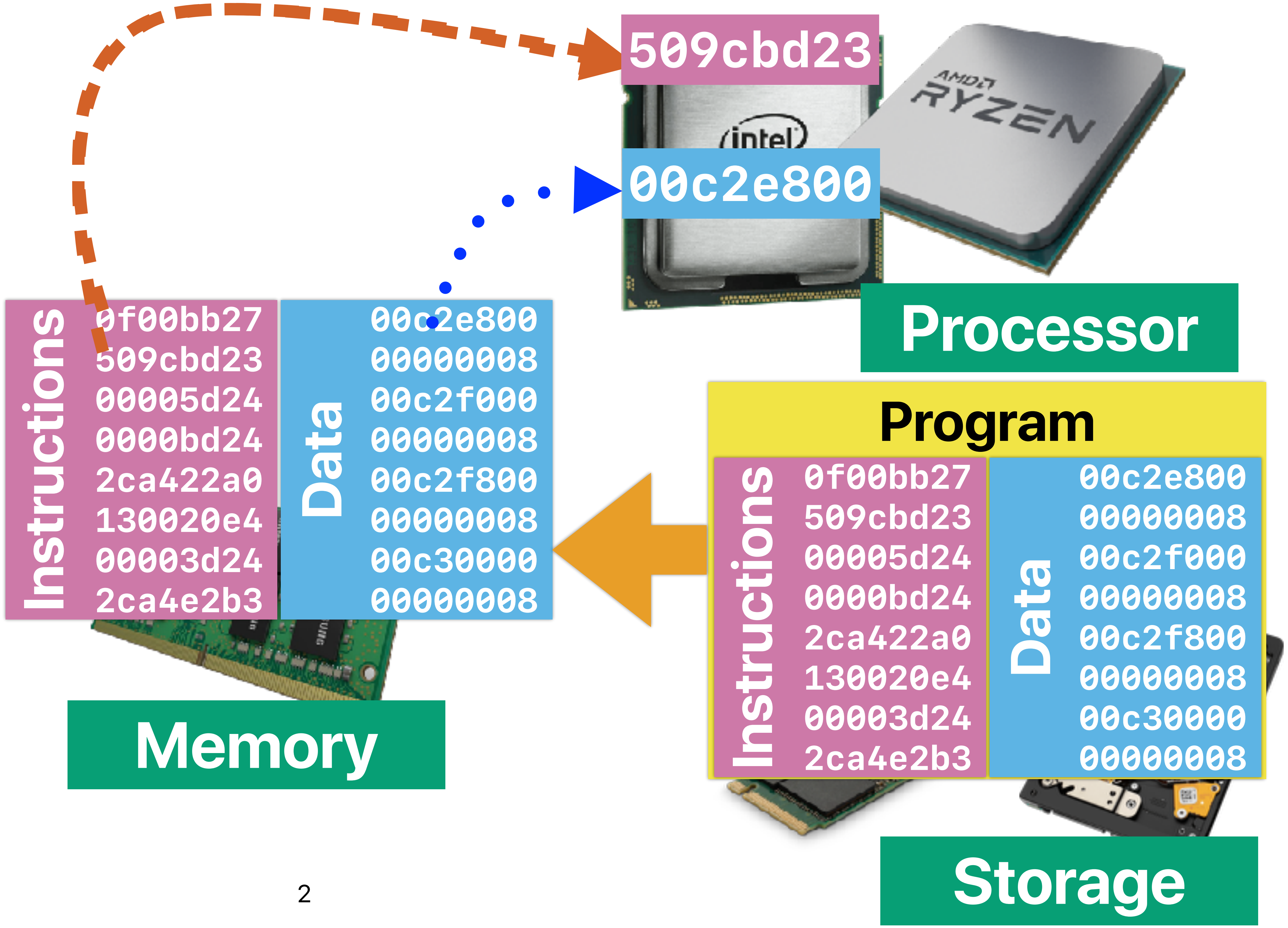
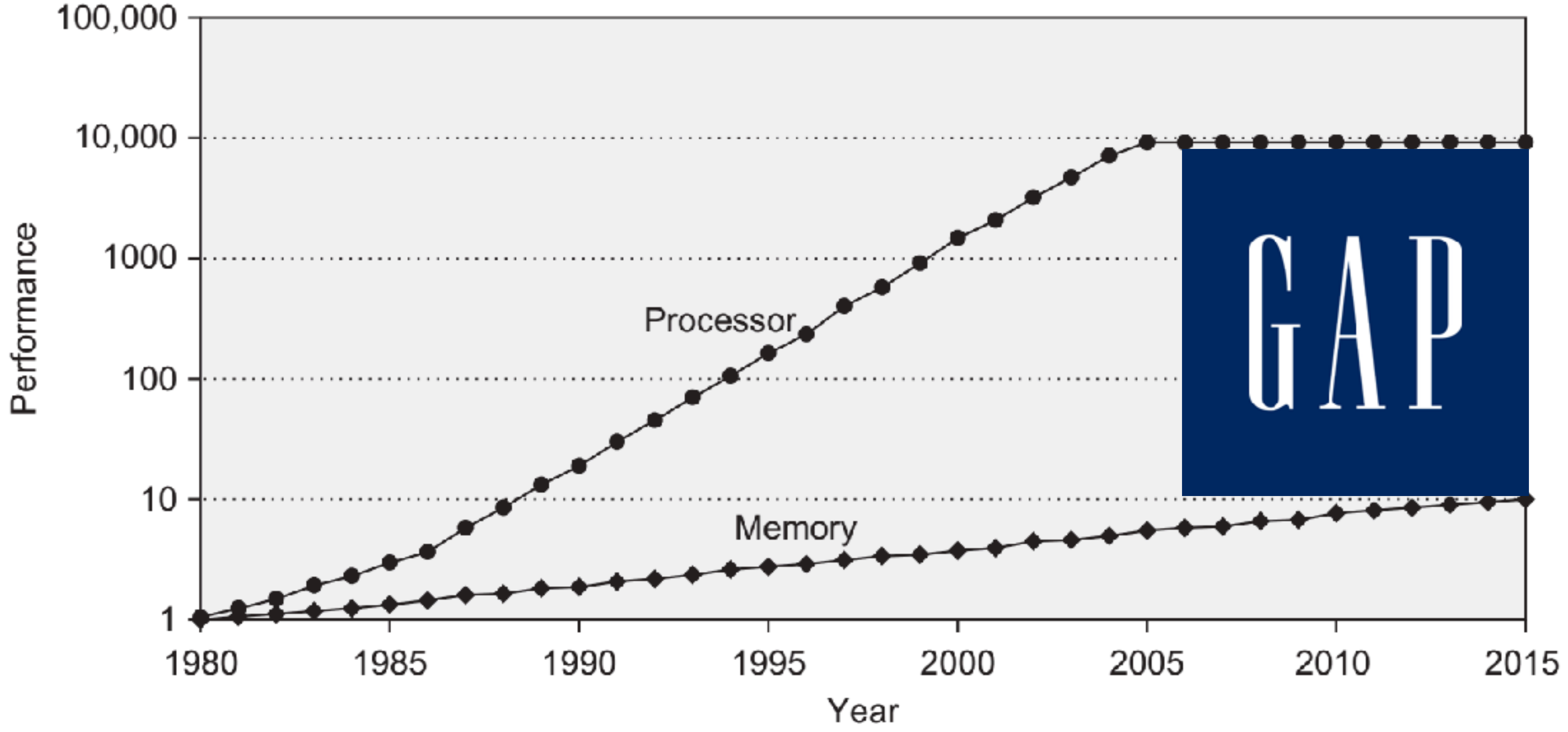
Memory Hierarchy (3): Cache misses and where to find them

