Computer Systems: A Programmer's Perspective
计算机系统

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      - procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly

Memory Hierarchy

- **The memory abstraction**
- **RAM : main memory building block**
- **Locality of reference**
- **The memory hierarchy**
- **Cache**

Writing & Reading Memory

存储器的基本操作： 读和写操作

- **Write**
  - Transfer data from CPU to memory

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

    movq 8(%rsp), %rax
  - "Load" operation

Typical PC organization

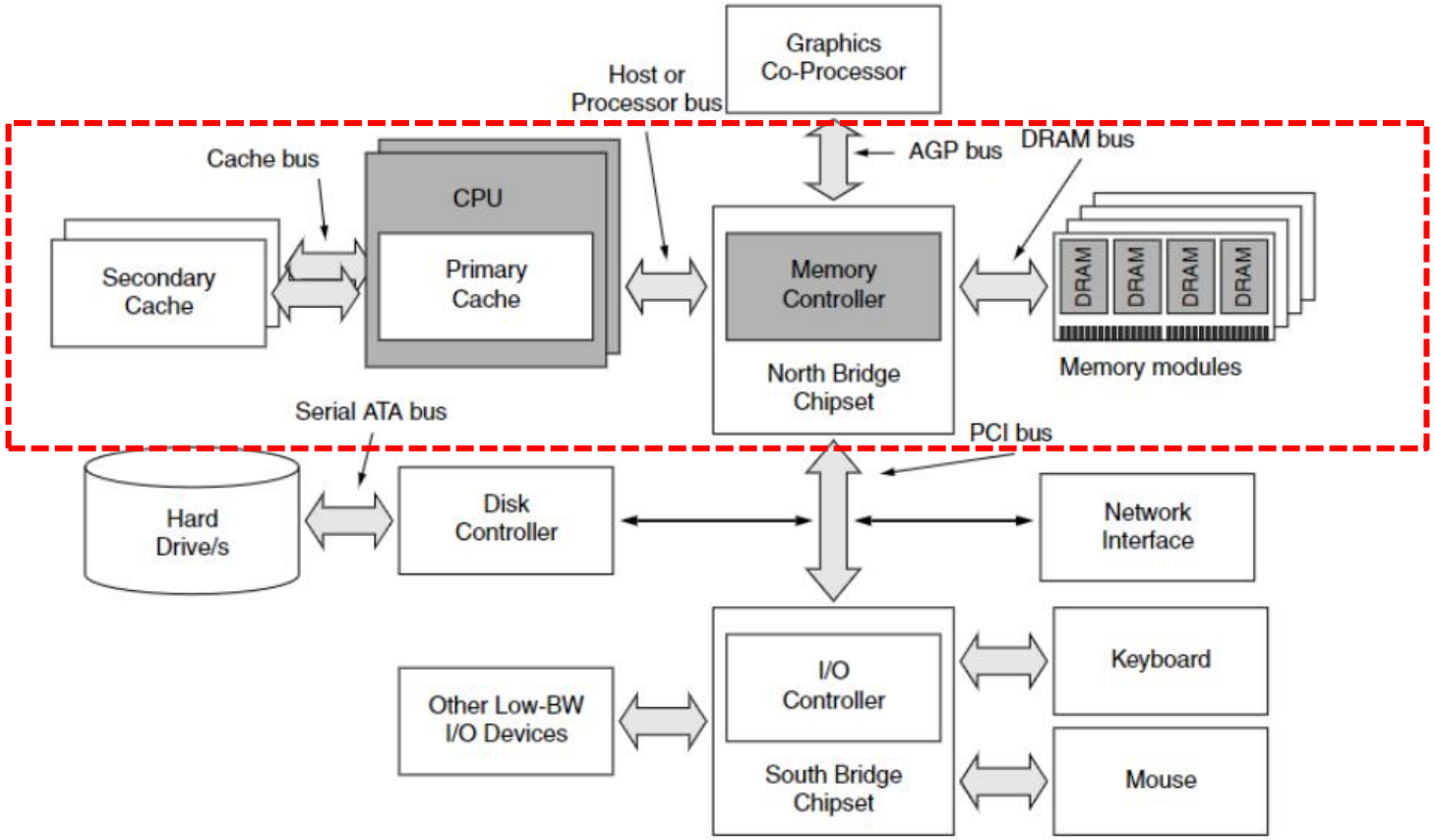
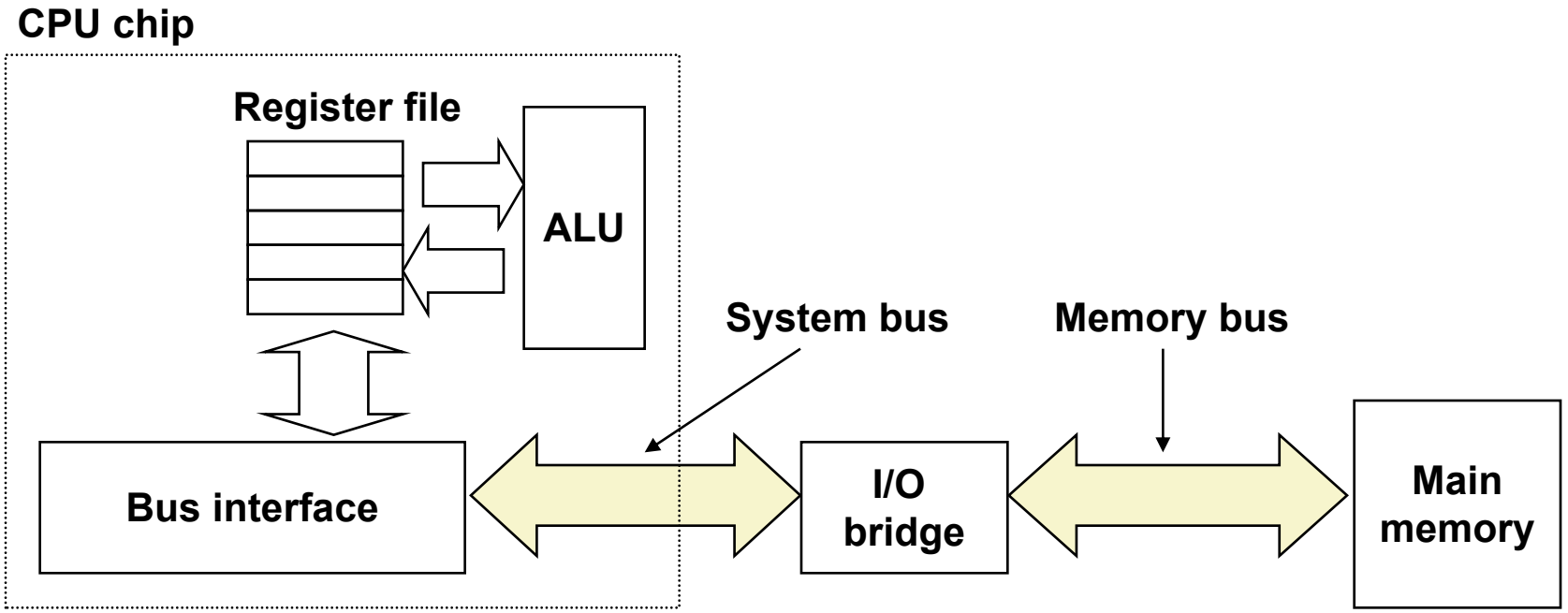


FIGURE 7.2: A typical PC organization. The DRAM subsystem is one part of a relatively complex whole. This figure illustrates a two-way multi-processor, with each processor having its own dedicated secondary cache. The parts most relevant to this report are shaded in darker grey: the CPU, the memory controller, and the individual DRAMs.

CPU与Memory的连接

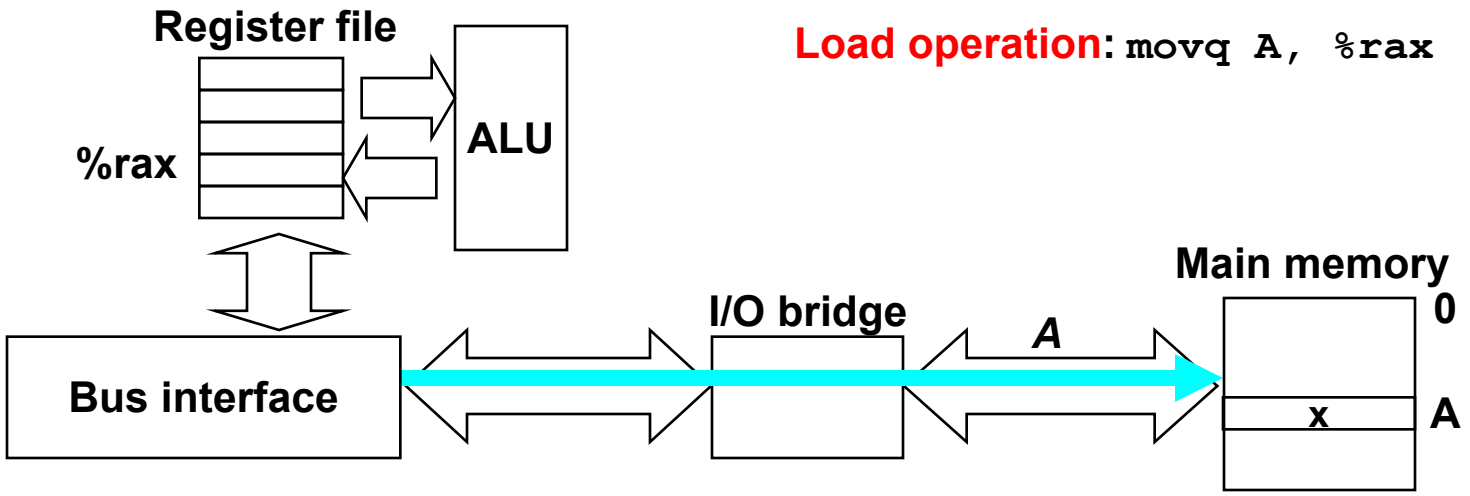
Traditional Bus Structure Connecting CPU and Memory

- **存储器通常通过总线与CPU链接（通信）**
- **总线(bus)是承载地址、数据和控制信号的并行导线的集合**
- **总线(bus)通常连接有多个设备，由多个设备共享**



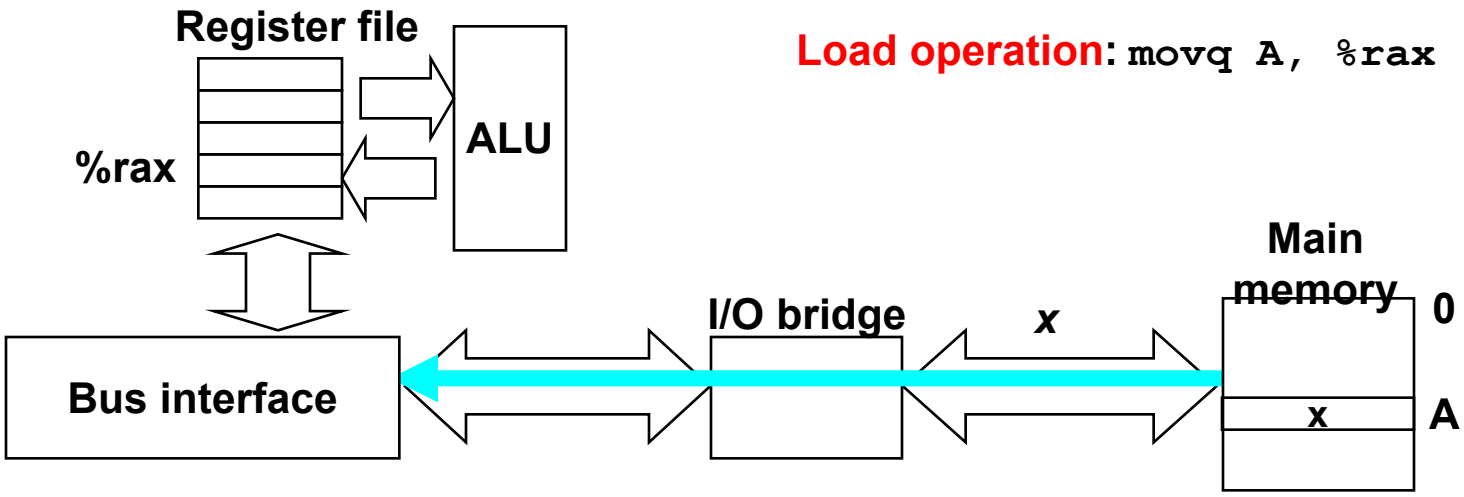
Memory Read Transaction (1/3)

1、CPU 将要读取对象的地址送到存储器总线上(memory bus)



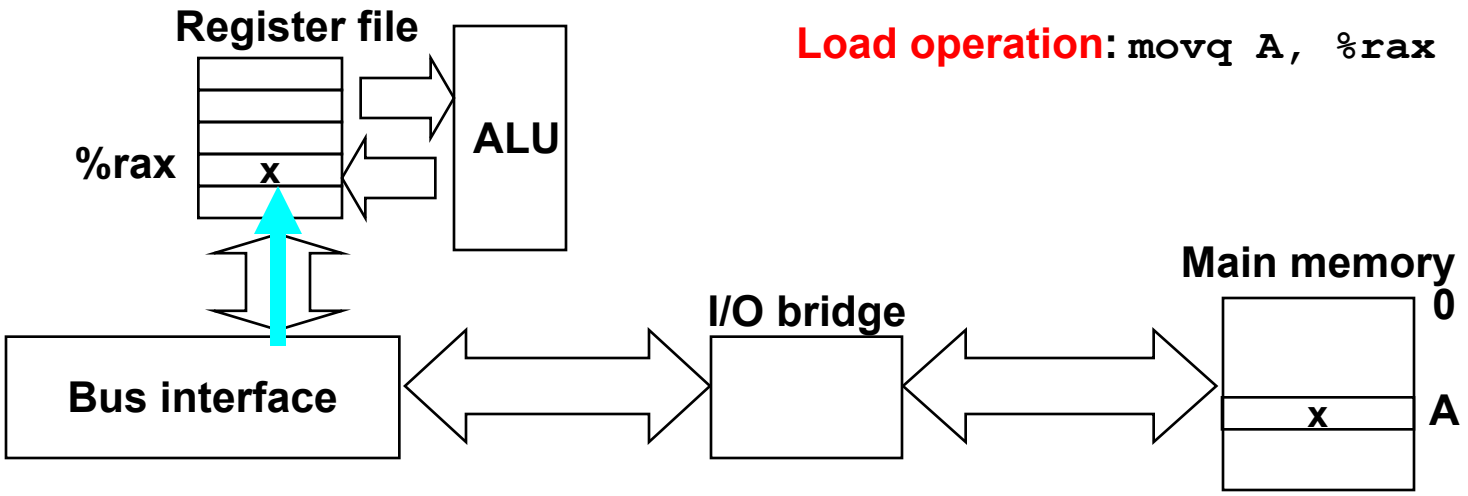
Memory Read Transaction (2/3)

2、总线控制器从存储器总线上接收地址A，从存储器中读取对象x，并送到总线上



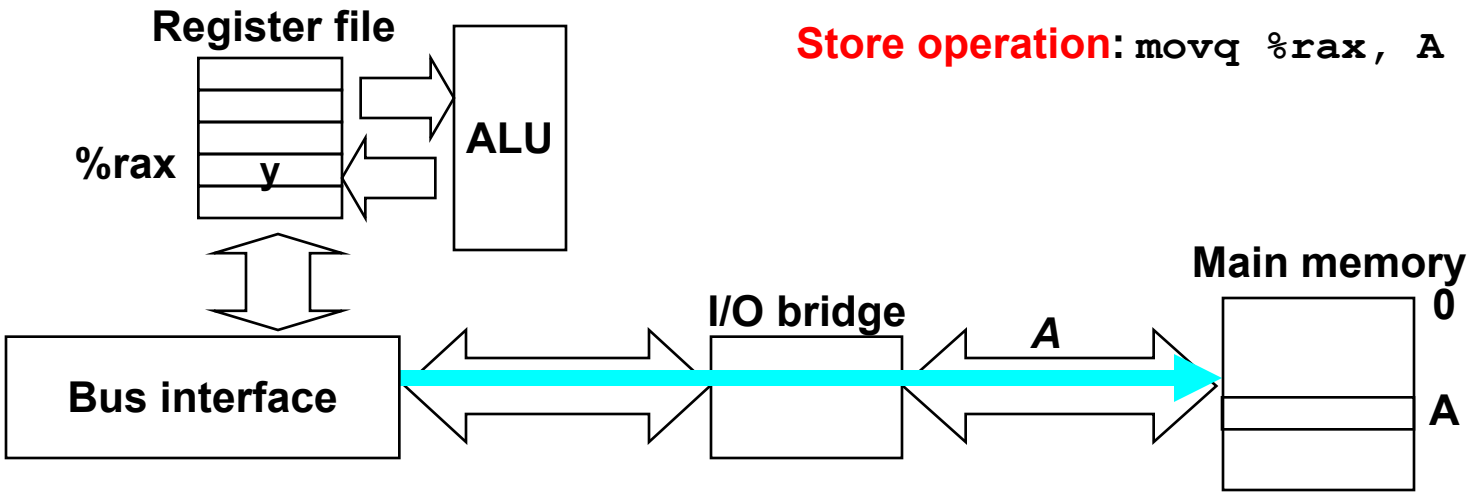
Memory Read Transaction (3/3)

3、CPU 从总线上读取x，并将其传送到寄存器中



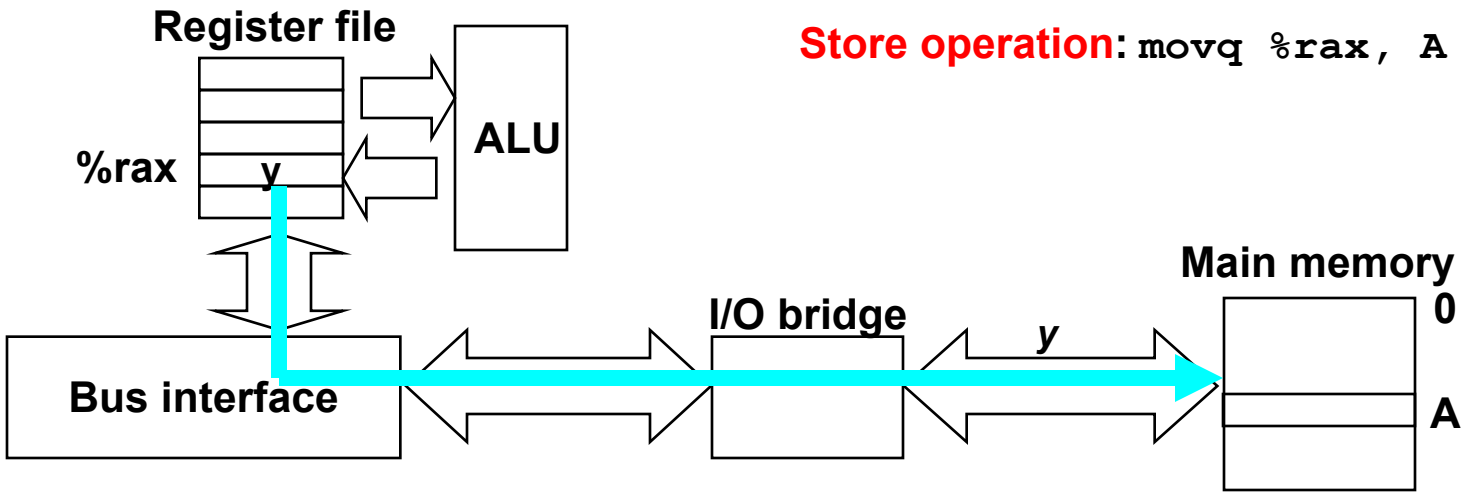
Memory Write Transaction (1/3)

1、CPU将地址传送到总线. 主存控制器读取地址，等待待写数据到达.



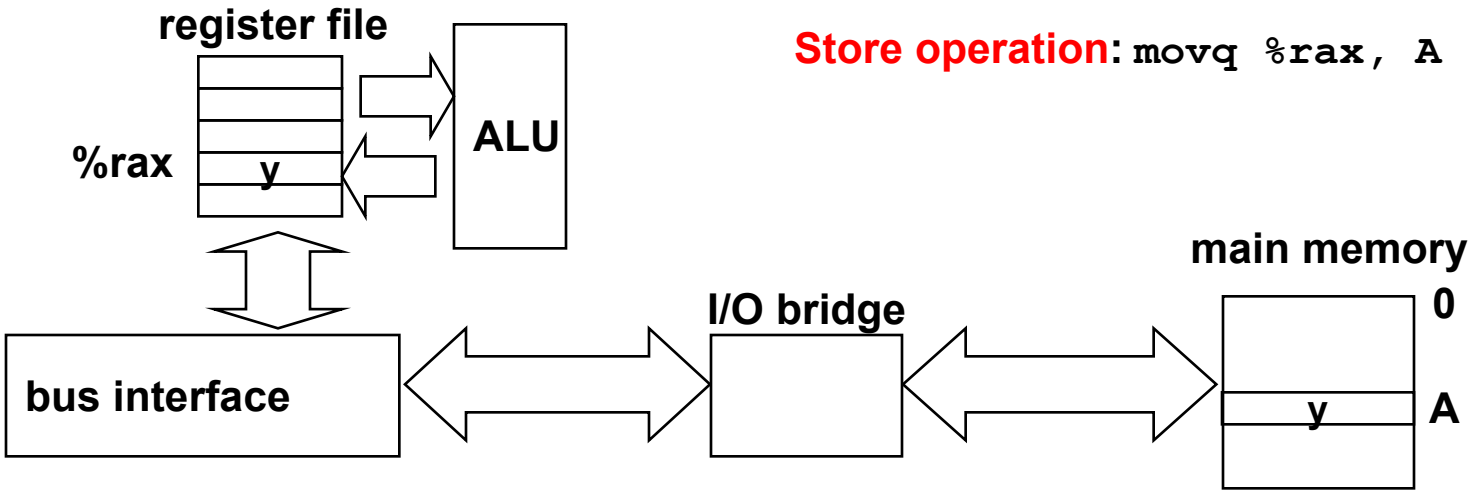
Memory Write Transaction (2/3)

2、 CPU将待写数据传送到总线



Memory Write Transaction (3/3)

3、主存控制器从总线上读取数据，将该数据存储到地址为A的存储单元中.



Memory Hierarchy

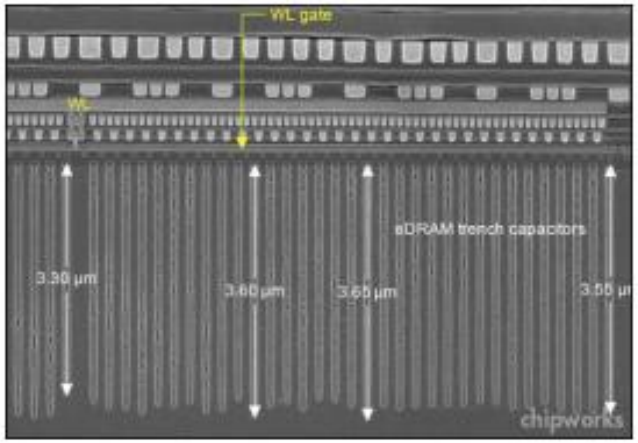
- The memory abstraction
- **RAM : main memory building block**
- Locality of reference
- The memory hierarchy
- Storage technologies and trends

Random-Access Memory (RAM)

- **冯诺伊曼计算机中：存储器指主存**
- **主要特性**
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.
- **静态随机存储器(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.
- **动态随机存储器(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.

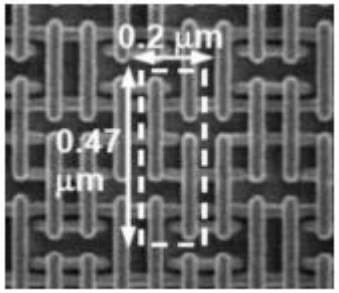
RAM Technologies

- DRAM



- 1 Transistor + 1 capacitor / bit
  - Capacitor oriented vertically
- Must refresh state periodically

- SRAM



- 6 transistors / bit
- Holds state indefinitely

SRAM vs DRAM Summary

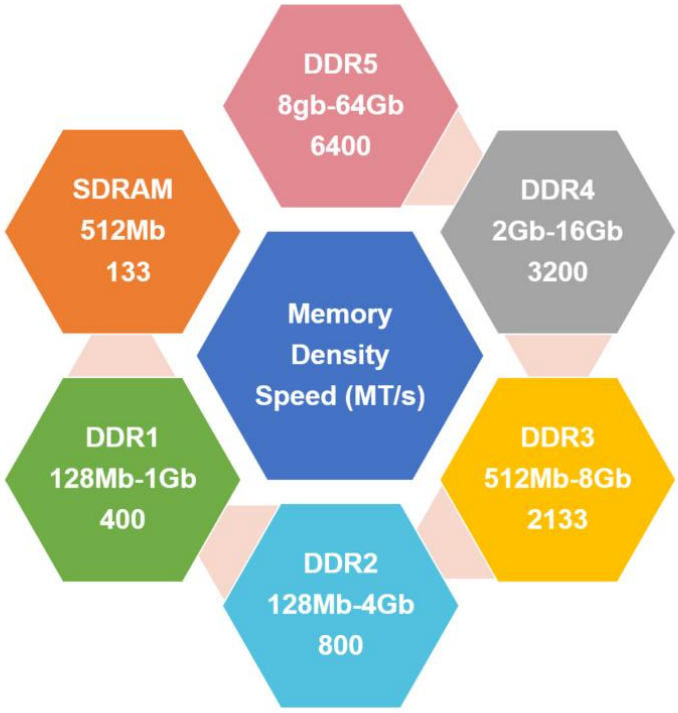
	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	6 or 8	1x	No	Maybe	100x	Cache memories
DRAM	1	10x	Yes	Yes	1x	Main memories, frame buffers

EDC: Error detection and correction

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

Enhanced DRAMs

- **DRAM的基本操作自发明以来没有变化**
  - Commercialized by Intel in 1970.
- **DRAM 核心具有更好的接口逻辑和更快的I/O速度:**
  - Synchronous DRAM (SDRAM)
  - Double data-rate synchronous DRAM (DDR SDRAM)
    - Double edge clocking sends two bits per cycle per pin
    - Different types distinguished by size of small prefetch buffer:
      - DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits), DDR4 (8bits)
    - By 2010, standard for most server and desktop systems
    - Intel Core i7 supports DDR3 and DDR4 SDRAM
  - HBM: HighBandwidth Memory (460GB/s)



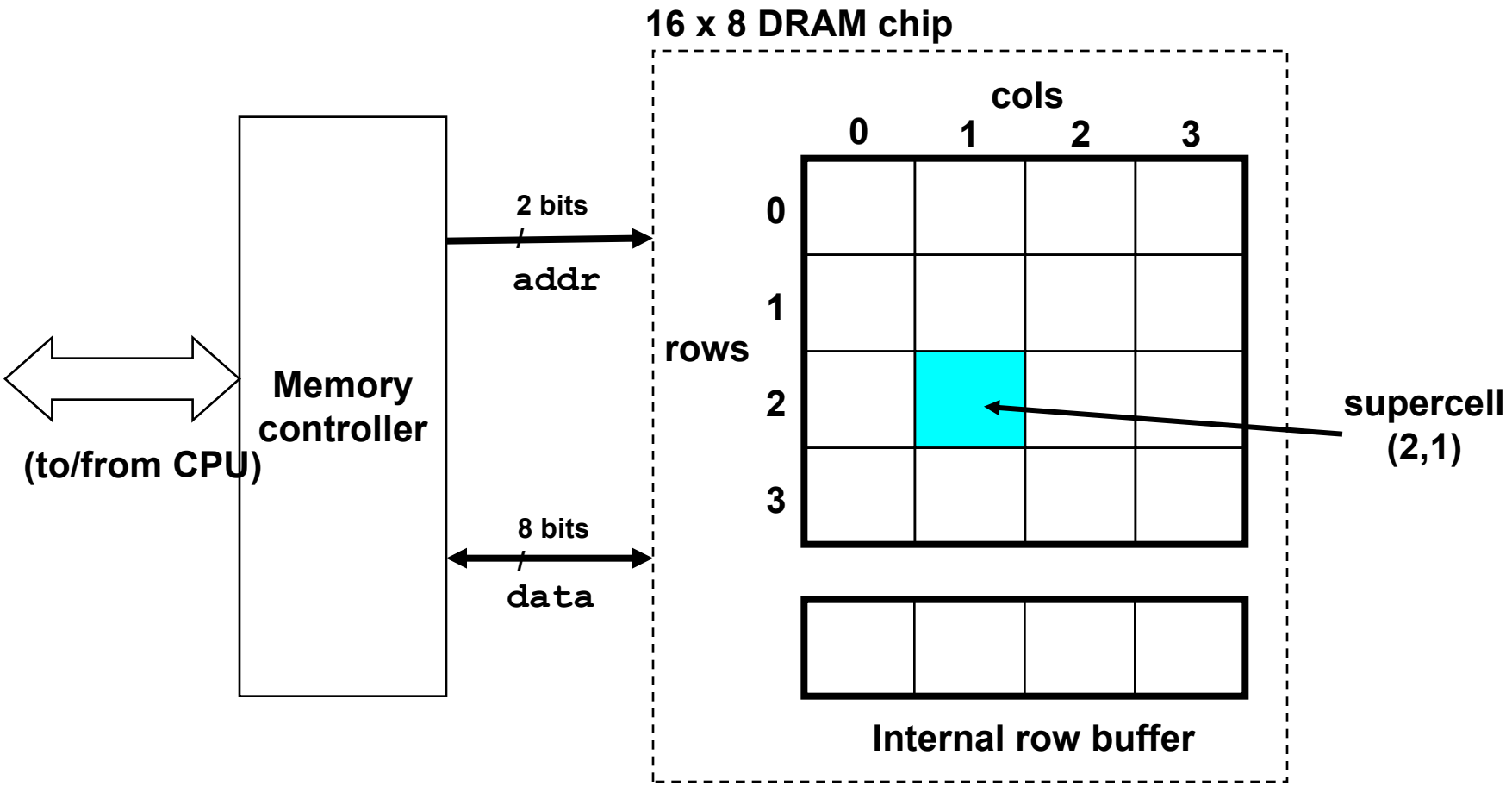
DRAM Memory Densities and Speed

Enhanced DRAMs

	Prefetch	Core Frequency	IO CLK Frequency	IO Data Rate
SDRAM	NA	100-150 MHz	100-150 MHz	100-150 Mbps
DDR	2	100-200 MHz	100-200 MHz	200-400 Mbps
DDR2	4	100-200 MHz	200-400 MHz	400-800 Mbps
DDR3	8	100-266 MHz	400-1066 MHz	800-2133 Mbps
DDR4	8	100-266 MHz	800-1600 MHz	1600-3200 Mbps
DDR5*	16	100-266 MHz	1600-3200 MHz	3200-6400 Mbps

