

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly  
language:

```
get_mpg:
    pushq    %rbp
    movq     %rsp, %rbp
    ...
    popq     %rbp
    ret
```

Machine  
code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer  
system:



Memory & data  
Integers & floats  
Machine code & C  
x86 assembly  
Procedures & stacks  
Arrays & structs  
**Memory & caches**  
Processes  
Virtual memory  
Memory allocation  
Java vs. C

OS:



*not drawn to scale*

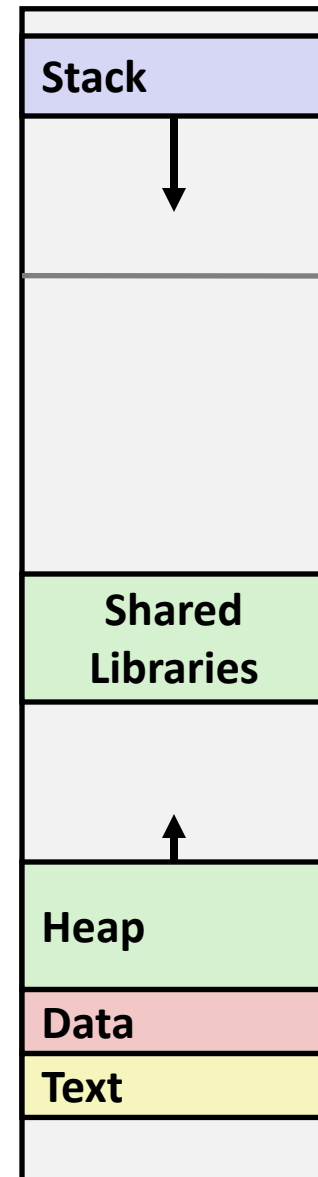
# Memory Allocation Example

```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

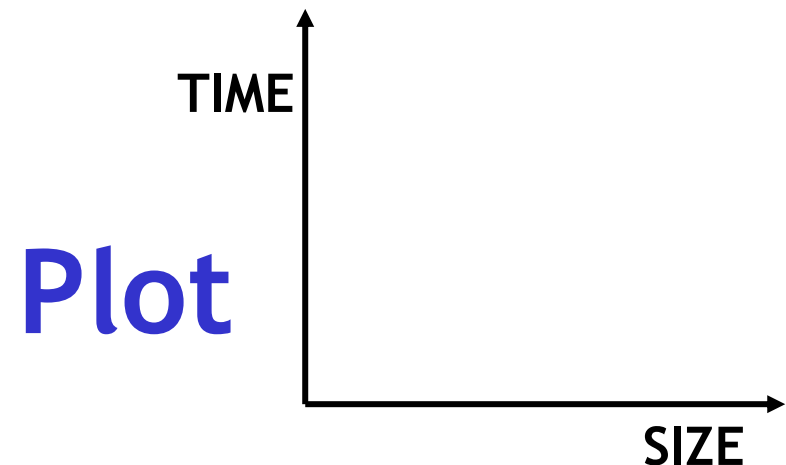
int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```



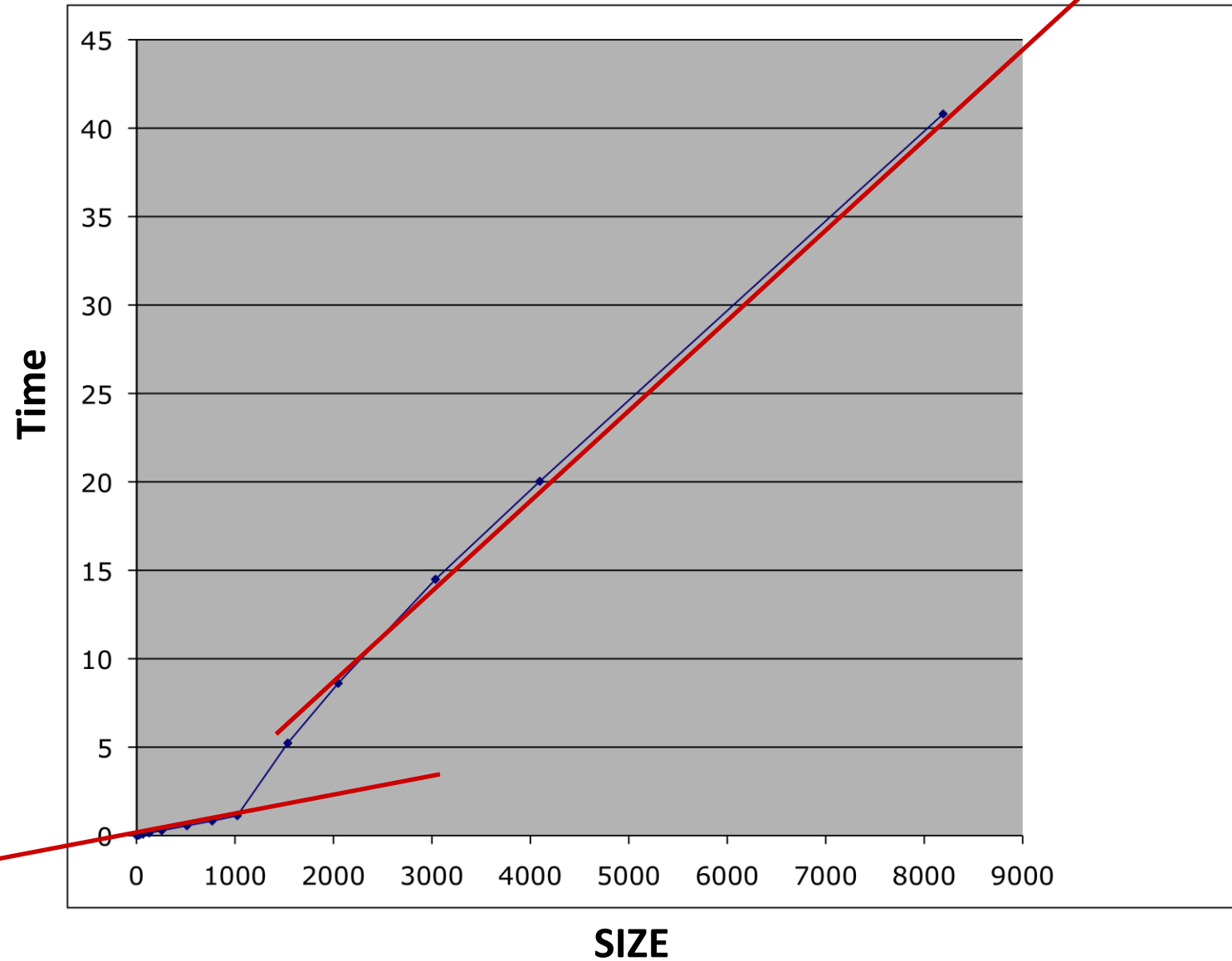
*Where does everything go?*

# How does execution time grow with SIZE?

```
int array[SIZE];  
int A = 0;  
  
for (int i = 0 ; i < 200000 ; ++ i) {  
    for (int j = 0 ; j < SIZE ; ++ j) {  
        A += array[j];  
    }  
}
```



# Actual Data

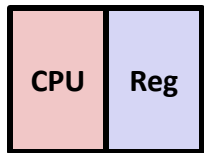


# Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

# Problem: Processor-Memory Bottleneck

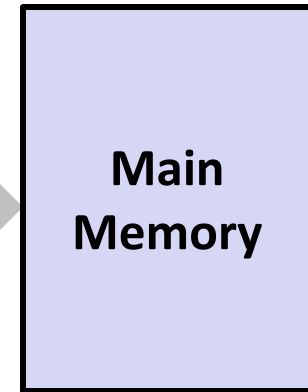
Processor performance  
doubled about  
every 18 months



Bus bandwidth  
evolved much slower



Main  
Memory



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

**Core 2 Duo:**  
Bandwidth  
2 Bytes/cycle  
Latency  
100 cycles

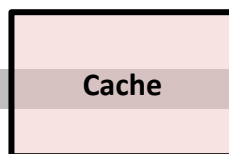
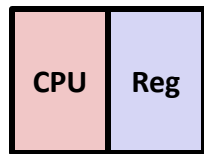


cycle = single fixed-time  
machine step

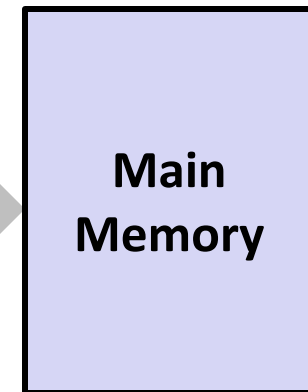
***Problem: lots of waiting on memory***

# Problem: Processor-Memory Bottleneck

Processor performance  
doubled about  
every 18 months



Bus bandwidth  
evolved much slower



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

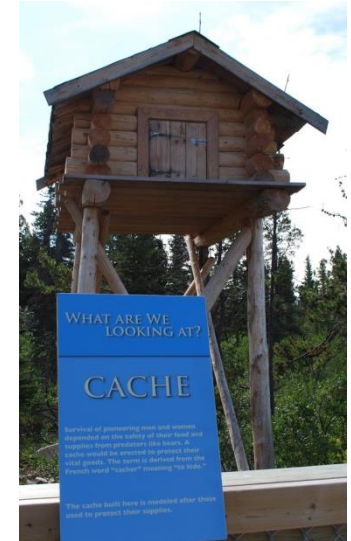
**Core 2 Duo:**  
Bandwidth  
2 Bytes/cycle  
Latency  
100 cycles

***Solution: caches***

cycle = single fixed-time  
machine step

# Cache

- **English definition:** a hidden storage space for provisions, weapons, and/or treasures



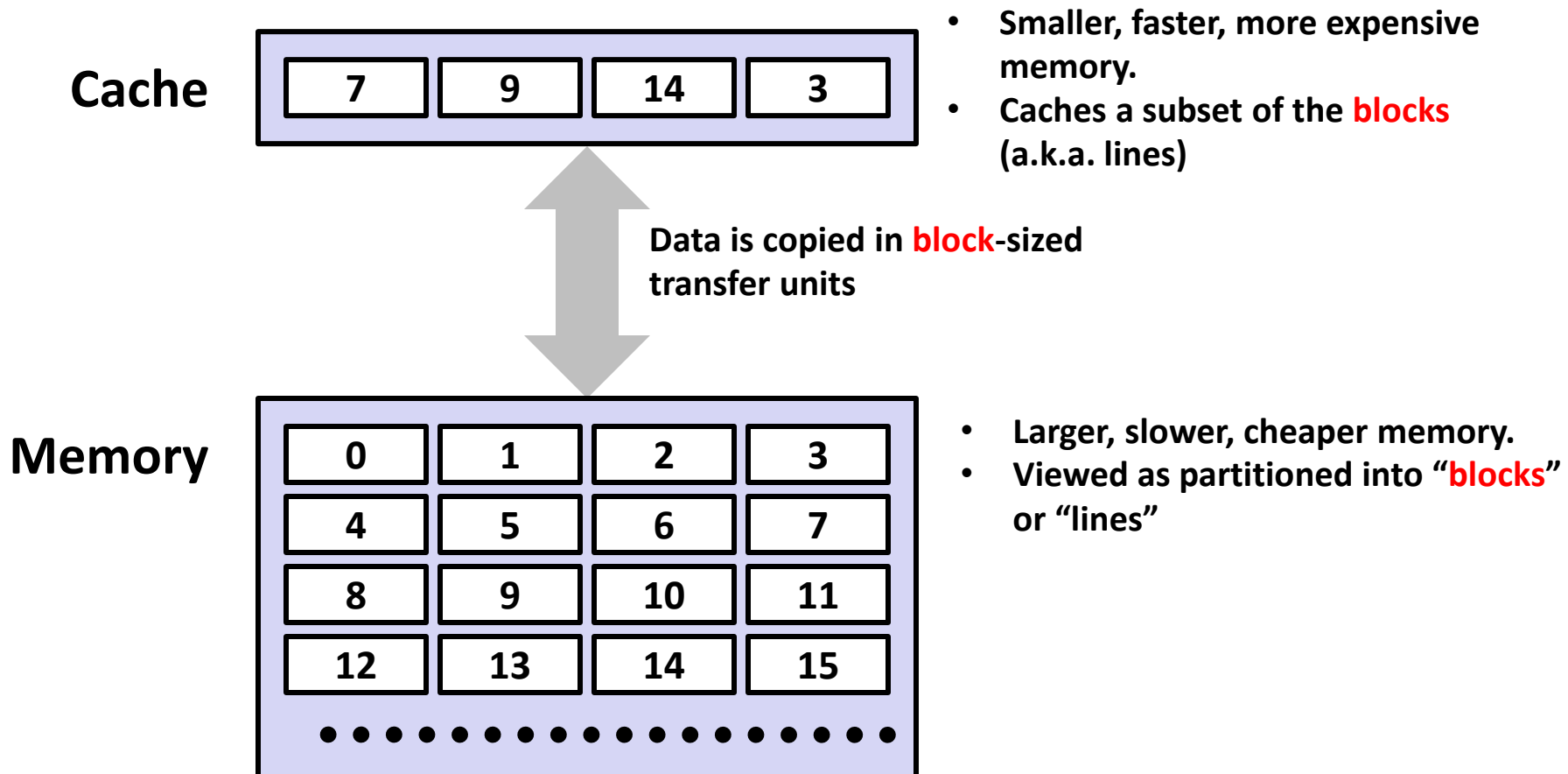
- **CSE definition:** computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

more generally,

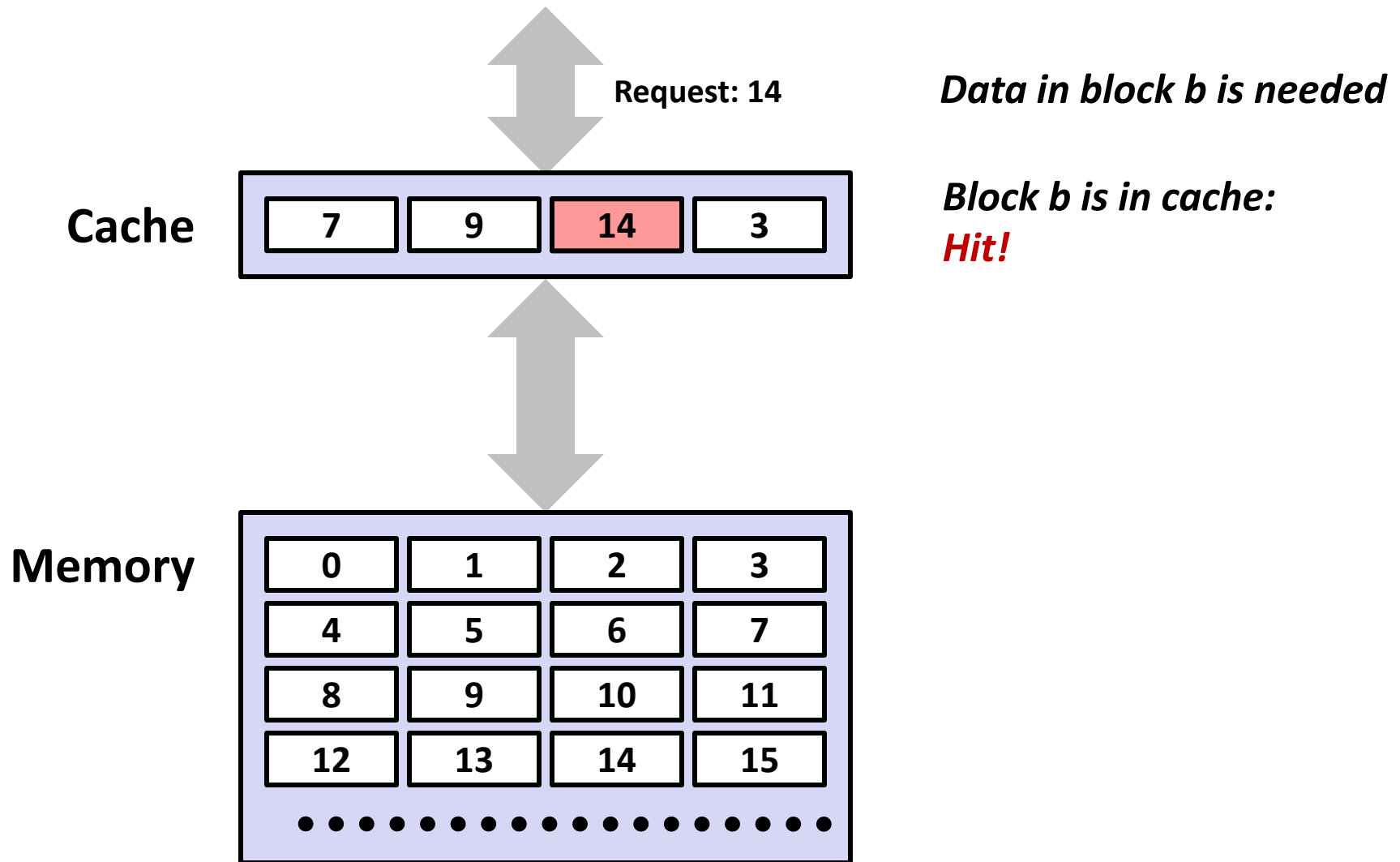
used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)



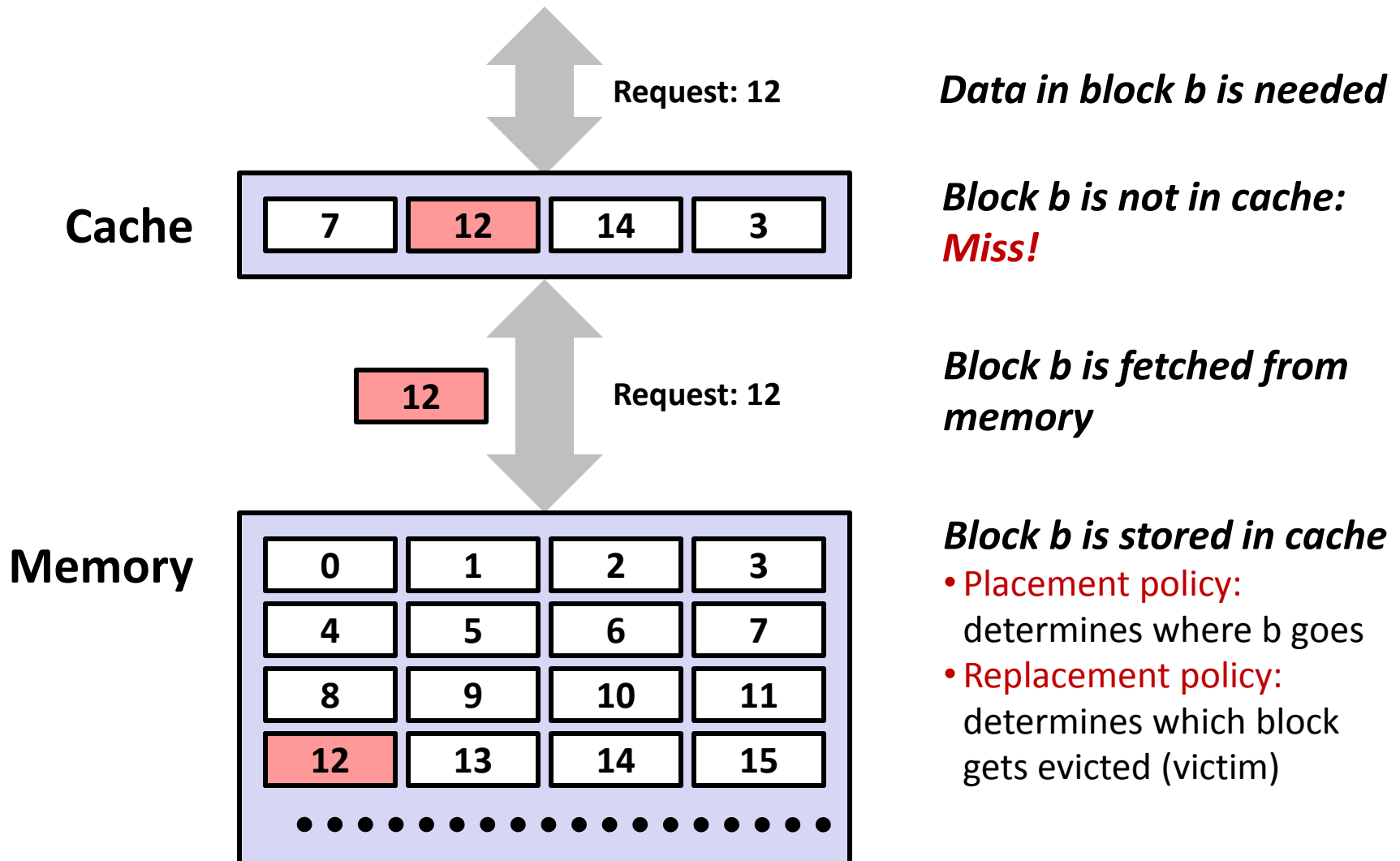
# General Cache Mechanics



# General Cache Concepts: **Hit**



# General Cache Concepts: Miss



# Why Caches Work

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

# Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future
  - Why is this important?



# Why Caches Work

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

- **Temporal locality:**

- Recently referenced items are *likely* to be referenced again in the near future



- **Spatial locality?**

# Why Caches Work

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

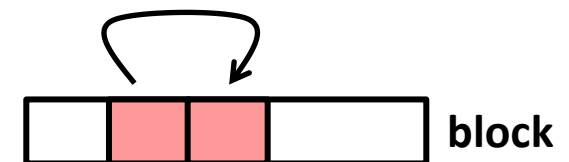
- **Temporal locality:**

- Recently referenced items are *likely* to be referenced again in the near future



- **Spatial locality:**

- Items with nearby addresses *tend* to be referenced close together in time
- How do caches take advantage of this?



# Example: Any Locality?

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



# Example: Any Locality?

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

## ■ Data:

- Temporal: **sum** referenced in each iteration
- Spatial: array **a** [ ] accessed in stride-1 pattern

## ■ Instructions:

- Temporal: cycle through loop repeatedly
- Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**

# Locality Example #1

