

# The Memory Hierarchy

Lecture 8 - 2015

Mads Chr. Olesen

**Credits to Randy Bryant & Dave O'Hallaron  
from Carnegie Mellon**

# 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.

## ■ Static RAM (SRAM)

- Each cell stores a bit with a four or six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to electrical noise (EMI), radiation, etc.
- Faster and more expensive than DRAM.

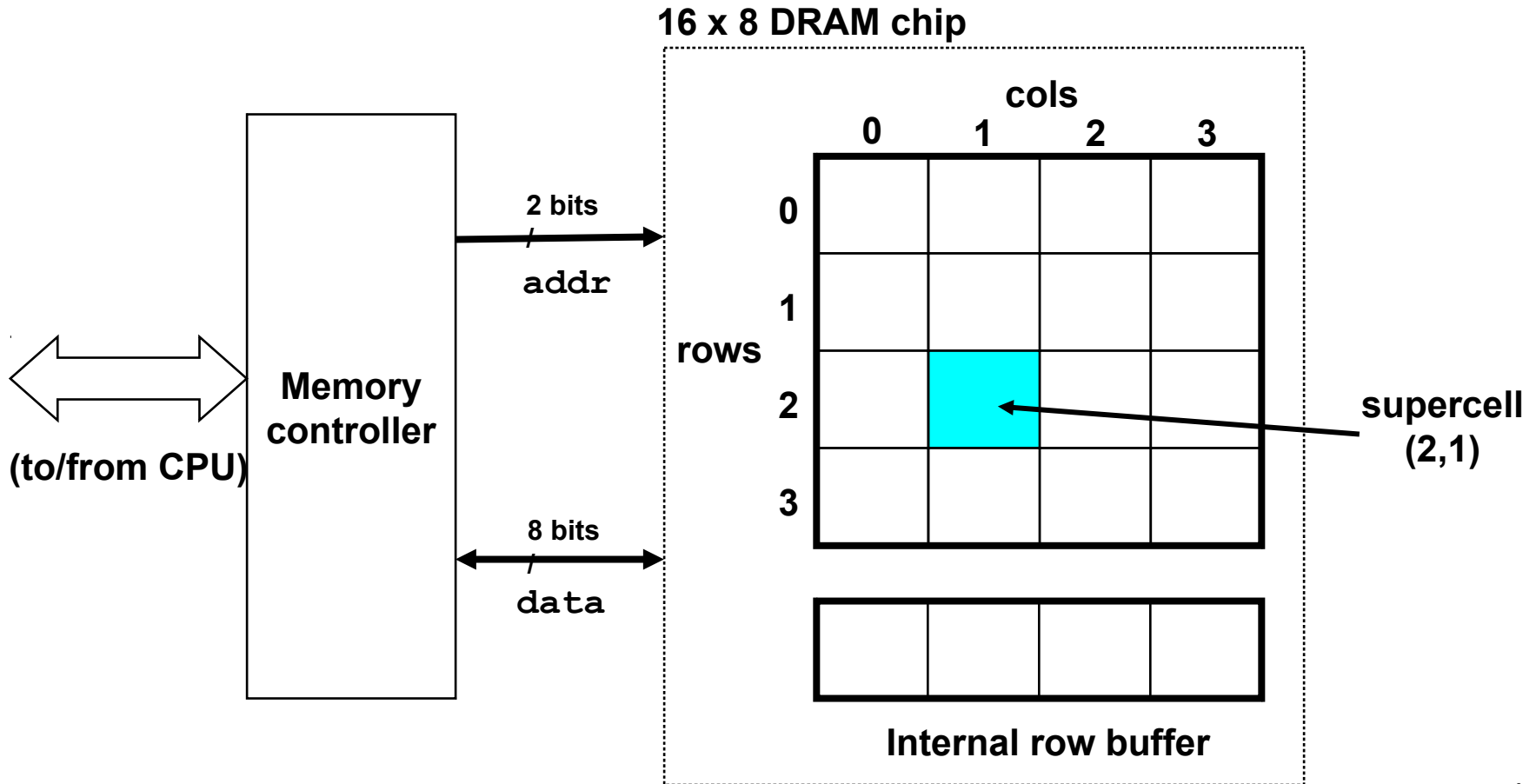
## ■ Dynamic RAM (DRAM)

- Each cell stores bit with a capacitor. One transistor is used for access
- Value must be refreshed every 10-100 ms.
- More sensitive to disturbances (EMI, radiation,...) than SRAM.
- Slower and cheaper than SRAM.

# Conventional DRAM Organization

## ■ $d \times w$ DRAM:

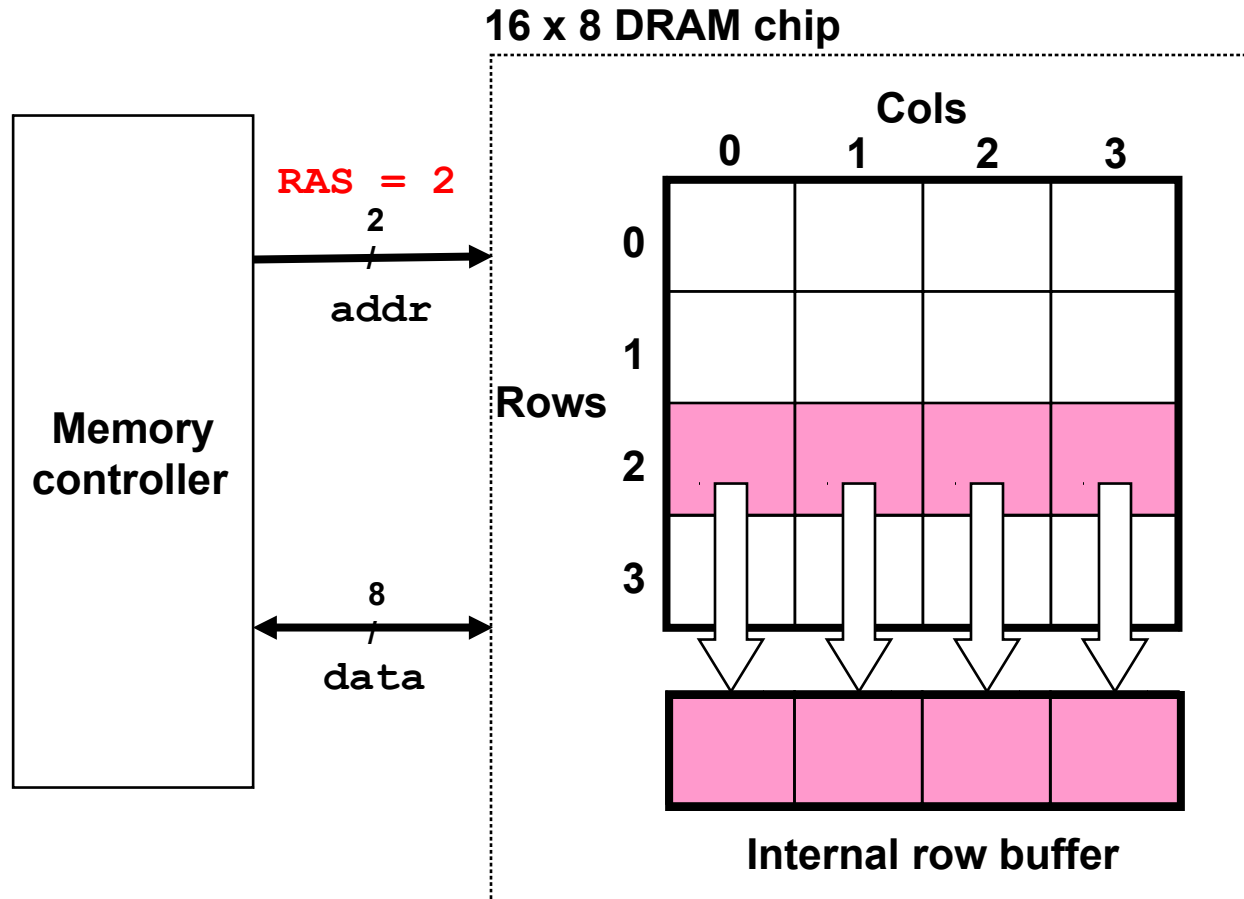
- $dw$  total bits organized as  $d$  **supercells** of size  $w$  bits



# Reading DRAM Supercell (2,1)

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

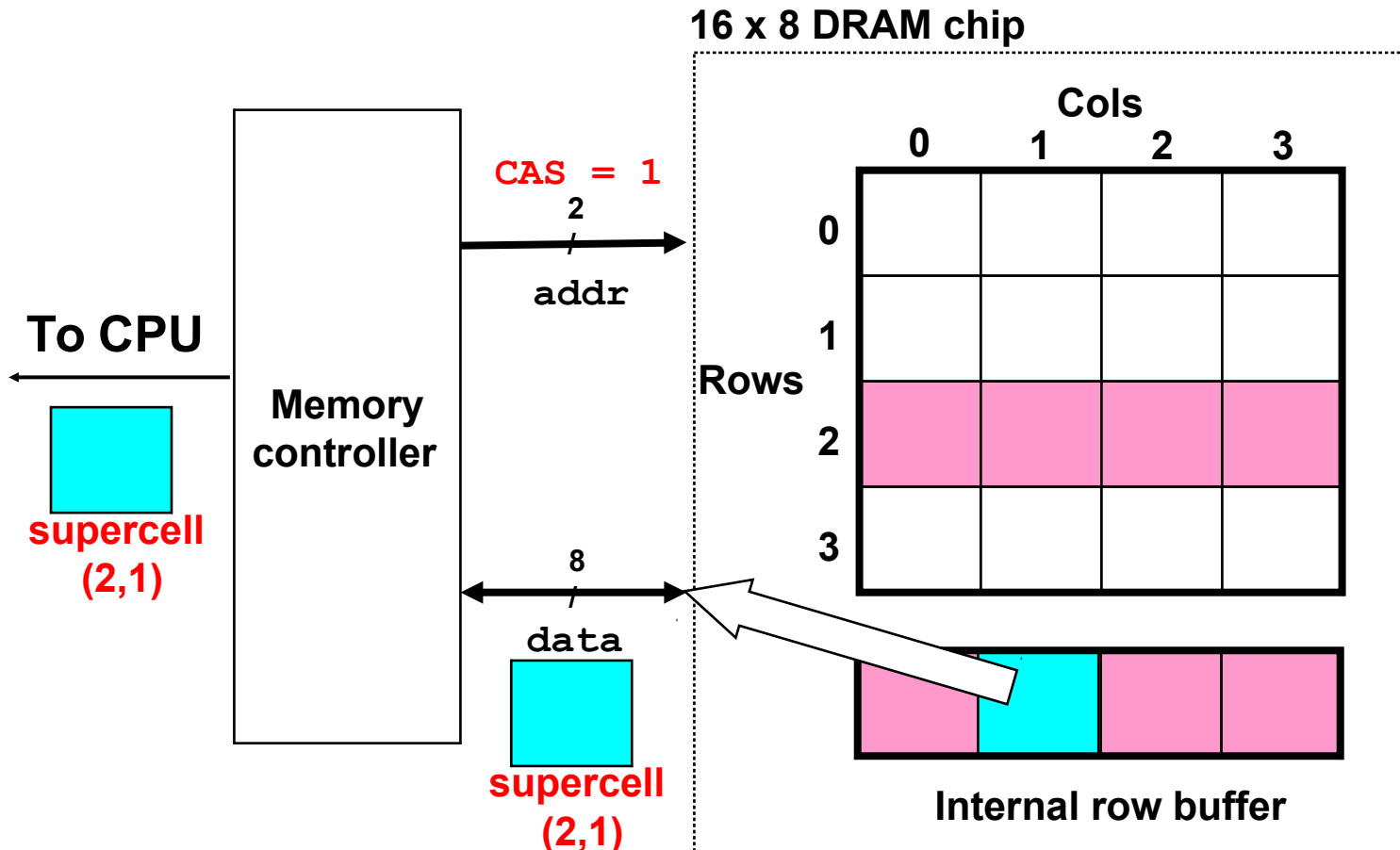
Step 1(b): Row 2 copied from DRAM array to row buffer.



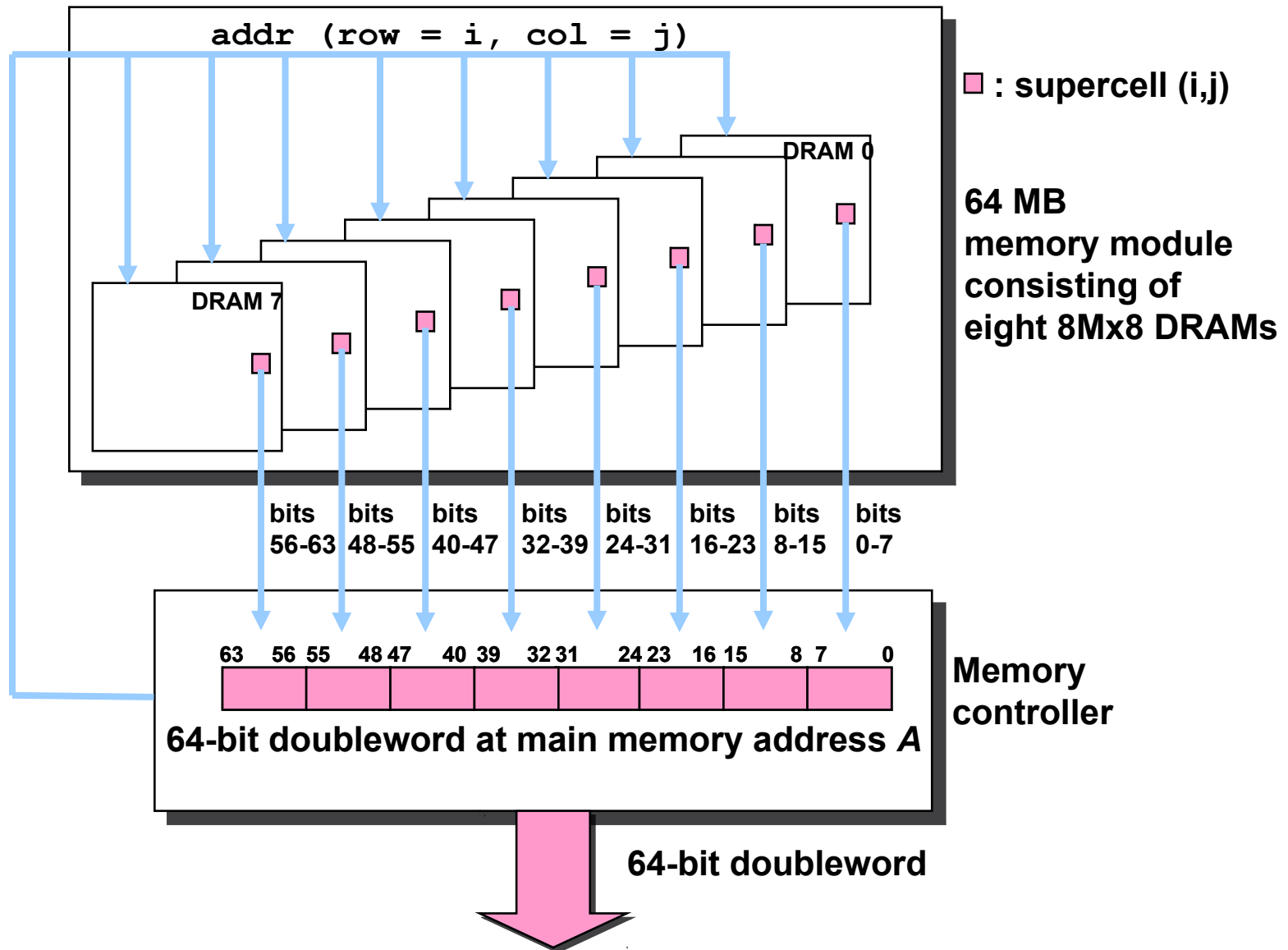
# Reading DRAM Supercell (2,1)

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.



