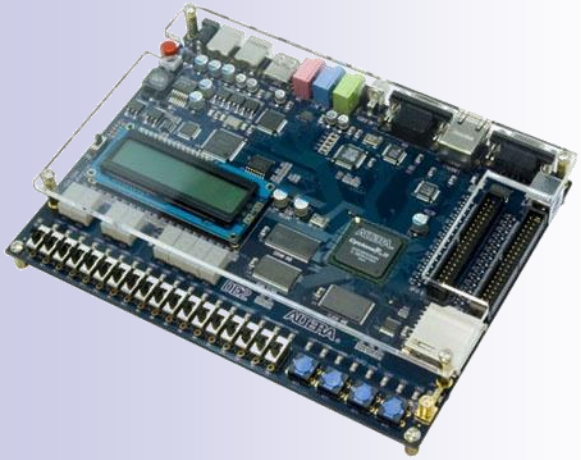
Logic Systems and Processors
cz:Logické systémy a procesory



Lecturer: Richard Šusta

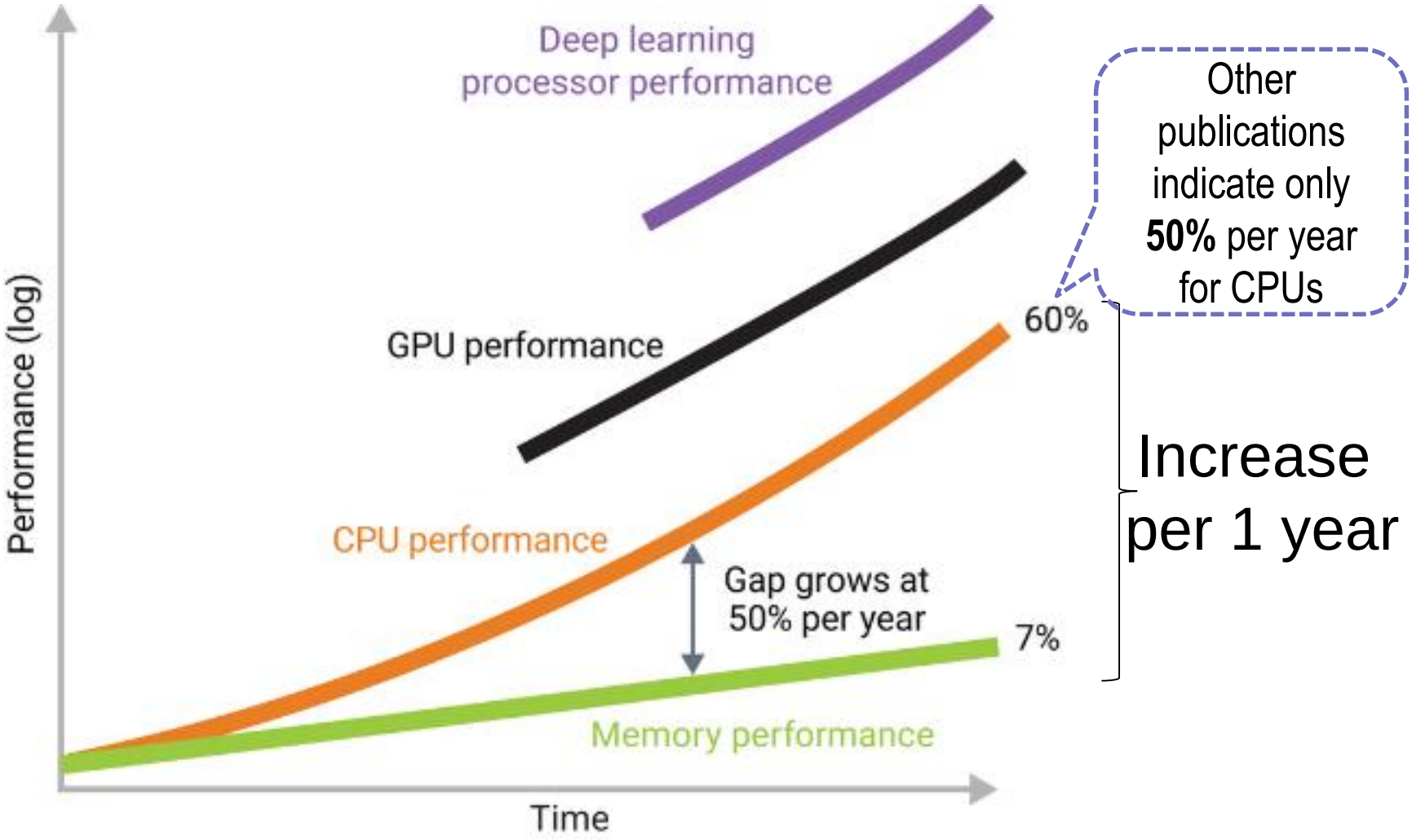
richard@susta.cz, susta@fel.cvut.cz,
+420 2 2435 **7359**

Version: 1.1 - corrections of typos

DRAM Memories

in modern computers

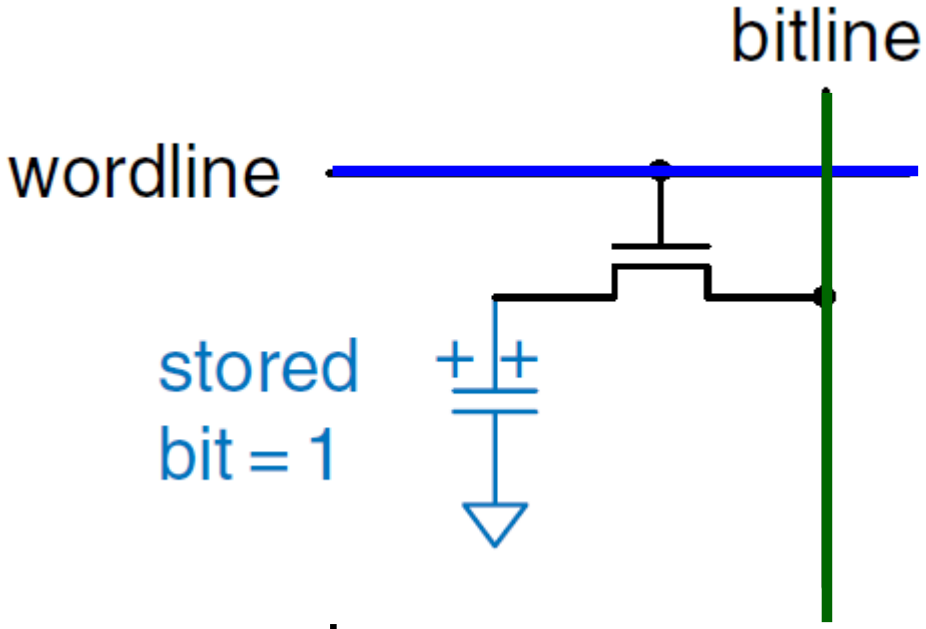
Processor-Memory Performance Gap



Source: medium.com

Detail of dynamic memory cell

Single-transistor
dynamic memory cell



NMOS transistor connects capacitor
to the "bitline" when the "wordline" wire is active.

Commercial DRAM parameters	
	Capacity fF [femtofarad]
Storage capacity	from 10 fF to 50 fF
Bit line capacity	around 2 fF

[Source: l'INSA de Toulouse]

fF - femtofarad

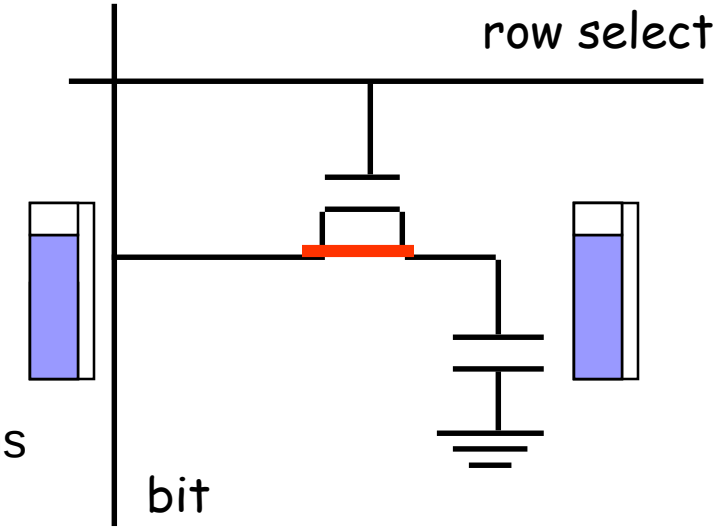
fF is an SI unit equal to 10^{-15} farads.

10^{-6} F = **1 µF** = 10^{3} nF = 10^{6} pF = 10^{9} fF

~9 fF capacitance is created in the vacuum between the plates 1 mm^2 spaced 1 mm apart,

DRAM - Dynamic RAM

- Write:
  - 1. Drive bit line
  - 2. Select row
- Read:
  - 1. Precharge bit line to Vcc/2
  - 2. Select row
  - 3. Cell and bit line share charges
    small voltage changes on the bit line
  - 4. Fancy sense amp
    must detect changes of ~1 million electrons
  - **5. Write:** restore the value
- Refresh
  - Just do a dummy read to every cell.



Each reading must be followed by a writing data back

Complex read/write operations limit DRAM access time. In the **50 years** of DRAM development, the density of bits/mm2 has increased by more than a million times, but cell read times have been cut only in half.

[Source: Vijay Narayanan: FPGA technolgies]

DRAM organisation

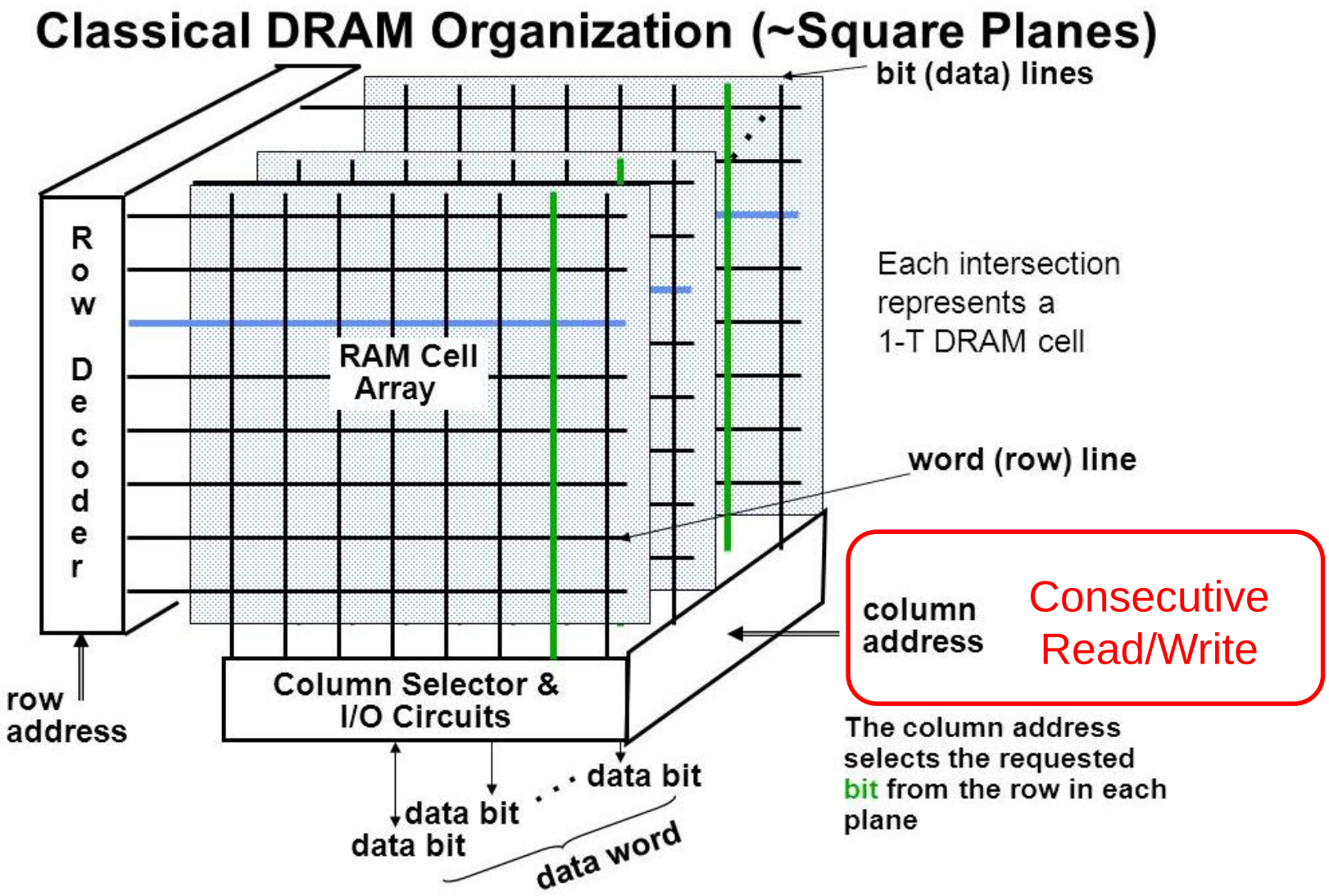
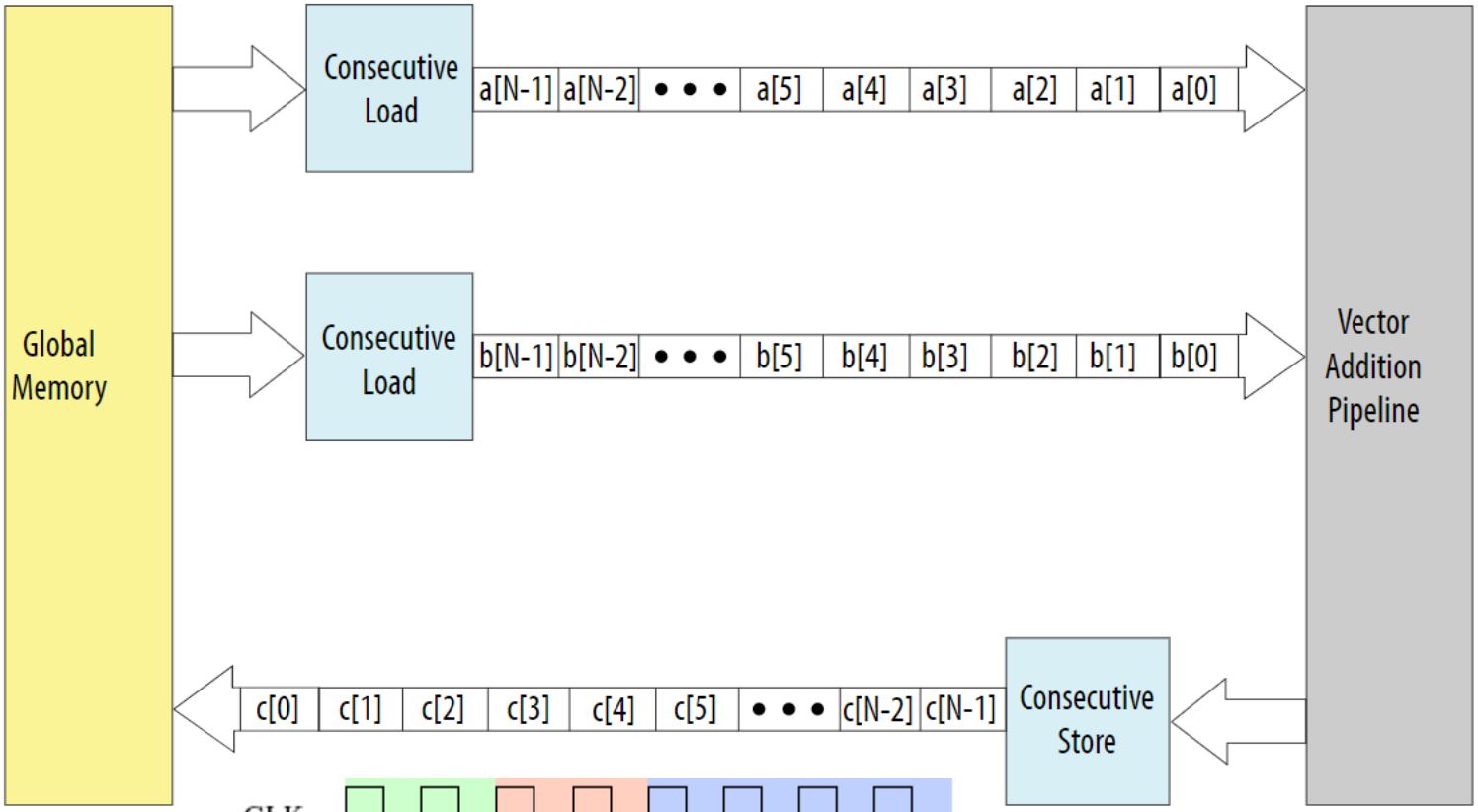
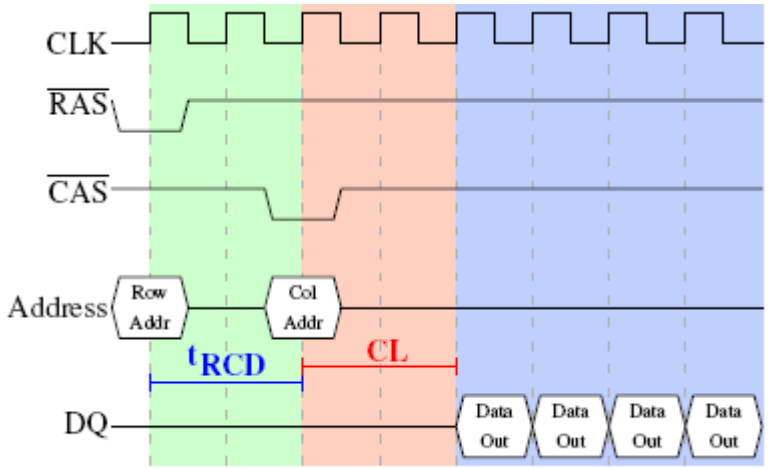


Image: Aleesha Owen

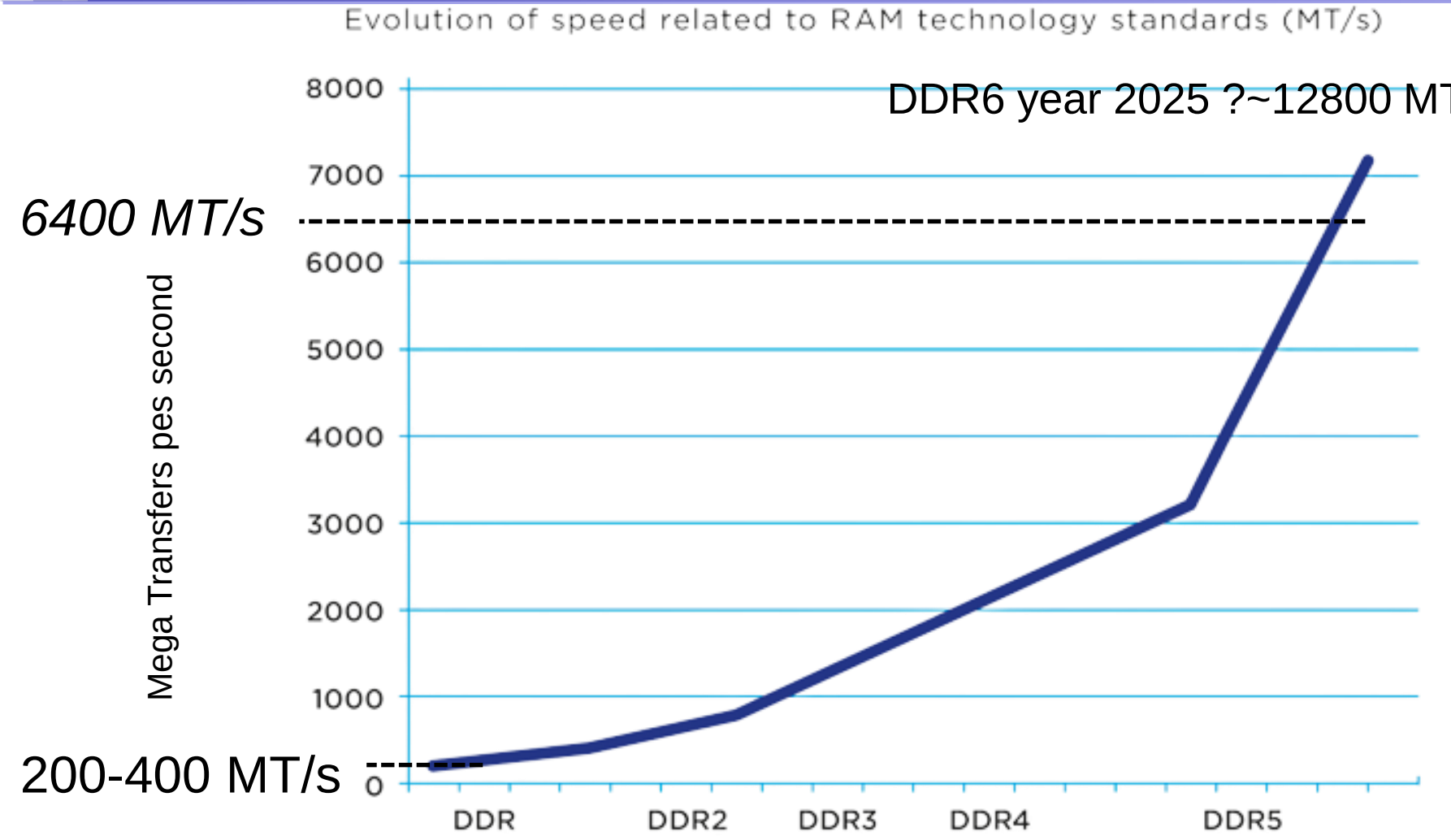
Consecutive Read/Write





RAS - Row Address Strobe,
CAS - Column Address Strobe

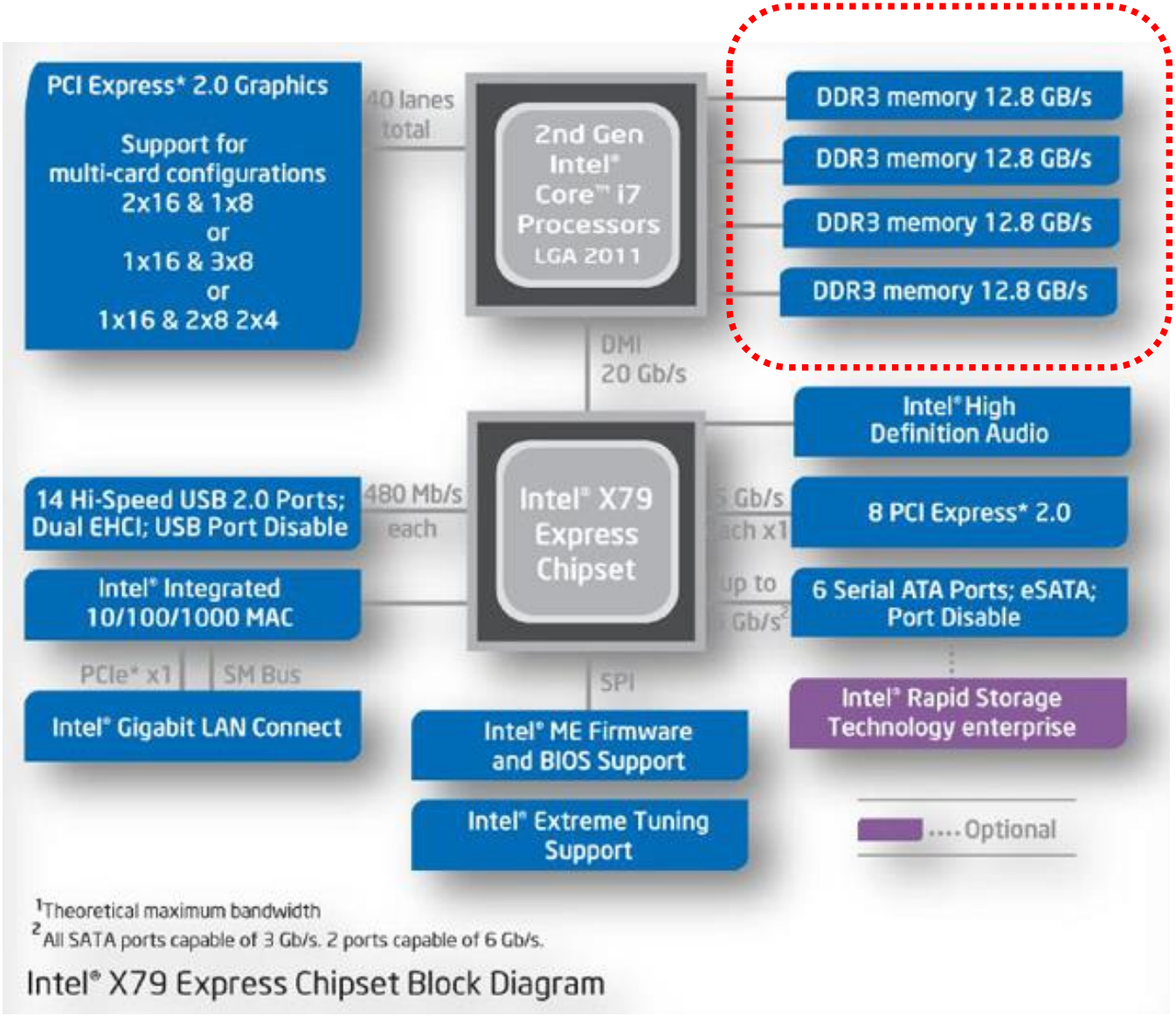
Evolution of DRAM read speed



Source : https://www.integralmemory.com/ddr5/

It is only about reading the sequence of consecutive data.
The access latency to the first item has not dropped below 25 to 35 ns.

