

113: Architecture

Spring 2018

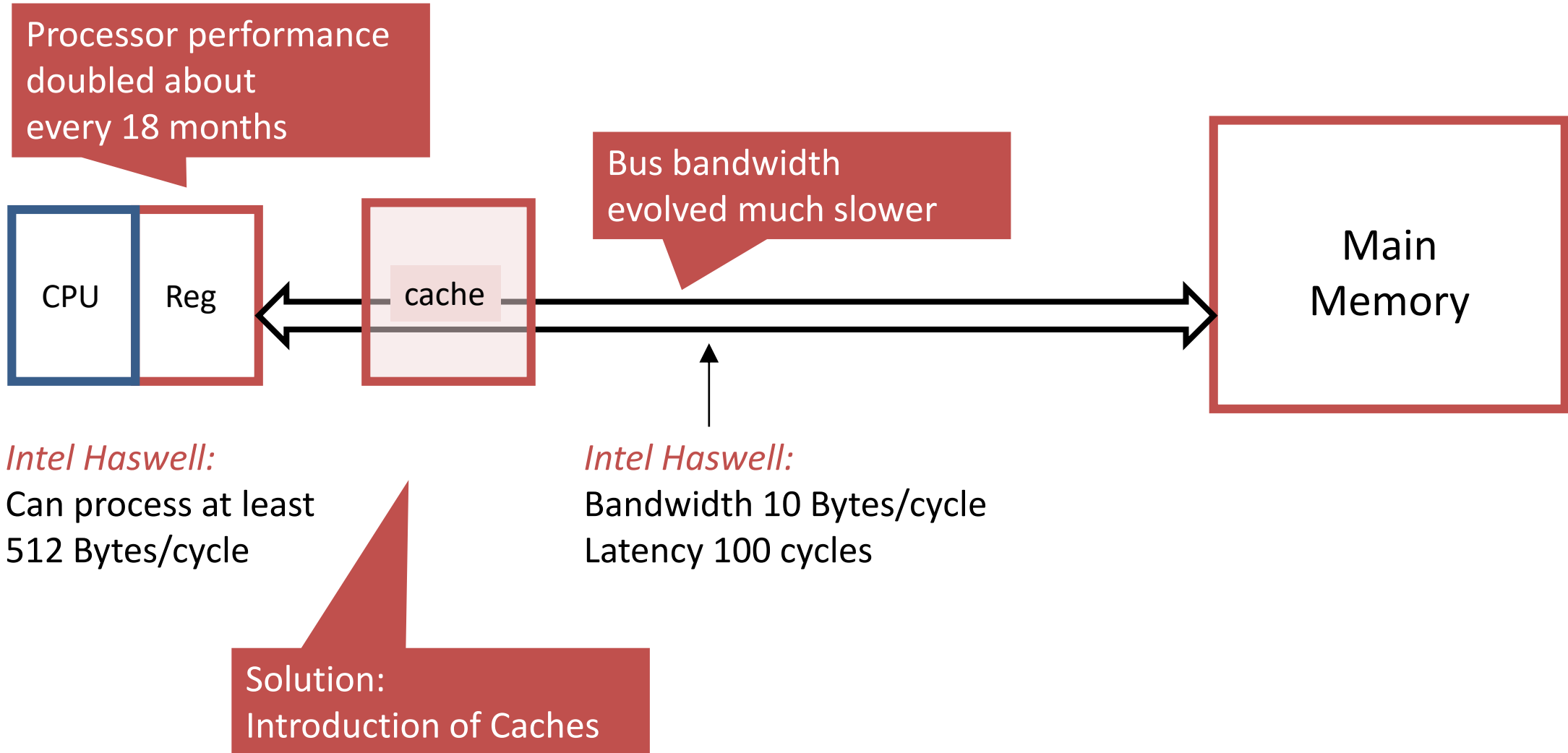
Lecture: Caches (Memory Hierarchy)

Instructor: Dr. Jana Giceva

Today: Caches and Memory Hierarchy

- **Cache basics**
- Principle of locality
- Memory hierarchies
- Cache organization
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches

Problem: Processor-memory bottleneck

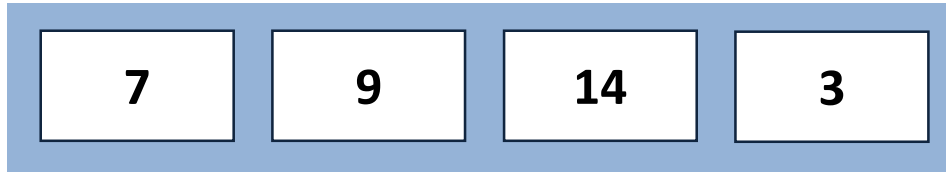


Cache

- ***Pronunciation:*** “cash”. We will abbreviate it as “\$”
- ***Definition (computer):***
Computer memory with short access time used for the storage of frequently or recently used instructions (code) or data.
- ***Definition (more general computing):***
Used to optimise data transfers between any system elements with different characteristics (e.g., network interface cache, I/O cache, etc.)

