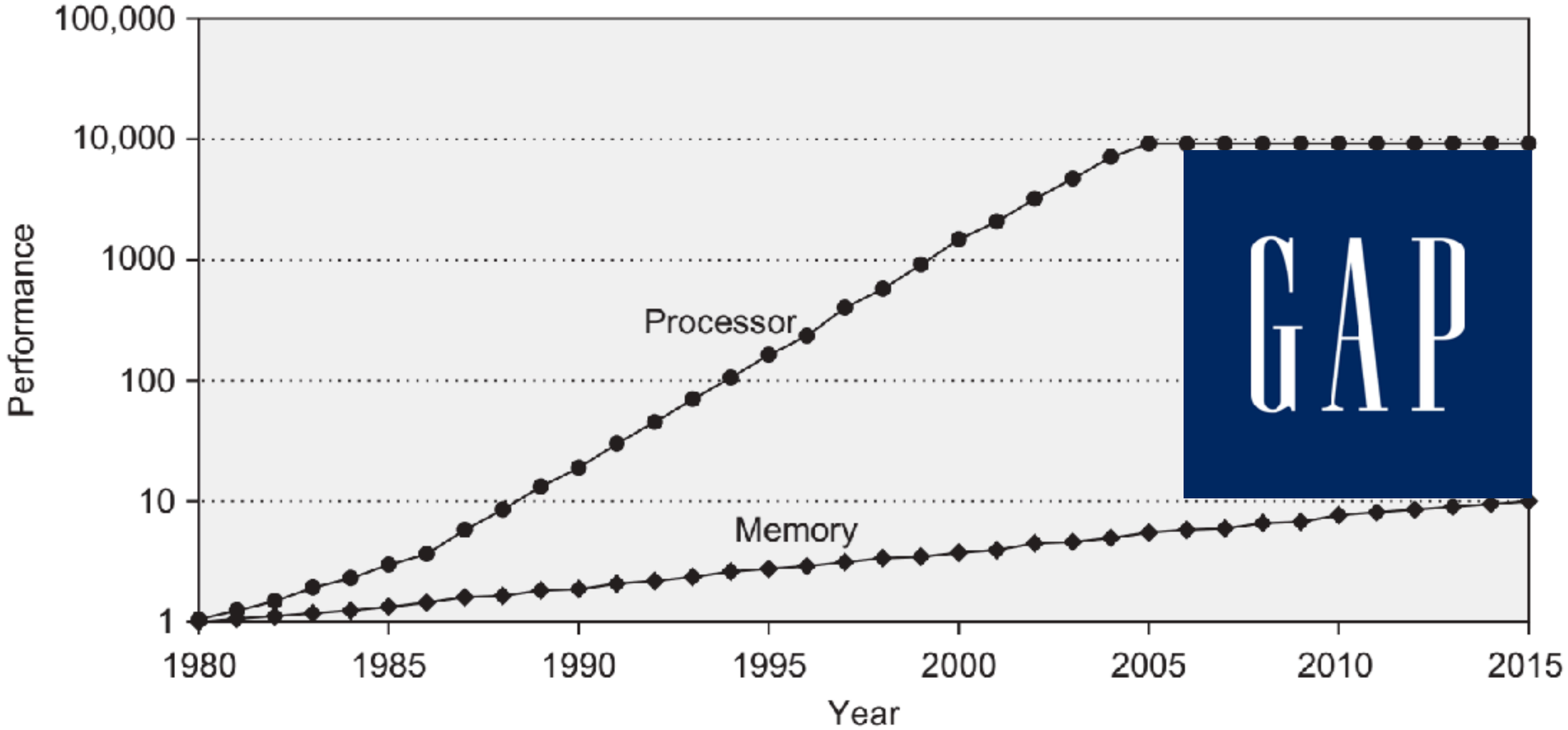
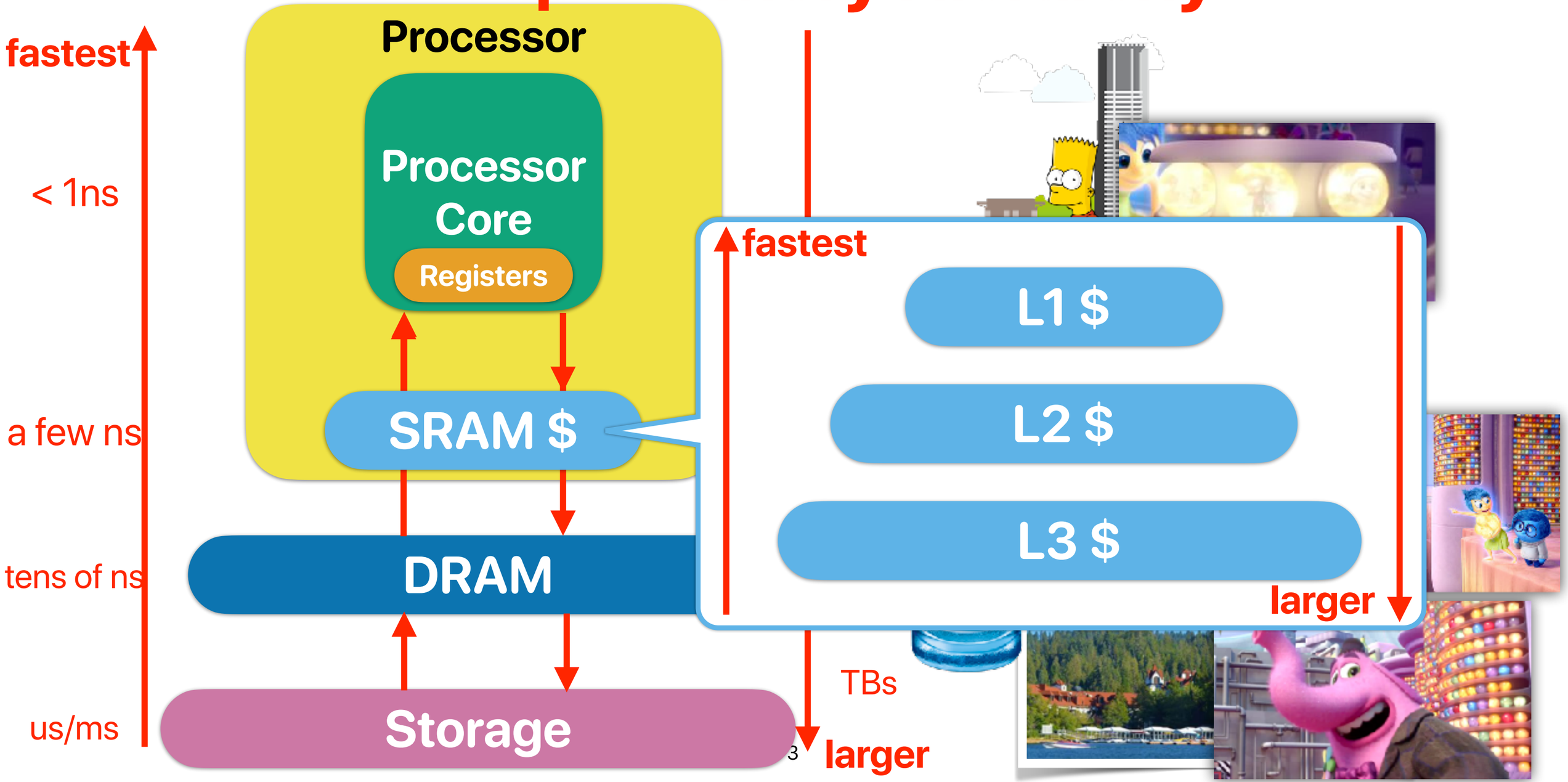
# Memory Hierarchy (III): Optimizing Cache Performance

