COMP4611: Design and Analysis of Computer Architectures

Memory System

Cache Design

Lin Gu

CSE, HKUST

Locality

```
int A[100], B[100], C[100], D;
for (i=0; i<100; i++) {
    C[i] = A[i] * B[i] + D;
}
```

			D	C[99]	C[98]	C[97]	C[96]
. . . . .							
C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
. . . . .							
B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]
B[3]	B[2]	B[1]	B[0]	A[99]	A[98]	A[97]	A[96]
. . . . .							
A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]

Memory Hierarchy: Motivation
The Principle Of Locality

- Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (loops, data arrays).

- Two types of locality:
  - **Temporal Locality: If an item is referenced, it will tend to be referenced again soon.**
  - **Spatial locality: If an item is referenced, items whose addresses are close by will tend to be referenced soon .**
- The presence of locality in program behavior (e.g., loops, data arrays), makes it possible to satisfy a large percentage of the program's data accesses (both instructions and operands) using memory levels closer to the CPU.

Locality Example

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

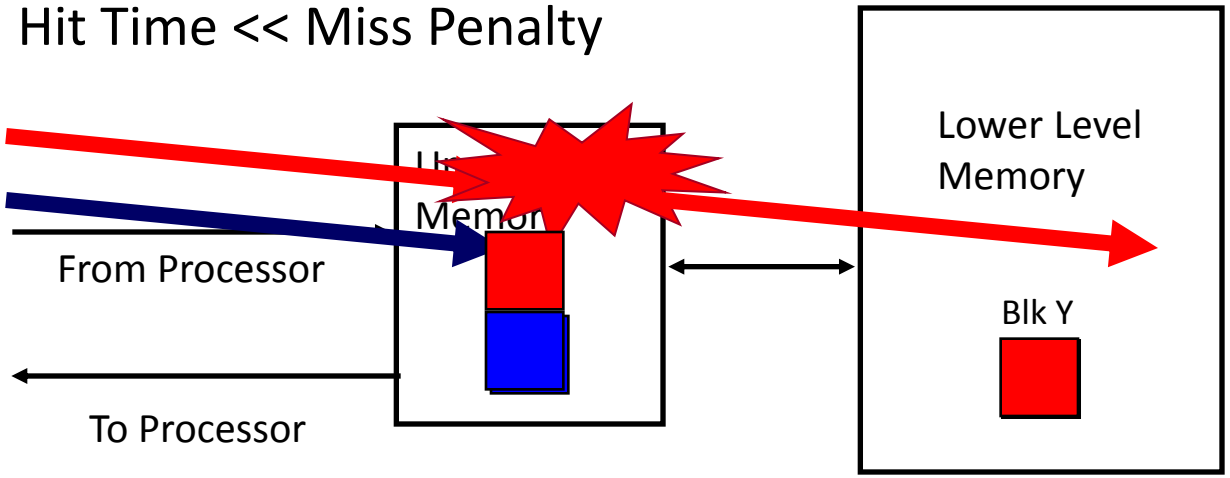
- Data
  - Reference array elements in succession (stride-1 reference pattern): Spatial locality
  - Reference `sum` each iteration: Temporal locality
- Instructions
  - Reference instructions in sequence: Spatial locality
  - Cycle through loop repeatedly: Temporal locality

Memory Hierarchy: Terminology

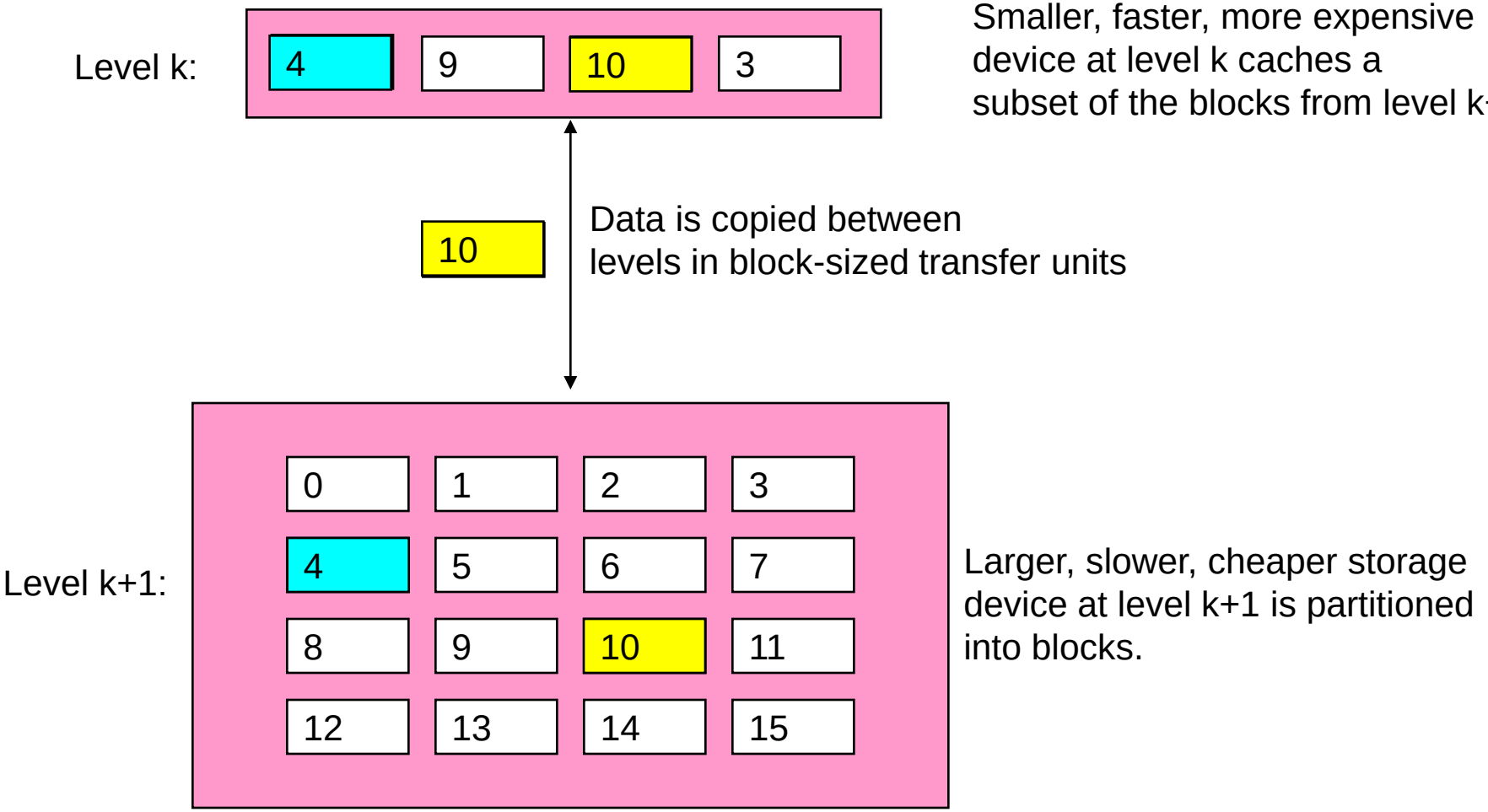
- **A Block:** The smallest unit of information transferred between two levels.
- **Hit:** Item is found in a block in the upper level (example: Block X)
  - **Hit Rate: The fraction of memory accesses found in the upper level.**
  - **Hit Time: Time to access the upper level which consists of**

    memory access time + time to determine hit/miss
- Miss: Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate = 1 - (Hit Rate)**
  - **Miss Penalty: Time to replace a block in the upper level +**

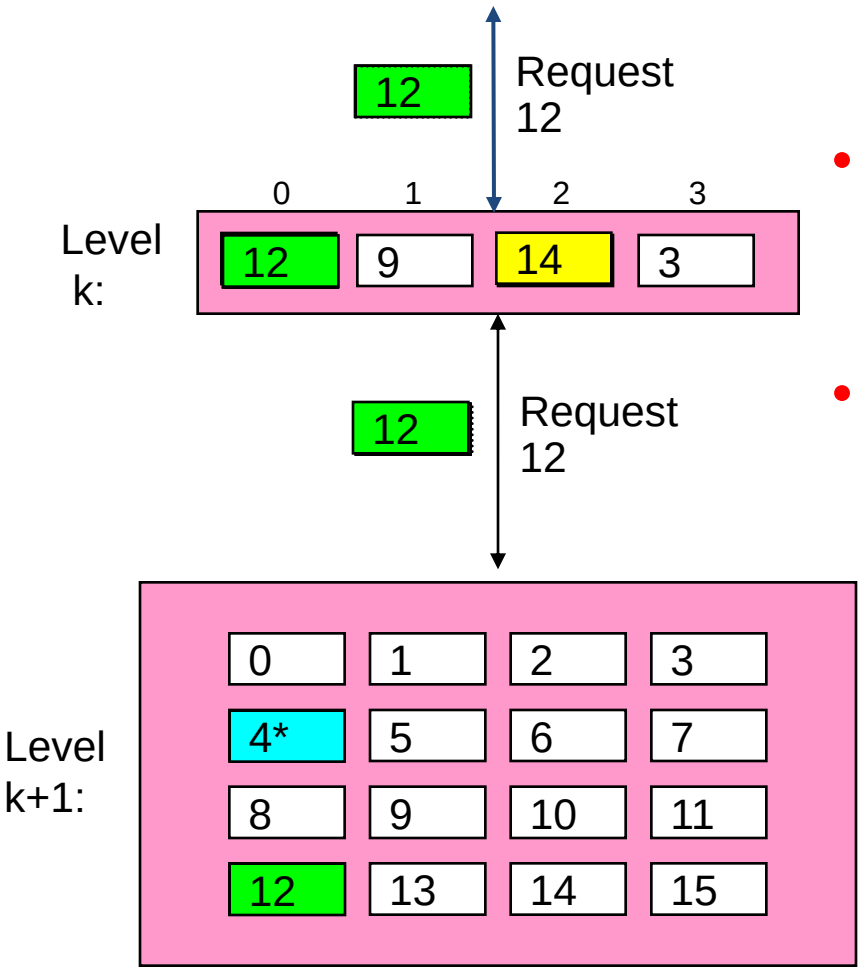
    Time to deliver the block to the processor (or the further-upper-level)
- Hit Time << Miss Penalty



Caching in a Memory Hierarchy



General Caching Concepts



- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
    - Placement policy: where can the new block go? E.g., b mod 4
    - Replacement policy: which block should be evicted? E.g., LRU

Cache Design & Operation Issues

- Q1: Where can a block be placed in cache?
  (Block placement strategy & Cache organization)
  - **Fully Associative, Set Associative, Direct Mapped.**
- Q2: How is a block found if it is in cache?
  (Block identification)
  - **Tag/Block.**
- Q3: Which block should be replaced on a miss?
  (Block replacement)
  - **Random, LRU.**
- Q4: What happens on a write?
  (Cache write policy)
  - **Write through, write back.**

Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frame addresses divide cache into three organizations:

- **Direct mapped cache:** A block can be placed in one location only, given by:

  (Block address) MOD (Number of blocks in cache)

  - Advantage: It is easy to locate blocks in the cache (only one possibility)
  - Disadvantage: Certain blocks cannot be simultaneously present in the cache (they can only have the same location)

Cache Organization: Direct Mapped Cache

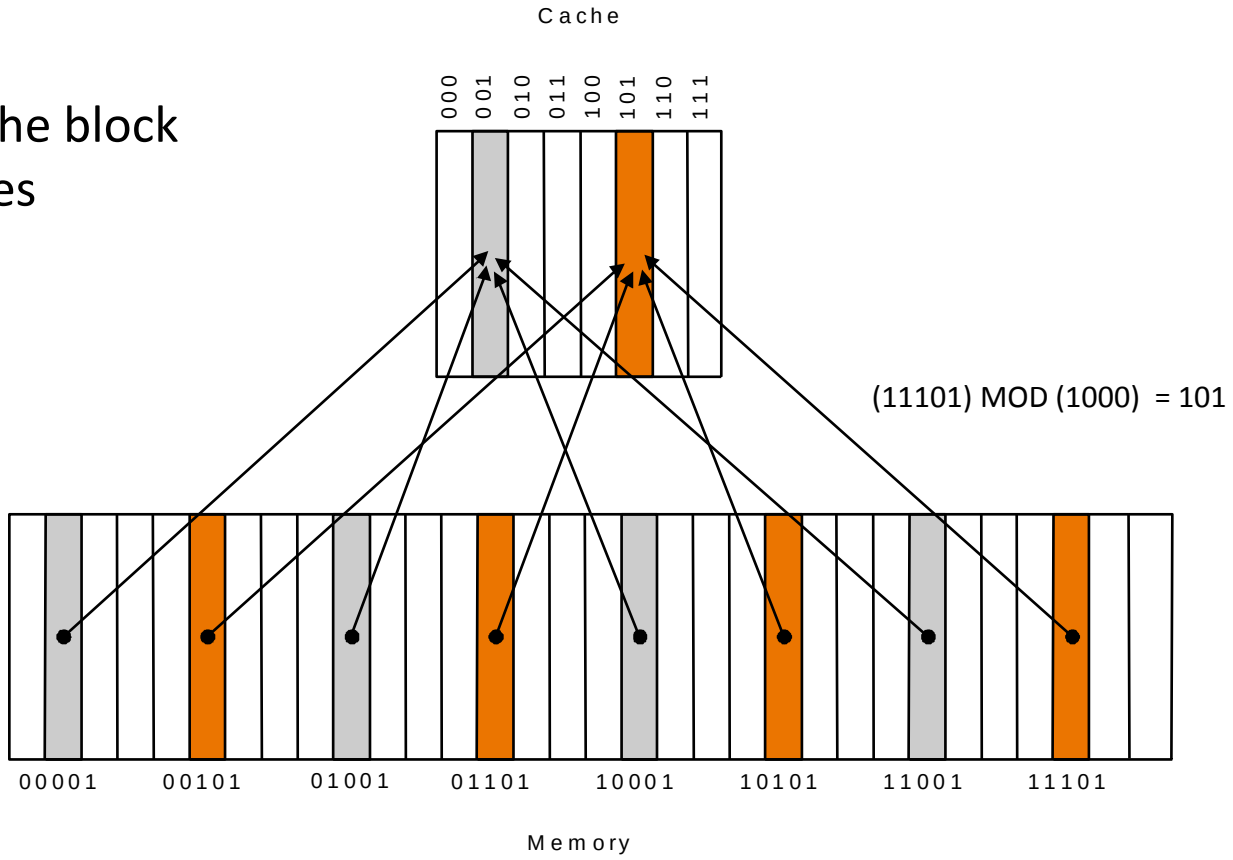
A block can be placed in one location only, given by:

(Block address) MOD (Number of blocks in cache)

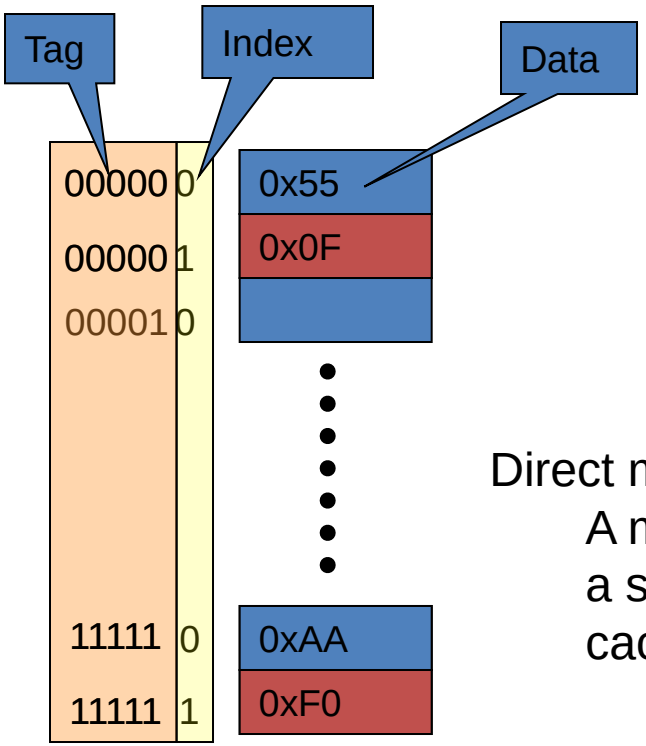
In this case: (Block address) MOD (8)

