18-740 Fall 2010 Computer Architecture Lecture 6: Caching Basics

Prof. Onur Mutlu
Carnegie Mellon University

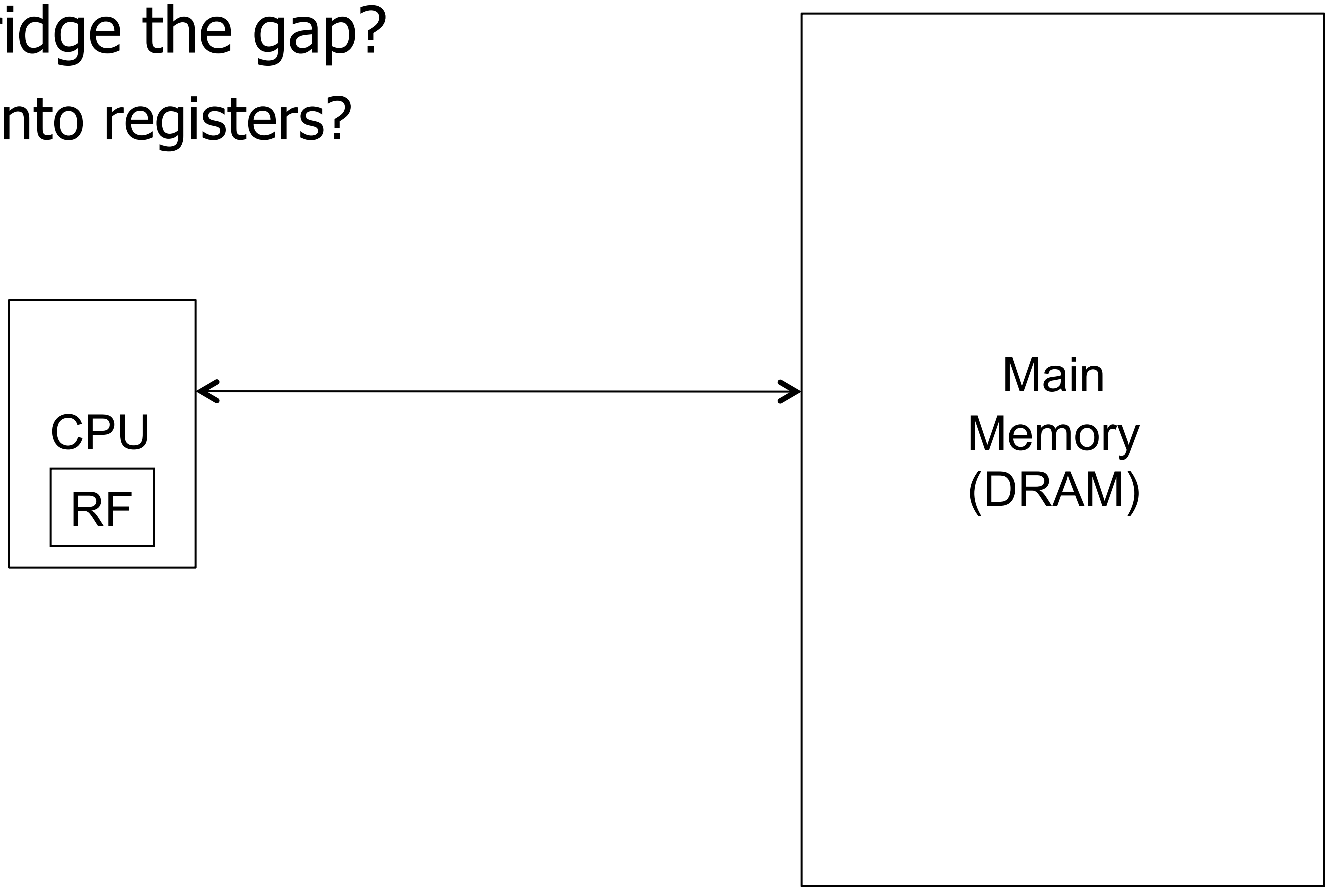
Readings

- Required:
  - Hennessy and Patterson, Appendix C.2 and C.3
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

- Recommended:
  - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

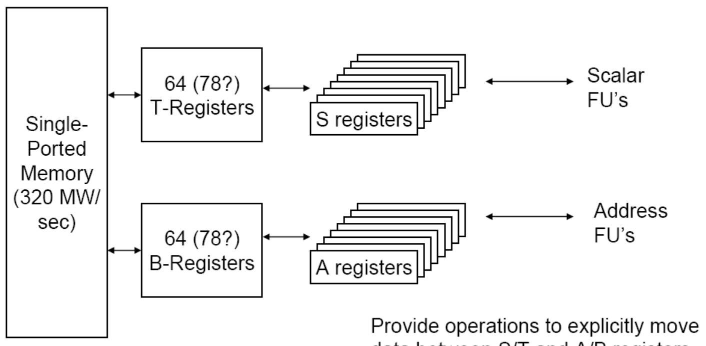
Memory Latency

- We would like to have small CPI (cycles per instruction)
- But it takes 100s of CPU cycles to access main memory
- How do we bridge the gap?
  - Put all data into registers?



