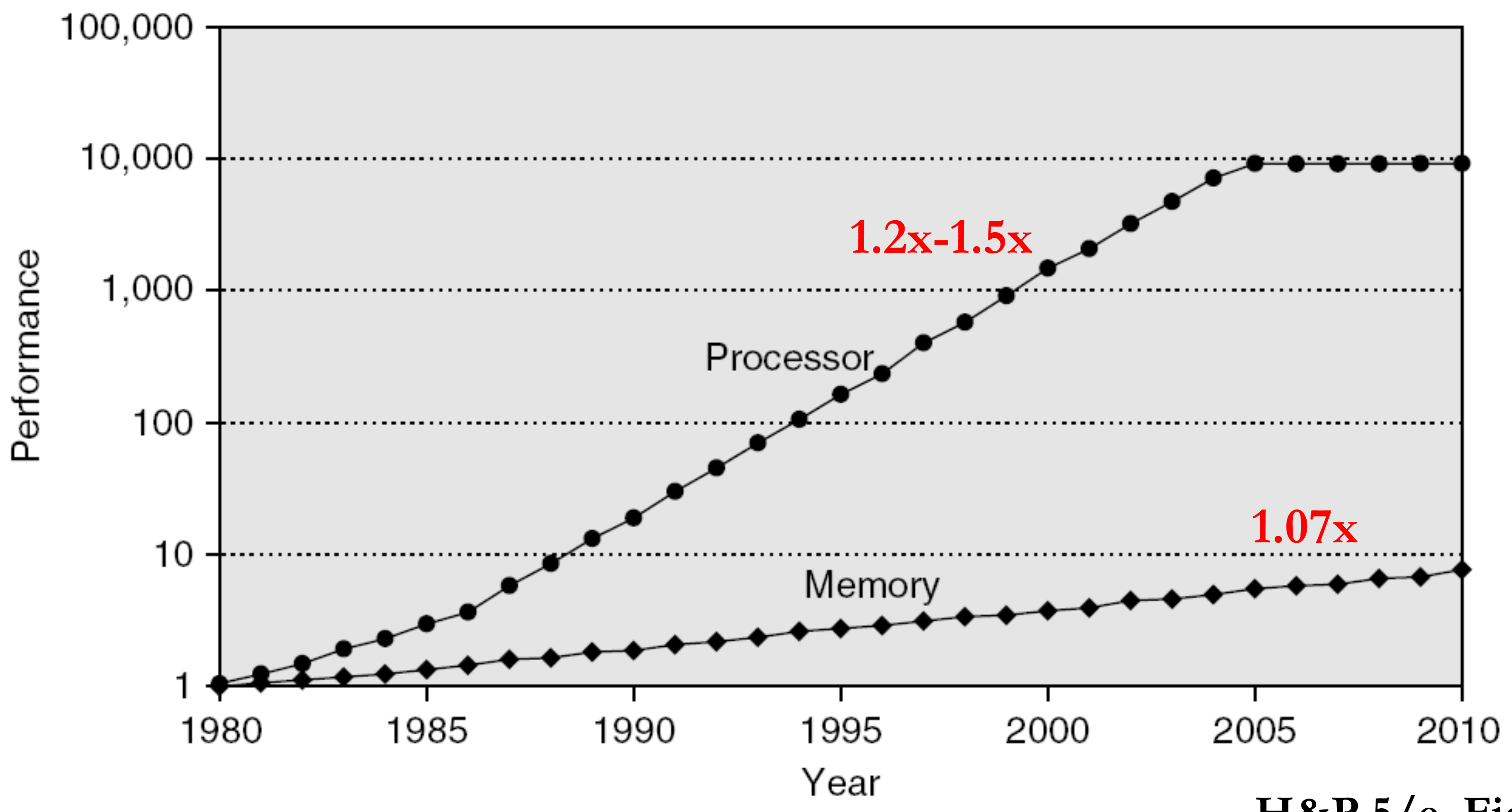
Announcements

- Assignment 1 due Mon (Feb 19) @ 4pm

- Next week: no classes

The Memory Gap



H&P 5/e, Fig. 2.2

Memory subsystem design increasingly important!

