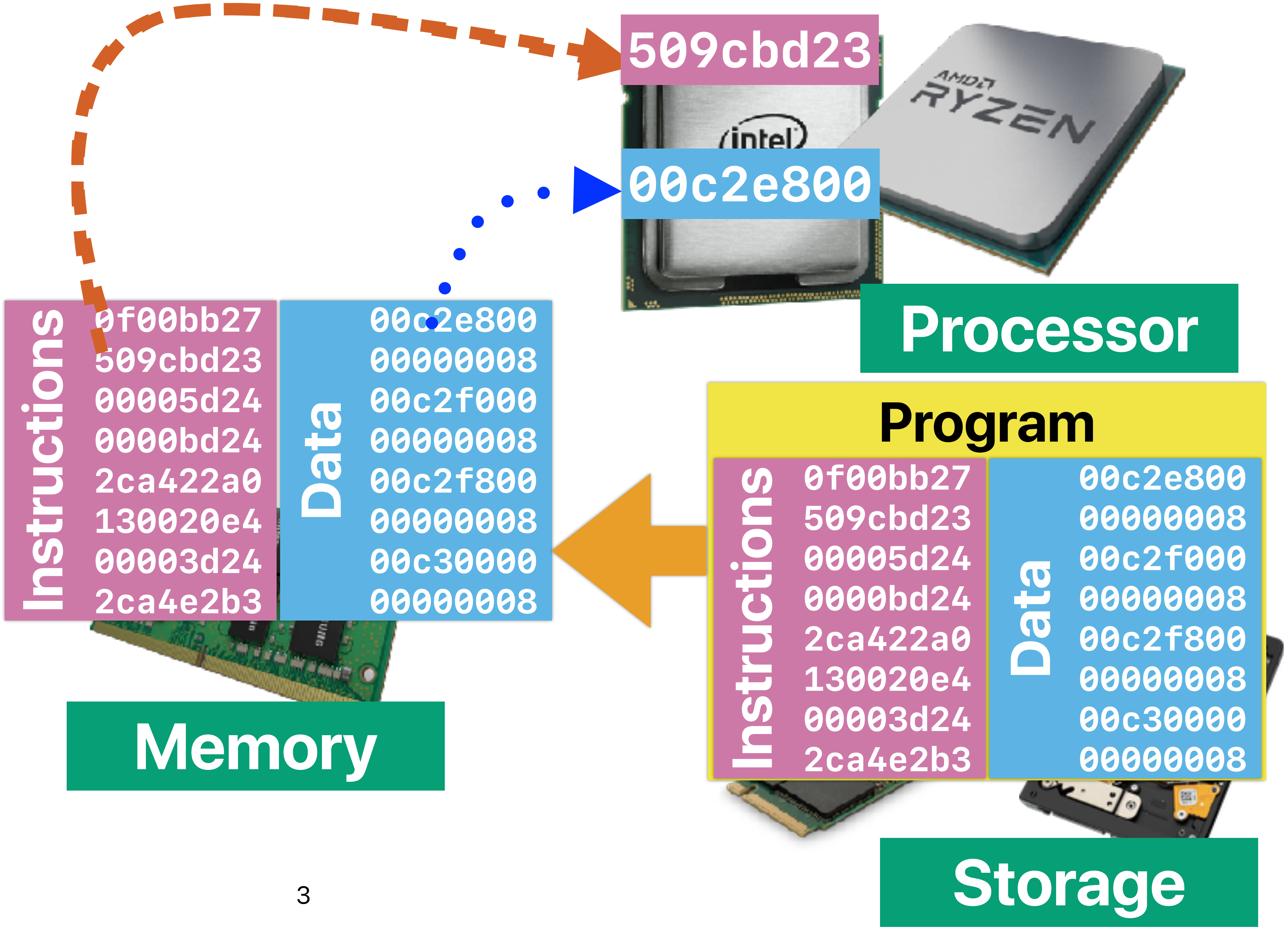
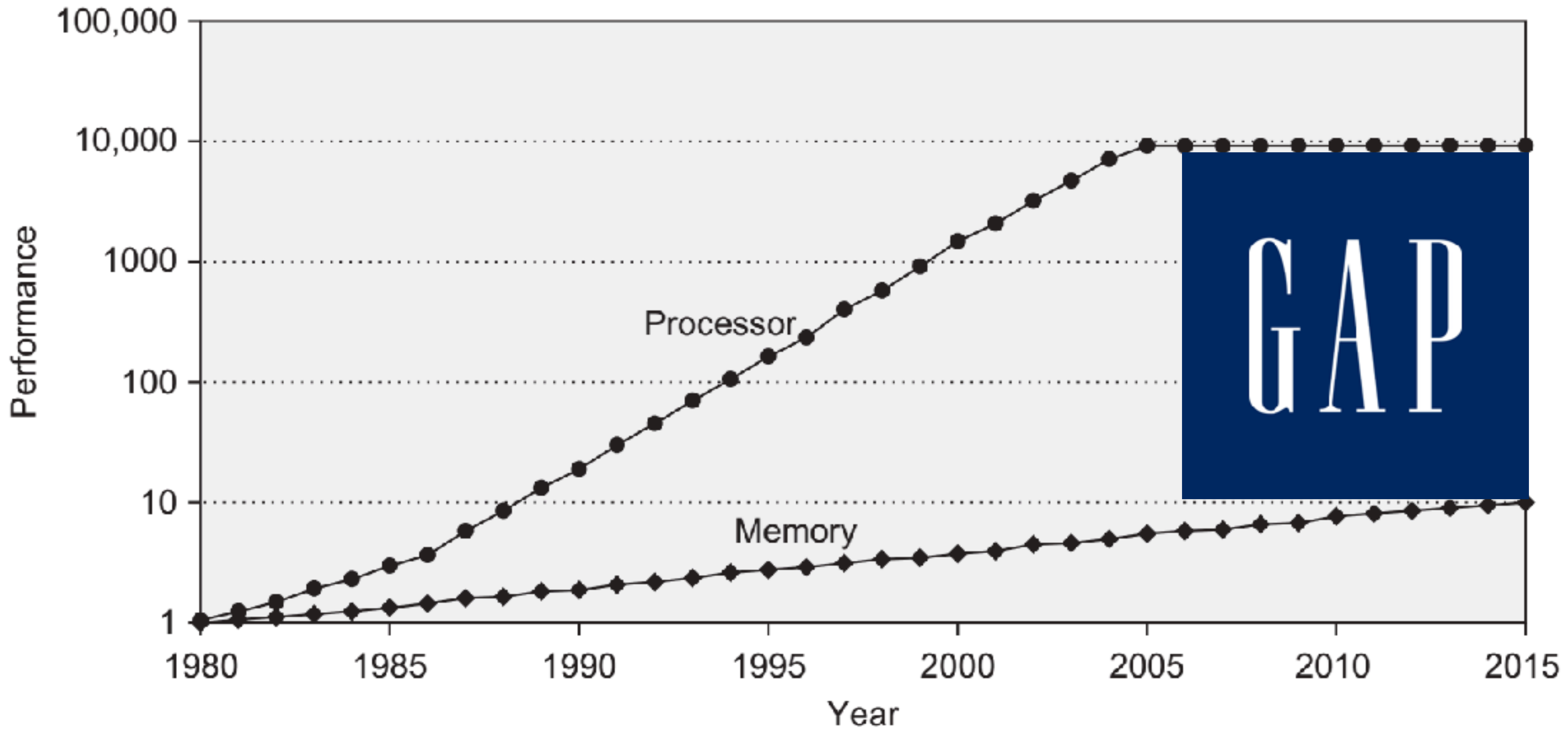
Memory Hierarchy Inside Out: (2) The A, B, C s of caches

