

Memory Hierarchy

Hung-Wei Tseng

Disney · PIXAR

INSIDE OUT

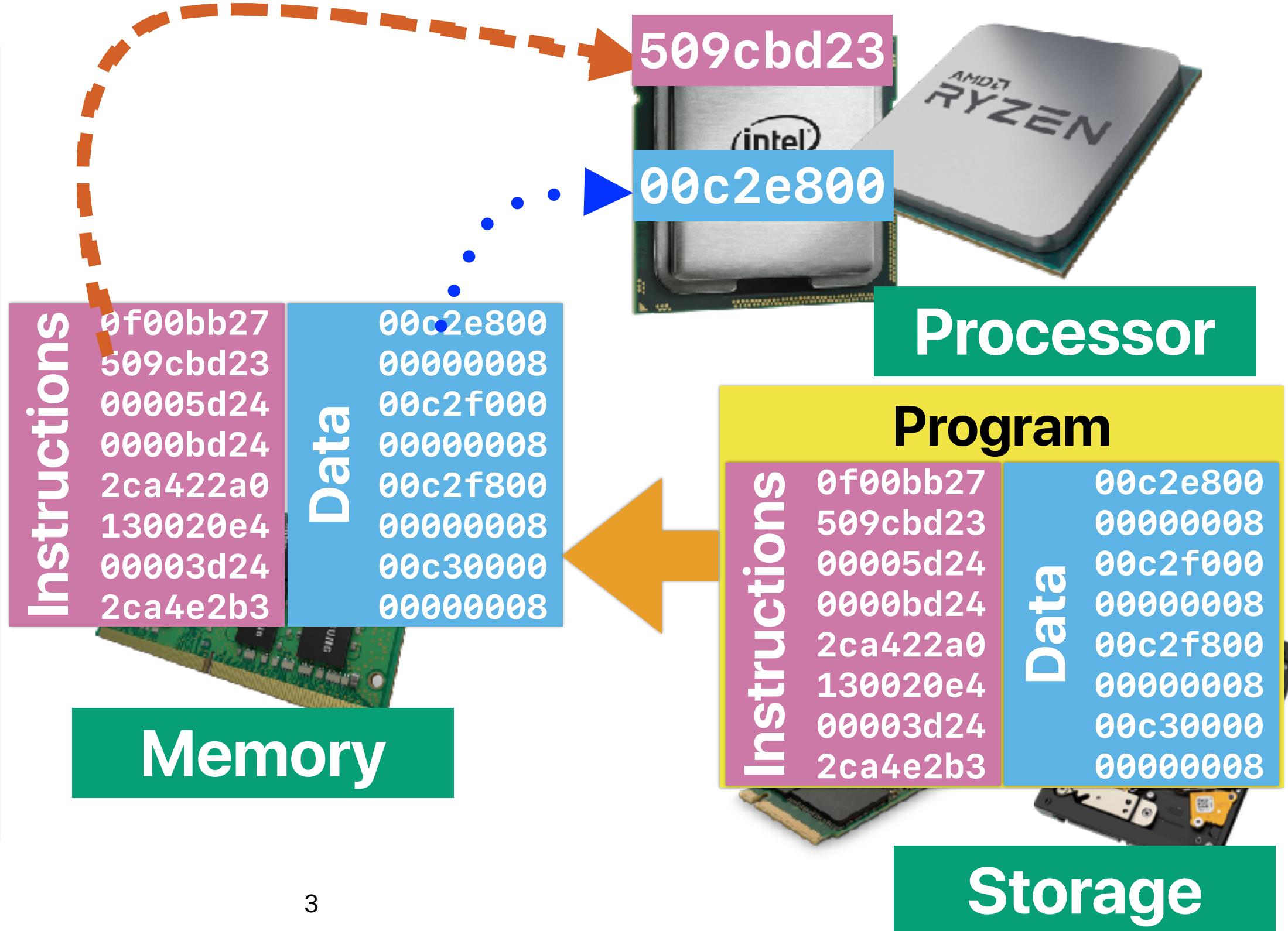
[GET DISNEY+](#)[▶ TRAILER](#)

PG 2015 • 1h 35m • Coming of age, Family, Animation

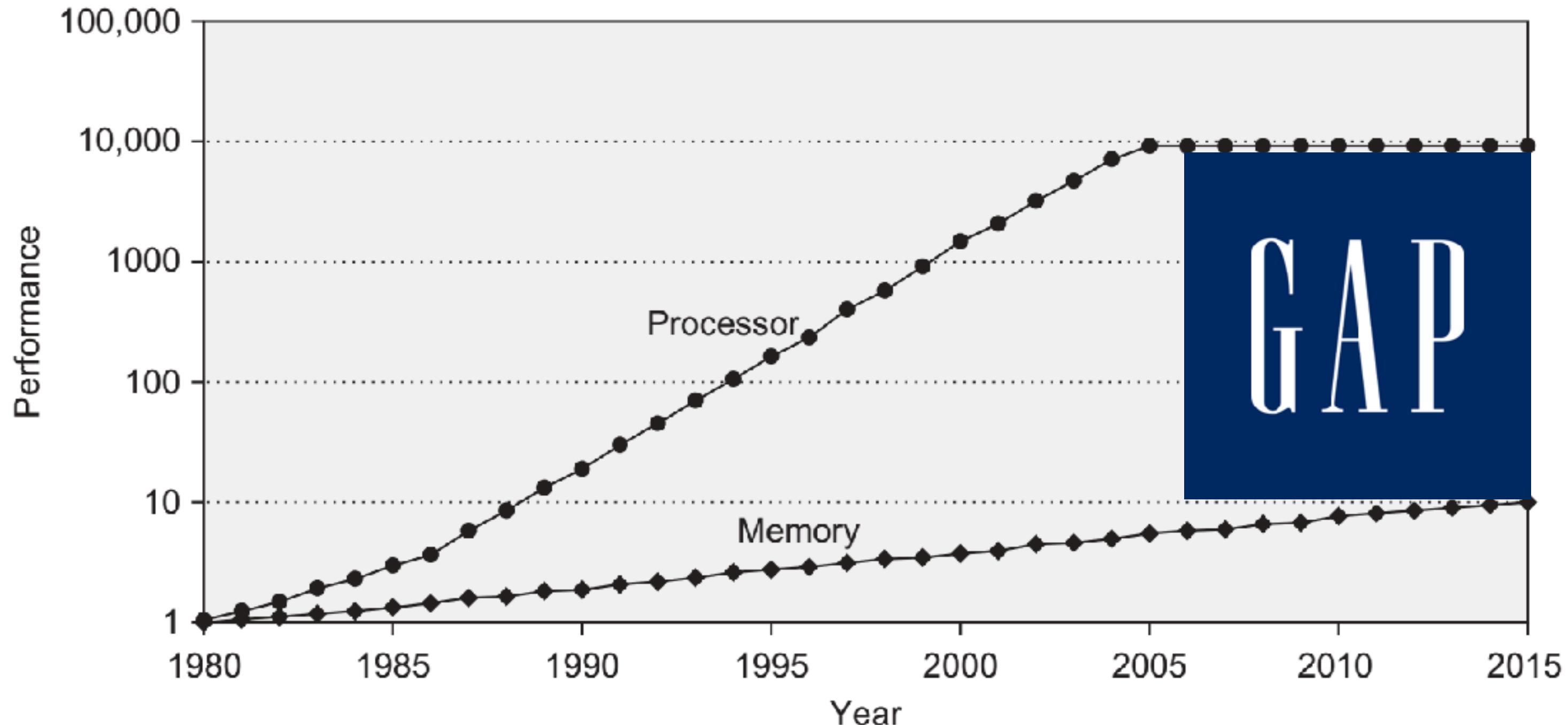
When 11-year-old Riley moves to a new city, her Emotions team up to help her through the transition. Joy, Fear, Anger, Disgust and Sadness work together, but when Joy and Sadness get lost, they must journey through unfamiliar places to get back home.



von Neumann Architecture



Recap: Performance gap between Processor/Memory



Outline

- The Basic Idea behind Memory Hierarchy
- How cache works

Modern DRAM performance

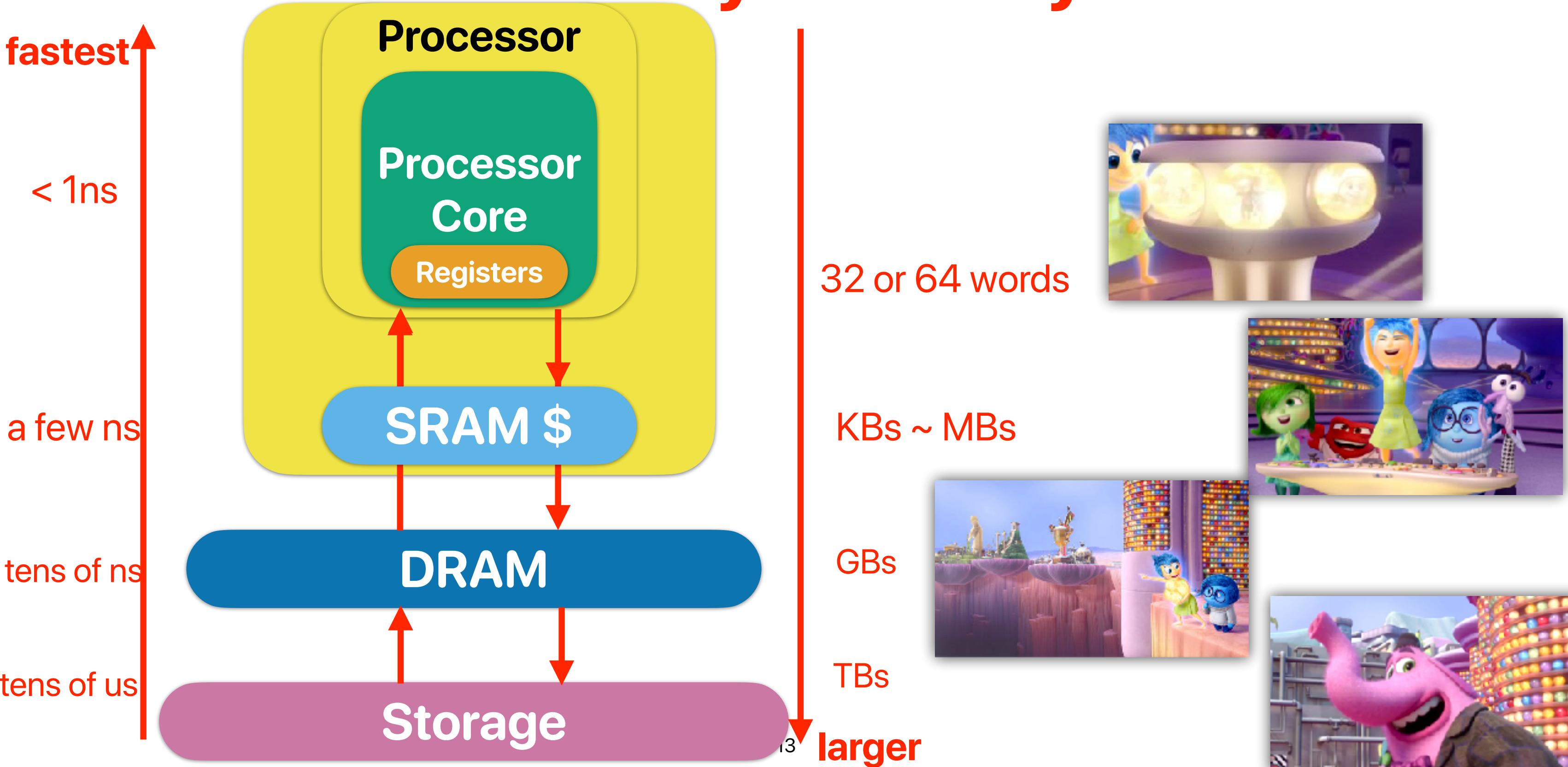
SDRAM					DDR				
Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year	Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year
100	0.80	3	24.00	1992	400	3.20	5	25.00	1998
133	1.07	3	22.50		667	5.33	5	15.00	
					800	6.40	6	15.00	
DDR 2					DDR 3				
400	3.20	5	25.00	2003	800	6.40	6	15.00	2007
667	5.33	5	15.00		1066	8.53	8	15.00	
800	6.40	6	15.00		1333	10.67	9	13.50	
					1600	12.80	11	13.75	
					1866	14.93	13	13.93	
					2133	17.07	14	13.13	
DDR 4					DDR 5				
1600	12.80	11	13.75	2014	3200	25.60	22	13.75	2020
1866	14.93	13	13.92		3600	28.80	26	14.44	
2133	17.07	15	14.06		4000	32.00	28	14.00	
2400	19.20	17	14.17		4400	35.20	32	14.55	
2666	21.33	19	14.25		4800	38.40	34	14.17	
2933	23.46	21	14.32		5200	41.60	38	14.62	
3200	25.20	22	13.75		5600	44.80	40	14.29	
					6000	48.00	42	14.00	
					6400	51.20	46	14.38	

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
Ext. Family	17	Ext. Model	8
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
Ext. Family	6	Ext. Model	9E
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3, TSX		

Clocks (Core #0)

