Caches

Samira Khan

March 23, 2017

Agenda

- Review from last lecture
  - Data flow model
  - Memory hierarchy
- More Caches

The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the instruction pointer
  - Sequential unless explicit control flow instruction

- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Memory locality is not exploited

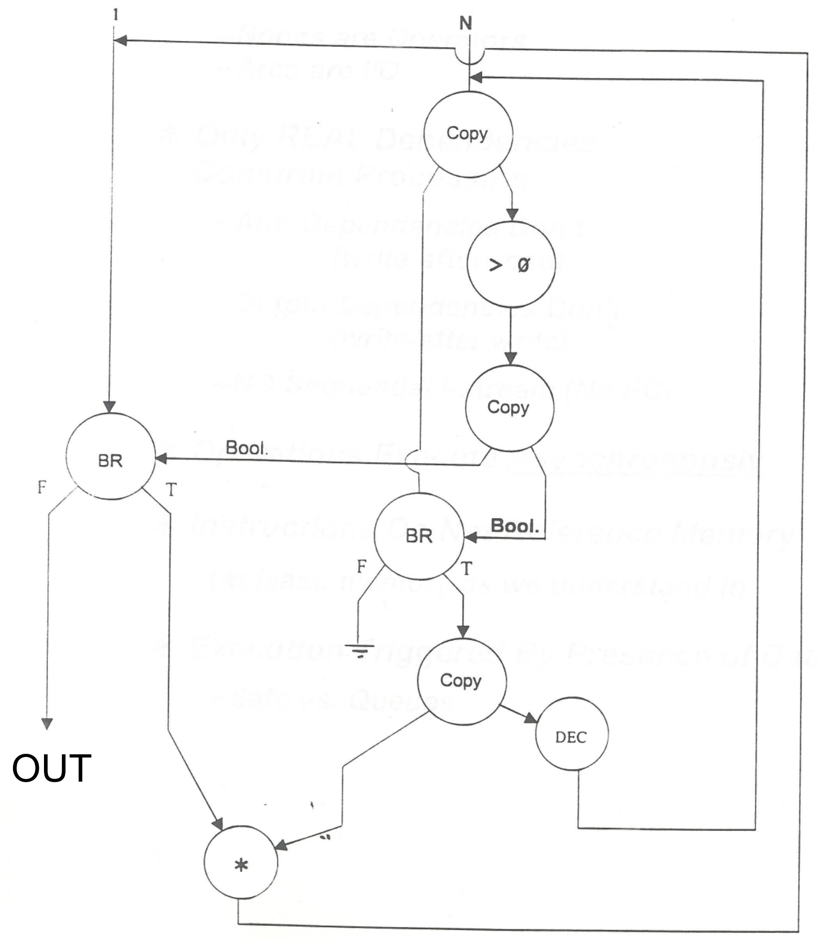
OOO EXECUTION: RESTRICTED DATAFLOW

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?

- The dataflow graph is limited to the instruction window
  - Instruction window: all decoded but not yet retired instructions

- Can we do it for the whole program?

An Example



The Memory Hierarchy

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

Why Memory Hierarchy?

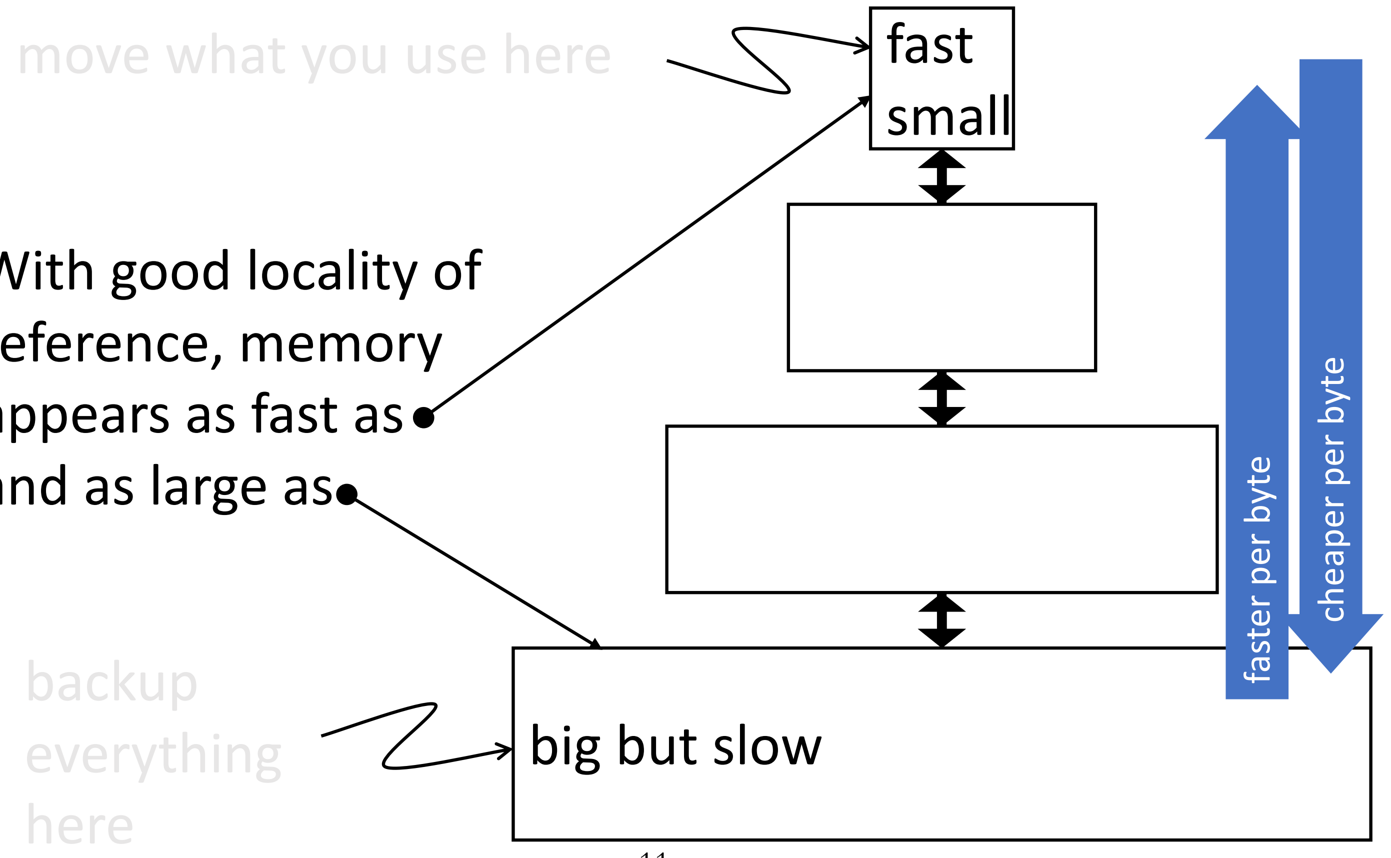
- We want both fast and large

- But we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

The Memory Hierarchy

With good locality of reference, memory appears as fast as ● and as large as ●



Memory Locality

- A “typical” program has a lot of locality in memory references
  - typical programs are composed of “loops”

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - instruction memory references
    - array/data structure references

Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of $t_{i,}$ The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

- Miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable

- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter management (replacement::anticipate what you don’t need, prefetching::anticipate what you will need)

- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

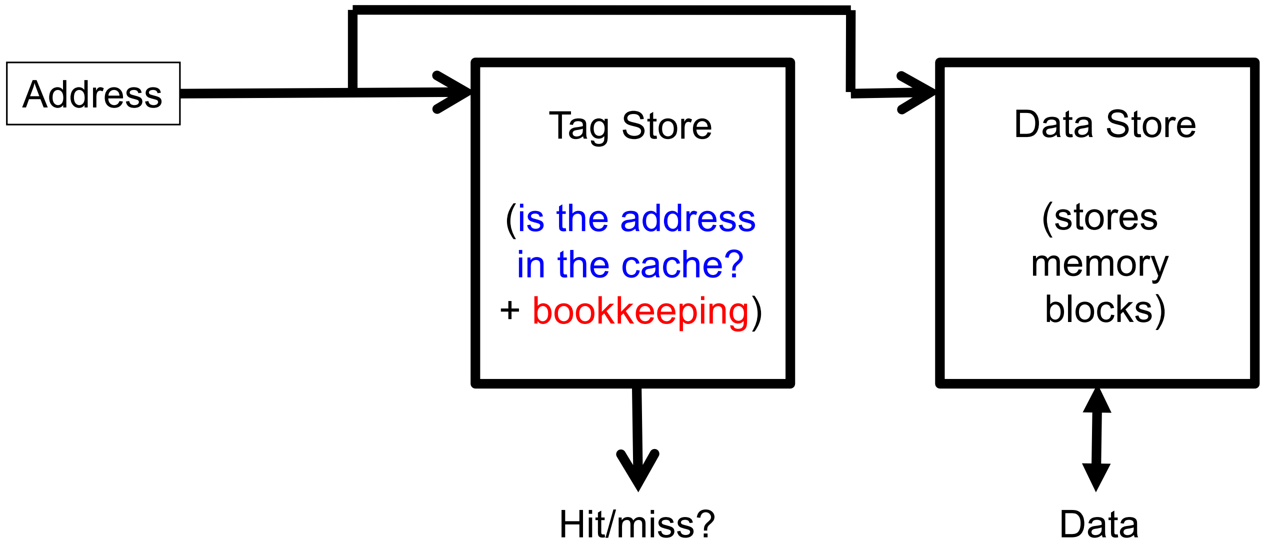
if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
- When data referenced
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large, small, uniform blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: Do we treat them separately?

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?*

A Basic Hardware Cache Design

- We will start with a basic hardware cache design

- Then, we will examine a multitude of ideas to make it better

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks

- Each block maps to a location in the cache, determined by the index bits in the address
  - used to index into the tag and data stores

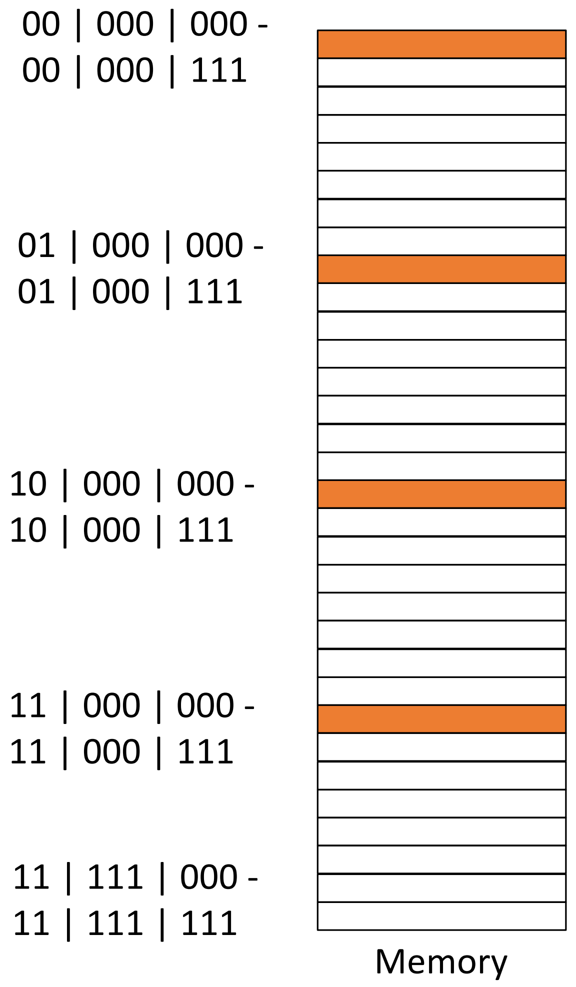
tag	index	byte in block
2 bits	3 bits	3 bits

8-bit address

- Cache access:
  - 1) index into the tag and data stores with index bits in address
  - 2) check valid bit in tag store
  - 3) compare tag bits in address with the stored tag in tag store

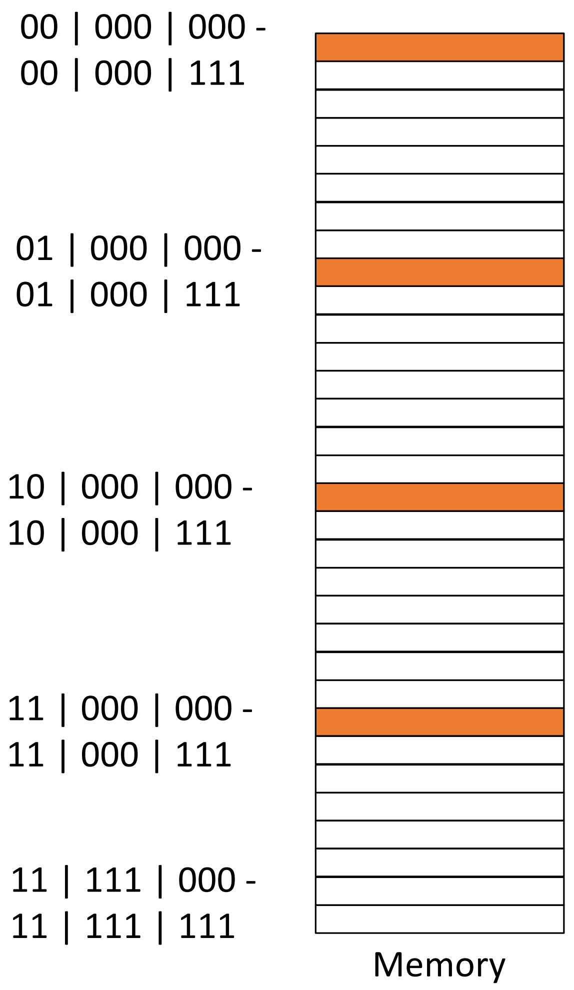
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache: Placement and Access

