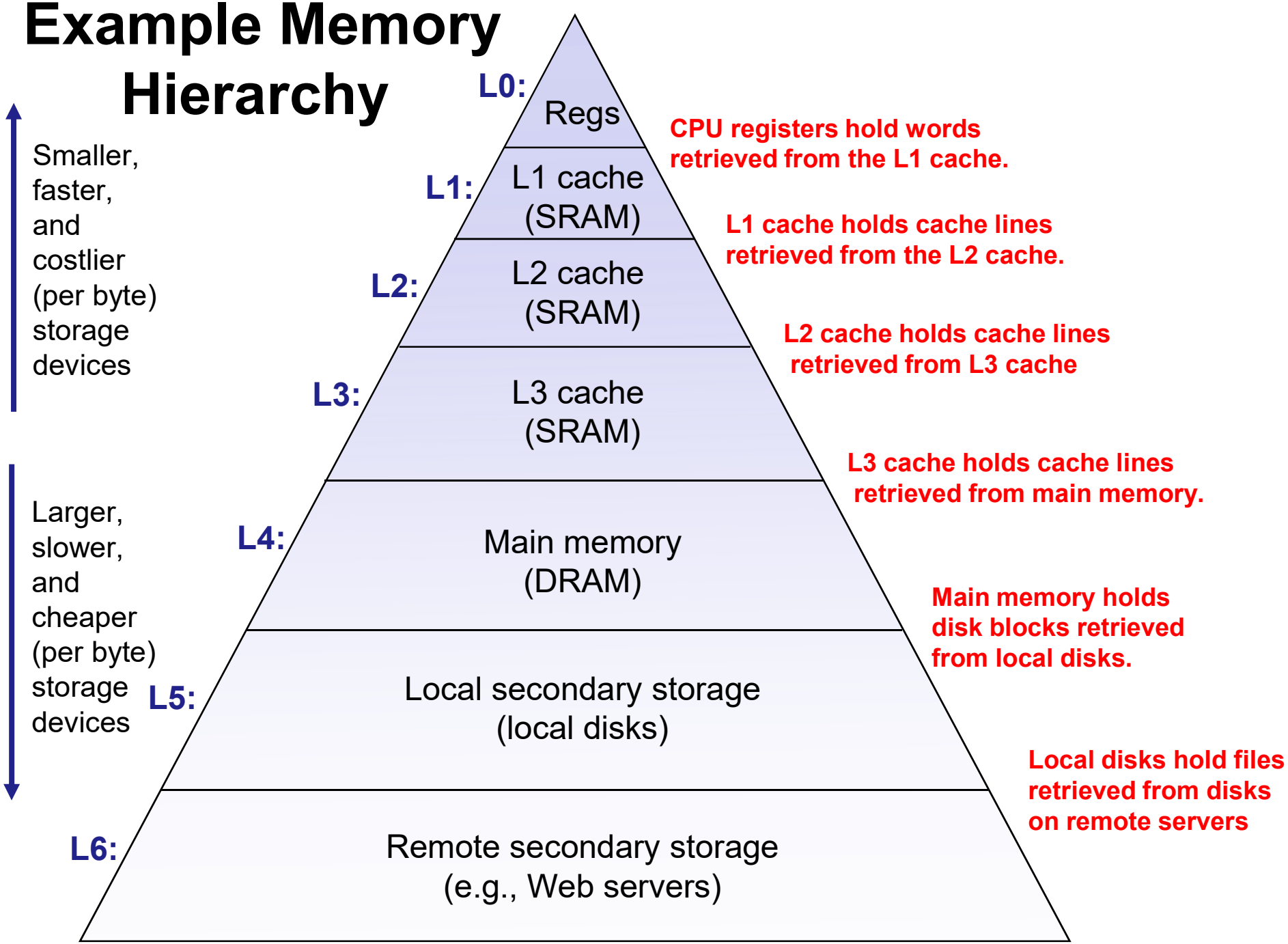
# Computer Systems Organization

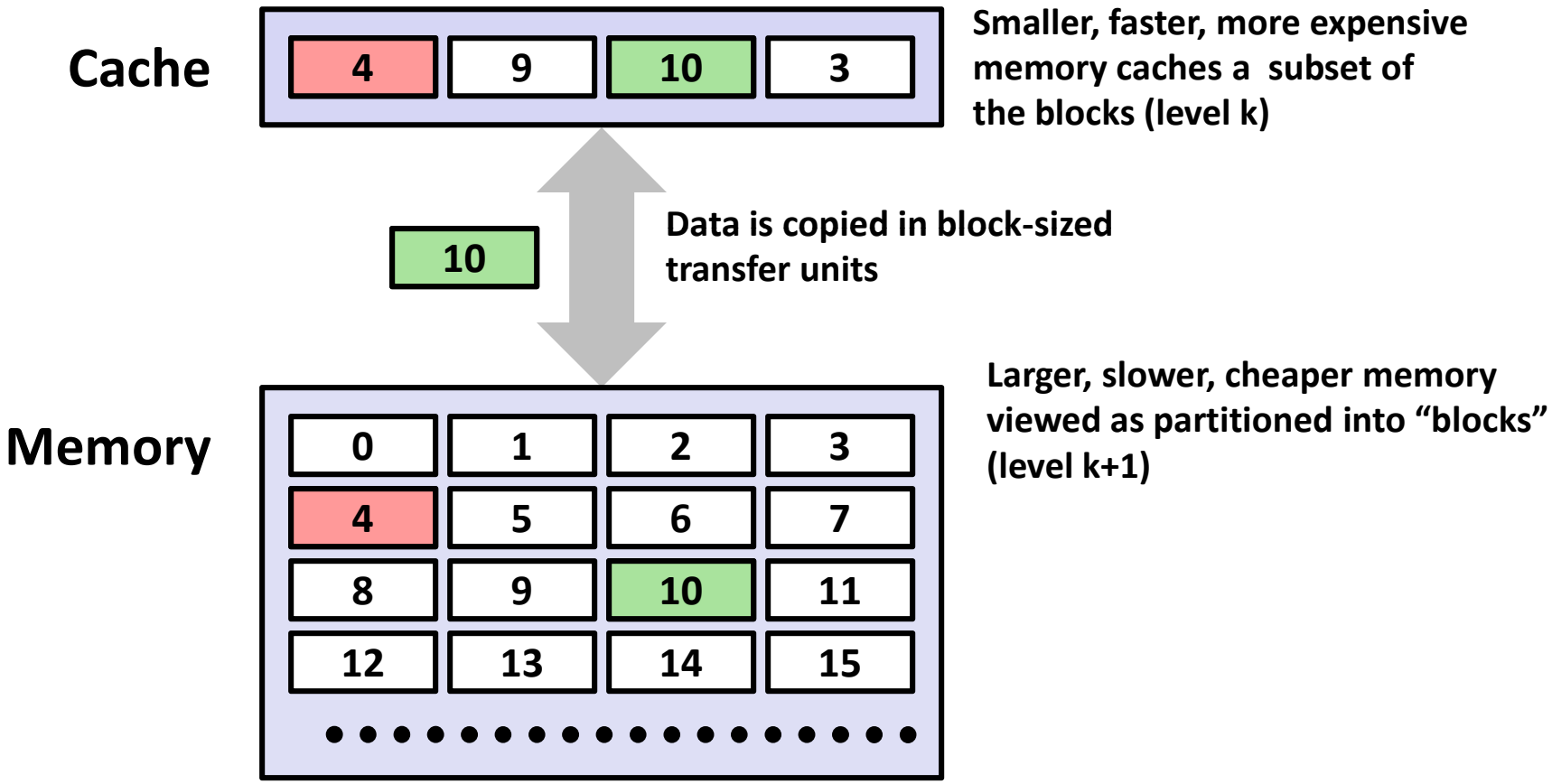
**Topic 5**

Based on chapter 6 from Computer Systems by Randal E. Bryant and David R. O'Hallaron

# Example Memory Hierarchy



# General Cache Concept



# Cache Concepts

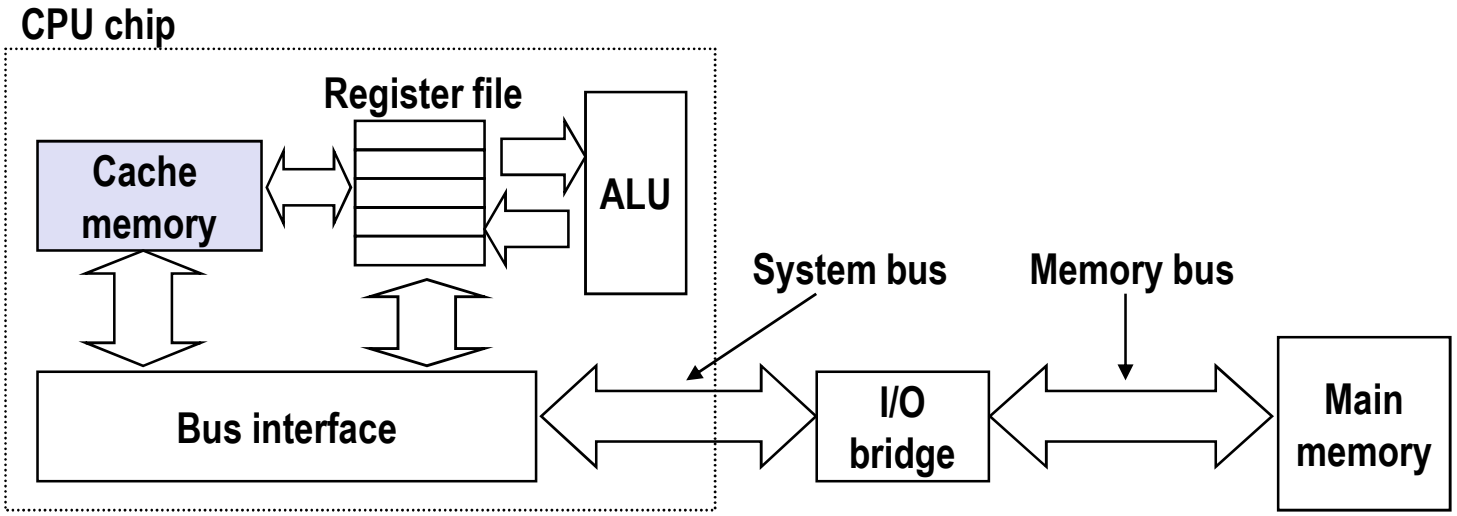
- When a program needs a particular object d from level k+1, first looks for blocks in k
  - If d is cached at level k, called cache hit
  - Else called a cache miss
- When a miss happens, cache at level k fetches from k+1 possibly overwriting an existing block called replacing or evicting the block (sometimes referred to as a victim block)
  - Which block to replace is governed by the cache's replacement policy
  - Random replacement policy would chose a random victim block
  - Cache with LRU (Least Recently Used) replacement policy would choose the block that was accessed furthest in the past
- Empty cache is sometimes referred to as cold cache and misses due to empty cache are called compulsory or cold misses
- Whenever there is a miss, cache at level k must implement a placement policy – determines where to place block it has retrieved

