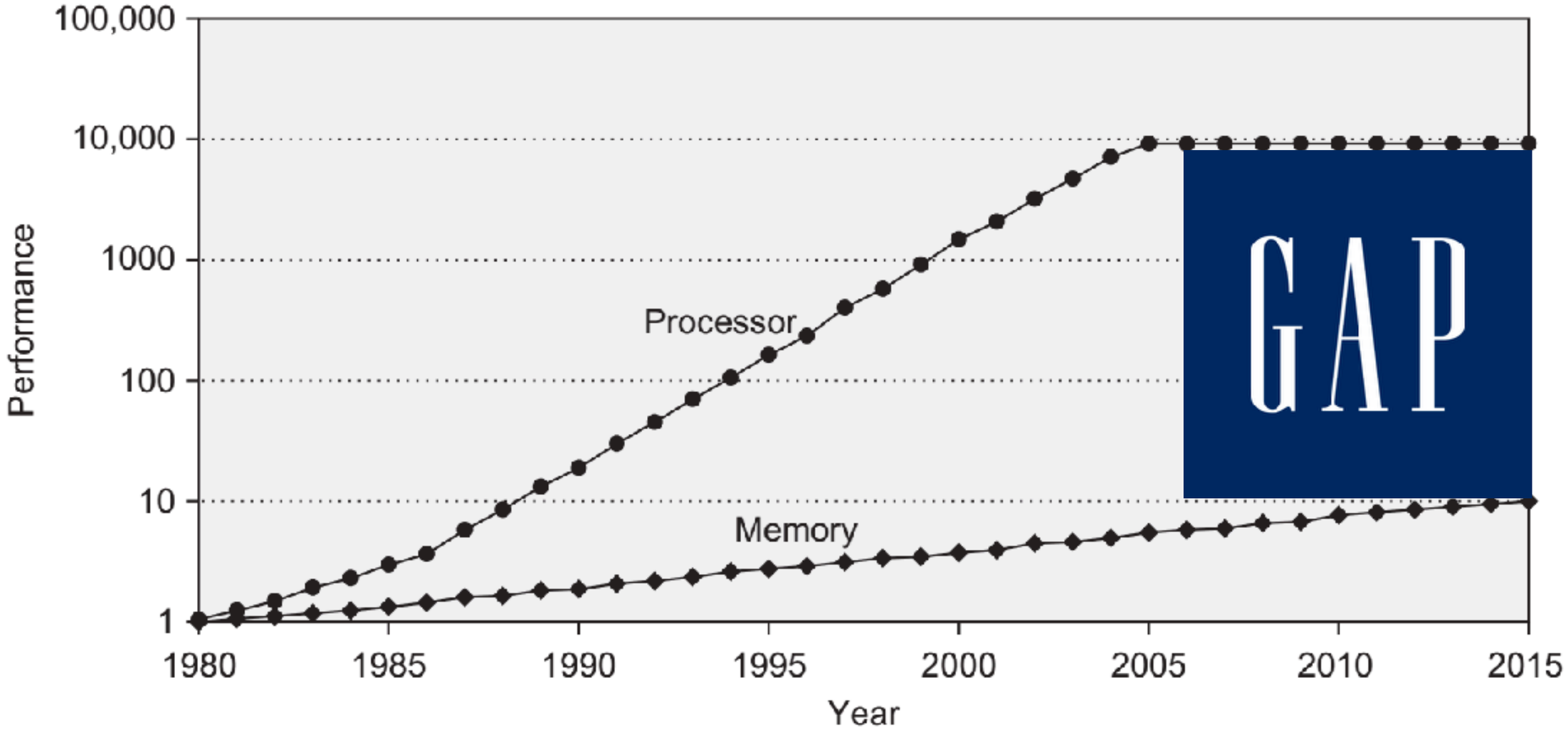
