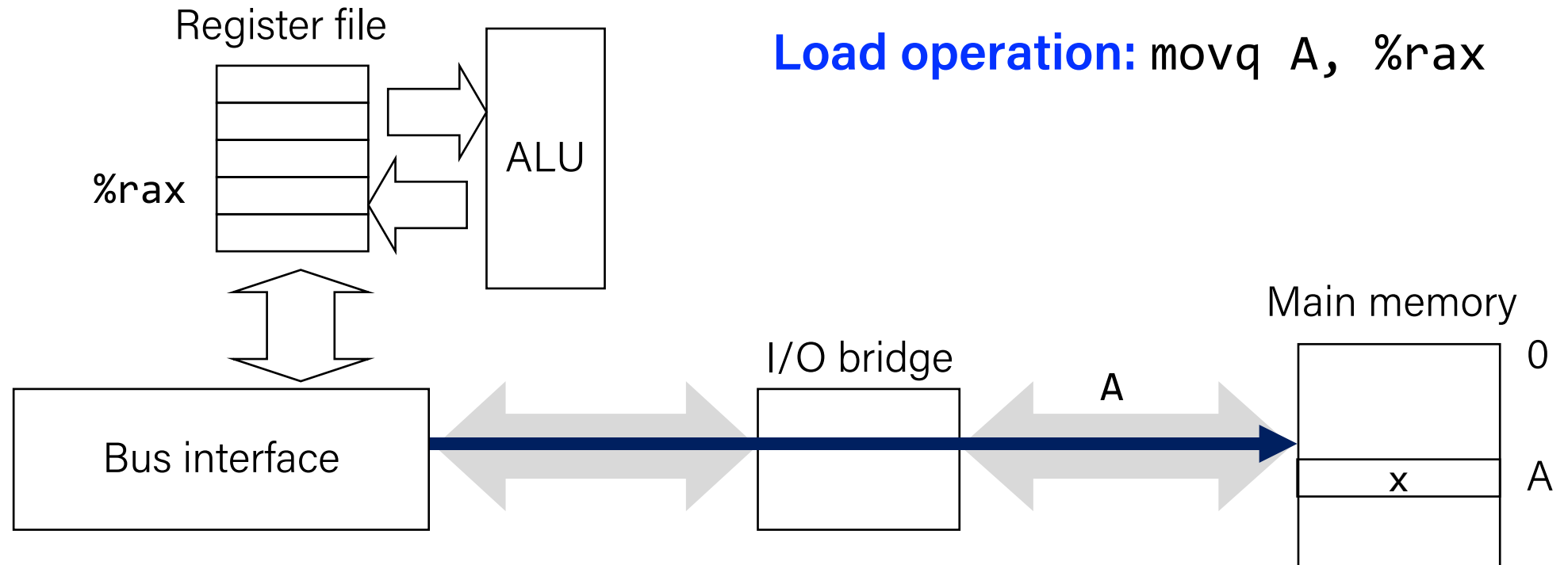
Memory Read Transaction (2)

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

Register file

%rax

ALU

Load operation: `movq A, %rax`

Bus interface

I/O bridge

x

Main memory

0

x

A
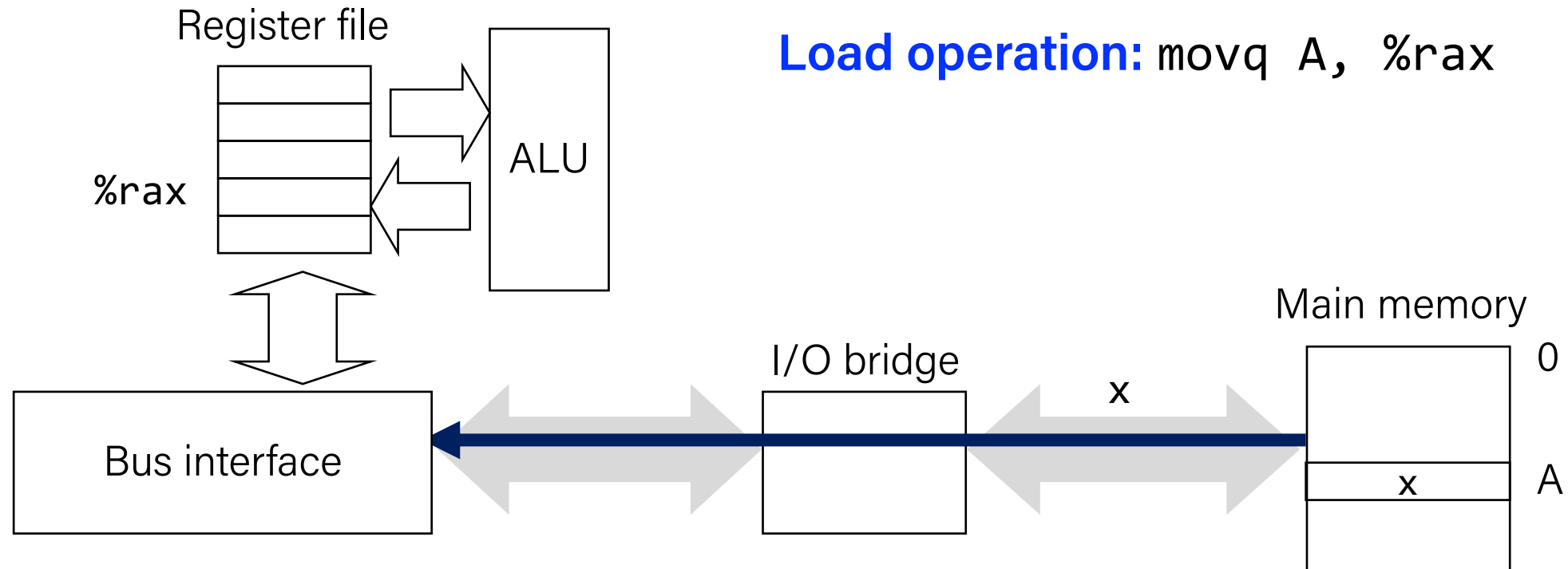
Memory Read Transaction (3)

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

Register file

Load operation: `movq A, %rax`

ALU

%rax | x

Bus interface

I/O bridge

Main memory

0

x   A
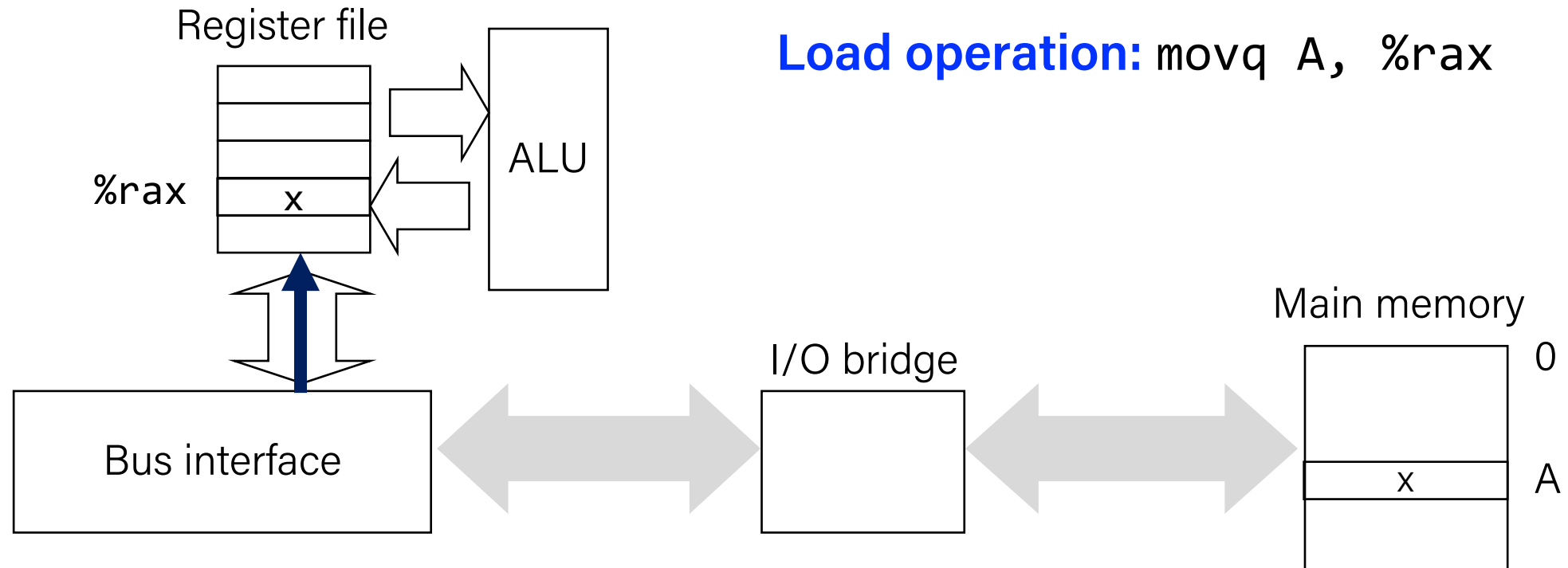
Memory Write Transaction (1)

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

Register file

%rax

y

ALU

Store operation: `movq %rax, A`

Bus interface

I/O bridge

A

Main memory

0

A
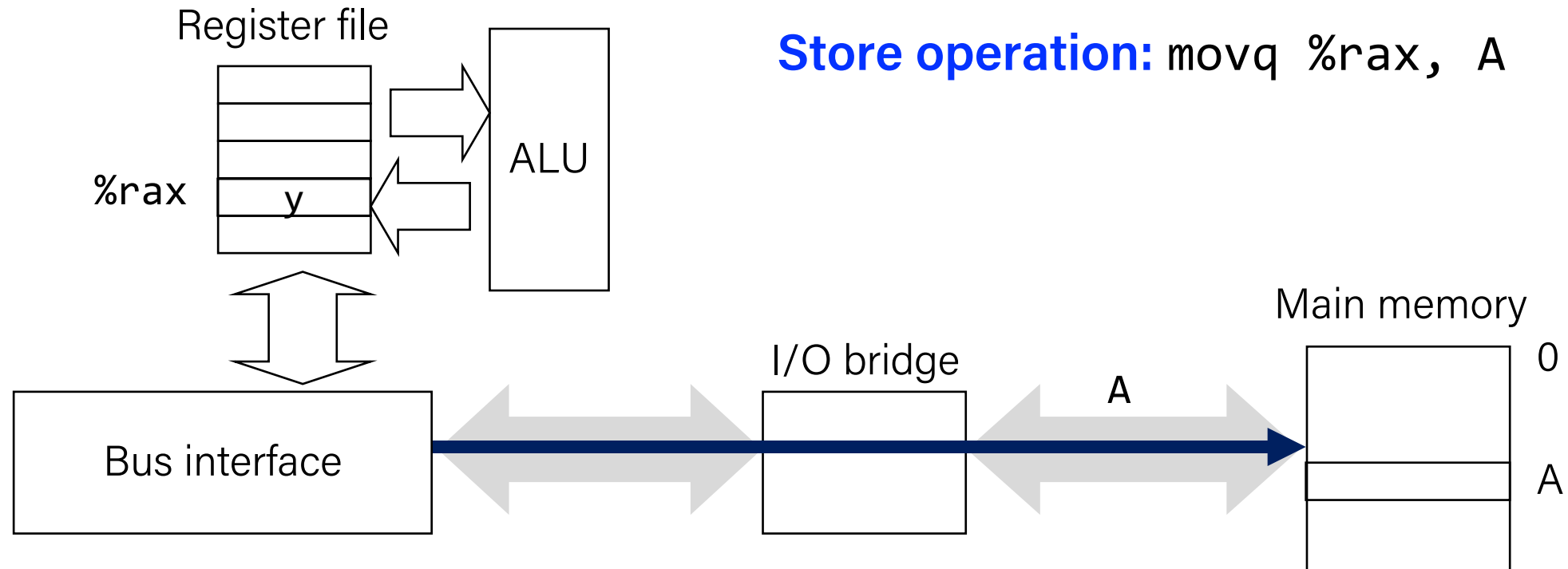
Memory Write Transaction (2)

- CPU places data word y on the bus.

