Memory Hierarchy (2): The A, B, Cs of Caches

Hung-Wei Tseng

von Neumman Architecture

509cbd23

00c2e800

Processor

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

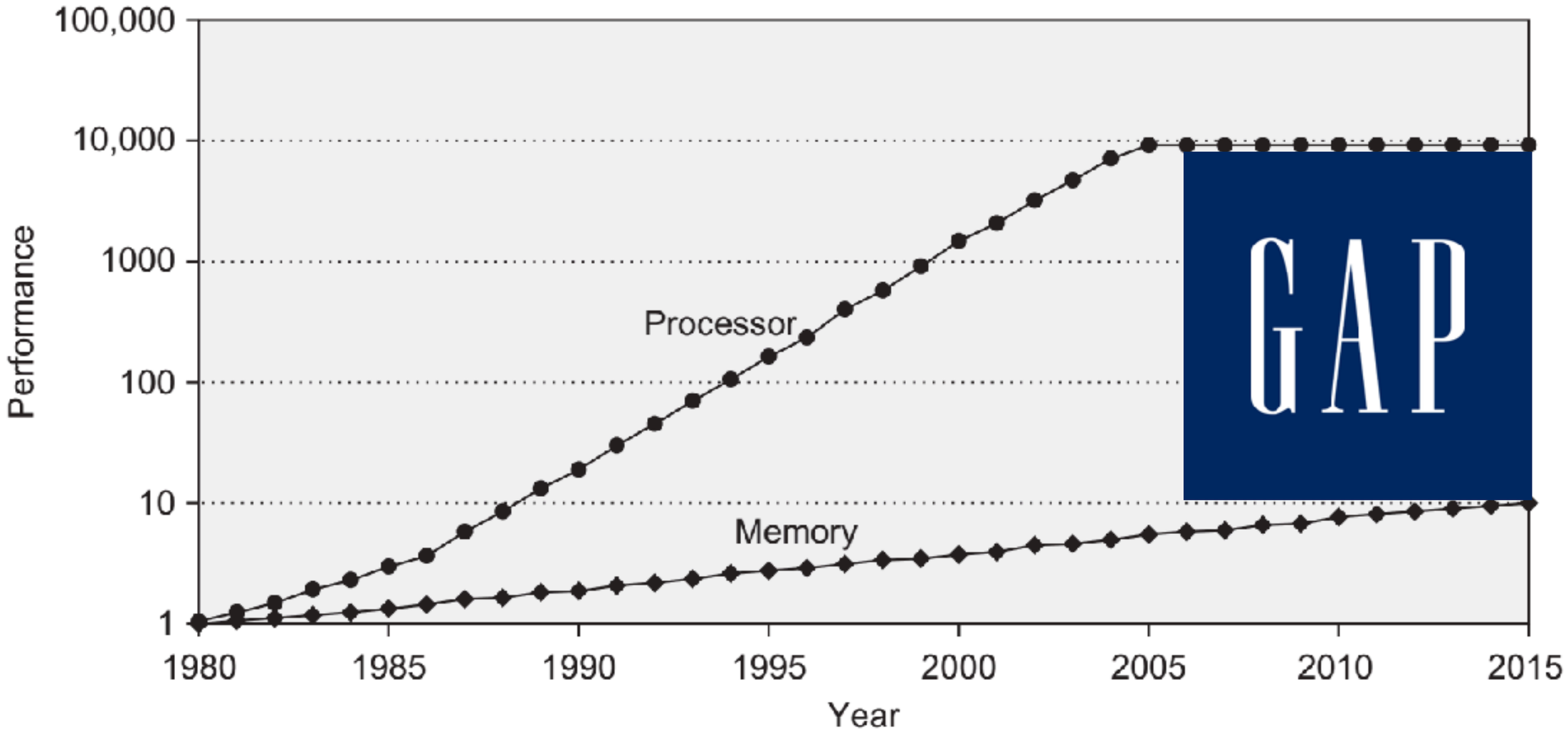
Memory

Program

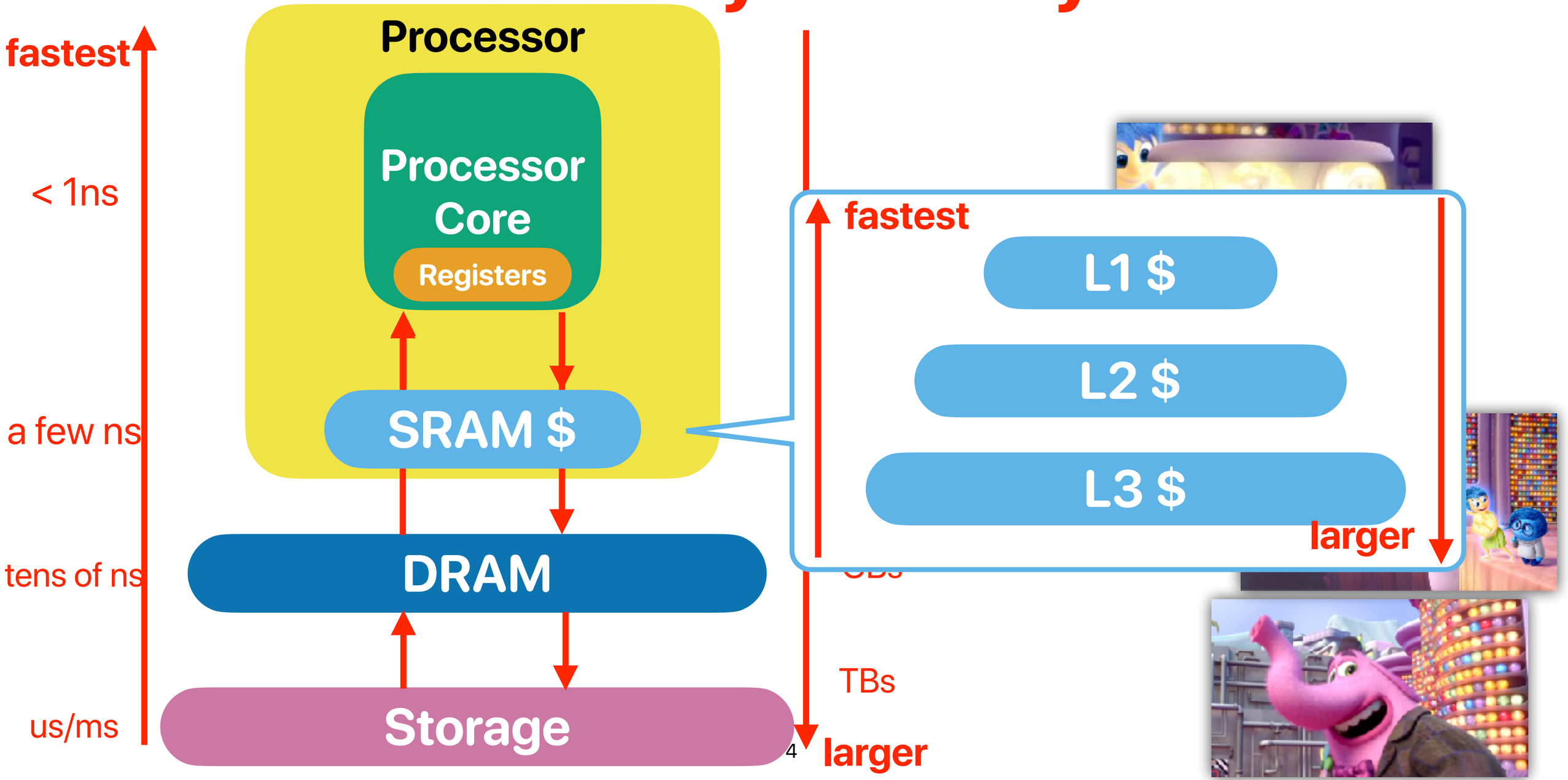
Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

Recap: Performance gap between Processor/Memory



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 data, and then the next one
- 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

Cache design principles — exploit localities

- The cache must be able to get chunks of near-by items every time to exploit spatial locality

