

# *Memory hierarchy*

---

Dr. Warisa Sritriratanarak  
Chulalongkorn University

# Today:



Why the memory hierarchy matters



Cache architecture



Cache performance

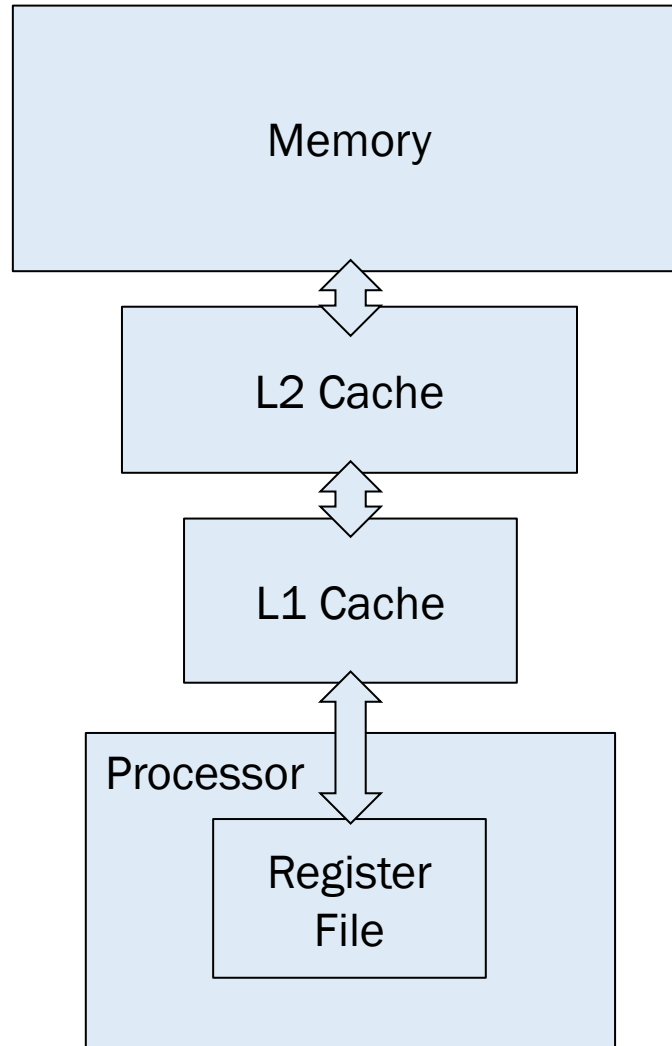
# Memory is slow

| Component    | Speed Unit       | Example Speed (Year)                              | Impact of Memory Wall  |
|--------------|------------------|---|--|
| Processor    | GHz              | 3.5 GHz (2023)                                    | Faster processing leads to increased reliance on faster memory access            |
| RAM          | MHz (Advertised) | DDR4-3200 MHz (2023)                              | Slower speed compared to processor can create bottlenecks                        |
| RAM          | MT/s (Actual)    | 6400 MT/s (DDR4-3200)                             | While MT/s is higher, the gap with processor speed persists                      |
| Cache Memory | MHz              | L3 Cache: 3200 MHz (Example - High-End CPU, 2023) | Smaller size limits data storage, but faster access mitigates memory wall impact |

It's pointless to get a faster processor if it spends its time waiting for memory

