

## Course Description

## Lecture 8: Cache Replacement Policies and Enhancements

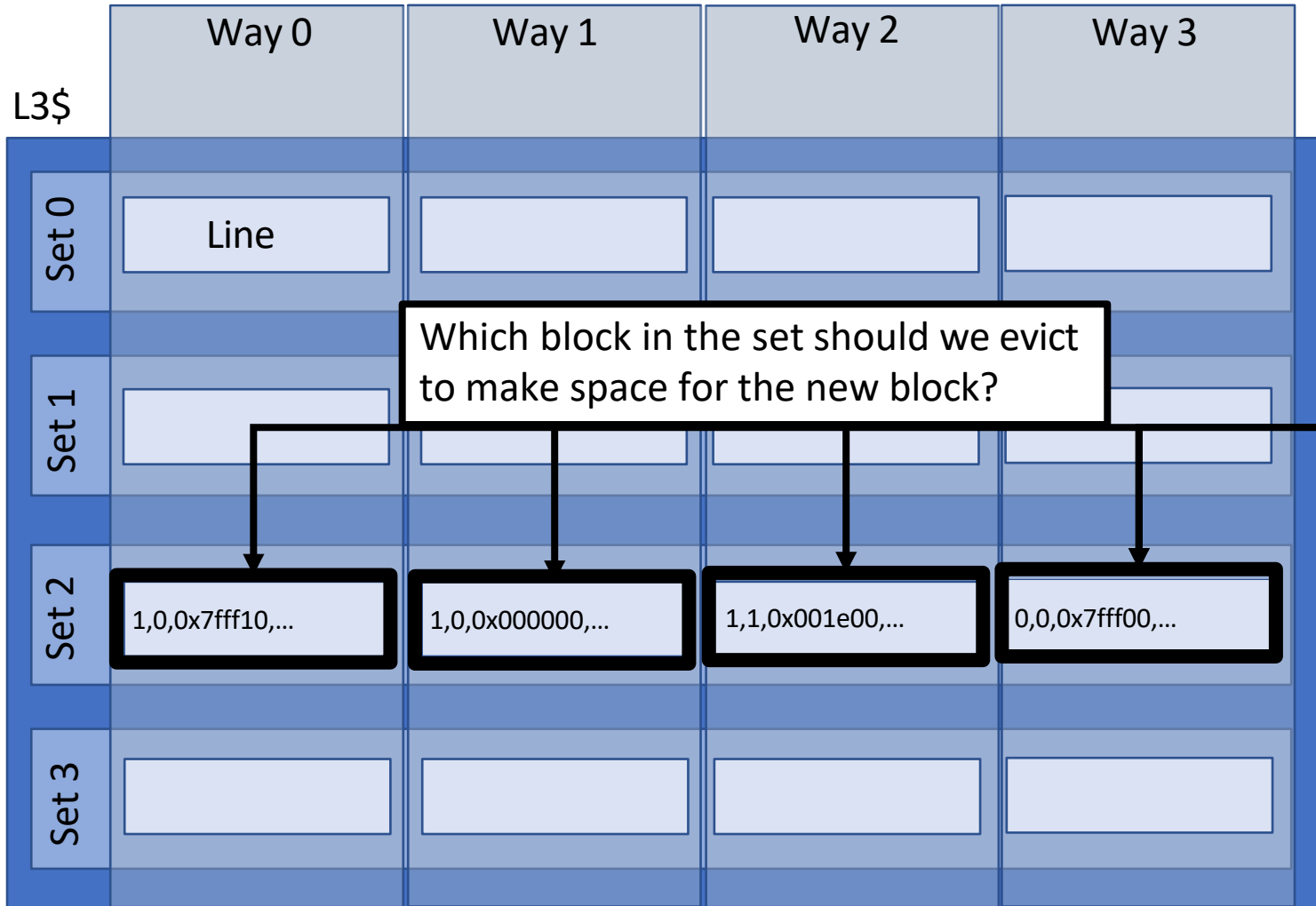
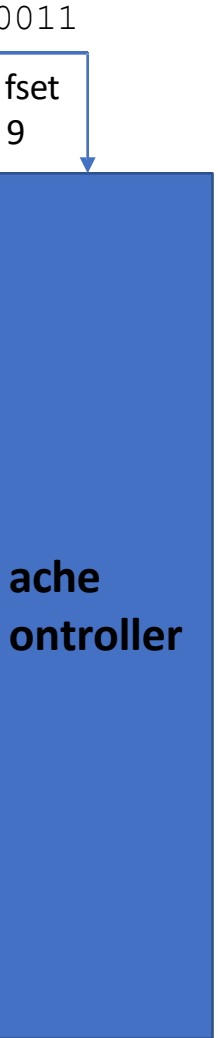
This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

**Credit: Brandon Lucia**

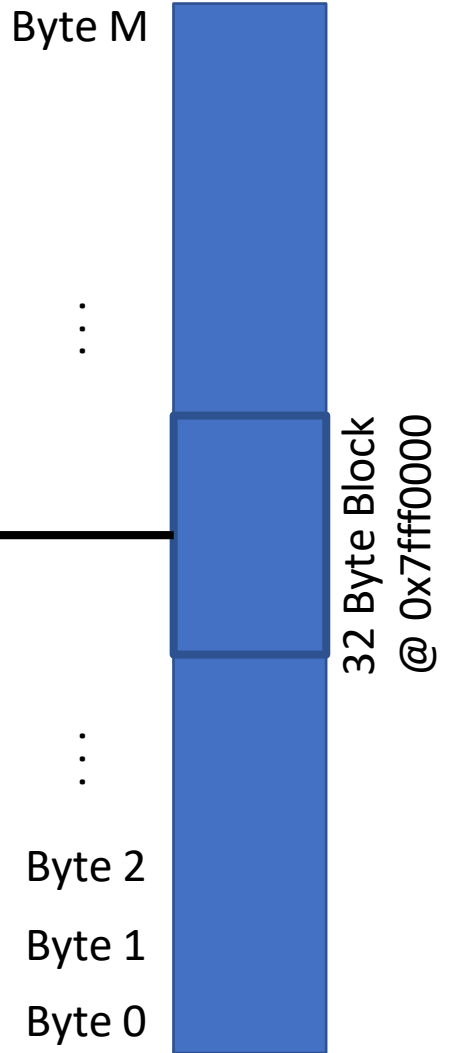
# Replacement Policies

# Replacement Policies

1b x6 0x7fff0053

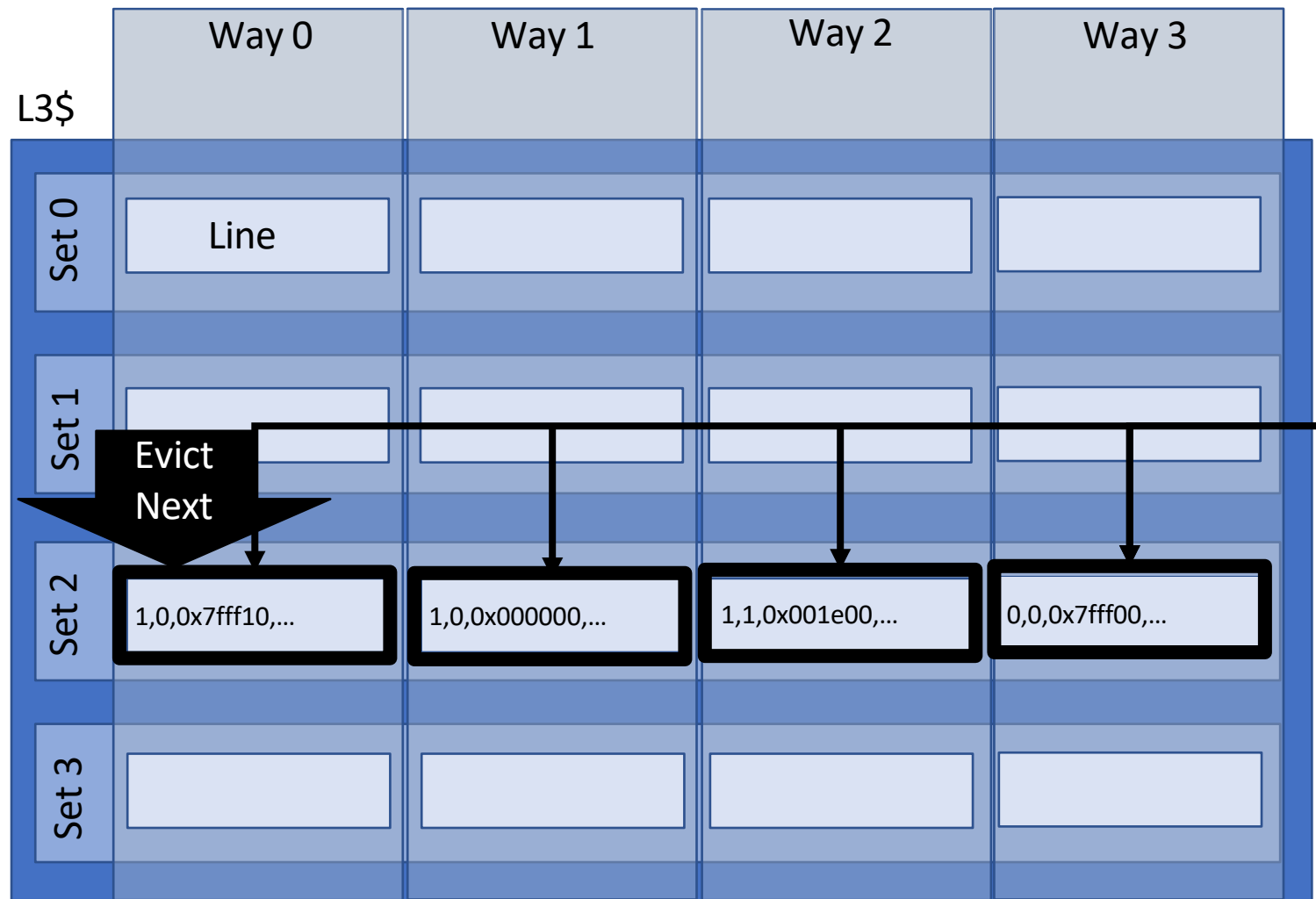
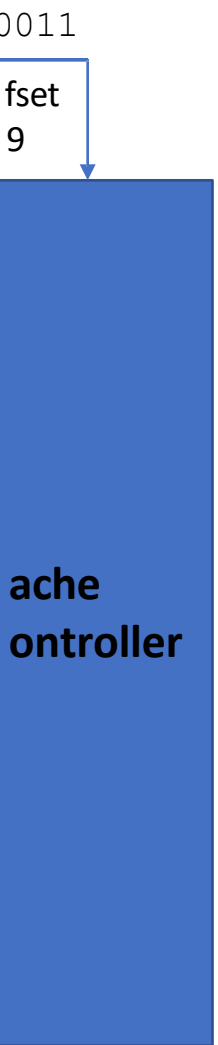


Which block in the set should we evict to make space for the new block?

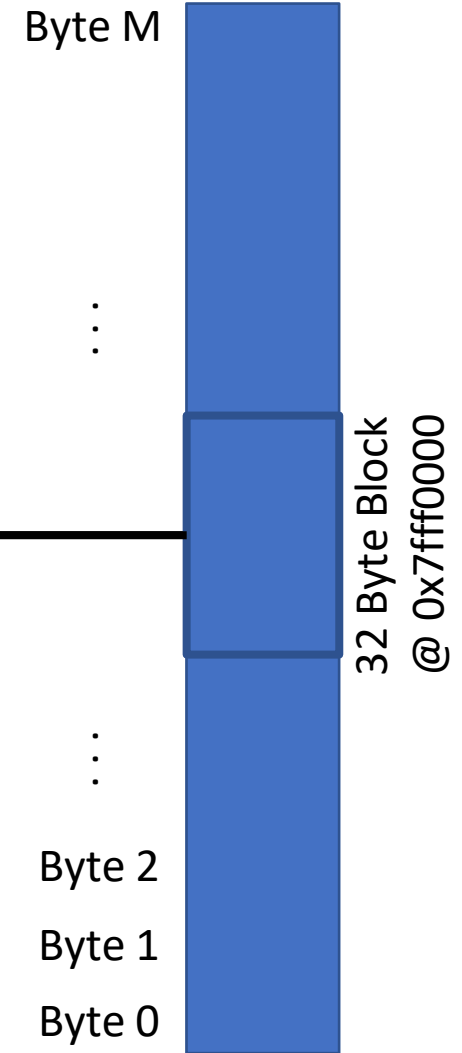


# Replacement Policies – Round Robin

1b x6 0x7fff0053



**Evict Next**

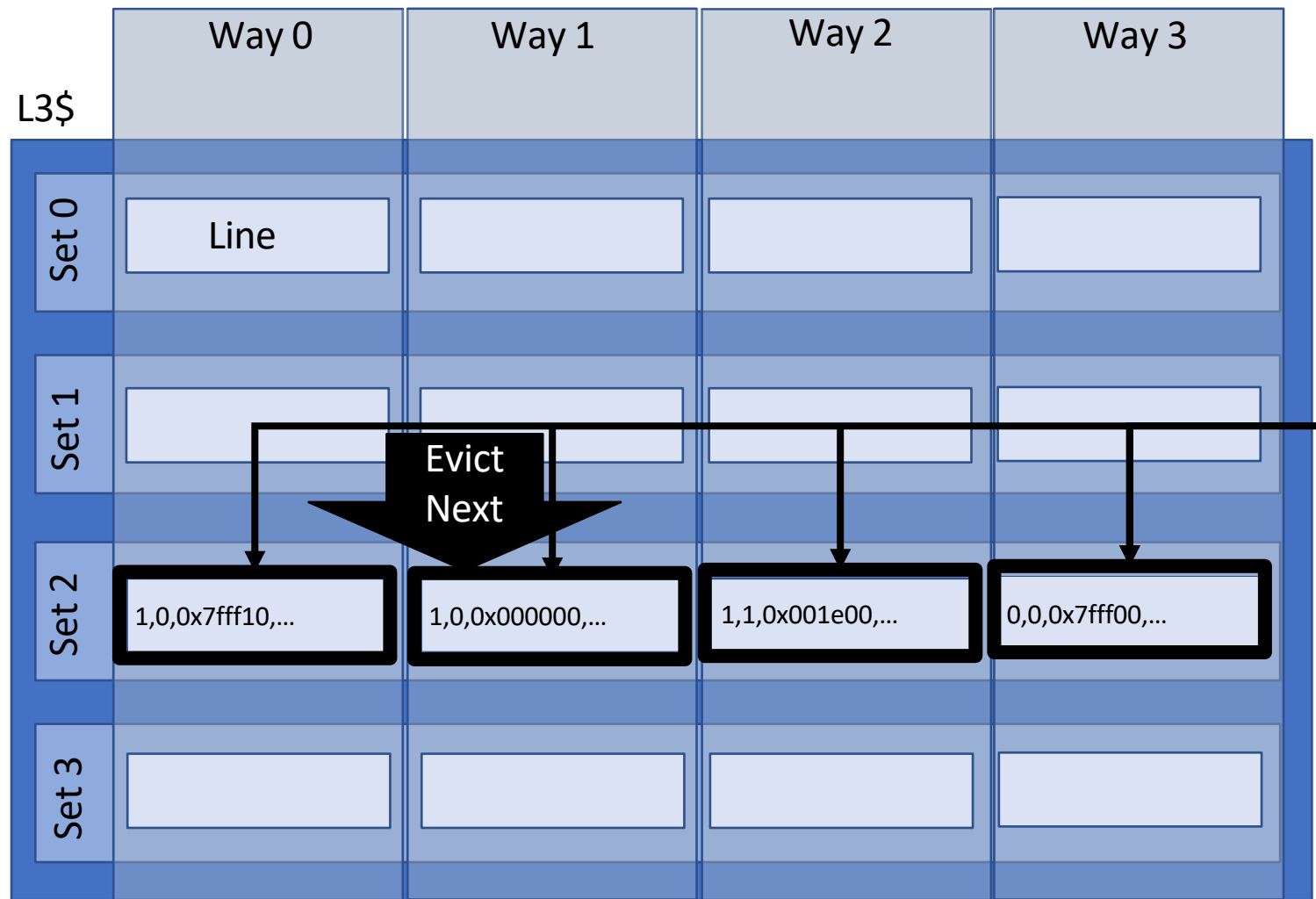
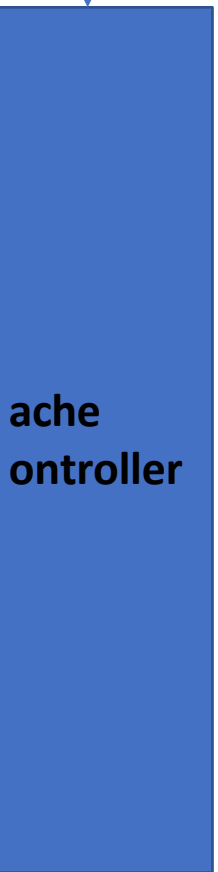