```
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;
}
```

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

# Locality Example #1

```

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

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

    return sum;
}

```

$M = 3, N = 4$

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

Order  
Accessed

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

Layout in Memory

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

76                      92                      108

76 is just one possible starting address of array a

**stride-1**

# Locality Example #2

```
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;
}
```

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

# Locality Example #2

```

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

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

```

$M = 3, N = 4$

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

Order  
Accessed

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

Layout in Memory

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

76                      92                      108

stride-**N**

# Locality Example #3

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

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

    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?

- What is wrong with this code?
- How can it be fixed?

The diagram illustrates a 128x4 memory array, totaling 512 bytes (4KB). The array is divided into four 64x4 blocks, each containing 256 bytes. The data is organized into four rows, each representing a different data type or format:

- Row 1 (a):** Contains 128 'a' characters, repeated in four groups of 32.
- Row 2 (0):** Contains 128 '0' characters, repeated in four groups of 32.
- Row 3 (0):** Contains 128 '0' characters, repeated in four groups of 32.
- Row 4 (0):** Contains 128 '0' characters, repeated in four groups of 32.

Arrows indicate memory access patterns for different data types, with labels below the array:

- 76:** Points to the first 32 bytes of the first row (a).
- 92:** Points to the first 32 bytes of the second row (0).
- 108:** Points to the first 32 bytes of the third row (0).
- Memory and Caches:** Points to the first 32 bytes of the fourth row (0).

# Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches



# Cost of Cache Misses

- **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

**cycle = single fixed-time  
machine step**

