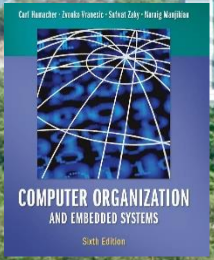
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization

Lecture 07: Cache in Action

Ming-Chang YANG

mcyang@cse.cuhk.edu.hk



Reading: Chap. 8.6

Recall: Memory Hierarchy

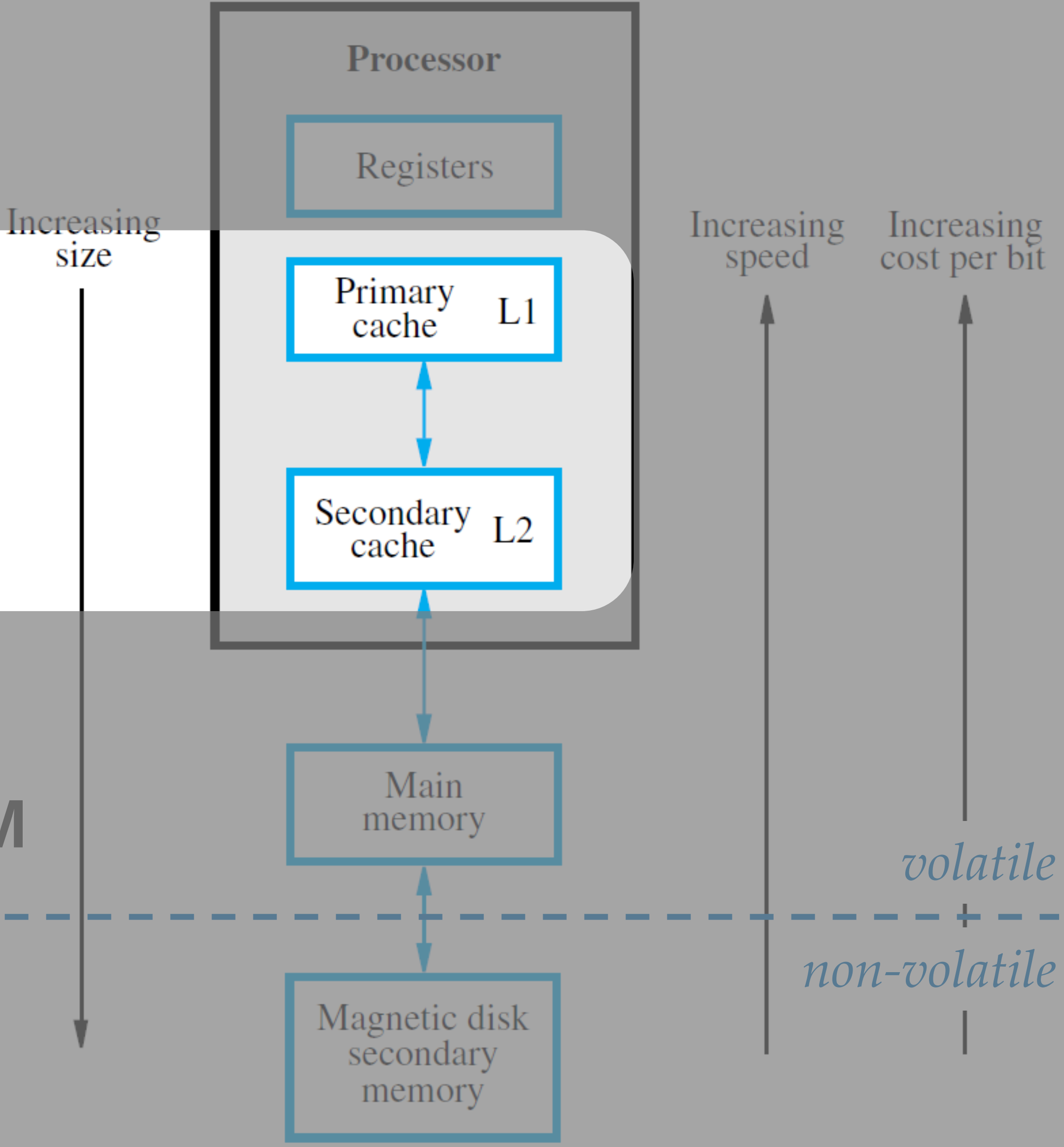
Processor

① Register: **SRAM**

① L1, L2 cache: **SRAM**

② Main memory: **SDRAM**

③ Secondary storage: **NVM/SSD/HDD**



Outline

- Cache Basics

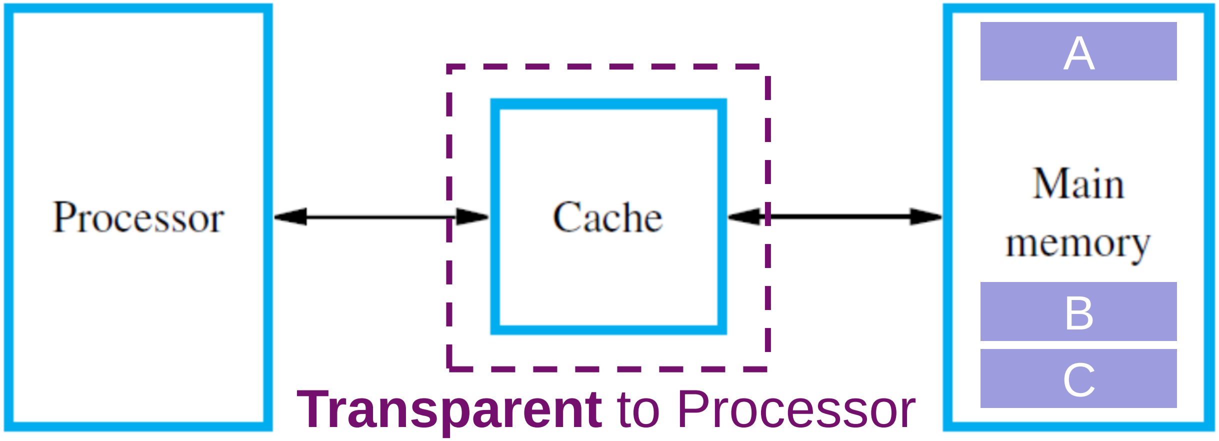
- Mapping Functions
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping

- Replacement Algorithms
  - Optimal Replacement
  - Least Recently Used (LRU) Replacement
  - Random Replacement

- Working Examples

Cache: Fast but Small

- The cache is a small but very fast memory.
  - Interposed between the processor and main memory.



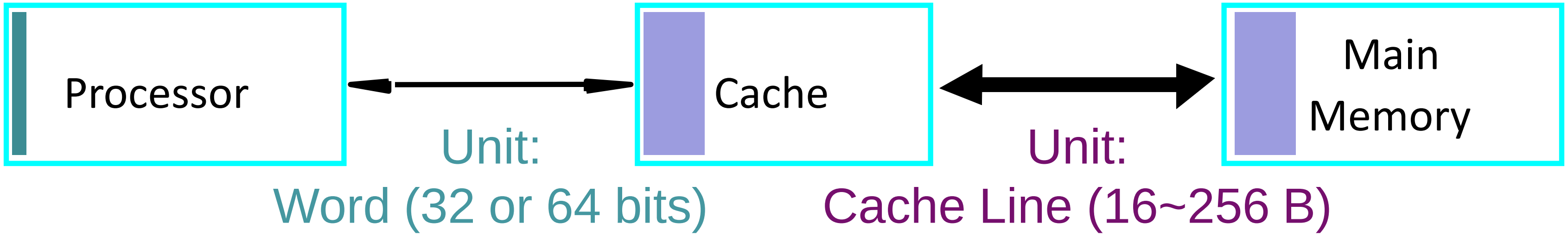
Transparent to Processor

- Its purpose is to make the main memory appear to the processor to be much faster than it actually is.
  - The processor does not need to know explicitly about the existence of the cache, but just feels faster!
- How to? Exploit the locality of reference to “properly” load some data from the main memory into the cache.

Locality of Reference

- **Temporal Locality** (locality in ***time***)
  - If an item is referenced, it will tend to be referenced again soon (e.g., recent calls).
  - **Strategy**: When the data are firstly needed, opportunistically bring it into cache (i.e., we hope it will be used soon).
- **Spatial Locality** (locality in ***space***)
  - If an item is referenced, neighboring items whose addresses are close-by will tend to be referenced soon.
  - **Strategy**: Rather than a single word, fetching more data of adjacent addresses (unit: cache block) from main memory into cache at a time.
- Cache takes both types of locality into considerations.

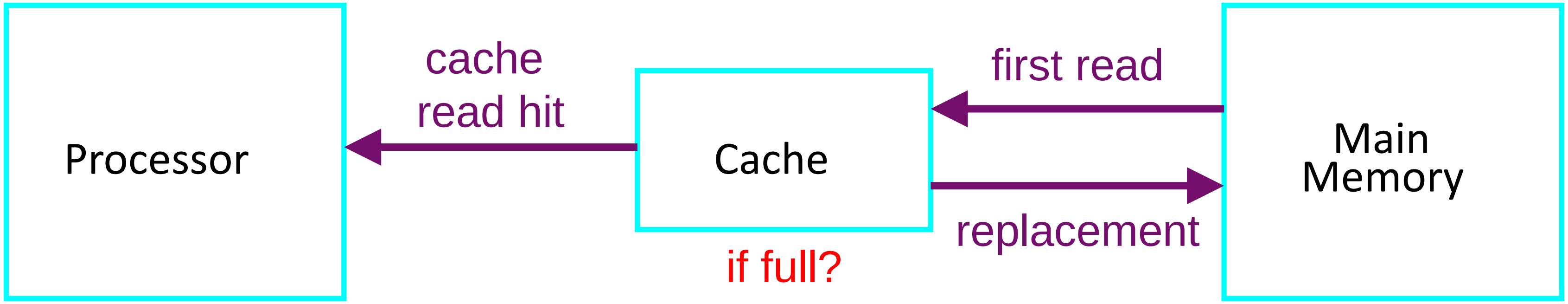
Cache at a Glance



- **Cache Block / Line**: The unit composed of *multiple successive* memory words (size: cache block > word).
  - The contents of a cache block (of memory words) will be loaded into or unloaded from the cache at a time.
- **Cache Read (or Write) Hit/Miss**: The read (or write) operation **can**/**cannot** be performed on the cache.
- **Cache Management**:
  - **Mapping Functions**: Decide how cache is organized and how addresses are mapped to the main memory.
  - **Replacement Algorithms**: Decide which item to be unloaded from cache when cache is full.

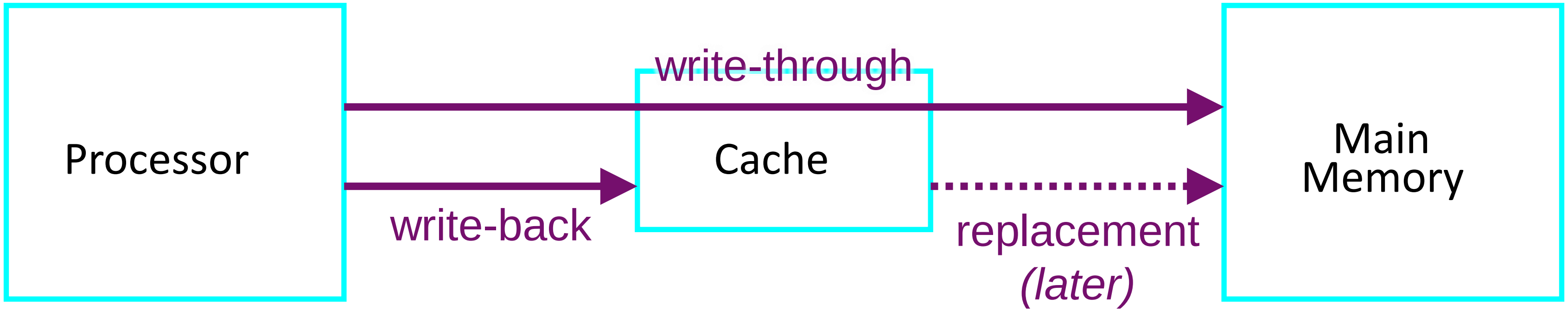
Read Operation in Cache

- **Read Operation**:
  - Contents of a cache block are loaded from the memory into the cache for the first read.
  - Subsequent accesses that can be (hopefully) performed on the cache, called a cache read hit.
  - The number of cache entries is relatively small, we need to keep the most likely to-be-used data in cache.
    - When an un-cached block is required (i.e., cache read miss) but the cache is already full, the replacement algorithm removes a cached block and to create space for the new one.



Write Operation in Cache

- **Write Operation**:
  - – **Write-Through Scheme**: The contents of cache and main memory are updated at the same time.
  - – **Write-Back Scheme**: Update cache only but mark the item as dirty. The corresponding contents in main memory will be updated later when cache block is unloaded.
    - **Dirty**: The data item needs to be written back to the main memory.



- Which scheme is simpler?
- Which one has better performance?

