CHAPTER
7

MEMORY HIERARCHY

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

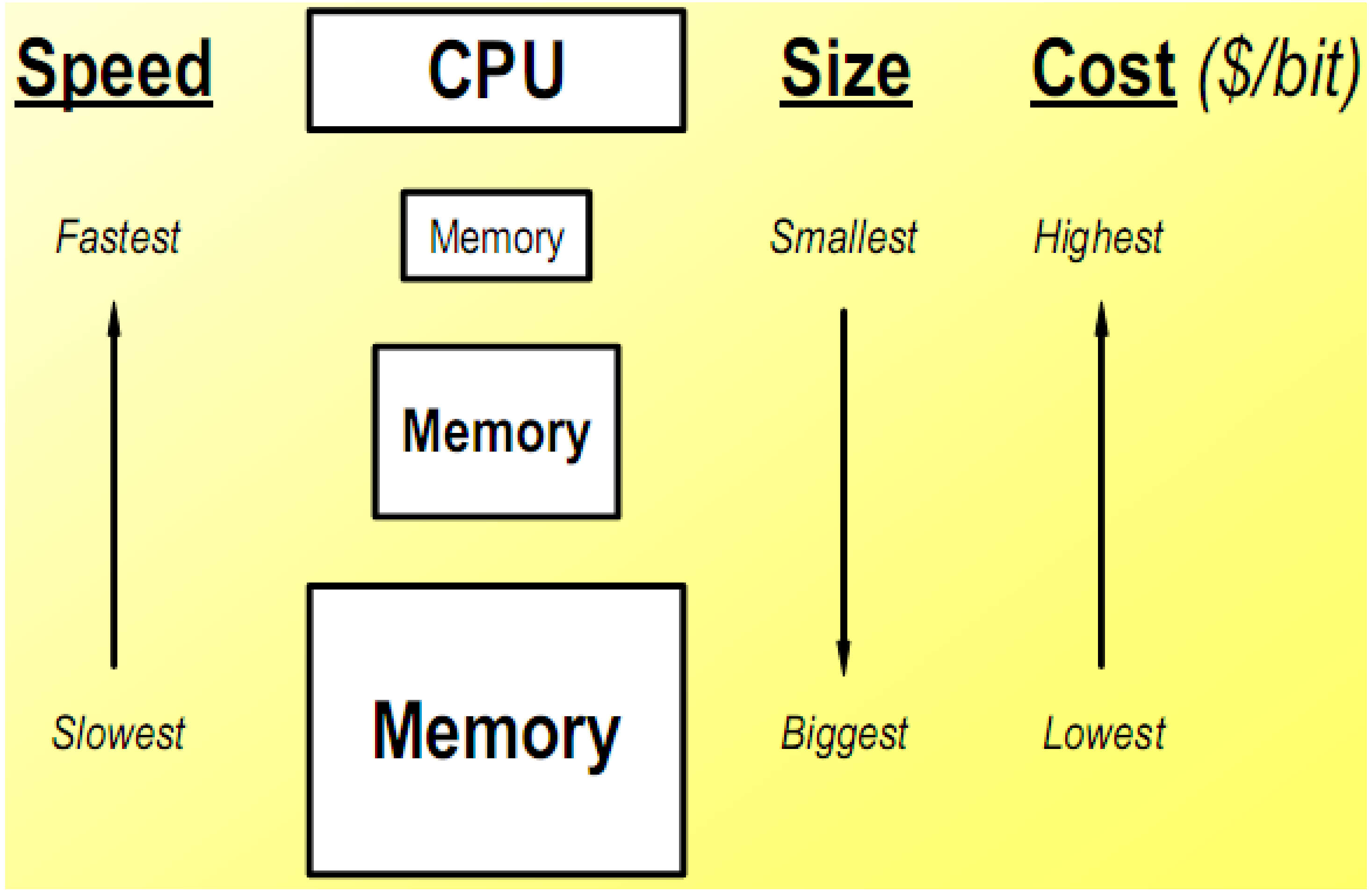
fit@hcmus

fit@hcmus

What will you learn?

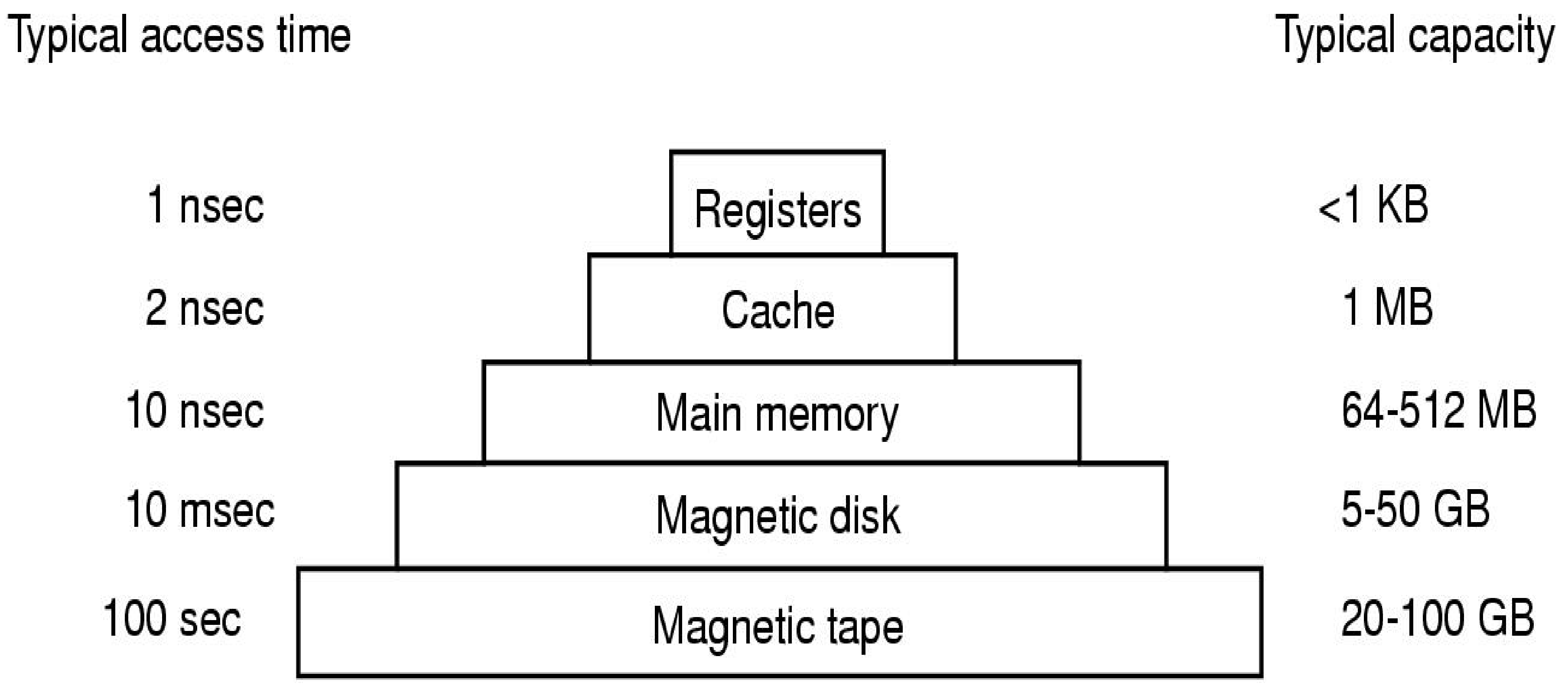
- ☐ Memory hierarchy levels
- ☐ Memory Technology
- ☐ Principle of locality
- ☐ Cache Memory
- ☐ Cache addressing scheme
- ☐ Replacement Policy
- ☐ Write Policy
- ☐ Parameter influent to Cache performance
- ☐ Interactions with advanced CPUs & Software
- ☐ Writing cache-friendly code

