

# Memory Hierarchy and Caching

Troels Henriksen

Based on material by Randal E. Bryant and David R. O'Hallaron.

# What we will learn today

- The notions of *temporal* and *spatial* locality.
- What the memory hierarchy is and what caching is.
- How a cache actually works, and its implications for performance.
- How to modify programs to improve their locality.

Locality of reference

Memory hierarchies

Cache organisation and operation

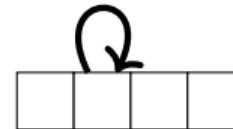
Cache performance

# Locality

## Principle of locality

Programs tend to access data located near that which was accessed recently.

### Temporal locality



Accessing data that was accessed recently.

### Spatial locality



Accessing data that is close to data that was accessed recently.

- **General principles** – definition of “close” depends on the exact form of storage.
  - ▶ E.g. addresses for memory.

## Locality example

```
double sum = 0;  
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

### Data references

- References array elements in succession (*stride* of 1).
- References variable `sum` each iteration.

# Locality example

```
double sum = 0;  
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

## Data references

- References array elements in succession (*stride* of 1).  
**Spatial locality.**
- References variable `sum` each iteration.  
**Temporal locality.**

# Locality example

```
double sum = 0;  
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

## Data references

- References array elements in succession (*stride* of 1).  
**Spatial locality.**
- References variable `sum` each iteration.  
**Temporal locality.**

## Instruction references

- Executes instructions in sequence.
- Cycles through loop repeatedly.

# Locality example

```
double sum = 0;  
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

## Data references

- References array elements in succession (*stride* of 1).  
**Spatial locality.**
- References variable `sum` each iteration.  
**Temporal locality.**

## Instruction references

- Executes instructions in sequence.  
**Spatial locality.**
- Cycles through loop repeatedly.  
**Temporal locality.**

**Code has good locality by default, so we tend to focus only on *data locality*.**

# C array layout

```
int A[M][N];
```

To represent multi-dimensional arrays, C uses *row major order*.

## Main consequence

- Rows are contiguous in memory.

### Example

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

is represented as 

11	12	13	14	21	22	23	24	31	32	33	34
----	----	----	----	----	----	----	----	----	----	----	----

## Implications

- $A[i][j]$  and  $A[i][j+1]$  are adjacent.
- $A[i][j]$  and  $A[i+1][j]$  are distant.

# Eyeballing locality

Being able to glance at code and get a qualitative sense of its locality properties is a key skill for a programmer.

Does this function have good locality with respect to array A?

```
int sumrows (int A[M] [N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            sum += A[i][j];  
    return sum;  
}
```

Does this function have good locality with respect to array A?

```
int sumcols(int A[M][N]) {
    int sum = 0;

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

## Transforming code for better locality

Can we permute the loops of this function such that we are accessing the memory of array A with a stride of 1?

```
int sum3d(int A[L][M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            for (int k = 0; k < L; k++)  
                sum += A[k][i][j];  
    return sum;  
}
```

# Transforming code for better locality

Can we permute the loops of this function such that we are accessing the memory of array A with a stride of 1?

```
int sum3d(int A[L][M][N]) {  
    int sum = 0;  
  
