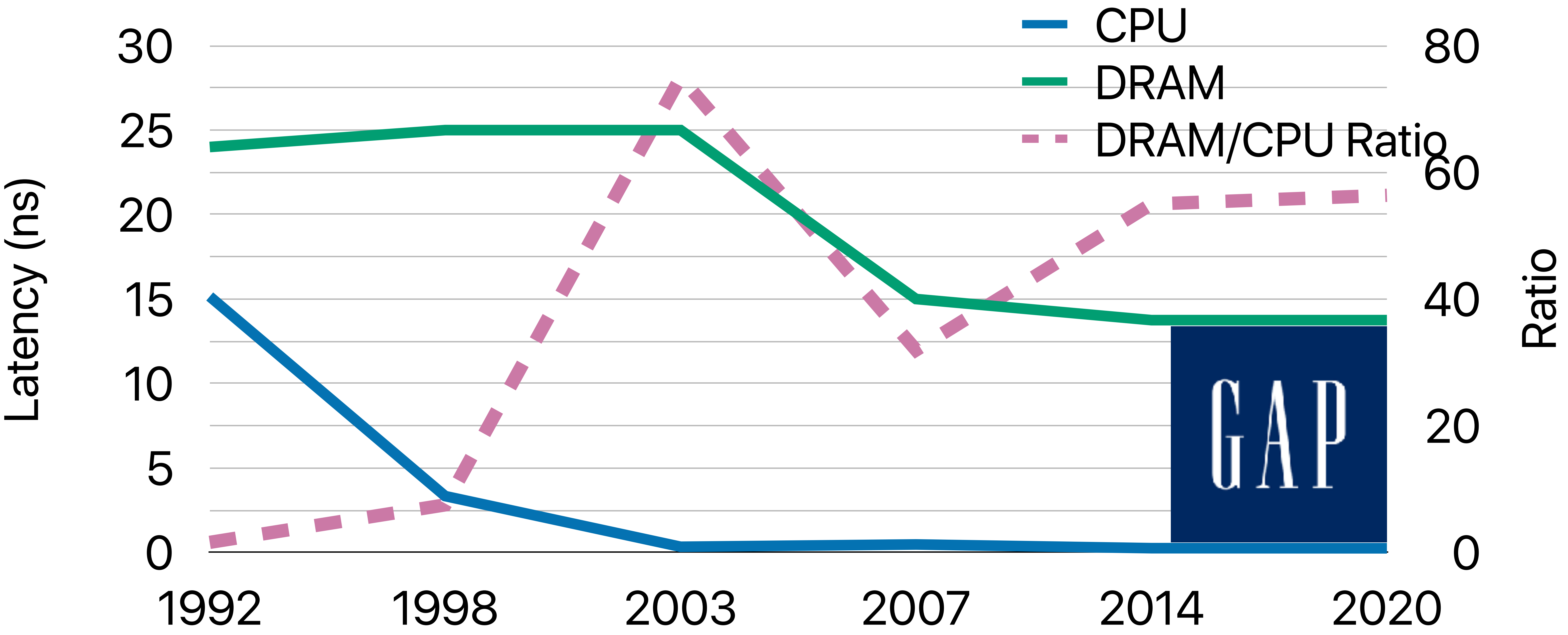
Memory Hierarchy (2): The ABCs of your caches

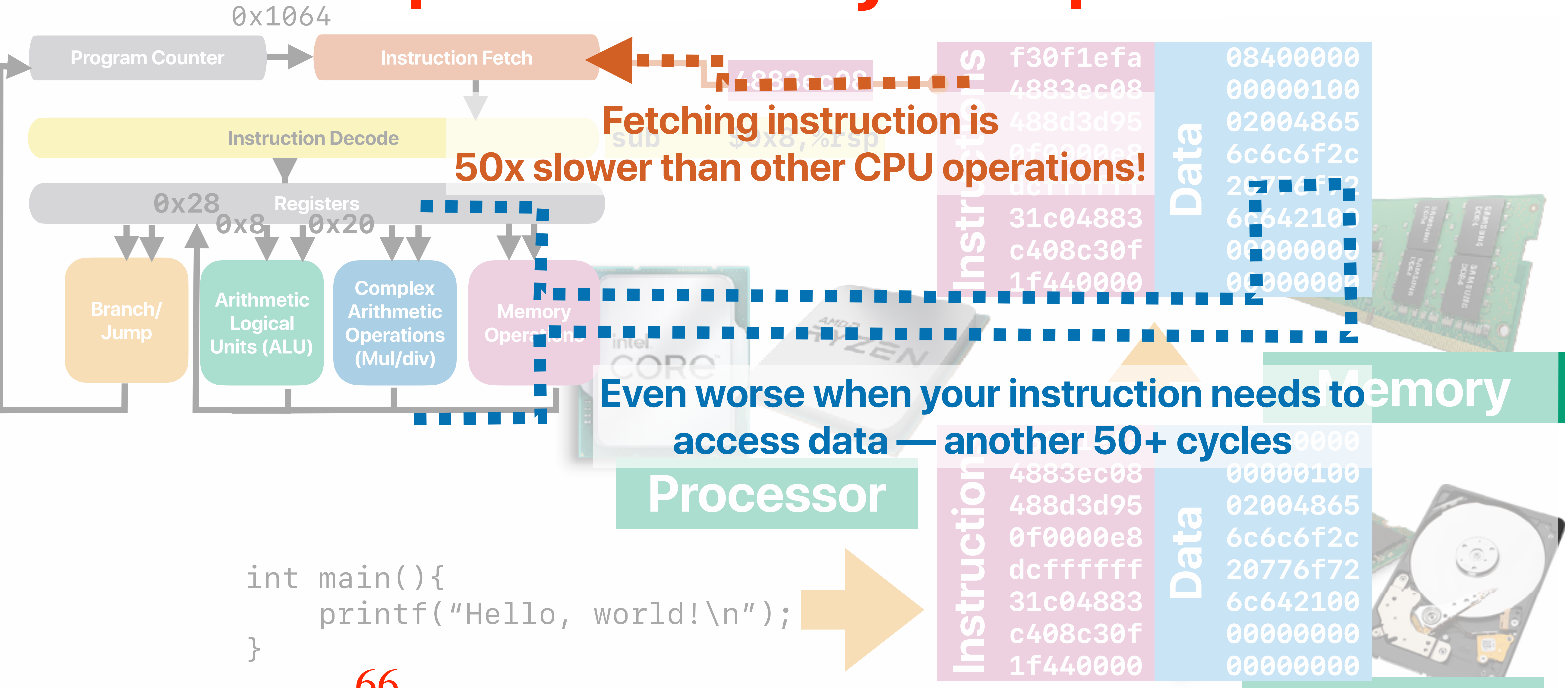
Hung-Wei Tseng

Recap: The “latency” gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

Recap: The memory-wall problem



Fetching instruction is 50x slower than other CPU operations!

Even worse when your instruction needs to access data — another 50+ cycles

```
int main(){
    printf("Hello, world!\n");
}
```

$$\frac{66}{67} = 98.5\ \%$$ of time, we're dealing with memory accesses!

