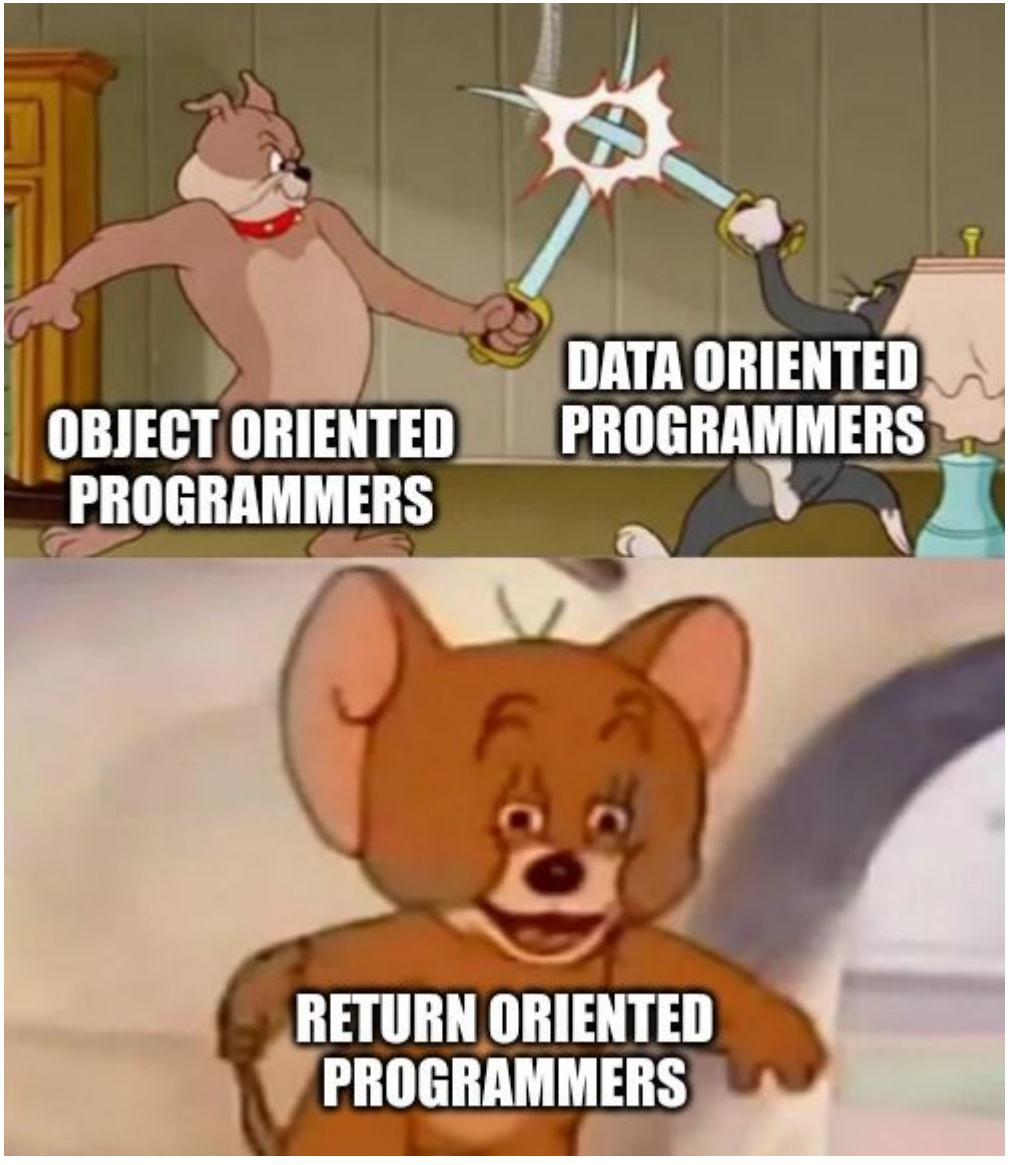

OBJECT ORIENTED PROGRAMMERS
DATA ORIENTED PROGRAMMERS
RETURN ORIENTED PROGRAMMERS

TODO

- Assignment 4 assigned due November 10
- Study for Exam 2 November 6

Overview

- Memory hierarchy

Structure of course

- First third (Exam 1):
  - Data Representation
  - Memory
- Second third (Exam 2):
  - Machine Code
  - Memory exploits
- Final third (Final exam):
  - Memory hierarchy ← here
  - Processes

Memory hierarchy

- Less *how* data is stored
- More *where* data is stored
- This can make a significant performance difference!

arraysum

Hardware of memory

- RAM: random access memory
- E.g., what main memory is
- “Random Access”?
  - Accessing any address takes the same amount of time
  - Compare: a hard drive…



Or in data structure terms

- Arrays: provide random access
  - O(1) access to any position
- Linked lists: do *not* provide random access
  - O(*n*) access to any position

Two types of RAM

- DRAM (dynamic RAM)
  - Typically what memory is implemented in
  - Dynamic because the contents need to be continually refreshed
  - Slow but cheap (comparatively)

Two types of RAM

- SRAM (static RAM)
  - Typically what CPU caches (special memory on the CPU) are implemented in
  - Static because the contents need to be continually refreshed
  - Fast but expensive (comparatively)

Honorable mention

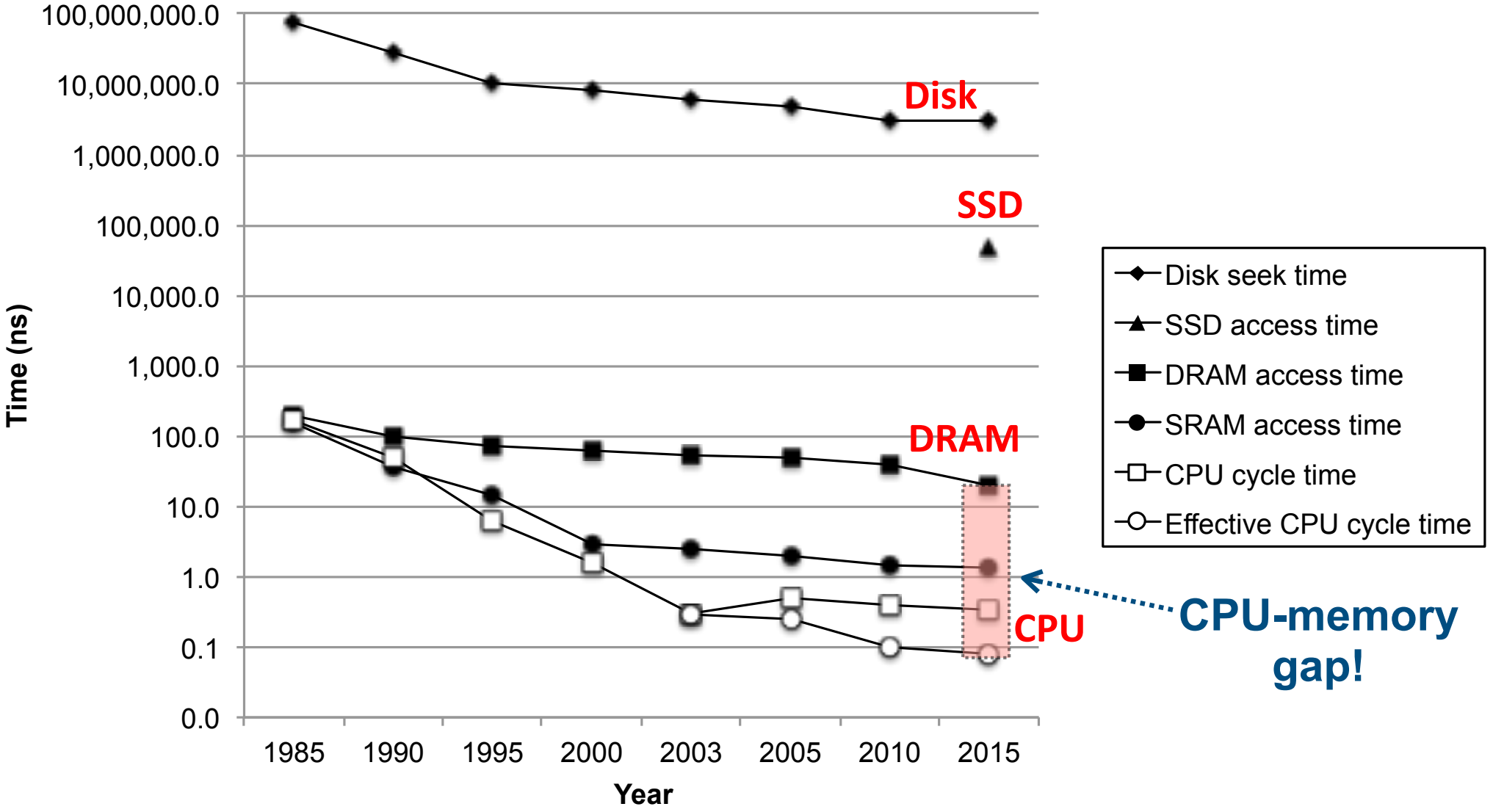
- Register file
  - CPU registers are implemented in an array of special memory
  - Same underlying technology as SRAM but not addressable so not technically RAM
  - More specialized and typically faster than SRAM
  - Addressable by *register name*