Intel i3/5/7 Generation



*Motivational example

Let the single-cycle processor read instructions from fast SRAM and complete each register instruction in 1 clock cycle.

- At f=1 **GHz**, 10^9 pure register instructions take 10^9 *1 ns = **1 s.** The processor has a performance of **1000 MIPS** (Million Instructions Per Second).

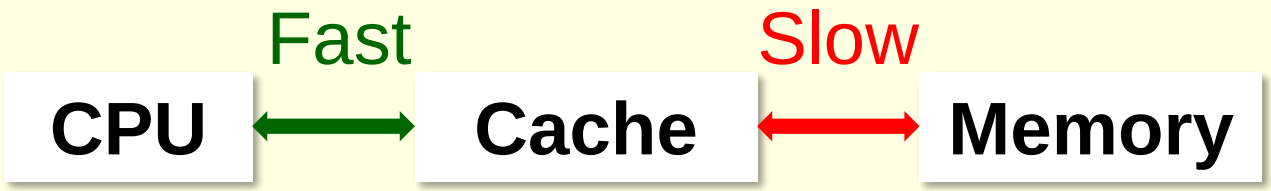
If the algorithm randomly accesses the external super-fast DRAM, i.e., non-sequentially, each read/write takes **30 ns**.

- If **20%** of the instructions work with DRAM, 10^9 instructions will take $8*10^8$ *1 ns+$2*10^8$ *30 ns= **6.8 s.**
  Performance drops from 1000 MIPS to 147 MIPS (=1000/6.8)

If we overclock the processor frequency to **4 GHz**, then 10^9 register instructions will take 10^9 * 0.25 ns = **0.25 s => 4000 MIPS.**

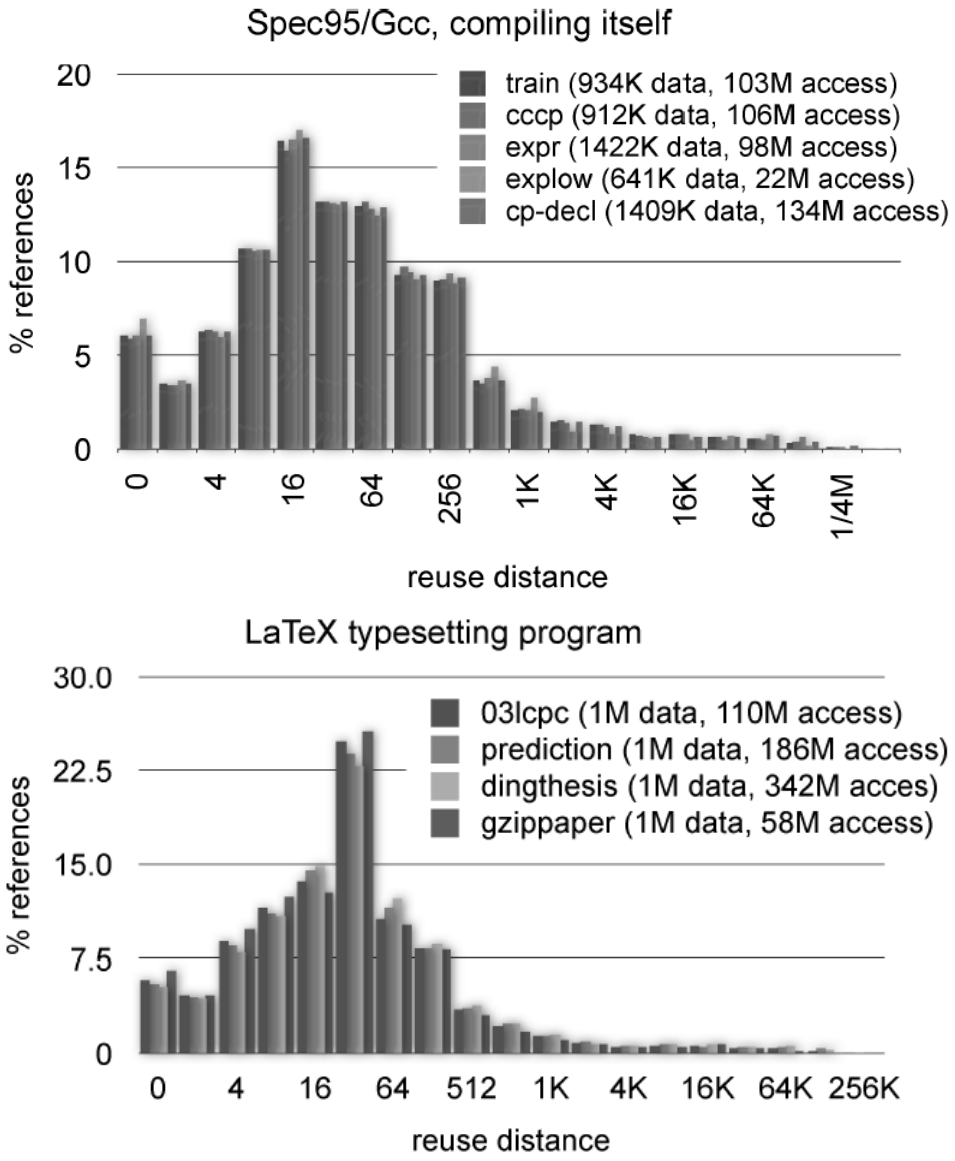
- However, if 20% of the instructions work with DRAM, then 10^9 instructions consume $8*10^8$ *0.25 ns+$2*10^8$ *30 ns = **6.2 s.**
  The overclocking increased the performance **from 147 MIPS to 161.3 MIPS.**

By quadrupling the clocks, only a **9.7%** increase in power was achieved **(**6.8/6.2=1.0967...).



Cache

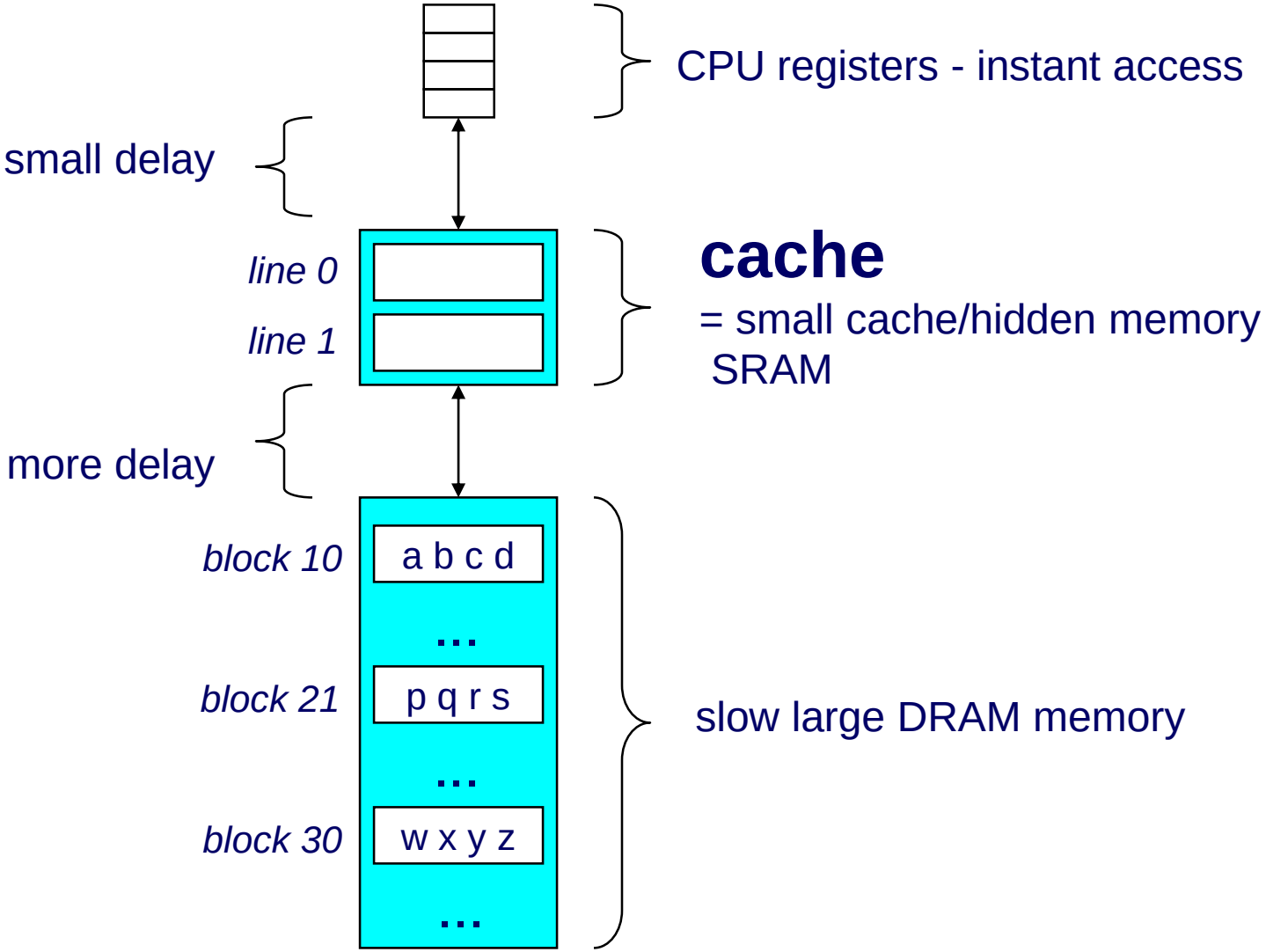
Repeatedly used variables in selected programs



The graphs show the distance between the same variable used last and again in a program.

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. **Program locality analysis using reuse distance.** *ACM Trans. Program. Lang. Syst. 31, 6, Article 20 (August 2009), 39 pg., available at:* https://dl.acm.org/doi/10.1145/1552309.1552310

Hidden memory - cache



Cache

Trapper's cache, Mackenzie County, Alberta by Provincial Archives of Alberta

Pronunciation:* /kæʃ/ kash, (cz: keš);

Etymology: from French Canadian trappers' slang word derived from French *cacher* = to hide, conceal = hiding place; especially used by settlers, explorers, or campers for concealing and preserving provisions or implements.

[*https://www.merriam-webster.com/dictionary/*]

Cache

Processors perform operations with registers much faster than the access time to large-scale DRAM. Although SRAM memories match processors' speed, using them as the main memory is not economical.
However, we can build from them a small block of fast memory called a *cache* and insert it between the main memory and the processor. When generating a memory request, the data is first searched in the cache memory; if it is not there, the search continues to the main memory.

Cache

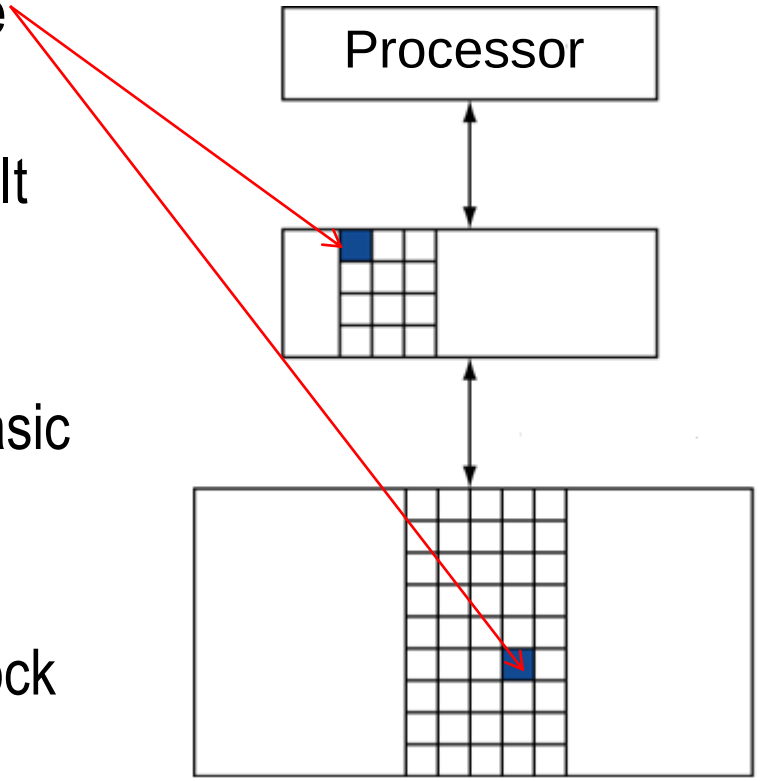
Hidden memory - a fast-access buffer used in computing.

- It belongs to two subsystems with different speeds, in which it balances different access times to information.
- **Speed up** access to reusable *data*, by storing copies of the data, or by delaying the writing of the data.

History: Cache memories first appeared in research computers in the early 1960s and a few years later in commercial computers. Today, almost every processor has them, from servers to small, low-cost processors.

Terminology around hidden memory

- **Cache hit** naming the situation when the requested value is read from the hidden memory (cache).
- **Cache miss**, opposite of cache hit. It must have read from the main memory.
- **Cache line** or **Cache block** - the basic copy able unit between hierarchical levels.
- In practice, the size of the cache block (rows) ranges from 8 bytes to 1 kilobyte, typically 64 bytes.



Terminology around hidden memory II.

- **Hit Rate** - the proportion of memory accesses that were successful (found their data) when accessing the cache.
- **Miss Rate** - similarly for the failed approach.
- **Miss Penalty** - the time it takes to read the block (data) from the memory of the lower hierarchical level.
  MissPenalty - can be calculated recursively.
- **Average Memory Access Time (AMAT)**

  AMAT = HitTime + MissRate × MissPenalty

* Why do we deal with organizing a cache?

We demonstrate the reasons with the example of autonomous driving of cars

- Many AI tasks are based on Deep Neural Networks
- Neural network passage needs matrix multiplications

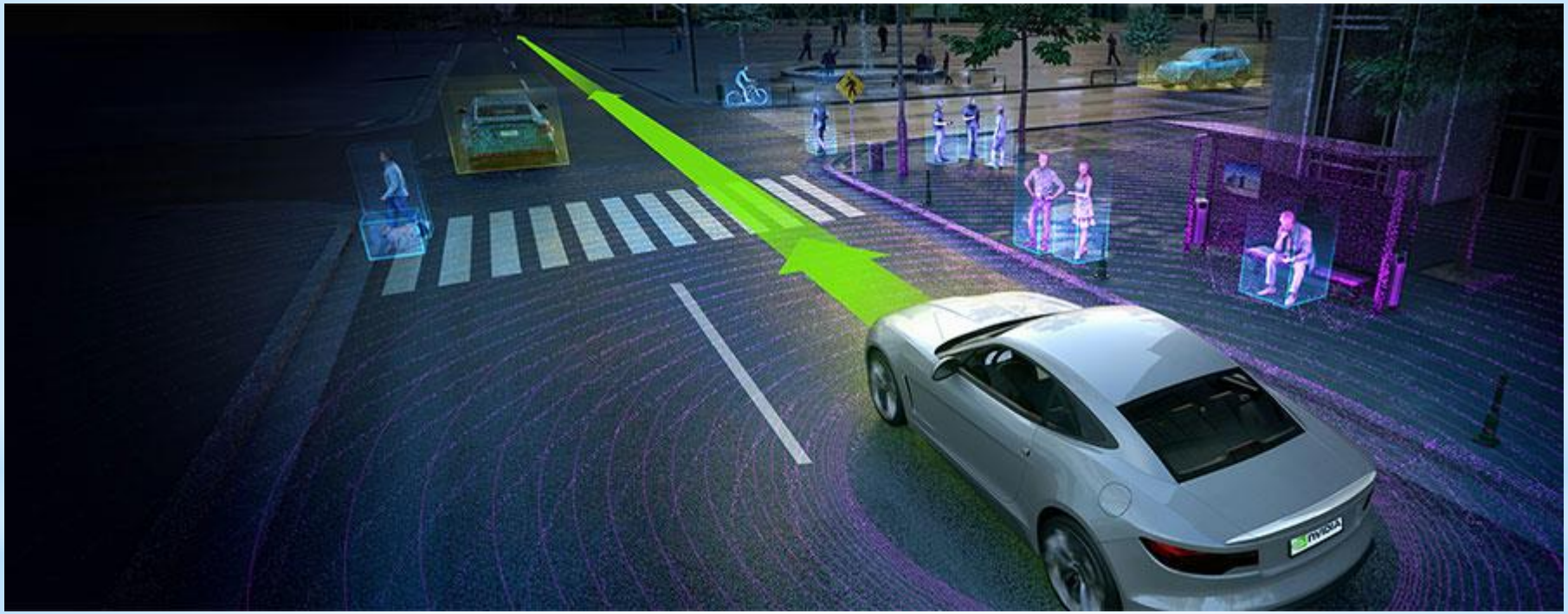


Image: http://www.nvidia.com/object/autonomous-cars.html

How to achieve acceleration in matrix multiplication?

The results of one of many experiments:

- Naive algorithm (3 × for-cycle) - **3.6 s** = 0.28 FPS

18.3 x

- Memory access optimization - **195 ms** = 5.13 FPS
  according to the knowledge of the structure of the used cache

1.7 x

- Four cores - **114 ms** = 8.77 FPS
  *(4*1 here is 1.7; the calculation is slowed down by their necessary synchronization)*

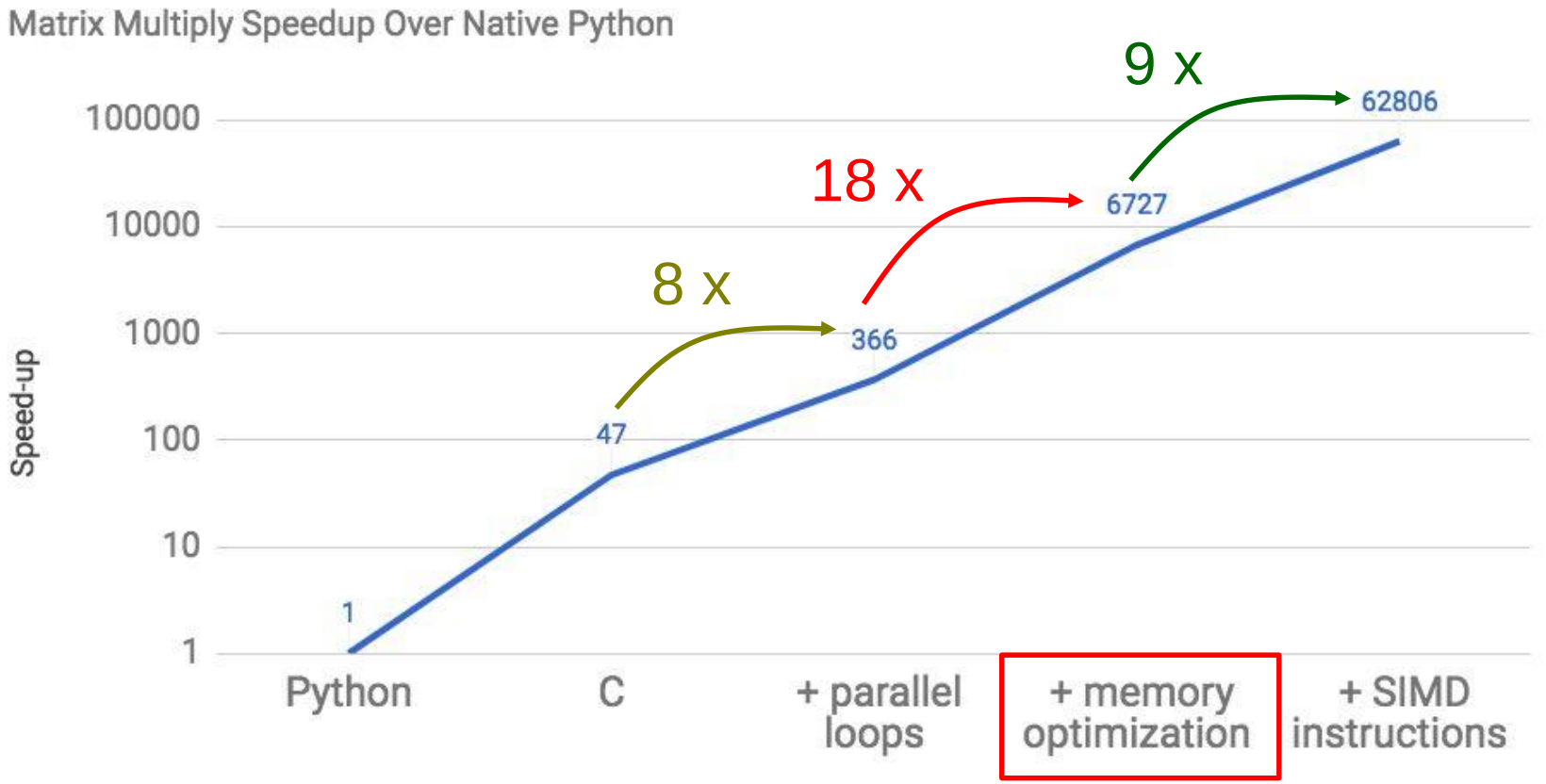
FPS = Frames Per Second

4.6 x

- GPU (256 cores) - **25 ms** = 40 FPS
  (256/4 is only 4.6 here because it delays the data transfer between CPU and processors)

- Naive algorithm - Eigen math library on i7-2520M,
  GPU based on measurements by Joel Matějka of the http://industrialinformatics.cz/ group, where they were working on operating system development as part of European projects and future software platforms for autonomous driving.

Other Attempt: Matrix Multiplication Acceleration



SIMD (Single Instruction on Multiple pieces of Data)
parallel multiplication instructions, today up to 8-member double vectors
or 16-member float vectors available in Intel processors since 2011

Example

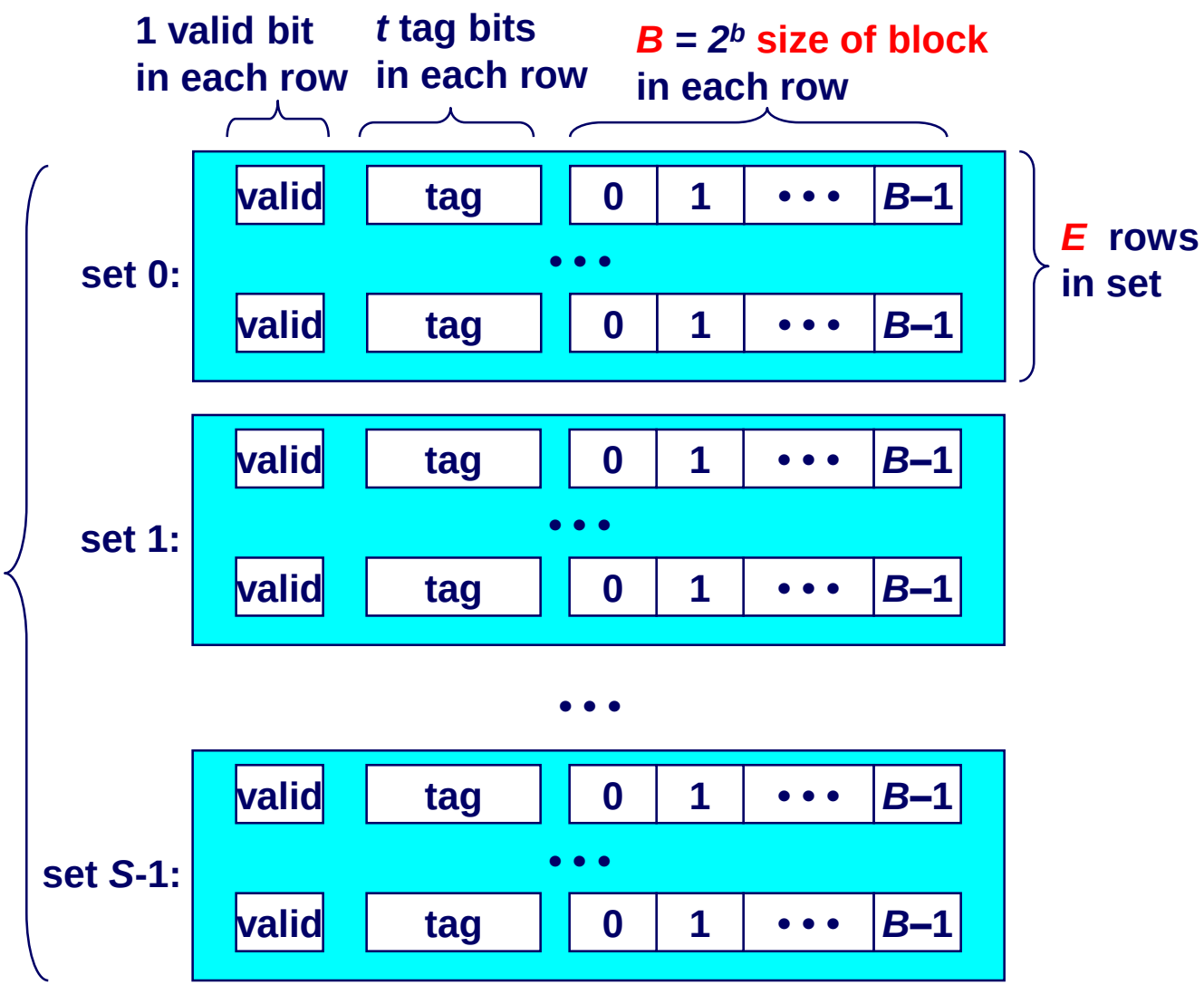
- Let's have a cache of 8 blocks. Where will the data from address 0xF0000014 be placed?
- It depends on the organization of the cache, which can be:
  - **Fully Associative Cache**
  - **Direct Mapped Cache**
  - With a limited degree of associativity,
    Set-Associative Cache
    for N=2 is a 2-way cache,
    2-Way Set-Associative Cache

