

18-600 Foundations of Computer Systems

Lecture 11: "Cache Memories & Non-Volatile Storage"

John P. Shen & Gregory Kesden
October 4, 2017

- Required Reading Assignment:
- Chapter 6 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron



18-600 Foundations of Computer Systems

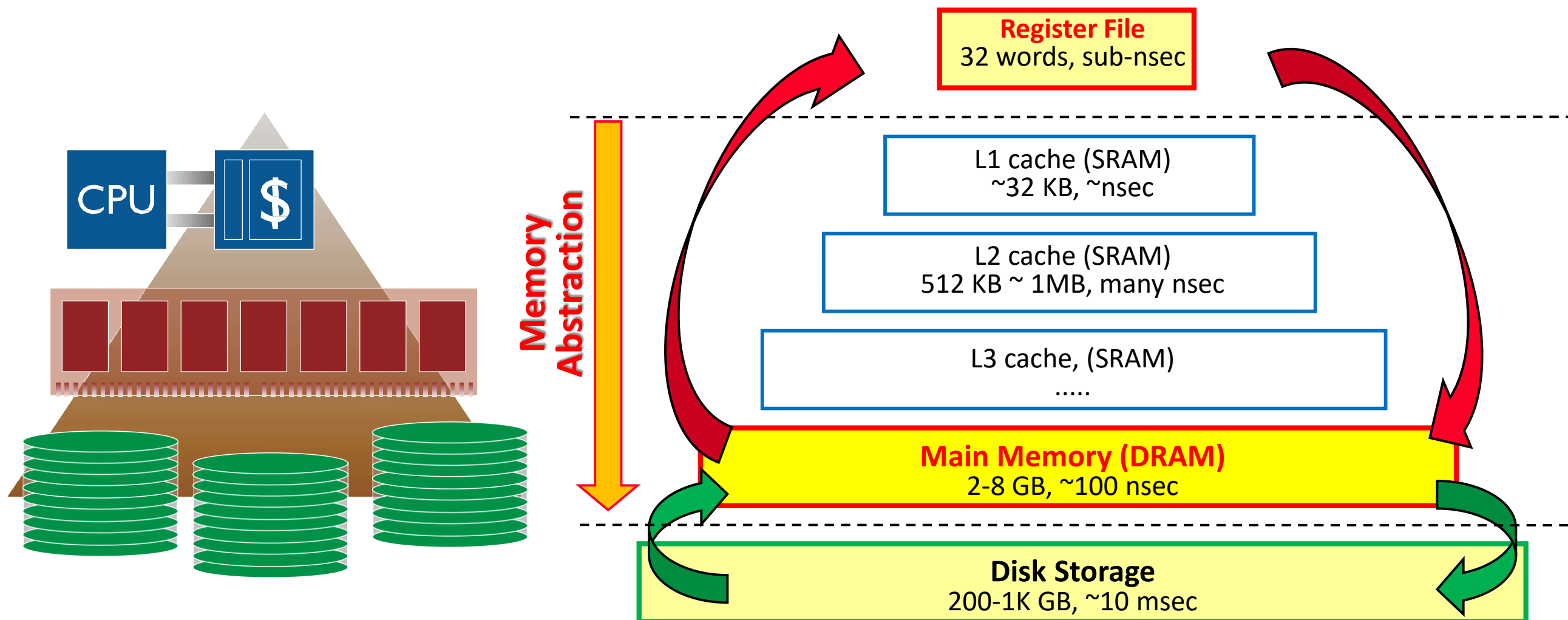
Lecture 11:

"Cache Memories & Non-Volatile Storage"

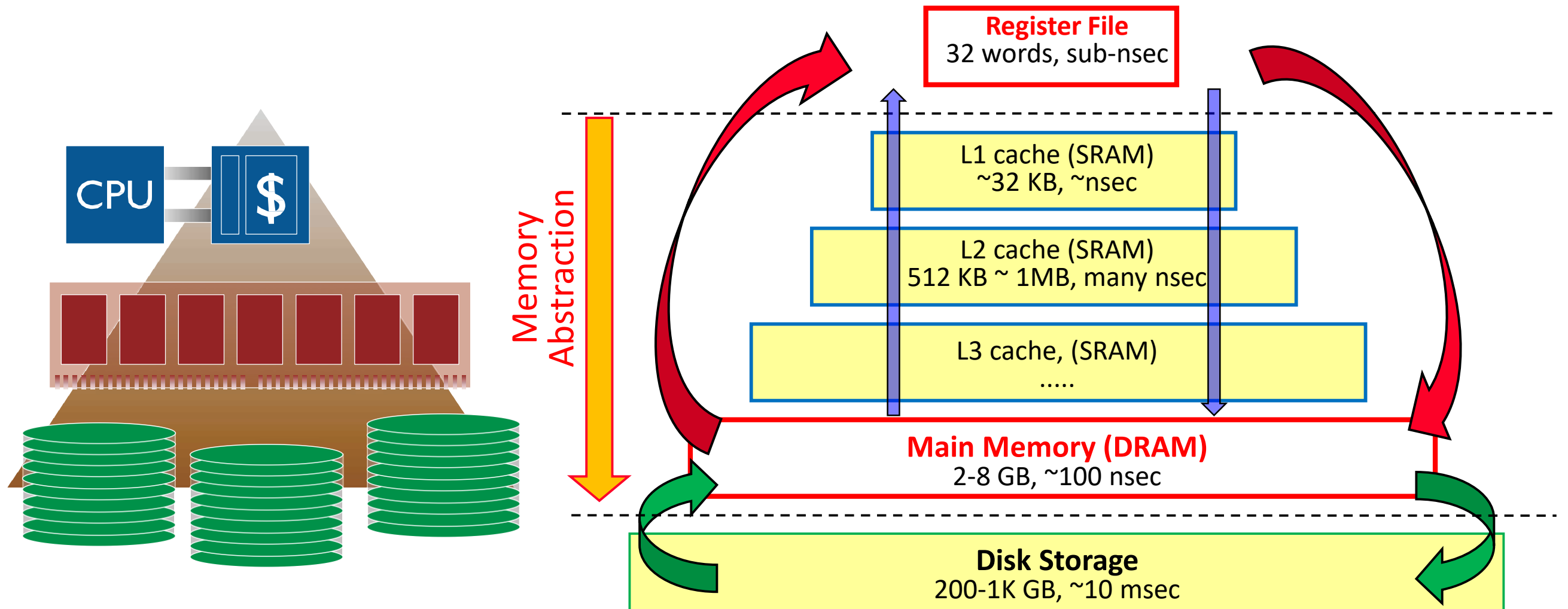
- A. Cache Organization and Operation
- B. Performance Impact of Caches
 - a. The Memory Mountain
 - b. Rearranging Loops to Improve Spatial Locality
 - c. Using Blocking to Improve Temporal Locality
- C. Non-Volatile Storage Technologies
 - a. Disk Storage Technology
 - b. Flash Memory Technology



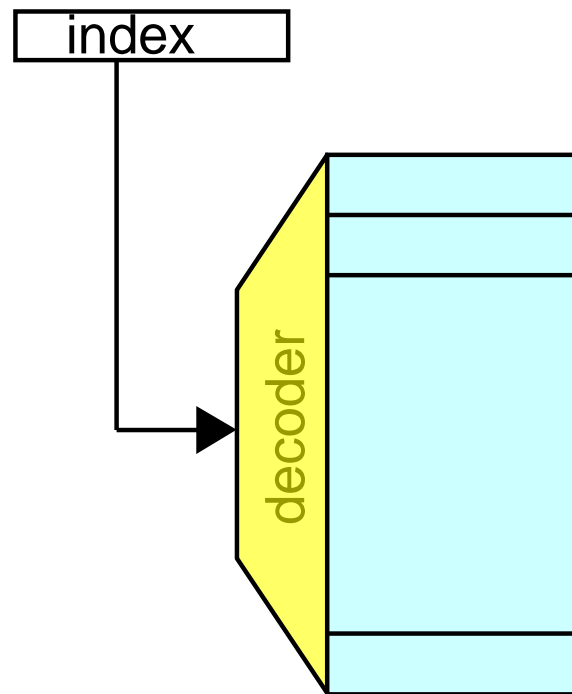
Memory Hierarchy (where do all the bits live?)



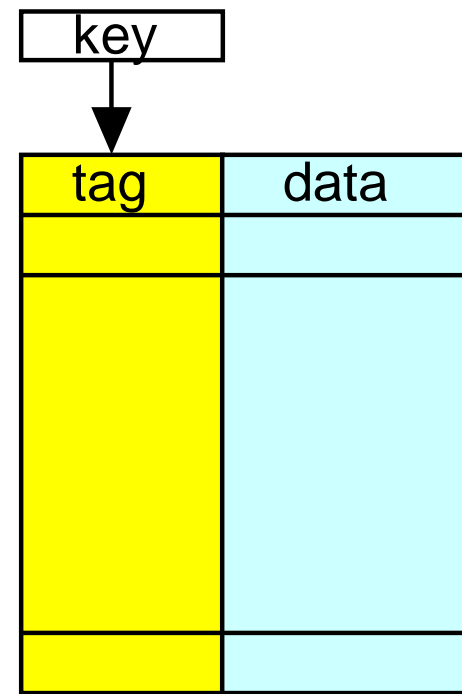
Memory Hierarchy (where do all the bits live?)



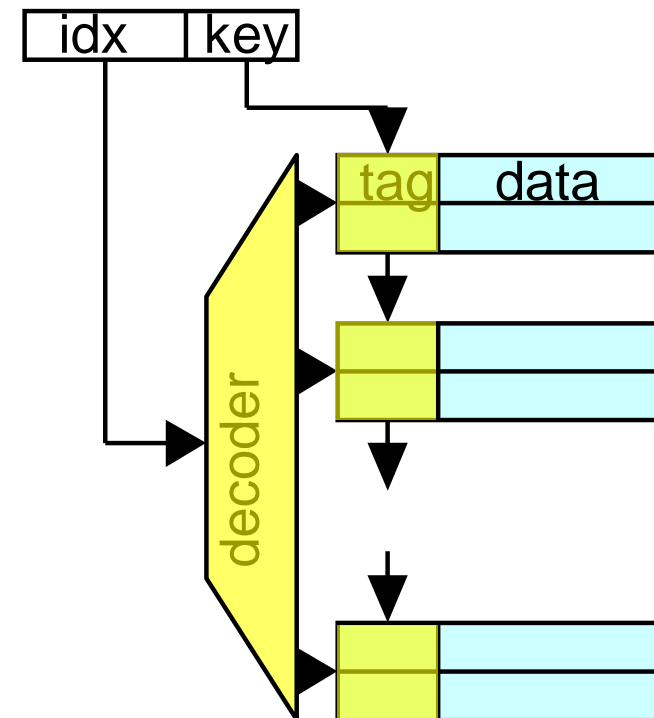
(Cache) Memory Implementation Options



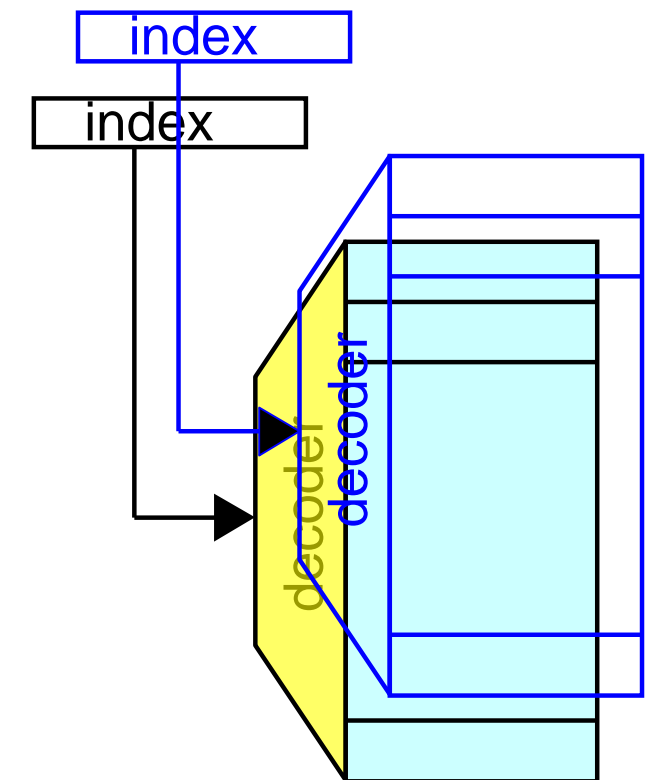
Indexed Memory
(RAM)
k-bit index
 2^k blocks



