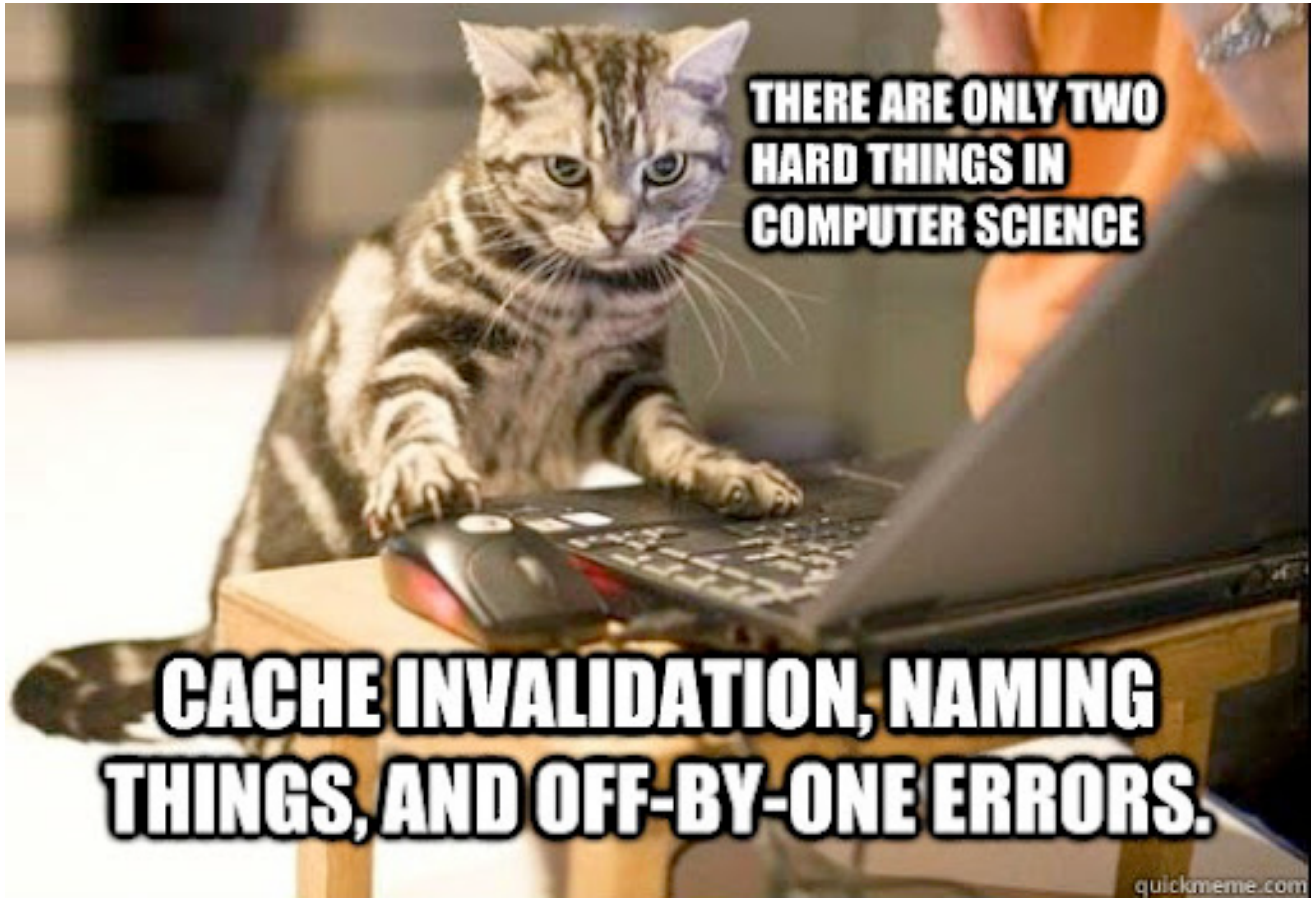
Data Cache & Locality

Based on slides by David Merchant and Troels Henriksen

Axel Kanne

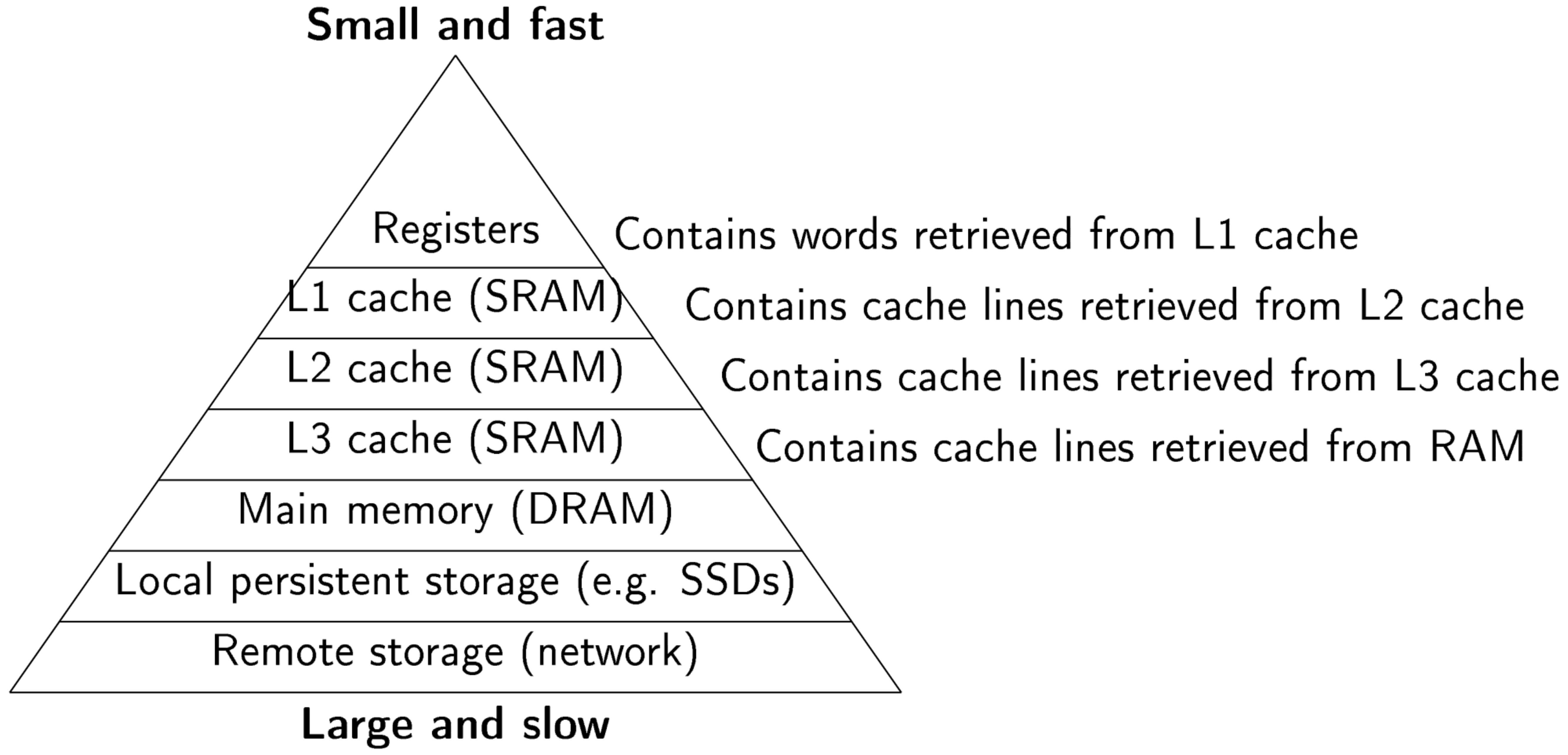
Agenda

- Theory
- Locality of code
- Cache exercises


THERE ARE ONLY TWO HARD THINGS IN COMPUTER SCIENCE
CACHE INVALIDATION, NAMING THINGS, AND OFF-BY-ONE ERRORS.
quickmeme.com