Storage

- Hard drives
  - Spinning disks + needles
- SSDs
  - NAND flash memory
  - More random access than hard drives but not completely random access

Memory hardware hierarchy

- Register file
- SRAM
- DRAM
- SSD
- Hard drives


100,000,000.0
10,000,000.0
1,000,000.0
100,000.0
10,000.0
1,000.0
100.0
10.0
1.0
0.1
0.0
Time (ns)
1985
1990
1995
2000
2003
2005
2010
2015
Year
Disk
SSD
DRAM
CPU
Disk seek time
SSD access time
DRAM access time
SRAM access time
CPU cycle time
Effective CPU cycle time
CPU-memory gap!

CPU-memory gap

- The issue we will be concerned with in this unit is the *CPU-memory gap*
  - The difference between processor speed and main memory access time
- Historically what has happened with this gap?

What can bridge the gap?

- A *cache*
- What is a cache?

What can bridge the gap?

- A *cache*
- What is a cache?
  - A temporary, higher speed storage
  - E.g., you could cache a file read from the hard disk in main memory
- We will focus on *CPU caches* in this class

CPU caches

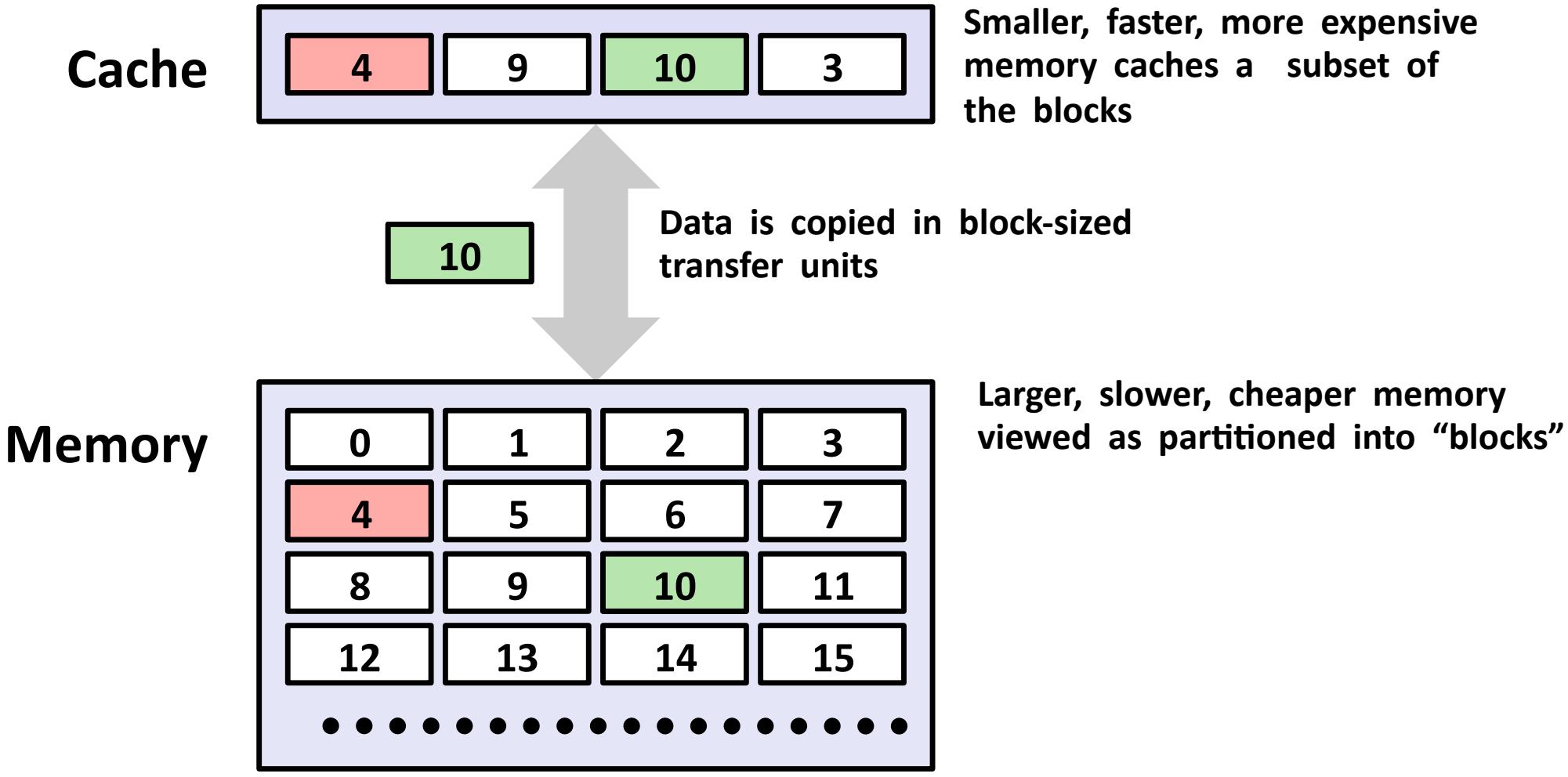
- Cache contents of RAM for faster access by the CPU
- Implemented as SRAM

Blocks

- Caches divide main memory into *blocks*
- Blocks are the atomic unit of transfer between memory and the CPU cache
- Size of a block can vary from cache to cache
- Always a power of 2
- These days almost always 64 bytes

Blocks

- The idea is that if you want to cache address 0 or address 7, you get addresses 0 through 63.
- Or if you want to cache address 0x1000 or 0x1007, you get addresses 0x1000 through 0x1000 + 63.
- 1024 bytes of main memory divides into 16 64-byte blocks
- Suppose we load block 10…