# 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



**DDR3 1333MHz 3GB CL7**  
Kit w/3x 1GB BallistiX modules, CL7-7-7-24, 1.65V, for Core i7, GREEN



Lagerstatus: ☒ 1 på lager.  
(1-2 dages leveringstid)

- ☐ Tilbehør
- ☐ Sammenlign dette produkt

Del med: [f](#) [t](#)

Varenummer: 438707    Producentens varenr.: BL3KIT12864TG1337    Vægt m/emballage: 0.13 kg    Producentens garanti (md.): 120

Produktinfo	Udvidet info	Produktanmeldelser	Tests	Producentlinks	Tilbehør (3)	Tip en ven
Generelt						
Lagerkapacitet				3 GB : 3 x 1 GB		
Opgraderingstype				Industristandard		
Hukommelse						
Type				DRAM		
Teknologi				DDR3 SDRAM		
Model				DIMM 240-pin		
Modul Højde (tommer)				1.18		
Hukommelsesfrekvens				1333 MHz ( PC3-10600 )		
Søgetids Timinger				CL7 ( 7-7-7-24 )		
Dataintegritetskontrol				Ikke-paritet		
RAM egenskaber				Aluminium varme spreder , ikke bufferet		
Modulkonfiguration				128 x 64		
Forsyningsspænding				1.65 V		
Blybelægning				Guld		

# Nonvolatile Memories

## ■ DRAM and SRAM are volatile memories

- Lose information if powered off.

## ■ Nonvolatile memories retain value even if powered off

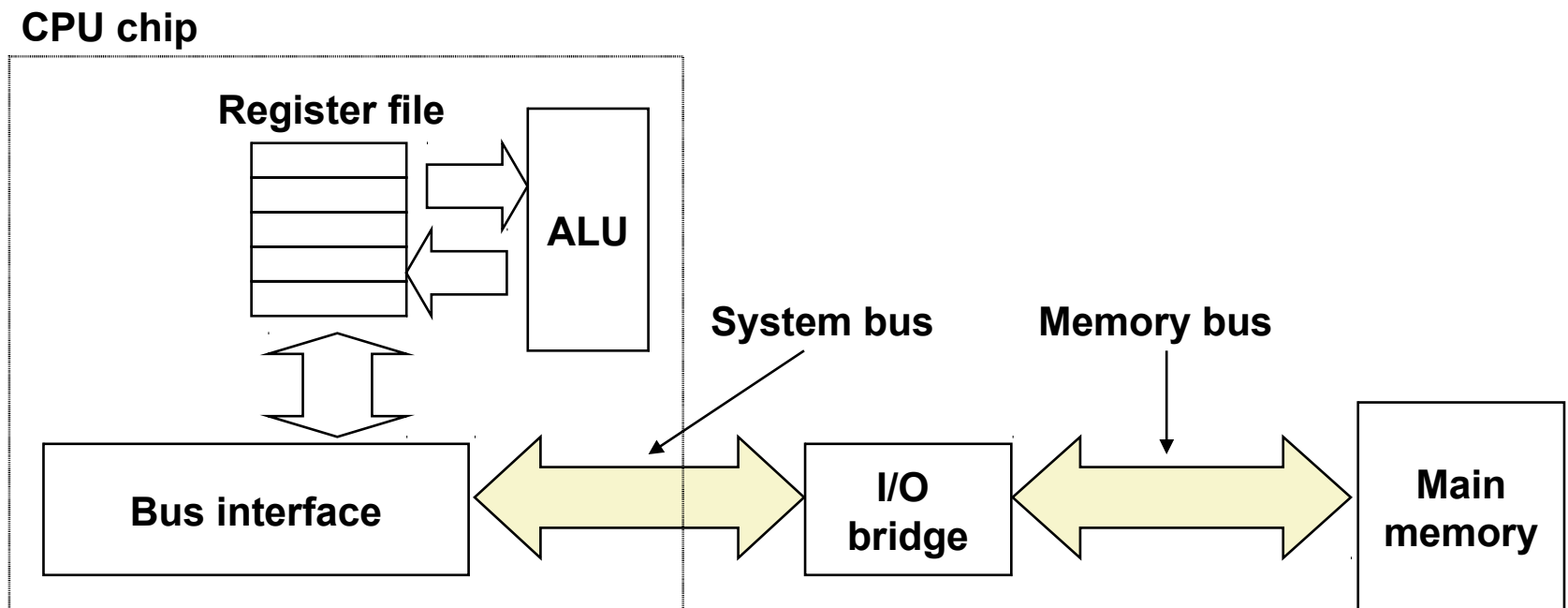
- Read-only memory (**ROM**): programmed during production
- Programmable ROM (**PROM**): can be programmed once
- Erasable PROM (**EPROM**): can be bulk erased (UV, X-Ray)
- Electrically erasable PROM (**EEPROM**): electronic erase capability
- Flash memory: EEPROMs with partial (sector) erase capability
  - 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

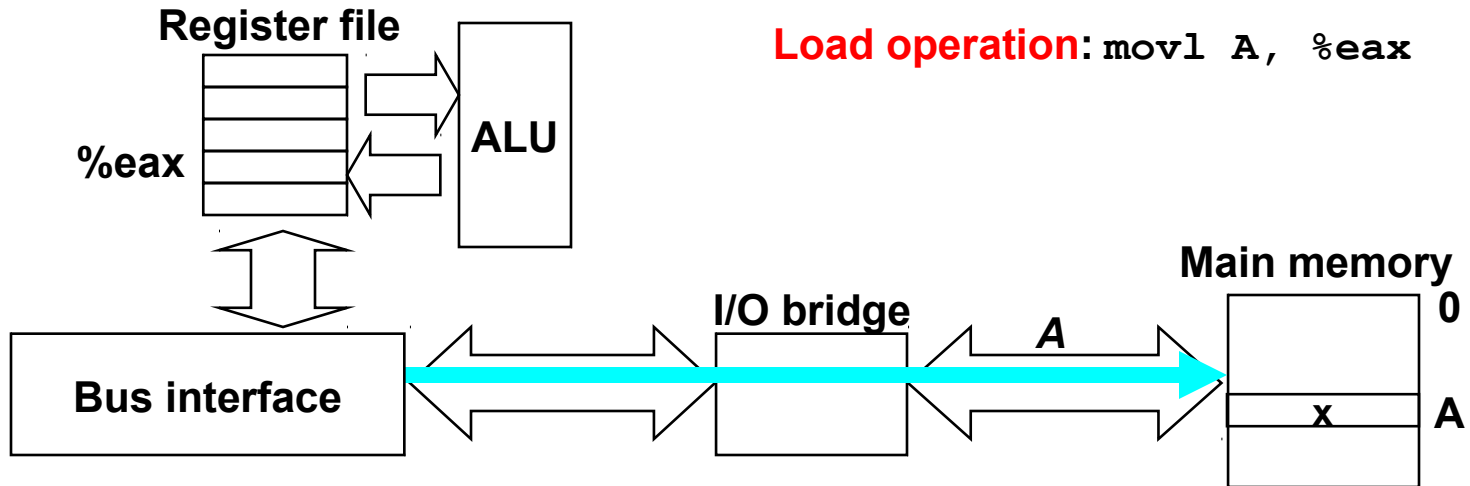
## Traditional Bus Structure Connecting CPU and Memory

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



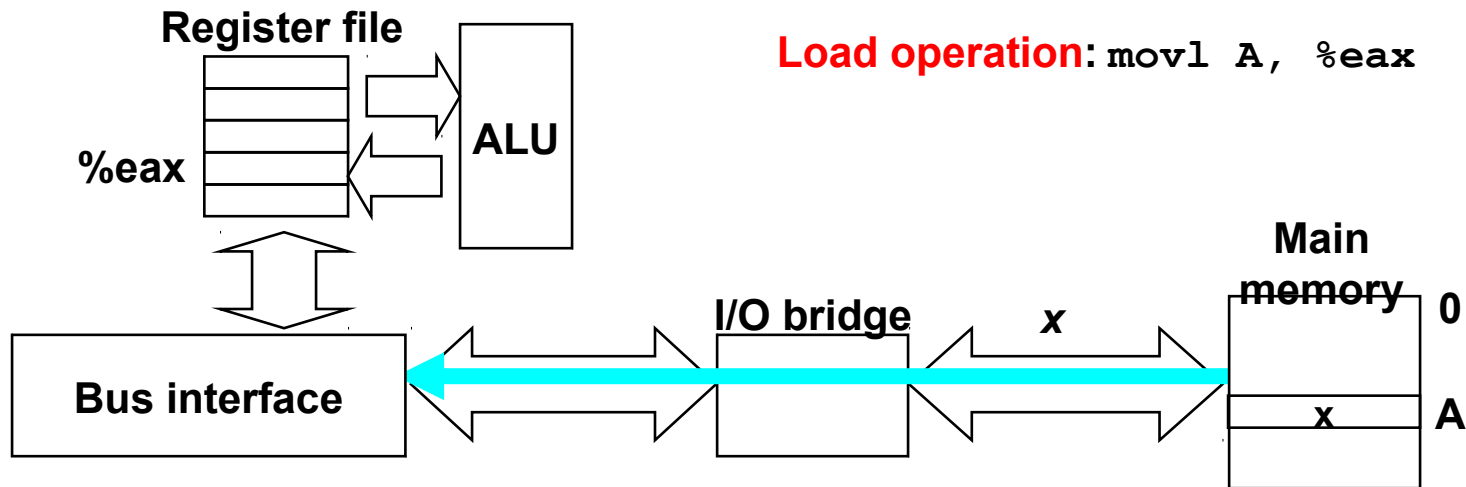
# Memory Read Transaction (1)

- CPU places address A on the memory bus.



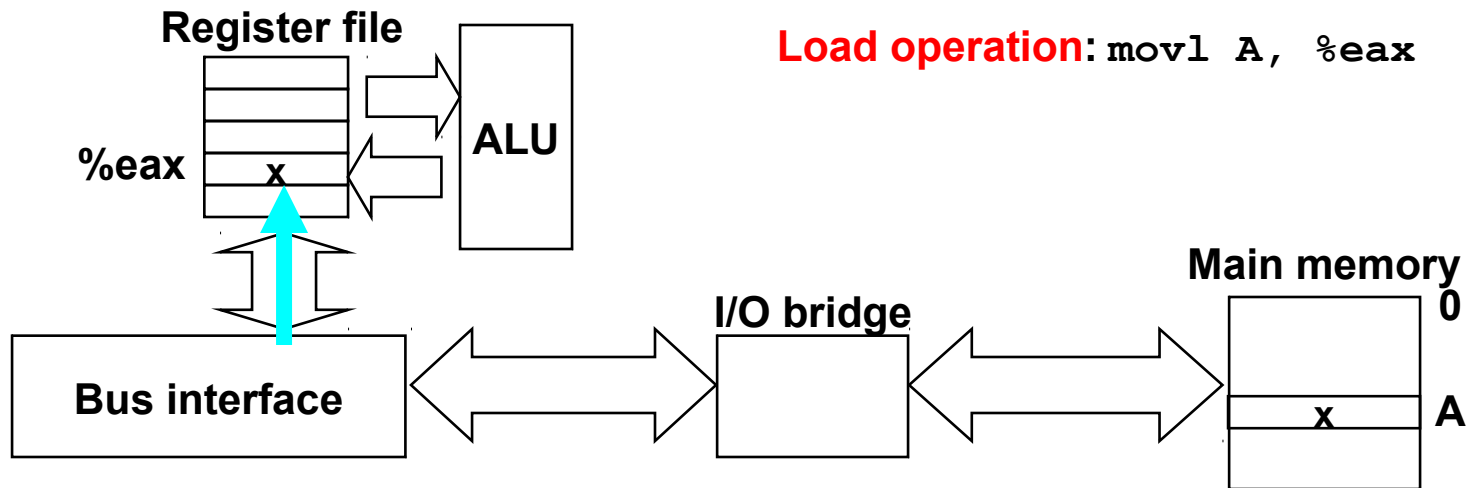
# Memory Read Transaction (2)

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



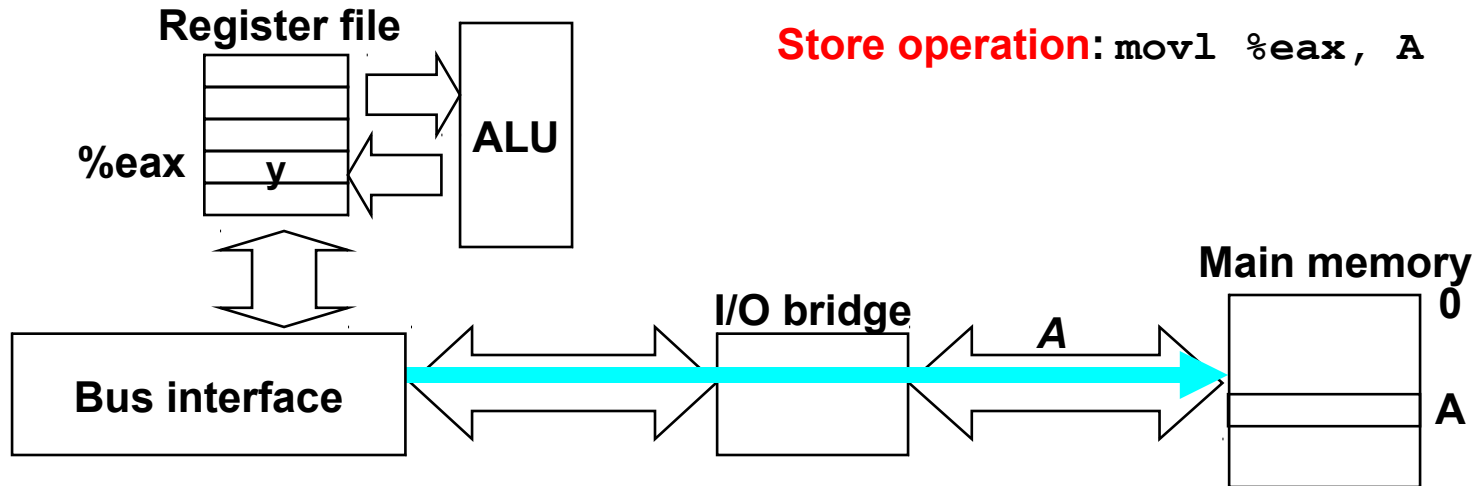
# Memory Read Transaction (3)