Idea of a memory hierarchy

- Use combination of memory kinds
  - Smaller amounts of expensive but fast memory closer to the processor
  - Larger amounts of cheaper but slower memory farther from the processor
- Idea is not new:

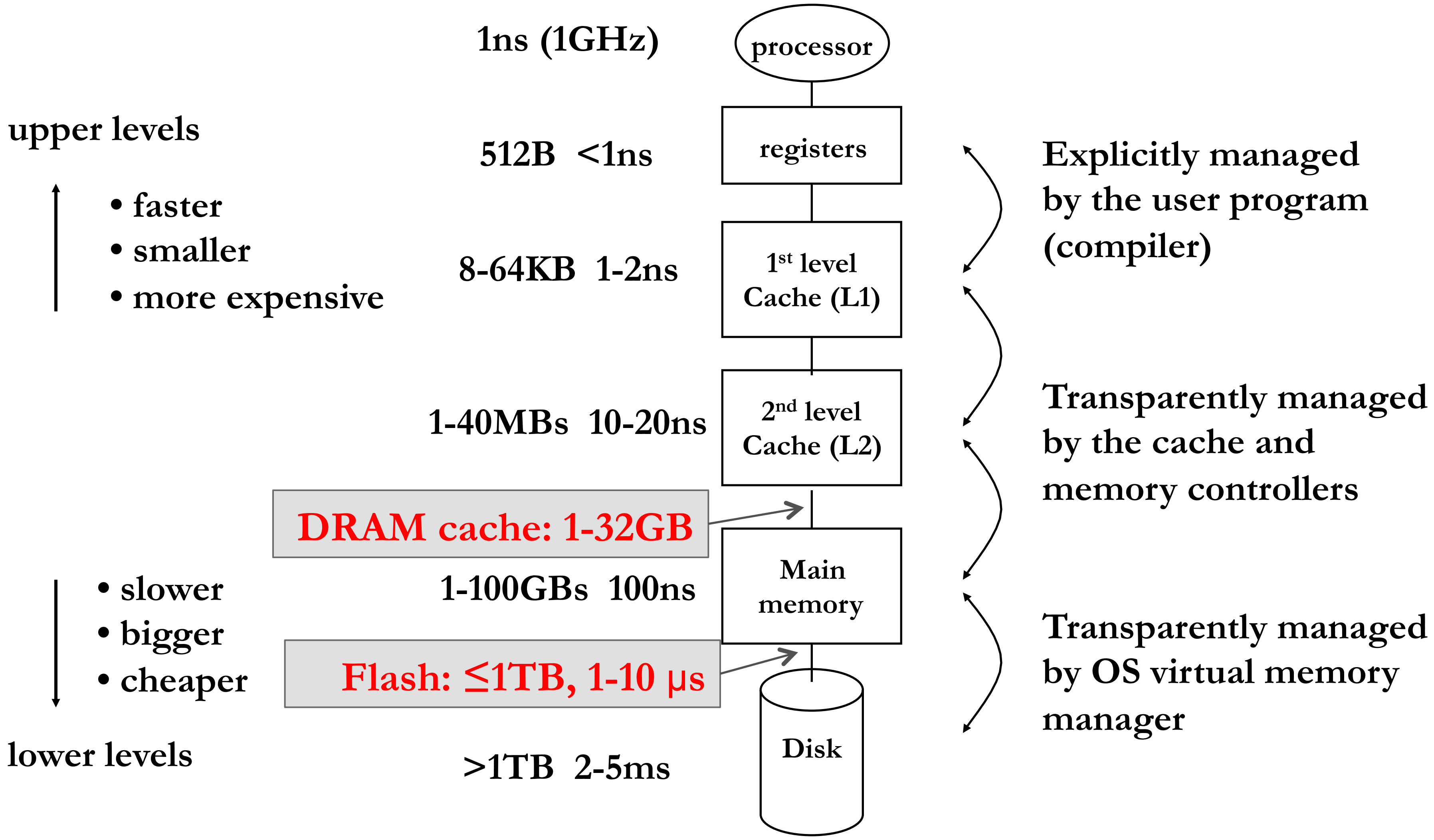
"Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available… we are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, and J. von Neumann - 1946

Why is a memory hierarchy effective?