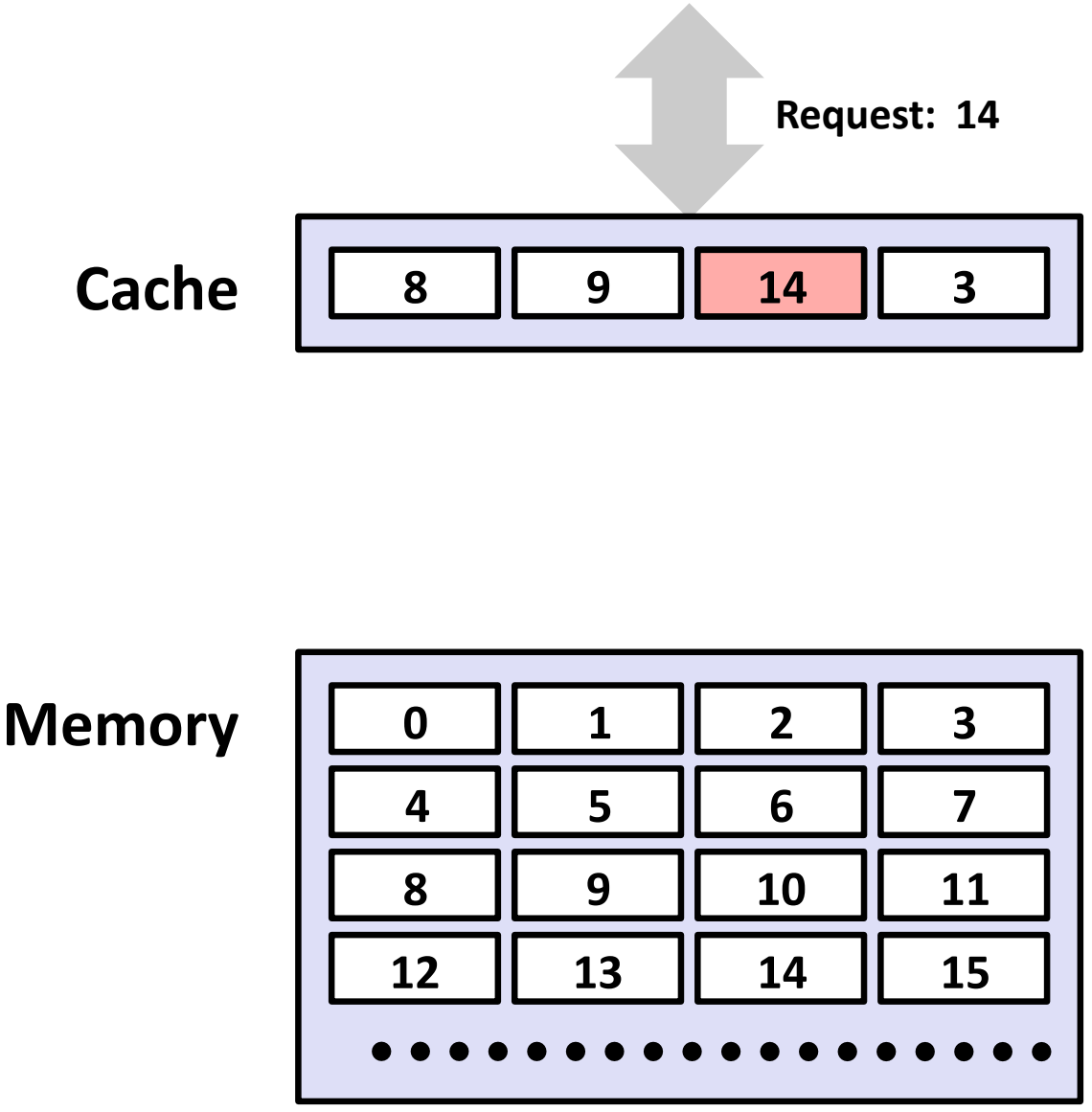

Cache
4
9
10
3
Smaller, faster, more expensive memory caches a subset of the blocks
10
Data is copied in block-sized transfer units
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Larger, slower, cheaper memory viewed as partitioned into “blocks”

Loads from memory

- When an instruction loads from memory
  - E.g., a memory operand as a source register
- First attempt read from cache (fast)
- Before reading from RAM (slow)

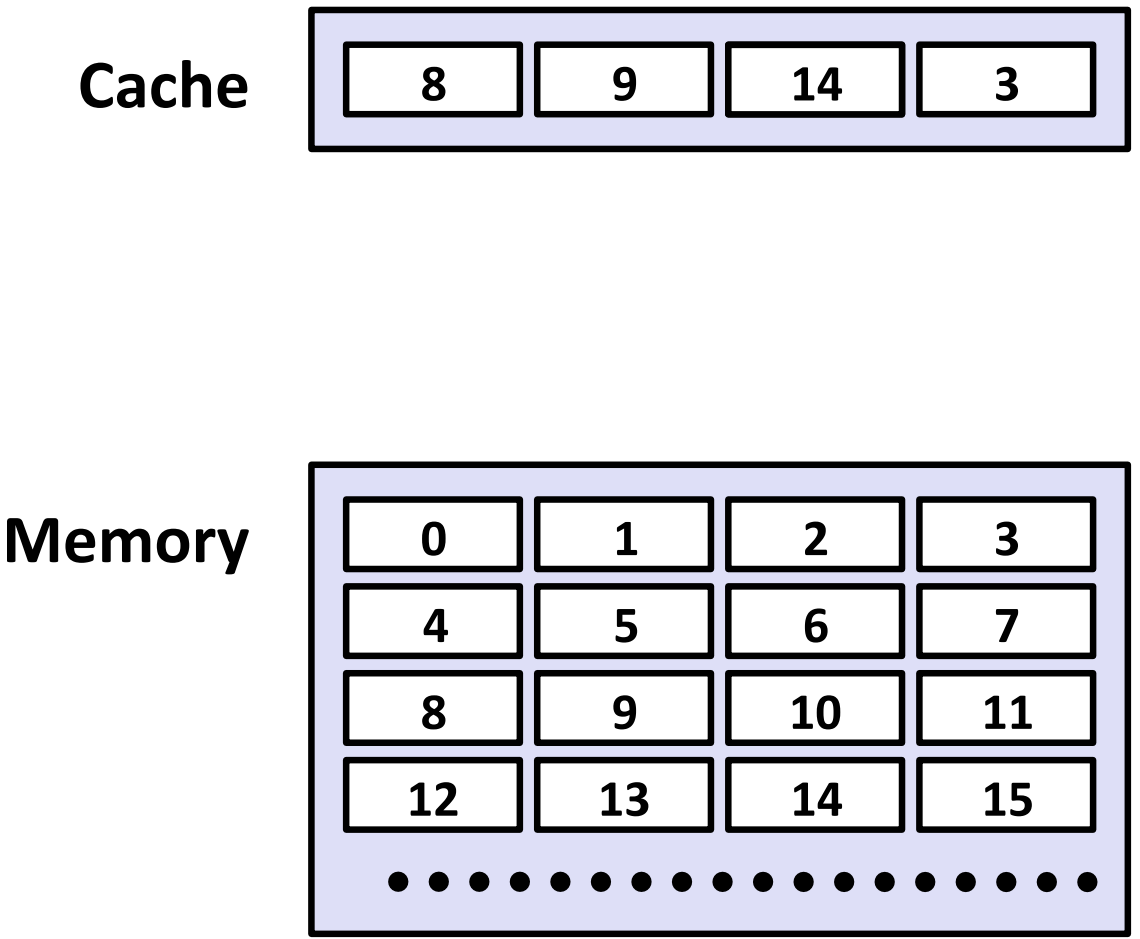
Cache hit

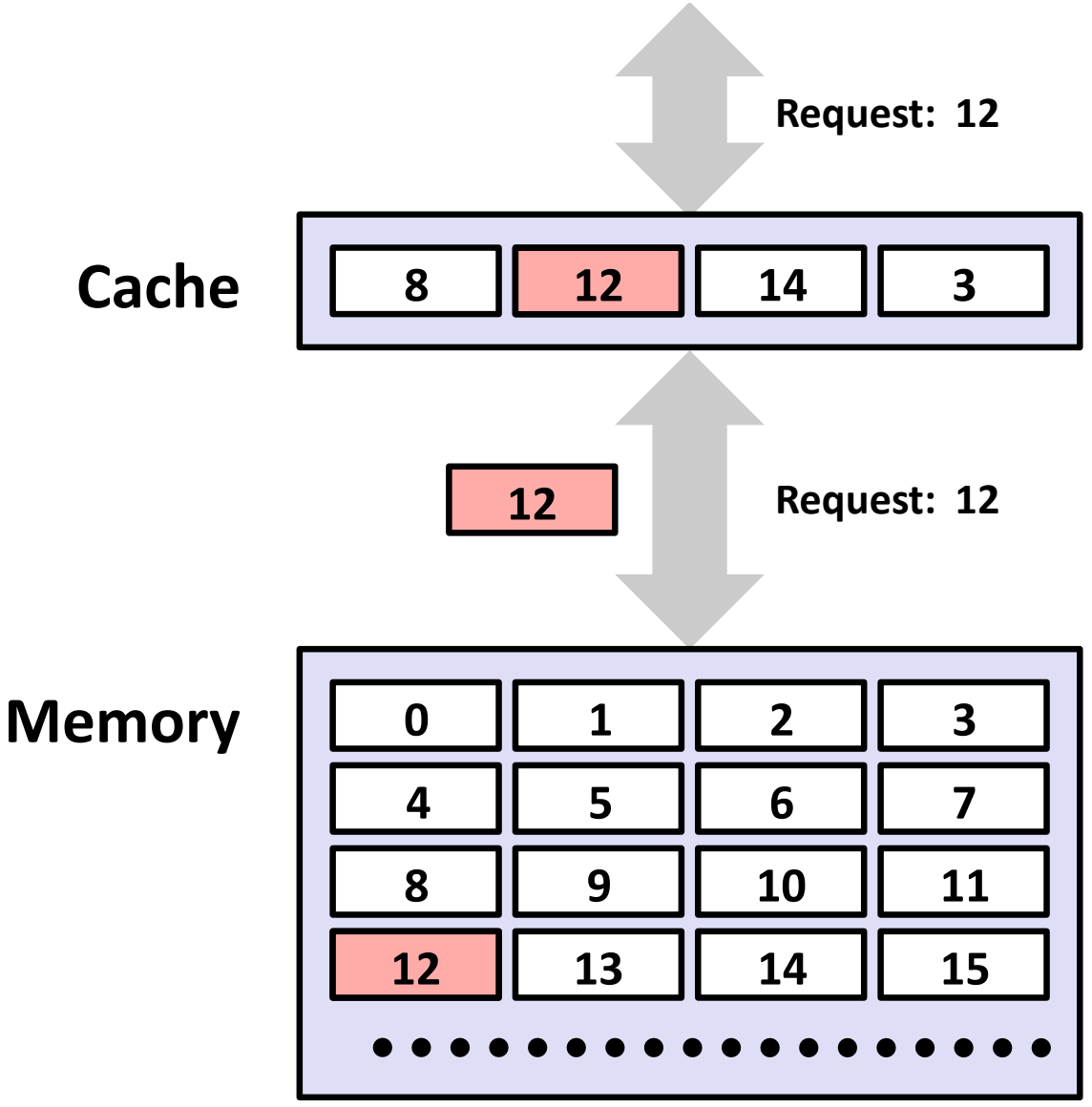
- If the loaded address’s block is in cache 🎉
- This is good!
- Loaded data served from cache without going to main memory
- We want as many hits as possible
- In this example, a read from block 14 is a hit…


