Compilers for Embedded Systems

Integrated Systems of Hardware and Software

Lecture 2

Dr. Vasilios Kelefouras

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

School of Computing
(University of Plymouth)

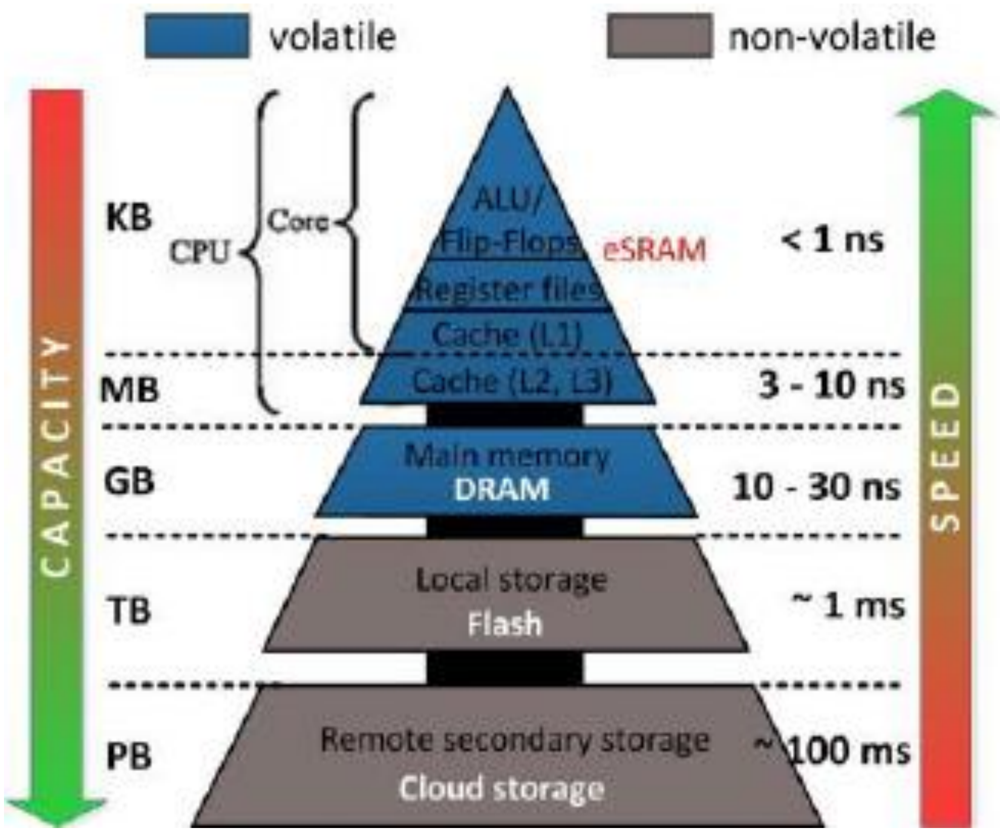
Outline of this Lecture

- Memory Hierarchy
- Cache
- Data Locality
- Examples

Memory Hierarchy (1)

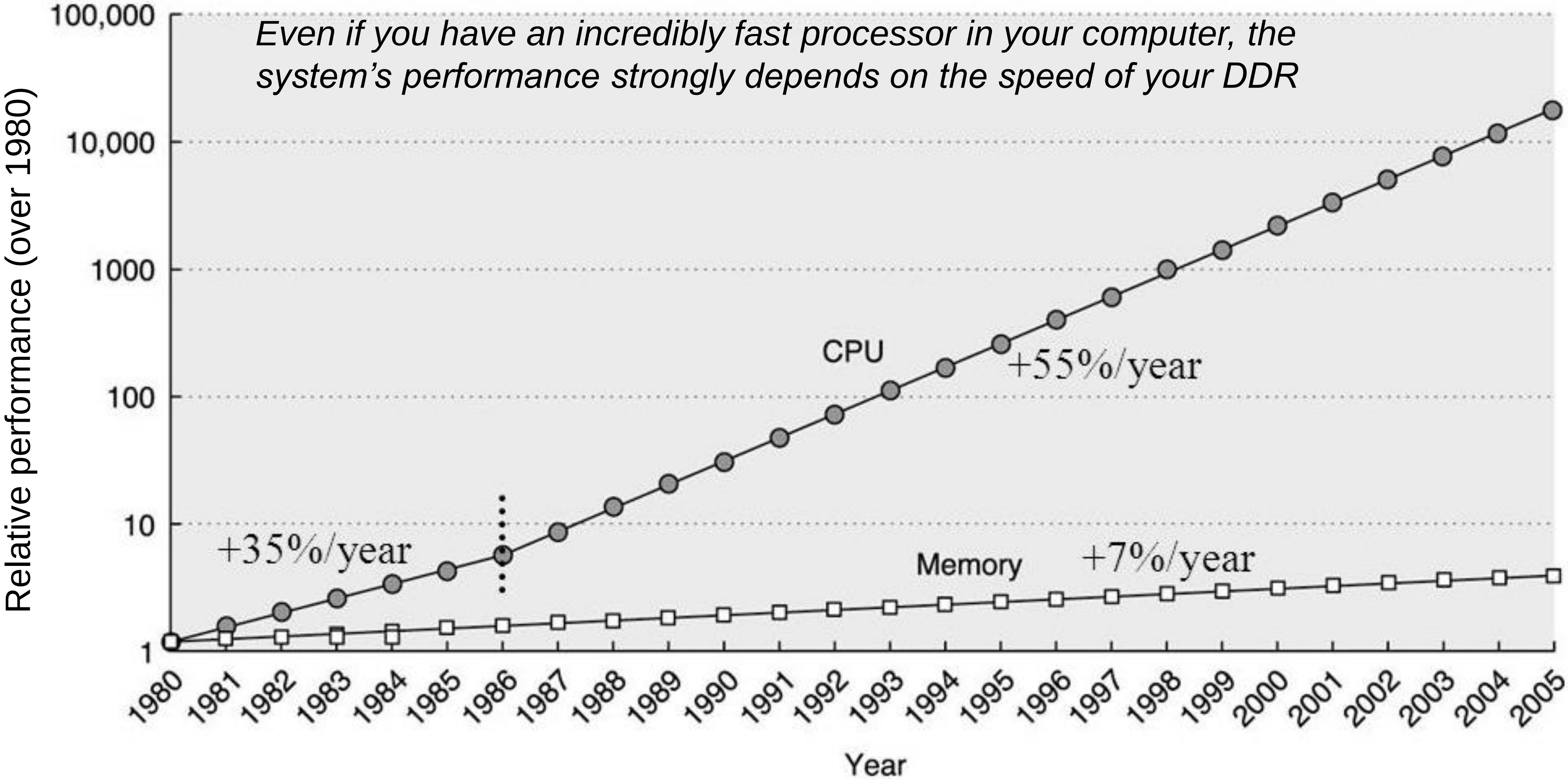
- The memory hierarchy is the **main performance bottleneck** in modern computer systems as the gap between the speed of the processor and the memory continues to grow larger
  - This is also known as the **Memory Wall Problem**
- This problem becomes even worse in an embedded system
  - In an embedded system, memory hierarchy takes a huge portion of both the
    - chip area
    - power consumption

Memory Hierarchy (2)