Hung-Wei Tseng

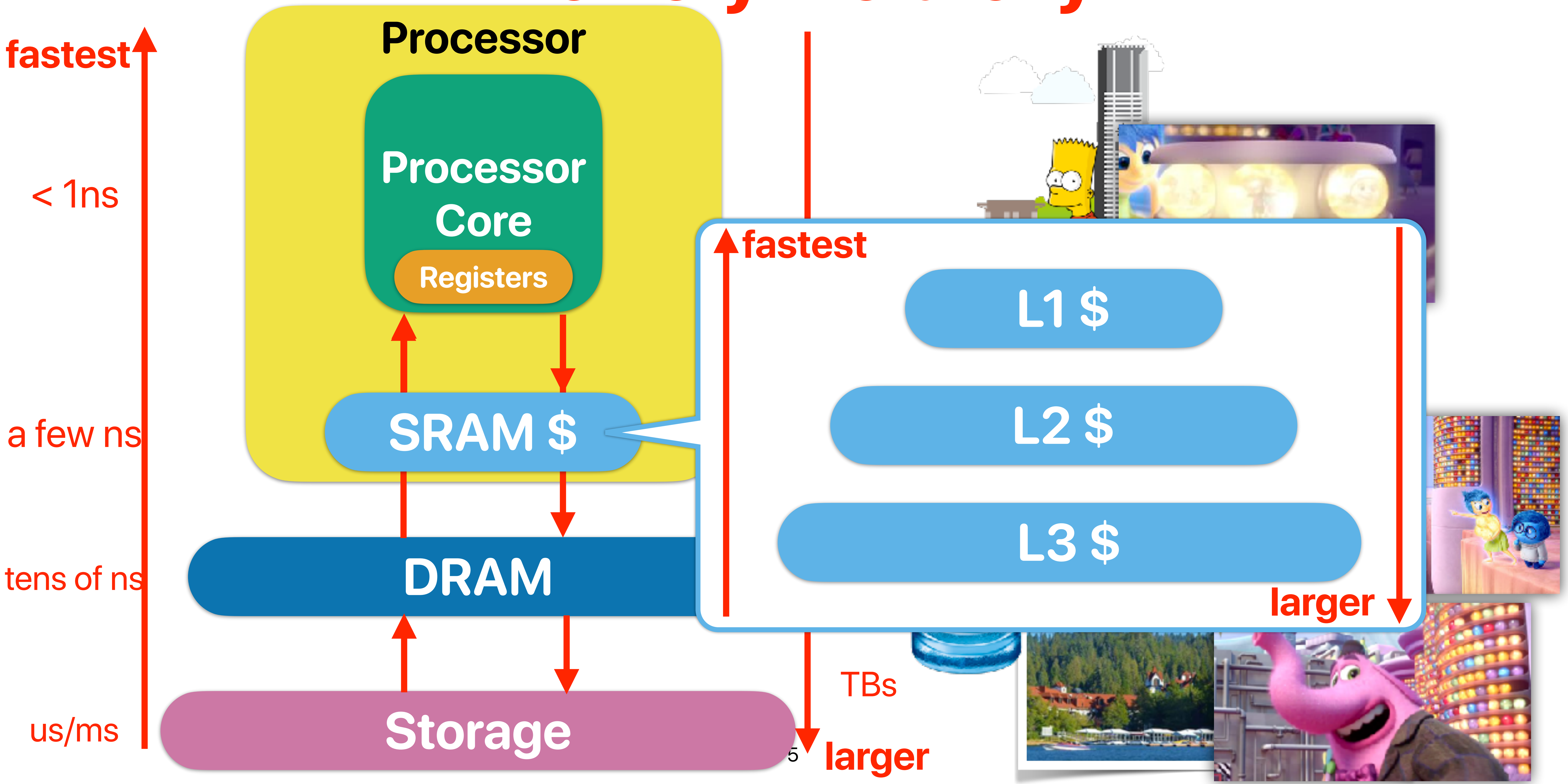
von Neumman Architecture



Recap: Performance gap between Processor/Memory



Memory Hierarchy



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;
}
```

keep going to the next instruction — spatial locality

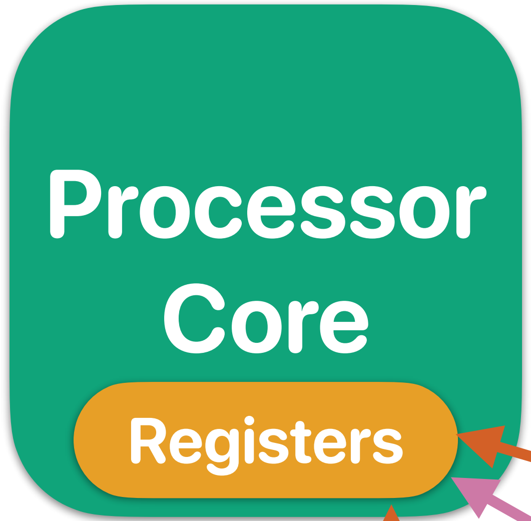
```
i = 0;
while(i < m) {
    result = 0;
    j = 0;
    while(j < n) {
        a = matrix[i][j];
        b = vector[j];
        temp = a*b;
        result = result + temp;
    }
    output[i] = result;
    i++;
}
```

repeat 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



How to tell w

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
```

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!

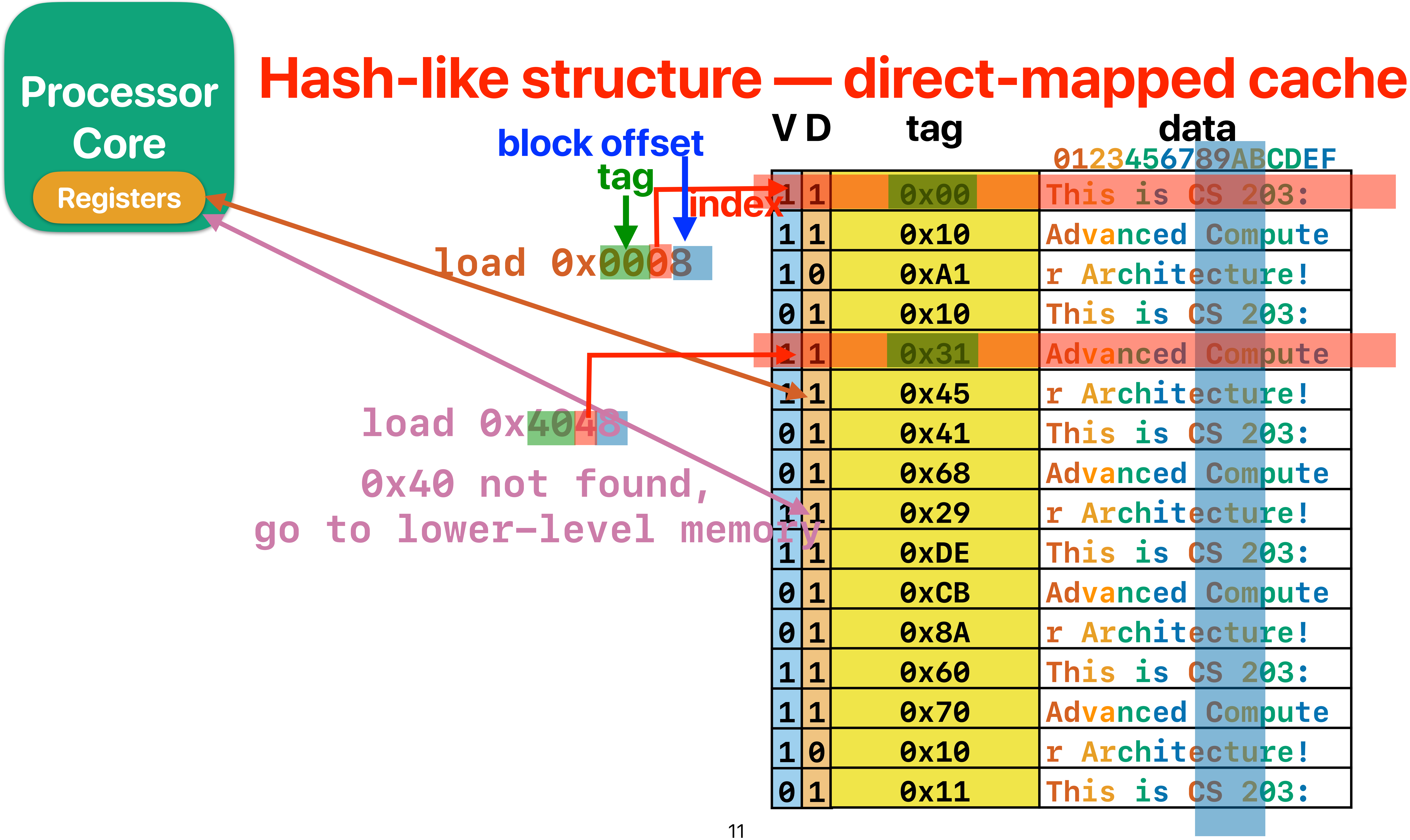
Can we search things faster?