General Cache Mechanics

Cache

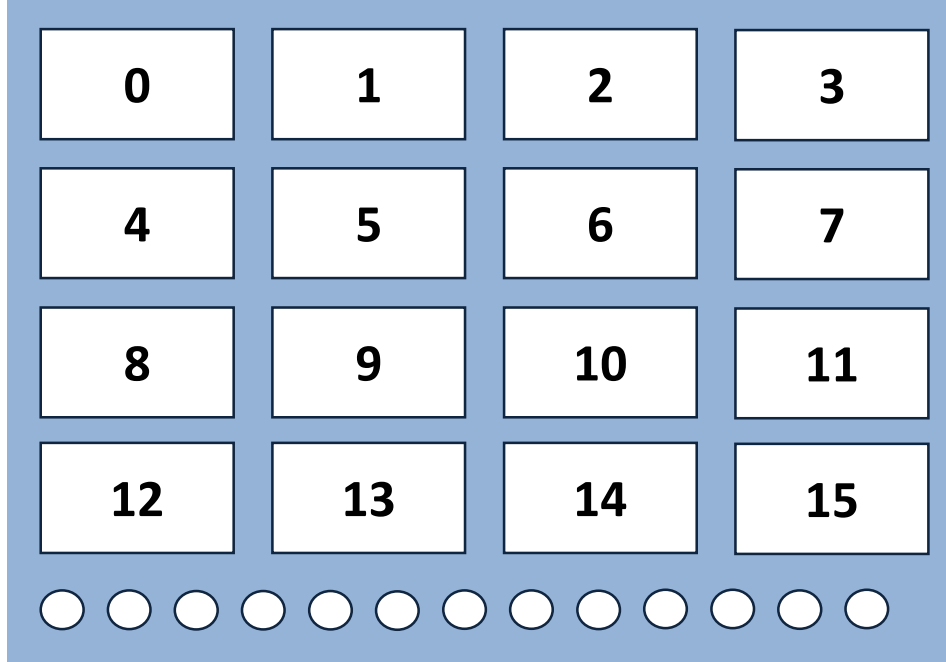


- Smaller, faster, more expensive memory
- Caches a subset of the blocks (aka *lines*)



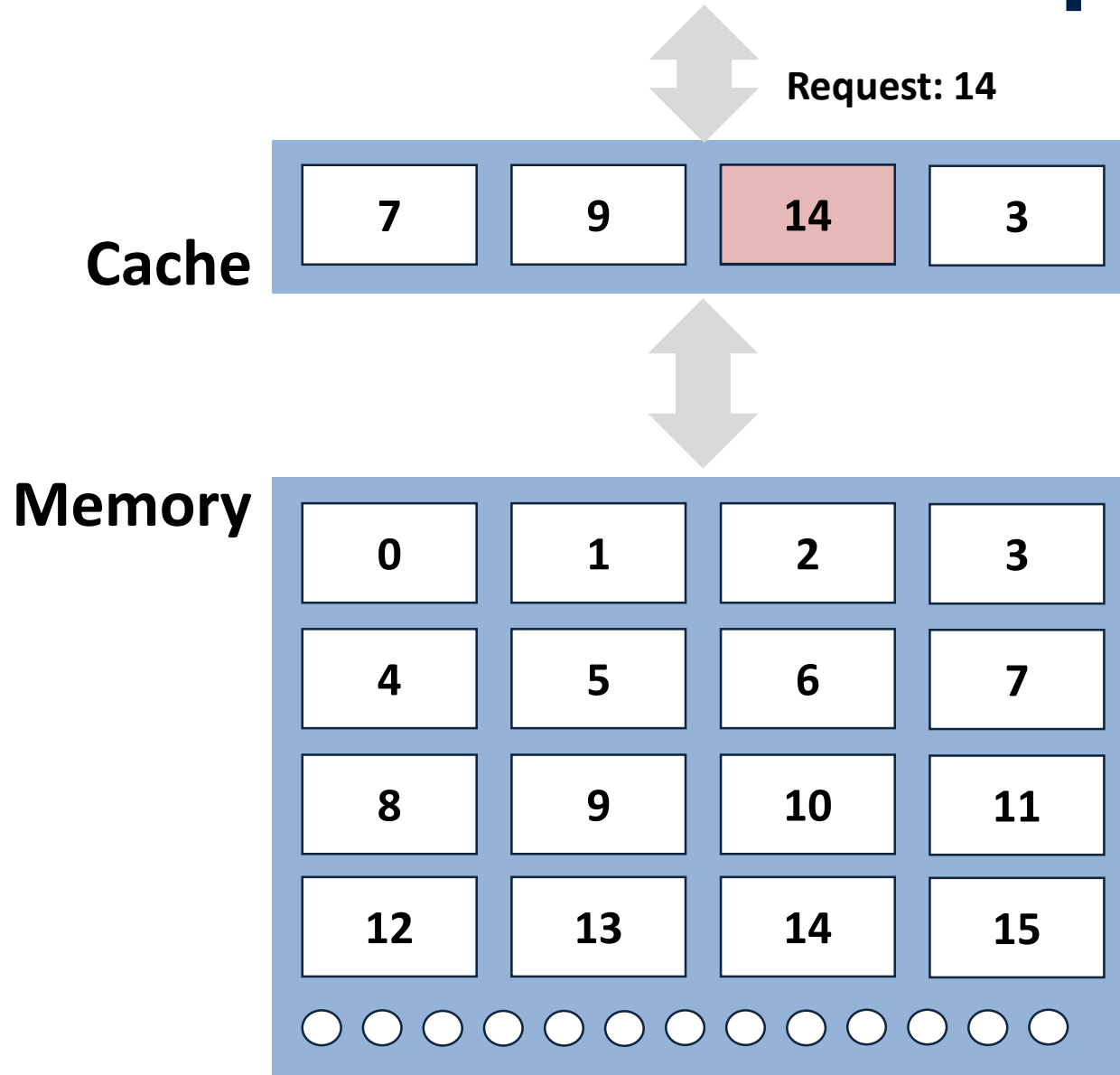
Data is copied in *block*
sized transfer units