Recap: Data locality

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

```
for(uint64_t i = 0; i < m; i++) {
  result = 0;
  for(uint64_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

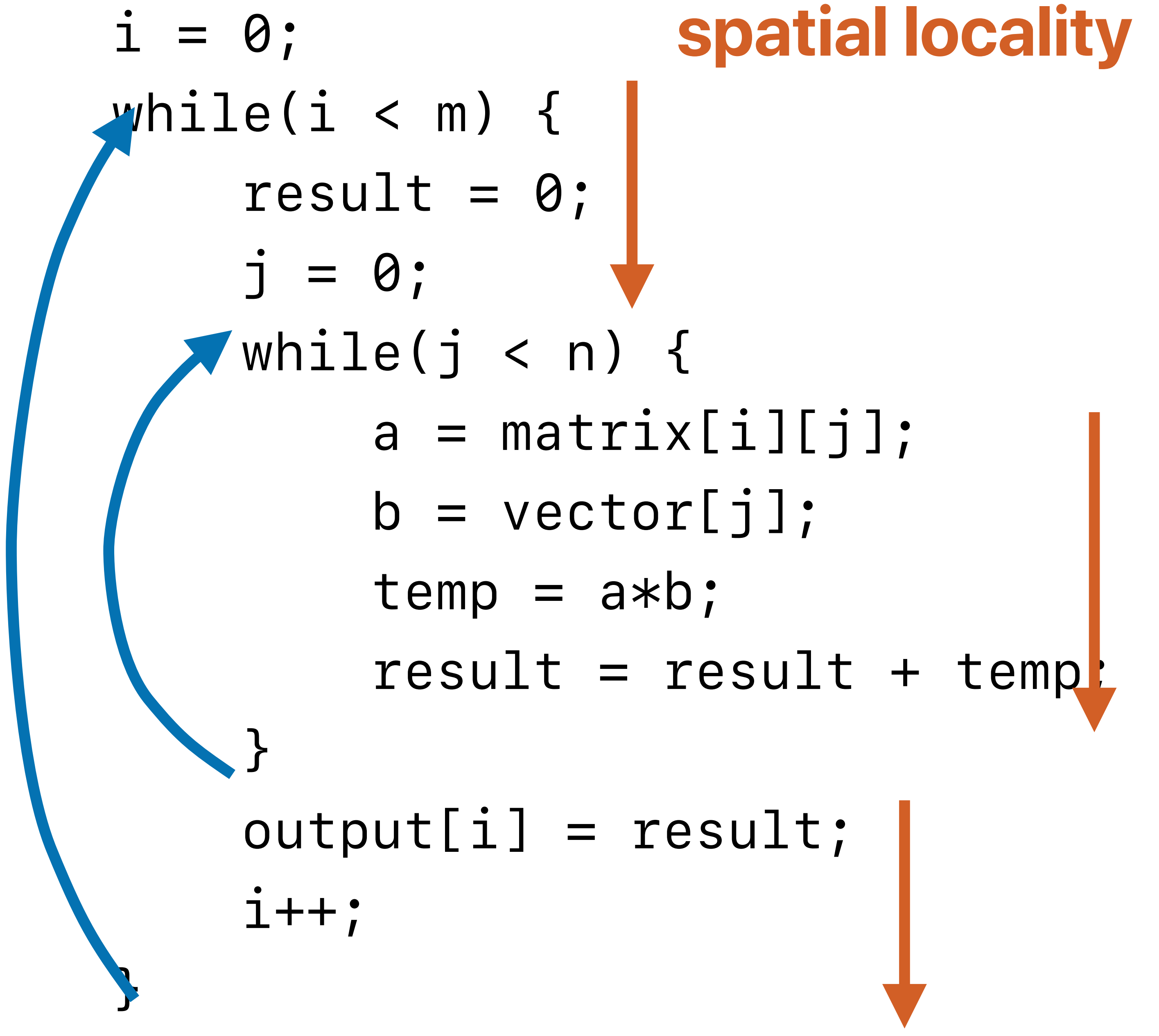
C. Access of `matrix` has spatial locality, `vector` has temporal locality

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]

Recap: Code also has locality

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_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!

Recap: Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory
    - Code — the current instruction, and then the next instruction
    - Data — the next element in an array, then the next

We need to "cache consecutive memory locations" every time

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

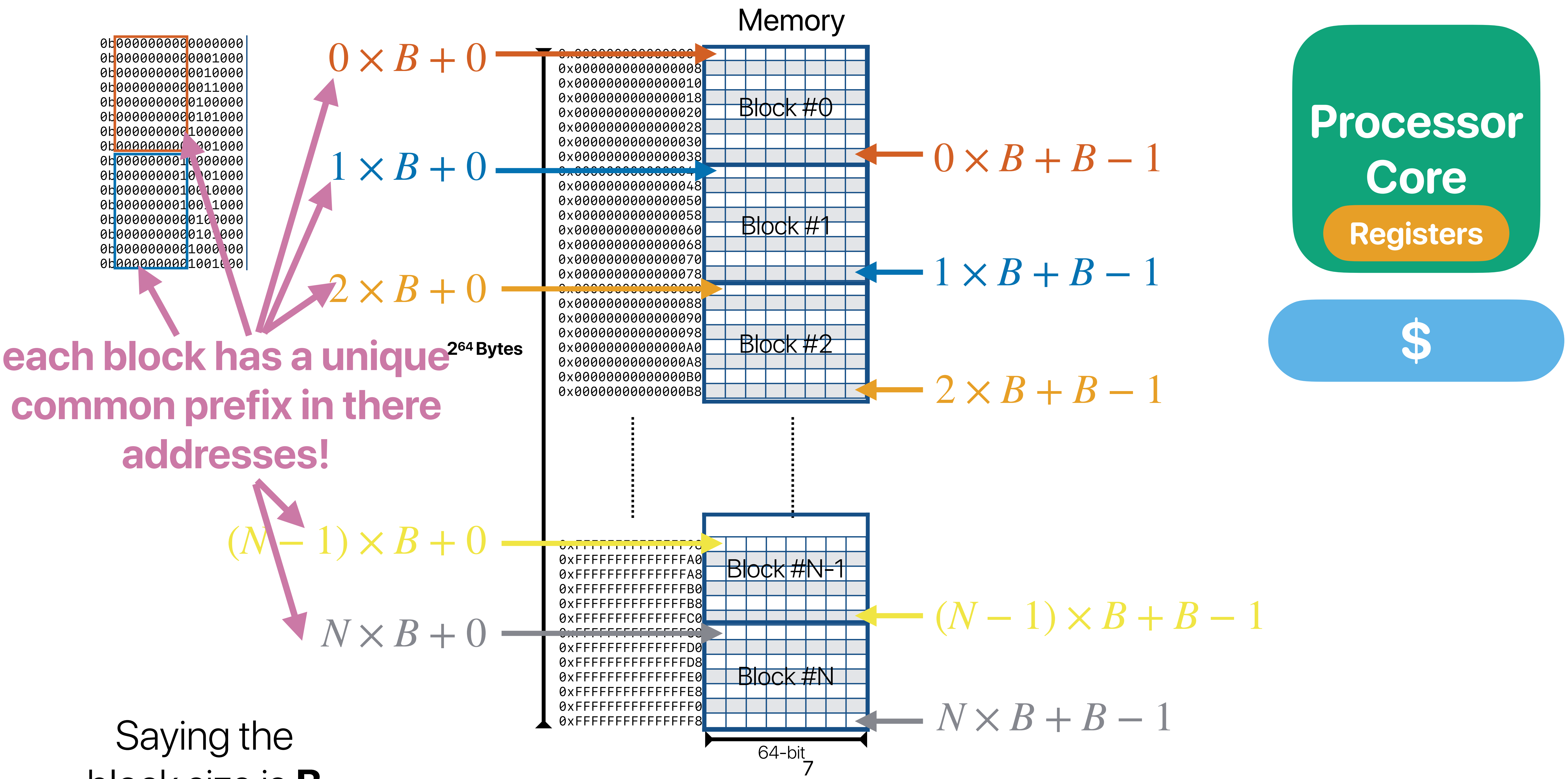
- Temporal locality — application revisit the same thing again and again
    - Code — loops, frequently invoked functions
        - Typically tens of static instructions — at most several KBs
    - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

We need to "cache frequently used memory blocks"

— the cache should store a few blocks

— the cache must be able to distinguish blocks

Recap: Partition memory addresses into fix-sized chunks



