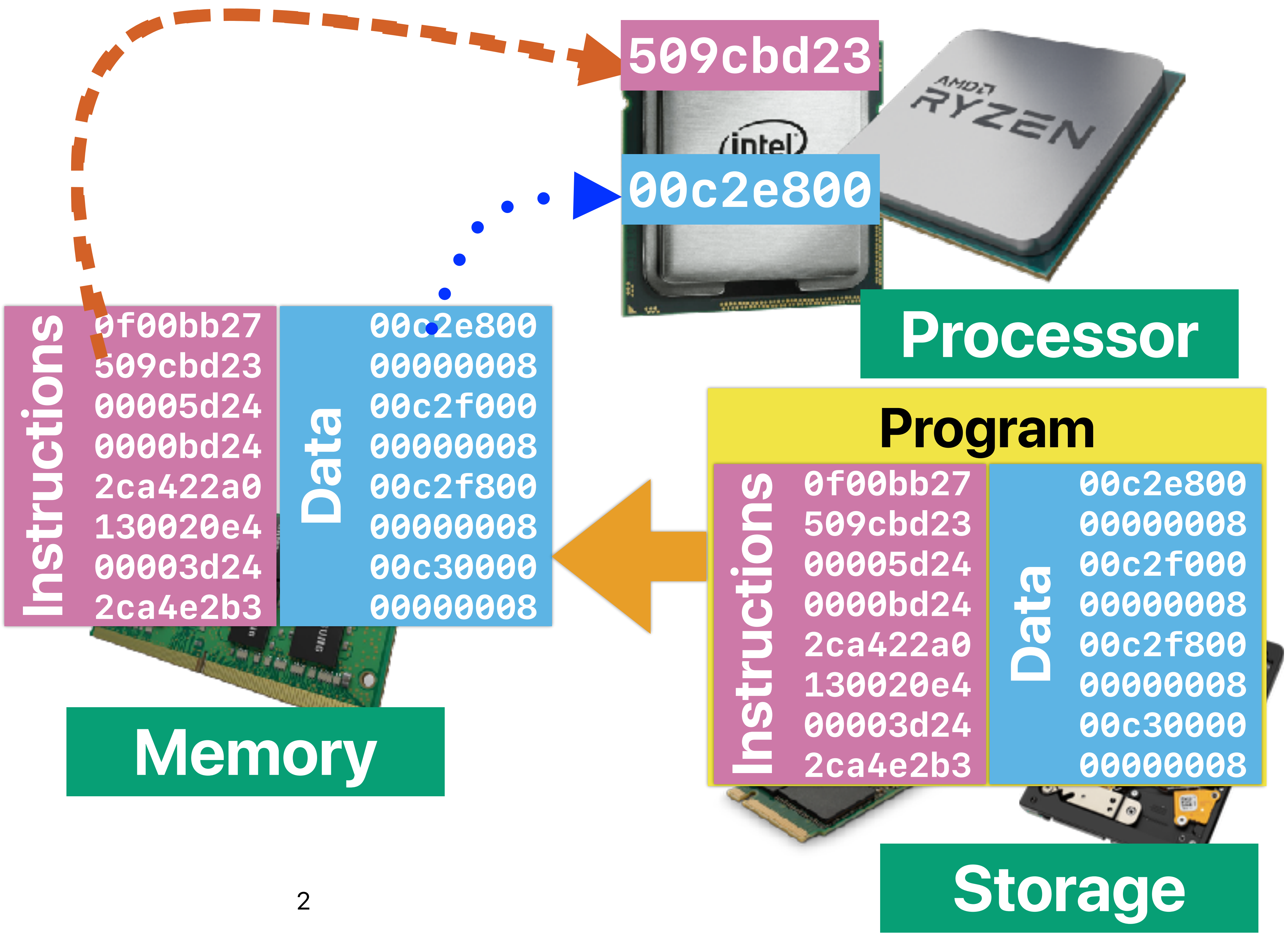
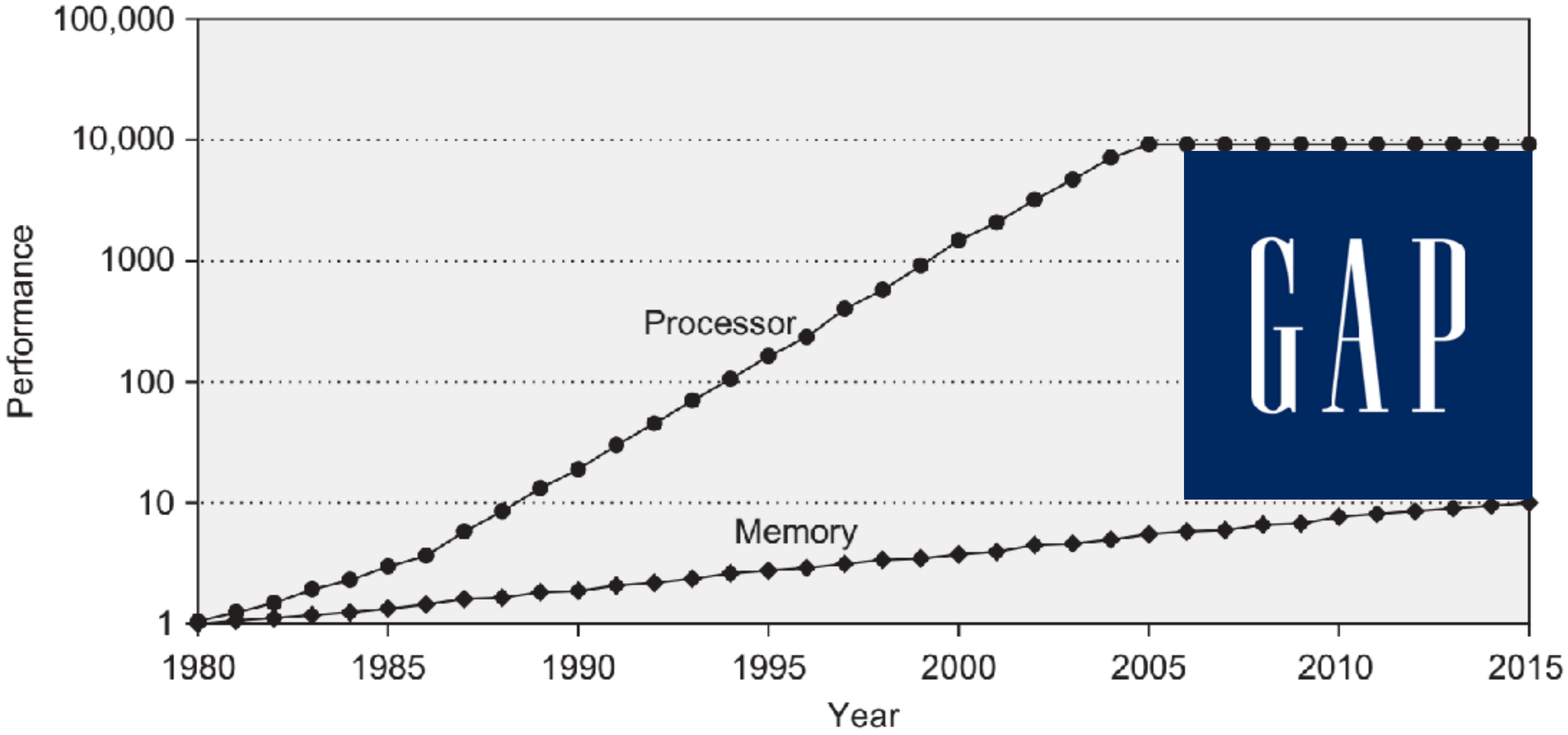
Memory Hierarchy (2)