Hung-Wei Tseng

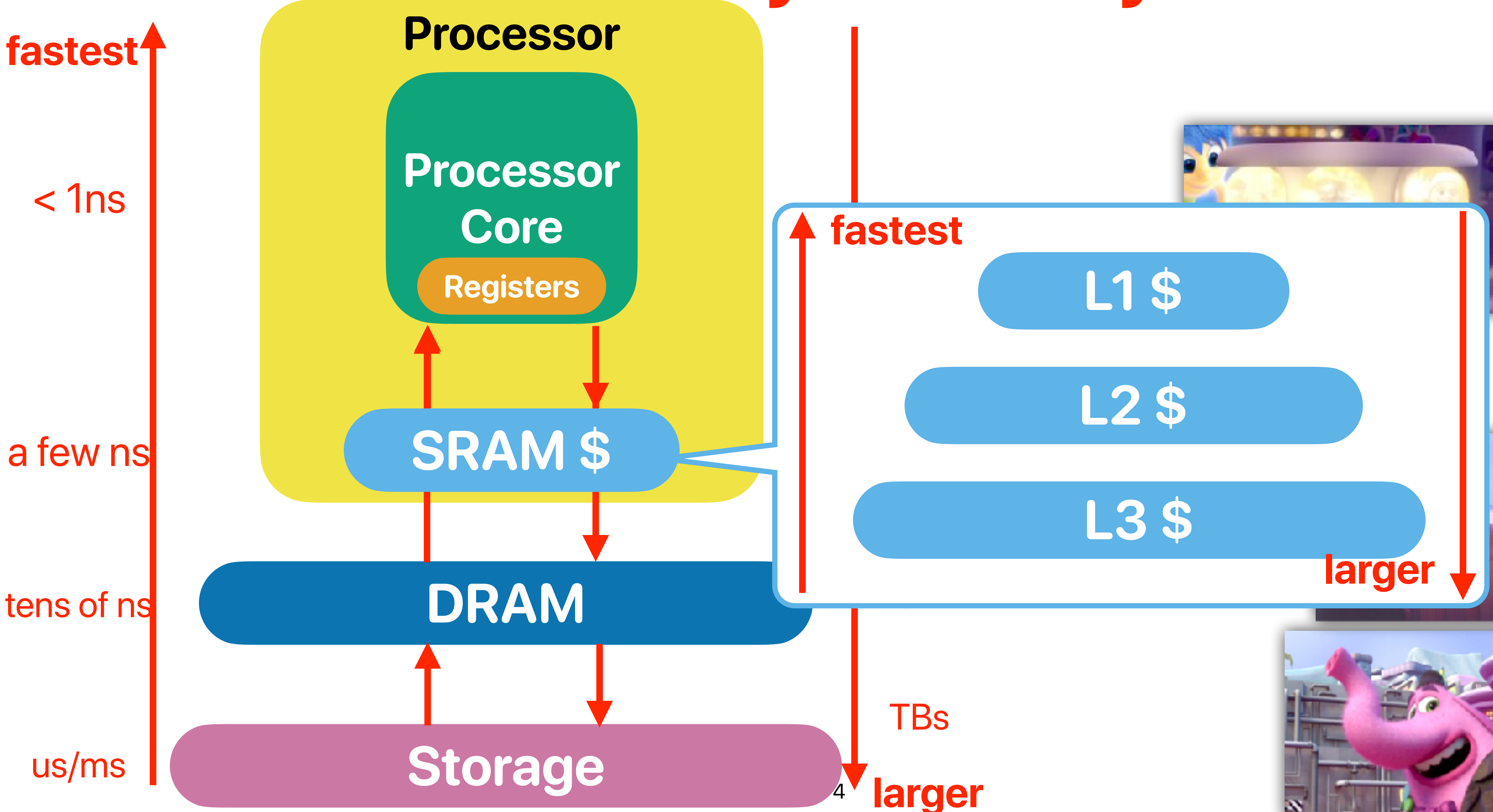
Recap: von Neumman Architecture

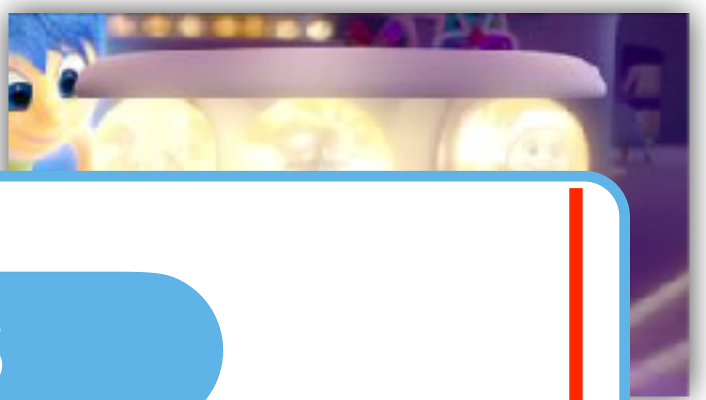


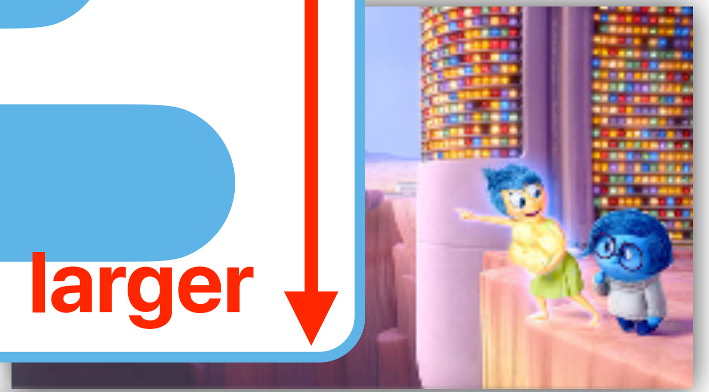
Recap: Performance gap between Processor/Memory



Memory Hierarchy







Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — [illegible] in an array, then the [illegible]

We need to "cache consecutive memory locations" every time

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

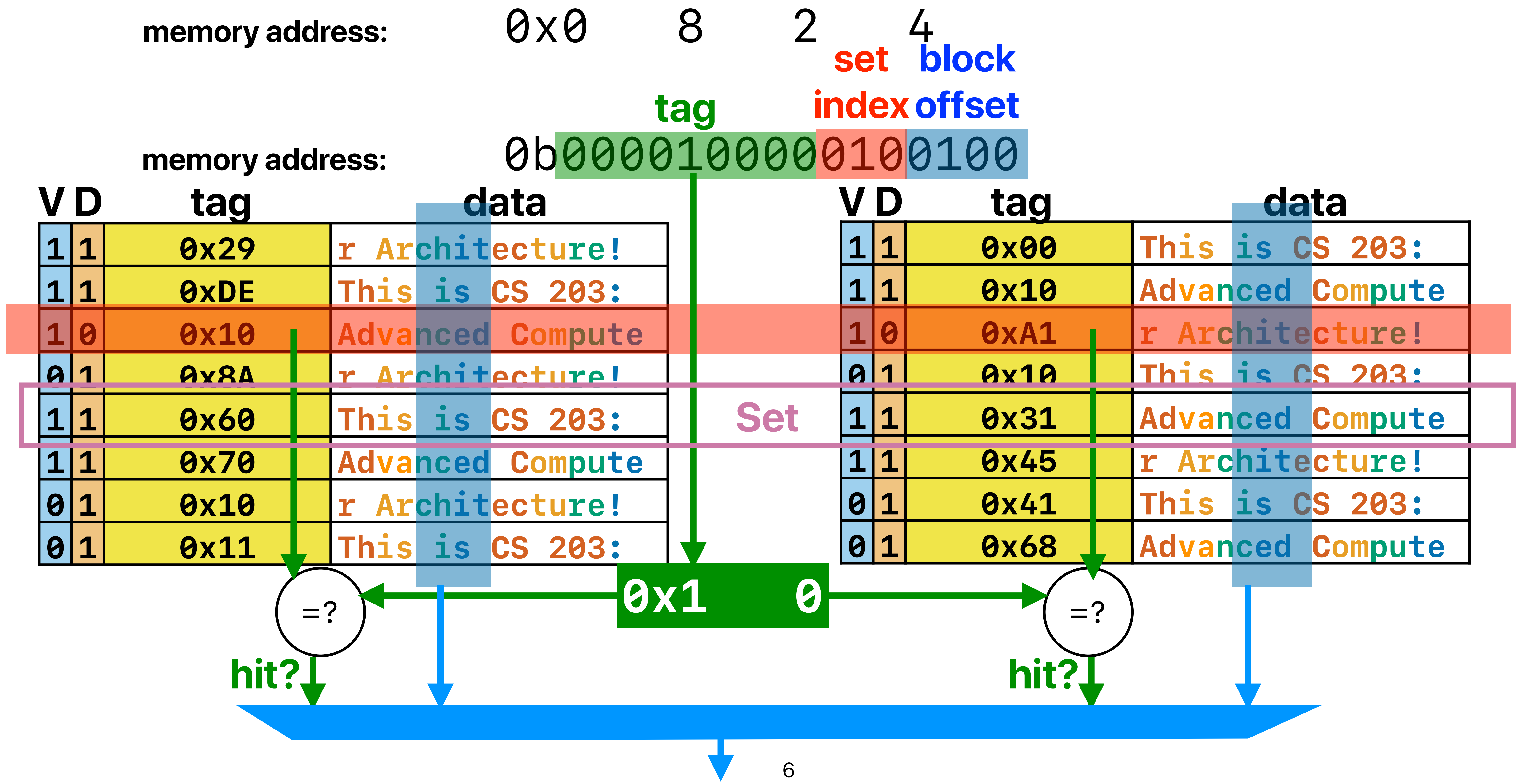
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

