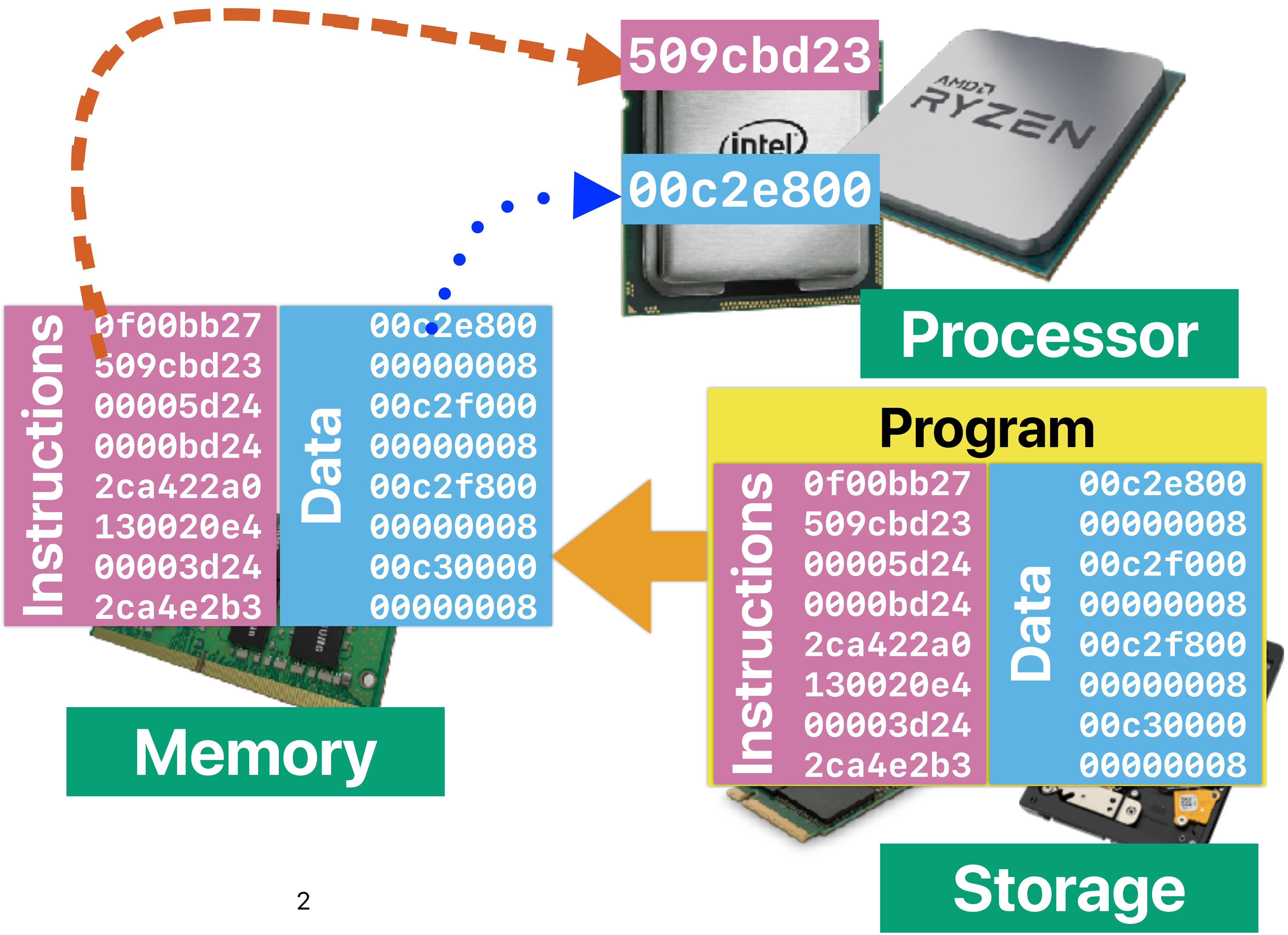
Memory Hierarchy Inside Out:
(3) Cache misses and their remedies

Hung-Wei Tseng

von Neumman Architecture



509cbd23

00c2e800

Processor

Memory

Instructions

```
0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
```

Data

```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