Hung-Wei Tseng

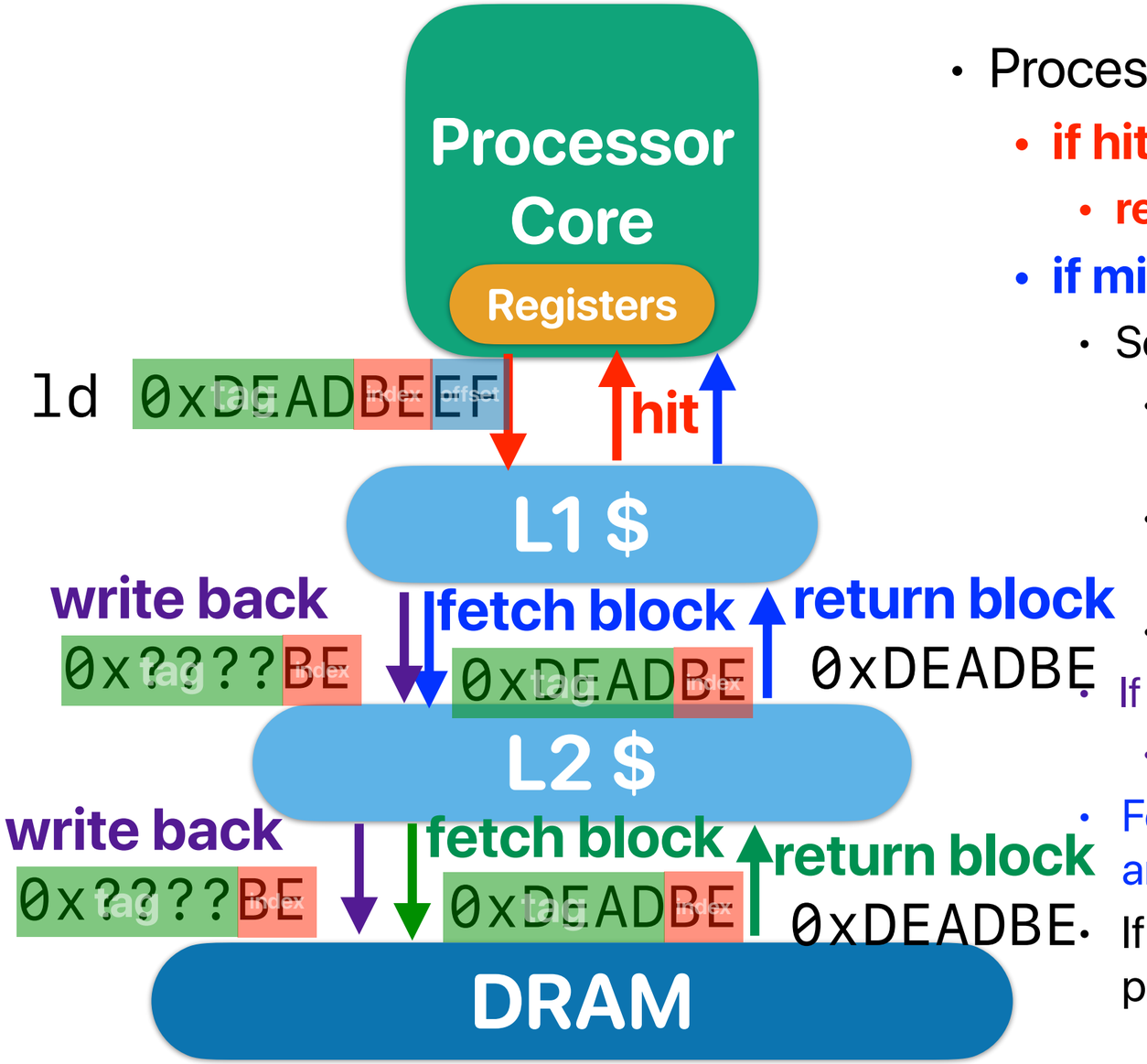
# Performance gap between Processor/Memory



