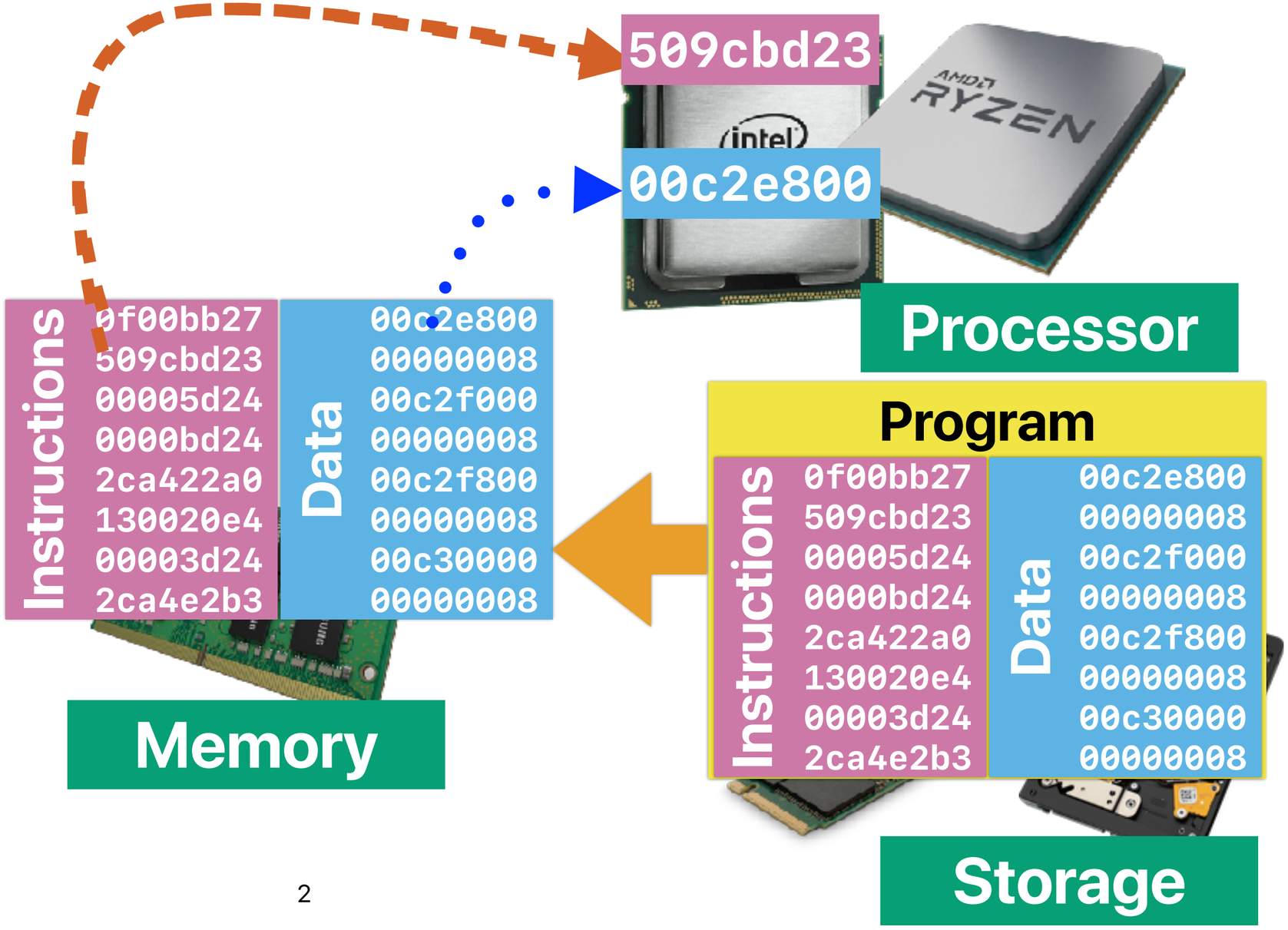
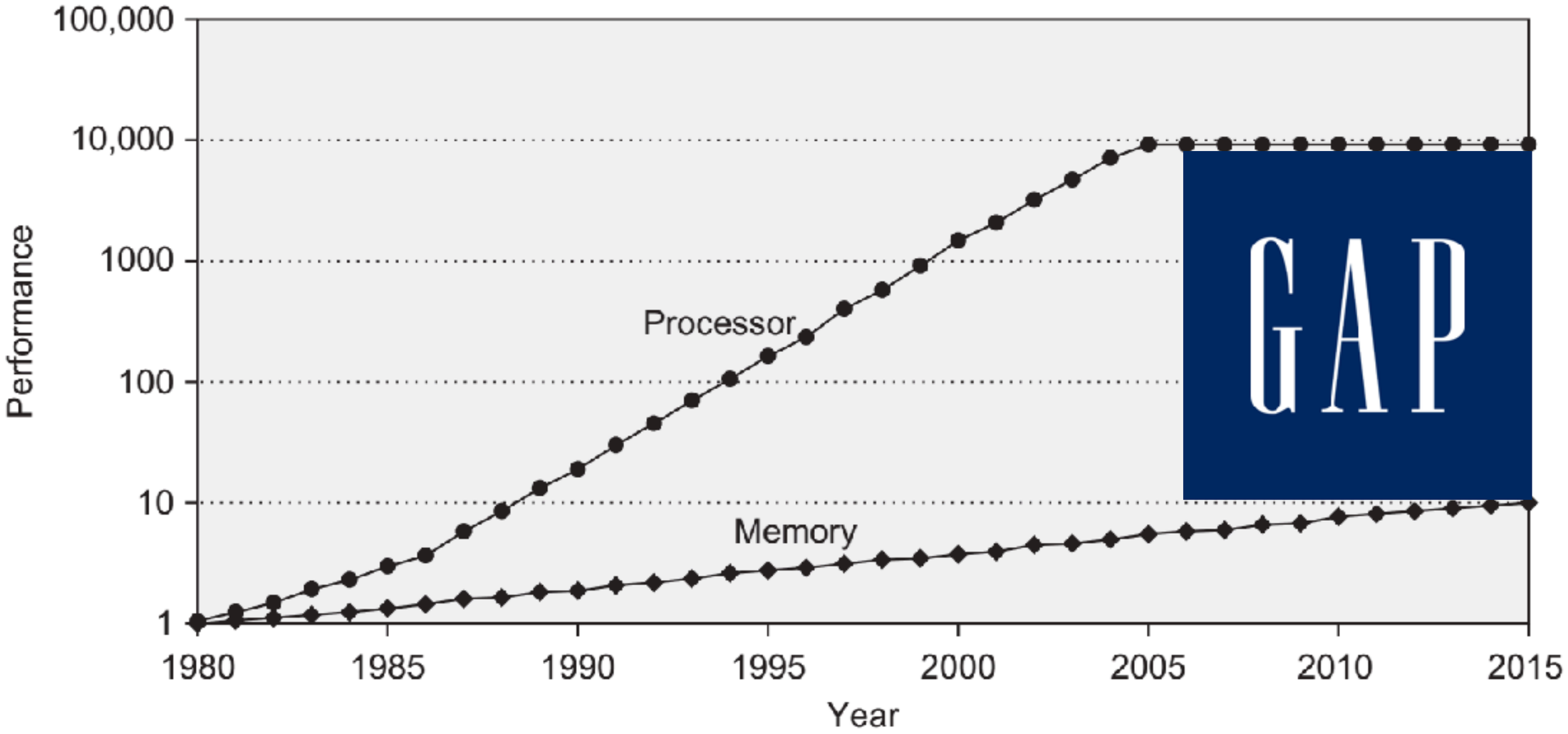
Memory Hierarchy Inside Out: (2) The A, B, C s of caches

