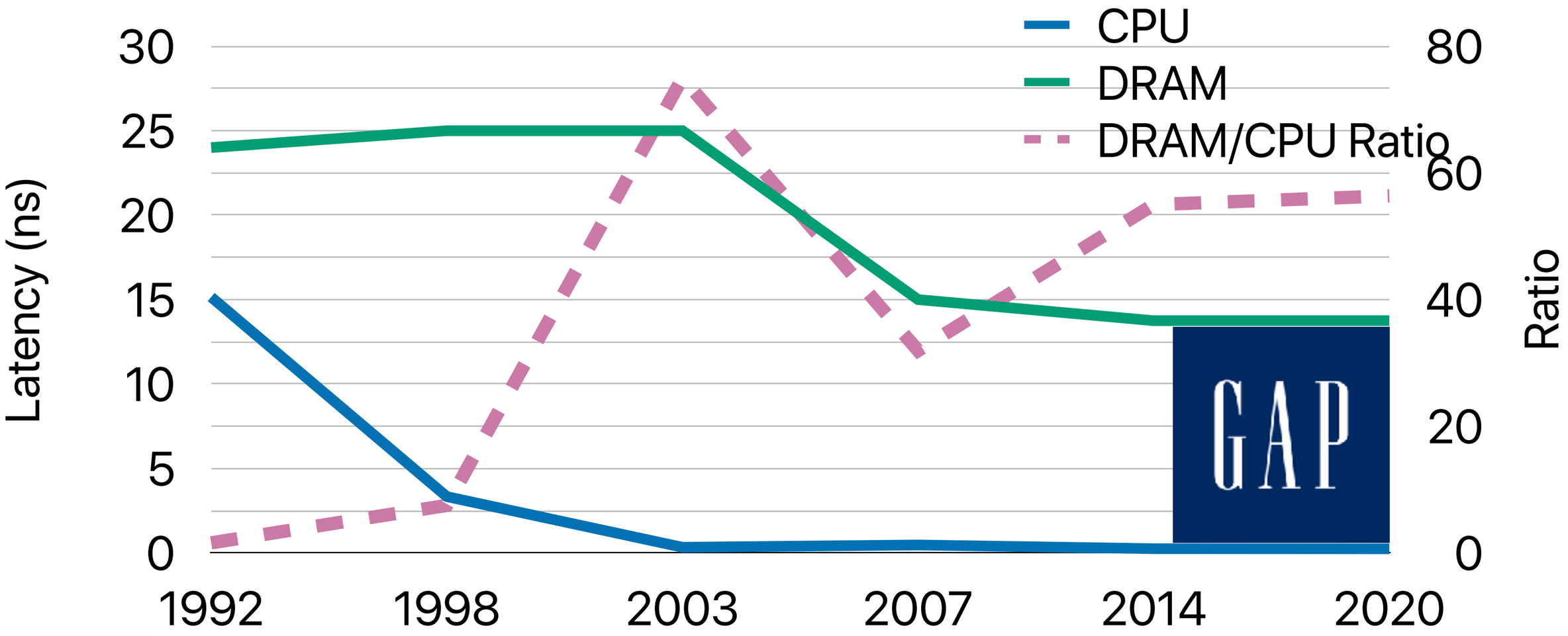
Memory Hierarchy (2): The A, B, Cs of Caches