  We need to "cache consecutive data elements" every time

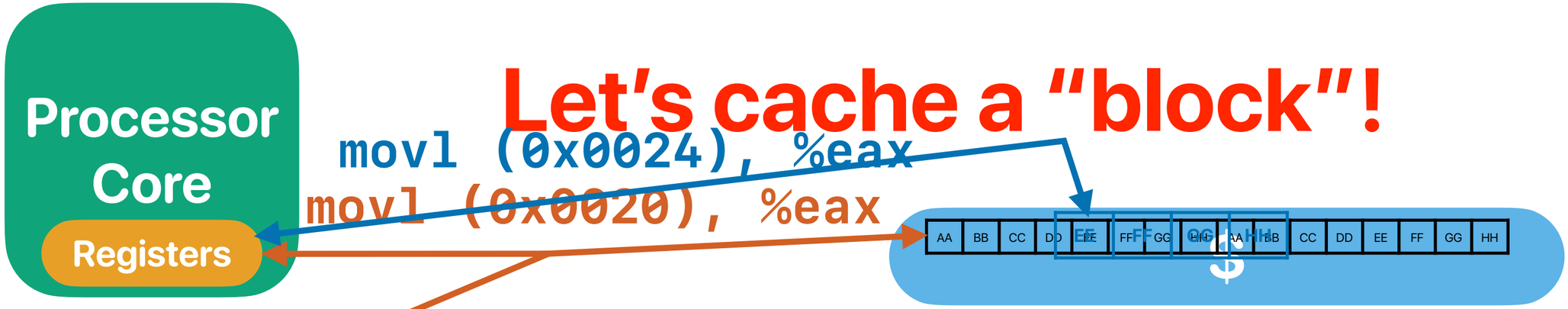
  — the cache should store a "block" of data

- The cache must be able to keep a frequently used block for a while to exploit temporal locality

  We need to store multiple blocks

  — the cache must be able to distinguish blocks

Let's cache a "block"!



Caching a block helps exploit "spatial locality"!

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F

How to tell w|

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

Valid Bit

Dirty Bit

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 CSE1 3:
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:

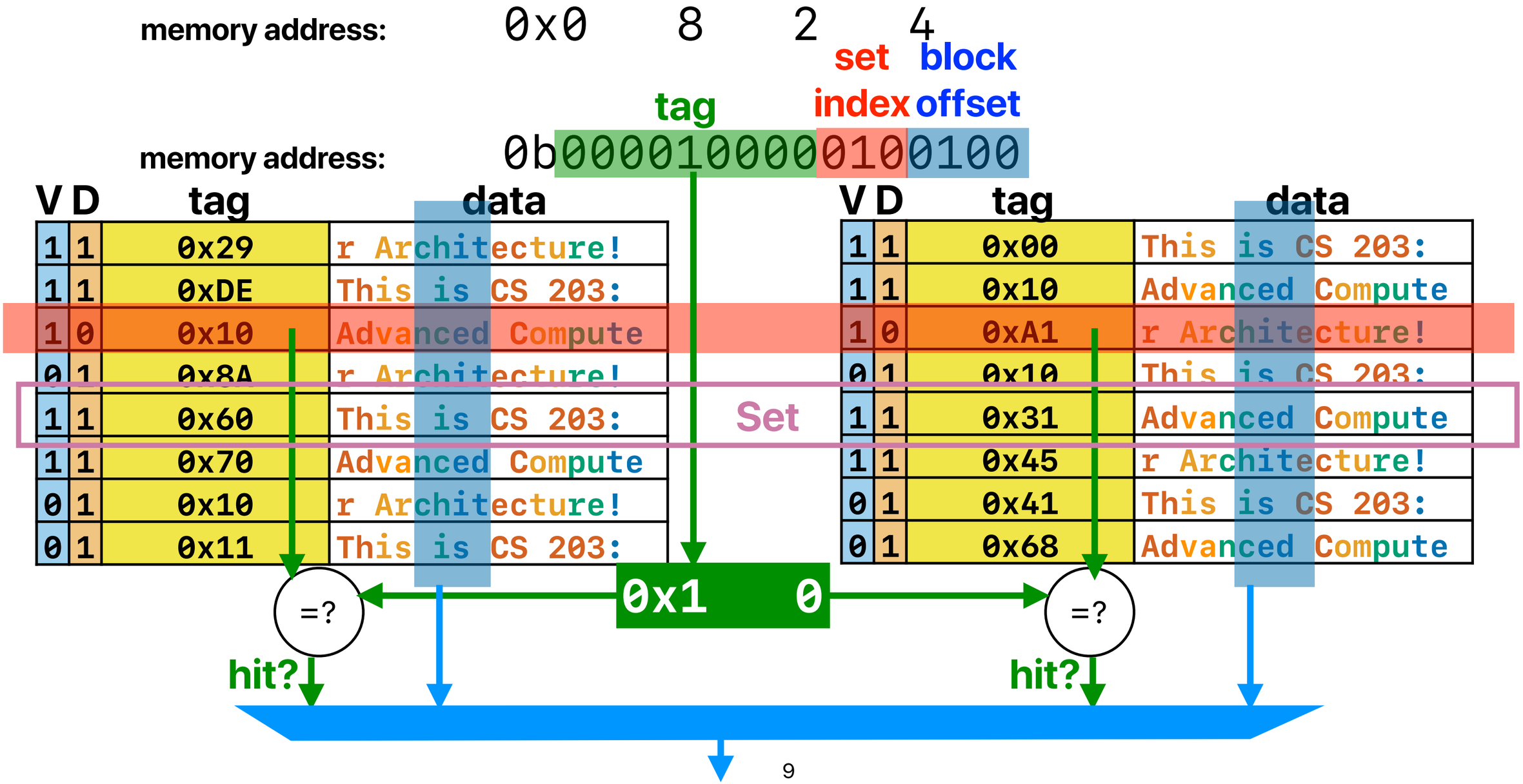
The complexity of search the matching tag—

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

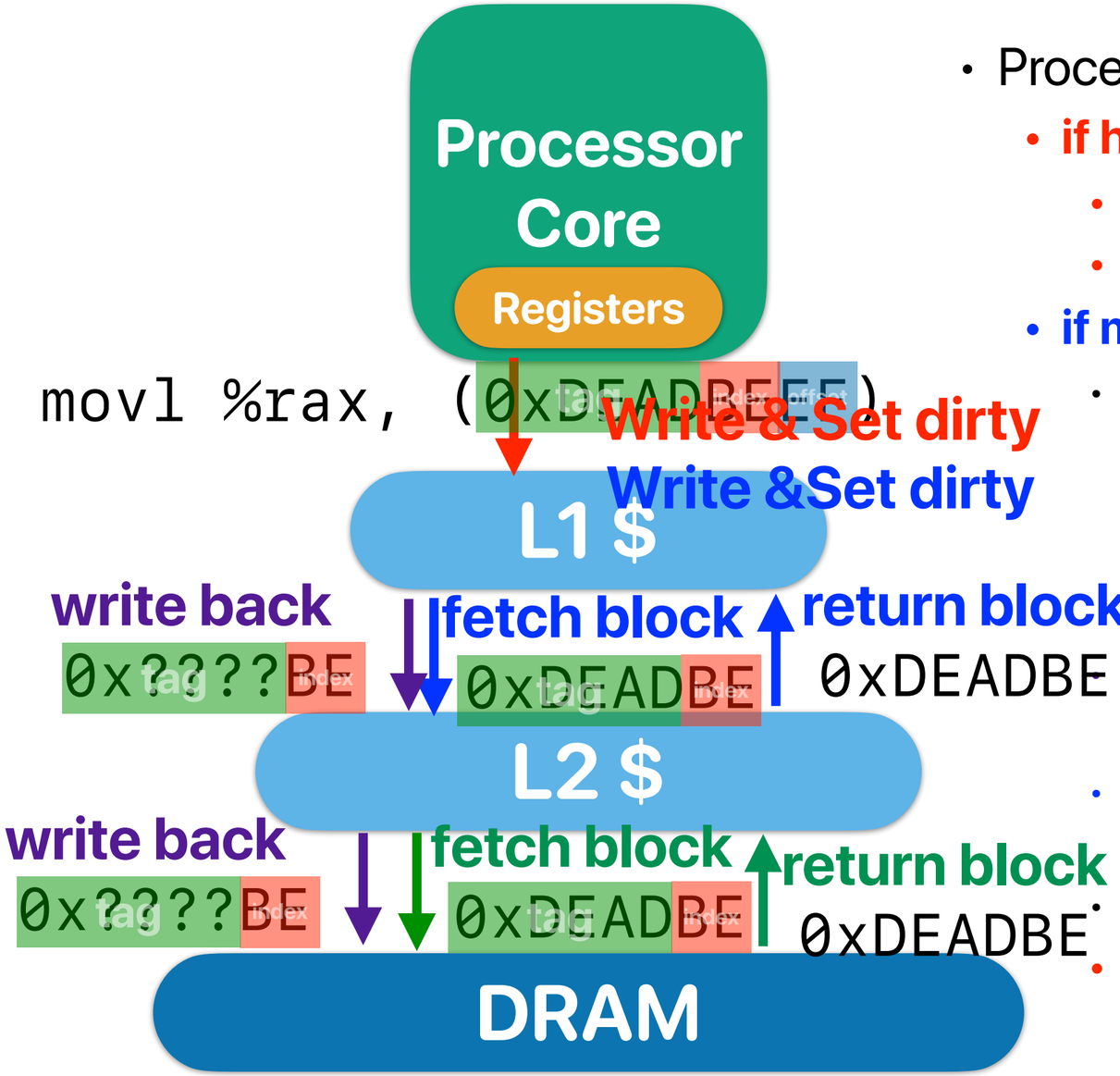
Can we search things faster?