Conventional DRAM Organization

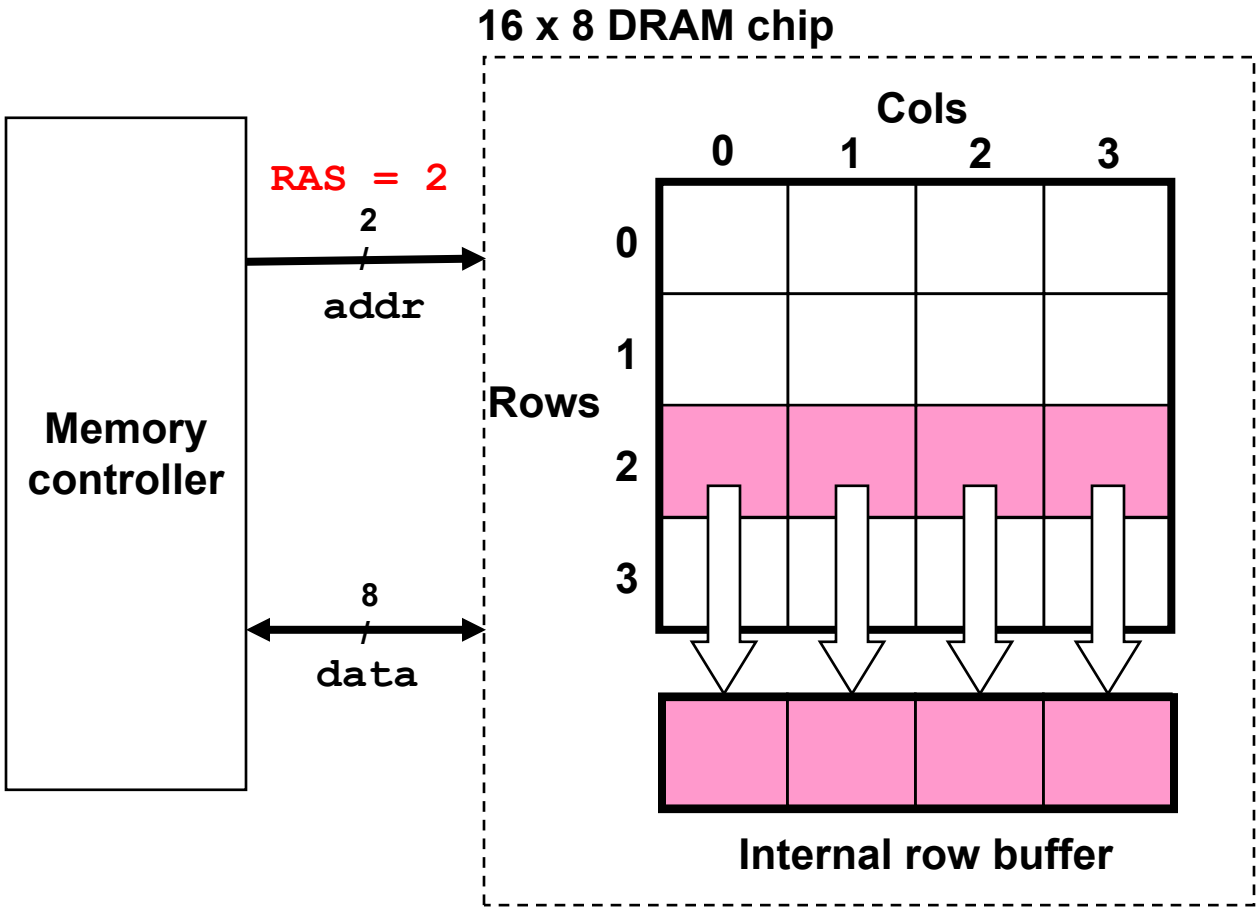
- **d x w DRAM:**
  - d x w total bits organized as d supercells of size w bits



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.

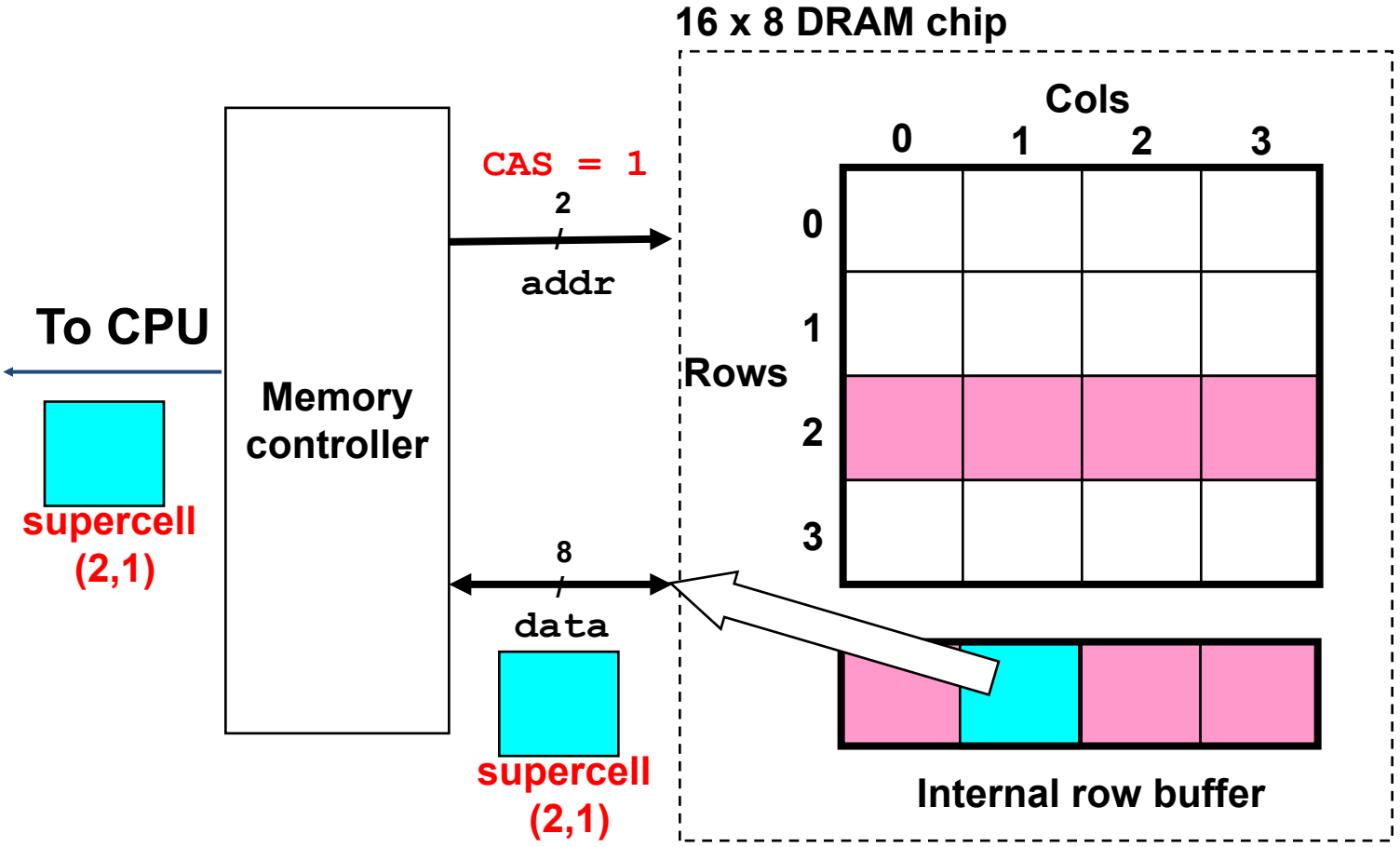


Reading DRAM Supercell (2,1)

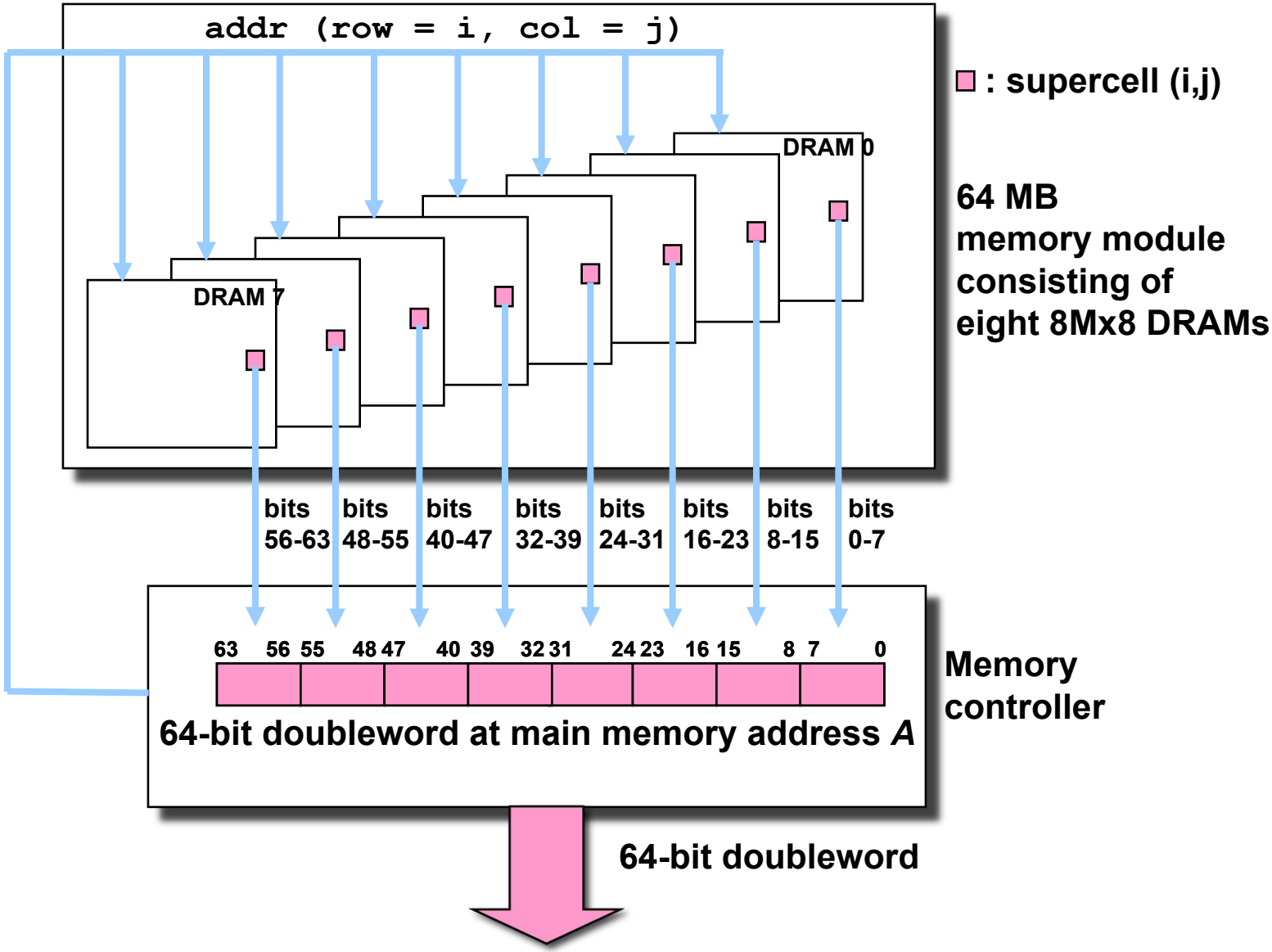
Step 2(a): Column access strobe (CAS) selects column 1.

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

Step 3: All data written back to row to provide refresh



Memory Modules



Memory Module Nomenclature

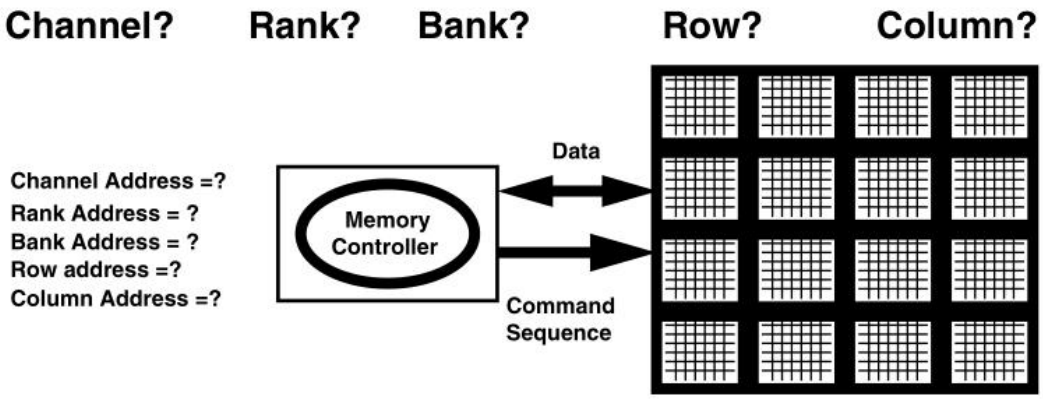
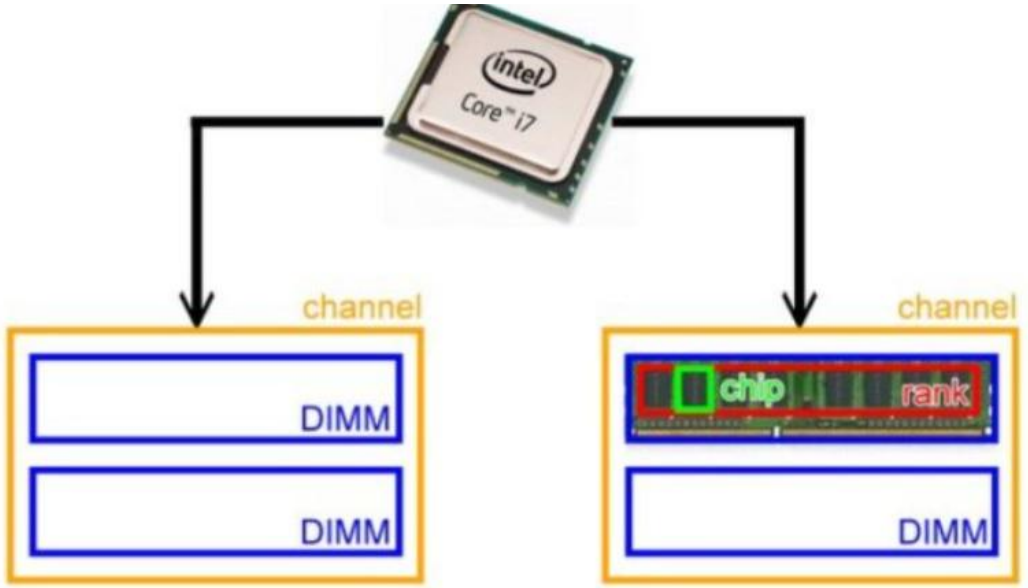


FIGURE 10.1: Multiple DRAM devices connected to a processor through a DRAM memory controller.



Channel

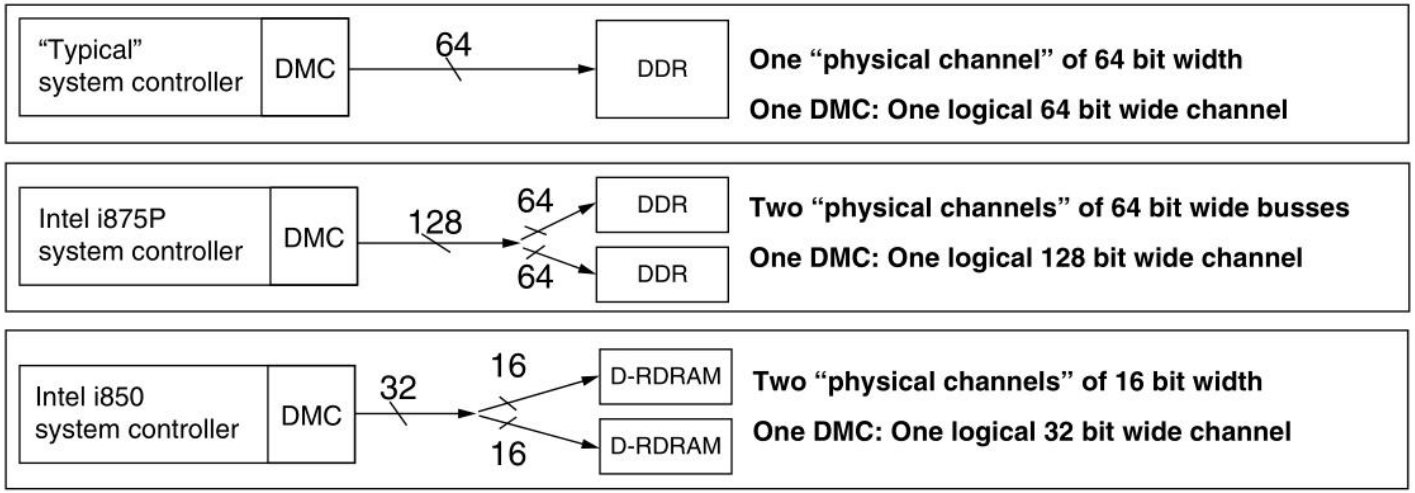


FIGURE 10.2: Systems with a single memory controller and different data bus widths.

DRAM memory controller (DMC)

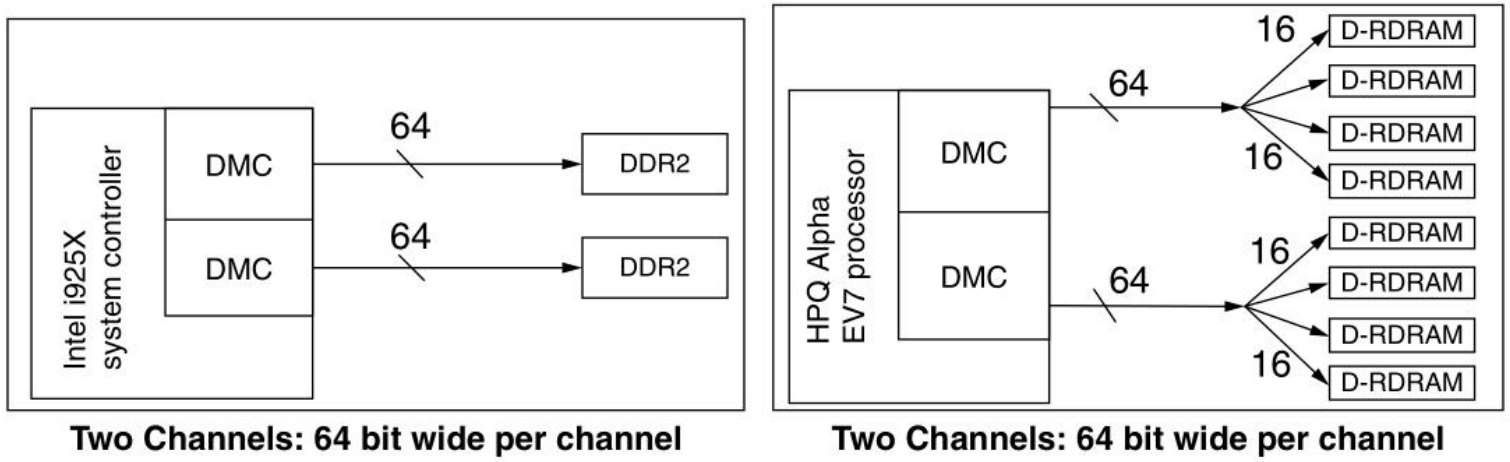


FIGURE 10.3: Systems with two independent memory controllers and two logical channels of memory.

Rank

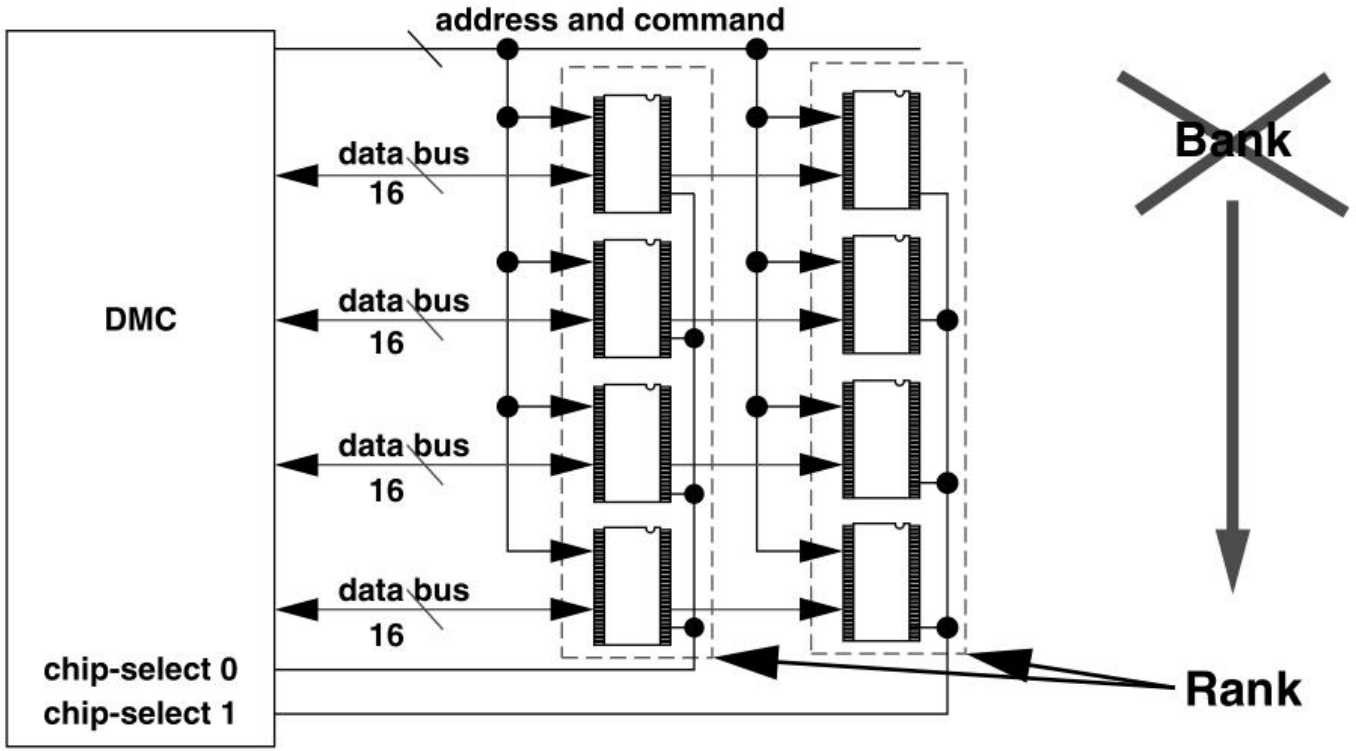


FIGURE 10.5: Memory system with 2 ranks of DRAM devices.

Bank

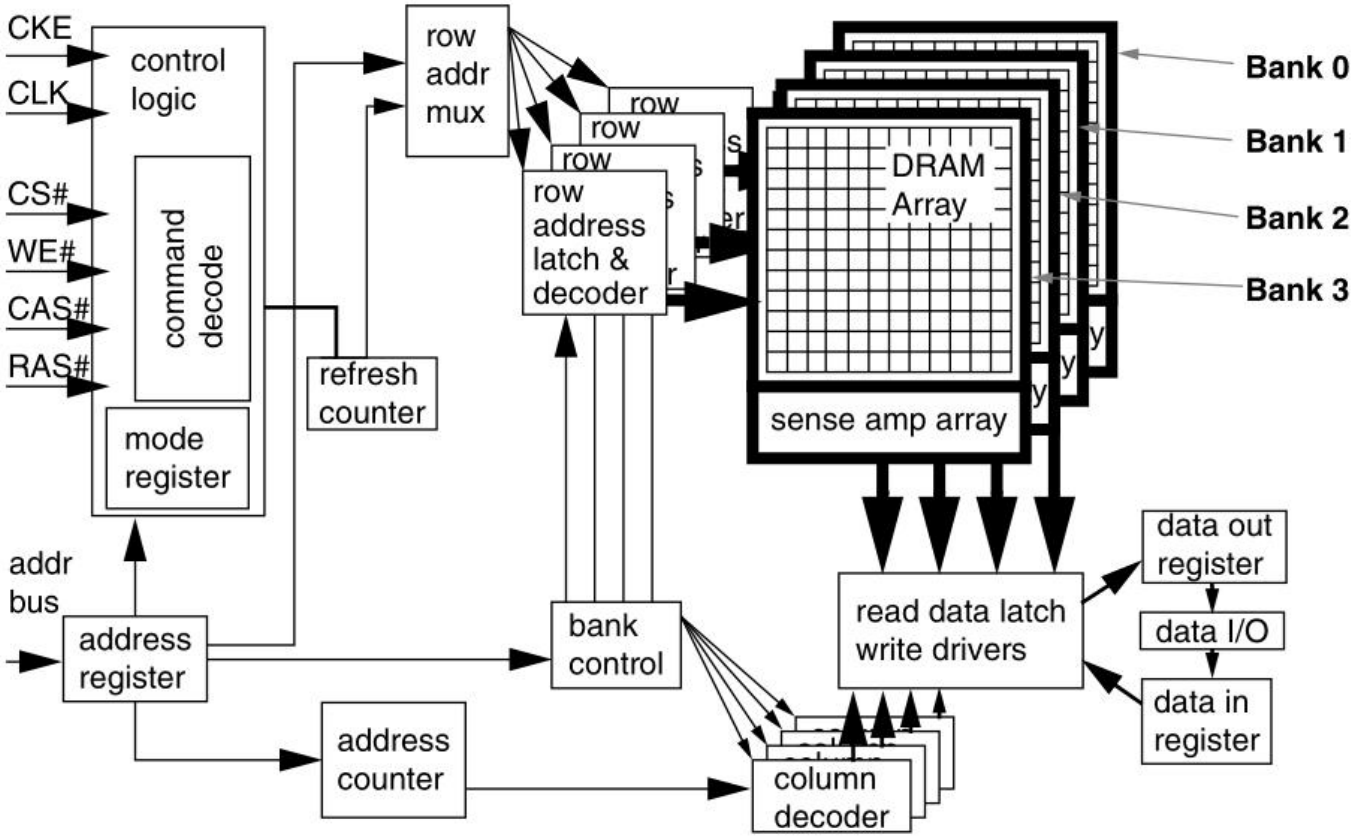


FIGURE 10.6: SDRAM device with 4 banks of DRAM arrays internally.

