

# Cache Memories

CS2011: Introduction to Computer Systems  
Lecture 12 (6.4, 6.5, 6.6)

# Cache Memories

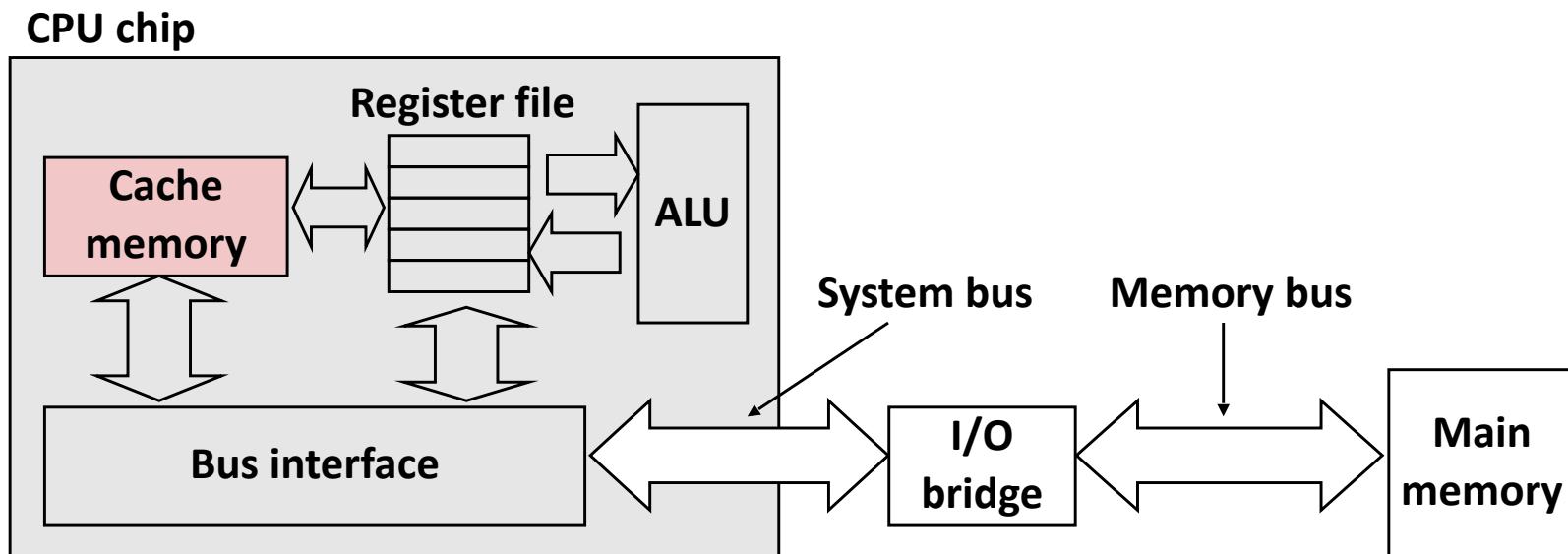
## ■ Cache memory organization and operation

## ■ Performance impact of caches

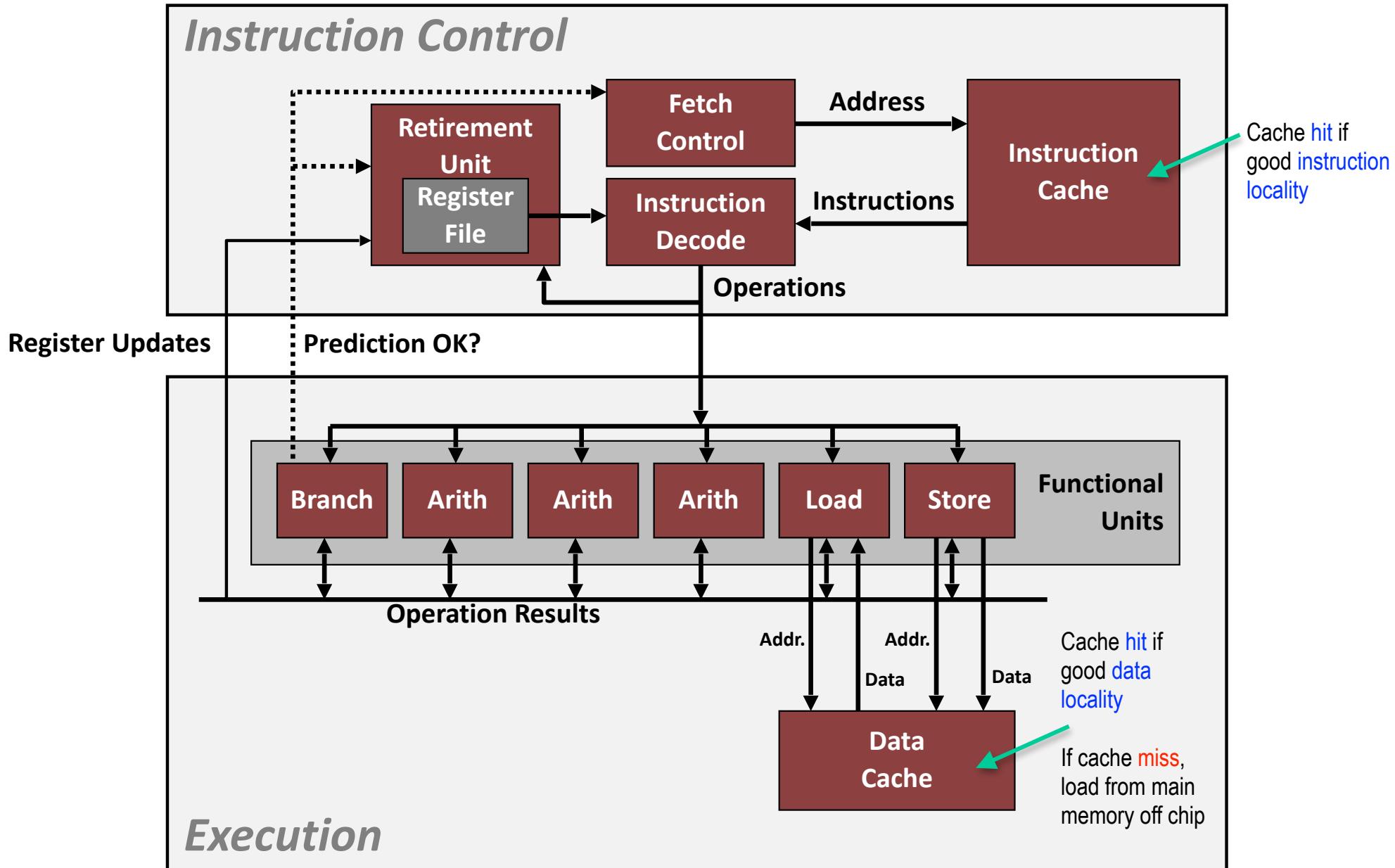
- The memory mountain
- Rearranging loops to improve spatial locality

# Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



# Modern CPU Design (CPU Logic)



# How it Really Looks Like Inside your PC

Desktop PC

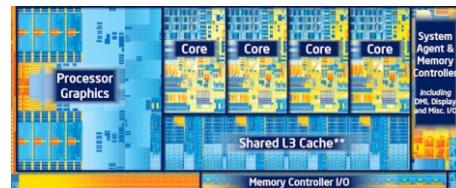


Source: Dell

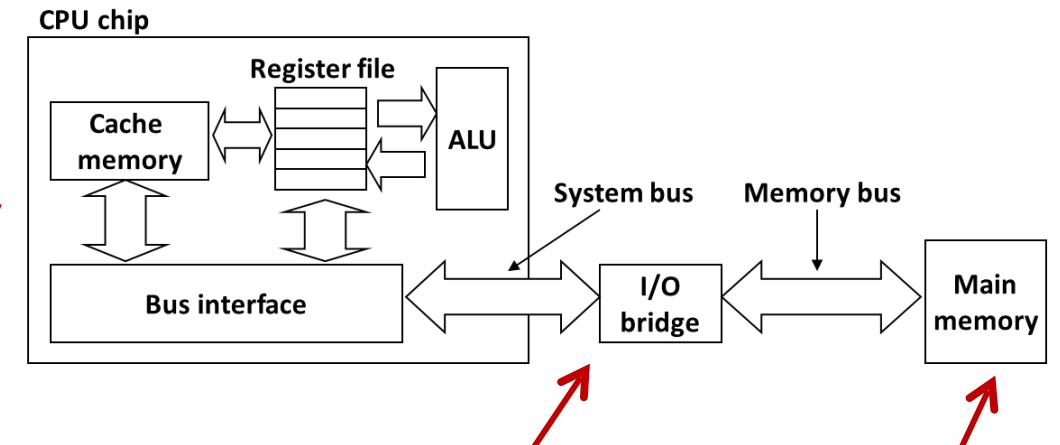
CPU (Intel Core i7)



