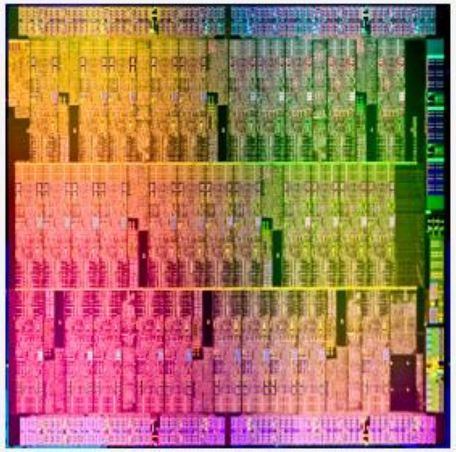
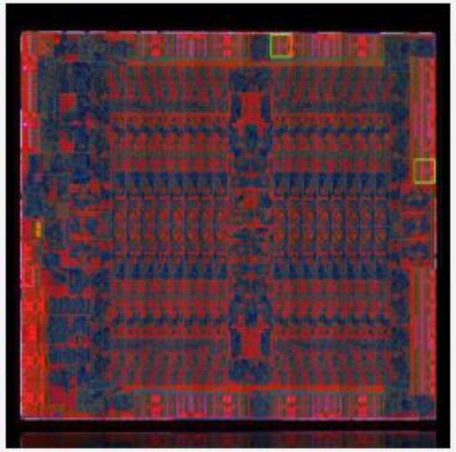
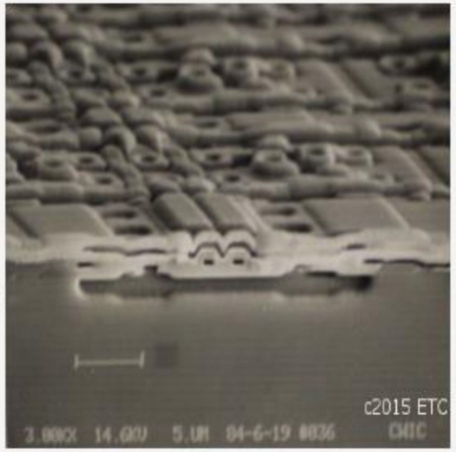
18-344: Computer Systems and the Hardware-Software Interface

Fall 2023



Course Description

Lecture 7: Caches and the Memory Hierarchy

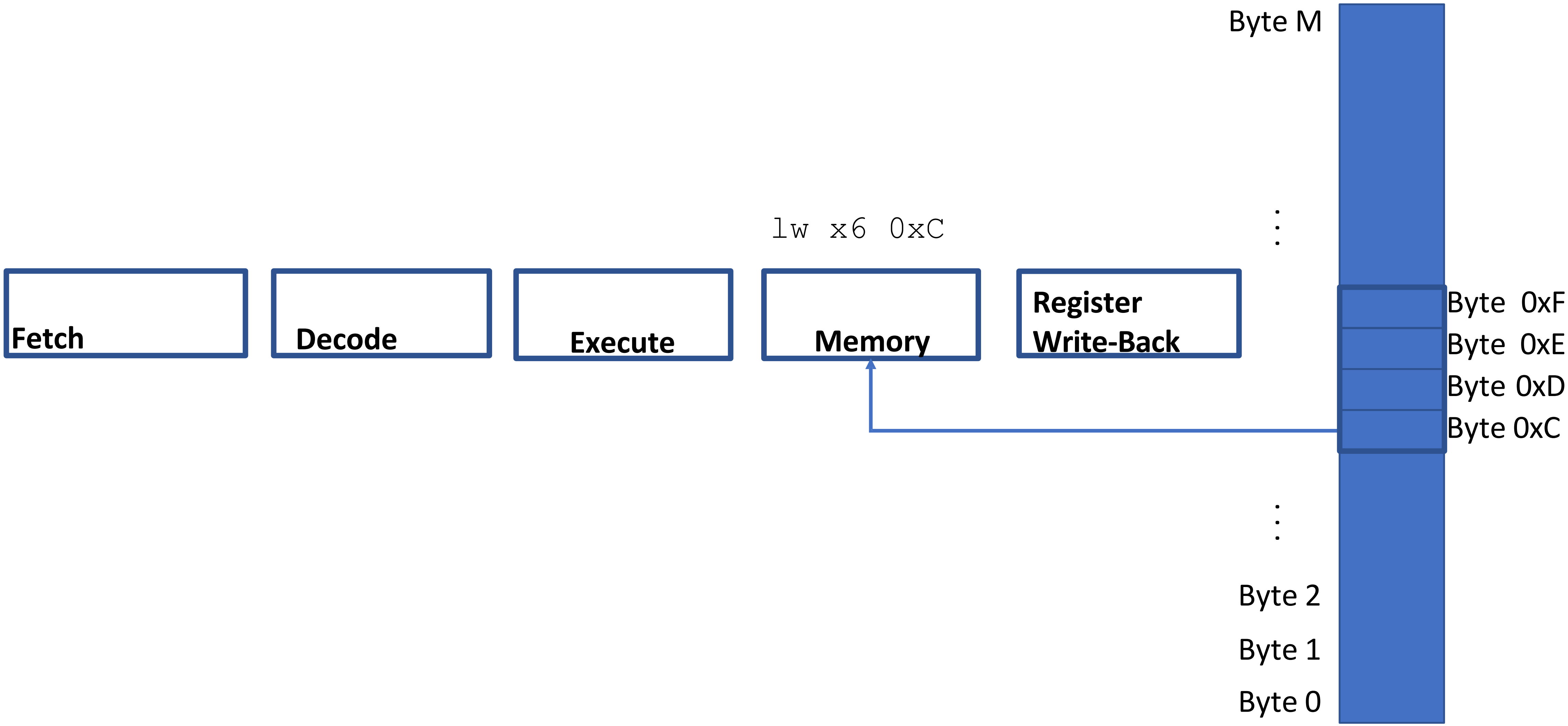
This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course 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

Today: Caches and the Memory Hierarchy

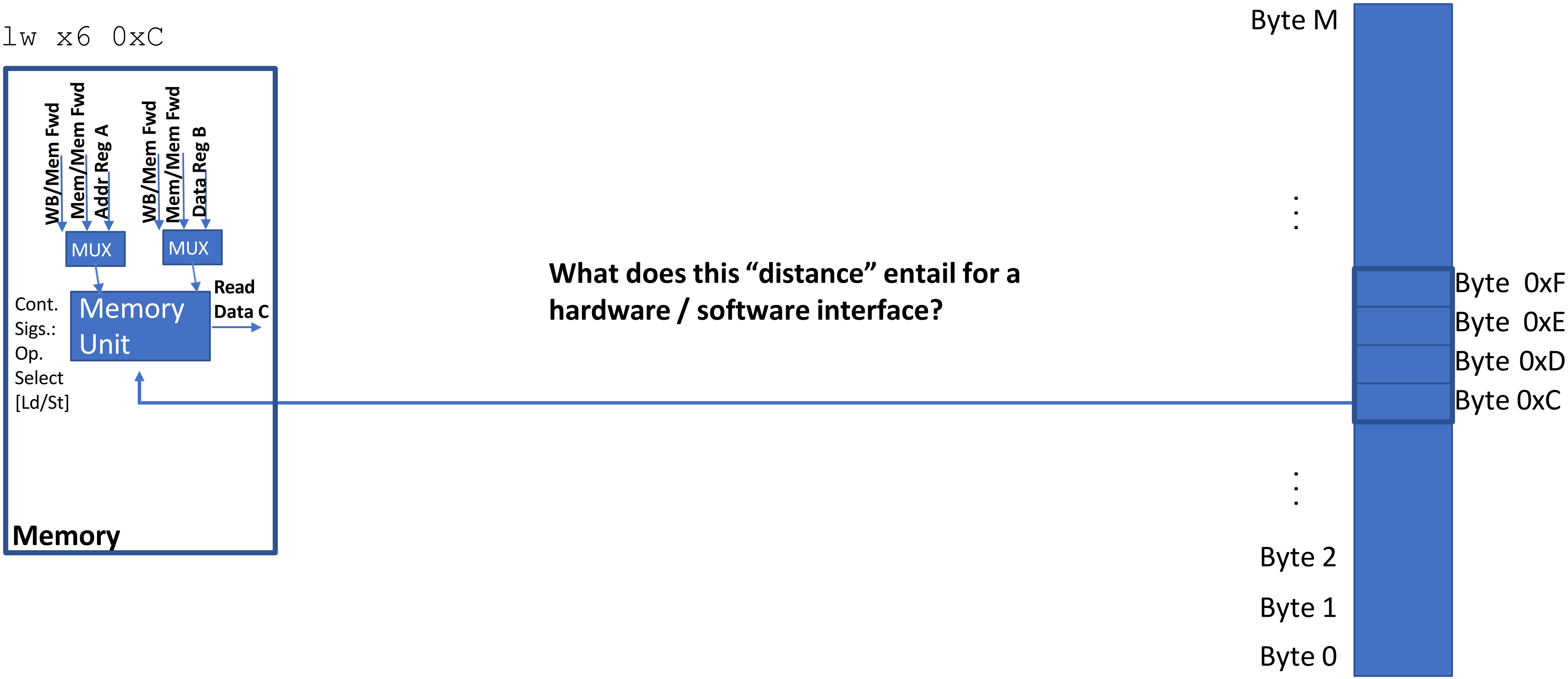
- Introduction to caches and cache organization
- Caches in the memory hierarchy
- Cache implementation choices
- Cache hardware optimizations
- Software-managed caches & scratchpad memories