Memory Hierarchy Levels



Memory Hierarchy Levels

Example:



Memory Technology

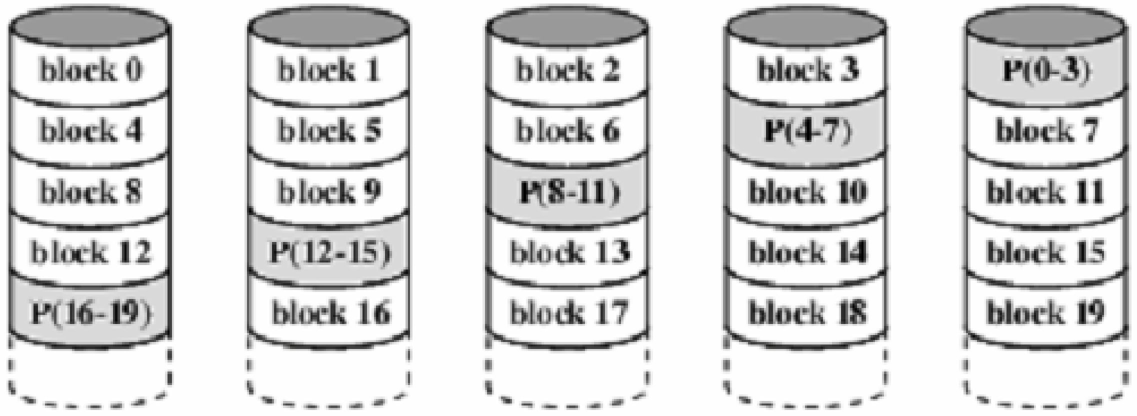
- Access types:
  - Serial/Sequential
  - Direct
  - Random
- Physical types:
  - Transistor (cache, register, RAM, ROM)
  - Magnetic disk (HDD, FDD)
  - Flash (CD, DVD, SD, SSD)

Mass storage device: RAID

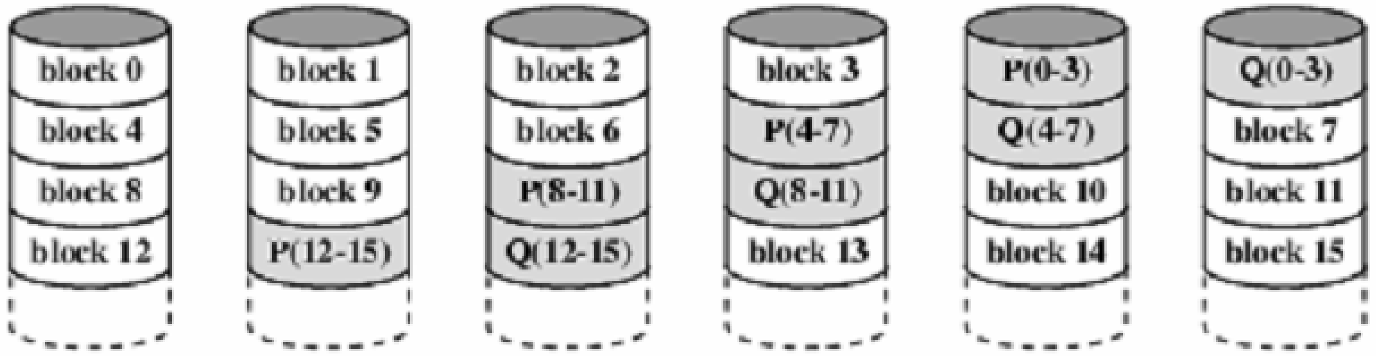
- ☐ Redundant Array of Inexpensive (Independent) Disks
- ☐ A data storage technology that combines multiple physical disk drive components into one or more logical units
- → Storing data in distributed physical disk
- → Using parity bits/ check byte to check data errors
- ☐ The common RAID system: RAID 0 → RAID 6

fit@hcmus

Mass storage device: RAID



(f) RAID 5 (block-level distributed parity)



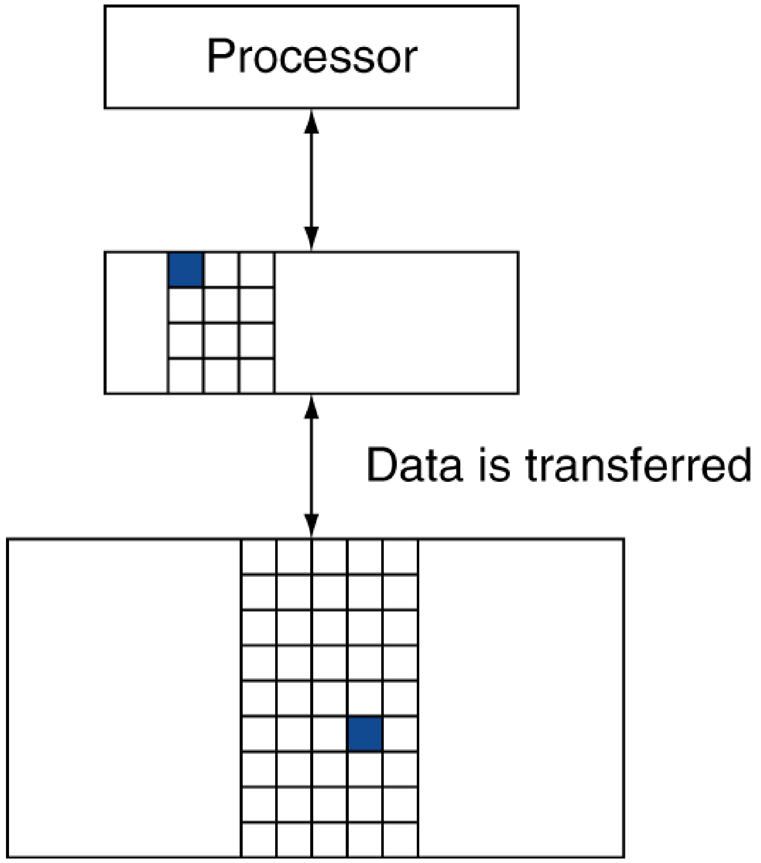
(g) RAID 6 (dual redundancy)