Memory Module

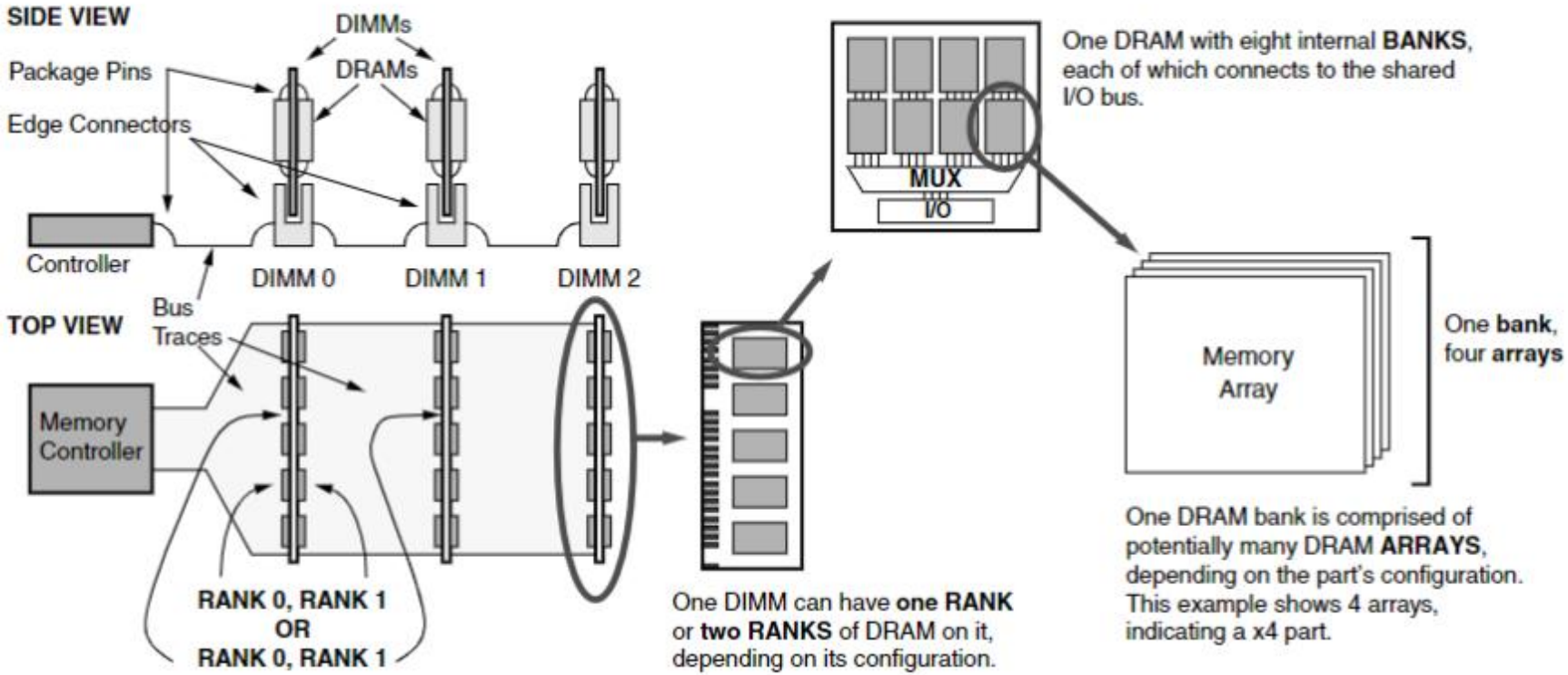


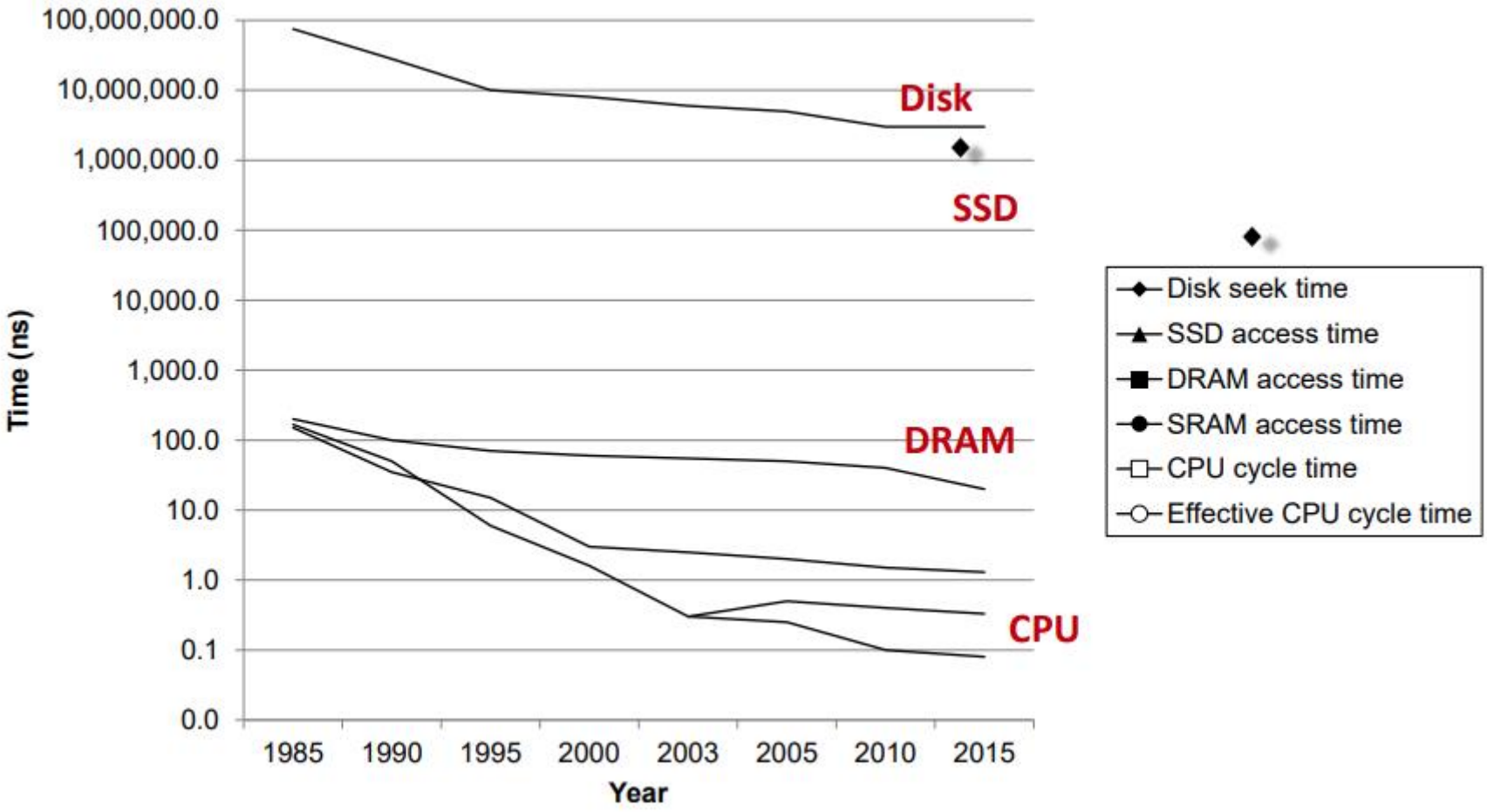
FIGURE 7.5: DIMMs, ranks, banks, and arrays. A system has potentially many DIMMs, each of which may contain one or more ranks. Each rank is a set of ganged DRAM devices, each of which has potentially many banks. Each bank has potentially many constituent arrays, depending on the part's data width.

Memory Hierarchy

- The memory abstraction
- RAM : main memory building block
- **Locality of reference**
- The memory hierarchy
- Storage technologies and trends

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



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

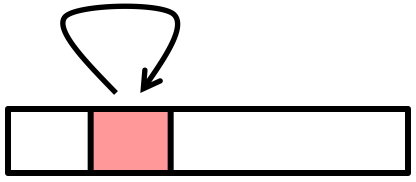
Locality to the Rescue!

弥补CPU -内存之间性能差异的关键是程序的基本属性，即局部性(locality)

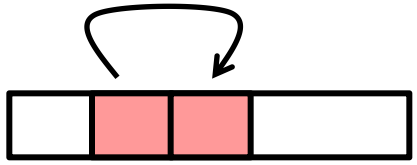
Locality

- **Principle of Locality: 程序倾向于使用地址接近或等于它们最近使用的地址的数据和指令**

- **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;
```

- **数据访问**
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable `sum` each iteration.

Spatial locality

Temporal locality

- **指令读取**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

Spatial locality

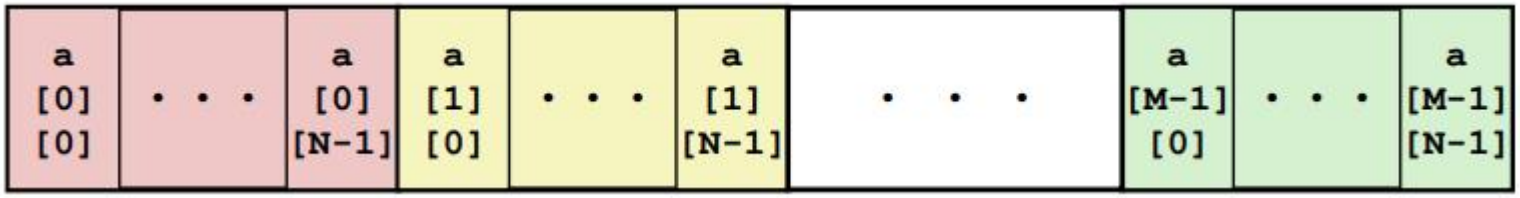
Temporal locality

Qualitative Estimates of Locality

- **Claim:** 对于专业程序员来说，能够查看代码并定性地了解其局部性是一项关键技能
- **Question:** 下列函数对数组a来说，局部性如何?

```
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;
}
```

array layout is row-major order



Locality Example

- **Question:** 下列函数对数组a的访问局部性如何?

