ECE 341

Lecture # 17

Instructor: Zeshan Chishti
zeshan@ece.pdx.edu

November 26, 2014

Portland State University

Lecture Topics

- The Memory System
  - Cache Memories
    - Cache Hits and Misses
    - Mapping Functions
      - Direct Mapping
      - Set-Associative Mapping
      - Fully-Associative Mapping
    - Replacement Algorithms

- <u>Reference:</u>
  - Chapter 8: Sections 8.6 and 8.7

Cache Hit

- When the processor needs to access some data, that data may or may not be found in the cache

- If the data is found in the cache, it is called a cache hit

- Read hit:
  - The processor reads data from cache and does not need to go to memory

- Write hit:
  - Cache has a replica of the contents of main memory, both cache and main memory need to be updated. Two options:
    - Update the contents of cache and main memory simultaneously. This is called write-through protocol
    - Update the contents of cache, and mark the cache block as updated by setting a bit known as dirty or modified bit. The main memory contents are updated when this block is evicted from cache. This is called write-back protocol

Cache Miss

- If the data is not found in the cache, it is called a cache miss
- Read miss:
  - The block containing the data is transferred from memory to cache
  - After the block is transferred, the desired data is forwarded to the processor.
  - The desired data may also be forwarded to the processor as soon as it is transferred without waiting for the entire cache block to be transferred. This is called load-through or critical word first
- Write-miss:
  - If *write-through* is used, then main memory contents are updated directly
  - If *write-back* is used, then the block containing the addressed word is first brought into the cache. The desired word is overwritten with new data

Cache Mapping and Replacement Functions

- Because a cache is smaller than main memory, only a subset of blocks in main memory can be held in the cache at a given time

- Mapping function:
    - The correspondence between main memory blocks and blocks in the cache is specified by a *mapping function*
    - It determines which memory addresses map to a given cache block

- Replacement Algorithm:
    - When cache is full, and a new block needs to be brought into the cache, another block needs to be removed from cache to create room for new block
    - The set of rules used to decide which block to remove constitutes a replacement algorithm