Associative Memory
(CAM)
no index
unlimited blocks

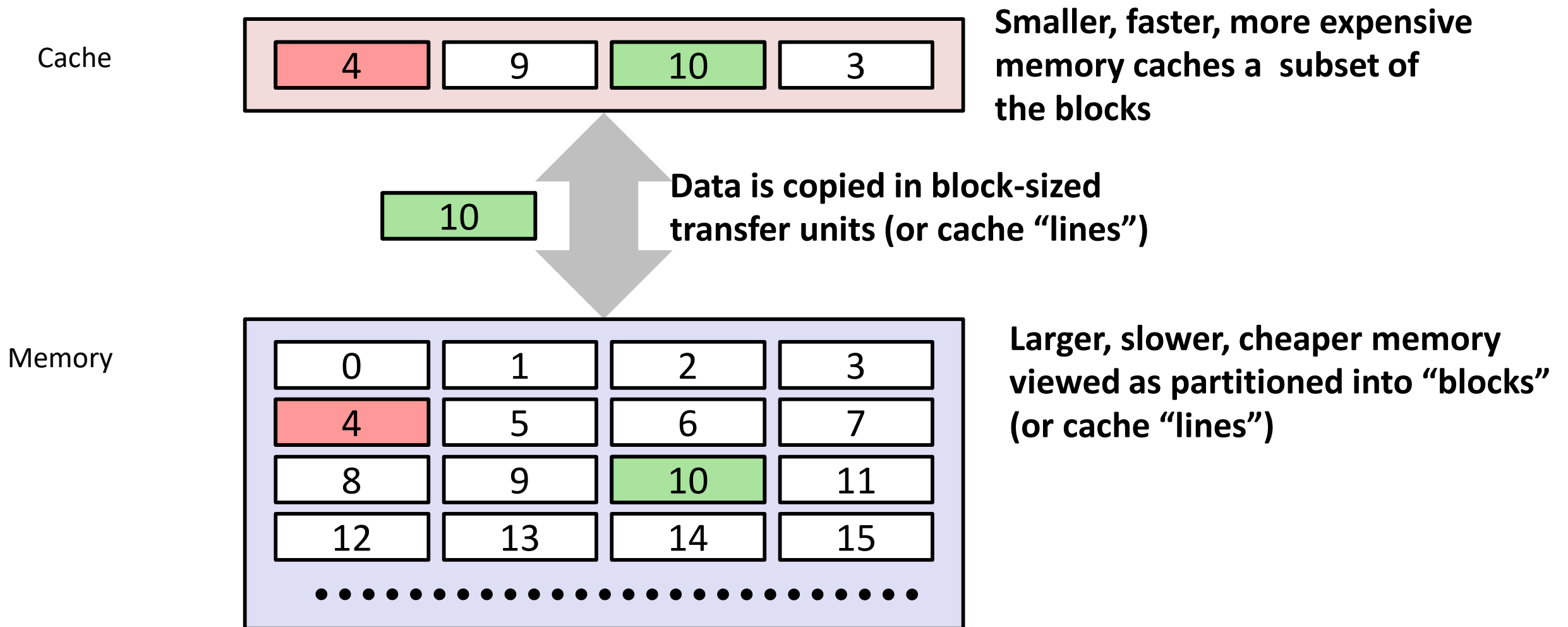


N-Way
Set-Associative Memory
k-bit index
 $2^k \cdot N$ blocks

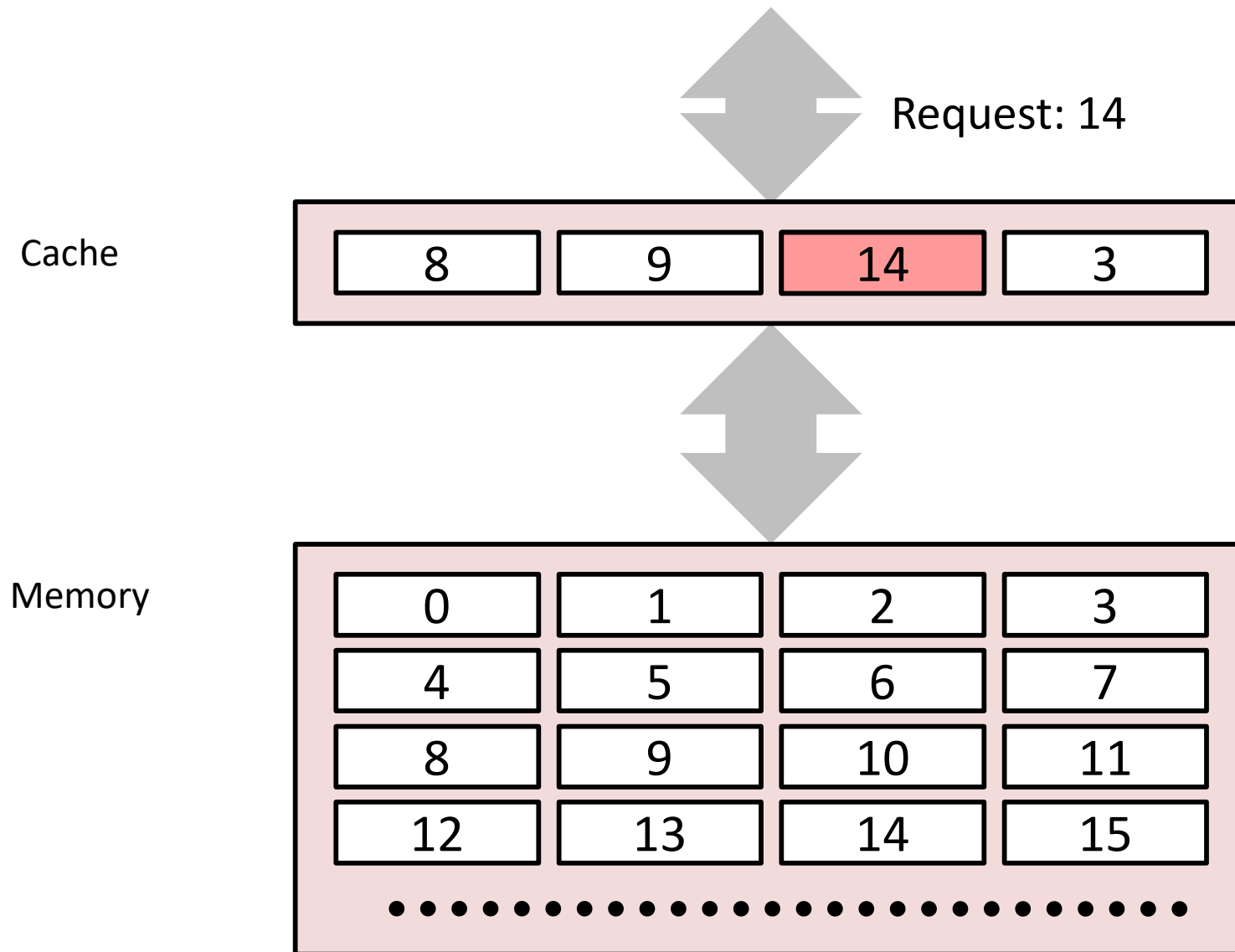


Indexed Memory
(Multi-Ported)
(2x) k-bit index
(2x) 2^k blocks

General Cache Concept



General Cache Concepts: Hit

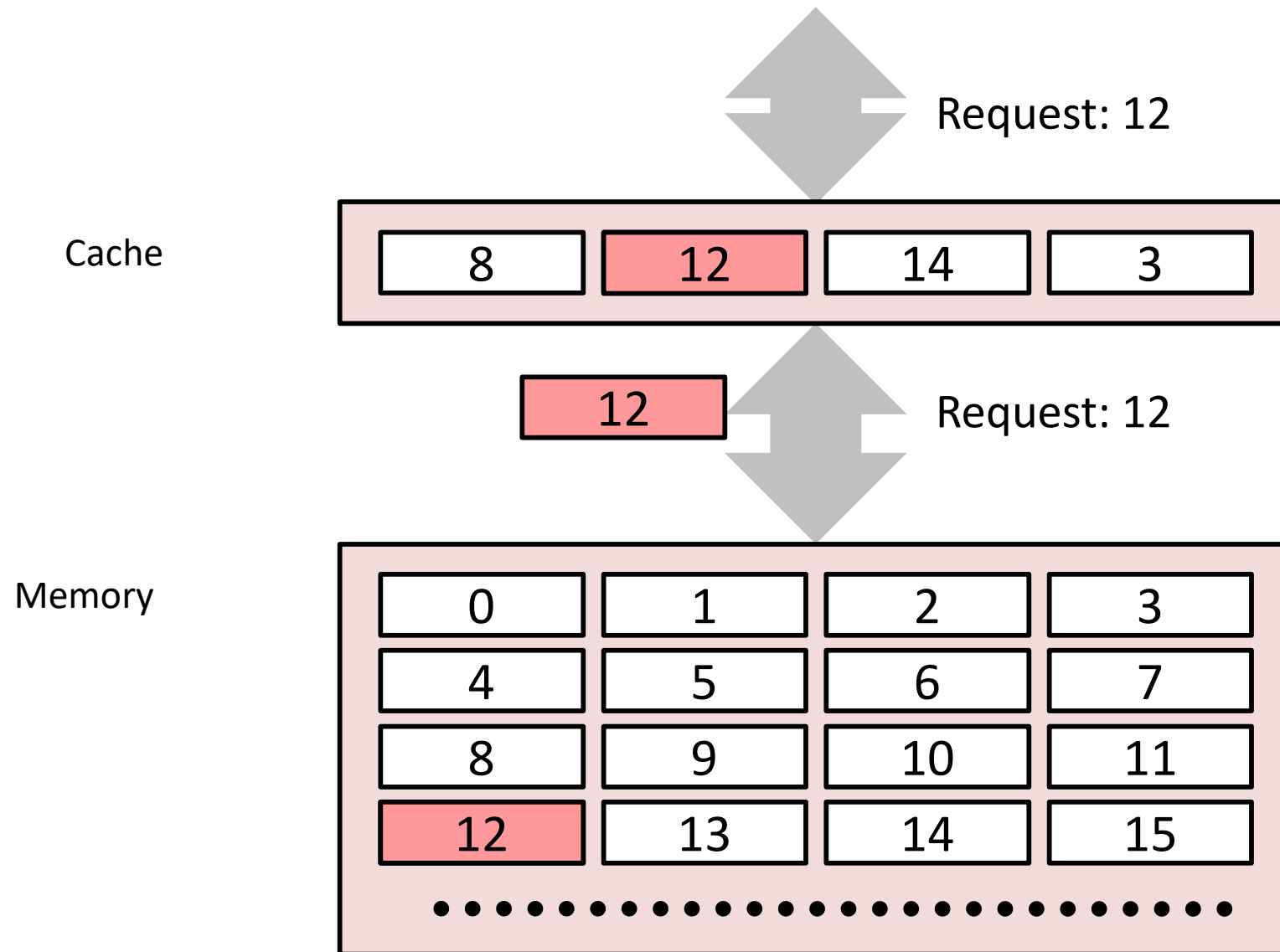


Data in block b is needed

Block b is in cache:

Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

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

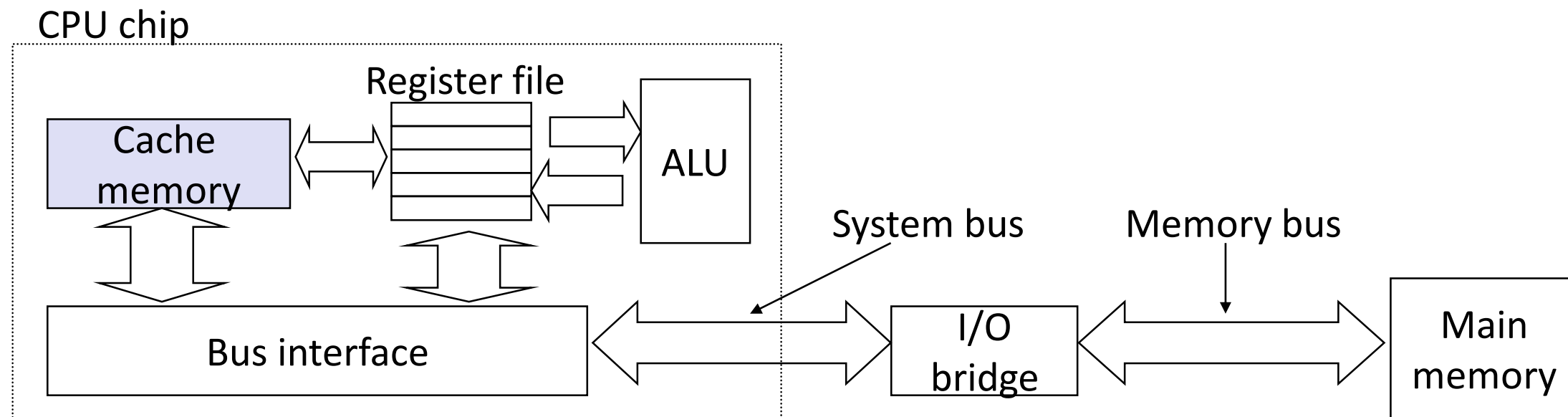
General Caching Concepts:

Types of Cache Misses (3 C's)

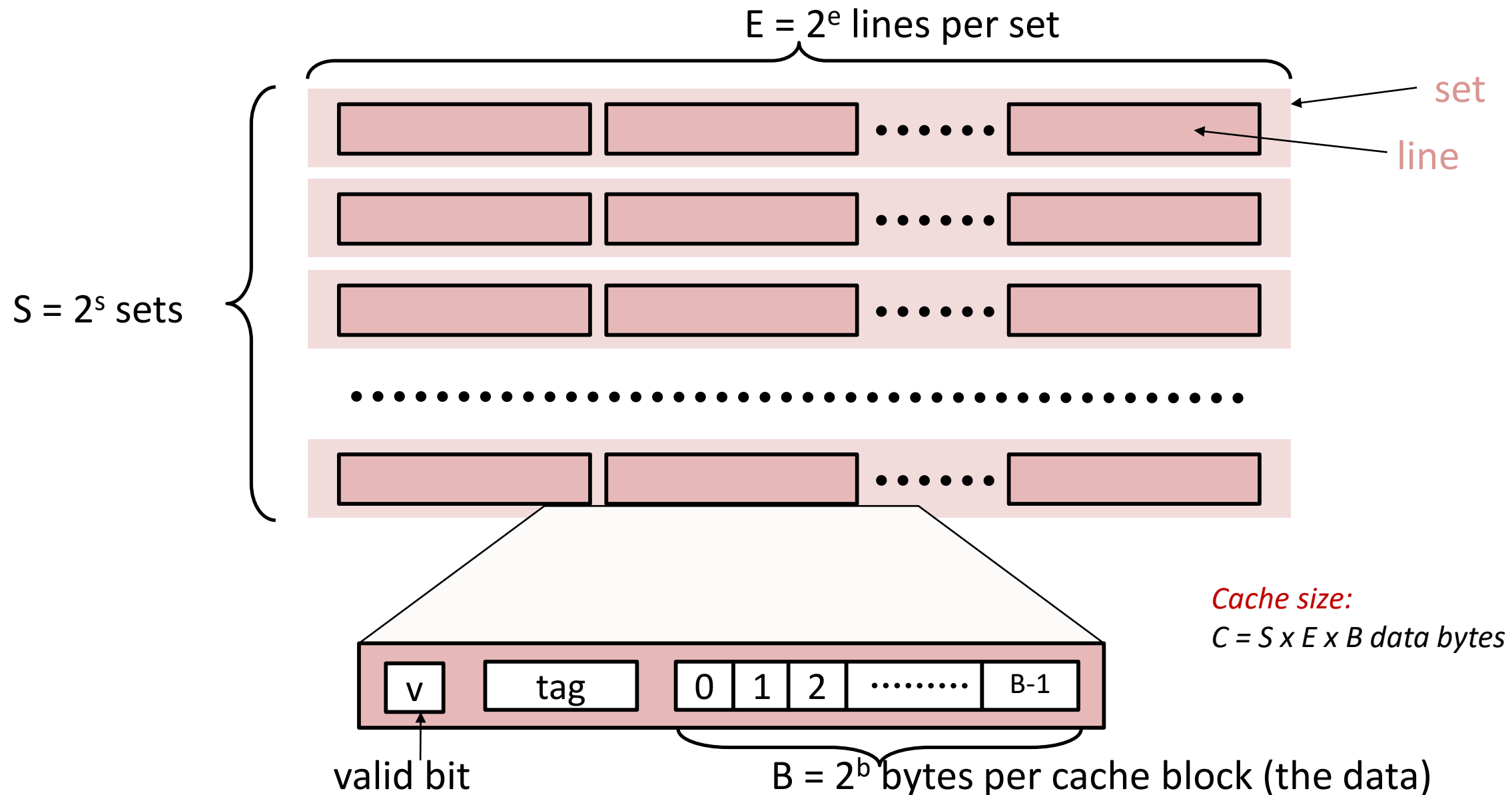
- **Cold (compulsory) miss**
 - Cold misses occur because the cache is empty.
- **Capacity miss**
 - Occurs when the set of active cache blocks (**working set**) is larger than the cache.
- **Conflict miss**
 - Occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

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:

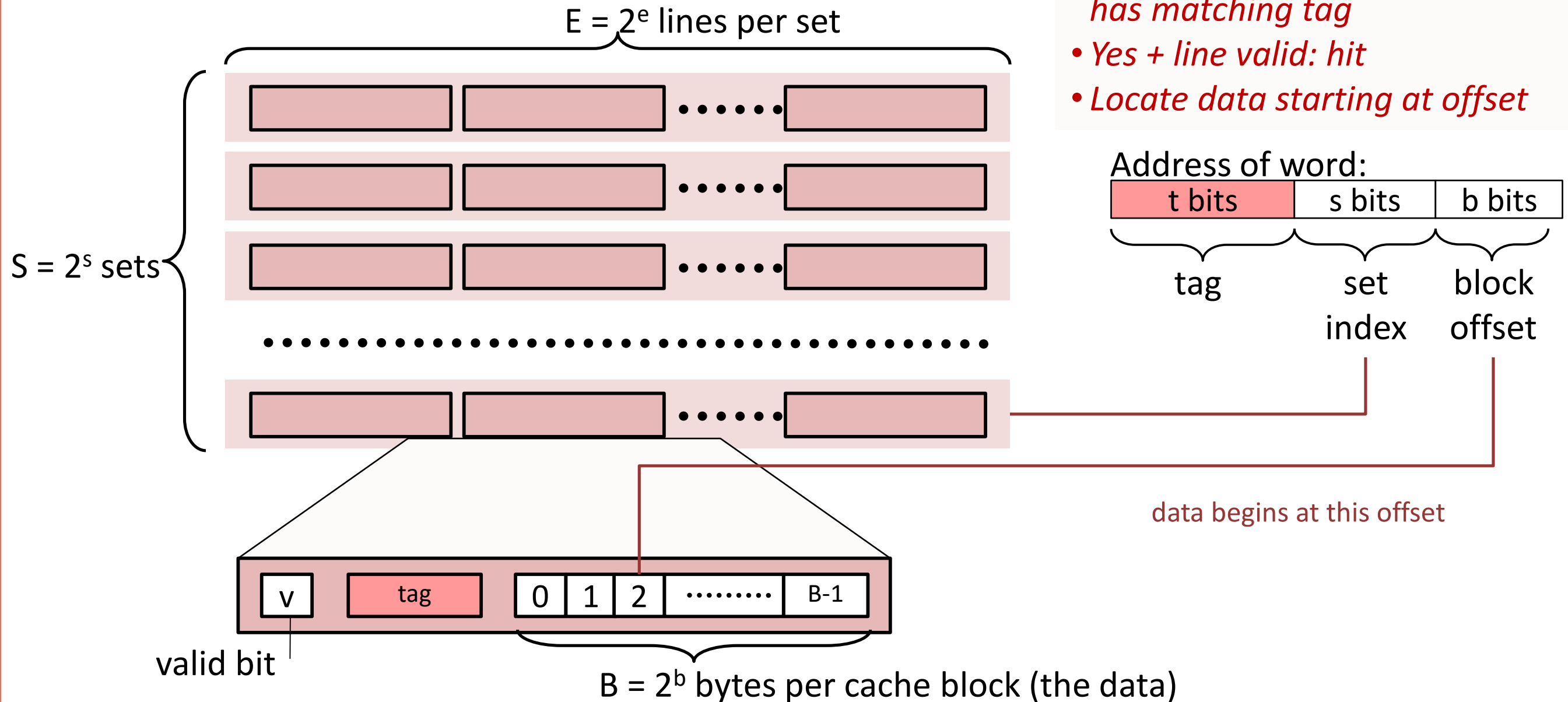


General Cache Organization (S, E, B)



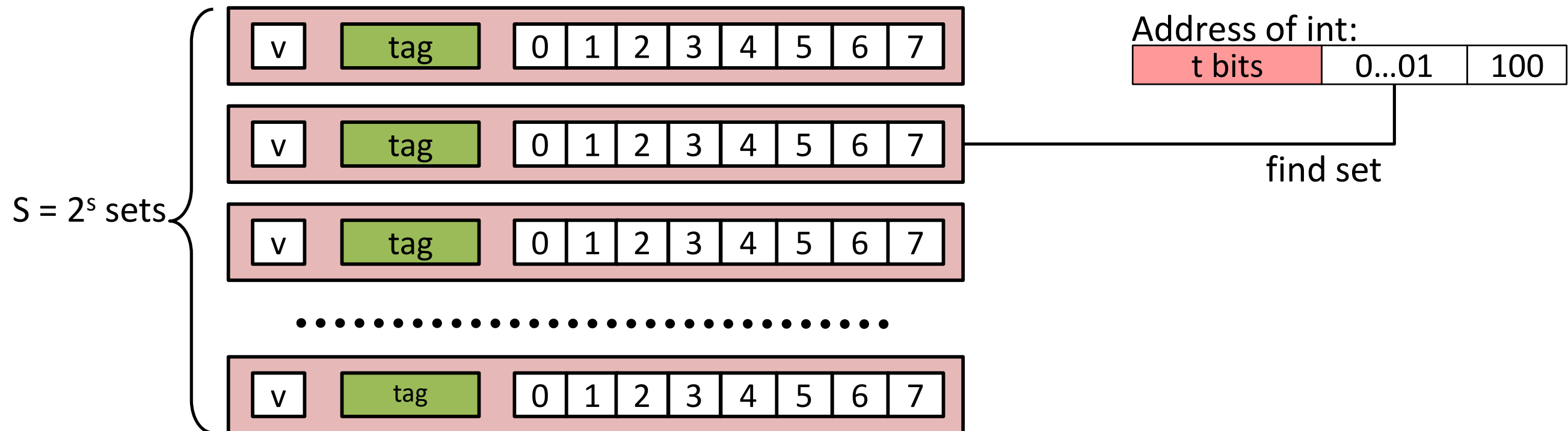
Cache Read

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



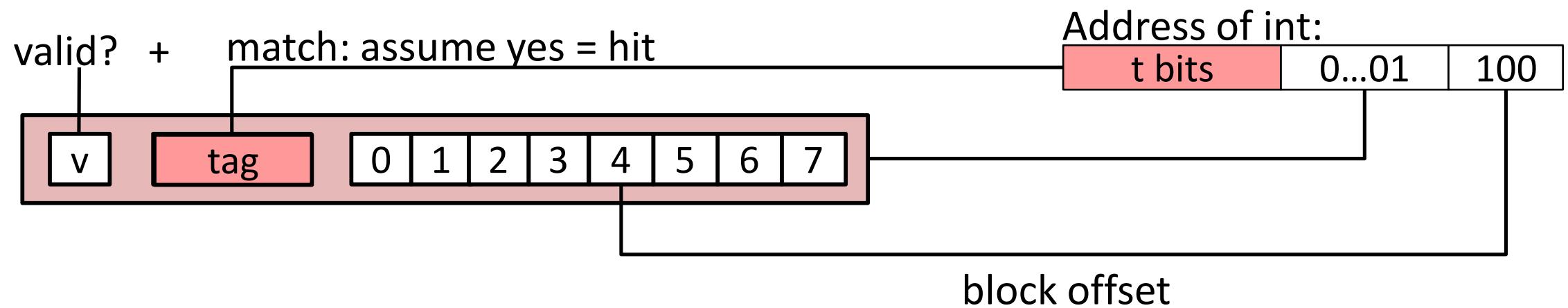
Example: Direct Mapped Cache ($E = 1$)

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



Example: Direct Mapped Cache (E = 1)

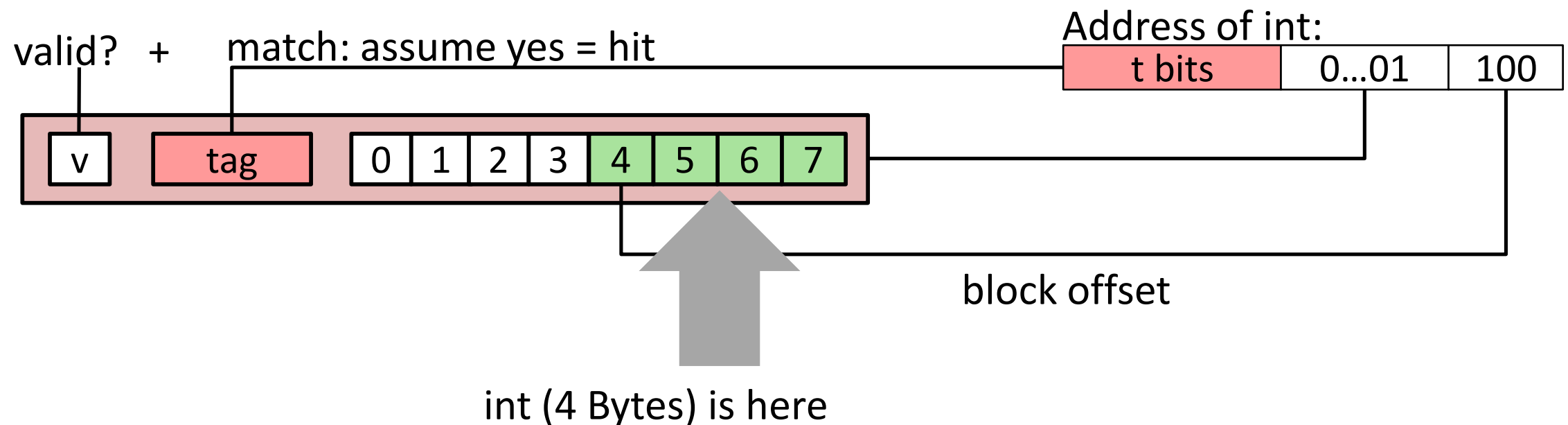
Direct mapped: One line per set
 Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