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



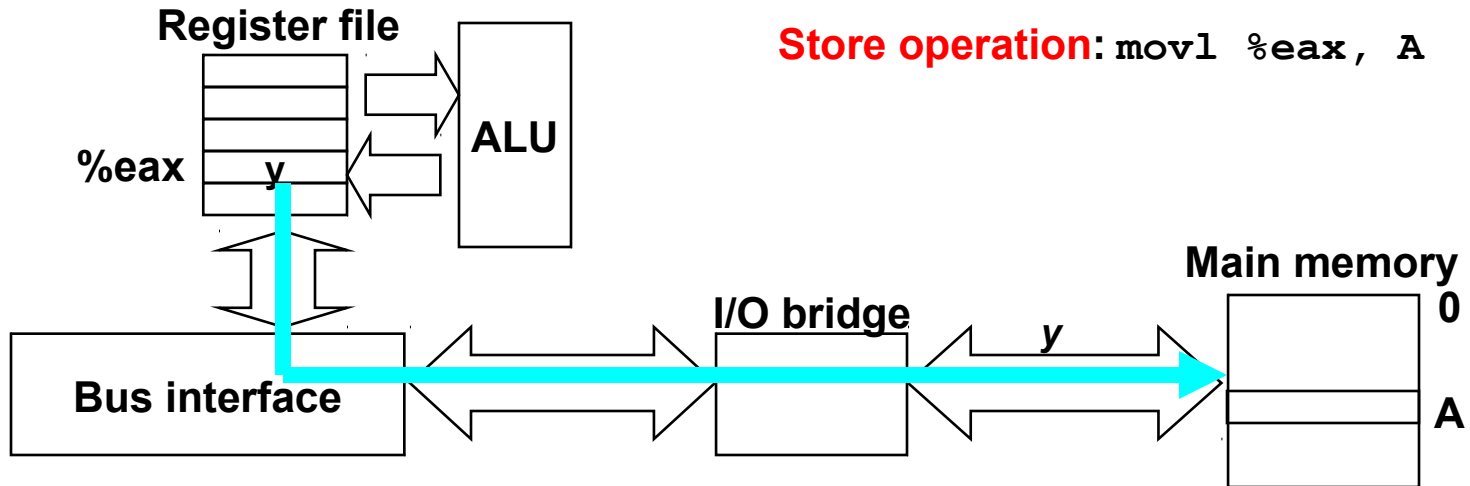
# Memory Write Transaction (1)

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



# Memory Write Transaction (2)

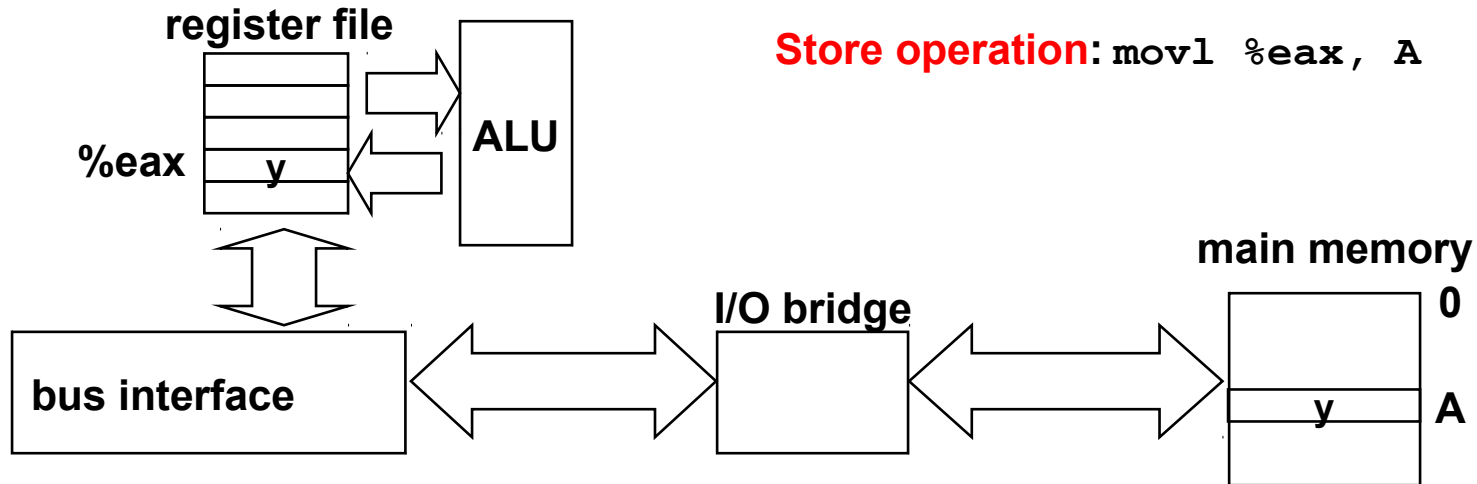
- CPU places data word  $y$  on the bus.



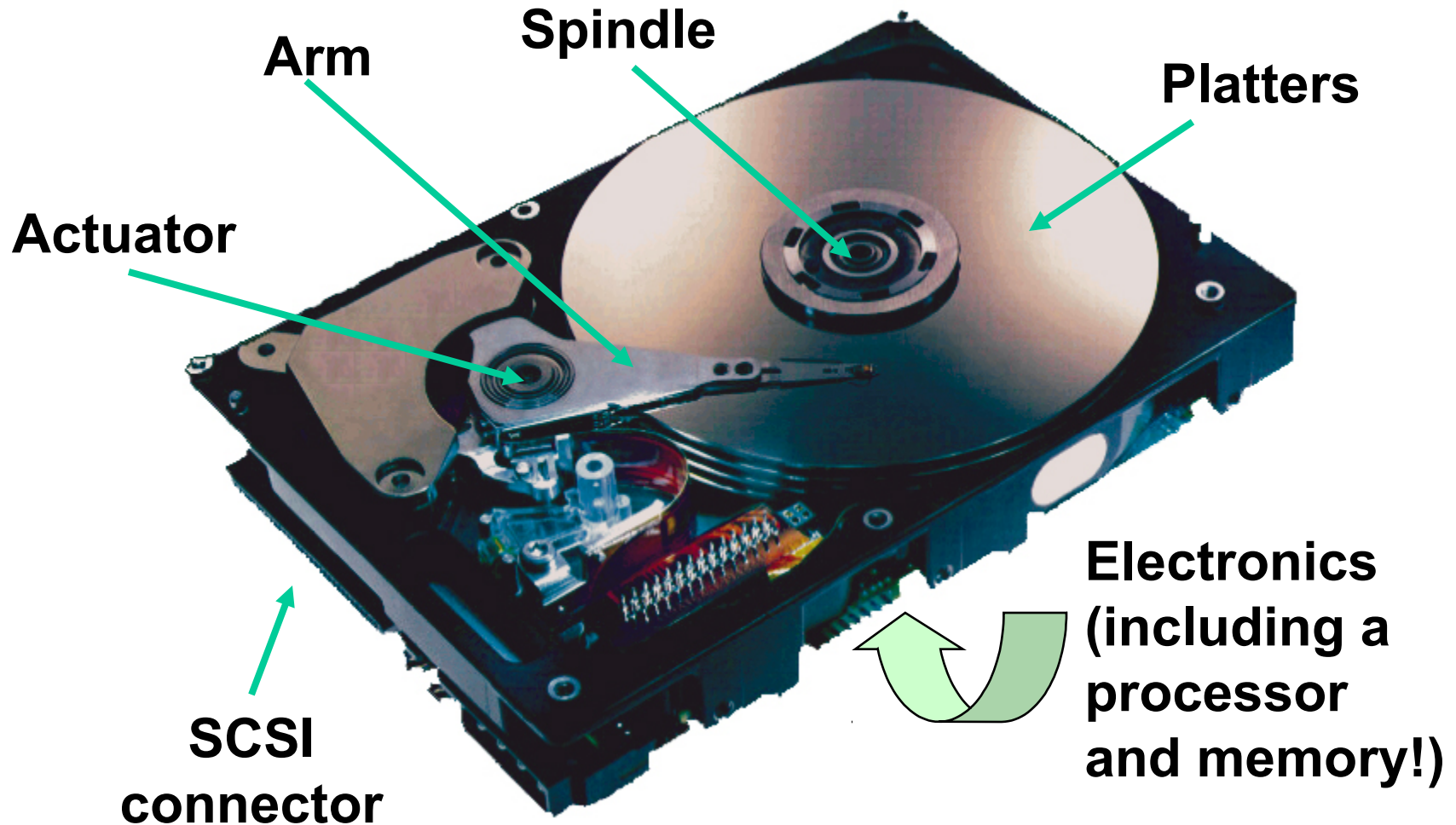


# Memory Write Transaction (3)

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



# What's Inside A Disk Drive?



*Image courtesy of Seagate Technology*

# Disk Access Time

## ■ Average time to access some target sector approximated by :

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

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

- Time to position heads over cylinder containing target sector.
- Typical  $T_{\text{avg seek}}$  is 3—9 ms

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

- Time waiting for first bit of target sector to pass under r/w head.
- $T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$
- Typical  $T_{\text{avg rotation}} = 7200 \text{ RPMs}$

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

- Time to read the bits in the target sector.
- $T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg \# sectors/track}) \times 60 \text{ secs}/1 \text{ min.}$

# Disk Access Time Example

## ■ Given:

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

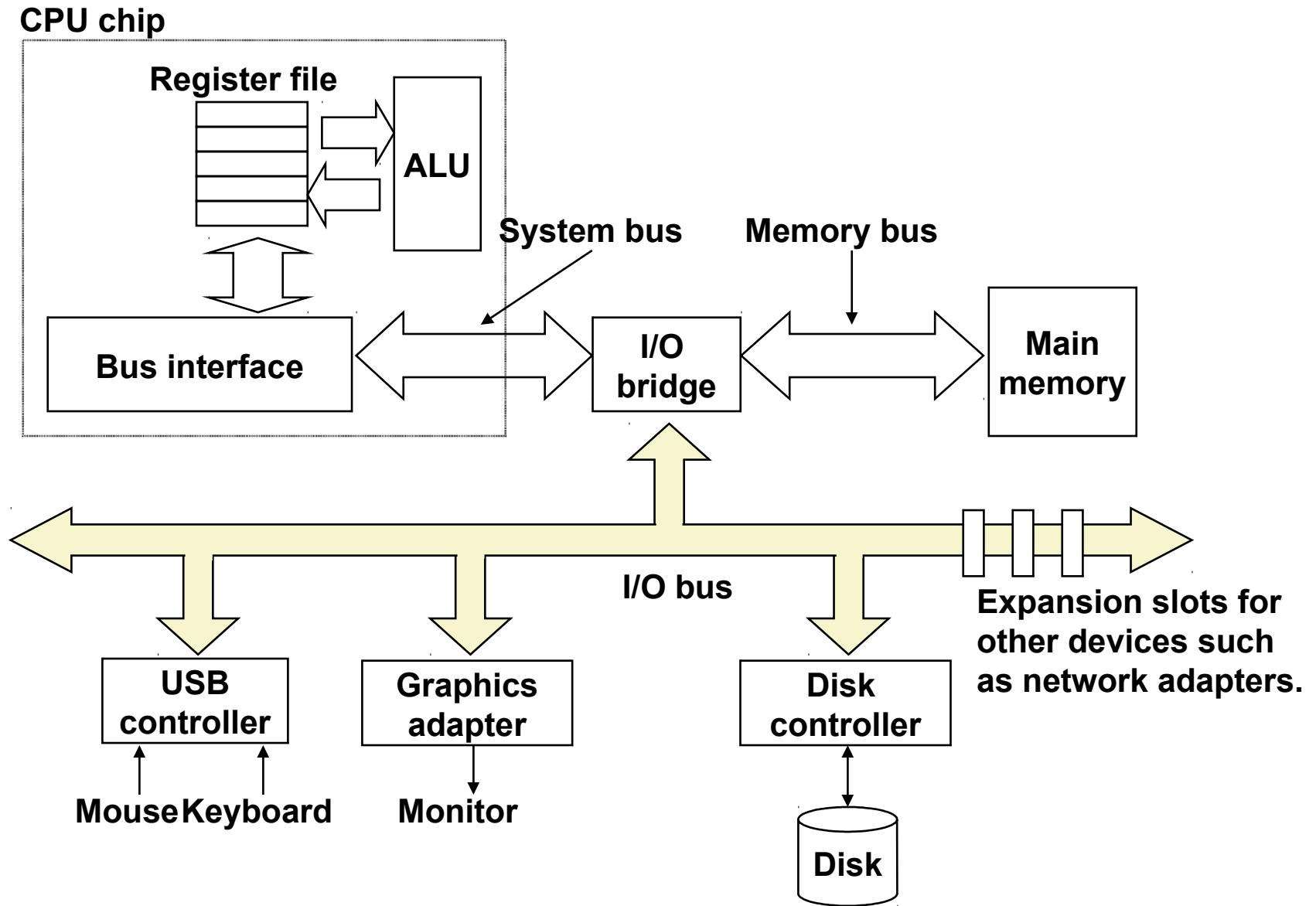
## ■ Derived:

- $T_{\text{avg rotation}} = 1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms}.$
- $T_{\text{avg transfer}} = 60/7200 \text{ RPM} \times 1/400 \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$
- $T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$