General Organization of Cashe

Cache is array of **S** sets

Each set contains **E** rows (blocks) = Degree of Associability

Each block has **B** words/columns



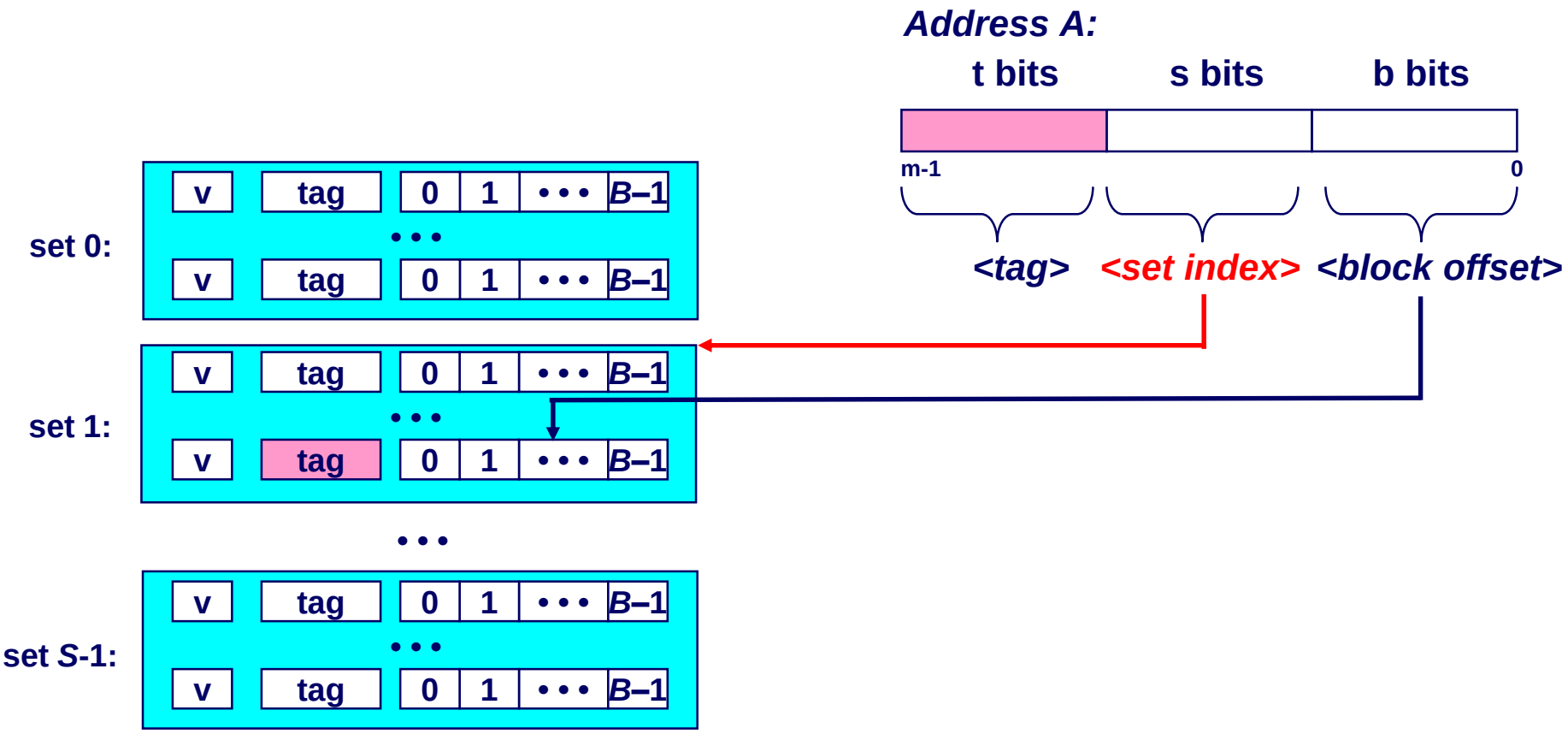
$S = 2^n$ sets
where n>=0

Set # ≡ hash code

Tag ≡ hash key

Size: $C = B \times E \times S$ words

Addressing of Cashe



Librerian analogy of cache organization

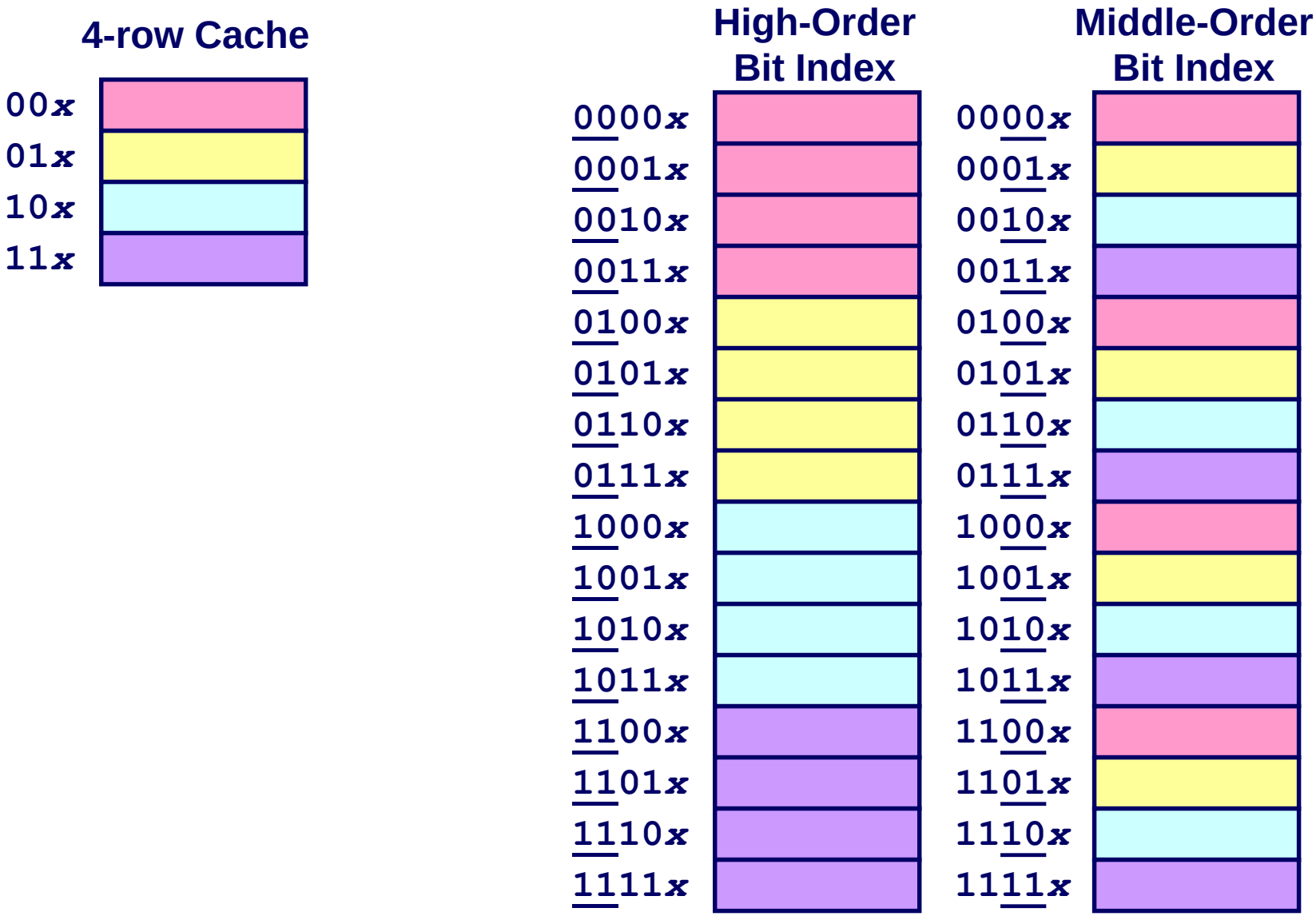
Cache organisation

- **Number of sets** = number of **study rooms** in the library;
- **Rows** = **number of tables** in each study room;
- **Block length** = **number of book slots** on each table. These must always be filled. If a table has room for four books, the delivery service must always be ordered to bring 4 consecutive volumes.

Search

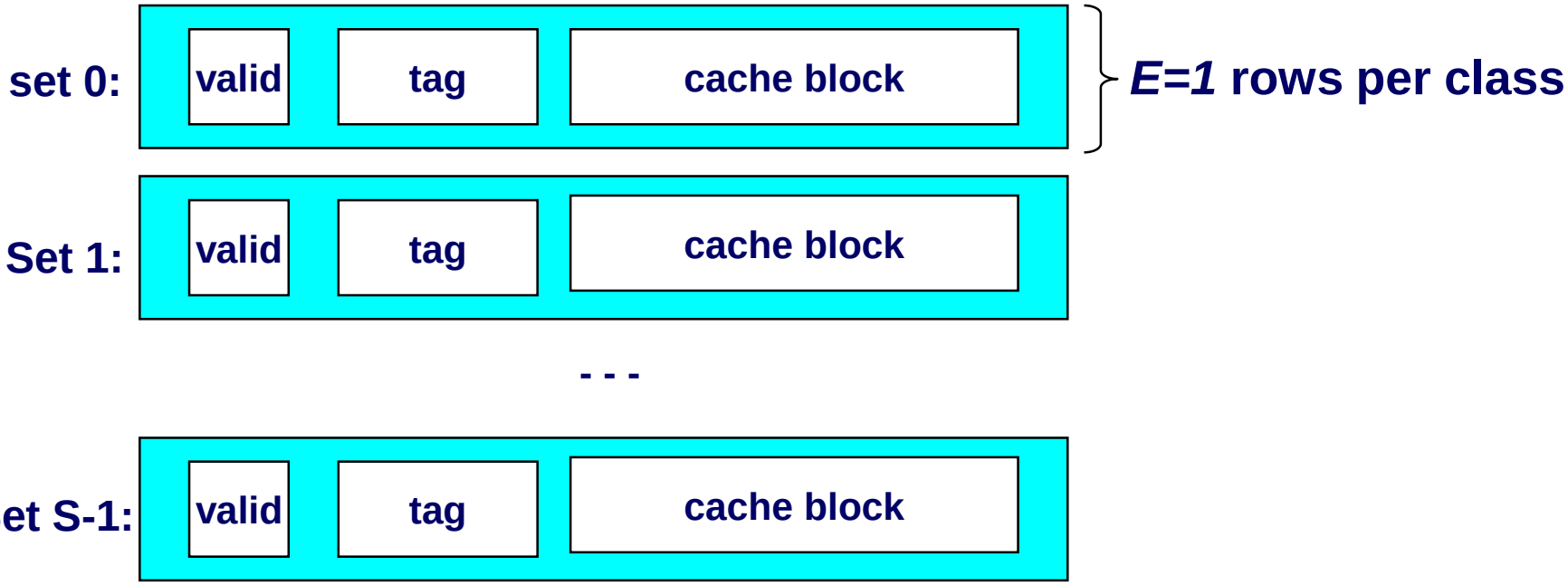
- **Tag** - information, or key, needed to determine whether the searched **block**, i.e. group of books, is on the given table.
- **Validity bit** - an additional field that specifies that the block and tag contain valid data. If the bit is FALSE, the block is not selected regardless of the data stored in it. It is considered invalid.

We are Middle Bits Used?



Direct-Mapped Cache

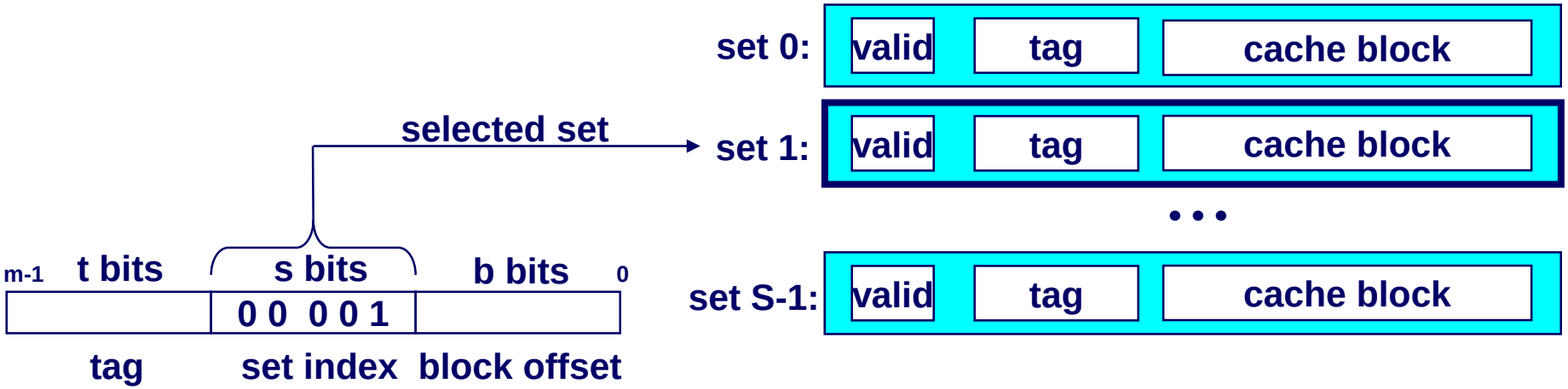
- 1 row (block) per 1 class (set)



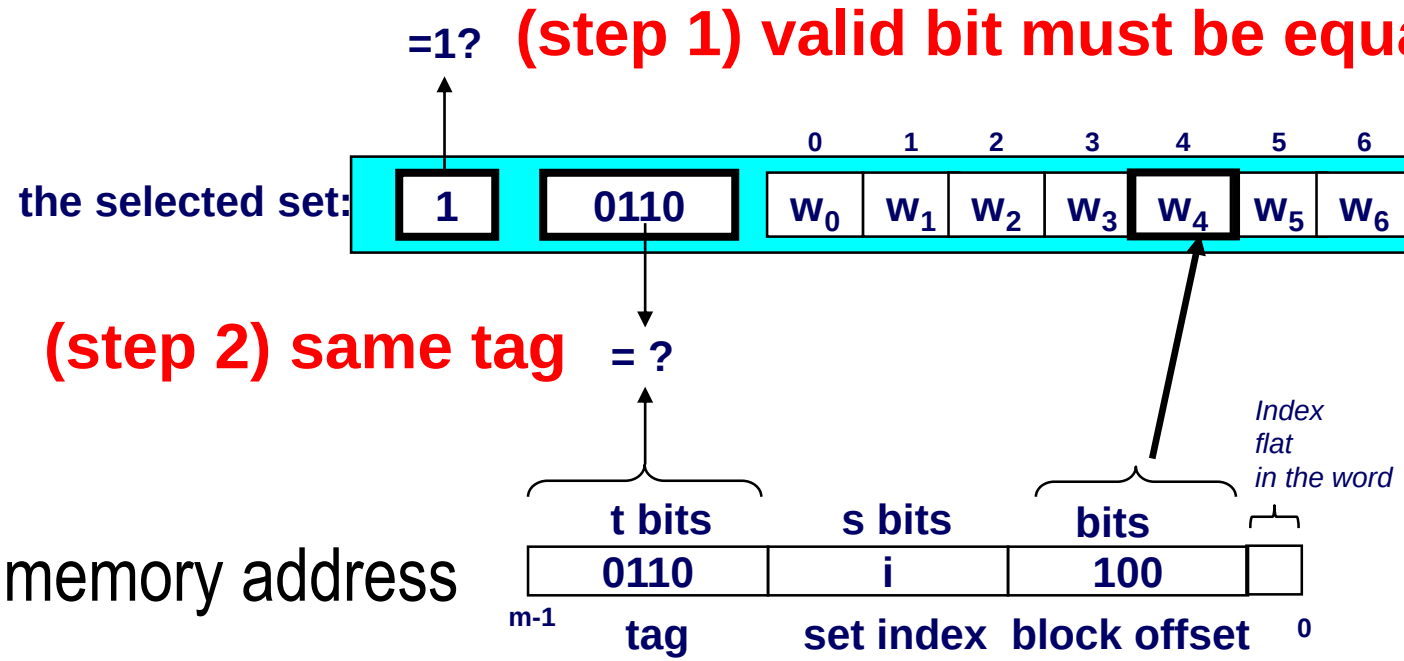
Accessing Direct-Mapped Caches

Set selection

- index of a set determines row, because 1 set has 1 row only in direct mapped cashes!



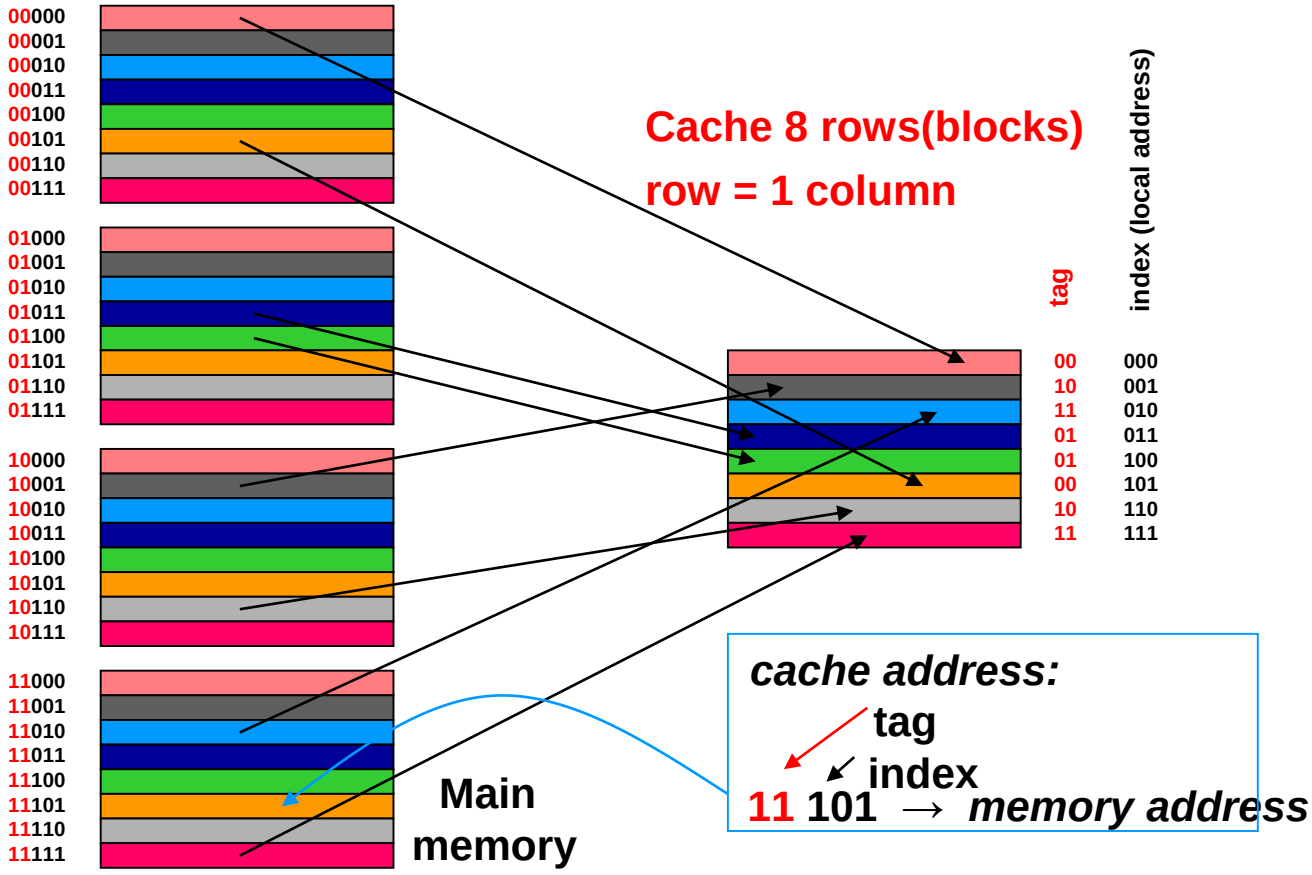
Access to Direct-Mapped Cache



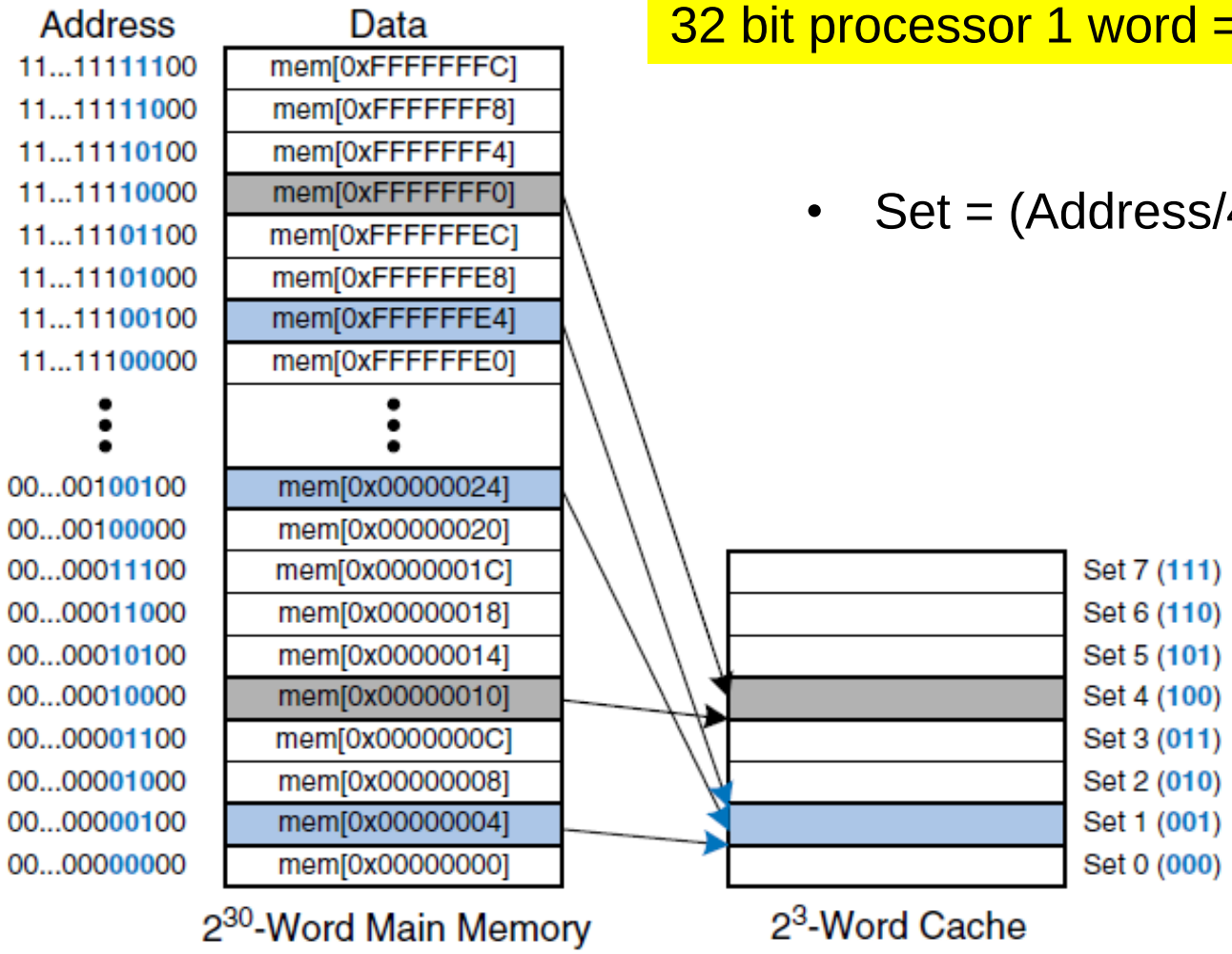
(Step 3) If conditions (1) and (2) are met,
then a "cache hit" occurs
and the word offset-em block is selected
otherwise a "cache miss" has occurred
and the entire line must be read from main memory