Memory



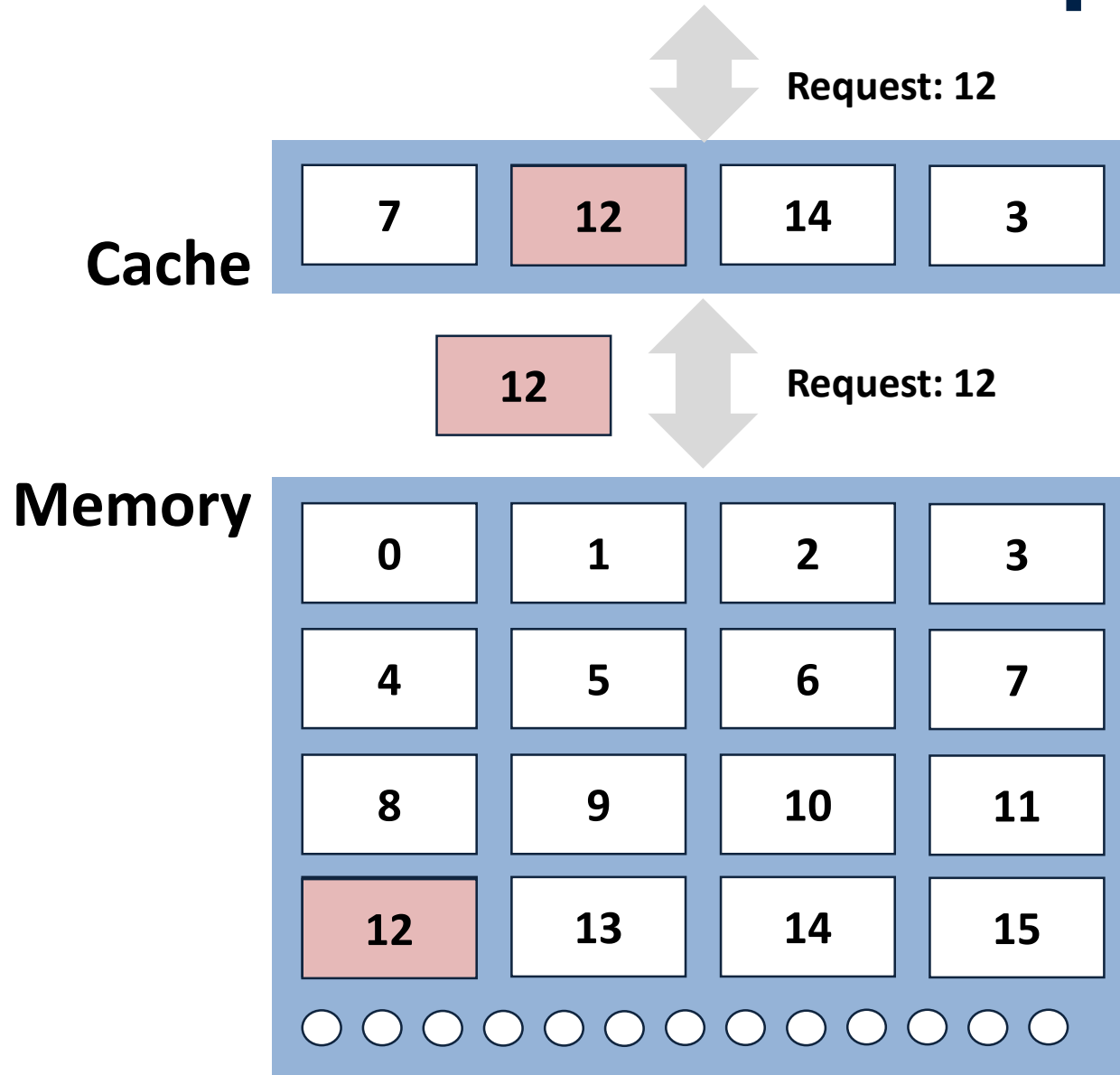
- Larger, slower, cheaper memory
- Viewed as partitioned into “blocks” or “lines”

General Cache Concepts: Hits



- Data in block b is needed
- If block b is in the cache:
it's a **hit**!

General Cache Concepts: **Miss**



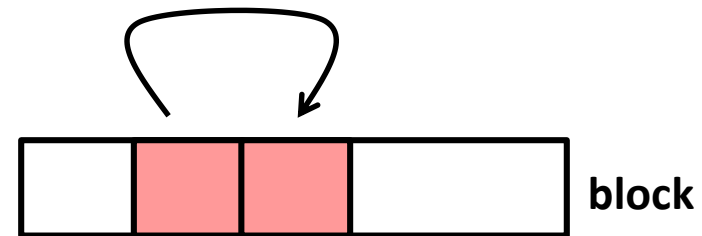
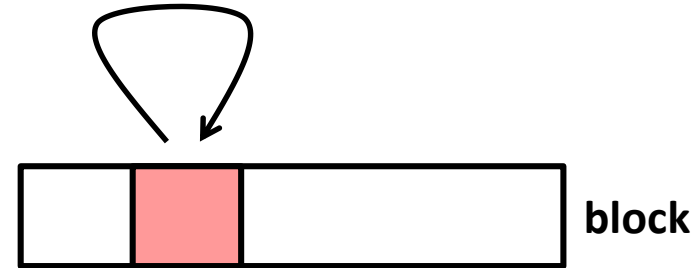
- Data in block b is needed
- Block b is not in cache: **Miss!**
- Block b is fetched from memory
- Block b is stored in cache:
 - **Placement policy:**
determines where b goes
 - **Replacement policy:**
determines which block gets evicted (victim)

