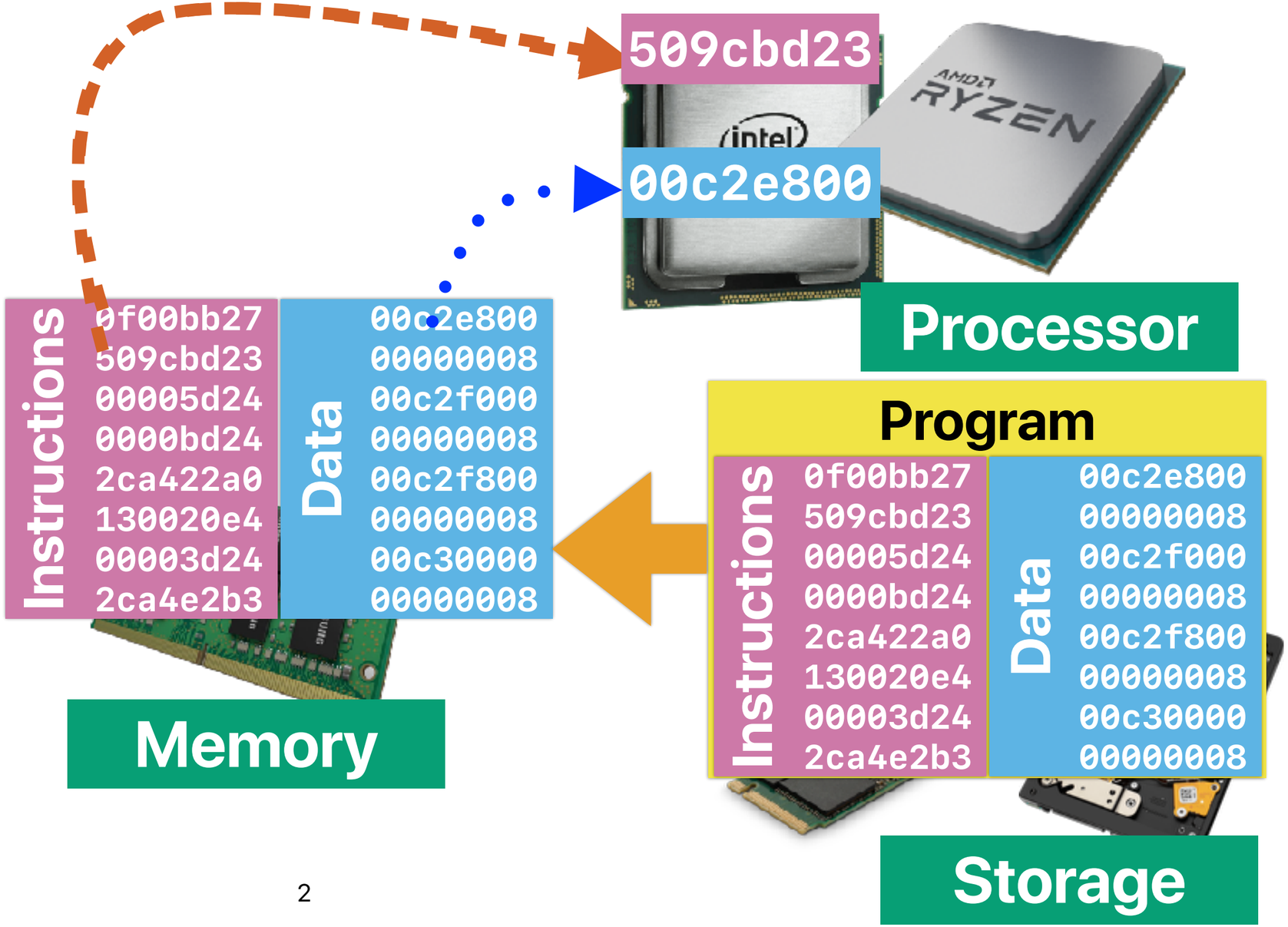
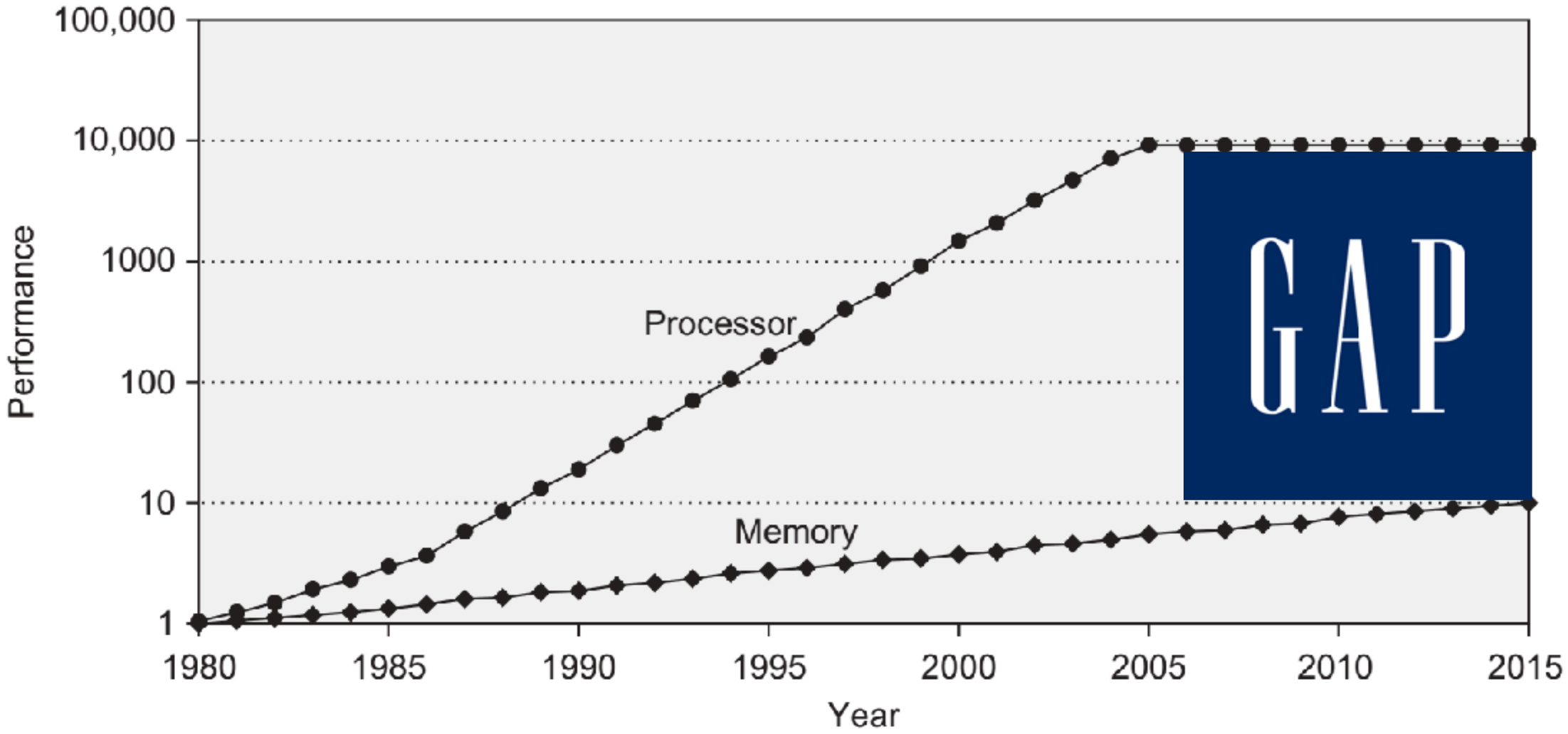
Memory Hierarchy (2): The A, B, Cs of Your Cache

Hung-Wei Tseng

Recap: von Neumman Architecture



Recap: Performance gap between Processor/Memory



Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

TBs

larger

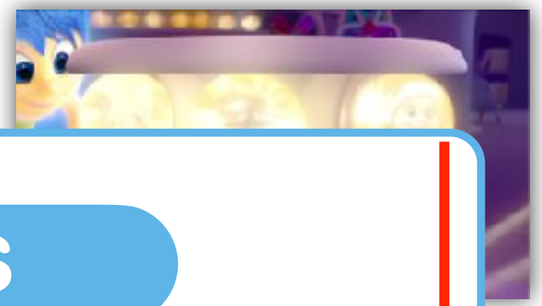
fastest

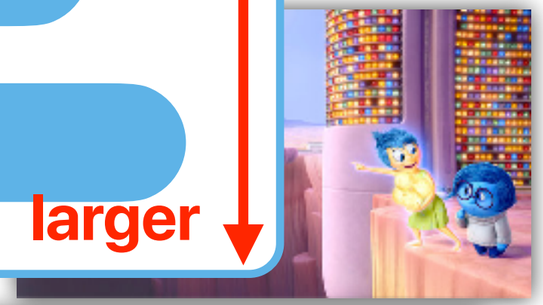
L1 $

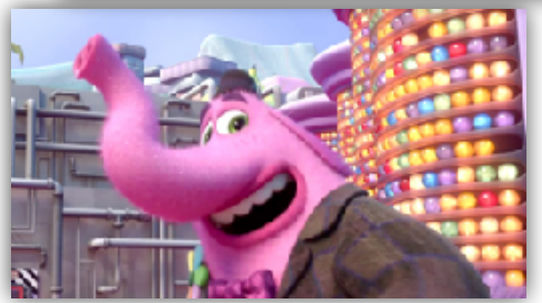
L2 $

L3 $

larger



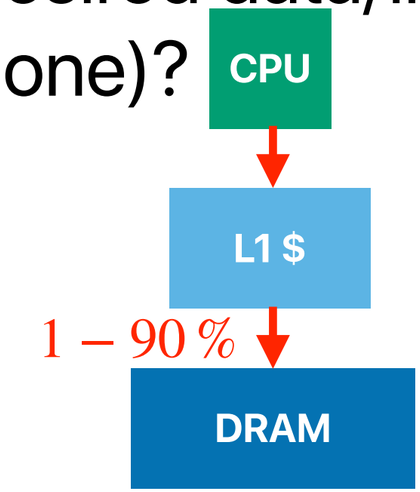




How can "memory hierarchy" help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture 90% of the desired data/instructions. what's the average CPI (pick the closest one)?