Register file

Store operation: `movq %rax, A`

ALU

%rax   y

Main memory

I/O bridge

Bus interface

y

0

A
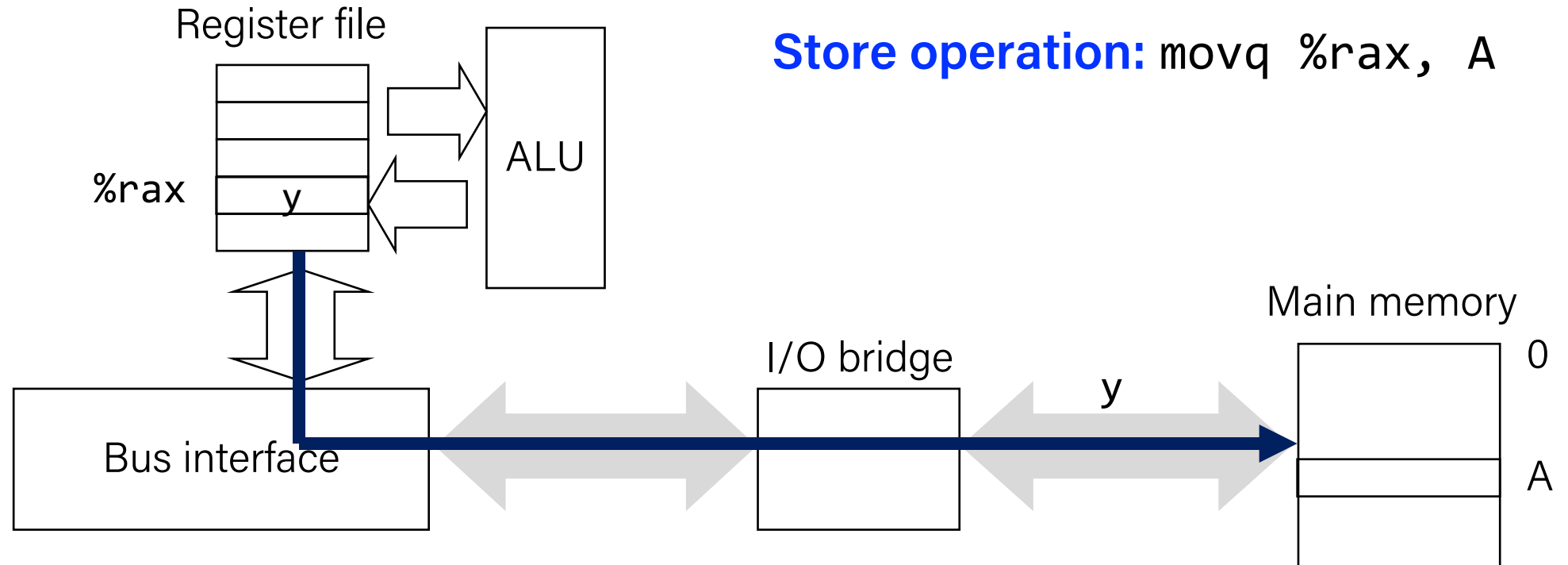
Memory Write Transaction (3)

- Main memory reads data word **y** from the bus and stores it at address **A**.
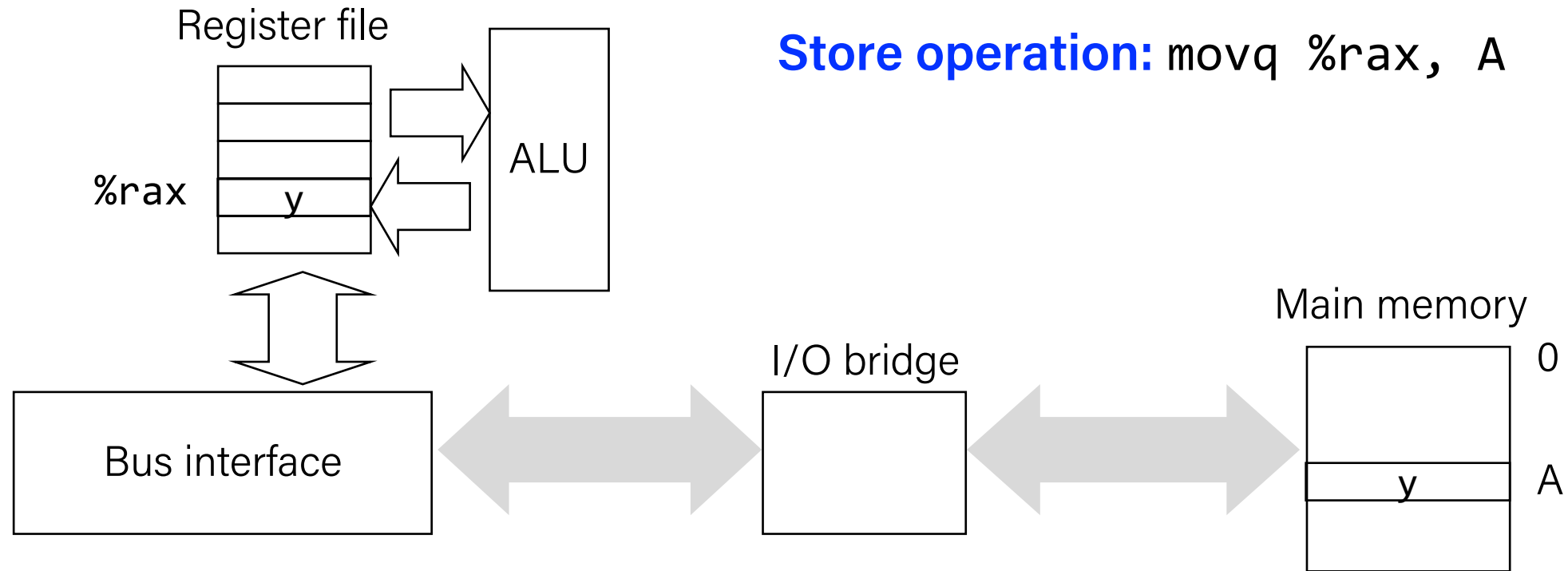
Register file

Store operation: `movq %rax, A`

%rax   y

ALU

Bus interface

I/O bridge

Main memory

0

y   A

Lecture Plan

- The memory abstraction

- **Storage technologies and trends**

- Locality of reference

- The memory hierarchy

- Cache basics

- Cache organization

Random-Access Memory (RAM)

- Key features
  - **RAM** is traditionally packaged as a chip.
  - Basic storage unit is normally a **cell** (one bit per cell).
  - Multiple RAM chips form a memory.

- RAM comes in two varieties:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)
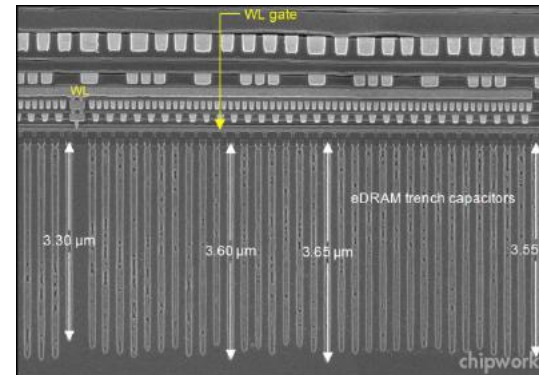
DRAM
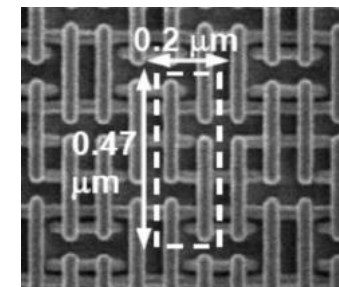
SRAM

SRAM vs DRAM

	Trans. per bit	Access time	Needs refresh?	Need EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100X	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

EDC: Error detection and correction

Trends

- SRAM scales with semiconductor technology
  - Reaching its limits
- DRAM scaling limited by need for minimum capacitance
  - Aspect ratio limits how deep can make capacitor
  - Also reaching its limits

Enhanced DRAMs

- Operation of DRAM cell has not changed since its invention
  - Commercialized by Intel in 1970.

- DRAM cores with better interface logic and faster I/O:
  - Synchronous DRAM (**SDRAM**)
    - Uses a conventional clock signal instead of asynchronous control

  - Double data-rate synchronous DRAM (**DDR SDRAM**)
    - Double edge clocking sends two bits per cycle per pin
    - Different types distinguished by size of small prefetch buffer:
      - **DDR** (2 bits), **DDR2** (4 bits), **DDR3** (8 bits), **DDR4** (16 bits)
    - By 2010, standard for most server and desktop systems
    - Intel Core i7 supports DDR3 and DDR4 SDRAM

I/O Bus
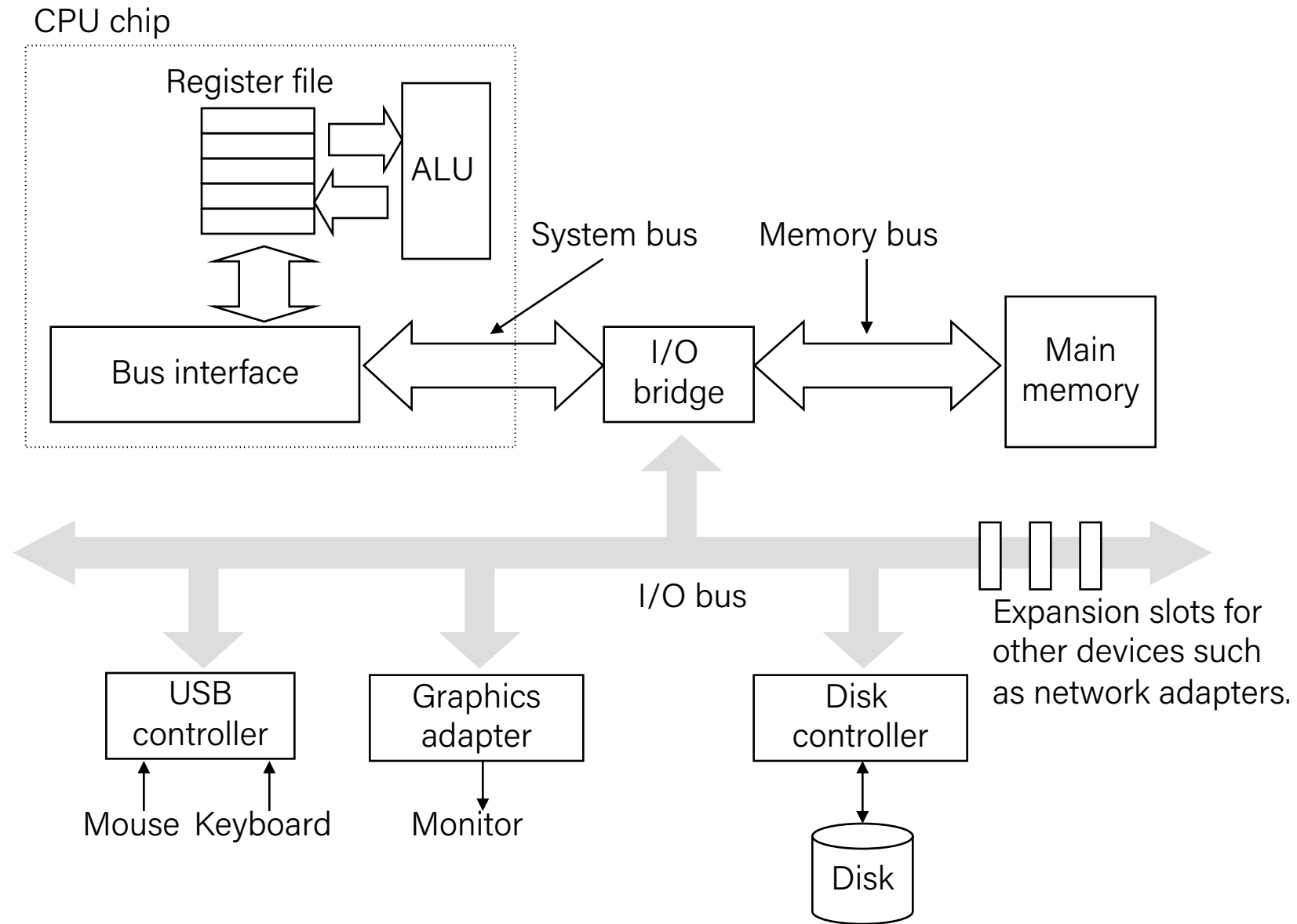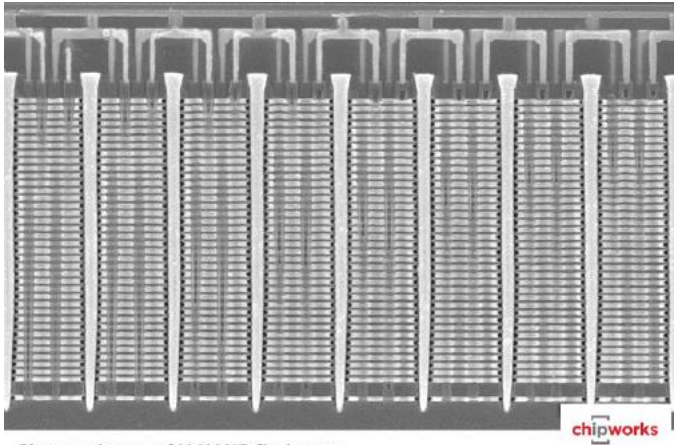
CPU chip