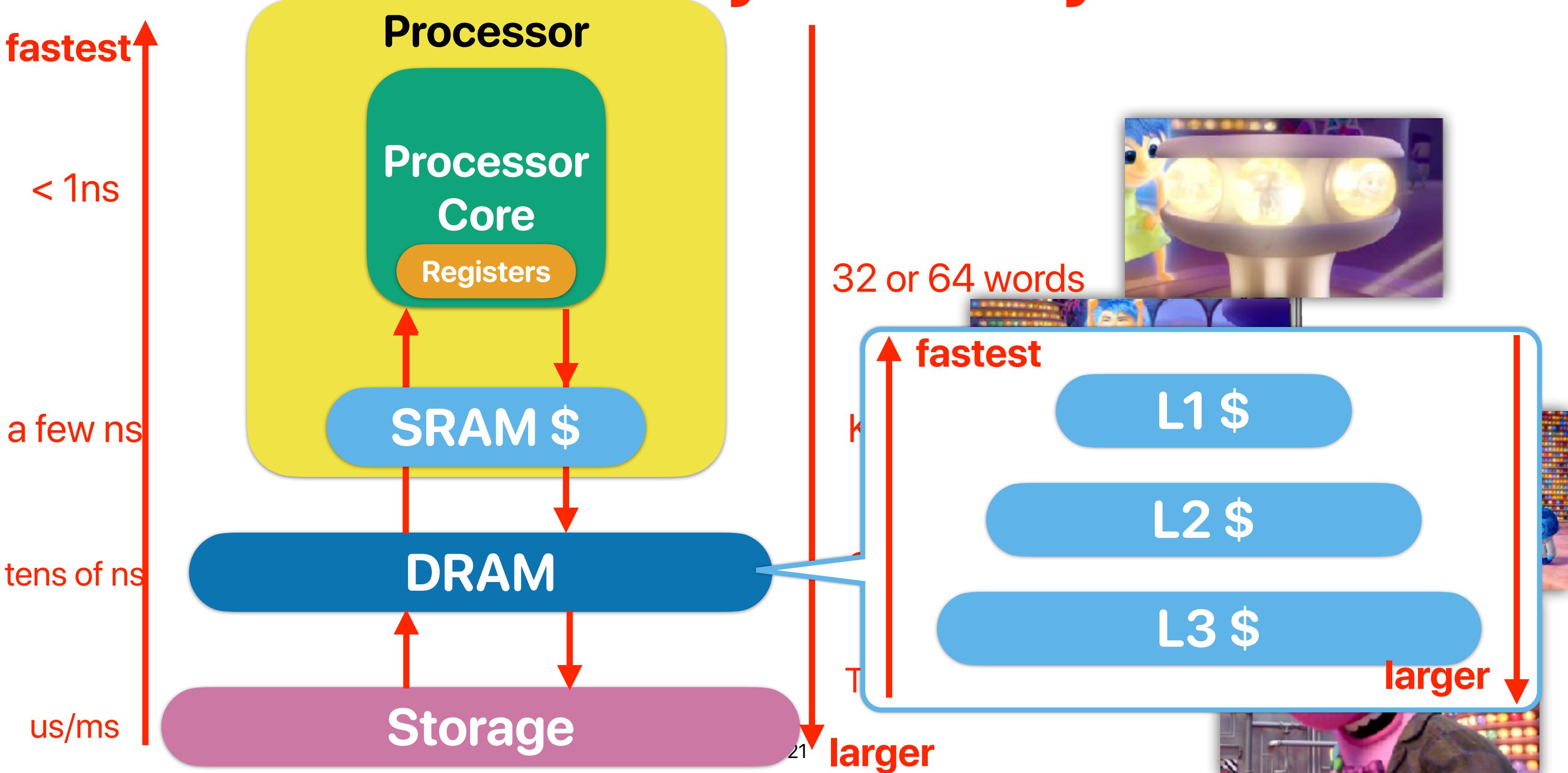
Core Speed	4798.85 MHz	
Multiplier	x 48.0 (8 - 49)	
Bus Speed	99.98 MHz	
Rated FSB		

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



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
Technology	12 nm
Core Voltage	1.35 V

Specification

Families	Family 17
Ext. Family	17
Instructions	MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86_64, AMD-V, AVX, AVX2, FMA3, SHA

Clocks (Core #0)

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

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

These are very small compared with your system main memory and program footprint. How does that work?

CPU | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name	Intel Core i7 9700K
Code Name	Tiger Lake
Max TDP	95 W
Technology	14 nm
Core Voltage	0.737 V

Specification

Families	Family 6
Ext. Family	6
Instructions	AVX, AVX2, SSSE3, SSE4.1, SSE4.2, FMA4, VT-x, AVX512, AVX512F, AVX512DQ, AVX512BW, TSX

Clocks (Core #0)

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

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

**Why adding small SRAMs would
work?**

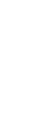
Code also has locality

```
for(uint32_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint32_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

repeat many times —
temporal locality!

```
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



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

- 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

Cache design principles — exploit localities

- The cache must be able to get chunks of near-by items every time to exploit spatial locality
- The cache must be able to keep a frequently used block for a while to exploit temporal locality

Architecting the Cache

Cache design principles — exploit localities

- The cache must be able to get chunks of near-by items every time to exploit spatial locality

We need to “cache consecutive data elements” every time

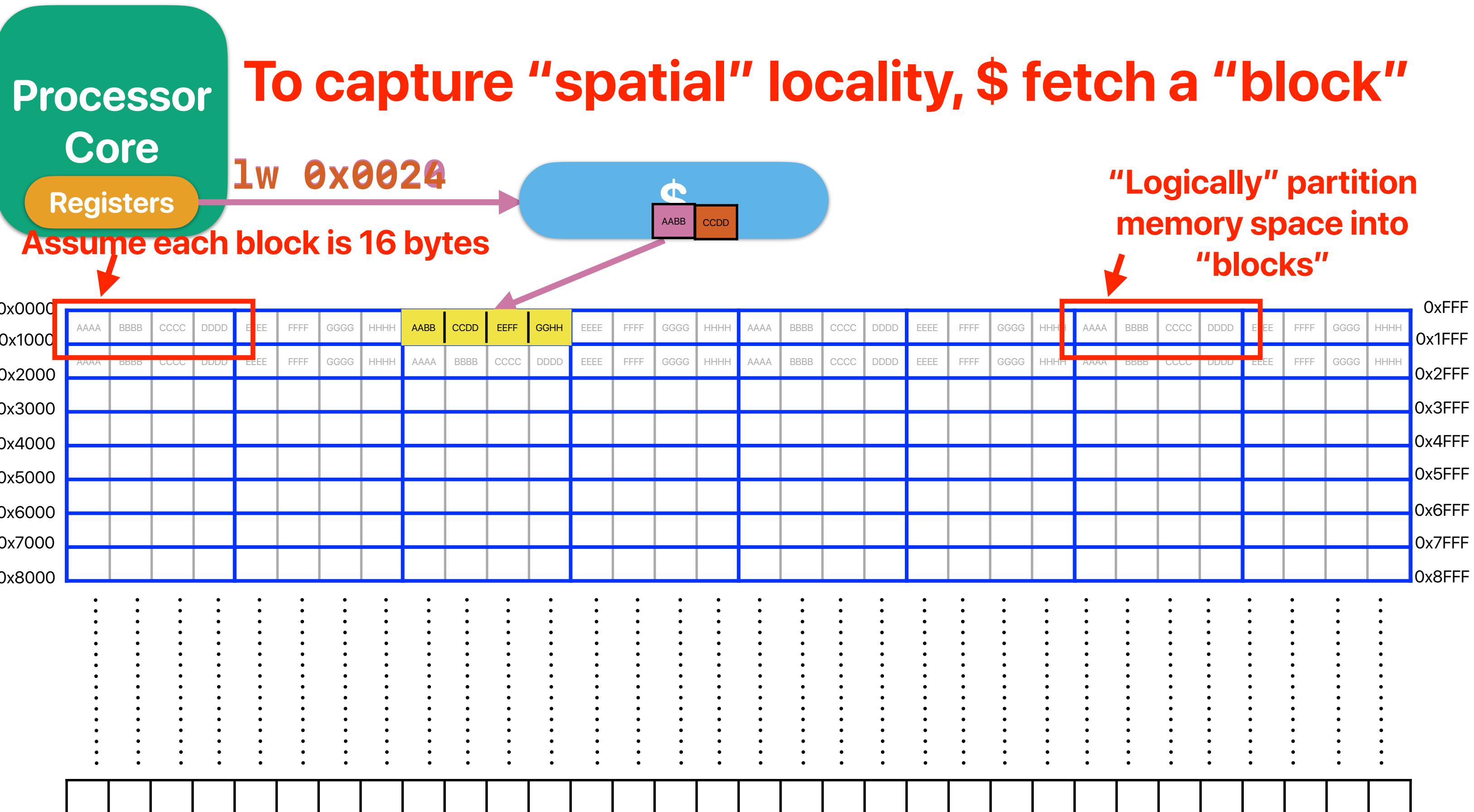
— the cache should store a “block” of data

- The cache must be able to keep a frequently used block for a while to exploit temporal locality

