

# CPU Caches

COS 316: Principles of Computer System Design

---

*Amit Levy* & Jennifer Rexford

## Why do we cache?

Use caches to mask performance bottlenecks by replicating data nearby

## Design decisions that characterize a cache

- *Look-aside vs. Look-through*
  - determines who is responsible for writing/fetching data from backing store
- *Write-through vs. Write-back*
  - determines whether items changed in the cache are written immediately to the backing store (write-through) or only upon eviction (write-back)
- *Write-allocate vs. Write-no-allocate*
  - determines whether we allocate space for an item when fetching and storing it (write-allocate) or only when fetching (write-no-allocate) it
- *Eviction policy*
  - determines which item(s) to evict when we run out of space in the cache

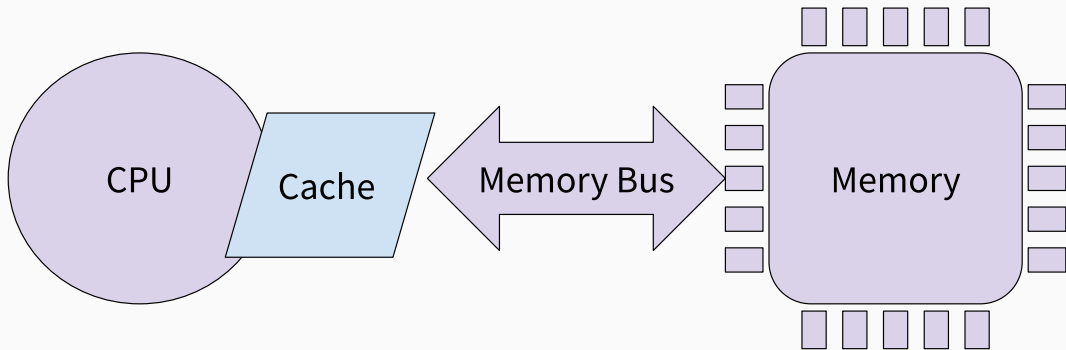


Figure 1: CPU Connected Directly to Memory

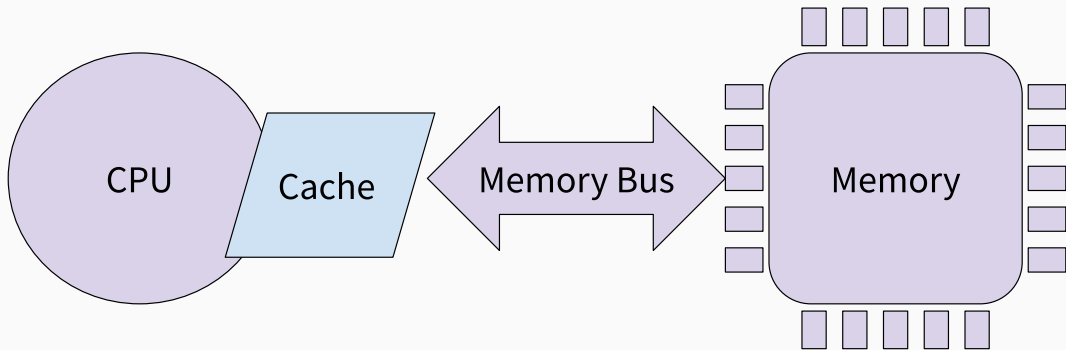


Figure 1: CPU Connected Directly to Memory

Which combination of look-aside vs look-through, write-through vs. write-back, and write-allocate vs. write-no-allocate would you choose?

## Locality: When a cache might be useful

- Useful data tends to continue to be useful



Figure 2: Temporal locality

- Useful data tends to be located “near” other useful data

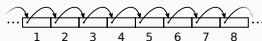


Figure 3: Spatial locality

CPU caches are particularly constrained:

- Size: typically many orders of magnitude smaller than backing store
  - E.g. 64KB L1 Cache vs 64GB main memory. **6 orders of magnitude!**
- Performance: speed, power consumption, physical die space
- General purpose workloads

CPU caches are particularly constrained:

- Size: typically many orders of magnitude smaller than backing store
  - E.g. 64KB L1 Cache vs 64GB main memory. **6 orders of magnitude!**
- Performance: speed, power consumption, physical die space
- General purpose workloads

The trick: exploit physical memory naming scheme



CPU caches exploit both kinds of locality:

- Exploit temporal locality by remembering the contents of recently accessed memory
- Exploit spatial locality by fetching blocks of data around recently accessed memory

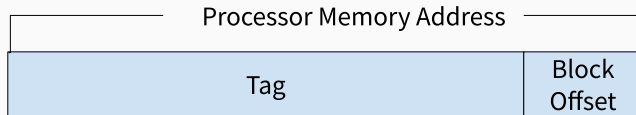


Figure 4: CPU Cache's View of Memory Address

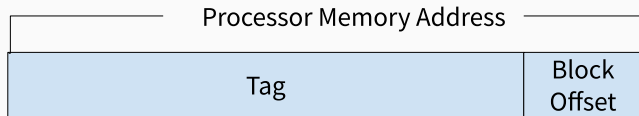


Figure 5: CPU Cache's View of Memory Address

- Addresses with the same tag are added to cache together
  - Spatial locality: bytes around previously accessed byte already in the cache
- Size of block offset determines block size:
  - $n$  bits of block offset means blocks are  $2^n$  bytes
  - E.g. 6 offset bits means 64 byte blocks

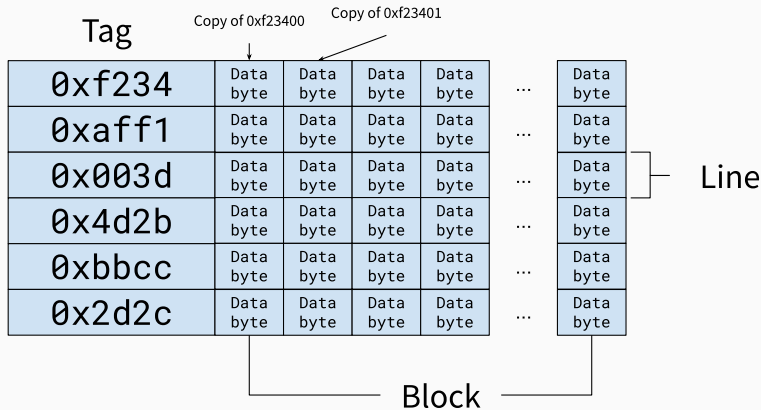


Figure 6: CPU cache stores a block at each Line

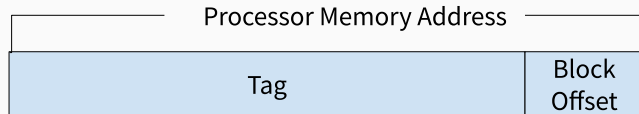
# Cache Read Algorithm

1. Look at memory address on processor
2. Search cache tags to find a matching block
3. Found in cache?
  - Hit: return data from cache at offset from block
  - Miss:
    - 3.1 Read data block from main memory
    - 3.2 Add data to cache
    - 3.3 Return data from cache at offset from block

## Exercise

Starting from 3-line cache that uses 4-bits for the offset, which of the following accesses, performed in order, are hits or misses?

1. `0xff1200df`
2. `0xff1200d3`
3. `0x01cd3310`
4. `0x01cd3310`
5. `0xff1200df`



# Cache Read Algorithm

1. Look at memory address on processor
2. Search cache tags to find a matching block
3. Found in cache?
  - Hit: return data from cache at offset from block
  - Miss:
    - 3.1 Read data block from main memory
    - 3.2 Add data to cache
    - 3.3 Return data from cache at offset from block

# Cache Read Algorithm

1. Look at memory address on processor
2. Search cache tags to find a matching block
3. Found in cache?
  - Hit: return data from cache at offset from block
  - Miss:
    - 3.1 Read data block from main memory
    - 3.2 Add data to cache
    - 3.3 Return data from cache at offset from block

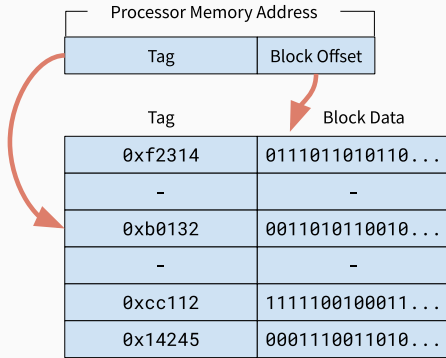
Which line do we evict for the new block?