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

Memory Wall Problem



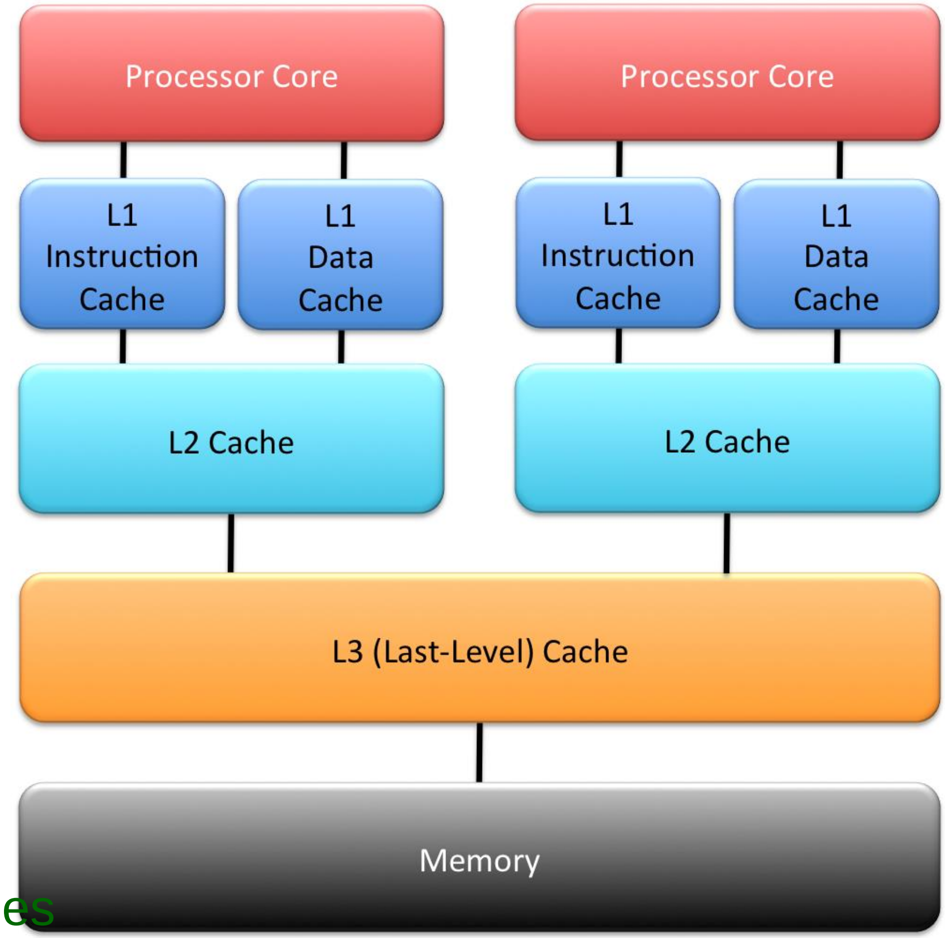
Take from https://slideplayer.com/slide/7075269/

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, 2 or 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

Cache 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
- There are two basic measurements of cache performance
  - The **hit rate** is the percentage of memory accesses that are handled by the cache
  - The **miss rate** (1 - hit rate) is the percentage of accesses that must be handled by the slower lower level memory
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster

Data Locality (1)

- Code and data are not accessed randomly
- Locality is the tendency of a processor to access the same set of memory locations repetitively over a short period of time
  - Data locality is a key to good performance on all modern CPUs
- It is very difficult and time consuming to figure out what data will be the “most frequently accessed” before a program actually runs
  - However, for **static programs** (the control flow path is known at compile time) it can be done
    - Only by experience programmers though
  - Regarding **dynamic** programs it is impossible
- This makes it hard to know what to store into the small, precious cache memory

Data Locality (2)

- But in practice, most programs exhibit ***locality***, which the cache can take advantage of
  - **The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again**
  - **The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses**

Temporal Locality in Data

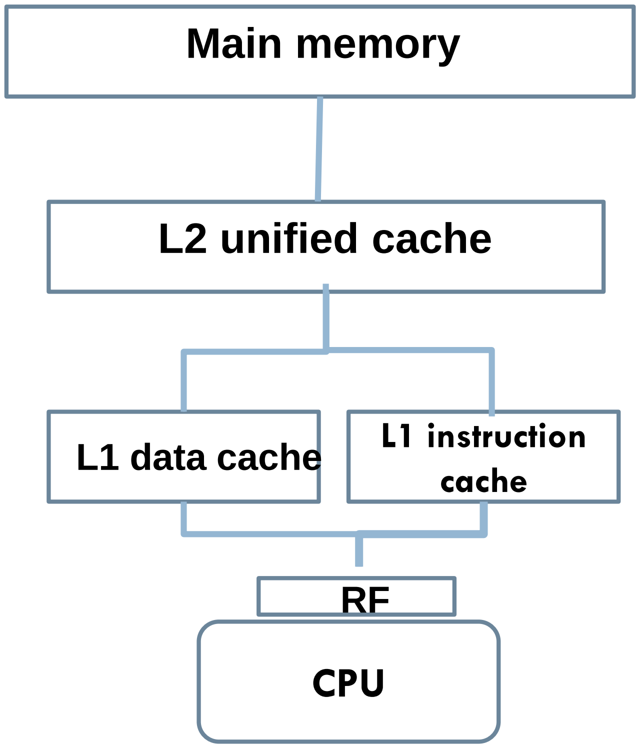
- Programs often access the same variables over and over, especially within loops, e.g., below, sum, i and B[5] are repeatedly read/written
- **Commonly-accessed variables can be kept in registers, but this is not always possible as there is a limited number of registers**

- Sum and i variables are a) of small size, b) reused many times, and therefore it is efficient to remain in the CPU's registers
- B[k] remains unchanged during the innermost loop and therefore it is efficient to remain in a CPU register
- The whole A[ ] array is accessed 3 times and therefore it will remain in the cache (depending on its size)

```
sum = 0;
for (k = 0; k < 3; k++)
  for (i = 0; i < N; i++)
   sum = sum + A[i] + B[k];
```

How caches take advantage of temporal locality

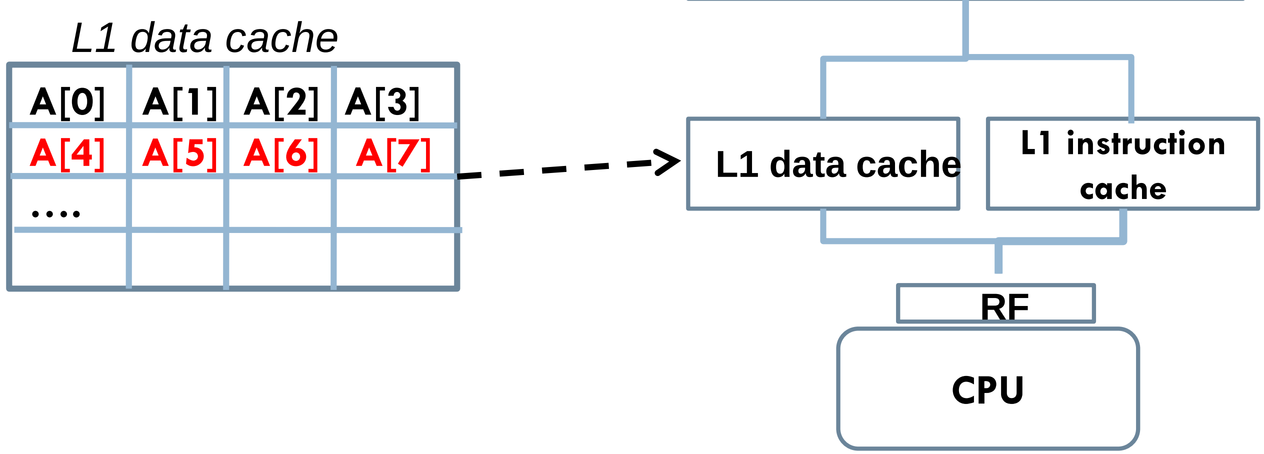
- Every time the processor reads from an address in main memory, a copy of that datum is also stored in the cache
  - The **next time** that the same address is read, the **datum is read from the cache *instead*** of accessing the slower DDR memory
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster
- This takes advantage of temporal locality - **commonly accessed data are stored in the faster cache memory**



