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

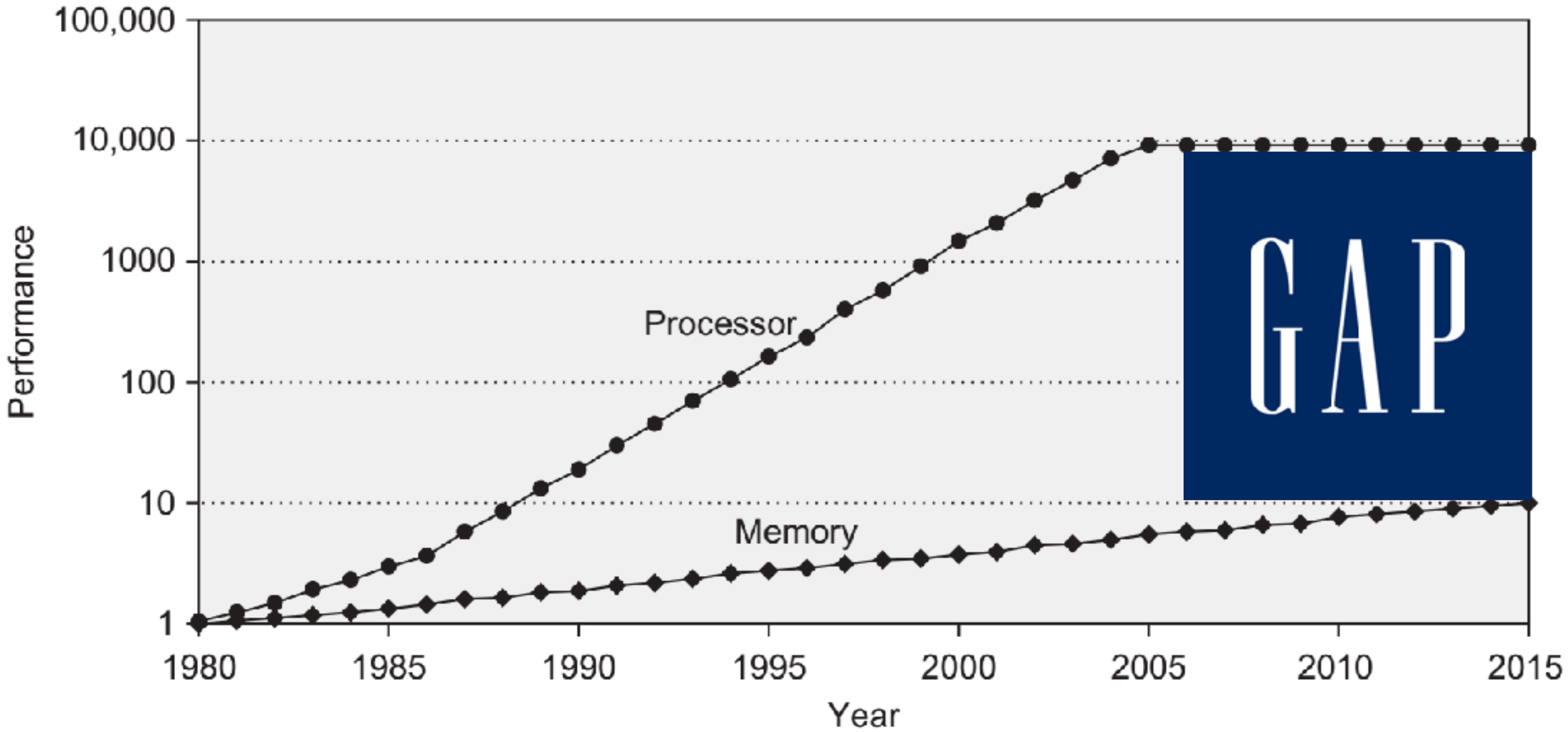
Memory

Program

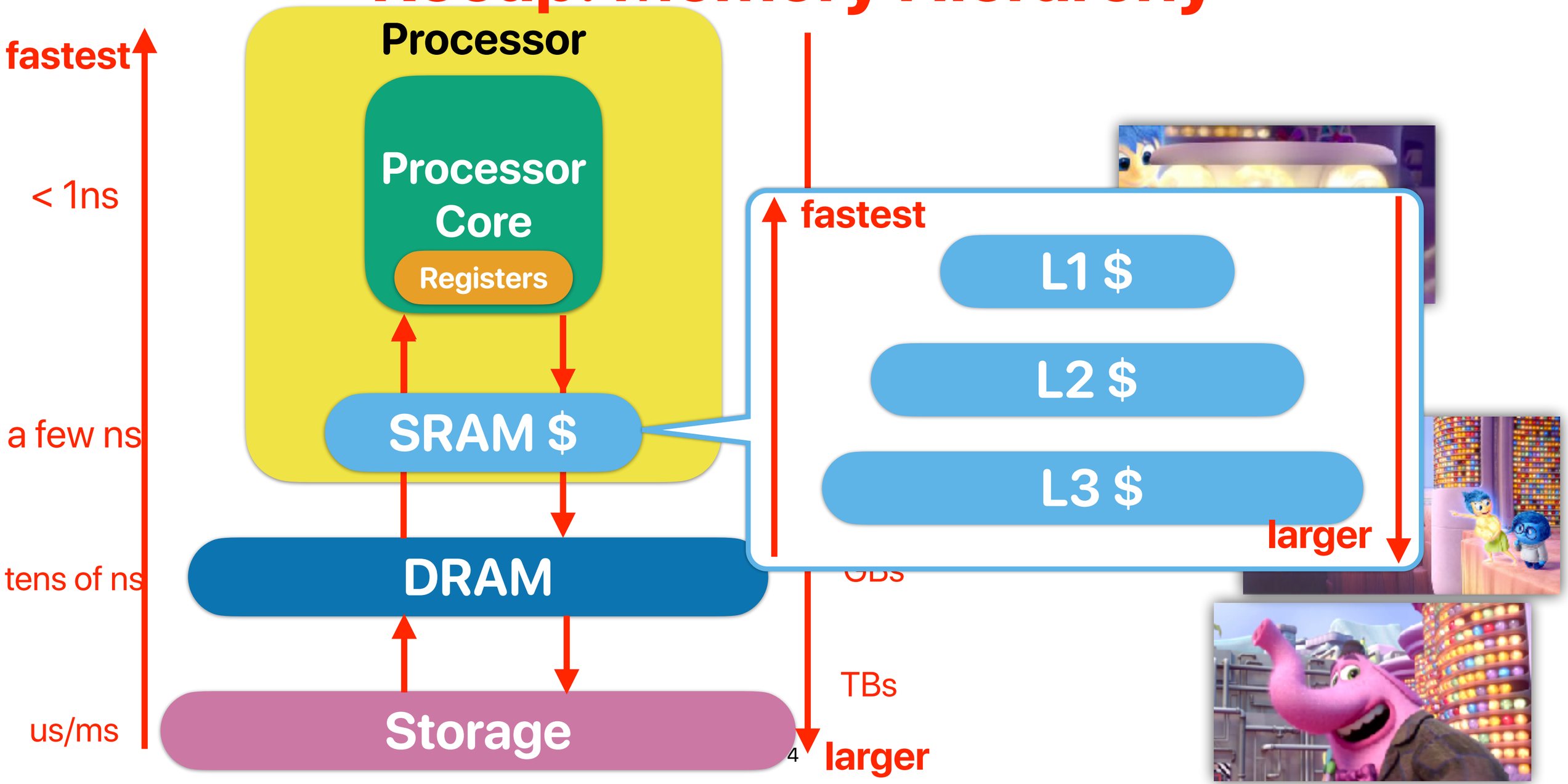
Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

Recap: Performance gap between Processor/Memory



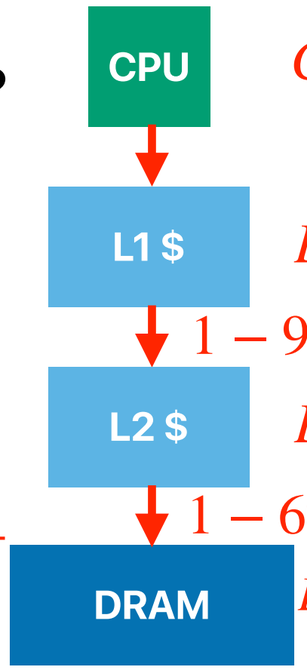
Recap: Memory Hierarchy



How can a deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got a 2-level SRAM caches with
  - it's 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5 ns and can capture 60% of the desired data/instructions

  We also got an SRAM cache with latency of just at 0.5 ns and can capture 90% of the desired data/instructions.
  what's the average CPI (pick the closest one)?
  A. 6
  B. 8
  C. 10
  D. 12
  E. 67



$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

$$Each\ L1_{\$}\ access = \frac{0.5}{0.25} = 2\ cycles$$

$$Each\ L2_{\$}\ access = \frac{5}{0.25} = 20\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55] +$$
$$20\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55)]$$
$$= 8.44\ cycles$$