Hung-Wei Tseng

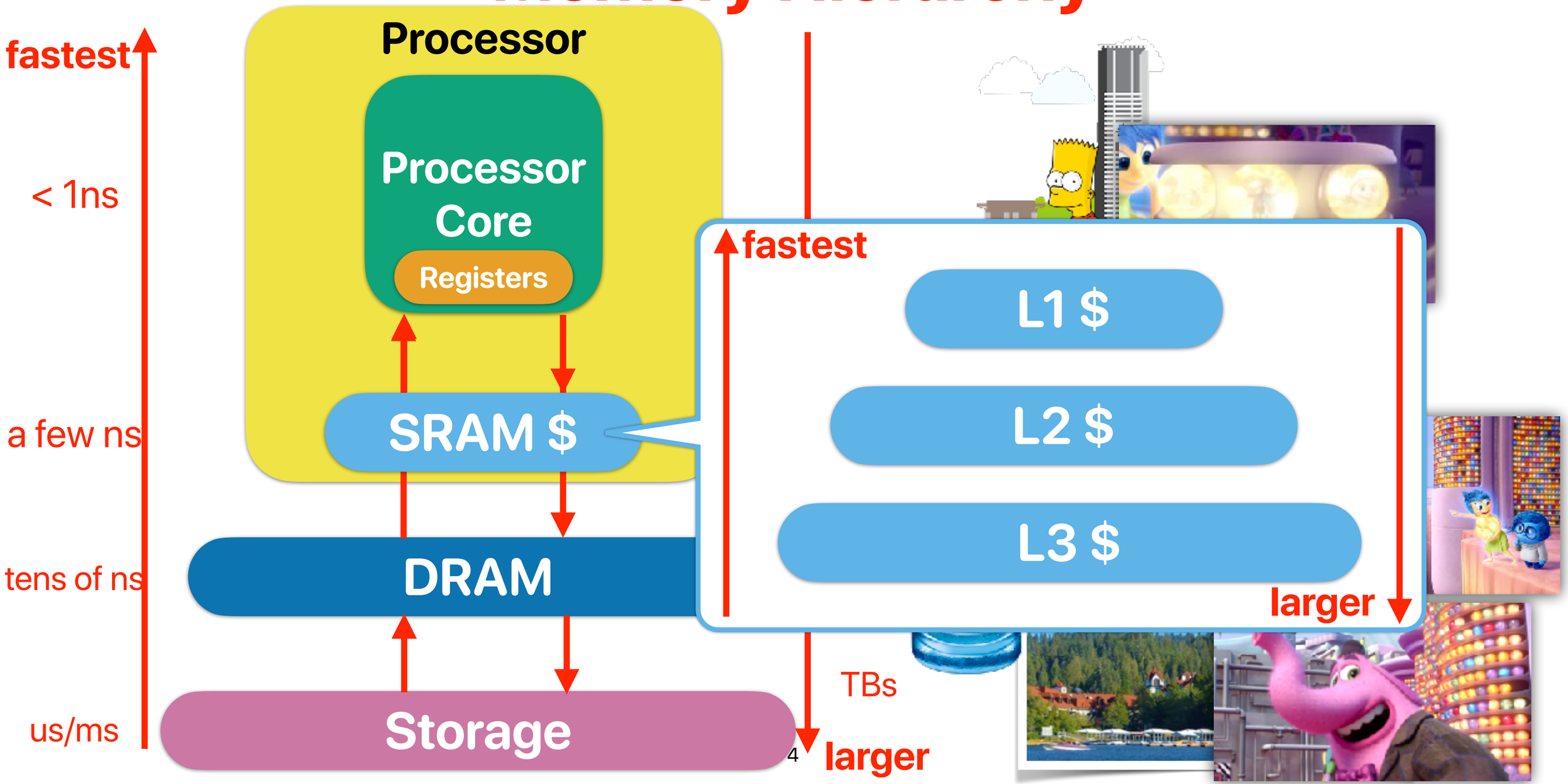
von Neumman Architecture



Recap: Performance gap between Processor/Memory



Memory Hierarchy



Outline

- Locality in applications (cont.)
- Architecting the cache
- The A, B, Cs of the cache

Why adding small SRAMs would work?

Because of localities of memory references!

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 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!

How do you prepare closed-book exams?

• Review questions from prior years	**Temporal locality**
• Review the whole chapter	**Spatial locality**
• Practice similar questions	**Spatial locality**
• Practice many times	**Temporal locality**

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then the next
  - Data — the current element in an array, then the next
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

Most of time, your program is just visiting a limited amount of data/instructions within a given timeframe

Locality and cache design

- The cache must be able to get chunks of near-by items every time to exploit spatial locality

- The cache must be able to keep a frequently used block for a while to exploit temporal locality

Architecting the Cache: capture the localities!