Processor Core Registers

Load/store only access a “word” each time

load 0x000A



Recap: Locality

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

```
for(uint32_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint32_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

Simply caching one block
isn't enough

Cache design principles — exploit localities

- The cache must be able to get chunks of near-by items every time to exploit spatial locality

We need to “cache consecutive data elements” every time

— the cache should store a “block” of data

- The cache must be able to keep a frequently used block for a while to exploit temporal locality

We need to store multiple blocks

— the cache must be able to distinguish blocks

Processor Core

Registers

the byte addresses of each byte in the block

What's a block?

0x0000, 0x0001, 0x0002, ..., 0x000F
0x0010, 0x0011, 0x0012, ..., 0x001F

the address in each block starts with the same “prefix”

the offset of the byte within a block

0123456789ABCDEF

This is CSE142:

Advanced Compute

the data in memory

How to tell who is there?

tag

0x000

This is CSE142:

Advanced Compute

r Architecture!

This is CSE142:

How to tell who is there?

the common address
prefix in each block

tag array	0123456789ABCDEF
0x000	This is CSE142:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CSE142:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CSE142:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CSE142:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CSE142:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CSE142:

Processor Core

Registers

How to tell whether

block offset
tag

1w 0x0008

1w 0x4048

0x404 not found,
go to lower-level memory

The complexity of search the matching tag—

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

Can we search things faster?

—hash table! $O(1)$

		Valid Bit	Dirty Bit	Tag	Data
1	1	0x000		This is CSE142:	0123456789ABCDEF
1	1	0x001		Advanced Compute	
1	0	0xF07		r Architecture!	
0	1	0x100		This is CSE142:	
1	1	0x310		Advanced Compute	
1	1	0x450		r Architecture!	
0	1	0x006		This is CSE142:	
0	1	0x537		Advanced Compute	
1	1	0x266		r Architecture!	
1	1	0x307		This is CSE142:	
0	1	0x265		Advanced Compute	
0	1	0x80A		r Architecture!	
1	1	0x620		This is CSE142:	
1	1	0x630		Advanced Compute	
1	0	0x705		r Architecture!	
0	1	0x216		This is CSE142:	

Tell if the block here can be used

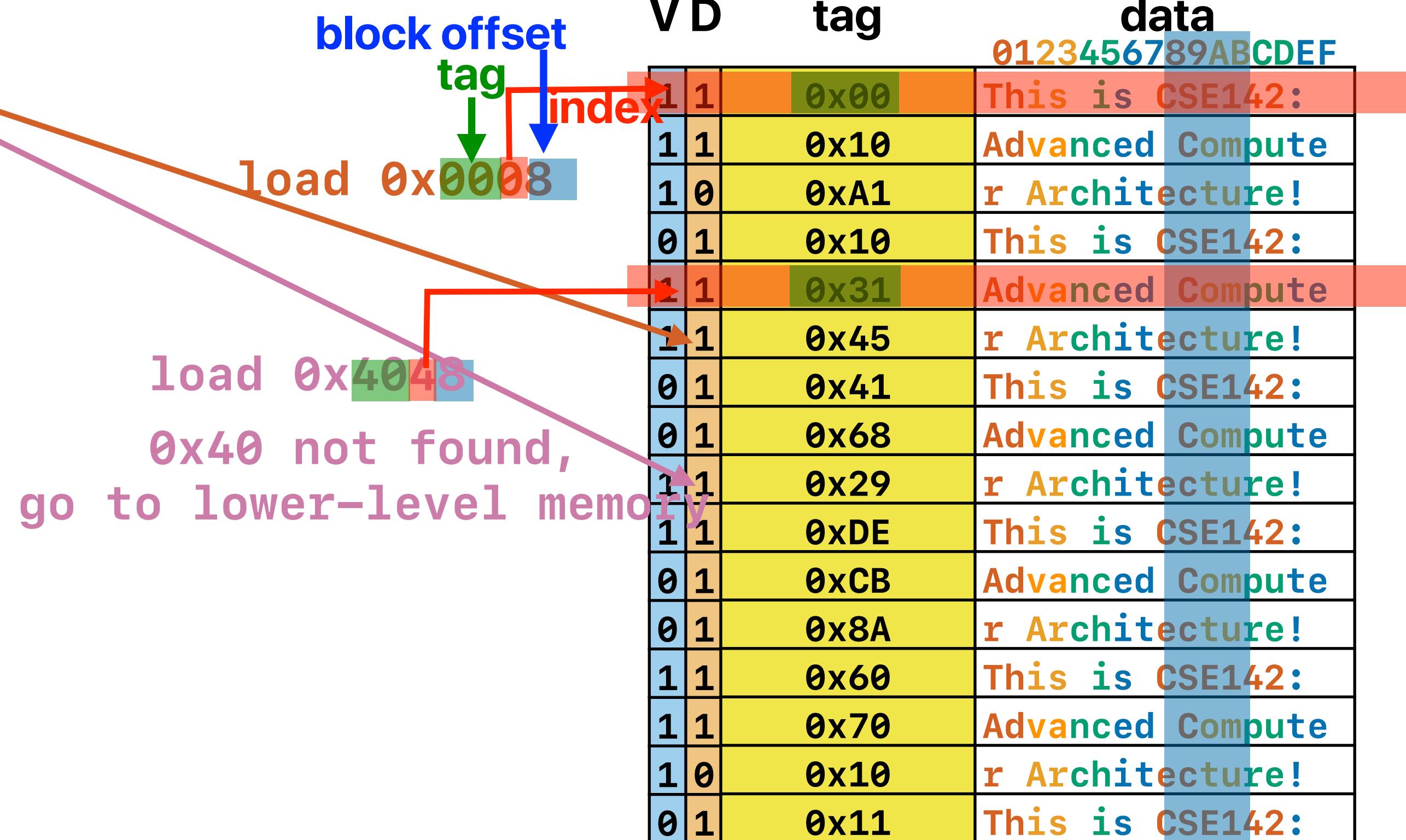
Tell if the block here is modified

Processor

Core

Registers

Hash-like structure — direct-mapped cache



Way-associative cache

memory address:

0x0 8 2 4
 set block
 index offset

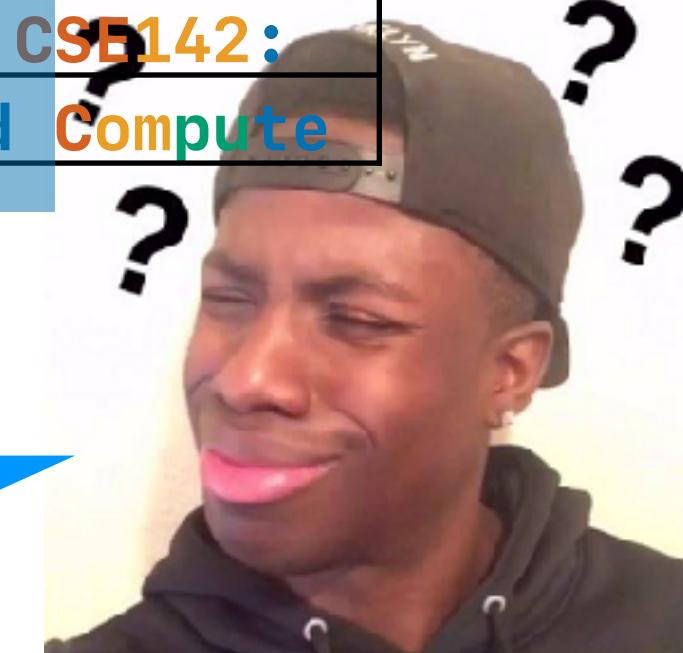
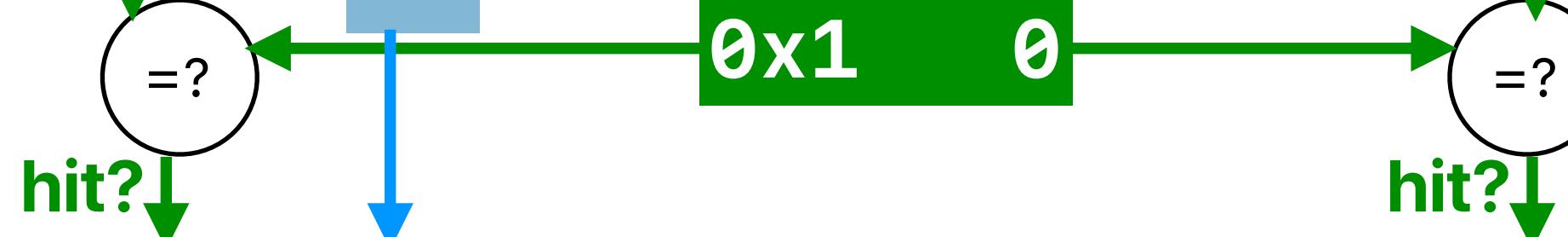
memory address:

0b0000100000100100

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

Set

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



Ways?

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
Ext. Family	17	Ext. Model	8
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
Ext. Family	6	Ext. Model	9E
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

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



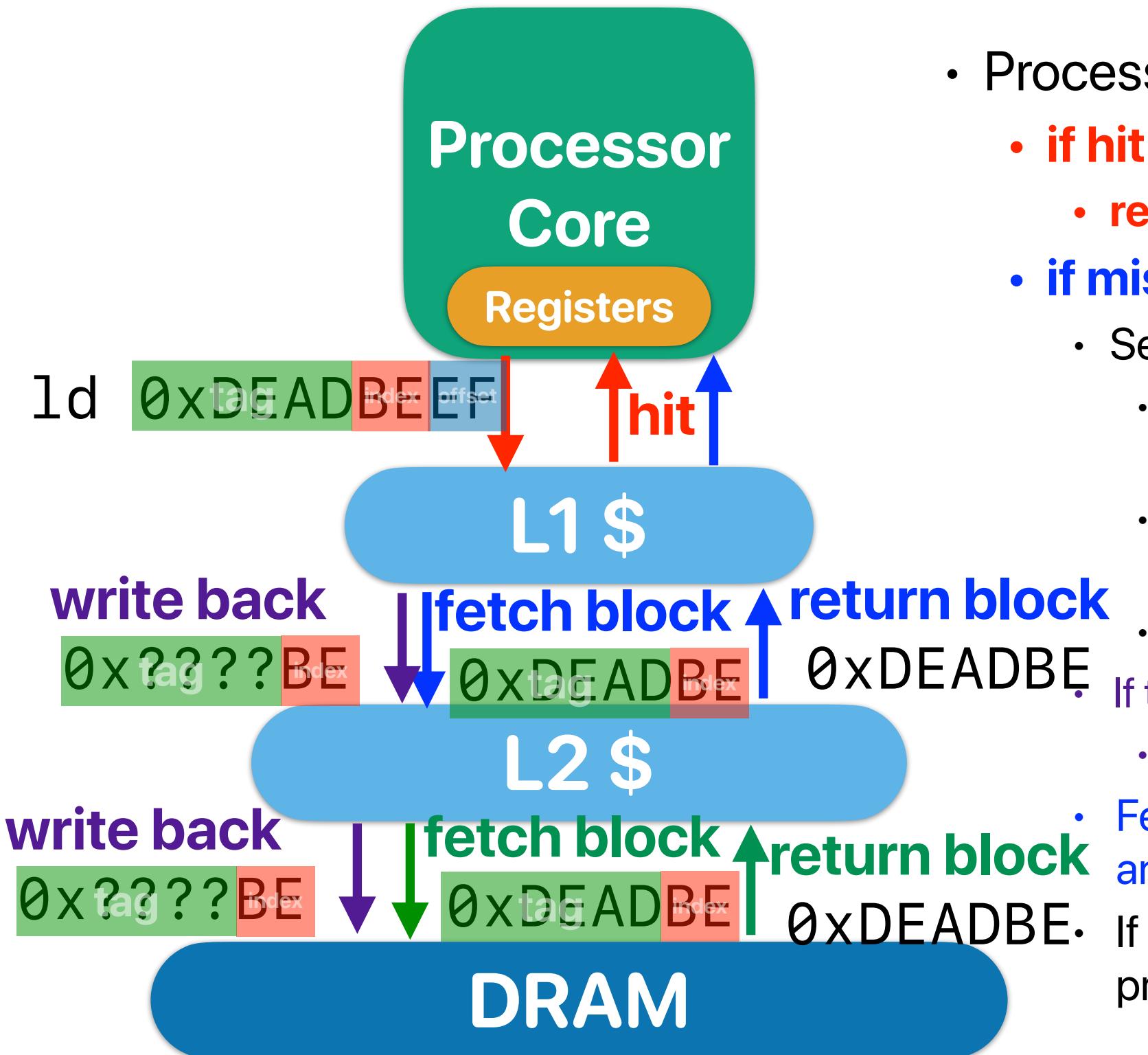
Corollary of C = ABS



- number of bits in **block offset** — $\lg(B)$
- number of bits in **set index**: $\lg(S)$
- tag bits: $\text{address_length} - \lg(S) - \lg(B)$
 - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- $(\text{address} / \text{block_size}) \% S = \text{set index}$

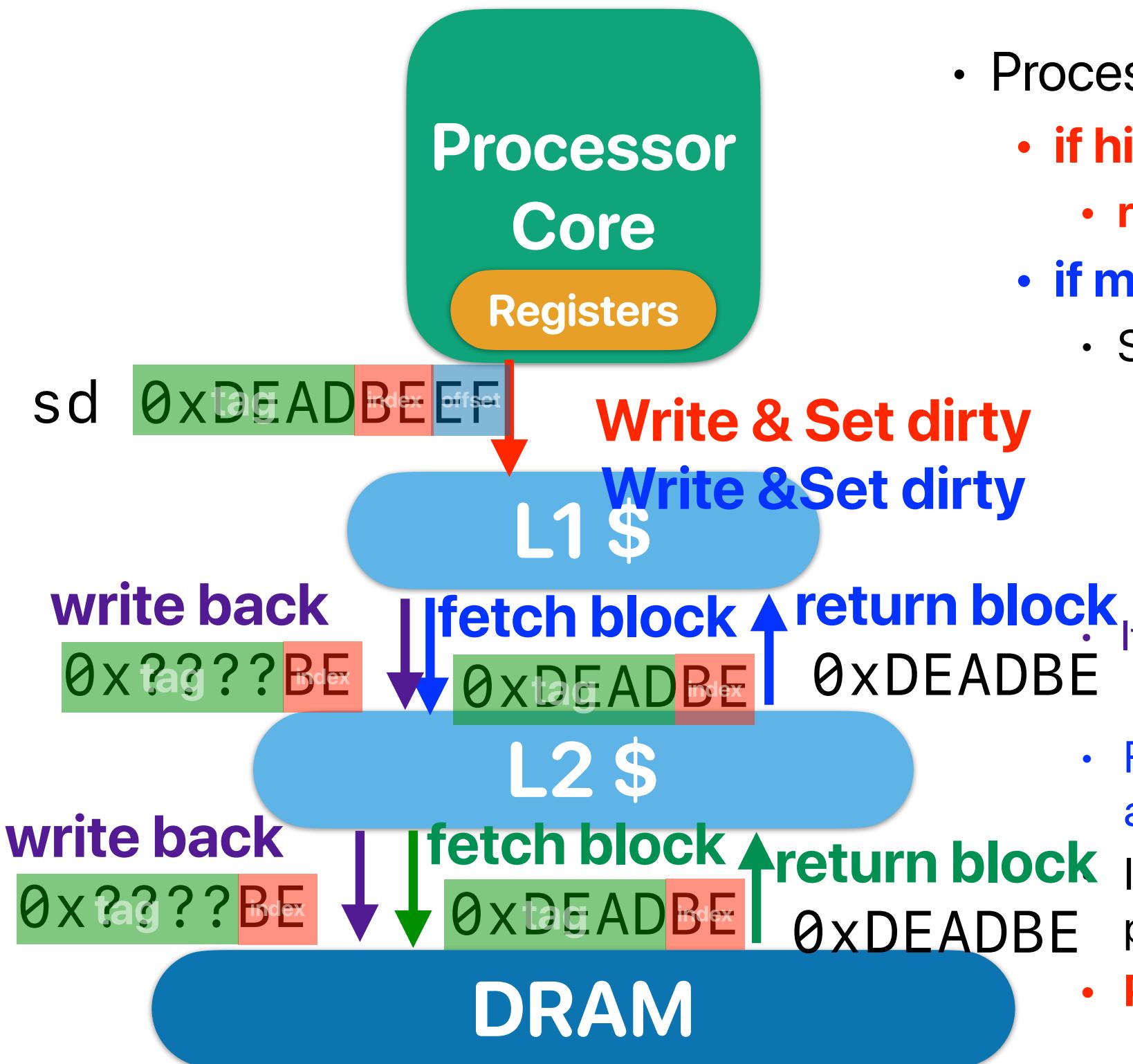
Put everything all together: How cache interacts with CPU

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:
 - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
 - $C = A \text{ } B \text{ } S$
 - $S = 256 / (16 * 1) = 16$
 - $\lg(16) = 4$: 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!
1	1	0	0b10	This is CSE142:
2	0	0		
3	0	0		
4	0	0		
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

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

Simulate a 2-way cache

- Consider a 2-way cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
 - 0b1000000000, 0b1000001000, 0b1000010000,
0b1000010100, 0b1100010000
 - $C = A \text{ } B \text{ } S$
 - $S = 256 / (16 * 2) = 8$
 - $8 = 2^3$: 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	tag	index	
0	1	0	0b10	r Architecture!	0	0			0b10	0000	miss
1	1	0	0b10	This is CSE142:	1	0	0b11	Advanced Compute	0b10	0000	hit!
2	0	0			0	0			0b10	0001	miss
3	0	0			0	0			0b10	0001	hit!
4	0	0			0	0			0b10	0000	miss
5	0	0			0	0			0b11	0001	miss
6	0	0			0	0			0b10	0000	hit!
7	0	0			0	0			0b10	0000	hit!

Taxonomy/reasons 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

Basic Hardware Optimization in Improving 3Cs

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.

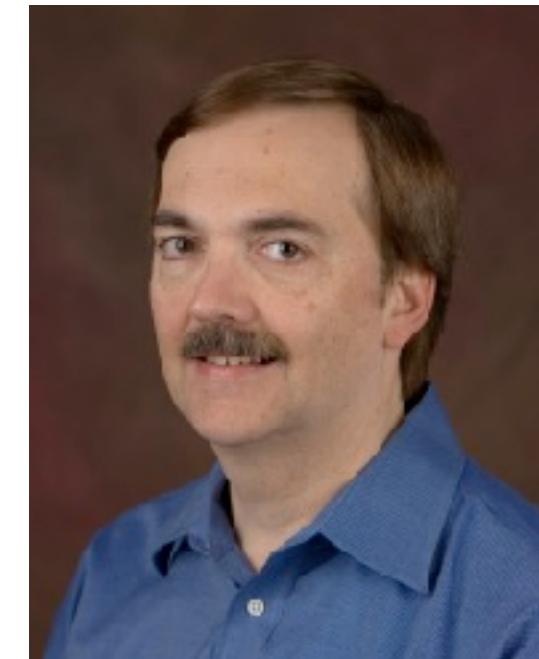
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

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

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

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

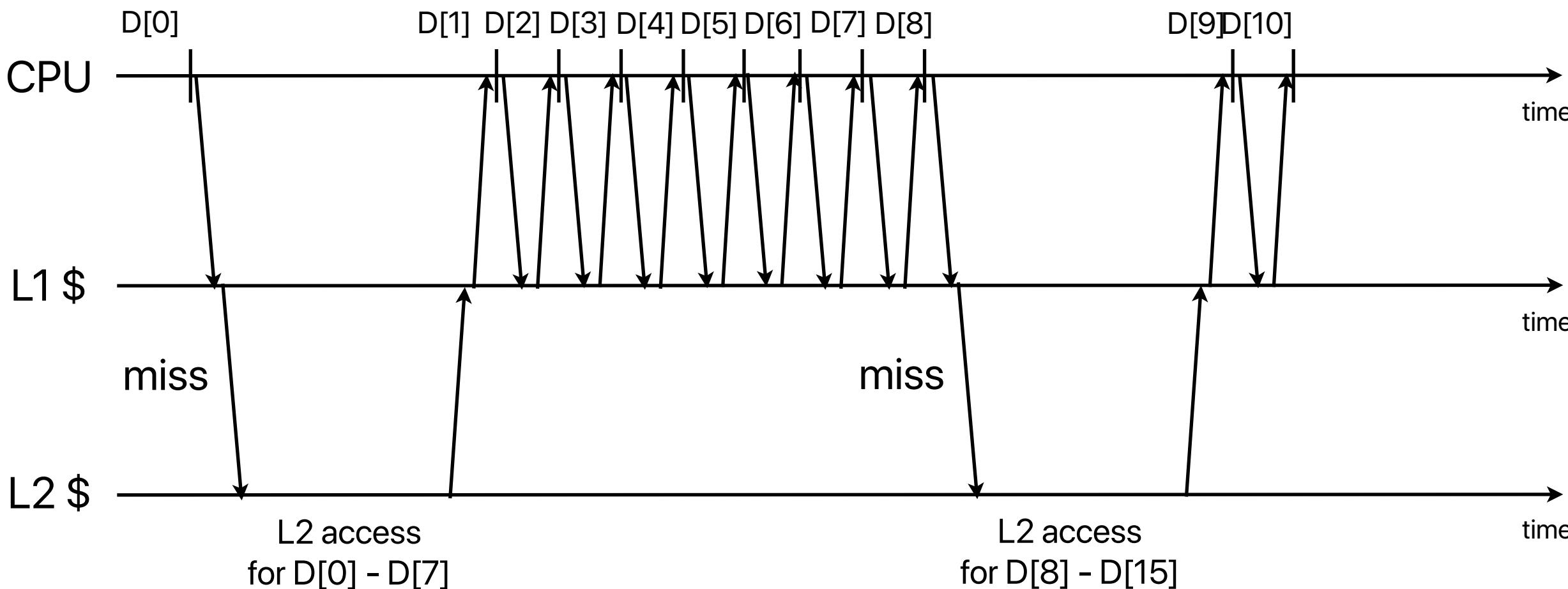
Norman P. Jouppi



Prefetching

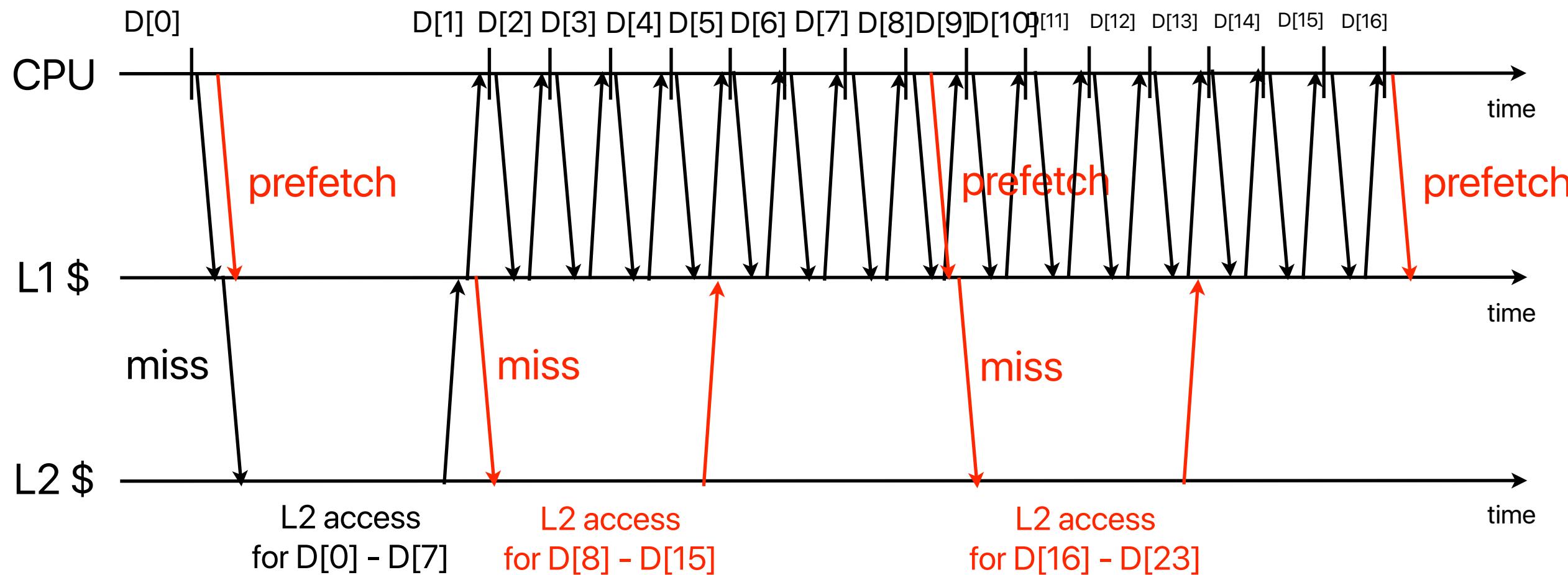
Characteristic of memory accesses