Hung-Wei Tseng

von Neumman Architecture



Recap: Performance gap between Processor/Memory

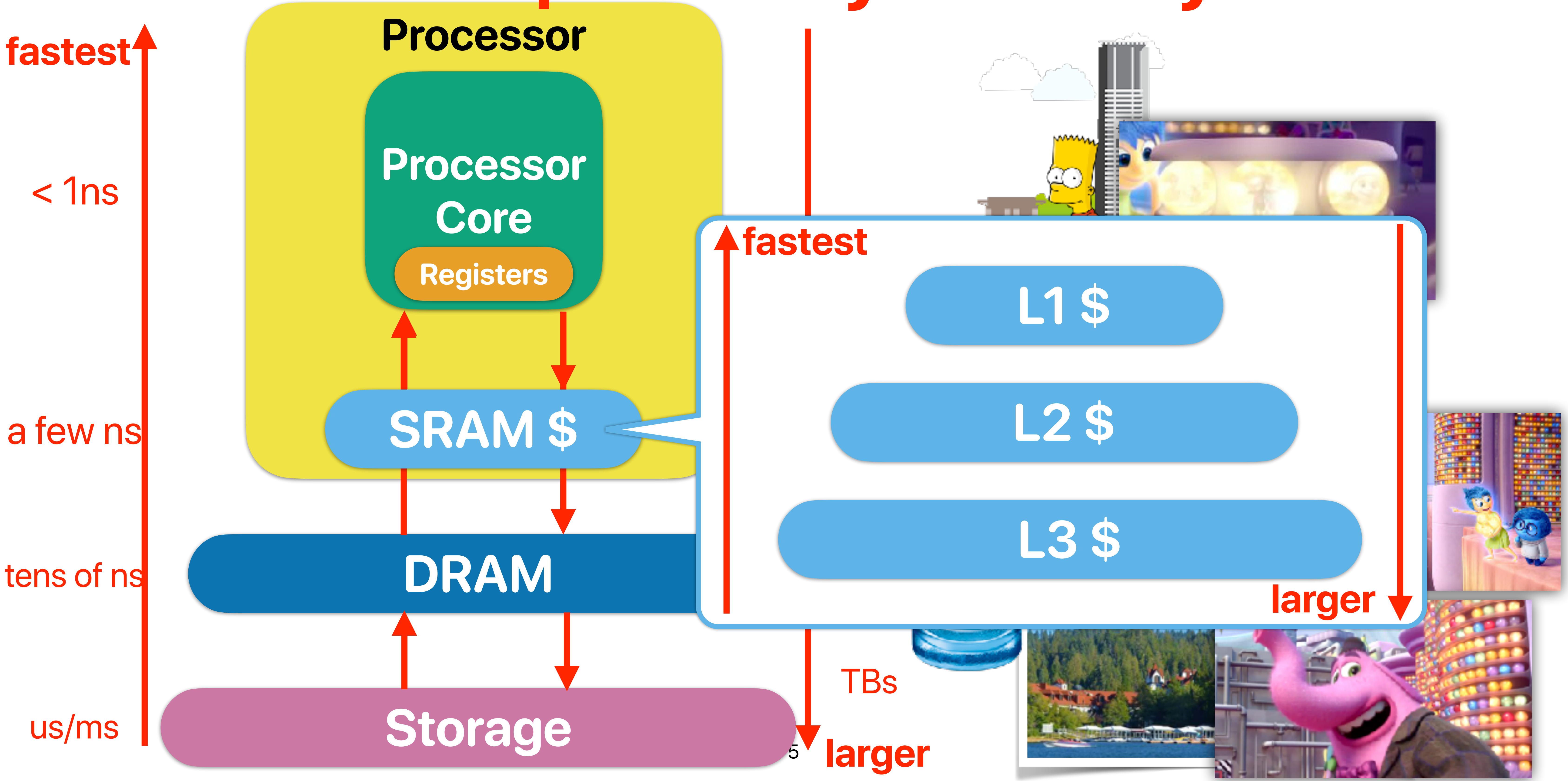


Recap: Alternatives?

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Fast, but expensive $$$