—hash table! $O(1)$

Way-associative cache



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**

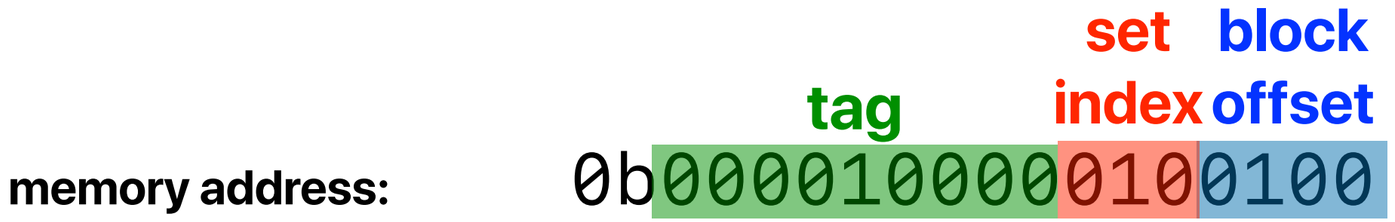
Outline

- The A, B, C s of cache
- Simulating cache behaviors

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



- 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 N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

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

$$C = A \times B \times S$$
$$32 \times 1024 = 4 \times 64 \times S$$
$$S = 128$$
$$Offset = log_2(64) = 6$$
$$Index = log_2(128) = 7$$
$$Tag = 64 - 7 - 6 = 51$$

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

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 48KB, 12-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
    E. All of the above are correct

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 48KB, 12-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