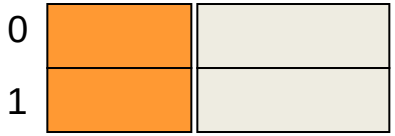
8 cache block frames

32 memory blocks cacheable



Direct Mapping





Direct mapping:

A memory value can only be placed at a single corresponding location in the cache

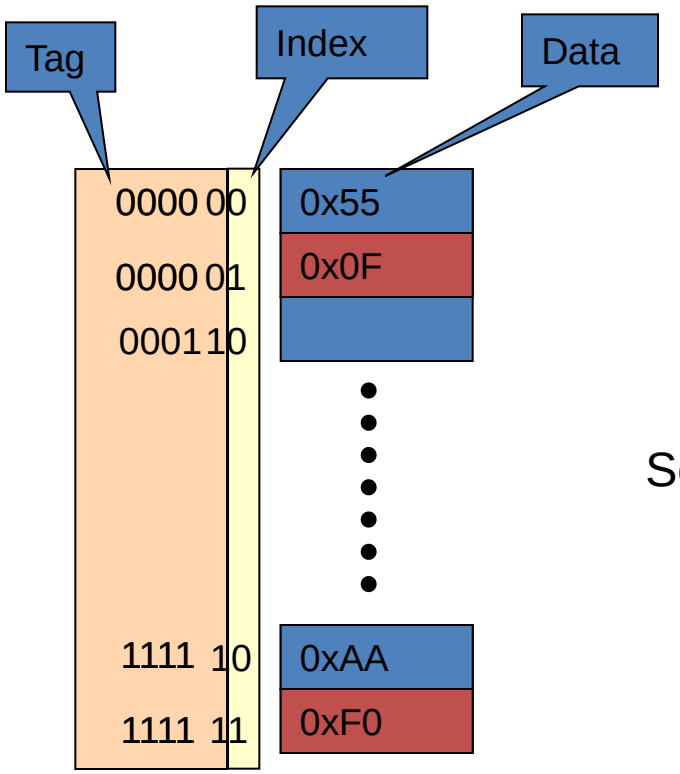
Cache Organization & Placement Strategies

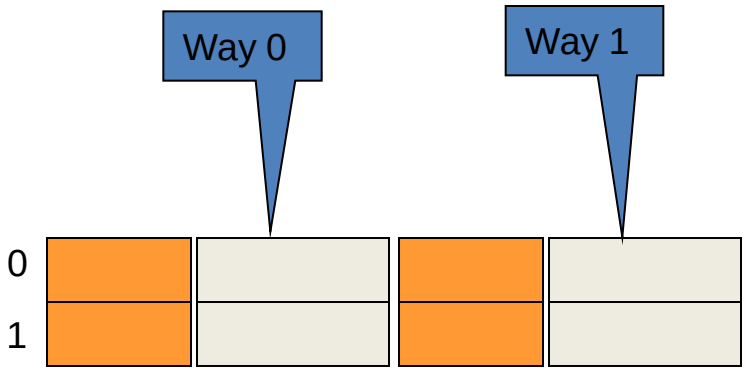
- **Fully associative cache:** A block can be placed anywhere in cache.
  - Advantage: No restriction on the placement of blocks. Any combination of blocks can be simultaneously present in the cache.
  - Disadvantage: Costly (hardware and time) to search for a block in the cache
- **Set associative cache:** A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:

  (Block address) MOD (Number of sets in cache)

  - If there are *n* blocks in a set the cache placement is called *n*-way set-associative, or n-associative.
  - A good compromise between direct mapped and fully associative caches (most processors use this method).

Set Associative Mapping (2-Way)

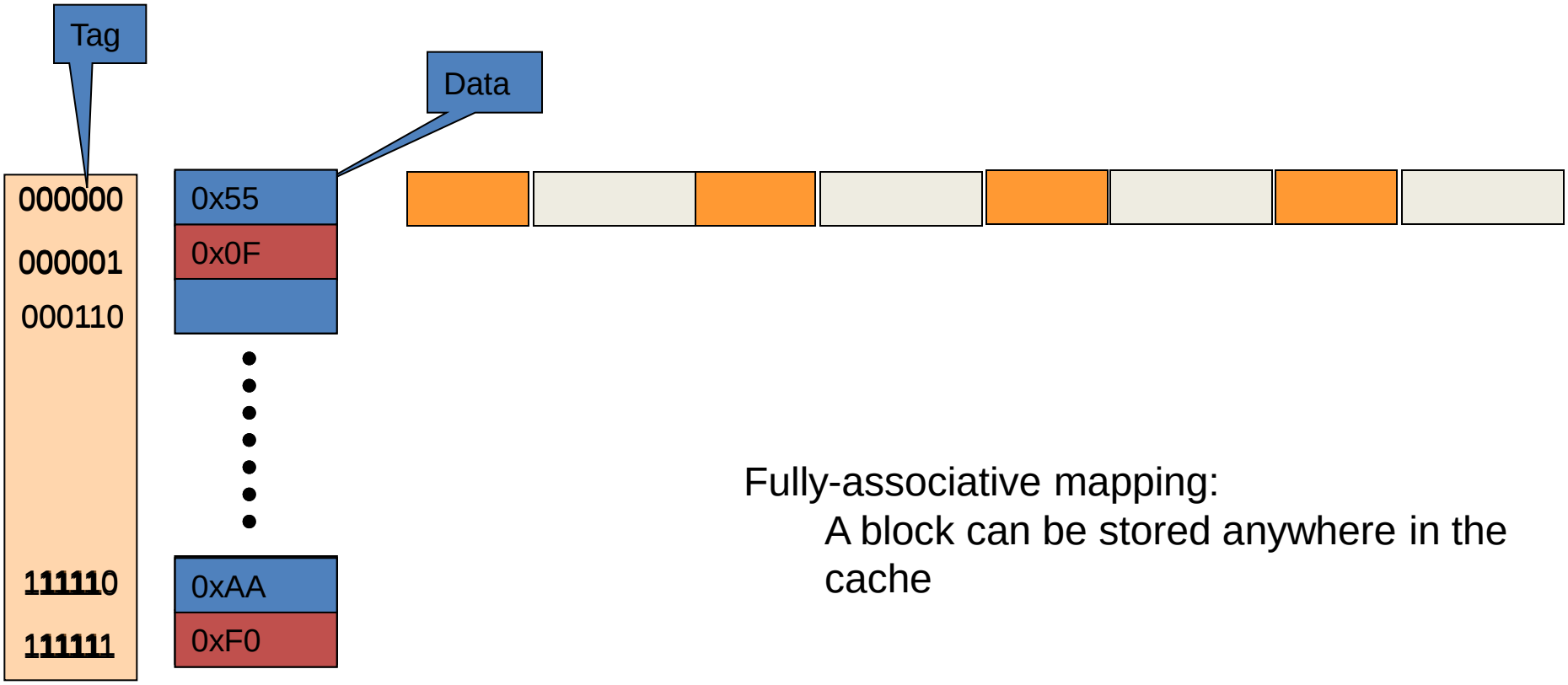




Set-associative mapping:

A memory value can be placed in any block frame of a set corresponding to the block address

Fully Associative Mapping



Fully-associative mapping:

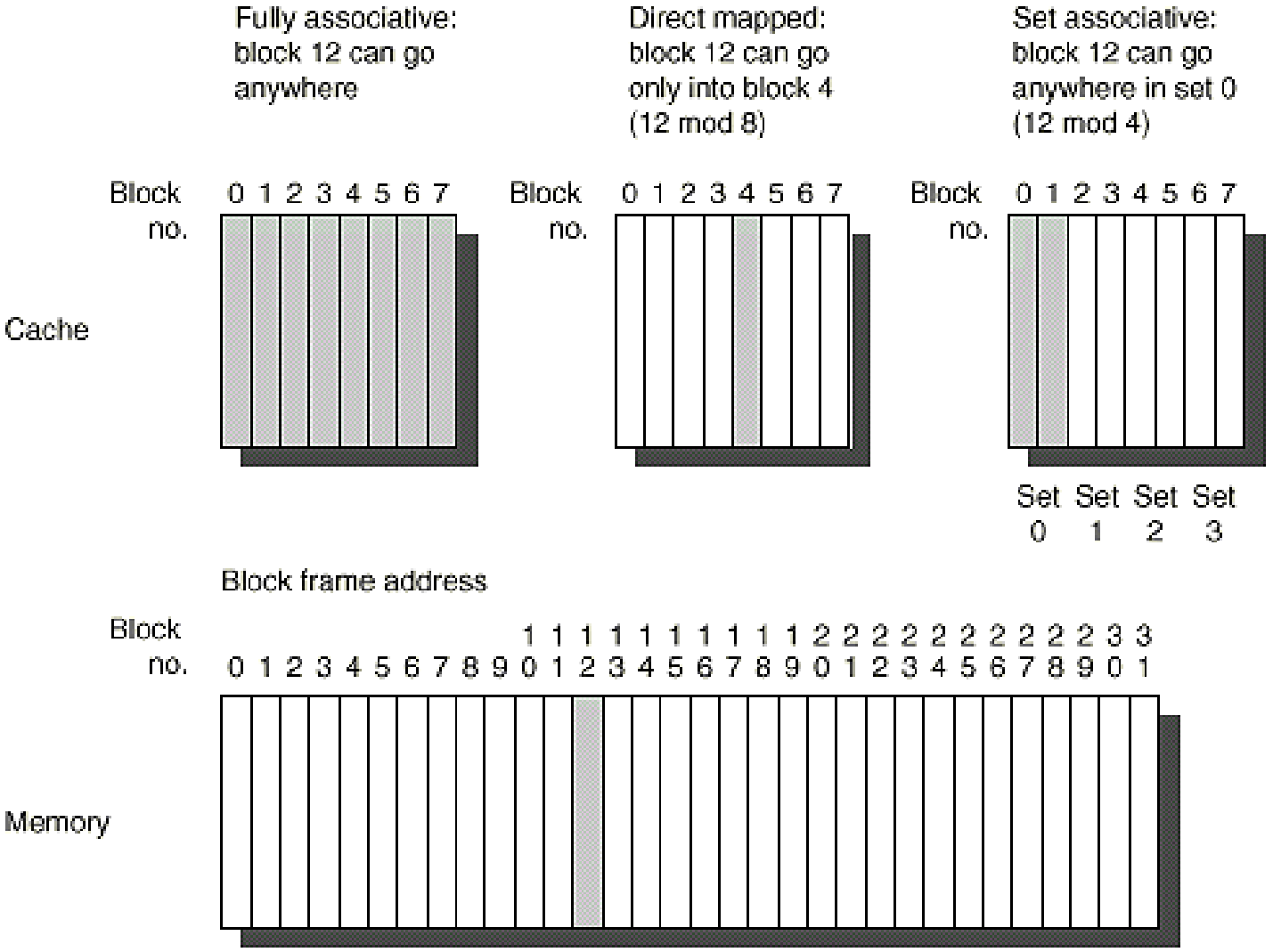
A block can be stored anywhere in the cache

Types of Caches: Organization

Type of cache	Mapping of data from memory to cache	Complexity of searching the cache
Direct mapped (DM)		Easy search mechanism
Set-associative (SA)	A memory value can be placed in **any of a set of locations** in the cache	Slightly more involved search mechanism
Fully-associative (FA)	A memory value can be placed in **any location** in the cache	Extensive hardware resources required to search (CAM)

- DM and FA can be thought as special cases of SA
  - DM → 1-way SA
  - FA → All-way SA

Cache Organization Example



Cache Organization Tradeoff

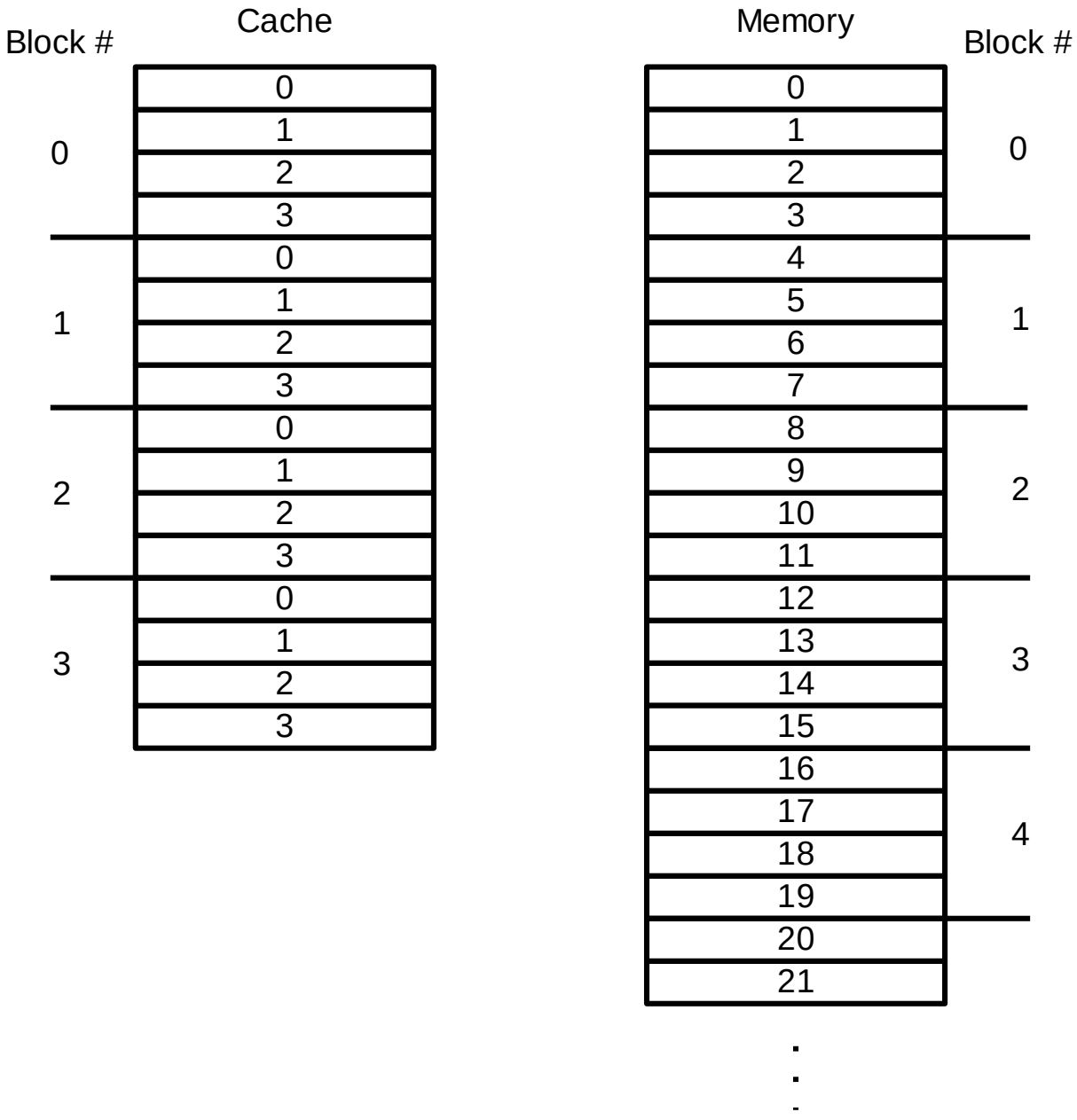
- For a given cache size, there is a tradeoff between hit rate and complexity
- If L = number of lines (blocks) in the cache,
  L = Cache Size / Block Size

How many places for a block to go	Name of cache type	Number of Sets
1	Direct Mapped	L
n	n-way associative	L/n
L	Fully Associative	1

