Topic V30

Cache Memory

Reading: (Section 5.3)

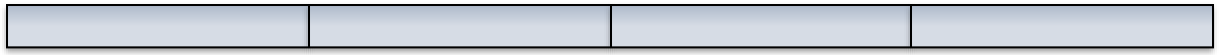
Cache Memory

A cache memory is a level of the memory hierarchy that is closer to the CPU than DRAM

Cache Organization (Data Storage)

A *cache block* contains multiple words

Example: a cache with four-word blocks



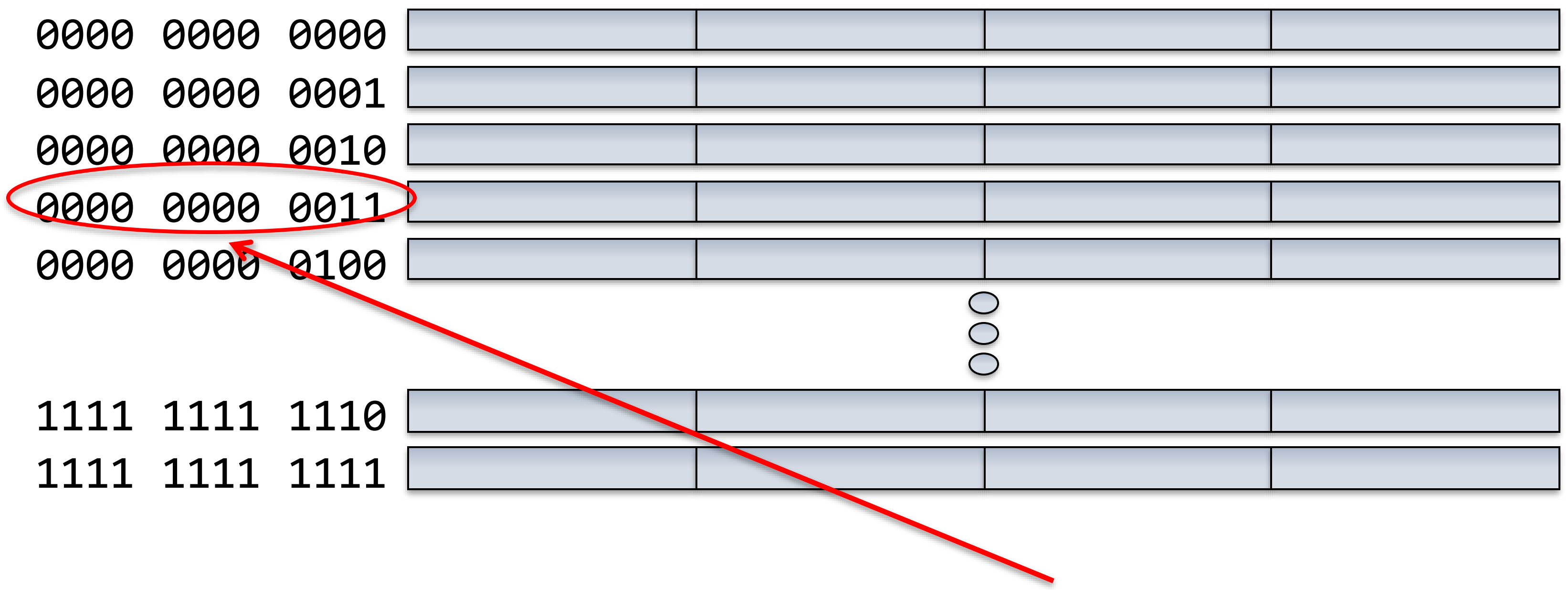
Cache Organization (Data Storage)

Each cache has multiple such blocks

Example: a cache with 4K (4K = 4096) four-word blocks:

We want to give a unique identifier to each block

How many bits do we need for each identifier?

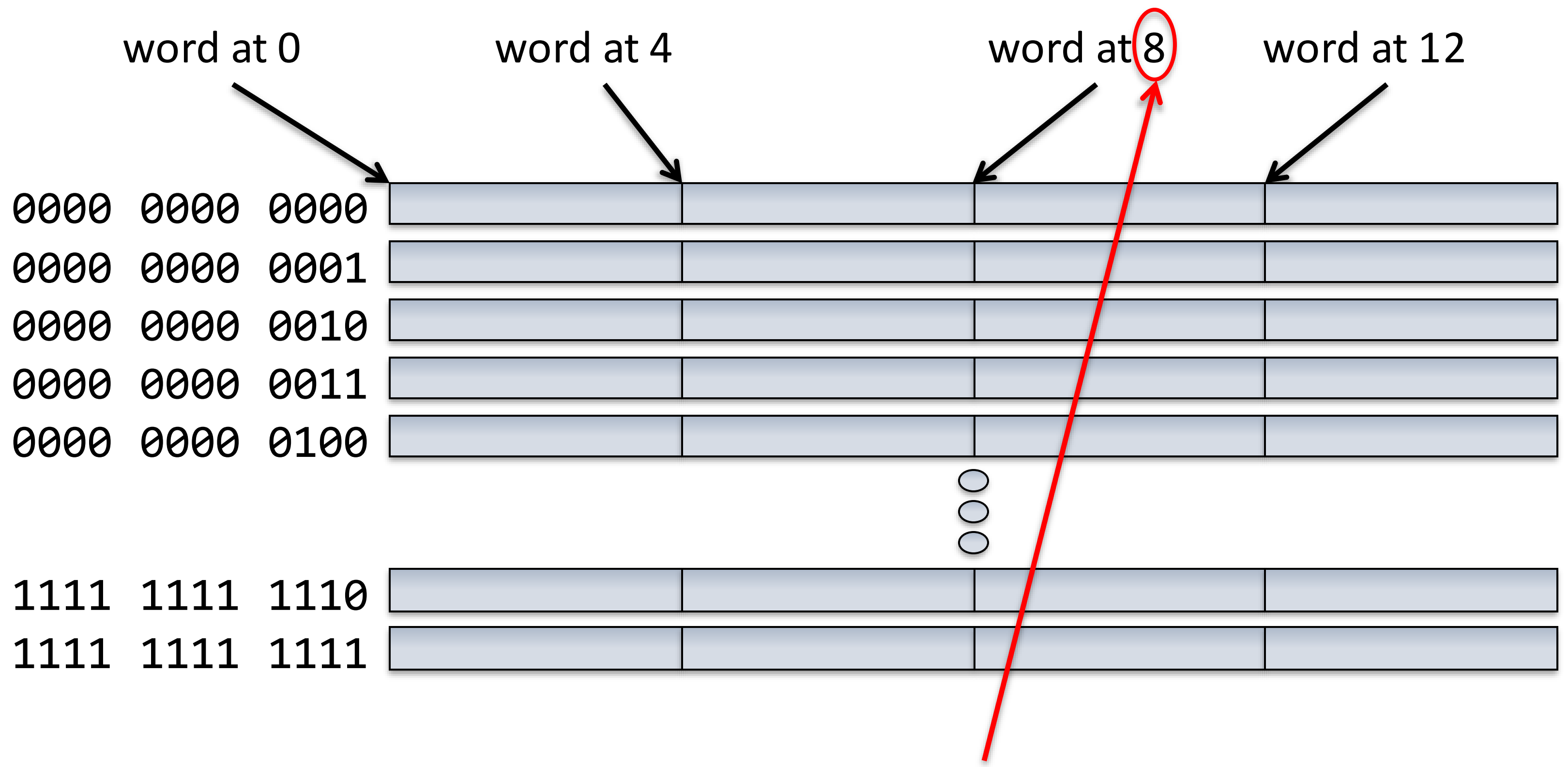


The unique ID of each cache entry is the *index* of the entry

Offset

How do we find a word inside a block?

We measure the distance, in bytes, from the beginning of the block

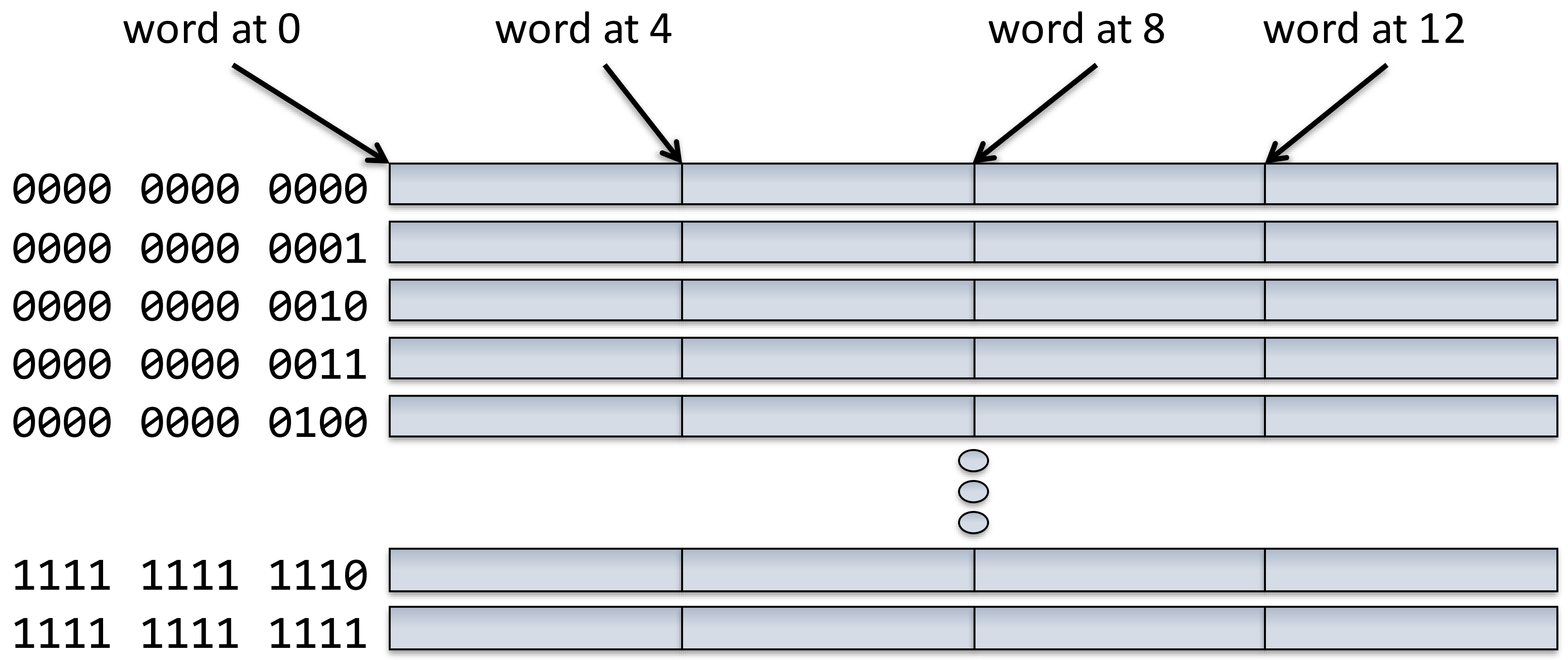


The position of a word within a block is the word ***offset***

Offset

How do we find a word inside a block?

We measure the distance, in bytes, from the beginning of the block



In the example, how many bits do I need for the offset?

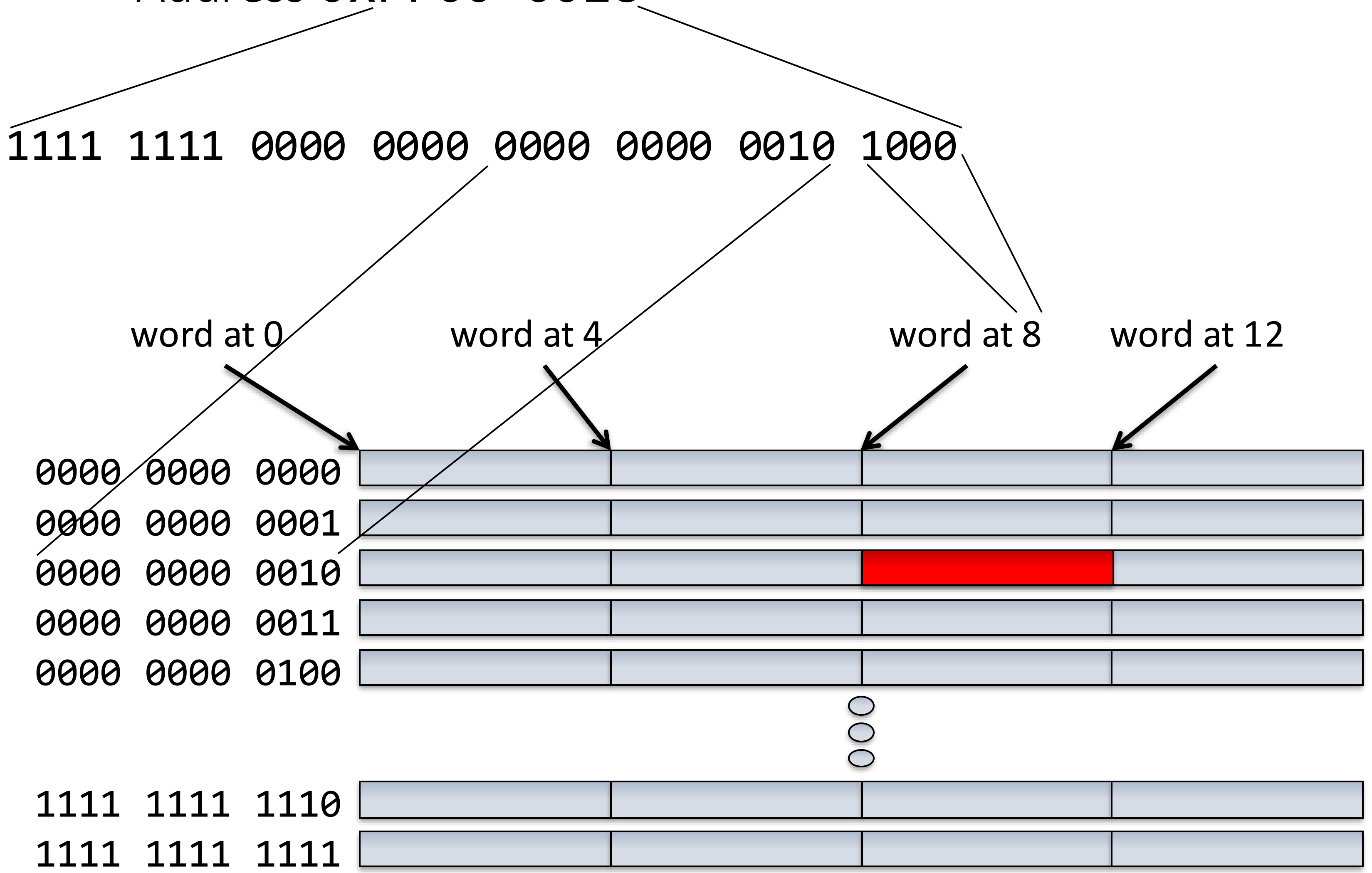
Each block has 16 bytes ⇒ need 4 bits.

Finding a word in the cache

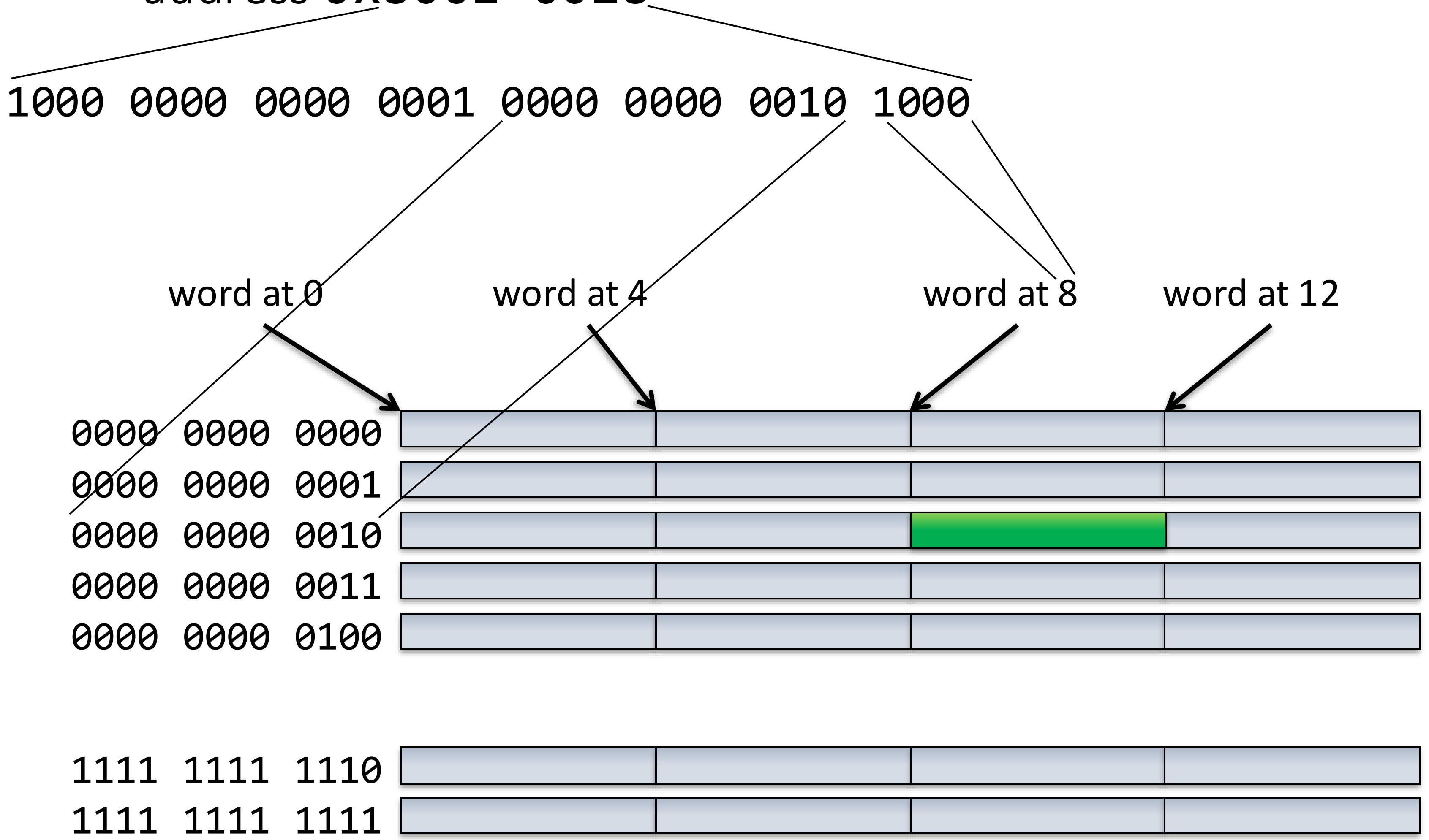
Example:

Processor requests word at address `0xFF00 0028`

Address `0xFF00 0028`

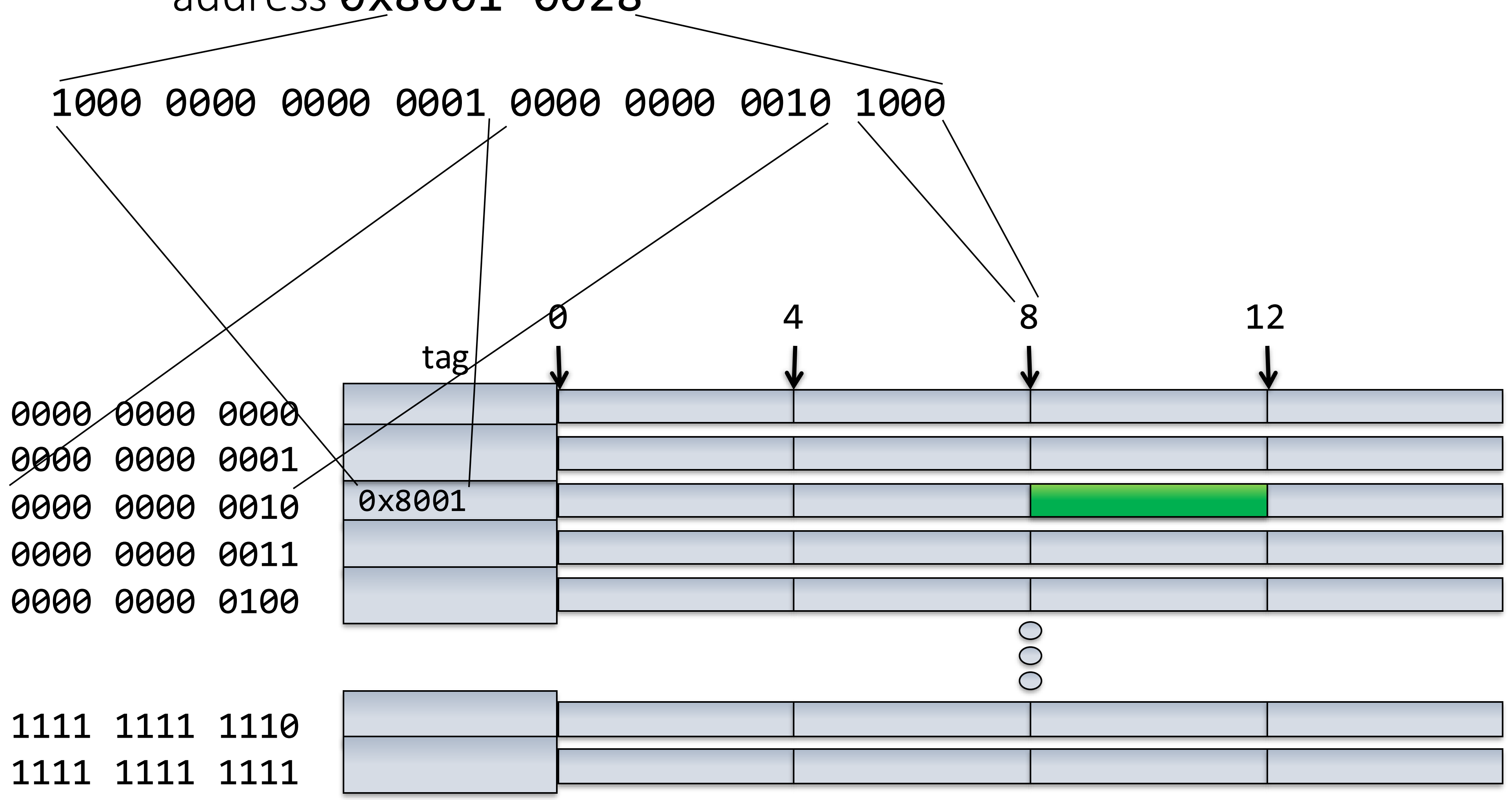


Now processor accesses address `0x8001 0028`



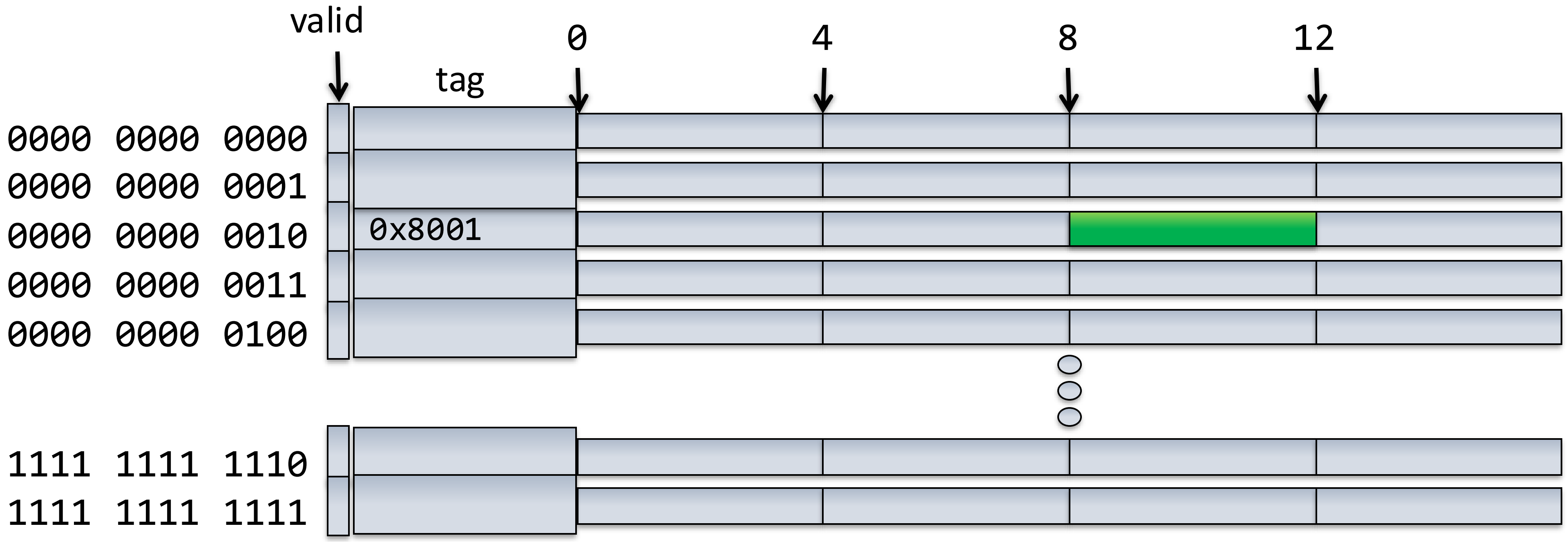
How do we know if the word in the cache is from `0xFF00 0028` or from `0x8001 8028`?

When the processor accesses address `0x8001 0028`



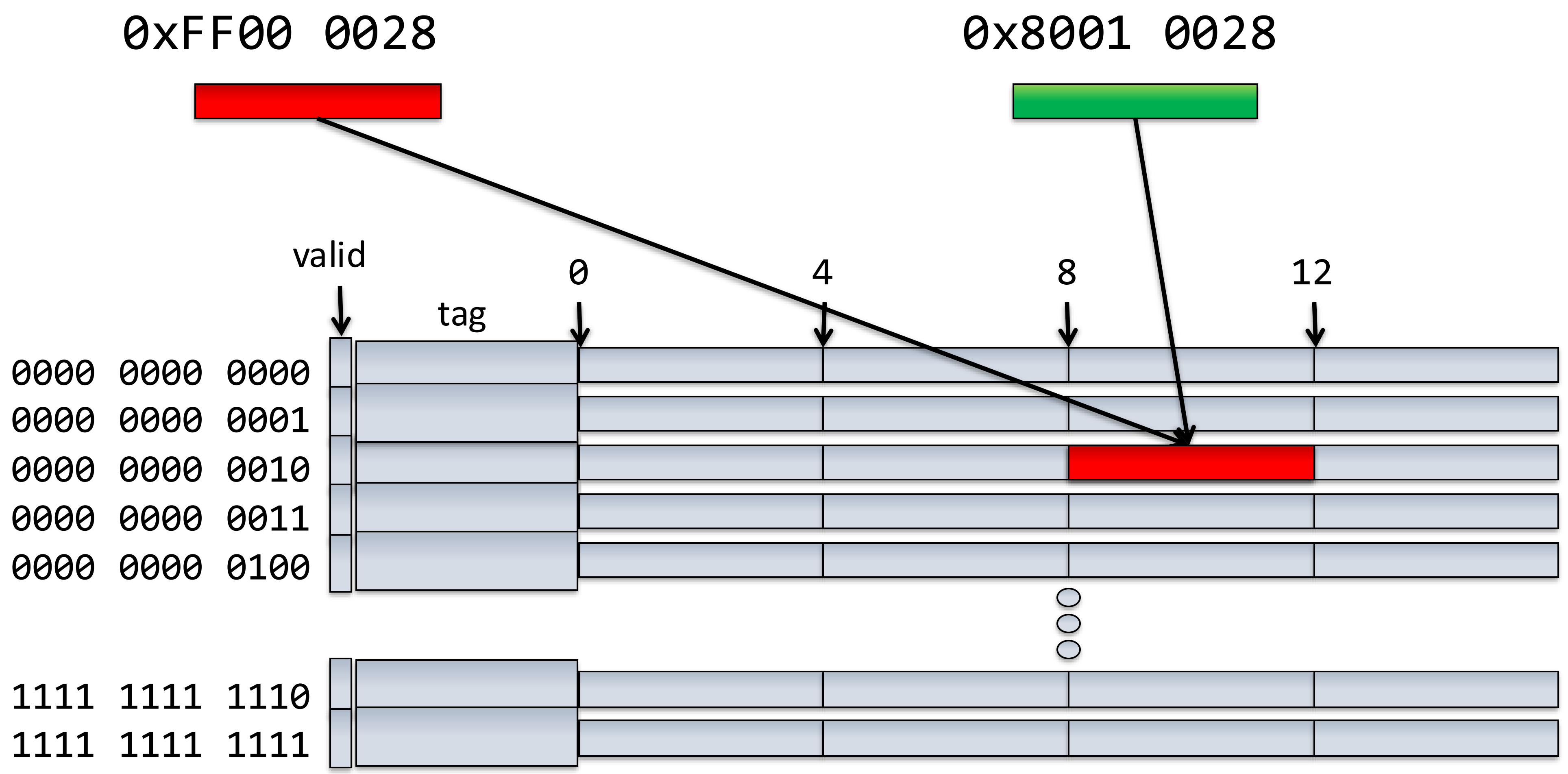
The *portion of the address* stored along with each cache block to identify which address in the memory the block came from is called **tag**

How do we know that the cache has a valid block for the requested address?



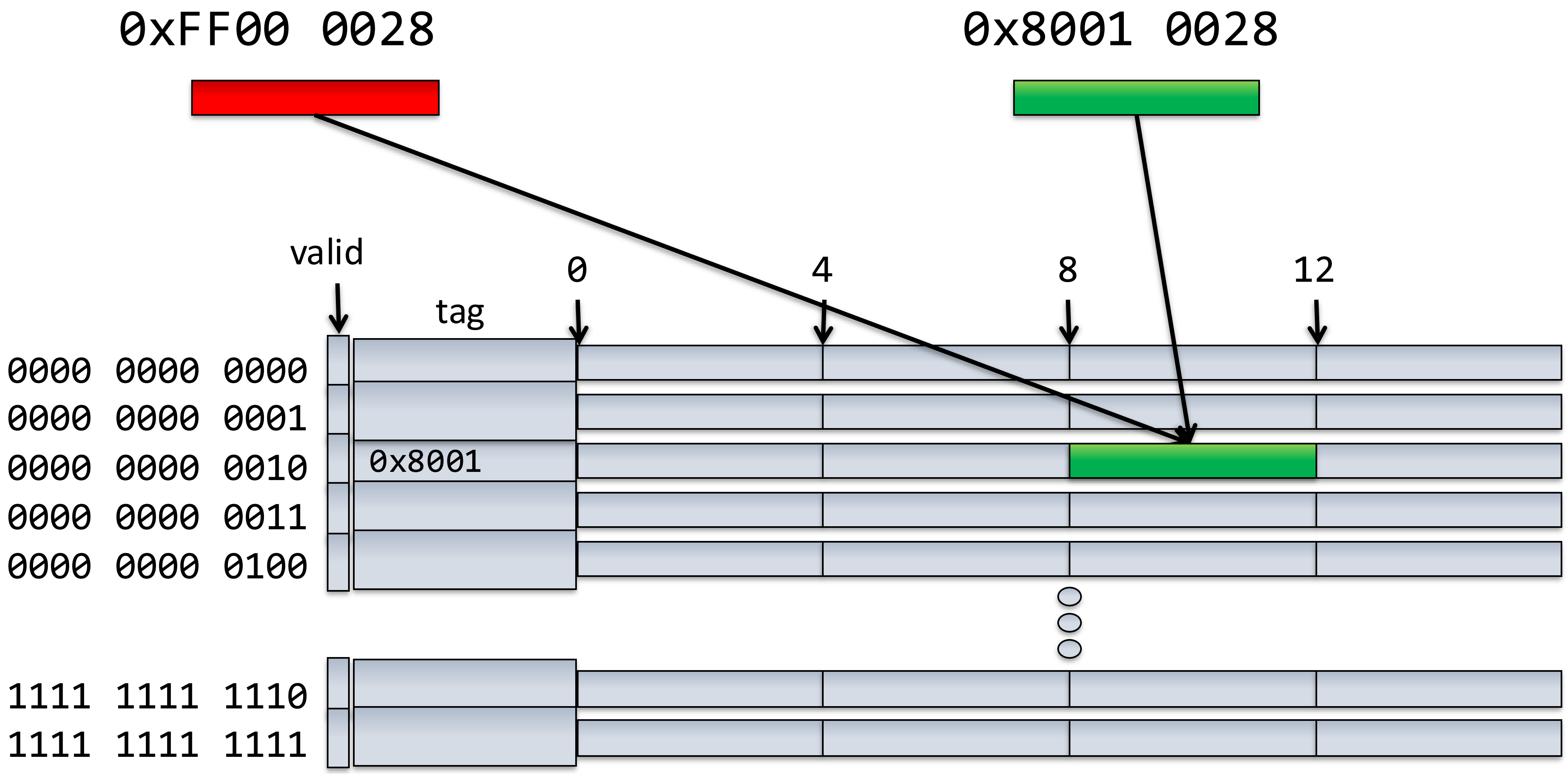
A **valid bit** associated with each block indicates if the content in the block is valid

In this design, there is only one place in the cache for each memory address



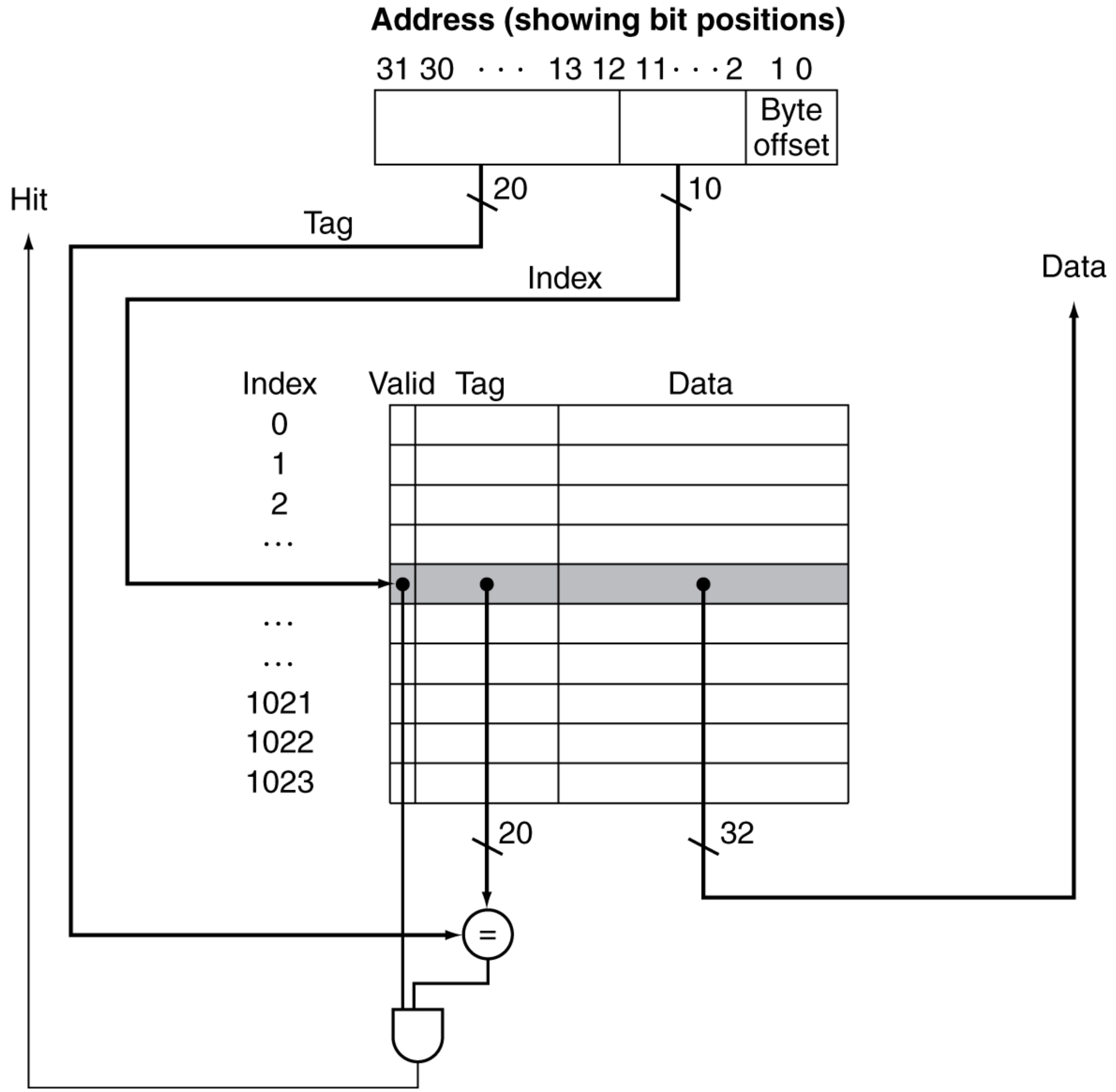
This mapping between memory addresses and cache blocks is called a **direct mapping**

In this design, there is only one place in the cache for each memory address



The problem with this mapping is that there are often **conflicts** into the cache for competing memory addresses

Address Subdivision



We want to change the mapping to allow a given memory address to be mapped to more than one block in the cache

Let’s group the cache blocks into sets

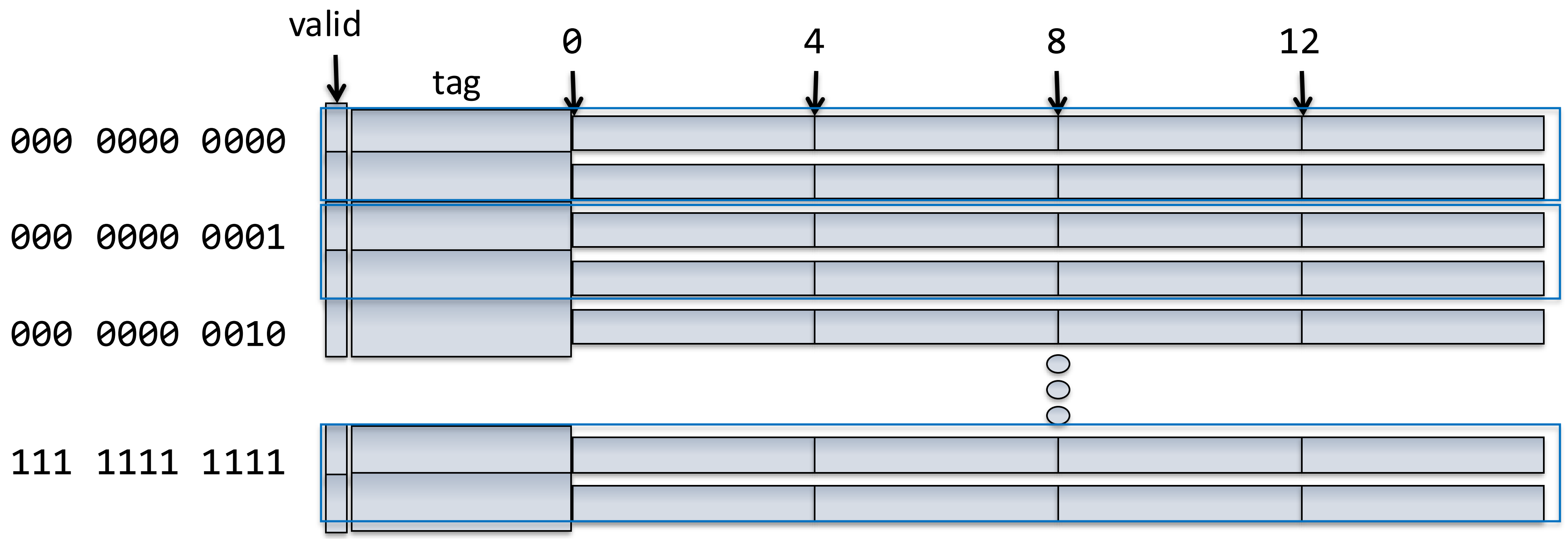
Example: lets put two blocks in each set

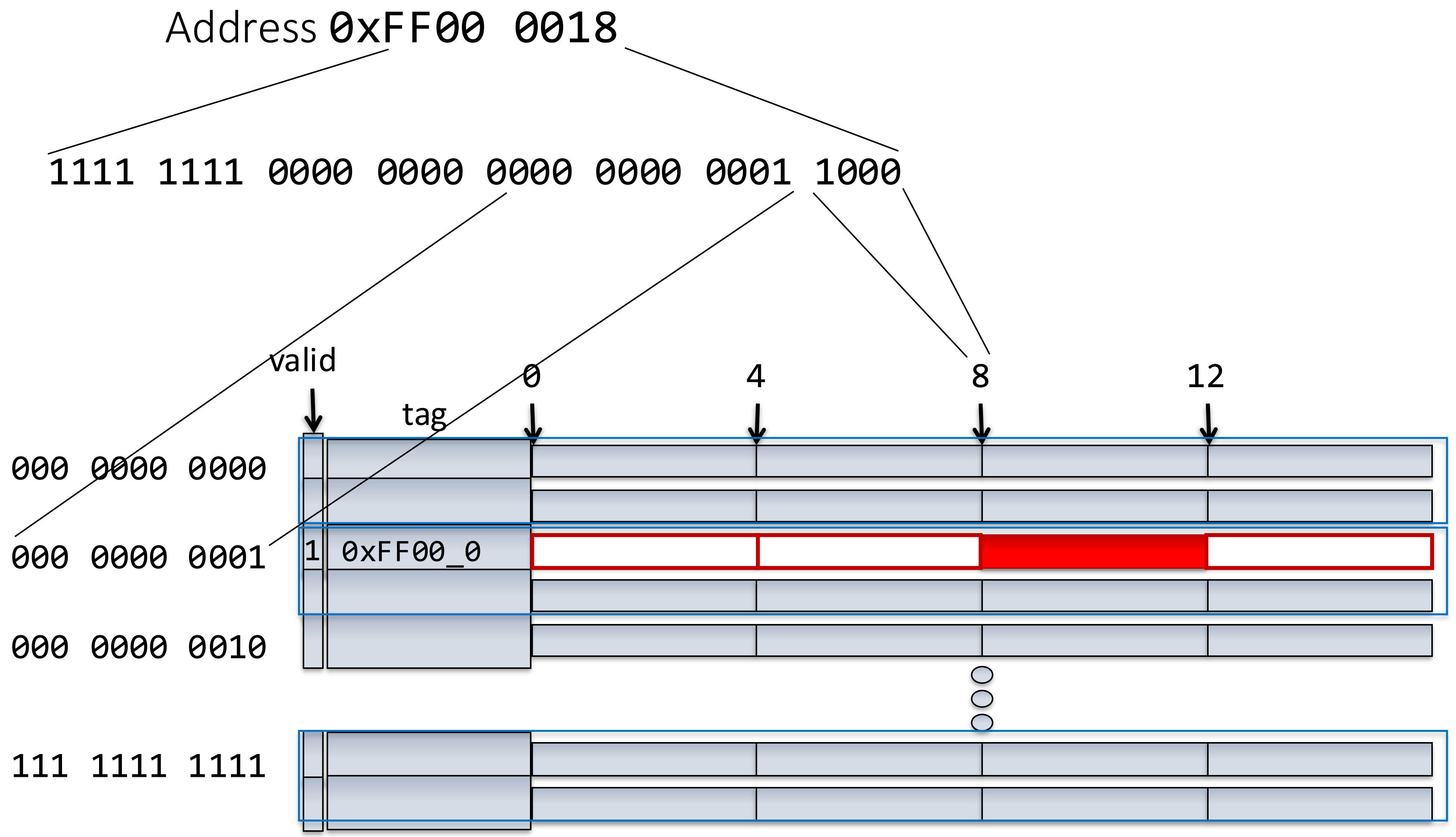
The cache stores 4K blocks

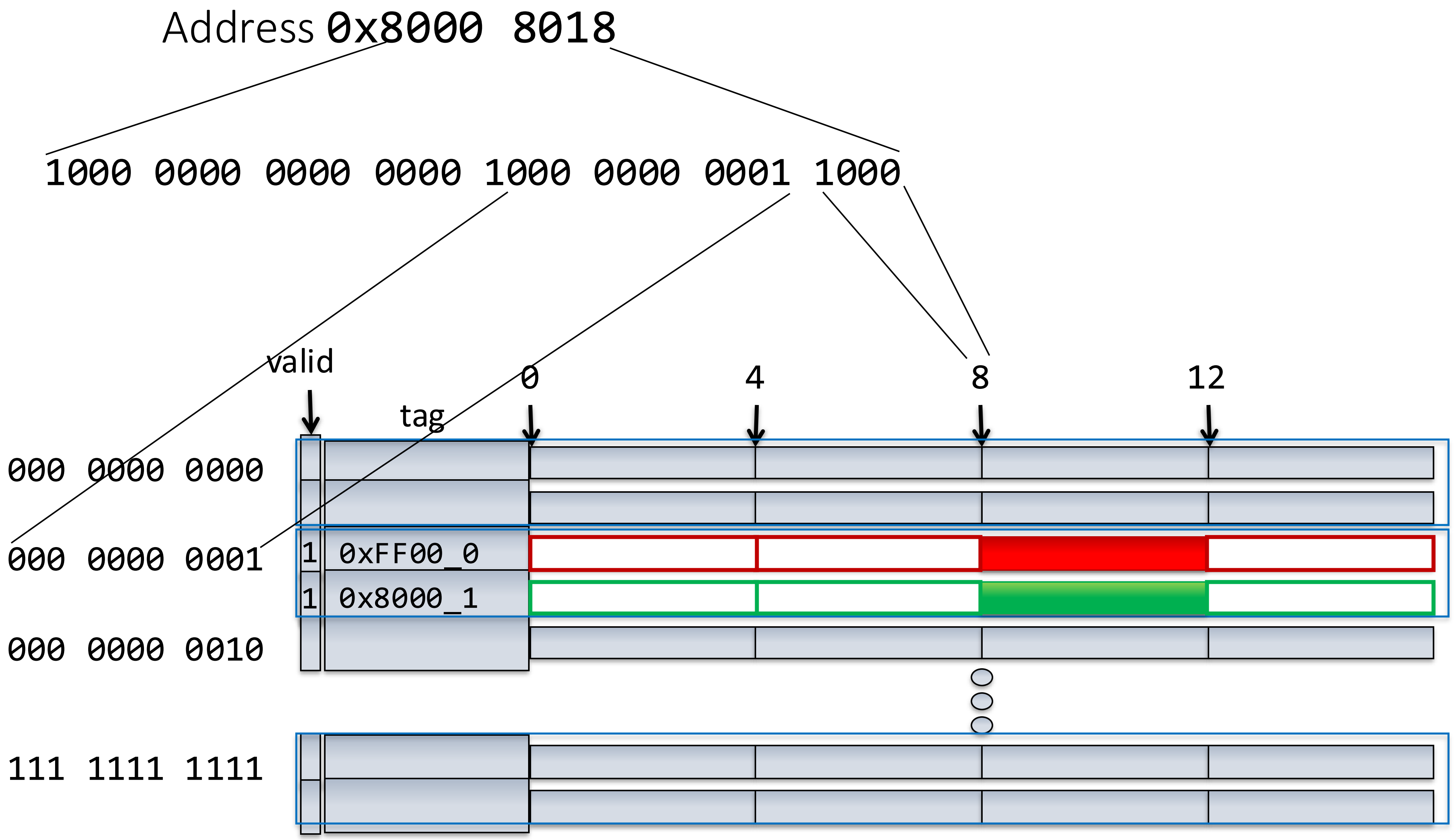
How many sets are in the cache? 2K sets

We want to give a unique ID for each set

How many bits do we need for each set ID? 11 bits

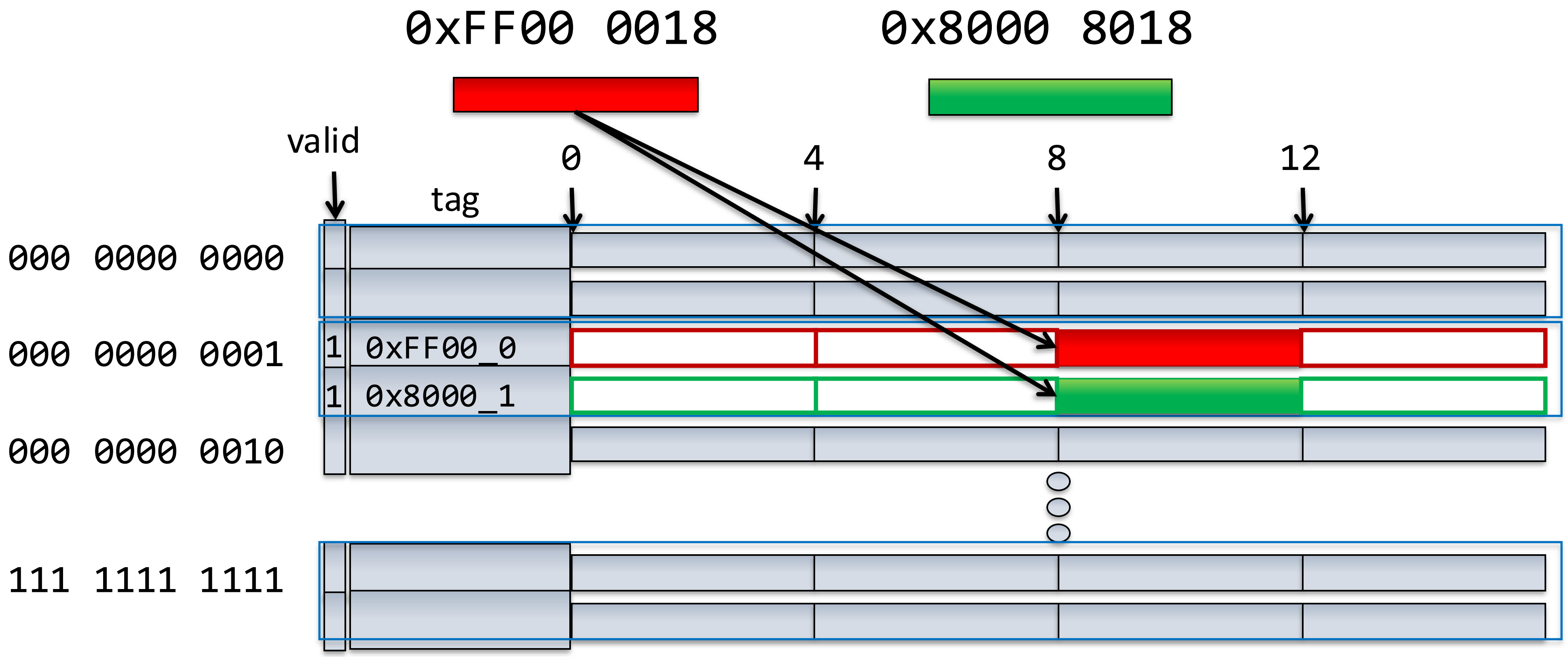



Address 0xFF00 0018
1111 1111 0000 0000 0000 0000 0001 1000
valid
tag
0
4
8
12
000 0000 0000
000 0000 0001
1
0xFF00_0
000 0000 0010
111 1111 1111


Address 0x8000 8018
1000 0000 0000 0000 1000 0000 0001 1000
valid
tag
0
4
8
12
000 0000 0000
000 0000 0001
000 0000 0010
111 1111 1111
1
0xFF00_0
1
0x8000_1

Now both memory locations can be in the cache at the same time

Each location can be in a set in two ways

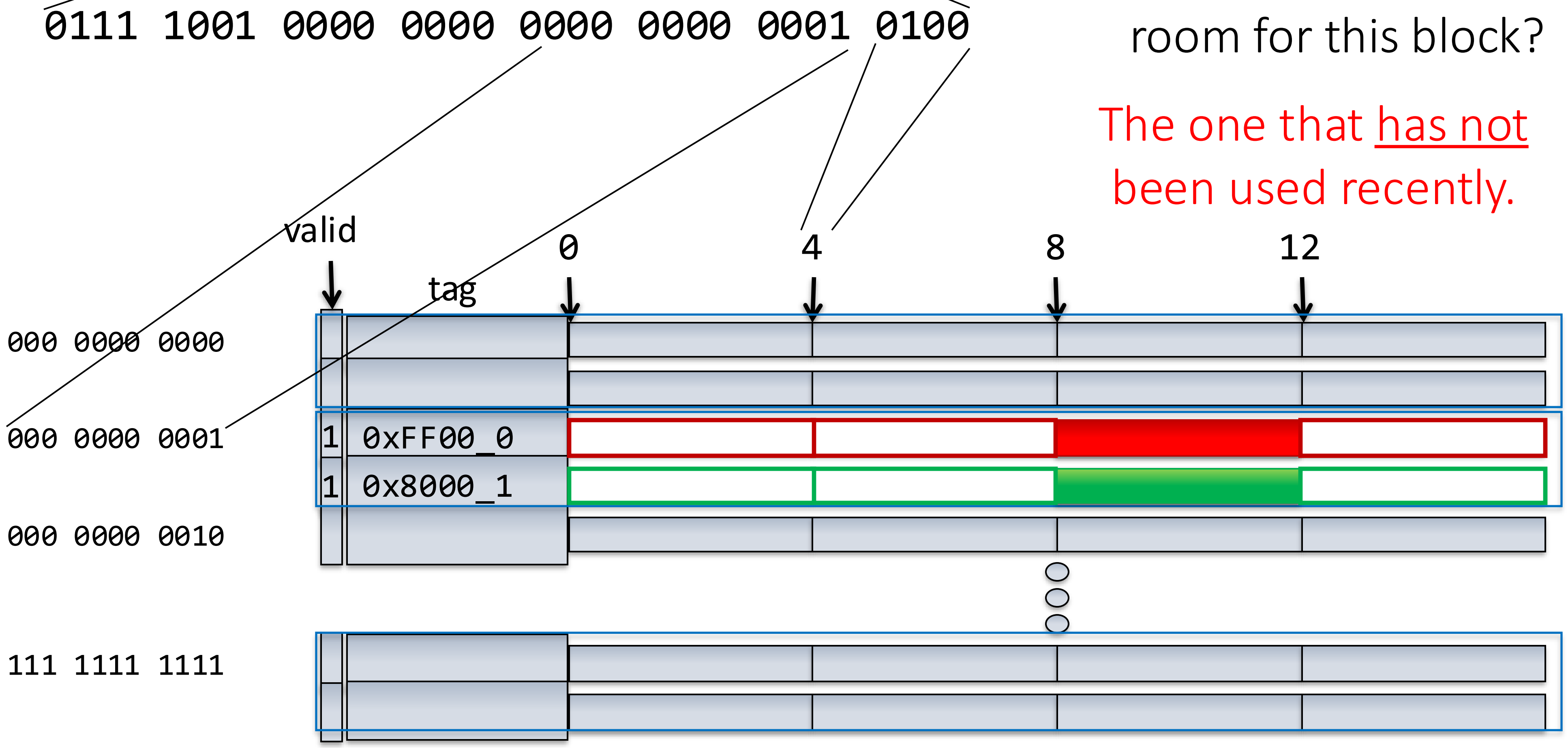


This cache is two-way set associative

Address 0x7900 0014

Which block should be removed from the cache to make room for this block?

The one that has not been used recently.

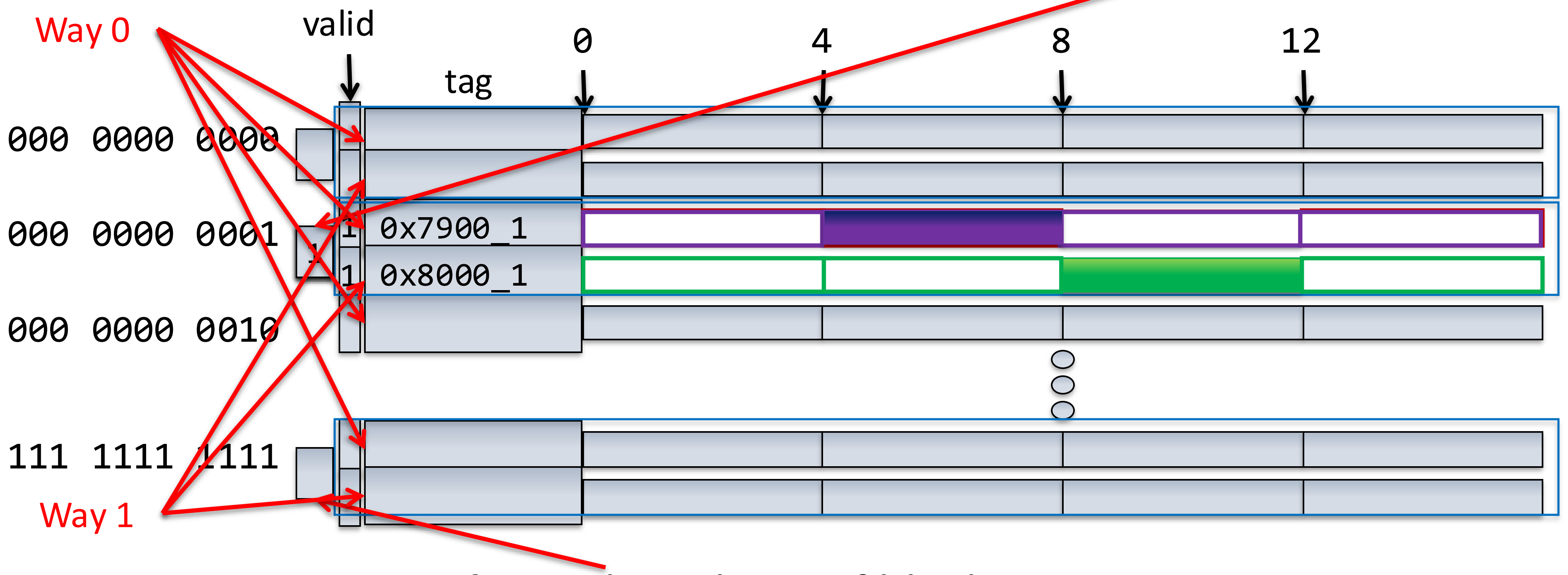


Tag in address does not match either tag in the set

⇒ This access is a cache miss

Least-Recently Used (LRU) bit

Processor reads	0xFF00 0018	🟥
Processor reads	0x8000 8018	🟩
Processor reads	0x7900 0014	🟪



We associate an **LRU bit** with each set of blocks

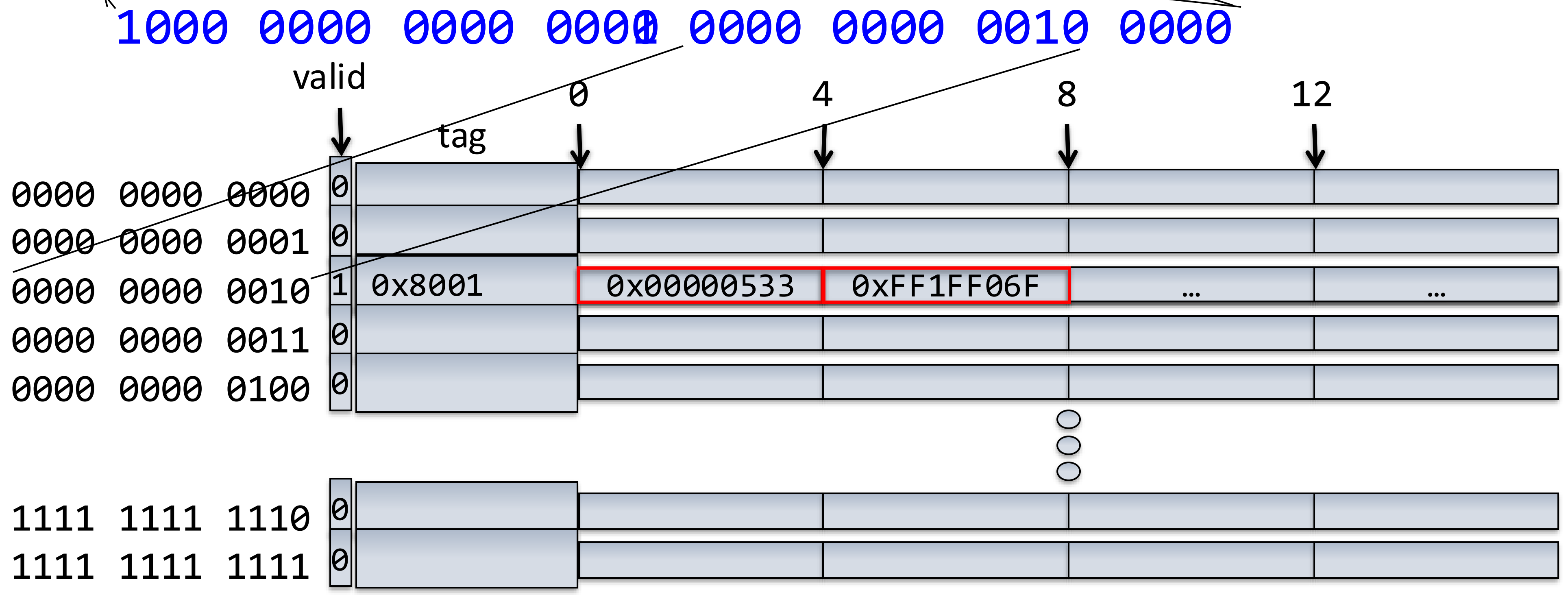
The LRU bit indicates which block **has not** been used recently

Associativity and Performance:

an example

Direct-mapped Cache

```
0x8000 0020  loop:   add   t1, t2, t3
0x8000 0024          beq   t1, zero, false
0x8000 0028          jal   zero, true
…
0x8001 0020  false:  mv    a0, zero
0x8001 0024  true:   jal   zero, loop
```



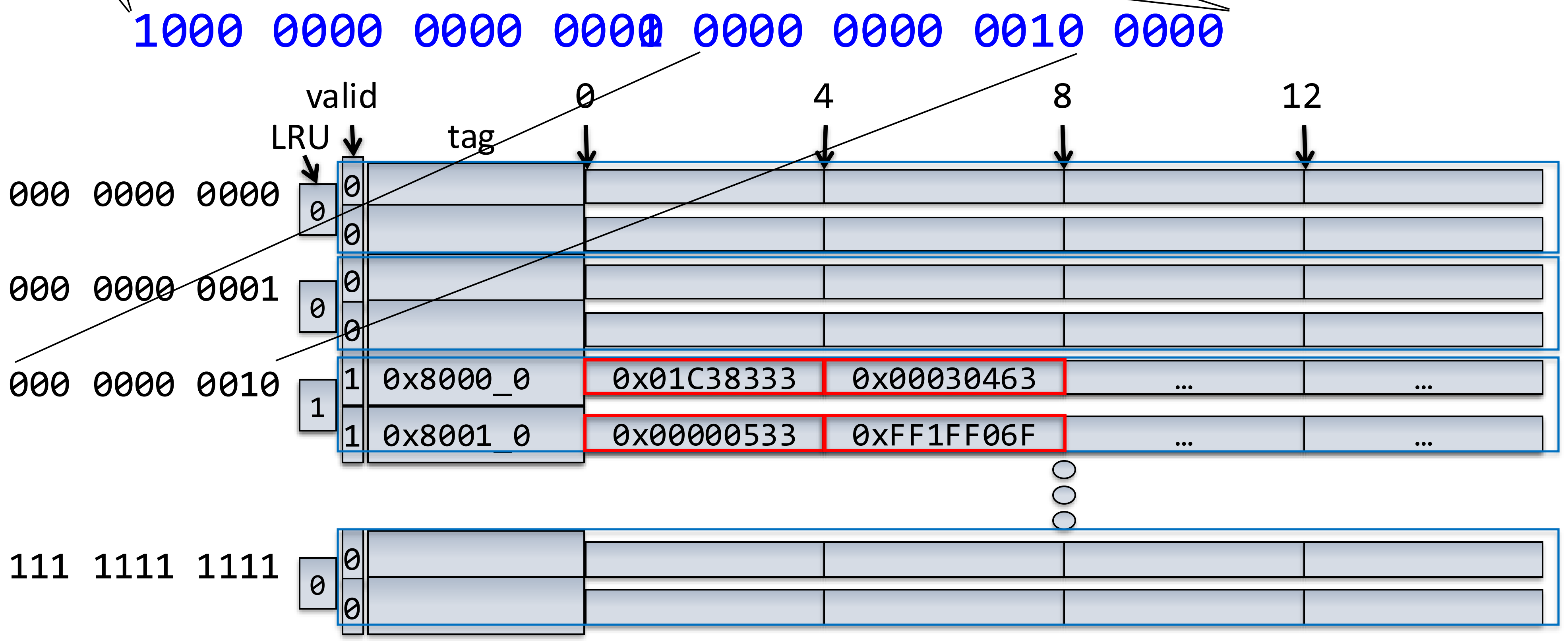
valid = 0 ⇒ cache miss

valid = 1 and tags match ⇒ cache hit

valid = 1 and tags do not match ⇒ cache miss

Two-way Set Associative Cache

```
0x8000 0020  loop:    add   t1, t2, t3
0x8000 0024           beq   t1, zero, false
0x8000 0028           jal   zero, true
…
0x8001 0020  false:   mv    a0, zero
0x8001 0024  true:    jal   zero, loop
```


1000 0000 0000 0001 0000 0000 0010 0000
valid
LRU
tag
0
4
8
12
000 0000 0000
0
0
0
000 0000 0001
0
0
0
000 0000 0010
1
1
0x8000_0
0x01C38333
0x00030463
…
…
1
0x8001_0
0x00000533
0xFF1FF06F
…
…
111 1111 1111
0
0
0