# Cache Concepts

- Most flexible placement policy is to allow any block from k+1 to be stored in any block at k
  - Can be expensive since randomly placed blocks can be expensive to locate (esp. high in memory hierarchy (close to CPU) where speed is at a premium)
- A simpler placement policy restricts a particular block at level k+1 to a small subset to a small subset of block at level k
  - E.g., blocks 0,4,8,12 map to block 0 in level k, 1,5,9,13 map to block 1 and so on
- Restrictive placement policies can lead to a conflict miss
  - While cache maybe large enough, since data objects map to the same cache block, cache keeps missing
  - E.g., request block 0, then block 8 and so on

# Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



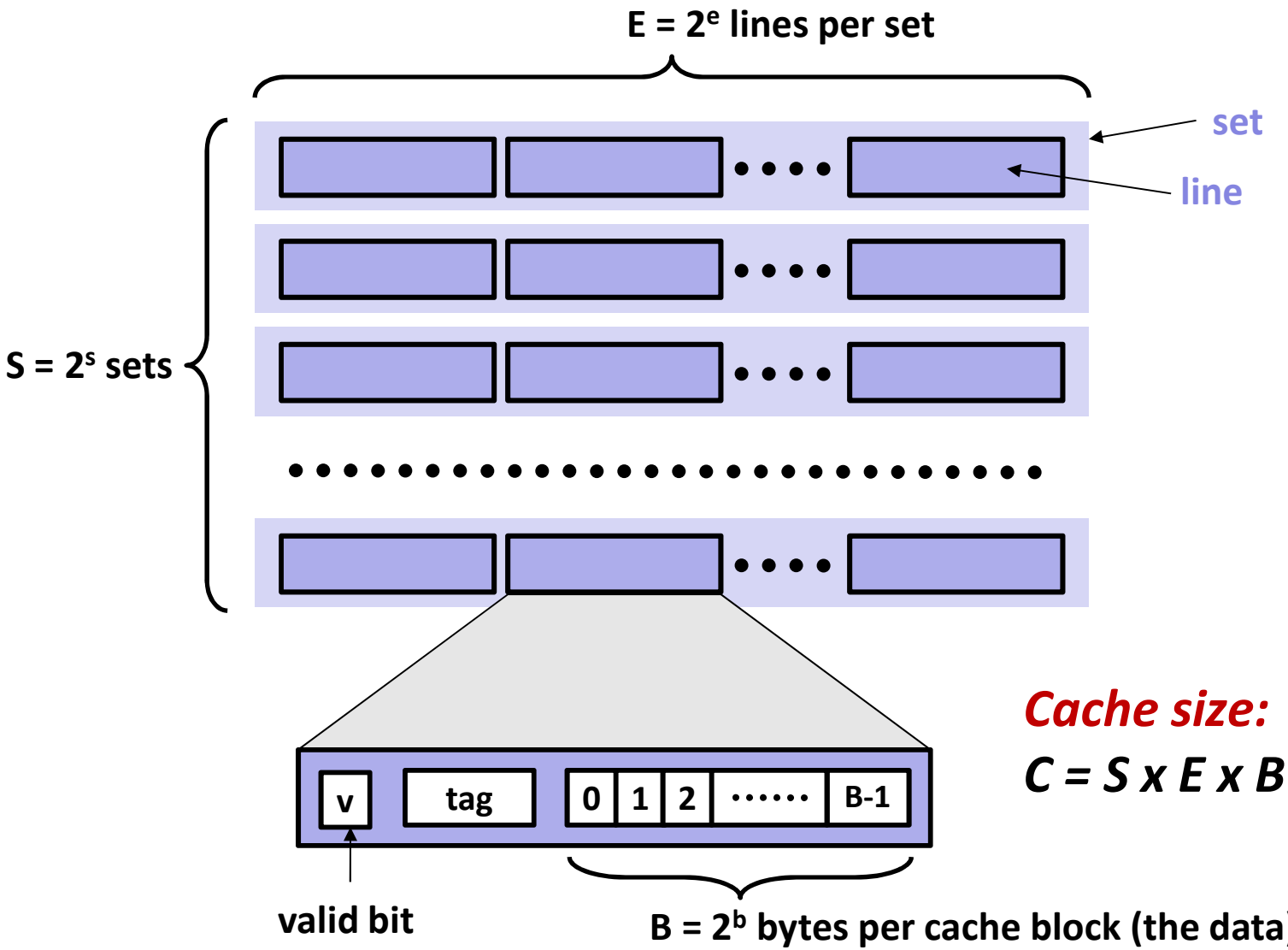
# Cache Memories

- L1 (level 1) cache is inserted between CPU register file and main memory – accessed pretty fast, typically in about 4 clock cycles
- L2 cache between L1 cache and main memory – accessed in about 10 clock cycles
- L3 cache between L2 cache and main memory – accessed in about 50 cycles
- While there is considerable variety in arrangements, general principles are same