We need to "cache frequently used memory blocks"

— the cache should store a few blocks

— the cache must be able to distinguish blocks

Recap: Way-associative cache



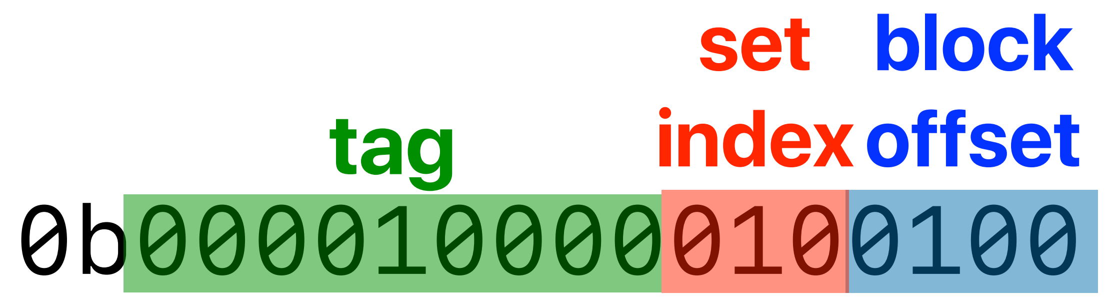
V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

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

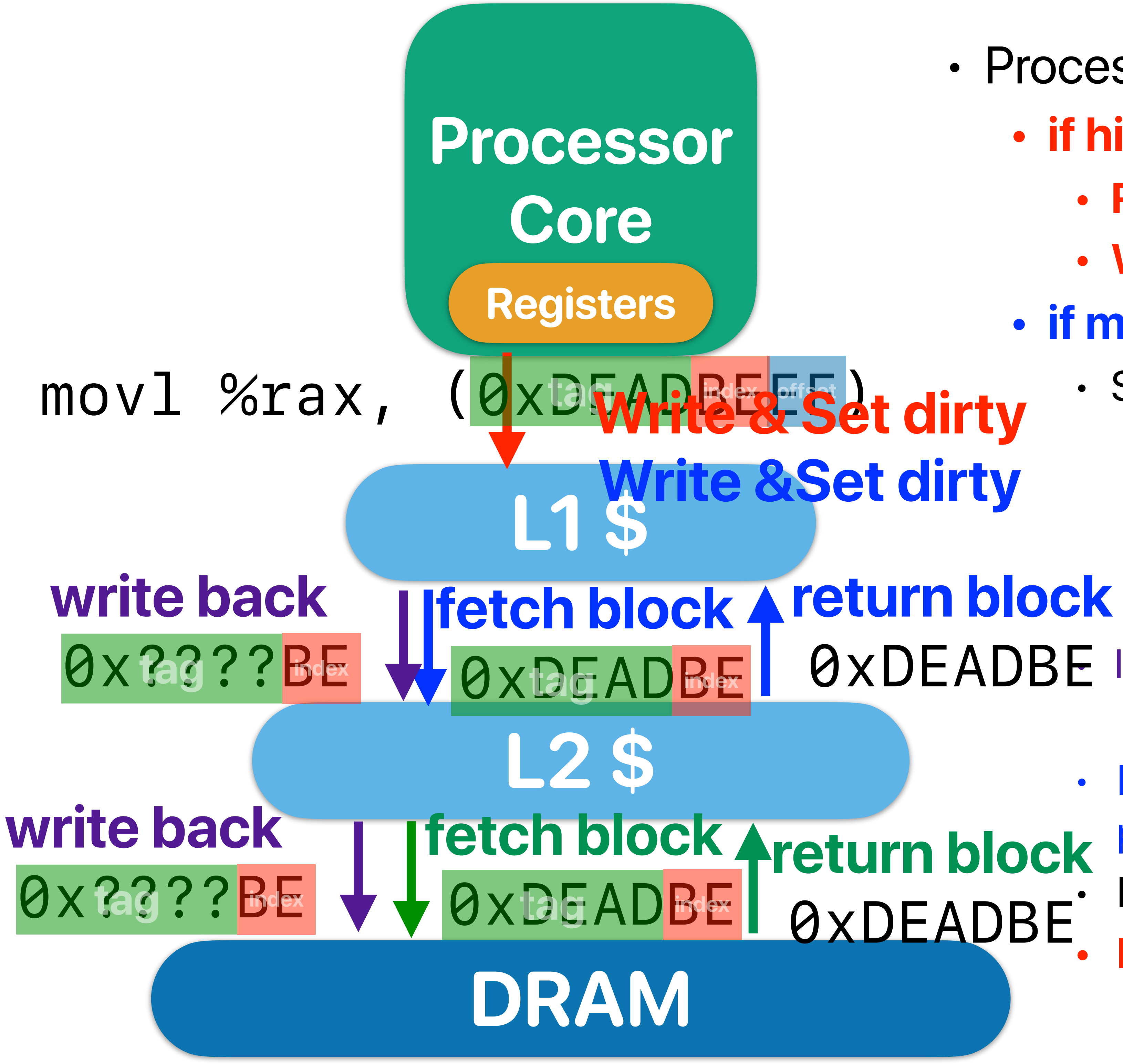
Review: C = ABS

- **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
- 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

memory address:



The complete picture



- Processor sends memory access request to L1-$
  - **if hit**
    - **Read - return data**
    - **Write - update & 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**

Takeaways: The A, B, Cs of memory hierarchy

- Cache works because our programs have localities
- Cache architecture aims at capturing localities
  - Blocks for spatial locality
  - Multiple blocks for temporal locality
  - Set associativity for efficiency
  - C = A B S
- Memory accesses are hierarchical
  - We always consult L1 caches and gradually walk down
  - Partition the address using parameters derived from C=ABS
  - Hit: return data
  - Miss: search the lower-level memory hierarchy and replace/write back

Outline

- Estimating the code performance on cache
- 3Cs — where cache misses are coming from
- Optimizing cache performance — the hardware perspective

Simulate the cache to know how my code performs!

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[0][0] = 0x558FE0A1D330
&vector[0] = 0x558FE0A1DC30
```

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&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

- 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`

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

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;
}
```

```
&matrix[0][0] = 0x558FE0A1D330
&vector[0] = 0x558FE0A1DC30
```

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!