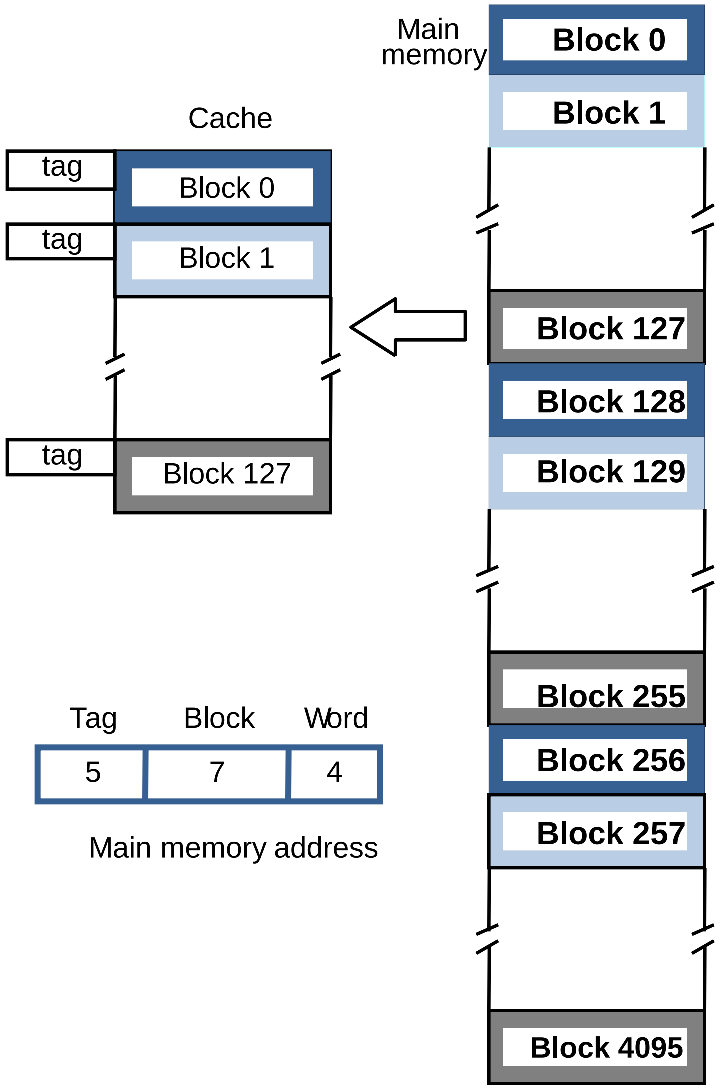
Mapping Functions

- Mapping function determines how memory blocks are placed in the cache

- A simple example:
  - Consider a cache consisting of 128 blocks of 16 words each
  - Total size of cache = 128 * 16 = 2048 or 2K words
  - Assume that main memory is addressable by a 16-bit address
  - Size of main memory = 2^{16} = 64K words
  - Main memory has 4K blocks of 16 words each

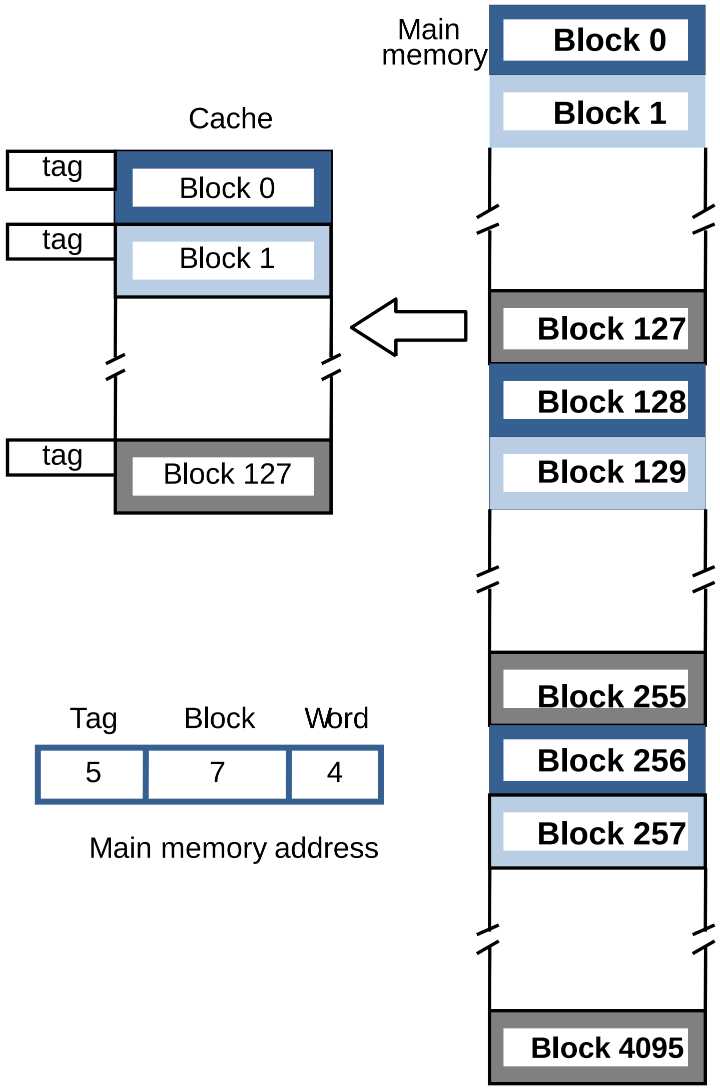
- Three mapping functions:
  - Direct mapping
  - Set-Associative mapping
  - Fully-associative mapping.

Direct Mapping



- Block *j* of main memory maps to block *j modulo 128* of the cache
  - ➢ e.g., memory blocks 0, 128, 256,…… map to cache block 0, memory blocks 1, 129, 257,….. map to cache block 1
- Direct mapping allows only one of the memory blocks mapped to a cache block to be present in the cache at any time
- Even if cache is not full, there may be contention for cache blocks
  - ➢ e.g., what happens if memory block 0 and 128 are accessed by the processor?
- Direct mapping resolves contention by allowing new block to replace the old block,

Direct Mapping (cont.)



