The Principle of Computer System

Hardware/Software interface

楼学庆

浙江大学计算机学院

http://10.214.47.99/

Email:hzlou@163.com

玉泉校区曹光彪东楼507室

联系方式

- **网站：**
  - http://10.214.47.99
- **邮箱：**
  - hzlou@163.com(*不收作业*)

Course outline

■ **Name：**

Computer Systems

■ **Students：**

Undergraduate students in department.

- Score： 4.5
- Hours/week：3.5-2
- Total：88 hours

THIRD EDITION

COMPUTER SYSTEMS
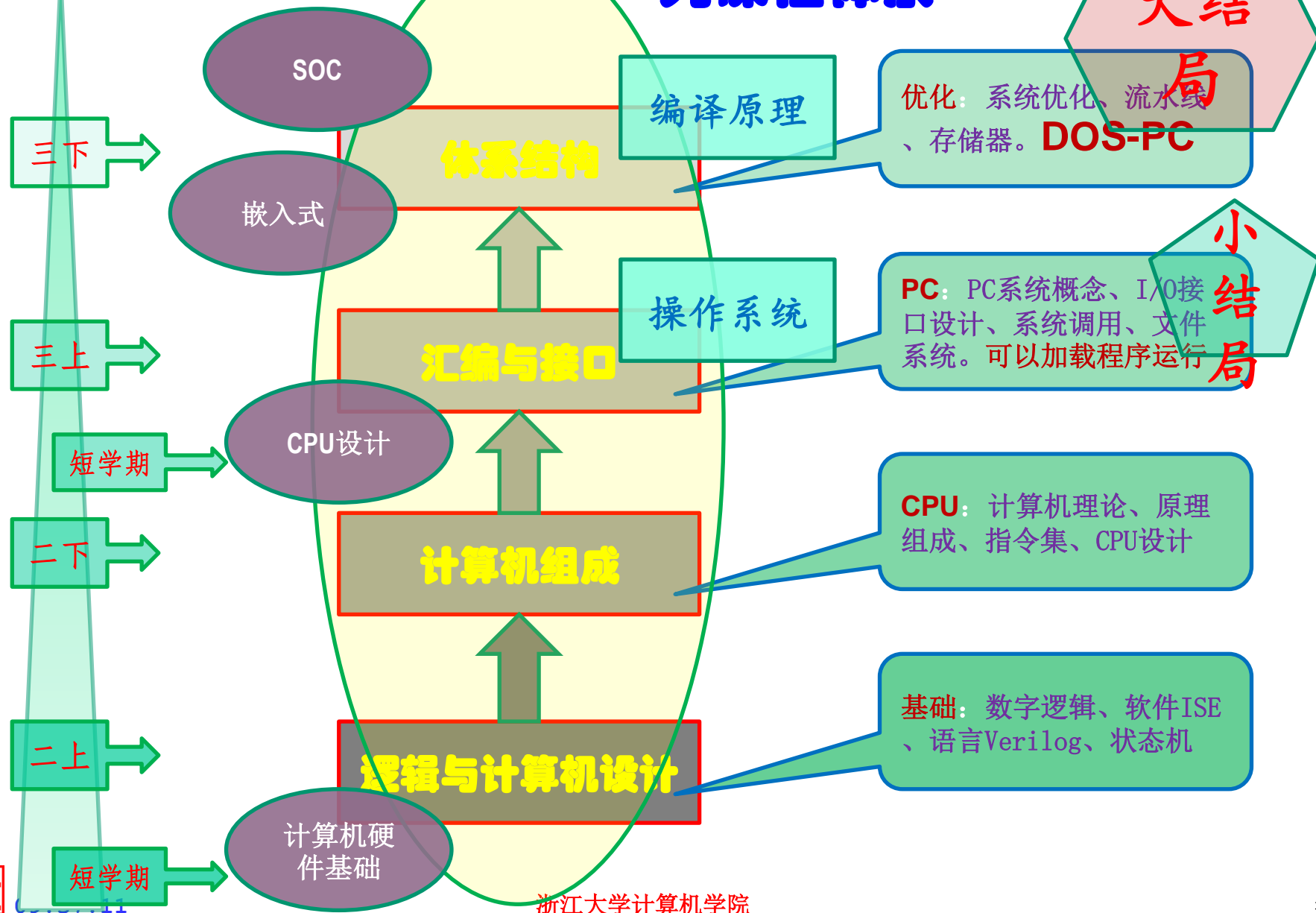A PROGRAMMER'S PERSPECTIVE

BRYANT • O'HALLARON
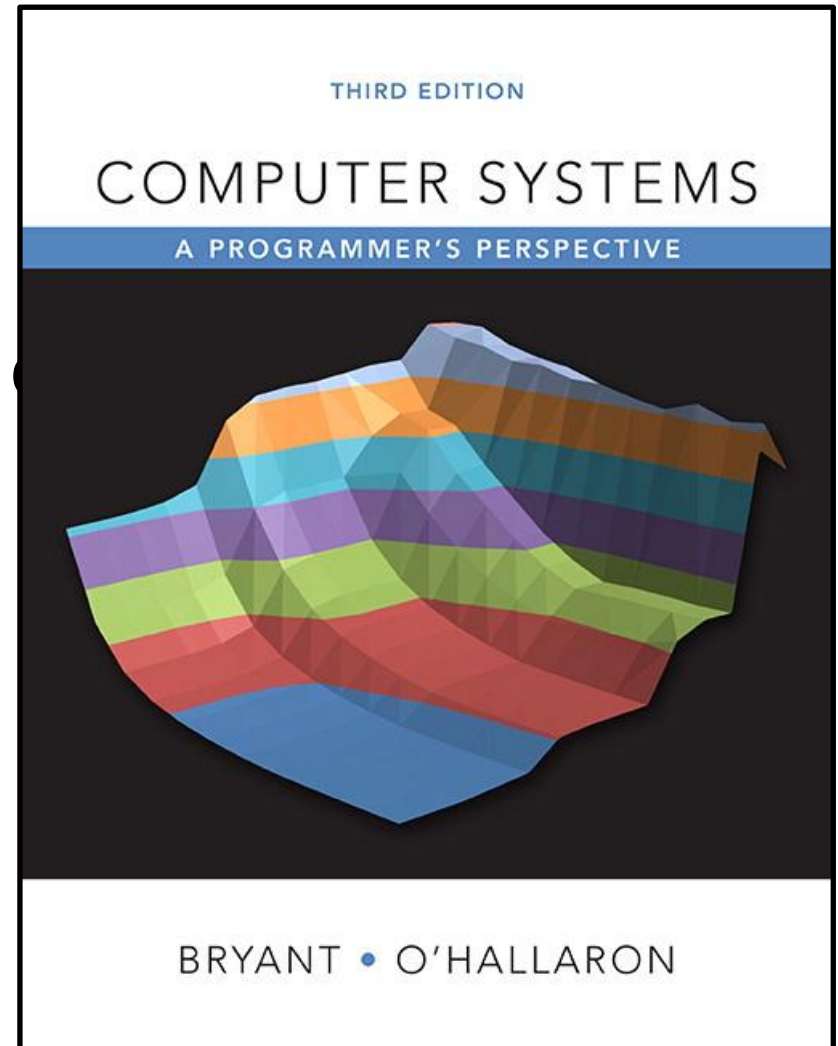
《计算机系统原理》课程目标

- 利用一个学期，覆盖《逻辑与计算机设计》、《计算机组成》、《体系结构》与《汇编与接口》等计算机硬件系统系列主要课程。作为非计算机技术方向，学习了解掌握计算机硬件、计算机系统方面知识的主要课程。
  - ☆前导课程：《C语言程序设计》
- 强烈建议不是只想玩软件的同学，改选　《计算机组成》！

时间轴

主干线

计算机系统能力培养系列课程体系

大结局

小结局

SOC

嵌入式

CPU设计

计算机硬件基础

三下

三上

短学期

二下

二上

短学期

编译原理

体系结构

操作系统

汇编与接口

计算机组成

逻辑与计算机设计

优化：系统优化、流水线、存储器。DOS-PC

PC：PC系统概念、I/O接口设计、系统调用、文件系统。可以加载程序运行

CPU：计算机理论、原理组成、指令集、CPU设计

基础：数字逻辑、软件ISE、语言Verilog、状态机

Machine-Level Programming IV: Data

15-213: Introduction to Computer Systems
8th Lecture, Apr. 24, 2018

Instructors:

Q.

The Memory Hierarchy

15-213: Introduction to Computer Systems
11th Lecture, Oct. 6, 2015

Instructors:

Randal E. Bryant and David R. O'Hallaron

Today

- **Storage technologies and trends**
- **Locality of reference**
- **Caching in the memory hierarchy**

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)

SRAM vs DRAM Summary

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

Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
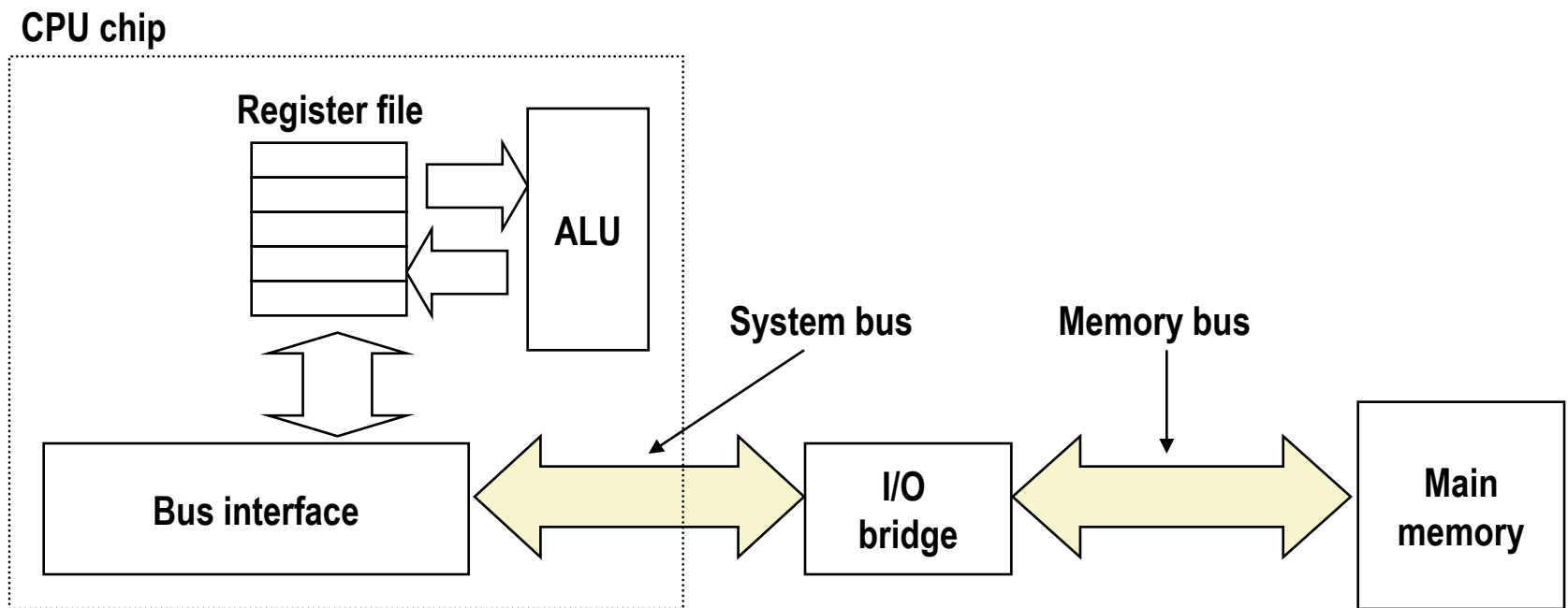 - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
- **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,…)
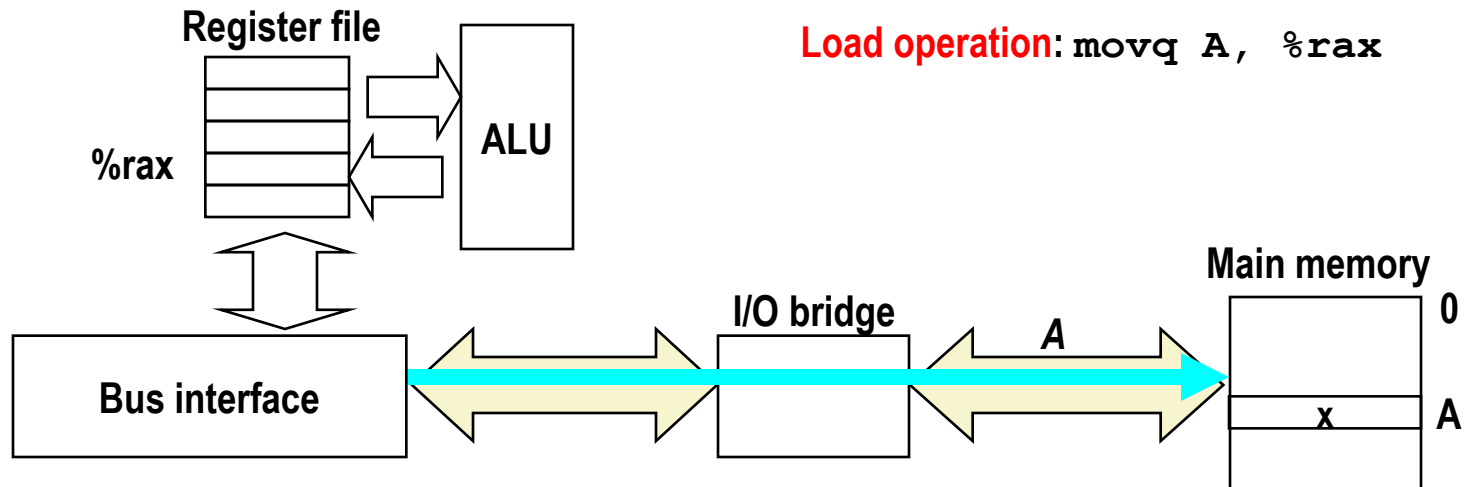 - Disk caches

Traditional Bus Structure Connecting CPU and Memory

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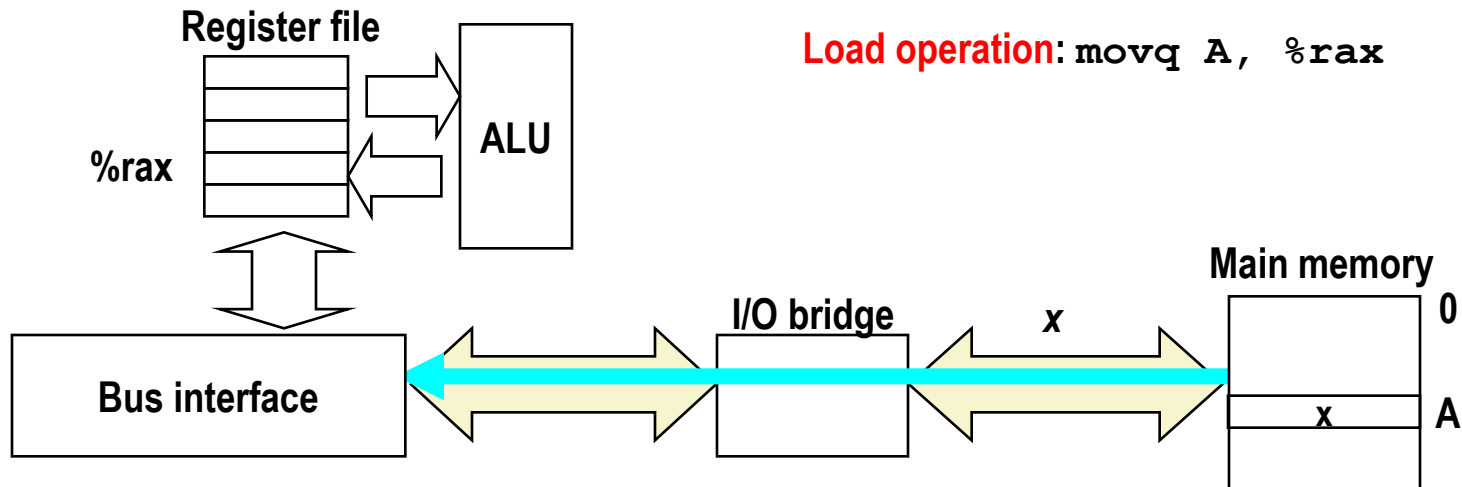
- Buses are typically shared by multiple devices.

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`
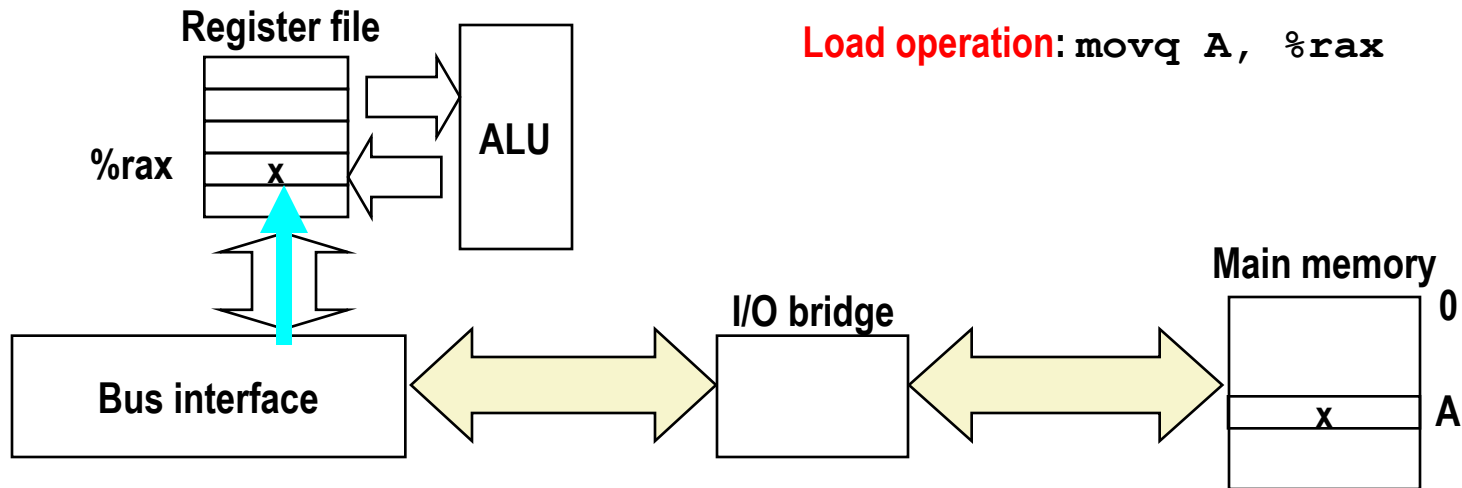
Bus interface

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.**

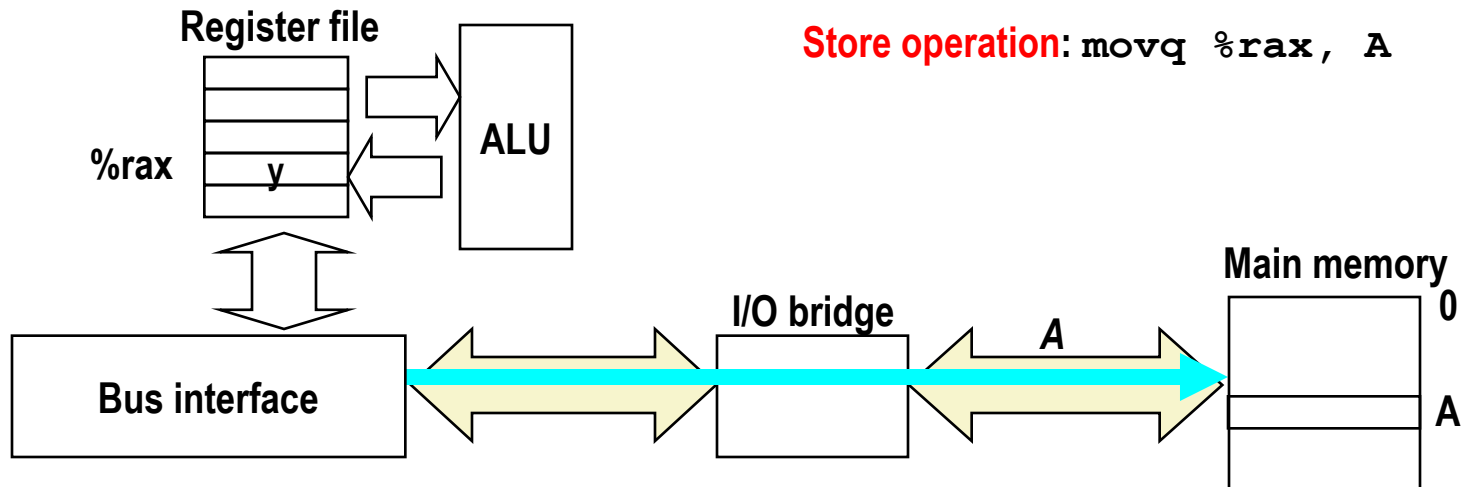**Register file**

%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.**

Load operation: `movq A, %rax`

Register file

%rax | x

ALU

Bus interface

I/O bridge

Main memory

0

x | A

Memory Write Transaction (1)