Storage

Program

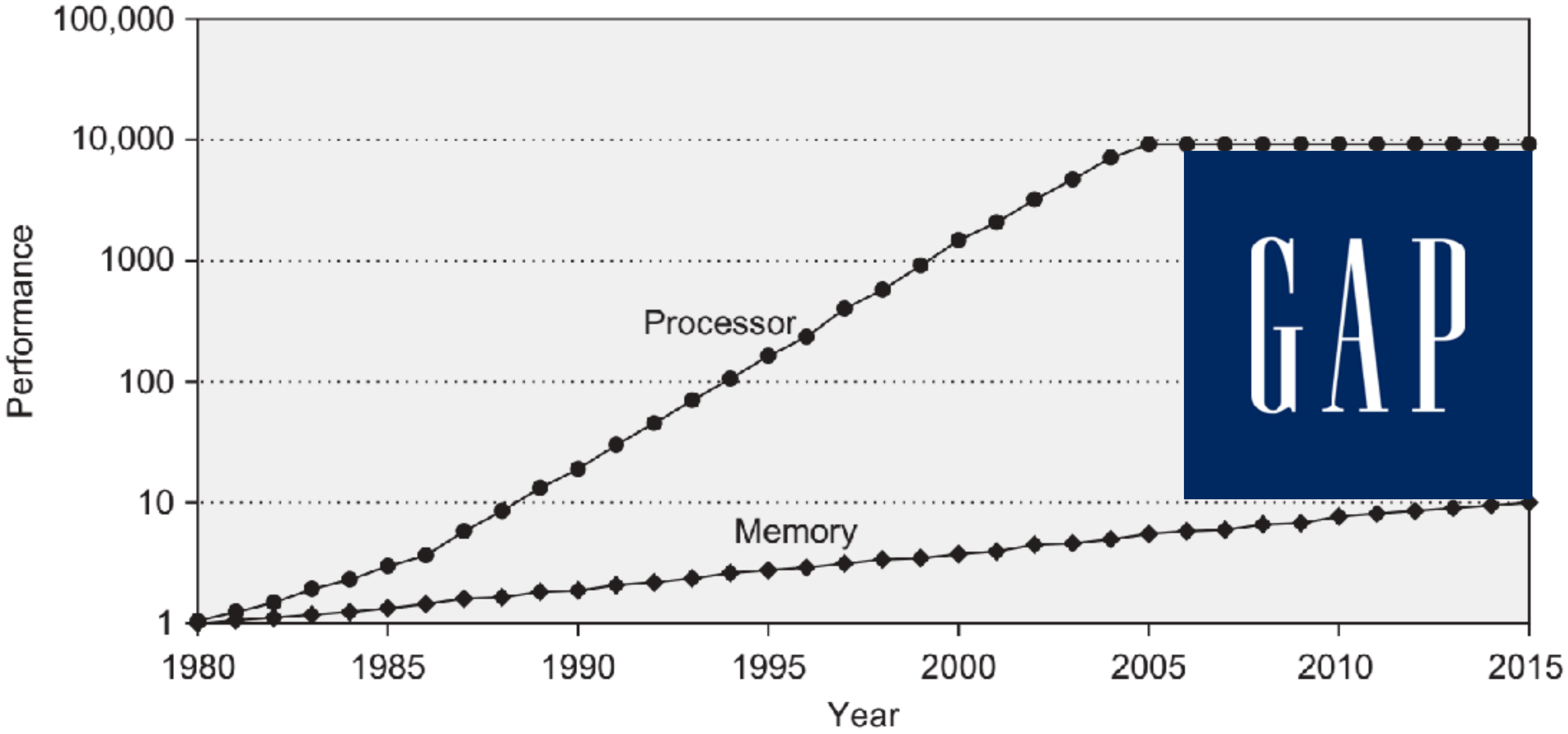
Instructions

```
0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
```

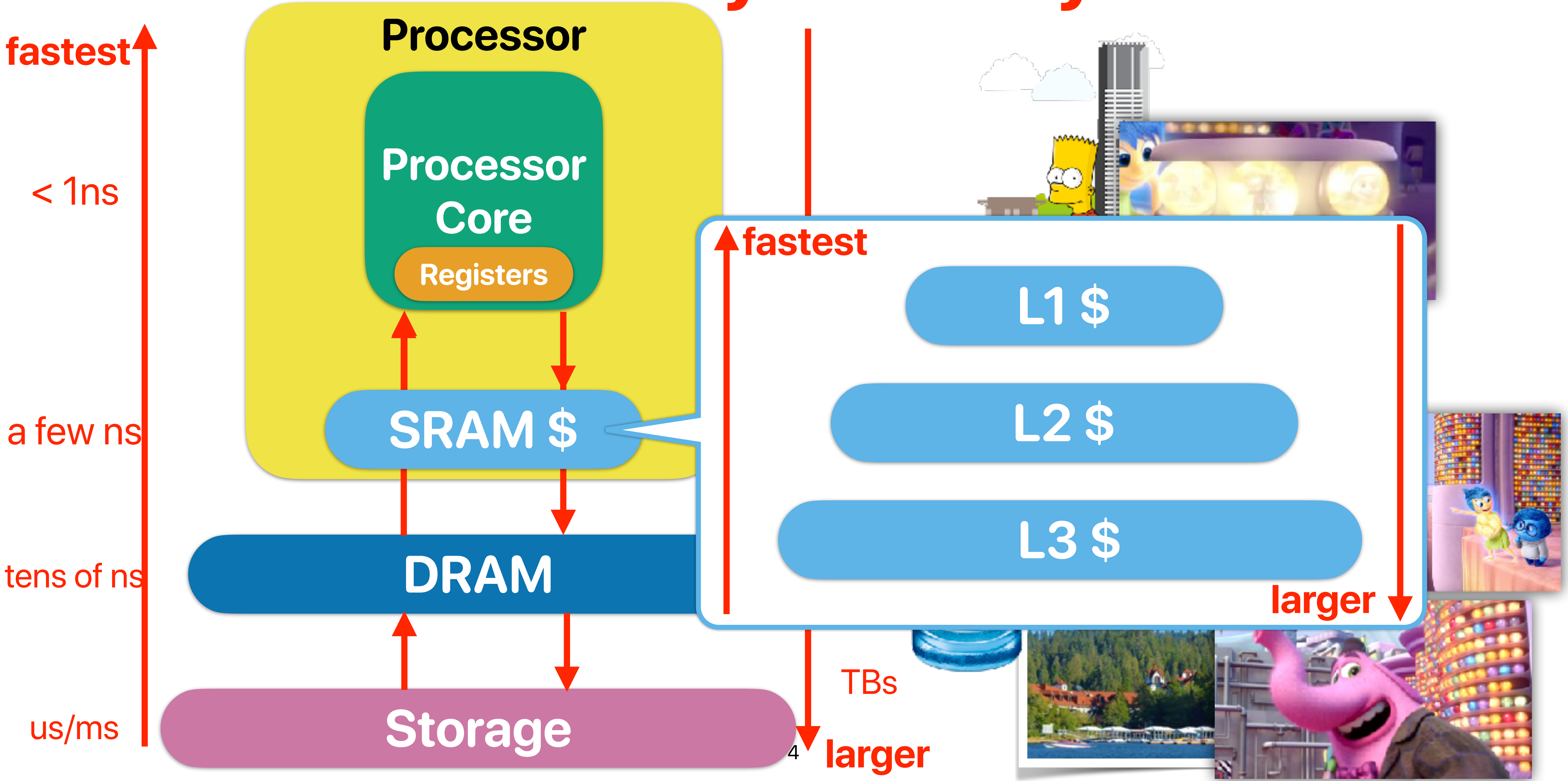
Data

```
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008
```