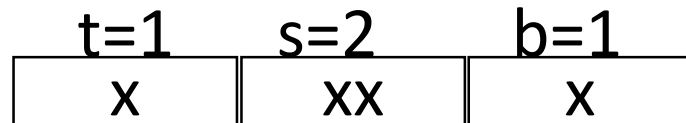
Direct mapped: One line per set

Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation



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

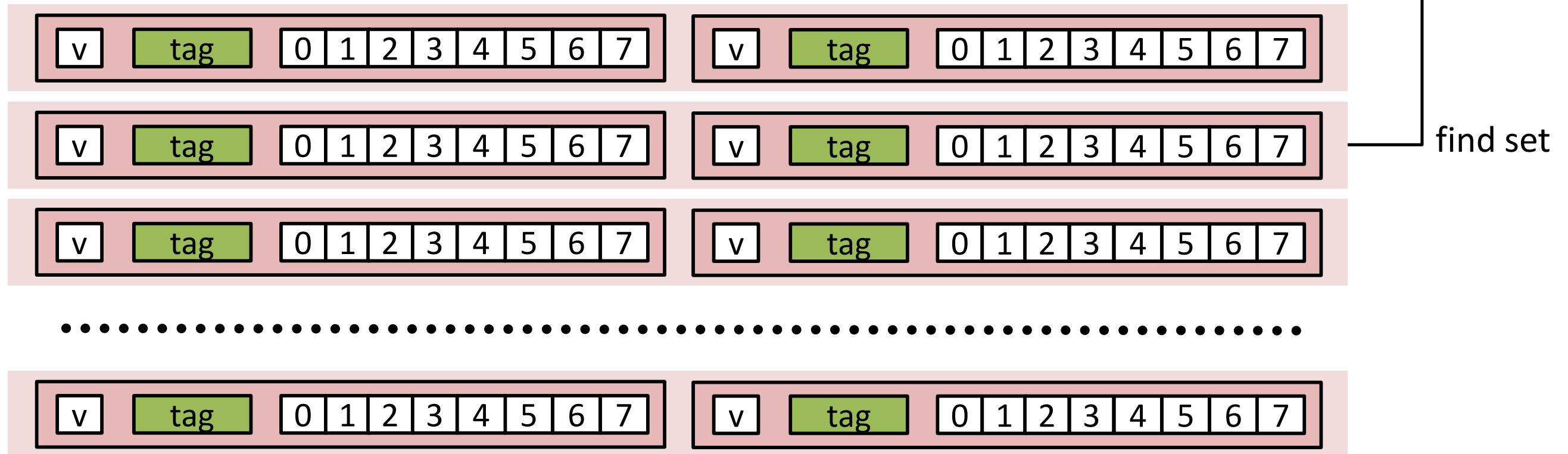
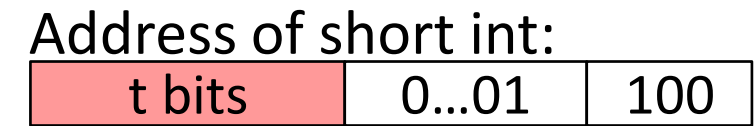
Address trace (reads, one byte per read):

0	[0000 ₂],	miss
1	[0001 ₂],	hit
7	[0111 ₂],	miss
8	[1000 ₂],	miss
0	[0000 ₂]	miss

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

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

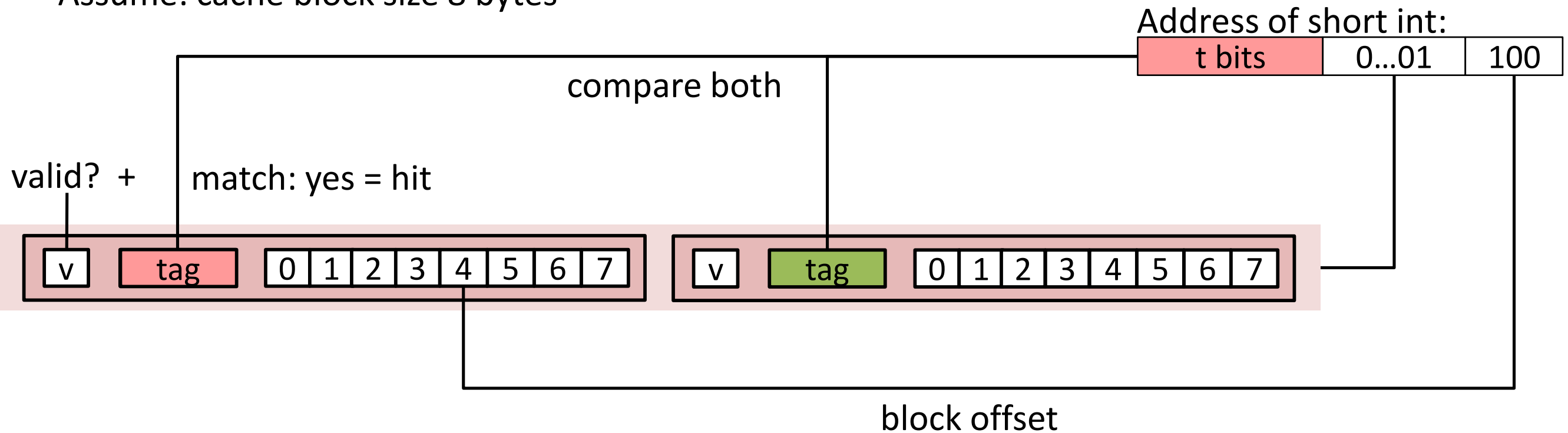
E = 2: Two lines per set
 Assume: cache block size 8 bytes



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

E = 2: Two lines per set

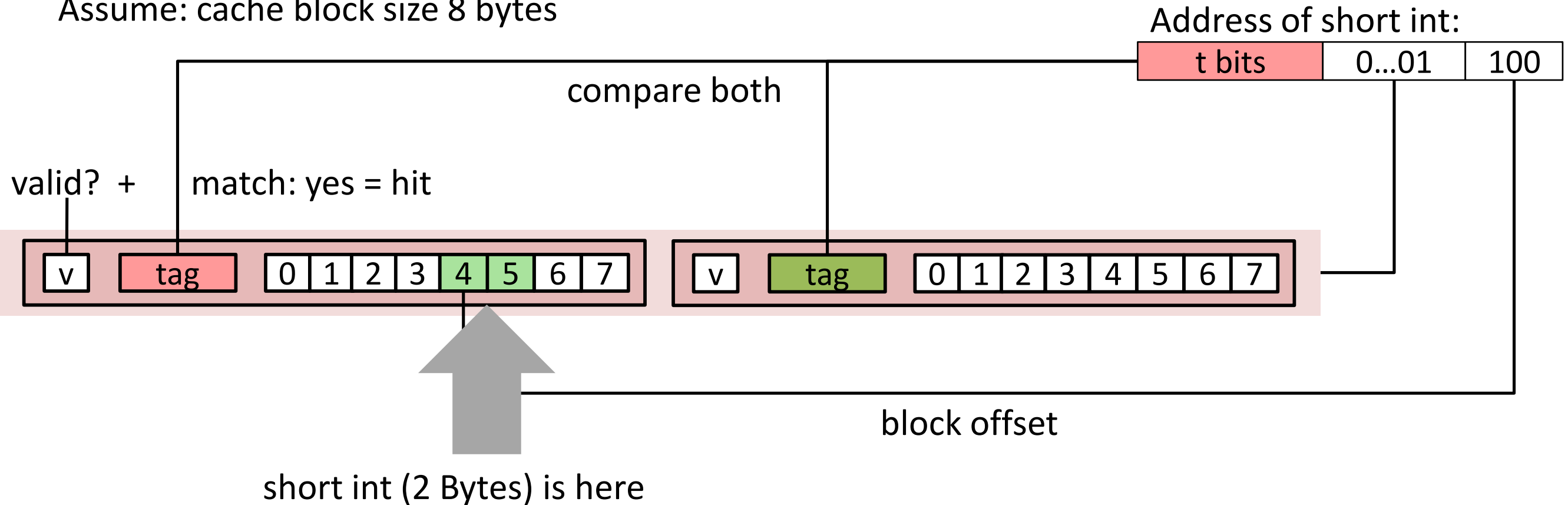
Assume: cache block size 8 bytes



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

$E = 2$: Two lines per set

Assume: cache block size 8 bytes



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):

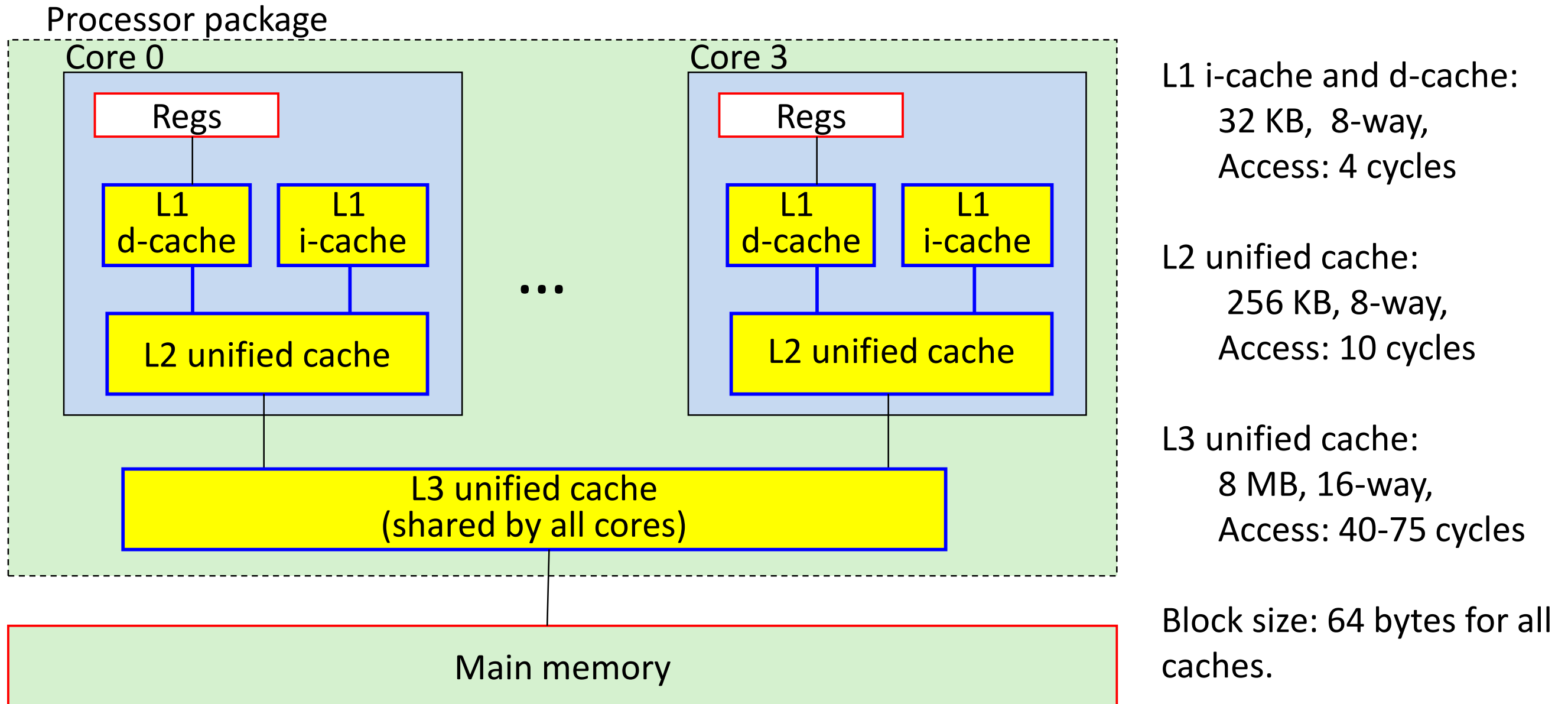
0	[00 <u>0</u> 0] ₂ ,	miss
1	[00 <u>0</u> 1] ₂ ,	hit
7	[01 <u>1</u> 1] ₂ ,	miss
8	[10 <u>0</u> 0] ₂ ,	miss
0	[00 <u>0</u> 0] ₂	hit

	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



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

18-600 Foundations of Computer Systems

Lecture 11:

"Cache Memories & Non-Volatile Storage"

- A. Cache Organization and Operation
- B. Performance Impact of Caches
 - a. The Memory Mountain
 - b. Rearranging Loops to Improve Spatial Locality
 - c. Using Blocking to Improve Temporal Locality
- C. Non-Volatile Storage Technologies
 - a. Disk Storage Technology
 - b. Flash Memory Technology



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`:

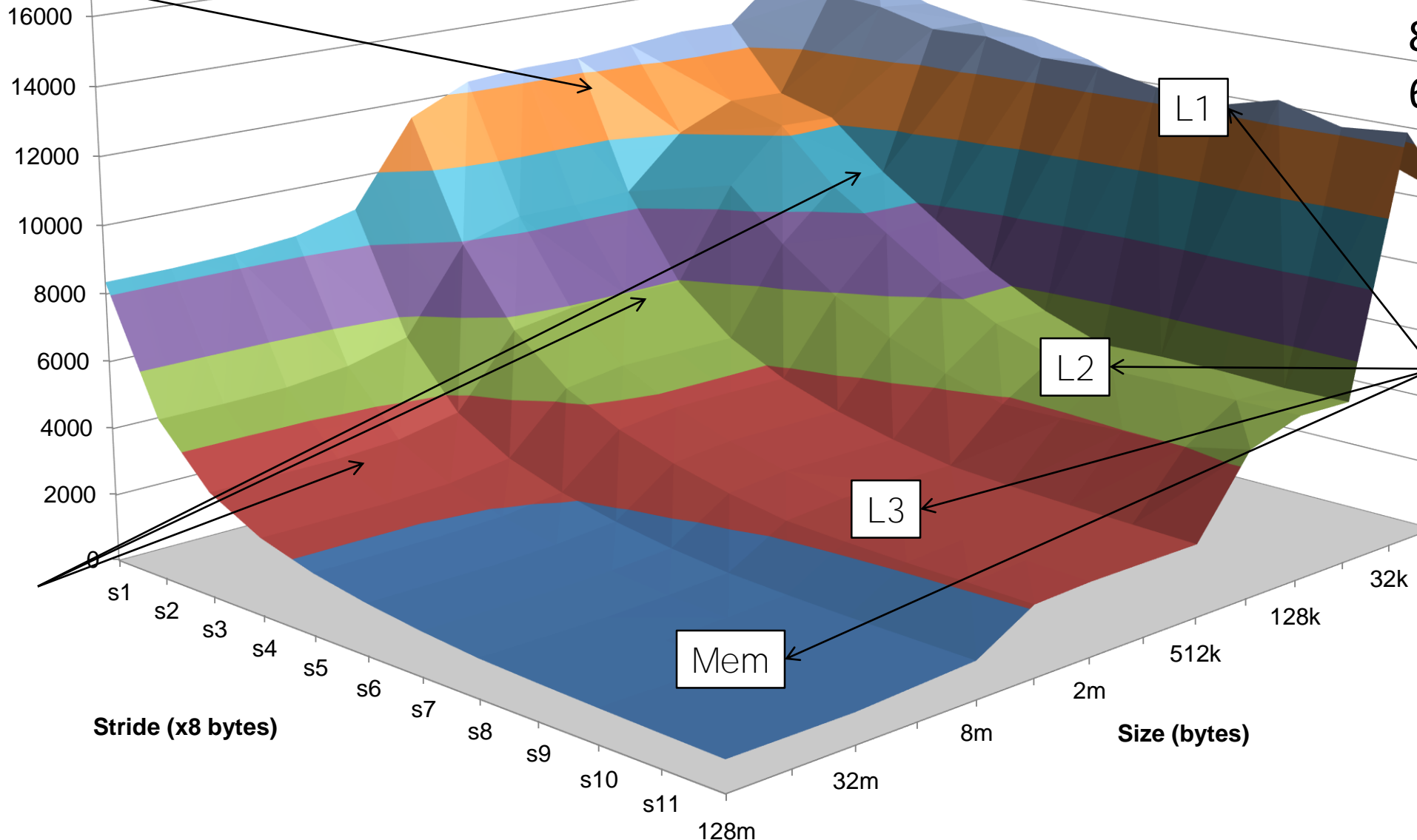
1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput (MB/s)

The Memory Mountain

Core i7 Haswell
 2.1 GHz
 32 KB L1 d-cache
 256 KB L2 cache
 8 MB L3 cache
 64 B block size

Aggressive prefetching

Read throughput (MB/s)



Ridges of temporal locality

Slopes of spatial locality

Matrix Multiplication Example

- Description:
 - Multiply $N \times N$ 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

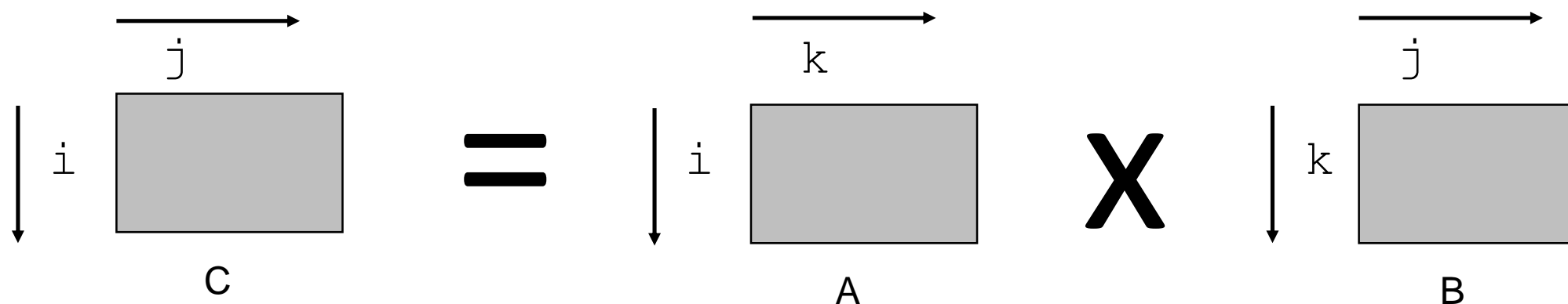
```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable `sum` held in register

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%)

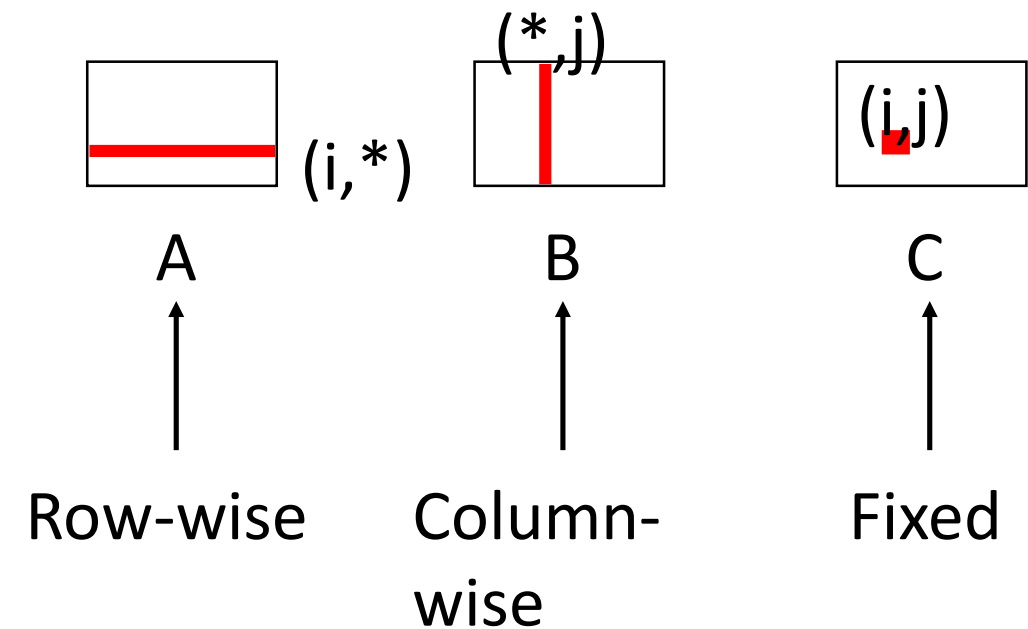
Matrix Multiplication (ijk)