Recap: Memory Hierarchy



Recap: 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

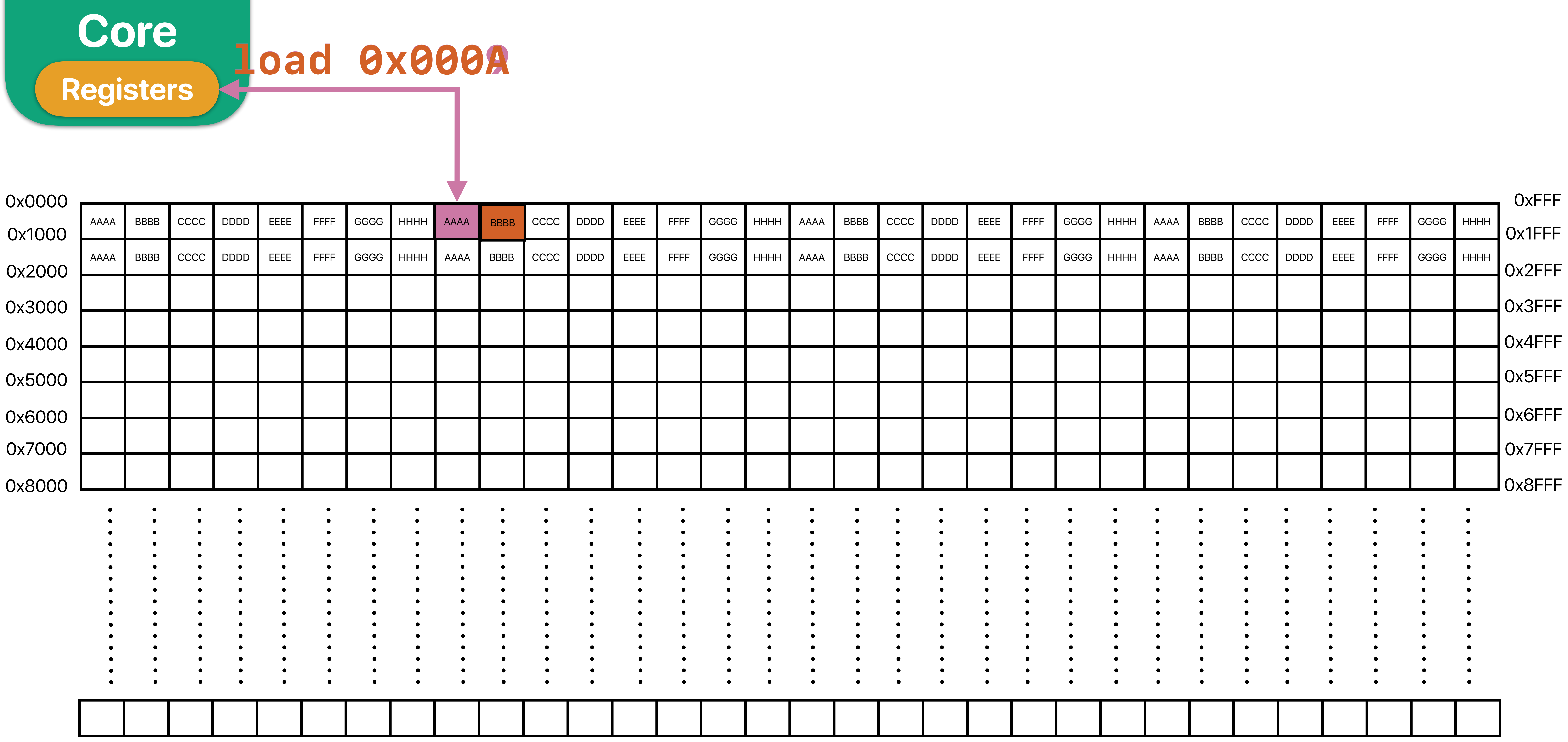
Most of time, your program is just visiting a very small amount of data/instructions within a given window