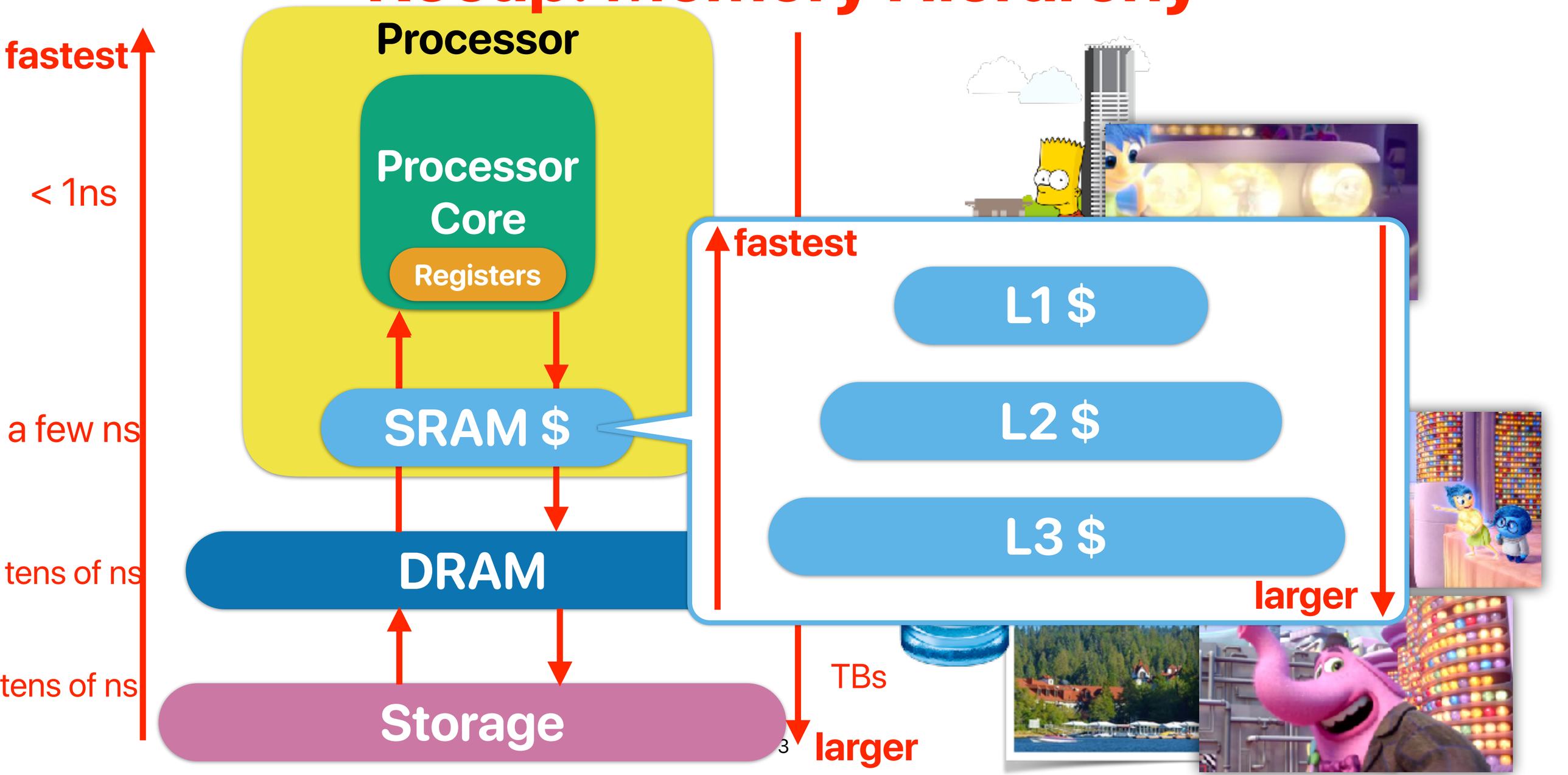
Memory Hierarchy (III)

