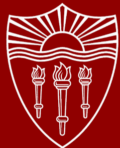
# **CS356**: Discussion #9

## Memory Hierarchy and Caches

Marco Paolieri (paolieri@usc.edu)
Illustrations from CS:APP3e textbook


USC University of Southern California

# The Memory Hierarchy

**So far...**

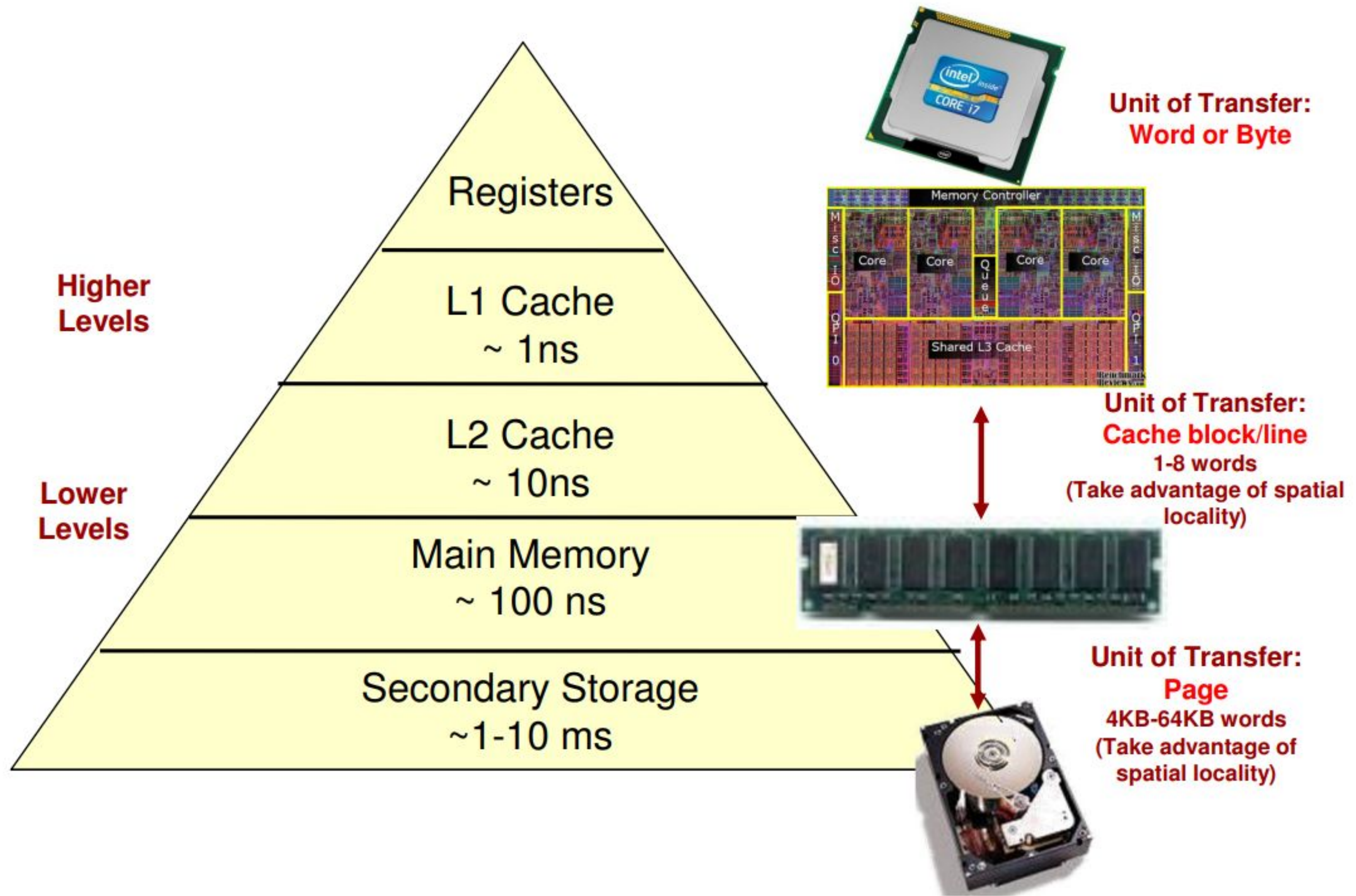
- We modeled the memory system as an abstract **array of bytes**.
- The CPU could access any location in **constant time**.

In practice, **the memory system is a hierarchy of storage devices** with different capacities, costs, and access times.