Number of comparators needed to compare tags

An Example

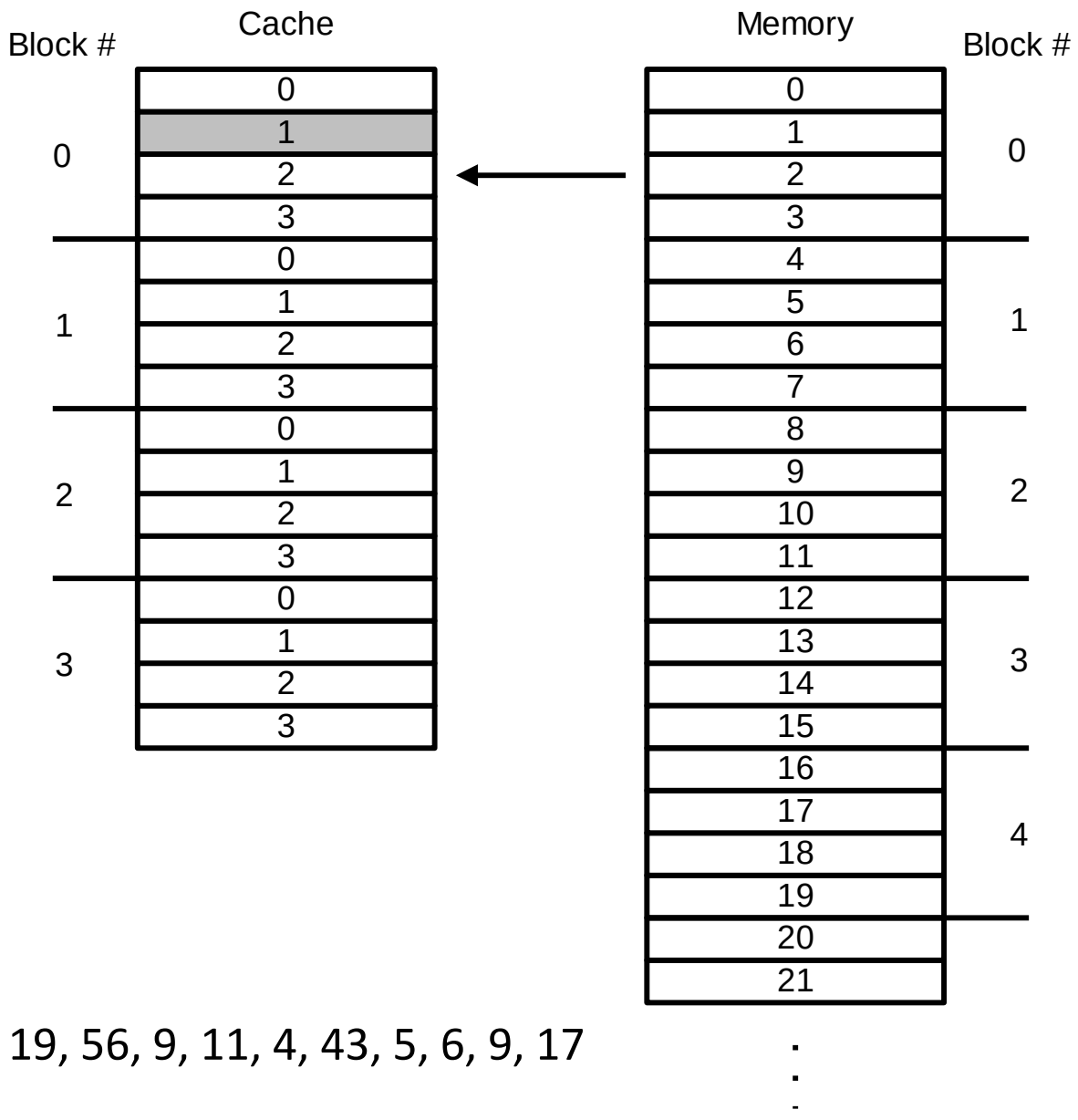
- Assume a direct mapped cache with 4-word blocks and a total size of 16 words.
- Consider the following string of address references given as word addresses:
  - 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17
- Show the hits and misses and final cache contents.


Block #
Cache
0
1
2
3
0
1
2
3
0
1
2
3
0
1
2
3
Memory
Block #
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
0
1
2
3
4

Address 1: Miss, bring block 0 to cache

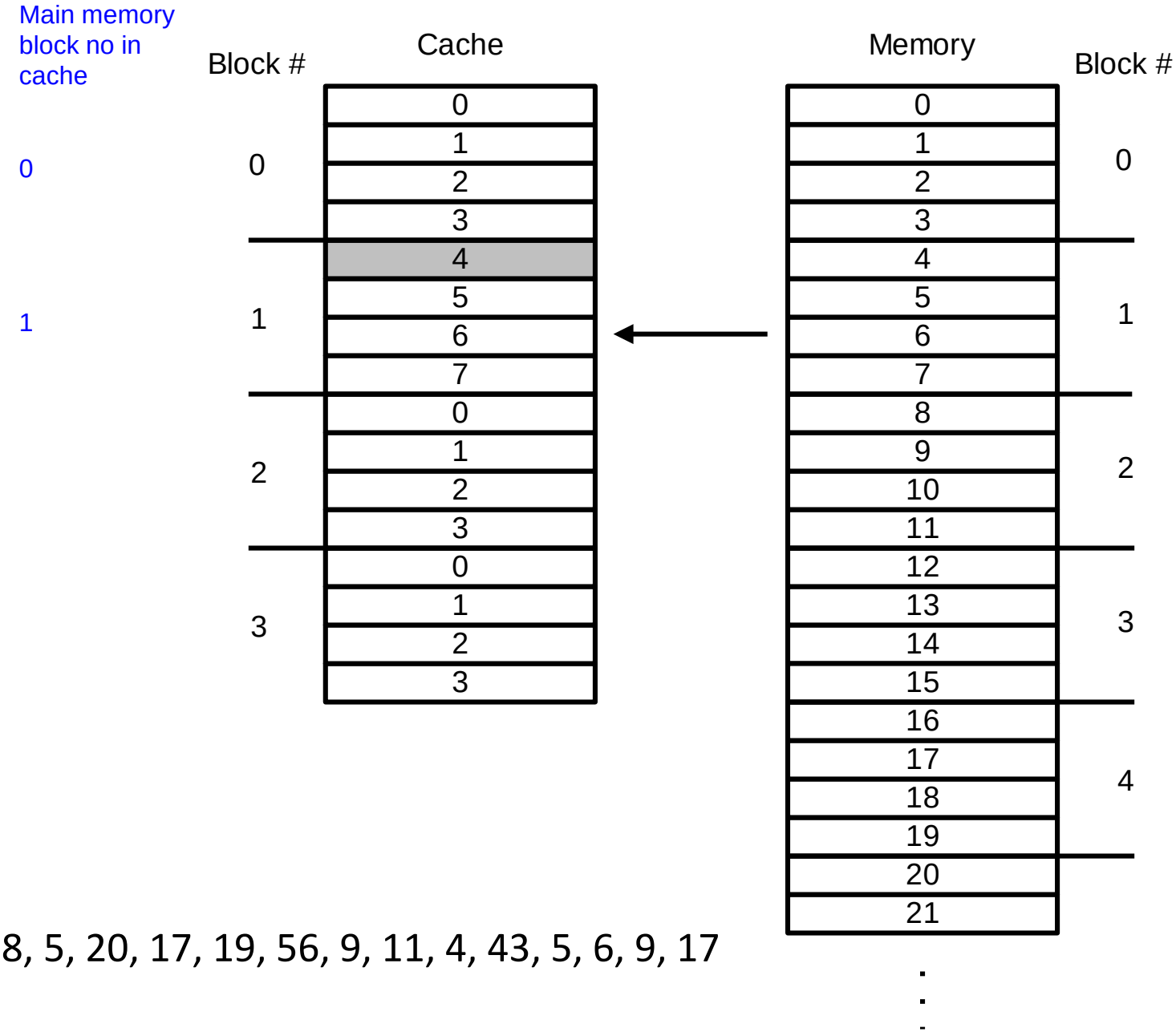
Main memory block no in cache

0



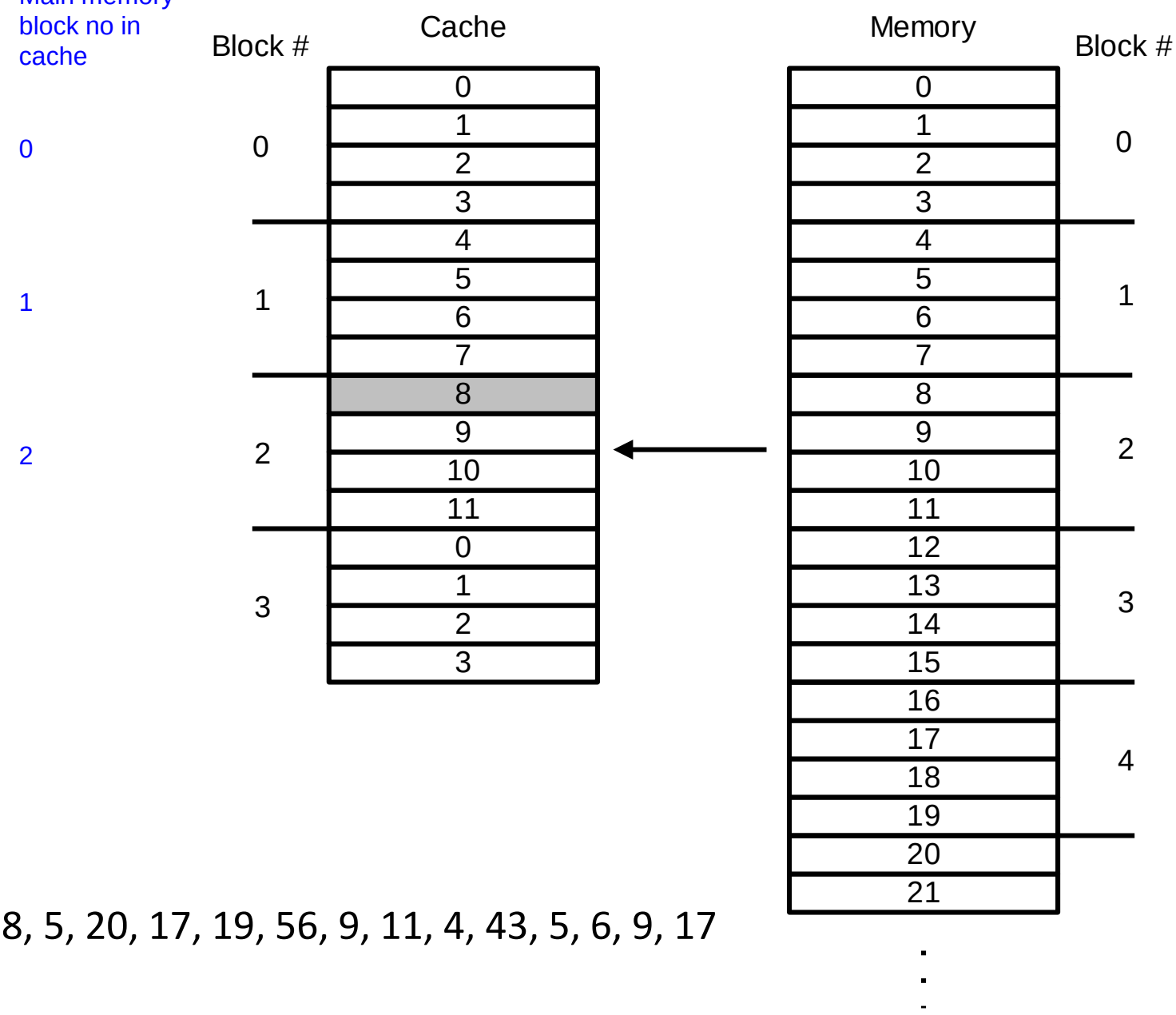
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 4: Miss, bring block 1 to cache



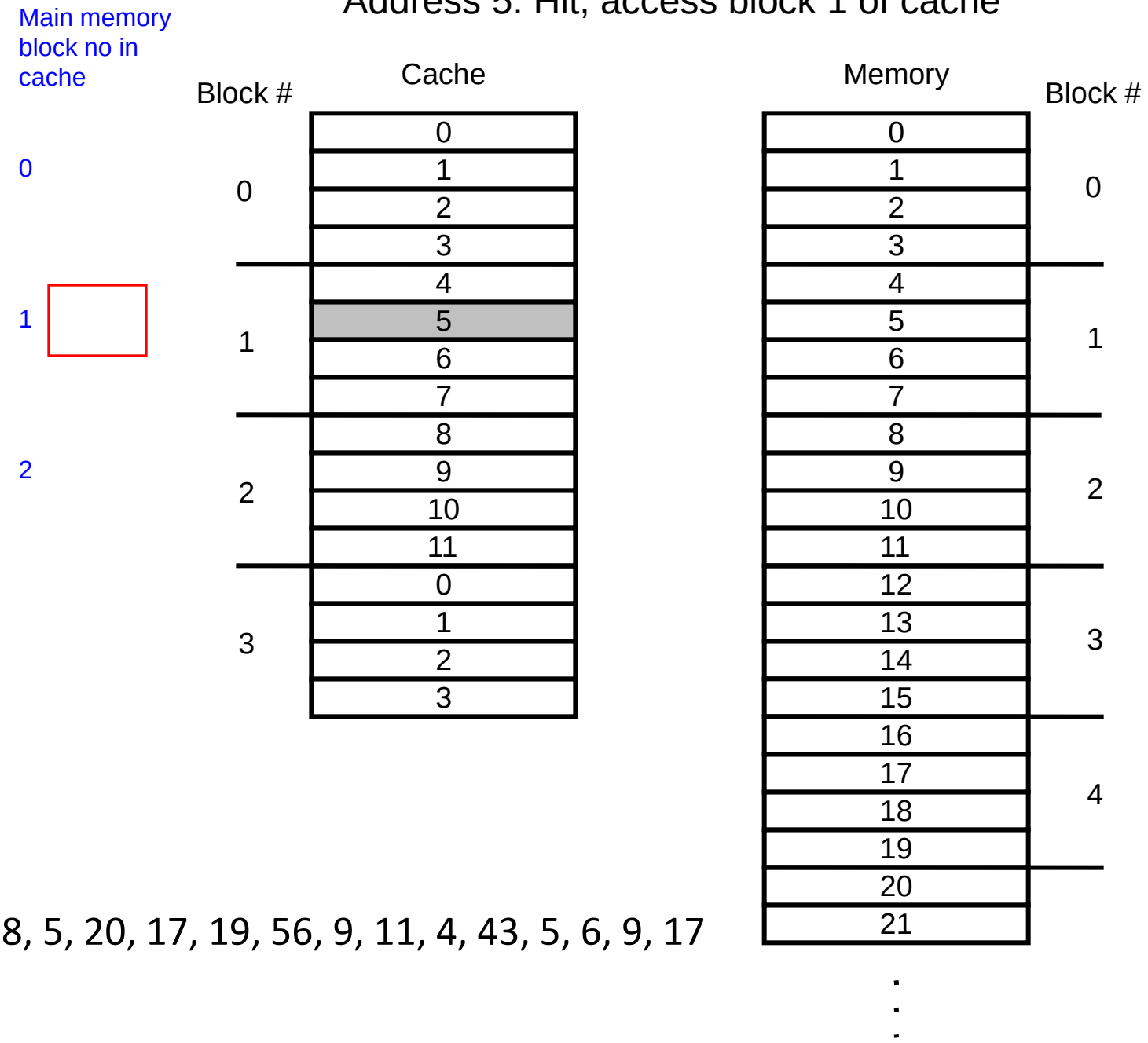
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 8: Miss, bring block 2 to cache



