

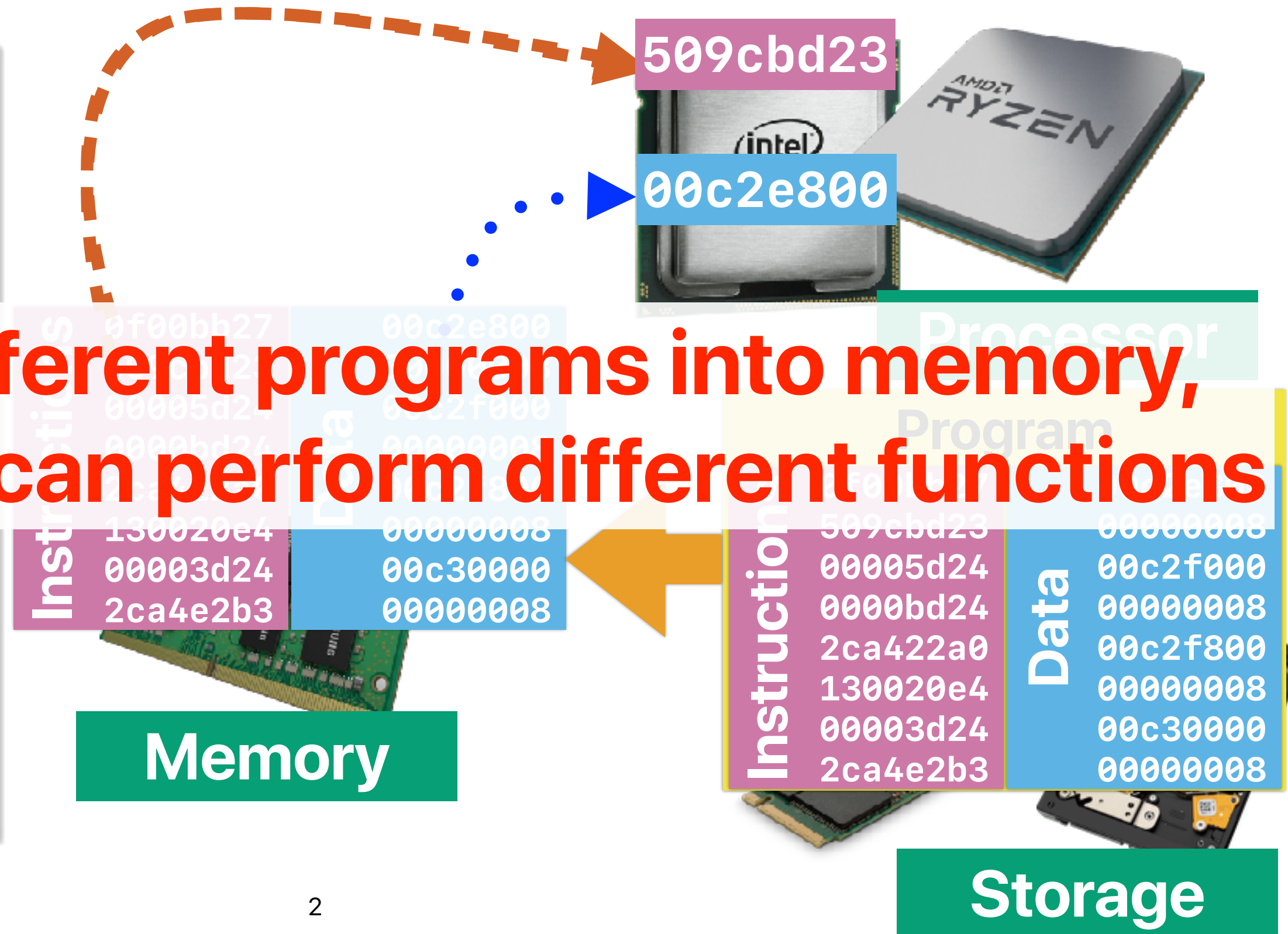
# Memory Hierarchy

Hung-Wei Tseng

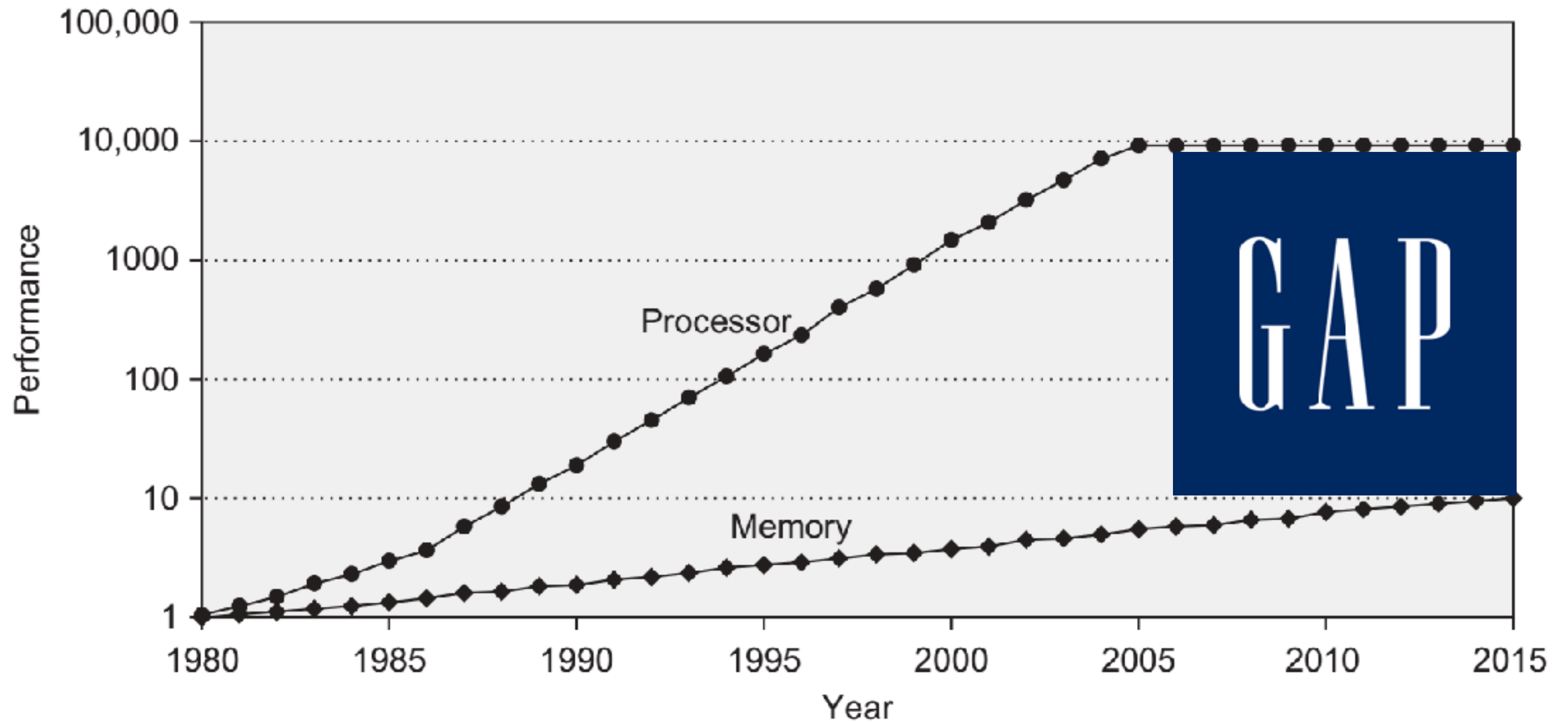
# von Neumann Architecture



By loading different programs into memory, your computer can perform different functions



# Performance gap between Processor/Memory



# Performance of modern DRAM

Production year	Chip size	DRAM type	Best case access time (no precharge)			Precharge needed
			RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

**Figure 2.4 Capacity and access times for DDR SDRAMs by year of production.** Access time is for a random memory word and assumes a new row must be opened. If the row is in a different bank, we assume the bank is precharged; if the row is not open, then a precharge is required, and the access time is longer. As the number of banks has increased, the ability to hide the precharge time has also increased. DDR4 SDRAMs were initially expected in 2014, but did not begin production until early 2016.



# Thinking about water



# Alternatives?

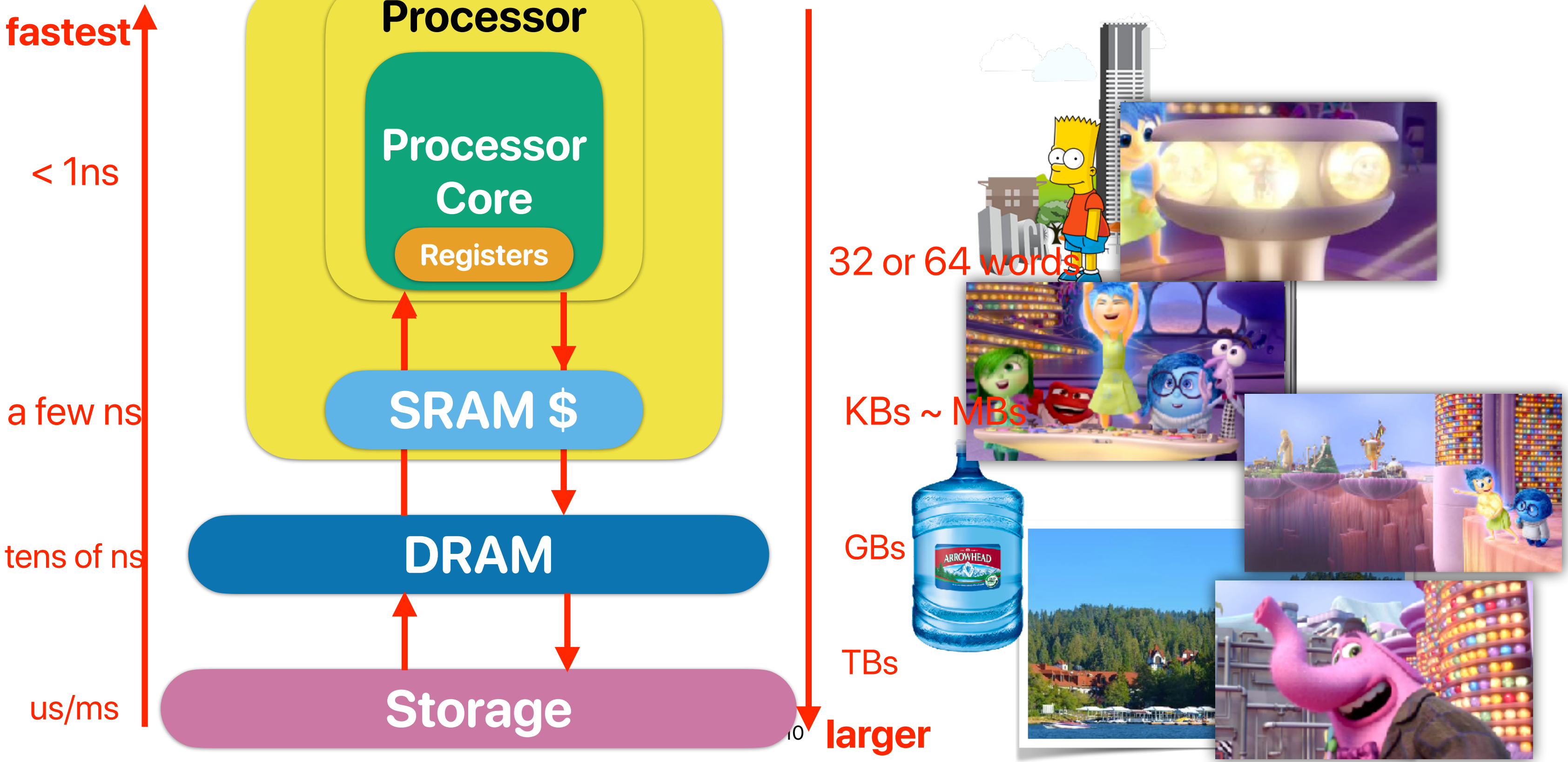
Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10