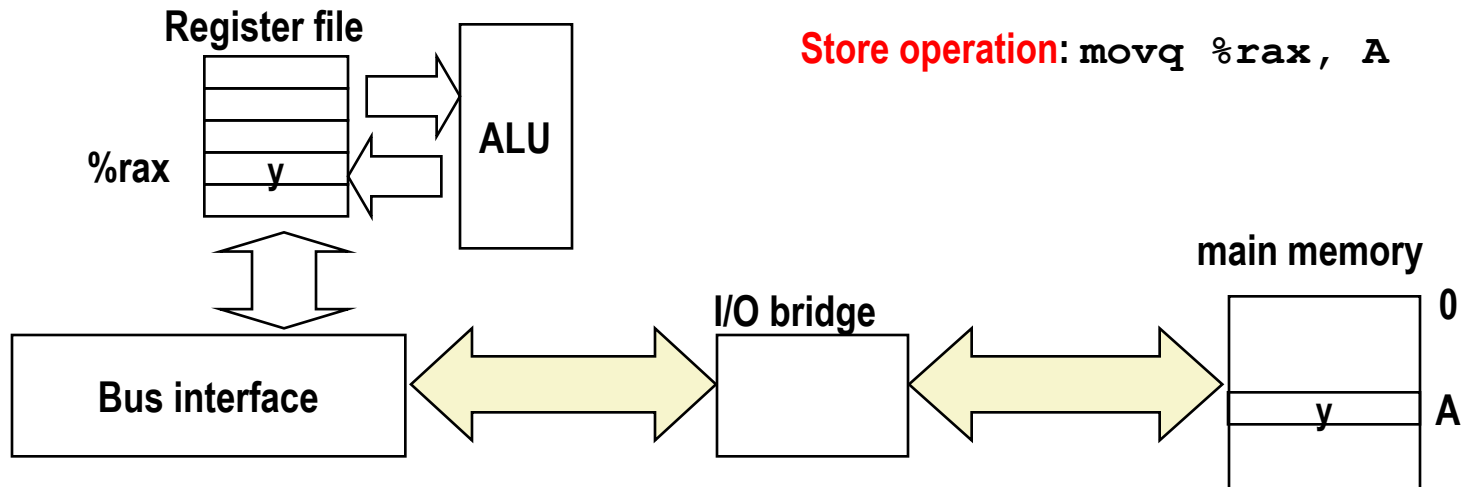■ **CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.**

Register file

Store operation: `movq %rax, A`

ALU

%rax y

Bus interface

I/O bridge

A

Main memory

0

A

Memory Write Transaction (2)

■ **CPU places data word y on the bus.**

Register file

%rax | y

ALU

Store operation: `movq %rax, A`

Bus interface

I/O bridge

y

Main memory

0

A
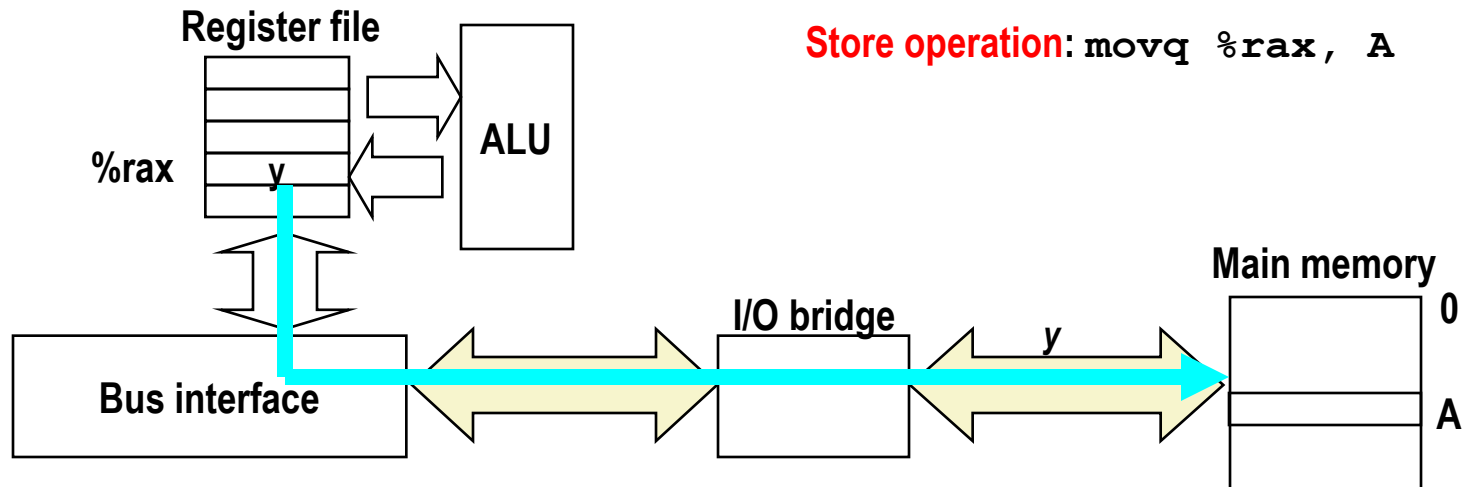
Memory Write Transaction (3)

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

Store operation: `movq %rax, A`

Register file

%rax [y]

ALU

Bus interface

I/O bridge

main memory

0

y   A

What's Inside A Disk Drive?

Spindle

Arm

Platters

Actuator

Electronics (including a processor and memory!)

SCSI connector
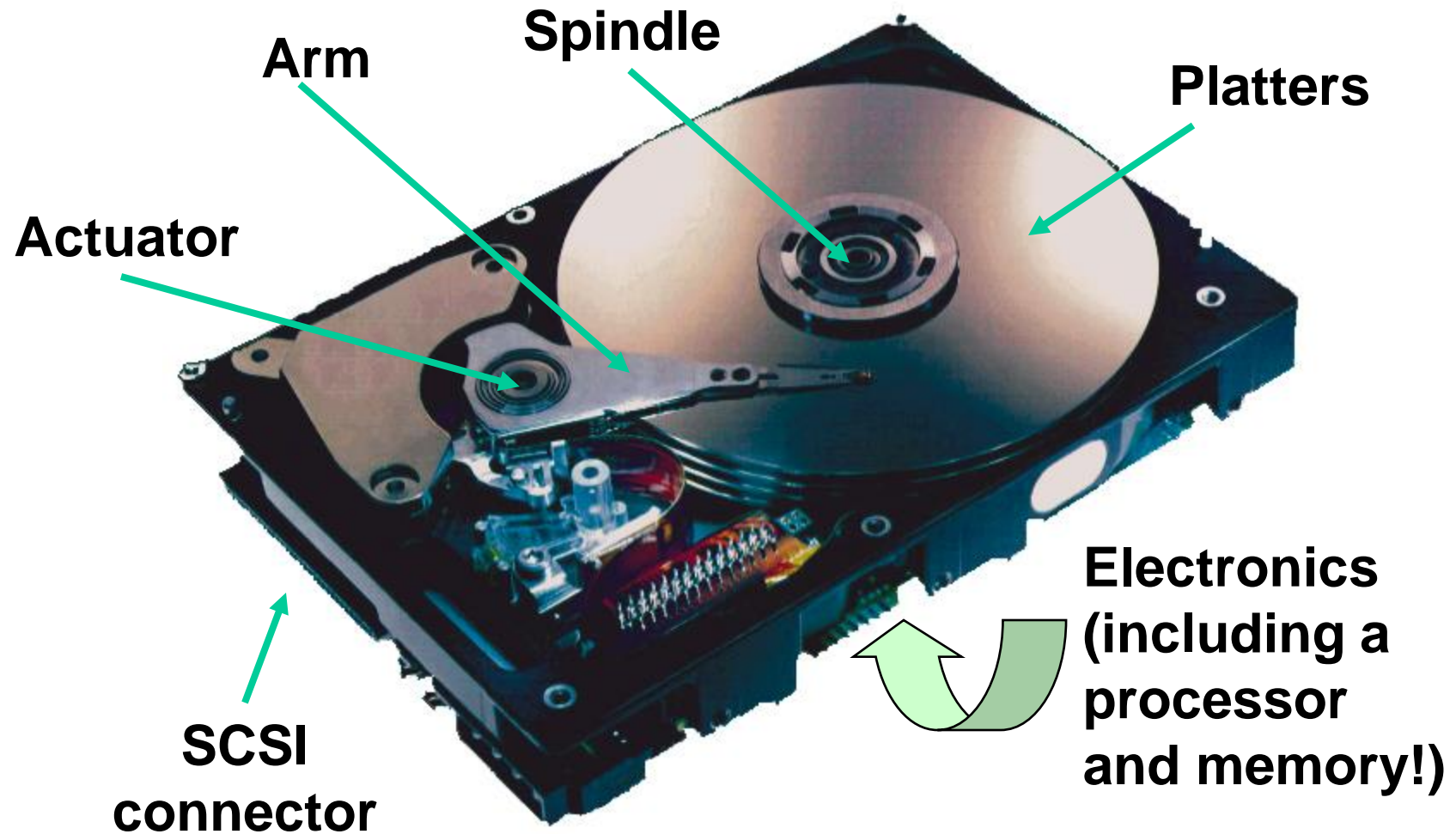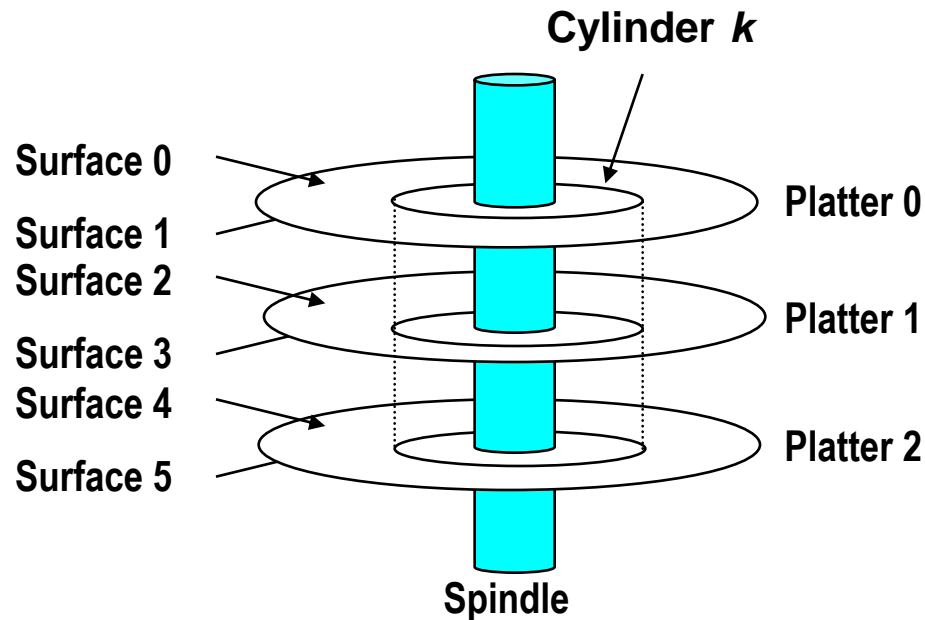
Image courtesy of Seagate Technology

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.**
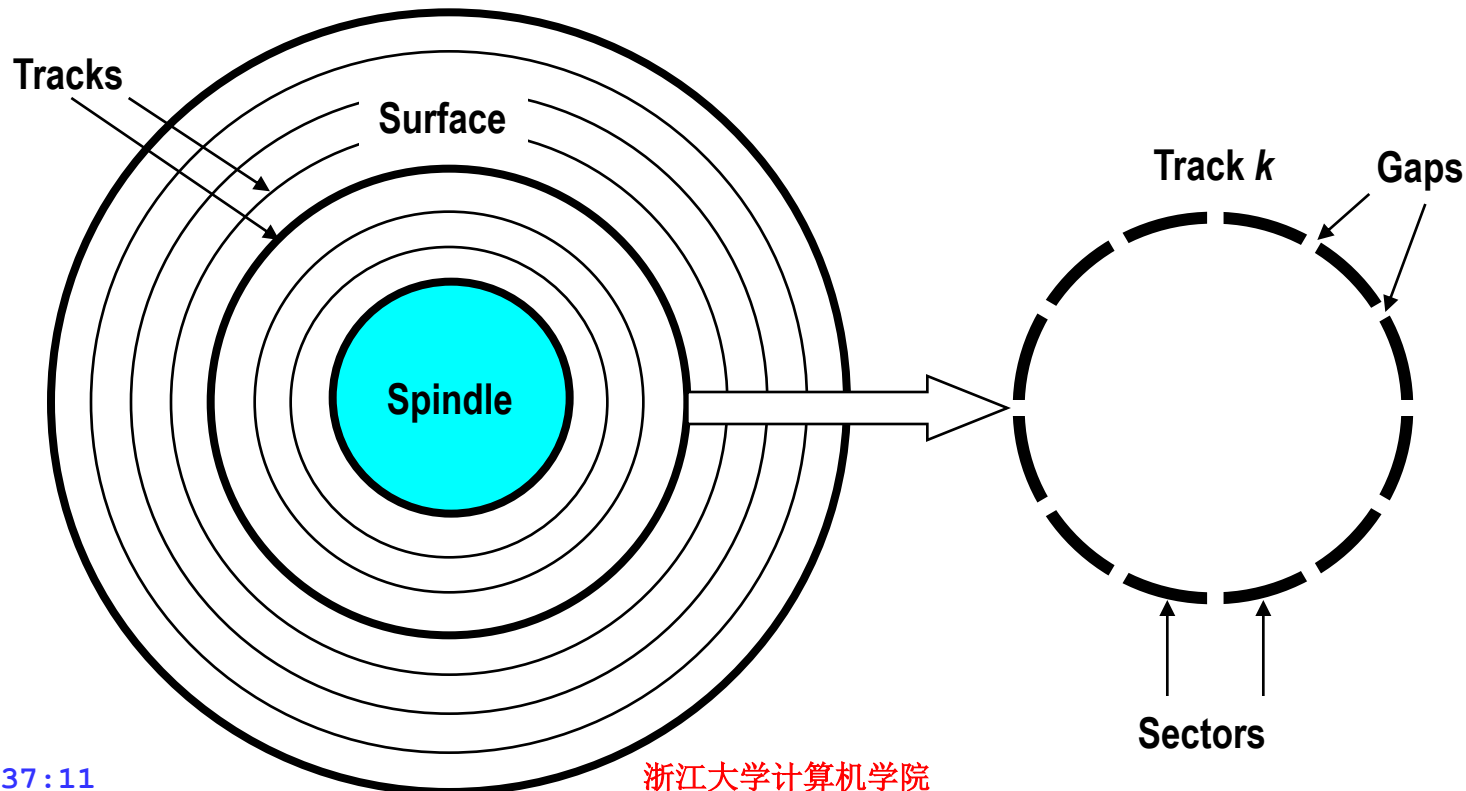
# Disk Geometry (Muliple-Platter View)

- **Aligned tracks form a cylinder.**

Disk Capacity

- **Capacity: maximum number of bits that can be stored.**
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9 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/in2): product of recording and track density.

Recording zones

- **Modern disks partition tracks into disjoint subsets called recording zones**

  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.

  - Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.

  - So we use **average** number of sectors/track when computing capacity.

Spindle
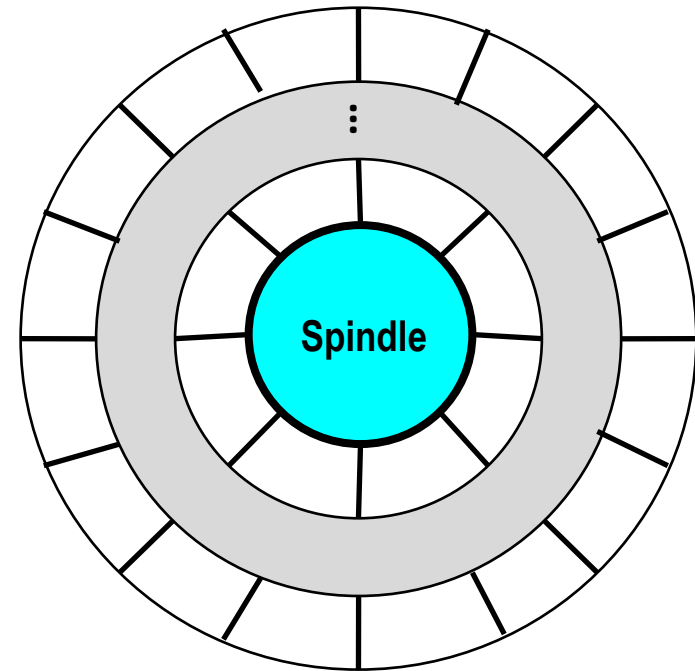
Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x

(# tracks/surface) x (# surfaces/platter) x

(# platters/disk)

Example:
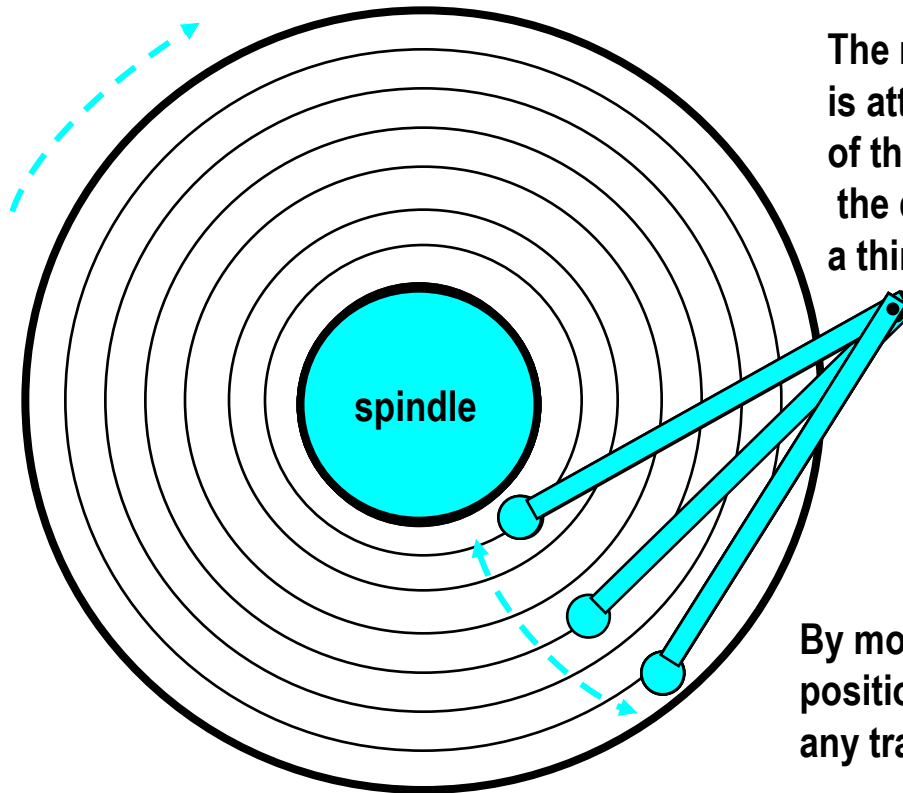
- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 x 300 x 20000 x 2 x 5

= 30,720,000,000

= 30.72 GB

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.