Memory is a big list of M bytes



Memory is conceptually far away from CPU

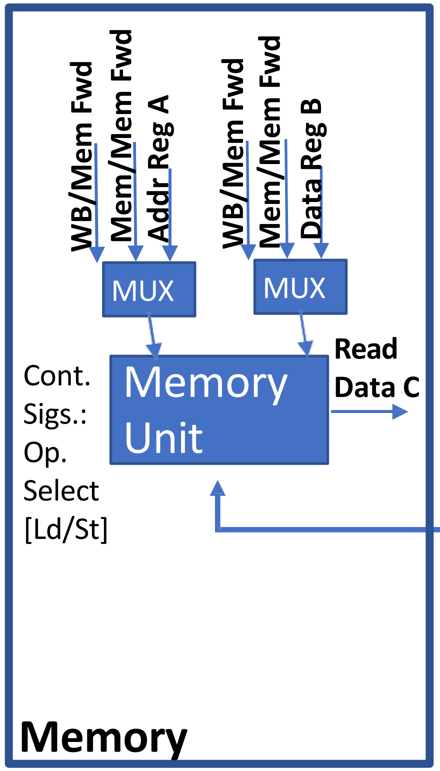
`lw x6 0xC`



What does this "distance" entail for a hardware / software interface?

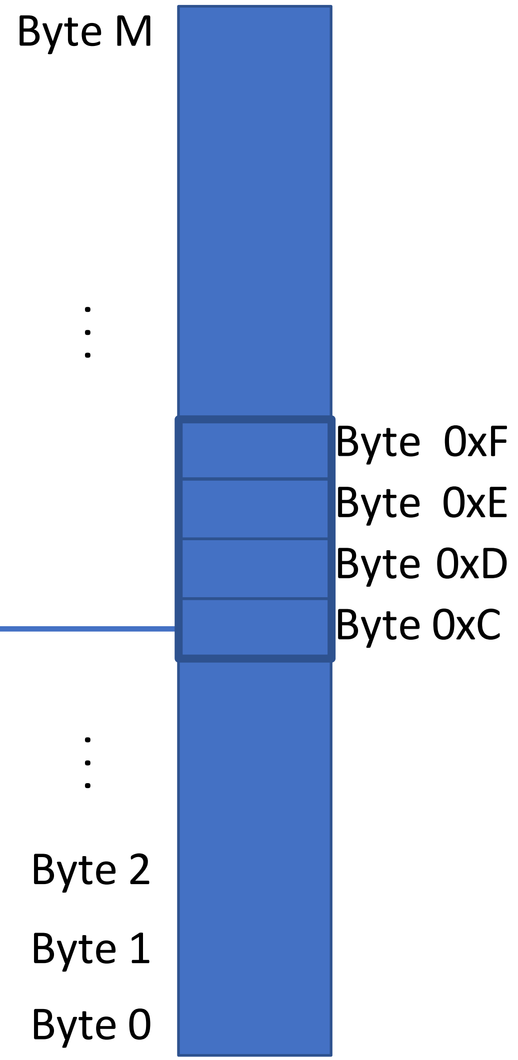
Memory is conceptually far away from CPU

`lw x6 0xC`

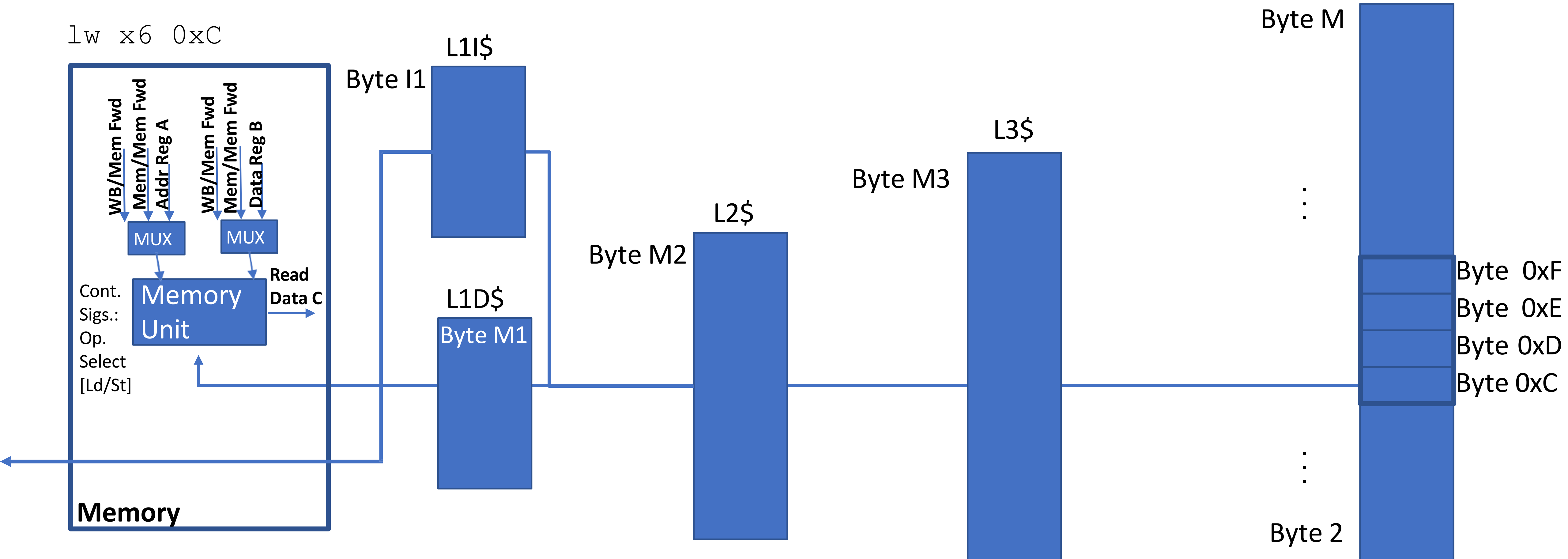


What does this “distance” entail for a hardware / software interface?