Outline

- Cache Basics

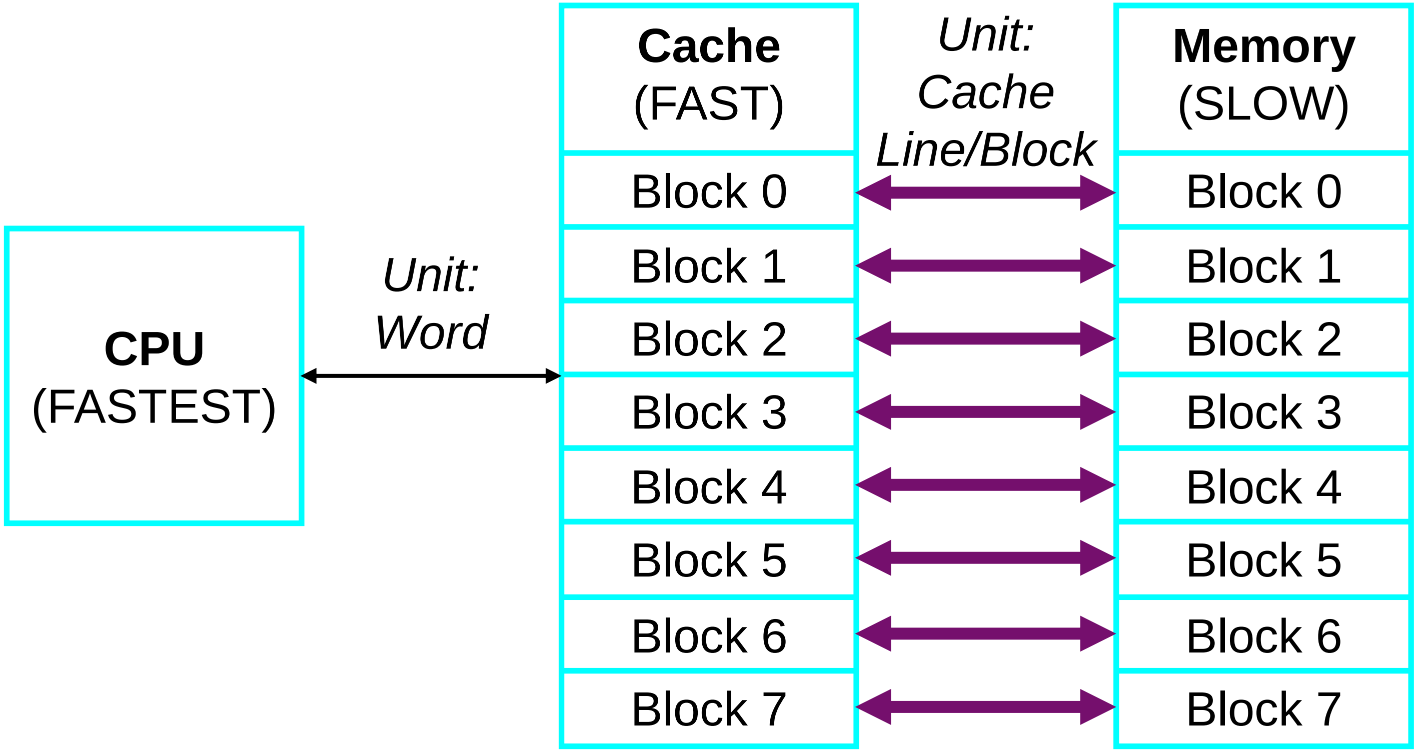
- Mapping Functions
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping

- Replacement Algorithms
  - Optimal Replacement
  - Least Recently Used (LRU) Replacement
  - Random Replacement

- Working Examples

Mapping Functions (1/3)

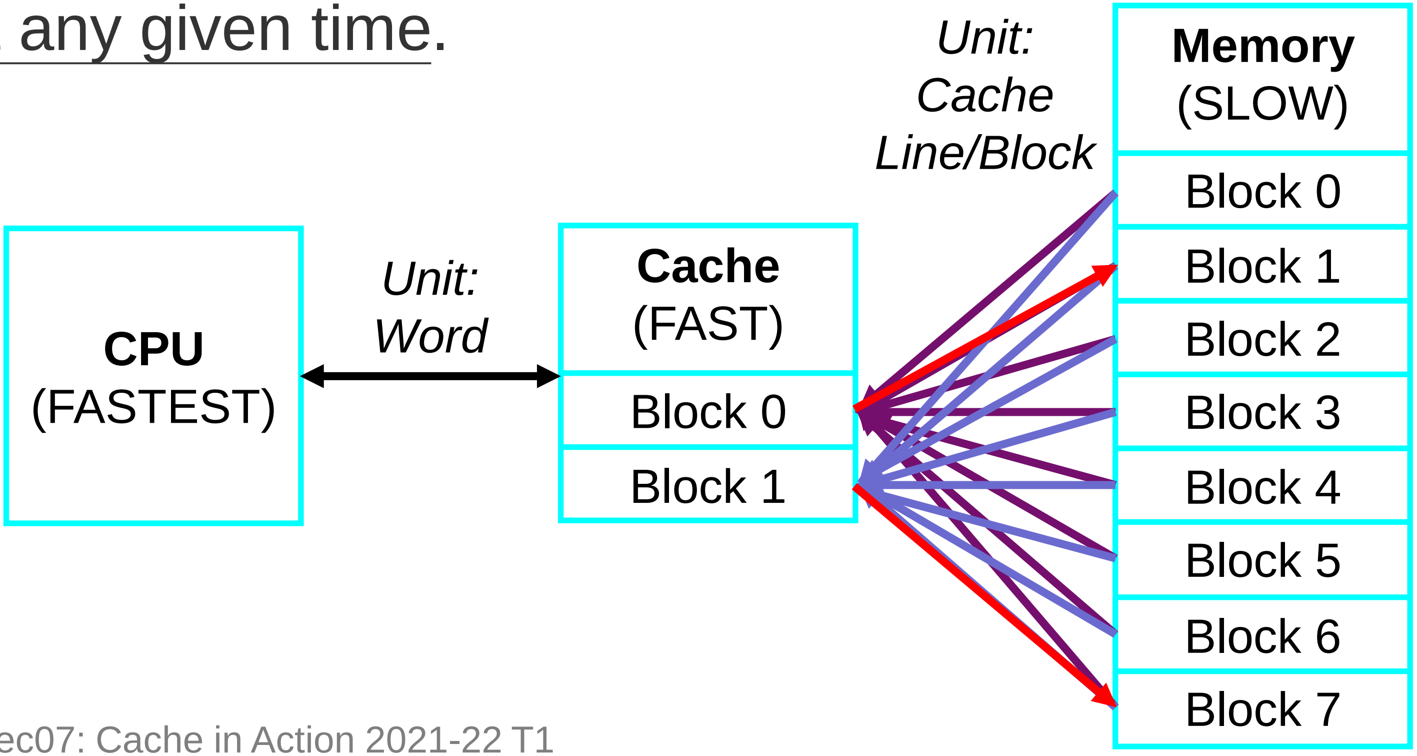
- **Cache-Memory Mapping Function**: A way to record which block of the main memory is now in cache.
- What if the case size == the main memory size?



- Trivial! **One-to-one mapping** is enough!

Mapping Functions (2/3)

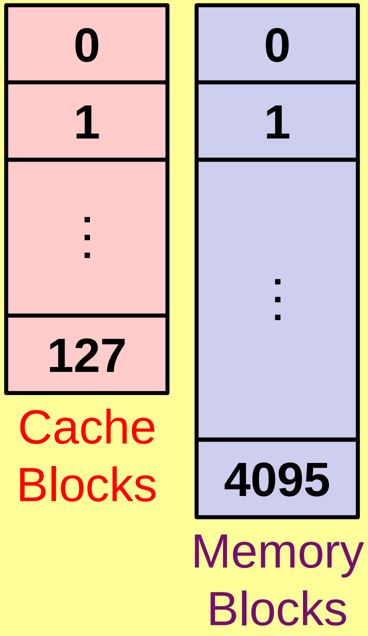
- **Reality**: The cache size is much smaller (<<<) than the main memory size.
- **Many-to-one mapping** is needed!
  - Many blocks in memory compete for one block in cache.
  - One block in cache can only represent one block in memory at any given time.



Mapping Functions (3/3)

- **Design Considerations of Mapping Functions**:
  - **Efficient**: Determine whether a block is in cache quickly.
  - **Effective**: Make full use of cache to increase cache hit ratio.
    - **Cache Hit/Miss Ratio**: the probability of cache hits/misses.
- In the following discussion, we assume:
  - **Synonym**: Cache Line = Cache Block = Block
    - *Note: A cache block is of successive memory words.*
  - 1 Word = 16 bits = 2^1 Bytes
  - 1 Block = 8 Words = 2^3 Words
  - **Cache Size**: 2K Bytes → 128 **Cache Blocks**
    - **Cache Block (CB)**: The block in the cache.
  - **Memory Size**: 16-bit Address → 2^{16} = 64K Bytes

    → 4096 **Memory Blocks**
    - **Memory Block (MB)**: The block in the main memory.



Recall: Big-Endian and Little-Endian

- **Big-Endian Ordering** (e.g., Motorola):
  - ① **Byte addresses** within a word are ordered left-to-right;
  - ② Lower byte addresses are used for more significant bytes of a multi-byte data (e.g., numbers).
- **Little-Endian Ordering** (e.g., Intel):
  - ① **Byte addresses** within a word are ordered right-to-left;
  - ② Lower byte addresses are used for less significant bytes of a multi-byte data (e.g., numbers).

Big Endian

Byte address

← 32 bits →

Word address	+0	+1	+2	+3
0	0	1	2	3
4	4	5	6	7
	⋮			
2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

Little Endian

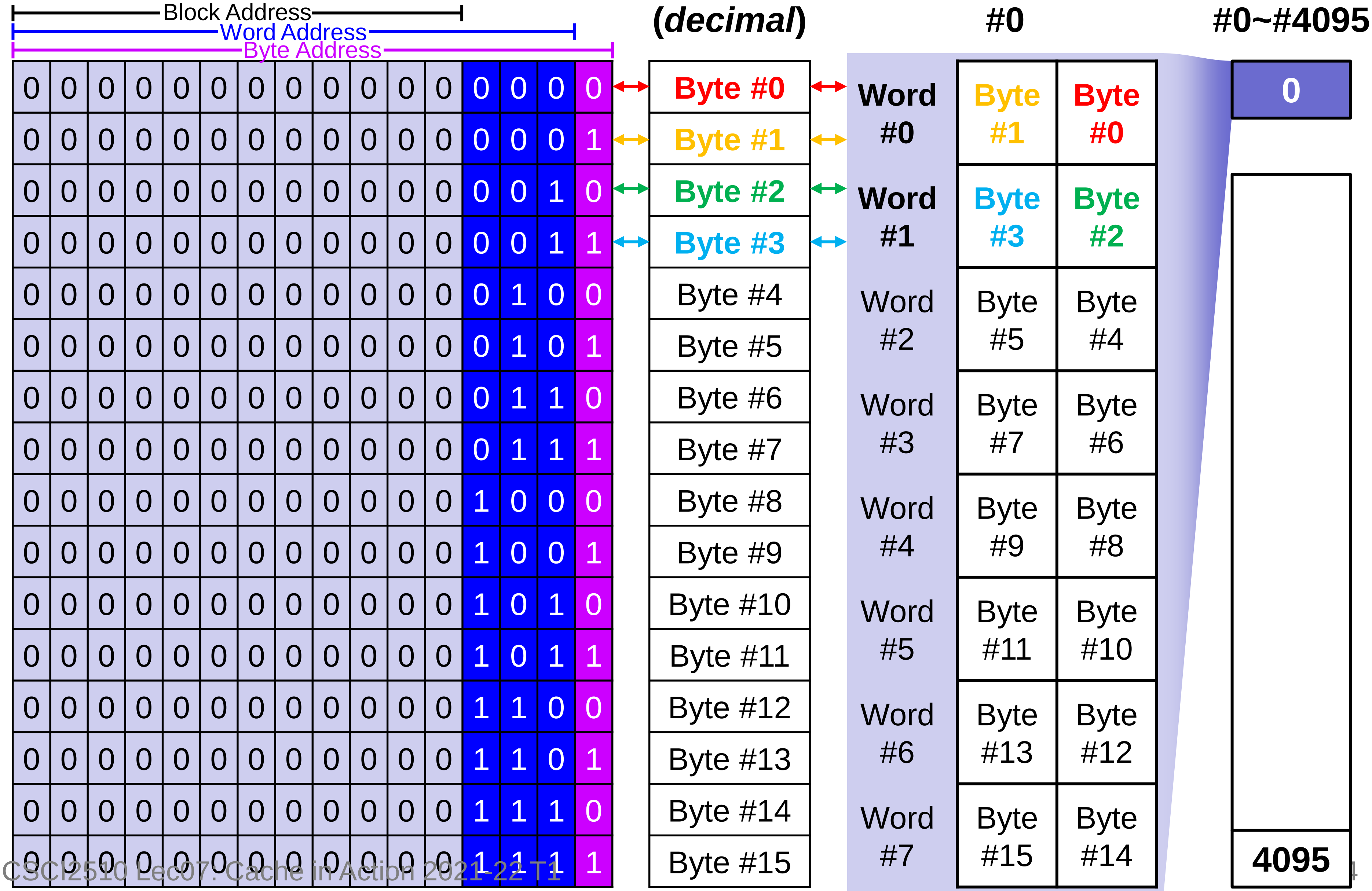
Byte address

← 32 bits →

Word address	+3	+2	+1	+0
0	3	2	1	0
4	7	6	5	4
	⋮			
2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

Example: Memory Block #0

1 Block = 2^3 Words
1 Word = 2^1 Bytes



16-bit Memory Address (*binary*)																Byte Addr. (*decimal*)
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Byte #0**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**Byte #1**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**Byte #2**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	**Byte #3**
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Byte #4
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Byte #5
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	Byte #6
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	Byte #7
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Byte #8
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Byte #9
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	Byte #10
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	Byte #11
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Byte #12
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	Byte #13
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	Byte #14
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	Byte #15

Memory Block #0

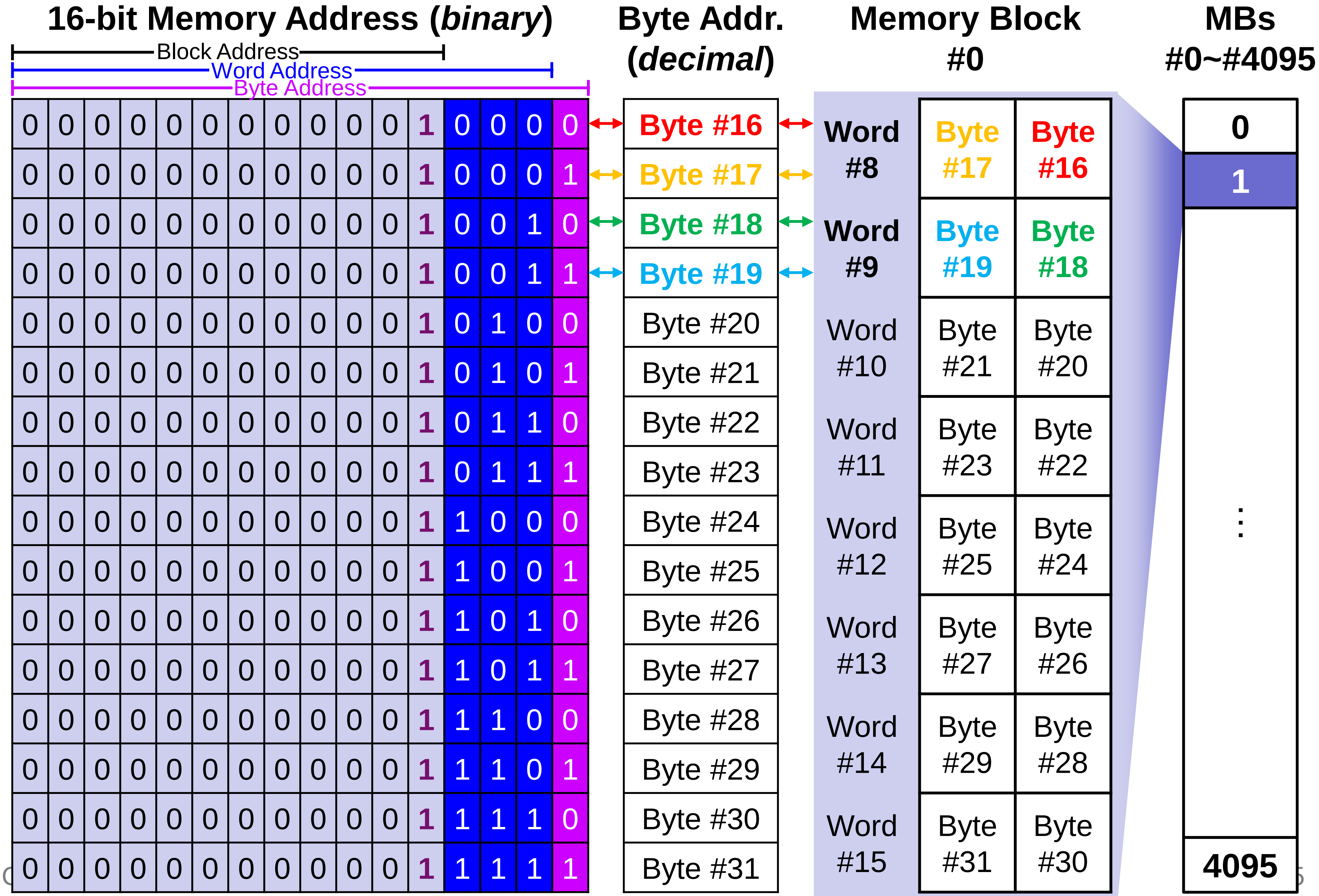
Word #0	**Byte #1**	**Byte #0**
Word #1	**Byte #3**	**Byte #2**
Word #2	Byte #5	Byte #4
Word #3	Byte #7	Byte #6
Word #4	Byte #9	Byte #8
Word #5	Byte #11	Byte #10
Word #6	Byte #13	Byte #12
Word #7	Byte #15	Byte #14

