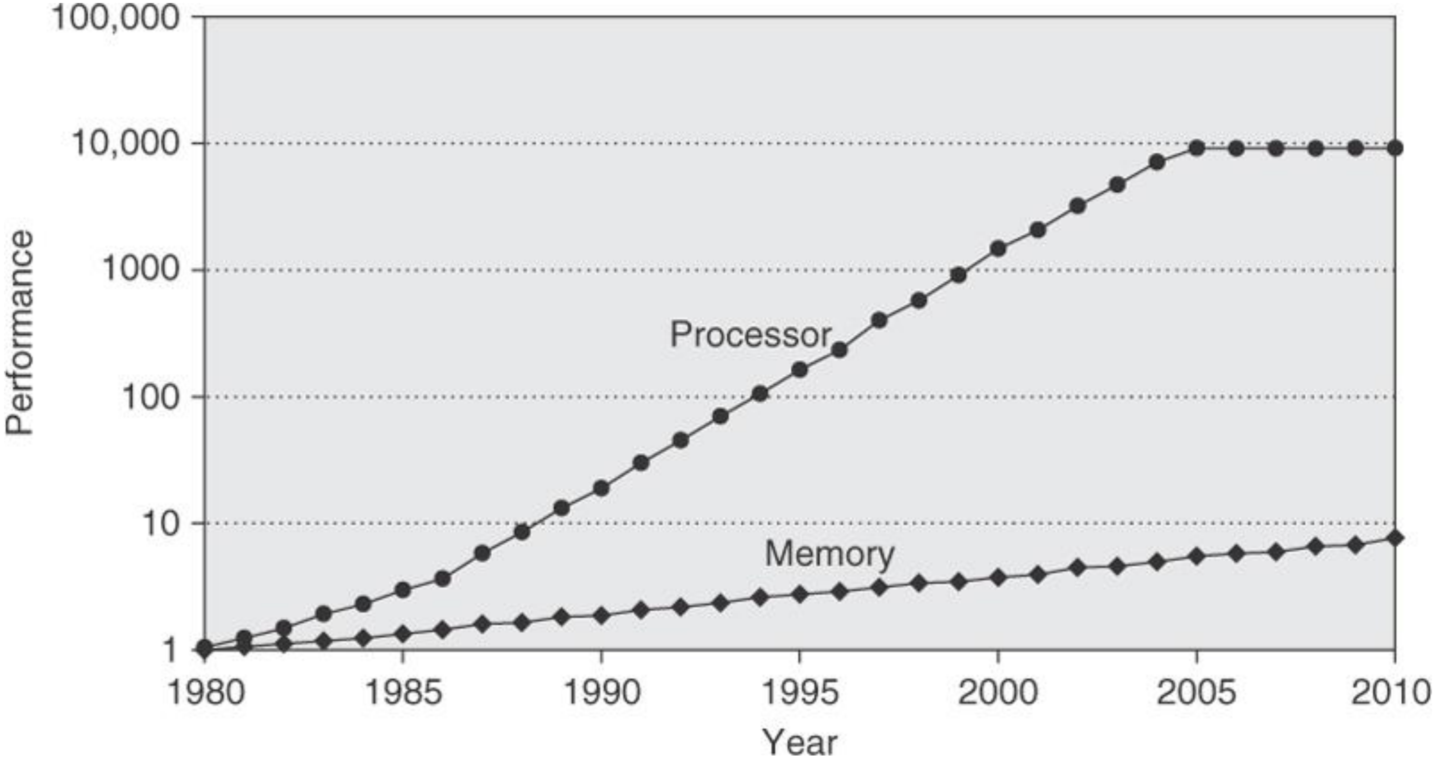
Cache memories

E. Sanchez

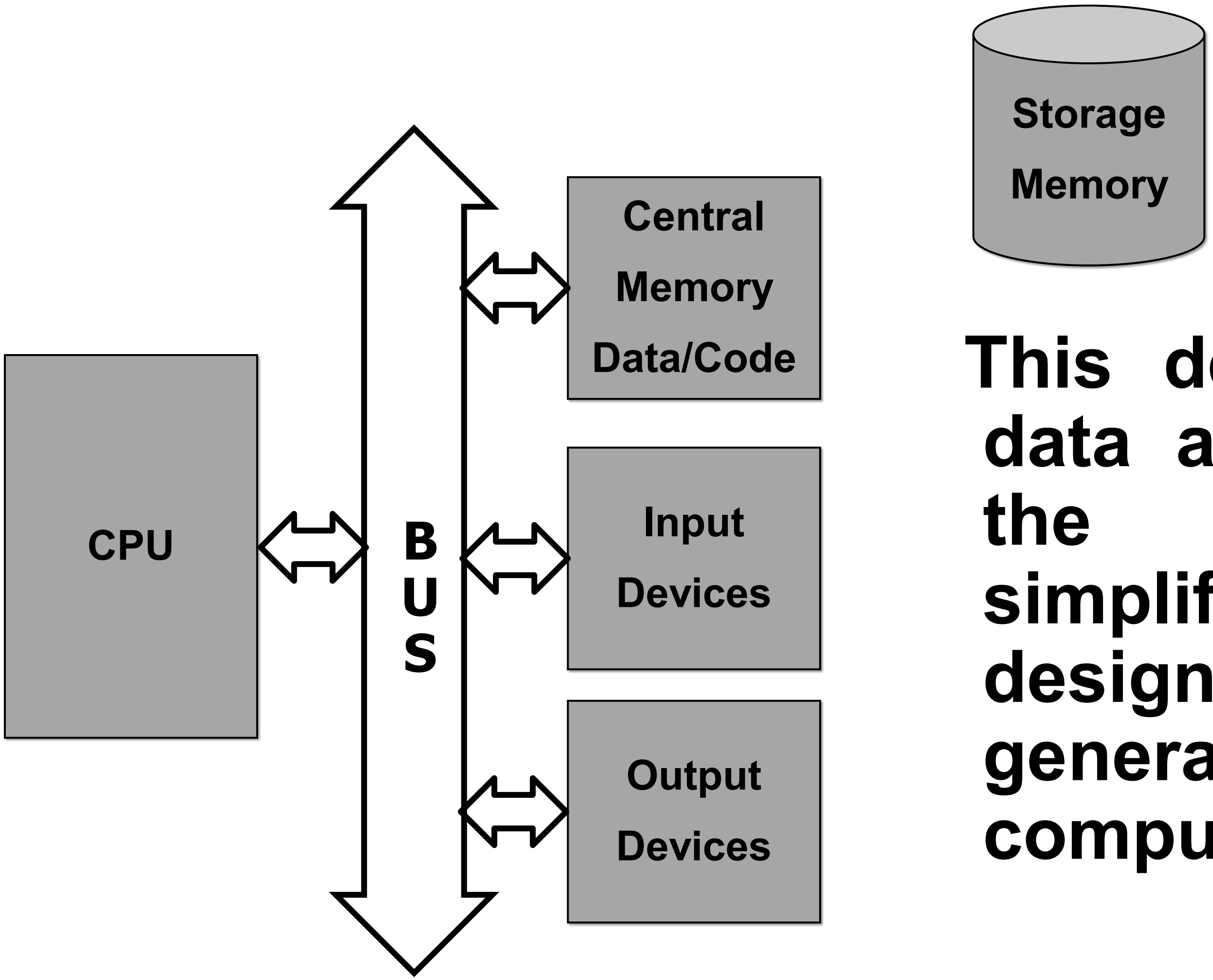
Politecnico di Torino
Dipartimento di Automatica e Informatica

Introduction



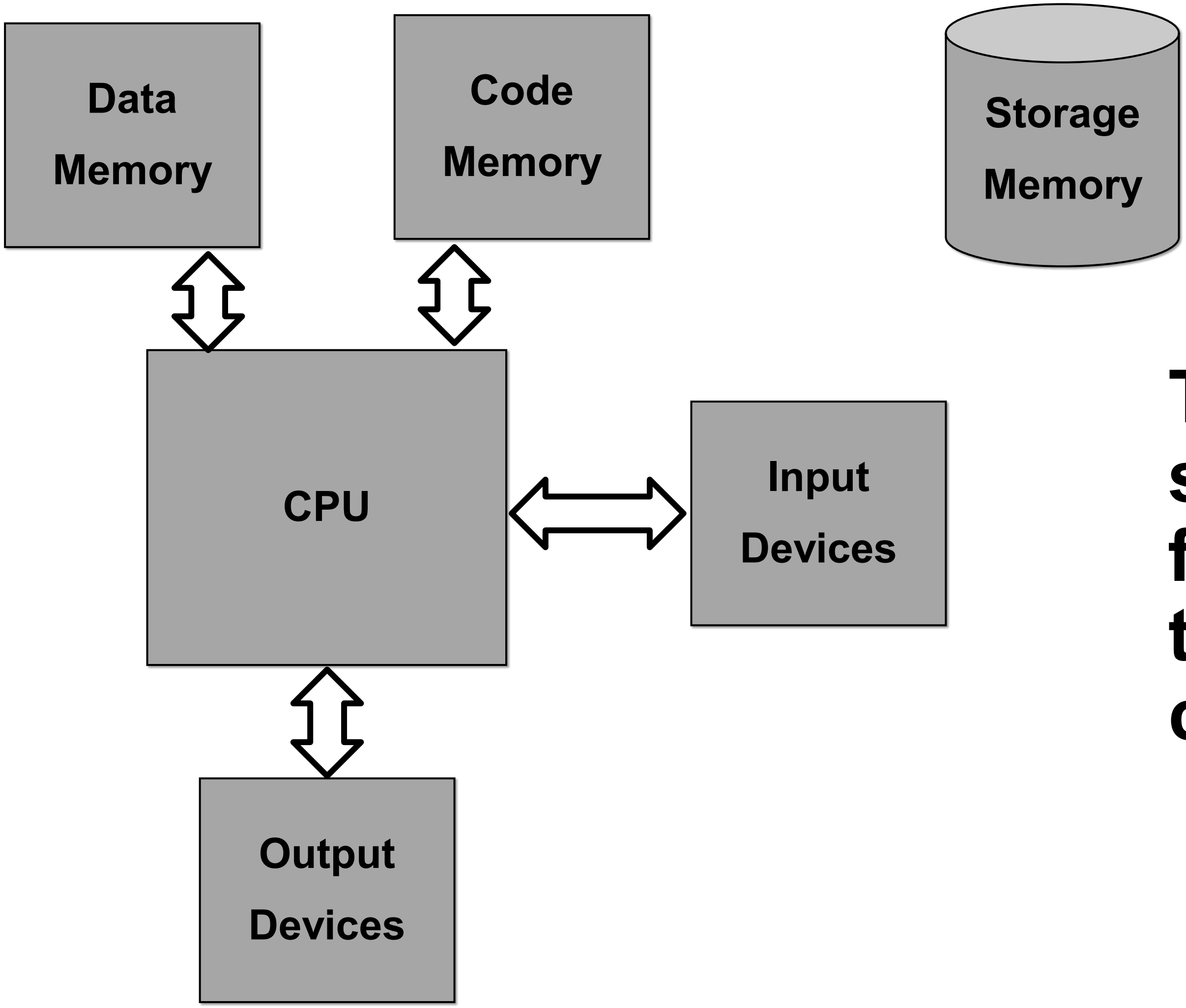
Cache memories are small but high-speed memories that are interposed between the processor and the main memory.

Von Neumann architecture



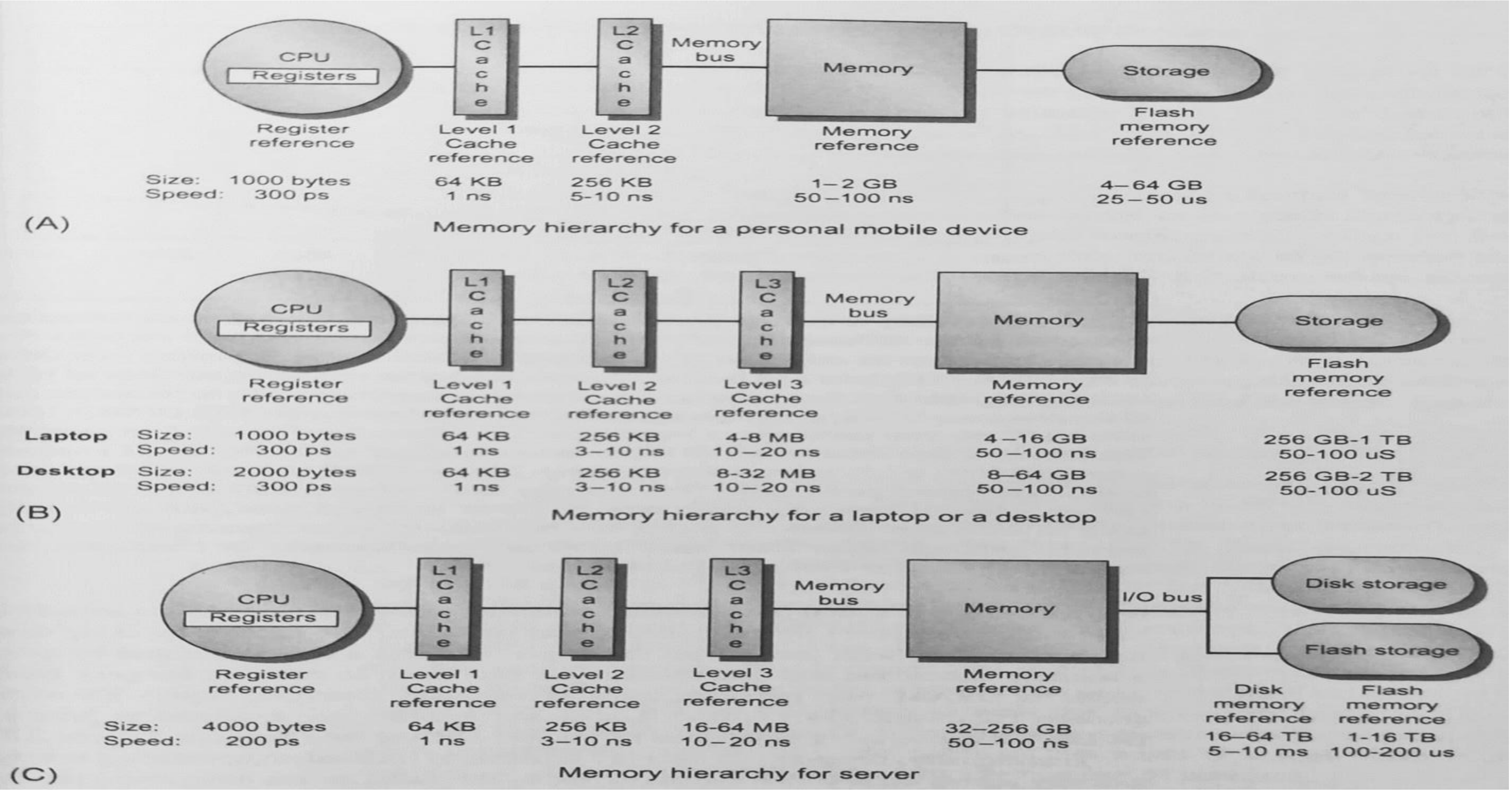
This design stores both data and instructions in the same memory, simplifying hardware design and enabling general-purpose computing.

Harvard architecture



There are two separated memories for Data and Code, these are usually cache memories.

Memory levels in a typical system



(A) Memory hierarchy for a personal mobile device

(B) Memory hierarchy for a laptop or a desktop

(C) Memory hierarchy for server

Locality of reference

The presence of a cache can improve the performance of a system due to the locality of references observed in most programs.

Locality takes two forms:

- *Temporal locality*: if at time `t` the program accesses to a given memory cell, it is highly probable that the program accesses again the same cell by the time $t_0 + \Delta t$
- *Spatial locality*: if at time t_0 the program accesses a memory cell with address `X`, it is highly likely that by the time $t_0 + \Delta t$ the program will also access the memory cell with address $X \pm e$.

Working principle

If the entire block is loaded in the cache at time t_0 (first access to a memory block), it is likely that for a certain time Δt the program will find in the cache all of the words it needs.

Performance

Let define the following elements:

- *h*: cache hit ratio
- *C*: cache access time
- *M*: memory access time when the data is not in the cache.