**Fast, but expensive \$\$\$**



# Memory Hierarchy




# L1? L2? L3?

CPU-Z - ID : wswpbb

CPU Caches Mainboard Memory SPD Graphics Bench About

Processor

Name	AMD Ryzen 7 2700X		
Code Name	Pinnacle Ridge	Max TDP	105 W
Package	Socket AM4 (1331)		
Technology	12 nm	Core Voltage	1.36 V



Specification

AMD Ryzen 7 2700X Eight-Core Processor

Family	F	Model	8	Stepping	2
Ext. Family	17	Ext. Model	8	Revision	PIR-B2
Instructions	MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AMD-V, AES, AVX, AVX2, FMA3, SHA				

Clocks (Core #0)

Core Speed	4290.73 MHz
Multiplier	x 43.0
Bus Speed	99.78 MHz
Rated FSB	

Cache

L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 64 KBytes	4-way
Level 2	8 x 512 KBytes	8-way
Level 3	2 x 8192 KBytes	16-way


Selection Processor #1 Cores 8 Threads 16

CPU-Z Ver. 1.86.0.x64 Tools Validate Close

CPU Caches Mainboard Memory SPD Graphics Bench About

Processor

Name	Intel Core i7 9700K		
Code Name	Coffee Lake	Max TDP	95.0 W
Package	Socket 1151 LGA		
Technology	14 nm	Core Voltage	0.737 V



Specification

Intel® Core™ i7-9700K CPU @ 3.60GHz (ES)

Family	6	Model	E	Stepping	C
Ext. Family	6	Ext. Model	9E	Revision	P0
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3, TSX				

Clocks (Core #0)

Core Speed	4798.85 MHz
Multiplier	x 48.0 ( 8 - 49 )
Bus Speed	99.98 MHz
Rated FSB	

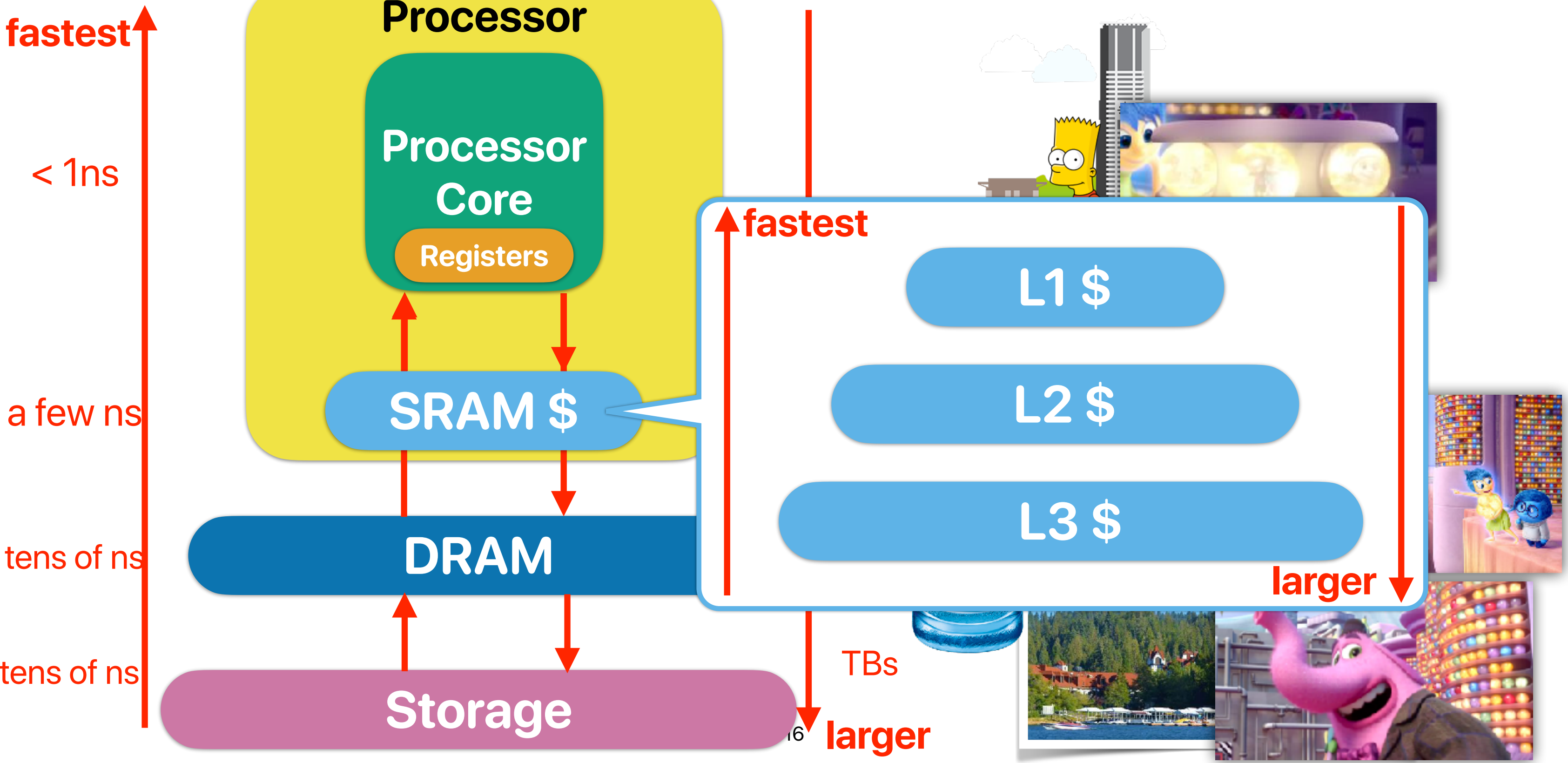
Cache

L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 32 KBytes	8-way
Level 2	8 x 256 KBytes	4-way
Level 3	12 MBytes	12-way

Selection Socket #1 Cores 8 Threads 8



# Memory Hierarchy



**Why adding small SRAMs would  
work?**

# Locality

- Spatial locality — application tends to visit nearby stuffs in the memory

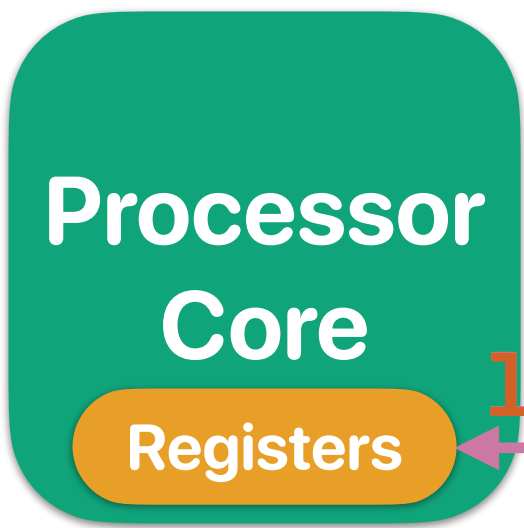
- Code — the current instruction, and then  $PC + 4$