```

/* 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:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (jik)

```

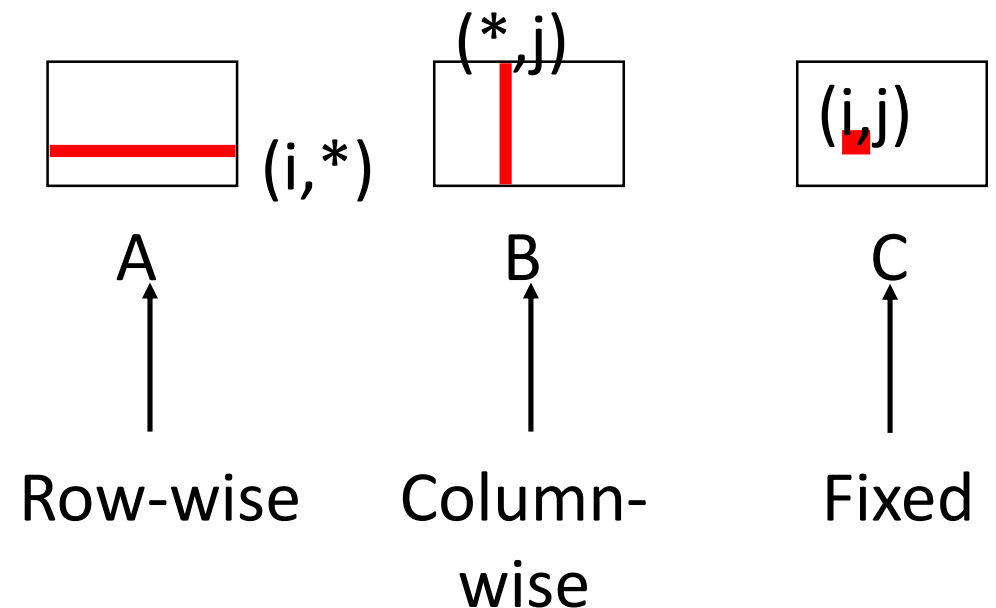
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
                                     matmult/mm.c

```

Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Inner loop:

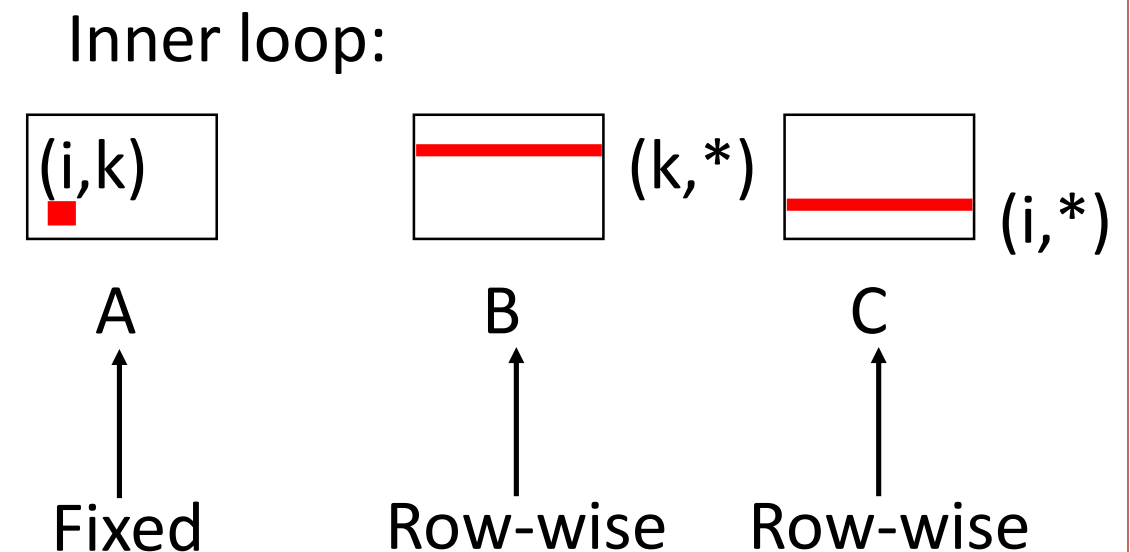


Matrix Multiplication (kij)

```

/* 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

```



Misses per inner loop iteration:

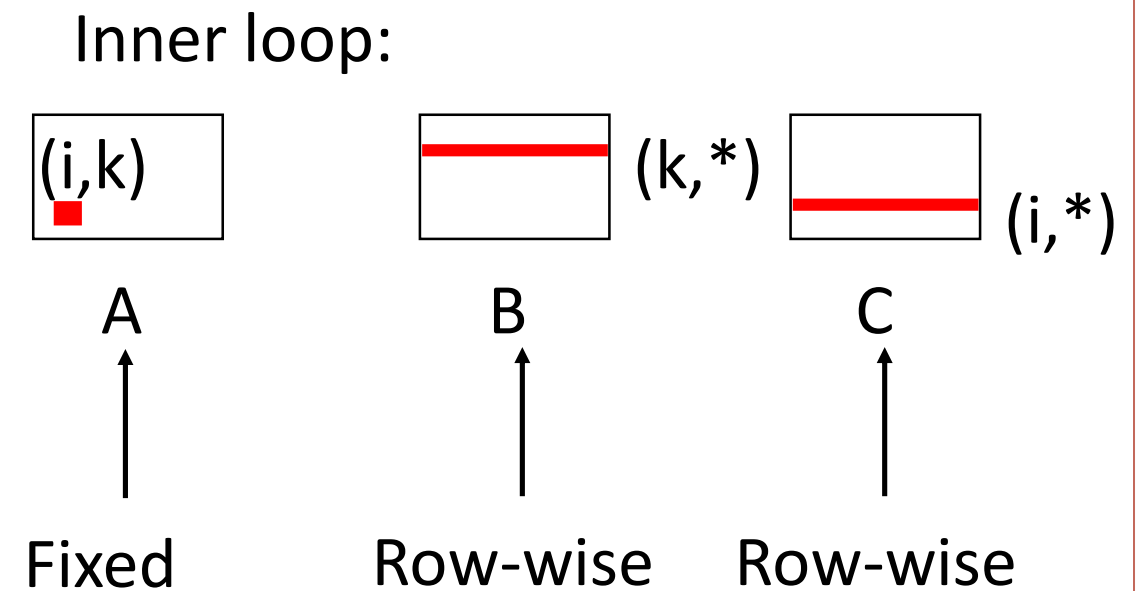
<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

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

```



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

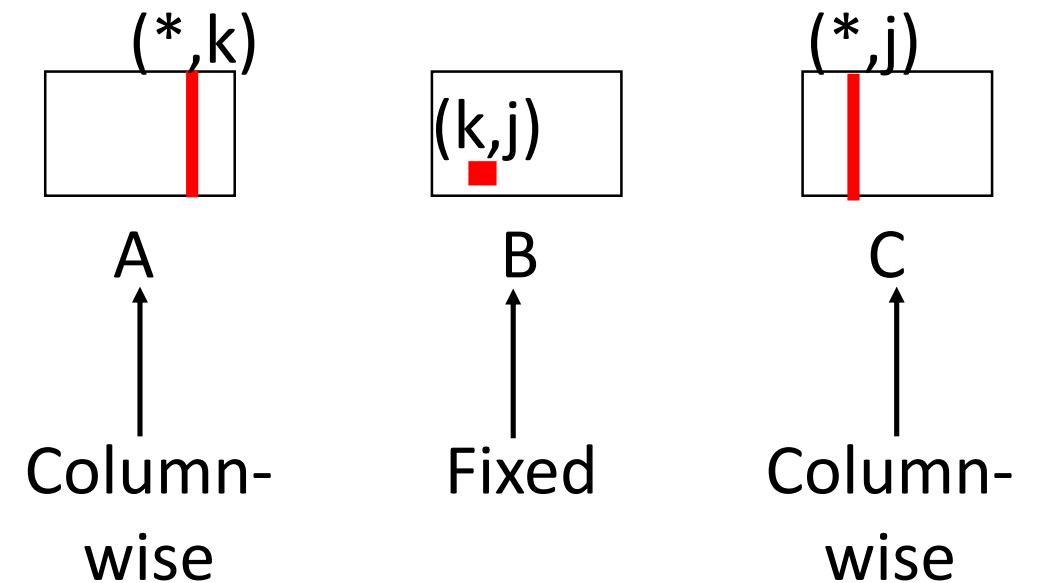
Matrix Multiplication (jki)

```

/* 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:



Misses per inner loop iteration:

A
1.0

B
0.0

C
1.0

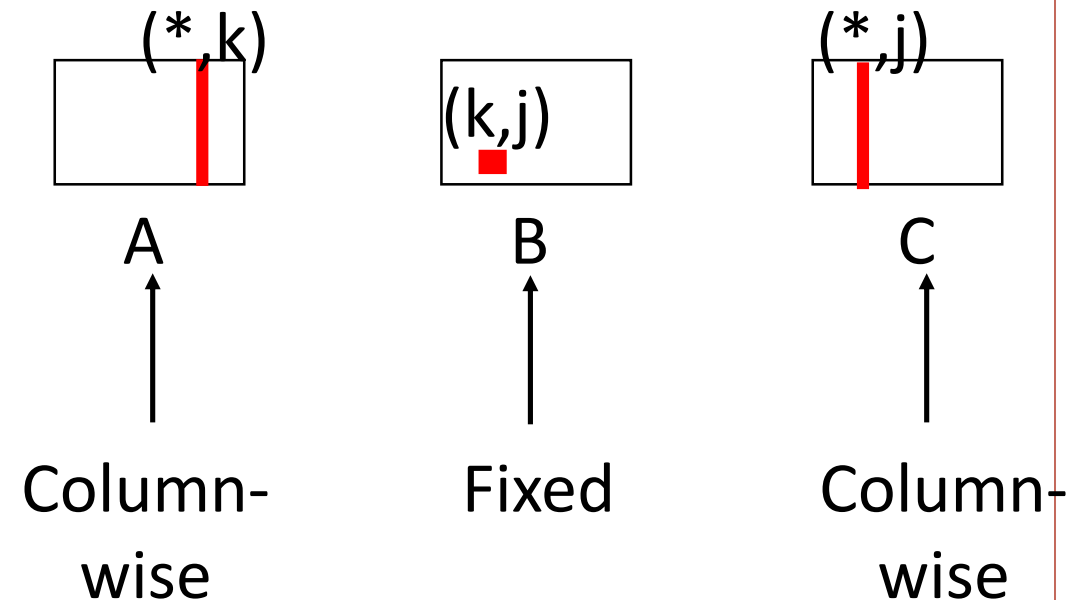
Matrix Multiplication (kji)

```

/* 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:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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;
  }
}
```

```
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];
  }
}
```

```
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;
  }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

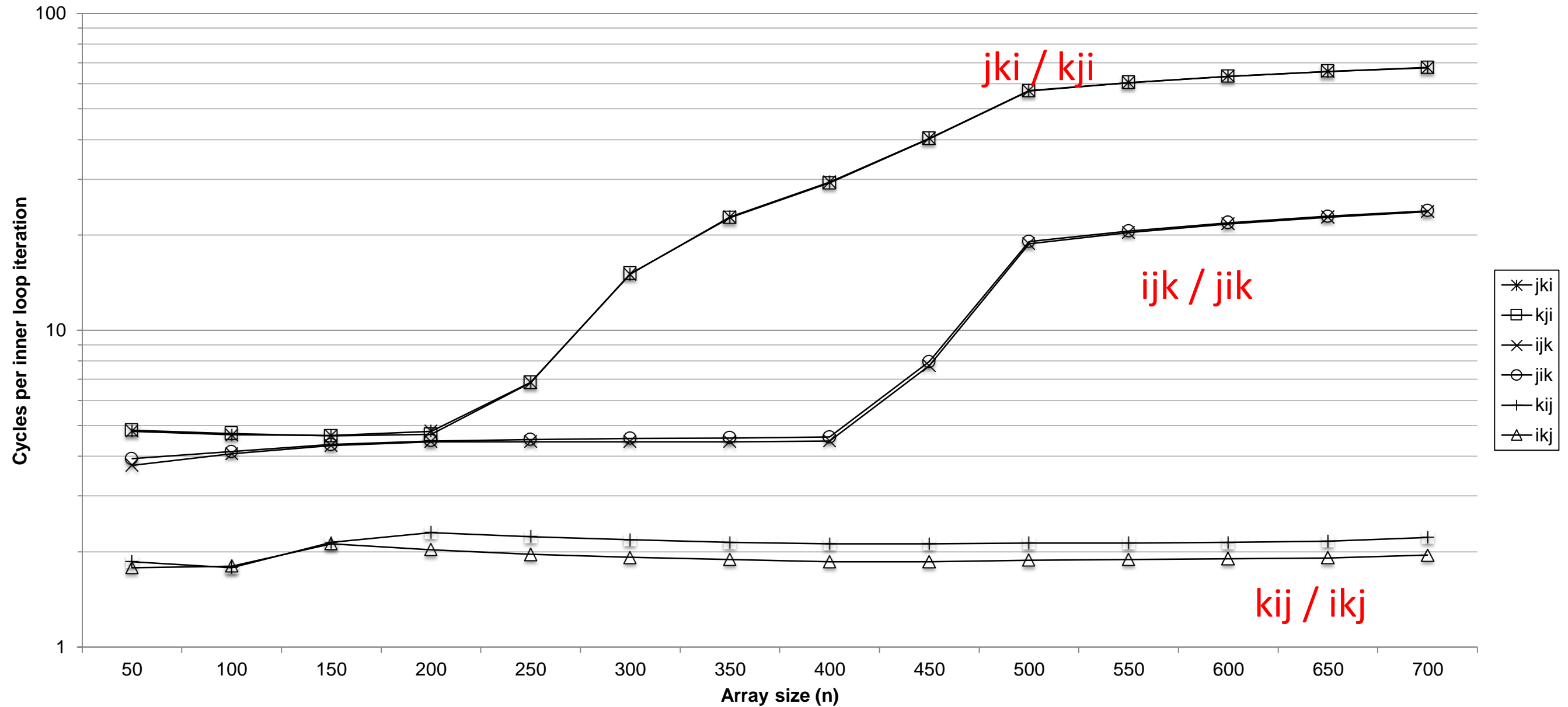
kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Core i7 Matrix Multiply Performance



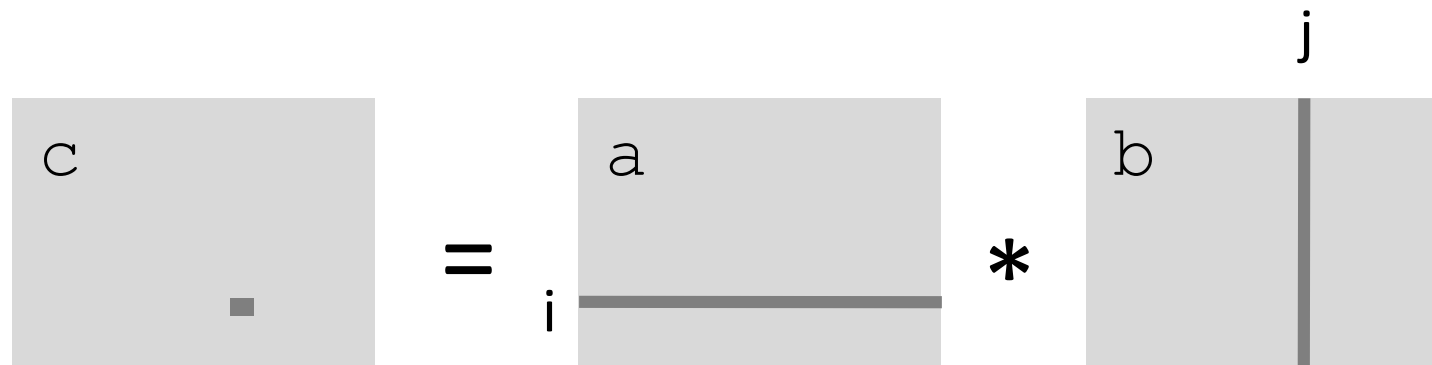
Example: Matrix Multiplication

```

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 $C \ll n$ (much smaller than n)

- First iteration:

- $n/8 + n = 9n/8$ misses



- Afterwards **in cache:**
(schematic)



Cache Miss Analysis

- Assume:
 - Matrix elements are doubles
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (much smaller than n)