Request: 14
Cache
8
9
14
3
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cache miss

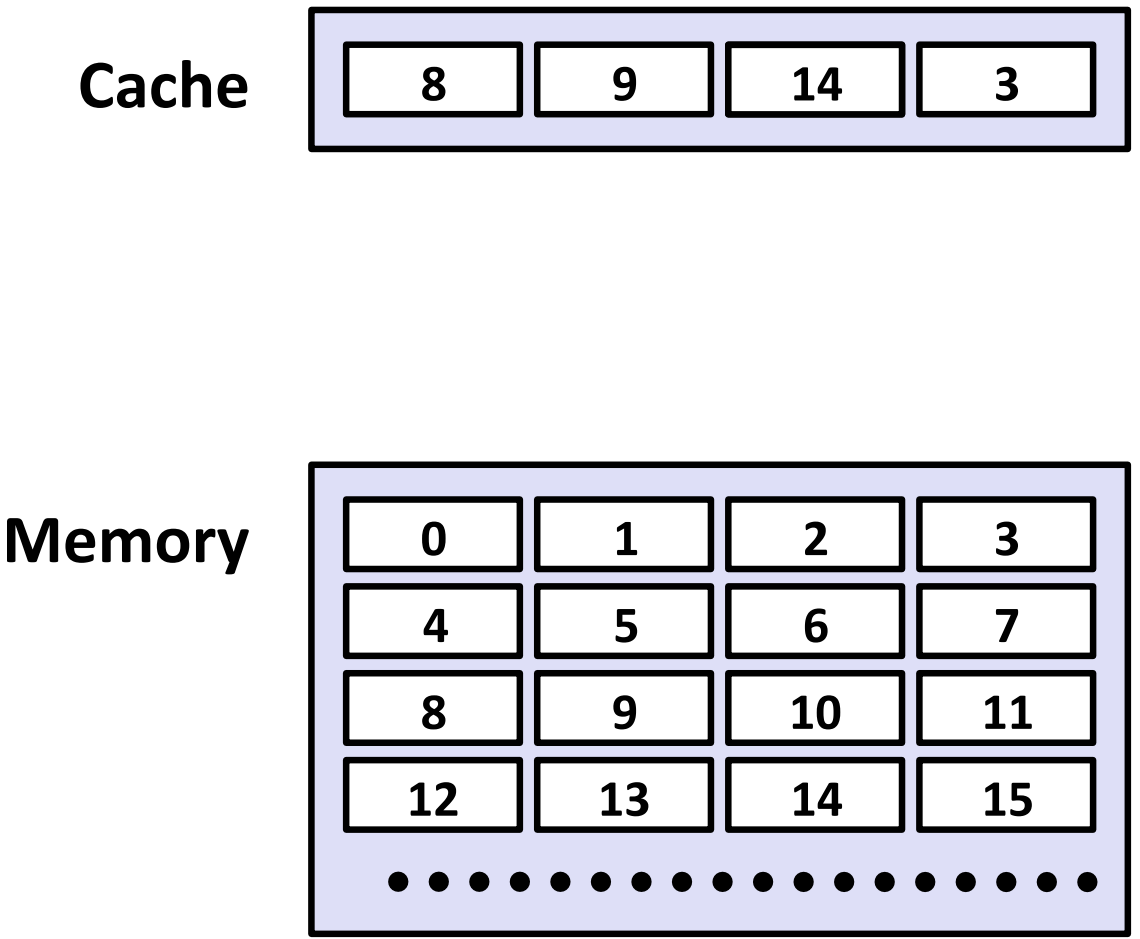
- If the loaded address's block is **not** in cache 😭
- This is bad!
- Has to load from main memory
- We want as few misses as possible
- In this example, a read from block 12 is a miss…

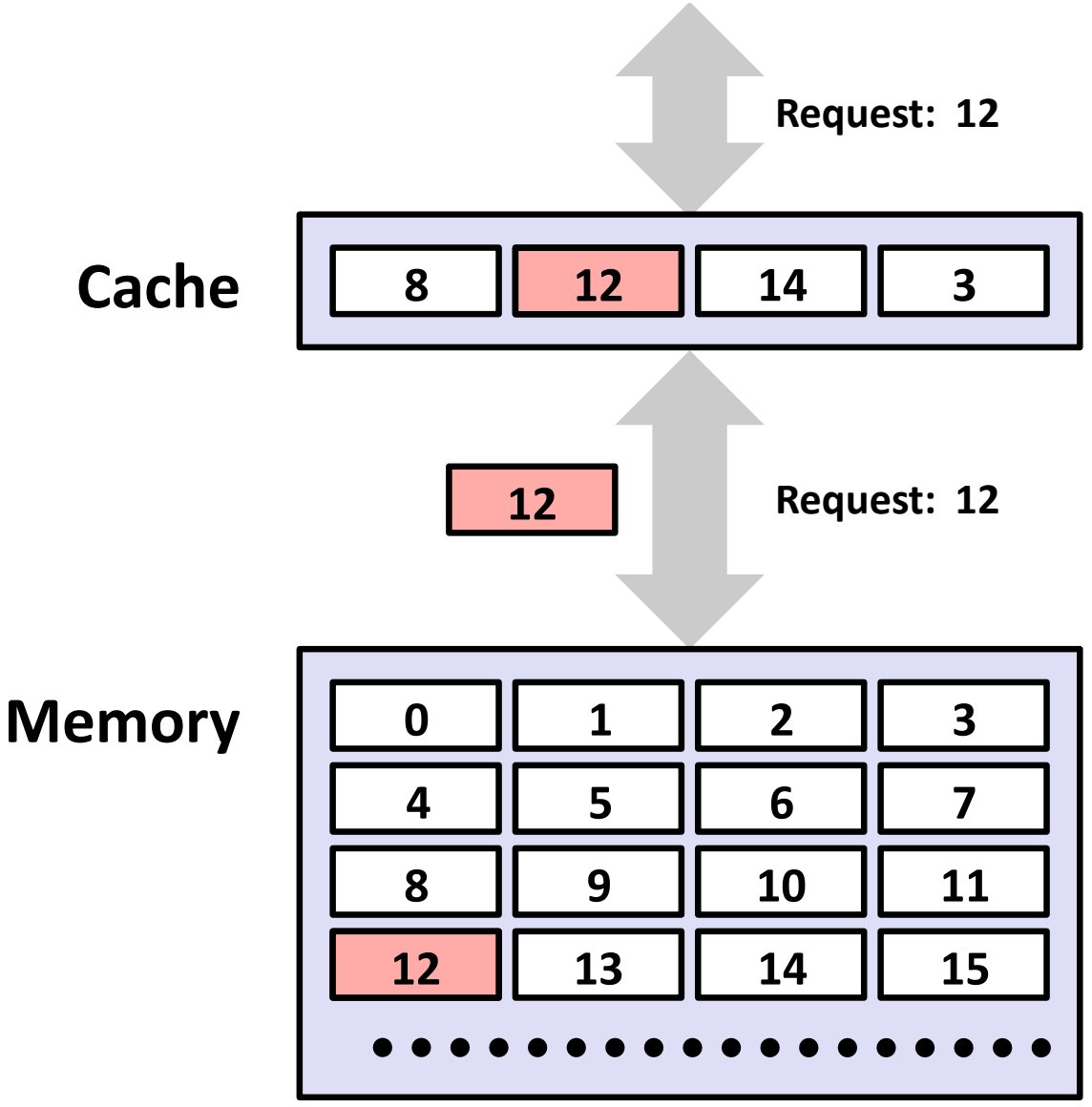

Cache
8
9
14
3
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


Request: 12
Cache
8
12
14
3
12
Request: 12
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cache fill