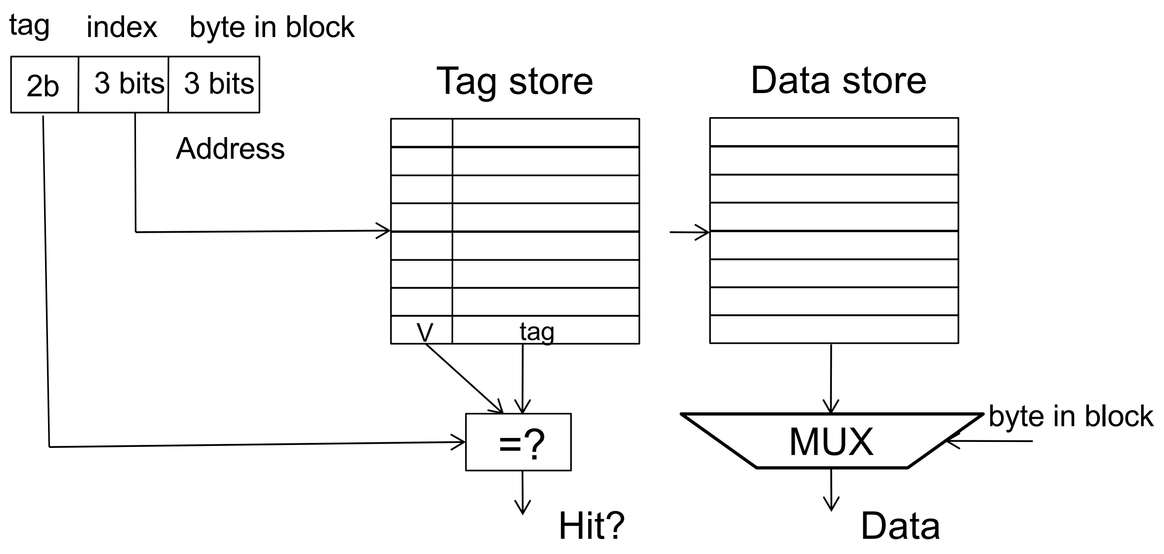


- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks

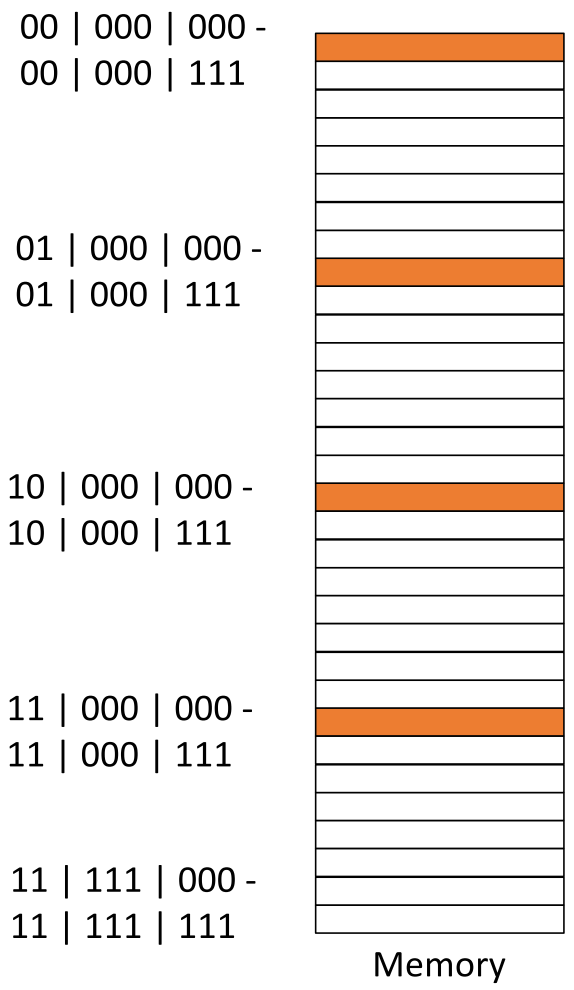
Direct-Mapped Cache: Placement and Access



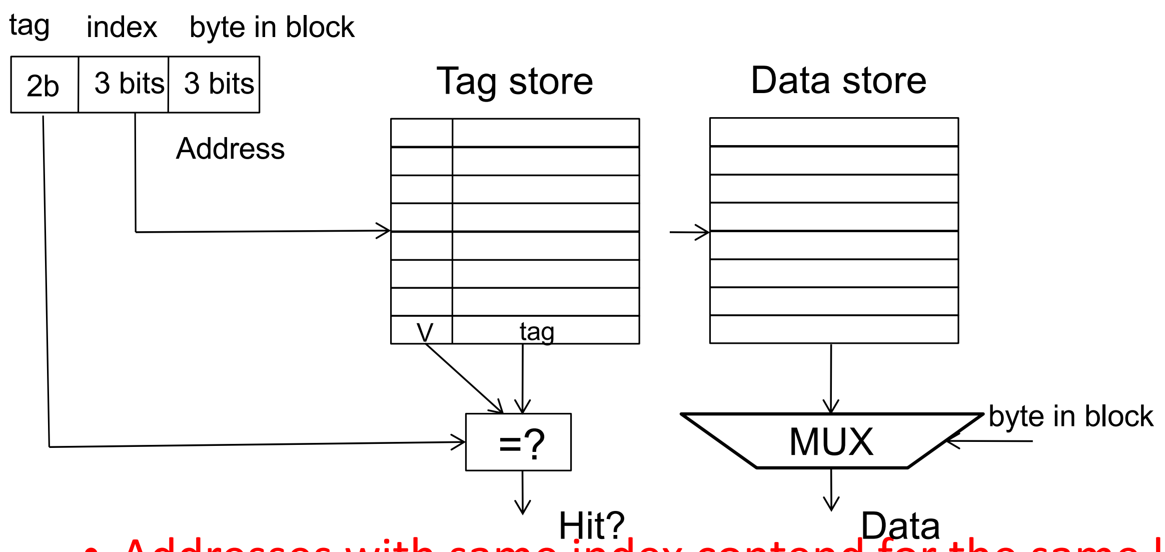
- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location



Direct-Mapped Cache: 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 go to only one location



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

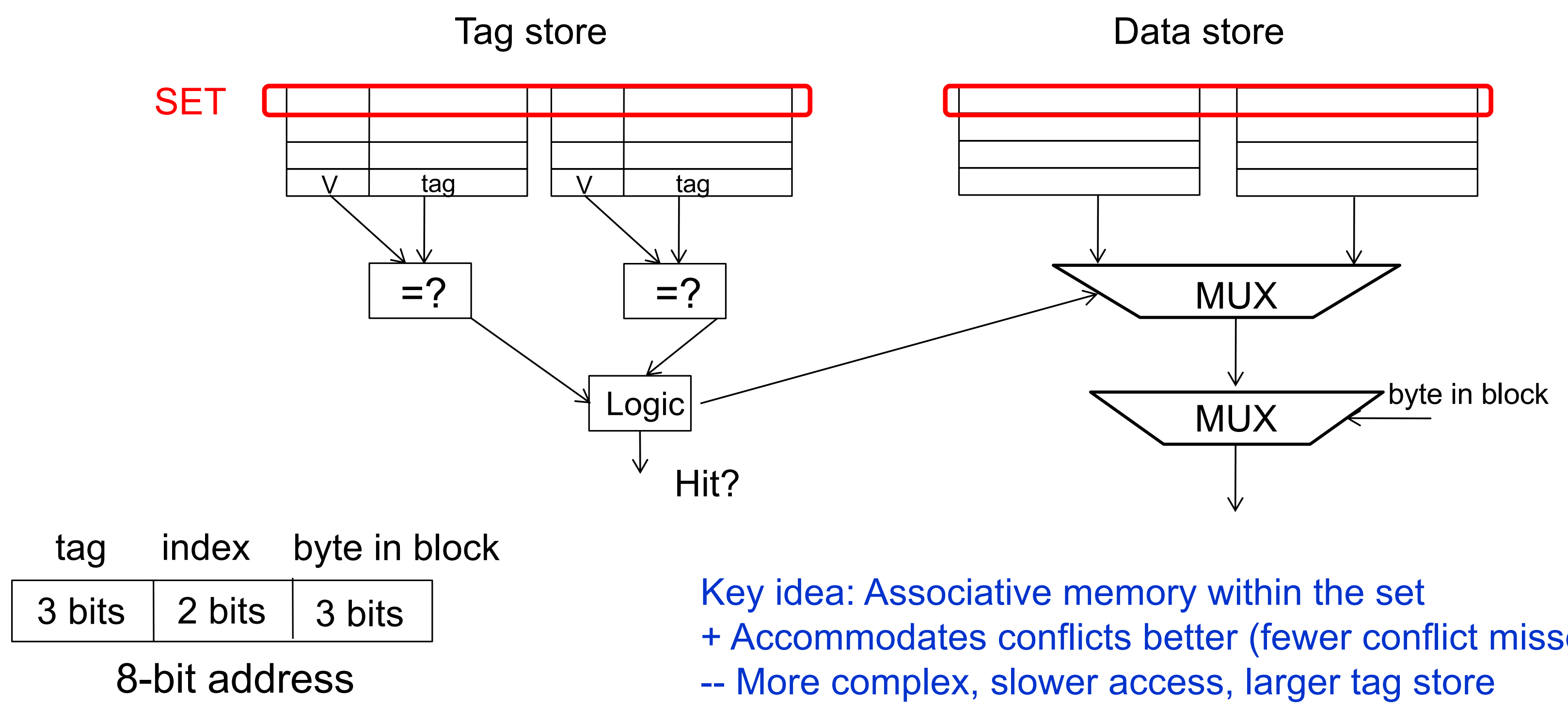
Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