—hash table! $O(1)$

Outline

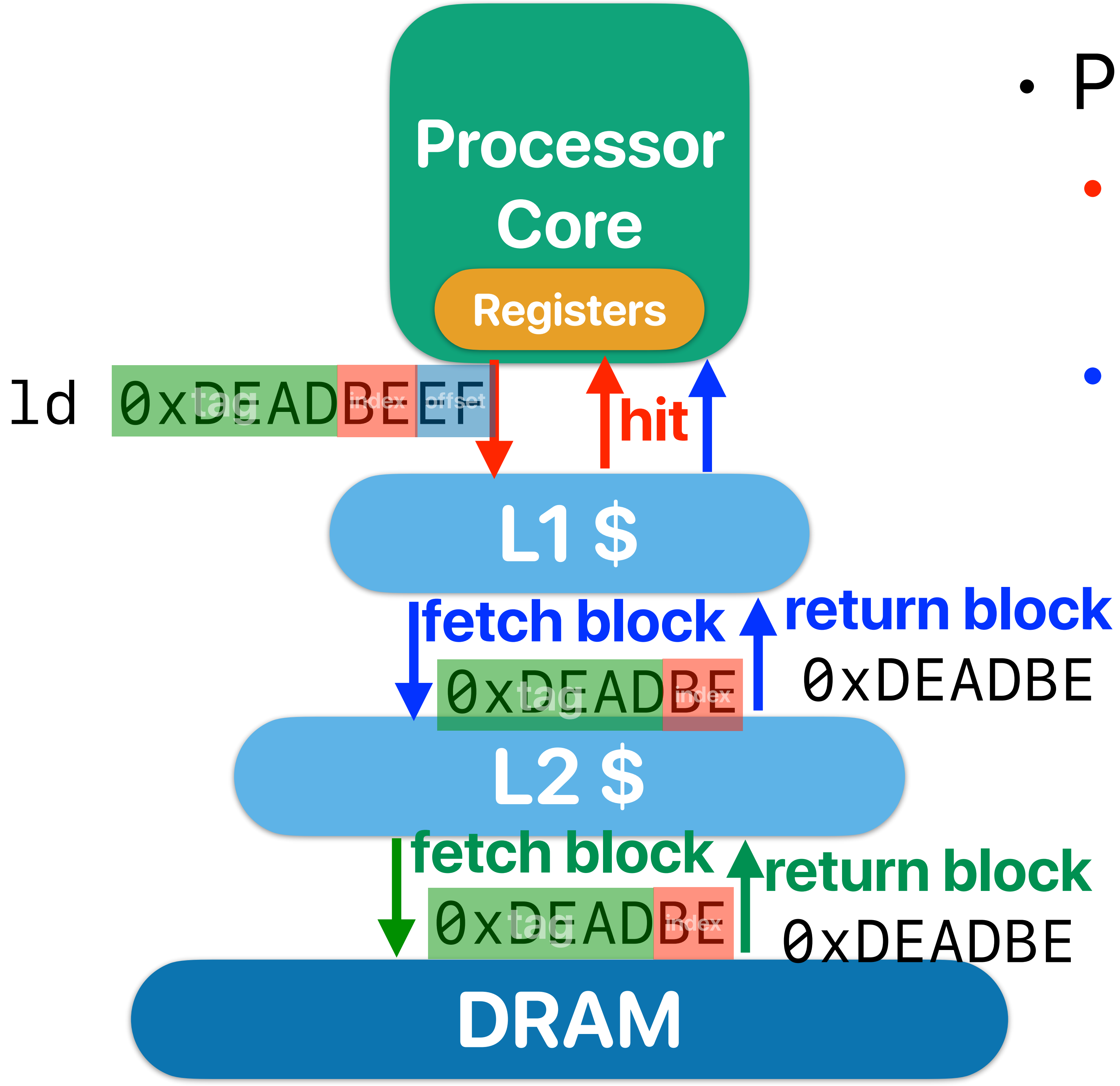
- Architecting the cache
- The A, B, Cs of the cache

Hash-like structure — direct-mapped cache



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`

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

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	0b101010110001111111100000101000011101001100110000
&b[0]	0x558FE0A1DC30	0b101010110001111111100000101000011101110000110000
&a[0][1]	0x558FE0A1D338	0b101010110001111111100000101000011101001100111000
&b[1]	0x558FE0A1DC38	0b101010110001111111100000101000011101110000111000
&a[0][2]	0x558FE0A1D340	0b101010110001111111100000101000011101001101000000
&b[2]	0x558FE0A1DC40	0b101010110001111111100000101000011101110001000000
&a[0][3]	0x558FE0A1D348	0b101010110001111111100000101000011101001101001000
&b[3]	0x558FE0A1DC48	0b101010110001111111100000101000011101110001001000
&a[0][4]	0x558FE0A1D350	0b101010110001111111100000101000011101001101010000
&b[4]	0x558FE0A1DC50	0b101010110001111111100000101000011101110001010000
&a[0][5]	0x558FE0A1D358	0b101010110001111111100000101000011101001101011000
&b[5]	0x558FE0A1DC58	0b101010110001111111100000101000011101110001011000
&a[0][6]	0x558FE0A1D360	0b101010110001111111100000101000011101001101100000
&b[6]	0x558FE0A1DC60	0b101010110001111111100000101000011101110001100000
&a[0][7]	0x558FE0A1D368	0b101010110001111111100000101000011101001101101000
&b[7]	0x558FE0A1DC68	0b101010110001111111100000101000011101110001101000
&a[0][8]	0x558FE0A1D370	0b101010110001111111100000101000011101001101110000
&b[8]	0x558FE0A1DC70	0b101010110001111111100000101000011101110001110000
&a[0][9]	0x558FE0A1D378	0b101010110001111111100000101000011101001101111000
&b[9]	0x558FE0A1DC78	0b101010110001111111100000101000011101110001111000