Example Direct-Mapped Cache in 8bits Processor

8bit processor with memory 32 bytes - 1 word = 1 byte



Example: Direct Mapped Cache

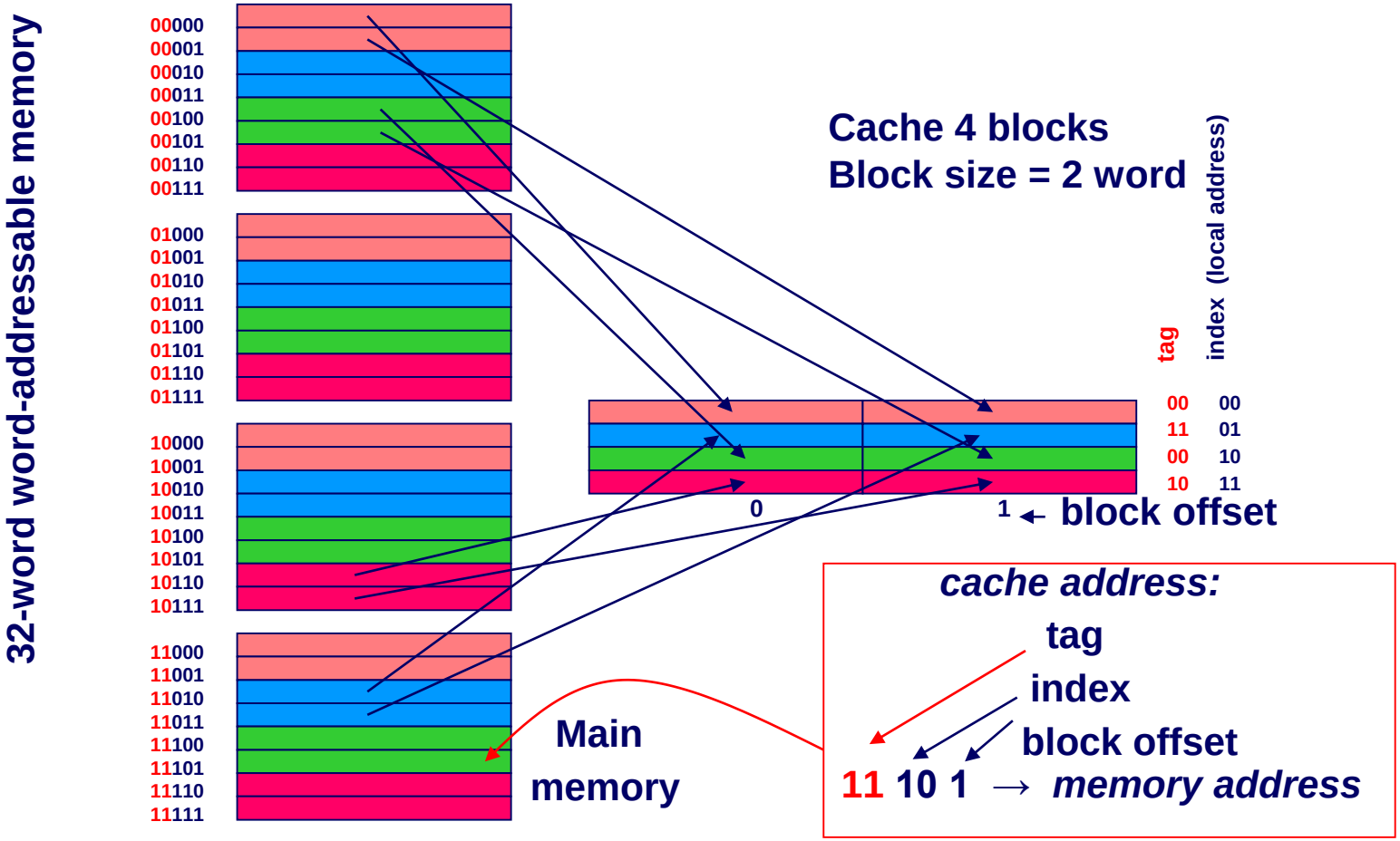


32 bit processor 1 word = 4 bytes

- Set = (Address/4) mod 8

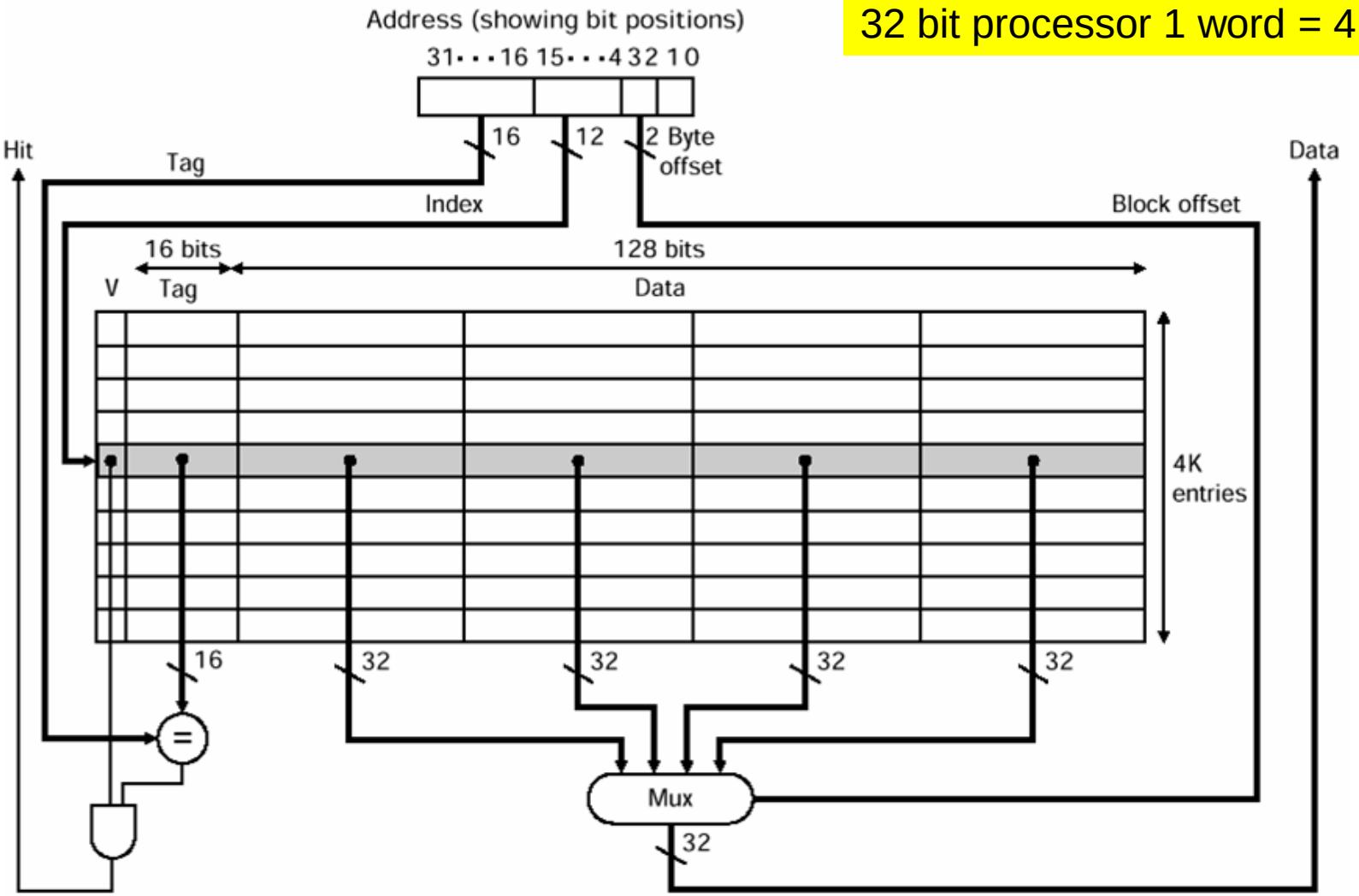
Direct-Mapped Cache - Block Size 2 Words

8bit processor with memory 32 bytes - 1 word = 1 byte



Direct mapped cache implementation

32 bit processor 1 word = 4 bytes



Example: Direct Mapped Cache

The following slides were created by

PSOE Dan Garcia

`www.cs.berkeley.edu/~ddgarcia`

and they were only slightly adapted.

Accessing data in a direct mapped cache

Ex.: 16KB of data, direct-mapped, 4 word blocks

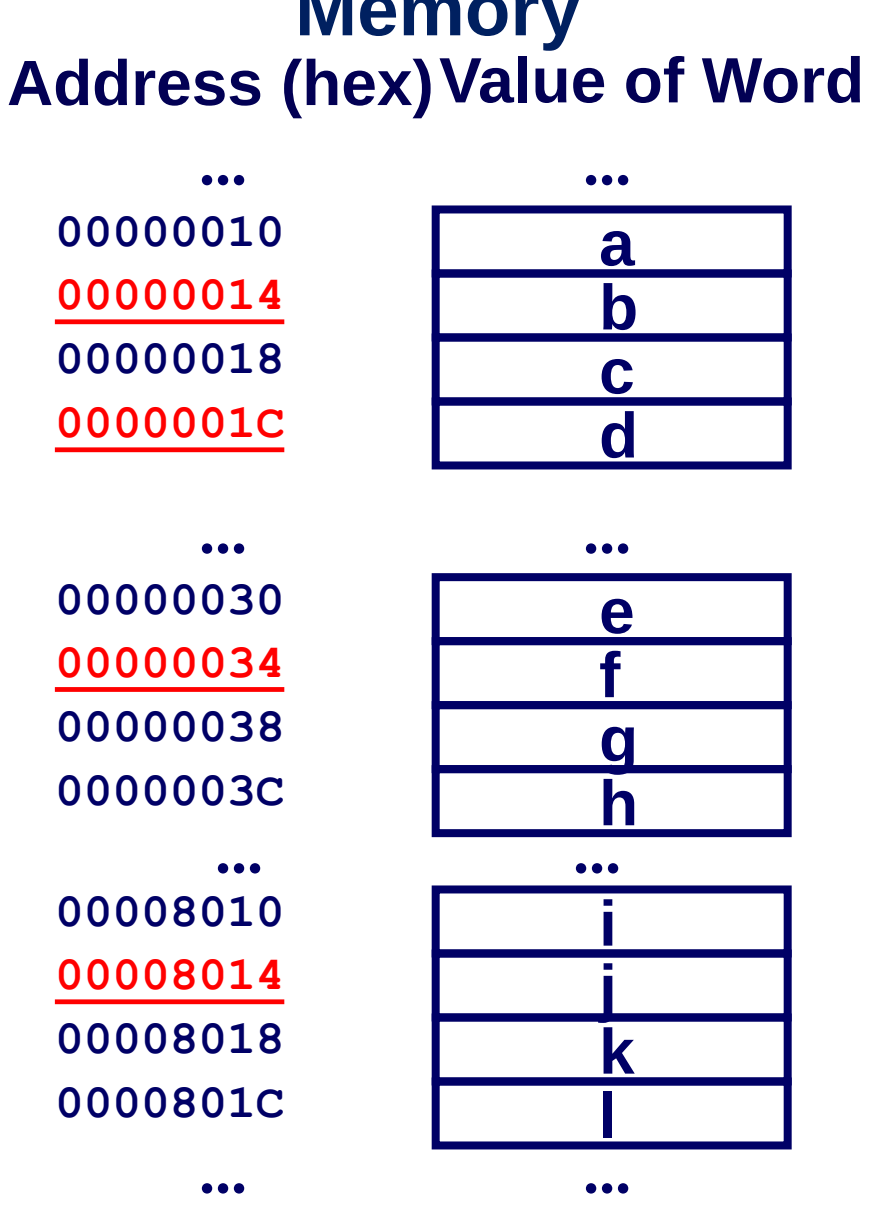
Read 4 addresses

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014

Memory values on right:

- The letters **a**, **b**..., and **l** - only indicate some values in main memory

32 bit processor 1 word = 4 bytes



Accessing data in a direct mapped cache

4 Addresses:

- `0x00000014, 0x0000001C,`
  `0x00000034, 0x00008014`

4 Addresses divided into
Tag | Index | Byte Offset fields

(they are separated below by spaces for convenience)

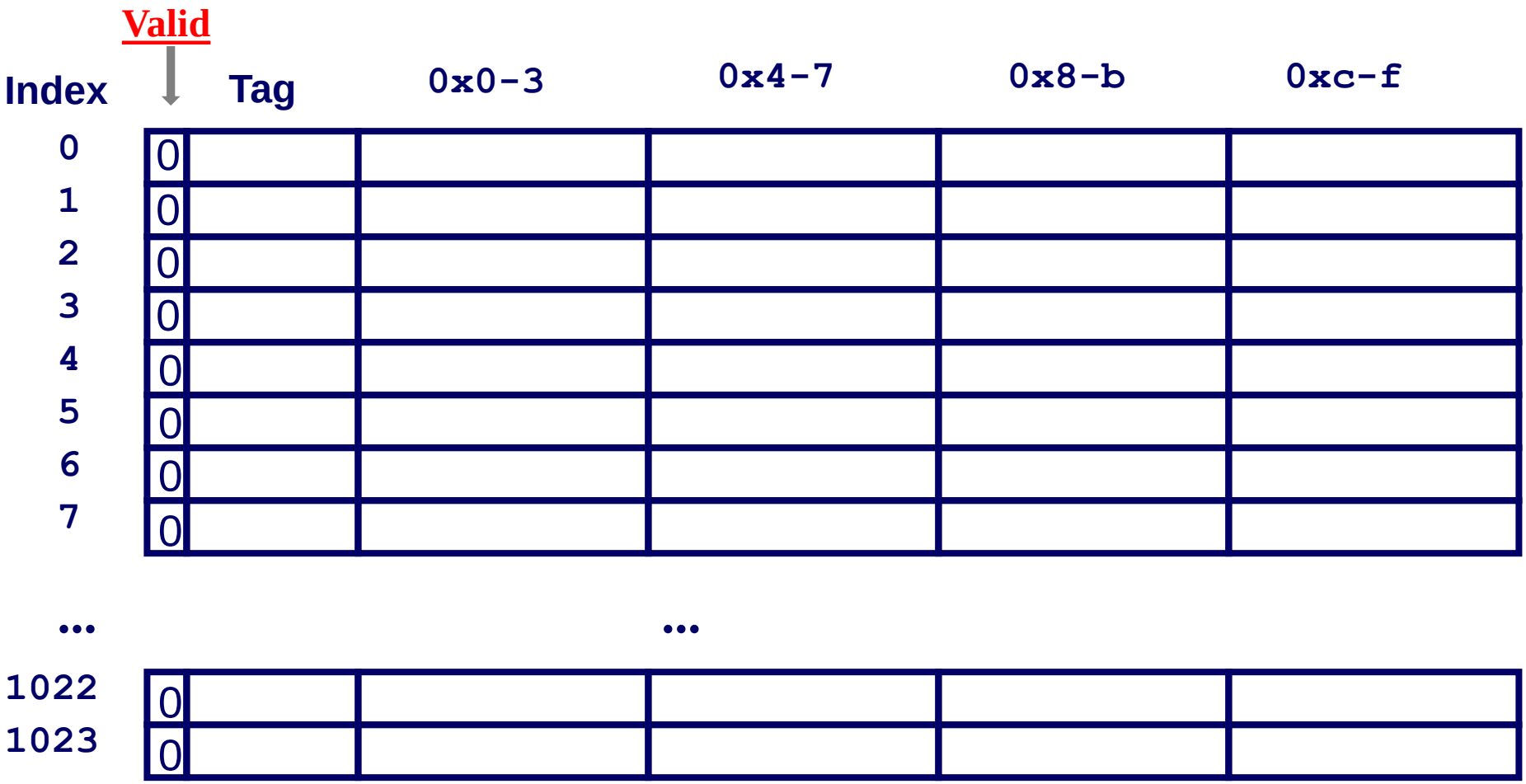
```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
```

Tag **Index** **Offset**

16 KB Direct Mapped Cache, 16B blocks

Valid bit: determines whether anything is stored in that row

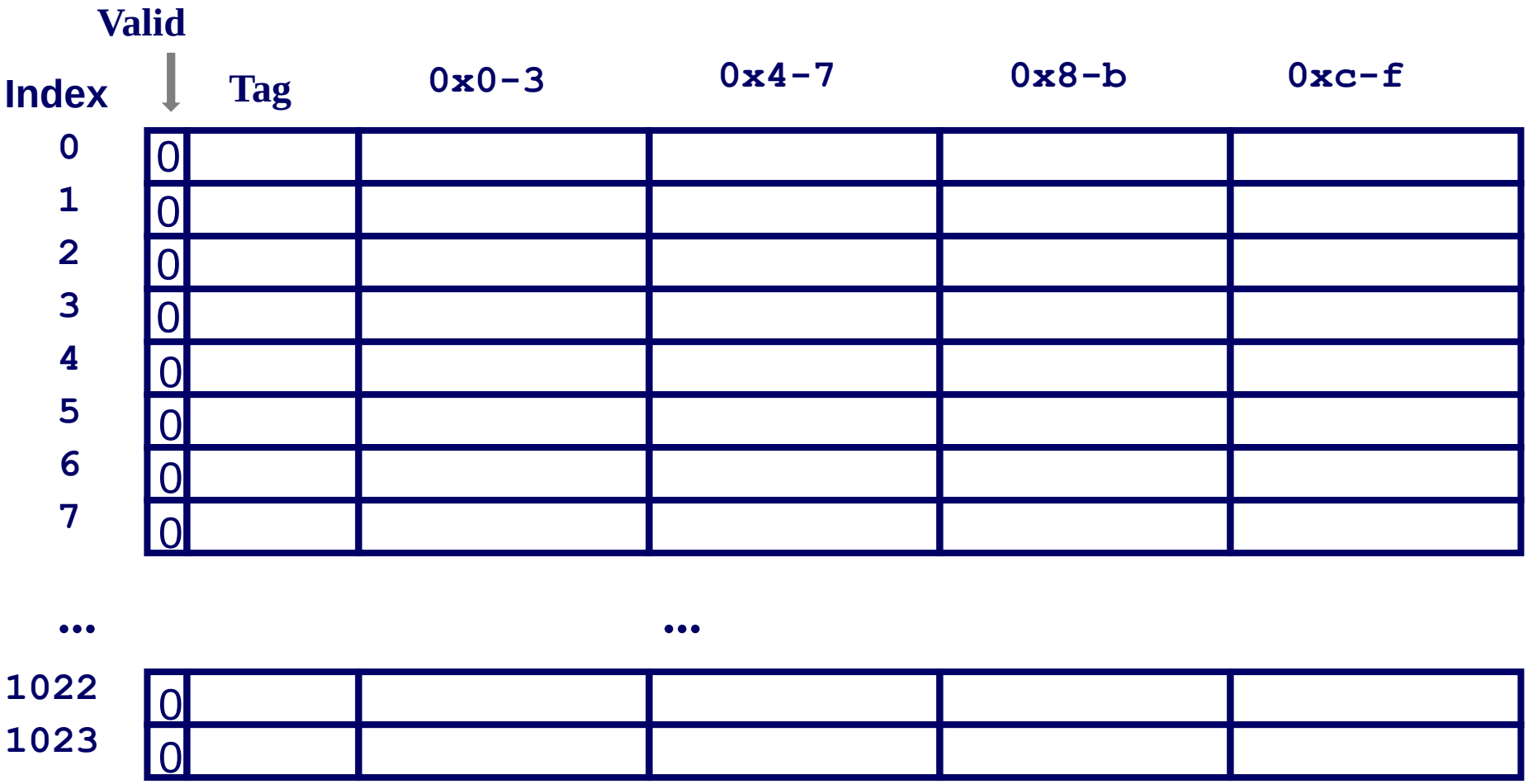
(when computer is initially turned on after power-up, all entries are invalid)



1. Read 0x00000014

0000000000000000 0000000001 0100

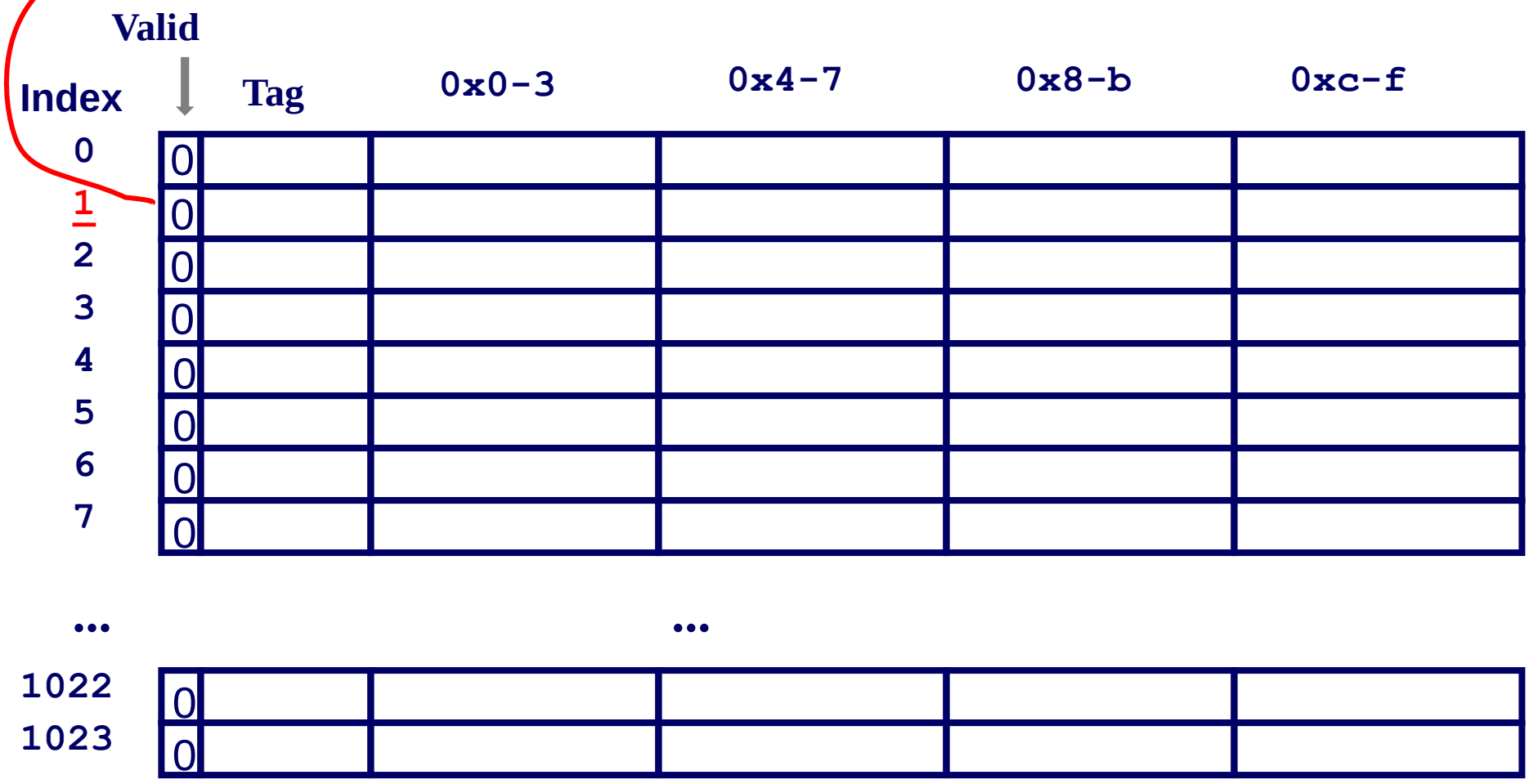
Tag field | Index field | Offset



So we read block 1 (0000000001)

0000000000000000 0000000001 0100

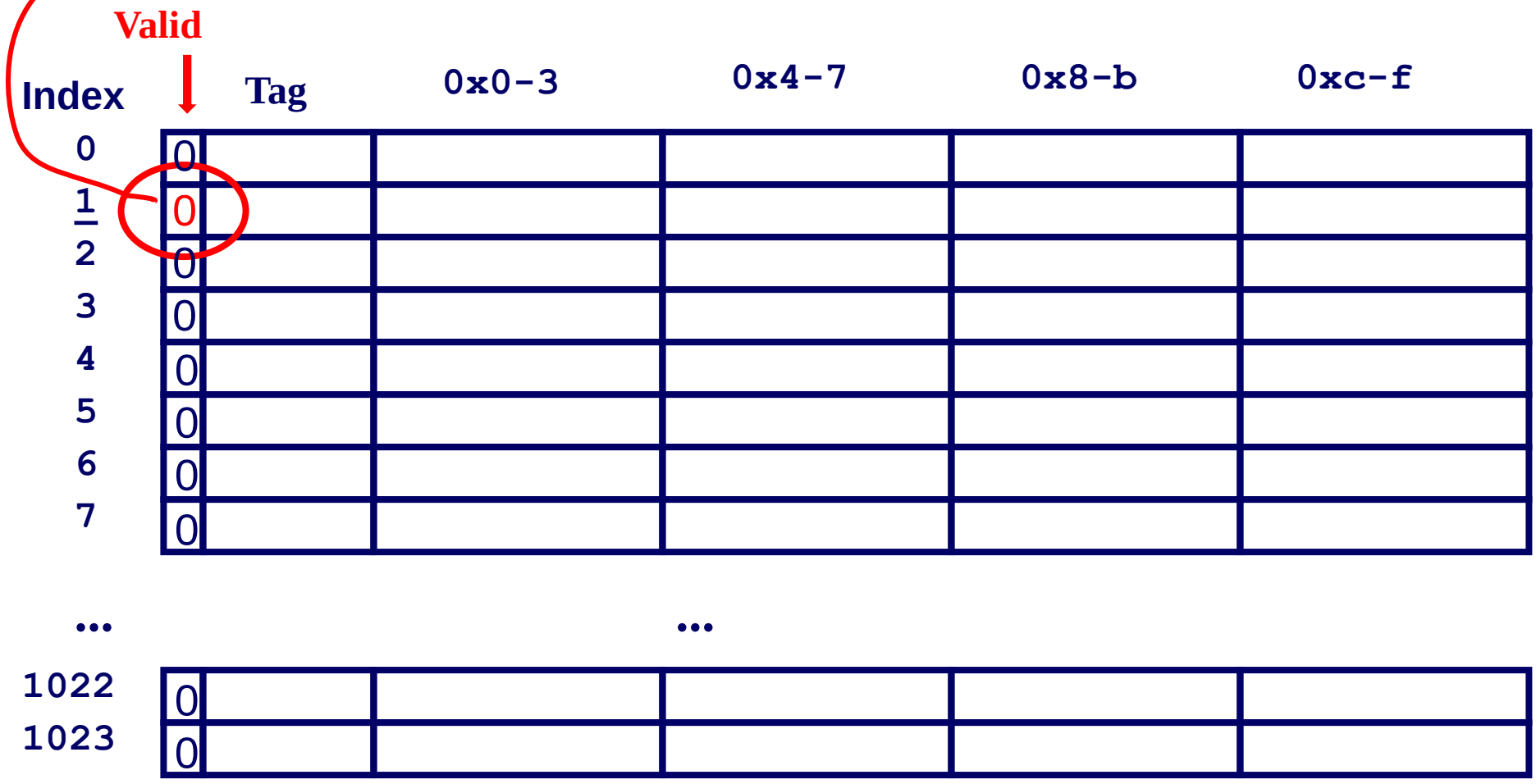
Tag field | Index field | Offset



No valid data !