    for (int i = 0; i < M; i++)  
        for (int j = 0; j < N; j++)  
            for (int k = 0; k < L; k++)  
                sum += A[k][i][j];  
    return sum;  
}
```

**Yes:** place them in order k, i, j.

Locality of reference

## Memory hierarchies

Cache organisation and operation

Cache performance

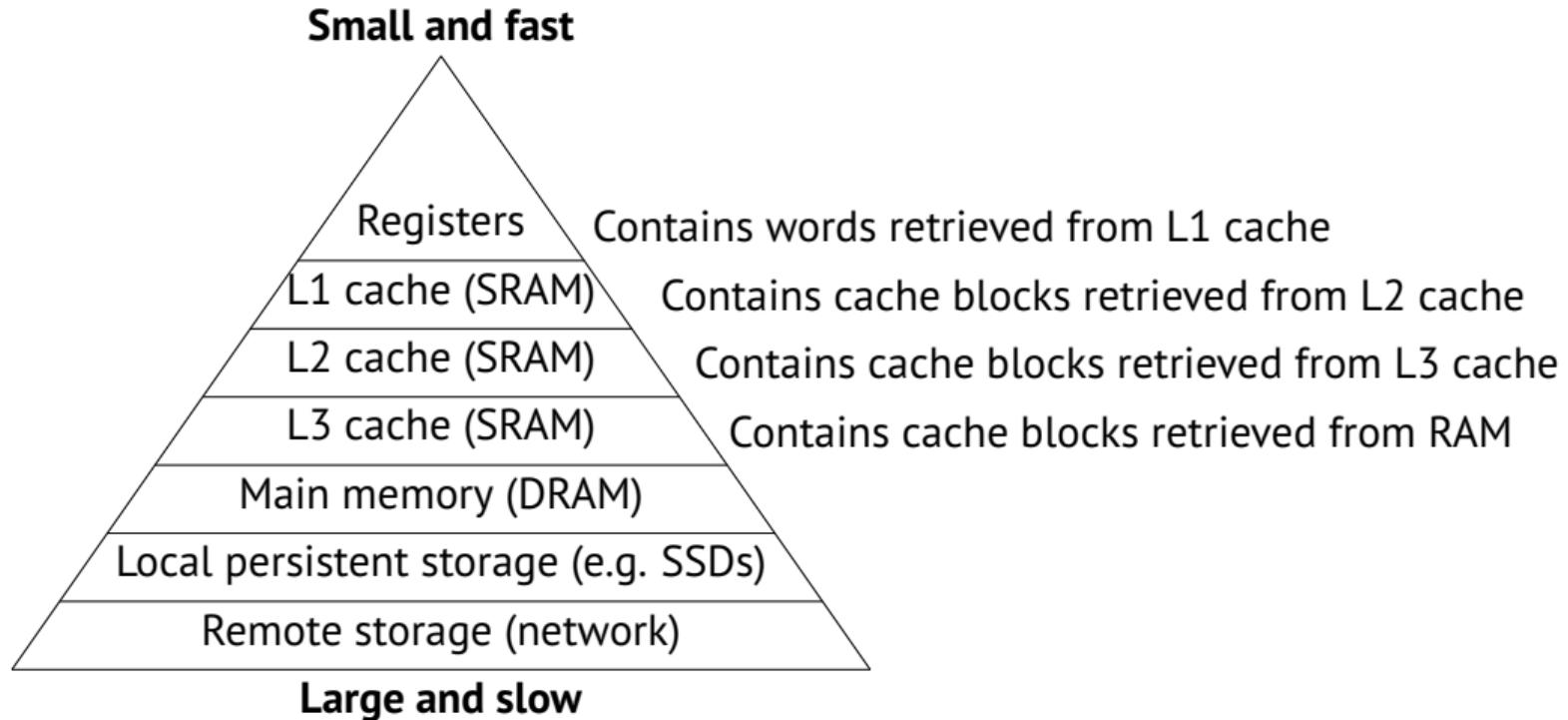
# Memory hierarchies

Some fundamental and enduring properties of hardware and software

- Fast storage is expensive, has smaller capacity, and requires more power.
- There is a large gap between computational speed and memory speed.
- Well-written programs tend to exhibit good locality.

These properties suggest an approach for organising memory and other storage systems known as a *memory hierarchy*.

# Example memory hierarchy



# Caches

## Definition of *cache*

A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- **Fundamental idea of the memory hierarchy**

- ▶ The smaller and faster device at level  $k$  acts as a cache for the larger slower device at level  $k + 1$ .

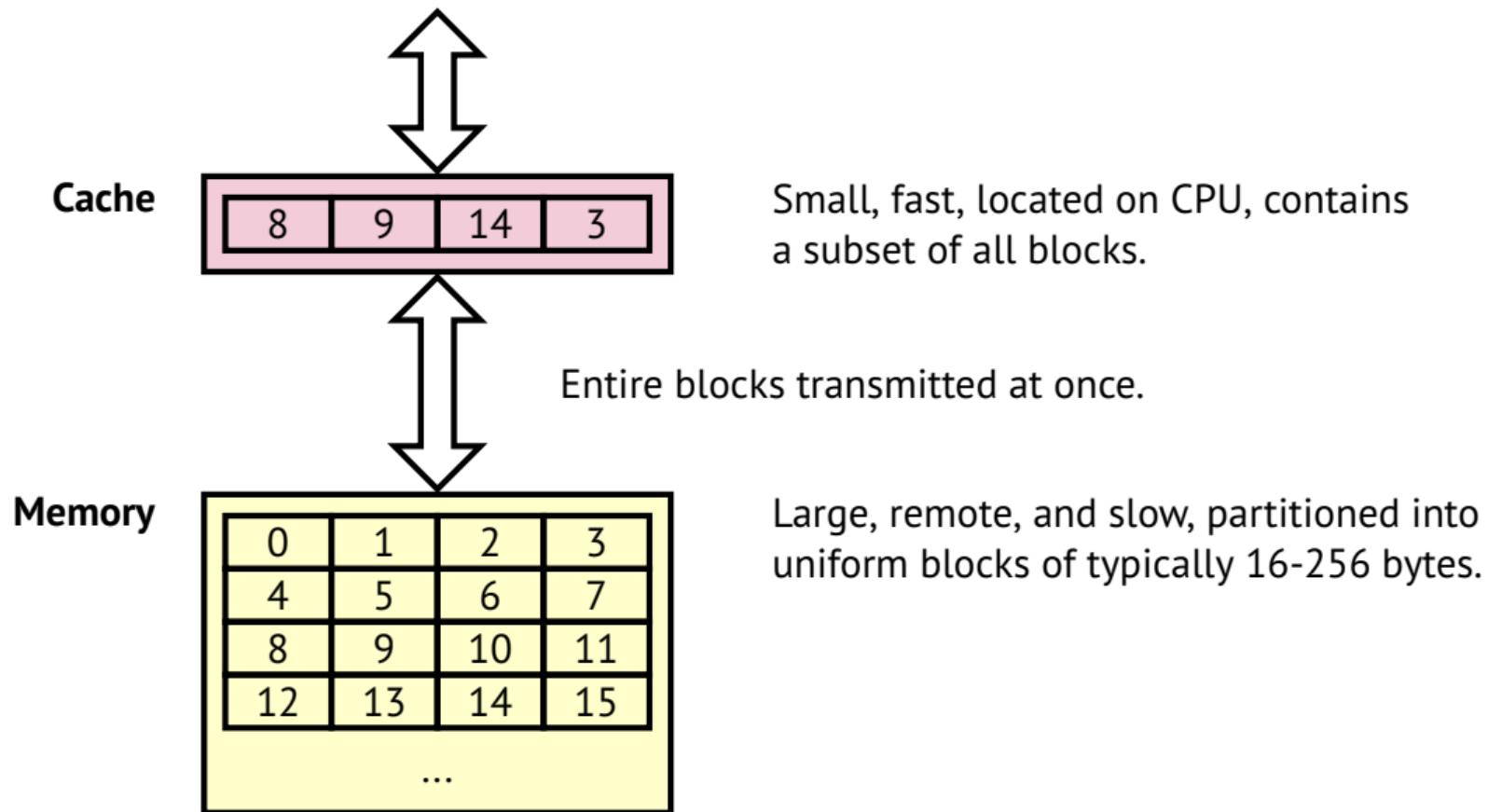
- **Why do they work?**

- ▶ Because of *locality*, most accesses tend to be towards the top of the hierarchy.

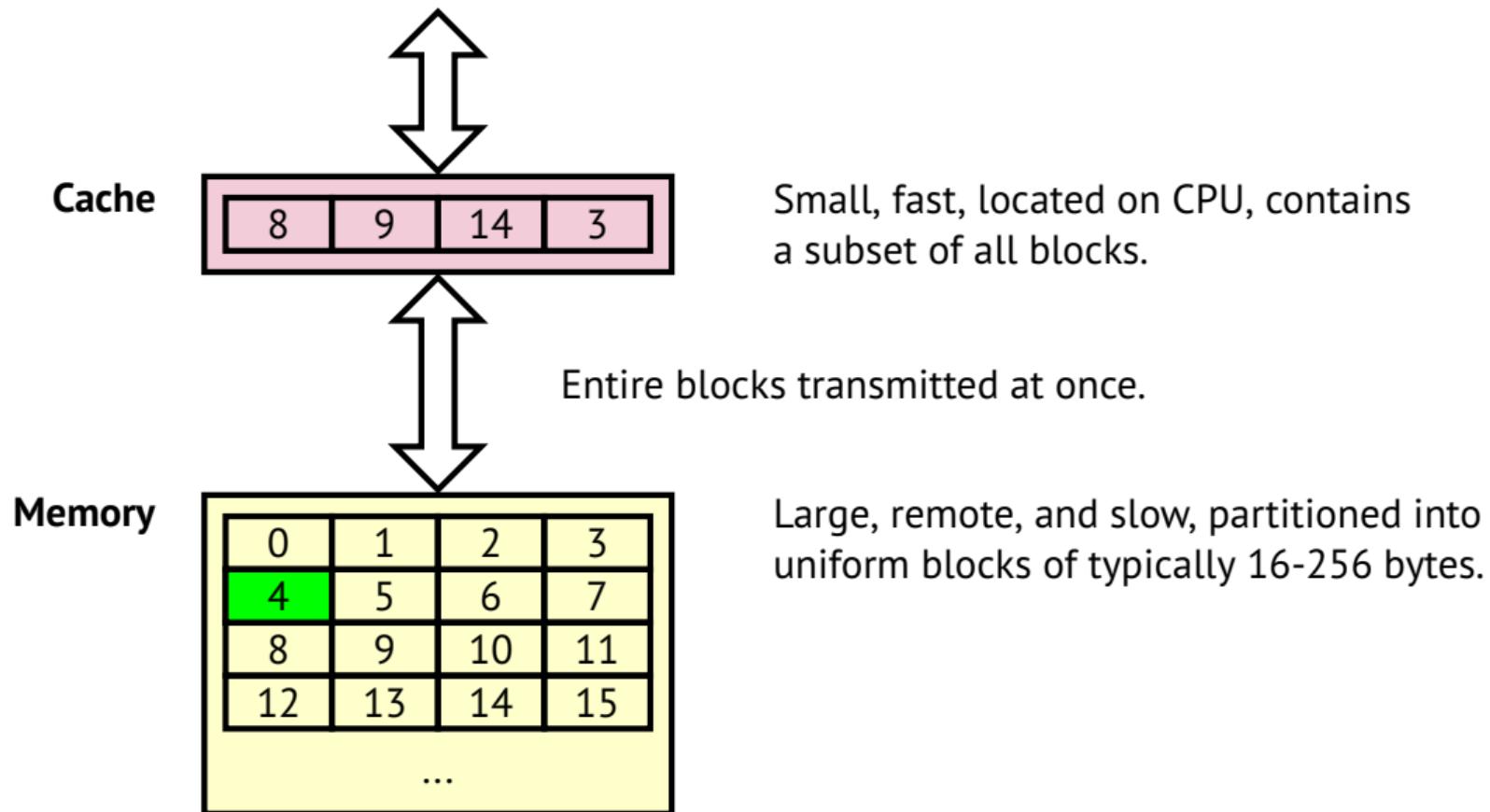
- **The ideal**

- ▶ A huge pool of storage that is as cheap as at the bottom of the hierarchy, but as fast as at the top of the hierarchy.

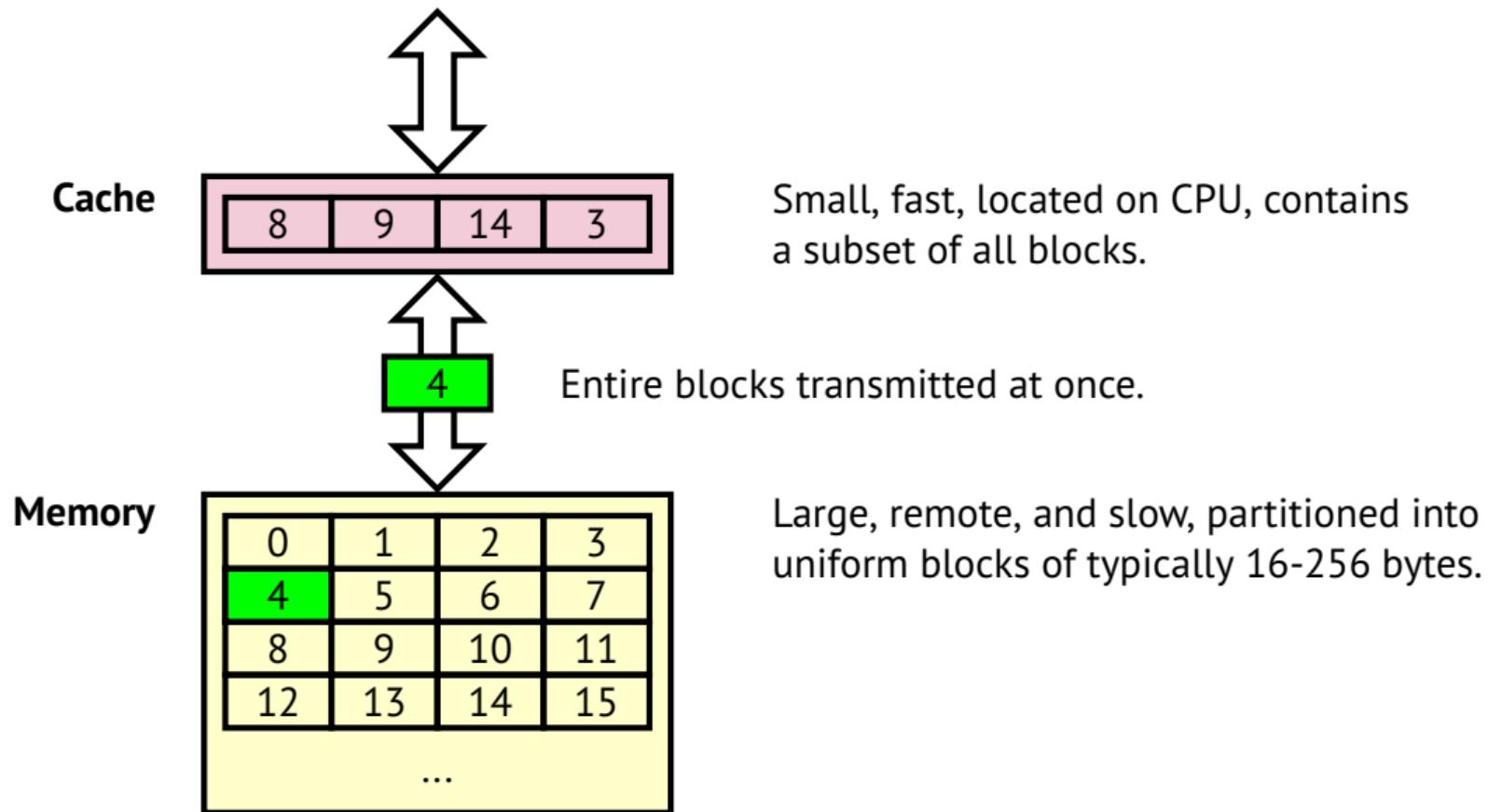
# Cache overview



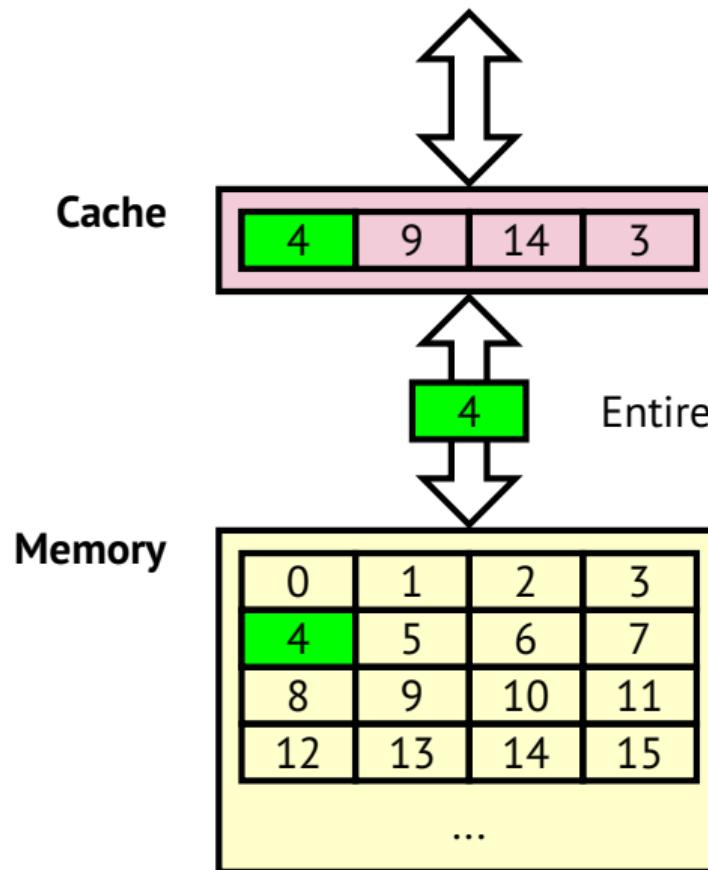
# Cache overview



# Cache overview



# Cache overview

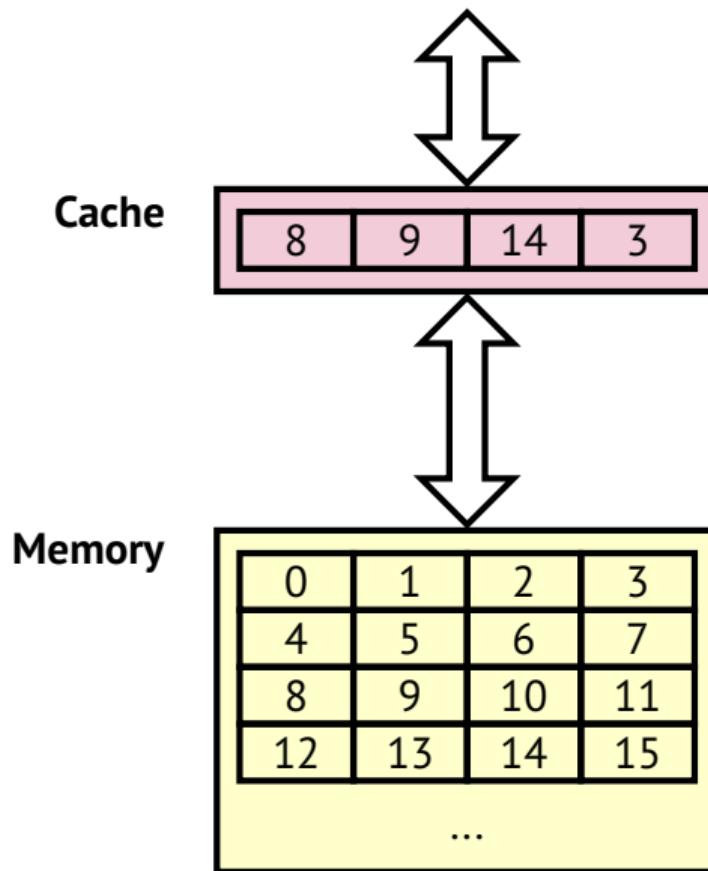


Small, fast, located on CPU, contains a subset of all blocks.

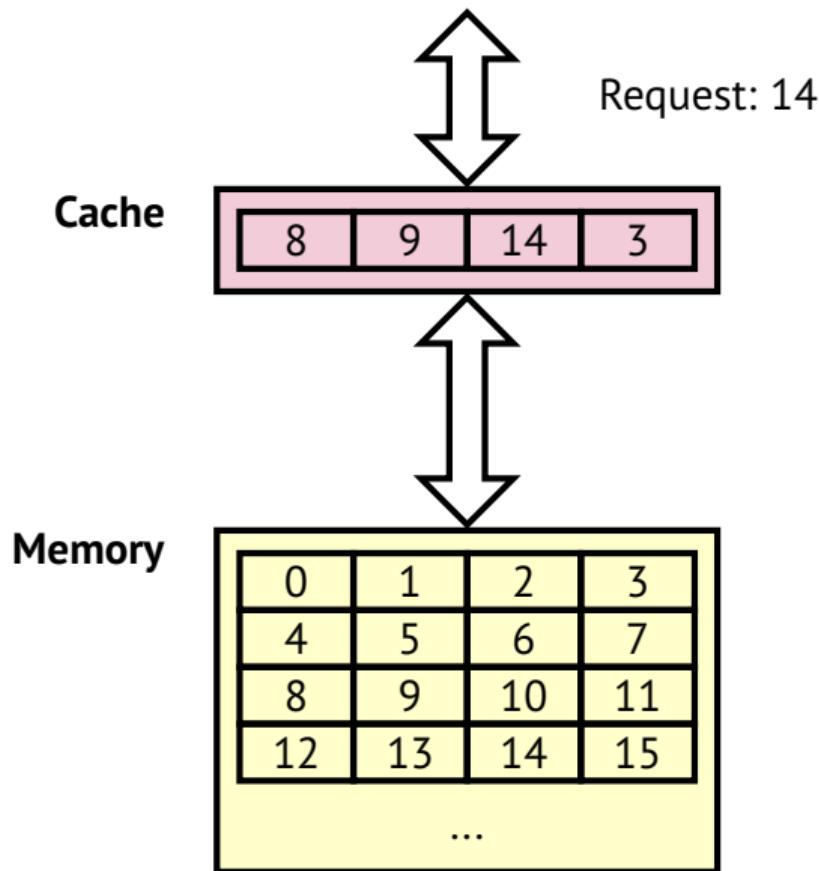
Entire blocks transmitted at once.

Large, remote, and slow, partitioned into uniform blocks of typically 16-256 bytes.

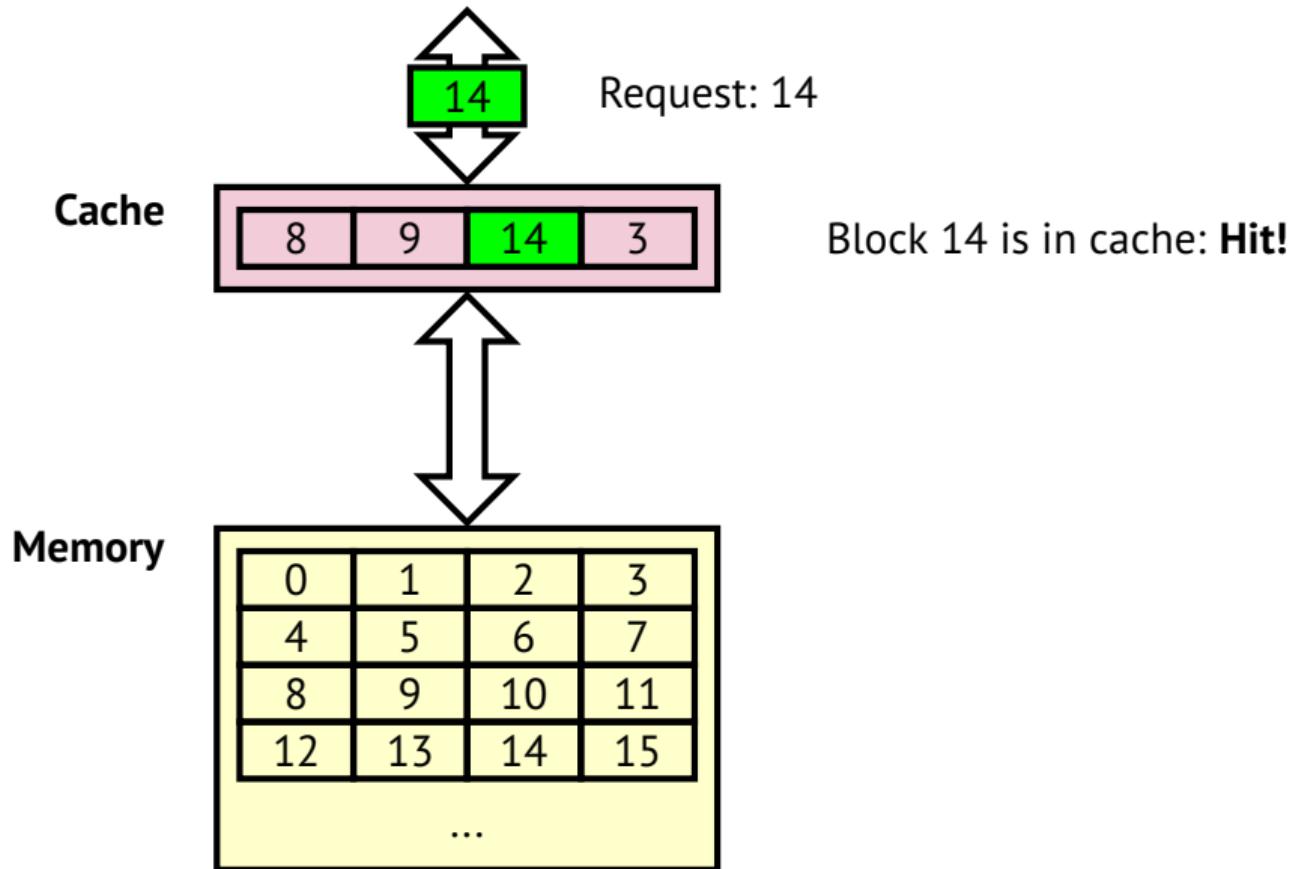
## Example: cache hit



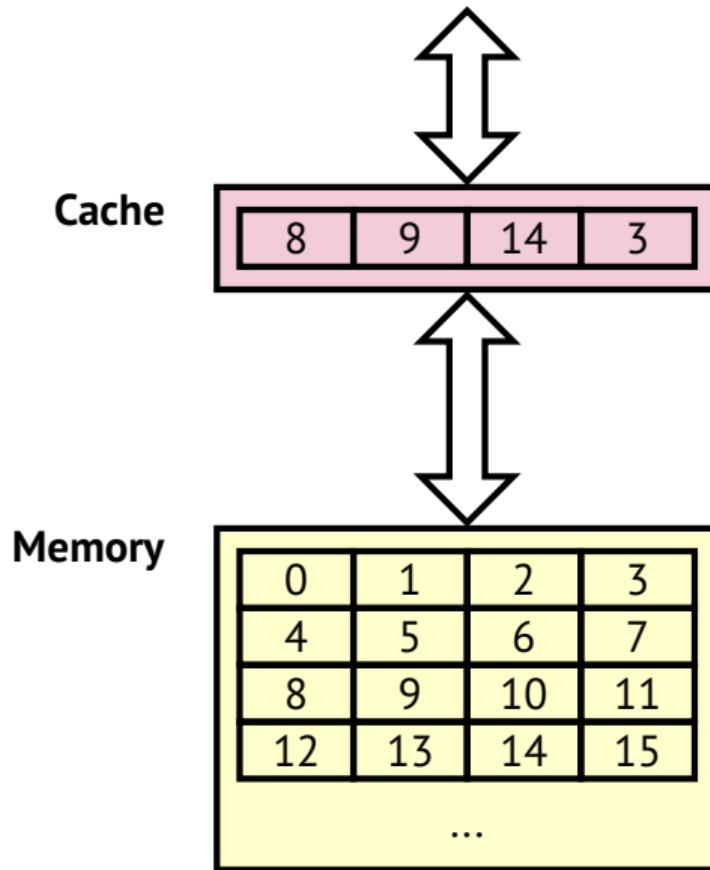
## Example: cache hit



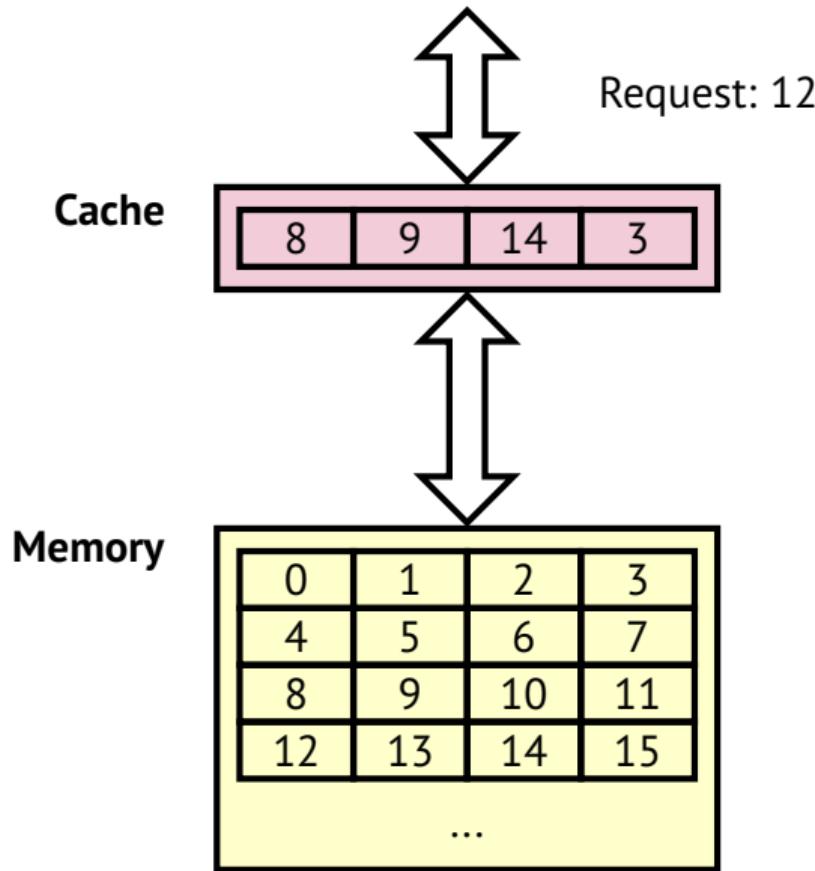
## Example: cache hit



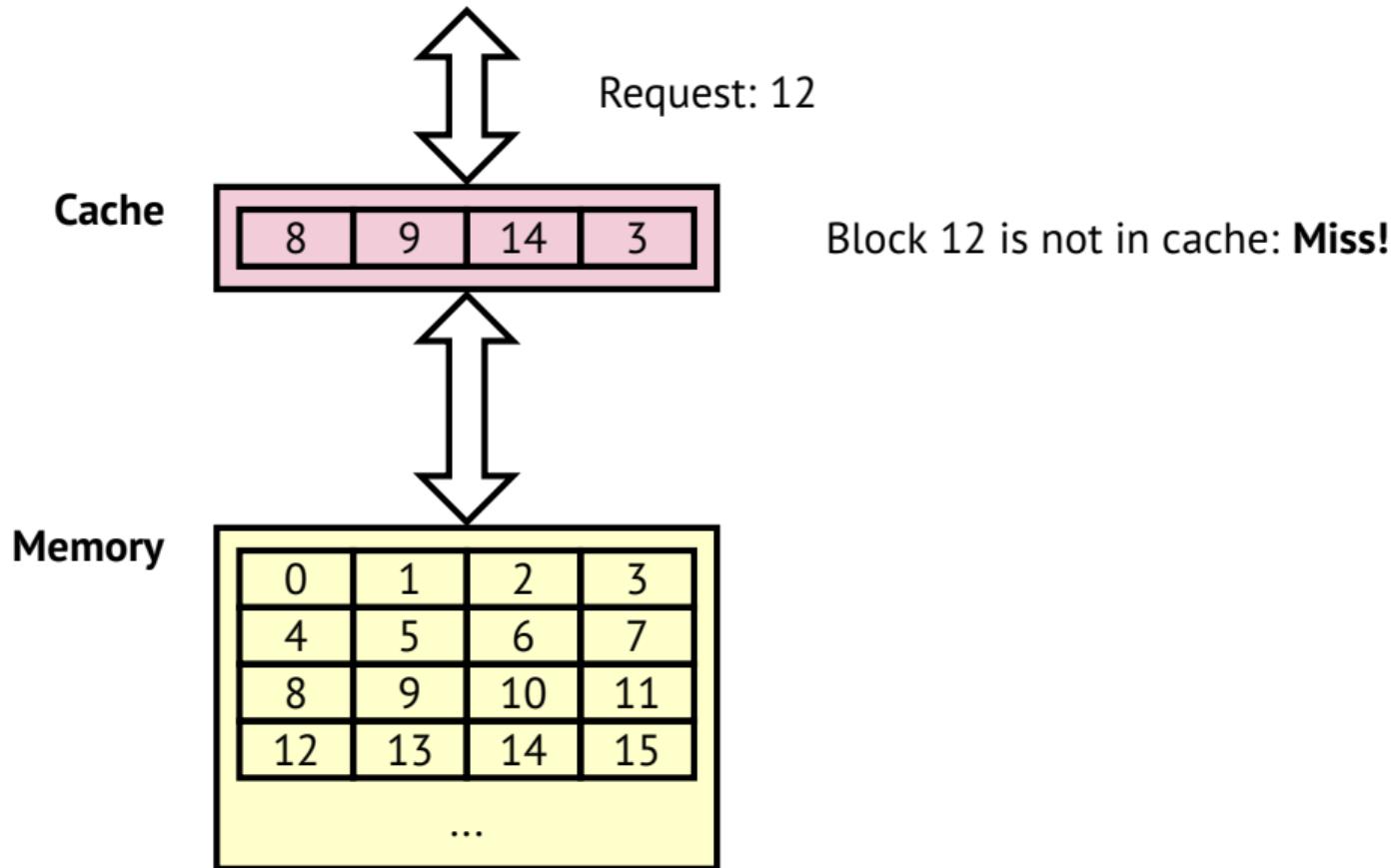
## Example: cache miss



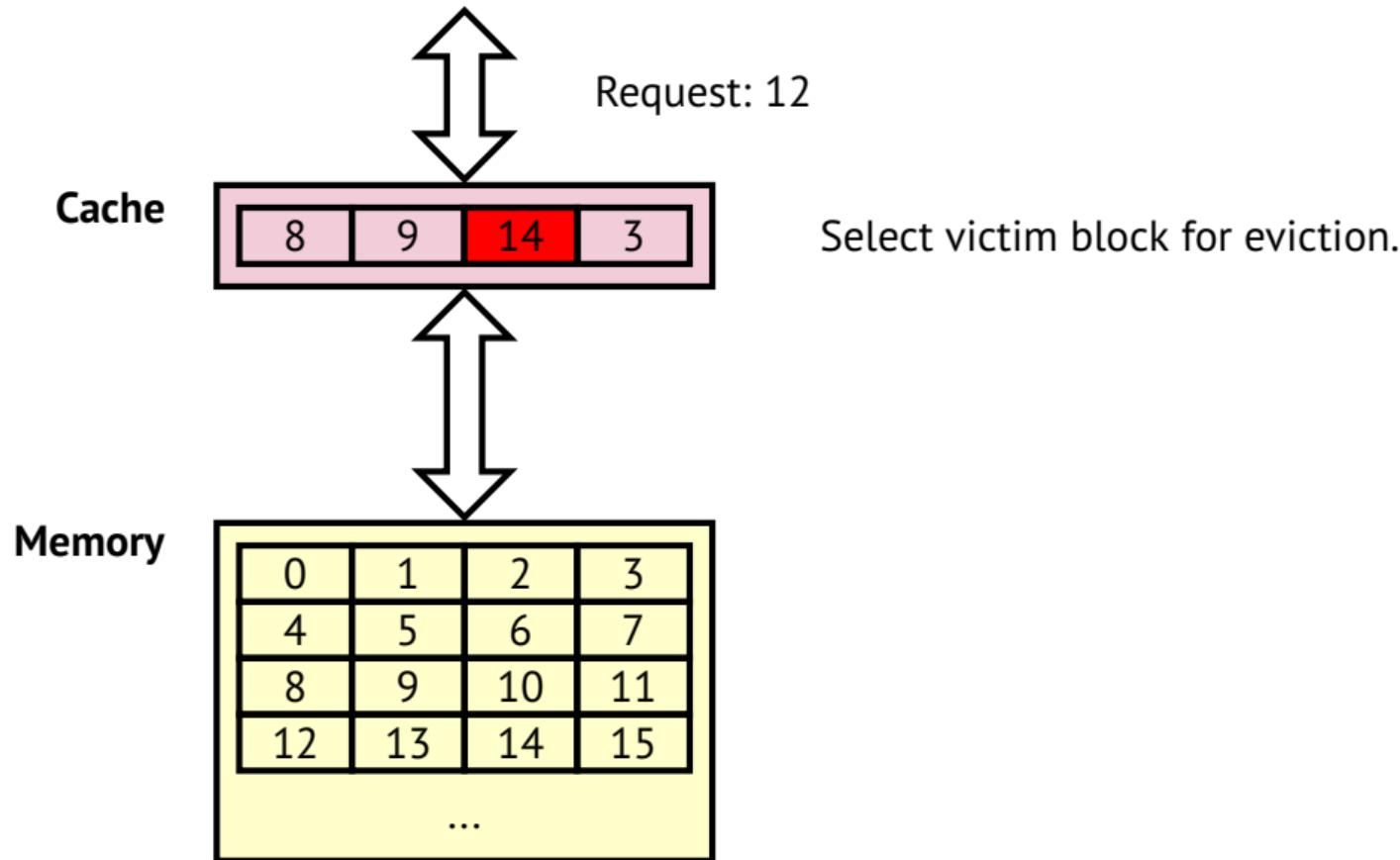
## Example: cache miss



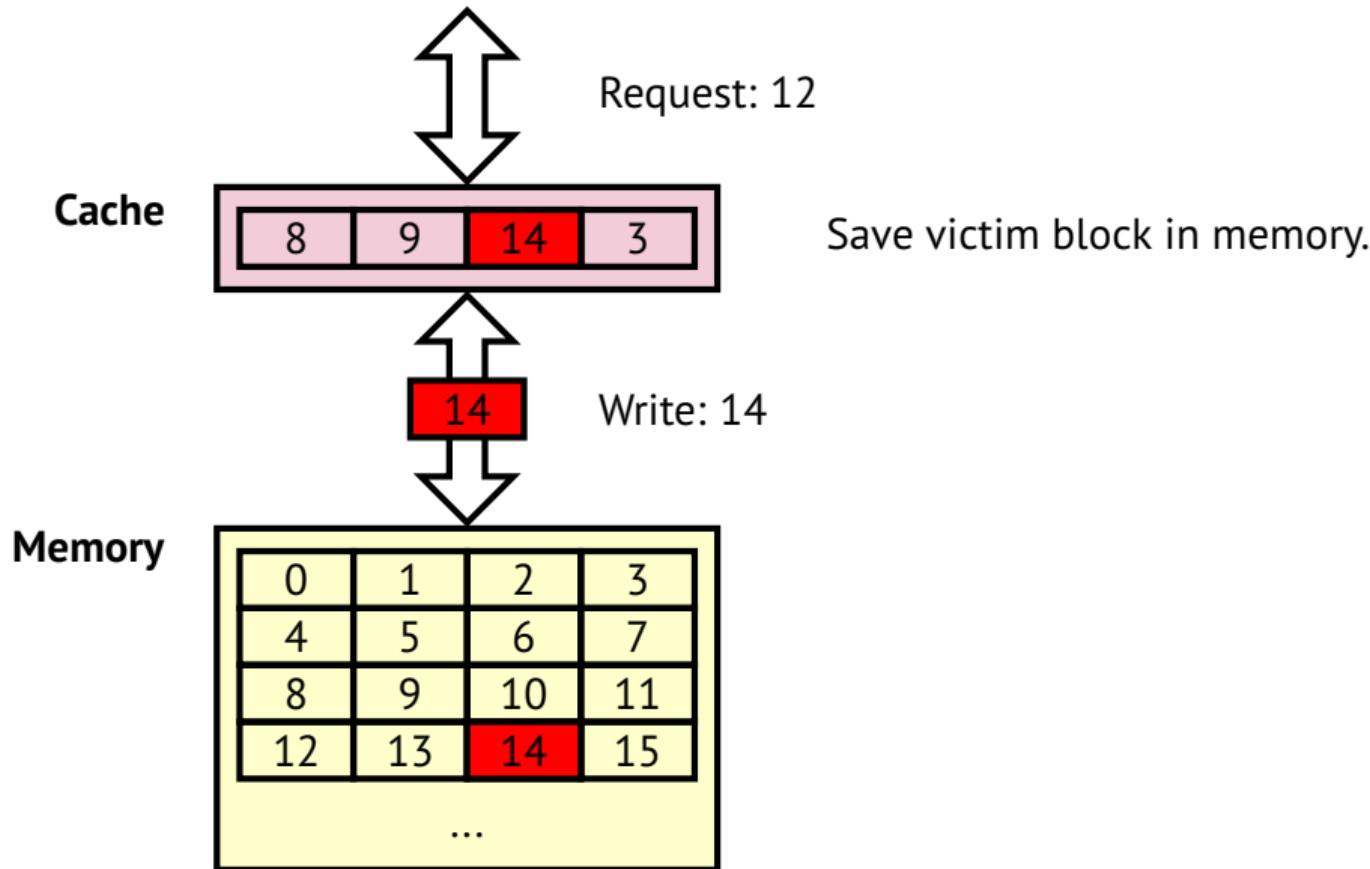
## Example: cache miss



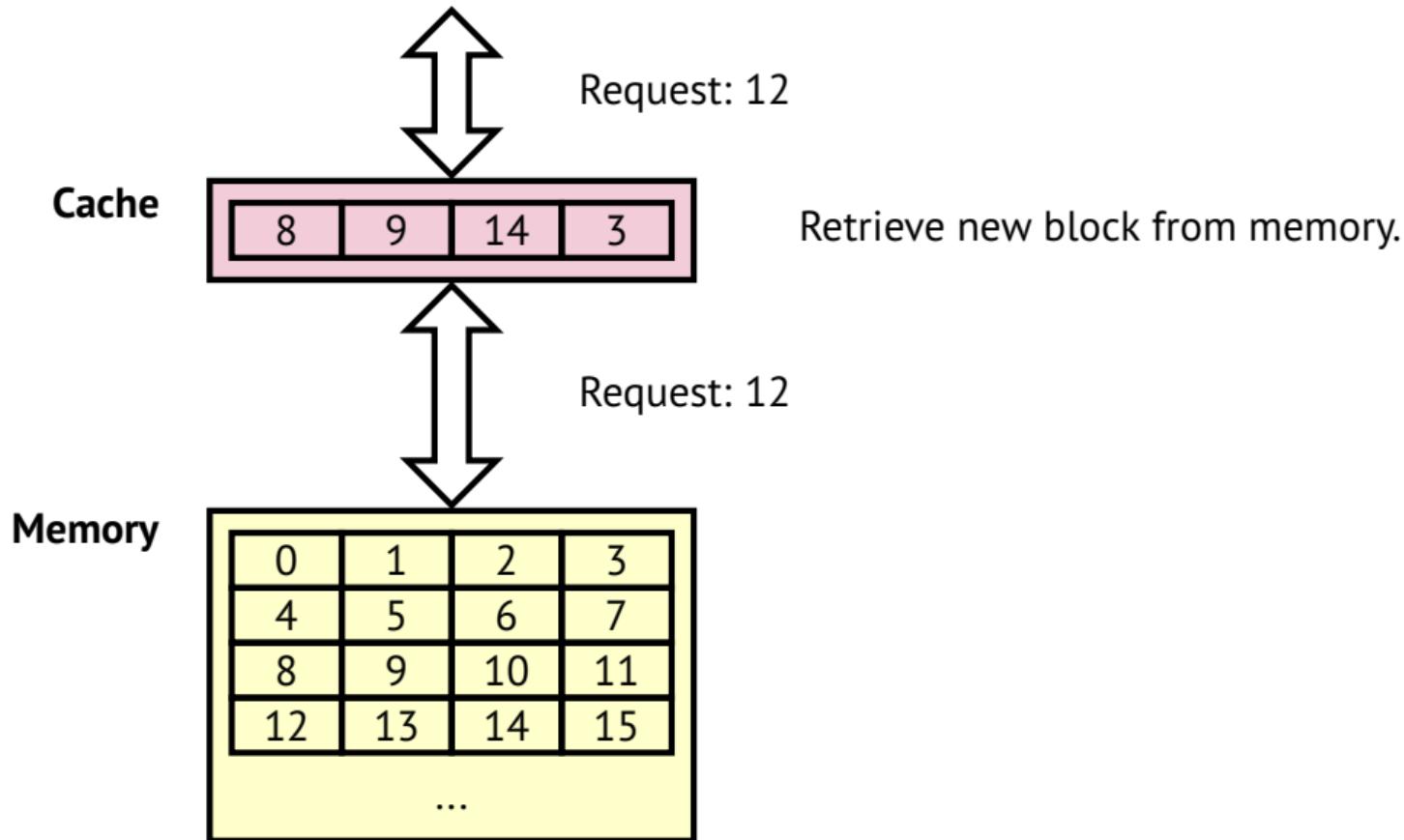
## Example: cache miss



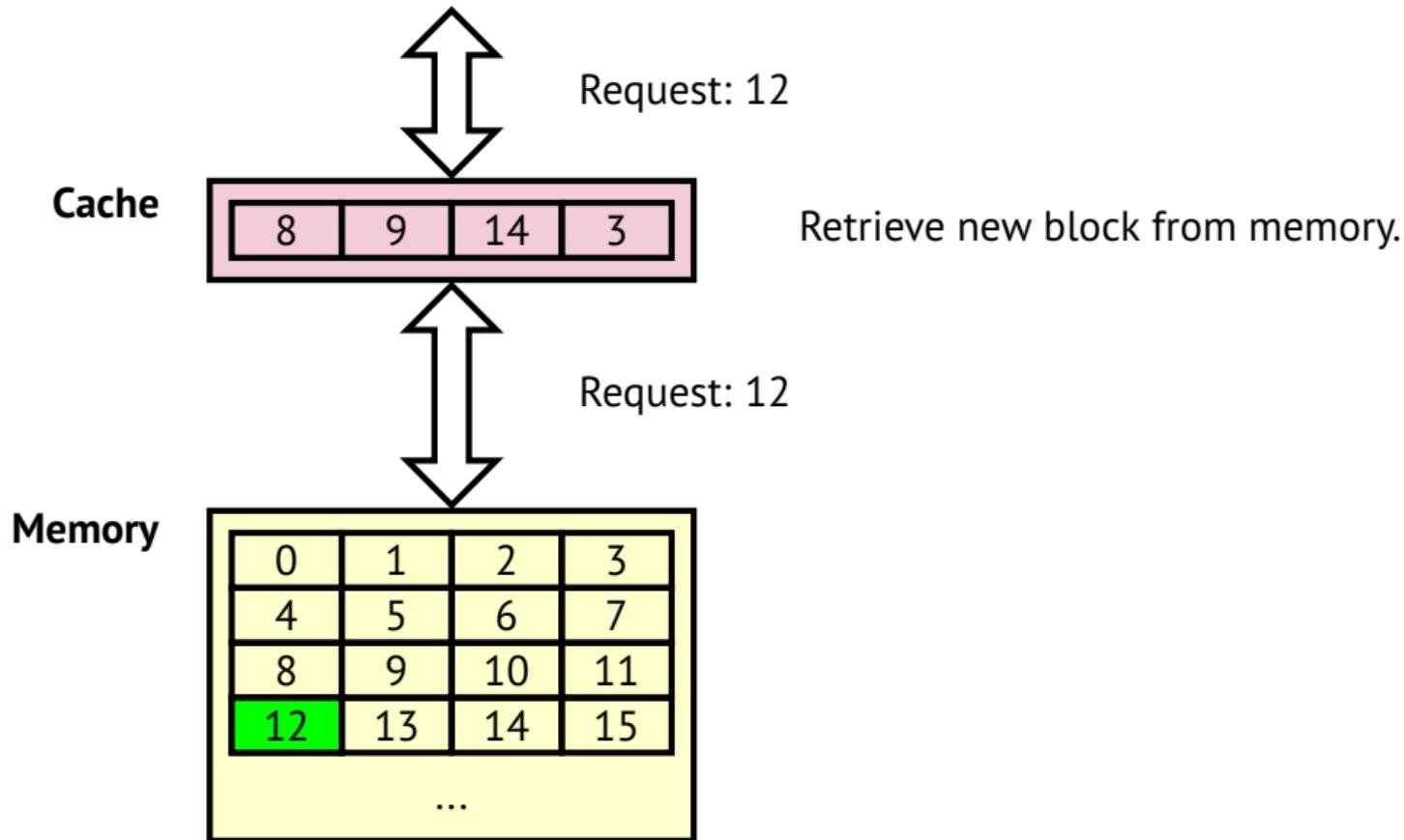
## Example: cache miss



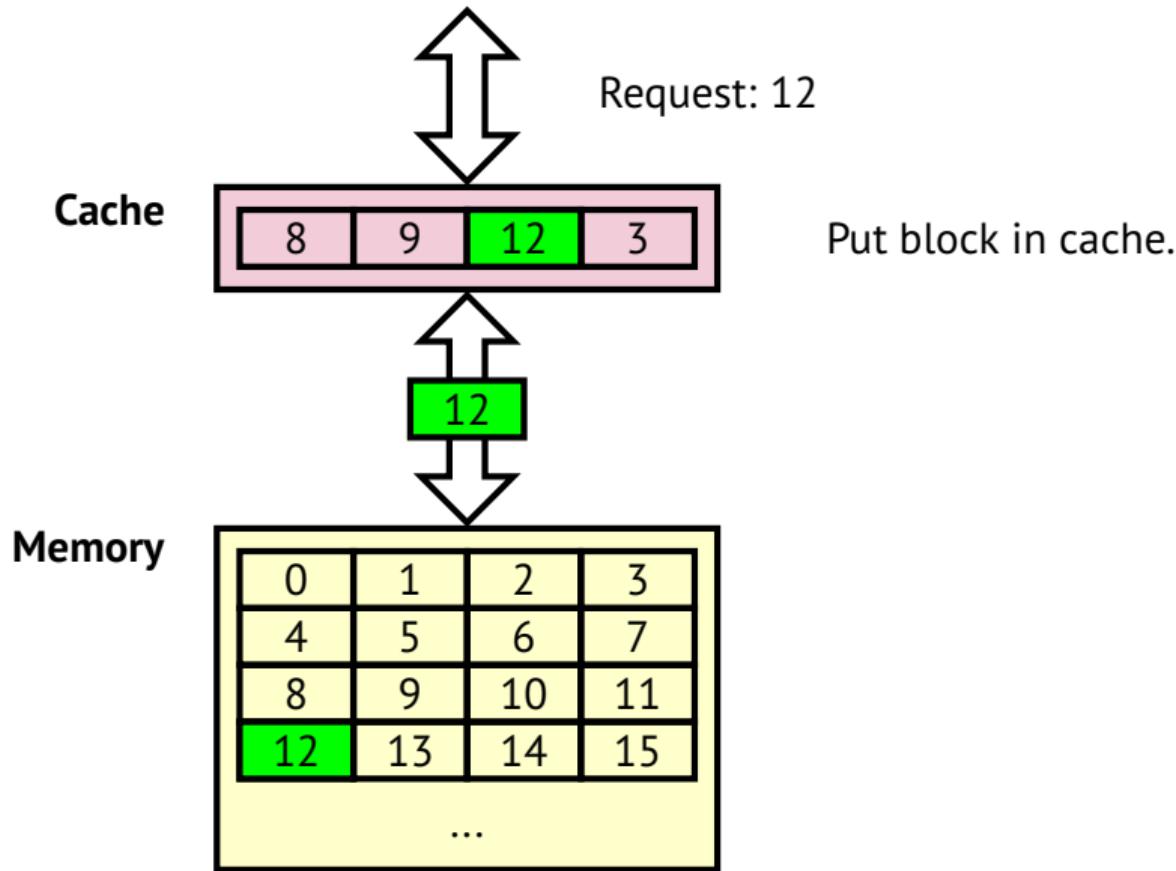
## Example: cache miss



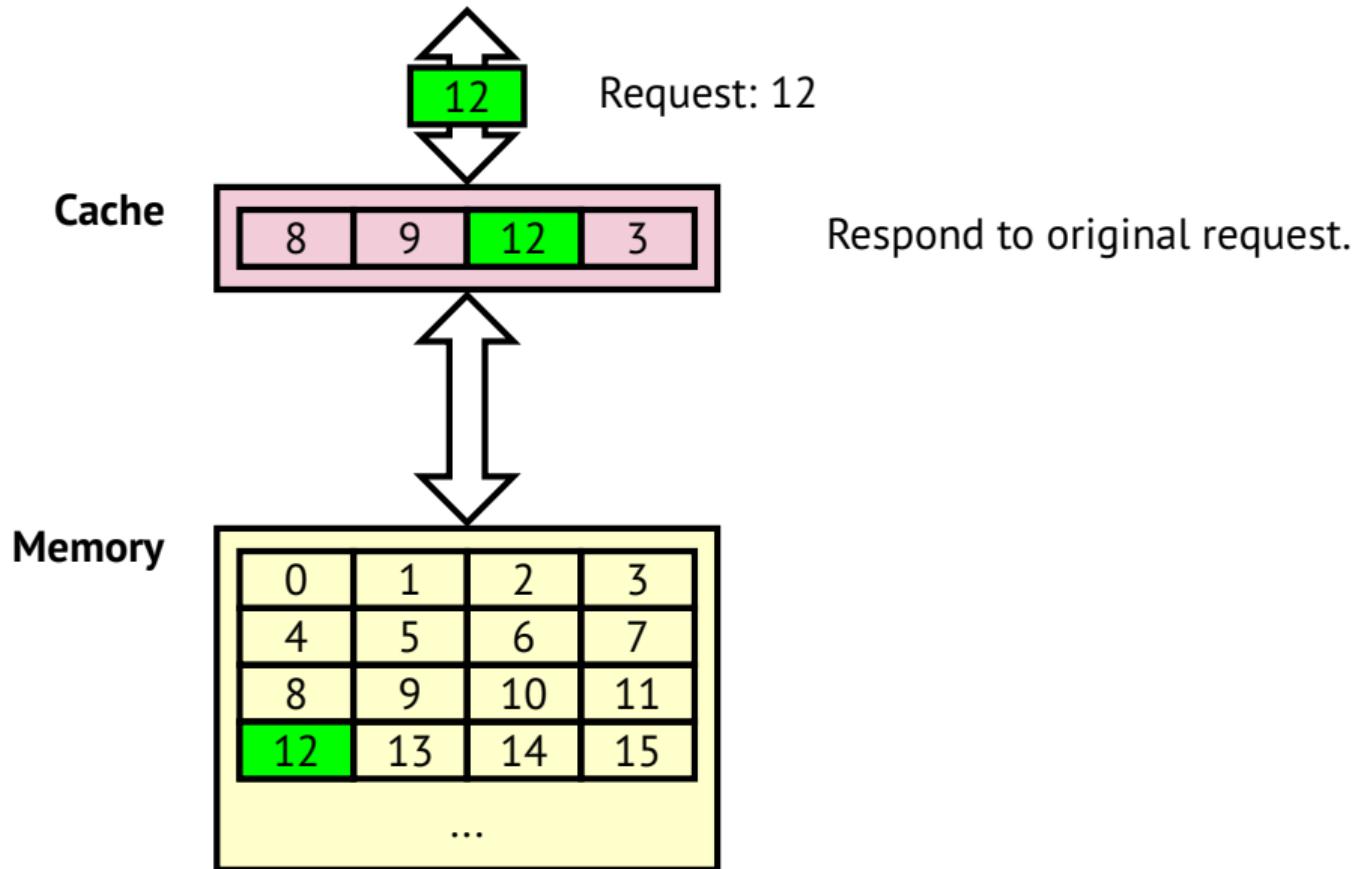
## Example: cache miss



## Example: cache miss



## Example: cache miss



# Types of cache misses

## Cold/compulsory miss

- Occur when the cache is empty.
- Unavoidable when a program first starts.

## Conflict miss

- Most caches limit blocks at level  $k + 1$  to a small subset of the slots at level  $k$ .
  - ▶ **Example:** Block  $i$  can only be located in slot  $i \bmod 4$ .
- Causes conflicts when cache is large enough, but the blocks being accessed all map to the same slot.

## Capacity miss

- Occurs when program *working set* exceeds size of cache.

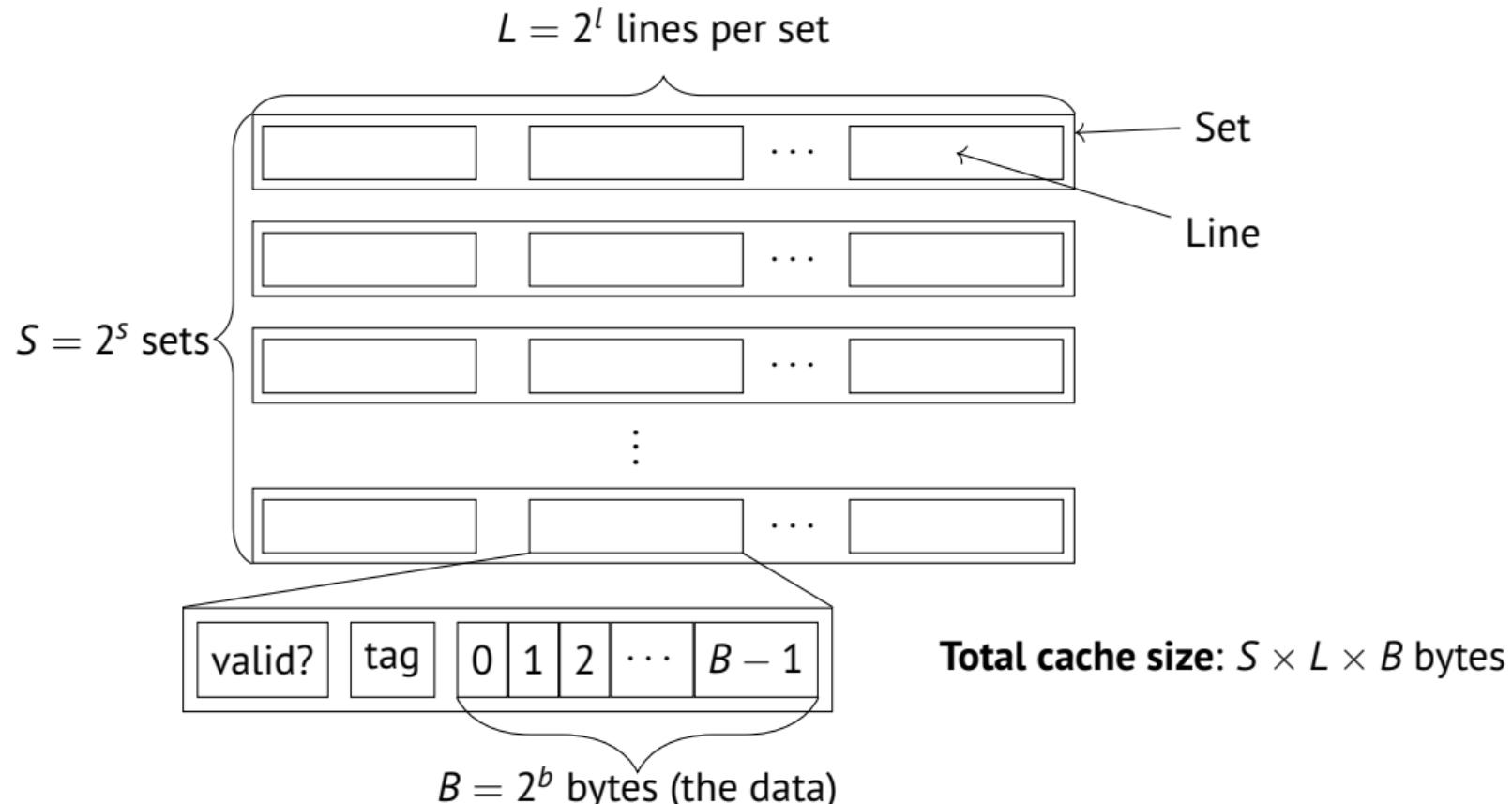
Locality of reference

Memory hierarchies

**Cache organisation and operation**

Cache performance

# General structure of a cache for $S$ , $L$ , and $B$



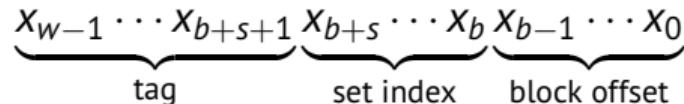
# Address structure

When  $S = 2^n, B = 2^m$  we can easily split a  $w$ -bit address into *fields*, writing  $x_i$  for bit  $i$ .