Outline

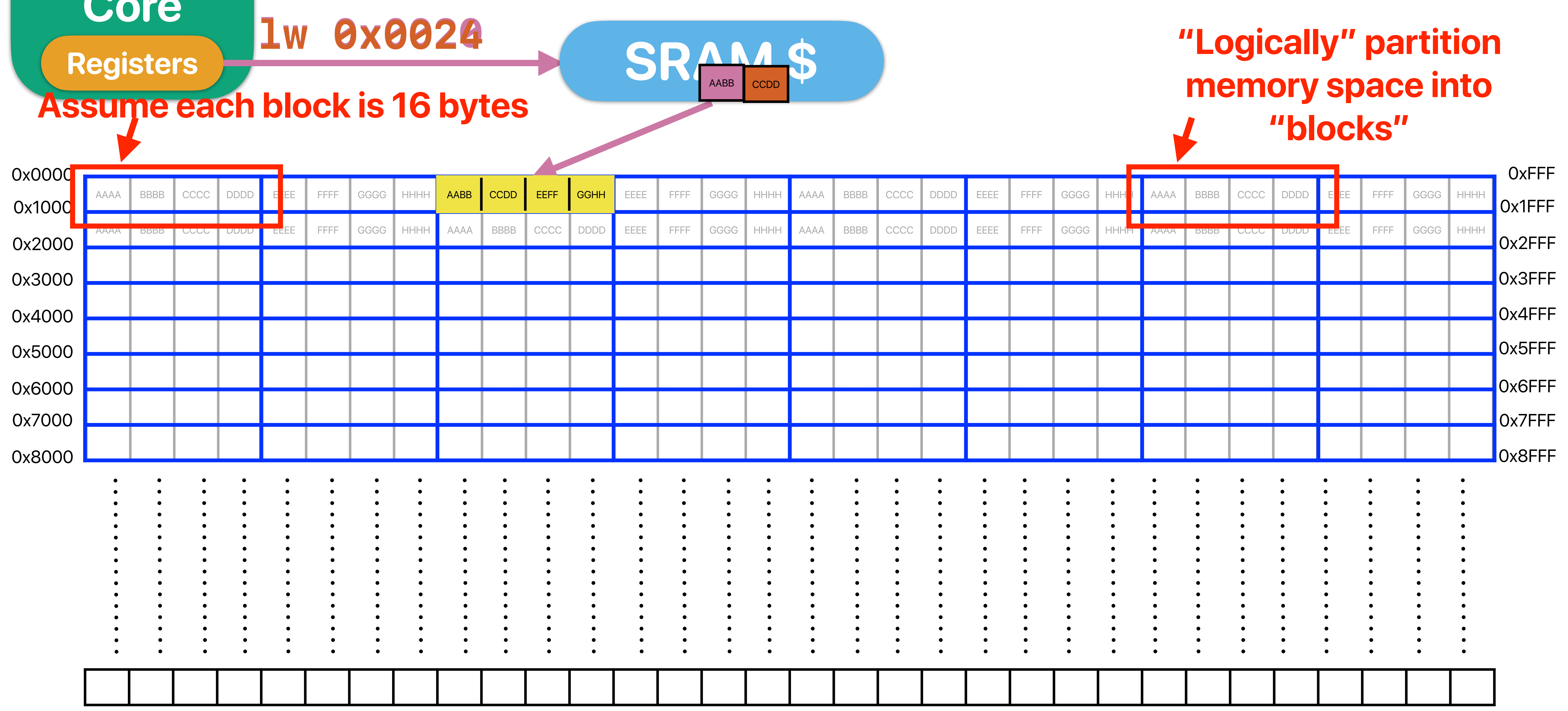
- Architecting the cache

Architecting the Cache

Load/store only access a “word” each time



To capture “spatial” locality, $ fetch a “block”



Processor Core

Registers

How to tell who is there?