Spatial Locality in Data

- Programs often access data that are stored in contiguous memory locations
  - Arrays, like A[ ] in the code below are always stored in memory contiguously – this task is performed by the compiler

```
sum = 0;
for (i = 0; i < N; i++)
    sum = sum + A[i];
```

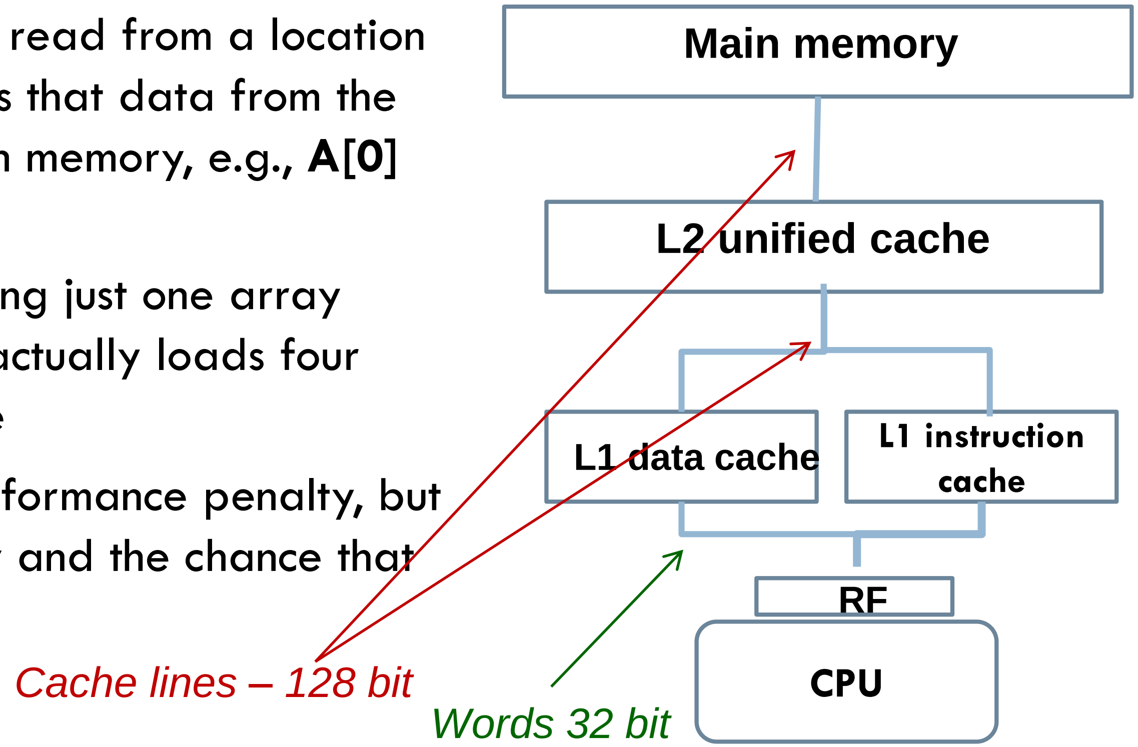


How caches take advantage of Spatial locality

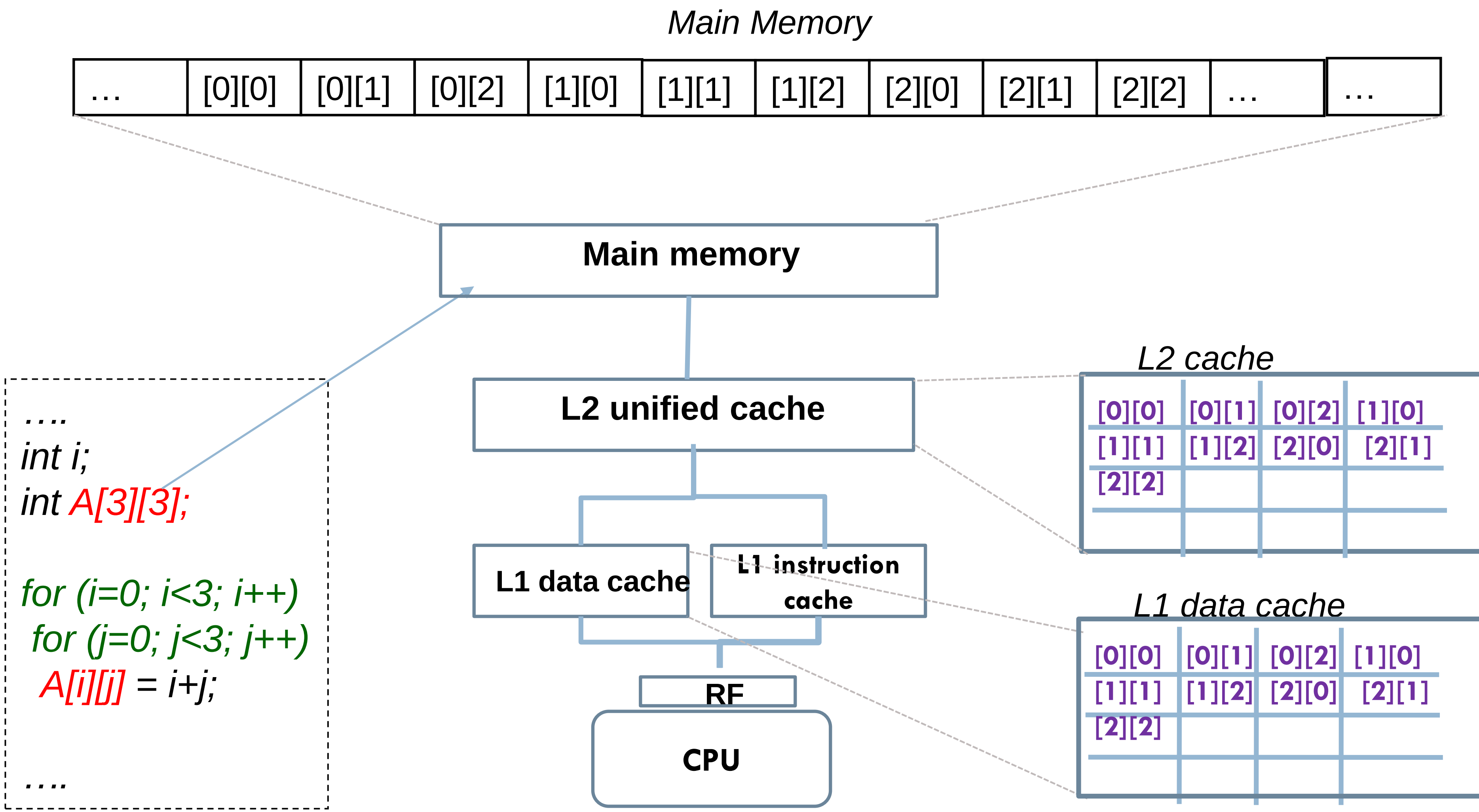
- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache
- **But instead of just copying the contents of location *i*, it copies *several* values into the cache at once (cache line)**
  - If the CPU later does need to read from a location in that cache line, it can access that data from the cache and not the slower main memory, e.g., **A[0] and A[3]**
  - For example, instead of loading just one array element at a time, the cache actually loads four /eight array elements at once
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data