**Most of time, your program is just visiting a very small amount of data/instructions within a given window**

- Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

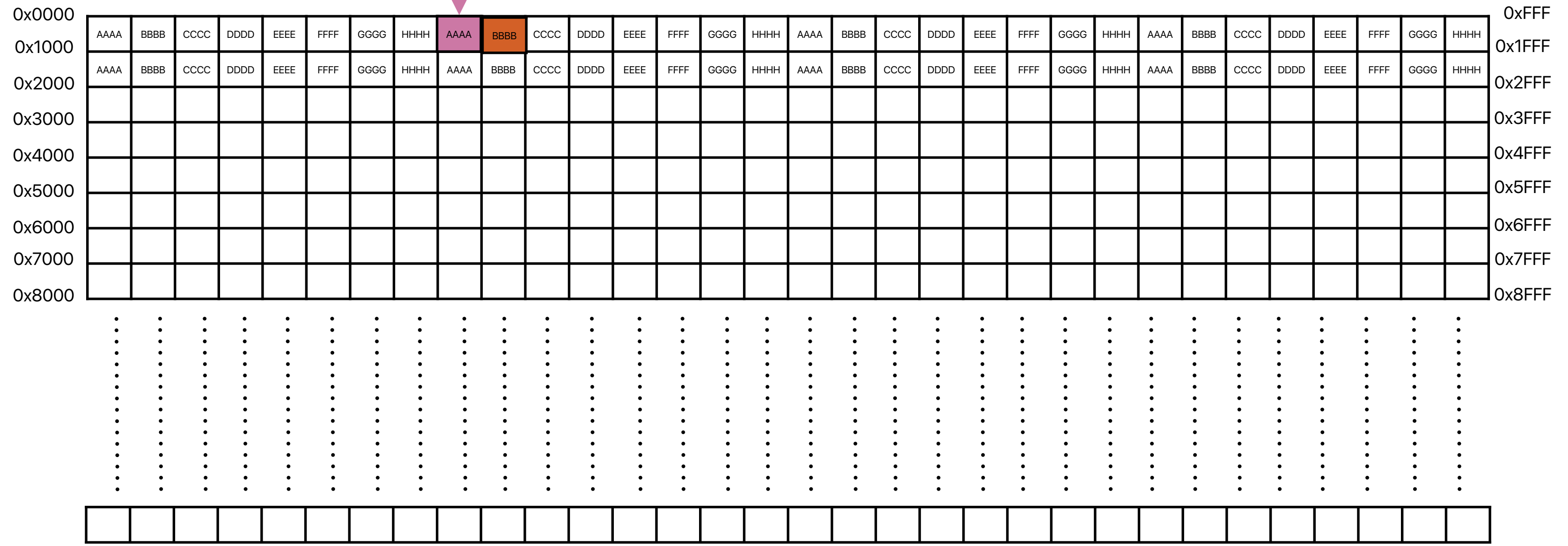



# Architecting the Cache



# Load/store only access a "word" each time

load 0x000A






# Processor Core

## Registers

1

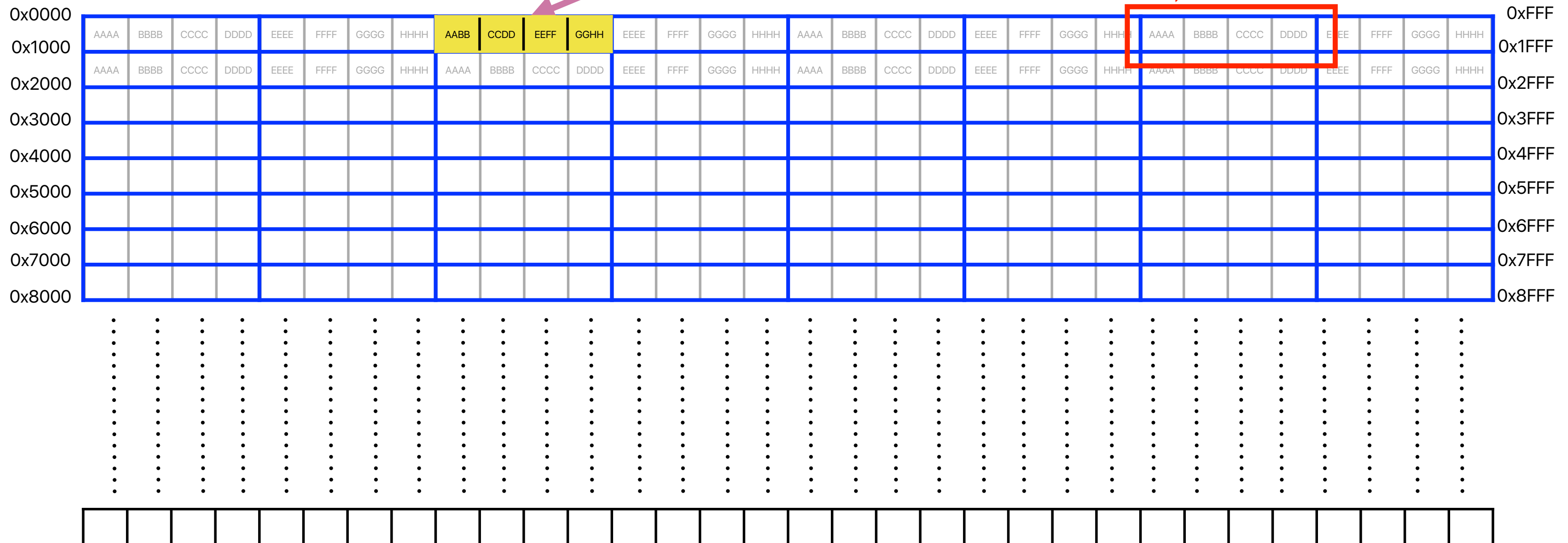
## Registers

Load 0x000A



A diagram showing a blue rounded rectangle containing the text "SRAM \$". Below the text, there are two adjacent squares: a purple one on the left labeled "AABB" and an orange one on the right labeled "CCDD".

**"Logically" partition  
memory space into  
↓  
"blocks"**







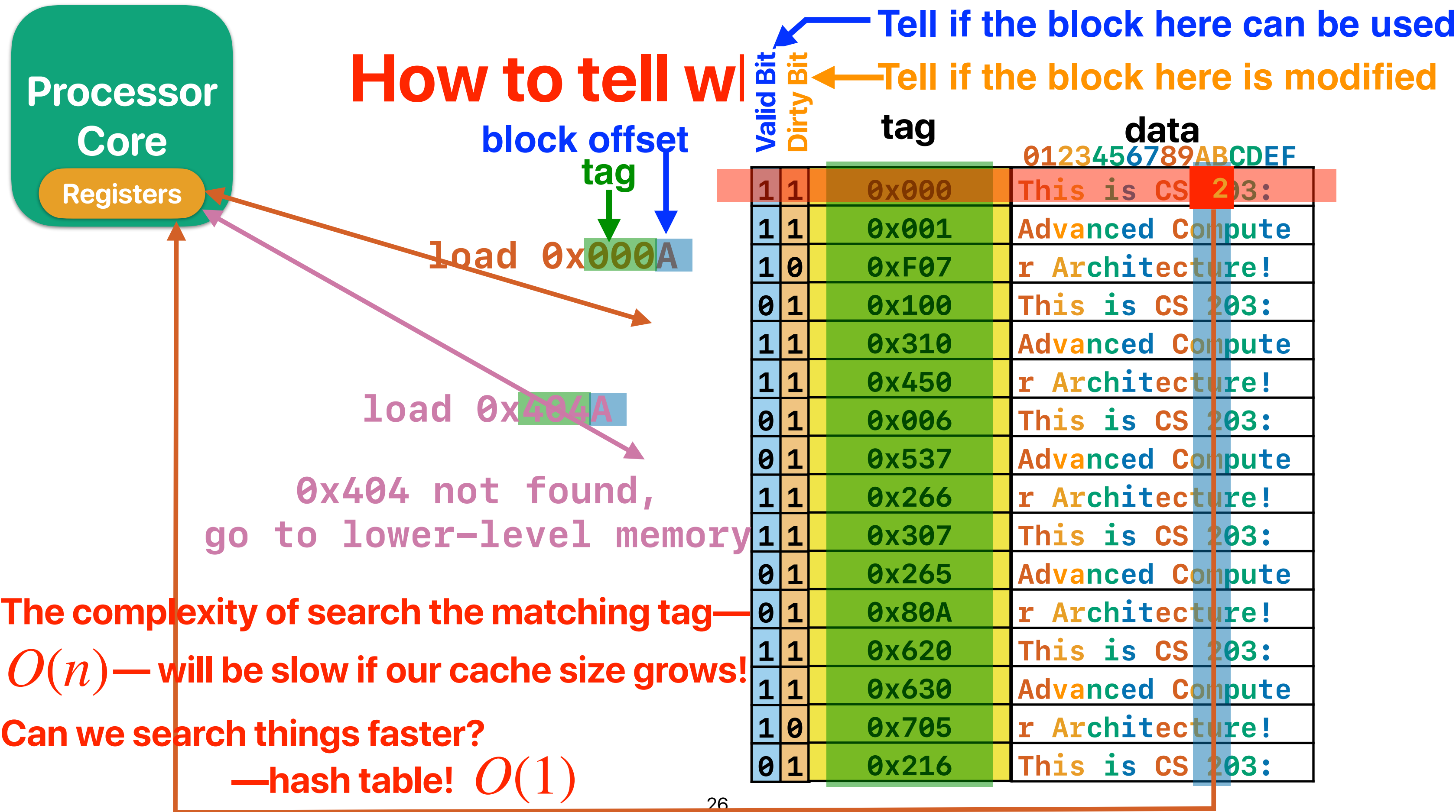
# How to tell who is there?

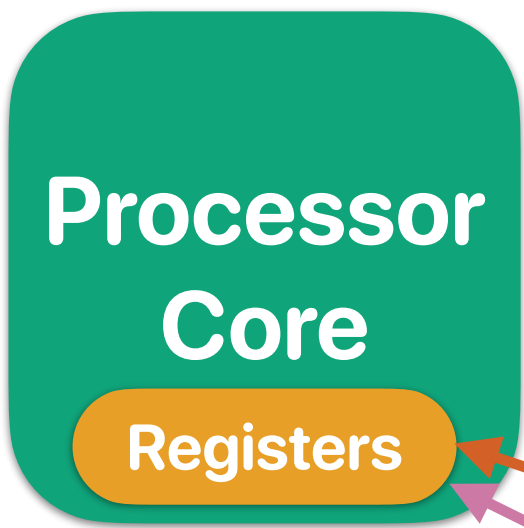
tag

00000000000000000000000000000000  
0123456789ABCDEF0123456789ABCDEF  
0x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000x0000

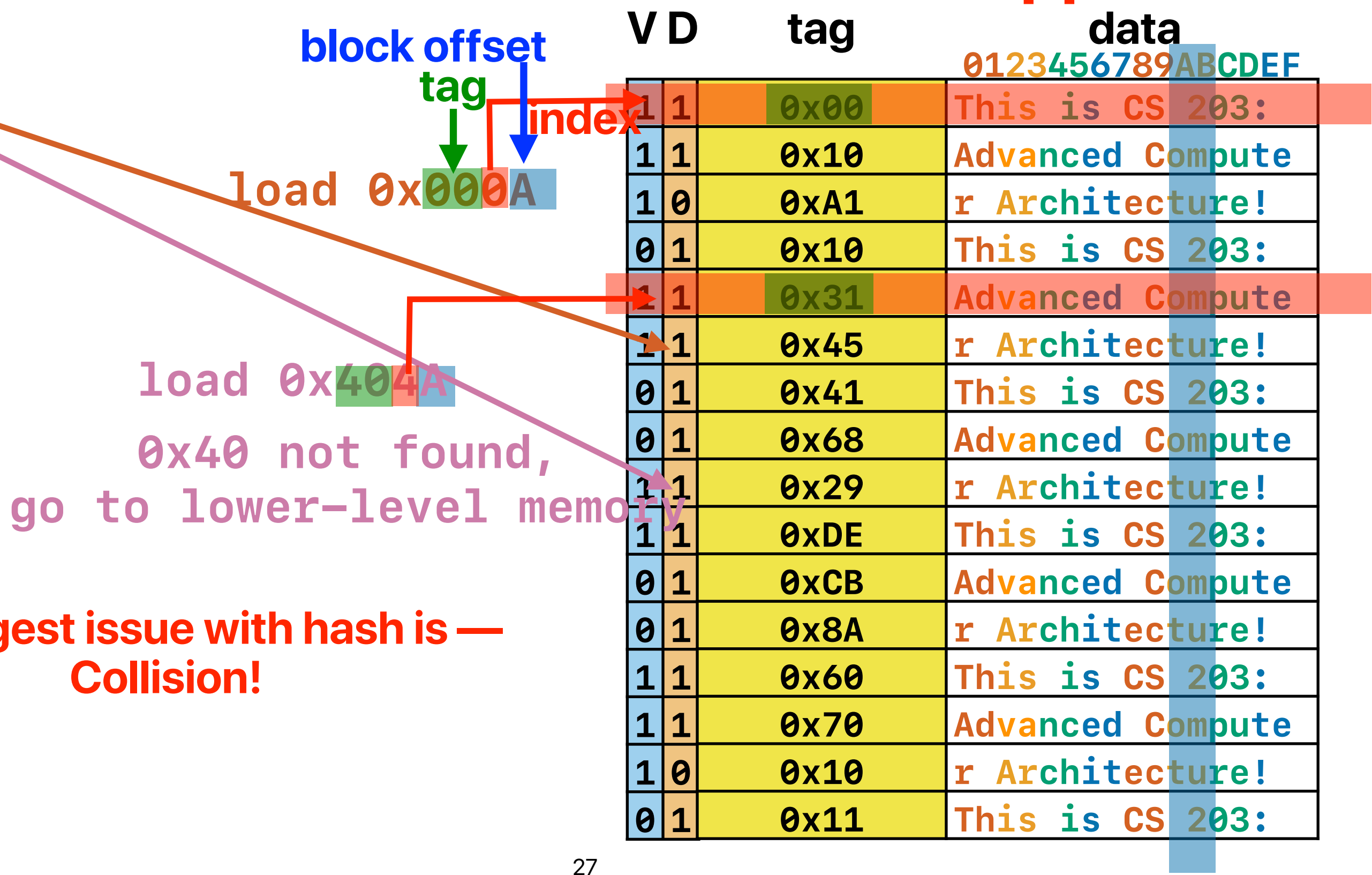
0x000	This is CS 203: Advanced Computer Architecture! This is CS 203: Advanced Computer Architecture! This is CS 203: Advanced Computer Architecture! This is CS 203: Advanced Computer Architecture! This is CS 203: Advanced Computer Architecture! This is CS 203:
-------	---

# How to tell w





# Hash-like structure — direct-mapped cache



The biggest issue with hash is — Collision!