Simulate a direct-mapped cache

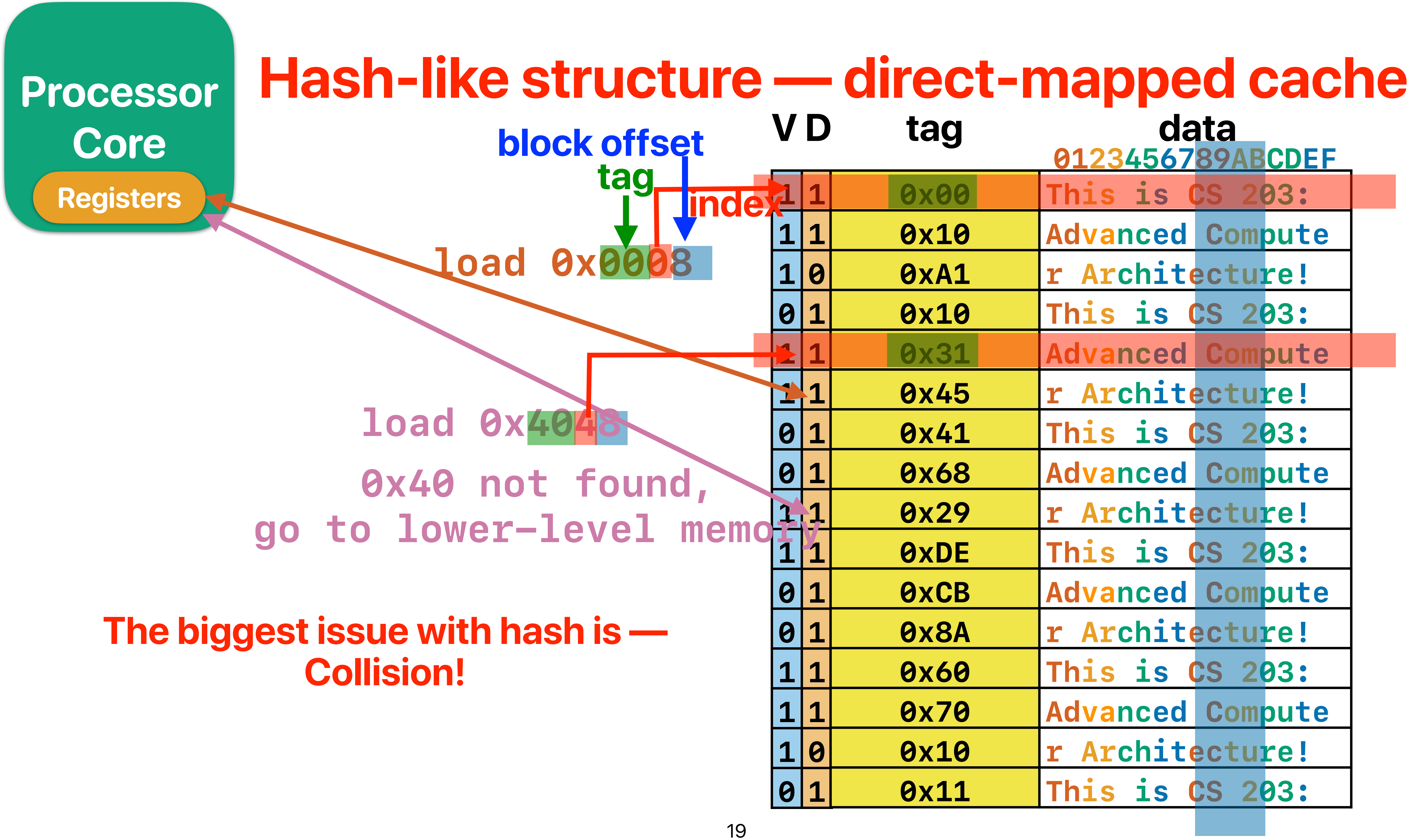
	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

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

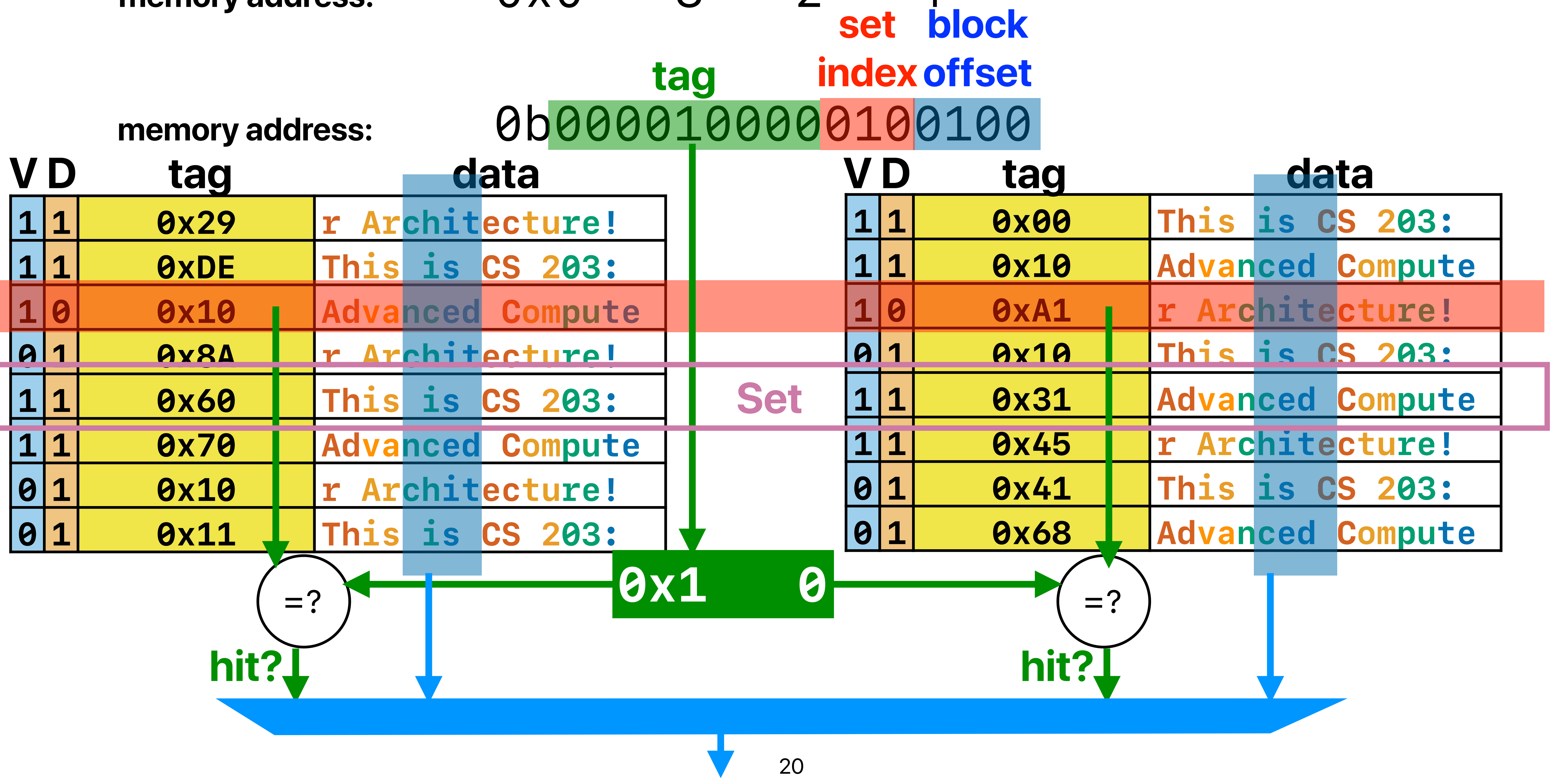


V	D	tag	data 0123456789ABCDEF
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced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