L1 data cache

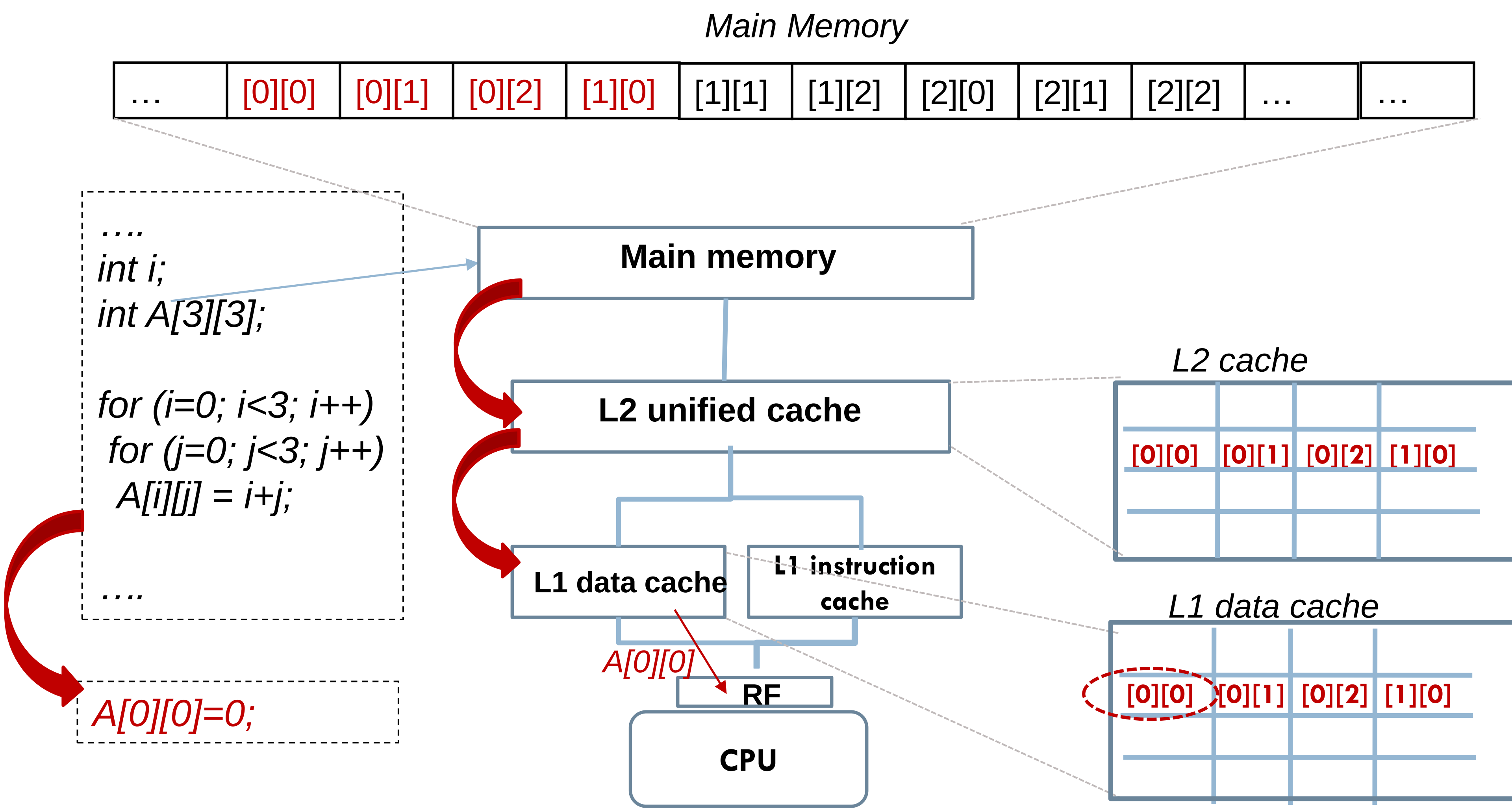
A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
….			



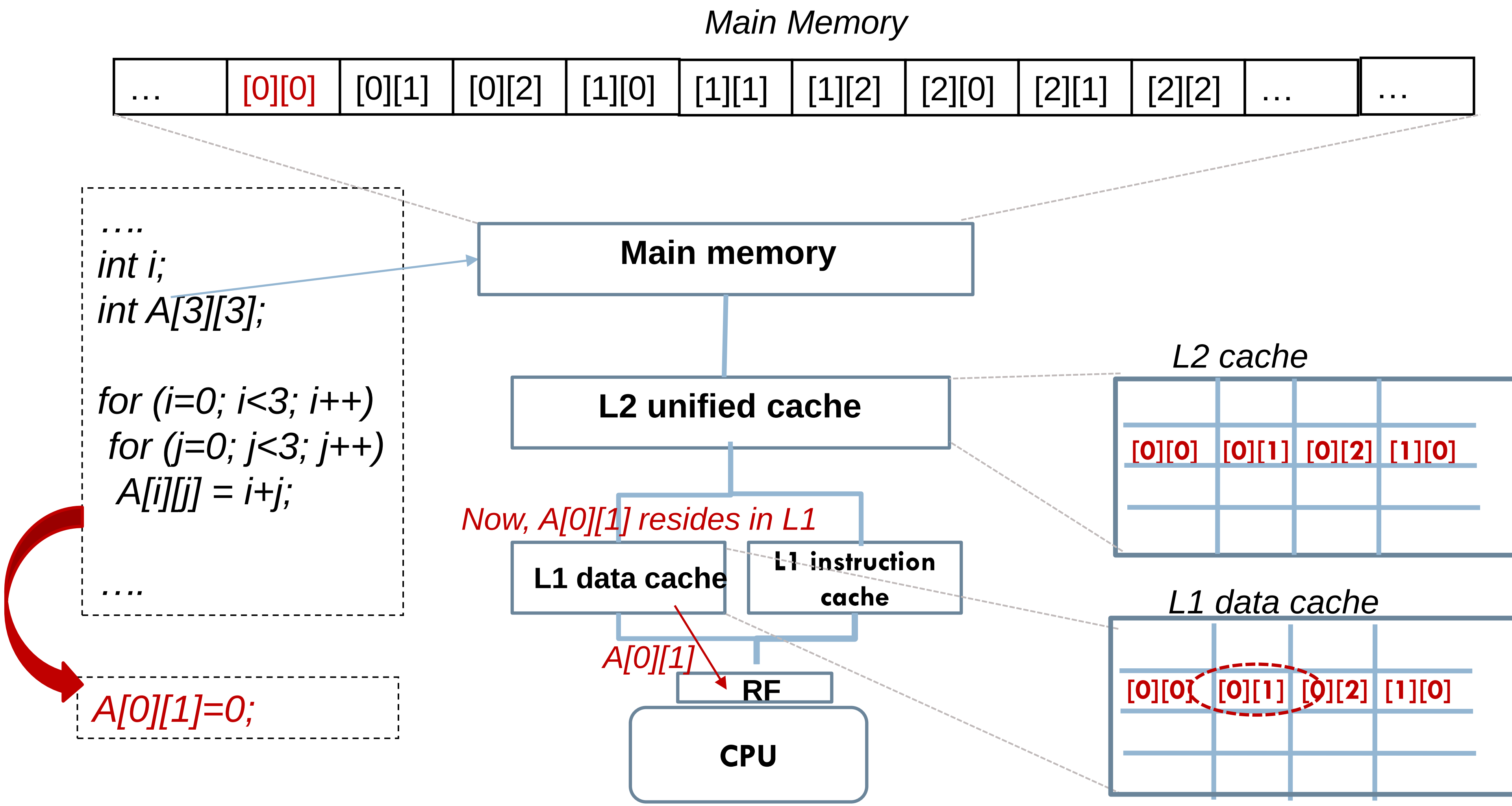
Accessing arrays – From a Hardware Perspective (1) In C/C++, row-wise is the right way