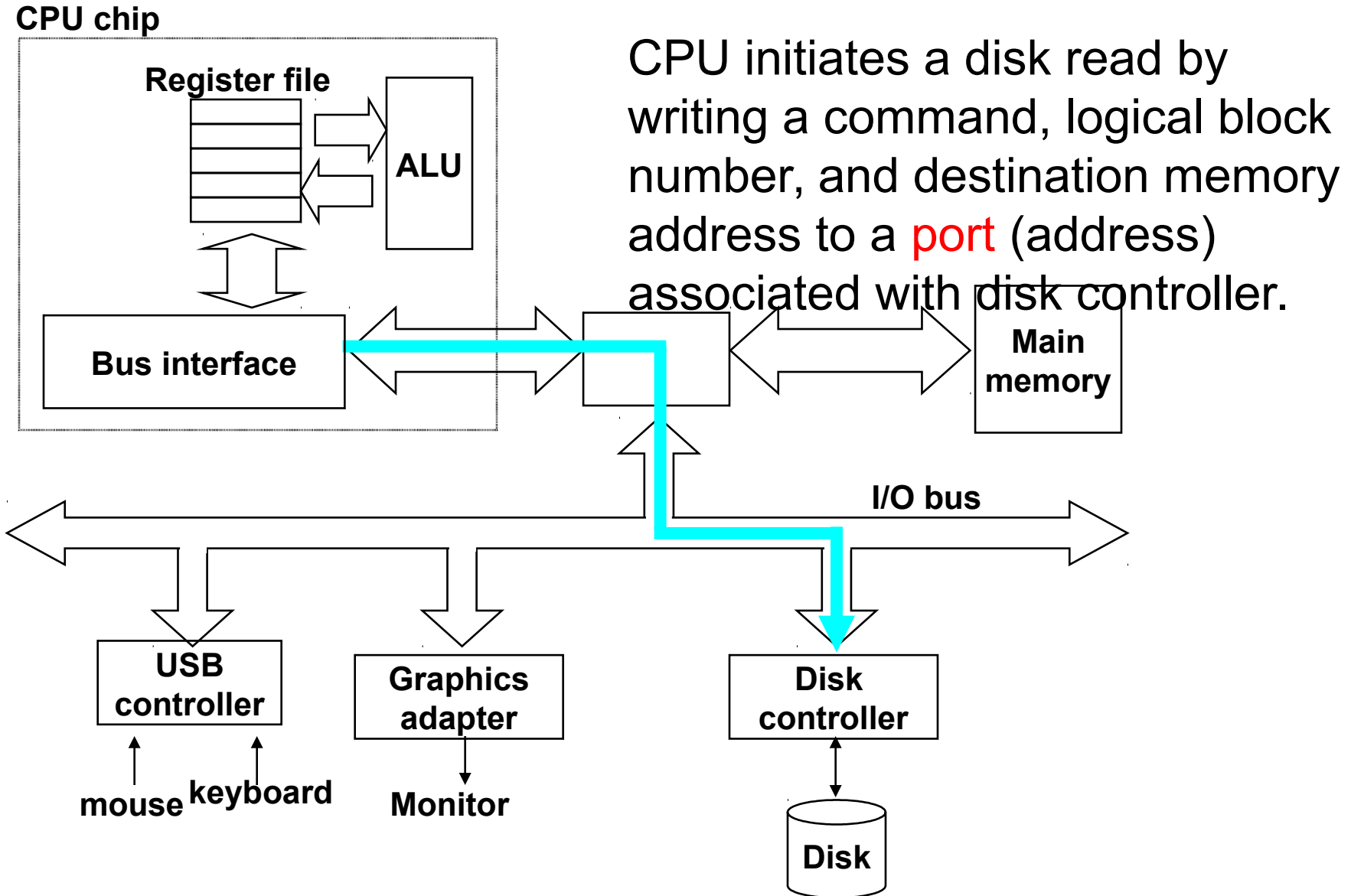
## ■ Important points:

- Access time dominated by seek time and rotational latency.
- First bit in a sector is the most expensive, the rest are free.
- SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
  - Disk is about 40,000 times slower than SRAM,
  - 2,500 times slower than DRAM.

# I/O Bus

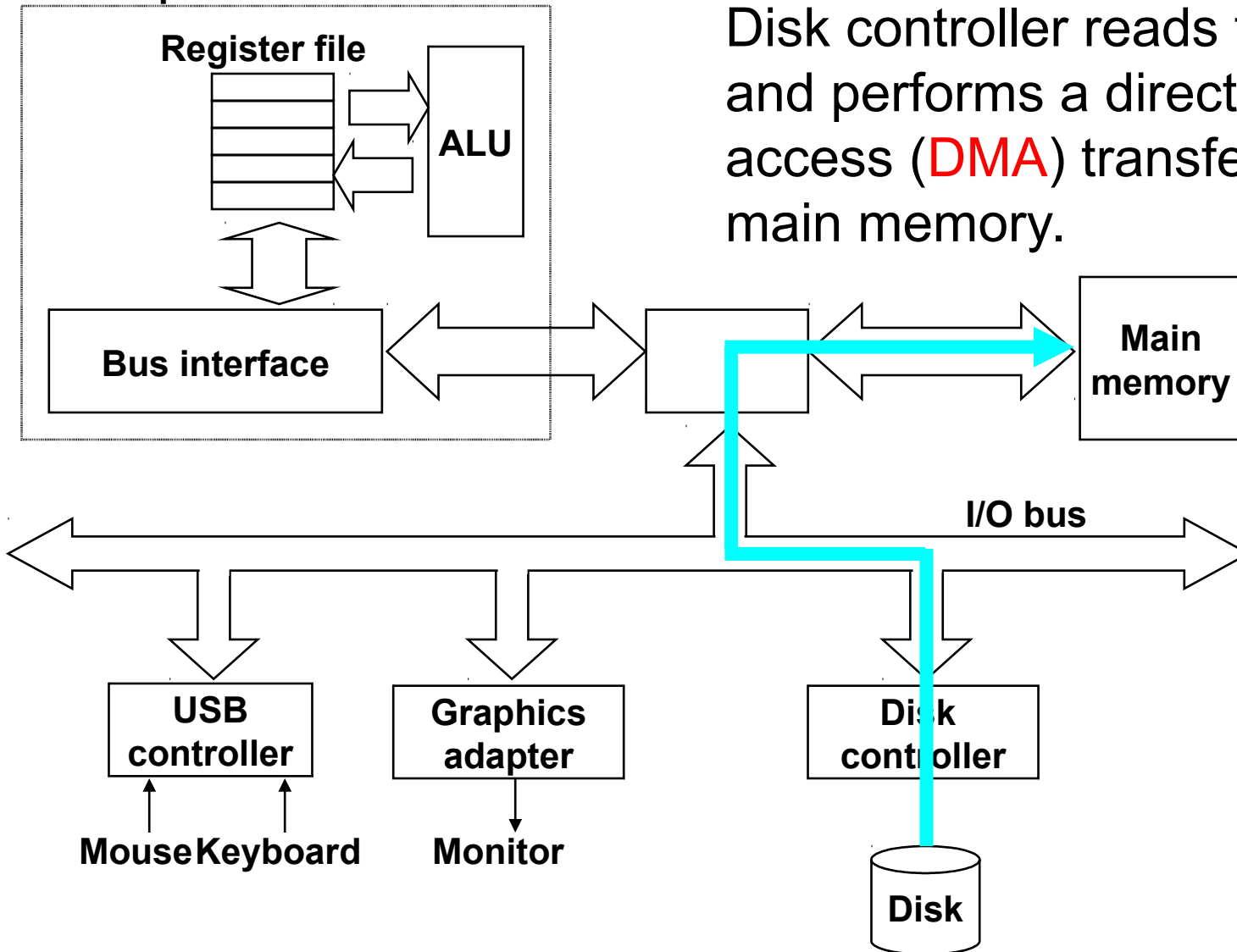


# Reading a Disk Sector (1)



# Reading a Disk Sector (2)

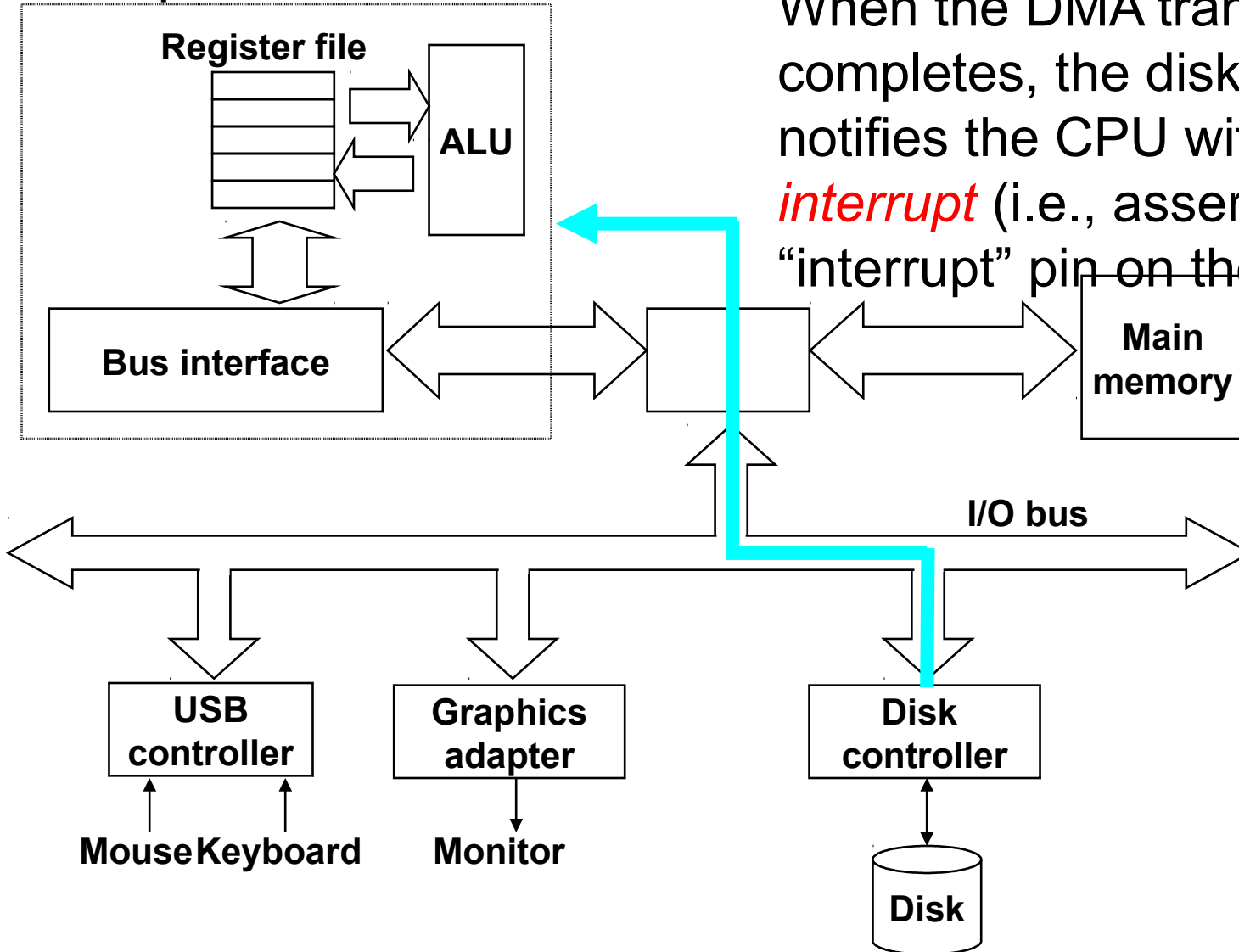
CPU chip



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

# Reading a Disk Sector (3)

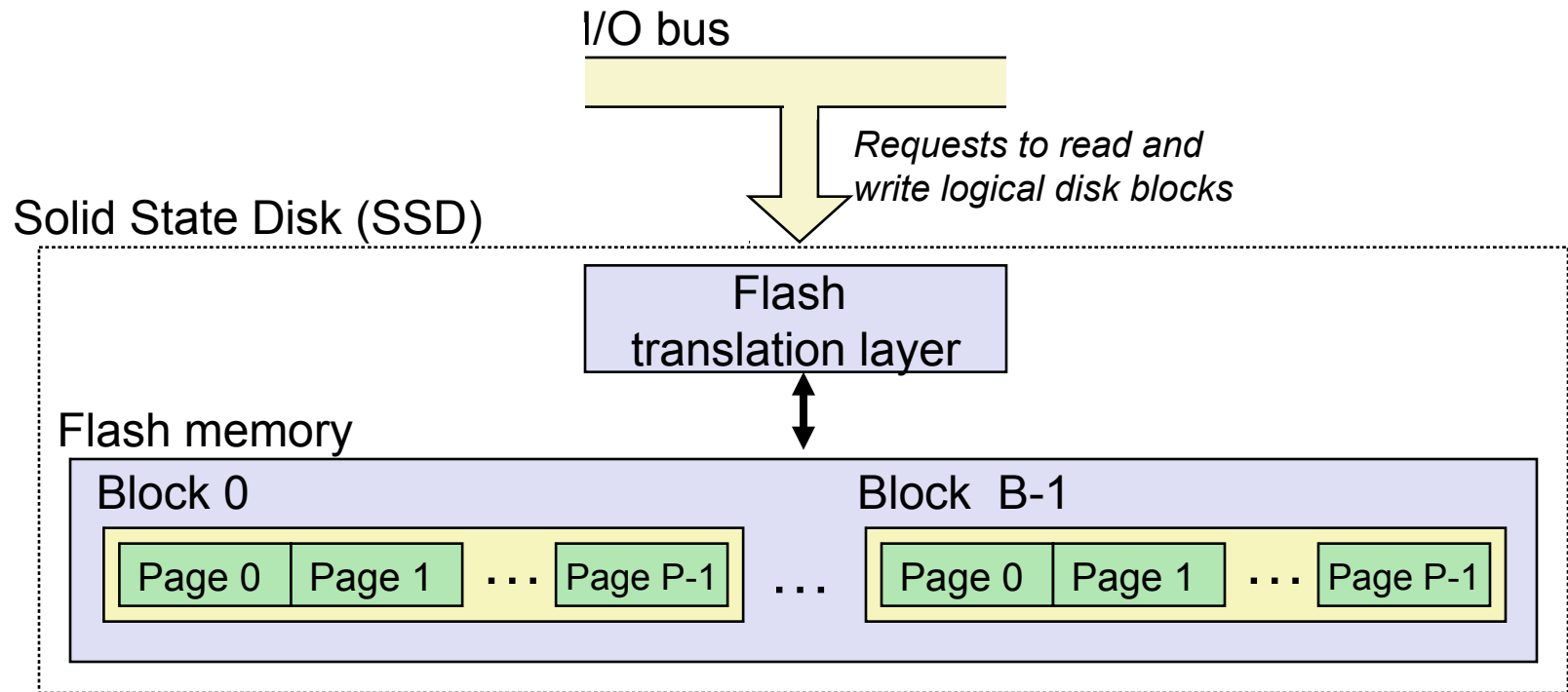
CPU chip



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



# 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 100,000 repeated writes.**

# SSD Performance Characteristics

Sequential read tput	250 MB/s	Sequential write tput	170 MB/s
Random read tput	140 MB/s	Random write tput	14 MB/s
Random read access	30 us	Random write access	300 us

## ■ Why are random writes so slow?

- Erasing a block is slow (around 1 ms)
- Write to a page triggers a copy of all useful pages in the block
  - Find a used block (new block) and erase it
  - Write the page into the new block
  - Copy other pages from old block to the new block

# 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 X25 guarantees 1 petabyte (10<sup>15</sup> bytes) of random writes before they wear out
- In 2010, about 100 times more expensive per byte

## ■ Applications

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

# Storage Trends

## SRAM

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	19,200	2,900	320	256	100	75	60	x320
access (ns)	300	150	35	15	3	2	1.5	x200