The memory hierachy

- Small memory: fast but small (it requires more power, and each storage unit is expensive)
- Large memory: slow but large (requires less power, and each storage unit is cheap)
- Solution: The smaller and faster device at level k acts as a cache for the larger slower device at level k + 1.
- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

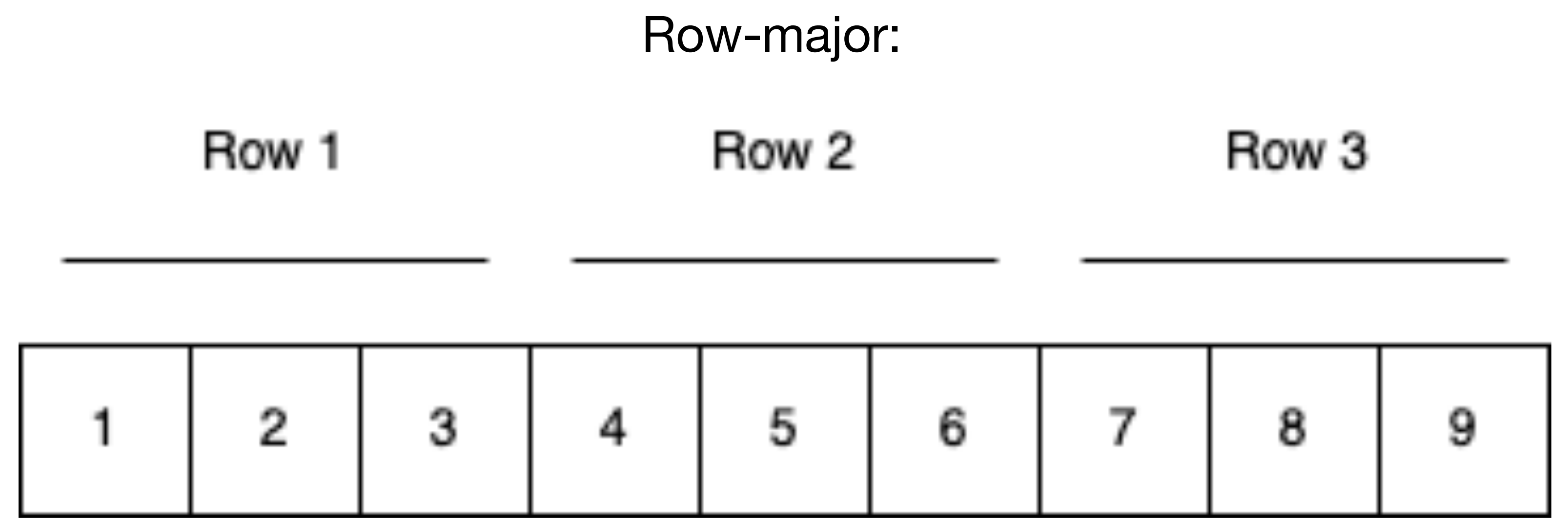

Small and fast
Registers
Contains words retrieved from L1 cache
L1 cache (SRAM)
Contains cache lines retrieved from L2 cache
L2 cache (SRAM)
Contains cache lines retrieved from L3 cache
L3 cache (SRAM)
Contains cache lines retrieved from RAM
Main memory (DRAM)
Local persistent storage (e.g. SSDs)
Remote storage (network)
Large and slow

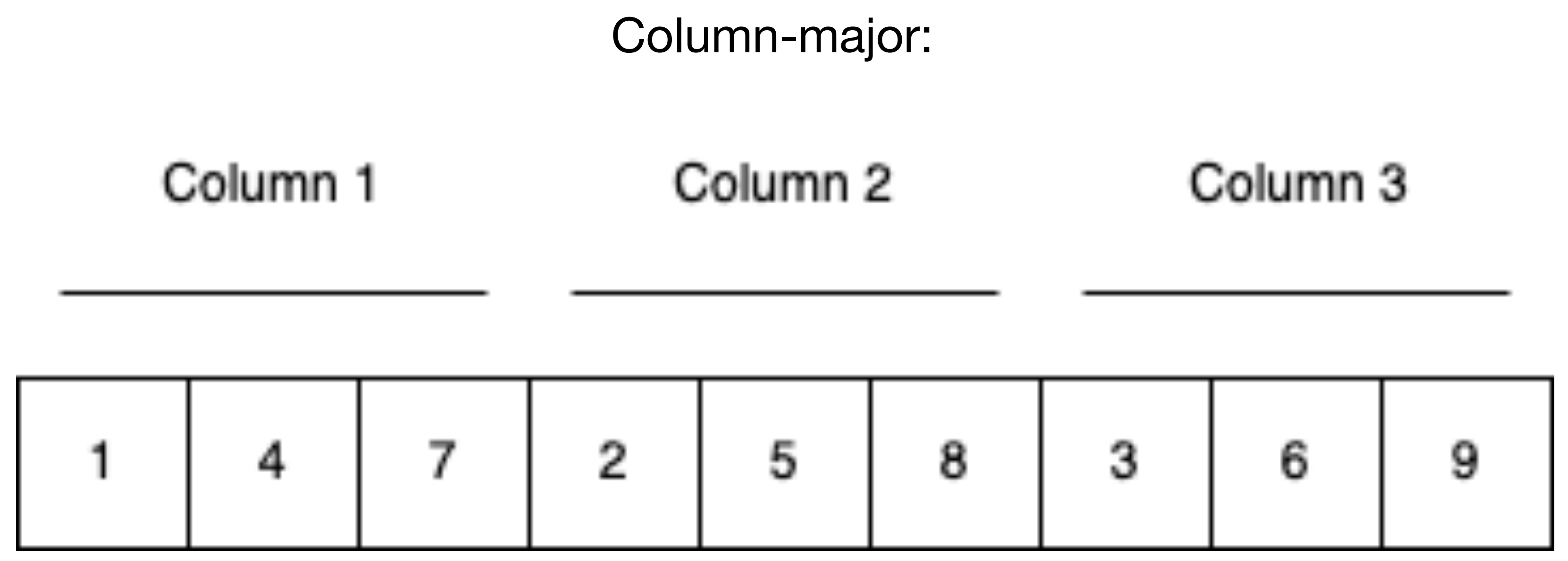
Borrowed from slides: Memory Hierarchy and Caching

Array layouts: row-major and column-major

- **Row-major order**: rows are contiguous in memory
    - Used by C
- **Column-major order**: columns are contiguous:
    - Used by MATLAB

Example: `A[3][3] = {{1,2,3}, {4,5,6},{7,8,9}}`



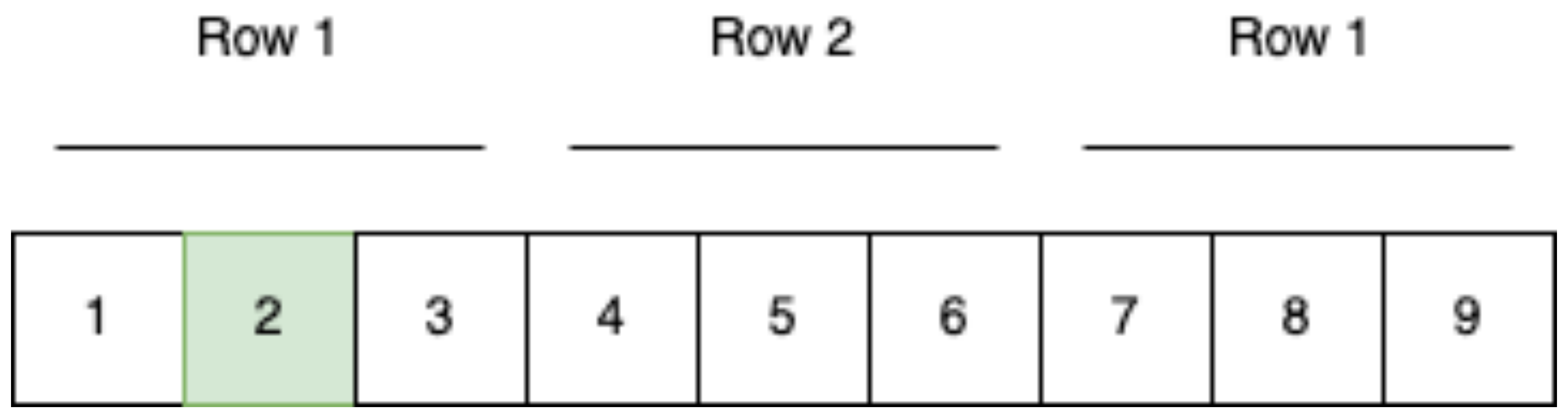


Example continued: row major

Example:

```
int A[3][3] =
        {{1,2,3},
         {4,5,6},
         {7,8,9}}
```