- Need to be judicious with **`lw & sw`**
- Compiler & programmer must carefully lay out memory
- Worth spending hardware resources to optimize
- Need hardware and software to co-optimize **data re-use**
- **Data movement is a fundamental limit on speed & energy**



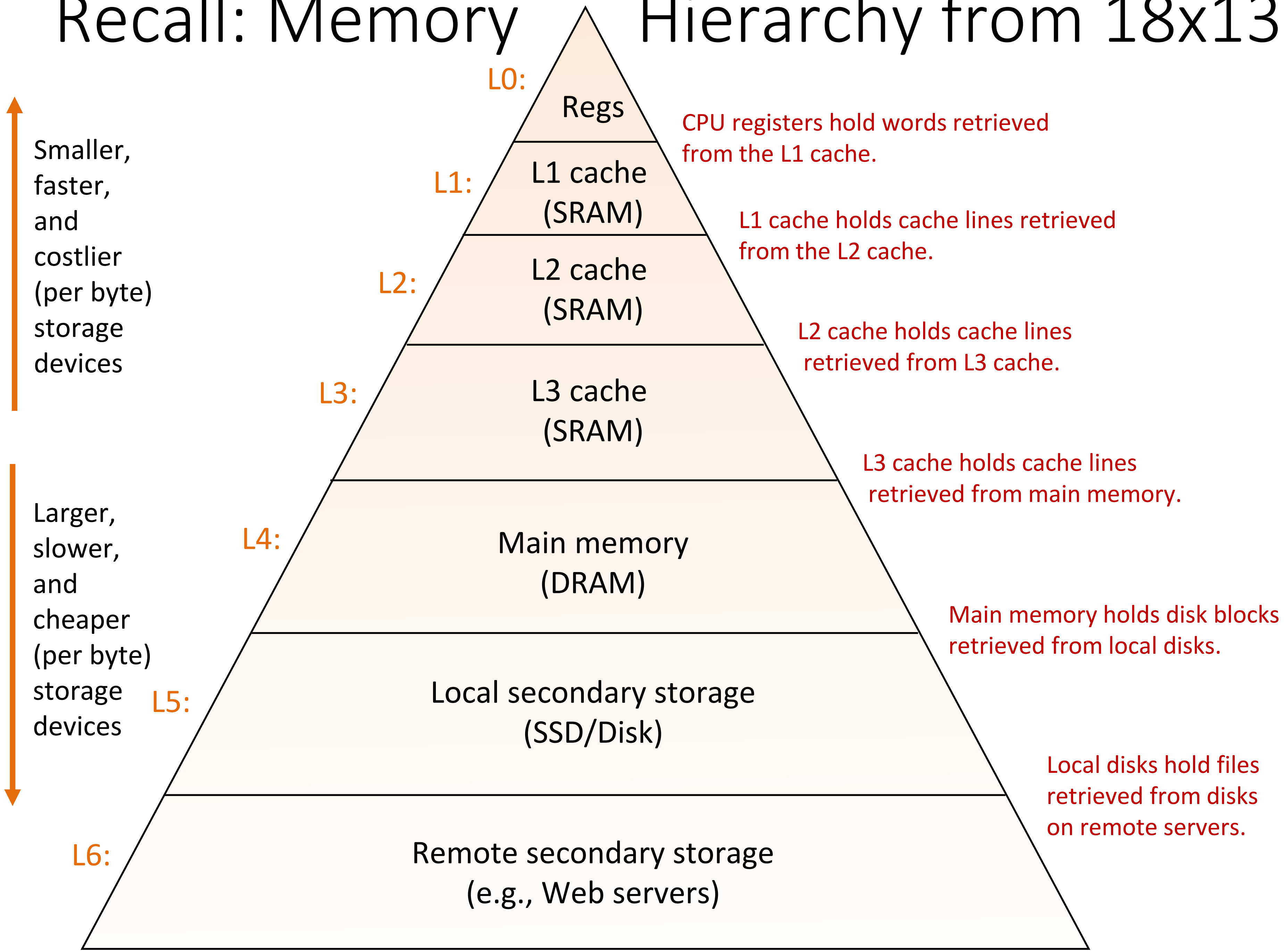
Memory hierarchy: large & slow vs. small & fast



Capacity inversely proportional to access cost

M > M3 > M2 > M1

Recall: Memory Hierarchy from 18x13



Recall from 18x13: The Working Set

- The data that is presently being use is called the *Working Set*.
- Imagine you are working on 18x13. Your working set might include:
  - The lab handout
  - A terminal window for editing
  - A terminal window for debugging
  - A browser window for looking up man pages
- If you changed tasks, you'd probably hide those windows and open new ones
- The data computer programs use works the same way.

Recall from 18x13: Guesstimating the Working Set

- How does the memory system (cache logic) know the working set?
  - This is tricky. There is no way it can really know what data the program needs or will need soon.
  - It could even be totally dynamic, based upon input.

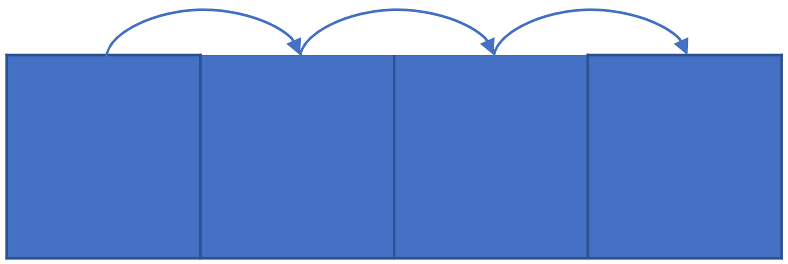
- It approximates it using a simple heuristic called *locality*:
  - *Temporal locality*: Data used recently is likely to be used again in the near future (local in time).
  - Spatial locality: Data near the data used recently is likely to be used soon (local in space, e.g. address space).

- The memory system will bring and keep the *Most Recently Used (MRU)* data and data near it in memory to the higher layers while evicting the *Least Recently Used (LRU)* data to the lower layers.

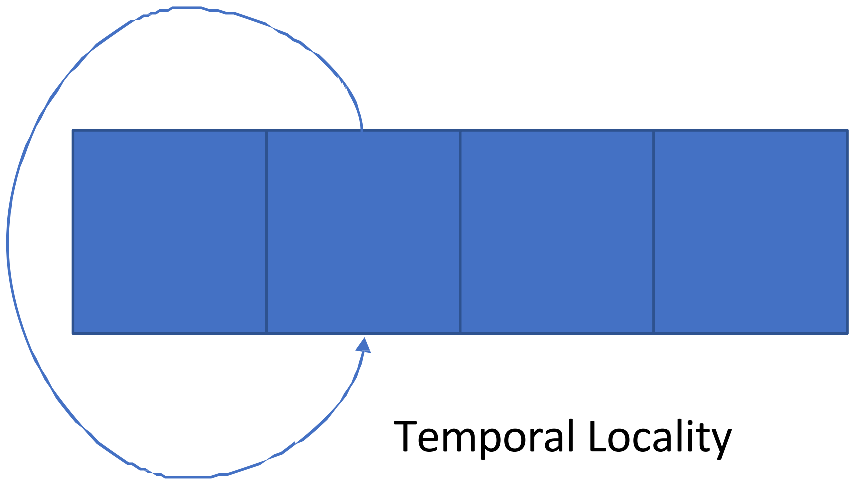
What's New Since 18x13?

- We want to think about a cache built natively in real hardware vs a software simulation of a cache
- The 18x13 cache was a software simulation of a somewhat ideal LRU cache
- Consider how you built an LRU cache simulator in 18x13:
  - A linked list- based queue?
  - A copy-to-shift array-based queue?
- Time for the "18-240 Thinking Cap": Consider the implementation of LRU in hardware
  - Can the 18x13 approach be translated to real hardware in a practical way?