MBs #0~#4095

0
4095

Example: Memory Block #1

1 Block = 2^3 Words
1 Word = 2^1 Bytes



16-bit Memory Address (*binary*)

Block Address
Word Address
Byte Address

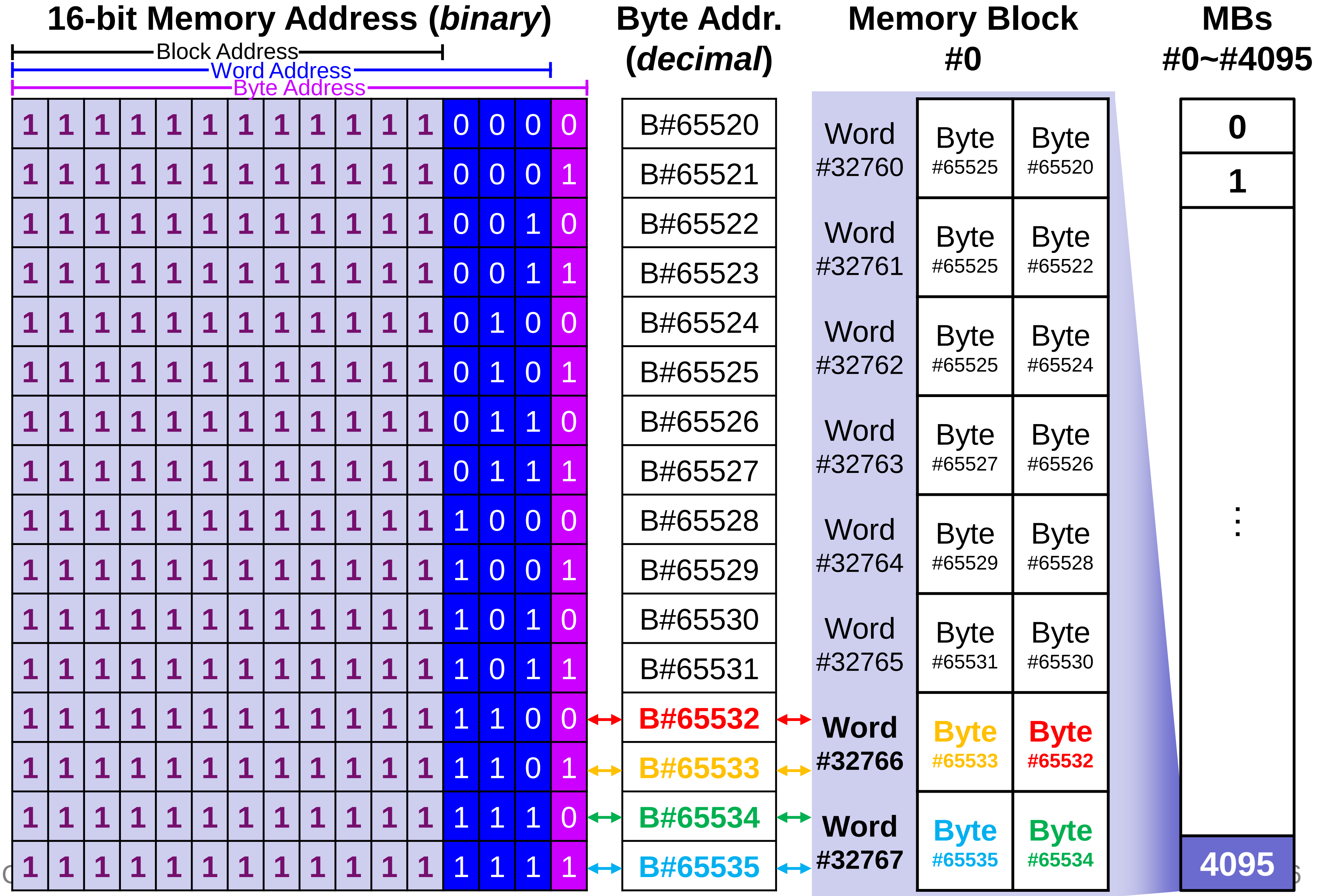
16-bit Memory Address (*binary*)																Byte Addr. (*decimal*)	Memory Block #0		
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**Byte #16**	**Word #8**	**Byte #17**	**Byte #16**
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	**Byte #17**			
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	**Byte #18**	**Word #9**	**Byte #19**	**Byte #18**
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	**Byte #19**			
0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	Byte #20	Word #10	Byte #21	Byte #20
0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	Byte #21			
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	Byte #22	Word #11	Byte #23	Byte #22
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	Byte #23			
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	Byte #24	Word #12	Byte #25	Byte #24
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Byte #25			
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	Byte #26	Word #13	Byte #27	Byte #26
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	Byte #27			
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	Byte #28	Word #14	Byte #29	Byte #28
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	Byte #29			
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	Byte #30	Word #15	Byte #31	Byte #30
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	Byte #31			

MBs #0~#4095

0
1
⋮
4095

Example: Memory Block #4095

1 Block = 2^3 Words
1 Word = 2^1 Bytes



16-bit Memory Address (*binary*)																Byte Addr. (*decimal*)
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	B#65520
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	B#65521
1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	B#65522
1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	B#65523
1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	B#65524
1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	B#65525
1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	B#65526
1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	B#65527
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	B#65528
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	B#65529
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	B#65530
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	B#65531
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	**B#65532**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	**B#65533**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	**B#65534**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**B#65535**

Memory Block #0

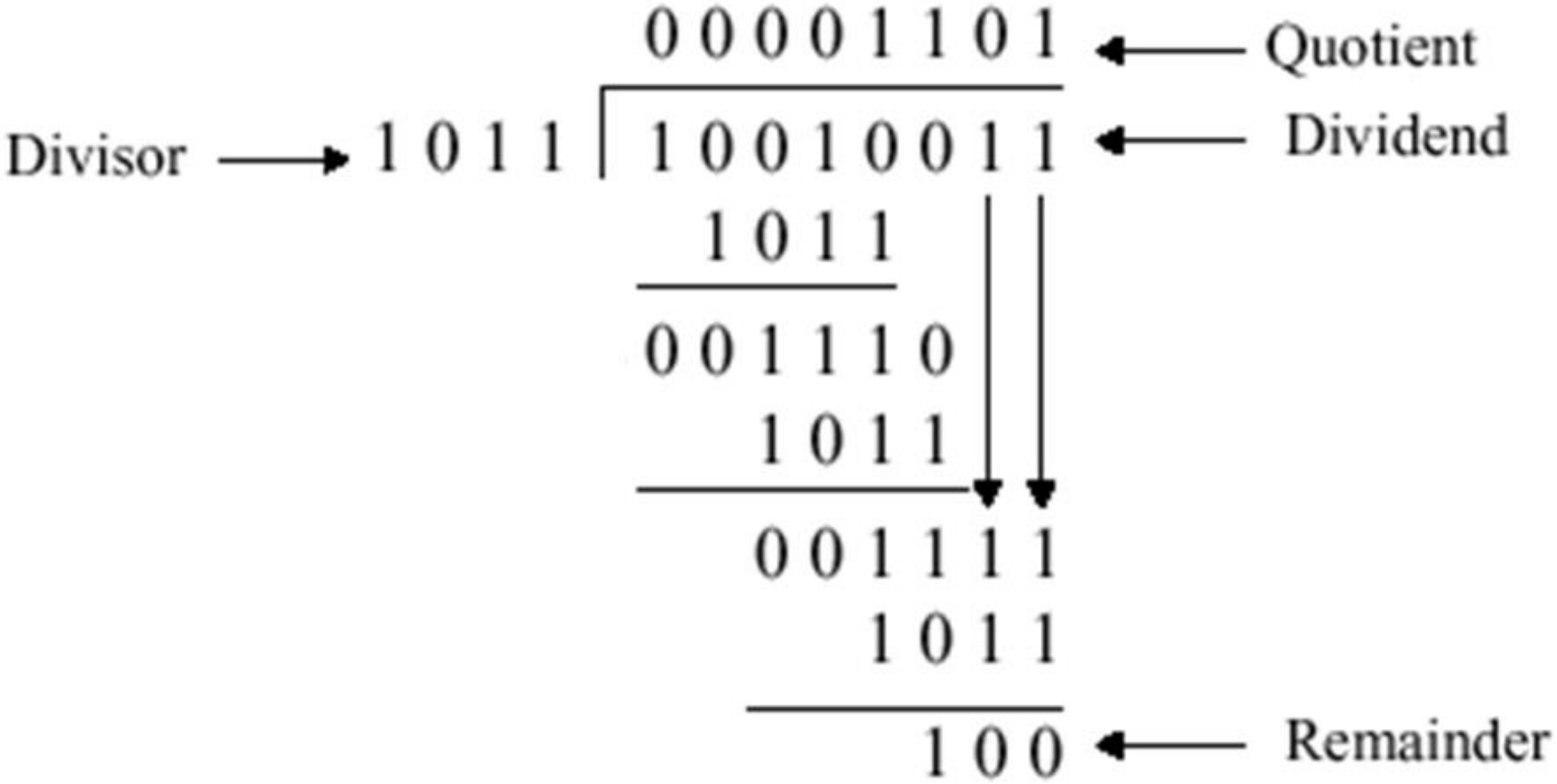
Word #32760	Byte #65525	Byte #65520
Word #32761	Byte #65525	Byte #65522
Word #32762	Byte #65525	Byte #65524
Word #32763	Byte #65527	Byte #65526
Word #32764	Byte #65529	Byte #65528
Word #32765	Byte #65531	Byte #65530
Word #32766	**Byte #65533**	**Byte #65532**
Word #32767	**Byte #65535**	**Byte #65534**

MBs #0~#4095

0
1
⋮
4095

Prior Knowledge: Modulo Operator

- The **modulo (%)** operator is used to divide two numbers and get the remainder.
- Example:



Class Exercise 7.1

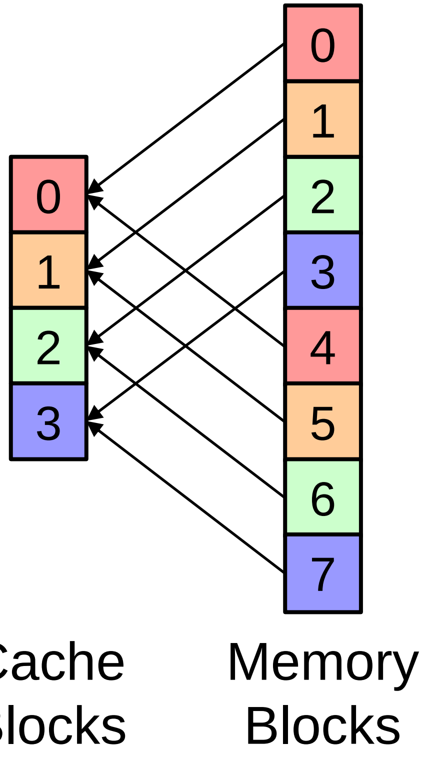
Student ID: __________ Date:
Name: ________________ _______

- Given the same dividend $(10010011)_2$ as the previous example, what will be the quotient and remainder if the divisor equals to $(10)_2$, $(100)_2$, …, $(10000000)_2$?

Direct Mapping (1/4)

Direct

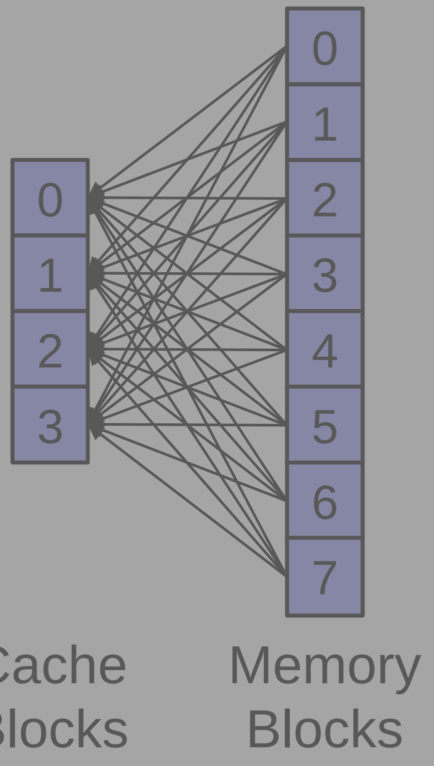
- A Memory Block is **directly mapped** (%) to a Cache Block.



Associative

- A Memory Block can be **mapped to any** Cache Block.

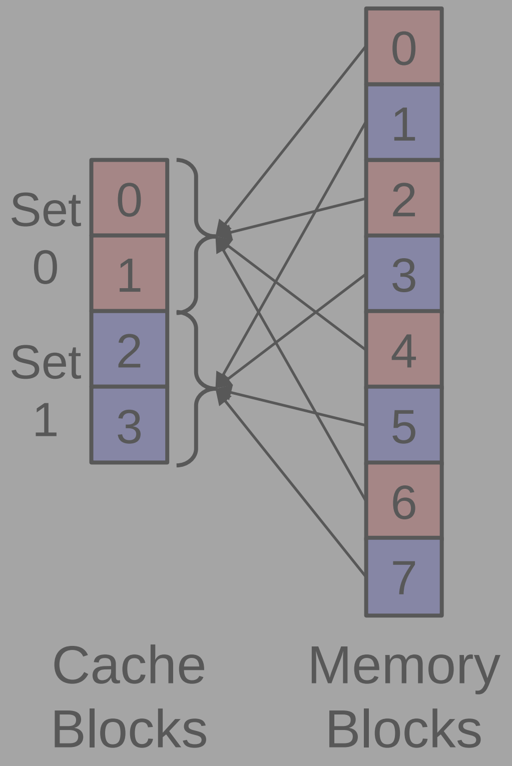
(First come first serve!)



Set Associative

- A Memory Block is **directly mapped** (%) to a Cache **Set**.

(In a set? Associative!)