  A. 6
  B. 8
  C. 10
  D. 12
  E. 67



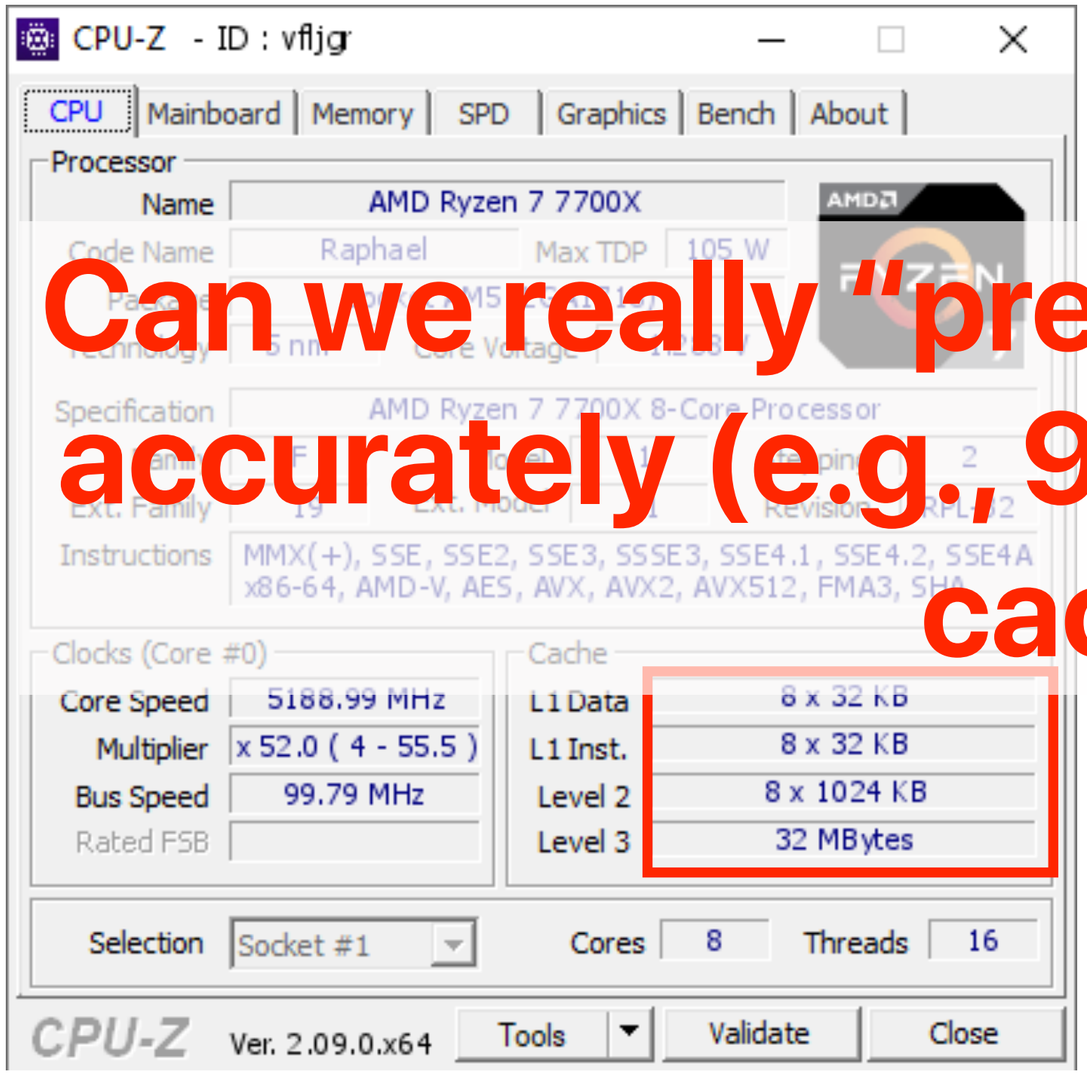
$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

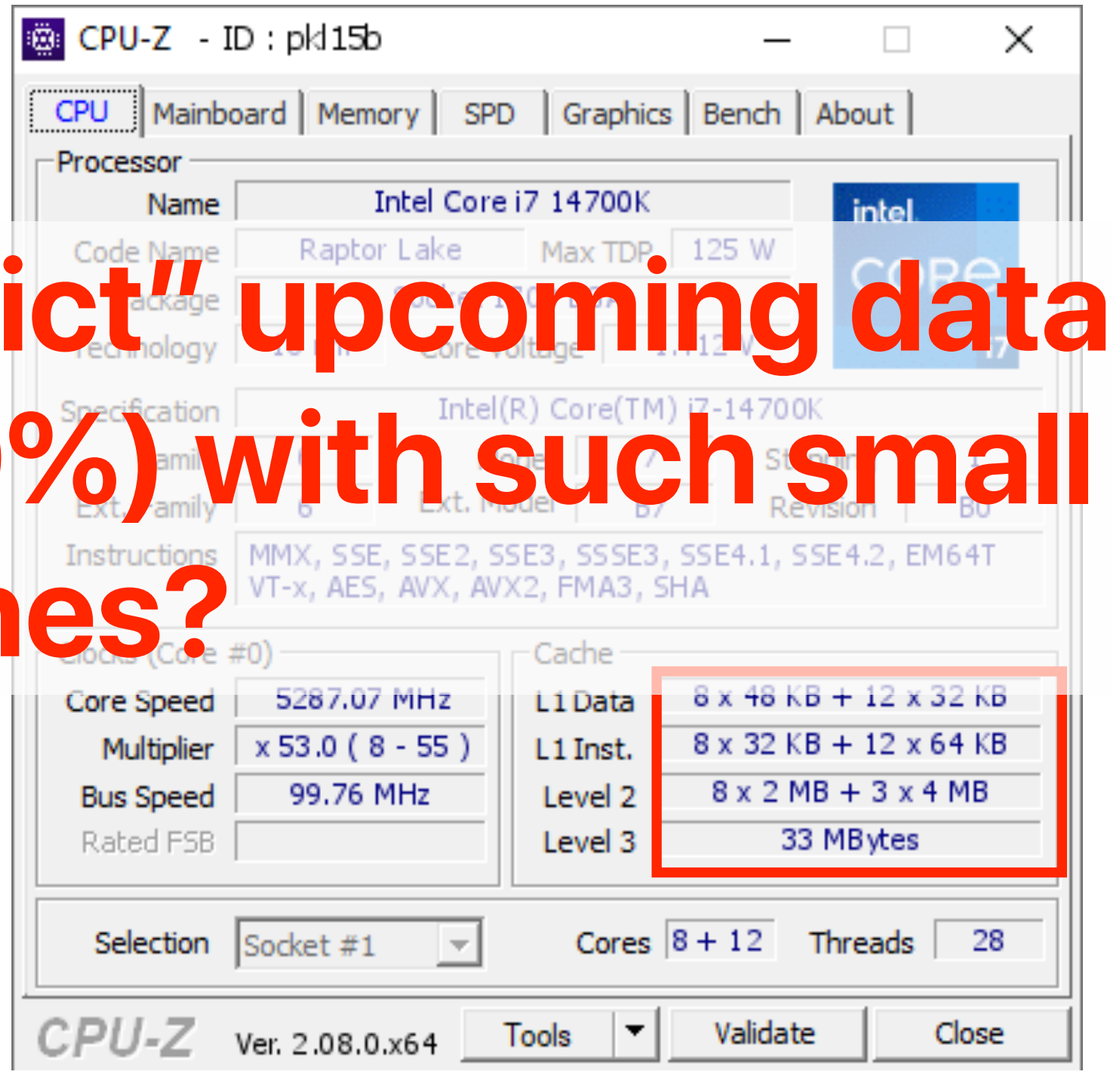
$$Each\ \$\ access = \frac{0.5}{0.25} = 2\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times [2 + (1 - 90\%) \times 55] + 20\% \times [2 + (1 - 90\%) \times 55] = 10\ cycles$$

L1? L2? L3?





Can we really "predict" upcoming data accurately (e.g., 90%) with such small caches?

Outline

- The “predictable” code behavior
- Designing a cache that captures the predictability
- Estimating the code performance on cache

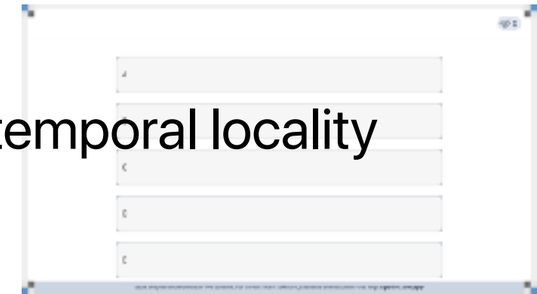
The predictability of your code

Locality of data

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint32_t i = 0; i < m; i++) {
     result = 0;
    for(uint32_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality



Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint32_t i = 0; i < m; i++) {
     result = 0;
    for(uint32_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

spatial locality:
matrix[0][0], matrix[0][1], matrix[0][2], ...
vector[0], vector[1], ... , vector[n]
temporal locality:
reuse of vector[0], vector[1], ... ,

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

Code also has locality

```
for(uint32_t i = 0; i < m; i++) {
    result = 0;
    for(uint32_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