浙江大学计算机学院

Disk Operation (Multi-Platter View)

Read/write heads
move in unison
from cylinder to cylinder

Arm

Spindle
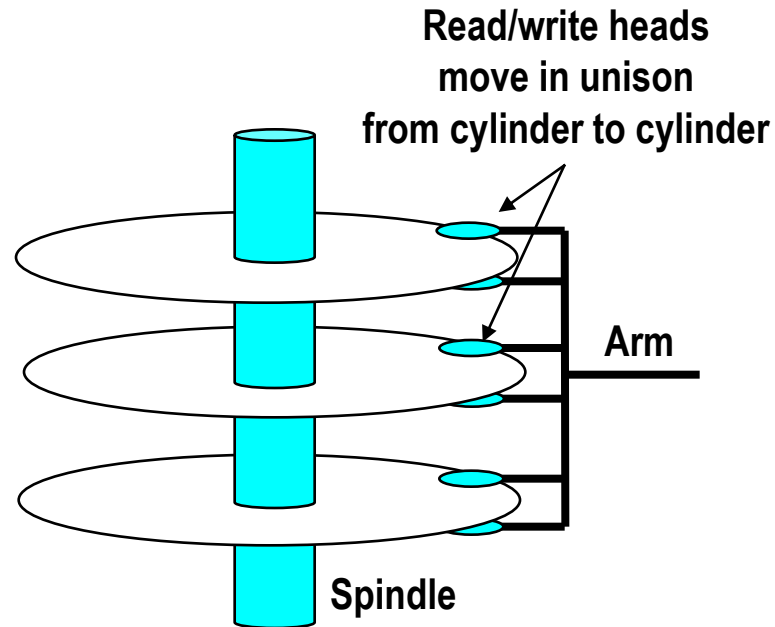
Disk Structure - top view of single platter
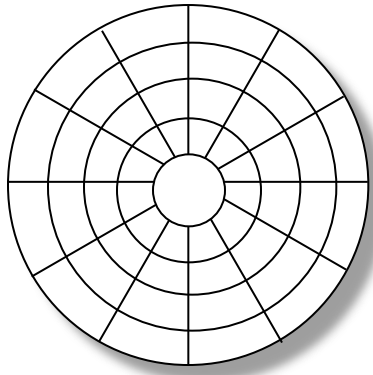
Surface organized into tracks

Tracks divided into sectors

Disk Access
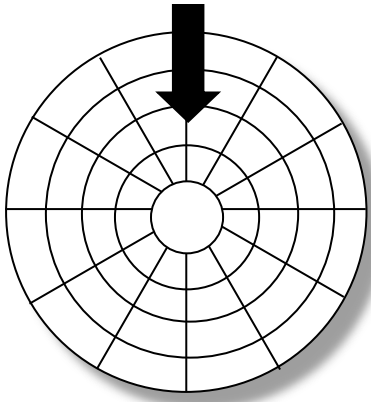
Head in position above a track

Disk Access
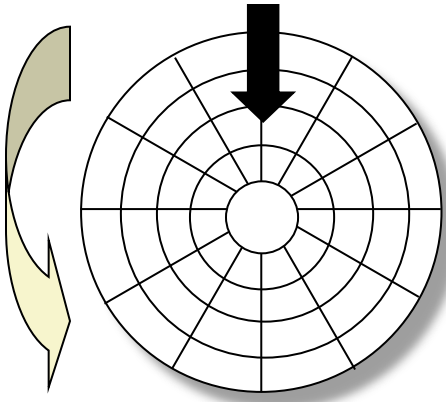
Rotation is counter-clockwise

Disk Access – Read
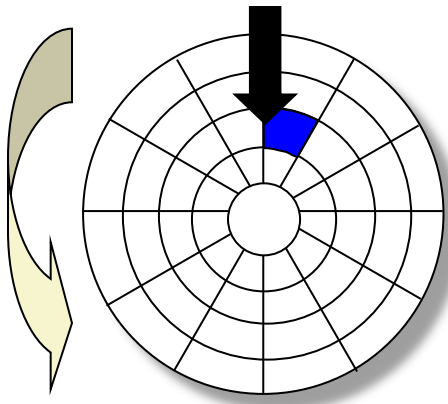
About to read blue sector

Disk Access – Read
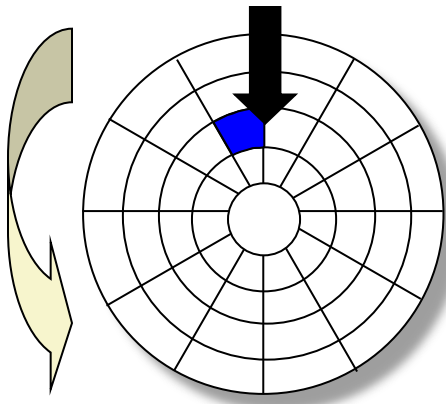
After **BLUE** read

After reading blue sector

Disk Access – Read
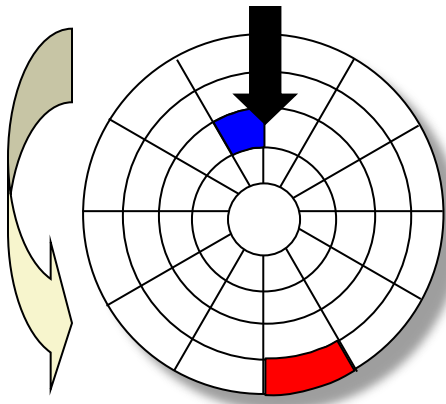
After **BLUE** read

Red request scheduled next

Disk Access – Seek
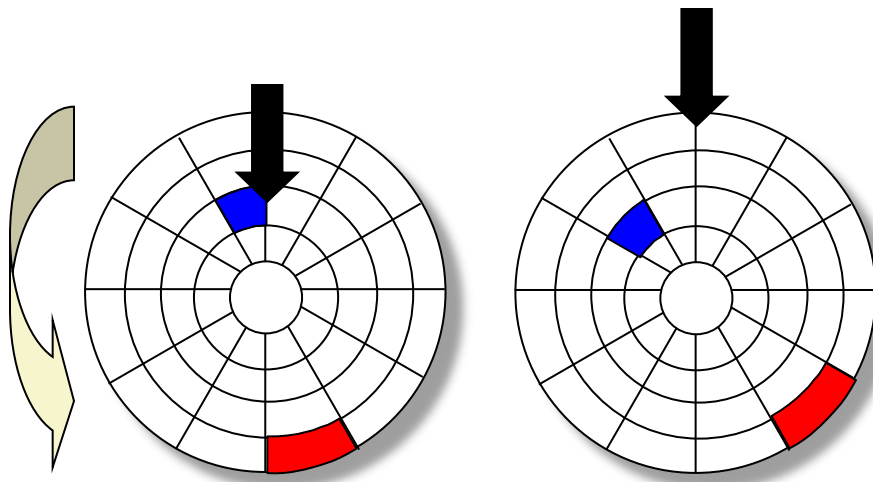
After BLUE read   **Seek for RED**

Seek to red's track

Disk Access – Rotational Latency

After BLUE read   **Seek for RED**   **Rotational latency**
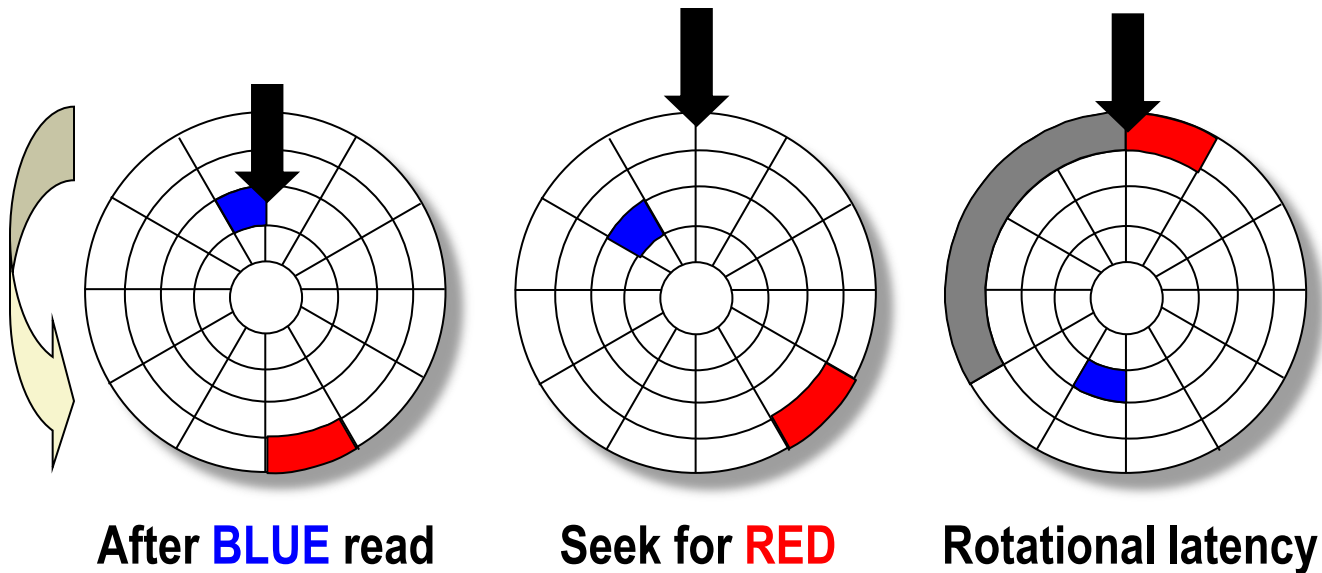
Wait for red sector to rotate around

Disk Access – Read

After BLUE read　　**Seek for RED**　　**Rotational latency**　　**After RED read**
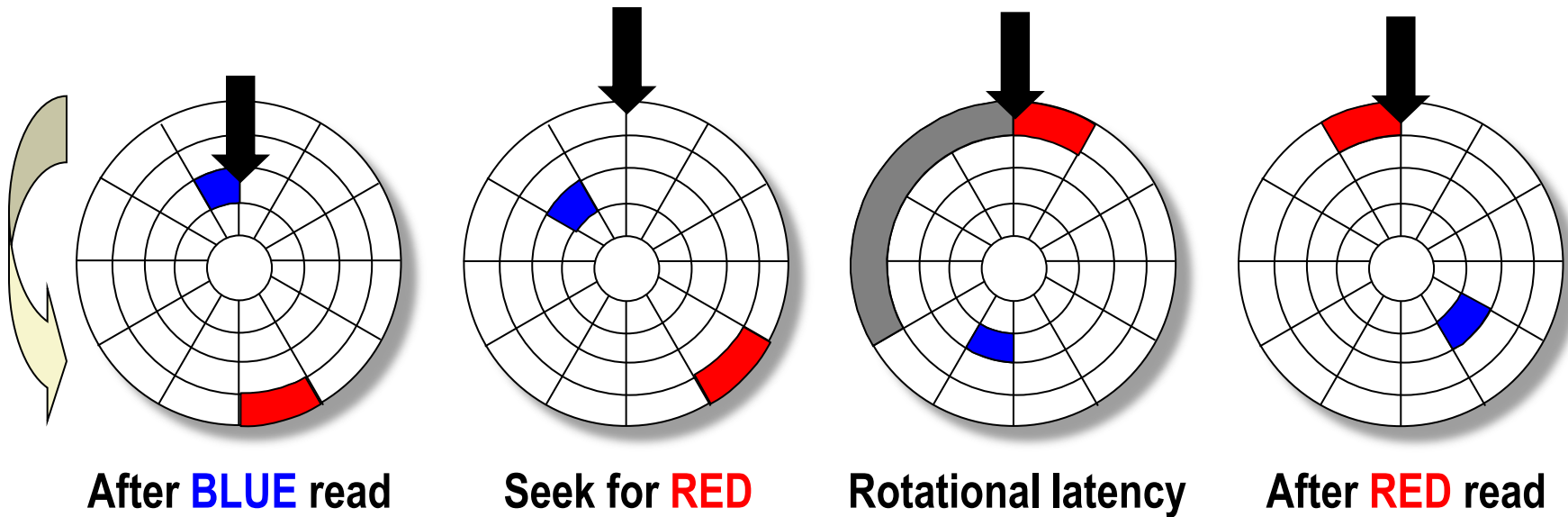
Complete read of red

Disk Access – Service Time Components

After **BLUE** read   Seek for **RED**   Rotational latency   After **RED** read

Data transfer   **Seek**   **Rotational latency**   **Data transfer**
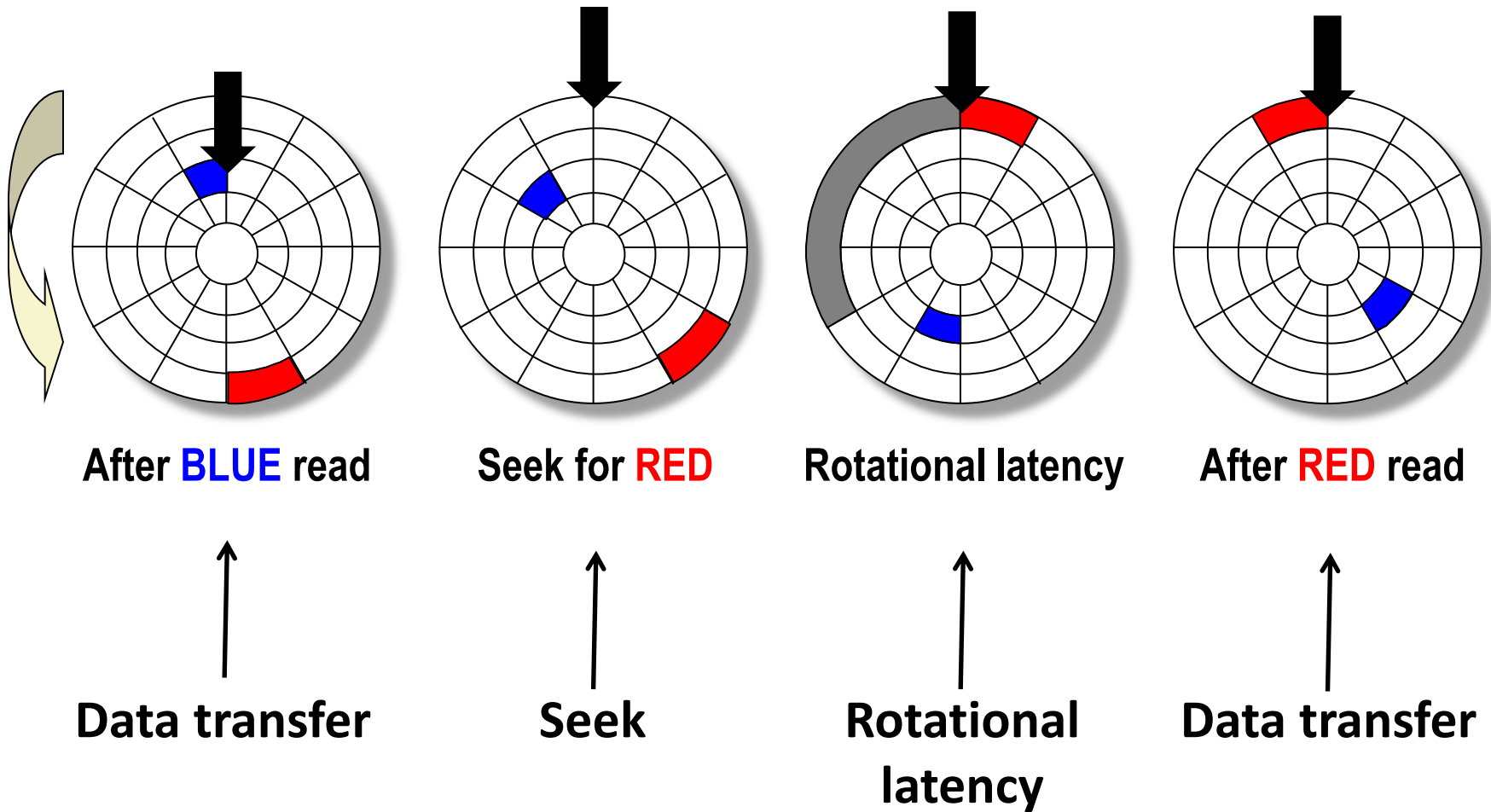
Disk Access Time

- **Average time to access some target sector approximated by :**
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer

- **Seek time (Tavg seek)**
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms

- **Rotational latency (Tavg rotation)**
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs

- **Transfer time (Tavg transfer)**
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

Disk Access Time Example

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

- **Derived:**
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM x 1/400 secs/track x 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 then DRAM

Logical Disk Blocks

- **Modern disks present a simpler abstract view of the complex sector geometry:**

  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, …)

- **Mapping between logical blocks and actual (physical) sectors**

  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.

- **Allows controller to set aside spare cylinders for each zone.**

  - Accounts for the difference in "formatted capacity" and "maximum capacity".

I/O Bus

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

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

I/O bus

USB controller

Graphics adapter

Disk controller

mouse   keyboard

Monitor

Disk

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
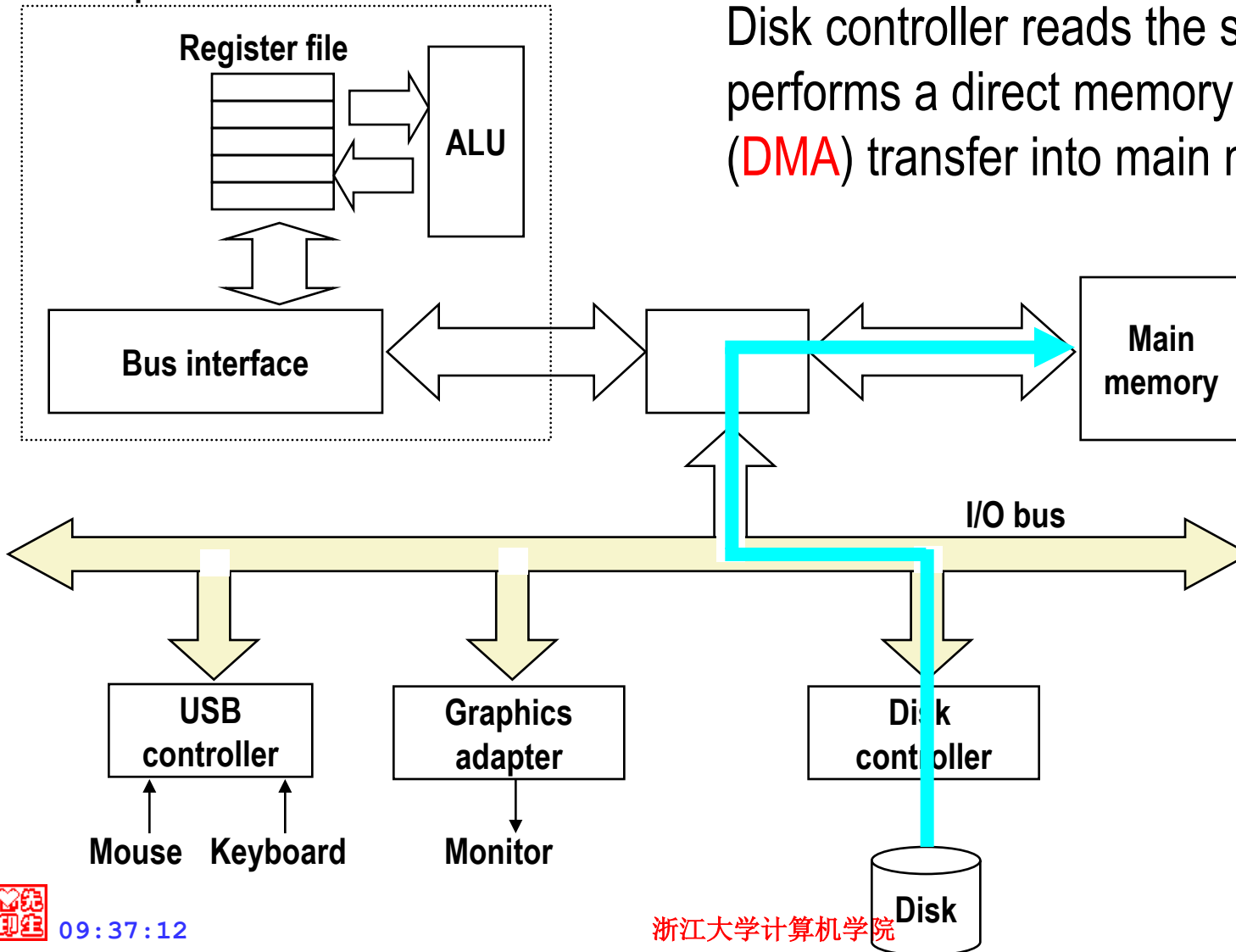
Disk

Reading a Disk Sector (3)