0x0000 0x0001 0x0002 0x0003 0x0004 0x0005 0x0006 0x0007 0x0008 0x0009 0x000A 0x000B 0x000C 0x000D 0x000E 0x000F

tag	0123456789ABCDEF
0x000	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:

Processor Core

Registers

How to tell w

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:

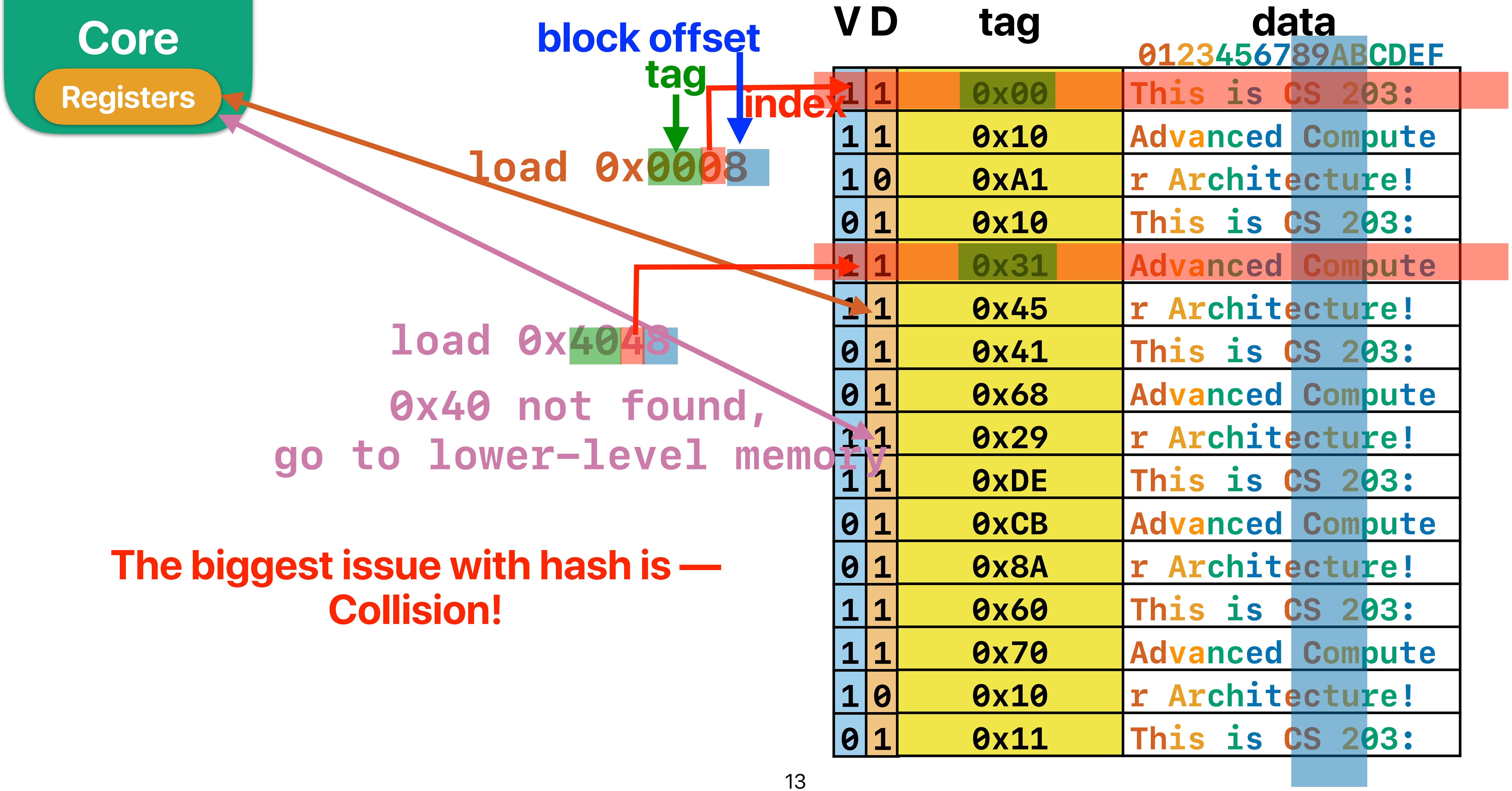
The complexity of search the matching tag—

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

Can we search things faster?