# Cost of Cache Misses

- **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

**cycle = single fixed-time  
machine step**

- Average access time: **check the cache every 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”**

# Cache Performance Metrics

## ■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)  
= 1 - hit rate
- Typical numbers (in percentages):
  - 3% - 10% for L1
  - Can be quite small (e.g., < 1%) for L2, depending on size, etc.

## ■ Hit Time

- Time to deliver a line in the cache to the processor
  - Includes time to determine whether the line is in the cache
- Typical hit times: 4 clock cycles for L1; 10 clock cycles for L2

## ■ Miss Penalty

- Additional time required because of a miss
- Typically 50 - 200 cycles for missing in L2 & going to main memory  
(Trend: increasing!)

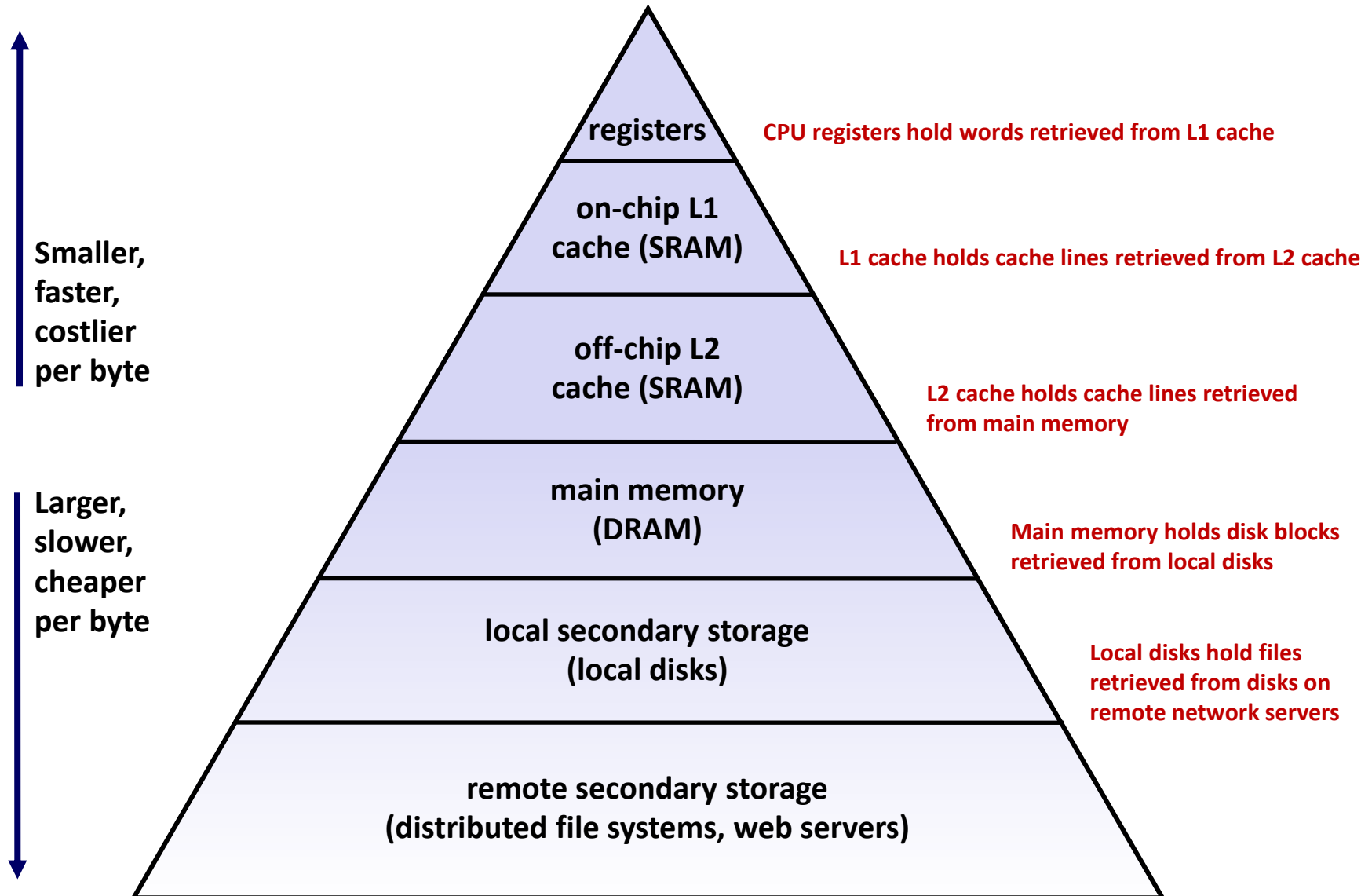
# Can we have more than one cache?

- Why would we want to do that?

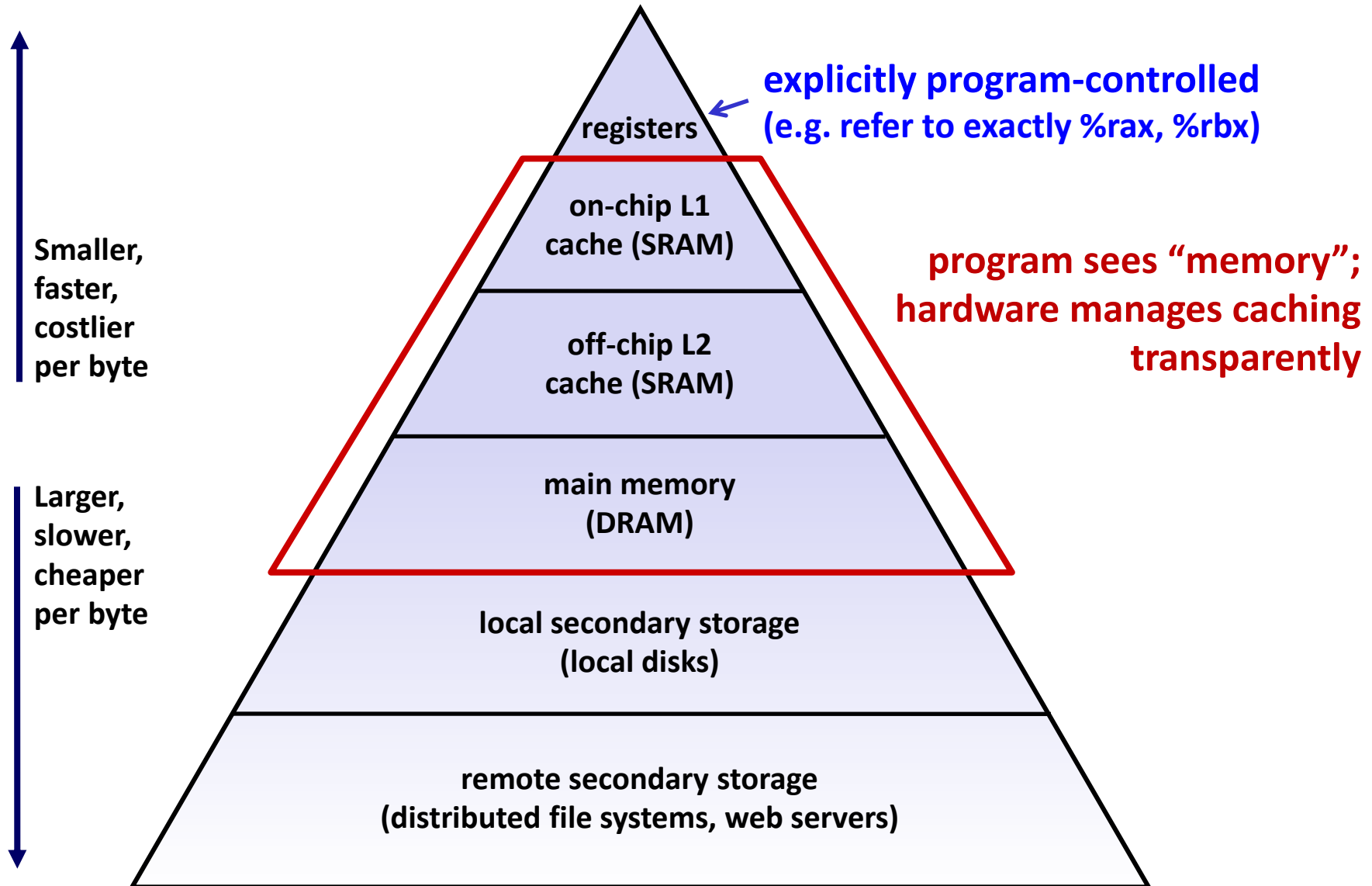
# Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software systems:**
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers  $\leftrightarrow$  cache, cache  $\leftrightarrow$  DRAM, DRAM  $\leftrightarrow$  disk, etc.
  - Well-written programs tend to exhibit good locality
- **These properties complement each other beautifully**
- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy**

# An Example Memory Hierarchy



# An Example Memory Hierarchy



# Memory Hierarchies

## ■ Fundamental idea of a memory hierarchy:

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

## ■ Why do memory hierarchies work?

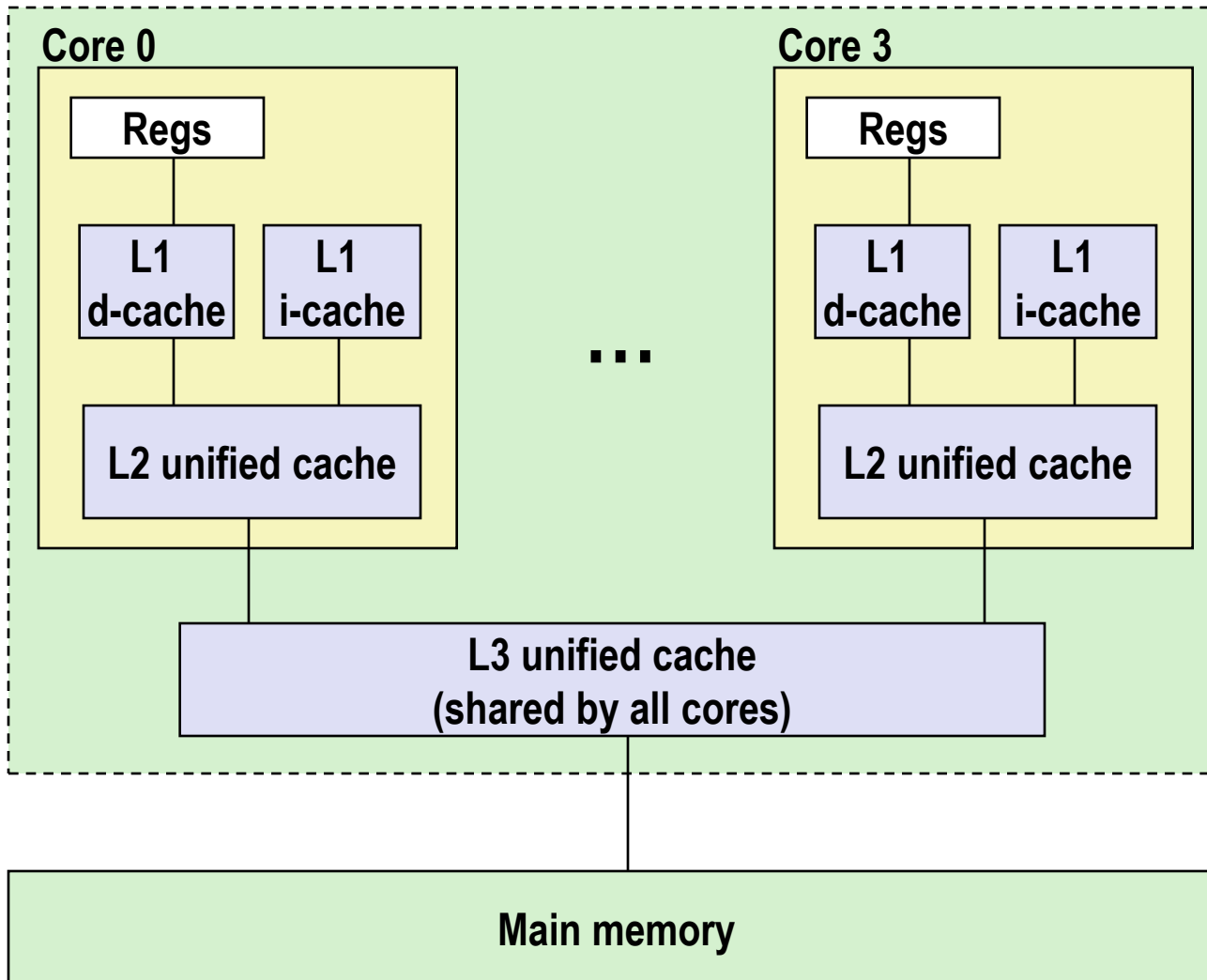
- Because of locality, programs tend to access the data at level  $k$  more often than they access the data at level  $k+1$ .
- Thus, the storage at level  $k+1$  can be slower, and thus larger and cheaper per bit.

- ***Big Idea:*** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.



# Intel Core i7 Cache Hierarchy

## Processor package



### L1 i-cache and d-cache:

32 KB, 8-way,  
Access: 4 cycles

### L2 unified cache:

256 KB, 8-way,  
Access: 11 cycles

### L3 unified cache:

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

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

# Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

# Cache Organization

- **Where should data go in the cache?**
  - We need a mapping from memory addresses to specific locations in the cache to make checking the cache for an address **fast**
    - Otherwise each memory access requires “searching the entire cache” (slow!)