Hung-Wei Tseng

Recap: Performance gap between Processor/Memory



Recap: Memory Hierarchy

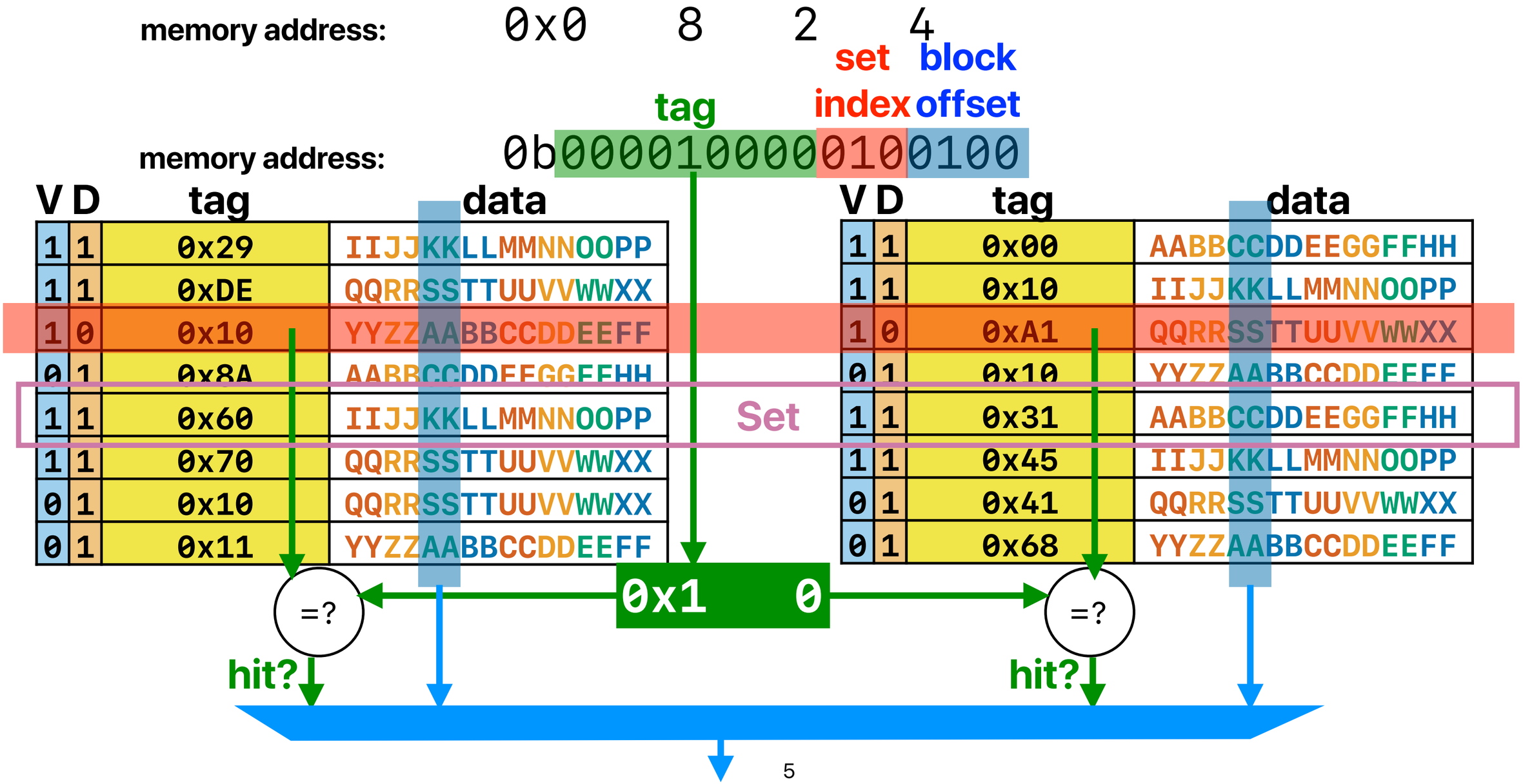


Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - 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 very small amount of data/instructions within a given window