Source: PC Magazine



Source: techreport.com



Motherboard



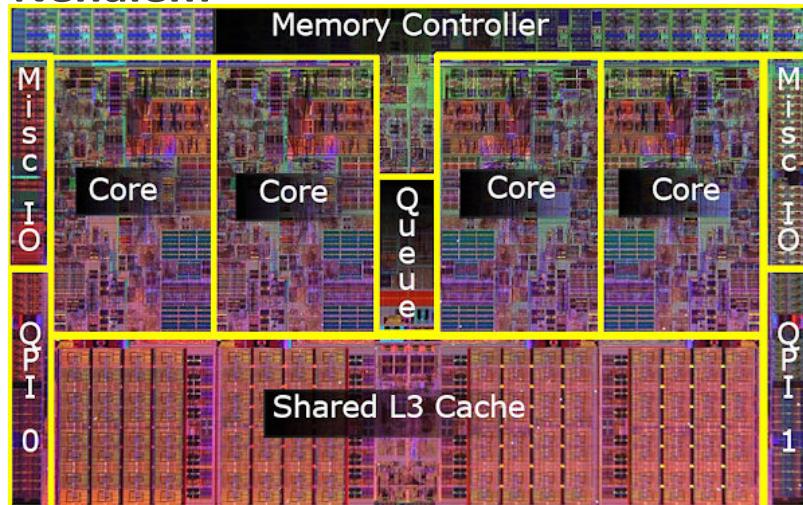
Source: Dell



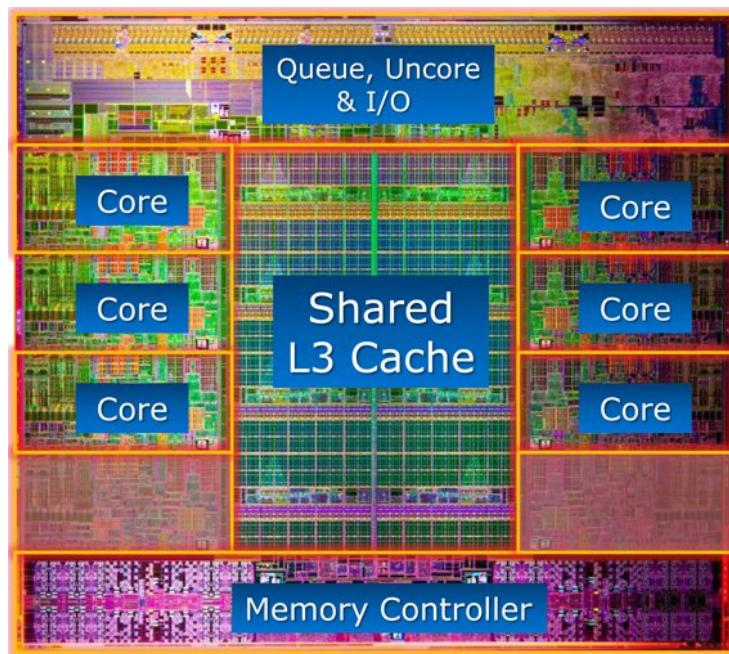
Source: Dell

# Modern CPUs: What it Really Looks Like

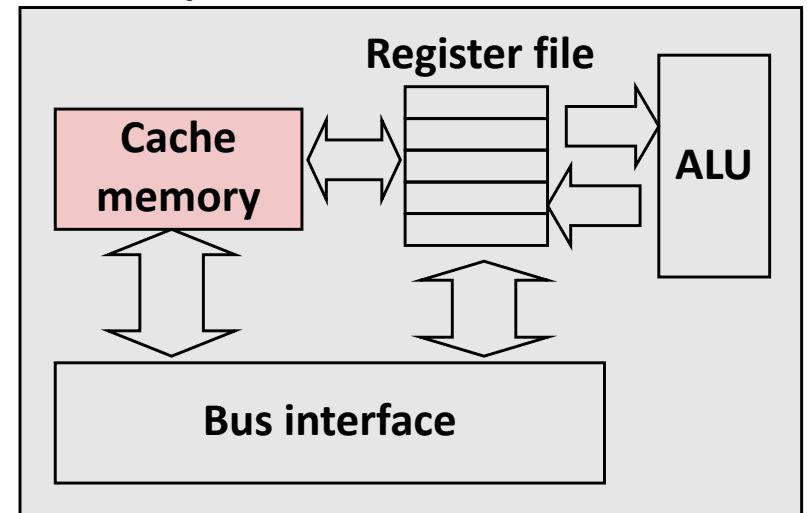
Nehalem



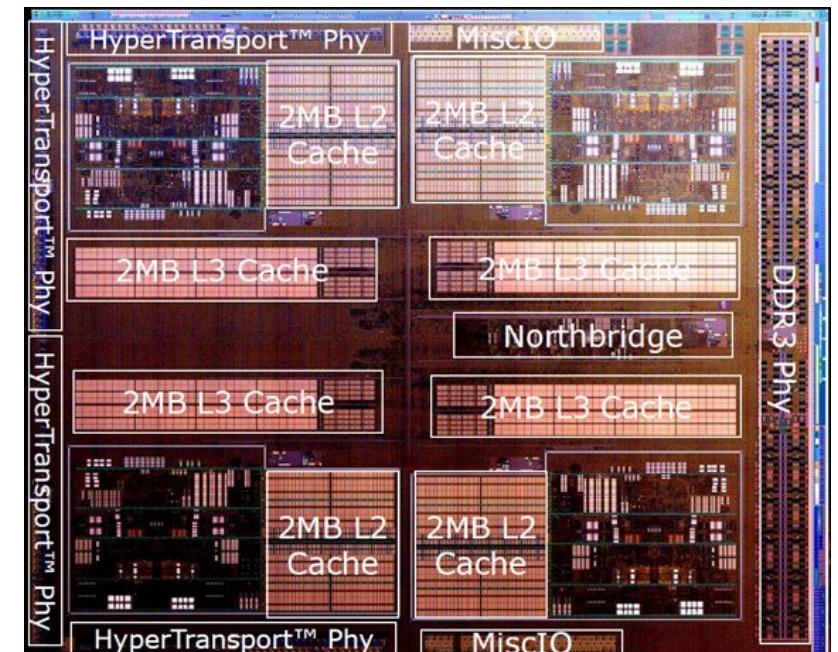
Core i7-3960X



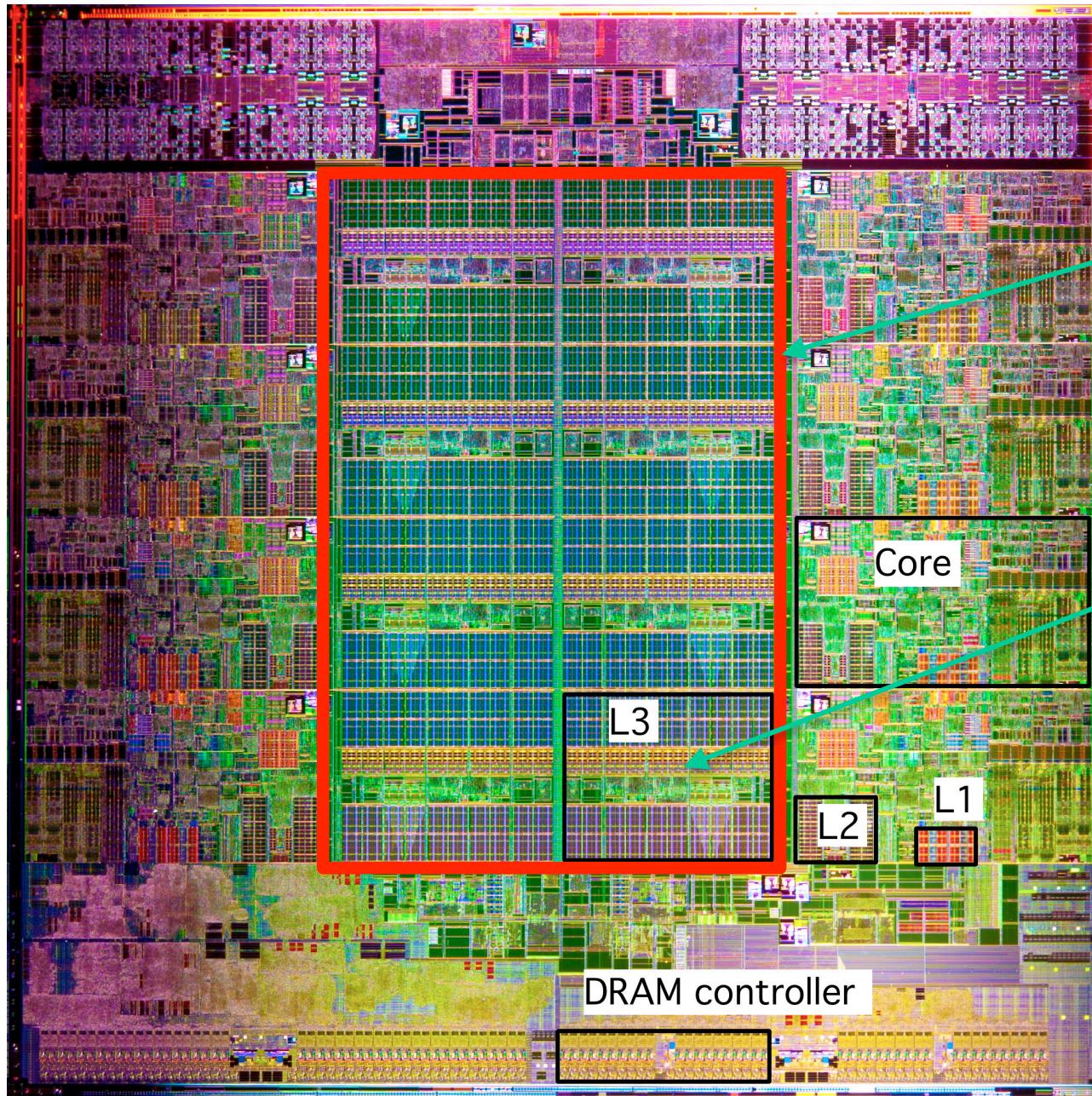
CPU chip



AMD FX 8150



# Modern CPUs: What it Really Looks Like (Cont.)



8 Banks L3 Shared Cache

L3 Cache Bank

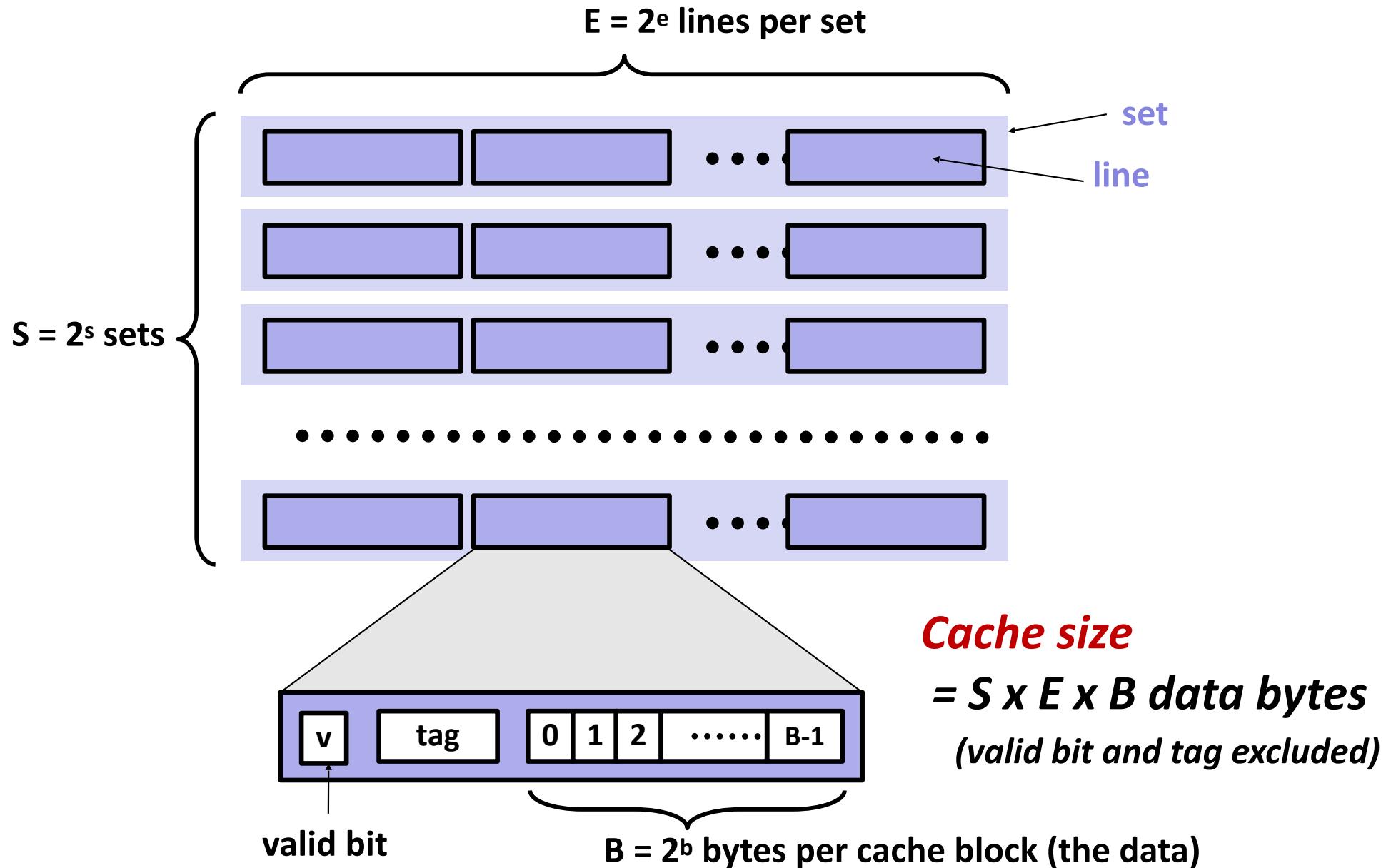
Intel Sandy Bridge  
Processor Die

**L1 Cache:** 32KB Instruction + 32KB Data

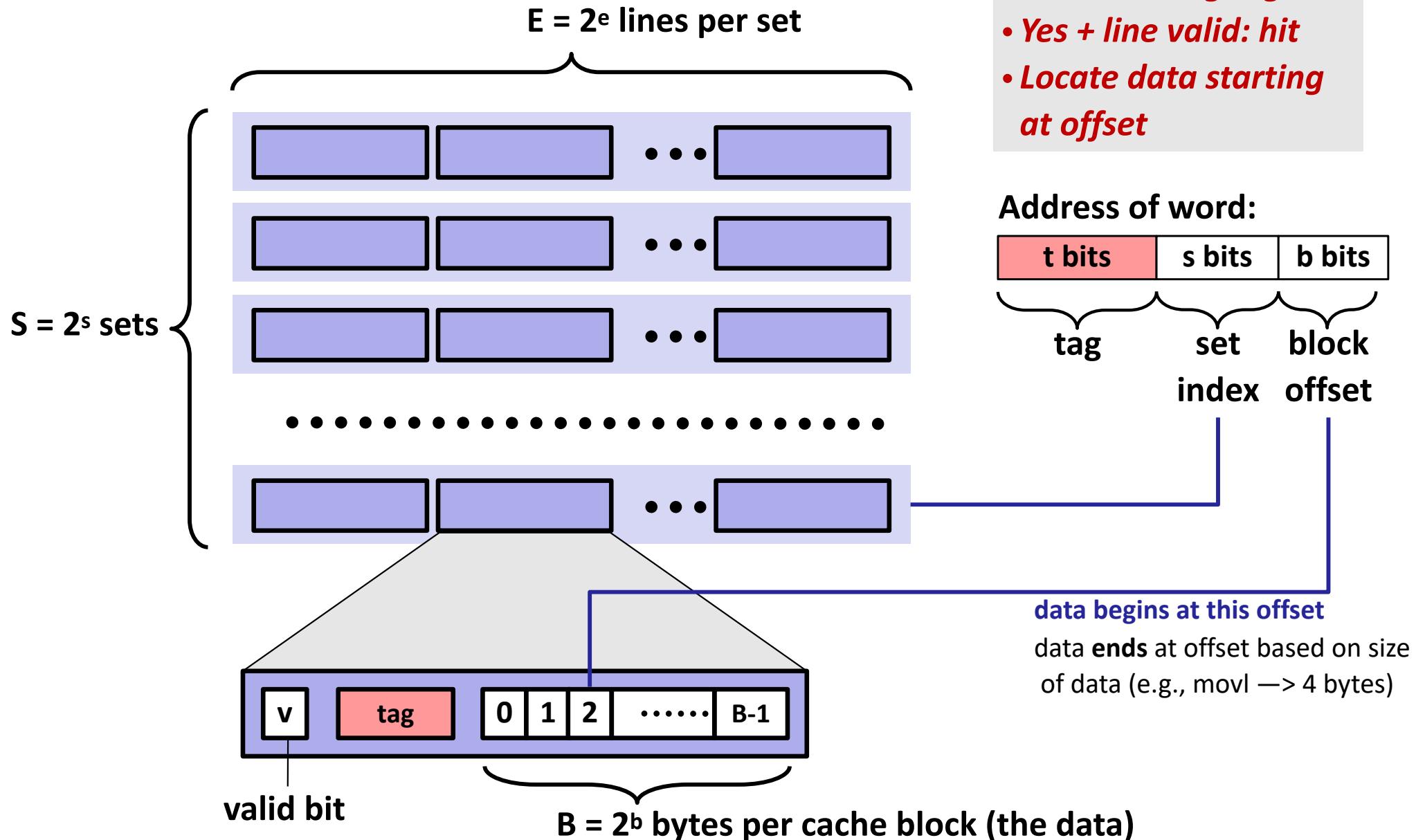
**L2 Cache:** 256KB (Backup for Instr/Data)