# Recap: Memory Hierarchy

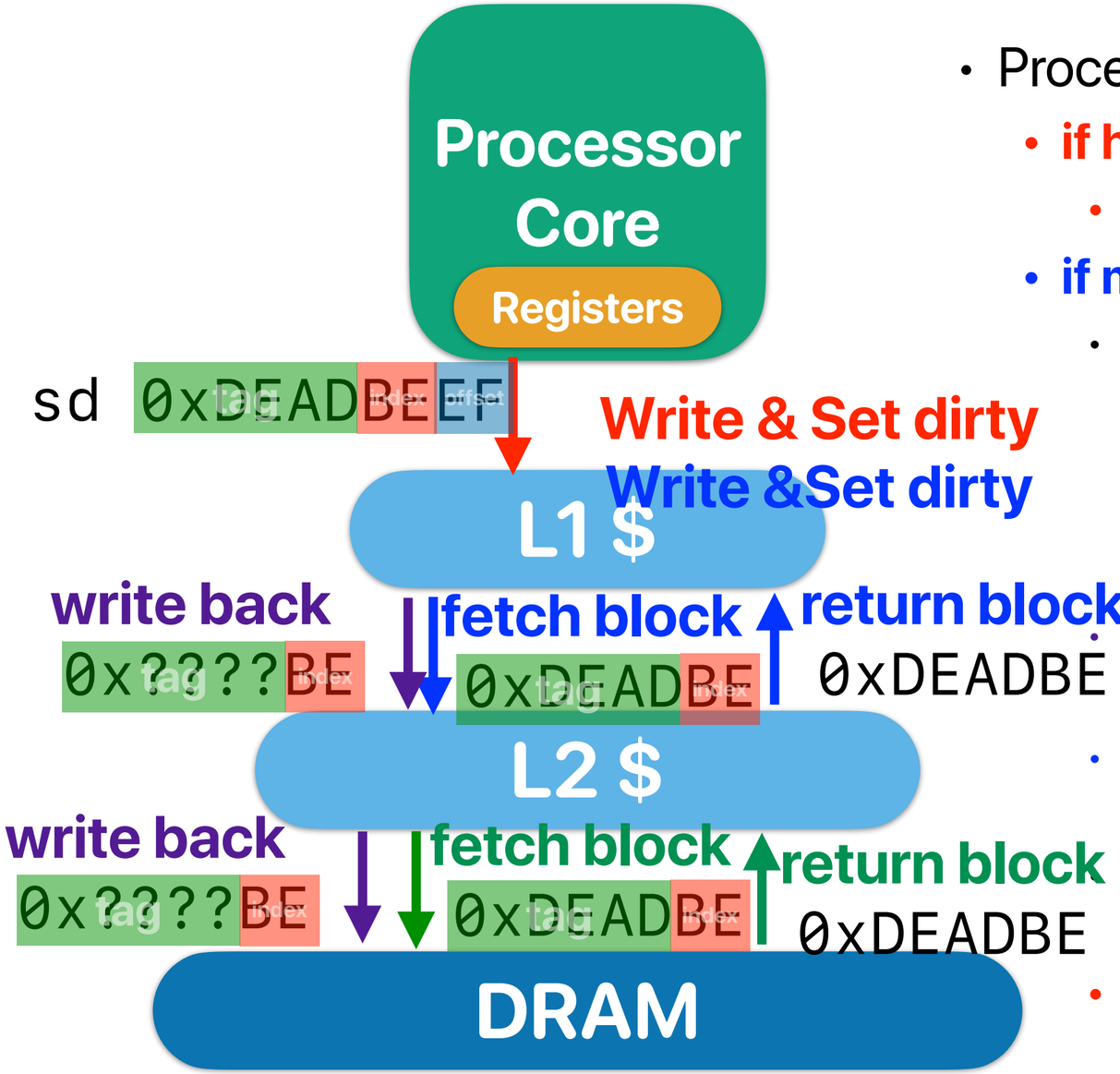


# What happens when we read data



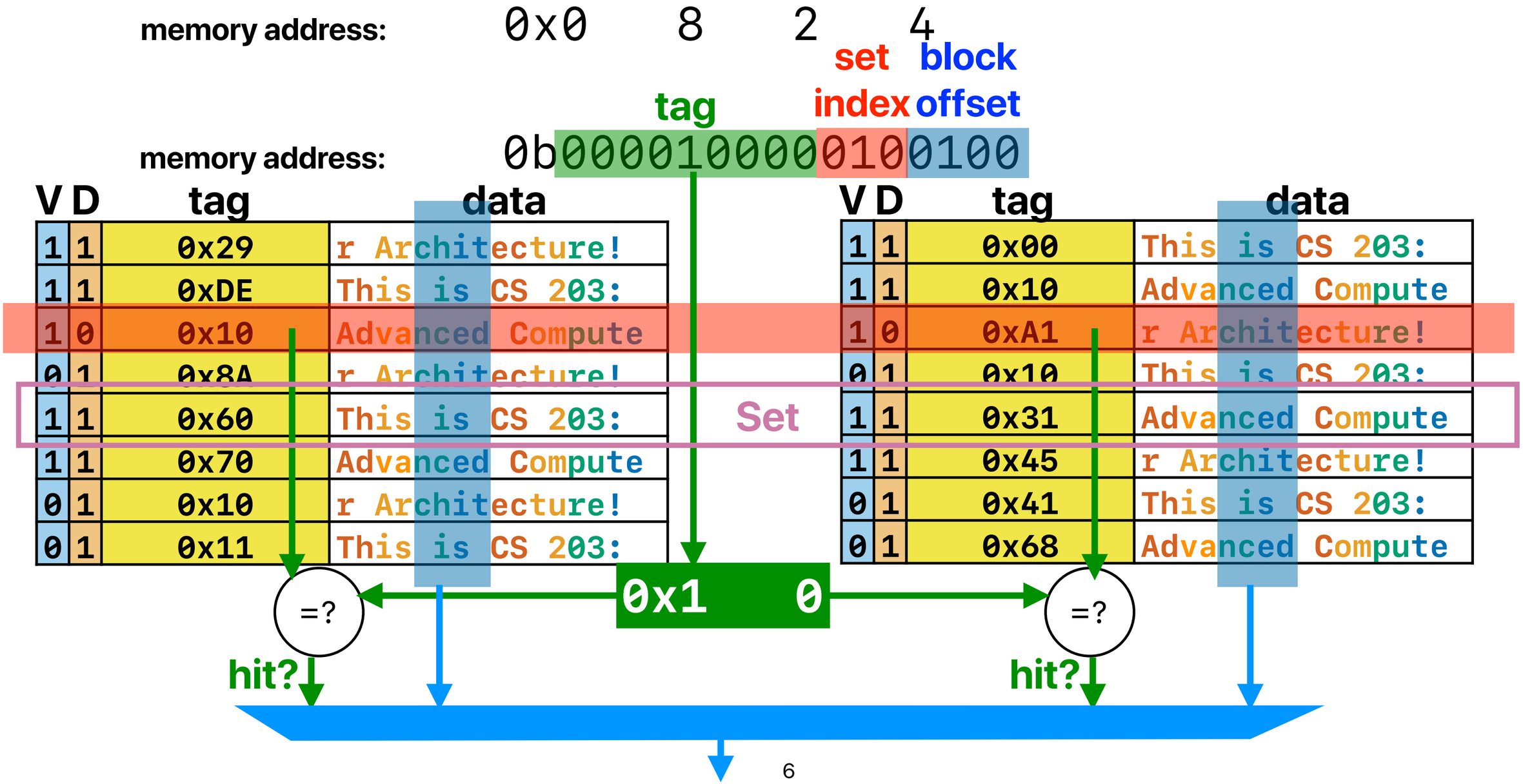

- Processor sends load request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - 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**

# Recap: Way-associative cache



# 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

# AMD Phenom II

**100% miss rate!**

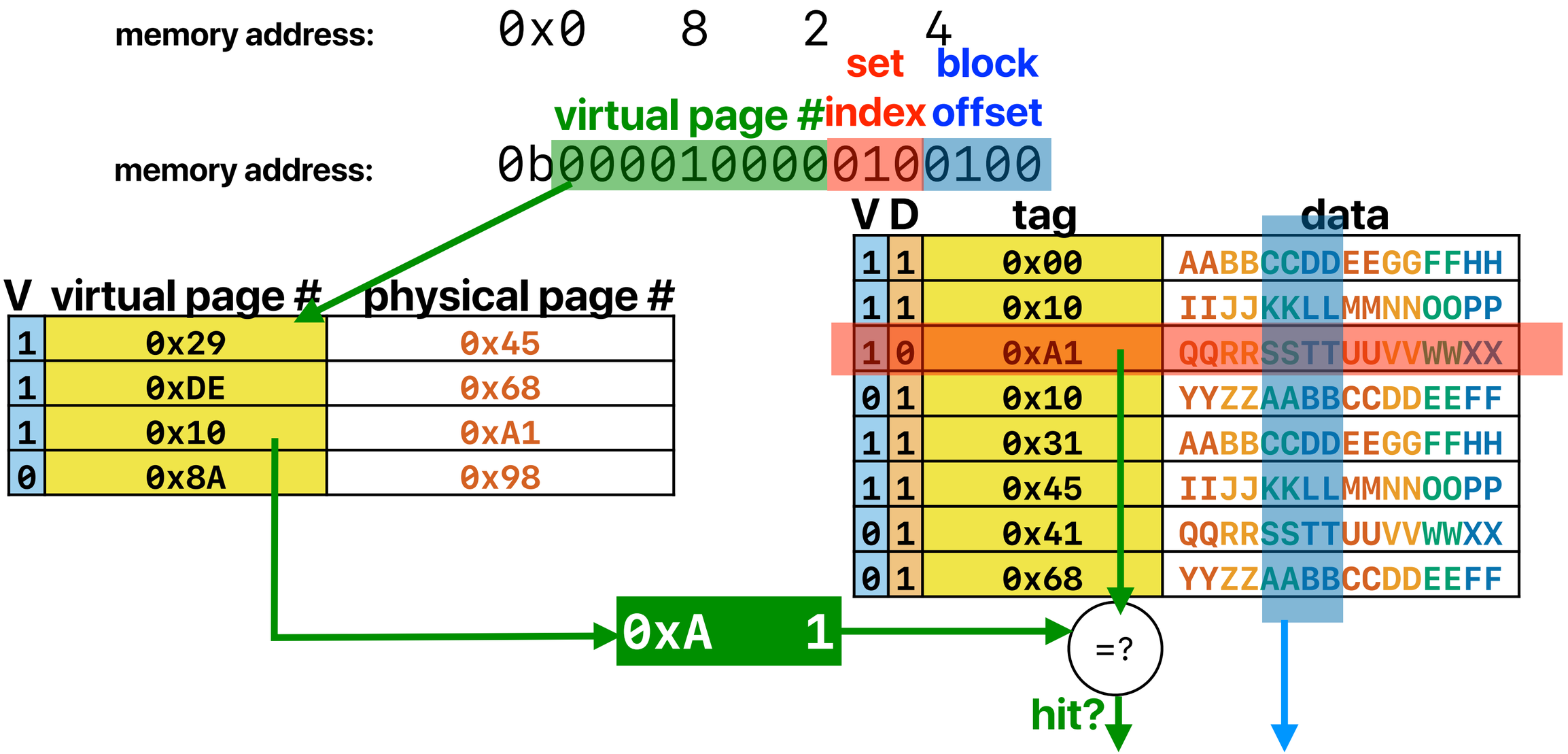
- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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[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 |

# Virtually indexed, physically tagged cache



# Team scores

| | | | |
|---|---|---|---|
| 3.5 | 3.5 | 3.5 | 4 |

# Outline

- Architectural optimizations for cache performance
- Programmer’s optimizations for cache performance

# Basic Hardware Optimization in Improving 3Cs

Poll close in 1:30

# 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

Poll close in 1:30

# 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

# 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

# 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 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

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

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

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

**Norman P. Jouppi**

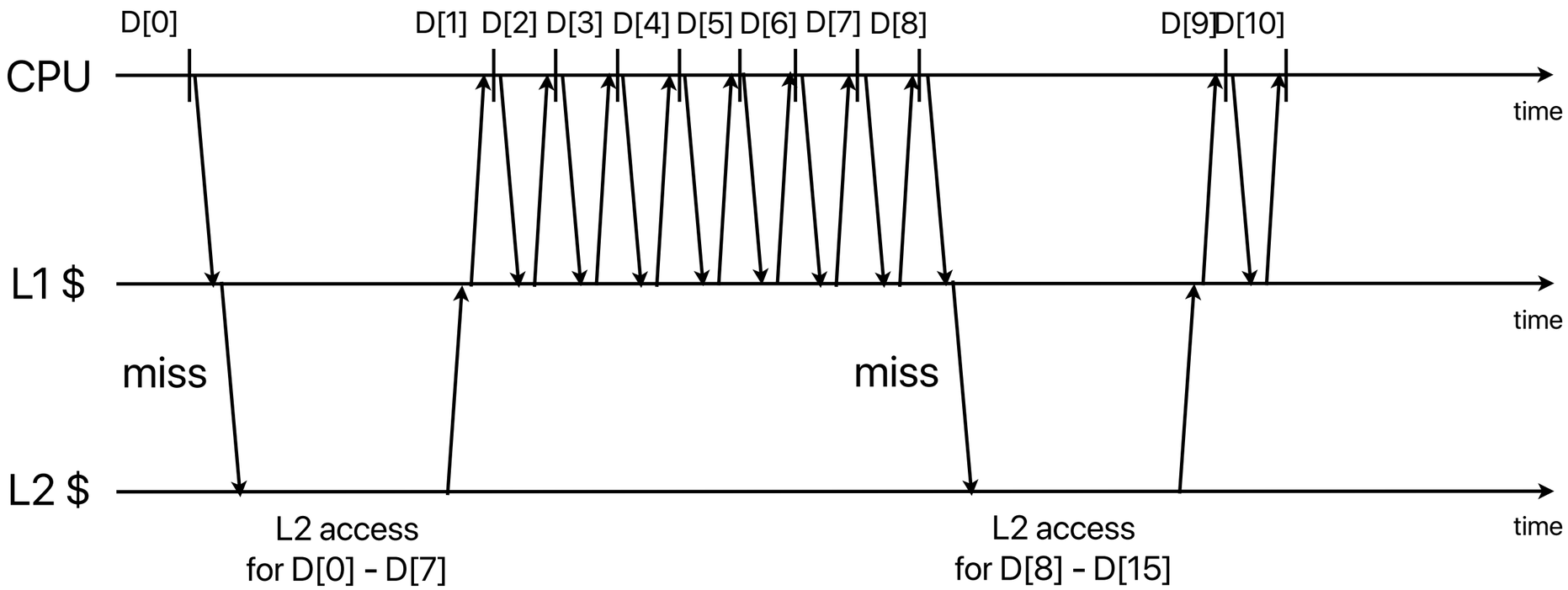
# Which of the following schemes can help AMD Phenom II?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help AMD Phenom II for the code in the previous slide?
  - ① Missing cache
  - ② Victim cache
  - ③ Prefetch
  - ④ Stream buffer
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

