Very Large Scale Integration II - VLSI II

Cache & Memory Systems

Prof. Dr. Berna Örs Yalçın

Res. Asst. Serdar Duran

ITU VLSI Laboratories

Istanbul Technical University

What is a Cache?

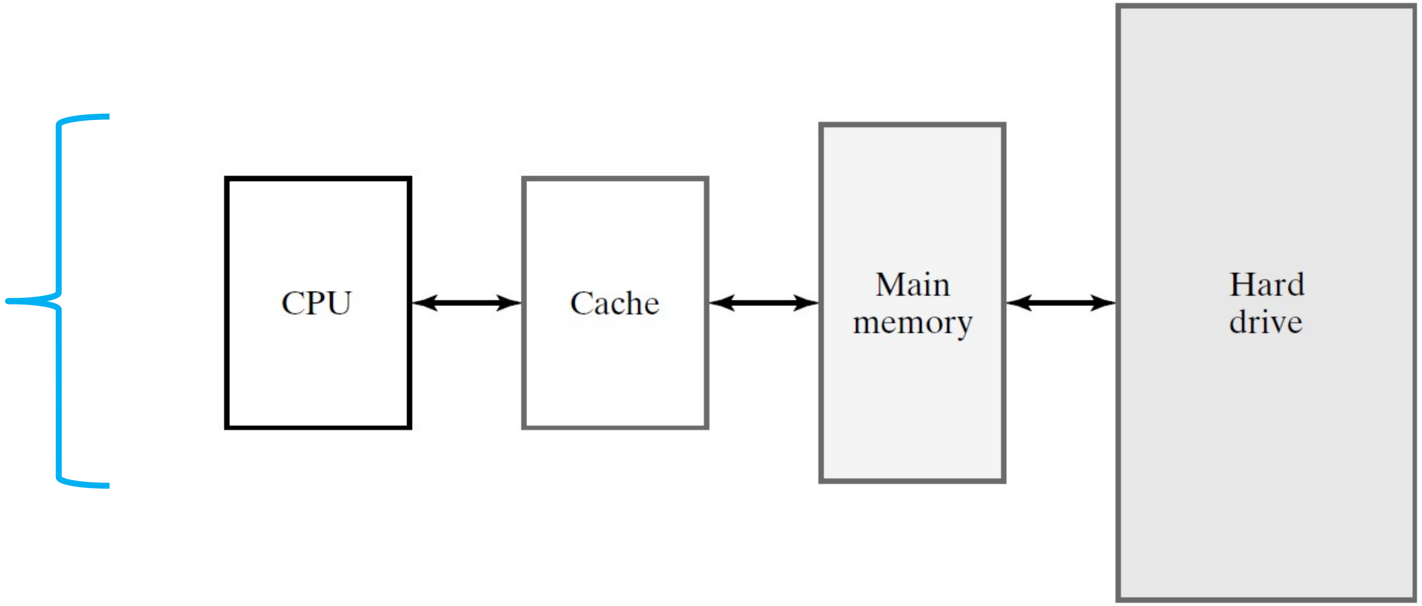
- All programs spend much of their time for accessing memory.
- Therefore, **memory system** is a **major factor** for the general computer performance.

- In order to improve computer performance, there are some mechanisms inluding **memory hierarchy** and **cache** design.

What is a Cache?

- In a computer system, the second lowest level of the memory hierarchy is a cache. First is the registers.
- The aim of the cache is that make the access memory faster while keep the memory size large!

Memory Hierarchy



Principle of Locality

- **The principle of locality** states that programs access a small portion of the address space repetitively over a short period of time!

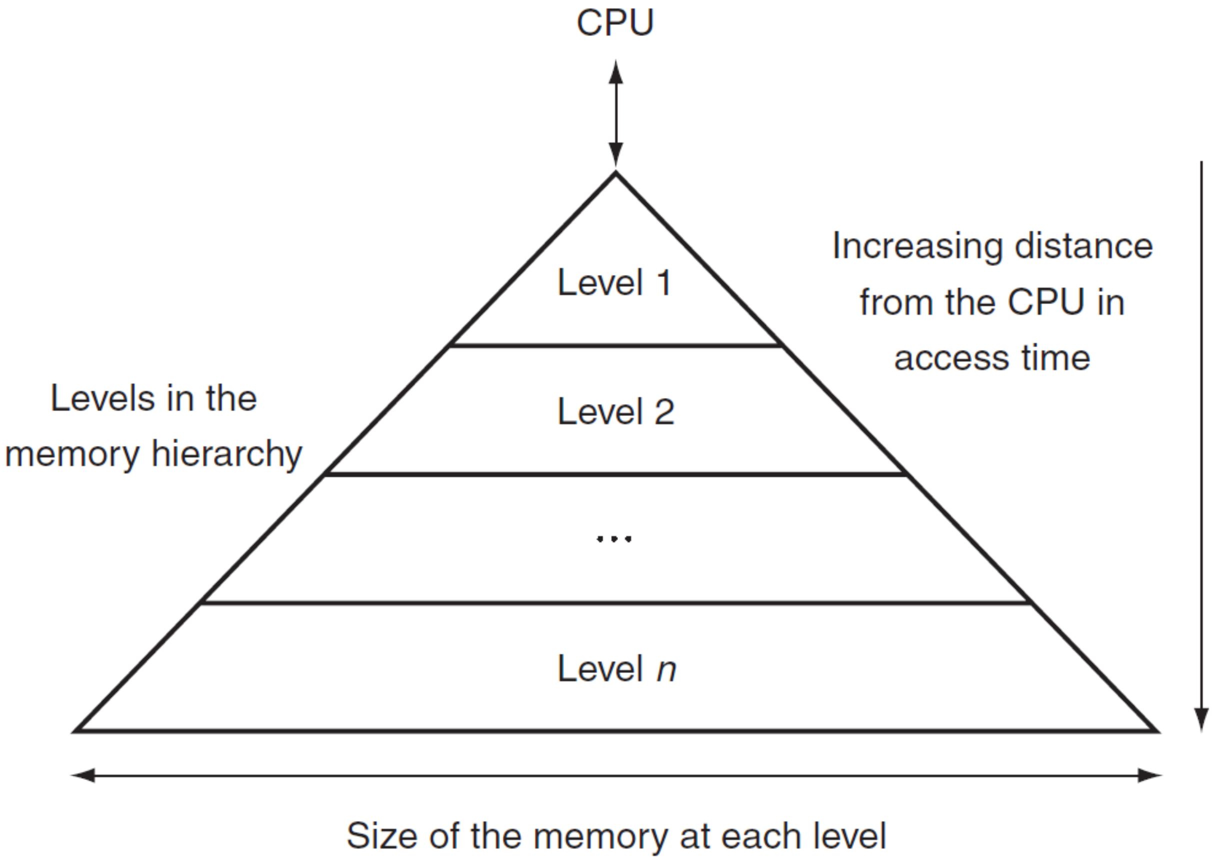
- Two different types of locality:
  - **Temporal locality:** the reuse of specific data within a small time duration.
  - **Spatial locality:** the use of data elements within close memory locations.

Principle of Locality

- **Temporal Locality:** Most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing high amounts of temporal locality.
- **Spatial Locality:** Since instructions are normally accessed sequentially (no branching), programs show high spatial locality. Besides, accessing to elements of an array or a record have high spatial locality.

Memory Hierarchy

- The advantage of the locality can be achieved by implementing a memory hierarchy.
- Because of the differences in cost and access time, building a memory as a hierarchy is advantageous.


CPU
Level 1
Level 2
...
Level *n*
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

	Speed	CPU	Size	Cost ($/bit)	Current Technology
CACHE	Fastest	Memory	Smallest	Highest	SRAM
MAIN MEMORY & RAM		Memory			DRAM
HARD DRIVE & DISK	Slowest	Memory	Biggest	Lowest	Magnetic Disk

Memory Hierarchy

- The idea is achieving the access time of the **fastest** memory component and yet having a memory size as large as the **largest** memory component.

- A large portion of the instruction and operand fetches are expected to be from the **cache** because it is fastest one.
- Some other portion of fetches not satisfied by the **cache** is performed by the **main memory**.
- The **hard drive** is accessed in the very infrequent cases in which a CPU instruction or operand fetch is not found in **main memory**.

Example

- **CPU :**
  - Memory address width 32 bits
  - Access time1-ns clock cycle.
- **Main Memory:**
  - 2^32 = 4 GB main memory
  - 2 gb for instruction memory, 2 gb for data memory.
  - Access time 10-ns clock cycles.

- One-tenth speed is too slow.
- We can use a more efficient way.

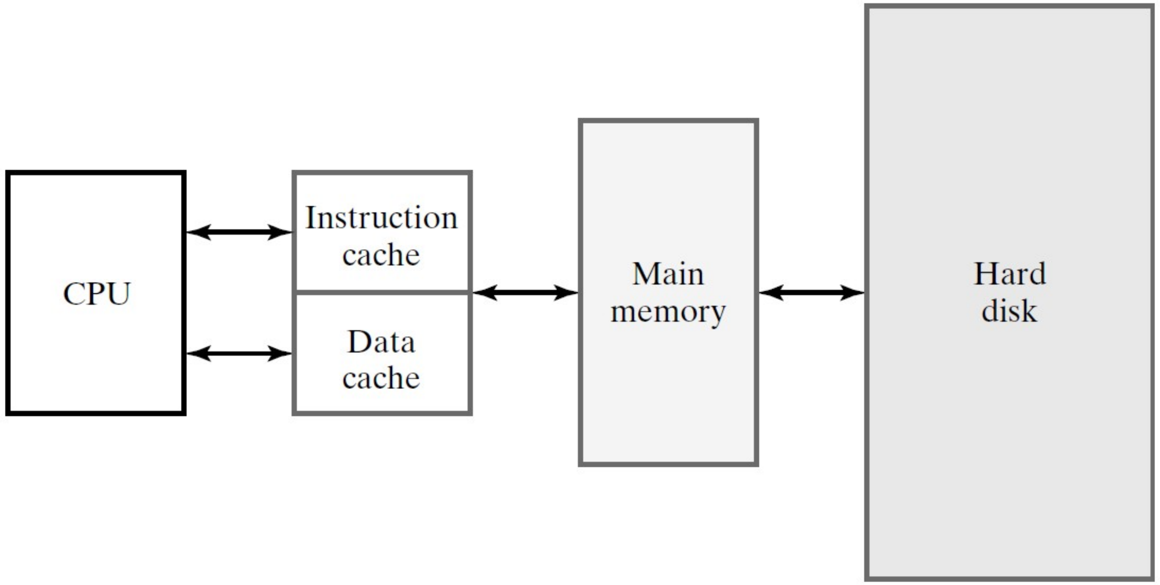
Example

- **Cache:**
  - 32 kb memory.
  - 16 kb for data-memory, 16 kb for instruction memory.
  - Fetch&Store 2-ns clock cycles.

- Suppose that %95 of the memory accesses performed from caches and remaining part %5 performed from main memory.
- Then the average access time is:
  - %95 x 2 + %5 x 10 = 2.4 ns

Example

- This means that, on 19 out of every 20 the CPU operates at full speed, while the CPU will have to wait for 10 clock cycles for 1 out of every 20.
- This wait can be accomplished by stalling the CPU pipeline.



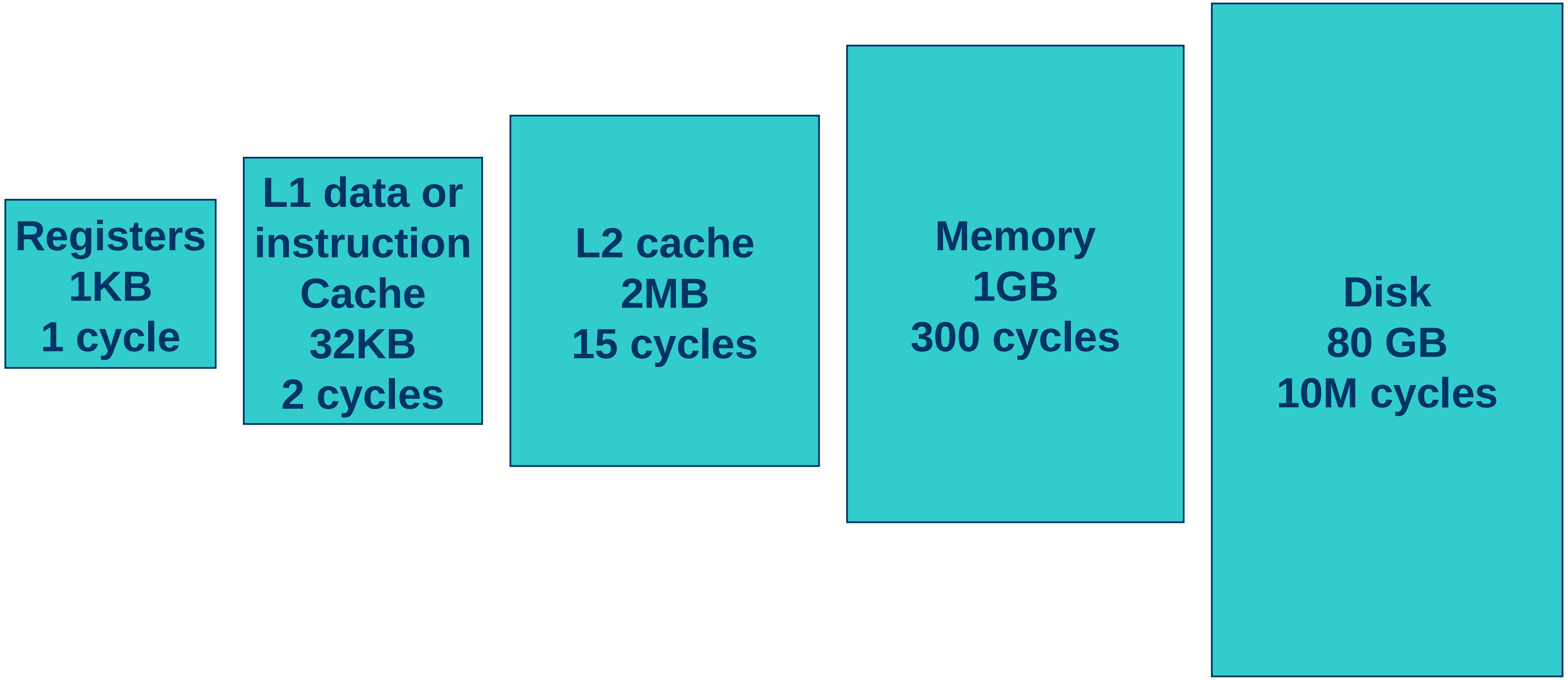
Example

- What about the size?
- Hard Disk:
  - 64 gb memory
  - Access time : 13ms = 1.3 x 10^7 ns

- Let's say %95 for cache %4.999995 for main memory.
- Then the average access time is:
  - 0.95 x 2 + 0.04999995 x 10 + 5 x 10^-8 x 1.3 x 10^7 = 3.05 ns

Memory Hierarchy

- As you go further, size and latency increase!



Registers
1KB
1 cycle

L1 data or instruction Cache
32KB
2 cycles

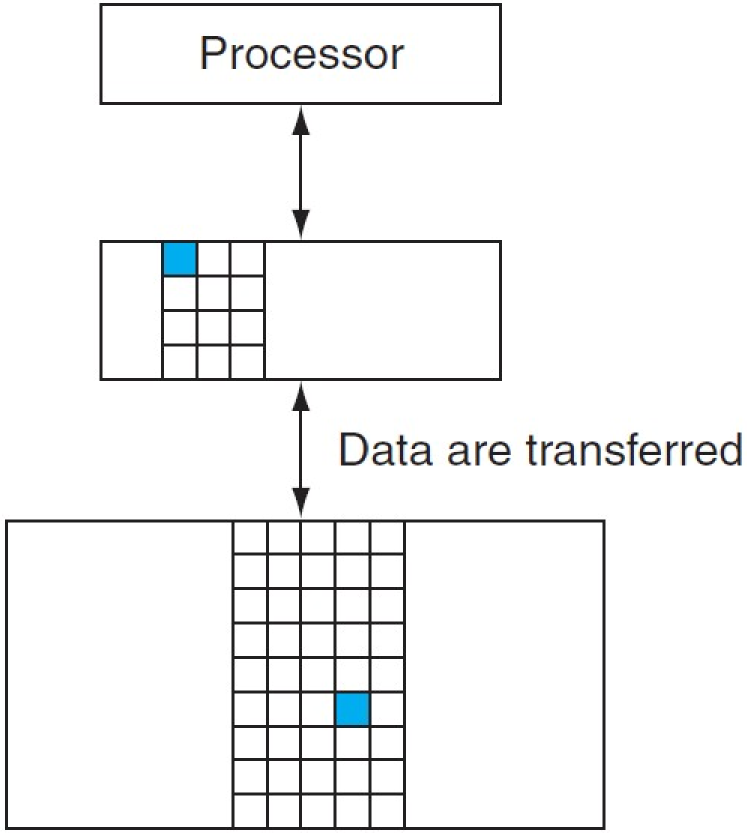
L2 cache
2MB
15 cycles

Memory
1GB
300 cycles

Disk
80 GB
10M cycles

Cache

- A memory hierarchy can consist of multiple levels, but data is copied between **only** two adjacent levels at a time. This is called **two-level hierarchy**.
- The minimum unit of data that can be either present or not in a level is called a **block**.



Cache

- **Hit:** If the data requested by the processor appears in the cache (or some other level).
- **Miss:** If the data is not found in the cache, it is called a miss.

- **Hit Rate:** It is the fraction of memory accesses found in the cache, it is often used as a measure of the performance.
- **Miss Rate:** (1 – hit rate) is the fraction of memory accesses not found in the cache.

Cache

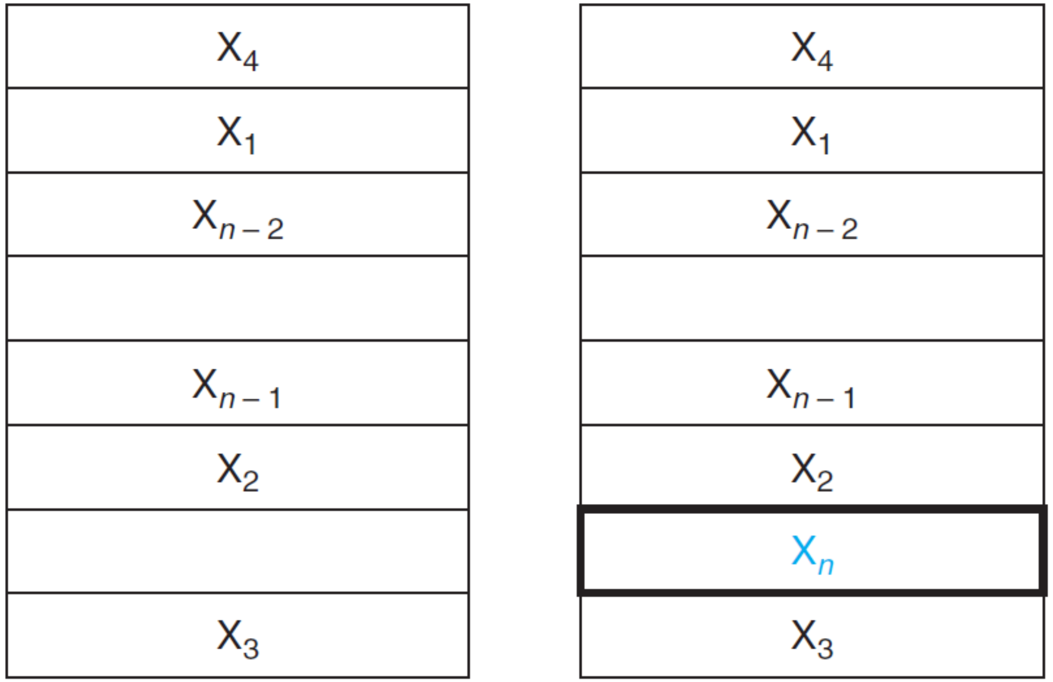
- **Hit Time:** The time needed to determine whether the access is a hit or a miss.
- **Miss penalty:** the time to replace a block in the cache with the corresponding block from the main memory.

Performance Metrics

- **Cache Capacity (C):** Memory size of the cache.
- **Block Size (b):** Number of data bytes can be transferred to the cache at one time.
- **Number of Blocks (B):** Number of blocks in the cache. (B = C / b)
- **Degree of Associativity(N):** Number of blocks in a set.
- **Number of Sets(S):** Number of words in a block. (S = B / N)

Cache

- Suppose that CPU requests an data item Xn.
- If Xn is initially in the cache. It is called **cache hit**.
- If not, this request results in a **miss**, and the word Xn is brought from memory into cache.



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

Cache

- How do we know if a data item is in the cache?
- If it is, how do we find it?

- There must be information in the cache to identify the **address** of the word in **main memory!**

Direct Mapped Cache

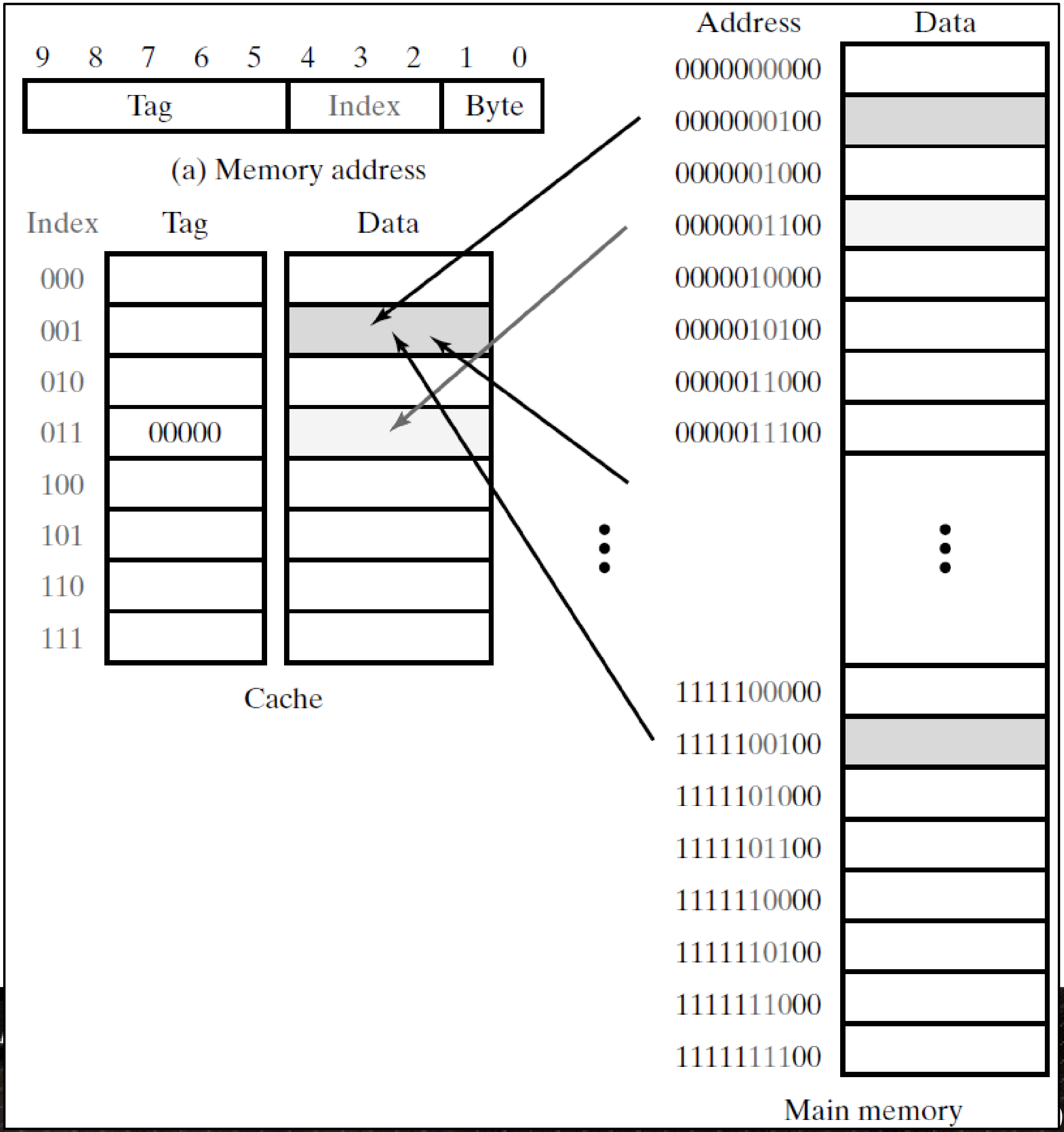
- The simplest way to assign a location in the cache for each word in main memory is **direct mapping**.
- In a **direct mapped cache**, each main memory location is mapped directly to the cache.

- This cache structure is called **direct mapped cache.**

- **Memory size:** 256-words length (1kB)
- **Memory address:** 10-bit

- **Cache size:** 8-words length cache
- **Block size:** One block is a 32-bit word in the cache
- **Cache address:** 3-bit

- 2-4 of the main memory address is the **cache address**. It is called **index**.
- Upper 5 bits of the main memory address is the **tag.**



9	8	7	6	5	4	3	2	1	0
Tag					Index			Byte	

(a) Memory address

Index	Tag	Data
000		
001		
010		
011	00000	
100		
101		
110		
111		

Cache

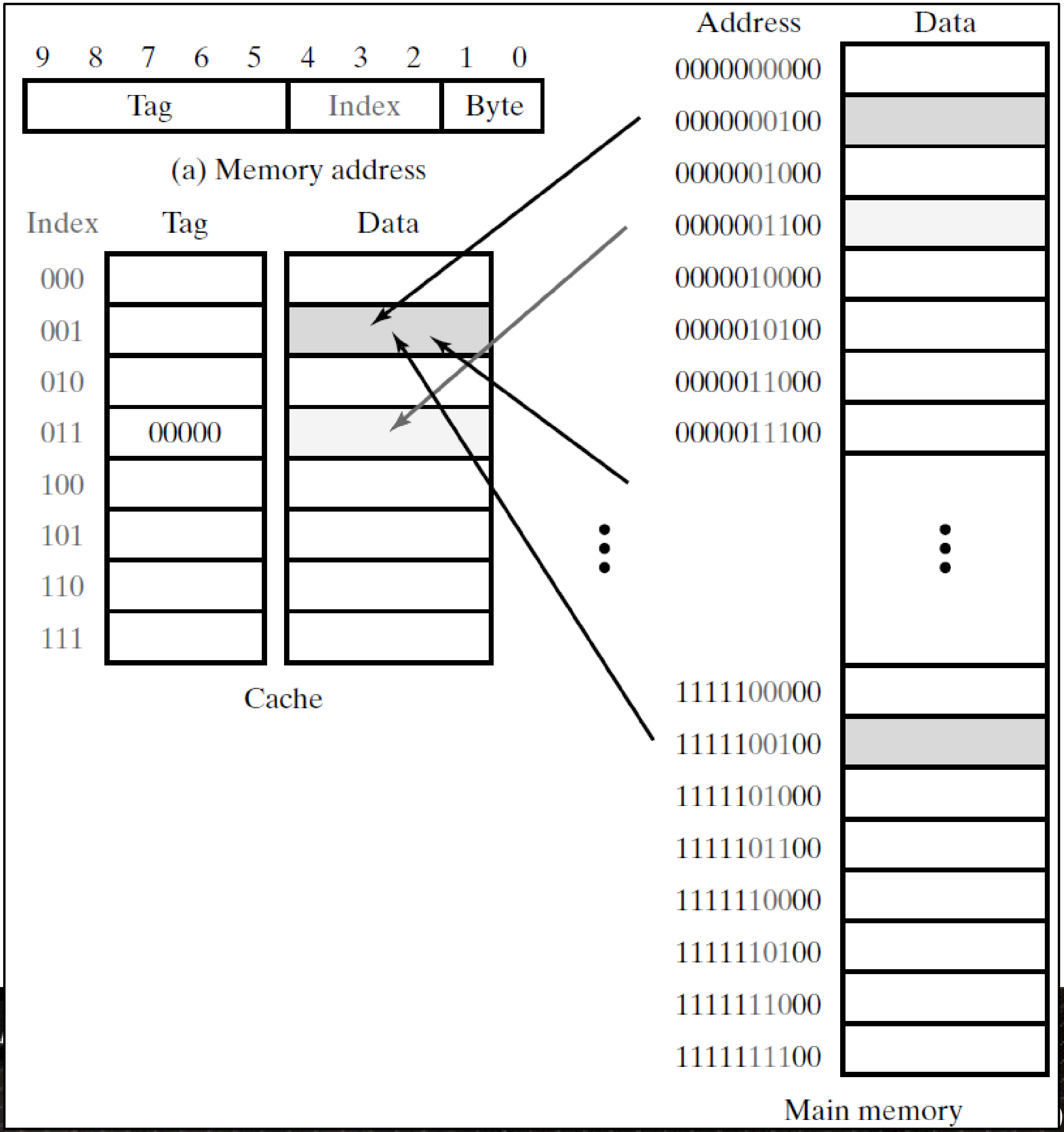
Address	Data
0000000000	
0000000100	
0000001000	
0000001100	
0000010000	
0000010100	
0000011000	
0000011100	
⋮	⋮
1111100000	
1111100100	
1111101000	
1111101100	
1111110000	
1111110100	
1111111000	
1111111100	

Main memory

- The first two bits of the memory address is excluded because memory is **word-addresable** (block size = 32 bits)

- **Cache Index** = 3 bits, since the cache has 8-bit words leng. 2^3 = 8
- **Cache index** is used to refer to one block.

- **Tag** = 5 bits, this is the remaining part of the memory address.
- **Tag** is necessary to identify the word address in main memory.



(a) Memory address

Index	Tag	Data
000		
001		
010		
011	00000	
100		
101		
110		
111		

Cache

Address	Data
0000000000	
0000000100	
0000001000	
0000001100	
0000010000	
0000010100	
0000011000	
0000011100	
⋮	⋮
1111100000	
1111100100	
1111101000	
1111101100	
1111110000	
1111110100	
1111111000	
1111111100	

Main memory

Direct Mapped Cache

- Suppose that the CPU fetch an instruction from the location 000001100 in main memory.
  - This instruction may fetch from either the cache or main memory!
  - The cache separates the **tag** 00000 from the **cache address** 011.
  - If the **tag** is 00000, then the tags match, and the stored word fetched from cache.

- If the **tag** fetched from cache memory is not 00000, then there is a **tag mismatch**.
  - The cache control notifies main memory that it must provide the memory word.
  - In expectation of future accesses to the same memory address the **tag** 00000 and the word from memory is written in cache location 011.

Fully Associative Cache

- Direct mapping for a cache is not practical.
- Suppose that:
  - Instructions and data are in the same cache
  - There is a loop instruction in location 0000001100.
  - Besides, data from location 1111101100 is frequently used.

- **Instruction :** **tag =** 11111, **index =** 011
- **Data :** **tag =** 00000, **index =** 011

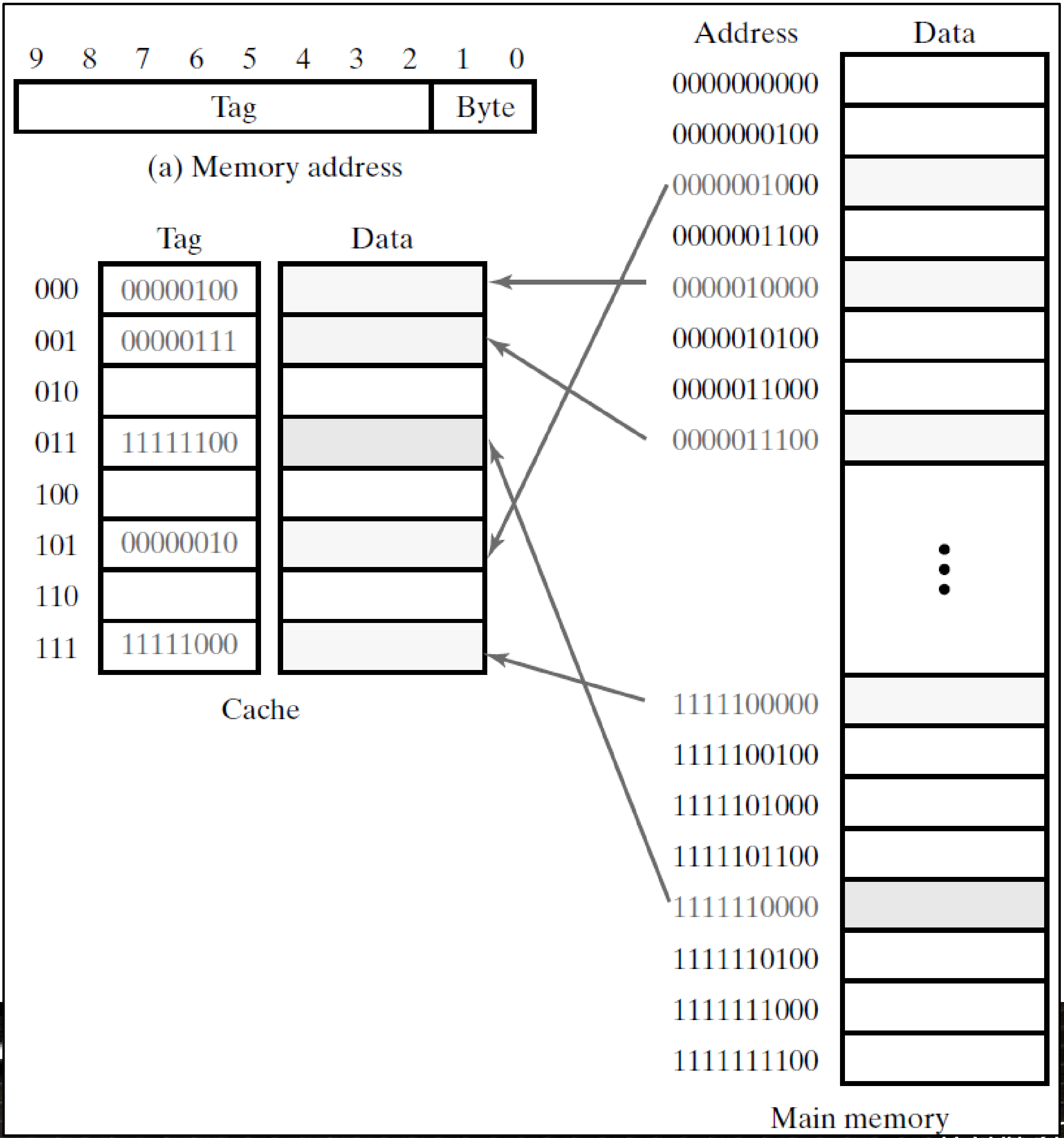
Fully Associative Cache

- In that case:
  - When the **instruction** in 0000001100 is fetched, location 011 in the probably contain the **data** from 1111101100 and tag 11111.
  - Then a cache miss occurs and tag 11111 would be replaced with tag 00000 in the cache.
  - But the next time the **data** is needed, another cache miss occurs, and the location in the cache would be replaced by the **instruction**.
  - Therefore, throughout the execution of the loop, both **instruction** fetch and **data** fetch cause many cache misses, significantly slowing CPU processing.

- In the **direct mapped cache**, only one of the 2^5 tag addresses can be mapped to the cache:

  00000, 00001, 00010....11111

- In contrast, we can let locations in main memory map into an **arbitrary** location in the cache.
- It means any location in memory can be mapped to any one of the eight addresses in the cache.



(a) Memory address

	Tag	Data
000	00000100	
001	00000111	
010		
011	11111100	
100		
101	00000010	
110		
111	11111000	

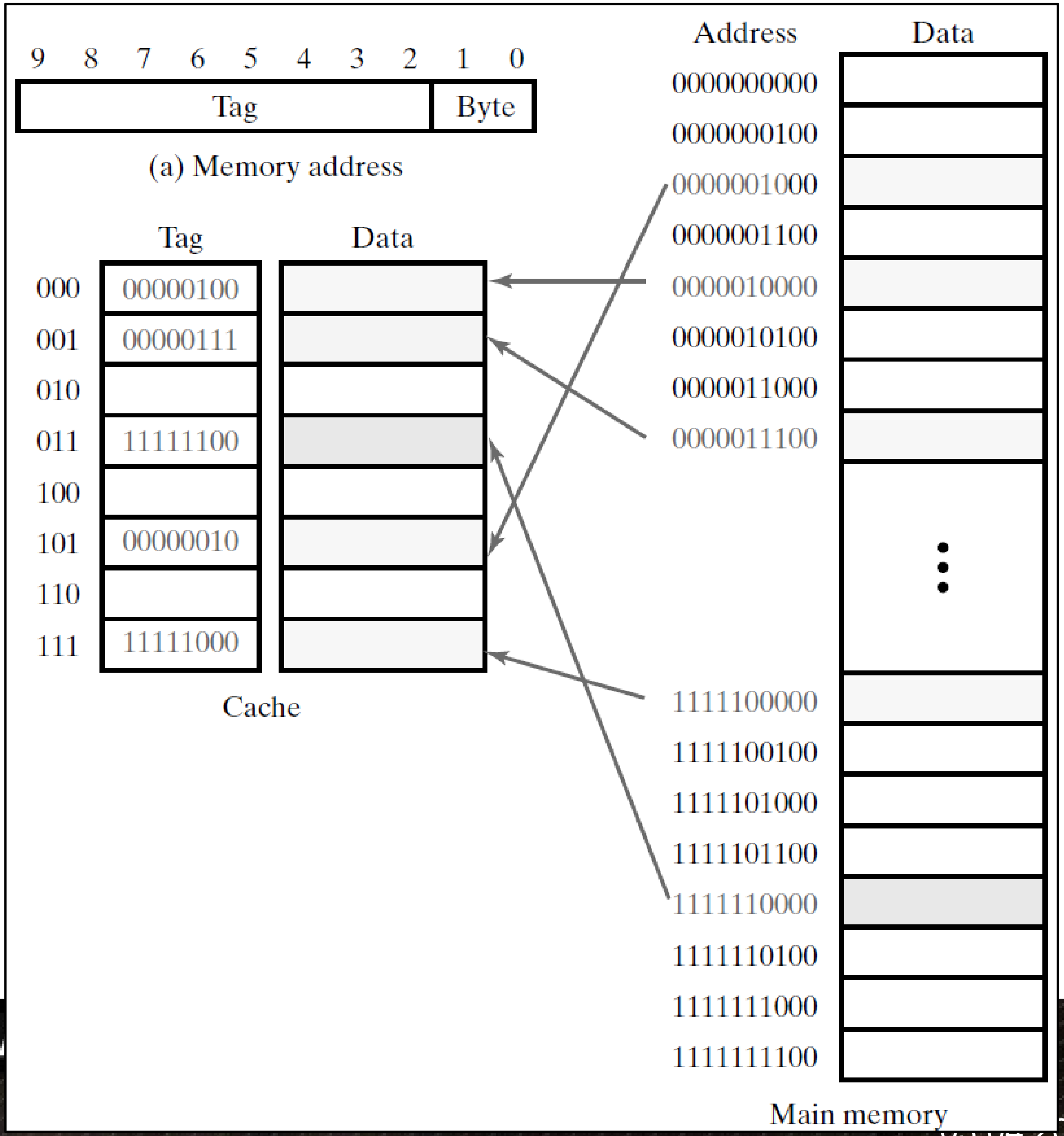
Cache

Address	Data
0000000000	
0000000100	
0000001000	
0000001100	
0000010000	
0000010100	
0000011000	
0000011100	
⋮	⋮
1111100000	
1111100100	
1111101000	
1111101100	
1111110000	
1111110100	
1111111000	
1111111100	

Main memory

- **Tag** is now the full main memory word address.
- This is called **full associative mapping**.

- In a **fetch** process, **tags** are compared and if any match occurs then there is a cache hit, otherwise a cache miss.



(a) Memory address

	Tag	Data
000	00000100	
001	00000111	
010		
011	11111100	
100		
101	00000010	
110		
111	11111000	

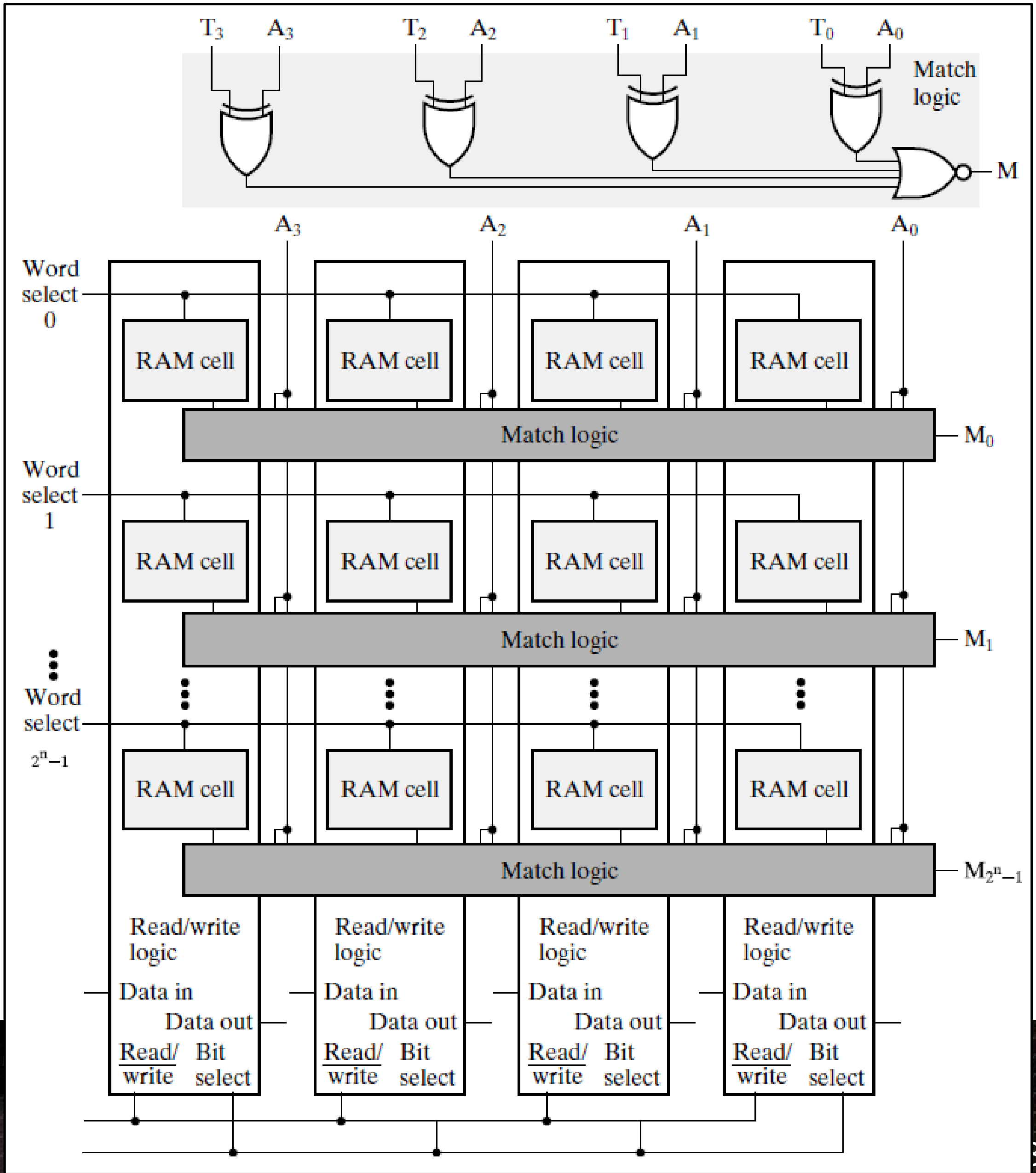
Cache

Address	Data
0000000000	
0000000100	
0000001000	
0000001100	
0000010000	
0000010100	
0000011000	
0000011100	
⋮	⋮
1111100000	
1111100100	
1111101000	
1111101100	
1111110000	
1111110100	
1111111000	
1111111100	

Main memory

- In **fully associative cache**, reading and comparing of tags can take too much time.
- Suppose that a fetch operation requires 2 ns, in the worst case 8 fetch is necessary and it takes 16 ns.

- Instead of **successive** reading of **tags**, associative memory structure (slicing model) can be used.
- Each tag storage row has match logic.



Fully Associative Cache

- The match logic does an equality comparison between the **tag T** and the applied address **A** from the CPU.
- Since all tags are unique, only two situations can arise in the associative memory:
  - there will be a match, with a 1 on the output of the match logic for one tag and a 0 on the remaining match logic outputs,
  - or there will be no match, and all of the match logic outputs will be 0.

Fully Associative Cache

- In a **cache miss** situation, the fully associative cache must bring the word and its address tag from the main memory for future accesses.
- Therefore, a replacement approach must be selected:
  - Random
  - FIFO : replace with the oldest entry
  - LRU : replace with the least used entry

- LRU approach is better than others.

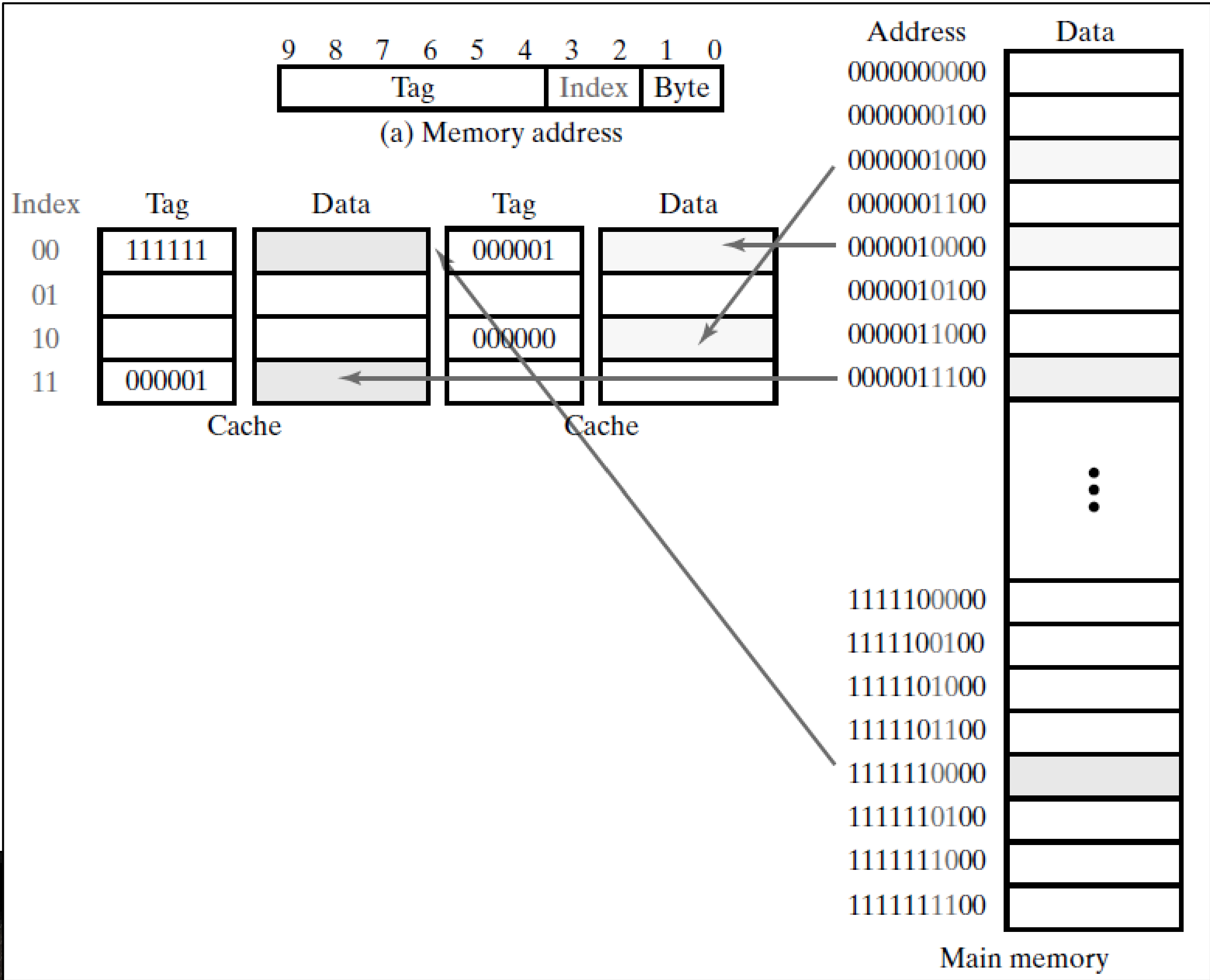
Set-Associative Mapping

- **Set-Associative Mapping** has better performance and eliminates the cost of most of the matching logic blocks.

- **Tags** and **words** from the cache are addressed by the lower-order address bits similar to **direct mapping** (index).
- Instead of having one location, there is a **set** of locations. If the set size equals **2**, then two tags and the two data words are read simultaneously.
- Tags are compared using just **2** math logic blocks.
- This is called **2-Way Associative Cache**.

- Eight cache locations are arranged in four rows of two locations each.
- A cache entry can be in either of the two columns.

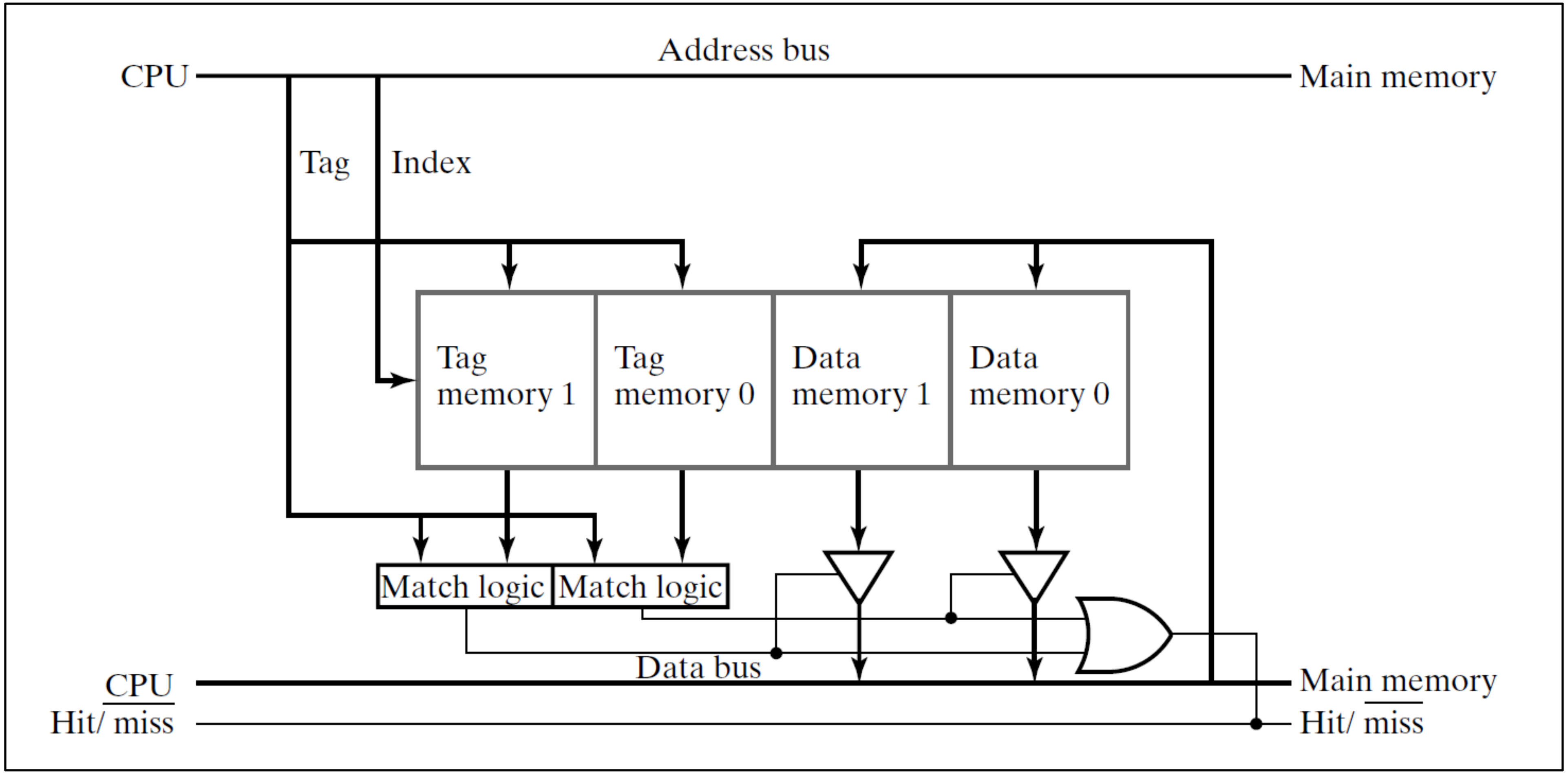
- **Index:** 2-bit
- **Tag:** 6 bits



Index	Tag	Data	Tag	Data
00	111111		000001	
01				
10			000000	
11	000001			

2-Way Associative Cache

- 00 index for both cells are occupied by addresses 0000010000 and 1111110000.
- In order to accommodate another 00 index address like 1111100000, the **set size** would need to be at least three.
- Therefore, full associative cache is more **flexible** but 2-way associative cache has a **less complicated** hardware.


Address bus
CPU
Main memory
Tag
Index
Tag memory 1
Tag memory 0
Data memory 1
Data memory 0
Match logic
Match logic
Data bus
CPU
Hit/ miss
Main memory
Hit/ miss

2-Way Associative Cache

- The index is used to address each row of the cache memory.
- The two tags read from the tag memories are compared with the tag to be fetch.
- If a match occurs, then the three-state buffer on the corresponding data memory output is activated, placing the data onto the data bus to the CPU.
- In addition, the match signal causes the output of the Hit/Miss.

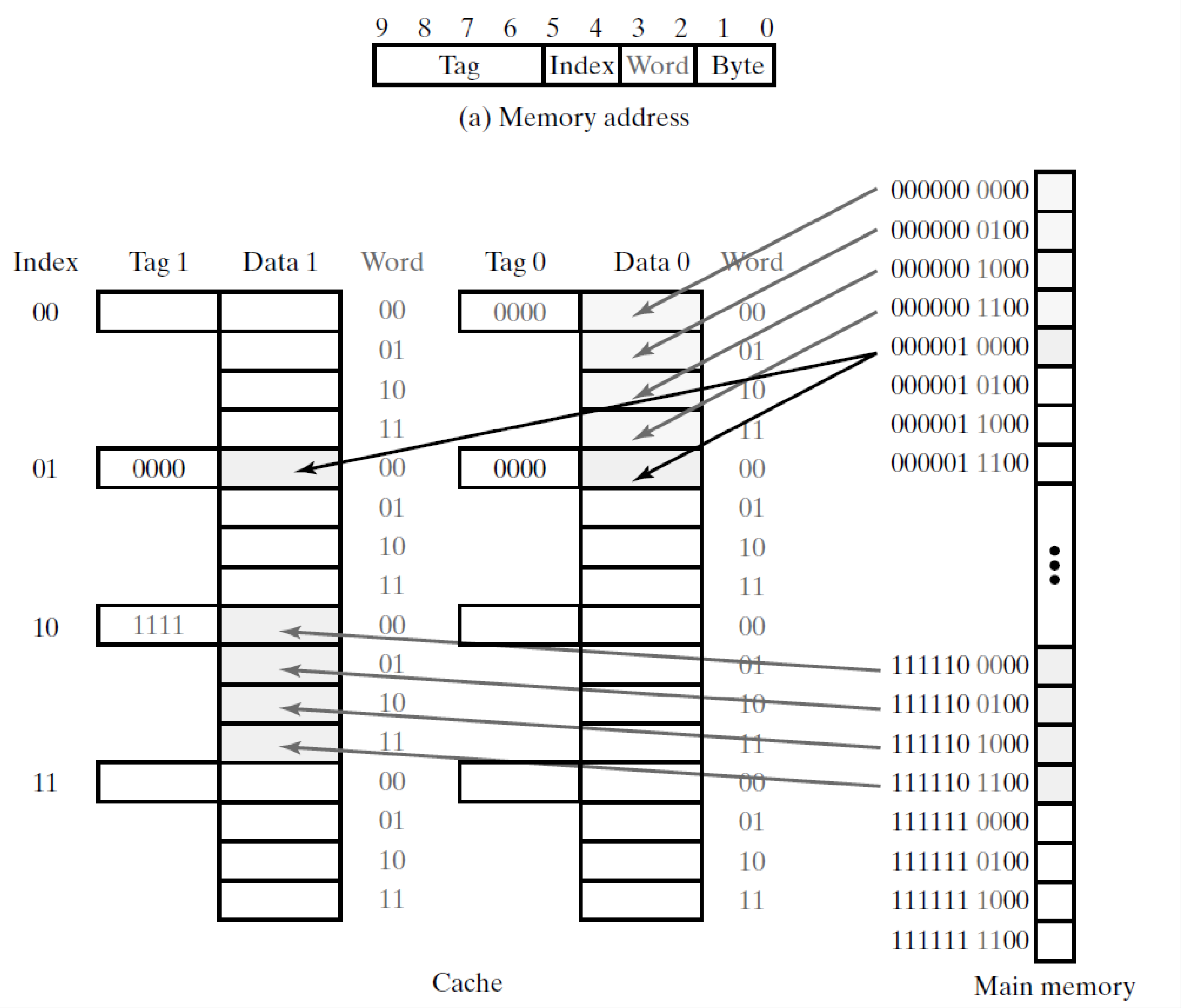
Line Size

- In real caches additional words close to the addressed one are also included in the cache.
- Therefore, when a cache miss occurs, a block of words called a **line** is fetched.
- By this way, spatial locality can be exploited.
- For example,
  - If four words are included in a line, then the addresses of the words in the line differ only in bits 2 and 3.
  - The use of a block of words changes the address fields.
  - The new field structure is **Word Field** including bits 2 and 3.

- **2-Way Associative Cache with 4-Word Lines**

- **Word** refers to one of the word in the set.
- **Index** refers to set.

- **Word:** 2 bits (4 words)
- **Index:** 2 bits (4 sets)
- **Tag:** 4 bits



2-Way Associative Cache with 4-Word Lines

- **Index** is used to read two **tags**.
- **Index** and the **Word address** are applied to read out two words from the two **tags**.

- 2 matching logic is used for comparison of the tags.
- If a match occurs, then the data word already read is placed on the memory bus to the CPU.
- Otherwise, a cache miss is signaled.

Path Size

- To load the entire line of words, the width of the memory bus is made more than one word wide.
- The path can be 4 × 32 = 128 bits wide. This allows the entire line to be placed in the cache in a single read cycle.
- If the path is narrower, then a sequence read cycles are required.

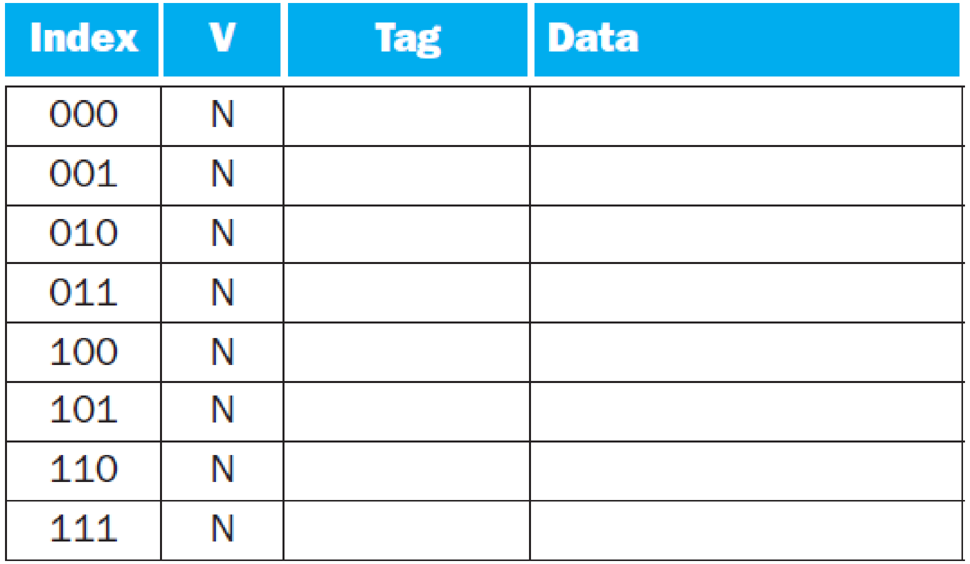
- A wide path can cause higher cost, but a narrower path can cause slow transfer of the line.

Cache Loading

- Before any words and tags have been loaded into the cache, all locations contain invalid information.
- As lines are fetched from main memory into the cache, cache entries become valid.

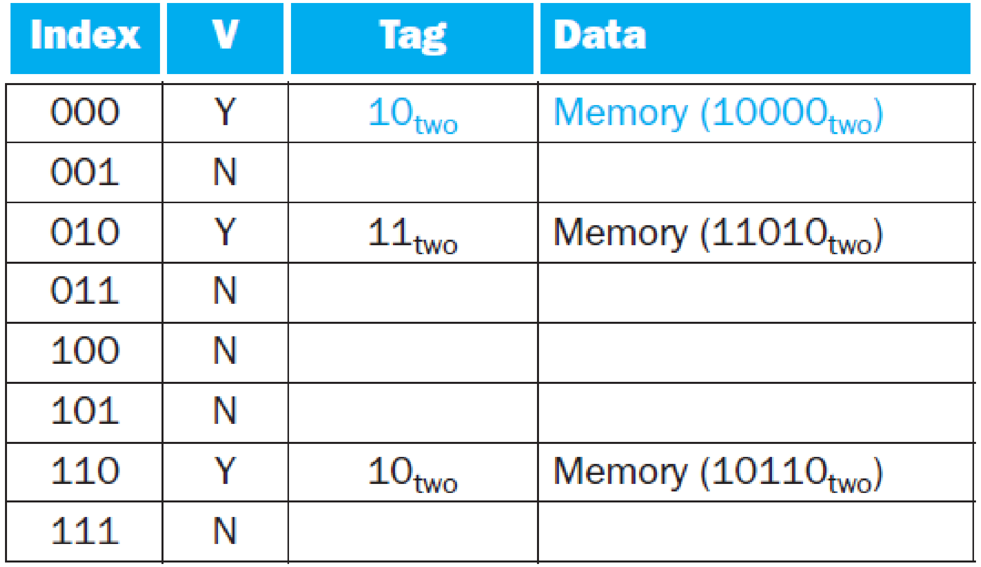
- To distinguish valid entries from invalid entries, a **valid bit** is used.
- A **valid bit** indicates whether an entry contains an valid address.

Cache Loading



Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on



Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Write Methods

- Write operations are:
  1. Write the result into main memory.
  2. Write the result into the cache.
  3. Write the result into both main memory and the cache.

- We have two categories:
  - write-through
  - write-back.

Write Methods

- In **write-through**, the result is always written to main memory.
  - This uses the main memory write time and can slow down processing.
  - The slowdown can be partially avoided by using **write buffering**.
  - In **write buffering**, address and word are stored in special registers called write buffers.
  - By this method, CPU can continue processing during the write to main memory.

Write Methods

- In **write-back method**, the result is written to the cache in case of a cache hit.
  - The goal of a write-back cache is to be able to write at the writing speed of the cache whenever there is a cache hit.
  - When a cache miss appears, the result is released from the cache and written into the main memory.
  - This avoids having all writes performed at the slower writing speed of main memory.

References

- D. A. Patterson, *Computer organization and Design*. San Francisco: Elsevier Science & Technology.
- M. M. Mano, and C. R. Kime,, *Logic and computer design fundamentals*. Boston: Pearson.