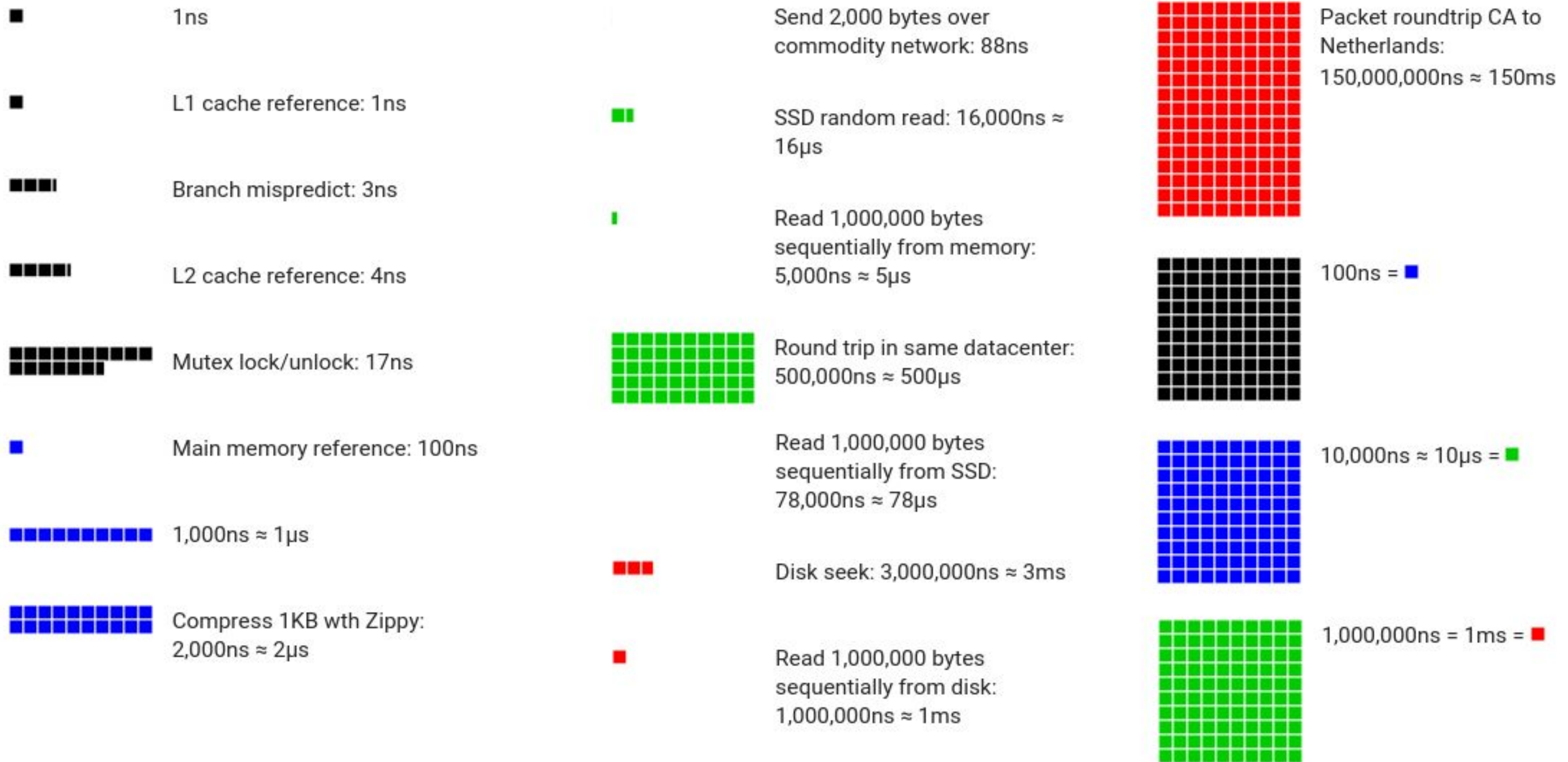
- Small, fast **cache memories close to the CPU**:
  staging area for data/instructions read from **main memory**.
- Main memory can be used as staging area for large, slow **local disks**.
- Local disks are often used as staging area for data from **network devices**.

“Well-written programs tend to access storage at any level more frequently than storage at the next lower level.”

# The Memory Hierarchy



# Latency Numbers (2018)




http://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

# Storage Technologies

**Static RAM**

- Used for cache memories (inside/outside CPU)
- Faster, more expensive: 6 transistors/bit
- Resistant to noise, persistent (bistable cells)
- About 10 MB on a desktop computer

