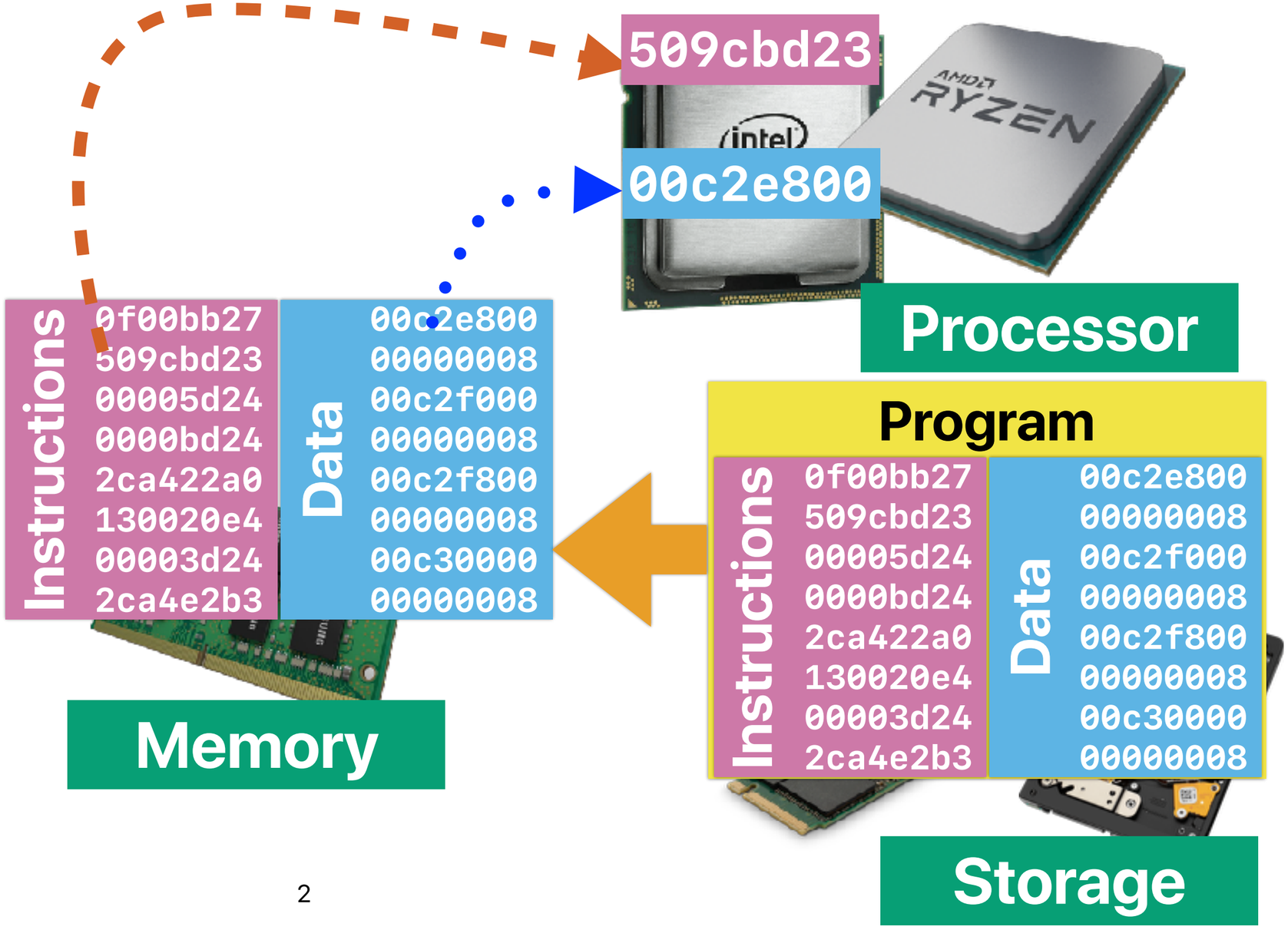
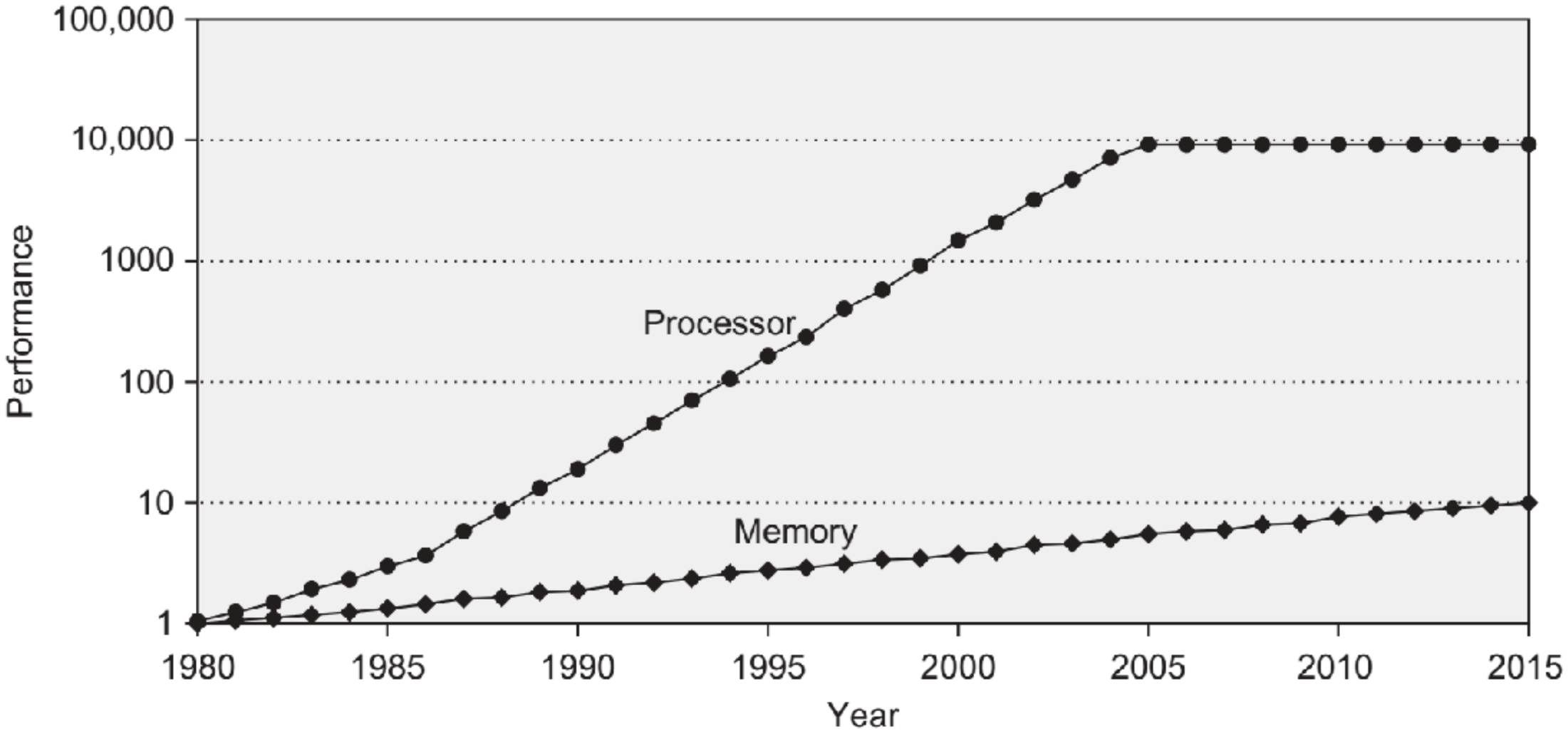
Memory Hierarchy (II)