$$\underbrace{x_{w-1} \cdots x_{b+s+1}}_{\text{tag}} \underbrace{x_{b+s} \cdots x_b}_{\text{set index}} \underbrace{x_{b-1} \cdots x_0}_{\text{block offset}}$$

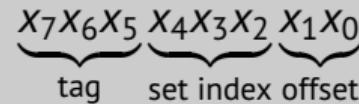
# Address structure

When  $S = 2^n, B = 2^m$  we can easily split a  $w$ -bit address into *fields*, writing  $x_i$  for bit  $i$ .



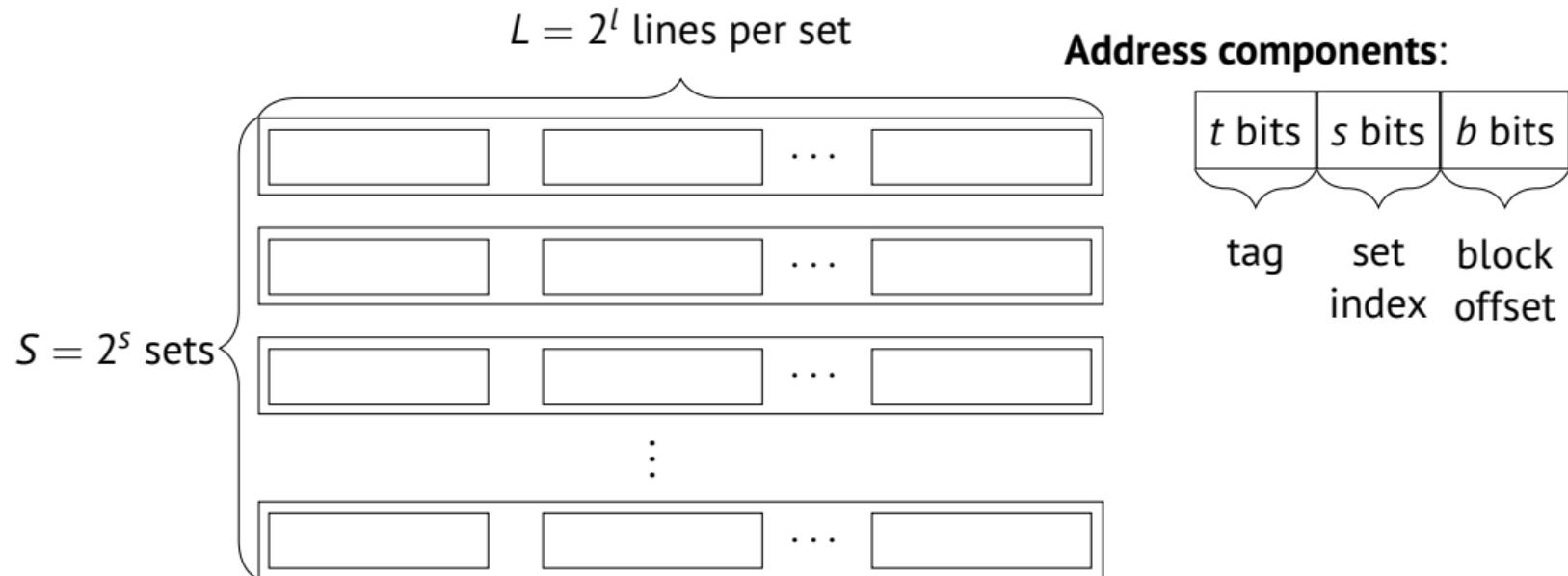
## Example

Consider an 8-bit address with  $m = 2, s = 3$ .

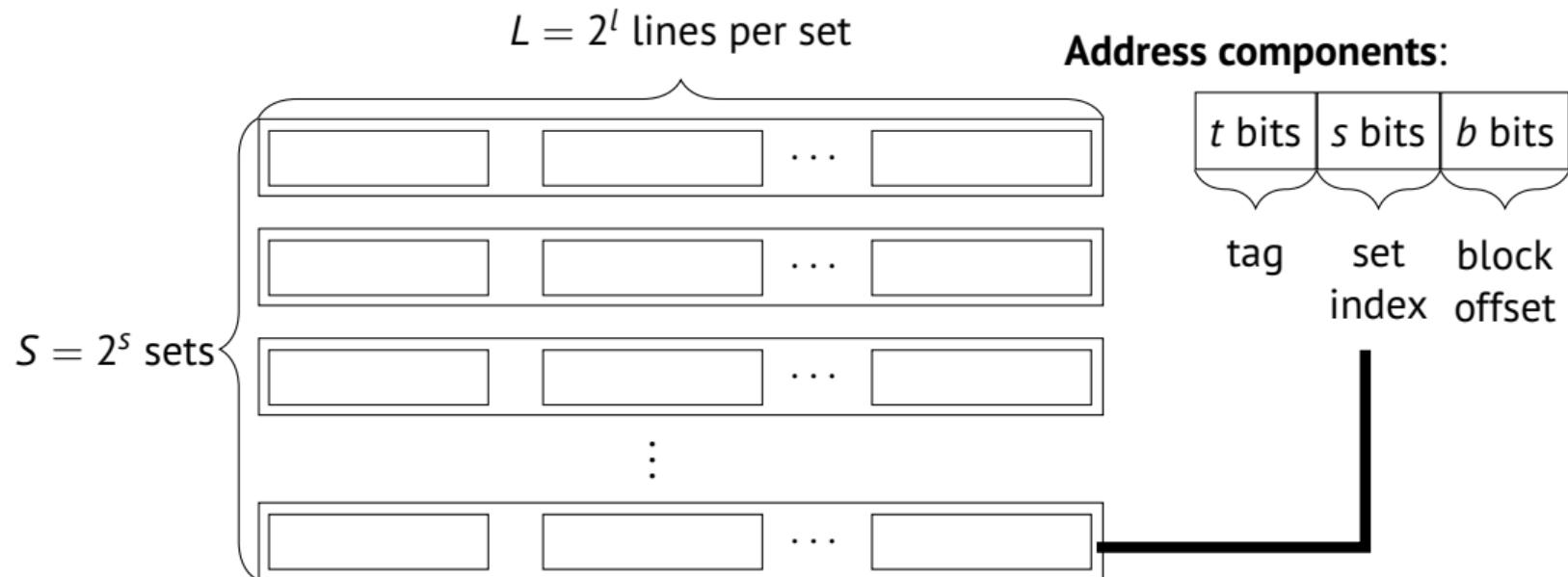


We look up an address in the cache by splitting the address into fields and looking up and checking based on their values.

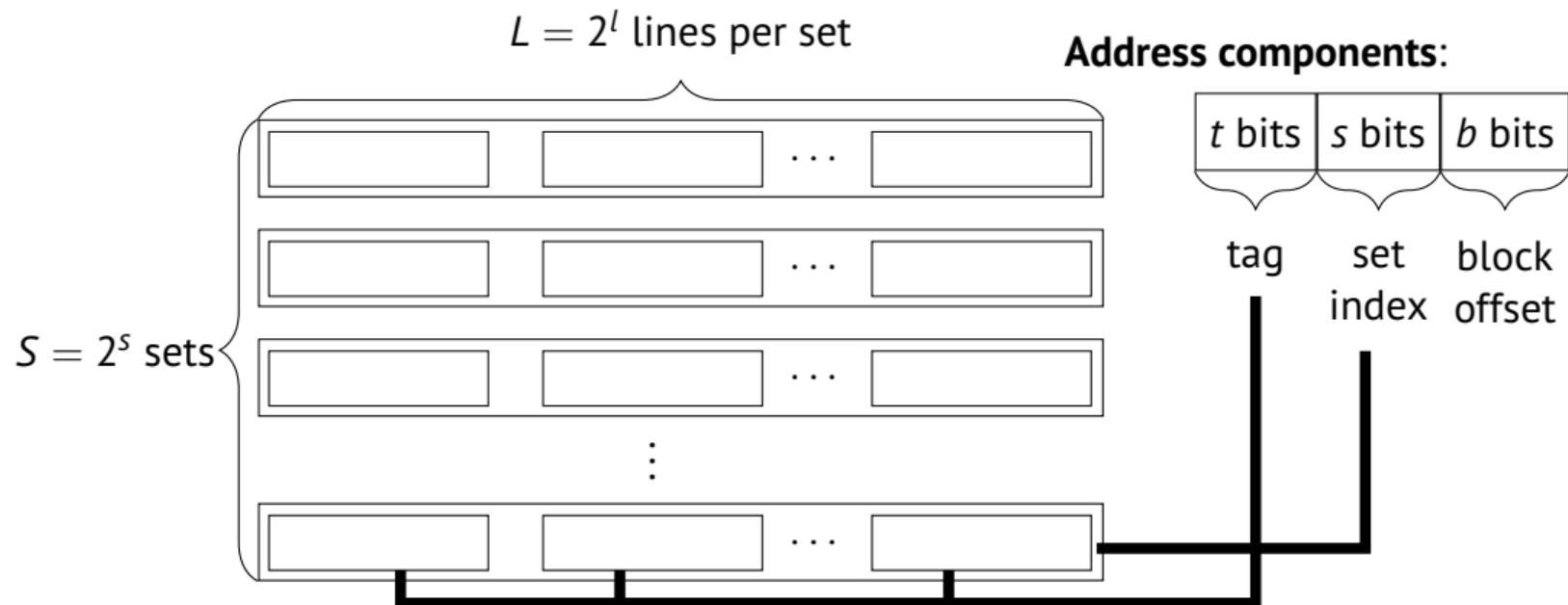
# Cache lookup



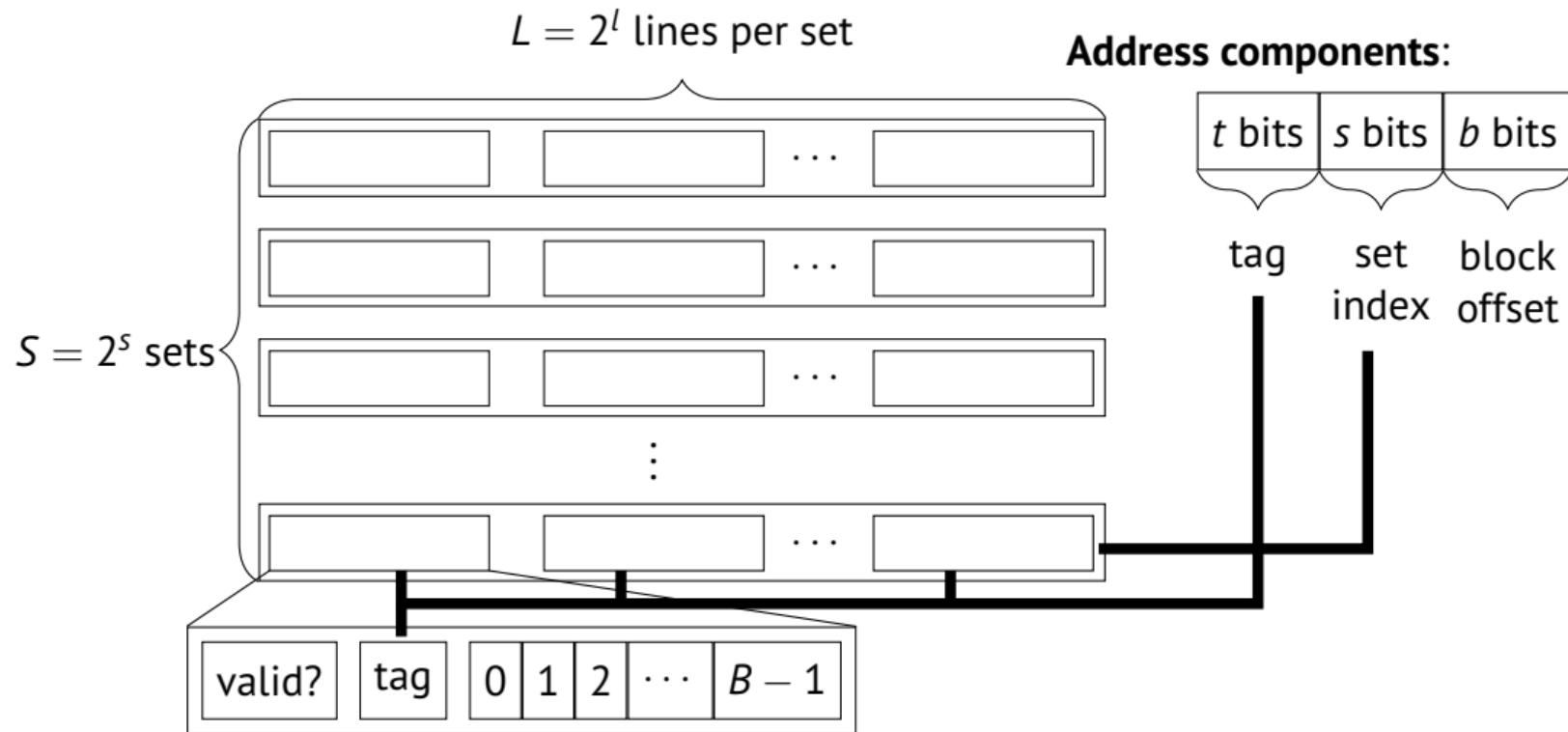
# Cache lookup



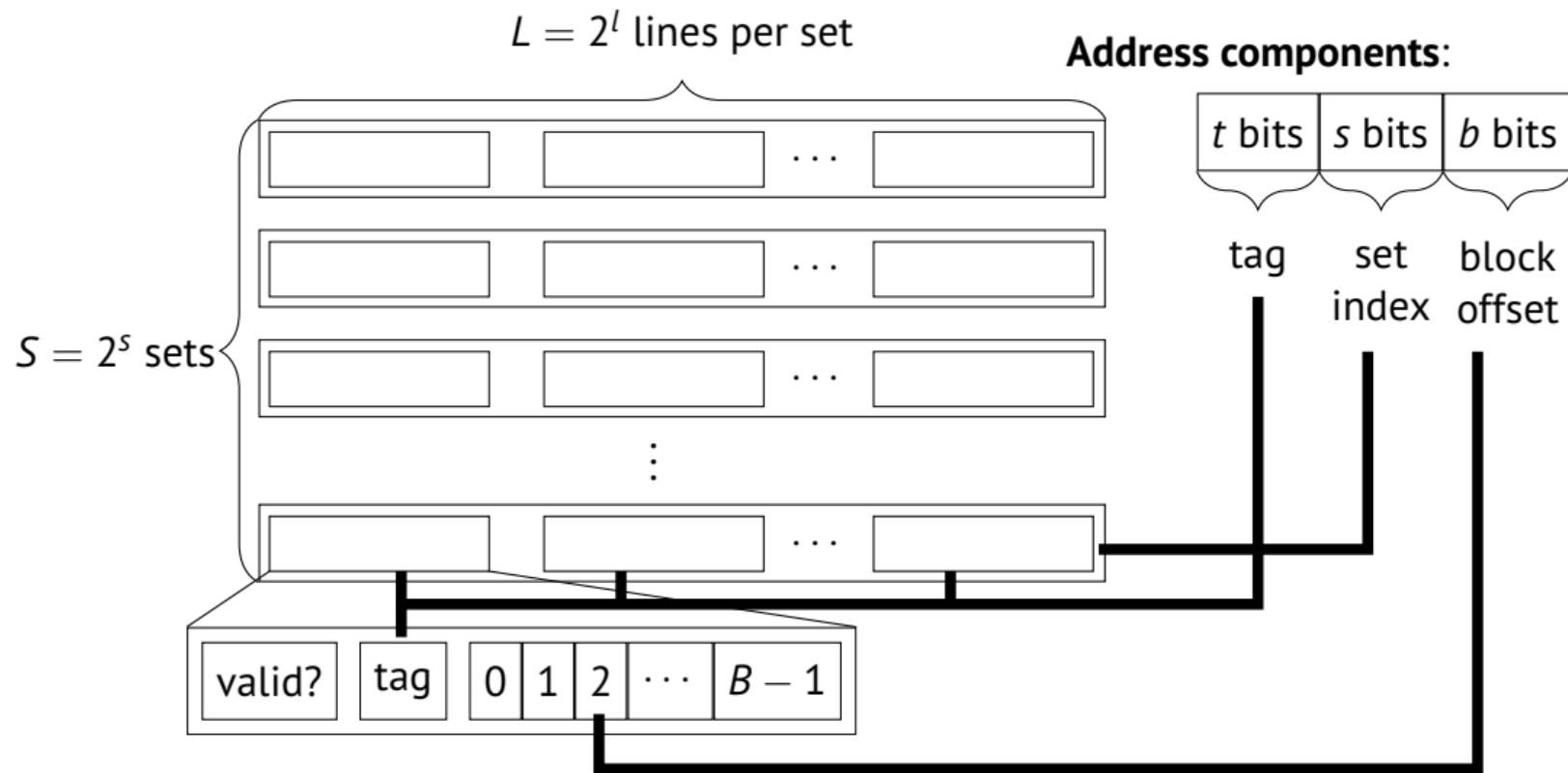
# Cache lookup



# Cache lookup

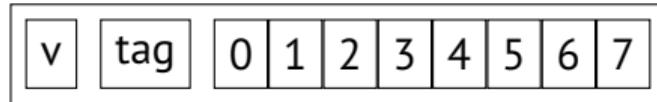
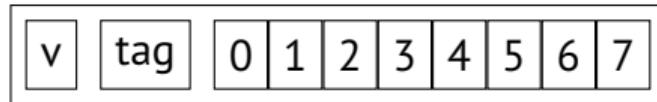
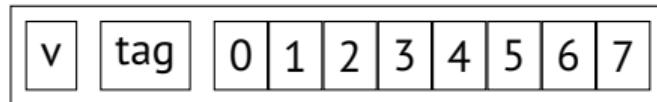
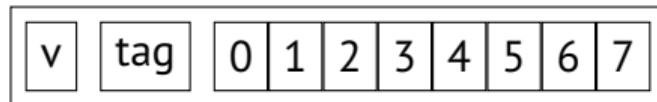


# Cache lookup



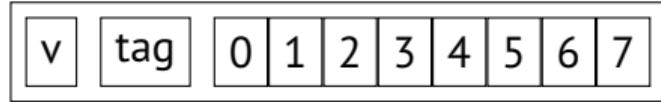
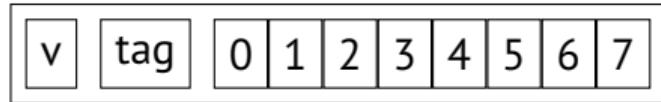
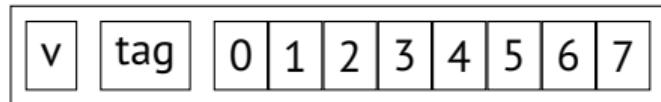
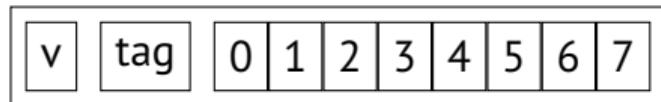
## Example: Direct-Mapped ( $L = 1$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

Suppose 10-bit addresses, so  $b = 3, s = 2, t = 6$ . **Note:** one line per set.



## Example: Direct-Mapped ( $L = 1$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

Suppose 10-bit addresses, so  $b = 3, s = 2, t = 6$ . **Note:** one line per set.

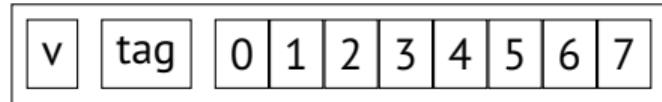
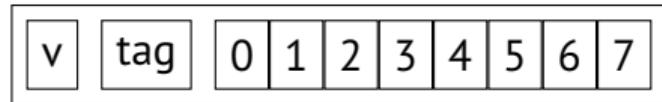
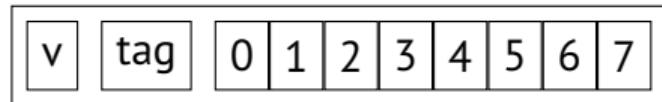
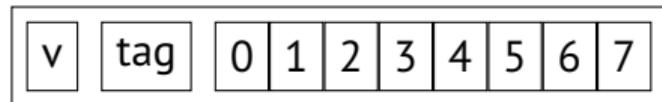


**Address of 4-byte word**

101010	10	100
--------	----	-----

## Example: Direct-Mapped ( $L = 1$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

Suppose 10-bit addresses, so  $b = 3, s = 2, t = 6$ . Note: one line per set.



Address of 4-byte word

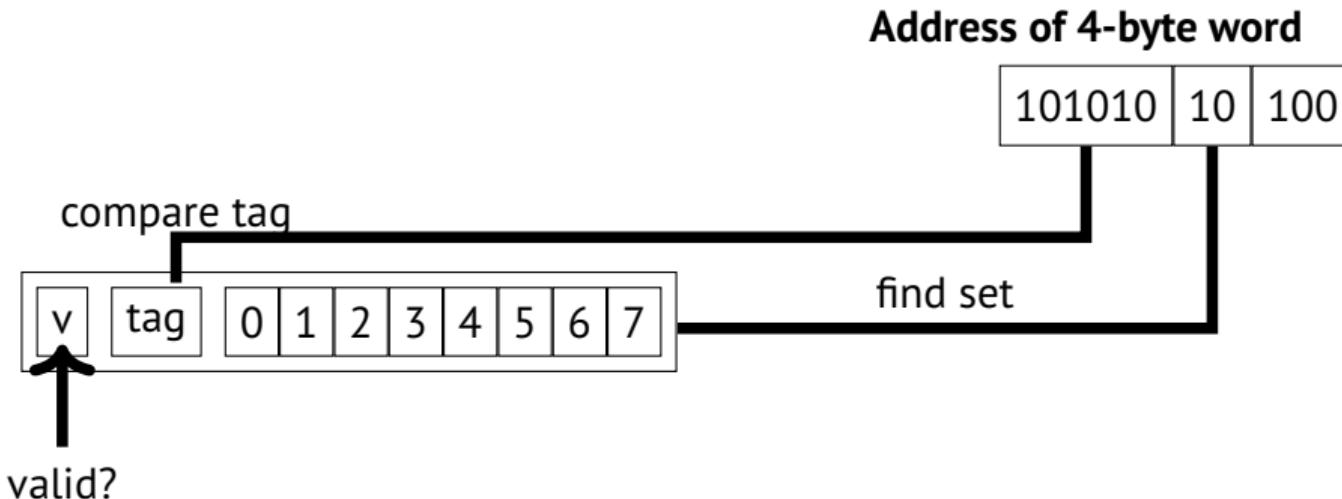
101010	10	100
--------	----	-----

find set



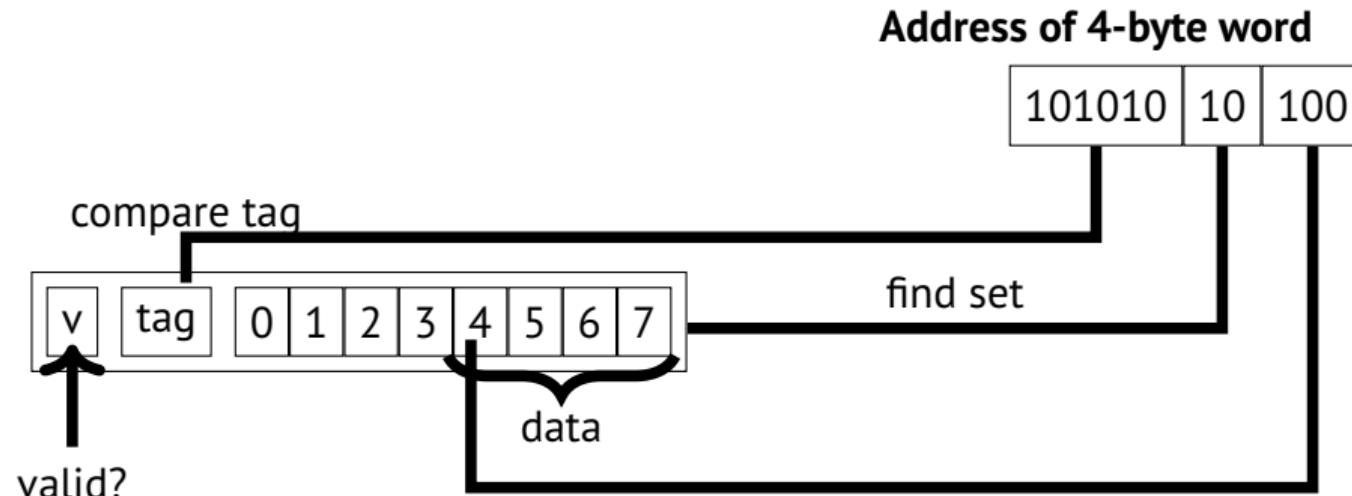
# Example: Direct-Mapped ( $L = 1$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

Suppose 10-bit addresses, so  $b = 3, s = 2, t = 6$ . **Note:** one line per set.



# Example: Direct-Mapped ( $L = 1$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

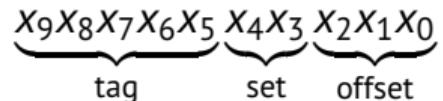
Suppose 10-bit addresses, so  $b = 3, s = 2, t = 6$ . **Note:** one line per set.



# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



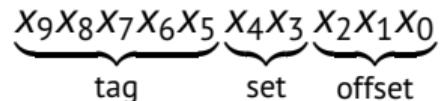
## Simulation, reading single bytes from address

	Valid	Tag	Block
<b>Set 0</b>	0		
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

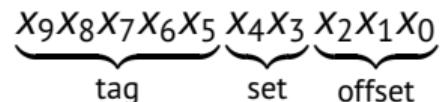
■ 00000 00 000

	Valid	Tag	Block
<b>Set 0</b>	0		
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

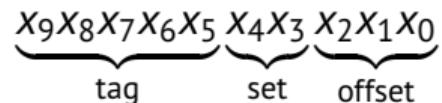
- 00000 00 000 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

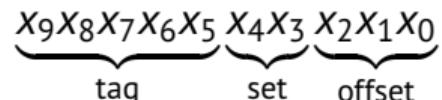
- 00000 00 000 **Miss**
- 00000 00 001

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

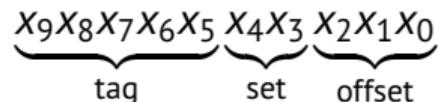
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

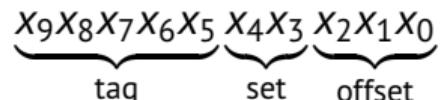
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01100 11 100