```
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  
}
```



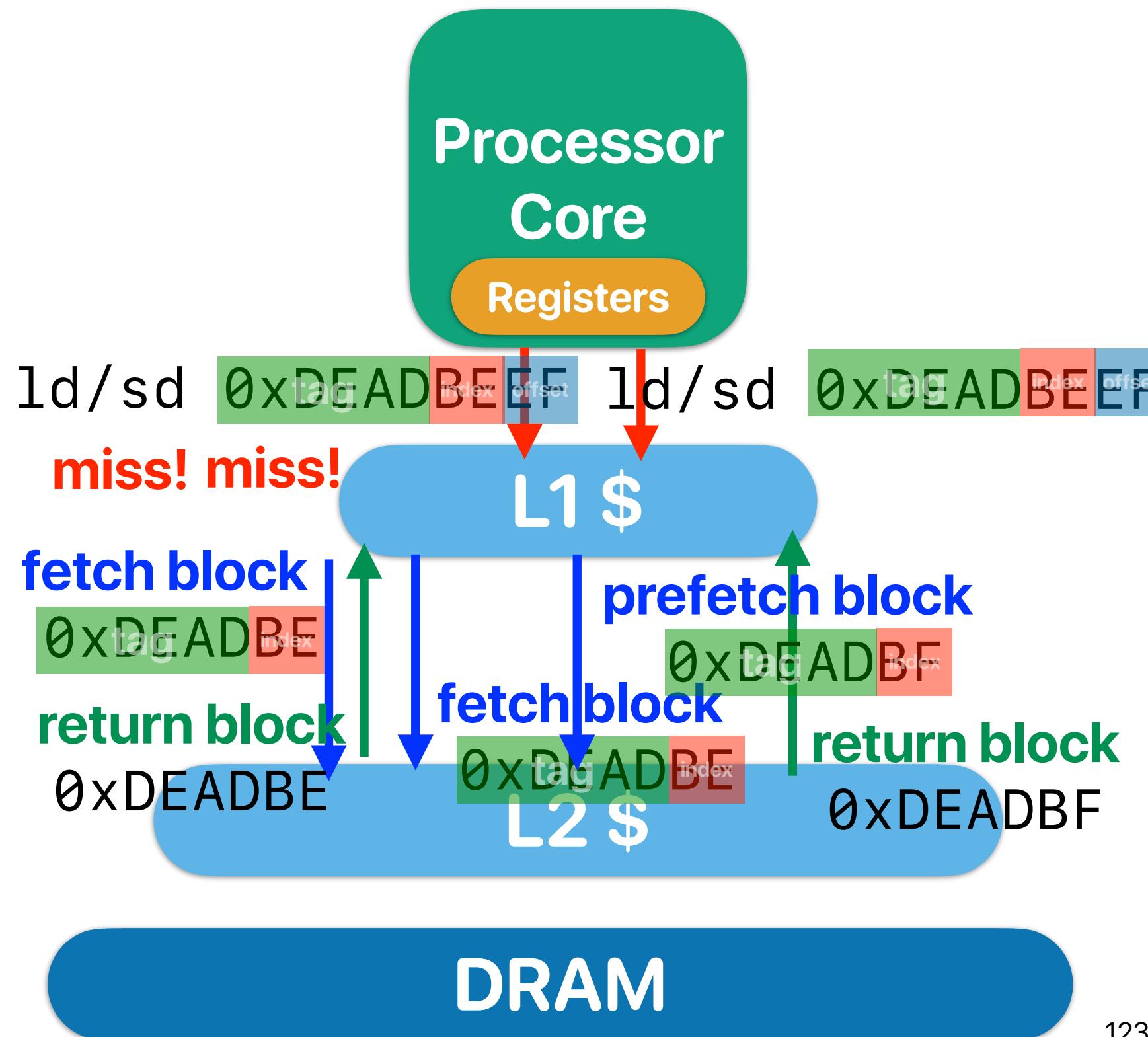
Prefetching

- 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

What's after prefetching?



NVIDIA Tegra X1 with prefetch

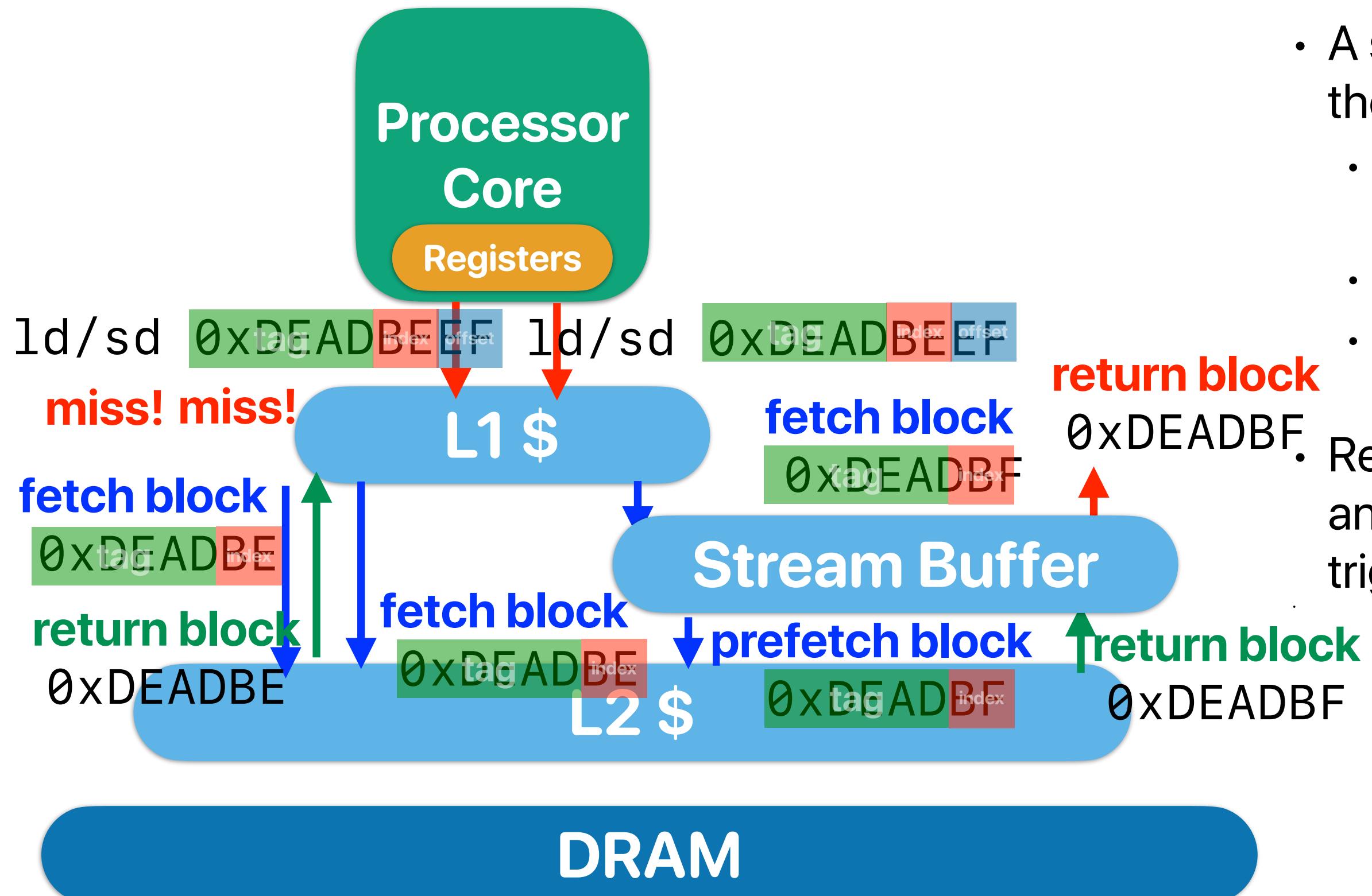
- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = \text{ABS}$
 $32\text{KB} = 4 * 64 * S$
 $S = 128$
 $\text{offset} = \lg(64) = 6 \text{ bits}$
 $\text{index} = \lg(128) = 7 \text{ bits}$
 $\text{tag} = \text{the rest bits}$

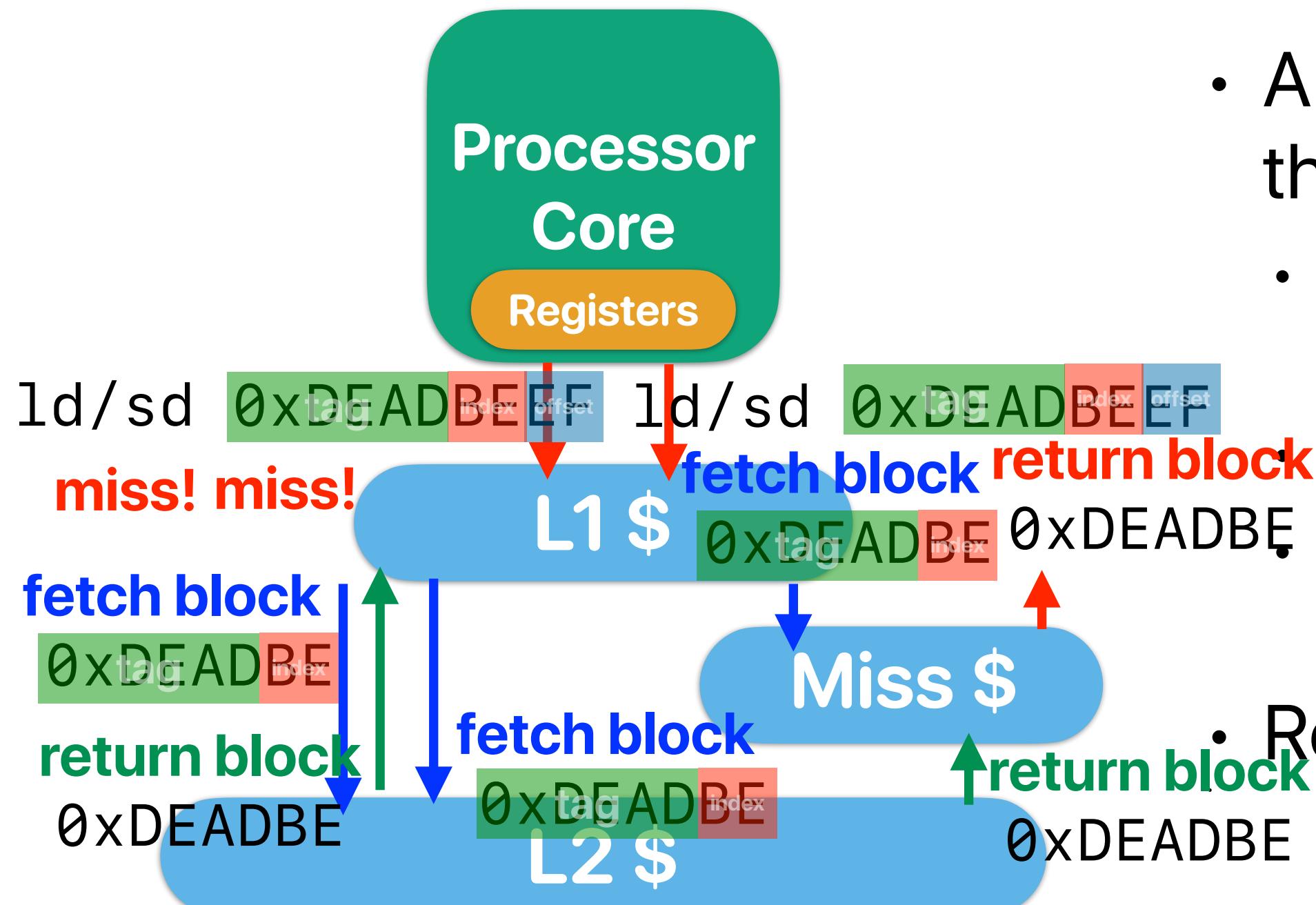
	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?	Prefetch
a[0]	0x10000	0b000100000000000000000000000000	0x8	0x0	Miss		a[8-15]
b[0]	0x20000	0b001000000000000000000000000000	0x10	0x0	Miss		b[8-15]
c[0]	0x30000	0b001100000000000000000000000000	0x18	0x0	Miss		c[8-15]
d[0]	0x40000	0b010000000000000000000000000000	0x20	0x0	Miss		d[8-15]