## DRAM

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	8,000	880	100	30	1	0.1	0.06	x130,000
access (ns)	375	200	100	70	60	50	40	x9
size (MB)	0.064	0.256	4	16	64	2,000	8,000	x125,000

## Disk

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	500	100	8	0.30	0.01	0.005	0.0003	x1,600,000
access (ms)	87	75	28	10	8	4	3	x29
size (MB)	1	10	160	1,000	20,000	160,000	1,5G	x1,500,000

# CPU Clock Rates

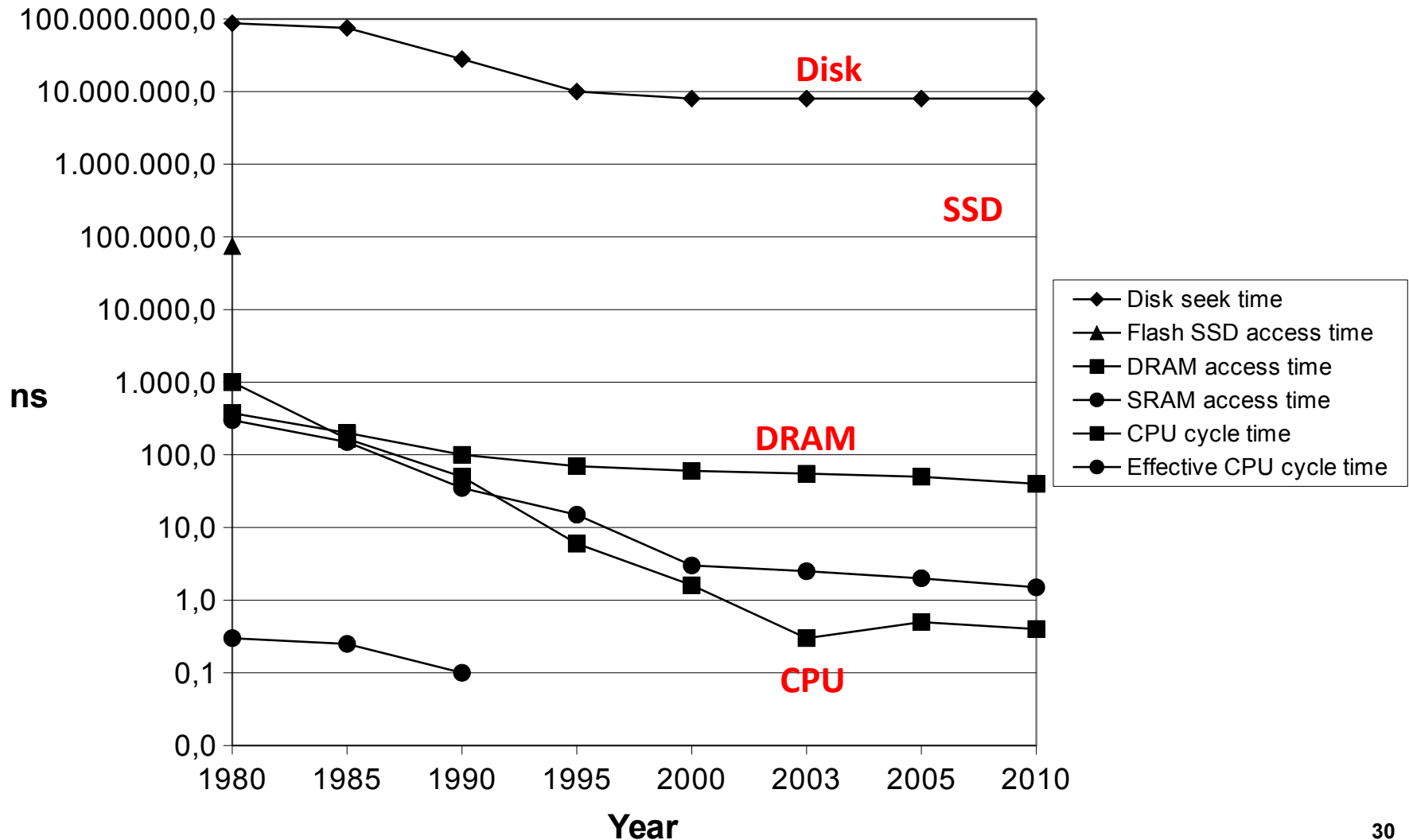
Inflection point in computer history  
when designers hit the “Power Wall”



	1980	1990	1995	2000	2003	2005	2010	2010:1980
<b>CPU</b>	8080	386	Pentium	P-III	P-4	Core 2	Core i7	
<b>Clock rate (MHz)</b>	1	20	150	600	3300	2000	2500	x2500
<b>Cycle time (ns)</b>	1000	50	6	1.6	0.3	0.50	0.4	x2500
<b>Cores</b>	1	1	1	1	1	2	4	x4
<b>Effective Cycle time (ns)</b>	1000	50	6	1.6	0.3	0.25	0.1	x10,000

# 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

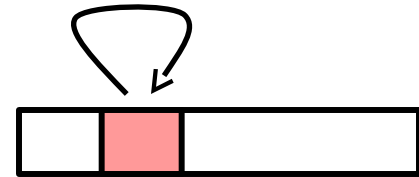


# Locality

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

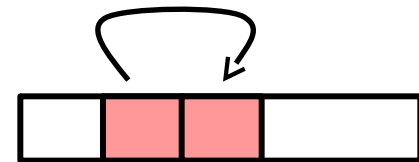
- **Temporal locality:**

- 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).
- Reference variable `sum` each iteration.

**Spatial locality**

**Temporal locality**

## ■ Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

**Spatial locality**

**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;
}
```

# 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;
}
```