- Assume integers in the array are 4 bytes
- Assume we have a cache with blocks of size 12 (≈ 3*sizeof(int)), that can only hold 1 block at a time.
- What happens to a cold cache (a cache with nothing in it) where we access the row and second column of our 2D array, assuming row-major order is used? (`A[0][1]`)

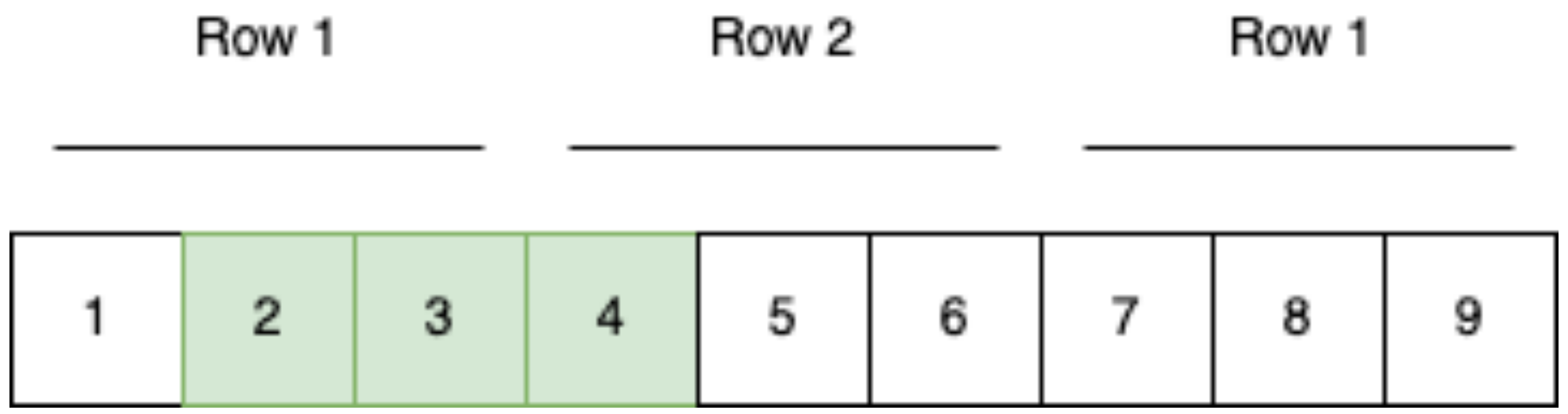


Example continued: row major

Example:

```
int A[3][3] =
        {{1,2,3},
         {4,5,6},
         {7,8,9}}
```

- Assume integers in the array are 4 bytes
- Assume we have a cache with blocks of size 12 (≈ 3*sizeof(int)), that can only hold 1 block at a time.
- What happens to a cold cache (a cache with nothing in it) where we access the row and second column of our 2D array, assuming row-major order is used? (`A[0][1]`)

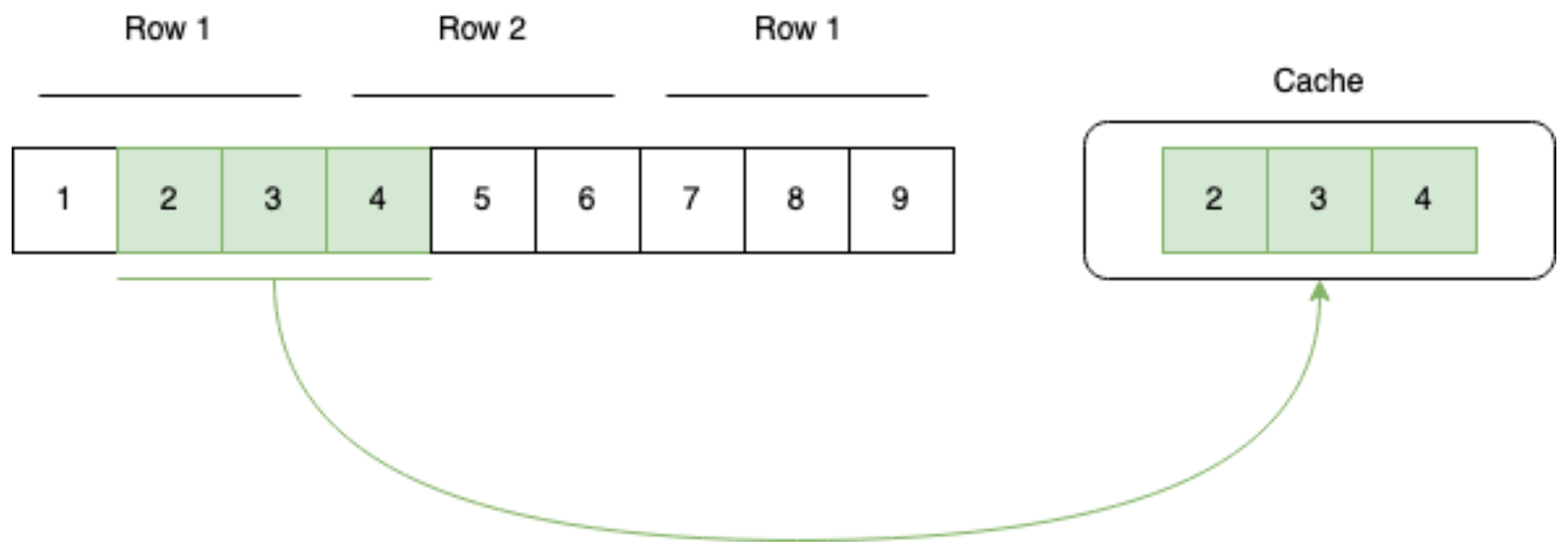


Example continued: row major

Example:

```
int A[3][3] =
    {{1,2,3},
     {4,5,6},
     {7,8,9}}
```

- Assume integers in the array are 4 bytes
- Assume we have a cache with blocks of size 12 (≈ 3*sizeof(int)), that can only hold 1 block at a time.
- What happens to a cold cache (a cache with nothing in it) where we access the row and second column of our 2D array, assuming row-major order is used? (`A[0][1]`)

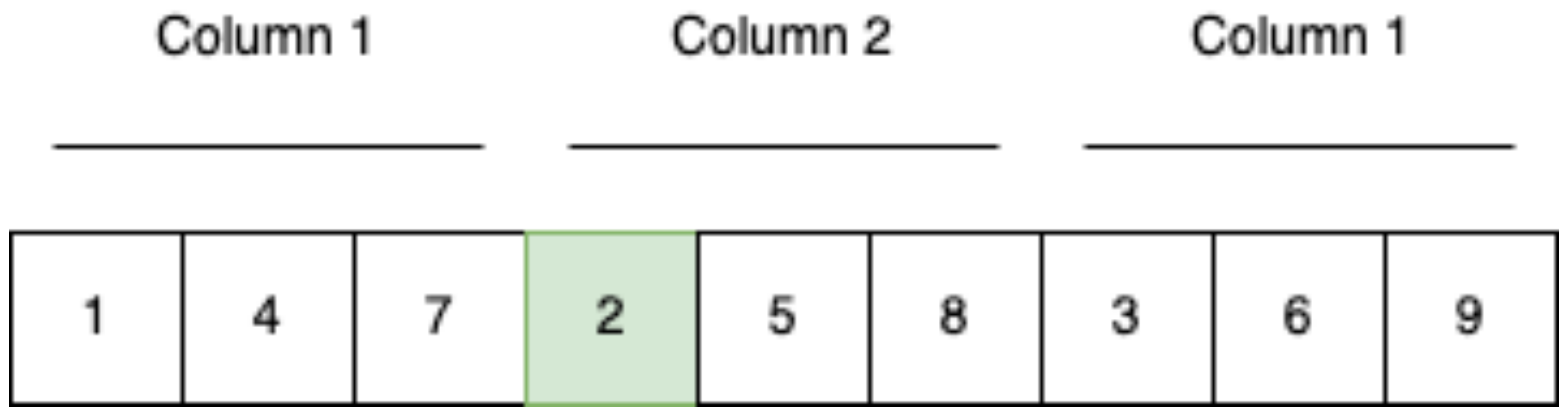


Example continued: column-major

Example:

```
int A[3][3] =
     {{1,2,3},
      {4,5,6},
      {7,8,9}}
```

- Assumptions are the same as the last slide
- What happens to a cold cache where we access the first row and second column of our 2D array, assuming column-major order is used? (`A[0][1]`)