- After a miss, we load the block into the cache for future use…


Cache
8
9
14
3
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


Request: 12
Cache
8
12
14
3
12
Request: 12
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cache behavior

- *Transparent* to programs
- Note: no instructions to load from cache, store to cache, etc.
- In Lab 5 we will perform timing measurements to infer properties of our own CPU cache
- But generally caches are transparent programs

Cache lines

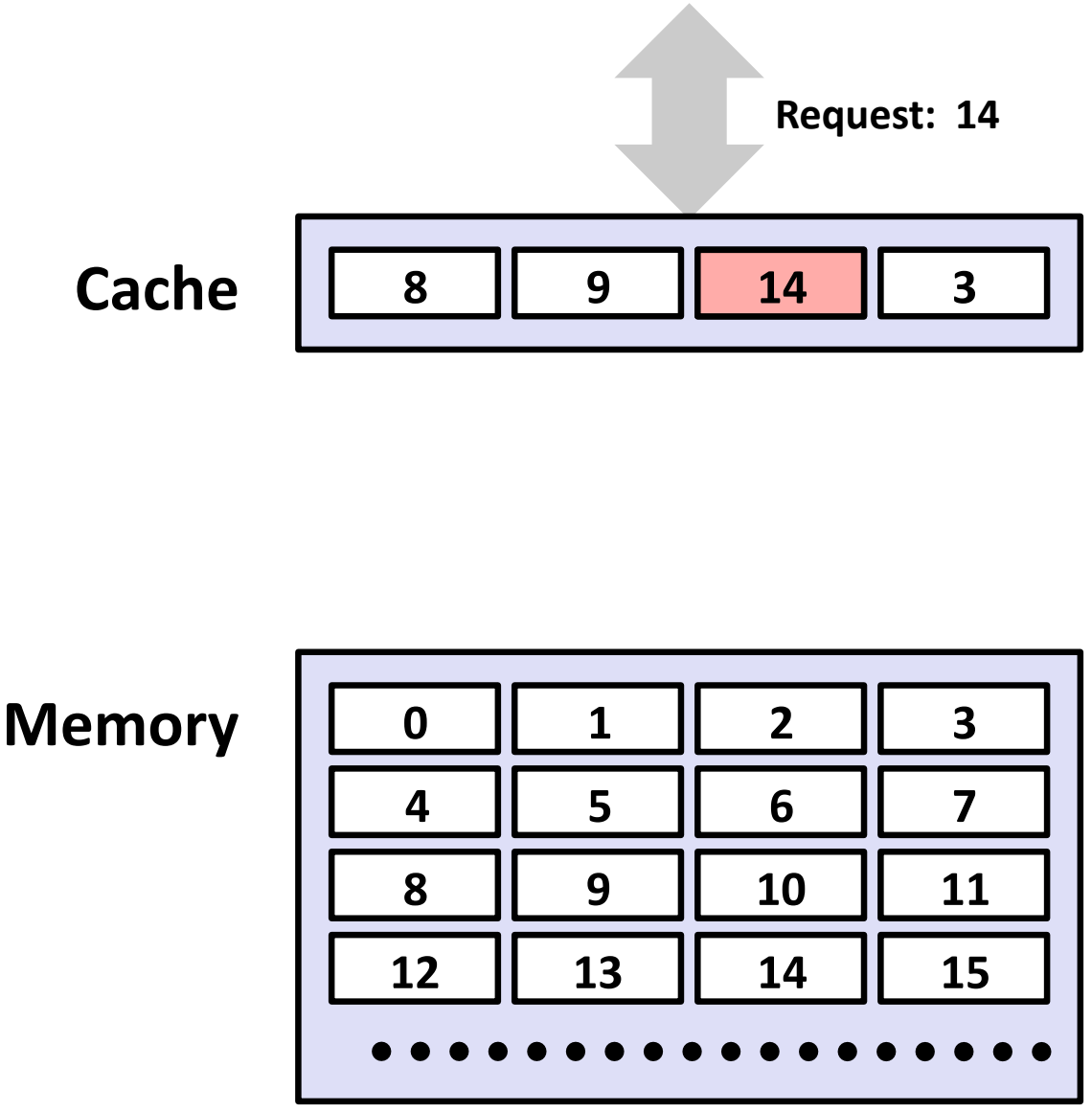
- Caches consist of *cache lines*
- Each cache line stores one data block from memory
- # of blocks = size of main memory / block size
- # of cache lines = size of cache / block size

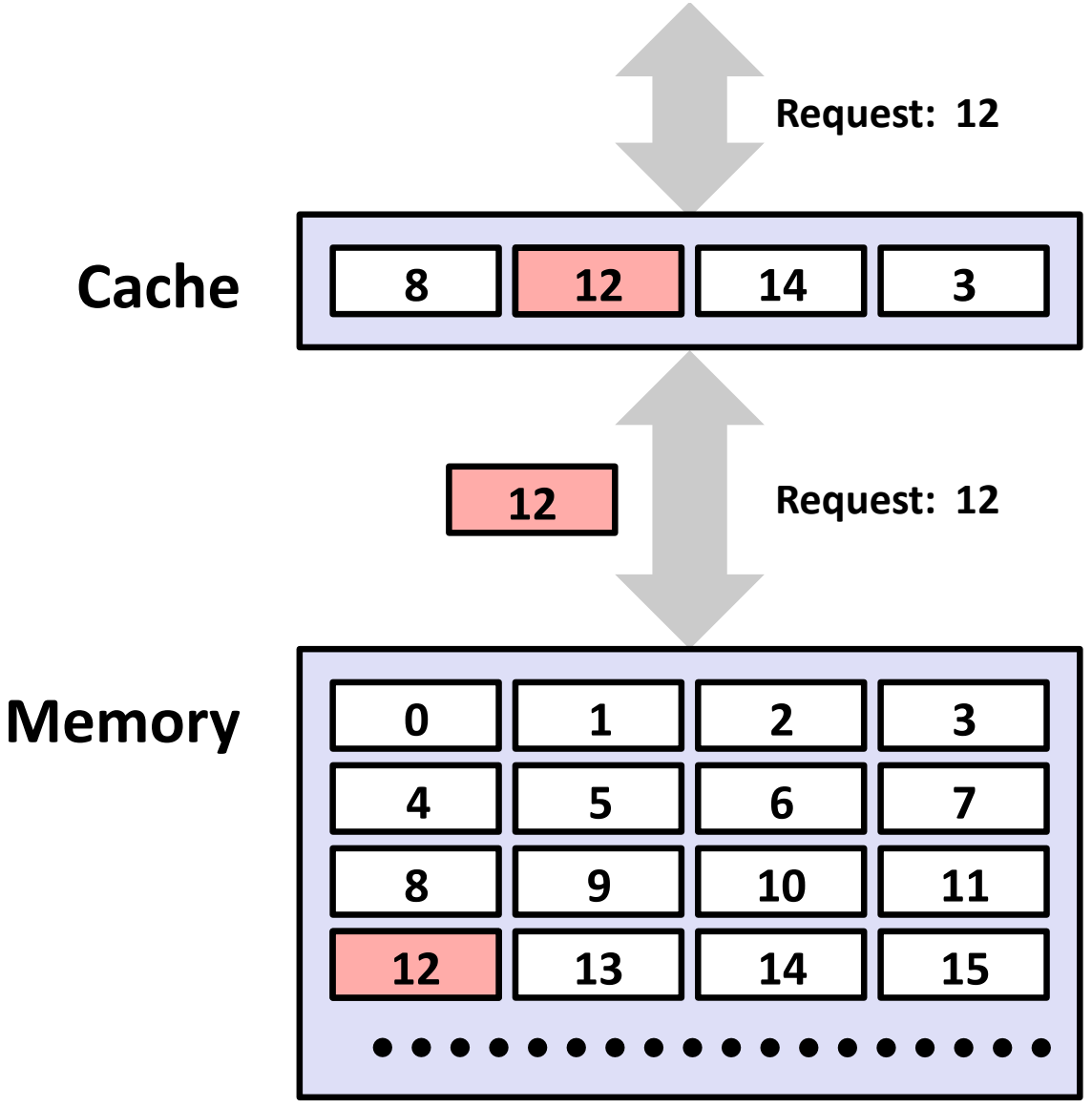
1024 cache lines
× 64-byte blocks
= 64 KB cache size

Line	Data (64 Bytes)
0	
1	
2	
3	
4	
…	…
1020	
1021	
1022	
1023	

Cache design goal

- Maximize hit rate
- Minimize miss rate
- Keep cache simple to keep it fast
  - Caches are implemented in hardware, and there can be physical constraints!