Locality is the key to cache performance



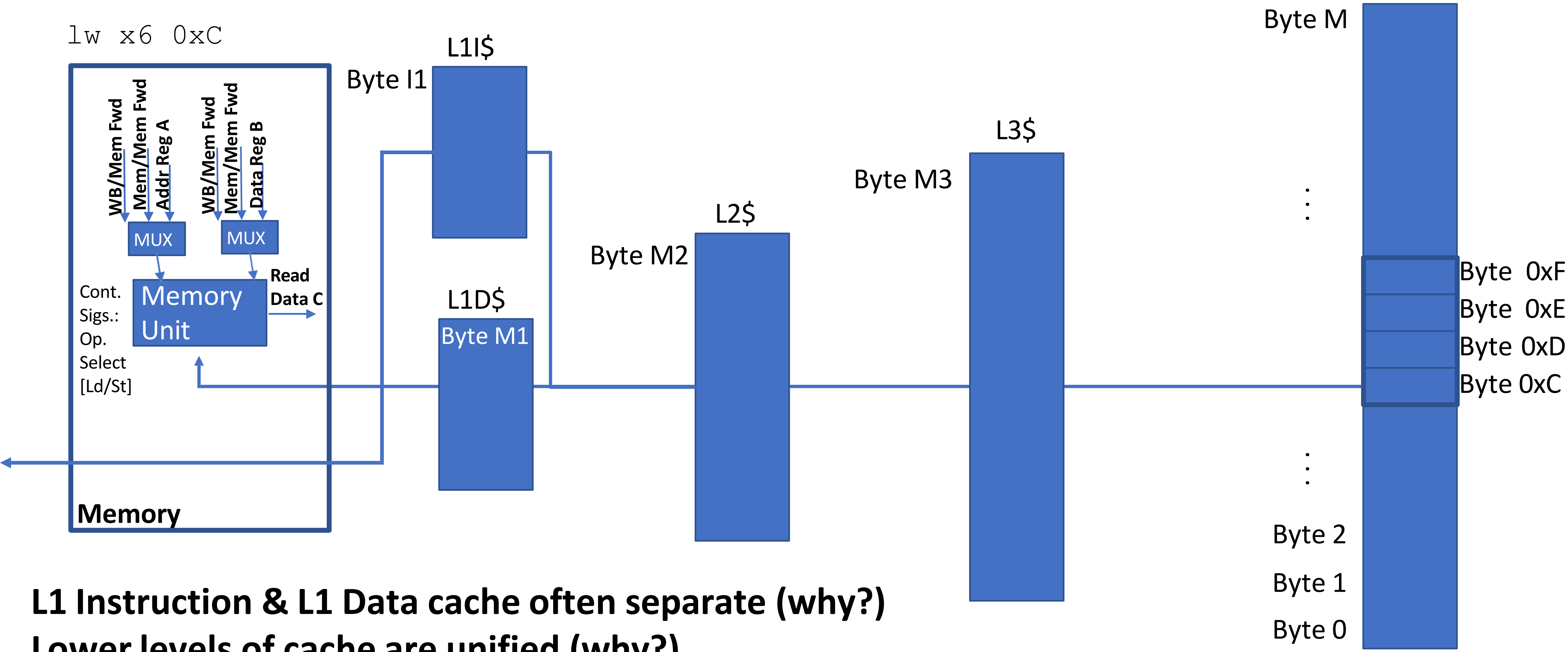
Spatial Locality



Temporal Locality

Why do we see locality? What are some examples of each?

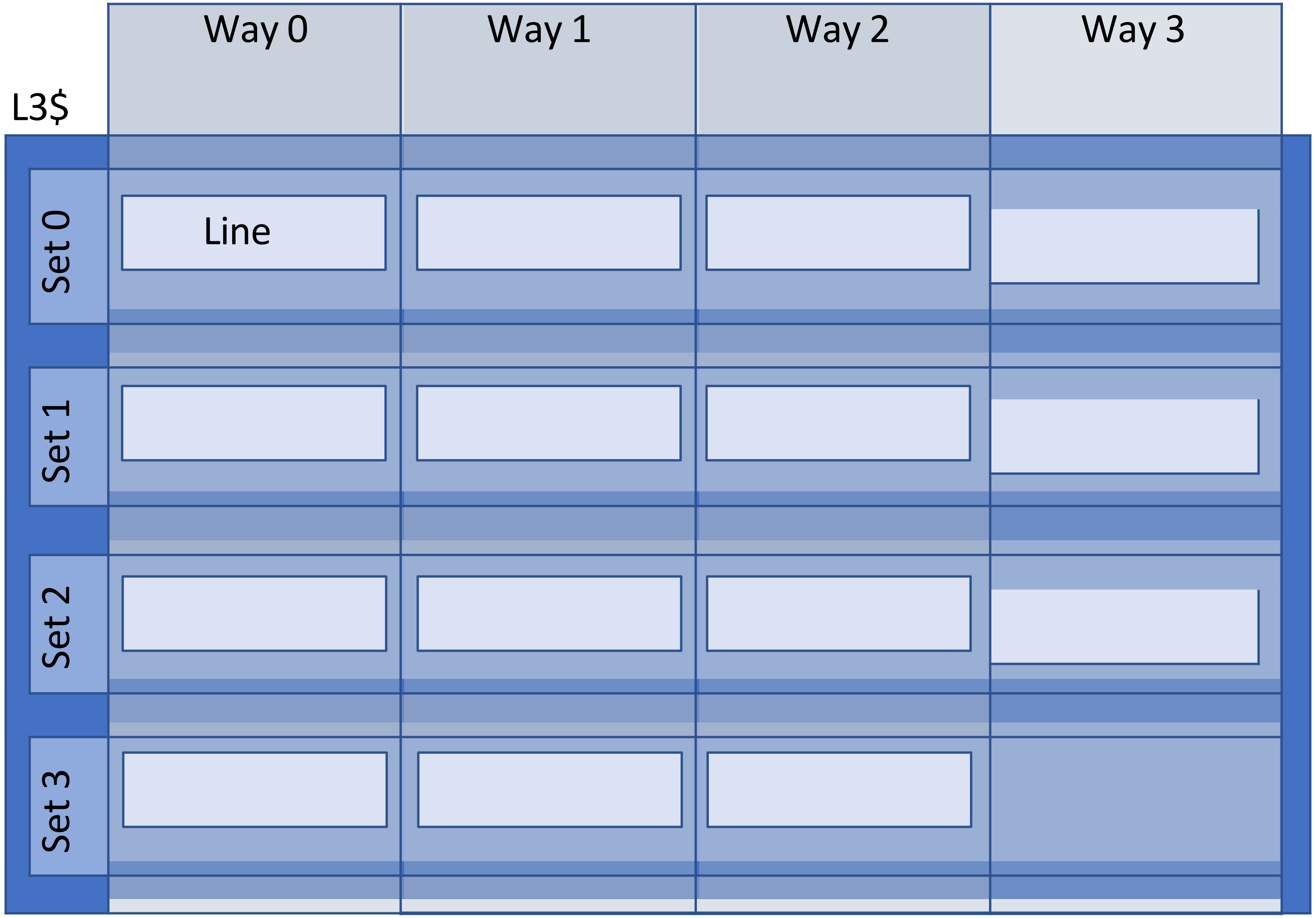
Memory hierarchy: Unified vs. Split ICache & DCache



L1 Instruction & L1 Data cache often separate (why?)

Lower levels of cache are unified (why?)