  - What is a data structure that provides fast lookup?

# Aside: Hash Tables for Fast Lookup

Insert:

5  
27  
34  
1002  
119

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Aside: Hash Tables for Fast Lookup

Insert:

000001

000101

110011

101010

100111

0    000

1    001

2    010

3    011

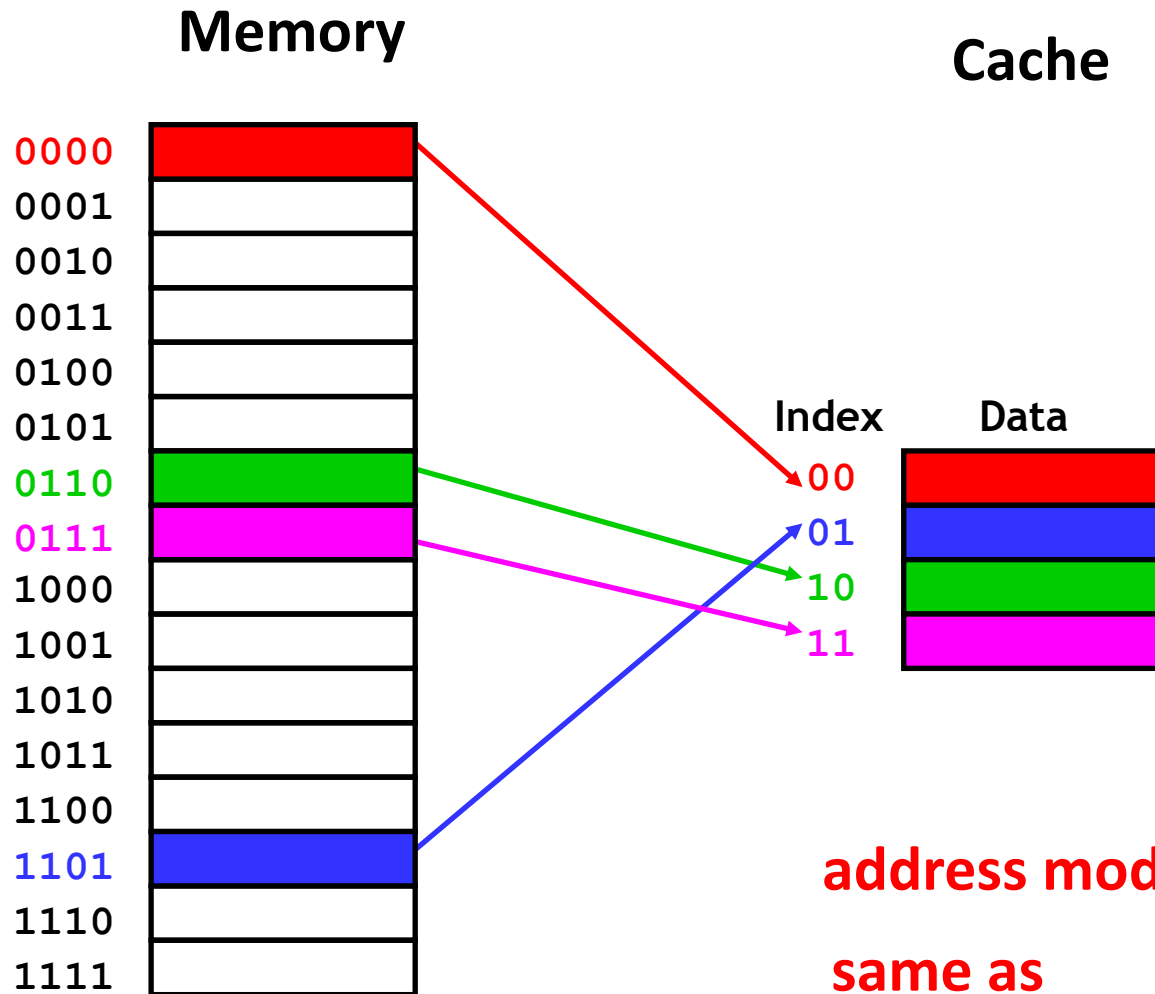
4    100

5    101

6    110

7    111


# Where should we put data in the cache?

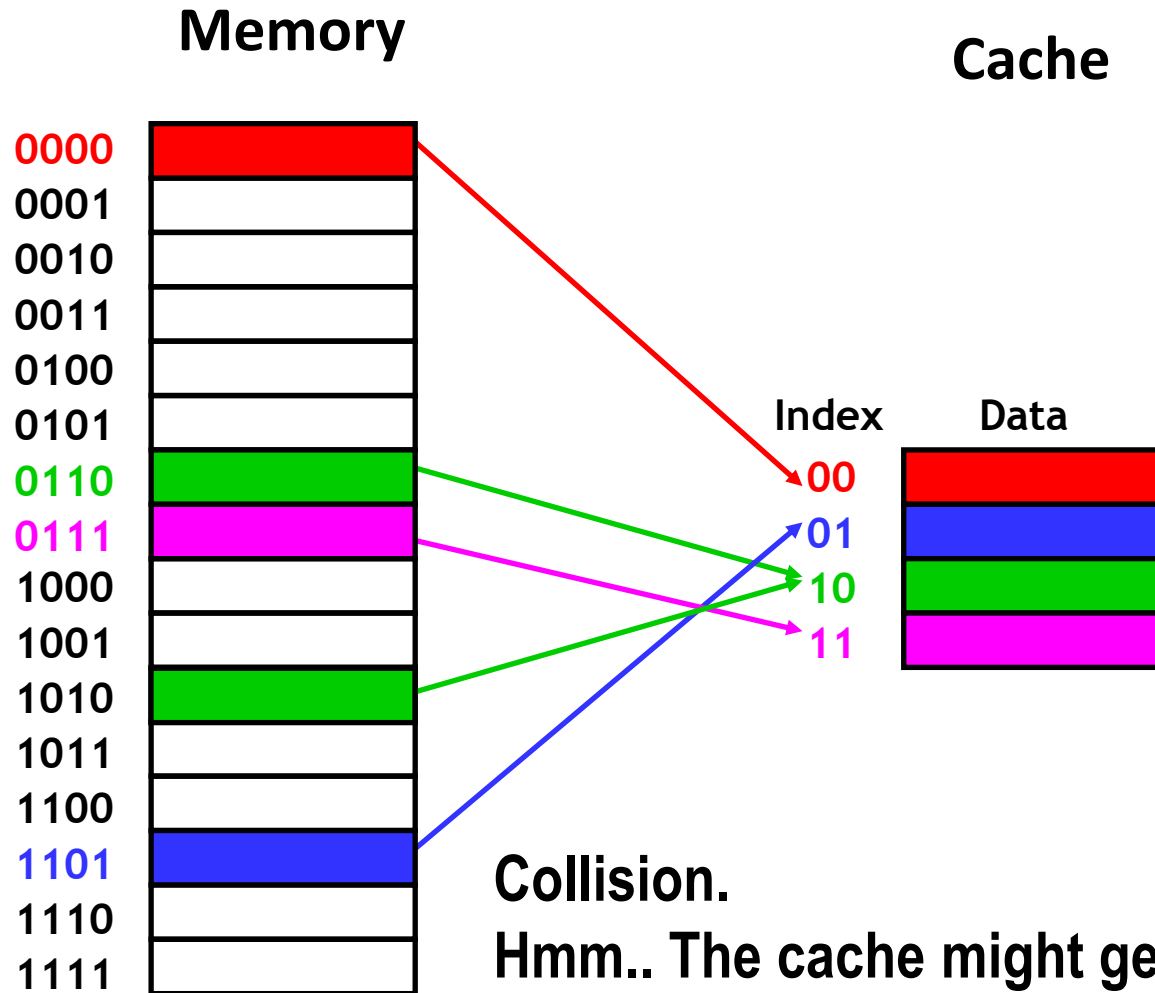


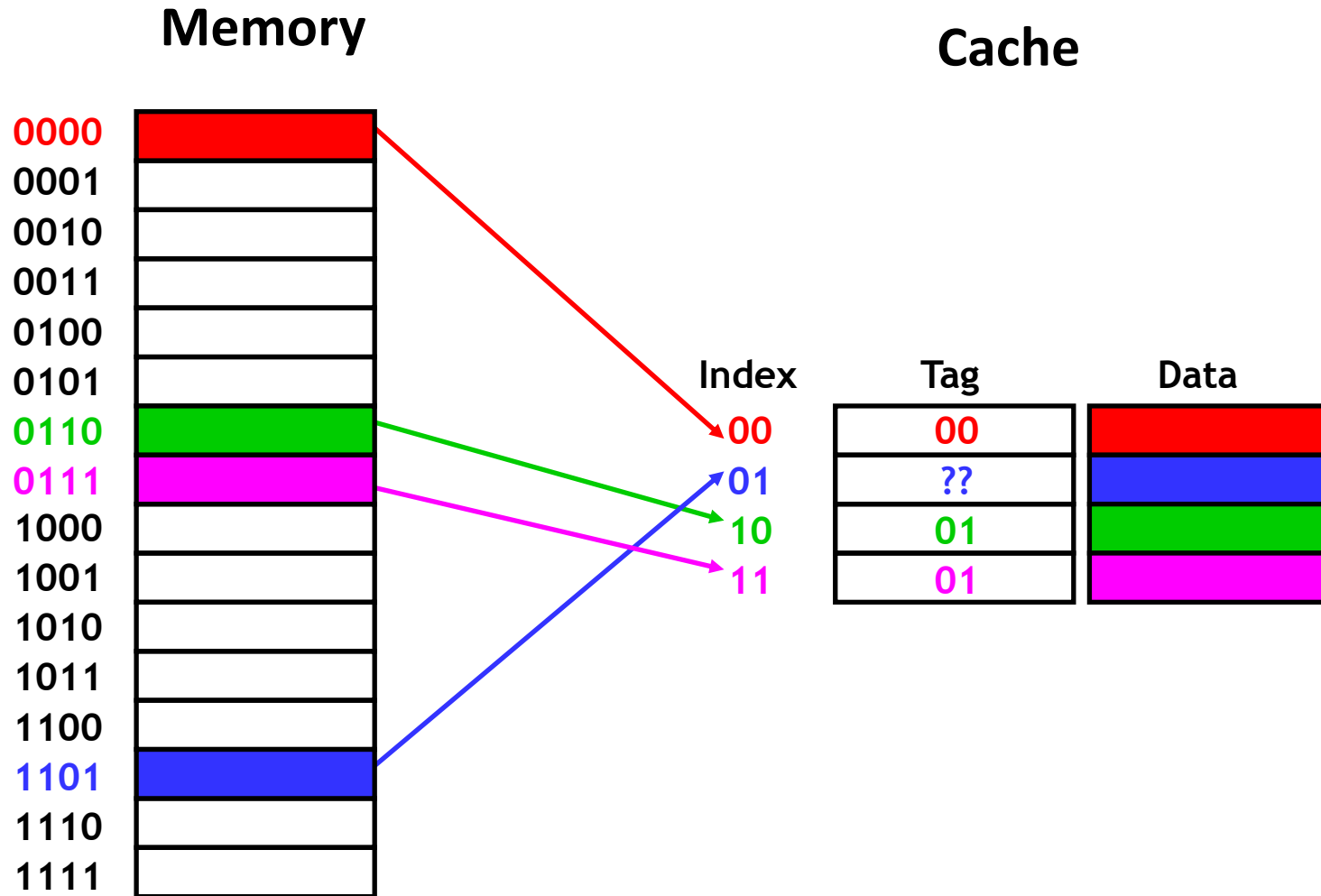
address mod cache size

same as

- How can we compute this mapping? **low-order  $\log_2(\text{cache size})$  bits**

# Where should we put data in the cache?

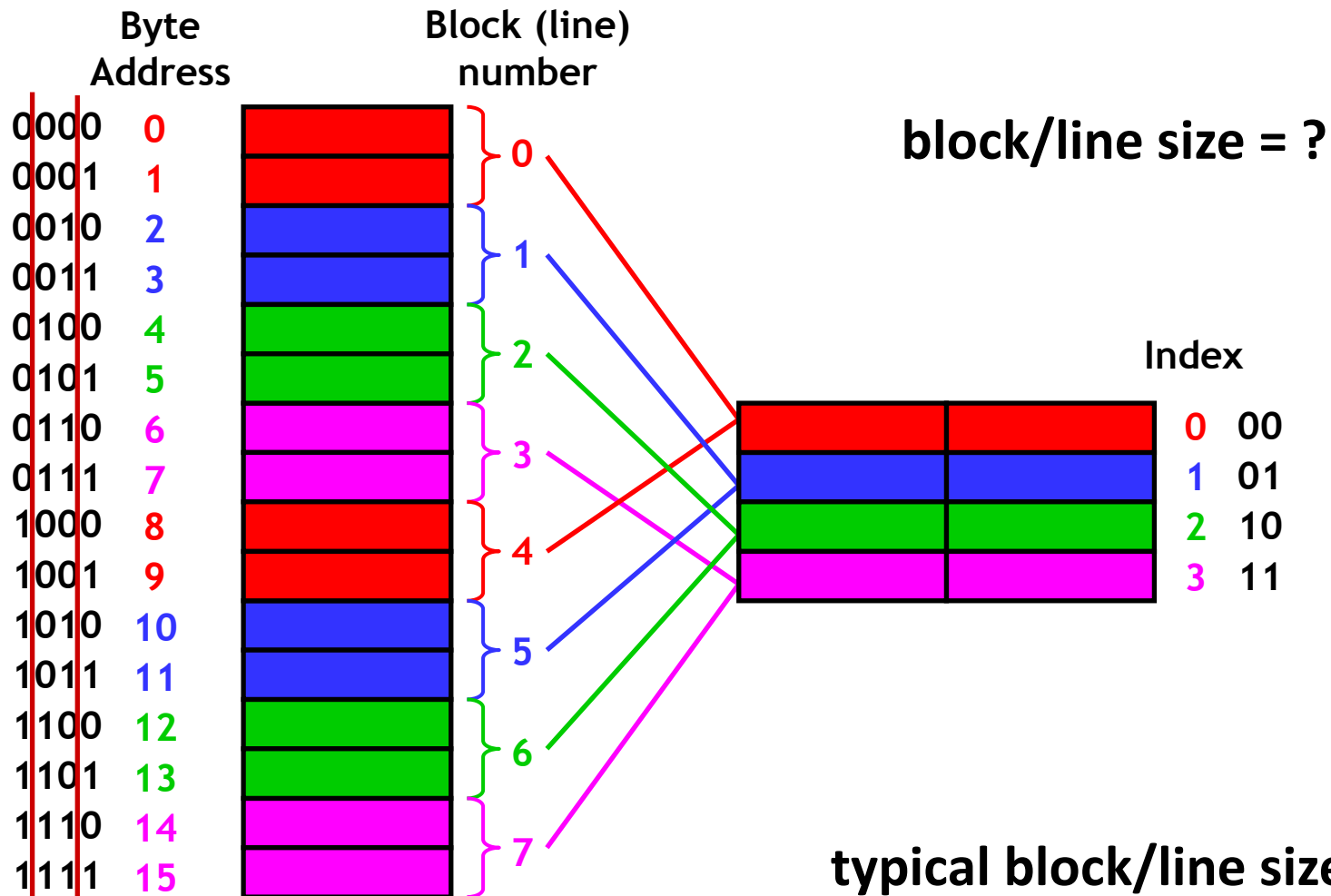




**tag = rest of address bits**




# What's a cache block? (or *cache line*)



# A puzzle.

- What can you infer from this:
- Cache starts *empty*
- Access (addr, hit/miss) stream:
- (10, miss), (11, hit), (12, miss)

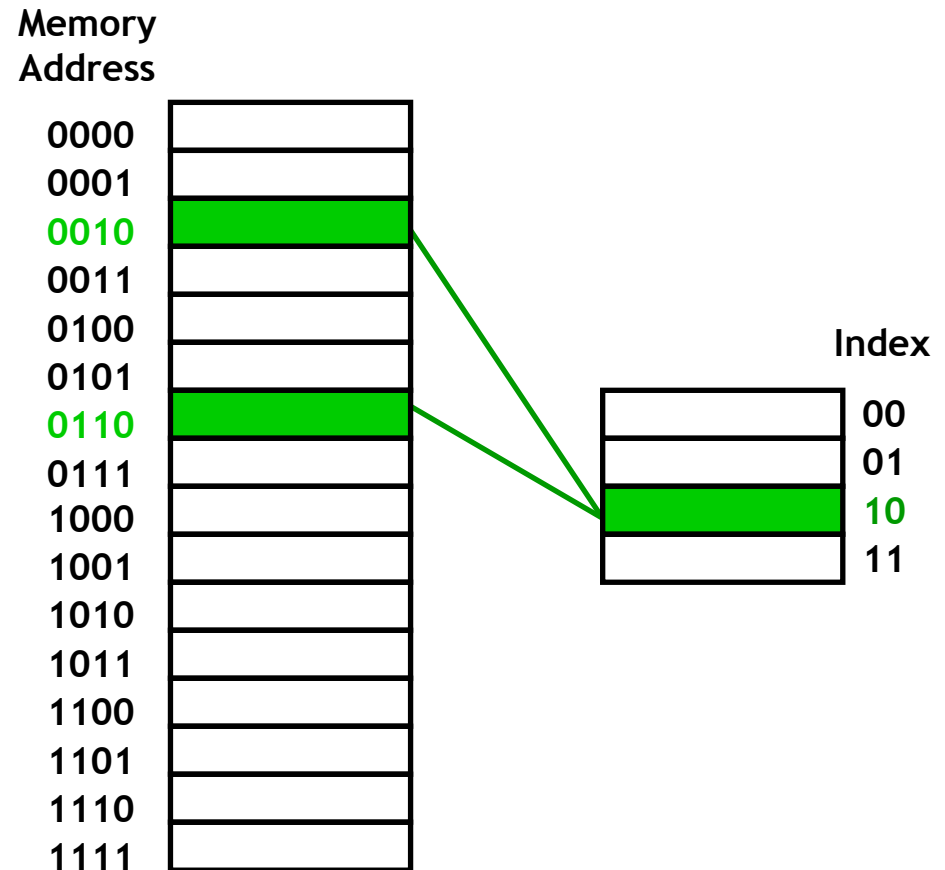
# A puzzle.

- What can you infer from this:
  - Cache starts *empty*
  - Access (addr, hit/miss) stream:
    - (10, miss), (11, hit), (12, miss)
- 
- block size  $\geq 2$  bytes**      **block size  $< 8$  bytes**

# Problems with direct mapped caches?

- **direct mapped:**
  - Each memory address can be mapped to exactly one index in the cache.
- **What happens if a program uses addresses 2, 6, 2, 6, 2, ...?**

**2 and 6 *conflict***



# Associativity

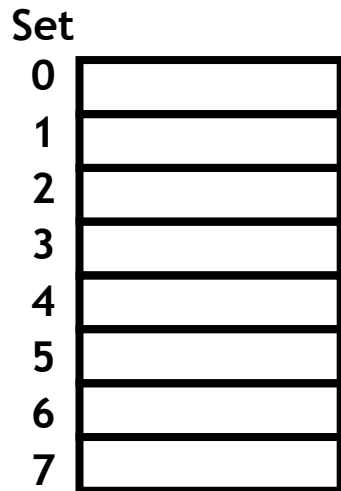
- What if we could store data in *any* place in the cache?

# Associativity

- What if we could store data in *any* place in the cache?
- That might slow down caches (more complicated hardware), so we do something in between.
- Each address maps to exactly one *set*.

**1-way**

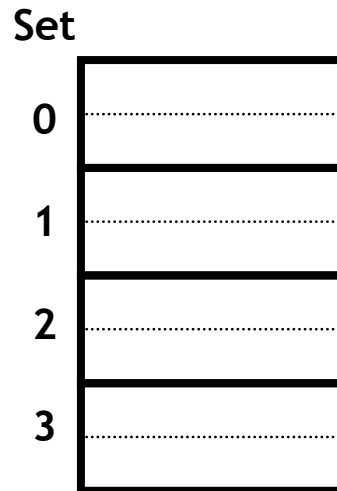
8 sets,  
1 block each



**direct mapped**

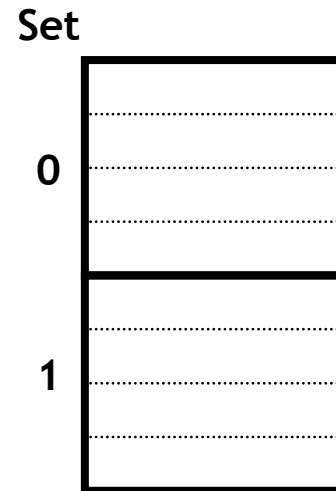
**2-way**

4 sets,  
2 blocks each



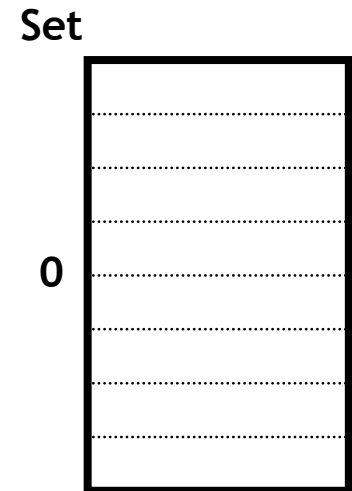
**4-way**

2 sets,  
4 blocks each



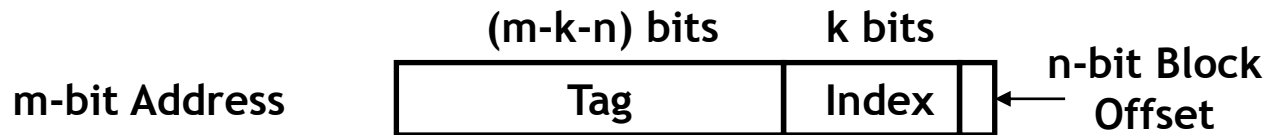
**8-way**

1 set,  
8 blocks

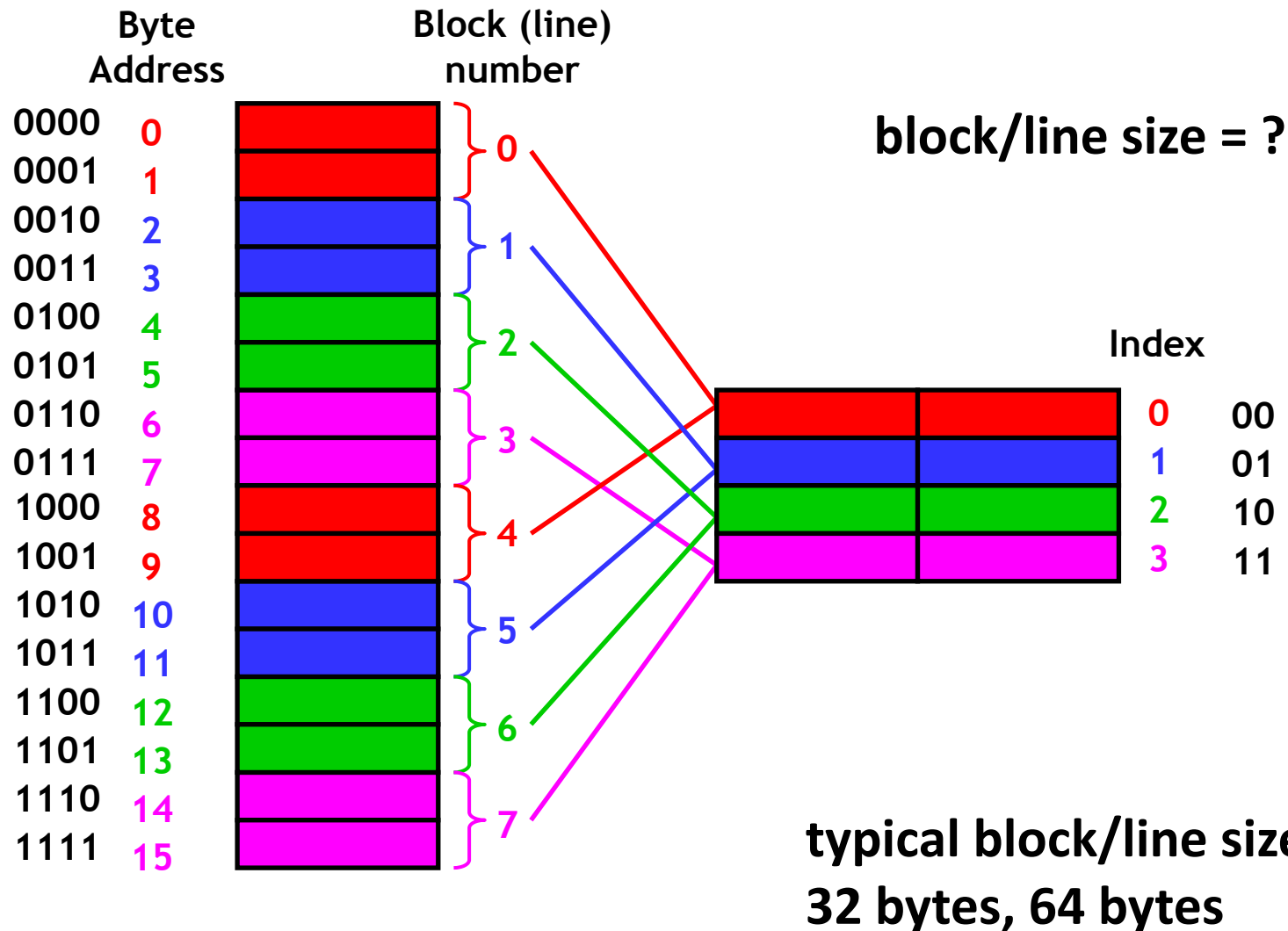


**fully associative**

# Now how do I know where data goes?

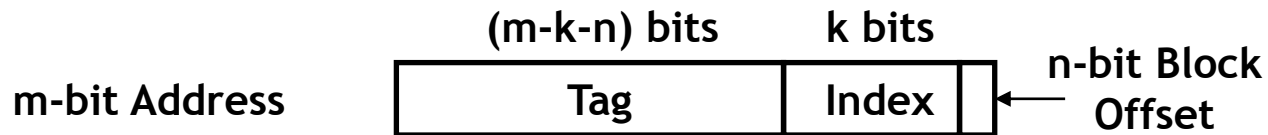


# What's a cache block? (or *cache line*)





# Now how do I know where data goes?

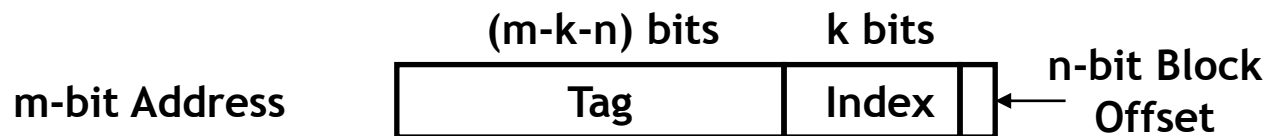


Our example used a  $2^2$ -block cache with  $2^1$  bytes per block. Where would 13 (1101) be stored?



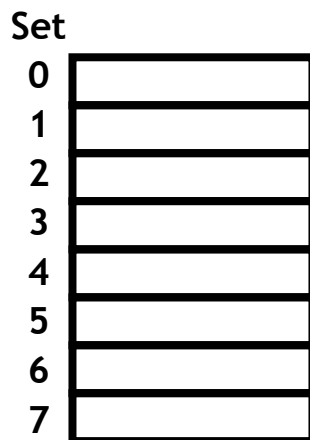
# Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 **011** **0011**.



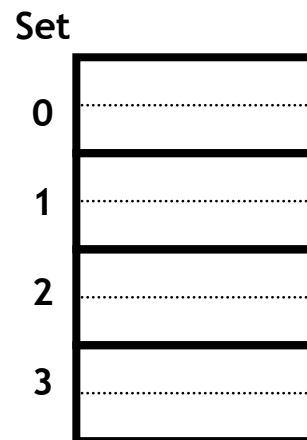
k = ?

1-way associativity  
8 sets, 1 block each



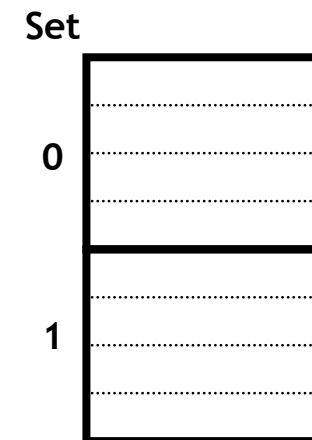
k = ?

2-way associativity  
4 sets, 2 blocks each



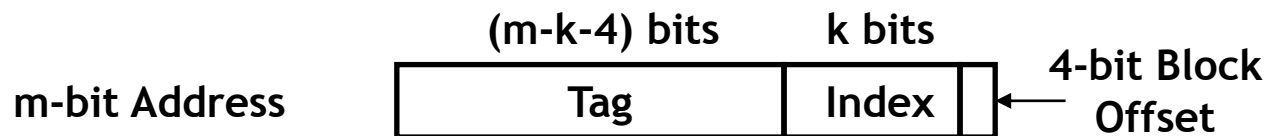
k = ?

4-way associativity  
2 sets, 4 blocks each



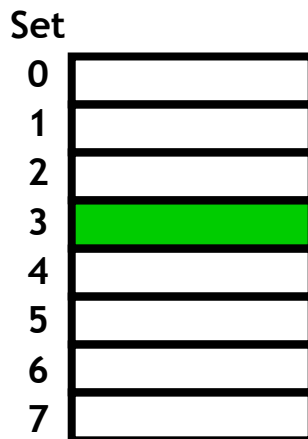
# Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 **011** **0011**.



$k = 3$

1-way associativity  
8 sets, 1 block each



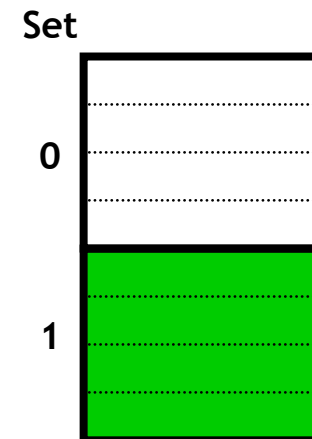
$k = 2$

2-way associativity  
4 sets, 2 blocks each



$k = 1$

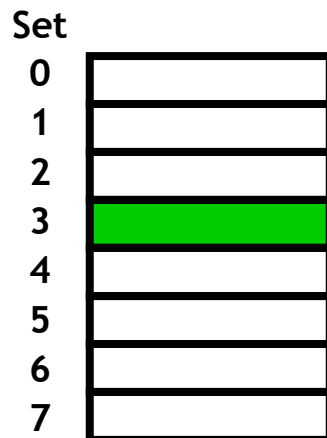
4-way associativity  
2 sets, 4 blocks each



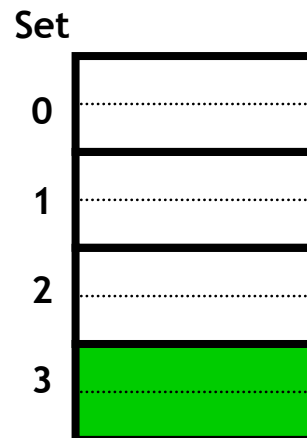
# Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, which one should we replace?

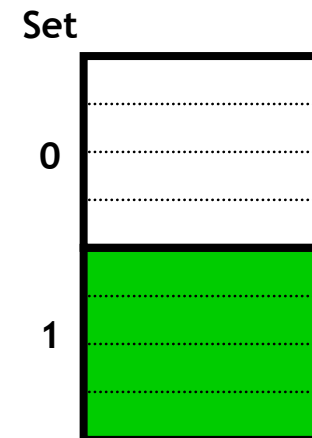
1-way associativity  
8 sets, 1 block each



2-way associativity  
4 sets, 2 blocks each



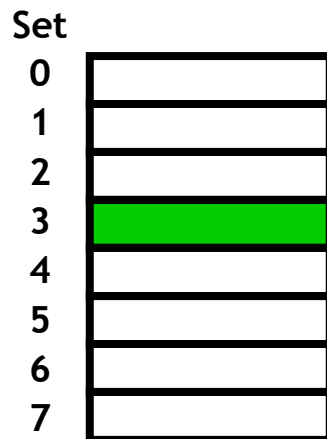
4-way associativity  
2 sets, 4 blocks each



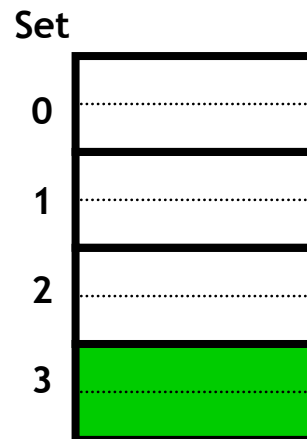
# Block replacement

- Replace something, of course, but what?

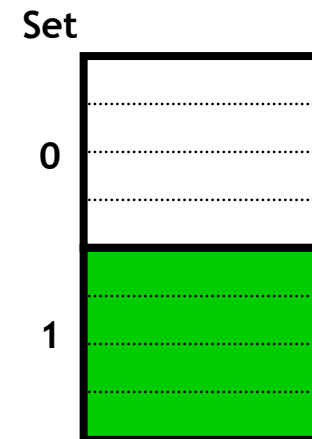
1-way associativity  
8 sets, 1 block each



2-way associativity  
4 sets, 2 blocks each



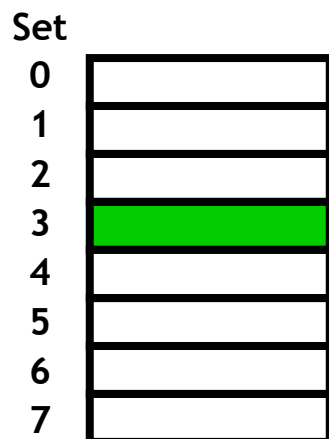
4-way associativity  
2 sets, 4 blocks each



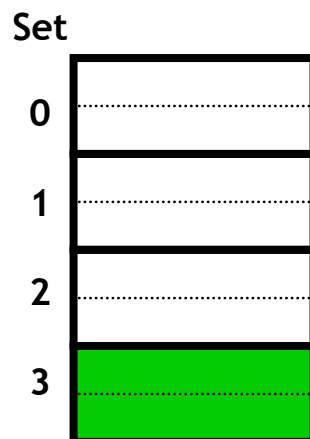
# Block replacement

- Replace something, of course, but what?
  - Obvious for direct-mapped caches, what about set-associative?

1-way associativity  
8 sets, 1 block each



2-way associativity  
4 sets, 2 blocks each



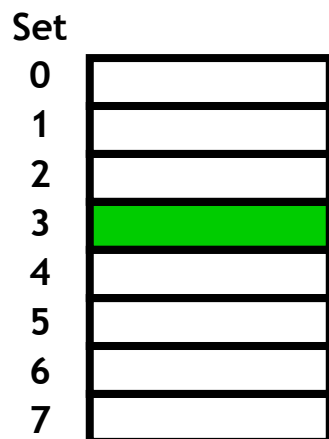
4-way associativity  
2 sets, 4 blocks each



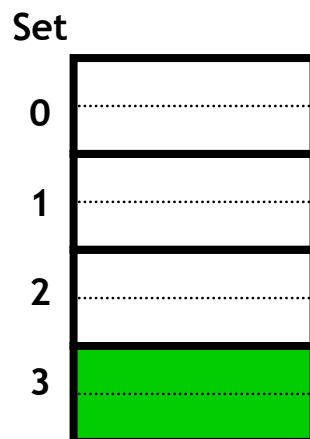
# Block replacement

- **Replace something, of course, but what?**
  - Caches typically use something close to **least recently used (LRU)**
  - (hardware usually implements “not most recently used”)

1-way associativity  
8 sets, 1 block each



2-way associativity  
4 sets, 2 blocks each



4-way associativity  
2 sets, 4 blocks each



# Another puzzle.

- What can you infer from this:
- Cache starts *empty*
- Access (addr, hit/miss) stream
- (10, miss); (12, miss); (10, miss)



# Another puzzle.

- What can you infer from this:
- Cache starts *empty*
- Access (addr, hit/miss) stream
- (10, miss); (12, miss); (10, miss)

**12 is not in the same  
block as 10**

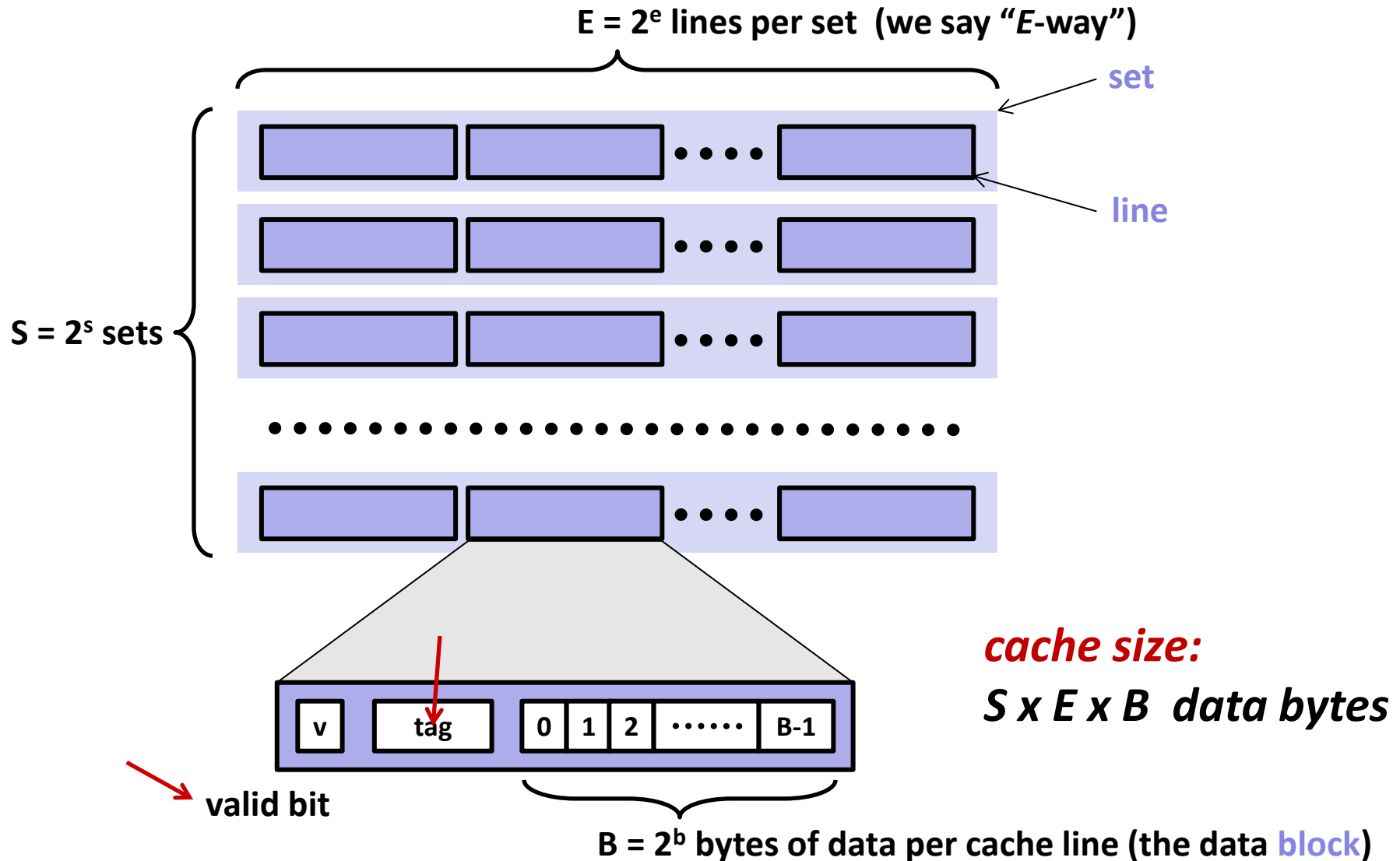


**12's block replaced 10's block**

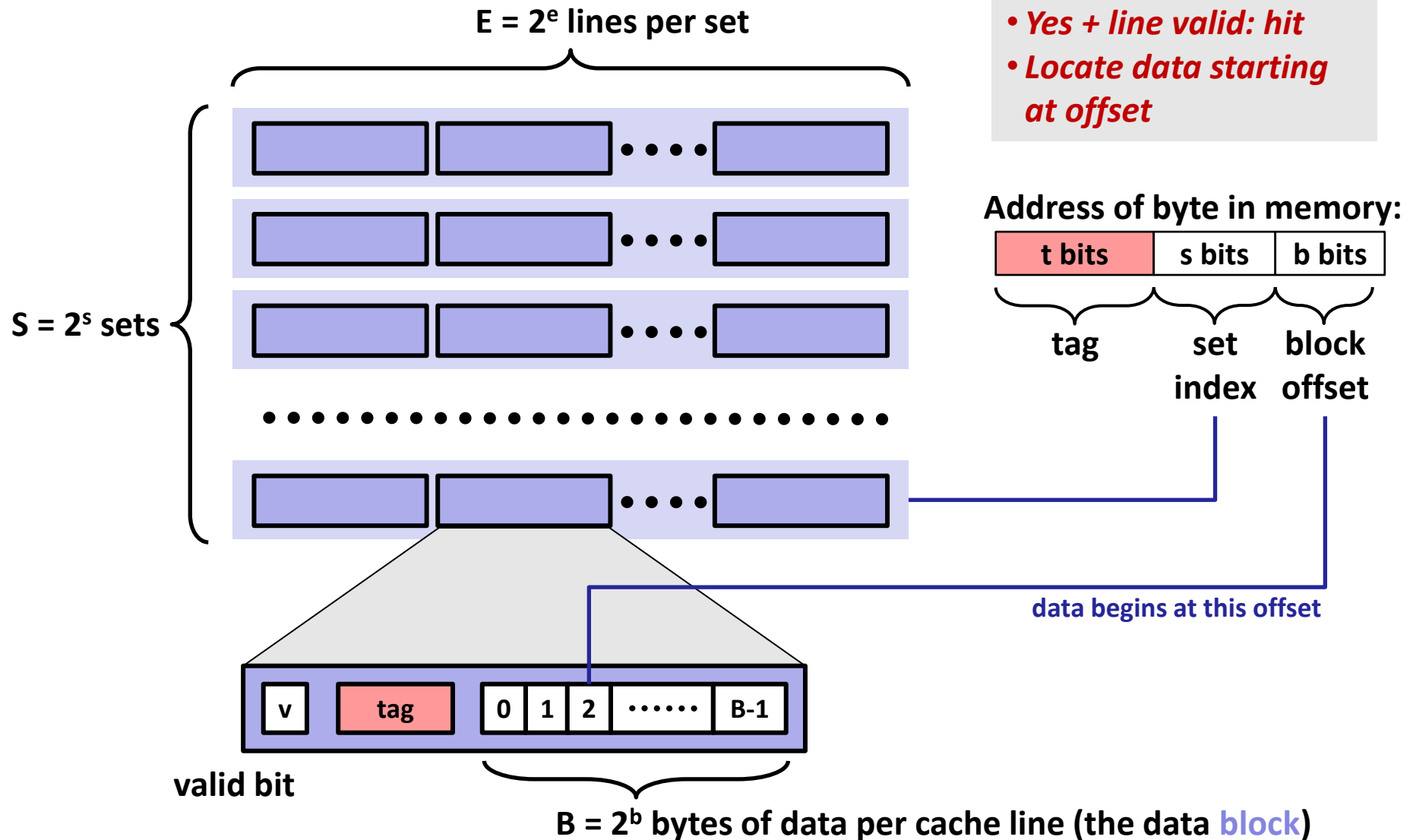