```
0b0000000000000000
0b0000000000001000
0b0000000000010000
0b0000000000011000
0b0000000000100000
0b0000000000101000
0b0000000001000000
0b0000000001001000
0b0000000010000000
0b0000000010001000
0b0000000010010000
0b0000000010011000
0b0000000000100000
0b0000000000101000
0b0000000001000000
0b0000000001001000
```

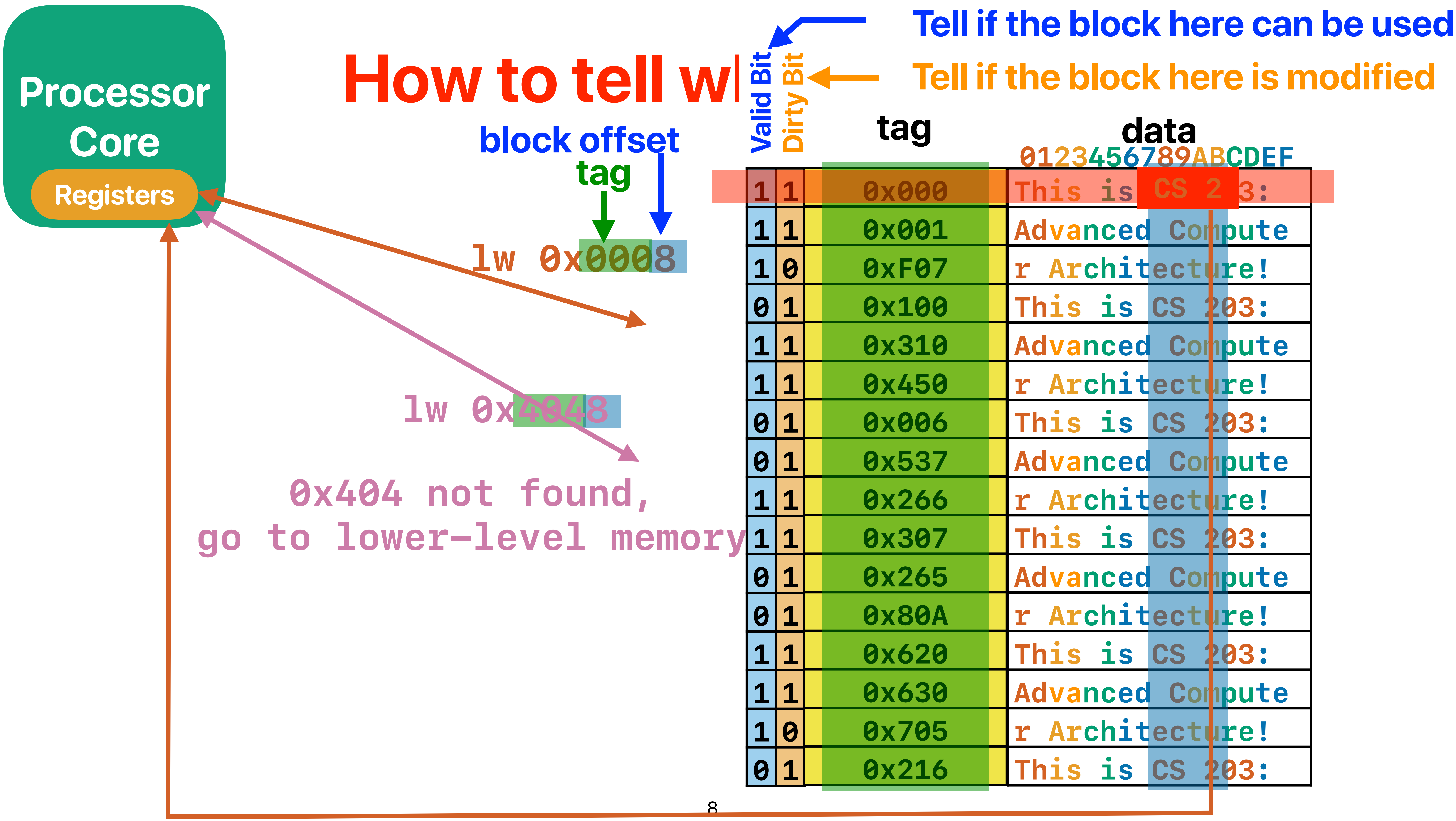
```
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040
0x0000000000000048
0x0000000000000050
0x0000000000000058
0x0000000000000060
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8
```

each block has a unique common prefix in there addresses!

Saying the block size is **B**

How to tell w



Tell if the block here can be used

Tell if the block here is modified

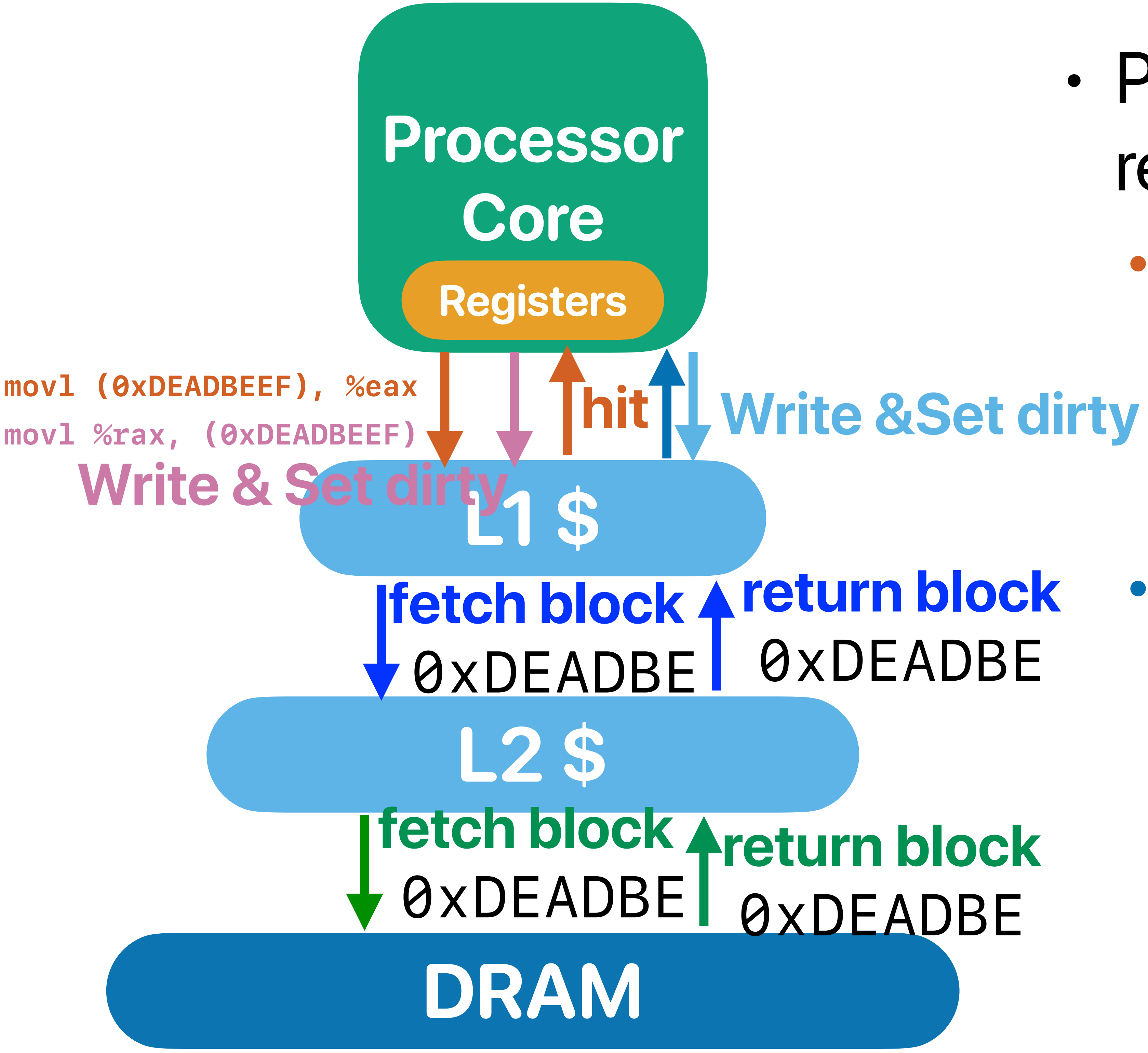
Valid Bit	Dirty Bit	tag	data 0123456789ABCDEF
1	1	0x000	This is CS 203:
1	1	0x001	Advanced Compute
1	0	0xF07	r Architecture!
0	1	0x100	This is CS 203:
1	1	0x310	Advanced Compute
1	1	0x450	r Architecture!
0	1	0x006	This is CS 203:
0	1	0x537	Advanced Compute
1	1	0x266	r Architecture!