Direct Mapping (2/4)

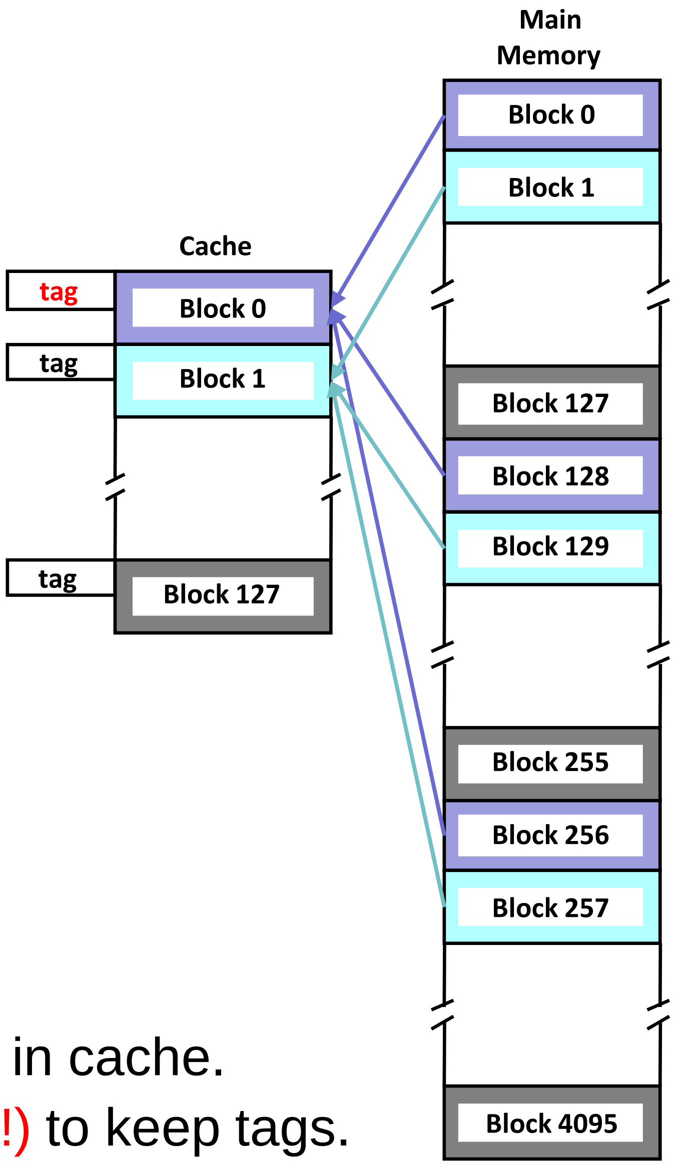
- **Direct Mapped Cache**: Each Memory Block will be directly mapped to a Cache Block.
- **Direct Mapping Function**:

  MB #`j` → CB #(`j mod 128`)

  – 128? There're 128 Cache Blocks.

  – 32 MBs are mapped to 1 CB.
    - MBs **0, 128, 256, …, 3968** → CB **0**.
    - MBs **1, 129, 257, …, 3969** → CB **1**.
    - …
    - MBs **127, 255, 383, …, 4095** → CB **127**.

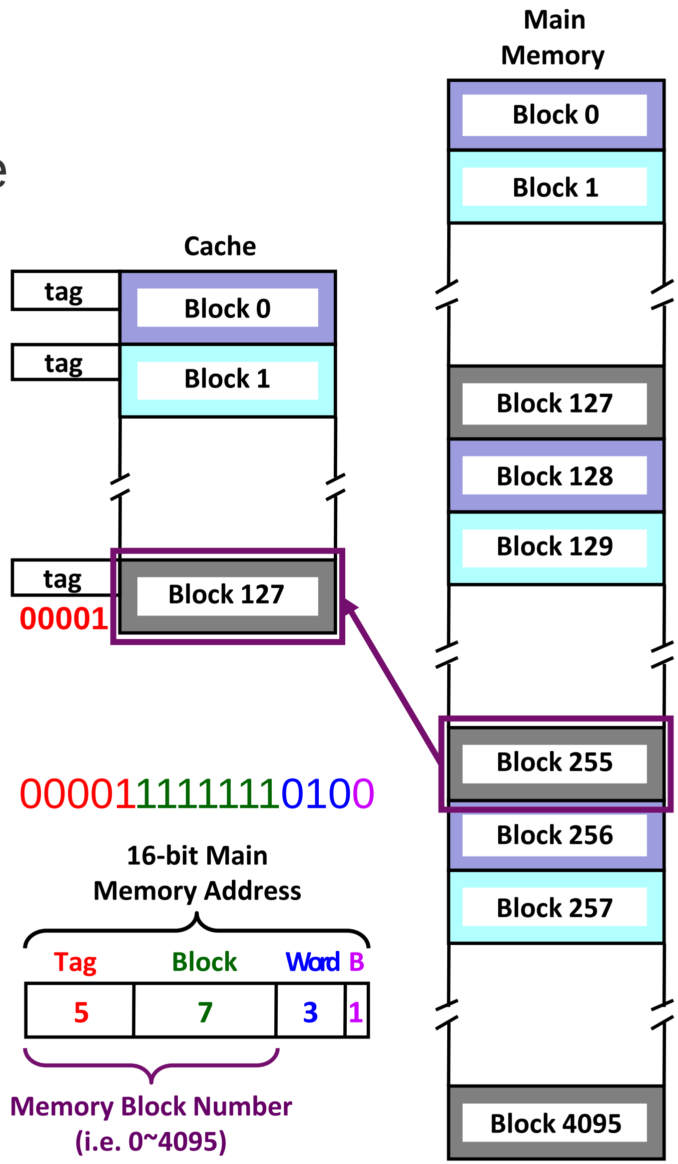
  – A **tag** is needed for each CB.
    - Many MBs will be mapped to a same CB in cache.
    - We need to use some cache space (cost!) to keep tags.



Direct Mapping (3/4)

1 Block = 2^3 Words
1 Word = 2^1 Bytes

- **Trick**: Interpret the 16-bit main memory address as follows:
  - **Tag**: Keep track of which MB is placed in the corresponding CB.
    - 5 bits: 16 – (7 + 4) = 5 bits.
  - **Block**: Determine the CB in cache.
    - 7 bits: There're 128 = 2^7 cache blocks.
  - **Word**: Select one word in a block.
    - 3 bits: There're 8 = 2^3 words in a block.
  - **Byte**: Select one byte in a word.
    - 1 bits: There're 2 = 2^1 bytes in a word.
- Ex: CPU is looking for $(0FF4)_{16}$
  - MAR = $(0000\ 1111\ 1111\ 0100)_2$
  - MB = $(0000\ 1111\ 1111)_2 = (255)_{10}$
  - CB = $(1111111)_2 = (127)_{10}$
  - Tag = $(00001)_2$



Direct Mapping (4/4)

- Why the first 5 bits for **tag**? And why the middle 7 bits for **block**?

MB #`j` → CB #(`j mod 128`)

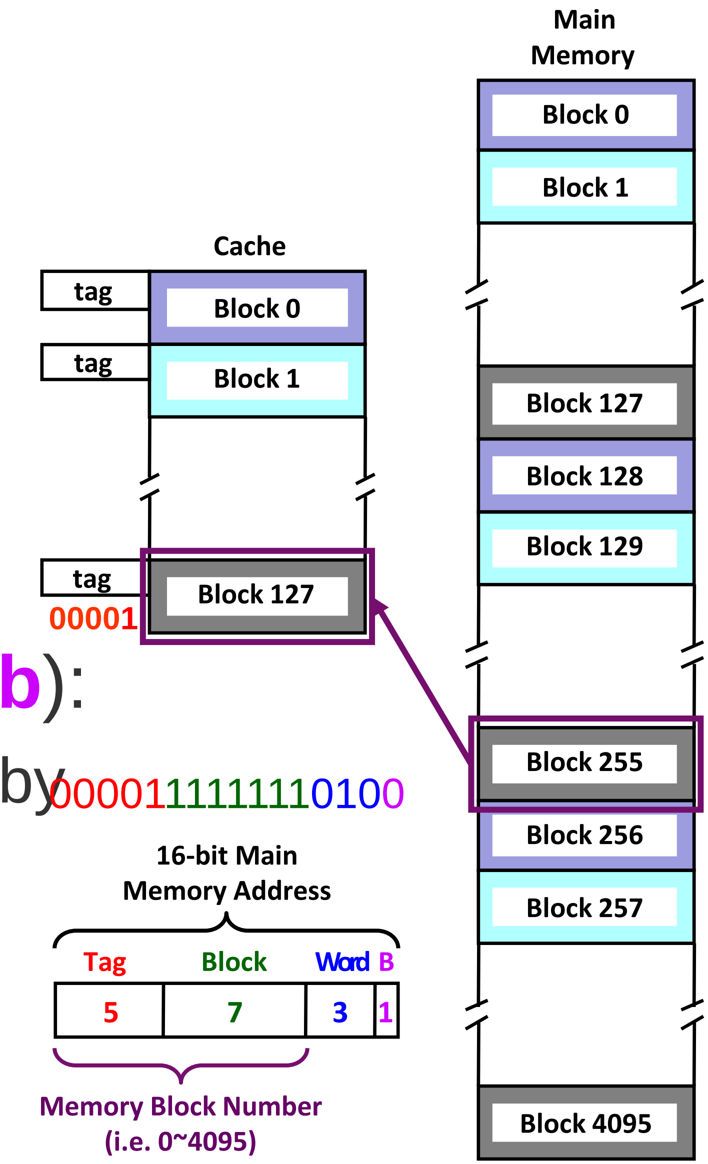
00001 Quotient

10000000) 000011111111~~0100~~

(128)$_{10}$ 10000000

1111111 Remainder

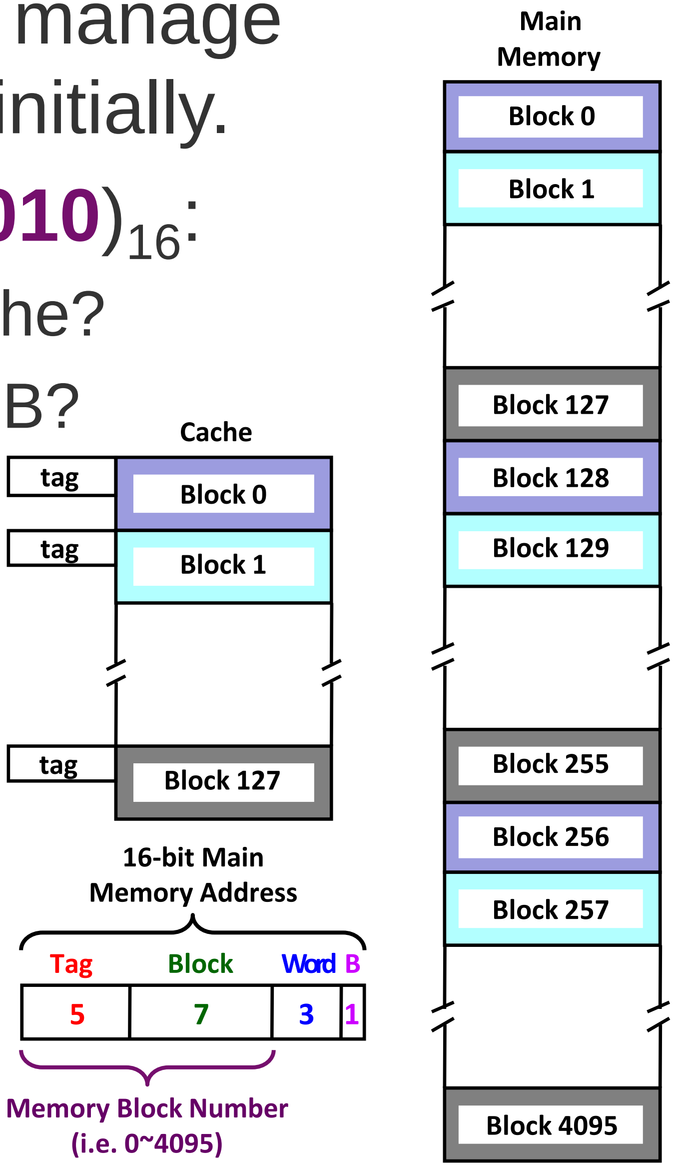
- Search a 16-bit address (**t**, **b**, **w**, **b**):
  ① See if MB (**t**, **b**) is already in CB **b** by comparing **t** with the **tag** of CB **b**.
  ② If not, replace CB **b** with MB (**t**, **b**) and update **tag** of CB **b** using **t**.
  ③ Finally access the word **w** in CB **b**.



Class Exercise 7.2

1 Block = 2^3 Words
1 Word = 2^1 Bytes

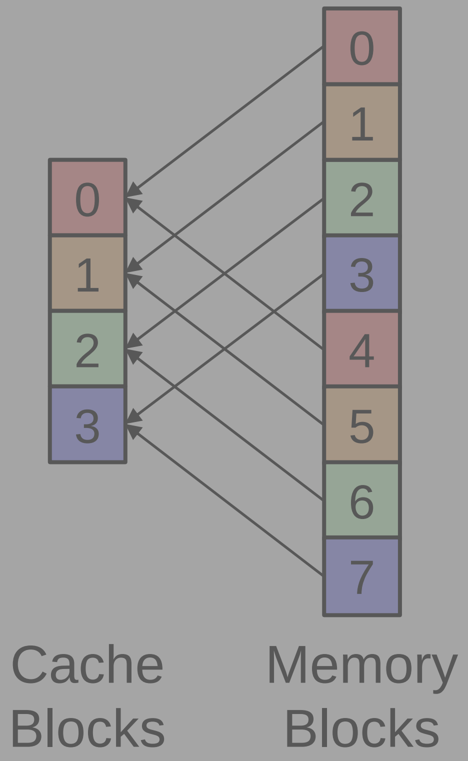
- Assume direct mapping is used to manage the cache, and all CBs are empty initially.
- Considering CPU is looking for $(\mathbf{8010})_{16}$:
  - Which MB will be loaded into the cache?
  - Which CB will be used to store the MB?
  - What is the new tag for the CB?



Associative Mapping (1/3)

Direct

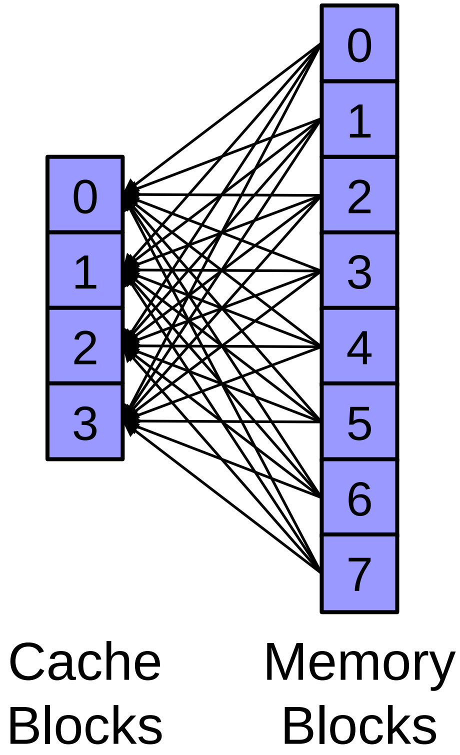
•A Memory Block is **directly mapped** (%) to a Cache Block.