Today: Caches and Memory Hierarchy

- Cache basics
- **Principle of locality**
- Memory hierarchies
- Cache organization
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches

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 references 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? Data? Instructions?

```
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

Locality Example #1

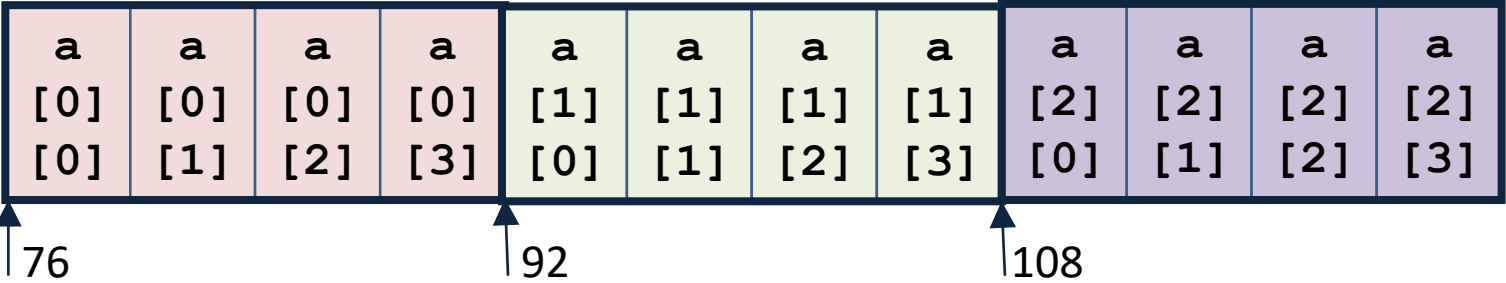
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

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]

Access Pattern: stride = ?	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

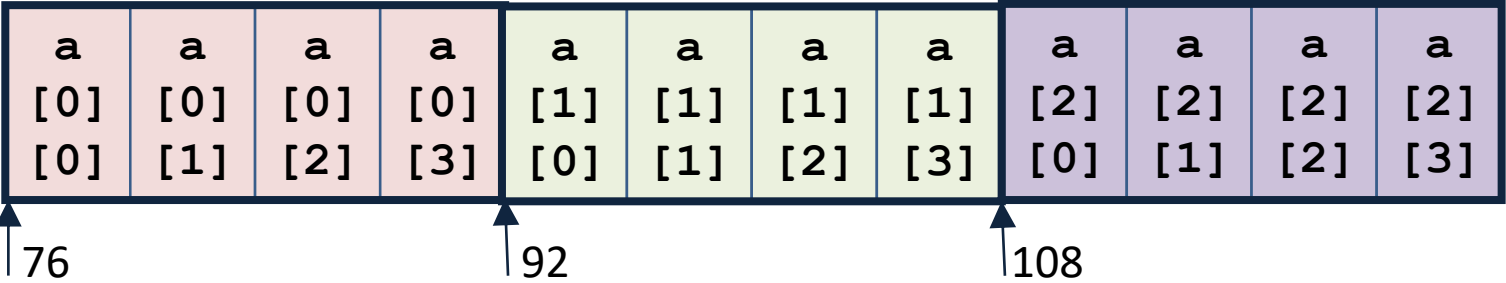


Note: 76 is just one possible starting address of array a

Locality Example #2