**Dynamic RAM**

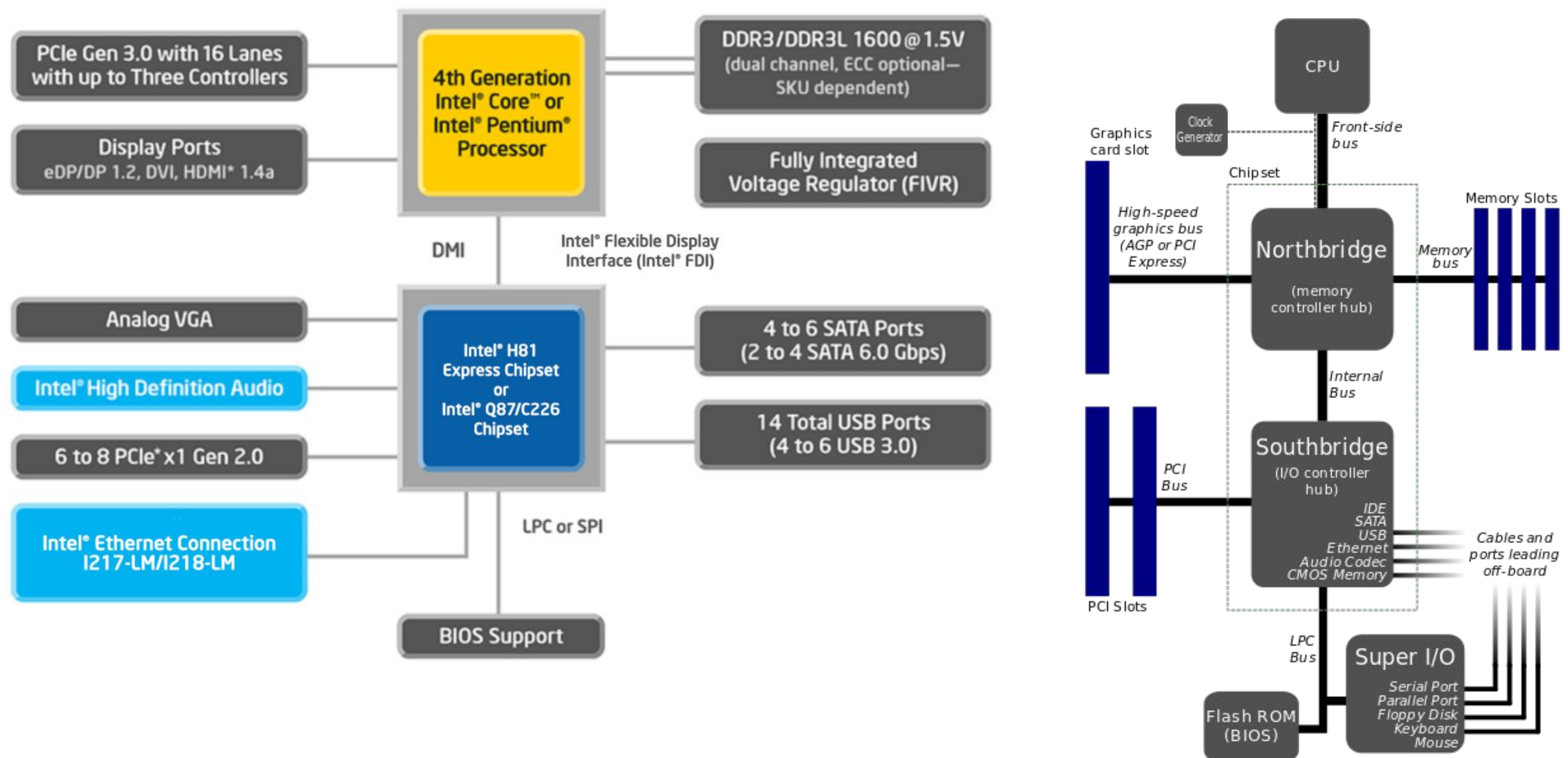
- Used for main memory and frame buffer of GPUs
- Very sensitive to noise, even light (array of capacitors)
- Must be periodically refreshed (recharge capacitors)
- 1000× cheaper, 10× slower
- About 16 GB on a desktop computer

**DRAM Organization**

- Bidimensional array of supercells *(i, j)* each storing *w* bits
- Memory controller sends RAS / CAS (on same pins), reads/writes values

# System Bus: Memory and I/O Access

- Once: northbridge (RAM / PCIe) and southbridge (PCI / IDE / SATA / USB).
- Since Sandy Bridge: northbridge functions integrated into CPU die.



# Locality

**Temporal Locality**
“A memory location referenced once is likely to be referenced again multiple times in the near future.”

**Spatial Locality**
“Nearby memory locations are likely to be referenced in the near future.”

Locality is exploited at all levels:

- **Hardware level:** CPU cache memories for main memory access.
- **OS level:** use main memory as cache for virtual memory or disk blocks.
- **Application level:**
    - Web browsers caching page elements.
    - Web servers caching frequently accessed pages/images in memory.

# Example of Data Locality

```c
#include <stdio.h>

int sum(int *array, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
      sum += array[i];
    }
    return sum;
}

int main() {
    int array[5] = {1, 2, 3, 10, 20};
    printf("%d\n", sum(array, 5));

    return 0;
}
```

The function `sum(int *array, int size)` has **good locality**:

- Variable `sum` is referenced once in every loop cycle: good temporal locality.
- The elements of `array` are read sequentially: good spatial locality.

**Stride-k reference pattern**: visiting every k-th element of an array.

- The smaller the stride, the better the spatial locality.