```
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: 是否能选择循环的合适排列方式，使该函数（在内存中）以步长1访问模式访问3维数组a（这样具有最好的局部性）？**

```
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 Hierarchy

- The memory abstraction
- RAM : main memory building block
- Locality of reference
- **The memory hierarchy**
- Storage technologies and trends

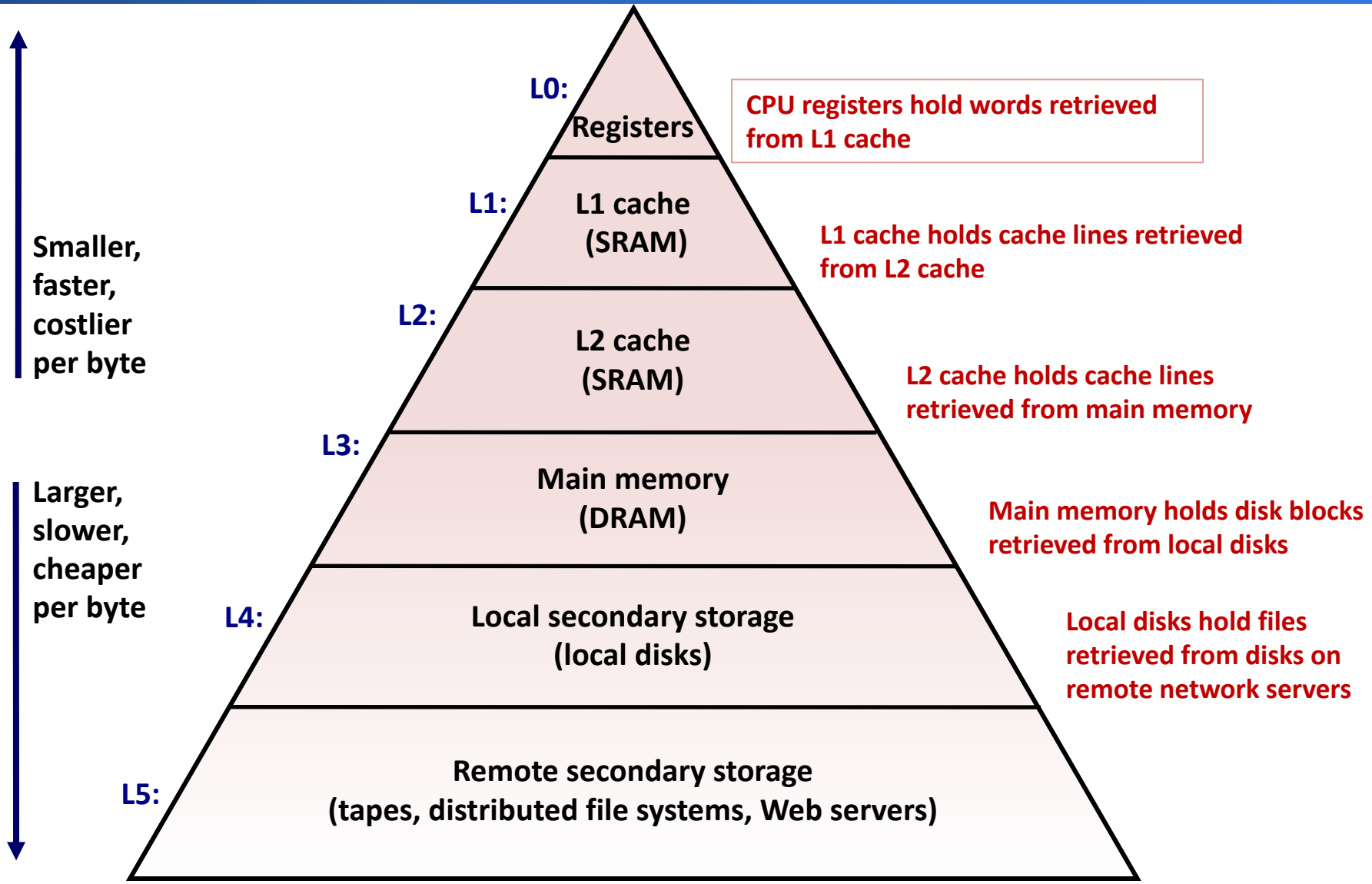
Memory Hierarchies

- **硬件和软件的一些基本的固有特性:**
  - 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.

- **这些基本特性完美地相互补充.**

- **研究者们提出了一种组织内外存存储系统的方法，称为存储层次结构(memory hierarchy).**

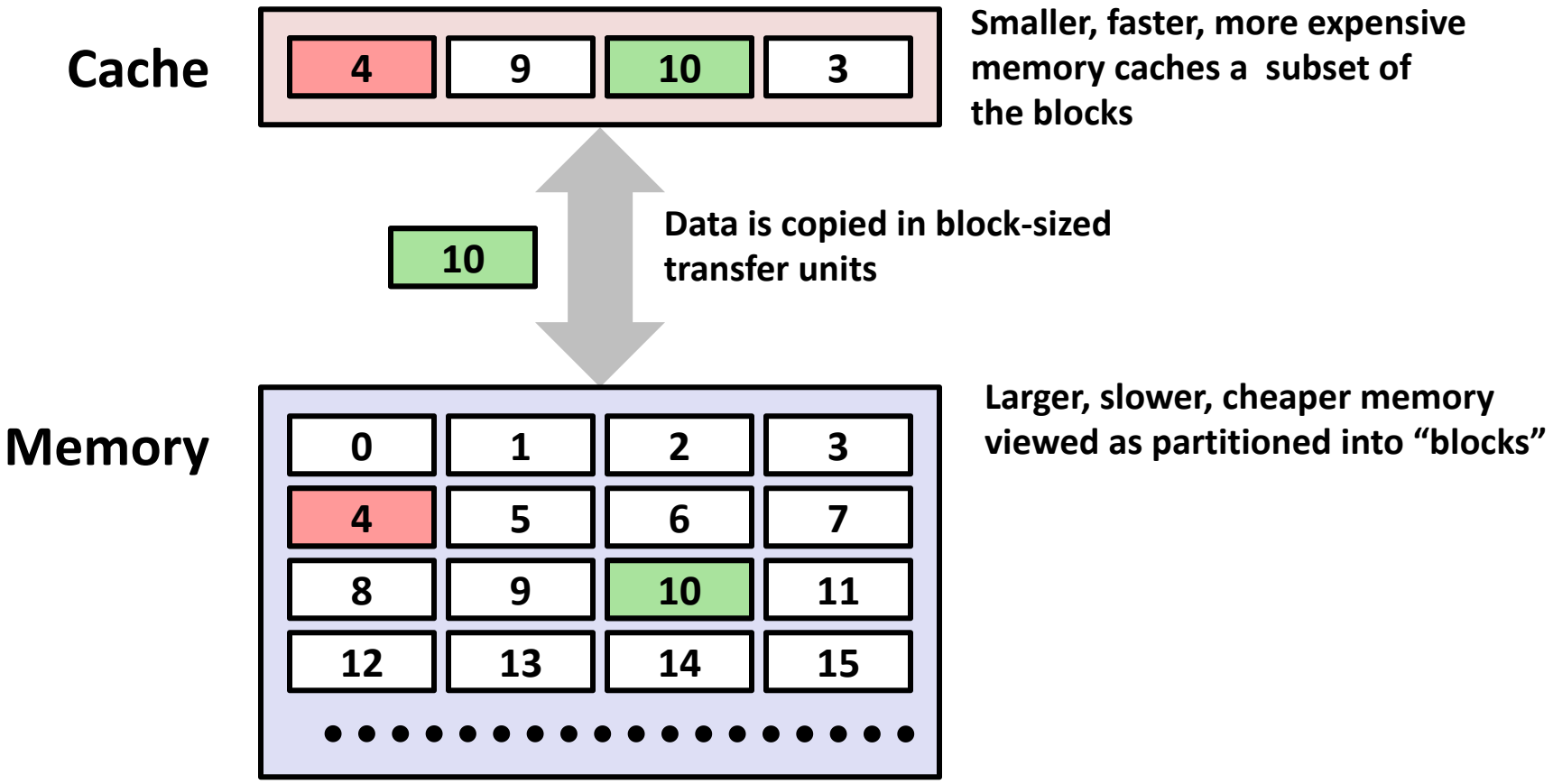
An Example Memory Hierarchy



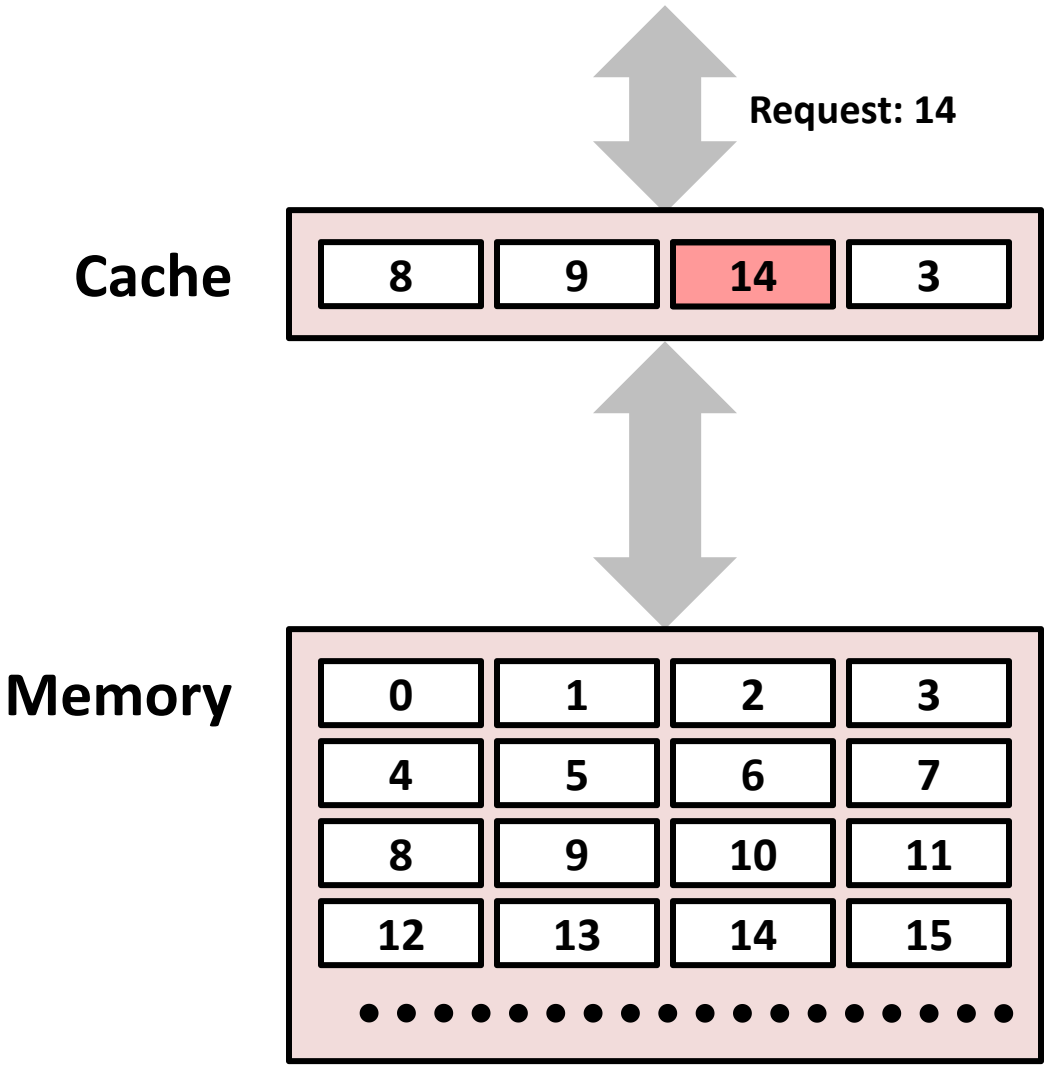
Caches

- ***Cache:***
  - **一种更小、更快的存储设备，它充当较大、较慢设备中数据子集的暂存区域。**
- **存储层次的基本思想:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- **为什么存储层次系统可以有效地工作?**
  - 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:* 存储层次结构**
  - **创建了一个大的存储池，其成本与接近底层的廉价存储接近，但它能以接近顶层的快速存储的速度为程序提供数据.**

General Cache Concepts



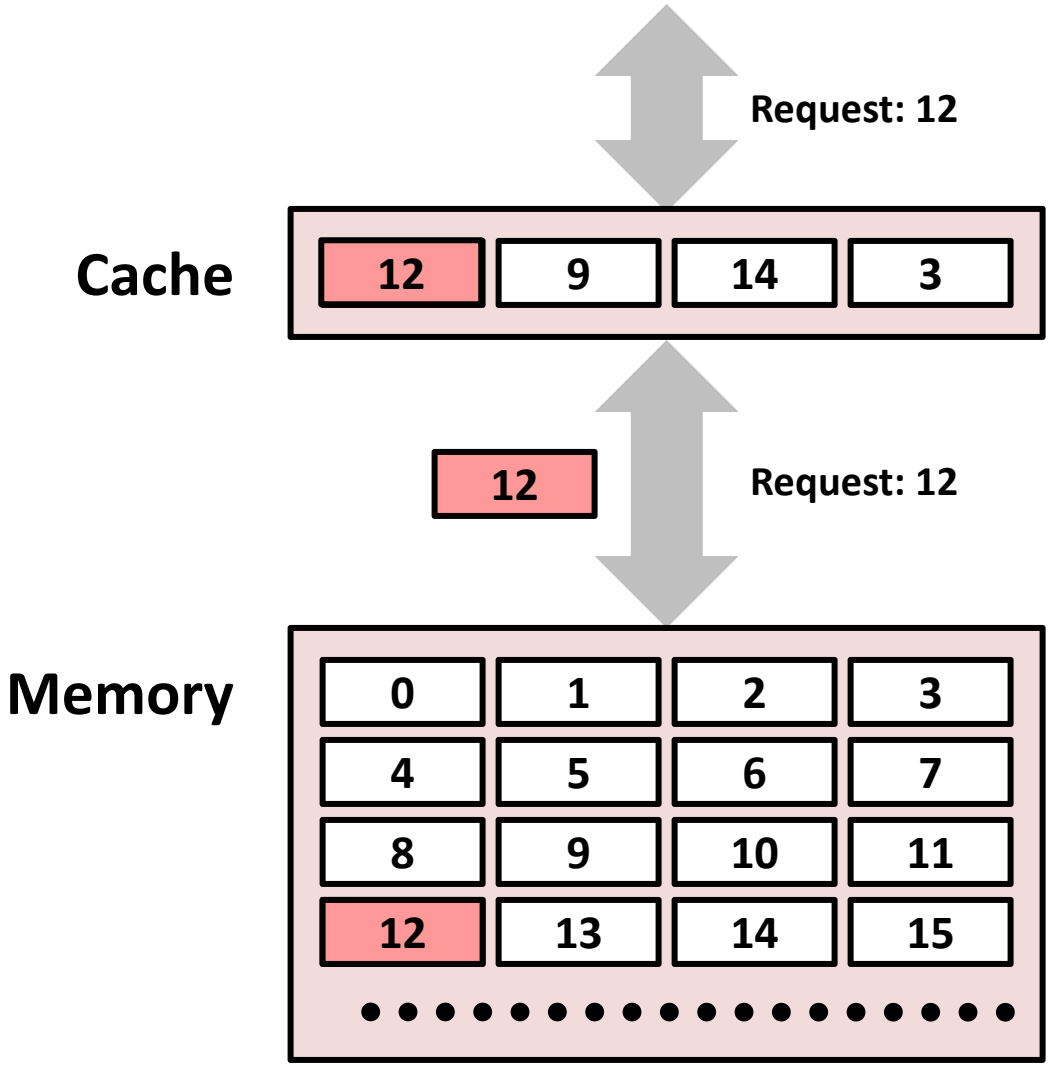
General Cache Concepts: Hit



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

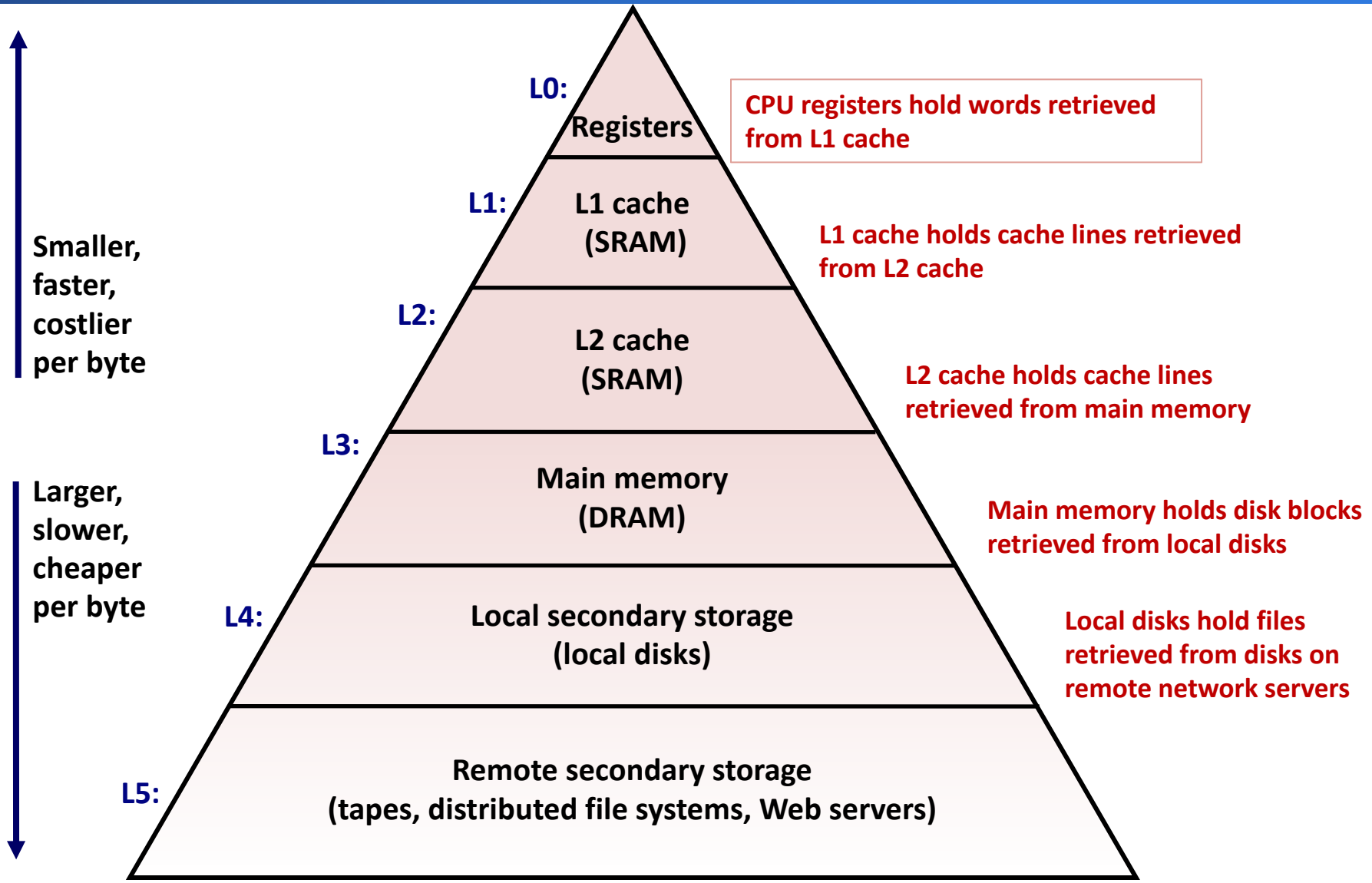
失效(Miss)的类型：3C + 1C (coherence)

- **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 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

An Example Memory Hierarchy

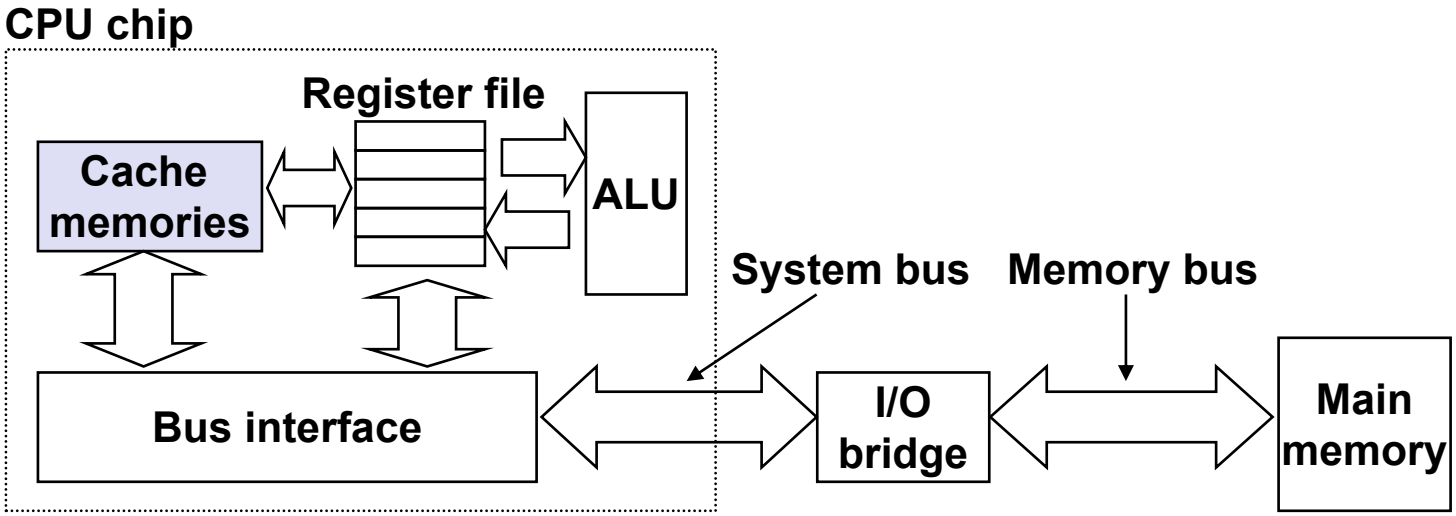


Cache Memories

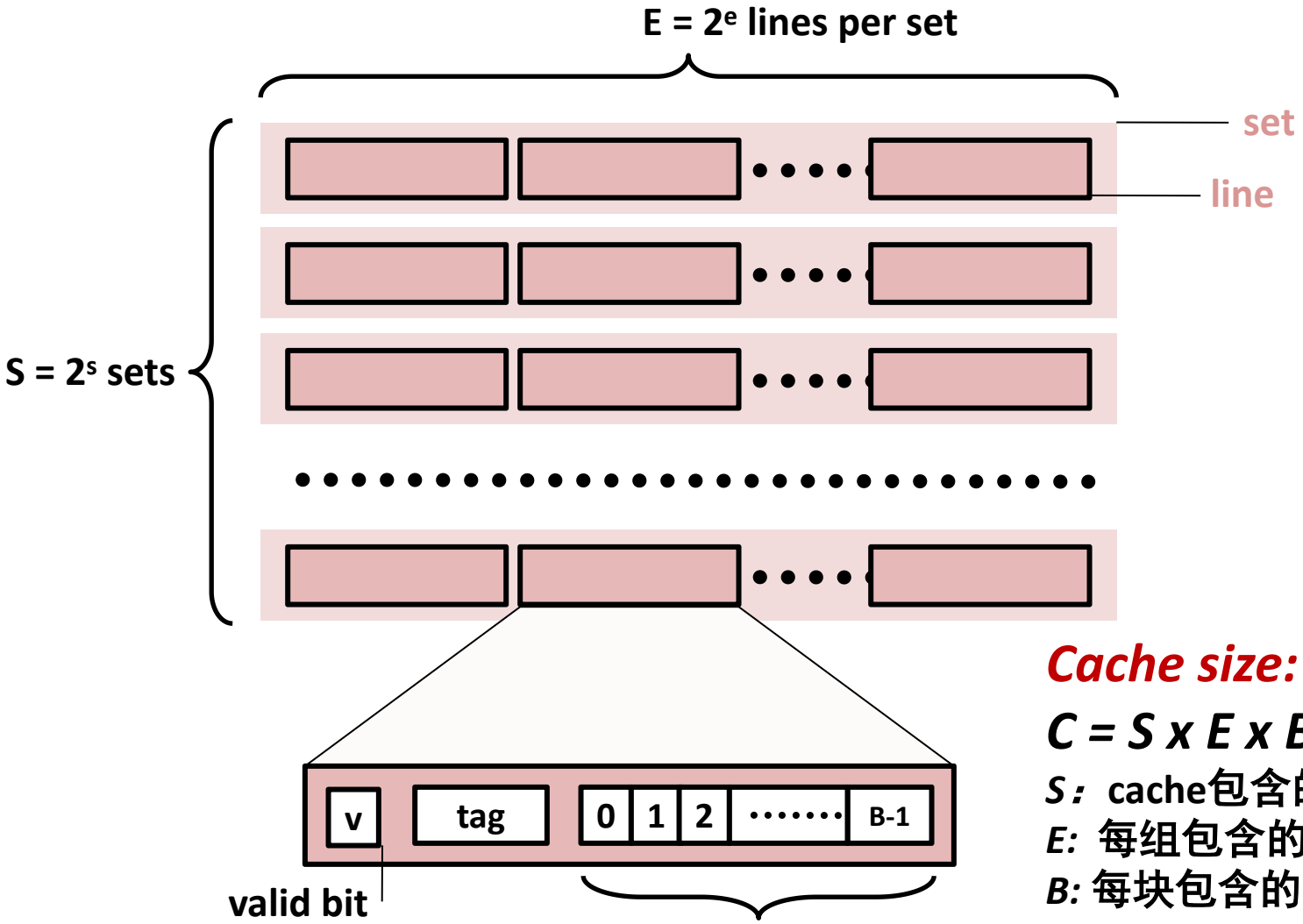
- **Cache memory organization and operation**
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Recall：Cache Memories

- **Cache memories 是容量较小、存取速度较快、由SRAM构成的存储区域，由硬件自动管理**
  - Hold frequently accessed blocks of main memory
- **CPU 首先在Cache中查找数据，如果未命中，然后访问主存.**
- **典型的带有Cache的系统结构:**



General Cache Organization (S, E, B)



Cache size:

C = S x E x B data bytes

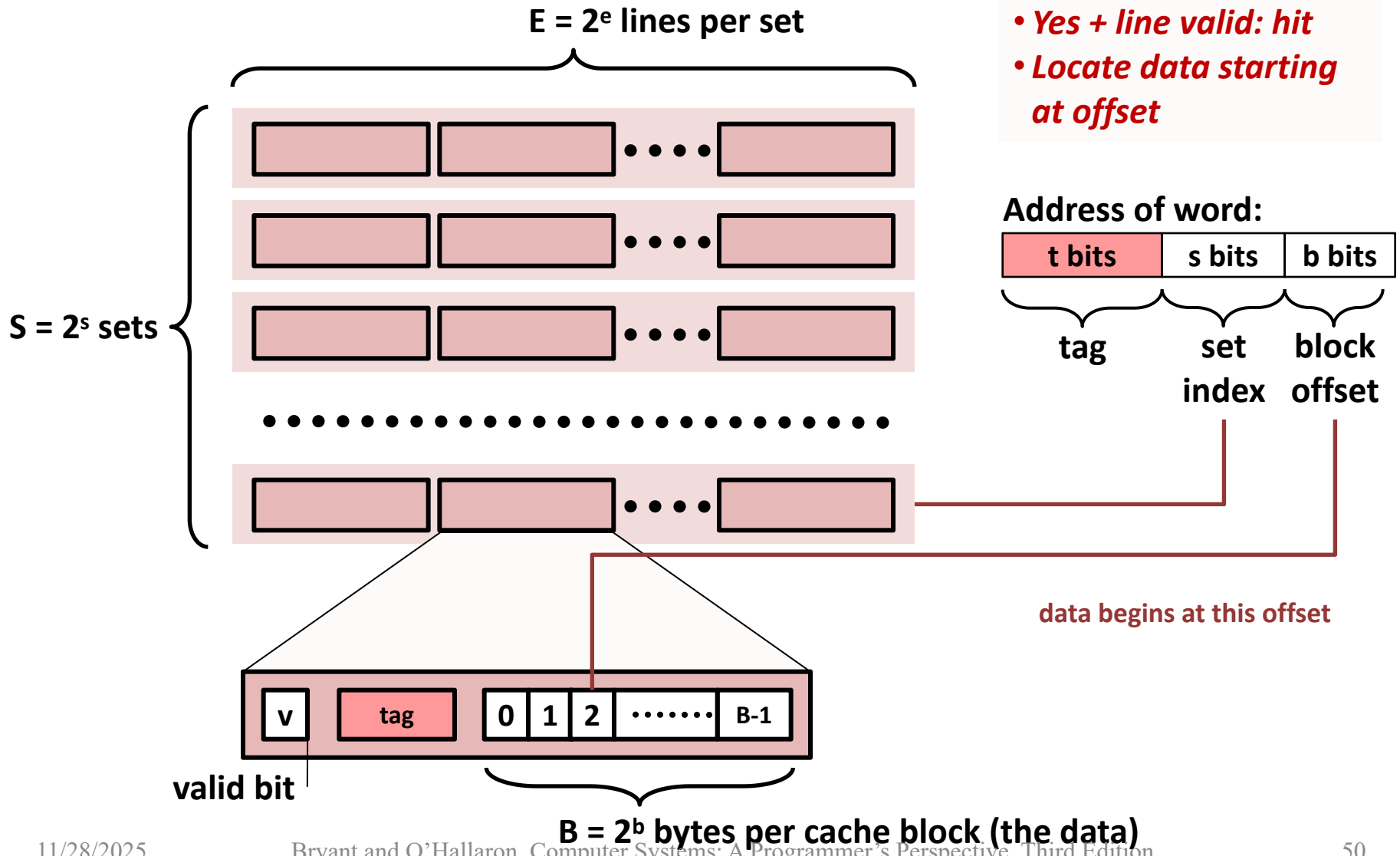
S： cache包含的组数

E: 每组包含的块数

B: 每块包含的字节数

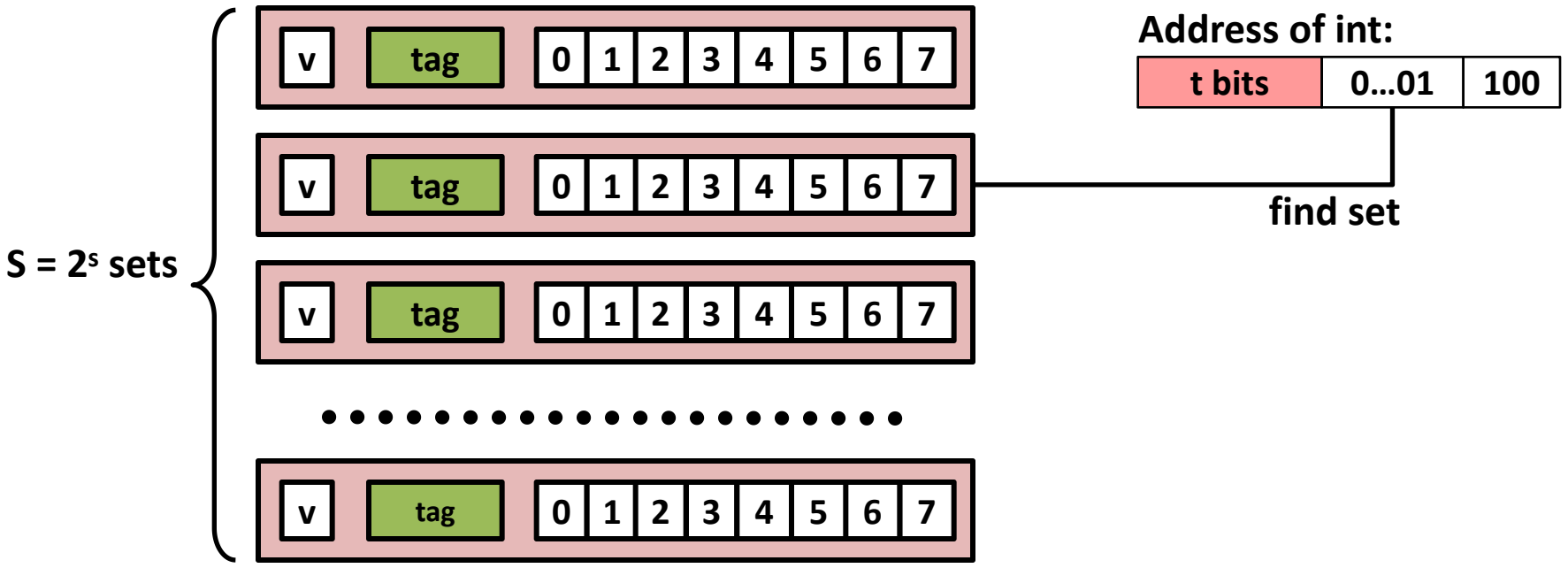
Cache Read

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



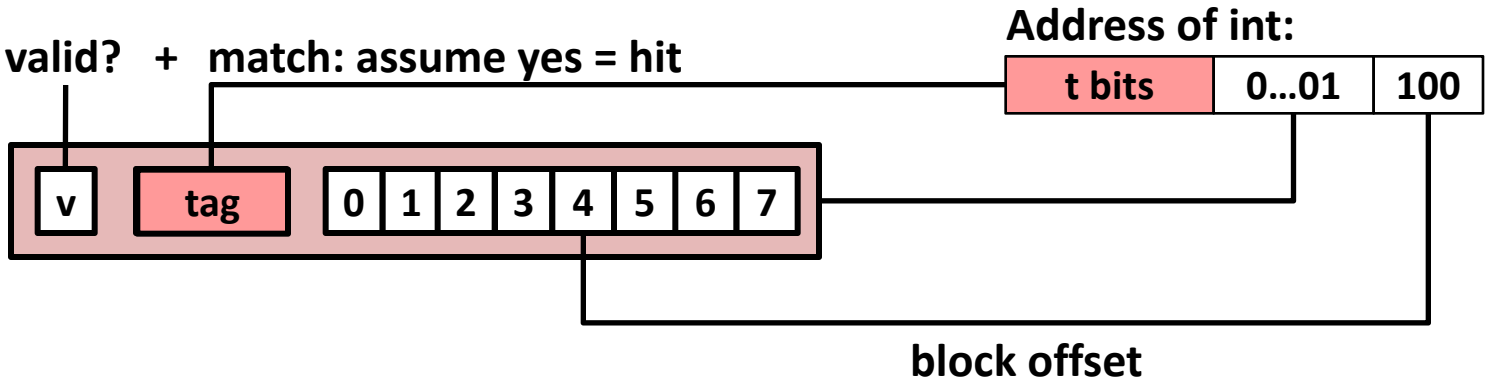
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set (Index : s bits)
Assume: cache block size 8 bytes (block offset: 3bits)



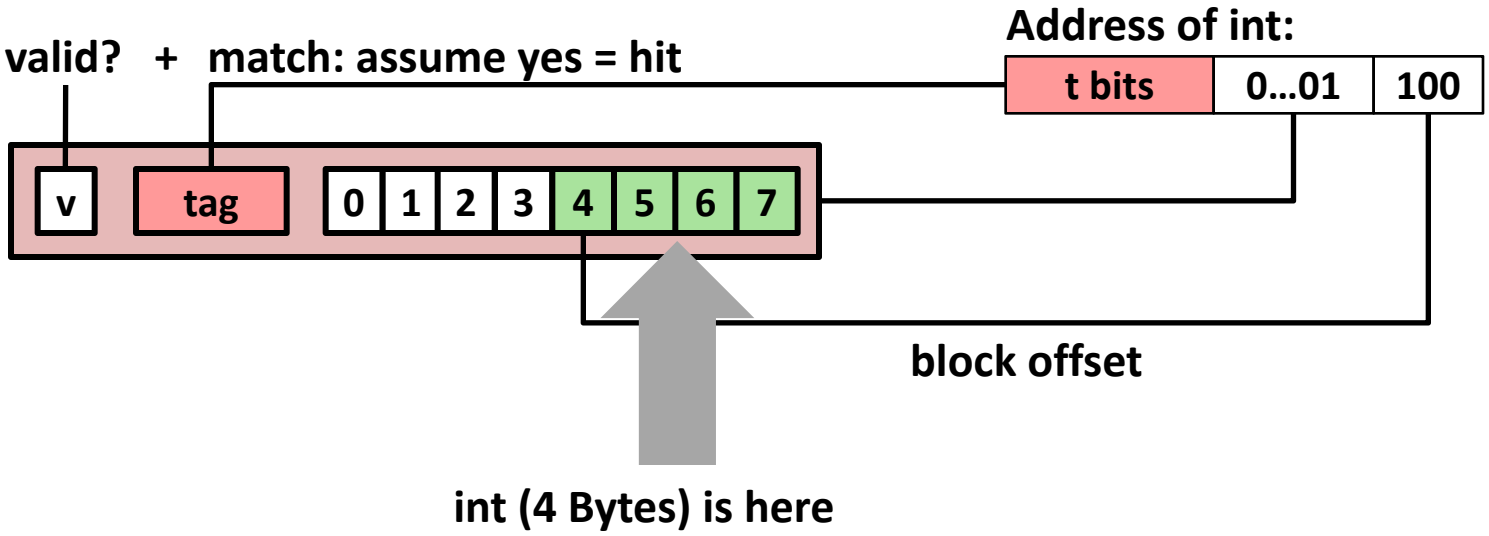
Example: Direct Mapped Cache (E = 1)

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



No match: old line is evicted and replaced

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	**[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]

Why Index Using Middle Bits?

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

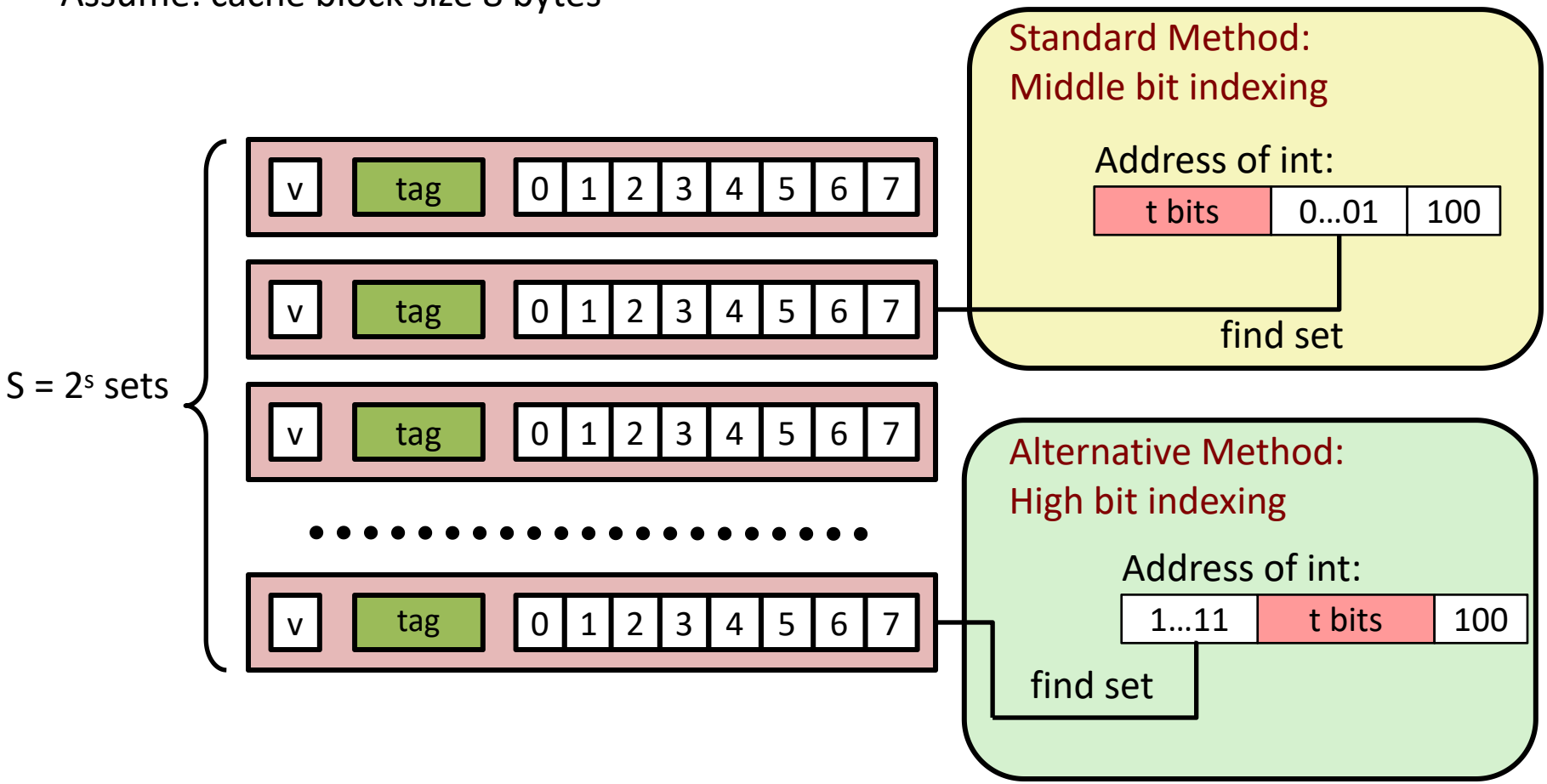
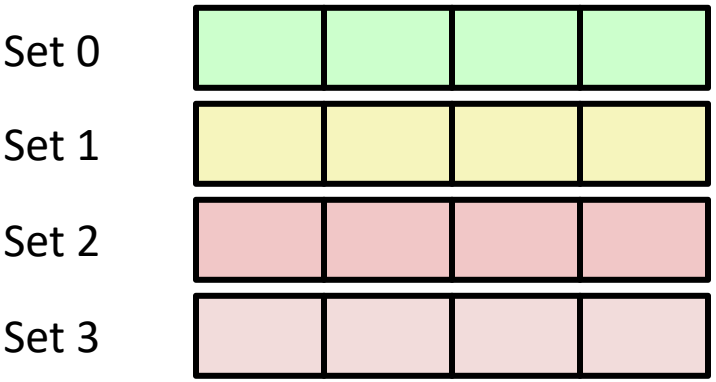
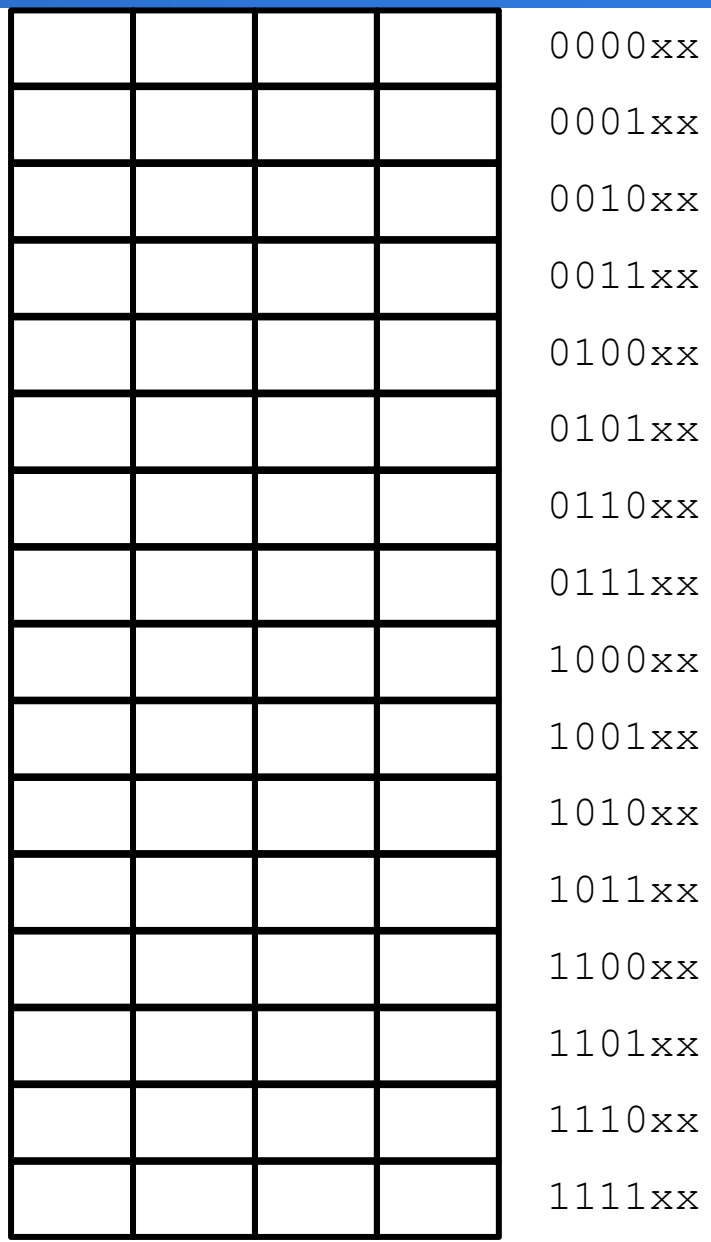


Illustration of Indexing Approaches

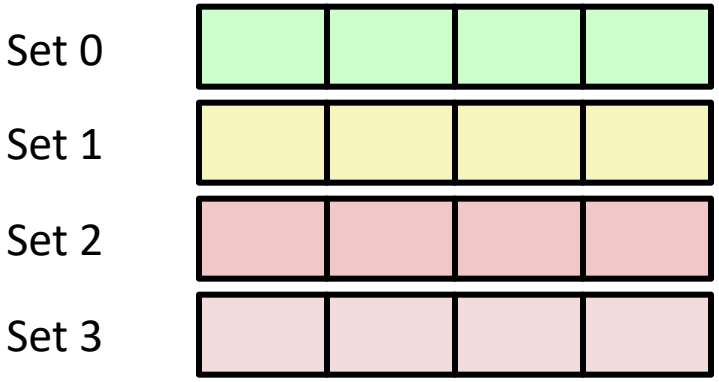
- **64-byte memory**
  - 6-bit addresses
- **16 byte, direct-mapped cache**
- **Block size = 4. (Thus, 4 sets; why?)**
- **2 bits tag, 2 bits index, 2 bits offset**



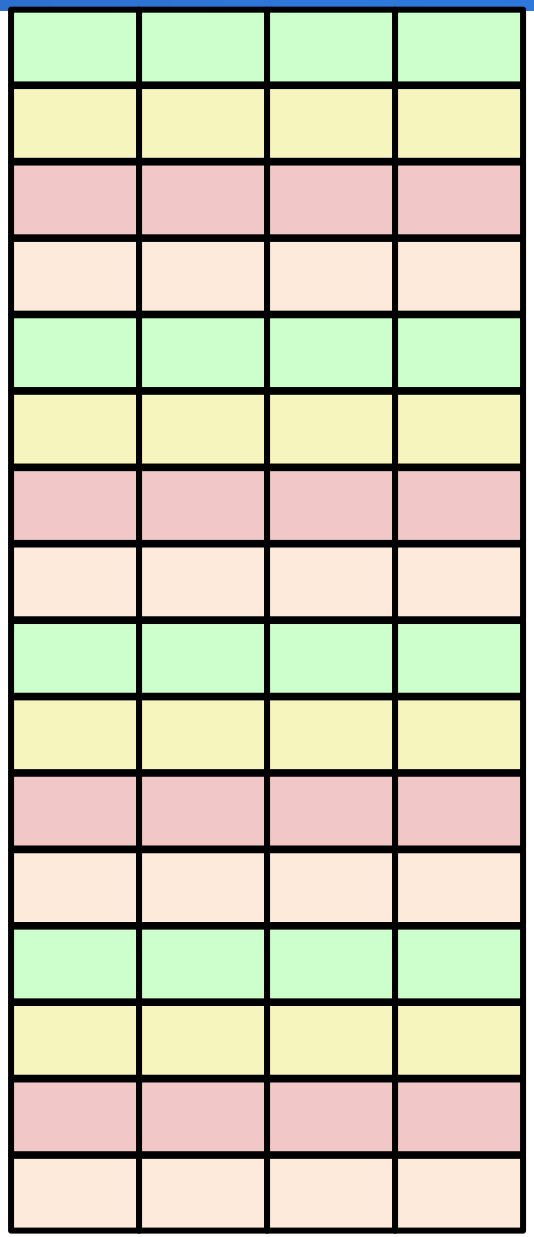


Middle Bit Indexing

- **Addresses of form TTSSBB**
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits
- **Makes good use of spatial locality**

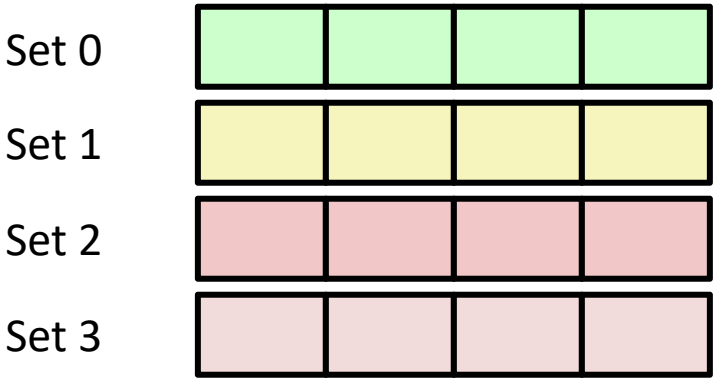


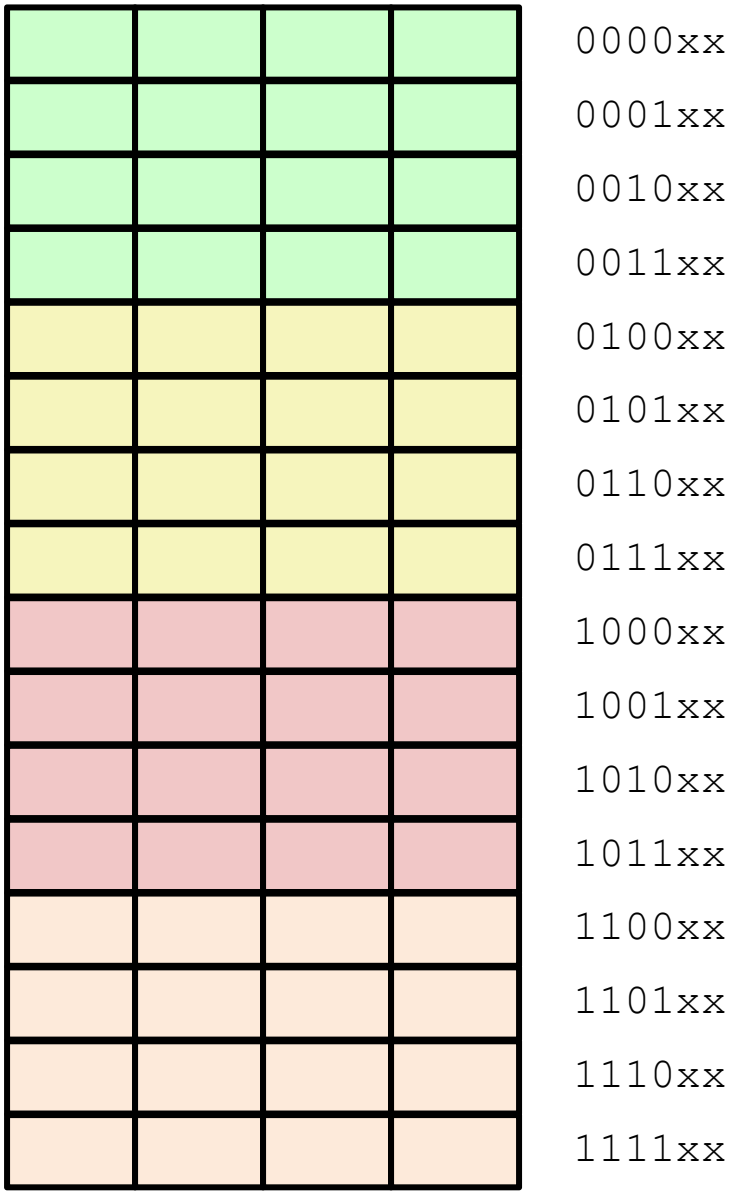
16 byte, direct-mapped cache



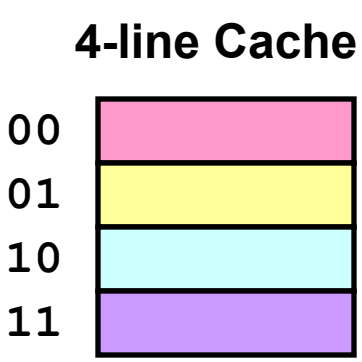
High Bit Indexing

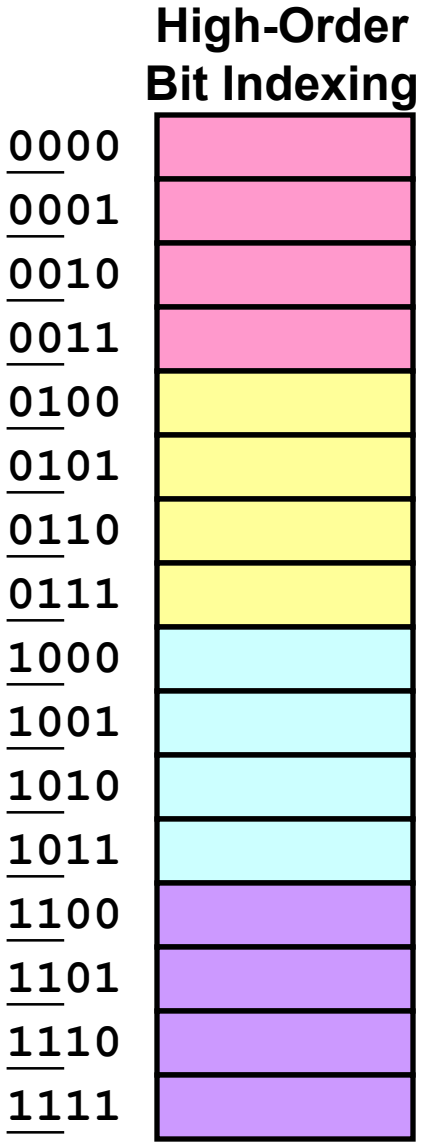
- **Addresses of form SSTTBB**
  - **SS** Set index bits
  - **TT** Tag bits
  - **BB** Offset bits
- **Program with high spatial locality would generate lots of conflicts**

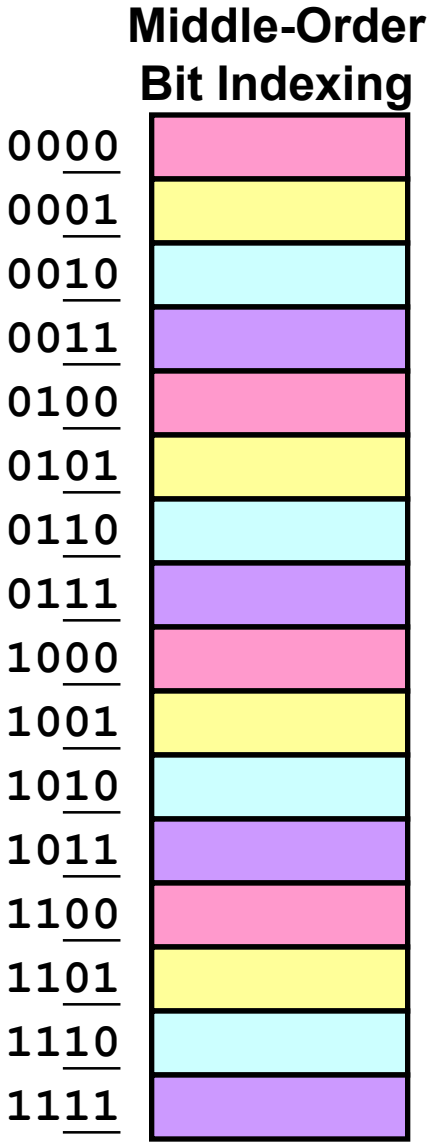




Why Use Middle Bits as Index?







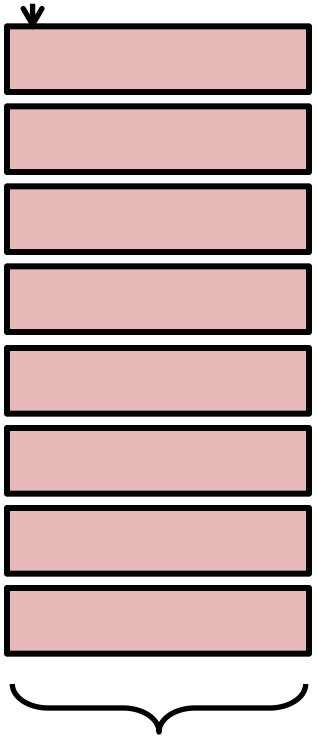
- **以地址高位部分(High-Order Bit)索引**
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality
- **以地址较低位部分(Middle-Order Bit)索引**
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time

A Higher Level Example

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

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



64 B = 8 doubles

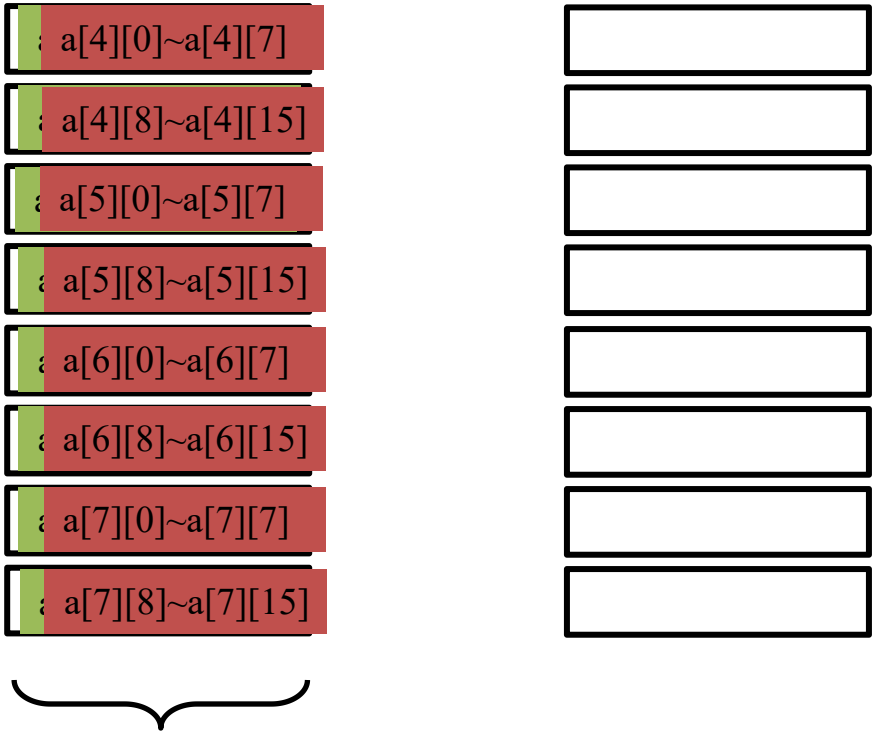
blackboard

Ignore the variables sum, i, j

BlackBoard

a[0][0]~a[0][7]	**T..T000BBB000**
a[0][8]~a[0][15]	T..T001BBB000
a[1][0]~a[1][7]	T..T010BBB000
a[1][8]~a[1][15]	T..T011BBB000
a[2][0]~a[2][7]	T..T100BBB000
a[2][8]~a[2][15]	T..T101BBB000
a[3][0]~a[3][7]	T..T110BBB000
a[3][8]~a[3][15]	T..T111BBB000
a[4][0]~a[4][7]	T..T000BBB000
a[4][8]~a[4][15]	T..T001BBB000
a[5][0]~a[5][7]	T..T010BBB000
a[5][8]~a[5][15]	T..T011BBB000
a[6][0]~a[6][7]	T..T100BBB000
a[6][8]~a[6][15]	T..T101BBB000
a[7][0]~a[7][7]	T..T110BBB000
a[7][7]~a[7][15]	T..T111BBB000