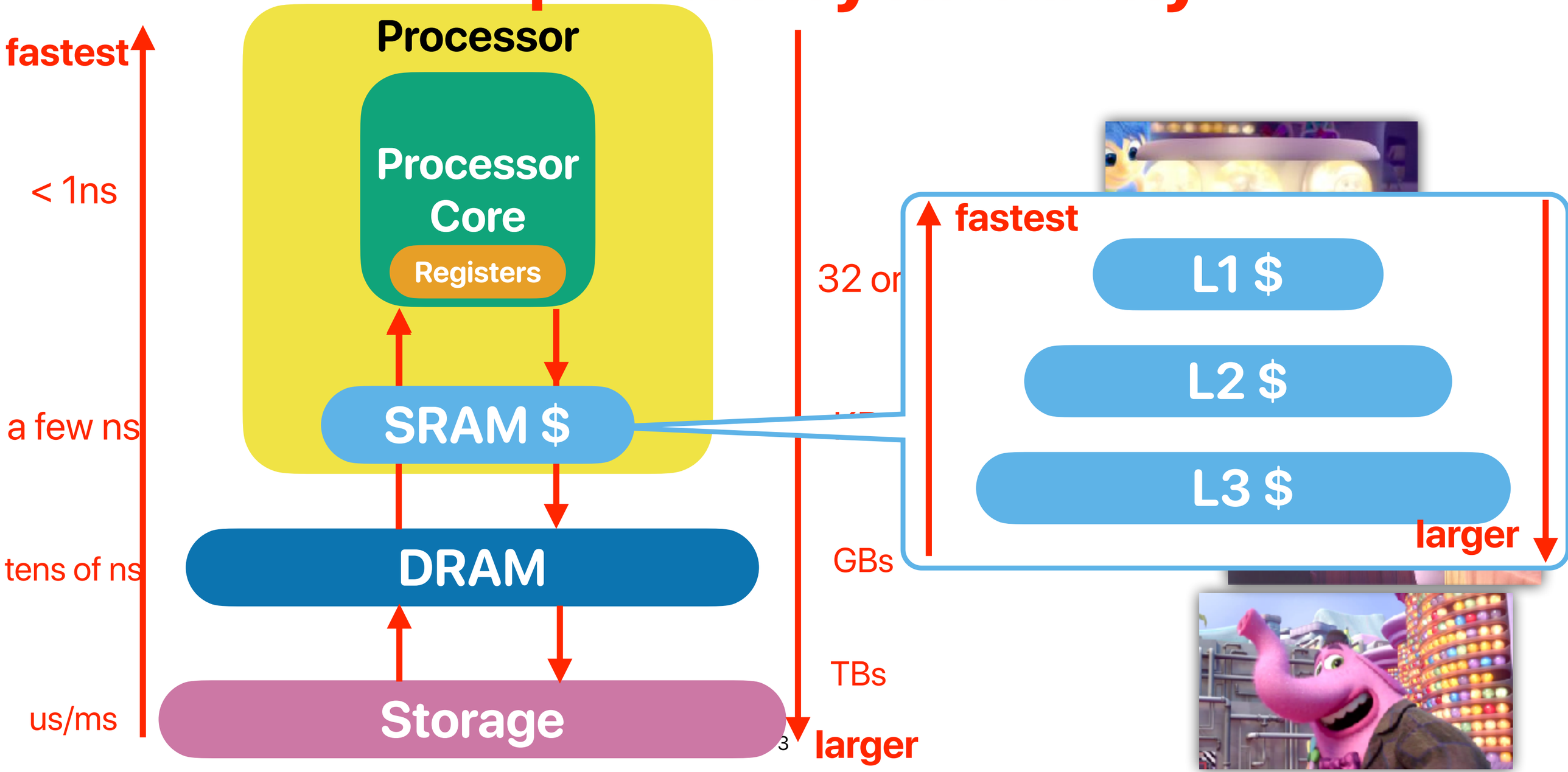
Hung-Wei Tseng

Recap: the "latency" gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

Recap: Memory Hierarchy



Recap: Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

spatial locality:
matrix[0][0], matrix[0][1], matrix[0][2], ...
vector[0], vector[1], ... , vector[n]
temporal locality:
reuse of vector[0], vector[1], ... ,

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

Recap: Code also has locality

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_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!