—hash table! $O(1)$

Hash-like structure — direct-mapped cache



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
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

tag | set index | block offset

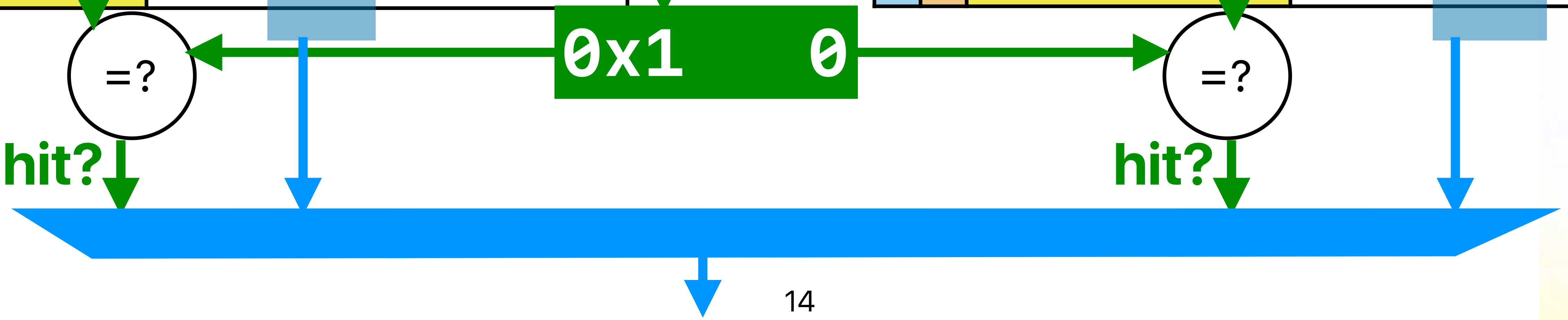
memory address: 0b0000100000 0100 100

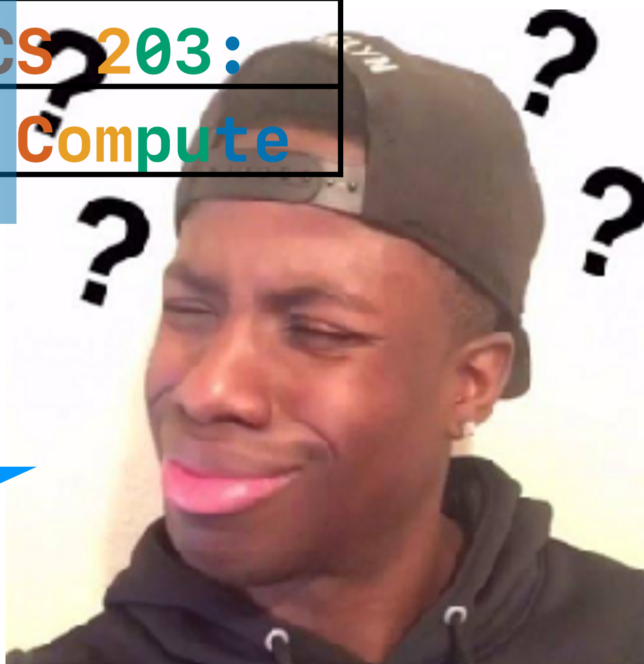
(tag: 0000100000, set index: 010, block offset: 0100)

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:

Set

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





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:	0b000010000	010	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

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

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

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. $\boxed{\text{Tag is 49 bits}}$
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

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. $\boxed{\text{Index is 6 bits}}$
C. Offset is 6 bits
D. The cache has 128 sets