```
int sum_array_rows(double a[16][16])
```



Cache容量： 8 blocks

Block： 64 B = 8 doubles

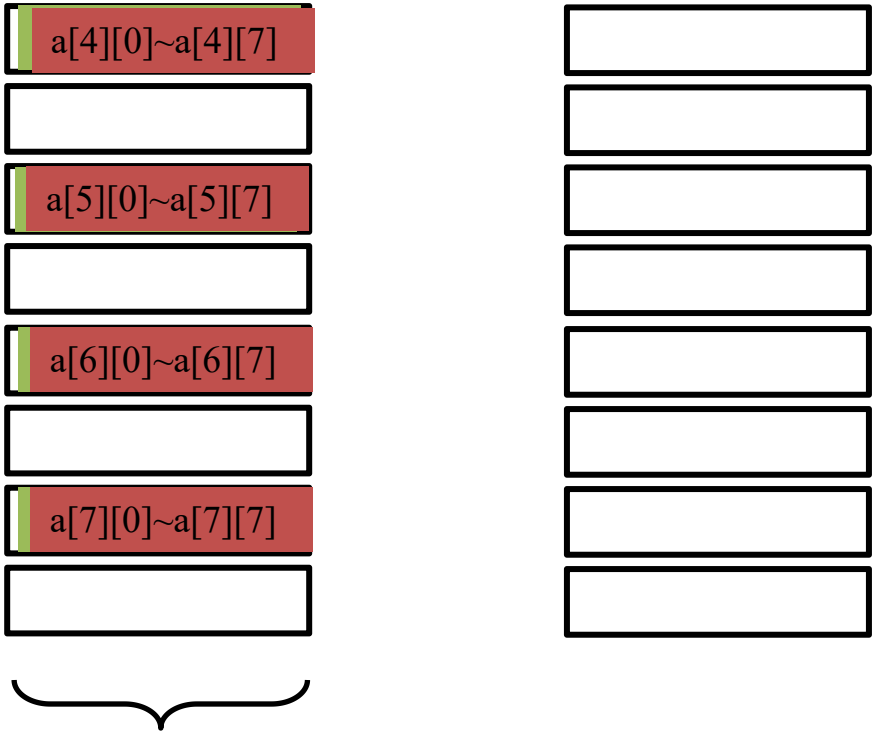
a[i][j] in Memory | T...T | SSS | BBB000 |

Addresses of form T...TSSSBBB000

BlackBoard

a[0][0]~a[0][7]	**T..T000BBB000**
a[0][8]~a[0][15]	T..T001BBB000
a[1][0]~a[1][7]	T..T010BBB000
a[1][8]~a[1][15]	T..T011BBB000
a[2][0]~a[2][7]	T..T100BBB000
a[2][8]~a[2][15]	T..T101BBB000
a[3][0]~a[3][7]	T..T110BBB000
a[3][8]~a[3][15]	T..T111BBB000
a[4][0]~a[4][7]	T..T000BBB000
a[4][8]~a[4][15]	T..T001BBB000
a[5][0]~a[5][7]	T..T010BBB000
a[5][8]~a[5][15]	T..T011BBB000
a[6][0]~a[6][7]	T..T100BBB000
a[6][8]~a[6][15]	T..T101BBB000
a[7][0]~a[7][7]	T..T110BBB000
a[7][7]~a[7][15]	T..T111BBB000

```
int sum_array_cols(double a[16][16])
```



Cache容量： 8 blocks

Block： 64 B = 8 doubles

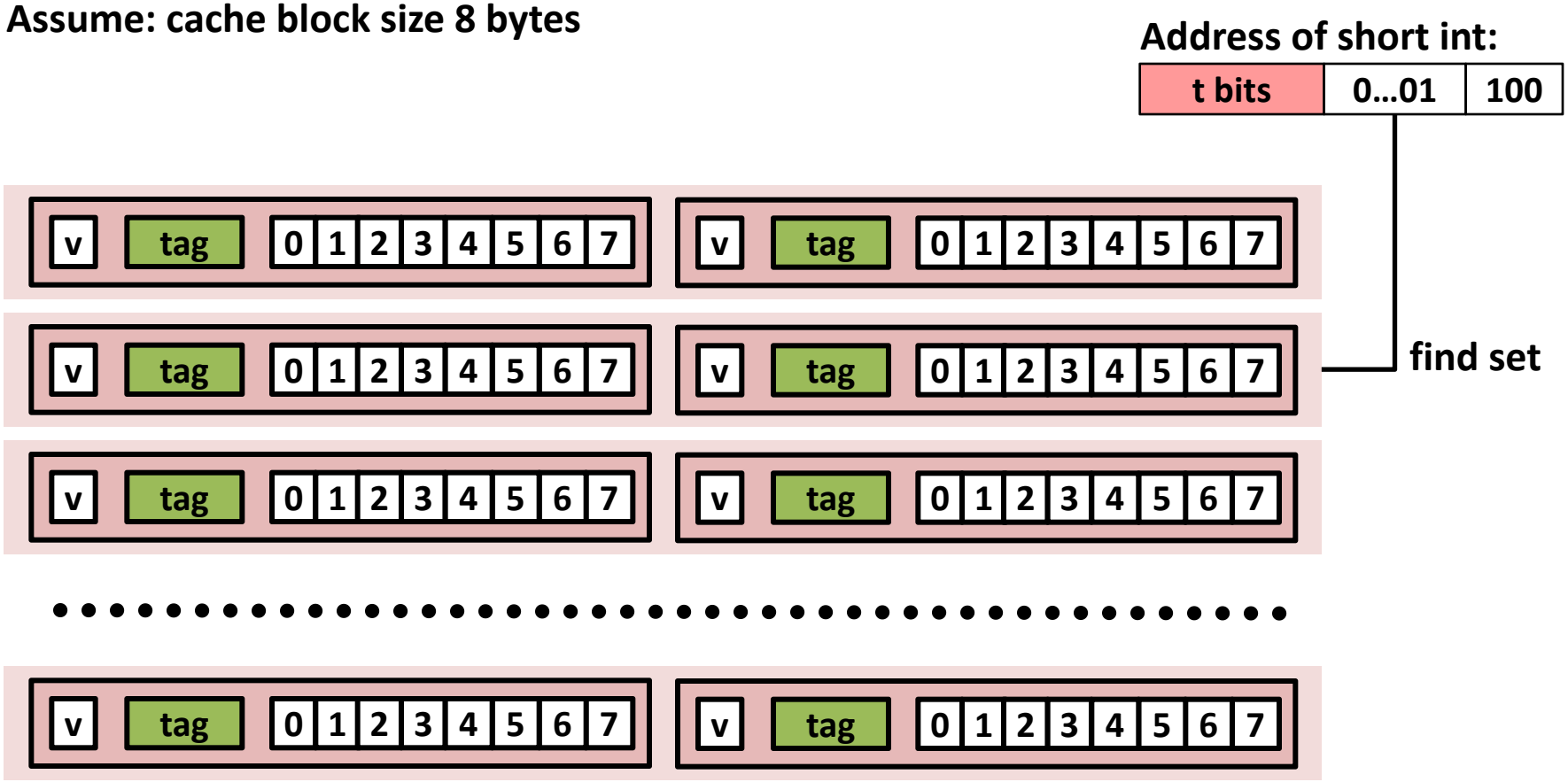
a[i][j] in Memory | T...T | SSS | BBB000 |

Addresses of form T...TSSSBBB000

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

E = 2: Two lines per set

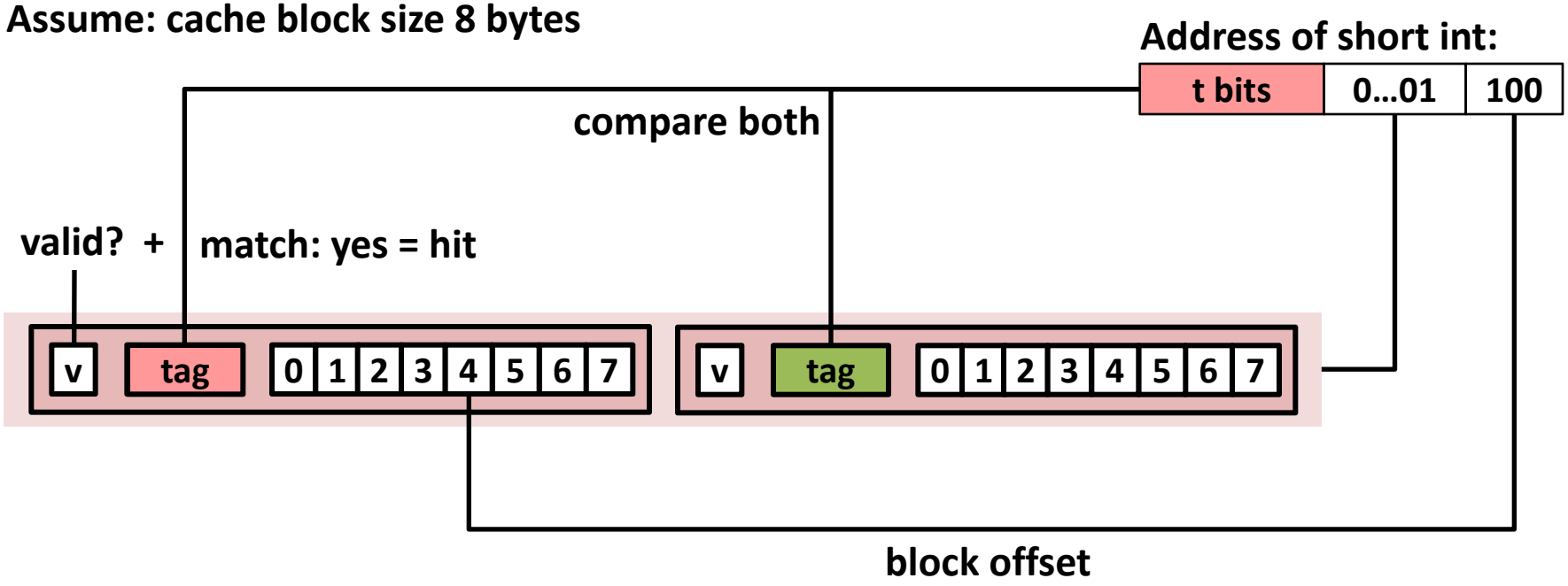
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

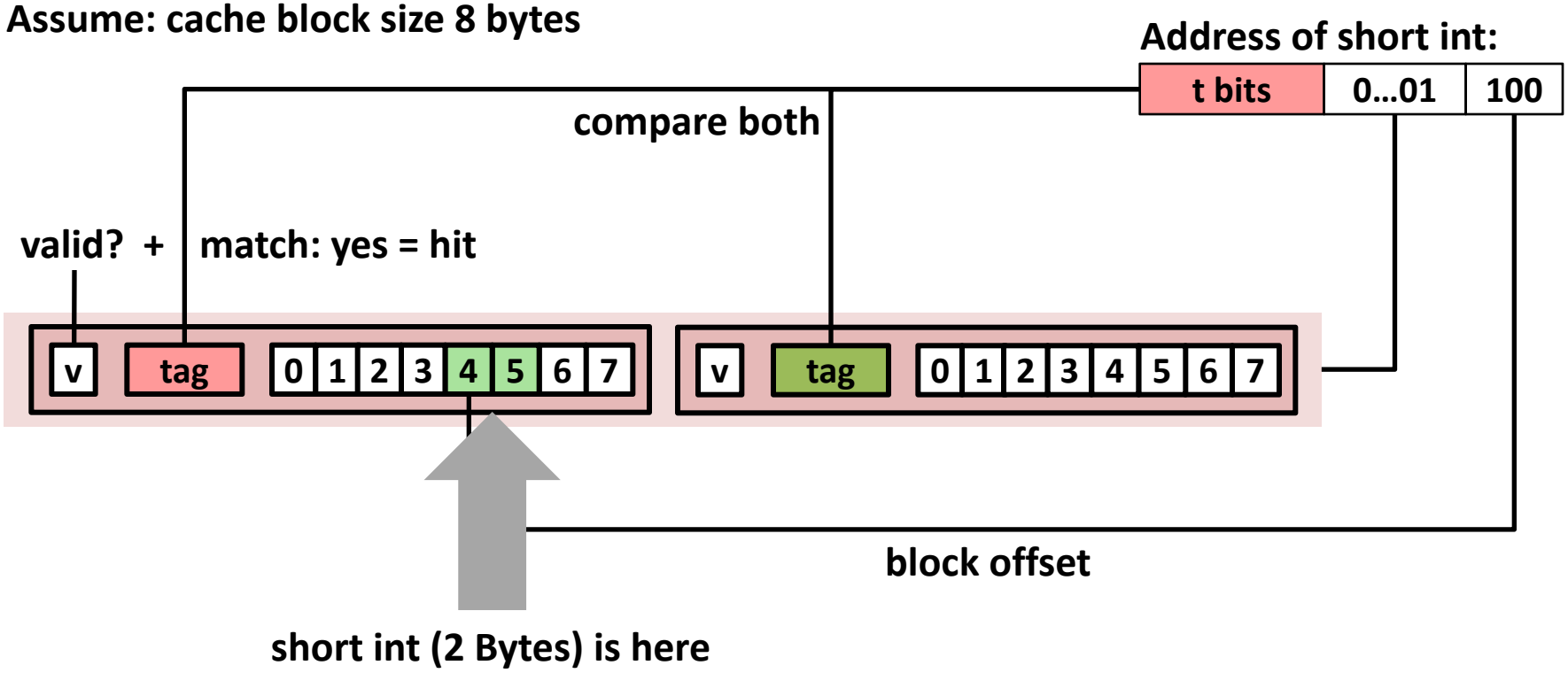
Assume: cache block size 8 bytes



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):

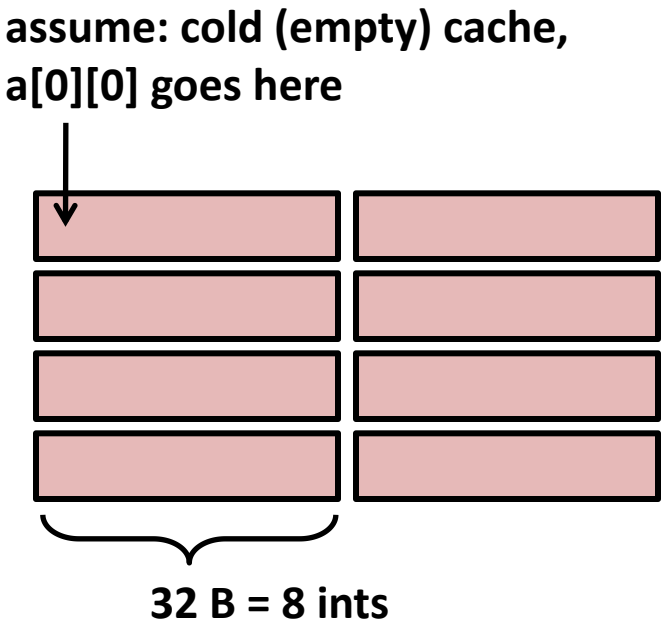
0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	hit

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

A Higher Level Example

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

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

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

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



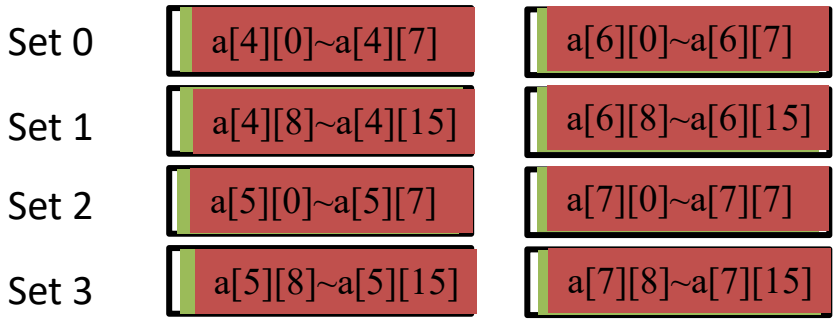
Ignore the variables sum, i, j

blackboard

BlackBoard

a[0][0]~a[0][7]	**T..T00BBB00**
a[0][8]~a[0][15]	T..T01BBB00
a[1][0]~a[1][7]	T..T10BBB00
a[1][8]~a[1][15]	T..T11BBB00
a[2][0]~a[2][7]	T..T00BBB00
a[2][8]~a[2][15]	T..T01BBB00
a[3][0]~a[3][7]	T..T10BBB00
a[3][8]~a[3][15]	T..T11BBB00
a[4][0]~a[4][7]	T..T00BBB00
a[4][8]~a[4][15]	T..T01BBB00
a[5][0]~a[5][7]	T..T10BBB00
a[5][8]~a[5][15]	T..T11BBB00
a[6][0]~a[6][7]	T..T00BBB00
a[6][8]~a[6][15]	T..T01BBB00
a[7][0]~a[7][7]	T..T10BBB00
a[7][7]~a[7][15]	T..T11BBB00

```
sum_array_rows(int a[16][16])
```



Cache容量：8 blocks
Block：32 B = 8 ints
2路组相联

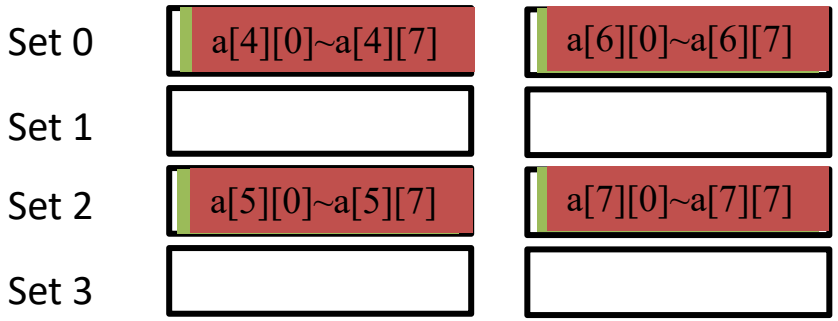
a[i][j] in Memory

Addresses of form T...TSSBBB00
与E=1相比，Set位少1位，Tag位多1位

BlackBoard

a[0][0]~a[0][7]	**T..T00BBB00**
a[0][8]~a[0][15]	T..T01BBB00
a[1][0]~a[1][7]	T..T10BBB00
a[1][8]~a[1][15]	T..T11BBB00
a[2][0]~a[2][7]	T..T00BBB00
a[2][8]~a[2][15]	T..T01BBB00
a[3][0]~a[3][7]	T..T10BBB00
a[3][8]~a[3][15]	T..T11BBB00
a[4][0]~a[4][7]	T..T00BBB00
a[4][8]~a[4][15]	T..T01BBB00
a[5][0]~a[5][7]	T..T10BBB00
a[5][8]~a[5][15]	T..T11BBB00
a[6][0]~a[6][7]	T..T00BBB00
a[6][8]~a[6][15]	T..T01BBB00
a[7][0]~a[7][7]	T..T10BBB00
a[7][7]~a[7][15]	T..T11BBB00