- Temporal Locality:
  - A recently accessed memory location (instruction or data) is likely to be accessed again in the near future
- Spatial Locality:
  - Memory locations (instructions or data) close to a recently accessed location are likely to be accessed in the near future
- Why does locality exist in programs?
  - Instruction reuse: loops, functions
  - Data working sets: arrays, temporary variables, objects
- **Bottom-line: small, fast caches backed up by larger, slower memories give the impression of a single, large, fast memory**

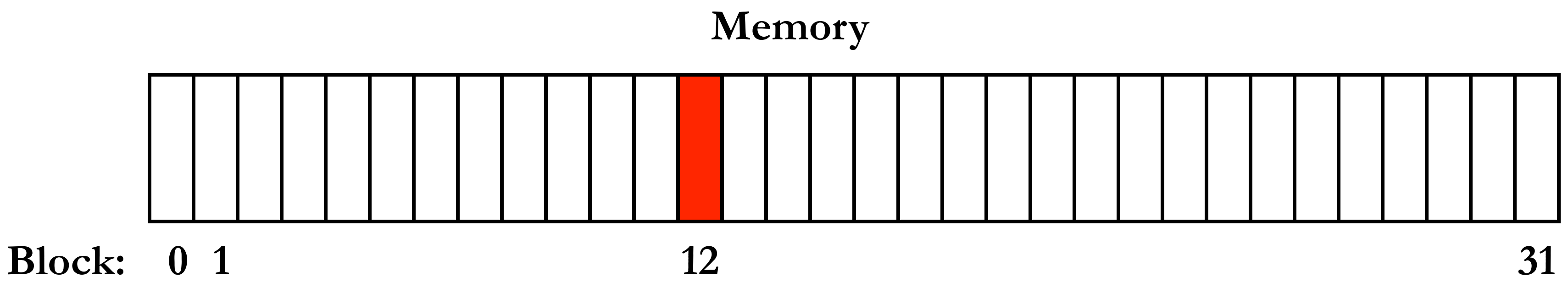
Memory Hierarchy

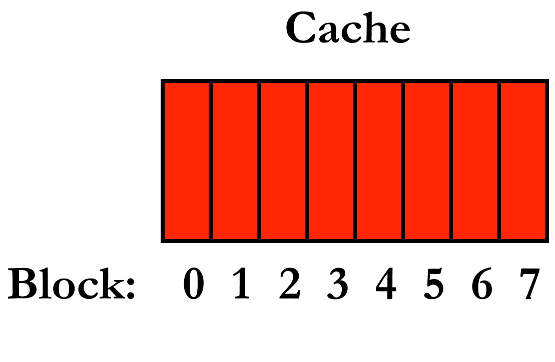


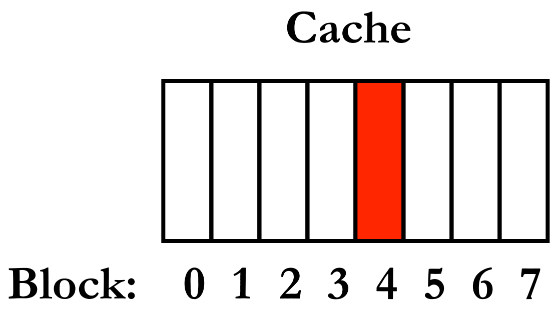
Memory Hierarchy Issues

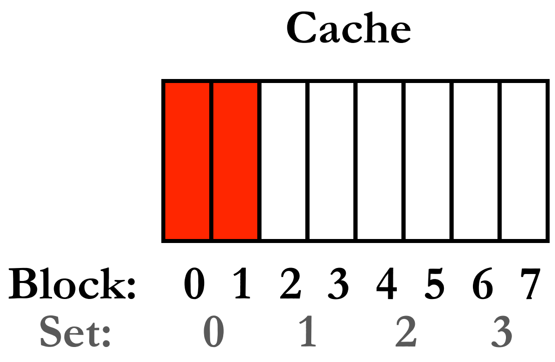
- Block size: smallest unit that is managed at each level
  - E.g., 64B for cache lines, 4KB for memory pages
- Block placement: Where can a block be placed?
  - E.g., direct mapped (exactly 1 location), fully associative (any location)
- Block identification: How can a block be identified?
  - E.g., hardware tag matching (e.g., cache), OS page table
- Block replacement: Which block should be replaced?
  - E.g., Random, Least recently used (LRU), Not recently used (NRU)
- Write strategy: What happens on a write?
  - E.g., write-through, write-back, write-allocate
- Inclusivity: whether next lower level contains all the data found in the current level
  - Inclusive, exclusive