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`

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

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	0b101010110001111111100000101000011101001100110000
&b[0]	0x558FE0A1DC30	0b101010110001111111100000101000011101110000110000
&a[0][1]	0x558FE0A1D338	0b101010110001111111100000101000011101001100111000
&b[1]	0x558FE0A1DC38	0b101010110001111111100000101000011101110000111000
&a[0][2]	0x558FE0A1D340	0b101010110001111111100000101000011101001101000000
&b[2]	0x558FE0A1DC40	0b101010110001111111100000101000011101110001000000
&a[0][3]	0x558FE0A1D348	0b101010110001111111100000101000011101001101001000
&b[3]	0x558FE0A1DC48	0b101010110001111111100000101000011101110001001000
&a[0][4]	0x558FE0A1D350	0b101010110001111111100000101000011101001101010000
&b[4]	0x558FE0A1DC50	0b101010110001111111100000101000011101110001010000
&a[0][5]	0x558FE0A1D358	0b101010110001111111100000101000011101001101011000
&b[5]	0x558FE0A1DC58	0b101010110001111111100000101000011101110001011000
&a[0][6]	0x558FE0A1D360	0b101010110001111111100000101000011101001101100000
&b[6]	0x558FE0A1DC60	0b101010110001111111100000101000011101110001100000
&a[0][7]	0x558FE0A1D368	0b101010110001111111100000101000011101001101101000
&b[7]	0x558FE0A1DC68	0b101010110001111111100000101000011101110001101000
&a[0][8]	0x558FE0A1D370	0b101010110001111111100000101000011101001101110000
&b[8]	0x558FE0A1DC70	0b101010110001111111100000101000011101110001110000
&a[0][9]	0x558FE0A1D378	0b101010110001111111100000101000011101001101111000
&b[9]	0x558FE0A1DC78	0b101010110001111111100000101000011101110001111000

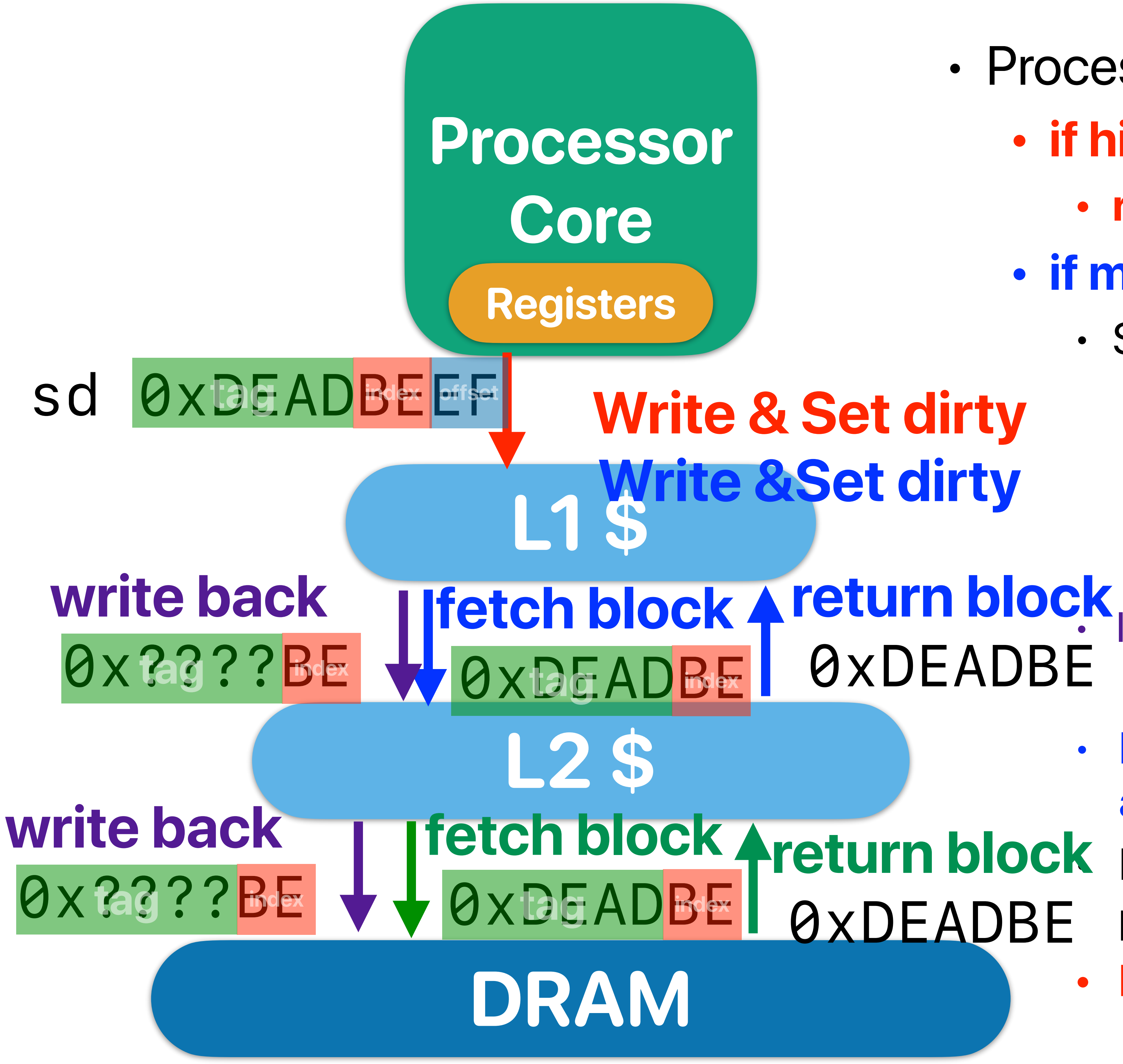
Simulate a 2-way cache

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

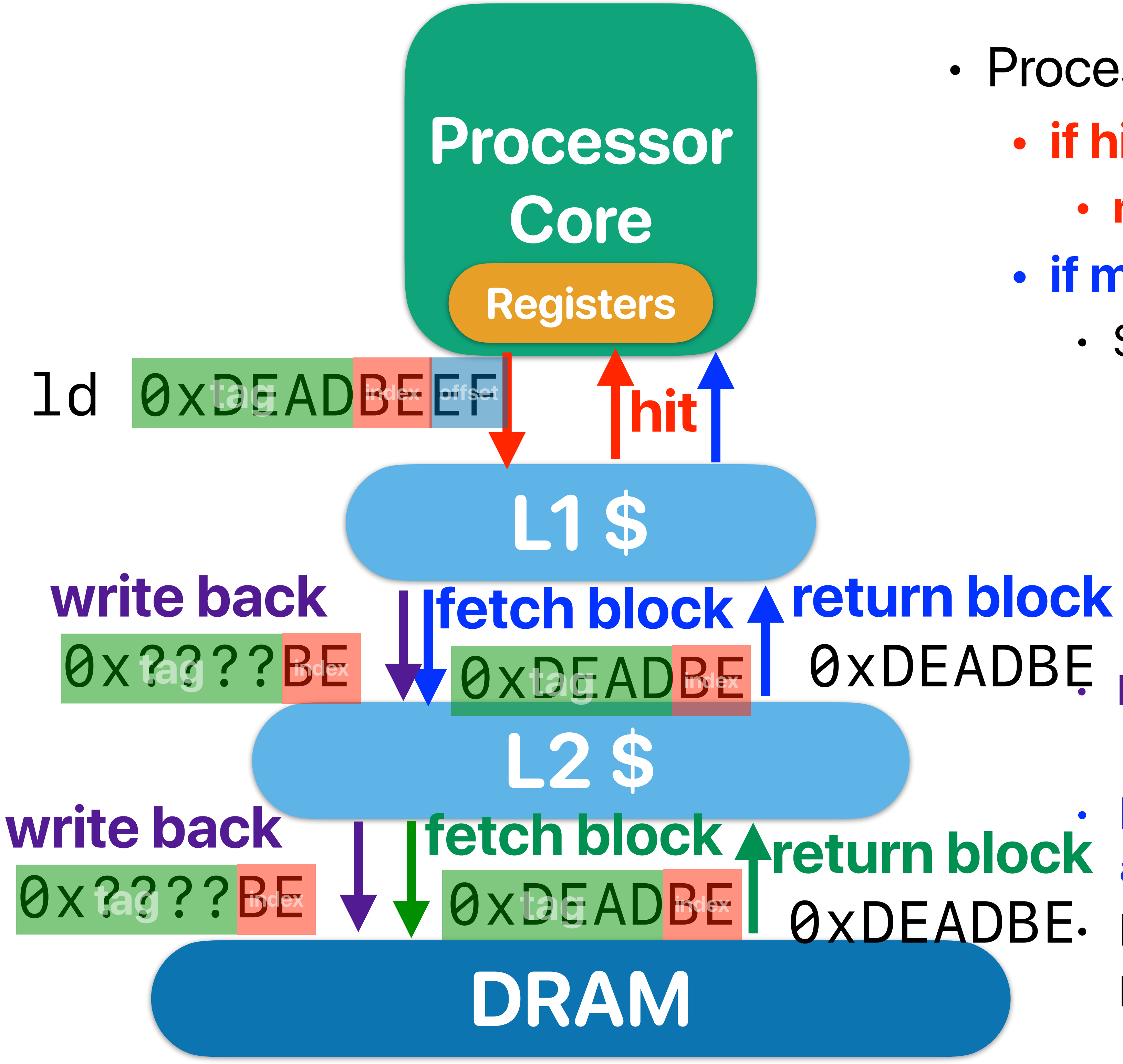
Put everything all together: How cache interacts with CPU