e[0]	0x50000	0b010100000000000000000000000000	0x28	0x0	Miss	a[0-7]	e[8-15]
a[1]	0x10008	0b0001000000000000000000001000	0x8	0x0	Miss	b[0-7]	e[8-15] will kick out a[8-15]
b[1]	0x20008	0b0010000000000000000000001000	0x10	0x0	Miss	c[0-7]	
c[1]	0x30008	0b0011000000000000000000001000	0x18	0x0	Miss	d[0-7]	
d[1]	0x40008	0b0100000000000000000000001000	0x20	0x0	Miss	e[0-7]	
e[1]	0x50008	0b0101000000000000000000001000	0x28	0x0	Miss	a[0-7]	

Stream buffer



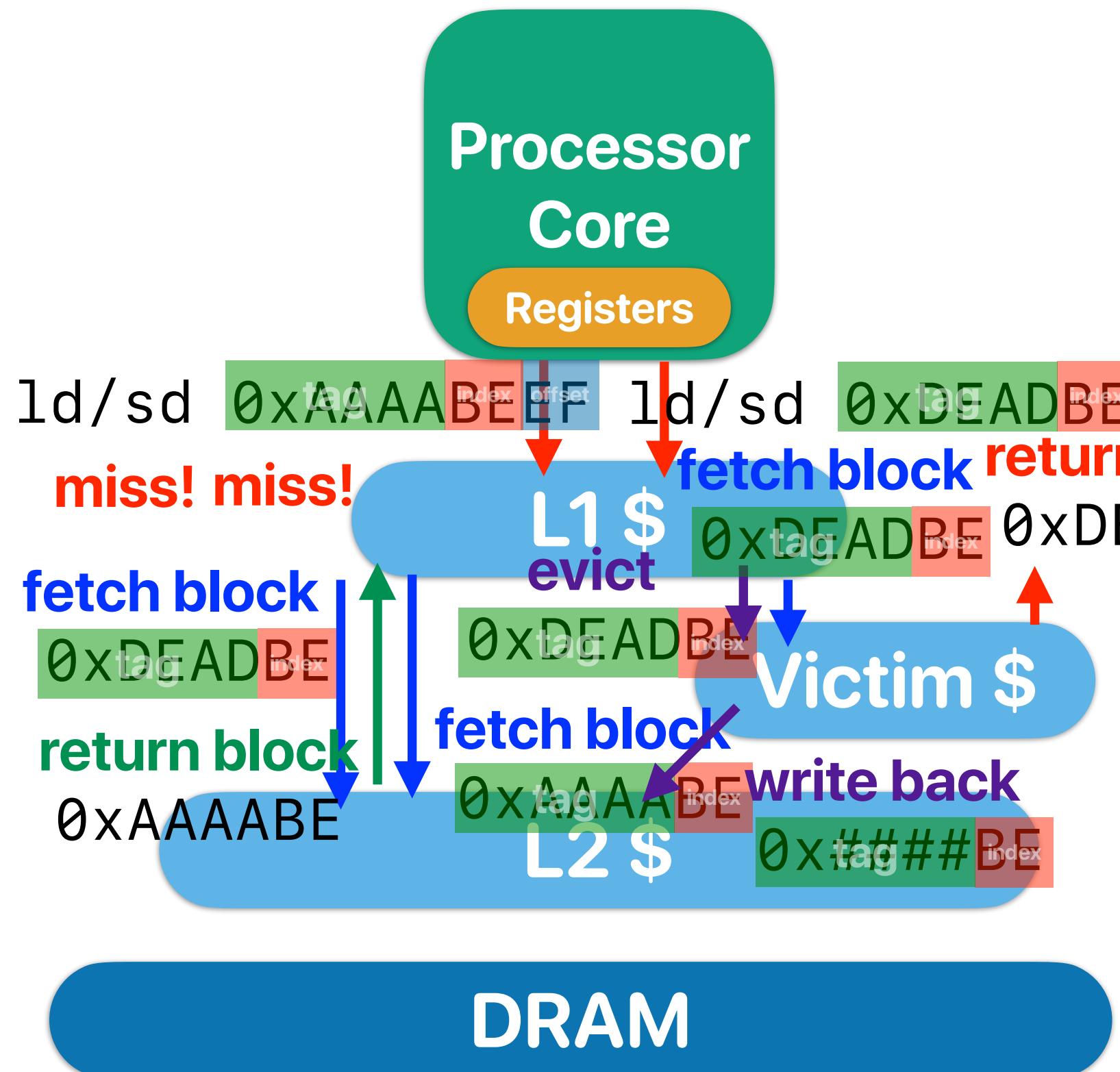
- A small cache that captures the prefetched blocks
 - Can be built as fully associative since it's small
 - Consult when there is a miss
 - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

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

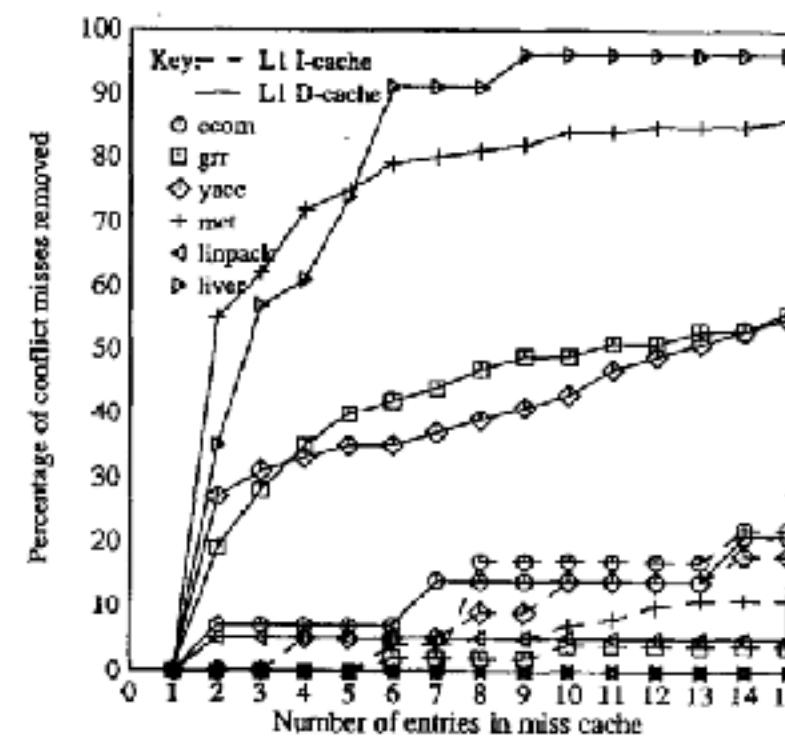


Figure 3-3: Conflict misses removed by miss caching