The average access time to memory is

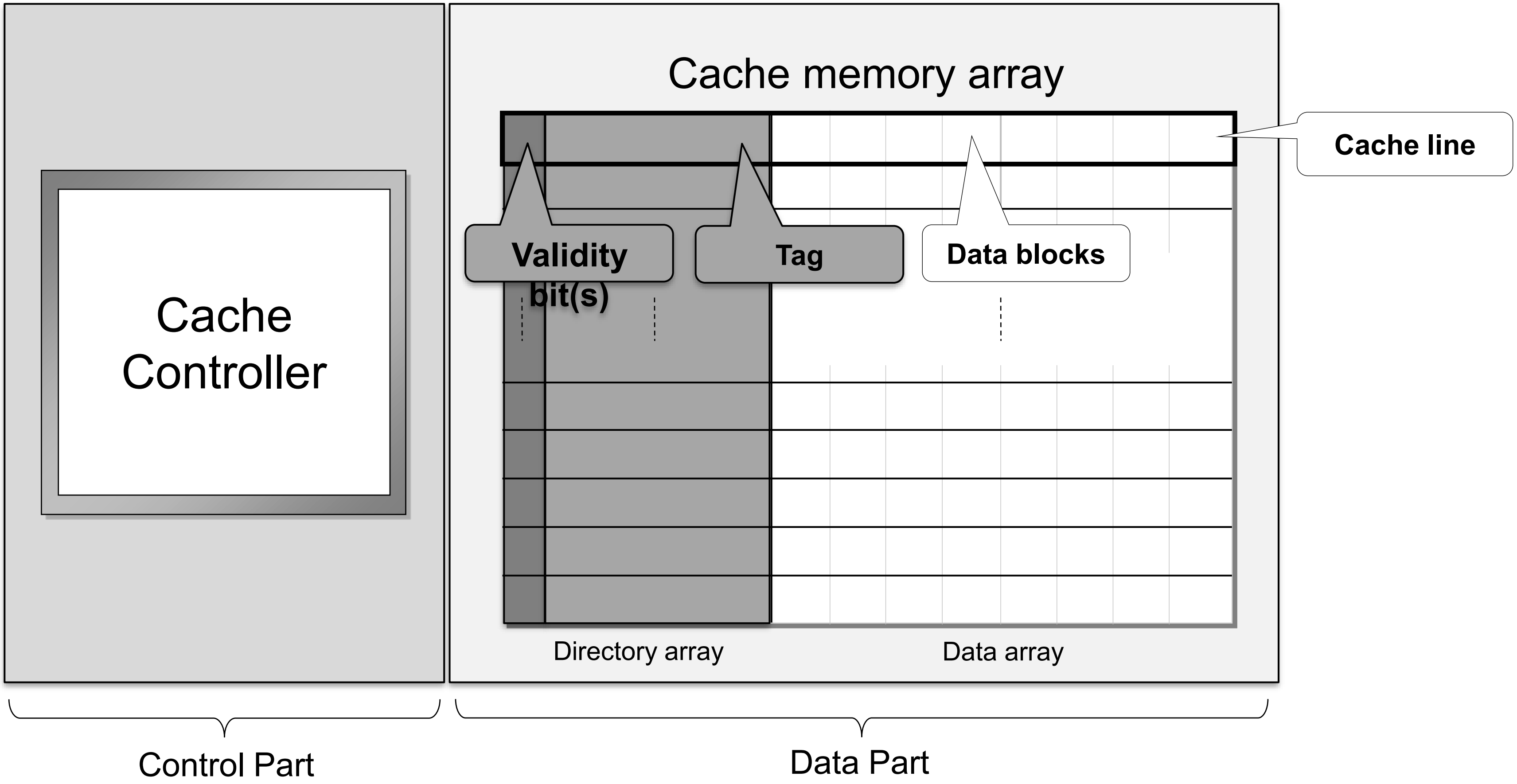
$$t_{ave} = h*C + (1-h)*M$$

Normal values for *h* are in the order of 0.8 - 0.9.

For example:

$$T_{avg}=0{,}8 \text{ x } 2\text{x}10^{-9} + 0{,}2 \text{ x } 50\text{x}10^{-6} = 10{,}001\ \mu\text{s}$$

Cache organization



Cache organization (II)

A cache is organized in lines.

A line contains a memory block that includes some memory words.

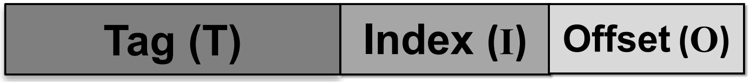
Each line is associated with a tag field, which indicates the memory block present in the line at that time.

The cache also contains the logic for

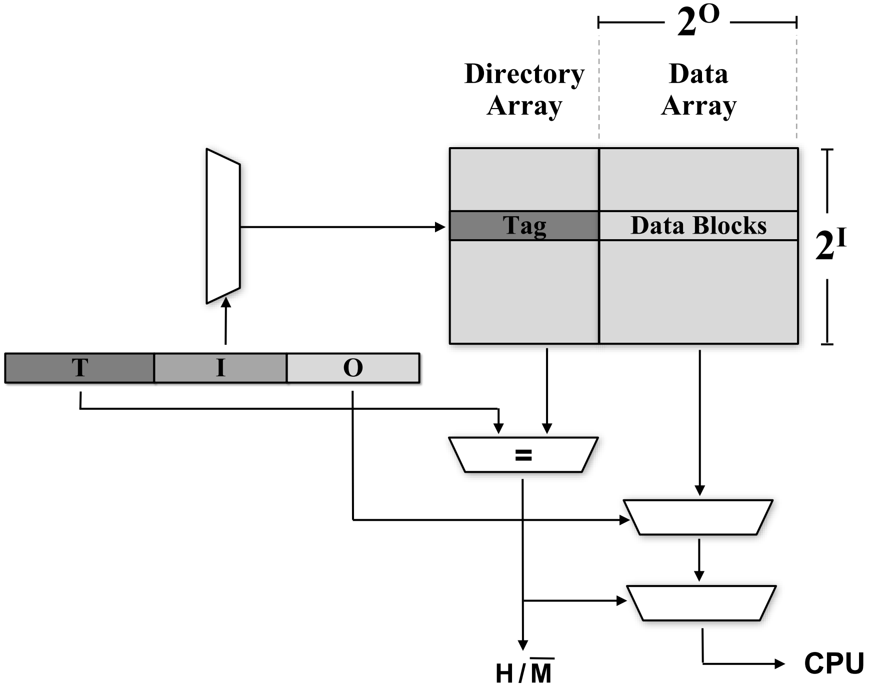
- intercepting the addresses produced by the processor
- checking inside the cache the possible presence of the block that the processor wants to access
- possibly loading the block.

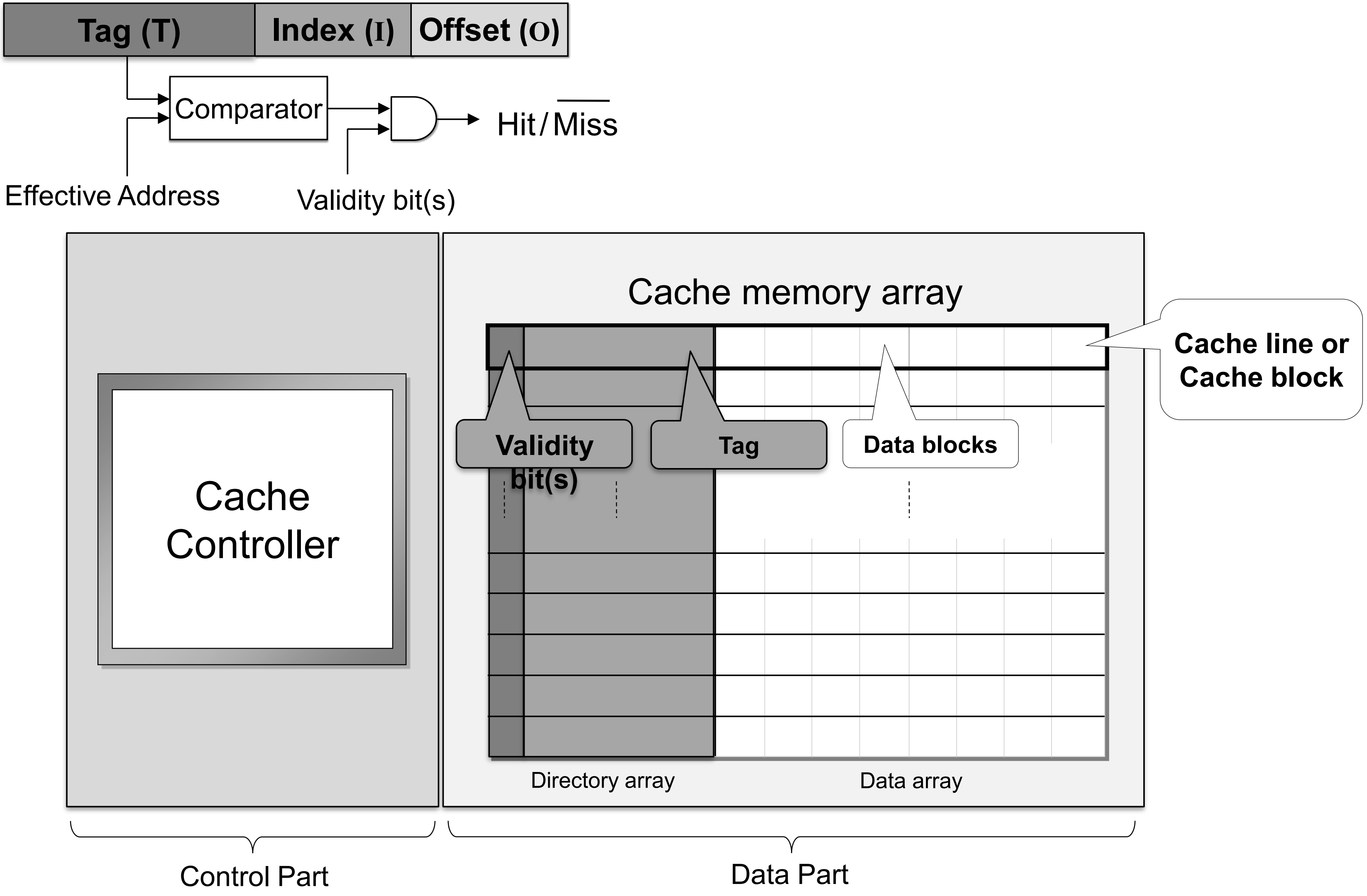
Finding a data block in the cache

CPU Address



Cache hit	if a required block is placed in the cache
Cache miss	if a required block is not placed in the cache




Tag (T)
Index (I)
Offset (O)
Comparator
Hit / Miss
Effective Address
Validity bit(s)
Cache memory array
Cache line or Cache block
Validity bit(s)
Tag
Data blocks
Cache Controller
Directory array
Data array
Control Part
Data Part

Cache behavior

The cache is located between the processor and the main memory.

Each time the processor performs an access to memory the cache intercepts the address and checks if the block to which the word belongs is in the cache, checking the value of the tags

- if yes: it extracts the word from the block and provides it to the CPU without any access to the main memory (*hit*)
- if no: it loads in the cache the entire block that the word is part of (*miss*).

Performance

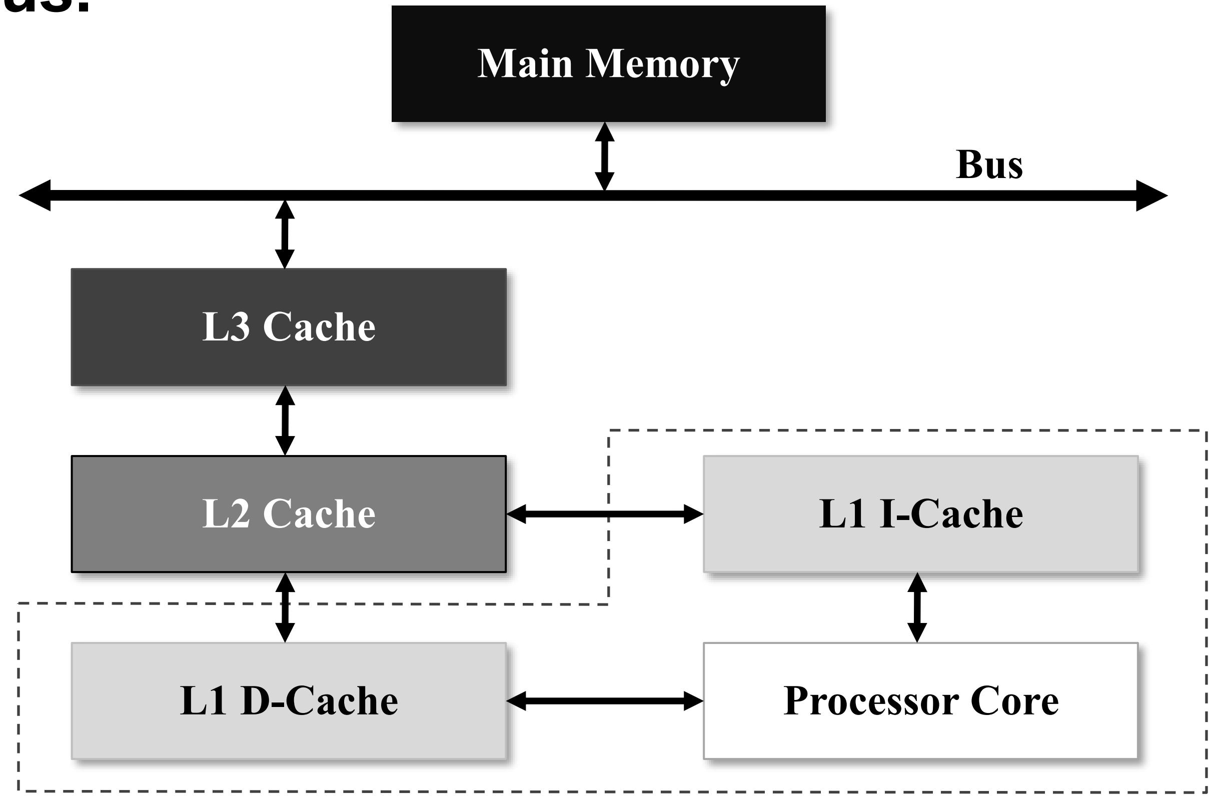
In the case of a hit, the cache reduces the memory access time by a factor that is dependent on the ratio between cache and primary memory access times.

In case of a miss, the cache may respond in two possible ways:

- It accesses the memory and loads the entire missing block; then it provides the requested word. The access time is therefore higher than in a cache-free system.
- It accesses the memory and immediately provides the requested word (*load-through* or *early restart*). This technique requires a higher cost in terms of cache hardware, but miss situations have a more limited impact on cache performance.

Cache position

The cache is normally located between the CPU and the bus, rather than between the main memory and the bus.



Cache position

The cache is normally located between the CPU and the bus, rather than between the main memory and the bus.

The benefits that are obtained in this way are:

- the bus load is reduced
- the solution is compatible with a multiprocessor architecture.

Instruction Cache and Data Cache

In some cases, there are separate caches for data and instructions; in other cases, there is only one for instructions and data.

The cache for instructions is typically easier to handle than data, as the instructions can not be changed (no write operations on the instruction cache).

Harvard architecture

If there are two caches, the architecture of the system falls into the scheme known as *Harvard architecture*, characterized by the existence of two separate data and code memories.

Harvard architecture contrasts with *von Neumann architecture*.

Characteristic parameters

They are:

- Cache size
- Block size
- Mapping
- Replacing algorithm
- Main memory update mechanism.

Cache size

Choosing the optimal size of the cache is very important for the system cost and the performance.

As the size increases

- The cost increases
- The system performance improves
- The cache itself becomes slower.

Frequent sizes range from a few kB to a few MBs.

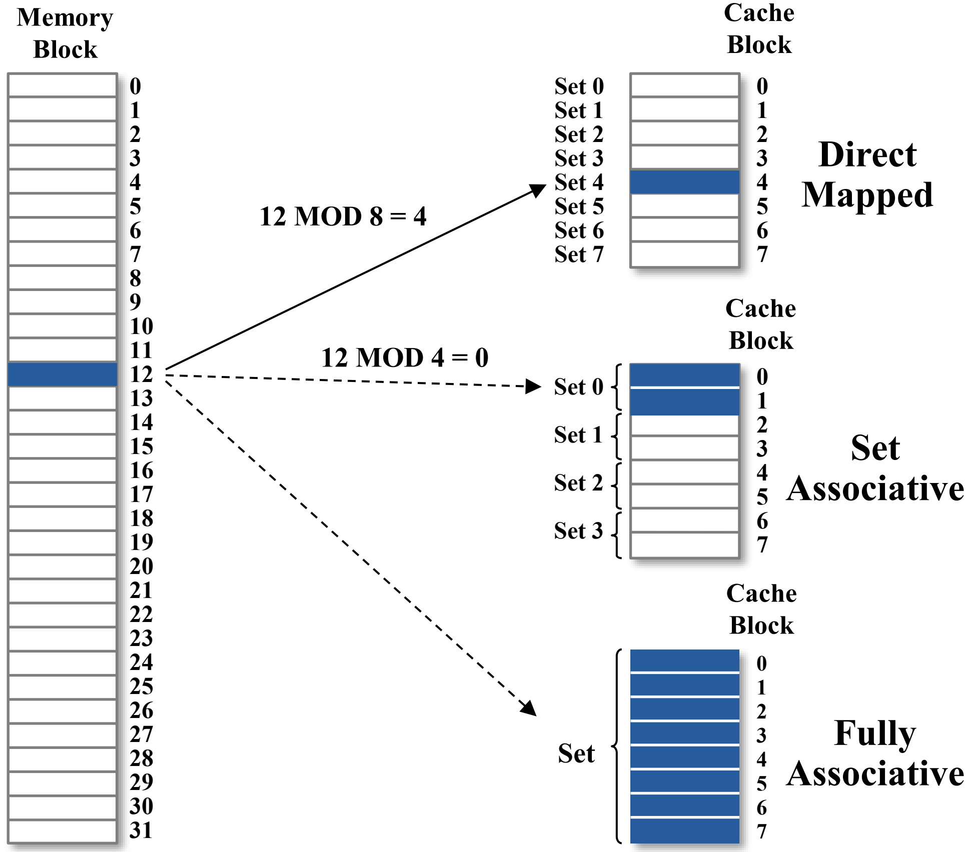
Mapping

The mapping mechanism defines in which line of the cache a certain memory block is written when it is uploaded in the cache.