What happens when we write data



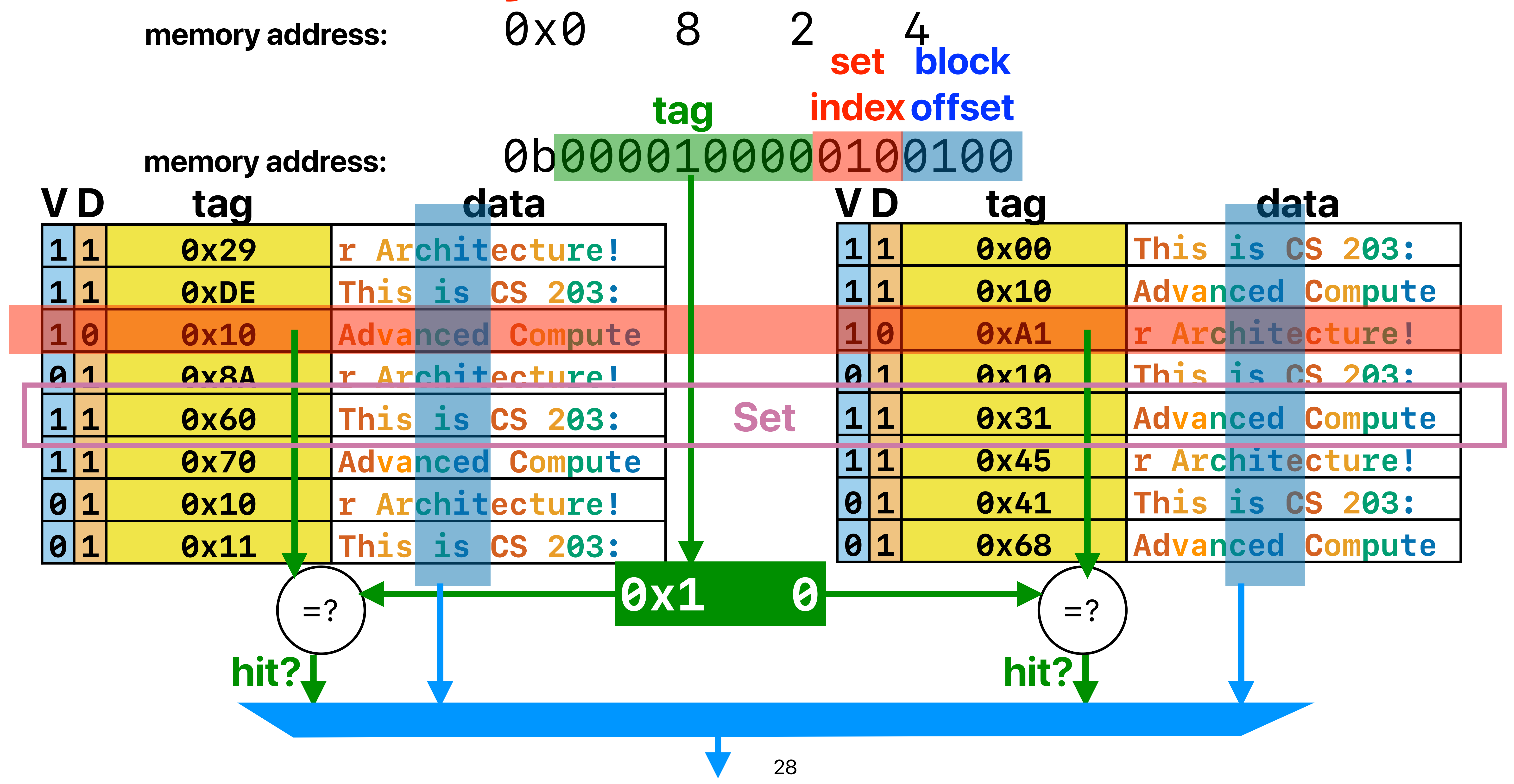
- Processor sends load request to L1-$
  - **if hit**
    - **return data — set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target “set” is not full — select an empty/invalidated block as the victim block
      - If the target “set is full — select a victim block using some policy
      - 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



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 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: 0b	000010000	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 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

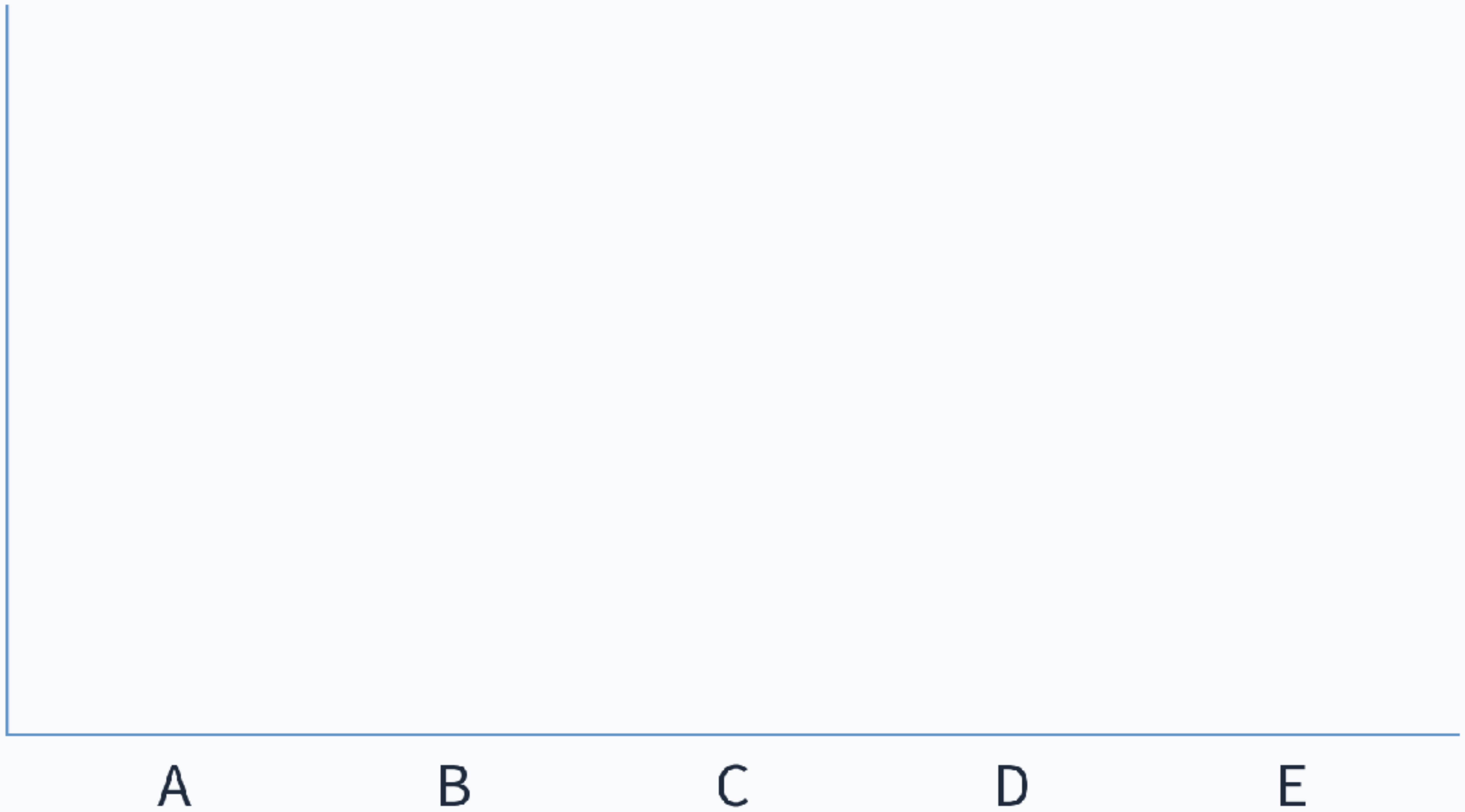
- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?