Review: Anatomy of a set-associative cache

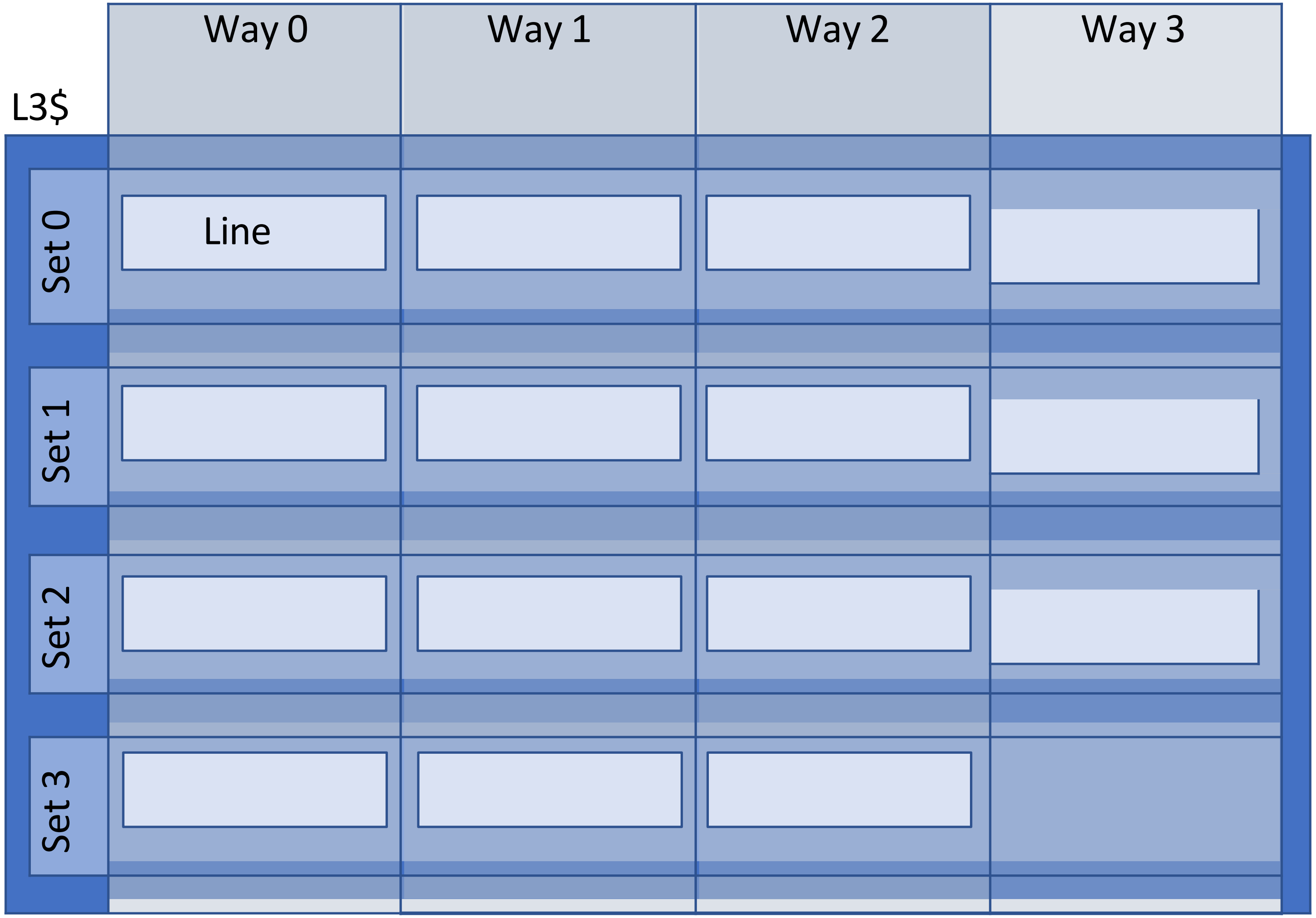


Typical Parameters
Line contains 16-64 bytes of data
1-8 number of sets
1 set contains all lines?
All sets contain 1 line?
Total size varies by level:
L1: 1kB – 32kB
L3: a few kB – 48MB