Request: 14
Cache
8
9
14
3
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

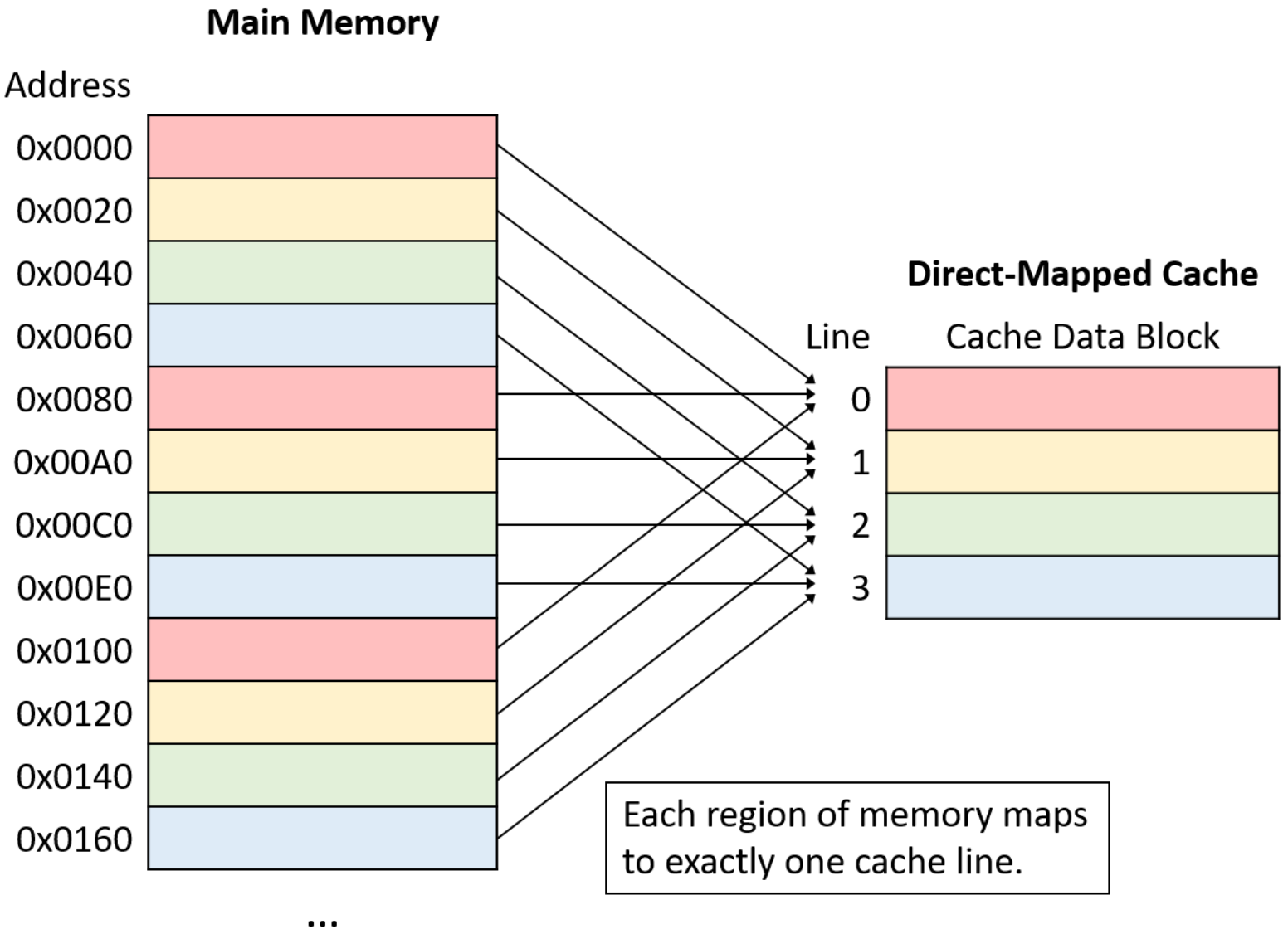

Request: 12
Cache
8
12
14
3
12
Request: 12
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cache block placement

- What happened to 9?
- It was *evicted* from the cache
- Why did we put 12 *there*?
  - We want to place blocks so as to minimize cache misses

Direct mapping

- Direct mapping is simplest block placement
- Each block maps to exactly one cache line


Main Memory
Address
0x0000
0x0020
0x0040
0x0060
0x0080
0x00A0
0x00C0
0x00E0
0x0100
0x0120
0x0140
0x0160
...
Direct-Mapped Cache
Line
Cache Data Block
0
1
2
3
Each region of memory maps to exactly one cache line.

Selecting cache line

- For memory block numbered n,
  select cache line $\ell = n \% L$, where L is the total number of cache lines

Direct mapping pros and cons

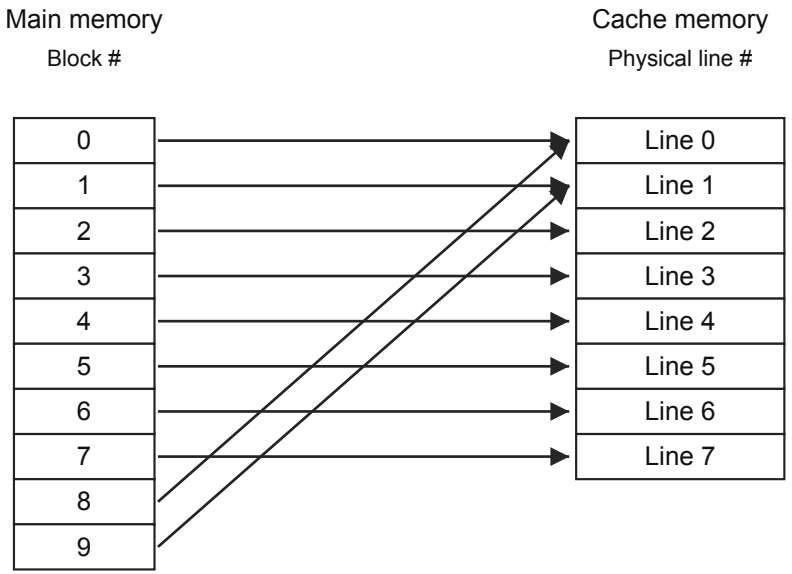
- Pro
  - Simple and fast
- Con
  - Not a lot of flexibility
  - Imagine a program that repeatedly accessed 0x0 and 0x100…

Reduce placement conflicts

- Allow a block to be in more than one cache line
- Set-associative cache – each block maps to a set of cache lines
- 2-way associative – a block maps to a set of 2 cache lines
- *n*-way associative – a block maps to a set of *n* cache lines…

Direct-mapped cache

S = 8 sets • E = 1 line/set • C = 8B

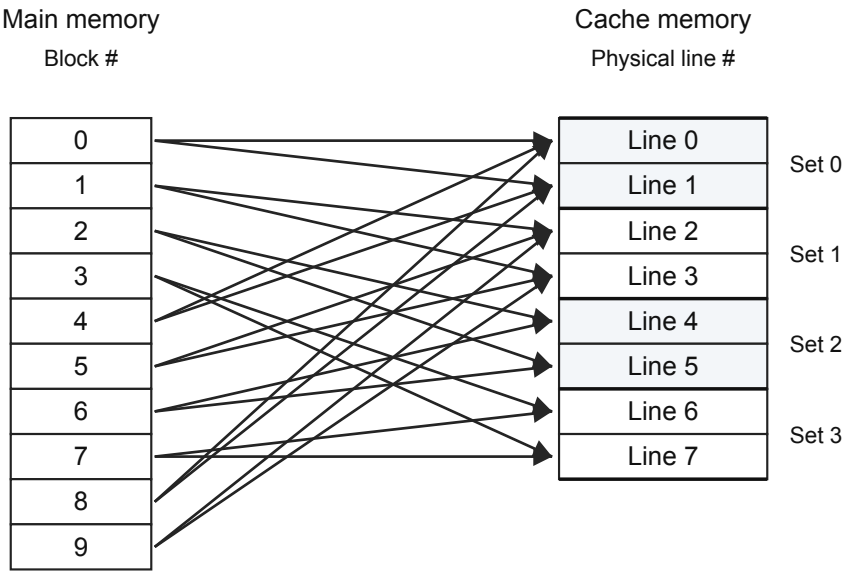


Each block maps to exactly one physical line.