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

Storage Technologies

- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array
chipworks

- Magnetic Disks

- Store as persistent charge

- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Store on magnetic medium

- Electromechanical access

Storage Technologies

- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array
chipworks

- Store as persistent charge

- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Magnetic Disks

- Store on magnetic medium

- Electromechanical access

Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
  - Read-only memory (**ROM**): programmed during production
  - Electrically eraseable PROM (**EEPROM**): electronic erase capability
  - Flash memory: EEPROMs with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
  - 3D XPoint (Intel Optane) & emerging NVMs
    - New materials

- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
  - Disk caches

Storage Technologies
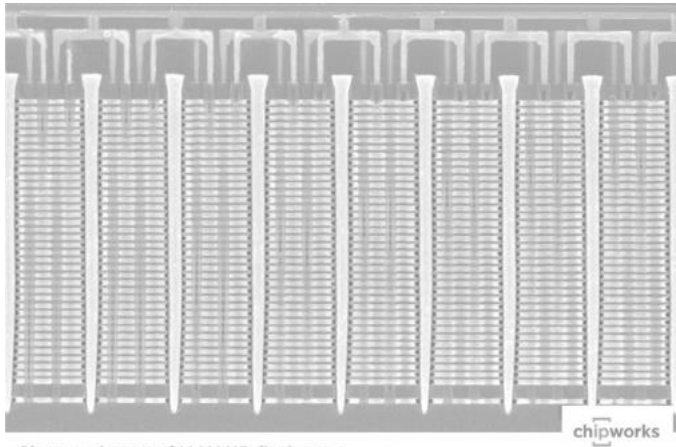
- Nonvolatile (Flash) Memory

Close-up image of V-NAND flash array

chipworks

- Store as persistent charge
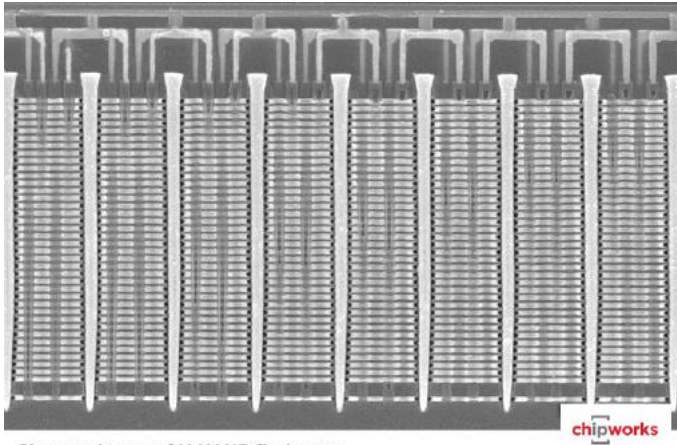- Implemented with 3-D structure
  - 100+ levels of cells
  - 3 bits data per cell

- Magnetic Disks

- Store on magnetic medium
- Electromechanical access

What's Inside A Magnetic Disk Drive?

Arm

Spindle

Platters

Actuator

Electronics (including a processor and memory!)

SCSI connector

Image courtesy of Seagate Technology
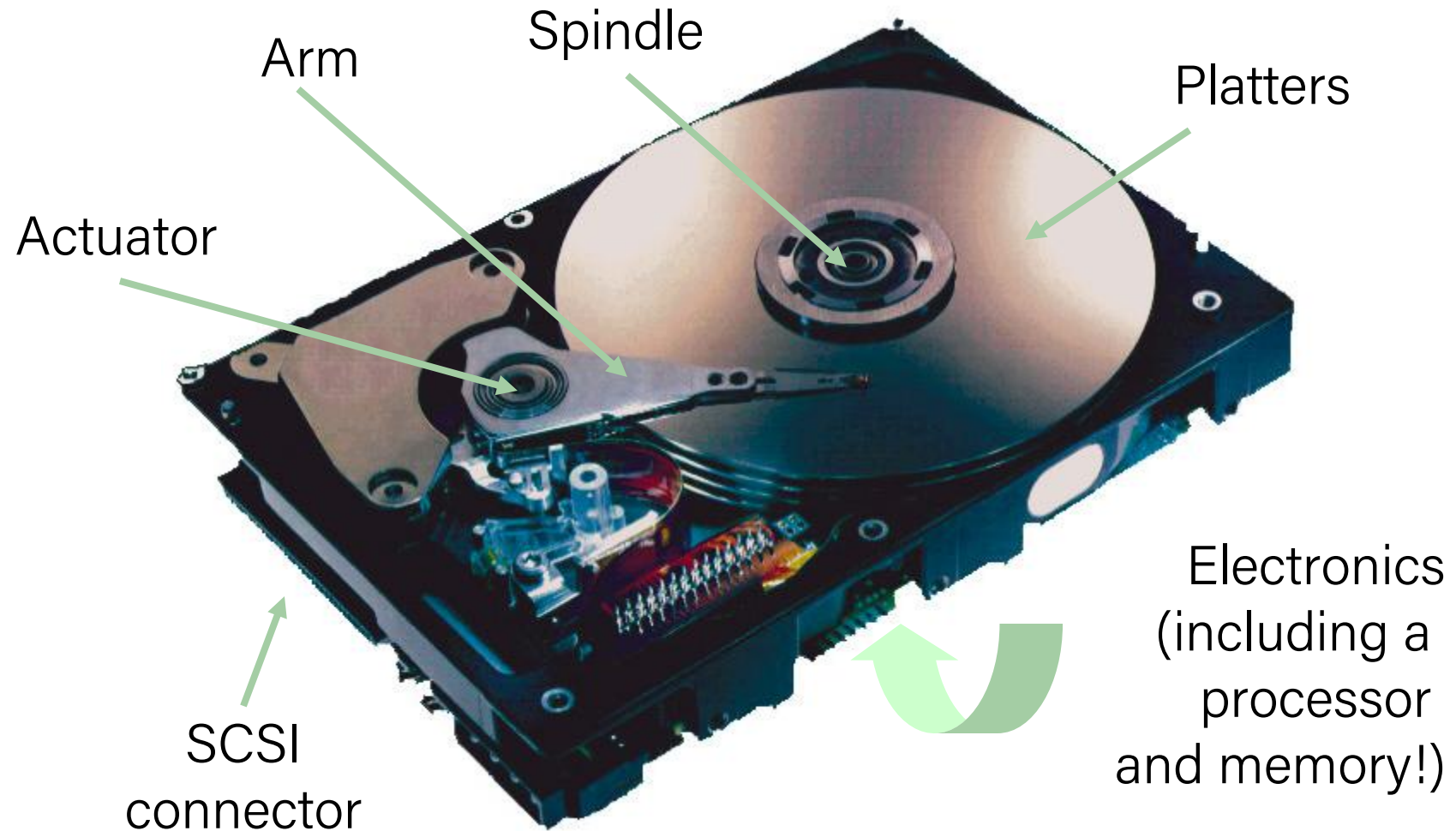
Disk Access Time

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.

- Derived:
  - Tavg rotation = 1/2 × (60 secs/7200 RPM) × 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM × 1/400 secs/track × 1000 ms/sec = 0.02 ms
  - Taccess = 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.

Solid State Disks (SSDs)
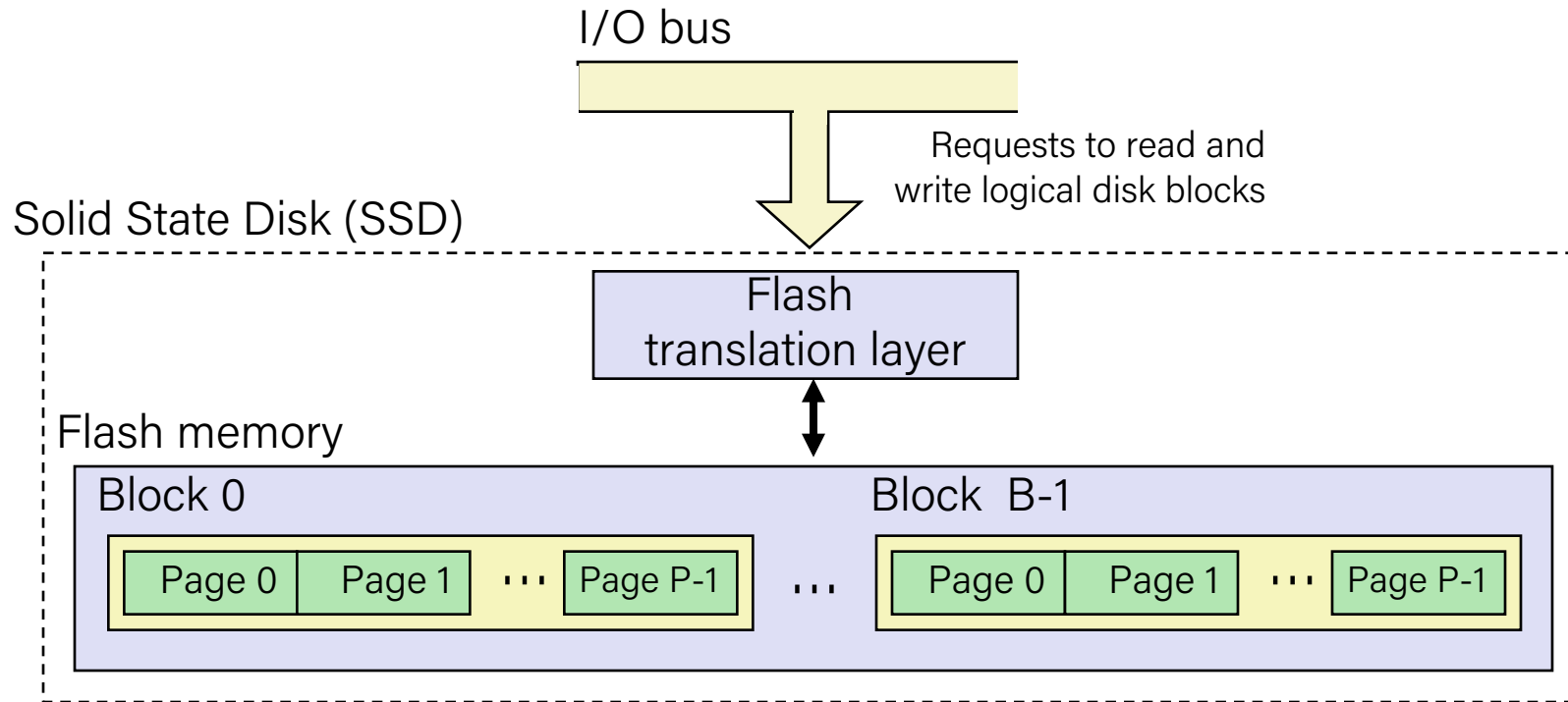
- Pages: 512KB to 4KB, Blocks: 32 to 128 pages

- Data read/written in units of pages.

- Page can be written only after its block has been erased

- A block wears out after about 100,000 repeated writes.