00000000000000000000 0000000001 0100

Tag field | Index field | Offset



So load that data into cache, setting tag, and valid

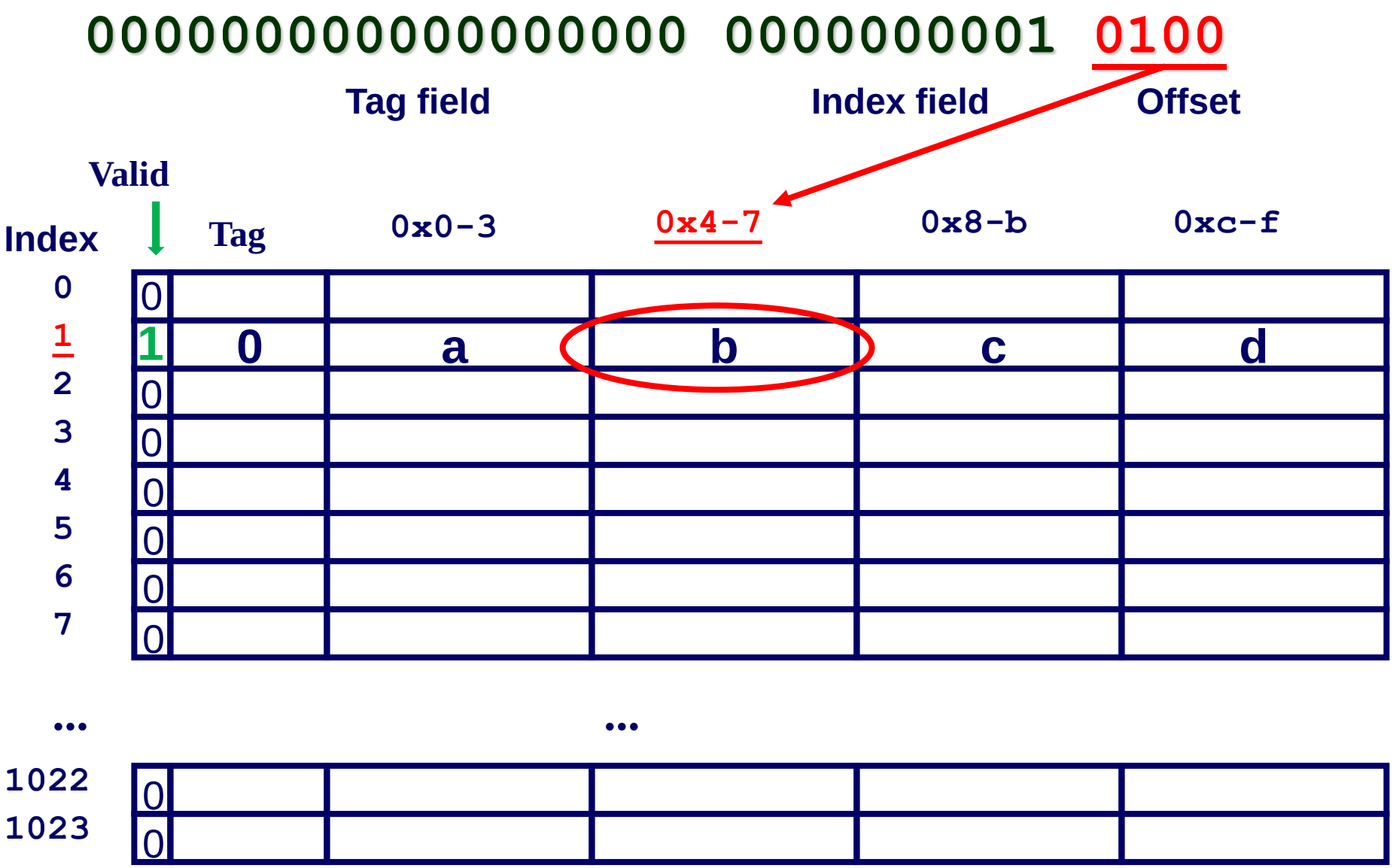
`00000000000000000000 0000000001 0100`

Tag field | Index field | Offset

All 4 words are read into the cache!

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

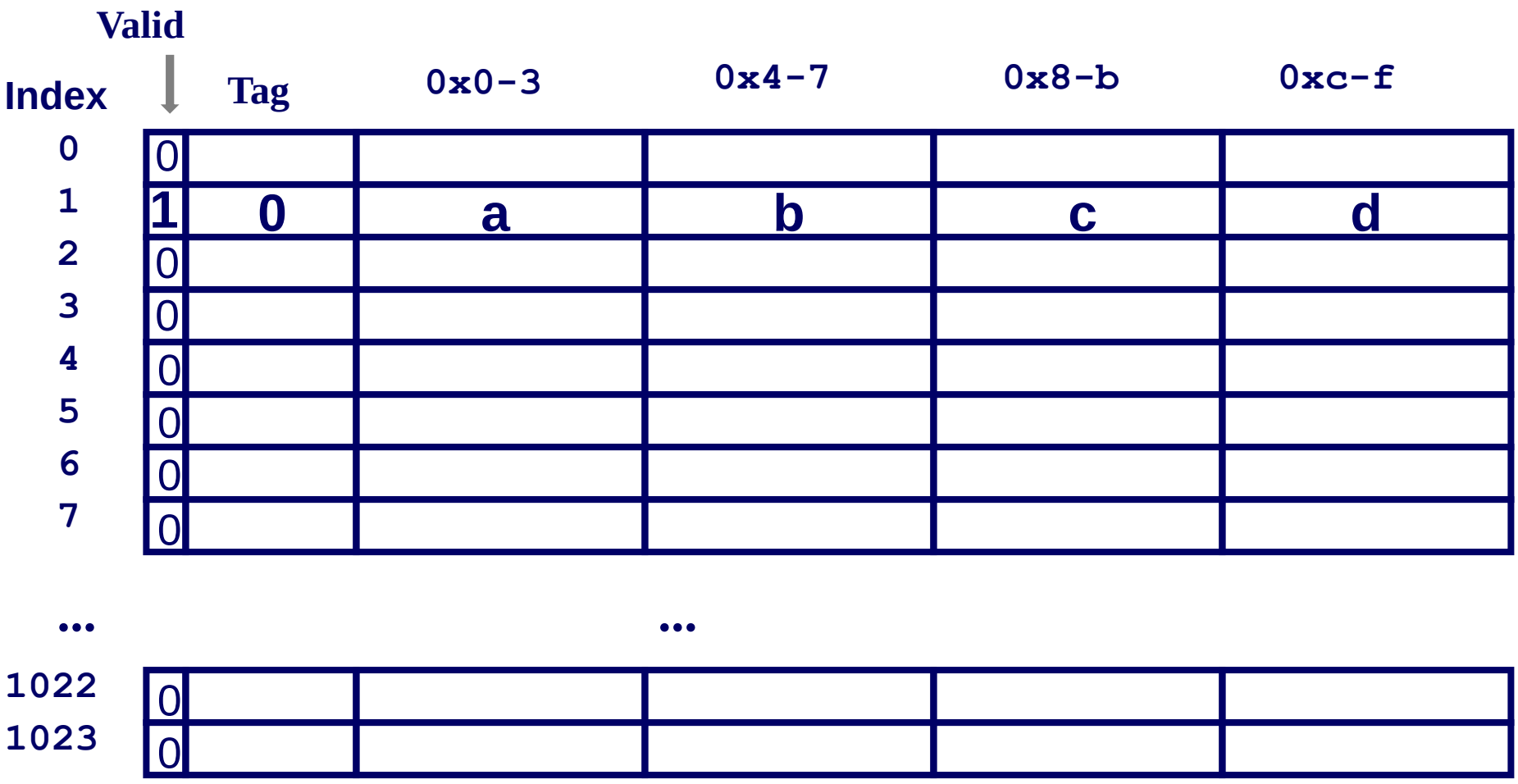
Read from cache at offset, return word b



2. Read 0x0000001C = 0…00 0..001 1100

0000000000000000 0000000001 1100

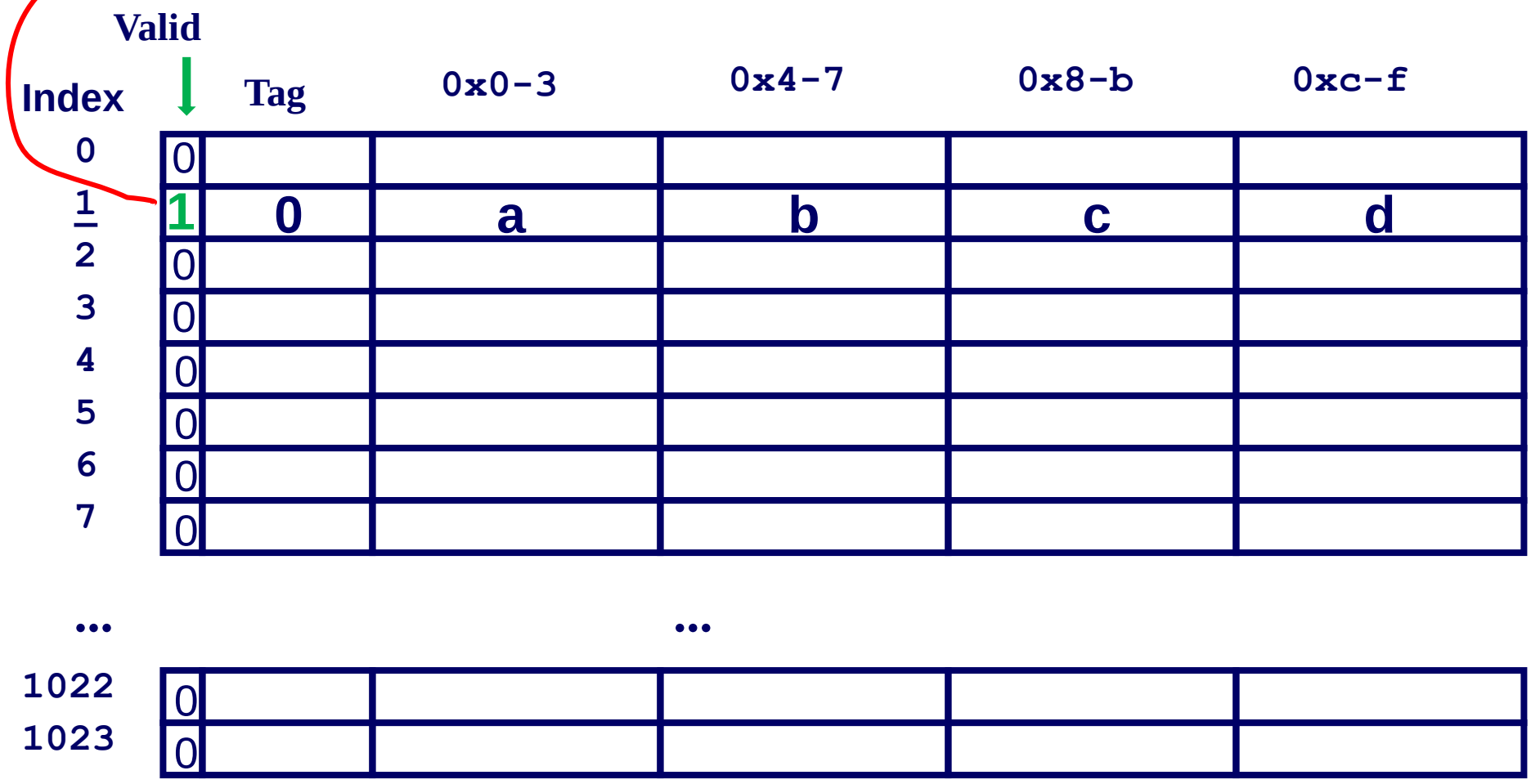
Tag field | Index field | Offset



Index is Valid !