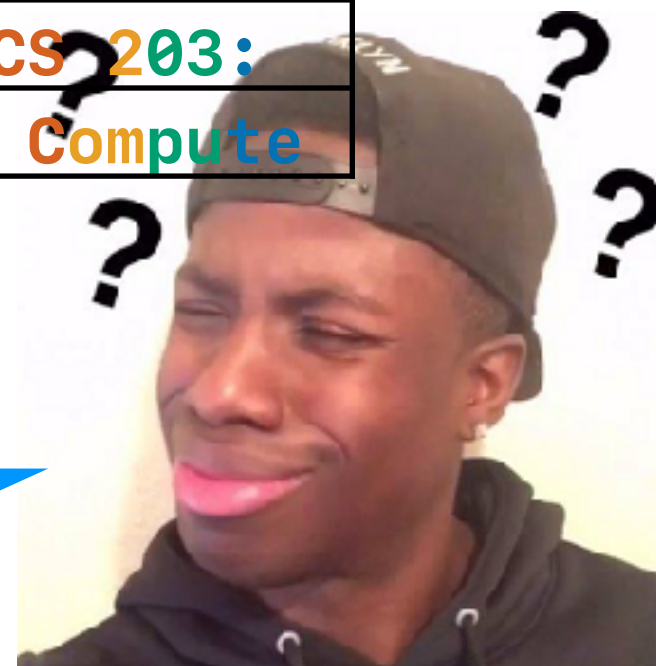
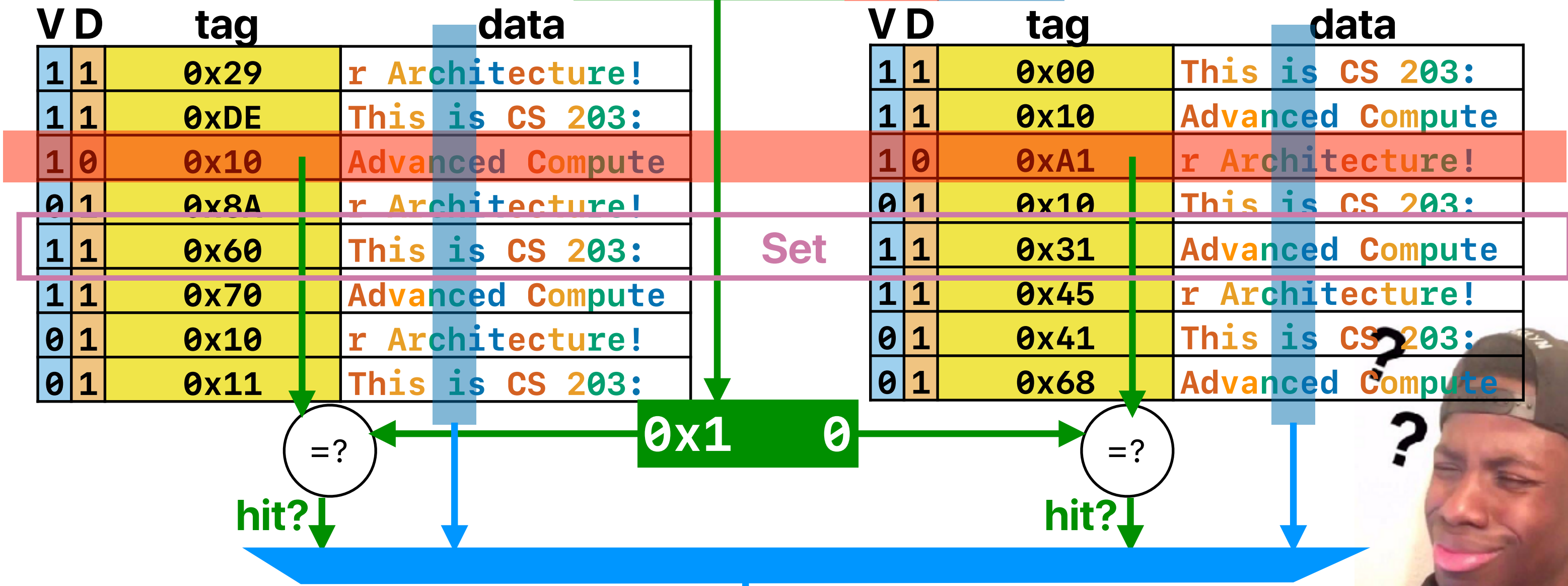
# Way-associative cache

memory address:      0x0      8      2      4

set    block

tag    index   offset

memory address:      0b00001000000100100



$$C = ABS$$

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



# Corollary of $C = ABS$

memory address:      0b 000010000 010 0100

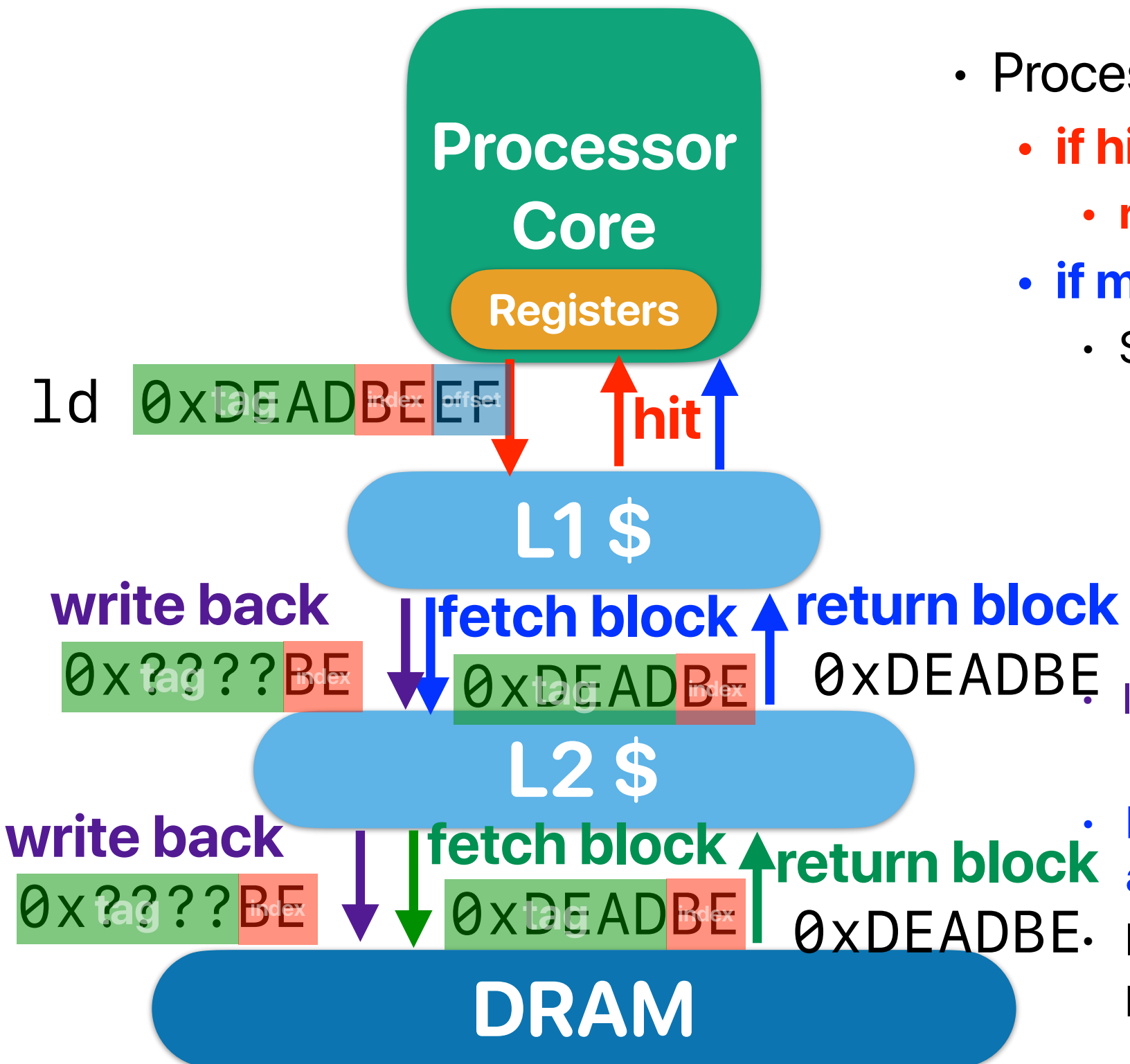
tag      set index block offset

- number of bits in **block** offset —  $\lg(\mathbf{B})$
- number of bits in **set** index:  $\lg(\mathbf{S})$
- tag bits:  $\text{address\_length} - \lg(\mathbf{S}) - \lg(\mathbf{B})$ 
  - address\_length is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \% \mathbf{S} = \text{set index}$



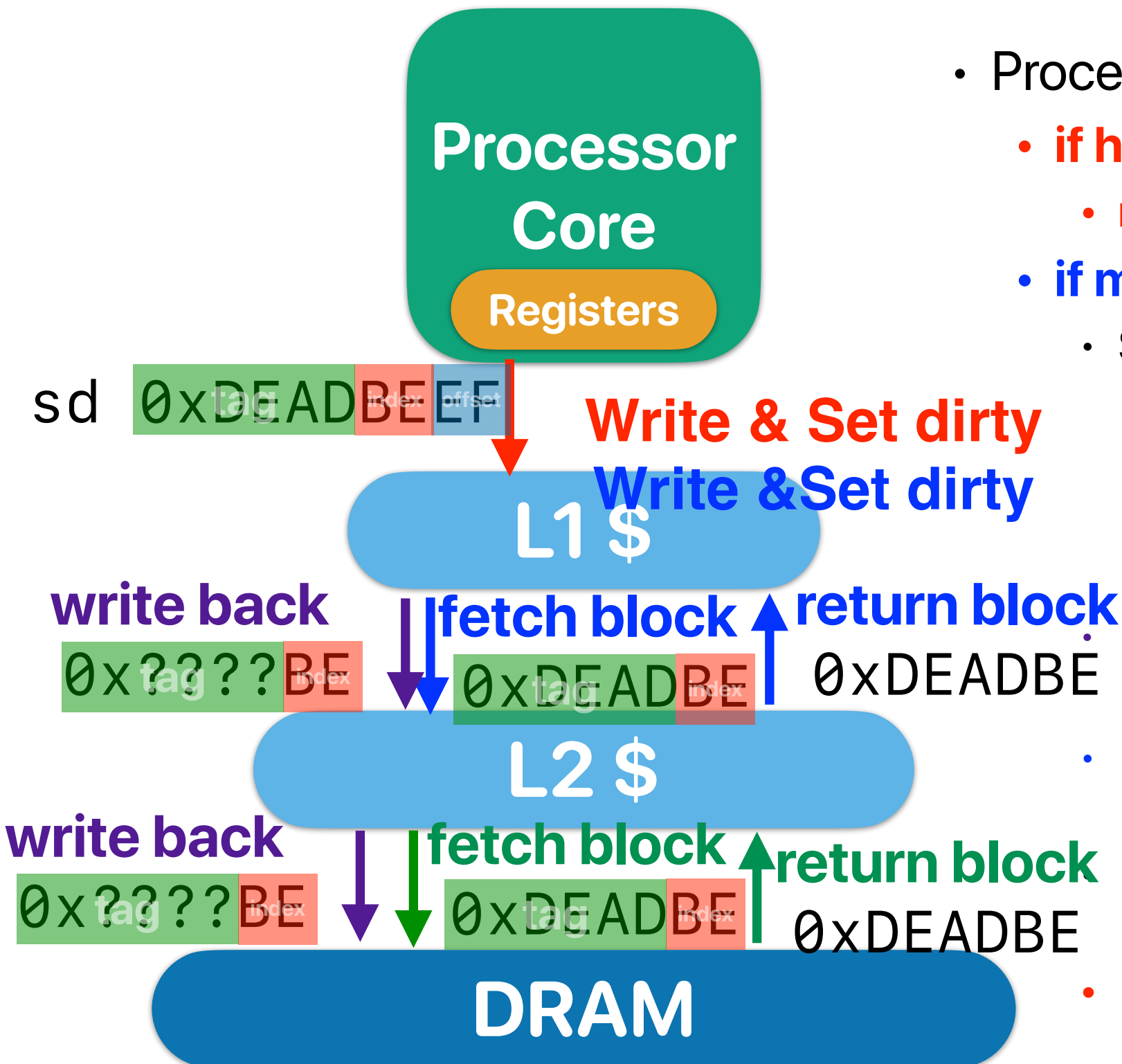
**Put everything all together:  
How cache interacts with CPU**