# Replacement Policies – Round Robin

1b x6 0x7fff0053

0011

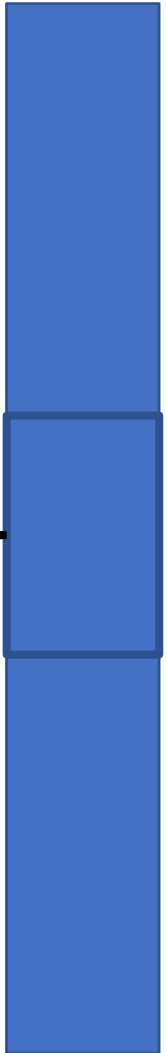
ffset  
9



Evict  
Next

Byte M

⋮



32 Byte Block @ 0x7fff0000

⋮

Byte 2

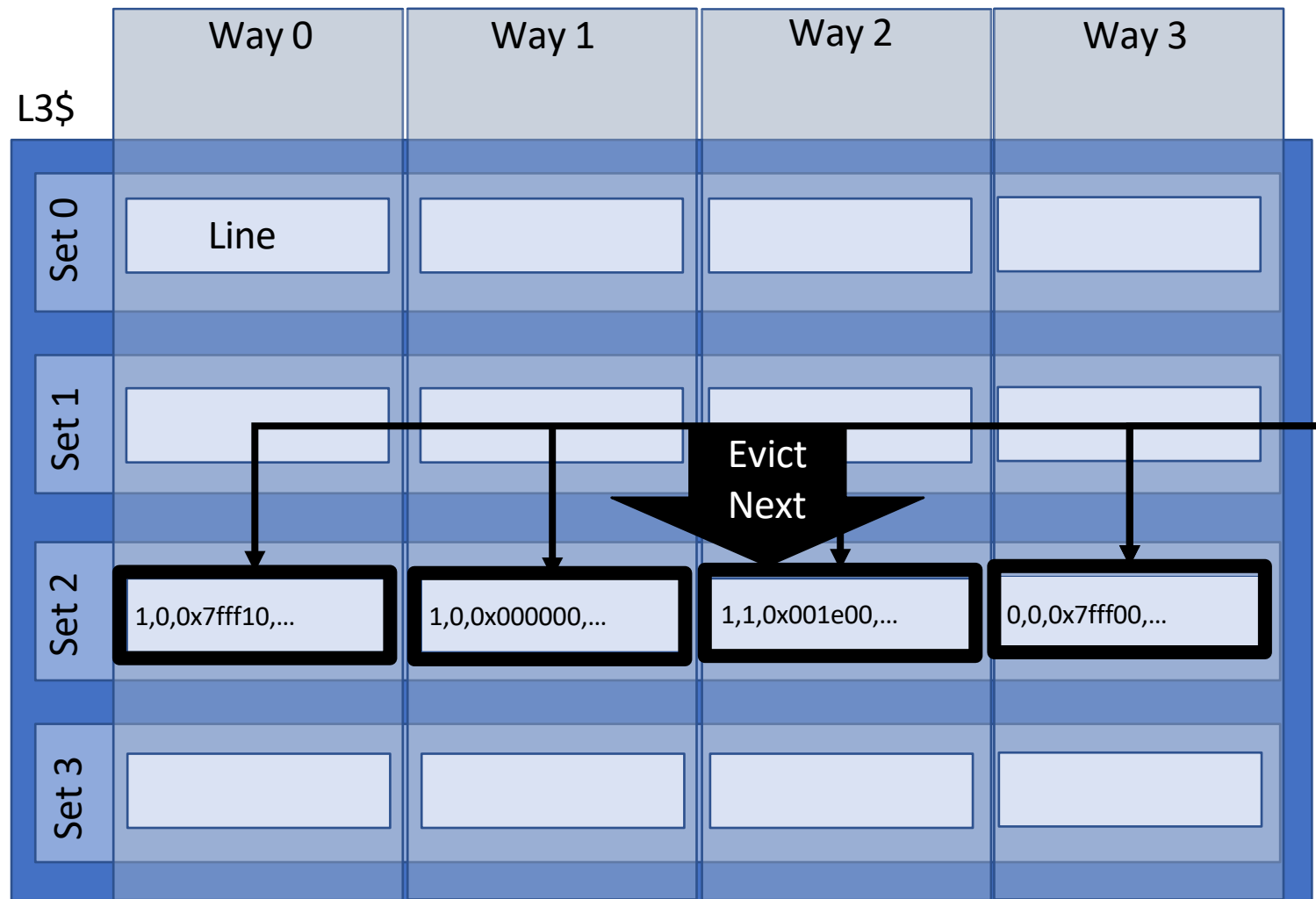
Byte 1

Byte 0

# Replacement Policies – Round Robin

1b x6 0x7fff0053

0011  
fset  
9  
Cache controller



Byte M

⋮

⋮

Byte 2

Byte 1

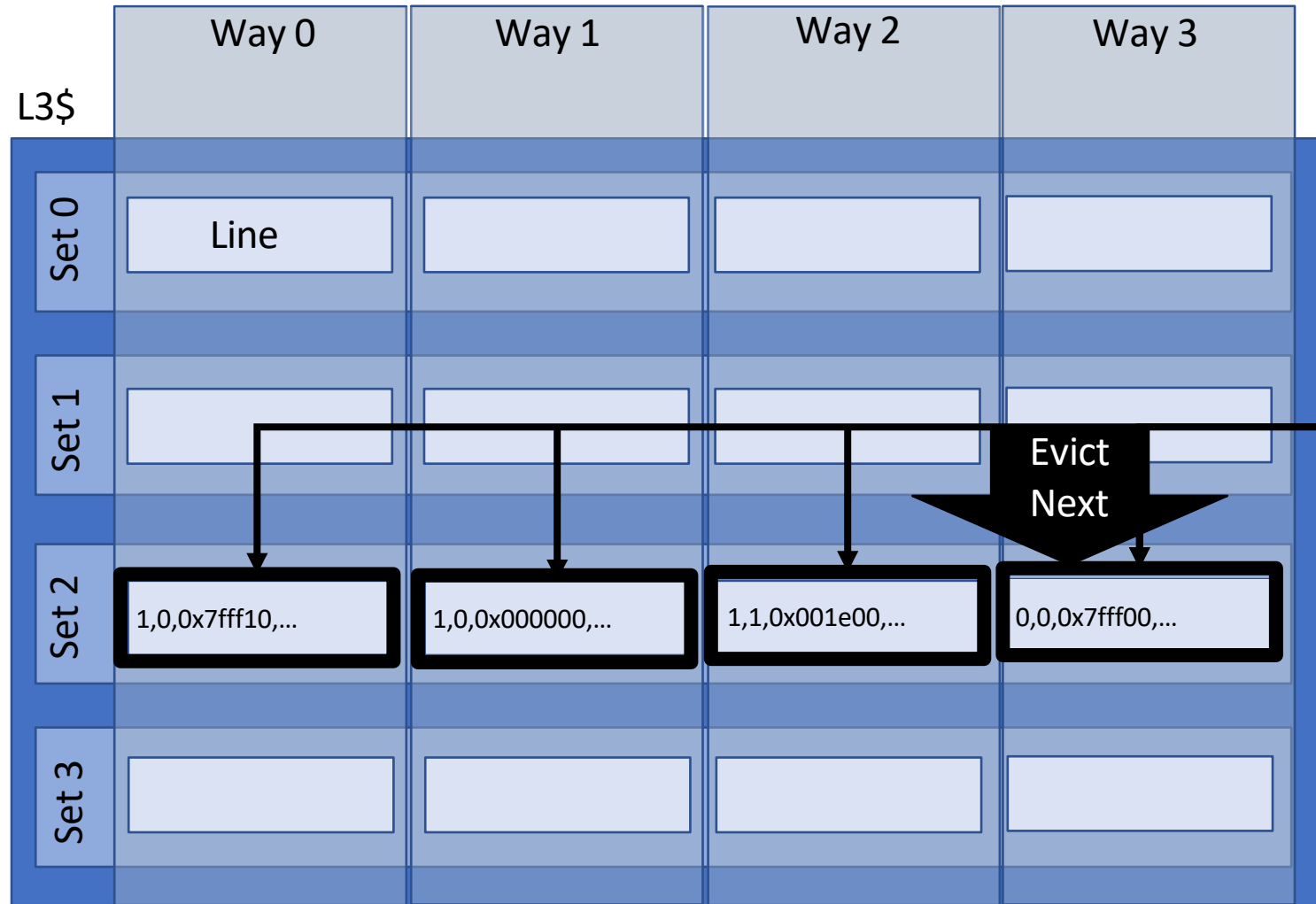
Byte 0

32 Byte Block  
@ 0x7fff0000

# Replacement Policies – Round Robin

1b x6 0x7fff0053

0011  
fset  
9  
ache  
ontroller



Byte M

⋮

Evict  
Next

32 Byte Block  
@ 0x7fff0000

⋮

Byte 2

Byte 1

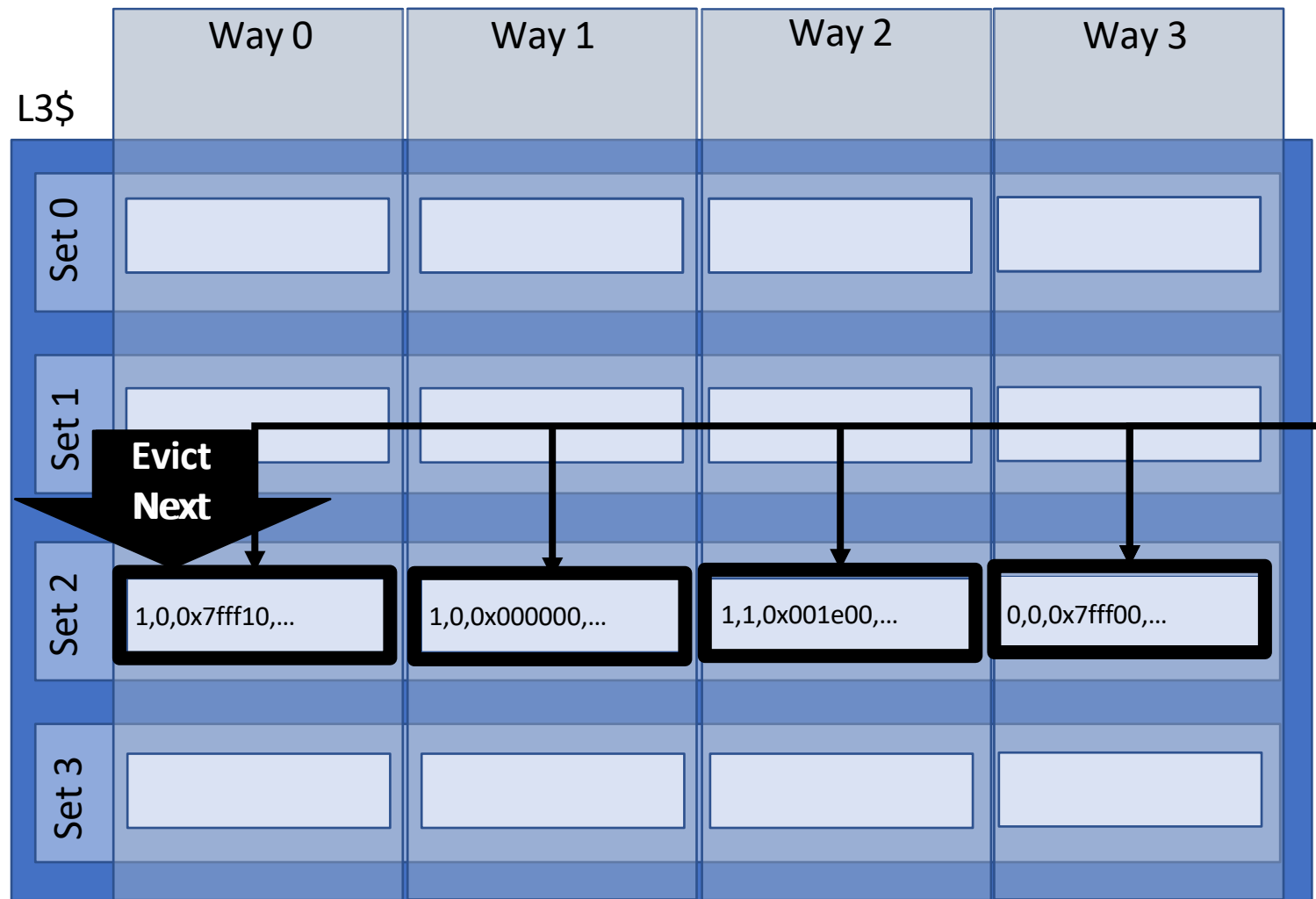
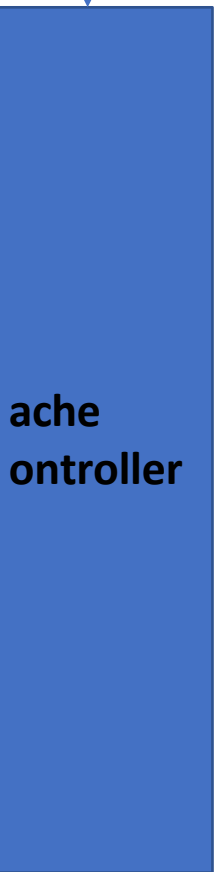
Byte 0

# Replacement Policies – Round Robin

1b x6 0x7fff0053

0011

ffset  
9



**Evict Next**

Byte M

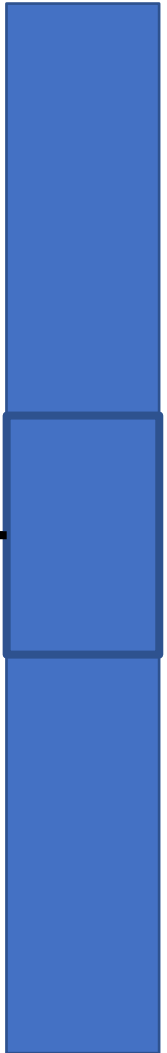
...

...

Byte 2

Byte 1

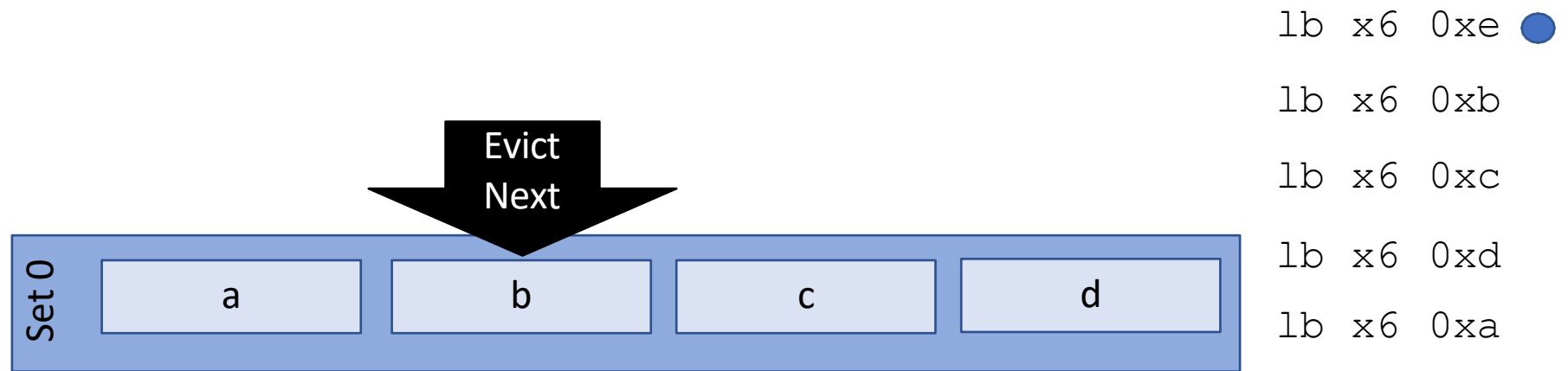
Byte 0



32 Byte Block @ 0x7fff0000



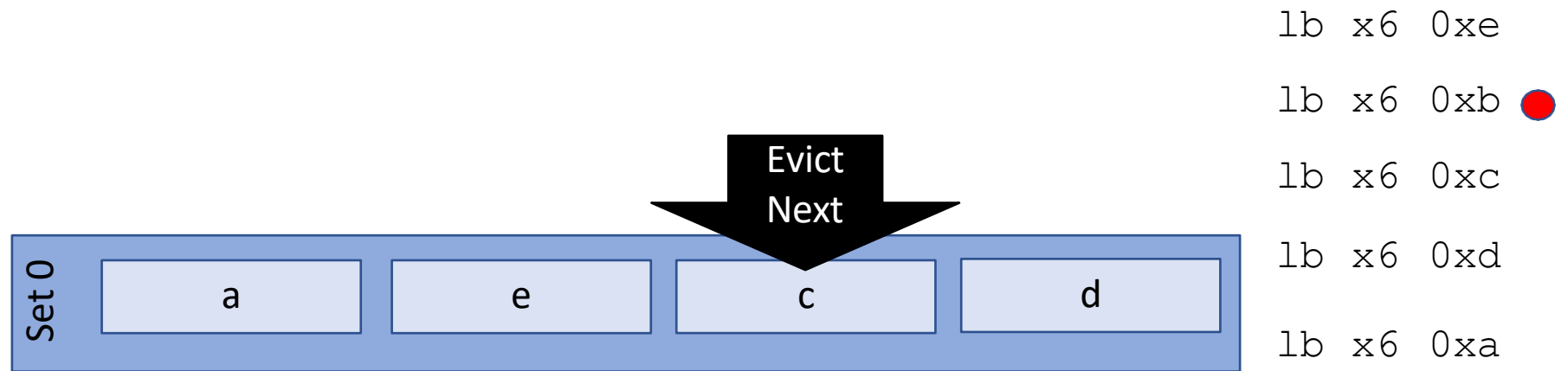
# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

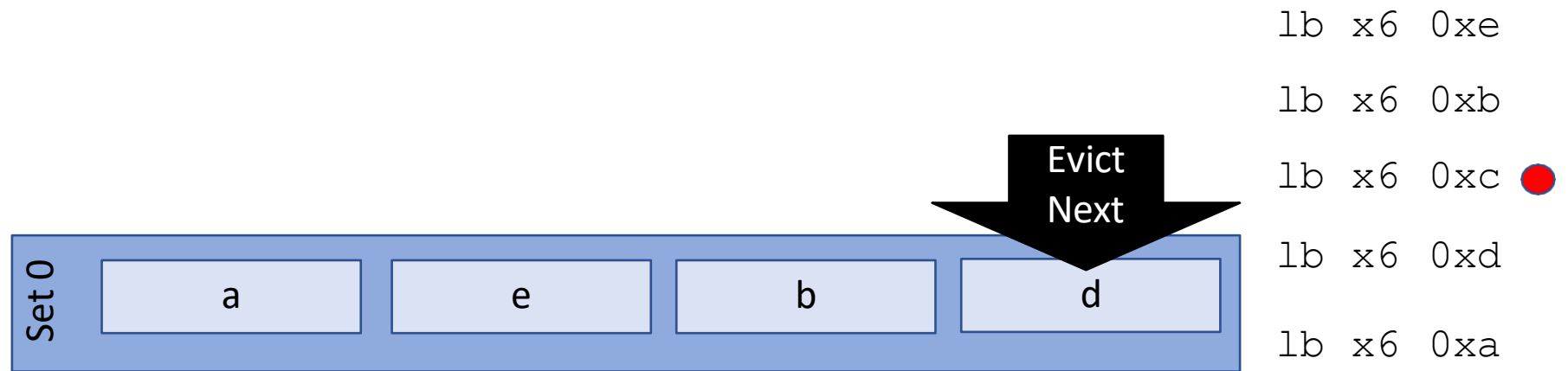
# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