Hung-Wei Tseng

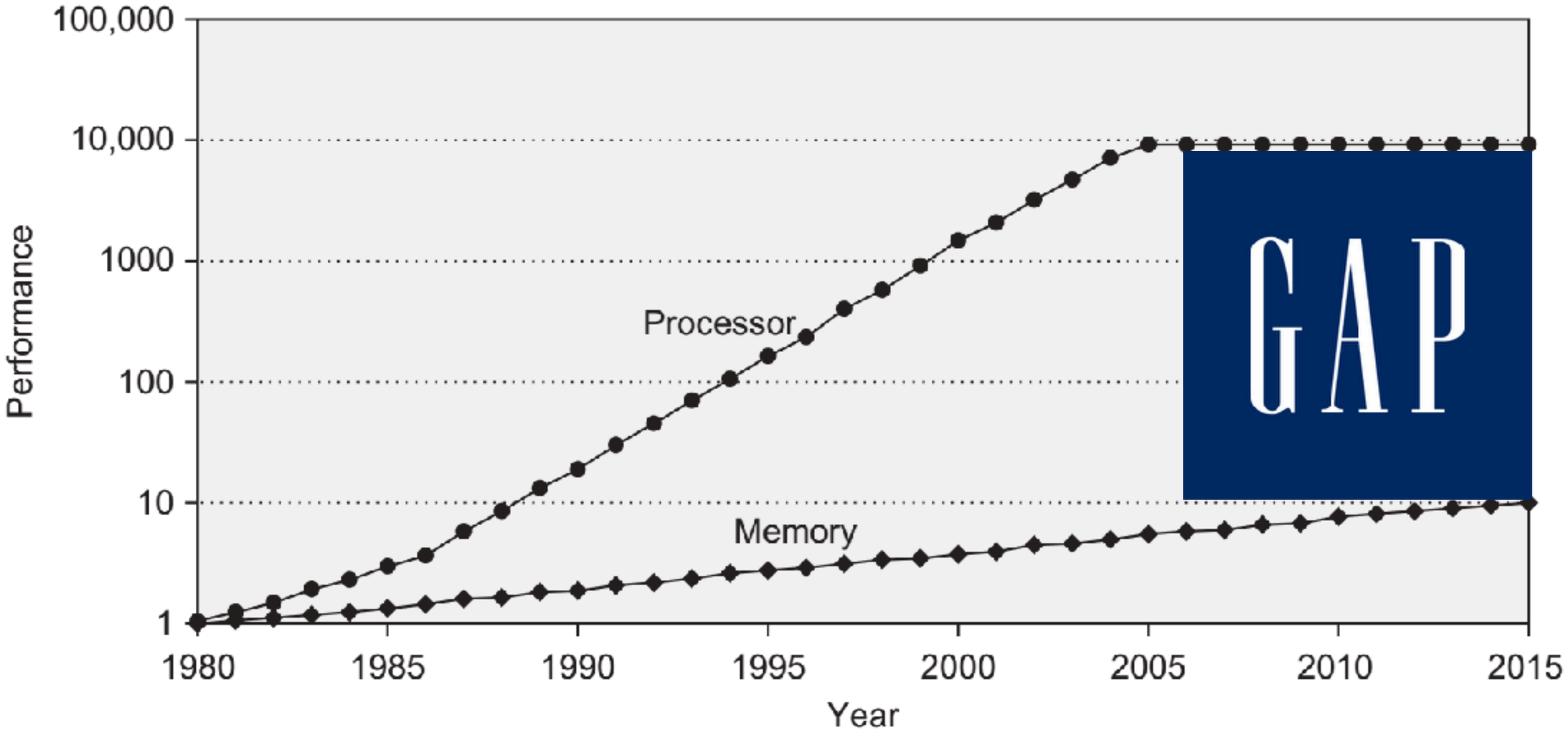
Recap: von Neumman Architecture



Recap: Performance gap between Processor/Memory



Recap: Performance gap between Processor/Memory



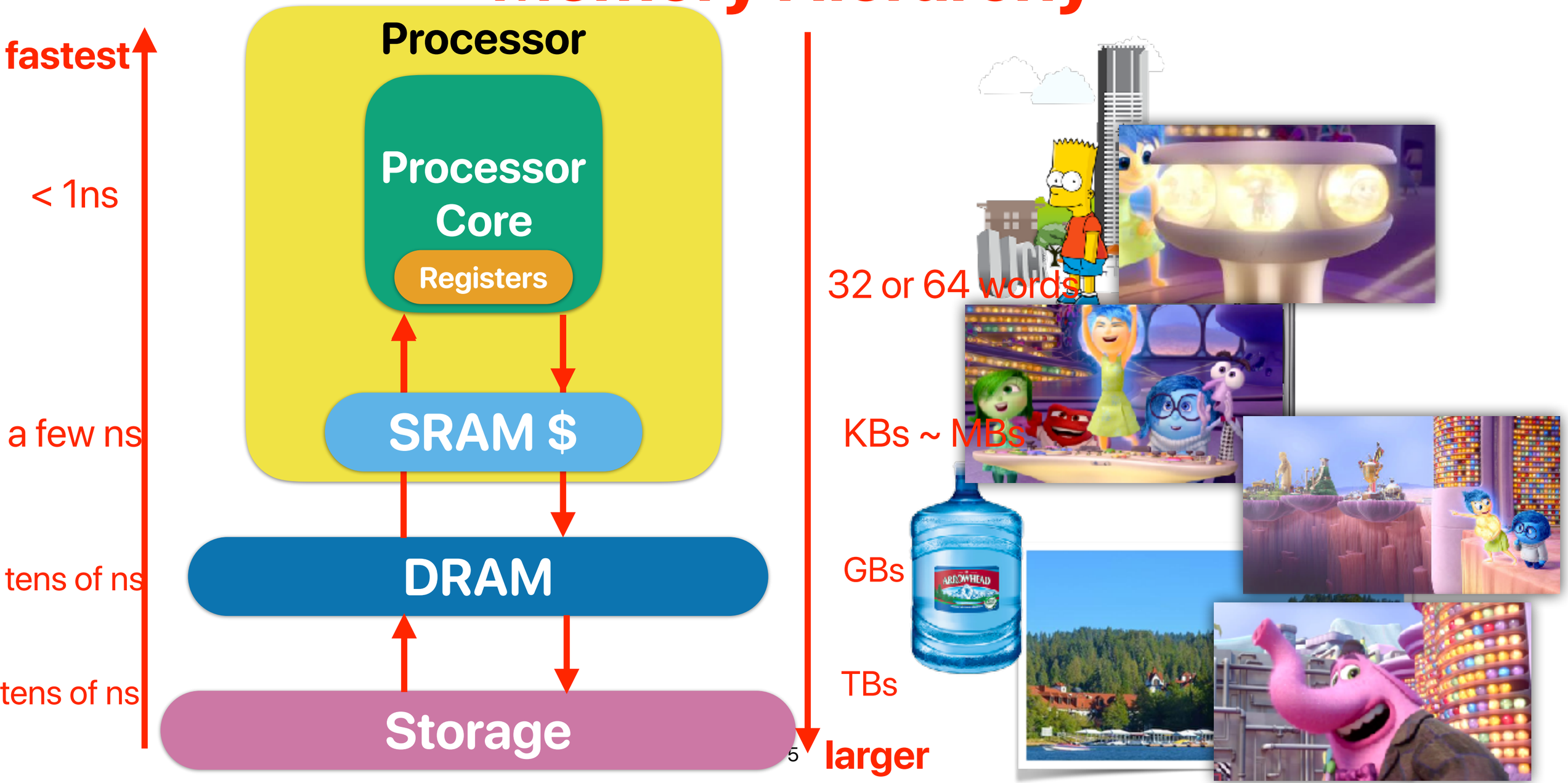
Recap: The impact of "slow" memory

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has "perfect" memory, the CPI is just 1. Now, consider we have DDR4 and the program is well-behaved that precharge is never necessary — the access latency is simply 26 ns. What's the average CPI (pick the most close one)?