Valid	Dirty	Tag	B bytes data

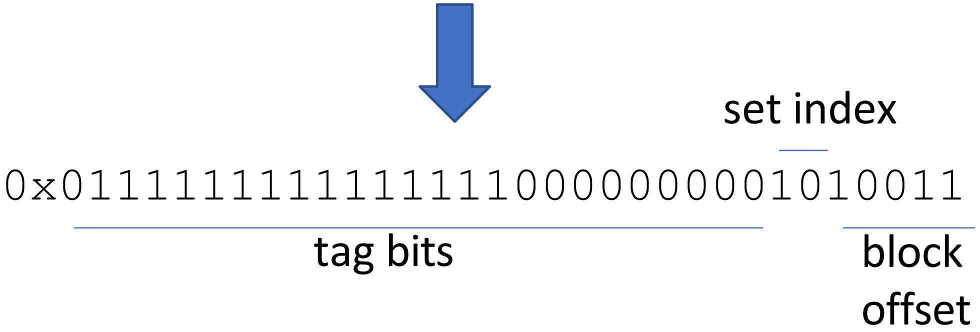
Anatomy of a Line

Review: Accessing the cache



Step 1: Partitioning the address

`lb x6 0x7fff0053`

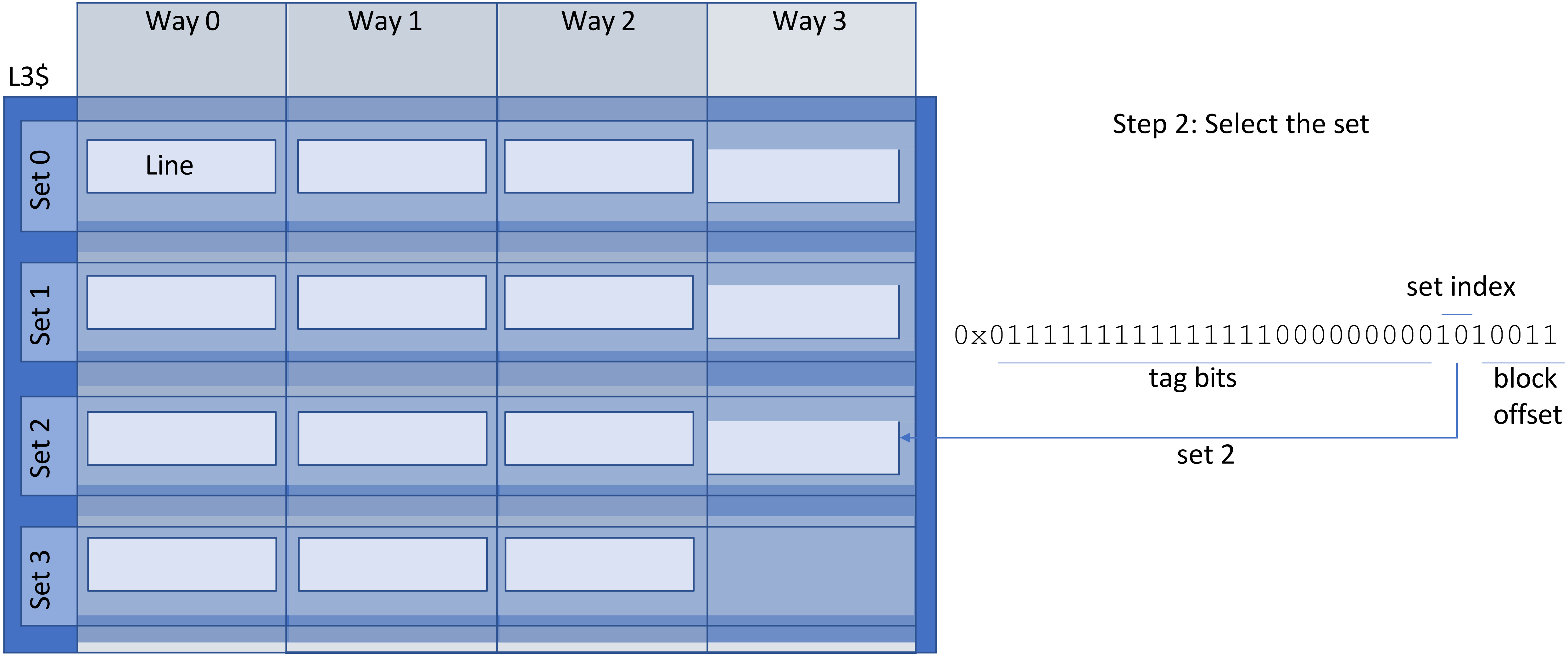


Valid	Dirty	Tag	32 bytes data

Total cache size = 32B x 4 sets x 4 ways = 512B

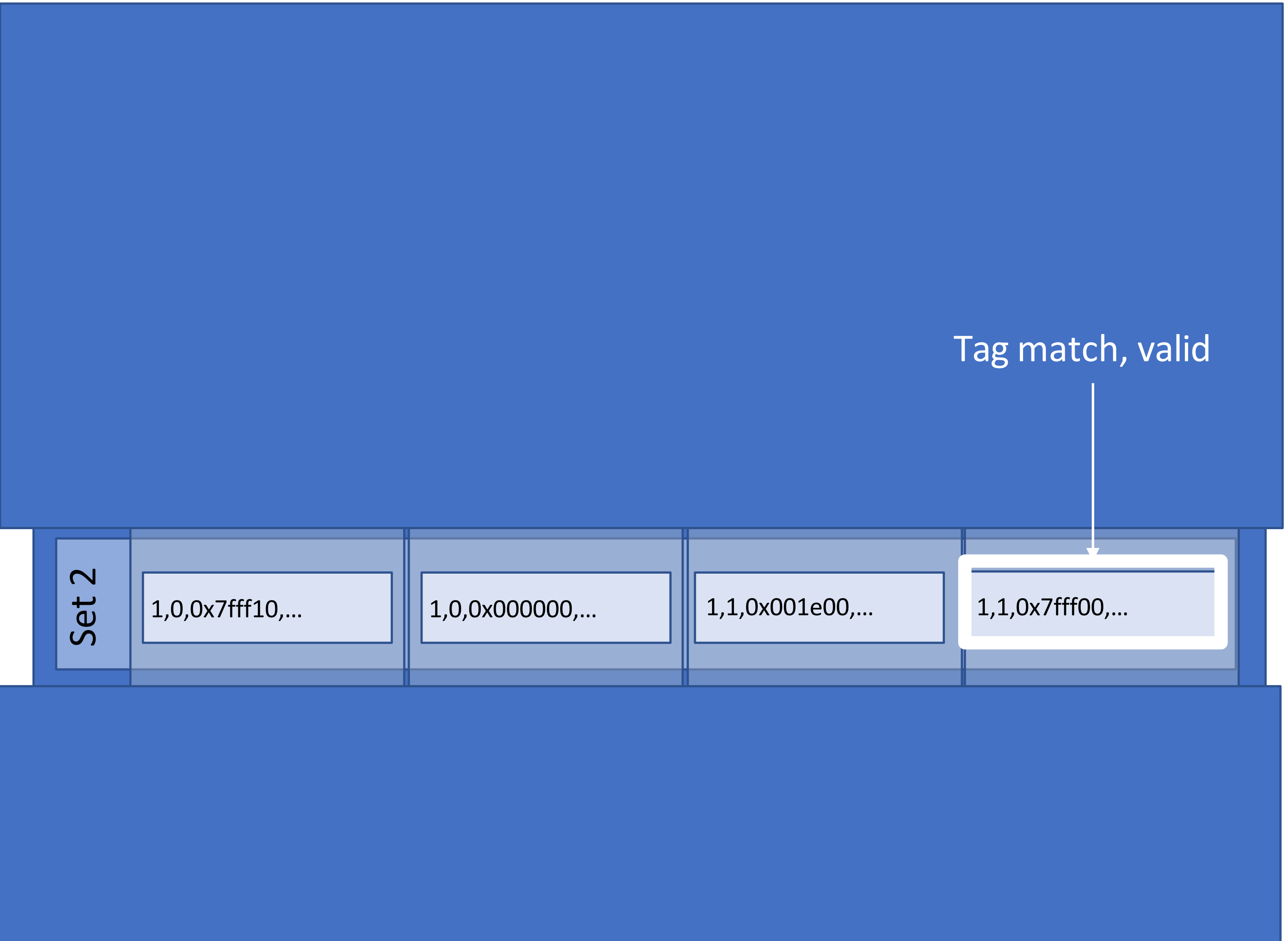
Review: Accessing the cache

```
lb x6 0x7fff0053
```