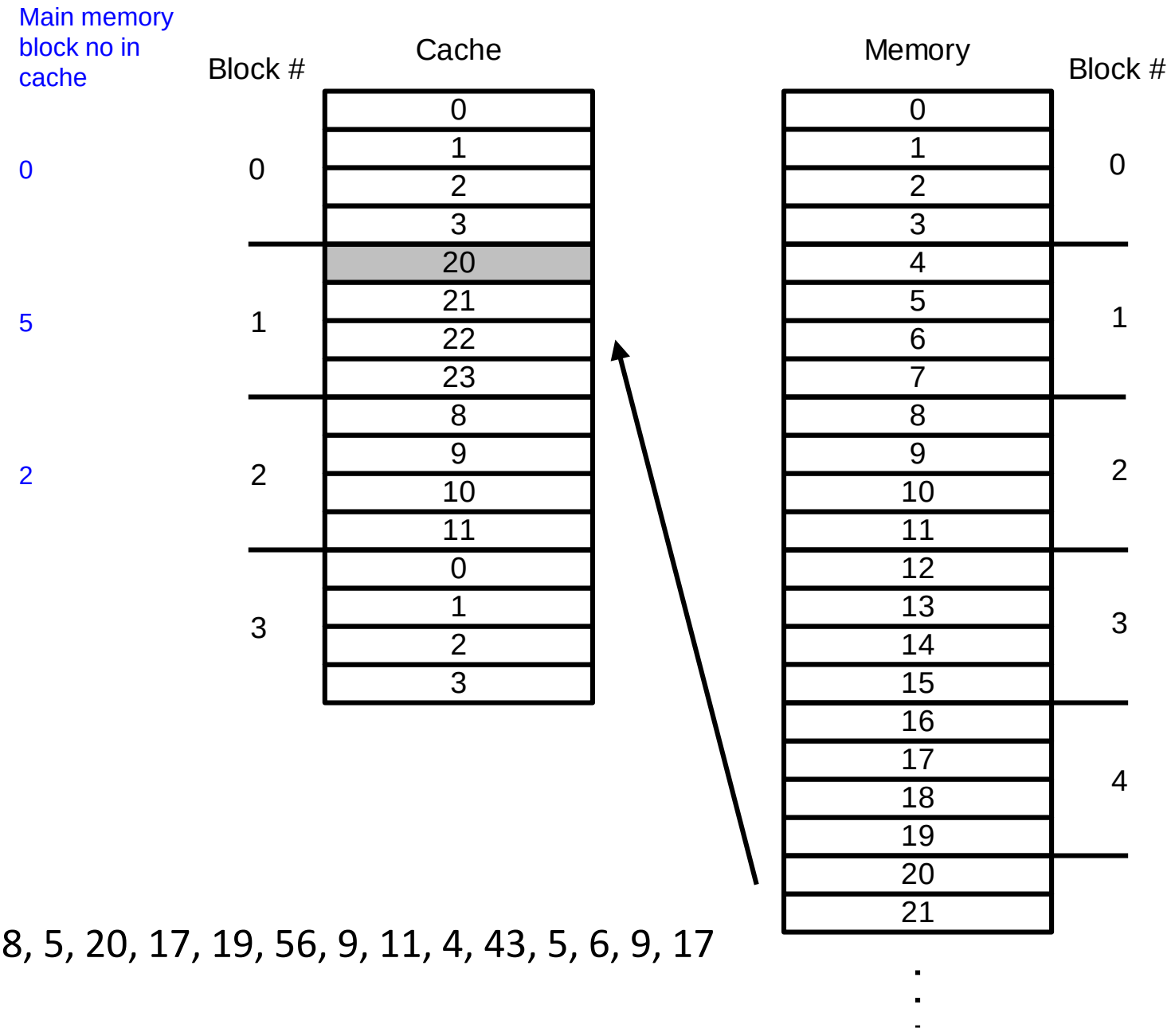
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 5: Hit, access block 1 of cache



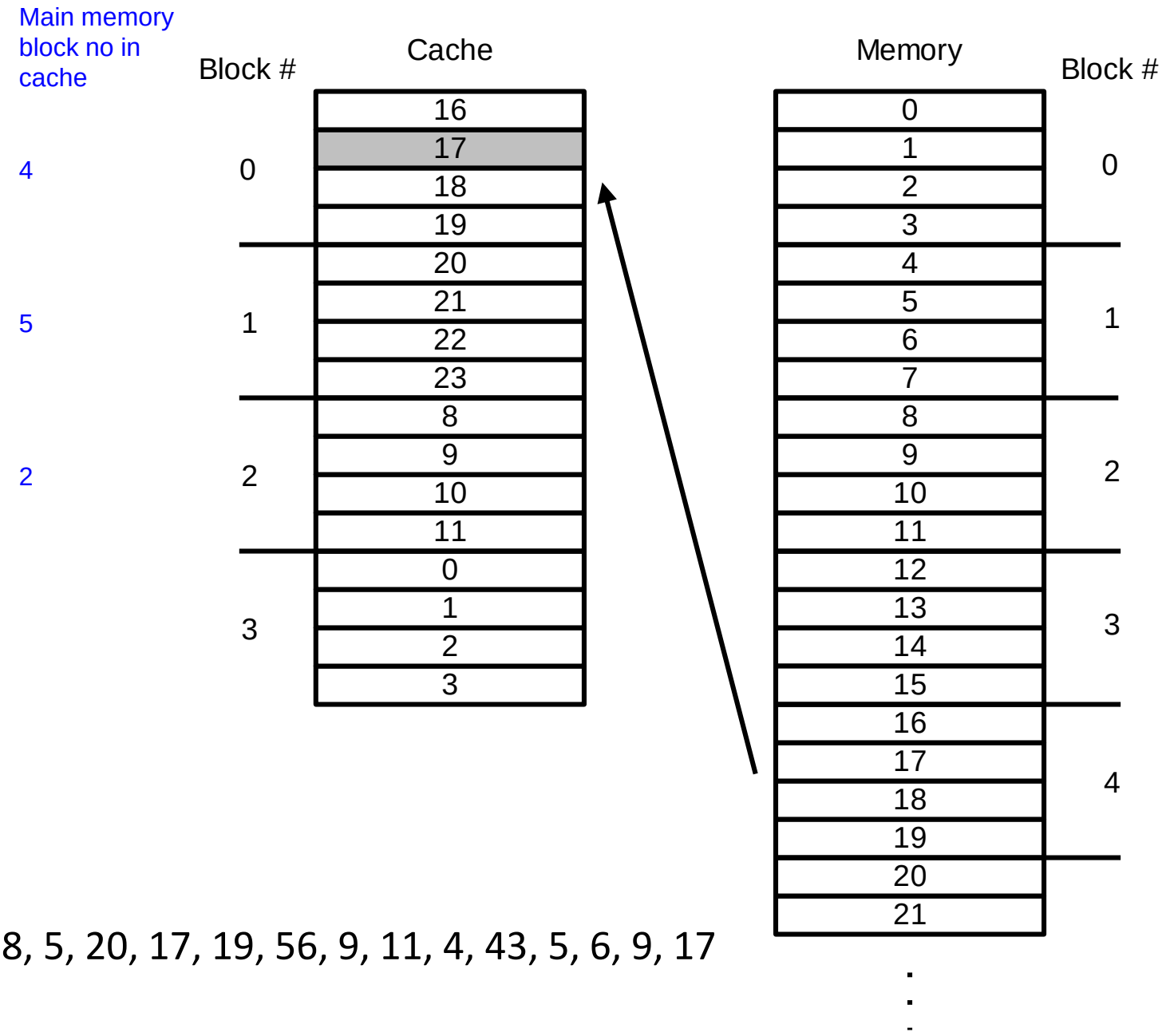
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 20: Miss, bring block 5 to cache block 1



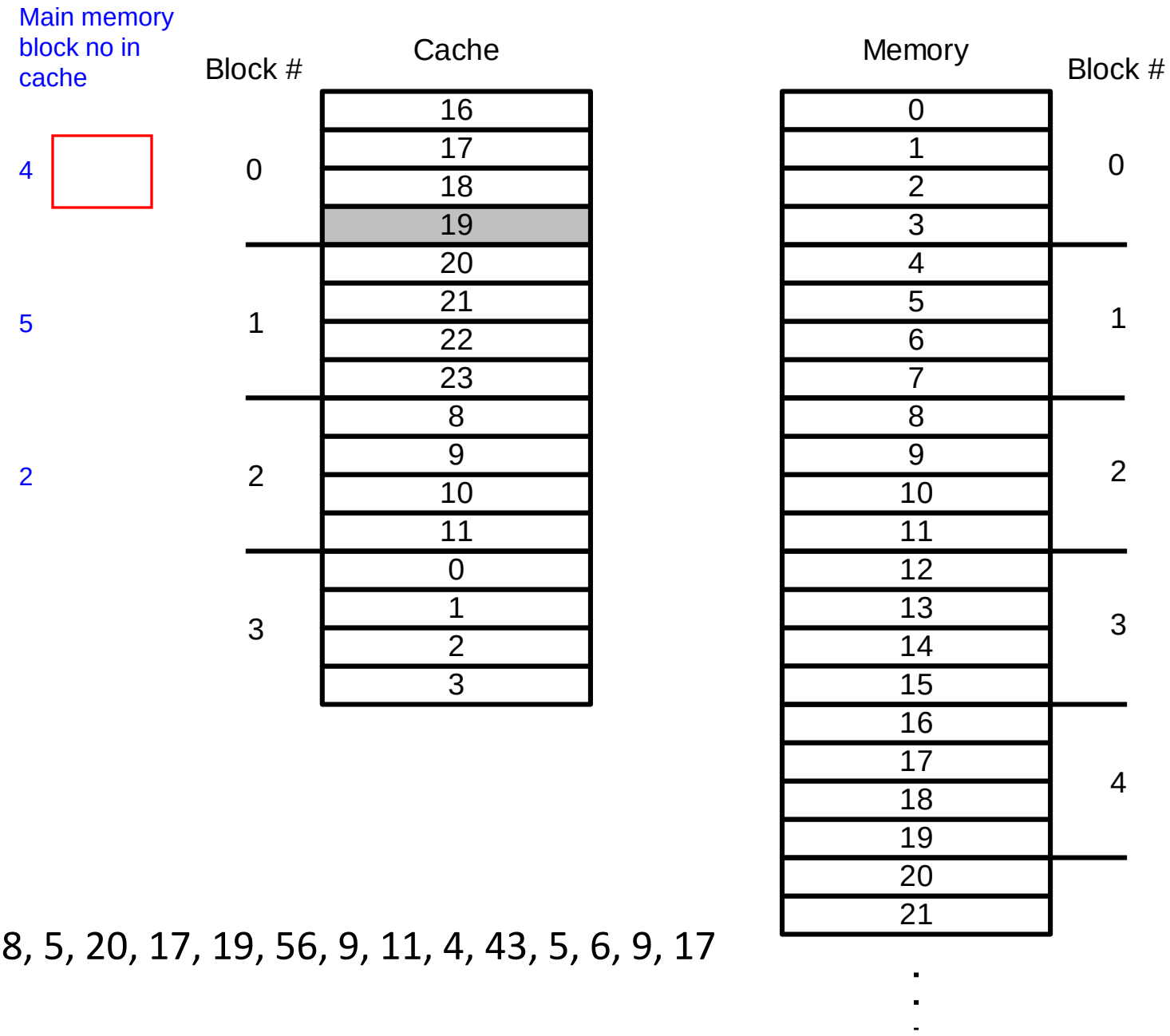
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 17: Miss, bring block 4 to cache block 0



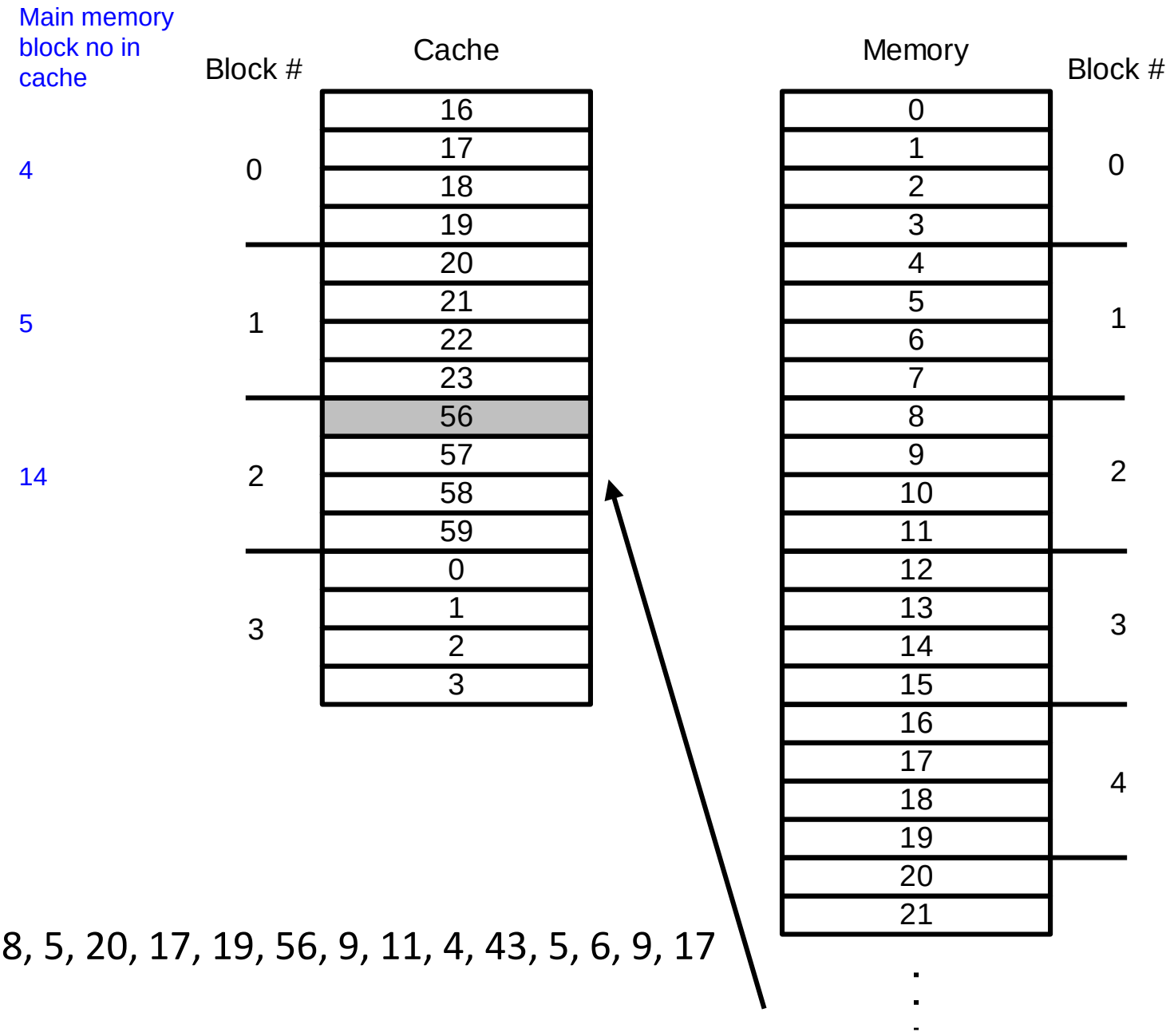
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 19: Hit, access block 0 of cache



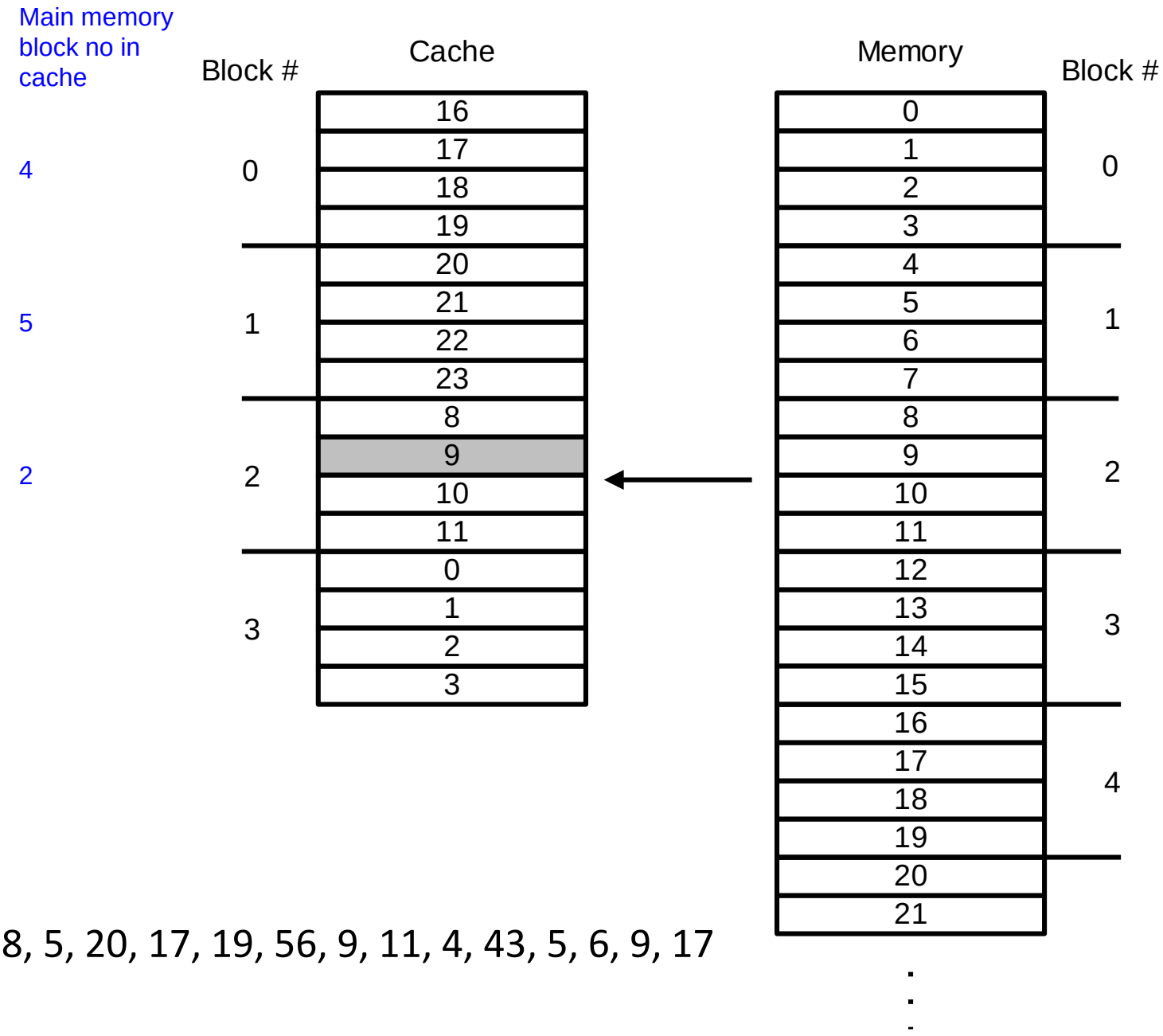
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 56: Miss, bring block 14 to cache block 2