```
i = 0;
while(i < m) {
    result = 0;
    j = 0;
    while(j < n) {
        a = matrix[i][j];
        b = vector[j];
        temp = a*b;
        result = result + temp;
    }
    output[i] = result;
    i++;
}
```

keep going to the next instruction — spatial locality

repeat many times — temporal locality!

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — the current data, and then the next one
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

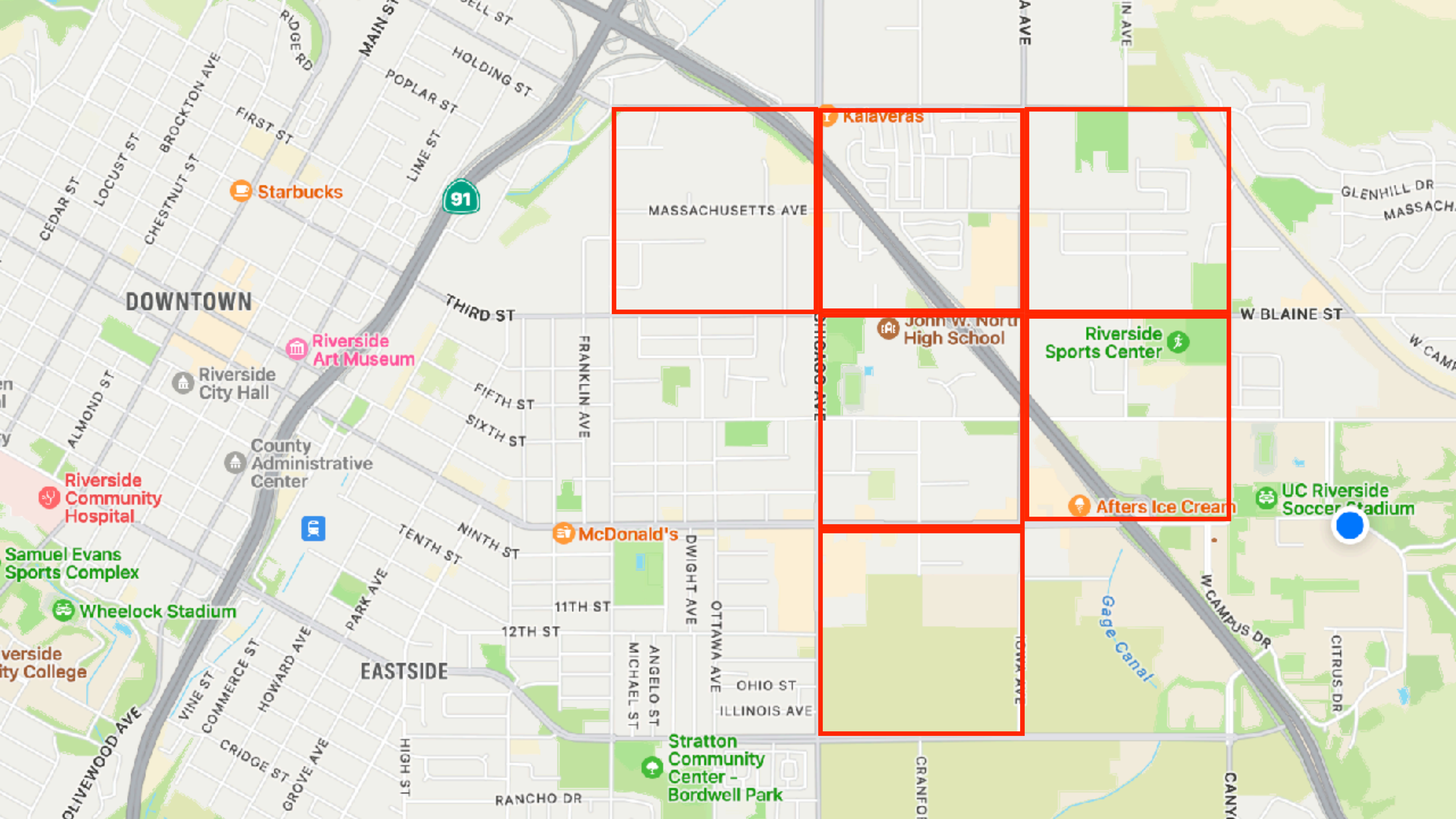
Most of time, your program is just visiting a very small amount of data/instructions within a given window

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

  We need to "cache consecutive memory locations" every time — the cache should store a "block" of code/data

- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

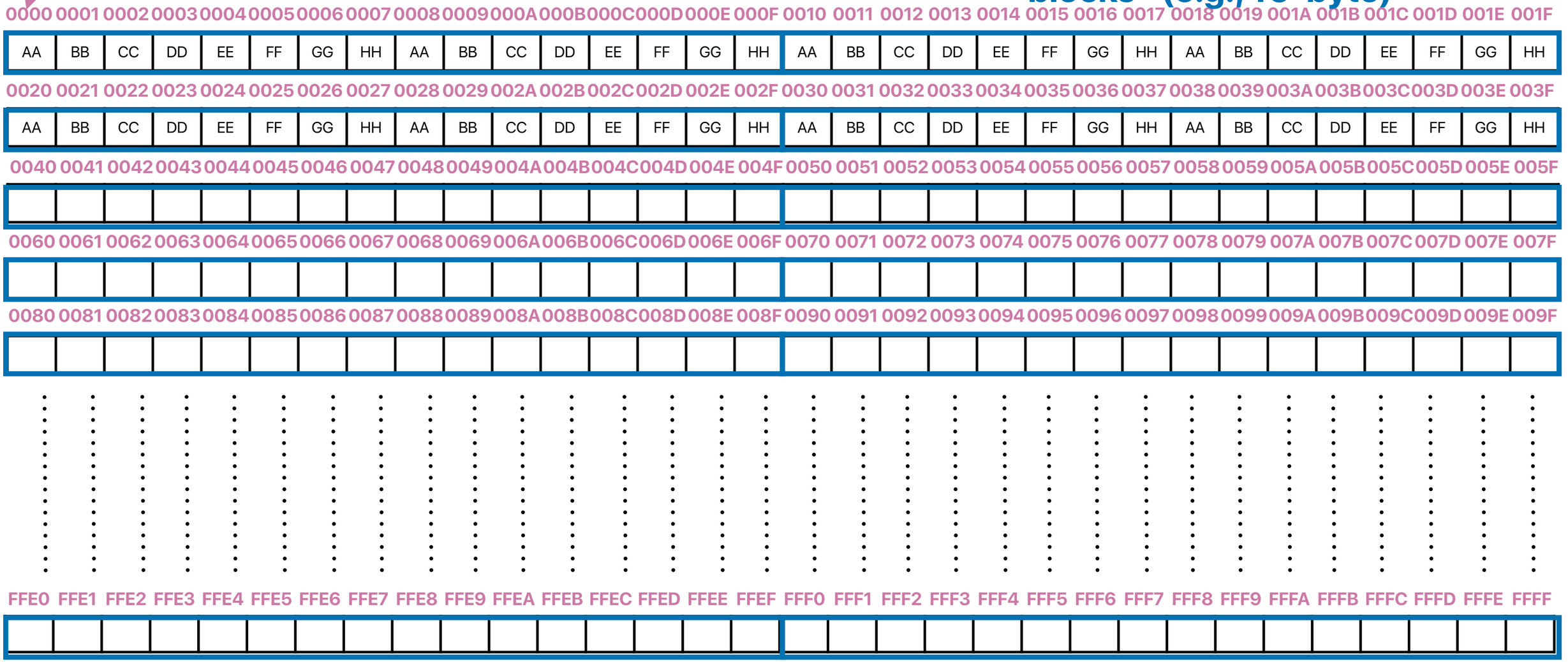


DOWNTOWN
EASTSIDE
Starbucks
Riverside Art Museum
Riverside City Hall
County Administrative Center
Riverside Community Hospital
Samuel Evans Sports Complex
Wheelock Stadium
McDonald's
Kalaveras
John W. North High School
Riverside Sports Center
Afters Ice Cream
UC Riverside Soccer Stadium
Stratton Community Center - Bordwell Park
91
MASSACHUSETTS AVE
THIRD ST
FIFTH ST
SIXTH ST
TENTH ST
NINTH ST
11TH ST
12TH ST
FRANKLIN AVE
DWIGHT AVE
OTTAWA AVE
OHIO ST
ILLINOIS AVE
MICHAEL ST
ANGELO ST
W BLAINE ST
GLENHILL DR
W CAMPUS DR
CITRUS DR
Gage Canal
MAIN ST
HOLDING ST
POPLAR ST
FIRST ST
LIME ST
BROCKTON AVE
CHESTNUT ST
LOCUST ST
CEDAR ST
ALMOND ST
PARK AVE
HOWARD AVE
COMMERCE ST
VINE ST
OLIVEWOOD AVE
CRIDGE ST
GROVE AVE
HIGH ST
RANCHO DR
RIDGE RD

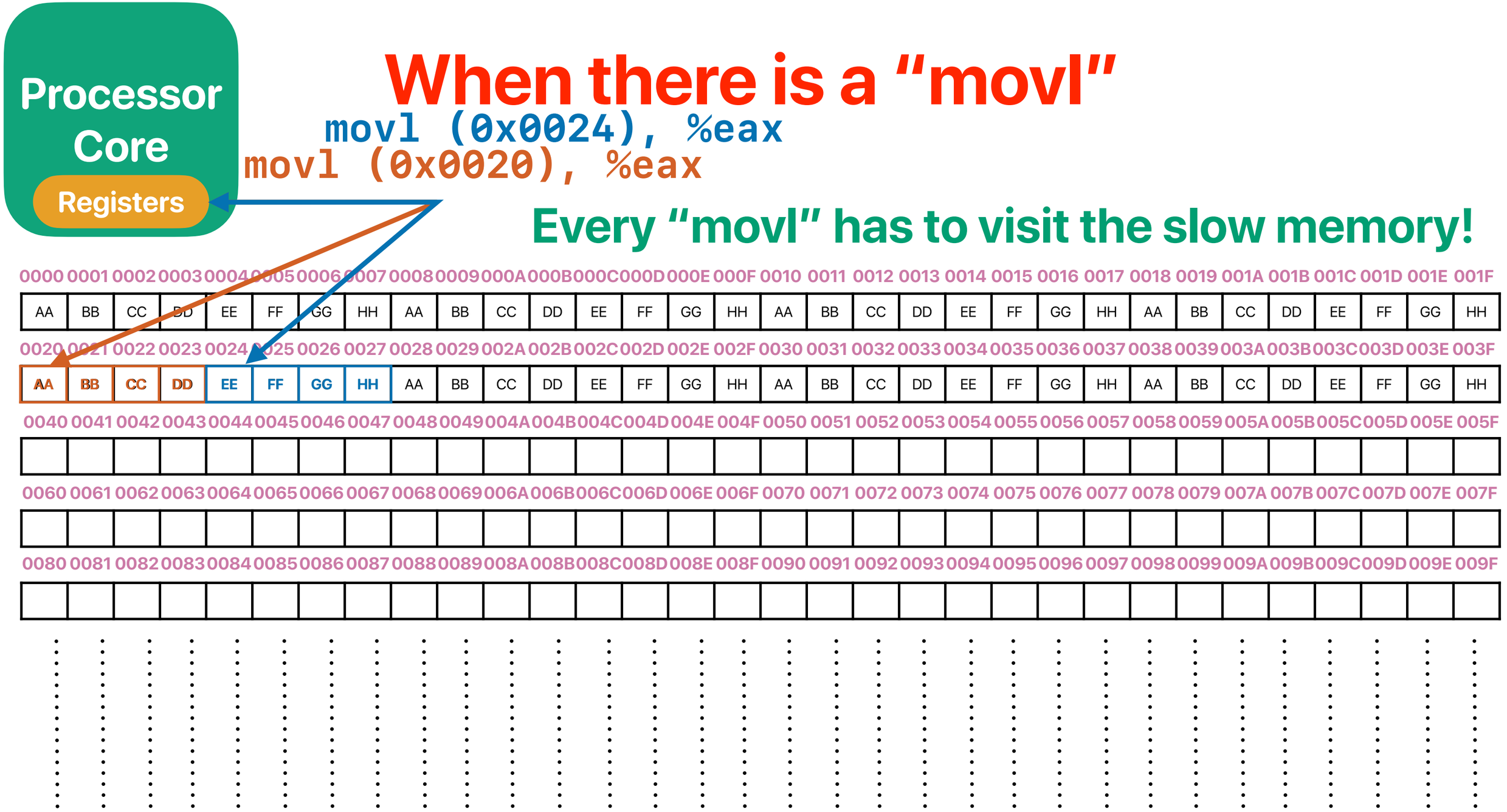
Block and the memory space

Each byte of memory location has an "address"

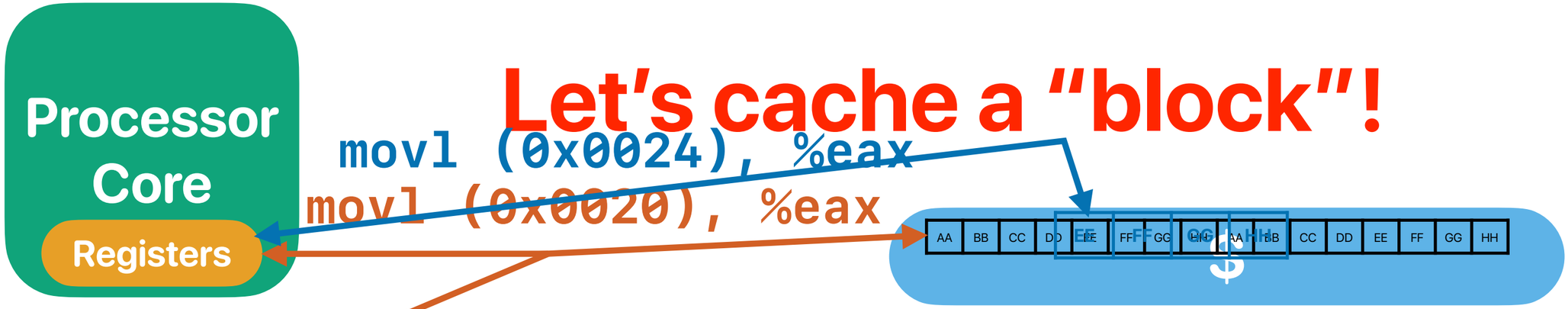
Partition the space with fixed-size "blocks" (e.g., 16-byte)



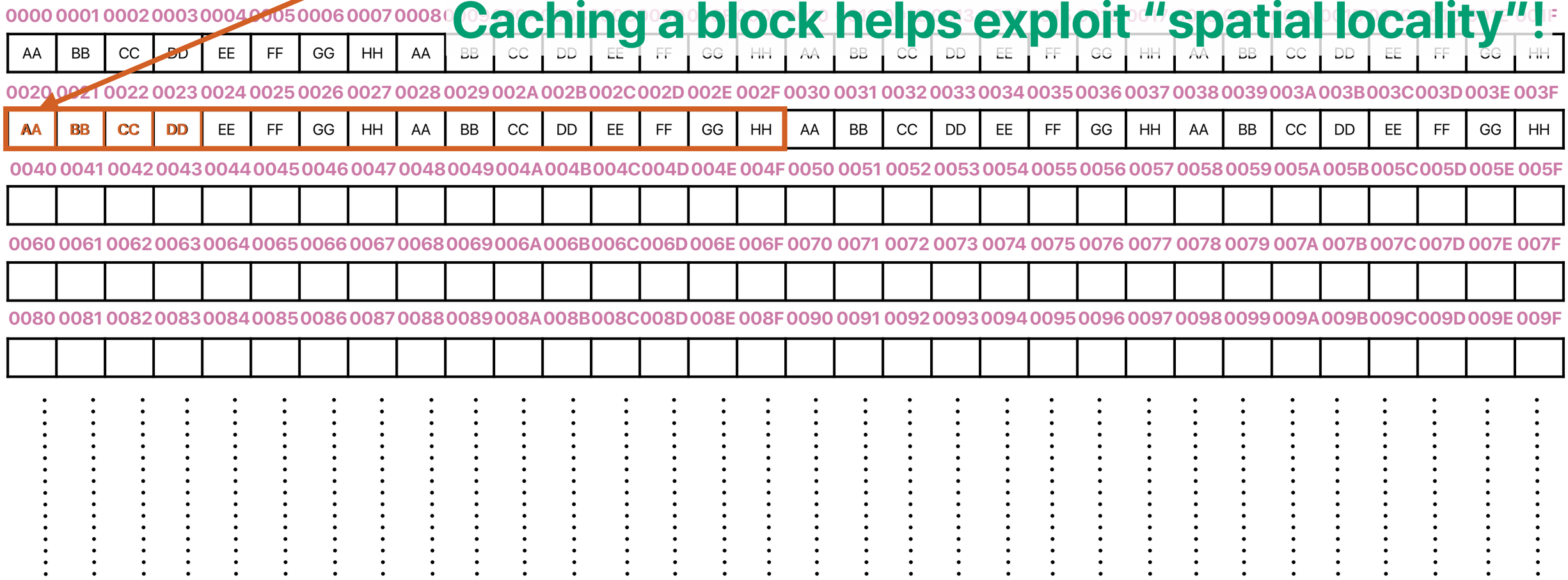
When there is a “movl”



Let's cache a "block"!



Caching a block helps exploit "spatial locality"!



Recap: Locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint32_t i = 0; i < m; i++) {
     result = 0;
    for(uint32_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
     output[i] = result;
 }
```