Why Not A Lot of Registers?

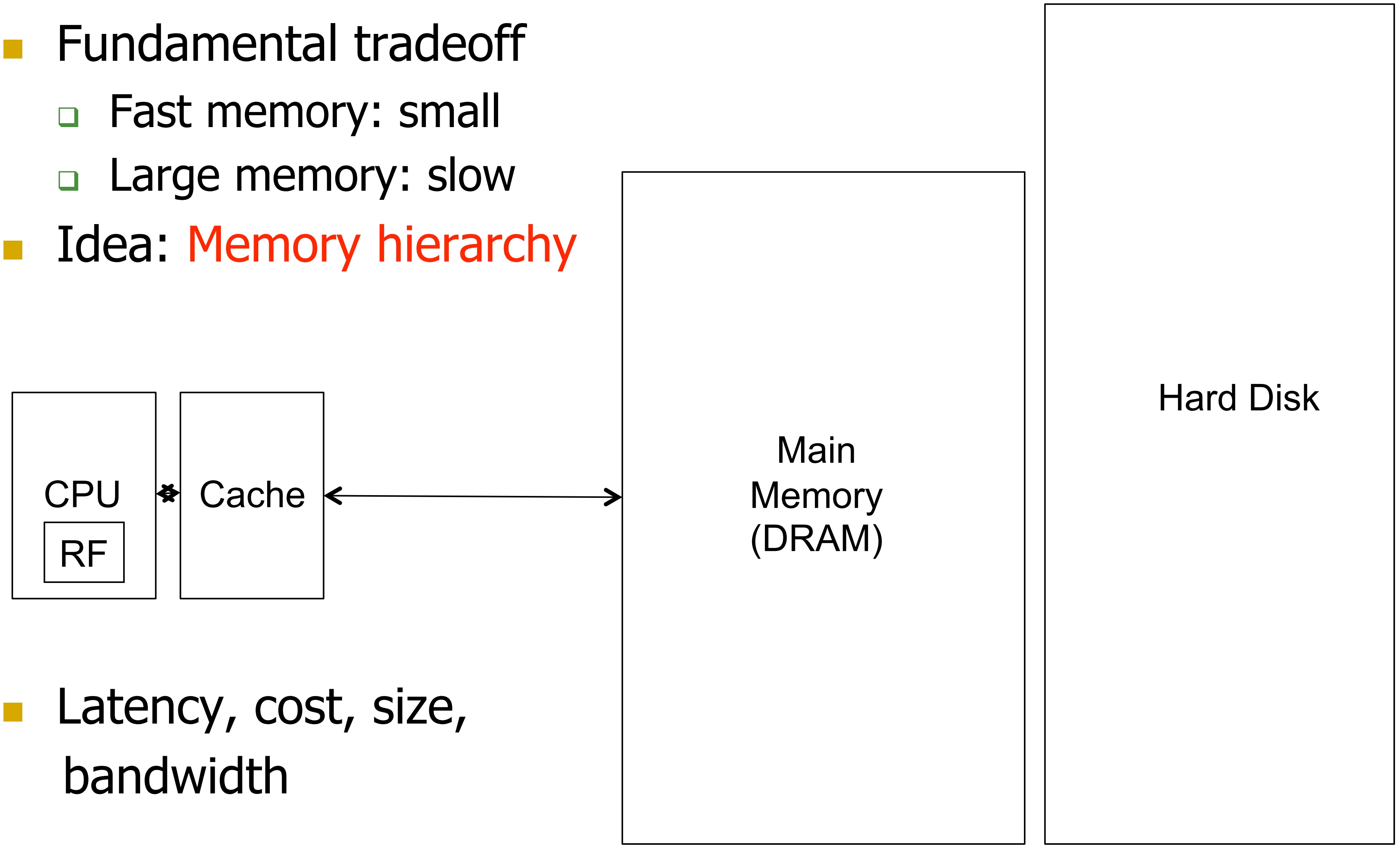
- Have a large number of architectural registers
  - Increases access time of register file
    - Cannot make memories large *and* fast
  - Increases bits used in instruction format for register ID
    - 1024 registers in a 3-address machine → 30 bits to specify IDs
- Multi-level register files
  - CRAY-1 had this
  - A small, fast register file connected to a large, slower file
  - Movement between files/memory explicitly managed by code
  - Explicit management not simple
    - Not easy to figure out which data is frequently used
  - Cache: automatic management of data