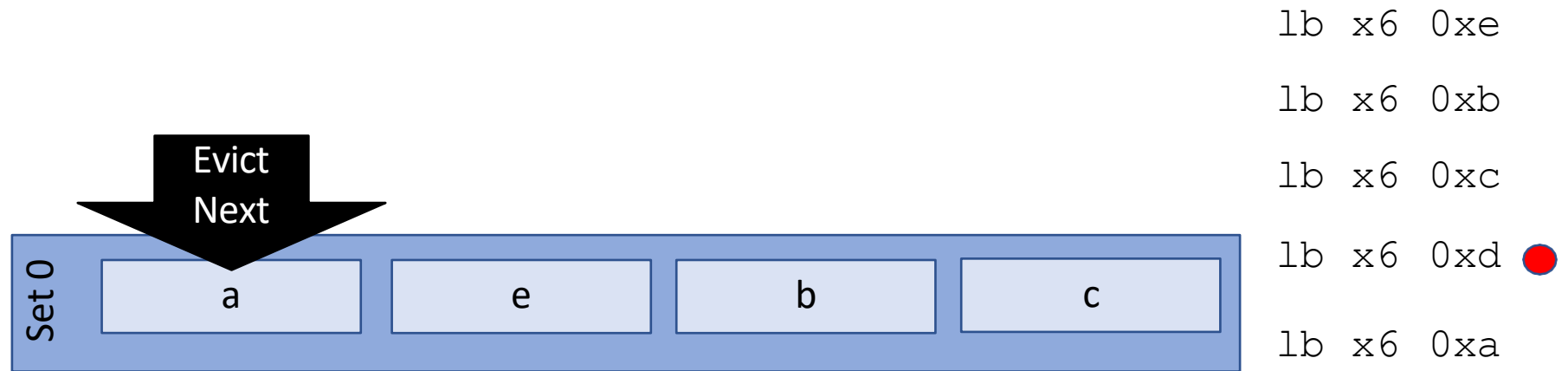
# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

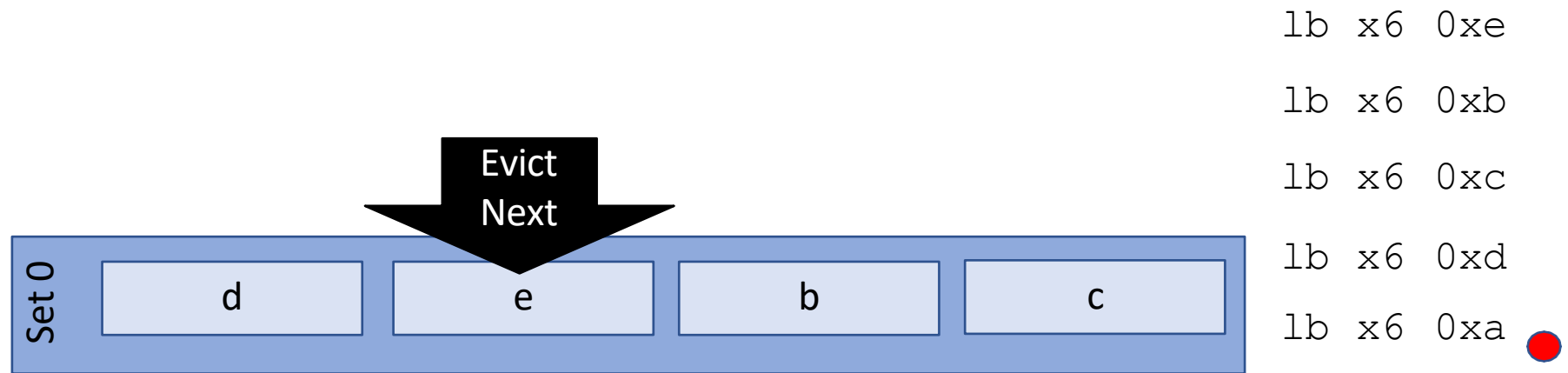
# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

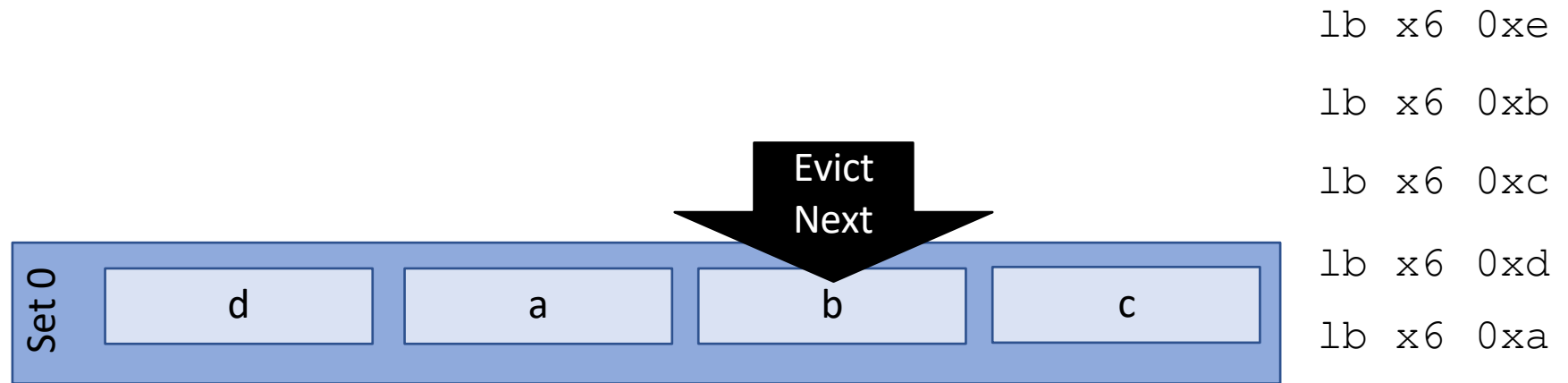
# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

# Replacement Policies – Round-Robin Analysis



Advantage: Simple to implement and understand

Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed...

# Minimum Number of Misses?



1b x6 0xe

1b x6 0xb

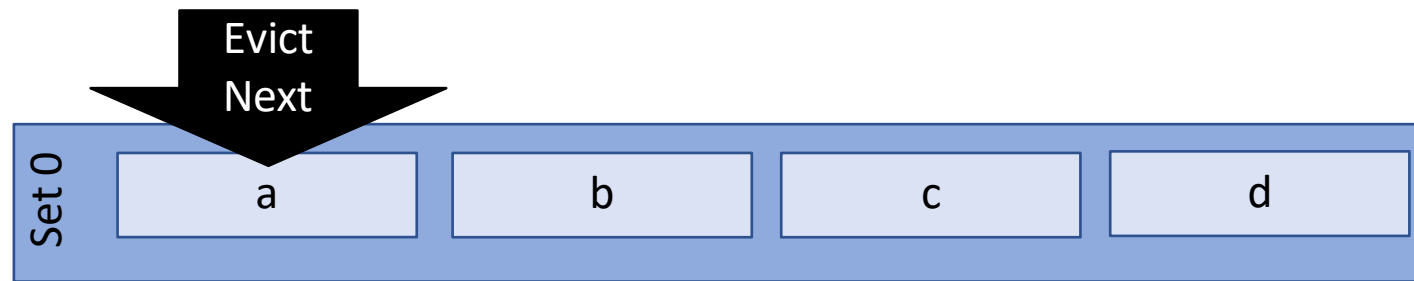
1b x6 0xc

1b x6 0xd

1b x6 0xa

What is the best replacement strategy to minimize misses & **why**?

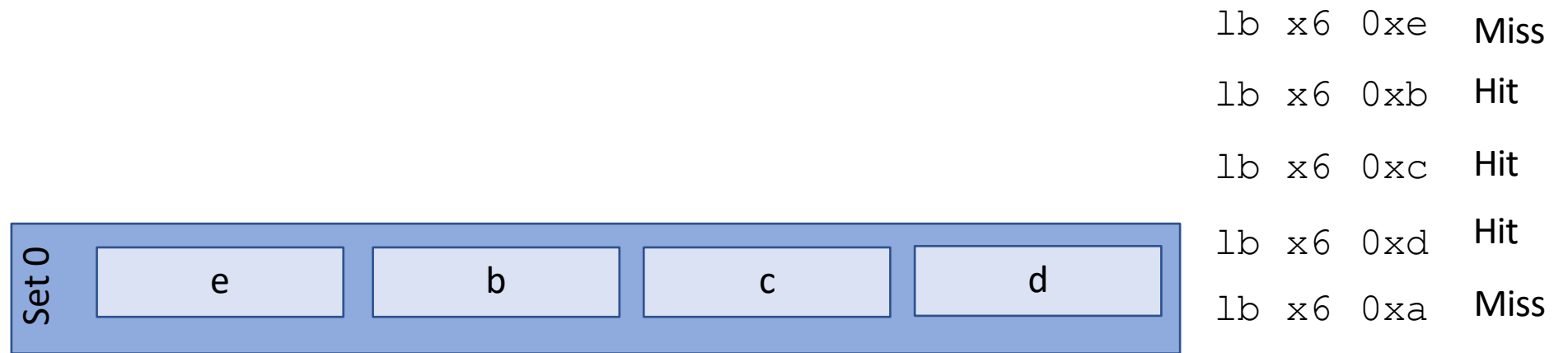
# Minimum Number of Misses?



- 1b x6 0xe ●
- 1b x6 0xb
- 1b x6 0xc
- 1b x6 0xd
- 1b x6 0xa



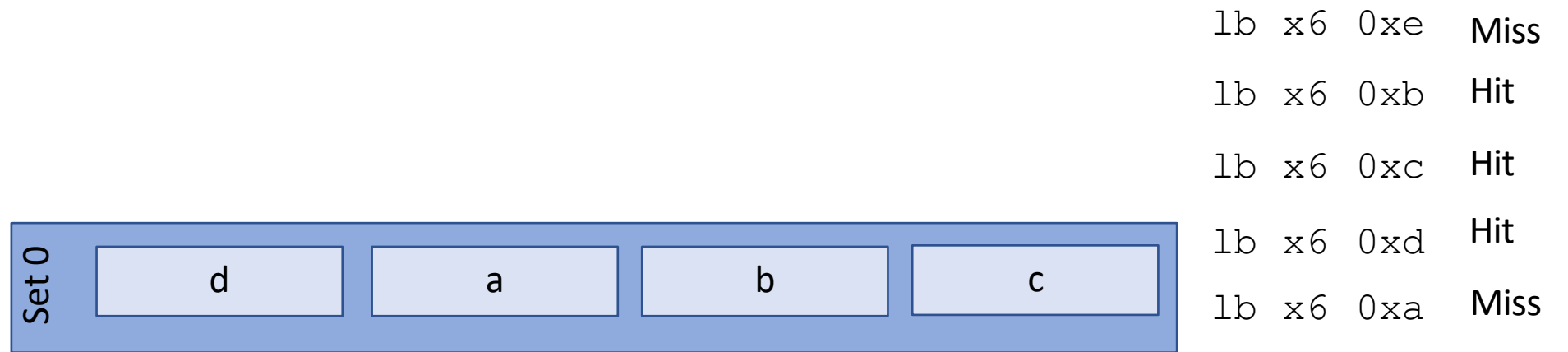
# When are we going to re-use cached data?



Replacement decisions must be informed by the next **reuse** of a block of data.

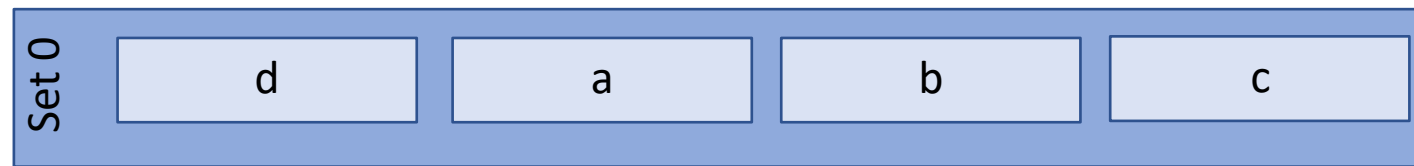
**Think: what is an optimal policy? How far in the future is something going to be used again?**

# When are we going to re-use cached data?



What defines optimality for a cache replacement algorithm?

# Belady's MIN Algorithm for Optimal Replacement



1b	x6	0xe	Miss
1b	x6	0xb	Hit
1b	x6	0xc	Hit
1b	x6	0xd	Hit
1b	x6	0xa	Miss

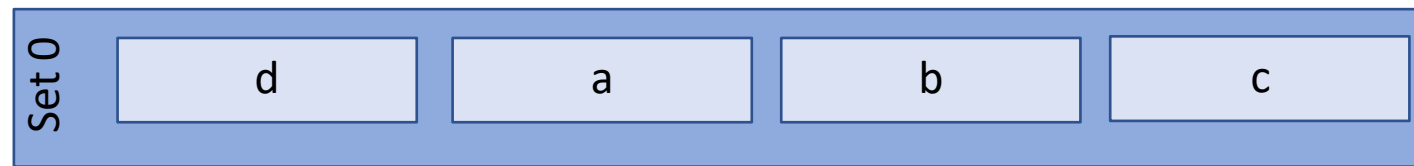


**Bélády László:**

“What defines optimality for a cache replacement algorithm?”

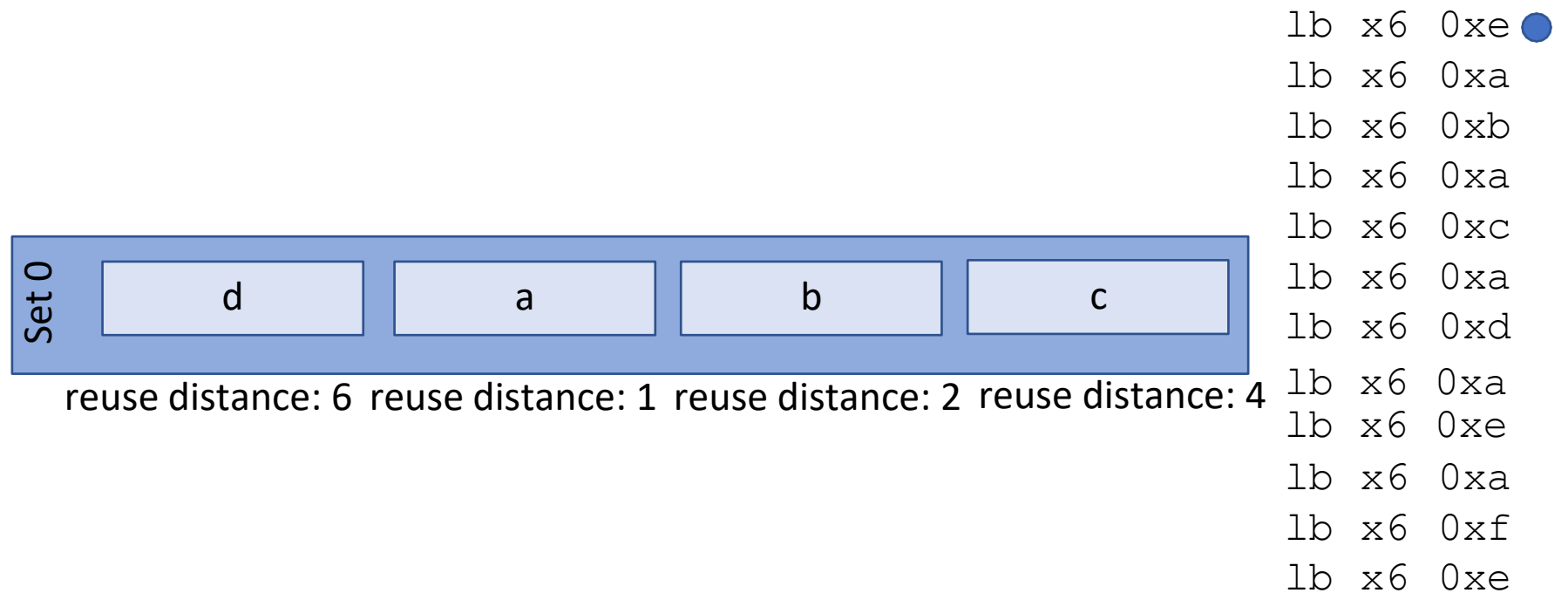
**Evict the cached element that will be used furthest in the future.**