0000000000000000 0000000001 1100

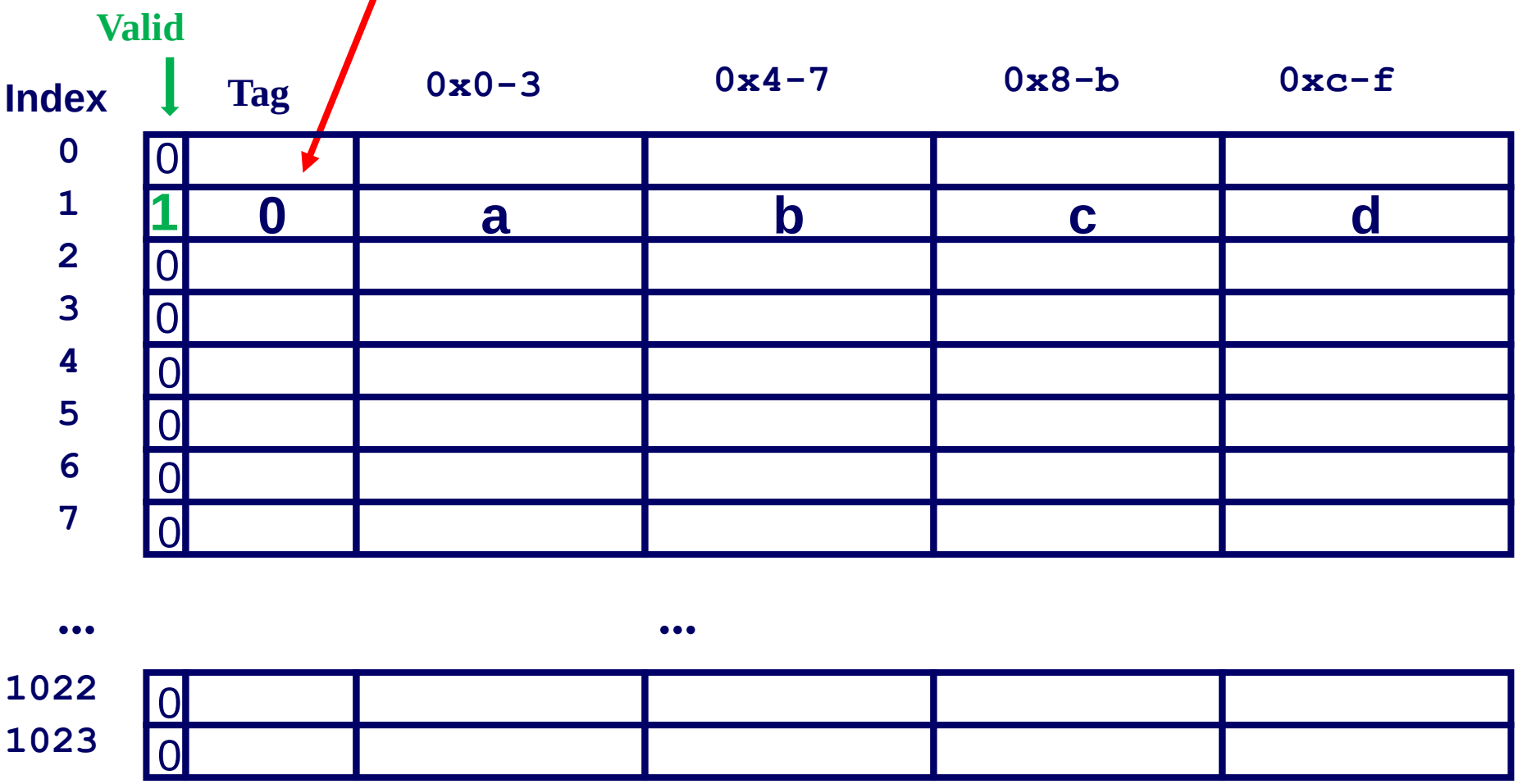
Tag field | Index field | Offset



Index valid, Tag Matches

`00000000000000000000 0000000001 1100`

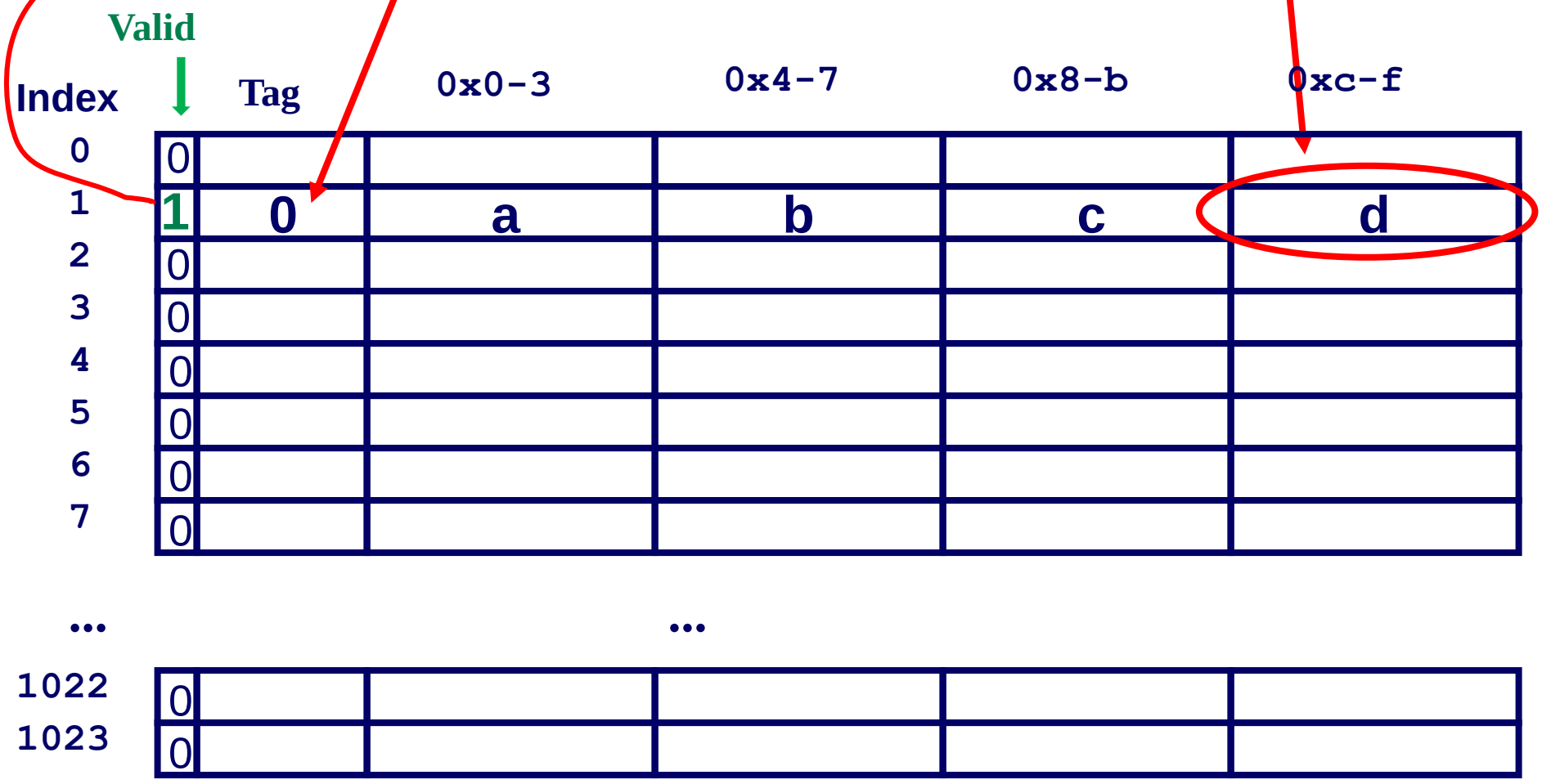
Tag field | Index field | Offset



Index Valid, Tag Matches, return d

0000000000000000 0000000001 1100

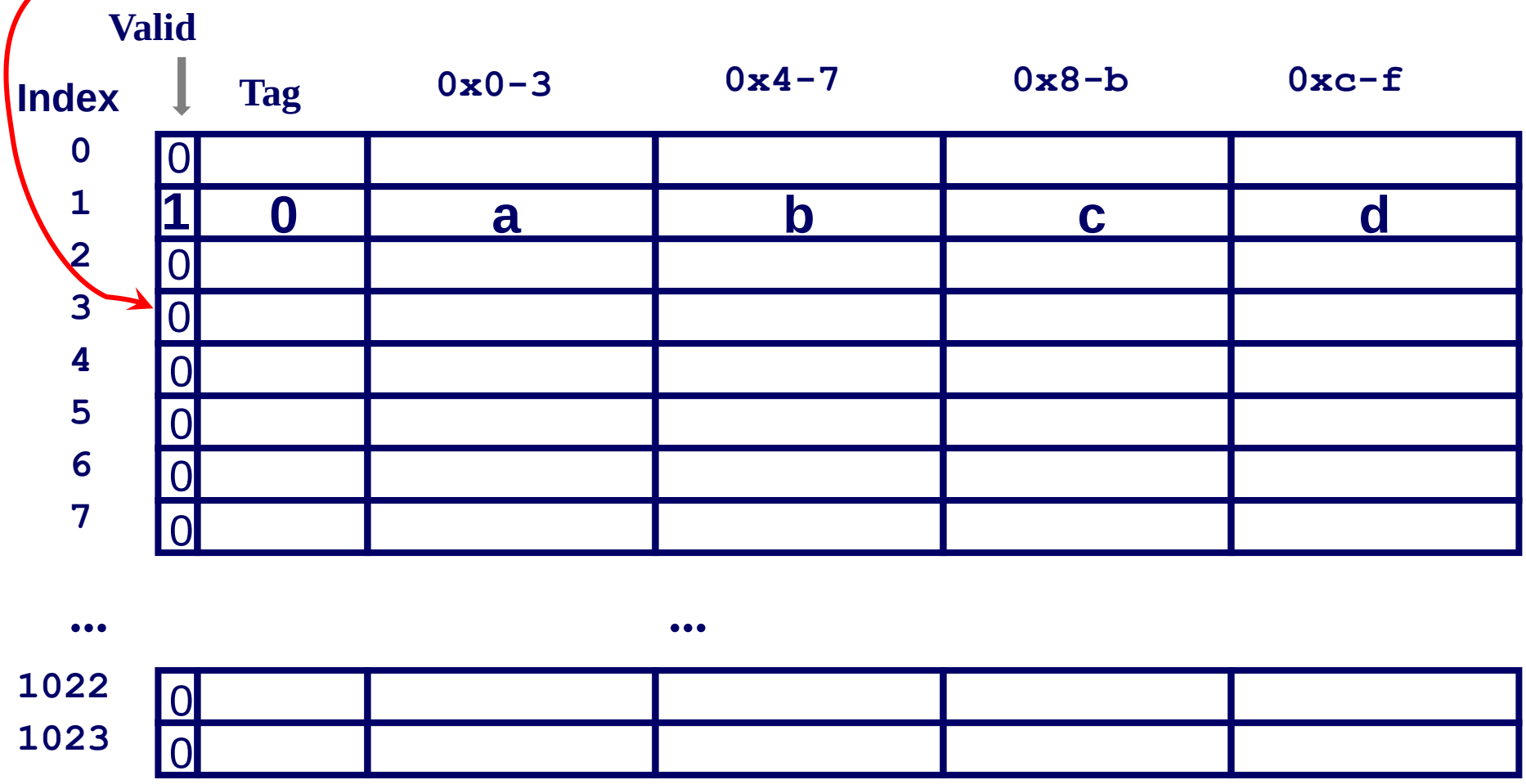
Tag field | Index field | Offset



3. Read 0x00000034 = ..011 0100

0000000000000000 0000000011 0100

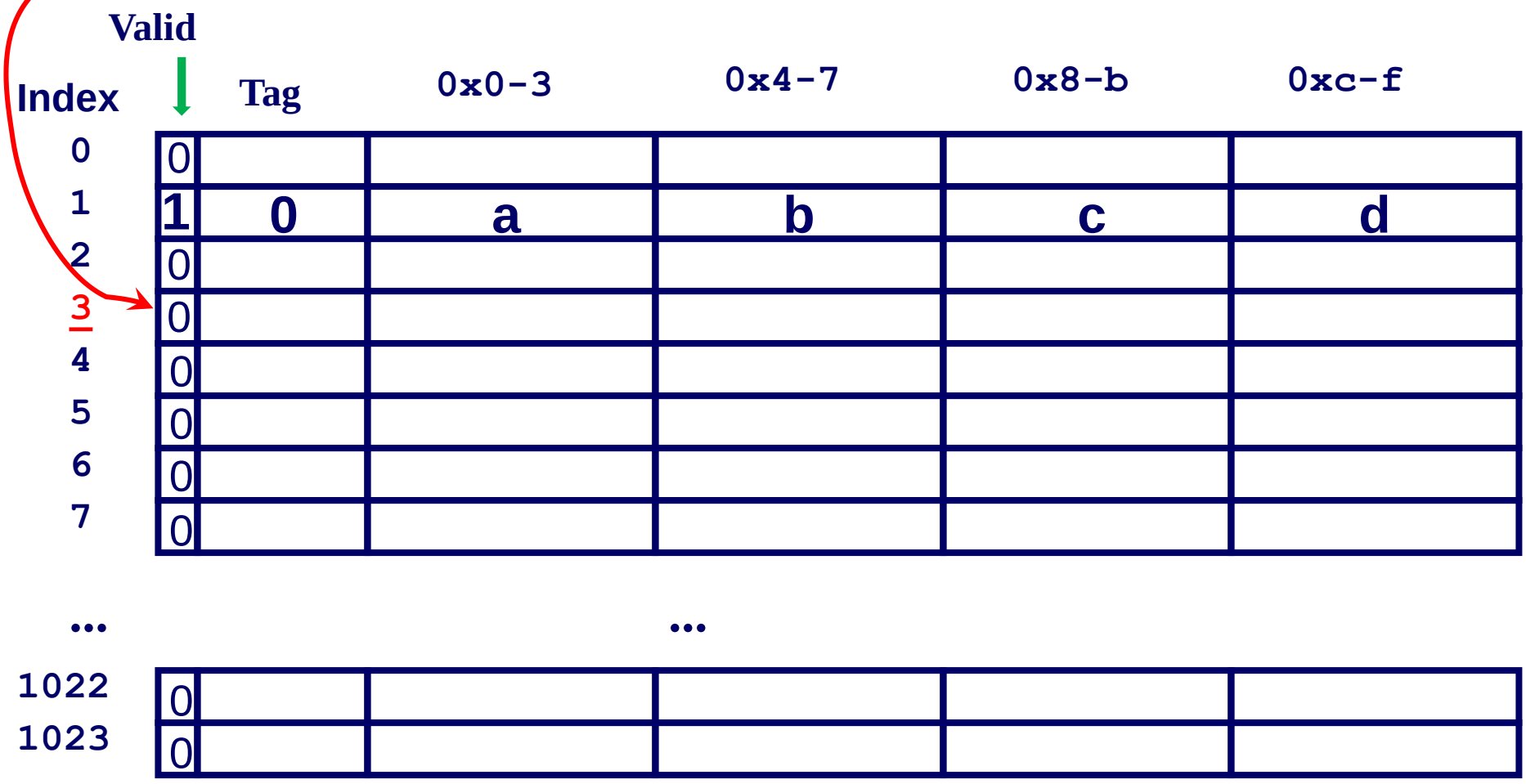
Tag field | Index field | Offset



So read block 3

0000000000000000 0000000011 0100

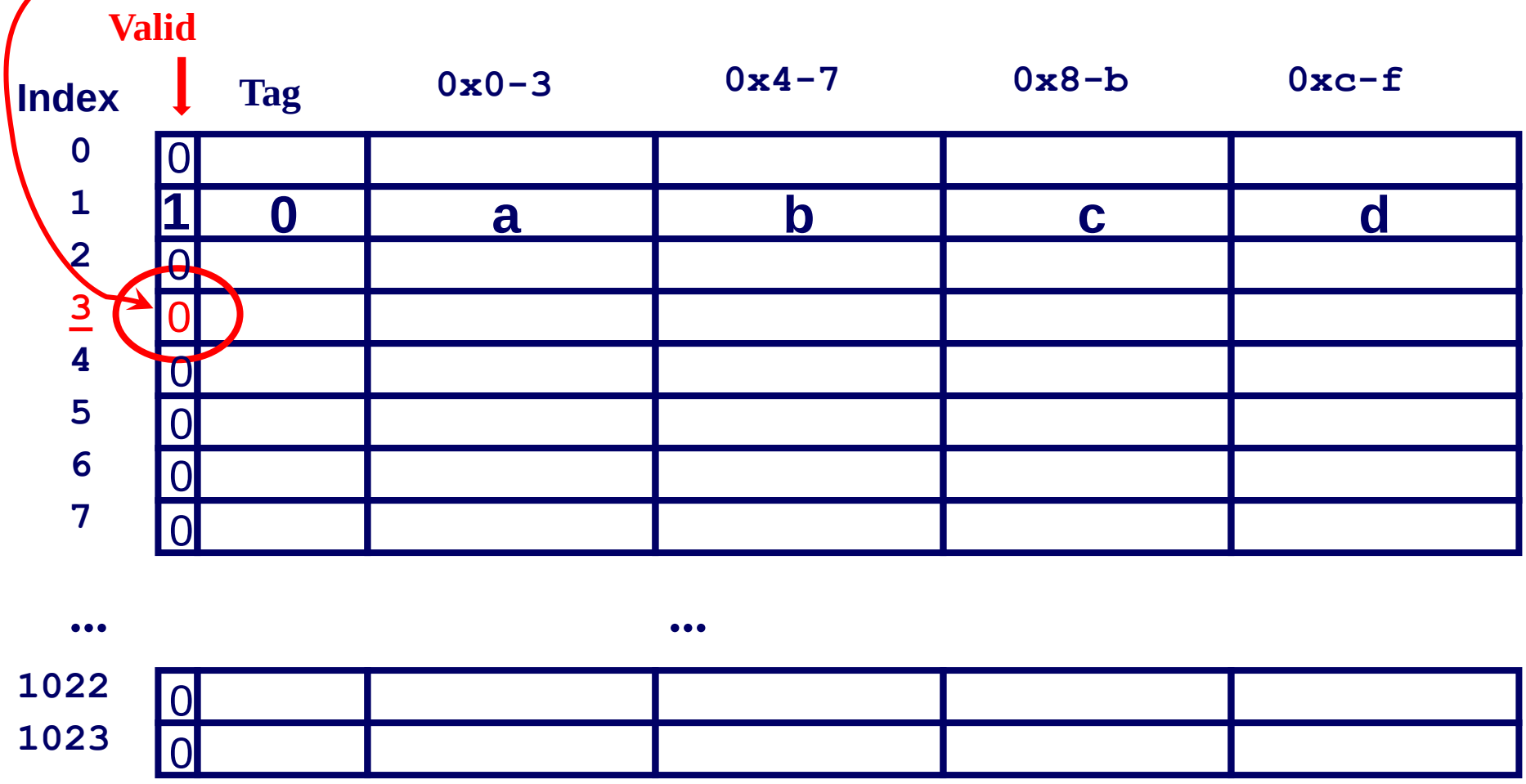
Tag field | Index field | Offset



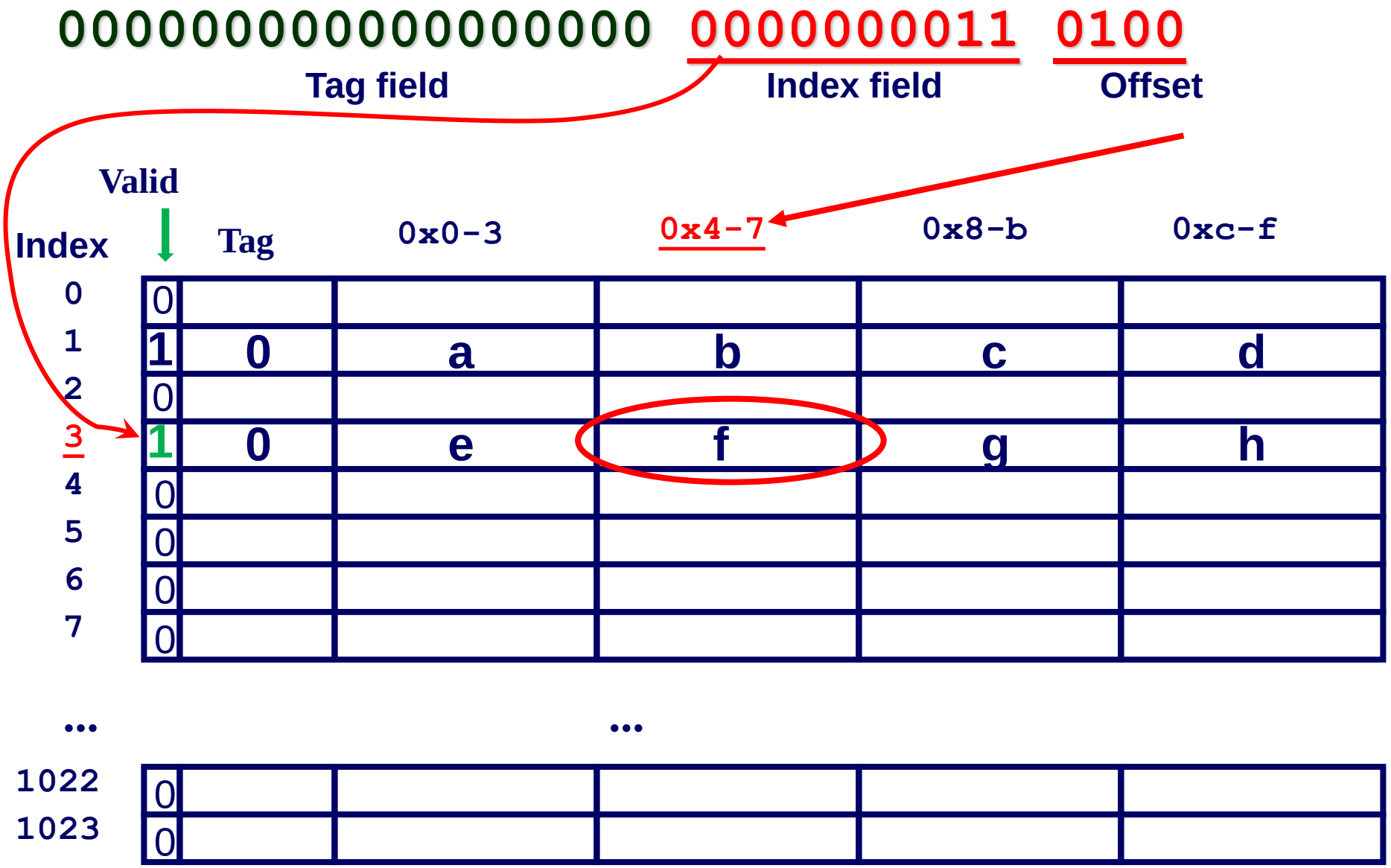
No valid data

0000000000000000 0000000011 0100

Tag field | Index field | Offset



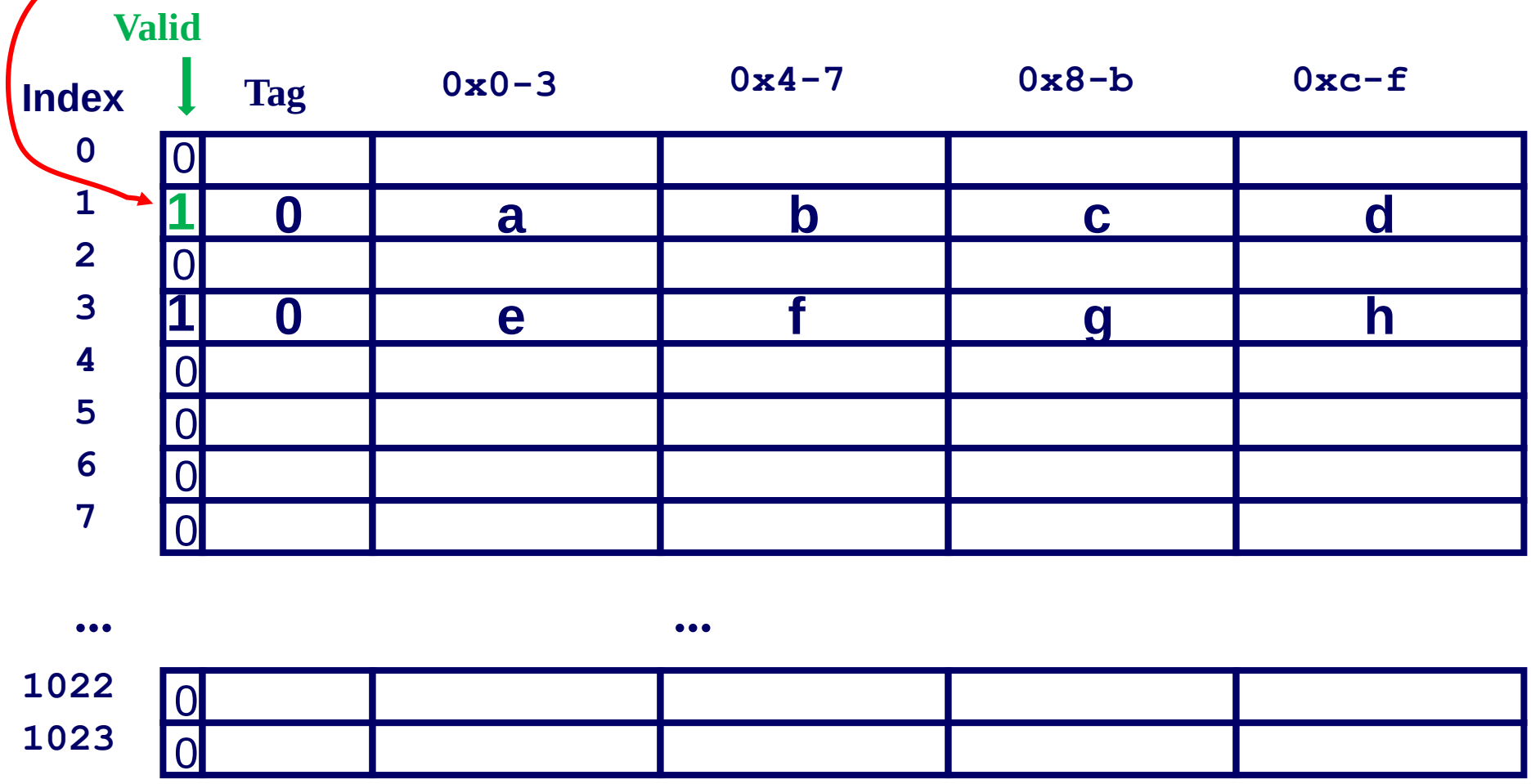
Load that cache block, return word f



4. Read 0x00008014 = ...1000 0000 0001 0100

0000000000000010 0000000001 0100

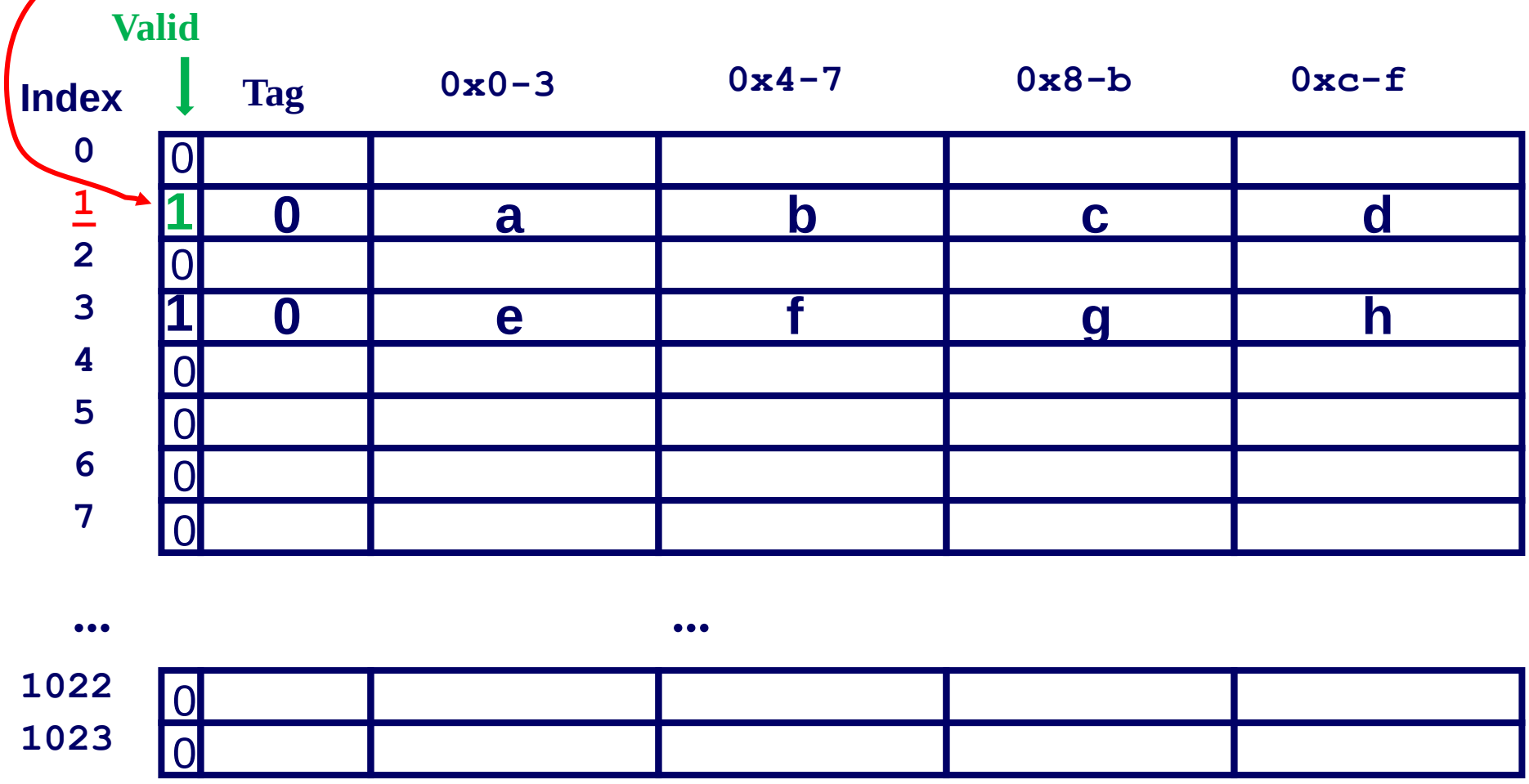
Tag field | Index field | Offset



So read Cache Block 1, Data is Valid

0000000000000010 0000000001 0100

Tag field | Index field | Offset



Cache Block 1 Tag does not match (0 != 2)

0000000000000010 0000000001 0100

Tag field is different

Index field

Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

0000000000000010 0000000001 0100

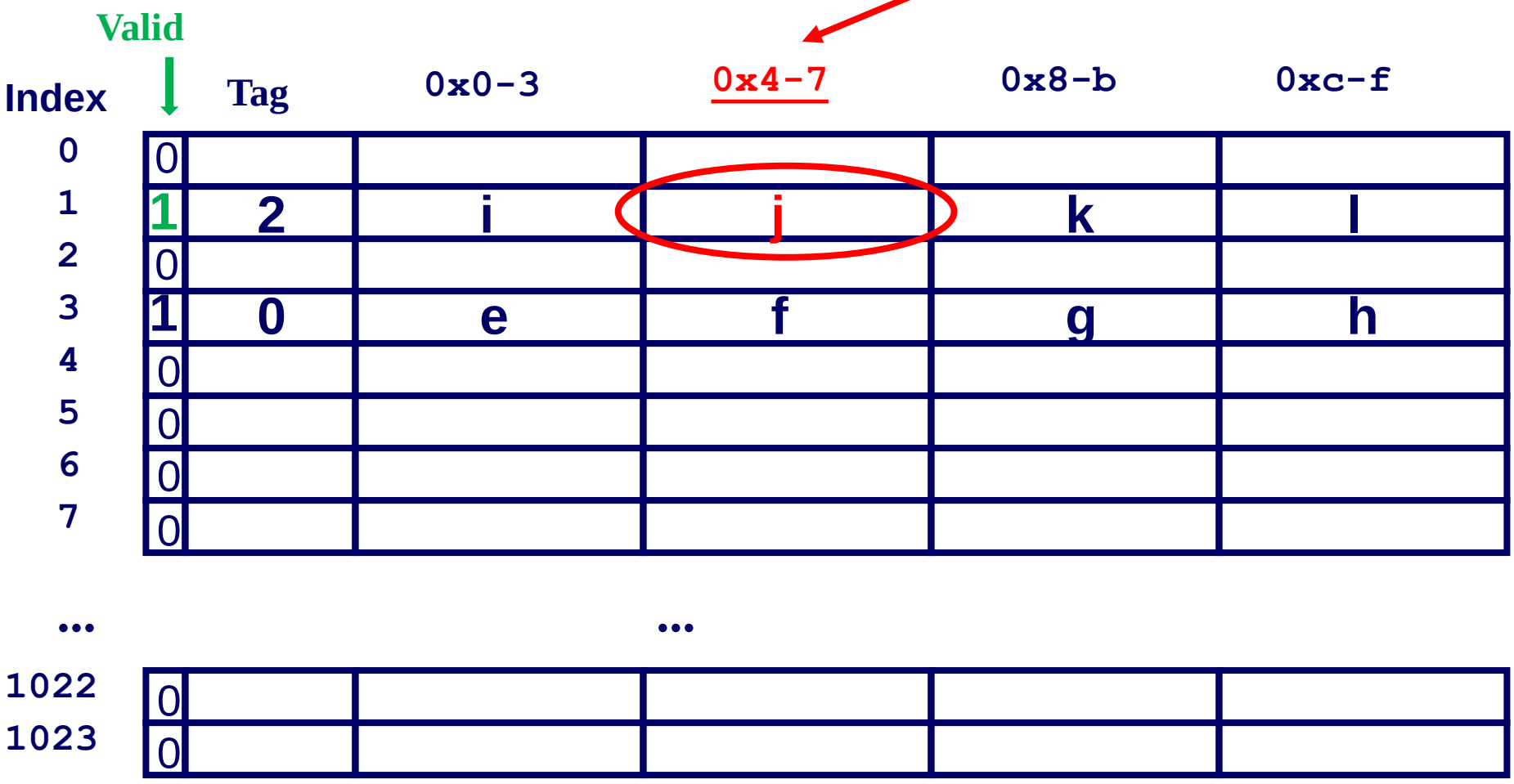
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

And return word j

0000000000000010 0000000001 0100

Tag field | Index field | Offset



The end of Garcia's example. Continue by yourself!

* Read address 0x00000030 !

  00000000000000000000 0000000011 0000

* Read address 0x0000001c !

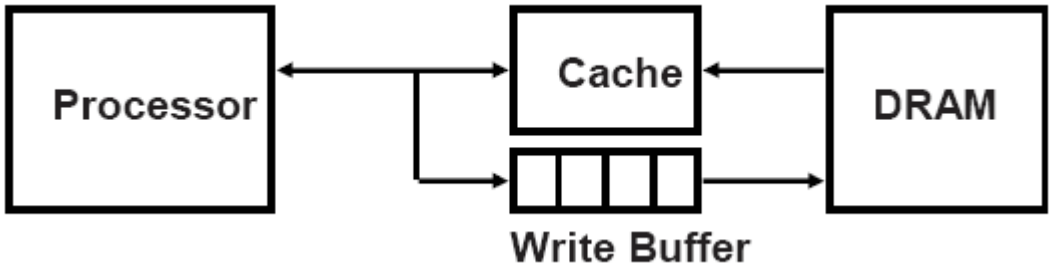
  00000000000000000000 0000000001 1100

Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		

A cache is on the way!

- **Data consistency** - the obvious requirement for the content of the same addresses to be consistent across different media.
- **Write through** - simultaneously with writing to the cache, the data is written to the write queue and then asynchronously to memory.
- **Write back** - data is written to the cache with the **Dirty** note (D bit lines). The actual writing of data to the main memory occurs only at the moment of possible interference of the corresponding cache line, when there is a risk of data loss.



More realistic cache line format

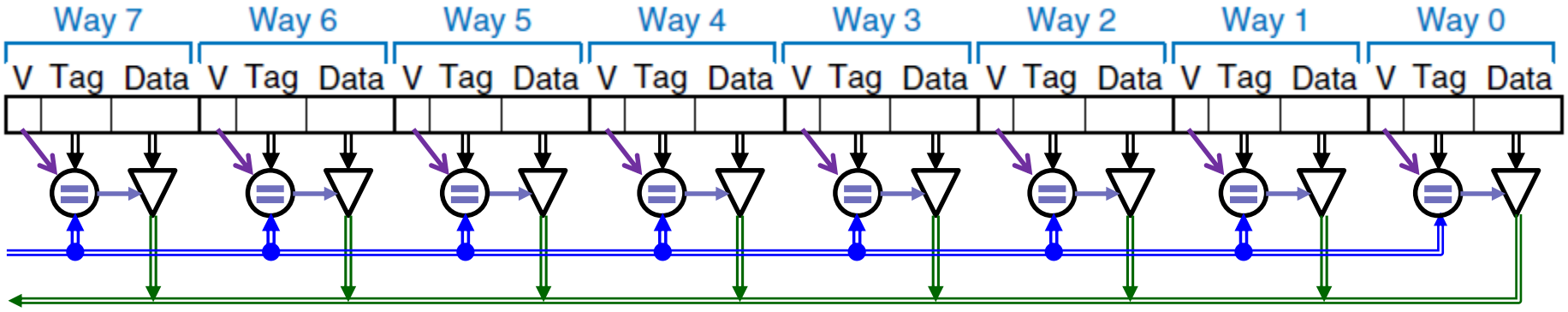
V	Other bits, e.g. D	Tag	Data

Dirty bit - auxiliary information in the cache that indicates that some item in the line contains a **different value** than in the main memory, so it will be necessary to write the change before loading new content into the line.

- **Tag** is the index of the corresponding block in the operational memory
  (basically it is the value of the pointer/address divided by the length of the whole block in bytes).
- **Data** fields containing eigenvalues at the corresponding addresses or the corresponding address.
- **Validity bit** - validity bit. Indicates whether the content of the Data field is valid at all.