CPU chip

Register file

ALU

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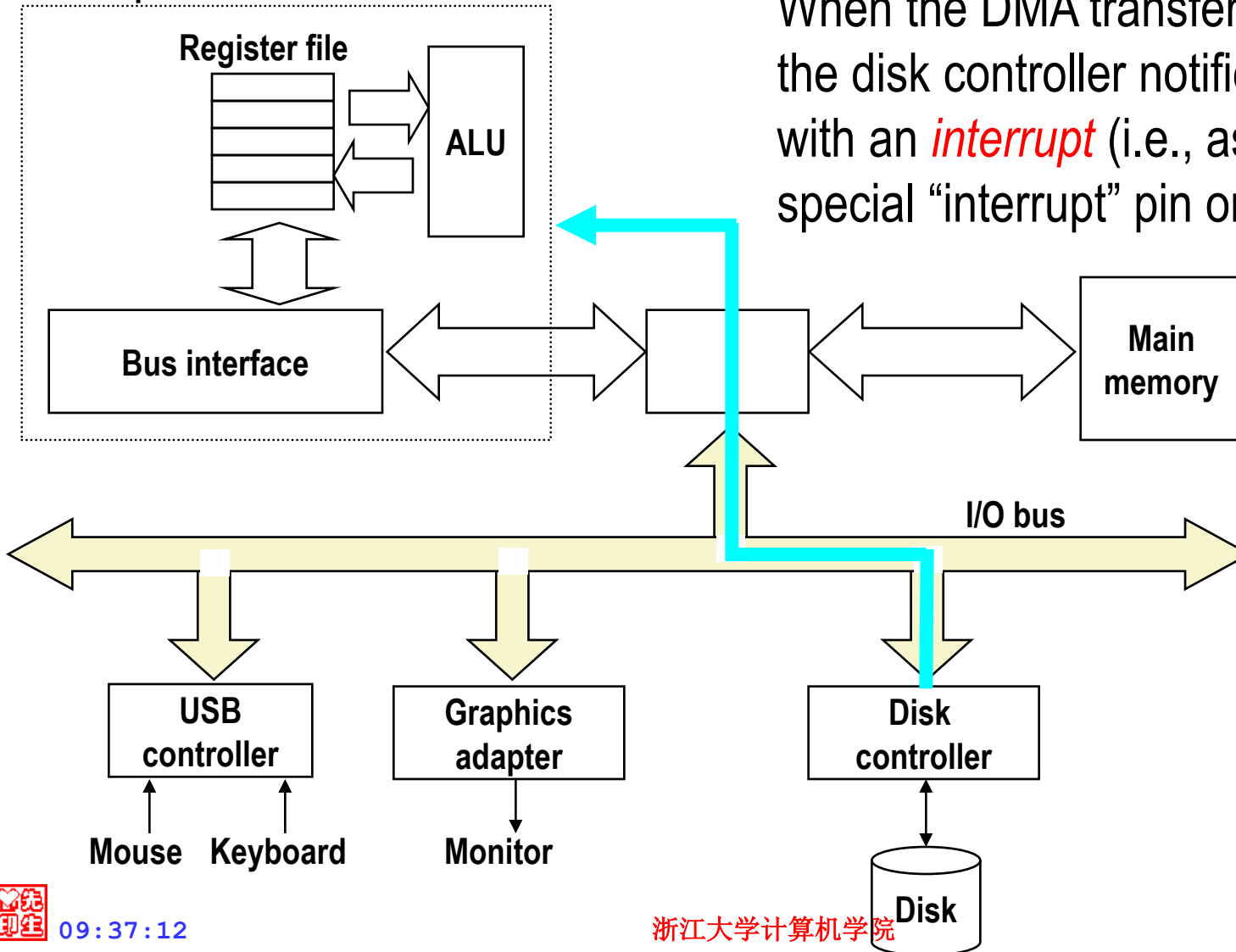
When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Mouse   **Keyboard**

Monitor

Disk

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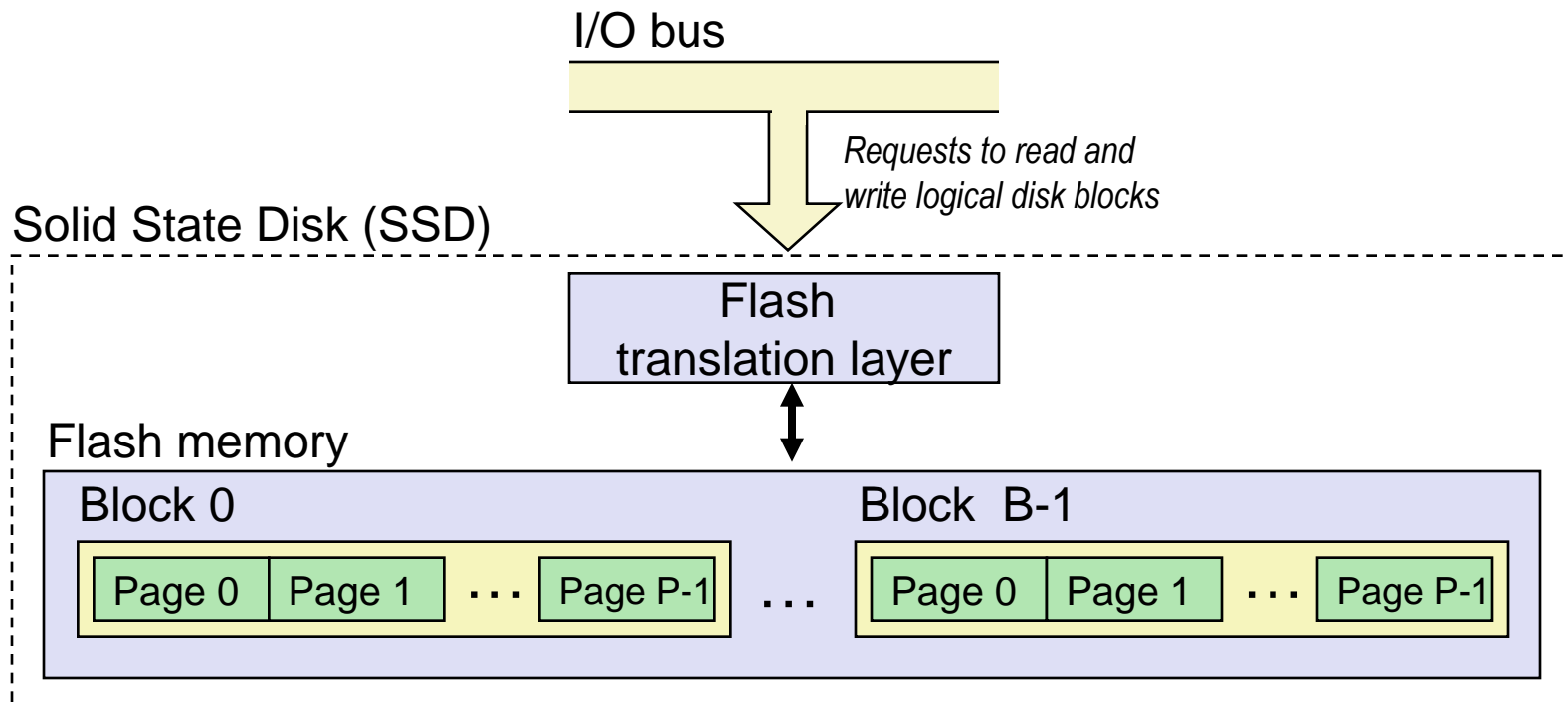
- **Pages: 512KB to 4KB, Blocks: 32 to 128 pages**

- **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 Performance Characteristics

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

- **Sequential access faster than random access**
  - Common theme in the memory hierarchy

- **Random writes are somewhat slower**
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - In earlier SSDs, the read/write gap was much larger.

Source: Intel SSD 730 product specification.

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. Intel SSD 730 guarantees 128 petabyte (128 x 10^{15} bytes) of writes before they wear out
  - In 2015, about 30 times more expensive per byte

- **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.
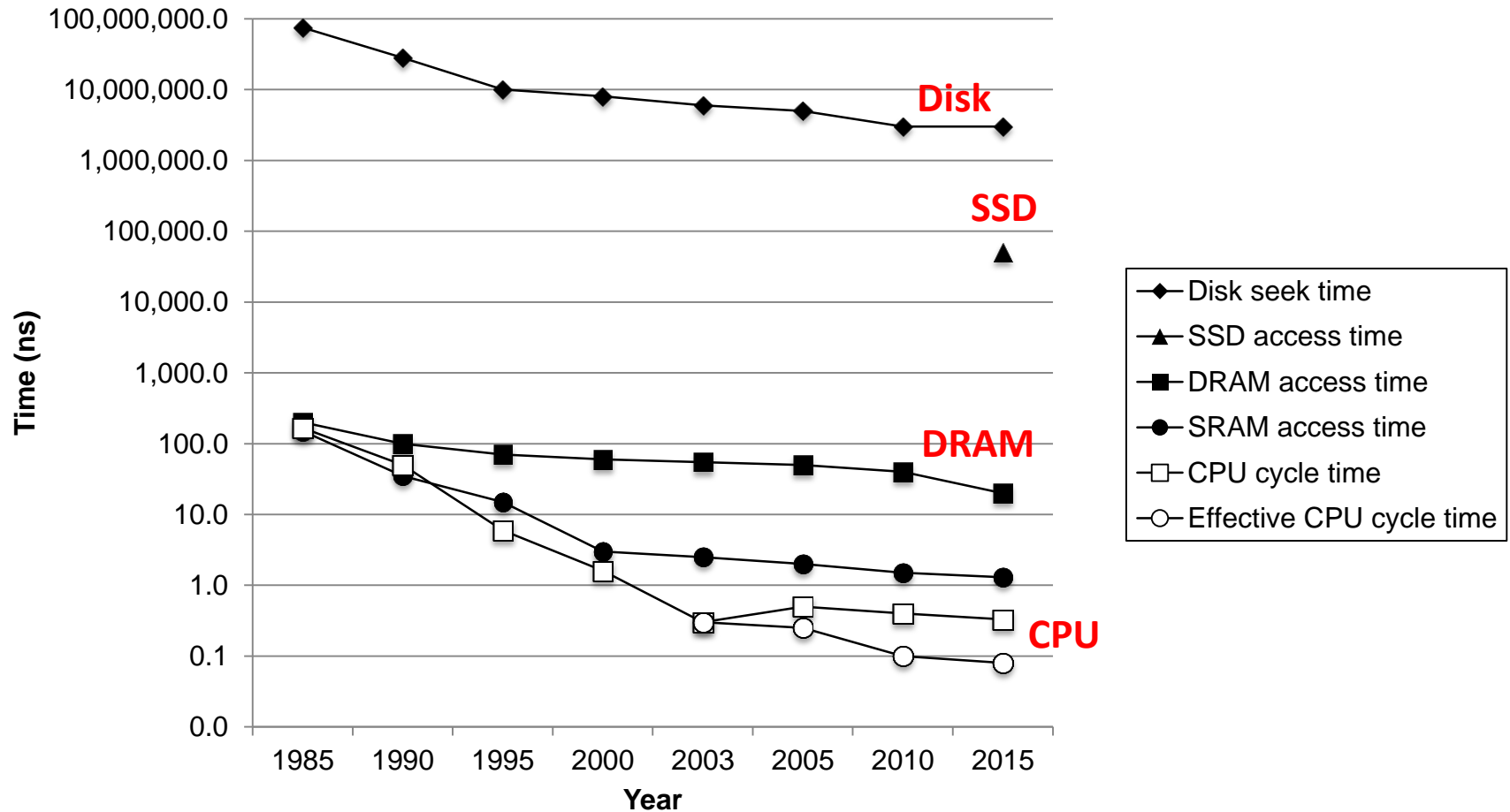
Locality to the Rescue!

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

Today

- **Storage technologies and trends**
- **Locality of reference**
- **Caching in the memory hierarchy**

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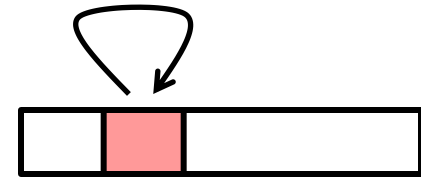
- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time
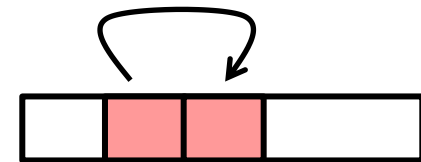
Locality Example

```
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**

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;
}
```