**L3 Shared Cache:** 3–20MB

# General Cache Organization (S, E, B)



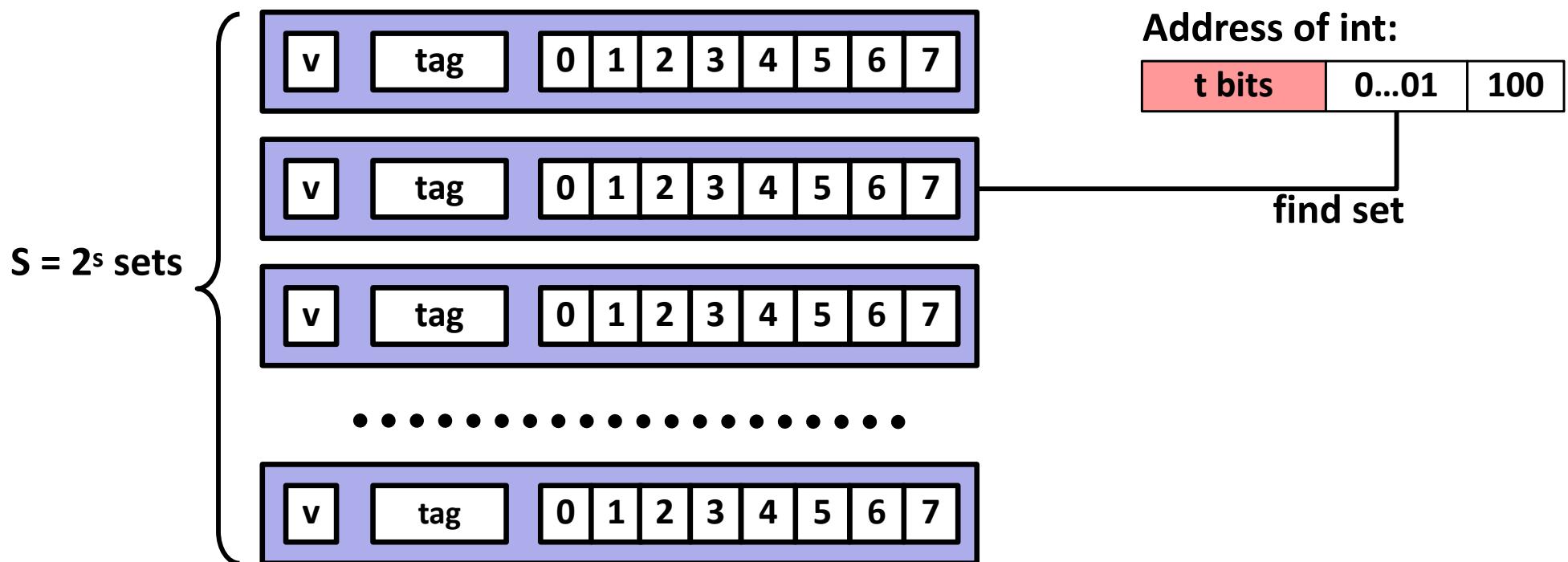
# Cache Read



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

Direct mapped: One line per set

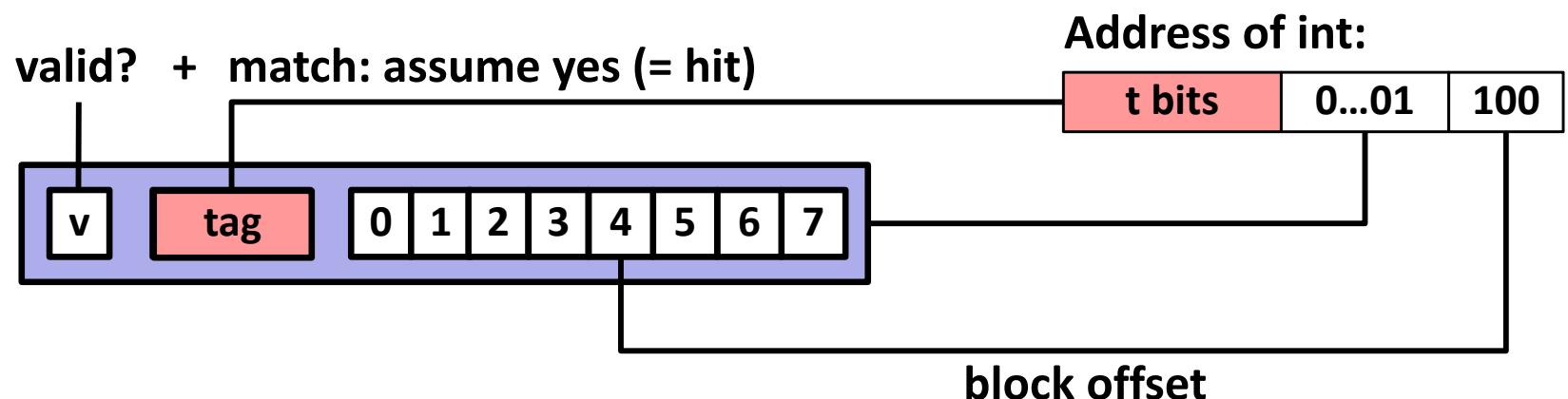
Assume: cache block size  $B=8$  bytes



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

Direct mapped: One line per set

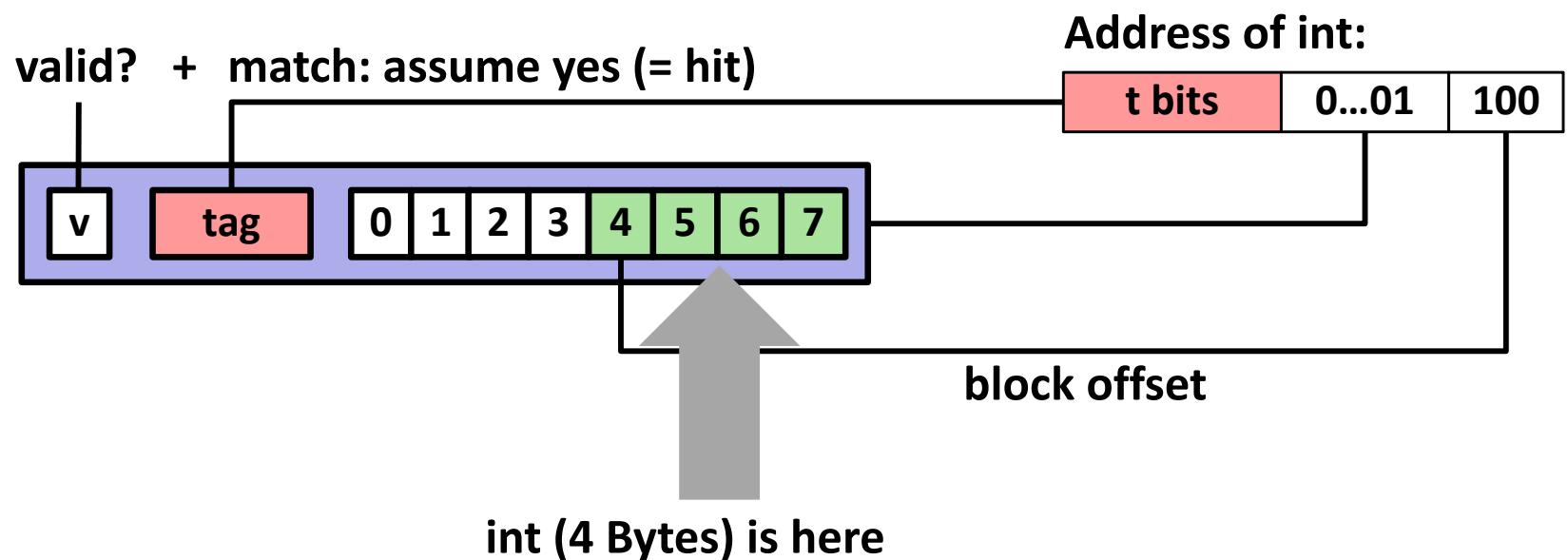
Assume: cache block size  $B=8$  bytes



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

Direct mapped: One line per set

Assume: cache block size  $B=8$  bytes



If tag doesn't match (= miss): old line is evicted and replaced

# Direct-Mapped Cache Simulation

$t=1$     $s=2$     $b=1$



4-bit addresses (address space size  $M=16$  bytes)

$S=4$  sets ( $s = 2$ )

$E=1$  block/line per set ( $e = 0$ )

$B=2$  bytes/block ( $b = 1$ )

Tag bits  $t = m - (b + s) = 4 - (2 + 1) = 1$

Address trace (reads, one byte per read):

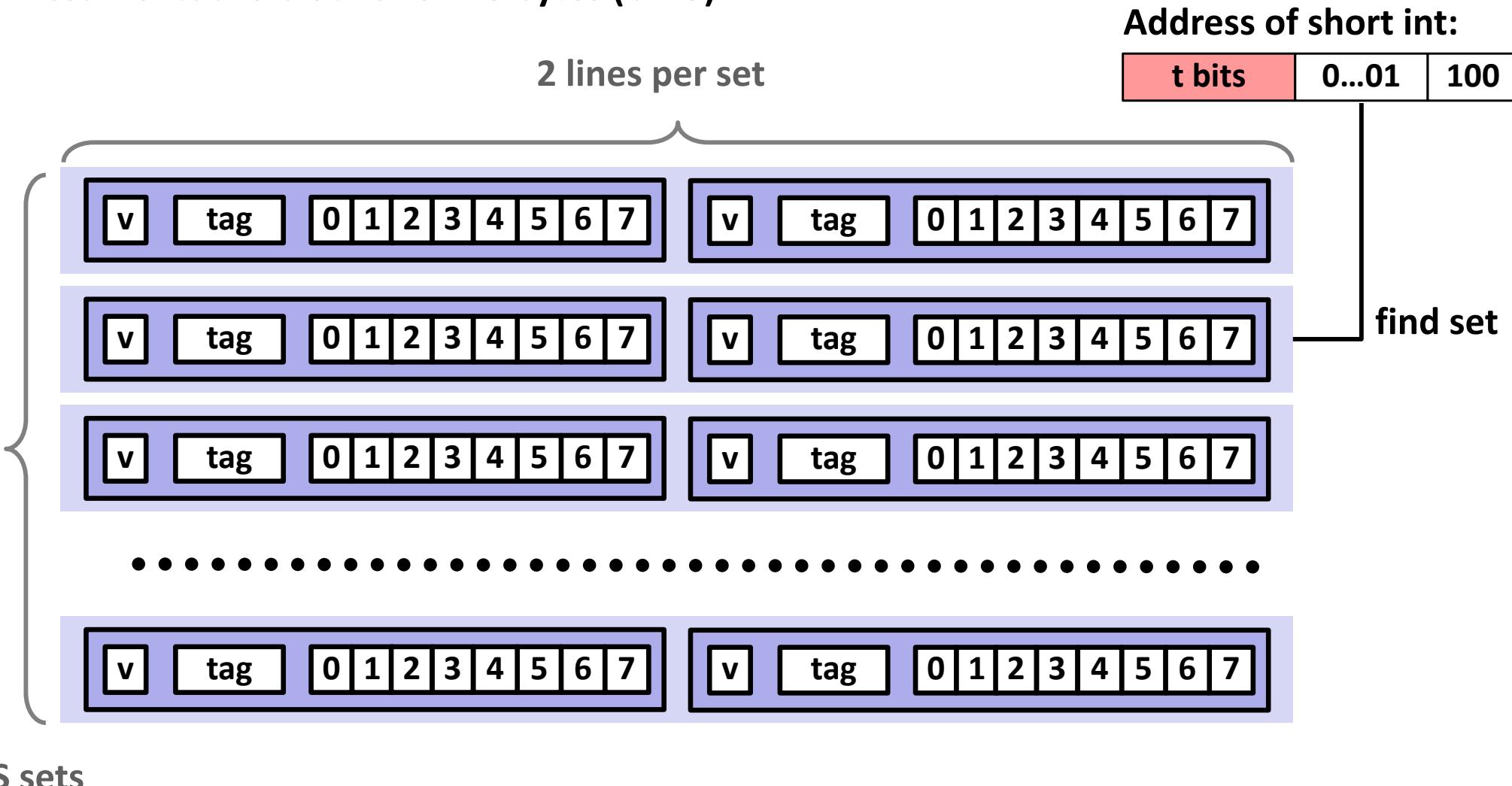
0	$[0000_2]$ ,	miss
1	$[0001_2]$ ,	hit
7	$[0111_2]$ ,	miss
8	$[1000_2]$ ,	miss
0	$[0000_2]$	conflict miss