```
int sum_array_cols(int a[16][16])
```



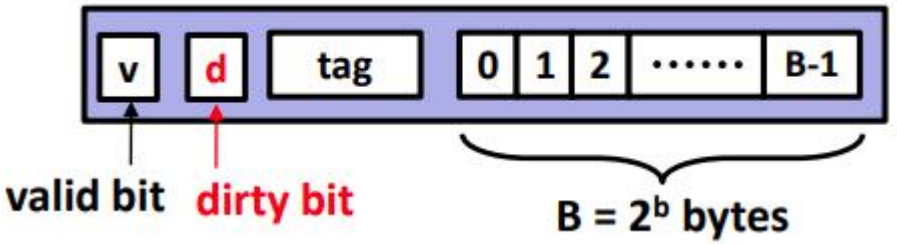
Cache容量：8 blocks
Block：32 B = 8 ints
2路组相联

a[i][j] in Memory

Addresses of form T...TSSSBBB00
与E=1相比，Set位少1位，Tag位多1位

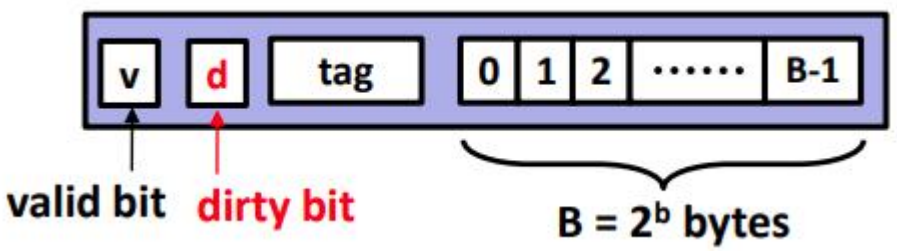
What about writes?

- **写操作时数据存在多个副本**
  - L1, L2, Main Memory, Disk
- **写命中时的操作策略**
  - 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)
- **写失效时的操作策略**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- **一般规则**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate



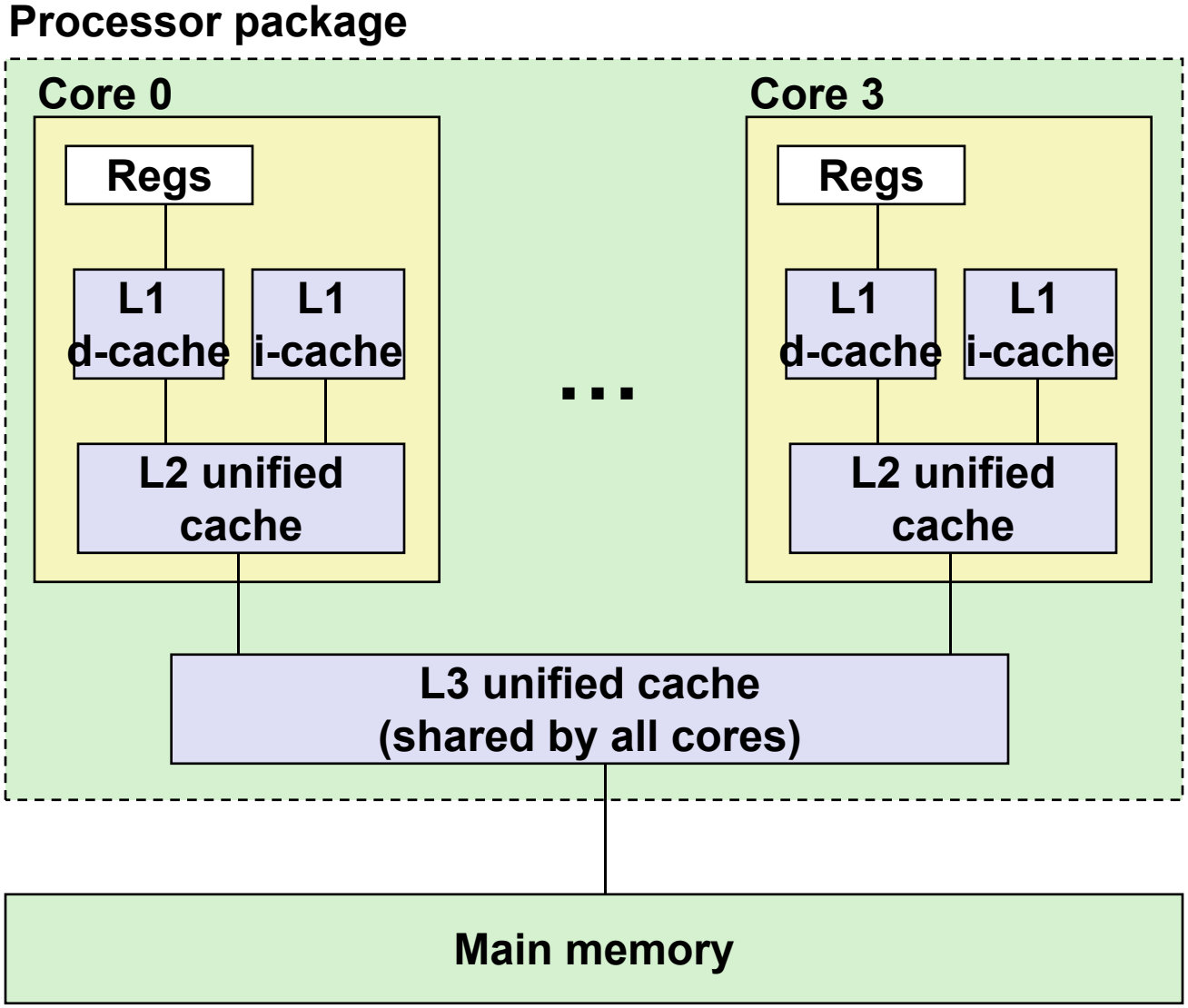
Practical Write-back Write-allocate

对地址X进行写操作时



- **如果在Cache中命中(hit)**
  - Update the contents of block
  - Set dirty bit to 1 (bit is sticky and only cleared on eviction)
- **如果在Cache中未命中(miss)**
  - Fetch block from memory (per a read miss)
  - The perform the write operations (per a write hit)
- **当Cache块(Cache line)被换出，且其dirty位为1**
  - The entire block of 2^b bytes are written back to memory
  - Dirty bit is cleared (set to 0)
  - Line is replaced by new contents

Intel Core i7 Cache Hierarchy



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

- **失效率(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.
- **命中时间(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
- **失效开销(Miss Penalty)**
  - Additional time required because of a miss
    - **typically 50-200 cycles for main memory (Trend: increasing!)**
- **平均访存时间（Average Memory Access Time**
  - **AMAT = Hit Time + MissRate × Miss Penalty**

Lets think about those numbers

- **存储器访问时，Cache命中的性能远远高于失效的性能**
  - Could be 100x, if just L1 and main memory
- **存储器访问时99% 命中率的性能 vs. 97%命中率的性能？**
  - 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

- **Tips1：让最常执行的代码执行性能更高些**
  - Focus on the inner loops of the core functions
- **Tips2：尽量时内循环的存储器访问失效率最小**
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: 我们对局部性的定性概念是通过对Cache的理解得到的.

Cache Memories

- Cache memory 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: 测量的读吞吐量作为空间和时间局部性的函数.**
  - Compact way to characterize memory system performance.

The Memory Mountain

```
long data[MAXELEMS]; /* Global array to traverse */
/* test - Iterate over first "elems" elements of
* array "data" with stride of "stride",
* 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));
}
```

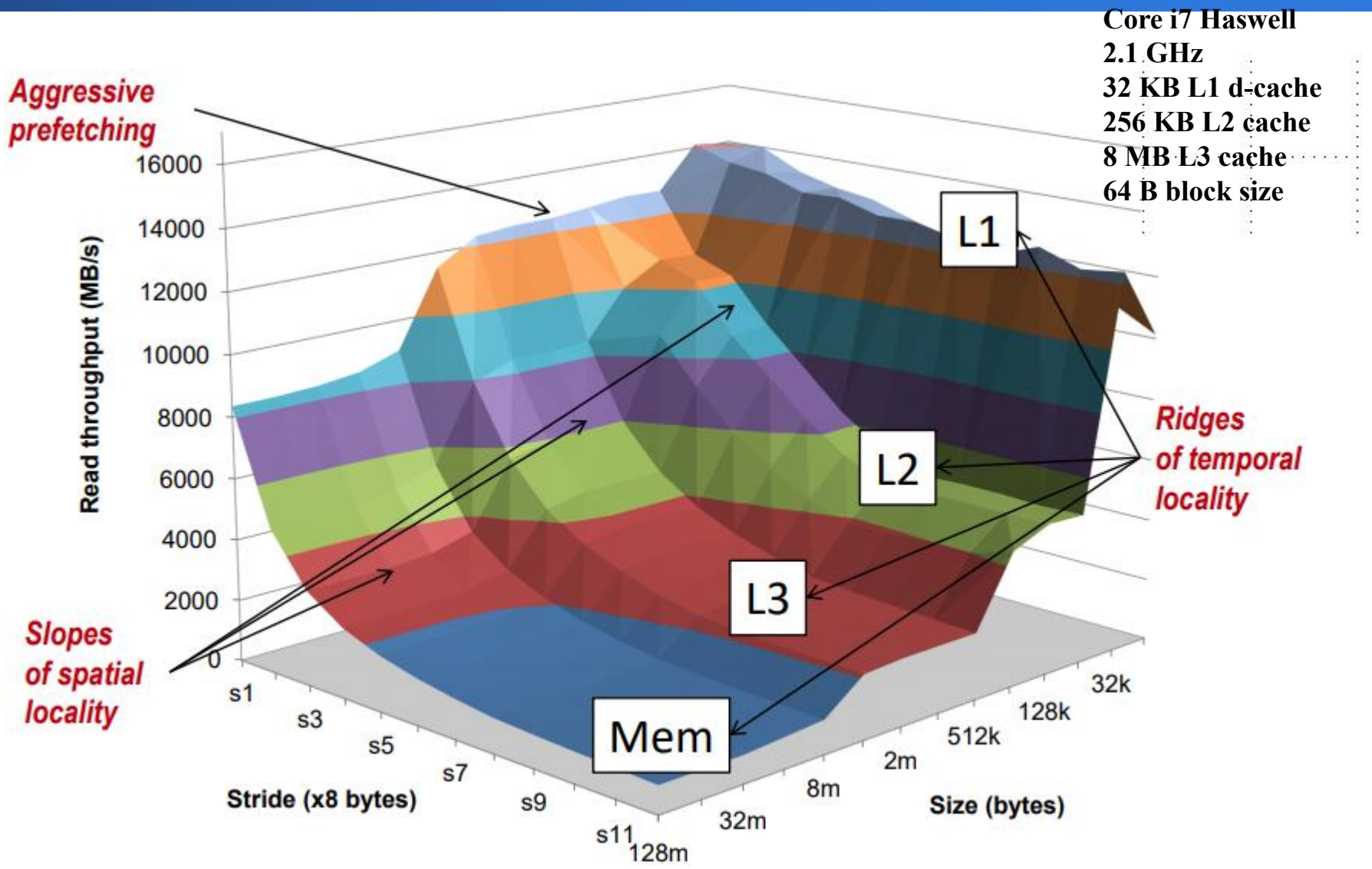
Call test() with many combinations of elems and stride.

For each elems and stride:

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

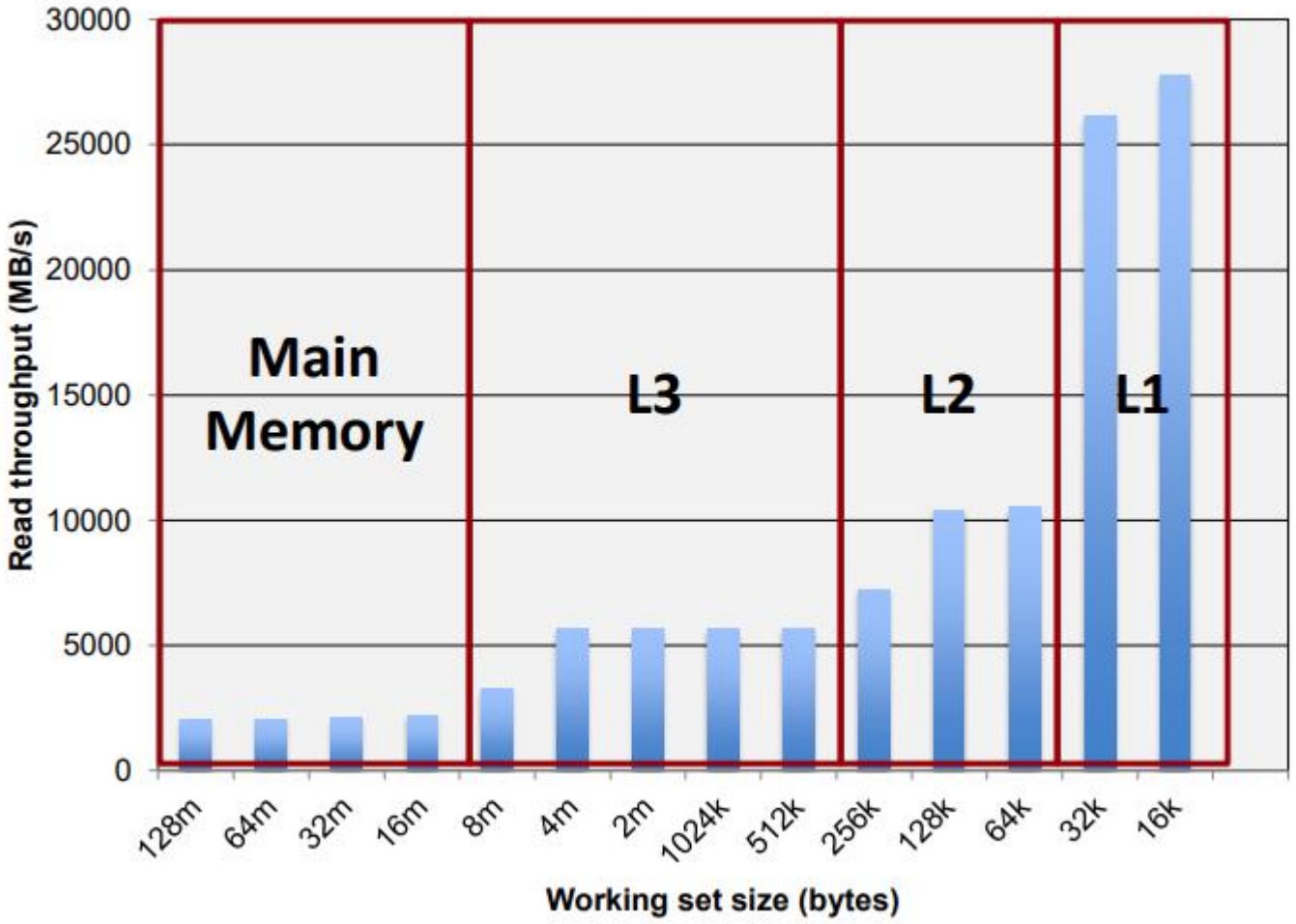
2. Call test() again and measure the read throughput(MB/s)

The Memory Mountain



Stride 对应test()函数中的Stride；Size：对应test()函数中的elems

Cache Capacity Effects from Memory Mountain



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

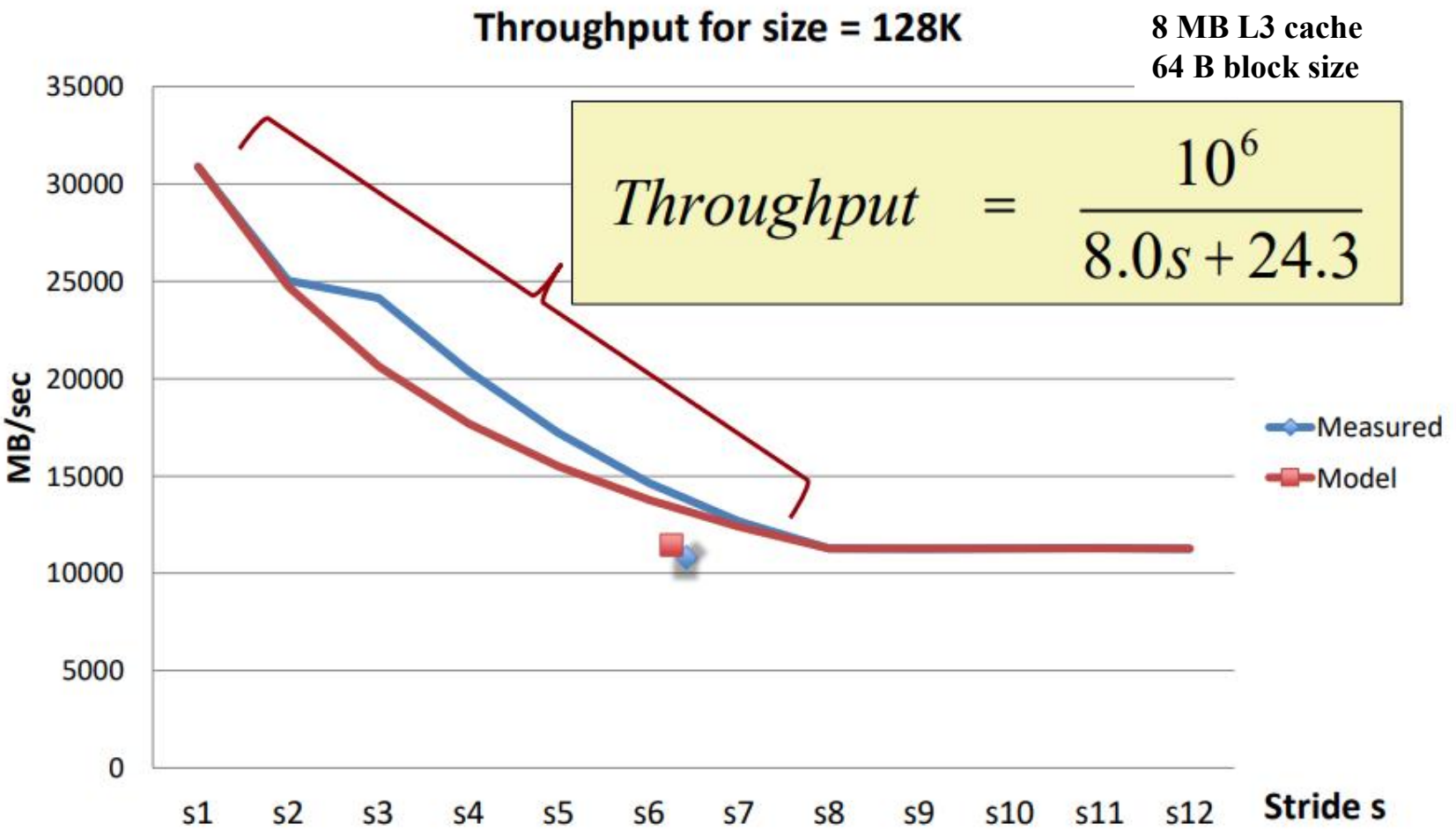
Slice through memory mountain with stride=8

Access Stride Effects from Memory Mountain

working set size 固定，访问步长对性能的影响

Modeling Stride Effects from Memory Mountain

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



Cache Memories

- Cache memory 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
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

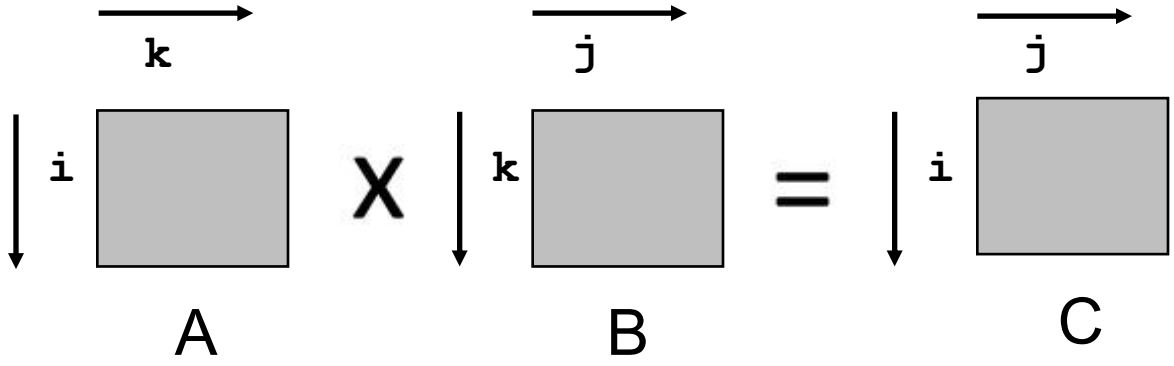
Variable sum held in register

```
/* 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;
  }
}
```

从Memory Mountain实验数据可观察到：
1、存储系统的性能是与时间和空间局部性有关的函数
2、在硬件结构确定的情况下
（1）工作集的大小 影响 程序的性能
（2）访问步长影响 程序的性能

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Line size = 32B (big enough for **four** 64-bit words)
  - 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)

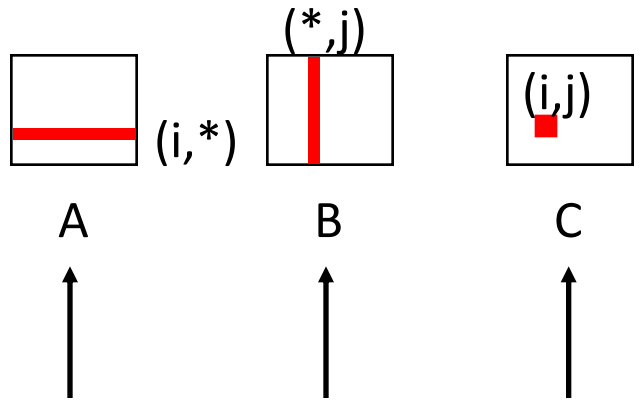
- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - int a[i][j] (4bytes)
  - accesses successive elements
  - if block size (B) > 4 bytes , exploit spatial locality
    - **compulsory miss rate = 4 bytes / B** （依次访问同行不同列数据）
- **Stepping through rows in one column:**
  - `for (i = 0; i < N; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - **compulsory miss rate = 1 (i.e. 100%)** （依次访问同列不同行数据）

Matrix Multiplication (ijk)

a[i][j],b[i][j],c[i][j] 占8个字节

```
/* 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;
  }
}
```

Inner loop:



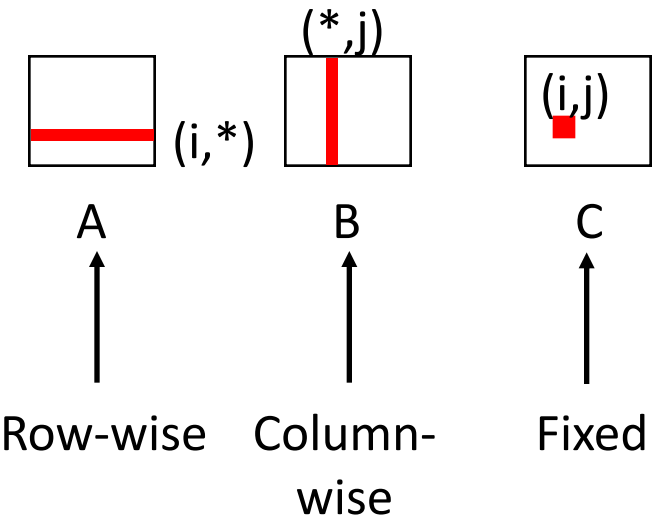
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)
Element size = 8B

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
  }
}
```



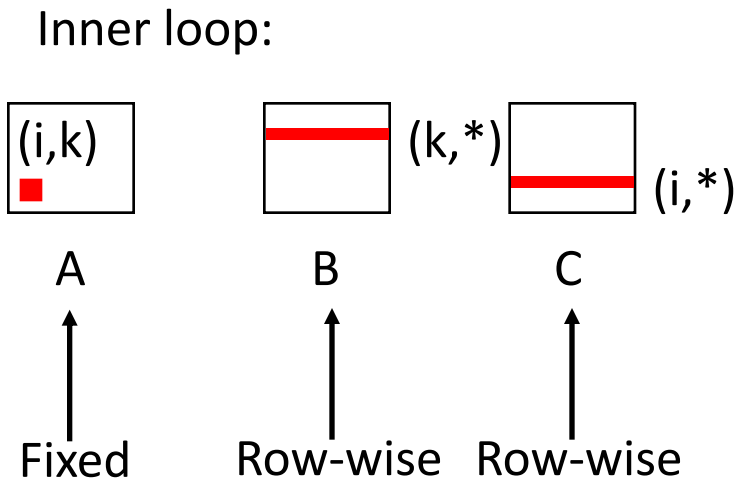
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)
Element size = 8B

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];
  }
}
```



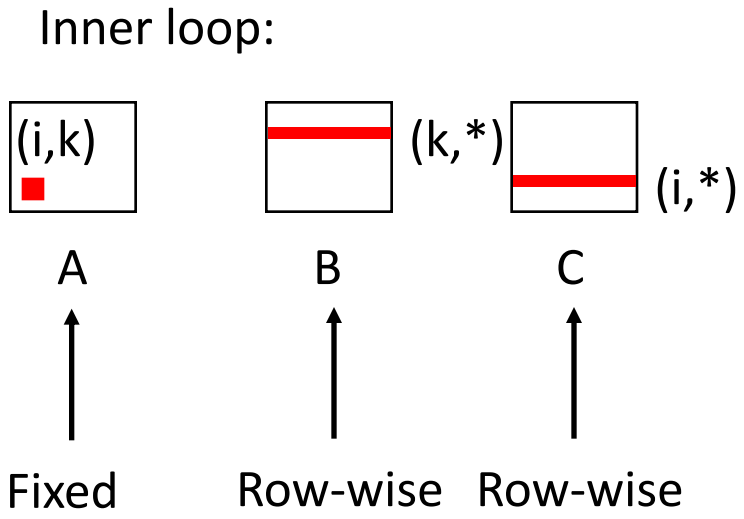
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)
Element size = 8B

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];
  }
}
```



Misses per inner loop iteration:

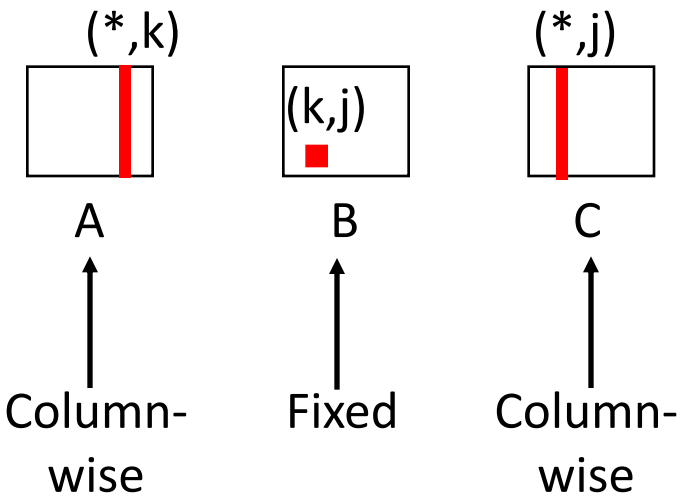
A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)
Element size = 8B