Fully Associative Cache

- A fully associative cache contains only a single set and its degree of associativity is equal to the number of blocks (N=B). The memory address can be mapped anywhere.
- ... has the least conflicts for a given capacity, but needs the most comparators - chip area grows
- ...is suitable for relatively small caches.



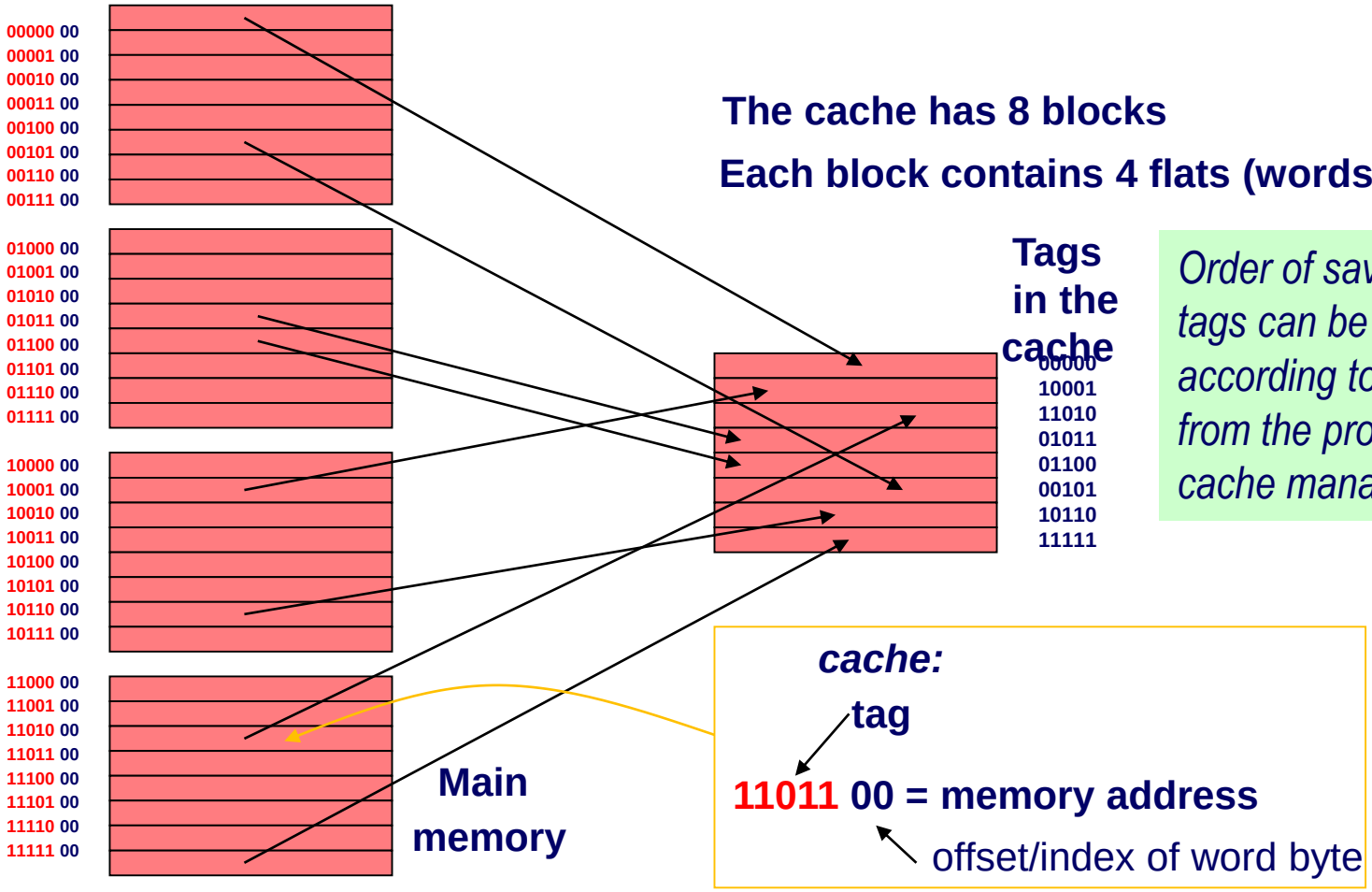
A fully associative cache has S=1, a single set !

Discussion on fully associative cache

- The width of the Tag field corresponds to the number of bits of the address minus the length of the byte index field in one word, which is 0 for an 8-bit processor, 1 for a 16-bit processor, 2 for a 32-bit processor, and 3 for a 64-bit processor. If the block contains more words, offset bits are reserved, i.e., subtracted from the tag width.
- Each cache block (row) has a separate bit-match comparator for two-bit strings of Tag width and Valid bit validity.
- The number of cache lines determines its capacity.
- The cache must have a strategy for releasing content (migrating data between hierarchical levels) when its capacity is exhausted.
- However, simpler variants can be implemented.

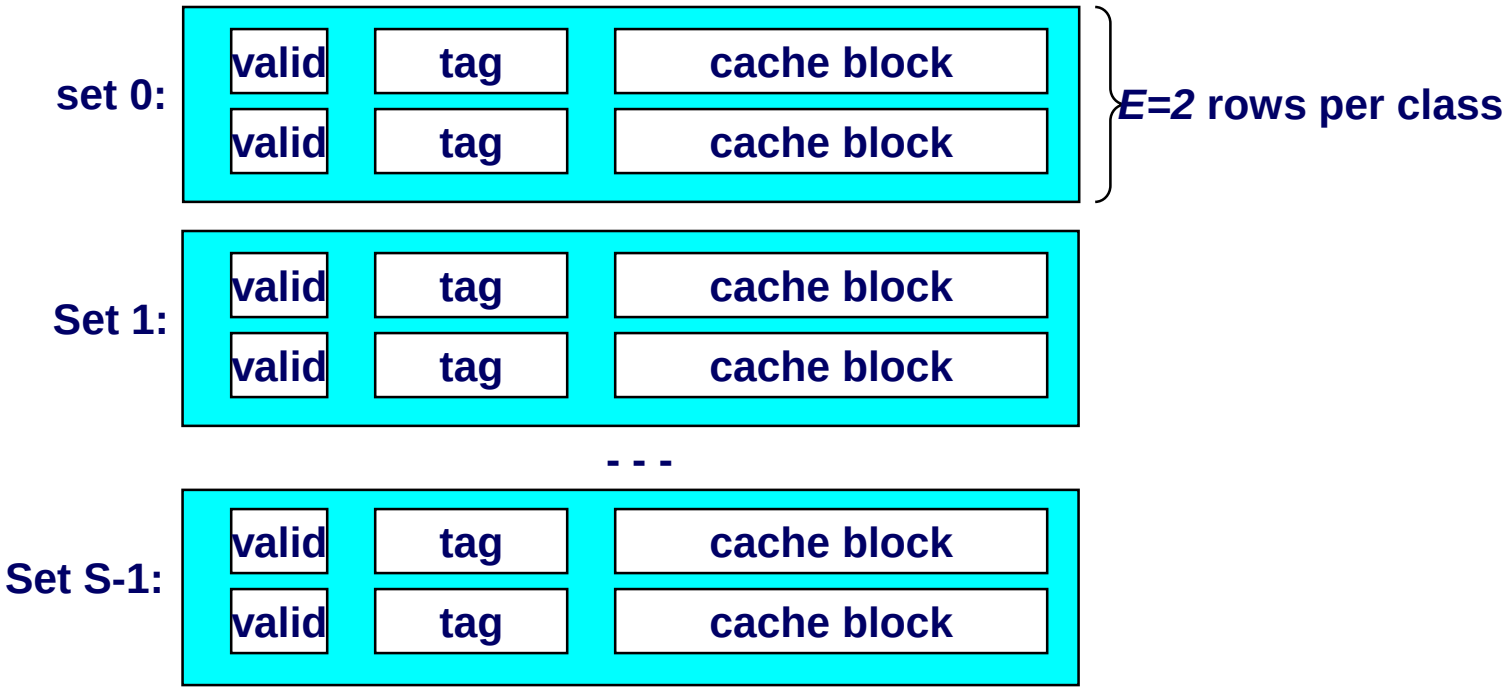
Fully-Associative Cache - Fully Associative Cache

A memory with 128 bytes, from which words of 4 bytes are read.



Set-Associative Cache
with a limited degree of associativity

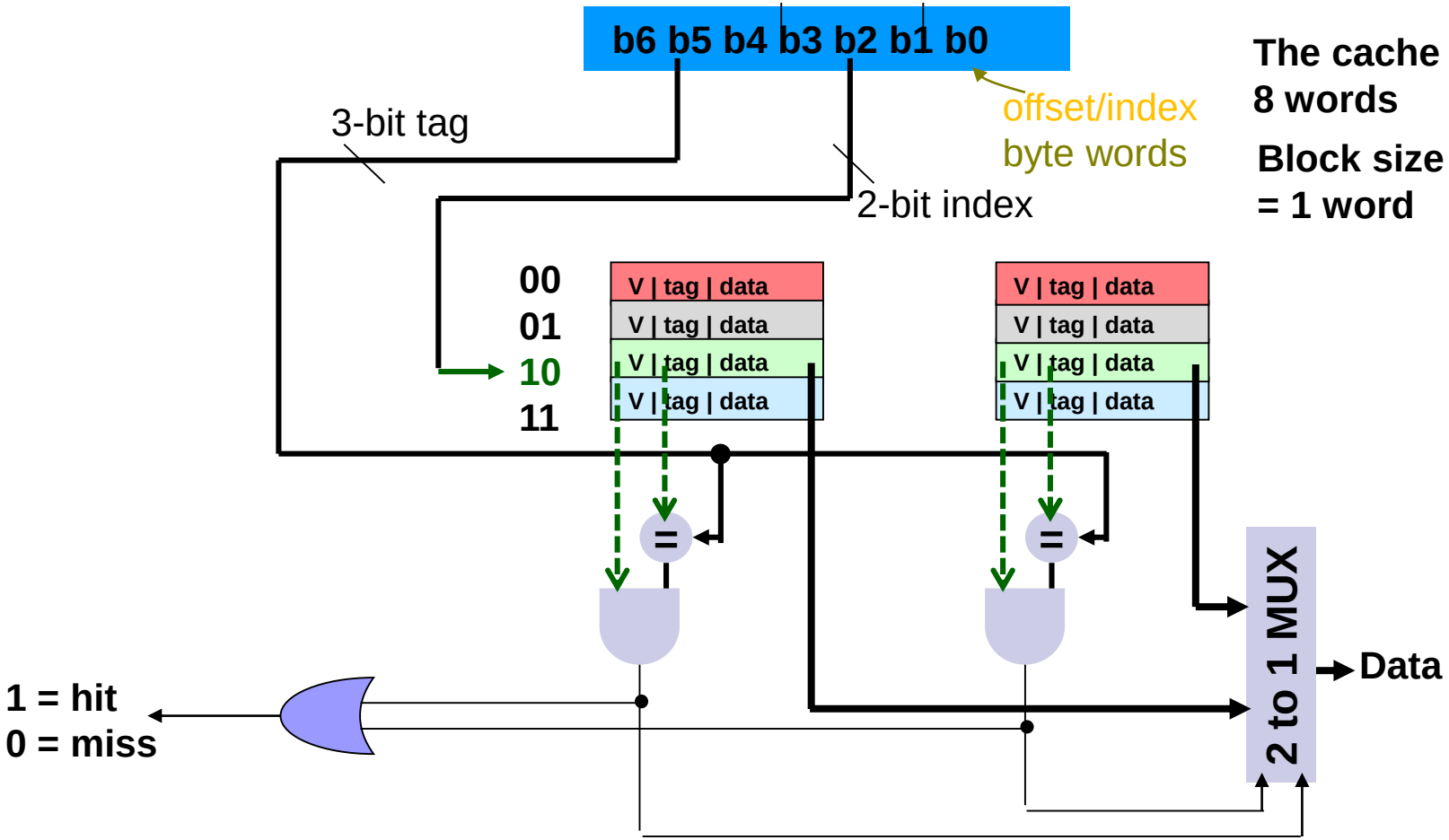
- Multiple rows (blocks) in one class (set)



Note: A fully associative cache is a special case of a B-way associative single-set cache.

Two-Way Set-Associative Cache

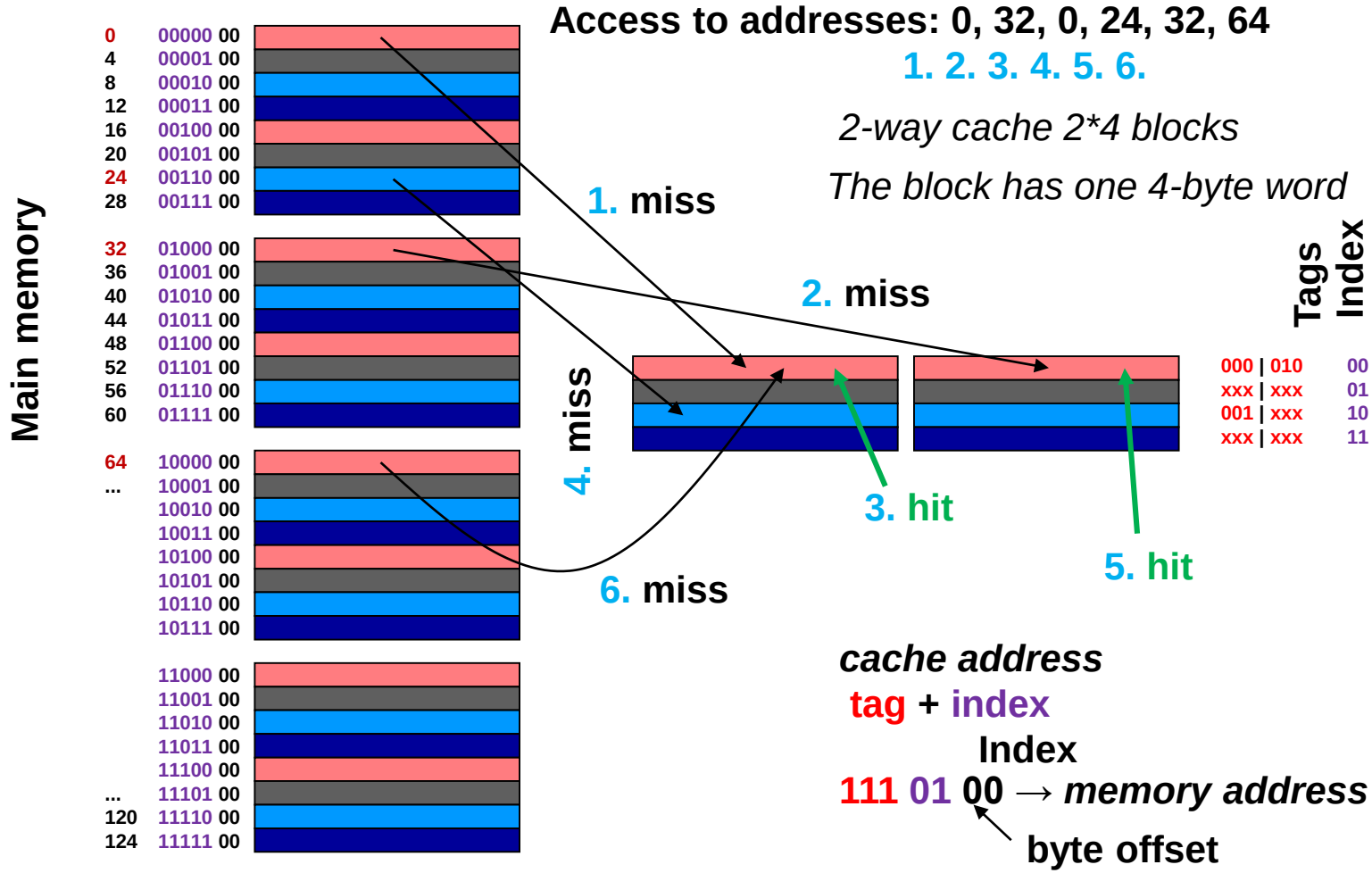
Memory of 128 bytes and read/written as 4-byte words



Design: two directly mapped caches are joined by associative data selection

Miss Rate: Two-Way Set-Associative Cache

Memory of 128 bytes and read/written as 4-byte words



The solution to the Cache Miss situation, data not in cache

Block release strategy/cache line

- Random - an arbitrary block is selected.
- **LRU** (Least Recently Used)
- **LFU** (Least Frequently Used
- **ARC** (Adaptive Replacement Cache), a combination of LRU and LFU.

- **Write-back**. Write-back of released lines depending on Dirty bit

The solution to the Cache Miss situation, i.e., data not in the cache

The data must be read from the main memory.
But if the associative cache is full, what do we replace it with?
The directly mapped cache had a clear choice, but not here.

Block release strategy/cache line

- Random - an arbitrary block is selected. Silly, but easy - the most common solution.
- **LRU** (Least Recently Used) - replaces the block that has been unused for the longest time. We need to keep information about the last use of this block and the complexity increases with the factor of the number of blocks in the set. For a two-way cache the implementation is easy; just add one bit that indicates the last element.
- **LFU** (Least Frequently Used), for each block we remember information about how often it was requested (even more complicated).
- **ARC** (Adaptive Replacement Cache), in which the LRU and LFU strategies are combined in an appropriate way.

- **Write-back**. At the same time we have to write the contents of the released cache lines to the main memory (D bits marked lines). Provided automatically.

LRU example in 2 way-cache

Tag 0, 2, 0, 1, 4

0: miss, bring into set 0 (loc 0)

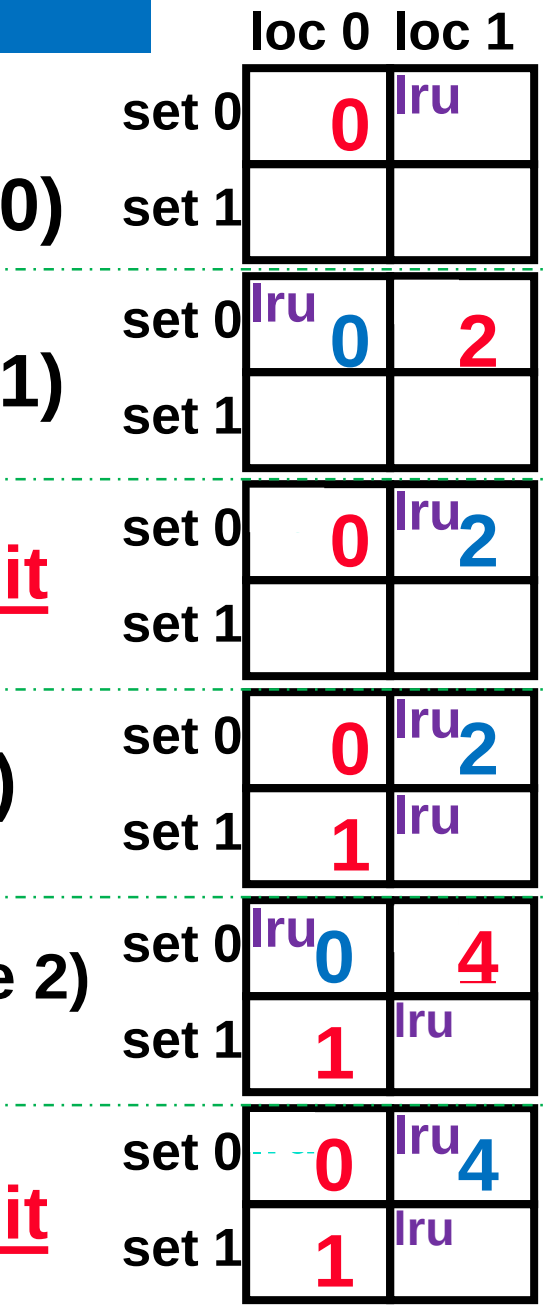
2: miss, bring into set 0 (loc 1)

0: hit

1: miss, bring into set 1 (loc 0)

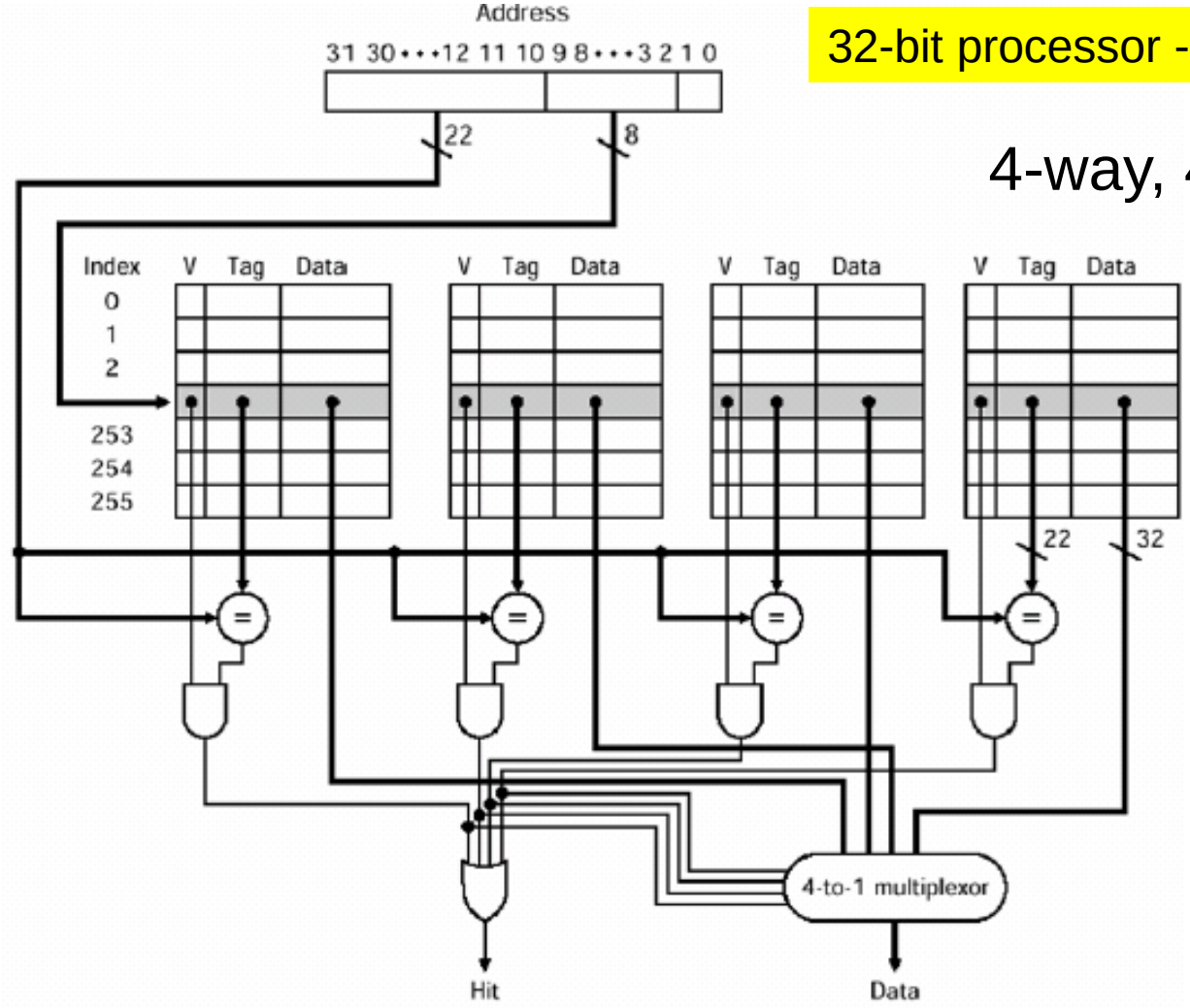
4: miss, bring into set 0 (loc 1, replace 2)

0: hit



[Source: Garcia, UCB]

Cache with limited degree of associativity N=4



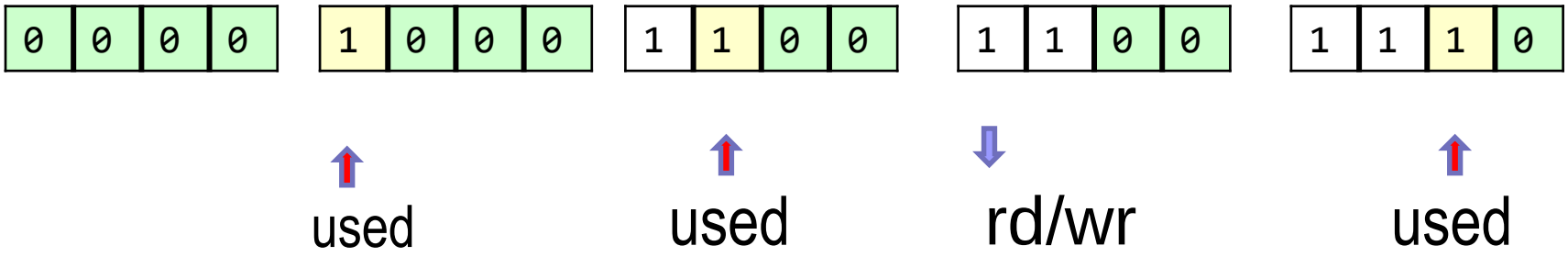
32-bit processor - 1 word = 4 bytes

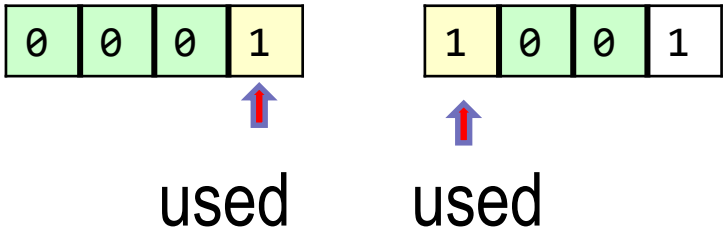
4-way, 4-way cache

Discussion: LRU Implementations

- For 2-way set associative cache, it is easy to keep track because we have only one LRU bit.
- With 4-way or greater, LRU requires more complicated hardware and much time. The number of states for full LRU implementation increases dramatically by factorial of the number of ways. It increases not only storage overhead but also time for updating!
- 4-way cache has 24 states and requires 5 bits per set. 8-way cache needs 40320 states and 16 bits per set.
- Therefore, LRU replacement policy is usually implemented by simpler mechanisms as **binary tree pseudo-LRU** but its algorithm is outside of APO topic, see e.g.: https://en.wikipedia.org/wiki/Pseudo-LRU , or K. Kędzierski, M. Moreto, F. J. Cazorla and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, 2010, pp. 1-12.