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	0		

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

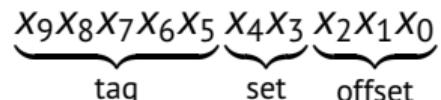
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01100 11 100 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	1	01100	Mem[280-287]

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

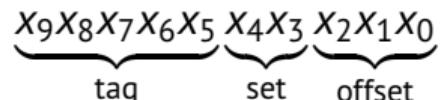
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01000 11 100 **Miss**
- 00001 00 000

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	1	01000	Mem[280-287]

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

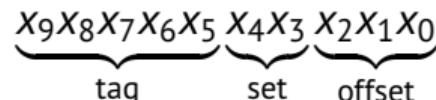
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01000 11 100 **Miss**
- 00001 00 000 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	00001	Mem[32-39]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	1	01000	Mem[280-287]

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

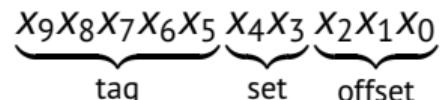
- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01000 11 100 **Miss**
- 00001 00 000 **Miss**
- 00000 00 000

	Valid	Tag	Block
<b>Set 0</b>	1	00001	Mem[32-39]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	1	01000	Mem[280-287]

# Simulation of Direct-Mapped Cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 4$ ,  $L = 1$ .



## Simulation, reading single bytes from address

- 00000 00 000 **Miss**
- 00000 00 001 **Hit**
- 01000 11 100 **Miss**
- 00001 00 000 **Miss**
- 00000 00 000 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	00000	Mem[0-7]
<b>Set 1</b>	0		
<b>Set 2</b>	0		
<b>Set 3</b>	1	01000	Mem[280-287]

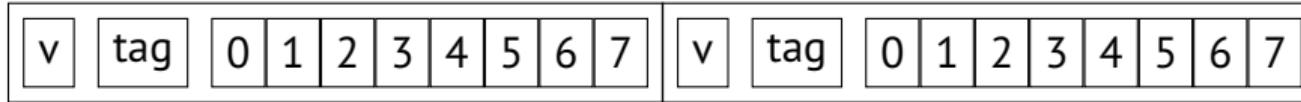
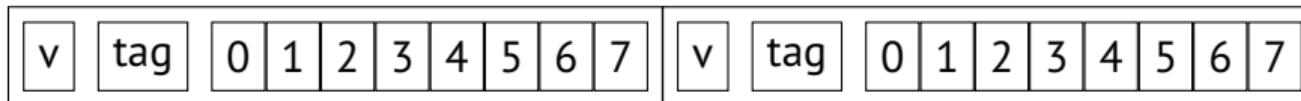
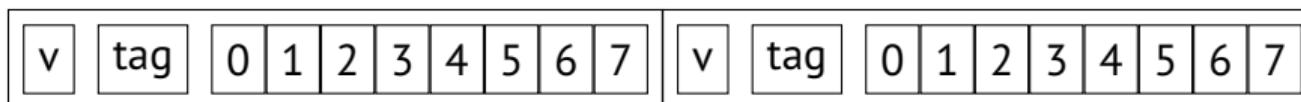
## Example: Set-associative ( $L = 2$ ), with 4 sets and 8-byte blocks ( $B = 8$ )



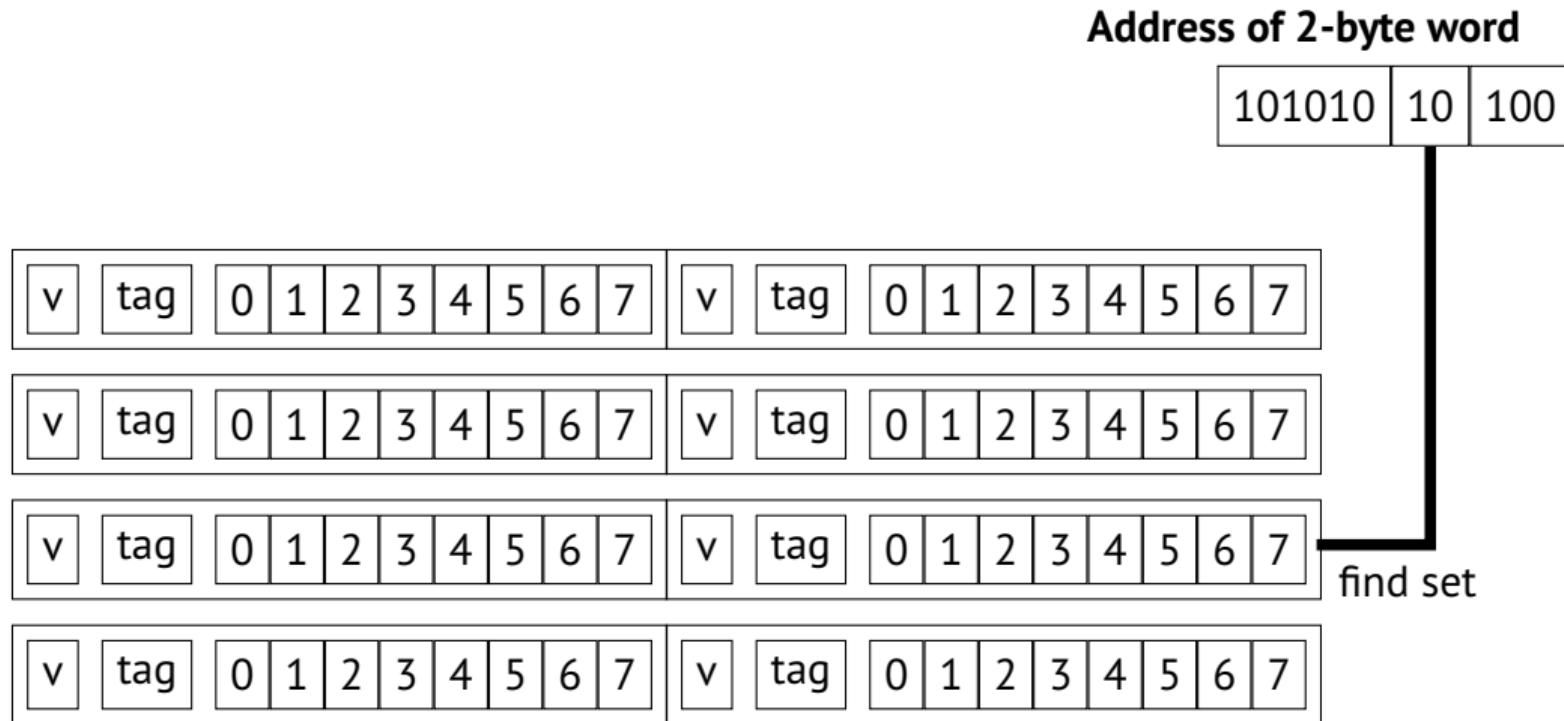
## Example: Set-associative ( $L = 2$ ), with 4 sets and 8-byte blocks ( $B = 8$ )

Address of 2-byte word

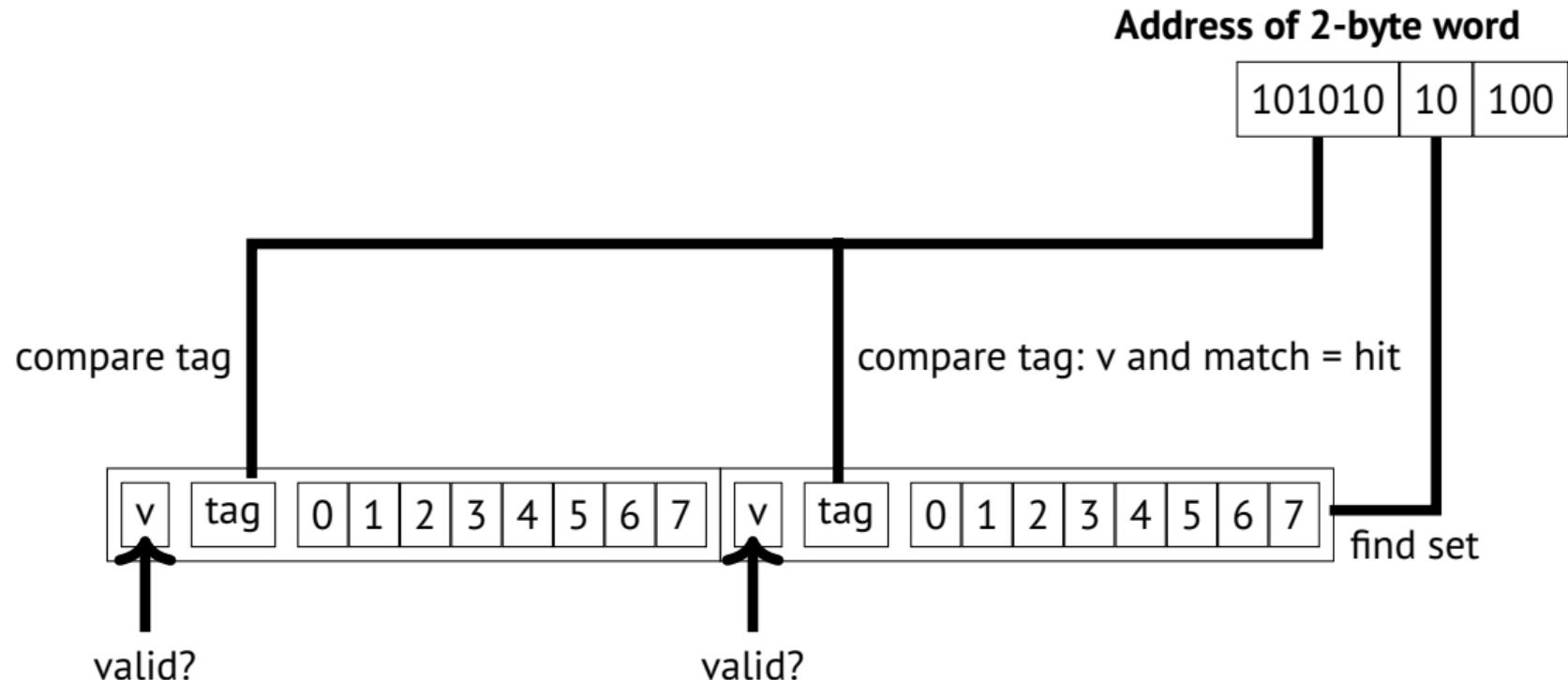
101010	10	100
--------	----	-----



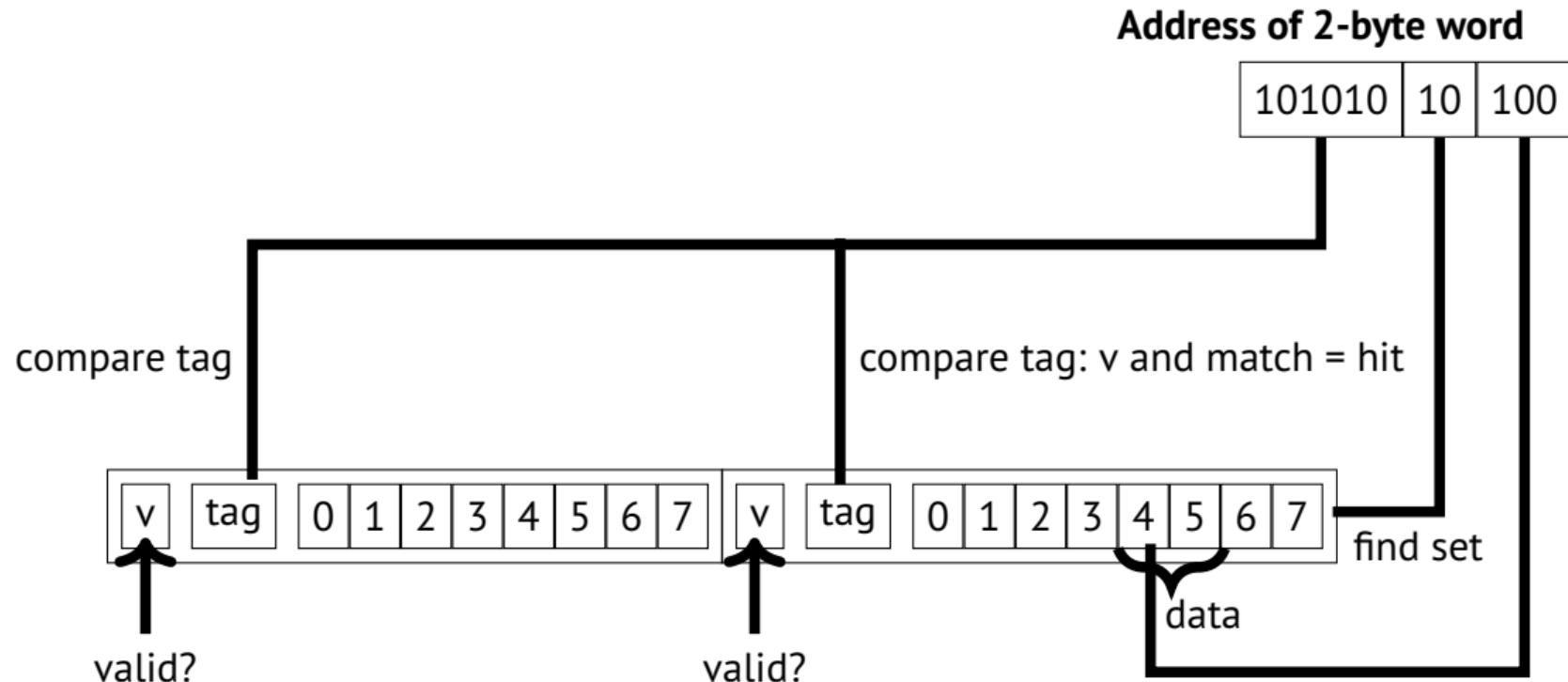
## Example: Set-associative ( $L = 2$ ), with 4 sets and 8-byte blocks ( $B = 8$ )



# Example: Set-associative ( $L = 2$ ), with 4 sets and 8-byte blocks ( $B = 8$ )



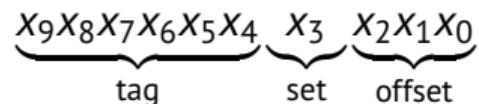
# Example: Set-associative ( $L = 2$ ), with 4 sets and 8-byte blocks ( $B = 8$ )



# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



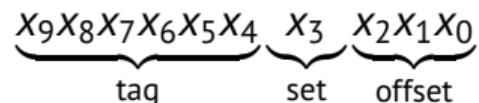
## Simulation, reading single bytes from address

	Valid	Tag	Block
Set 0	0		
	0		
Set 1	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

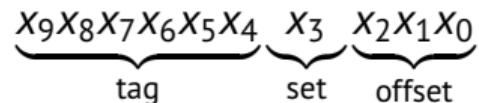
- 000000 0 000

	Valid	Tag	Block
Set 0	0		
	0		
Set 1	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

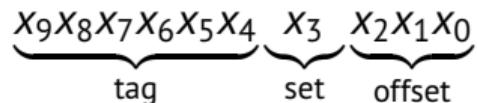
- 000000 0 000 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	000000	Mem[0-7]
	0		
<b>Set 1</b>	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

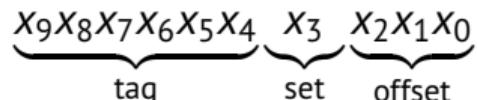
- 000000 0 000 **Miss**
- 000000 0 001

	Valid	Tag	Block
<b>Set 0</b>	1	000000	Mem[0-7]
	0		
<b>Set 1</b>	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

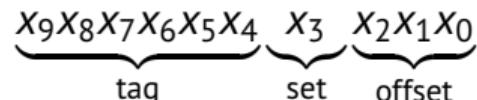
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**

	Valid	Tag	Block
Set 0	1	000000	Mem[0-7]
	0		