Cache Block Placement









Fully associative:

block 12 can go anywhere in the cache

Direct mapped:

block 12 can only go into block 4 (12 mod 8)

Set associative:

block 12 can go anywhere in set 0 (12 mod 4)

Summary of Cache Associativity

- Fully associative
  - The block from the lower level can go into any location in the cache
  - Most flexible approach → lowest miss rate
  - Must search the whole cache to find the block
    → increased access time and high power consumption
- Direct mapped
  - The block from the lower level can only go into one location in the cache
  - Simplest approach to implement
  - Blocks mapping to the same location → increased miss rates
- Set associative
  - Split the cache into groups of *m* blocks (**sets**) → m-way set-associative
  - The block from the lower level can only go into one set, but within that set it can go anywhere
  - What's a good degree of associativity?
    - Higher level caches: 2- or 4-way common
    - Lower level caches: 8- to 32-way common

Cache Block Identification

- Every block is identified by a name or tag, which is part of the memory address
- Block tag is stored alongside the block data in the cache
- Block tags in the cache are compared with the tag of the requested block → often in parallel for set-associative caches, for speed
- Block tag usually comprises the high-order bits of the memory address

Full memory address:

Tag	**Index**	**Block offset**

- Data address: the address of the byte being referenced → 32 bits for MIPS
- Offset: the byte within the block; e.g., 6 bits for a 64B block
- Index: the set where the block can be found; e.g., 8 bits for a 4-way 64KB cache
- Tag: the "ID" of the block; e.g., 32-8-6=18 bits

Address Mapping Example

- Cache: 32 KBytes, 2-way, 64 byte lines
- Address: 32 bits

- Example: 0x000249F0 = $(0000\ 0000\ 0000\ 0010\ 0100\ 1001\ 1111\ 0000)_2$

Address Mapping Example

- Cache: 32 KBytes, 2-way, 64 byte lines
- Address: 32 bits

- Example: 0x000249F0 = $(\underbrace{0000\ 0000\ 0000\ 0010\ 01}_{\text{tag}}\underbrace{00\ 1001\ 11}_{\text{index}}\underbrace{11\ 0000}_{\text{offset}})_2$
  - Byte offset