Provide operations to explicitly move data between S/T and A/B registers

Memory Hierarchy

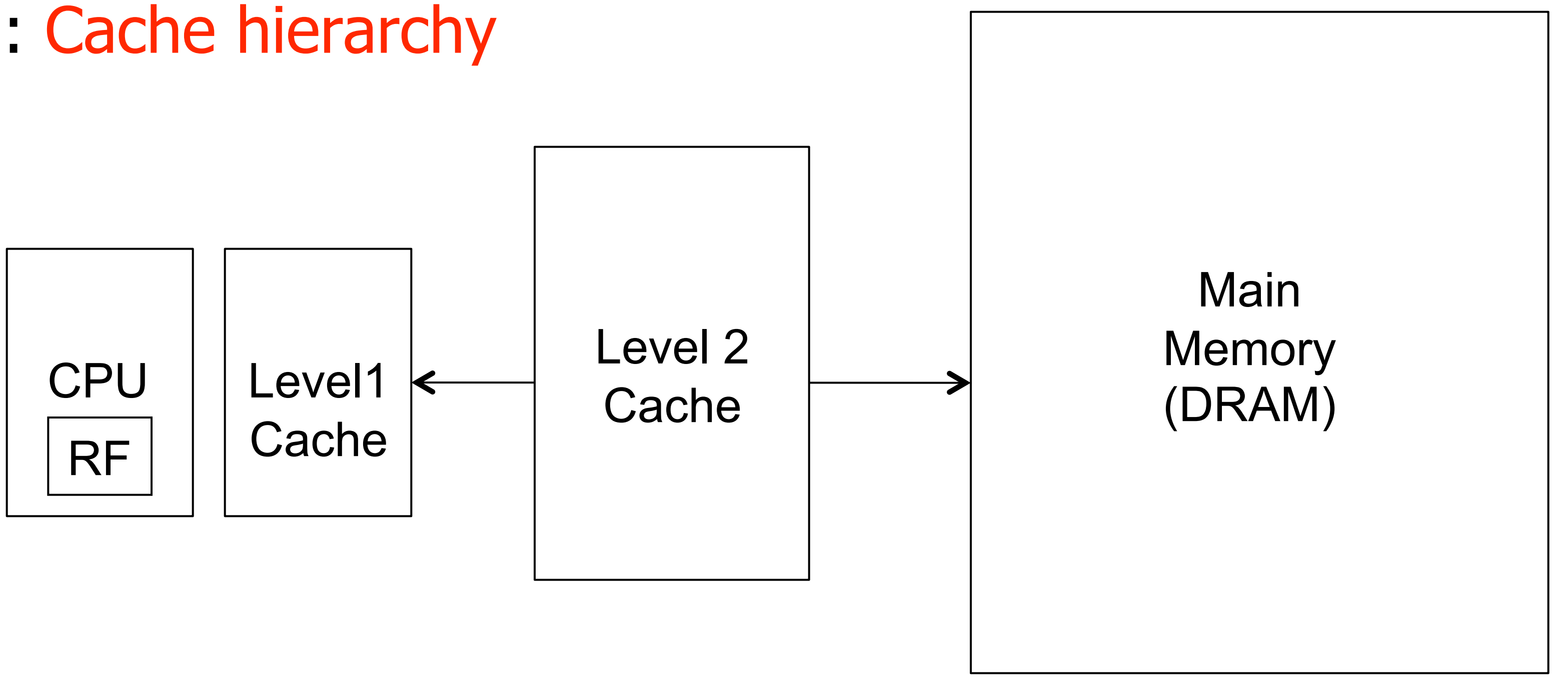
- Fundamental tradeoff
    - Fast memory: small
    - Large memory: slow
- Idea: Memory hierarchy



- Latency, cost, size, bandwidth

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy



Caching Basics: Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

- Temporal locality principle
    - Recently accessed data will be again accessed in the near future
    - This is what Maurice Wilkes had in mind:
        - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
        - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Caching Basics: Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.