Locality and cache design

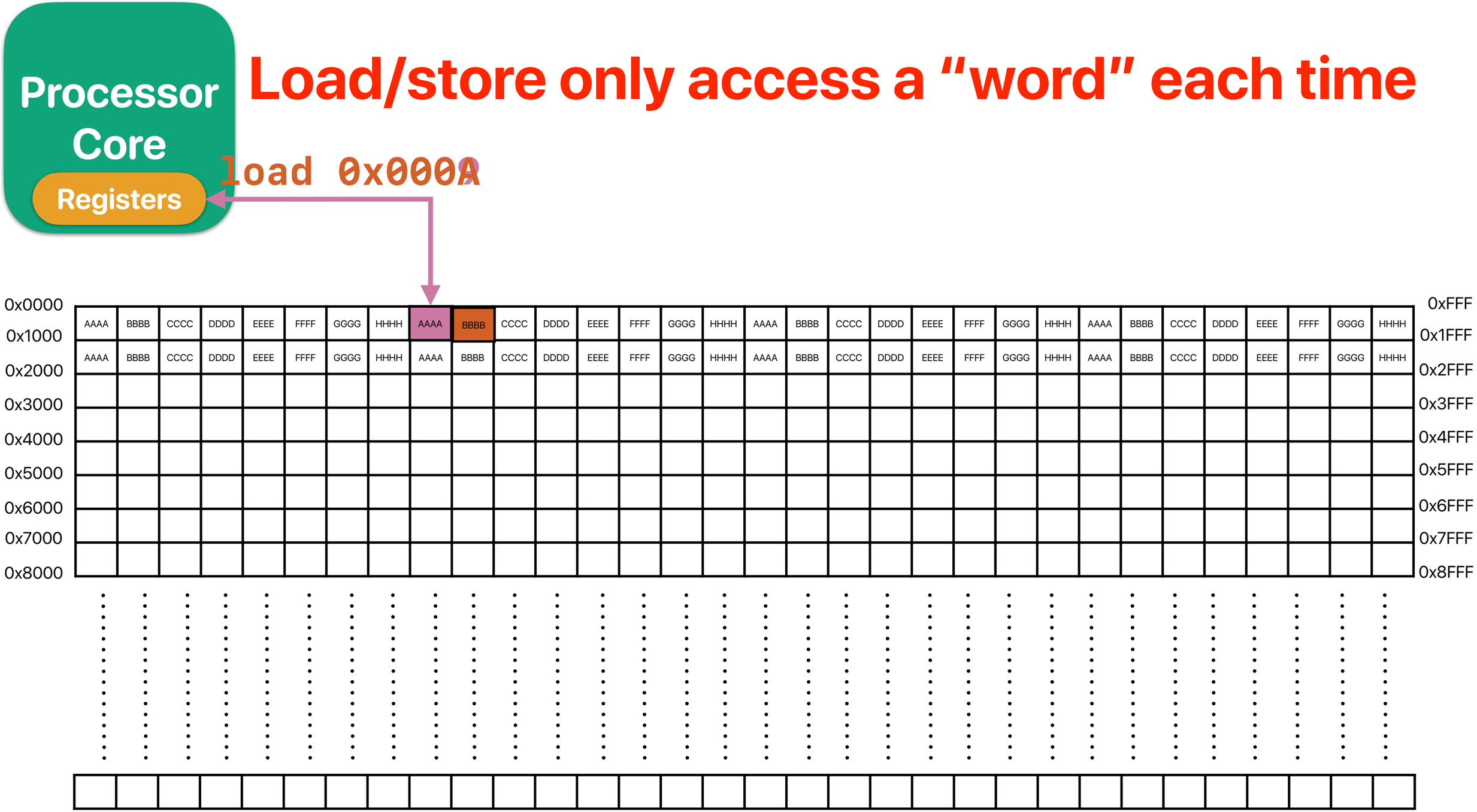
- The cache must be able to get chunks of near-by items every time to exploit spatial locality