  A. 9
  B. 17
  C. 27
  D. 35
  E. 69

$$1 + 100\% \times (52) + 30\% \times 52 = 68.6\ cycles$$

Memory Hierarchy



Recap: How can memory hierarchy help in performance?

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has "perfect" memory, the CPI is just 1. Now, in addition to DDR4, whose latency 26 ns, we also got an SRAM cache with latency of just at 0.5ns and can capture 90% of the desired data/instructions. what's the average CPI (pick the most close one)?

  A. 2

  B. 4

  C. 8 $\quad 1 + (1 - 90\%) \times [100\% \times (52) + 30\% \times 52] = 7.76\ cycles$

  D. 16

  E. 32

Outline

- Why can small pieces of SRAMs improve performance
- Architecting the cache
- Put all together
- Cache simulation

L1? L2? L3?

CPU-Z - ID : wswpbb

CPU | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name	AMD Ryzen 7 2700X		
Code Name	Pinnacle Ridge	Max TDP	105 W
Package	Socket AM4 (1331)		
Technology	12 nm	Core Voltage	1.36 V
Specification	AMD Ryzen 7 2700X Eight-Core Processor		
Family	F	Model	8	Stepping	2
Ext. Family	17	Ext. Model	8	Revision	PiR-B2
Instructions	MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A x86-64, AMD-V, AES, AVX, AVX2, FMA3, SHA		

Clocks (Core #0)

Core Speed	4290.73 MHz
Multiplier	x 43.0
Bus Speed	99.78 MHz
Rated FSB	

Cache

L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 64 KBytes	4-way
Level 2	8 x 512 KBytes	8-way
Level 3	2 x 8192 KBytes	16-way

Selection: Processor #1 | Cores: 8 | Threads: 16

CPU-Z Ver. 1.86.0.x64 | Tools | Validate | Close

CPU | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name	Intel Core i7 9700K		
Code Name	Coffee Lake	Max TDP	95.0 W
Package	Socket 1151 LGA		
Technology	14 nm	Core Voltage	0.737 V
Specification	Intel® Core™ i7-9700K CPU @ 3.60GHz (ES)		
Family	6	Model	E	Stepping	C
Ext. Family	6	Ext. Model	9E	Revision	P0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3, TSX		

Clocks (Core #0)

Core Speed	4798.85 MHz
Multiplier	x 48.0 (8 - 49)
Bus Speed	99.98 MHz
Rated FSB	

Cache

L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 32 KBytes	8-way
Level 2	8 x 256 KBytes	4-way
Level 3	12 MBytes	12-way

Selection: Socket #1 | Cores: 8 | Threads: 8

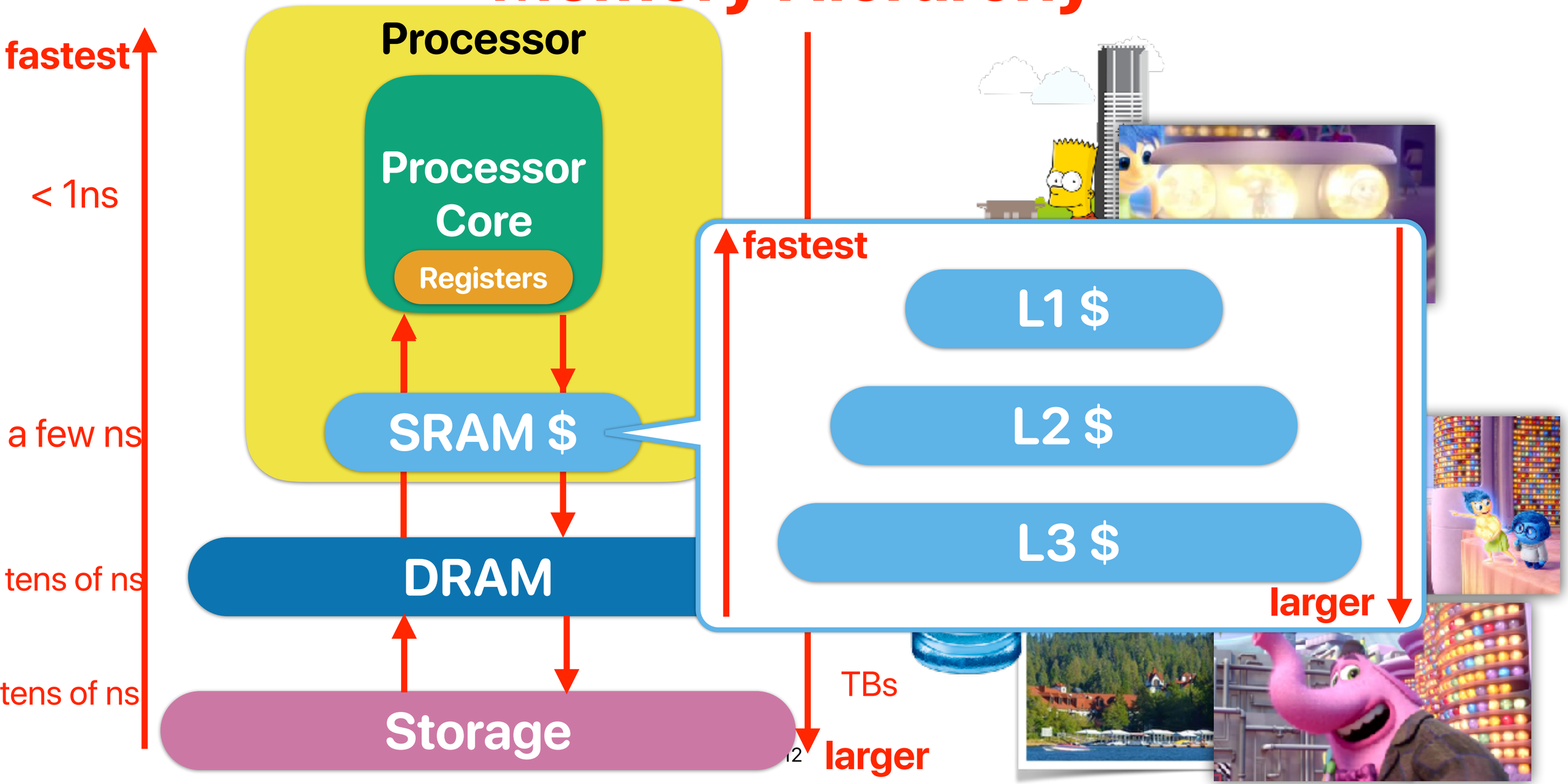
How can deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has "perfect" memory, the CPI is just 1. Now, in addition to DDR4, whose latency 26 ns, we also got a 2-level SRAM caches with
  - it's 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5ns and can capture 60% of the desired data/instructions

  What's the average CPI (pick the most close one)?