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



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

Blocked Matrix Multiplication

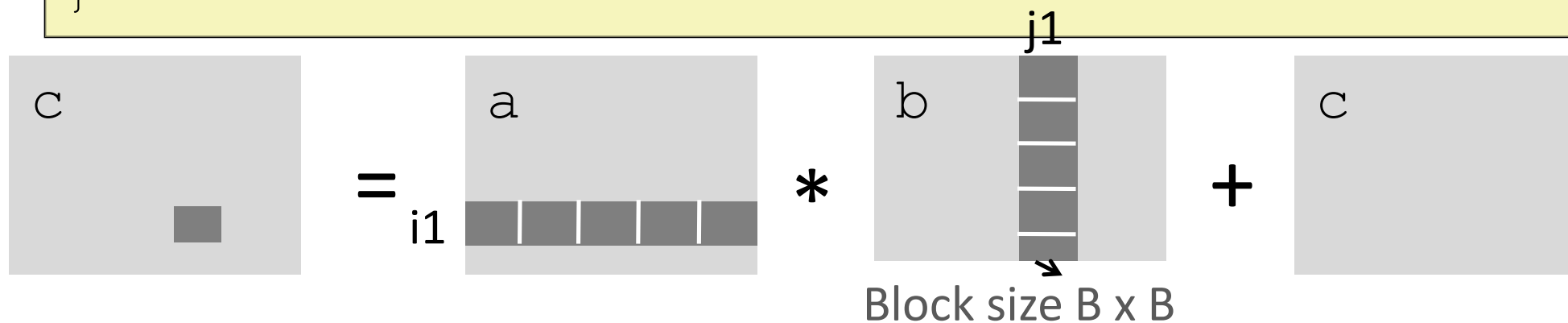
```

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



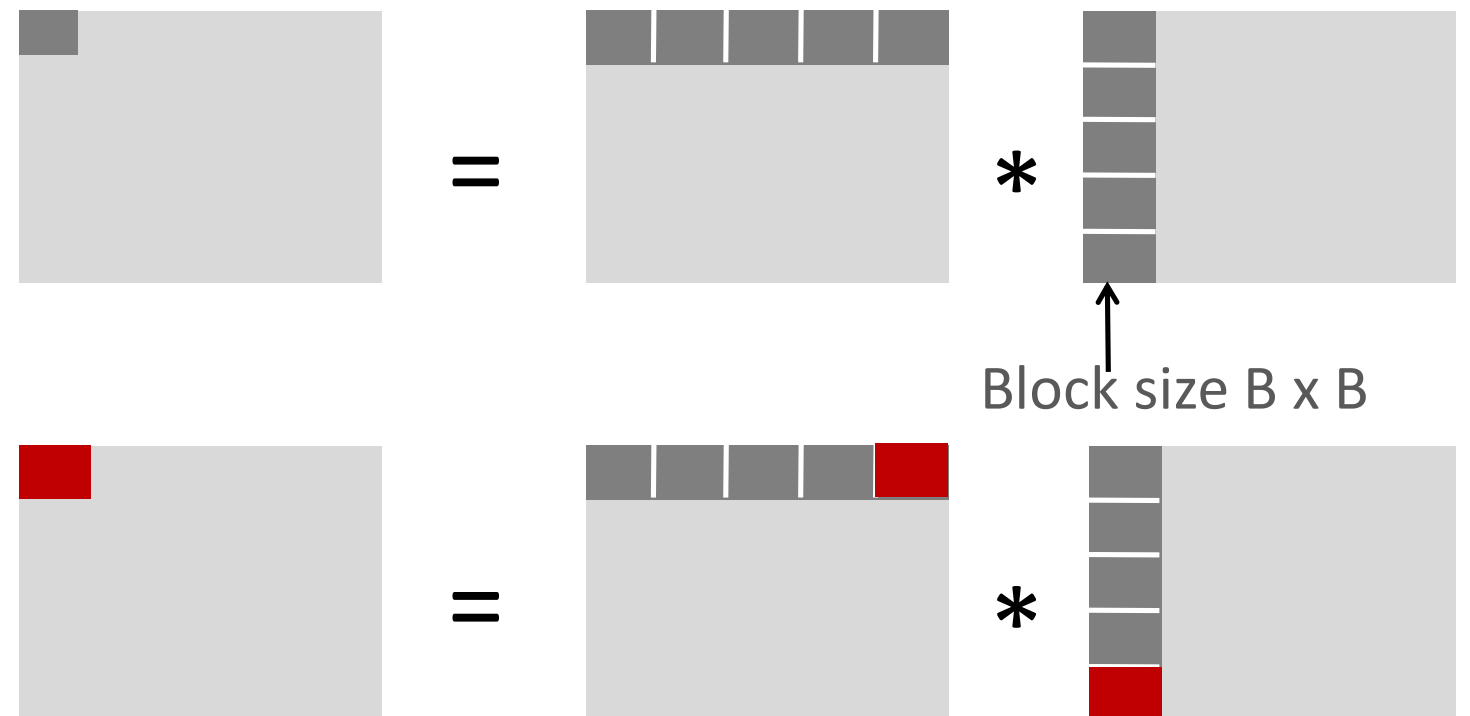
Cache Miss Analysis

- Assume:
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (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)

- Afterwards in cache
(schematic)

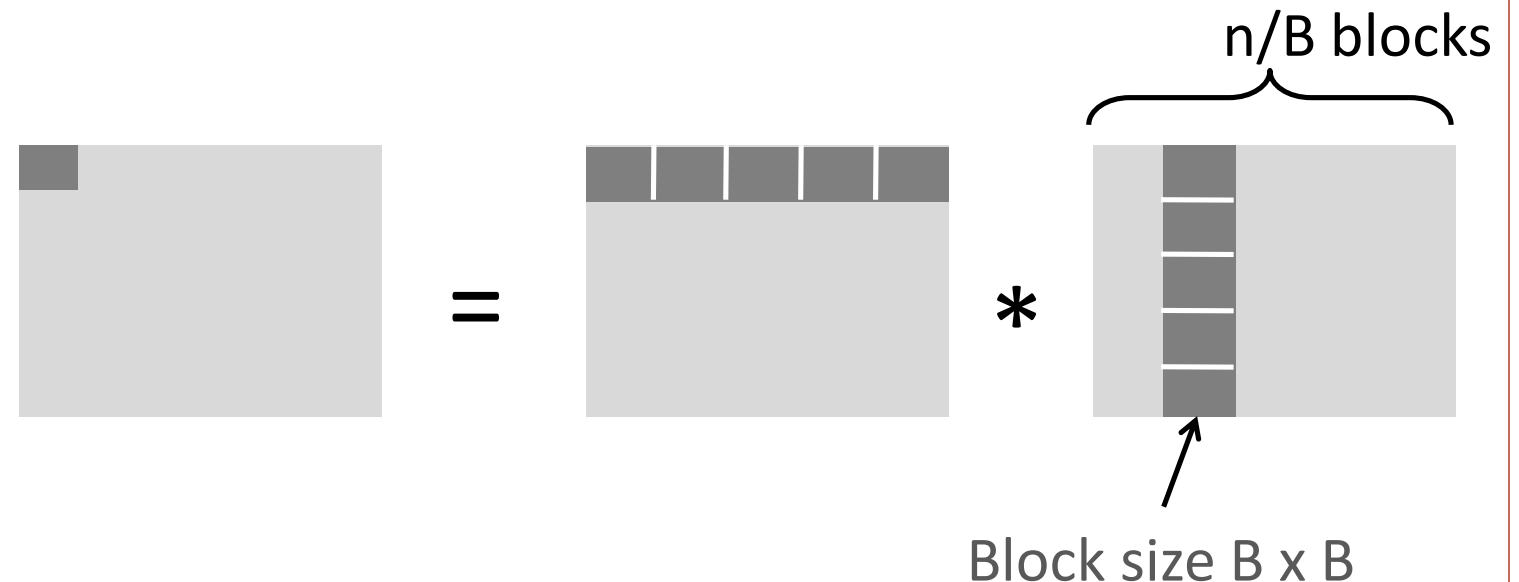


Cache Miss Analysis

- Assume:
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (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:

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

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 sequentially with stride 1.
 - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

18-600 Foundations of Computer Systems

Lecture 11:

"Cache Memories & Non-Volatile Storage"

- A. Cache Organization and Operation
- B. Performance Impact of Caches
 - a. The Memory Mountain
 - b. Rearranging Loops to Improve Spatial Locality
 - c. Using Blocking to Improve Temporal Locality
- C. Non-Volatile Storage Technologies
 - a. Disk Storage Technology
 - b. Flash Memory Technology



What's Inside A Disk Drive?

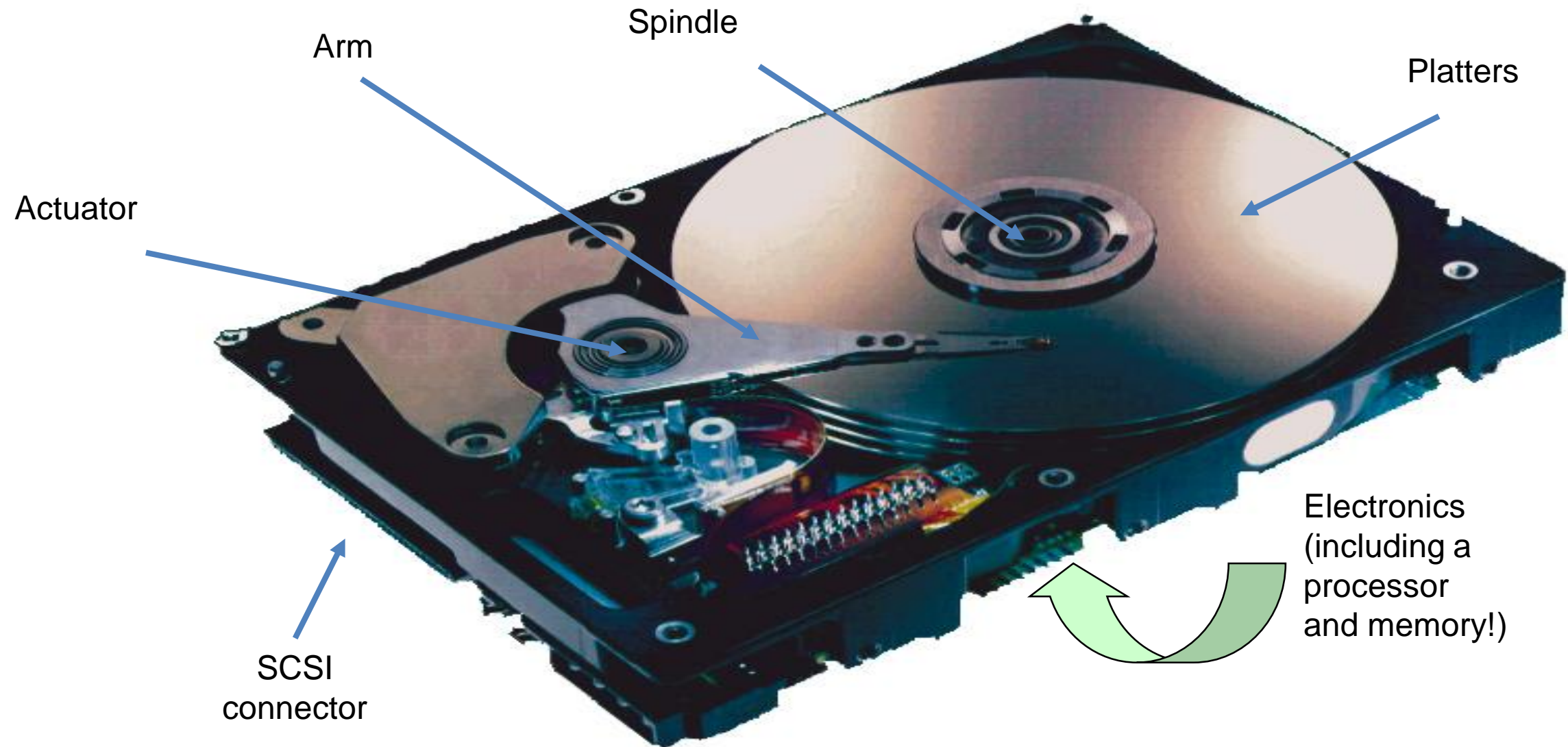
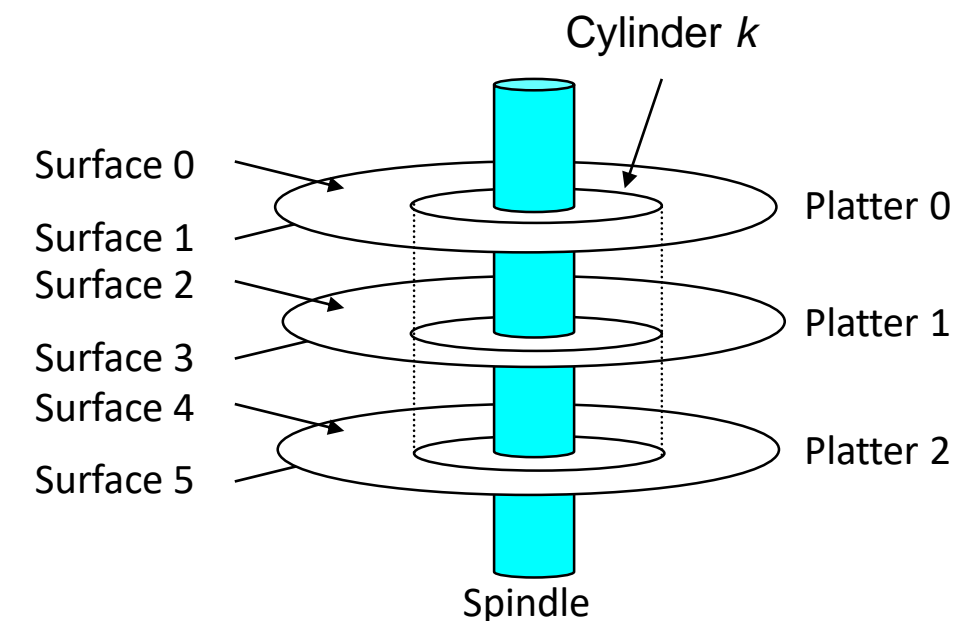
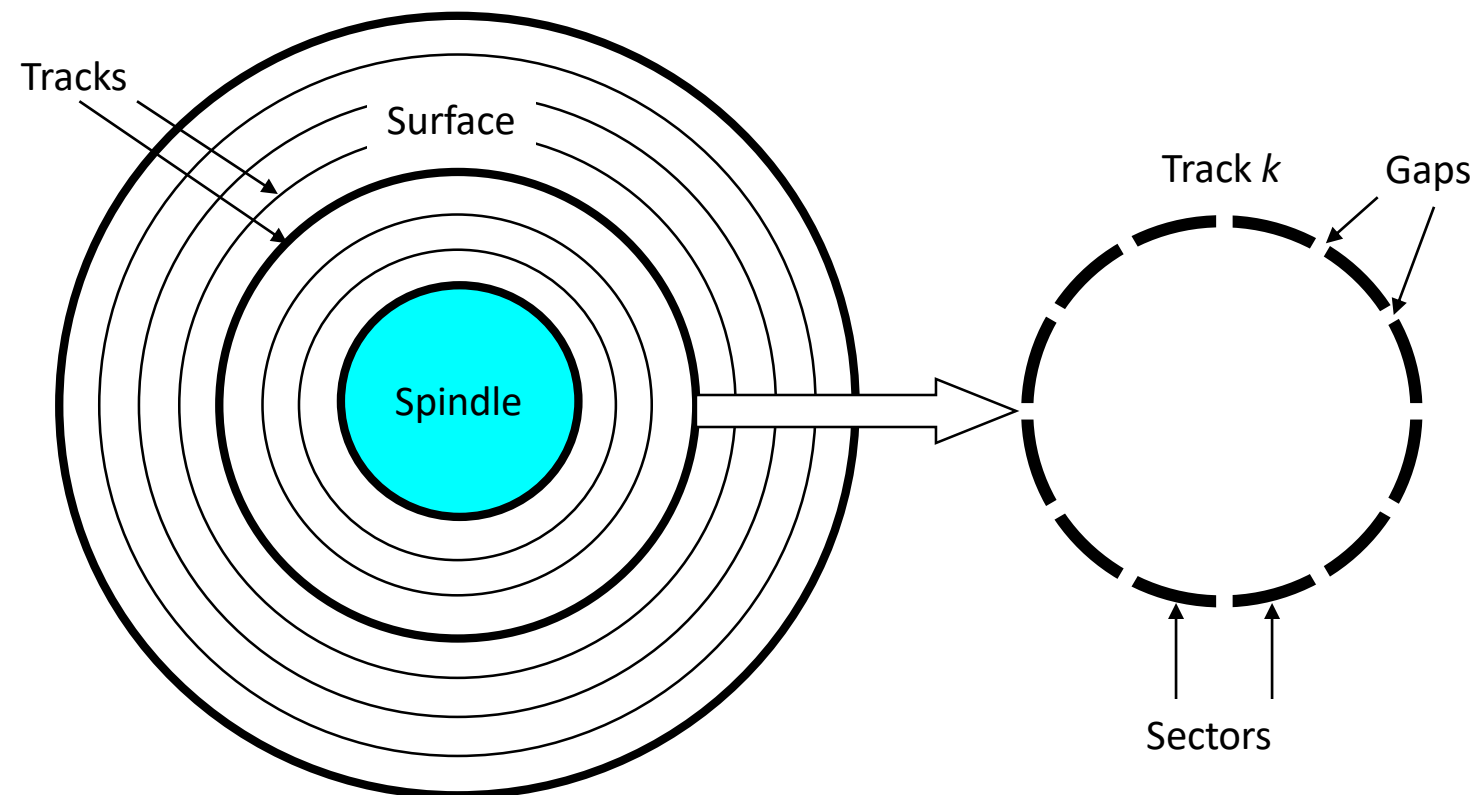


Image courtesy of Seagate Technology

Disk Geometry

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



Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)

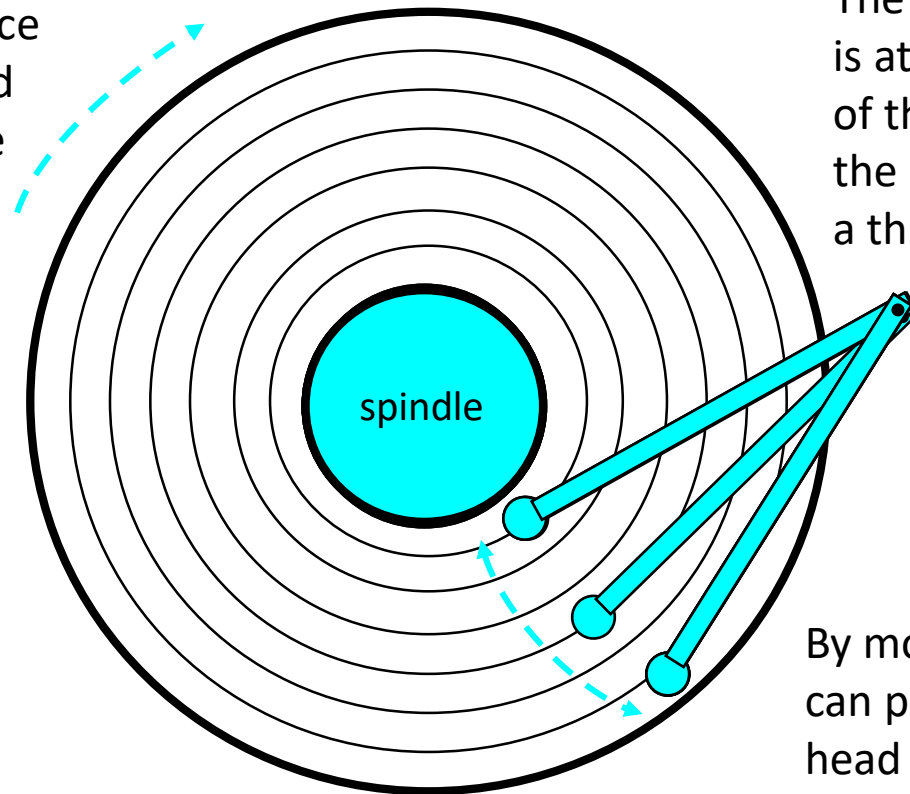
Example:

- 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

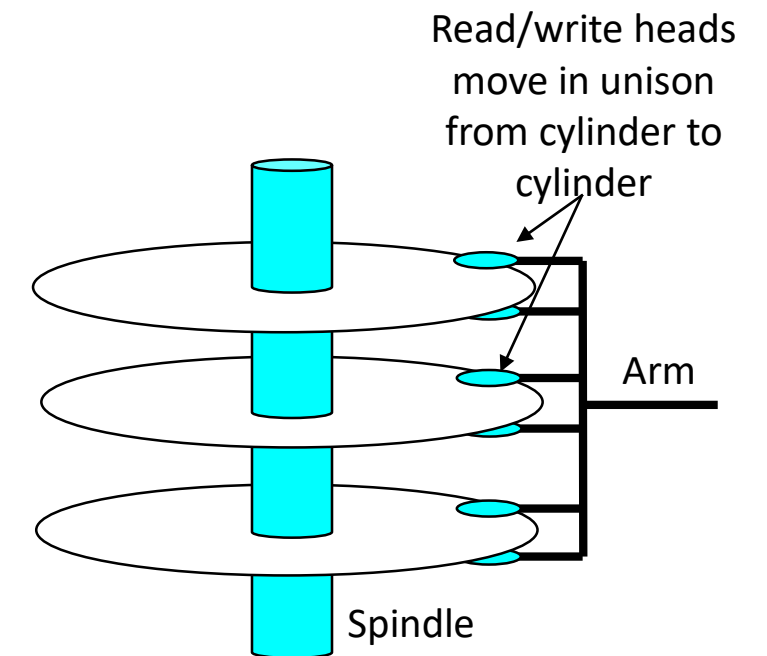
The disk surface spins at a fixed rotational rate



Single-Platter View

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

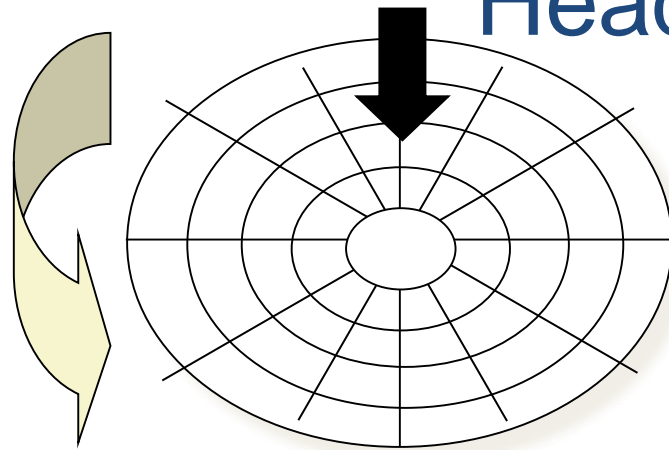