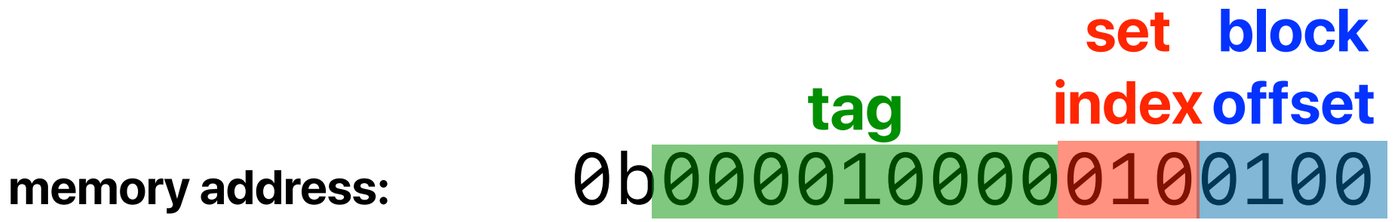
Recap: Way-associative cache



C = ABS

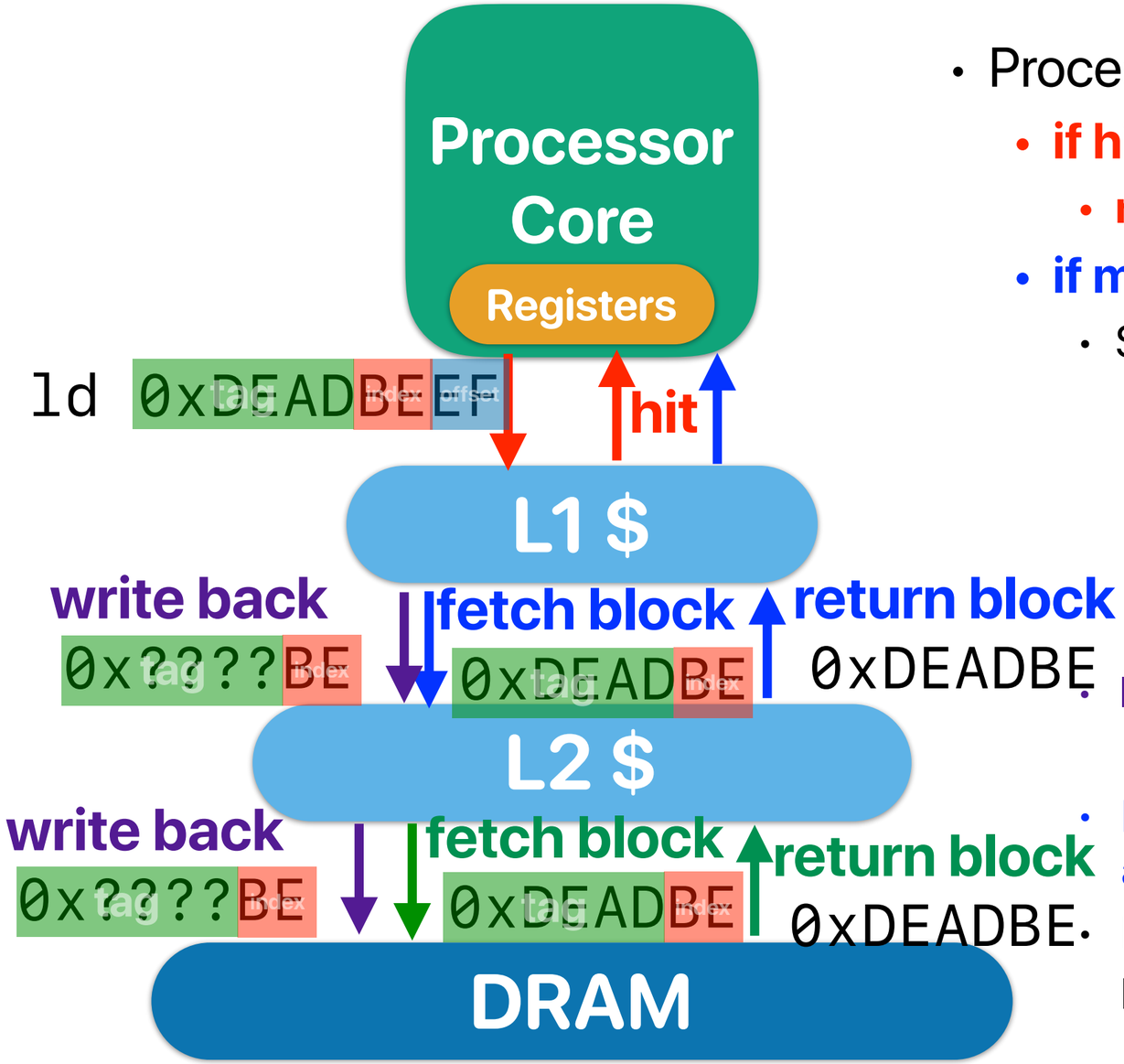
- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Cacheline)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache

Corollary of C = ABS



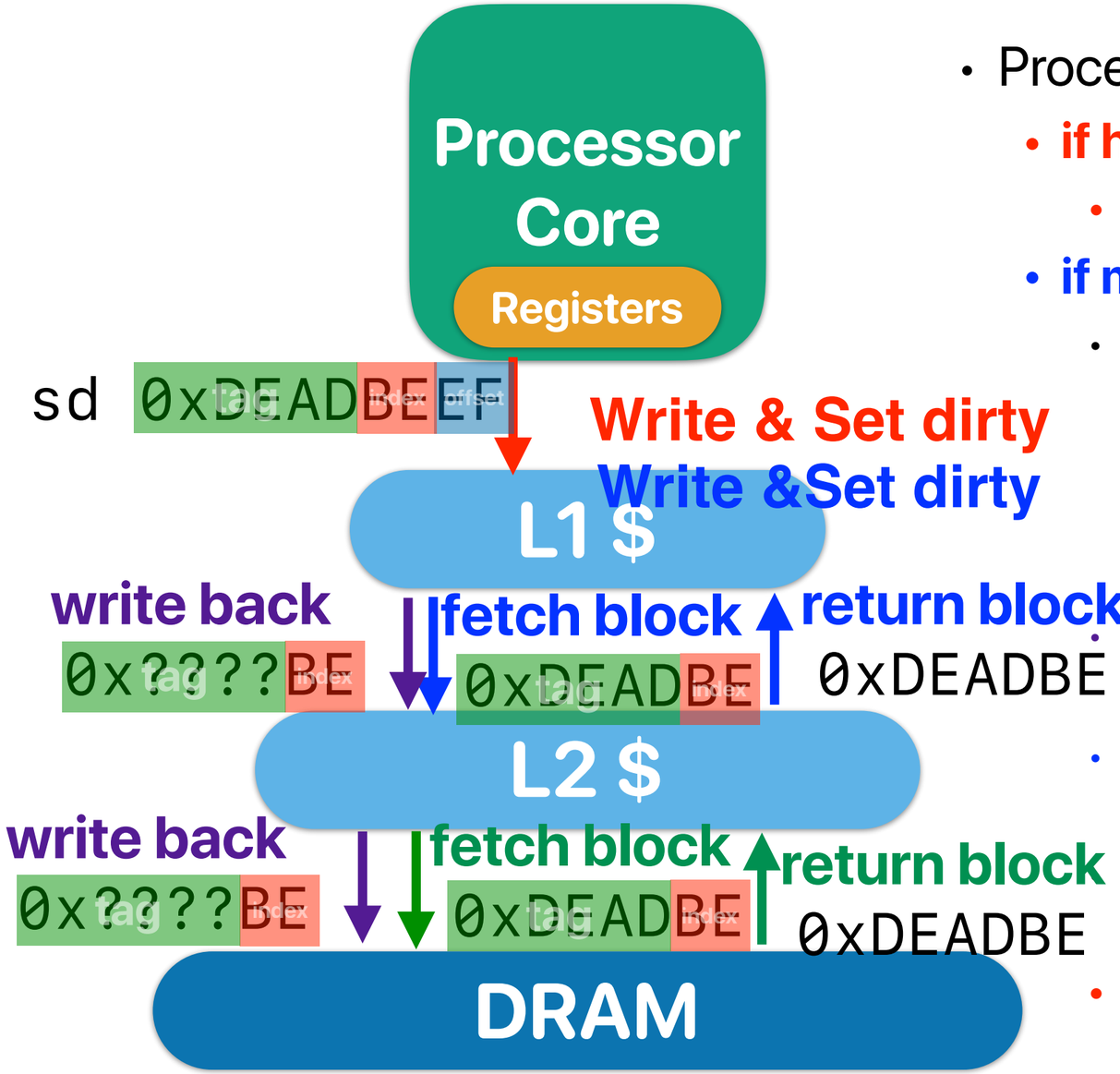
- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is 32 bits for 32-bit machine
- (address / block_size) % S = set index

Recap: What happens on a read



- 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

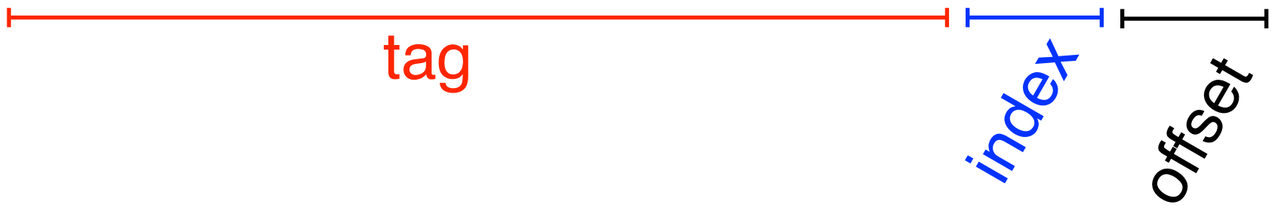
Recap: What happens on a write



- 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**

Recap: 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



Recap: Simulate a direct-mapped cache

	V	D	Tag	Data
0	1	0	0b10	
1	1	0	0b10	
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!