# Prefetching

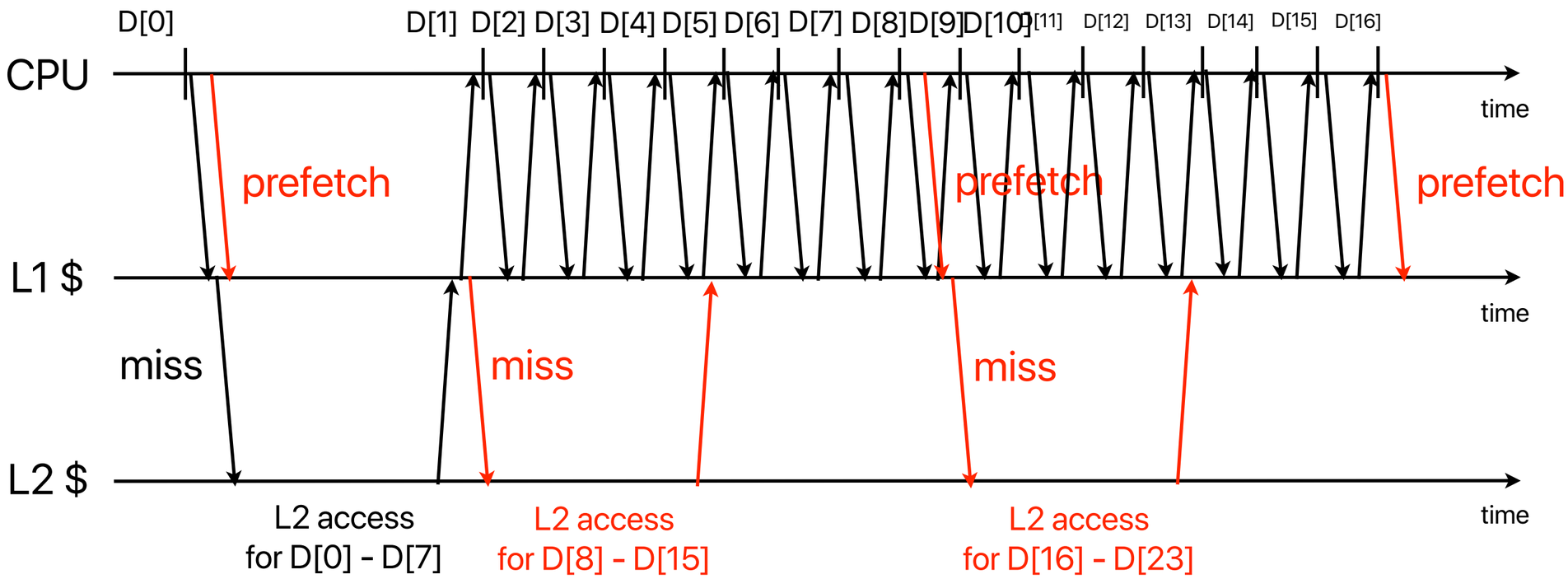
# Characteristic of memory accesses

```
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 miss_data_address+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

Poll close in 1:30

# Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)
```
while(node){
    node = node->next;
}
```

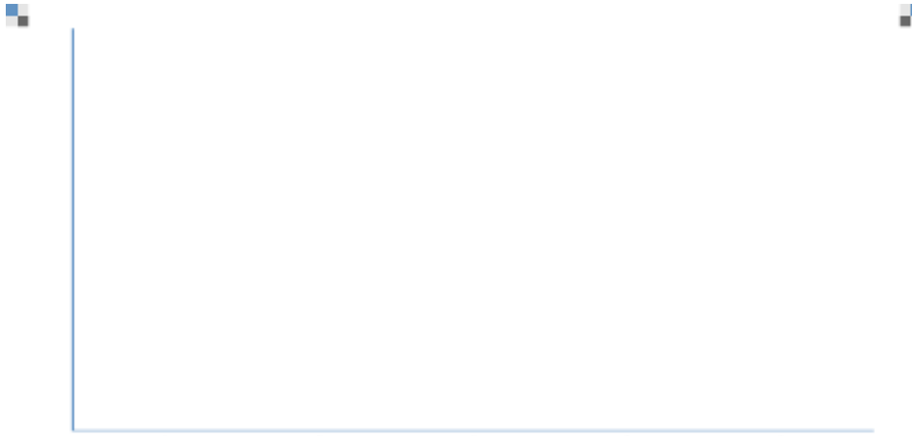
(2)
```
while(++i<100000)
    a[i]=rand();
```

(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        else if (key < root->data)
            root = root->left;
        else
            return true;
}
```

(4)
```
    for (i = 0; i < 65536; i++) {
      mix_i = ((i * 167) + 13) & 65536;
      results[mix_i]++;
    }
```

A. 0
B. 1
C. 2
D. 3
E. 4



Poll close in 1:30

# Where can prefetch work effectively


- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)
```
while(node){
     node = node->next;
}
```

(2)
```
while(++i<100000)
     a[i]=rand();
```

(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        else if (key < root->data)
            root = root->left;
        else
            return true;
}
```

(4)
```
    for (i = 0; i < 65536; i++) {
      mix_i = ((i * 167) + 13) & 65536;
      results[mix_i]++;
    }
```

A. 0
B. 1
C. 2
D. 3
E. 4

# Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)
```
while(node){
    node = node->next;
}
```
— where the next pointing to is hard to predict

(2)
```
while(++i<100000)
    a[i]=rand();
```

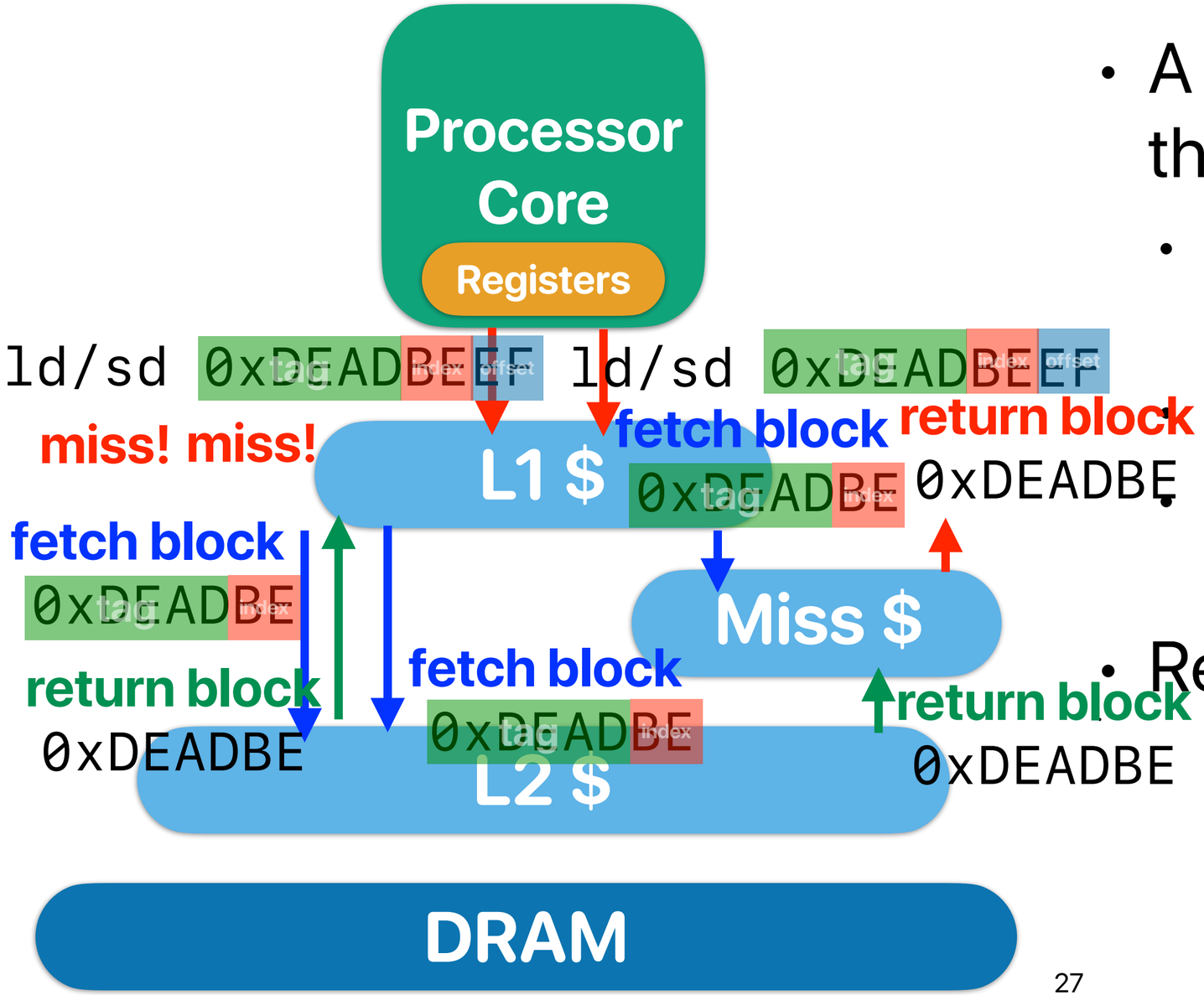
(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        else if (key < root->data)
            root = root->left;
        else
            return true;
}
```
— where the next node is also hard to predict