**direct-mapped cache**

# General Cache Organization (S, E, B)

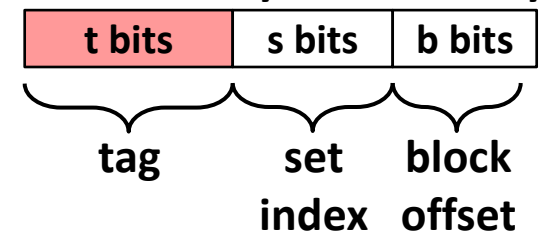


# Cache Read



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

Address of byte in memory:



data begins at this offset

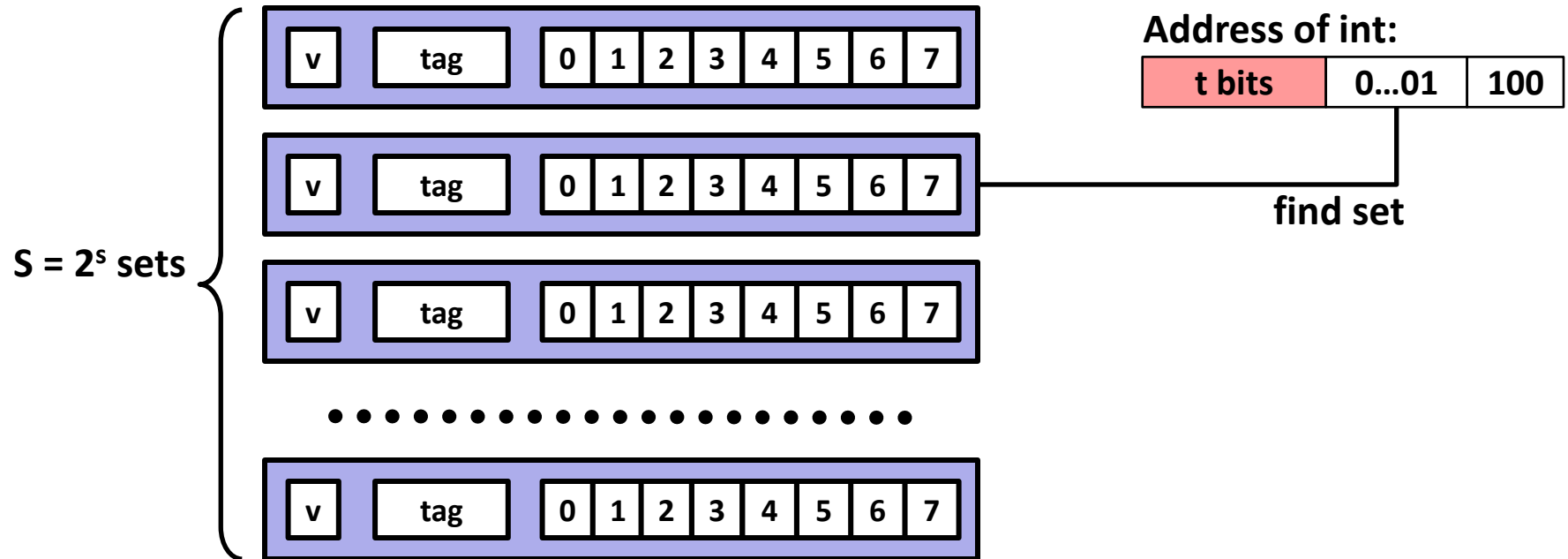
valid bit

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

# Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

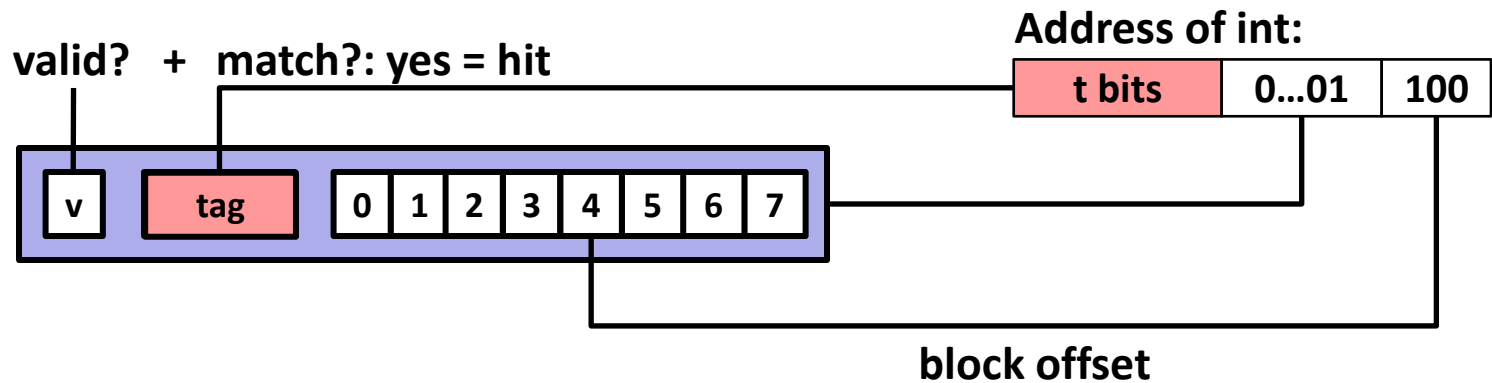
Assume: cache block size 8 bytes



# Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

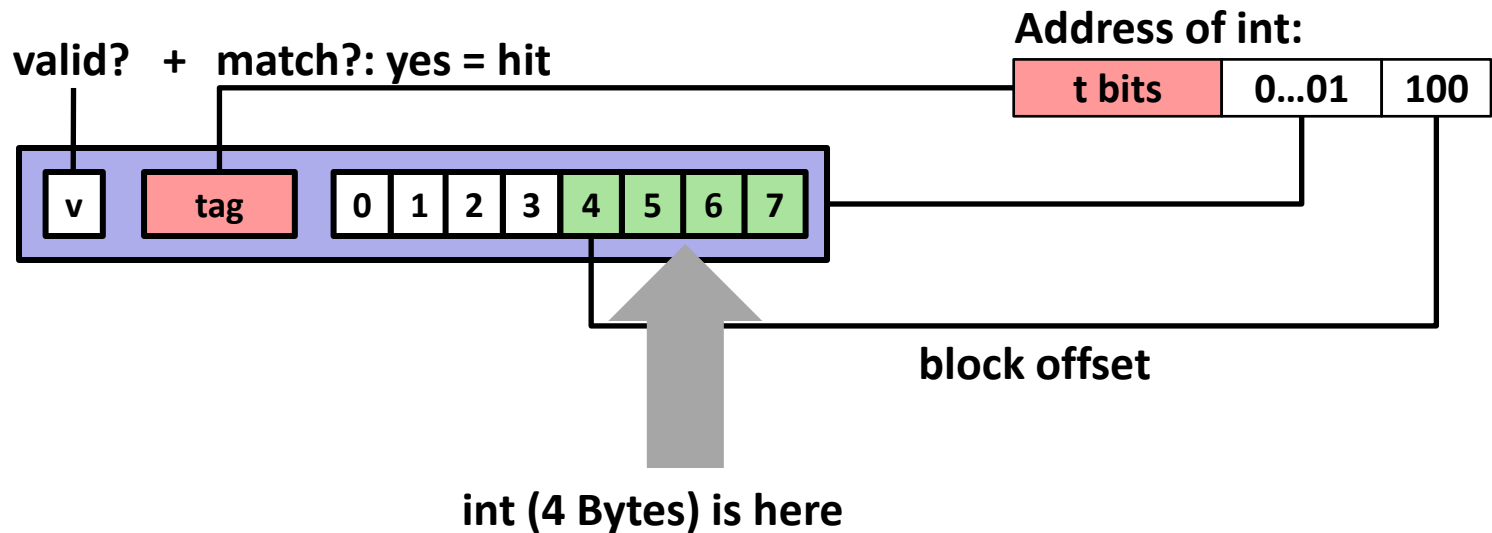
Assume: cache block size 8 bytes



# Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set

Assume: cache block size 8 bytes



**No match:** old line is evicted and replaced

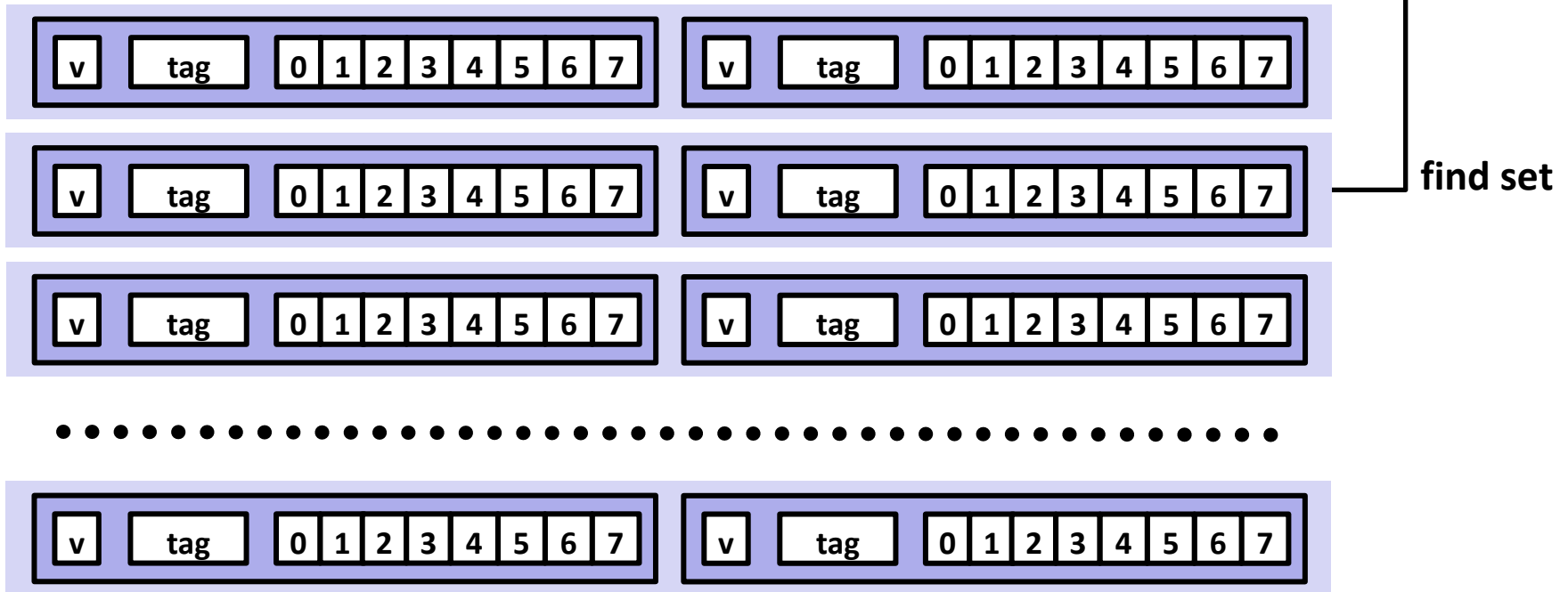
# E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

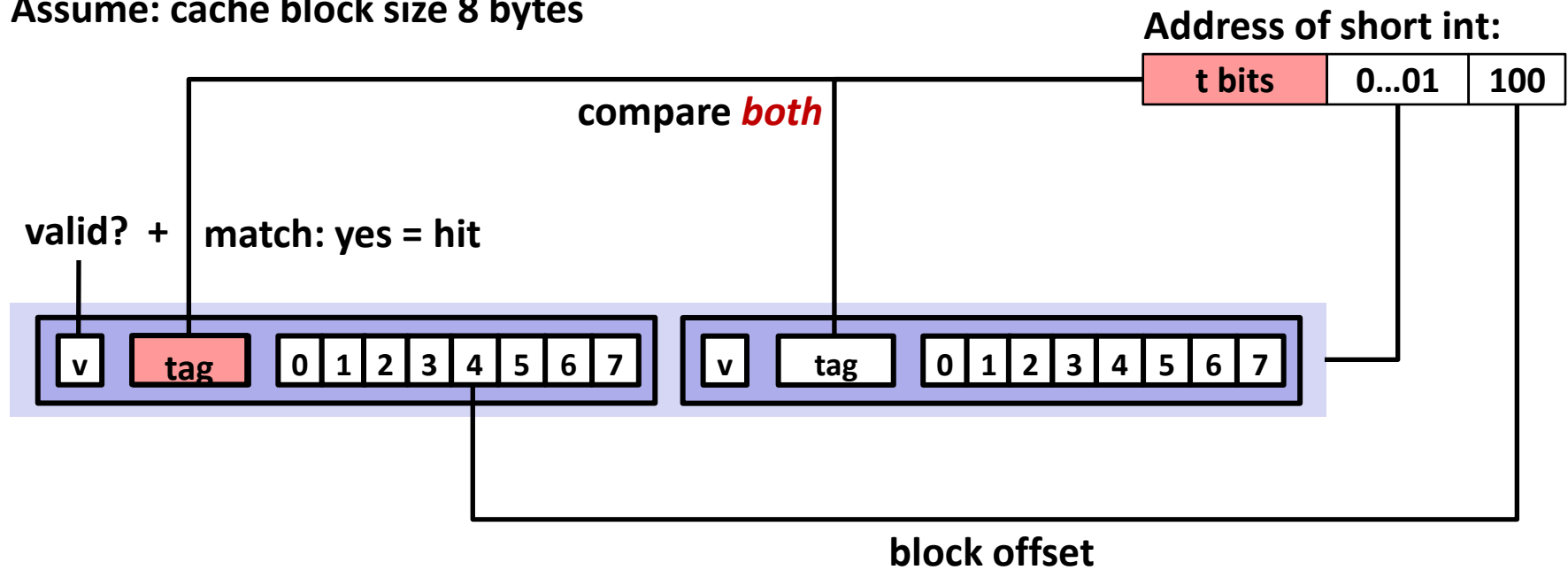
t bits	0...01	100
--------	--------	-----



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

E = 2: Two lines per set

Assume: cache block size 8 bytes

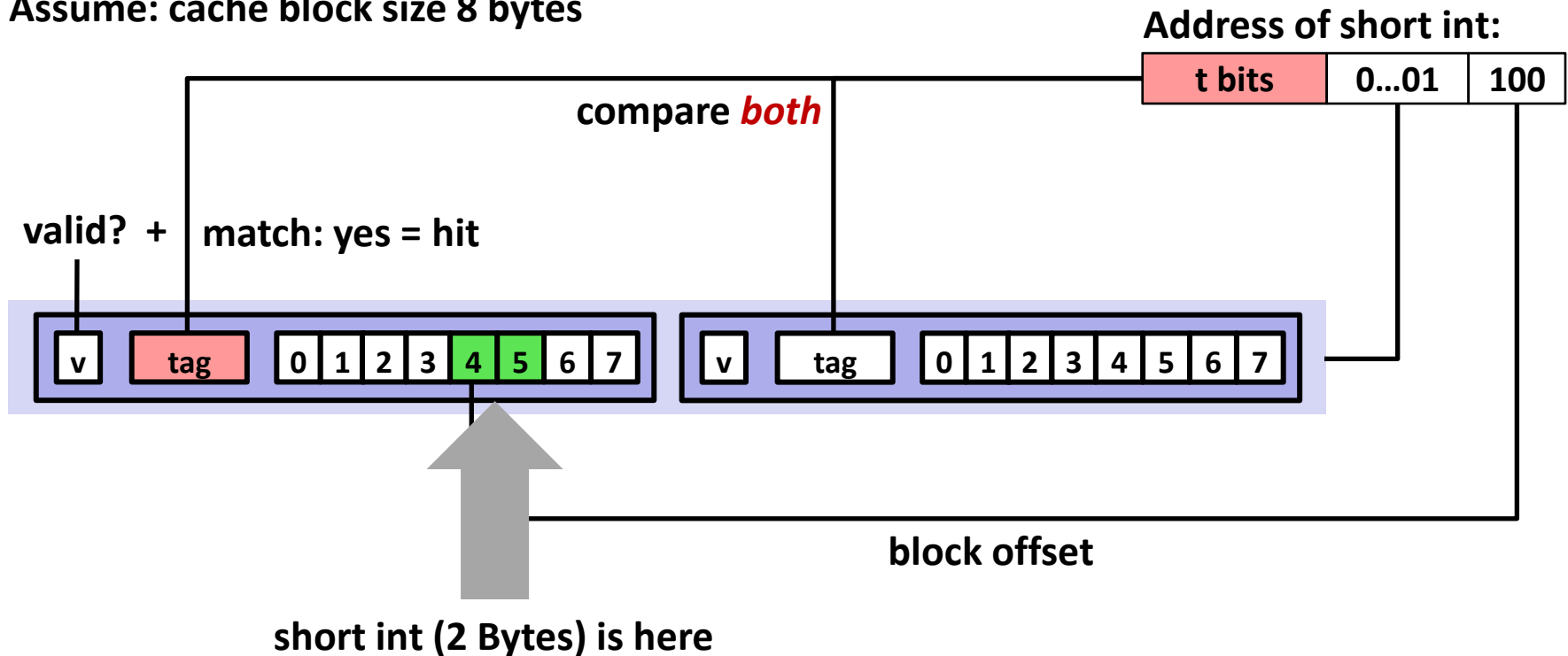




# E-way Set-Associative Cache (Here: E = 2)

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), ...

# Types of Cache Misses

## ■ Cold (compulsory) miss

- Occurs on first access to a block

## ■ Conflict miss

- Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
- direct-mapped caches have more conflict misses than n-way set-associative (where n is a power of 2 and  $n > 1$ )

## ■ Capacity miss

- Occurs when the set of active cache blocks (the working set) is larger than the cache (just won't fit)

# What about writes?

- **Multiple copies of data exist:**
  - L1, L2, possibly L3, main memory
- **What is the main problem with that?**

# What about writes?

## ■ Multiple copies of data exist:

- L1, L2, possibly L3, main memory


## ■ What to do on a write-hit?

- **Write-through**: write immediately to memory, all caches in between.
- **Write-back**: defer write to memory until line is evicted (replaced)
  - Need a *dirty bit* to indicate if line is different from memory or not

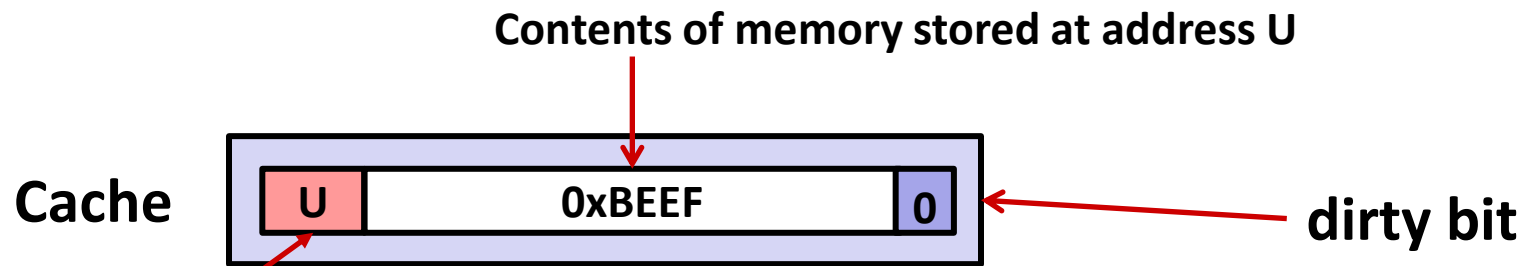
## ■ What to do on a write-miss?

- **Write-allocate**(“**fetch on write**”): load into cache, update line in cache.
  - Good if more writes or reads to the location follow
- **No-write-allocate**(“**write around**”): just write immediately to memory.

## ■ Typical caches:

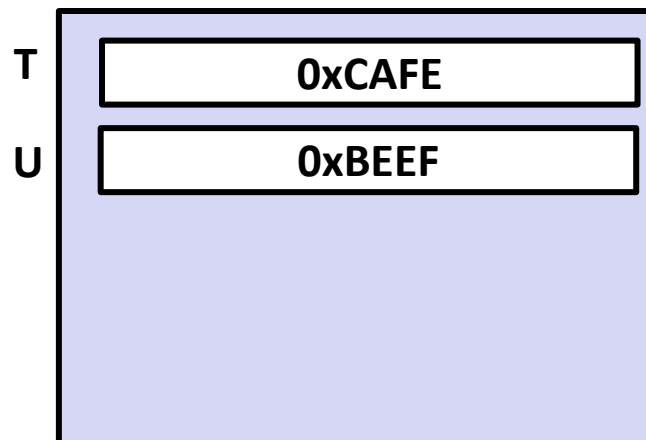
- Write-back + Write-allocate, usually  **why?**