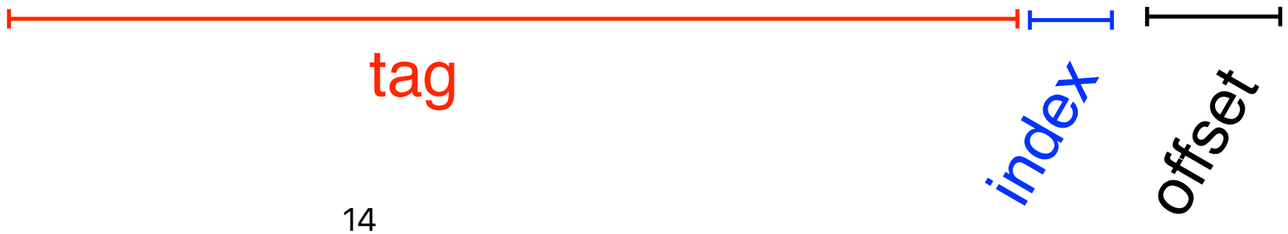
Outline

- Cache simulation
- Causes of cache misses
- Techniques in improving cache performance

Simulate the cache! (cont.)

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	1	0	0b10		0	0		
1	1	0	0b10		1	0	0b11	
2	0	0			0	0		
3	0	0			0	0		
4	0	0			0	0		
5	0	0			0	0		
6	0	0			0	0		
7	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!

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

C = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits

AMD Phenom II

100% miss rate!

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

	address in hex	address in binary (tag / index / offset)	tag	index	hit? miss?
load a[0]	0x20000	0b10 0000 0000 0000 0000	0x4	0	miss
load b[0]	0x30000	0b11 0000 0000 0000 0000	0x6	0	miss
store c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	miss, evict 0x4
load a[1]	0x20004	0b10 0000 0000 0000 0100	0x4	0	miss, evict 0x6
load b[1]	0x30004	0b11 0000 0000 0000 0100	0x6	0	miss, evict 0x2
store c[1]	0x10004	0b01 0000 0000 0000 0100	0x2	0	miss, evict 0x4
⋮	⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0b10 0000 0000 0011 1100	0x4	0	miss, evict 0x6
load b[15]	0x3003C	0b11 0000 0000 0011 1100	0x6	0	miss, evict 0x2
store c[15]	0x1003C	0b01 0000 0000 0011 1100	0x2	0	miss, evict 0x4
load a[16]	0x20040	0b10 0000 0000 0100 0000	0x4	1	miss
load b[16]	0x30040	0b11 0000 0000 0100 0000	0x6	1	miss
store c[16]	0x10040	0b01 0000 0000 0100 0000	0x2	1	miss, evict 0x4

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

C = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits

intel Core i7

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

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code?

A. 6.25%
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

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

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	?
load a[0]	0x20000	0x20	0	miss
load b[0]	0x30000	0x30	0	miss
store c[0]	0x10000	0x10	0	miss
load a[1]	0x20004	0x20	0	hit
load b[1]	0x30004	0x30	0	hit
store c[1]	0x10004	0x10	0	hit
⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0x20	0	hit
load b[15]	0x3003C	0x30	0	hit
store c[15]	0x1003C	0x10	0	hit
load a[16]	0x20040	0x20	1	miss
load b[16]	0x30040	0x30	1	miss
store c[16]	0x1003C	0x10	1	miss

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)

intel Core i7

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

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code?