Recap: Performance gap between Processor/Memory



Memory Hierarchy



fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

fastest

L1 $

L2 $

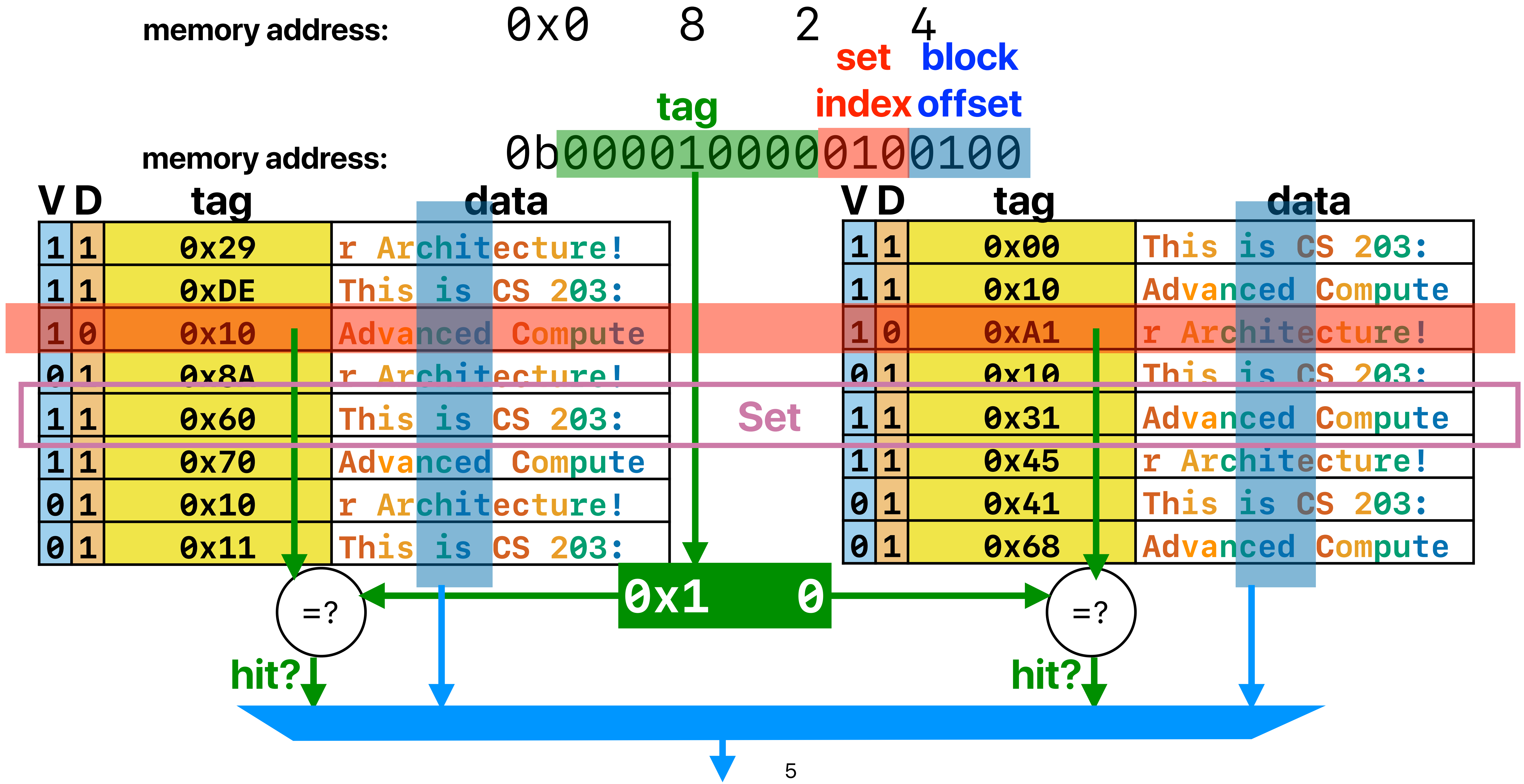
L3 $

larger

TBs

larger

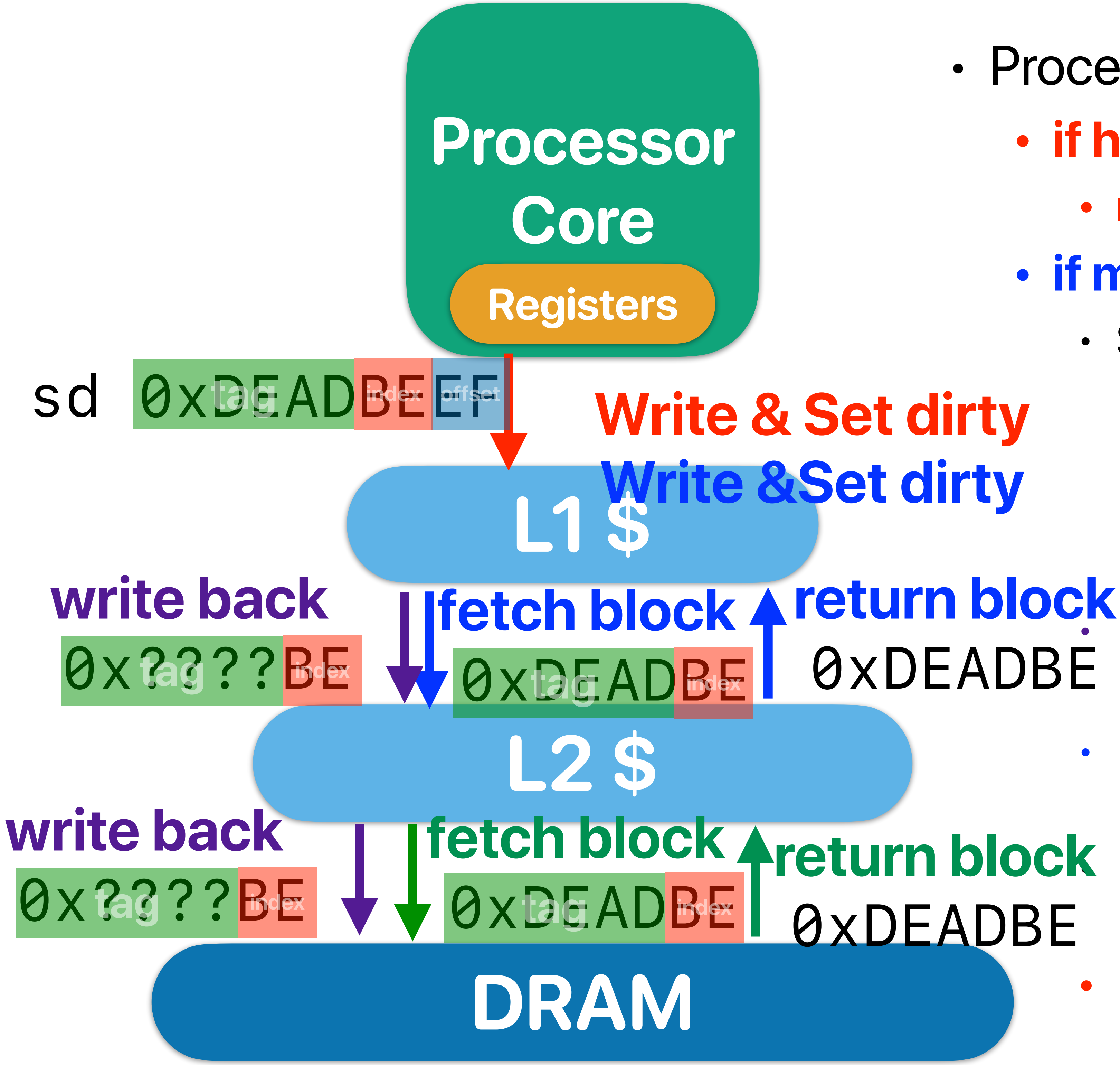
Way-associative cache



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