A. Tag is 52 bits

B. Index is 6 bits

C. Offset is 6 bits

D. The cache has 128 sets


A
B
C
D
E

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

intel Core i7

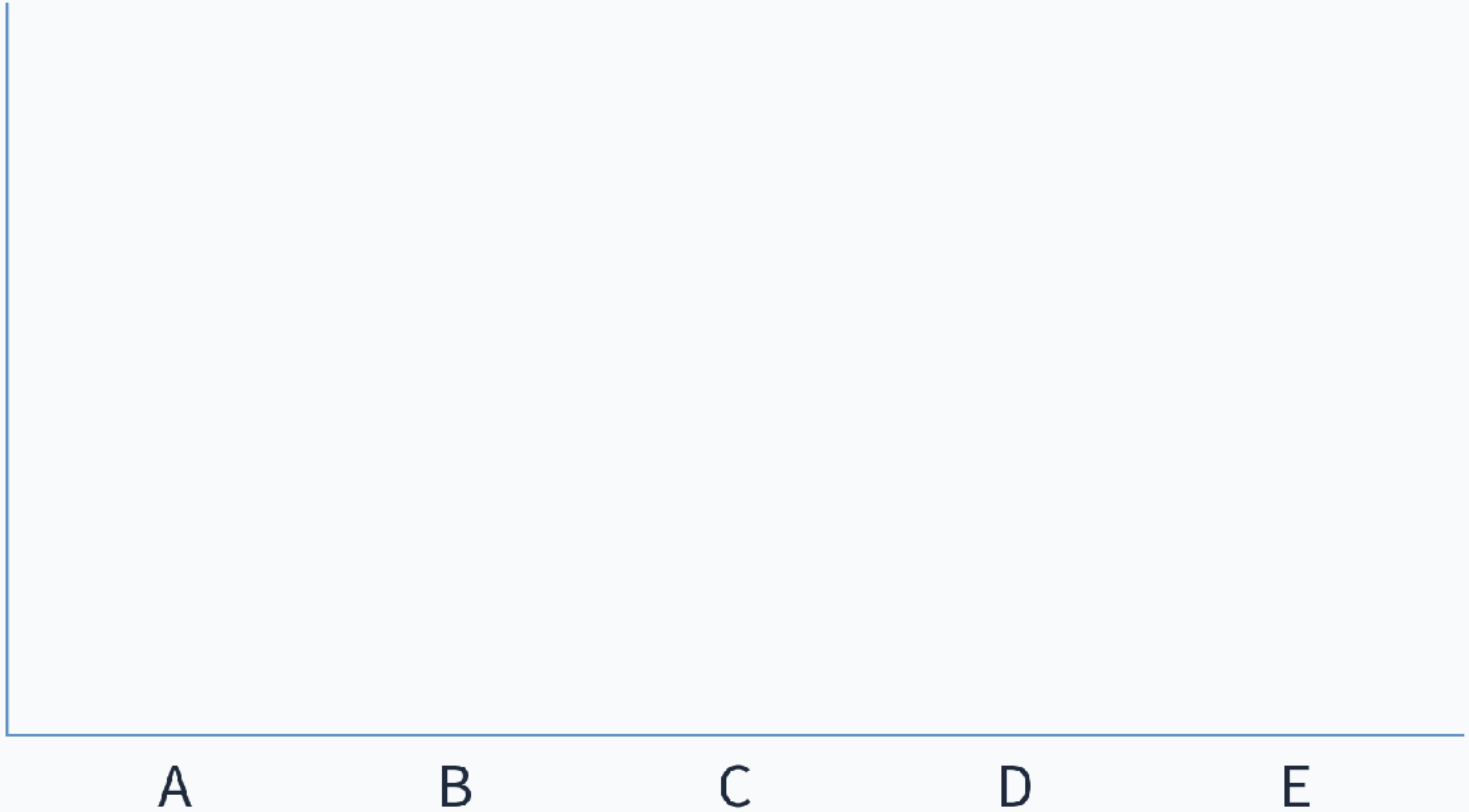
- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?

A. Tag is 52 bits

B. Index is 6 bits

C. Offset is 6 bits

D. The cache has 128 sets


A
B
C
D
E

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]

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

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

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

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

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

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

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[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; 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?