Simply caching one block isn't enough

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

  We need to "cache consecutive memory locations" every time

  — the cache should store a "block" of code/data

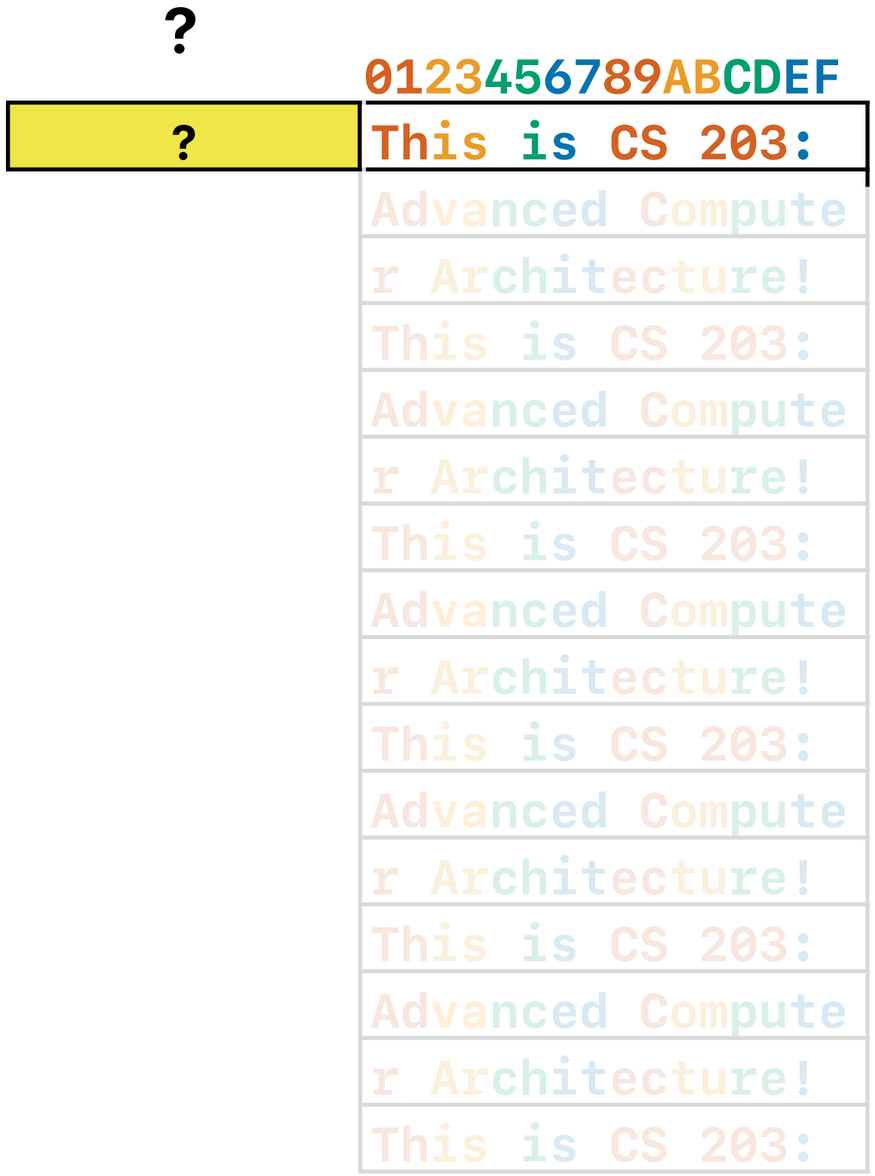
- Temporal locality — application revisit the same thing again and again

  We need to "cache frequently used memory blocks"

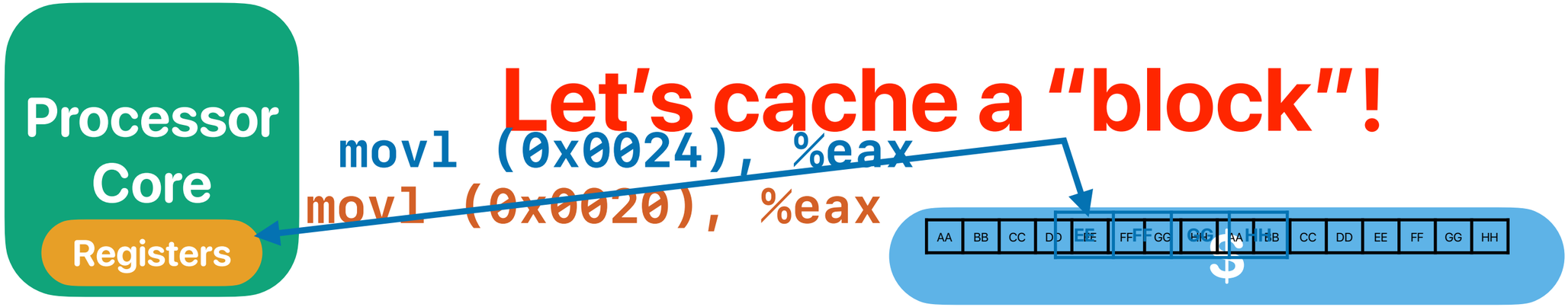
  — the cache should store a few blocks

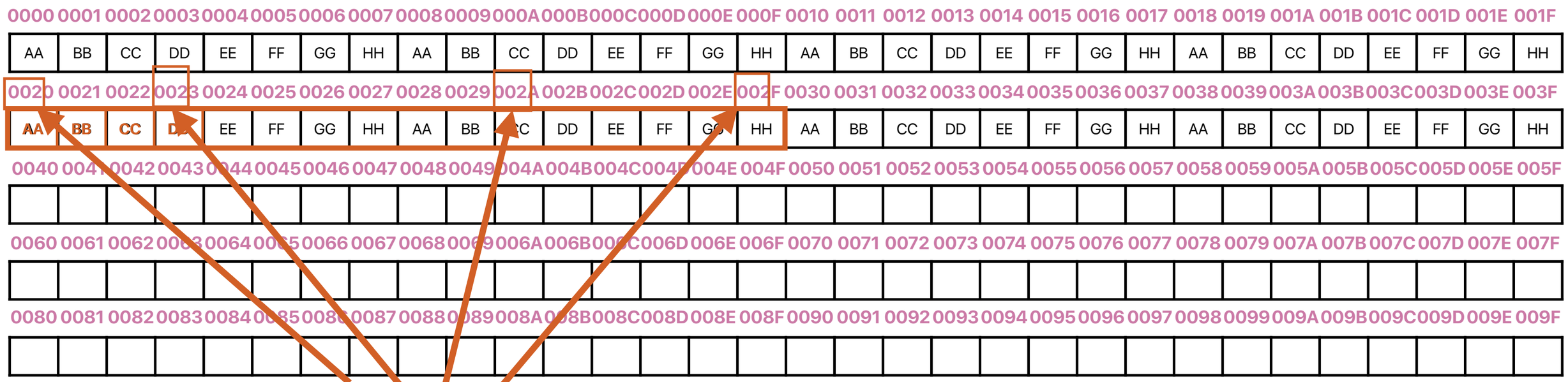
  — the cache must be able to distinguish blocks

How to tell who is there?