# 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:
  - 0b10000000000, 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! This is CS 203:
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		

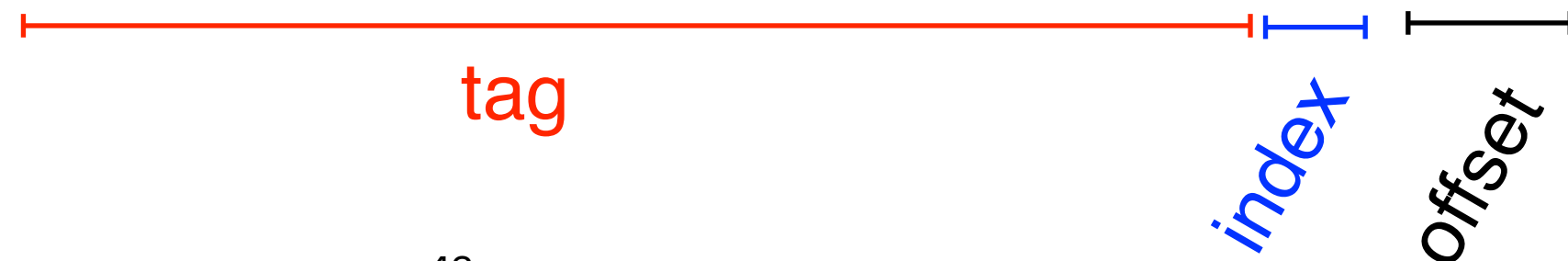
	tag	index		
0	0b10	0000	0000	miss
1	0b10	0000	1000	hit!
2	0b10	0001	0000	miss
3	0b10	0001	0100	hit!
4	0b11	0001	0000	miss
5	0b10	0000	0000	hit!
6	0b10	0000	1000	hit!
7	0b10	0001	0000	miss
8	0b10	0001	0100	hit!

# 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	r Architecture!	0	0		
1	1	0	0b10	This is CS 203:	1	0	0b11	Advanced Compute
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!

# 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:
  - 0b10000000000, 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		
0	0b10	0000	0000	compulsory miss
1	0b10	0000	1000	hit!
2	0b10	0001	0000	compulsory miss
3	0b10	0001	0100	hit!
4	0b11	0001	0000	compulsory miss
5	0b10	0000	0000	hit!
6	0b10	0000	1000	hit!
7	0b10	0001	0000	conflict miss
8	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:
  - 0b10000000000, 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!	

# Improving 3Cs



# 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

# 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

# Array of structures or structure of arrays

	Array of objects	object of arrays									
	<pre>struct grades {     int id;     double *homework;     double average; };</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; };</pre>									
ID   *homework   average   ID   *homework   average		<table><tr><td>ID</td><td>ID</td><td>ID</td></tr><tr><td>homework</td><td>homework</td><td>homework</td></tr><tr><td>average</td><td>average</td><td>average</td></tr></table>	ID	ID	ID	homework	homework	homework	average	average	average
ID	ID	ID									
homework	homework	homework									
average	average	average									
average of each homework	<pre>for(i=0;i&lt;homework_items; i++) {     gradesheet[total_number_students].homework[i] = 0.0;     for(j=0;j&lt;total_number_students;j++)         gradesheet[total_number_students].homework[i]             +=gradesheet[j].homework[i];     gradesheet[total_number_students].homework[i] /=         (double)total_number_students; }</pre>	<pre>for(i = 0;i &lt; homework_items; i++) {     gradesheet.homework[i][total_number_students] = 0.0;     for(j = 0; j &lt;total_number_students;j++)     {         gradesheet.homework[i][total_number_students] +=             gradesheet.homework[i][j];     }     gradesheet.homework[i][total_number_students] /=         total_number_students; }</pre>									



# What data structure is performing better

	Array of objects	object of arrays
	<pre>struct grades {     int id;     double *homework;     double average; };</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; };</pre>
average of each homework	<pre>for(i=0;i&lt;homework_items; i++) {     gradesheet[total_number_students].homework[i] = 0.0;     for(j=0;j&lt;total_number_students;j++)     gradesheet[total_number_students].homework[i]     +=gradesheet[j].homework[i];     gradesheet[total_number_students].homework[i] /=     (double)total_number_students; }</pre>	<pre>for(i = 0;i &lt; homework_items; i++) {     gradesheet.homework[i][total_number_students] = 0.0;     for(j = 0; j &lt;total_number_students;j++)     {         gradesheet.homework[i][total_number_students] +=         gradesheet.homework[i][j];     }     gradesheet.homework[i][total_number_students] /=     total_number_students; }</pre>

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance? **What if we want to calculate average scores for each student?**

A. Array of objects

B. Object of arrays

# Column-store or row-store

- If you're designing an in-memory database system, will you be using

RowId	Empld	Lastname	Firstname	Salary
1	10	Smith	Joe	40000
2	12	Jones	Mary	50000
3	11	Johnson	Cathy	44000
4	22	Jones	Bob	55000

- column-store — stores data tables column by column

10:001, 12:002, 11:003, 22:004;  
Smith:001, Jones:002, Johnson:003, Jones:004,  
Joe:001, Mary:002, Cathy:003, Bob:004;  
40000:001, 50000:002, 44000:003, 55000:004;

if the most frequently used query looks like —  
**select Lastname, Firstname from table**

- row-store — stores data tables row by row

001:10, Smith, Joe, 40000;  
002:12, Jones, Mary, 50000;  
003:11, Johnson, Cathy, 44000;  
004:22, Jones, Bob, 55000;

# Loop interchange/fission/fusion

# Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
```

$O(n^2)$

Complexity

$O(n^2)$

Same

Instruction Count?

Same

Same

Clock Rate

Same

Better

CPI

Worse

# 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%

$$C = ABS$$

$$64KB = 2 * 64 * S$$

$$S = 512$$

$$\text{offset} = \lg(64) = 6 \text{ bits}$$

$$\text{index} = \lg(512) = 9 \text{ bits}$$

$$\text{tag} = 64 - \lg(512) - \lg(64) = 49 \text{ bits}$$

# Loop Fusion

```
/* Before */  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1)  
        a[i][j] = 1/b[i][j] * c[i][j];  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1)  
        d[i][j] = a[i][j] + c[i][j];
```