  We need to keep a block of data every time we put things in the cache

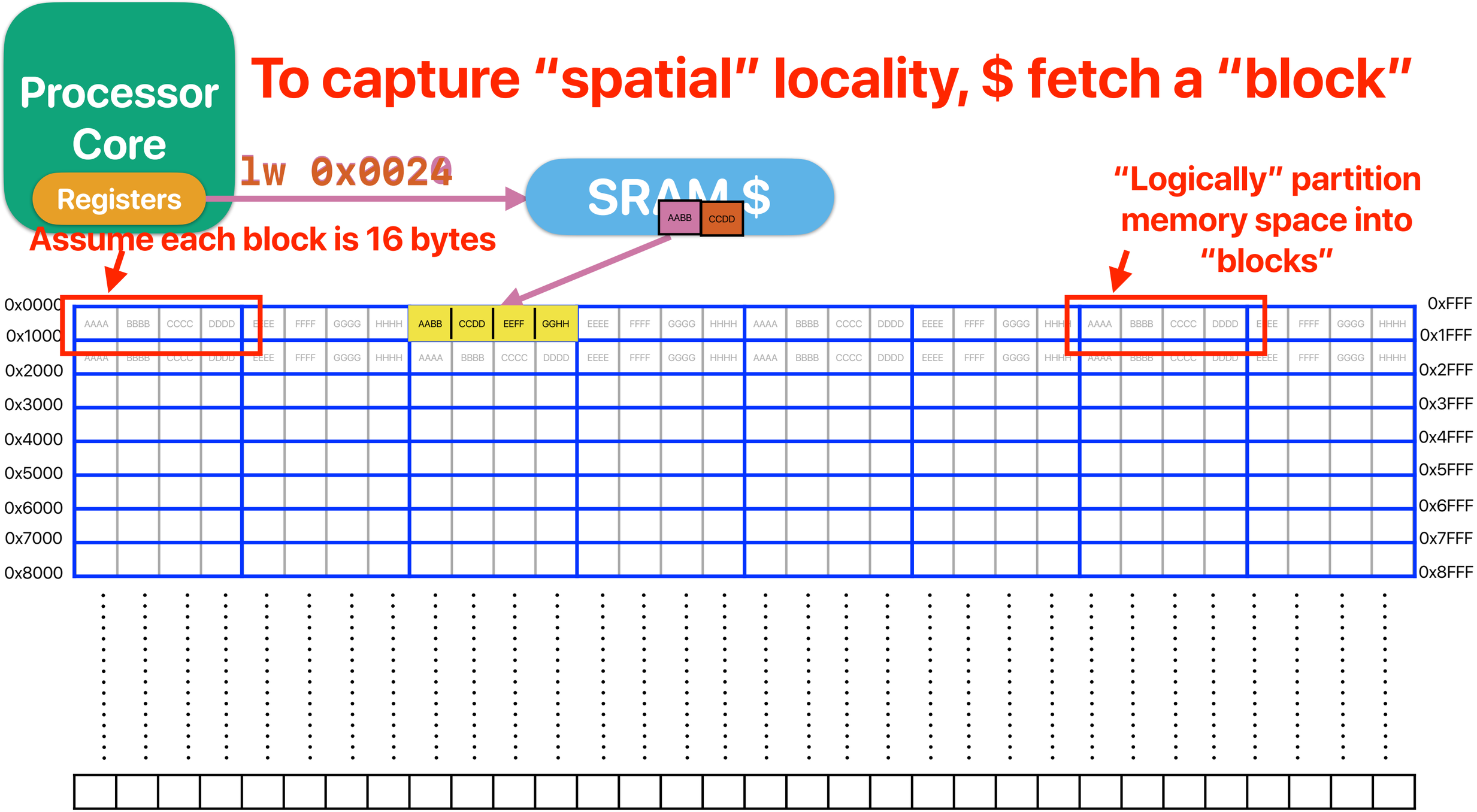
- The cache must be able to keep a frequently used block for a while to exploit temporal locality

  We need to keep multiple blocks of data in the cache

Load/store only access a “word” each time



To capture "spatial" locality, $ fetch a "block"



What's a block?

the offset of the byte within a block

the data in memory

```
0123456789ABCDEF
0x0000, 0x0001, 0x0002, …, 0x000F   This is CS 203:
0x0010, 0x0011, 0x0012, …, 0x001F   Advanced Compute
```

the byte addresses of each byte in the 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	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

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

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:

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

block offset

tag

index

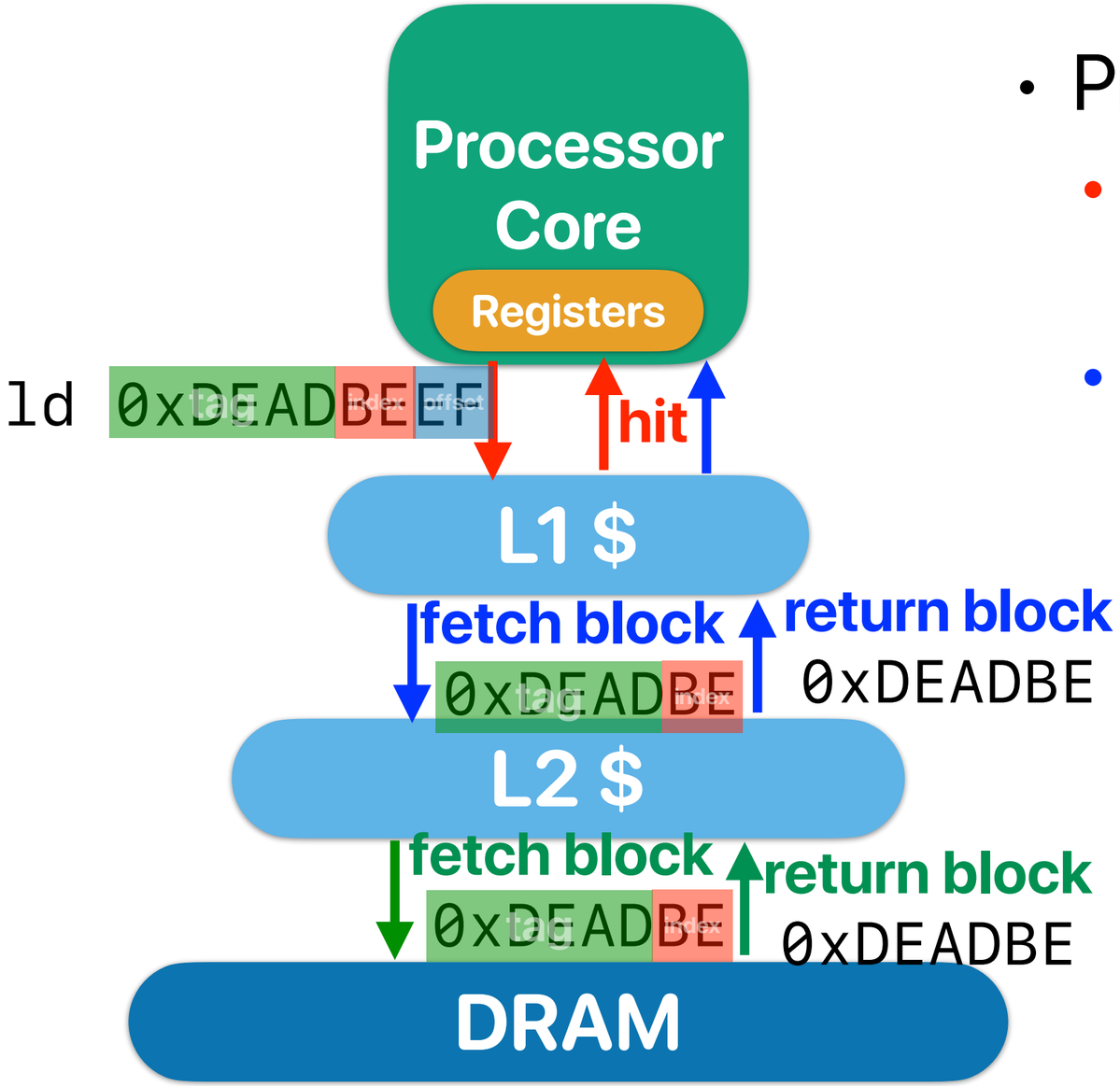
load 0x0008

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:

What happens when we read data



- Processor sends load request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - Fetch a block
    - Select a victim block
      - If the target is not occupied — place the fetched block in the target location
      - If the target is full — select a victim block using some policy

Let’s simulate the simple 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 = $\frac{256}{16} = 16$
  - 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: `0x 8 0 0 0 0 0 8 0`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
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;
 }
```

Matrix vector revisited

```
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;
}
```

tag index tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

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	

Hash-like structure — direct-mapped cache

block offset

tag

index

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:

The biggest issue with hash is —
Collision!

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b00001000001000100

tag | set index | block offset

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

Set

0x1 0

=? =?

hit? hit?

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`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
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;
}
```