(4)
```
    for (i = 0; i < 65536; i++) {
      mix_i = ((i * 167) + 13) & 65536;
      results[mix_i]++;
    }
```
— the stride to the next element is hard to predict…

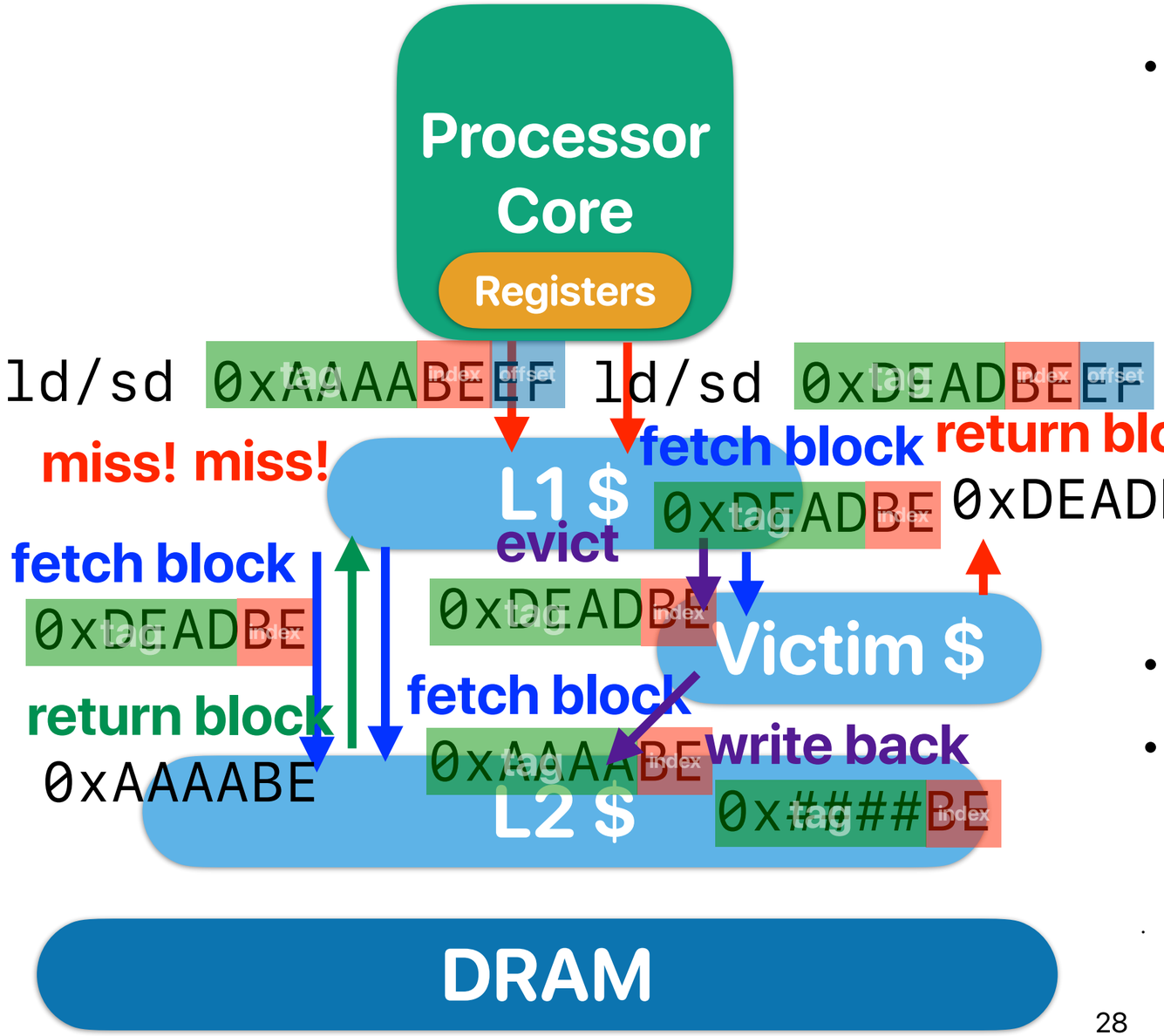
A. 0
B. 1
C. 2
D. 3
E. 4

# Miss cache




- A small cache that captures the missing blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the missing cache
- Reduce conflict misses