    64 bytes → 6 bits ⇒ 11 0000

  - Index: 32K/64 = 512 lines in the cache

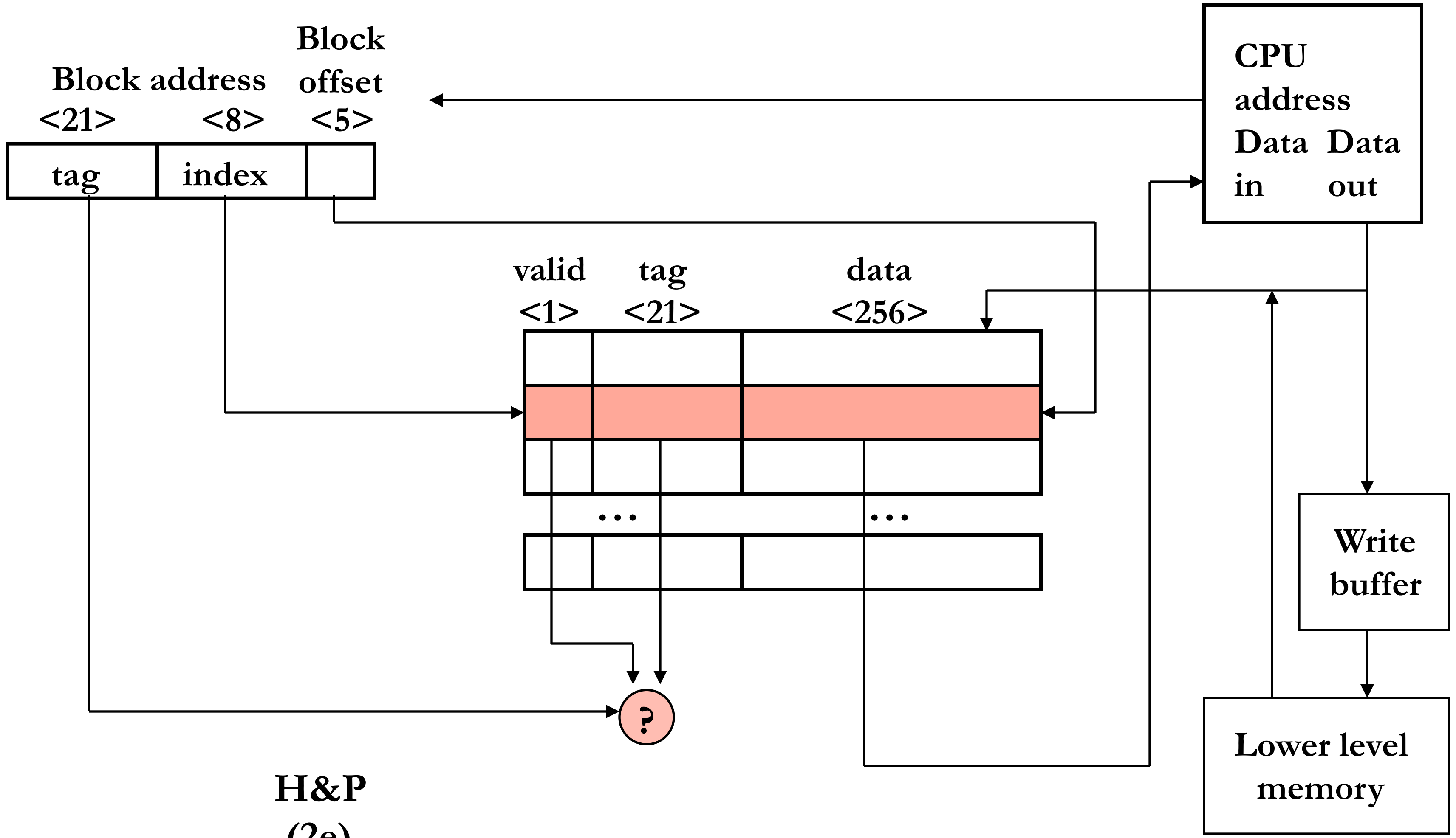
    512/2 = 256 sets in the cache

    256 sets → 8 bits ⇒ 00 1001 11 = 39

  - Tag:

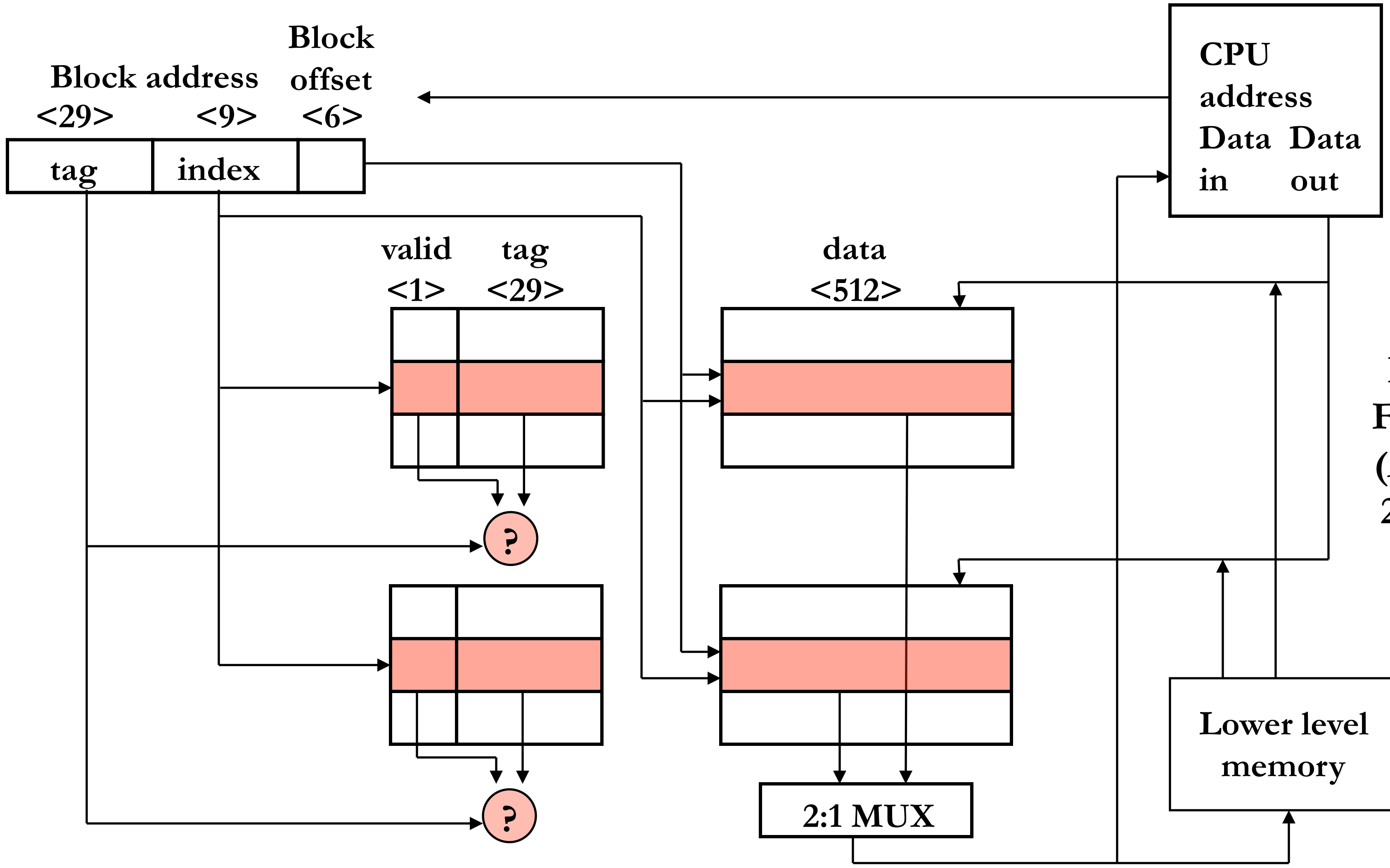
    0000 0000 0000 0010 01

Cache Organization: Direct mapped



H&P (2e) Fig. 5.5 (Alpha 21064)

Cache Organization: 2-way set associative



H&P Fig. 5.7 (Alpha 21264)

Cache Block Replacement

- To bring a new block in the cache, another block must be evicted
- Direct mapped caches: there is only one choice of block to evict
- Associative caches: how to choose a “victim”?
  - Random: select a victim block in the set randomly
  - Least-recently-used (LRU): select the block that has not been used for the longest period of time
    → works well in practice because of the principle of locality
  - Not-recently-used (NRU): select a block other than the most-recently used block.
    - Need less storage than LRU and performs better than random
  - Ideal: select the block that will not be used for the longest period of time
    → requires knowledge of the future → unrealistic!

Cache Write Strategies

How are the writes handled (i.e, when do stores reach a lower level of the memory hierarchy)?

- **Write through**: write to lower level as cache is modified
  - Writes will perform at the speed of the lower level of memory hierarchy ☹
  - Generates more traffic ☹
  - Lower level is kept **coherent** with higher level (particularly important for multi-processors) ☺
- **Write back**: only write to lower level when the block is evicted
  - Writes will perform at the speed of the higher level ☺
  - Generates less traffic ☺
  - Lower level can have stale data for some time (**cache-coherency** problem) ☹

Cache Write Strategies

What happens if the block is not found in the cache?

- **Write allocate**: bring the block into the cache and write to it
  - Good if block will soon be used by another memory access (locality)
  - Usually used with write back
- **Write no-allocate**: do not bring block into cache and modify data in the lower level
  - Good if no memory access to the same block occur in the near future
  - Usually used with write through

Multi-Level Caches

Do lower-level caches keep a copy of the block that's brought into a higher-level cache?

- Inclusive caches:
  - Lower-level cache has a copy of every block in higher-level caches
  - Wastes capacity of lower-level caches ☹
  - Simplifies finding a cache block by another entity (e.g., by other processors) ☺
- Exclusive caches:
  - A block may reside in only one level of the cache hierarchy
  - Maximizes aggregate capacity of the cache hierarchy ☺
  - Requires a uniform block size for all cache levels ☹