浙江大学计算机学院

Locality Example

- **Question:** 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;
}
```

Locality Example

- **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)?

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

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

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 fundamental properties complement each other beautifully.**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

Today

- **Storage technologies and trends**
- **Locality of reference**
- **Caching in the 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:* **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"
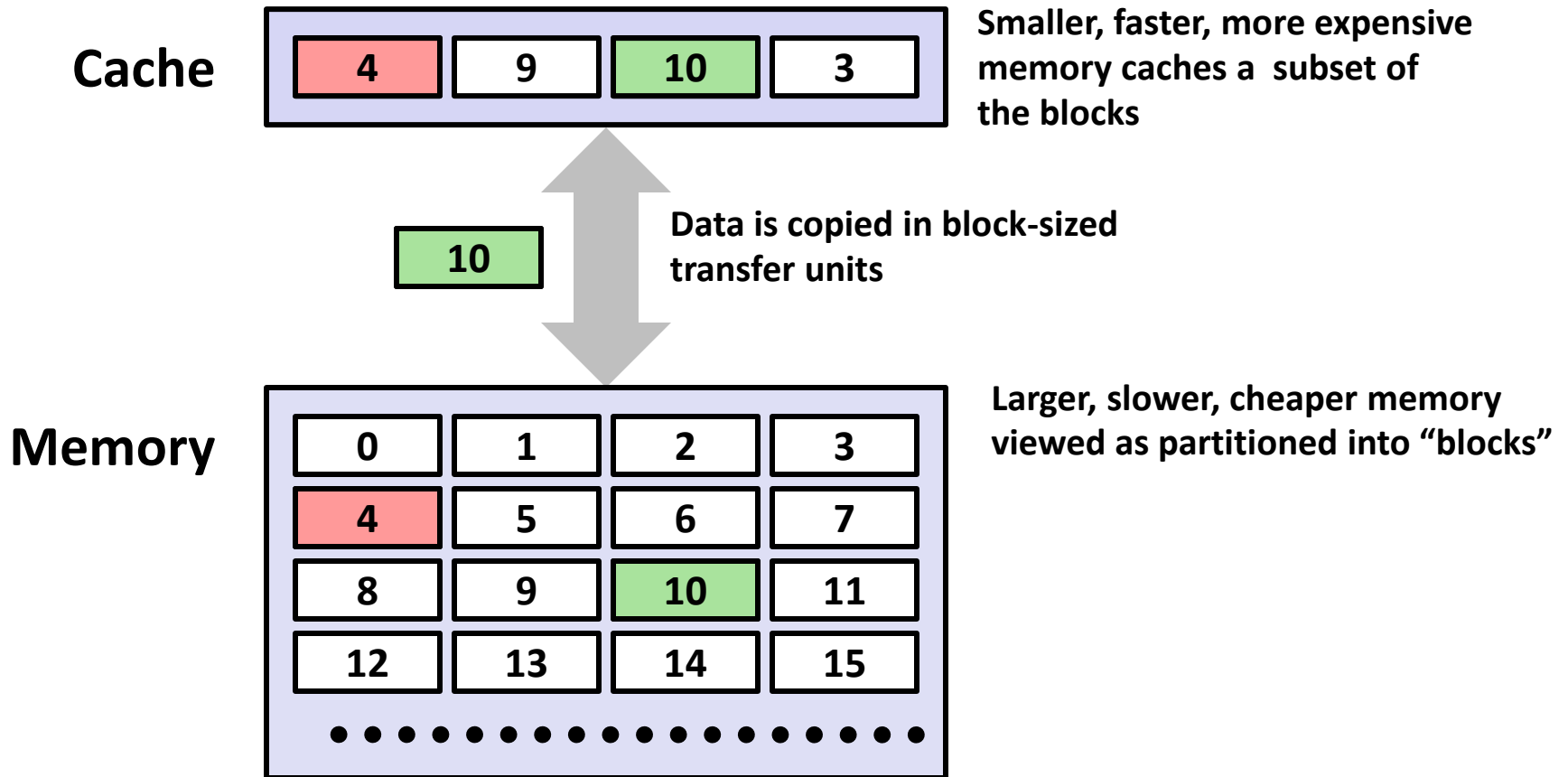
General Cache Concepts: Hit

Request: 14
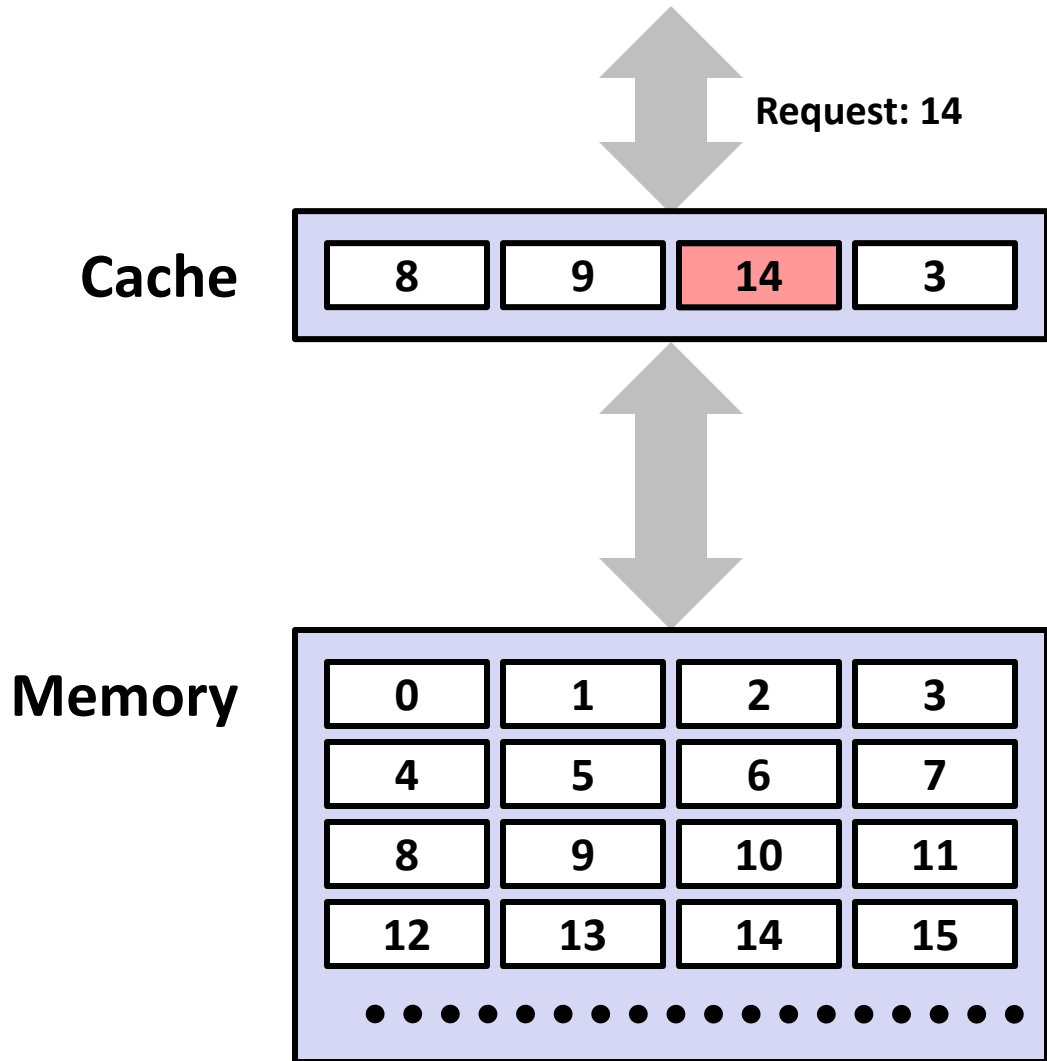
Data in block b is needed

Block b is in cache:
Hit!

Cache

| 8 | 9 | 14 | 3 |

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 |

Request: 12

| 12 |

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
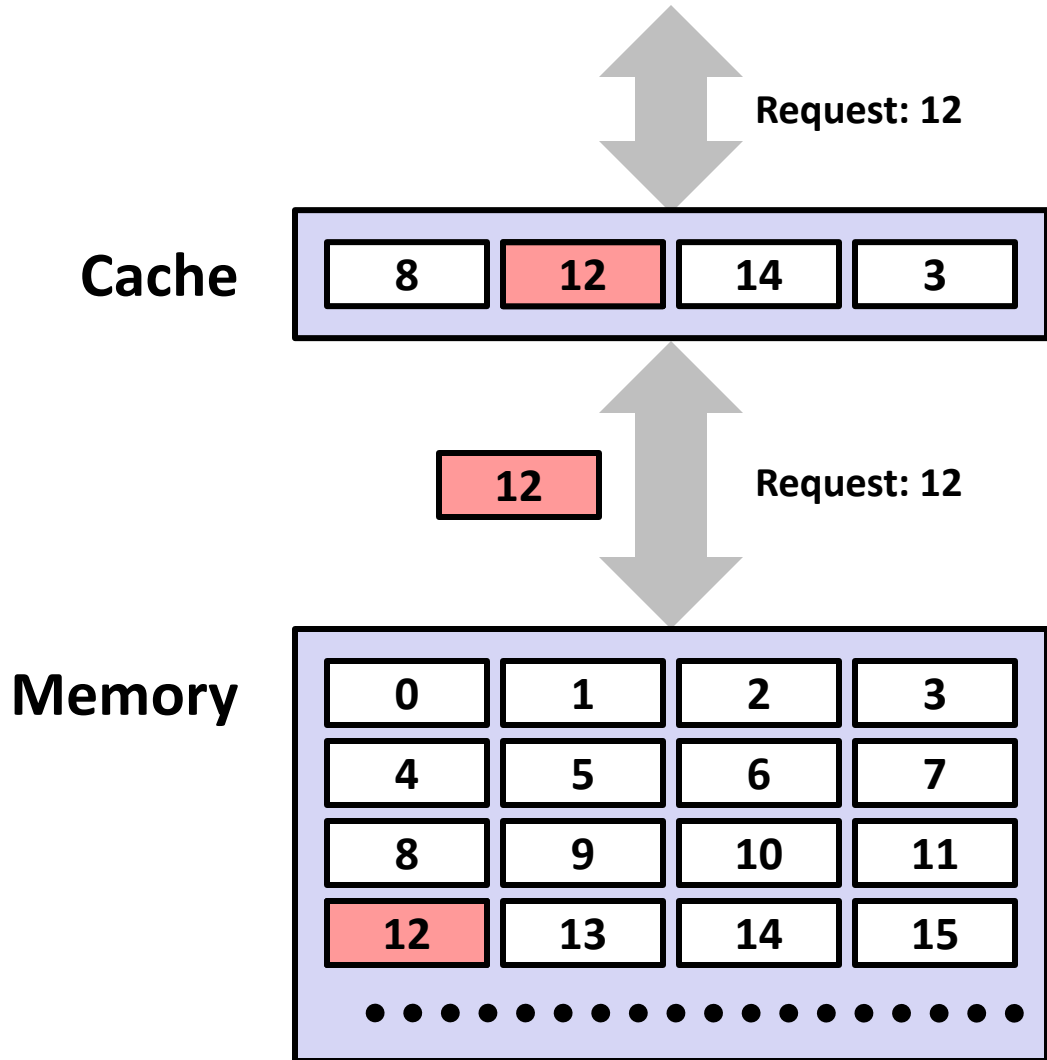
Data in block b is needed

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)

General Caching Concepts: 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.

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

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.**

Supplemental slides

Conventional DRAM Organization

- **d x w DRAM:**
  - dw total bits organized as d supercells of size w bits

16 x 8 DRAM chip

浙江大学计算机学院
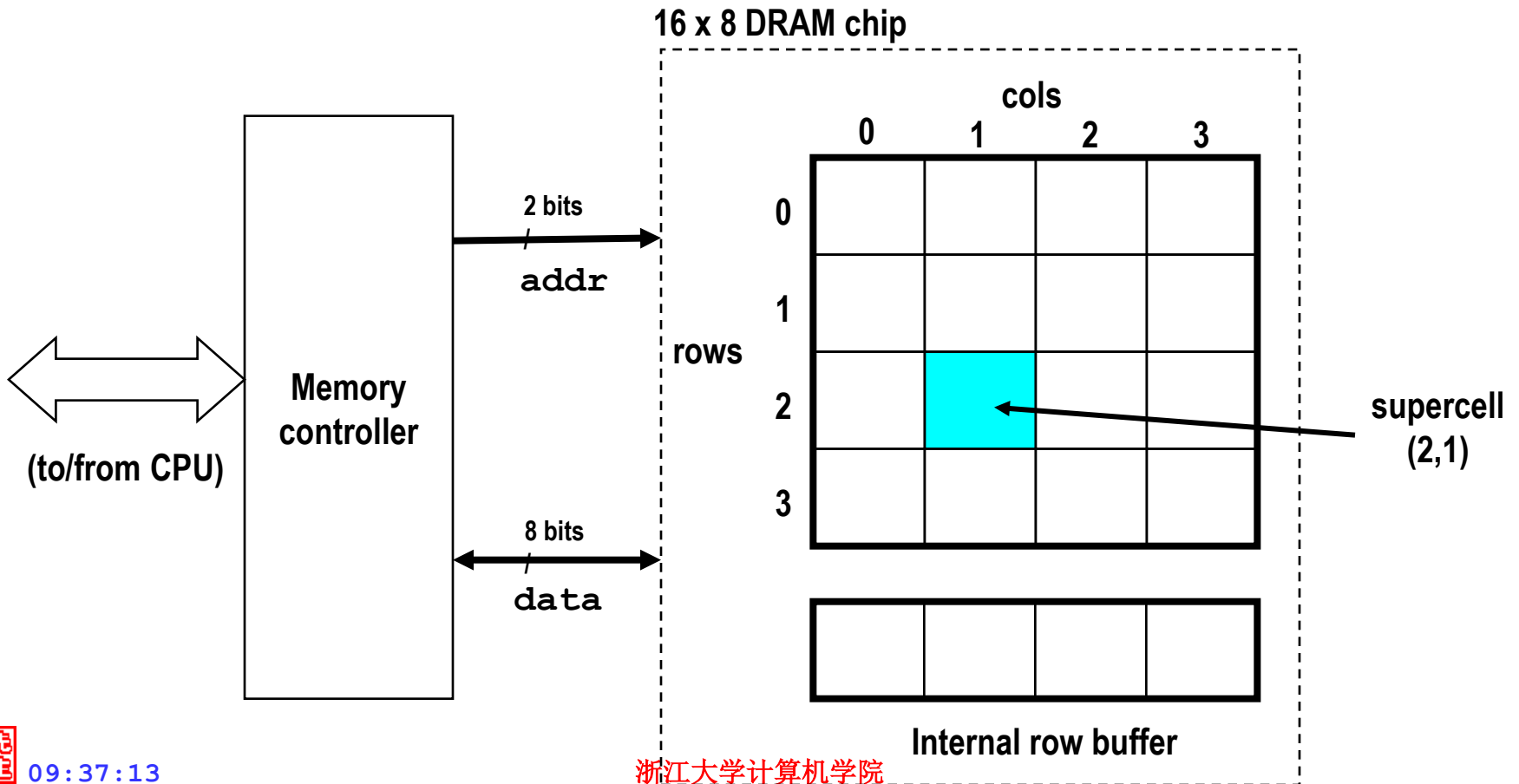
Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.

Step 1(b): Row 2 copied from DRAM array to row buffer.

16 x 8 DRAM chip

Memory controller

RAS = 2

2

addr

8

data

Cols

0 1 2 3

Rows

0

1

2

3

Internal row buffer
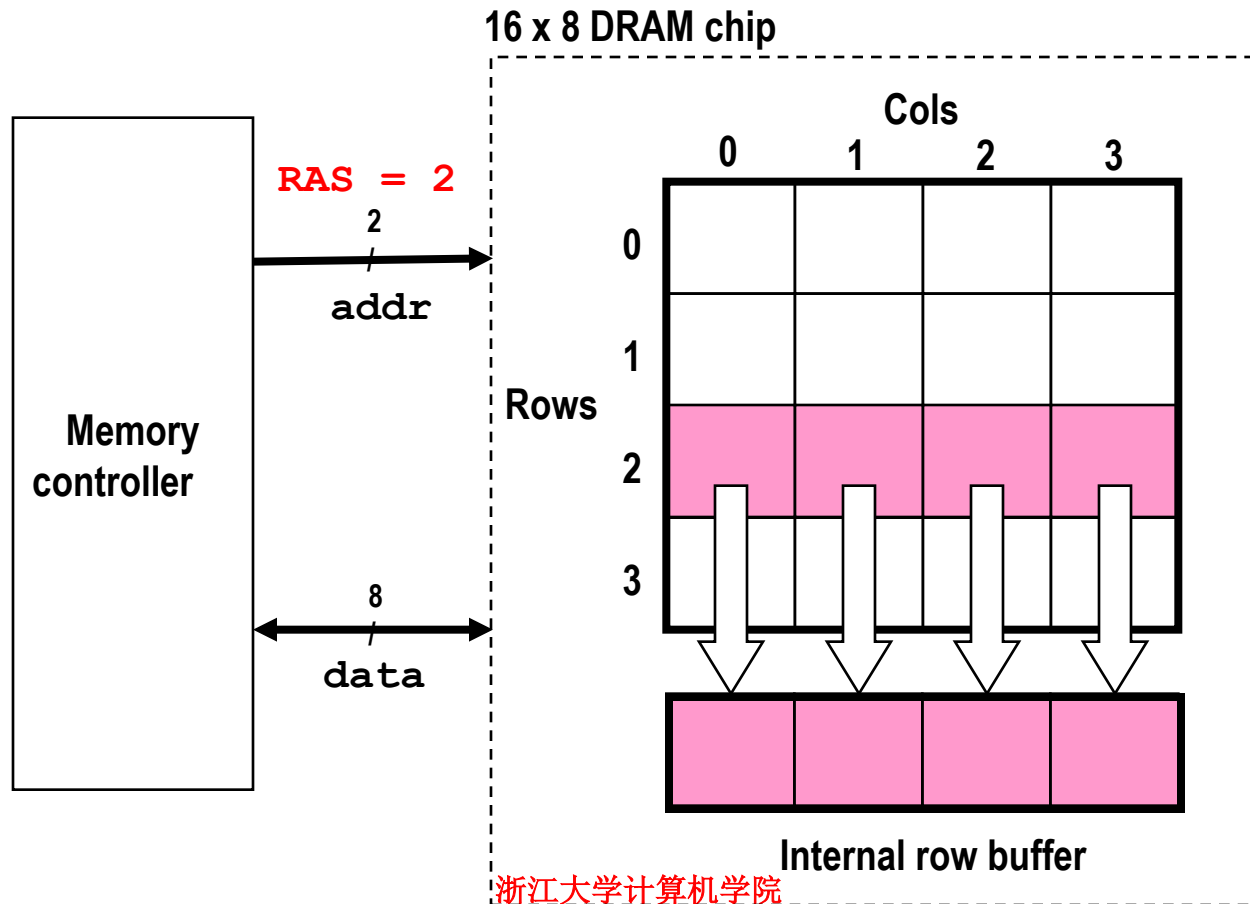
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.

16 x 8 DRAM chip

To CPU

Memory controller

supercell (2,1)

CAS = 1

2

addr

8

data

supercell (2,1)

Cols

0   1   2   3

Rows

0

1

2

3

Internal row buffer
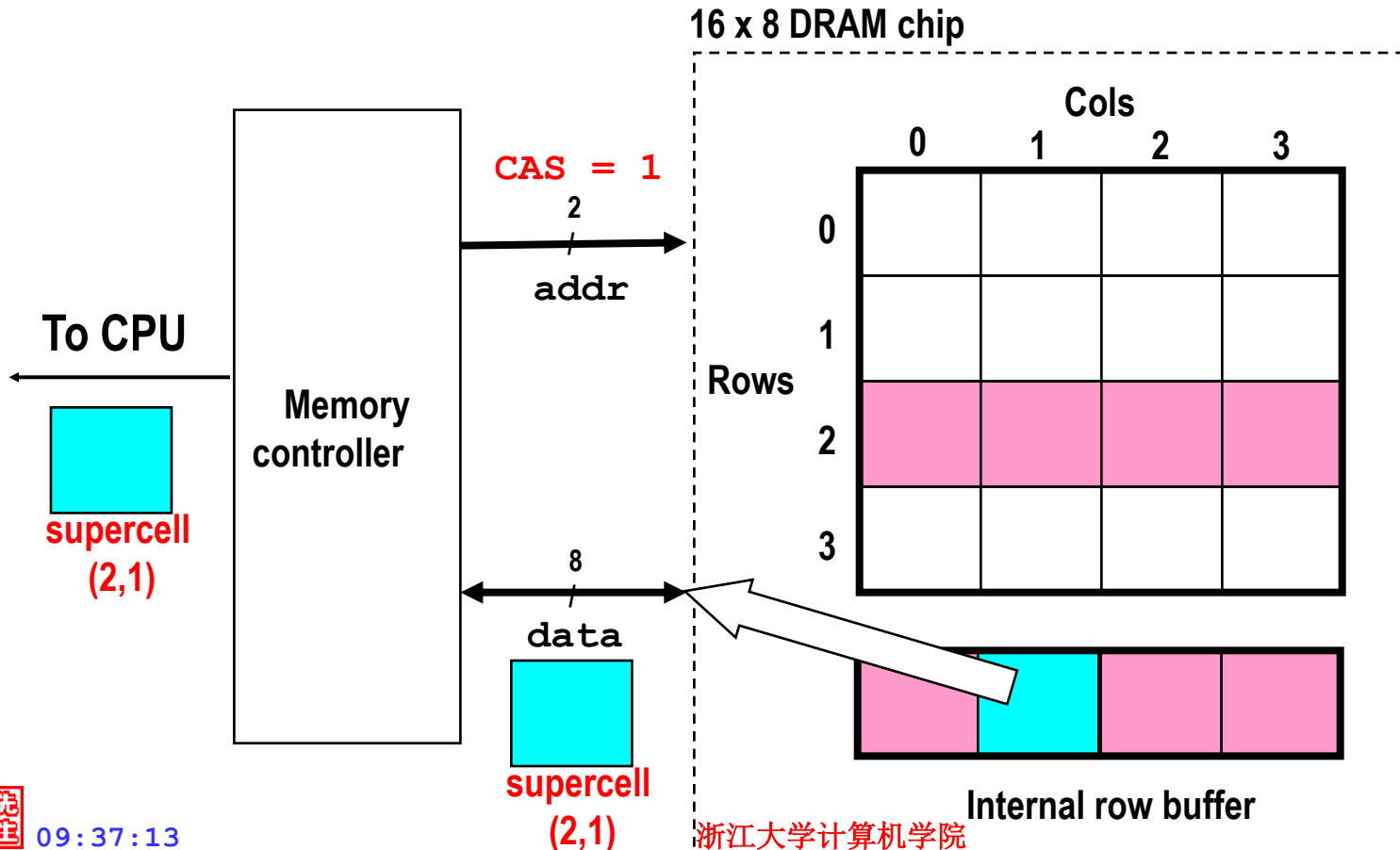
Memory Modules

Enhanced DRAMs

- **Basic DRAM cell has not changed since its invention in 1966.**
  - 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
    - Allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)

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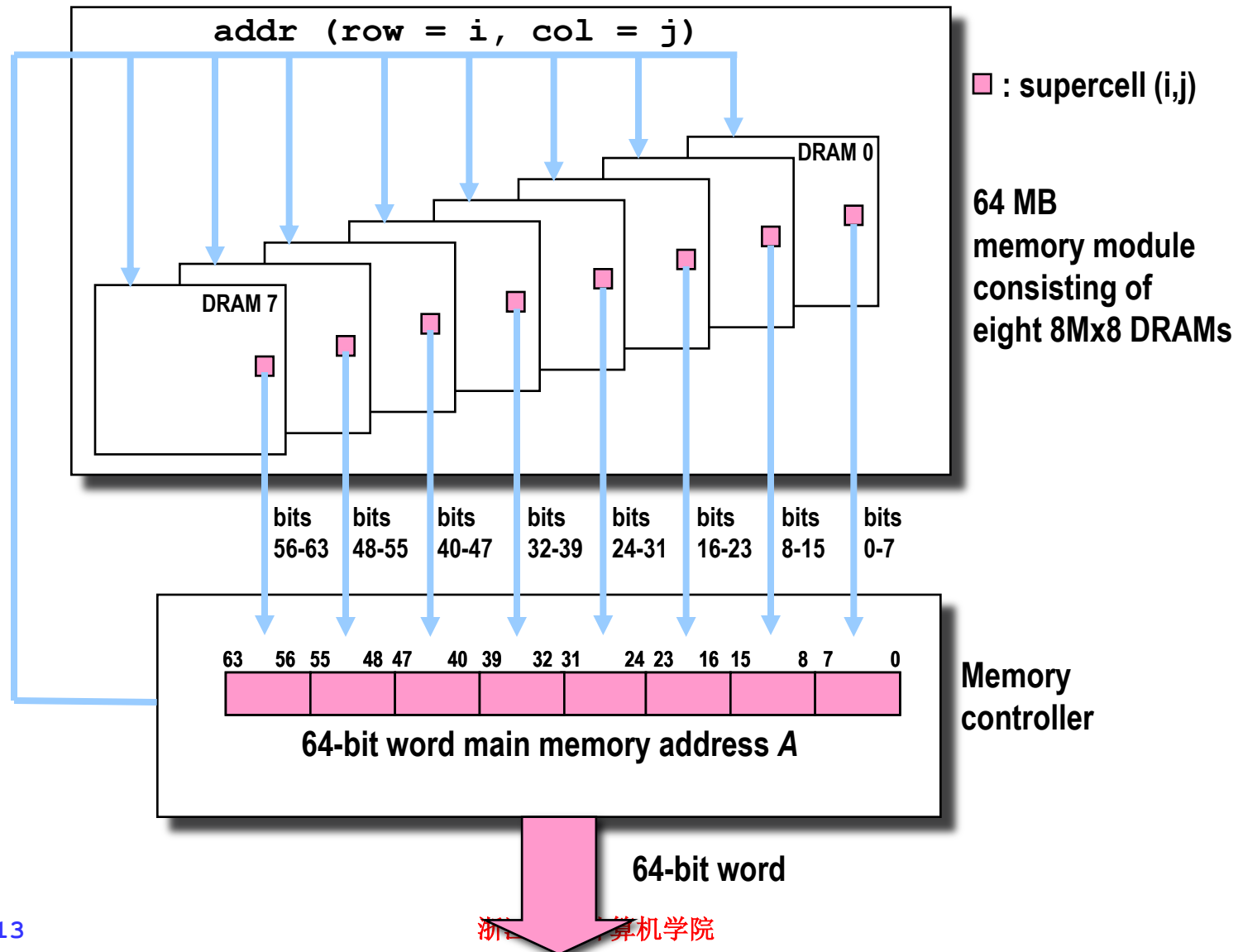   - Different types distinguished by size of small prefetch buffer:
      - DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits)
    - By 2010, standard for most server and desktop systems
    - Intel Core i7 supports only DDR3 SDRAM

Storage Trends

SRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	2,900	320	256	100	75	60	*320*	*116*
access (ns)	150	35	15	3	2	1.5	*200*	*115*

DRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	880	100	30	1	0.1	0.06	0.02	*44,000*
access (ns)	200	100	70	60	50	40	20	*10*
typical size (MB)	0.256	4	16	64	2,000	8,000	16.000	*62,500*

Disk

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/GB	100,000	8,000	300	10	5	0.3	0.03	*3,333,333*