# Sum by row or by column?

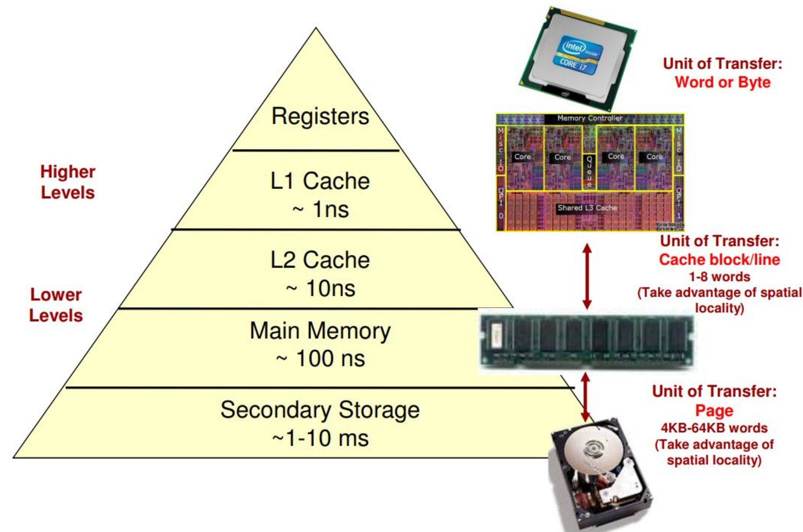
```c
#include <stdio.h>
#define N 3

int sum_by_row(int matrix[N][N]) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          sum += matrix[i][j];
      }
    }
    return sum;
}

int sum_by_col(int matrix[N][N]) {
    int sum = 0;
    for (int j = 0; j < N; j++) {
      for (int i = 0; i < N; i++) {
          sum += matrix[i][j];
      }
    }
    return sum;
}
```

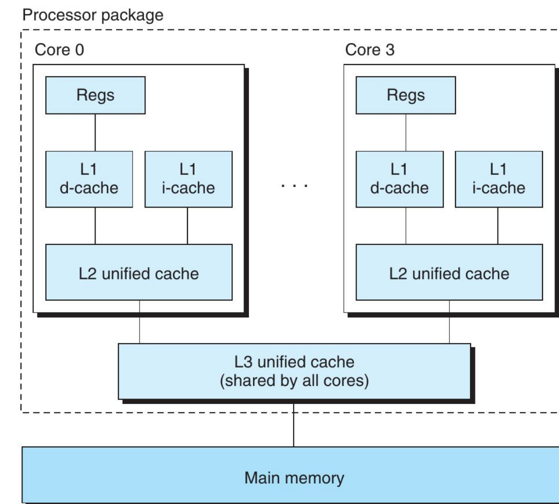
Function `sum_by_row(int matrix[N][N])` has better space locality.

- Multidimensional arrays are stored in row-major format in C.
- `sum_by_row` results in stride-1 accesses
- `sum_by_col` results in stride-N accesses

Loops also have great temporal and spatial locality with respect to **instruction fetches**.

# Caching




A **cache** is a small, fast staging area for objects from a larger, slower device.

**Cache Hit.** When an object in found in the cache of a level.
**Cache Miss.** When an object is fetched from the next level.

- In turn, another cache miss can be caused in the next level.
- Once data is fetched, an **eviction policy** decides whether to store the data in the cache and, if the cache is full, which block to evict (random, LRU).
- Data is transferred between levels in **blocks** containing many objects:
  - If another object in the block is required, it will be already in the cache.

# Types of Cache Misses

**Compulsory Miss (or Cold Miss)**
When the cache is initially empty (or “cold”).

**Conflict Miss**
When there are restrictive **placement policies** inside the cache, and two referenced data map to the same cache block.

**Capacity Miss**
When the cache is not large enough for the “working set” of a program phase.

**Who takes care of cache misses?**

- Compiler manages the register file.
- CPU L1, L2, L3 caches are managed by hardware logic.
- For virtual memory, DRAM is managed by the OS and by the address translation hardware of the CPU.

# Cache Organization

1 valid bit per line
$t$ tag bits per line
$B = 2^b$ bytes per cache block

Set 0:
Valid | Tag | 0 | 1 | · · · | $B-1$
⋮
Valid | Tag | 0 | 1 | · · · | $B-1$

$K$ lines per set

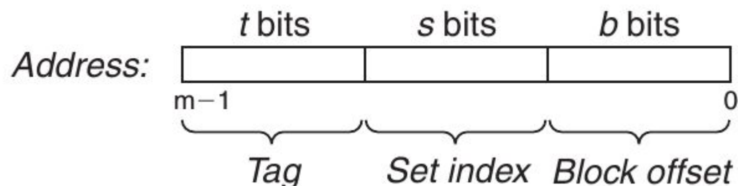
Set 1:
Valid | Tag | 0 | 1 | · · · | $B-1$
⋮
Valid | Tag | 0 | 1 | · · · | $B-1$

$S = 2^s$ sets

⋮

Set $S-1$:
Valid | Tag | 0 | 1 | · · · | $B-1$
⋮
Valid | Tag | 0 | 1 | · · · | $B-1$