$C = ABS$

$32KB = 8 * 64 * S$

$S = 64$

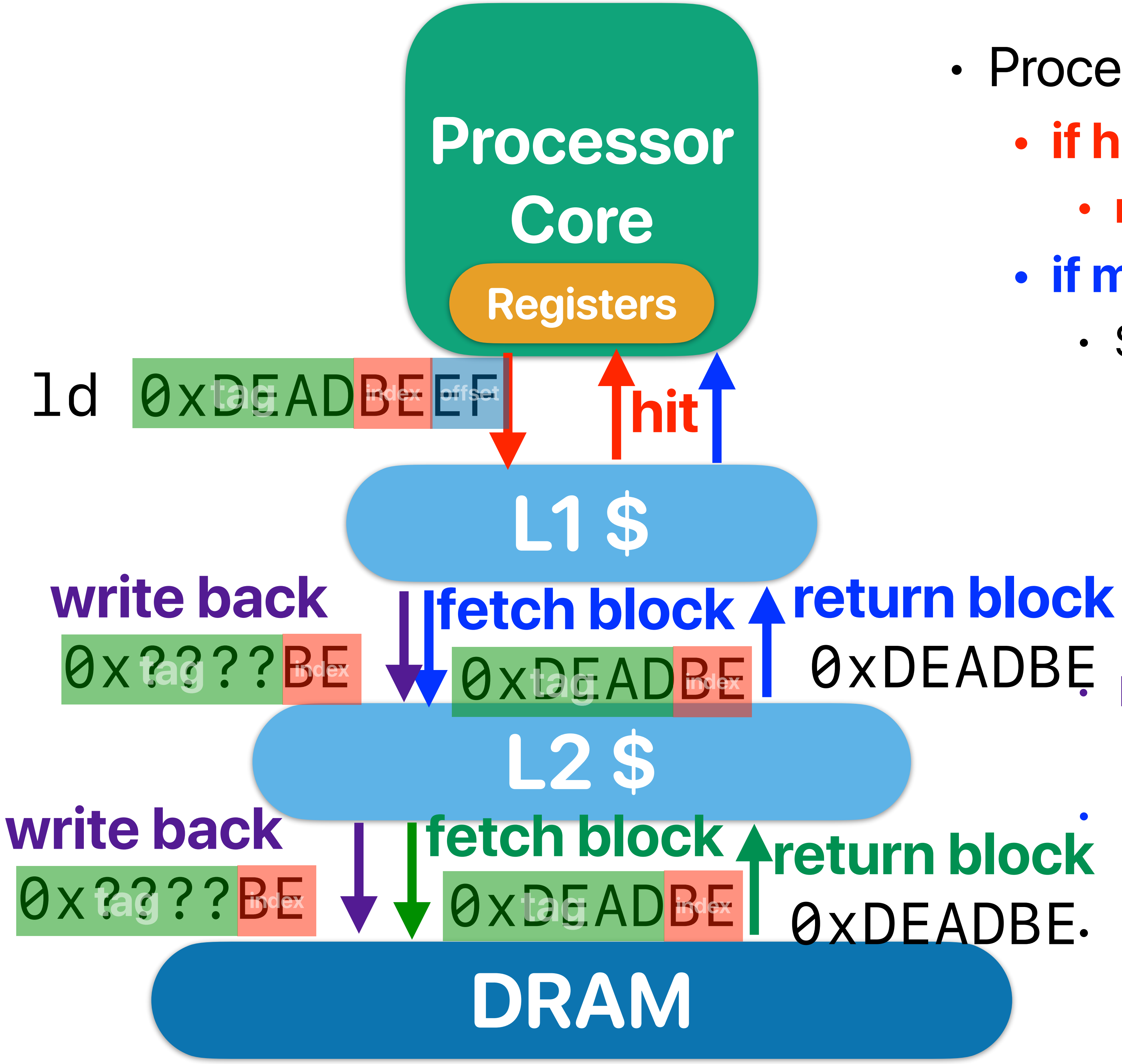
offset = lg(64) = 6 bits

index = lg(64) = 6 bits

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

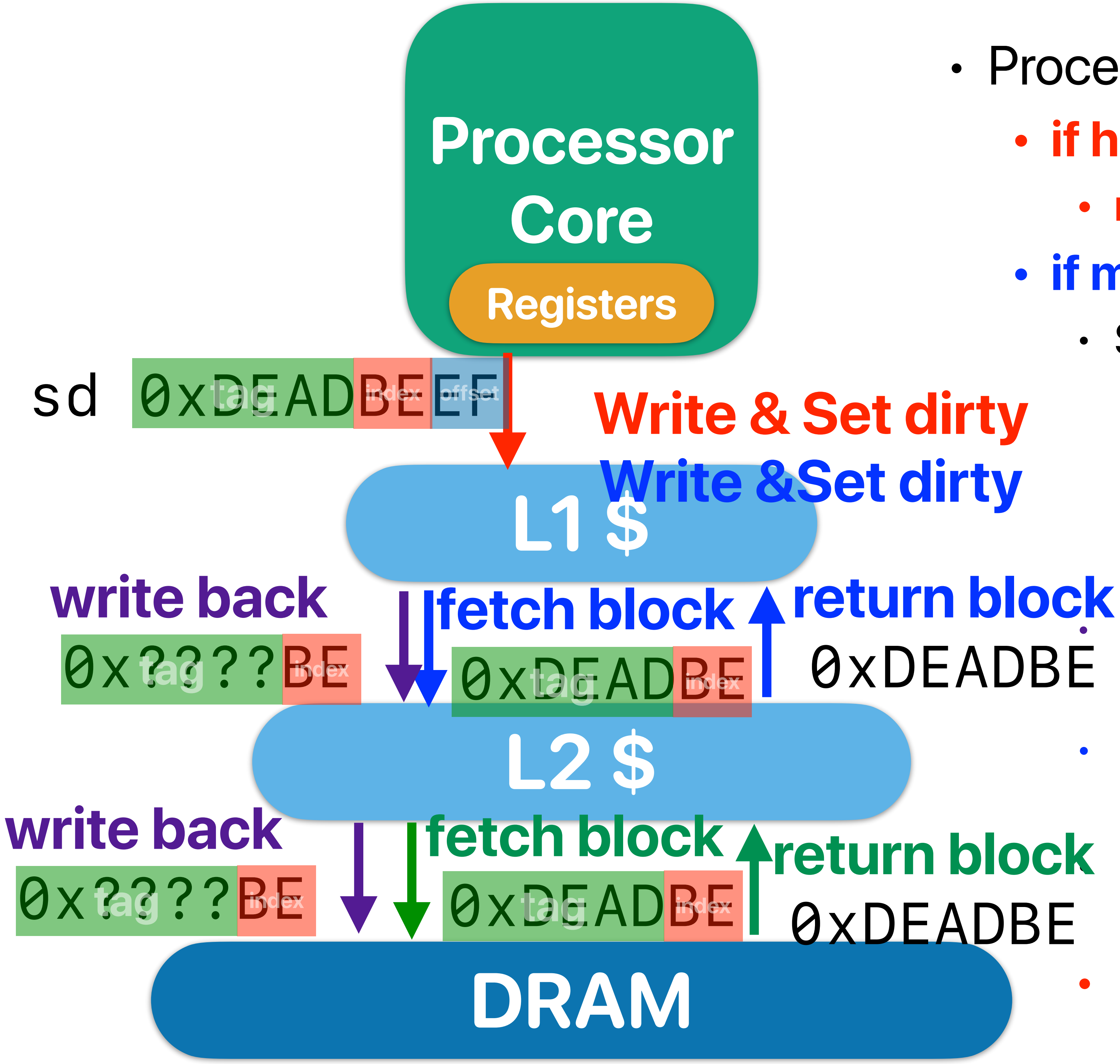
Put everything all together: How cache interacts with CPU