```
/* After */  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1)  
    {  
        a[i][j] = 1/b[i][j] * c[i][j];  
        d[i][j] = a[i][j] + c[i][j];  
    }
```

**2 misses per access to a & c vs. one miss per access**



# Blocking

# Case study: Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

**Algorithm class tells you it's  $O(n^3)$**

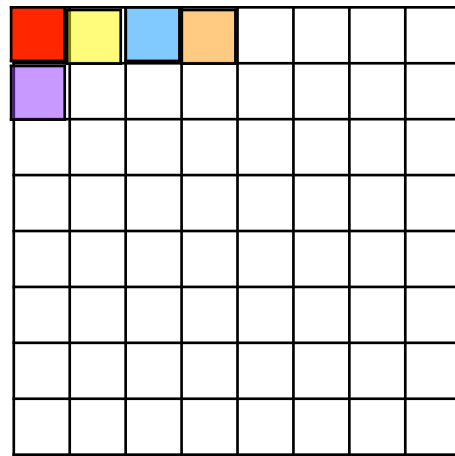
**If  $n=1024$ , it takes about 1 sec**

**How long is it take when  $n=2048$ ?**

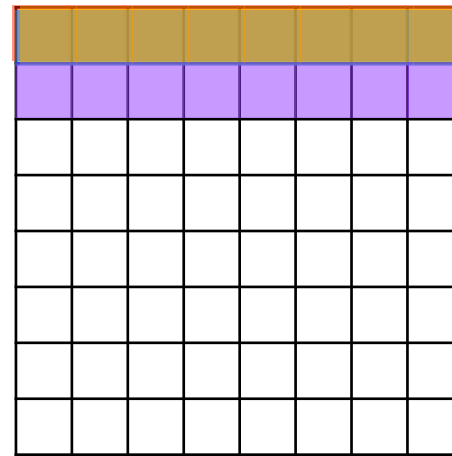
# Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

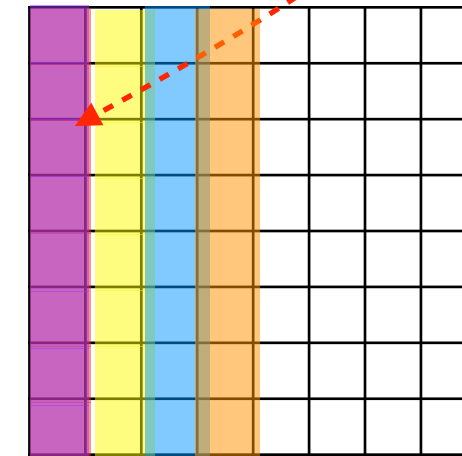
Very likely a miss if  
array is large



c



a



b

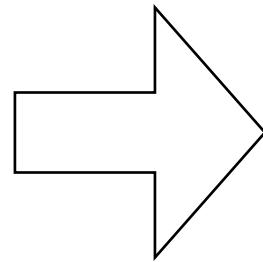
- If each dimension of your matrix is 2048
  - Each row takes  $2048 \times 8$  bytes = 16KB
  - The L1 \$ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 2 rows/columns of each matrix!
  - You need the same row when j increase!

# Block algorithm for matrix multiplication

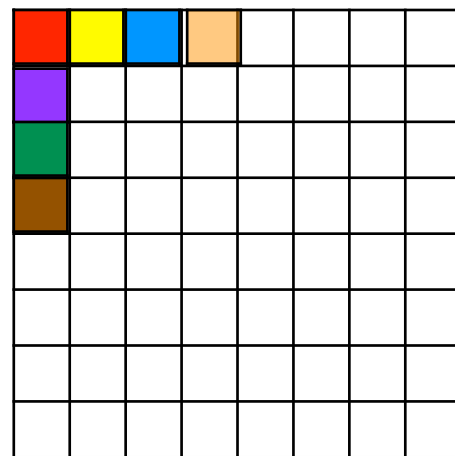
- Discover the cache miss rate
  - `valgrind --tool=cachegrind cmd`
    - cachegrind is a tool profiling the cache performance
- Performance counter
  - Intel® Performance Counter Monitor <http://www.intel.com/software/pcm/>

# Block algorithm for matrix multiplication

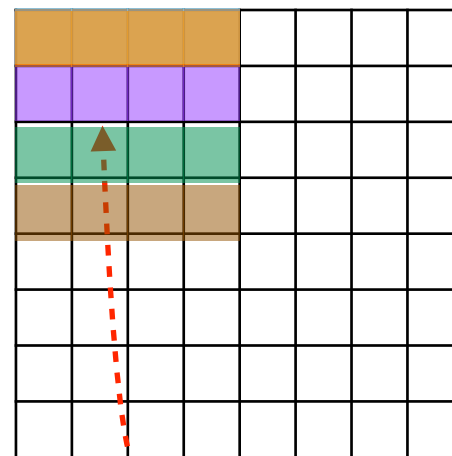
```
for(i = 0; i < ARRAY_SIZE; i++) {  
  for(j = 0; j < ARRAY_SIZE; j++) {  
    for(k = 0; k < ARRAY_SIZE; k++) {  
      c[i][j] += a[i][k]*b[k][j];  
    }  
  }  
}
```



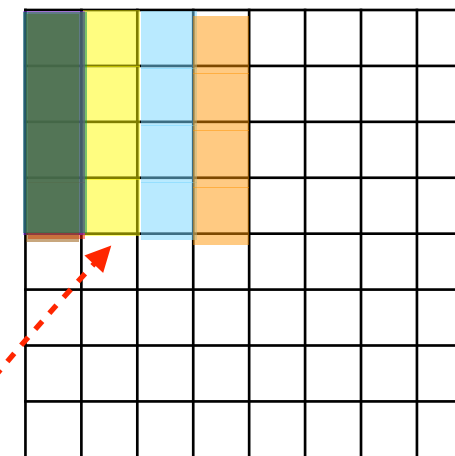
```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
            c[ii][jj] += a[ii][kk]*b[kk][jj];  
    }  
  }  
}
```



c



a

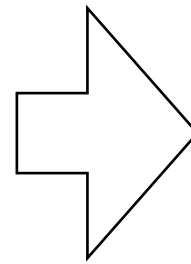


b

**You only need to hold these  
sub-matrices in your cache**

# Matrix Transpose

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```



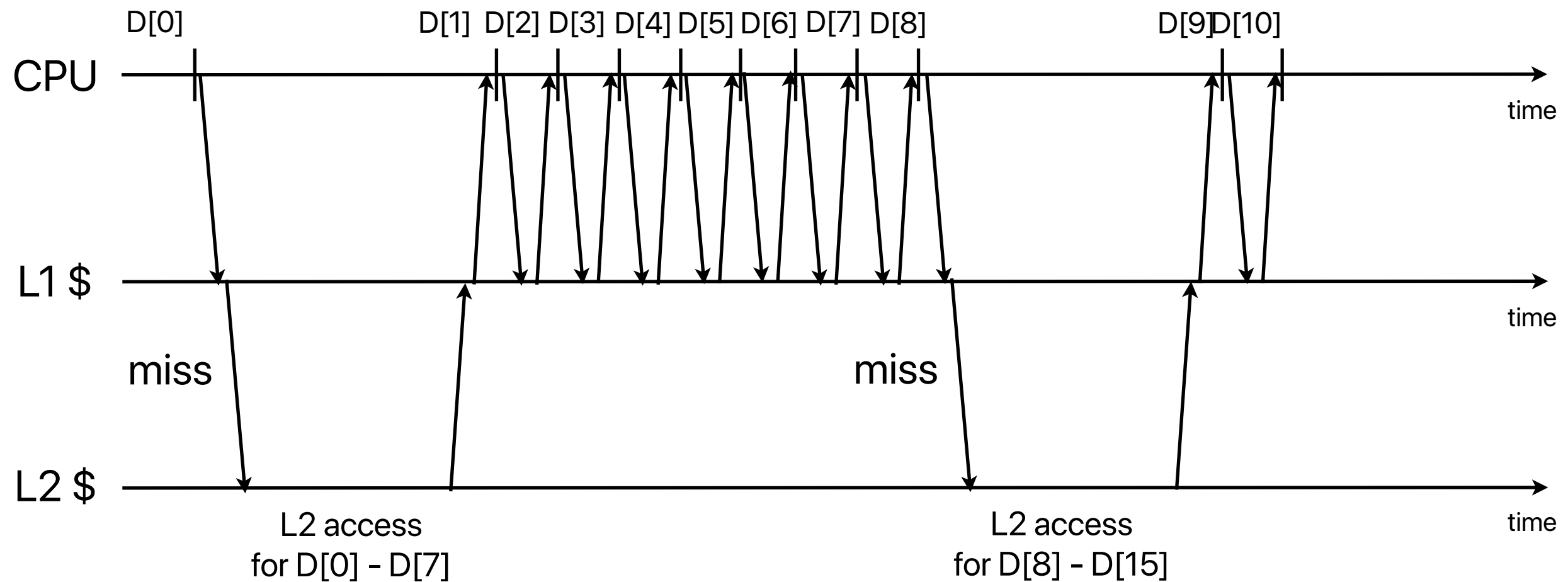
```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```



# Prefetching

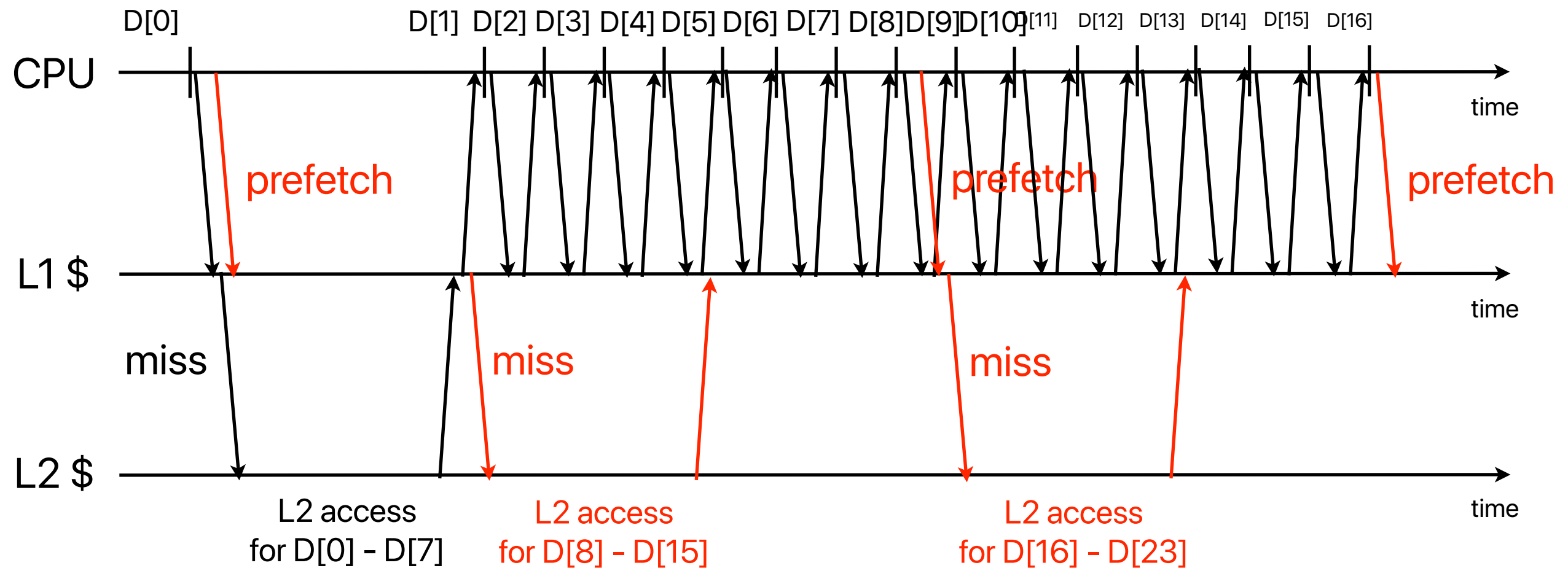
# Characteristic of memory accesses

```
for(i = 0; i < 1000000; i++) {  
    D[i] = rand();  
}
```



# Prefetching

```
for(i = 0; i < 1000000; i++) {  
    D[i] = rand();  
    // prefetch D[i+8] if i % 8 == 0  
}
```



# Prefetching

- Identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
  - Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction
- Hardware prefetch
  - The processor can keep track the distance between misses. If there is a pattern, fetch  $\text{miss\_data\_address} + \text{distance}$  for a miss
- Software prefetch
  - Load data into X0
  - Using prefetch instructions