Three common placement policies:

- Fully Associative
  - Evict with: LRU, FIFO, NLRU, ...
- Direct Mapped
  - Eviction is trivial
- N-way Associative
  - Combination of both

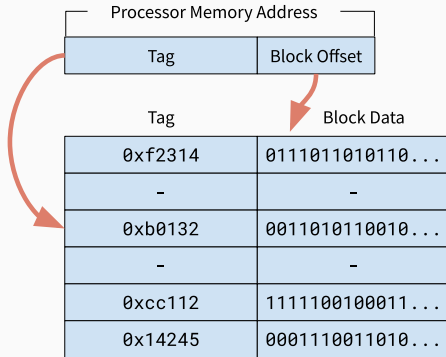


# Fully Associative



Check all lines in the cache for a matching tag

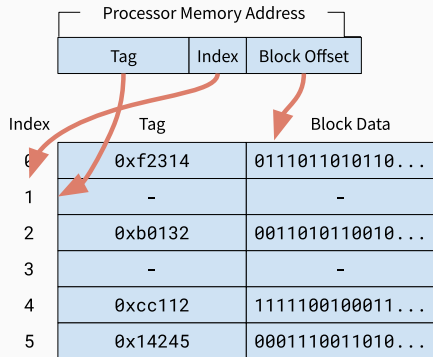
# Fully Associative



Check all lines in the cache for a matching tag

What's the disadvantage of fully associative cache?

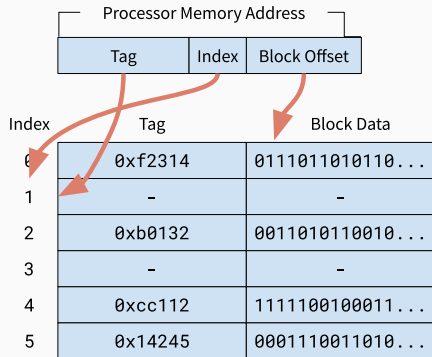
# Direct Mapped



Index size determines number of indices

Check tag at line with matching index: if equal "hit", "miss" otherwise

# Direct Mapped

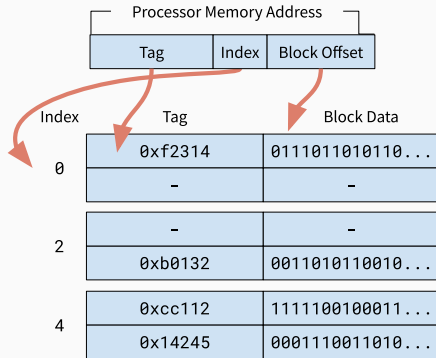


Index size determines number of indices

Check tag at line with matching index: if equal "hit", "miss" otherwise

What's the disadvantage of a direct mapped cache?

# N-way Associative

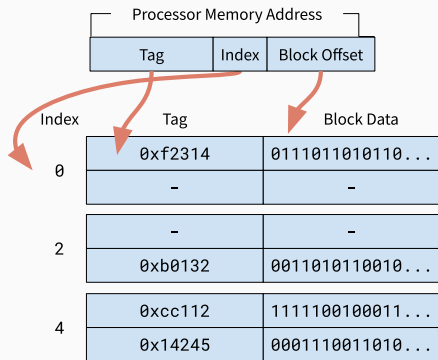


Check *all* tags at line with matching index: if equal “hit”, “miss” otherwise

N = number of lines in each set

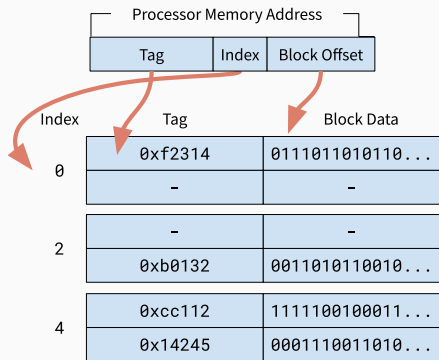
Index size determines number of sets

## Exercise: N-way Associative



How many index bits for a 2-way set associative cache with 128 cache lines?

## Exercise: N-way Associative



How many index bits for a 2-way set associative cache with 128 cache lines?

128 cache lines, 2 lines per set, how many sets?  $128/2 = 64$ , how many bits?  $\log_2(64) = 6$

- Next time: Web caching with CDNs