Principle of Locality

- Programs access a small portion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

fit@hcmus

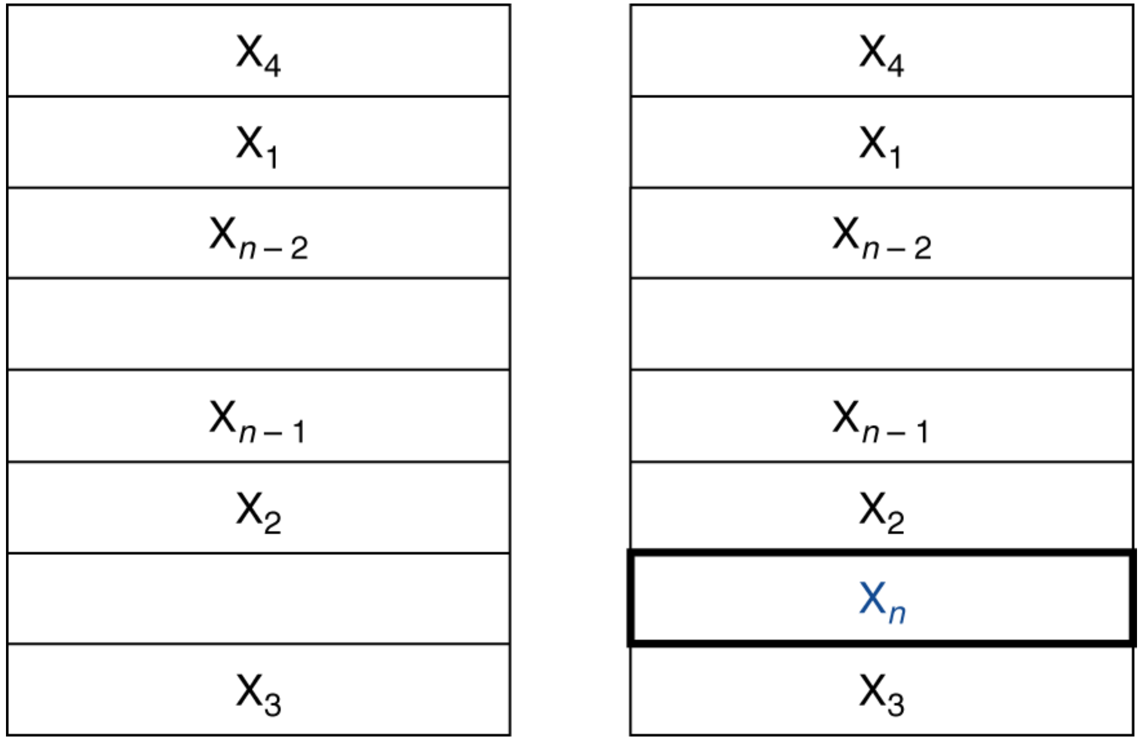
Hit and Miss



- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - $\text{Hit ratio} = \frac{hits}{accessed}$
- If accessed data is absent
  - Miss: block copied from lower level
    - $\text{Miss ratio} = \frac{misses}{accessed} = 1 - hit\ ratio$

fit@hcmus

Cache Memory



a. Before the reference to X_n b. After the reference to X_n

- Using SRAM technology which has higher access speed than main memory (using DRAM technology)
- The level of the memory hierarchy closest to the CPU
- A partial copy of the main memory
- Given access X_1, X_2, .., X_{n-1}, X_n

Reference to X_n causes miss so it is fetched from memory

Discussion

- When you need to access a memory cell, how do you know if that cell is already in the cache? If so, how to identify that place?

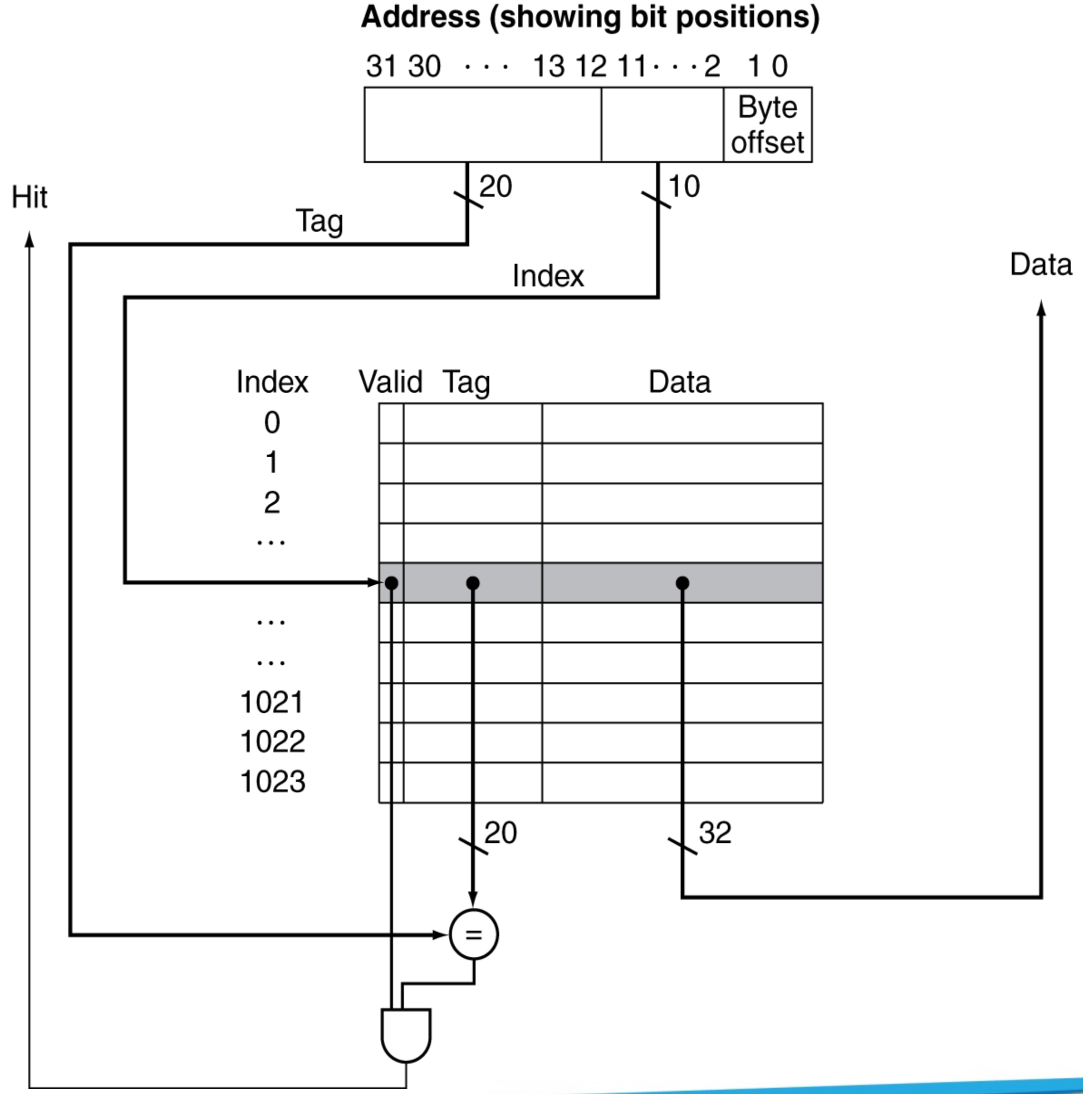
- Which memory cells will be selected for loading to cache? When does the selection happen?