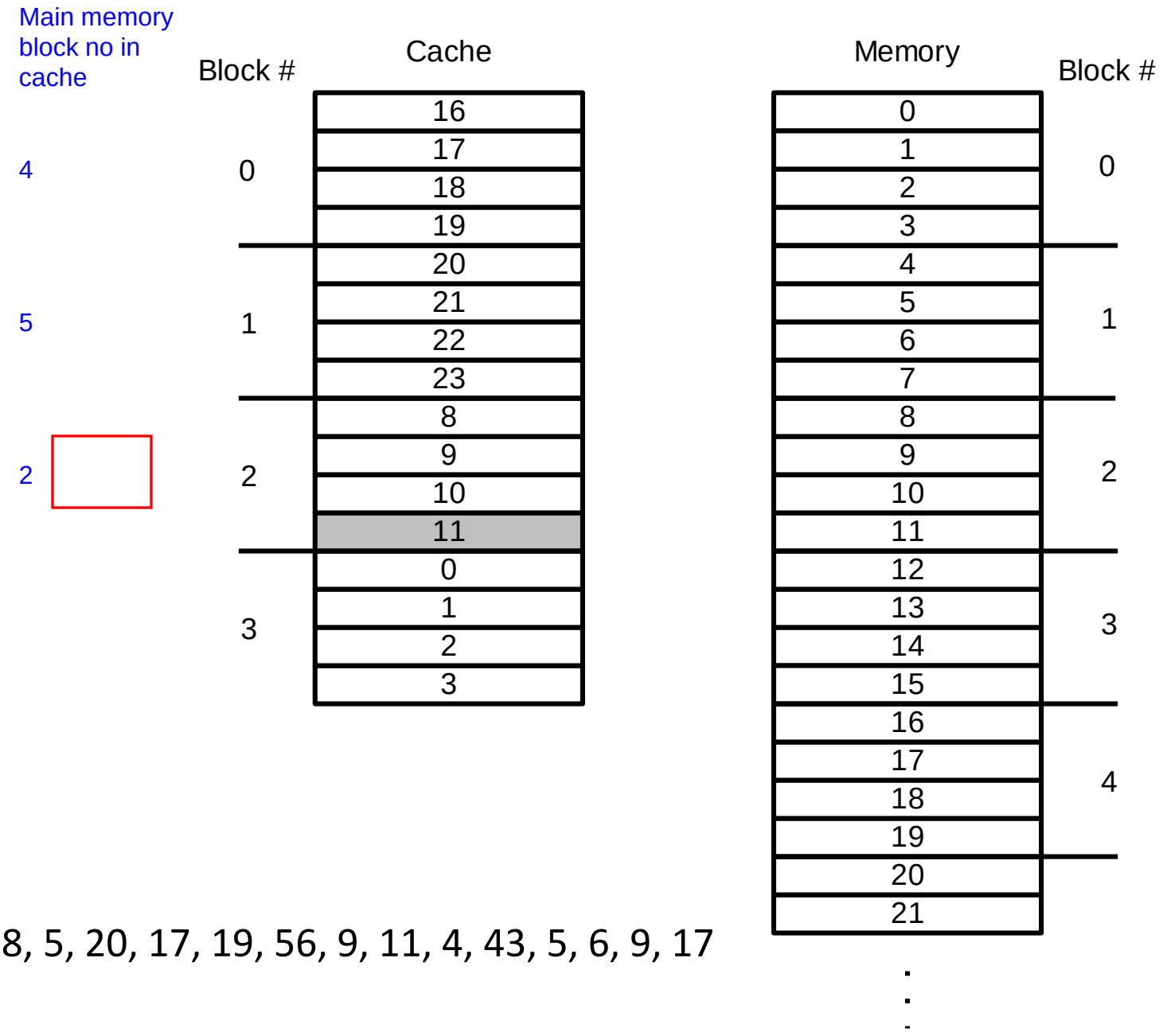
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 9: Miss, bring block 2 to cache block 2



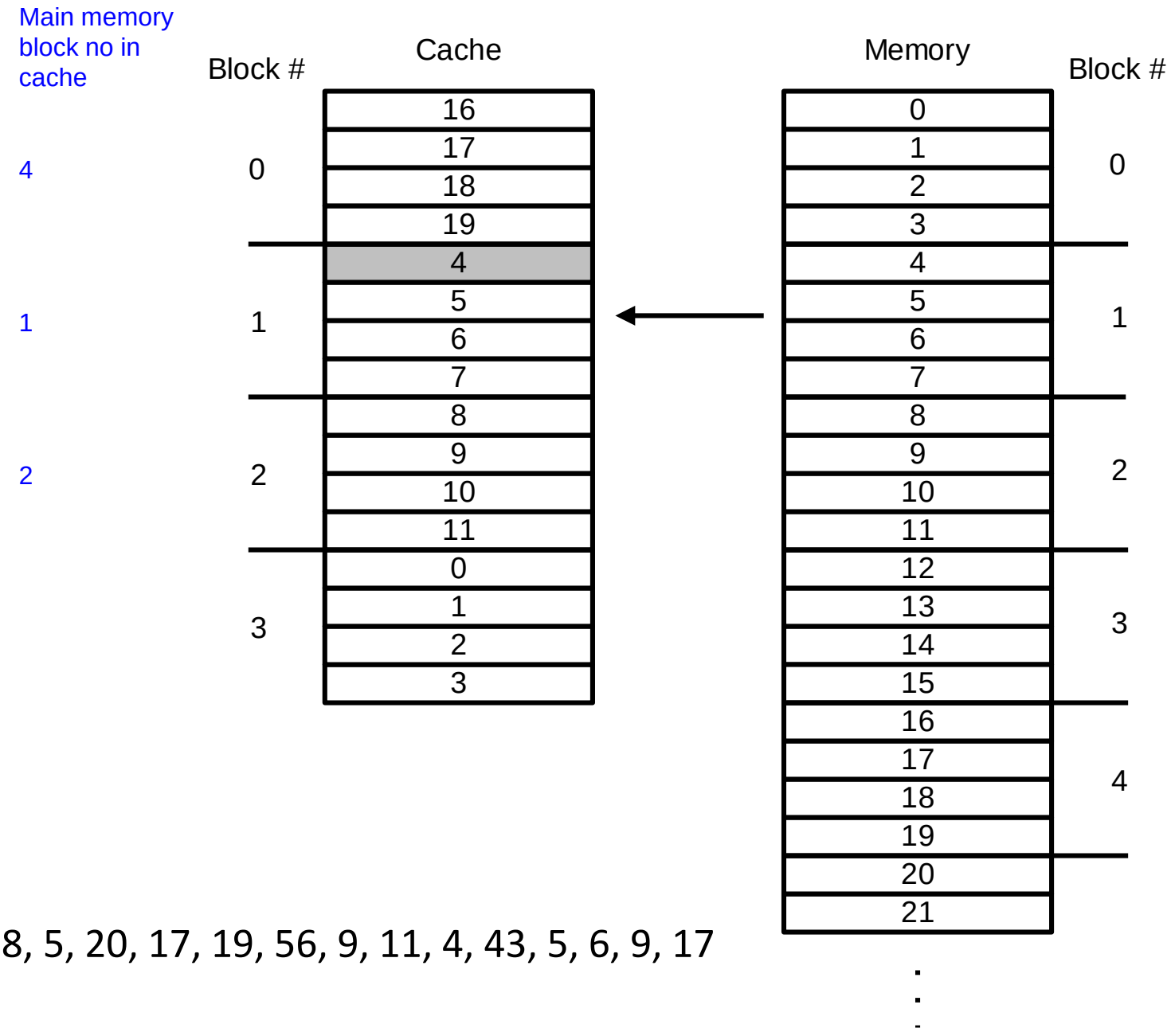
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 11: Hit, access block 2 of cache



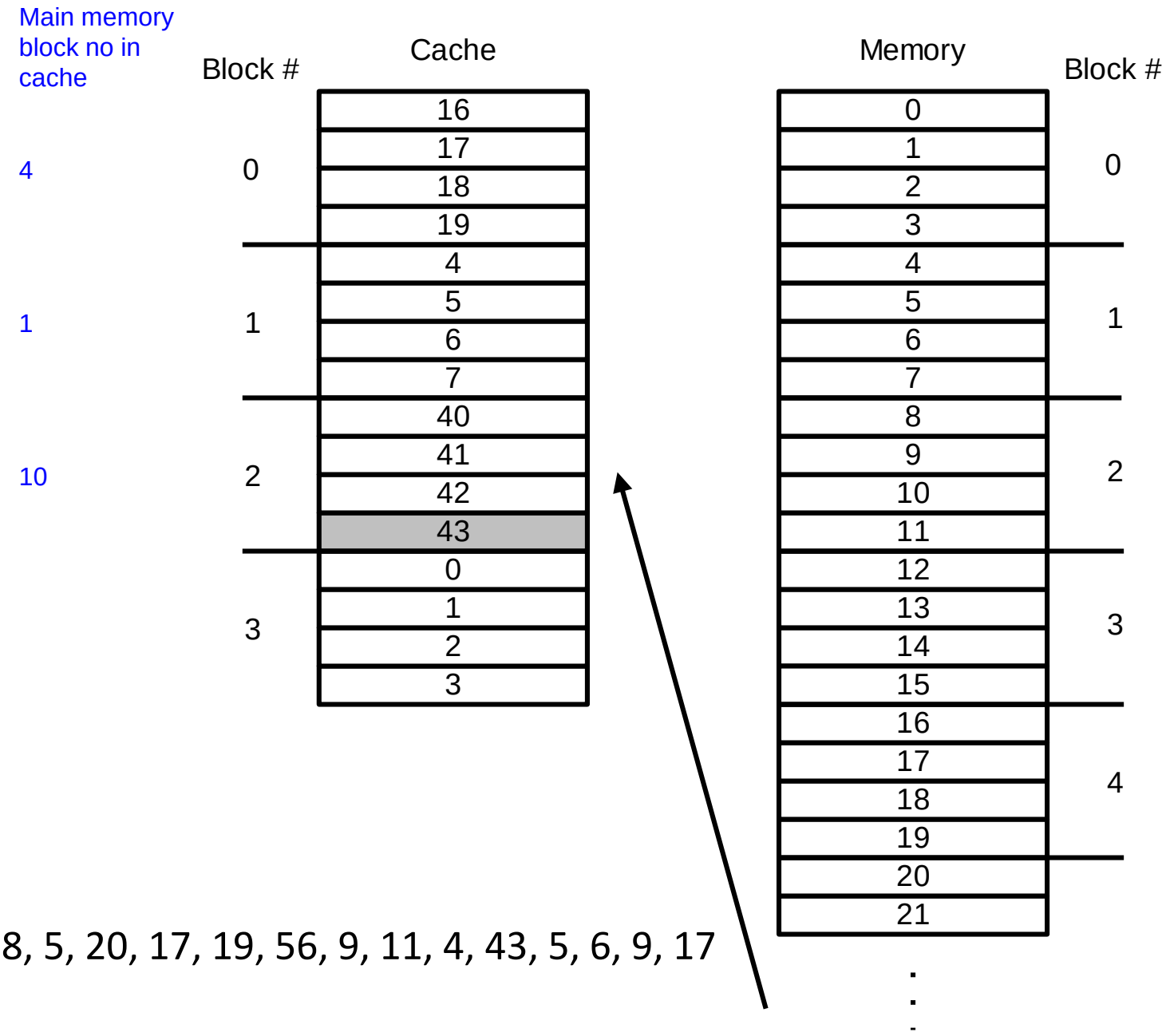
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 4: Miss, bring block 1 to cache block 1



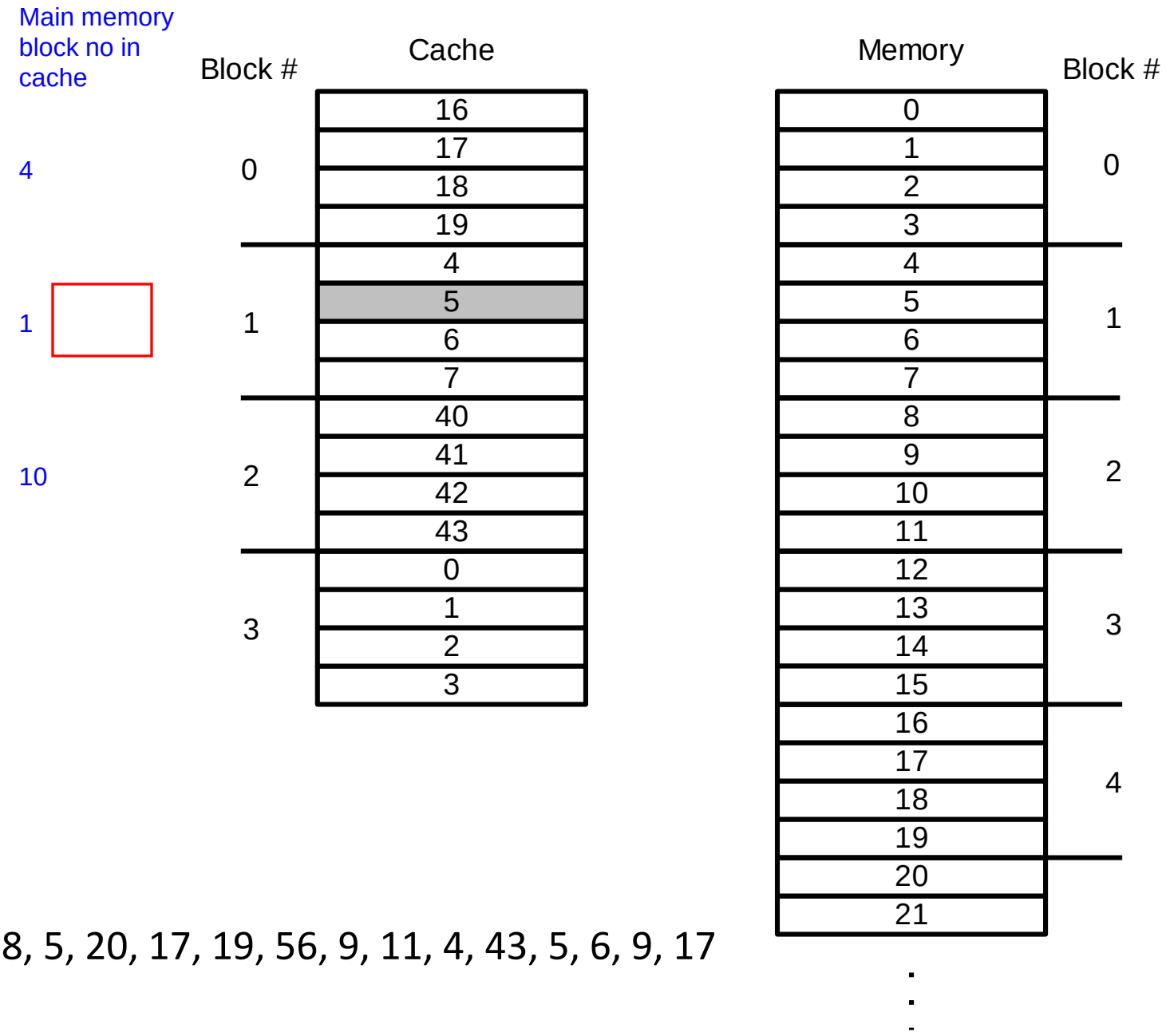
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 43: Miss, bring block 10 to cache block 2