Cache size: $C = B \times K \times S$ data bytes



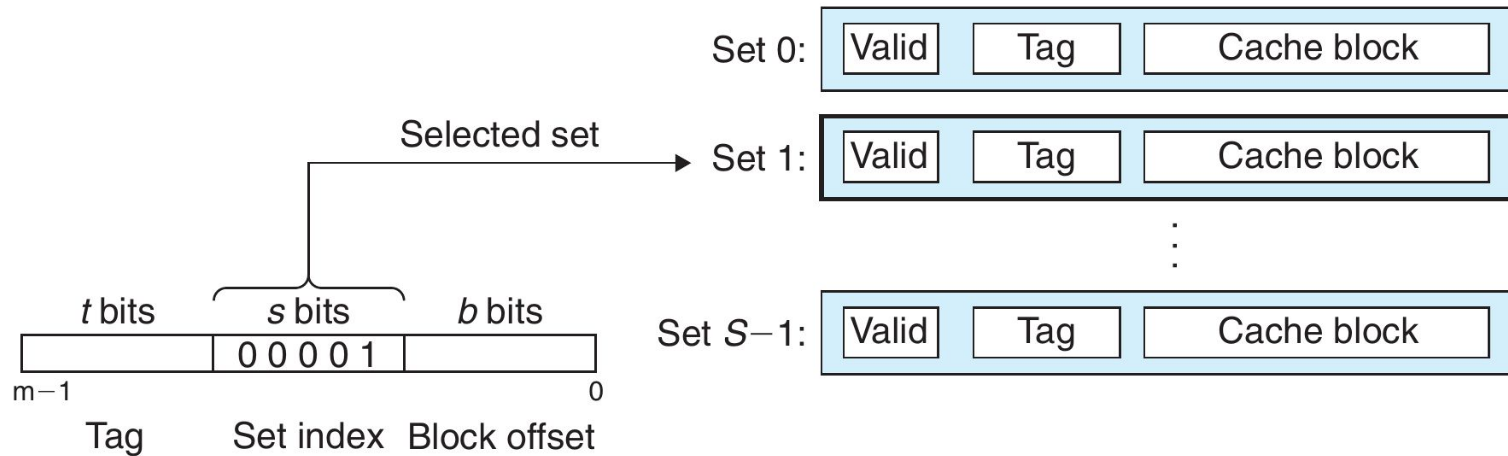

**Memory**: addresses of $m$ bits
⇒ $M = 2^m$ memory locations

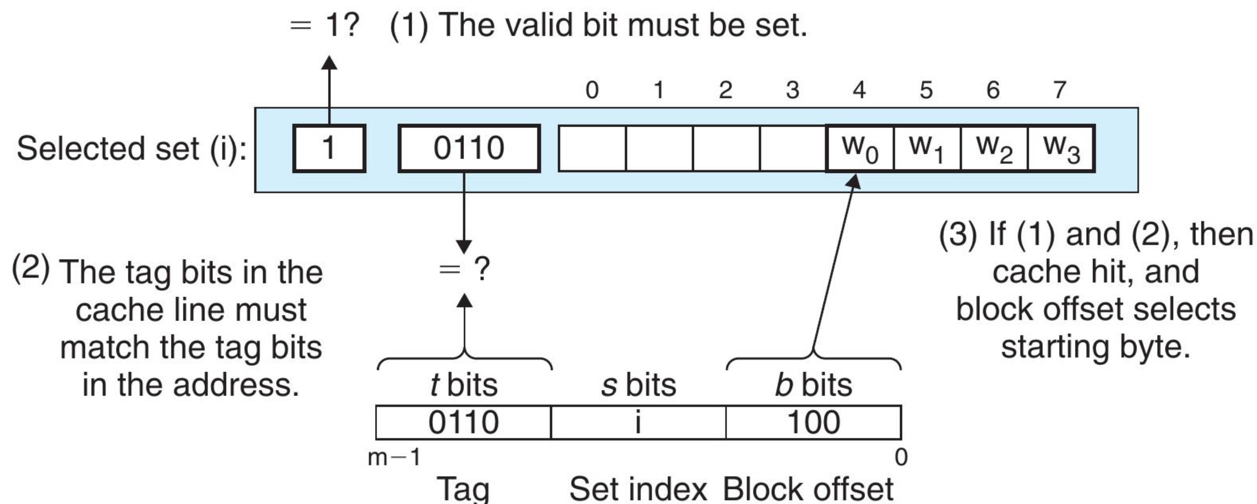
**Cache**:

- $S = 2^s$ **cache sets**
- Each set has $K$ **lines**
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

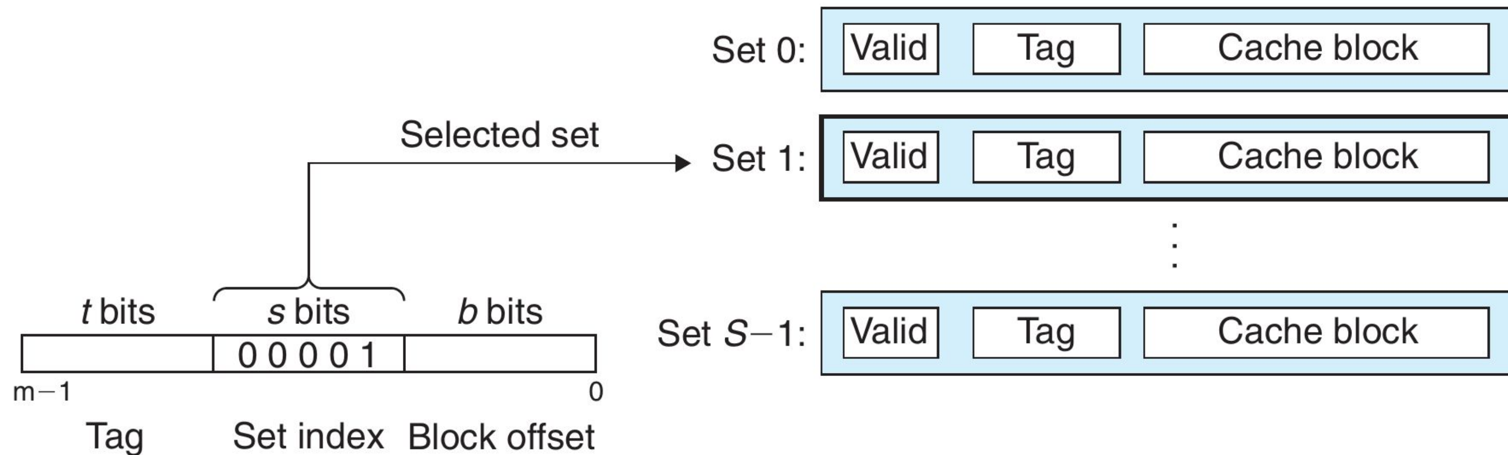
**How to check if the word at an address is in the cache?**