Let's cache a "block"!





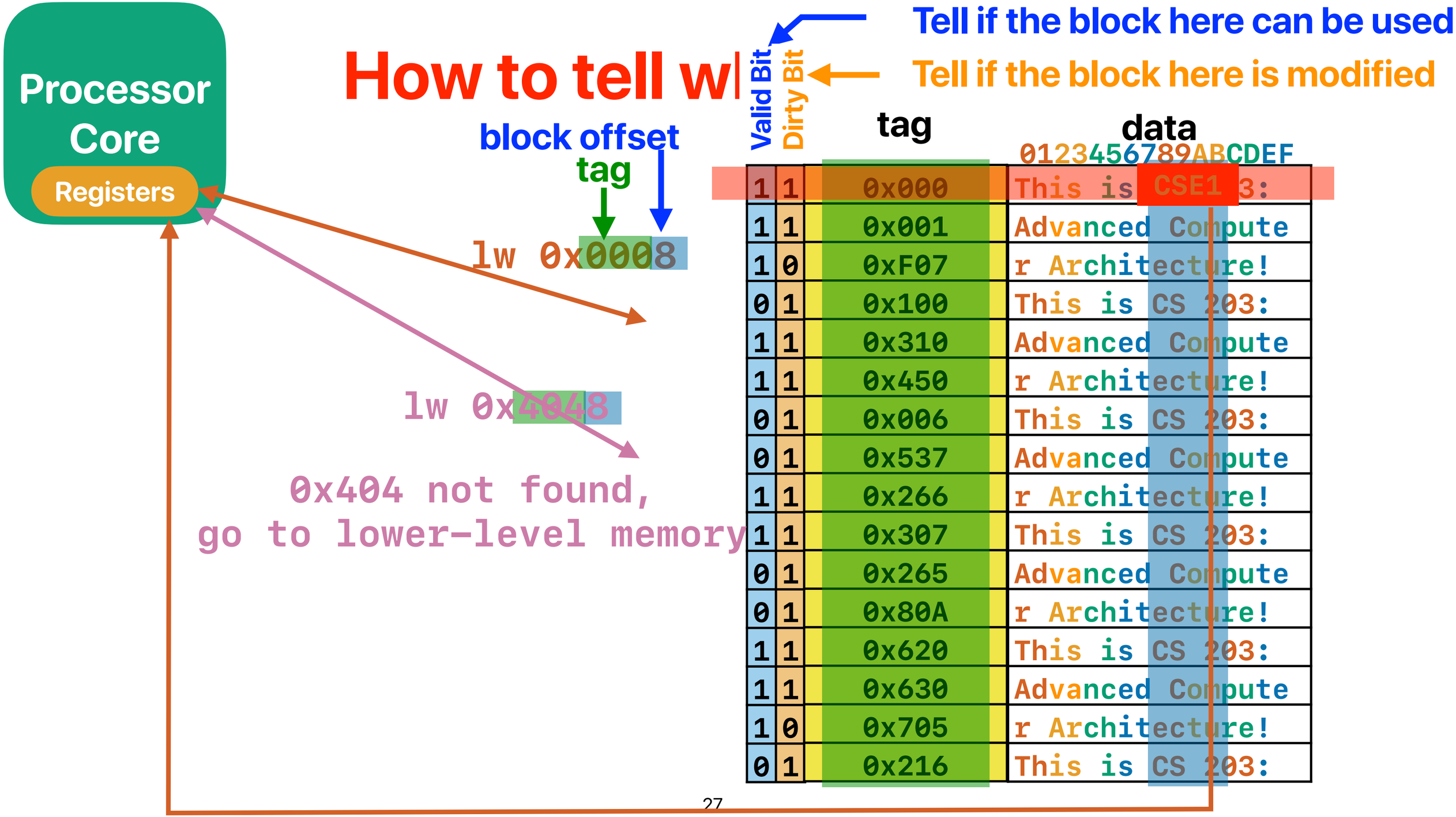
the address in each block starts with the same "prefix"

How to tell who is there?

the common address prefix in each block

tag array	0x0000 0x0001 0x0002 0x0003 0x0004 0x0005 0x0006 0x0007 0x0008 0x0009 0x000A 0x000B 0x000C 0x000D 0x000E 0x000F 0123456789ABCDEF
0x000	This is CS 203:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CS 203:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CS 203:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CS 203:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CS 203:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CS 203:

How to tell w


Processor Core
Registers
Valid Bit
Dirty Bit
Tell if the block here can be used
Tell if the block here is modified
block offset
tag
lw 0x0008
lw 0x4048
0x404 not found,
go to lower-level memory
tag
data
0123456789ABCDEF
1 1 0x000 This is CSE1 3:
1 1 0x001 Advanced Compute
1 0 0xF07 r Architecture!
0 1 0x100 This is CS 203:
1 1 0x310 Advanced Compute
1 1 0x450 r Architecture!
0 1 0x006 This is CS 203:
0 1 0x537 Advanced Compute
1 1 0x266 r Architecture!
1 1 0x307 This is CS 203:
0 1 0x265 Advanced Compute
0 1 0x80A r Architecture!
1 1 0x620 This is CS 203:
1 1 0x630 Advanced Compute
1 0 0x705 r Architecture!
0 1 0x216 This is CS 203:

How to tell w

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

Valid Bit

Dirty Bit

Tell if the block here can be used

Tell if the block here is modified

Valid Bit	Dirty Bit	tag	data (0123456789ABCDEF)
1	1	0x000	This is CSE1 3:
1	1	0x001	Advanced Compute
1	0	0xF07	r Architecture!
0	1	0x100	This is CS 203:
1	1	0x310	Advanced Compute
1	1	0x450	r Architecture!
0	1	0x006	This is CS 203:
0	1	0x537	Advanced Compute
1	1	0x266	r Architecture!
1	1	0x307	This is CS 203:
0	1	0x265	Advanced Compute
0	1	0x80A	r Architecture!
1	1	0x620	This is CS 203:
1	1	0x630	Advanced Compute
1	0	0x705	r Architecture!
0	1	0x216	This is CS 203:

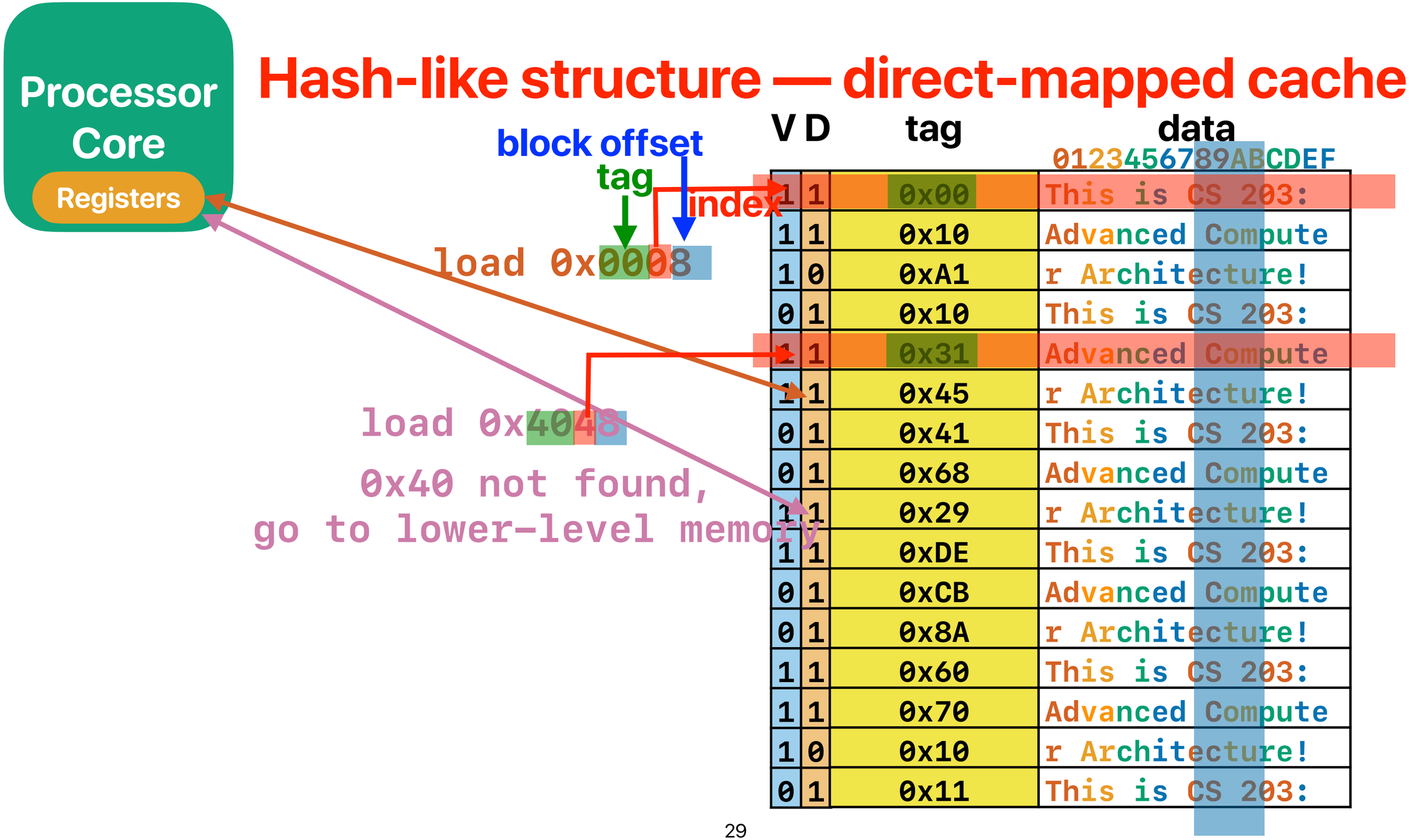
The complexity of search the matching tag—

$O(n)$— will be slow if our cache size grows!

Can we search things faster?