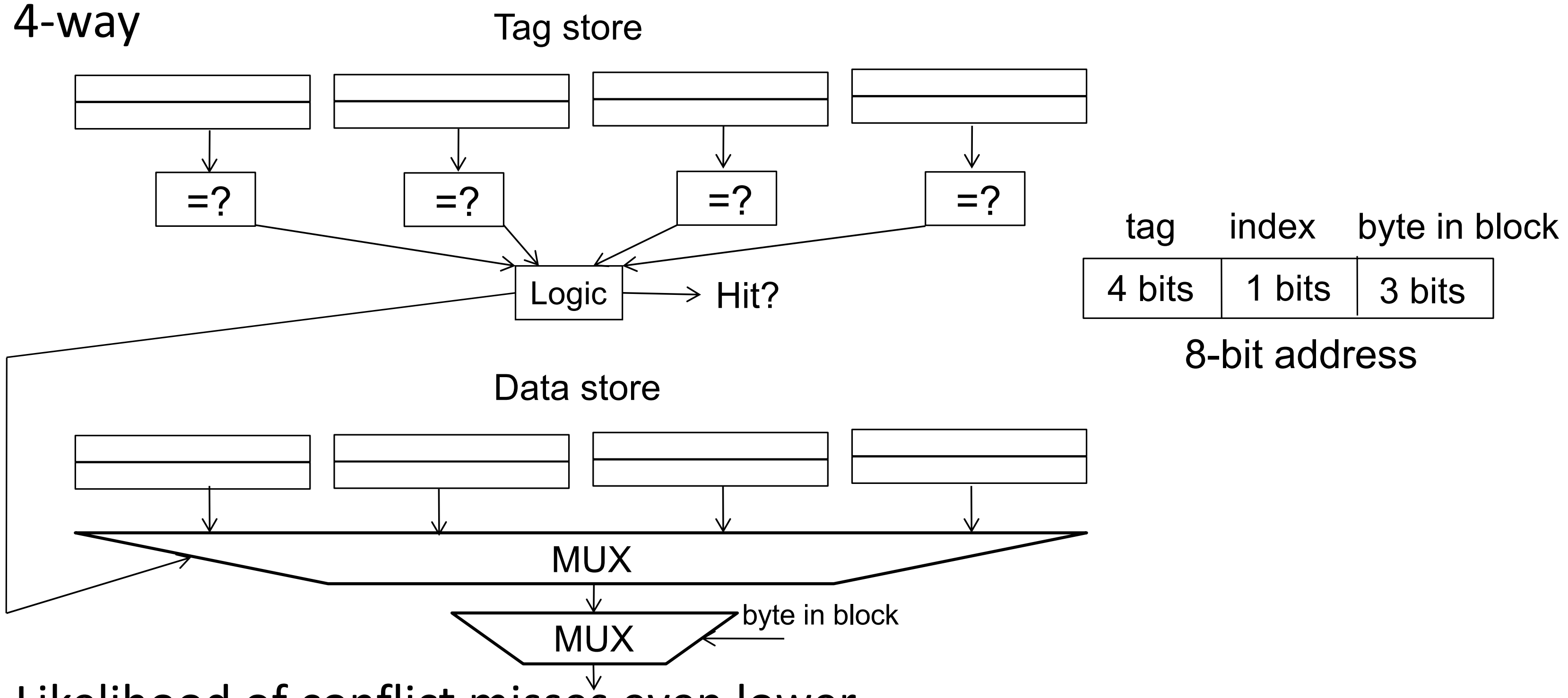
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



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

Higher Associativity

- 4-way

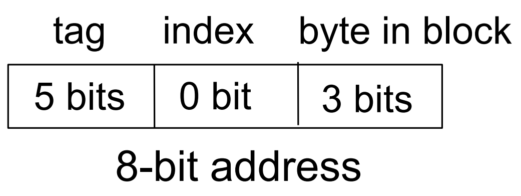


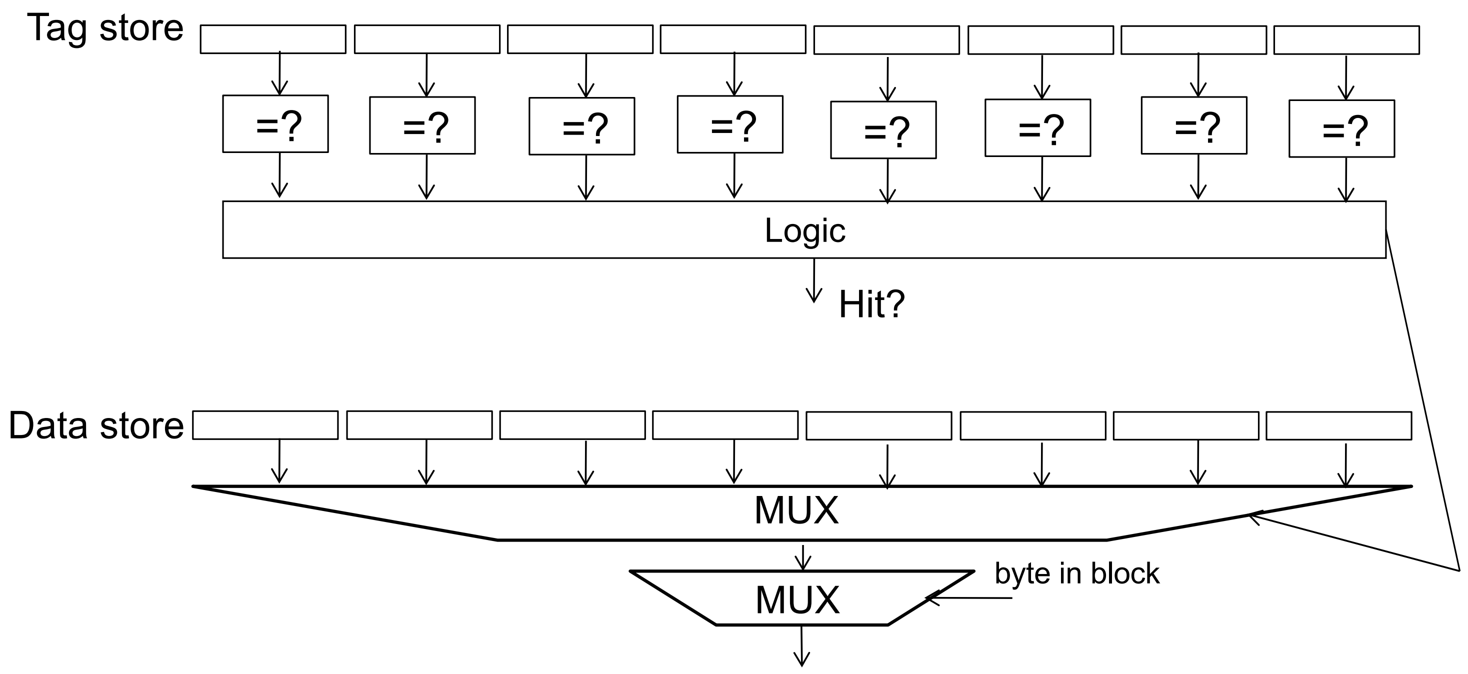
+ Likelihood of conflict misses even lower

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

Full Associativity

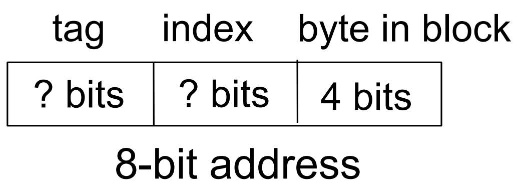
- Fully associative cache
  - A block can be placed in any cache location



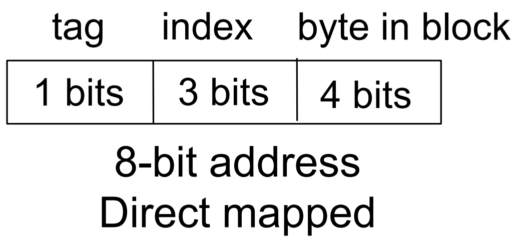


Exercise on Cache Indexing

- We assumed 8 byte blocks
- What happens if we have 16 byte blocks?