access (ms)	75	28	10	8	5	*3*	*3*	*25*
typical size (GB)	0.01	0.16	1	20	160	1,500	3,000	*300,000*

CPU Clock Rates

Inflection point in computer history when designers hit the "Power Wall"

	1985	1990	1995	2003	2005	2010	2015	*2015:1985*
CPU	80286	80386	Pentium	P-4	Core 2	Core i7(n)	Core i7(h)	
Clock rate (MHz)	6	20	150	3,300	2,000	2,500	3,000	500
Cycle time (ns)	166	50	6	0.30	0.50	0.4	0.33	500
Cores	1	1	1	1	2	4	4	4
Effective cycle time (ns)	166	50	6	0.30	0.25	0.10	0.08	2,075

(n) Nehalem processor
(h) Haswell processor

Cache Memories

15-213: Introduction to Computer Systems
12th Lecture, Oct. 8, 2015

Instructors:

Randal E. Bryant and David R. O'Hallaron

Today

- **Cache memory organization and operation**
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

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

General Cache Concept

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

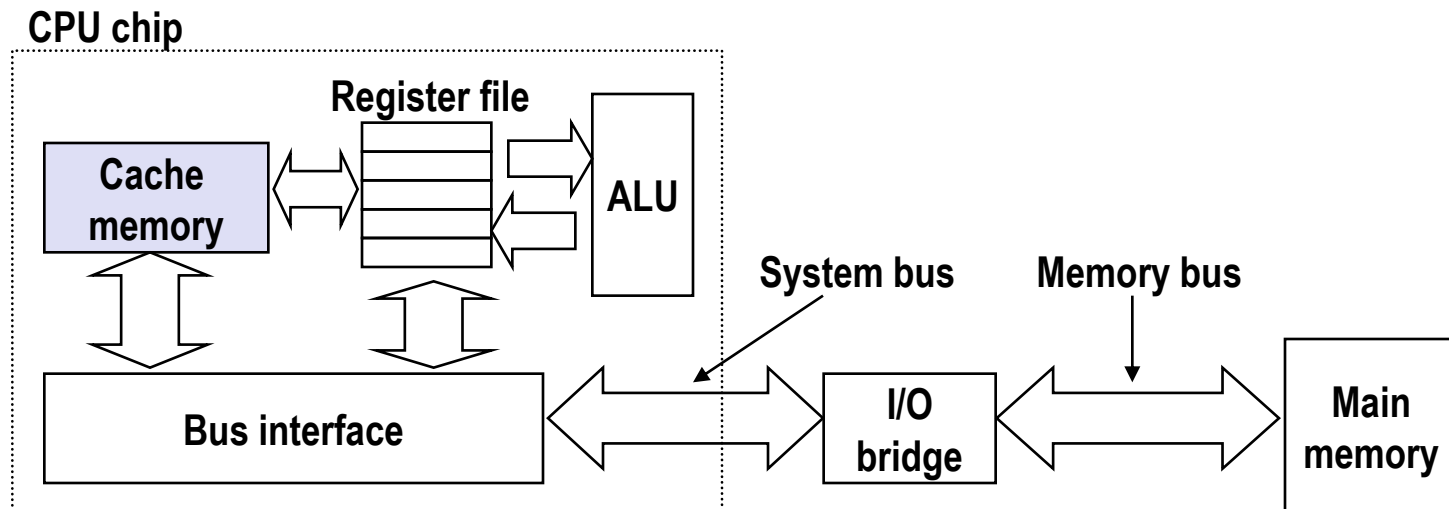| **0**	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

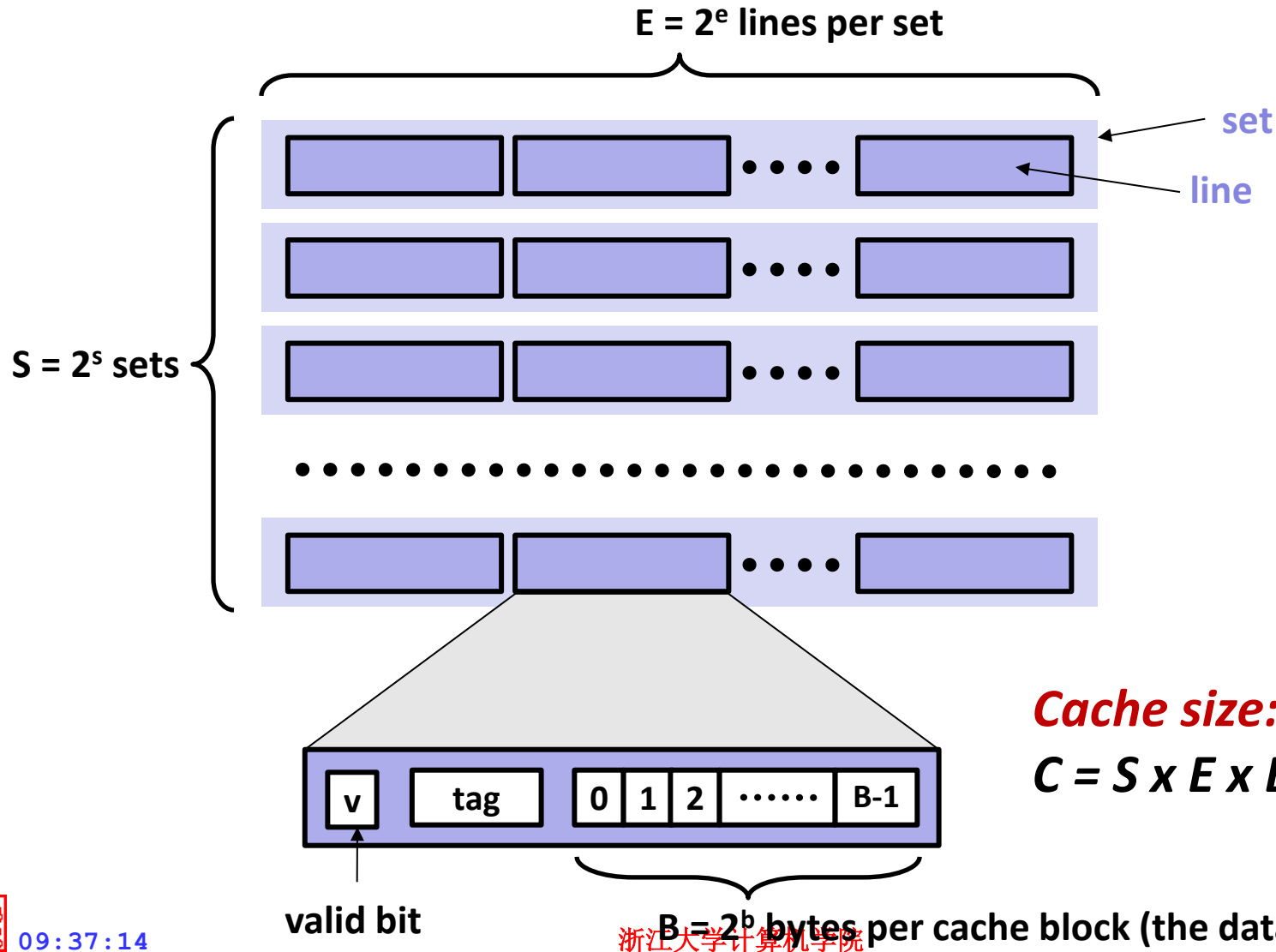Larger, slower, cheaper memory viewed as partitioned into "blocks"
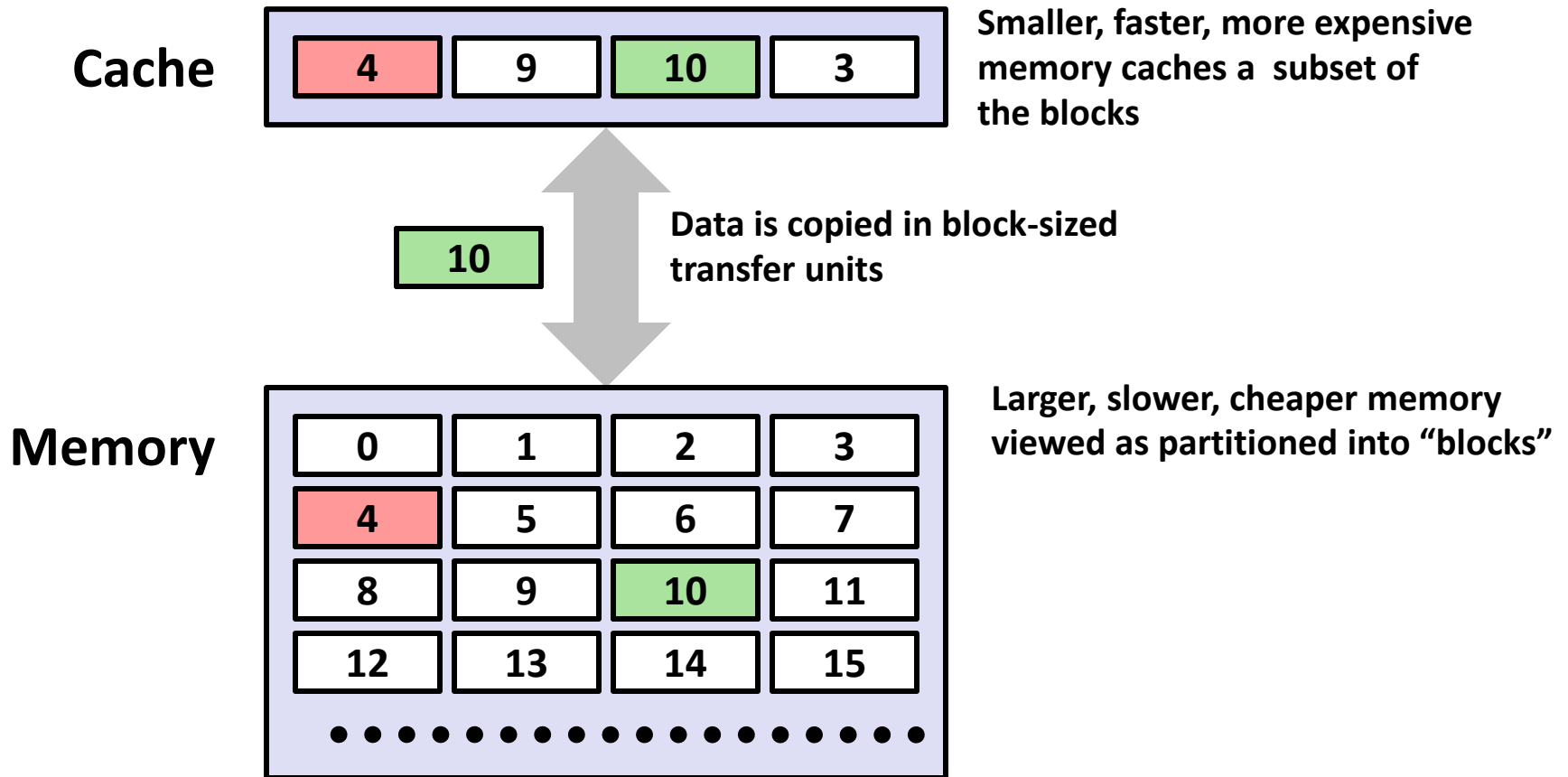
Cache Memories

- **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**
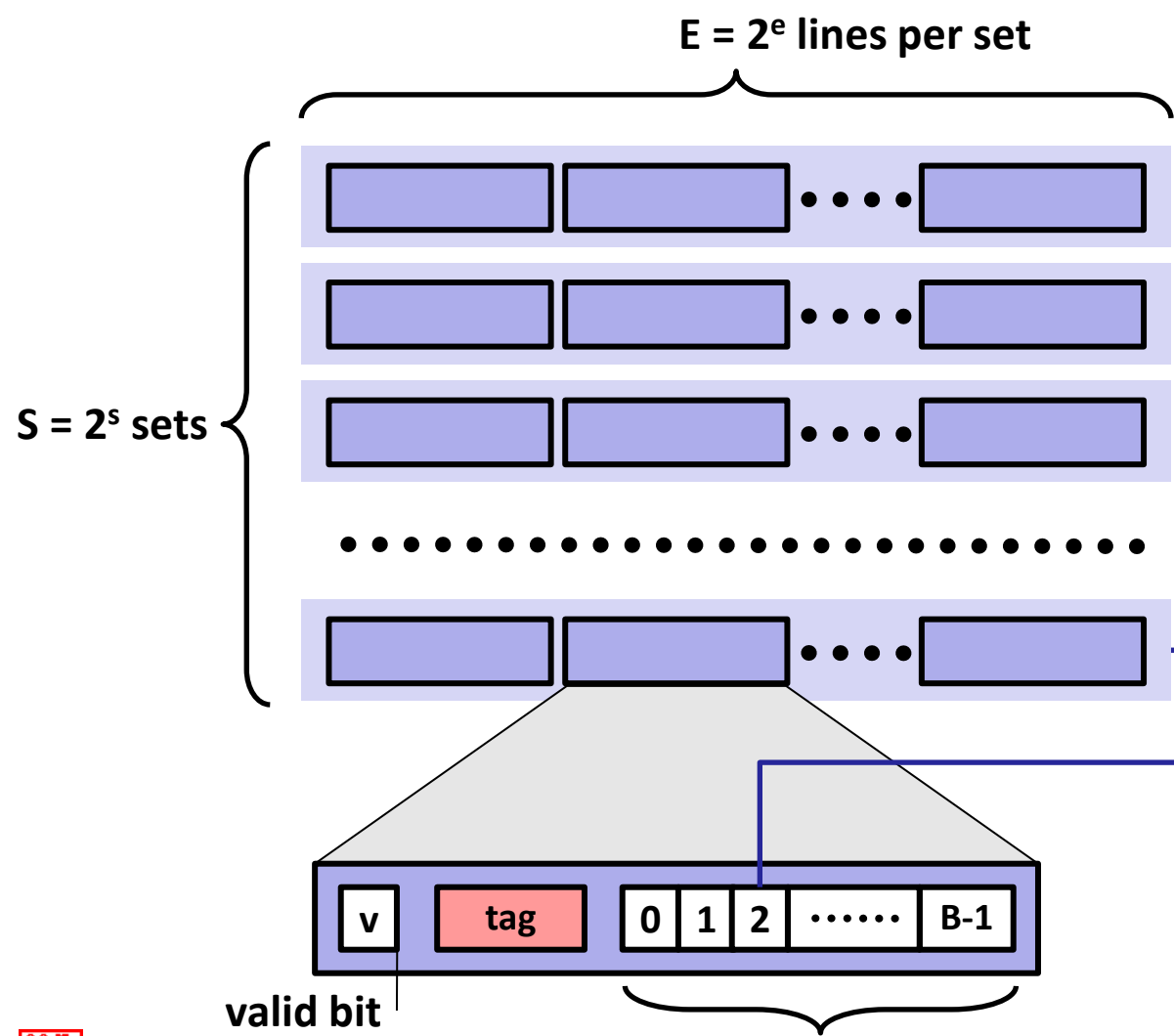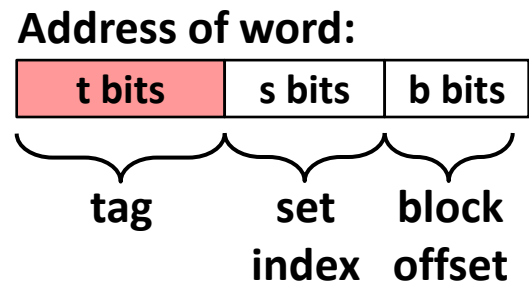- **Typical system structure:**

General Cache Organization (S, E, B)

$E = 2^e$ lines per set

set

line

$S = 2^s$ sets

valid bit

v | tag | 0 | 1 | 2 | ······ | B-1

$B = 2^b$ bytes per cache block (the data)

Cache size:

C = S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

E = 2^e lines per set

S = 2^s sets

Address of word:

t bits	s bits	b bits

tag   set index   block offset

data begins at this offset

v	tag	0	1	2	......	B-1

valid bit

B = 2^b bytes per cache block (the data)

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

S = 2^s sets

Address of int:

t bits	0...01	100

find set

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

valid? + match: assume yes = hit

Address of int:

t bits	0...01	100

v	tag	0	1	2	3	4	5	6	7

block offset

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

valid? + **match: assume yes = hit**

Address of int:

t bits	0...01	100

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

int (4 Bytes) is here
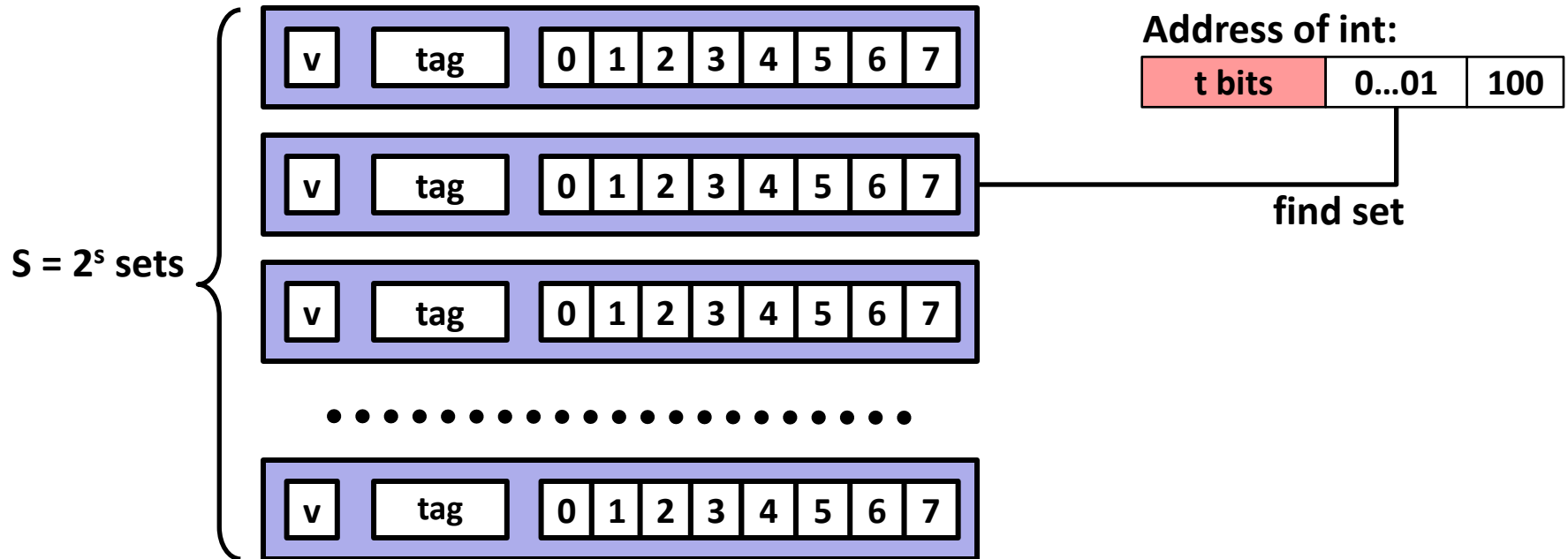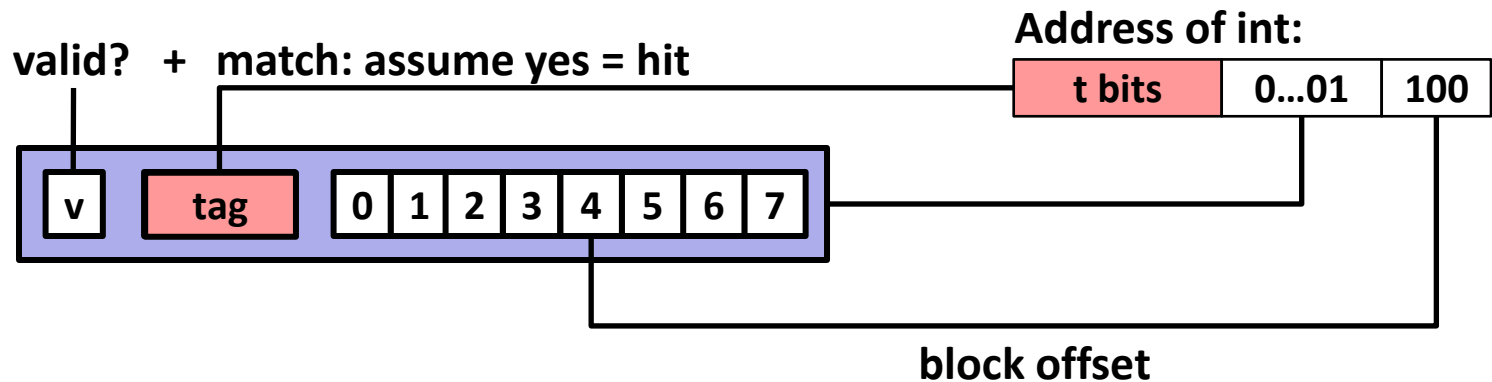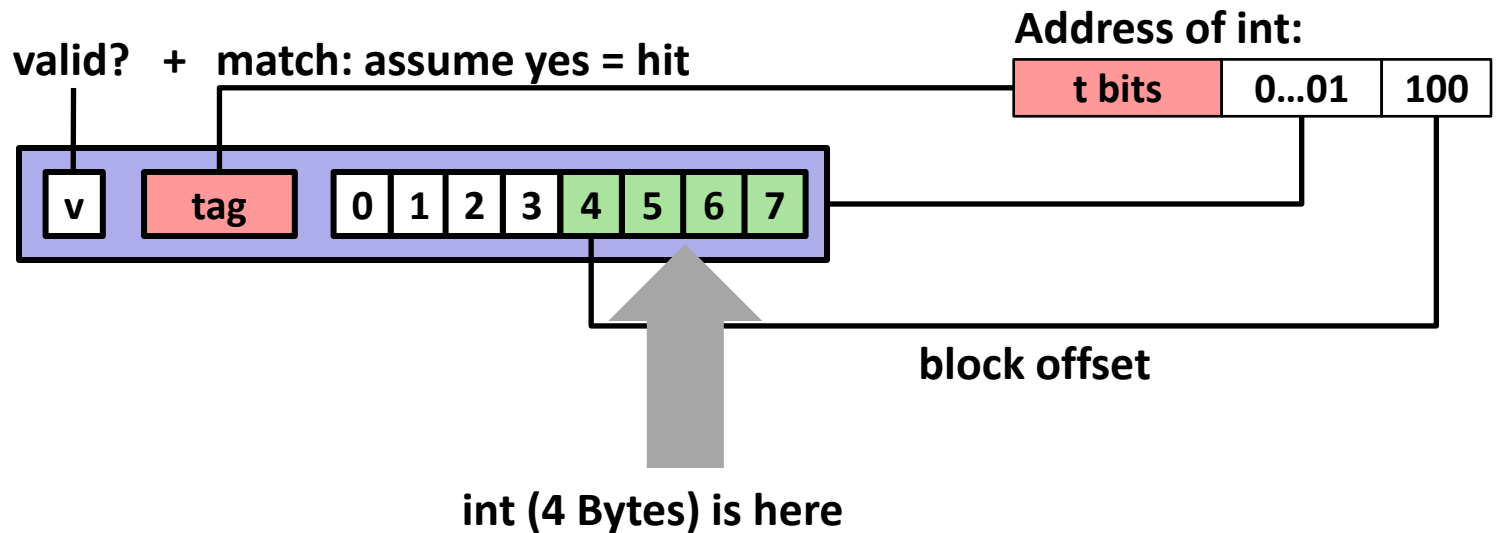
If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block, S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	[0$\underline{00}$0$_2$],	miss
1	[0$\underline{00}$1$_2$],	hit
7	[0$\underline{11}$1$_2$],	miss
8	[1$\underline{00}$0$_2$],	miss
0	[0$\underline{00}$0$_2$]	miss

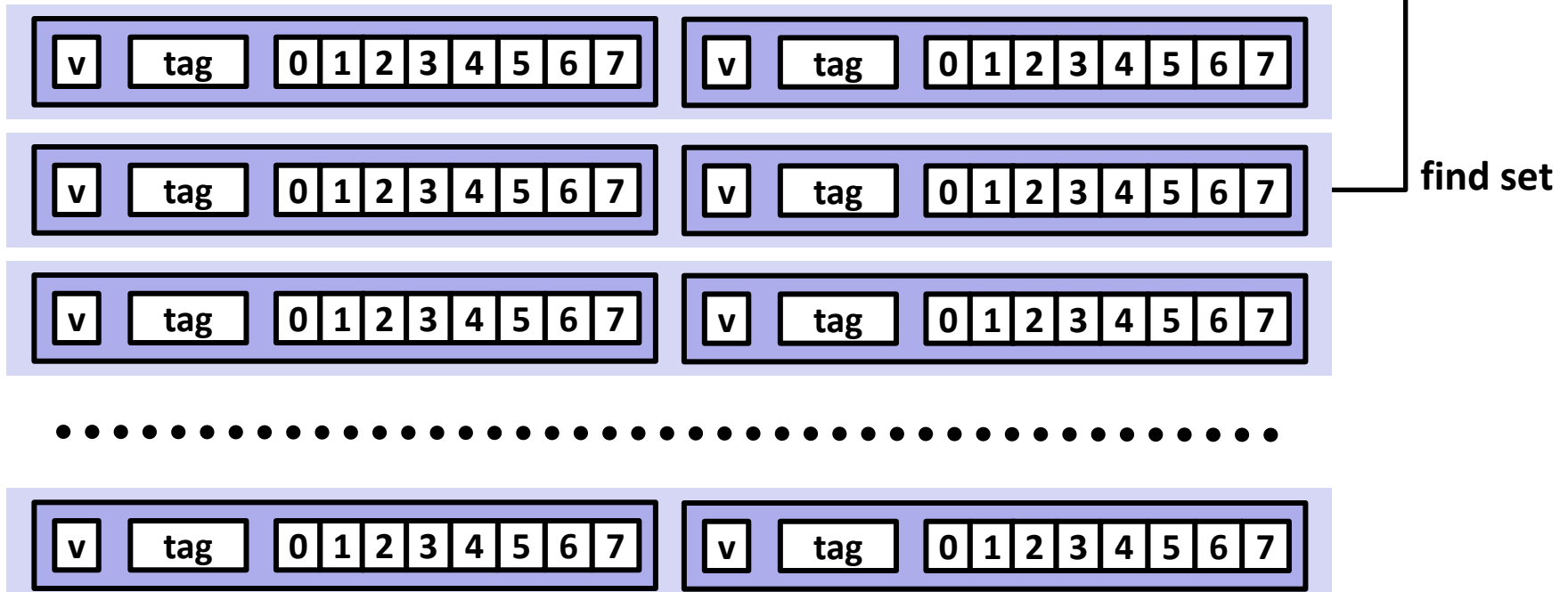|	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

浙江大学计算机学院

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
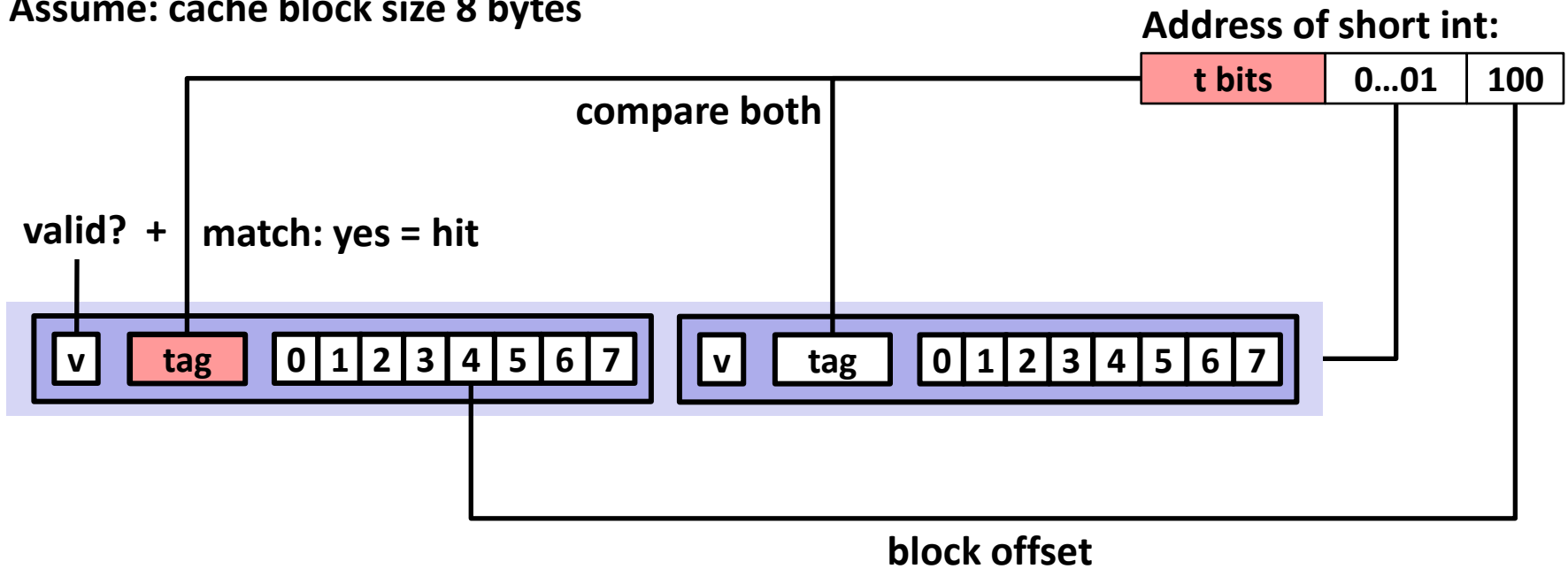**Assume: cache block size 8 bytes**

Address of short int:

t bits	0...01	100