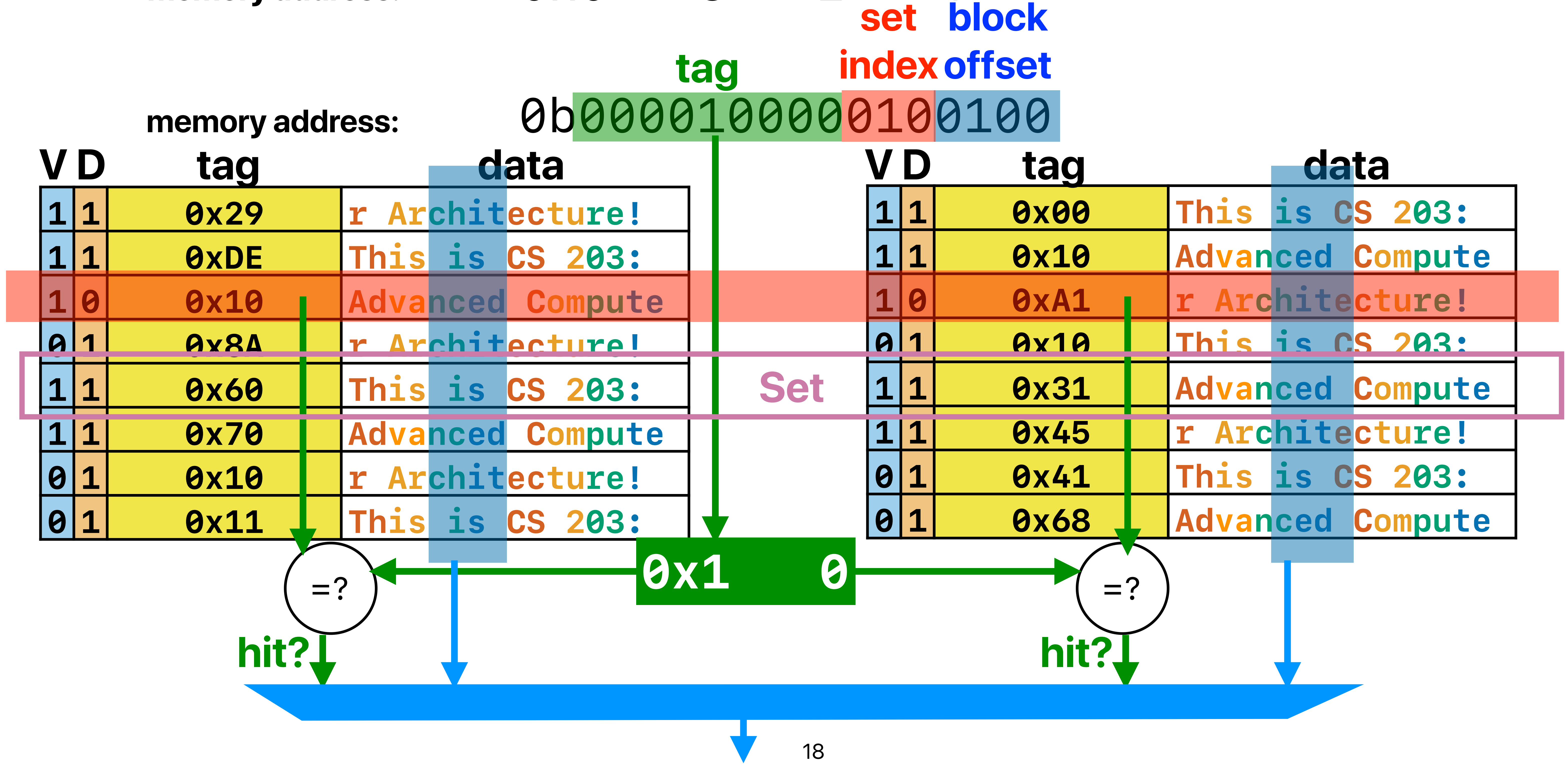
tag index

	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

memory address: 0b000010000 010 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:

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

tag index

```
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;
}
```

Address (Hex)		Tag	Index	
&a[0][0]	0x558FE0A1D330	0xAB1FC143A6	0x3	000
&b[0]	0x558FE0A1DC30	0xAB1FC143B8	0x3	000
&a[0][1]	0x558FE0A1D338	0xAB1FC143A6	0x3	000
&b[1]	0x558FE0A1DC38	0xAB1FC143B8	0x3	000
&a[0][2]	0x558FE0A1D340	0xAB1FC143A6	0x4	000
&b[2]	0x558FE0A1DC40	0xAB1FC143B8	0x4	000
&a[0][3]	0x558FE0A1D348	0xAB1FC143A6	0x4	000
&b[3]	0x558FE0A1DC48	0xAB1FC143B8	0x4	000
&a[0][4]	0x558FE0A1D350	0xAB1FC143A6	0x5	000
&b[4]	0x558FE0A1DC50	0xAB1FC143B8	0x5	000
&a[0][5]	0x558FE0A1D358	0xAB1FC143A6	0x5	000
&b[5]	0x558FE0A1DC58	0xAB1FC143B8	0x5	000
&a[0][6]	0x558FE0A1D360	0xAB1FC143A6	0x6	000
&b[6]	0x558FE0A1DC60	0xAB1FC143B8	0x6	000
&a[0][7]	0x558FE0A1D368	0xAB1FC143A6	0x6	000
&b[7]	0x558FE0A1DC68	0xAB1FC143B8	0x6	000
&a[0][8]	0x558FE0A1D370	0xAB1FC143A6	0x7	000
&b[8]	0x558FE0A1DC70	0xAB1FC143B8	0x7	000
&a[0][9]	0x558FE0A1D378	0xAB1FC143A6	0x7	000
&b[9]	0x558FE0A1DC78	0xAB1FC143B8	0x7	000

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[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%

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	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

- 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%]

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[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%