—hash table! $O(1)$

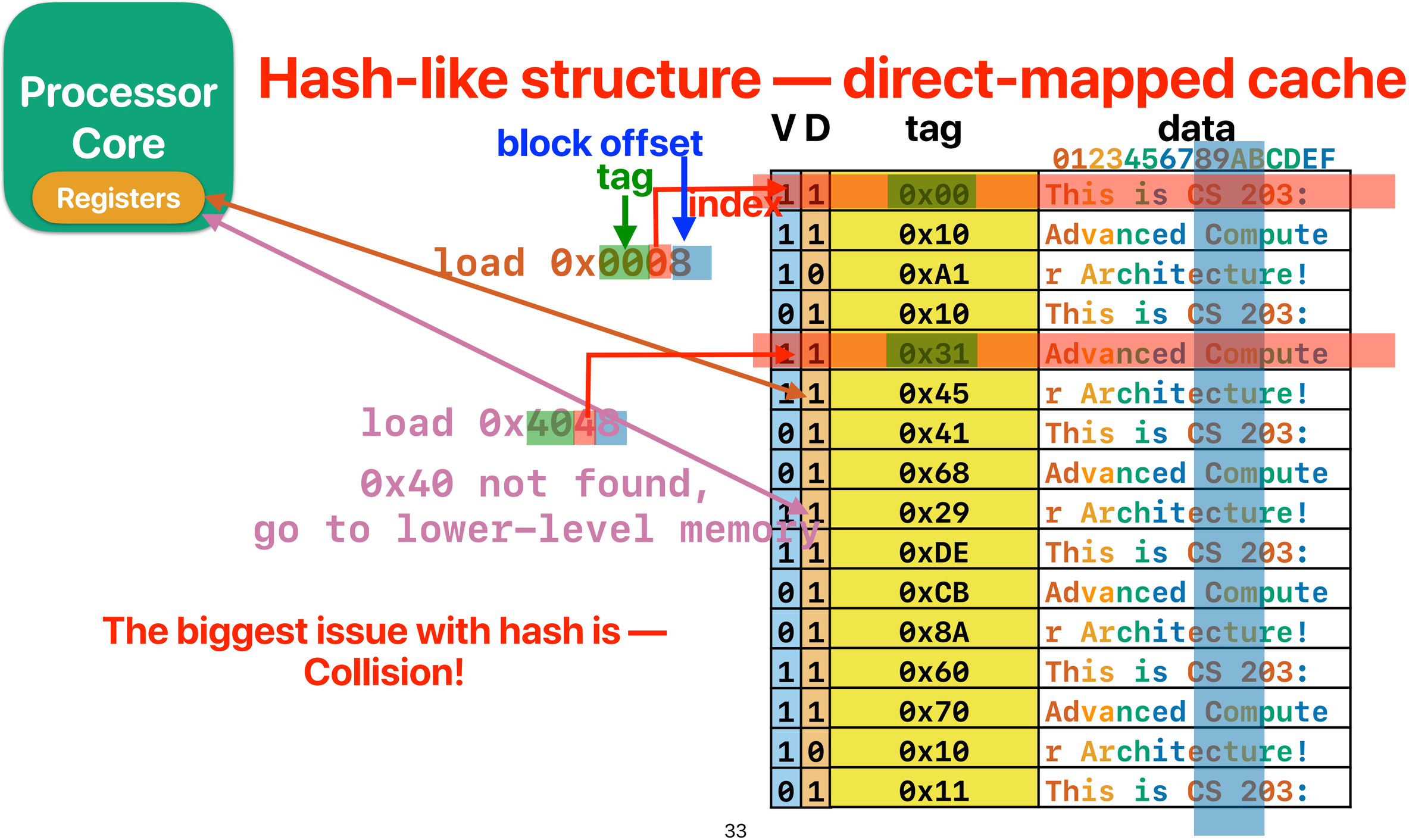
Hash-like structure — direct-mapped cache



Blocksize == Linesize

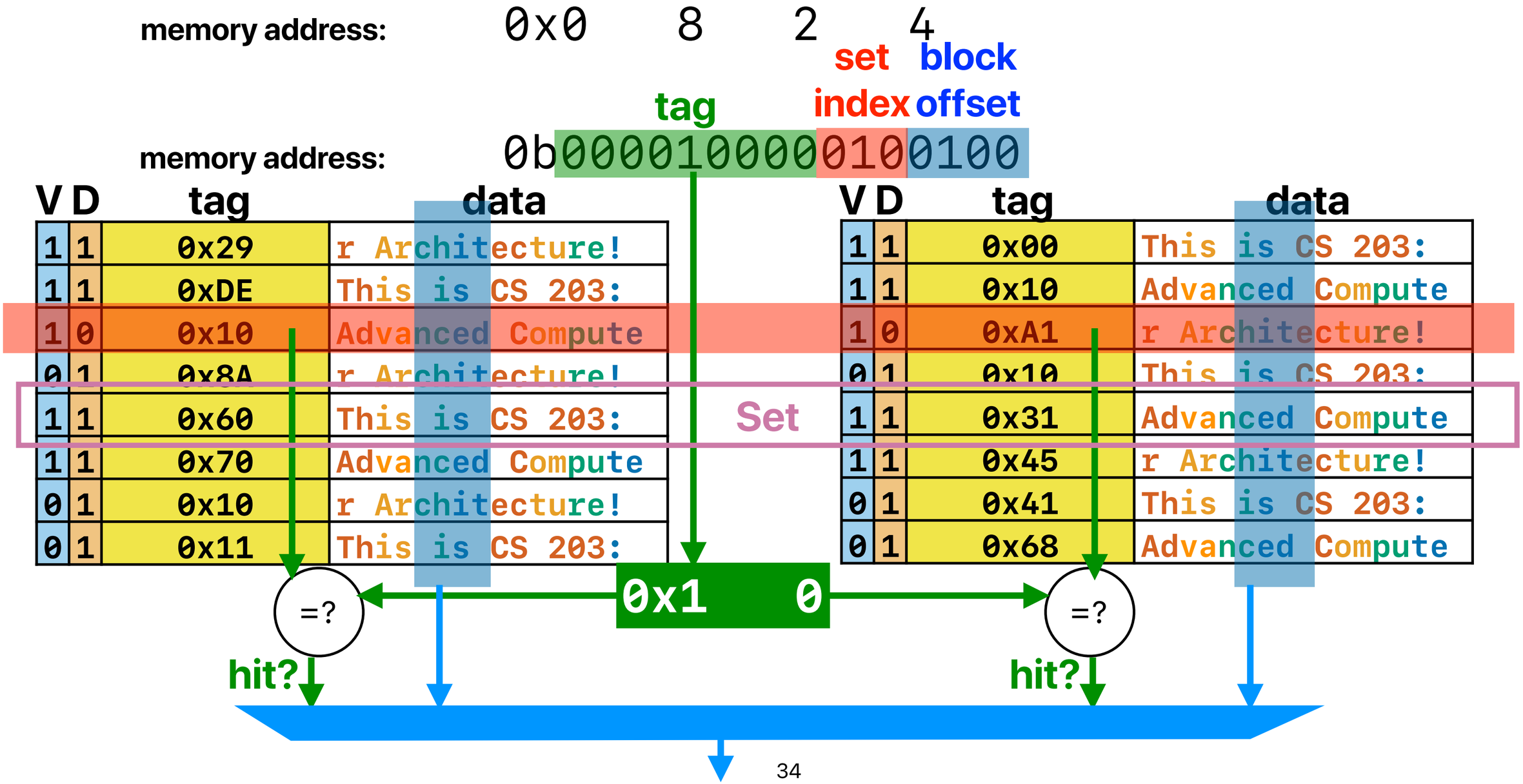
```
[5]: # Your CS203 Cluster
     ! cs203 demo "lscpu | grep 'Model name'; getconf -a | grep CACHE"

     ssh htseng@horsea " srun -N1 -p datahub lscpu | grep 'Model name'"
     Model name:                           12th Gen Intel(R) Core(TM) i3-12100F
     ssh htseng@horsea " srun -N1 -p datahub  getconf -a | grep CACHE"
     LEVEL1_ICACHE_SIZE                    32768
     LEVEL1_ICACHE_ASSOC                   8
     LEVEL1_ICACHE_LINESIZE                64
     LEVEL1_DCACHE_SIZE                    49152
     LEVEL1_DCACHE_ASSOC                   12
     LEVEL1_DCACHE_LINESIZE                64
     LEVEL2_CACHE_SIZE                     1310720
     LEVEL2_CACHE_ASSOC                    10
     LEVEL2_CACHE_LINESIZE                 64
     LEVEL3_CACHE_SIZE                     12582912
     LEVEL3_CACHE_ASSOC                    12
     LEVEL3_CACHE_LINESIZE                 64
     LEVEL4_CACHE_SIZE                     0
     LEVEL4_CACHE_ASSOC                    0
     LEVEL4_CACHE_LINESIZE                 0
```

Hash-like structure — direct-mapped cache



The biggest issue with hash is — Collision!

Way-associative cache



What is Associativity?

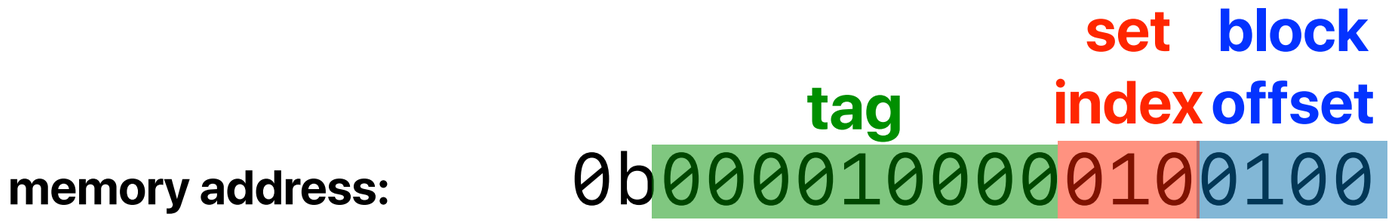
```
[5]: # Your CS203 Cluster
     ! cs203 demo "lscpu | grep 'Model name'; getconf -a | grep CACHE"

     ssh htseng@horsea " srun -N1 -p datahub lscpu | grep 'Model name'"
     Model name:                           12th Gen Intel(R) Core(TM) i3-12100F
     ssh htseng@horsea " srun -N1 -p datahub  getconf -a | grep CACHE"
     LEVEL1_ICACHE_SIZE                    32768
     LEVEL1_ICACHE_ASSOC                   8
     LEVEL1_ICACHE_LINESIZE                64
     LEVEL1_DCACHE_SIZE                    49152
     LEVEL1_DCACHE_ASSOC                   12
     LEVEL1_DCACHE_LINESIZE                64
     LEVEL2_CACHE_SIZE                     1310720
     LEVEL2_CACHE_ASSOC                    10
     LEVEL2_CACHE_LINESIZE                 64
     LEVEL3_CACHE_SIZE                     12582912
     LEVEL3_CACHE_ASSOC                    12
     LEVEL3_CACHE_LINESIZE                 64
     LEVEL4_CACHE_SIZE                     0
     LEVEL4_CACHE_ASSOC                    0
     LEVEL4_CACHE_LINESIZE                 0
```

The A, B, Cs of your cache

C = ABS

- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Linesize)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache

Corollary of C = ABS



- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

https://www.pollev.com/hungweitseng close in 1:30

NVIDIA Tegra X1

- L1 data (D-L1) cache configuration of NVIDIA Tegra X1 (used by Nintendo Switch and Jetson Nano)
  - Size 32KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

NVIDIA Tegra X1

- L1 data (D-L1) cache configuration of NVIDIA Tegra X1 (used by Nintendo Switch and Jetson Nano)
  - Size 32KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

$$C = A \times B \times S$$
$$32 \times 1024 = 4 \times 64 \times S$$
$$S = 128$$
$$Offset = log_2(64) = 6$$
$$Index = log_2(128) = 7$$
$$Tag = 64 - 7 - 6 = 51$$

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?