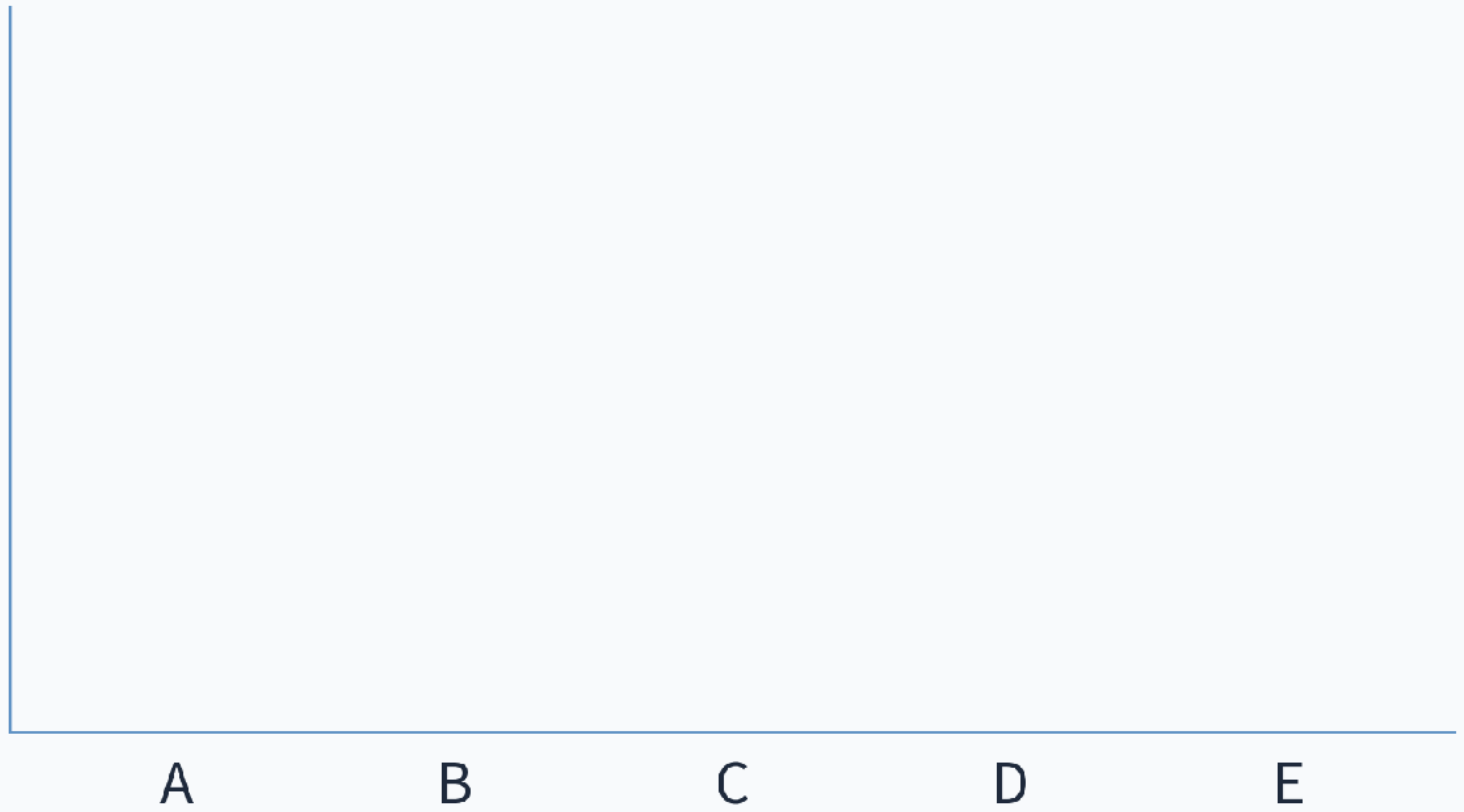
A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%


A
B
C
D
E

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[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; 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?

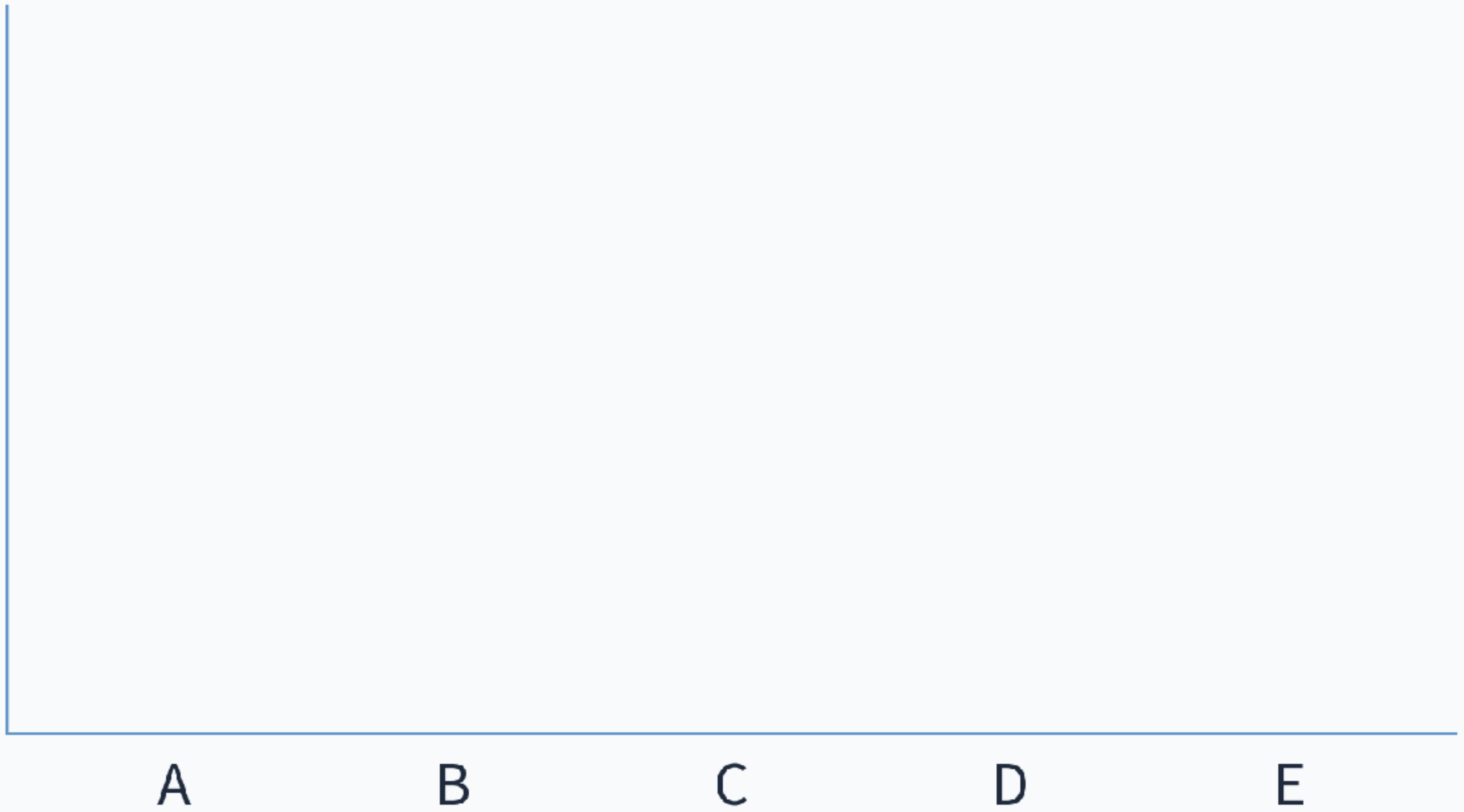
A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%


A
B
C
D
E

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[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; 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	0b00010000000000000000	0x8	0x0	Miss	
b[0]	0x20000	0b00100000000000000000	0x10	0x0	Miss	
c[0]	0x30000	0b00110000000000000000	0x18	0x0	Miss	
d[0]	0x40000	0b01000000000000000000	0x20	0x0	Miss	
e[0]	0x50000	0b01010000000000000000	0x28	0x0	Miss	a[0-7]
a[1]	0x10008	0b00010000000000001000	0x8	0x0	Miss	b[0-7]
b[1]	0x20008	0b00100000000000001000	0x10	0x0	Miss	c[0-7]
c[1]	0x30008	0b00110000000000001000	0x18	0x0	Miss	d[0-7]
d[1]	0x40008	0b01000000000000001000	0x20	0x0	Miss	e[0-7]
e[1]	0x50008	0b01010000000000001000	0x28	0x0	Miss	a[0-7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

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[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; 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%]

Knowing the cache performance in the play!

- Valgrind
  - valgrind --tool=cachegrind cmd
  - cachegrind is a tool profiling the cache performance
- Performance counter
  - Intel® Performance Counter Monitor http://www.intel.com/software/pcm/
  - perf stat -d -d -d [cmd]

Announcement

- Regarding assignments
    - Please follow the EXACT instructions — any small thing you missed in the document can lead to undesirable outcome
    - Some of you complain more guidance and instructions and some you complain about the length of the content — that's why we need office hours
    - 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 already up and please START NOW
    - C++ programming — can't the demo regarding performance convince you to use C/C++?
    - Any kind of cheating is NOT ALLOWED
        - Gradescope has similarity check on your code, we've identified cases with 100% similarity and will directly send these cases to misconduct office if any were identified again.
        - We log everything on our servers.
        - Midterm will be on gradescope as well and we will run these checks
- Reading quiz due this Wednesday

Computer
Science &
Engineering

203

つづく