Accessing arrays – From a Hardware Perspective (2)
In C/C++, row-wise is the right way



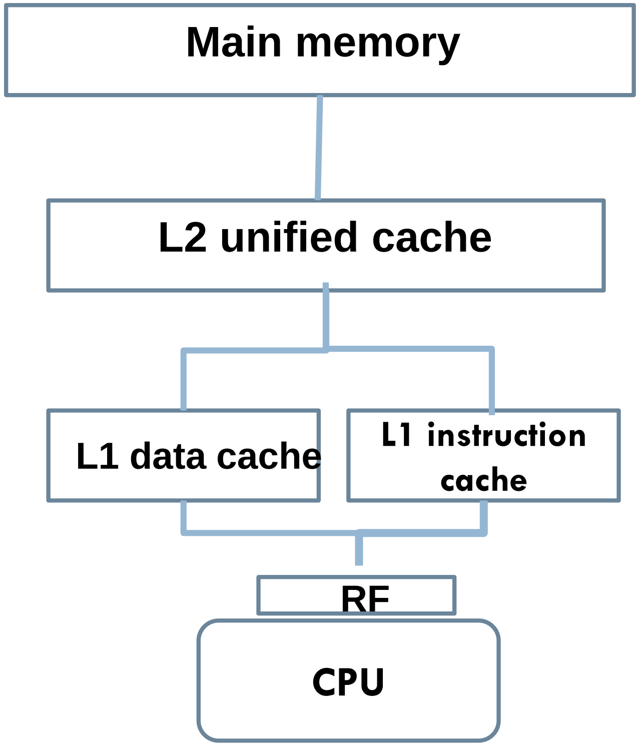
Accessing arrays – From a Hardware Perspective (3)
In C/C++, row-wise is the right way



Accessing arrays – the wrong way (1)

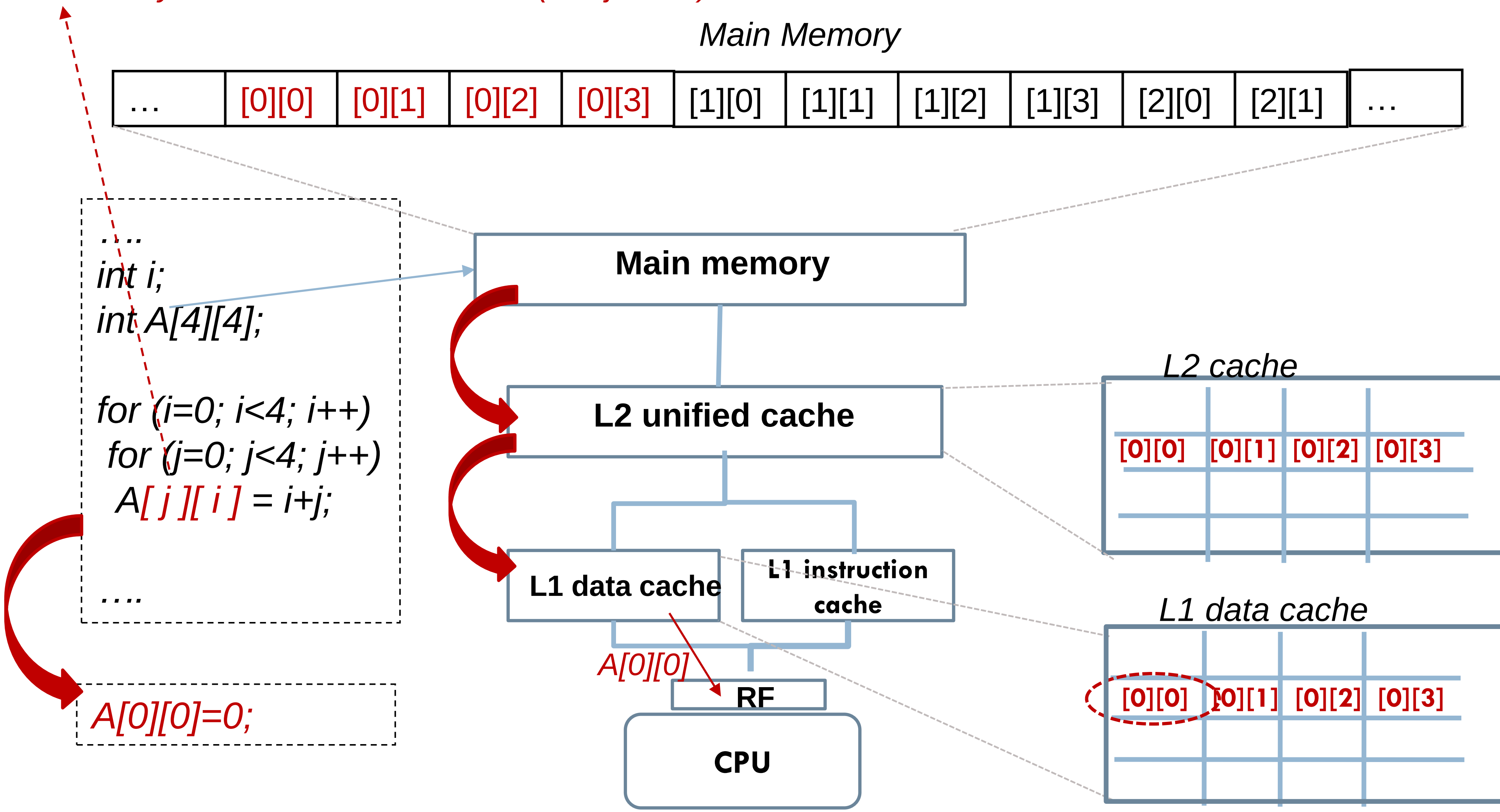
- It is efficient to accesses arrays' elements in sequential order
  - Array elements are loaded into cache **in blocks**, e.g., A[0-3], A[4-7] etc
  - Accessing A[3] just after A[0] is a cache hit – **spatial locality**

- Let's have a look at the next slide where the array's elements are not accessed in sequential order
- In each iteration an entire L2 and L1 cache line is loaded, which is inefficient