# Direct-Mapped Caches ($K$ = 1)



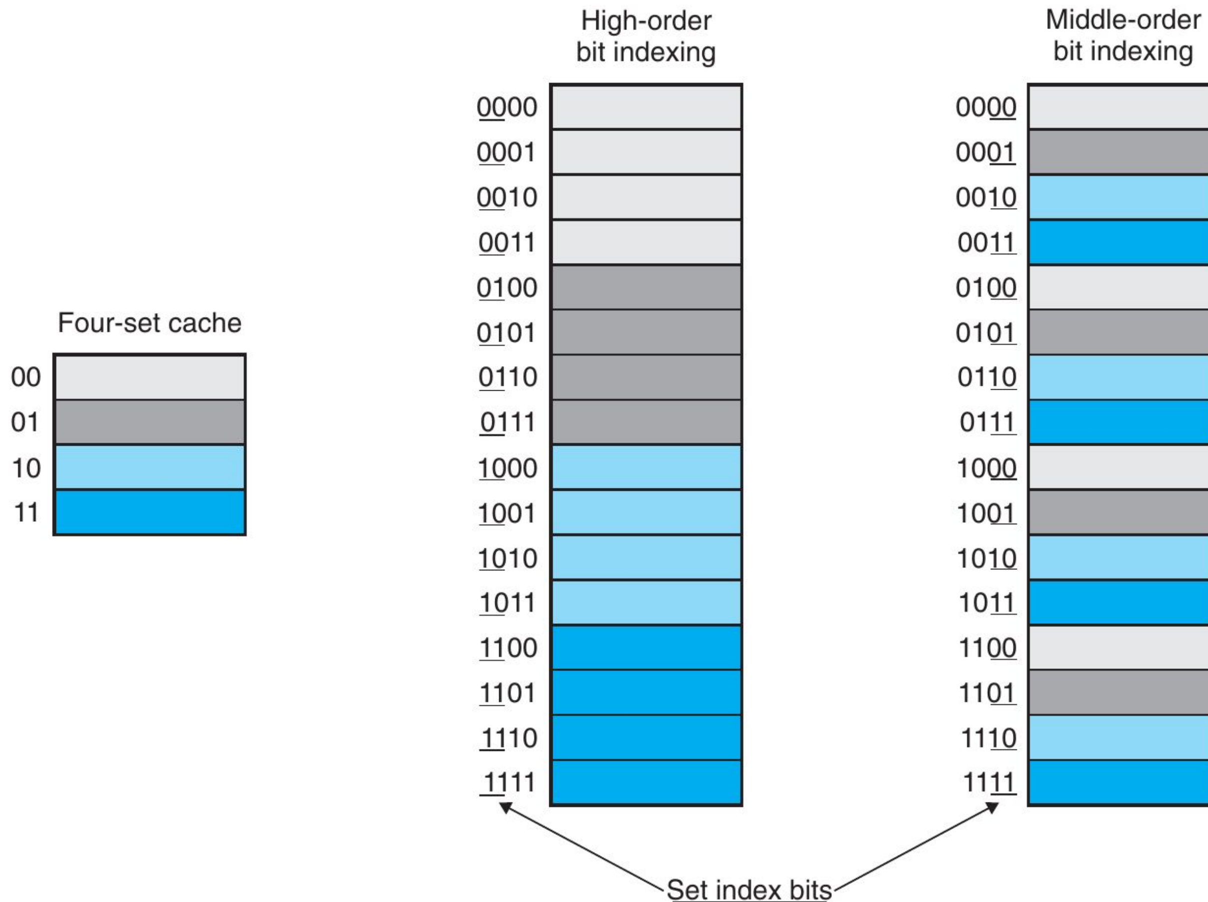
# Conflict Misses in Direct-Mapped Caches