The Bookshelf Analogy

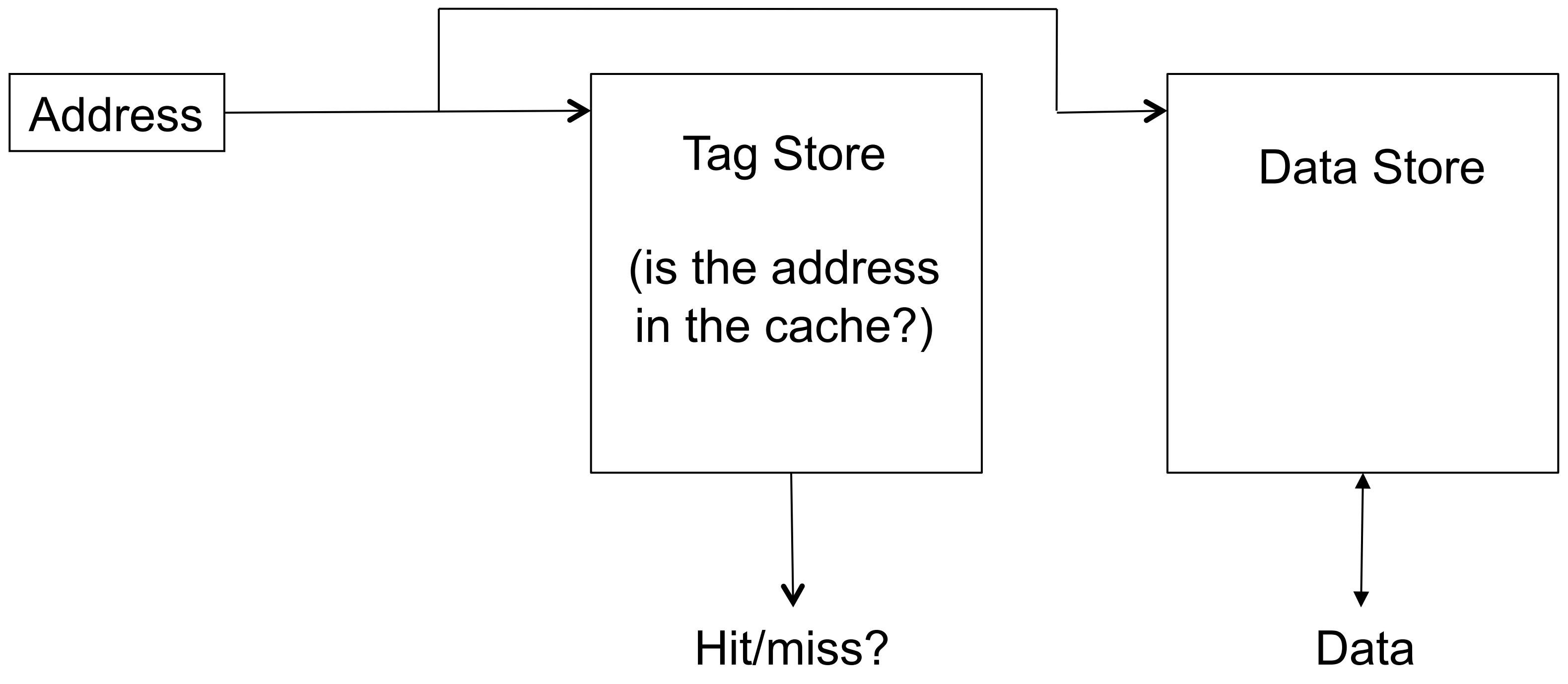
- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

Caching Basics

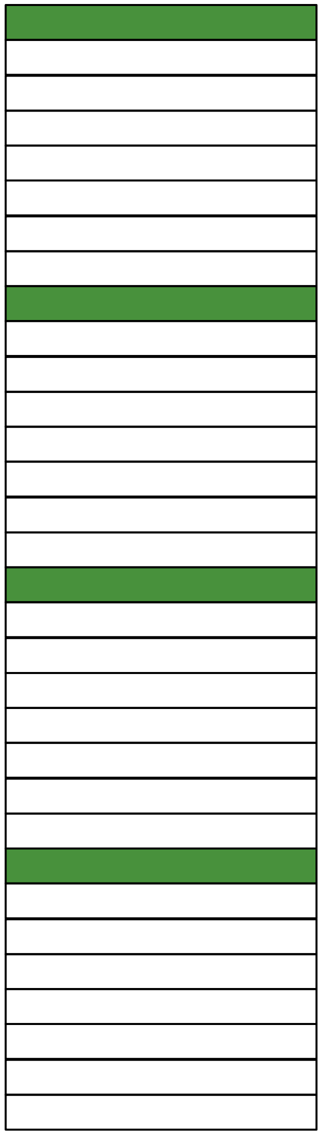
- When data referenced
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring into cache
    - Maybe have to kick something else out to do it
- Important decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Write policy: what do we do about writes?
  - Instructions/data
- Block (line): Unit of storage in the cache
  - IBM vs. DEC terminology