Associative

•A Memory Block can be **mapped to any** Cache Block.

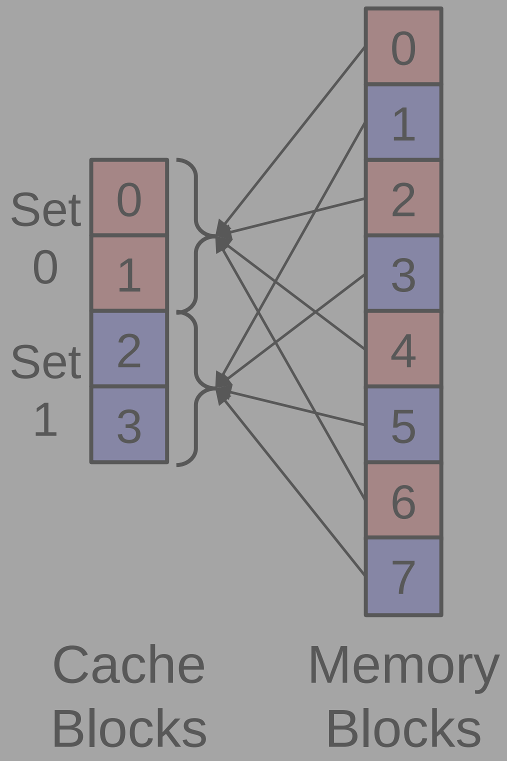
(First come first serve!)



Set Associative

- A Memory Block is **directly mapped** (%) to a Cache **Set**.

(In a set? Associative!)

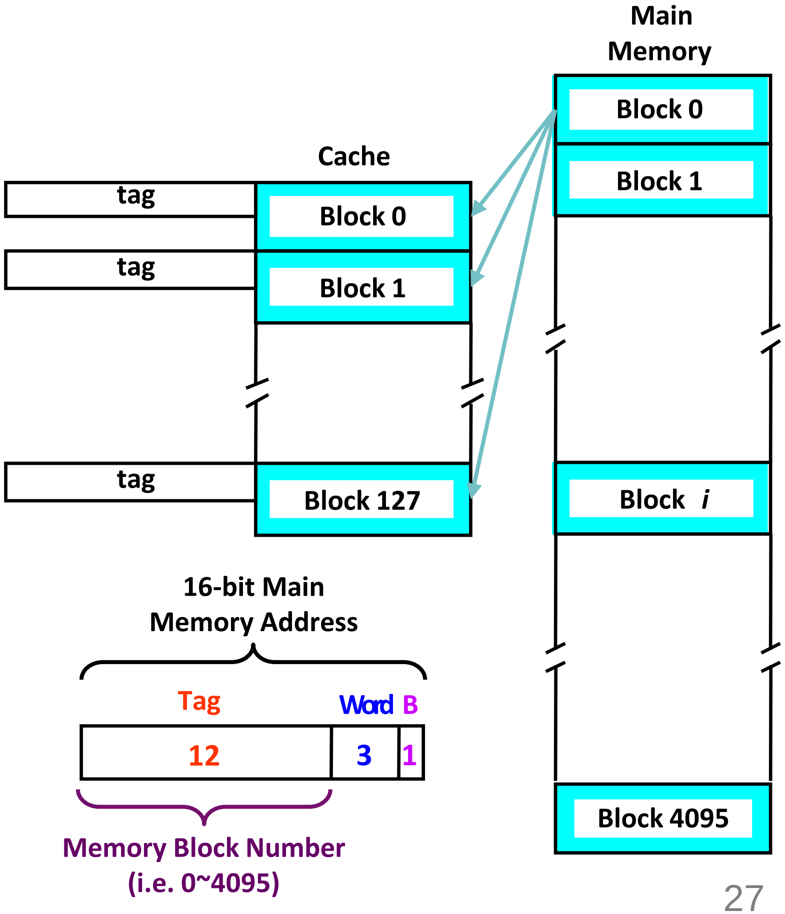


Associative Mapping (2/3)

- **Direct Mapping**: A MB is restricted to a particular CB determined by mod operation.
- **Associative Mapping:**

  Allow a MB to be mapped to any CB in the cache.

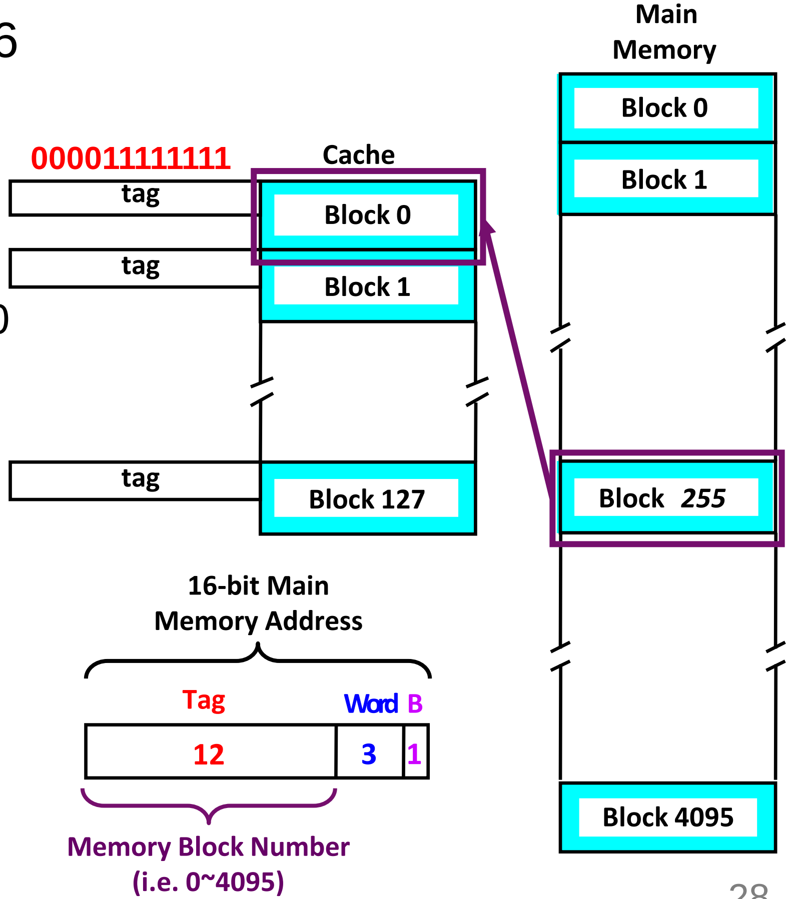
- **Trick**: Interpret the 16-bit main memory address as follows:
  - **Tag**: The first **12** bits (i.e., the MB number) are all used to represent a MB.
  - **Word** & **Byte**: The last **3** & **1** bits for selecting a word & byte in a block.



Associative Mapping (3/3)

1 Block = 2^3 Words
1 Word = 2^1 Bytes

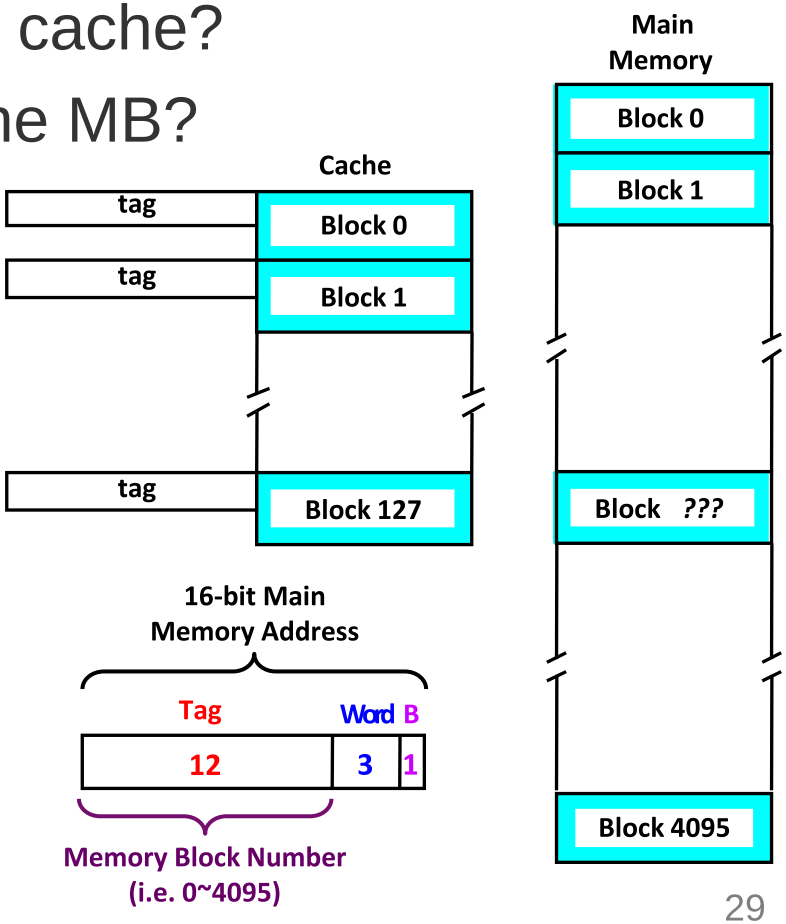
- How to determine the CB?
  - There's no pre-determined CB for any MB.
  - All CBs are used in the first-come-first-serve (FCFS) basis.
- Ex: CPU is looking for $(0FF4)_{16}$
  - Assume all CBs are empty.
  - MAR = $(0000\ 1111\ 1111\ 0100)_2$
  - MB = $(0000\ 1111\ 1111)_2 = (255)_{10}$
  - Tag = $(0000\ 1111\ 1111)_2$
- Search a 16-bit addr. (**t**, **w**, **b**):
  - **ALL tags** of **128 CBs** must be **compared** with t to see whether MB **t** is currently in the cache.
    - 128 tag comparisons can be done **in parallel** by hardware (cost!).



Class Exercise 7.3

1 Block = 2^3 Words
1 Word = 2^1 Bytes

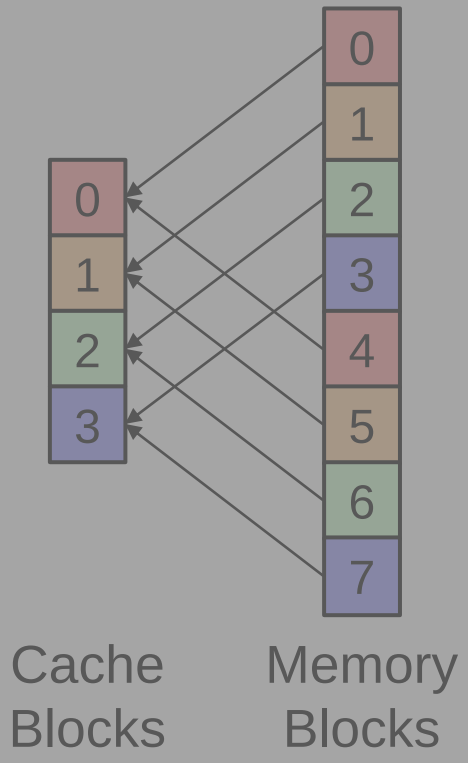
- Assume associative mapping is used to manage the cache, and all CBs are empty initially.
- Considering CPU is looking for $(\mathbf{8010})_{16}$:
  - Which MB will be loaded into the cache?
  - Which CB will be used to store the MB?
  - What is the new tag for the CB?



Set Associative Mapping (1/3)

Direct

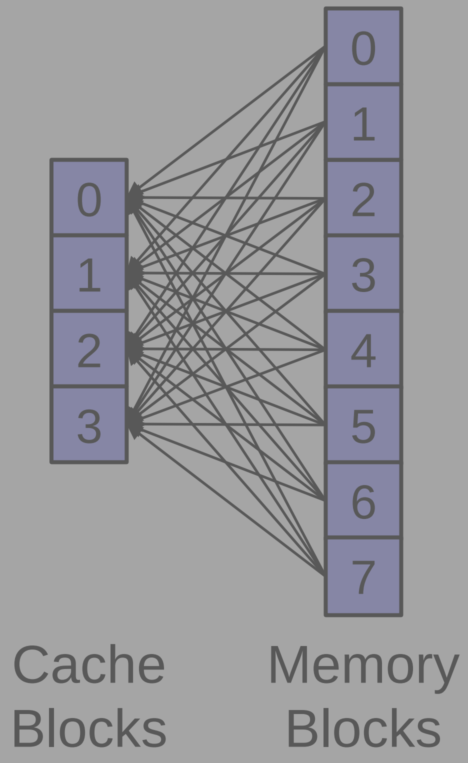
- A Memory Block is **directly mapped** (%) to a Cache Block.



Associative

- A Memory Block can be **mapped to any** Cache Block.

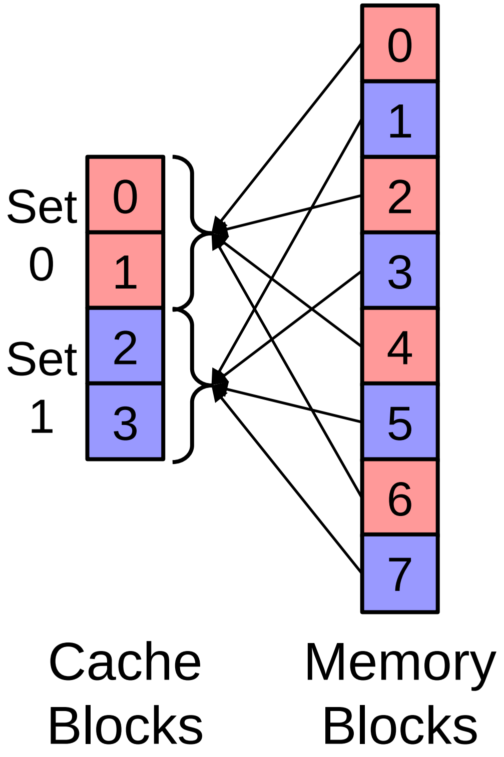
(First come first serve!)