**Multiple memory blocks can map to the same set/line!**

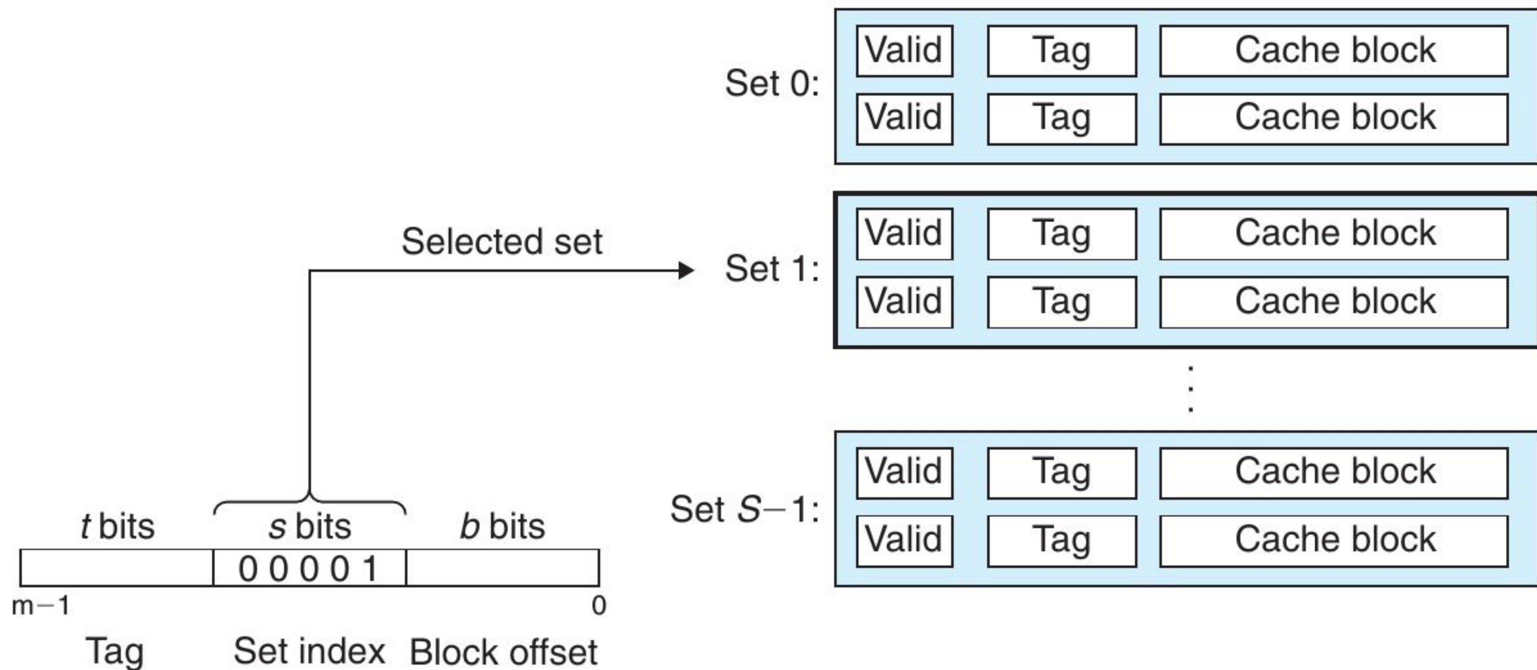
- They are identified by different tags.
- They generate conflict misses.
- Eviction policy is trivial: must remove the only line in the set.

# Why caches index with middle bits




With high-order bit indexing, contiguous memory blocks would map to the same cache set.

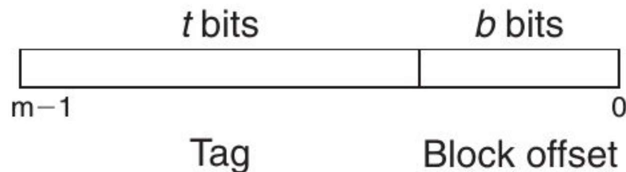
# K-way Set Associative Caches (1 < K < C/B)

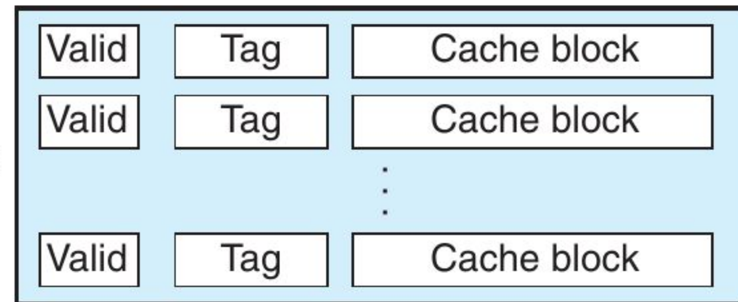



**More than one line per set.**

- Line-matching is more difficult: must check the tag of multiple lines.
- Requires a policy for cache eviction (when all lines in set are full).
  - Random, FIFO, least-recently used (LRU), least-frequently used (LFU).

# Fully Associative Caches ($K = C/B$)




**A single set contains all the cache lines.**

- Line-matching is very difficult: must check the tags of all lines.
- Appropriate for small caches (e.g, TLBs in virtual memory buffers).