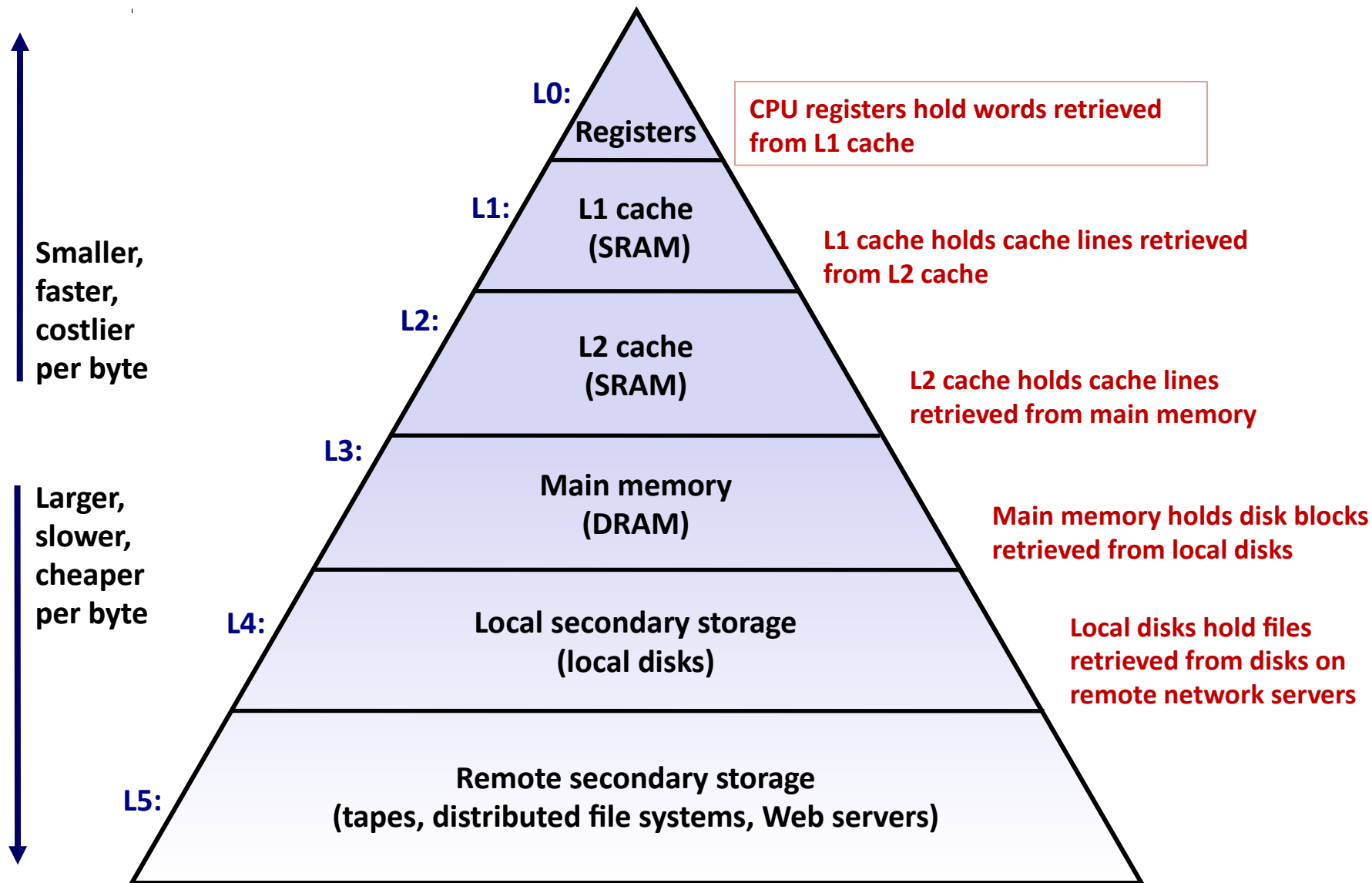
# Memory Hierarchies

- **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**

# An Example Memory Hierarchy

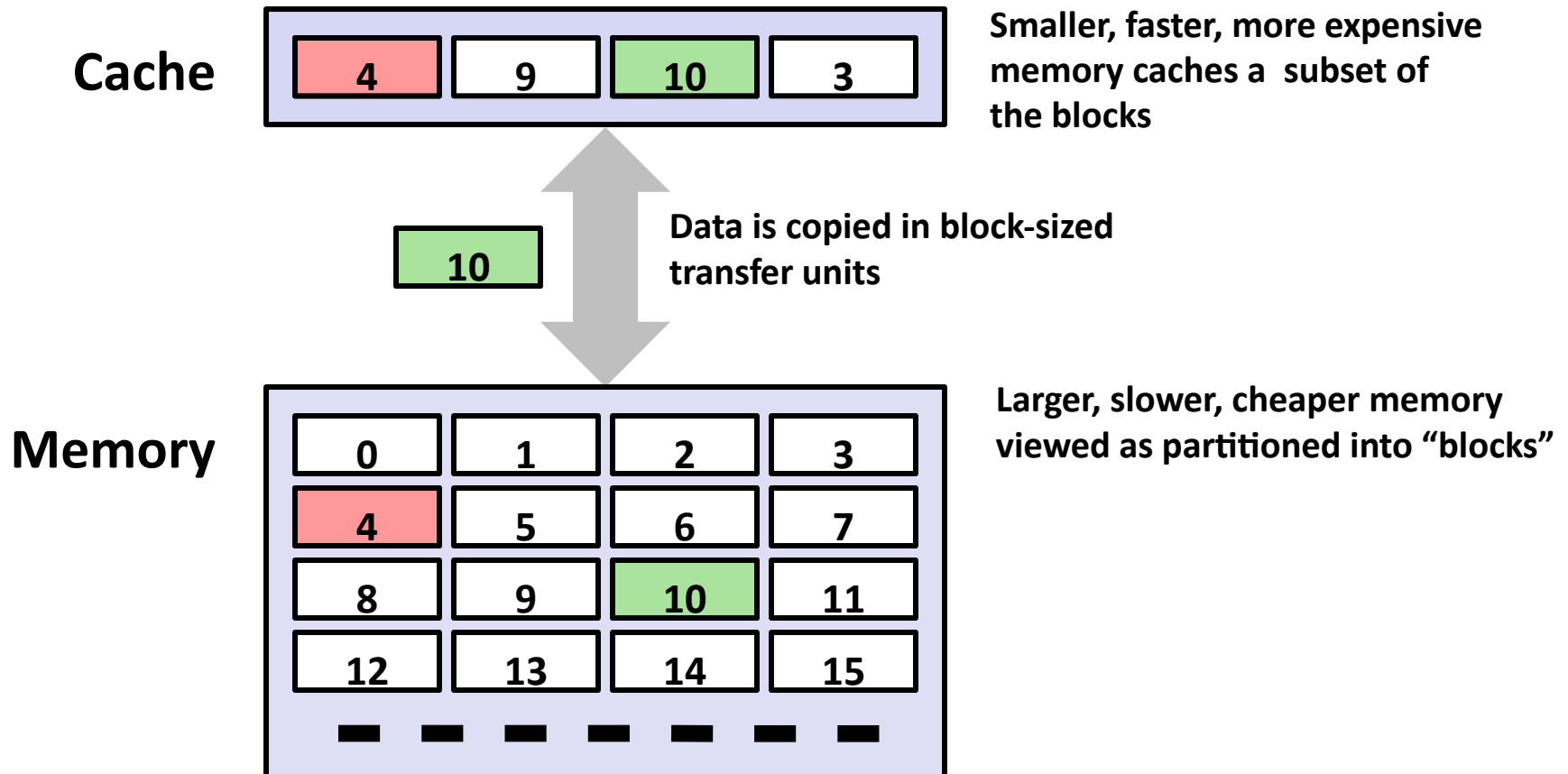




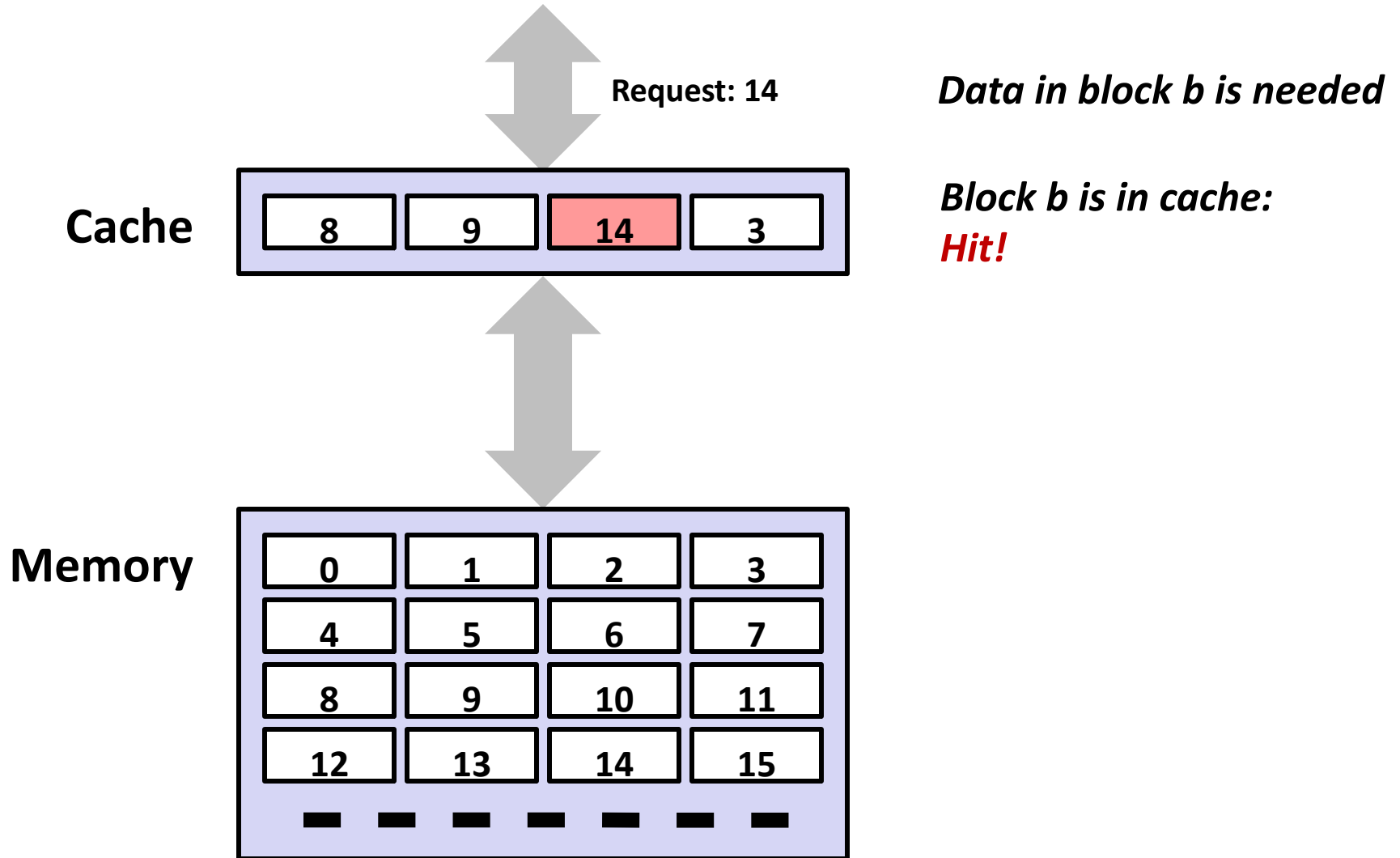
# Caches

- **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.

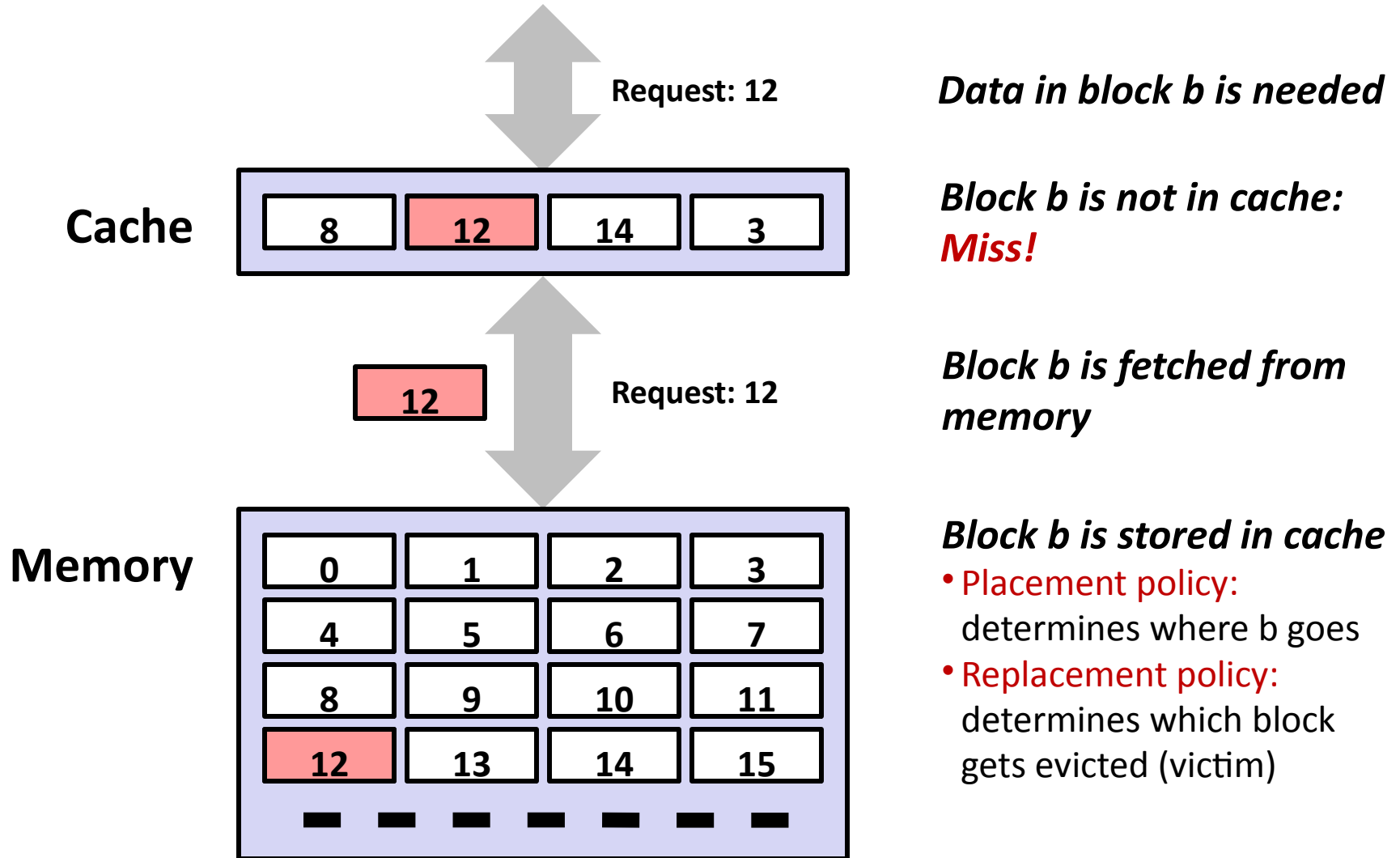
# General Cache Concepts



# General Cache Concepts: Hit



# General Cache Concepts: Miss



# 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  $(i \bmod 4)$  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 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
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	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	AFS/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.**

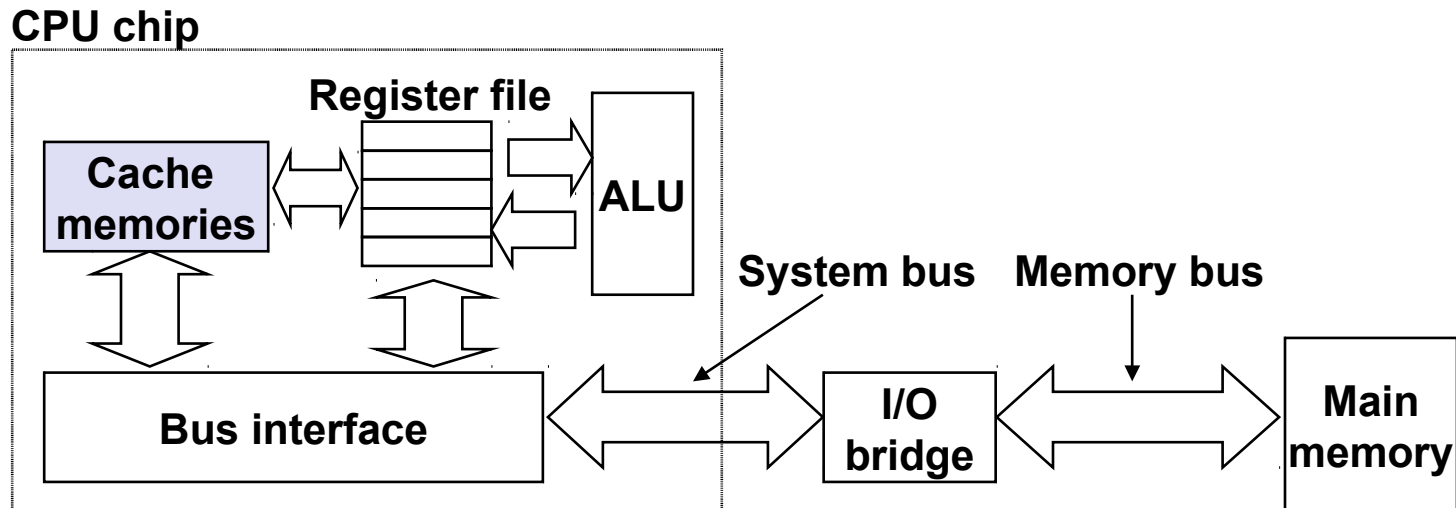
# Today

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

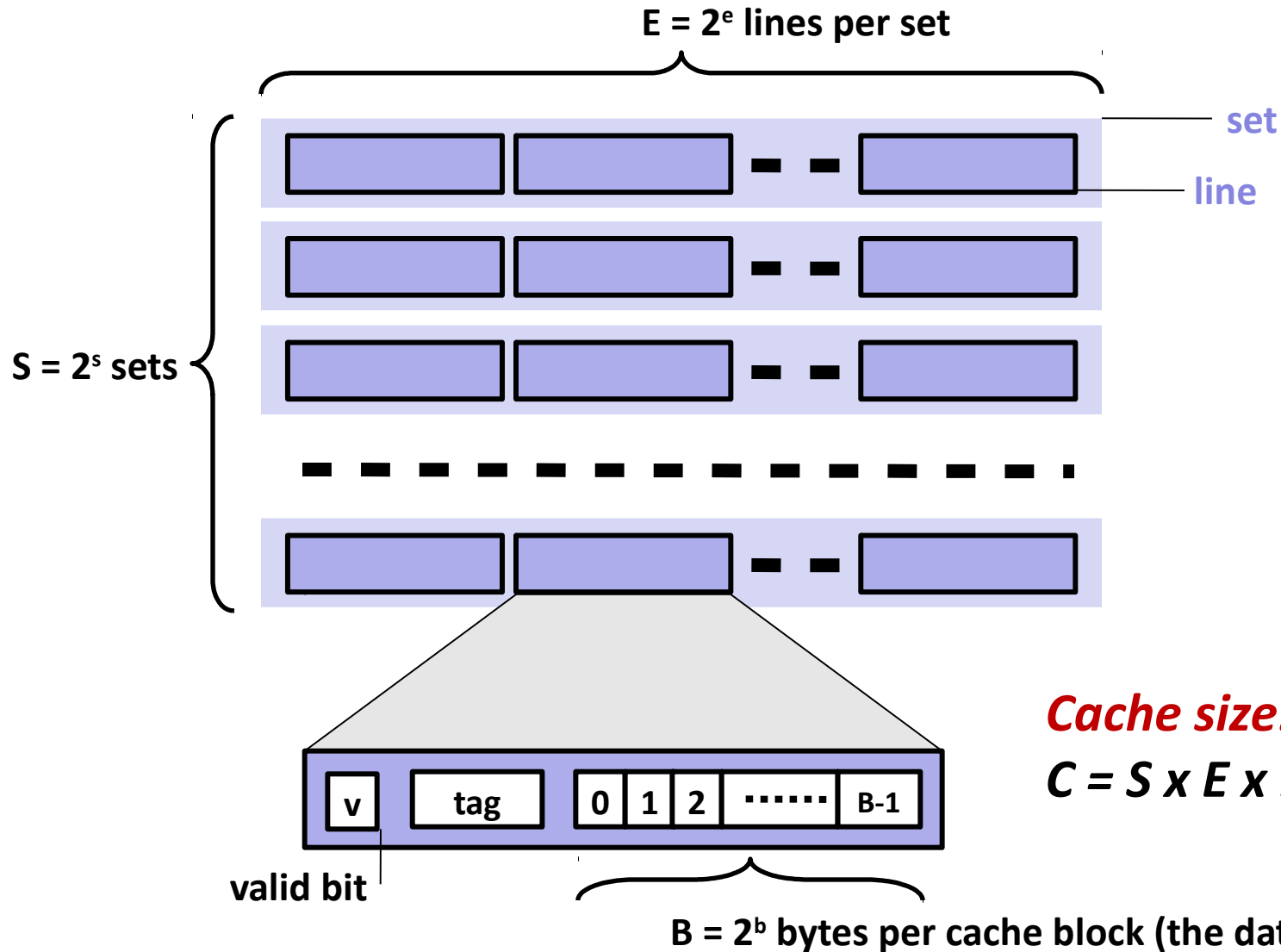


# Cache Memories

- **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 caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



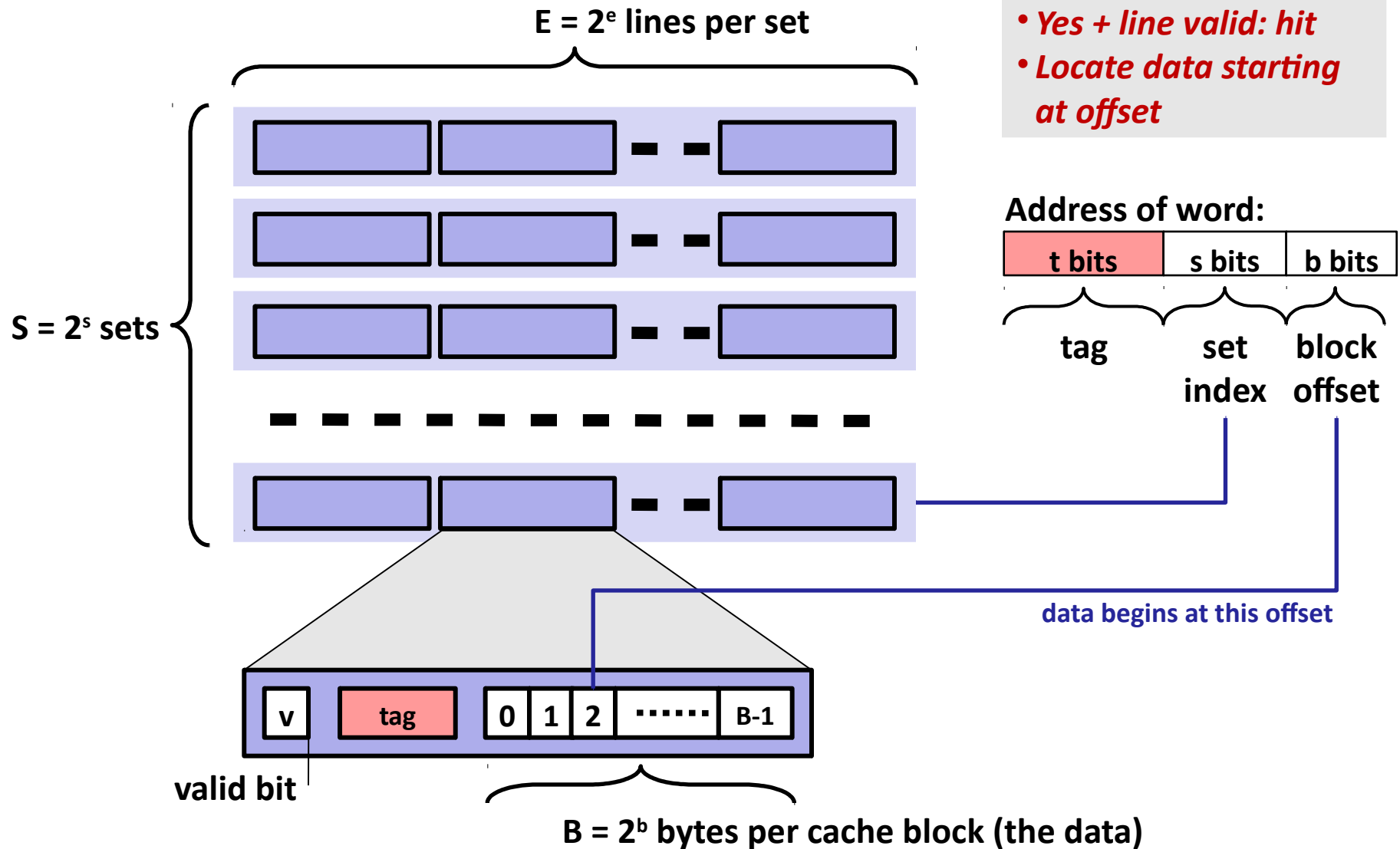
# General Cache Organization (S, E, B)