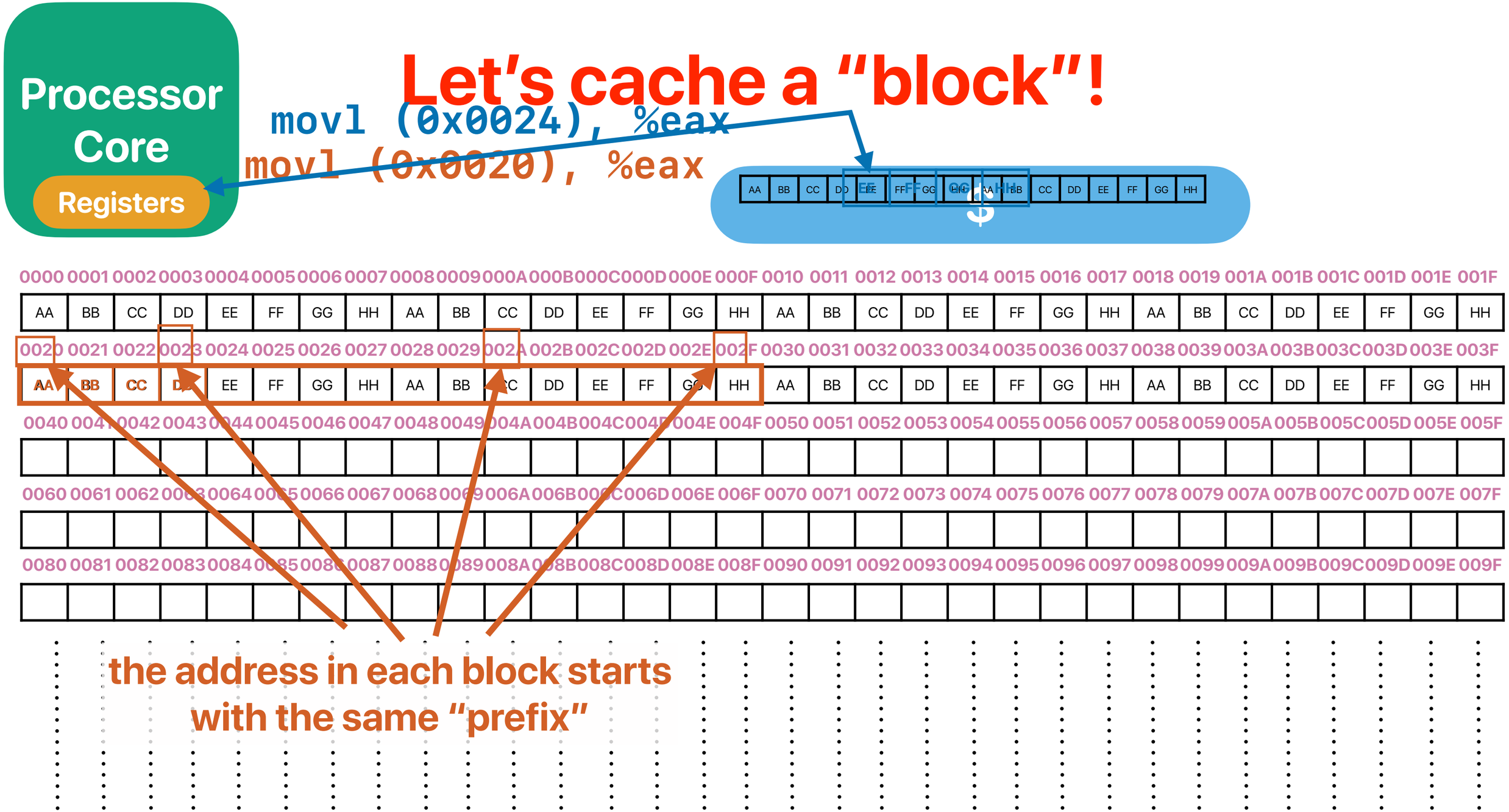
Recap: inside out our memory hierarchy

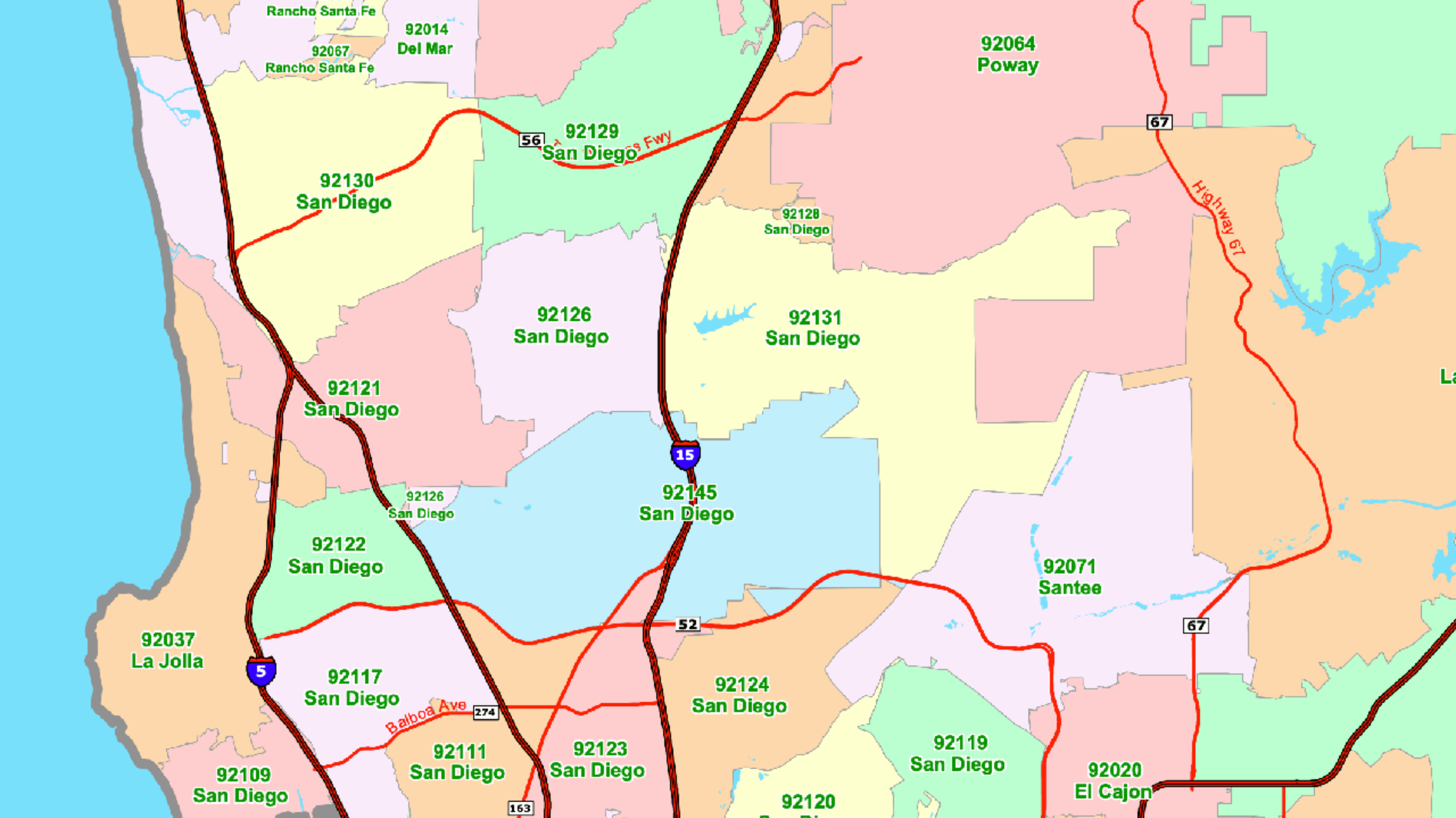
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality

Outline

- Designing a cache that captures the predictability
- The A, B, Cs of caches
- Estimating how cache friendly is our code

Let's cache a "block"!




Rancho Santa Fe
92014
Del Mar
92067
Rancho Santa Fe
92064
Poway
56
92129
San Diego
Fwy
67
92130
San Diego
92128
San Diego
Highway 67
92126
San Diego
92131
San Diego
92121
San Diego
15
92145
San Diego
92126
San Diego
92122
San Diego
92071
Santee
52
67
92037
La Jolla
5
92117
San Diego
Balboa Ave
274
92124
San Diego
92111
San Diego
92123
San Diego
92119
San Diego
92109
San Diego
92020
El Cajon
163
92120

What's a block?

the offset of the byte within a block

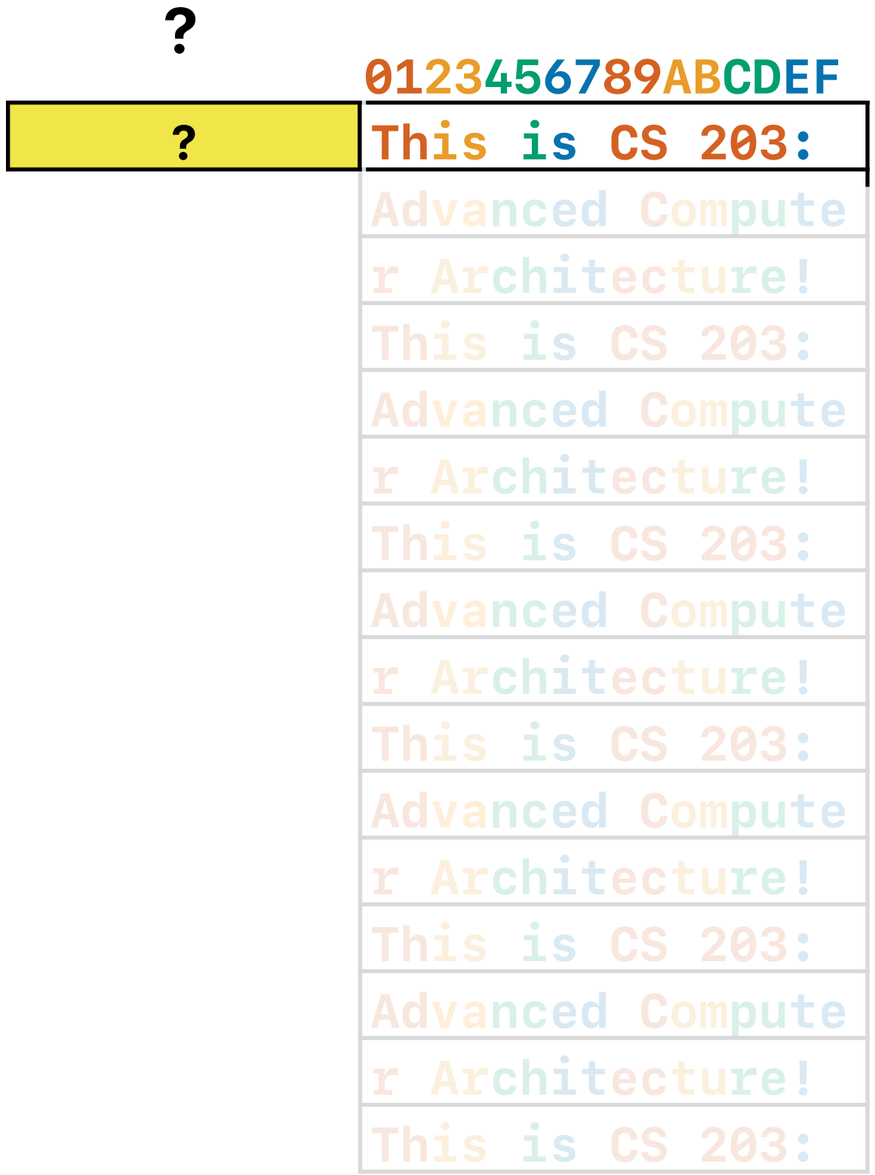
the data in memory

```
0123456789ABCDEF
0x0000, 0x0001, 0x0002, …, 0x000F   This is CS 203:
0x0010, 0x0011, 0x0012, …, 0x001F   Advanced Compute
```

the byte addresses of each byte in the block

the address in each block starts with the same "prefix"

How to tell who is there?