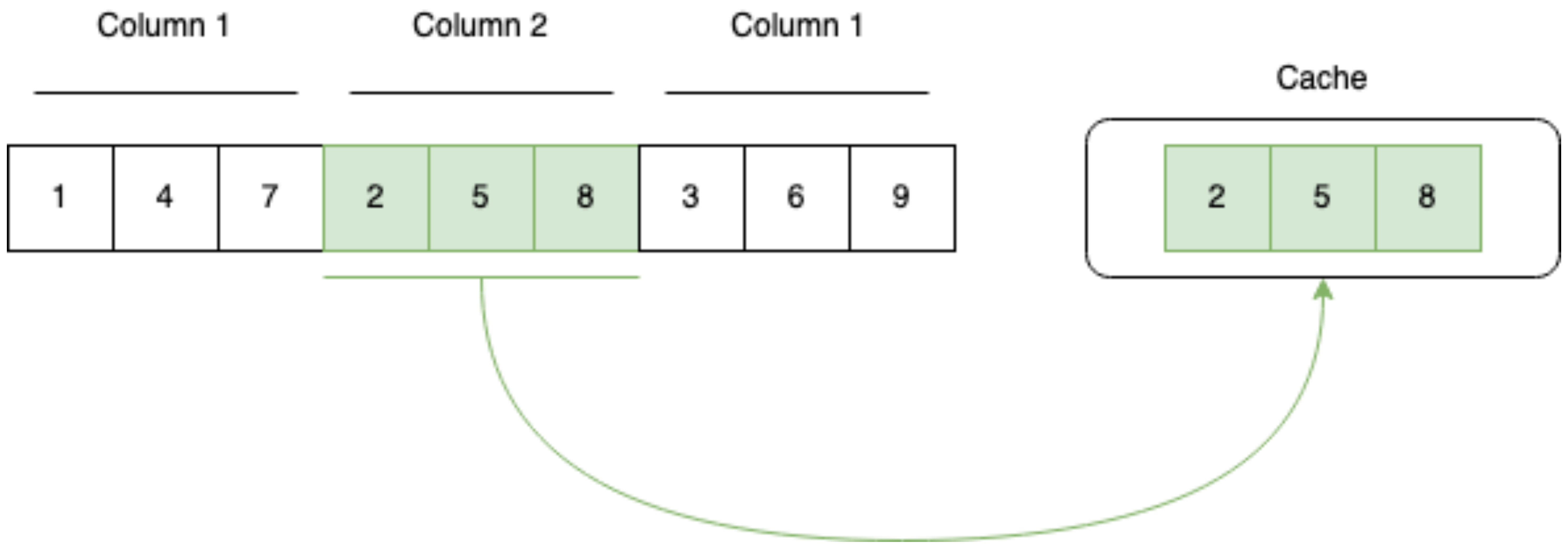


Example continued: column-major

Example:

```
int A[3][3] =
        {{1,2,3},
         {4,5,6},
         {7,8,9}}
```

- Assumptions are the same as the last slide
- What happens to a cold cache where we access the first row and second column of our 2D array, assuming column-major order is used? (`A[0][1]`)



Locality

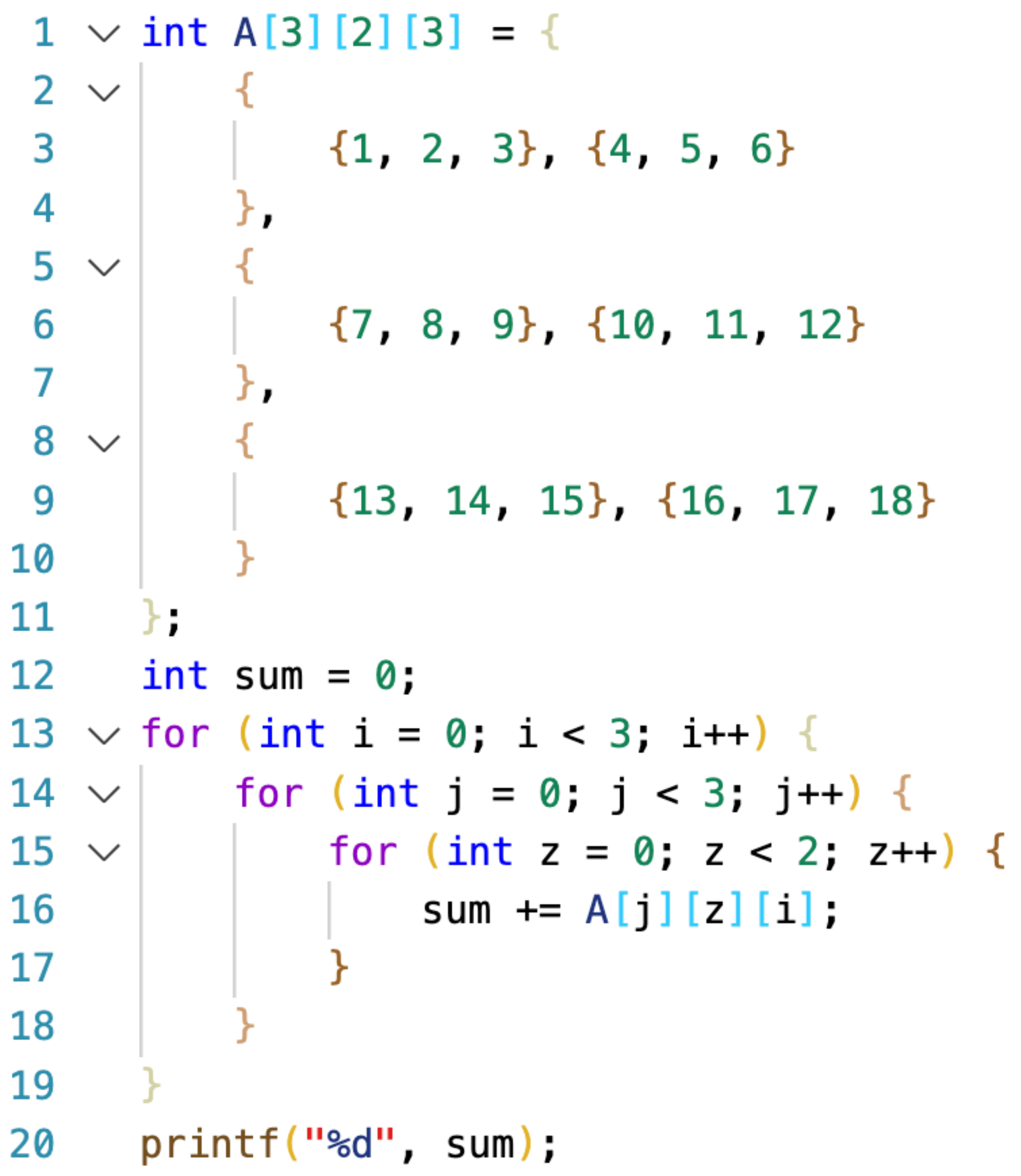
- **Spatial locality**: Data tends to be accessed at the same place as recently accessed memory.
- **Temporal locality**: Data that has recently been accessed, tends to be accessed again soon.
- **Stride**: How large are the jumps between memory accesses?