# General Cache Organization

- Consider a system with m bits for each memory address that forms M = 2^m unique addresses
- Cache organized as an array S = 2^s cache sets
- Each set has E cache lines
- Each line consists of:
  - A data block B = 2^b bytes
  - A valid bit v indicates whether or not line contains meaningful information and
  - Tag bits to uniquely identify block stored in cache line

# General Cache Organization (S, E, B)




***Cache size:***

***C = S x E x B data bytes***

# General Cache Organization

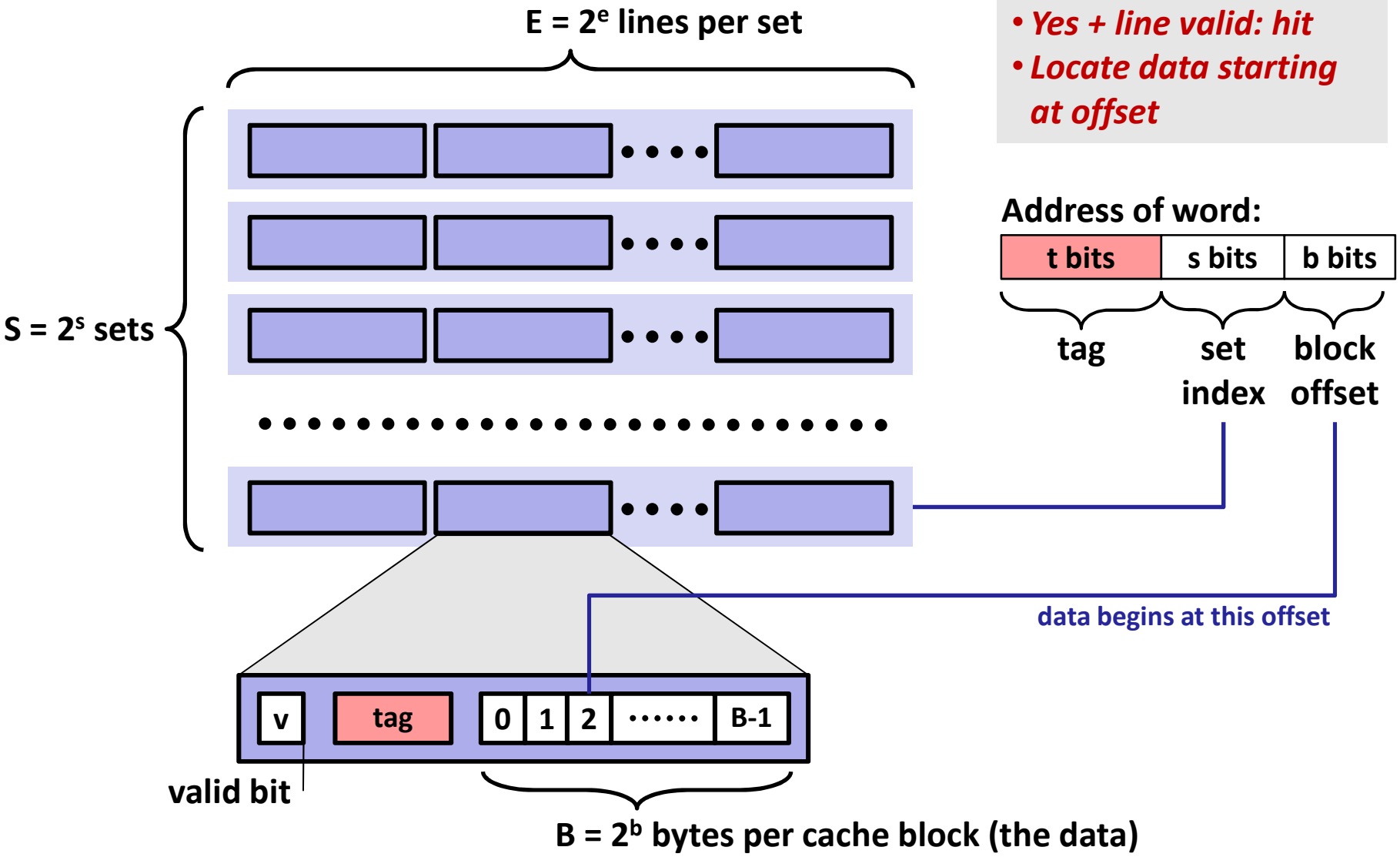
- When CPU instructed by load instruction to read a word from address A of main memory, sends address A to cache
- If cache holds a copy of the word at address A, sends the word immediately back to the CPU
- How does the cache know whether it had copy ?
- Cache is organized so it can find the requested word by simply inspecting the bits of the address
- Parameters S and B result in partitioning of the m address bits into 3 fields
- s set index bits in A form an index into the array of S sets
  - First set is set 0, second set is set 1 and so on
  - Set index bits tell us which set the word must be stored in
- Once we know which set the word must be contained in, the t tag bits in A tell which line (if any) in the set contains the word