  A. 2
  B. 4 $1 + (1 - 90\%) \times [10 + (1 - 60\%) \times 52 + 30\% \times (10 + (1 - 60\%) \times 52)] = 5\ cycles$
  C. 8
  D. 16
  E. 32

Memory Hierarchy



Why adding small SRAMs would work?

Locality

- Which description about locality of arrays `sum` and A in the following code is the most accurate?

```
for(i = 0; i< 100000; i++)
{
    sum[i%10] += A[i];
}
```

spatial locality:
A[0], A[1], A[2], A[3], ...
sum[0], sum[1], ... , sum[9]
temporal locality:
reuse of sum[0], sum[1], ... , sum[9]

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

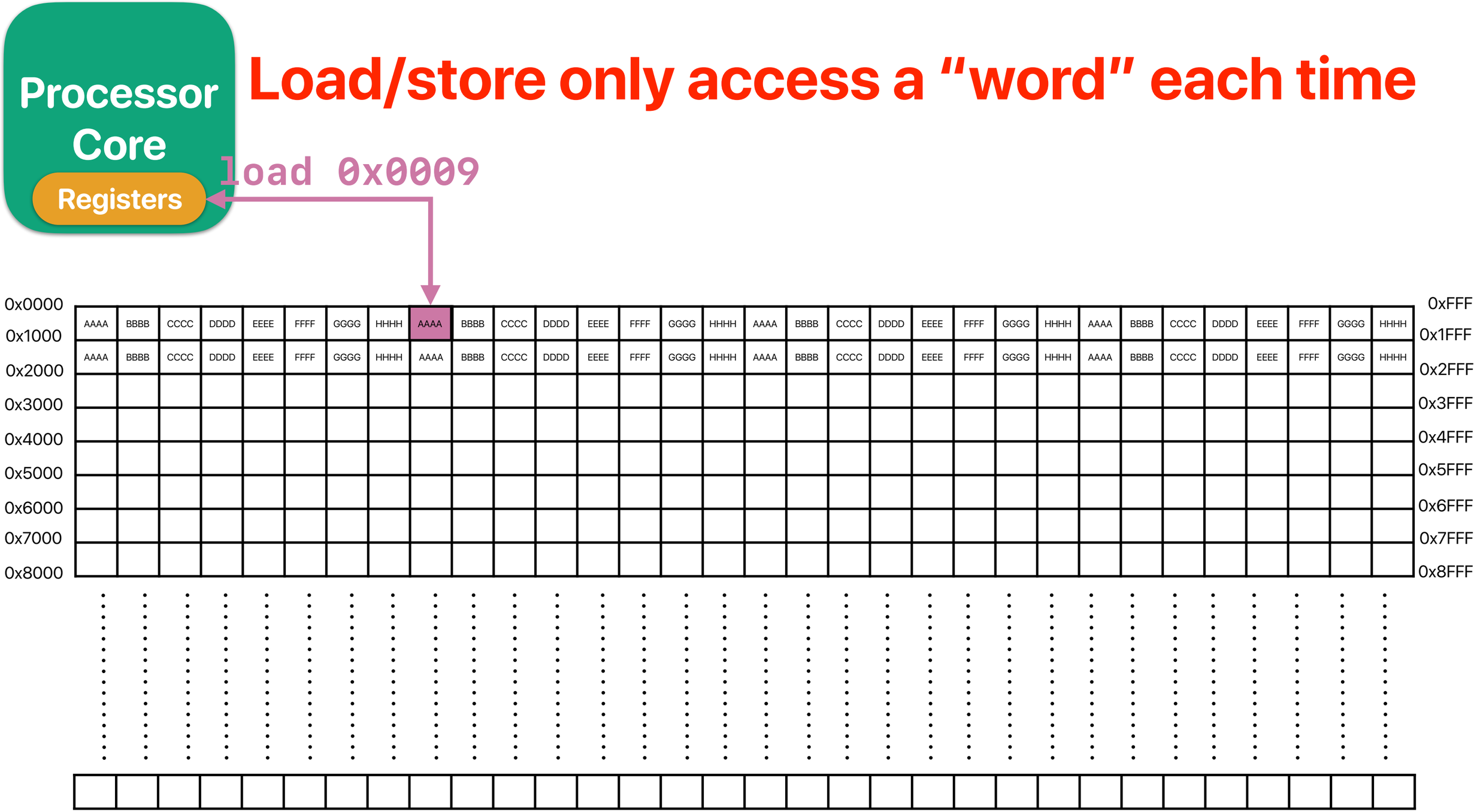
Locality

- 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
- 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

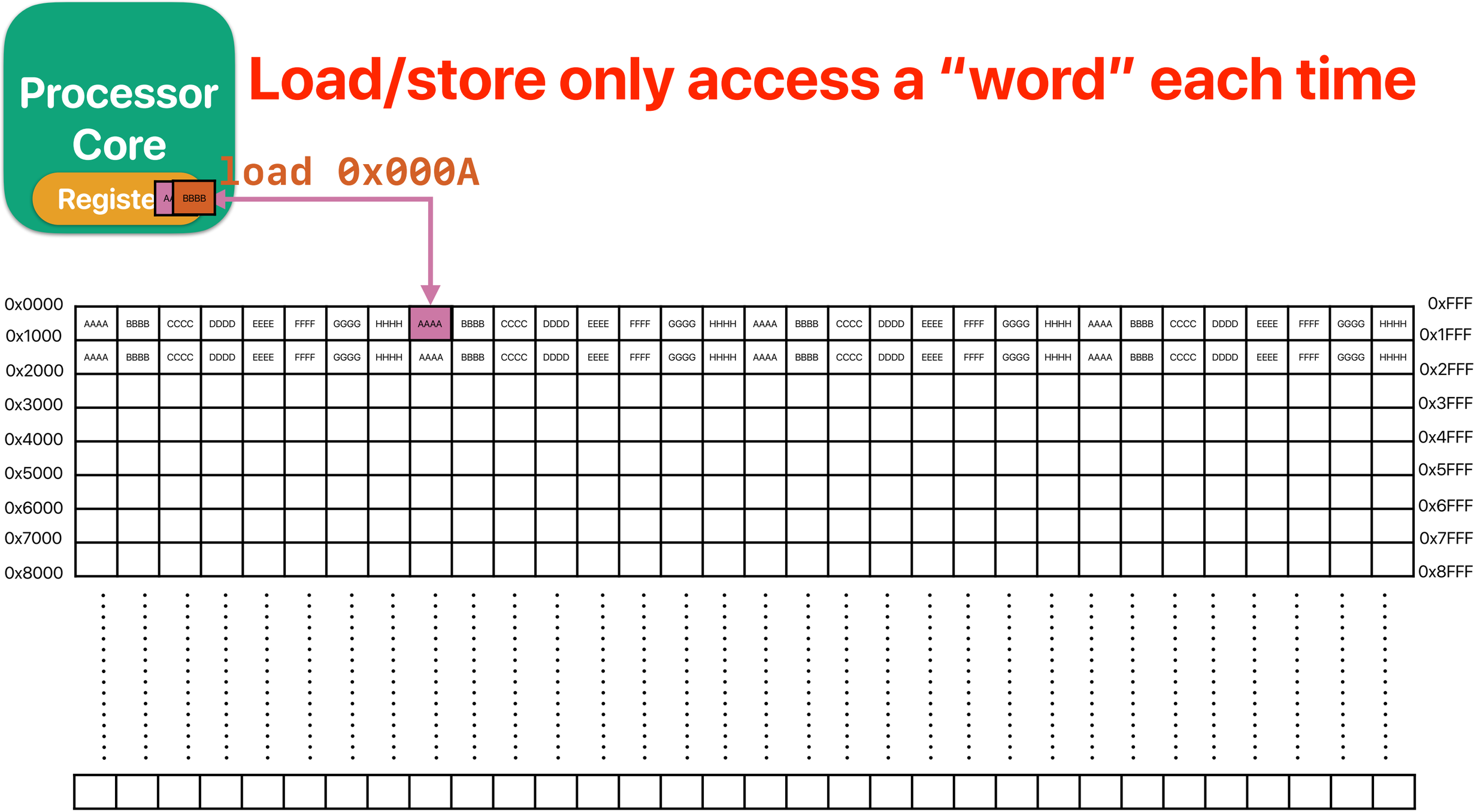
Most of time, your program is just visiting a very small amount of data/instructions within a given window