- Memory address divided into 3 fields:
  - ➢ High order 5 bits determine which of the possible 32 memory blocks is currently mapped to the cache block. These bits are called “tag” bits
  - ➢ The next 7 bits determine the number of cache block
  - ➢ Low order 4 bits determine one of the 16 words in a block
- Simple to implement but not very flexible
  - ➢ What happens if memory blocks 0 and 128 are repeatedly accessed by the processor in an alternating fashion?

Example: Direct-Mapped Cache

Problem Statement:

A computer system uses **16-bit** memory addresses. It has a **512-byte** cache organized in a direct-mapped manner with **64 bytes** per cache block. Assume that the size of each memory word is 1 byte. The processor reads data from the following word addresses: **64, 128**, **80, 576, 64**

For each address, indicate whether the cache access will result in a hit or a miss.

Solution:

Block size = 64 bytes = 2^6 bytes = 2^6 words (since 1 word = 1 byte)

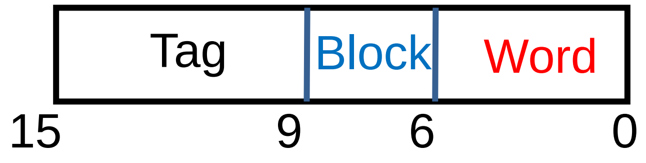
Therefore, **Number of bits in the *Word* field = 6**

Cache size = 512-bytes = 2^9 **=>** No. of cache blocks = Cache size / Block size = $2^9/2^6 = 2^3$

Therefore, **Number of bits in the *Block* field = 3**

Total number of address bits = 16

Therefore, **Number of bits in the *Tag* field = 16 - 6 - 3 = 7**



16-bit Address

Example: Direct-Mapped Cache

Solution (cont.):

Access # 1:

Address = $(\mathbf{64})_{10}$ = $(0000000001000000)_2$

Tag = 0000000, *Block* = 001, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for cache block 001 is set to 0000000**

Block	Tag
000	x
001	0000000
010	x
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Example: Direct-Mapped Cache

Solution (cont.):

Access # 1:

Address = (**64**)$_{10}$ = (0000000001000000)$_2$

Tag = 0000000, *Block* = 001, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for cache block 001 is set to 0000000**

Access # 2:

Address = (**128**)$_{10}$ = (0000000010000000)$_2$

Tag = 0000000, *Block* = 010, *Word* = 000000

Block 010 is empty before this access, this will be a cache **miss**

Block	Tag
000	x
001	0000000
010	x
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Example: Direct-Mapped Cache

Solution (cont.):

Access # 1:

Address = (**64**)$_{10}$ = (0000000001000000)$_2$

Tag = 0000000, *Block* = 001, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for cache block 001 is set to 0000000**

Block	Tag
000	x
001	0000000
010	x
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Access # 2:

Address = (**128**)$_{10}$ = (0000000010000000)$_2$

Tag = 0000000, *Block* = 010, *Word* = 000000

Block 010 is empty before this access, this will be a cache **miss**

After this access, **Tag field for cache block 010 is set to 0000000**

000	x
001	0000000
010	0000000
011	x
100	x
101	x
110	x
111	x

Example: Direct-Mapped Cache

Solution (cont.):

Access # 3:

Address = $(\mathbf{80})_{10}$ = $(0000000001010000)_2$

Tag = 0000000, *Block* = 001, *Word* = 010000

Tag for block 001 (0000000) matches address tag

=> this will be a cache **hit**

Block	Tag
000	x
001	0000000
010	0000000
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Example: Direct-Mapped Cache

Solution (cont.):

Access # 3:

Address = $(\mathbf{80})_{10}$ = $(0000000001010000)_2$

Tag = 0000000, *Block* = 001, *Word* = 010000

Tag for block 001 (0000000) matches address tag

=> this will be a cache **hit**

Block	Tag
000	x
001	0000000
010	0000000
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Access # 4:

Address = $(\mathbf{576})_{10}$ = $(0000001001000000)_2$

Tag = 0000001, *Block* = 001, *Word* = 000000

Tag for block 001 (0000000) does not match the address tag (0000001) => this will be a cache **miss**.