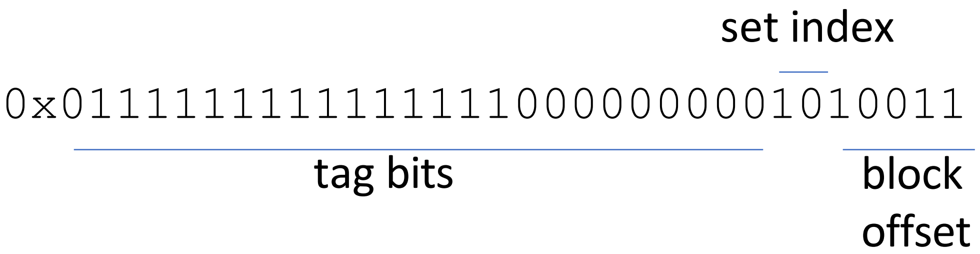


Review: Accessing the cache - Hit

```
lb x6 0x7fff0053
```



Step 3: Check valid, compare tags

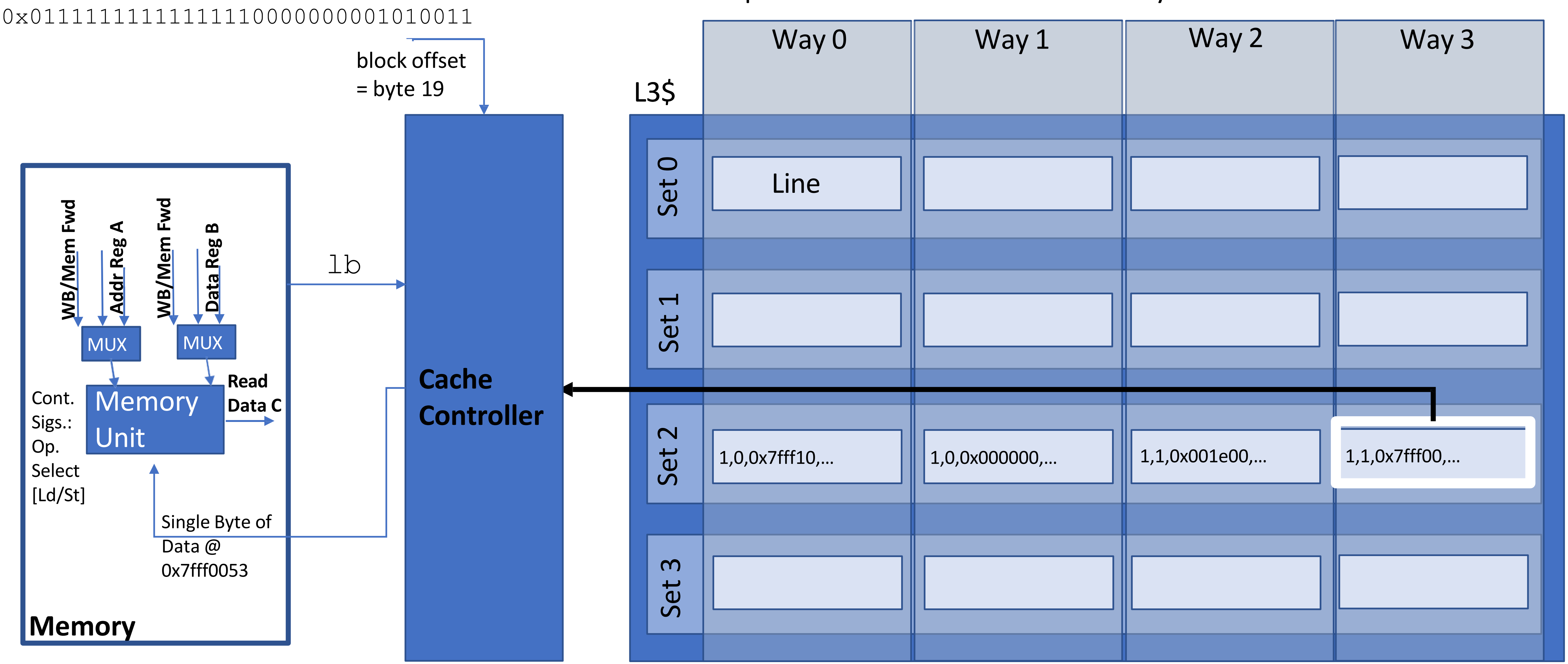


Valid	Dirty	Tag	32 bytes data

Review: Accessing the cache - Hit

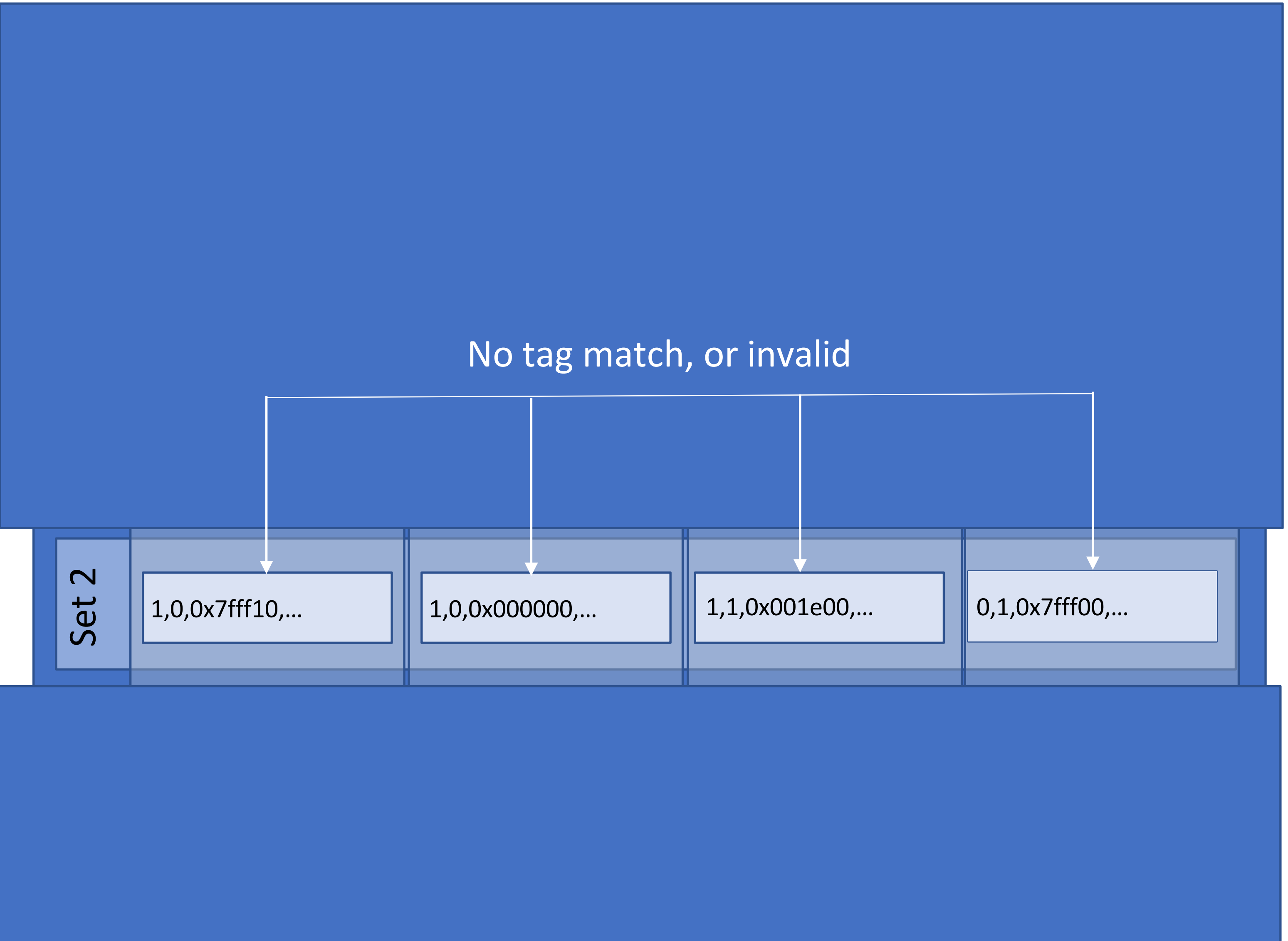
```
lb x6 0x7fff0053
```

Step 4: Fetch cache block for memory unit via cache controller



Review: Accessing the cache - Miss

```
lb x6 0x7fff0053
```



Step 3: Check valid, compare tags

set index

0x01111111111111110000000001010011

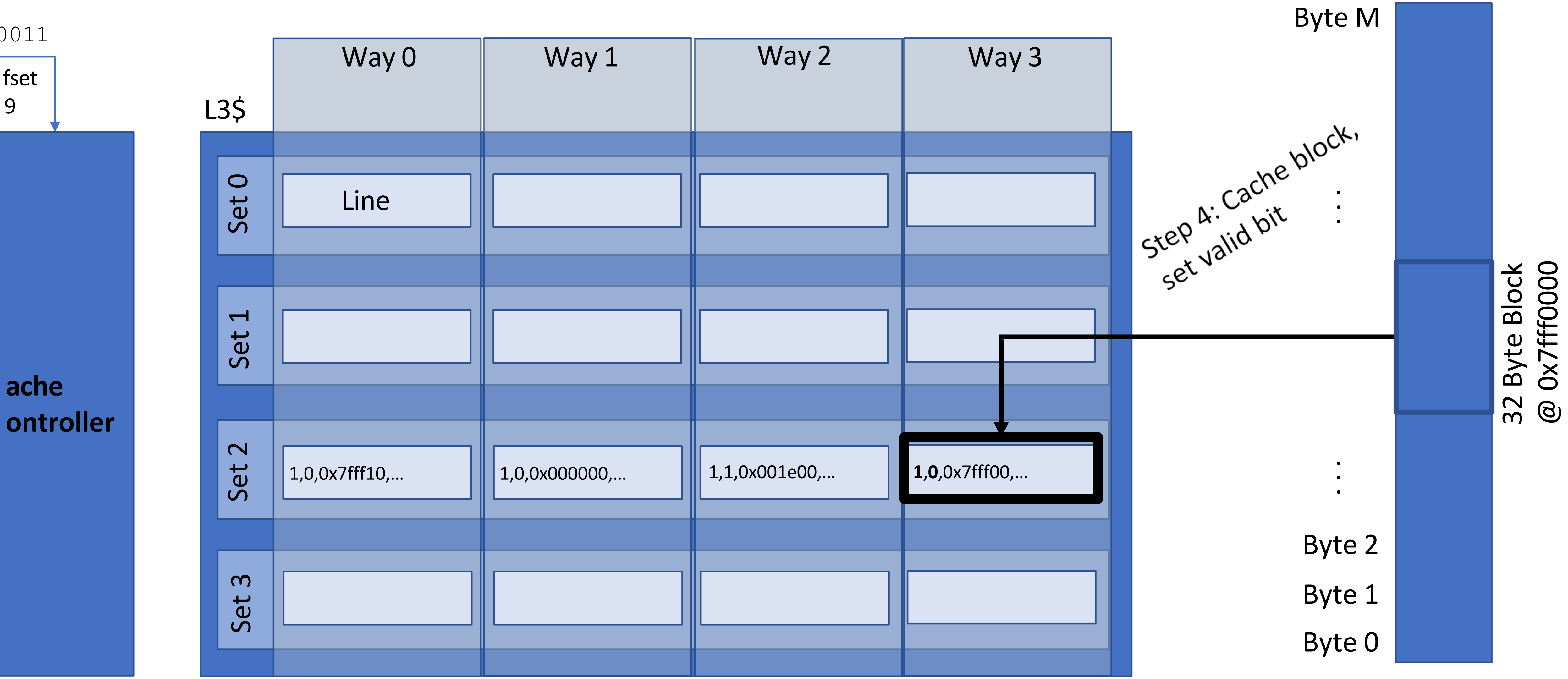
tag bits

block offset