SSD Tradeoffs 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
    - E.g. Samsung 940 EVO Plus guarantees 600 writes/byte of writes before they wear out
    - Controller migrates data to minimize wear level
  - In 2019, about 4 times more expensive per byte
    - And, relative cost will keep dropping

- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

The CPU-Memory Gap

- The gap widens between DRAM, disk, and CPU speeds.
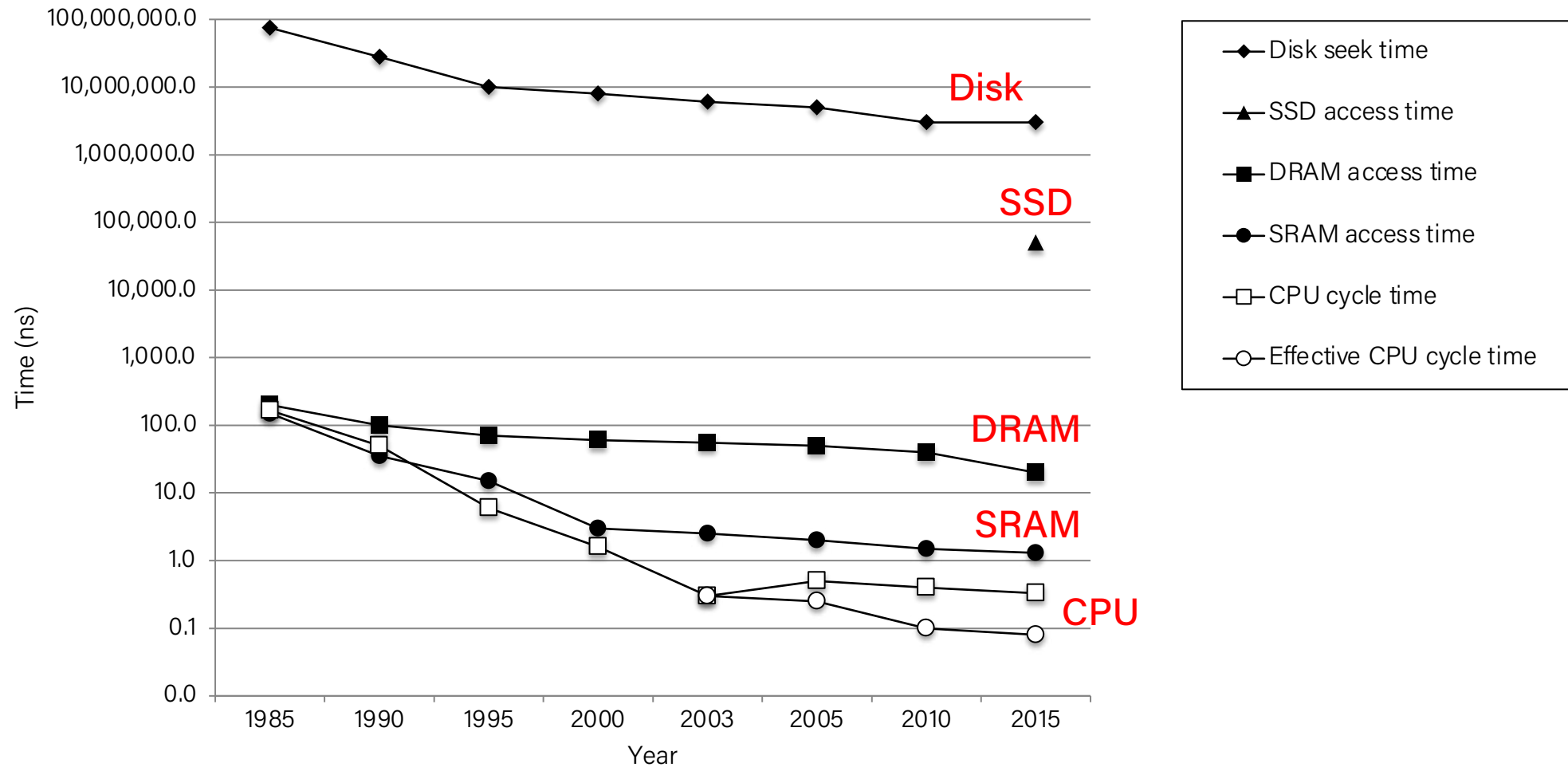
Locality to the Rescue!

- The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

Lecture Plan

- The memory abstraction

- Storage technologies and trends

- **Locality of reference**

- The memory hierarchy

- Cache basics

- Cache organization

Locality

- **Principle of Locality:** 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
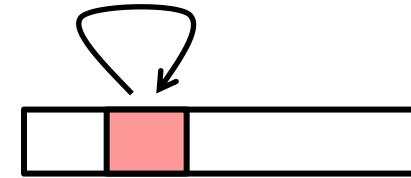
- **Spatial locality:**
  – Items with nearby addresses tend
   to be referenced close together in time
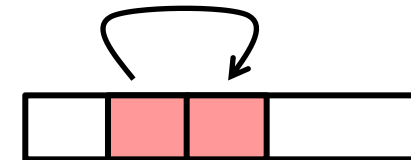
Locality Example

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

- Data references
  - Reference array elements in succession (stride-1 reference pattern). Spatial locality
  - Reference variable sum each iteration. Temporal locality

- Instruction references
  - Reference instructions in sequence. Spatial locality
  - Cycle through loop repeatedly. Temporal locality

Lecture Plan

- The memory abstraction

- Storage technologies and trends

- Locality of reference

- **The memory hierarchy**

- Cache basics

- Cache organization

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.
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.
  - For each level k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1

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

43

Example Memory Hierarchy

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

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

Regs

L1 cache (SRAM)

L2 cache (SRAM)

L3 cache (SRAM)

Main memory (DRAM)

Local secondary storage (local disks)

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

explicitly program-controlled (e.g. refer to exactly %rax, %rbx)

program sees "memory"; hardware manages caching transparently

44

Example Memory Hierarchy

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

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

<1 ns — Regs — 5-10 s

1 ns — L1 cache (SRAM) —

5-10 ns — L2 cache (SRAM) — 1-2 min

L3 cache (SRAM) — 15-30 min

100 ns — Main memory (DRAM) —

150,000 ns — SSD — 31 days

Local secondary storage (local disks)

10,000,000 ns (10 ms) — Disk — 66 months = 5.5 years

1-150 ms — Remote secondary storage (e.g., Web servers) — 1 - 15 years

45

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:
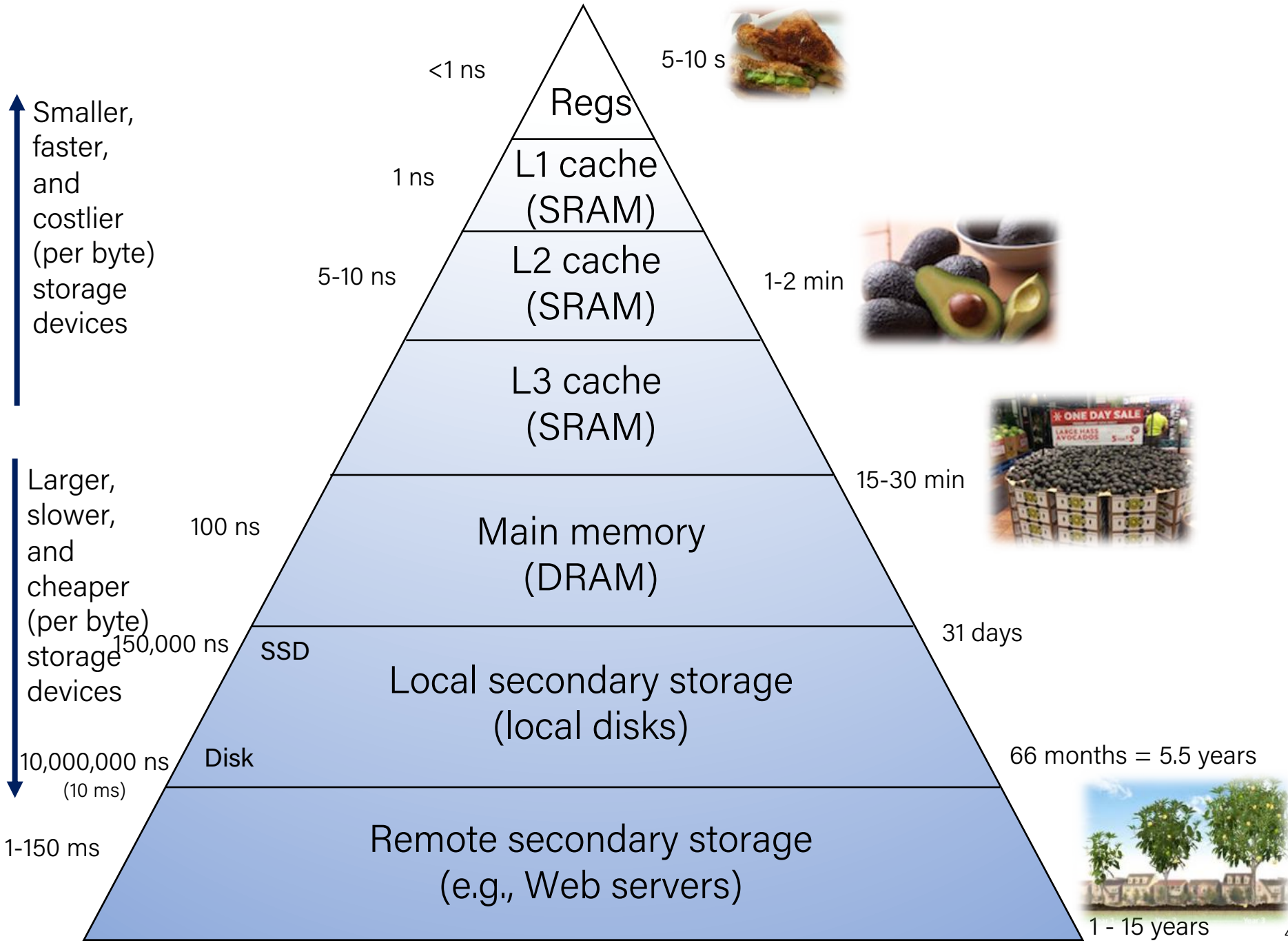 - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

- 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: 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.

Intel Core i7 Cache Hierarchy

Processor package
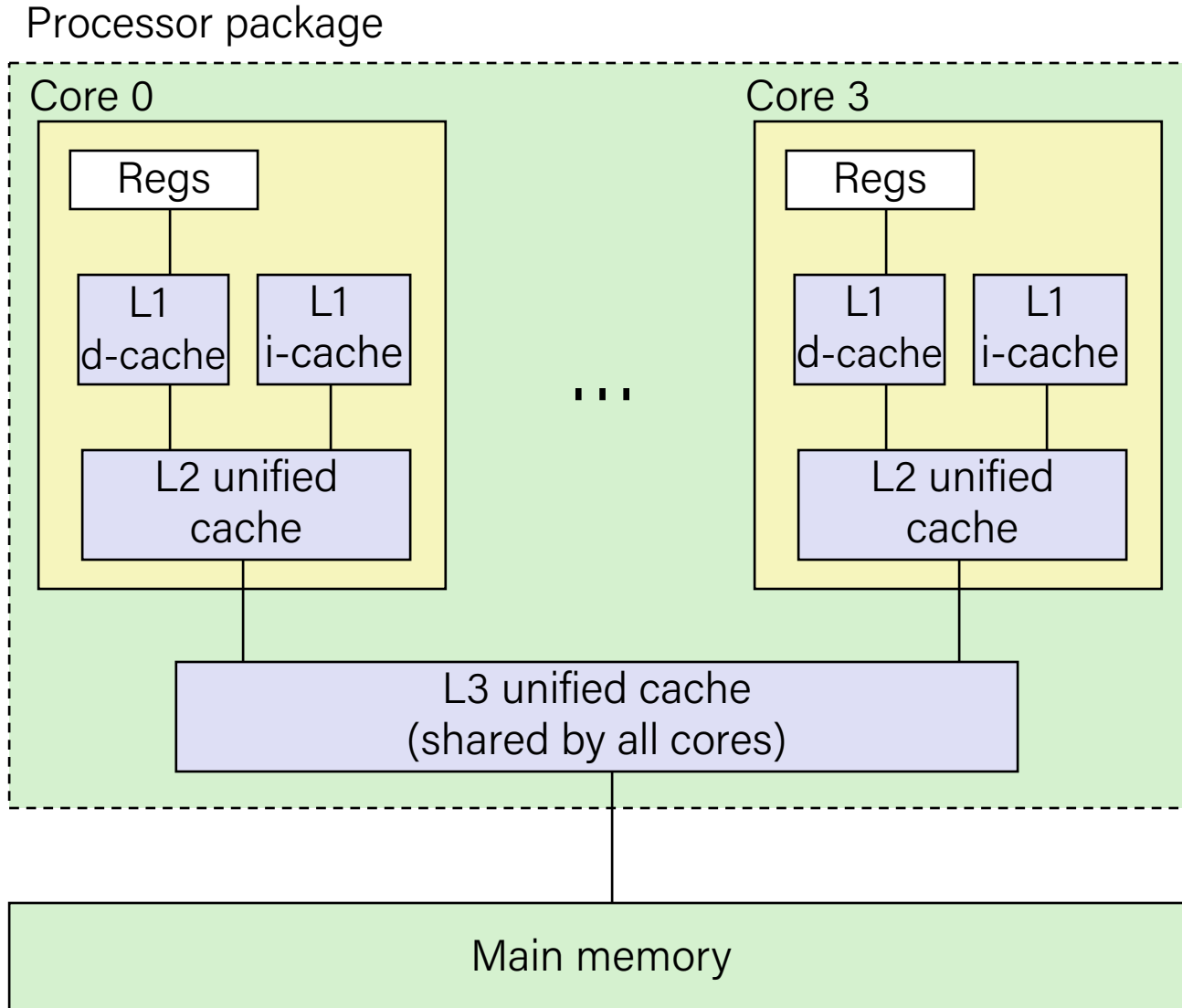
L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes 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