```
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;
}
```

Layout in Memory



Note: 76 is just one possible starting address of array a

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]

Access Pattern: stride = ?	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]

Cache Performance Metrics

- **Huge difference between cache hit and a cache miss**
 - Could be 100x speed difference between accessing cache and main memory (measured in *clock cycles*)
- **Miss Rate (MR)**
 - Fraction of memory references not found in cache
(misses / accesses) = $1 - \text{Hit Rate}$
- **Hit Time (HT)**
 - Time to deliver a block in the cache to the processor
 - Includes time to determine whether the block is in the cache
- **Miss Penalty (MP)**
 - Additional time required because of a miss

Cache Performance

- **Two things hurt the performance of a cache:**
 - Miss rate and miss penalty
- **Average Memory Access Time (AMAT):** average time to access memory considering both hits and misses

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- 99% hit rate is twice as good as 97% hit rate!
 - Assume HT of 1 clock cycle and MP of 100 clock cycles
 - 97%: $\text{AMAT} = 1 + (1 - 0.97) \times 100 = 1 + 3 = 4$ clock cycles
 - 99%: $\text{AMAT} = 1 + (1 - 0.99) \times 100 = 1 + 1 = 2$ clock cycles

Can we have more than one cache?

■ Why would we want to do that?

- Avoid going to memory!

■ Typical performance numbers:

- Miss Rate:
 - L1 MR = 3-10%
 - L2 MR = Quite small (e.g., <1%), depending on parameters, etc.
- Hit Time:
 - L1 HT = 4 clock cycles
 - L2 HT = 10 clock cycles
- Miss Penalty:
 - P = 50-200 cycles for missing in L2 and going to main memory
 - Trend: increasing!

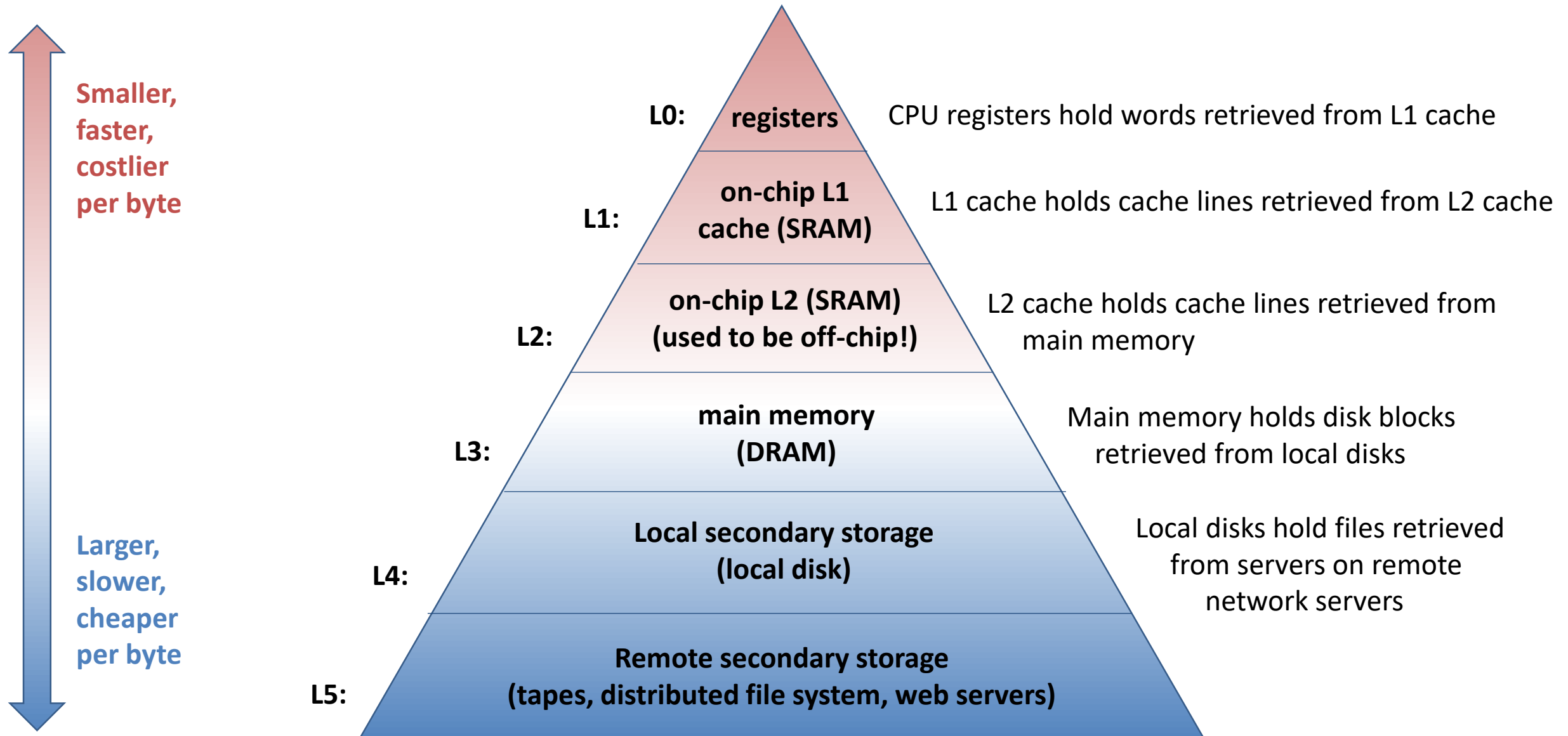
Today: Caches and Memory Hierarchy

- Cache basics
- Principle of locality
- **Memory hierarchies**
- Cache organization
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches

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 gap 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**
 - They suggest an approach for organizing memory and storage systems known as *memory hierarchy*

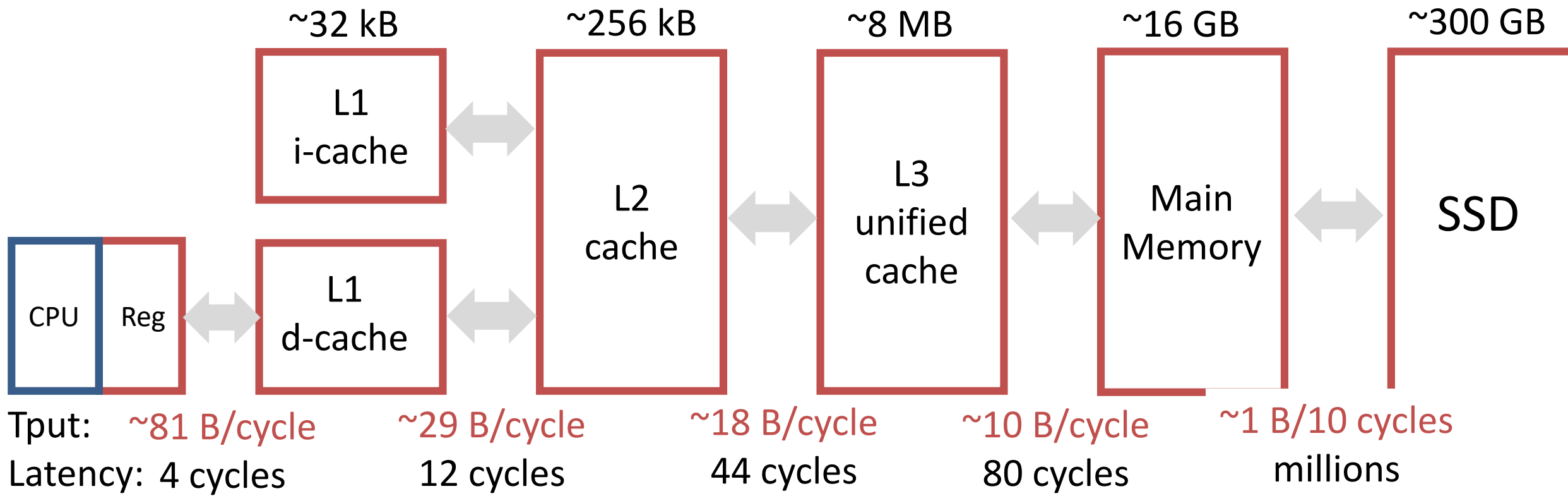
An Example Memory Hierarchy



Examples of caching in the hierarchy

Cache type	What is cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4/8-byte words	CPU core	0	Compiler
TLB	Address translation	On-chip TLB	0	Hardware
L1 cache	64-byte blocks	On-chip L1	1	Hardware
L2 cache	64-byte blocks	On-chip L2	10	Hardware
Virtual Memory	4kB page	Main memory (RAM)	100	Hardware + OS