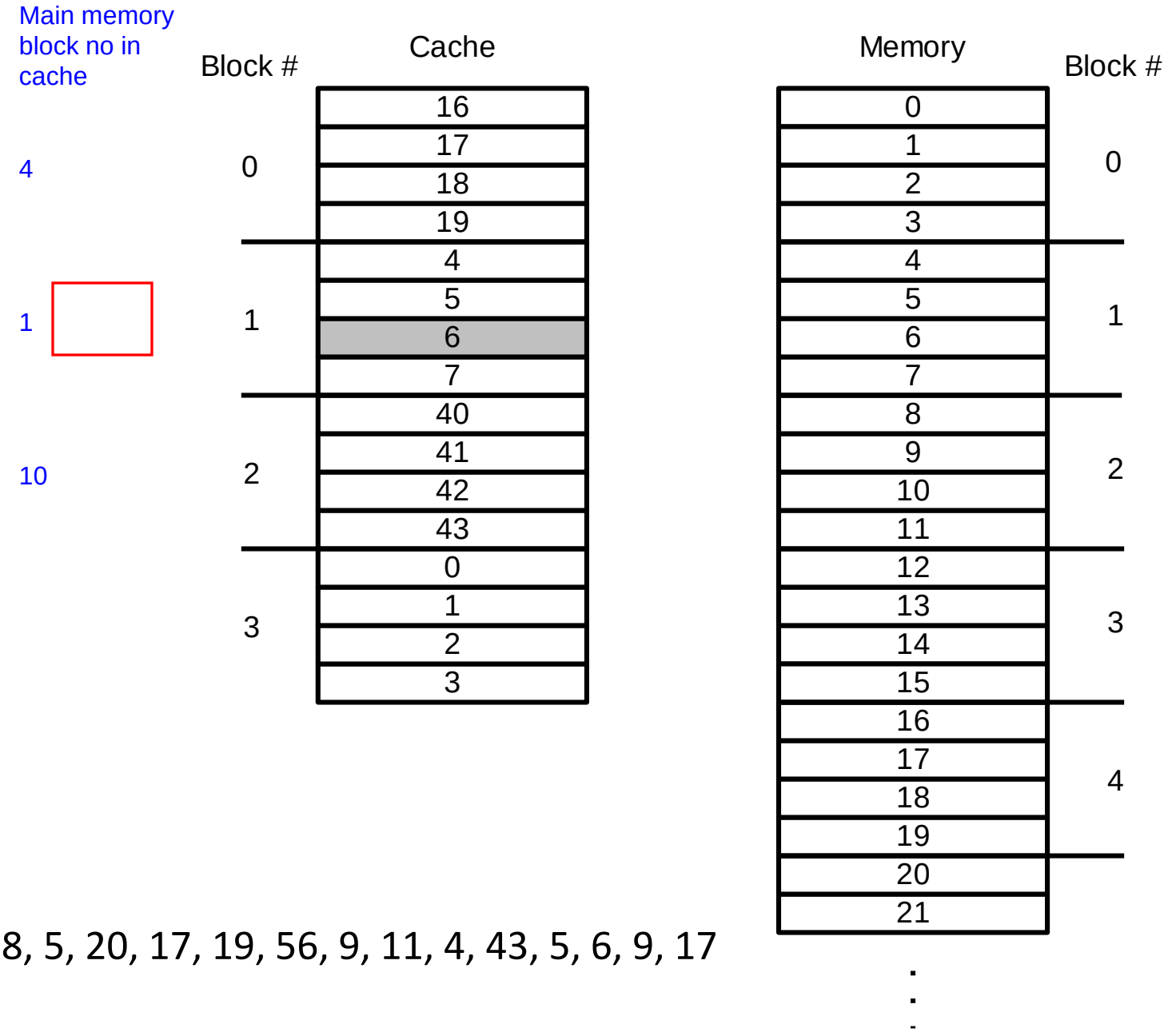
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 5: Hit, access block 1 of cache



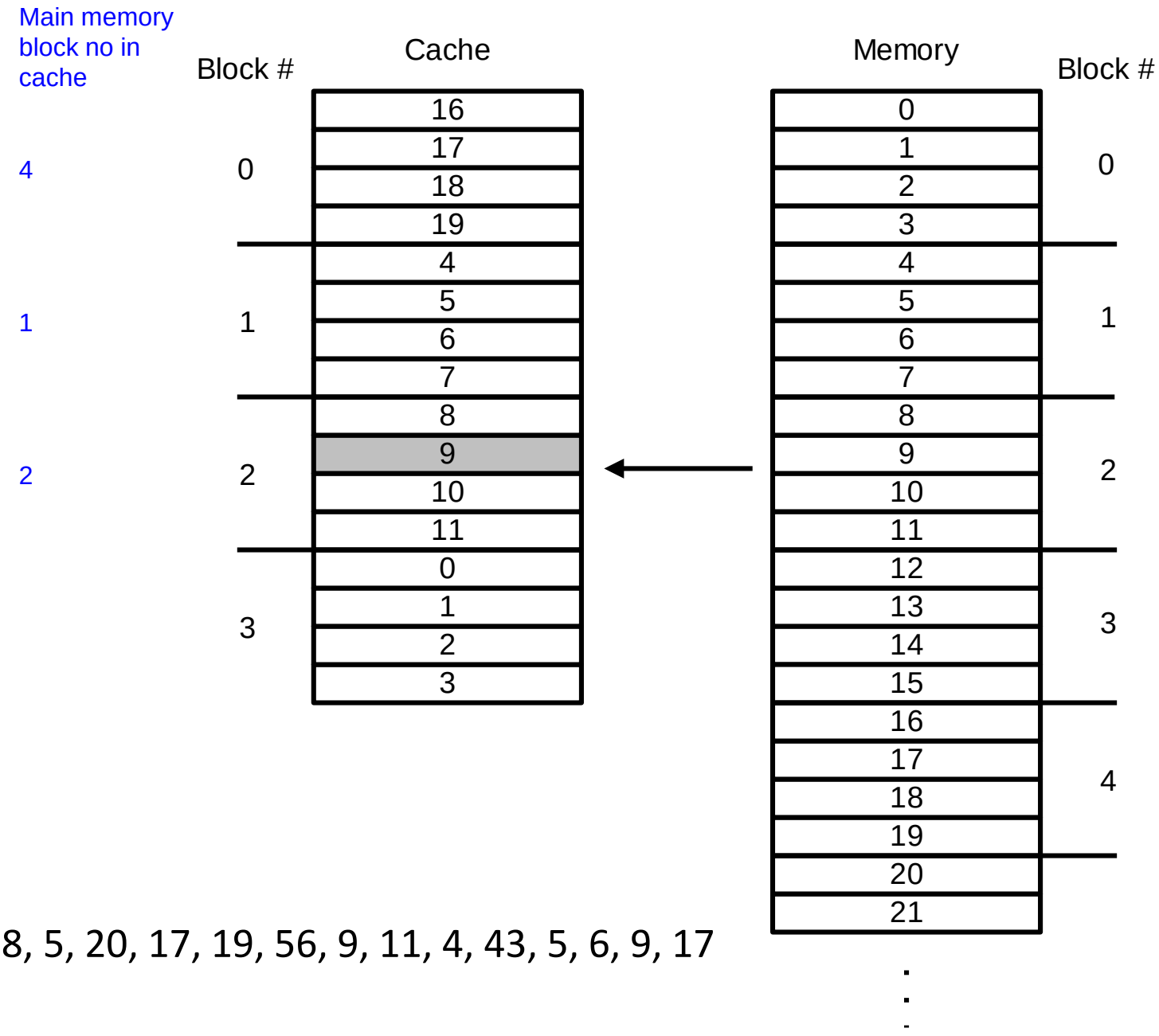
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 6: Hit, access block 1 of cache



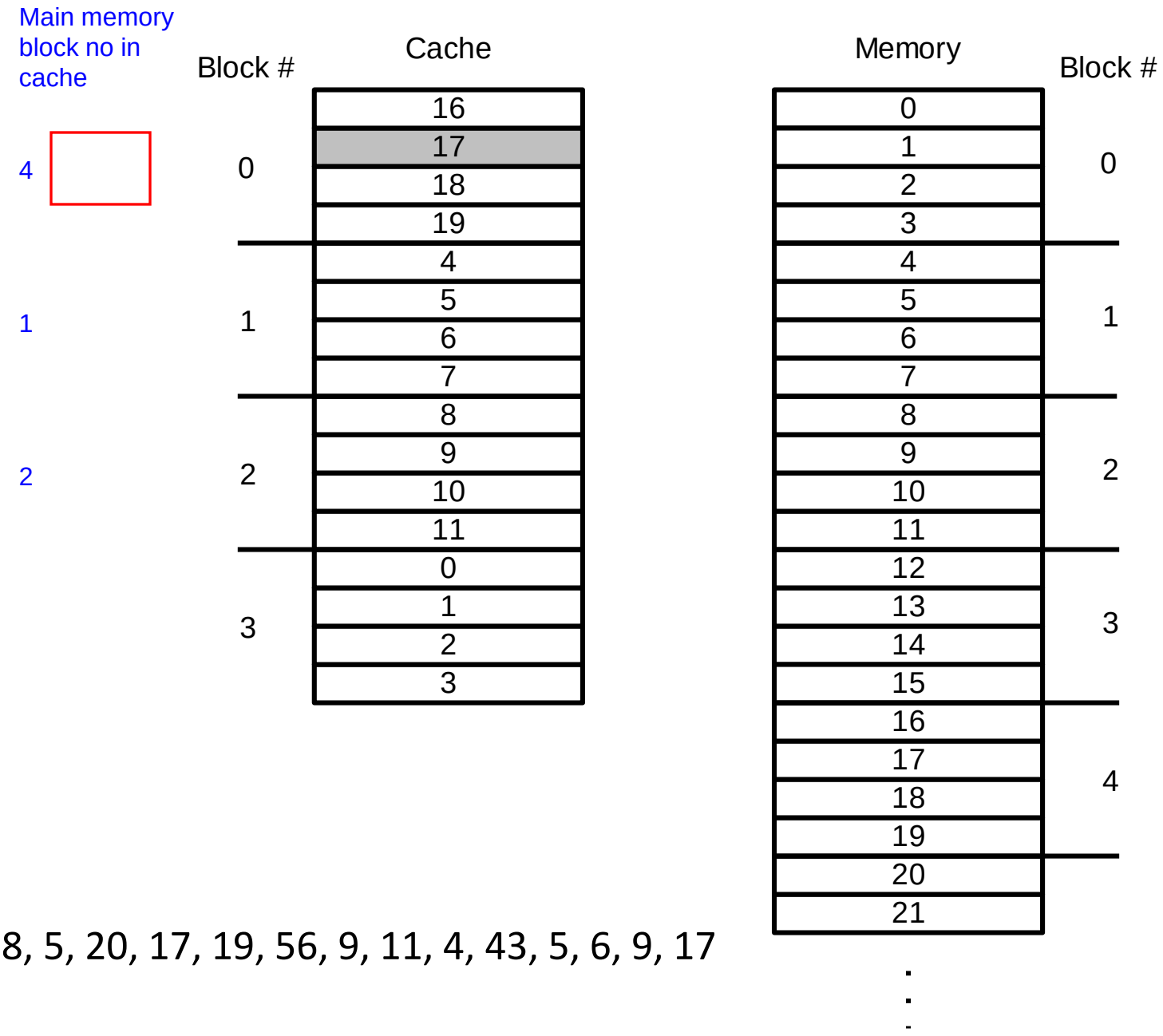
1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 9: Miss, bring block 2 to cache block 2



1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Address 17: Hit, access block 0 of cache



1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17

Summary

- Number of Hits = 6
- Number of Misses = 10

- Hit Ratio: 6/16 = 37.5% Unacceptable

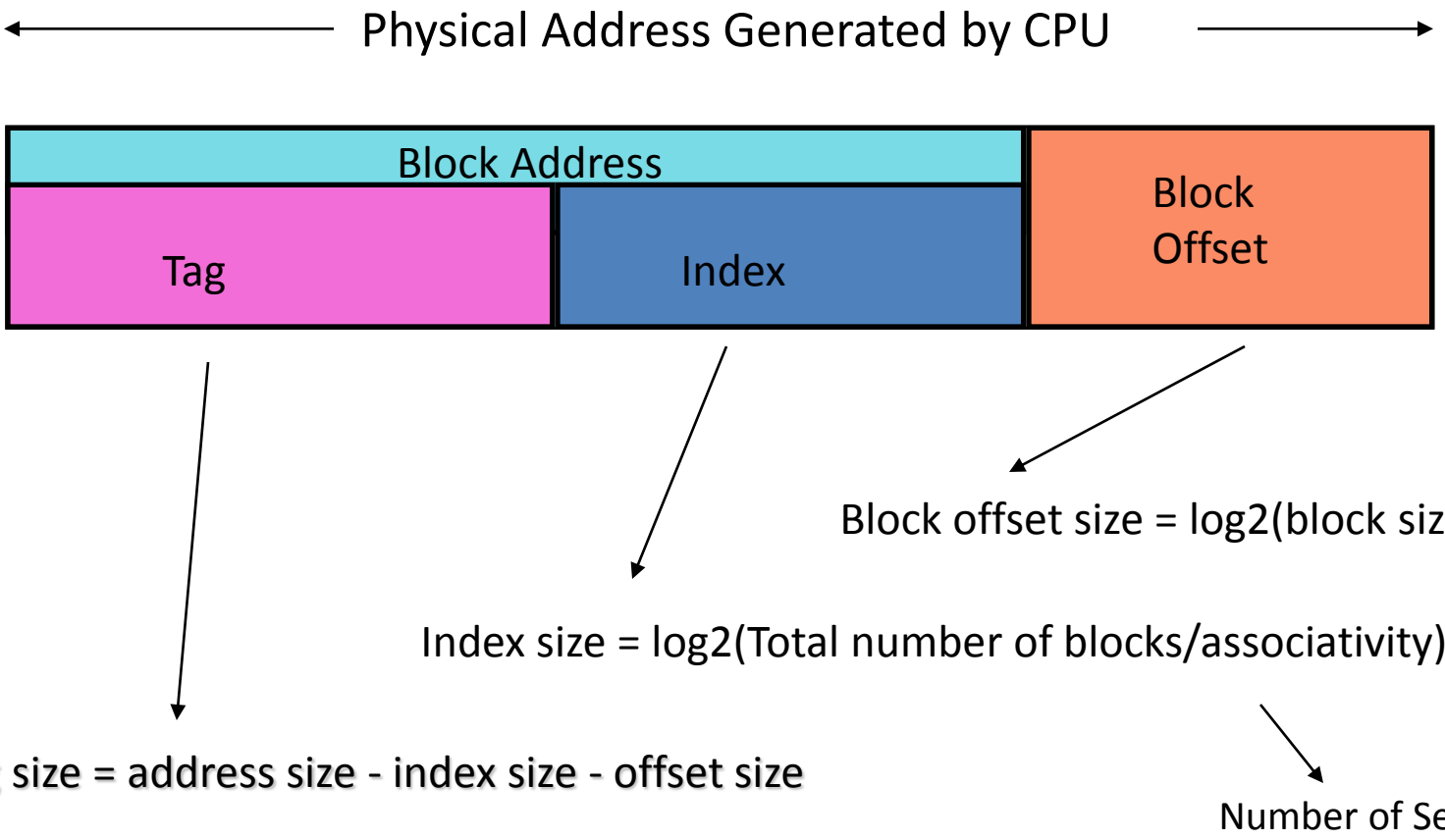
- Typical Hit ratio: > 90%

Address	Miss/Hit
1	Miss
4	Miss
8	Miss
5	Hit
20	Miss
17	Miss
19	Hit
56	Miss
9	Miss
11	Hit
4	Miss
43	Miss
5	Hit
6	Hit
9	Miss
17	Hit

Locating A Data Block in Cache

- Each block in the cache has an address tag.
- The tags of every cache block that might contain the required data are checked in parallel.
- A valid bit is added to the tag to indicate whether this cache entry is valid or not.
- The address from the CPU to the cache is divided into:
  - **A block address, further divided into:**
    - An index field to choose a block set in the cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - **A block offset to select the data from the block**.