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

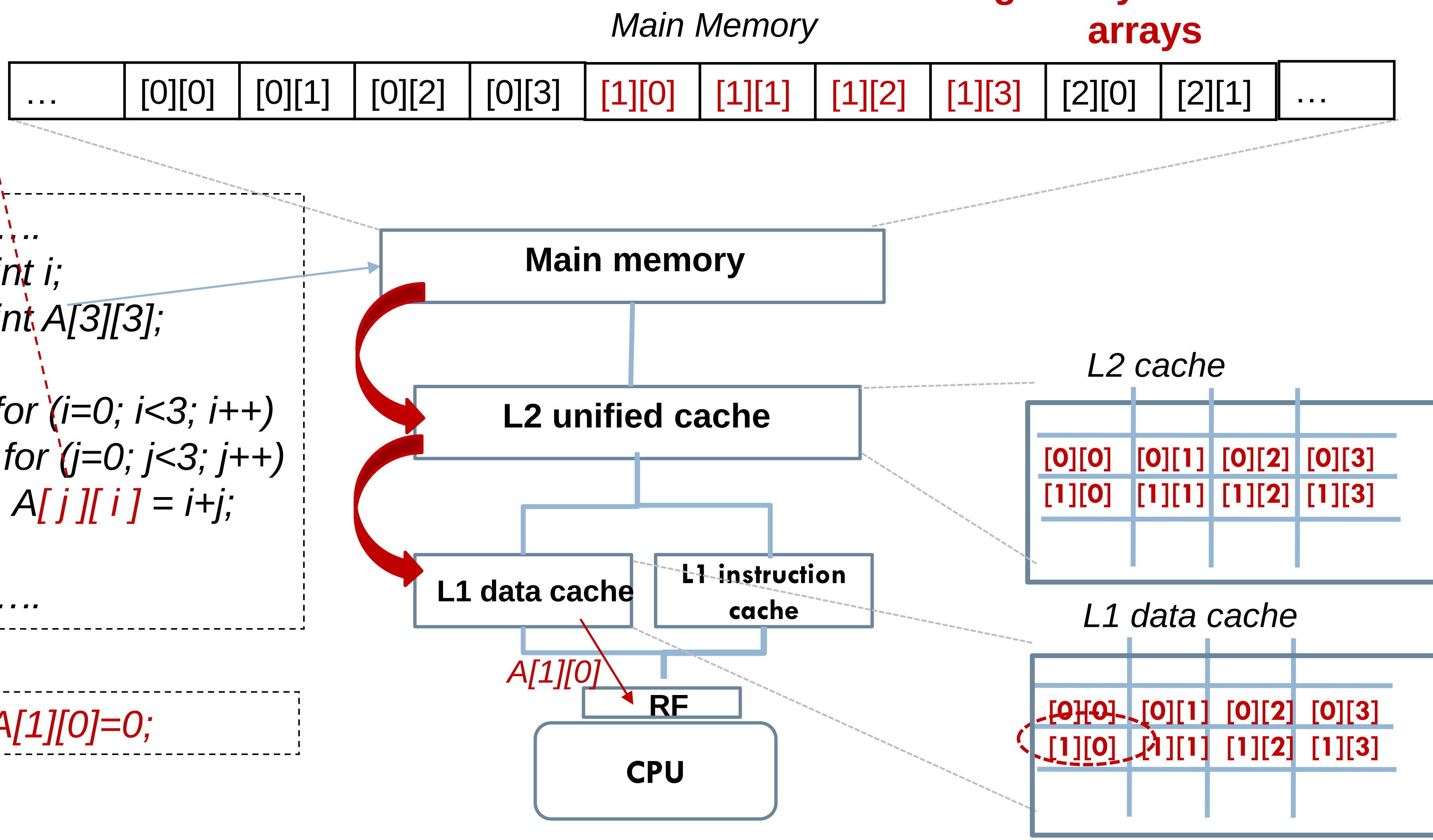
The array is accessed column-wise (first j then i)



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

In Fortran, the arrays are stored into memory column-wise and therefore this is the right way to access the arrays

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




```
….
int i;
int A[3][3];

for (i=0; i<3; i++)
 for (j=0; j<3; j++)
  A[ j ][ i ] = i+j;

….
```

A[1][0]=0;

Accessing arrays
Simulation Results using Valgrind Cachegrind
Column-Wise case - N=1000

Activities Text Editor ▾ Thu 09:02 en ▾

Open ▾ column.txt ~/Desktop/comp3001/my/Labs/cache Save

*TXT_File.txt × | init_arrays.c × | row.txt × | column.txt ×

```
--------------------------------------------------------------------------------
-- Auto-annotated source: /home/user01/Desktop/comp3001/my/Labs/cache/init_arrays.c
--------------------------------------------------------------------------------
Ir                 I1mr        ILmr        Dr          D1mr        DLmr        Dw                 D1mw               DLmw

-- line 4 ----------------------------------------
        .           .           .           .           .           .                  .                  .                .     #define N 1000  //arrays input size
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     //In C, all the routines must be declared
        .           .           .           .           .           .                  .                  .                .     void initialize();
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     int A[N][N];
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
    2,001 ( 0.04%) 0           0           0           0           0                  0                  0                0     int main( ) {
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     initialize();
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     return 0; //normally, by returning zero, we mean that the program ended successfully.
        2 ( 0.00%) 0           0           1 ( 0.00%) 1 ( 0.07%) 0                  0                  0                0     }
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     void initialize(){
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     int i,j;
        .           .           .           .           .           .                  .                  .                .
    6,002 ( 0.12%) 1 ( 0.12%) 1 ( 0.13%) 0           0           0                  0                  0                0     for (i=0;i<N;i++)
2,000,000 (39.00%) 0           0           0           0           0                  0                  0                0      for (j=0;j<N;j++){
3,000,000 (58.51%) 0           0           0           0           0          1,000,000 (99.05%) 1,000,000 (99.95%) 62,500 (99.18%)   A[j][i]=i+j;
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     }
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .     }
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
        .           .           .           .           .           .                  .                  .                .
-- line 36 ----------------------------------------

--------------------------------------------------------------------------------
The following files chosen for auto-annotation could not be found:
--------------------------------------------------------------------------------
```

1000000 writes

1000000 dL1 write misses

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 INS

Accessing arrays

Simulation Results using Valgrind Cachegrind

Row-Wise case - N=1000

tivities Text Editor ▾ Thu 09:02 en₁ ▾

Open ▾ row.txt ~/Desktop/comp3001/my/Labs/cache Save

*TXT_File.txt × init_arrays.c × row.txt × column.txt ×

```
-- Auto-annotated source: /home/user01/Desktop/comp3001/my/Labs/cache/init_arrays.c
--------------------------------------------------------------------------------
Ir                  I1mr       ILmr       Dr         D1mr       DLmr       Dw                D1mw              DLmw

-- line 4 ----------------------------------------
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
declared
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
    2,001 ( 0.04%)  0          0          0          0          0          0                 0                 0
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
zero, we mean that the program ended successfully.
        2 ( 0.00%)  0          0          1 ( 0.00%) 1 ( 0.07%) 0         0                 0                 0
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
    4,002 ( 0.08%)  0          0          0          0          0          0                 0                 0
2,000,000 (39.02%)  0          0          0          0          0          0                 0                 0
3,000,000 (58.53%)  0          0          0          0          0          1,000,000 (99.05%) 62,500 (99.13%) 62,500 (99.18%)
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
        .           .          .          .          .          .          .                 .                 .
-- line 36 ----------------------------------------

--------------------------------------------------------------------------------
The following files chosen for auto-annotation could not be found:
--------------------------------------------------------------------------------
  /build/glibc-OTsEL5/glibc-2.27/elf/../sysdeps/x86_64/dl-machine.h
  /build/glibc-OTsEL5/glibc-2.27/elf/dl-lookup.c
```

```c
#define N 1000  //arrays input size

//In C, all the routines must be

void initialize();

int A[N][N];

int main( ) {

initialize();

return 0; //normally, by returning

}

void initialize(){

int i,j;

for (i=0;i<N;i++)
 for (j=0;j<N;j++){
  A[i][j]=i+j;

}

}
```

There are 16 times less misses as each dL1 cache line contains 16 elements. Keep in mind that 62,500x16=1,000,000.