1	1	0x307	This is 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:

Outline

- Make cache more efficient
- The geometry of cache architecture: A, B, C, and S.
- How well does my code work on the cache?
- Why does cache miss?

Put everything all together:
How cache interacts with CPU

Processor/cache interaction



- Processor sends memory access request to L1-$
  - if hit & it's a read
    - Read: return data
    - Write: Update "ONLY" in L1 and set DIRTY

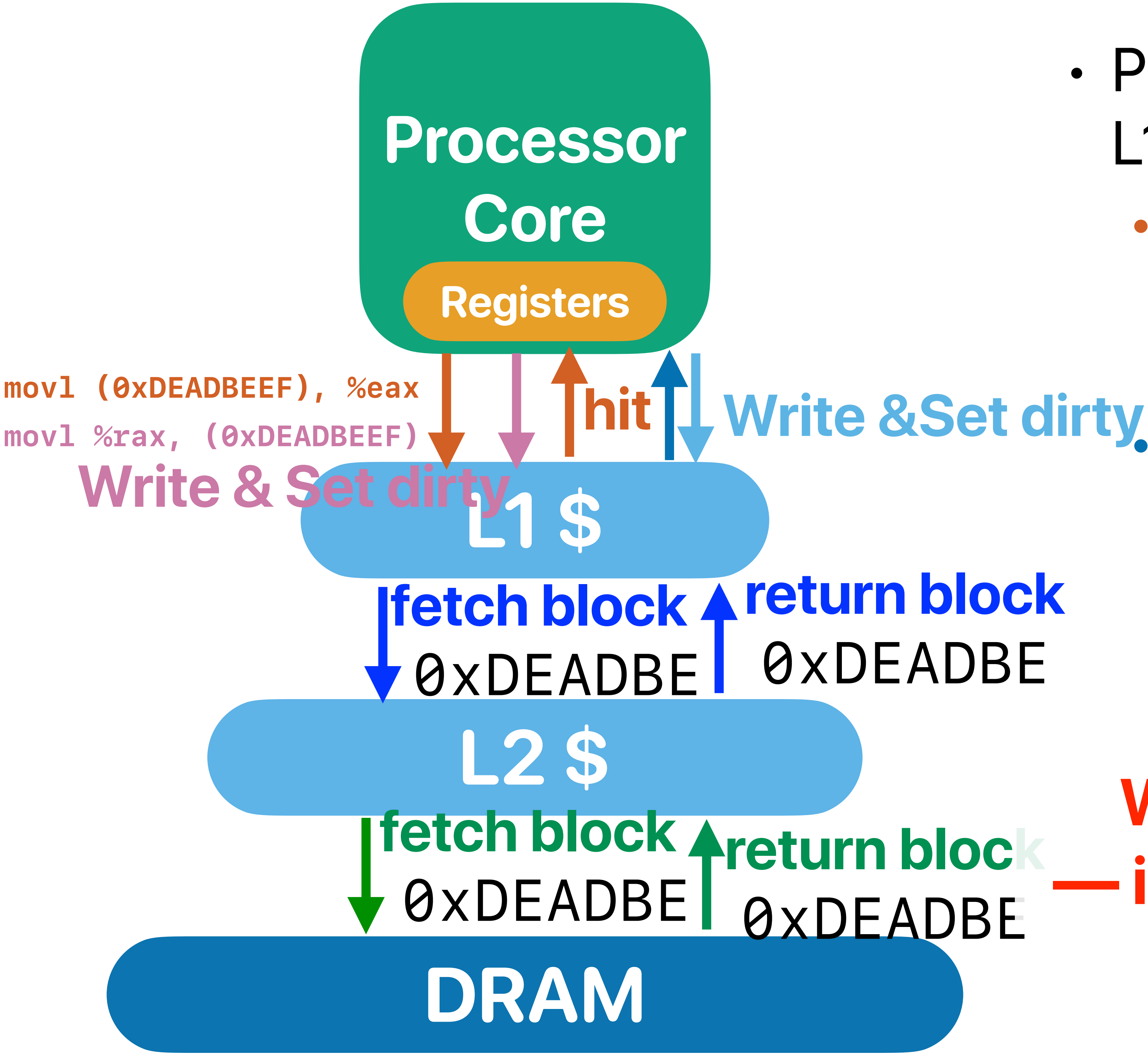
Why don't we write to L2?

— Too slow

  - if miss
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - Present the write "ONLY" in L1 and set DIRTY

What if we run out of $ blocks?

Processor/cache interaction



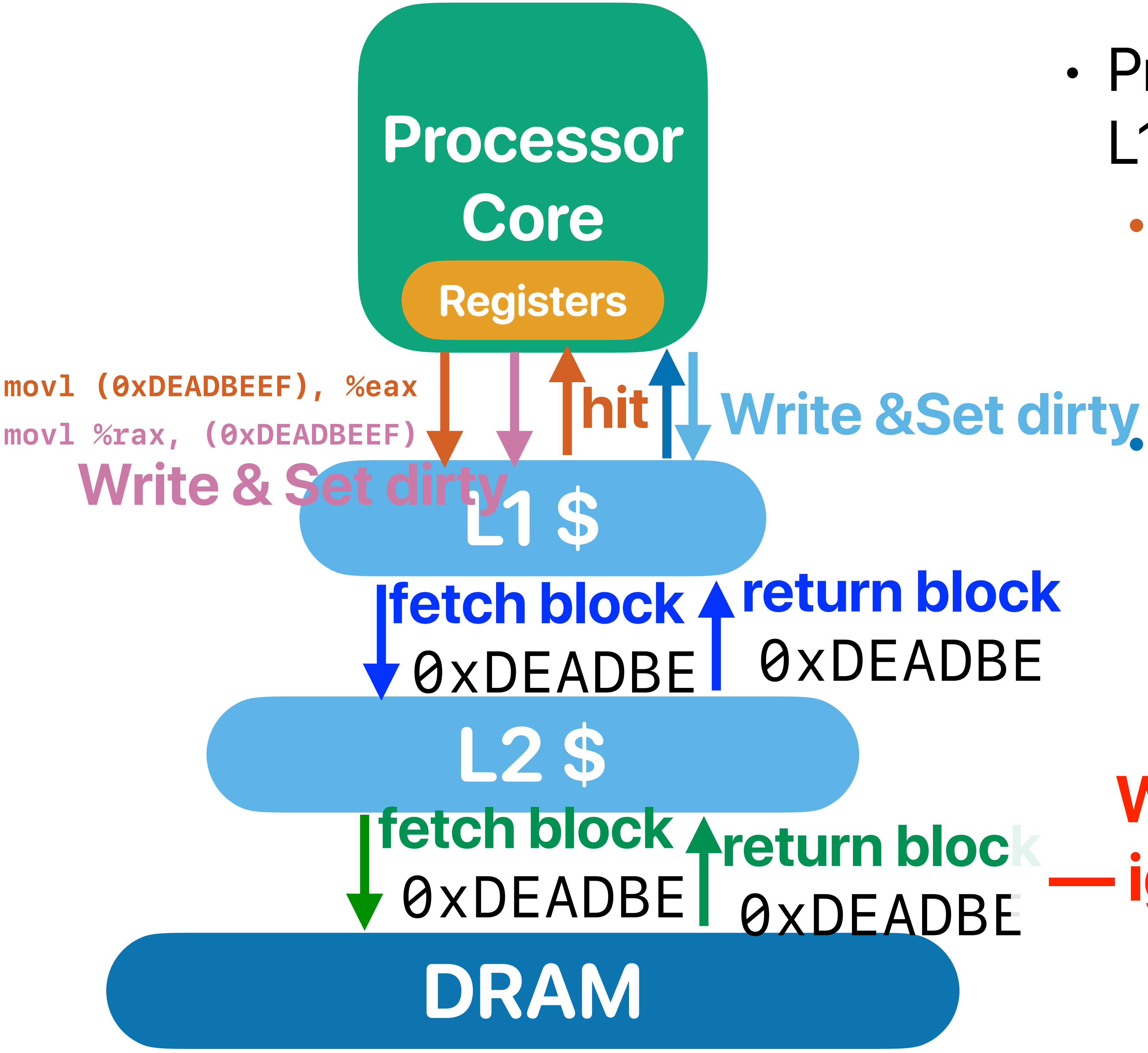
```
movl (0xDEADBEEF), %eax
movl %rax, (0xDEADBEEF)
```

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

What if the victim block is modified?

— ignoring the update is not acceptable!

Processor/cache interaction



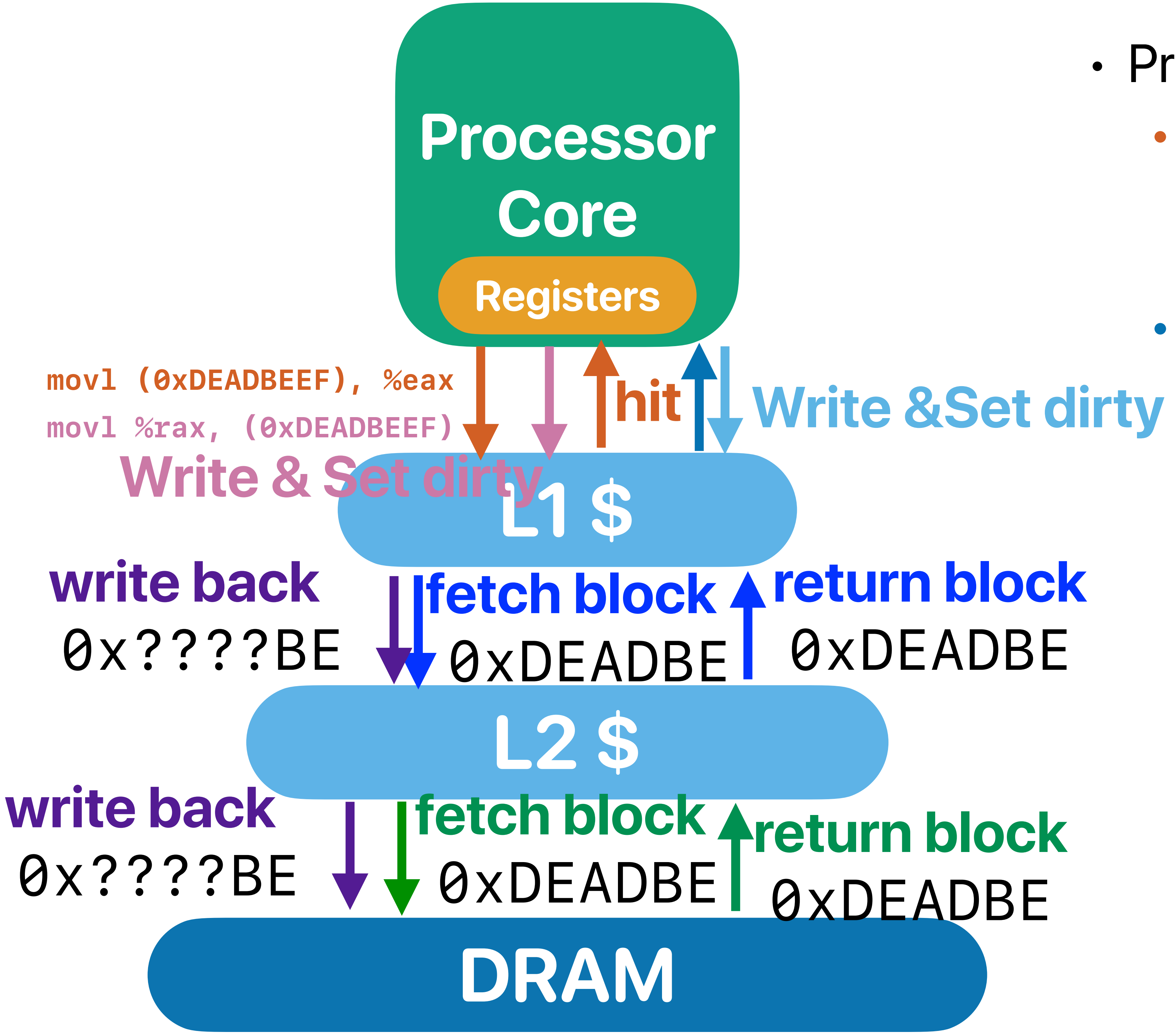
```
movl (0xDEADBEEF), %eax
movl %rax, (0xDEADBEEF)
```