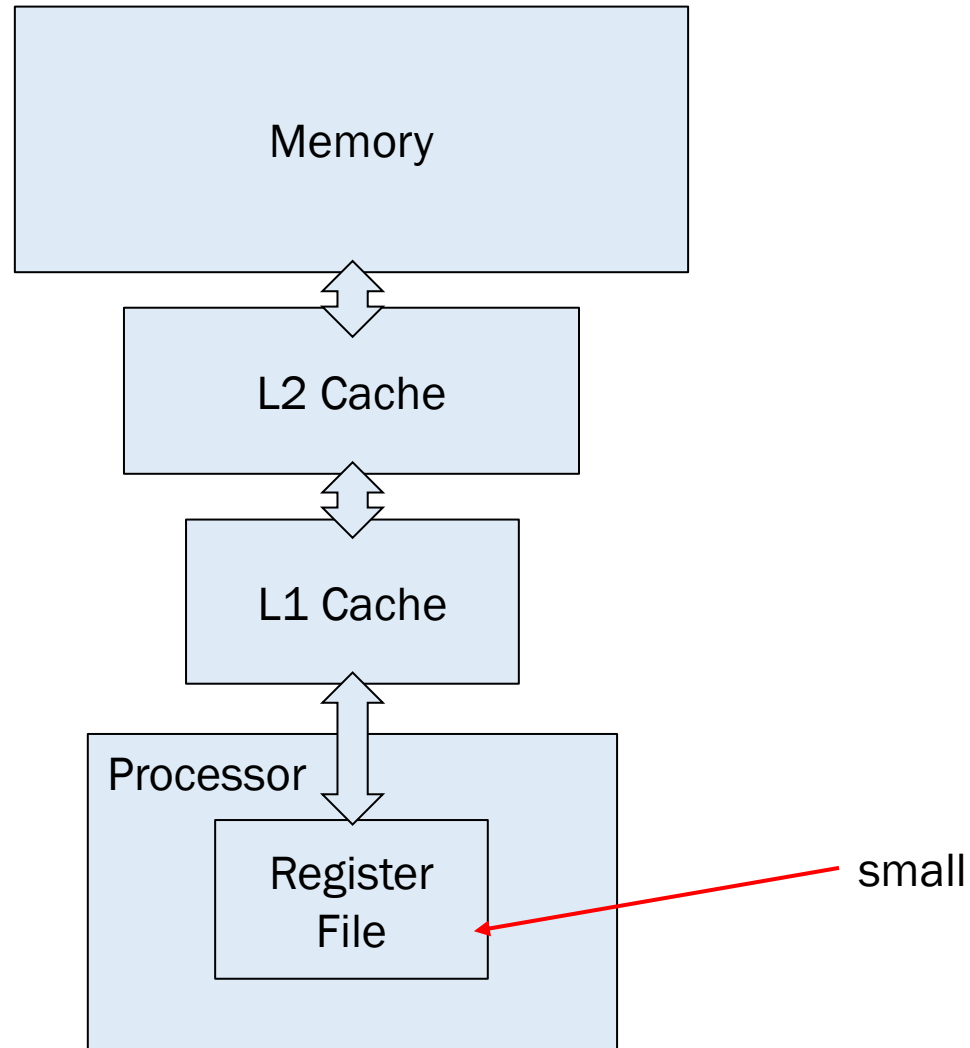
# Solution: memory hierarchy

- Smaller is faster



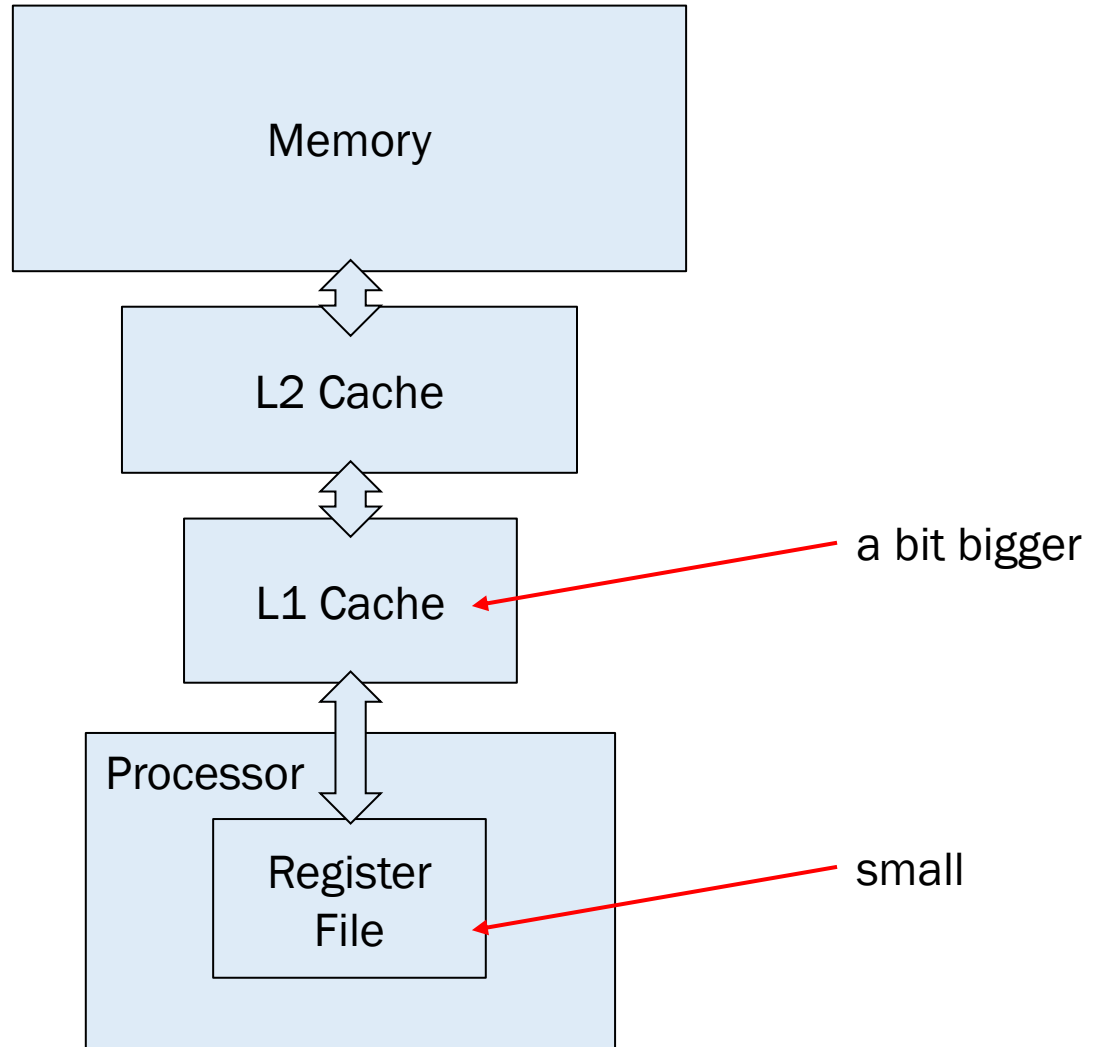
# Solution: memory hierarchy

- Smaller is faster



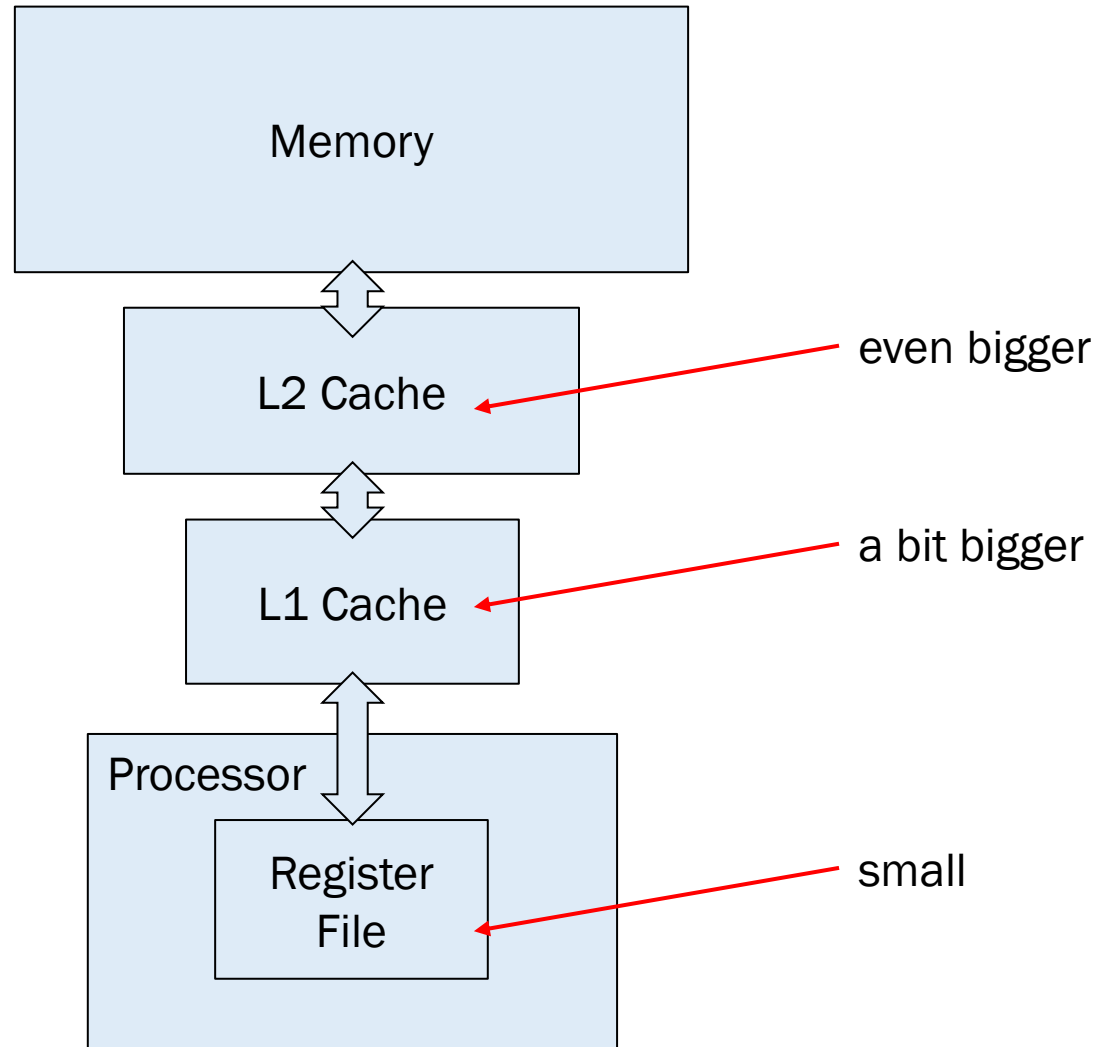
# Solution: memory hierarchy

- Smaller is faster



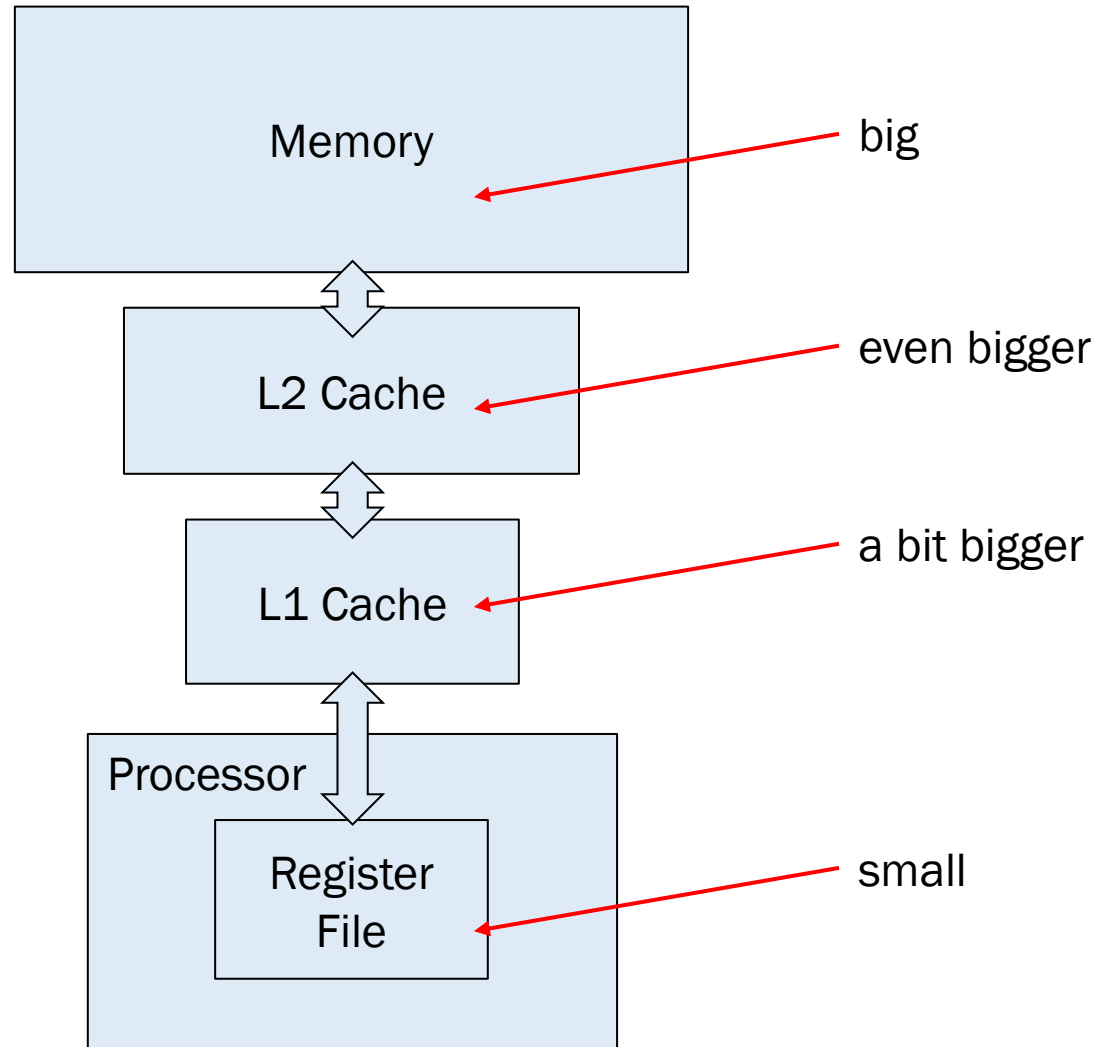
# Solution: memory hierarchy

- Smaller is faster



# Solution: memory hierarchy

- Smaller is faster





# Solution: memory hierarchy

- Why does this improve performance?
  - Isn't the slowest memory the bottleneck?

# Solution: memory hierarchy

- Why does this improve performance?
  - Isn't the slowest memory the bottleneck?
  
- Software has two interesting properties
  - **Temporal locality**
  - **Spatial locality**

# Solution: memory hierarchy

- **Temporal locality**
  - If you used the datum once, you'll probably use it again
- **Spatial locality**
  - If you used the datum once, you'll probably use the data close to it soon

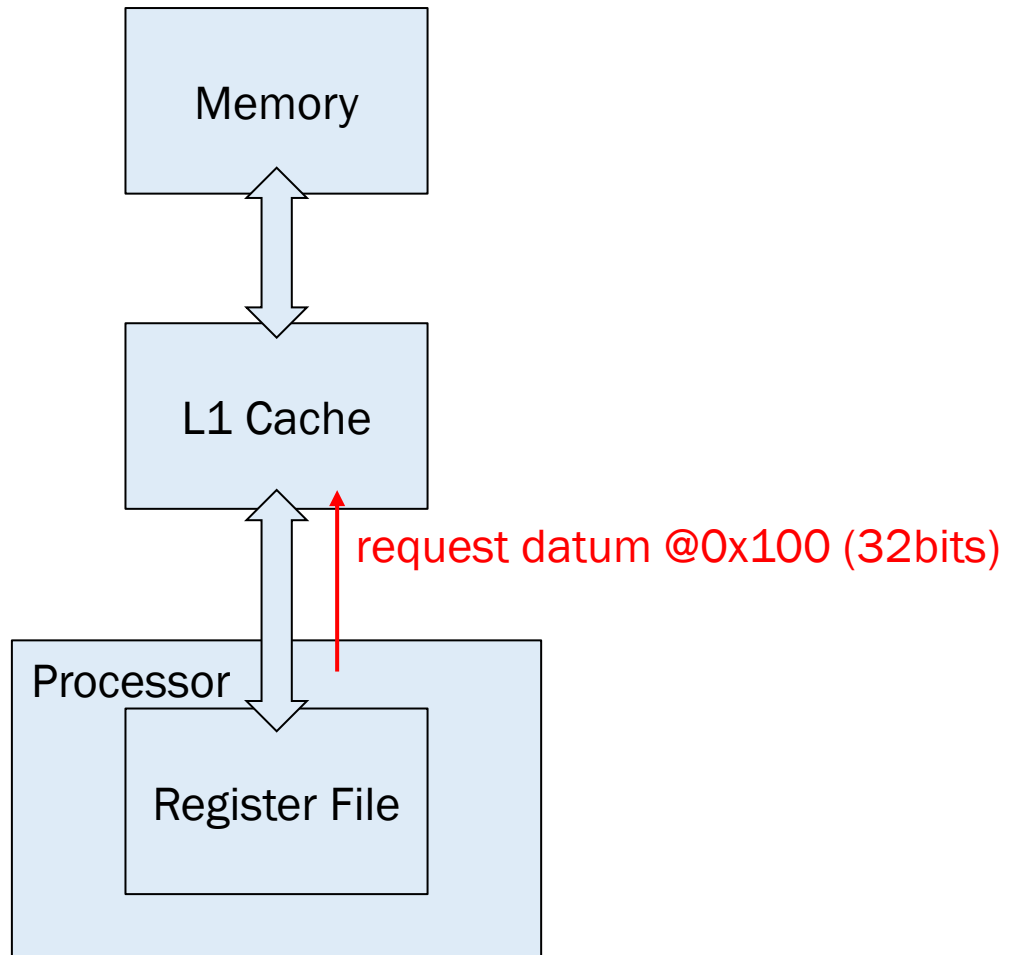


# Solution: memory hierarchy

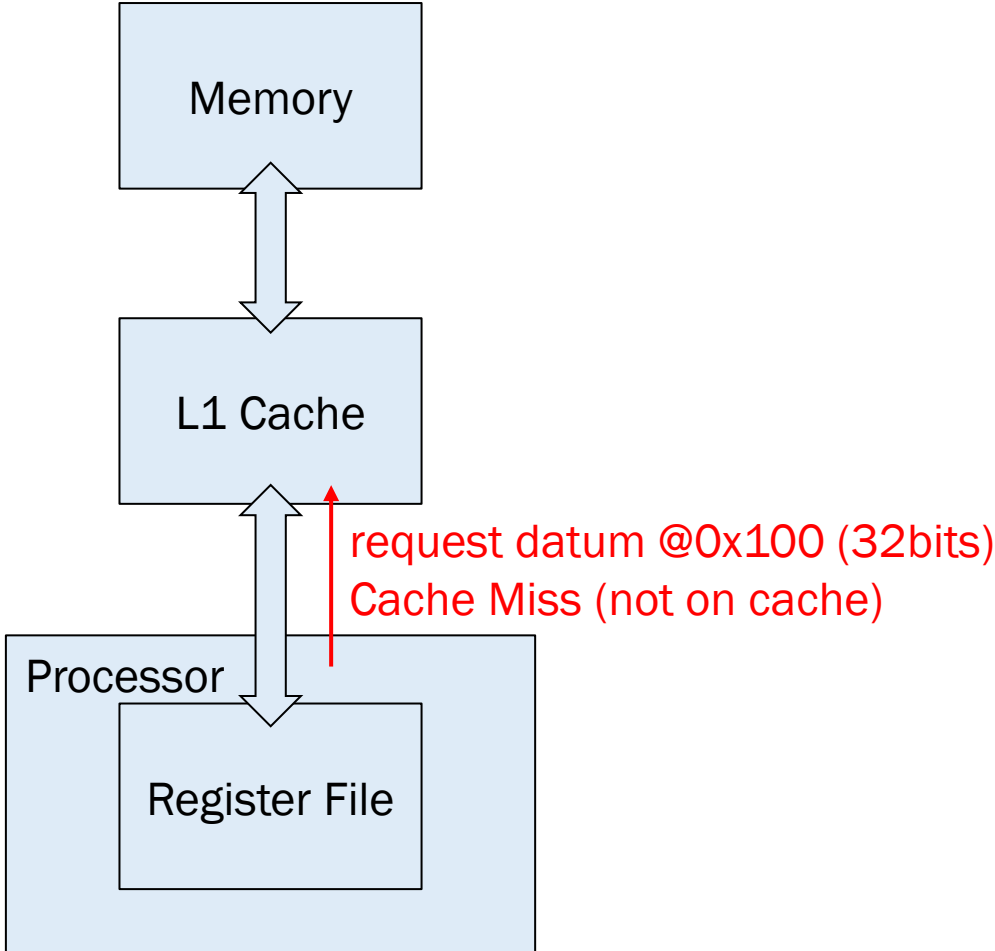
- **Temporal locality**
  - If you used the datum once, you'll probably use it again
- **Spatial locality**
  - If you used the datum once, you'll probably use the data close to it soon

```
for(i=0;i<100;i++)    //i: temporal locality  
    my_array[i] = 0; //my_array: spatial locality
```