# Belady's MIN Algorithm for Optimal Replacement

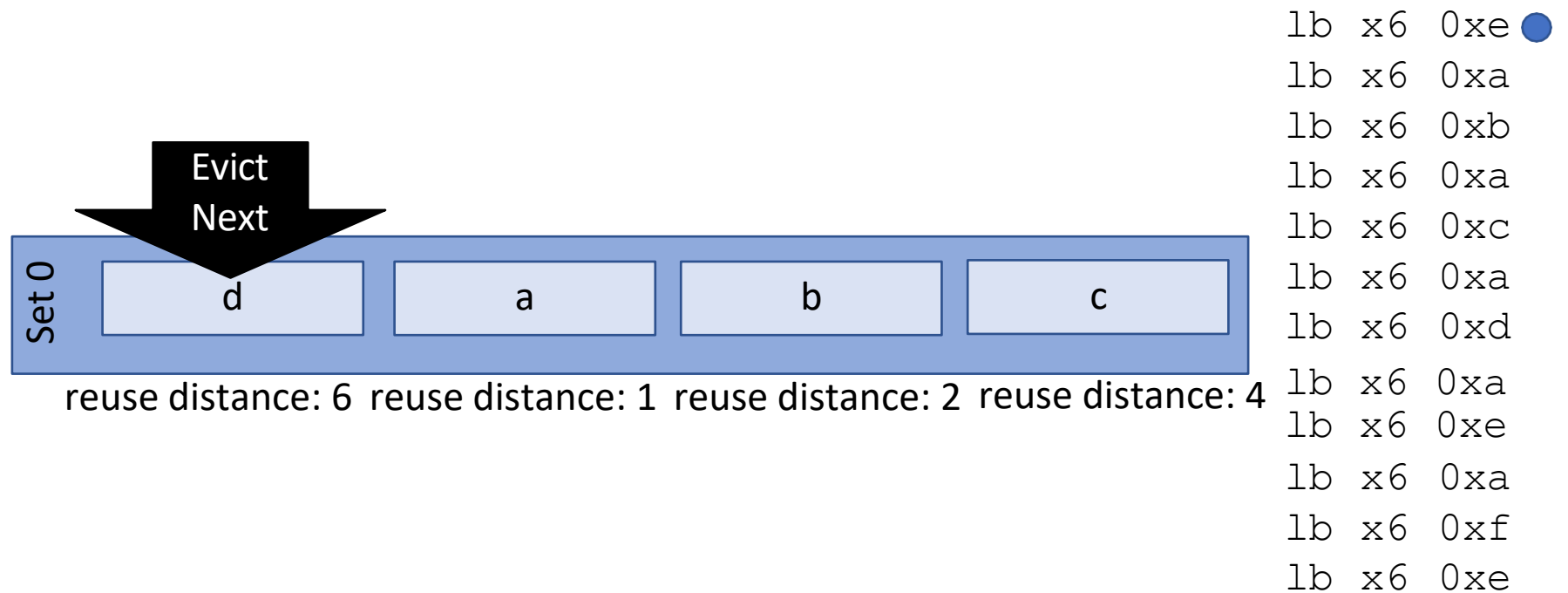


- 1b x6 0xe ●
- 1b x6 0xa
- 1b x6 0xb
- 1b x6 0xa
- 1b x6 0xc
- 1b x6 0xa
- 1b x6 0xd
- 1b x6 0xa
- 1b x6 0xe
- 1b x6 0xa
- 1b x6 0xf
- 1b x6 0xe

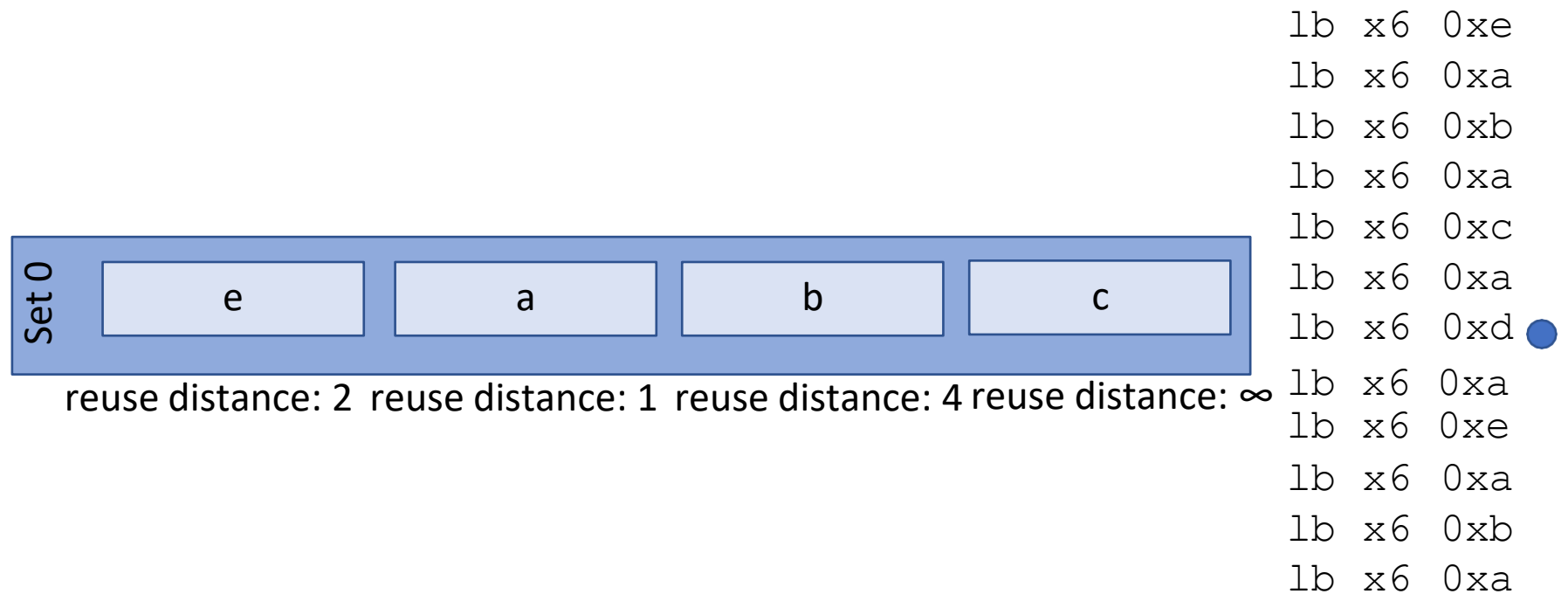
# Belady's MIN Algorithm for Optimal Replacement



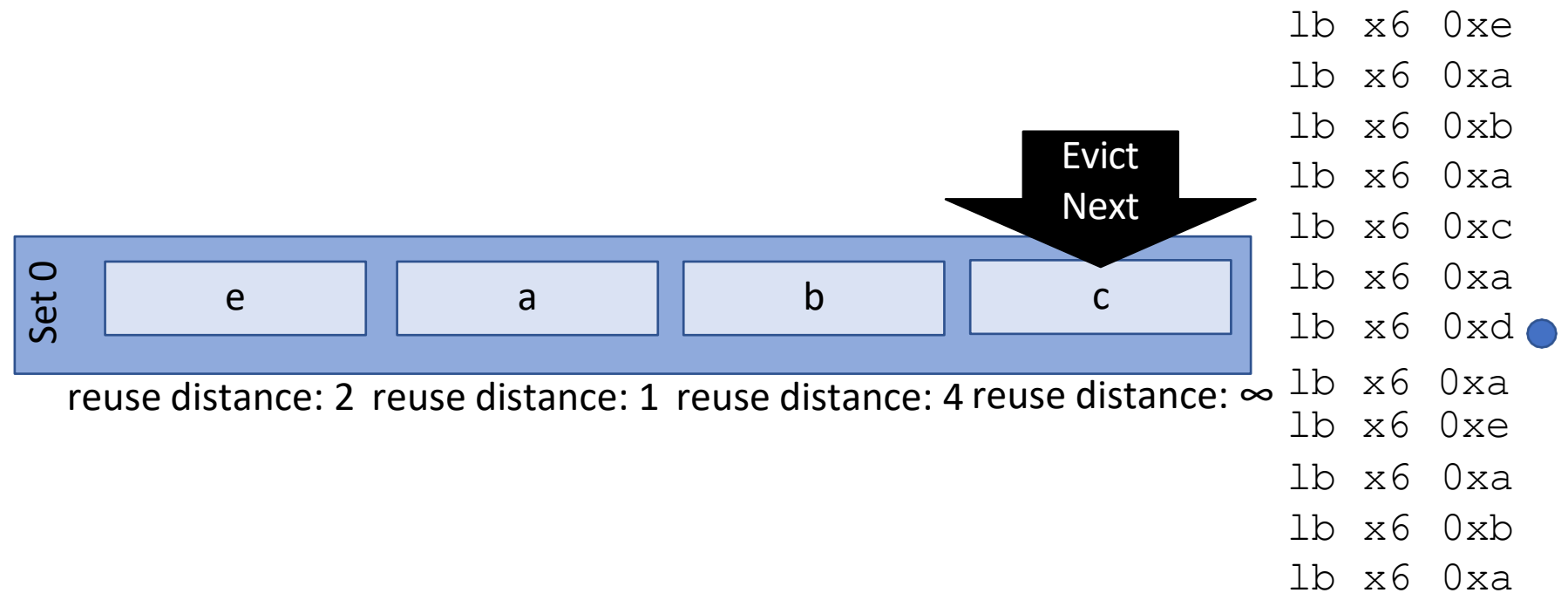
# Belady's MIN Algorithm for Optimal Replacement



# Belady's MIN Algorithm for Optimal Replacement



# Belady's MIN Algorithm for Optimal Replacement





# Belady's MIN Algorithm for Optimal Replacement

```
findBlockMIN() {
    0://init reuse distances
    1:for each block in cache, b:
    2:    RD[b] = 0; RD_done[b] = false;
    3://look forward in the execution trace
    4:for each access, a, forward in execution trace:
    5://increment reuse distance for each block not already seen
    6:    for each block in cache, b:
    7:        if RD_done[b] == false:
    8:            RD[b]++;
    9:    RD_done[a.block] = true
    10://MIN finds the block with maximum RD
    11:return argmax(b, RD[b])
}
```

**MIN results in the MINimum number of replacements in a cache for an execution trace.**

# Belady's MIN Algorithm for Optimal Replacement

```
findBlockMIN() {
    0://init reuse distances
    1:for each block in cache, b:
    2:    RD[b] = 0; RD_done[b] = false;
    3://look forward in the execution trace
    4:for each access, a, forward in execution trace:
    5://increment reuse distance for each block not already seen
    6:    for each block in cache, b:
    7:        if RD_done[b] == false:
    8:            RD[b]++;
    9:    RD_done[a.block] = true
    10://MIN finds the block with maximum RD
    11:return argmax(b, RD[b])
}
```

**See any limitations of the MIN algorithm for cache replacement?**

# Belady's MIN Algorithm for Optimal Replacement

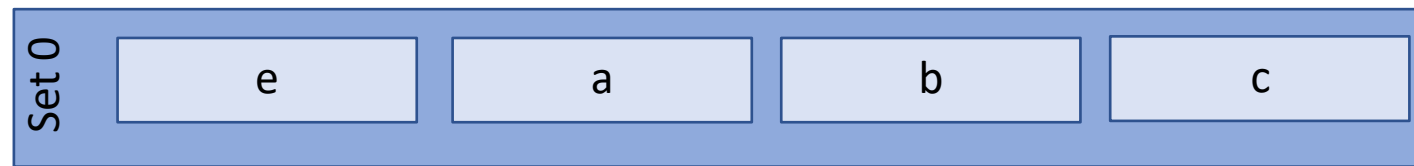
```
findBlockMIN() {
    0://init reuse distances
    1:for each block in cache, b:
    2:    RD[b] = 0; RD_done[b] = false;
    3://look forward in the execution trace
    4:for each access, a, forward in execution trace:
    5://increment reuse distance for each block not already seen
    6:    for each block in cache, b:
    7:        if RD_done[b] == false:
    8:            RD[b]++;
    9:    RD_done[a.block] = true
    10://MIN finds the block with maximum RD
    11:return argmax(b, RD[b])
}
```

**Need omniscient future knowledge of the execution trace of your program!**

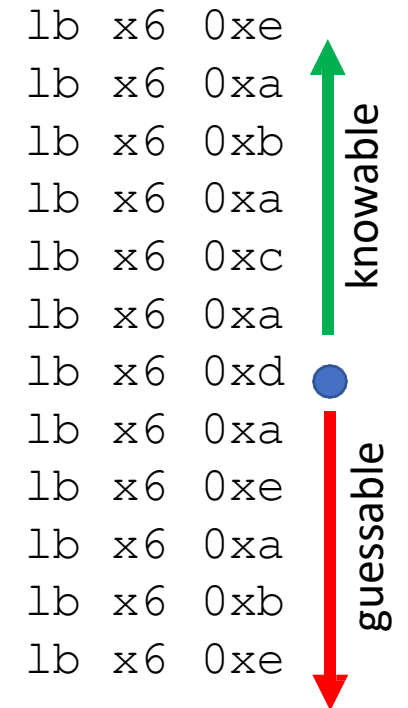
**MIN is optimal, but not practically implementable**

# Practical Replacement Algorithms

General idea: Assume the near future is similar to the recent past

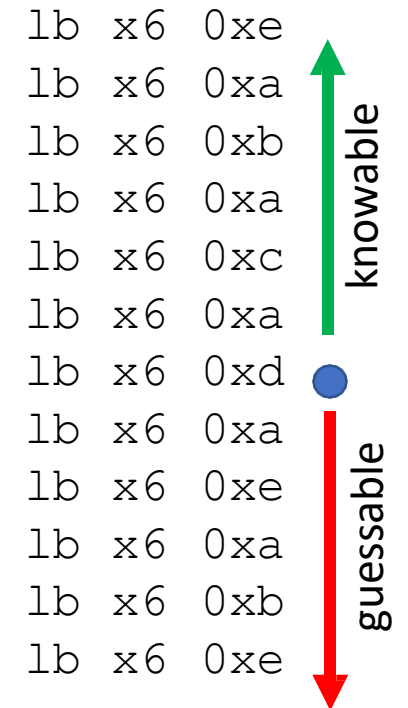
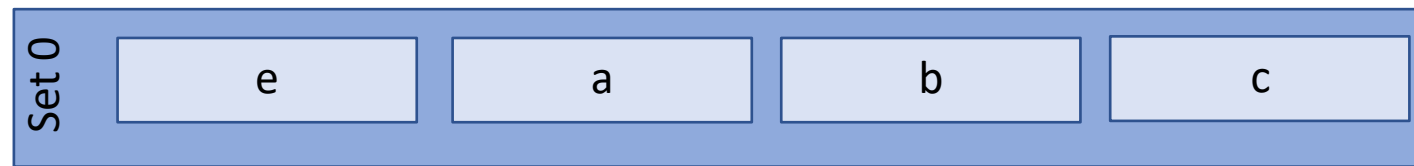


**If a block was used recently, it will be used again soon**



# Least-Recently Used (LRU) Replacement

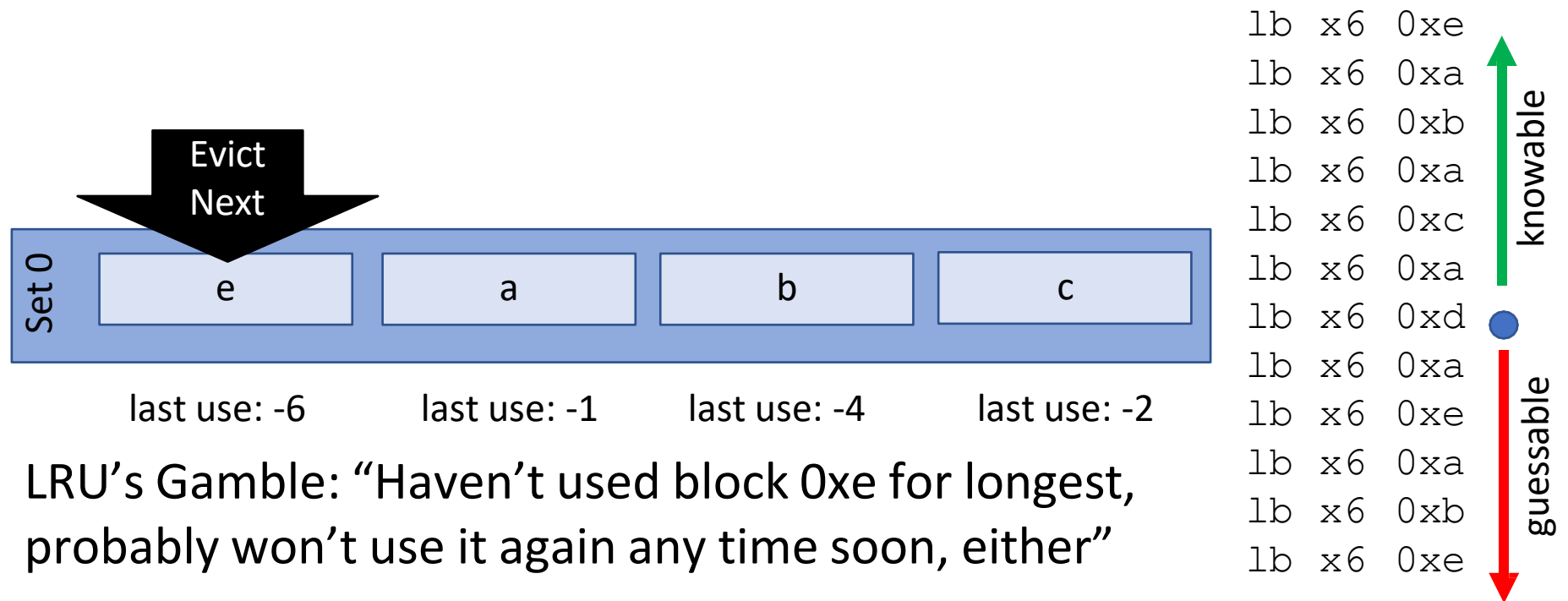
Evict the block that was used the furthest in the execution's past



If a block was **not** used recently, it will **not** be used again soon

# Least-Recently Used (LRU) Replacement

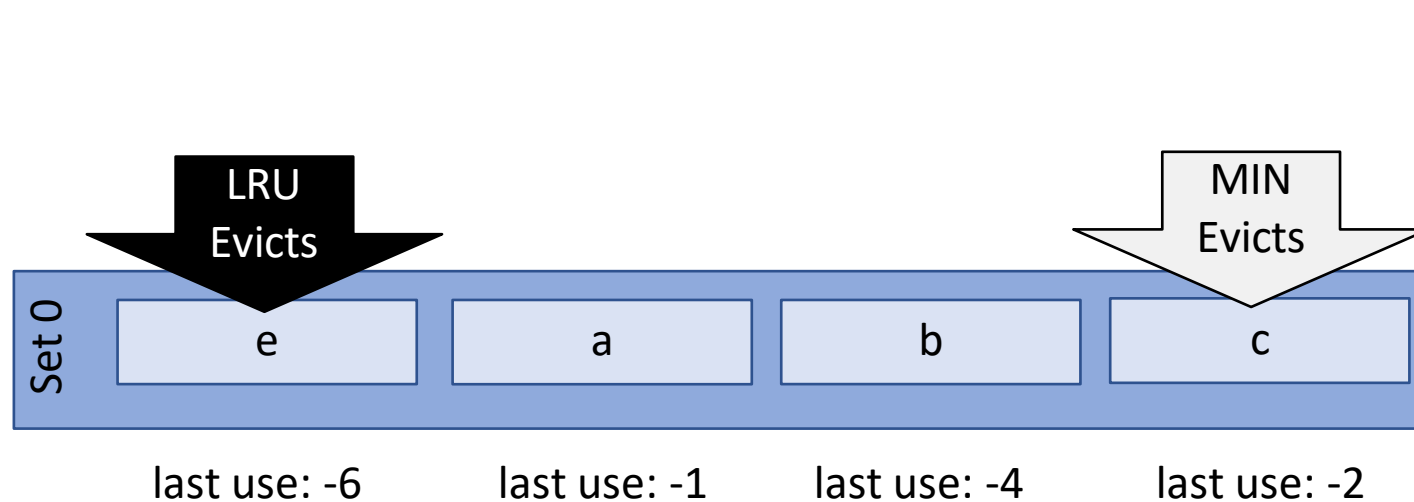
Evict the **block** that was used the furthest in the execution's past