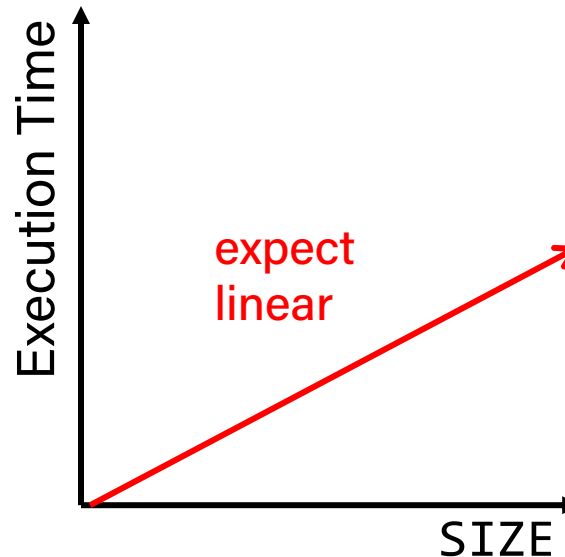# How does execution time grow with SIZE?

```
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    sum += array[j];  ← execute SIZE×200,000 times
  }
}
```
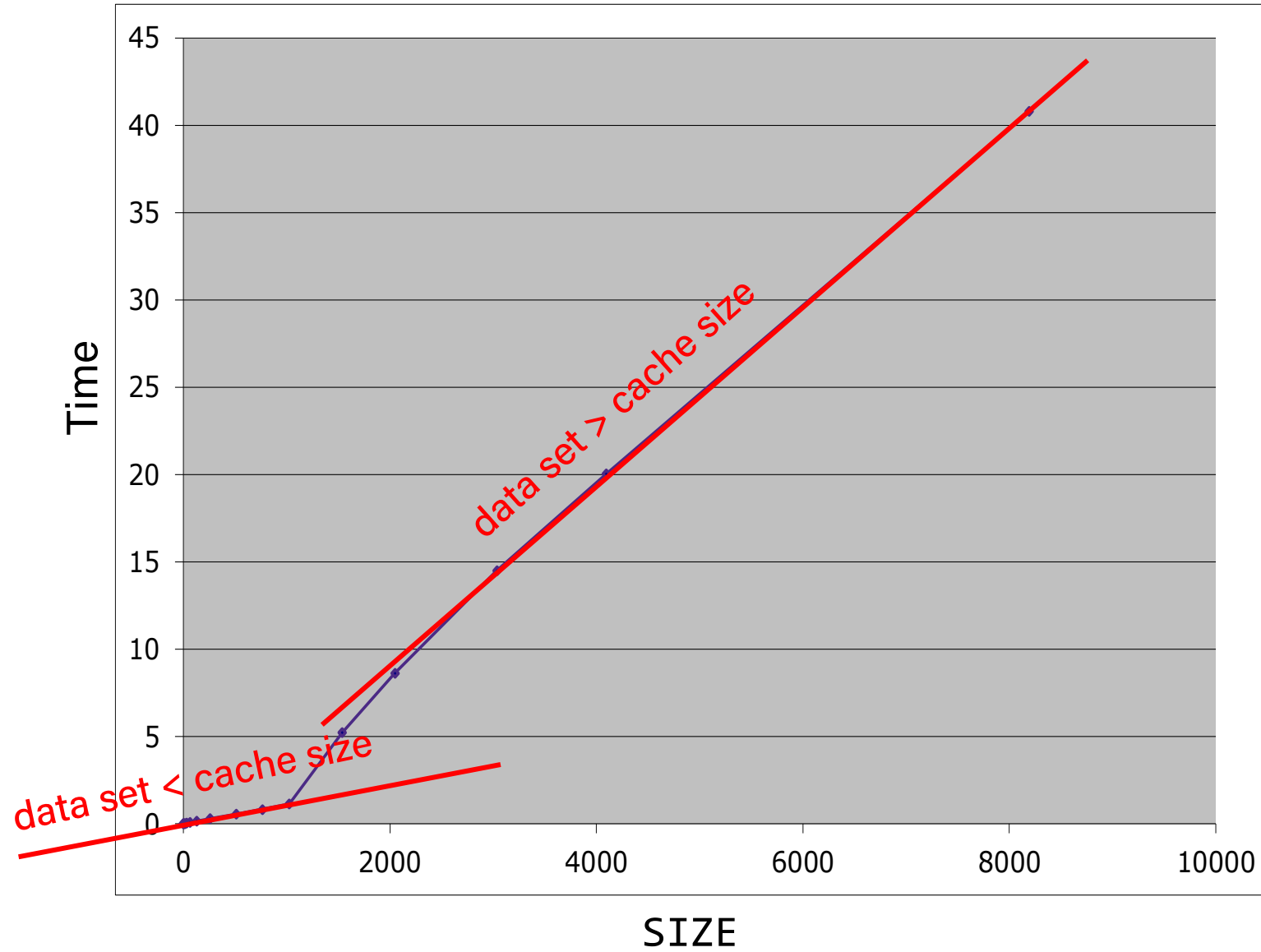
repeat 200,000 times

Plot:

Execution Time

expect linear

SIZE

Actual Data

Processor-Memory Gap
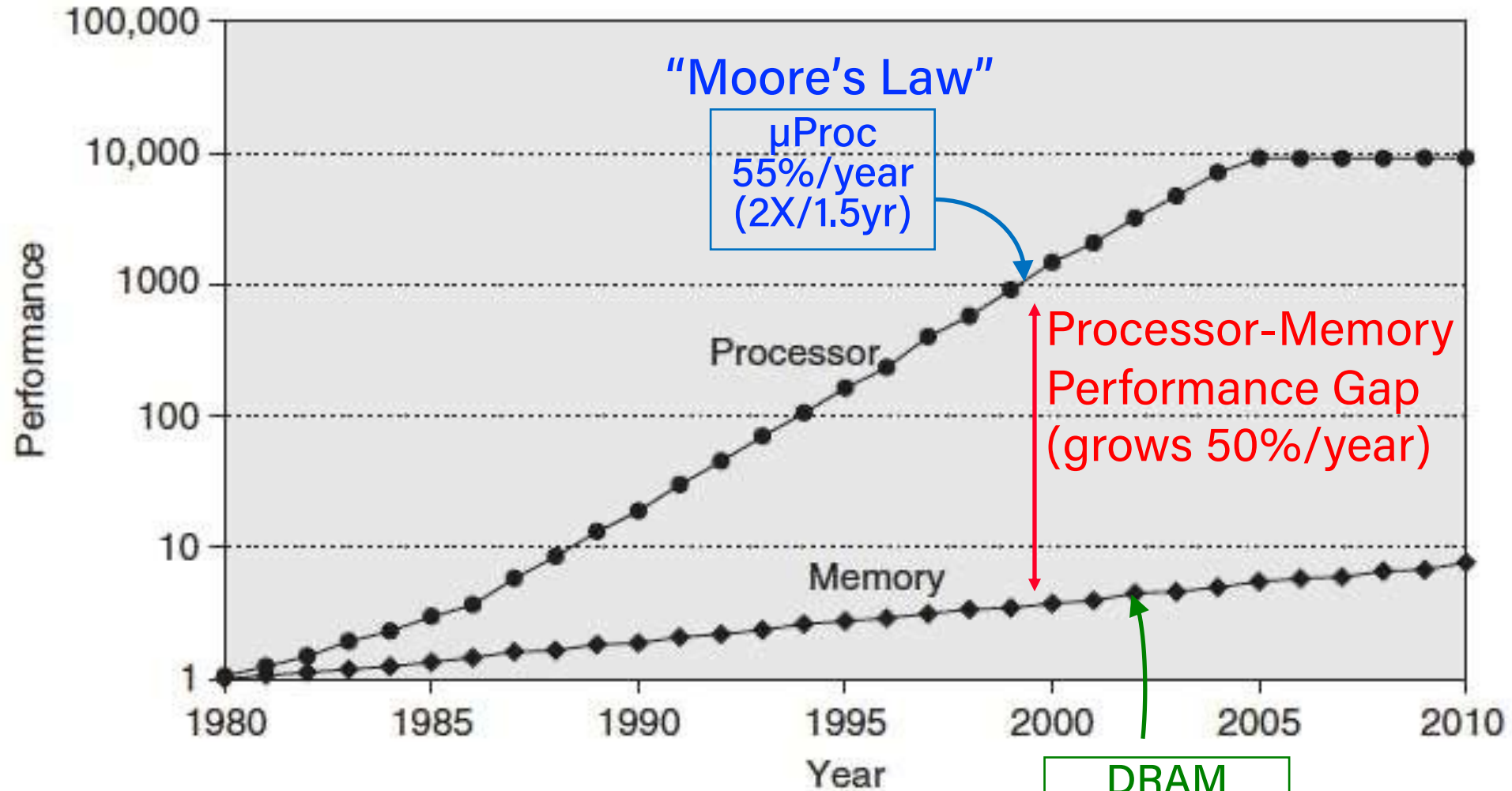
1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip

52

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

| CPU | Reg |

Main Memory
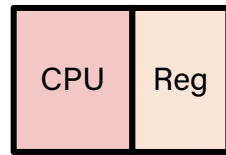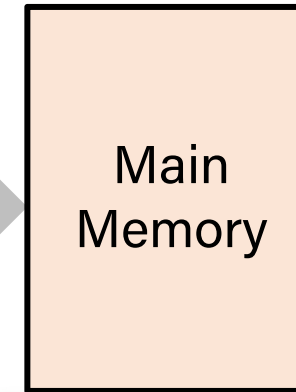
Core 2 Duo:
Can process at least
256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)

cycle: single machine step (fixed-time)

Problem: lots of waiting on memory

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

CPU | **Reg**

Cache

Main Memory
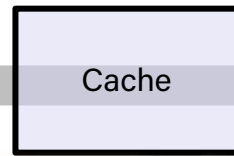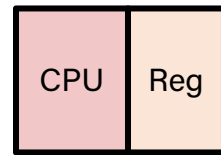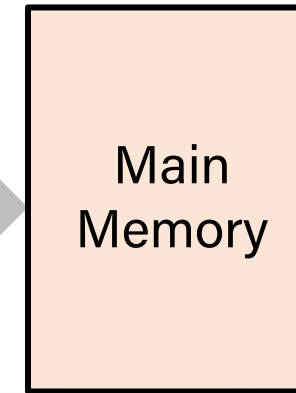
Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100-200 cycles (30-60ns)

cycle: single machine step (fixed-time)