# Demo

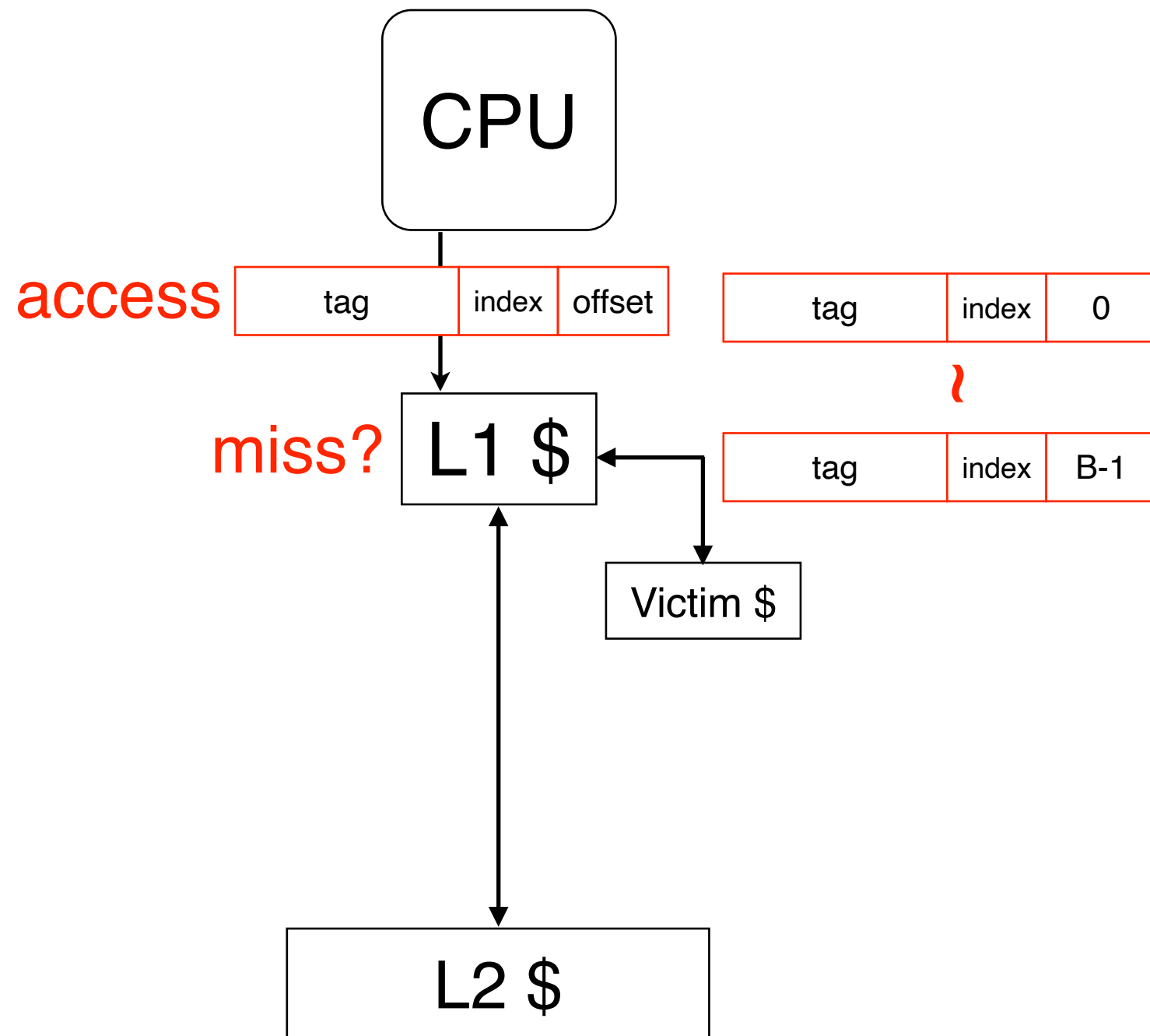
- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag `"-fprefetch-loop-arrays"` to automatically insert software prefetch instructions

# **Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers**

**Norman P. Jouppi**



# Victim cache



- A small cache that captures the evicted blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Swap the entry if hit in victim cache
  - Athlon has an 8-entry victim cache
- Reduce conflict misses
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache



# Victim cache v.s. miss caching

- Both of them improves conflict misses
- Victim cache can use cache block more efficiently — swaps when miss
  - Miss caching maintains a copy of the missing data — the cache block can both in L1 and miss cache
  - Victim cache only maintains a cache block when the block is kicked out
- Victim cache captures conflict miss better
  - Miss caching captures every missing block

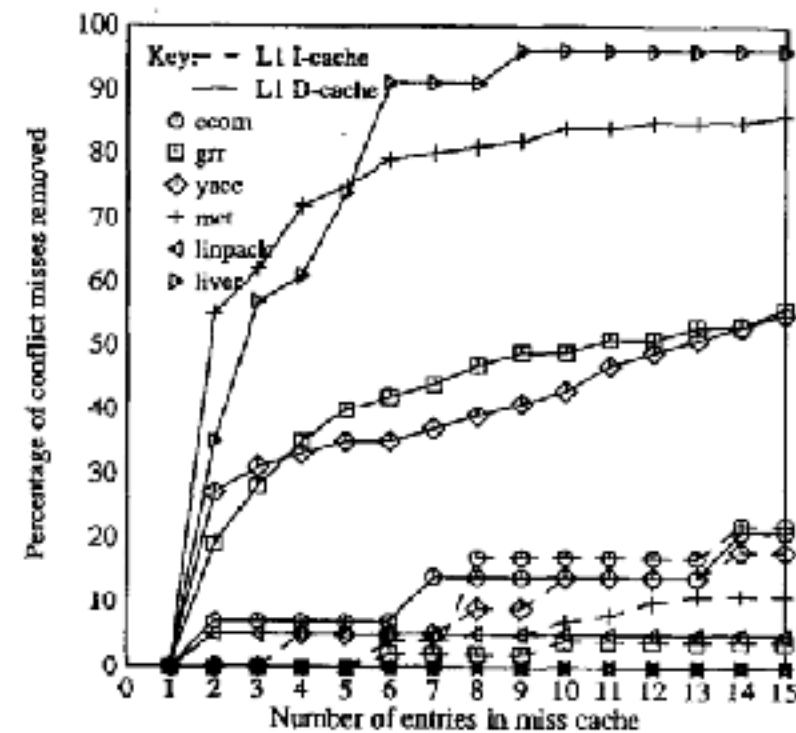


Figure 3-3: Conflict misses removed by miss caching

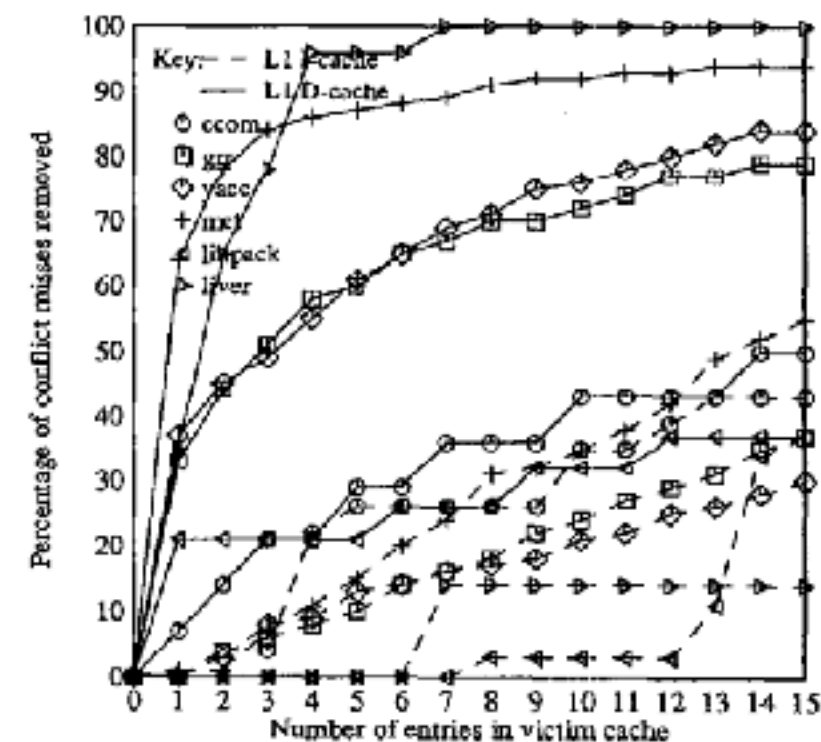
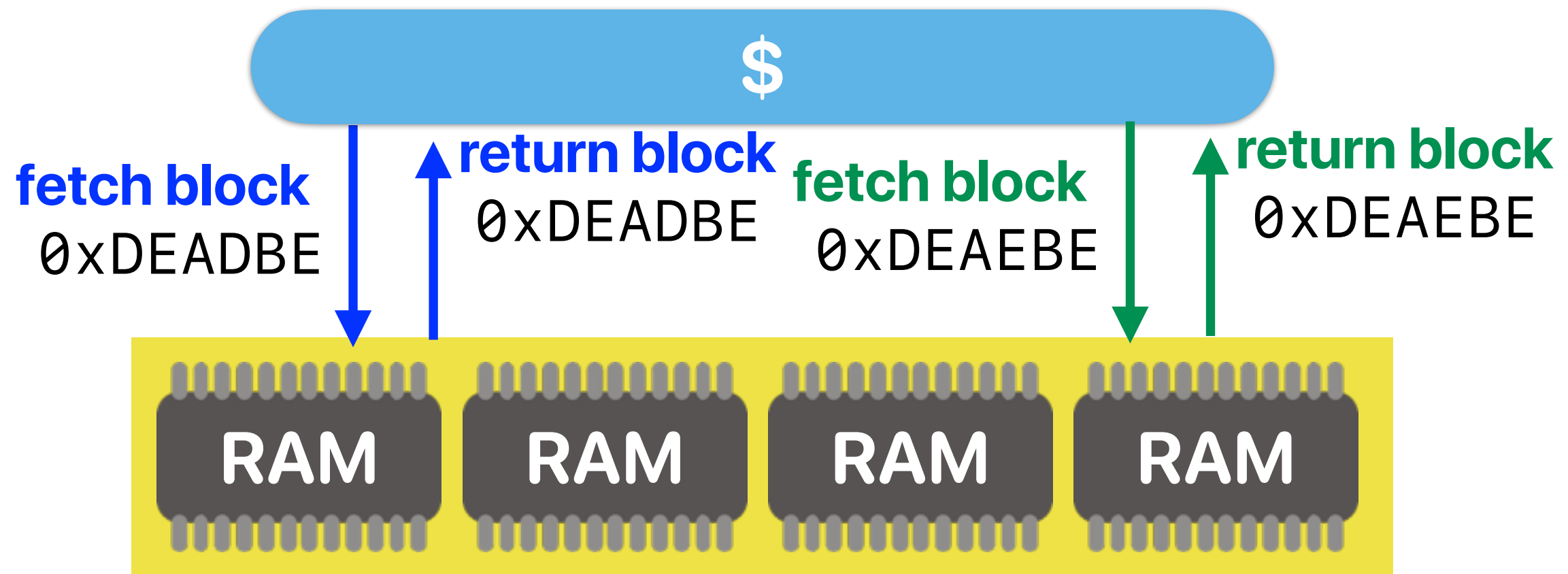


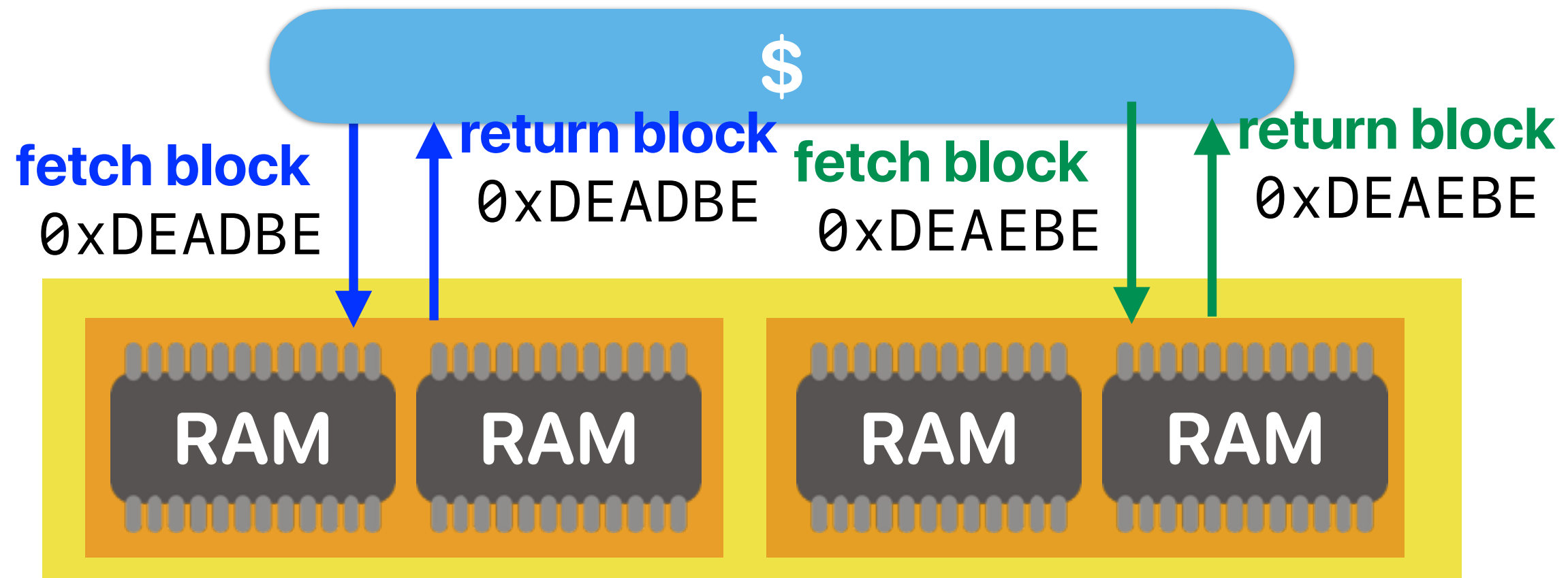
Figure 3-5: Conflict misses removed by victim caching