# Cache behavior

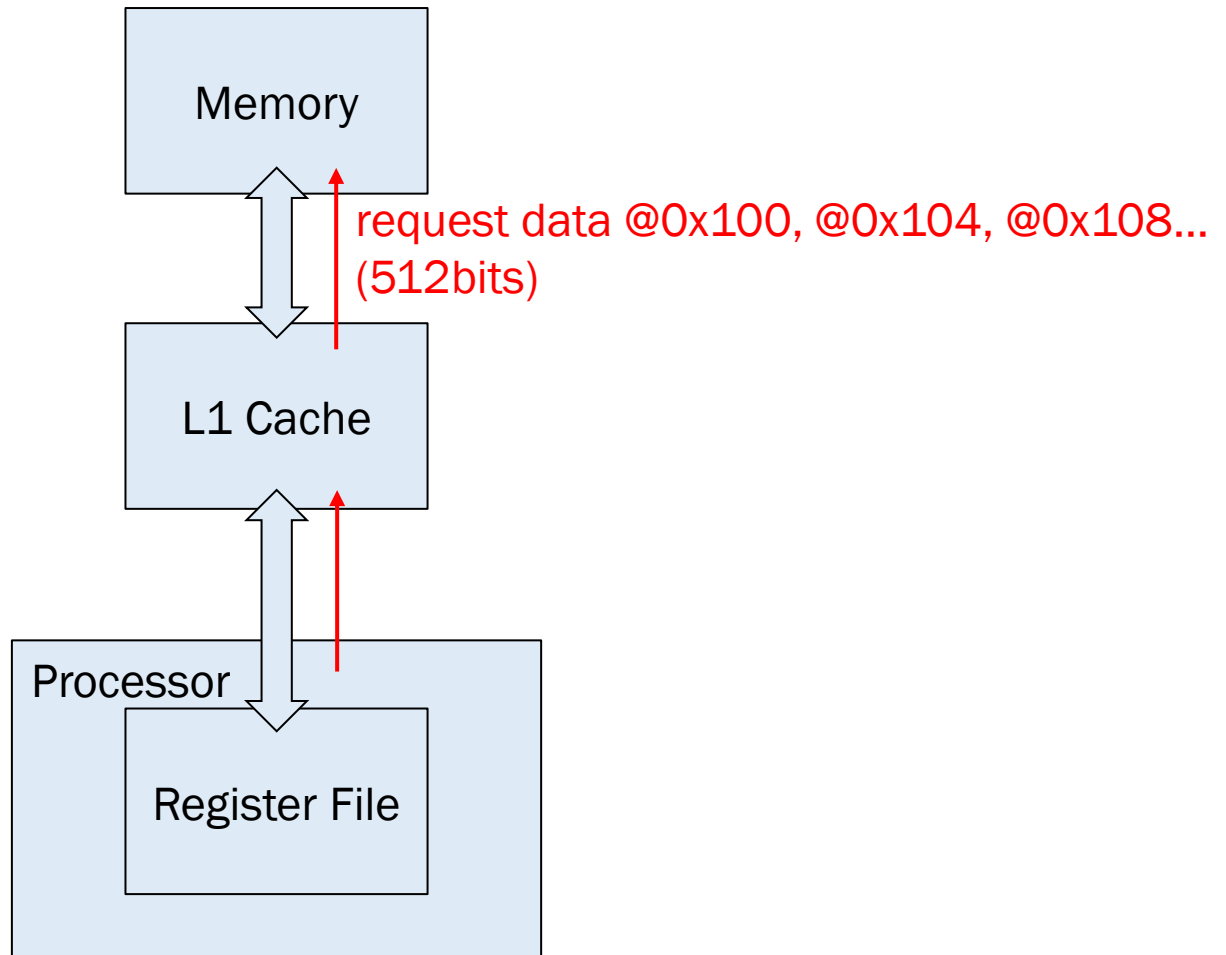


# Cache behavior



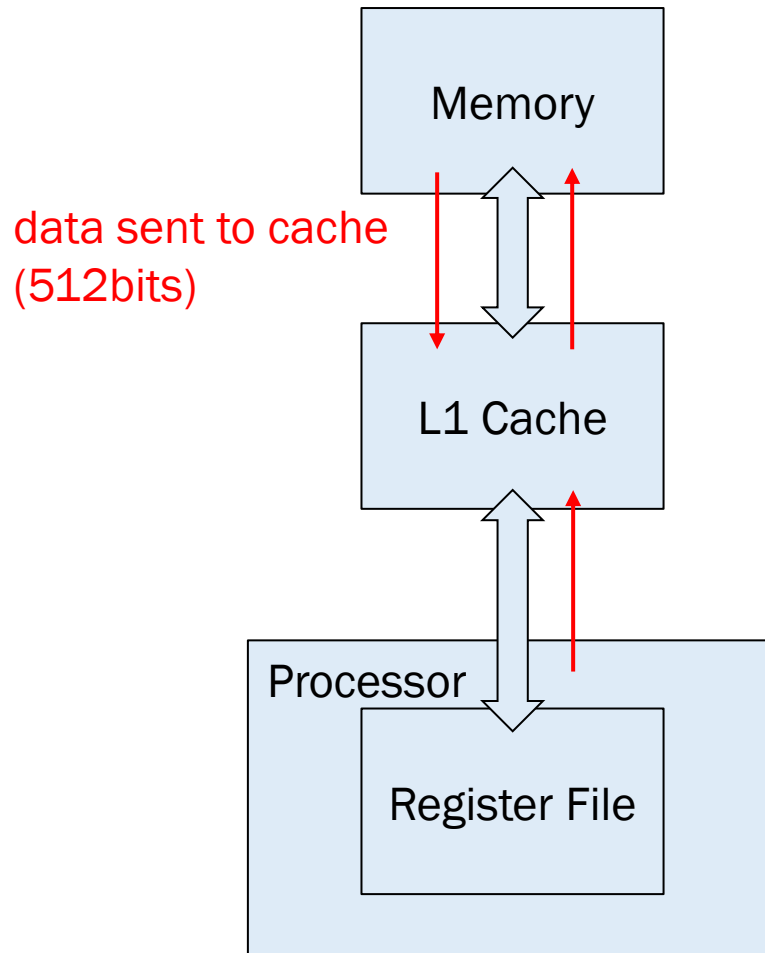
# Cache behavior

- Access to main memory is slow

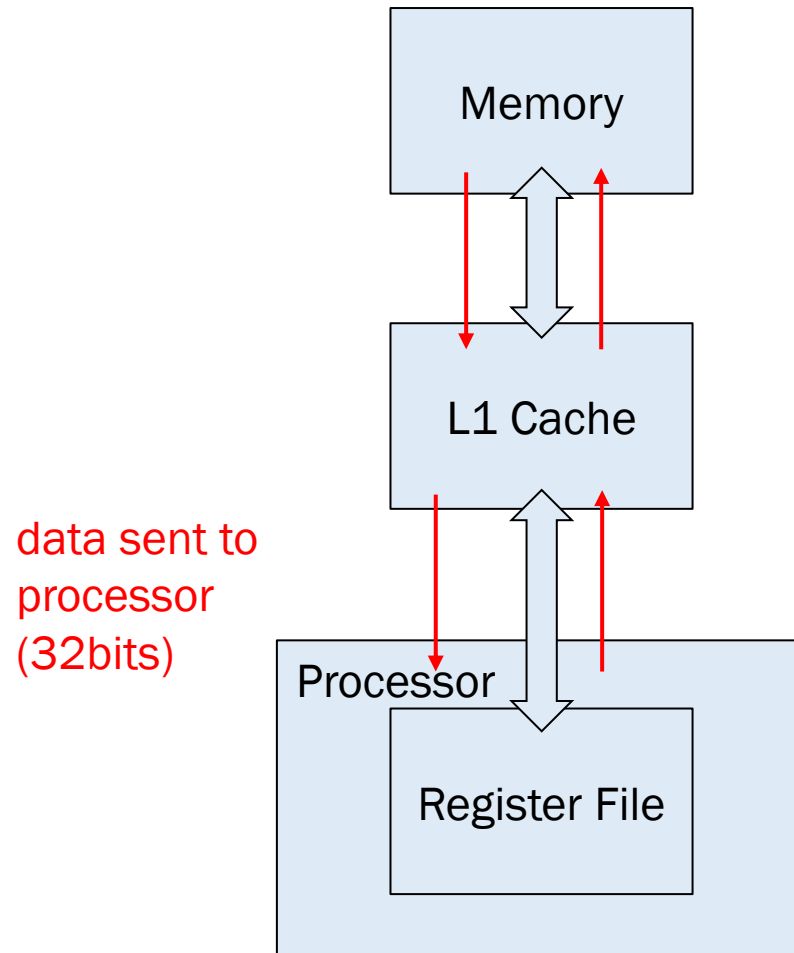




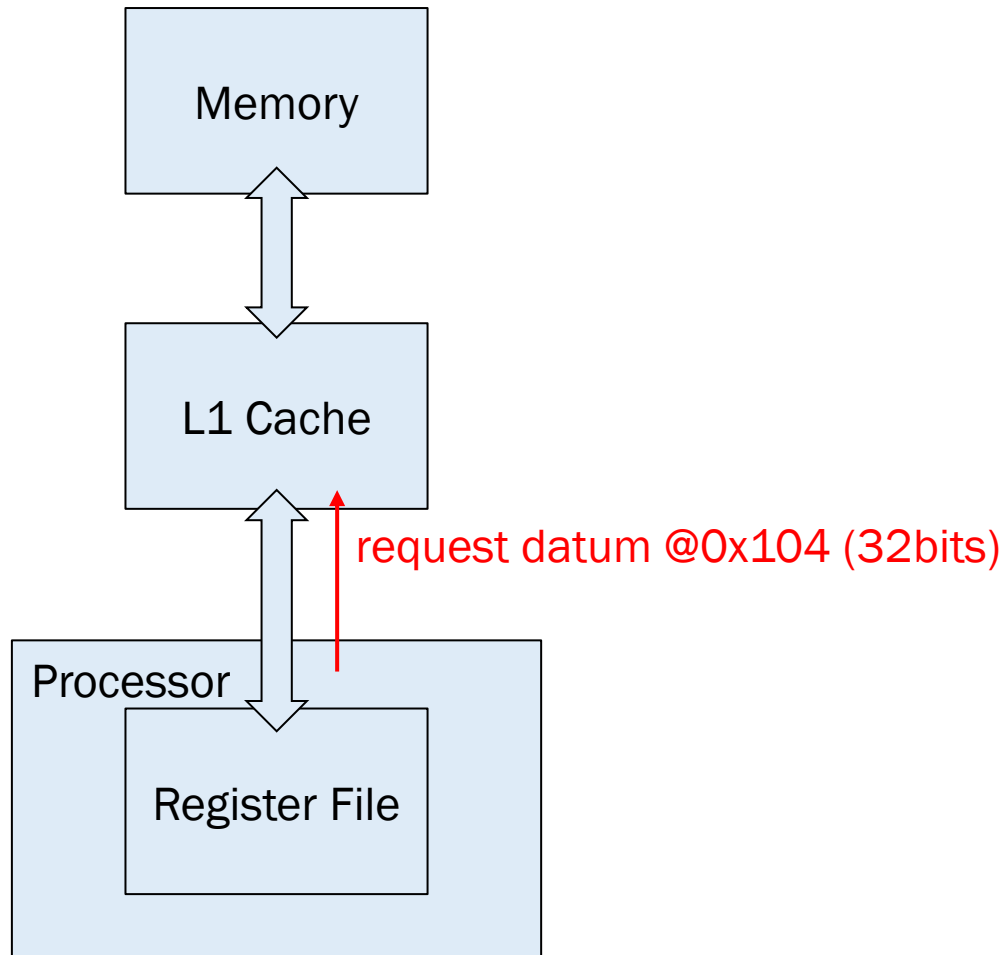
# Cache behavior



# Cache behavior

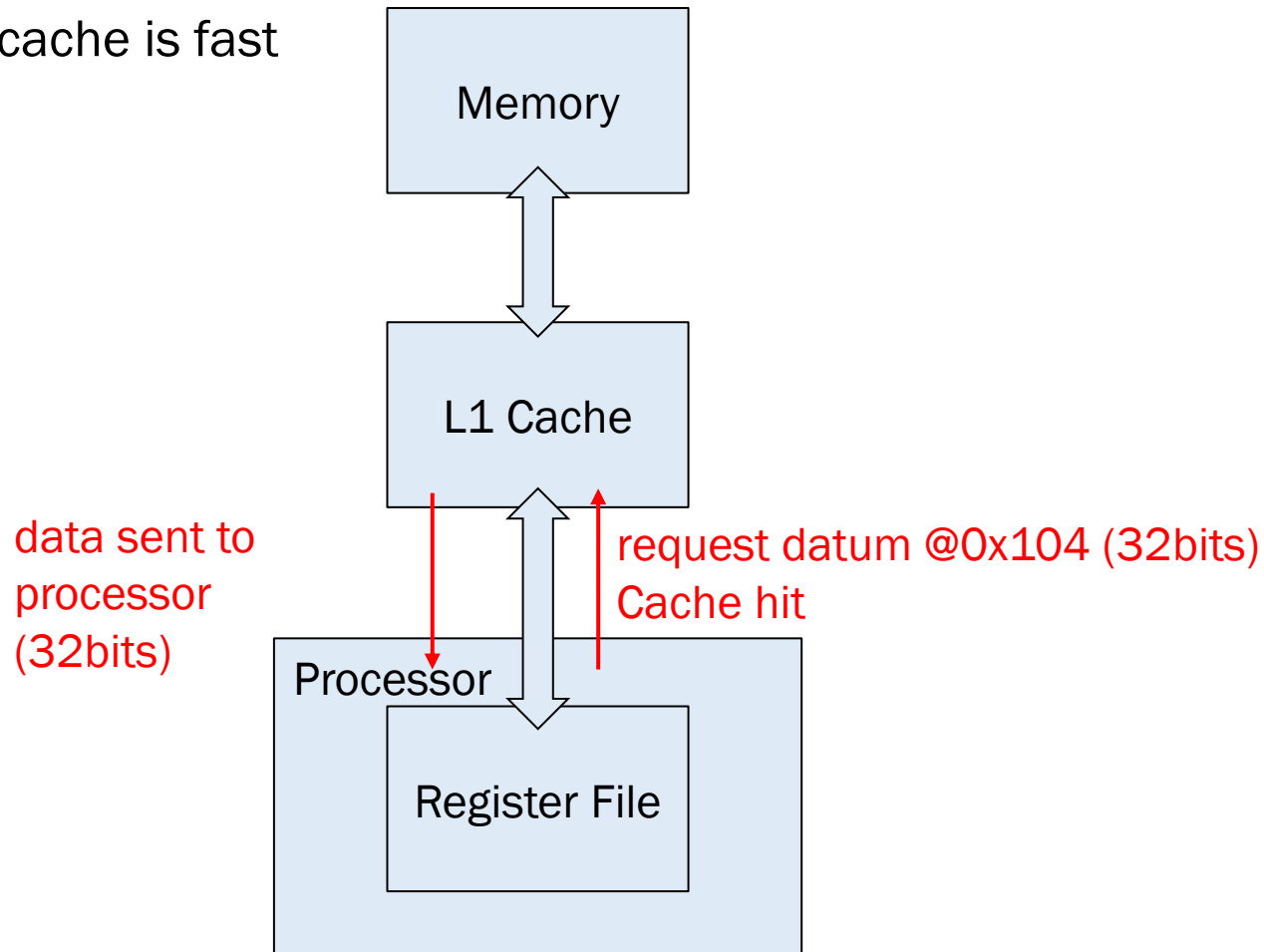


# Cache behavior



# Cache behavior

- Access to cache is fast

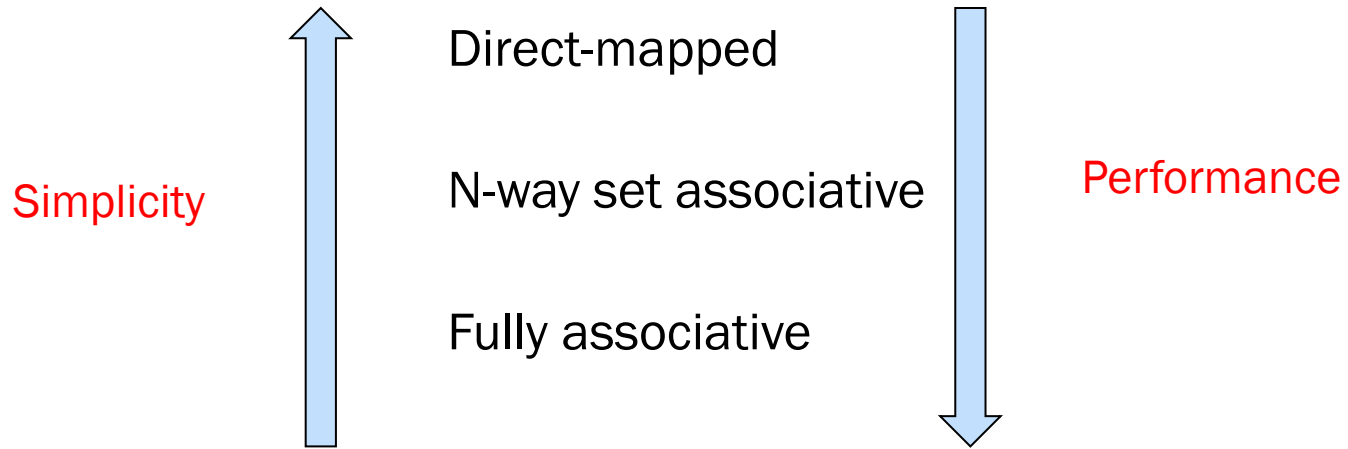


# Cache

- Keep in mind:
  - Except for a few exceptions (we'll look at them soon)...
  - Caches are microarchitectural
    - Software behaves the same way, with or without caches
    - Software is not aware of caches
    - But performance differs

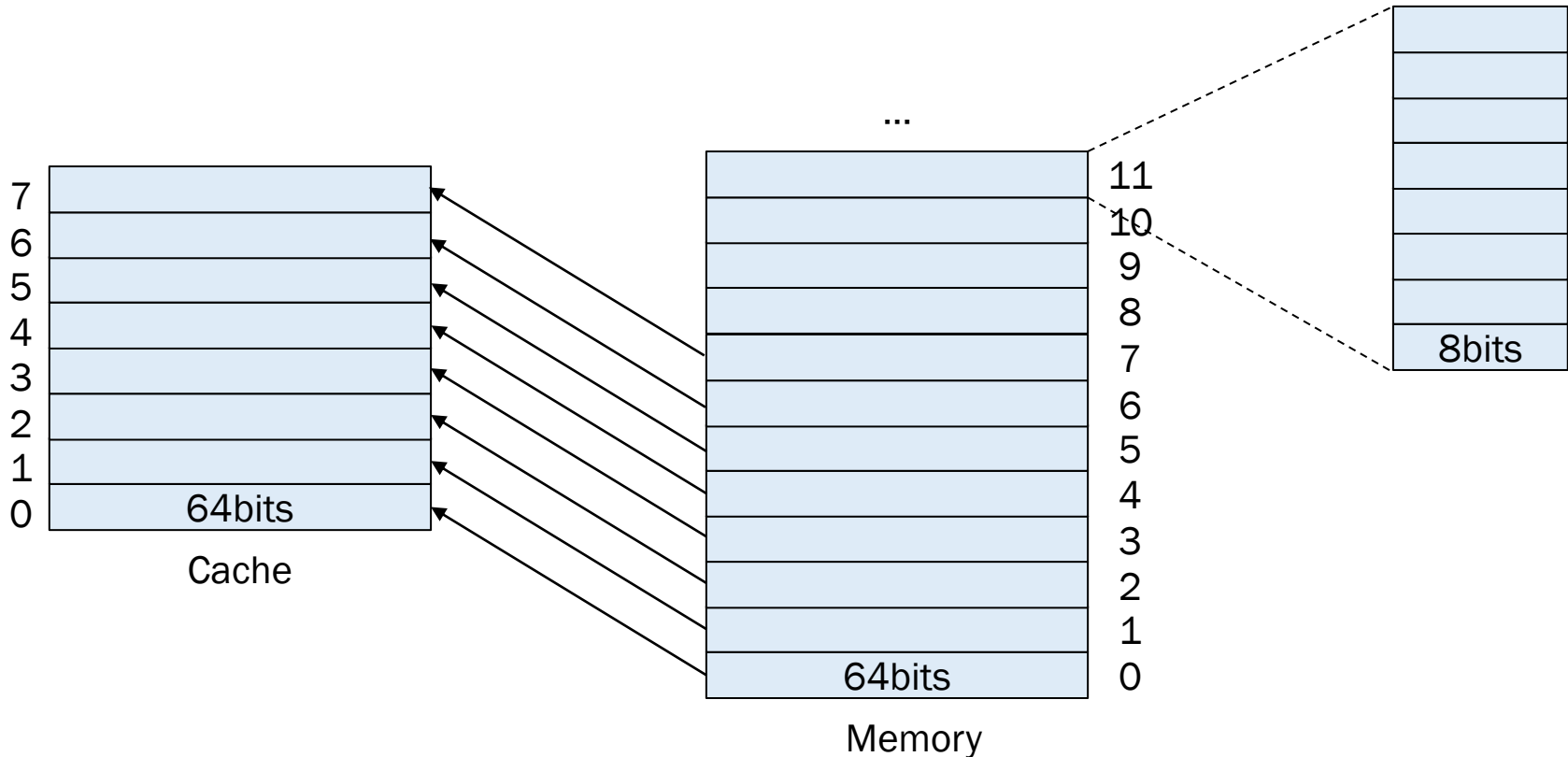
# Cache

- How does a cache work?



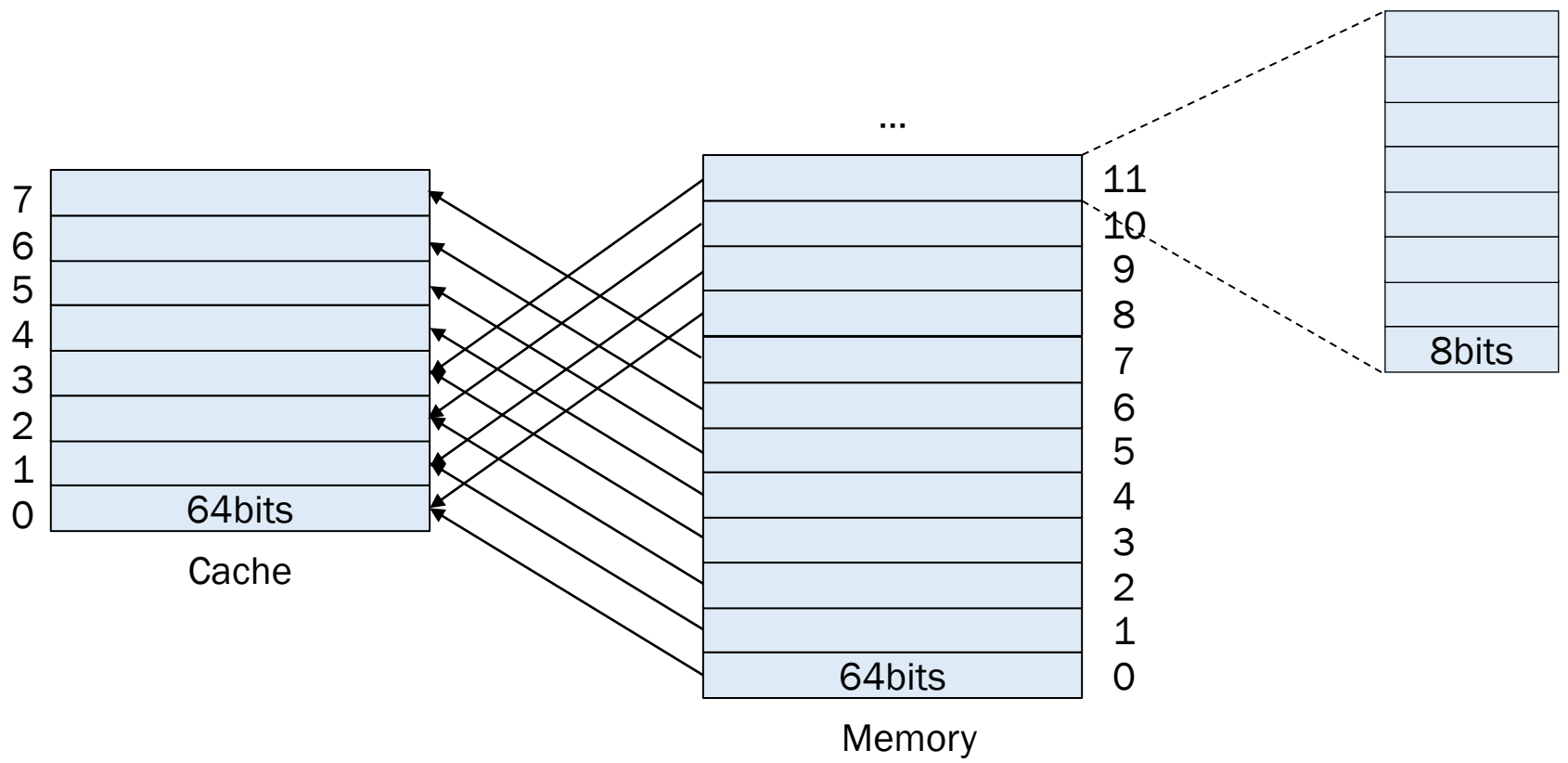
# Direct mapped cache

- Each memory block is directly mapped to a specific cache block
  - E.g., cache has 8 blocks, each 64bits



# Direct mapped cache

- Each memory block is directly mapped to a specific cache block
  - E.g., cache has 8 blocks, each 64bits





# Direct mapped cache

- In this example, both blocks 0 and 8 (memory) are mapped to cache block 0
  - General rule: Cache block = memory block **modulo** number of blocks
  - How do we know which memory block is in cache to determine hit or miss?
    - Each cache block has an associated **tag**

**Tags indicate which memory block is currently in cache**

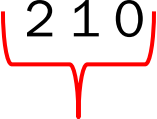
# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes
  - Address bits:           15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

• Address bits:            15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

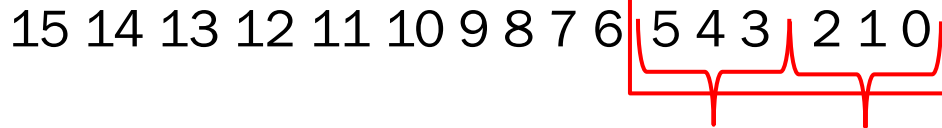


Selects the  
byte in  
block  
(3 bits can  
select 8  
different  
bytes)