By moving radially, the arm can position the read/write head over any track.



Multi-Platter View

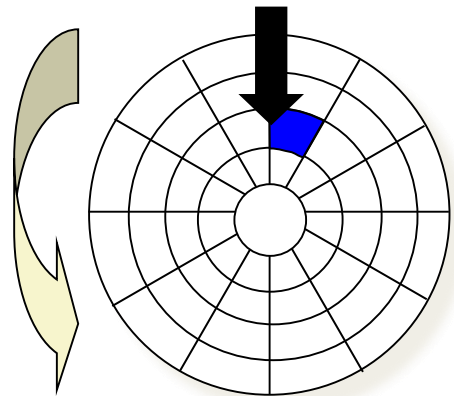
Disk Access

Head in position above a track

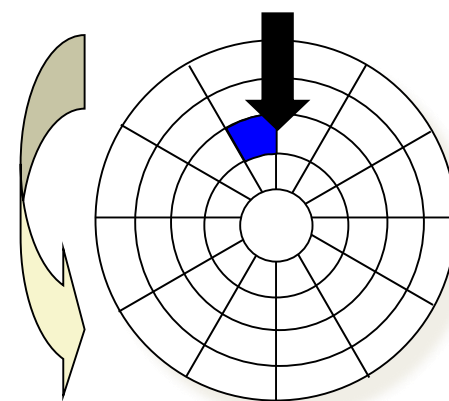


Rotation is counter-clockwise

Disk Access – Read

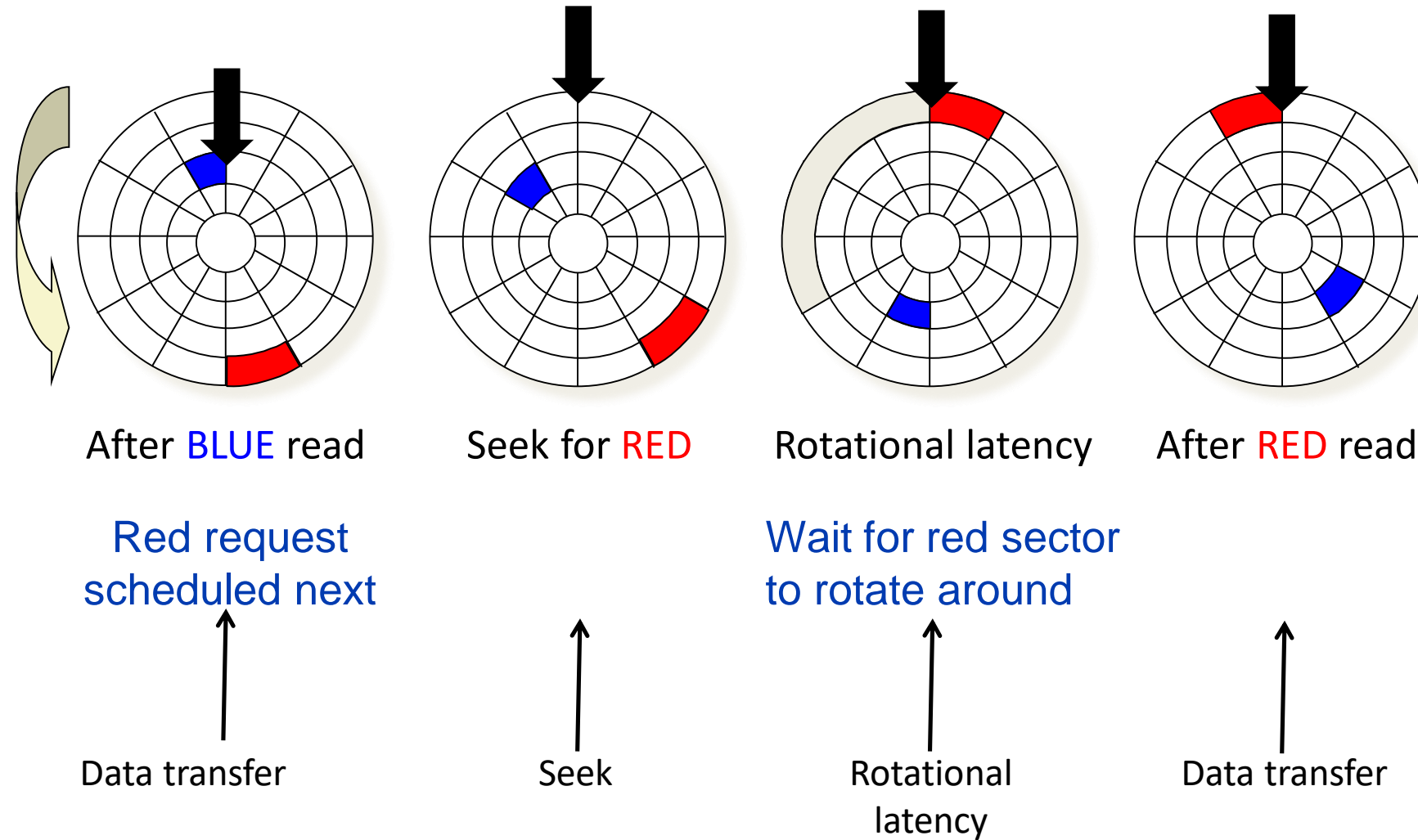


About to read
blue sector



After **BLUE**
read

Disk Access of RED



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

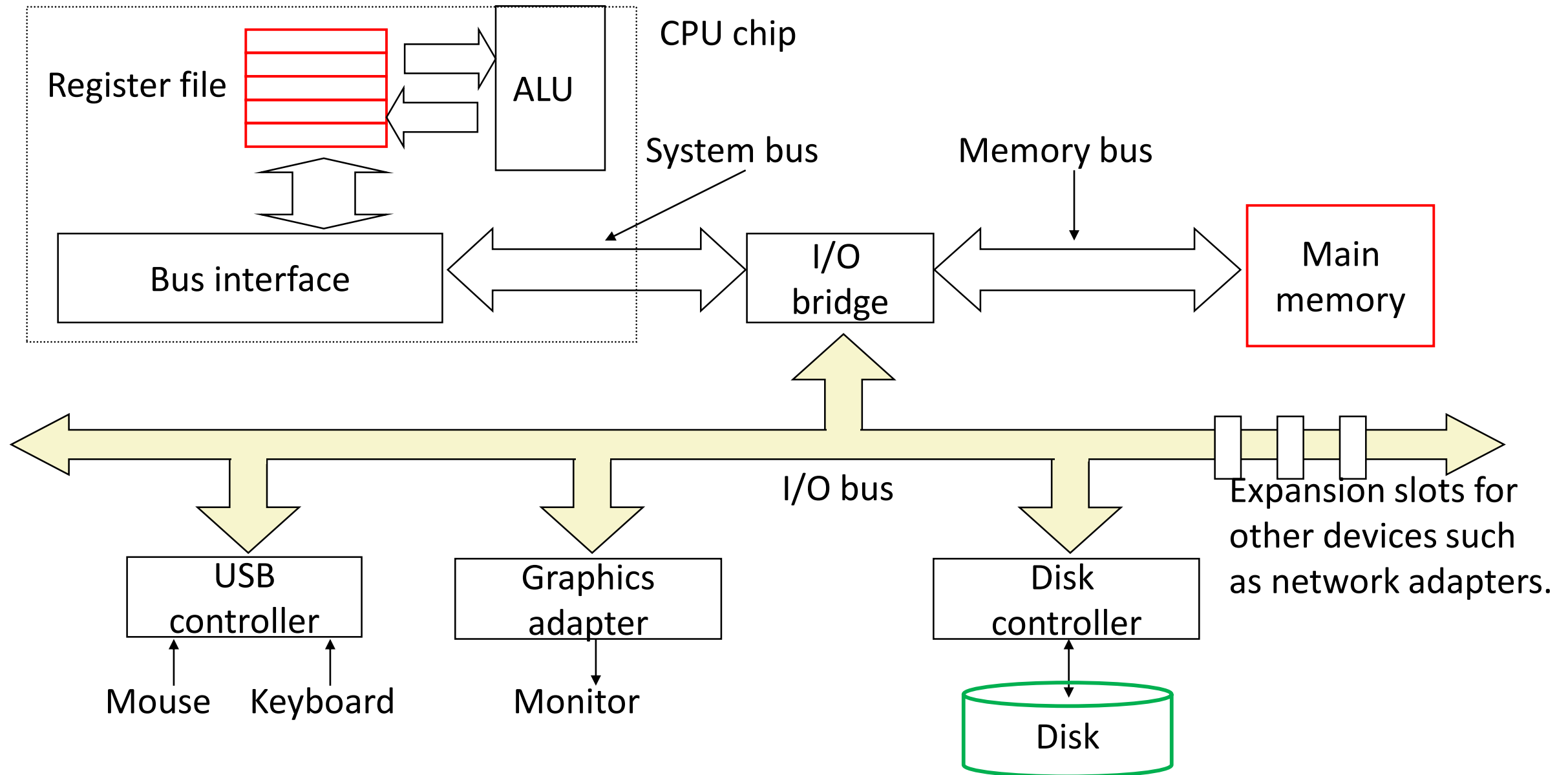
Disk Access Time Example

- 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.

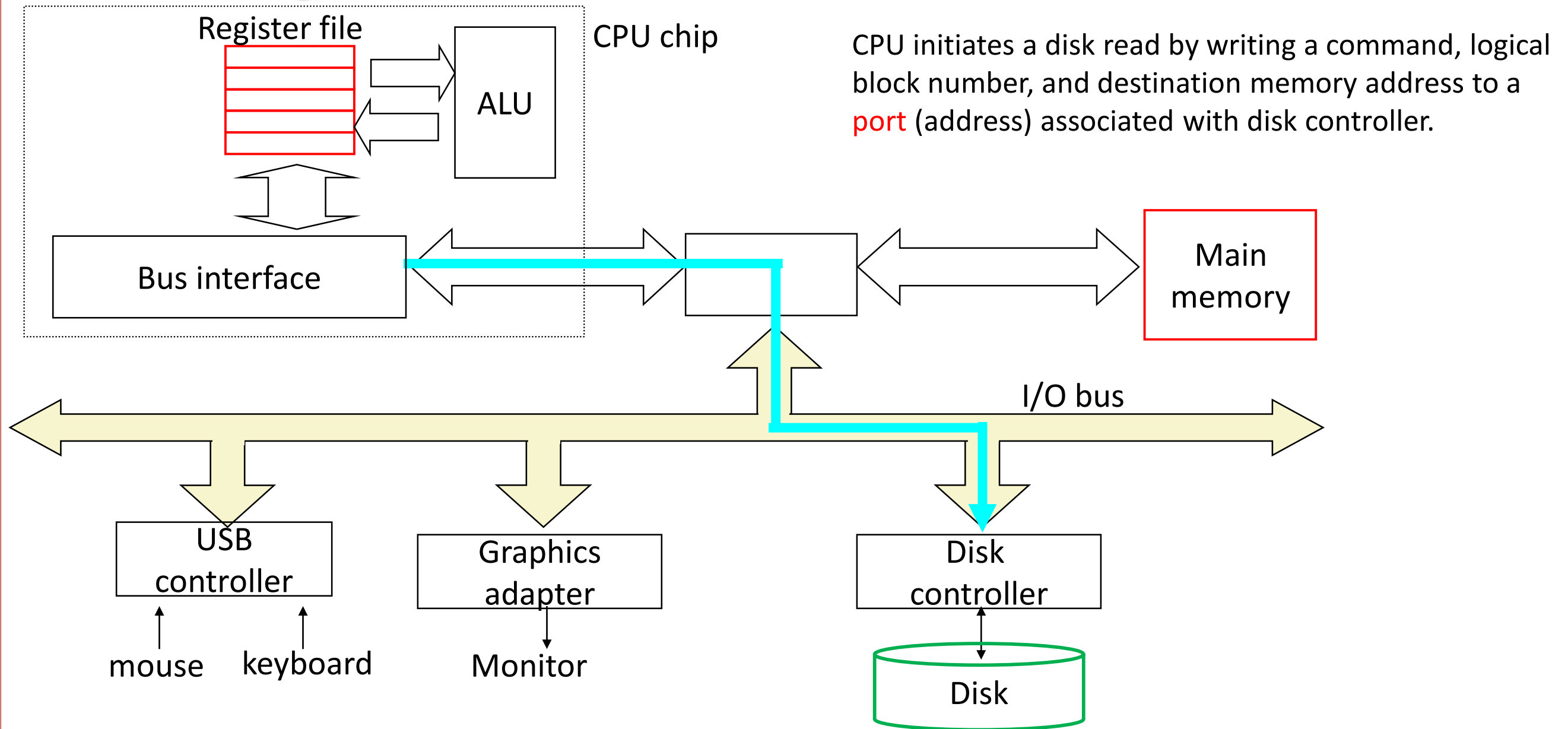
Logical Disk Blocks

- 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”.

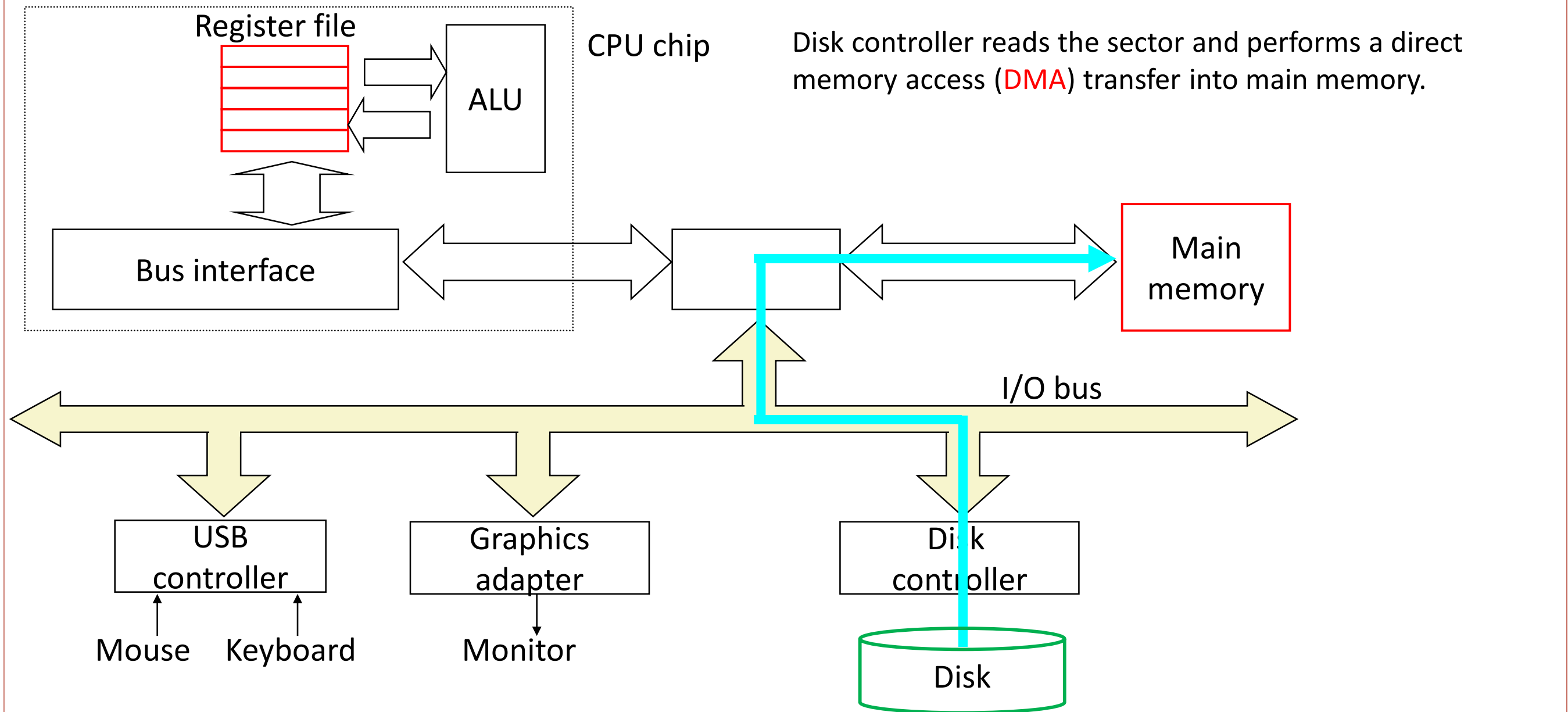
I/O Bus



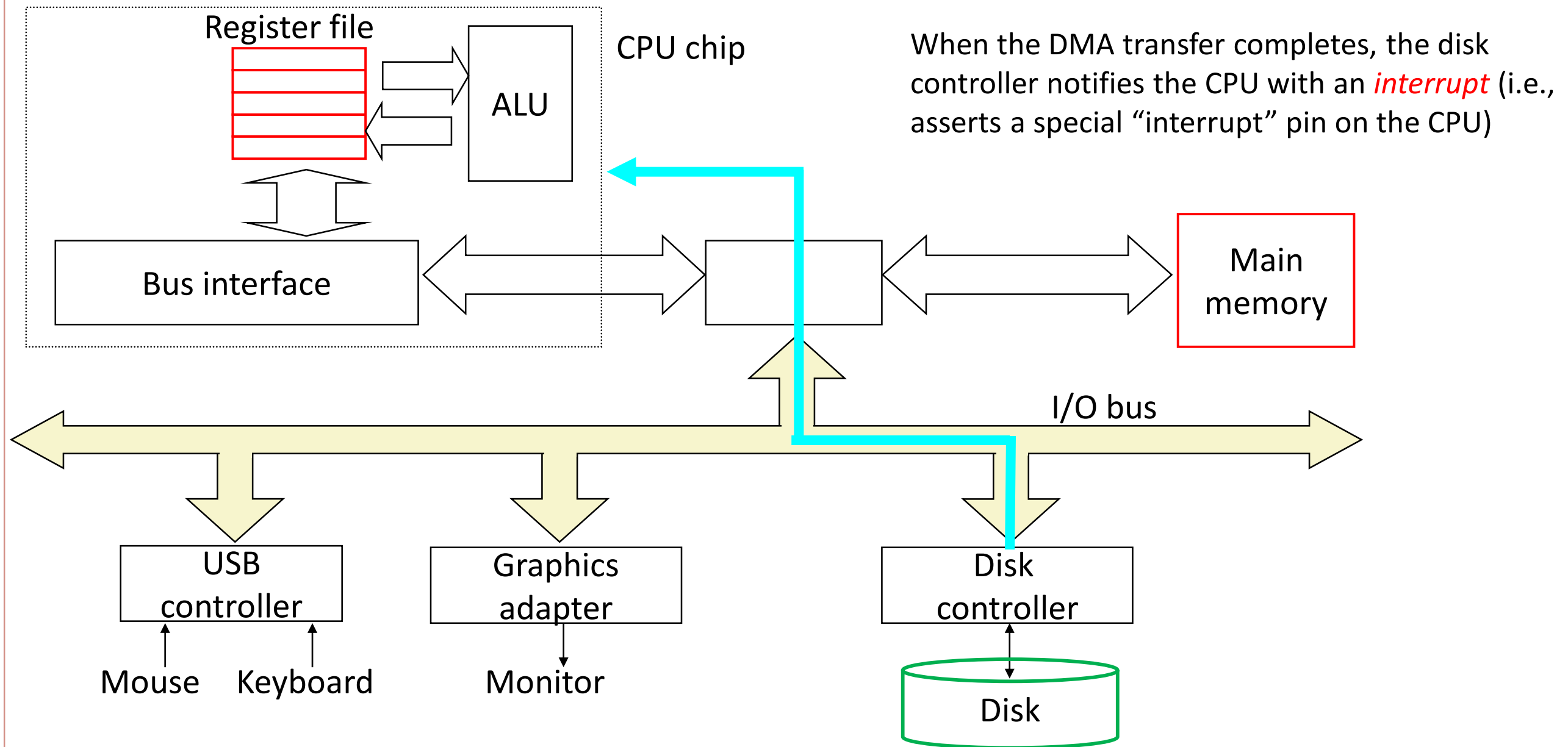
Reading a Disk Sector (1)



Reading a Disk Sector (2)

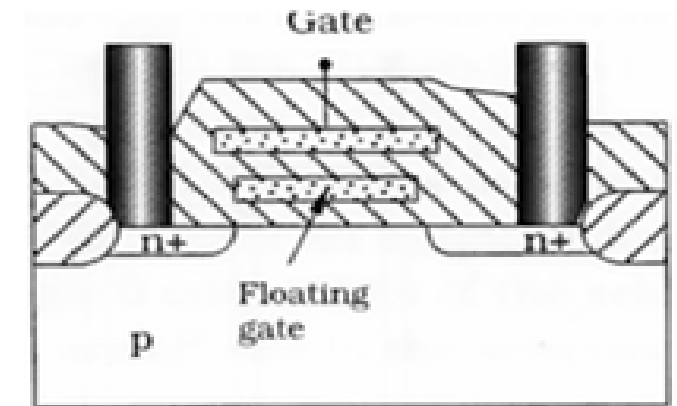


Reading a Disk Sector (3)



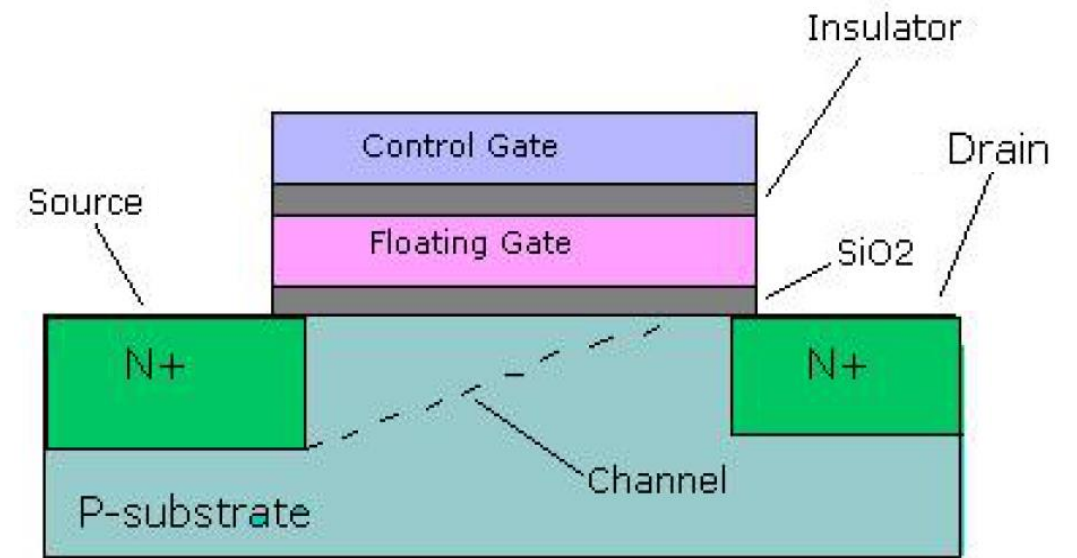
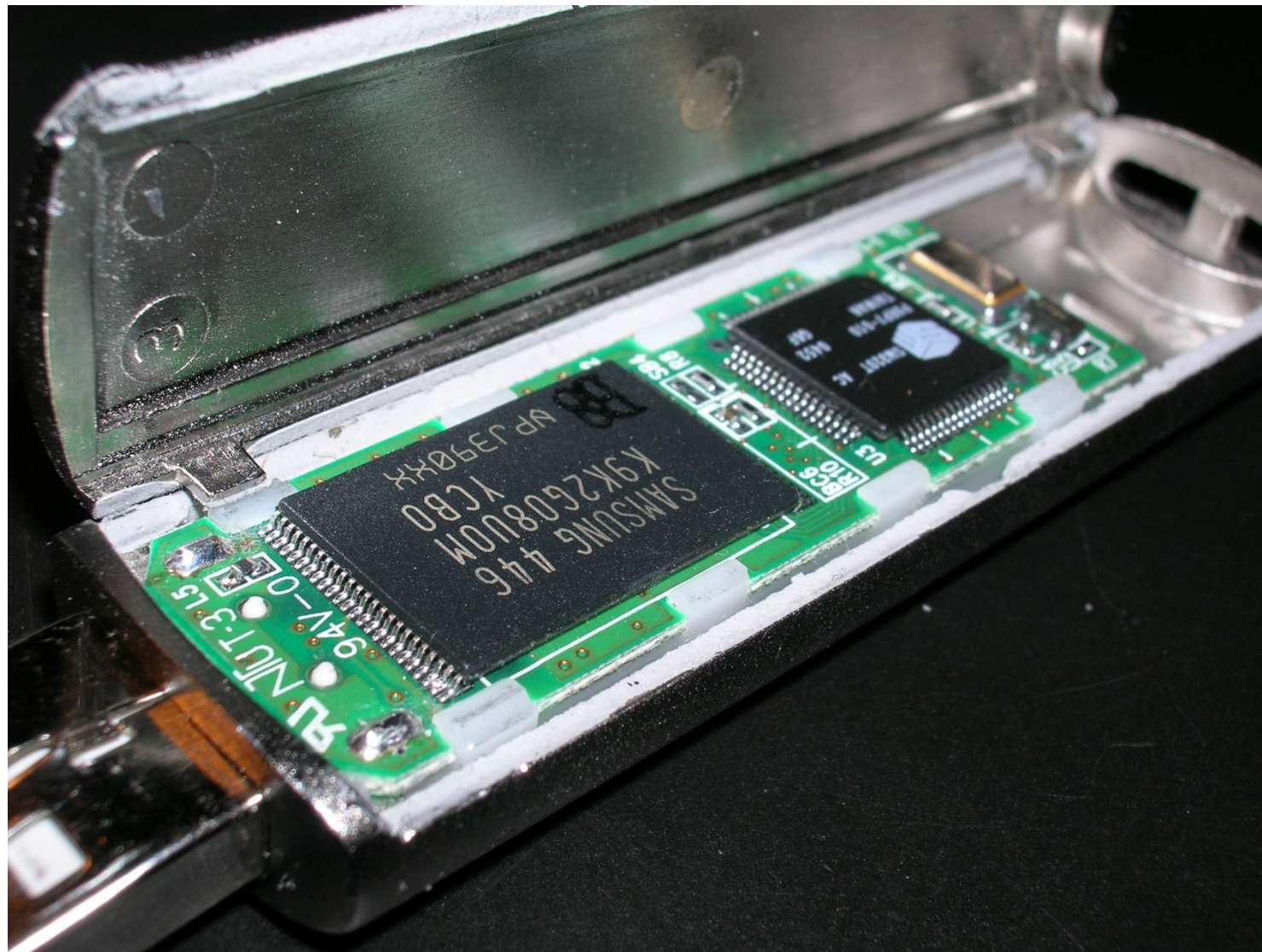
Non-Volatile Memories

- DRAM and SRAM are volatile memories
 - Lose information if powered off.
- Non-volatile 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 (block-level) erase capability
 - Wears out after about 100,000 erasing cycles
- Uses for Non-volatile 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 in large database systems.

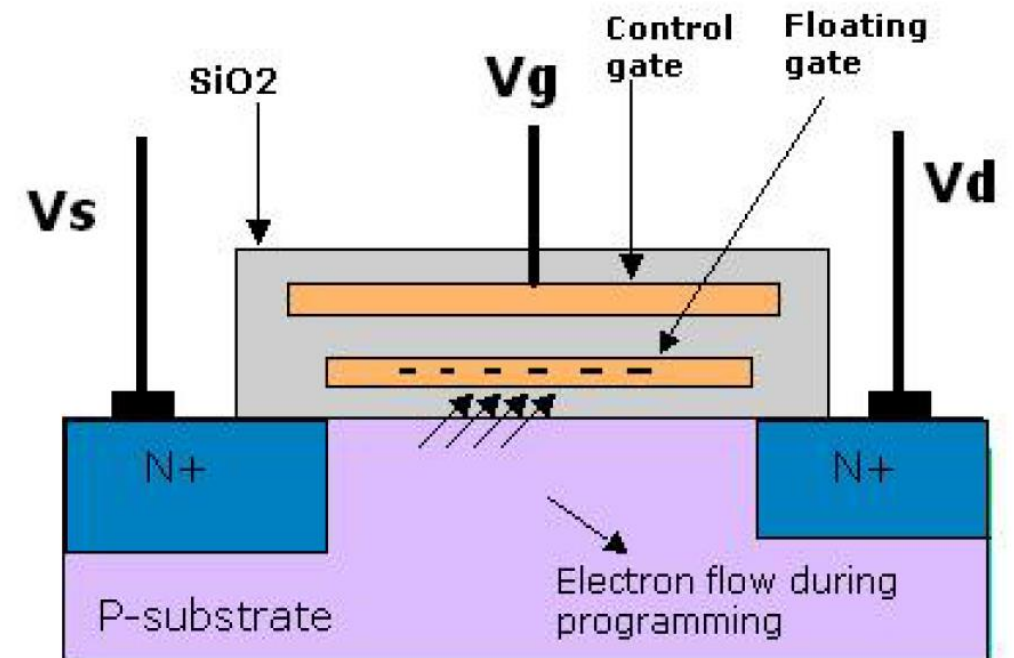


EPROM device structure

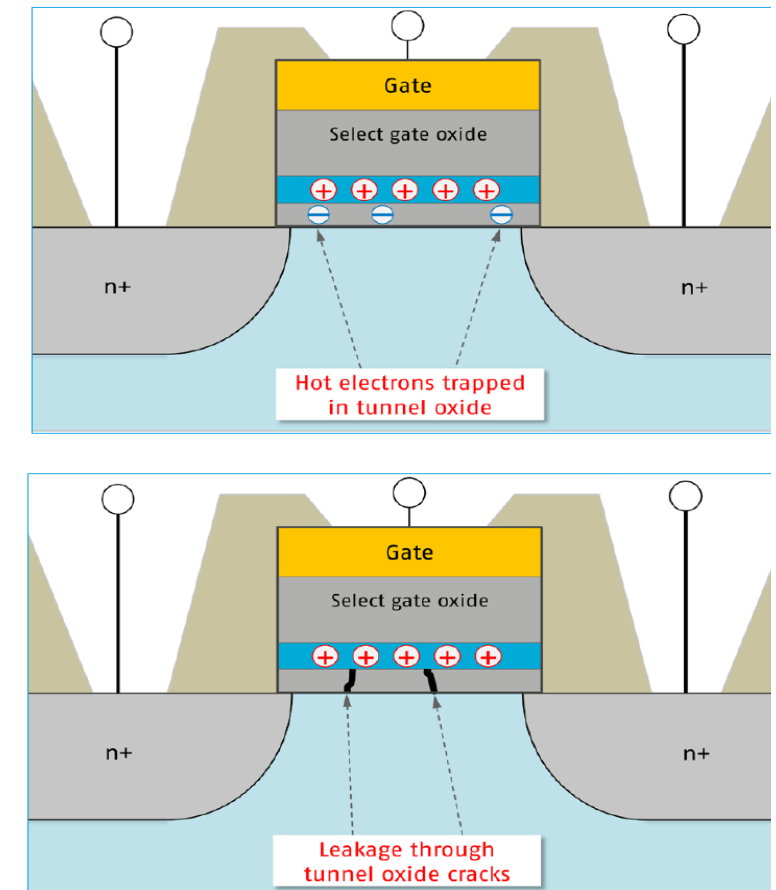
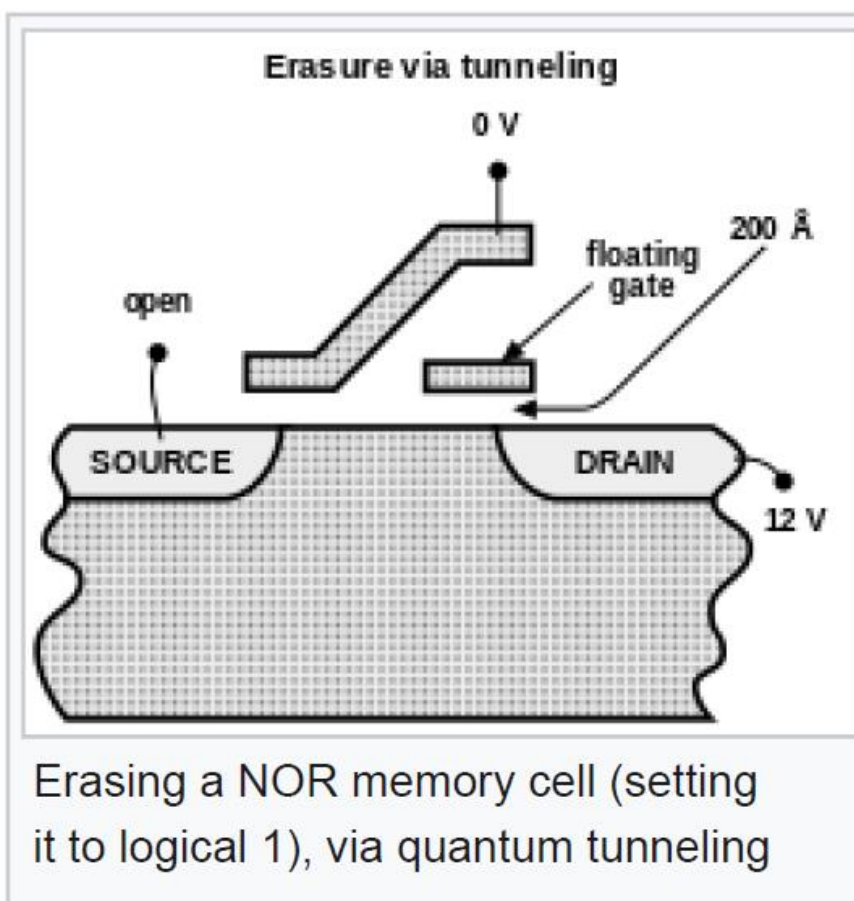
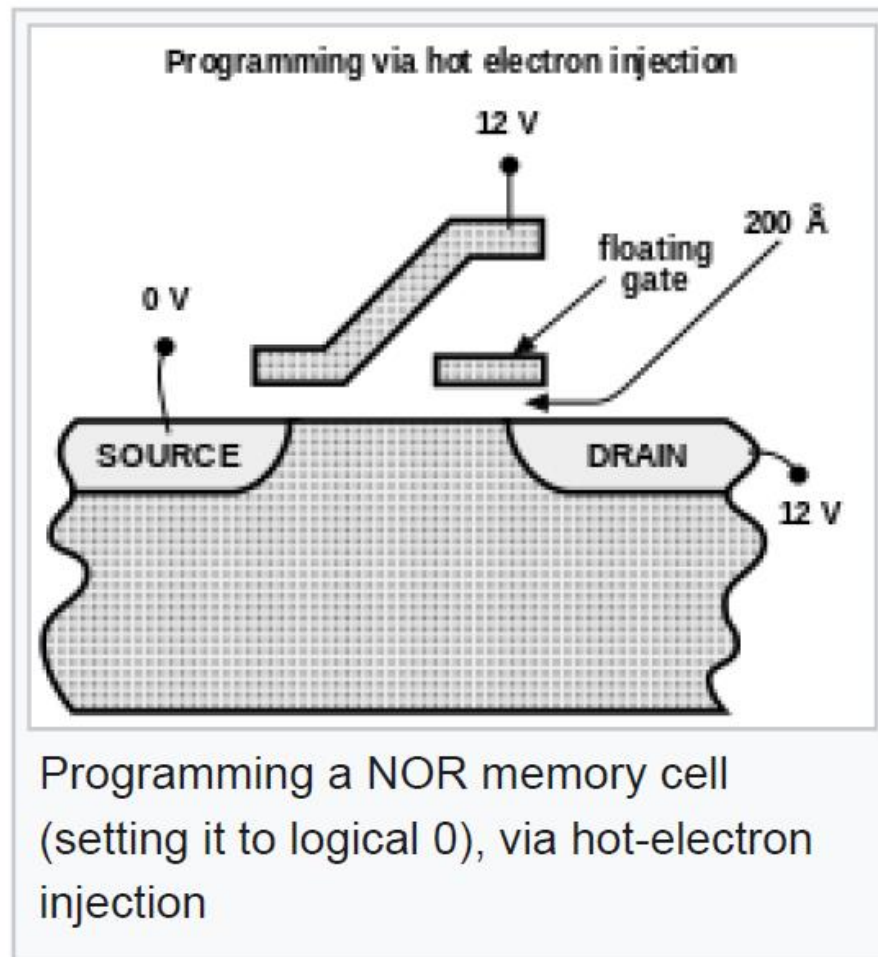
Flash Memory Technology



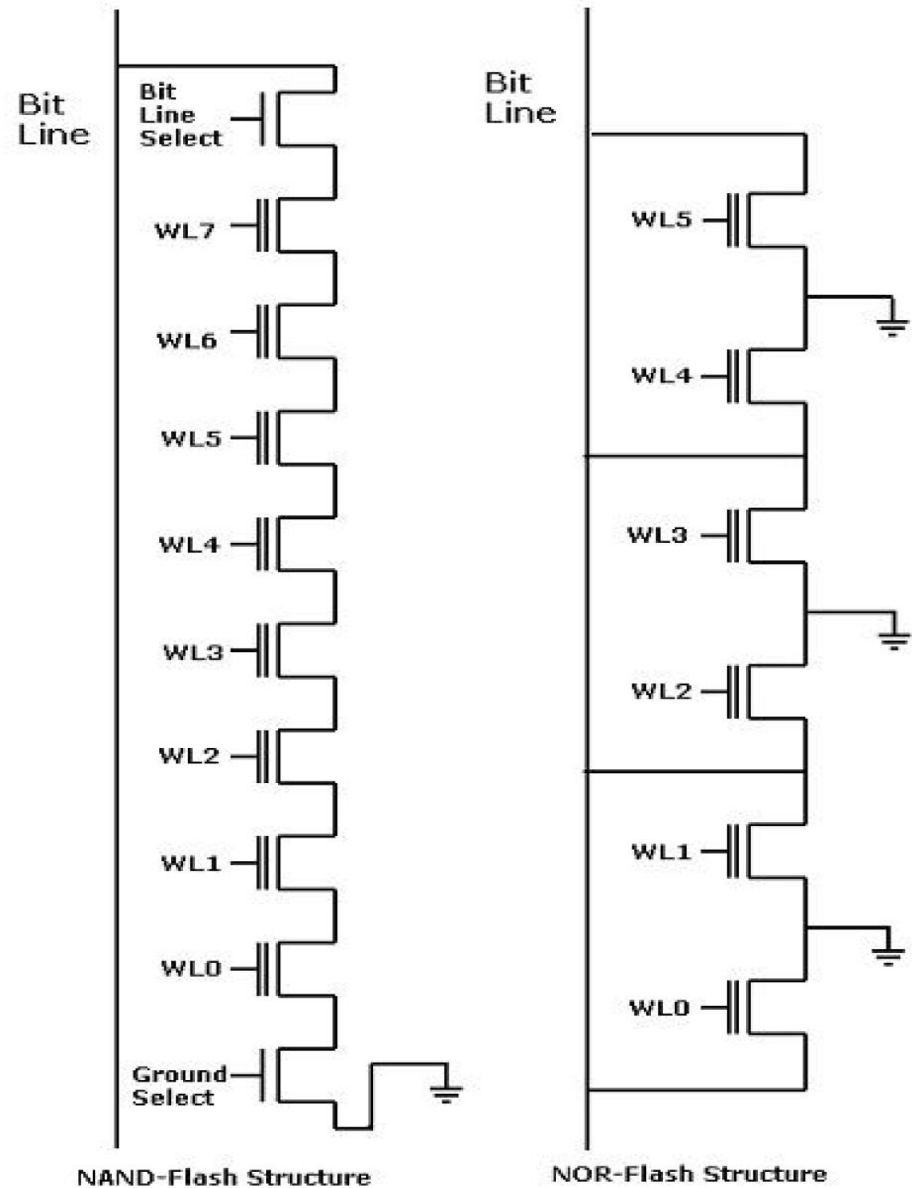
A flash memory cell (FGMOS)



Flash Memory Cell Operation



NAND vs. NOR Flash Memories



	NAND	NOR