If a block was **not** used recently, it will **not** be used again soon

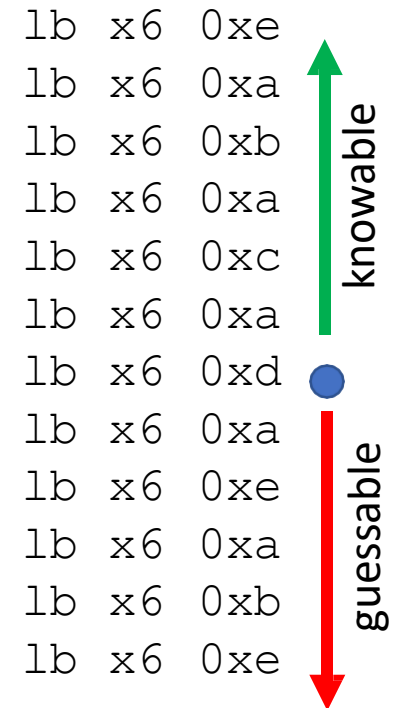
# Least-Recently Used (LRU) Replacement

Evict the block that was used the furthest in the execution's past



Caveat: LRU is wrong if past does not predict future

**Caveat to caveat: past *usually* predicts future well**



If a block was **not** used recently, it will **not** be used again soon

# (Naïve) LRU Algorithm & Implementability

```
accessCacheLRU(access a) {  
    for each block in cache, b:  
        if b != a.block:  
            LRU_Age[b]++  
        LRU_Age[b] = 0  
}
```

```
findBlockLRU() {  
    return argmax(b, LRU_Age)  
}
```

**Implementability** and **limitations** of LRU?



# (Naïve) LRU Algorithm & Implementability

```
accessCacheLRU(access a) {  
    for each block in cache, b:  
        if b != a.block:  
            LRU_Age[b]++  
        LRU_Age[b] = 0  
}
```

```
findBlockLRU() {  
    return argmax(b, LRU_Age)  
}
```

**Implementability!** Does not require unknowable information about future of execution

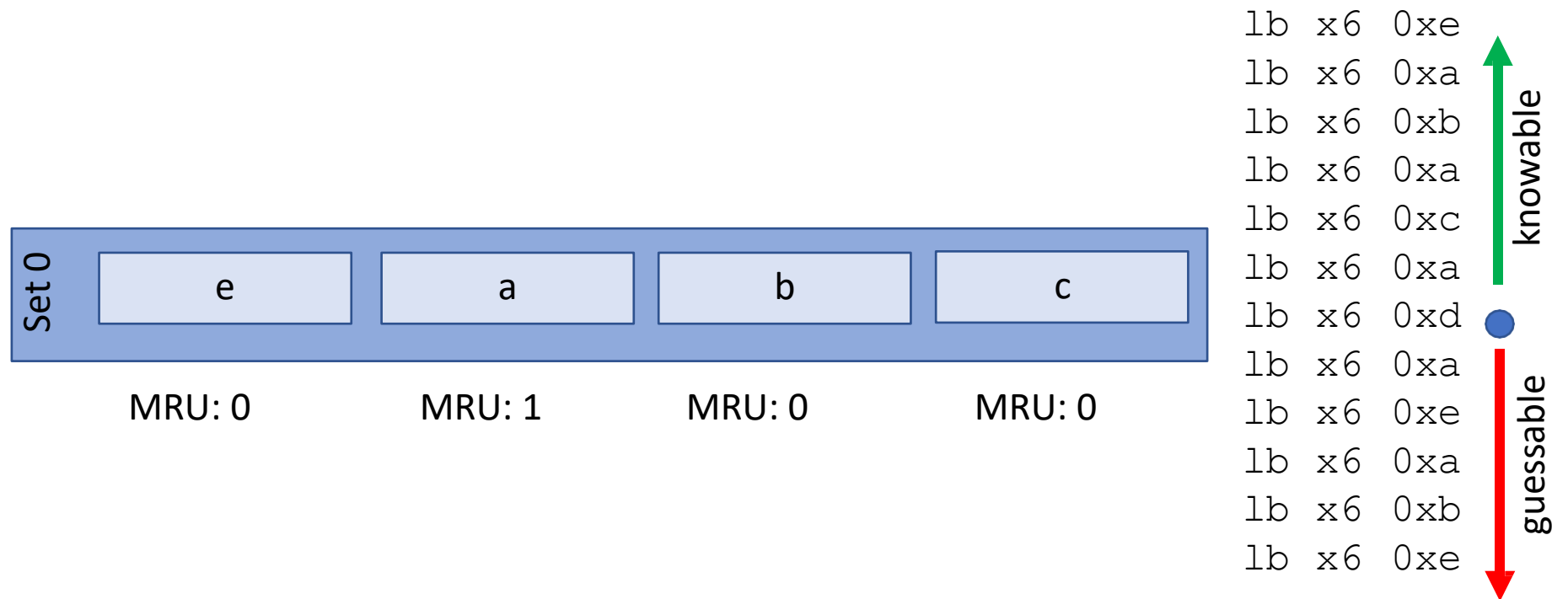
**Limitation!** Requires accessing metadata for *every* block on each access to *any* block.

**Time & energy cost to update ages. Area & power cost to store age values.**

***Does not scale beyond about 4 way set associativity.***

# Bit-Pseudo-Least-Recently Used (Bit-PLRU)

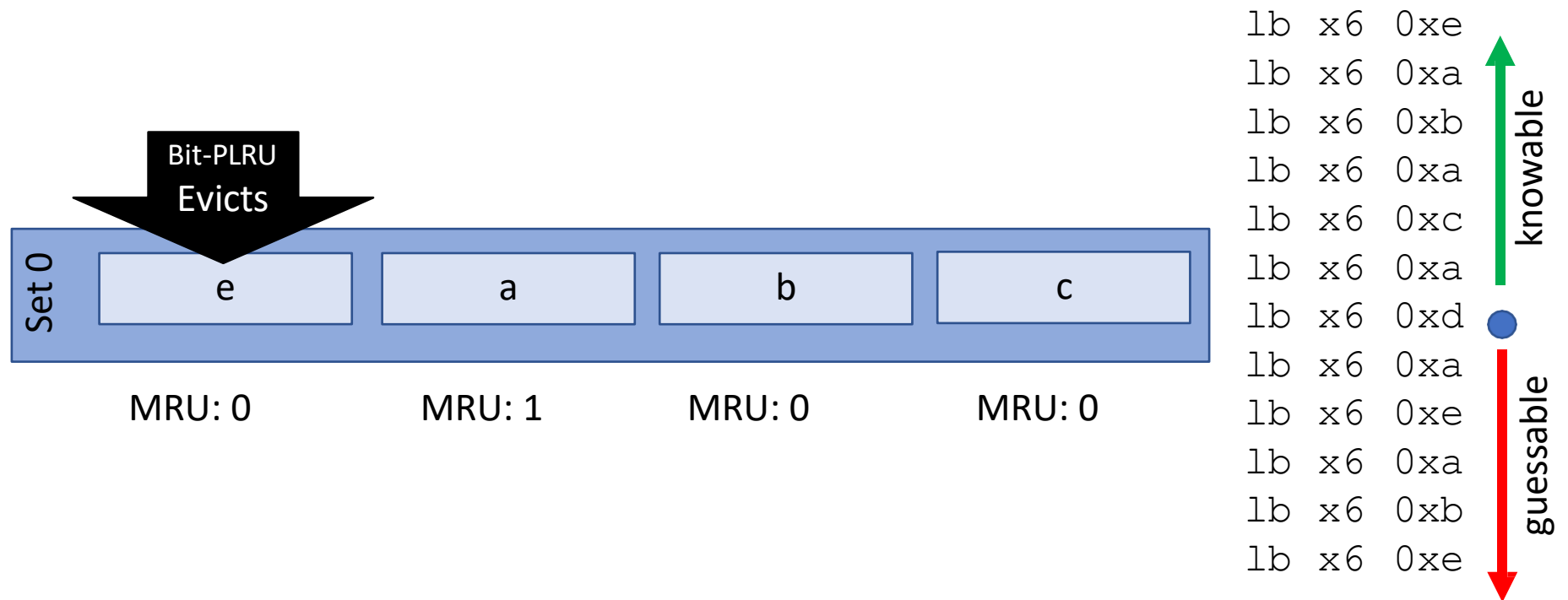
Evict a block that was definitely not most recently used



Set MRU bit when block is used (most recently), clear all MRU bits when all MRU bits are set, evict the left-most block with unset MRU bit

# Bit-Pseudo-Least-Recently Used (Bit-PLRU)

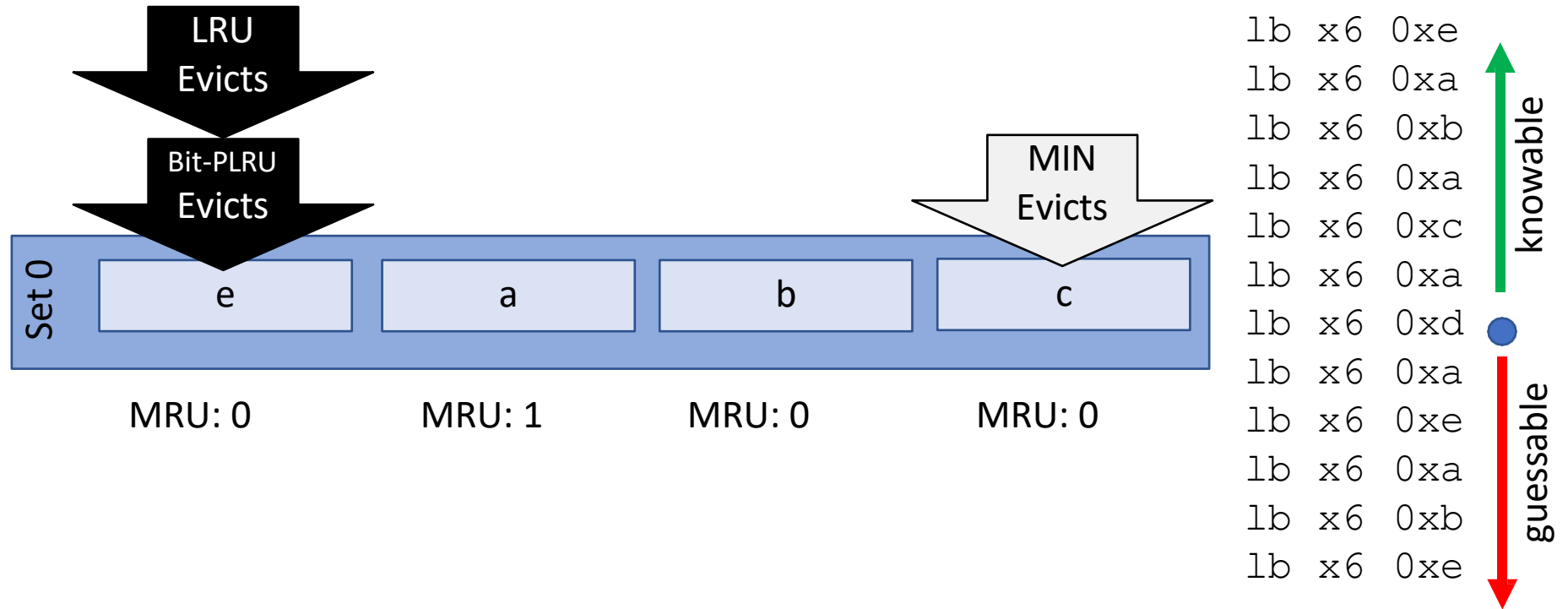
Evict a block that was definitely not most recently used



Set MRU bit when block is used (most recently), clear all MRU bits when all MRU bits are set, evict the left-most block with unset MRU bit

# Bit-Pseudo-Least-Recently Used (Bit-PLRU)

Evict a block that was definitely not most recently used



Bit-PLRU is a decent approximation of LRU

# Bit-PLRU Algorithm & Implementability

```
accessCachePLRU(access a){
    MRU_Bit[a.block] = 1
    if ++MRU_BitSum == setSize:
        for each block in cache, b:
            MRU_Bit[b] = 0
        MRU_BitSum = 0
}
```

```
findBlockLRU(){
    for i in 0..setSize:
        if !MRU_Bit[i]:
            return block(i);
}
```

**Implementability** and **limitations** of Bit-PLRU?

# Bit-PLRU Algorithm & Implementability

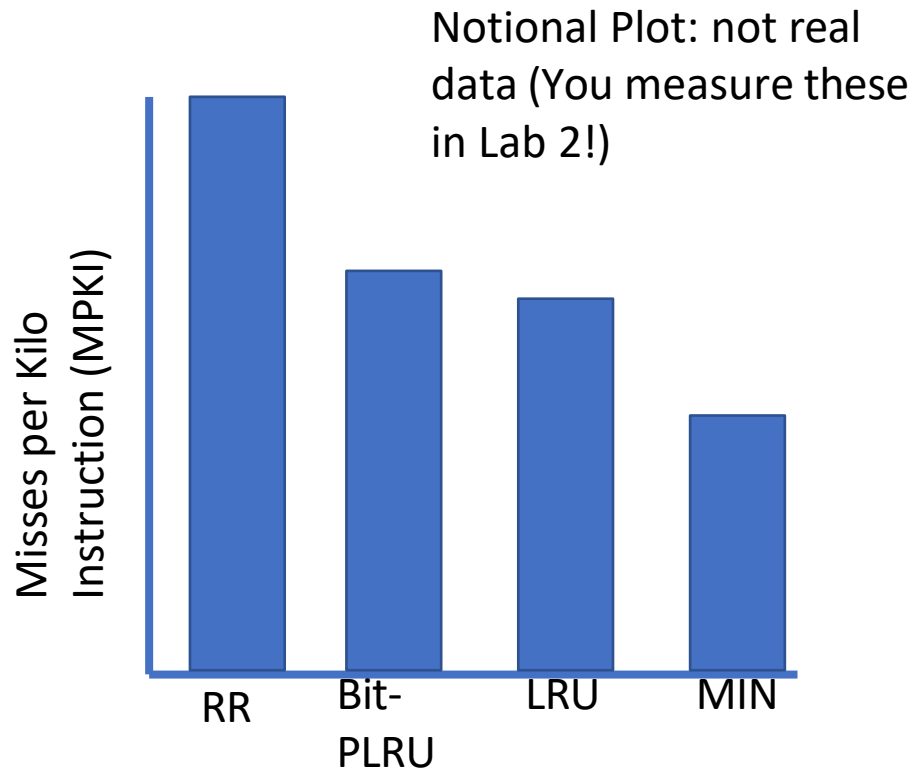
```
accessCachePLRU(access a){
    MRU_Bit[a.block] = 1
    if ++MRU_BitSum == setSize:
        for each block in cache, b:
            MRU_Bit[b] = 0
        MRU_BitSum = 0
}
```

```
findBlockLRU(){
    for i in 0..setSize:
        if !MRU_Bit[i]:
            return block(i);
}
```

**Implementability!** No future knowledge, 1 bit/block overhead, block-local metadata updates on access (no  $O(n)$  aging operation)

**Limitation!** Approximates LRU, which approximates MIN by guessing based on history...

# Replacement Policies – Performance & Complexity Cost/Benefit Analysis



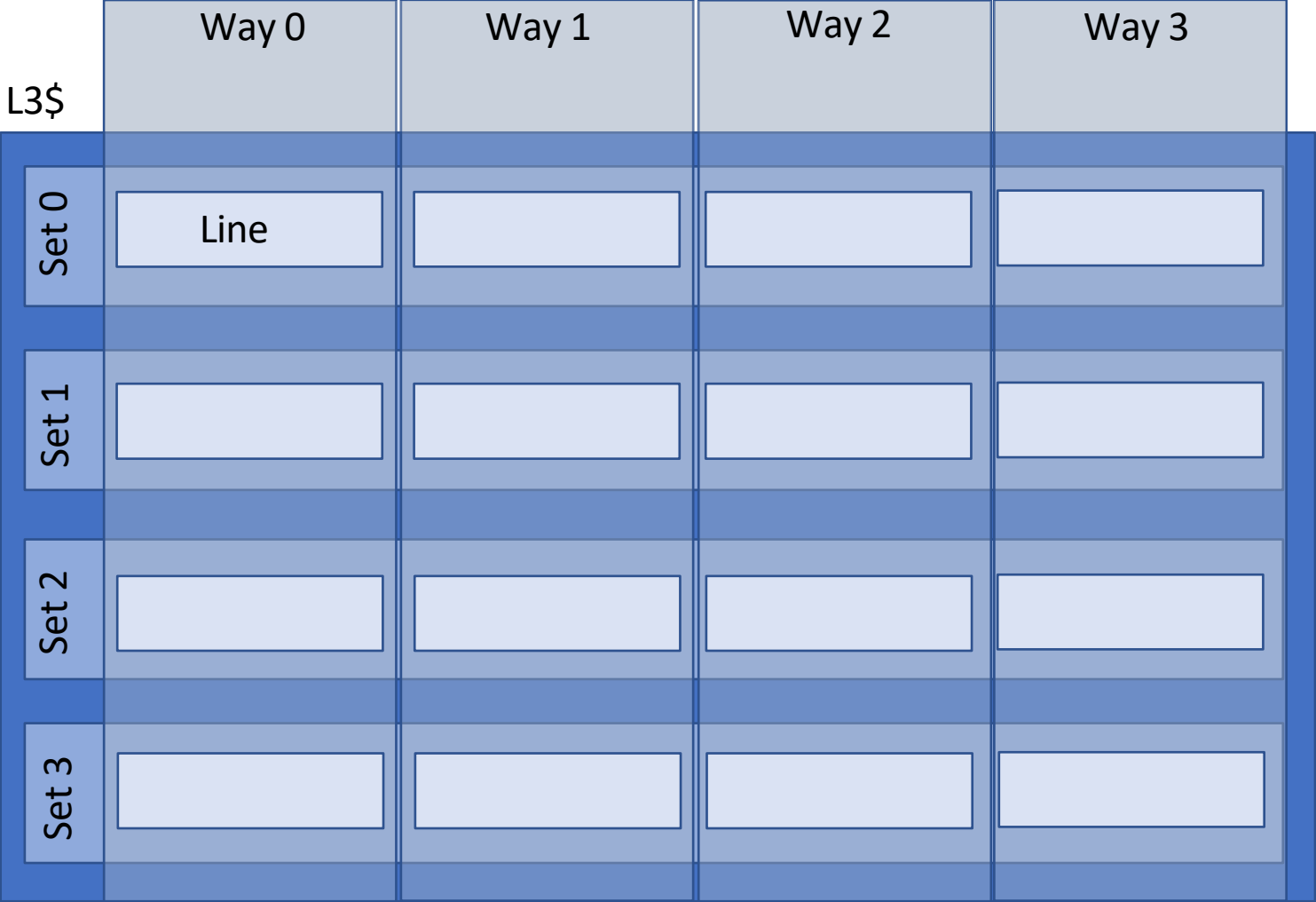
**RR:**  $\log(\text{set size})$  bits per set to track next to evict, no action on access