# **Advanced Hardware Techniques in Improving Memory Performance**

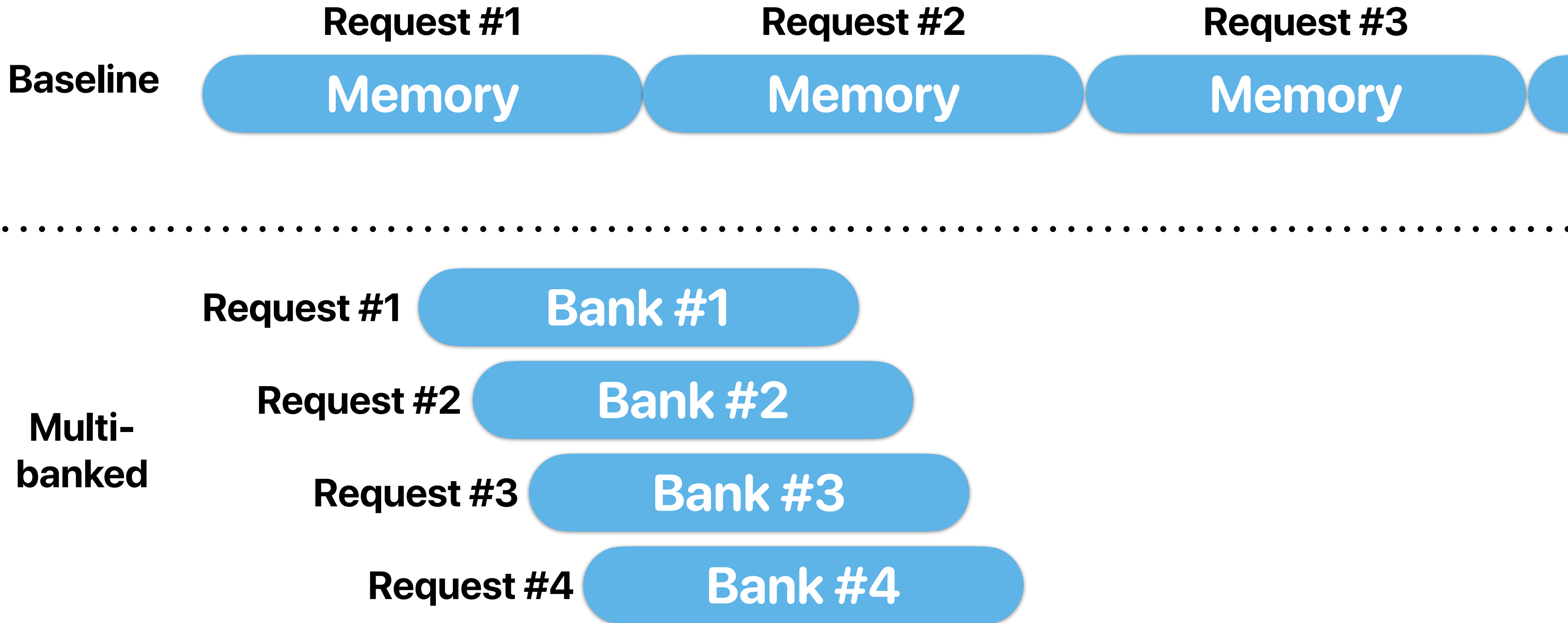
# Without banks



# Multibanks & non-blocking caches



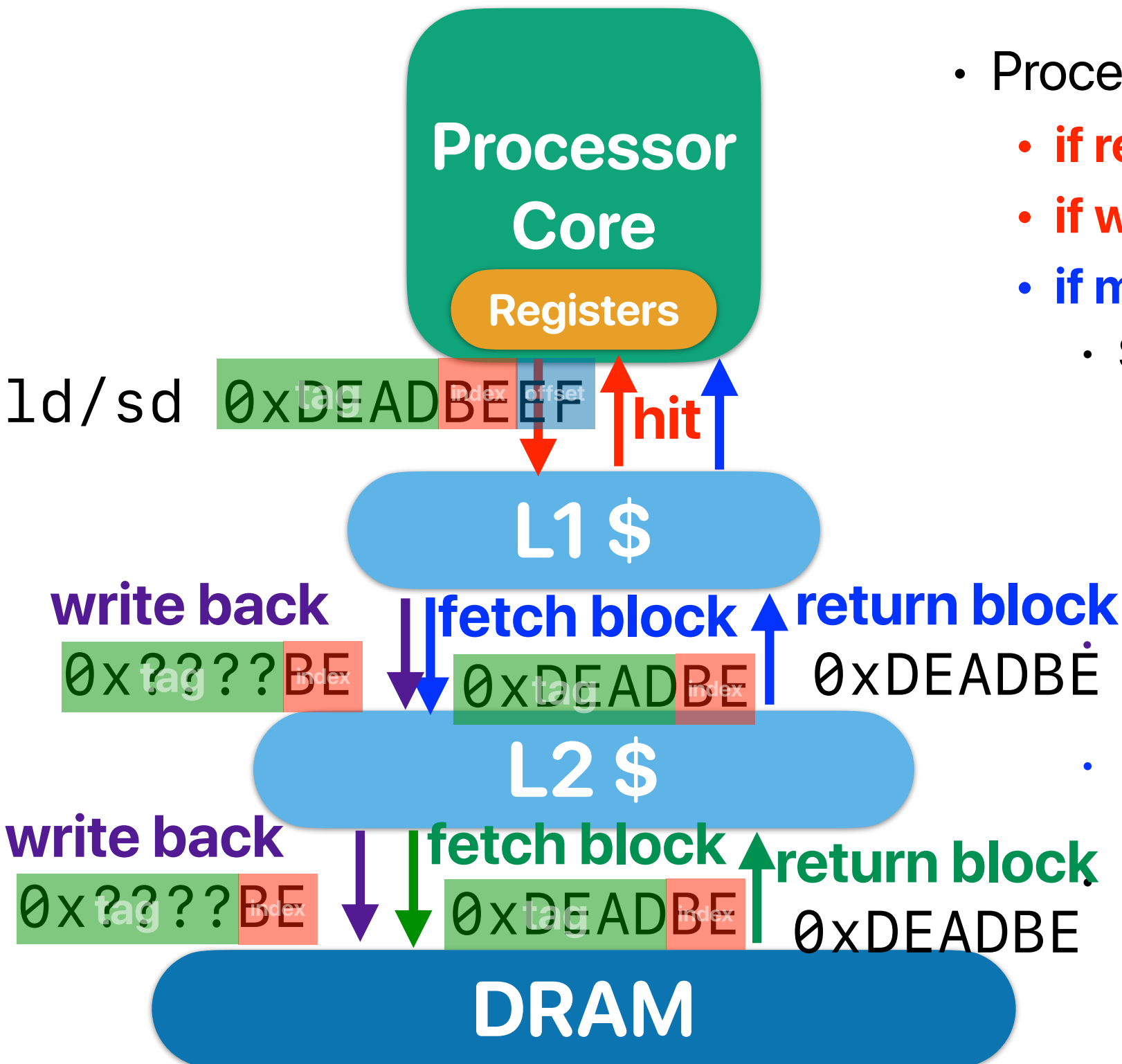
# Pipelined access and multi-banked caches



# Early Restart and Critical Word First

- Don't wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Most useful with large blocks
- Spatial locality is a problem; often we want the next sequential word soon, so not always a benefit (early restart).

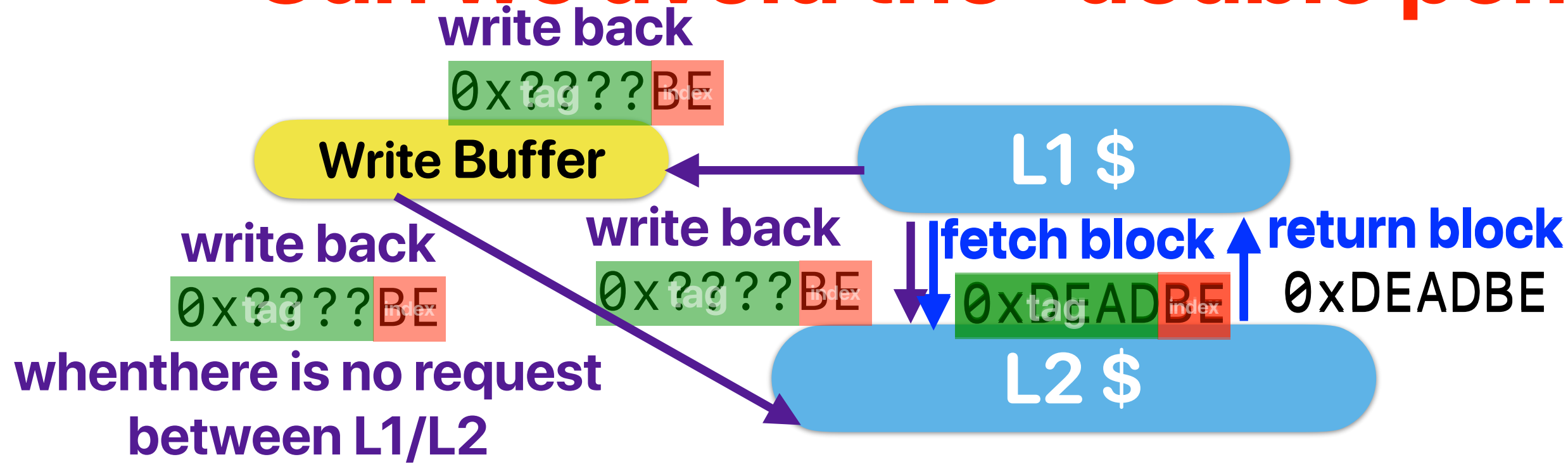
# What happens when we access data



- Processor sends load request to L1-\$
  - **if read hit — return data**
  - **if write hit — set dirty and update in the block**
  - **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



# Can we avoid the "double penalty"?



- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.