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

What if the victim block is modified?

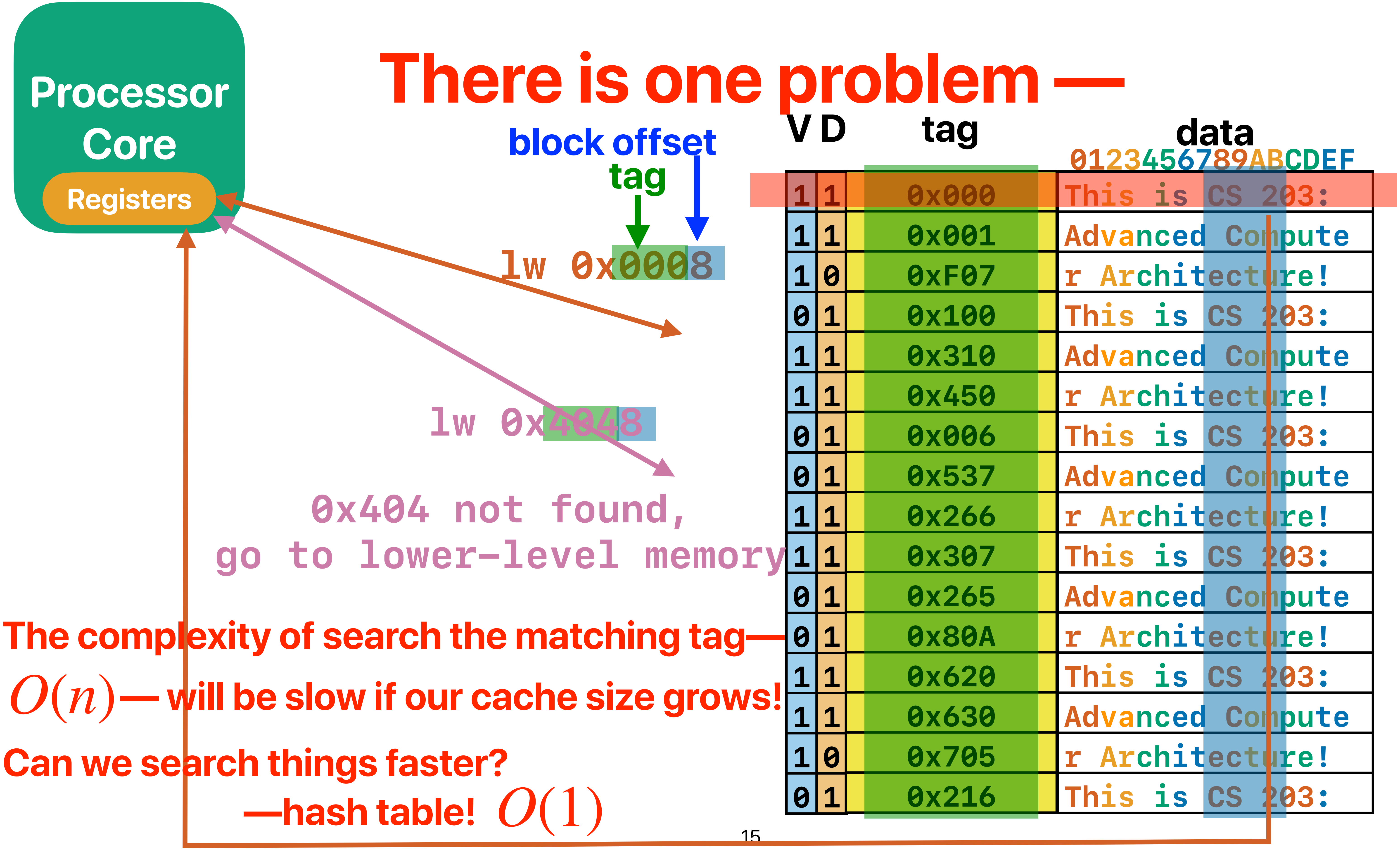
— ignoring the update is not acceptable!

Processor/cache interaction



- Processor sends memory access request to L1-$
  - if hit & it's a read
    - Read: return data
    - Write: Update "ONLY" in L1 and set DIRTY
  - if miss
    - 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
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

There is one problem —



V	D	tag	data 0123456789ABCDEF
1	1	0x000	This is CS 203:
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:

0x404 not found,
go to lower-level memory

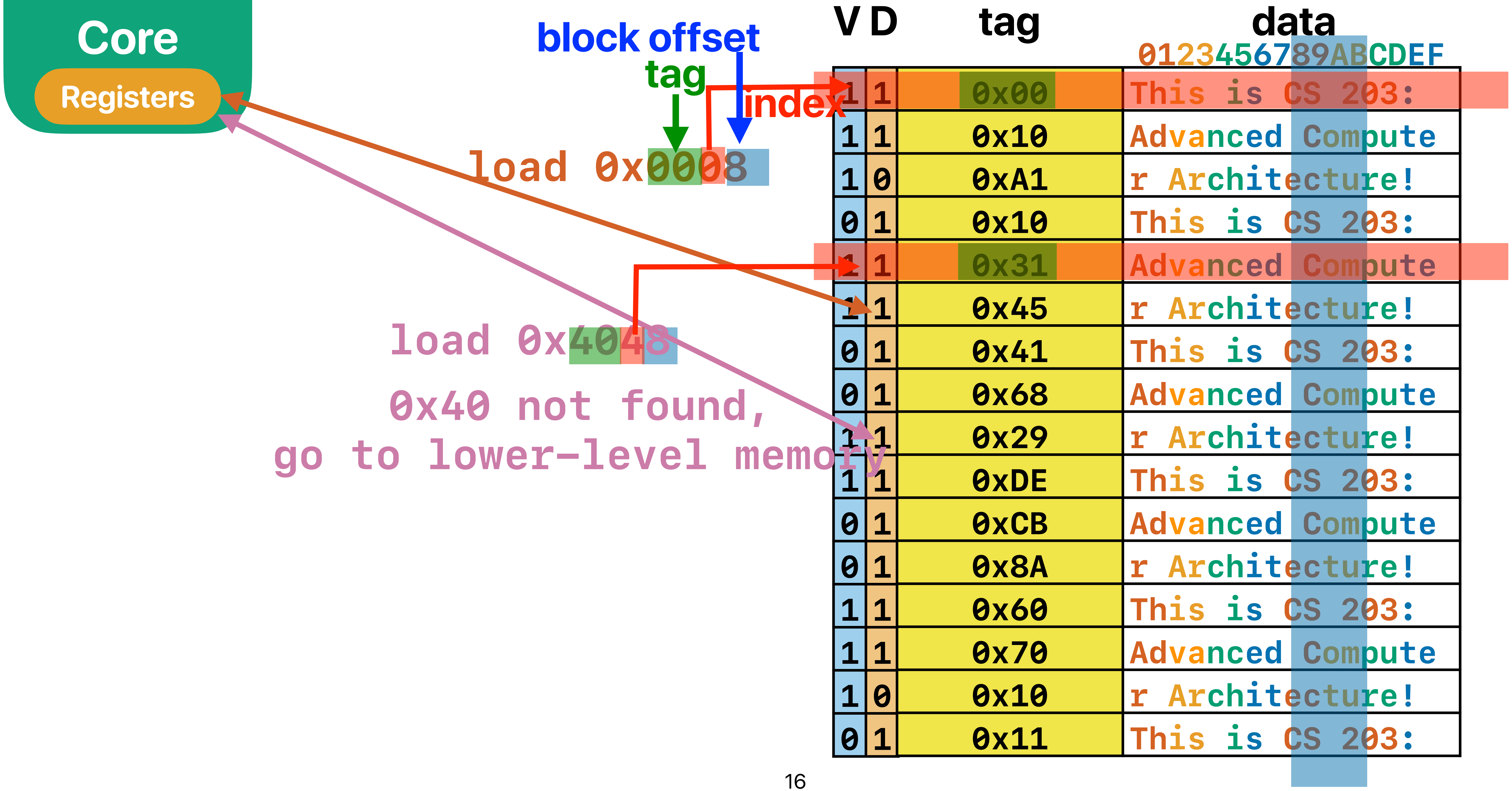
The complexity of search the matching tag—

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

Can we search things faster?

—hash table! $O(1)$

Hash-like structure — direct-mapped cache



load 0x0008

load 0x4048

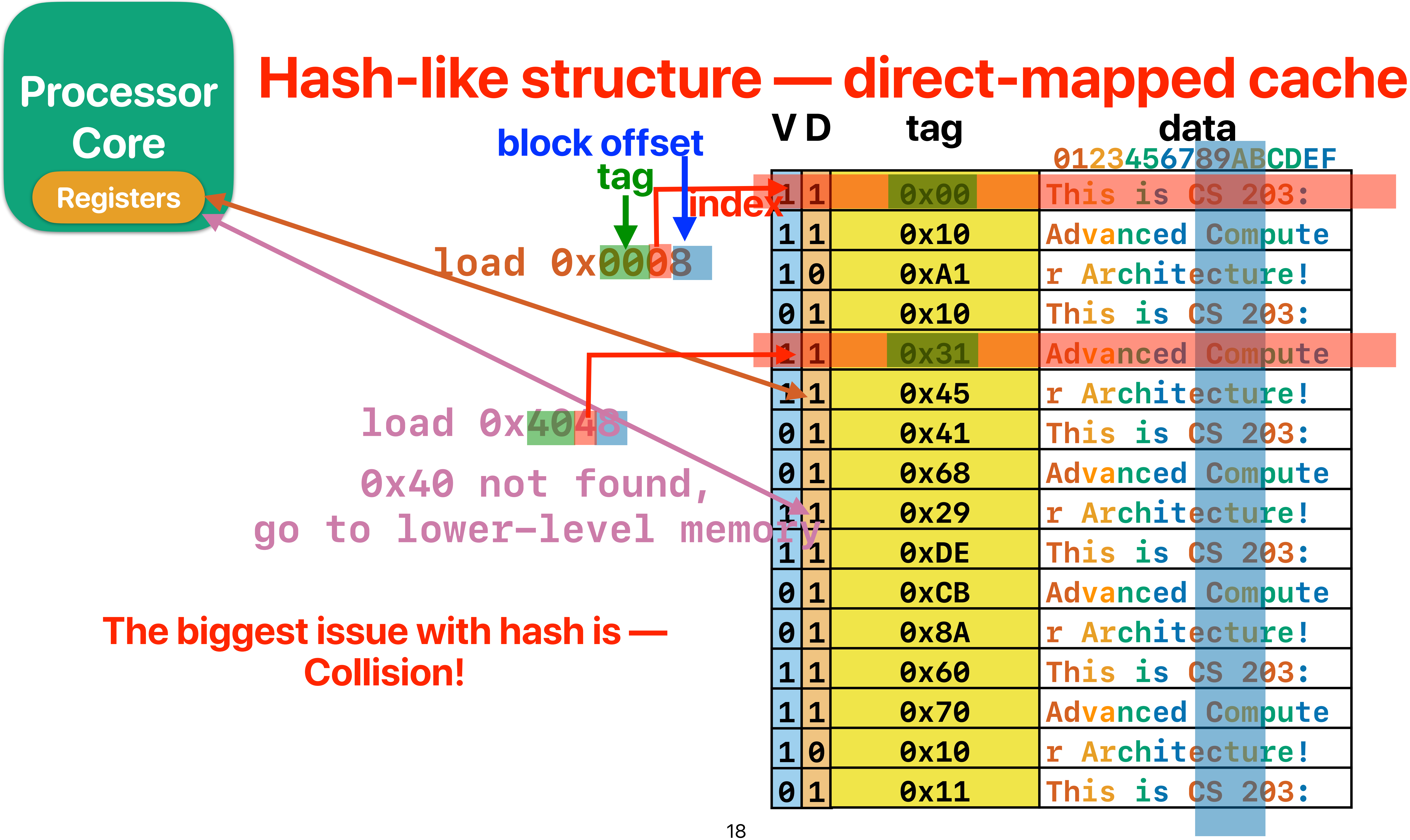
0x40 not found,
go to lower-level memory

V	D	tag	data 0123456789ABCDEF
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

Take-aways: designing caches