**Bit-PLRU:** 1 MRU bit per block +  $\log(\text{set size})$  bits per set (or equivalent logic) to detect all set,  
Clear bits on access if all bits set

**LRU:** 1 age per block + logic to track max.  
Update  $(\text{set size} - 1)$  ages on any access

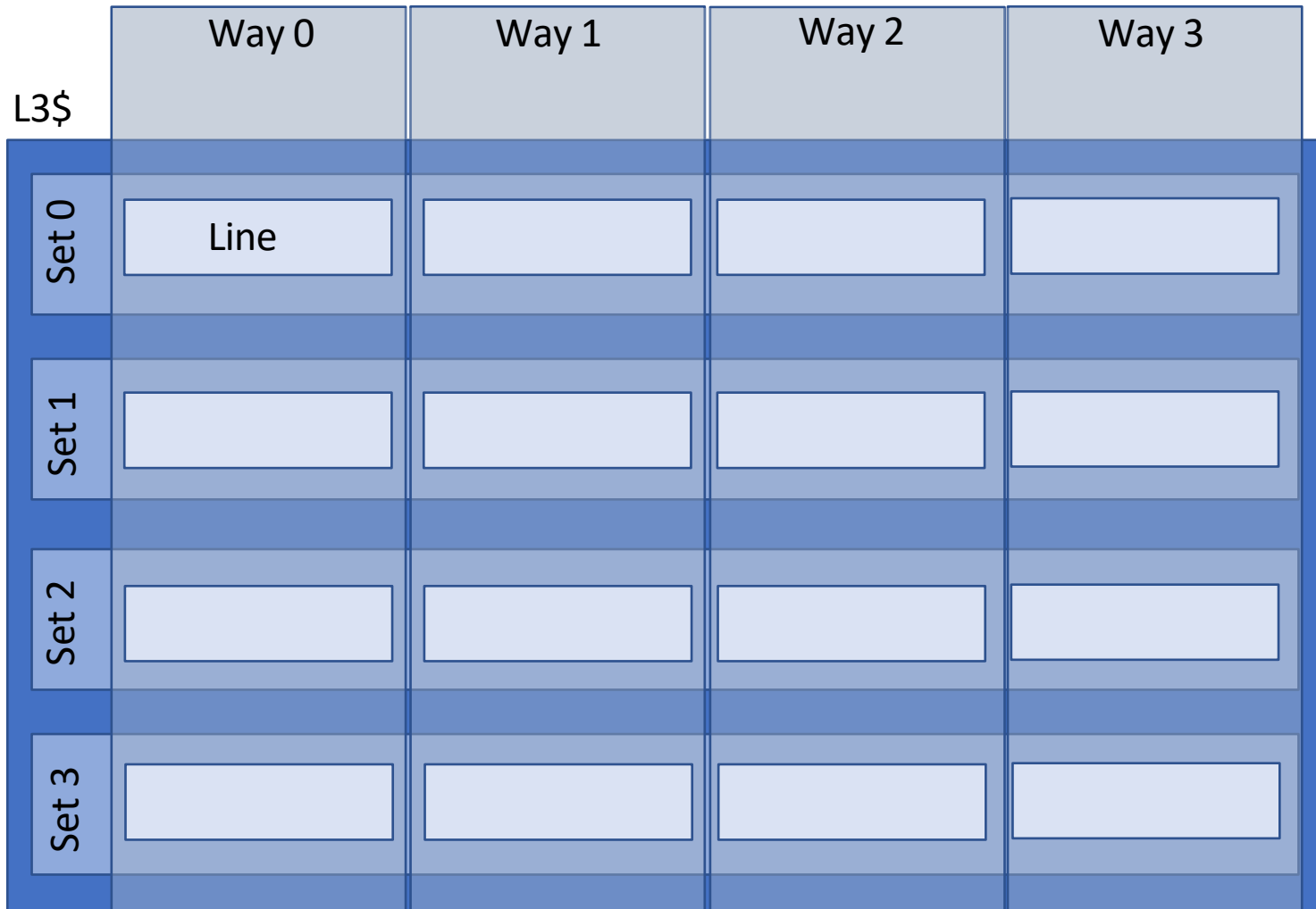
**MIN:** unimplementable, requires future knowledge of execution trace.

# More cache-related optimizations





# Recall a Set Associative Caches



What type of miss can be addressed by cache design?

- Cold?
- Capacity?
- Conflict?

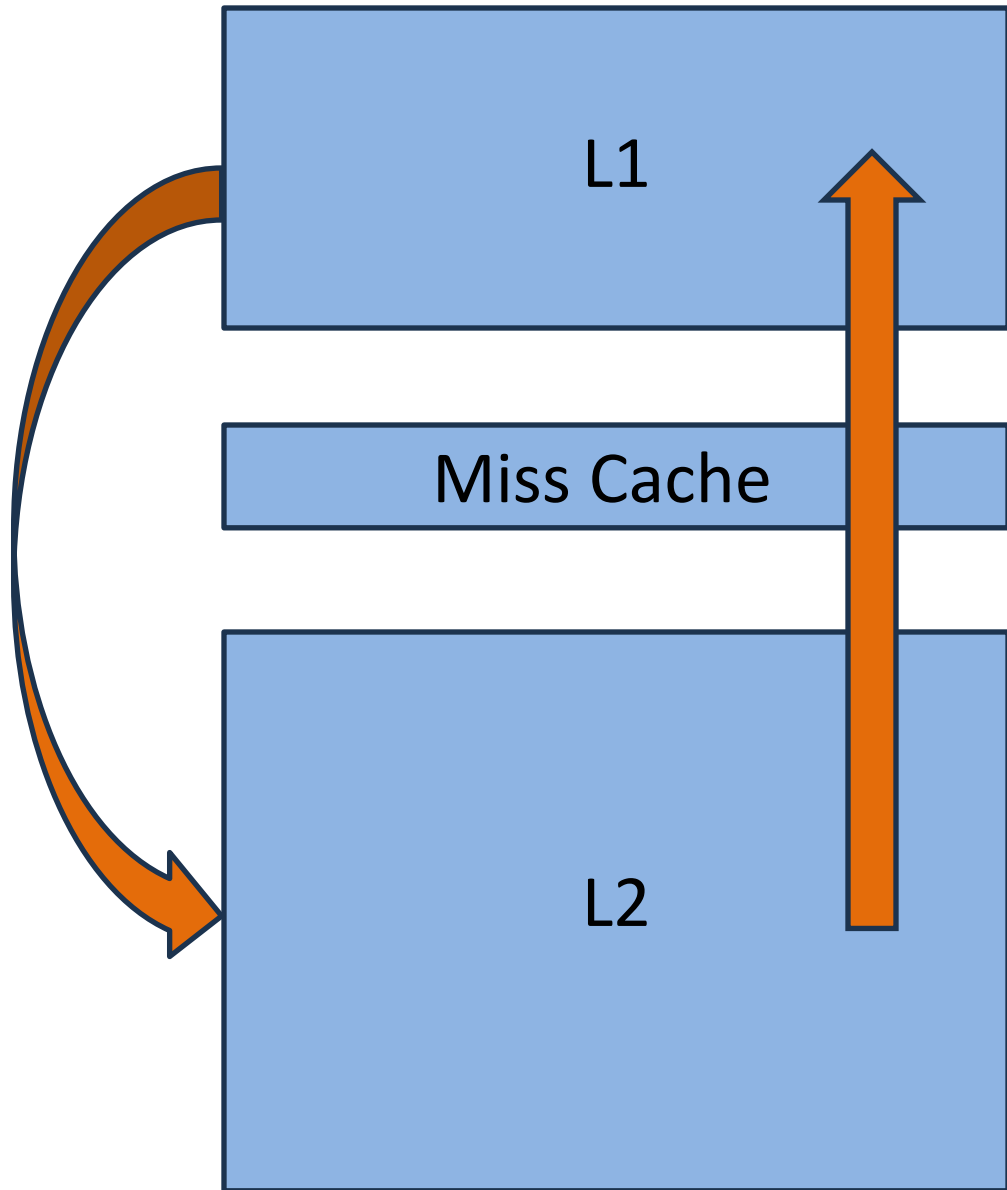
What can we do to address those misses without doing the impractical:

- Increasing cache size significantly(costly)
- Increasing associativity (slower)

Address the most glaring misses caused by partitioning, i.e. conflict misses

- But how?

# Miss Cache

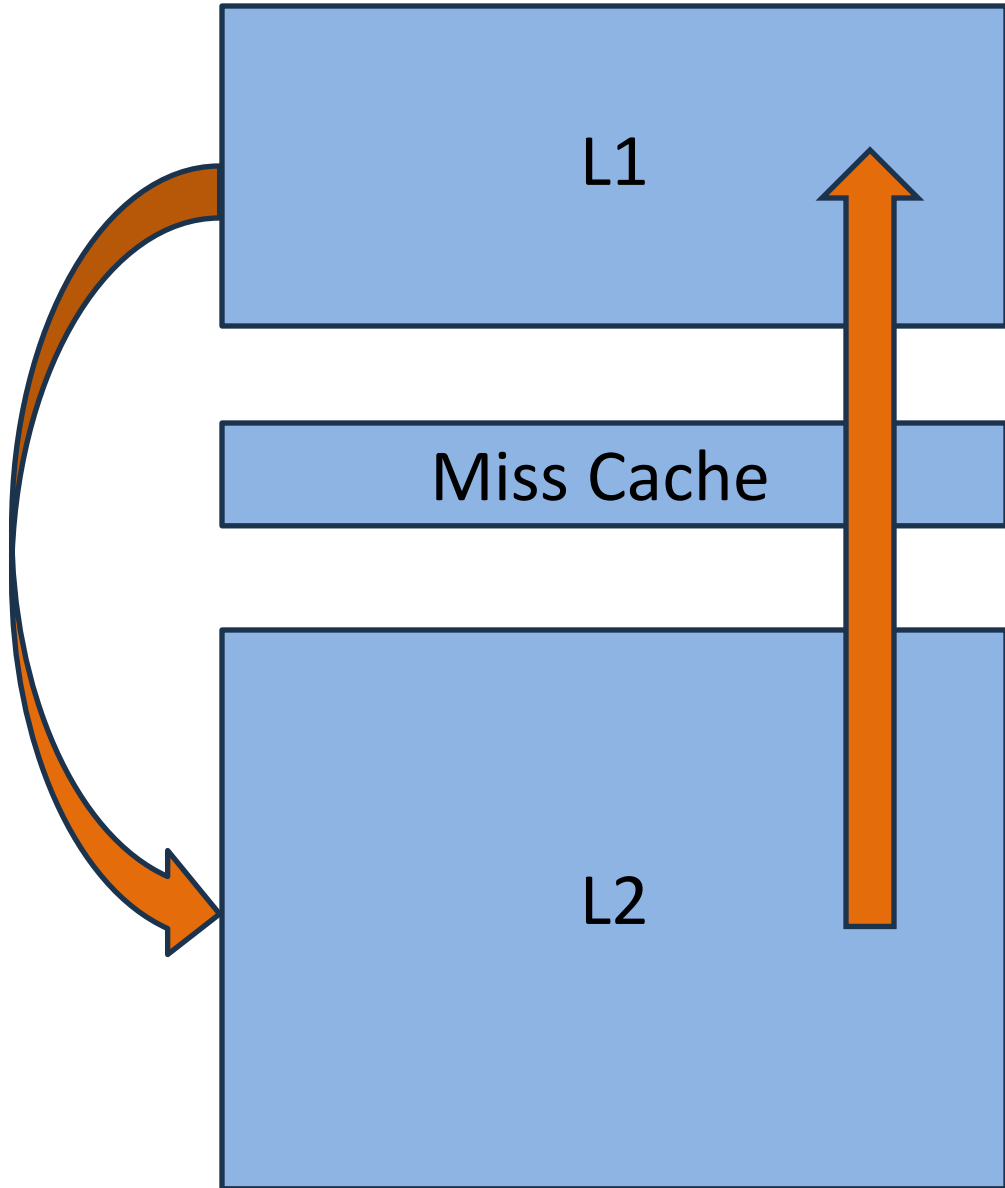


Set associative  
Probably write-through (Why?)

Just a few lines, i.e. 2 – 4, but  
fully associative

The most recent few reads brought  
into L1 from L2 also hang out for a  
very short time in the “Miss Cache”  
which is fully associative. L1 misses  
that is satisfied by the “Miss Cache”  
very low penalty.

# Miss Cache



But, most of the time, the miss cache stores values that are already stored in the L1 cache, as they were just read into both L1 and the miss cache from the L2 cache, which is a waste of space.

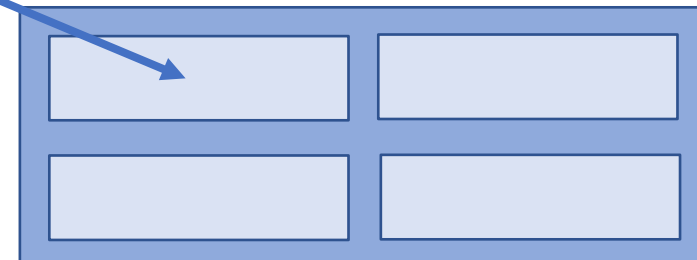
What can we do about that?

# Victim Caches/Buffers



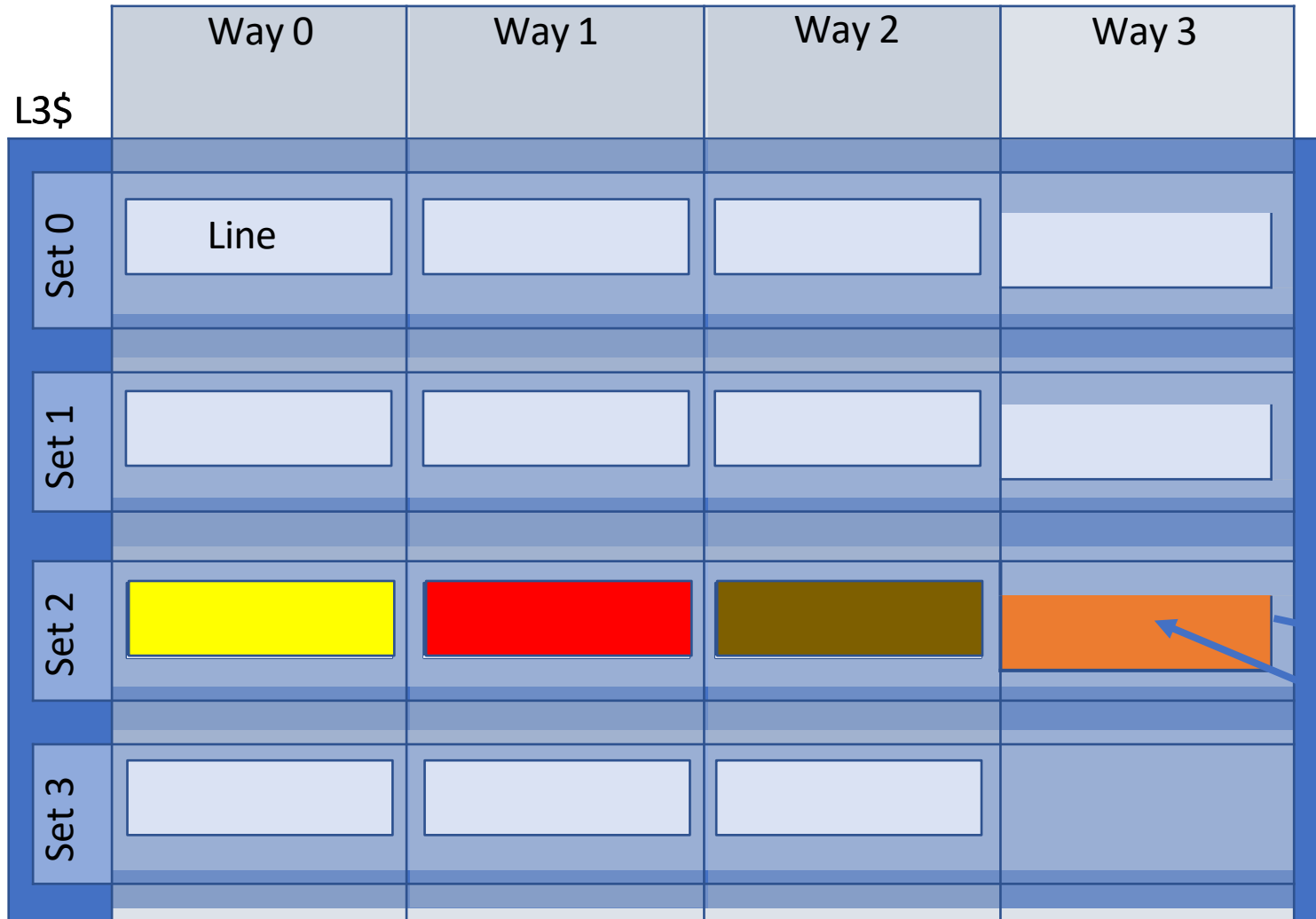
**Only store things into the cache upon eviction – then the only copy is in the small cache between L1 and L2, now called victim buffer.**