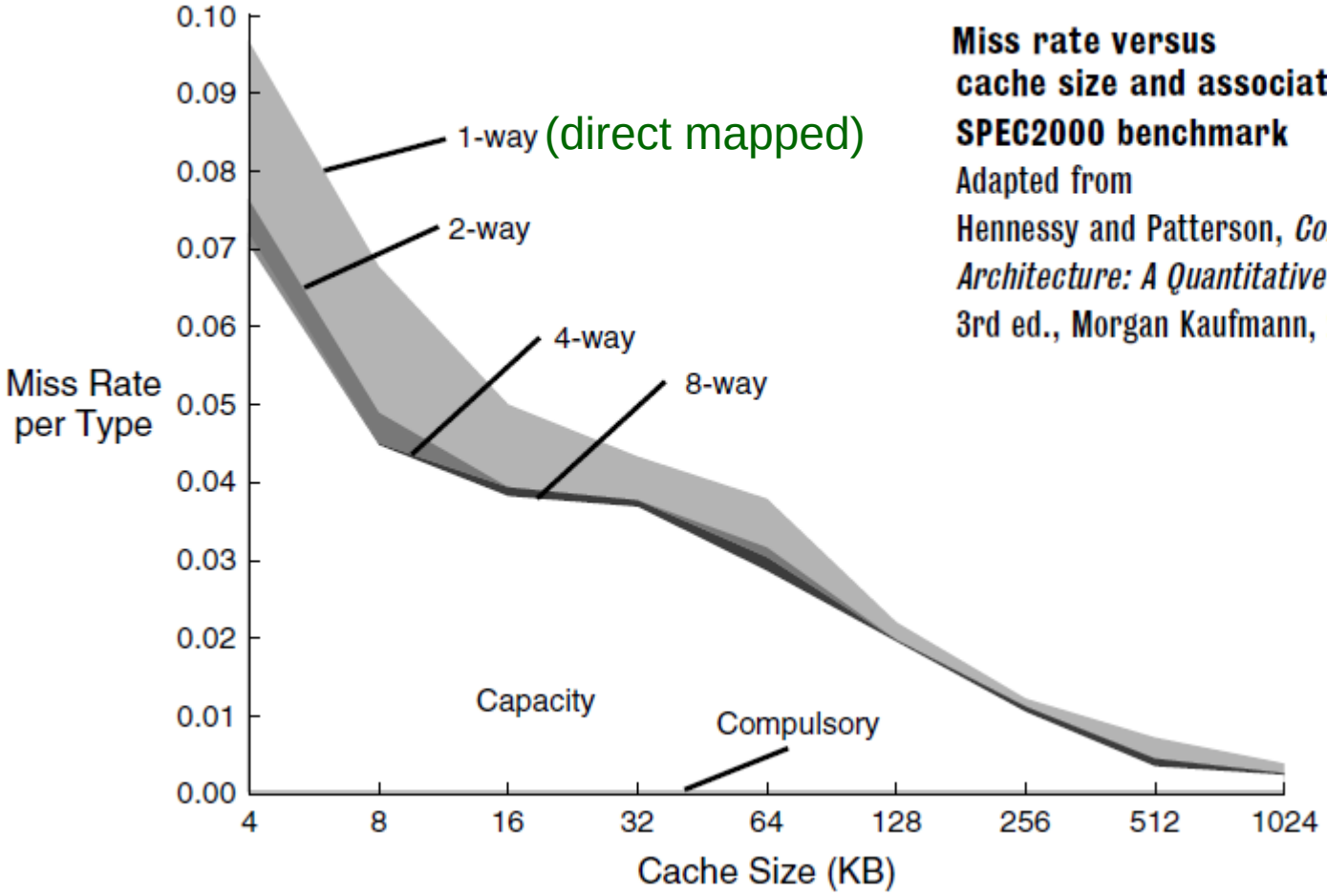
4-way associative cache
1 bit per line





When a request for a row is made, the first available row is selected and its bit is set to '1'.
When you run out of free cells, zero all except for the last one written.
The mechanism is not perfect, but easy to implement.

Comparison of different cache sizes and organizations



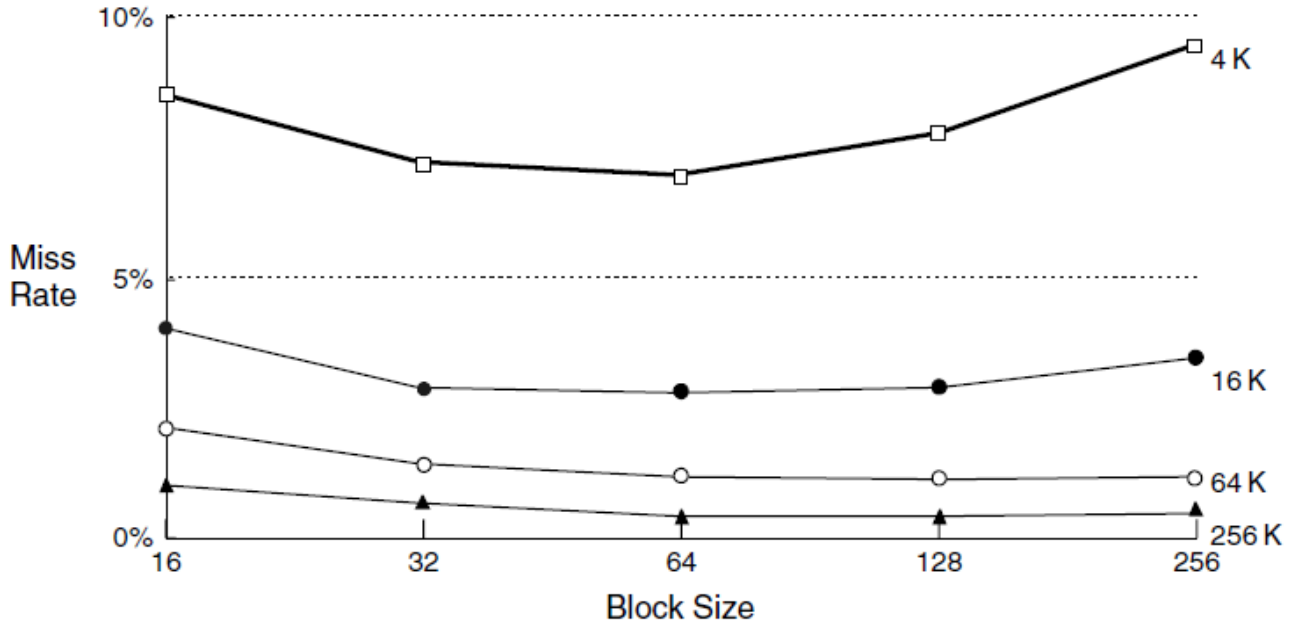
Miss rate versus cache size and associativity on SPEC2000 benchmark
Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Remember: 1. miss rate is not a cache property!

2. miss rate is not a feature of the program!

What does spatial location bring?

We can reduce the Miss Rate by increasing the block size, i.e., by using the spatial locality principle. If the cache memory has a fixed byte length, then reorganizing it into more blocks will reduce the number of sets, which will result in an increase in conflicts (increase in Miss Rate)...



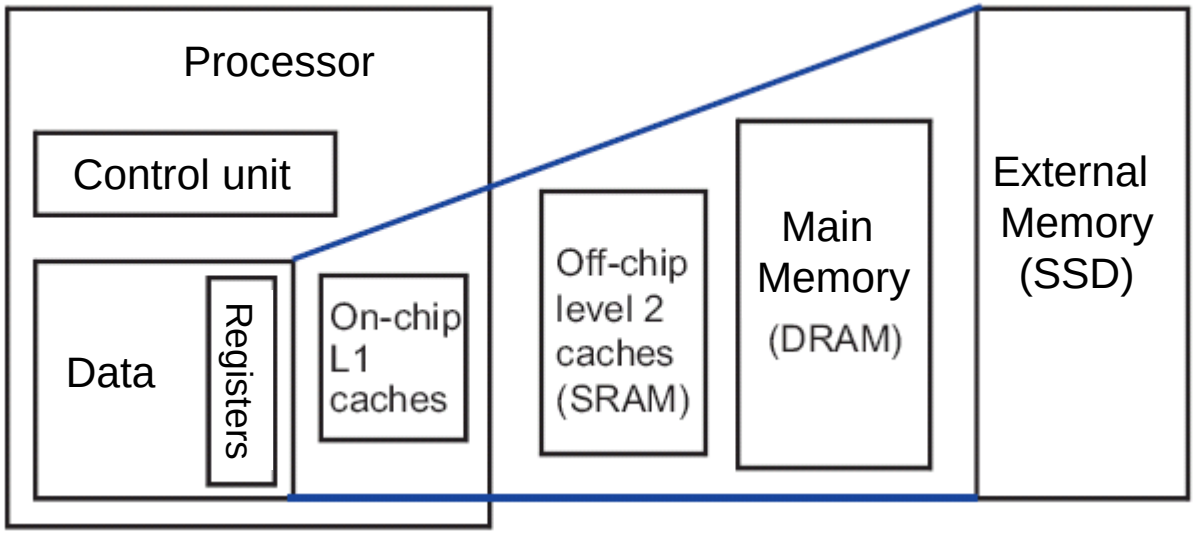
Miss rate versus block size and cache size on SPEC92 benchmark Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Trend - Multilevel SP

- The primary SP is directly connected to the processor
  - Quick, small. Most important: minimum Hit Time, usually 2 to 8-way
- L2 SP treats primary SP outages
  - Bigger, slower, but still faster than main memory. Highlights: low miss rate
- Main memory handles L2 failures
- Today's most powerful systems also have L3

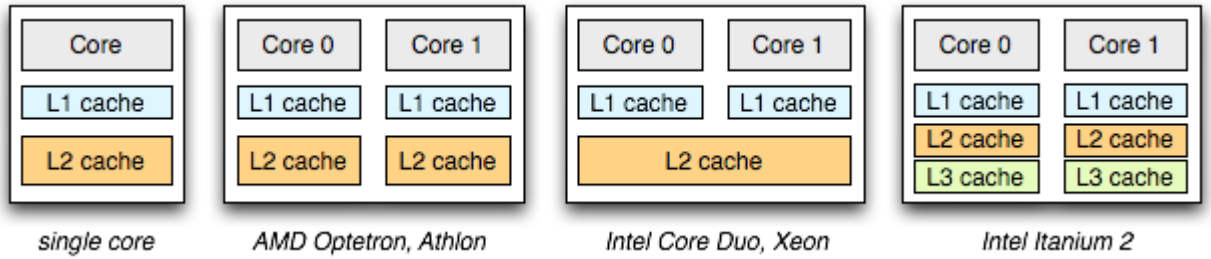
	Typically for L1	Typically for L2
Number of blocks	250-2000	15 000-250 000
KB	16-64	2 000-3 000
Block size in B	16-64	64-128
Miss penalty (in throw)	10-25	100-1 000
Miss rates	2-5%	0,1-2%

Memory hierarchy

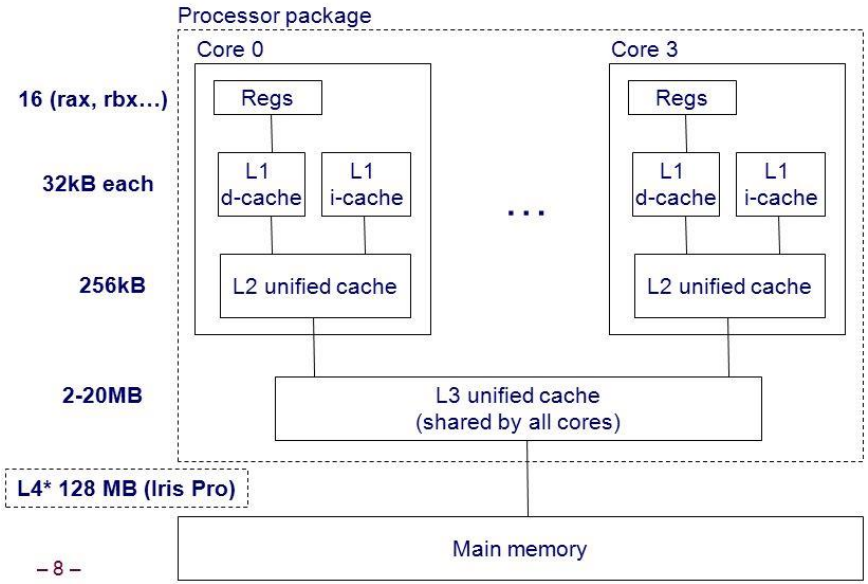


The same data can appear in multiple parts inside the hierarchical memories!

Organization on some Intel processors



Intel Core i7 cache hierarchy (2014)



- L1 cache
  - Instructions: 4-way
    Latency: 3 clock cycles
  - Data: 8-way
    Latency 4 clock cycles
- L2: 8-way
  Latency 10 clock cycles
- L3-16-way
  Latency 40 cycle clocks

Source.

*Again motivational cache example

Let the single-cycle processor read instructions from fast SRAM and complete each register instruction in 1 clock cycle.

- At f=1 **GHz**, 10^9 pure register instructions take 10^9 *1 ns = **1 s. The processor has a performance of 1000 MIPS (Million Instructions Per Second)**

If the algorithm accesses the external super-fast DRAM but randomly, i.e. non-sequentially, it takes 30 ns to read each data:

- If 20% of the instructions work with DRAM, 10^9 instructions will take $8*10^8$ *1 ns+$2*10^8$ *30 ns= **6.8 s. Performance drops to 147 MIPS (=1000/6.8)**

If we add one cache level with an access time of 1 ns, then

- **at Miss Rate=100% the program will slow down because before each access it will be unnecessary to look first into the cache** $8*10^8$ *1 ns+$2*10^8$ *(**30+1**) ns= **7 sec.**
- **However, with a normal Miss rate=2%, the program speeds up:** $8*10^8$ *1 ns+$2*10^8$ *1 ns + $4*10^{6*}$ **30** ns= **1.12 s, i.e. 892 MIPS.**

Did you get the cashe?

- If so, you already realize that we use the principle of temporal and spatial locality of data. Efficient use of cache leads to significant speed-up of programs...!!!
- There are also techniques to make caching more efficient, and compilers can also be set to optimize the code...
  (Their analysis is beyond the scope of LSP and belongs in A4M36PAP.)
- However, even the best compiler only compiles the code written by the programmer responsible for selecting algorithms, storing data structures in memory, and manipulating them. By making appropriate choices, the program's speed can be significantly affected.
- Of course, we don't accelerate every code row; our time is also one of the critical variables that we optimize. We mainly focus on repeated parts of the code; their slight acceleration is more visible on the outside.
- And sometimes, it's enough if we do not create clumsy constructions in our code…

Did you get the cashe?

- Data cache - easy techniques
  - Appropriate arrangement of data - data that we plan to use sequentially, sort sequentially in memory, etc.
  - Merging arrays or related data structures
  - Work after blocks of data - use already used blocks as soon as possible
  - iterations in nested cycles - see examples
    - in order to traverse memory sequentially and not in jumps
  - merging two loops into one - Loop fusion, etc.
- Instructional cache - already advanced techniques
  - Appropriate arrangement of the code or rearrangement of functions in memory is usually done by the compiler
  - Profiling, special tools for development environments find the places in the code that are most restrictive

Did you get the Cache Dirty bit?

```
void SlowBubble(int* arr, int len) {
  for (int i = 0; i < len; i++) {
     for (int j = i + 1; j < len; j++) {
       if (arr[j] < arr[i]) // Swap
          { int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
          }
       }
  }
} // Each memory write sets also dirty bit in cache!!


void FastBubble(int* arr, int len) {
  for (int i = 0; i < len; i++) {
     int min = arr[i];
     int ixmin = i;
     for (int j = i + 1; j < len; j++)
     {
        if (arr[j] < min) { min = arr[j]; ixmin = j; }
     }
     int tmp = arr[i]; arr[i] = arr[ixmin]; arr[ixmin] = tmp;
  }
} // We minimized write back!!
```

Did you understand the cache space location?

- Spatial location - cache conflicts:

```
/* Before optimization */
int values[SIZE];
int keys[SIZE];
int scores[SIZE];

/* After optimization */
struct item{
 int value;
 int key;
 int score;
};
struct item records[SIZE];
```

Assume a multipath associative cache...

```
for(i=0; i<SIZE; i++)
  for(j=0; j<SIZE; j++)
    ...
```

Place data that you access in short succession next to each other (group them).

Did you understand the temporal location of the cache?

- Time location:

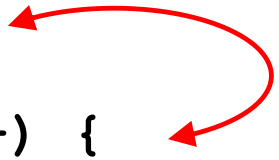
```
/* Before optimization */
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      d[i][j] = a[i][j] - c[i][j];

/* After optimization, so called Loop fusion */
for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] - c[i][j];}
```

It's not just about saving instructions, but also using the cache more efficiently...

- Another example is matrix multiplication

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++) {
    tmp = 0;
    for (k=0; k < N; k++)
       tmp += y[i][k]*z[k][j];
    x[i][j] = tmp;
  }
```



Does it help if we swap these two lines? Will the program be equivalent?

The answer depends on the storage of the matrices in memory...

Did you get the cache?

- Matrix multiplication can be accelerated by block multiplication

Idea: the calculation is split into BxB submatrices that fit into the cache => elimination of "capacity misses"

```
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B-1,N); j++)
      {
        tmp = 0;
        for (k = kk; k < min(kk+B-1,N); k++)
           tmp += y[i][k]*z[k][j];
        x[i][j] = x[i][j] + tmp;
      }
```

To read: http://suif.stanford.edu/papers/lam-asplos91.pdf

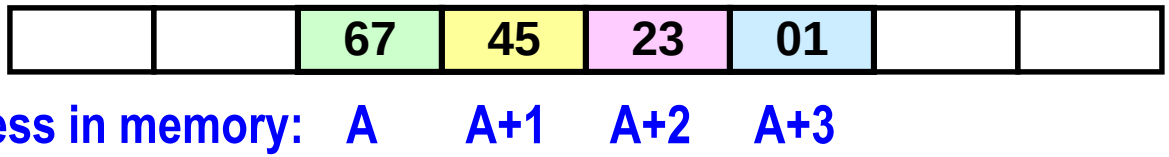
*Multi-byte numbers

and their placement in different architectures, which is reflected in their interconnection

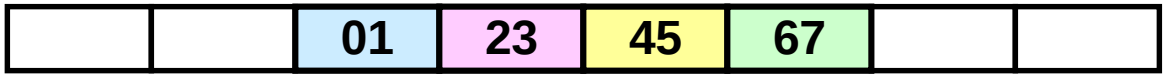
Ways to store/transfer numbers

Number 0x1234567 saved

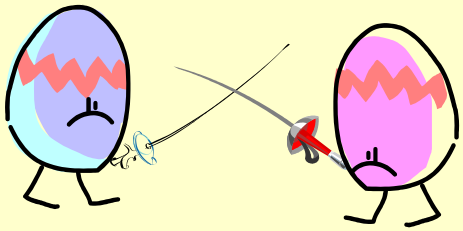
Little Endian - it's
Intel, AMD processors



Big Endian - downto
aka Network Byte Order



*the dominant method of **transmission over networks**, in IBM mainframes and some industrial computers, although their newer types are also gradually moving to Little Endian*



***Little-Endien** comes from the book Gulliver's Travels, Jonathon Swift 1726, in which it referred to one of the two opposing factions of the Lilliputians. Its followers ate eggs from the narrower end to the wider, while **Big Endien** did the opposite. And the war didn't take long...*

Do you remember how the war ended?

Alignent - cz:data alignment?

In assembler, it is easy to control the alignment of data in memory

```
.DATA
.ALIGN 2
    var1: .BYTE 3, 5, 'L', 'S', 'P'
    var2: .WORD 0x12345678
.ALIGN 3
    var3: .HALF 1000
```

Big Endian

var1 ↓ var2 ↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	4C	53	50				12	34	56	78				
0x2010	10	00														

var3 ↑

Little Endian

var1 ↓ var2 ↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	4C	53	50				78	56	34	12				
0x2010	00	10														

var3 ↑

Storage versus Memory Alignment

address	+0	+1	+2	+3
0				
XNA=4	v1	v2	n1	n2
8	n3	n4	v3	v4
YA=12	m1	m2	m3	m4

address	+0	+1	+2	+3
0				
XNA=4	v1	v2	d1	d2
8	d3	d4	v3	v4
YA=12	x1	x2	x3	x4

*YA← |x1|x2|x3|x4|;

*XNA ← |d1|d2|d3|d4|;

v1	v2	n1	n2	tmp ← *XNA;
v1	v2	0	0	tmp ← tmp & 0xFFFF0000;
0	0	d1	d2	**tmp1** ← d>>16;
v1	v2	d1	d2	**tmp1** ← tmp | **tmp1**;
				*XNA ← tmp1;
n3	n4	v3	v4	tmp ← *(XNA+4);
0	0	v3	v4	tmp ← tmp & 0xFFFF;
d3	d4	0	0	**tmp1** ← d<<16;
d3	d4	v3	v4	**tmp1** ← tmp | **tmp1**;
				*(XNA+4) ← tmp1;

If the memory has hardware support for non-aligned reads/writes, i.e. it can write them into a part of a word, the processor will only perform framed operations.

Waste of memory

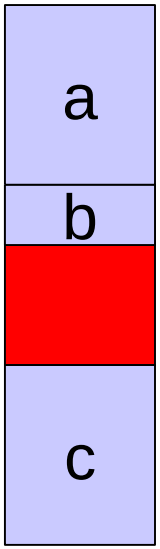
- Do you see a difference in these declarations?

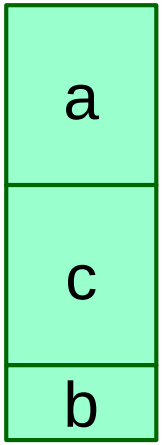
- **/* Before optimization */**

```
int a=0;
char b='a';
int c=1;
```

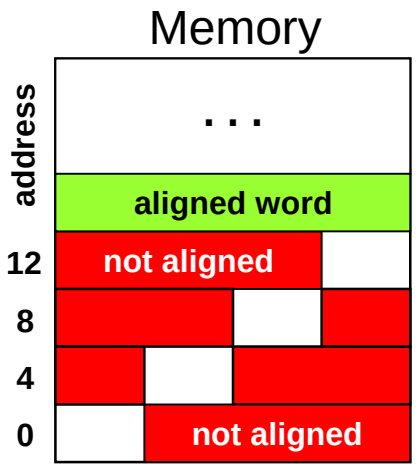
- **/* After optimization */**

```
int a=0;
int c=1;
char b='a';
```





- The memory is organized by bytes, but commonly used memories are configured mainly for fast writing/reading of whole words. The only exceptions are usually accesses to peripherals whose hardware can also support setting the write/read of a specific word byte (byte enable mask).
- If we have a 32-bit processor, then its word length is 4 bytes. The addresses issued by the processor when reading and writing to memory end with 00. For a 64-bit processor, it ends with 000.
- *Note: Some 64-bit processors can switch to 32-bit mode, but still cache 64-bit words internally.*



Did you understand memory alignment?

Sometimes it is also useful to think about aligning the data in memory, because an unaligned element can take more words in the cache or overlap the page.
It is a good idea to check if the compiler aligns double numbers according to the words in the cache,
and if not:

- allocate as much as we need + one extra element
- the AND operation aligns the address to the beginning of the data

Newer languages already have allocation functions where you can set the alignment. For example, in C++ Visual Studio it is about

```
void * _aligned_malloc(
        size_t size,
        size_t alignment );
```

The alignment parameter must be equal to 2^N , where N is a positive number.

If the system has automatic memory management, then of course, it aligns itself.

Did you understand memory alignment?

Manual alignment:

```
const int SIZE_OF_ARRAY = 64; const int SIZED = sizeof(double); // 8
// trick for debugger - field definition in the structure allows us to see the whole field
struct MyArr { double Item[SIZE_OF_ARRAY]; } *myArr;
double *p = (double*) malloc( SIZED * (SIZE_OF_ARRAY+1) );
// set the lower bits to 0 ( -SIZED = 0b1111 ... 1111 1000 )
myArr = (MyArr*) ( (long) ( (unsigned char *)p + SIZED-1 ) & -SIZED );

int i=0; double d=1.0/2; double * pd = myArr->Item; // example of array usage
do { *pd++ = (d *= 2); } while (++i < SIZE_OF_ARRAY);

// note1. do...while loop executes faster than while...do or for, see previous lecture
// note2. we will use something similar later to align virtual memory pages
```