Set 1	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

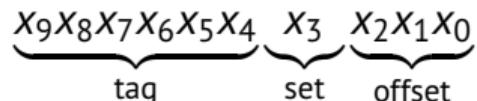
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100

	Valid	Tag	Block
<b>Set 0</b>	1	000000	Mem[0-7]
	0		
<b>Set 1</b>	0		
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

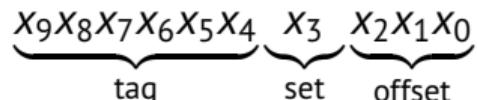
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100 **Miss**

	Valid	Tag	Block
Set 0	1	000000	Mem[0-7]
	0		
Set 1	1	010001	Mem[280-287]
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

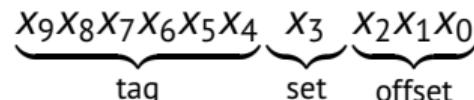
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100 **Miss**
- 000010 0 000

	Valid	Tag	Block
Set 0	1	000000	Mem[0-7]
	0		
Set 1	1	010001	Mem[280-287]
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

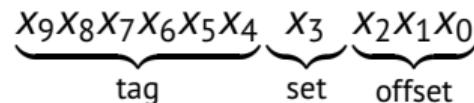
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100 **Miss**
- 000010 0 000 **Miss**

	Valid	Tag	Block
<b>Set 0</b>	1	000000	Mem[0-7]
	1	000010	Mem[32-39]
<b>Set 1</b>	1	010001	Mem[280-287]
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

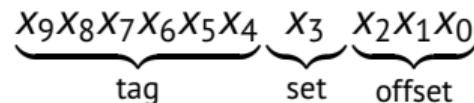
- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100 **Miss**
- 000010 0 000 **Miss**
- 000000 0 000

	Valid	Tag	Block
<b>Set 0</b>	1	000000	Mem[0-7]
	1	000010	Mem[32-39]
<b>Set 1</b>	1	010001	Mem[280-287]
	0		

# Simulation of 2-way set-associative cache

## Characteristics

10-bit addresses,  $B = 8$ ,  $S = 1$ ,  $L = 2$ .



## Simulation, reading single bytes from address

- 000000 0 000 **Miss**
- 000000 0 001 **Hit**
- 010001 1 100 **Miss**
- 000010 0 000 **Miss**
- 000000 0 000 **Hit**

	Valid	Tag	Block
Set 0	1	000000	Mem[0-7]
	1	000010	Mem[32-39]
Set 1	1	010001	Mem[280-287]
	0		

# What about writes?

## Multiple copies of data exist

- L1, L2, L3 caches, main memory, disk, backup in the cloud...

## What do we do on a write hit?

Write-through: writing immediately to the next level of the hierarchy.

Write-back: defer write until the cache block is evicted.

- Needs a *dirty bit* indicating whether block changed since it was loaded.

## What do we do on a write miss?

Write-allocate: load block into cache and update there.

- Good if more writes follow.

No-write-allocate: write straight to next level, do not load into cache.

**CPU caches are typically write-back and write-allocate.**

Locality of reference

Memory hierarchies

Cache organisation and operation

**Cache performance**

# A real cache

- Use `sudo dmidecode -t cache` or `lscpu` on Linux to see hardware details, including CPU cache specs.
- On a Ryzen 1700X, cache blocks are 64 bytes each, and
  - ▶ **L1:** 96KiB, 8-way set-associative, split into 32KiB for data (L1d) and 64KiB for instructions (L1i).
  - ▶ **L2:** 512KiB, 8-way set-associative.
  - ▶ **L3:** 16MiB, 8-way set associative.
- **But:** Each of the 8 cores have their own L1 and L2 caches, but L3 cache is shared.
  - ▶ For those who think caches are very fascinating, read up on *cache coherency protocols* and *false sharing*.

# Cache performance metrics

- |                     |  |
|---------------------|--|
| <b>Miss rate</b>    | <ul style="list-style-type: none"><li>▪ Fraction of memory references not found in cache (<math>\text{misses} \div \text{accesses}</math>).</li><li>▪ Typical numbers:<ul style="list-style-type: none"><li>▶ 3-10% for L1</li><li>▶ Can be very small (&lt; 1%) for L2.</li></ul></li></ul>   |