Example: Direct-Mapped Cache

Solution (cont.):

Access # 3:

Address = $(\mathbf{80})_{10}$ = $(0000000001010000)_2$

Tag = 0000000, *Block* = 001, *Word* = 010000

Tag for block 001 (0000000) matches address tag => this will be a cache **hit**

Block	Tag
000	x
001	0000000
010	0000000
011	x
100	x
101	x
110	x
111	x

"x" denotes empty or invalid

Access # 4:

Address = $(\mathbf{576})_{10}$ = $(0000001001000000)_2$

Tag = 0000001, *Block* = 001, *Word* = 000000

Tag for block 001 (0000000) does not match the address tag (0000001) => this will be a cache **miss**. After this access, **Tag field for cache block 001 is set to 0000001**

Block	Tag
000	x
001	0000001
010	0000000
011	x
100	x
101	x
110	x
111	x

Example: Direct-Mapped Cache

Solution (cont.):

Access # 5:

Address = $(64)_{10}$ = $(0000000001000000)_2$

Tag = 0000000, *Block* = 001, *Word* = 000000

Tag for block 001 (0000001) does not match the address tag (0000000) => this will be a cache **miss**.

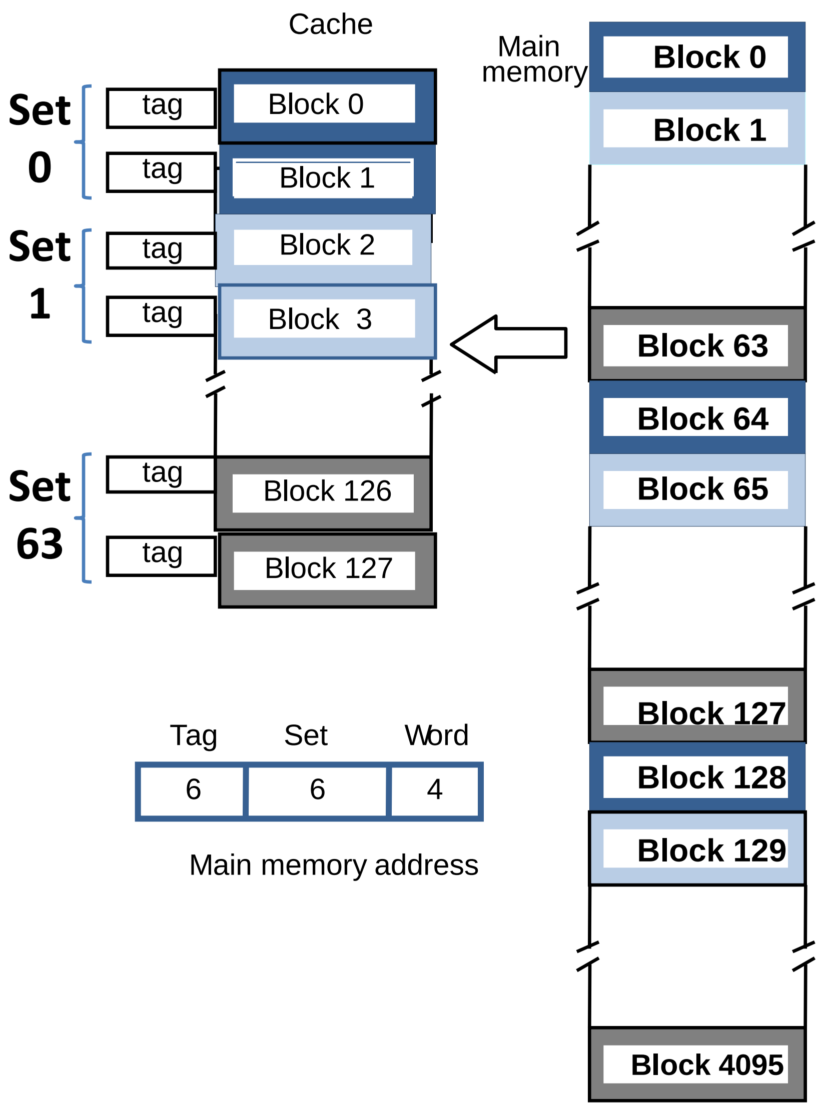
Even though address 64 was previously loaded into the cache, this access is a miss due to the conflict between address 64 and 576 (both addresses map to the same block)