You must ensure (at an acceptable cost) that the cache can quickly verify if it contains the data corresponding to a certain address.

Mapping (II)

Associativity Models



Finding the Set in the cache

# of Memory Block
MOD
Total of Blocks in Cache

# of Memory Block
MOD
$$\frac{\text{Total of Blocks in Cache}}{\text{Associativity level of cache}}$$

Direct Mapping

Each memory block *i* is statically associated to a set *k* in the cache using the expression

$$k = i \bmod N$$

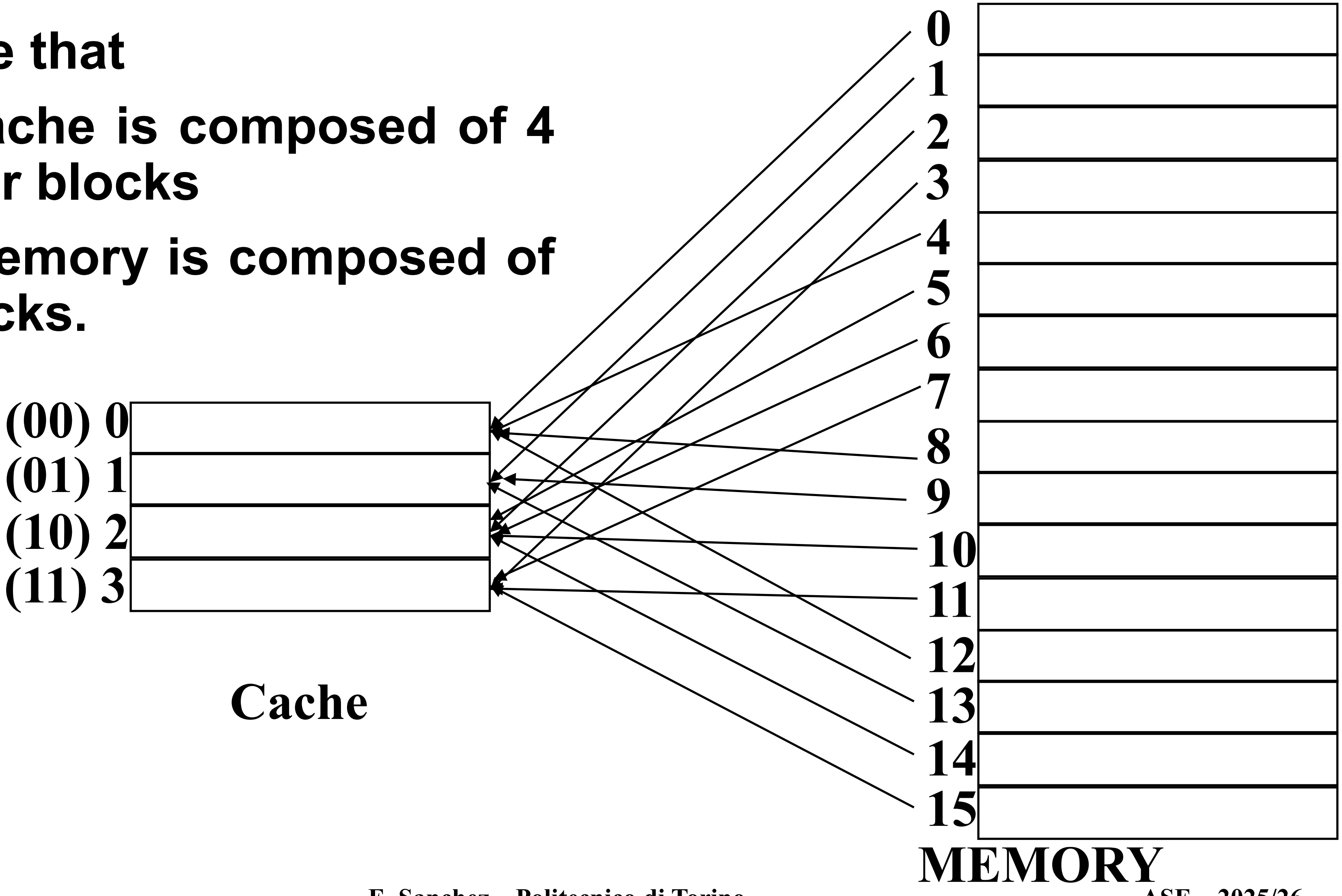
where *N* is the number of lines or blocks in the cache.

The computation of *k* can be easily performed by just taking the least significant bits in the value *i*.

Direct Mapping: example

Let assume that

- The cache is composed of 4 lines or blocks
- The memory is composed of 16 blocks.



Direct Mapping

Advantages:

- the mechanism can be easily implemented in hardware (the least significant block identifier bits identify the cache line)

Disadvantages:

- if the program frequently accesses 2 blocks corresponding to the same cache line, a miss occurs at each memory access.

Set Associative Mapping

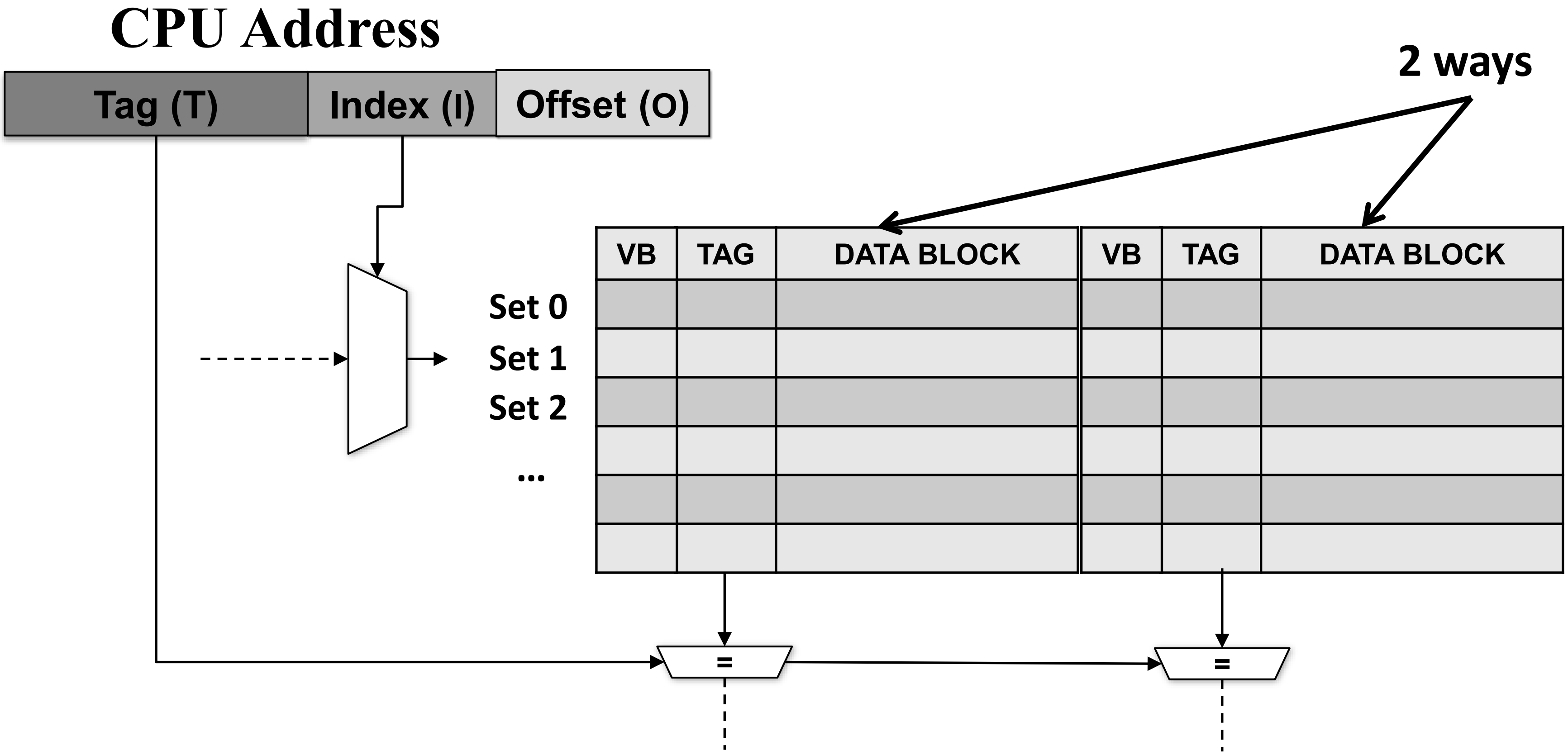
Characteristics:

- the cache lines or blocks are subdivided into *S* sets, each consisting of *W (ways)* lines
  - S = N / W where N is the number of cache lines or blocks
- a memory block *i* is associated with the set *k*, with
  - *k = i mod S*
- the block *i* can be put into any of the *W* lines of the set k.

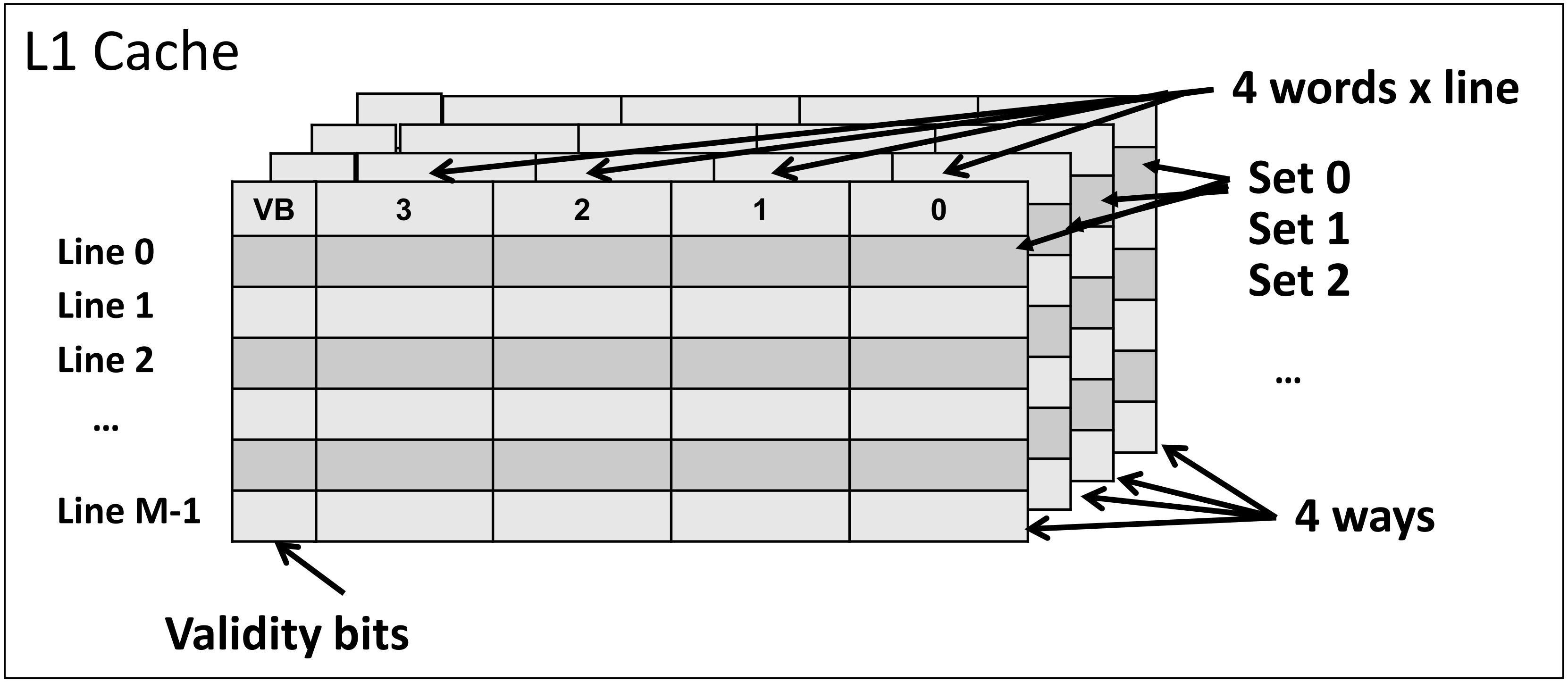
A set associative cache with *W* lines in each set is called a *W-ways* cache.

Common values for *W* are 2 and 4.

Set Associative Mapping



Set Associative Mapping



Fully Associative Mapping

Each block of the main memory can be stored in any cache block.

Advantages:

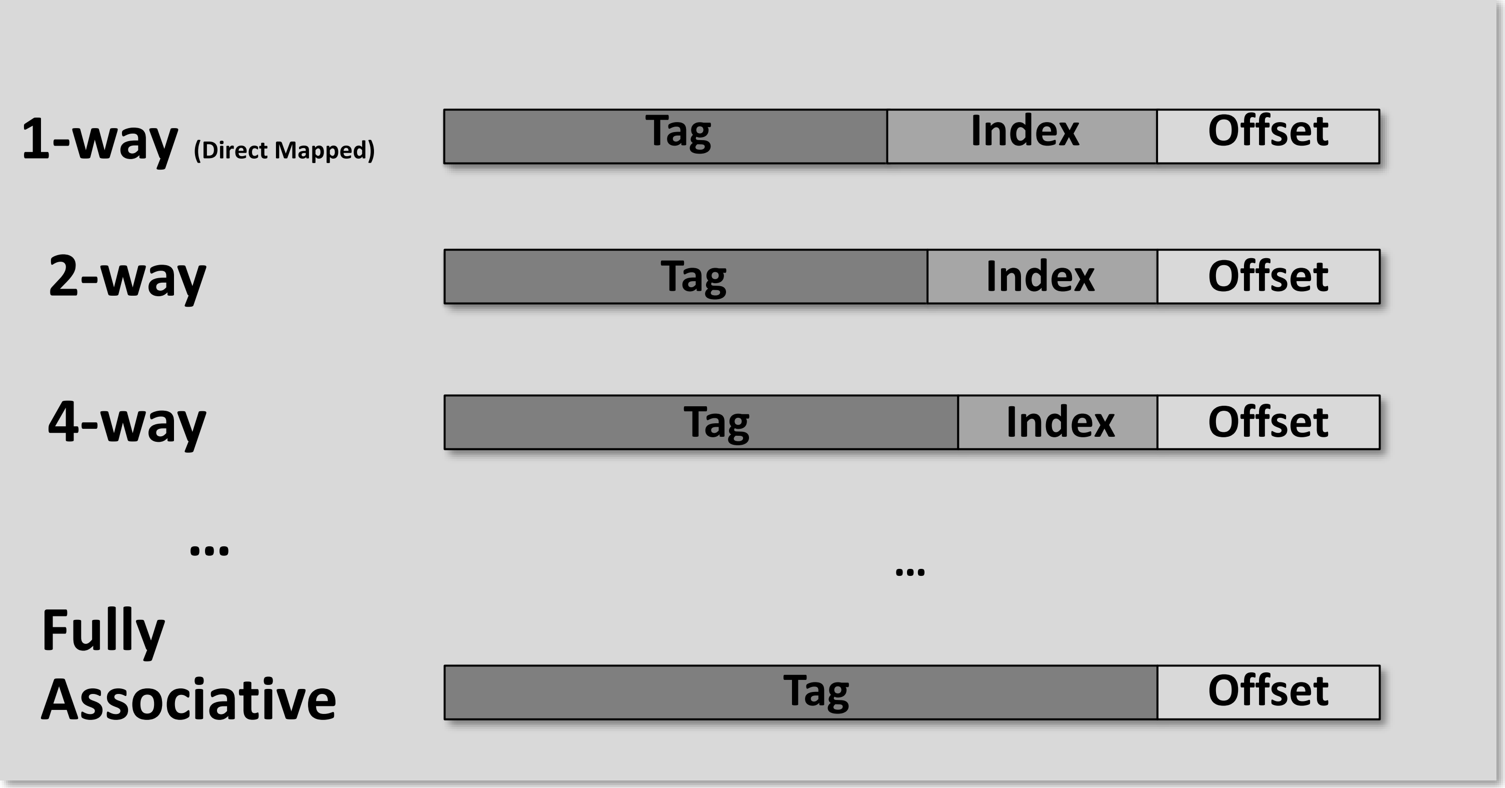
- maximum flexibility in choosing the cache block to use

Disadvantages:

- complexity of search hardware (usually adopting associative memory).

Effects of associativity

Effect of the cache associativity level on the address bits



1-way (Direct Mapped)
Tag | Index | Offset

2-way
Tag | Index | Offset

4-way
Tag | Index | Offset

…

…

Fully Associative
Tag | Offset

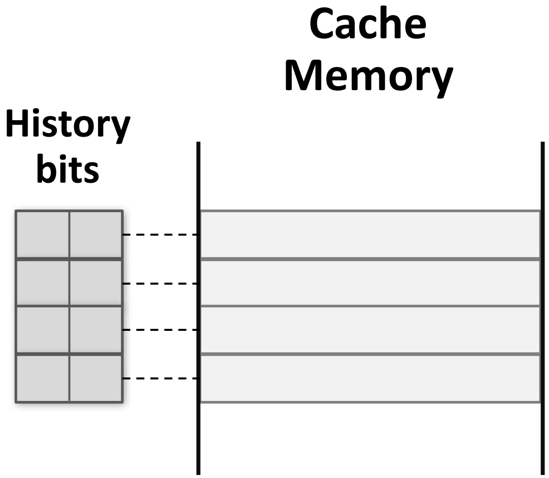
Replacing algorithm

It defines which cache line should be used to store a memory block, amongst those associated with the block (in the case of associative or set associative mapping).