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

Simulate a direct-mapped cache

	V	D	Tag	Data
0	1	0	0b10	r Architecture!
1	1	0	0b10	This is CS 203:
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!

Simulate a 2-way cache

- Consider a 2-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*2) = 8
    - 8 = 2^3 : 3 bits are used for the index
    - 16 = 2^4 : 4 bits are used for the byte offset
    - The tag is 32 - (3 + 4) = 25 bits
    - For example: $0b\underbrace{\texttt{1000 0000 0000 0000 0000 0000 0}}_{\text{tag}}\underbrace{\texttt{001}}_{\text{index}}\ \underbrace{\texttt{0000}}_{\text{offset}}$

Simulate a 2-way cache

	V	D	Tag	Data	V	D	Tag	Data
0	1	0	0b10	r Architecture!	0	0		
1	1	0	0b10	This is CS 203:	1	0	0b11	Advanced Compute
2	0	0			0	0		
3	0	0			0	0		
4	0	0			0	0		
5	0	0			0	0		
6	0	0			0	0		
7	0	0			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	hit
0b10	0001	0100	hit!

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

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

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

A. 6.25%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

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

A. 6.25%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

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

AMD Phenom II

100% miss rate!

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

	address in hex	address in binary (tag, index, offset)	tag	index	hit? miss?
load a[0]	0x20000	0b10 0000 0000 0000 0000	0x4	0	miss
load b[0]	0x30000	0b11 0000 0000 0000 0000	0x6	0	miss
store c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	miss, evict 0x4
load a[1]	0x20004	0b10 0000 0000 0000 0100	0x4	0	miss, evict 0x6
load b[1]	0x30004	0b11 0000 0000 0000 0100	0x6	0	miss, evict 0x2
store c[1]	0x10004	0b01 0000 0000 0000 0100	0x2	0	miss, evict 0x4
⋮	⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0b10 0000 0000 0011 1100	0x4	0	miss, evict 0x6
load b[15]	0x3003C	0b11 0000 0000 0011 1100	0x6	0	miss, evict 0x2
store c[15]	0x1003C	0b01 0000 0000 0011 1100	0x2	0	miss, evict 0x4
load a[16]	0x20040	0b10 0000 0000 0100 0000	0x4	1	miss
load b[16]	0x30040	0b11 0000 0000 0100 0000	0x6	1	miss
store c[16]	0x10040	0b01 0000 0000 0100 0000	0x2	1	miss, evict 0x4

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

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

A. 6.25%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

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

Announcement

- Assignment #1 due tonight!!!
  - Assignments SHOULD BE done/submitted individually — if discussed with others, make sure their names on your submission
  - We will drop your least performing assignment as well
- Reading Quiz #4 next Monday before the lecture
- Office Hours
  - Walk-in, no appointment is necessary
  - Hung-Wei/Prof. Usagi: MTu 2p-3p (WCH 406 or on Zoom)
  - Abenezer Wudenhe: WTh 3p-4p (Zoom only)