- Cache is 128B, 8 blocks
- Direct mapped
- 2-way?
- 4-way?
- 8-way?



Tag-Index-Offset

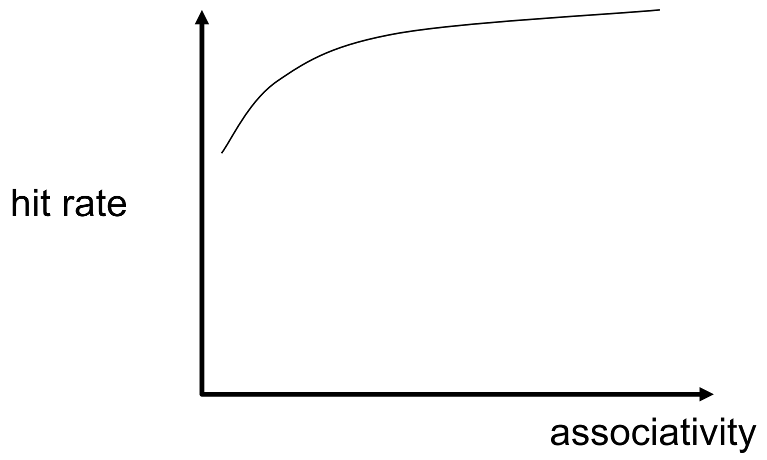
- m memory address bits
- $S = 2^s$ number of sets
- s (set) index bits
- $B = 2^b$ block size
- b (block) offset bits
- $t = m - (s + b)$ tag bits
- $C = B * S$ cache size (if direct-mapped)

Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

- Higher associativity
  ++ Higher hit rate
  -- Slower cache access time (hit latency and data access latency)
  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



Issues in Set-Associative Caches

- Think of each block in a set having a “priority”
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

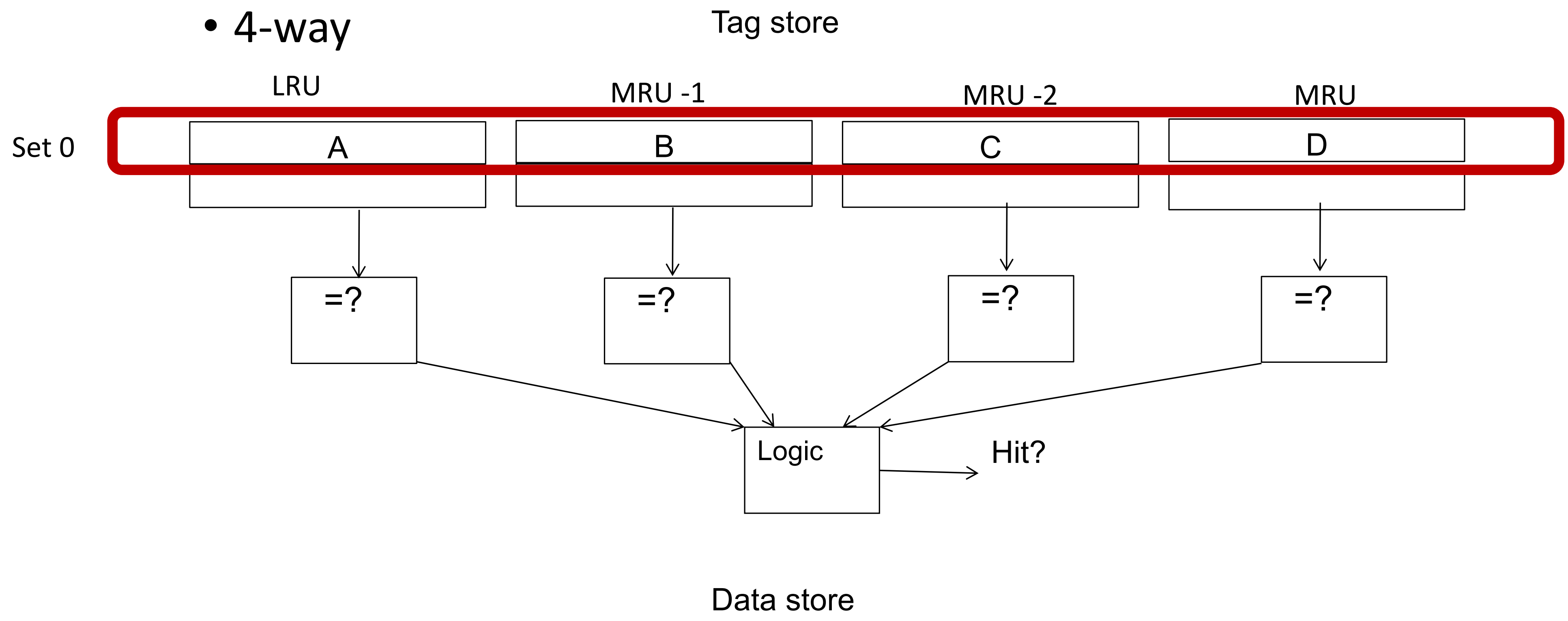
- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- 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
    - Hybrid replacement policies
    - Optimal replacement policy?