# 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/Phenom 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

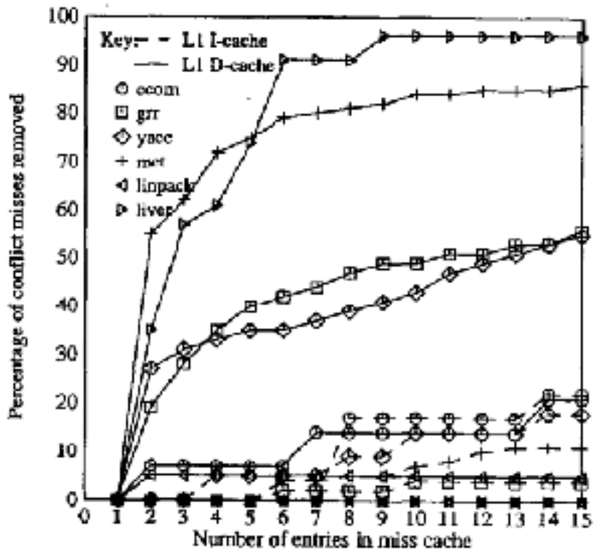



**Figure 3-3:** Conflict misses removed by miss caching

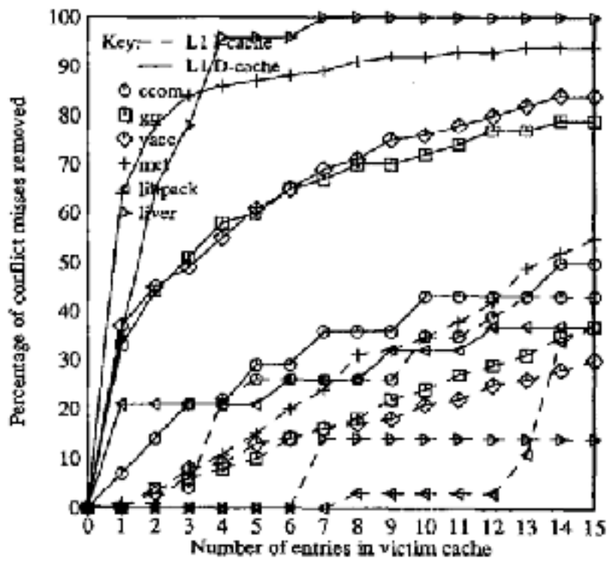


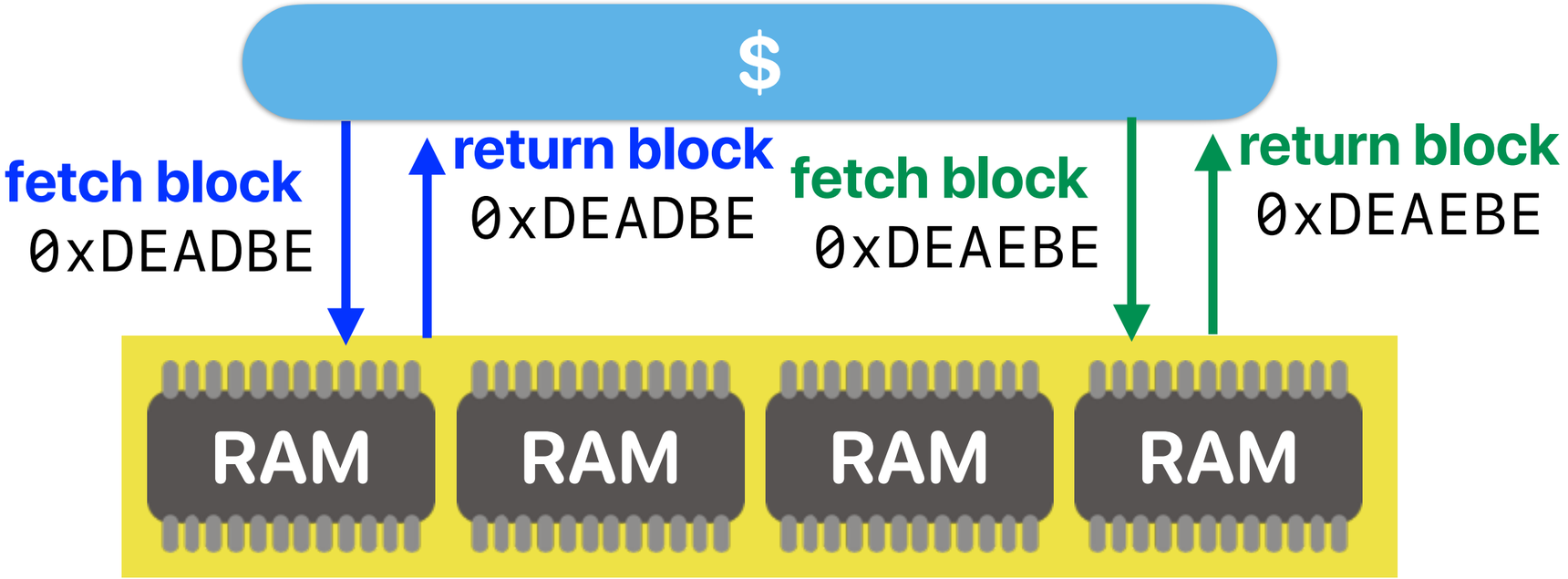

**Figure 3-5:** Conflict misses removed by victim caching

# Which of the following schemes can help Athlon 64?

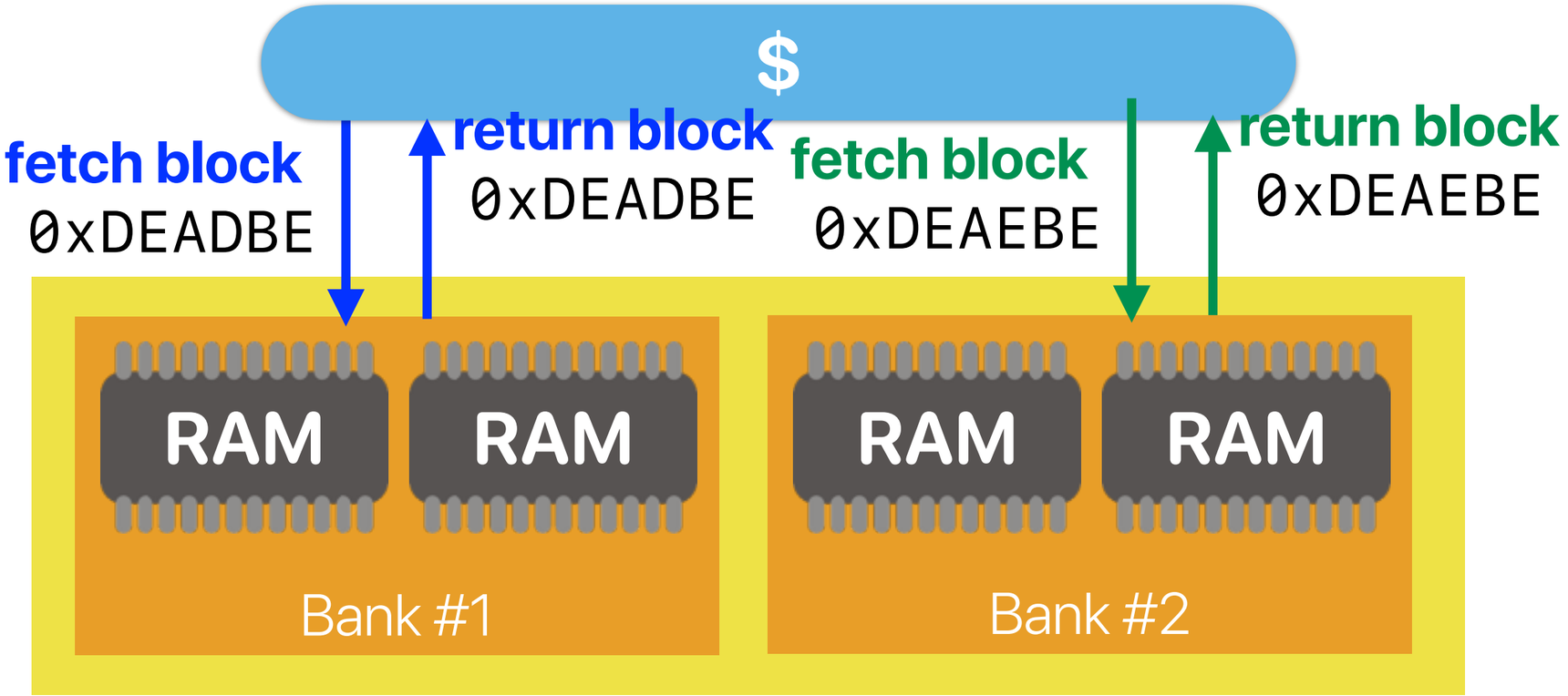
- How many of the following schemes mentioned in "improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers" would help AMD Phenom II for the code in the previous slide?
  - ① Missing cache — help improving conflict misses
  - ② Victim cache — help improving conflict misses
  - ③ Prefetch — improving compulsory misses , but can potentially hurt, if we did not do it right
  - ④ Stream buffer — only help improving compulsory misses
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