cdio 4.0

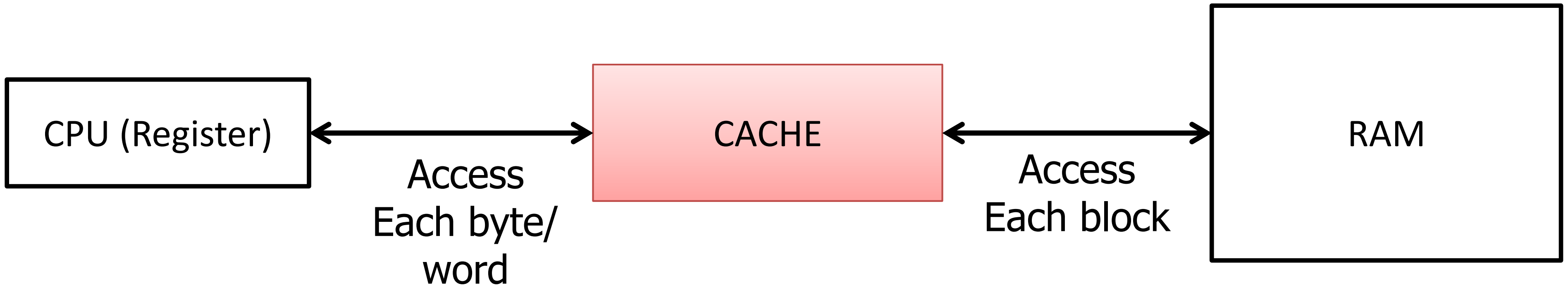
fit@hcmus

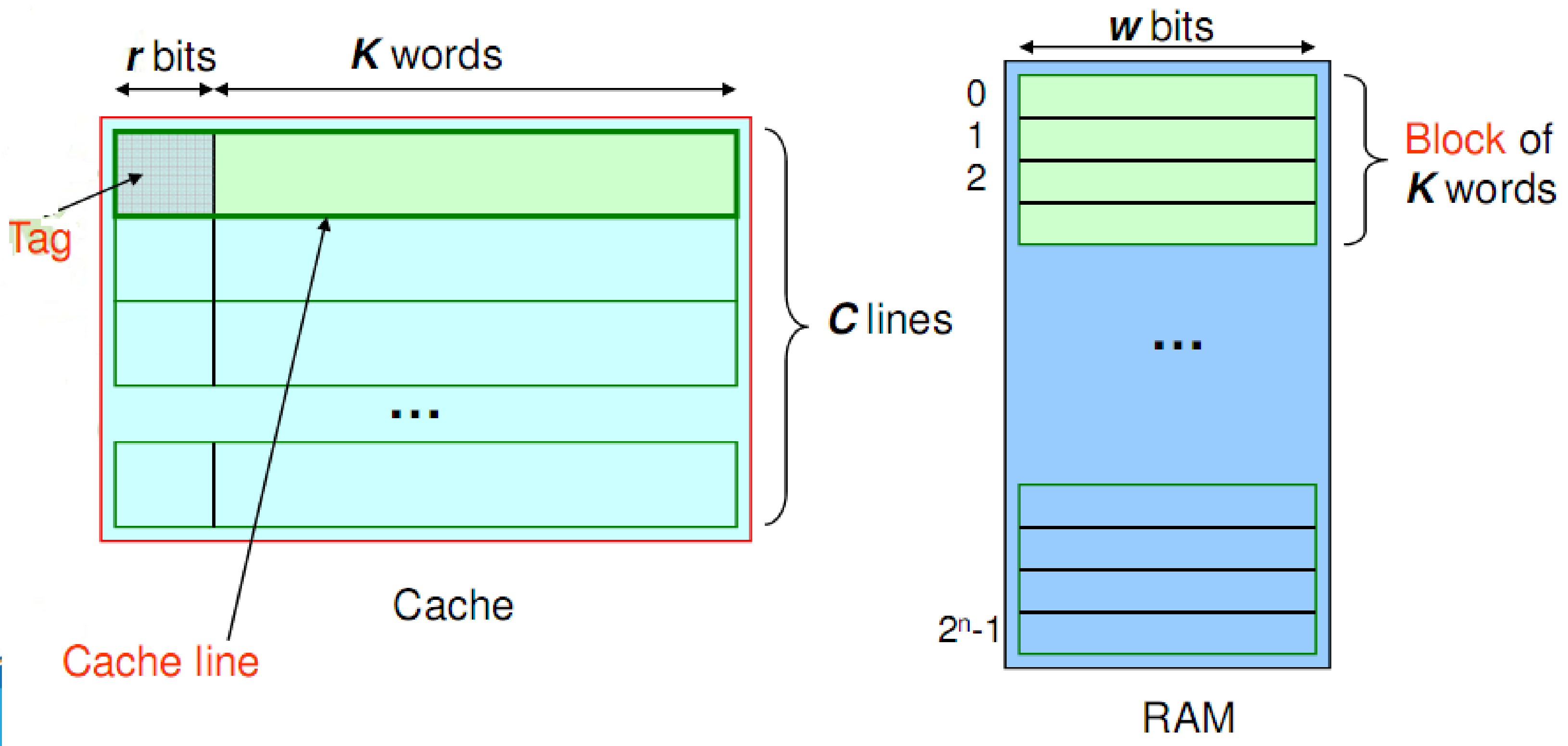
Cache Memory: Structure

Index	Valid	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Cache Access Method





Cache Access Method

- Main memory has 2^n bytes of memory, numbered $0 \rightarrow 2^n - 1$
- The main memory and the cache are divided into equal-sized blocks

1 block of main memory = 1 line in cache

- Some main memory blocks are loaded into lines of cache
- Tag content shows which block of main memory is currently stored on that line (*not the serial number of that line in the Cache*)