Cache Abstraction and Metrics

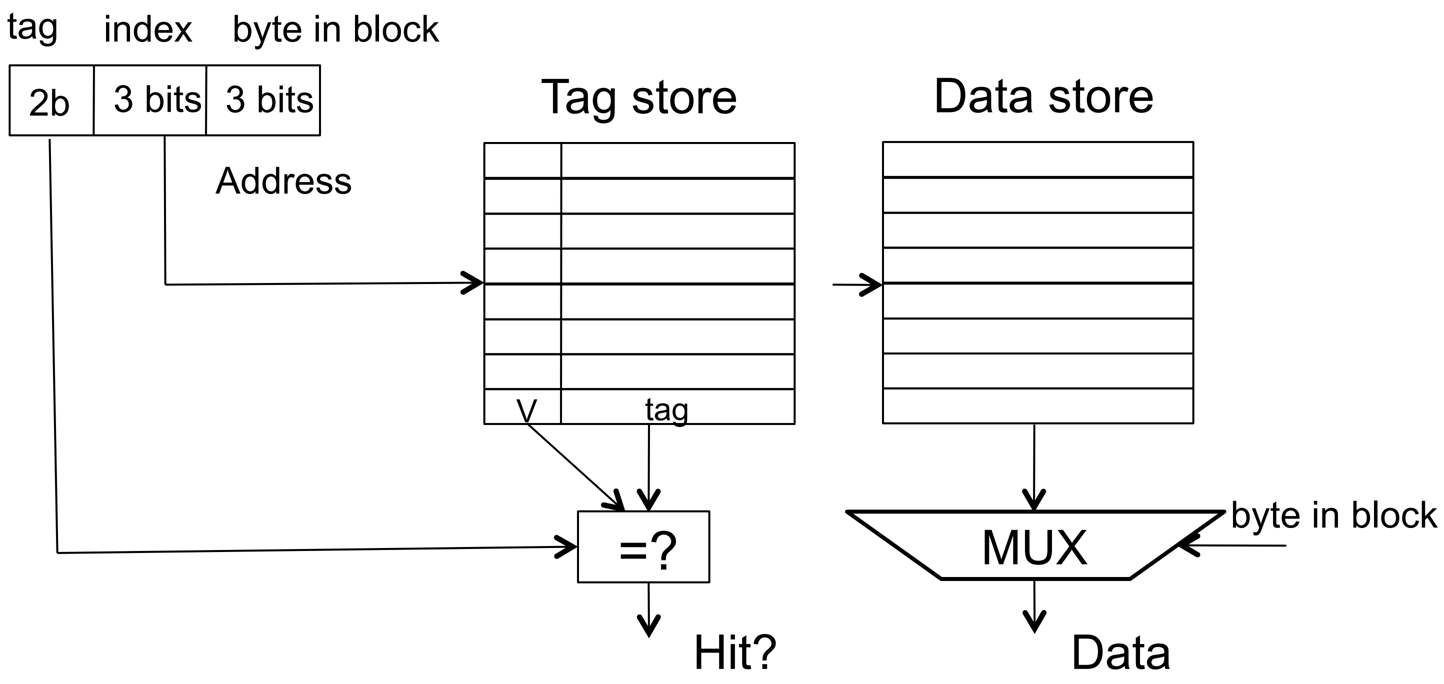


- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: *Can reducing AMAT reduce performance?*