Code also has 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;
}
```

```
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++;
}
```

keep going to the next instruction — spatial locality

repeat many times — temporal locality!

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

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b000010000 010 0100

tag | index | offset

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CSE142:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CSE142:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CSE142:

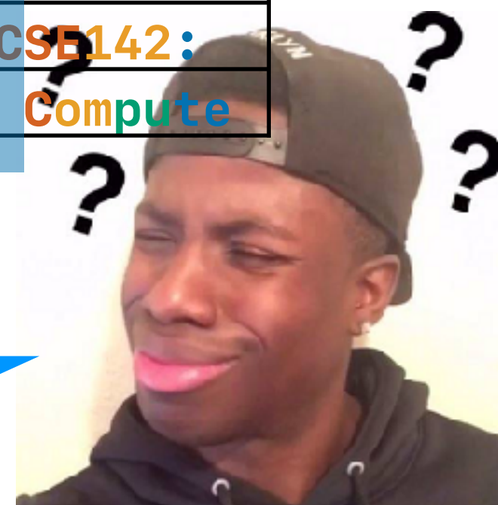
Set

V	D	tag	data
1	1	0x00	This is CSE142:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CSE142:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CSE142:
0	1	0x68	Advanced Compute

0x1 0

=? =?

hit? hit?



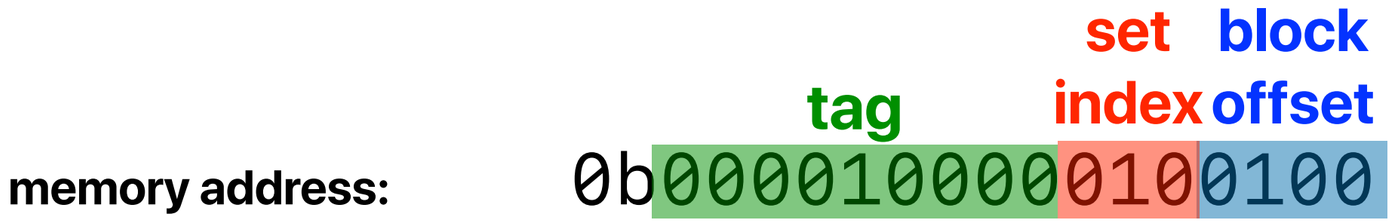
Outline

- The A, B, C, S of your cache
- Taxonomy/reasons of cache misses

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 = ABS

32KB = 4 * 64 * S

S = 128

offset = lg(64) = 6 bits

index = lg(128) = 7 bits

tag = 64 - lg(64) - lg(128) = 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
    E. All of the above are correct

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

E. All of the above are correct

C = ABS

32KB = 8 * 64 * S