Cell Array	<p>Diagram of a NAND cell array showing a vertical Bit line and horizontal Word lines. A Unit Cell is highlighted with a dashed box, showing a series of transistors connected to the Bit line and a Source line. Labels include Contact, Word line, Unit Cell, Bit line, and Source line.</p>	<p>Diagram of a NOR cell array showing a vertical Bit line and horizontal Word lines. A Unit Cell is highlighted with a dashed box, showing a series of transistors connected to the Bit line and a Source line. Labels include Bit line, Contact, Word line, Unit Cell, Bit line, and Source line.</p>
Layout	<p>Layout diagram for NAND showing a square unit cell with side length $2F$.</p>	<p>Layout diagram for NOR showing a rectangular unit cell with width $2F$ and height $5F$.</p>
Cross Section	<p>Cross-section diagram for NAND showing a vertical stack of transistors.</p>	<p>Cross-section diagram for NOR showing a horizontal stack of transistors.</p>
Cell Size	$4F^2$	$10F^2$

NAND vs. NOR Flash Memories

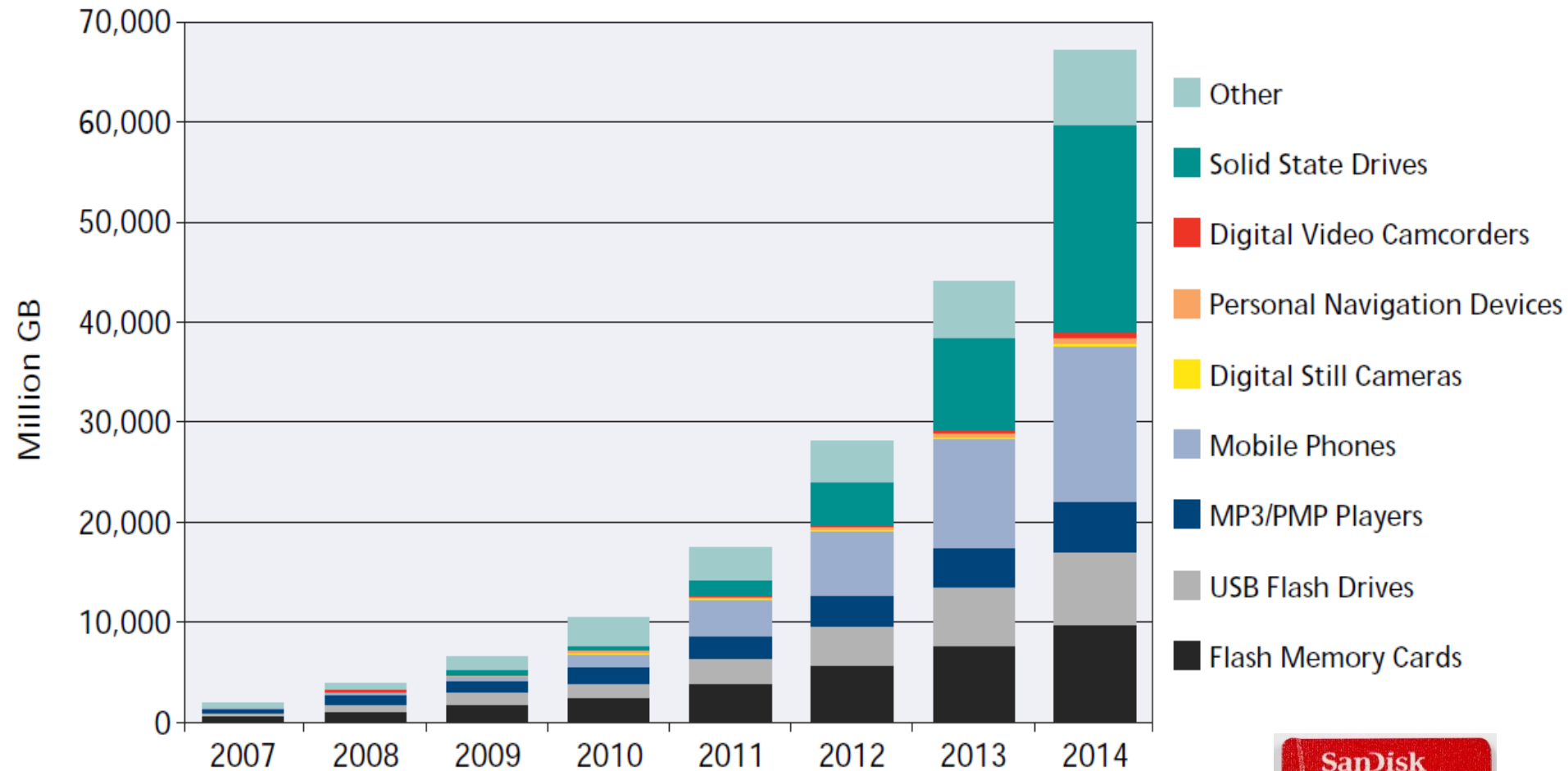
Attribute	NAND	NOR
Main Application	File storage	Code execution
Storage capacity	High	Low
Cost per bit	Better	
Active Power	Better	
Standby Power		Better
Write Speed	Good	
Read Speed		Good

Comparison characteristics	MLC : SLC	NAND : NOR
Persistence ratio	1 : 10	1 : 10
Sequential write ratio	1 : 3	1 : 4
Sequential read ratio	1 : 1	1 : 5
Price ratio	1 : 1.3	1 : 0.7

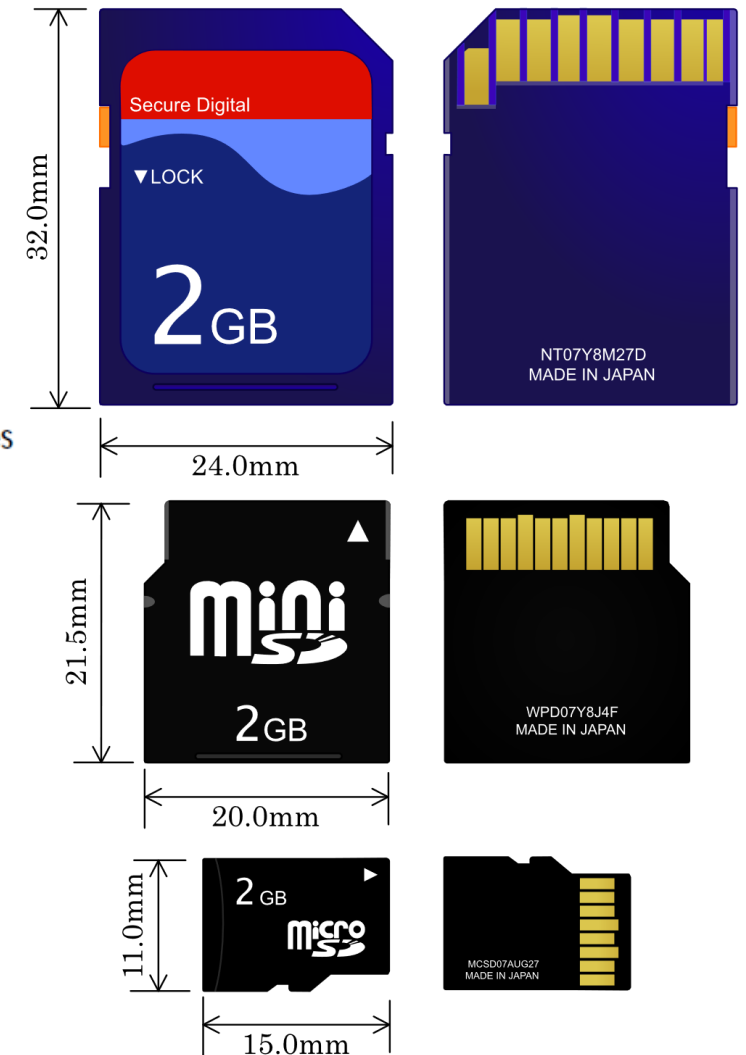
Characteristic	NAND Flash: MT29F2G08A	NOR Flash: TE28F128J3
Random access READ	25 μ s (first byte) 0.025 μ s each for remaining 2111 bytes	0.075 μ s
Sustained READ speed (sector basis)	26 MB/s (x8) or 41 MB/s (x16)	31 MB/s (x8) or 62 MB/s (x16)
Random WRITE speed	\approx 220 μ s/2112 bytes	128 μ s/32 bytes
Sustained WRITE speed (sector basis)	7.5 MB/s	0.250 MB/s
Erase block size	128KB	128KB
ERASE time per block (TYP)	500 μ s	1 sec

NAND Flash & Secured Digital (SD) Cards

Major Markets Driving NAND Flash



Source: Forward Insights, NAND Quarterly Insights Q3/09, www.forward-insights.com/ Report No. FI-NFL-NQI-Q309 September 2009, accessed 4/14/2010; used with permission.