- Write-through + No-write-allocate, occasionally

# Write-back, write-allocate example



tag (there is only one set in this tiny cache, so the tag is the entire address!)

Memory



In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

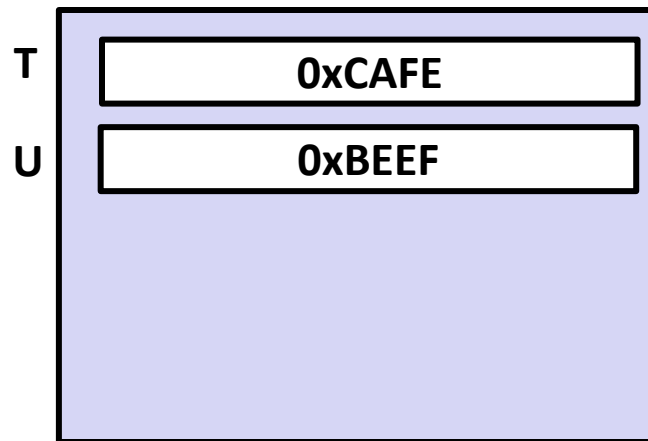
Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

# Write-back, write-allocate example

`mov 0xFACE, T`

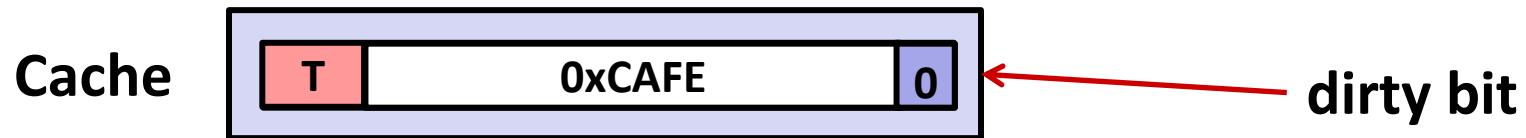


Memory

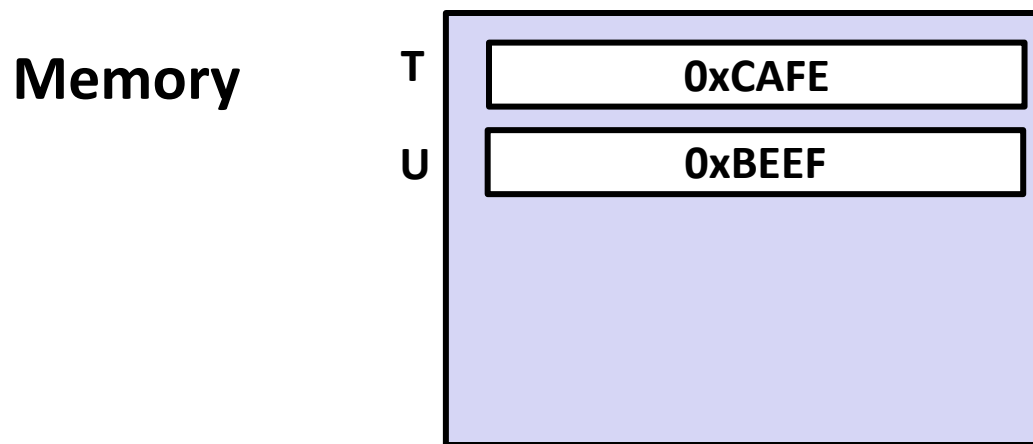


# Write-back, write-allocate example

`mov 0xFACE, T`

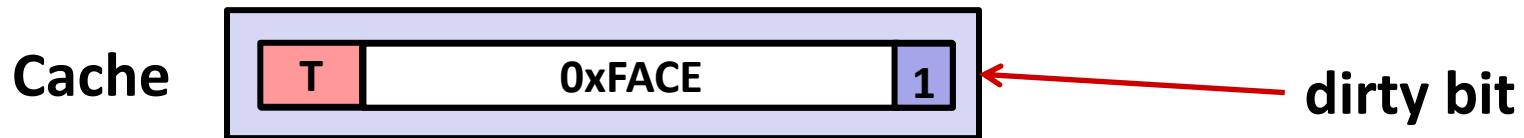


Step 1: Bring T into cache

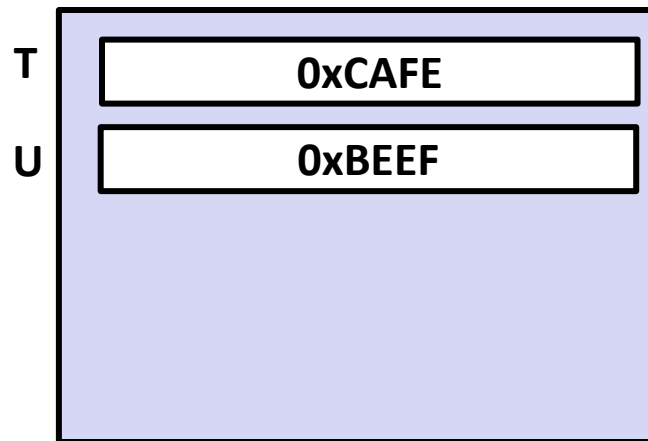


# Write-back, write-allocate example

`mov 0xFACE, T`



Memory

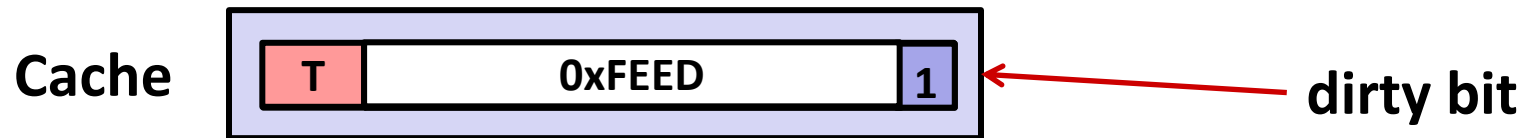


Step 2: Write 0xFACE to cache only and set dirty bit.

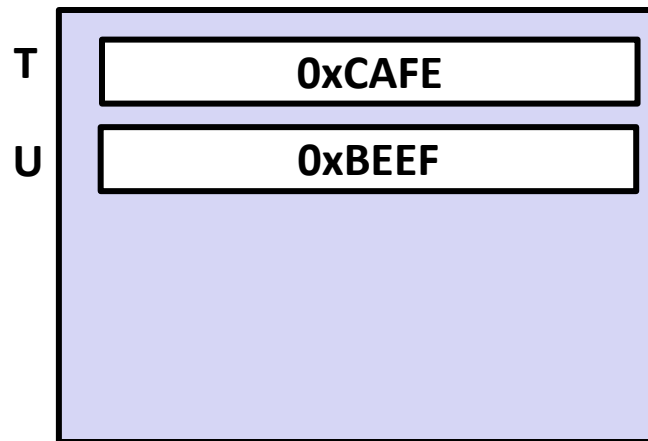


# Write-back, write-allocate example

`mov 0xFACE, T`    `mov 0xFEED, T`



Memory



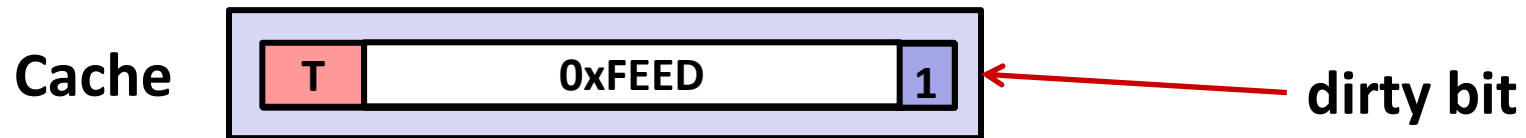
Write hit!  
Write 0xFEED to  
cache only

# Write-back, write-allocate example

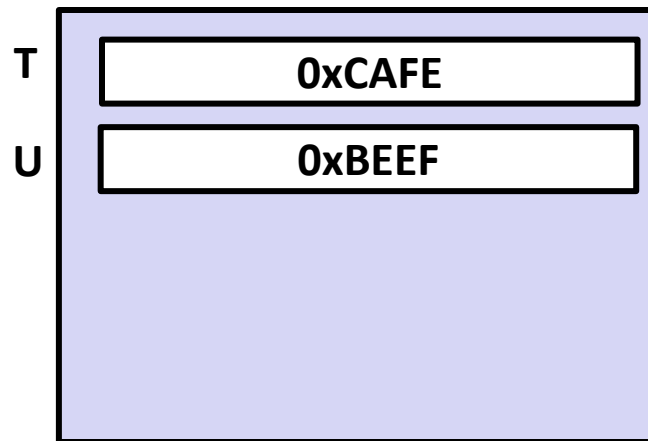
`mov 0xFACE, T`

`mov 0xFEED, T`

`mov U, %rax`



Memory



# Write-back, write-allocate example

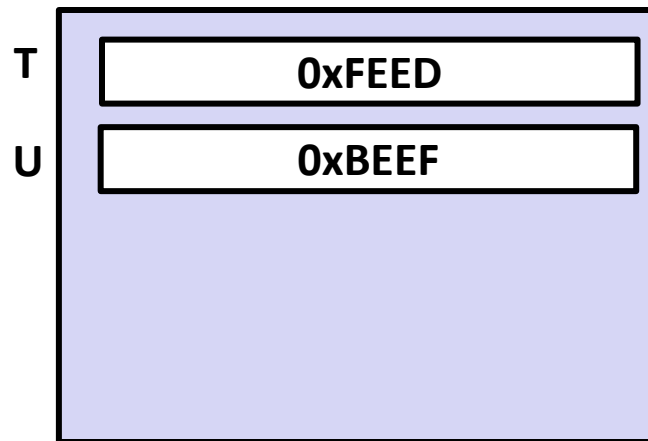
`mov 0xFACE, T`

`mov 0xFEED, T`

`mov U, %rax`



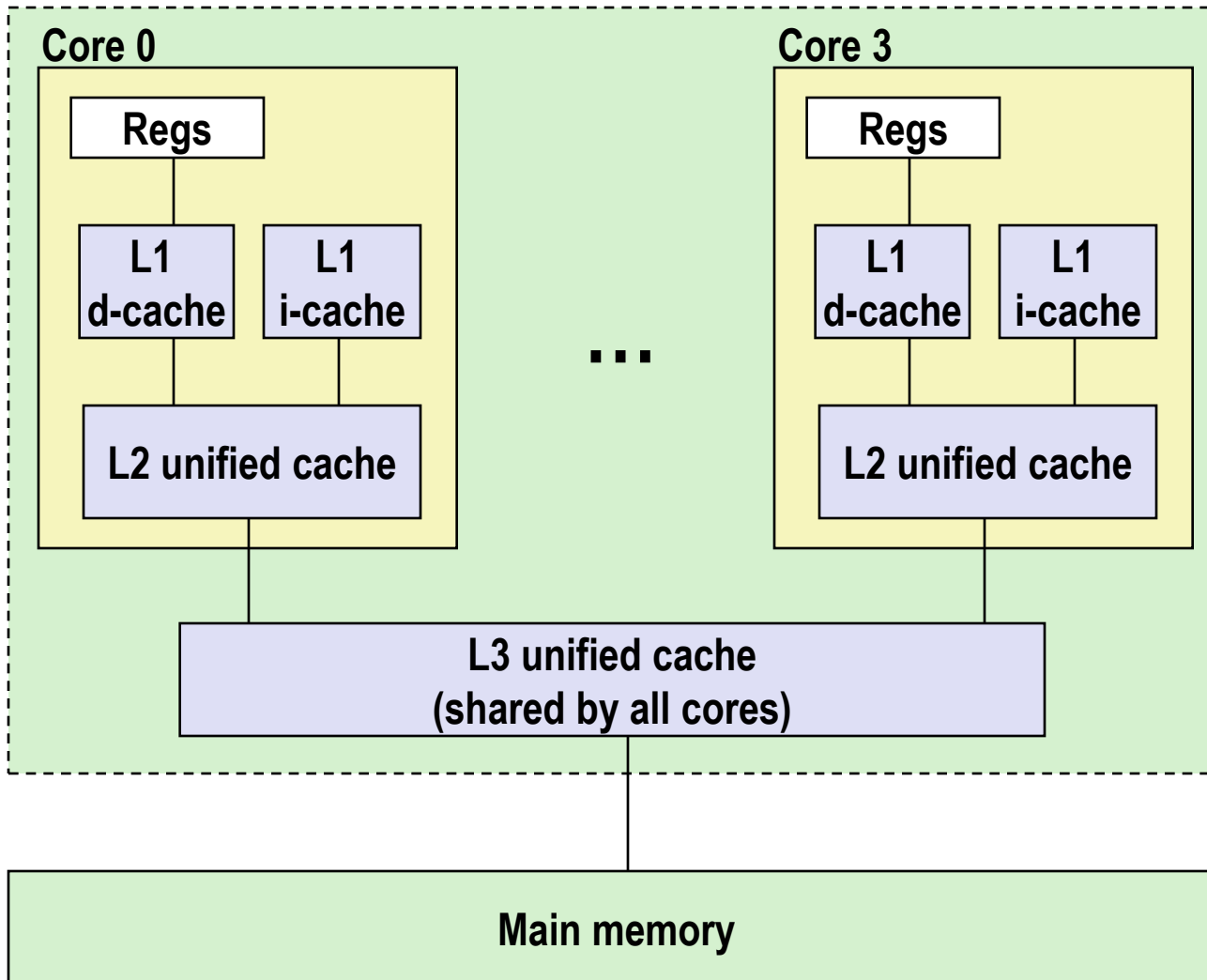
Memory



1. Write T back to memory since it is dirty.
2. Bring U into the cache so we can copy it into %rax

# Back to the Core i7 to look at ways

## Processor package



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

**L2 unified cache:**  
256 KB, 8-way,  
Access: 11 cycles

**L3 unified cache:**  
8 MB, 16-way,  
Access: 30-40 cycles

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

slower, but  
more likely  
to hit

# Where else is caching used?

# Software Caches are More Flexible

## ■ Examples

- File system buffer caches, web browser caches, etc.

## ■ Some design differences

- Almost always fully-associative
  - so, no placement restrictions
  - index structures like hash tables are common (for placement)
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single “block” transfers
  - may fetch or write-back in larger units, opportunistically

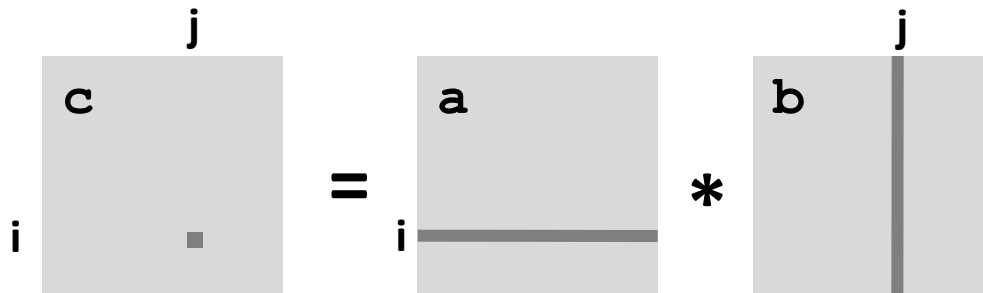
# Optimizations for the Memory Hierarchy

- **Write code that has locality!**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- **How can you achieve locality?**
  - Proper choice of algorithm
  - Loop transformations

# 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];
}
```