Set Associative

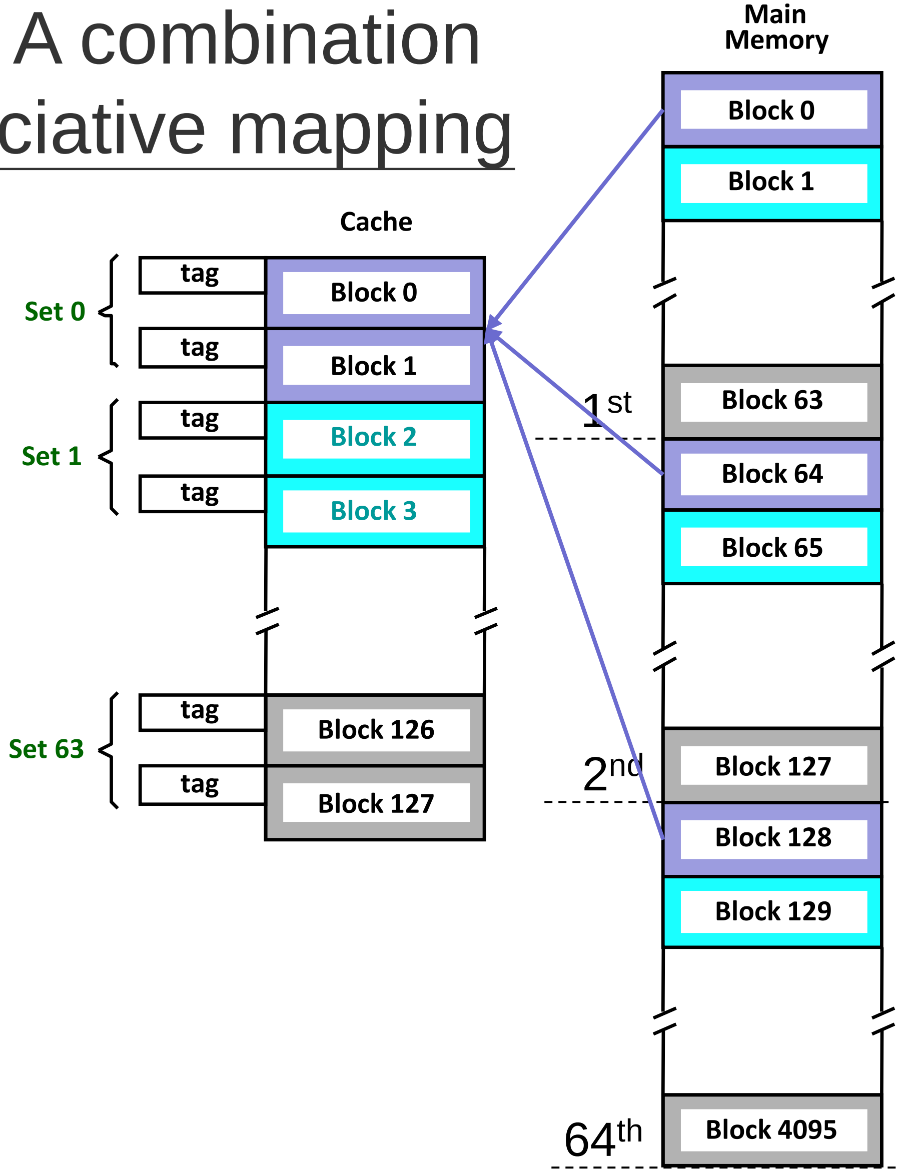
- A Memory Block is **directly mapped** (%) to a Cache **Set**.

(In a set? Associative!)



Set Associative Mapping (2/3)

- **Set Associative Mapping**: A combination of direct mapping and associative mapping
  - **Direct**: First map a MB to a cache set (instead of a CB)
  - **Associative**: Then map to any CB in the cache set
- ***K*-way Set Associative**: A cache set is of ***k*** CBs.
  - Ex: 2-way set associative
    - $128 \div 2 = 64\ (sets)$
    - For MB #`j`, (`j mod 64`) derives the **Set** number.
      - E.g. MBs 0, 64, 128, …, 4032 → Cache Set #0.



Set Associative Mapping (3/3)

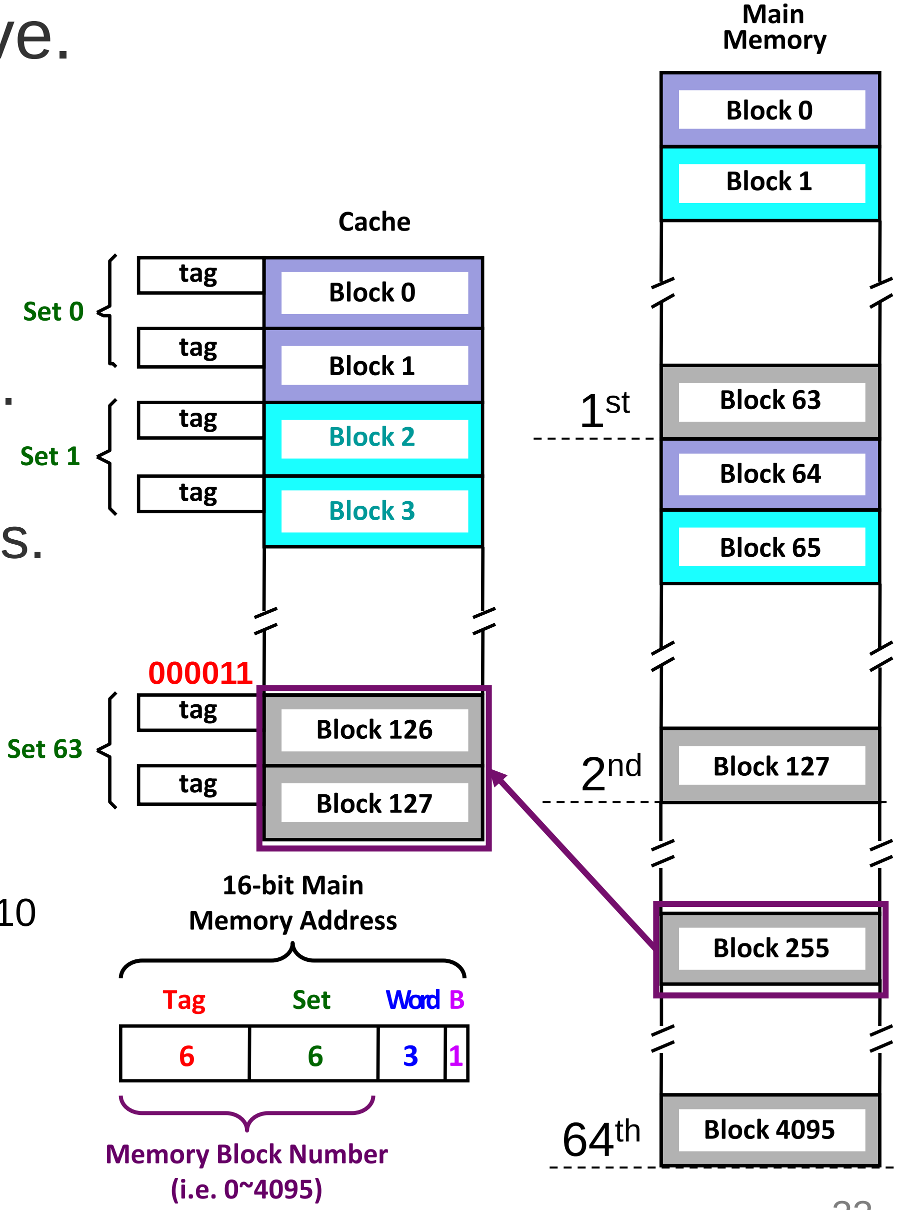
1 Block = 2^3 Words
1 Word = 2^1 Bytes

- Consider 2-way set associative.
- **Trick**: Interpret the 16-bit address as follows:
  - **Tag**: The first **6** bits (**quotient**).
  - **Set**: The middle **6** bits (**remainder**).
    - **6** bits: There're 2^6 cache sets.
  - **Word** & **Byte**: The last **3** & **1** bits.

Ex: CPU is looking for $(0FF4)_{16}$

- Assume all CBs are empty.
- MAR = $(0000\ 1111\ 1111\ 0100)_2$
- MB = $(0000\ 1111\ 1111)_2 = (255)_{10}$
- Cache Set = $(111111)_2 = (63)_{10}$
- Tag = $(000011)_2$

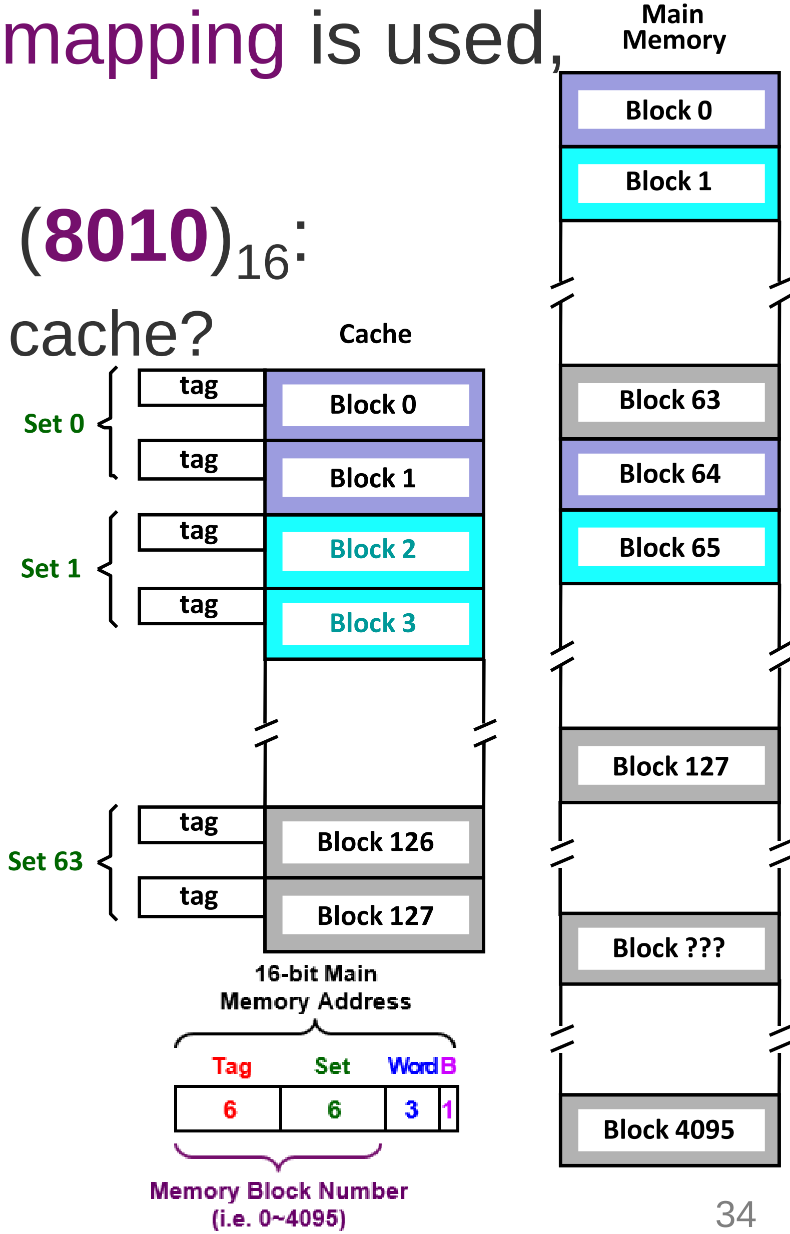
Note: **ALL** **tags** of CBs in a set must be compared (done in parallel by hardware).



Class Exercise 7.4

1 Block = 2^3 Words
1 Word = 2^1 Bytes

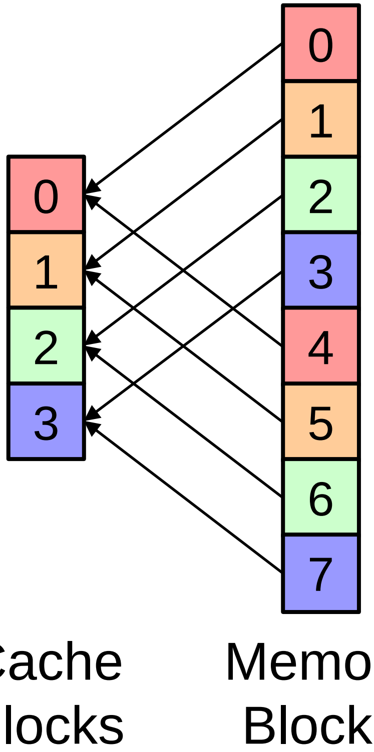
- Assume 2-way set associative mapping is used, and all CBs are empty initially.
- Considering CPU is looking for $(\mathbf{8010})_{16}$:
  - Which MB will be loaded into the cache?
  - Which CB will store the MB?
  - What is the new tag for the CB?



Summary of Mapping Functions (1/2)

Direct

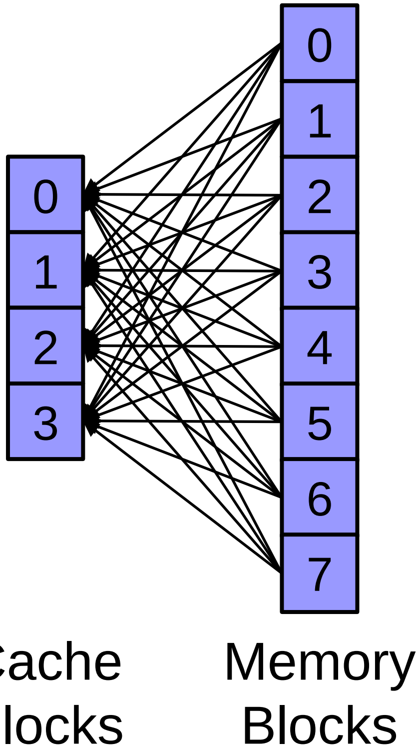
A Memory Block is **directly mapped** (%) to a Cache Block.



Associative

A Memory Block can be **mapped to any** Cache Block.

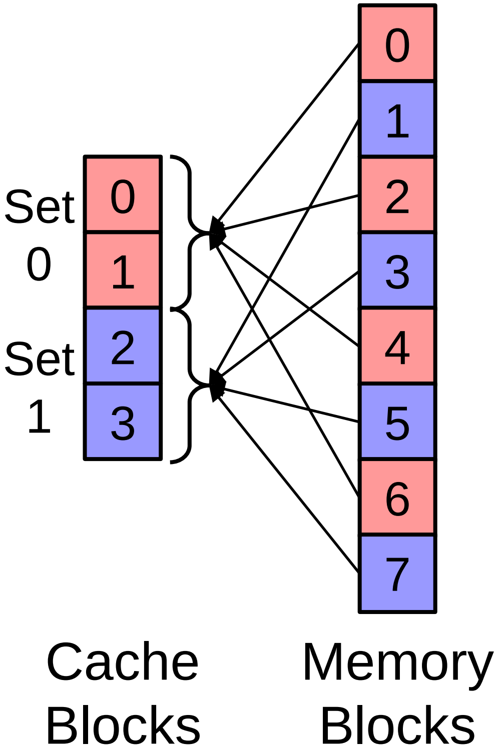
(First come first serve!)