Architecting the Cache

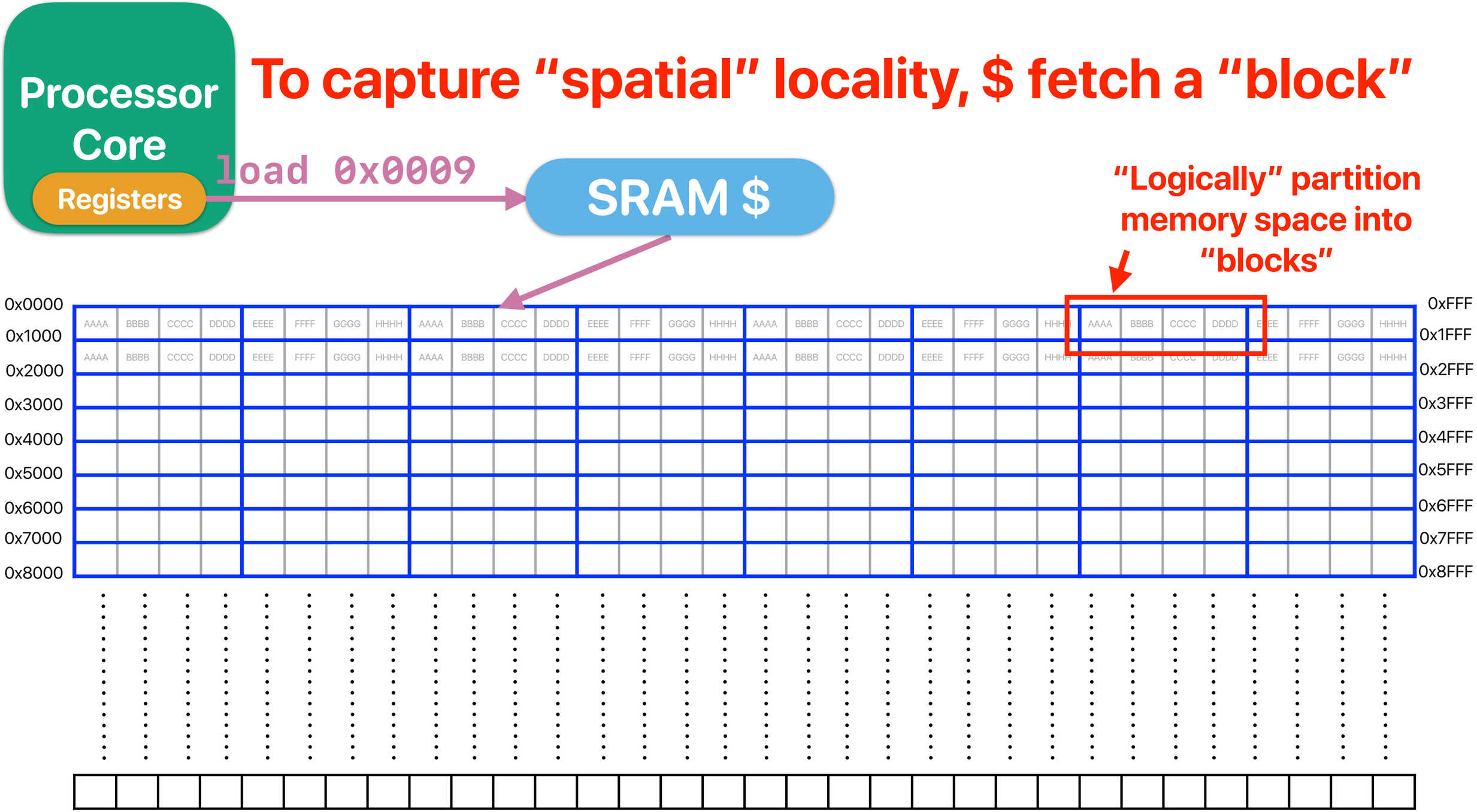
Load/store only access a “word” each time