# General Cache Organization

- A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A
- Once located the line identified by the tag in the set identified by the set index, the b block offset bits give us the offset of the word in the B-byte data block

# Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

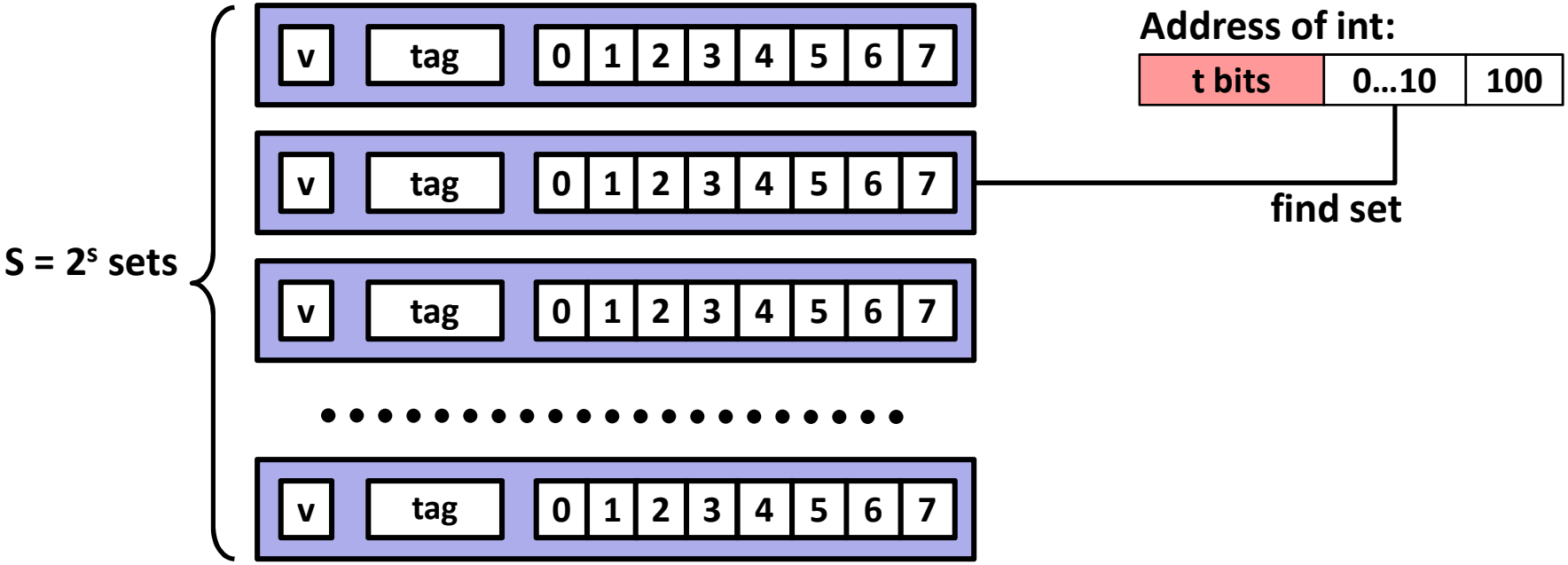


# Classes of Caches

- Caches are mapped into different classes based on E, the number of cache lines per set
- Cache with exactly one line per set (E = 1) is known as direct mapped cache
  - Are the simplest both to implement and understand
- Process that a cache goes through determining whether a request is a hit or a miss and then extracting the requested word consists of three steps:
  - Set selection
  - Line matching
  - Word extraction

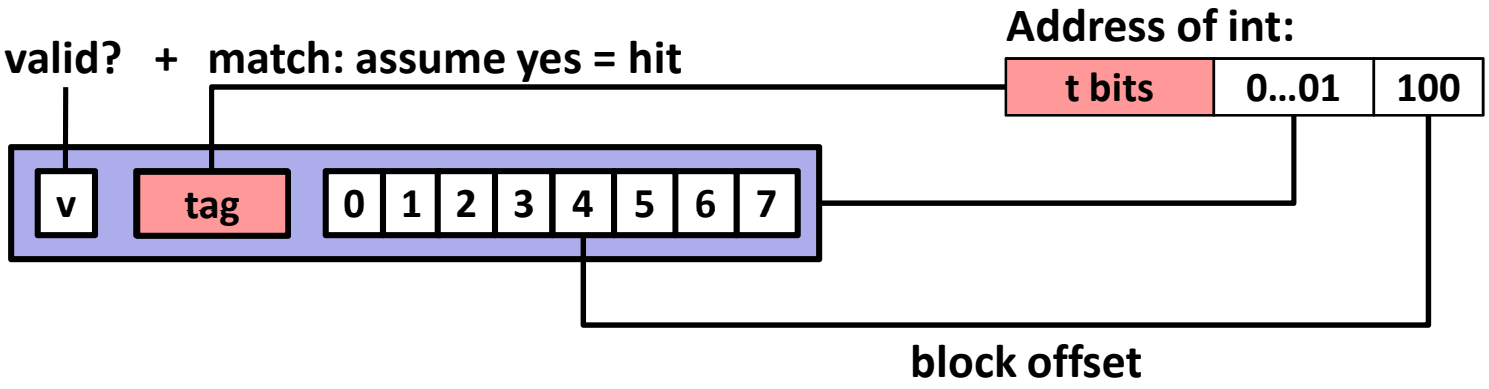
# Example: Direct Mapped Cache (E = 1)