Least Recently Used Replacement Policy

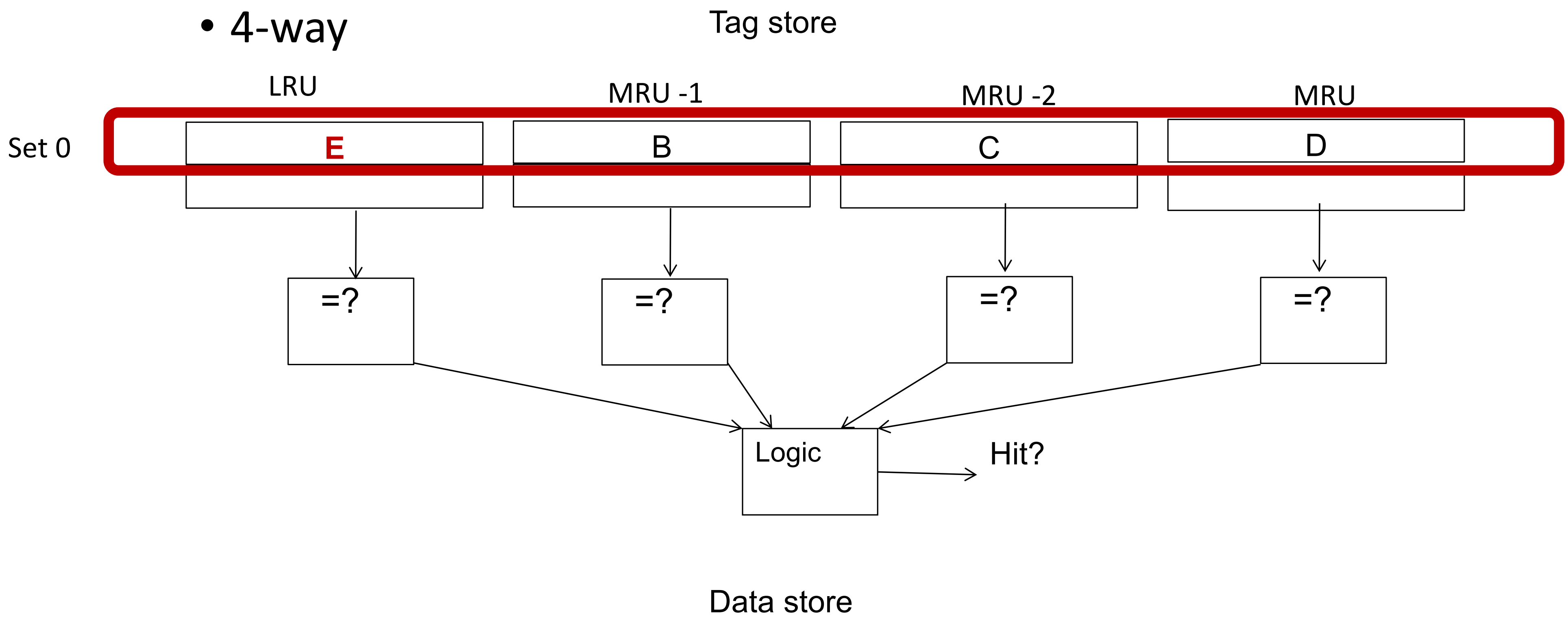
- 4-way



ACCESS PATTERN: ACBD

Least Recently Used Replacement Policy

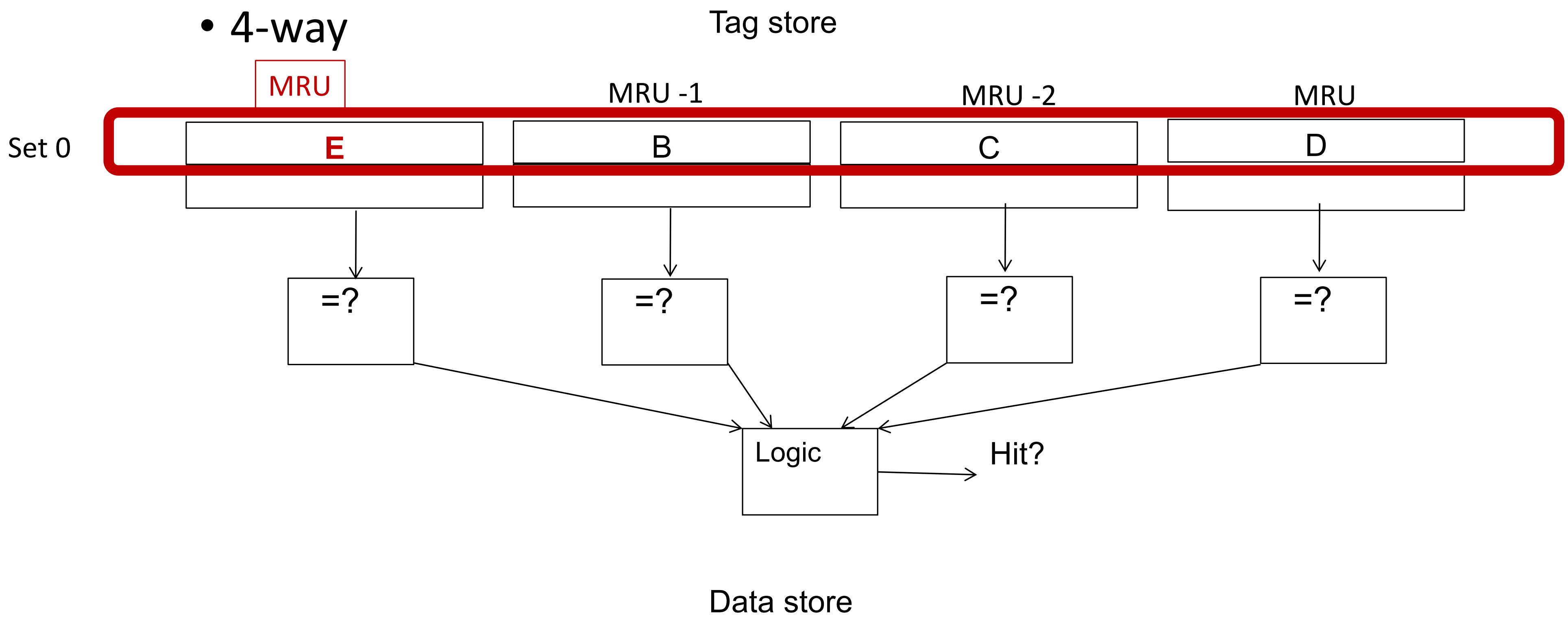
- 4-way



ACCESS PATTERN: ACBDE

Least Recently Used Replacement Policy

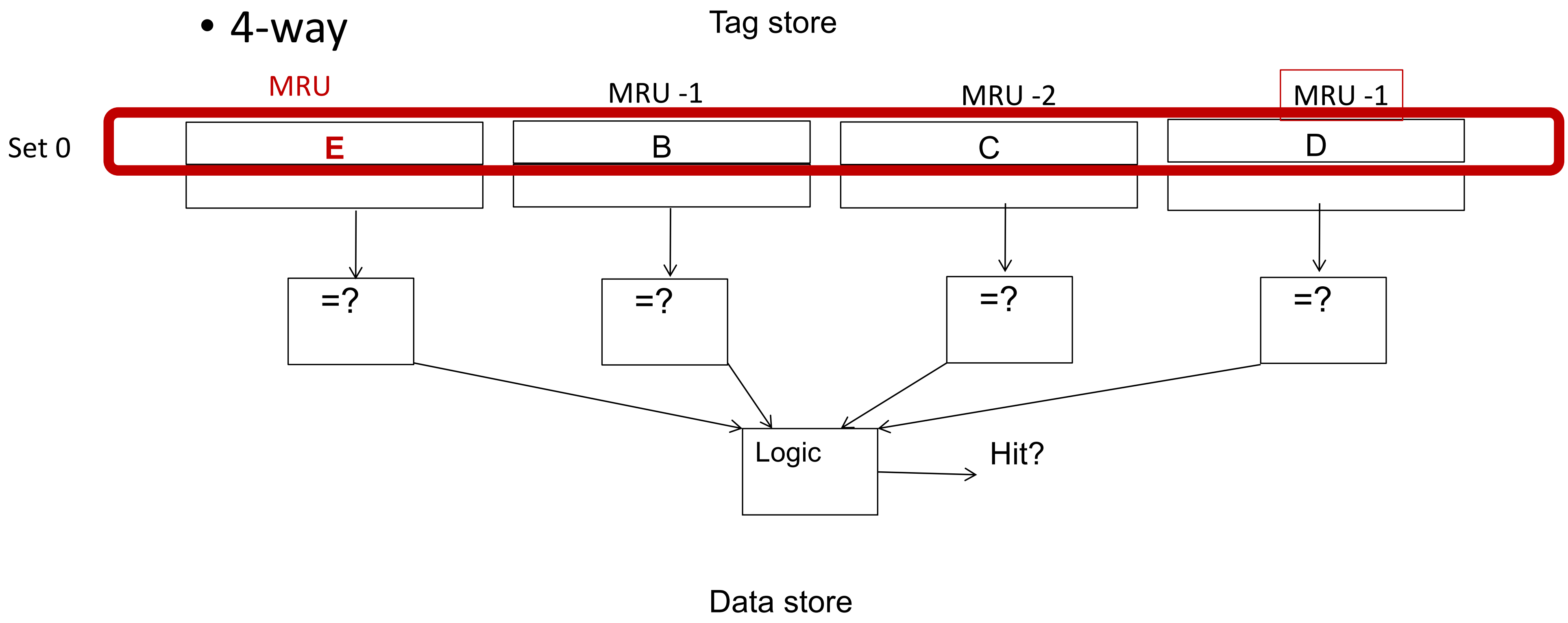
- 4-way



ACCESS PATTERN: ACBDE

Least Recently Used Replacement Policy