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

• Address bits:



Selects which block (3 bits can select 8 different blocks)

Selects the byte in block (3 bits can select 8 different bytes)

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

• Address bits:            15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

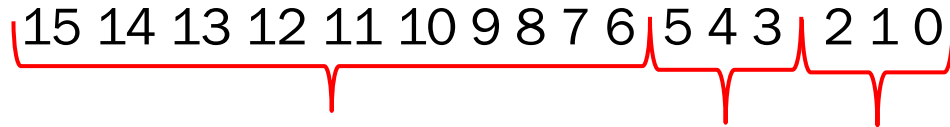
Notice: 6 bits can address 64 different bytes (total cache)

Selects which block (3 bits can select 8 different blocks)  
Selects the byte in block (3 bits can select 8 different bytes)

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

- Address bits:



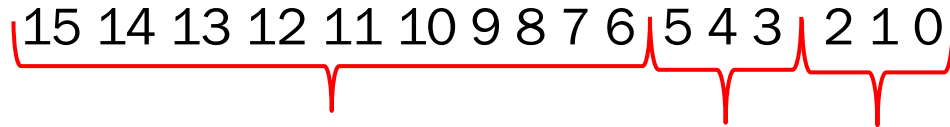
This is the tag: tells us which memory block is in cache

Selects which block (3 bits can select 8 different blocks)      Selects the byte in block (3 bits can select 8 different bytes)

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

• Address bits:



Notice: 10 bits can address 1024 different blocks

This is the tag: tells us which memory block is in cache

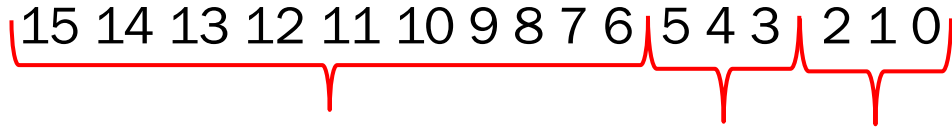
Selects which block (3 bits can select 8 different blocks)  
Selects the byte in block (3 bits can select 8 different bytes)

# Direct mapped cache

- Let's say addresses are 16 bits, access at byte level
  - Processor can address a total range of 65536 bytes (0 - 0xFFFF)
  - Cache has 8 blocks, each 64 bits (8bytes), total of  $8 * 8 = 64$  bytes

$$65536 / 64 = 1024$$

- Address bits:



Notice: 10 bits can address 1024 different blocks

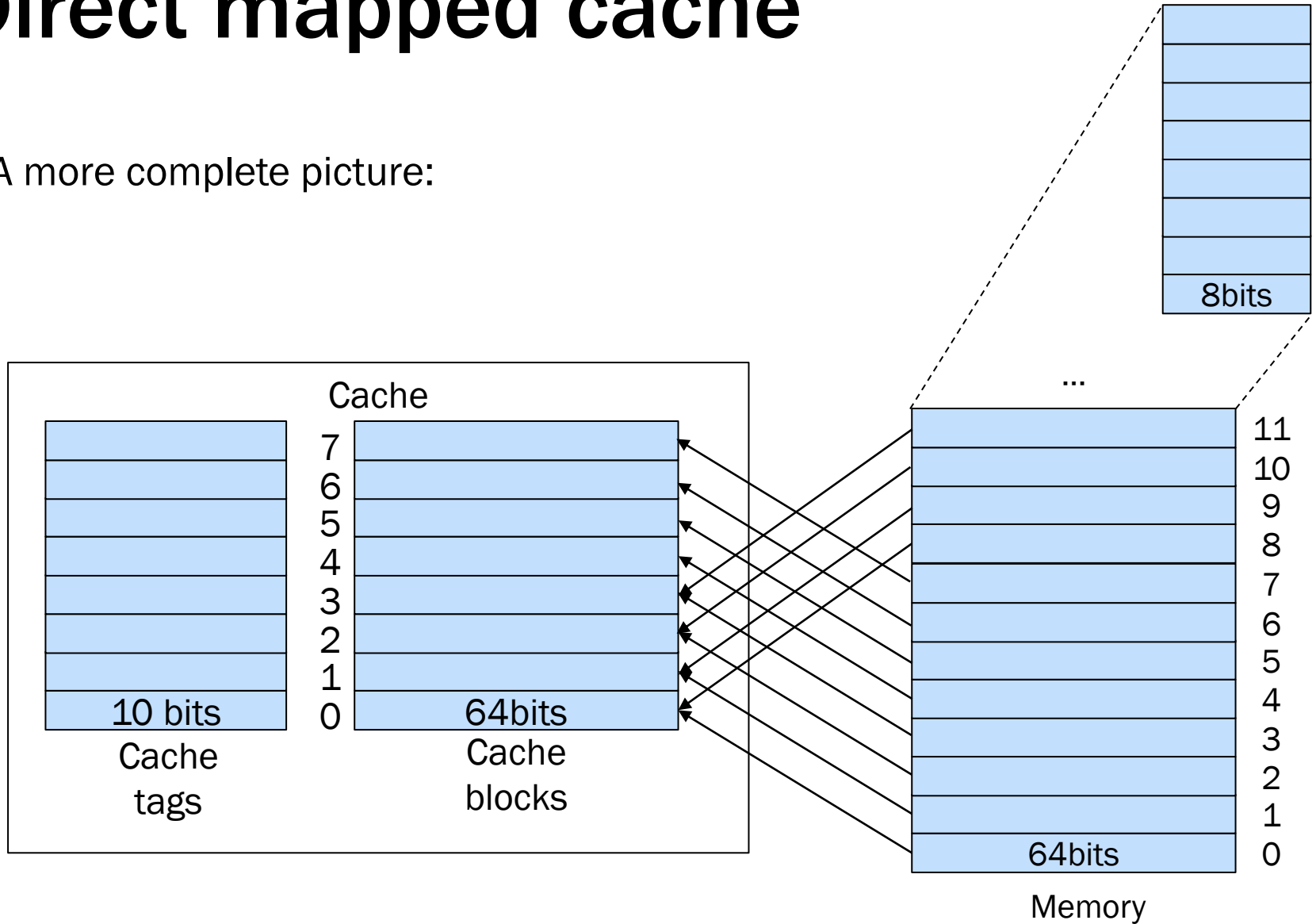
This is the tag: tells us which memory block is in cache

Selects which block (3 bits can select 8 different blocks)  
Selects the byte in block (3 bits can select 8 different bytes)



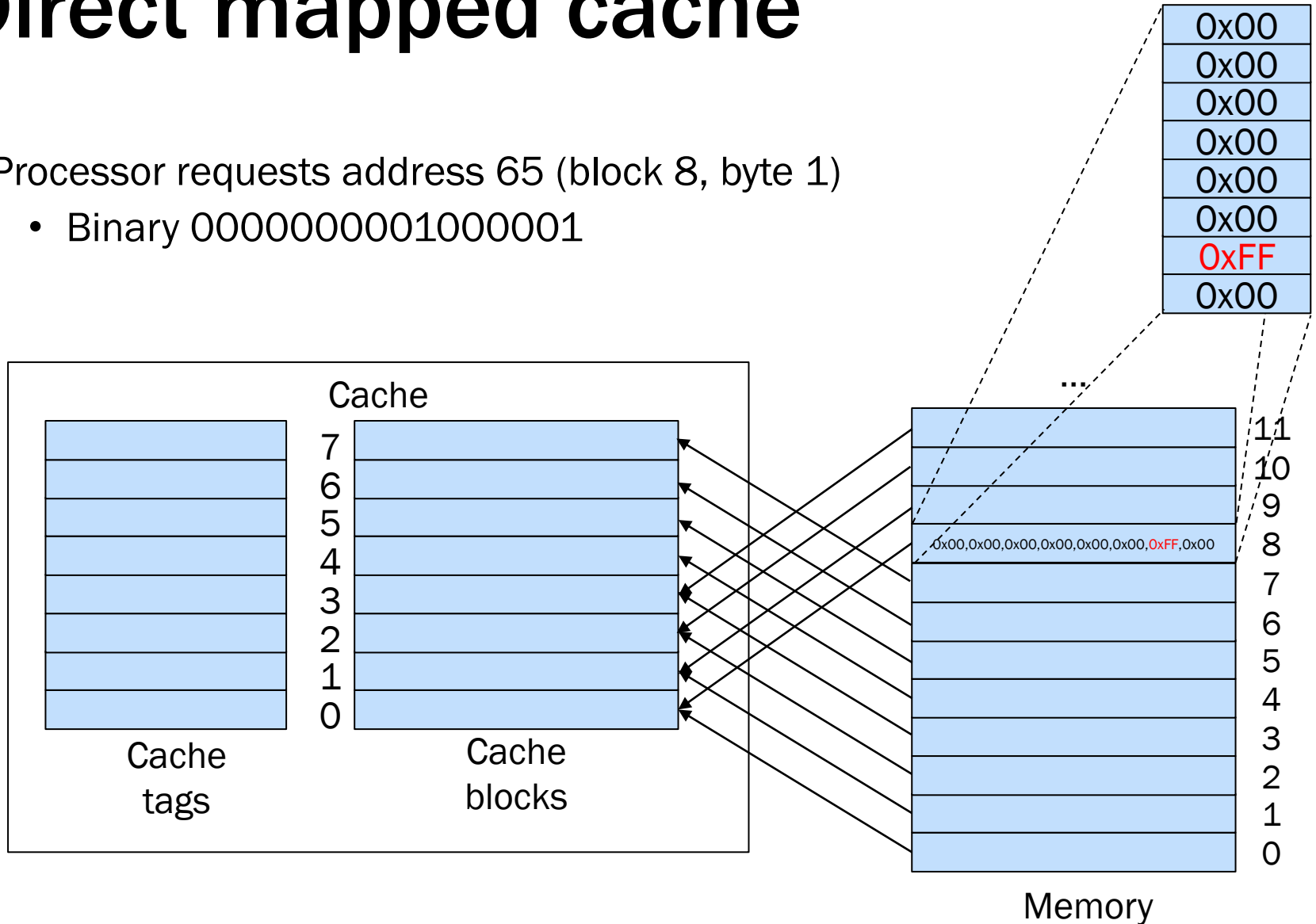
# Direct mapped cache

- A more complete picture:



# Direct mapped cache

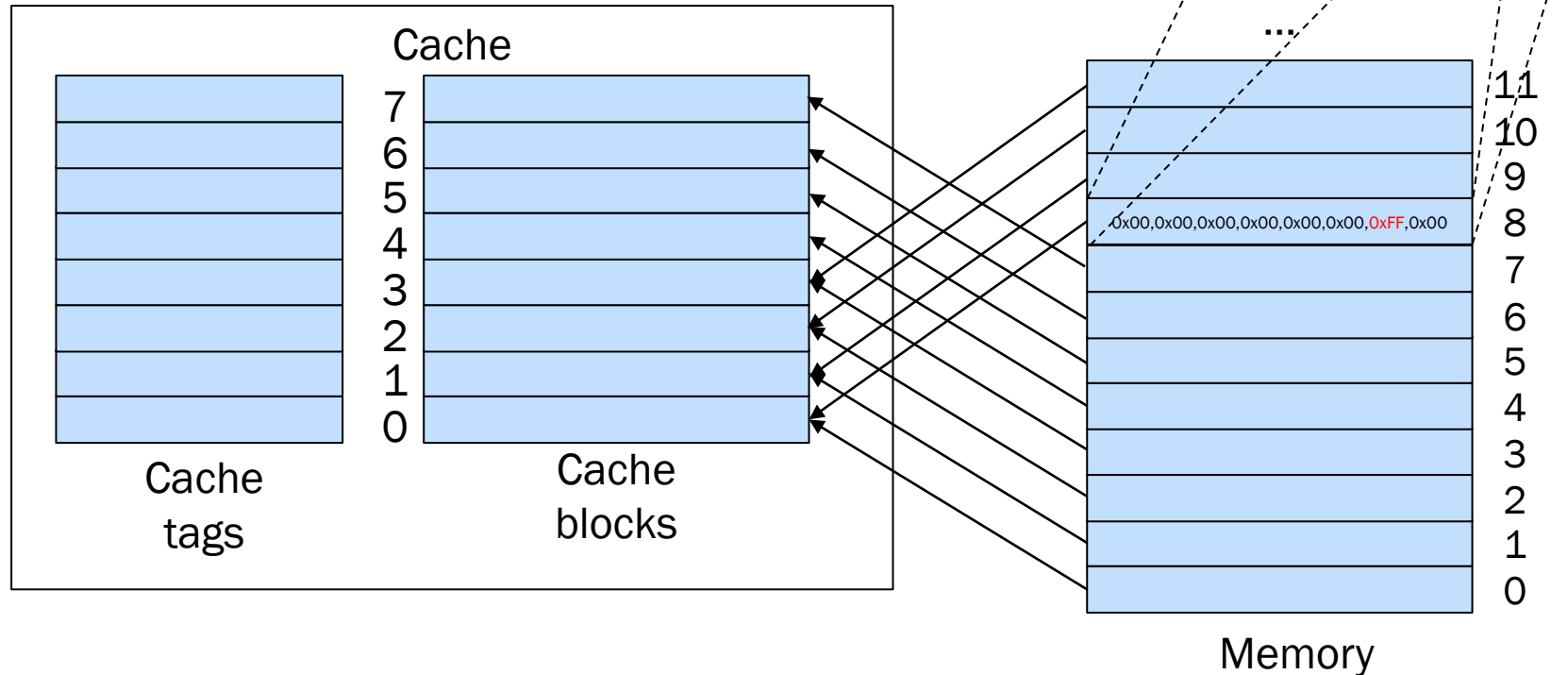
- Processor requests address 65 (block 8, byte 1)
  - Binary 0000000001000001