**Cache size:**

$$C = S \times E \times B \text{ data bytes}$$

# Cache Read

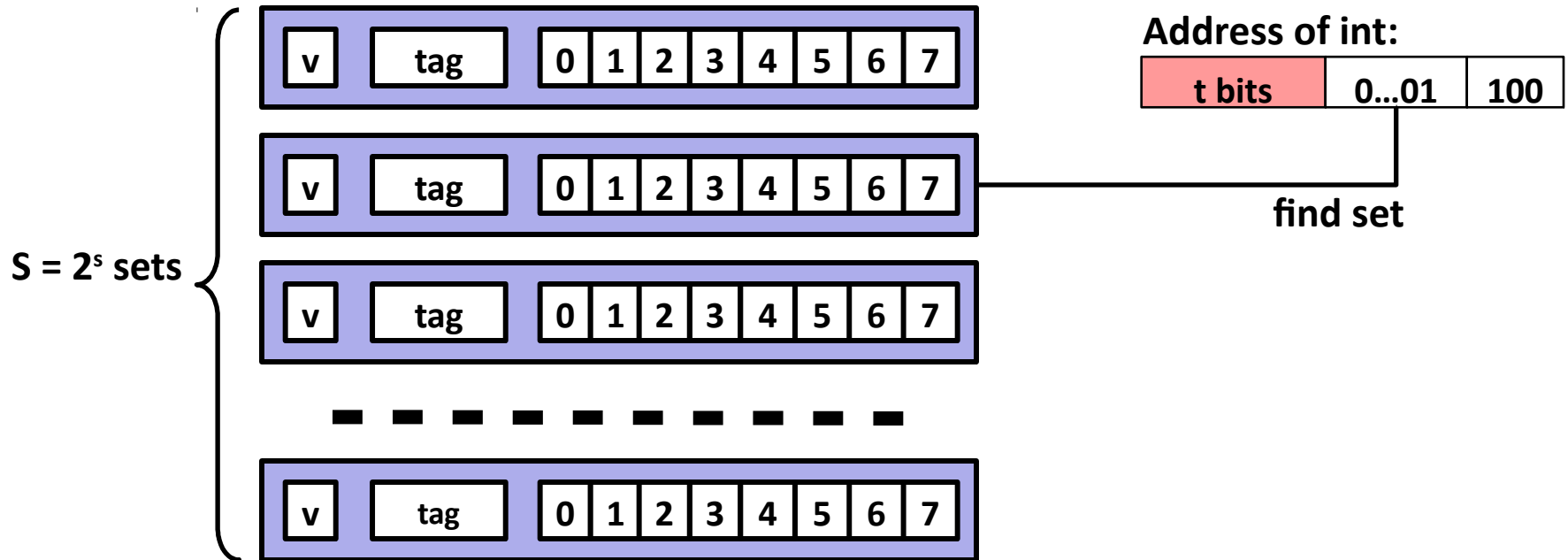


- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

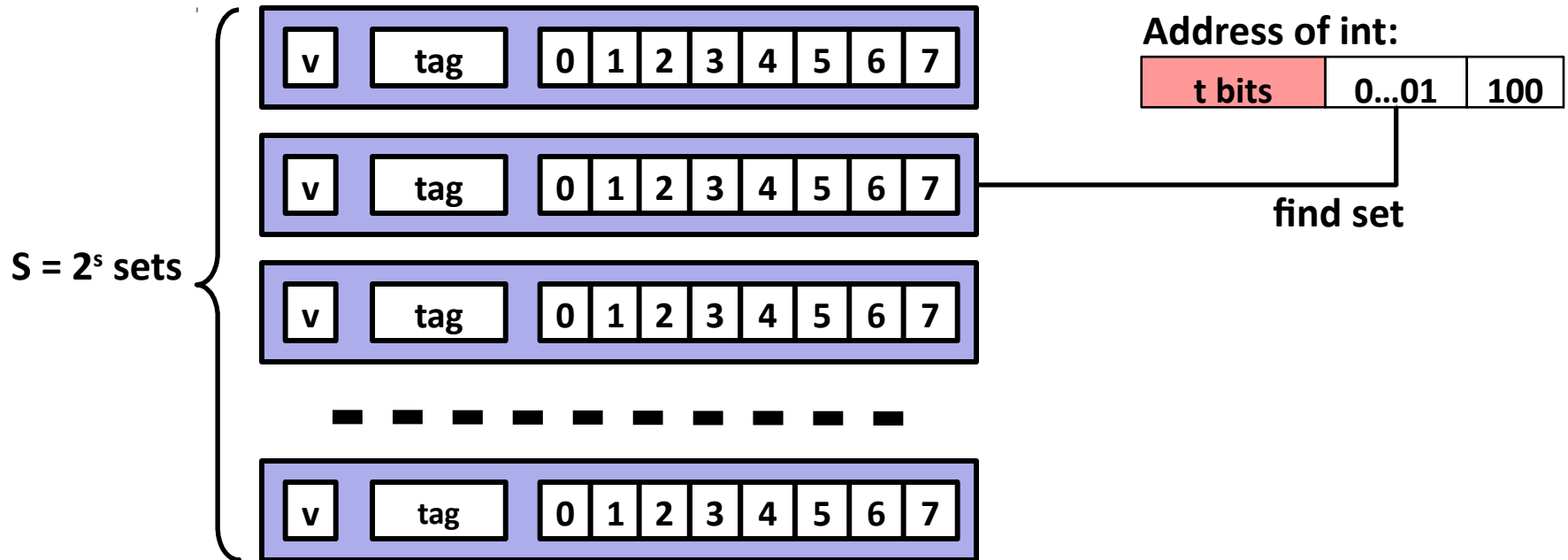
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

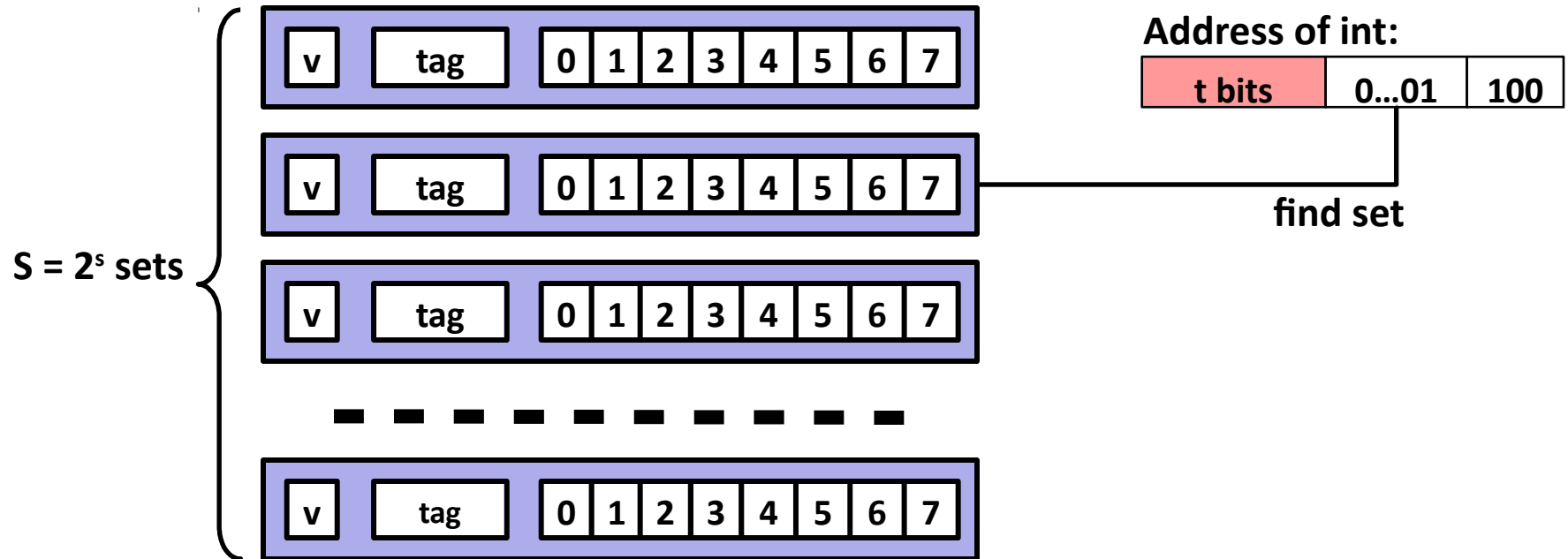
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size 8 bytes



# Direct-Mapped Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,  
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	[ <u>0000</u> <sub>2</sub> ],	miss
1	[ <u>0001</u> <sub>2</sub> ],	hit
7	[ <u>0111</u> <sub>2</sub> ],	miss
8	[ <u>1000</u> <sub>2</sub> ],	miss
0	[ <u>0000</u> <sub>2</sub> ]	miss

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

# A Higher Level Example

*Ignore the variables sum, i, j*

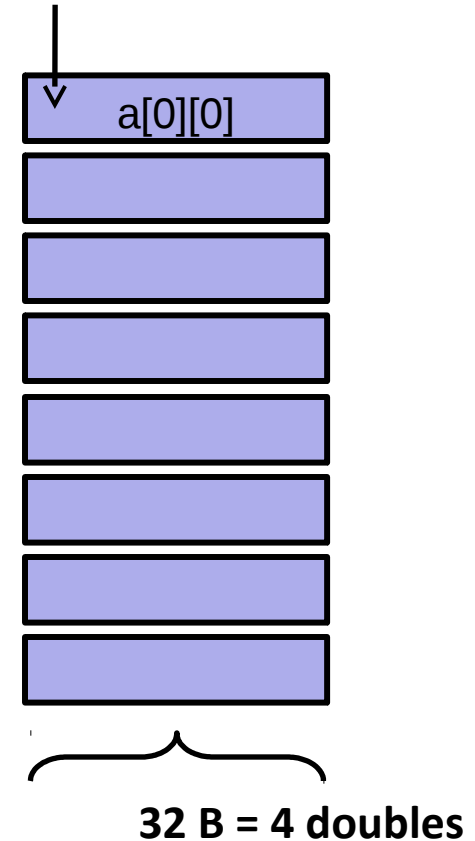
assume: cold (empty) cache,  
a[0][0] goes here

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```





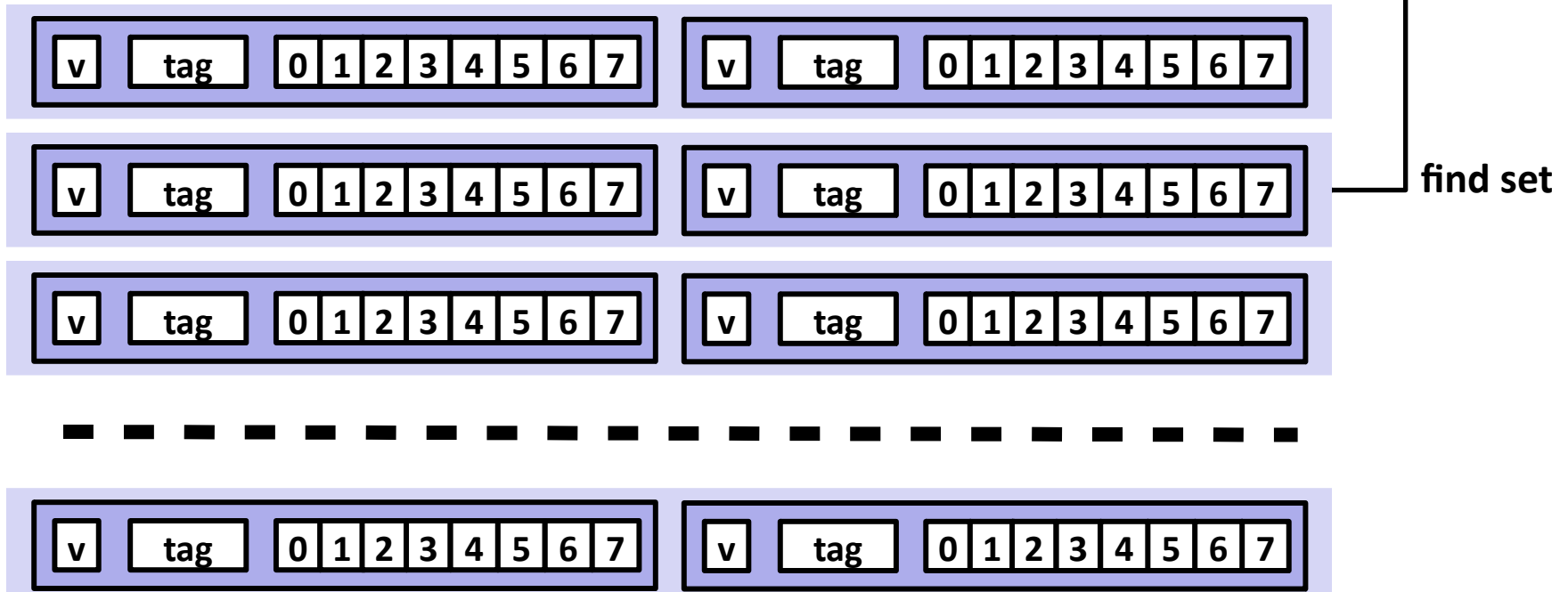
# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