**Direct mapped: One line per set**
**Assume: cache block size 8 bytes**



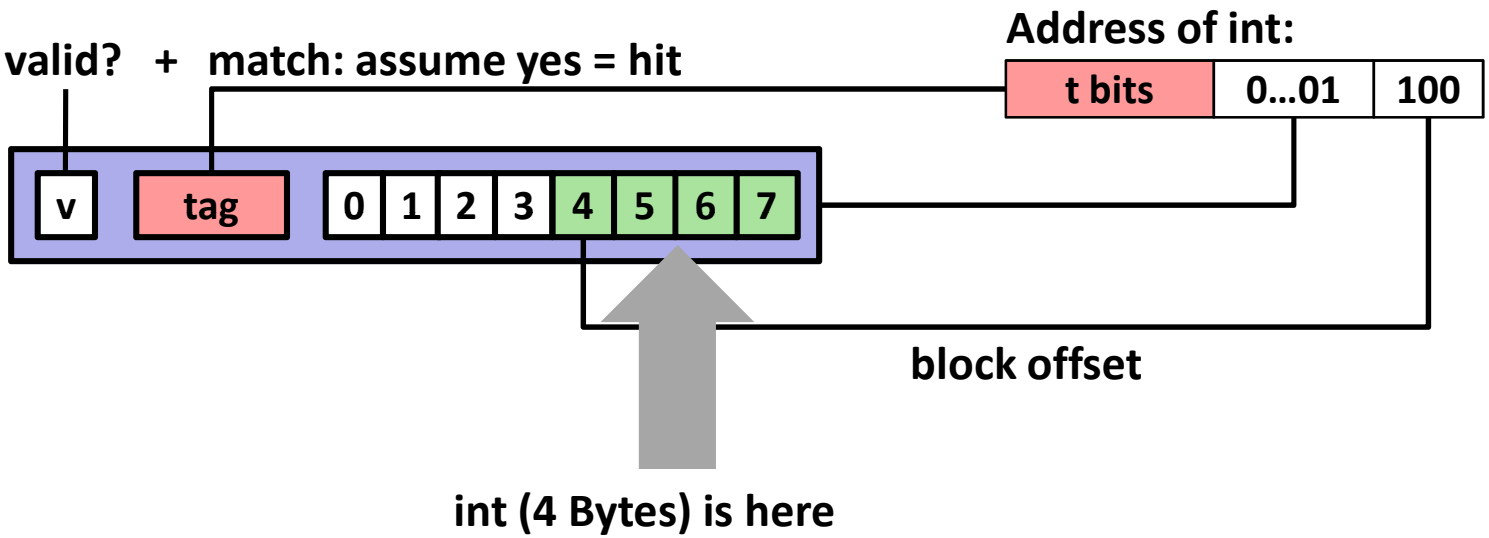
# Example: Direct Mapped Cache (E = 1)