# Direct mapped cache

- Processor requests address 65 (block 8, byte 1)
  - Binary 0000000001000 001

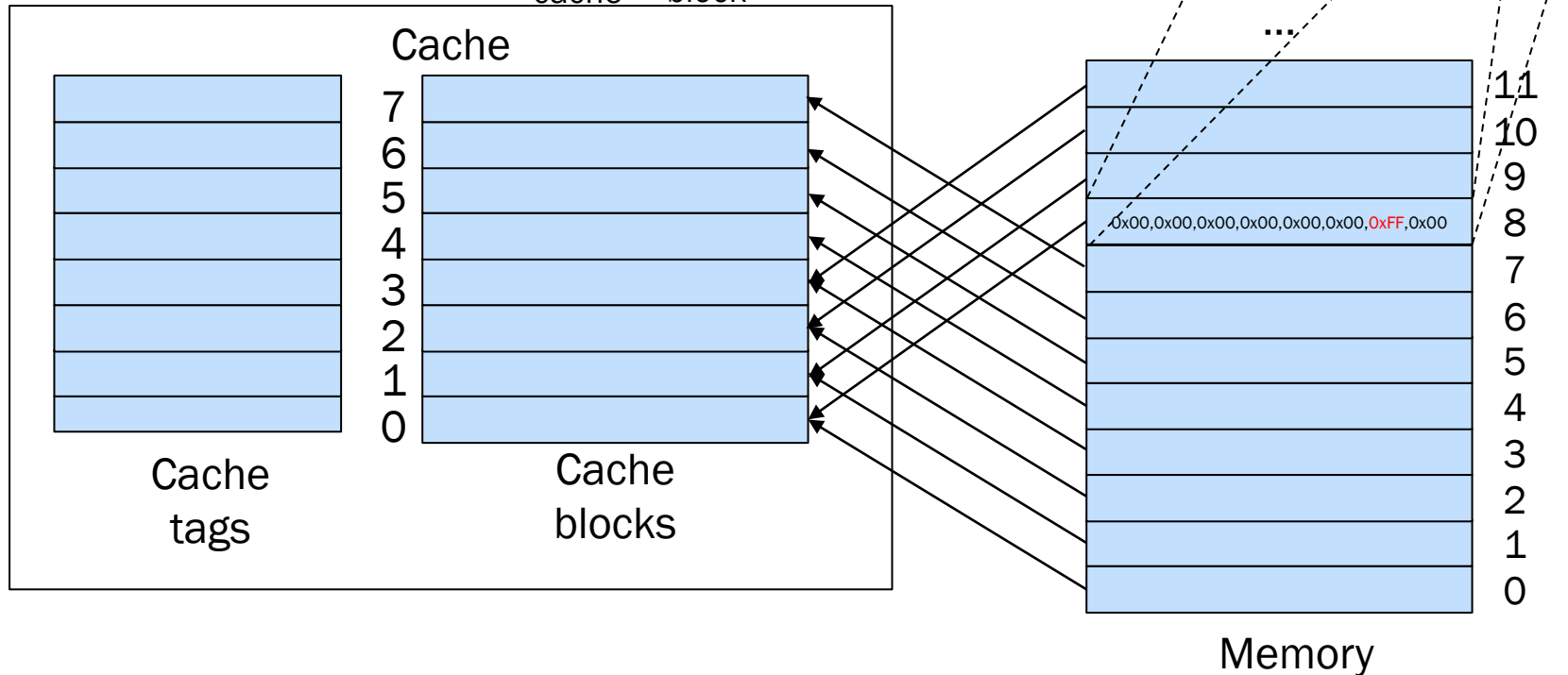
byte in block



# Direct mapped cache

- Processor requests address 65 (block 8, byte 1)
  - Binary 0000000001 000 001

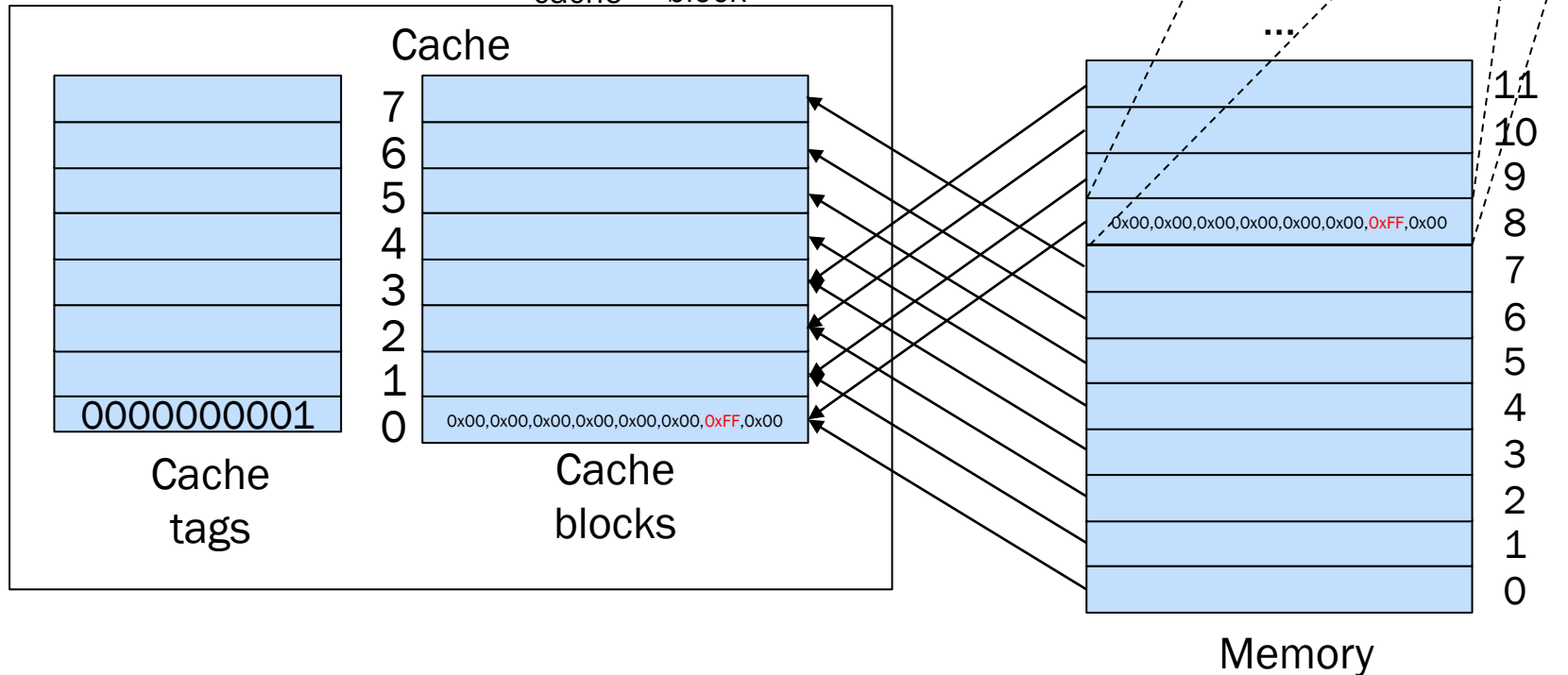
block in cache    byte in block



# Direct mapped cache

- Processor requests address 65 (block 8, byte 1)

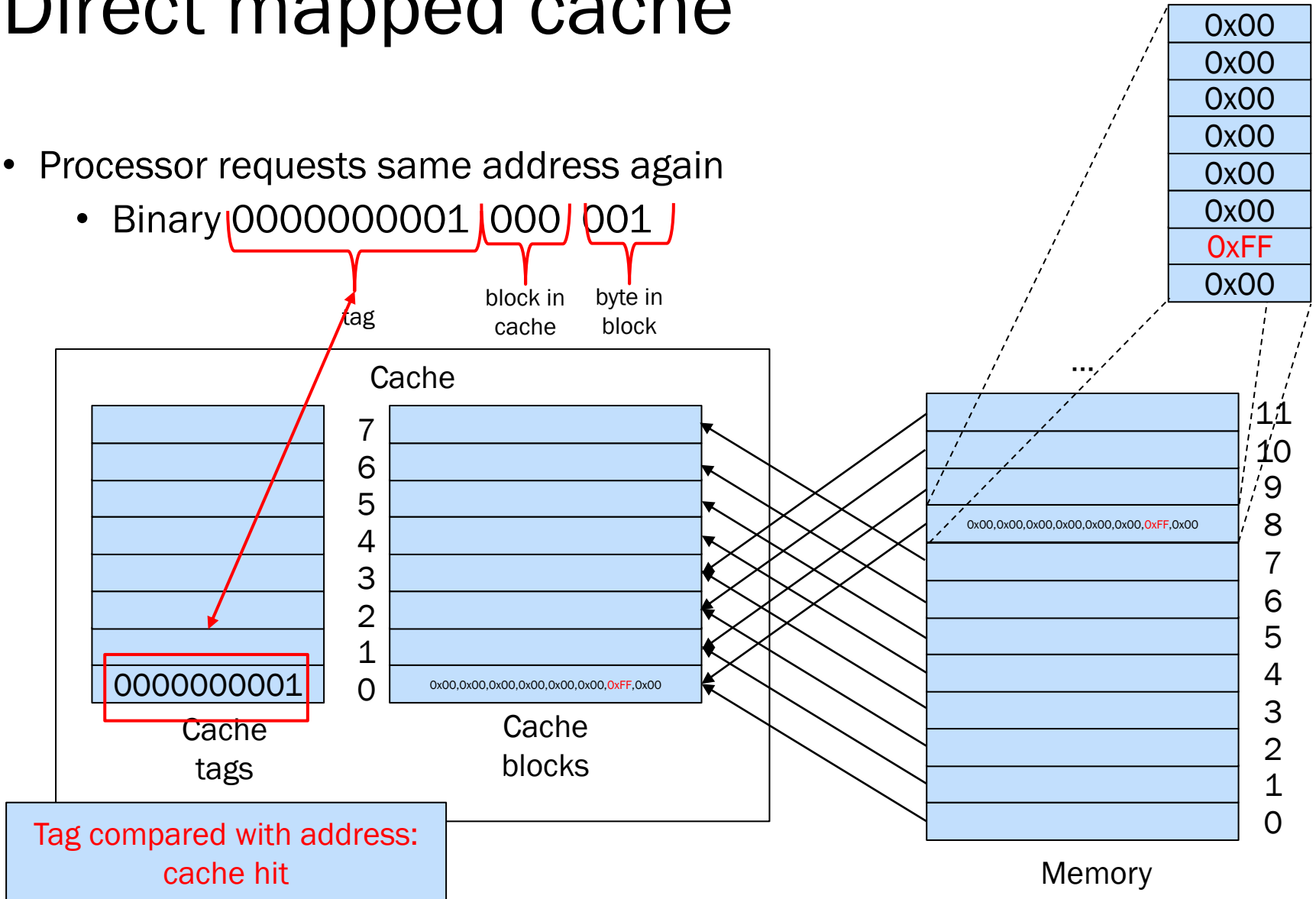
- Binary `0000000001 000 001`  
tag                      block in cache    byte in block



# Direct mapped cache

- Processor requests same address again

- Binary  $0000000001\ 000\ 001$ 
  - tag
  - block in cache
  - byte in block



# Direct mapped cache

- Let's examine performance:
  - Memory access takes 100ms
  - Cache access takes 1ms
  - Address sequence (remember, bytes, not blocks):
    - 0, 1, 2, 3, 8, 1, 2, 8

# Direct mapped cache

- Let's examine performance:
  - Memory access takes 100ms
  - Cache access takes 1ms
  - Address sequence (remember, bytes, not blocks):
    - 0, 1, 2, 3, 8, 1, 2, 8
    - Without cache: 800ms



# Direct mapped cache

- Let's examine performance:
  - Memory access takes 100ms
  - Cache access takes 1ms
  - Address sequence (remember, bytes, not blocks):
    - 0, 1, 2, 3, 8, 1, 2, 8
    - With cache:

# Direct mapped cache

- Let's examine performance:

- Memory access takes 100ms
- Cache access takes 1ms

- Address sequence (remember, bytes, not blocks):

- 0, 1, 2, 3, 8, 1, 2, 8

- **With cache:**

- Cache miss (100ms) – but loads the whole block 0 (bytes 0 to 7)

Misses: 1  
Hits: 0

# Direct mapped cache

- Let's examine performance:
  - Memory access takes 100ms
  - Cache access takes 1ms
  - Address sequence (remember, bytes, not blocks):

- 0, 1, 2, 3, 8, 1, 2, 8

- **With cache:**
  - 3 cache hits (3ms)

Misses: 1  
Hits: 3

# Direct mapped cache

- Let's examine performance:

- Memory access takes 100ms
- Cache access takes 1ms

- Address sequence (remember, bytes, not blocks):

• 0, 1, 2, 3, 8, 1, 2, 8

- **With cache:**

- Cache miss (100ms) – but loads the whole block 1 (bytes 8 to 15)

Misses: 2  
Hits: 3

# Direct mapped cache

- Let's examine performance:
  - Memory access takes 100ms
  - Cache access takes 1ms
  - Address sequence (remember, bytes, not blocks):

• 0, 1, 2, 3, 8, 1, 2, 8

- With cache:
- 3 cache hits (3ms)

Misses: 2  
Hits: 6

# Direct mapped cache

- Let's examine performance:

- Memory access takes 100ms
- Cache access takes 1ms

- Address sequence (remember, bytes, not blocks):

- 0, 1, 2, 3, 8, 1, 2, 8

- With cache:

- Total time: 203ms (4x faster than without cache)

Misses: 2

Hits: 6

Hit Rate = 75%

# Direct mapped cache

- To determine hit rates (whether something is on cache or not), you need to consider:
  - Number of cache blocks
  - Block size
  - For every address, where is it mapped in cache, and how many other addresses are also cached (same block)

# Direct mapped cache

- 8-bit address space (256 addresses)
  - Each cache block is 2 bytes
  - Direct mapped cache, 16 blocks
  - Memory access, 100ms, Cache access, 1ms
  - Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1
    - Hit rate and execution time?

5 minutes



# Direct mapped cache

- 8-bit address space (256 addresses)

- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 1  
Hits: 0  
Time: 100

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache miss (block 0)

# Direct mapped cache

- 8-bit address space (256 addresses)

- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 1  
Hits: 1  
Time: 101

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache hit (block 0)

# Direct mapped cache

- 8-bit address space (256 addresses)

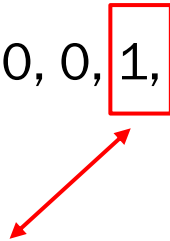
- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 1  
Hits: 2  
Time: 102

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache hit (block 0)



# Direct mapped cache

- 8-bit address space (256 addresses)

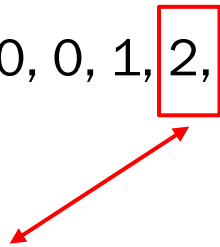
- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 2  
Hits: 2  
Time: 202

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache miss (block 1)



# Direct mapped cache

- 8-bit address space (256 addresses)

- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache miss (block 1)

Misses: 3  
Hits: 2  
Time: 302

# Direct mapped cache

- 8-bit address space (256 addresses)

- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache miss (block 1)

Misses: 4  
Hits: 2  
Time: 402

# Direct mapped cache

- 8-bit address space (256 addresses)

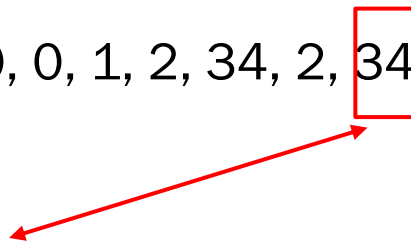
- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 5  
Hits: 2  
Time: 502

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache miss (block 1)



# Direct mapped cache

- 8-bit address space (256 addresses)

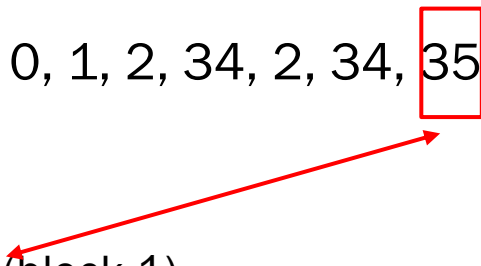
- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

Misses: 5  
Hits: 3  
Time: 503

- Memory access, 100ms, Cache access, 1ms

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache hit (block 1)





# Direct mapped cache

- 8-bit address space (256 addresses)

- Each cache block is 2 bytes
- Direct mapped cache, 16 blocks

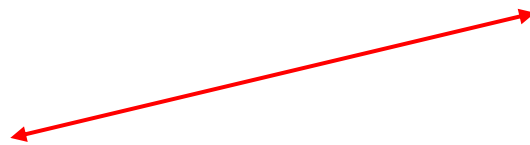
Misses: 5  
Hits: 4  
Time: 504

- Memory access, 100ms, Cache access, 1ms

Hit rate: 44%

- Address sequence: 0, 0, 1, 2, 34, 2, 34, 35, 1

- Cache hit (block 0)



# Direct mapped cache

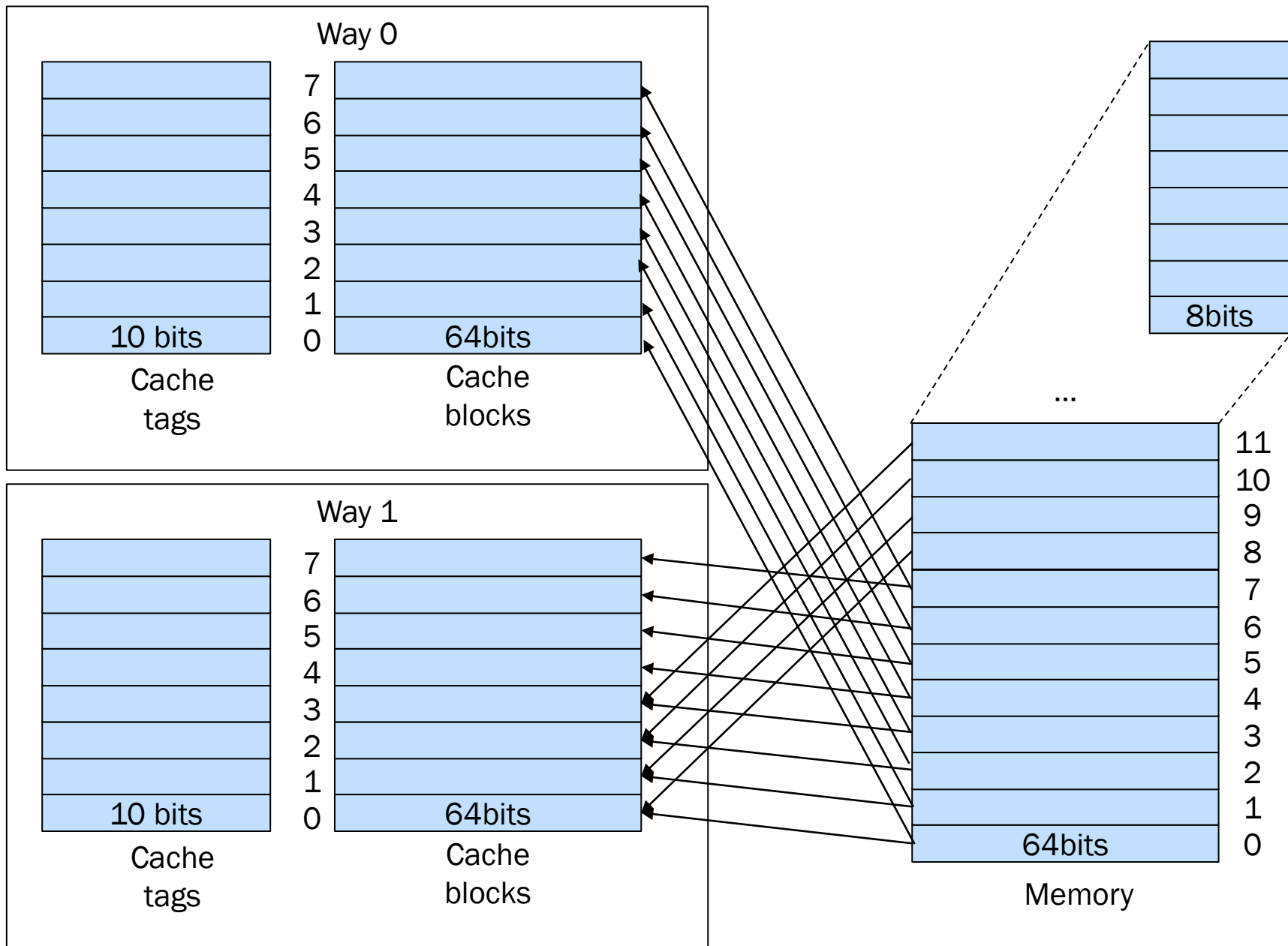
- Works pretty well
- Simple to implement
- Downside:
  - Performance suffers if there is heavy cache trashing (blocks are constantly being evicted)
  - Address sequence: 0, 64, 0, 64, 0, 64...
    - All mapped to block 0: all cache misses!

# N-way set associative

- Solution:
  - N-way set associative caches
    - More complicated to implement
    - Uses more hardware
    - Potentially much higher performance

# N-way set associative

- Example: 2-Way set associative
  - Basically: 2 sets of tags, 2 sets of blocks



# 2-Way set associative

- If an address is mapped to cache block 0
  - It can be placed in either block 0, way 0
  - Or in block 0, way 1
- This prevents cache thrashing
  - For two addresses that are conflicting
  - More addresses still result in thrashing....
    - Require higher associativity

# 2-Way set associative

- On cache access
  - All tags (every way) are checked in parallel
    - If one of them matches, cache hit: great!
    - If cache miss, one way has to be updated
      - Which one?

# 2-Way set associative

- On cache access
  - All tags (every way) are checked in parallel
    - If one of them matches, cache hit: great!
    - If cache miss, one way has to be updated
      - Which one?
        - Remember: **temporal locality**



# 2-Way set associative

- Keep what has been most recently used
- Replacement algorithm (choosing which way)
  - LRU – Least Recently Used

# 2-Way set associative

- Keep what has been most recently used
- Replacement algorithm (choosing which way)
  - LRU – Least Recently Used
    - 2-Ways is easy:
      - 1 bit – whenever way 0 is accessed, bit cleared; way 1, bit set
      - Way to replace: NOT the bit

# N-Way set associative

- What if there are 4 ways?
  - LRU requires two bits per way
    - so it can store “1<sup>st</sup>”, “2<sup>nd</sup>”, “3<sup>rd</sup>”, “4<sup>th</sup>” least recently accessed per way
    - Plus logic that updates every single counter on every access
    - This is not easy: expensive in terms of hardware

# N-Way set associative

- 8-way:
  - LRU requires 3 bits per way
- 16-way
  - LRU requires 4 bits per way
  - Logic gets expensive very quickly

# N-Way set associative

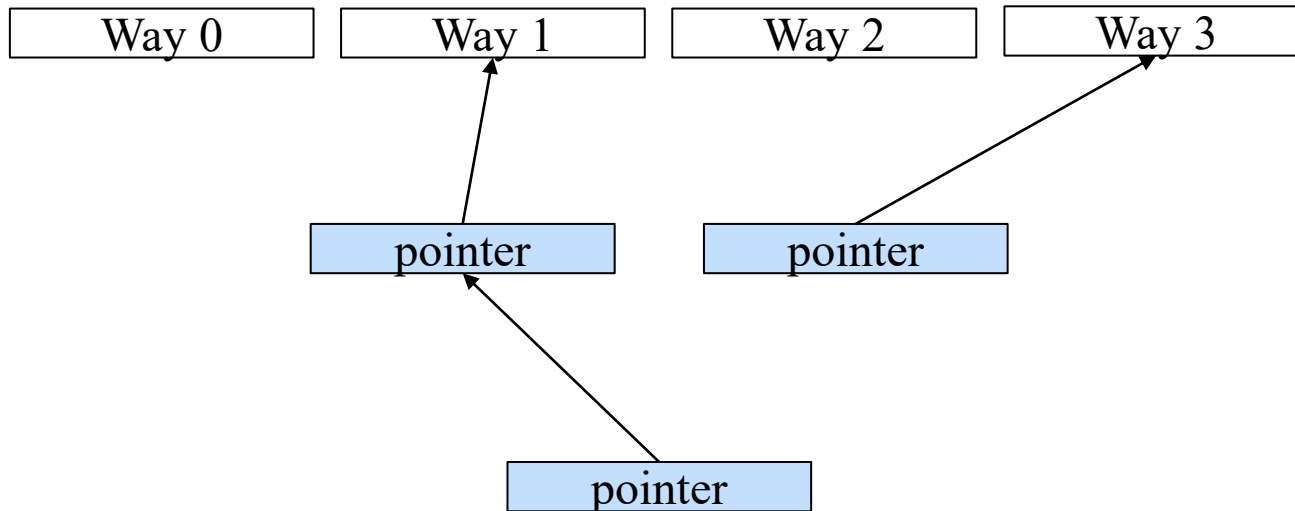
- Which is why N-way set associative caches use
  - Pseudo-LRU
    - Not as efficient in terms of way selection as true LRU
    - But far easier to implement
    - Good trade-off