1000000 writes

62500 dL1 write misses

x16 times faster

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

How important is cache size?

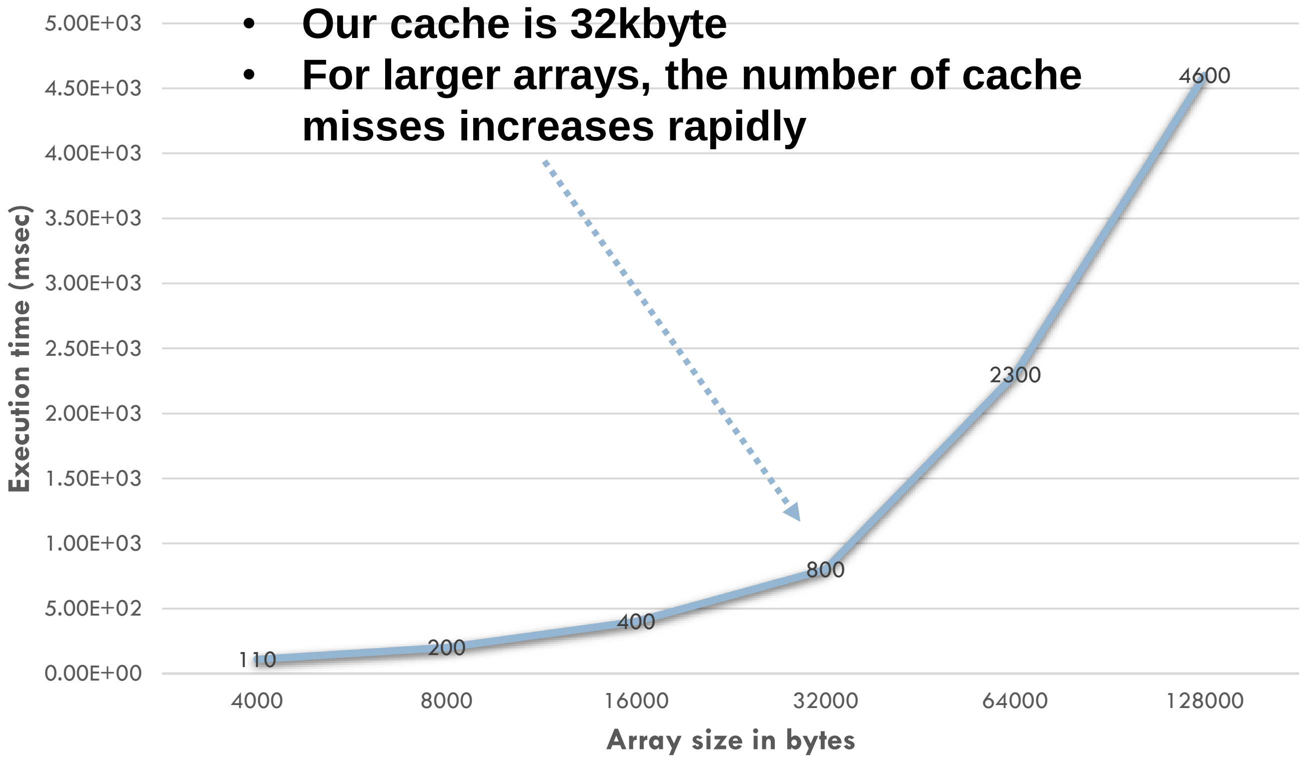
The following code can be seen as a benchmark that experimentally finds the cache size

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

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

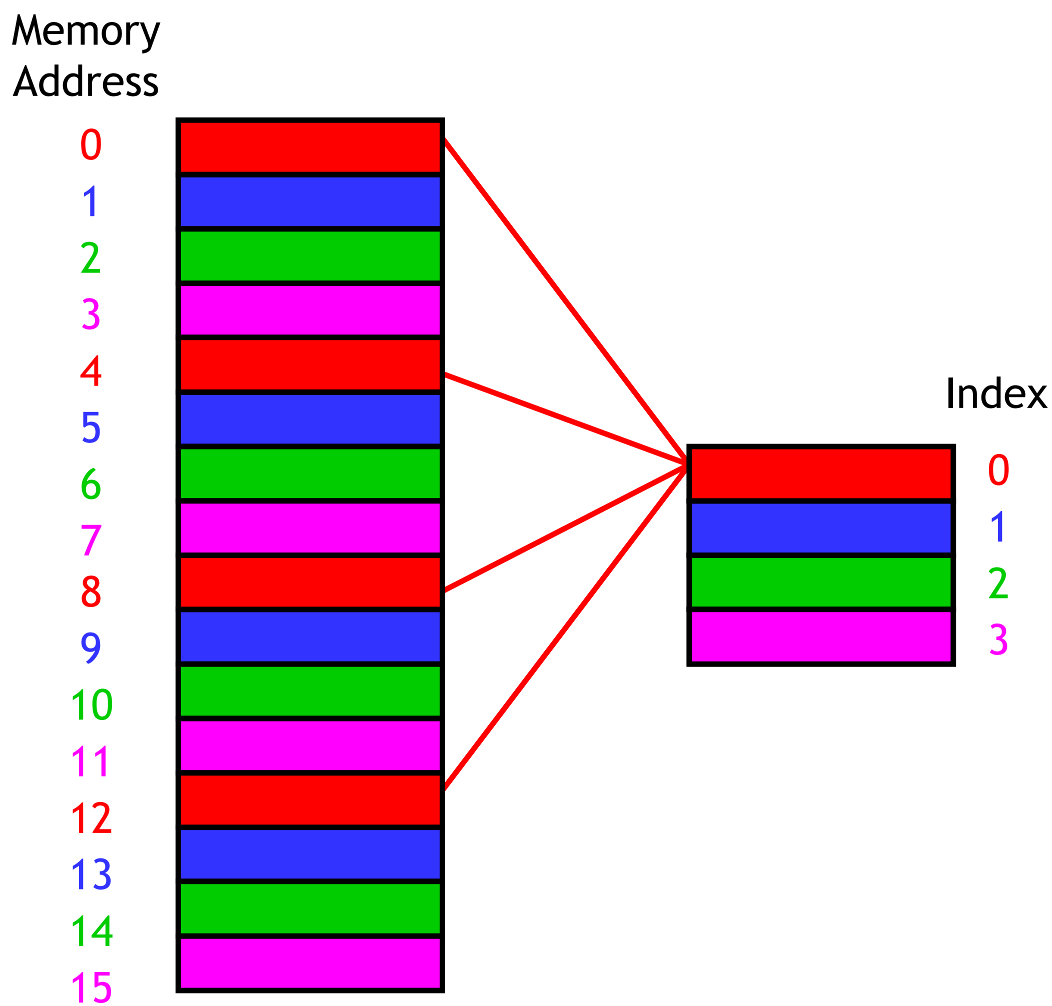
// N=1000, 2000, 4000, 8000, 16000, 32000

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



Direct Mapped Cache
(not used by modern processors)

- 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



Direct Mapped Cache (2)
(not used by modern processors)

- One way to figure out which cache block a particular memory address should go to is to use the **modulo** (remainder) operator