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;
  }
}
```

Inner loop:



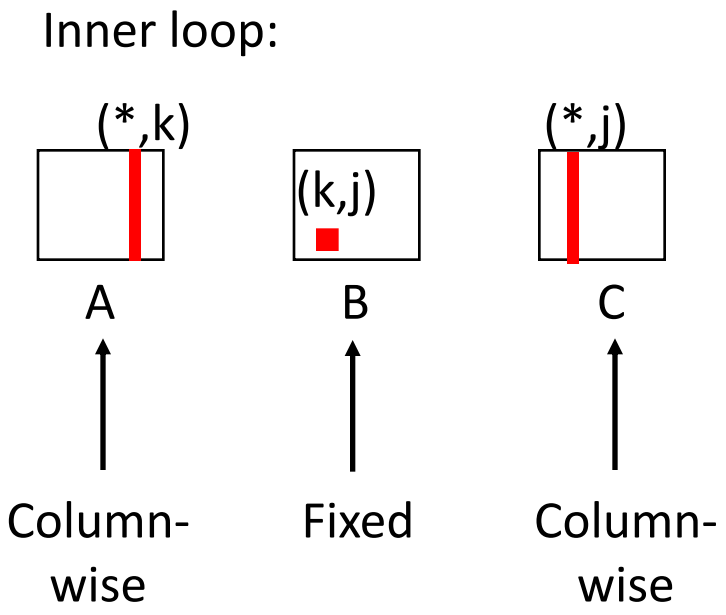
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)
Element size = 8B

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;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)
Element size = 8B

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):

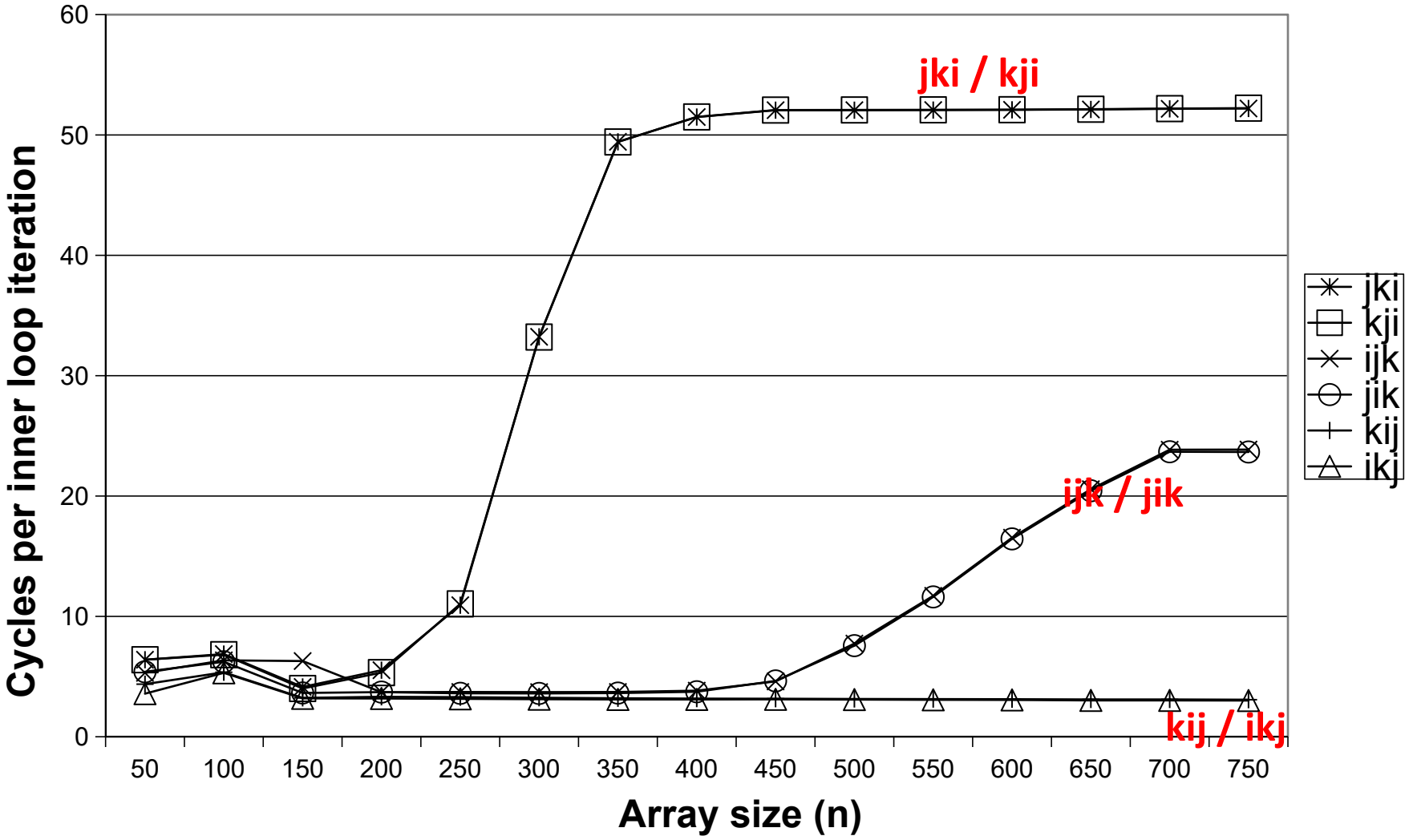
- 2 loads, 1 store
- misses/iter = **0.5**

```
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

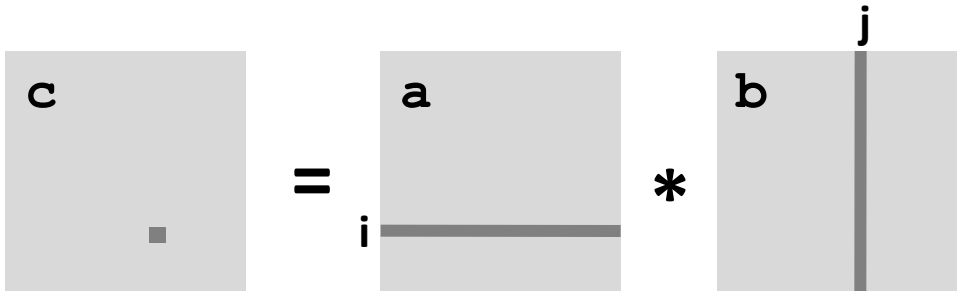


Cache Memories

- 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: Matrix Multiplication

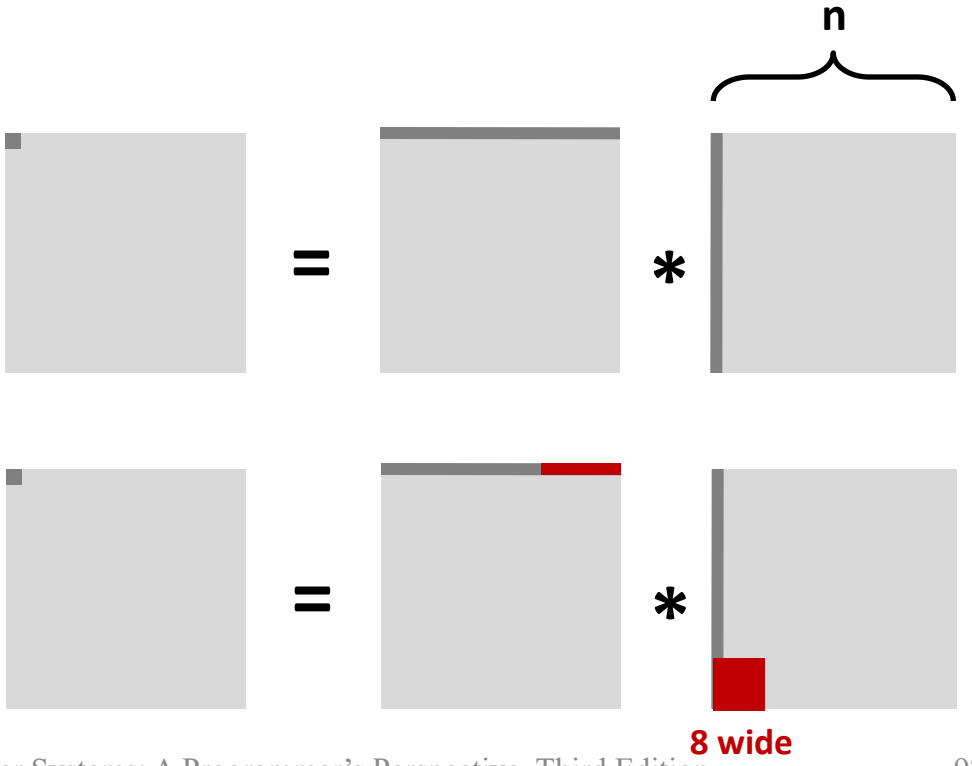
```
c = (double *) calloc(sizeof(double), n*n);

/* 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)

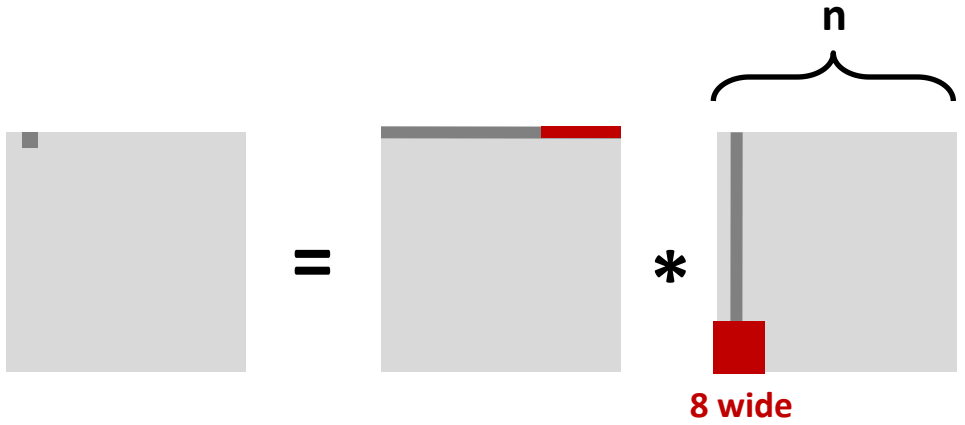
- **First iteration:**
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



Cache Miss Analysis

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

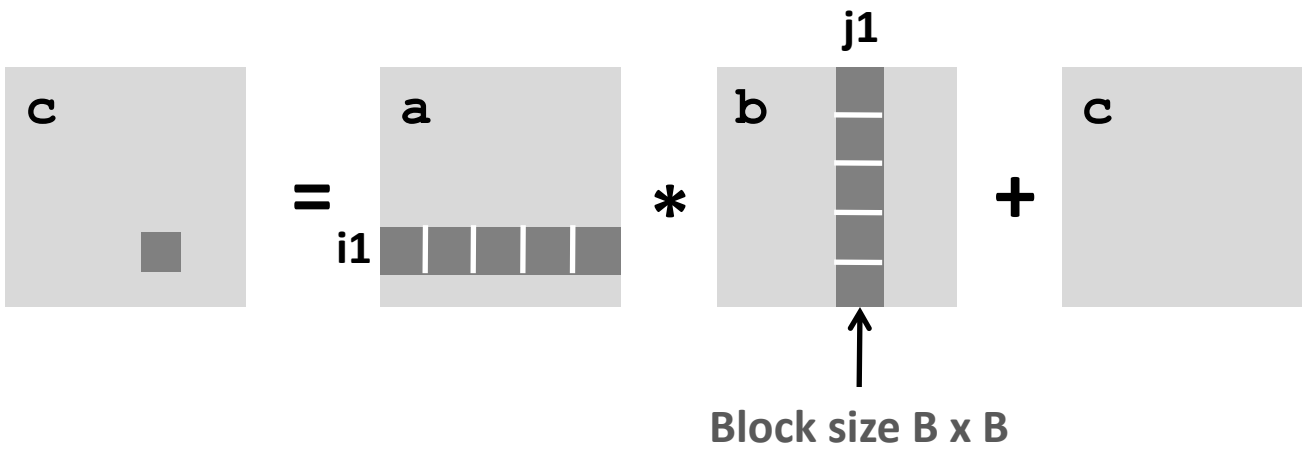
- **Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses



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

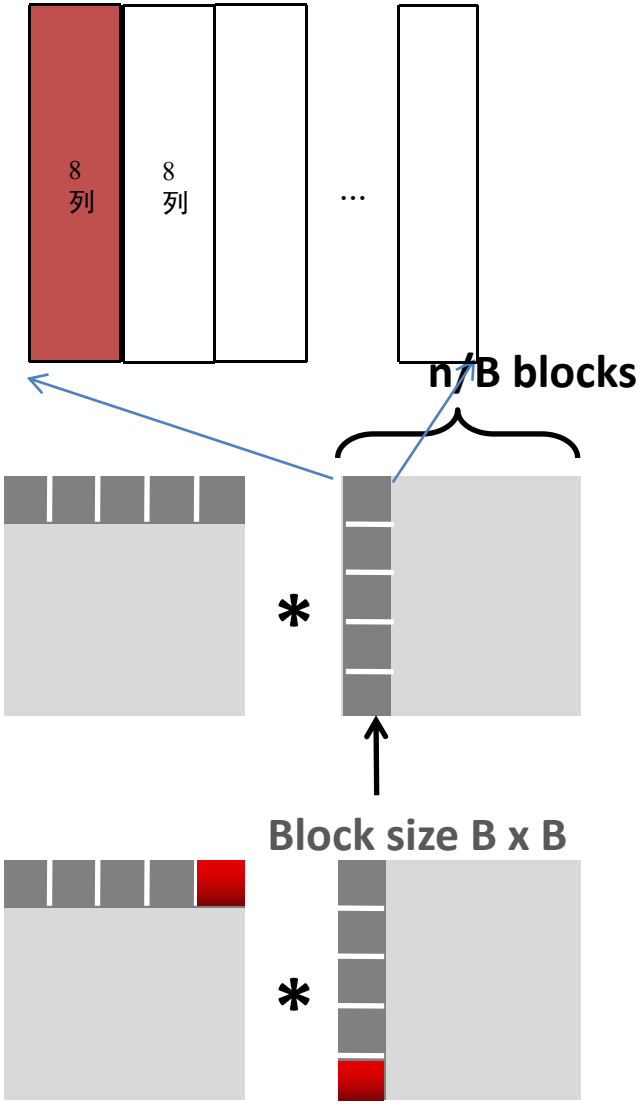
Blocked Matrix Multiplication

```
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; i1++)
                      for (j1 = j; j1 < j+B; j1++)
                          for (k1 = k; k1 < k+B; k1++)
                          c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache Miss Analysis

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

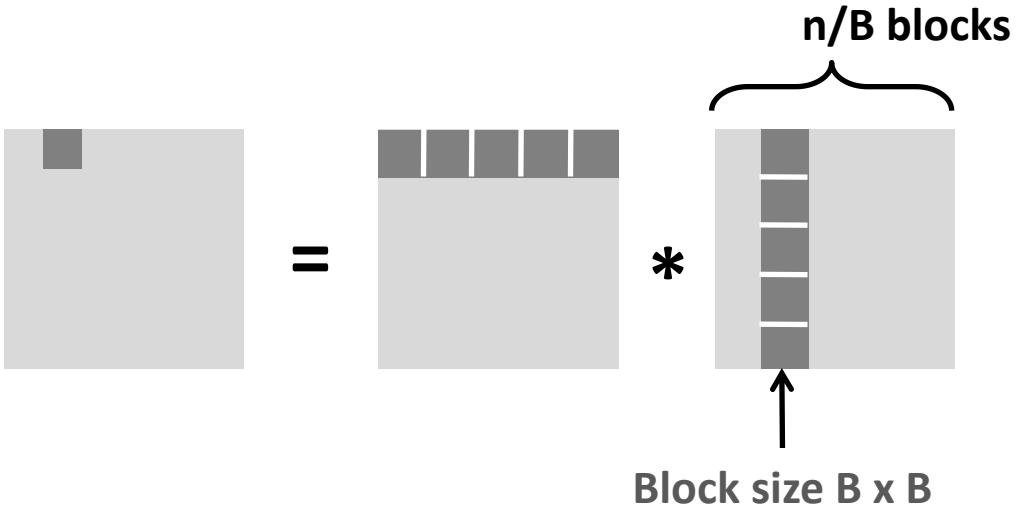


- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)

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

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

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)**

Memory Hierarchy

- **The memory abstraction**
- **RAM : main memory building block**
- **Locality of reference**
- **The memory hierarchy**
- **Cache**
- **Storage technologies and trends**

Nonvolatile Memories

- **DRAM 和 SRAM 是易失性存储器**
  - Lose information if powered off.
- **非易失性存储器在电源关闭时仍然能保存数据**
  - 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 (sector) erase capability
    - Wears out after about 100,000 erasings.
- **非易失性存储器的用途及种类**
  - 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

What's Inside A Disk Drive?

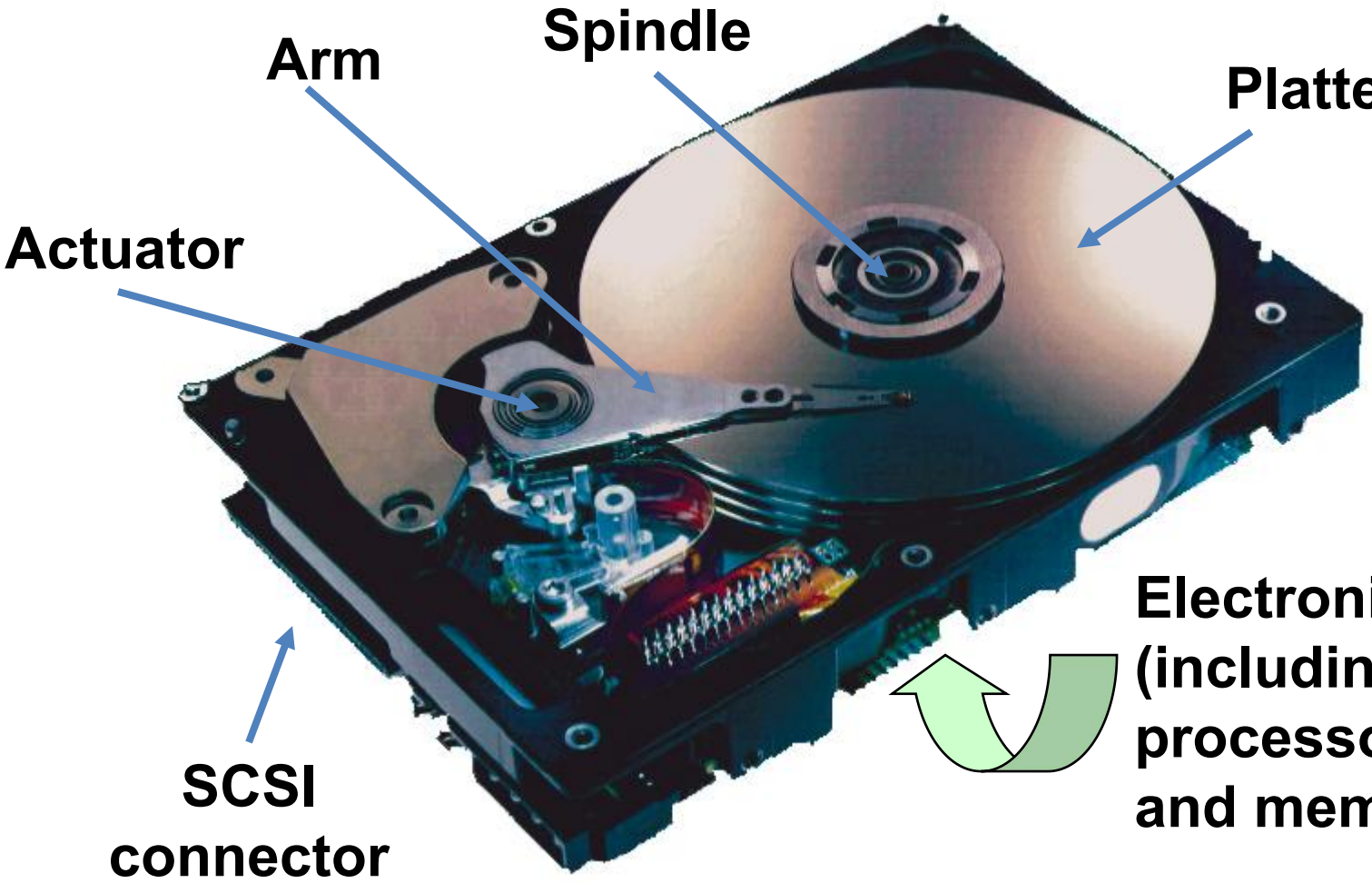
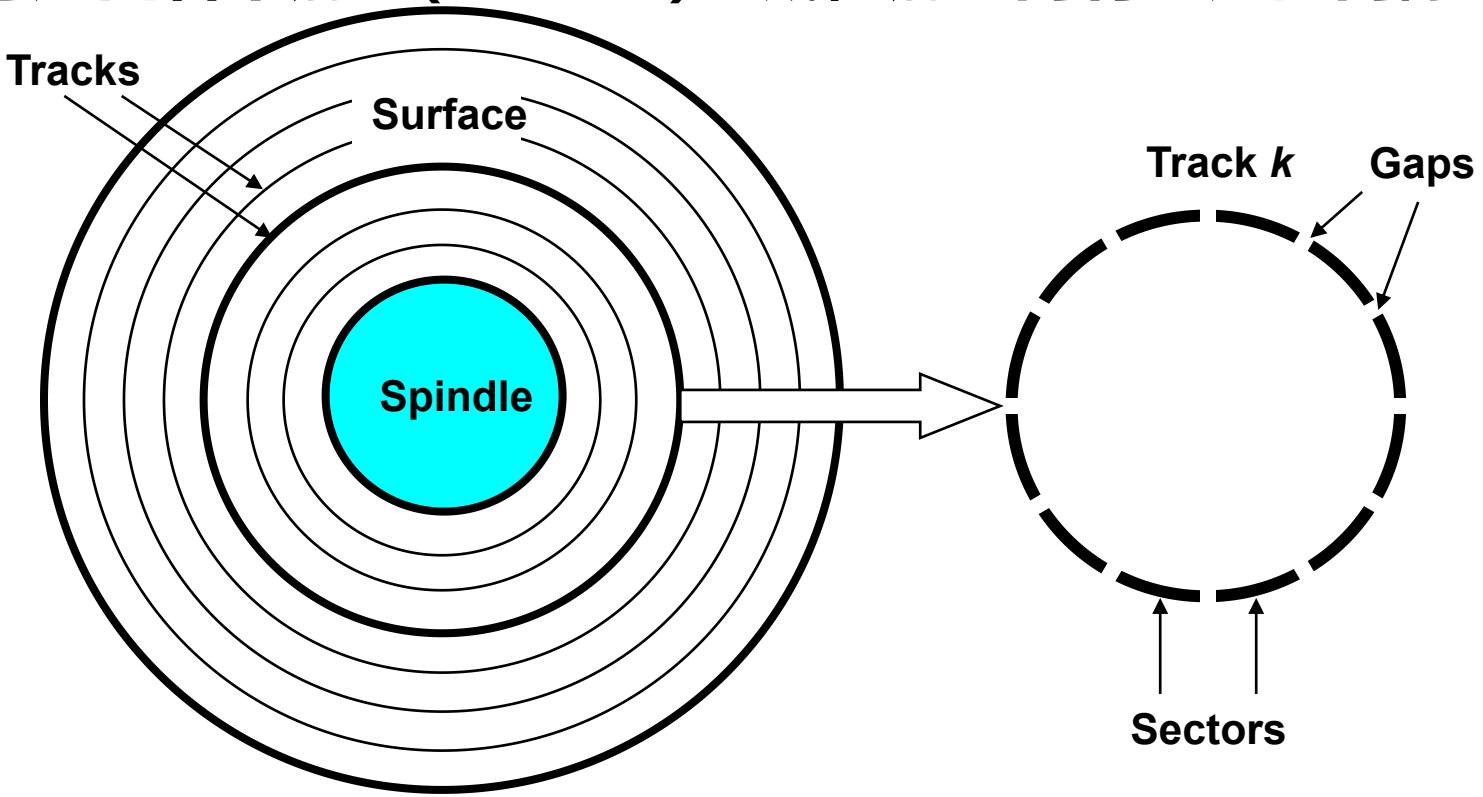


Image courtesy of Seagate Technology

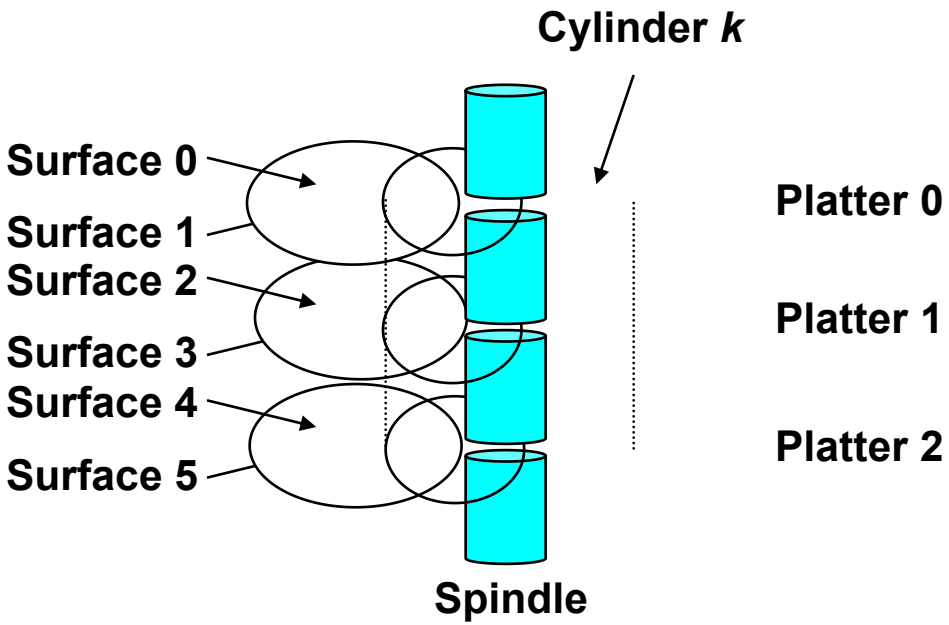
Disk Geometry

- Disks 由一组磁盘片(platters)构成，每片有两个面(surfaces).
- 每面由一组同心环组成，称为磁道(tracks).
- 每道由若干扇区(sectors)组成，扇区间有一定的间隙.



Disk Geometry (Multiple-Platter View)

- **对齐的磁道构成柱面(cylinder).**



Disk Capacity

- **Capacity: 可存储的最大位数.**
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9 Bytes (Lawsuit pending! Claims deceptive advertising).
- **影响容量的主要技术因素:**
  - Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - Areal density (bits/in^2): product of recording and track density.
- **现代磁盘将磁道划分为互不相交的区域，称为 (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

Computing Disk Capacity

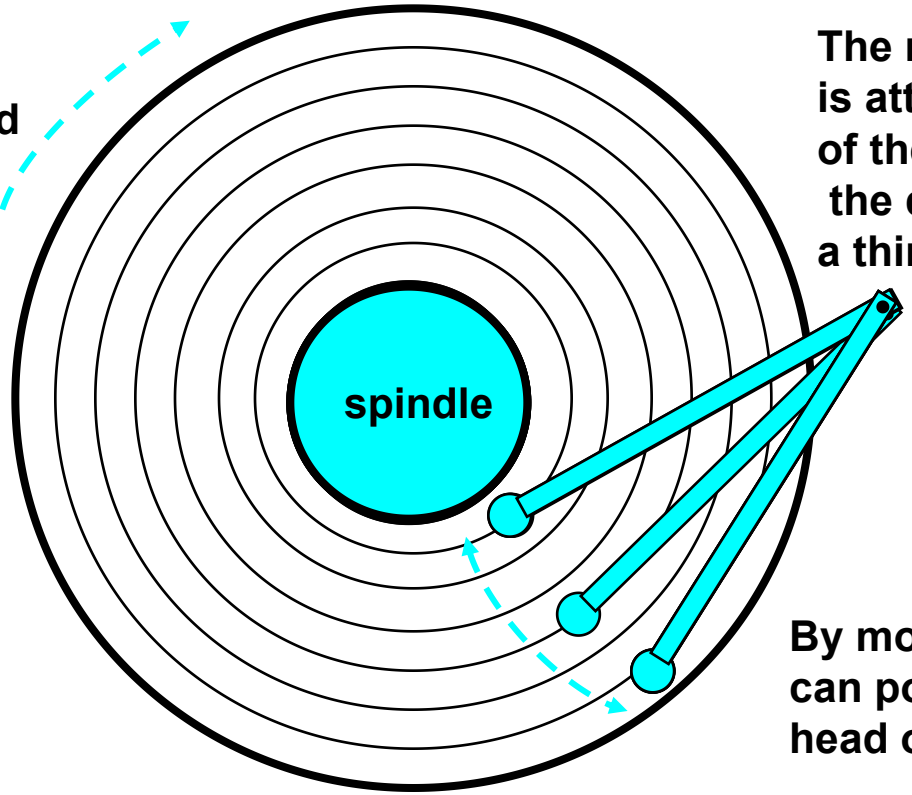
Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)

Example:

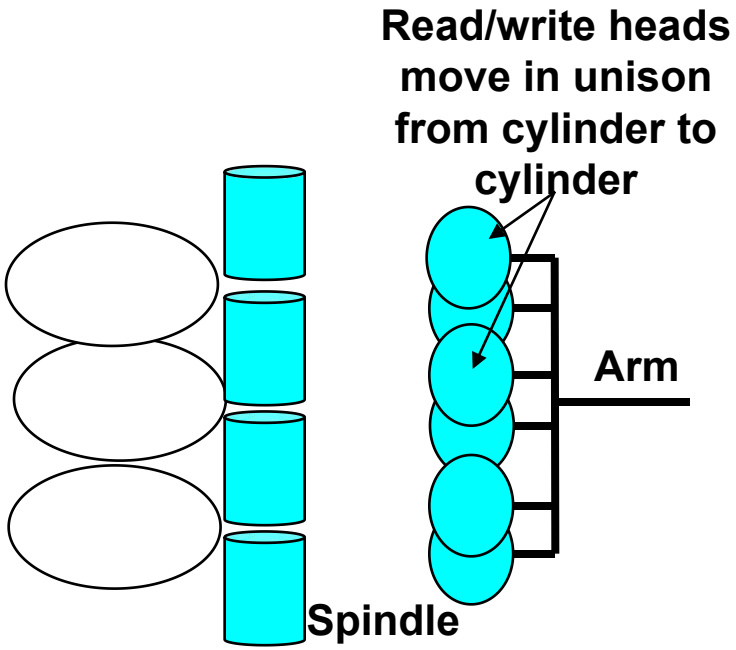
- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

Capacity = 512 x 300 x 20000 x 2 x 5
= 30,720,000,000
= 30.72 GB

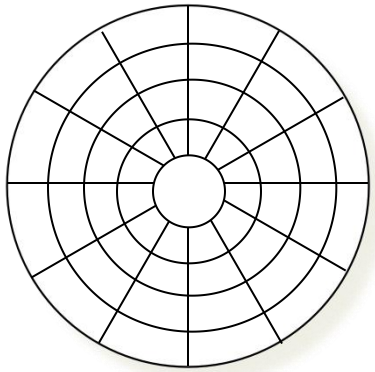
Disk Operation (Single-Platter View)



Disk Operation (Multi-Platter View)