After this access, **Tag field for cache block 001 is set to 0000000**

Block	Tag
000	x
001	0000000
010	0000000
011	x
100	x
101	x
110	x
111	x

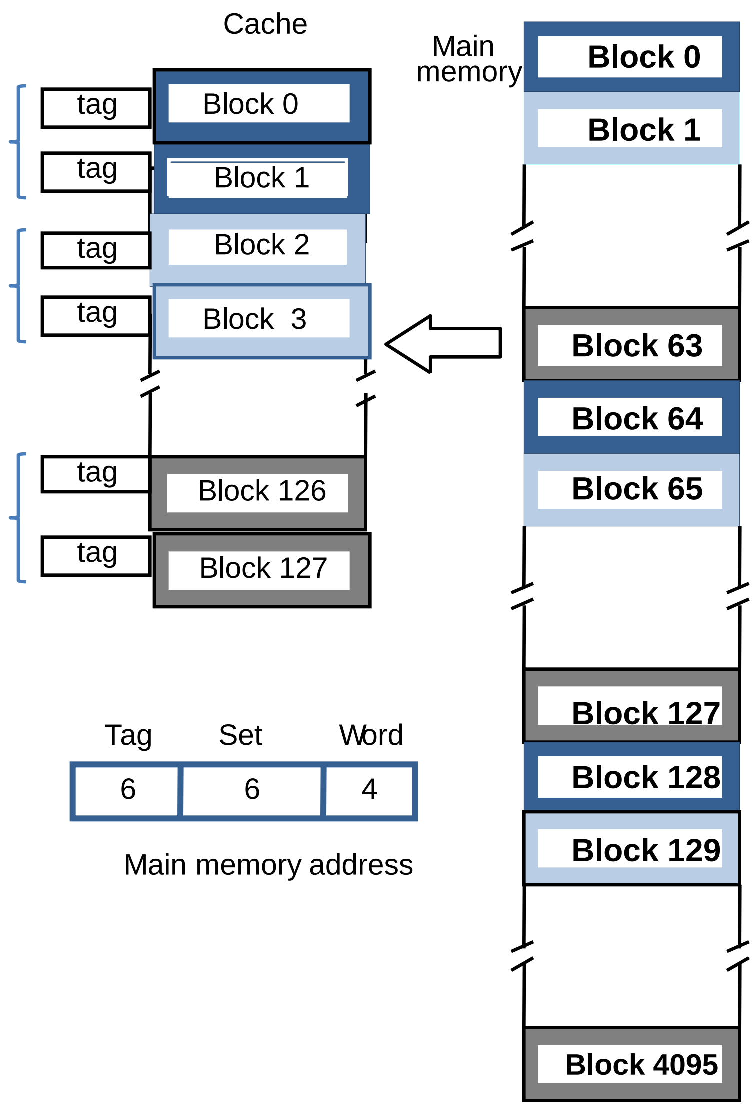
“x” denotes empty or invalid

Set-Associative Mapping



- Cache blocks are grouped into **sets**
- The number of blocks per set is called **associativity** of the cache
- Set-Associative mapping allows a block of main memory to be mapped to any block of a specific set
- Example (2-way set-associative cache):
  - ➢ Divide the cache into 64 sets, with two blocks per set
  - ➢ Memory block 0, 64, 128, ….. etc. map to set 0, and they can occupy either of the two cache blocks in set 0
  - ➢ Memory block 1, 65, 129, ….. etc. map to set 1, and they can occupy either of the two cache blocks in set 1

Set-Associative Mapping (cont.)



- Memory address is divided into 3 fields:
  - High order 6 bits determine the tag field
  - Next 6 bits determine the set number
  - Low order 4 bits determine one of the 16 words in a block
- Cache associativity (number of blocks per cache set) is a design parameter
  - One extreme is to have one block per set, the same as direct mapping
  - Other extreme is to have all the blocks in one set, requiring no set bits (fully associative mapping)

Example: Set-Associative Cache

Problem Statement:

Repeat the previous problem, assuming that the cache is organized as a 2-way set-associative cache.