What happens when we write data



Processor Core

Registers

sd 0xDEADBEEF tag index offset

Write & Set dirty

Write &Set dirty

L1 $

write back

0x????BE tag index

fetch block

0xDEADBE tag index

return block

0xDEADBE

L2 $

write back

0x????BE tag index

fetch block

0xDEADBE tag index

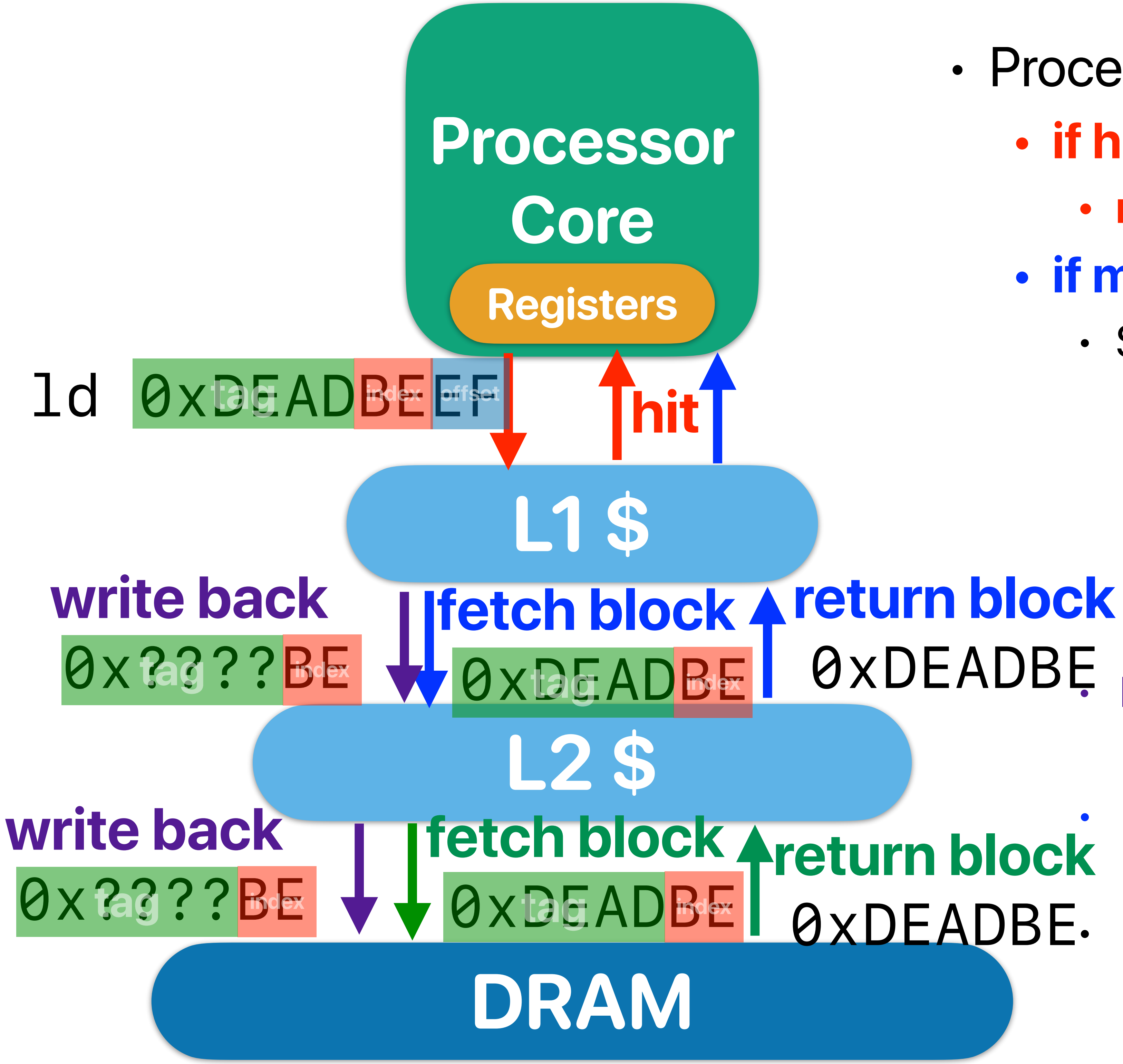
return block

0xDEADBE

DRAM

- 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

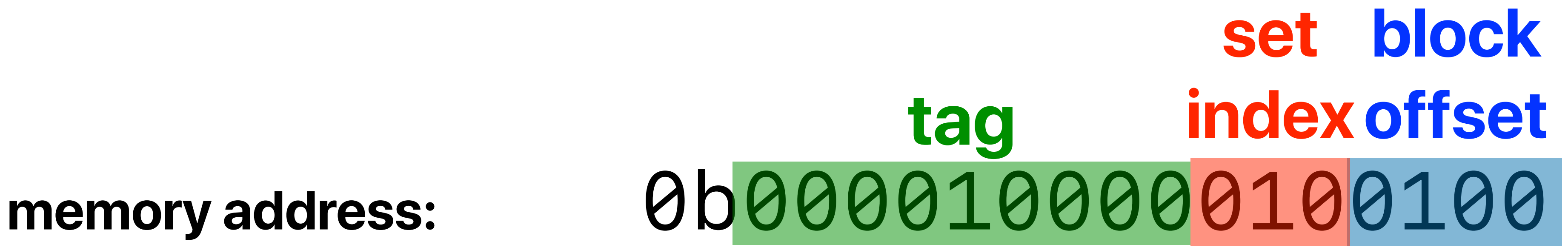


- Processor sends load request to L1-$
  - if hit
    - return data
  - if miss
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred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

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



set index

block offset

tag

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

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

Outline

- The A, B, Cs of the cache (cont.)