- Optimizing cache structures
  - Hash block into “sets” to reduce the search time

Hash-like structure — direct-mapped cache



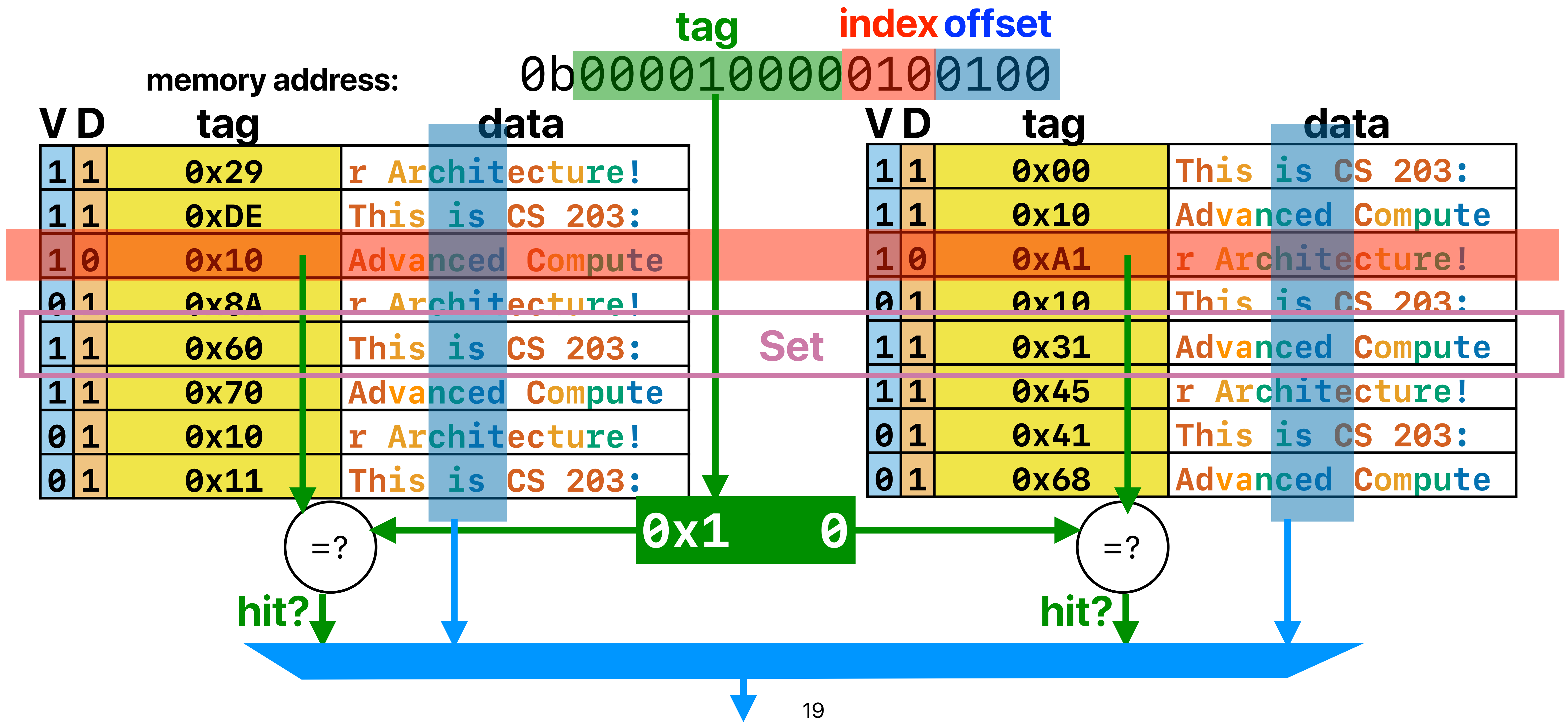
0x40 not found, go to lower-level memory

V	D	tag	data 0123456789ABCDEF
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

The biggest issue with hash is — Collision!

Way-associative cache

memory address: 0x0 8 2 4



memory address: 0b0000100000100100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

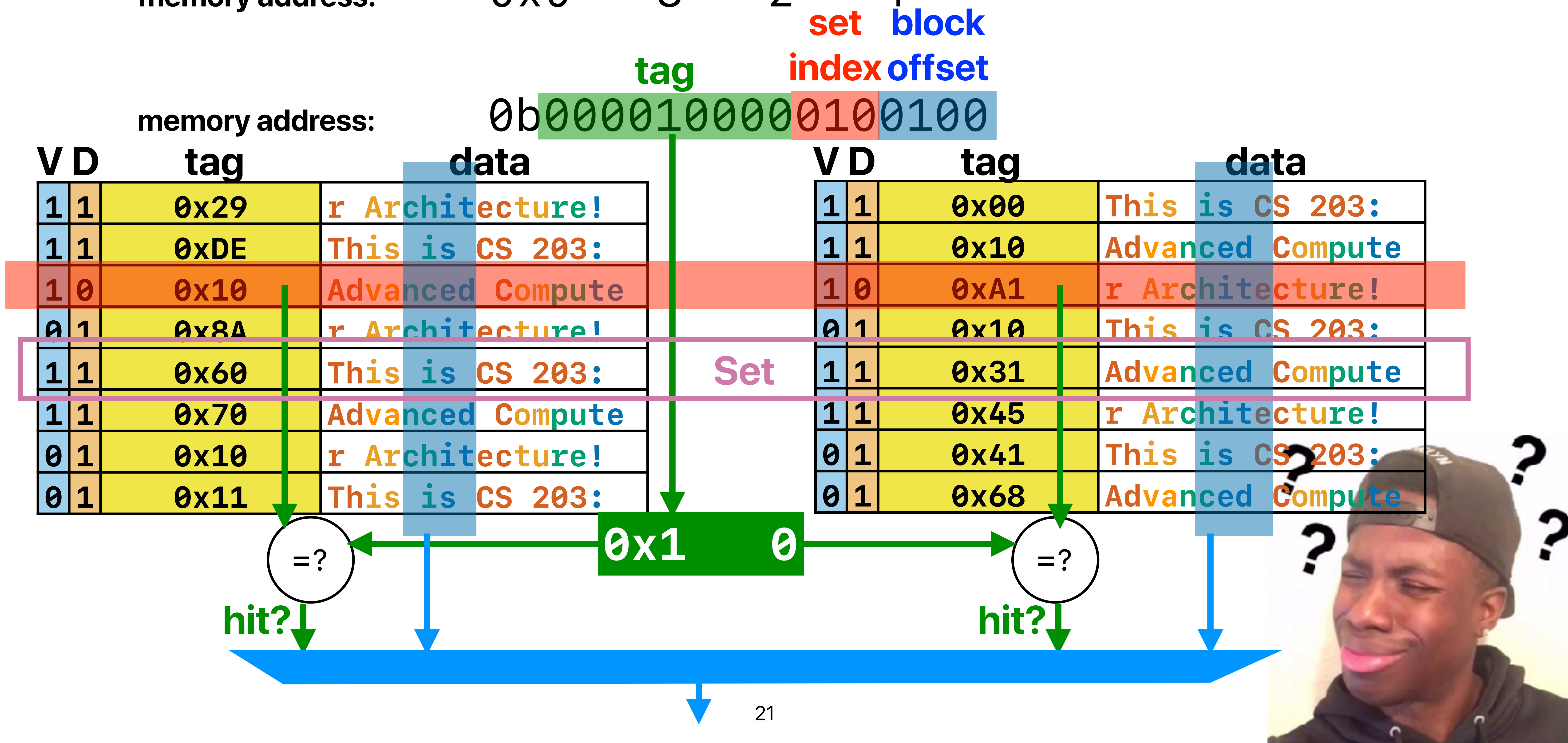
Take-aways: designing caches

- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b0000100000100100



V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

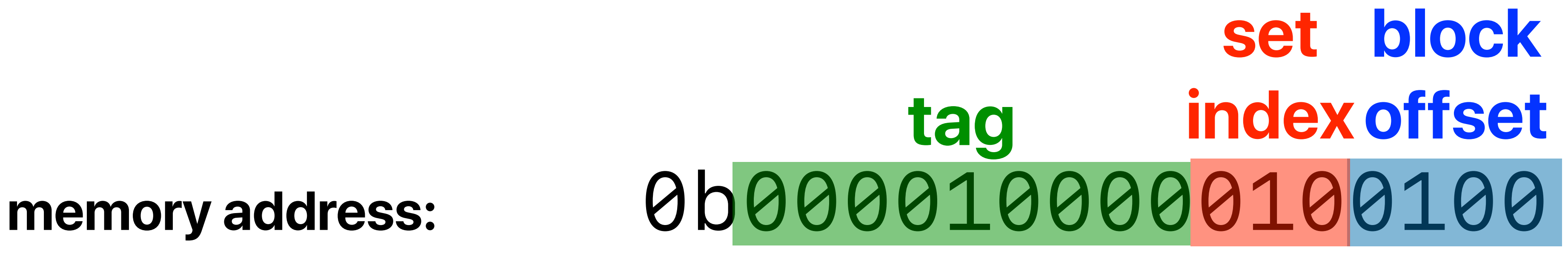
V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

The ABCs 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

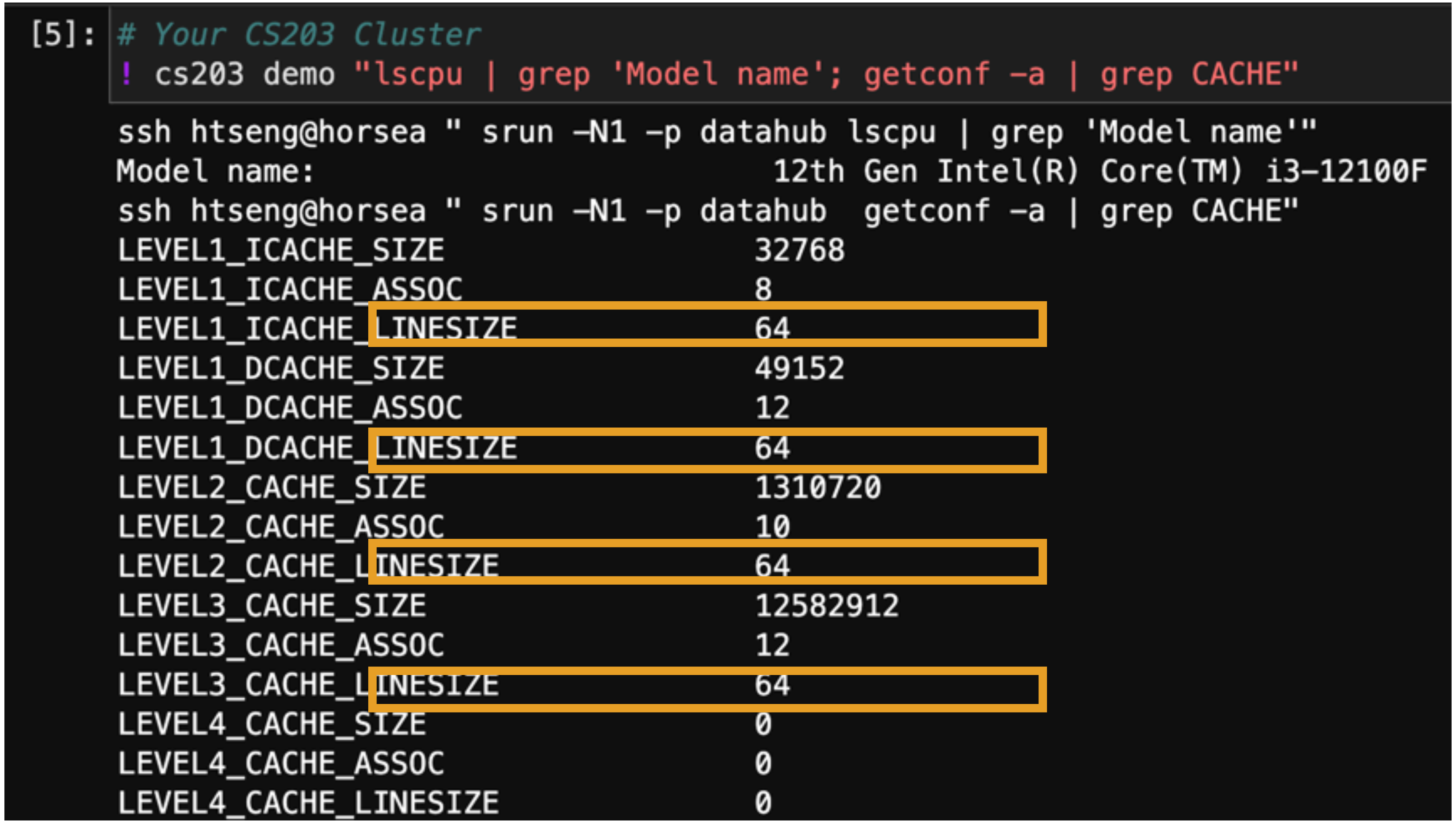
Corollary of C = ABS


memory address:

0b0000100000100100

- 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

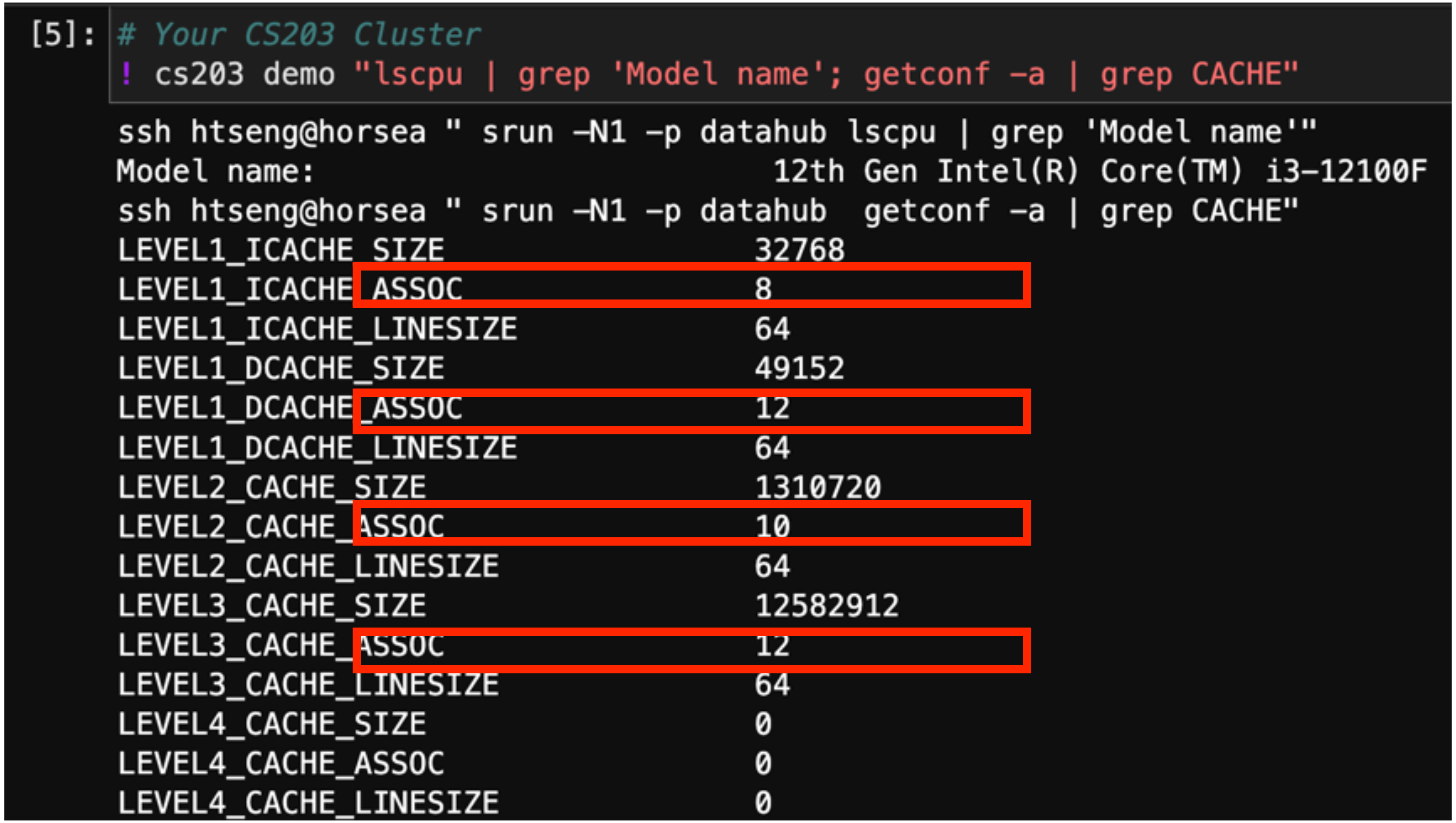
Blocksize == Linesize