**Direct mapped: One line per set**
**Assume: cache block size 8 bytes**



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes




If tag doesn't match: old line is evicted and replaced

# Direct-Mapped Cache Simulation

| t=1 | s=2 | b=1 |
|---|---|---|
| x | xx | x |

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

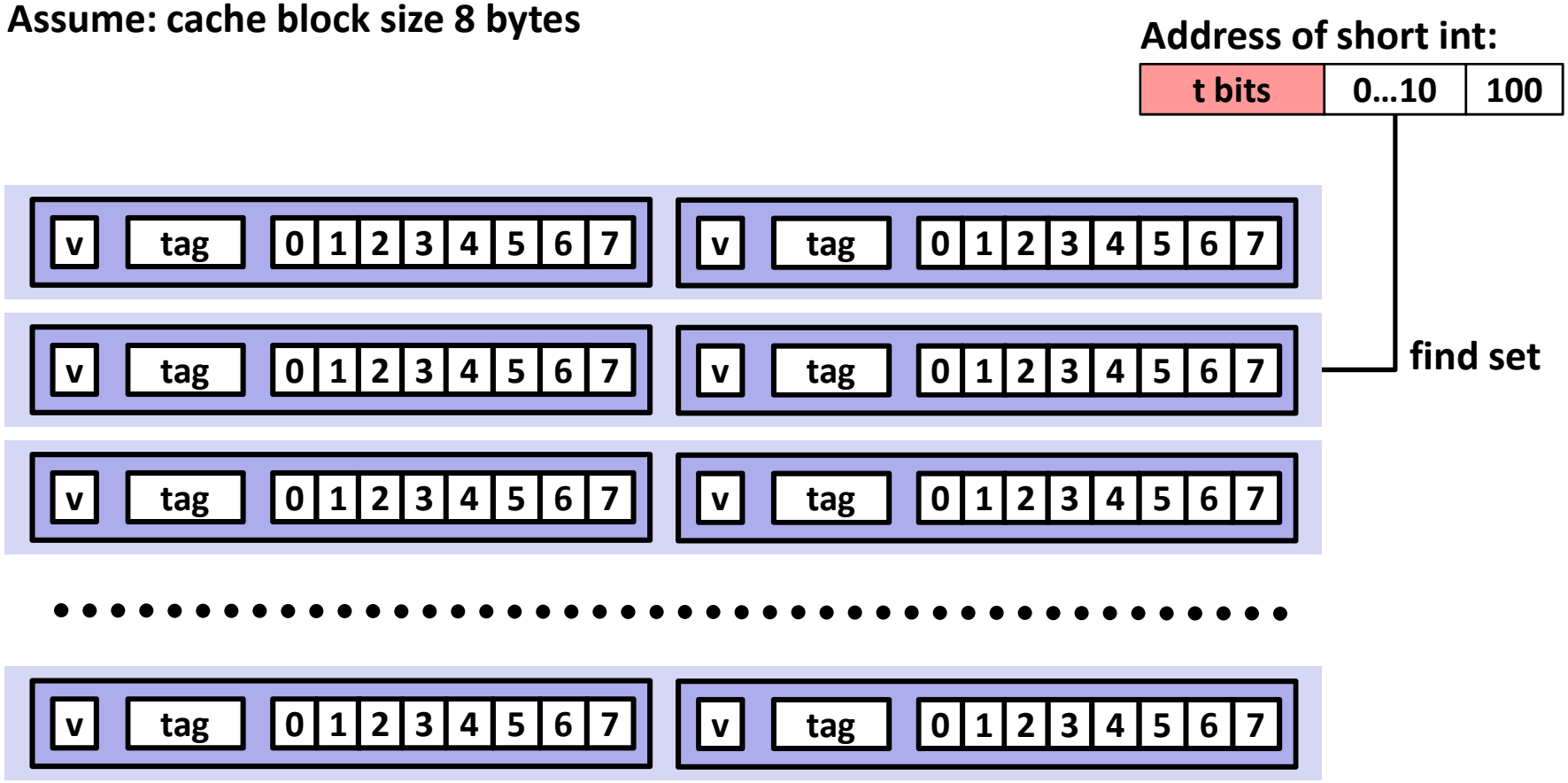
| | | |
|---|---|---|
| **0** | **[0$\underline{00}$0$_2$],** | miss |
| **1** | **[0$\underline{00}$1$_2$],** | hit |
| **7** | **[0$\underline{11}$1$_2$],** | miss |
| **8** | **[1$\underline{00}$0$_2$],** | miss |
| **0** | **[0$\underline{00}$0$_2$]** | miss |

| | v | Tag | Block |
|---|---|---|---|
| **Set 0** | 1 | 0 | M[0-1] |
| **Set 1** | | | |
| **Set 2** | | | |
| **Set 3** | 1 | 0 | M[6-7] |

# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

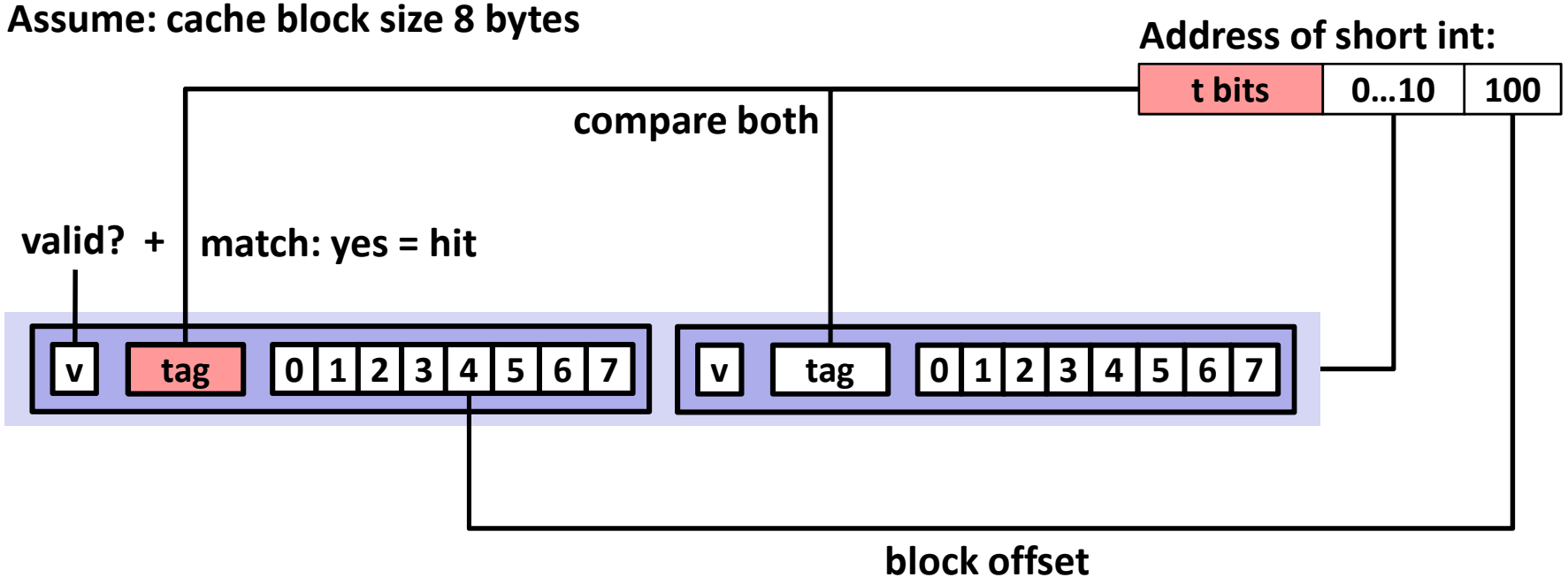
Assume: cache block size 8 bytes



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

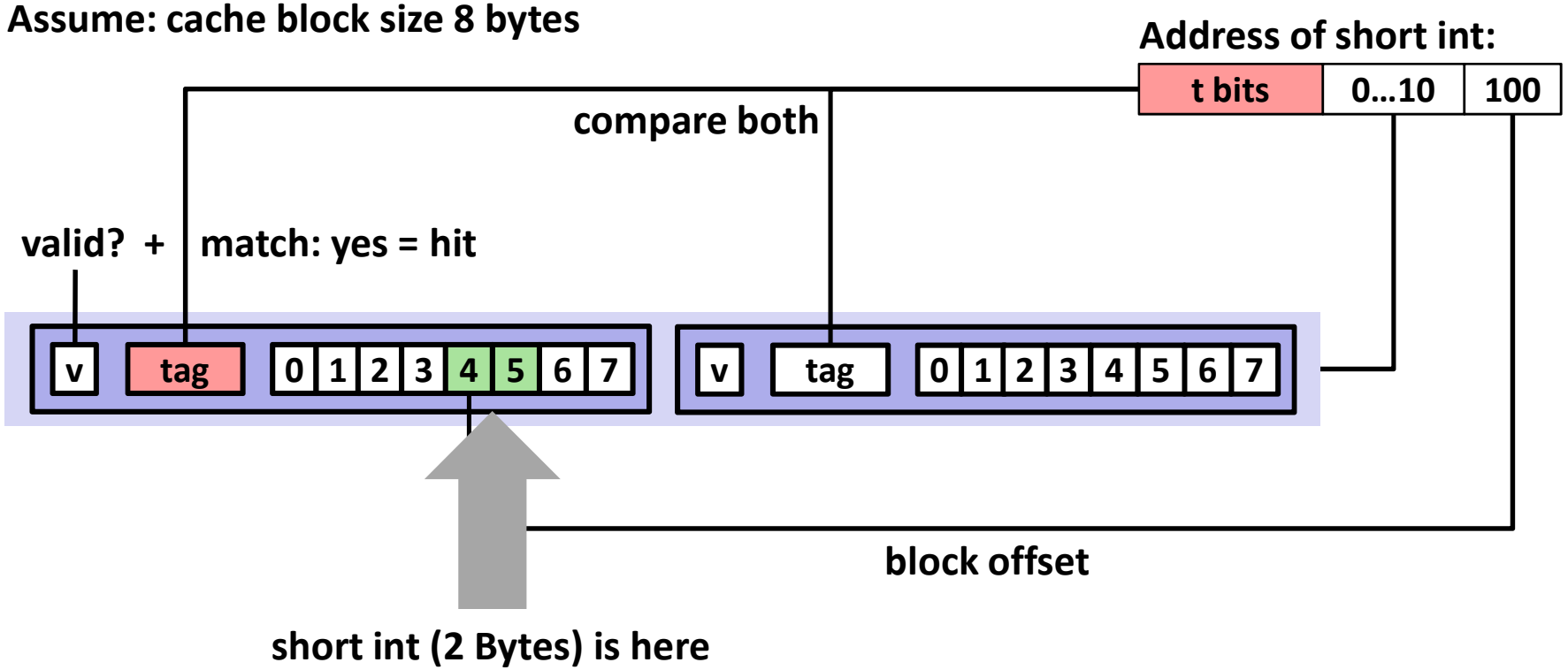
Assume: cache block size 8 bytes



# E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes




**No match:**

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

# 2-Way Set Associative Cache Simulation

| t=2 | s=1 | b=1 |
|---|---|---|
| xx | x | x |

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

| | | |
|---|---|---|
| **0** | **[000$_2$],** | miss |
| **1** | **[001$_2$],** | hit |
| **7** | **[111$_2$],** | miss |
| **8** | **[1000$_2$],** | miss |
| **0** | **[000$_2$]** | hit |

| | v | Tag | Block |
|---|---|---|---|
| **Set 0** | 1 | 00 | M[0-1] |
| | 1 | 10 | M[8-9] |
| **Set 1** | 1 | 01 | M[6-7] |
| | 0 | | |

# Fully Associative Caches

- Consists of a single set that contains all the cache lines
- No set index bits needed since only one set
- Line matching and word selection in a fully associative cache work the same as with a set associative cache
- Since the cache circuitry needs to search for many tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast, hence used for small caches