Solid State Drive (SSD) vs. Hard Disk Drive (HDD)



SSD Top Side



SSD Bottom Side



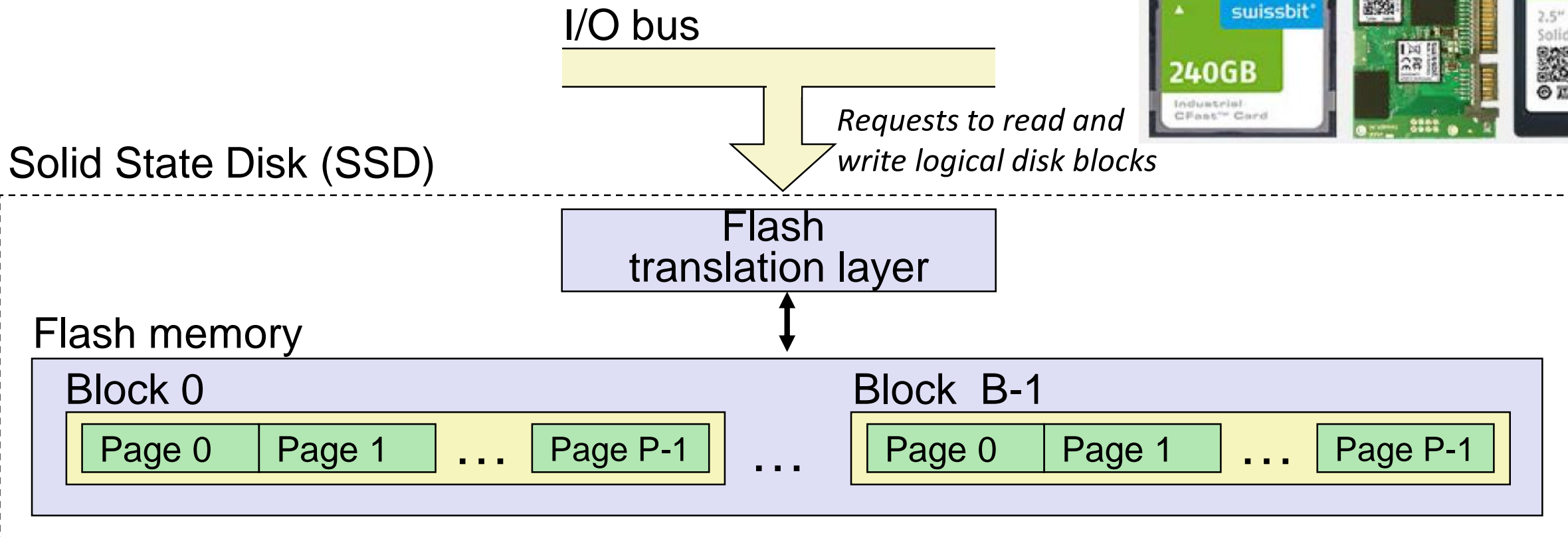
HDD Top Side



HDD Bottom Side

Attribute	SSD (Solid State Drive)	HDD (Hard Disk Drive)
Power Draw / Battery Life	Less power draw, averages 2 – 3 watts, resulting in 30+ minute battery boost	More power draw, averages 6 – 7 watts and therefore uses more battery
Cost	Expensive, roughly \$0.20 per gigabyte (based on buying a 1TB drive)	Only around \$0.03 per gigabyte, very cheap (buying a 4TB model)
Capacity	Typically not larger than 1TB for notebook size drives; 4TB max for desktops	Typically around 500GB and 2TB maximum for notebook size drives; 10TB max for desktops
Operating System Boot Time	Around 10-13 seconds average bootup time	Around 30-40 seconds average bootup time
Noise	There are no moving parts and as such no sound	Audible clicks and spinning can be heard
Vibration	No vibration as there are no moving parts	The spinning of the platters can sometimes result in vibration
Heat Produced	Lower power draw and no moving parts so little heat is produced	HDD doesn't produce much heat, but it will have a measurable amount more heat than an SSD due to moving parts and higher

Solid State Disks (SSDs)

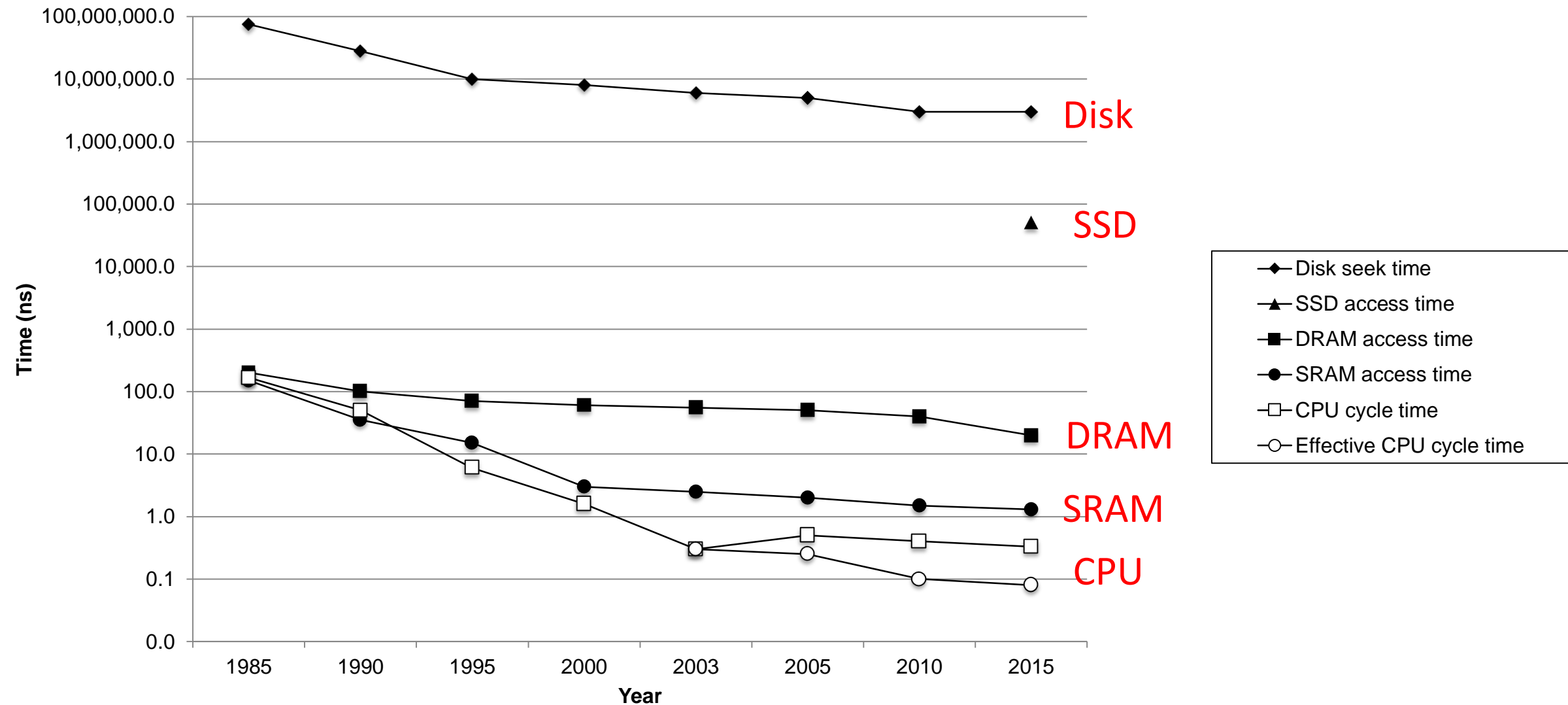


- Pages: 512B to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

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. Intel SSD 730 guarantees 128 petabyte (128×10^{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 (as disk cache)

The CPU-Memory-Storage Gaps

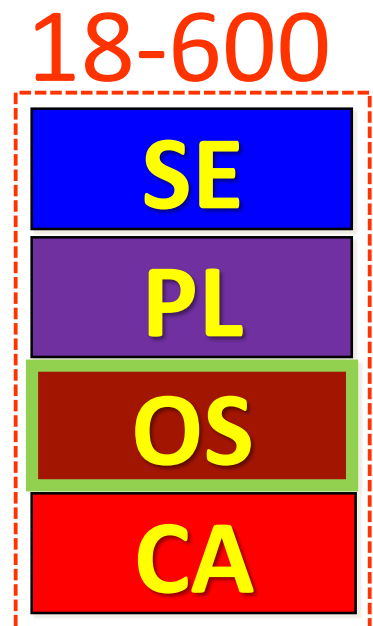


18-600 Foundations of Computer Systems

Lecture 12: "ECF I: Exceptions and Processes"

John P. Shen & Gregory Kesden
October 9, 2017

Next Time ...



➤ Required Reading Assignment:

- Chapter 5 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.



Electrical & Computer
ENGINEERING