```
[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
```

What is my 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
```

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]

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

$$32 \times 1024 = 4 \times 64 \times S$$

$$S = 128$$

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

$$Index = log_2(S) = log_2(128) = 7$$

$$Tag = 64 - 7 - 6 = 51$$

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

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

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

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

$$S = 64$$

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

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

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

Take-aways: designing caches

- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem
- C = A B S
  - C: capacity
  - A: Associativity
  - S: Number of sets
  - lg(S): Number of bits in set index
  - lg(B): Number of bits in block offset

Simulate the cache!

Simulate a direct-mapped cache

- A direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\dfrac{256}{16} = 16$
  - lg(16) = 4 : 4 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: `0x80000080`

$$= 0b\underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 0000}_{\text{tag}}\ \underbrace{1000}_{\text{index}}\ \underbrace{0000}_{\text{offset}}$$

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
     output[i] = result;
}
```

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
     for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
     }
      output[i] = result;
}
```

	tag index	tag index
	Address (Hex)	**Address (Binary)**
&a[0][0]	0x558FE0A1D330	0b10101011000111111100000101000011101001100110000
&b[0]	0x558FE0A1DC30	0b10101011000111111100000101000011101110000110000
&a[0][1]	0x558FE0A1D338	0b10101011000111111100000101000011101001100111000
&b[1]	0x558FE0A1DC38	0b10101011000111111100000101000011101110000111000
&a[0][2]	0x558FE0A1D340	0b10101011000111111100000101000011101001101000000
&b[2]	0x558FE0A1DC40	0b10101011000111111100000101000011101110001000000
&a[0][3]	0x558FE0A1D348	0b10101011000111111100000101000011101001101001000
&b[3]	0x558FE0A1DC48	0b10101011000111111100000101000011101110001001000
&a[0][4]	0x558FE0A1D350	0b10101011000111111100000101000011101001101010000
&b[4]	0x558FE0A1DC50	0b10101011000111111100000101000011101110001010000
&a[0][5]	0x558FE0A1D358	0b10101011000111111100000101000011101001101011000
&b[5]	0x558FE0A1DC58	0b10101011000111111100000101000011101110001011000
&a[0][6]	0x558FE0A1D360	0b10101011000111111100000101000011101001101100000
&b[6]	0x558FE0A1DC60	0b10101011000111111100000101000011101110001100000
&a[0][7]	0x558FE0A1D368	0b10101011000111111100000101000011101001101101000
&b[7]	0x558FE0A1DC68	0b10101011000111111100000101000011101110001101000
&a[0][8]	0x558FE0A1D370	0b10101011000111111100000101000011101001101110000
&b[8]	0x558FE0A1DC70	0b10101011000111111100000101000011101110001110000
&a[0][9]	0x558FE0A1D378	0b10101011000111111100000101000011101001101111000
&b[9]	0x558FE0A1DC78	0b10101011000111111100000101000011101110001111000

Simulate a direct-mapped cache

	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

This cache doesn't work!!!
— collisions!

tag index

	Address (Hex)	
&a[0][0]	0x558FE0A1D330	miss
&b[0]	0x558FE0A1DC30	miss
&a[0][1]	0x558FE0A1D338	miss
&b[1]	0x558FE0A1DC38	miss
&a[0][2]	0x558FE0A1D340	miss
&b[2]	0x558FE0A1DC40	miss
&a[0][3]	0x558FE0A1D348	miss
&b[3]	0x558FE0A1DC48	miss
&a[0][4]	0x558FE0A1D350	miss
&b[4]	0x558FE0A1DC50	miss
&a[0][5]	0x558FE0A1D358	miss
&b[5]	0x558FE0A1DC58	miss
&a[0][6]	0x558FE0A1D360	miss
&b[6]	0x558FE0A1DC60	miss
&a[0][7]	0x558FE0A1D368	miss
&b[7]	0x558FE0A1DC68	miss
&a[0][8]	0x558FE0A1D370	miss
&b[8]	0x558FE0A1DC70	miss
&a[0][9]	0x558FE0A1D378	
&b[9]	0x558FE0A1DC78	