Solution: caches

Suggested by Sinemis Toktaş
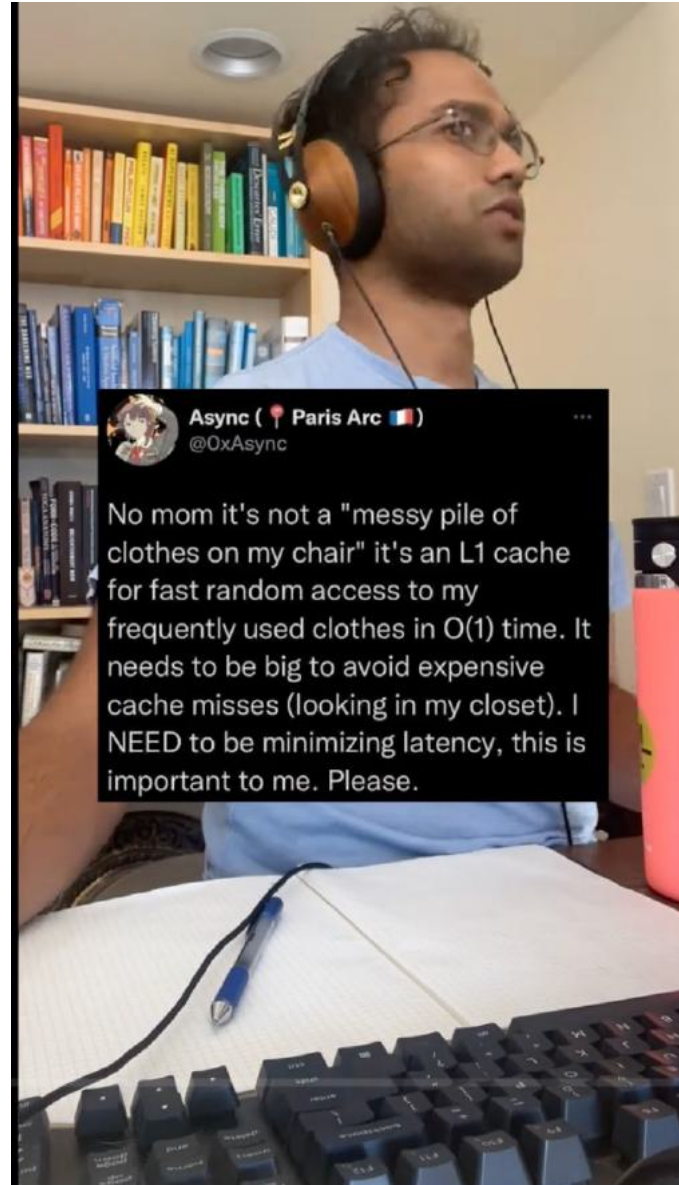
Lecture Plan

- The memory abstraction

- Storage technologies and trends

- Locality of reference

- The memory hierarchy

- Cache basics

- Cache organization

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:

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

Block b is in cache: **Hit!**

Memory

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

General Cache Concepts: Miss

Request: 12

Cache

| 8 | 12 | 14 | 3 |

12   Request: 12

Memory

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

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)

Types of Cache Misses

- **Cold (compulsory) miss**

  – Cold misses occur because the cache is empty.

- **Conflict miss**

  – Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.

    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.

  – Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.

    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, … would miss every time.

- **Capacity miss**

  – Occurs when the set of active cache blocks (working set) is larger than the cache.

Why Caches Work

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

- **Temporal locality:**
  - Recently referenced items are <u>likely</u> to be referenced again in the near future
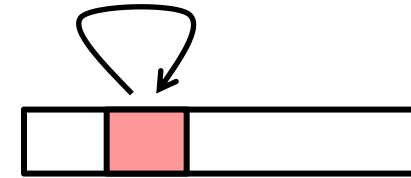
- **Spatial locality:**
  - Items with nearby addresses <u>tend to</u> be referenced close together in time
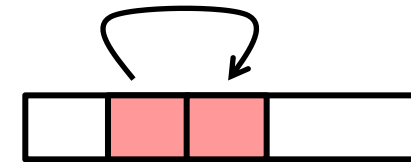
Qualitative Estimates of Locality

- **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`?

```
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;
}
```

Locality Example 1

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

```
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;
}
```

Locality Example 1

- Does this function have good locality with respect to array `a`?

```
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;
}
```

M = 3,
N = 4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = 1

1) a[0][0]
2) a[0][1]
3) a[0][2]
4) a[0][3]
5) a[1][0]
6) a[1][1]
7) a[1][2]
8) a[1][3]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

Layout in Memory

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Note: 76 is just one possible starting address of array a

76   92   108

Locality Example 2

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

```
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;
}
```

M = 3,
N = 4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Layout in Memory

| a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Note: 76 is just one possible starting address of array a

76   92   108

Locality Example 2

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

```
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;
}
```
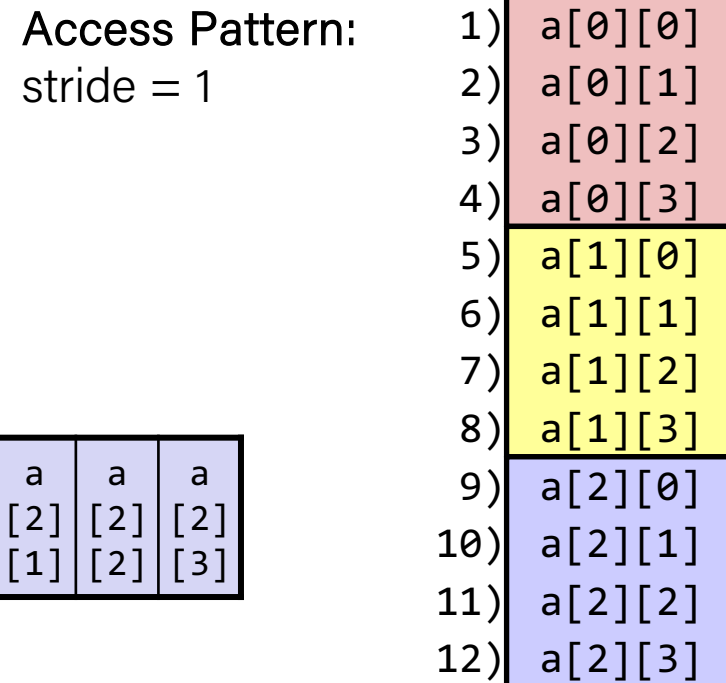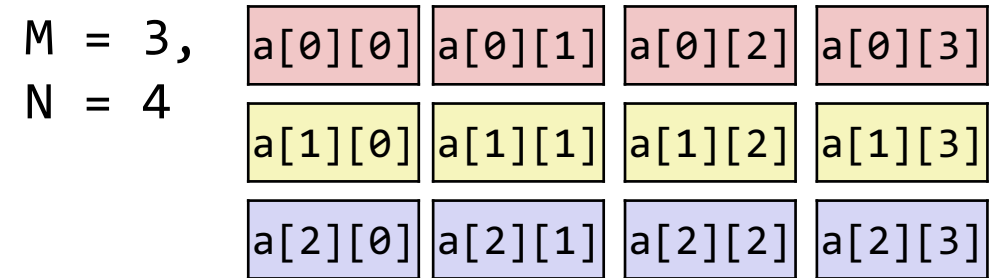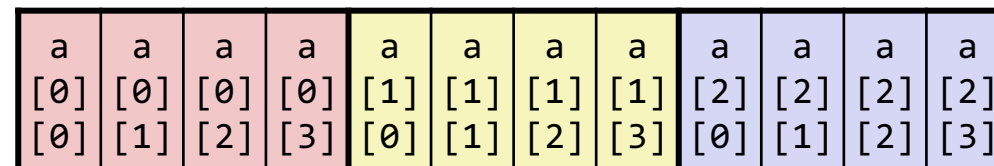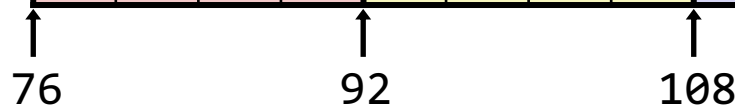
M = 3,
N = 4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = 4

1) a[0][0]
2) a[1][0]
3) a[2][0]
4) a[0][1]
5) a[1][1]
6) a[2][1]
7) a[0][2]
8) a[1][2]
9) a[2][2]
10) a[0][3]
11) a[1][3]
12) a[2][3]

Layout in Memory

| a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Note: 76 is just one possible starting address of array a

76   92   108

Locality Example 3

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

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

- What is wrong with this code?

  Access Pattern: stride-N×L

- How can it be fixed?

  Inner loop: i → stride-1
  j → stride-1
  k → stride-N×L

Layout in Memory (M = 2, N = 3, L = 4)
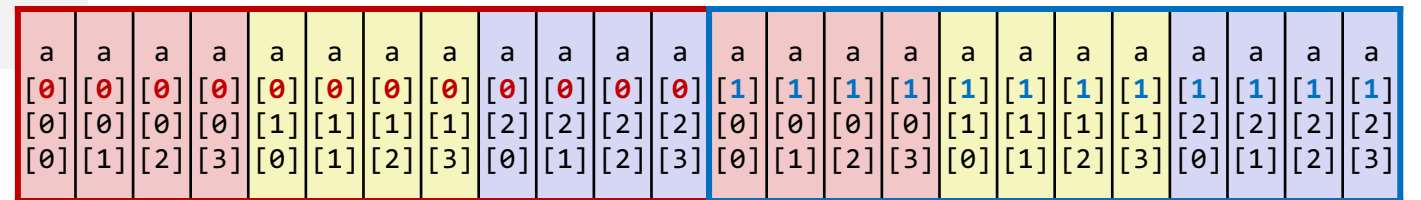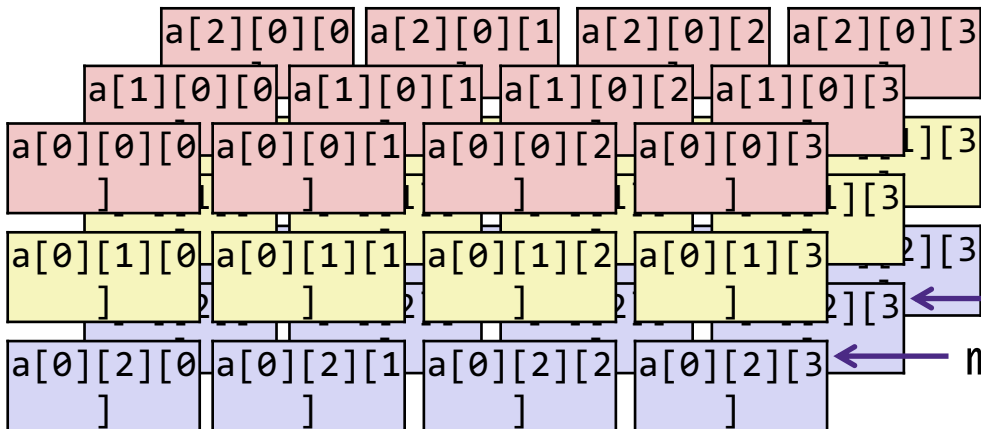
Cache Performance Metrics

- Huge difference between a cache hit and a cache miss
  - Could be 100x speed difference between accessing cache and main memory (measured in *clock cycles*)

- Miss Rate (MR)

  - Fraction of memory references not found in cache (misses / accesses)
    = 1 - Hit Rate

- Hit Time (HT)

  - Time to deliver a block in the cache to the processor
    - Includes time to determine whether the block is in the cache

- Miss Penalty (MP)

  - Additional time required because of a miss

Can we have more than one cache?

- Why would we want to do that?
  - Avoid going to memory!

- Typical performance numbers:
  - Miss Rate
    - L1 MR = 3-10%
    - L2 MR = Quite small (*e.g.* < 1%), depending on parameters, etc.
  - Hit Time
    - L1 HT = 4 clock cycles
    - L2 HT = 10 clock cycles
  - Miss Penalty
    - P = 50-200 cycles for missing in L2 & going to main memory
    - Trend: increasing!

(1) Optimize L1 for high HT
(2) Optimize L2 for low MR

Lecture Plan

- The memory abstraction

- Storage technologies and trends

- Locality of reference

- The memory hierarchy

- Cache basics

- **Cache organization**

Cache Organization

- **Block Size (B):** unit of transfer between cache and main memory
  - Given in bytes and always a power of 2 (*e.g.* 64 bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

Cache Organization

- **Block Size (B):** unit of transfer between cache and main memory
  - Given in bytes and always a power of 2 (*e.g.* 64 bytes)
  - Blocks consist of adjacent bytes (differ in address by 1)
    - Spatial locality!