# Advanced Hardware Techniques in Improving Memory Performance

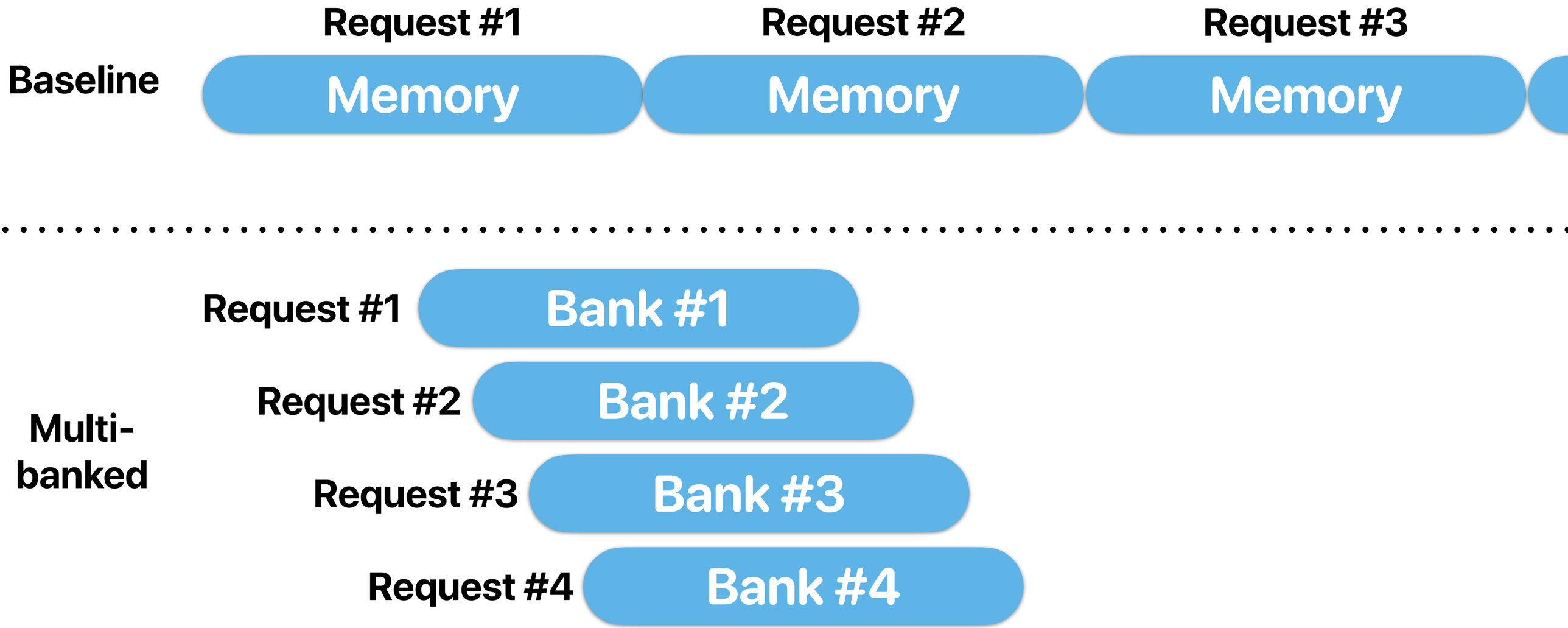
# Blocking cache



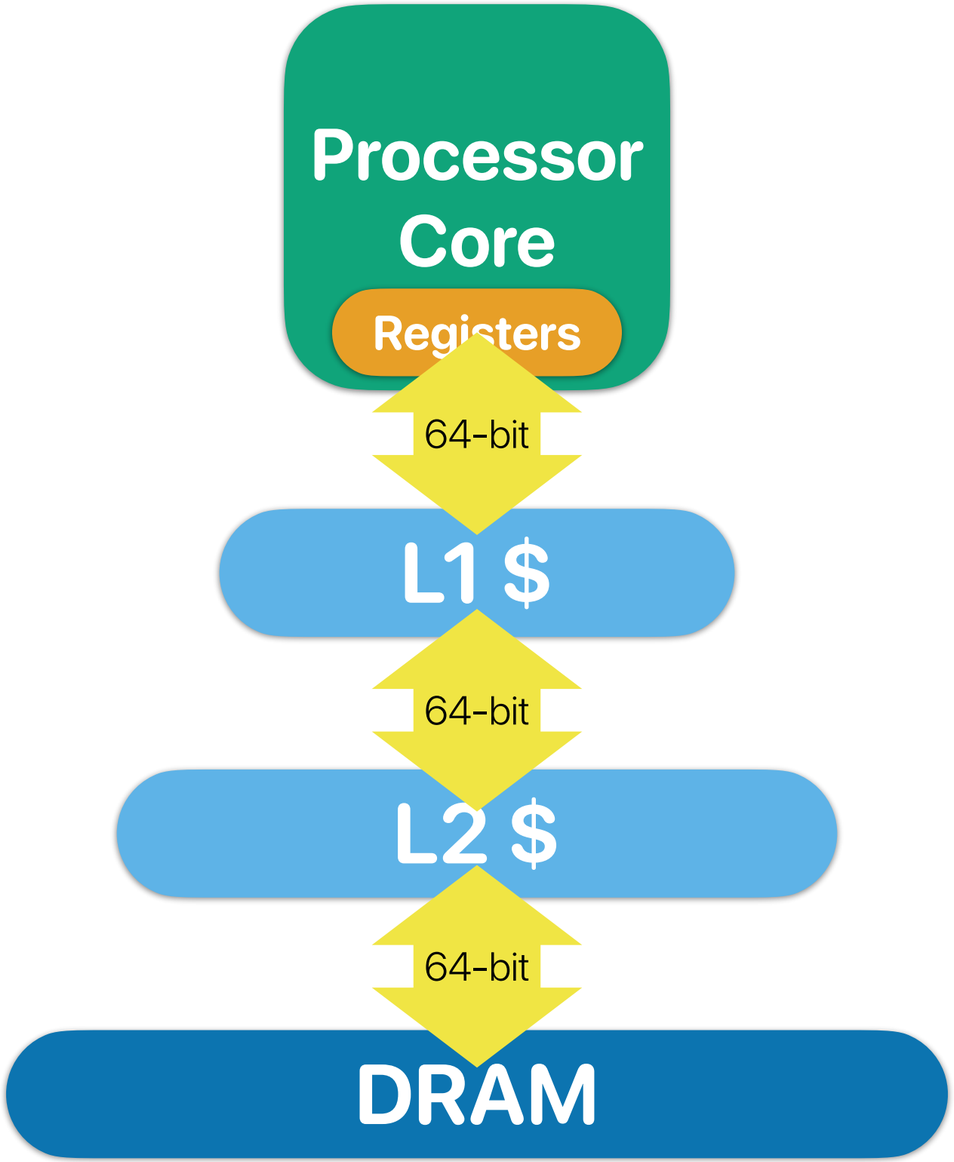
# Multibanks & non-blocking caches



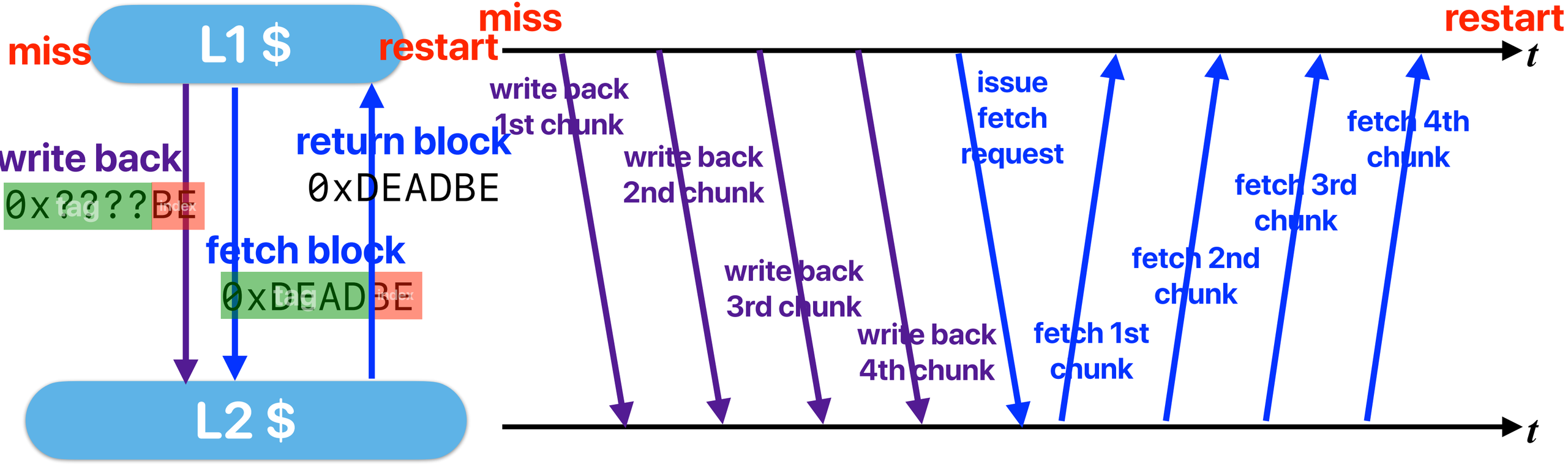
# Pipelined access and multi-banked caches



# The bandwidth between units is limited

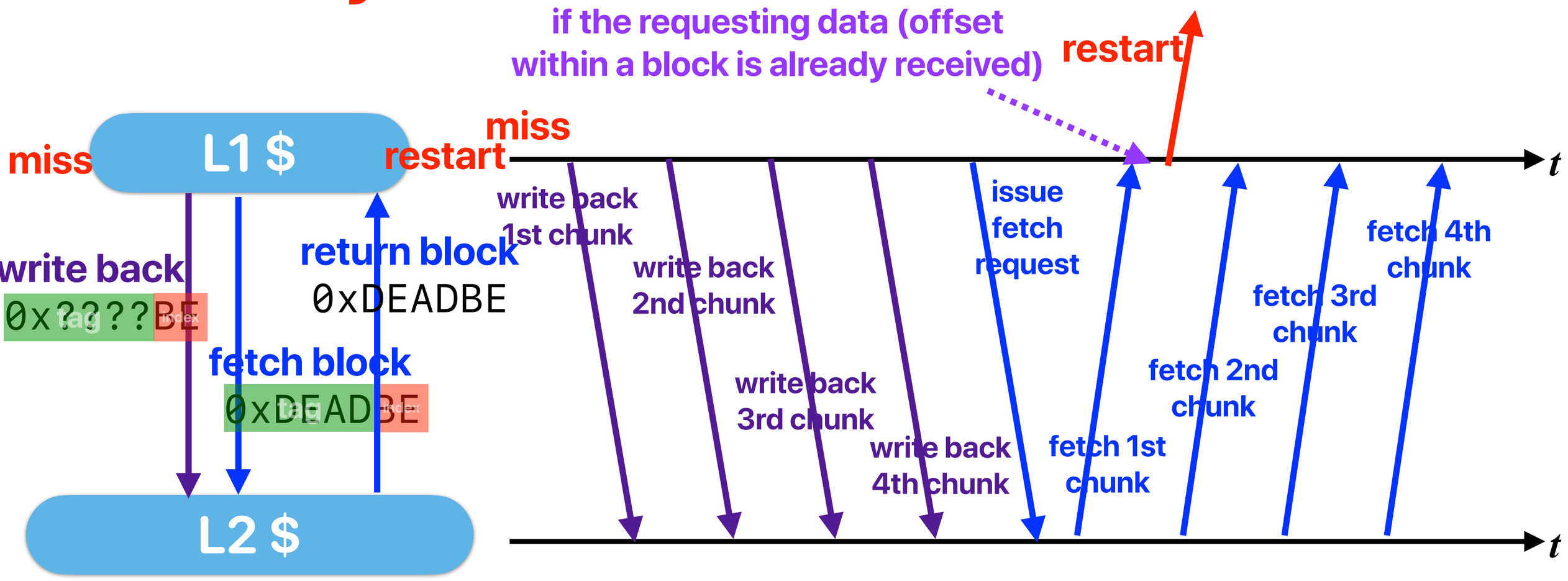


# When we handle a miss




assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Early Restart and Critical Word First