Processor Core

Registers

How to tell who is there?

the common address prefix in each block

tag array	0x000 0x001 0x002 0x003 0x004 0x005 0x006 0x007 0x008 0x009 0x00A 0x00B 0x00C 0x00D 0x00E 0x00F 0123456789ABCDEF
0x000	This is CS 203:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CS 203:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CS 203:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CS 203:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CS 203:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CS 203:

How to tell w

Tell if the block here can be used

Tell if the block here is modified

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

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:

Blocksize == Linesize

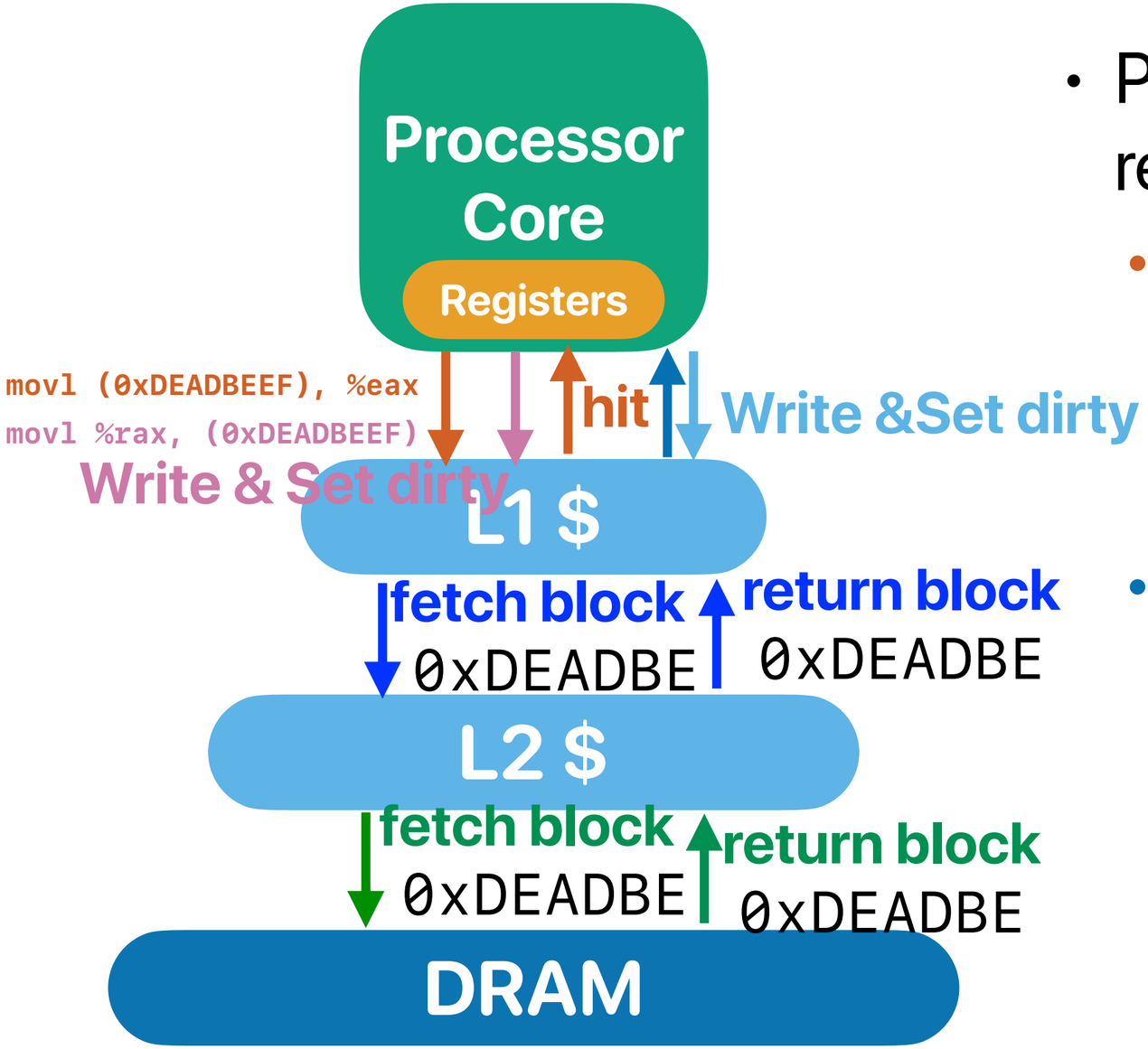
```
[5]: # Your CS203 Cluster
     ! cs203 demo "lscpu | grep 'Model name'; getconf -a | grep CACHE"

     ssh htseng@horsea " srun -N1 -p datahub lscpu | grep 'Model name'"
     Model name:                          12th Gen Intel(R) Core(TM) i3-12100F
     ssh htseng@horsea " srun -N1 -p datahub  getconf -a | grep CACHE"
     LEVEL1_ICACHE_SIZE                 32768
     LEVEL1_ICACHE_ASSOC                8
     LEVEL1_ICACHE_LINESIZE             64
     LEVEL1_DCACHE_SIZE                 49152
     LEVEL1_DCACHE_ASSOC                12
     LEVEL1_DCACHE_LINESIZE             64
     LEVEL2_CACHE_SIZE                  1310720
     LEVEL2_CACHE_ASSOC                 10
     LEVEL2_CACHE_LINESIZE              64
     LEVEL3_CACHE_SIZE                  12582912
     LEVEL3_CACHE_ASSOC                 12
     LEVEL3_CACHE_LINESIZE              64
     LEVEL4_CACHE_SIZE                  0
     LEVEL4_CACHE_ASSOC                 0
     LEVEL4_CACHE_LINESIZE              0
```