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | **find set**

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

• •

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

| t bits | 0...01 | 100 |

compare both

valid? + match: yes = hit

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

t bits	0...01	100

compare both

valid? + **match: yes = hit**

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

short int (2 Bytes) is here
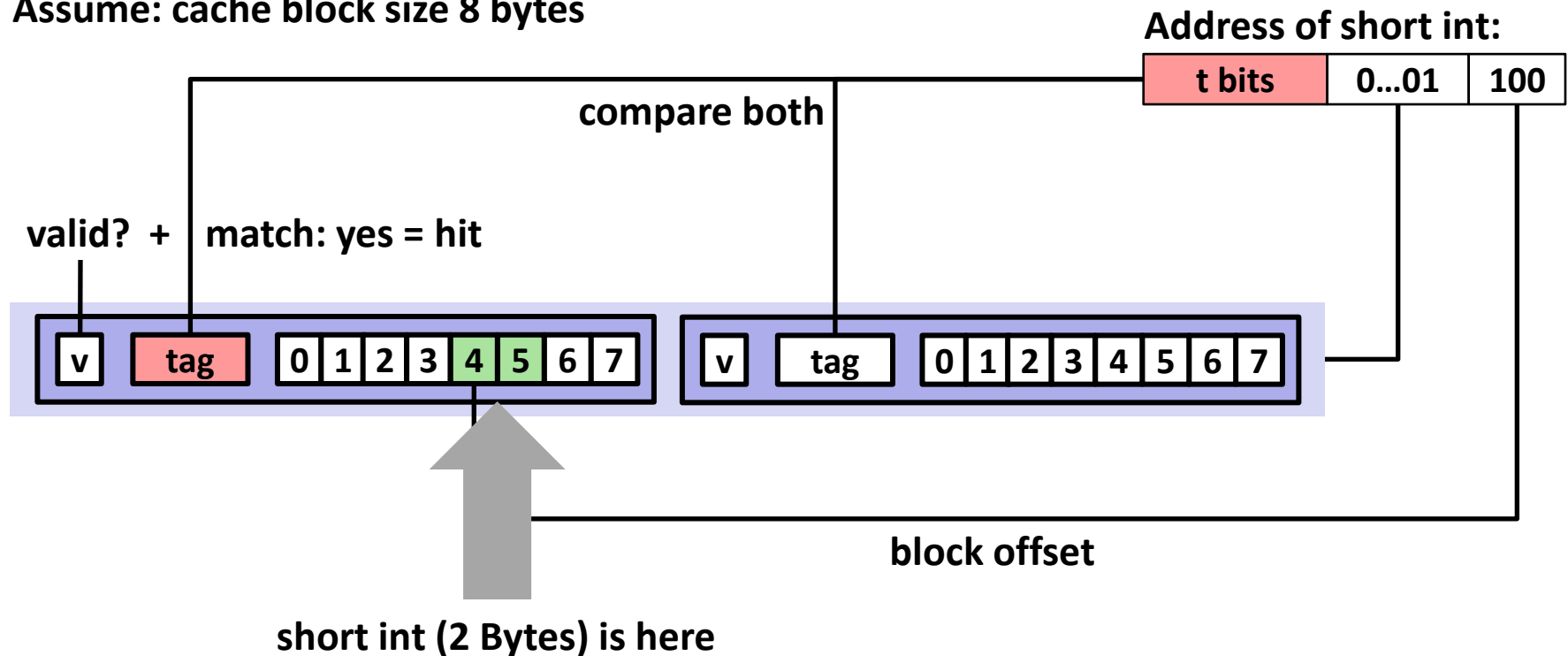
No match:

- **One line in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU), ...**

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	miss
0	$[00\underline{0}0_2]$	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk

- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)

- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
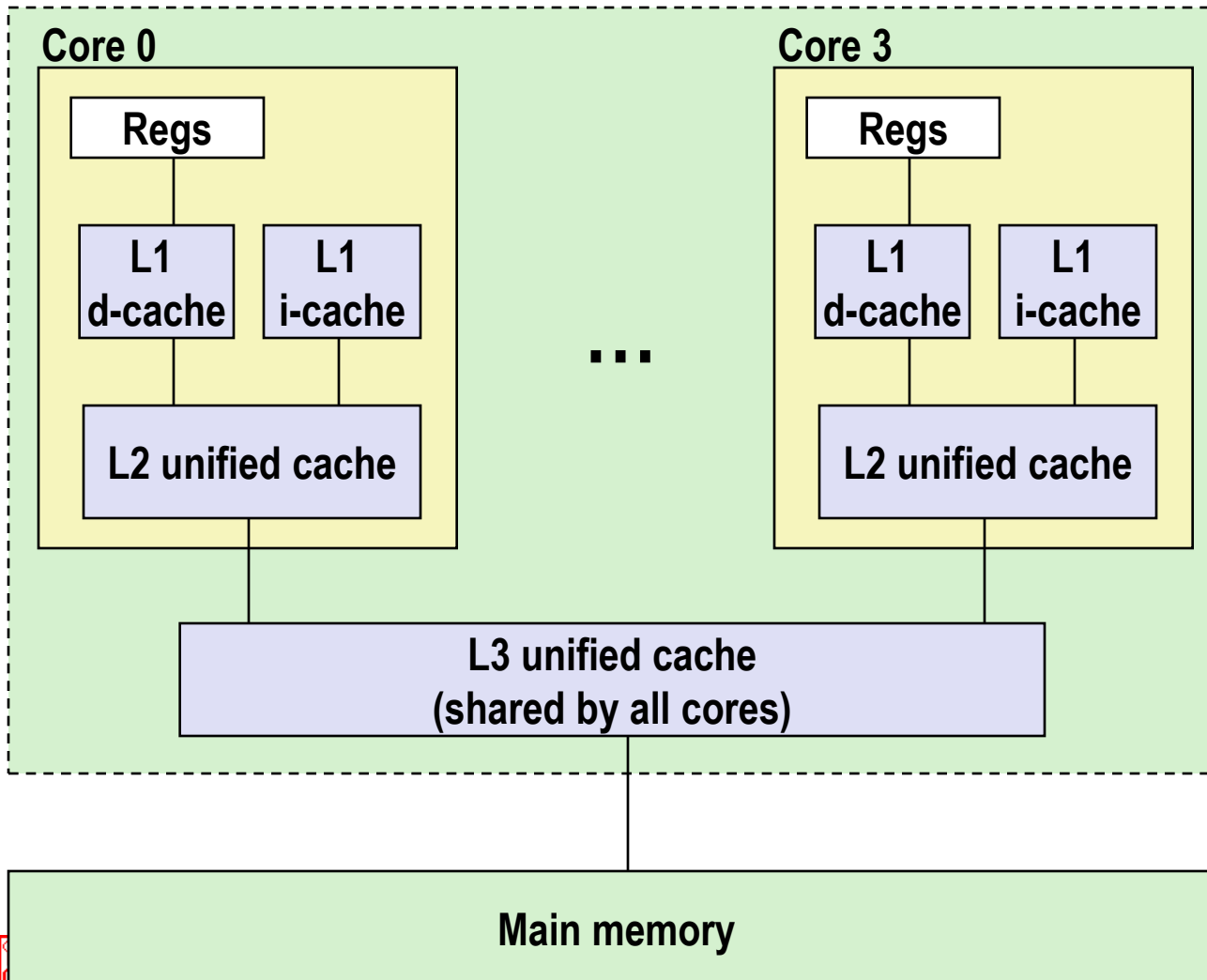 - No-write-allocate (writes straight to memory, does not load into cache)

- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

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.

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Let's think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- **This is why "miss rate" is used instead of "hit rate"**

Writing Cache Friendly Code

- **Make the common case go fast**
  - Focus on the inner loops of the core functions

- **Minimize the misses in the inner loops**
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

Today

- **Cache organization and operation**

- **Performance impact of caches**
    - The memory mountain
    - Rearranging loops to improve spatial locality
    - Using blocking to improve temporal locality

The Memory Mountain

- **Read throughput** (read bandwidth)
    - Number of bytes read from memory per second (MB/s)

- **Memory mountain: Measured read throughput as a function of spatial and temporal locality.**
    - Compact way to characterize memory system performance.

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

For each elems and stride:

1. Call test() once to warm up the caches.

2. Call test() again and measure the read throughput(MB/s)

The Memory Mountain

Aggressive prefetching

Ridges of temporal locality

L1

L2

L3

Mem
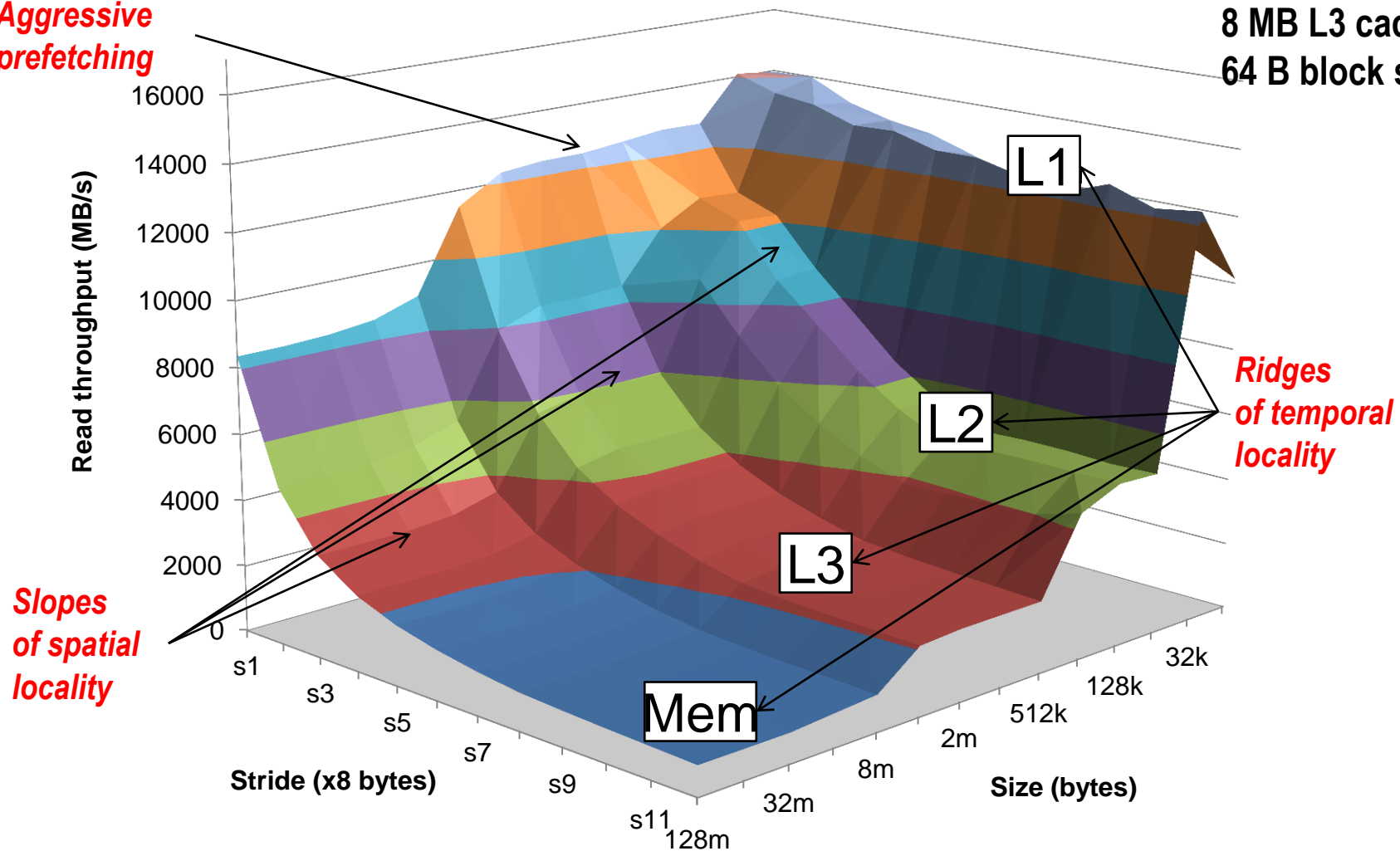
Slopes of spatial locality

Read throughput (MB/s)

16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes)

s1
s3
s5
s7
s9
s11

Size (bytes)

32k
128k
512k
2m
8m
32m
128m

Today

- **Cache organization and operation**
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

Variable sum held in register
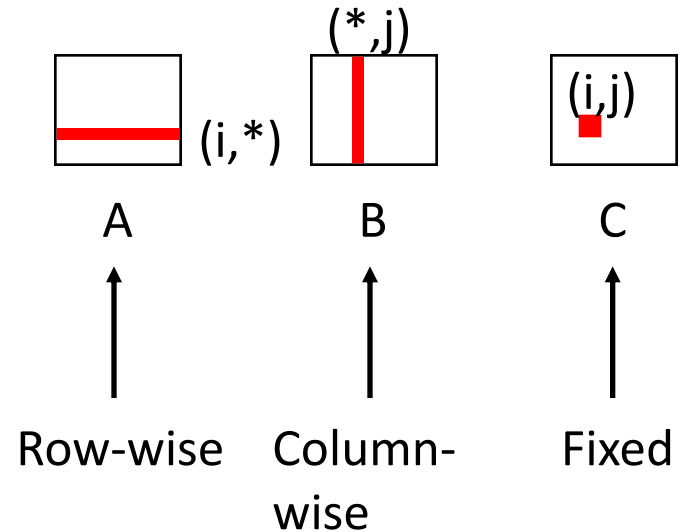
```
/* ijk */
for (i=0; i<n; i++)    {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
                        matmult/mm.c
```

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows

- **Analysis Method:**
  - Look at access pattern of inner loop
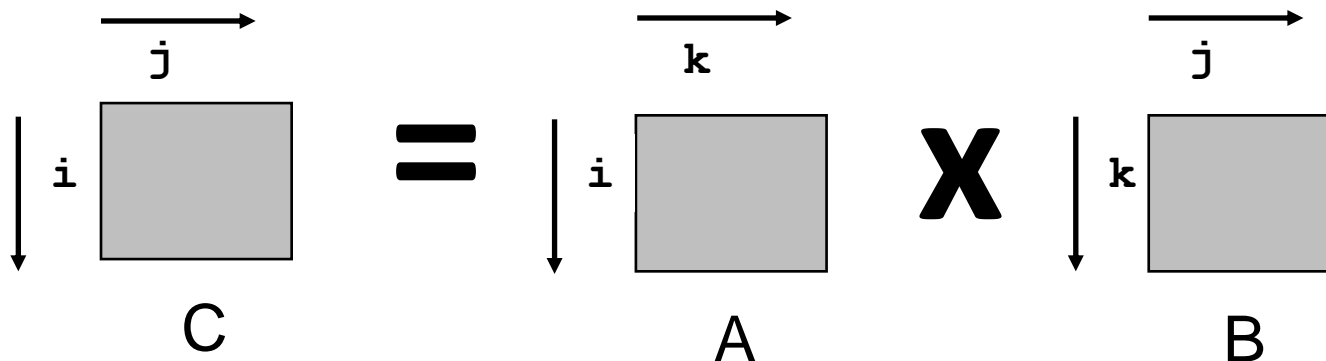
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}                        matmult/mm.c
```
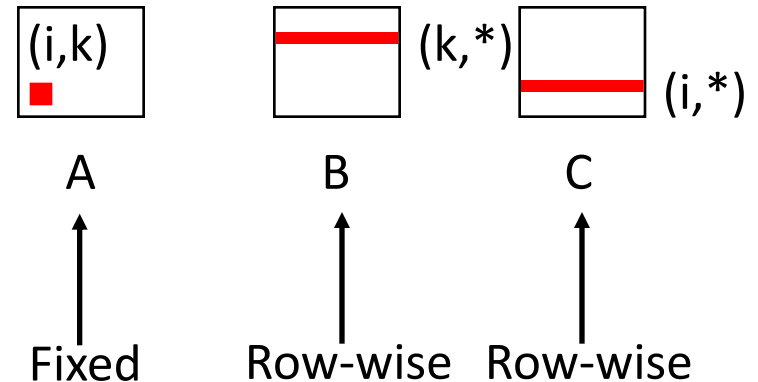
Inner loop:

Row-wise   Column-wise   Fixed

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}                                matmult/mm.c
```

Inner loop:

A — Row-wise
B — Column-wise
C — Fixed

Misses per inner loop iteration:

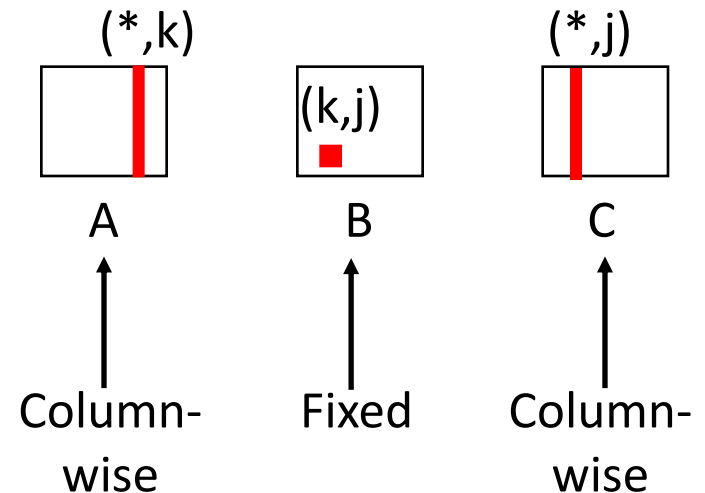| A	B	C
0.25	1.0	0.0
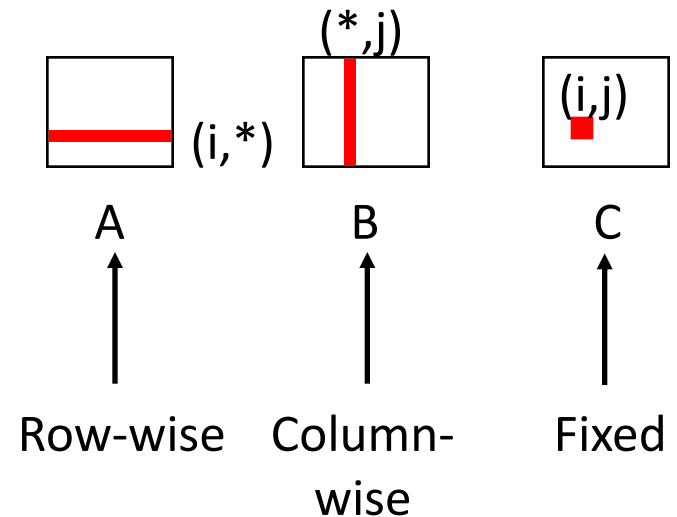
Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
                        matmult/mm.c
```

Inner loop:

(i,k)

A

Fixed

(k,*)

B

Row-wise

(i,*)

C

Row-wise

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
                          matmult/mm.c
```

Inner loop:

A	B	C
(i,k)	(k,*)	(i,*)
Fixed	Row-wise	Row-wise

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25
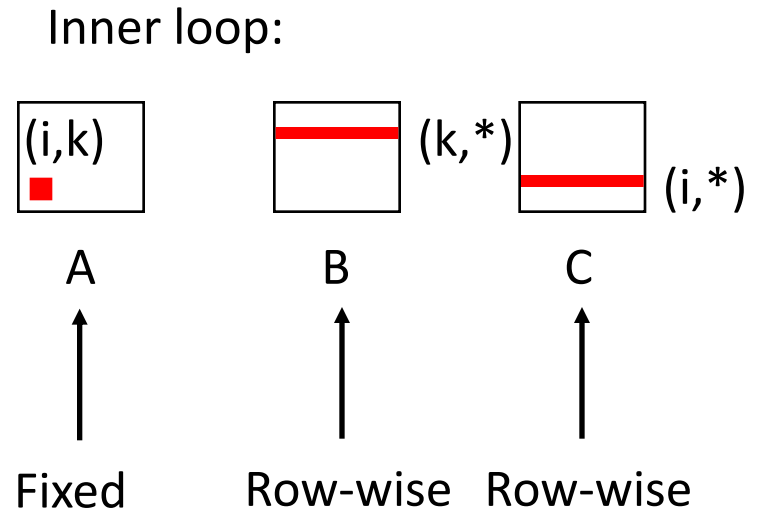
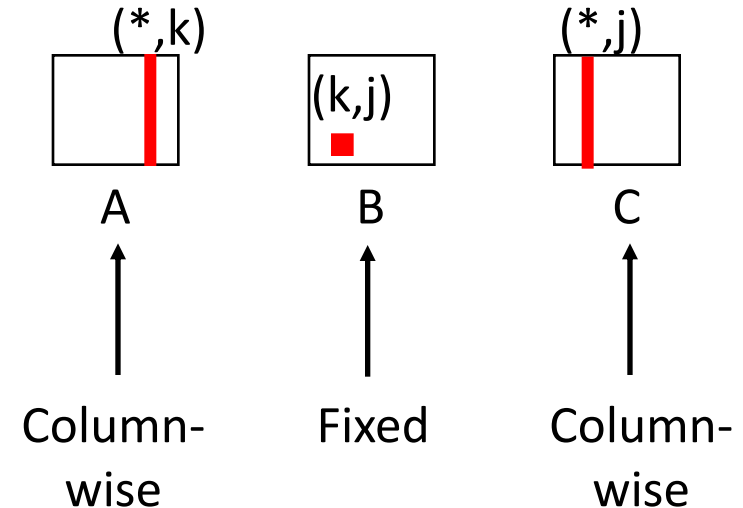# Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
                        matmult/mm.c
```

Inner loop:

```
        (*,k)                      (*,j)
      ┌──────┐      ┌──────┐     ┌──────┐
      │    █ │      │(k,j) │     │  █   │
      │    █ │      │ ■    │     │  █   │
      └──────┘      └──────┘     └──────┘
         A             B            C

      Column-         Fixed      Column-
      wise                        wise
```

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
                              matmult/mm.c
```

Inner loop:

A — Column-wise

B — Fixed

C — Column-wise

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance
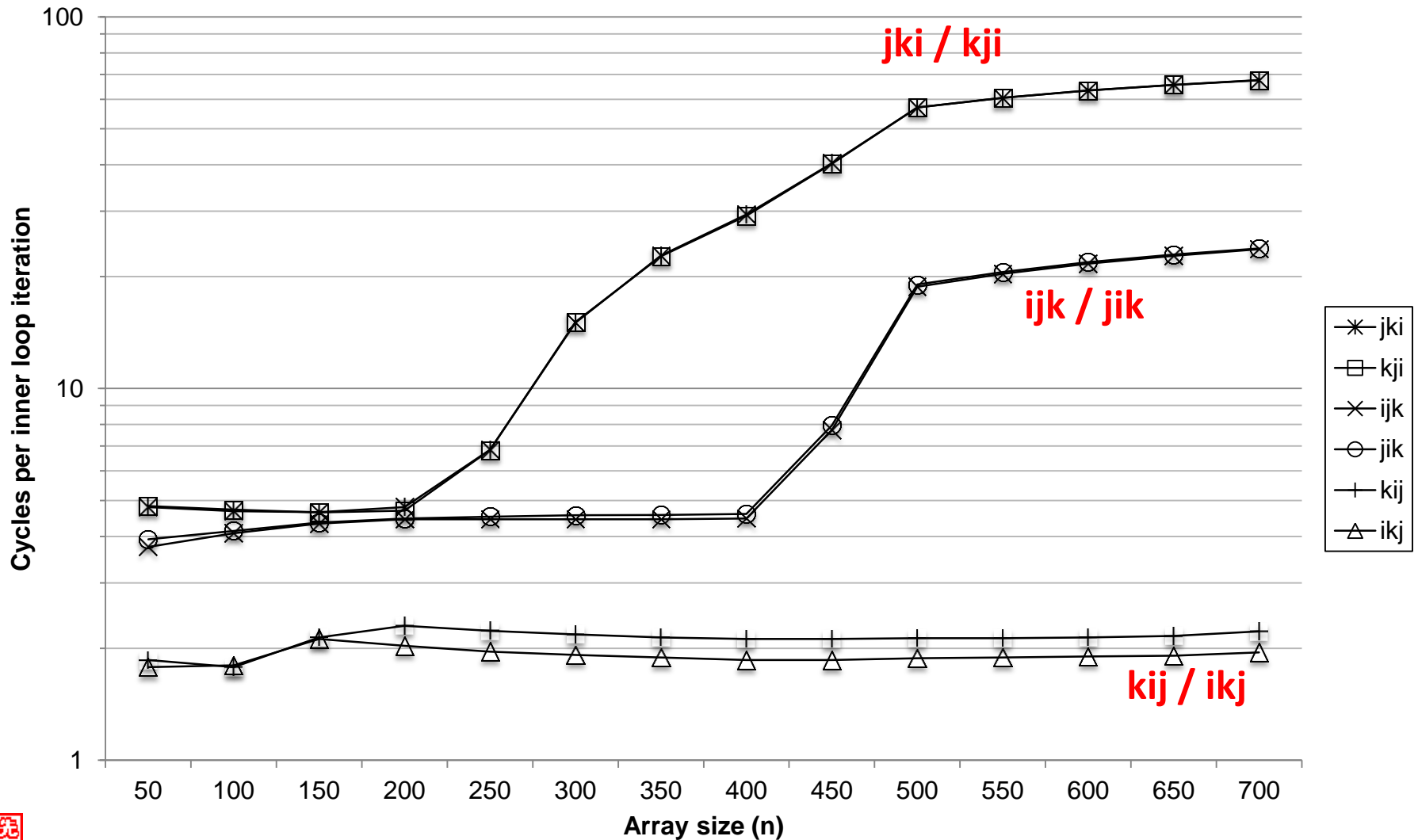
Today

- **Cache organization and operation**

- **Performance impact of caches**
    - The memory mountain
    - Rearranging loops to improve spatial locality
    - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```
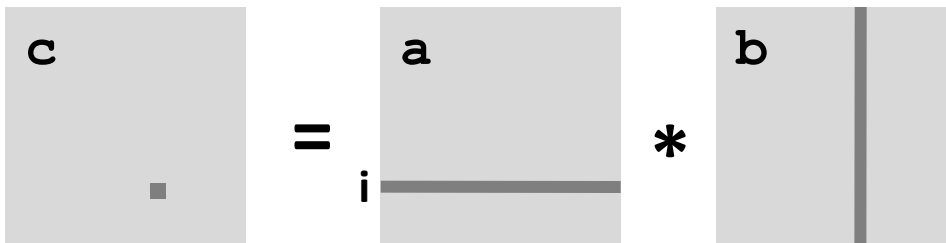
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
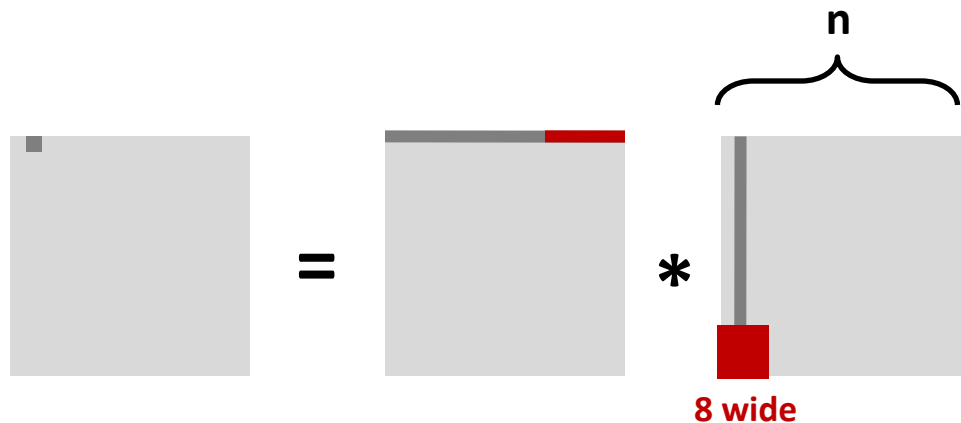 - n/8 + n = 9n/8 misses

  - Afterwards in cache:
    (schematic)
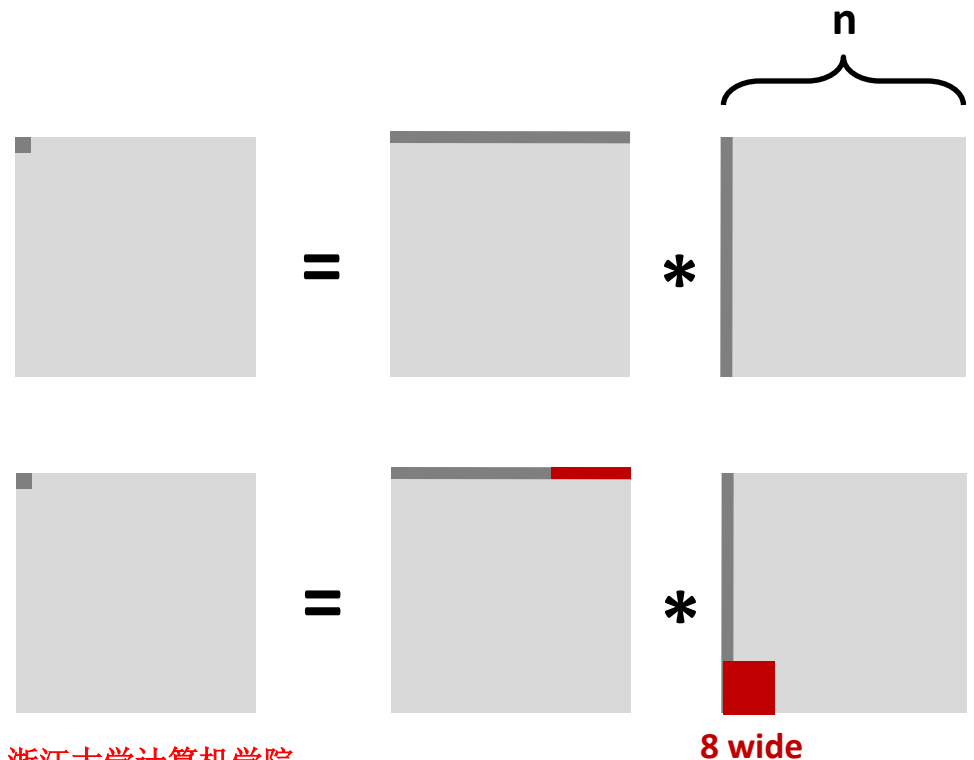
浙江大学计算机学院

Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again:
    $n/8 + n = 9n/8$ misses

n

= *

8 wide

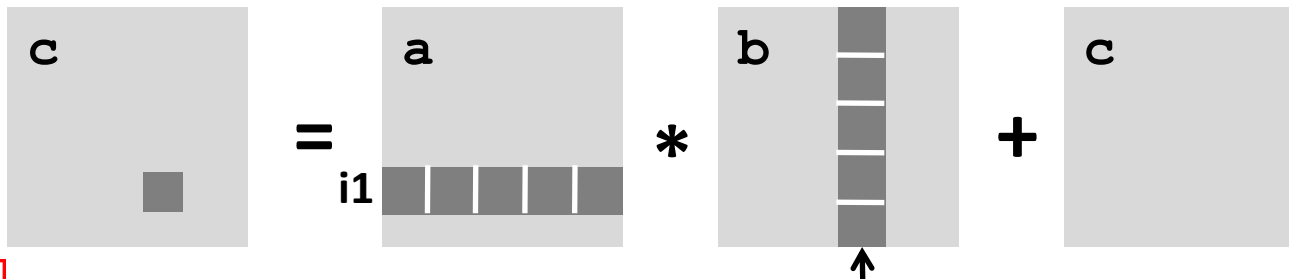- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
                                                    matmult/bmm.c
```
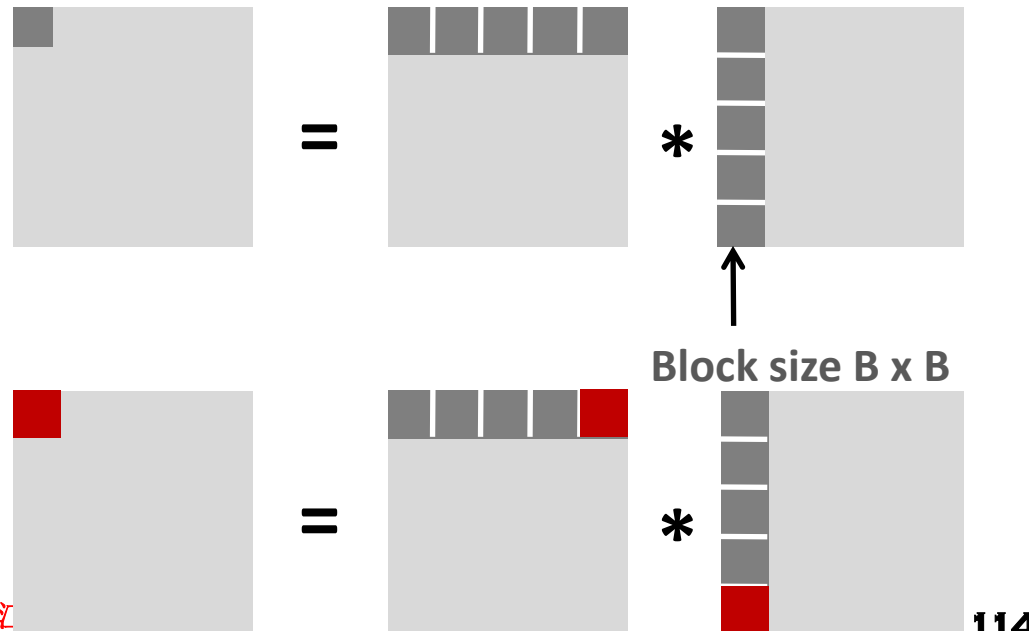
j1

c = a * b + c

i1

Block size B x B

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)

    n/B blocks

  - Afterwards in cache (schematic)

    Block size B x B

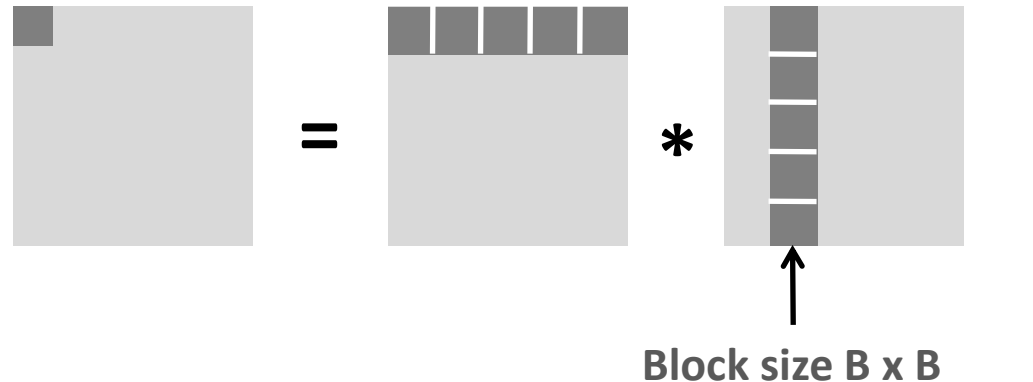# Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ▪ fit into cache: $3B^2 < C$

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

n/B blocks

$=$ $*$

Block size B x B

Blocking Summary

- **No blocking: (9/8) * n^3**
- **Blocking: 1/(4B) * n^3**

- **Suggest largest possible block size B, but limit $3B^2 < C$!**

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used $O(n)$ times!
  - But program has to be written properly

Cache Summary

- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.