A. 6.25%
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

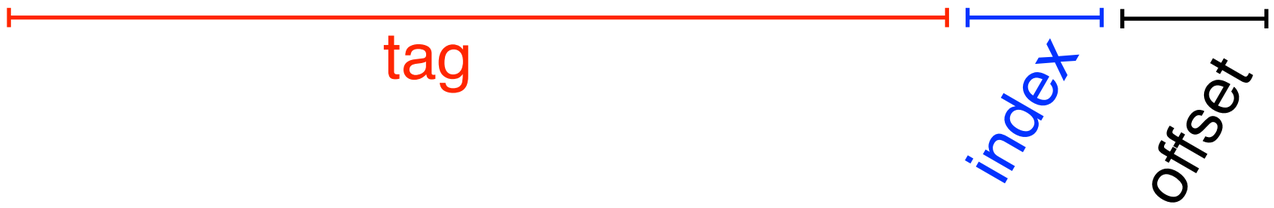
Cause of cache misses

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`



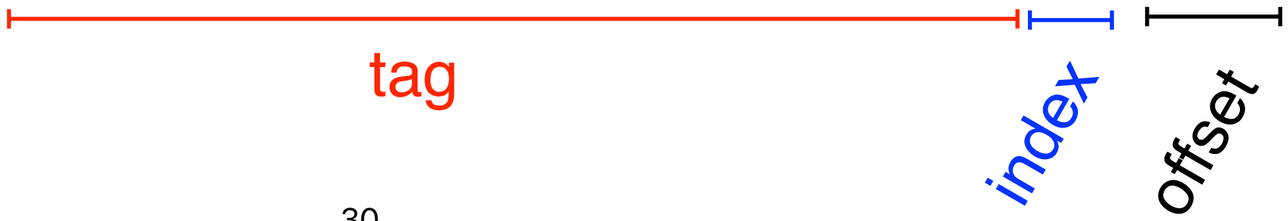
Simulate a direct-mapped cache

	V	D	Tag	Data
0	1	0	0b10	
1	1	0	0b10	
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	compulsory miss
0b10	0000	1000	hit!
0b10	0001	0000	compulsory miss
0b10	0001	0100	hit!
0b11	0001	0000	compulsory miss
0b10	0000	0000	hit!
0b10	0000	1000	hit!
0b10	0001	0000	conflict 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	V	D	Tag	Data
0	1	0	0b10		0	0		
1	1	0	0b10		1	0	0b11	
2	0	0			0	0		
3	0	0			0	0		
4	0	0			0	0		
5	0	0			0	0		
6	0	0			0	0		
7	0	0			0	0		

tag	index		
0b10	0000	0000	compulsory miss
0b10	0000	1000	hit!
0b10	0001	0000	compulsory miss
0b10	0001	0100	hit!
0b11	0001	0000	compulsory miss
0b10	0000	0000	hit!
0b10	0000	1000	hit!
0b10	0001	0000	hit
0b10	0001	0100	hit!

AMD Phenom II

100% miss rate!

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

	address in hex	address in binary (tag / index / offset)	tag	index	hit? miss?
load a[0]	0x20000	0b10 0000 0000 0000 0000	0x4	0	compulsory miss
load b[0]	0x30000	0b11 0000 0000 0000 0000	0x6	0	compulsory miss
store c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	compulsory miss, evict
load a[1]	0x20004	0b10 0000 0000 0000 0100	0x4	0	conflict miss, evict 0x6
load b[1]	0x30004	0b11 0000 0000 0000 0100	0x6	0	conflict miss, evict 0x2
store c[1]	0x10004	0b01 0000 0000 0000 0100	0x2	0	conflict miss, evict 0x4
⋮	⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0b10 0000 0000 0011 1100	0x4	0	miss, evict 0x6
load b[15]	0x3003C	0b11 0000 0000 0011 1100	0x6	0	miss, evict 0x2
store c[15]	0x1003C	0b01 0000 0000 0011 1100	0x2	0	miss, evict 0x4
load a[16]	0x20040	0b10 0000 0000 0100 0000	0x4	1	compulsory miss
load b[16]	0x30040	0b11 0000 0000 0100 0000	0x6	1	compulsory miss
store c[16]	0x10040	0b01 0000 0000 0100 0000	0x2	1	compulsory miss, evict

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

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

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%

C = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits

intel Core i7

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

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	?
load a[0]	0x20000	0x20	0	compulsory miss
load b[0]	0x30000	0x30	0	compulsory miss
store c[0]	0x10000	0x10	0	compulsory miss
load a[1]	0x20004	0x20	0	hit
load b[1]	0x30004	0x30	0	hit
store c[1]	0x10004	0x10	0	hit
⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0x20	0	hit
load b[15]	0x3003C	0x30	0	hit
store c[15]	0x1003C	0x10	0	hit
load a[16]	0x20040	0x20	1	compulsory miss
load b[16]	0x30040	0x30	1	compulsory miss
store c[16]	0x1003C	0x10	1	compulsory miss