| <b>Hit time</b>     | <ul style="list-style-type: none"><li>▪ Time to deliver a cache block to the processor.<ul style="list-style-type: none"><li>▶ Includes time to determine whether a hit (checking tag).</li></ul></li><li>▪ Typical numbers<ul style="list-style-type: none"><li>▶ L1: 4 clock cycles.</li><li>▶ L2: 10 clock cycles.</li><li>▶ L3: 40-75 cycles (depends on sharing).</li></ul></li></ul> |
| <b>Miss penalty</b> | <ul style="list-style-type: none"><li>▪ Additional time needed because of a miss.</li><li>▪ Often over 100 cycles for main DRAM memory, historically increasing.</li></ul>   |

# Conceptualising those numbers

- Huge difference between a hit and a miss!
  - ▶ Could be 100 × between L1 and main memory.
- 99% hit rate may be twice as good as 97%
  - Cache hit time: 1 cycle
  - Miss penalty: 100 cycles
  - Average access time:
    - ▶ 97% hits:  $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$
    - ▶ 99% hits:  $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$
- This is why we use *miss rate* instead of *hit rate*.

# Writing cache friendly code

- **Make the common case go fast.**
  - ▶ Focus on inner loops.
  - ▶ Optimising non-loopy code is almost never worth it.

# Writing cache friendly code

- **Make the common case go fast.**

- ▶ Focus on inner loops.
- ▶ Optimising non-loopy code is almost never worth it.

- **Ensure good locality.**

- ▶ Avoid chasing pointers.
- ▶ If you `malloc()` a million tiny blocks of memory, there is no guarantee they will be anywhere near each other.

# Writing cache friendly code

- **Make the common case go fast.**

- ▶ Focus on inner loops.
- ▶ Optimising non-loopy code is almost never worth it.

- **Ensure good locality.**

- ▶ Avoid chasing pointers.
- ▶ If you `malloc()` a million tiny blocks of memory, there is no guarantee they will be anywhere near each other.
  - ▶ Linked lists are *terrible*.

# Writing cache friendly code

- **Make the common case go fast.**

- ▶ Focus on inner loops.
  - ▶ Optimising non-loopy code is almost never worth it.

- **Ensure good locality.**

- ▶ Avoid chasing pointers.
  - ▶ If you `malloc()` a million tiny blocks of memory, there is no guarantee they will be anywhere near each other.
    - ▶ Linked lists are *terrible*.

- **Minimise footprint.**

- ▶ E.g. if all the data you work on fits in L3 cache, then you will never see an L3 cache miss after the compulsory ones.

## Example: matrix multiplication

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

- Multiply  $n$ -by- $n$  matrices in row-major order.
- Each element is a `double`
- $O(n^3)$  total operations.
- $n$  values summed per destination.

# Miss rate analysis

- Assume**
- Cache block size  $B = 64\text{bytes}$ 
    - ▶ Enough for 8 doubles.
  - $n$  is very large—approximate  $1/n$  as 0.
  - Cache is not even big enough to hold a single row or column.
- Method**
- Look at access pattern of inner loop.

# Miss rate analysis

- Assume**
- Cache block size  $B = 64\text{bytes}$ 
    - ▶ Enough for 8 doubles.
  - $n$  is very large—approximate  $1/n$  as 0.
  - Cache is not even big enough to hold a single row or column.
- Method**
- Look at access pattern of inner loop.

When accessing successive elements miss rate is  
 $\text{sizeof}(\text{double})/B = 0.125$ .