Locality: Example 1

```c
int A[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {6, 7, 8}
};
int sum = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum += A[i][j];
    }
}
printf("%d", sum);
```

- Stride:
  - row-major = 1
  - column-major = 3
- Spatial locality:
  - row-major = Better (we make jumps of 1 elements)
  - column-major = Worse (we make jumps of 3 elements)
- Temporal locality: `sum`, `i` and `j`

Locality: Example 2

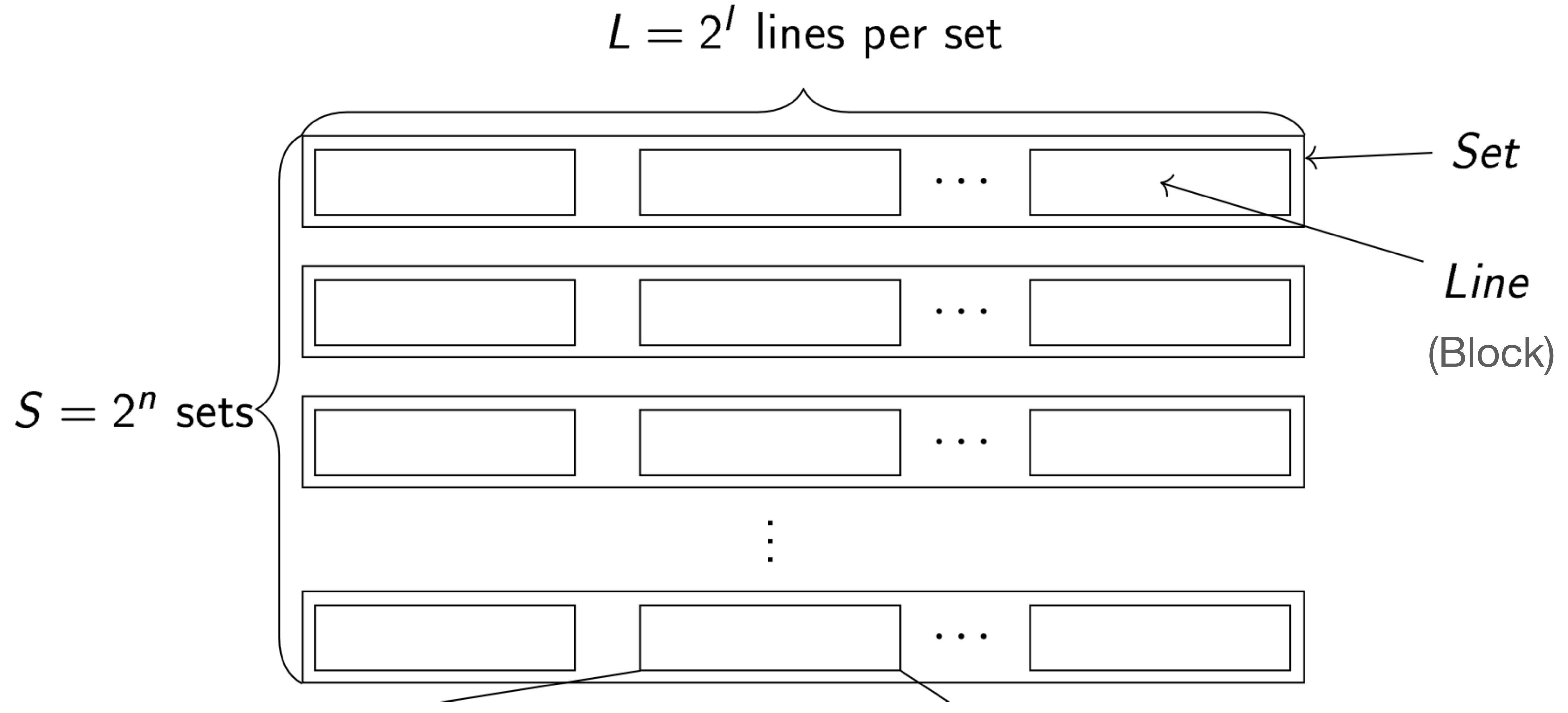


```c
int A[3][2][3] = {
    {
        {1, 2, 3}, {4, 5, 6}
    },
    {
        {7, 8, 9}, {10, 11, 12}
    },
    {
        {13, 14, 15}, {16, 17, 18}
    }
};
int sum = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int z = 0; z < 2; z++) {
            sum += A[j][z][i];
        }
    }
}
printf("%d", sum);
```

- Is this the most optimal access order for either row-major or column-major?
  - Row-major: No, `i` that is used to access elements in the first dimension is incremented last
  - Column-major: No, `j`, that accesses 2-D arrays in the third dimension is not incremented first.
- What is the most optimal arrangement of the for-loops (from outer to inner)?
  - Row-major: `j`, `z`, `i`
  - Column-major: `i`, `z`, `j`

Cache organisation

- **N-way set associative**:
  - N: how many blocks there are in each set.
- **Direct mapped/**One-way set associative: one block for each set, so each memory location is mapped to one location in the cache.
- **Fully associative**: cache is one set, a memory location can be mapped to anywhere in the cache.
  - Example: If the cache has space for 8 blocks, and it is fully associative, it would be an 8-way set associative.



Borrowed from slides: Memory Hierarchy and Caching

Address translation

- Order in bits left to right: tag, set index, block offset.
- Offset bits (b): how many bits do you need to represent a block?

$$b = \log_2(blocksize\_in\_bytes)$$

- How many sets are in an n-way associative cache?

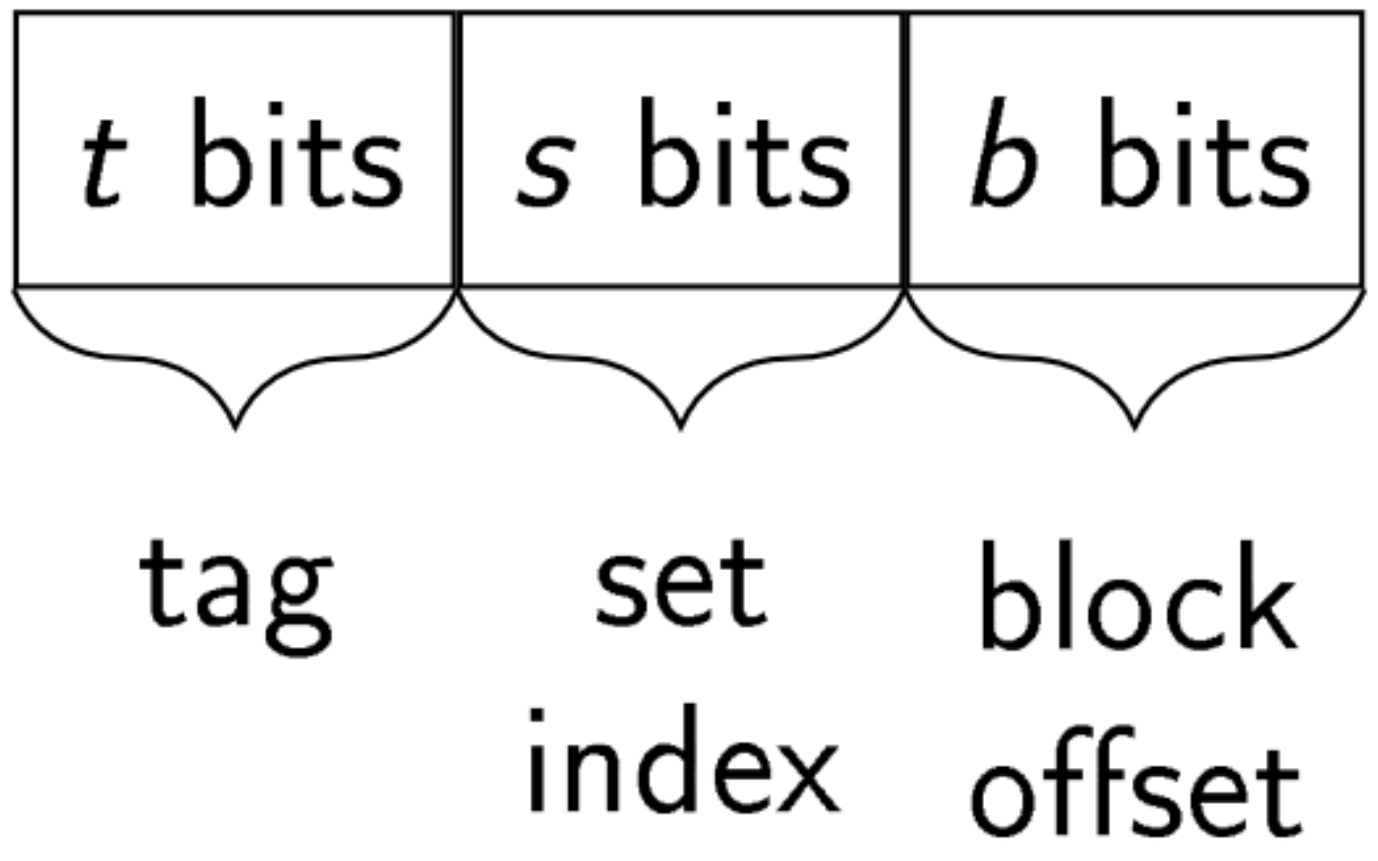
$$set\_count = \frac{cachesize\_in\_bytes}{n \cdot blocksize\_in\_bytes}$$