tag index tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

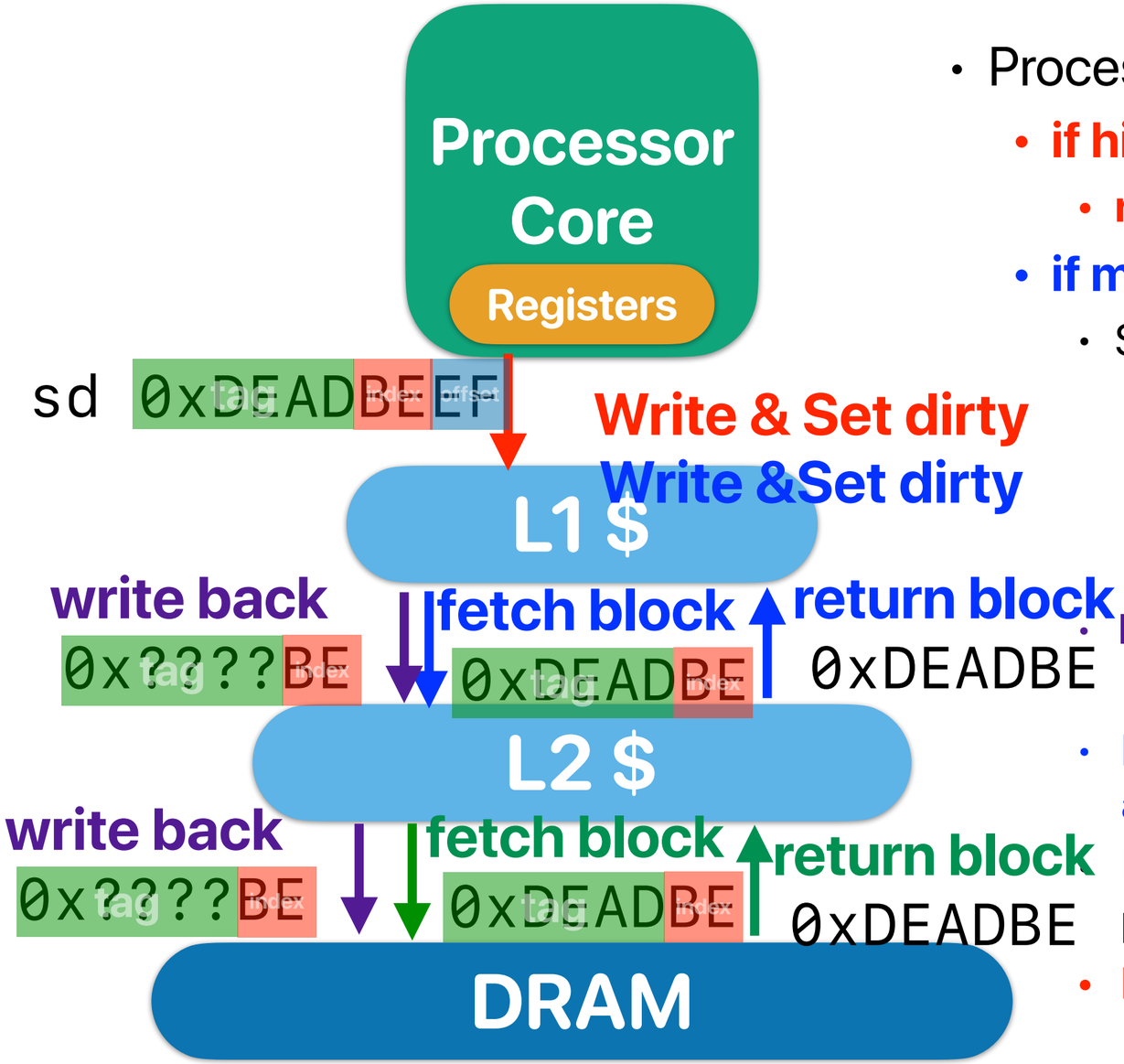
Simulate a 2-way cache

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

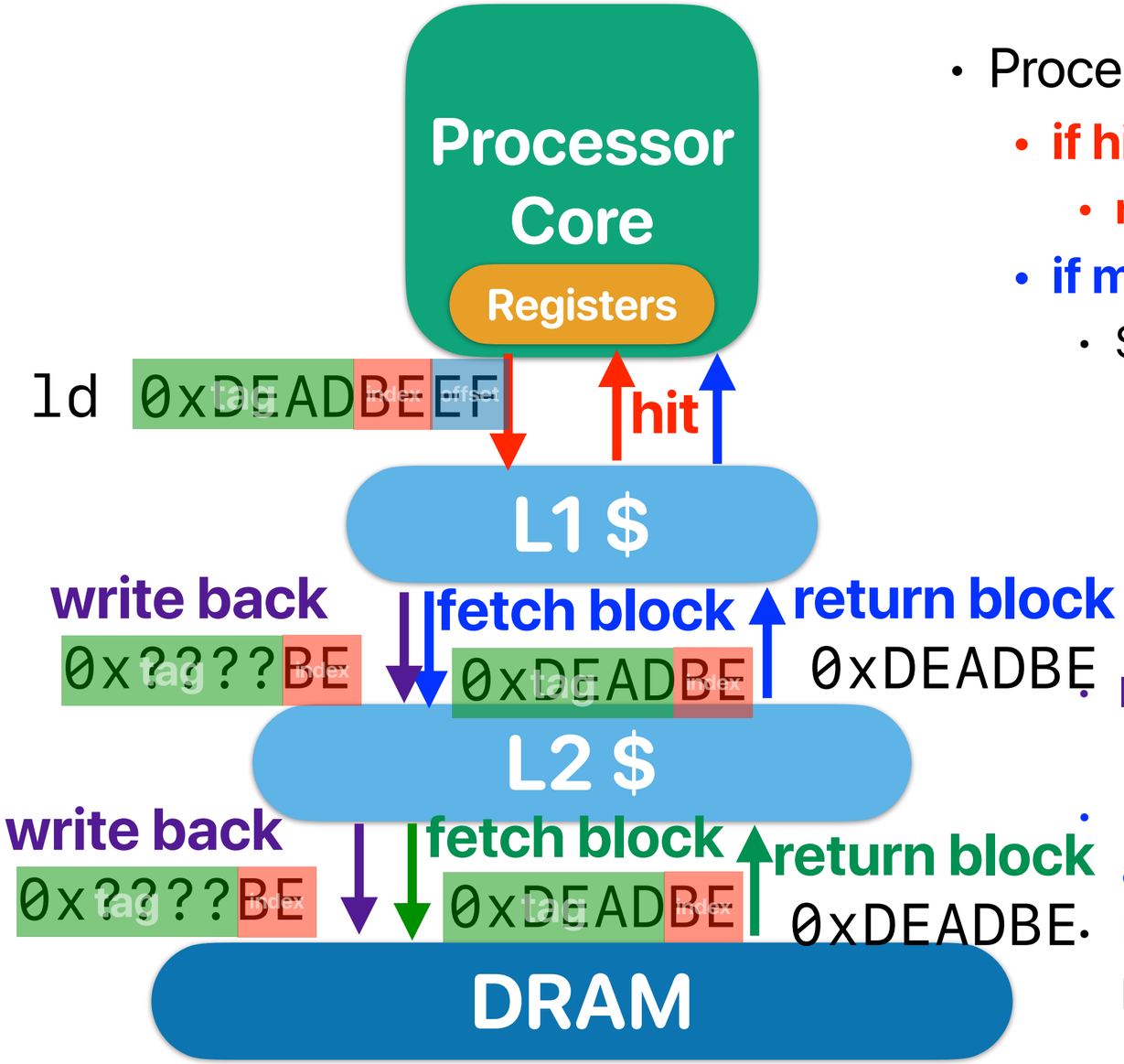
Put everything all together: How cache interacts with CPU

What happens when we write data



- Processor sends load request to L1-$
  - **if hit**
    - **return data — 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**

What happens when we read data



- Processor sends load request to L1-$
  - **if hit**
    - **return data**
  - **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

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b00001000001000100

tag | set index | block offset

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

Set

0x1 0

=? =?

hit? hit?

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 (Cacheline)
  - 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

	tag	set index	block offset
memory address:	0b00001000	0100	0100

- 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 64 bits for 64-bit machine
- $\frac{address}{block\_size} \pmod{S}$ = 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

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

$$32KB = 4 \times 64B \times S$$

$$S = \frac{32KB}{4 \times 64B} = 128$$

$$index = log_2(128) = 7\ bits$$

$$offset = log_2(64) = 6\ bits$$

$$tag = 64 - 7 - 6 = 51\ bits$$

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

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-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