Buffer cache	4kB sectors	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10'000'000	SMB/NSF client
Browser cache	Web pages	Local disk	10'000'000	Web browser
Web cache	Web pages	Remote server disks	1'000'000'000	Web proxy server

Memory hierarchy: Intel Core i7-6700K



Not drawn to scale

Summary

■ Memory hierarchy

- Successively higher levels contain “most used” data from lower levels
- Exploits *temporal* and *spatial* locality
- Caches are intermediate storage levels used to optimise data transfers between any system elements with different characteristics

■ Cache Performance

- Ideal case: found in cache (hit)
- Bad case: not found in cache (miss), search in next level
- Average Memory Access Time (AMAT) = $HT + MR \times MP$
 - Hurt by Miss rate and Miss Penalty

Today: Caches and Memory Hierarchy

- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches

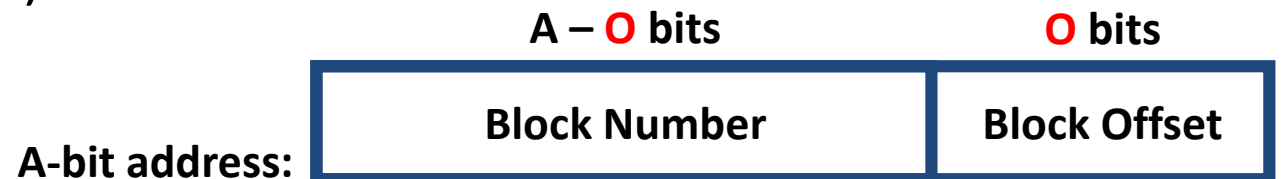
Cache Organisation (1)

■ Block Size (K)

- Unit of transfer between cache and memory
- Given in bytes and always a power of 2 (e.g., 64 B)
- Blocks consist of adjacent bytes, for spatial locality!

■ Offset field

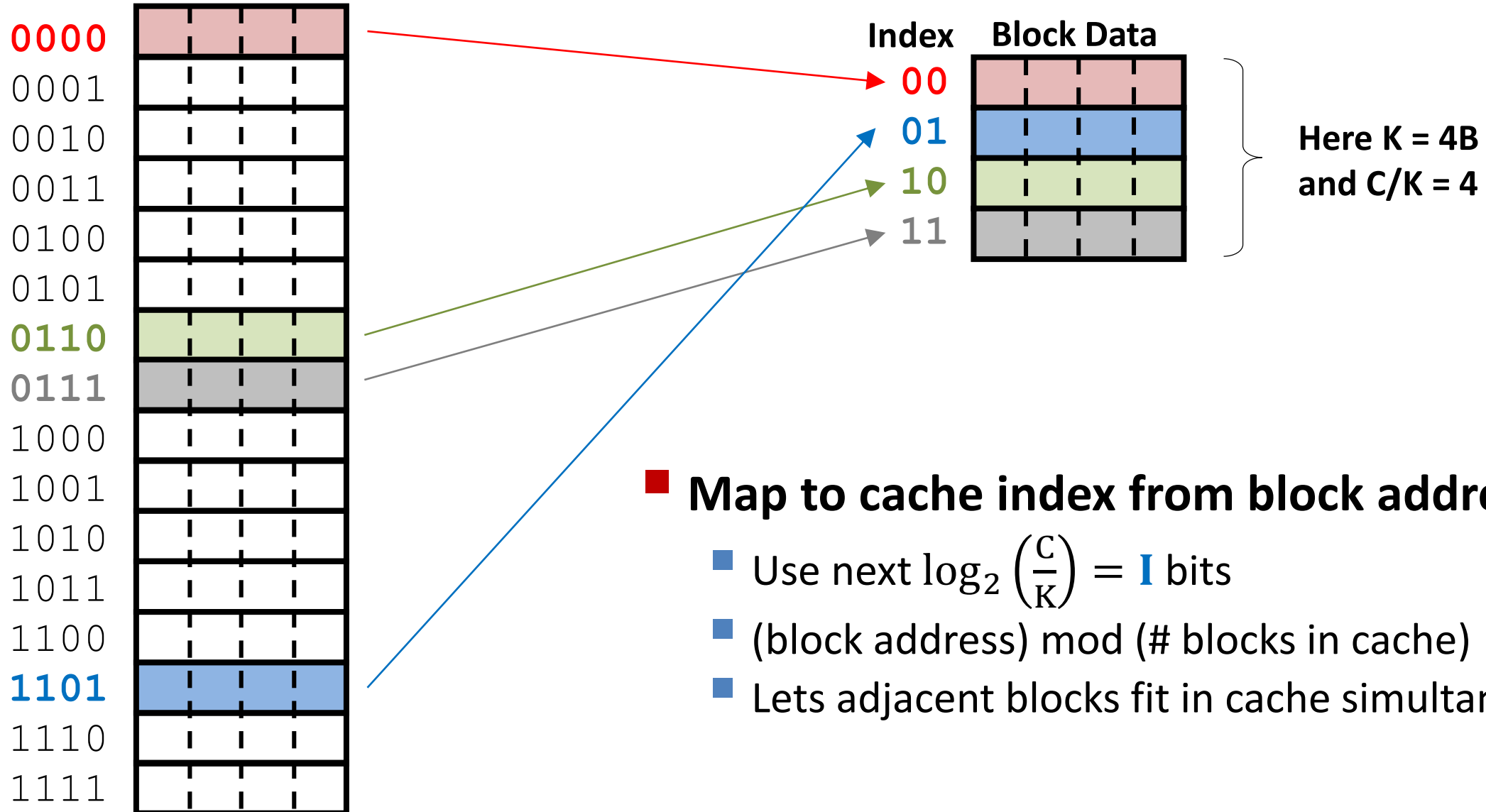
- Low-order $\log_2(\mathbf{K}) = \mathbf{O}$ bits of address tell you which byte within a block