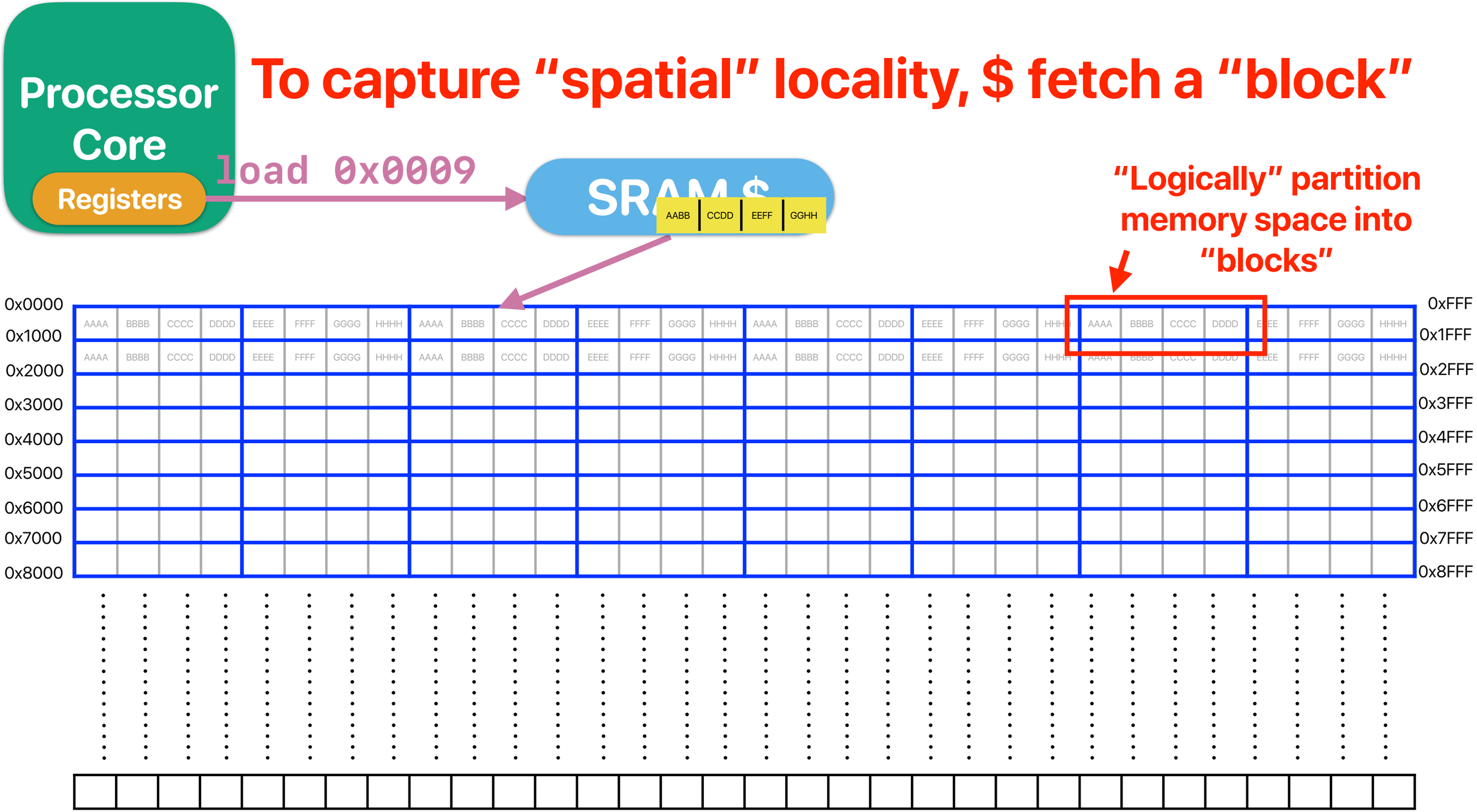
Load/store only access a “word” each time



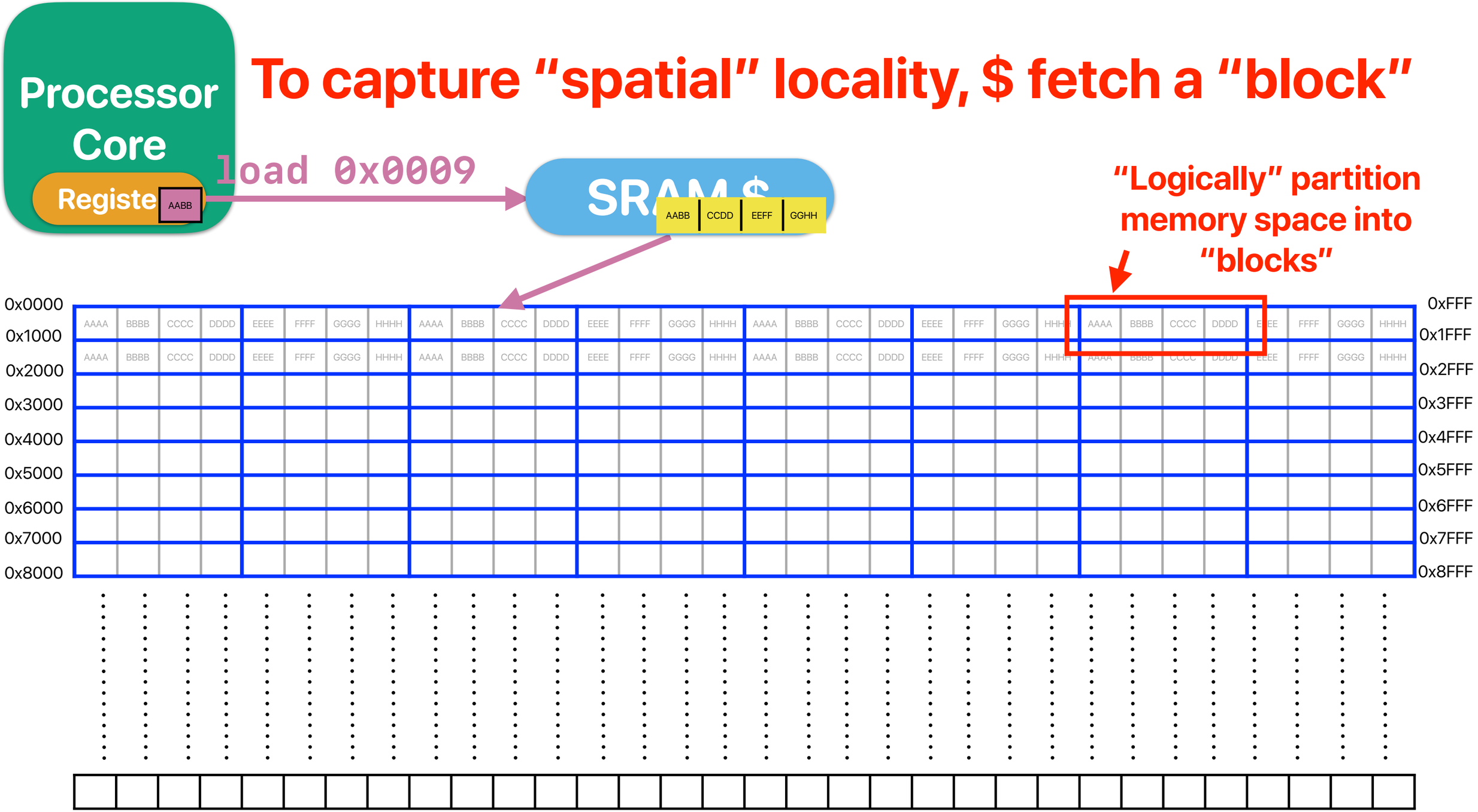
To capture "spatial" locality, $ fetch a "block"