Solution:

Block size = 64 bytes = 2^6 bytes = 2^6 words (since 1 word = 1 byte)

Therefore, **Number of bits in the *Word* field = 6**

Cache size = 512-bytes = 2^9 bytes

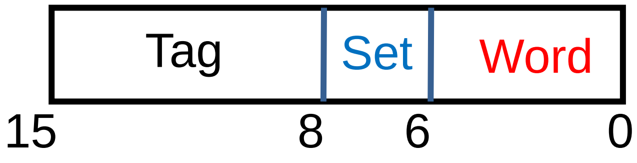
No. of cache blocks per set = 2 (2-way set-associative)

No. of sets = Cache size / (Block size * Blocks per set) = $2^9 / (2^6 * 2) = 2^2 = 4$

Therefore, **Number of bits in the *Set* field = 2**

Total number of address bits = 16

Therefore, **Number of bits in the *Tag* field = 16 - 6 - 2 = 8**



16-bit Address

Example: Set-Associative Cache

Solution (cont.):

Access # 1:

Address = (**64**)$_{10}$ = (0000000001000000)$_2$

Tag = 00000000, *Set* = 01, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for way-0 in set 01 is set to 00000000**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	x
	1	x
11	0	x
	1	x

Example: Set-Associative Cache

Solution (cont.):

Access # 1:

Address = (**64**)$_{10}$ = (0000000001000000)$_2$

Tag = 00000000, *Set* = 01, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for way-0 in set 01 is set to 00000000**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	x
	1	x
11	0	x
	1	x

Access # 2:

Address = (**128**)$_{10}$ = (0000000010000000)$_2$

Tag = 00000000, *Set* = 10, *Word* = 000000

Set 10 is empty before this access, this will be a cache **miss**

Example: Set-Associative Cache

Solution (cont.):

Access # 1:

Address = $(64)_{10} = (0000000001000000)_2$

Tag = 00000000, *Set* = 01, *Word* = 000000

Since the cache is empty before this access, this will be a cache **miss**

After this access, **Tag field for way-0 in set 01 is set to 00000000**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	x
	1	x
11	0	x
	1	x

Access # 2:

Address = $(128)_{10} = (0000000010000000)_2$

Tag = 00000000, *Set* = 10, *Word* = 000000

Set 10 is empty before this access, this will be a cache **miss**

After this access, **Tag field for way-0 in set 10 is set to 00000000**

00	0	x
	1	x
01	0	00000000
	1	x
10	0	00000000
	1	x
11	0	x
	1	x

Example: Set-Associative Cache

Solution (cont.):

Access # 3:

Address = (**80**)$_{10}$ = (0000000001010000)$_2$

Tag = 00000000, *Set* = 01, *Word* = 010000

Tag for way-0 in set 01 (0000000) matches address tag => this will be a cache **hit**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	00000000
	1	x
11	0	x
	1	x

Example: Set-Associative Cache

Solution (cont.):

Access # 3:

Address = (**80**)$_{10}$ = (0000000001010000)$_2$

Tag = 00000000, *Set* = 01, *Word* = 010000

Tag for way-0 in set 01 (0000000) matches address tag => this will be a cache **hit**

Access # 4:

Address = (**576**)$_{10}$ = (0000001001000000)$_2$

Tag = 00000010, *Set* = 01, *Word* = 000000

Neither of the tags in set 01 match the address tag (00000010) => this will be a cache **miss**.

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	00000000
	1	x
11	0	x
	1	x

Example: Set-Associative Cache

Solution (cont.):

Access # 3:

Address = $(\mathbf{80})_{10} = (0000000001010000)_2$

Tag = 00000000, *Set* = 01, *Word* = 010000

Tag for way-0 in set 01 (0000000) matches address tag => this will be a cache **hit**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	x
10	0	00000000
	1	x
11	0	x
	1	x

Access # 4:

Address = $(\mathbf{576})_{10} = (0000001001000000)_2$

Tag = 00000010, *Set* = 01, *Word* = 000000

Neither of the tags in set 01 match the address tag (00000010) => this will be a cache **miss**.