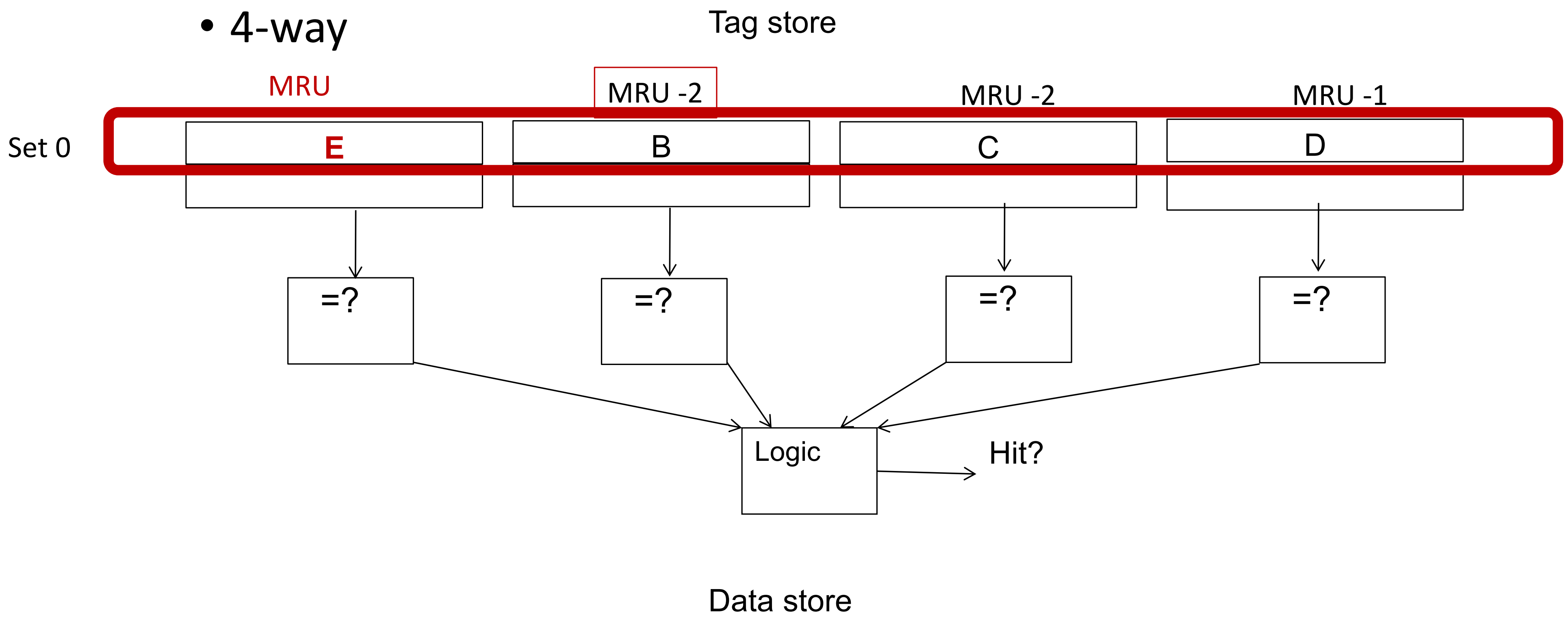
- 4-way



ACCESS PATTERN: ACBDE

Least Recently Used Replacement Policy

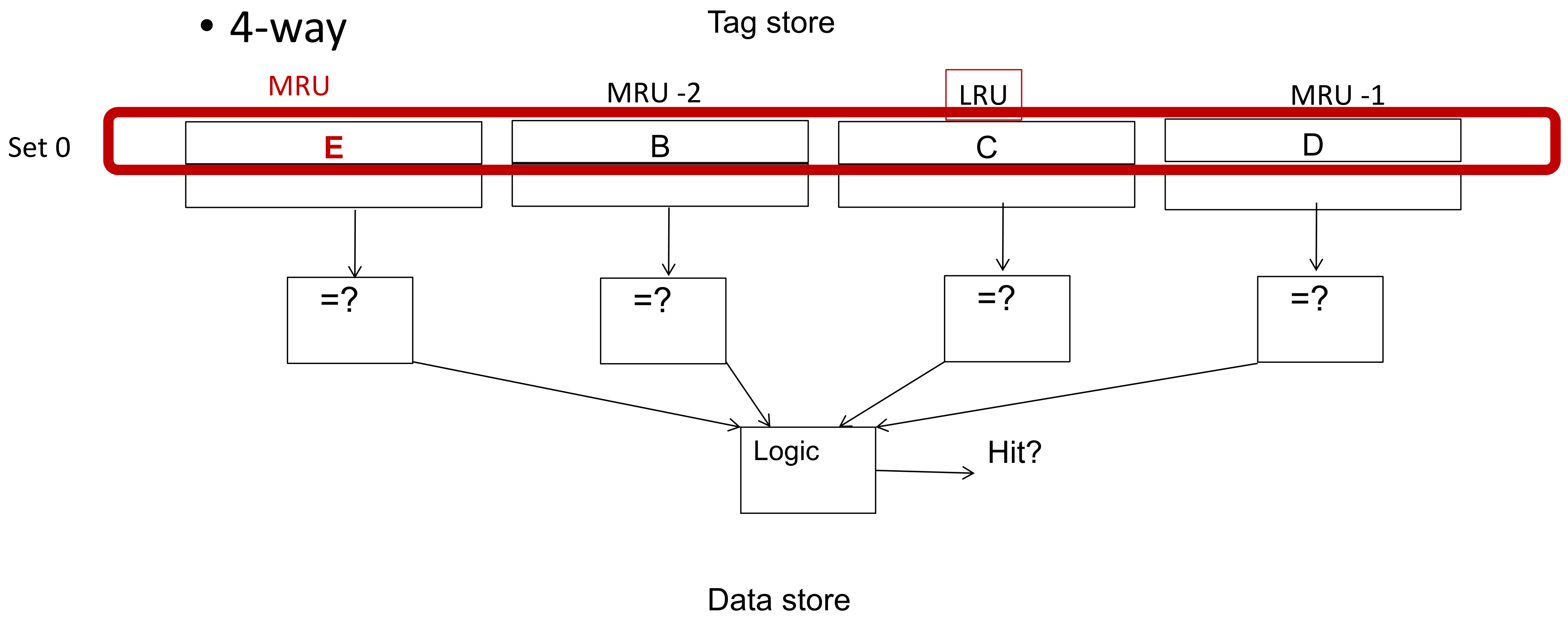
- 4-way



ACCESS PATTERN: ACBDE

Least Recently Used Replacement Policy

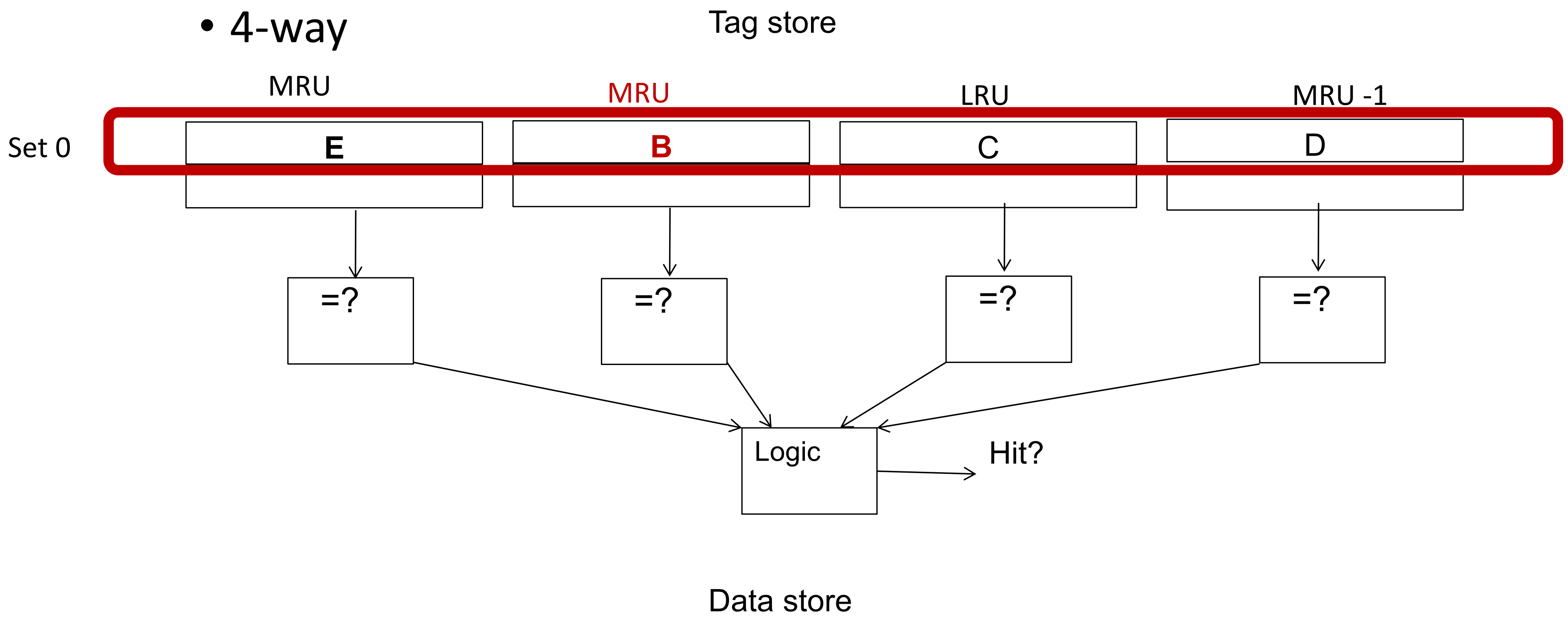
- 4-way



ACCESS PATTERN: ACBDE

Least Recently Used Replacement Policy

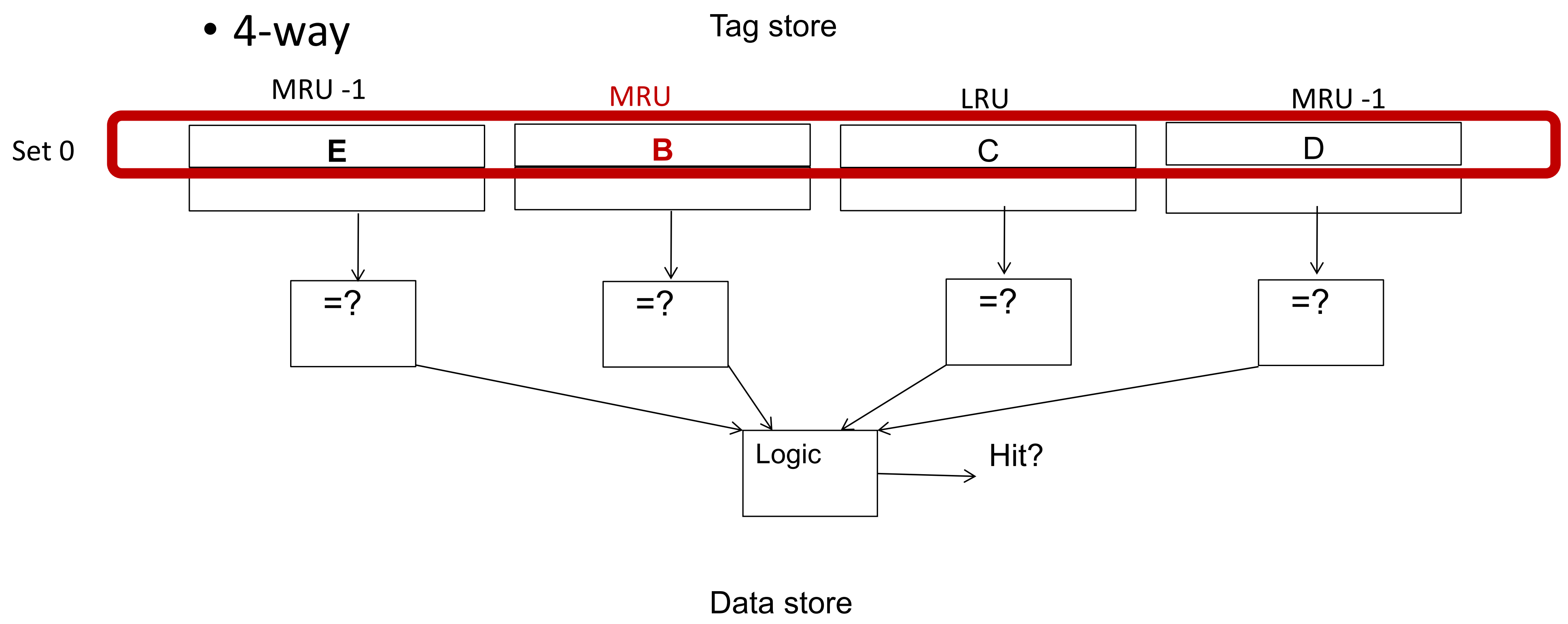