Line number $\ell = n \bmod 8$.

2-way set-associative cache

S = 4 sets • E = 2 lines/set • C = 8B



Each block can occupy either of two lines in its set.

For set $s = n \bmod 4$, use lines $2s$ and $2s + 1$.

S/E/B cache model

- A cache can be described by three values
  - *S*: number of cache sets
  - *E*: number of lines per set a.k.a. the associativity
  - *B*: number of bytes per block
- The cache size $C = S * E * B$
- For a memory block numbered *n*, the model selects set $n \% S$

S/E/B cache model

- A cache can be described by three values
  - *S*: number of cache sets
  - *E*: number of lines per set a.k.a. the associativity
  - *B*: number of bytes per block
- For memory block numbered *n*,
  select set $s = n \% S$
  - The first cache line of *s* is $\ell = s * E$
  - The last cache line of *s* is $\ell' = s * (E + 1) - 1$

Cache miss → Which to evict?

- Direct-mapped: only one choice!
- *n*-way set associative: *n* choices!
- Which should we evict?

Cache miss → Which to evict?

- Direct-mapped: only one choice!
- *n*-way set associative: *n* choices!
- Which should we evict?
  - LRU cache – evict least recently used
  - Modern caches use proprietary algorithms but have LRU-like eviction

Why do caches work?

- They work because of *locality*
- Assumptions that are generally true about most programs
  - Not *all* programs: programs that read random bytes across a very large array are unlikely to see benefit

Temporal locality

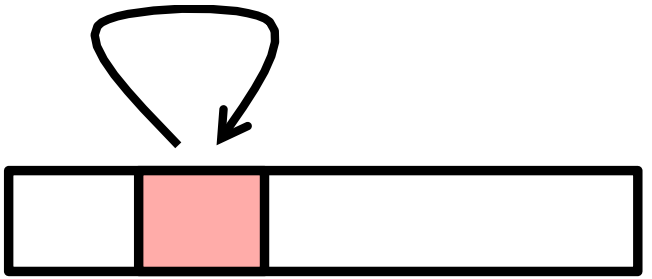
- If you read an address from memory, you will read it again soon
- True, for example, when repeatedly reading a local variable from the stack in a loop
- When you load an address, caches cache this load so that it will be faster in the future

Spatial locality

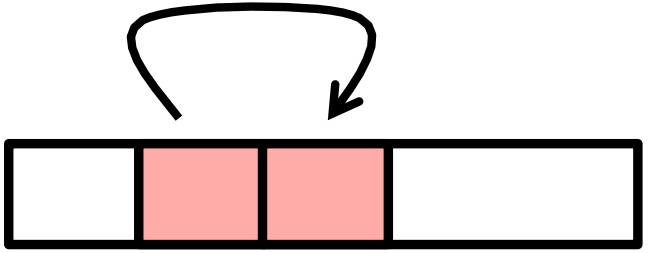
- If you read an address from memory, you will read different, nearby addresses soon
- True, for example, when iterating over an array
- When you load an address, caches cache the address’s entire block so that nearby loads will be faster in the future

Locality illustrated

- **Temporal locality:**



- **Spatial locality:**



These assumptions may not hold

- Sometimes you read from an address and never read it again
- Sometimes you read from an address and never read from nearby addresses
- In such cases caches aren’t helping
- Programs that read random bytes across a very large array are unlikely to see benefit

Important cracks in abstractions

- RAM (main memory): accessing any address is equally fast (hence, *random* access)
- RAM + cache: accessing the same or nearby addresses is *much* faster

Identifying locality

- Assume that all variables are stored on the stack, not in registers

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

Identifying locality

- Remember: caches don’t affect reads / writes to registers, just loads / stores to memory
- Registers are faster than caches, so why would we want to cache registers!

Optimize code for caches

- As programmers, we rarely know the exact details of our CPU cache
- Make our code have as much temporal and spatial locality as possible

(v1)

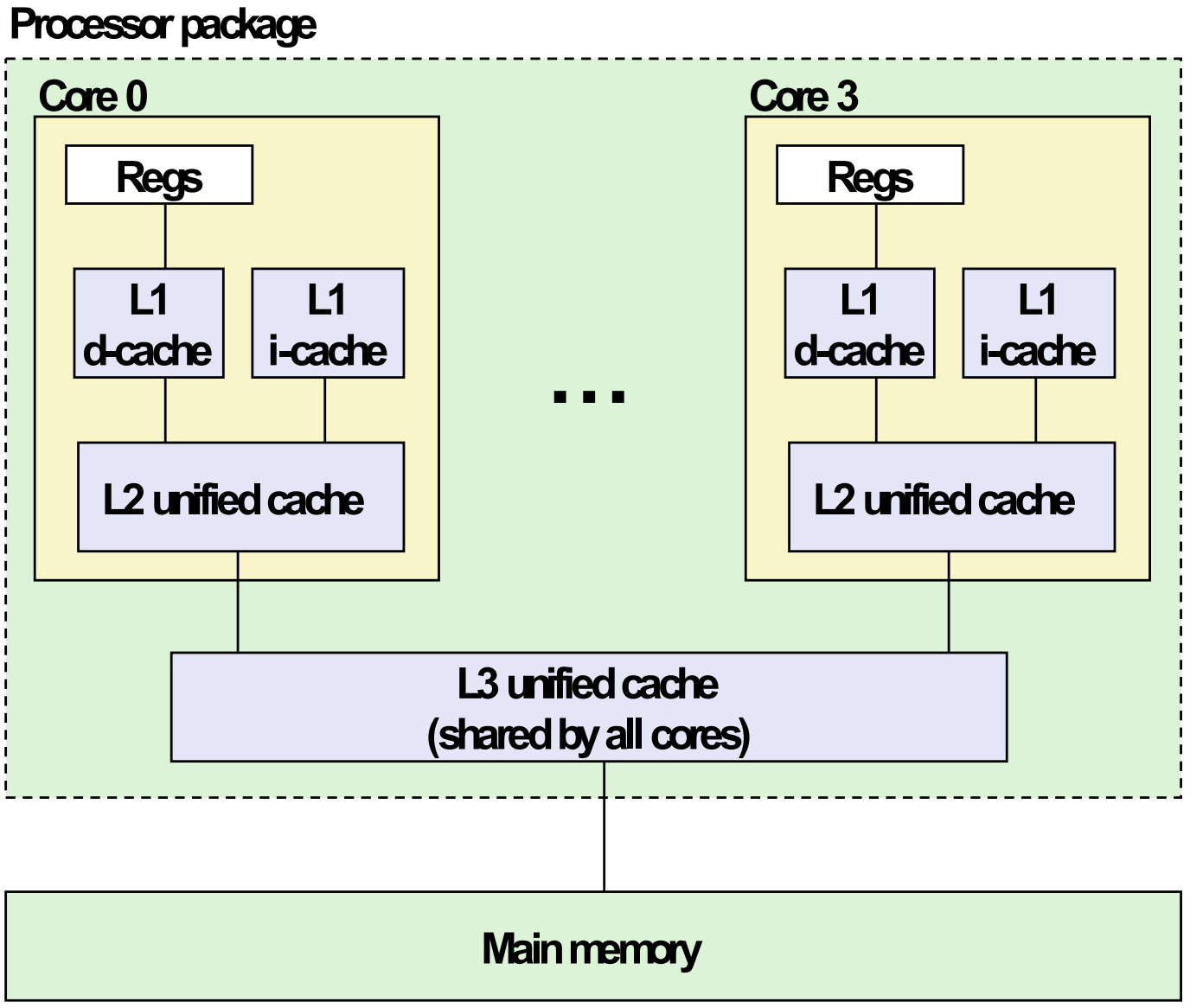
```
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;

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

(v2)