E. All of the above are correct

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

$$48 \times 1024 = 12 \times 64 \times S$$

$$S = 64$$

$$Offset = log_2(64) = 6$$

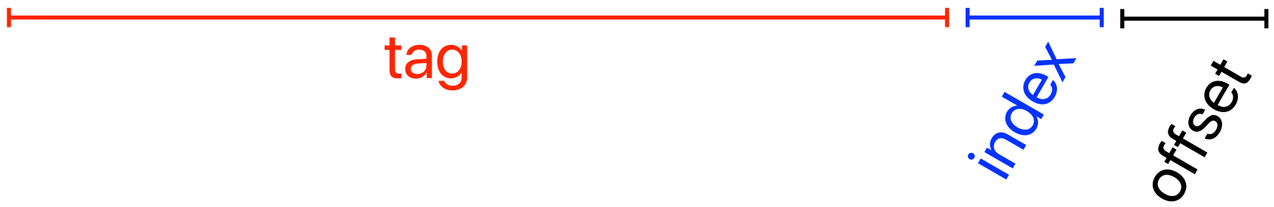
$$Index = log_2(64) = 6$$

$$Tag = 64 - 6 - 6 = 52$$

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`



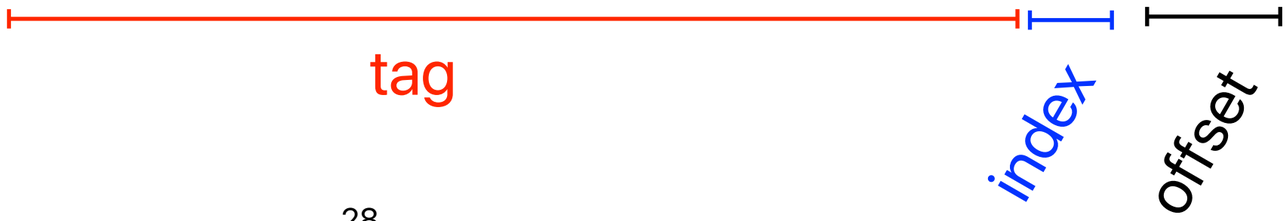
Simulate a direct-mapped cache

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



Simulate a 2-way cache

	V	D	Tag	Data
0	1	0	0b100	r Architecture!
1	1	0	0b100	This is CS 203:
2	0	0		
3	0	0		
4	0	0		
5	0	0		
6	0	0		
7	0	0		

V	D	Tag	Data
0	0		
1	0	0b110	Advanced Compute
0	0		
0	0		
0	0		
0	0		
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!

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	0b0001000000000000000000	0x8	0x0	Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Miss	a[0–7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	Miss	b[0–7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	Miss	c[0–7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	Miss	d[0–7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	Miss	e[0–7]
e[1]	0x50008	0b0101000000000000001000	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%

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%

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	0b0001000000000000000000	0x10	0x0	Miss	
b[0]	0x20000	0b0010000000000000000000	0x20	0x0	Miss	
c[0]	0x30000	0b0011000000000000000000	0x30	0x0	Miss	
d[0]	0x40000	0b0100000000000000000000	0x40	0x0	Miss	
e[0]	0x50000	0b0101000000000000000000	0x50	0x0	Miss	
a[1]	0x10008	0b0001000000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b0010000000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b0011000000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b0100000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b0101000000000000001000	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	0b0001000000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b0010000000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b0011000000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b0100000000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b0101000000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b0001000000000001000000	0x10	0x1	Miss	
b[8]	0x20040	0b0010000000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b0011000000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b0100000000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b0101000000000001000000	0x50	0x1	Miss	
a[9]	0x10048	0b0001000000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b0010000000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b0011000000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b0100000000000001001000	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!

It’s OK and sometimes great that you’re **inspired** by something and **make a better artifact**.
But it’s not OK and **illegal** if plagiarize something.

Announcement

- Reading quiz #4 due **next Tuesday** before the lecture
  - Please do read the textbook before/while taking the quiz
  - We take the "average"
- Assignment #3 is up. Due in next Thursday (in 9 days)
- Plagiarism:
  - Please review the course website and the slide from the first lecture
    - You have to give "credits" to who you have consulted
    - Please review ACM's policy on the use of generative AI
      - https://www.acm.org/publications/policies/frequently-asked-questions
      - https://www.acm.org/binaries/content/assets/public-policy/ustpc-approved-generative-ai-principles
  - I am still waiting for a group to give me a response by tonight
    - If we figured out by today, programming assignment will be 0
    - If I don't hear from the plagiarism group — the group don't need to turn in programming assignments anymore and the incident will be reported to the school.

Computer
Science &
Engineering

203

つづく