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

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

$E = 2$ : Two lines/blocks per set

Assume: cache block size  $B=8$  bytes ( $b = 3$ )

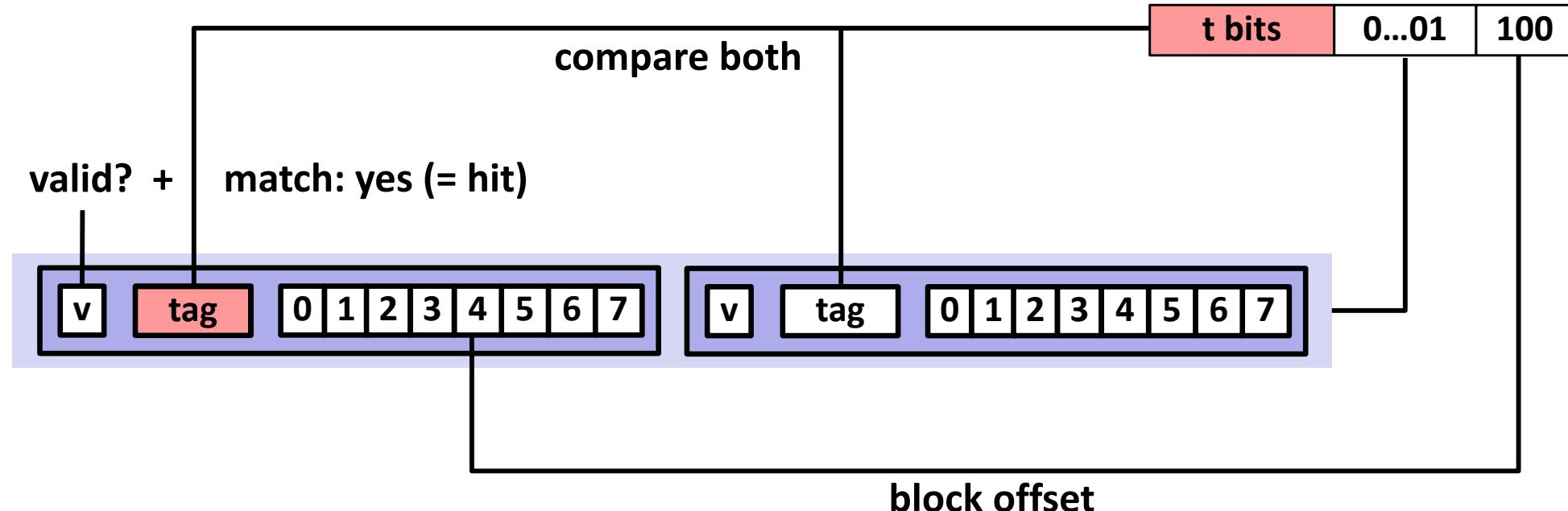


# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines/blocks per set

Assume: cache block size B=8 bytes (b = 3)

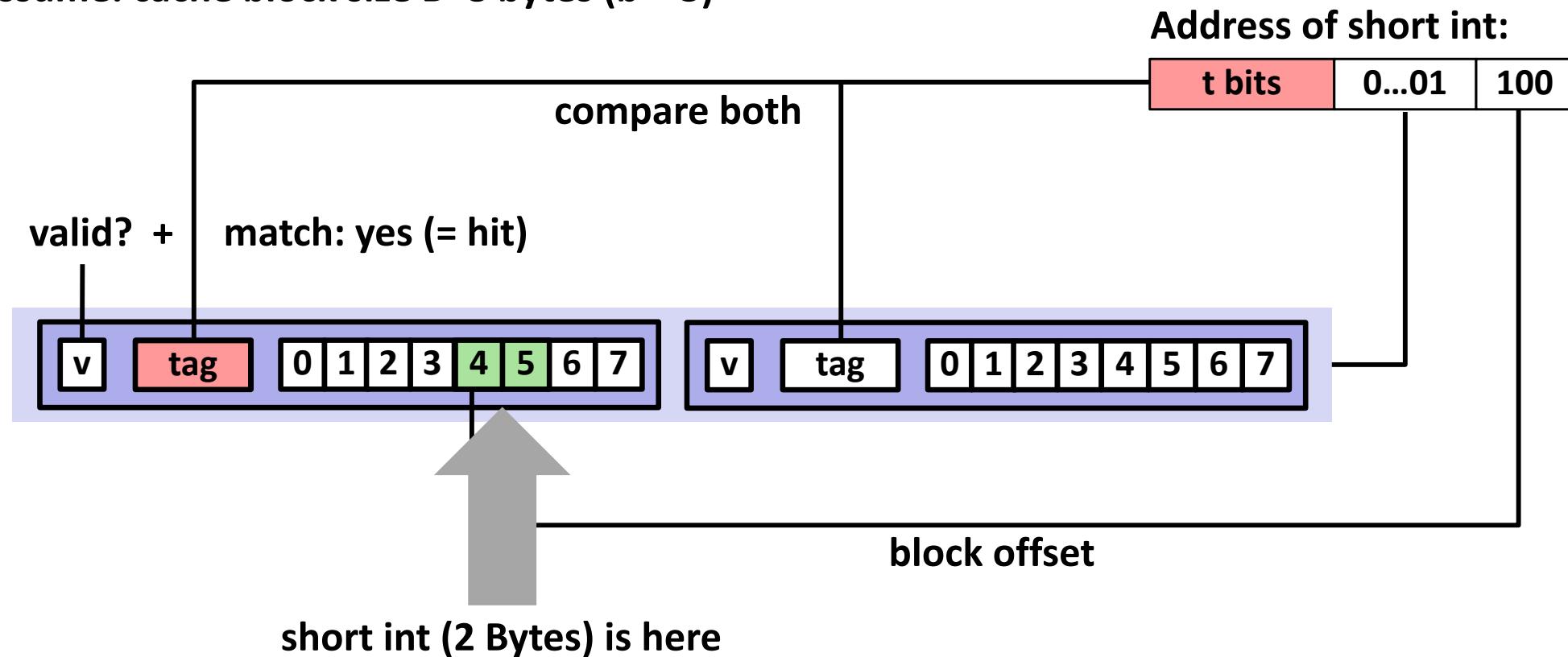
Address of short int:



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

$E = 2$ : Two lines/blocks per set

Assume: cache block size  $B=8$  bytes ( $b = 3$ )



**No match or not valid (= miss):**

- 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
----	---	---

4-bit addresses (address space size  $M=16$  bytes)

S=2 sets ( $s = 1$ )

E=2 blocks/lines per set ( $e = 1$ )

B=2 bytes/block ( $b = 1$ )

Tag bits  $t = m - (b + s) = 4 - (1 + 1) = 2$

Address trace (reads, one byte per read):

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

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

# What about writes?

## ■ Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

## ■ What to do on a write-hit?

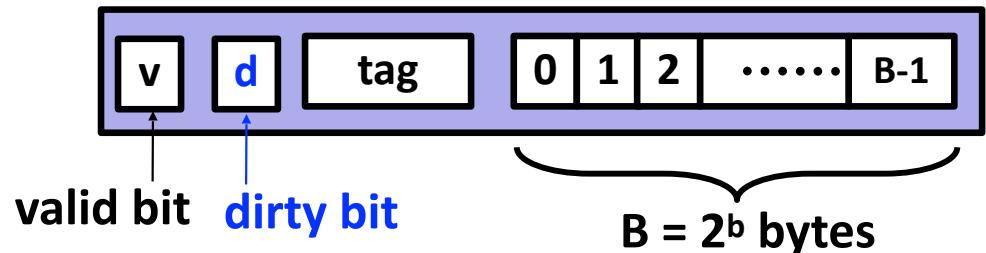
- Write-through (write immediately to memory) - **very slow**
- Write-back (defer write to memory until replacement of line)
  - Each cache line needs a **dirty bit** (set if data has been written to)

## ■ What to do on a write-miss?

- Write-allocate (load into cache, update line in cache)
  - Good if more writes to the location will follow
- No-write-allocate (writes **straight to memory**, does not load into cache)

## ■ Typical

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

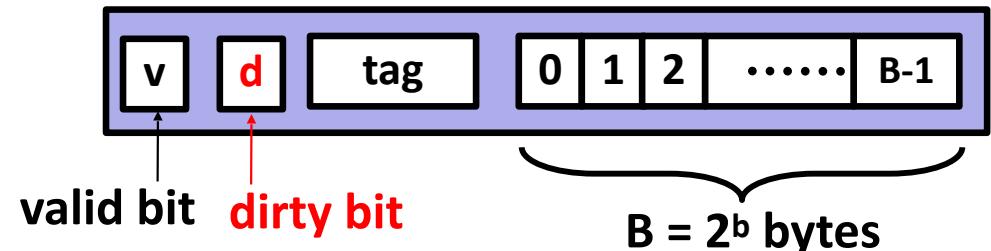


# Practical Write-back Write-allocate

## ■ A write to address X is issued

## ■ If it is a **hit**

- Update the contents of block
- Set **dirty bit to 1** (bit is sticky and only cleared on eviction)



## ■ If it is a **miss**

- Fetch block from memory (per a read miss)
- Then perform the write operations (per a write hit)

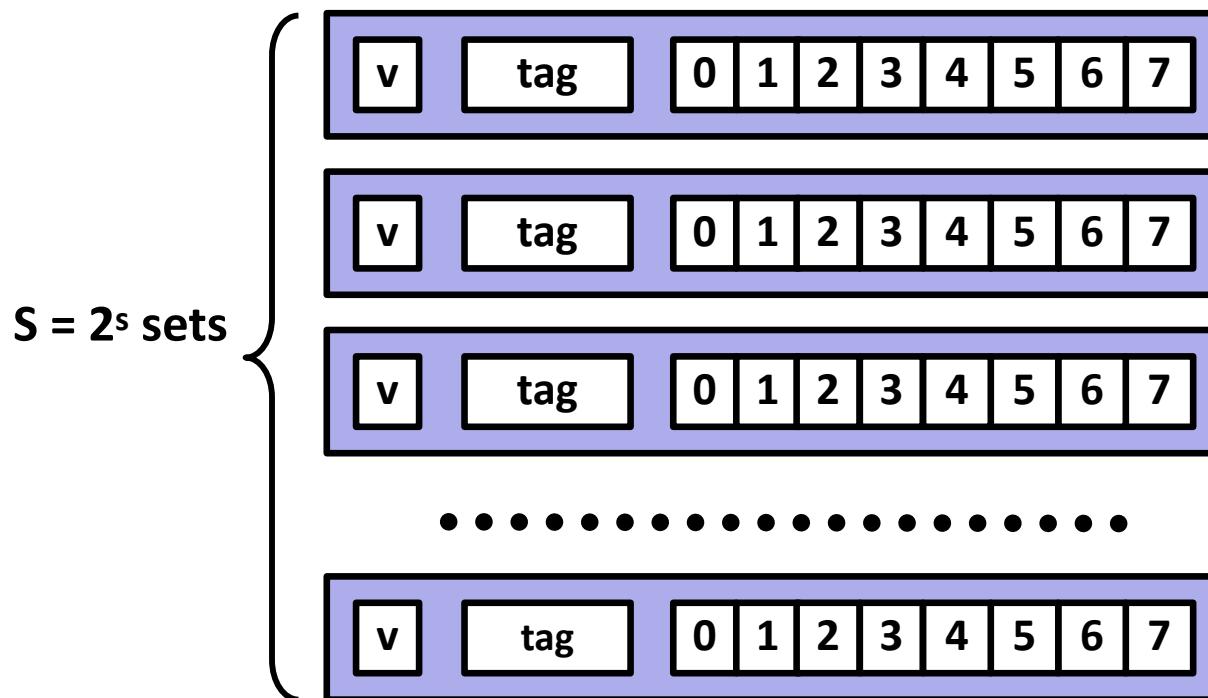
## ■ If a line is evicted and **dirty bit** is set to **1**