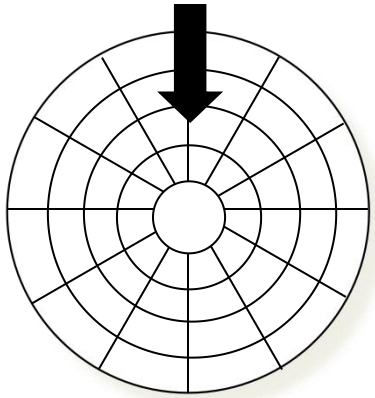
Disk Structure - top view of single platter



Surface organized into tracks

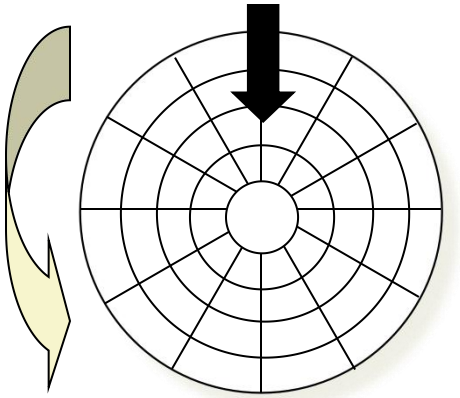
Tracks divided into sectors

Disk Access



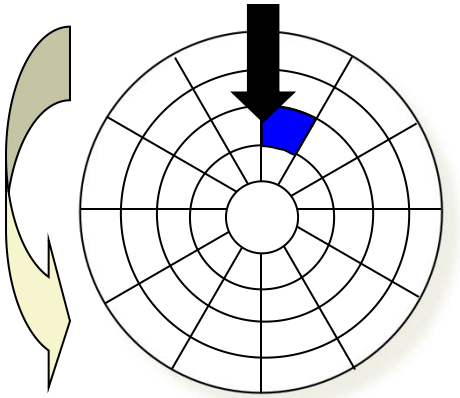
Head in position above a track

Disk Access



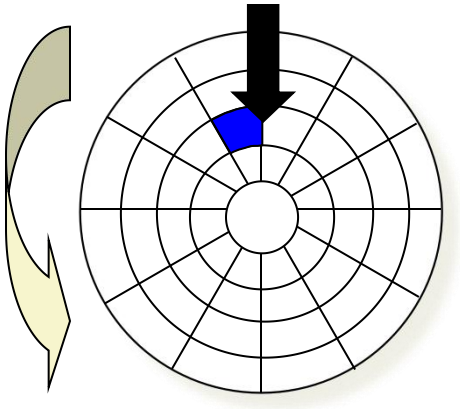
Rotation is counter-clockwise

Disk Access – Read



About to read blue sector

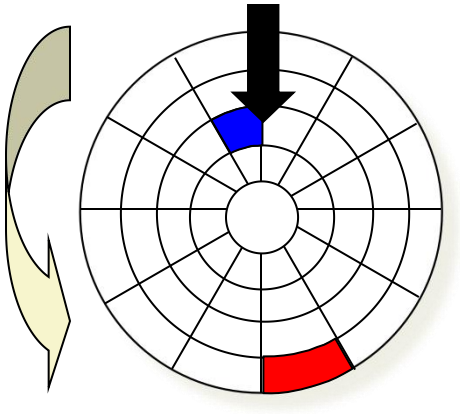
Disk Access – Read



After **BLUE** read

After reading blue sector

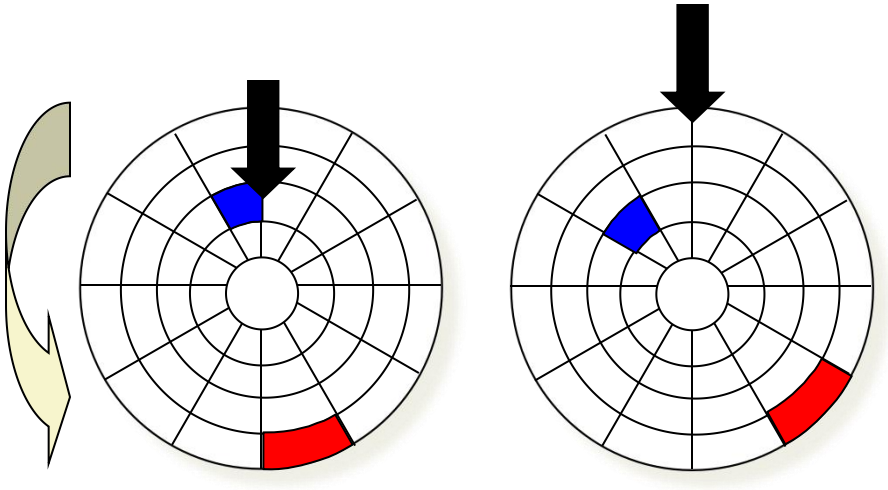
Disk Access – Read



After BLUE read

Red request scheduled next

Disk Access – Seek

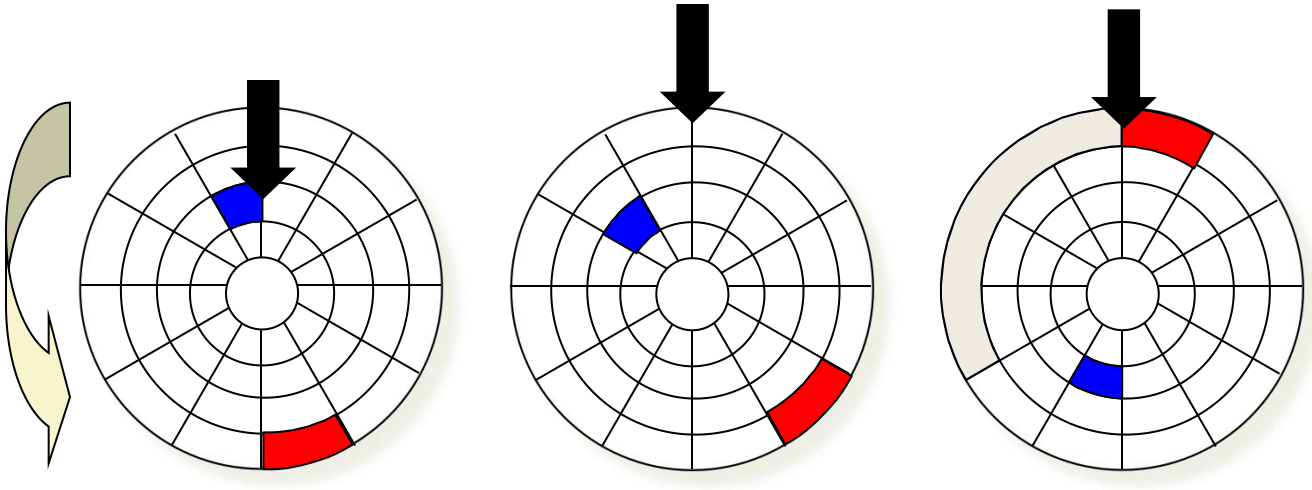


After BLUE read

Seek for RED

Seek to red's track

Disk Access – Rotational Latency



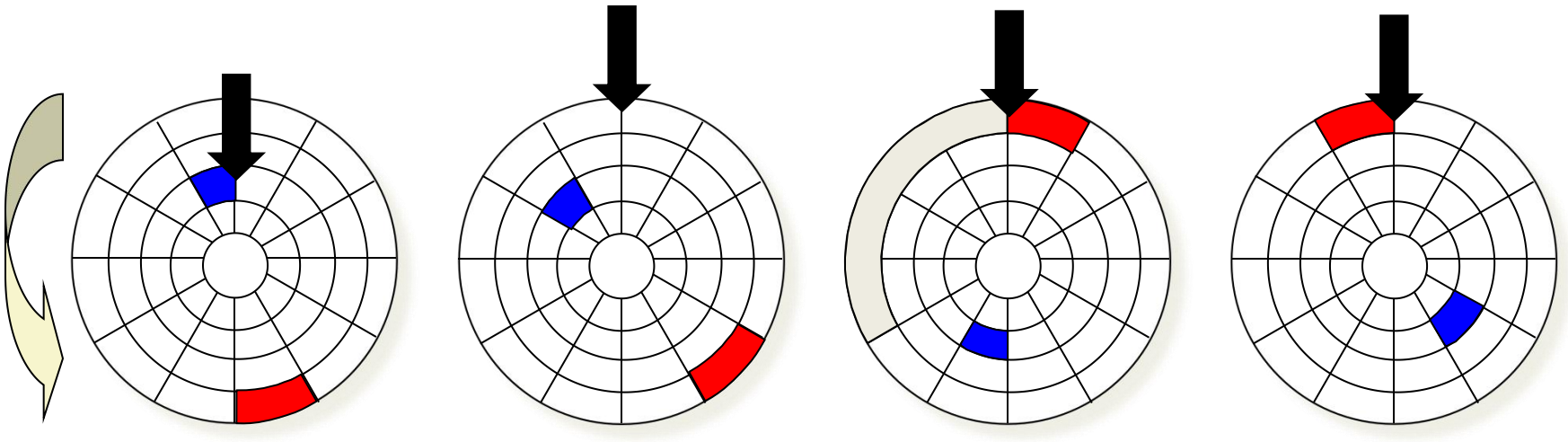
After BLUE read

Seek for RED

Rotational latency

Wait for red sector to rotate around

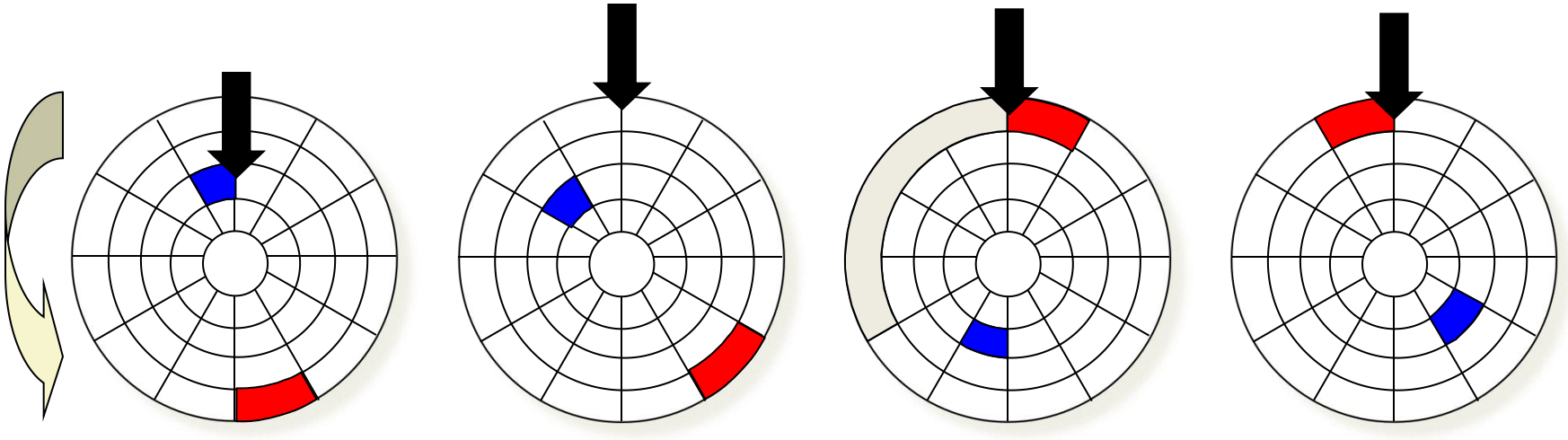
Disk Access – Read



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

Complete read of red

Disk Access – Service Time Components



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 × 1/RPMs × 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 × 1/(avg # sectors/track) × 60 secs/1 min.

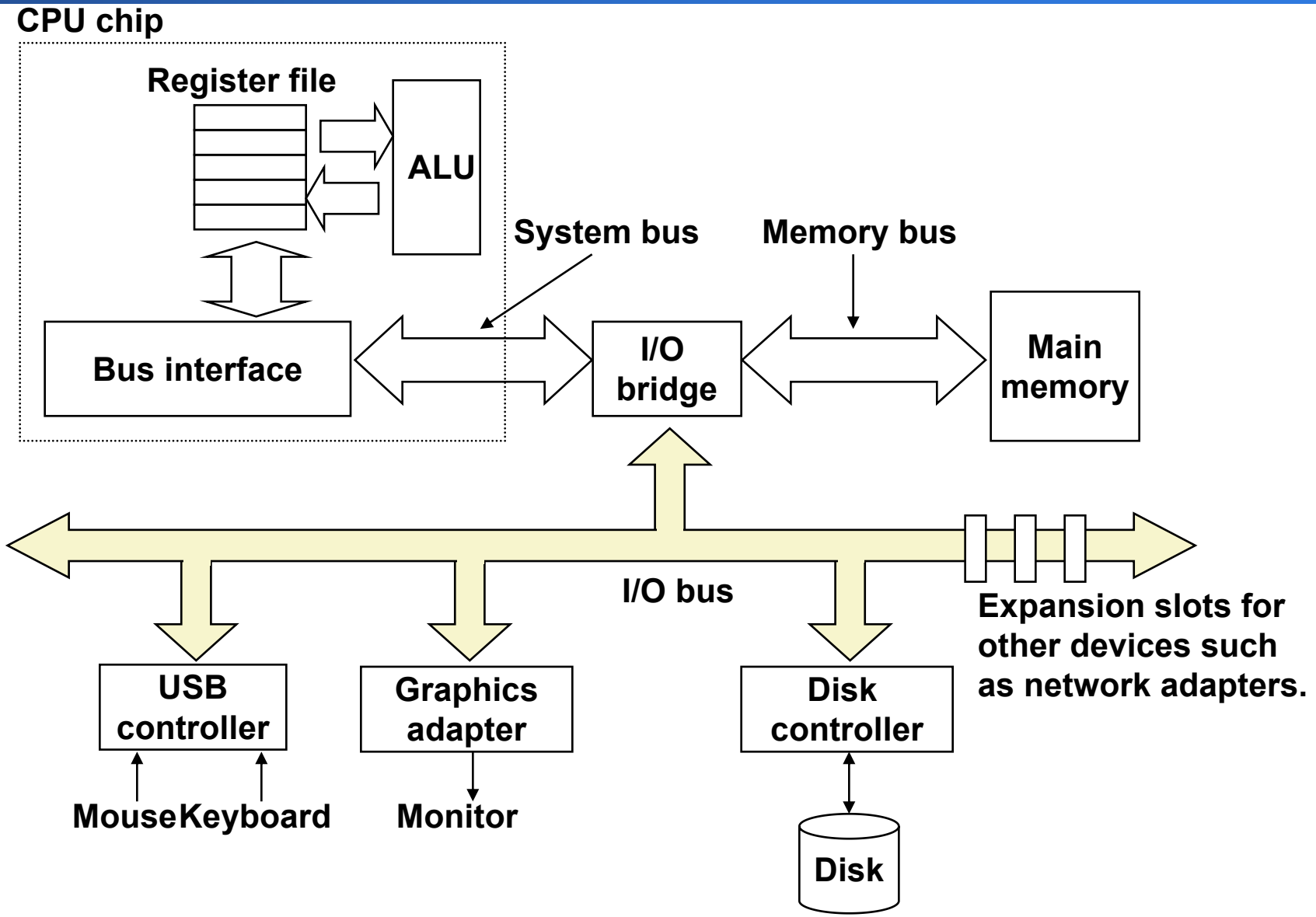
Disk Access Time Example

- **Given:**
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- **Derived:**
  - Tavg rotation = 1/2 × (60 secs/7200 RPM) × 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM× 1/400 secs/track × 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- **Important points:**
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower then DRAM.

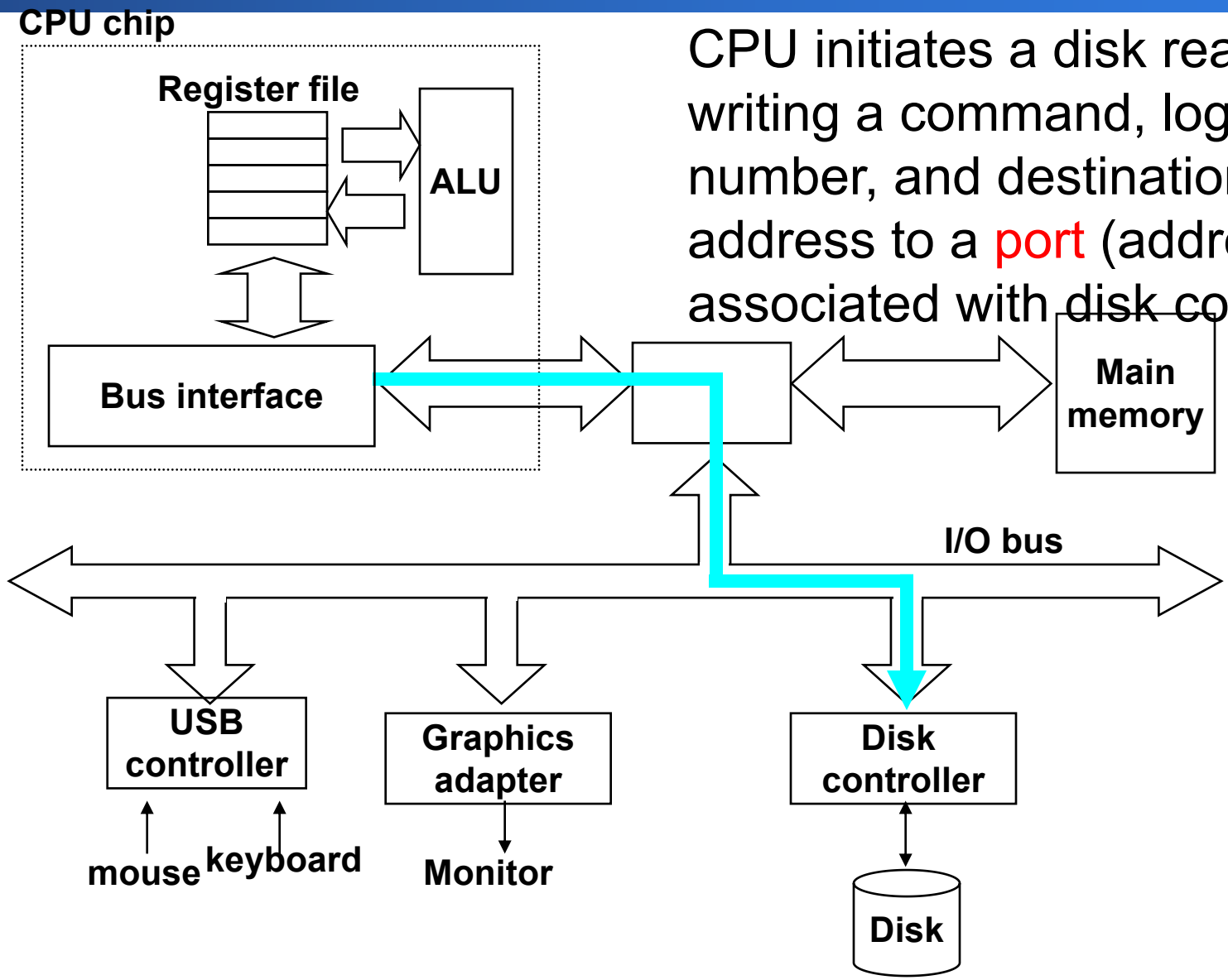
Logical Disk Blocks

- **现代磁盘复杂的扇形几何被抽象为一个简单逻辑块模型:**
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- **逻辑块和实际(物理)扇区之间的映射**
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.
- **允许控制器为每个区域保留spare cylinders(空闲柱面)**
  - Accounts for the difference in “formatted capacity” and “maximum capacity”.

I/O Bus

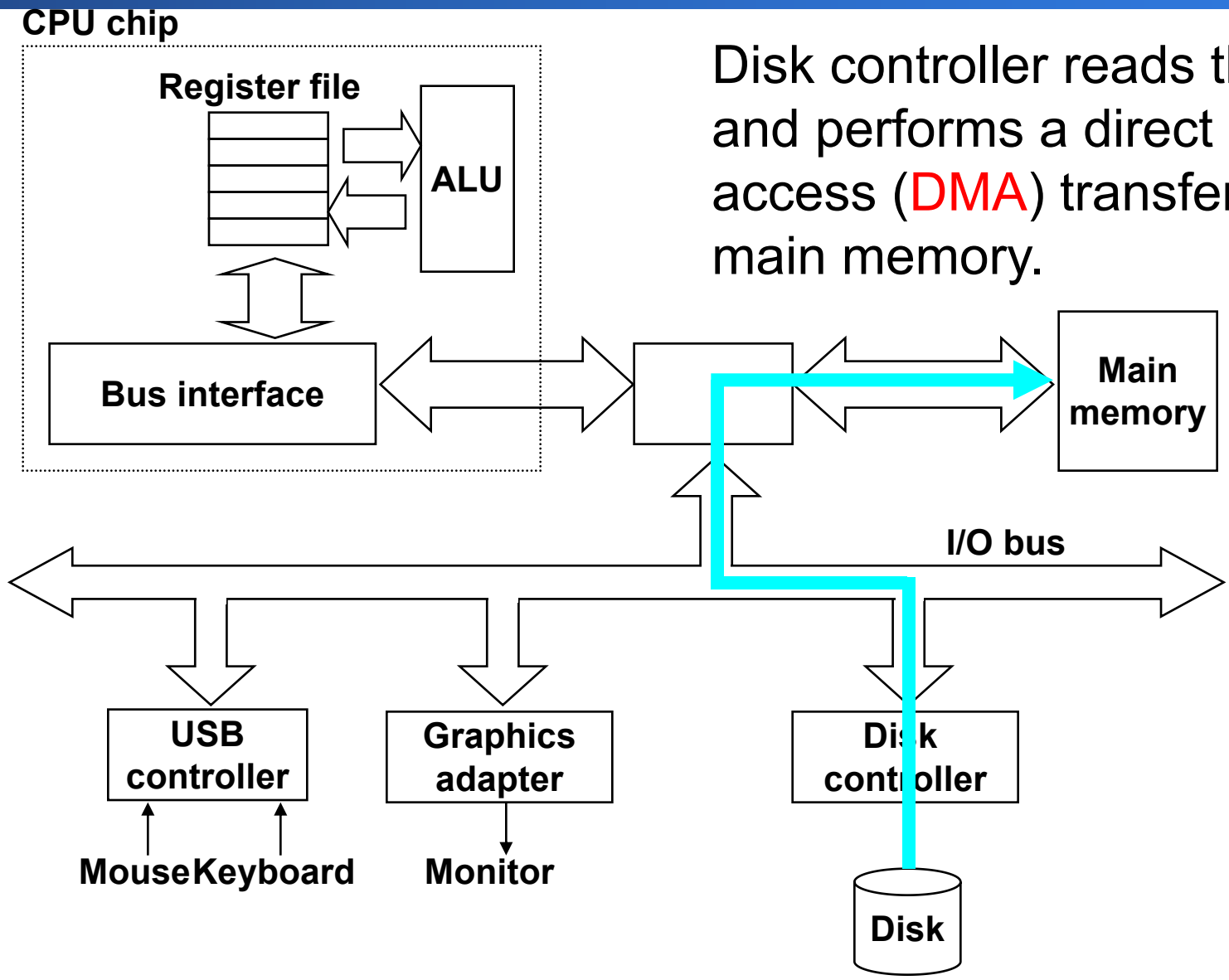

CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

Reading a Disk Sector (1)



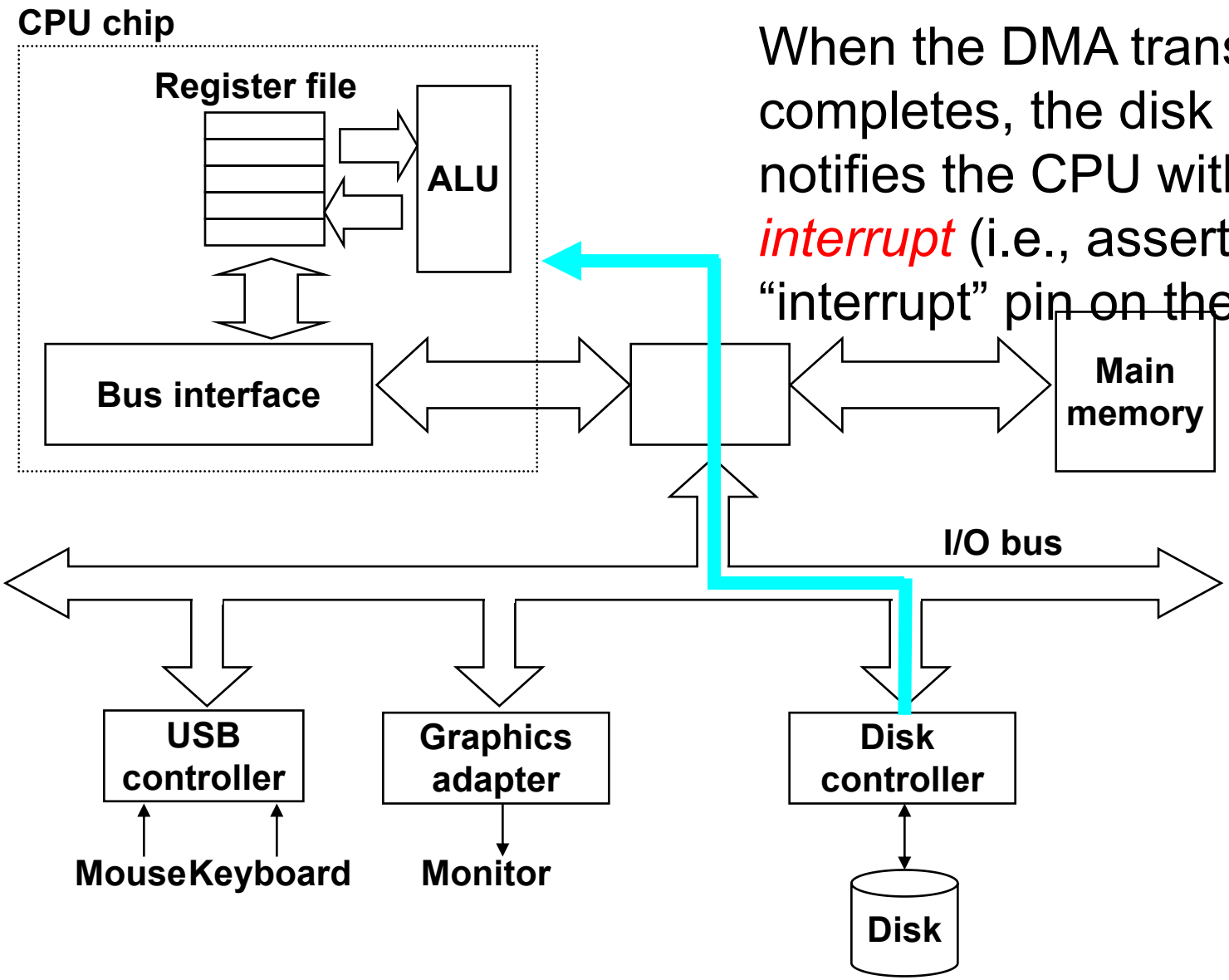
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector (2)



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

Reading a Disk Sector (3)



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)

属性	NOR	NAND
容量	较小	很大
XIP(可执行 code)	可以	不行
读取速度	很快	快
写入速度	慢	快
擦除速度	很慢	快
可擦除次数	10,000-100,000	100,000-1,000,000
可靠性	高	较低
访问方式	可随机访问	块方式
价格	高	微信号: UEFIBlog

- **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

- **Benchmark of Samsung 940 EVO Plus**
  - https://ssd.userbenchmark.com/SpeedTest/711305/Samsung-SSD-970-EVO-Plus-250GB

Sequential read throughput	**2126 MB/s**	**Sequential write throughput**	**1880 MB/s**
Random read throughput	**140 MB/s**	**Random write tput**	**59 MB/s**

- **Sequential access faster than random access**
- **Random writes are somewhat slower**
  - Erasing a block is slow (around 1 ms)
  - Modifying a block page requires all other pages to be copied to new block.
  - Flash translation layer allows accumulating series of small writes before doing block write.

SSD Tradeoffs vs Rotating Disks

- **优势**
  - No moving parts → faster, less power, more rugged
- **劣势**
  - Have the potential to wear out
    - Mitigated by “wear leveling logic” in flash translation layer
    - E.g. Samsung 940 EVO Plus guarantees 600 writes/byte of writes before they wear out
    - Controller migrates data to minimize wear level
  - In 2019, about 4 times more expensive per byte
    - And, relative cost will keep dropping
- **应用**
  - MP3 players, smart phones, laptops
  - Increasingly common 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	*320*
access (ns)	300	150	35	15	3	2	1.5	*200*

DRAM

Metric	1980	1985	1990	1995	2000	2005	2010	*2010:1980*
$/MB	8,000	880	100	30	1	0.1	0.06	*130,000*
access (ns)	375	200	100	70	60	50	40	*9*
typical size (MB)	0.064	0.256	4	16	64	2,000	8,000	*125,000*

Disk

Metric	1980	1985	1990	1995	2000	2005	2010	*2010:1980*
$/MB	500	100	8	0.30	0.01	0.005	0.0003	*1,600,000*
access (ms)	87	75	28	10	8	*4*	*3*	*29*
typical size (MB)	1	10	160	1,000	20,000	160,000	1,500,000	*1,500,000*

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.**

Acknowledgements

- **This course was developed and fine-tuned by Randal E. Bryant and David O'Hallaron. They wrote *The Book*!**
- **http://www.cs.cmu.edu/~./213/schedule.html**