    A. Tag is 52 bits

    B. Index is 6 bits

    C. Offset is 6 bits

    D. The cache has 128 sets

    E. All of the above are correct

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?

A. Tag is 52 bits
B. Index is 6 bits
C. Offset is 6 bits
D. The cache has 128 sets
E. All of the above are correct

$$C = A \times B \times S$$

$$48 \times 1024 = 12 \times 64 \times S$$

$$S = 64$$

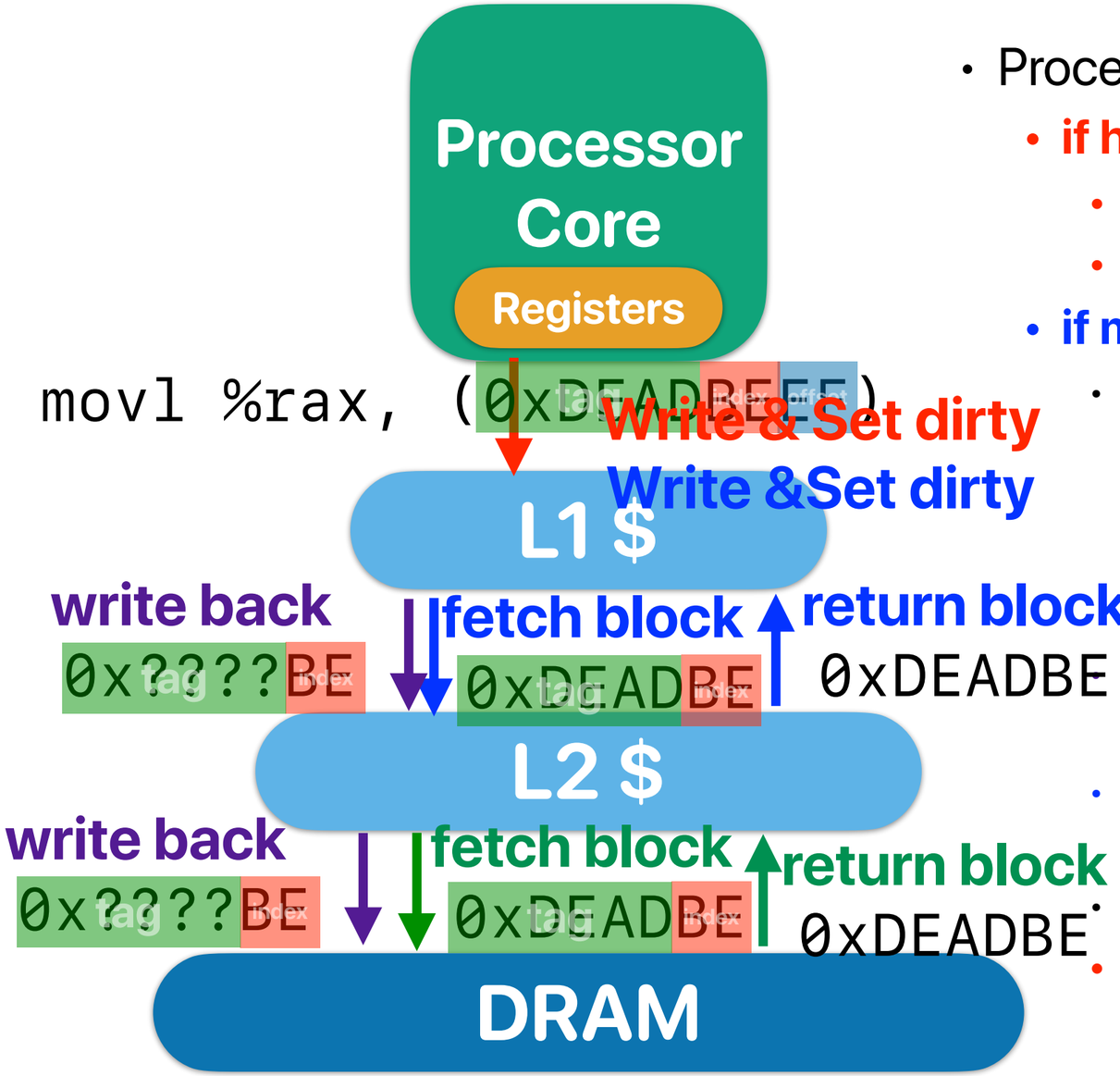
$$Offset = log_2(64) = 6$$

$$Index = log_2(64) = 6$$

$$Tag = 64 - 6 - 6 = 52$$

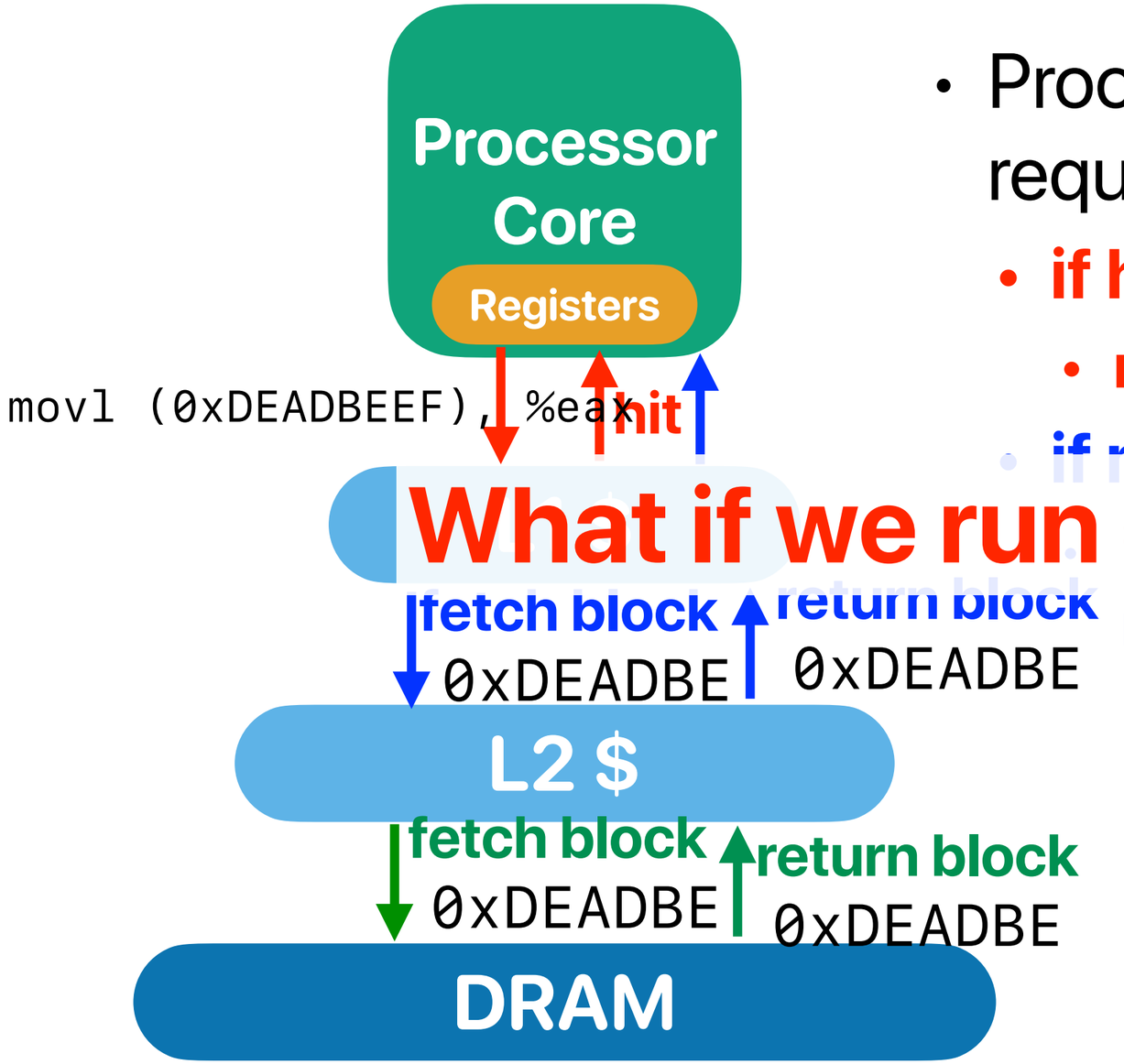
Put everything all together: How cache interacts with CPU

The complete picture



- Processor sends memory access request to L1-$
  - **if hit**
    - **Read - return data**
    - **Write - update & set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
  - **Present the write "ONLY" in L1 and set DIRTY**

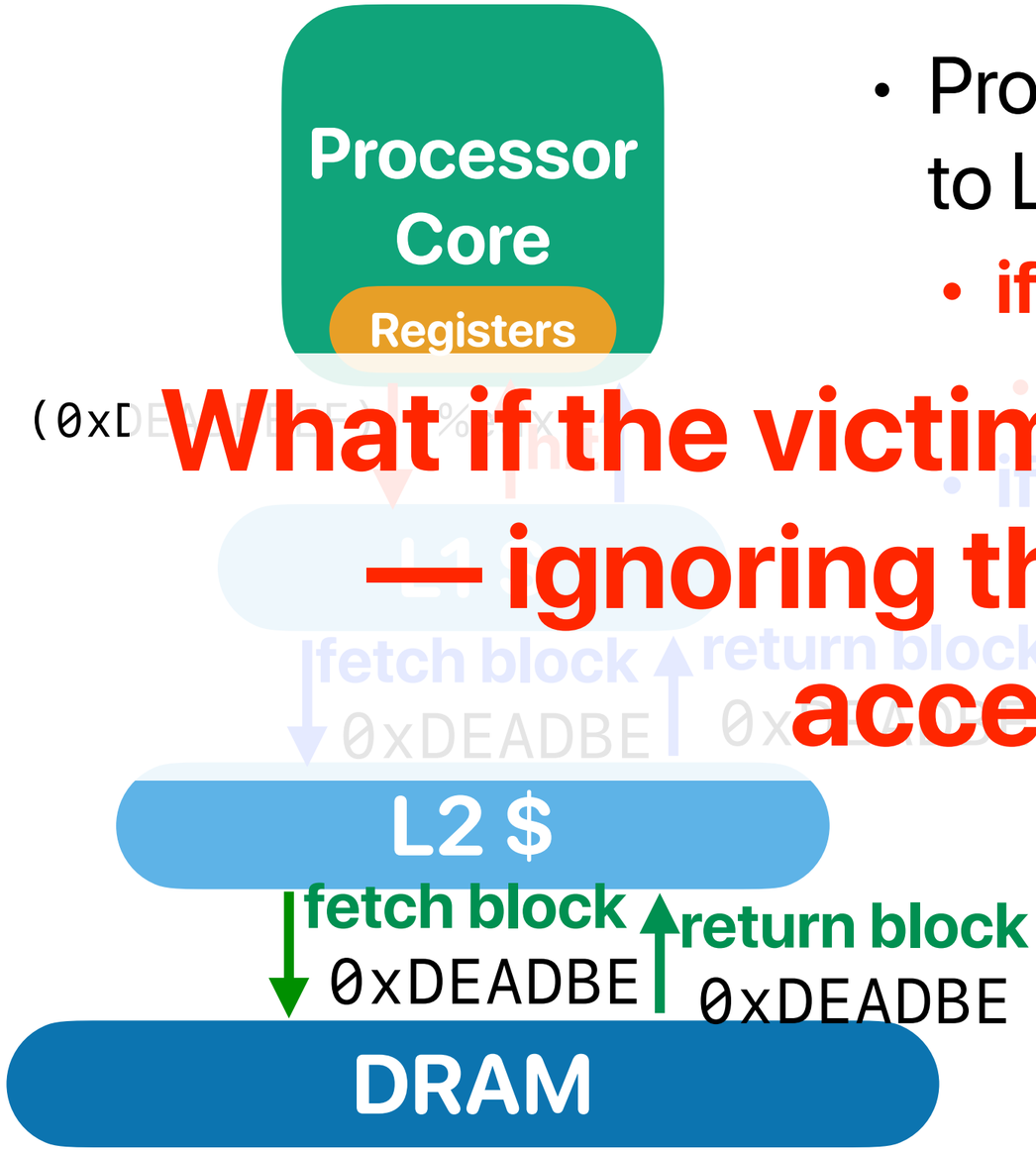
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - **fetch the requesting block from lower-level memory hierarchy and place in the cache**

What if we run out of $ blocks?