Placement and Access



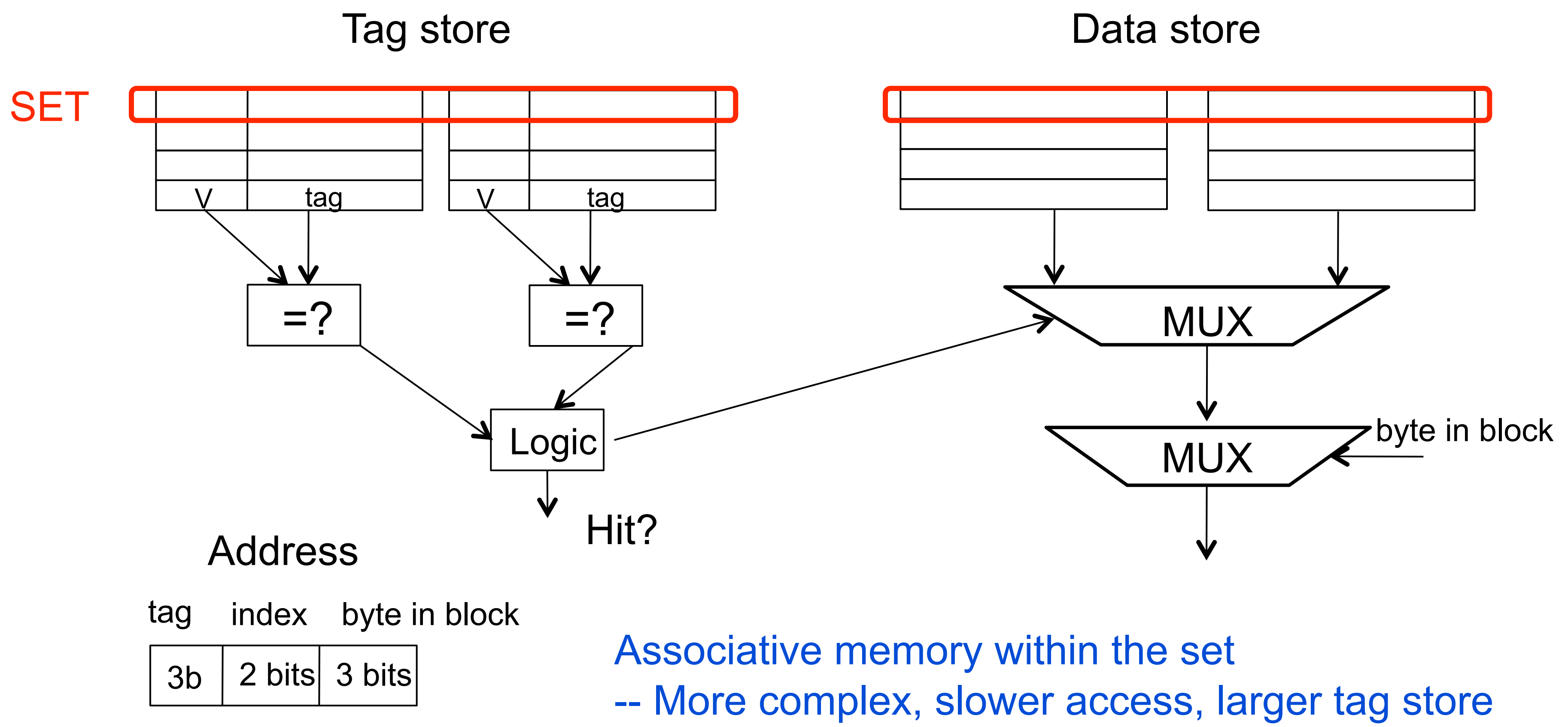
- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can only go to one location



  - Addresses with same index contend for the same location
    - Cause conflict misses

Set Associativity

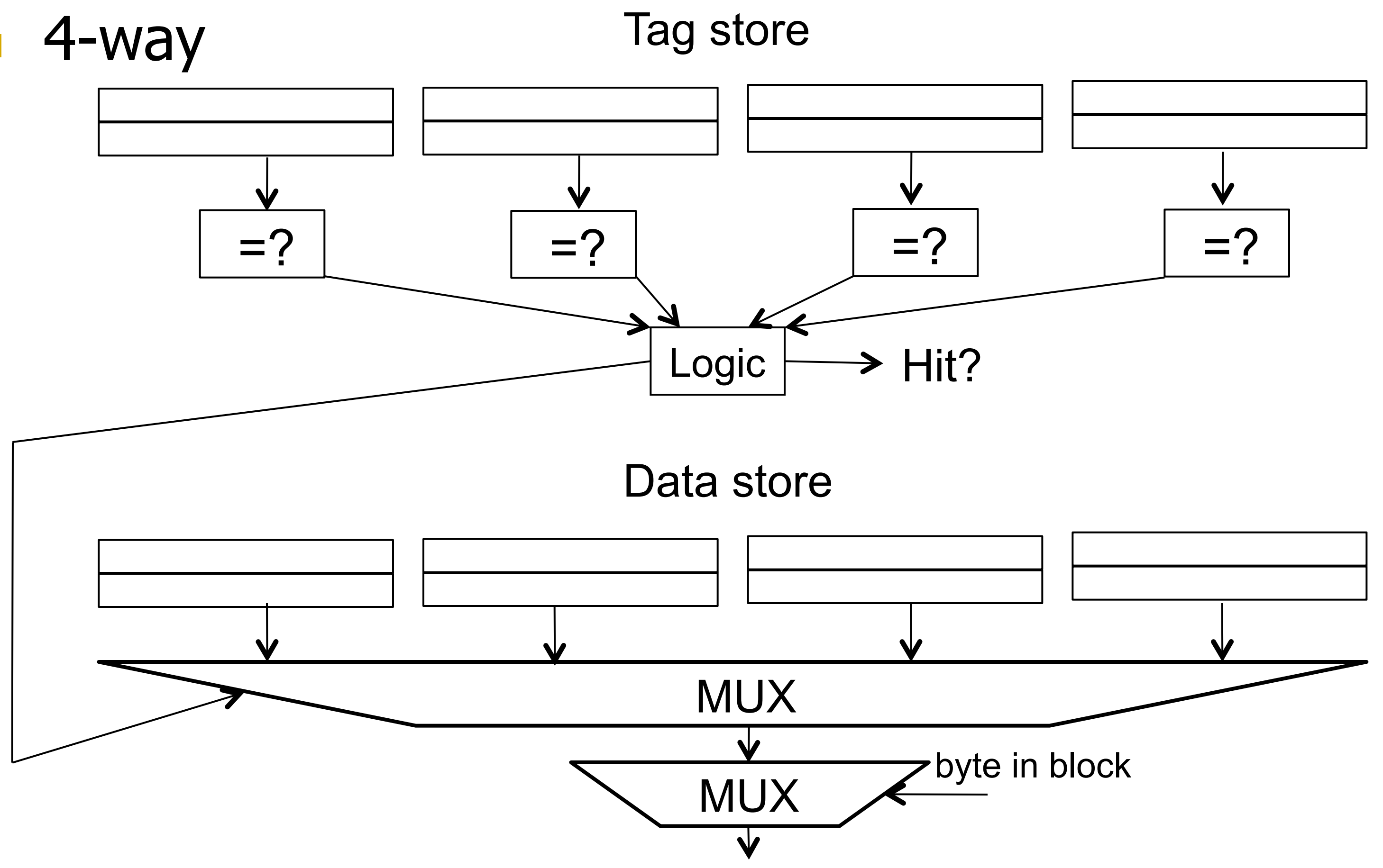
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Associative memory within the set
-- More complex, slower access, larger tag store
+ Accommodates conflicts better (fewer conflict misses)