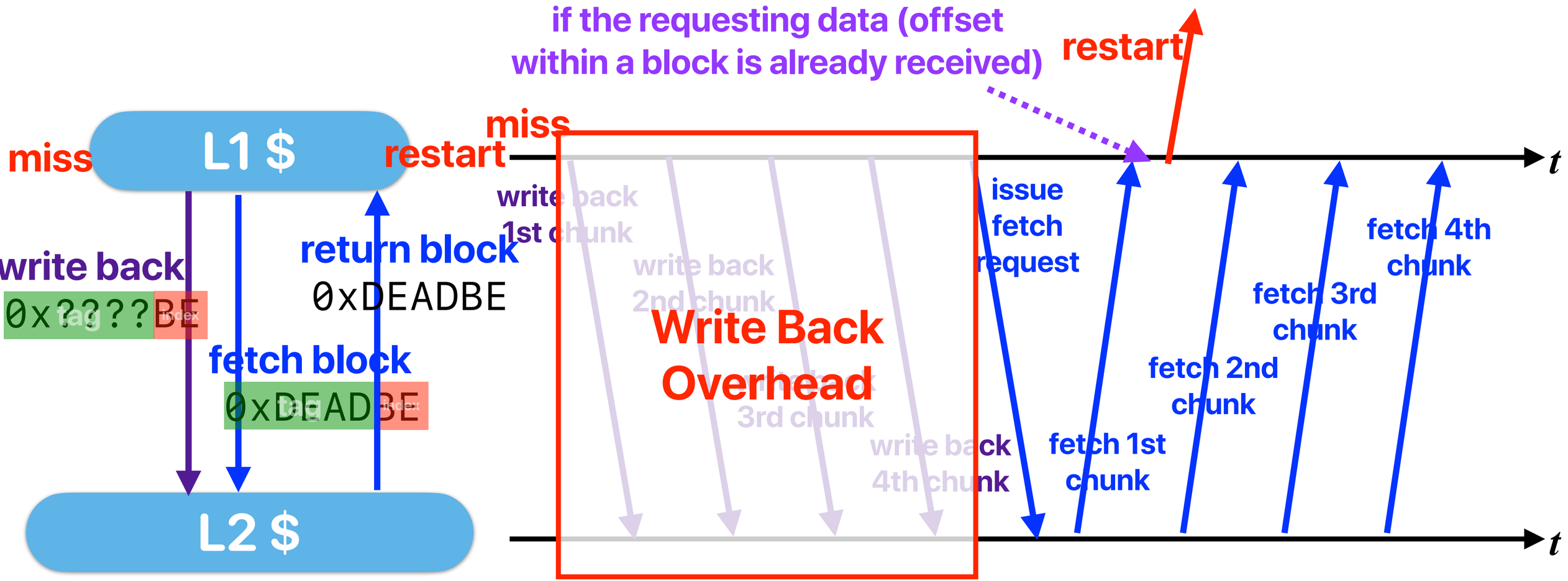



assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Early Restart and Critical Word First

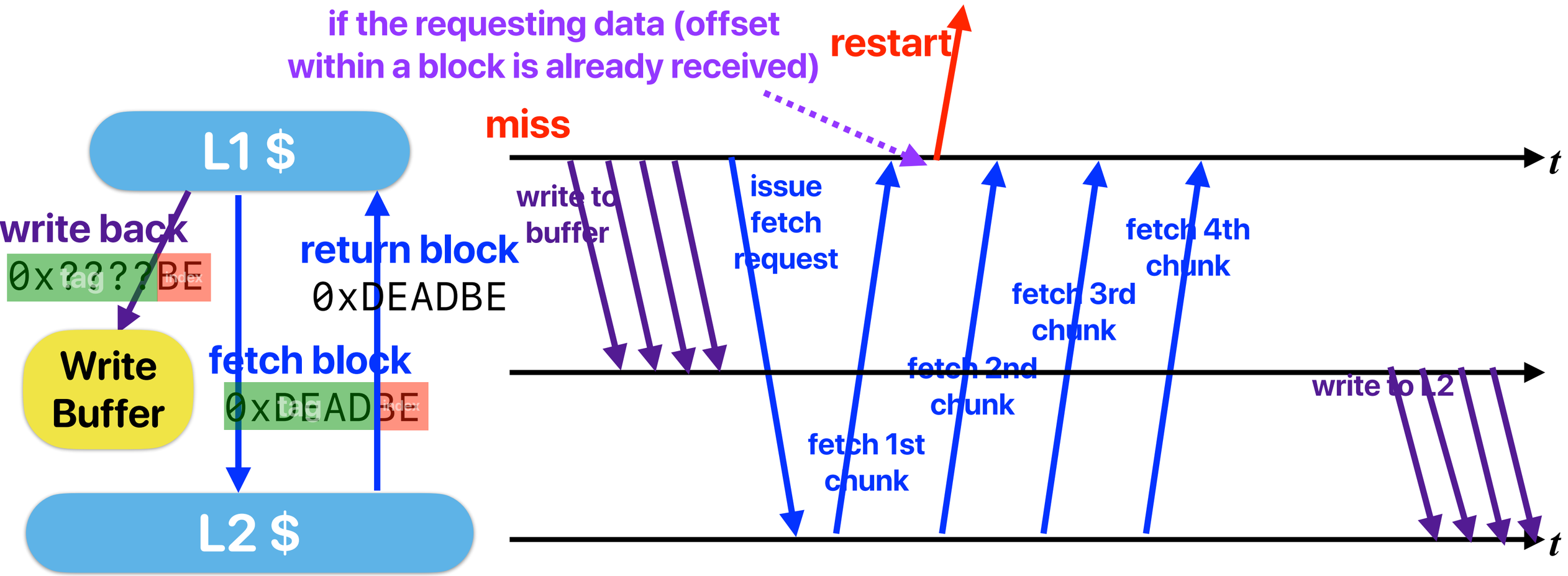
- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let 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).

# Can we avoid the overhead of writes?




assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Write buffer!




assume the bus between L1/L2 only allows a quarter of the cache block go through it

# 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.

Poll close in 1:30

# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Poll close in 1:30

# Summary of Optimizations


- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache **Miss penalty/Bandwidth**
  - ② Critical word first and early restart **Miss penalty**
  - ③ Prefetching **Miss rate (compulsory)**
  - ④ Write buffer **Miss penalty**

  A. 0

  B. 1

  C. 2

  D. 3

  E. 4

# Summary of Optimizations

- Hardware
  - Prefetch — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss panelty

# Programming and memory performance

# Data layout

Poll close in 1:30

# 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

Poll close in 1:30

# The result of `sizeof(struct stude`


- 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

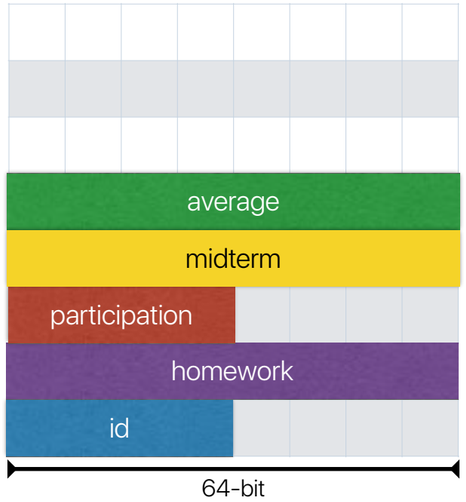
# Memory addressing/alignment

- 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

# Announcement

- Assignment #2 due tonight
- Reading quiz due Wednesday
- Project is up — check the website
- Assignment #3 due next Monday
- Office Hours on Zoom (the office hour link, not the lecture one)
  - Hung-Wei/Prof. Usagi: M 8p-9p, W 2p-3p
  - Quan Fan: F 1p-3p

Computer
Science &
Engineering

203

つづく