**Block evicted from cache goes into (usually fully associative, small) victim buffer.**



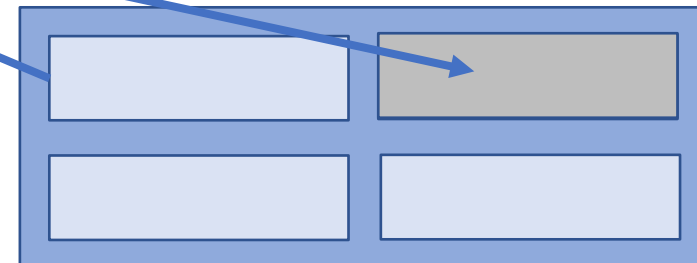
Victim Cache

# Victim Caches/Buffers



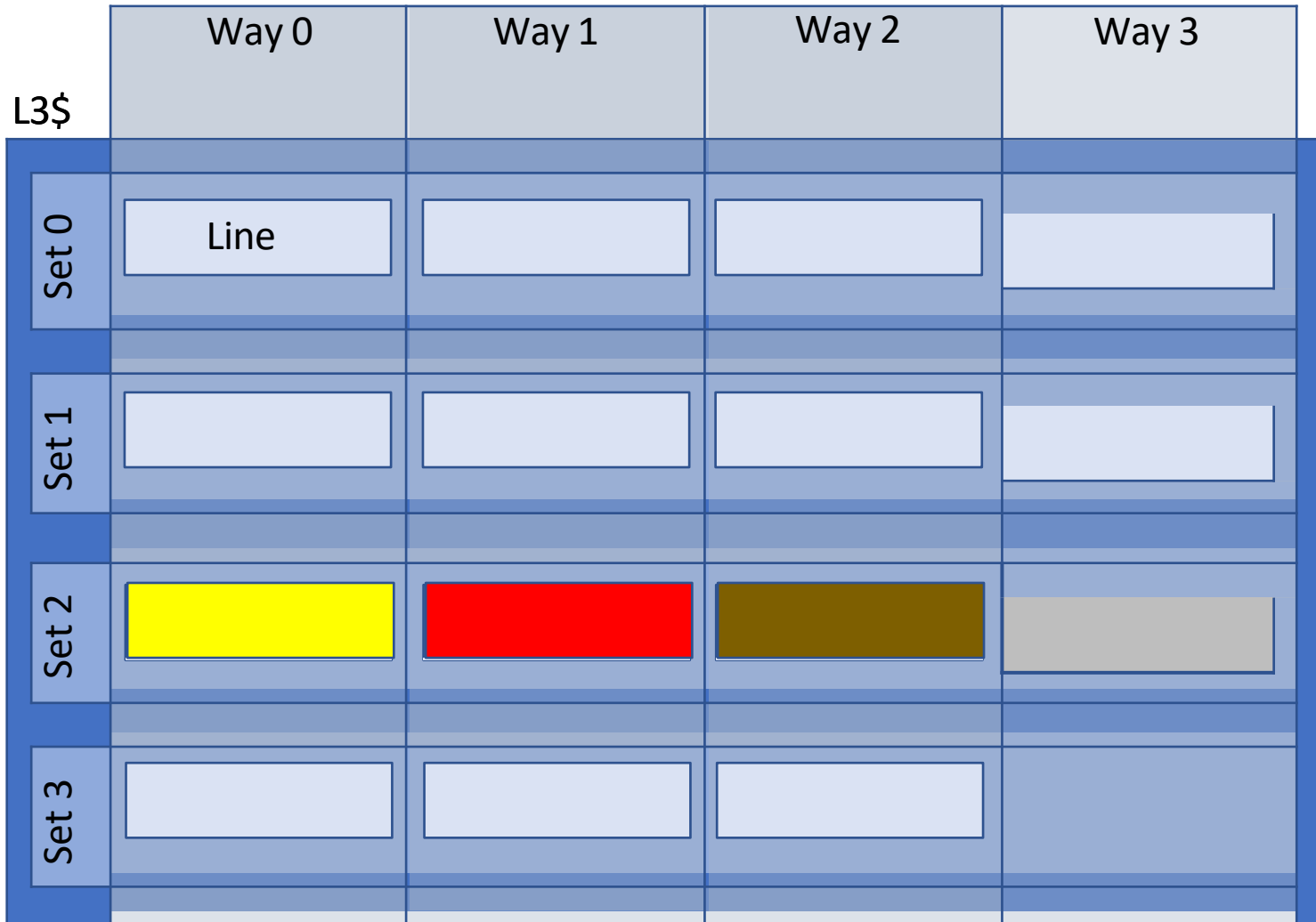
**Block evicted from cache goes into (usually fully associative, small) victim buffer.**

**On next access, "victim" can be re-cached without going down the hierarchy.**



Victim Cache

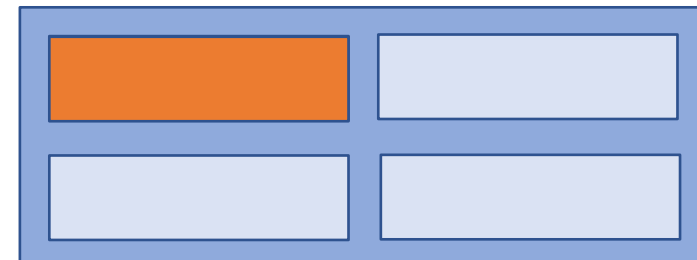
# Victim Caches/Buffers



**Block evicted from cache goes into (usually fully associative, small) victim buffer.**

**On next access, “victim” can be re-cached without going down the hierarchy.**

**What problem does a victim cache solve?**



Victim Cache

# Miss Caching vs Victim Caching

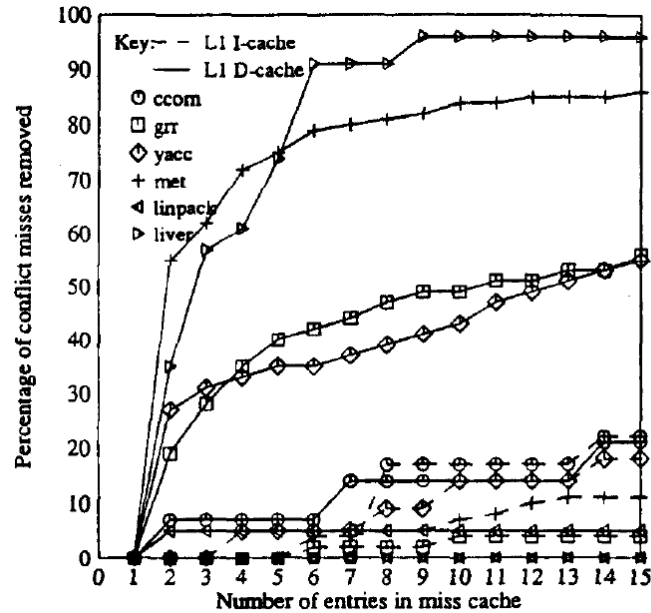


Figure 3-3: Conflict misses removed by miss caching

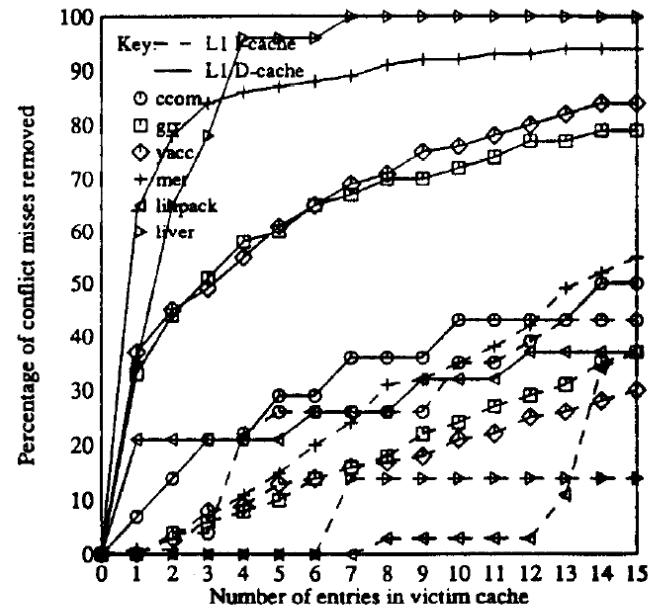
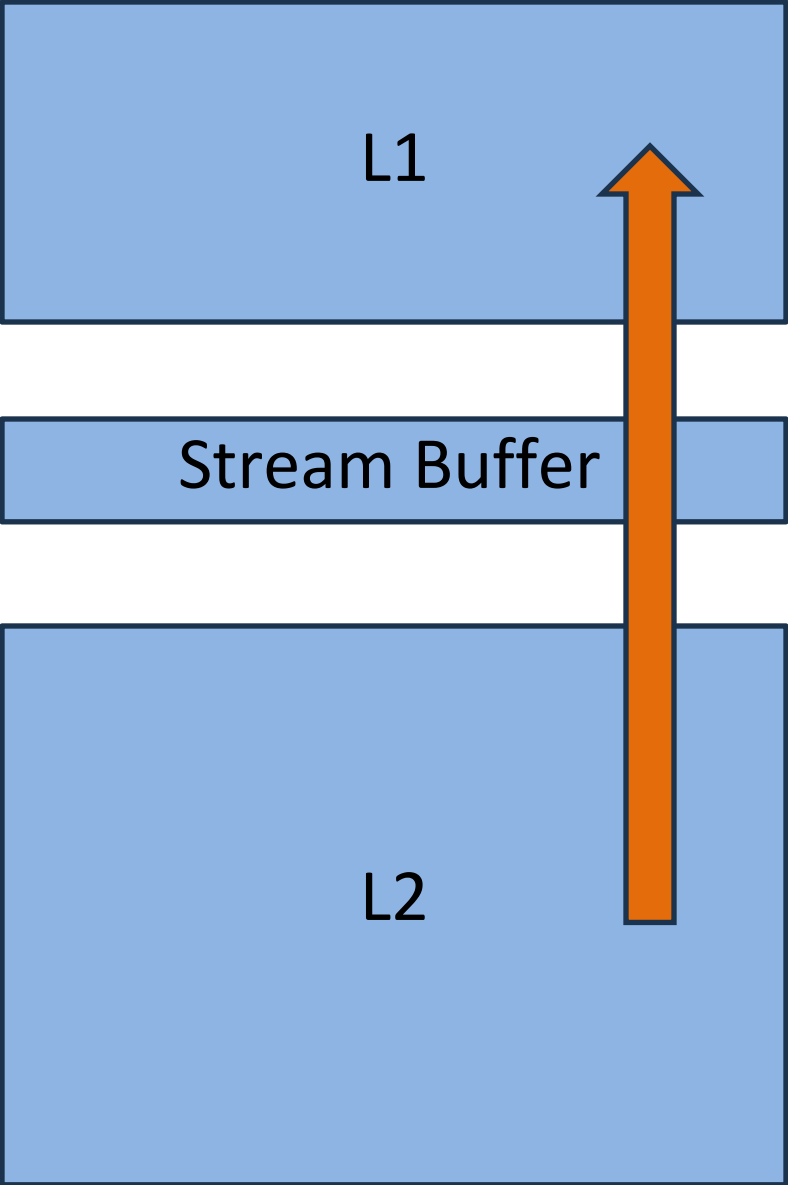


Figure 3-5: Conflict misses removed by victim caching

Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Computer Architecture News 18(3):388-397.

- Victim Caches better than Miss Caches
- But, better for d-Caches than i-Caches. Why? (Think about locality and distance)

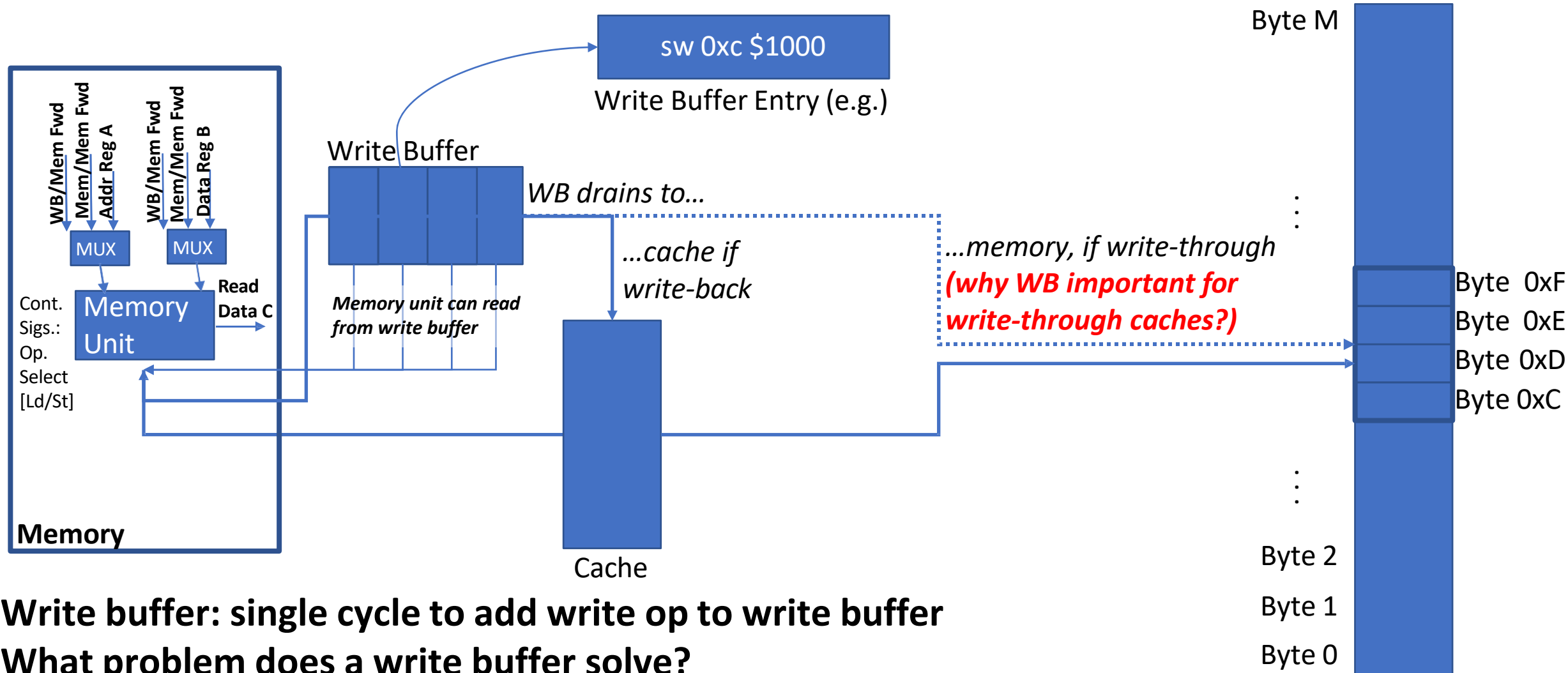
# Stream Buffer



Benefits i-caches  
Upon miss, prefetch next n instructions  
Gets back ahead after jumps