- The causes of cache misses
- The remedies of cache misses — the hardware version

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]

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

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

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

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

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

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

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%

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.

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

```
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]
```

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]

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

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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%

intel Core i7

- D-L1 Cache configuration of intel Core i7
    - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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%

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

intel Core i7

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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[8192], b[8192], c[8192], d[8192], e[8192];
/* 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!

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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%

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

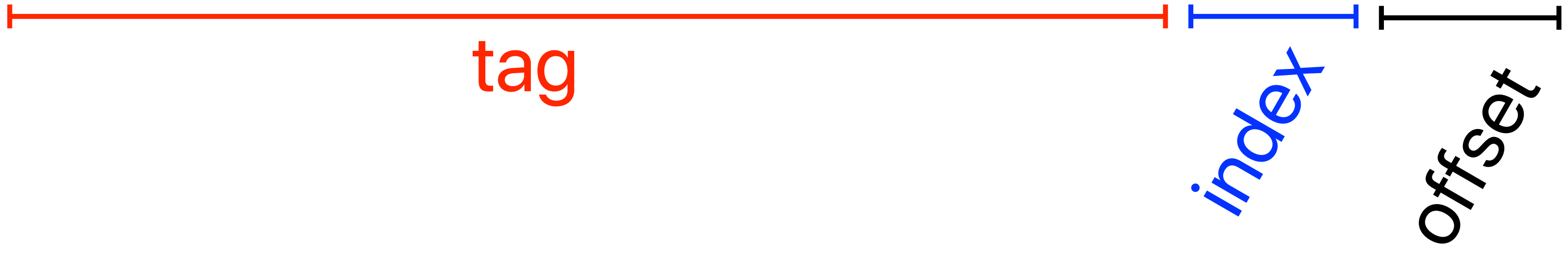
Cause of cache misses

3Cs of misses

- Compulsory miss
    - Cold start miss. First-time access to a block
- Capacity miss
    - The working set size of an application is bigger than cache size
- Conflict miss
    - Required data replaced by block(s) mapping to the same set
    - Similar collision in hash

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

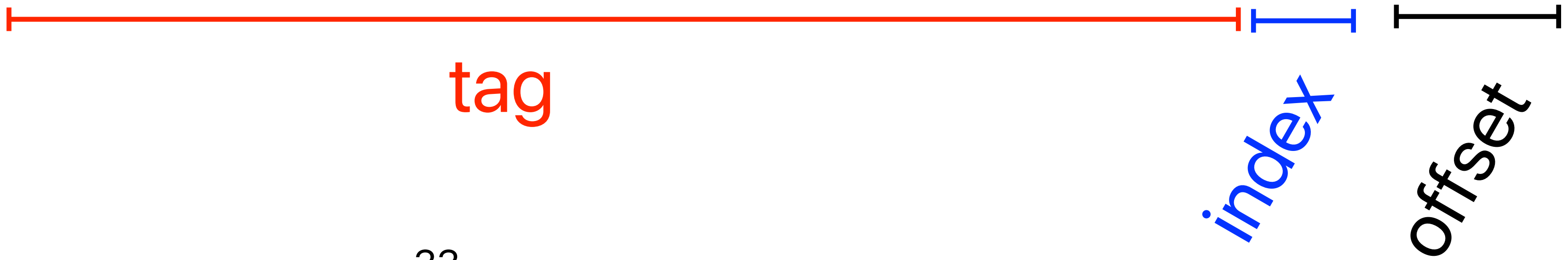
	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

tag index

	Address (Hex)	
&a[0][0]	0x558FE0A1D330	compulsory miss
&b[0]	0x558FE0A1DC30	compulsory miss
&a[0][1]	0x558FE0A1D338	conflict miss
&b[1]	0x558FE0A1DC38	conflict miss
&a[0][2]	0x558FE0A1D340	compulsory miss
&b[2]	0x558FE0A1DC40	compulsory miss
&a[0][3]	0x558FE0A1D348	conflict miss
&b[3]	0x558FE0A1DC48	conflict miss
&a[0][4]	0x558FE0A1D350	compulsory miss
&b[4]	0x558FE0A1DC50	compulsory miss
&a[0][5]	0x558FE0A1D358	conflict miss
&b[5]	0x558FE0A1DC58	conflict miss
&a[0][6]	0x558FE0A1D360	compulsory miss
&b[6]	0x558FE0A1DC60	compulsory miss
&a[0][7]	0x558FE0A1D368	conflict miss
&b[7]	0x558FE0A1DC68	conflict miss
&a[0][8]	0x558FE0A1D370	compulsory miss
&b[8]	0x558FE0A1DC70	compulsory miss
&a[0][9]	0x558FE0A1D378	conflict miss
&b[9]	0x558FE0A1DC78	conflict miss

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: `0b1000 0000 0000 0000 0000 0000 0001 0000`



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

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[8192], b[8192], c[8192], d[8192], e[8192];
/* 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]