- 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

# Why Index Using Middle Bits?

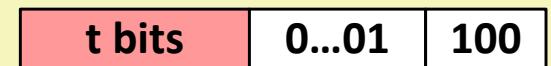
Direct mapped: One line per set

Assume: cache block size 8 bytes



## Standard Method: Middle bit indexing

Address of int:



find set

## Alternative Method: High bit indexing

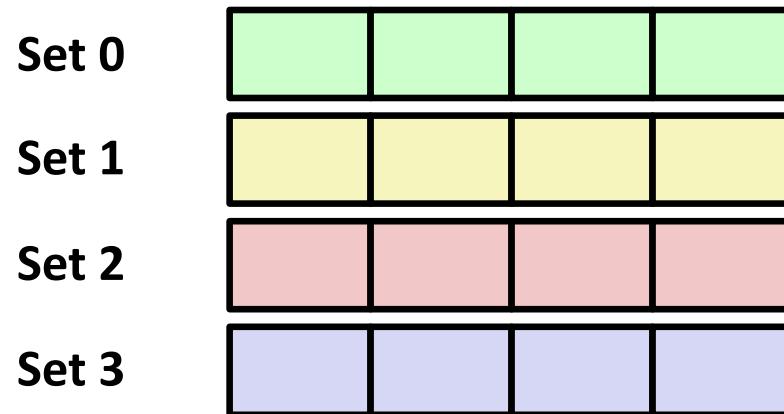
Address of int:



find set

# Illustration of Indexing Approaches

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



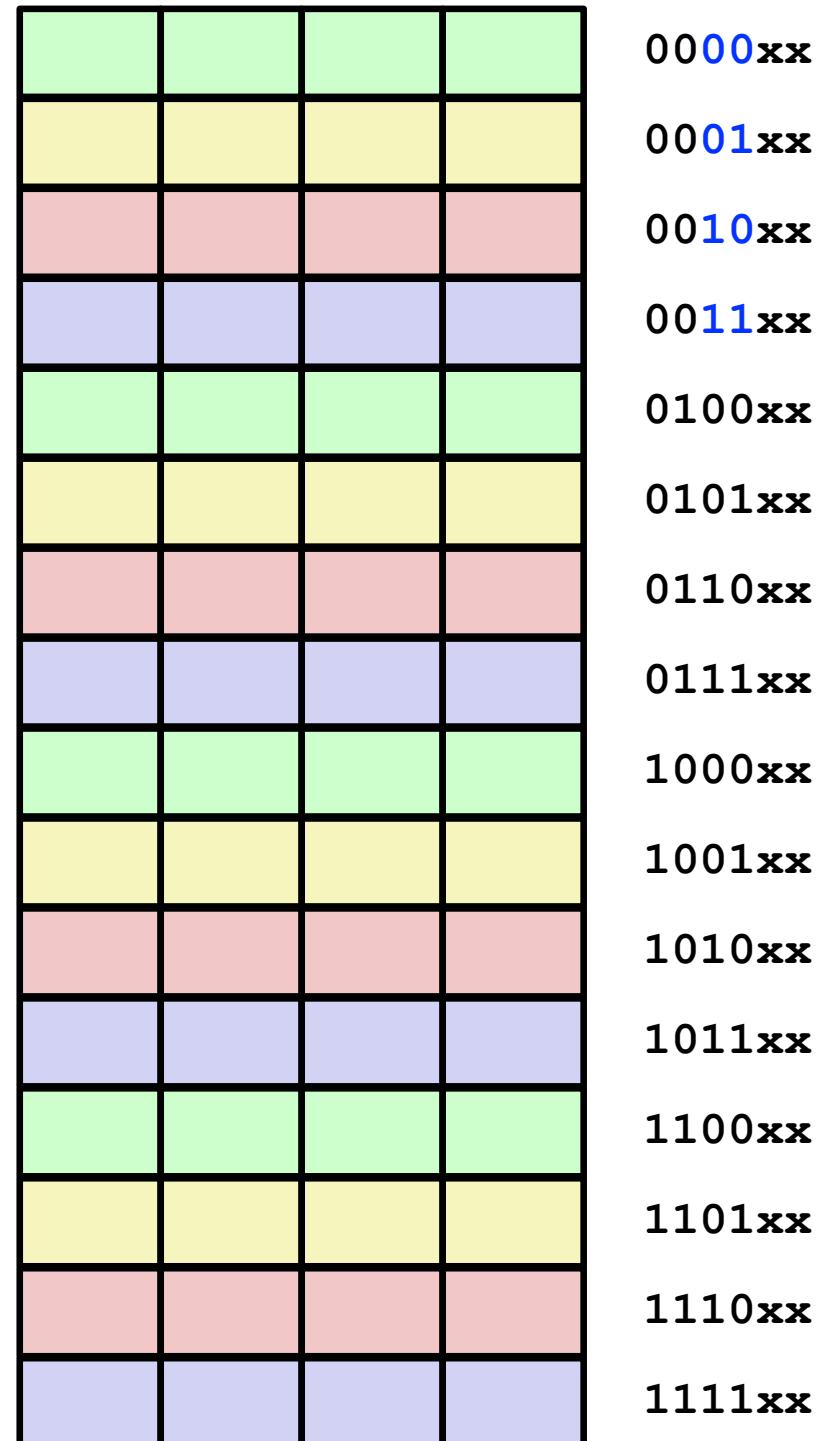
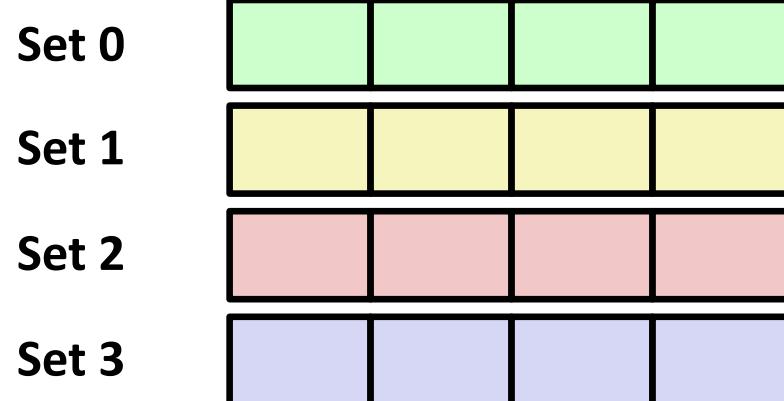
				0000xx
				0001xx
				0010xx
				0011xx
				0100xx
				0101xx
				0110xx
				0111xx
				1000xx
				1001xx
				1010xx
				1011xx
				1100xx
				1101xx
				1110xx
				1111xx

# Middle Bit Indexing

## ■ Addresses of form **TTSSBB**

- **TT** Tag bits
- **SS** Set index bits
- **BB** Offset bits

## ■ Makes good use of spatial locality

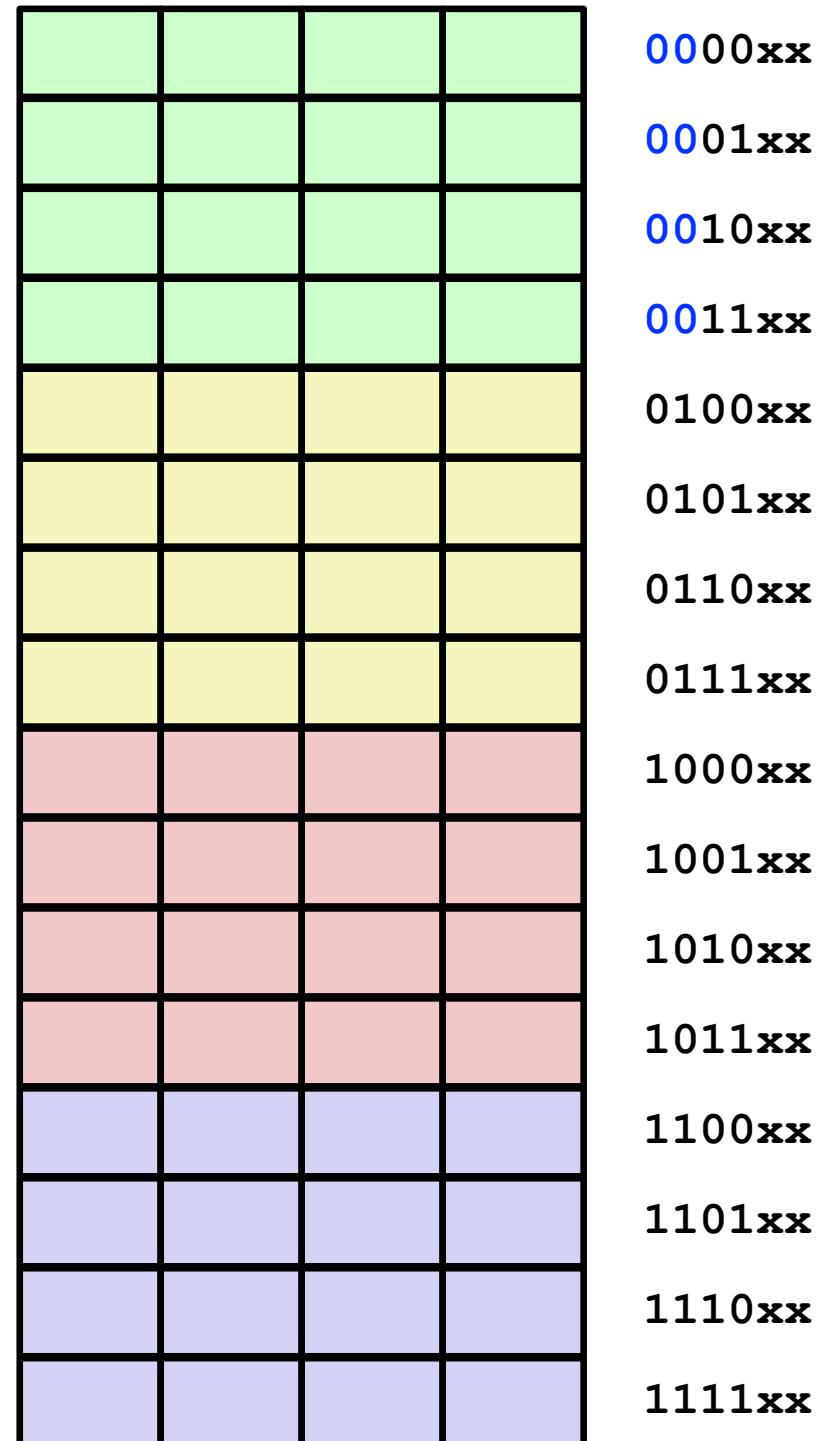
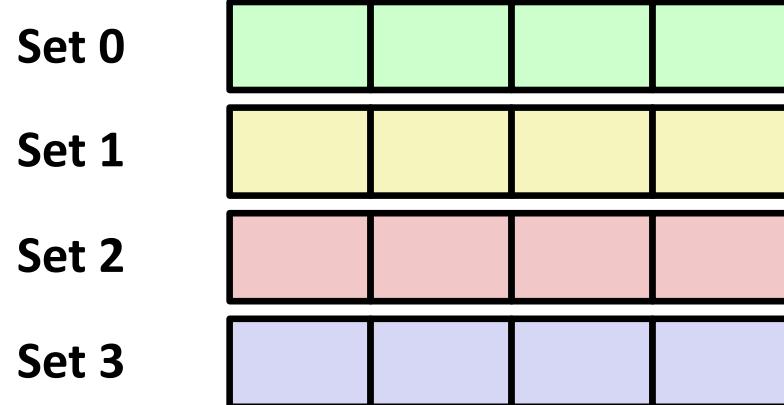