Address Field Sizes



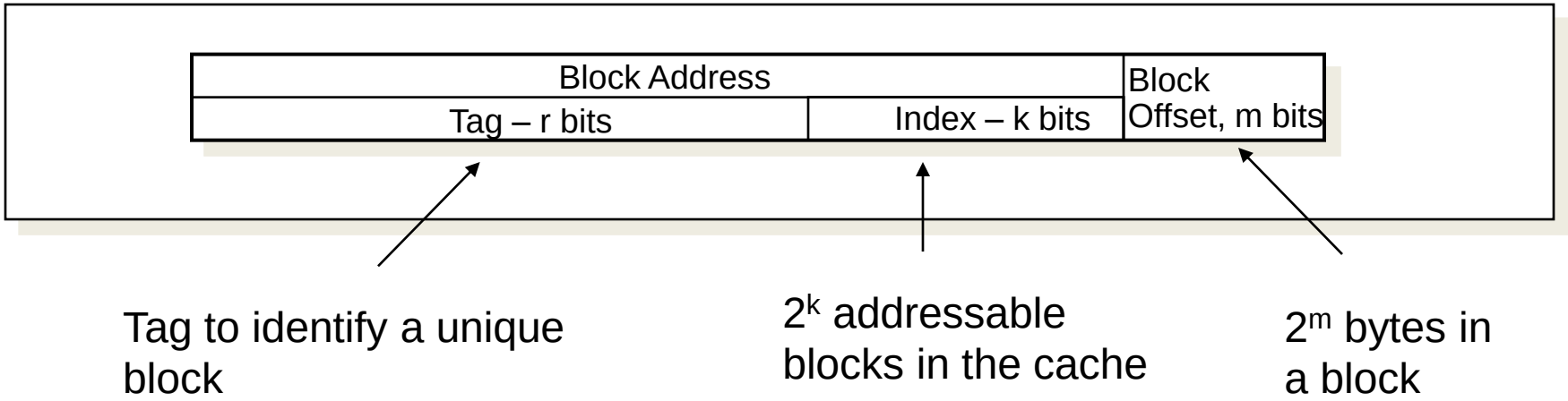
Mapping function:

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

Locating A Data Block in Cache

- Increasing associativity shrinks index, expands tag
  - Block index not needed for fully associative cache



Direct-Mapped Cache Example

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 4 words = 16 bytes = 2^4 bytes
  - need 4 bits to specify correct byte

Direct-Mapped Cache Example

- Index: (~index into an "array of blocks")
  - need to specify correct row in cache
  - cache contains 16 KB = 2^{14} bytes
  - block contains 2^4 bytes (4 words)

  # rows/cache =# blocks/cache (since there's one block/row)

  = $\frac{\text{bytes/cache}}{\text{bytes/row}}$

  = $\frac{2^{14}\ \text{bytes/cache}}{2^4\ \text{bytes/row}}$

  = 2^{10} rows/cache

  need 10 bits to specify this many rows

Direct-Mapped Cache Example

- Tag: use remaining bits as tag
  - tag length = mem addr length
    - offset
    - index
    = 32 - 4 - 10 bits
    = 18 bits
  - so tag is leftmost 18 bits of memory address

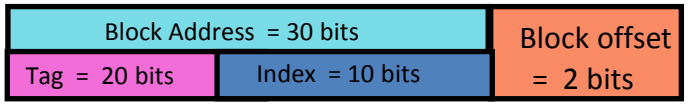
4KB Direct Mapped Cache Example

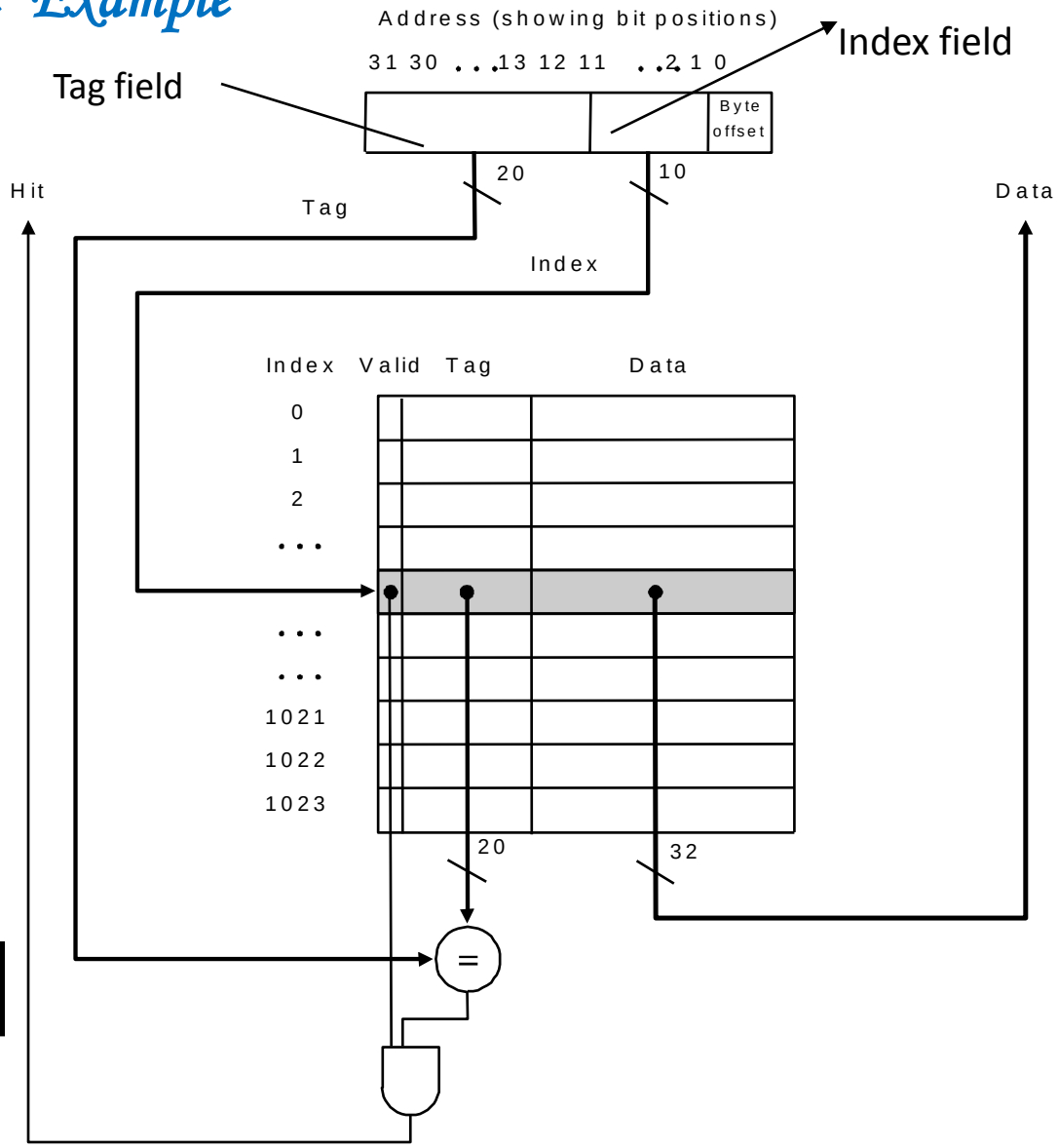
1K = 1024 Blocks
Each block = one 32b word

Cache for a
2^{32} bytes = 4 GB
memory space

Mapping function:

Cache Block frame number =
(Block address) MOD (1024)

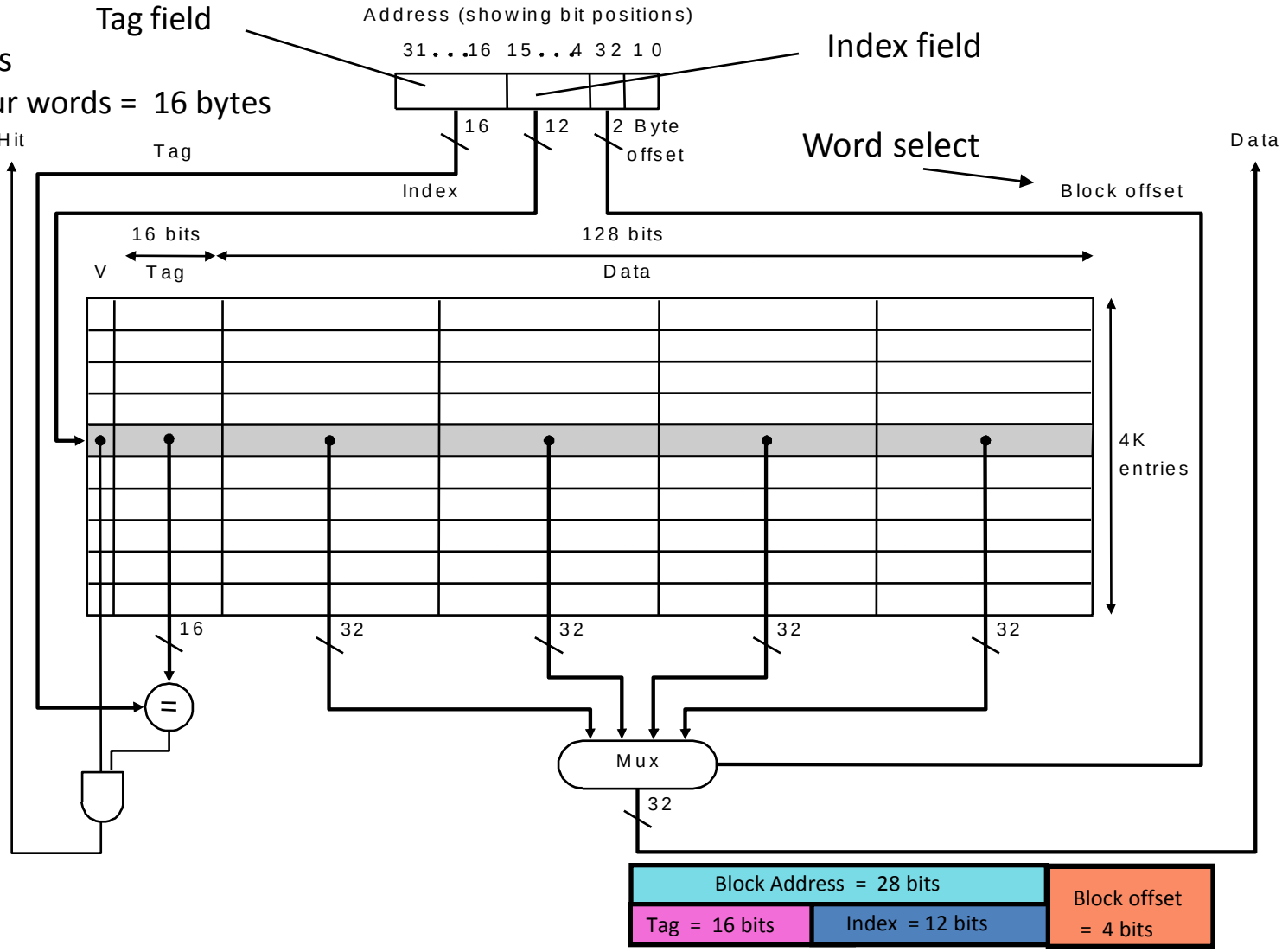




64KB Direct Mapped Cache Example

4K= 4096 blocks
Each block = four words = 16 bytes

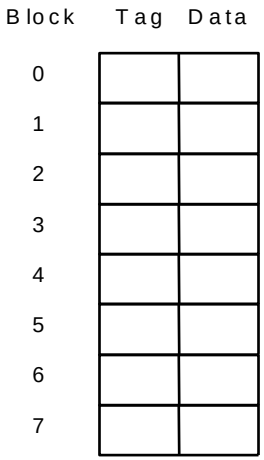
Can cache up to 2^{32} bytes = 4 GB of memory



Mapping Function: Cache Block frame number = (Block address) MOD (4096)
Larger blocks take better advantage of spatial locality

Cache Organization: Set Associativity

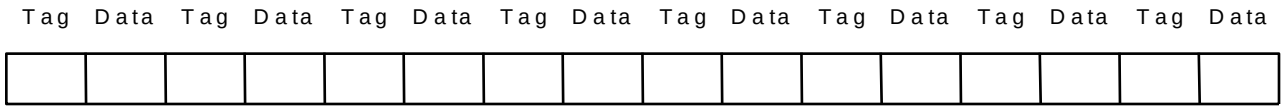
One-way set associative
(direct mapped)



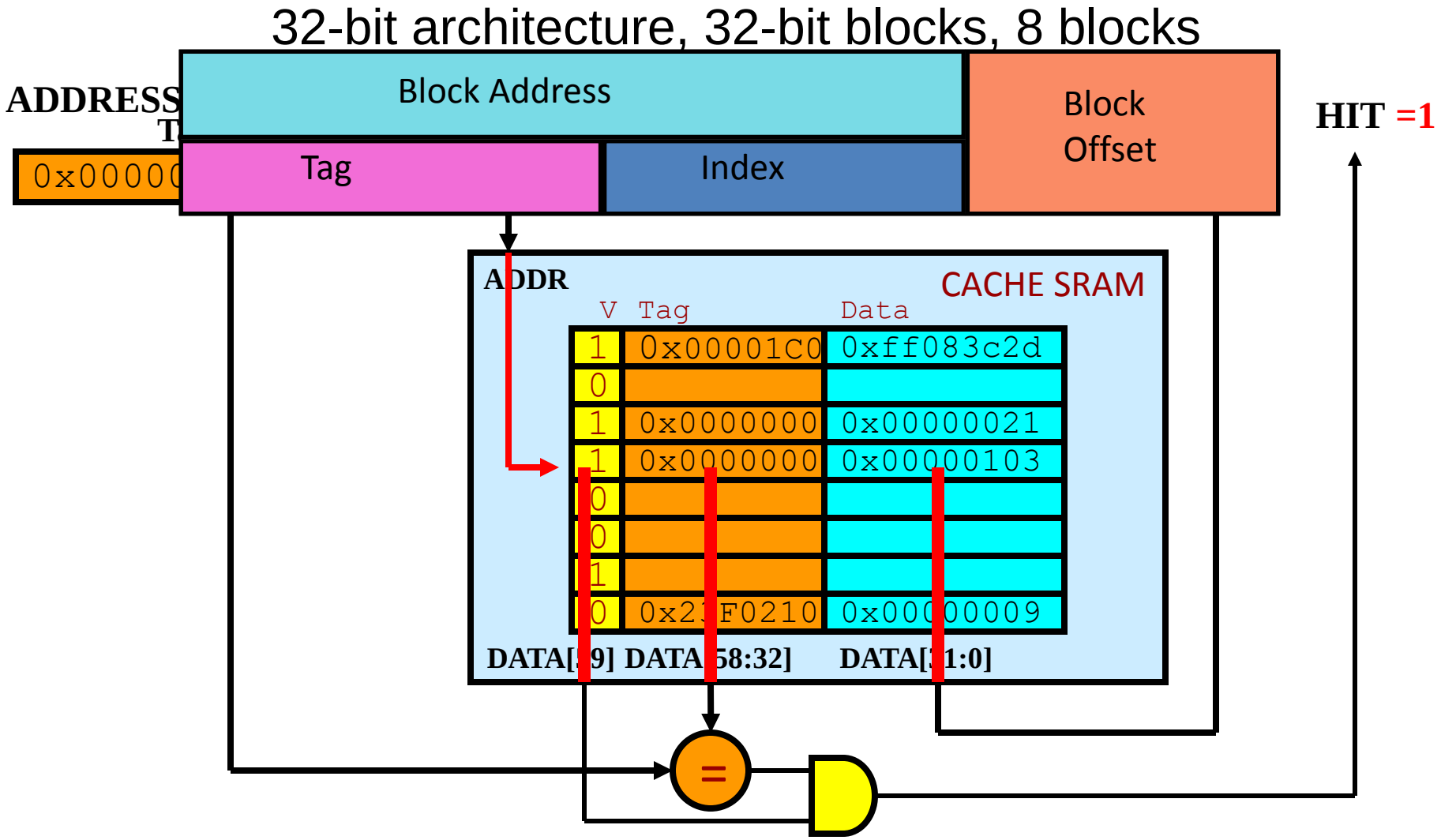
Two-way set associative

Four-way set associative

Eight-way set associative (fully associative)