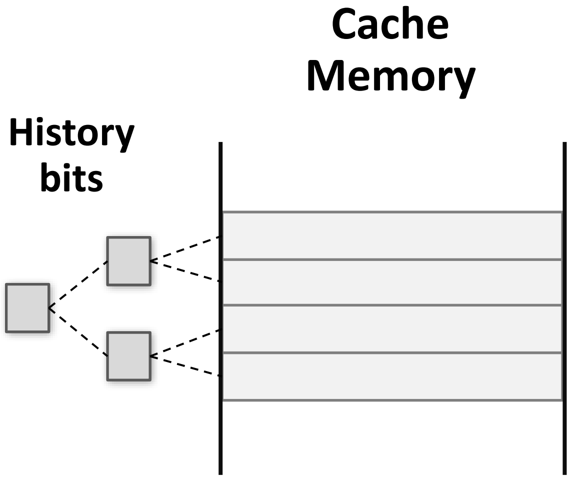
It is chosen from:

- *LRU* (Least Recently Used): the most used
- *FIFO* (First-In First-Out): the cheapest
- *LFU* (Least Frequently Used): theoretically, the most effective
- *random*: simple and effective.

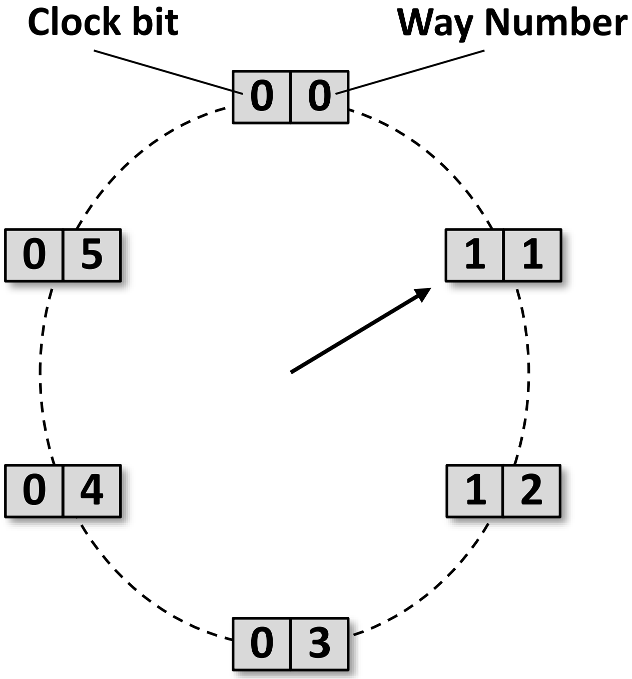
Replacement algorithm (II)



4-way
LRU



4-way
pseudo-
LRU

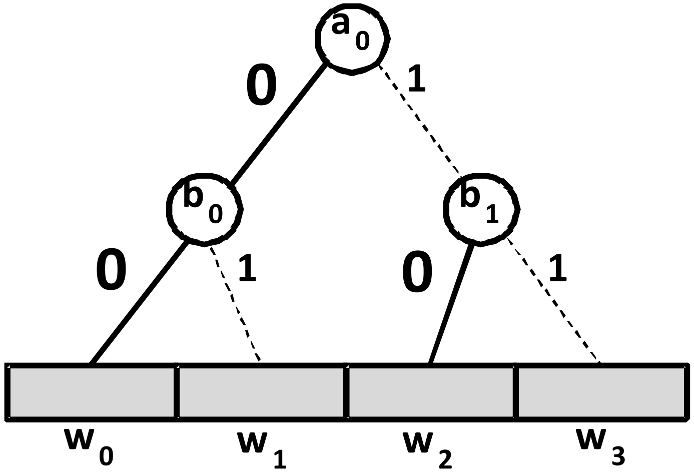


Clock
(FIFO-SC)

Example: pLRU replacement algorithm

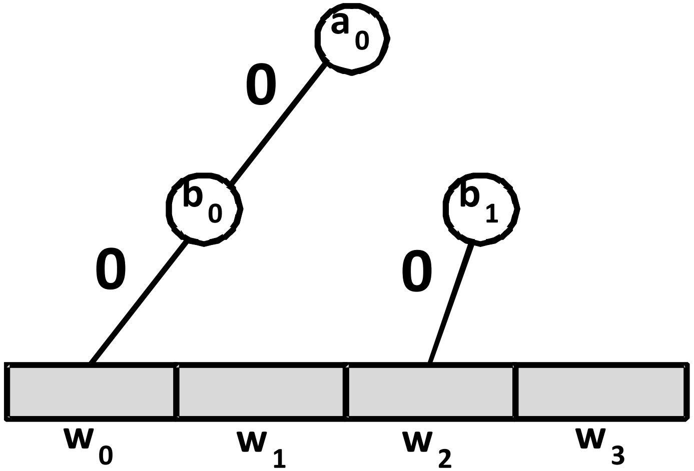
pLRU is an efficient approximation of a LRU algorithm

- The *age* of the cache ways is arranged in a binary tree
- Every node represents a *history bit*
- Access Order for a Way: *AOW_x*
  - $AOW_0 = a_0 + b_0$
  - $AOW_1 = a_0 + \overline{b_0}$
  - $AOW_2 = \overline{a_0} + b_1$
  - $AOW_3 = \overline{a_0} + \overline{b_1}$



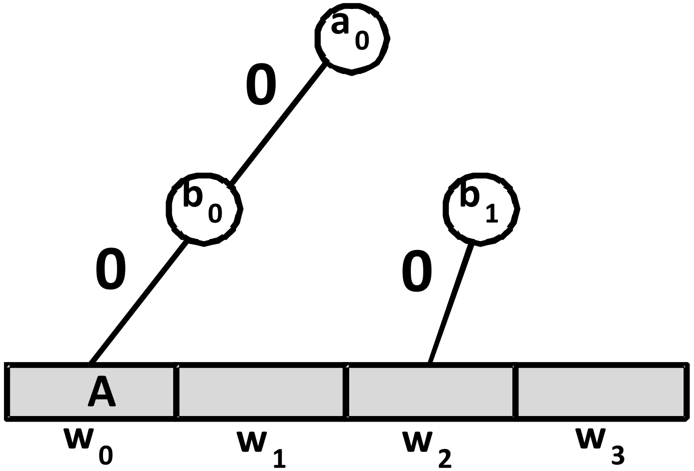
Example: pLRU replacement algorithm

Assume following data to be written: <u>A</u> B C D E



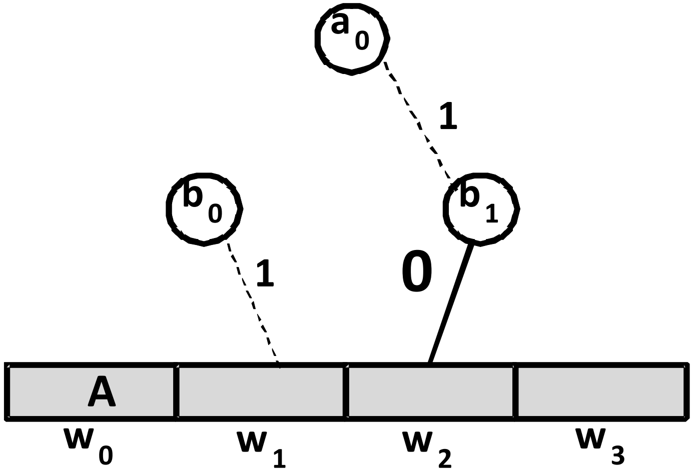
Example: pLRU replacement algorithm

Assume following data to be written: <u>A</u> B C D E



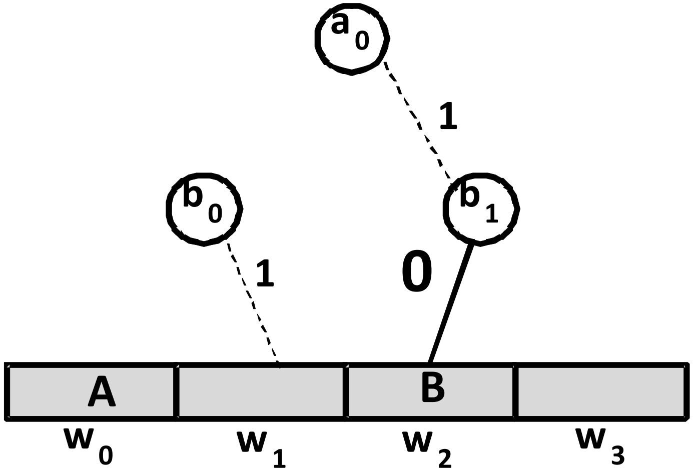
Example: pLRU replacement algorithm

Assume following data to be written: A <u>B</u> C D E



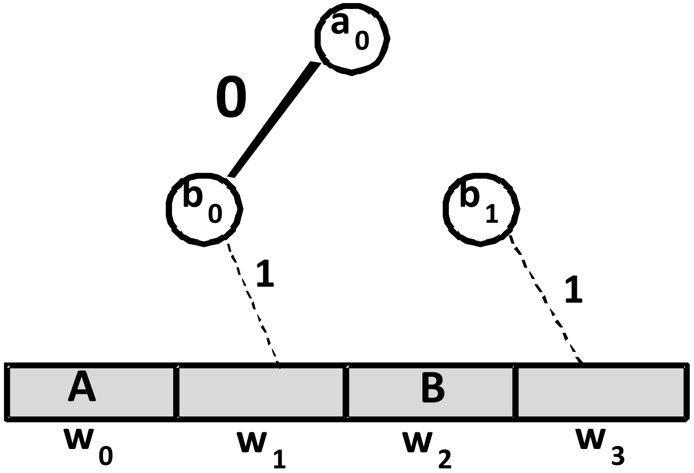
Example: pLRU replacement algorithm

Assume following data to be written: A B C D E



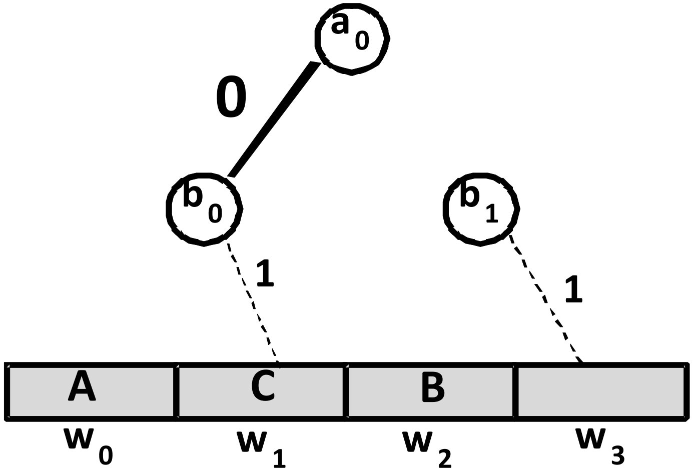
Example: pLRU replacement algorithm

Assume following data to be written: A B <u>C</u> D E



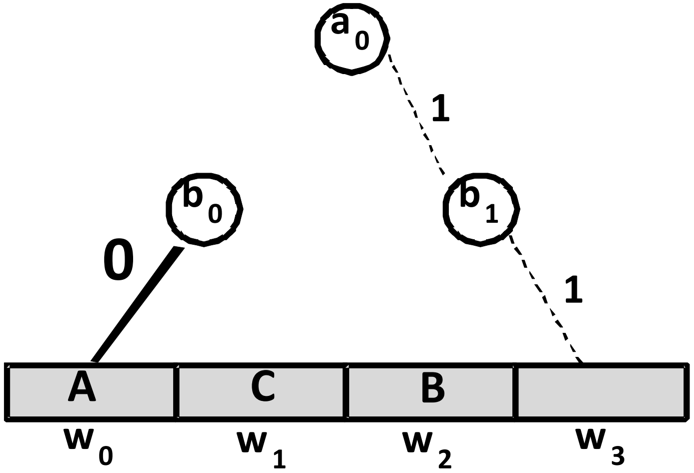
Example: pLRU replacement algorithm

Assume following data to be written: A B <u>C</u> D E



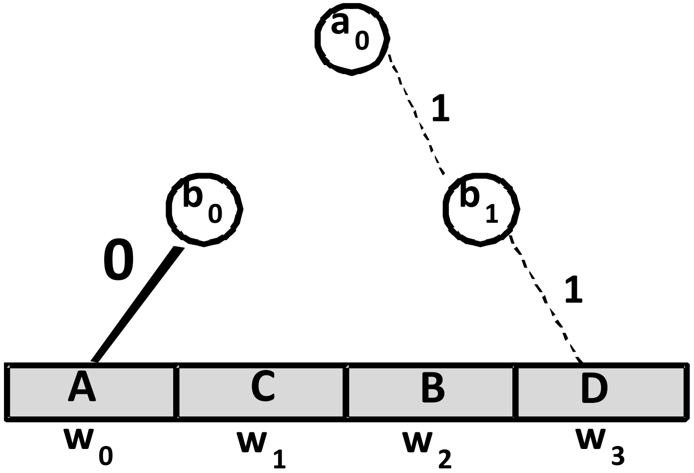
Example: pLRU replacement algorithm

Assume following data to be written: A B C <u>D</u> E



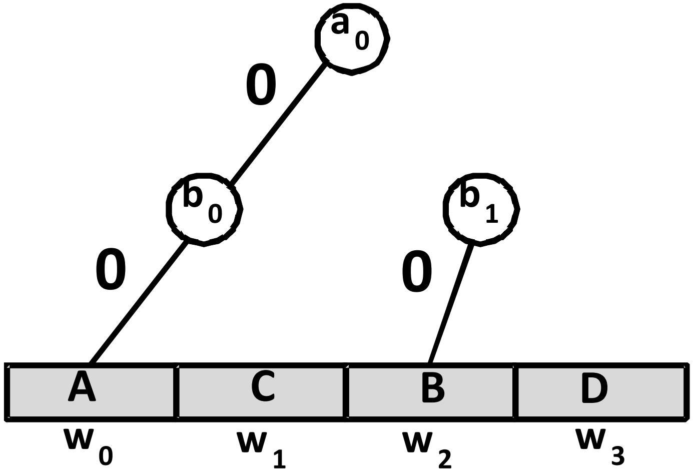
Example: pLRU replacement algorithm

Assume following data to be written: A B C D E



Example: pLRU replacement algorithm

Assume following data to be written: A B C D E



Main memory update

When a write operation is performed on a data in the cache, we also need to update the main memory.

Two solutions can be adopted:

- ***write-back***
- ***write-through.***

Write-Back

For each cache block, a flag (called *dirty bit*) is introduced, which remembers whether or not the block has been changed since it was loaded into the cache.

When a block is evicted from the cache and the dirty bit is set, the block is copied from the cache to the main memory.

Disadvantages:

- the replacement is slower because it sometimes requires copying in the main memory the replaced block
- in multiprocessor systems there may be incoherence between the caches of different processors
- it may not be possible to restore memory data after possible system failures.

Write-through

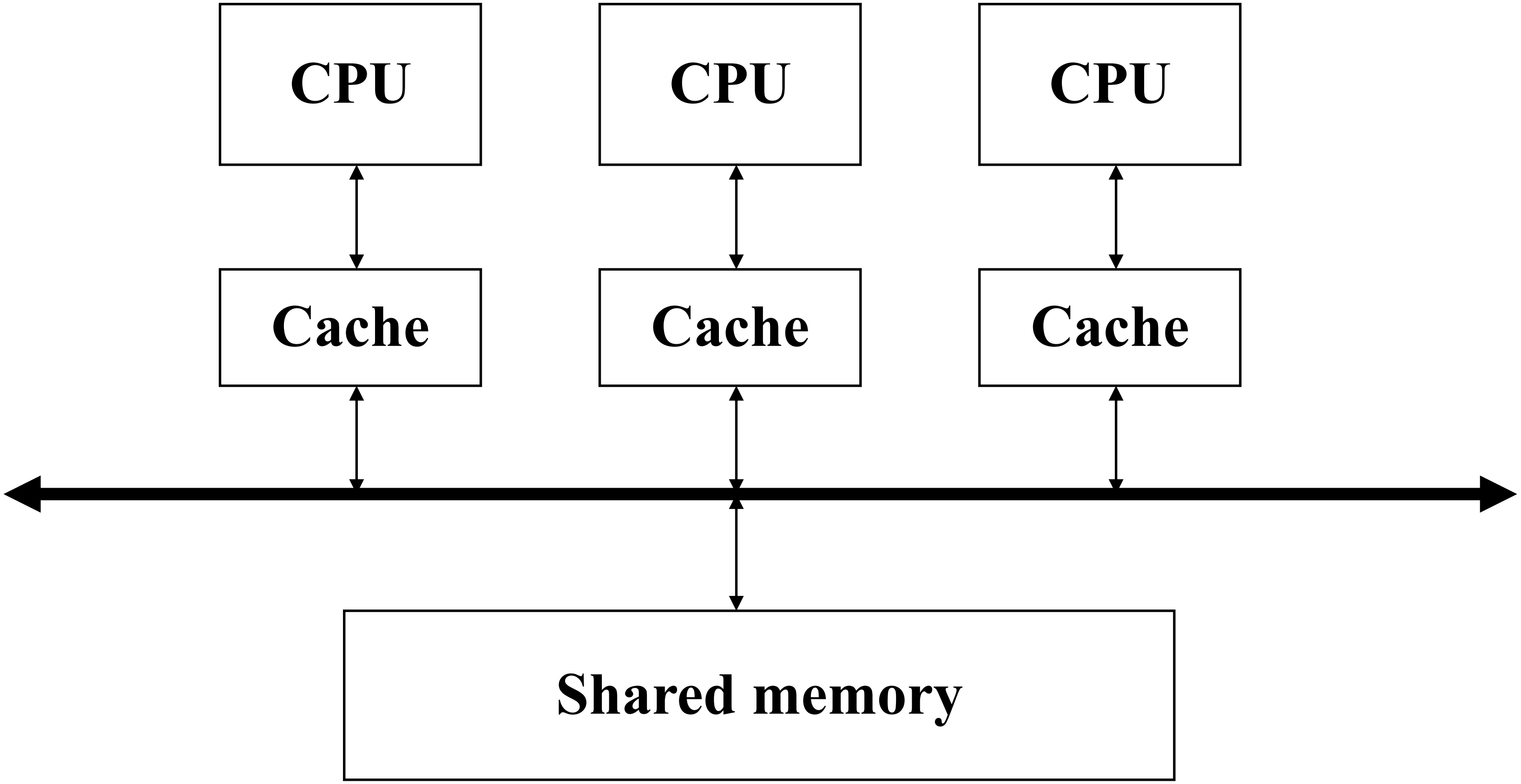
Each time the CPU performs a write operation, it writes on both the cache data and the main memory data.

The resulting loss of efficiency is limited by the fact that writing operations are usually much less numerous than reading ones.

Cache coherence

This is a major problem in multiprocessor systems with shared memory, in which each processor has its own cache.

Similar problems may occur if the system uses a DMA controller.



Validity bit

To achieve cache coherence, a *validity bit* is introduced for every cache line.

If it is disabled, it means that any access to the block must produce a miss.

At the power-on, the validity bits of all cache lines are disabled.

First, second, third level caches

It may be convenient to have multiple levels of caches:

- **a first level cache (L1), smaller and faster**
- **a second level cache (L2), slower but larger**
- **a third level cache (L3), even slower and bigger.**

First, second, third level caches: behavior

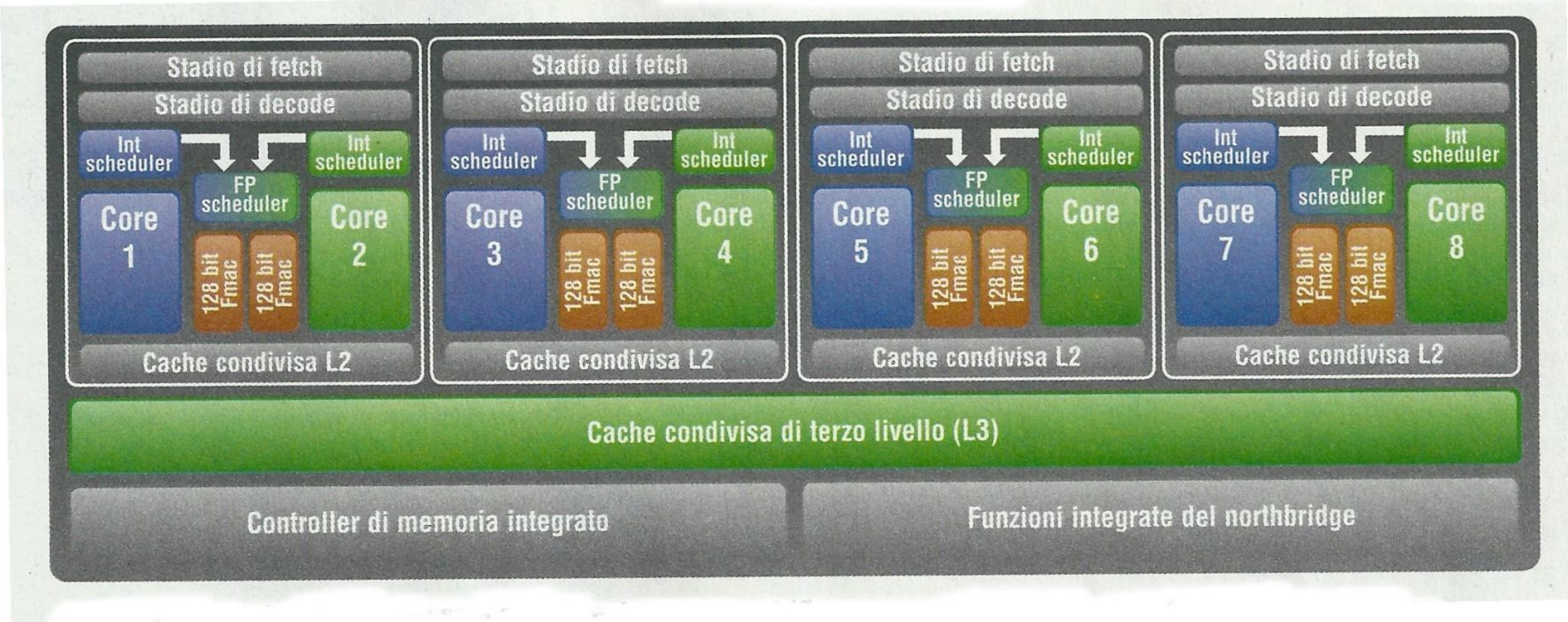
Each time the processor performs a memory access

- It checks first whether the word is in L1
- If so, it can access the word in L1
- If not, it checks whether the word is in L2
  - If so, it will access to the word in L2 and eventually update L1
  - If not, it checks whether the word is in L3
    - If yes, L3 is accessed and L2 is updated
    - If not, it accesses to the main memory and eventually updates L3.

Example: AMD Zambezi

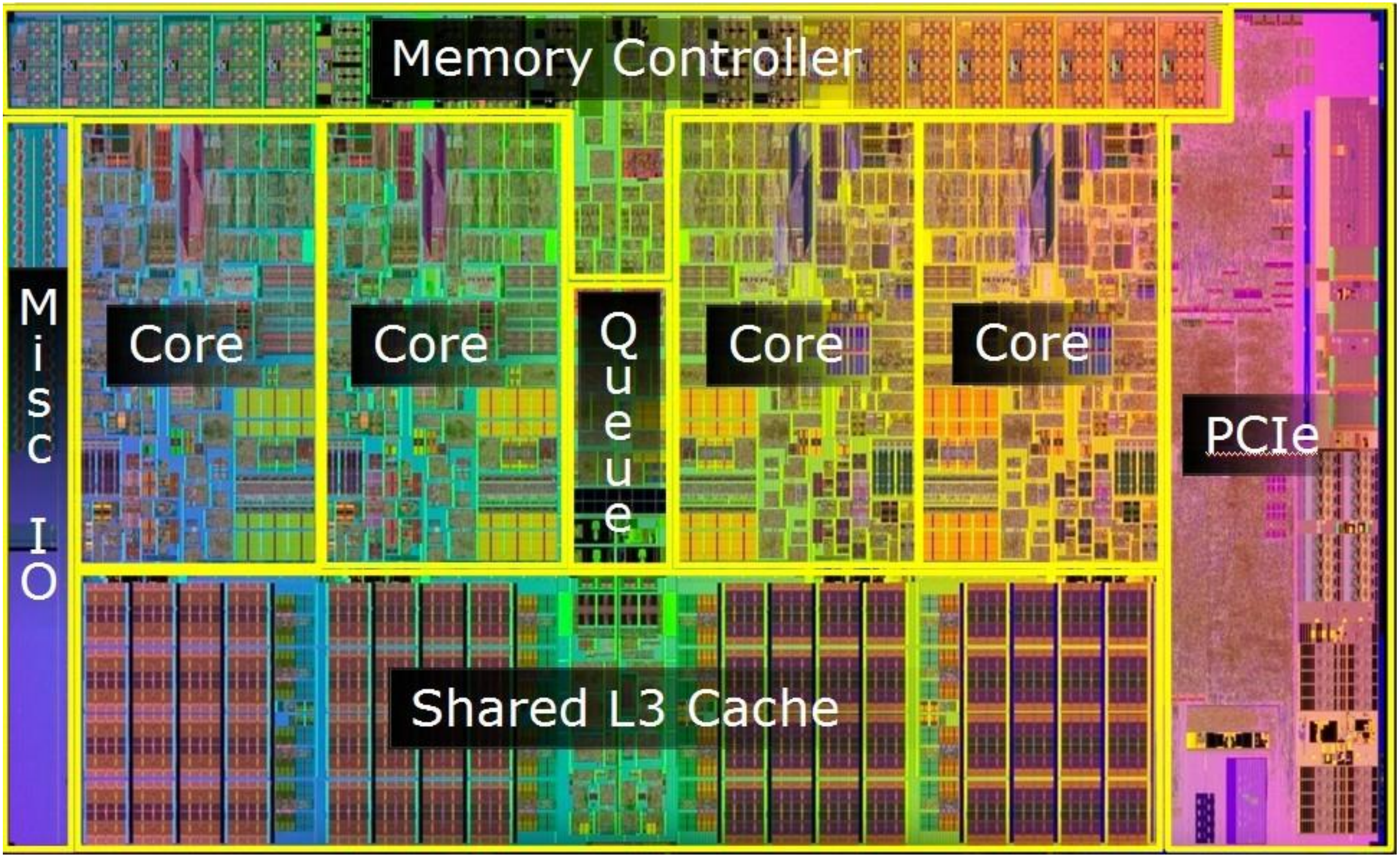
Zambezi is the high end CPU of the AMD Fusion family: it includes up to 8 cores, each equipped with its first level cache (L1) of 192 kB.

Each core pair uses a second level (L2) cache of 2 or 4 MBytes. The cores of the same device share the third level cache (L3) of 8 MBytes.



Example: Intel Lynnfield

L1= 64 kB
L2= 256 kB
L3= 8MB



Example of cache size

Let us consider a cache with the following characteristics:

- 64 kByte size (data only)
- direct mapping
- 4 bytes blocks
- 32 bits addresses.

You are asked to determine the structure of the cache (number of lines, size of the tag field).

Example of cache size

Since each block is compose of 4 bytes, the Memory is composed of 2^{30} lines or blocks.

The number of lines in the cache is 64kB / 4 = 16kB =2^{14}.

The tag field identifies the block currently stored in each line.

Hence, the tag field should be composed of 30 bit.

However, since in the generic cache line only blocks whose index has the 14 least significant bits equal to the line index, the tag field is composed of 16 bits, only.

Hence, the total size of the cache is given by:

$$2^{14} \times (32 + 16) = 2^{14} \times 48 = 768\text{kbit} = 96\text{kByte}$$

Example II

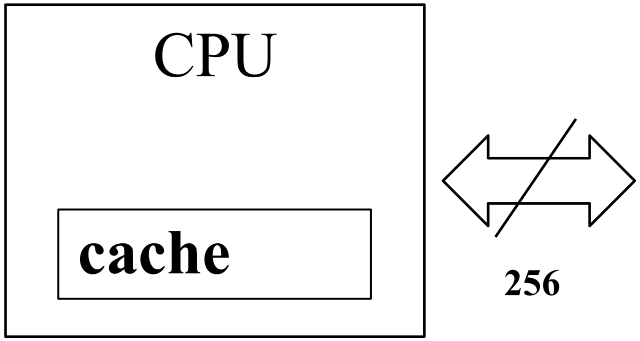
It is given a system composed by the following memorization elements

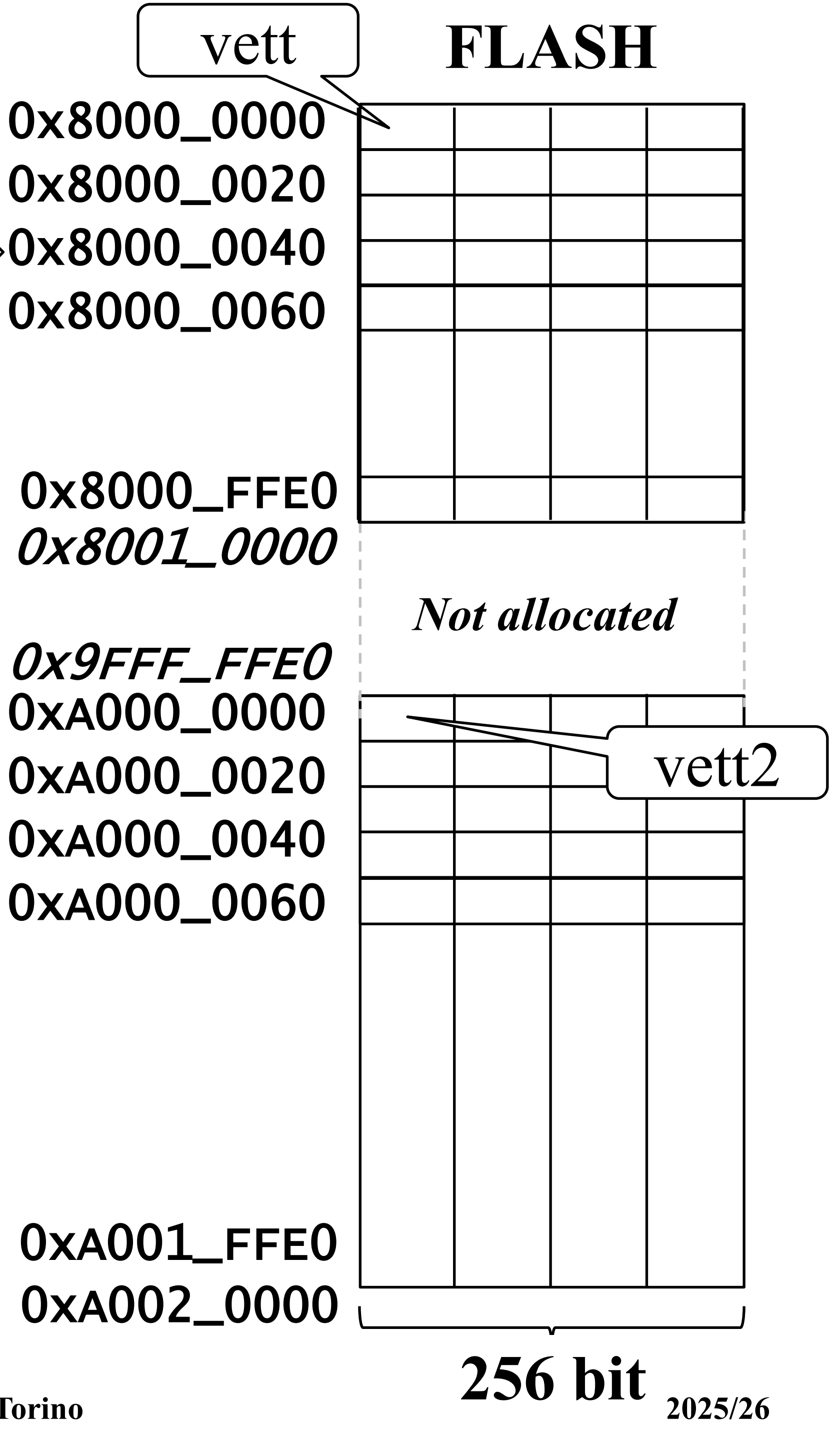
- A Central Data memory, located in many Flash cores
  - With non-contiguous addressing (see the illustration)
  - Every Flash can emit up to 256 bits (32 bytes) per read
- A cache memory of 4KB (data only), organized as 2-ways set associative
- CPU can fetch 64 bits, but cache lines contain 256 bits = 32 bytes
- Addresses are expressed on 32 bits

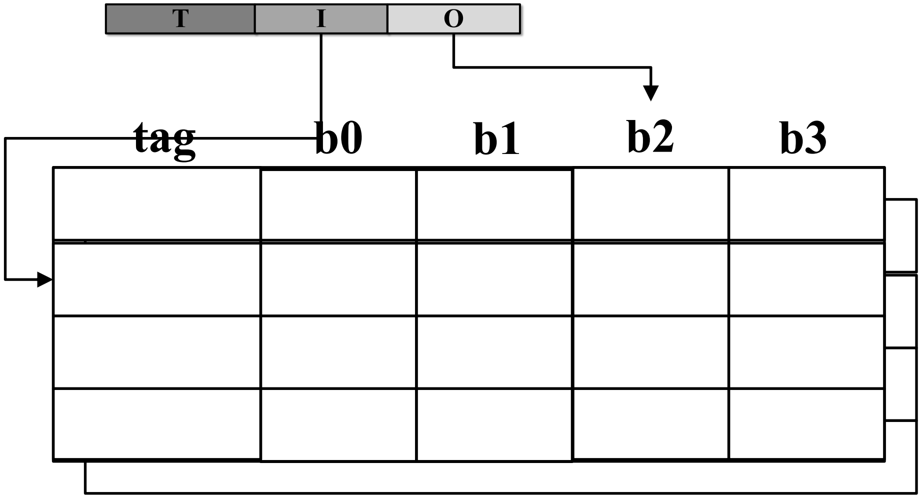
Determine the size of tag, index and offset that better fit the memory organization.

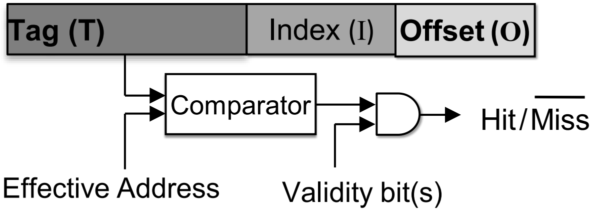
Determine the overall size of the cache memory.

```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

   add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```

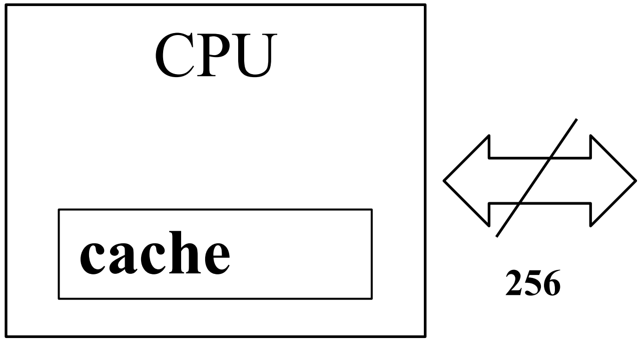


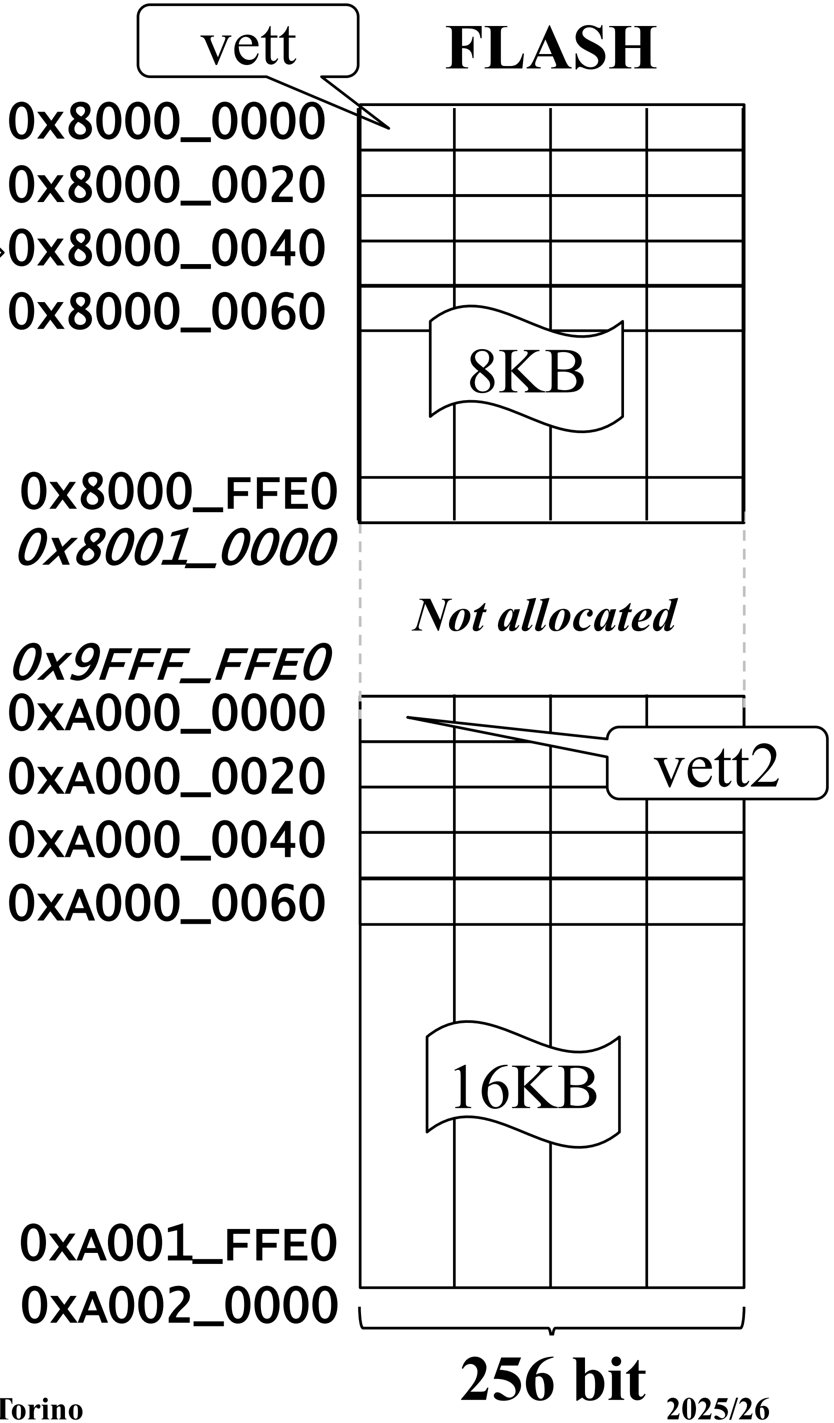


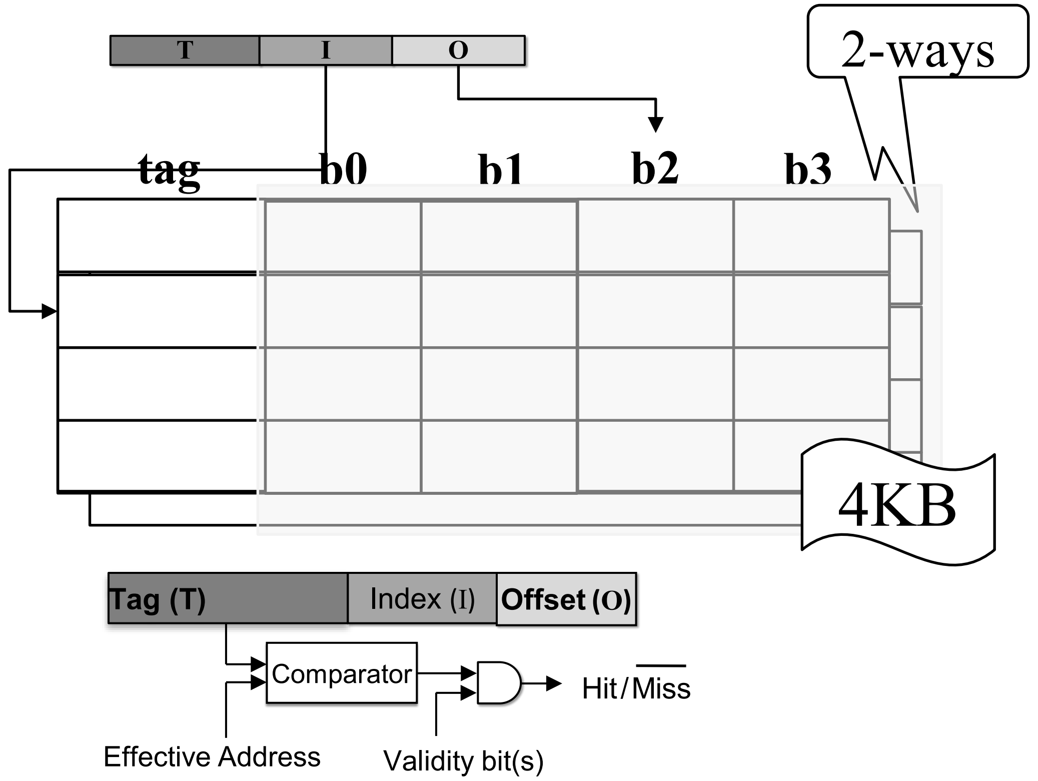




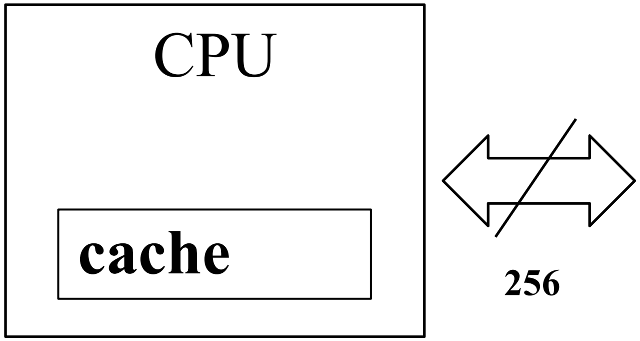
```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```

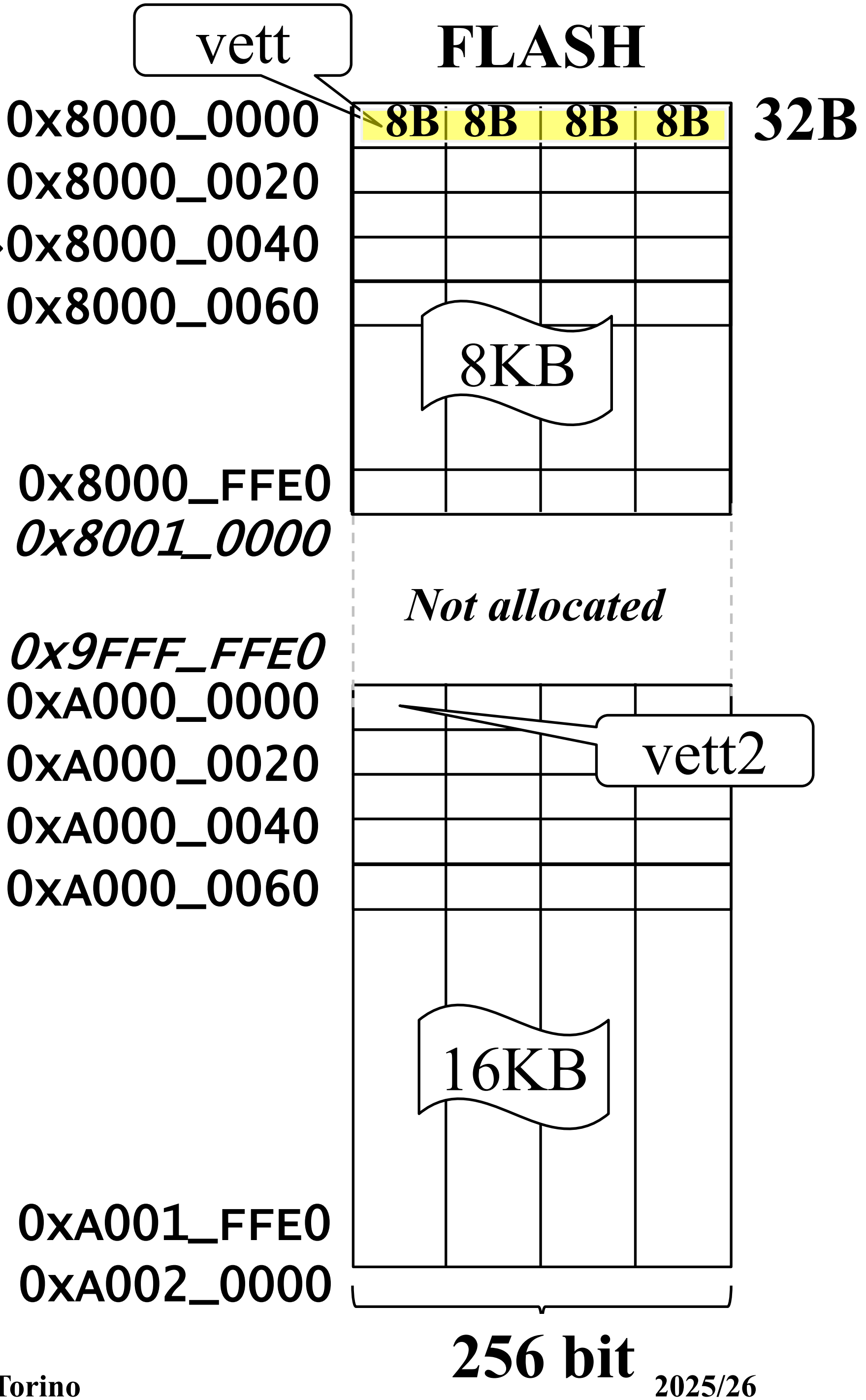


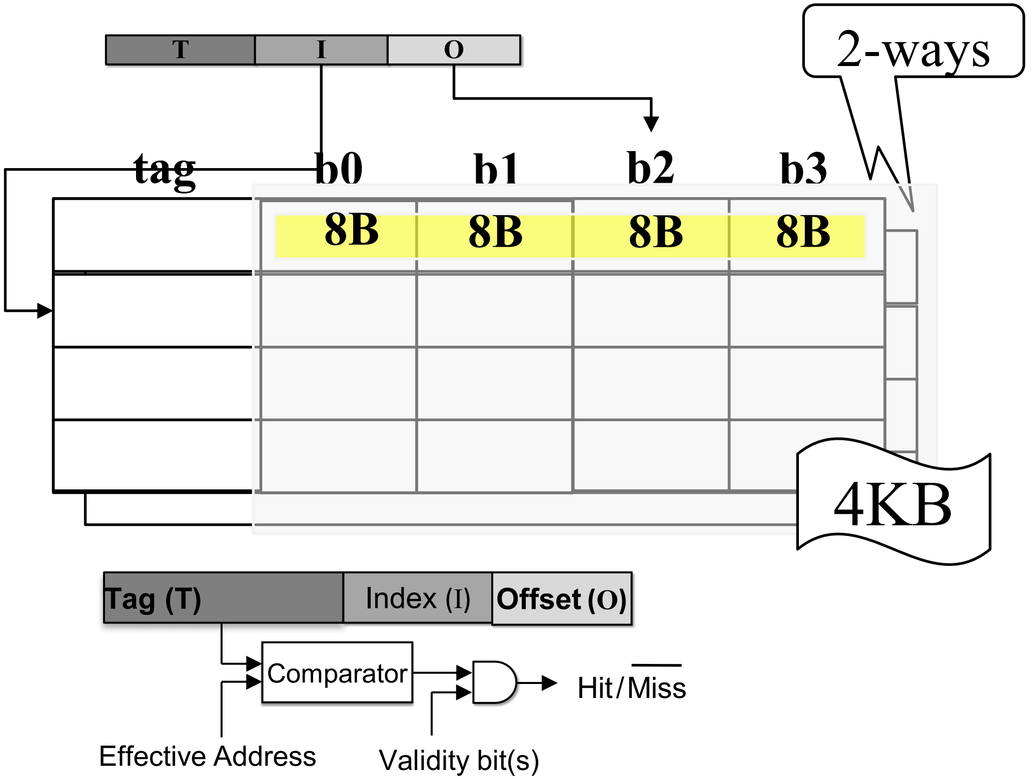




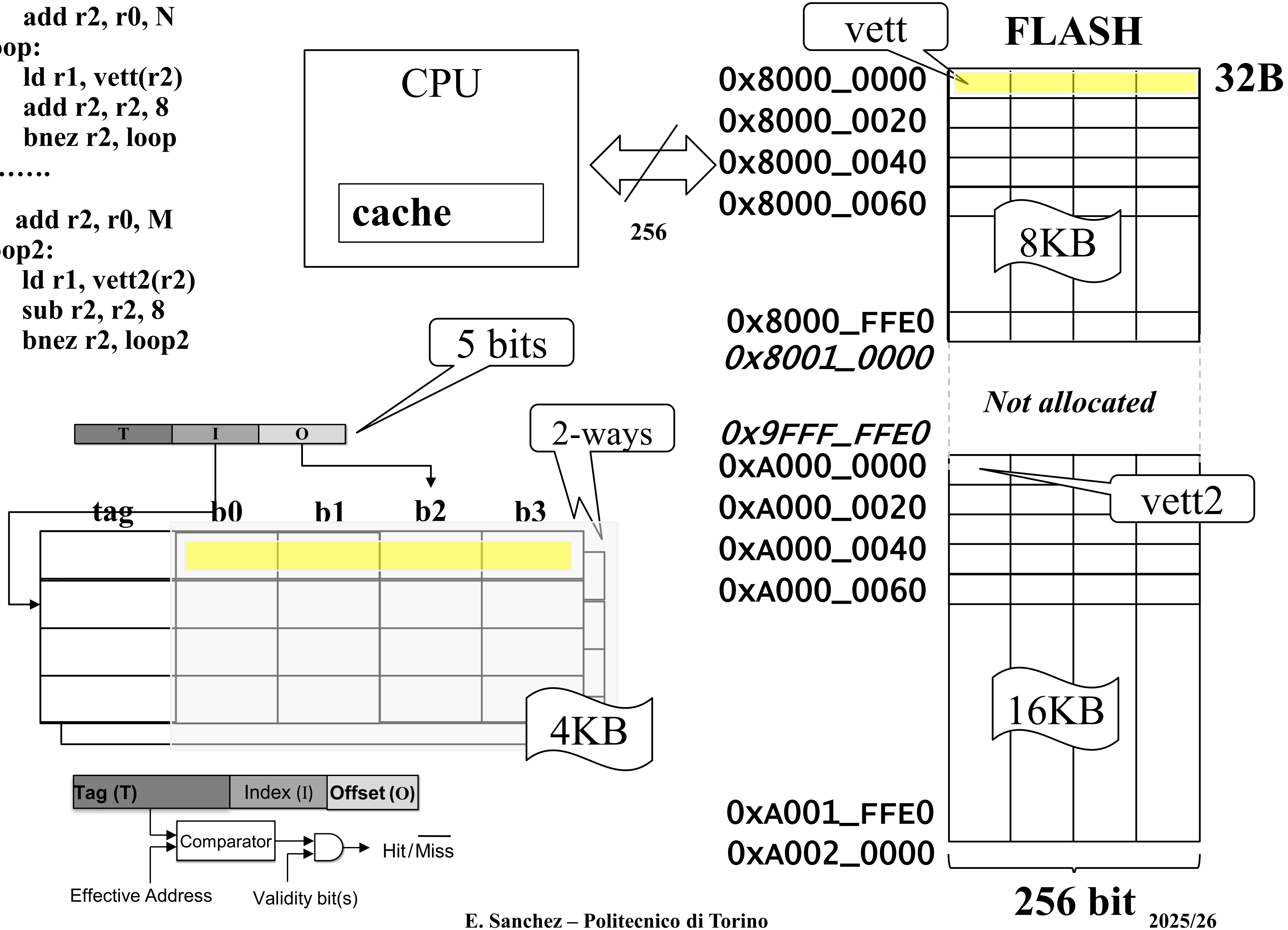
```
     add r2, r0, N
loop:
     ld r1, vett(r2)
     add r2, r2, 8
     bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



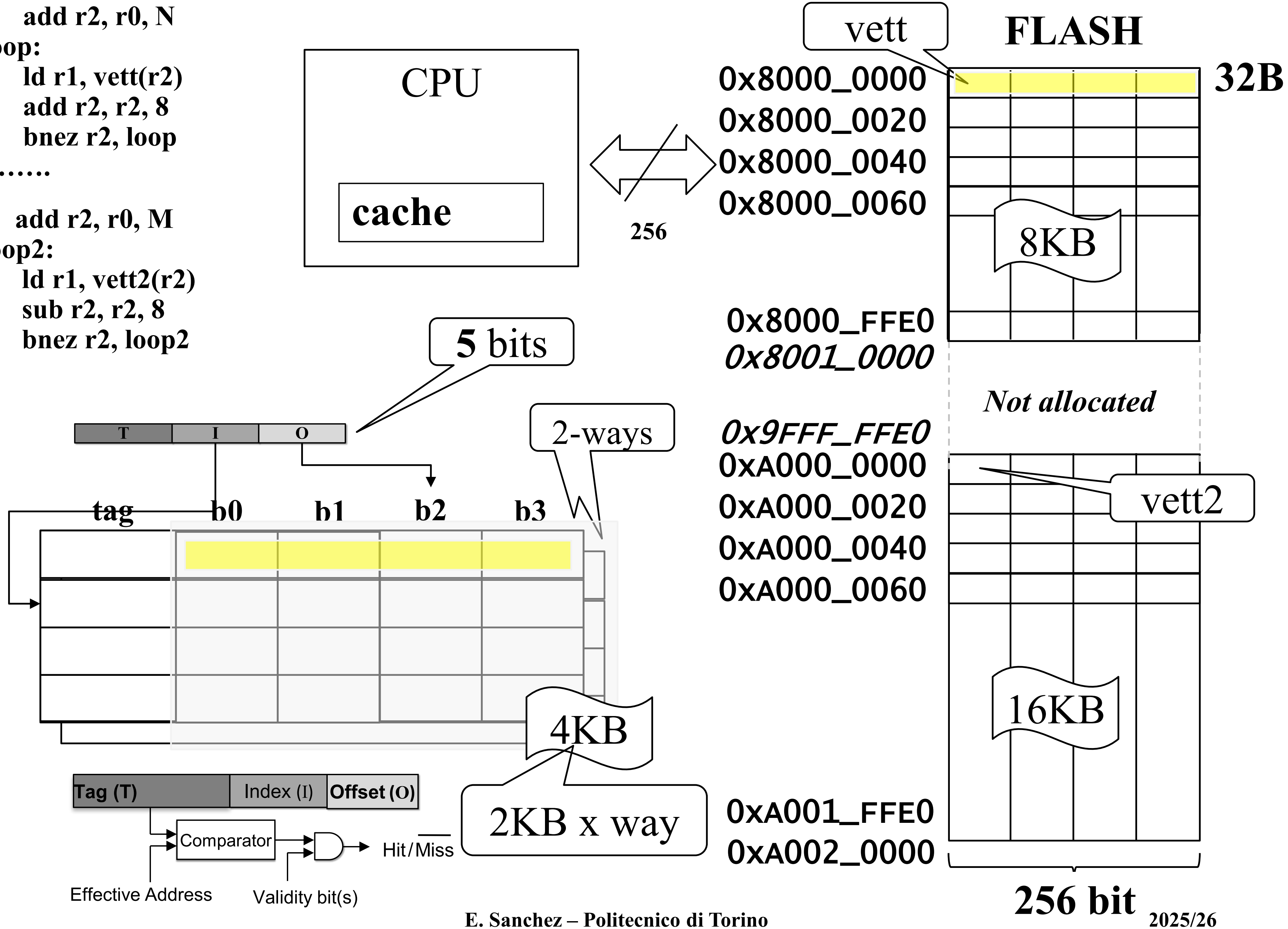




```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



```
     add r2, r0, N
loop:
     ld r1, vett(r2)
     add r2, r2, 8
     bnez r2, loop
 …….

    add r2, r0, M
loop2:
     ld r1, vett2(r2)
     sub r2, r2, 8
     bnez r2, loop2
```



CPU

cache

256

vett

FLASH

32B

0x8000_0000

0x8000_0020

0x8000_0040

0x8000_0060

8KB

0x8000_FFE0

0x8001_0000

Not allocated

0x9FFF_FFE0

0xA000_0000

0xA000_0020

0xA000_0040

0xA000_0060

vett2

16KB

0xA001_FFE0

0xA002_0000

256 bit

5 bits

T I O

2-ways

tag b0 b1 b2 b3

4KB

2KB x way

Tag (T) Index (I) Offset (O)

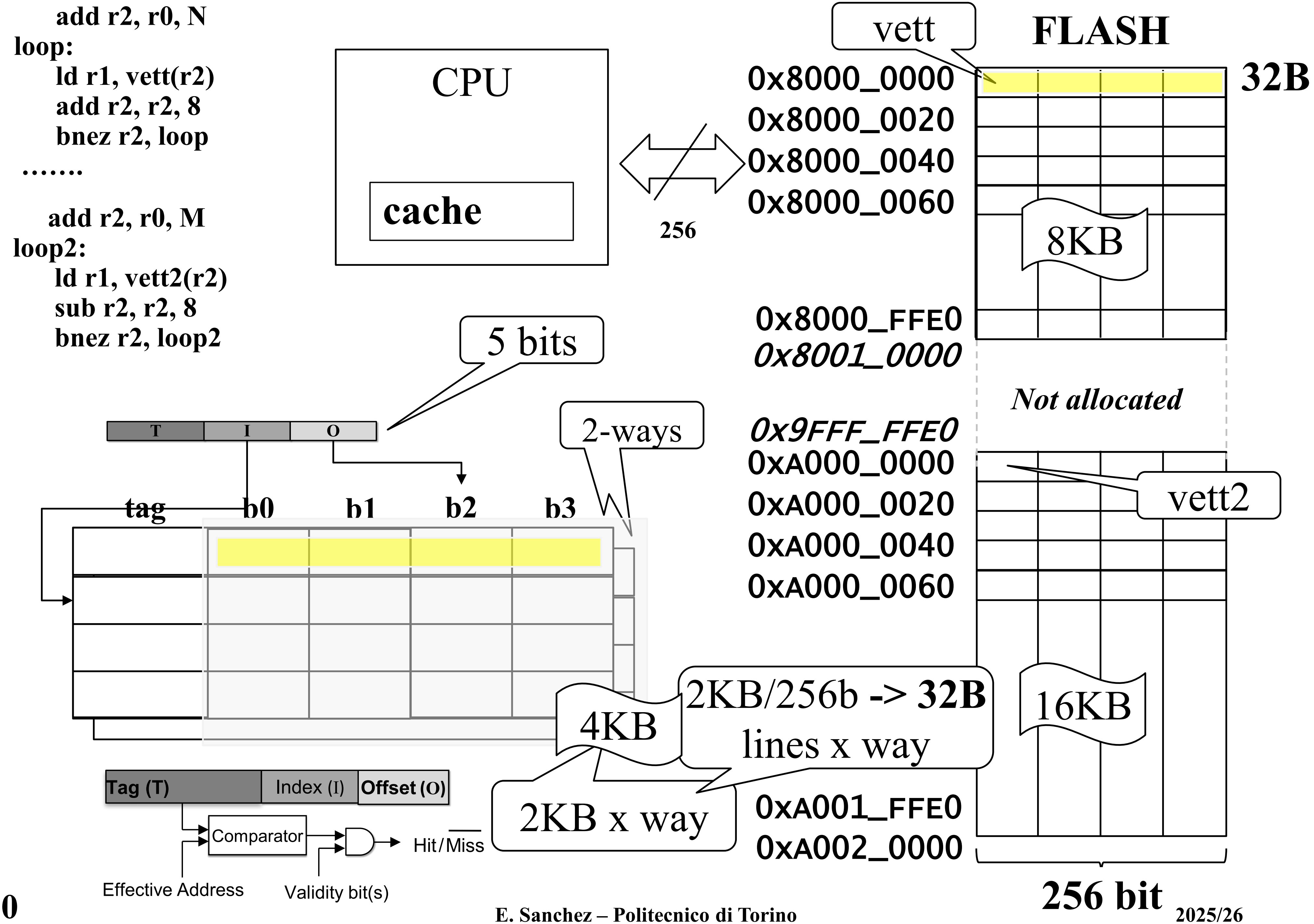
Comparator

Hit/Miss

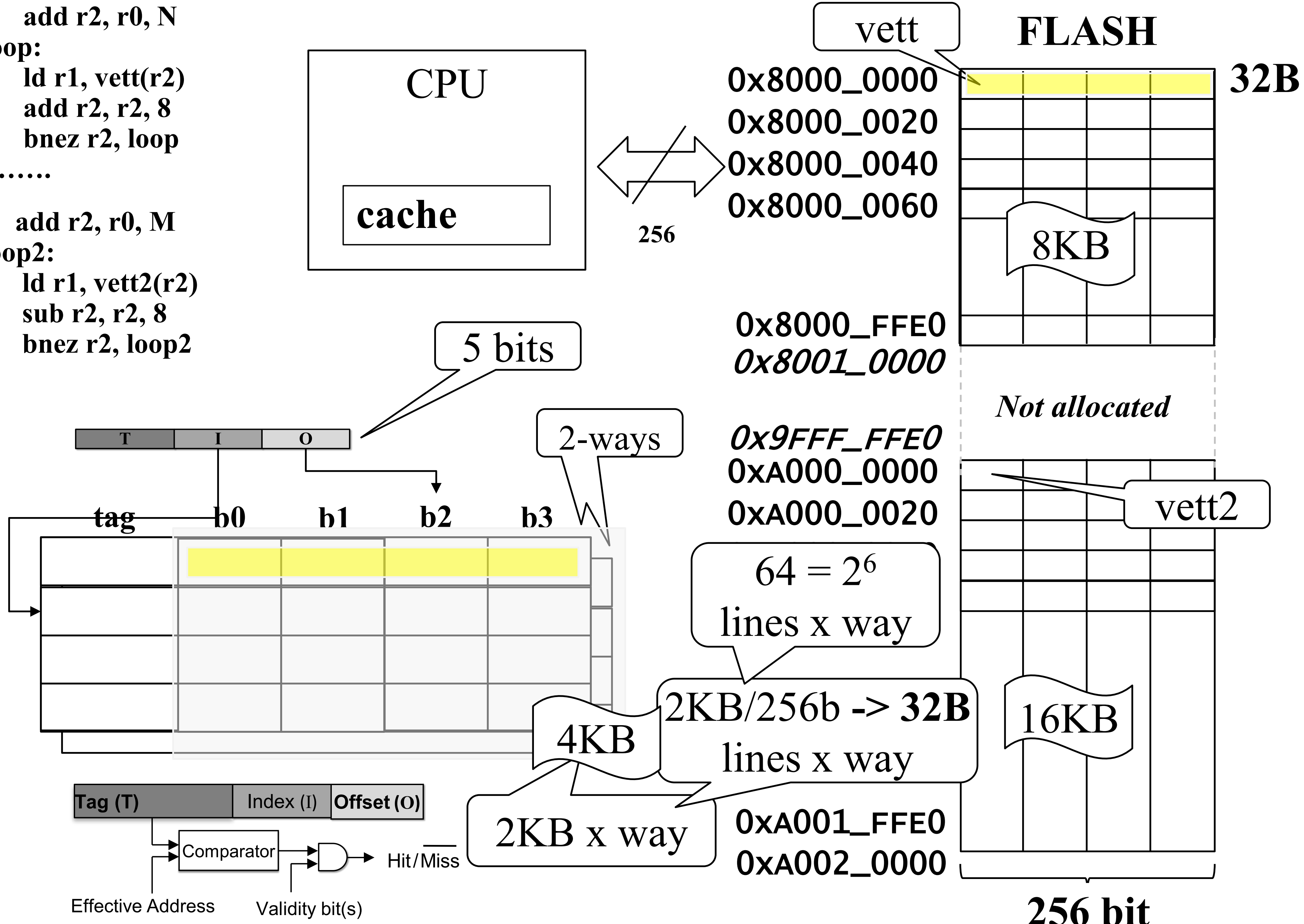
Effective Address

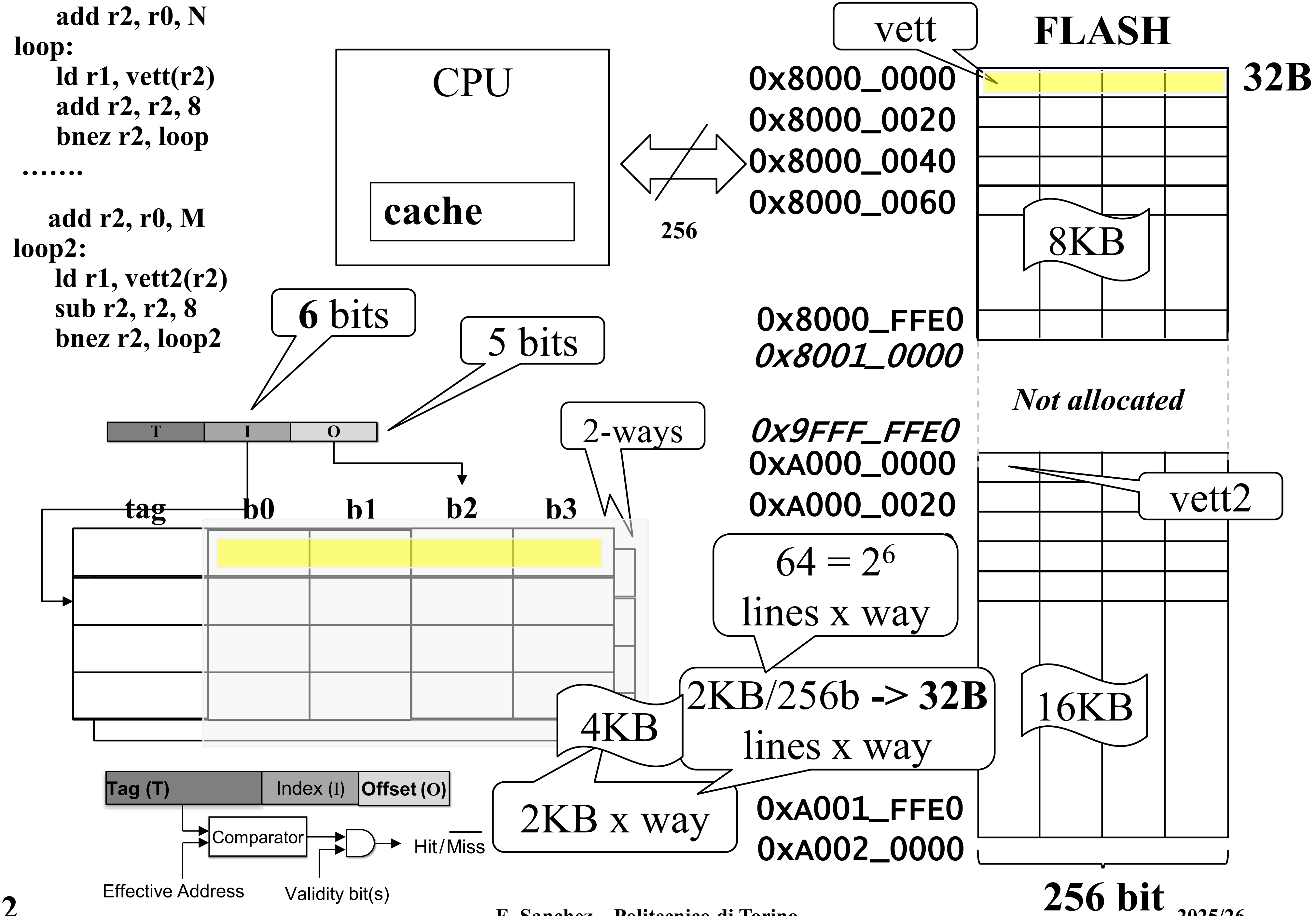
Validity bit(s)

```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

   add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



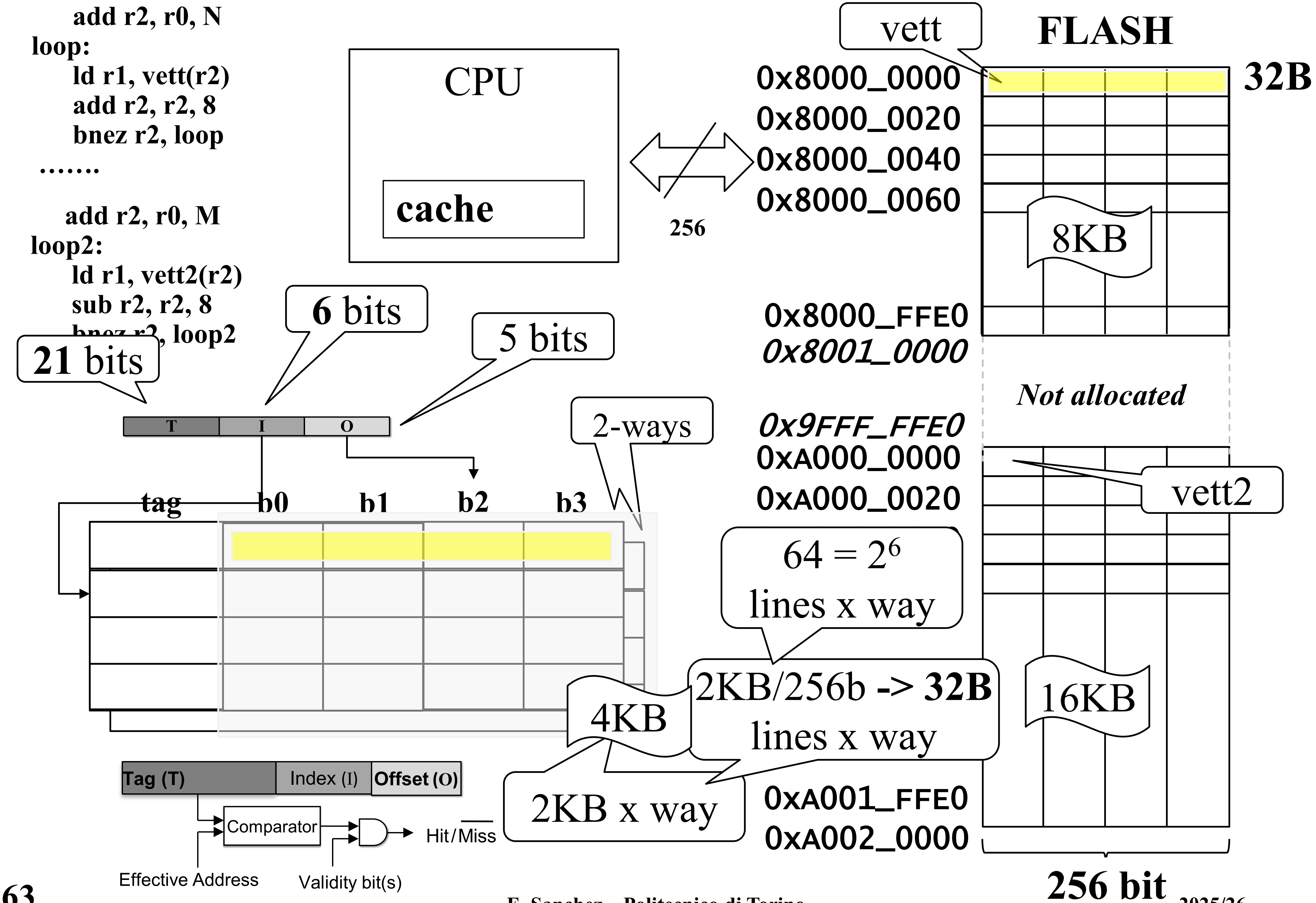
```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



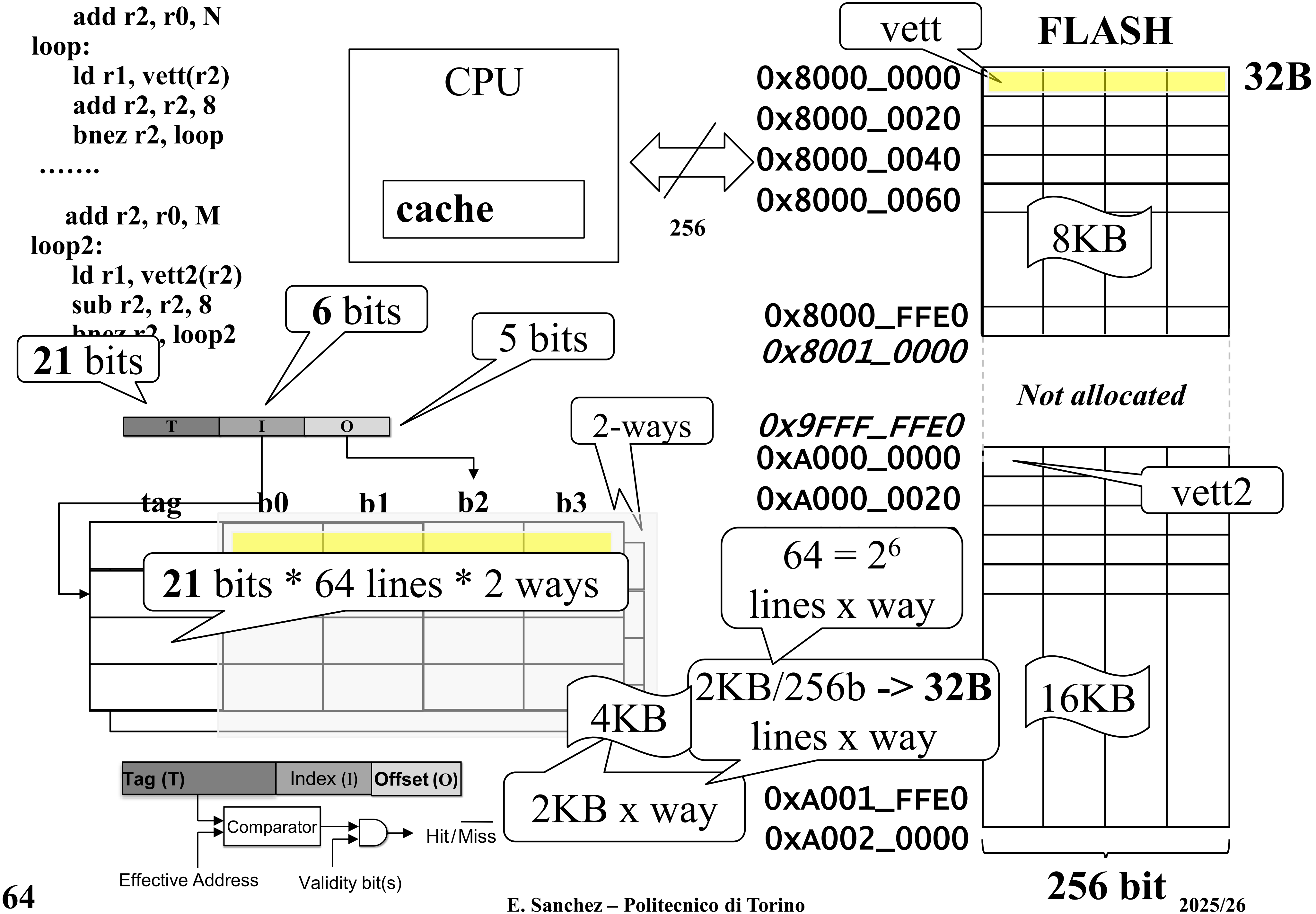


```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```

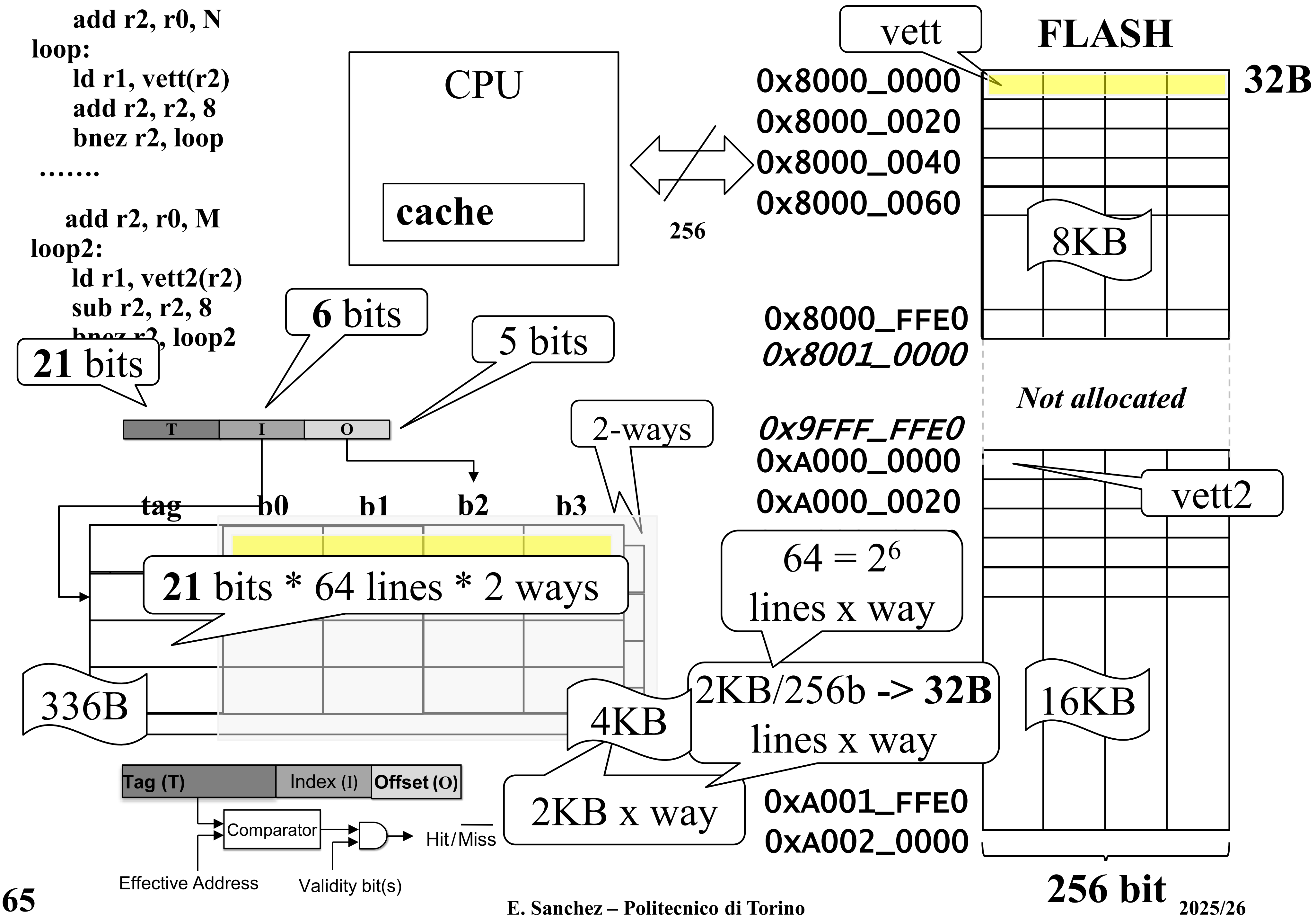
```
     add r2, r0, N
loop:
     ld r1, vett(r2)
     add r2, r2, 8
     bnez r2, loop
 …….

     add r2, r0, M
loop2:
     ld r1, vett2(r2)
     sub r2, r2, 8
     bnez r2, loop2
```



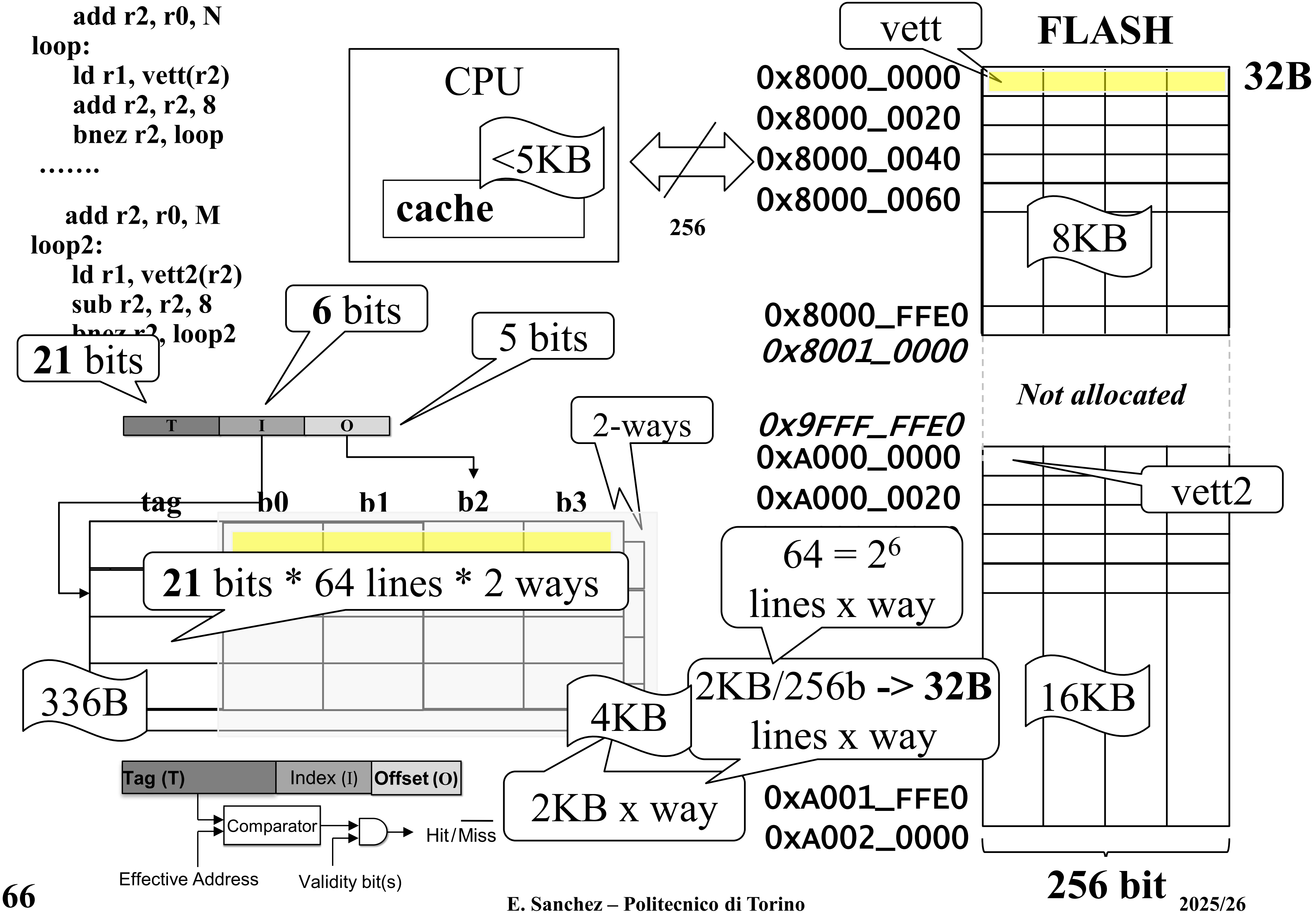
```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



```
    add r2, r0, N
loop:
    ld r1, vett(r2)
    add r2, r2, 8
    bnez r2, loop
 …….

    add r2, r0, M
loop2:
    ld r1, vett2(r2)
    sub r2, r2, 8
    bnez r2, loop2
```



```
     add r2, r0, N
loop:
     ld r1, vett(r2)
     add r2, r2, 8
     bnez r2, loop
 …….

     add r2, r0, M
loop2:
     ld r1, vett2(r2)
     sub r2, r2, 8
     bnez r2, loop2
```


CPU
<5KB
cache
256
vett
FLASH
32B
0x8000_0000
0x8000_0020
0x8000_0040
0x8000_0060
8KB
0x8000_FFE0
0x8001_0000
Not allocated
0x9FFF_FFE0
0xA000_0000
0xA000_0020
vett2
16KB
0xA001_FFE0
0xA002_0000
256 bit
21 bits
6 bits
5 bits
T
I
O
2-ways
tag
b0
b1
b2
b3
21 bits * 64 lines * 2 ways
336B
64 = 2^6
lines x way
2KB/256b -> 32B
lines x way
4KB
2KB x way
Tag (T)
Index (I)
Offset (O)
Comparator
Hit/Miss
Effective Address
Validity bit(s)