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-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

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

$$32KB = 8 \times 64B \times S$$

$$S = \frac{32KB}{8 \times 64B} = 64$$

$$index = log_2(64) = 6\ bits$$

$$offset = log_2(64) = 6\ bits$$

$$tag = 64 - 6 - 6 = 52\ bits$$

Announcement

- Regarding assignments
  - We will drop your lowest scored assignment
    - Please don't email to ask for extension — the dropping policy is to accommodate any potential reason for that
    - We don't accept late assignment
  - Please follow the EXACT instructions — any small thing you missed in the document can lead to undesirable outcome
  - Start early
    - We don't work 24/7 and we cannot help you last minute
    - Server could get busy last minute, too.
    - Gradescope has different test cases than released ones to prevent any shortcut of performance results — you have to test your code carefully to prevent failed execution on gradescope.
  - Assignment 2 is up tomorrow and please START EARLY
  - C++ programming — can't the demo regarding performance convince you to use C/C++?
  - Any kind of cheating is NOT ALLOWED and each assignment should be an individual work
    - Gradescope already identified several 100% similar ones — we will send those cases to Student Conduct & Academic Integrity Programs Office

First & Last Name Swap	File	Match Length	% Similarity	Top Source
Si	sum.cpp	96	100%	S
Si	sum.cpp	96	98%	S

Announcement (cont.)

- Reading quiz due next Tuesday

Computer
Science &
Engineering

203

つづく