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)

intel Core i7

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

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

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

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
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

Improving 3Cs

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

Improvement of 3Cs

- 3Cs and A, B, C of caches
  - Compulsory miss
    - Increase B: increase miss penalty (more data must be fetched from lower hierarchy)
  - Capacity miss
    - Increase C: increase cost, access time, power
  - Conflict miss
    - Increase A: increase access time and power
- Or modify the memory access pattern of your program!

Programming and memory performance

Data layout

User-defined data structure

- Programming languages allow user to define their own data types
- In C, programmers can use `struct` to define new data structure

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

How many bytes each "struct node" will occupy?

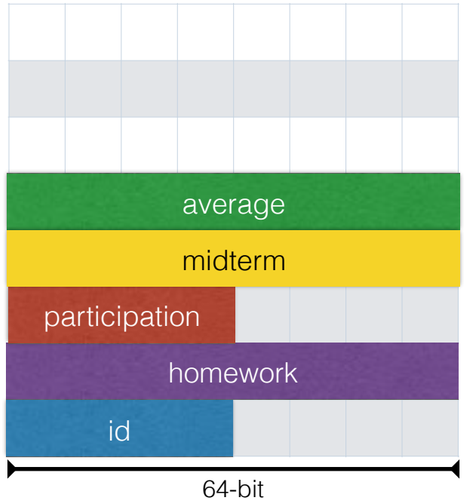
Memory addressing/alignment

- Almost every popular ISA architecture uses "byte-addressing" to access memory locations
- Instructions generally work faster when the given memory address is aligned
  - Aligned — if an instruction accesses an object of size n at address X, the access is **aligned** if $X \bmod n = 0$.
  - Some architecture/processor does not support aligned access at all
  - Therefore, compilers only allocate objects on "aligned" address

The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```



What's the output of `printf("%lu\n",sizeof(struct student))`?

A. 20
B. 28
C. 32
D. 36
E. 40

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;    // get bits for floating value
    int i = *(int*)&x;       // gives initial guess y0
    i = 0x5f3759df - (i>>1); // convert bits back to float
    x = *(float*)&i;         // Newton step, repeating
    x = x*(1.5f - xhalf*x*x);// increases accuracy
    return x;
}
```

Some quick testing in Visual C++.NET [2] showed the code above to be roughly 4 times faster than the naive (float)(1.0/sqrt(x)),

Good/bad practices in coding interviews

Upon receiving the question

	Things
✅	Repeat the question back at the interviewer.
✅	Clarify any assumptions you made subconsciously. Many questions are under-specified on purpose. E.g. a tree-like diagram could very well be a graph that allows for cycles and a naive recursive solution would not work.
✅	Clarify input format and range. Ask whether input can be assumed to be well-formed and non-null.
✅	Work through a small example to ensure you understood the question.
✅	Explain a high level approach even if it is a brute force one.
✅	Improve upon the approach and optimize. Reduce duplicated work and cache repeated computations.
✅	Think carefully, then state and explain the time and space complexity of your approaches.
✅	If stuck, think about related problems you have seen before and how they were solved. Check out the tips in this section.
❌	Ignore information given to you. Every piece is important.
❌	Jump into coding straightaway.
❌	Start coding without interviewer's green light.
❌	Appear too unsure about your approach or analysis.

Excerpted from https://yangshun.github.io/tech-interview-handbook/cheatsheet/

Good practices in assignments

- Clarify the question if you have doubts — using piazza
- Clarify any assumption that you made for the given question — write it down on your answer
- Explain a high-level approach/overview of your solution — write it down on you answer

Announcement

- Make up lecture **7pm next Monday (10/28) @ WCH 143** — will be midterm review
  - Will use 70 minutes to highlight the important concepts for midterm
  - Will present a sample midterm
- Assignments
  - We will drop the lowest assignment grade — Josep's talk summary treated as a bonus one (technically we drop two)
  - Please make sure you have CLEAR description on what formula you're applying and state why this is a good one if necessary.
  - If you have any assumption on numbers/terms, please write them down. You won't get any grade if you did not do so.