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks

Put everything all together: How cache interacts with CPU

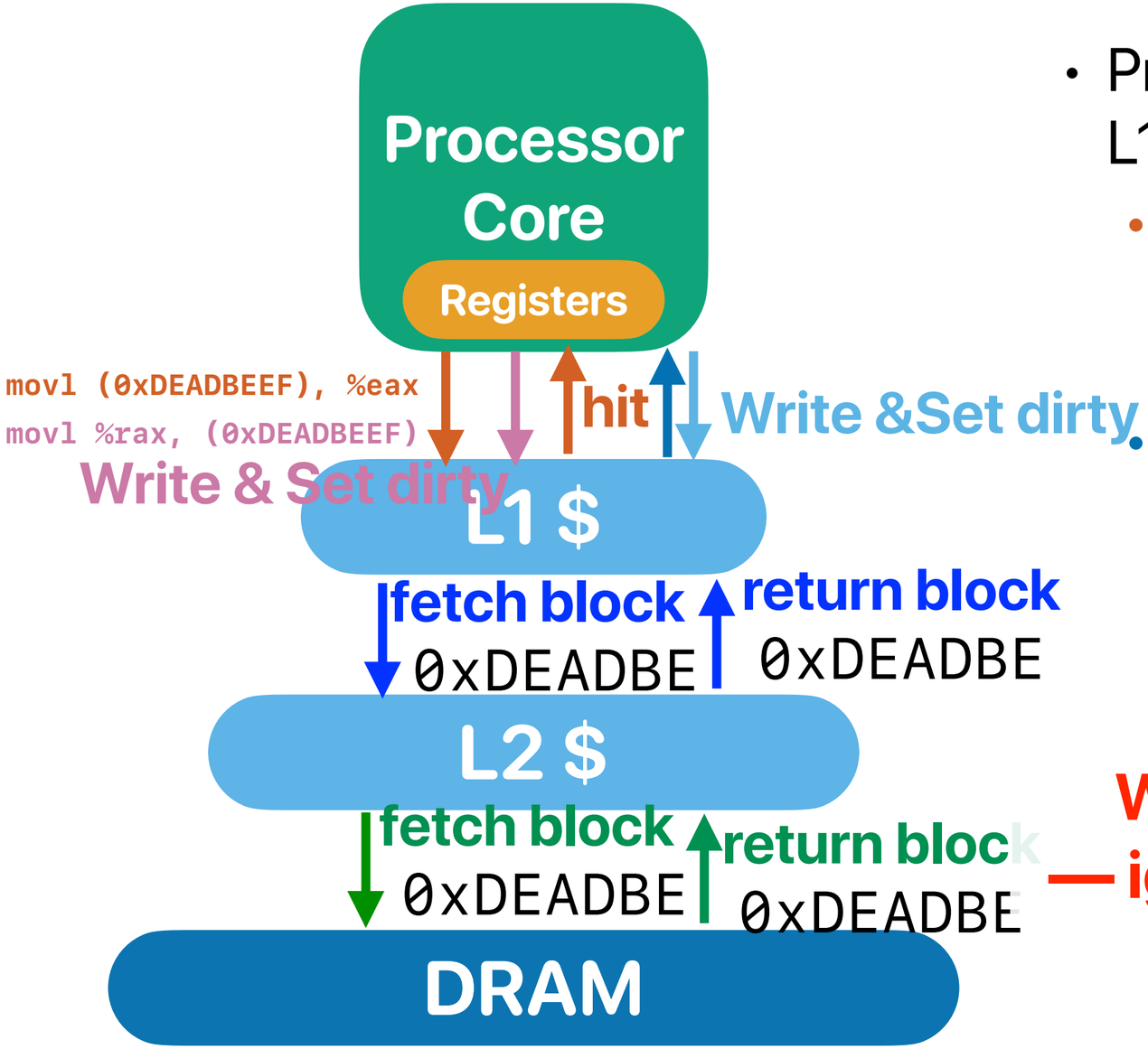
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY** **Why don't we write to L2? — Too slow**
  - **if miss**
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