Set Associative

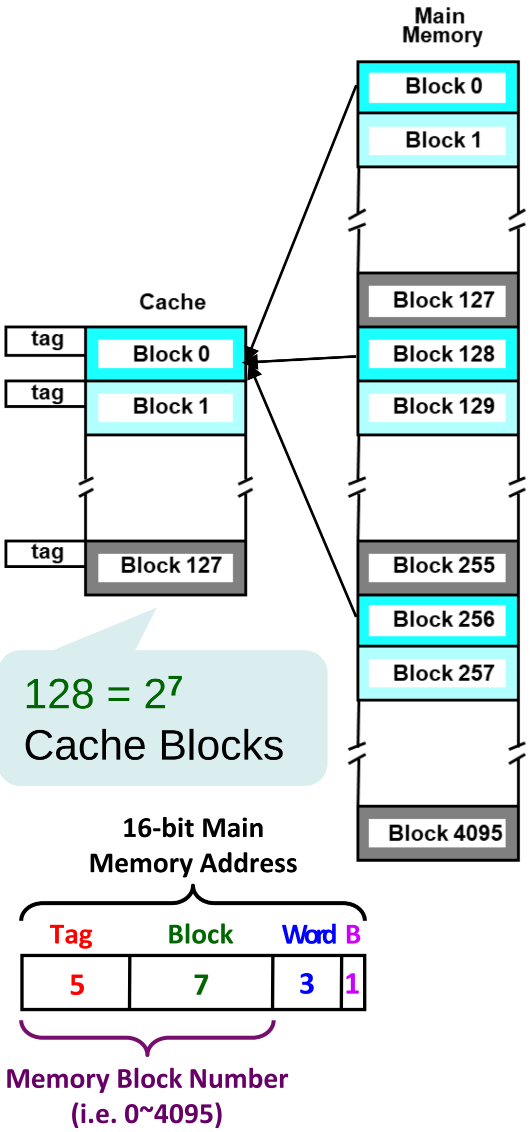
A Memory Block is **directly mapped** (%) to a **Cache Set**.

In a **Set**? **Associative**!



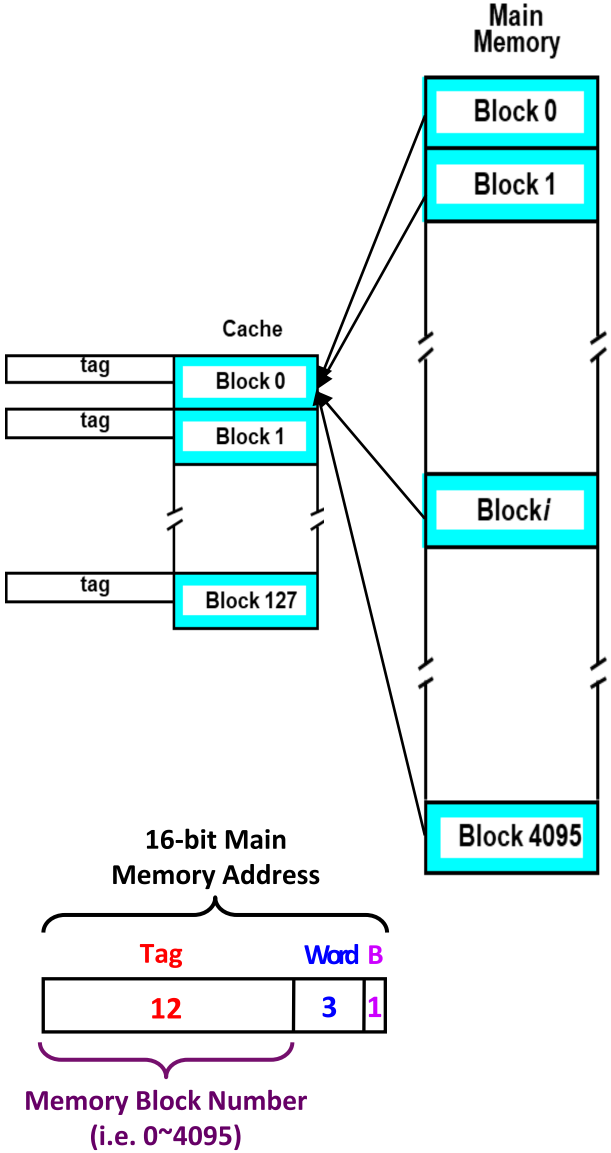
Summary of Mapping Functions (2/2)

Direct



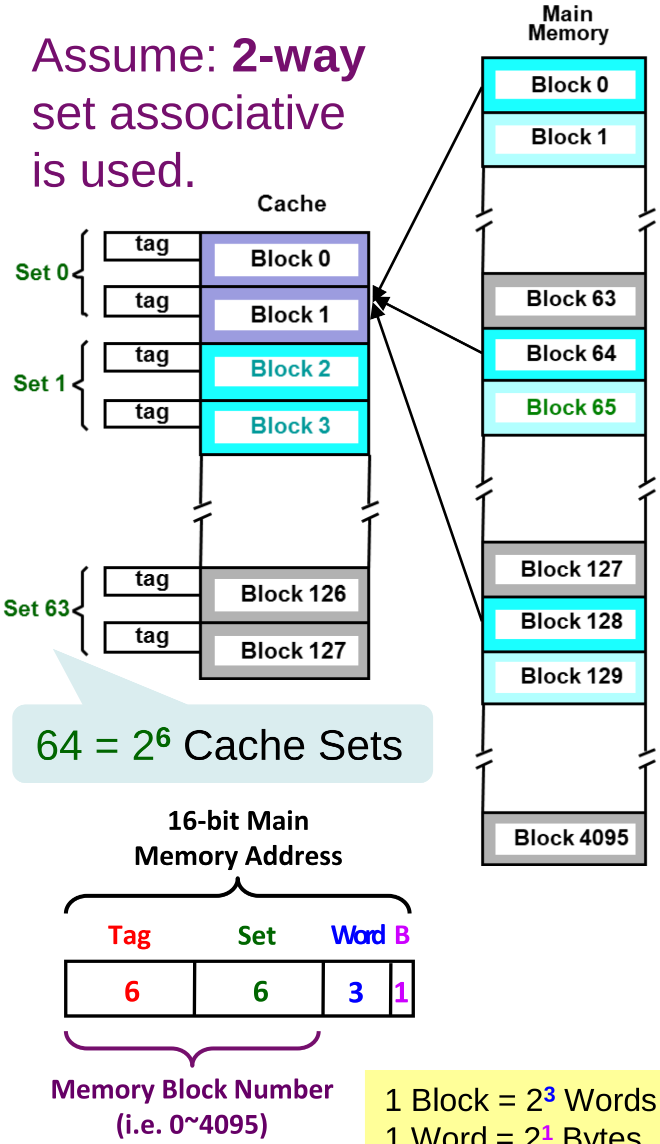
$128 = 2^7$
Cache Blocks

Associative



Set Associative

Assume: **2-way** set associative is used.



$64 = 2^6$ Cache Sets

1 Block = 2^3 Words
1 Word = 2^1 Bytes

Outline

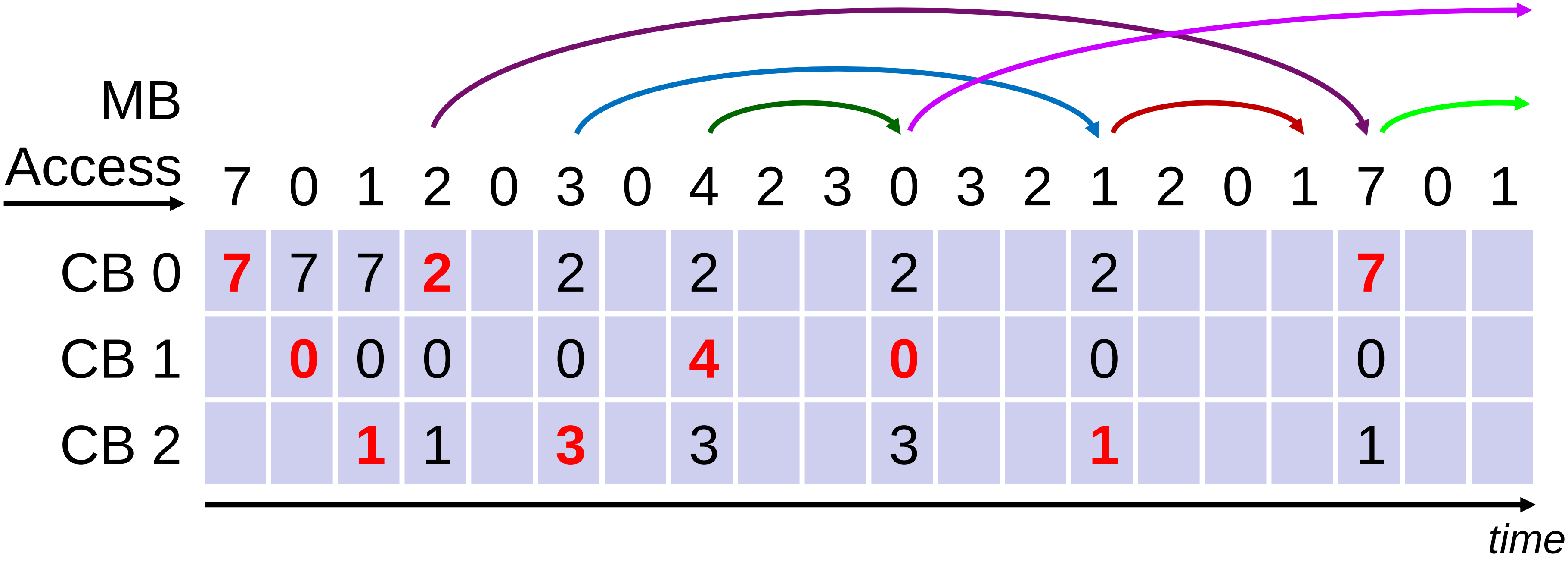
- Cache Basics
- Mapping Functions
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping
- Replacement Algorithms
  - Optimal Replacement
  - Least Recently Used (LRU) Replacement
  - Random Replacement
- Working Examples

Replacement Algorithms

- **Replace**: Write Back (to old MB) & Overwrite (with new MB)
- **Direct Mapped Cache**:
  - The CB is pre-determined directly by the memory address.
  - The replacement strategy is trivial: Just replace the pre-determined CB with the new MB.
- **Associative and Set Associative Mapped Cache**:
  - Not trivial: Need to determine which block to replace.
    - **Optimal Replacement**: Always keep CBs, which will be used sooner, in the cache, if we can look into the future (not practical!!!).
    - **Least recently used (LRU)**: Replace the block that has gone the longest time without being accessed by looking back to the past.
      - Rationale: Based on temporal locality, CBs that have been referenced recently will be most likely to be referenced again soon.
    - **Random Replacement**: Replace a block randomly.
      - Easier to implement than LRU, and quite effective in practice.

Optimal Replacement Algorithm

- **Optimal Algorithm**: Replace the CB that will not be used for the longest period of time (in the **future**).
- Given an associative mapped cache, which is composed of 3 Cache Blocks (CBs 0~2).

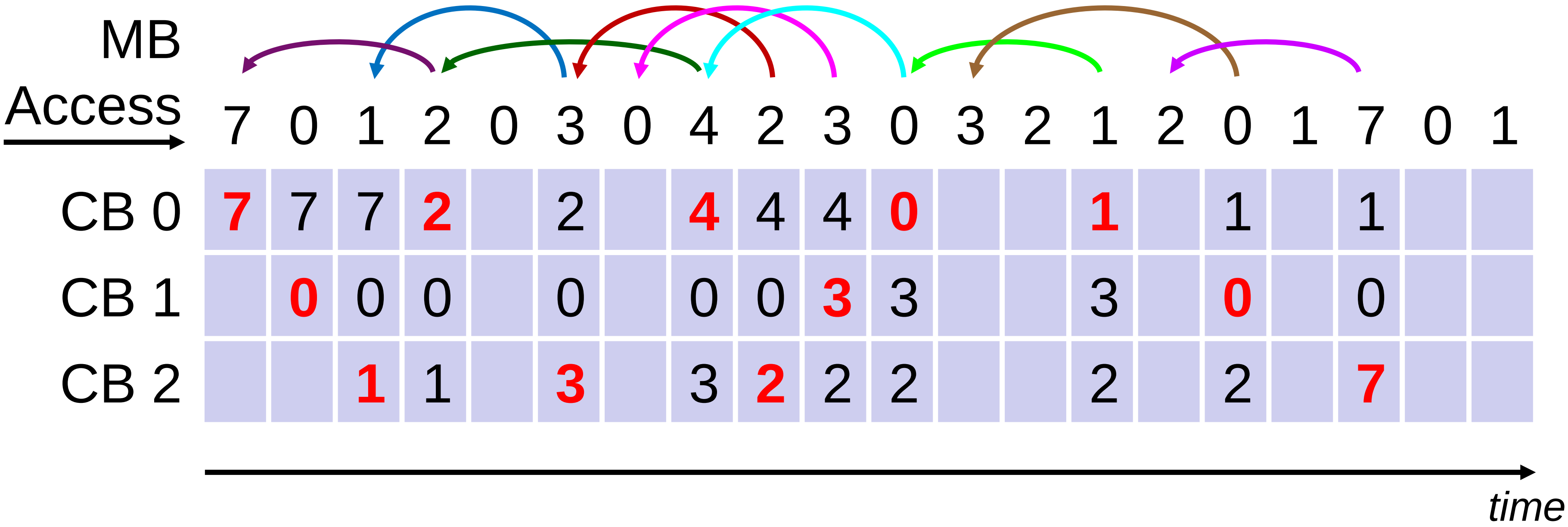


MB Access	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
CB 0	**7**	7	7	**2**		2		2			2			2				**7**		
CB 1		**0**	0	0		0		**4**			**0**			0				0		
CB 2			**1**	1		**3**		3			3			**1**				1		

  - The optimal algorithm causes **9** times of cache misses.

LRU Replacement Algorithm

- **LRU Algorithm**: Replace the CB that has not been used for the longest period of time (in the **past**).
- Given an associative mapped cache, which is composed of 3 Cache Blocks (CBs 0~2).

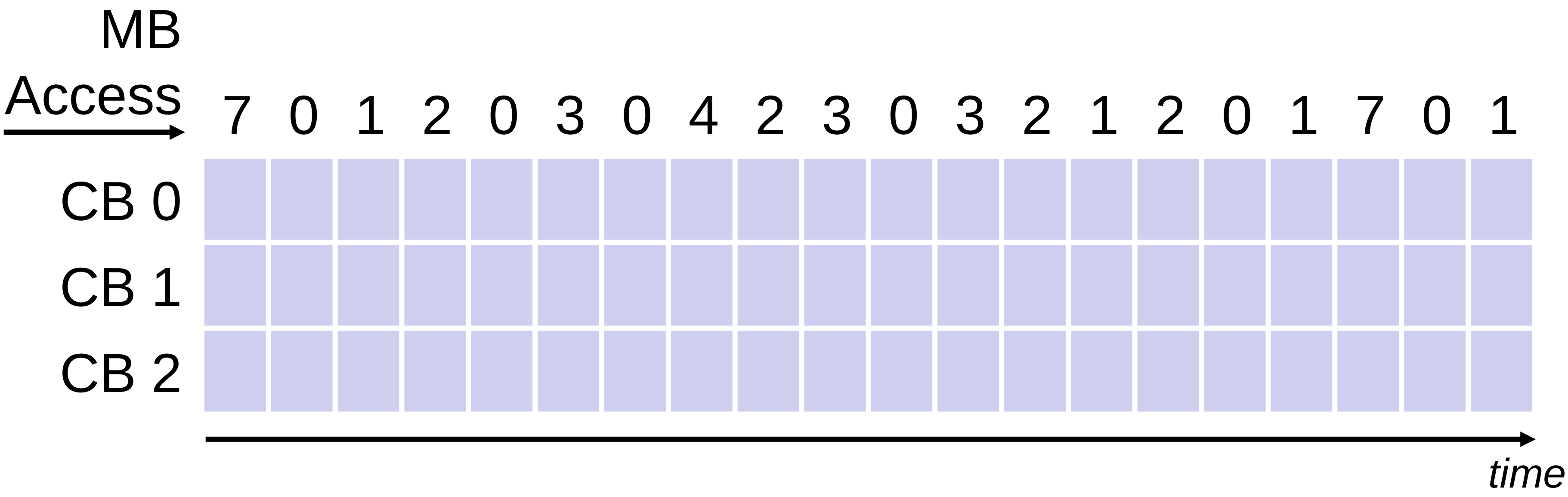


MB Access	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
CB 0	**7**	7	7	**2**		2		**4**	4	4	**0**			**1**		1		1		
CB 1		**0**	0	0		0		0	0	**3**	3			3		**0**		0		
CB 2			**1**	1		**3**		3	**2**	2	2			2		2		**7**		

  - The LRU algorithm causes **12** times of cache misses.