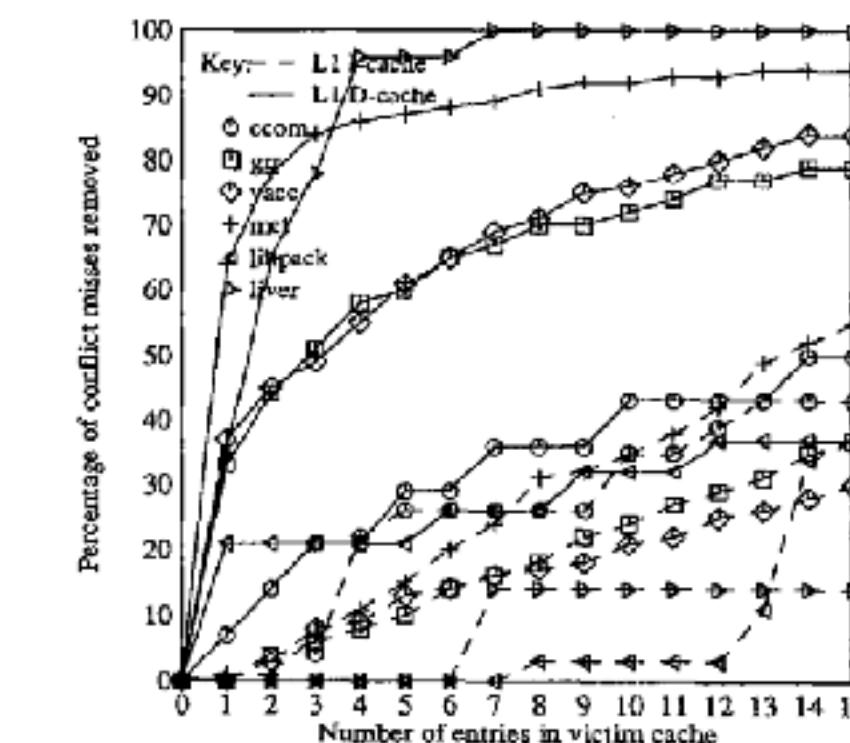


Figure 3-5: Conflict misses removed by victim caching

Which of the following schemes can help Tegra?

- 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 NVIDIA’s Tegra 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

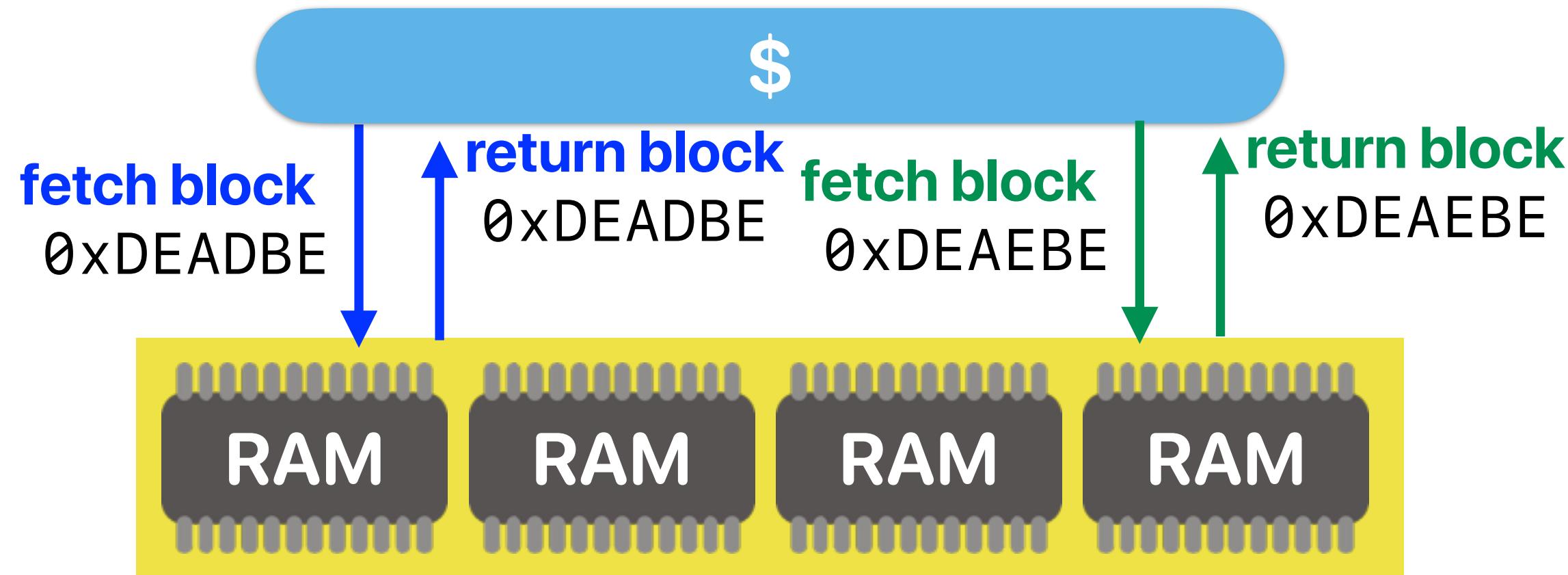
C. 2

D. 3

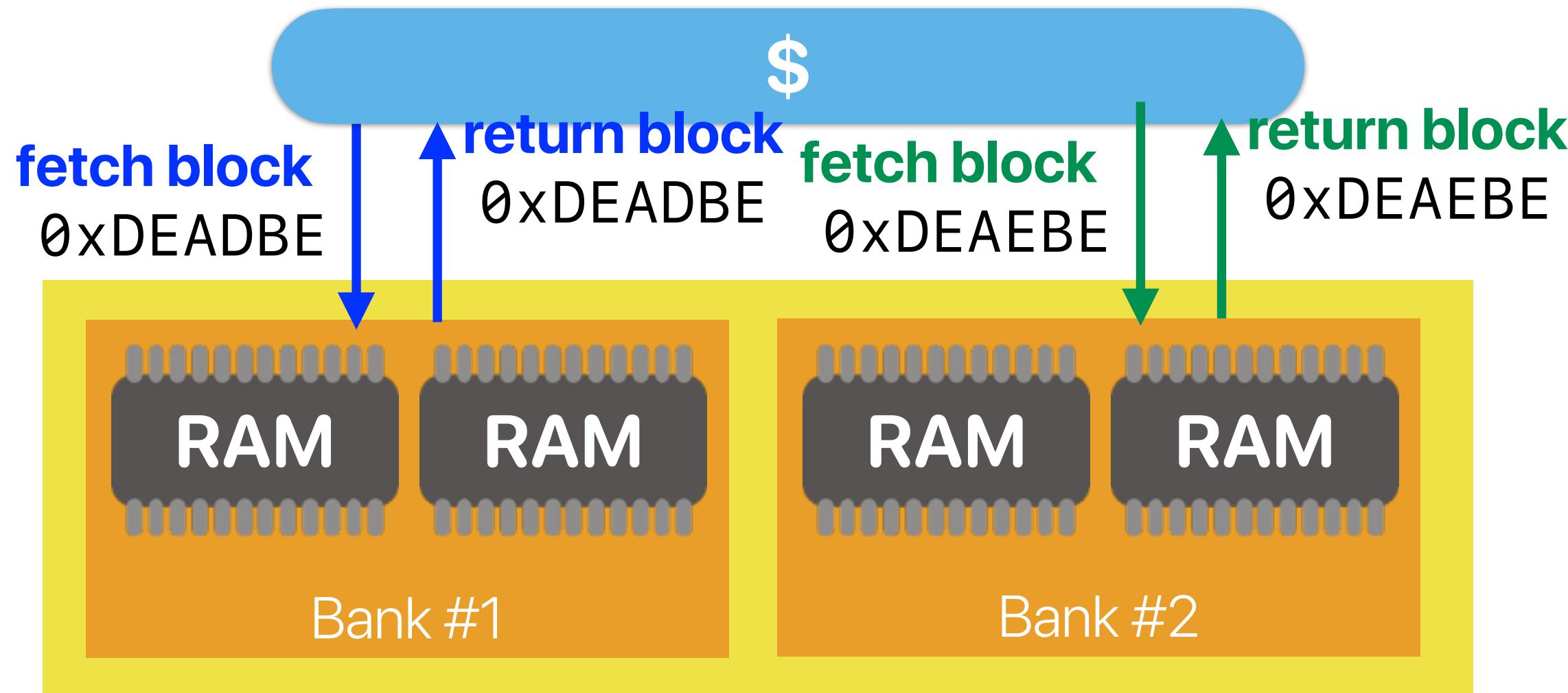
E. 4

Advanced Hardware Techniques in Improving Memory Performance

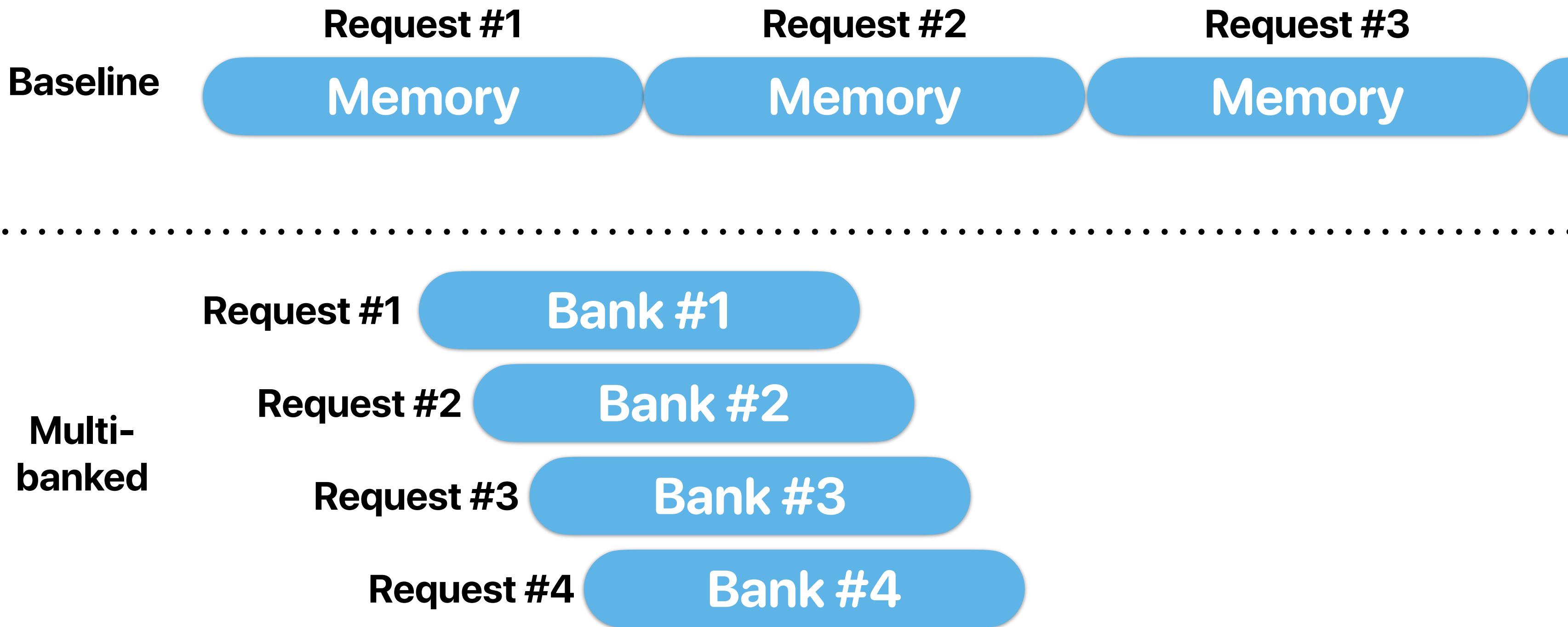
Blocking cache



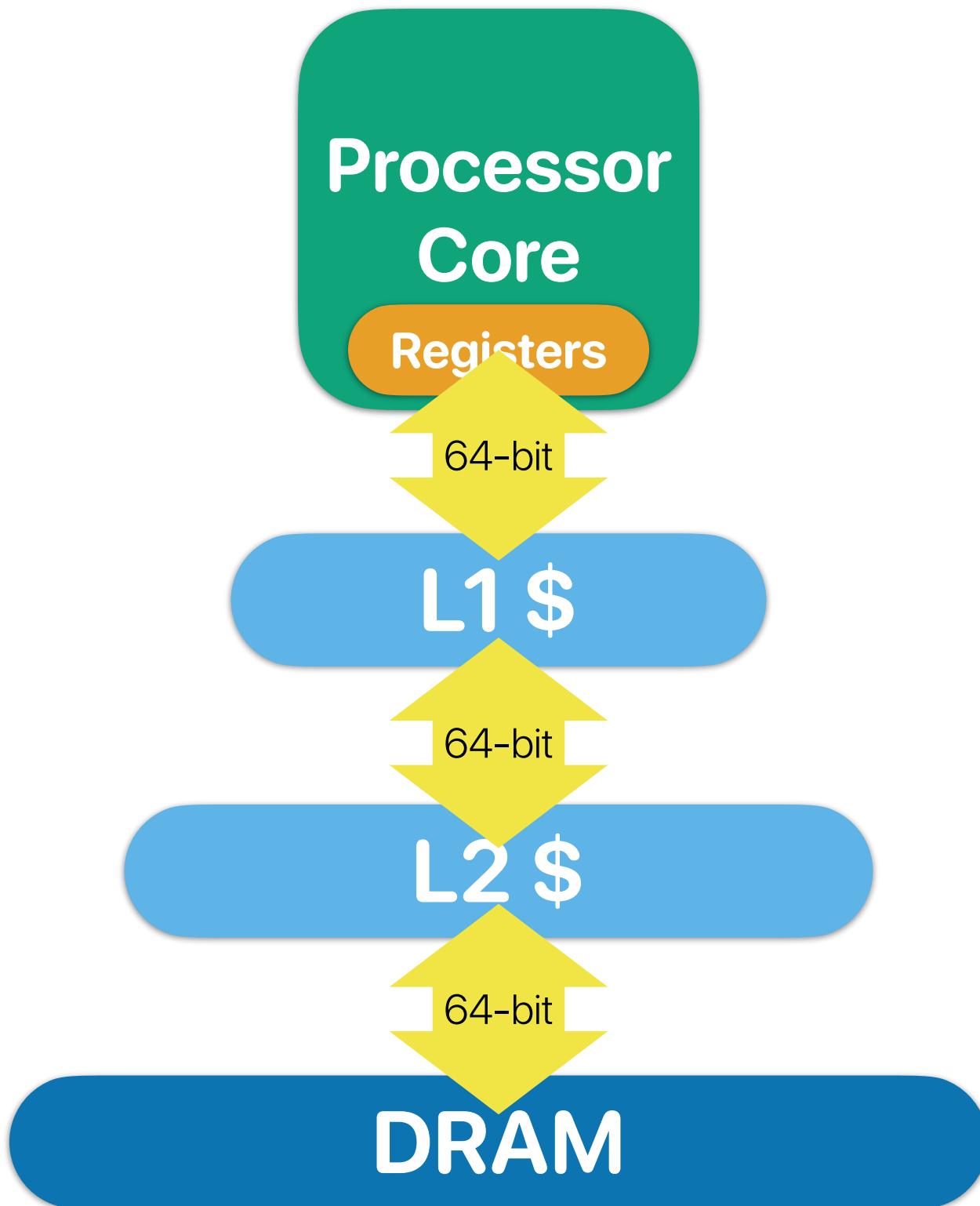
Multibanks & non-blocking caches



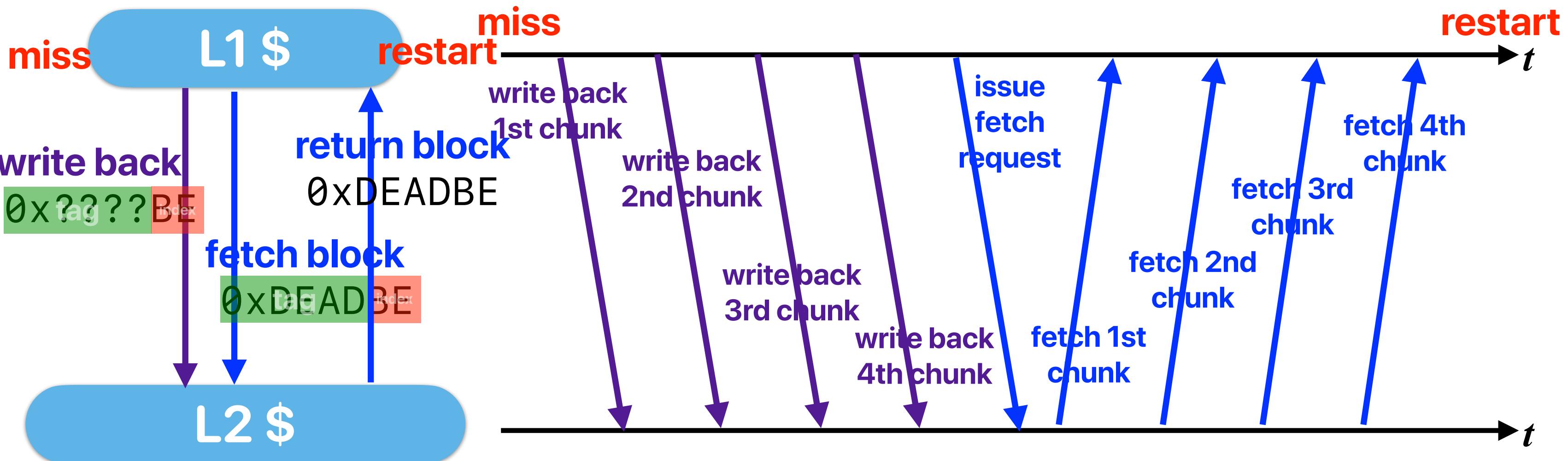
Pipelined access and multi-banked caches



The bandwidth between units is limited

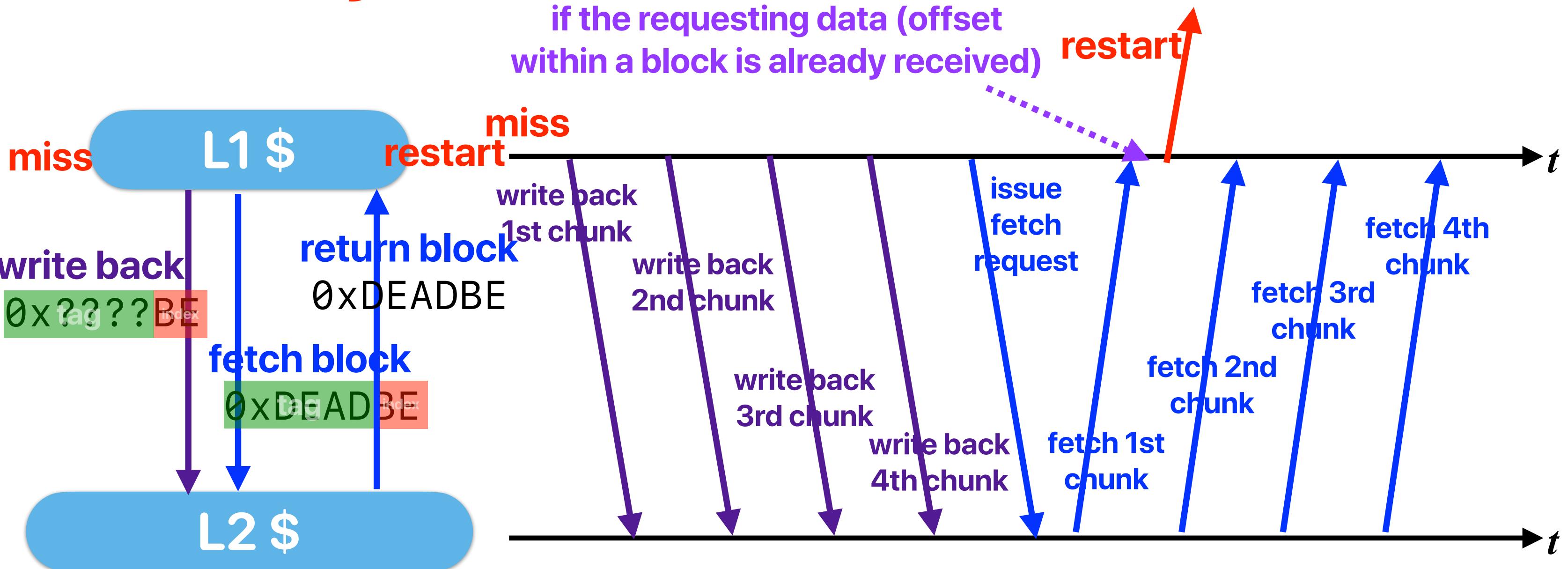


When we handle a miss



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

Early Restart and Critical Word First