```
for (int k=0; k<n; k++)  
    sum += a[0][k]
```

# Miss rate analysis

- Assume**
- Cache block size  $B = 64\text{bytes}$ 
    - ▶ Enough for 8 doubles.
  - $n$  is very large—approximate  $1/n$  as 0.
  - Cache is not even big enough to hold a single row or column.
- Method**
- Look at access pattern of inner loop.

When accessing successive elements miss rate is  
 $\text{sizeof}(\text{double})/B = 0.125$ .

```
for (int k=0; k<n; k++)
    sum += a[0][k]
```

When accessing distant elements miss rate is 1.

```
for (int k=0; k<n; k++)
    sum += a[k][0]
```

# Matrix multiplication (ijk)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        for (int k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (ijk)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        for (int k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
0.125	1	0

# Matrix multiplication (jik)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        for (int k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (jik)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        for (int k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
0.125	1	0

# Matrix multiplication (kij)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int k=0; k<n; k++)
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (kij)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int k=0; k<n; k++)
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
0	0.125	0.125

# Matrix multiplication (ikj)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int i=0; i<n; i++)
    for (int k=0; k<n; k++)
        for (int j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (ikj)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int i=0; i<n; i++)
    for (int k=0; k<n; k++)
        for (int j=0; j<n; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
0	0.125	0.125

# Matrix multiplication (jki)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int j=0; j<n; j++)
    for (int k=0; k<n; k++)
        for (int i=0; i<n; i++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (jki)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int j=0; j<n; j++)
    for (int k=0; k<n; k++)
        for (int i=0; i<n; i++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
1	0	1

# Matrix multiplication (kji)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int k=0; k<n; k++)
    for (int j=0; j<n; j++)
        for (int i=0; i<n; i++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a

b

c

# Matrix multiplication (kji)

- Cache block size  $B = 64\text{bytes}$  (8 doubles) –  $n$  very large –  $1/n \sim 0$
- Successive elements miss rate: 0.125
- Distant elements miss rate: 1