- Set index bits (s): how many bits do you need to represent the sets?

$$s = \log_2(set\_count) = \log_2(\frac{cachesize\_in\_bytes}{n \cdot blocksize\_in\_bytes})$$

- Tag: the rest of the address: $t = bits\_in\_address - (s + b)$

Address components



Borrowed from slides: Memory Hierarchy and Caching

Unit conversion

- Remember to convert between the correct units:
  - Kibibyte (KiB) = 1024 bytes
  - Mebibyte (MiB) = 1024 KiB
  - kb = kbit = kilobit = 125 bytes
- Memory is byte-addressed, so it is a good idea to convert everything to bytes.

Exam 2022-2023 question 1.1.1

1.1 Data Cache and Locality (about 8 %)

bits_in_address= 32 cachesize = 16KiB n = 4

A byte-addressed machine with 32-bit addresses is equipped with a 16 kibibyte 4-way set-associative cache with a block size of 32 bytes. The cache uses "least-recently-used replacement policy".

blocksize = 32 bytes

Question 1.1.1: The address is separated into the following fragments when the cache is accessed

- block offset,
- cache tag, and
- set index.

What is bit-size of each and how are they ordered?

	31		0
Address fragment	Cache tag	Set index	Block offset
Bit-size	20	7	5

$$b = \log_2(blocksize\_in\_bytes) = \log_2(32) = 5$$

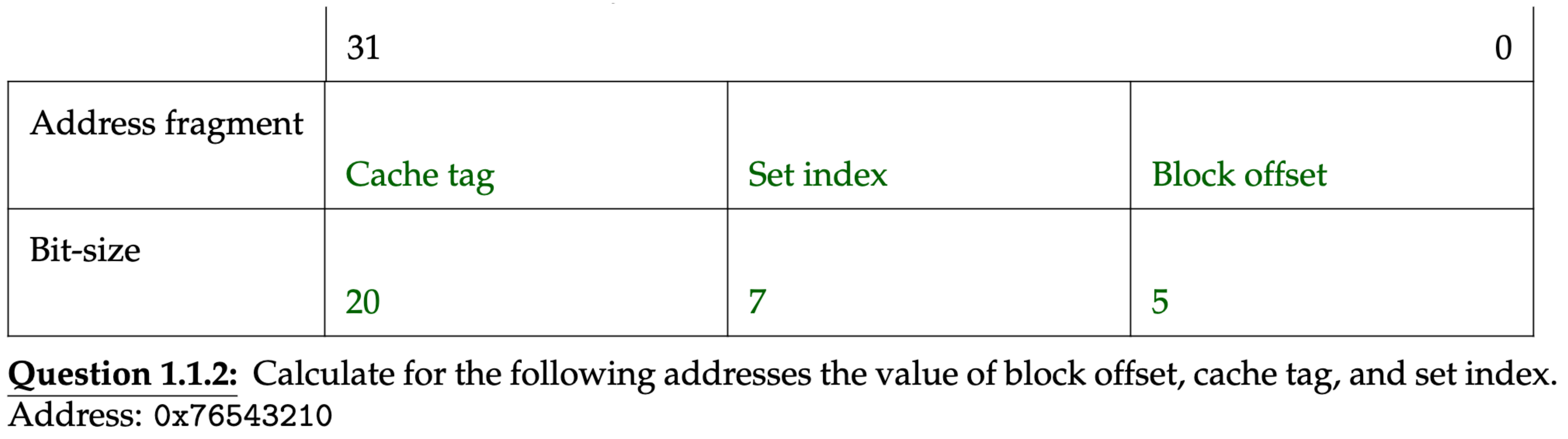
$$cachesize\_in\_bytes = 16\text{ KiB} * 1024 = 16384\text{ bytes}$$

$$set\_count = \frac{cachesize\_in\_bytes}{n \cdot blocksize\_in\_bytes} = \frac{16384}{4 \cdot 32} = 128$$

$$s = \log_2(set\_count) = \log_2(128) = 7$$

$$t = bits\_in\_address - (s + b) = 32 - (7 + 5) = 20$$

Exam 2022-2023 question 1.1.2



	31		0
Address fragment	Cache tag	Set index	Block offset
Bit-size	20	7	5

Question 1.1.2: Calculate for the following addresses the value of block offset, cache tag, and set index.
Address: `0x76543210`

Cache tag = 0x76543 Block offset = 0x10

0x76543210 = 0111 0110 0101 0100 0011 | 0010 000 | 1 0000

Set index = 0x10

Replacement schemes

- **LRU (least recently used)**: the block that is replaced in the set is the one that has been unused for the longest time.
- **Random**: candidate blocks are randomly selected, possibly using hardware assistance.
- Assuming LRU:
  - Find the set index of the address.
  - Search for the block in the set by comparing tags.
  - If it is found = cache hit: update this block such that it is visible that it was just used.
  - On cache miss:
    - If there are one or more blocks that are not valid (check the valid bit), replace one of these blocks with the new block.
    - If all blocks in the set are valid, replace the least recently used block, and update the new block such that it is visible that it was just used.

Exam 2022-2023 question 1.1.3

1.1 Data Cache and Locality (about 8 %)

bits_in_address= 32 cachesize = 16KiB n = 4

A byte-addressed machine with 32-bit addresses is equipped with a 16 kibibyte 4-way set-associative cache with a block size of 32 bytes. The cache uses "least-recently-used replacement policy".

blocksize = 32 bytes Cache uses LRU

Question 1.1.1: The address is separated into the following fragments when the cache is accessed

- block offset,
- cache tag, and
- set index.

What is bit-size of each and how are they ordered?

	31		0
Address fragment	Cache tag	Set index	Block offset
Bit-size	20	7	5

$$b = \log_2(blocksize\_in\_bytes) = \log_2(32) = 5$$

$$cachesize\_in\_bytes = 16 \text{ KiB} * 1024 = 16384 \text{ bytes}$$

$$set\_count = \frac{cachesize\_in\_bytes}{n \cdot blocksize\_in\_bytes} = \frac{16384}{4 \cdot 32} = 128$$

$$s = \log_2(set\_count) = \log_2(128) = 7$$

$$t = bits\_in\_address - (s + b) = 32 - (7 + 5) = 20$$

Exam 2022-2023 question 1.1.3

Question 1.1.3: On the machine, the following stream of cache accesses are performed (read from top to bottom). Indicate for each of the address references:

- the set index,
- if the cache access is a miss or a hit, and
- the cache tags in LRU-order that are in the affected set after the access.

Only write tags of the affected set

Addresses are given in hexa-decimal notation. Assume the cache is cold on entry.