t bits	0...01	100
--------	--------	-----



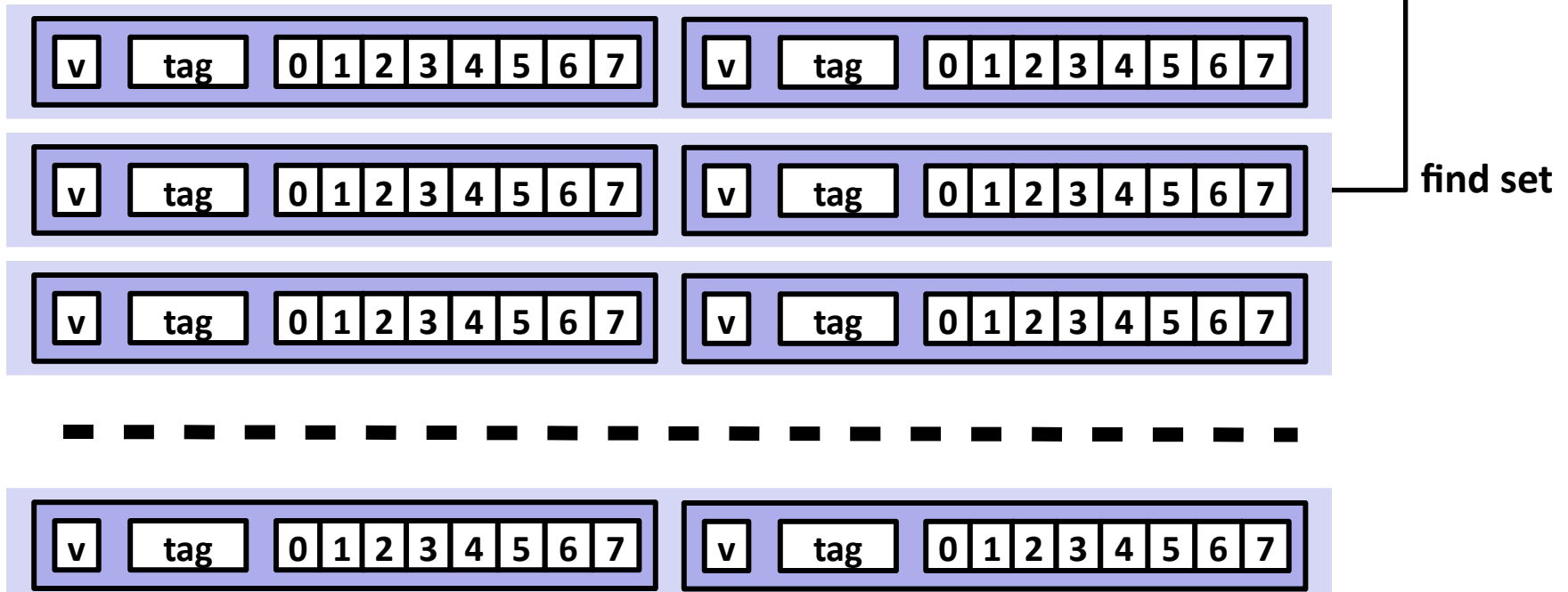
# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

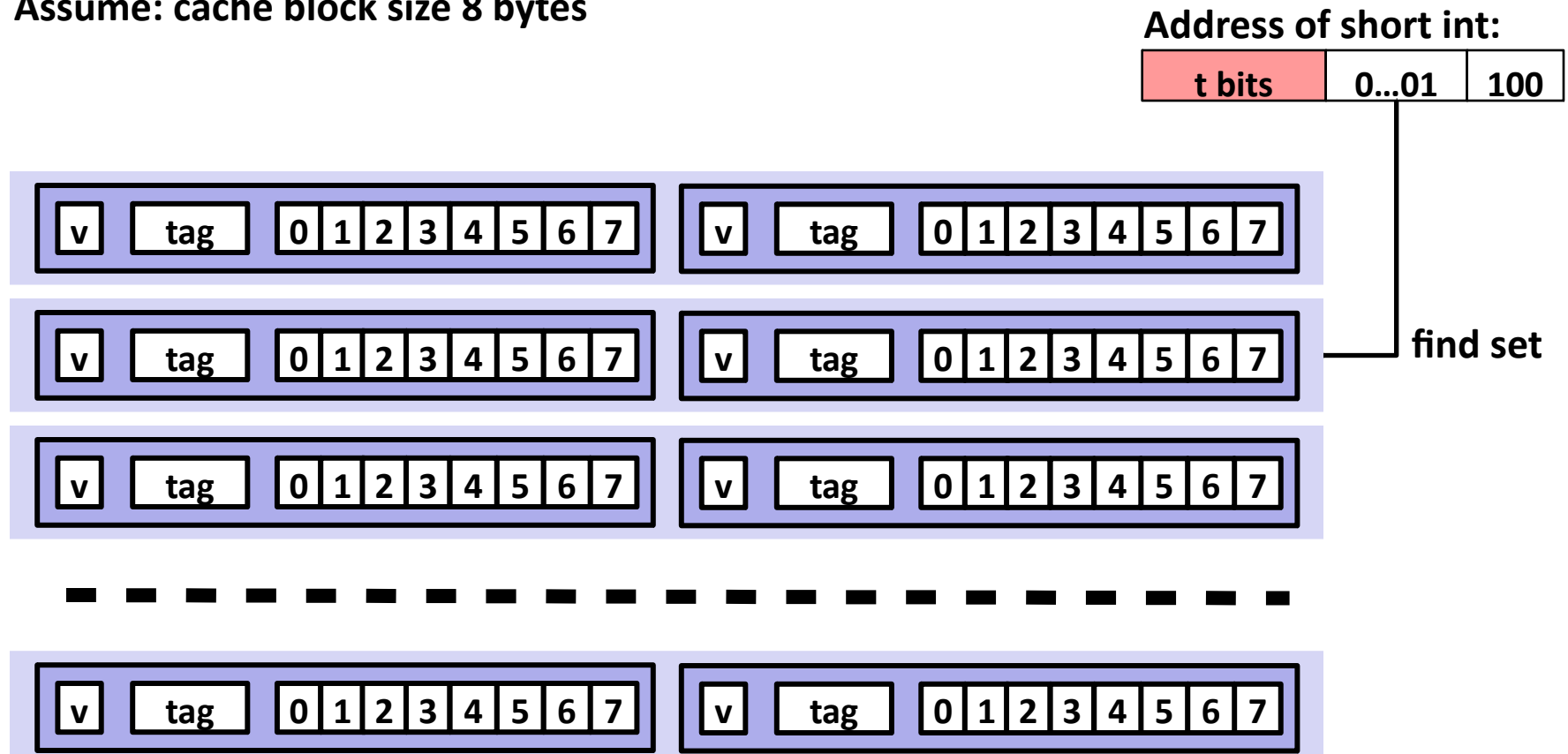
t bits	0...01	100
--------	--------	-----



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

E = 2: Two lines per set

Assume: cache block size 8 bytes



# 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 <u>00</u> <sub>2</sub> ],	miss
1	[00 <u>01</u> <sub>2</sub> ],	hit
7	[01 <u>11</u> <sub>2</sub> ],	miss
8	[10 <u>00</u> <sub>2</sub> ],	miss
0	[00 <u>00</u> <sub>2</sub> ]	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, 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?

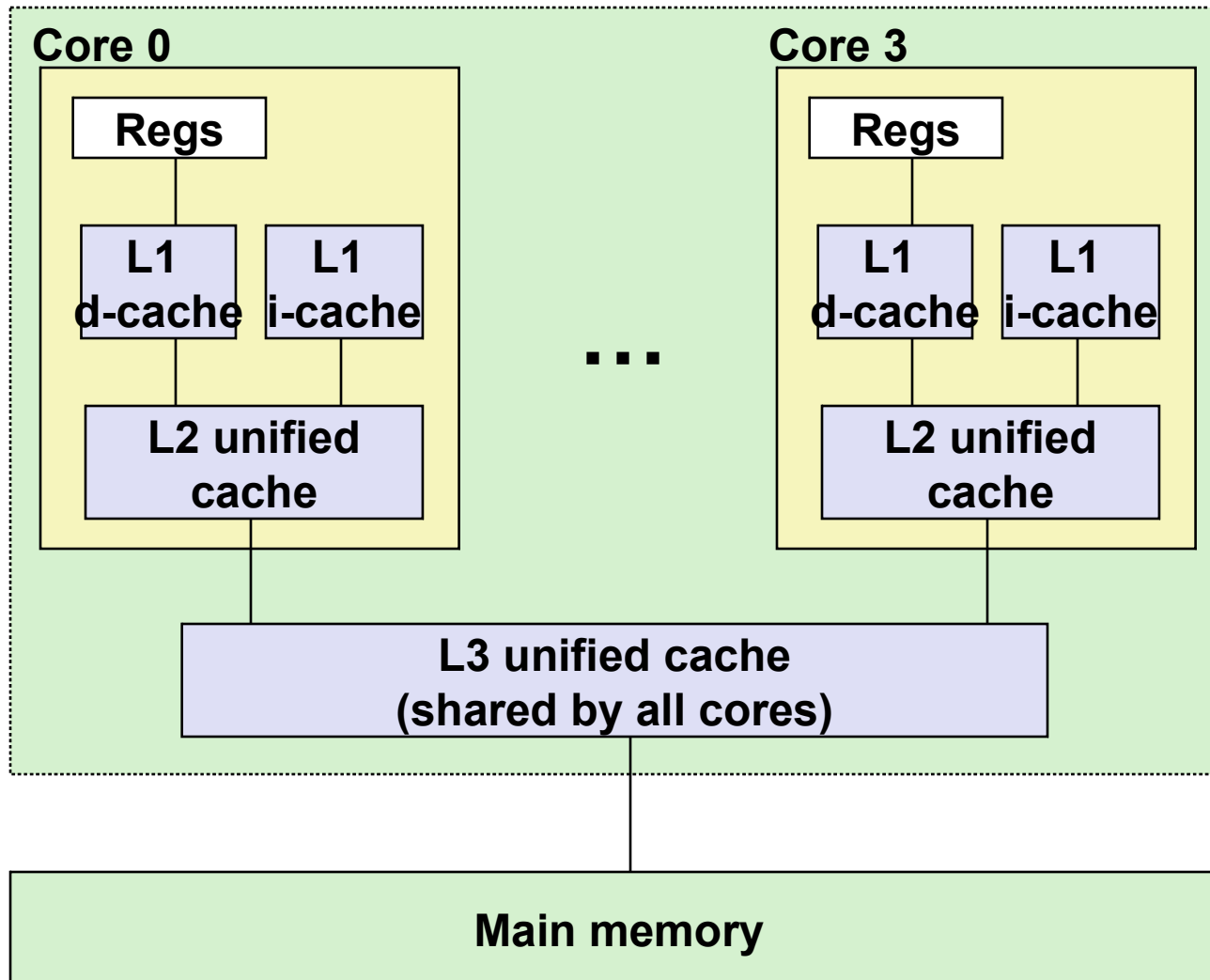
- **Write-allocate** (load into cache, update line in cache)
  - Good if more writes to the location follow
- **No-write-allocate** (writes immediately to memory)

## ■ 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: 11 cycles

### L3 unified cache:

8 MB, 16-way,  
Access: 30-40 cycles

**Block size:** 64 bytes  
for all caches.

# Cache Performance Metrics

## ■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)  
=  $1 - \text{hit rate}$
- 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:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

## ■ Miss Penalty

- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)

# Lets 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:  $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$   
99% hits:  $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$

- **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**)

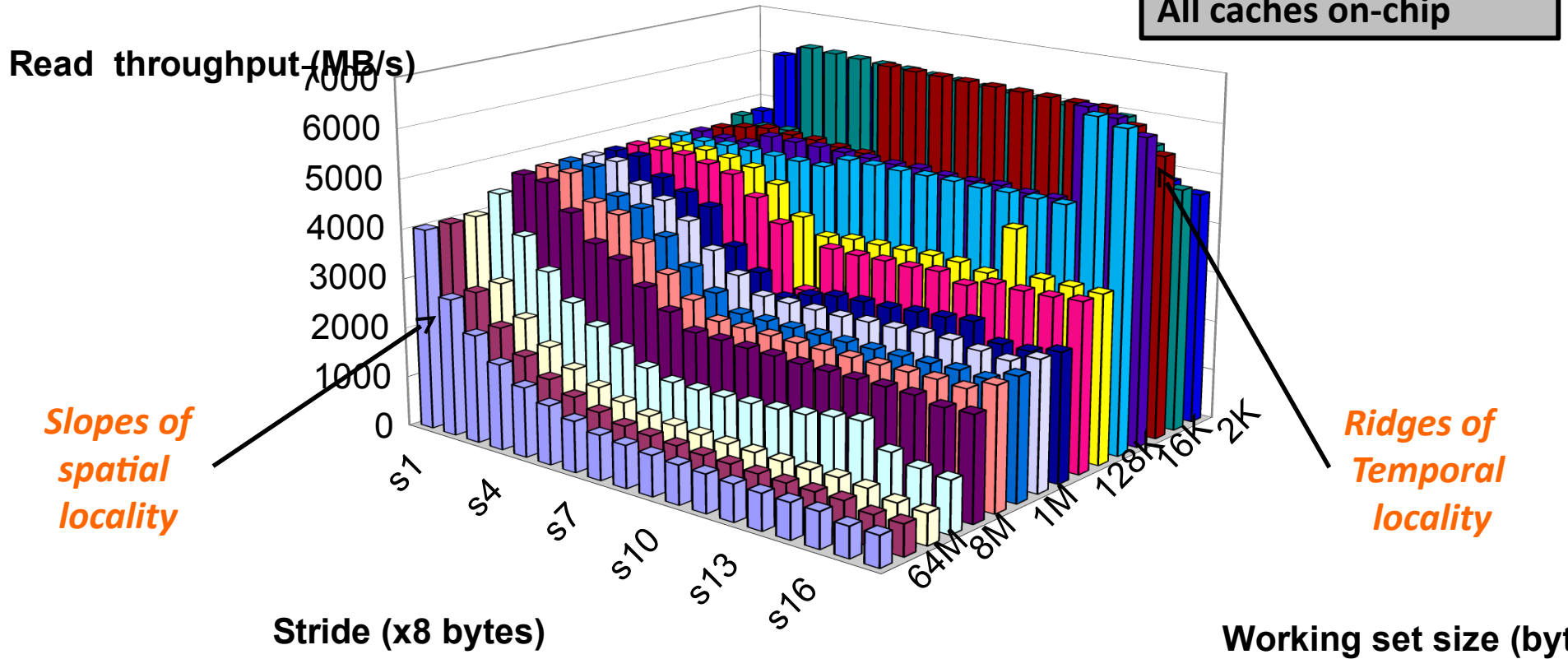
**Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.**

# 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.

# The Memory Mountain

Intel Core i7  
32 KB L1 i-cache  
32 KB L1 d-cache  
256 KB unified L2 cache  
8M unified L3 cache  
  
All caches on-chip



# Concluding Observations

- **Programmer can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- **All systems favor “cache friendly code”**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)