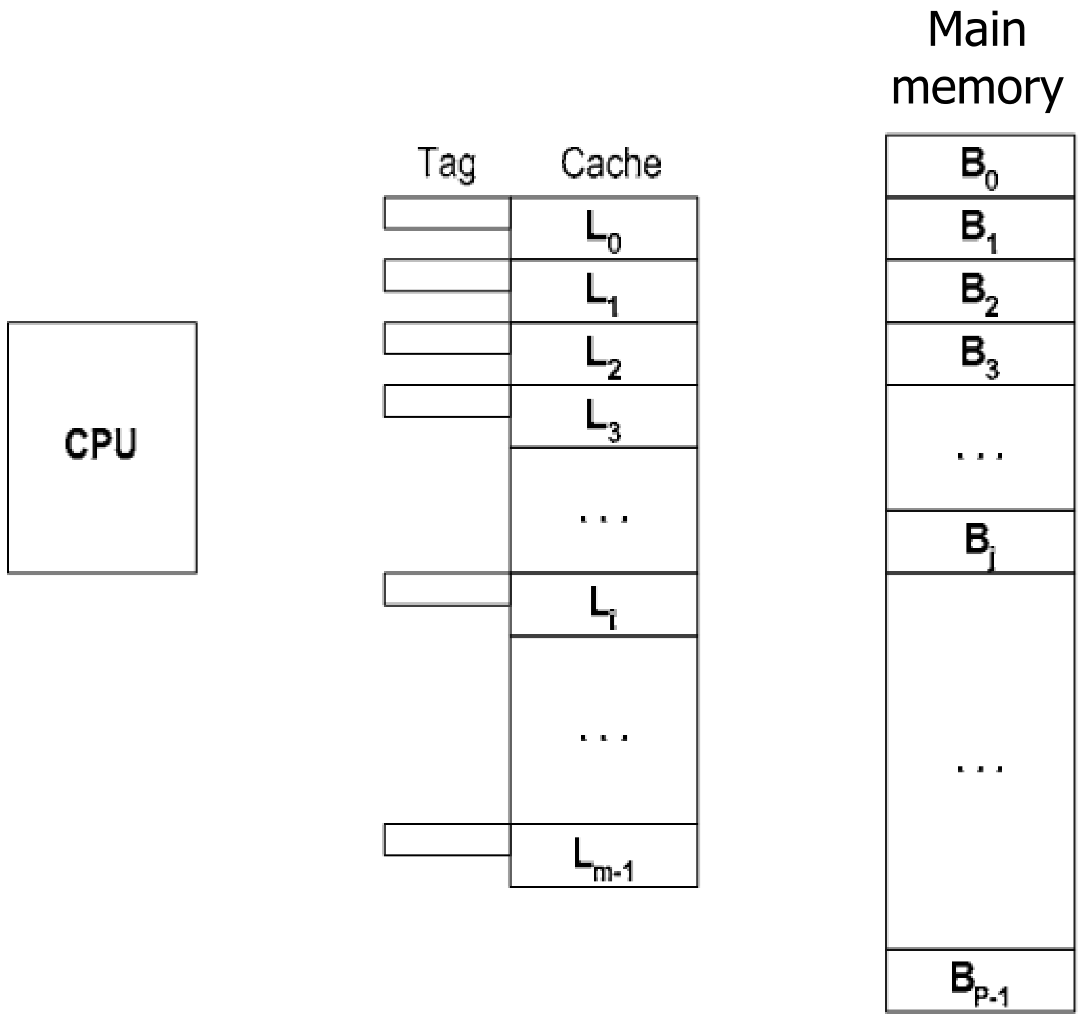
Cache Addressing Schemes

- Direct mapping
- Associative mapping (Full associative mapping)
- Set associative mapping

fit@hcmus

Direct Mapping

- Each block of main memory can only be loaded into 1 line of cache
  - $B_j \rightarrow L_{j \bmod m}$
  - m: the number of lines in cache

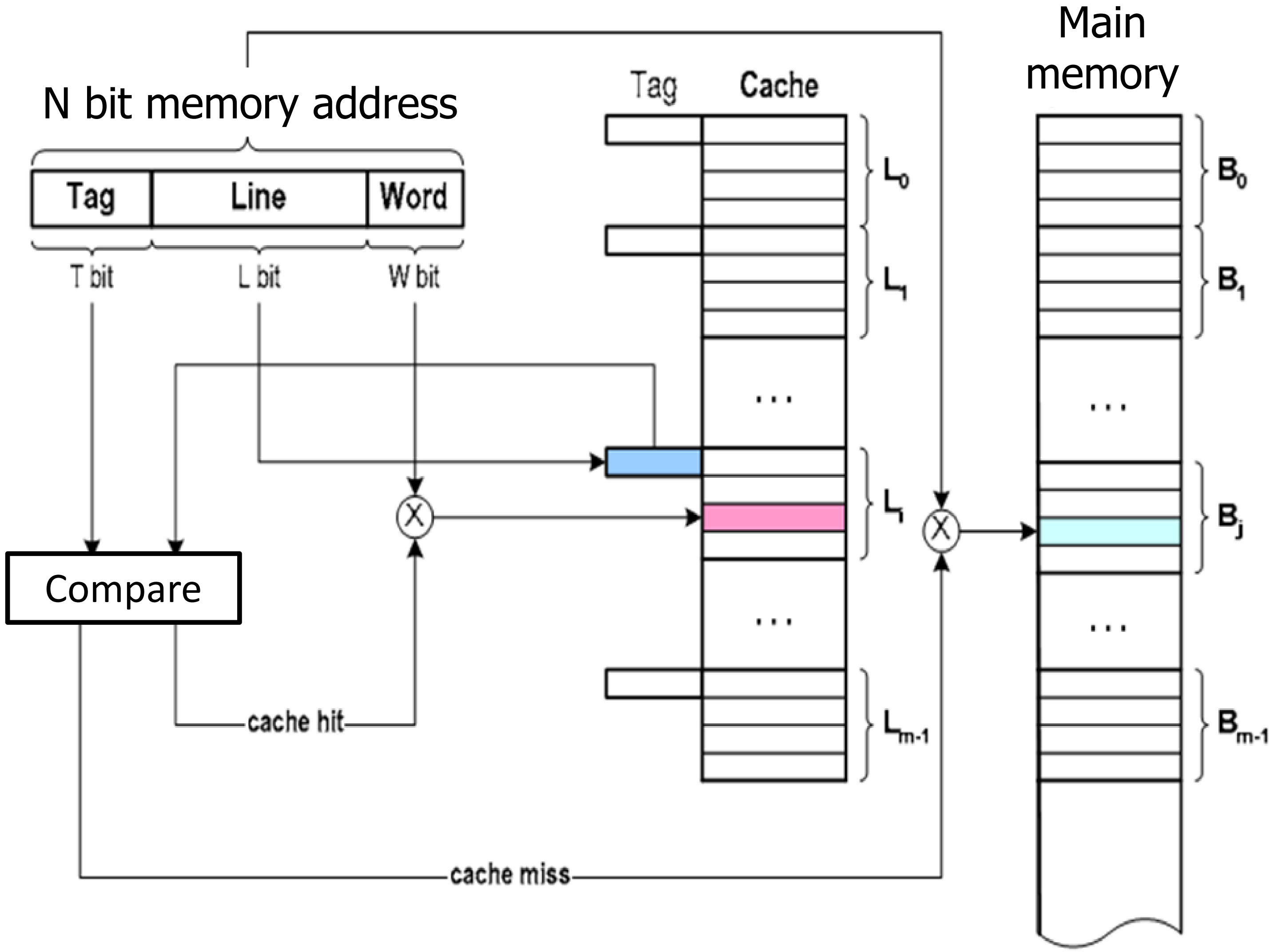


Direct Mapping

Tag	Line	Word

Each X address in main memory consists of N bits divided into three fields:

- Word (W-bits): Defines the word address in 1 block
- Line (L-bits): Defines the line address in cache
- Tag (T-bits): the block address



Direct Mapping: Example

Ex1: Suppose a direct-mapped cache has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the index (Line)?
- How many address bits are used for the tag?

Direct Mapping: Example

Solution:

- The size of main memory = 4GB = 2^{32} bytes → N = 32 bits
- The size of cache = 256KB = 2^{18} bytes → using 18 bits to address each memory word in cache
- The size of 1 line in cache = 32 = 2^5 bytes → W = 5 bits (using 5 bits to address each memory word in a line of cache)

$\rightarrow \textit{The number of lines in cache} = 2^{18}/2^5 = 2^{13}\ \textit{Lines}$ → L = 13 bits

- Tag = N – L – W = 32 – 13 – 5 = 14 bits

Tag	Line	Word
14 bits	13 bits	5 bits

Direct Mapping: Example

Ex2: Reuse the assumptions in example 1. Suppose that we have Mth Block (27 bits, value from 0 to 2^{27} - 1) that want to store in the cache. Where will it be stored in cache?

Solution:

$$\mathit{The\ number\ of\ blocks\ in\ main\ memory} = 2^{32}/2^{5} = 2^{27}$$

→ Using 27 bit to address 1 block in the main memory

The place of block M in cache :

Line: L = M % the number of lines in cache = M % 2^{13}

Tag: T = M / the number of lines in cache = M / 2^{13}

fit@hcmus

Direct Mapping

- ☐ Advantage:

Simple comparison

- ☐ Disadvantage:

Low probability of cache hit

Ex: Suppose that accessing the memory word (cell) X at Block 0 and cell Y at Block 2^L (2^L is the number of lines in cache) at the same time?

→Conflict, both of these cells will be saved in Line 0

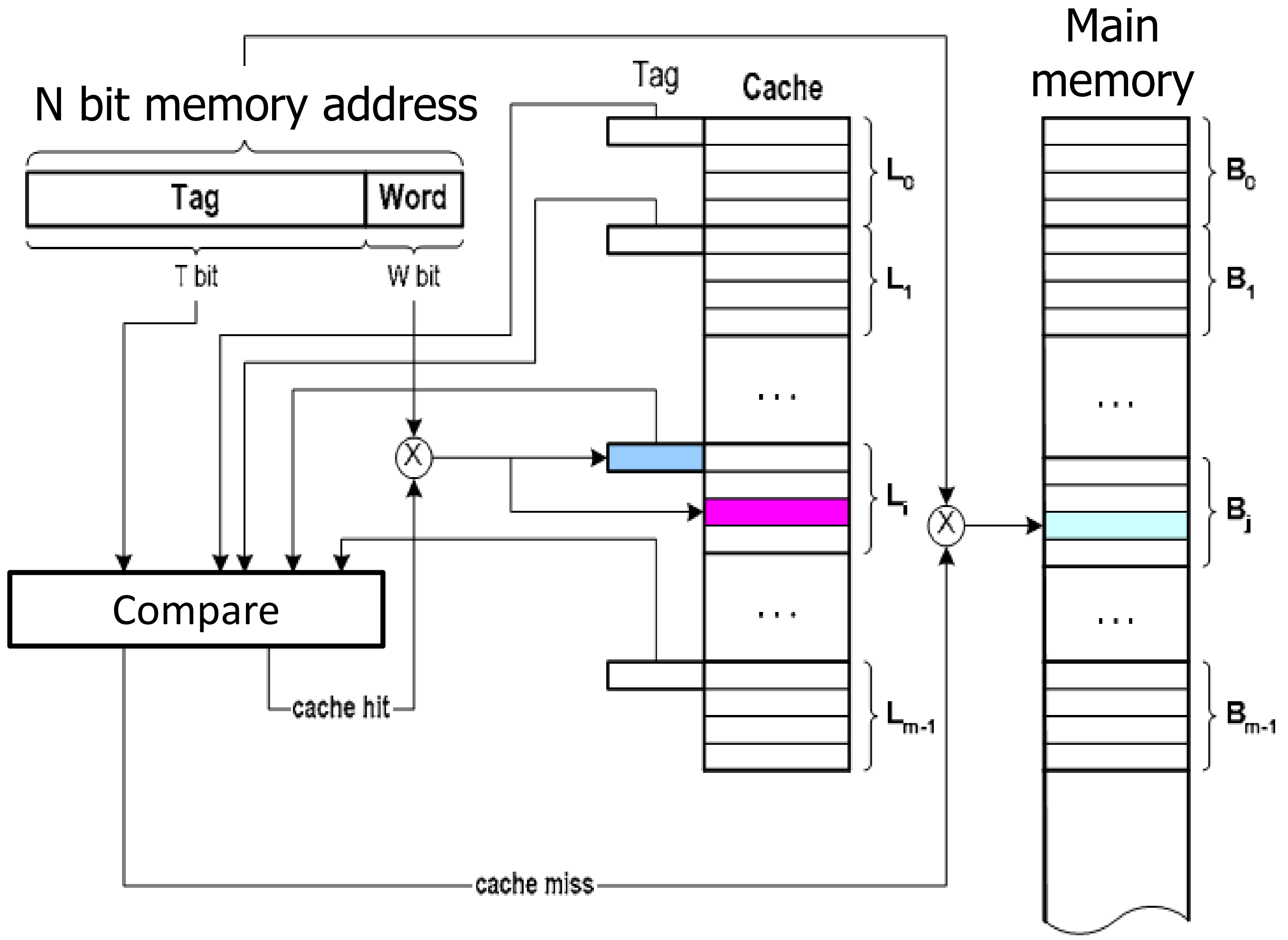
($0 \% 2^L = 2^L \% 2^L = 0$)

Associative Mapping

Tag	Word

Each X address in main memory consists of N bits divided into two fields:

- Word (W-bits): Defines the word address in 1 block
- Tag (T-bits): the block address (which block of main memory is placed in this line)



Direct Mapping: Example

Ex1: Suppose an associative-mapped cache has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the tag?

cdio 4.0
fit@hcmus

Associative Mapping: Example

Solution:

- The size of main memory = 4GB = 2^{32} bytes → N = 32 bits
- The size of 1 line in cache = 32 = 2^5 bytes → W = 5 bits (using 5 bits to address each memory word in a line of cache)
- Tag = N – W = 32 – 5 = 27 bits

Tag	Word
27 bits	**5 bits**

Direct Mapping: Example

Ex2: Reuse the assumptions in example 1. Suppose that we have M th Block (27 bits, value from 0 to 2^{27} - 1) that want to store in the cache. Where will it be stored in cache?

Solution:

$The\ number\ of\ blocks\ in\ main\ memory = 2^{32}/2^{5} = 2^{27}$

→ Using 27 bit to address 1 block in the main memory

The position of the M block in the cache is any line containing the tag equal to M

Tag: T = M

Associative Mapping

- ☐ Advantage:

High probability of cache hit

- ☐ Disadvantage:

Complex comparison

To find out which Line contains the content of 1 Block, we need to detect and compare in turn with the Tag of all Lines of the Cache

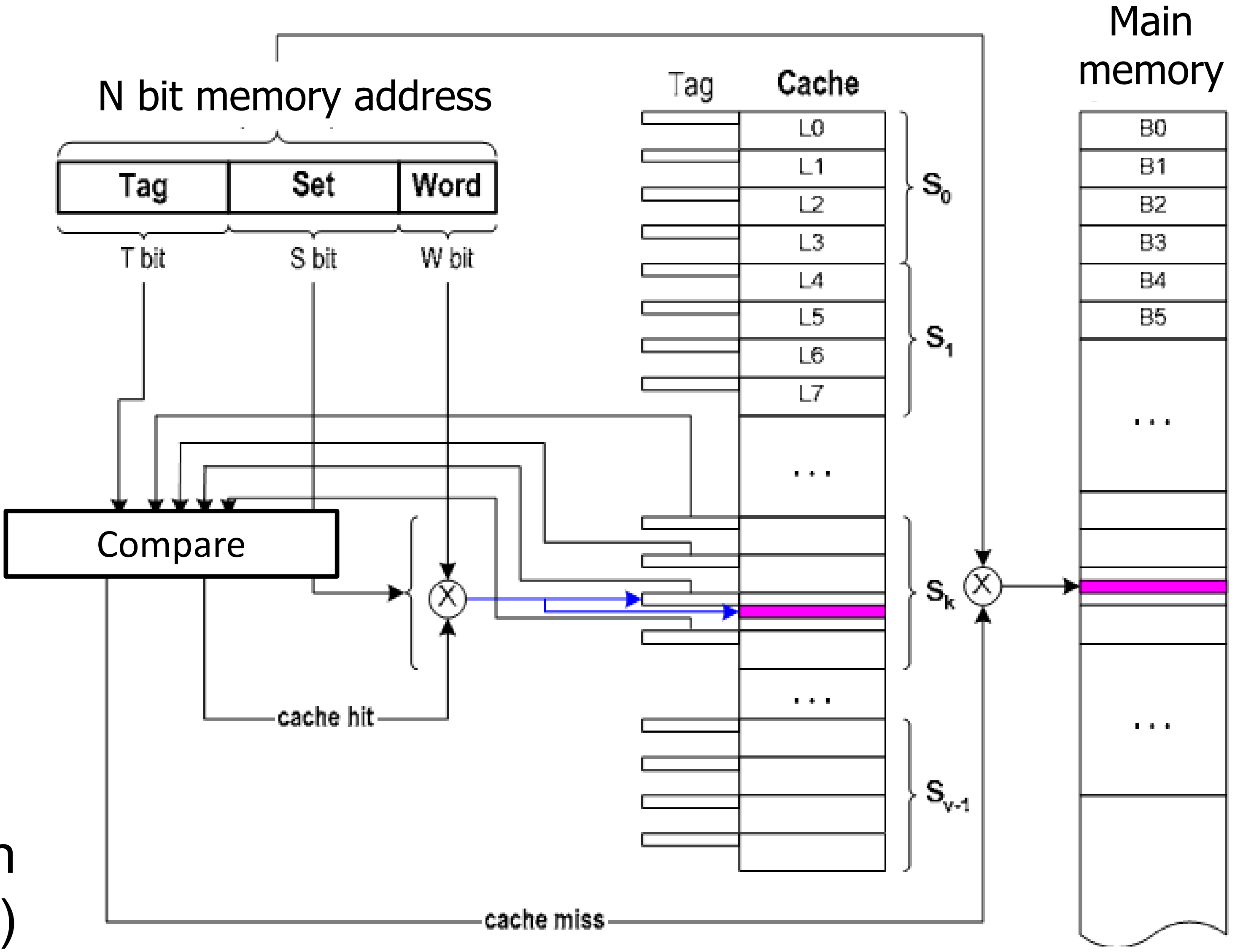
→ Takes a long time to compare

Set Associative Mapping

Tag	Set	Word

Each X address in main memory consists of N bits divided into three fields:

- Word (W-bits): Defines the word address in 1 block
- Set (S-bits): Defines the set address in cache, each set contents multiple lines
- Tag (T-bits): the block address (which block of main memory is placed in this line)



Set Associative Mapping: Example

Ex1: Suppose an 4-ways associative-mapped cache has 32-bytes blocks, The size of the cache and main memory are respectively are 256KB, 4GB. The machine has 32-bits addresses

- How many address bits are used for the byte offset (Word)?
- How many address bits are used for the set?
- How many address bits are used for the tag?

cdio 4.0
fit@hcmus

Set Associative Mapping: Example

Solution:

- The size of main memory = 4GB = 2^{32} bytes → N = 32 bits
- The size of cache = 256KB = 2^{18} bytes → using 18 bits to address each memory word in cache
- The size of 1 line in cache = 32 = 2^5 bytes → W = 5 bits (using 5 bits to address each memory word in a line of cache)

→ $\mathit{The\ number\ of\ lines\ in\ cache} = 2^{18}/2^5 = 2^{13}\ \mathit{Lines}$ → using 13 bits to address each line in cache

- The number of lines in each set is 4 (4-ways) = 2^2 lines

→ $\mathit{The\ number\ of\ sets\ in\ cache} = 2^{13}/2^2 = 2^{11}\ \mathit{Sets}$ → S = 11 bits (using 11 bits to address each set in cache)

- Tag = N – S – W = 32 – 11 – 5 = 16 bits

Tag 16 bits	Set 11 bits	Word 5 bits

Set Associative Mapping

- Advantage:

High probability of cache hit

Reduce time to compare

- Disadvantage:

Complex to implement → the higher cost

Replacement Algorithms

- ☐ Random
- ☐ FIFO (First In First Out)
- ☐ LFU (Least Frequently Used)
- ☐ LRU (Least Recently Used)

→ The optimal algorithm is LRU

Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
  - Using LRU algorithm is simple for 2-ways, difficult to manage 4-ways associative-mapped cache
  - Using random algorithm gives approximately the same performance as LRU for high associativity

cdio 4.0
fit@hcmus

Discussion

- If a line is changed in the cache, when will the RAM rewrite operation be performed?
- If multiple processors share RAM, each one has its own cache, which block will be updated on RAM?

Write Policy

- Write through
  - Update both upper and lower levels when the data has changed
  - Simplifies replacement, but may require write buffer
- Write back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual Memory
  - Only write-back is feasible, given disk write latency

Write Policy

- **Bus watching with Write through:**
  - removes the line when it is modified in another cache
- **Hardware transparency**
  - automatically update other caches when the line is changed by one cache
- **Noncacheable shared memory**
  - shared memory will not be put into cache

fit@hcmus

Multiple level cache

- Level cache:
  - Primary cache: focus on minimal hit time
  - L1: block size smaller than L2 block size
  - L2: focus on low miss rate to avoid main memory access
  - L3,...
- The low-level caches can be on-chip, while the high-level caches are usually off-chip and accessed via external bus or dedicated bus
- The cache can be used for both data and instruction or for each type

cdio 4.0
fit@hcmus

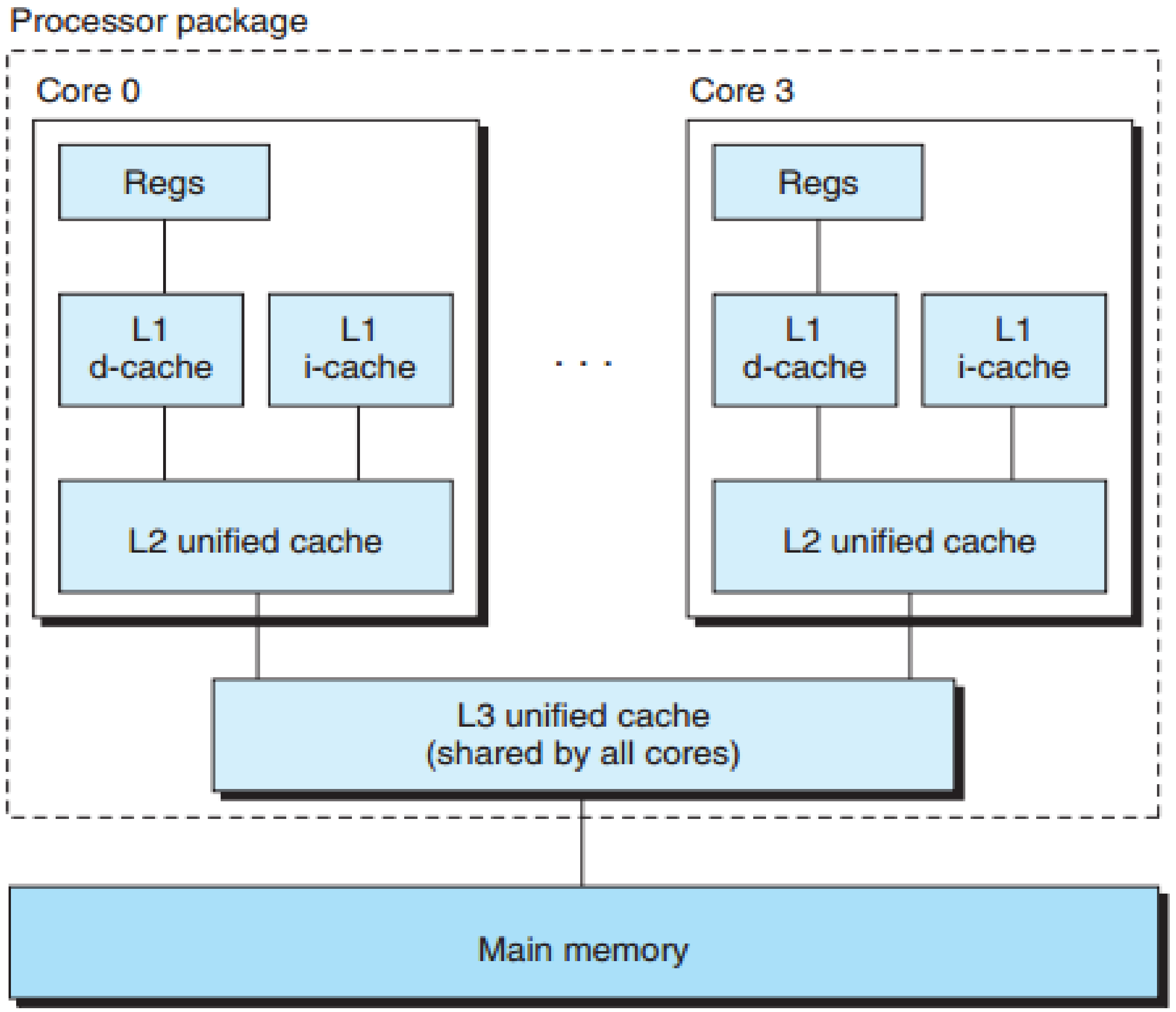
Parameter influent to Cache performance

- Block size:
  - Too small: decrease spatial locality
  - Too large: the number of blocks in the cache is small, the time to move the block to the cache is long (miss penalty)
- Cache size:
  - Too small: The volume that can be stored in the buffer is too small, increasing the rate of cache miss
  - Large too: The ratio of the actual memory needed to the cache memory will be lower, meaning overhead will be high, cache access speed will decrease.

Intel core i7 hierarchy

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	10	256 KB	8	64 B	512
L3 unified cache	40–75	8 MB	16	64 B	8,192

Characteristic of the Intel core i7 cache hierarchy



fit@hcmus

Interactions with advanced CPUs & Software

- With advanced CPUs:
  - Out-of-order CPUs can execute during cache miss
  - Effect of miss depends on program data flow
- With software:
  - Algorithm behavior
  - Compiler optimization for memory access

Writing cache-friendly code

- Make the common case go fast:
  - The core functions of the program
  - Especially the loops inside functions
- Minimize the number of cache misses in each inner loop:
  - The total number of loads and stores, loops with higher hit rates will run faster

→ Maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory

→ Maximize the spatial locality in your programs by reading data objects sequentially, in the order they are stored in memory

Writing cache-friendly code

- Example: Compare the following two code snippets

```
int sumarr(int a[M][N])
{
        int i, j, sum =0;
        for (i=0; i<M; i++)
                for(j=0; j<N; j++)
                        sum += a[i][j];
        return sum;
}
```

```
int sumarr(int a[M][N])
{
        int i, j, sum =0;
        for (j=0; j<N; j++)
                for(i=0; i<M; i++)
                        sum += a[i][j];
        return sum;
}
```

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**