Cache is cold

Reference: 0x0100004 = 0000 0001 0000 0000 | 0000 000|0 0100

Tag = 0x100

Set index = 0

Cache is cold (empty), so it is a miss.

```
Cache { 0: {0x100} }
```

Reference	Set index	Hit/Miss	State of Tags
0x0100004	0	miss	0x100

Exam 2022-2023 question 1.1.3

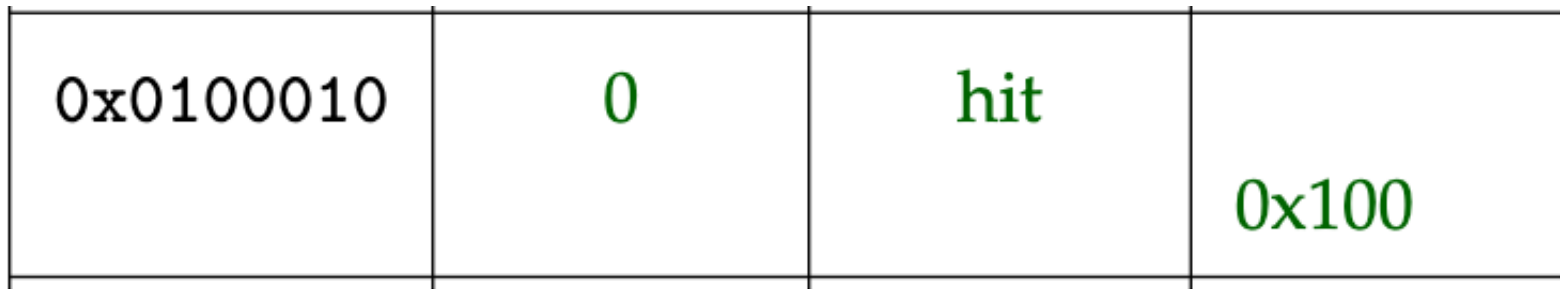
```
Cache before { 0: {0x100} }
```

Reference: 0x0100010 = 0000 0001 0000 0000 | 0000 0001 | 0000

Tag = 0x100 Set index =0

Cache hit, state of cache does not change:

```
Cache after { 0: {0x100} }
```



0x0100010	0	hit	0x100

Exam 2022-2023 question 1.1.3

```
Cache before { 0: {0x100} }
```

Reference: 0x0010008 = 0000 0000 0001 0000 | 0000 000 | 0 1000

Tag = 0x10 Set index =0

Cache miss, set 0 is not full, so tag 0x10 is added to set 0, and 0x10 is first in LRU-order:

```
Cache after { 0: {0x10, 0x100} }
```

Important tip!

We write the tags in the set in LRU-order, such that it easy to keep track

Exam 2022-2023 question 1.1.3

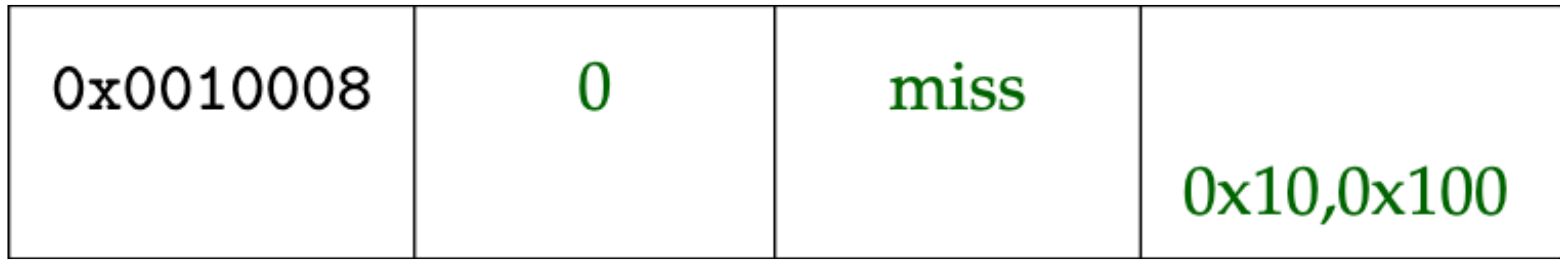
```
Cache before { 0: {0x100} }
```

Reference: 0x0010008 = 0000 0000 0001 0000 | 0000 0000 | 1000

Tag = 0x10 Set index =0

Cache miss, set 0 is not full, so tag 0x10 is added to set 0, and 0x10 is first in LRU-order:

```
Cache after { 0: {0x10, 0x100} }
```



0x0010008	0	miss	0x10,0x100

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x10, 0x100} }
```

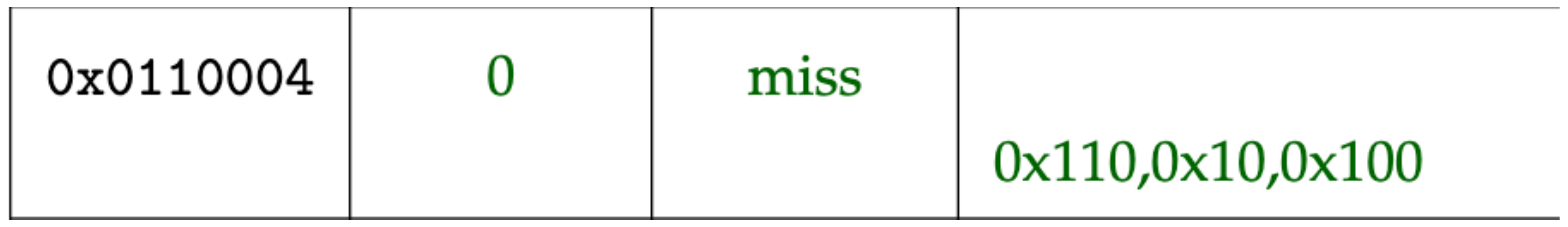
Reference: 0x0110004 = 0000 0001 0001 0000 0000 0000 0100

Tag = 0x110 Set index =0

Cache miss, set 0 is not full, so tag 0x110 is added to set 0, and LRU-order is changed:

```
Cache after:

{ 0: {0x110, 0x10, 0x100} }
```



0x0110004	0	miss	0x110,0x10,0x100

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x110, 0x10, 0x100} }
```

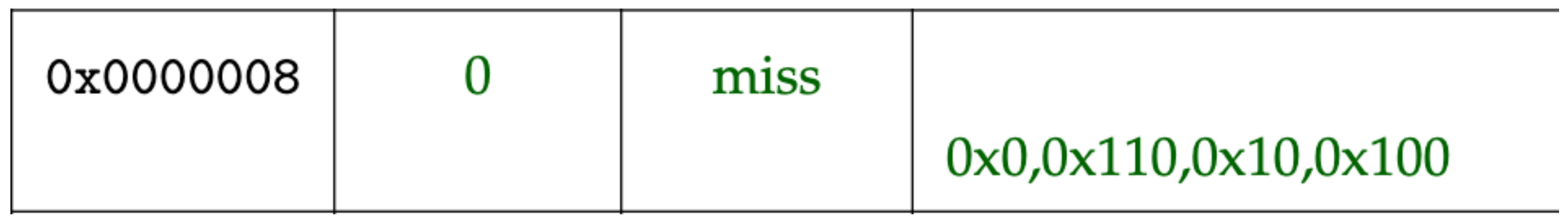
Reference: 0x0000008 = 0000 0000 0000 0000 0000 0000 1000

Tag = 0x0 Set index = 0

Cache miss, set 0 is not full, so tag 0x0 is added to set 0, and LRU-order is changed:

```
Cache after:

{ 0: {0x0, 0x110, 0x10, 0x100} }
```



0x0000008	0	miss	0x0,0x110,0x10,0x100

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x0, 0x110, 0x10, 0x100} }
```

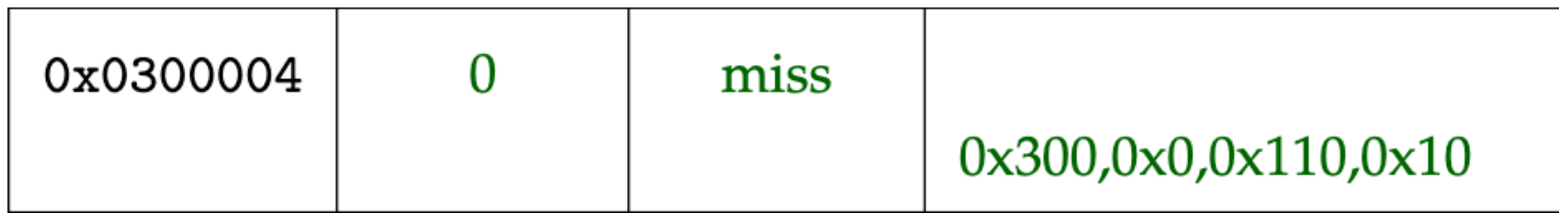
Reference: 0x0300004 = 0000 0011 0000 0000 | 0000 0000 | 0100

Tag = 0x300 Set index = 0

Cache miss, set 0 is full (it can only hold 4 blocks), so tag 0x100 is replaced with 0x300 since it is last in the LRU-order list.