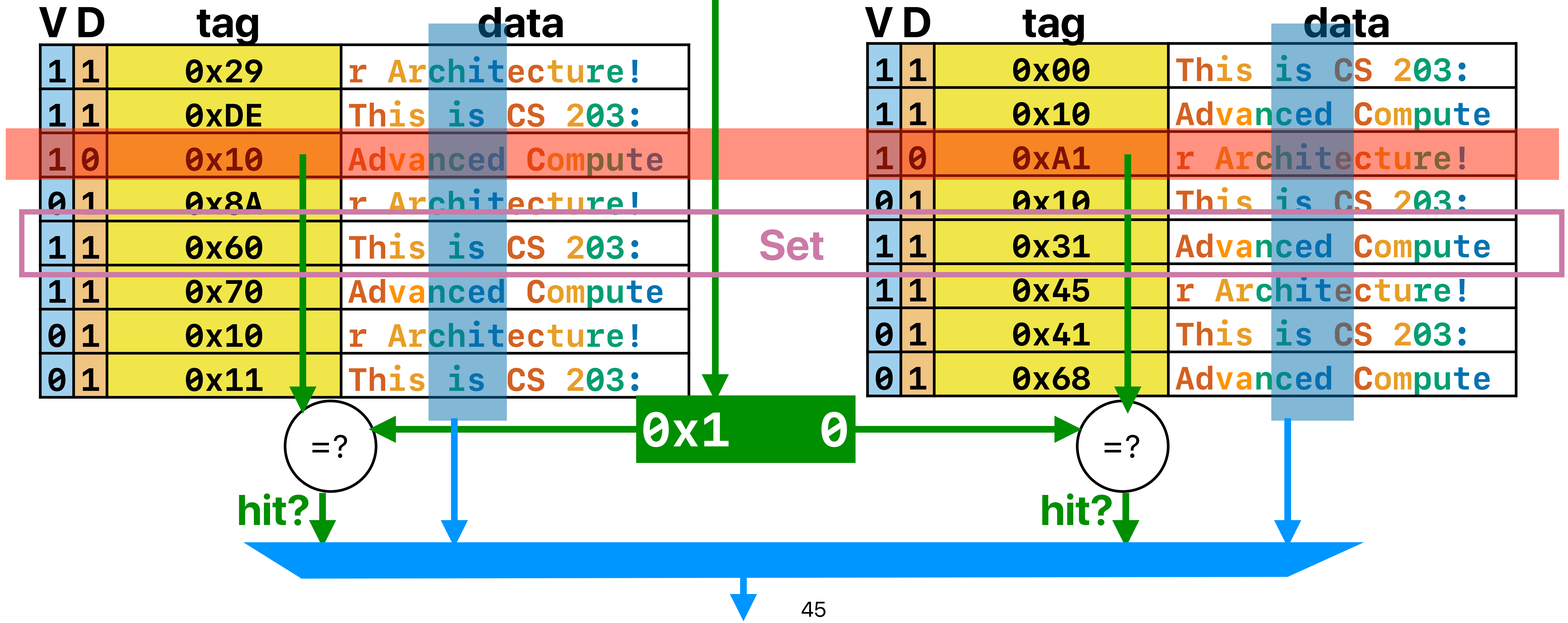
Way-associative cache

memory address: 0x0 8 2 4

set block

tag index offset

memory address: 0b0000100000100100



V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

Now, 2-way, same-sized cache

- A 2-way cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\frac{256}{16} = 16$
  - # of sets = $\frac{16}{2} = 8$ (2-way: 2 blocks in a set)
  - lg(8) = 3 : 3 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: 0x 8 0 0 0 0 0 8 0

    $= \texttt{0b}\underbrace{\texttt{1000 0000 0000 0000 0000 0000 1}}_{\text{tag}}\underbrace{\texttt{000}}_{\text{index}}\ \underbrace{\texttt{0000}}_{\text{offset}}$

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
     for(uint64_t j = 0; j < n; j++) {
           result += matrix[i][j]*vector[j];
     }
      output[i] = result;
}
```

	tag index	tag index
	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111111000001010000111010011001100000
&b[0]	0x558FE0A1DC30	0b1010101100011111111000001010000111011100001100000
&a[0][1]	0x558FE0A1D338	0b1010101100011111111000001010000111010011001111000
&b[1]	0x558FE0A1DC38	0b1010101100011111111000001010000111011100001111000
&a[0][2]	0x558FE0A1D340	0b1010101100011111111000001010000111010011010000000
&b[2]	0x558FE0A1DC40	0b1010101100011111111000001010000111011100010000000
&a[0][3]	0x558FE0A1D348	0b1010101100011111111000001010000111010011010011000
&b[3]	0x558FE0A1DC48	0b1010101100011111111000001010000111011100010011000
&a[0][4]	0x558FE0A1D350	0b1010101100011111111000001010000111010011010100000
&b[4]	0x558FE0A1DC50	0b1010101100011111111000001010000111011100010100000
&a[0][5]	0x558FE0A1D358	0b1010101100011111111000001010000111010011010111000
&b[5]	0x558FE0A1DC58	0b1010101100011111111000001010000111011100010111000
&a[0][6]	0x558FE0A1D360	0b1010101100011111111000001010000111010011011000000
&b[6]	0x558FE0A1DC60	0b1010101100011111111000001010000111011100011000000
&a[0][7]	0x558FE0A1D368	0b1010101100011111111000001010000111010011011011000
&b[7]	0x558FE0A1DC68	0b1010101100011111111000001010000111011100011011000
&a[0][8]	0x558FE0A1D370	0b1010101100011111111000001010000111010011011100000
&b[8]	0x558FE0A1DC70	0b1010101100011111111000001010000111011100011100000
&a[0][9]	0x558FE0A1D378	0b1010101100011111111000001010000111010011011111000
&b[9]	0x558FE0A1DC78	0b1010101100011111111000001010000111011100011111000

Simulate a 2-way cache

V	D	Tag	Data	V	D	Tag	Data
0	0			0	0		
0	0			0	0		
0	0			0	0		
1	0	0xAB1FC143A6	a[0][0], a[0][1]	1	0	0xAB1FC143B8	b[0], b[1]
1	0	0xAB1FC143A6	a[0][2], a[0][3]	1	0	0xAB1FC143B8	b[2], b[3]
0	0			0	0		
0	0			0	0		
0	0			0	0		

	Address (Hex)	Tag	Index	
&a[0][0]	0x558FE0A1D330	0xAB1FC143A6	0x3	miss
&b[0]	0x558FE0A1DC30	0xAB1FC143B8	0x3	miss
&a[0][1]	0x558FE0A1D338	0xAB1FC143A6	0x3	hit
&b[1]	0x558FE0A1DC38	0xAB1FC143B8	0x3	hit
&a[0][2]	0x558FE0A1D340	0xAB1FC143A6	0x4	miss
&b[2]	0x558FE0A1DC40	0xAB1FC143B8	0x4	miss
&a[0][3]	0x558FE0A1D348	0xAB1FC143A6	0x4	hit
&b[3]	0x558FE0A1DC48	0xAB1FC143B8	0x4	hit
&a[0][4]	0x558FE0A1D350	0xAB1FC143A6	0x5	miss
&b[4]	0x558FE0A1DC50	0xAB1FC143B8	0x5	miss
&a[0][5]	0x558FE0A1D358	0xAB1FC143A6	0x5	hit
&b[5]	0x558FE0A1DC58	0xAB1FC143B8	0x5	hit
&a[0][6]	0x558FE0A1D360	0xAB1FC143A6	0x6	miss
&b[6]	0x558FE0A1DC60	0xAB1FC143B8	0x6	miss
&a[0][7]	0x558FE0A1D368	0xAB1FC143A6	0x6	hit
&b[7]	0x558FE0A1DC68	0xAB1FC143B8	0x6	hit
&a[0][8]	0x558FE0A1D370	0xAB1FC143A6	0x7	miss
&b[8]	0x558FE0A1DC70	0xAB1FC143B8	0x7	miss
&a[0][9]	0x558FE0A1D378	0xAB1FC143A6	0x7	hit
&b[9]	0x558FE0A1DC78	0xAB1FC143B8	0x7	hit

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

NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

NVIDIA Tegra X1

100% miss rate!

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS

32KB = 4 * 64 * S

S = 128

offset = lg(64) = 6 bits

index = lg(128) = 7 bits

tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b00010000000000000000000	0x8	0x0	Miss	
b[0]	0x20000	0b00100000000000000000000	0x10	0x0	Miss	
c[0]	0x30000	0b00110000000000000000000	0x18	0x0	Miss	
d[0]	0x40000	0b01000000000000000000000	0x20	0x0	Miss	
e[0]	0x50000	0b01010000000000000000000	0x28	0x0	Miss	a[0-7]
a[1]	0x10008	0b00010000000000000001000	0x8	0x0	Miss	b[0-7]
b[1]	0x20008	0b00100000000000000001000	0x10	0x0	Miss	c[0-7]
c[1]	0x30008	0b00110000000000000001000	0x18	0x0	Miss	d[0-7]
d[1]	0x40008	0b01000000000000000001000	0x20	0x0	Miss	e[0-7]
e[1]	0x50008	0b01010000000000000001000	0x28	0x0	Miss	a[0-7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

NVIDIA Tegra X1 (cont.)

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS

32KB = 4 * 64 * S

S = 128

offset = lg(64) = 6 bits

index = lg(128) = 7 bits

tag = the rest bits

tag **index** **offset**

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x8	0x0	Miss	
b[7]	0x20038	0b00100000000000111000	0x10	0x0	Miss	
c[7]	0x30038	0b00110000000000111000	0x18	0x0	Miss	
d[7]	0x40038	0b01000000000000111000	0x20	0x0	Miss	
e[7]	0x50038	0b01010000000000111000	0x28	0x0	Miss	a[0-7]
a[8]	0x10040	0b00010000000001000000	0x8	0x1	Miss	
b[8]	0x20040	0b00100000000001000000	0x10	0x1	Miss	
c[8]	0x30040	0b00110000000001000000	0x18	0x1	Miss	
d[8]	0x40040	0b01000000000001000000	0x20	0x1	Miss	
e[8]	0x50040	0b01010000000001000000	0x28	0x1	Miss	a[8-15]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

100% miss rate!

NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What’s the data cache miss rate for this code?

A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

[✓ E. 100%]

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

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What’s the data cache miss rate for this code?

A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

intel Core i7

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS

48KB = 12 * 64 * S

S = 64

offset = lg(64) = 6 bits

index = lg(64) = 6 bits

tag = the rest bits

tag **index** **offset**

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b00010000000000000000	0x10	0x0	Miss	
b[0]	0x20000	0b00100000000000000000	0x20	0x0	Miss	
c[0]	0x30000	0b00110000000000000000	0x30	0x0	Miss	
d[0]	0x40000	0b01000000000000000000	0x40	0x0	Miss	
e[0]	0x50000	0b01010000000000000000	0x50	0x0	Miss	
a[1]	0x10008	0b00010000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b00100000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b00110000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b01000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b01010000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7 (cont.)

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

C = ABS

48KB = 12 * 64 * S

S = 64

offset = lg(64) = 6 bits

index = lg(64) = 6 bits

tag = the rest bits

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b00100000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b00110000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b01000000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b01010000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b00010000000001000000	0x10	0x1	Miss	
b[8]	0x20040	0b00100000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b00110000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b01000000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b01010000000001000000	0x50	0x1	Miss	
a[9]	0x10048	0b00010000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b00100000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b00110000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b01000000000001001000	0x40	0x1	Hit	

$$\frac{5 \times \frac{512}{8}}{5 \times 512} = \frac{1}{8} = 12.5\,\%$$

Miss when the array index is a multiply of 8!

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What’s the data cache miss rate for this code?

[✓ A. 12.5%]

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

Computer
Science &
Engineering

203

つづく