Considering we have limited space in $

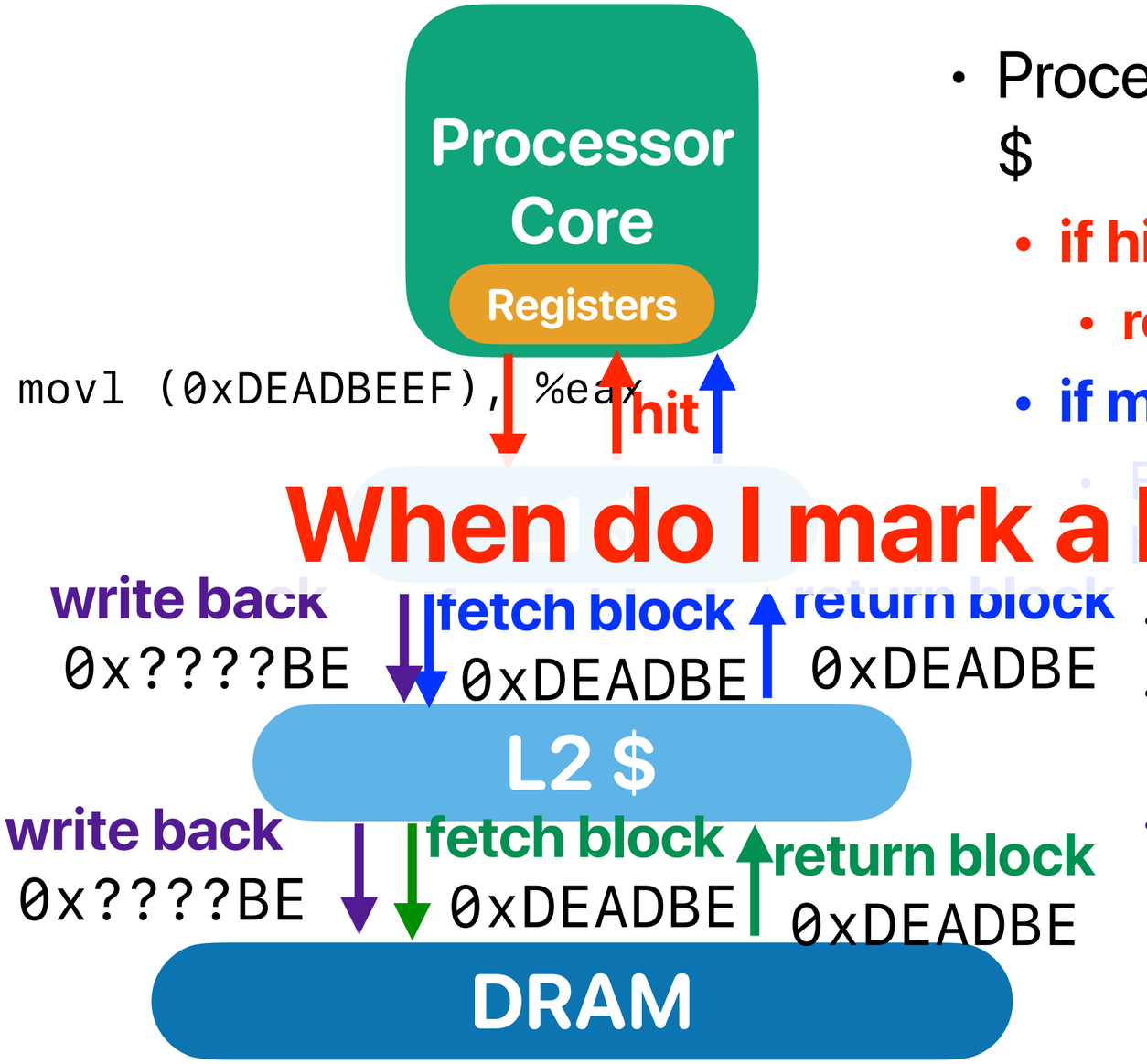


- Processor sends memory access request to L1-$
  - **if hit**
    - return data
  - **if miss**
    - fetch block from lower-level memory hierarchy and place in the cache and then return data

What if the victim block is modified? — ignoring the update is not acceptable!

- If there an empty block — place the data there
- If NOT (most frequent case) — select a **victim block**
  - Least Recently Used (LRU) policy

Considering a victim block may be modified

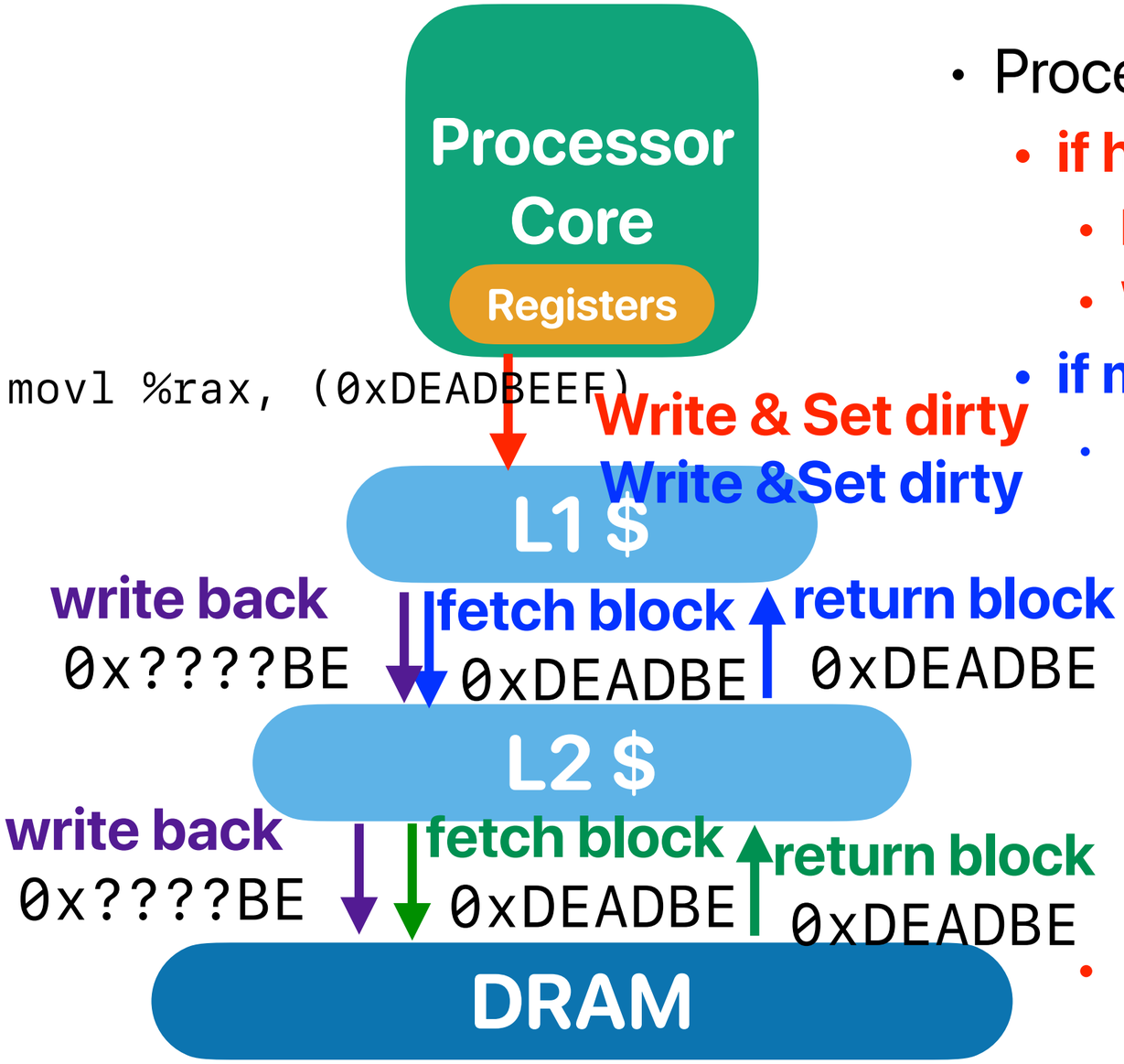


- Processor sends memory access request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - Fetch the requesting block from lower-level memory
    - Replacement: Select a victim block
    - return data

When do I mark a block as modified?

- If there an empty block — place the data there
- If NOT (most frequent case) — select a **victim block**
  - Least Recently Used (LRU) policy
- If the victim block is "dirty" & "valid"
  - **Write back** the block to lower-level memory hierarchy

Considering we have "writes"



- Processor sends memory access request to L1-$
  - **if hit**
    - **Read: Return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache and then return data
      - If there an empty block — place the data there
      - If NOT (most frequent case) — select a **victim block**
        - Least Recently Used (LRU) policy
      - If the victim block is "dirty" & "valid"
        - **Write back** the block to lower-level memory hierarchy
    - **Write: Update "ONLY" in L1 and set DIRTY**

Announcement

- Assignment #2 due **tonight**
  - Review the "demo"s of previous lectures if you need inspiration for programming assignment
  - You should run the experiments yourself and calculate results based on that
    - Everyone should have a different answer
    - The autograder won't credit you if your answer does not contain working progress, equations in LaTeX format (if appropriate) and numbers from your experiments
    - We won't give you credits if you don't show your equations but only answers in the midterm either
  - If you consult your classmates, you need to put their names in the cell where you state your name
- Reading quiz #3 due **next Tuesday** before the lecture
- Assignment #3 will be ready this weekend
  - Please check our course website https://www.escalab.org/classes/cs203-2024sp/

Computer
Science &
Engineering

203

つづく