Higher Associativity

- 4-way

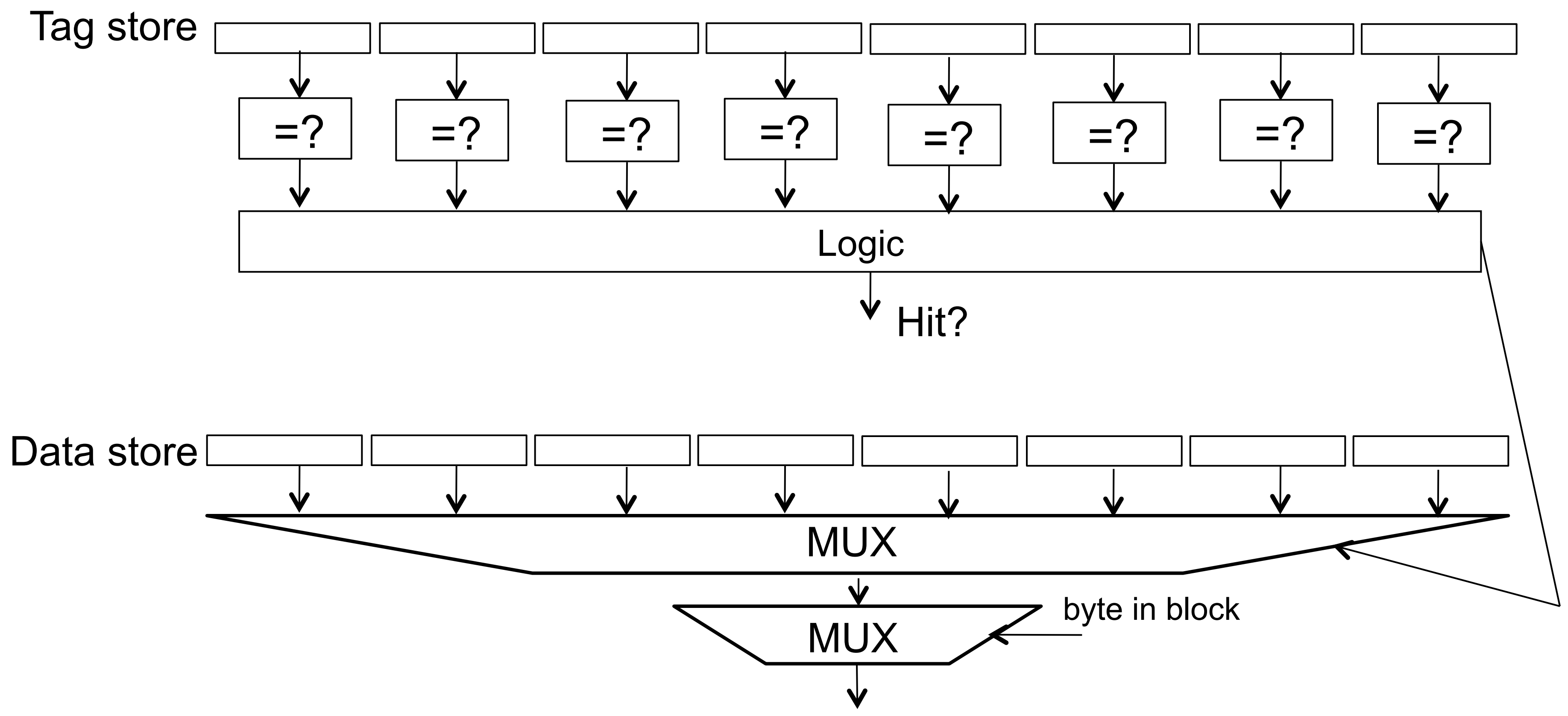


-- More tag comparators and wider data mux; larger tags

+ Likelihood of conflict misses even lower

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location

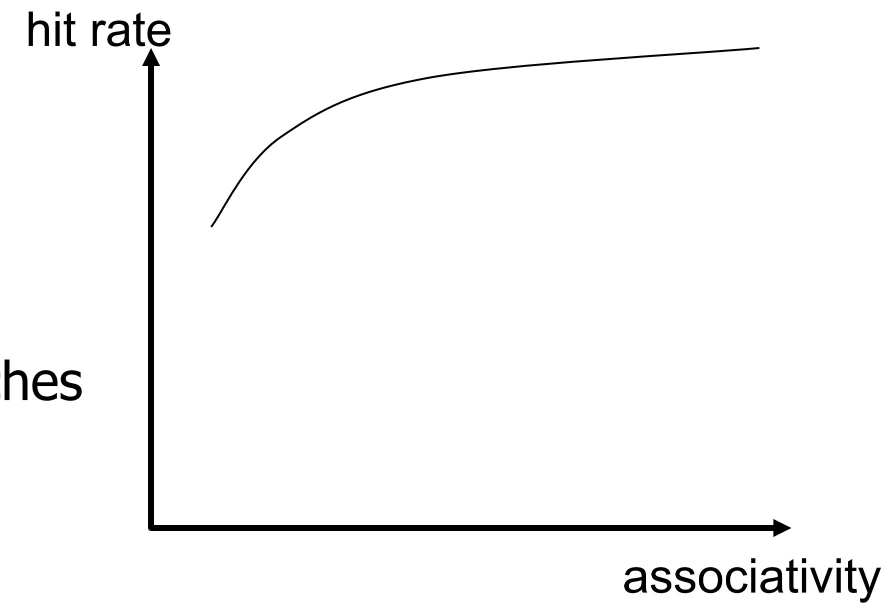


Associativity

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns

- Smaller associativity
  - lower cost
  - faster hit time
    - Especially important for L1 caches



Set-Associative Caches (I)

- Diminishing returns in hit rate from higher associativity
- Longer access time with higher associativity
- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - **Least costly to re-fetch?**
      - **Why would memory accesses have different cost?**
    - Hybrid replacement policies
    - Optimal replacement policy?

Replacement Policy

- LRU vs. Random
  - Set thrashing: When the "program working set" in a set is larger than set associativity
  - 4-way: Cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

- Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Set-Associative Caches (II)

- Belady’s OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

  - Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. “A Case for MLP-Aware Cache Replacement,“ ISCA 2006.

Handling Writes (Stores)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back

  -- Need a bit in the tag store indicating the block is "modified"

  + Can consolidate multiple writes to the same block before eviction

  - Potentially saves bandwidth between cache levels + saves energy

- Write-through

  + Simpler

  + All levels are up to date. Consistency: Simpler cache coherence because no need to check lower-level caches

  -- More bandwidth intensive

Handling Writes (Stores)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