```
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;
}
```

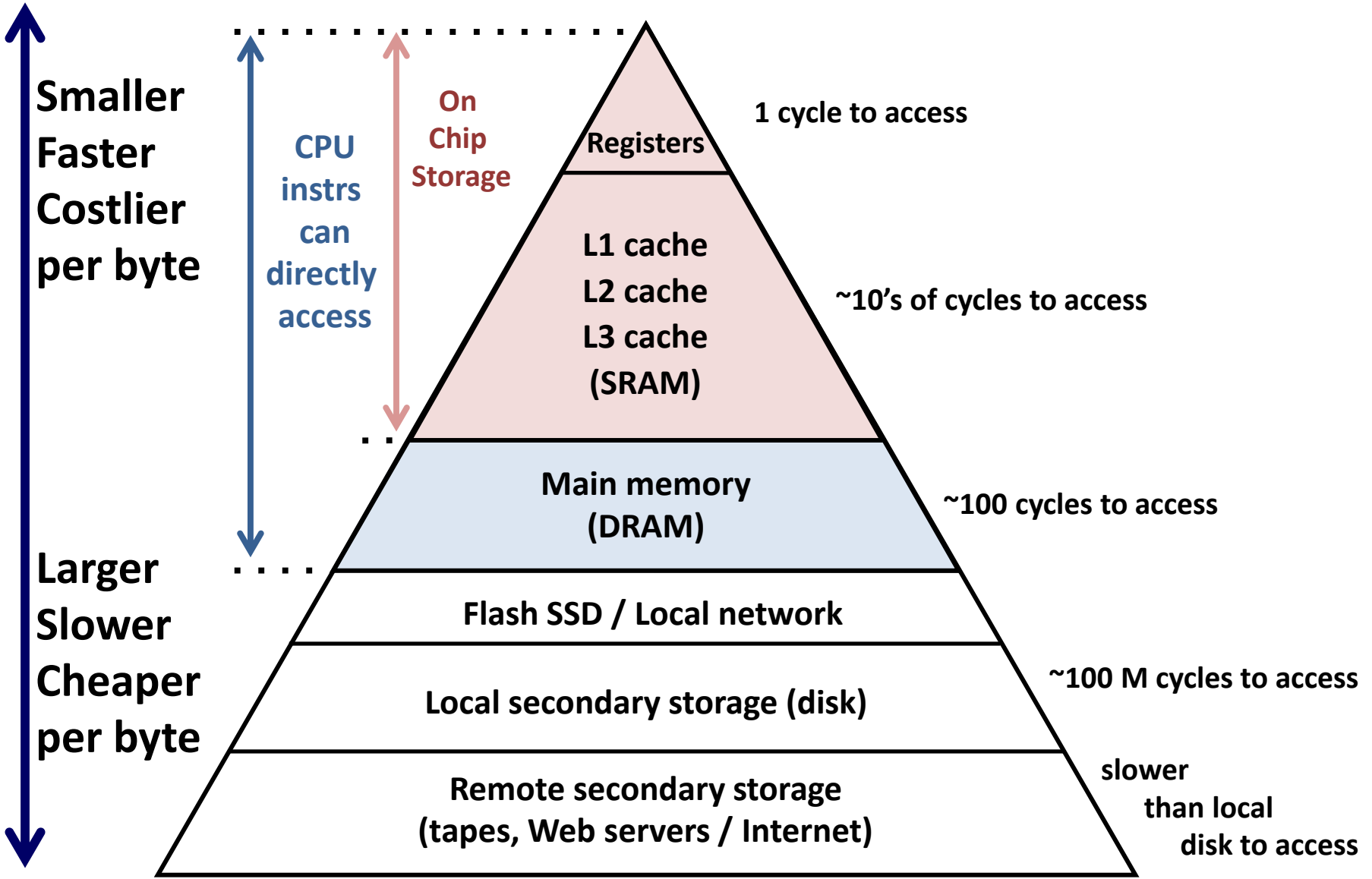


L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

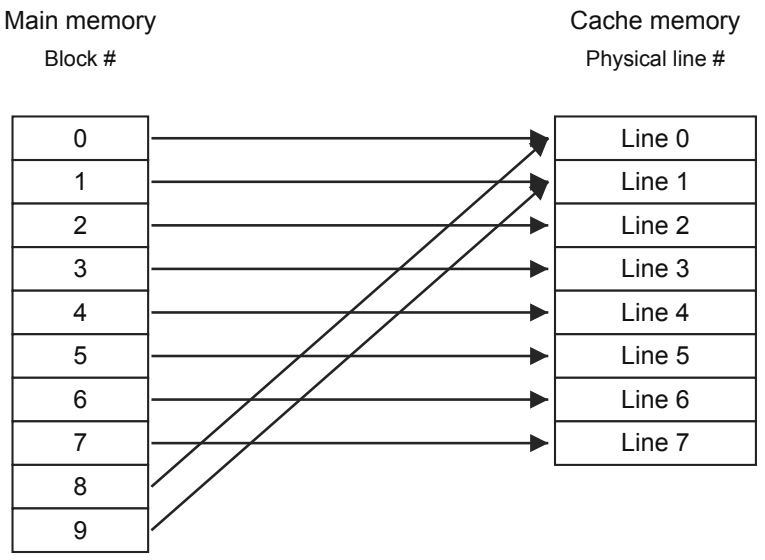
Block size: 64 bytes for all caches.


Smaller
Faster
Costlier
per byte
Larger
Slower
Cheaper
per byte
CPU instrs can directly access
On Chip Storage
Registers
1 cycle to access
L1 cache
L2 cache
L3 cache
(SRAM)
~10's of cycles to access
Main memory
(DRAM)
~100 cycles to access
Flash SSD / Local network
Local secondary storage (disk)
~100 M cycles to access
Remote secondary storage
(tapes, Web servers / Internet)
slower than local disk to access

Cache exercises

Direct-mapped cache

S = 8 sets • E = 1 line/set • C = 8B

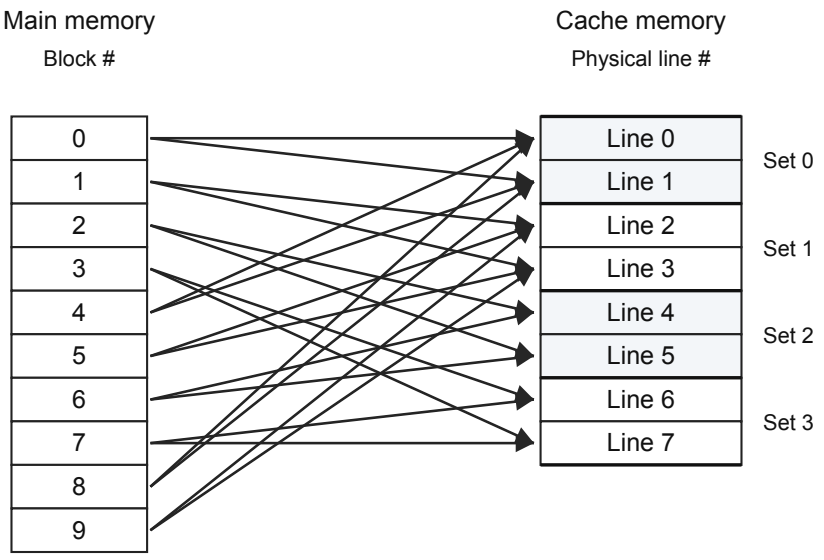


Each block maps to exactly one physical line.

Line number $\ell = n \bmod 8$.

2-way set-associative cache

S = 4 sets • E = 2 lines/set • C = 8B



Each block can occupy either of two lines in its set.

For set $s = n \bmod 4$, use lines $2s$ and $2s + 1$.

TODO

- Assignment 4 assigned due November 10
- Study for Exam 2 November 6