S = 64

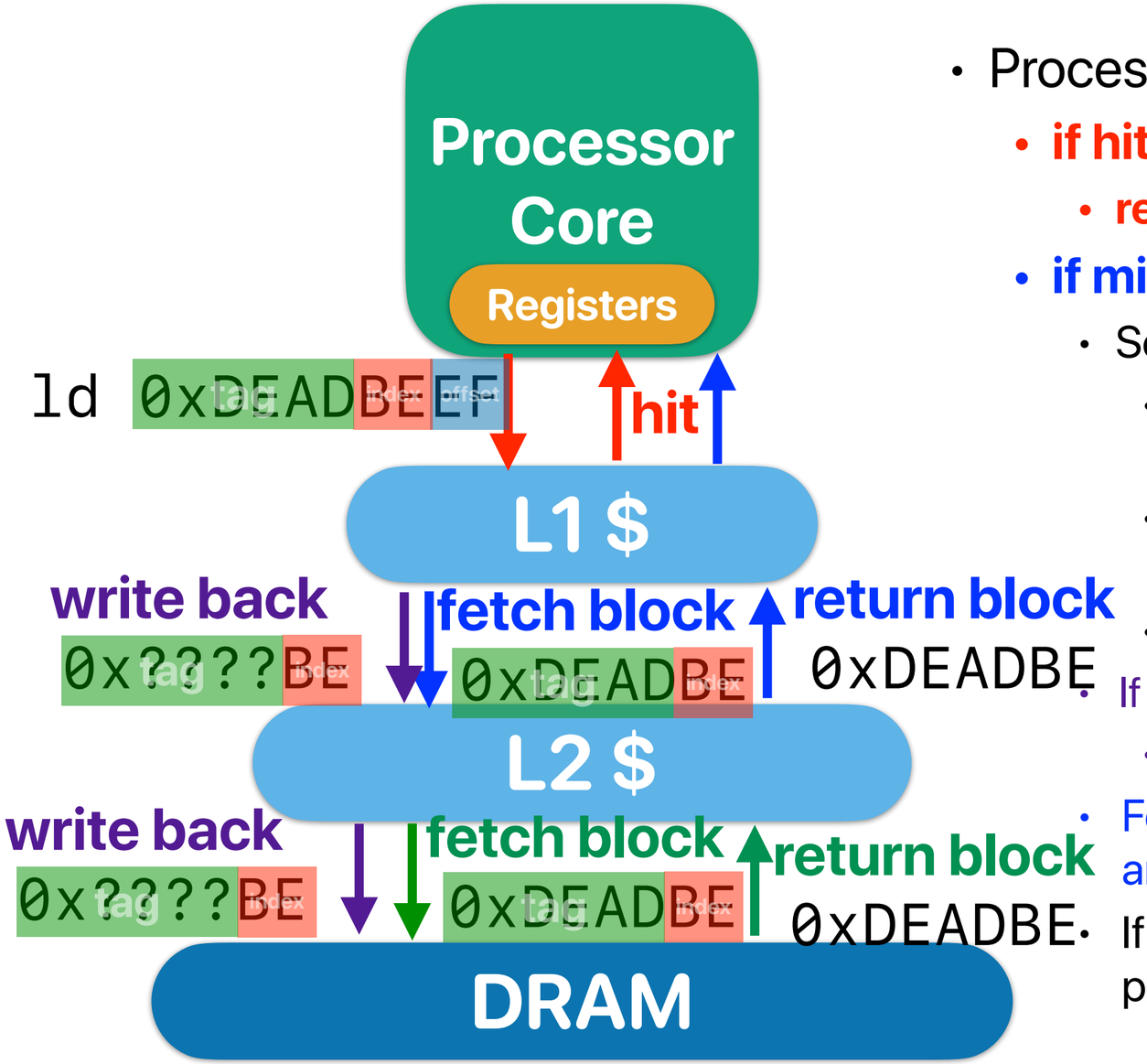
offset = lg(64) = 6 bits

index = lg(64) = 6 bits

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

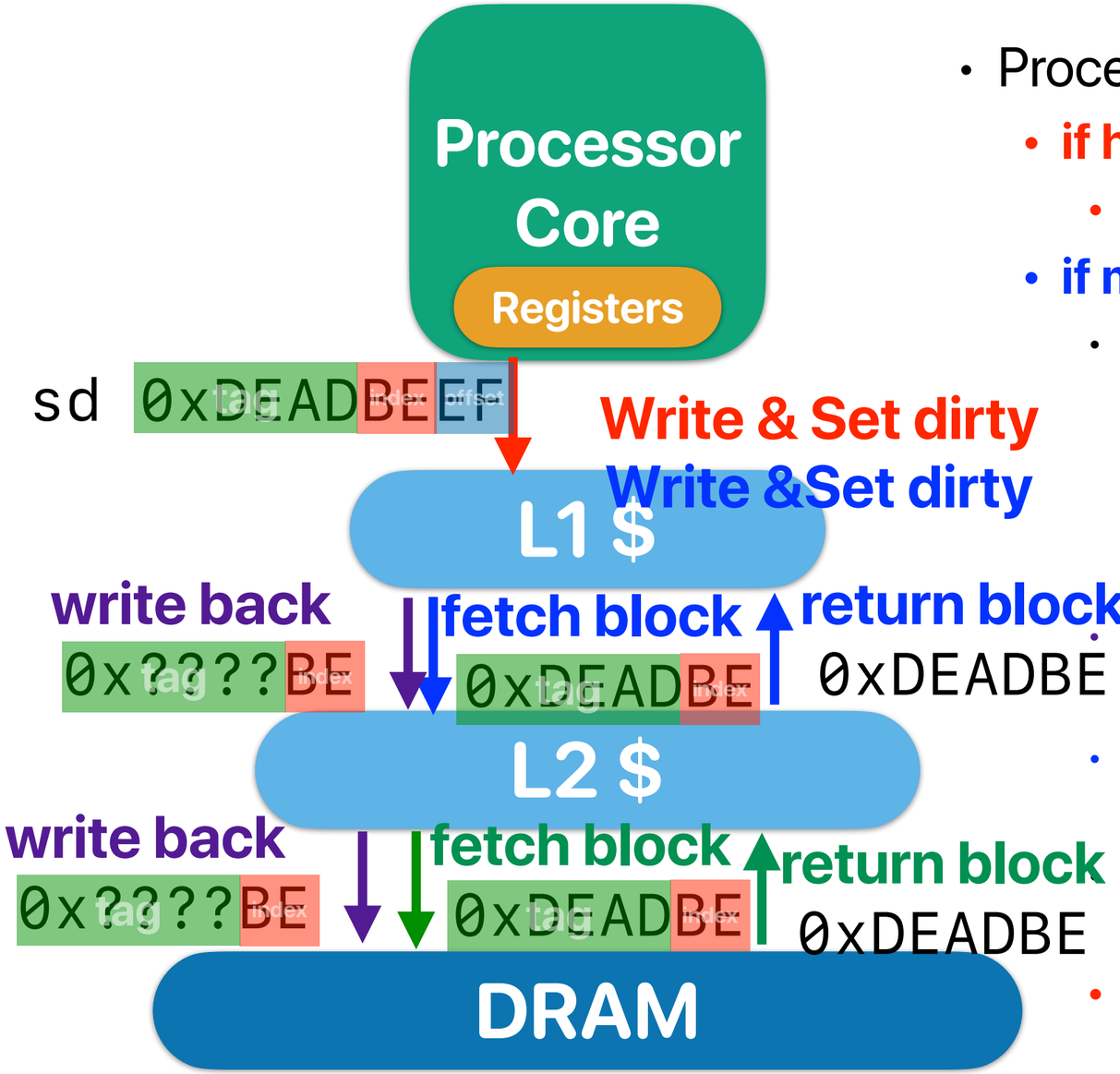
Put everything all together: How cache interacts with CPU

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

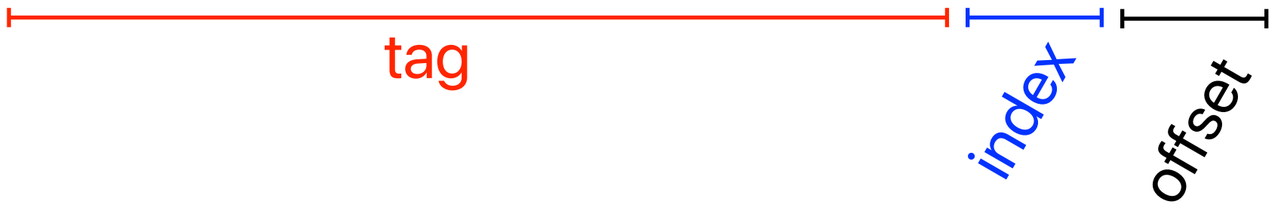


- Processor sends load request to L1-$
  - **if hit**
    - **return data — set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - 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 64 - (4 + 4) = 56 bits
    - For example: `0b1000 0000 0000 0000 0000 0000 1000 0000`



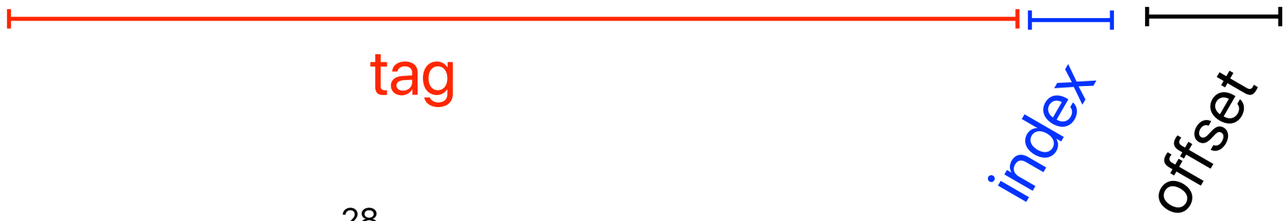
Simulate a direct-mapped cache

	V	D	Tag	Data
0	1	0	0b10	r Architecture!
1	1	0	0b10	This is CSE142:
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 64 - (3 + 4) = 57 bits
    - For example: `0b1000 0000 0000 0000 0000 0000 0001 0000`



Simulate a 2-way cache

	V	D	Tag	Data
0	1	0	0b10	r Architecture!
1	1	0	0b10	This is CSE142:
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	0b11	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%

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[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 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[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	0b0001000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b0010000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b0011000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b0100000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b0101000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b0001000000001000000	0x10	0x1	Miss	
b[8]	0x20040	0b0010000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b0011000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b0100000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b0101000000001000000	0x50	0x1	Miss	
a[9]	0x10048	0b0001000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b0010000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b0011000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b0100000000001001000	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 due tomorrow before the lecture
  - Please do read the textbook before/while taking the quiz
  - We take the "average"
- Assignment #2 due the upcoming Sunday
  - 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
  - Check the website tonight for the template and questions
- No lab lecture today (only CSE142L lecture on Thursdays)
- Midterm next Monday — will release a midterm review online only lecture by Friday

Computer
Science &
Engineering

142

つづく