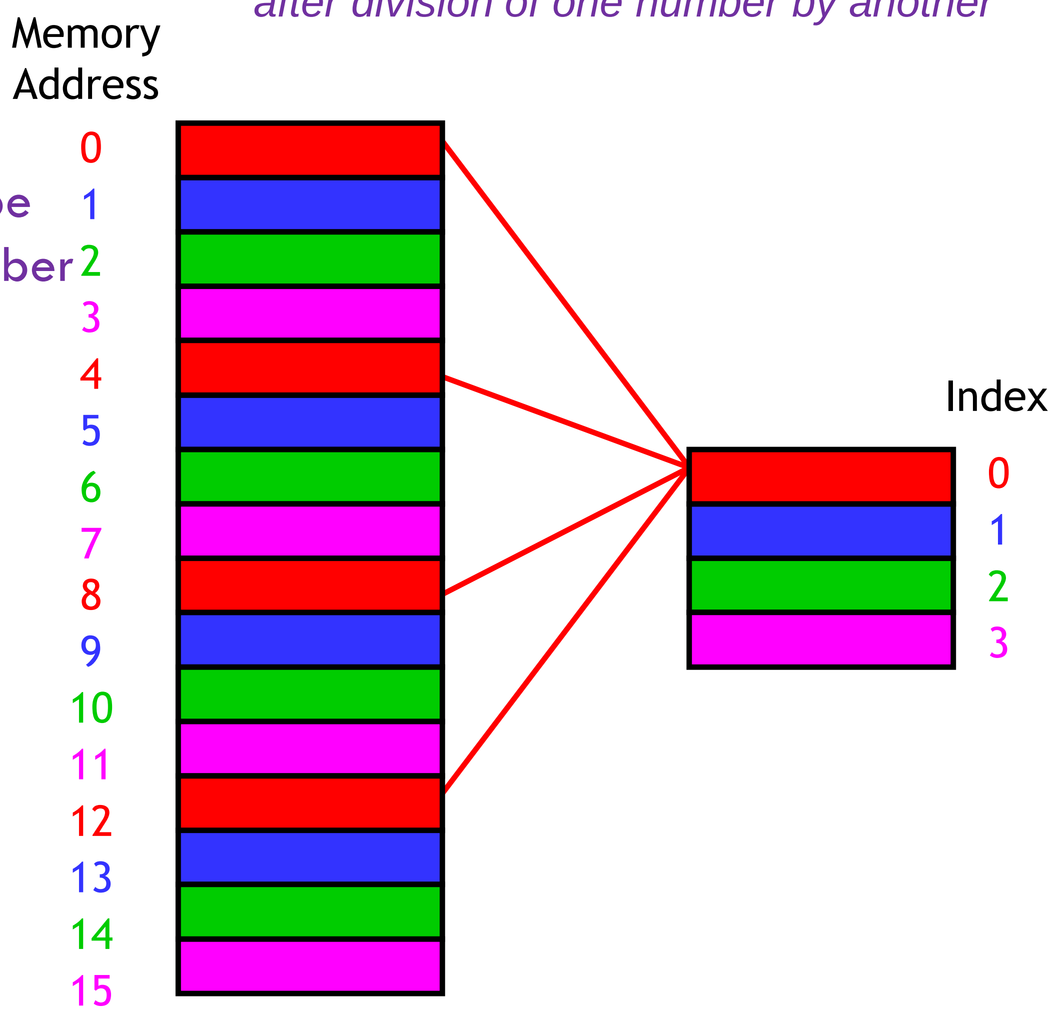
- Let x be block number in cache, y be block number of DDR, and n be number of blocks in cache, then mapping is done with the help of the equation

  $$x = y \bmod n$$

- 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



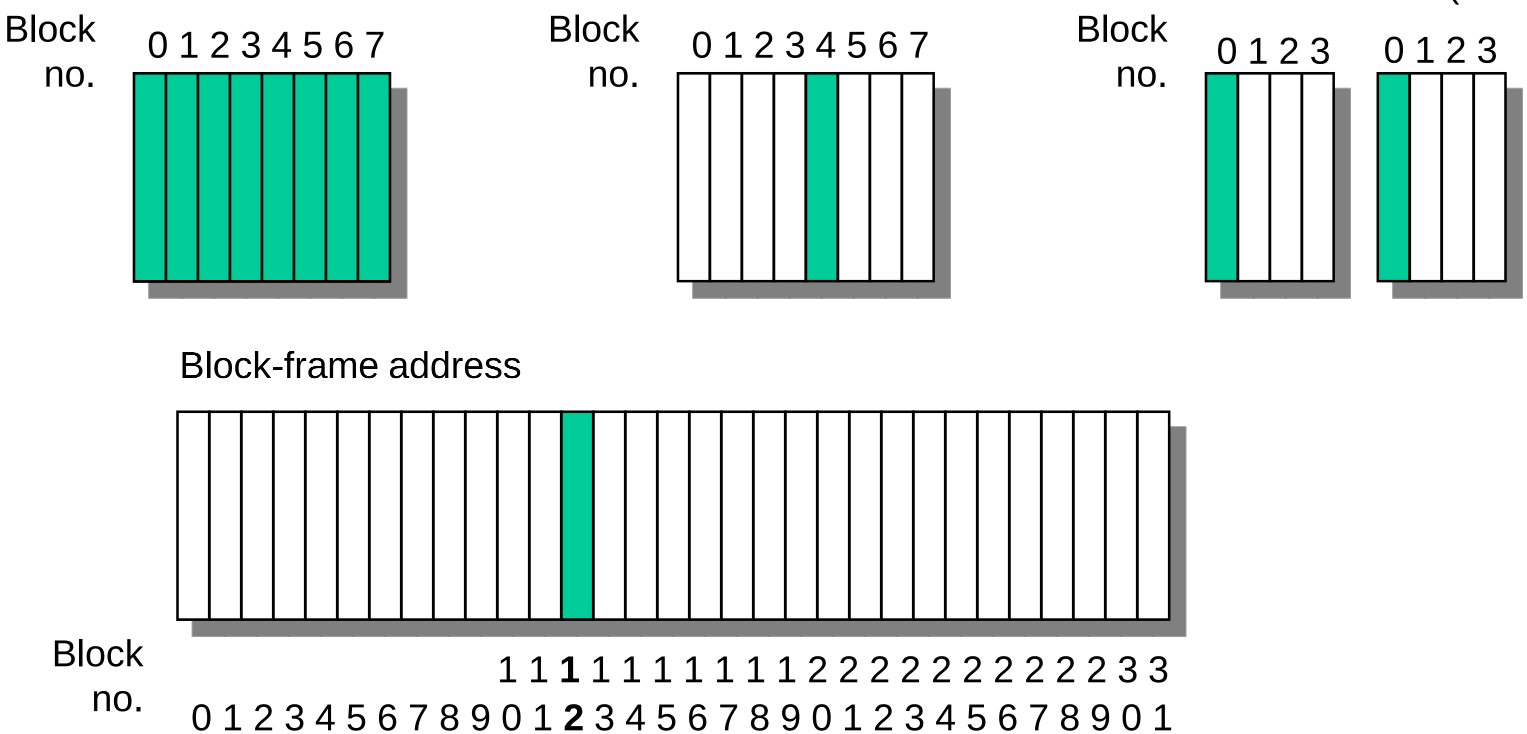
Modern cache memories are Associative Caches

- Block 12 placed in 8 block cache:

Fully associative: block 12 can go anywhere

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

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)



Further Reading

- Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65-76. DOI=10.1145/1498765.1498785, available at https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf
- *[for cache memories]* 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