}
```

How many of the cache misses are **conflict** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

E. 100%

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[8192], b[8192], c[8192], d[8192], e[8192];
/* 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]
}
```

How many of the cache misses are **conflict** misses?

A. 12.5%

B. 66.67%

C. 68.75%

[✓ D. 87.5%]

E. 100%

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

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

E. 100%

intel Core i7

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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	Compulsory Miss	
b[0]	0x20000	0b00100000000000000000	0x20	0x0	Compulsory Miss	
c[0]	0x30000	0b00110000000000000000	0x30	0x0	Compulsory Miss	
d[0]	0x40000	0b01000000000000000000	0x40	0x0	Compulsory Miss	
e[0]	0x50000	0b01010000000000000000	0x50	0x0	Compulsory 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[8192], b[8192], c[8192], d[8192], e[8192];
/* 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	Compulsory Miss	
b[8]	0x20040	0b00100000000001000000	0x20	0x1	Compulsory Miss	
c[8]	0x30040	0b00110000000001000000	0x30	0x1	Compulsory Miss	
d[8]	0x40040	0b01000000000001000000	0x40	0x1	Compulsory Miss	
e[8]	0x50040	0b01010000000001000000	0x50	0x1	Compulsory 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	

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* 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]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

[✓ E. 100%]

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

Basic Hardware Optimization in Improving 3Cs

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

3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
  How many of the following are correct?
  ① Increasing associativity can reduce conflict misses
  ② Increasing associativity can reduce hit time
  ③ Increasing block size can increase the miss penalty
  ④ Increasing block size can reduce compulsory misses
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

A
B
C
D
E
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
How many of the following are correct?
    ① Increasing associativity can reduce conflict misses
    ② Increasing associativity can reduce hit time
    ③ Increasing block size can increase the miss penalty
    ④ Increasing block size can reduce compulsory misses
    A. 0
    B. 1
    C. 2
    [✓ D. 3]
    E. 4

Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You need to fetch more data for each miss

You bring more into the cache when a miss occurs

Announcement

- Reading quiz due next Tuesday as usual
  - We will drop two of your lowest reading quizzes
  - No make up, no extension — they are designed to let you "preview", makes no sense of extension and it's unfair.
- Assignment #2 is up, due next Thursday
  - 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
    - Library outside STL is not allowed — gradescope does not have it
  - 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.

Computer
Science &
Engineering

203

つづく