# Key point

- Something to remember:
  - “Trade-off” is a critical aspect of computer architecture
    - (and all design in general)
  - It’s almost always impossible to improve one aspect without making another worse
  - The art is in choosing something that gives you the right balance

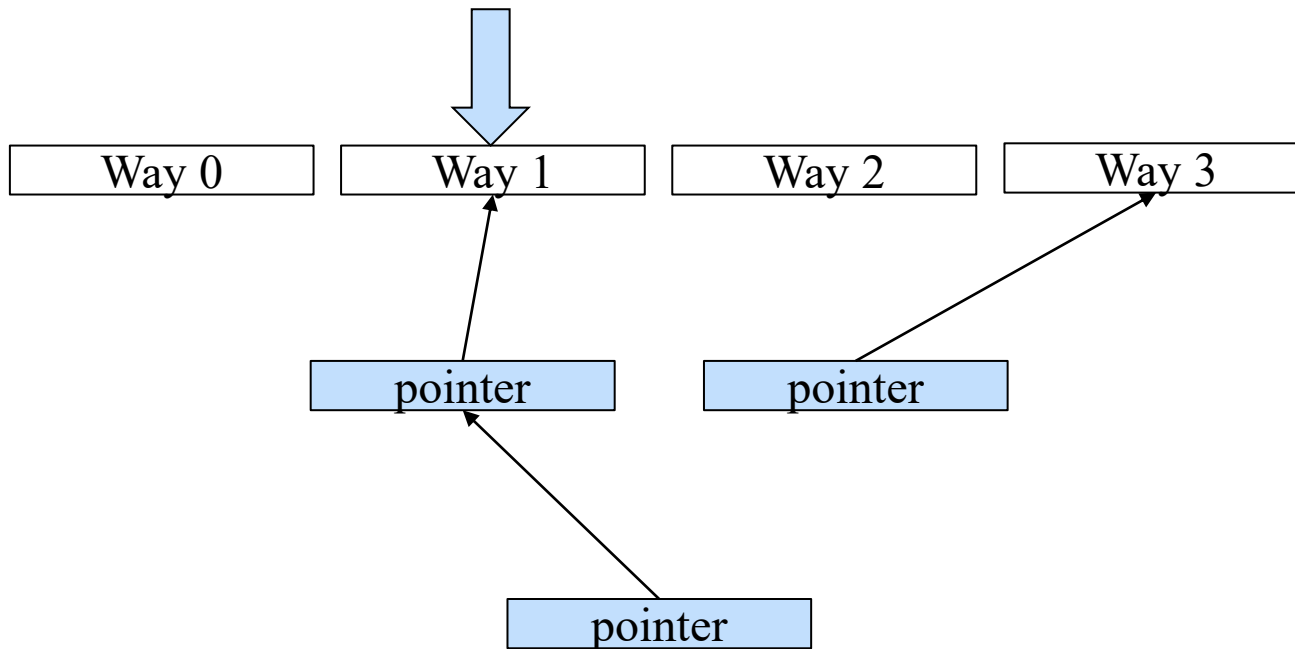
# Pseudo LRU

- Many different implementations
  - Example: pointer chain (1 bit per pointer) on 4-way
  - Always points to (pseudo) least recently used



# Pseudo LRU

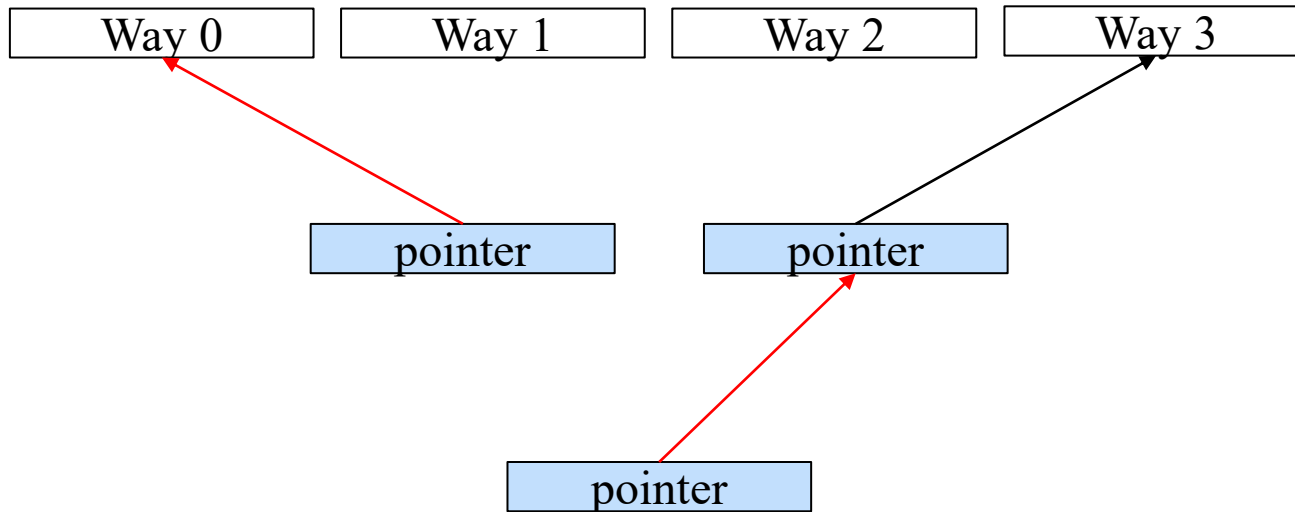
- Pointer chain
  - Access to pointed one: change all correct pointers





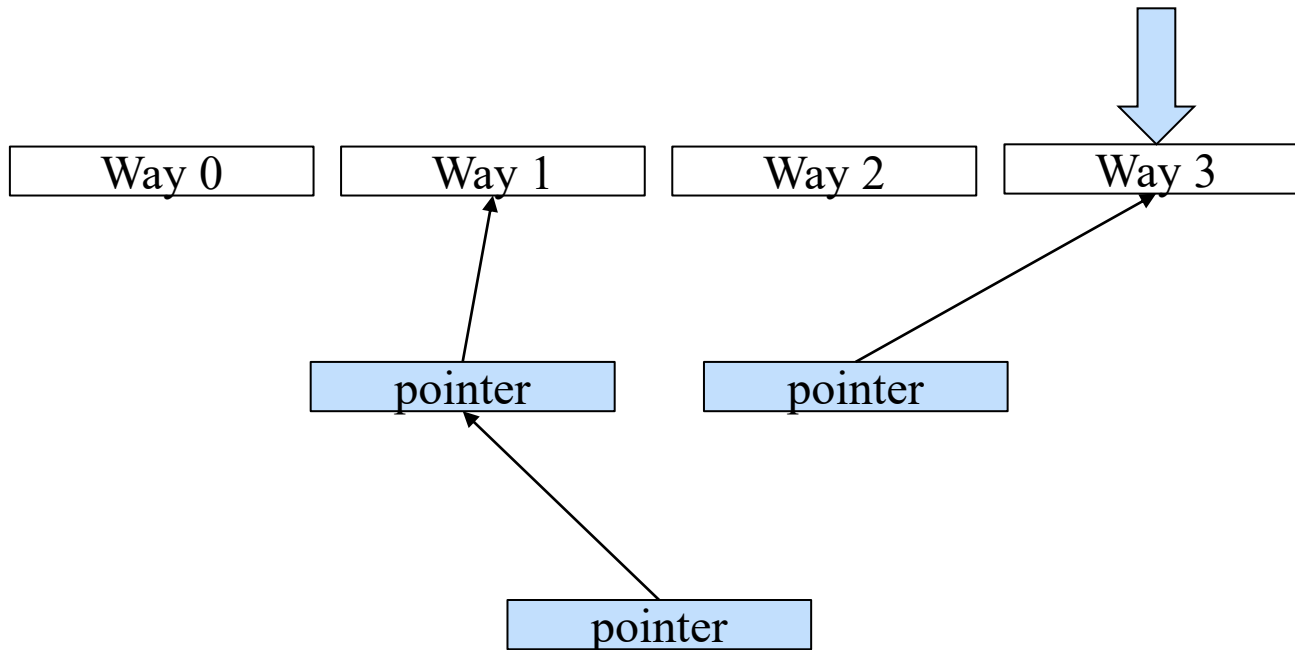
# Pseudo LRU

- Pointer chain
  - Access to pointed one: change all correct pointers



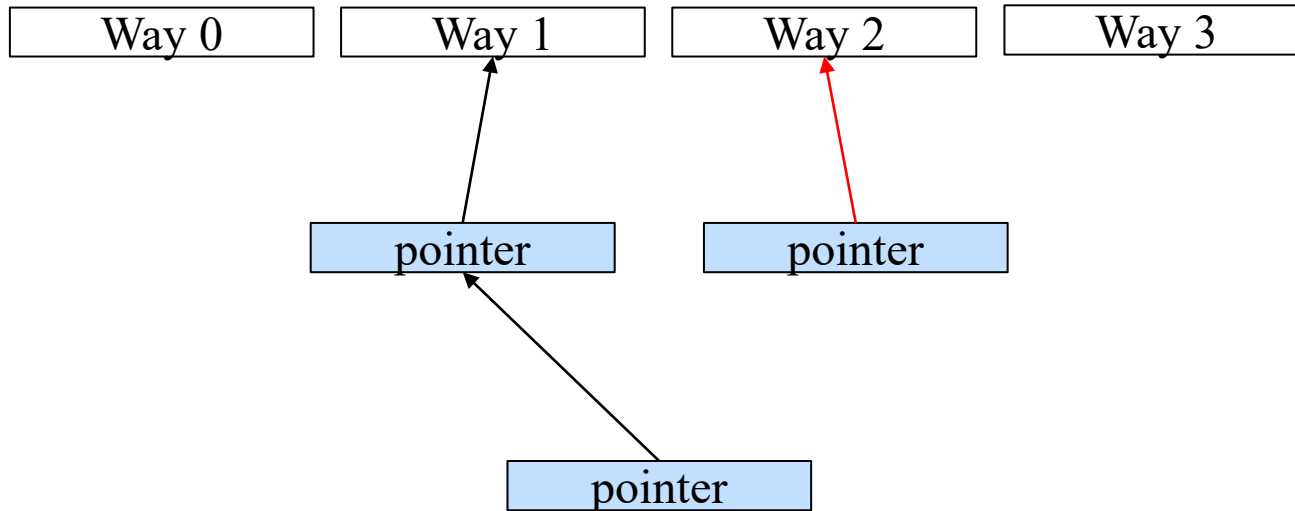
# Pseudo LRU

- Pointer chain
  - Access to pointed one: change all correct pointers



# Pseudo LRU

- Pointer chain
  - Access to pointed one: change all correct pointers



# Can we go further?

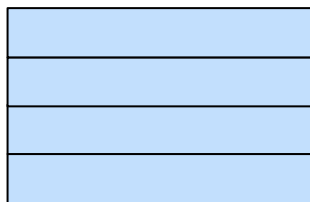
- Direct mapped
  - Each address can only go in one cache block
- N-Way set associative
  - Each address can go into one of N blocks

# Fully associative

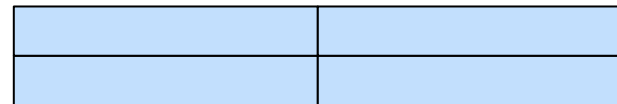
- Any address can go anywhere
  - Again, some pseudo-LRU allocation mechanism

# Fully associative

- Any address can go anywhere
  - Again, some pseudo-LRU allocation mechanism
  - A useful visualization:

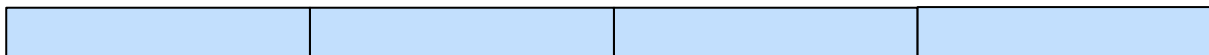


Direct mapped



2-Way set  
associative

Fully associative



# Last bit about caches

- All our examples were about **reading** data from memory
- What about writing?
  - Two ways
    - Write through
    - Write-back

# Last bit about caches

- Write through
  - Cache hit: update in cache, update in memory
  - Cache miss: update in memory
- Write-back
  - Cache hit: update in cache only
    - Will update in memory when datum is evicted from cache



# Last bit about caches

- Write back
  - Cache miss?
  - Depends on policy
    - Allocate on miss: put datum in cache, update in memory when evicted
    - No-allocate on miss: behave like write-through

# Review

- Why the memory hierarchy is important
- Different cache implementations
- Cache replacement policies
  - LRU and Pseudo-LRU
- Write types and policies