 - (address) $\text{mod } 2^n = n$ lowest bits of address
- (address) modulo (# of bytes in a block)



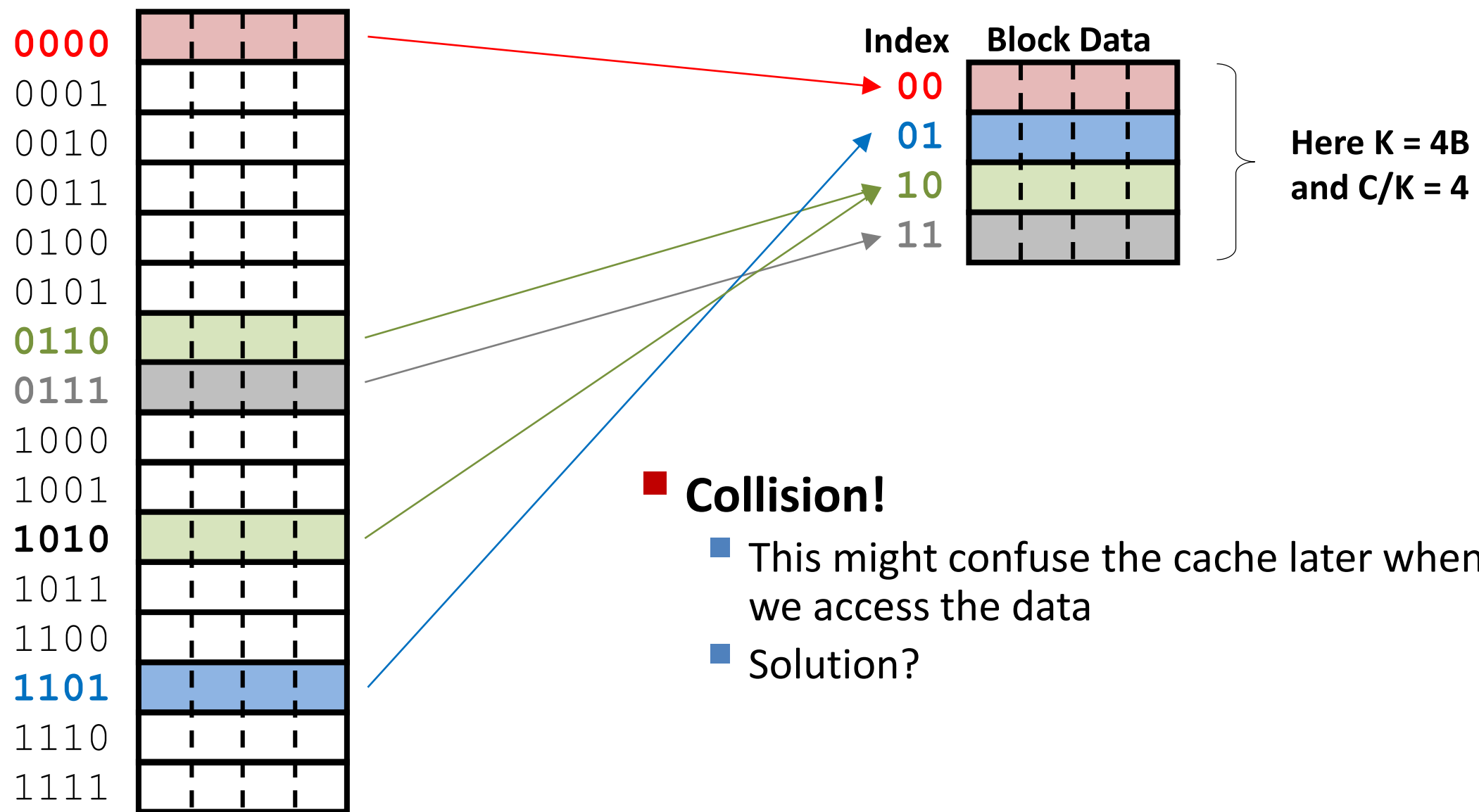
Cache Organisation (2)

- **Cache Size (C):** amount of *data* the cache can store
 - Cache can only hold so much data (subset of next level)
 - Given in bytes (C) or number of blocks (C/K)
 - Example: $C = 32 \text{ KiB} = 512 \text{ blocks}$ if using 64-B blocks
- **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**.
- **What is a data structure that provides fast lookup?**
 - Hash table!

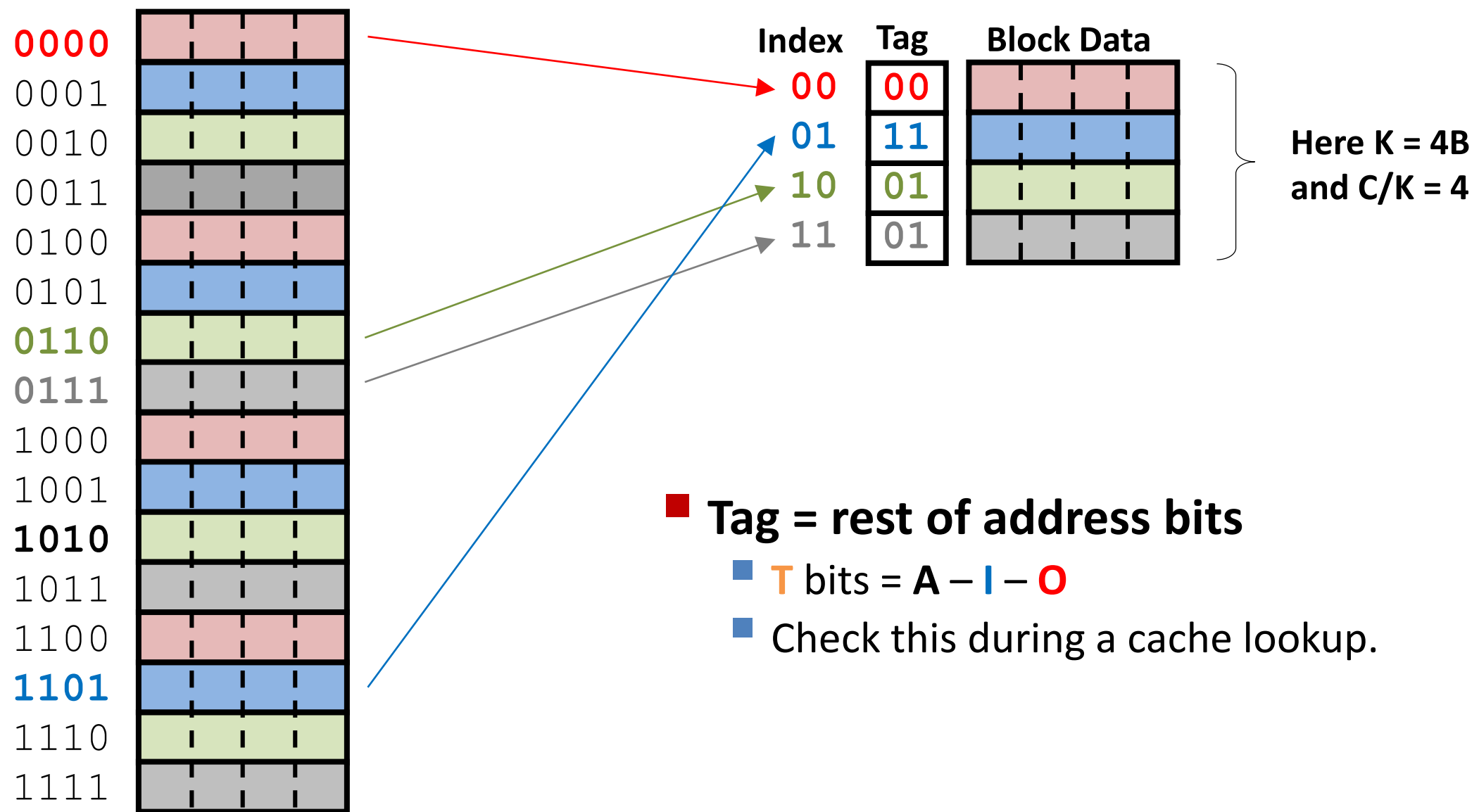
Place Data in Cache by Hashing Address



Place Data in Cache by Hashing Address



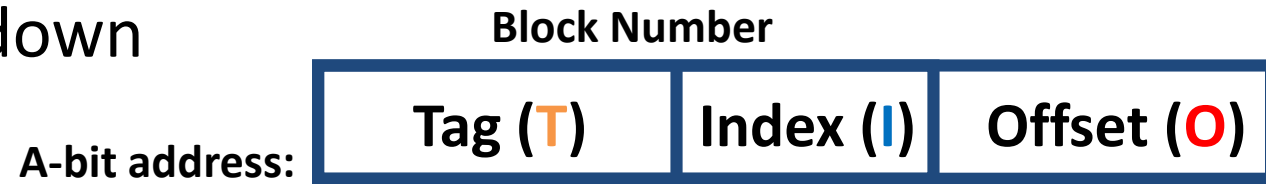
Place Data in Cache by Hashing Address



Checking for a Requested Address

- CPU sends address request for chunk of data
 - Address and requested data are not the same thing!
 - Analogy: your friend \neq his or her phone number

- TIO address breakdown



- **Index** field tells you where to look in cache
- **Tag** field lets you check that data is the block you want
- **Offset** field selects specified start byte within block
- Note: **T** and **I** sizes will change based on hash function

Today: Caches and Memory Hierarchy

- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches