COMP1001

Computer Systems

Dr. Vasilios Kelefouras

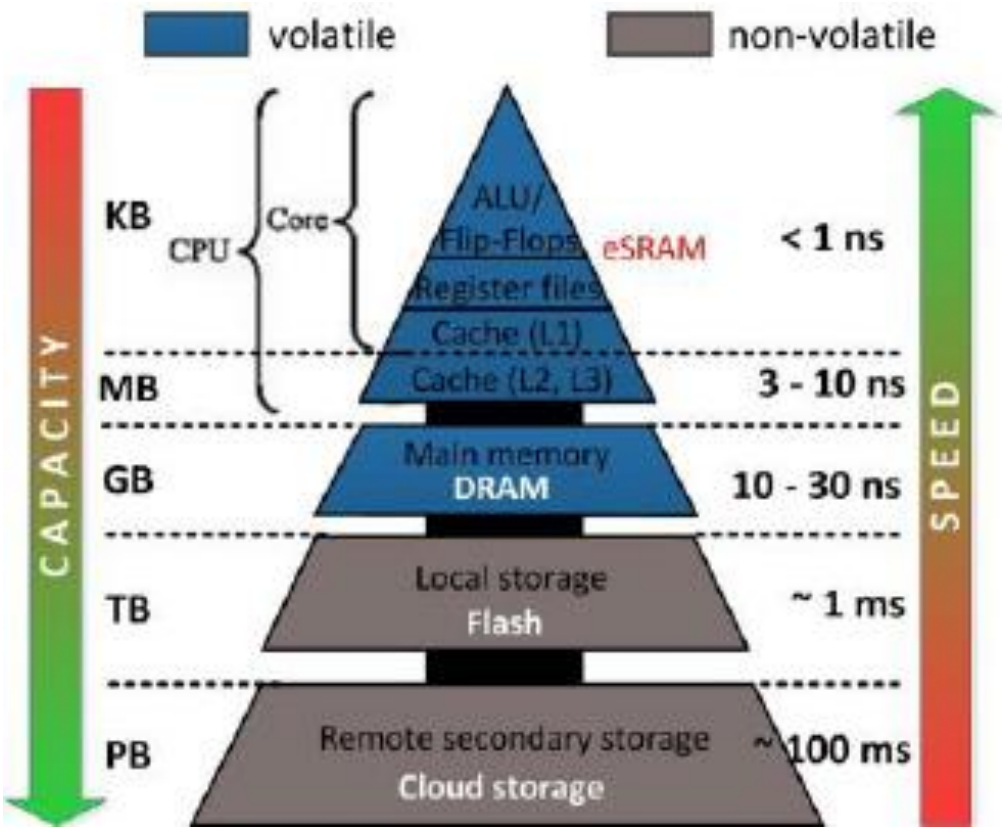
Email: v.kelefouras@plymouth.ac.uk
Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

School of Computing
(University of Plymouth)

Outline

- Memory hierarchy
- Cache memories
- Cache design
  - Cache hit/miss
  - Direct mapped, set associative, fully associative
  - Replacement policy

Memory Hierarchy (1)



Taken from *https://www.researchgate.net/publication/281805561_MTJ-based_hybrid_storage_cells_for_normally-off_and_instant-on_computing/figures?lo=1*

Cache memories

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- The solution is to add from 1 up to 3 levels of cache memories, which are small, fast, but expensive memories
  - The cache goes between the processor and the slower, main memory (DDR)
  - It keeps a copy of the most frequently used data from the main memory
  - Faster reads and writes to the most frequently used addresses
  - We only need to access the slower main memory for less frequently used data
- Cache memories occupy the largest part of the chip area
- They consume a significant amount of the total power consumption
- Add complexity to the design
- Cache memories are of key importance regarding performance

Memory Hierarchy (2)

- Consider that CPU needs to perform a load instruction
  - First it looks at L1 data cache. If the datum is there then it loads it and no other memory is accessed (**L1 hit**)
  - If the datum is not in the L1 data cache **(L1 miss**), then the CPU looks at the L2 cache
  - If the datum is in L2 (**L2 hit**) then no other memory is accessed.

  Otherwise (**L2 miss**), the CPU looks at main memory

L1 cache access time:	1-4 CPU cycles
L2 cache access time :	6-14 CPU cycles
L3 cache access time :	40-70 CPU cycles
DDR access time :	100-200 CPU cycles

Definitions: Hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for. Hits are desirable, because the cache can return the data much faster than main memory

- A **cache miss** occurs if the cache does not contain the requested data. This is inefficient, since the CPU must then wait accessing the slower next level of memory

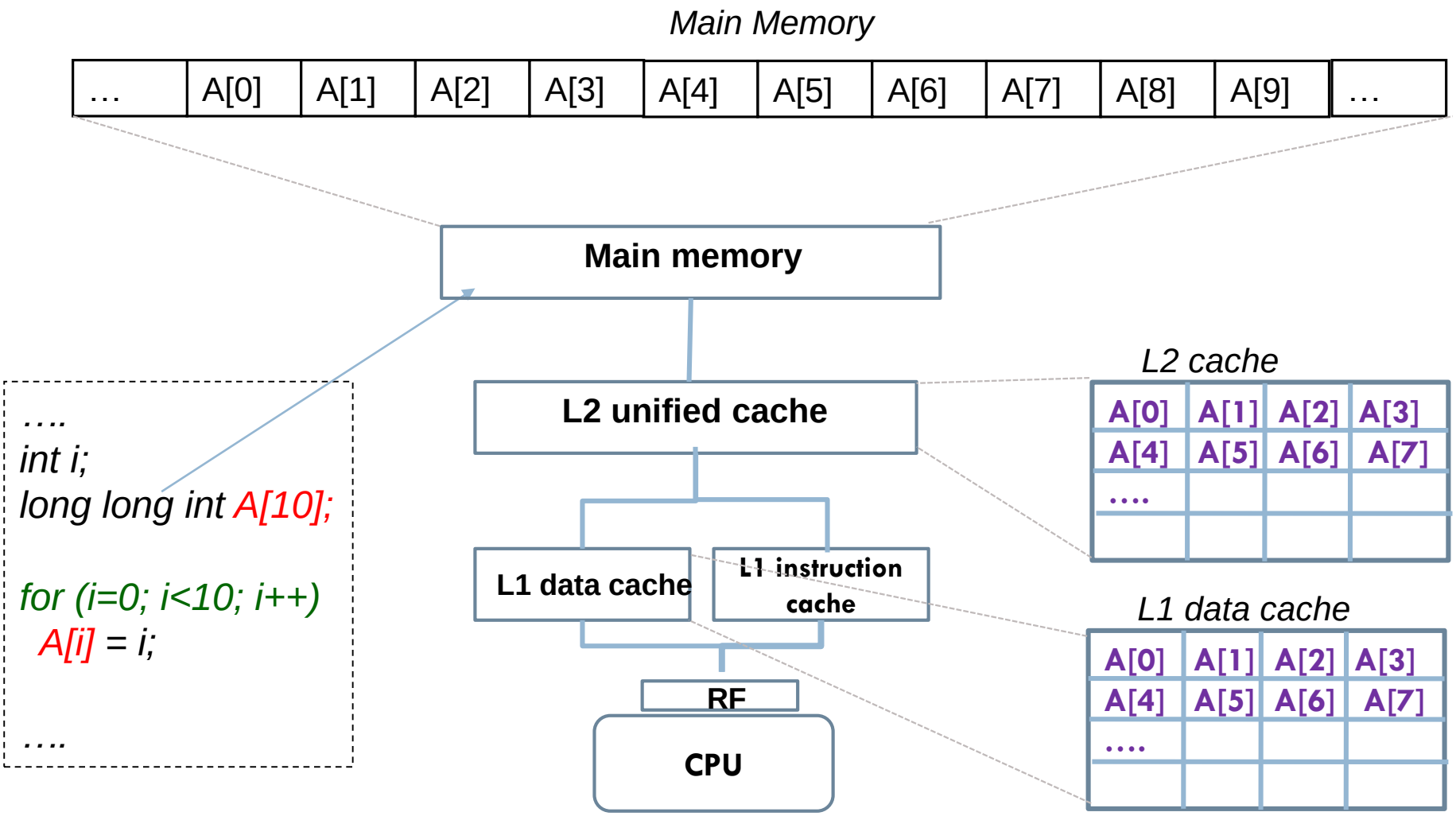
Why should I care about how memory hierarchy works?

- Here is an example why …
  - The first code can be up to x16 times faster than the second on a common PC.
    - But why?
  - Many such examples exist
  - We cannot write performance efficient code without knowing how the hardware works

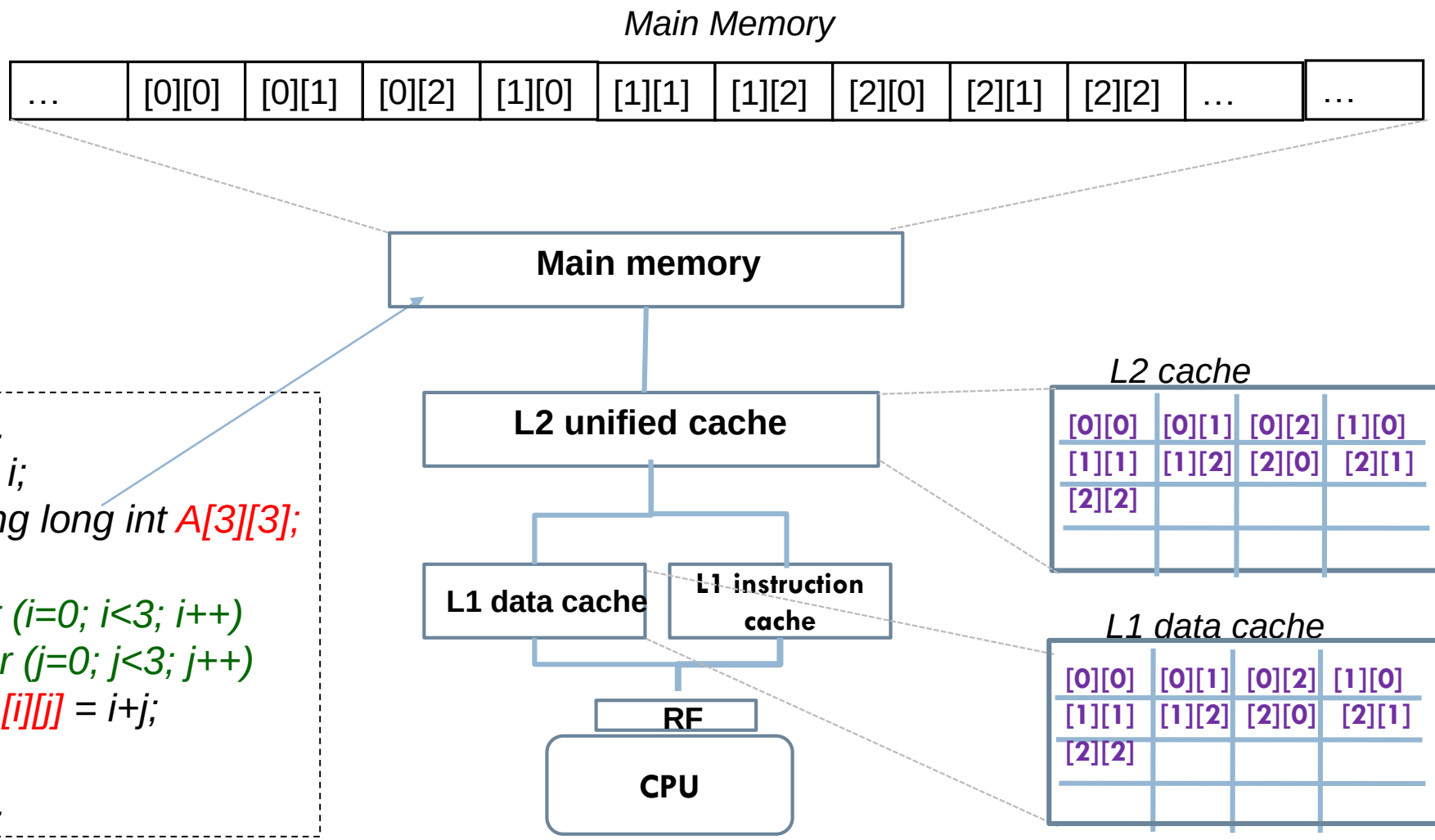
```
for (i=0; i<1024; i++)
 for (j=0; j<1024; j++)
  A[i][j] = …;
```

```
for (j=0; j<1024; j++)
 for (i=0; i<1024; i++)
  A[i][j] = …;
```

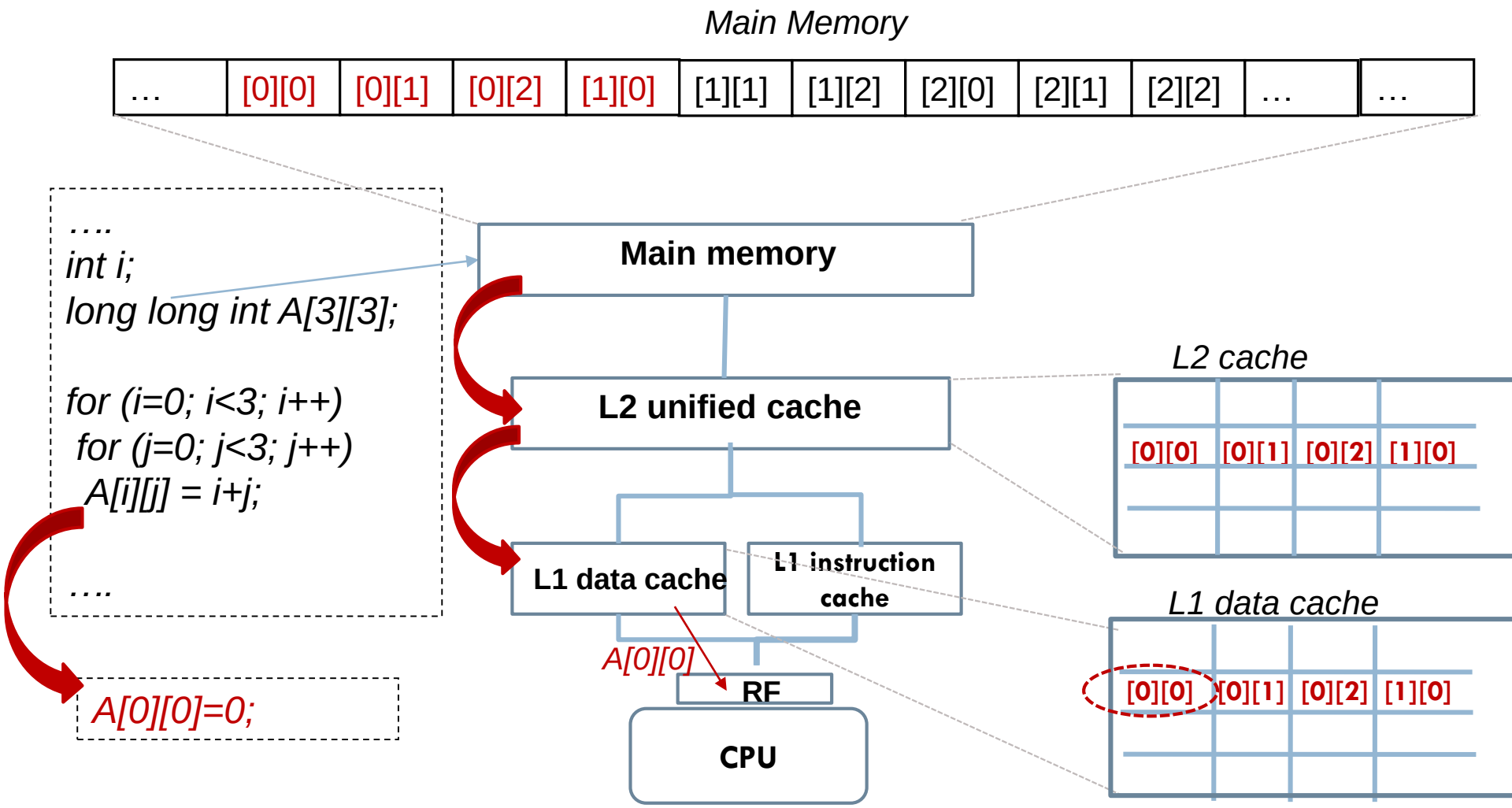
Arrays - memory allocation for 1d arrays – From a Hardware Perspective



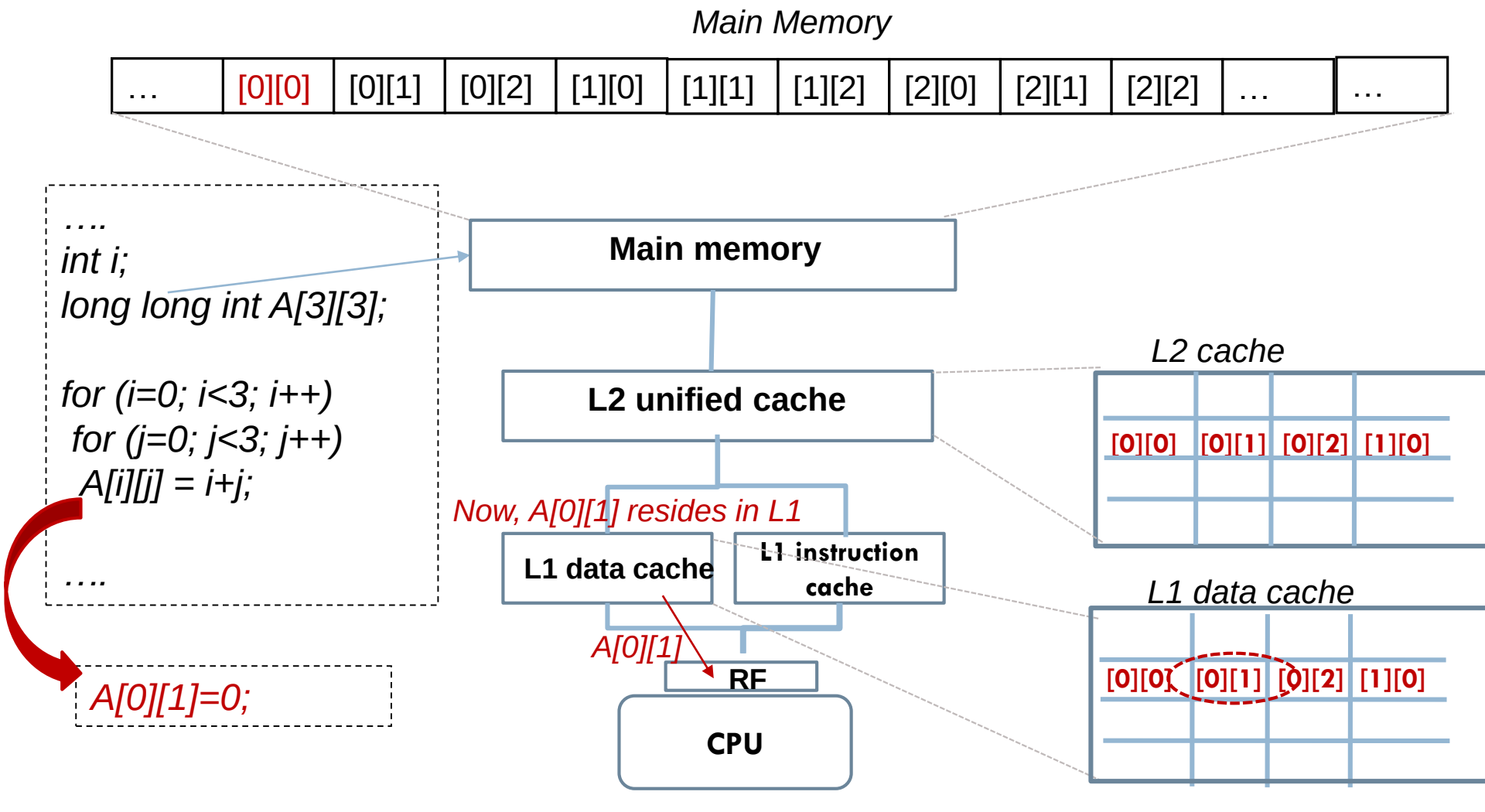
Arrays - memory allocation for 2d arrays – From a Hardware Perspective (1)



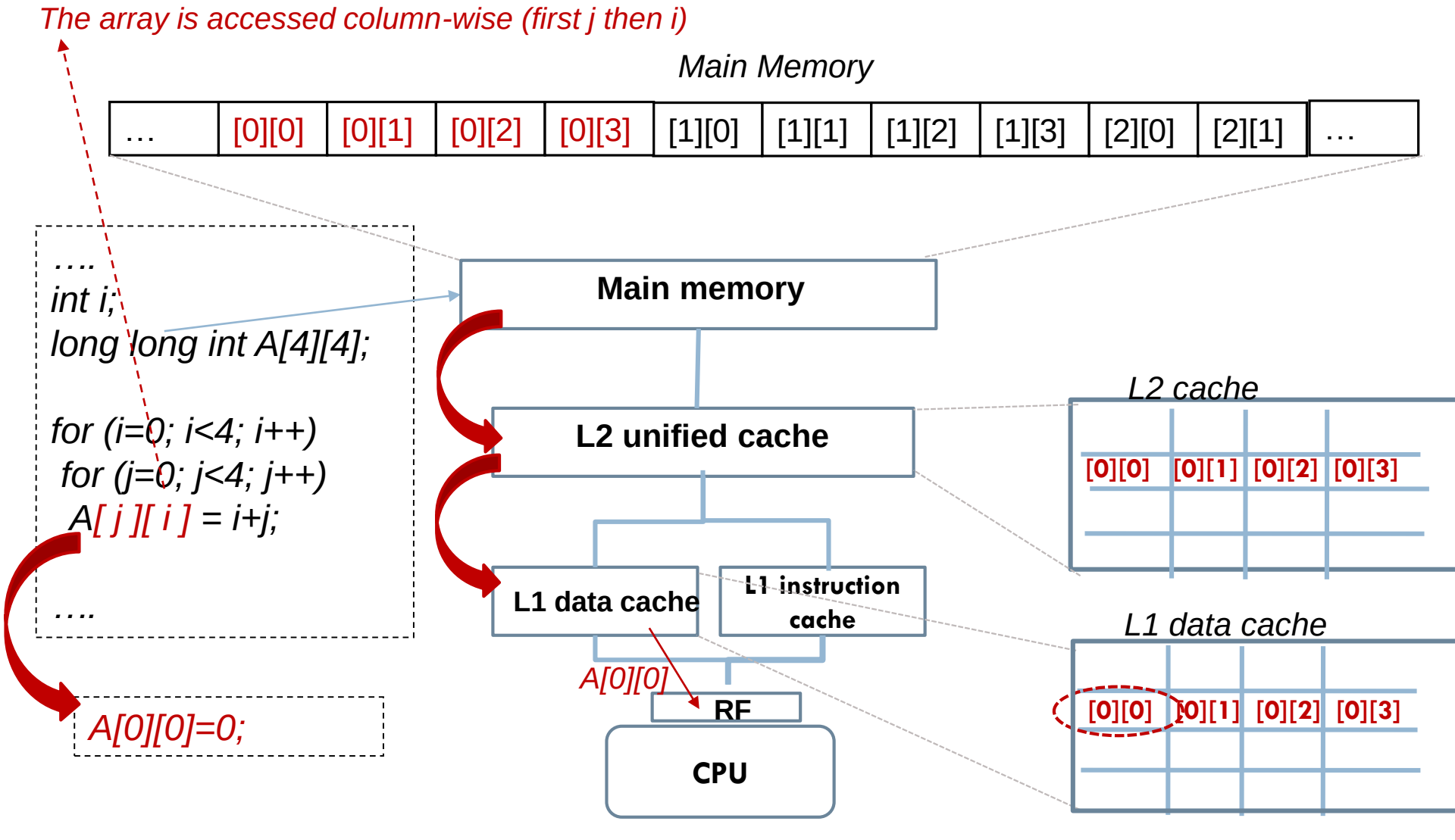
Arrays - memory allocation for 2d arrays – From a Hardware Perspective (2)



Arrays - memory allocation for 2d arrays – From a Hardware Perspective (3)

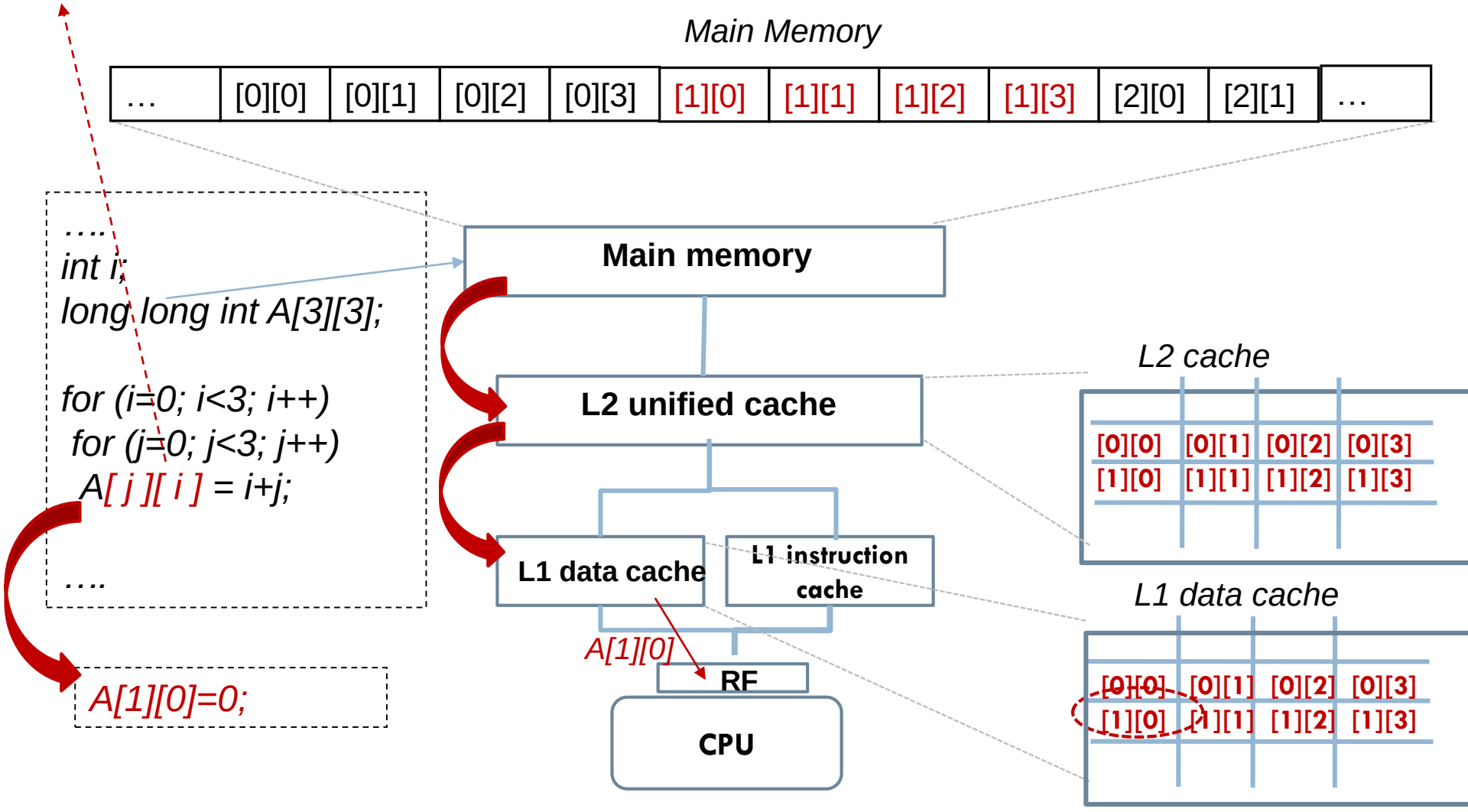


Accessing arrays – the wrong way (2) column-wise



Accessing arrays – the wrong way (3) column-wise

The array is accessed column-wise (first j then i)



Activity

- Create a VS project and measure the execution time of the following loop kernels

```
for (i=0; i<1024; i++)
 for (j=0; j<1024; j++)
  A[i][j] = …;
```

```
for (j=0; j<1024; j++)
 for (i=0; i<1024; i++)
  A[i][j] = …;
```

How important is cache size?

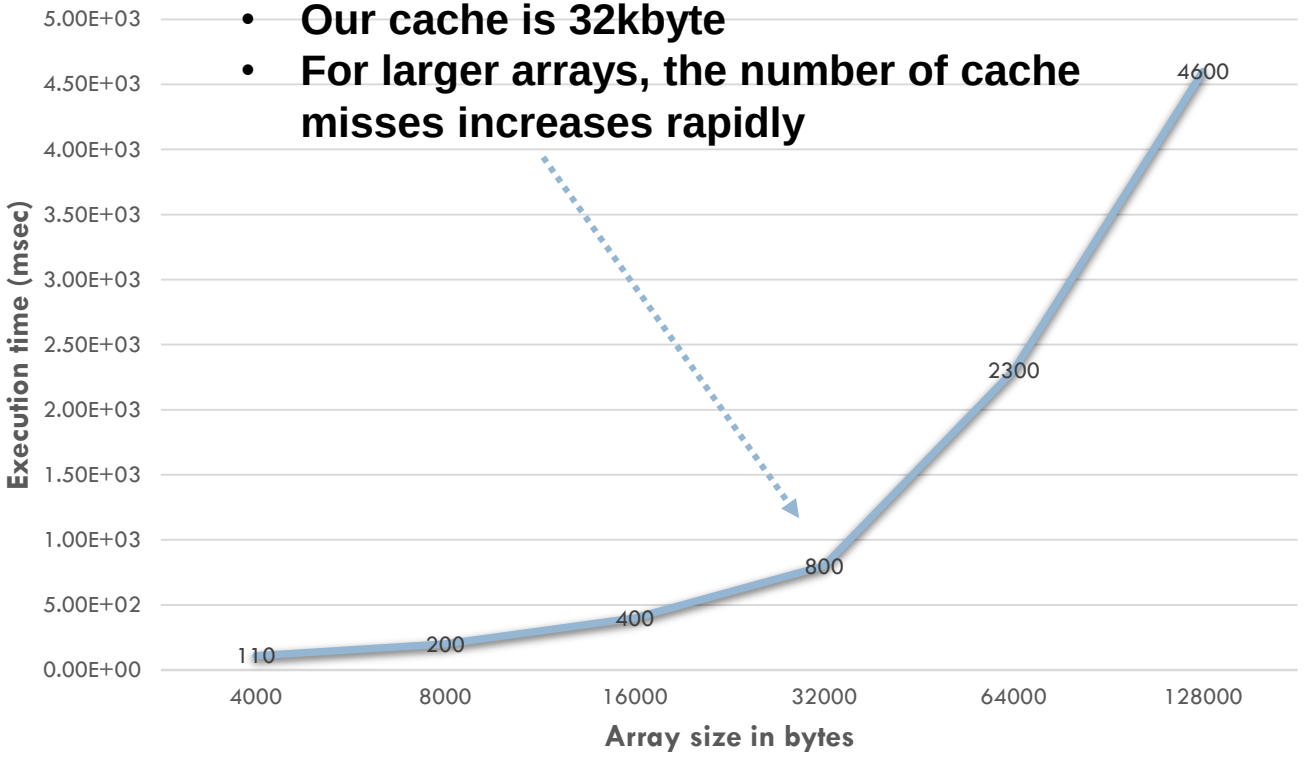
The following code can be seen as a benchmark that experimentally finds the cache size

```
#define N  1000
Int X[N];


for (i=0; i<1000000; i++)
 for (j=0; j<N; j++){
 X[j]=i;
}
```

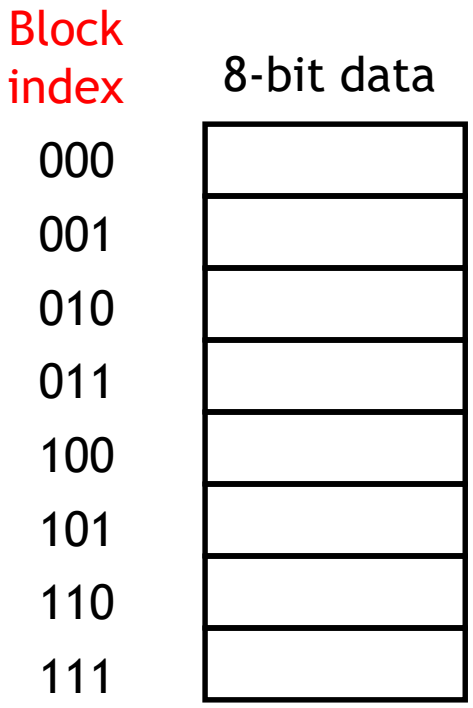
// N=1000, 2000, 4000, 8000, 16000, 32000

- **Our cache is 32kbyte**
- **For larger arrays, the number of cache misses increases rapidly**



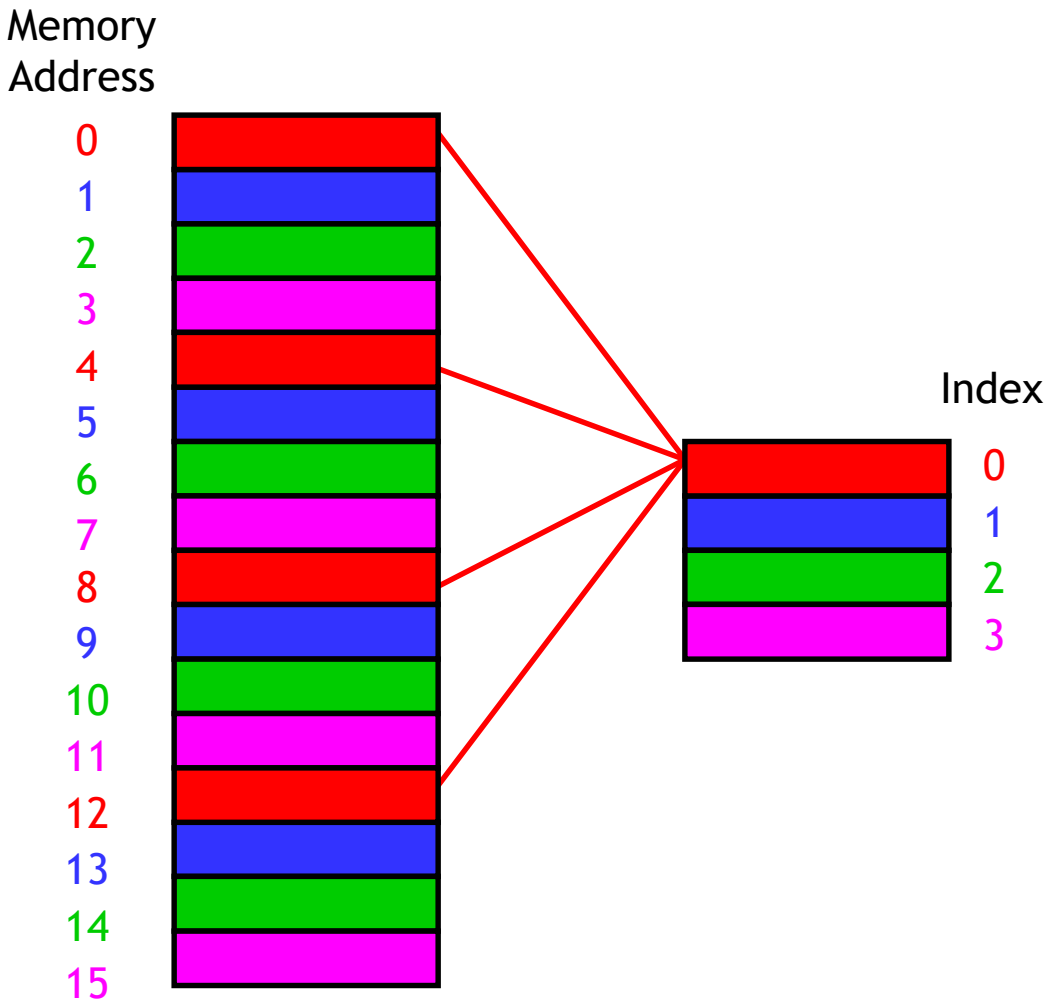
A simple cache design

- Caches are divided into blocks, which may be of various sizes
  - The number of blocks in cache memories are always in power of 2
  - For now consider that each block contains just one byte (not true in practice). Of course this cannot take advantage of spatial locality
- Here is an example of cache with eight blocks, each holding one byte



Where should we put data in the cache? (1)

- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block
- In the following figure a 16-entry main memory and a 4-entry cache (four 1-entry blocks) are shown
- Memory locations 0, 4, 8 and 12 all map to cache block 0
- Addresses 1, 5, 9 and 13 map to cache block 1, etc



Where should we put data in the cache? (2)

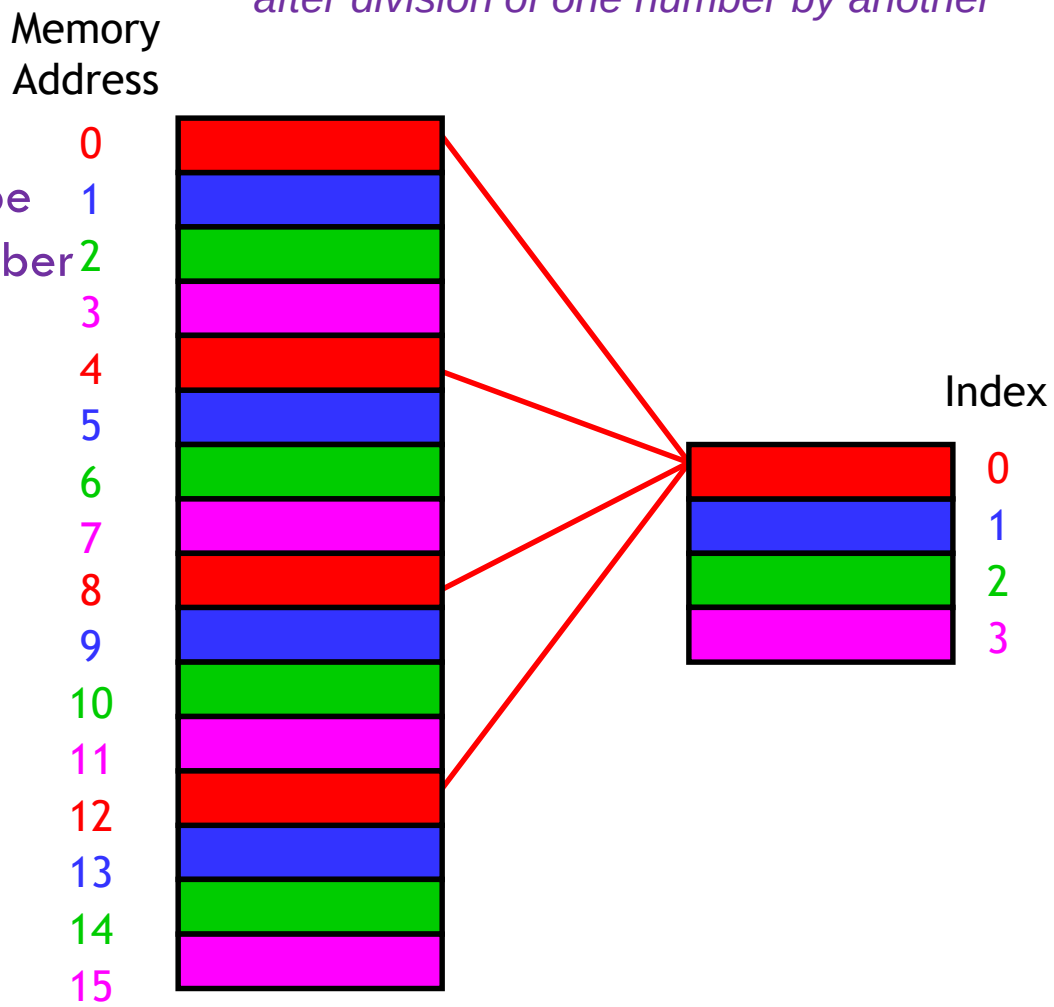
- One way to figure out which cache block a particular memory address should go to is to use the **modulo** (remainder) operator
- Let x be block number in cache, y be block number of DDR, and n be number of blocks in cache, then mapping is done with the help of the equation

  $$x = y \bmod n$$

- For instance, with the four-block cache here, address 14 would map to cache block 2

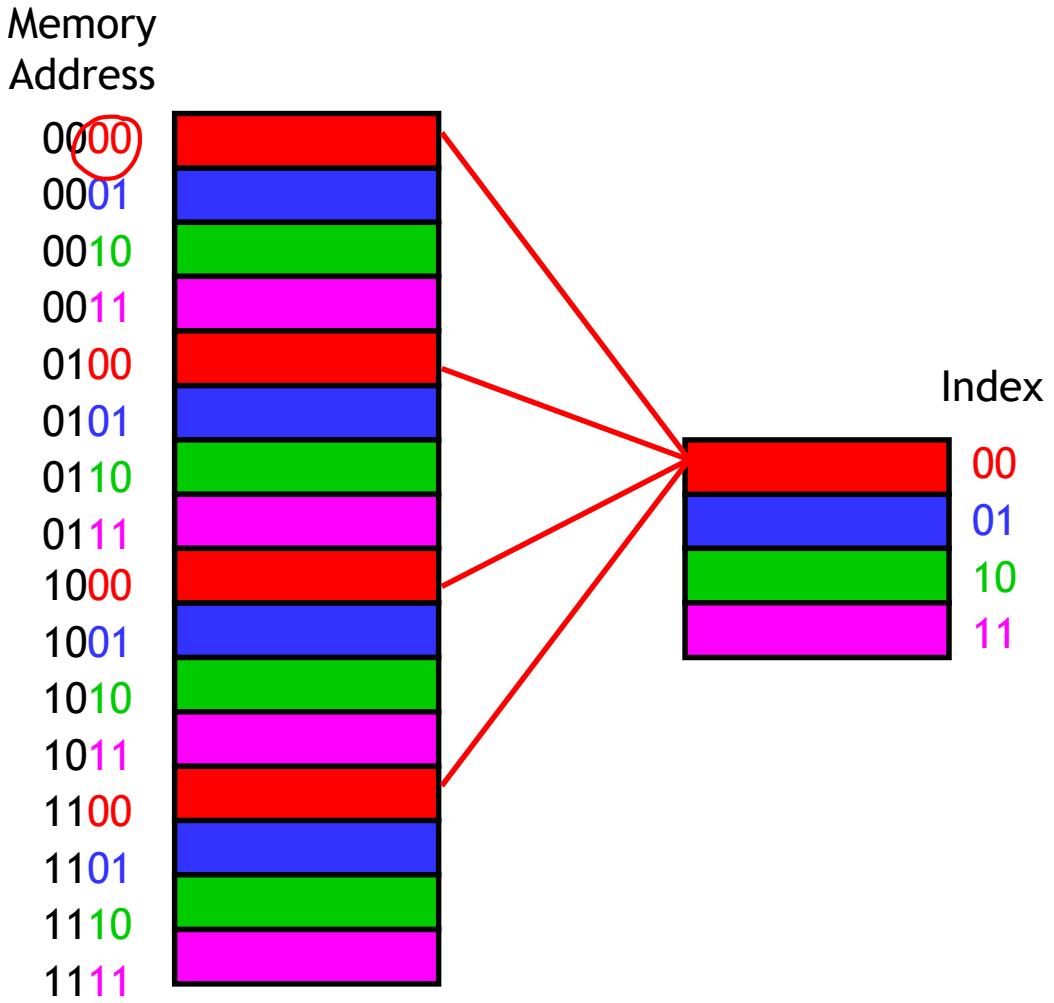
  $$14 \bmod 4 = 2$$

the modulo operation finds the remainder after division of one number by another



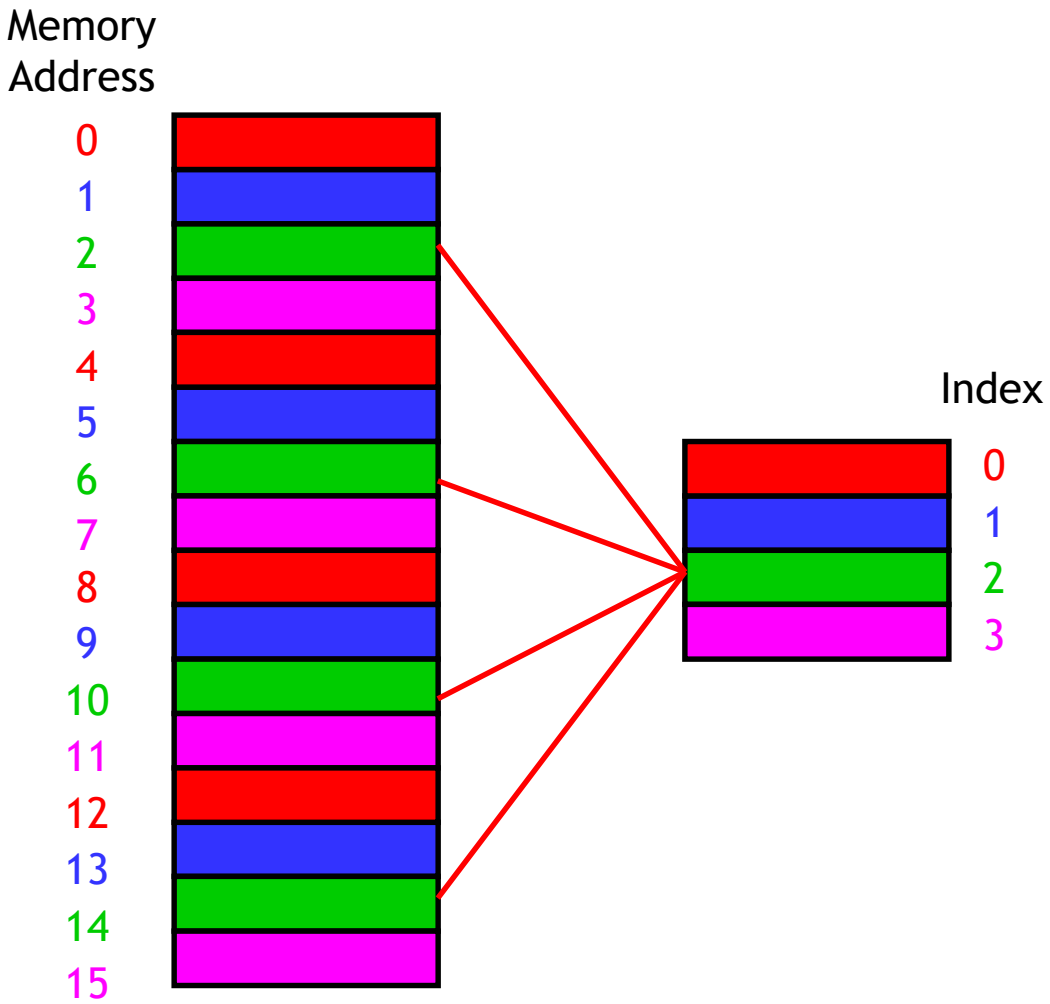
Where should we put data in the cache? (3)

- **An equivalent way to find the placement of a memory address in the cache is to look at the least significant *k* bits of the address**
- In a four-entry cache we would check the two least significant bits of our memory addresses
- Again, you can check that address 14_{10} (1110_2) maps to cache block 2_{10} (10_2)
- Taking the least *k* bits of a binary value is the same as computing that value mod n



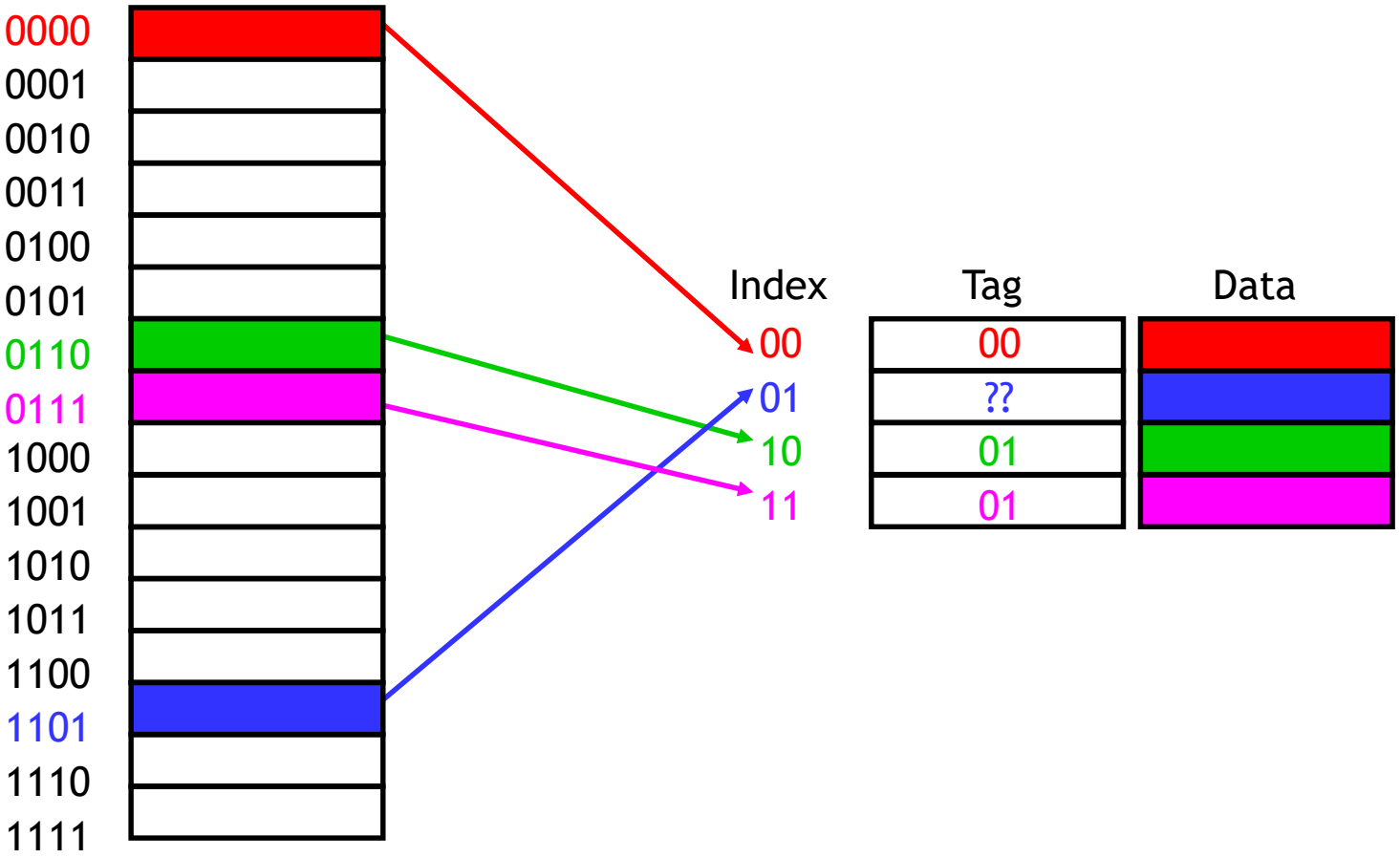
How can we find data in the cache?

- **How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?**
- If we want to read memory address *i*, we can use the mod trick to determine which cache block would contain *i*
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14



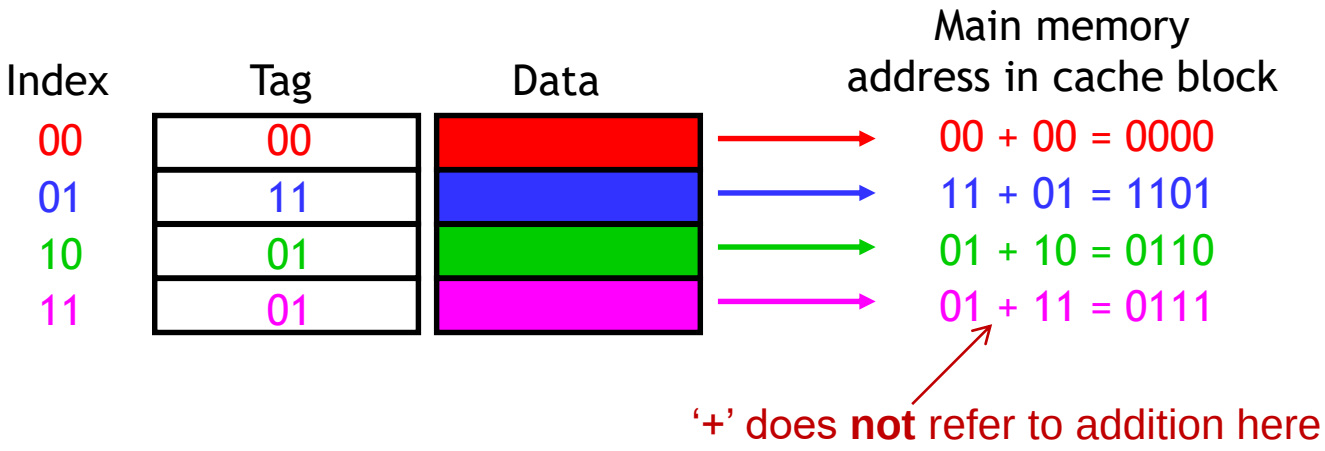
Adding tags

- **The solution is to add tags to the cache,** which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block



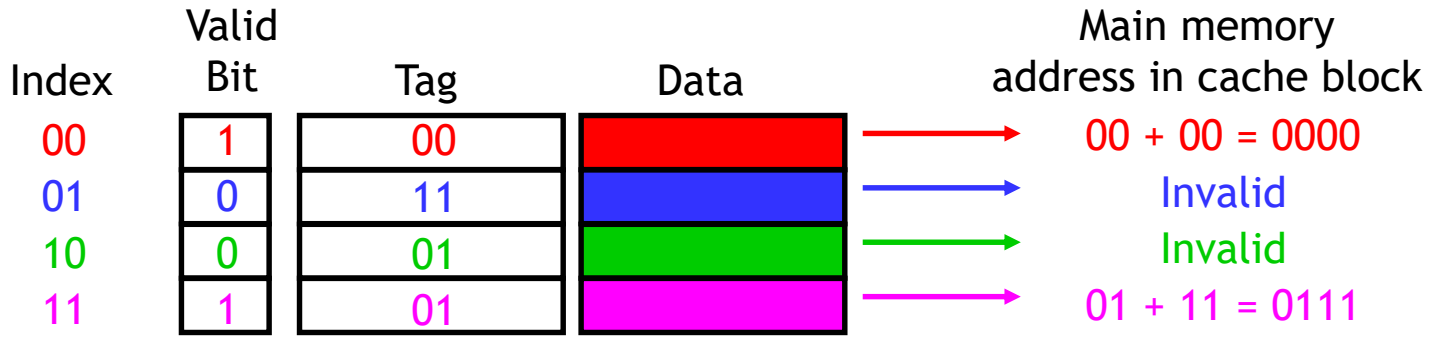
what's in the cache

- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices



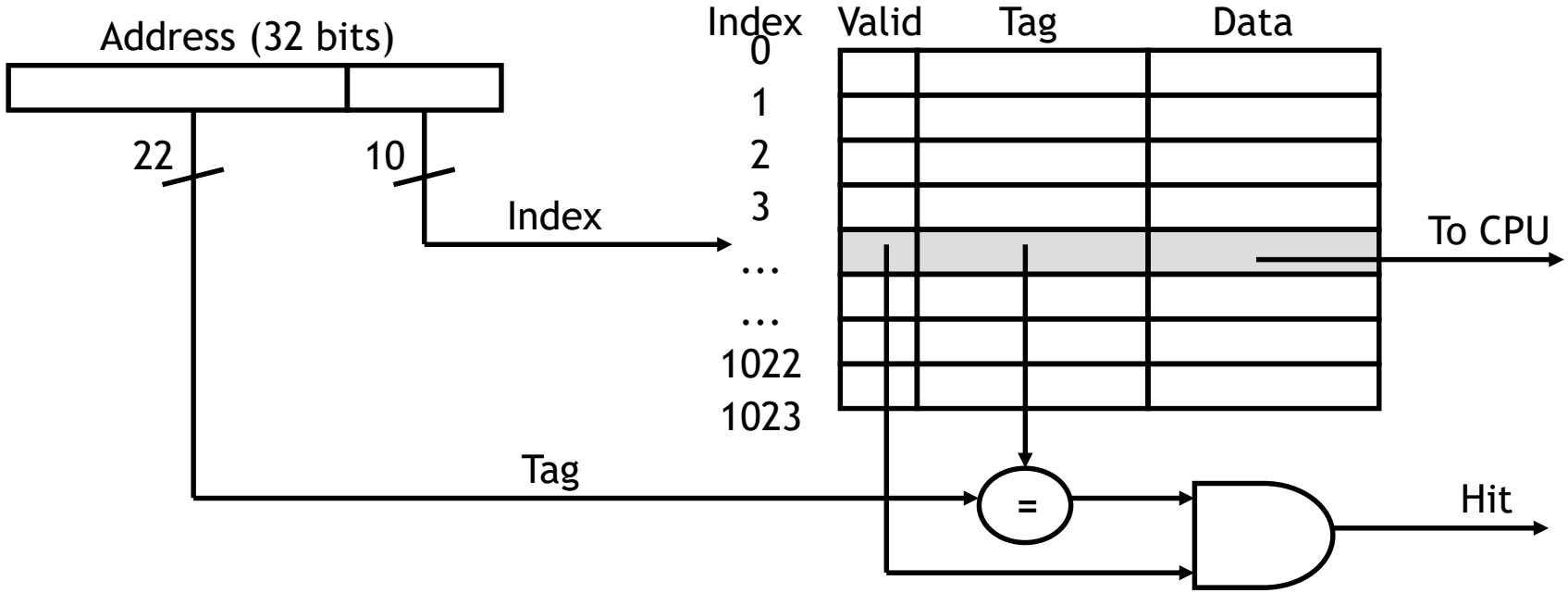
the valid bit

- **Initially, the cache is empty and does not contain valid data, but trash**
- Thus, we add a **valid bit** for each cache block
  - When the system is initialized, all the valid bits are set to 0
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1



cache hit

- **Every memory has its memory controller, a hardware mechanism responsible for finding the words in memory, loading/storing etc**
- When the CPU tries to read from memory, the address will be sent to the cache controller
  - The lowest *k* bits of the address will index a block in the cache
  - If the block is valid and the tag matches the upper (*m* - *k*) bits of the *m*-bit address, then that data will be sent to the CPU
- Here is a diagram of a 32-bit memory address and a 2^{10} byte cache



cache miss

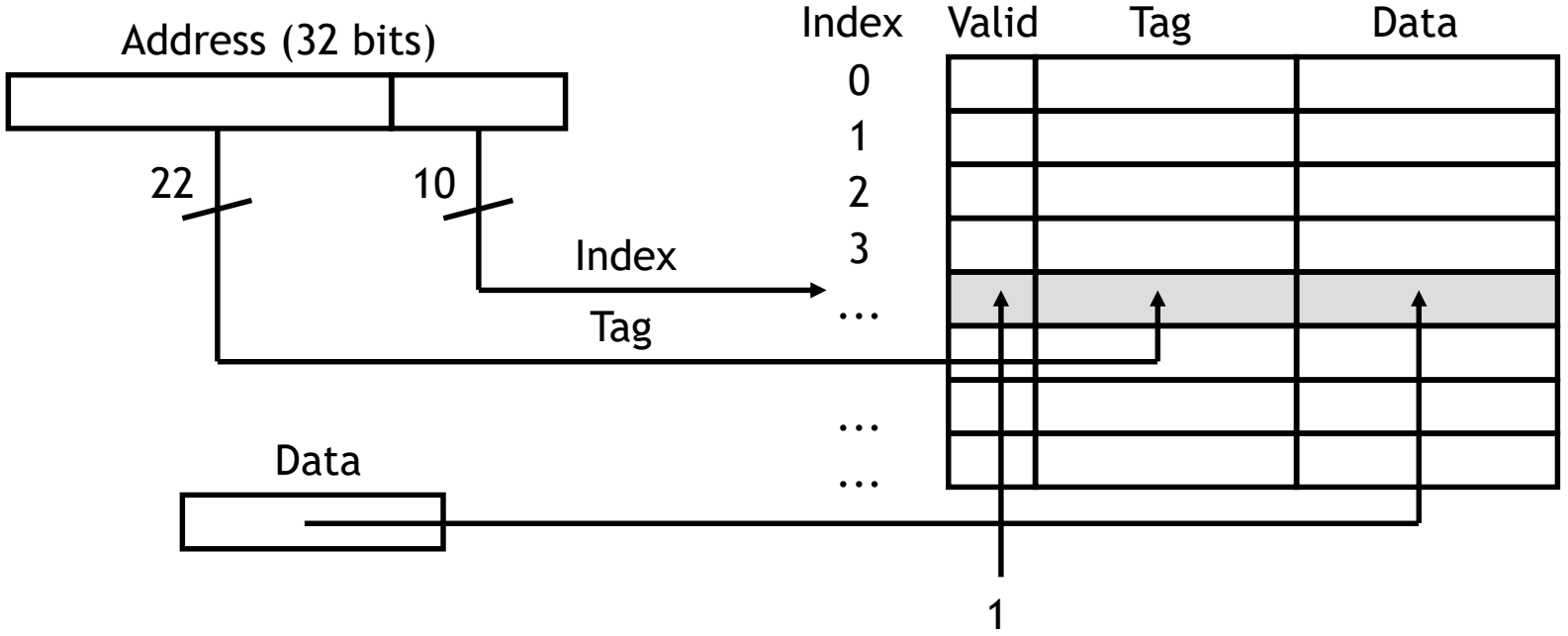
- **The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache)**

- In a two level memory hierarchy, L1 Cache misses are somehow expensive, but L3 cache misses are very expensive

- However, the slower main memory accesses are inevitable on an L3 cache miss

Copying a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward
  - The lowest *k* bits of the address specify a cache block
  - The upper (*m* − *k*) address bits are stored in the block's tag field
  - The data from main memory is stored in the block's data field
  - The valid bit is set to 1

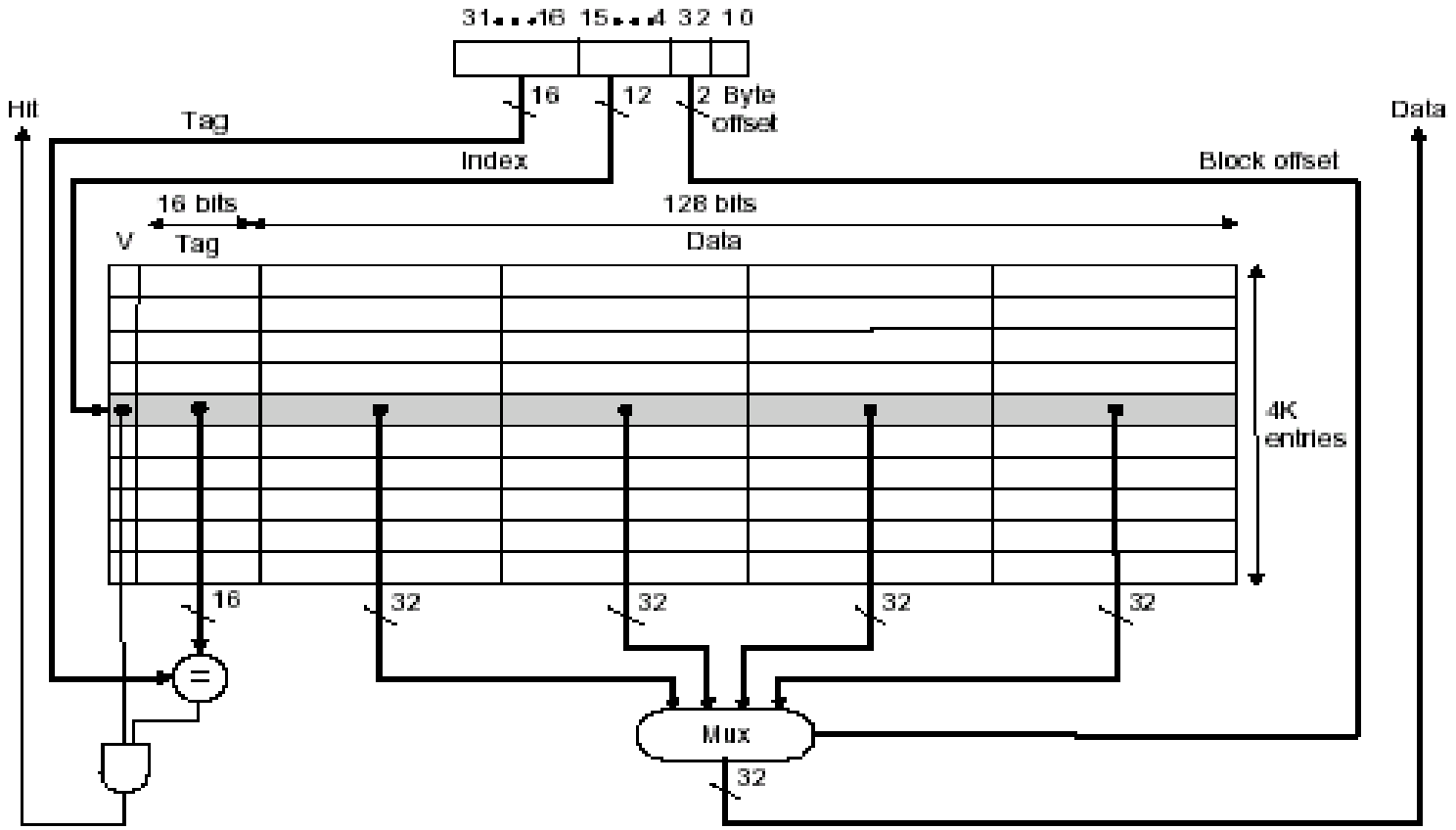


What if the cache fills up?

- Eventually, the small cache memory might fill up. To load a new block from DDR, we'd have to replace one of the existing blocks into the cache... which one?
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data
  - Normally, the least recently used (LRU) replacement policy is used, which assumes that least recently used data are less likely to be requested than the most recently used ones
  - So, in a cache miss, cache throws out the cache line that has been unused for the longest time

A more realistic **direct mapped** cache memory

- Normally, a cache line contains 128/256/512 bits of data. In most programming languages, an integer uses 32 bits (4 bytes)



Associativity

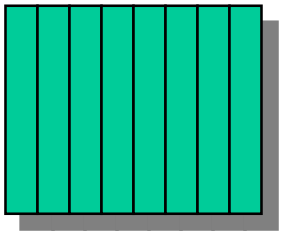
- The replacement policy decides where in the cache a copy of a particular entry of main memory will go
- **So far, we have seen directed mapped cache only**
  - each entry in main memory can go in just one place in the cache
- Although direct mapped caches are simpler and cheaper they are not performance efficient as they give a large number of cache misses
- **There are three types of caches regarding associativity**
  - **Direct mapped**
  - **N-way Associative**
  - **Fully associative**

Associative Caches

- Block 12 placed in 8 block cache:

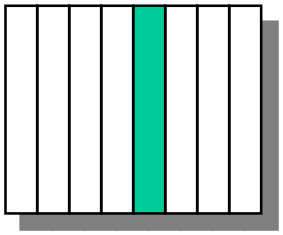
Fully associative: block 12 can go anywhere

Block no. 0 1 2 3 4 5 6 7



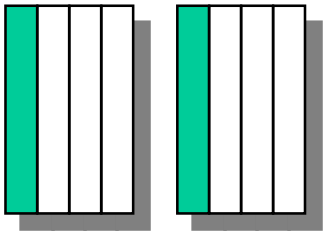
Direct mapped: block 12 can go only into block 4 (12 mod **8** = 4)

Block no. 0 1 2 3 4 5 6 7

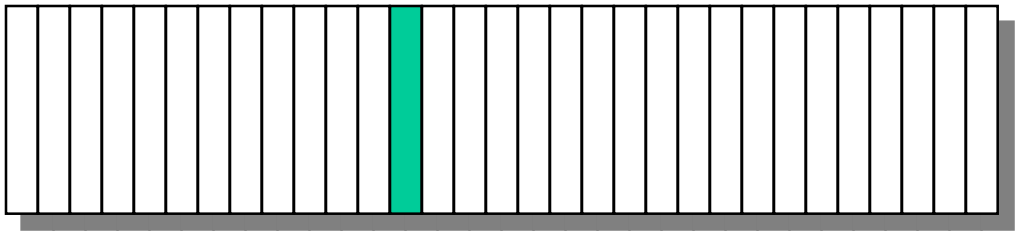


2 way Set associative (like having two half size direct mapped caches): block 12 can go in either of the two block 0 (12 mod **4** = 0)

Block no. 0 1 2 3 0 1 2 3



Block-frame address



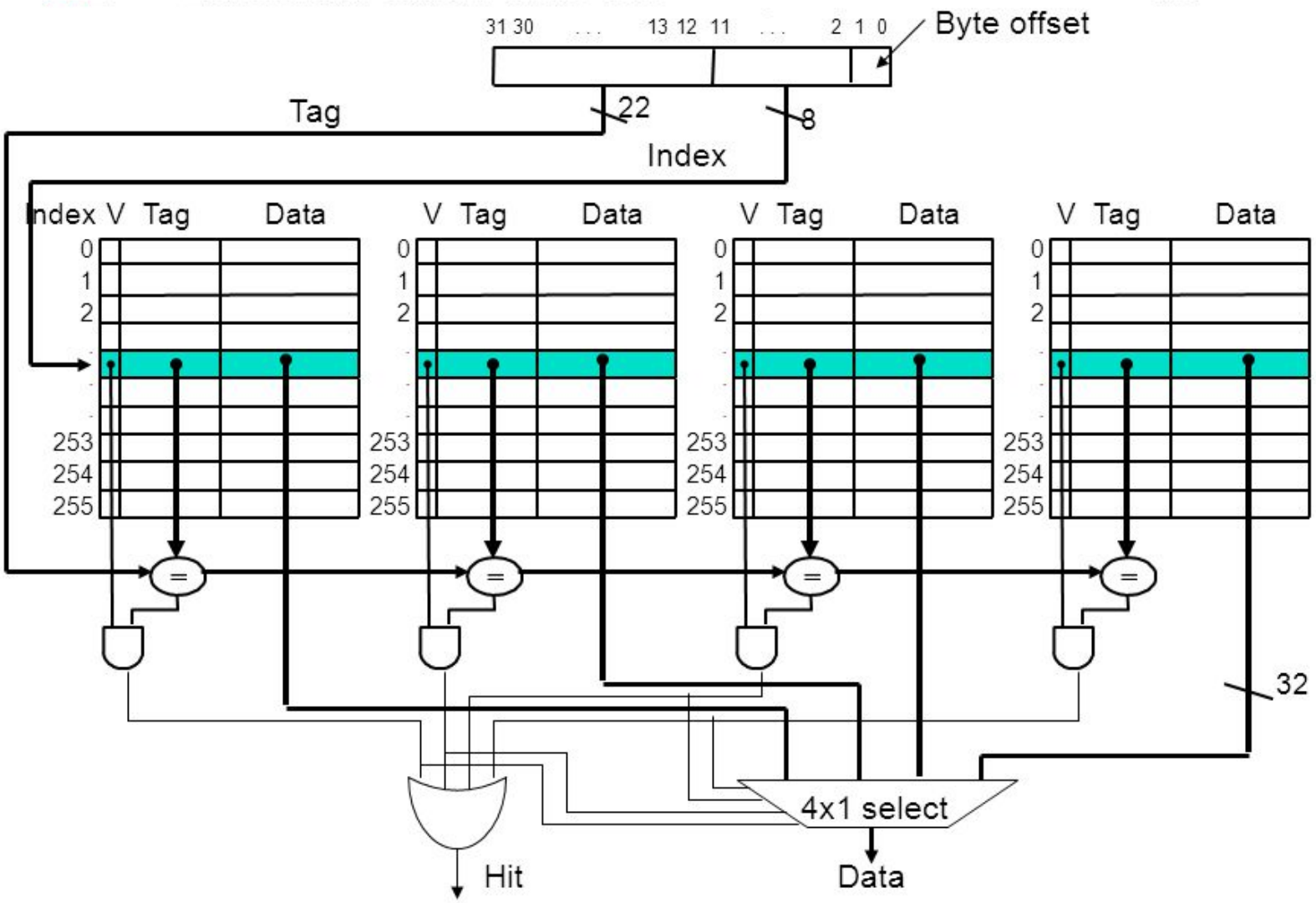
Block no. 0 1 2 3 4 5 6 7 8 9 0 1 **2** 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Set-associative cache memories

- **A 4-way associative cache consists of four direct-mapped caches that work in parallel**
- **Normally, L1 caches are of 8 way associative, while L2/L3 caches are of 16/24 way associative, i.e., 16/24 direct mapped caches in parallel**
- Data are found in one cache among four by using an address which is stored in one of the four caches

- **Set associative caches are the most used as they present the best compromise between cost and performance**

The 4-way set-associative cache architecture



Write policies - In a write request, are we going to write to all memories in memory hierarchy or not?

- ***Write through*** – **Data are written to all memories in memory hierarchy**
  - Disadvantage: Data are written into multiple memories every time and thus it takes more time

- ***Write back*** – **Data are written only to L1 data cache; only when this block is replaced, it is written in L2. If this block is replaced, then it is written in main memory**
  - Requires a "dirty bit"
  - more complex to implement, since it needs to track which of its locations have been written over, and mark them as dirty for later writing to the lower memory

Further Reading

- Chapter 4 in 'Computer Organization and architecture' available at http://home.ustc.edu.cn/~leedsong/reference_books_tools/Computer%20Organization%20and%20Architecture%2010th%20-%20William%20Stallings.pdf