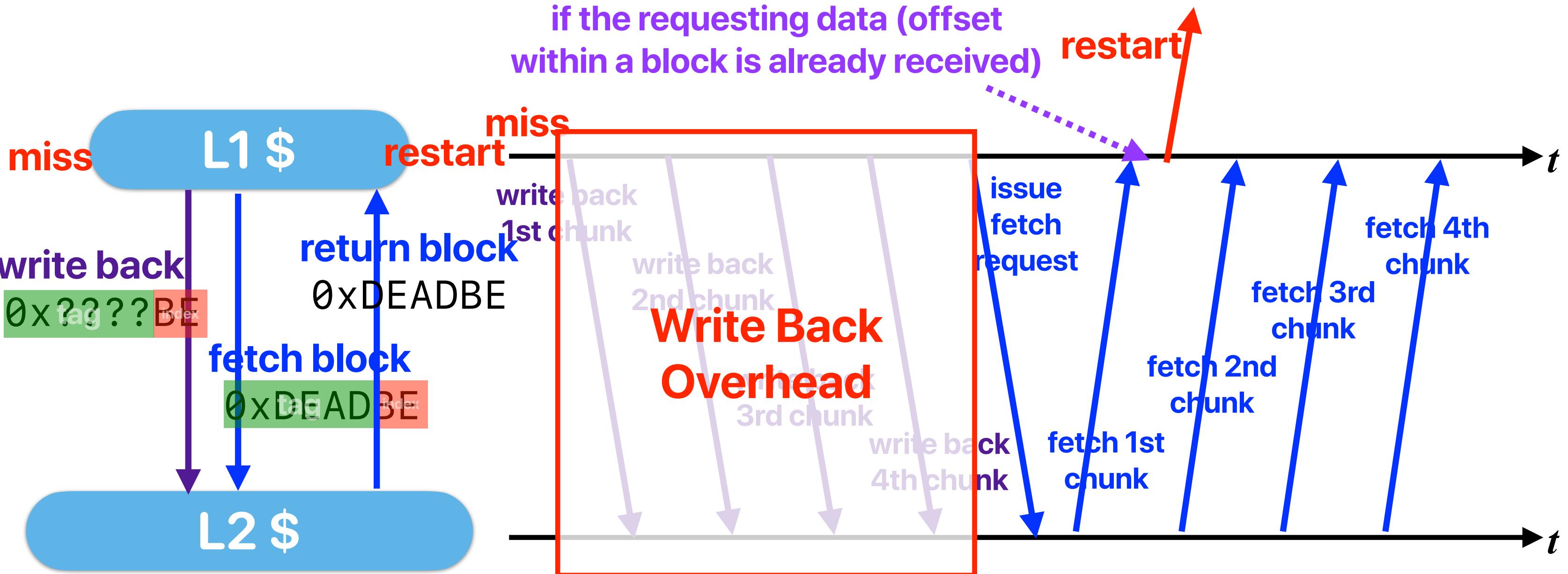


assume the bus between L1/L2 only allows a quarter of the cache block go through it

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).

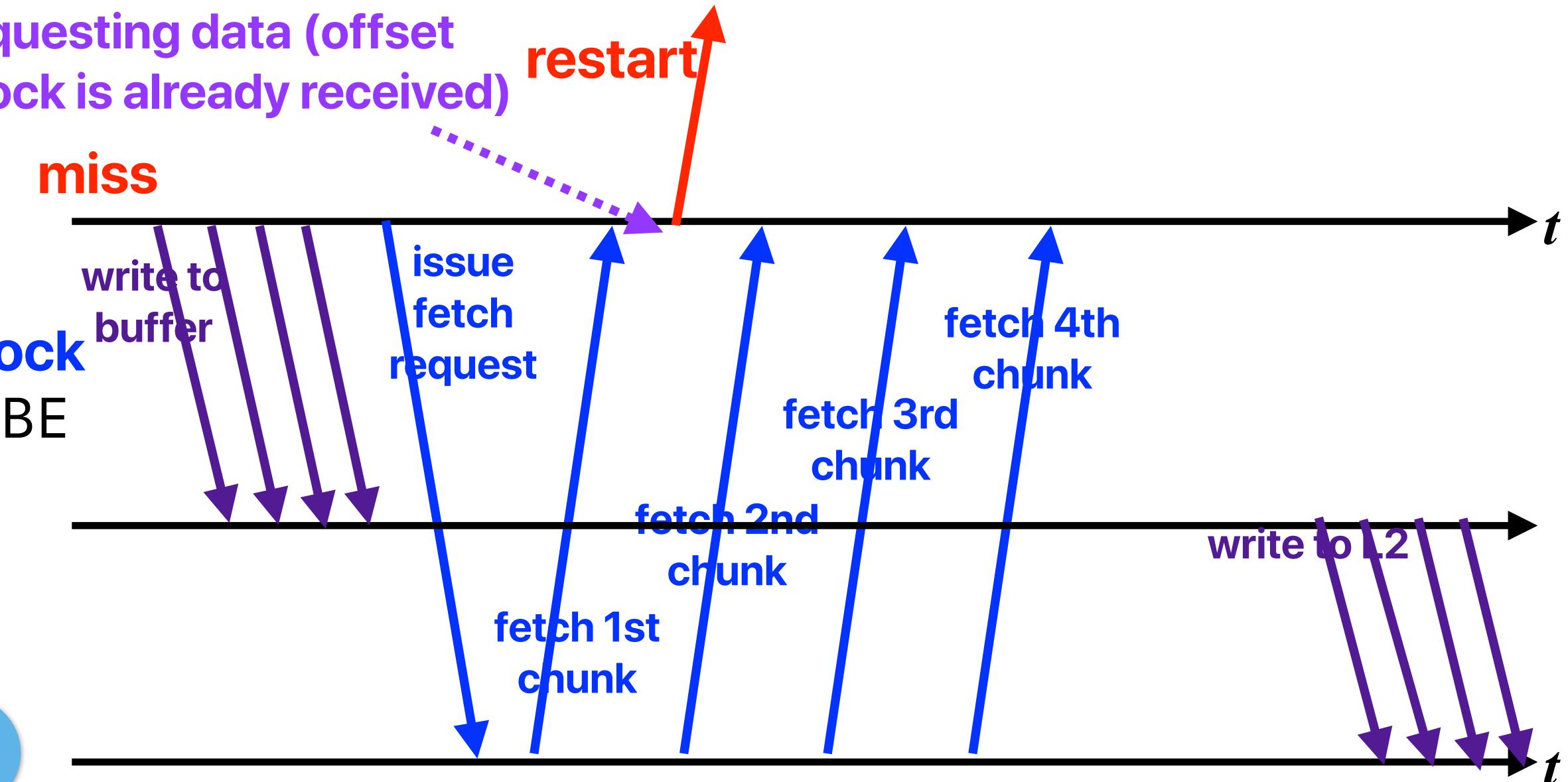
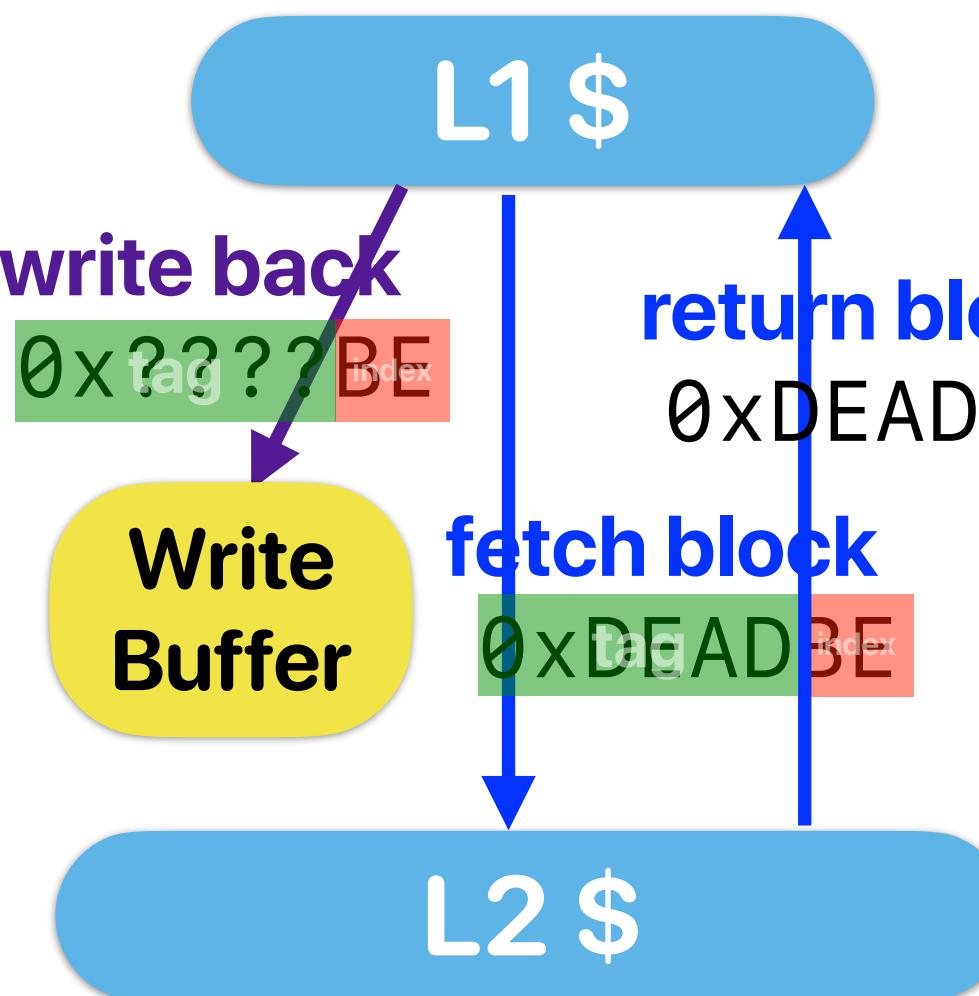
Can we avoid the overhead of writes?



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

Write buffer!

if the requesting data (offset within a block is already received)



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.

Summary of Optimizations

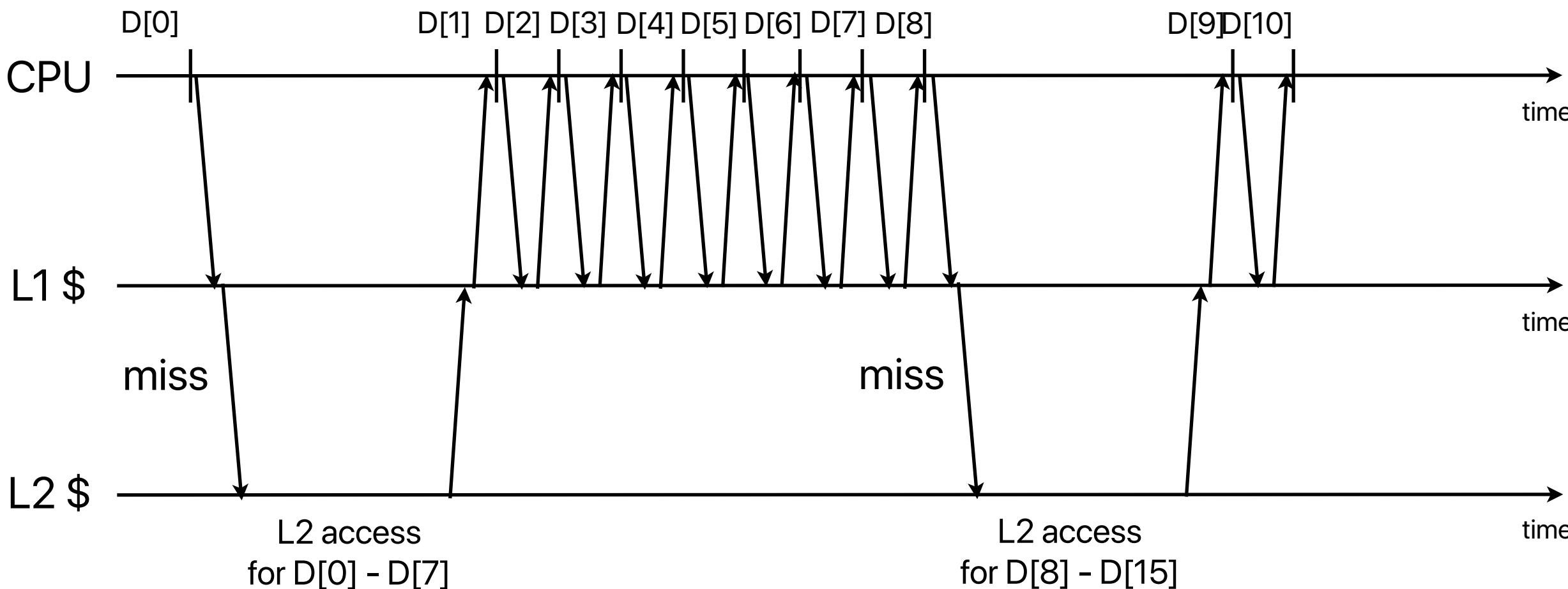
- Hardware
 - Prefetch — compulsory miss
 - Write buffer — miss penalty
 - Bank/pipeline — miss penalty
 - Critical word first and early restart — miss panelty

How can programmer improve memory performance?

Prefetching

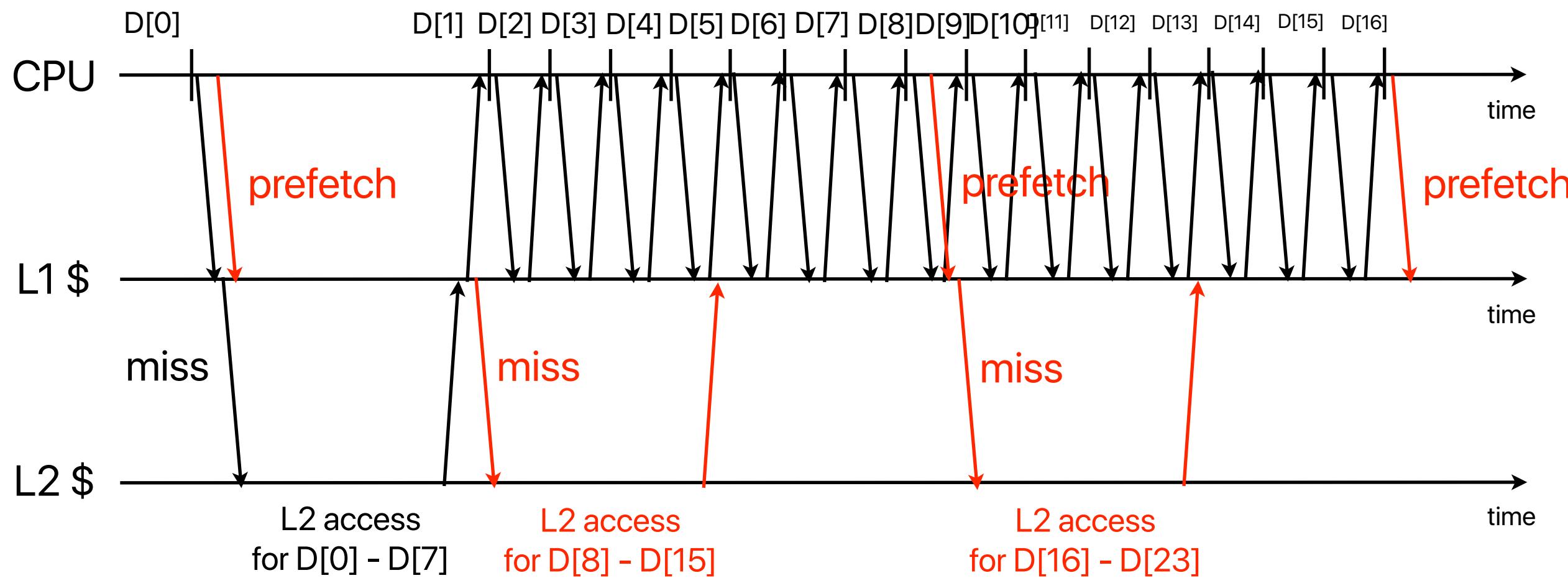
Characteristic of memory accesses