- Offset field
  - Low-order $\log_2(B) = b$ bits of address tell you which byte within a block
    - (address) mod 2^n = n lowest bits of address
  - (address) modulo (# of bytes in a block)
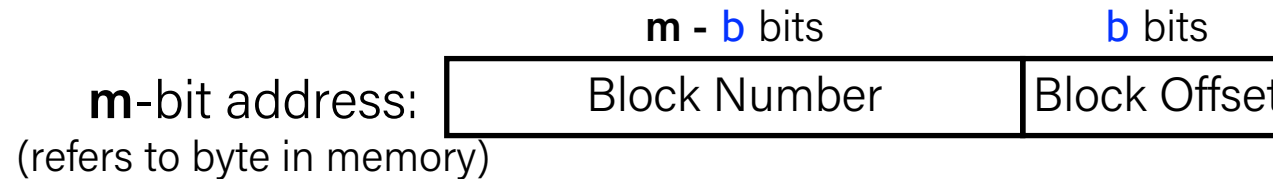
	m - **b** bits	**b** bits
m-bit address:	Block Number	Block Offset

(refers to byte in memory)

Question

- If we have 6-bit addresses and block size B = 4 bytes, which block and byte does `0x15` refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	5	1
D.	5	5
E.	We're lost…	

slido

If we have 6-bit addresses and block size B = 4 bytes, which block and byte does 0x15 refer to?

ⓘ Start presenting to display the poll results on this slide.

Question

- If we have 6-bit addresses and block size B = 4 bytes, which block and byte does `0x15` refer to?

	Block Num	Block Offset
A.	1	1
B.	1	5
C.	**5**	**1**
D.	5	5
E.	We're lost…	

0x   1   5

Address: 0b 0 1 0 1 0 1

Offset width = $\log_2(B) = \log_2(4) = 2$ bits

Cache Organization

- **Cache Size (C):** amount of *data* the cache can store
  - Cache can only hold so much data (subset of next level)
  - Given in bytes (C) or number of blocks (C/B)
  - <u>Example</u>: C = 32 KiB = 512 blocks if using 64-byte blocks

- Where should data go in the cache?
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**

- What is a data structure that provides fast lookup?
  - Hash table!

Review: Hash Tables for Fast Lookup

Insert:

5

27

34

102

119

Apply hash function to map
data to "buckets"

hash(27) % N (10) = 7

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Review: Hash Tables for Fast Lookup

Insert:

5

27

34

102

119

Apply hash function to map
data to "buckets"

hash(27) % N (10) = 7

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Place Data in Cache by Hashing Address

Memory

Block Num   Block Data

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Cache

Index   Block Data

00
01
10
11

Here B = 4 bytes
and C/B = 4
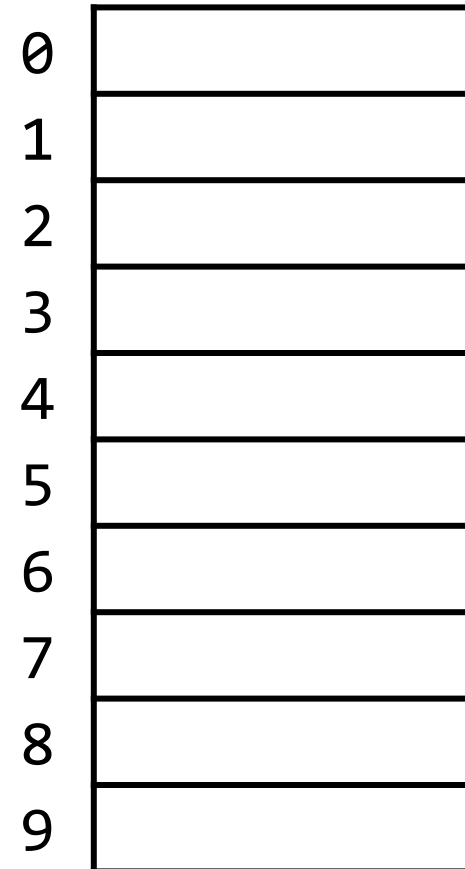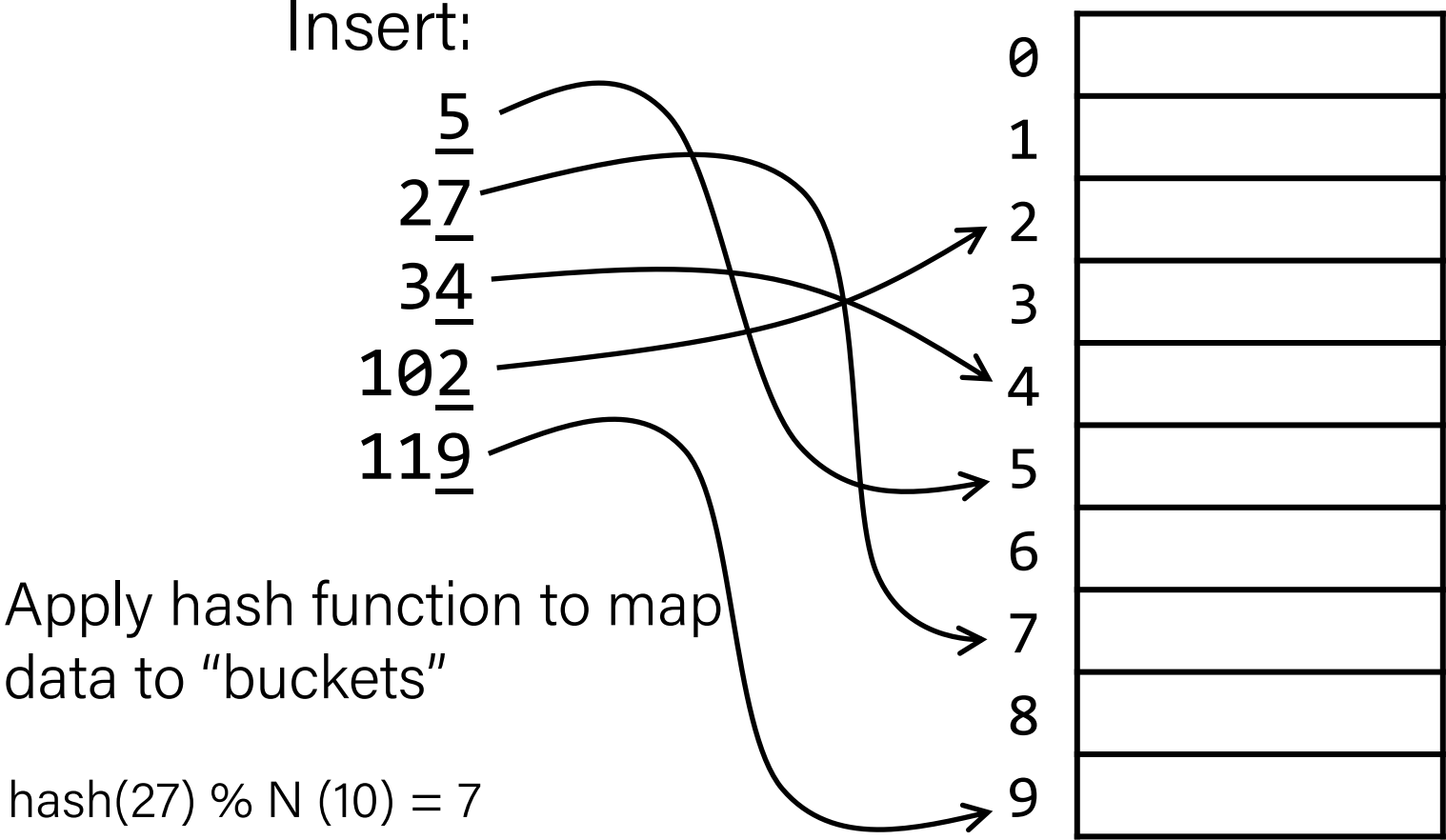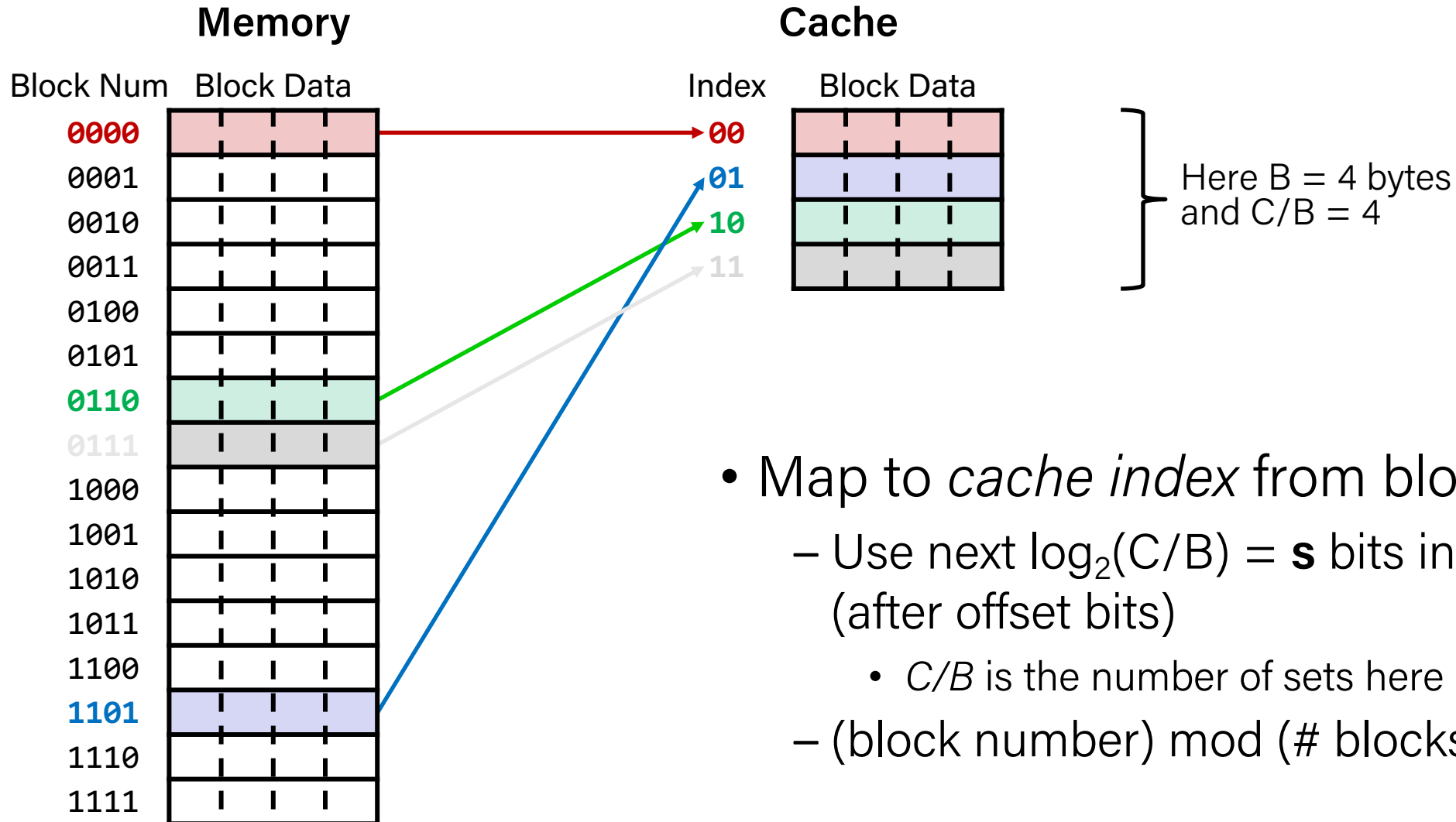
- Map to *cache index* from block number
  - Use next $\log_2(C/B) = $ **s** bits in the address (after offset bits)
    - *C/B* is the number of sets here
  - (block number) mod (# blocks in cache)

Place Data in Cache by Hashing Address

Memory

Cache

Block Num   Block Data

Index   Block Data

Block Num	Block Data
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Index	Block Data
00	
01	
10	
11	