```
Cache after:

{ 0: {0x300, 0x0, 0x110, 0x10} }
```



0x0300004	0	miss	0x300,0x0,0x110,0x10

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x300, 0x0, 0x110, 0x10} }
```

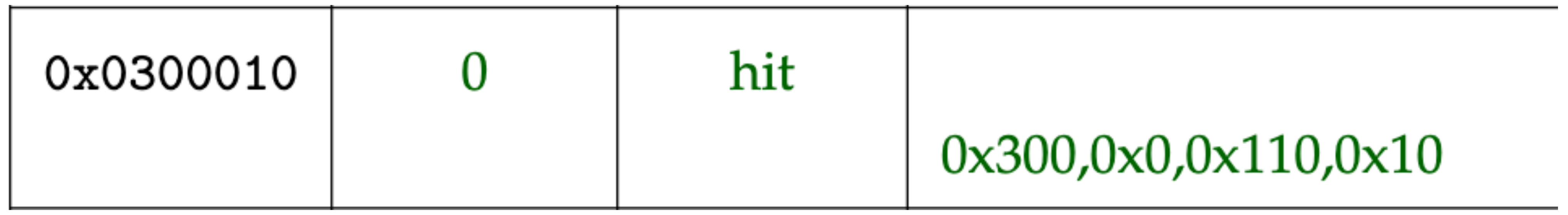
Reference: 0x0300010 = 0000 0011 0000 0000 0000 0001 0000

Tag = 0x300 Set index = 0

Cache hit, but tag 0x300 is already the first in the LRU-order list, so nothing is changed:

```
Cache after:

{ 0: {0x300, 0x0, 0x110, 0x10} }
```



0x0300010	0	hit	0x300,0x0,0x110,0x10

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x300, 0x0, 0x110, 0x10} }
```

Reference: 0x0110008 = 0000 0001 0001 0000 | 0000 000 | 0 1000

Tag = 0x110 Set index = 0

Cache hit, LRU-order is updated in set 0 so tag 0x110 is first:

```
Cache after:

{ 0: {0x110, 0x300, 0x0, 0x10} }
```

0x0110008	0	hit	0x110,0x300,0x0,0x10

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x110, 0x300, 0x0, 0x10} }
```

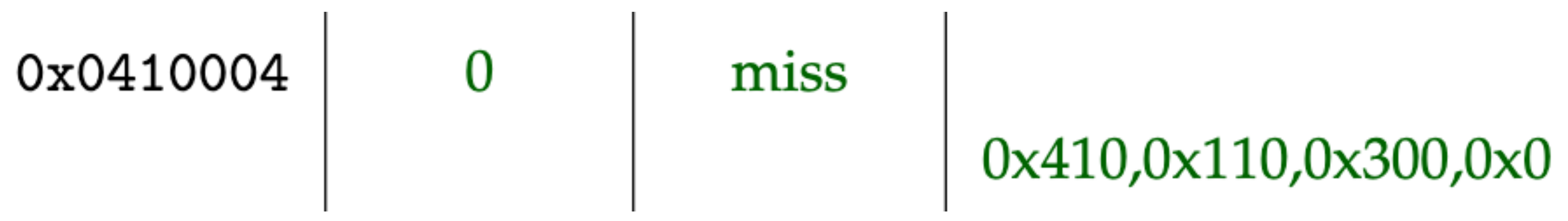
Reference: 0x0410004 = 0000 0100 0001 0000 | 0000 000 | 0 0100

Tag = 0x410 Set index = 0

Cache miss, set 0 is full, so tag 0x10 is replaced with 0x410 since it is last in the LRU-order list

```
Cache after:

{ 0: {0x410, 0x110, 0x300, 0x0} }
```



0x0410004	0	miss	0x410,0x110,0x300,0x0

Exam 2022-2023 question 1.1.3

```
Cache before: { 0: {0x410, 0x110, 0x300, 0x0} }
```

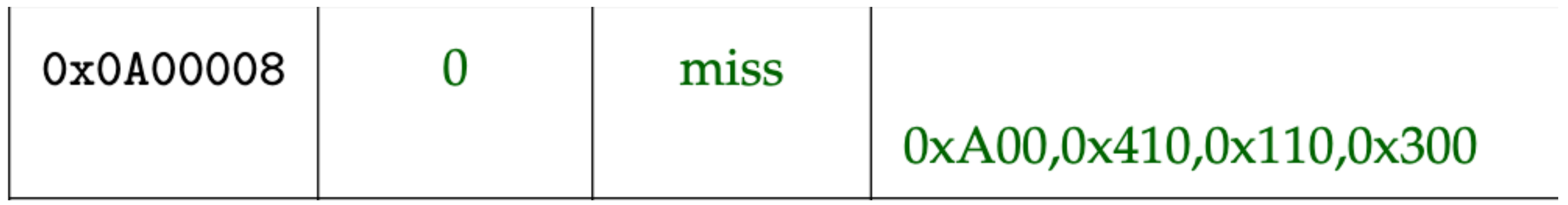
Reference: 0x0A00008 = 0000 1010 0000 0000 | 0000 0000 | 1000

Tag = 0xA00   Set index = 0

Cache miss, set 0 is full, so tag 0x0 is replaced with 0xA00 since it is last in the LRU-order list

```
Cache after:

{ 0: {0xA00, 0x410, 0x110, 0x300} }
```



0x0A00008	0	miss	0xA00,0x410,0x110,0x300

Exam 2022-2023 question 1.1.3: own example

`Cache before: { 0: {0xA00, 0x410, 0x110, 0x300} }`

Reference: 0xFFFFFFFF = 1111 1111 1111 1111 1111 1111 1111 1111

Tag = 0xFFFFF

Set index = 127

Cache miss, **but now in a new set:** 127. We add the block in the new set:

```
Cache after:

{0: {0xA00, 0x410, 0x110, 0x300},
  127: {0xFFFFF} }
```

Remember: only write accessed set in LRU-order!

0xFFFFFFFF	127	Miss	0xFFFFF

Exam 2022-2023 question 1.1.3: own example

```
Cache before:

{ 0: {0xA00, 0x410, 0x110, 0x300}, 127: {0xFFFFF} }
```

Reference: 0x4206924B = 0100 0010 0000 0110 1001 0010 0100 1011

Tag = 0x42069

Set index = 18

Cache miss, new set 18, we add the block in the new set:

```
Cache after:

{0: {0xA00, 0x410, 0x110, 0x300},

  18: {0x42069}

  127: {0xFFFFF} }
```

Remember: only write accessed set in LRU-order!

0x42069	18	Miss	0x42069

Product placement!

Some friends of mine, Phillip and Mahmood, are doing a Project in Practice, and they are creating learning tools for Compsys-related topics.

So if you have run out of Compsys exam sets, then you can try out the tools:

https://abdsecondhand.site/CACHE/dist/index.html for some cache exercises

https://abdsecondhand.site/VMAT/dist/index.html for virtual table exercises

Good luck with the exam!