$$(AB)_{ij} = \sum_{k=1}^m A_{ik} B_{kj}.$$

memory access pattern?



# Cache Miss Analysis

spatial locality:  
chunks of 8 items in a row  
in same cache line

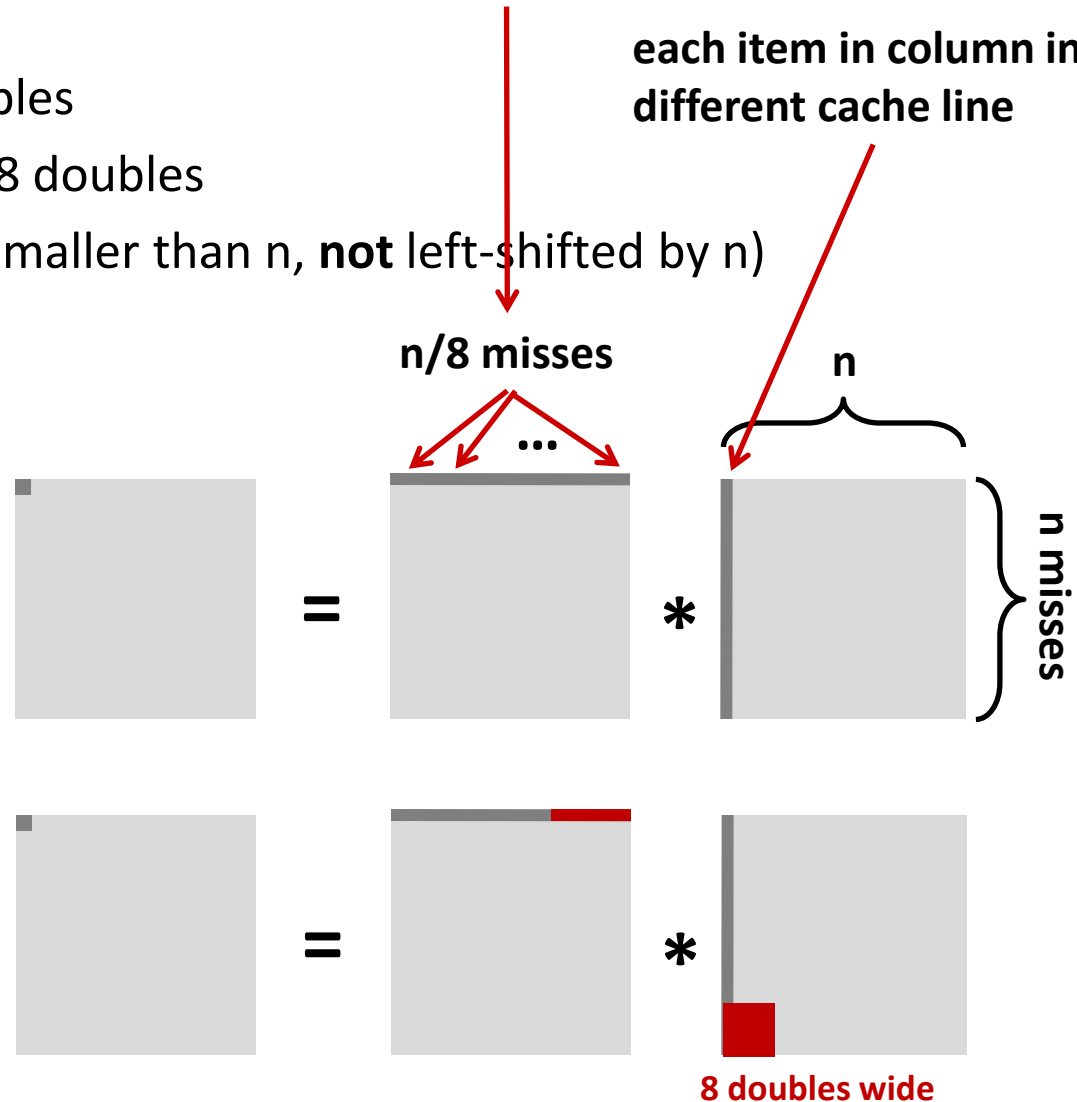
each item in column in  
different cache line

## ■ Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ , **not** left-shifted by  $n$ )

## ■ First iteration:

- $n/8 + n = 9n/8$  misses  
(omitting matrix  $c$ )



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

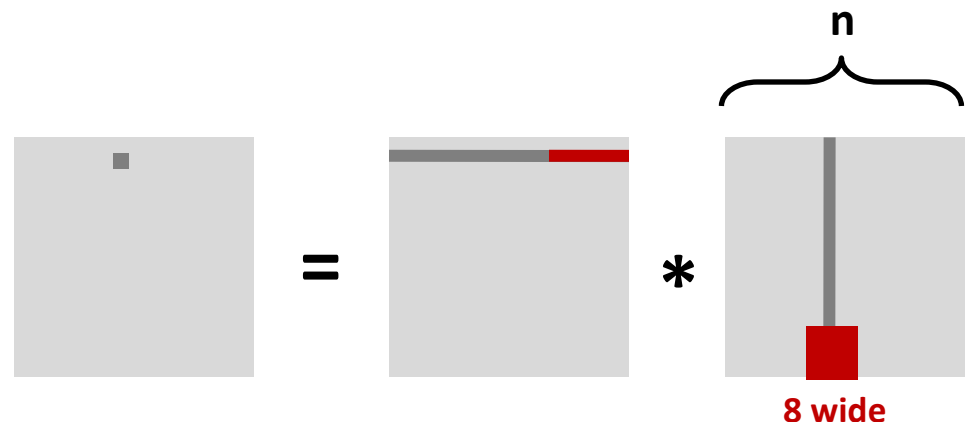
# Cache Miss Analysis

## ■ Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ )

## ■ Other iterations:

- Again:  
 $n/8 + n = 9n/8$  misses  
 (omitting matrix  $c$ )



## ■ Total misses:

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

once per element

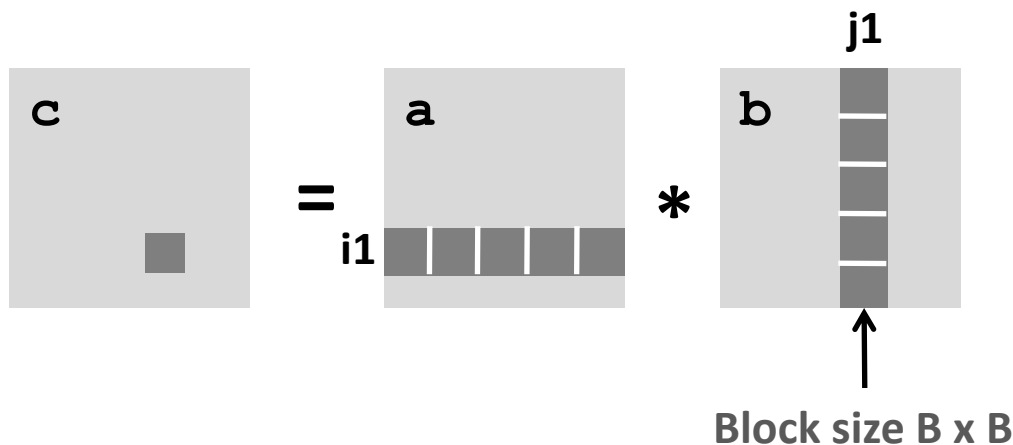
# Blocked Matrix Multiplication

```

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 = 64 bytes = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ )
- Three blocks  fit into cache:  $3B^2 < C$

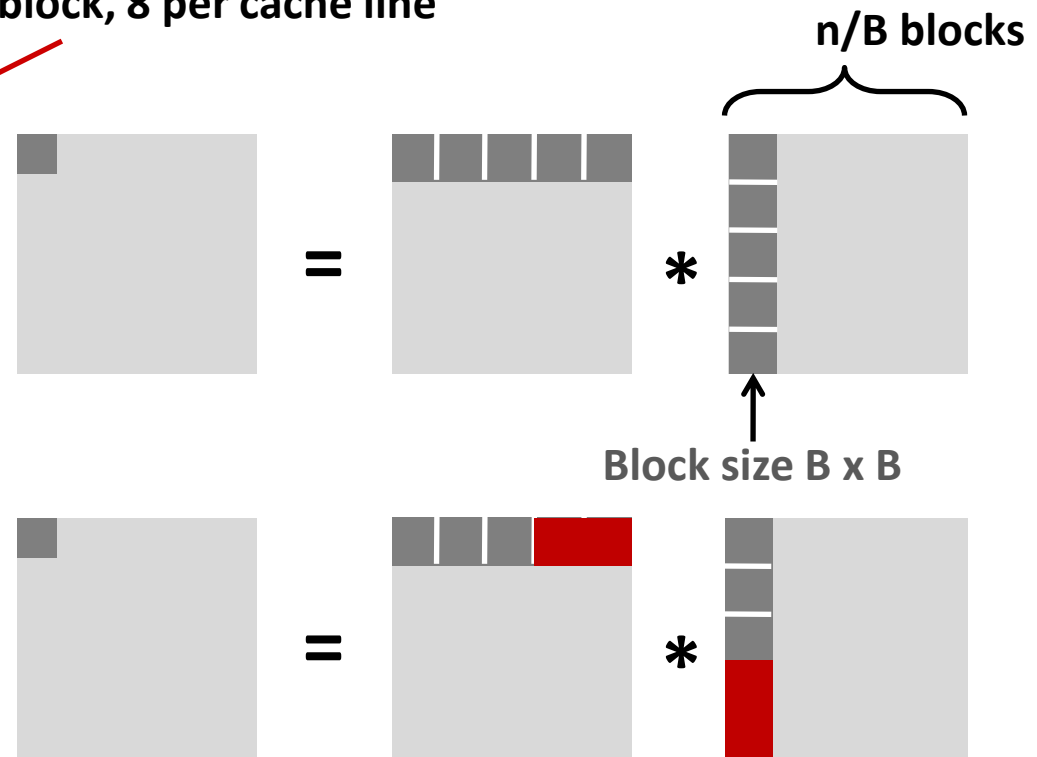
$B^2$  elements per block, 8 per cache line

## ■ First (block) iteration:

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


$n/B$  blocks per row,  
 $n/B$  blocks per column

- Afterwards in cache  
(schematic)



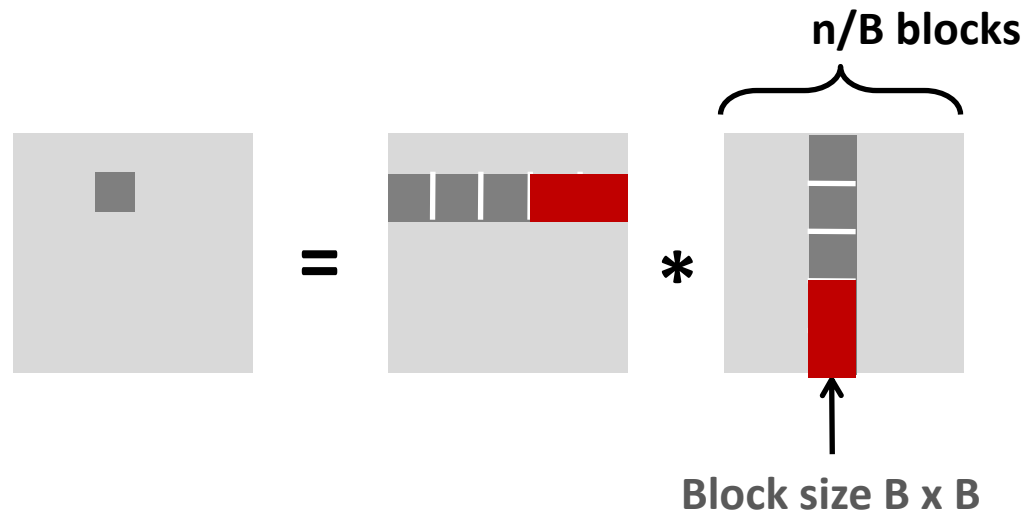
# Cache Miss Analysis

## ■ Assume:

- Cache block = 64 bytes = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ )
- Three blocks  fit into cache:  $3B^2 < C$

## ■ Other (block) iterations:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



## ■ Total misses:

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

# Summary

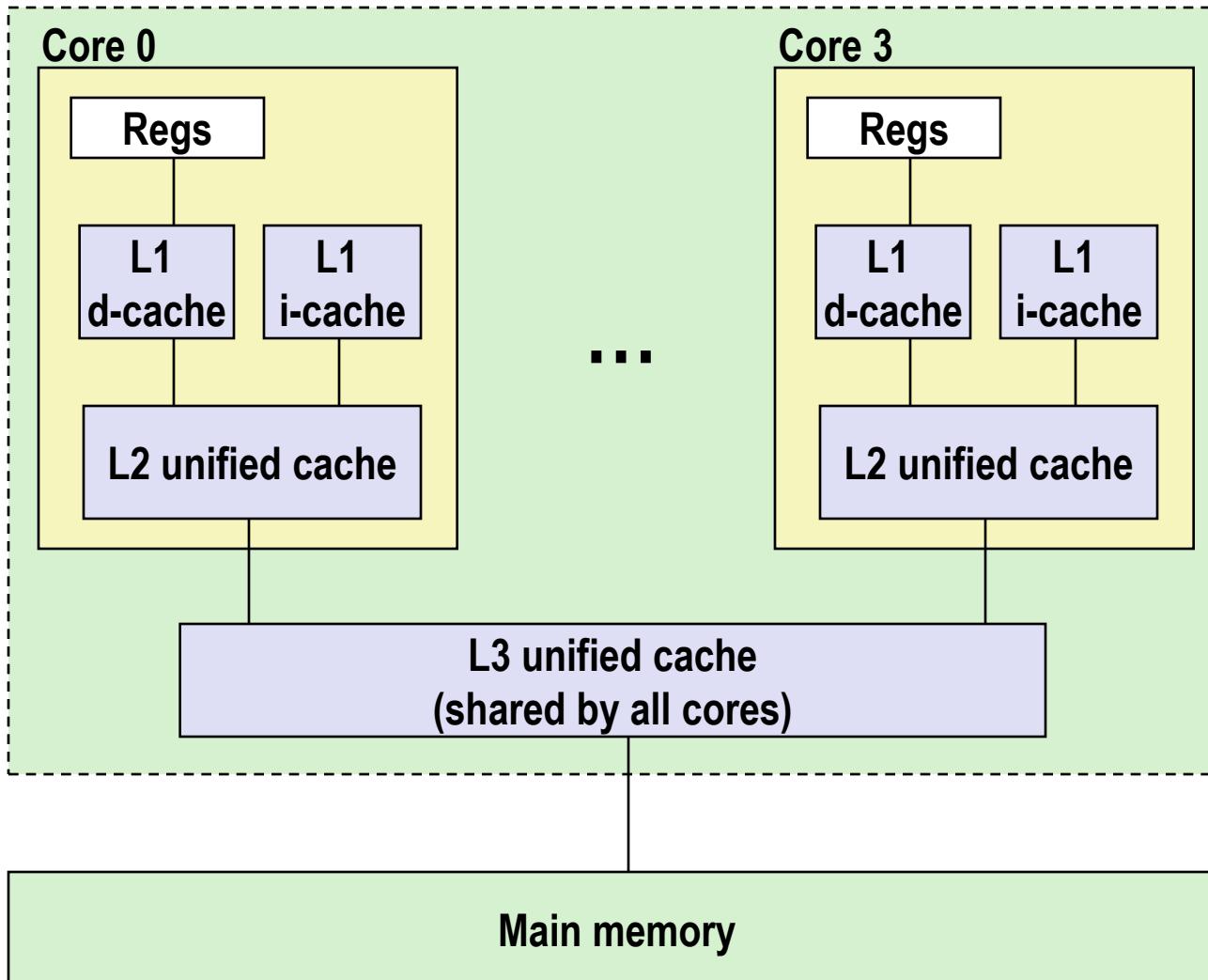
- No blocking:  $(9/8) * n^3$
- Blocking:  $1/(4B) * n^3$
- If  $B = 8$  difference is  $4 * 8 * 9 / 8 = 36x$
- If  $B = 16$  difference is  $4 * 16 * 9 / 8 = 72x$
- Suggests largest possible block size  $B$ , but limit  $3B^2 < C!$
- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data:  $3n^2$ , computation  $2n^3$
    - Every array element used  $O(n)$  times!
  - But program has to be written properly

# Cache-Friendly Code

- **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)
    - Focus on inner loop code

# Intel Core i7 Cache Hierarchy

## Processor package



### L1 i-cache and d-cache:

32 KB, 8-way,  
Access: 4 cycles

### L2 unified cache:

256 KB, 8-way,  
Access: 11 cycles

### L3 unified cache:

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

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



# 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*