# High Bit Indexing

## ■ Addresses of form **SSTTBB**

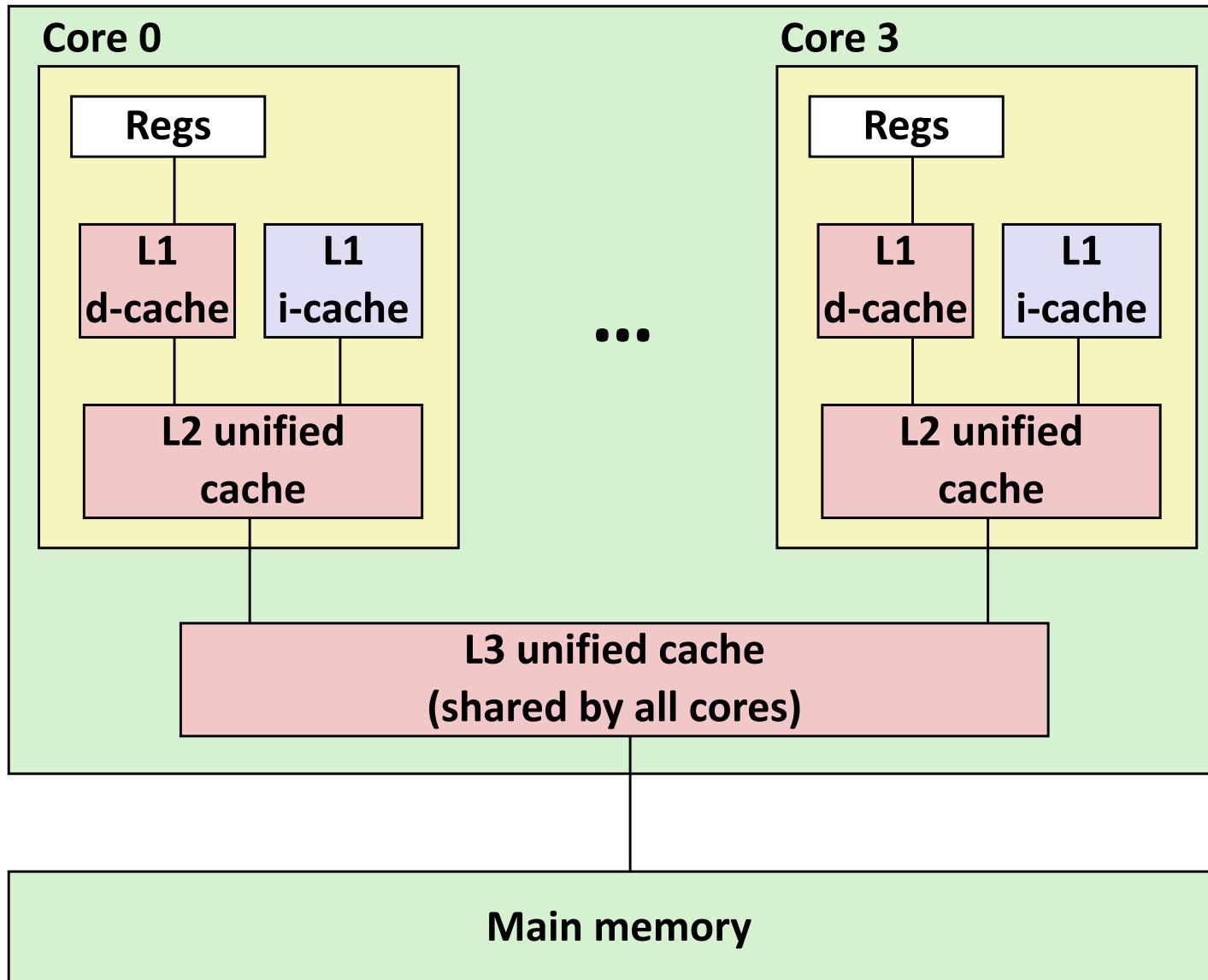
- **SS** Set index bits
- **TT** Tag bits
- **BB** Offset bits

■ Program with **high spatial locality** would generate **lots of conflicts**



# Intel Core i7 Cache Hierarchy

Processor package



**L1 i-cache and d-cache:**

32 KB, 8-way,  
Access: **4 cycles**

**L2 unified cache:**

256 KB, 8-way,  
Access: **10 cycles**

**L3 unified cache:**

8 MB, 16-way,  
Access: **40-75 cycles**

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

# Example: Core i7 L1 Data Cache

32 kB 8-way set associative

64 bytes/block

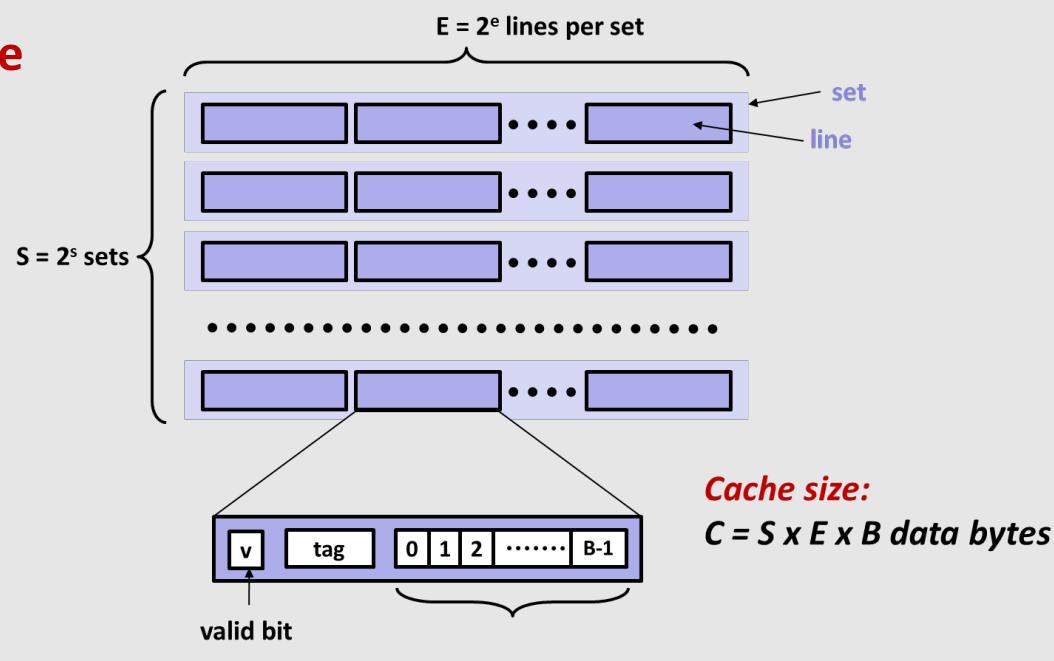
47 bit address range

$B =$

$S = \text{?}, s = \text{?}$

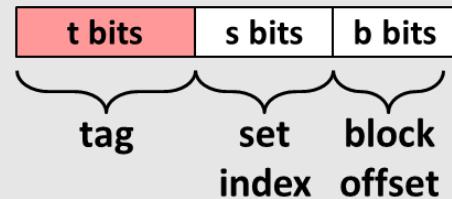
$E = \text{?}, e = \text{?}$

$C = \text{?}$



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: ? bits

Set index: ? bits

Tag: ? bits

Stack Address:

0x00007f7262a1e010

Block offset:

0x??

Set index:

0x??

Tag:

0x??