  + Can consolidate writes instead of writing each of them individually to next level

  -- Requires (?) transfer of the whole cache block

  + Simpler because write misses can be treated the same way as read misses

- No-allocate

  + Conserves cache space if locality of writes is low

Inclusion vs. Exclusion

- Inclusive caches
  - Every block existing in the first level also exists in the next level
  - When fetching a block, place it in all cache levels. Tradeoffs:

    -- Leads to duplication of data in the hierarchy: less efficient

    -- Maintaining inclusion takes effort (forced evictions)

    + But makes cache coherence in multiprocessors easier
    - Need to track other processors' accesses only in the highest-level cache
- Exclusive caches
  - The blocks contained in cache levels are mutually exclusive
  - When evicting a block, do you write it back to the next level?

  + More efficient utilization of cache space

  + (Potentially) More flexibility in replacement/placement

  -- More blocks/levels to keep track of to ensure cache coherence; takes effort
- Non-inclusive caches
  - No guarantees for inclusion or exclusion: simpler design
  - Most Intel processors

Maintaining Inclusion and Exclusion

- When does maintaining inclusion take effort?
  - L1 block size < L2 block size
  - L1 associativity > L2 associativity
  - Prefetching into L2
  - When a block is evicted from L2, need to evict all corresponding subblocks from L1 → keep 1 bit per subblock in L2
  - When a block is inserted, make sure all higher levels also have it

- When does maintaining exclusion take effort?
  - L1 block size != L2 block size
  - Prefetching into any cache level
  - When a block is inserted into any level, ensure it is not in any other