What if we run out of $ blocks?

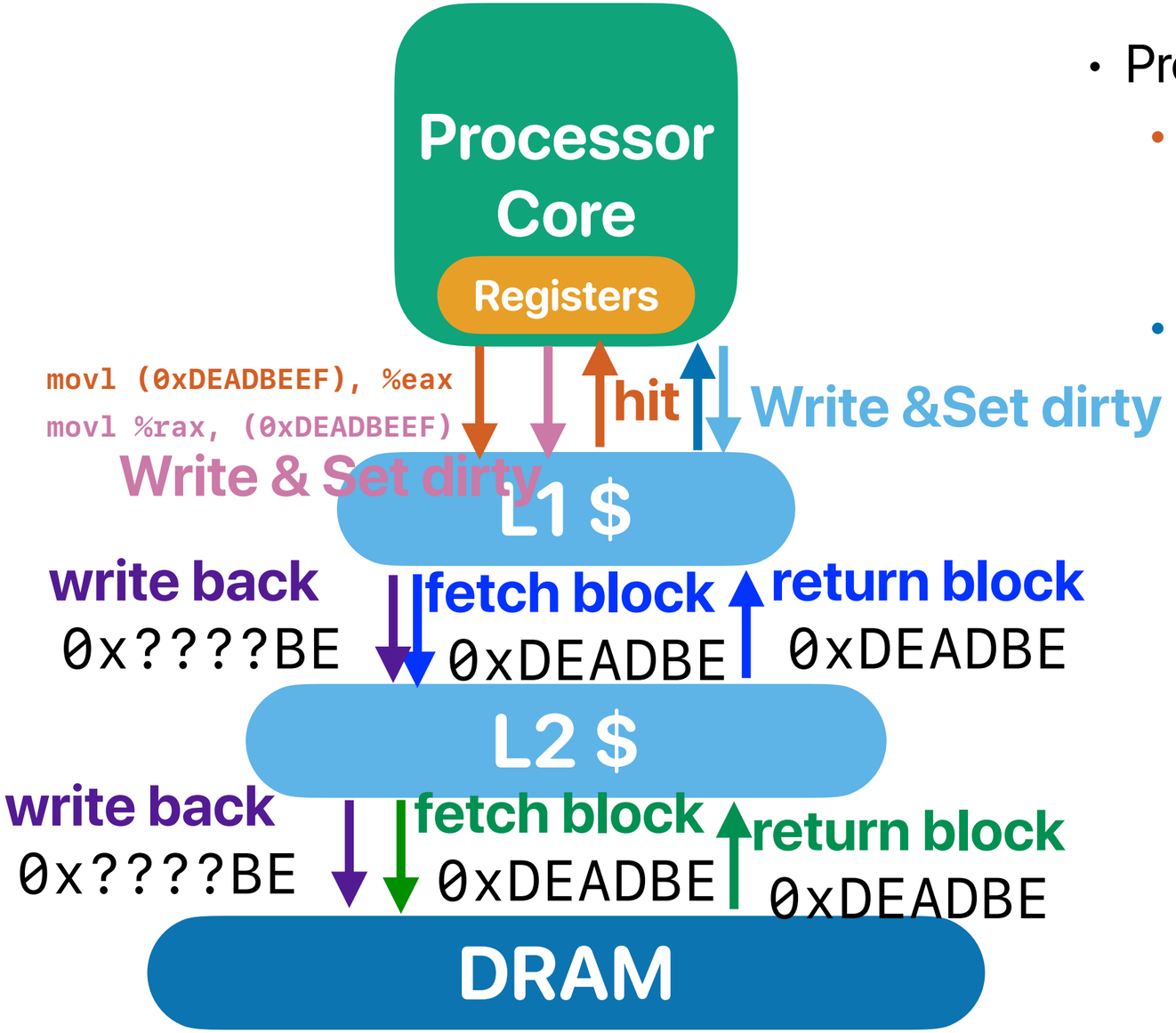
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - Fetch the requesting block from lower level memory hierarchy and place in the cache
    - Present the write "ONLY" in L1 and set DIRTY

What if the victim block is modified? — ignoring the update is not acceptable!

Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use

How to tell w

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

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:

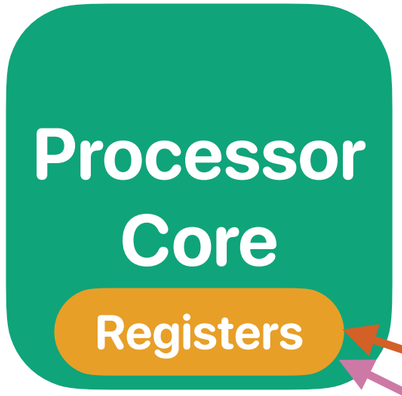
The complexity of search the matching tag—

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

Can we search things faster?

—hash table! $O(1)$

Hash-like structure — direct-mapped cache



block offset

tag

index

load 0x0008

load 0x4048

0x40 not found,
go to lower-level memory

V	D	tag	data (0123456789ABCDEF)
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
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time

Hash-like structure — direct-mapped cache

block offset

tag

index

load 0x0008

load 0x4048

0x40 not found,
go to lower-level memory

V	D	tag	data (0123456789ABCDEF)
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
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

The biggest issue with hash is —
Collision!

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b0000100000100100

tag | set index | block offset

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

Set

0x1 0

=? =?

hit? hit?

What is Associativity?

```
[5]: # Your CS203 Cluster
     ! cs203 demo "lscpu | grep 'Model name'; getconf -a | grep CACHE"

     ssh htseng@horsea " srun -N1 -p datahub lscpu | grep 'Model name'"
     Model name:                          12th Gen Intel(R) Core(TM) i3-12100F
     ssh htseng@horsea " srun -N1 -p datahub  getconf -a | grep CACHE"
     LEVEL1_ICACHE_SIZE                   32768
     LEVEL1_ICACHE_ASSOC                  8
     LEVEL1_ICACHE_LINESIZE               64
     LEVEL1_DCACHE_SIZE                   49152
     LEVEL1_DCACHE_ASSOC                  12
     LEVEL1_DCACHE_LINESIZE               64
     LEVEL2_CACHE_SIZE                    1310720
     LEVEL2_CACHE_ASSOC                   10
     LEVEL2_CACHE_LINESIZE                64
     LEVEL3_CACHE_SIZE                    12582912
     LEVEL3_CACHE_ASSOC                   12
     LEVEL3_CACHE_LINESIZE                64
     LEVEL4_CACHE_SIZE                    0
     LEVEL4_CACHE_ASSOC                   0
     LEVEL4_CACHE_LINESIZE                0
```

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time
  - Set-associativity to reduce the "collision" problem

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b00001000001000100

tag | set index | block offset

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:

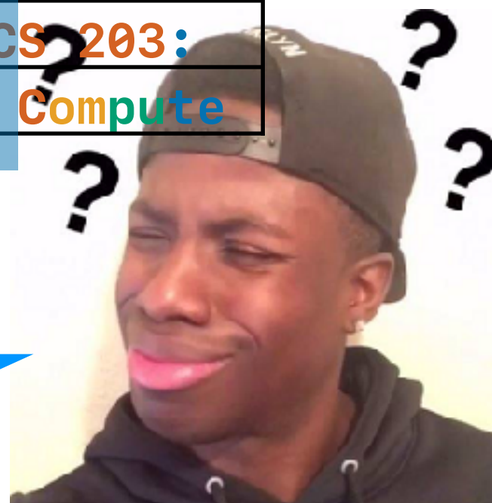
V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

Set

0x1 0

=? =?

hit? hit?



The A, B, Cs of your 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 (Linesize)
  - 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

	tag	set index	block offset
memory address:	0b00001000	0100	0100

- number of bits in **b**lock offset — $log_2(B)$
- number of bits in **s**et index: $log_2(S)$
- tag bits: address_length - $log_2(S)$- $log_2(B)$
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

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

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

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

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

$$S = 64$$

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

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

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

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time
  - Set-associativity to reduce the "collision" problem
- C = A B S
  - C: capacity
  - A: Associativity
  - S: Number of sets
  - lg(S): Number of bits in set index
  - lg(B): Number of bits in block offset

Estimating code performance on caches

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`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
     for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
     }
      output[i] = result;
 }
```

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

	Address (Hex) (tag index)	Address (Binary) (tag index)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

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: 0b00001000001000100

tag | set index | block offset

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

Set

0x1 0

=? =?

hit? hit?

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`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

tag index tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

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%

Announcement

- Reading quiz #4 due **Wednesday** before the lecture
- Assignment #2 due **this Saturday**
  - You should run the performance measurement yourself and calculate results based on that — everyone should have a different answer
  - All questions this time require **correct** estimations in cache performance to help you better prepare the examines

Computer
Science &
Engineering

つづく