```
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 some register
 - Using prefetch instructions

Data structures

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

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, Firsntname 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];  
    }  
}
```

$O(n^2)$

Same

Same

Better

Complexity

Instruction Count?

Clock Rate

CPI

$O(n^2)$

Same

Same

Worse

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?

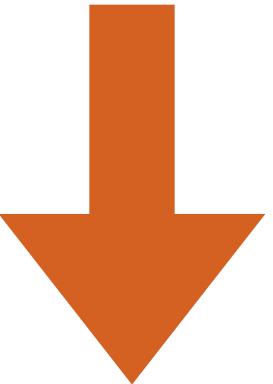
- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

Loop fission

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Loop fission



A

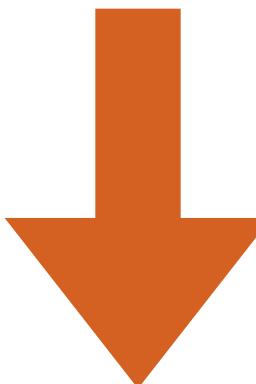
```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

Loop optimizations

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

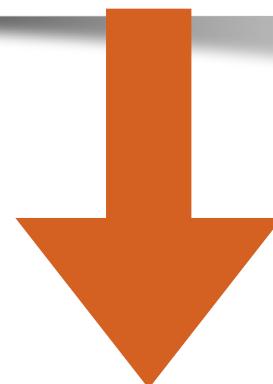
Loop fission



A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

Loop fusion



A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Tiling

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 $O(n^3)$

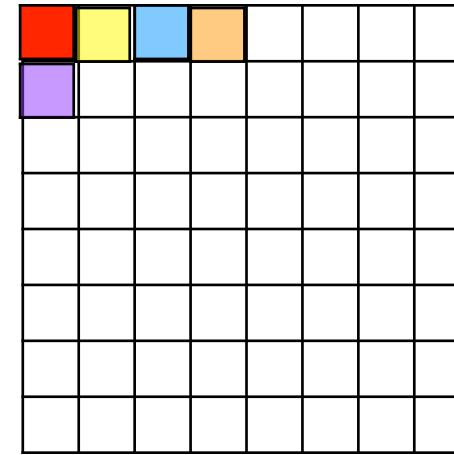
If $n=1024$, it takes about 1 sec

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

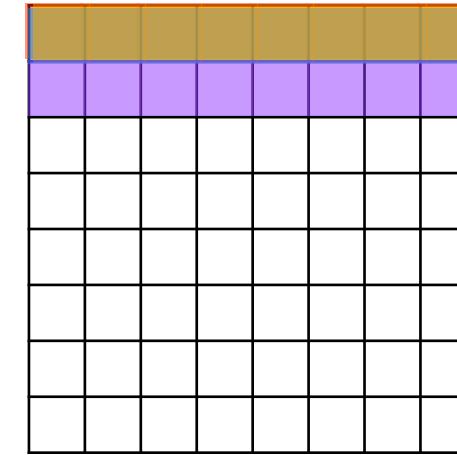
Matrix Multiplication

```
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];  
        }  
    }  
}
```

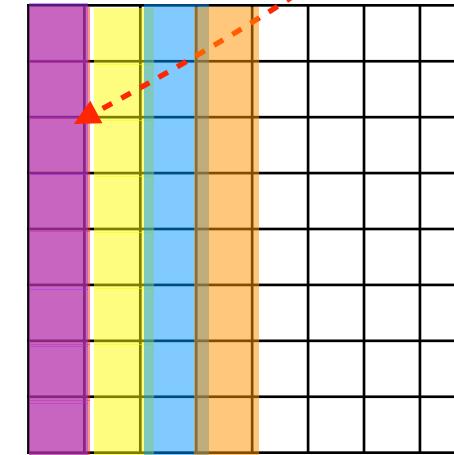
Very likely a miss if
array is large



c



a



b

- If each dimension of your matrix is 2048
 - Each row takes 2048×8 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!

Tiling algorithm for matrix multiplication

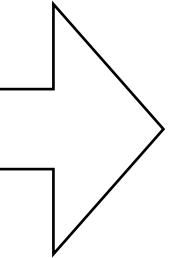
- Discover the cache miss rate
 - `valgrind --tool=cachegrind cmd`
 - `cachegrind` is a tool profiling the cache performance
 - Performance counter
 - Intel® Performance Counter Monitor <http://www.intel.com/software/pcm/>

Matrix Transpose

```
// 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++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}

// Compute on b_t
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_t[jj][kk];
        }
    }
}
```



Summary of Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Blocking/tiling — capacity miss, conflict miss
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity
- Loop interchange — conflict/capacity miss

Q & A



Announcement

- Reading quiz #3 due **next Tuesday** before the lecture
 - Please do read the textbook before/while taking the quiz
 - We take the “average”
- Assignment #2 due tonight
- Assignment #3 will be up tomorrow. Due in 2 weeks

Computer Science & Engineering

203

つづく