# Non-blocking Writes & Write Buffering

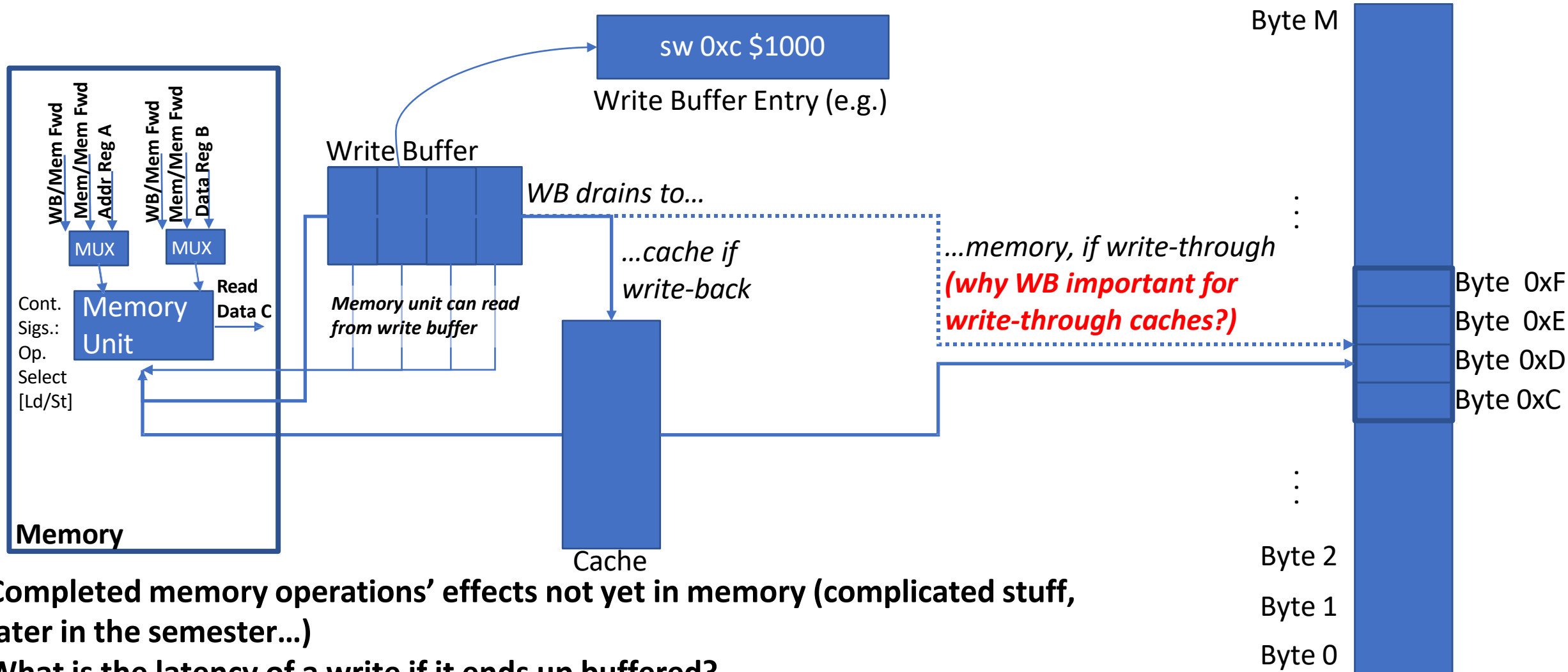


**Write buffer: single cycle to add write op to write buffer**

**What problem does a write buffer solve?**

**What are key challenges associated with a write buffer?**

# Non-blocking Writes & Write Buffering



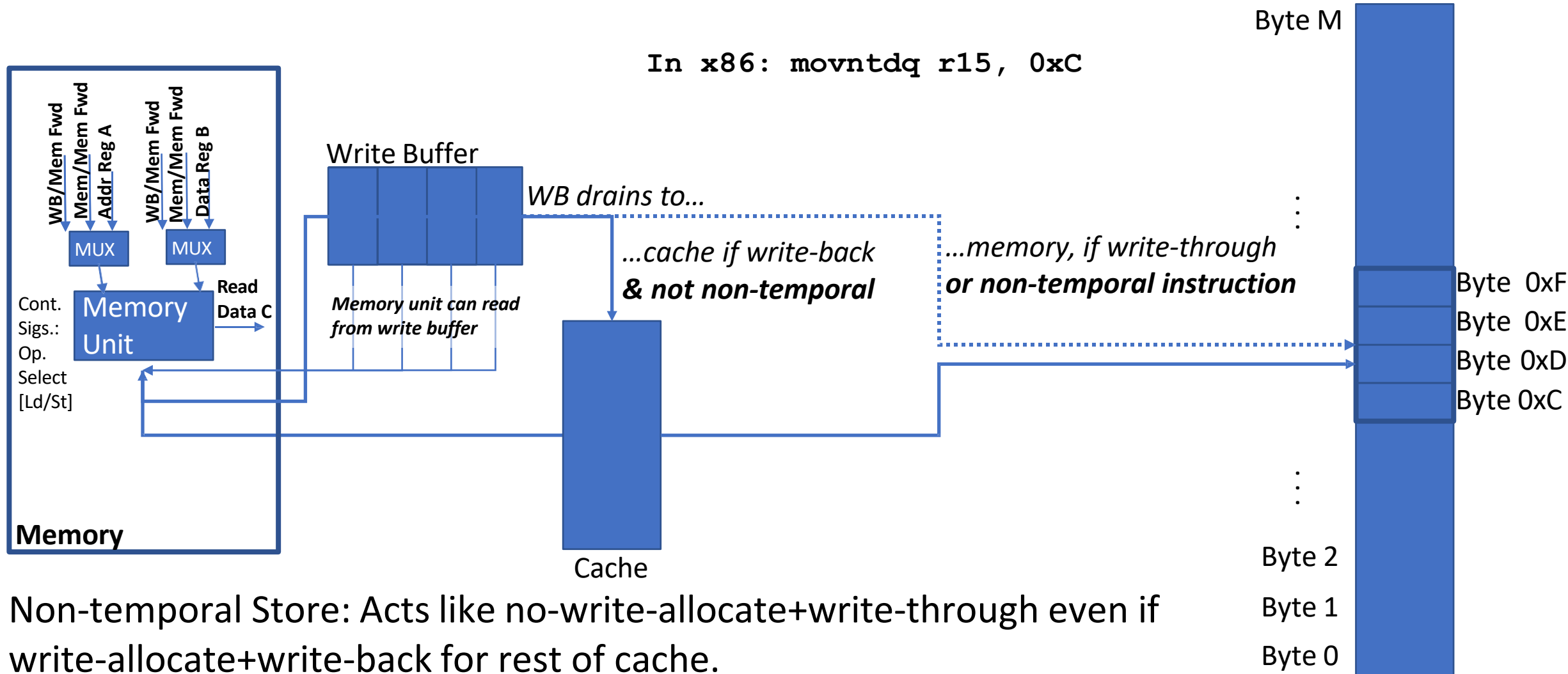
Completed memory operations' effects not yet in memory (complicated stuff, later in the semester...)

What is the latency of a write if it ends up buffered?

*Unpredictable write completion latency. Need **ordering** logic.*

# Non-temporal/Streaming Stores

In x86: `movntdq r15, 0xC`

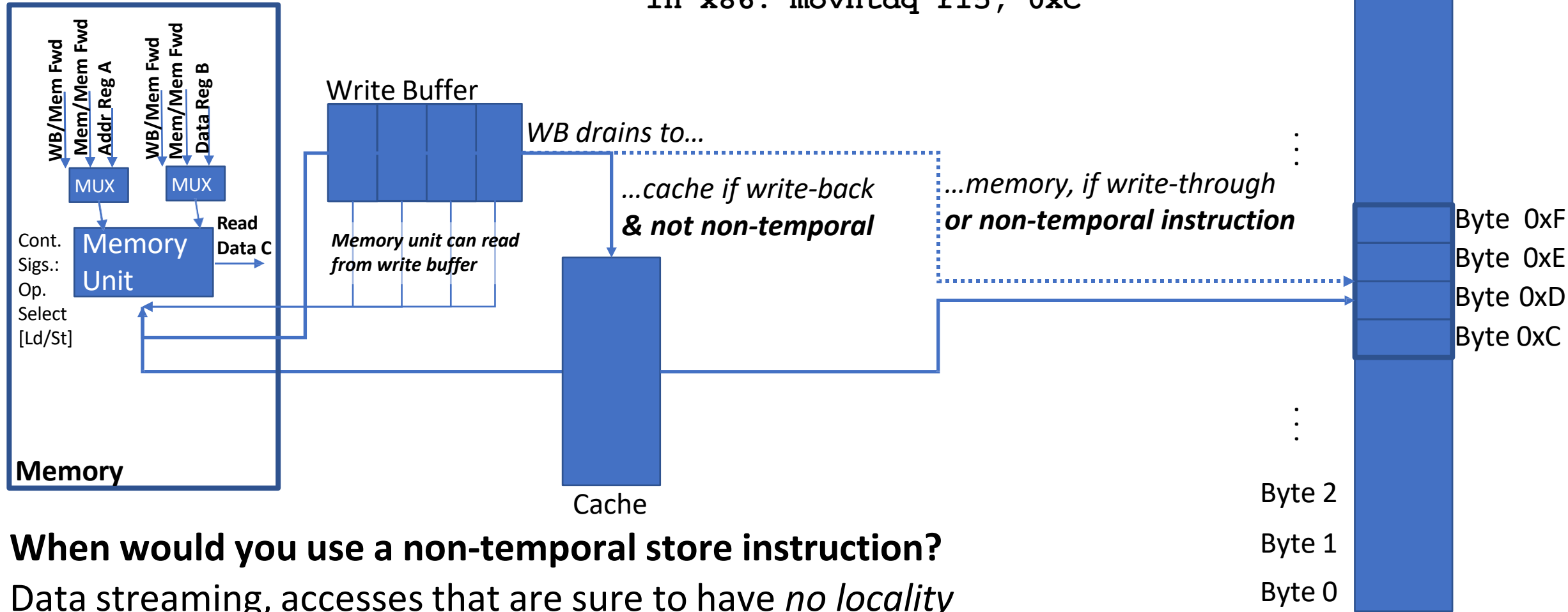


Non-temporal Store: Acts like no-write-allocate+write-through even if write-allocate+write-back for rest of cache.

**When would you use a non-temporal store instruction?**

# Non-temporal/Streaming Stores

In x86: `movntdq r15, 0xC`



# Not in RISC-V (yet)!

*RISC-V Specification:*

*“RV32I reserves a large encoding space for **HINT instructions, which are usually used to communicate performance hints to the microarchitecture.** HINTs are encoded as integer computational instructions with  $rd=x0$ . Hence, like the NOP instruction, **HINTs do not change any architecturally visible state**, except for advancing the pc and any applicable performance counters.*

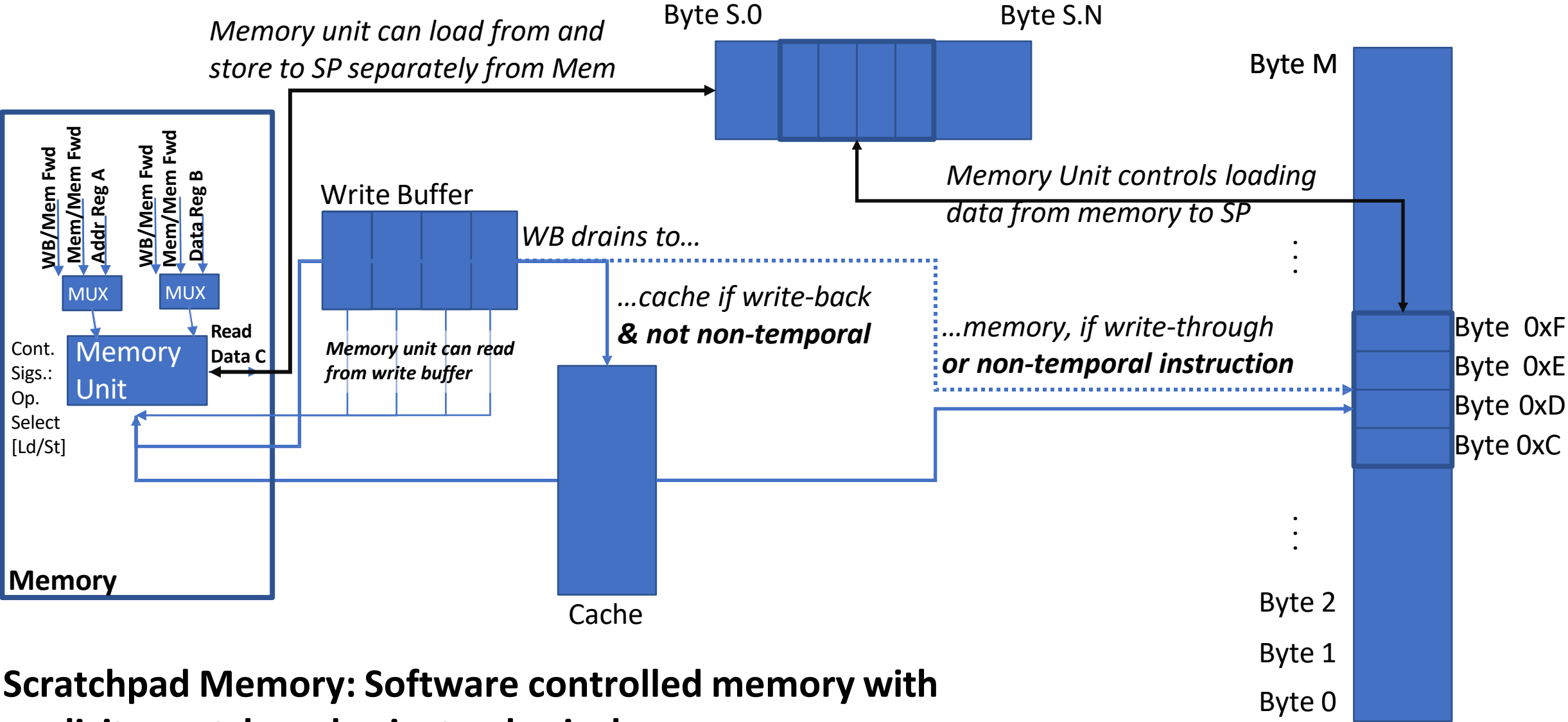
***Implementations are always allowed to ignore the encoded hints.”***

*No standard hints are presently defined (except the privileged WFI instruction which uses a separately reserved encoding). We anticipate standard hints to eventually include memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation.*

Instruction	Constraints	Code Points	Purpose
LUI	$rd=x0$	$2^{20}$	<i>Reserved for future standard use</i>
AUIPC	$rd=x0$	$2^{20}$	
ADDI	$rd=x0$ , and either $rs1 \neq x0$ or $imm \neq 0$	$2^{17} - 1$	
ANDI	$rd=x0$	$2^{17}$	
ORI	$rd=x0$	$2^{17}$	
XORI	$rd=x0$	$2^{17}$	
ADD	$rd=x0$	$2^{10}$	
SUB	$rd=x0$	$2^{10}$	
AND	$rd=x0$	$2^{10}$	
OR	$rd=x0$	$2^{10}$	
XOR	$rd=x0$	$2^{10}$	
SLL	$rd=x0$	$2^{10}$	
SRL	$rd=x0$	$2^{10}$	
SRA	$rd=x0$	$2^{10}$	
SLTI	$rd=x0$	$2^{17}$	<i>Reserved for custom use</i>
SLTIU	$rd=x0$	$2^{17}$	
SLLI	$rd=x0$	$2^{10}$	
SRLI	$rd=x0$	$2^{10}$	
SRAI	$rd=x0$	$2^{10}$	
SLT	$rd=x0$	$2^{10}$	
SLTU	$rd=x0$	$2^{10}$	

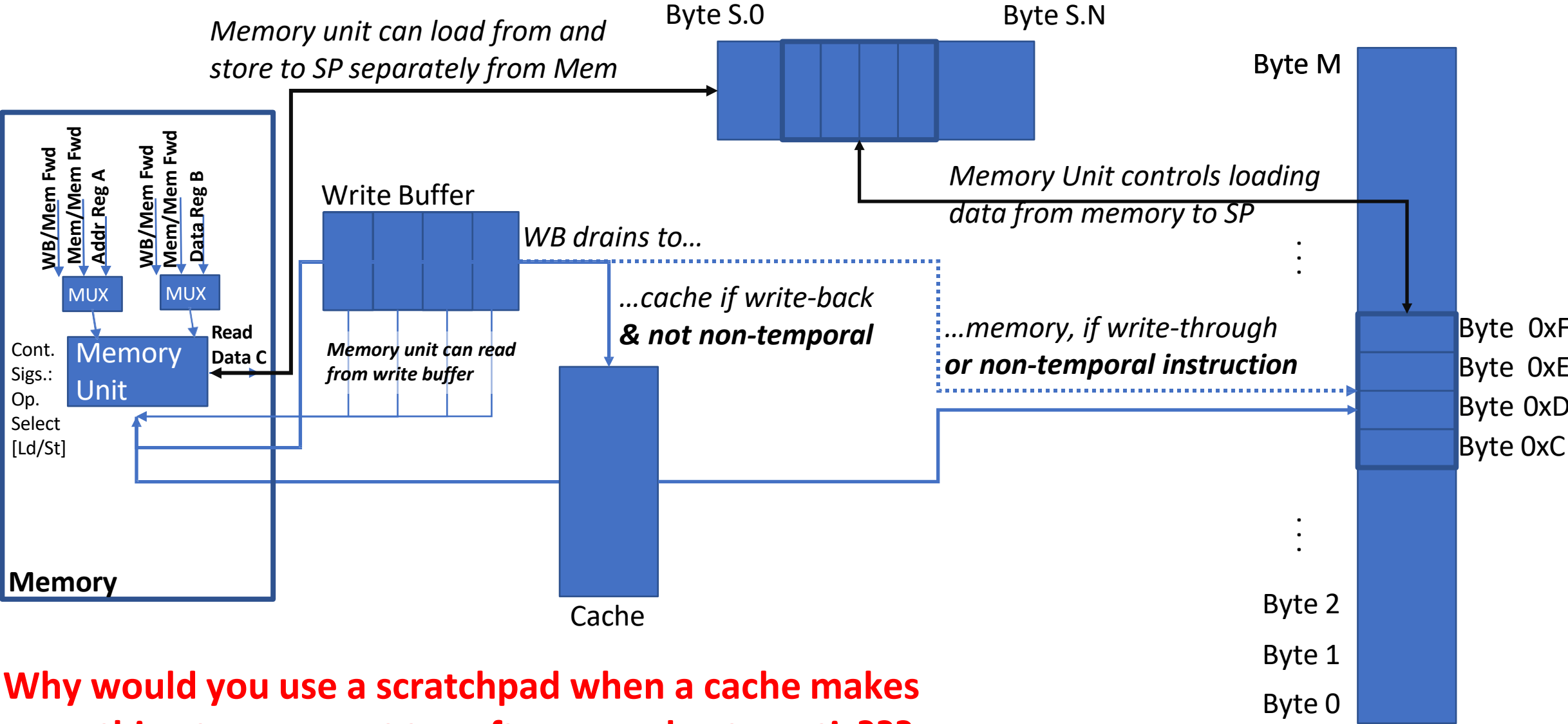
Table 2.3: RV32I HINT instructions.

# Scratchpad Memories



**Scratchpad Memory: Software controlled memory with explicit, scratch-pad-private physical memory space**

# Scratchpad Memories



**Why would you use a scratchpad when a cache makes everything transparent to software and automatic???**





# Review Questions to Ponder: The Memory Hierarchy and the Hardware Software Boundary

Part of the architecture: Cache? Scratchpad? Replacement policy?

Part of the HW/SW boundary: Cache? Scratchpad? Replacement policy?

What does a programmer need to know about how the cache works?

What does the architect need to know about how the machine will be used?

# What did we just learn?

- Replacement is a one of the key dimensions of cache design
- Different replacement algorithms present different design trade offs
- Optimal replacement is infeasible, practical replacement is non-optimal
- Many microarchitectural and architectural choices make up a memory hierarchy and the architect and programmer need to share information

# What to think about next?

- Performance Evaluation (next time)
  - Design spaces, Pareto Frontiers, and design space exploration
- Miscellaneous (micro)architectural tricks & optimizations (future)
  - Vector processors, SIMD/SIMT, dataflow

# What to think about next?

- Caches as a microarchitectural optimization (next time)
  - Implementation of cache hierarchies
  - Cache design tradeoffs
- Performance Evaluation (next next time)
  - Design spaces, Pareto Frontiers, and design space exploration
- Miscellaneous (micro)architectural tricks & optimizations (future)
  - Vector processors, SIMD/SIMT, dataflow