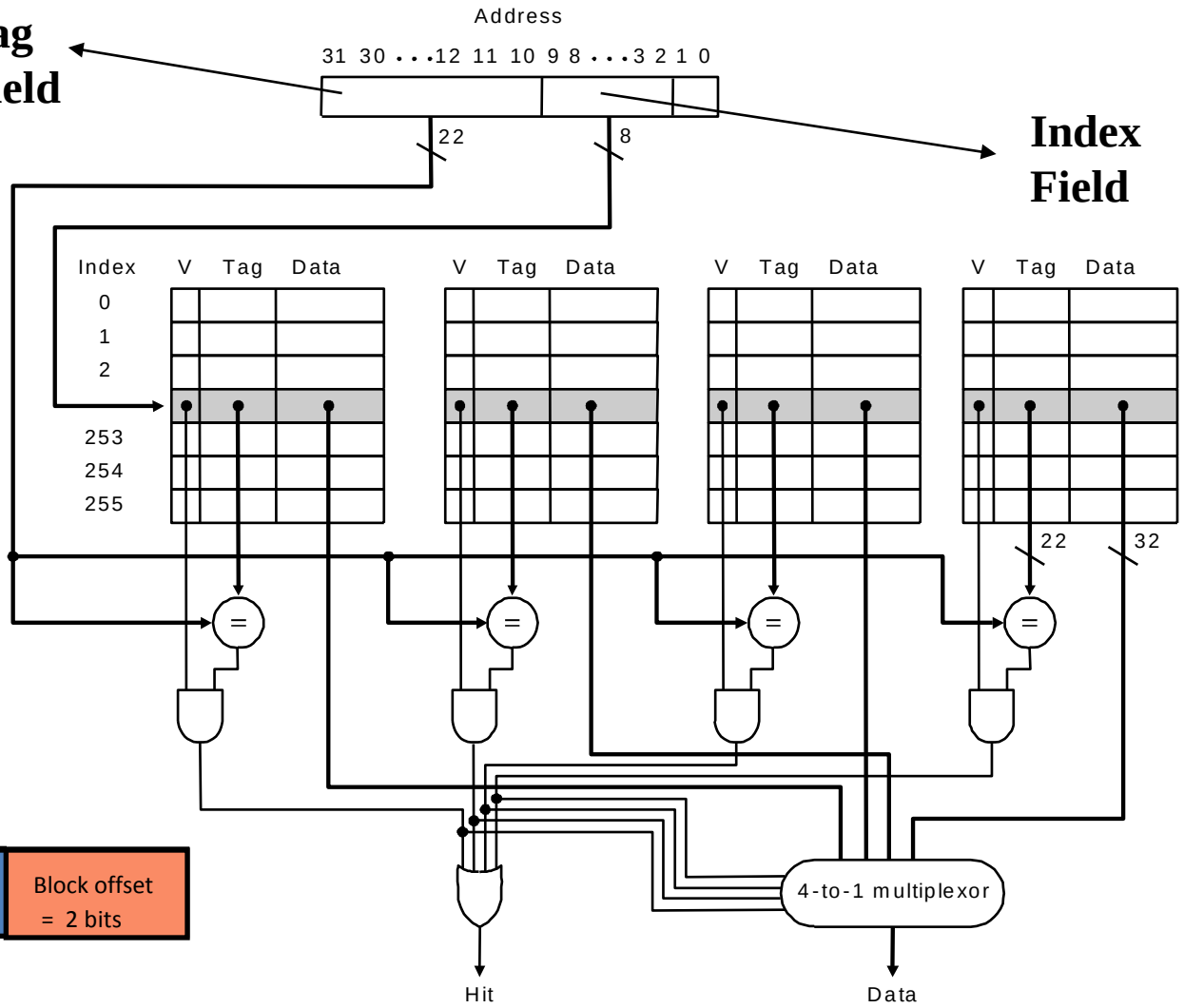
Direct-Mapped Cache Design

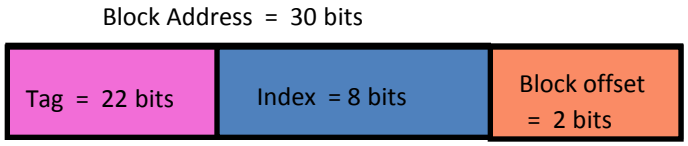


4KB Four-Way Set Associative Cache: MIPS Implementation Example

1024 block frames
1 block = one 32b word
4-way set associative
256 sets

Cache for a
2^{32} bytes = 4 GB
memory space

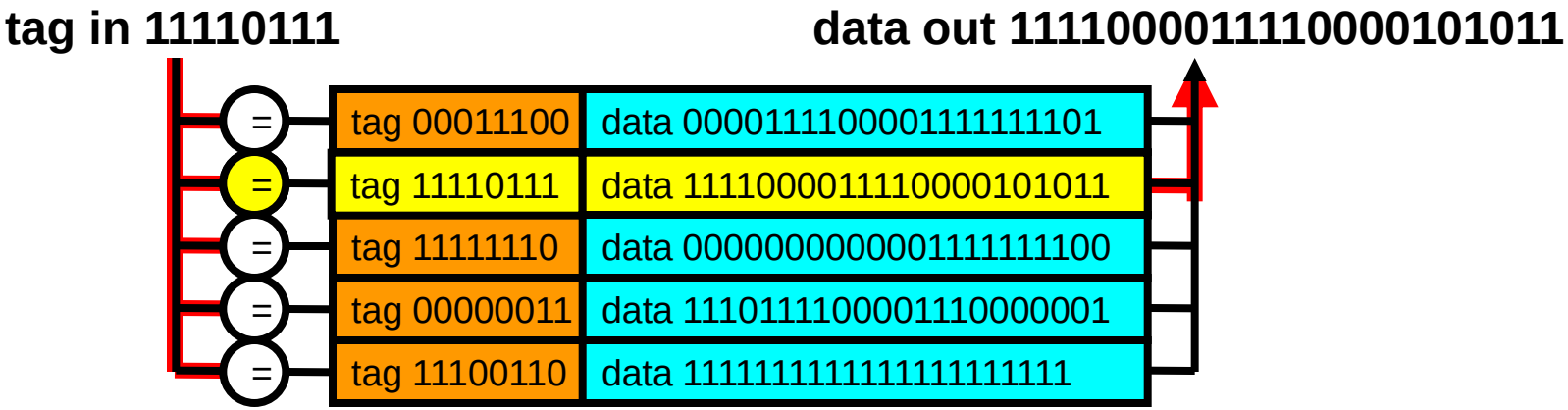




Mapping Function: **Cache Set Number = (Block address) MOD (256)**

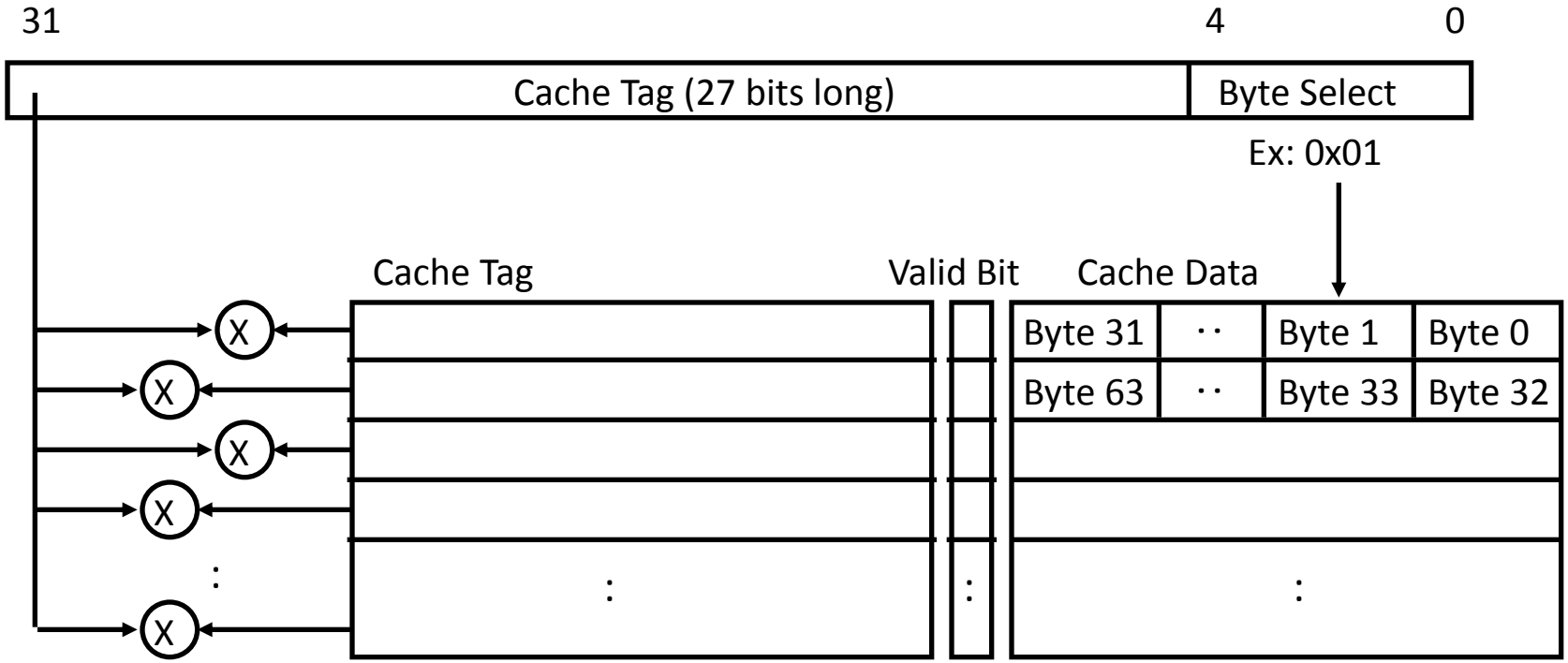
Fully Associative Cache Design

- Key idea: set size of one block
  - 1 comparator required for each block
  - No address decoding
  - Practical only for small caches due to hardware demands



Fully Associative

- Fully Associative Cache with N block frames
  - 0 bit for cache index
  - Compare the tags of all cache entries in parallel
  - Example: Block Size = 32 B blocks, we need N 27-bit comparators



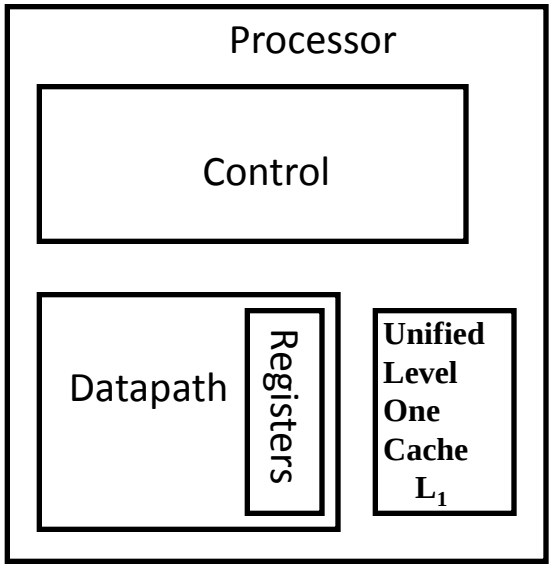
Unified vs. Separate Level 1 Cache

- Unified Level 1 Cache

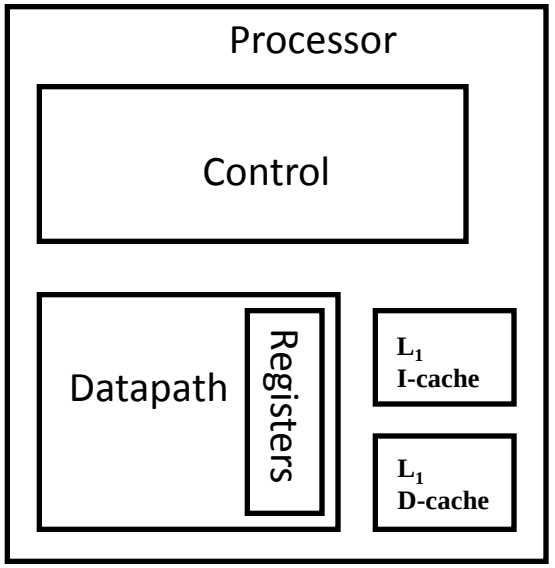
  A single level 1 cache is used for both instructions and data.

- Separate instruction/data Level 1 caches (Harvard Memory Architecture):

  The level 1 (L_1) cache is split into two caches, one for instructions (instruction cache, L_1 I-cache) and the other for data (data cache, L_1 D-cache).



Unified Level 1 Cache



Separate Level 1 Caches (Harvard Memory Architecture)

Why have separate caches?

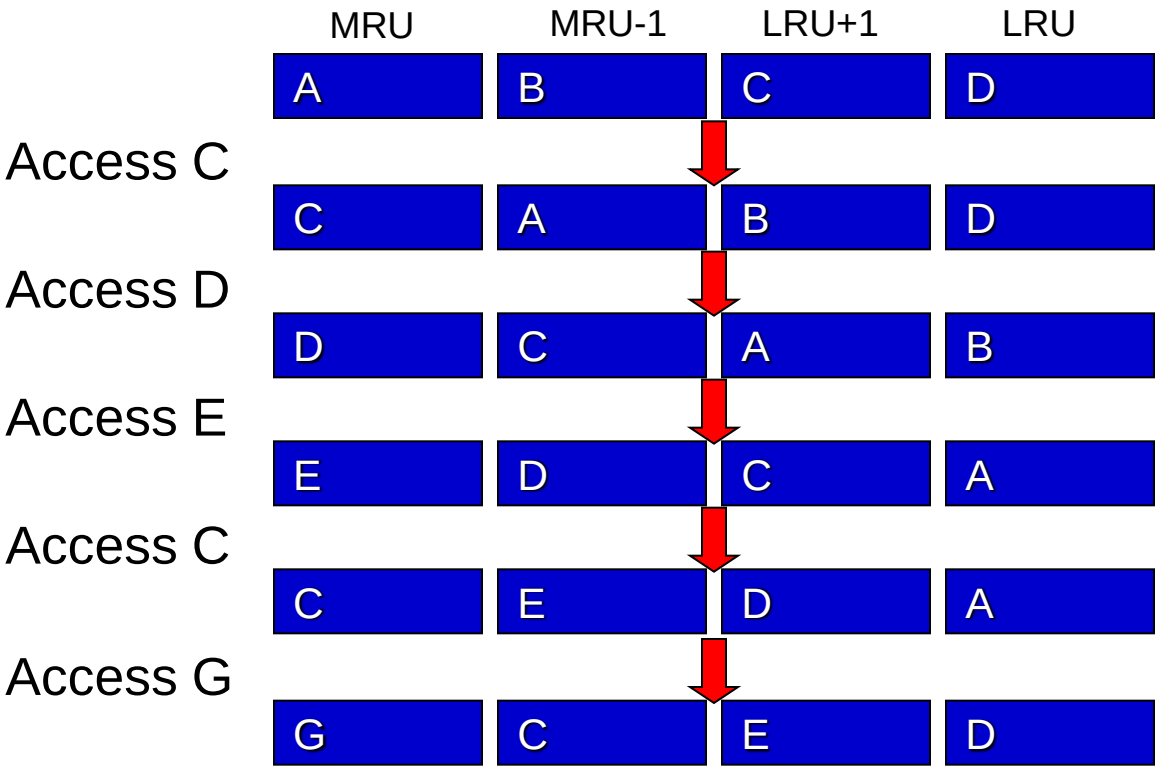
- Bandwidth: lets us access instructions and data in parallel (less structural hazards)
- Most programs don't modify their instructions
  - I-Cache can be simpler than D-Cache, since instruction references are never writes
- Instruction stream has high locality of reference, can get higher hit rates with small cache
  - Data references never interfere with instruction references

Cache Replacement Policy

When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is usually selected by one of two methods (for direct mapped cache, there is only one choice):

- **Random:**
  - Any block is randomly selected for replacement providing uniform allocation.
  - Simple to build in hardware.
  - The most widely used cache replacement strategy.
- **Least-recently used (LRU):**
  - Accesses to blocks are recorded and the block replaced is the one that was not used for the longest period of time.
  - LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated.

LRU Policy



MISS, replacement needed

MISS, replacement needed

Representative Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

Associativity:	**2-way**		**4-way**		**8-way**	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%