- 4-way



ACCESS PATTERN: ACBDEB

Least Recently Used Replacement Policy

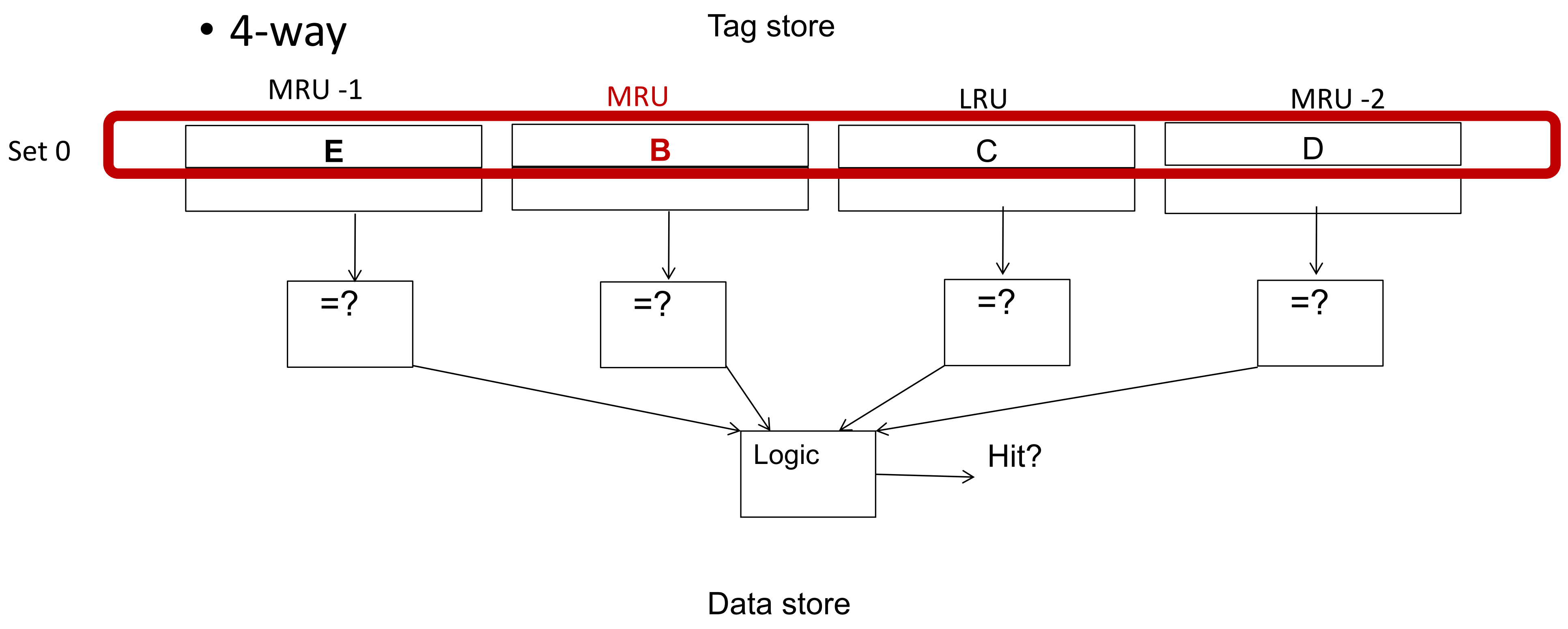
- 4-way



ACCESS PATTERN: ACBDEB

Least Recently Used Replacement Policy

- 4-way



ACCESS PATTERN: ACBDEB

Eviction/Replacement Policy

- 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
    - Hybrid replacement policies
    - Optimal replacement policy?