After this access, **Tag field for way-1 in set 01 is set to 00000010**

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	00000010
10	0	00000000
	1	x
11	0	x
	1	x

Example: Set-Associative Cache

Solution (cont.):

Access # 5:

Address = $(\mathbf{64})_{10}$ = $(0000000001000000)_2$

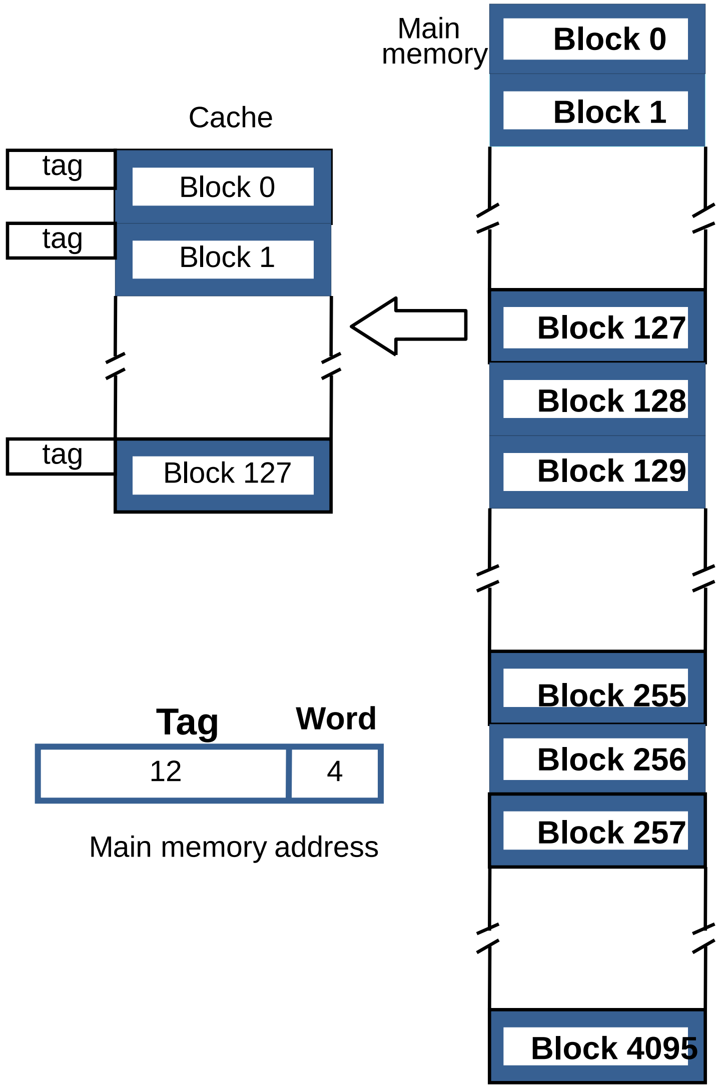
Tag = 00000000, *Set* = 01, *Word* = 000000

Tag for way-0 in set 01 (0000000) matches address tag => this will be a cache **hit**

By using a set-associative cache (as opposed to a direct-mapped cache), this access was converted from a miss to a hit.

Set	Way	Tag
00	0	x
	1	x
01	0	00000000
	1	00000010
10	0	00000000
	1	x
11	0	x
	1	x

Fully Associative Mapping



- Main memory block can be mapped into any cache block
- Memory address is divided into two fields:
  - ➢ High order 12 bits or tag bits identify which one of the 4096 memory blocks is mapped to a given cache block
  - ➢ Low order 4 bits determine of the 16 words within a block
- Complete freedom in choosing the cache location in which to place a memory block
  - Flexible, and uses cache space efficiently
- Higher complexity than direct-mapped cache because of the need to search all 128 tags to determine whether a given block is in the cache. This is called associative search

Replacement Algorithm

- When the cache is full and some new data needs to be brought into the cache, some existing cache block needs to be replaced
- Replacement algorithm decides which block to chose among all candidates

- For direct-mapped cache:
  - New block can be placed in only one cache location
  - Replacement algorithm is trivial, since there is only 1 replacement candidate

- For set-associative cache:
  - New block can be placed in any of the multiple ways in the set
  - Number of replacement candidates is equal to the associativity of the cache
  - Replacement algorithm should chose the block, which is least likely to be accessed in future
  - Least Recently Used (LRU) Replacement Algorithm
    - High probability that bocks accessed recently will be accessed soon
    - Replace the block which has gone the longest time without being accessed