Class Exercise 7.5

- **First-In-First-Out Algorithm**: Replace the CB that has arrived for the longest period of time (in the **past**).
- Given an associative mapped cache, which is composed of 3 Cache Blocks (CBs 0~2).
- Please fill in the cache and state cache misses.



MB Access	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
CB 0																				
CB 1																				
CB 2																				

Outline

- Cache Basics

- Mapping Functions
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping

- Replacement Algorithms
  - Optimal Replacement
  - Least Recently Used (LRU) Replacement
  - Random Replacement

- Working Examples

Cache Example

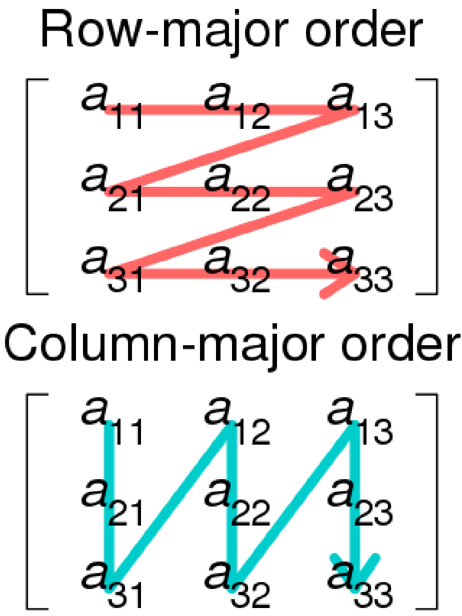
- Cache Configuration:
  - Cache has 8 blocks.
  - A block is of 1 (= 2^0) word.
  - A word is of 16 bits.

```
short  A[10][4];
int    sum = 0;
int    j, i;
double mean;

// 1) forward loop
for (j = 0; j <= 9; j++)
  sum += A[j][0];
mean = sum / 10.0;

// 2) backward loop
for (i = 9; i >= 0; i--)
  A[i][0] = A[i][0] / mean;
```

- Consider a program:
  1) Computes the sum of the first column of an array using a forward loop.
  2) Normalizes the first column of an array by its mean (i.e. average) using a backward loop.
  - **A[10][4]** is an array of words located at memory $(7A00)_{16}$~$(7A27)_{16}$ in **row-major** order.

Row-Major vs. Column-Major Order

- Row-major order and column-major order are methods for storing multidimensional arrays in memory.
  - **Row-Major**: The consecutive elements of a **row** reside next to each other.
  - **Column-Major**: The consecutive elements of a **column** reside next to each other.



- For example,

Values as stored in Memory:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Column major: $\begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$ Row major: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$

https://en.wikipedia.org/wiki/Row-_and_column-major_order

Cache Example (Cont'd)

`A[10][4];`
at $(7A00)_{16}$~$(7A27)_{16}$
in **row-major order**.

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[3][0]	A[3][1]	A[3][2]	A[3][3]
A[4][0]	A[4][1]	A[4][2]	A[4][3]
A[5][0]	A[5][1]	A[5][2]	A[5][3]
…	…	…	…

Program

j i first column

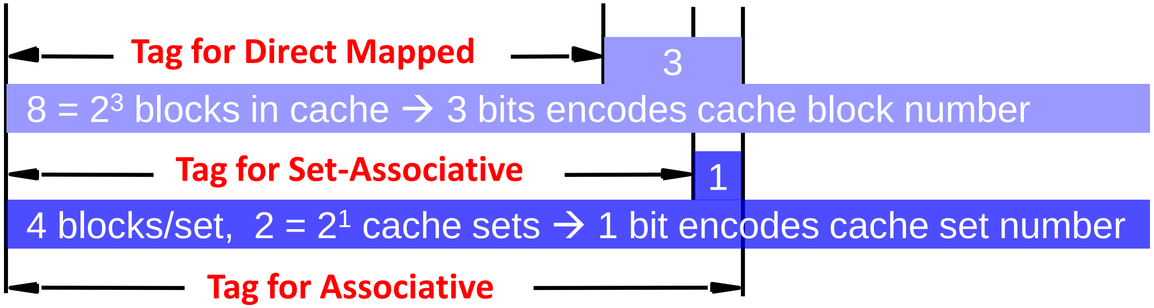
A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

Memory Block/Word Address (15-bit)

Hex.	Binary
$(7A00)_{16}$	**$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ X)_2$**
$(7A01)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ X)_2$
$(7A02)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ X)_2$
$(7A03)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ X)_2$
$(7A04)_{16}$	**$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ X)_2$**
⋮	⋮
$(7A24)_{16}$	**$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ X)_2$**
$(7A25)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ X)_2$
$(7A26)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ X)_2$
$(7A27)_{16}$	$(1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ X)_2$

Memory Contents
(40 array elements)

A[0][0]
A[0][1]
A[0][2]
A[0][3]
A[1][0]
⋮
A[9][0]
A[9][1]
A[9][2]
A[9][3]



Tag: 12 bits — **Tag for Direct Mapped** — 3

$8 = 2^3$ blocks in cache → 3 bits encodes cache block number

Tag: 14 bits — **Tag for Set-Associative** — 1

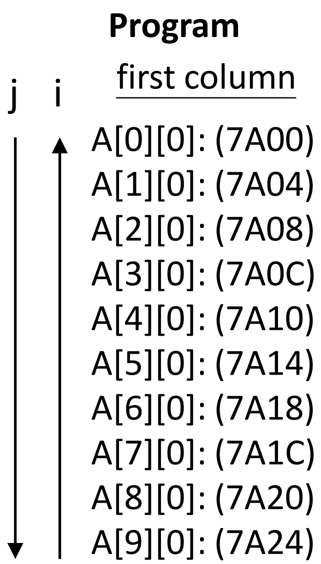
4 blocks/set, $2 = 2^1$ cache sets → 1 bit encodes cache set number

Tag: 15 bits — **Tag for Associative**

- A block is of 2^0 word: There is **no** "word" bit.
- A word is of 2^1 bytes: There is **one** "byte" bit (**X**).

Direct Mapping

- The last 3-bits of address decide the CB.
  - Memory Block Num. % 8 → Cache Block Num.
- No replacement algorithm is needed.
- When i = 9 and i = 8: **2** cache hits in total.
- Only 2 out of the 8 cache positions are used.
  - Very poor cache utilization: 25%



Program

first column

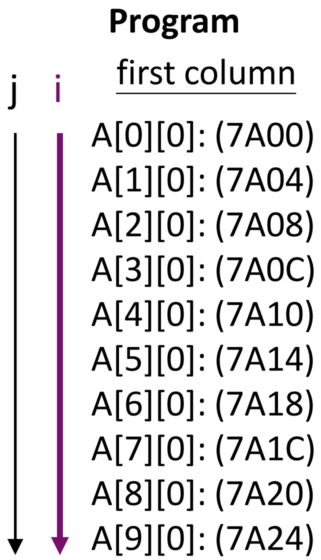
A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

		Content of Cache Blocks after Loop Pass (i.e. Timeline)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Cache Block Number	**0**	A[0][0]	A[0][0]	A[2][0]	A[2][0]	A[4][0]	A[4][0]	A[6][0]	A[6][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[6][0]	A[6][0]	A[4][0]	A[4][0]	A[2][0]	A[2][0]	A[0][0]
	1																				
	2																				
	3																				
	4		A[1][0]	A[1][0]	A[3][0]	A[3][0]	A[5][0]	A[5][0]	A[7][0]	A[7][0]	A[9][0]	A[9][0]	A[9][0]	A[7][0]	A[7][0]	A[5][0]	A[5][0]	A[3][0]	A[3][0]	A[1][0]	A[1][0]
	5																				
	6																				
	7																				

Tags not shown but are needed

Class Exercise 7.6

- Assume direct mapped cache is used.
- What if the *i* loop is a forward loop?



Program

first column

j i

A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

		Content of Cache Blocks after Loop Pass (i.e. Timeline)																			
		j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
Cache Block Number	**0**	A[0][0]	A[0][0]	A[2][0]	A[2][0]	A[4][0]	A[4][0]	A[6][0]	A[6][0]	A[8][0]	A[8][0]										
	1																				
	2																				
	3																				
	4		A[1][0]	A[1][0]	A[3][0]	A[3][0]	A[5][0]	A[5][0]	A[7][0]	A[7][0]	A[9][0]										
	5																				
	6																				
	7																				

Tags not shown but are needed

Associative Mapping

- All CBs are used in the FCFS basis.
- LRU replacement policy is used.
- When i = 9, 8, …, 2: **8** cache hits in total.
- 8 out of the 8 cache positions are used.
  - Optimal cache utilization: 100%

Program

first column

j i

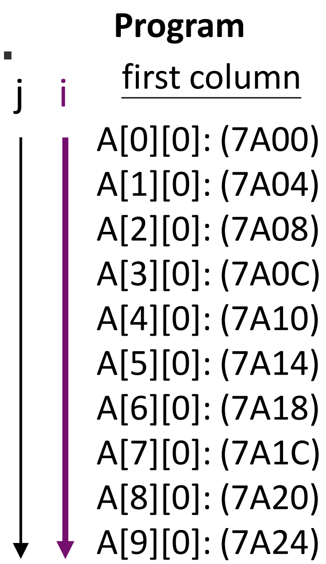
A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

		Content of Cache Blocks after Loop Pass (i.e. Timeline)																			
		j = 0	**j = 1**	**j = 2**	**j = 3**	**j = 4**	**j = 5**	**j = 6**	**j = 7**	**j = 8**	**j = 9**	**i = 9**	**i = 8**	**i = 7**	**i = 6**	**i = 5**	**i = 4**	**i = 3**	**i = 2**	**i = 1**	**i = 0**
Cache Block Number	**0**	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
	4					A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]
	5						A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]
	6							A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]
	7								A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]

Tags not shown but are needed

Class Exercise 7.7

- Assume associative mapped cache is used.
- What if the *i* loop is a forward loop?



Program

first column

A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

		Content of Cache Blocks after Loop Pass (i.e. Timeline)																			
		j = 0	**j = 1**	**j = 2**	**j = 3**	**j = 4**	**j = 5**	**j = 6**	**j = 7**	**j = 8**	**j = 9**	**i = 0**	**i = 1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
Cache Block Number	**0**	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[8][0]	A[8][0]										
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[9][0]										
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[2][0]										
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]										
	4					A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]										
	5						A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]										
	6							A[6][0]	A[6][0]	A[6][0]	A[6][0]										
	7								A[7][0]	A[7][0]	A[7][0]										

Tags not shown but are needed

4-way Set Associative Mapping

- There are total $8 \div 4 = 2$ Cache Sets.
  - Memory Block Num. % 2 → Cache Set Num.
- The numbers of accessed MBs are all “even” (e.g. 7A00, 7A04) → Mapped to **Cache Set #0**.
- LRU replacement policy is used.
- When i = 9, 8, …, 6: **4** cache hits in total.
- 4 out of the 8 cache positions are used (50% Util.).

Program

first column

j ↓ i ↑

A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

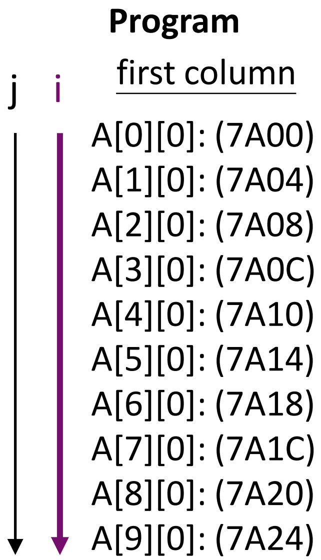
Content of Cache Blocks after Loop Pass (i.e. Timeline)

	CB #	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7	j = 8	j = 9	i = 9	i = 8	i = 7	i = 6	i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
Set 0	0	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[8][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[0][0]
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[9][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[1][0]	A[1][0]
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]	A[2][0]	A[2][0]	A[2][0]
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[7][0]	A[3][0]	A[3][0]	A[3][0]	A[3][0]
Set 1	4																				
	5																				
	6																				
	7																				

Tags not shown but are needed

Class Exercise 7.8

- Assume 4-way set associative mapped cache is used.
- What if the *i* loop is a forward loop?



Program

first column

A[0][0]: (7A00)
A[1][0]: (7A04)
A[2][0]: (7A08)
A[3][0]: (7A0C)
A[4][0]: (7A10)
A[5][0]: (7A14)
A[6][0]: (7A18)
A[7][0]: (7A1C)
A[8][0]: (7A20)
A[9][0]: (7A24)

		Content of Cache Blocks after Loop Pass (i.e. Timeline)																			
	CB #	**j = 0**	**j = 1**	**j = 2**	**j = 3**	**j = 4**	**j = 5**	**j = 6**	**j = 7**	**j = 8**	**j = 9**	**i = 0**	**i = 1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
Set 0	**0**	A[0][0]	A[0][0]	A[0][0]	A[0][0]	A[4][0]	A[4][0]	A[4][0]	A[4][0]	A[8][0]	A[8][0]										
	1		A[1][0]	A[1][0]	A[1][0]	A[1][0]	A[5][0]	A[5][0]	A[5][0]	A[5][0]	A[9][0]										
	2			A[2][0]	A[2][0]	A[2][0]	A[2][0]	A[6][0]	A[6][0]	A[6][0]	A[6][0]										
	3				A[3][0]	A[3][0]	A[3][0]	A[3][0]	A[7][0]	A[7][0]	A[7][0]										
Set 1	**4**																				
	5																				
	6																				
	7																				

Tags not shown but are needed

Summary

- Cache Basics

- Mapping Functions
  - Direct Mapping
  - Associative Mapping
  - Set Associative Mapping

- Replacement Algorithms
  - Optimal Replacement
  - Least Recently Used (LRU) Replacement
  - Random Replacement

- Working Examples