Here B = 4 bytes
and C/B = 4

- Map to *cache index* from block number
  - Let's adjacent blocks fit in cache simultaneously!
    - Consecutive blocks go in consecutive cache indices
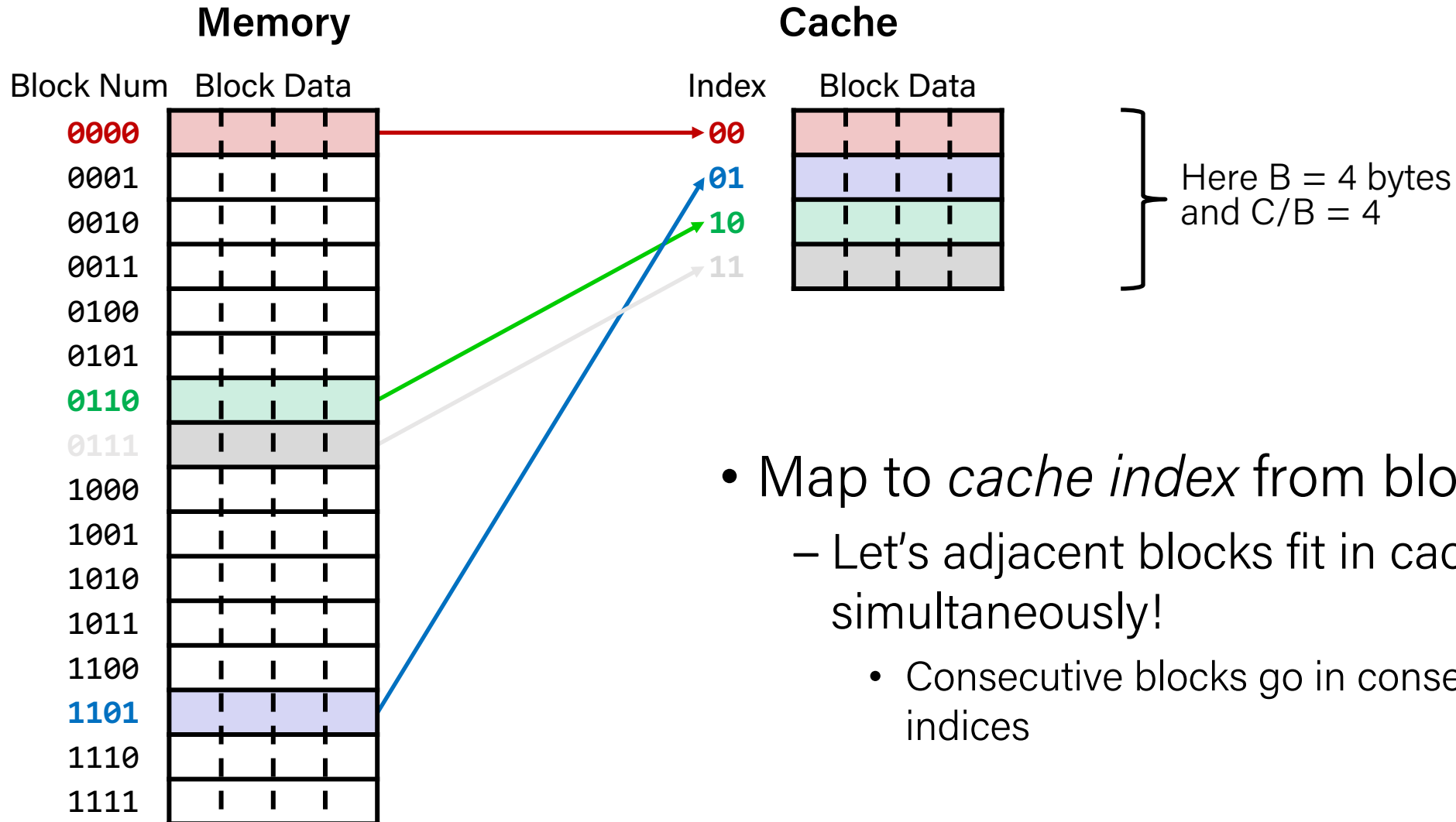
Practice Question

- 6-bit addresses, block size B = 4 bytes, and our cache holds S = 4 blocks.

- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?

Practice Question

- 6-bit addresses, block size B = 4 bytes, and our cache holds S = 4 blocks.

  C = S × B = 16 bytes   b = $\log_2(4)$ = 2 bits   s = $\log_2(4)$ = 2 bits

- A request for address **0x2A** results in a cache miss. Which set index does this block get loaded into and which 3 other addresses are loaded along with it?

  0x   2   A

  Address: 0b 1 0 | 1 0 | 1 0
                          index offset

  block number

  addresses w/block number 1010
  0b101000 = 0x28
  0b101001 = 0x29
  0b101010 = 0x2A
  0b101011 = 0x2B

  These are loaded into cache!

Place Data in Cache by Hashing Address

Memory

Cache

Block Num   Block Data

Index   Block Data

Block Num	
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Index
00
01
10
11

Here B = 4 bytes and C/B = 4

- Collusion!
  - This might confuse the cache later when we access the data
  - Solution?

Tags Differentiate Blocks in Same Index

Memory

Block Num   Block Data

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Cache

Index   Tag   Block Data

Index	Tag	Block Data
00	**00**	
01		
10	01	
11	01	

Here B = 4 bytes
and C/B = 4

- Tag = rest of address bits
  - **t** bits = **m** − **s** − **b**
  - Check this during a cache lookup
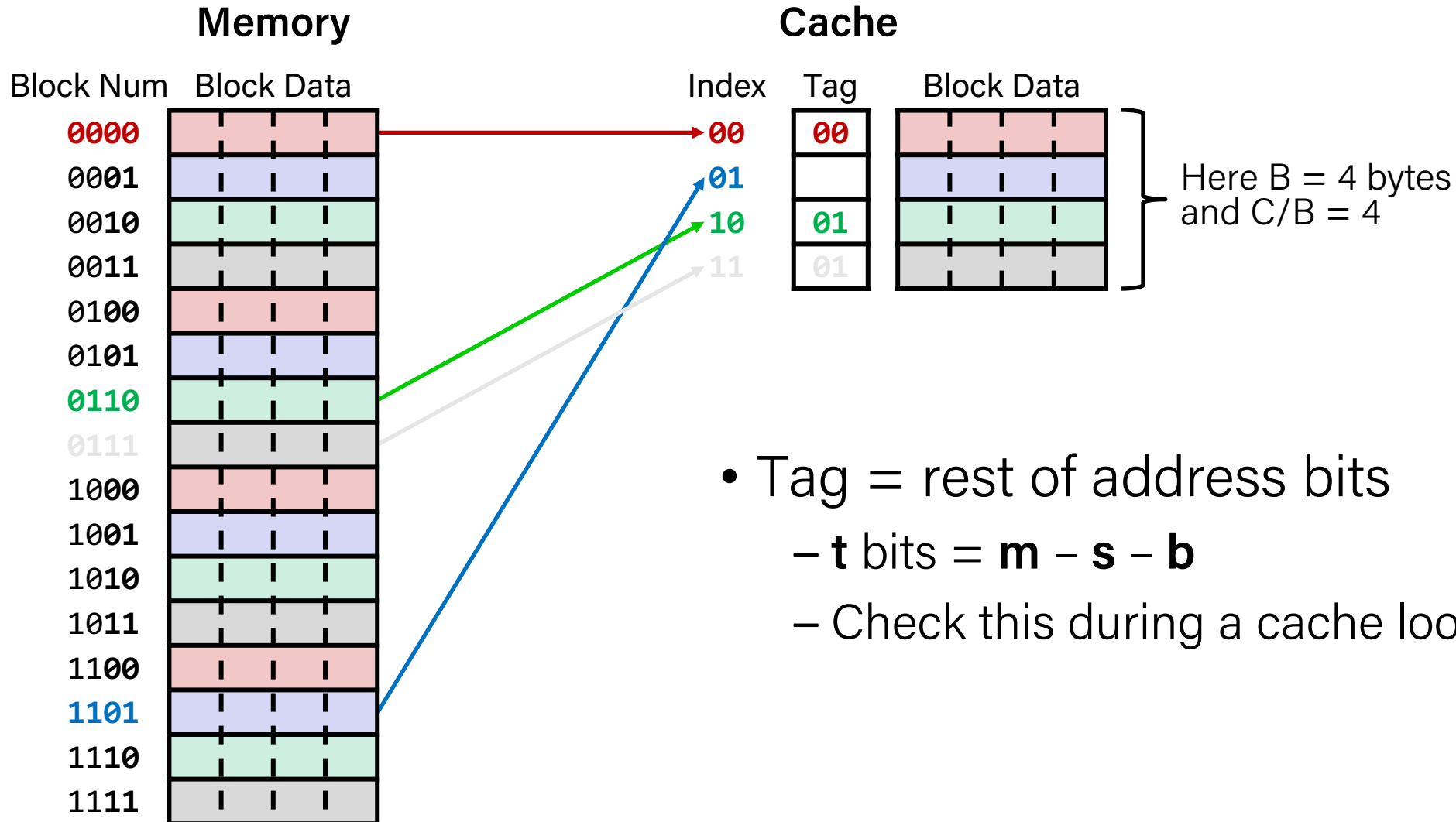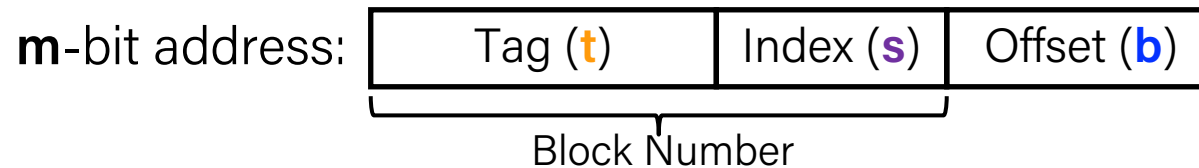
Checking for a Requested Address

- CPU sends address request for chunk of data
  - Address and requested data are not the same thing!
   - Analogy: your friend ≠ their phone number

- TIO address breakdown:

m-bit address:

Tag (**t**)	Index (**s**)	Offset (**b**)

Block Number

  - Index field tells you where to look in cache
  - Tag field lets you check that data is the block you want
  - Offset field selects specified start byte within block

  - **Note: t** and **s** sizes will change based on hash function

Checking for a Requested Address Example

- Using 8-bit addresses.
- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is C/B = 8 sets).
  - Offset bits (b) = $\log_2(B)$ = 2 bits
  - Index bits (s) = $\log_2$(number of sets) = 3 bits
  - Tag bits (t) = Rest of the bits in the address = 8 – 2 – 3 = 3 bits

m-bit address:

Tag (**t**)	Index (**s**)	Offset (**b**)

Block Number
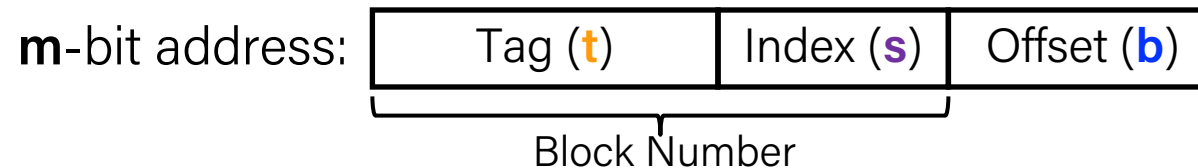
- What are the fields for address 0xBA?
  - Tag bits (unique id for block):
  - Index bits (cache set block maps to):
  - Offset bits (byte offset within block):

Checking for a Requested Address Example

- Using 8-bit addresses.

- Cache Params: block size (B) = 4 bytes, cache size (C) = 32 bytes (which means number of sets is C/B = 8 sets).
  - Offset bits (b) = $\log_2$(B) = 2 bits
  - Index bits (s) = $\log_2$(number of sets) = 3 bits
  - Tag bits (t) = Rest of the bits in the address = 8 – 2 – 3 = 3 bits

m-bit address:

Tag (**t**)	Index (**s**)	Offset (**b**)

Block Number
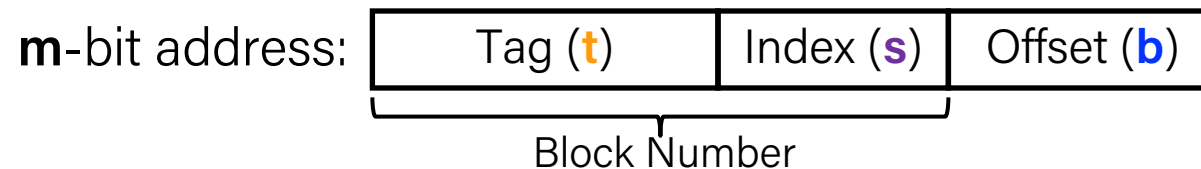
- What are the fields for address 0xBA?
  - Tag bits (unique id for block):   0x5   101 110 10
  - Index bits (cache set block maps to):   0x6   5   6   2
  - Offset bits (byte offset within block): 0x2

Recap

- The memory abstraction
- Storage technologies and trends
- Locality of reference
- The memory hierarchy
- Cache basics
- Cache organization

Next: *More on cache memories*