# Example: Core i7 L1 Data Cache

**32 kB 8-way set associative**

**64 bytes/block**

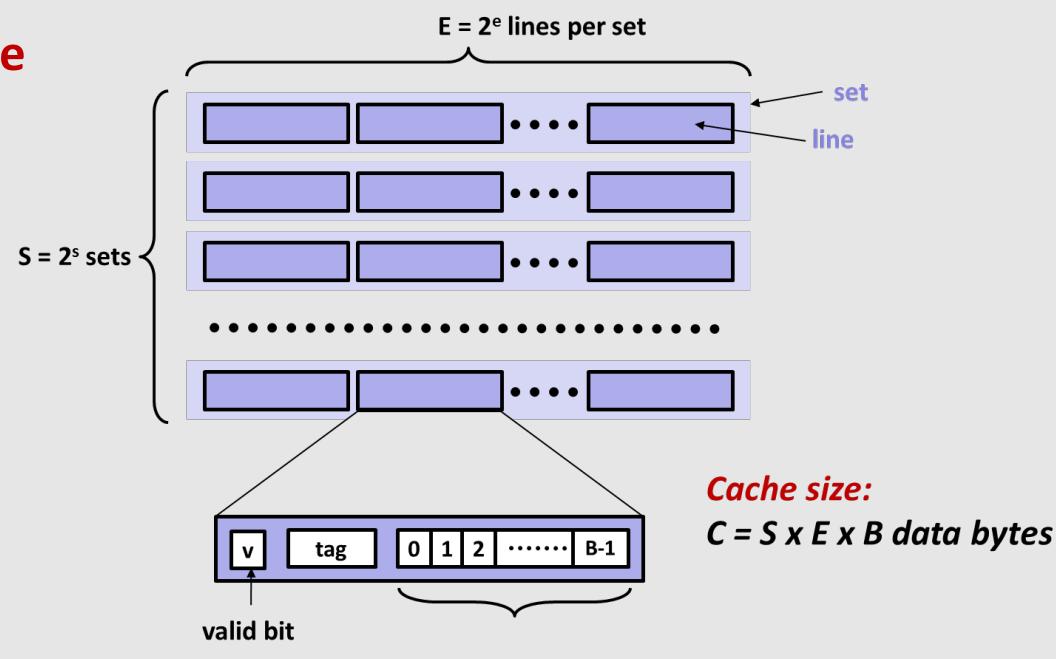
**47 bit address range**

**B = 64**

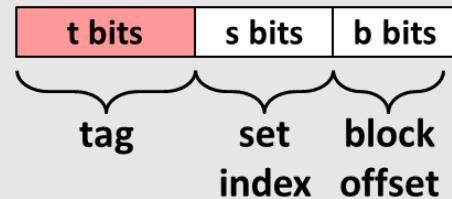
**S = 64, s = 6**

**E = 8, e = 3**

**C =  $64 \times 64 \times 8 = 32,768$**



**Address of word:**



**Block offset:** 6 bits

**Set index:** 6 bits

**Tag:** 35 bits

**Stack Address:**

**0x00007f7262a1e010**

**Block offset:**

**0x??**

**Set index:**

**0x??**

**Tag:**

**0x??**

# Example: Core i7 L1 Data Cache

**32 kB 8-way set associative**

**64 bytes/block**

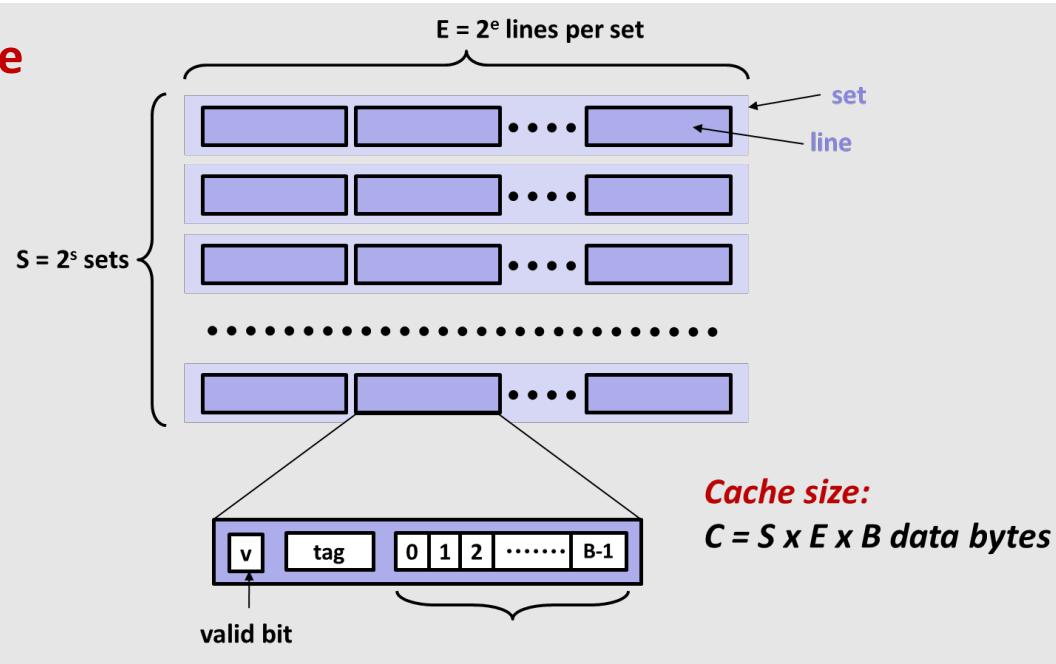
**47 bit address range**

**B = 64**

**S = 64, s = 6**

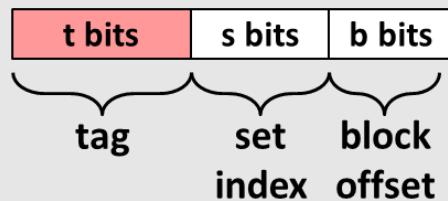
**E = 8, e = 3**

**C =  $64 \times 64 \times 8 = 32,768$**



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

**Address of word:**



**Block offset: 6 bits**

**Set index: 6 bits**

**Tag: 35 bits**

**Stack Address:**

**0x00007f7262a1e010**

**Block offset:**

**0x??**

**Set index:**

**0x??**

**Tag:**

**0x??**

# Example: Core i7 L1 Data Cache

**32 kB 8-way set associative**

**64 bytes/block**

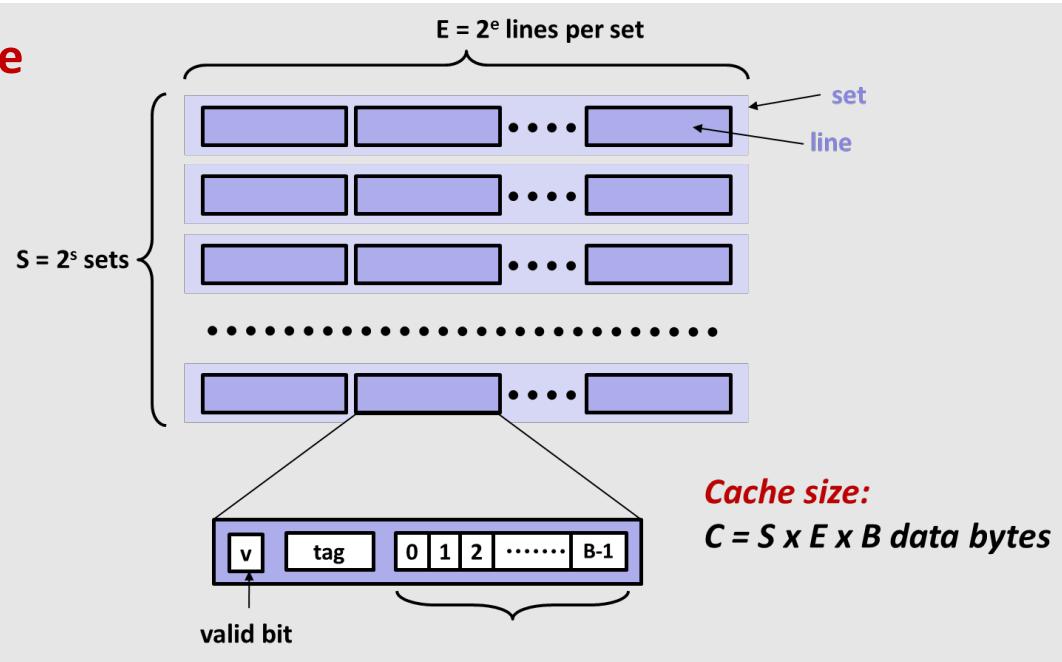
**47 bit address range**

**B = 64**

**S = 64, s = 6**

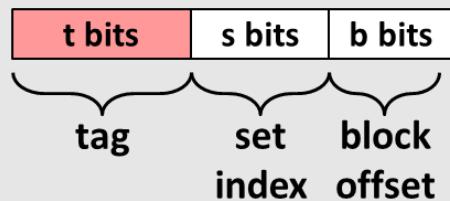
**E = 8, e = 3**

**C =  $64 \times 64 \times 8 = 32,768$**



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

**Address of word:**



**Block offset: 6 bits**

**Set index: 6 bits**

**Tag: 35 bits**

**Stack Address:**

**0x00007f7262a1e010**

0000 0001 0000  
Binary

**Block offset:**

**0x10**

**Set index:**

**0x0**

**Tag:** 0x7f7262a1e

# Cache Performance Metrics

## ■ Miss Rate

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

## ■ Hit Time

- *Time to deliver a line in the cache to the processor*
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - **4 clock cycles** for L1
  - **10 clock cycles** for L2

## ■ Miss Penalty

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

# Let's think about those numbers

## ■ Huge difference between a hit and a miss

- Could be 100x, if just L1 and main memory

## ■ Would you believe 99% hits is twice as good as 97%?

- Consider this simplified example:

cache hit time of 1 cycle

miss penalty of 100 cycles

- Average access time:

97% hits: 1 cycle +  $0.03 \times 100$  cycles = **4 cycles**

99% hits: 1 cycle +  $0.01 \times 100$  cycles = **2 cycles (twice as fast)**

## ■ This is why “miss rate” is used instead of “hit rate”

# Writing Cache Friendly Code

## ■ Make the common case go fast

- Focus on the inner loops of the core functions

## ■ Minimize the misses in the inner loops

- Repeated references to variables are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)

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

# Cache Memories

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

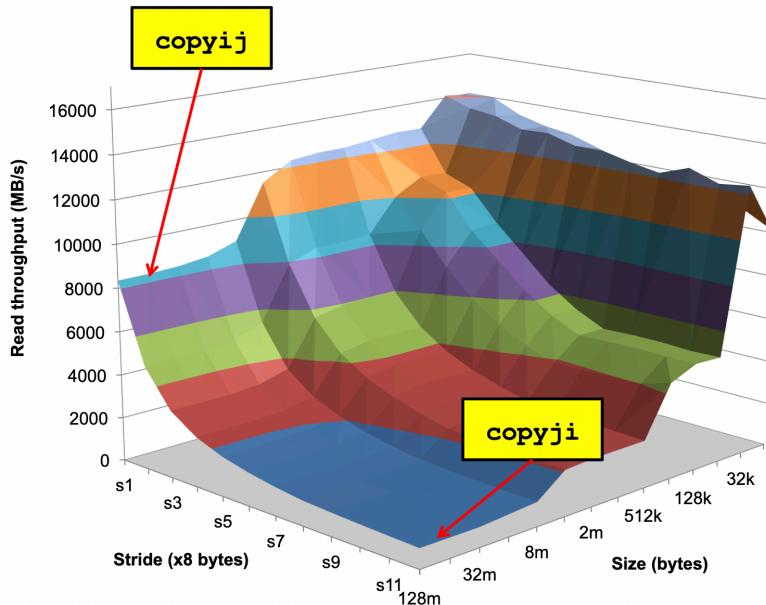
# The Memory Mountain

## ■ Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)

## ■ Memory mountain: Measured read throughput as a function of spatial and temporal locality.

- Compact way to characterize memory system performance.



```
void copyij(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

4.3ms

2.0 GHz Intel Core i7 Haswell

```
void copyji(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

81.8ms

# Memory Mountain Test Function

```

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

/* test - Iterate over first "elems" elements of
 *         array "data" with stride of "stride",
 *         using 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`.

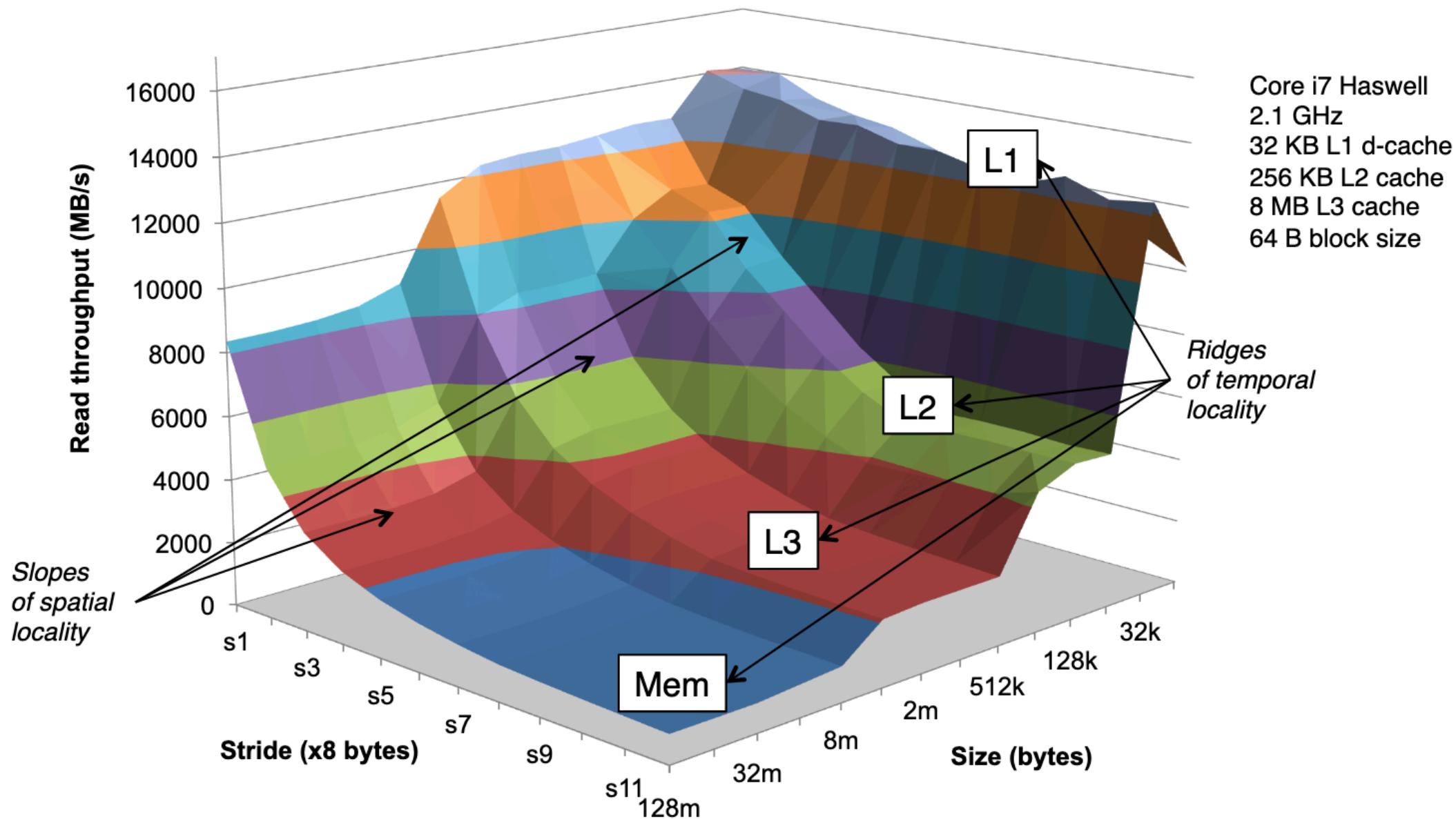
For each `elems` and `stride`:

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

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

*mountain/mountain.c*

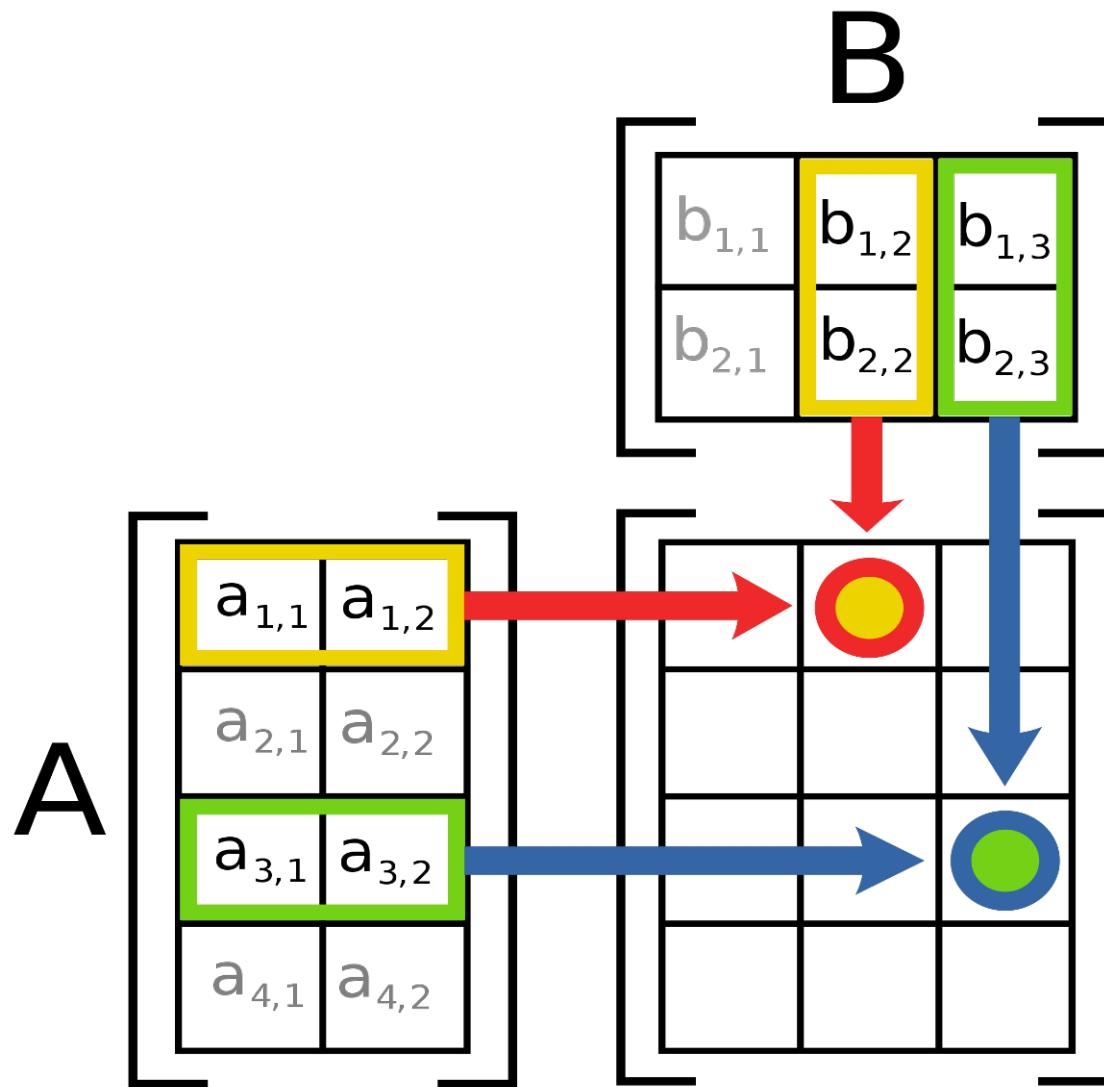
# The Memory Mountain



# Cache Memories

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

# Remember matrix multiplication



$\text{Out}[i, j] =$   
dot product( $A[i, ..]$ ,  $B[.., j]$ )  
= sum (  
     $a[i, 0] * b[0, j]$ ,  
     $a[i, 1] * b[1, j]$   
)

# Matrix Multiplication Example

## ■ Description:

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

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

*matmult/mm.c*

*Variable sum  
held in register*

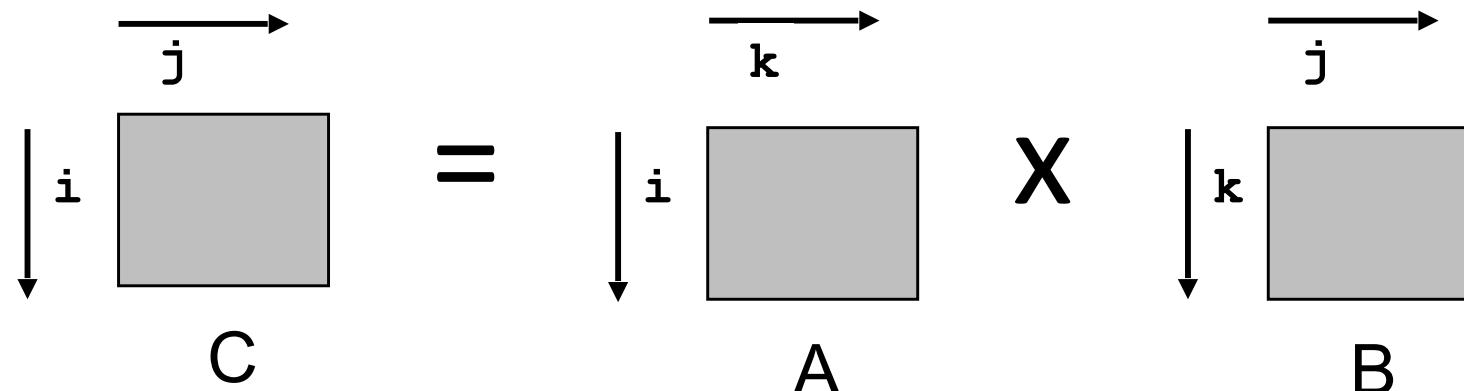
# Miss Rate Analysis for Matrix Multiply

## ■ Assume:

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

## ■ Analysis Method:

- Look at access pattern of inner loop



# Layout of C Arrays in Memory (review)

## ■ C arrays allocated in **row-major** order

- each row in contiguous memory locations
- $a[i][j] = a[i*N + j]$  where N is the number of columns

## ■ Stepping through columns in one row:

- ```
for (i = 0; i < N; i++)
    sum += a[0][i];
```
- accesses successive elements
- if block size (B) > sizeof( $a_{ij}$ ) bytes, exploit spatial locality
  - **miss rate =  $\text{sizeof}(a_{ij}) / B$ , e.g.,  $8/32 = 0.25$ , (cold miss)**

## ■ Stepping through rows in one column:

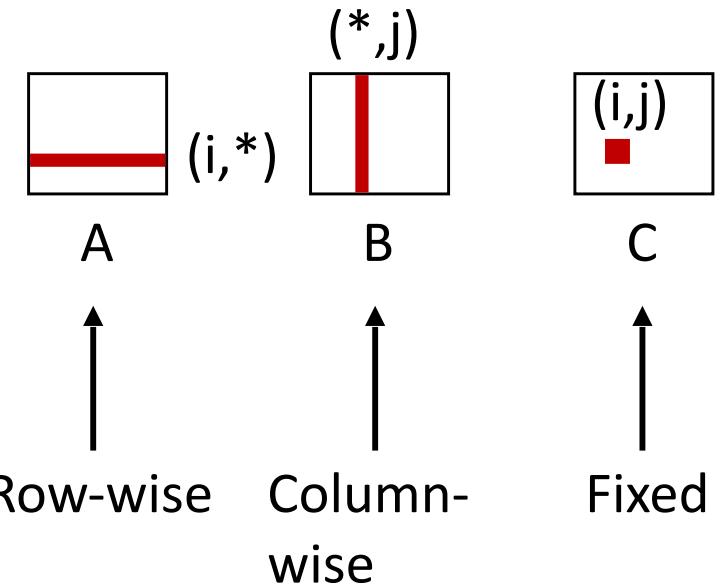
- ```
for (i = 0; i < n; i++)
    sum += a[i][0];
```
- accesses distant elements
- no spatial locality!
  - **miss rate = 1 (i.e. 100%)**

# Matrix Multiplication (ijk)

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

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

A                  B                  C

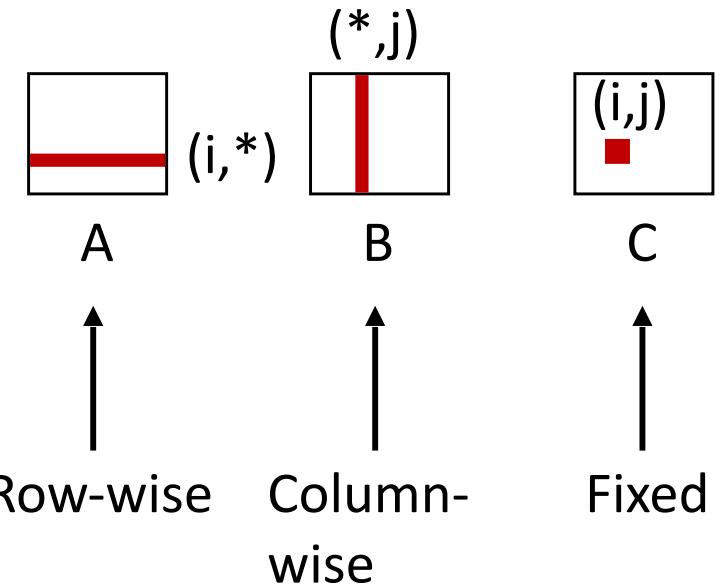
**Block size = 32B (four doubles)**

# Matrix Multiplication (ijk)

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

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

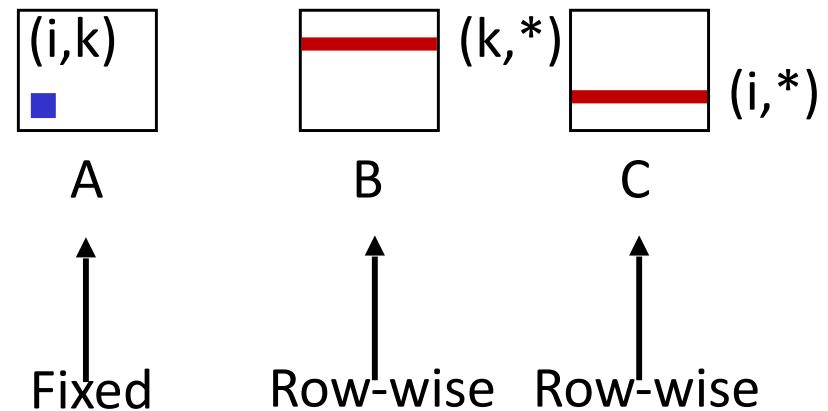
**Block size = 32B (four doubles)**

# Matrix Multiplication ( $k \times j$ )

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

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

A

B

C

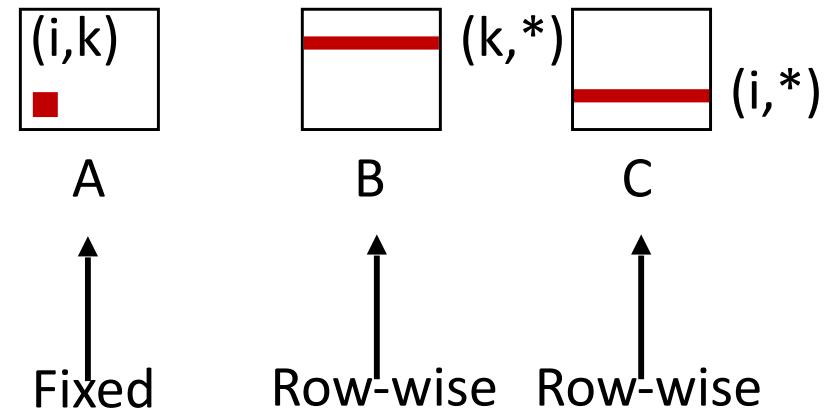
**Block size = 32B (four doubles)**

# Matrix Multiplication ( $k \times j$ )

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

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

A  
0.0

B  
0.25

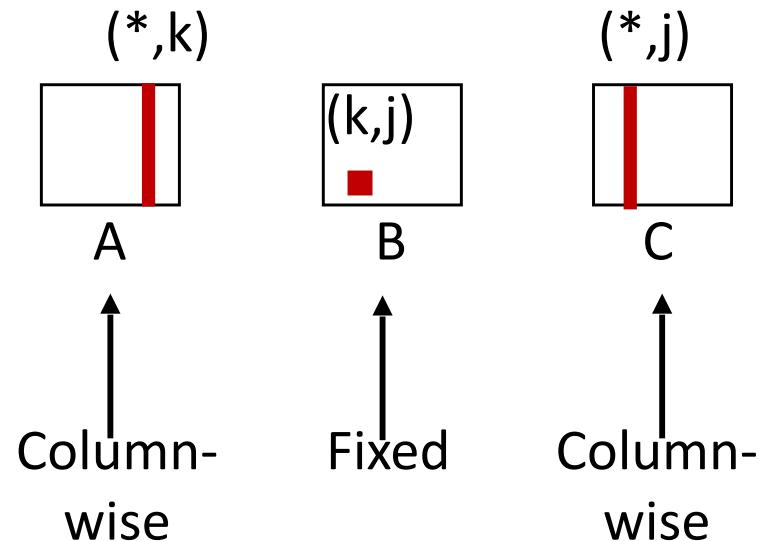
C  
0.25

**Block size = 32B (four doubles)**

# Matrix Multiplication (jki)

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

Inner loop:



Miss rate for inner loop iterations:

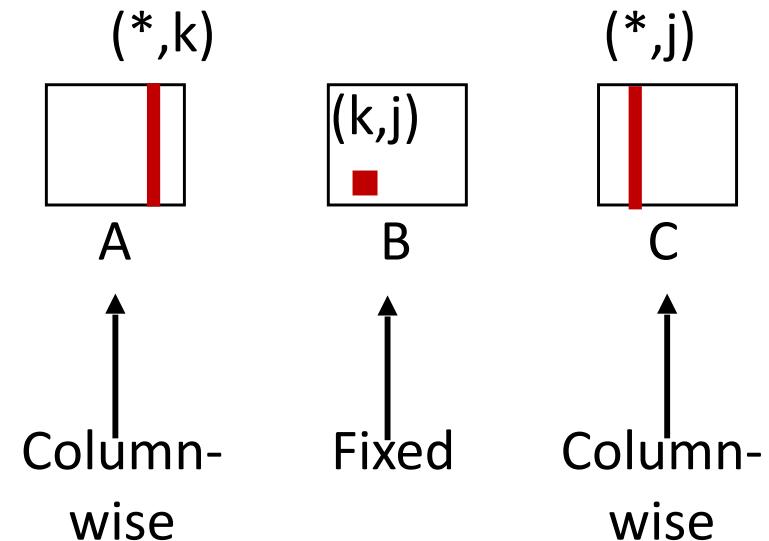
A      B      C

**Block size = 32B (four doubles)**

# Matrix Multiplication (jki)

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

Inner loop:



Miss rate for inner loop iterations:

A  
1.0

B  
0.0

C  
1.0

**Block size = 32B (four doubles)**

# Summary of Matrix Multiplication

```

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

```

```

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

```

```

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

```

## **ijk (& jik):**

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

## **kij (& ikj):**

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

## **jki (& kji):**

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

# Core i7 Matrix Multiply Performance

Cycles per inner loop iteration

100

jki / kji (2.0)

- ▲ jki
- ◆ kji
- ◆ ijk
- ▲ jik
- ◆ kij
- ▲ ikj

10

ijk / jik (1.25)

1

kij / ikj (0.5)

50 100 150 200 250 300 350 400 450 500 550 600 650 700

Array size (n)

# Cache Summary

■ Cache memories can have significant performance impact

■ You can write your programs to exploit this!

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