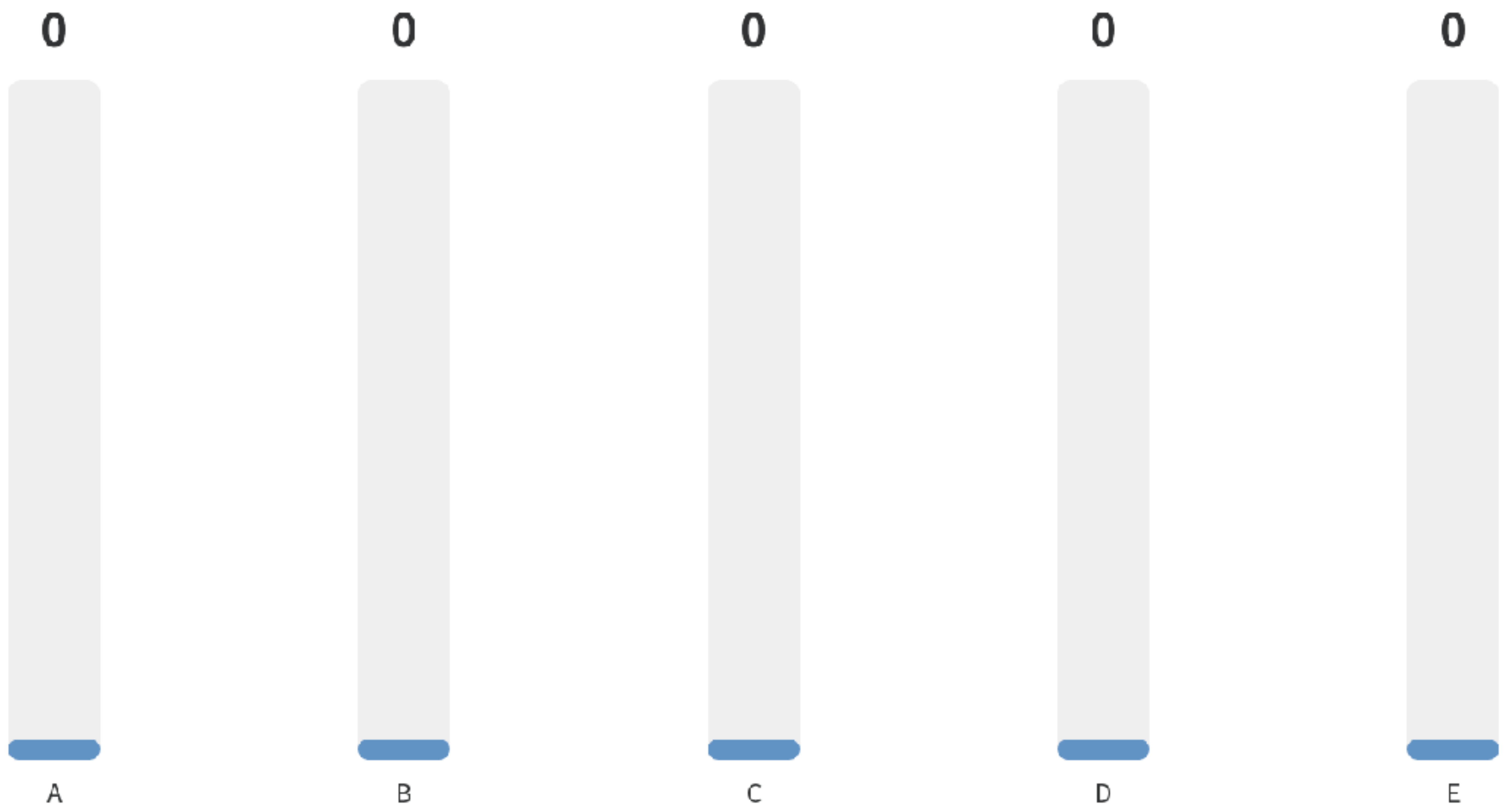
B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

0



intel Core i7

- Size 48KB, 12-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 = 8 * 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 32KB, 8-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 = 8 * 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[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!

Announcement

- Reading quiz #4 due **next Tuesday** before the lecture
- Assignment #3 is released, due **next Thursday**
  - Please check our course website https://www.escalab.org/classes/cs203-2024sp/
  - Start early to get feedback from the autograder
- Check your participation grades at https://escalab.org/my_grades
  - We match your records based on your UCRNetID@ucr.edu
  - Please make sure your gradescope uses your UCRNetID@ucr.edu
  - We count only 50% attendance for full credits in participation — you still have lots of chances to improve
  - We drop 2 reading quizzes and 1 assignment and we won't allow late/make-ups
- Hung-Wei's office hour this week moves to Thursday 2p-4p

Computer
Science &
Engineering

つづく