# Memory Writes and Caching

**Write hit** (writing to a word in the cache). Two options:

- **Write-through:** immediately update the word in the next cache level.
- **Write-back:** wait until the word is evicted from this cache level.
    - Can significantly reduce traffic on the bus.
    - Additional complexity (needs a “dirty bit”).
    - More common at lower levels (e.g., virtual memory).
    - Also used for Intel L1.

**Write miss** (the word is not in the cache). Two options:

- **Write-allocate:** load the word from next cache level, then write.
- **No-write-allocate:** bypass the cache and write directly into the next level.

Write-through caches are typically no-write-allocate.
Write-back caches are typically write-allocate.

# Performance Tuning of Caches

**Average Access Time** = (**Hit Time**) + (**Miss Ratio**) × (**Miss Penalty**)

- Large caches decrease the miss ratio, but increase the hit time.
- Large blocks decrease the miss ratio with spatial locality, but having fewer lines per set can hurt programs where temporal locality dominates.
- Large blocks can also increase the miss penalty.
- Large associativity *K* decreases the chance of conflict misses, but it is more expensive to implement and hard to make fast.
  - More tag bits per line.
  - Additional LRU state bits per line.
  - Additional control logic.
  - … can increase both hit time and miss penalty.

# Exercise: Cache Size and Address

**Problem**

A processor has a **32-bit** memory address space. The memory is broken into blocks of **32 bytes** each. The cache is capable of storing **16 kB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **4-way set-associative cache**.

**Solution**

- 16 kB / 32 bytes per block = 512 blocks.
- Direct-mapping: 18-bit tag (rest), 9-bit set address, 5-bit block offset.
- 4-way set-associative: each set has 4 lines, so there are 512 / 4 = 128 sets.
  - 20-bit tag (rest)
  - 7-bit set address
  - 5-bit block offset

# Exercise: Cache Size and Address

**Problem**

A processor has a **36-bit** memory address space. The memory is broken into blocks of **64 bytes** each. The cache is capable of storing **1 MB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **8-way set-associative cache**.

**Solution**

- 1 MB / 64 bytes per block = 2**(20-6) = 16k blocks.
- Direct-mapping: 16-bit tag (rest), 14-bit set address, 6-bit block offset.
- 8-way set-associative: each set has 8 lines, so there are 16k / 8 = 2k sets
  - 19-bit tag (rest)
  - 11-bit set address
  - 6-bit block offset

# Exercise: Direct-Mapping Performance

You are asked to optimize a cache capable of storing **8 bytes** total for the given references. There are three direct-mapped cache designs possible by varying the block size:

- C1 has one-byte blocks,
- C2 has two-byte blocks, and
- C3 has four-byte blocks.

In terms of miss rate, which cache design is best?

If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design? (Every access, hit or miss, requires an access to the cache.)

**Trace (LSB)**

```
  1 0000 0001
134 1000 0110
212 1101 0100
  1 0000 0001
135 1000 0111
213 1101 0101
162 1010 0010
161 1010 0001
  2 0000 0010
 44 0010 1100
 41 0010 1001
221 1101 1101
```

# Solution: Direct-Mapping Performance

**Address breakdown**

- C1 has no block offset, 3-bit set address
- C2 has 1-bit block offset, 2-bit set address
- C3 has 2-bit block offset, 1-bit set address

**How to run a trace**: extract set address (3, 2, 1 bits) from LSB; on miss, load (1, 2, 4) bytes.

**Running C3:**

- **Get 1: miss.** Put bytes 0-3 in bucket 0.
- **Get 134: miss.** Put 132-135 in bucket 1.
- **Get 212: miss.** Put 212-215 in bucket 1.
- **Get 1: hit.**
- **Get 135: miss.** Put 132-135 in bucket 1.
- **Get 213: miss.** Put 212-215 in bucket 1.
- **Get 162: miss.** Put 160-163 in bucket 0.
- **Get 161: hit.**

**Trace**

| MEM | LSB | C1 | C2 | C3 |
|---|---|---|---|---|
| **1** | 0000 0001 | 1m | 0m | 0m |
| **134** | 1000 0110 | 6m | 3m | 1m |
| **212** | 1101 0100 | 4m | 2m | 1m |
| **1** | 0000 0001 | 1h | 0h | 0h |
| **135** | 1000 0111 | 7m | 3h | 1m |
| **213** | 1101 0101 | 5m | 2h | 1m |
| **162** | 1010 0010 | 2m | 1m | 0m |
| **161** | 1010 0001 | 1m | 0m | 0h |
| 2 | 0000 0010 | 2m | 1m | 0m |
| 44 | 0010 1100 | 4m | 2m | 1m |
| 41 | 0010 1001 | 1m | 0m | 0m |
| 221 | 1101 1101 | 5m | 2m | 1m |

m_rate: 11/12 9/12 10/12

# Solution: Performance

**Average Access Time** = (**Hit Time**) + (**Miss Rate**) × (**Miss Penalty**)

In terms of miss rate, C2 is best.

If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design? (Every access, hit or miss, requires an access to the cache.)

- For C1, access time = 2 + 11/12 × 25 = 24.92 cycles
- For C2, access time = 3 + 9/12 × 25 = 21.75 cycles
- For C3, access time = 5 + 10/12 × 25 = 25.83 cycles