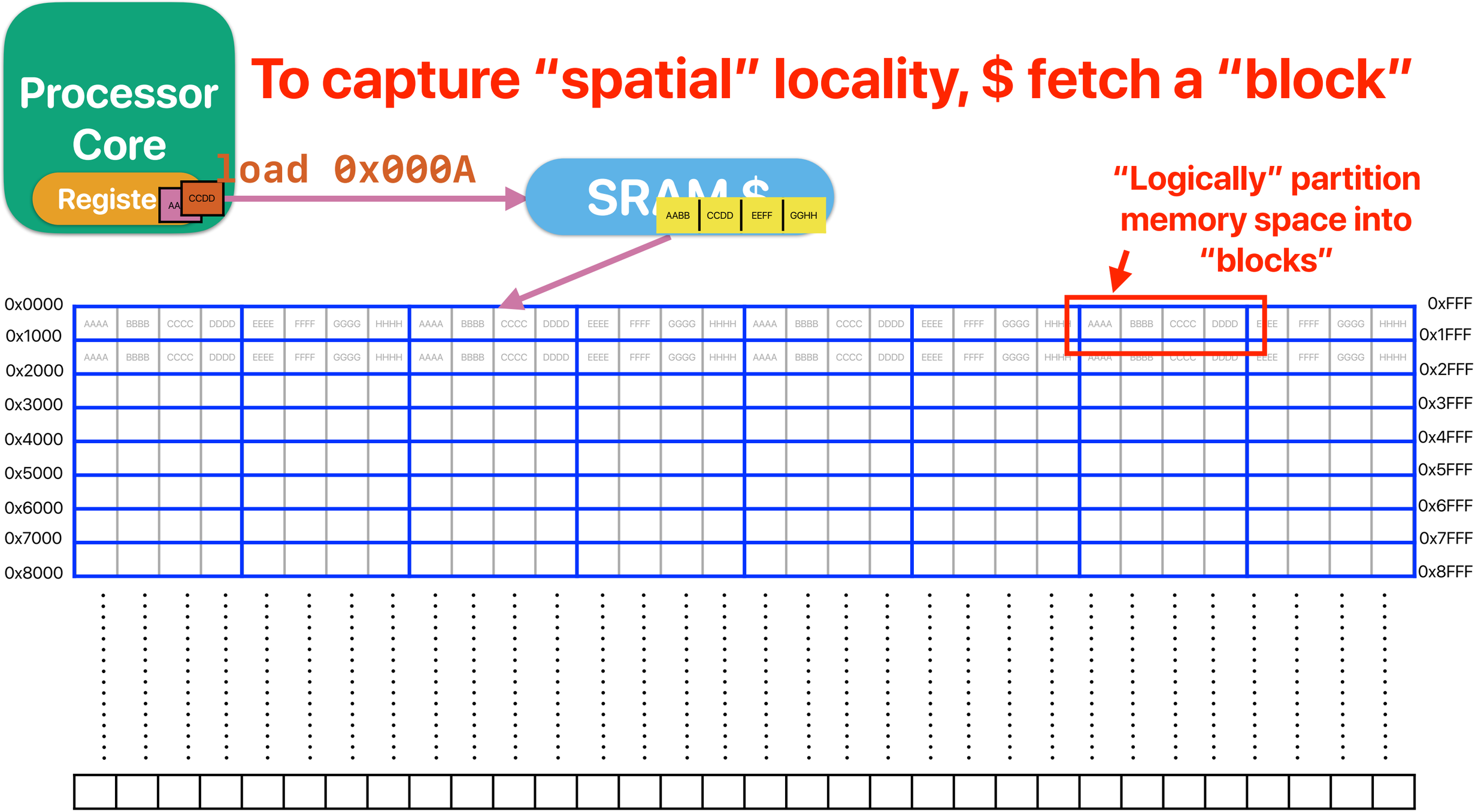
To capture "spatial" locality, $ fetch a "block"



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



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



How to tell w

Processor Core

Registers

GG

block offset

tag

load 0x000A

load 0x404A

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	AABBCCDDEEGGFFHH
1	1	0x001	IIJJKKLLMMNNOOPP
1	0	0xF07	QQRRSSTTUUVVWWXX
0	1	0x100	YYZZAABBCCDDEEFF
1	1	0x310	AABBCCDDEEGGFFHH
1	1	0x450	IIJJKKLLMMNNOOPP
0	1	0x006	QQRRSSTTUUVVWWXX
0	1	0x537	YYZZAABBCCDDEEFF
1	1	0x266	IIJJKKLLMMNNOOPP
1	1	0x307	QQRRSSTTUUVVWWXX
0	1	0x265	YYZZAABBCCDDEEFF
0	1	0x80A	AABBCCDDEEGGFFHH
1	1	0x620	IIJJKKLLMMNNOOPP
1	1	0x630	QQRRSSTTUUVVWWXX
1	0	0x705	QQRRSSTTUUVVWWXX
0	1	0x216	YYZZAABBCCDDEEFF

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 0x000A

load 0x404A

0x40 not found,
go to lower-level memory

V	D	tag	data (0123456789ABCDEF)
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF
1	1	0x29	IIJJKKLLMMNNOOPP
1	1	0xDE	QQRRSSTTUUVVWWXX
0	1	0xCB	YYZZAABBCCDDEEFF
0	1	0x8A	AABBCCDDEEGGFFHH
1	1	0x60	IIJJKKLLMMNNOOPP
1	1	0x70	QQRRSSTTUUVVWWXX
1	0	0x10	QQRRSSTTUUVVWWXX
0	1	0x11	YYZZAABBCCDDEEFF

The biggest issue with hash is — Collision!

Way-associative cache

memory address: 0x0 8 2 4

set block

tag index offset

memory address: 0b0000100000100100

V	D	tag	data
1	1	0x29	IIJJKKLLMMNNOOPP
1	1	0xDE	QQRRSSTTUUVVWWXX
1	0	0x10	YYZZAABBCCDDEEFF
0	1	0x8A	AABBCCDDEEGGFFHH
1	1	0x60	IIJJKKLLMMNNOOPP
1	1	0x70	QQRRSSTTUUVVWWXX
0	1	0x10	QQRRSSTTUUVVWWXX
0	1	0x11	YYZZAABBCCDDEEFF

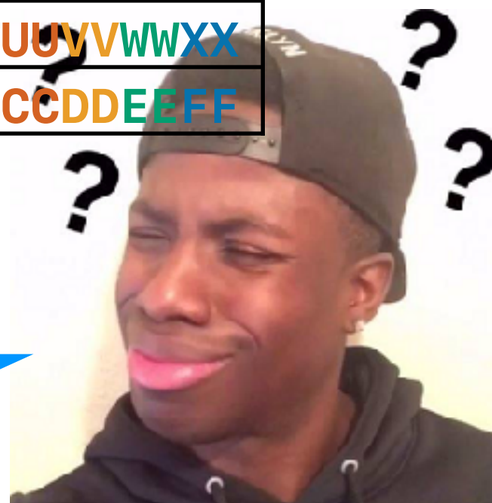
V	D	tag	data
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF

Set

0x1 0

=? =?

hit? hit?



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 32 bits for 32-bit machine
- (address / block_size) % S = set index

AMD Phenom II

- L1 data (D-L1) cache configuration of AMD Phenom II
  - Size 64KB, 2-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 = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits

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 = ABS

32KB = 8 * 64 * S

S = 64

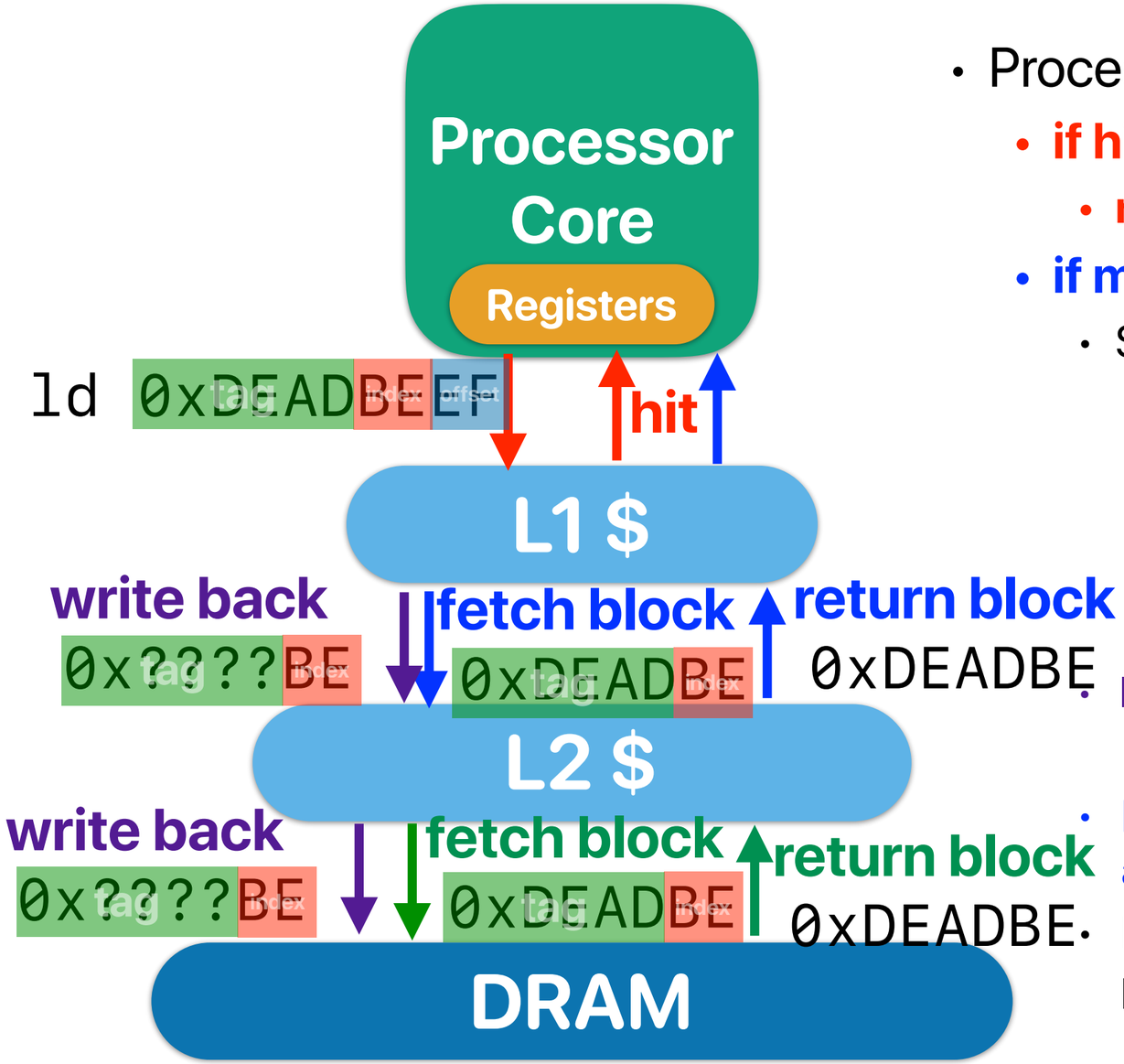
offset = lg(64) = 6 bits

index = lg(64) = 6 bits

tag = 64 - lg(64) - lg(64) = 52 bits

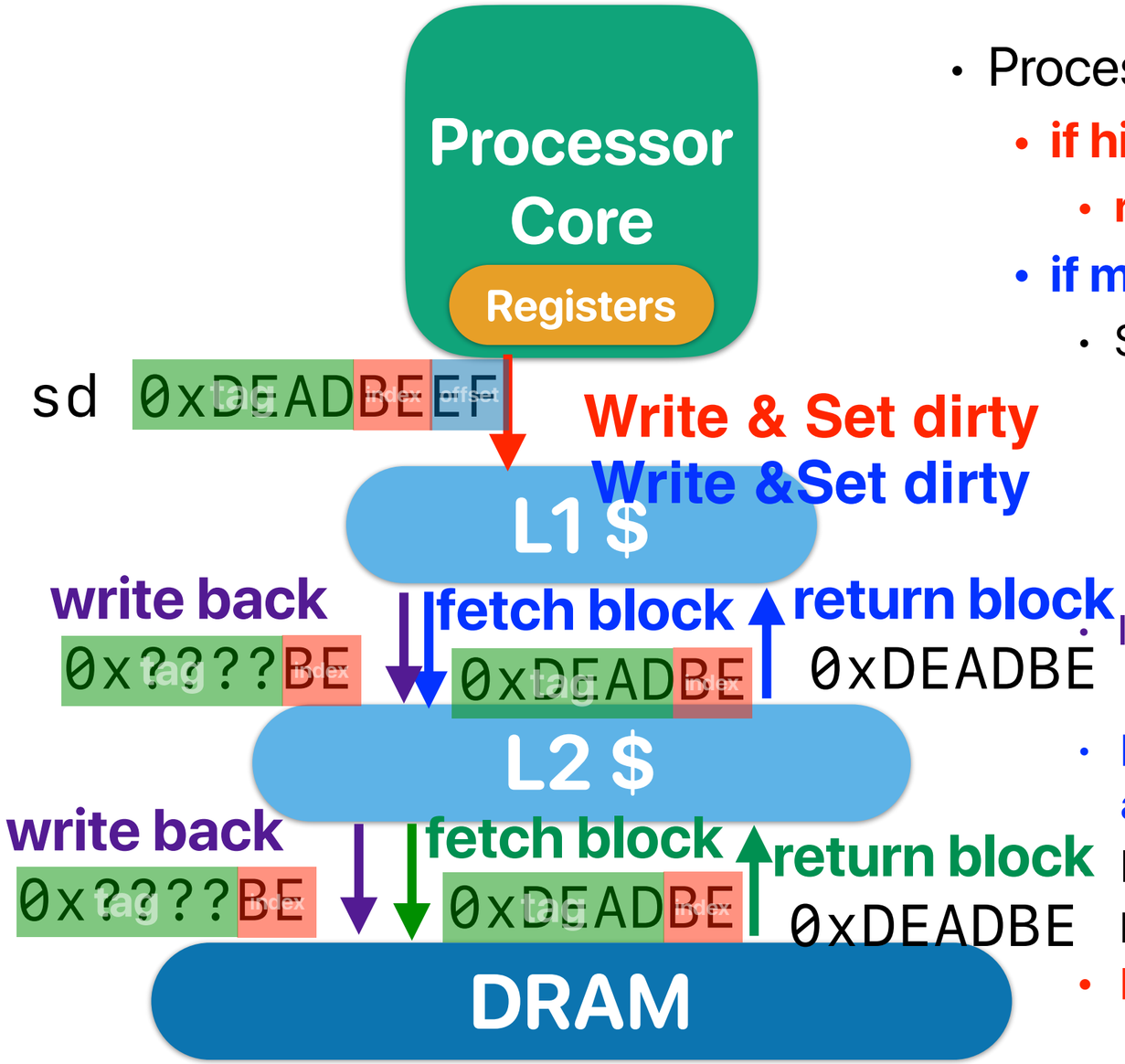
Put everything all together: How cache interacts with CPU

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

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**

Simulate the cache!

Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
    - C = A B S
    - S=256/(16*1) = 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 48 - (4 + 4) = 40 bits
    - For example: `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Simulate a direct-mapped cache

	V	D	Tag	Data
0	1	0	0b10	
1	1	0	0b10	
2	0	0		
3	0	0		
4	0	0		
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		

tag	index		
0b10	0000	0000	miss
0b10	0000	1000	hit!
0b10	0001	0000	miss
0b10	0001	0100	hit!
0b11	0001	0000	miss
0b10	0000	0000	hit!
0b10	0000	1000	hit!
0b10	0001	0000	miss
0b10	0001	0100	hit!

Announcement

- Make up lecture **7pm next Monday (10/28) @ WCH 143** — will be midterm review
- Check
  - website for slides/schedules
  - iLearn for quizs/assignments/podcasts
  - piazza for discussions