```
for (int k=0; k<n; k++)
    for (int j=0; j<n; j++)
        for (int i=0; i<n; i++)
            c[i][j] += a[i][k] * b[k][j];
```

Misses per innermost loop iteration

a	b	c
1	0	1

# Summary of variants

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

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

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

## ijk and jik (inner loop):

- 2 loads, 0 stores
- 1.125 misses/iter

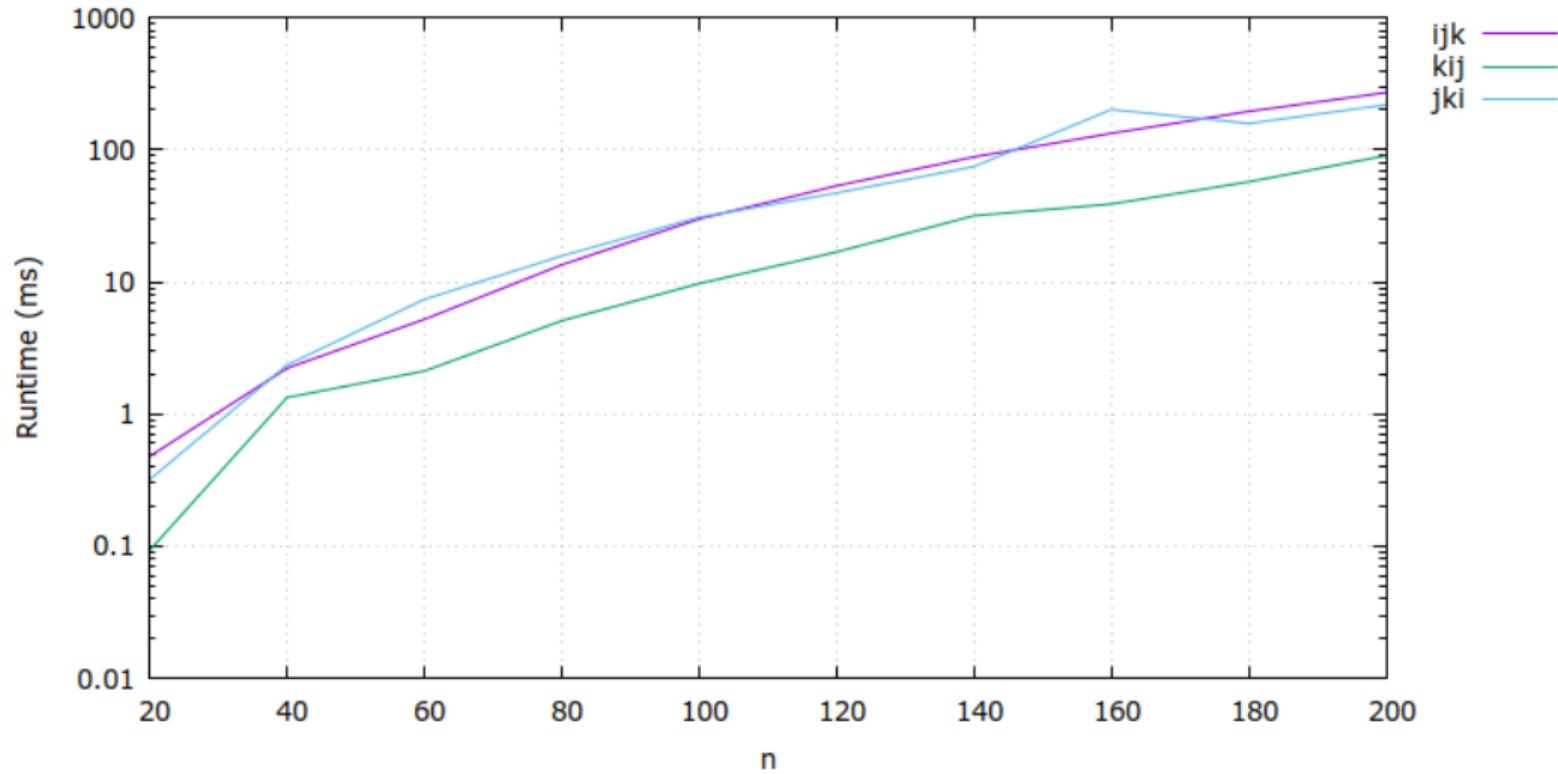
## kij and ikj (inner loop):

- 2 loads, 1 store
- 0.25 misses/iter

## jki and kji (inner loop):

- 2 loads, 1 store
- 2 misses/iter

# On a real machine



# What did we learn today?

# What did we learn today?

## What I tried to teach

- The notions of *temporal* and *spatial* locality.
- What the memory hierarchy is and what caching is.
- How a cache actually works, and its implications for performance.
- How to modify programs to improve their locality.

# What did we learn today?

## What I tried to teach

- The notions of *temporal* and *spatial* locality.
- What the memory hierarchy is and what caching is.
- How a cache actually works, and its implications for performance.
- How to modify programs to improve their locality.

## The most crucial points

- Memory hierarchies are part of all nontrivial systems.
- Cache misses have dramatic impact on performance.
- Significant speedup can be achieved just by permuting loops.