Valid	Dirty	Tag	32 bytes data

Review: Accessing the cache - Miss

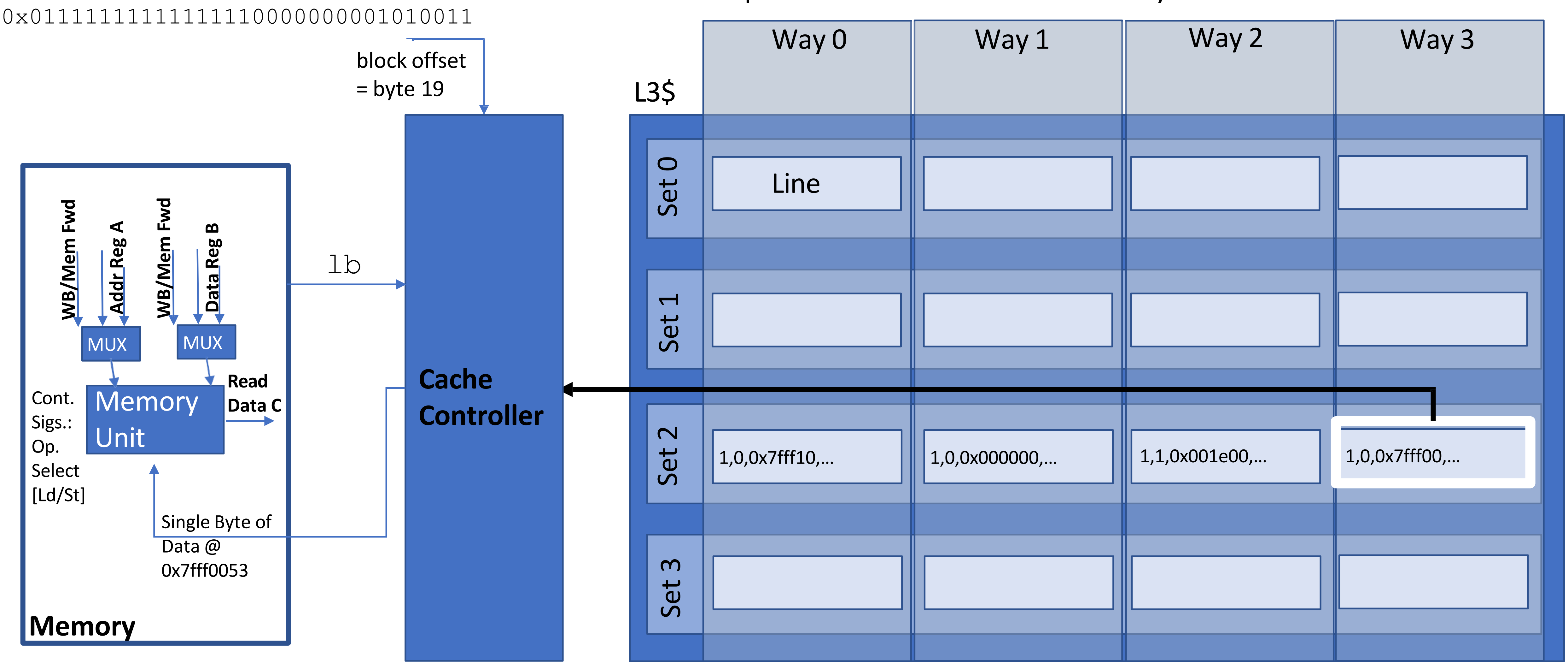
```
lb x6 0x7fff0053
```



Review: Accessing the cache - Miss

```
lb x6 0x7fff0053
```

Step 5: Fetch cache block for memory unit via cache controller



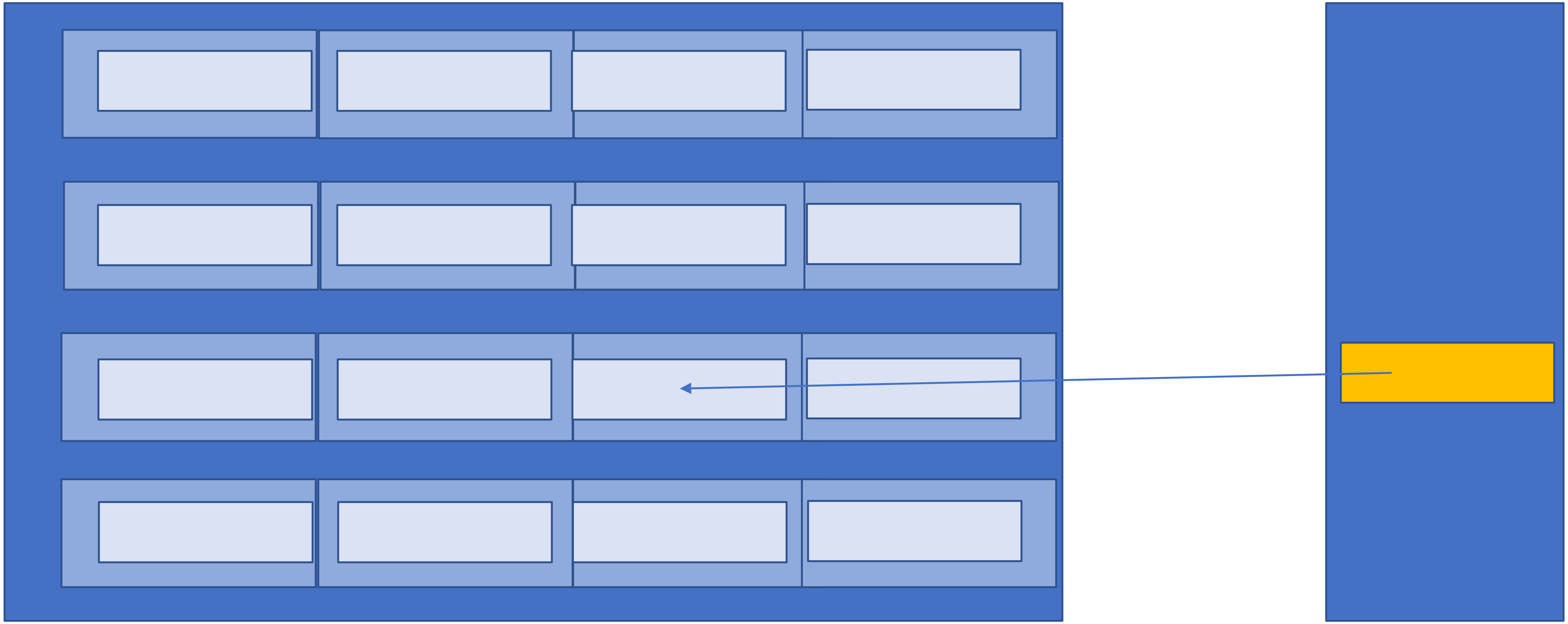
Why do we miss in the cache?

Why do we miss in the cache?

- The 3 C’s of misses
  - Compulsory
  - Conflict
  - Capacity

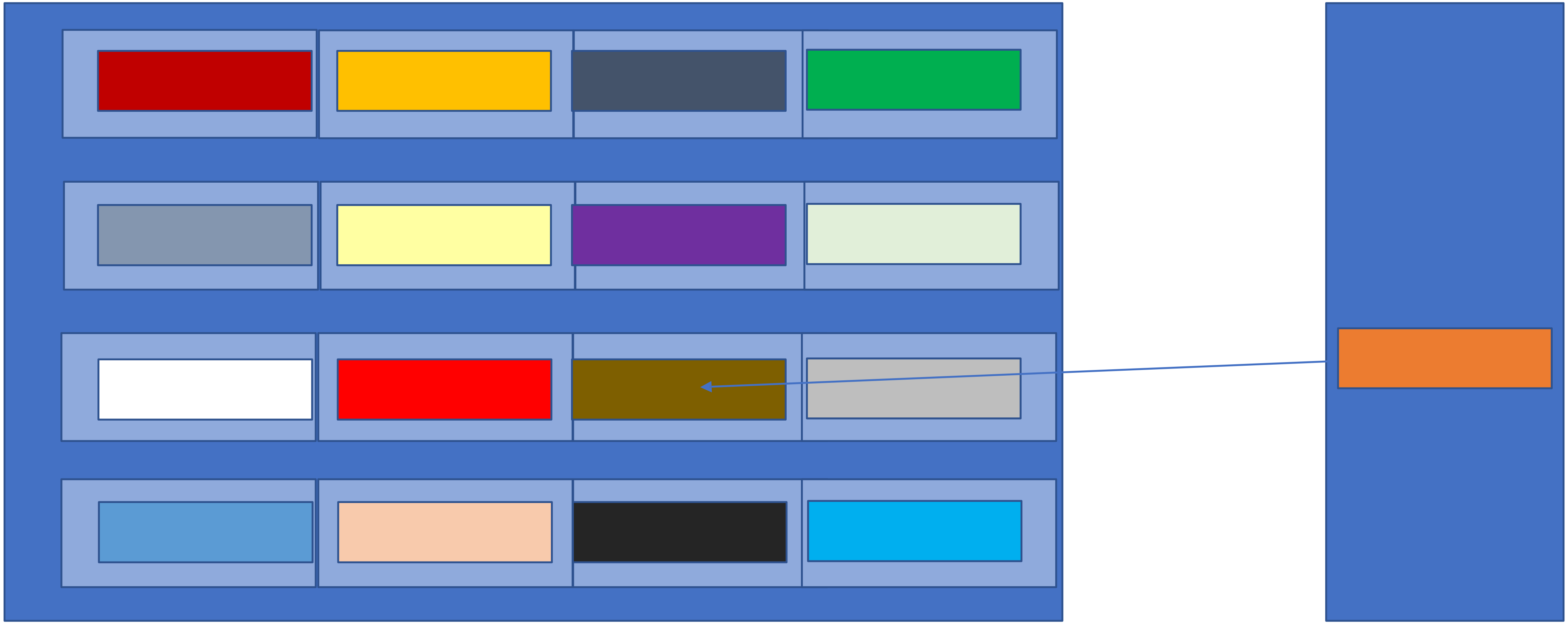
Why miss? Compulsory misses

First access to any block of memory is always a miss; these misses are **compulsory**



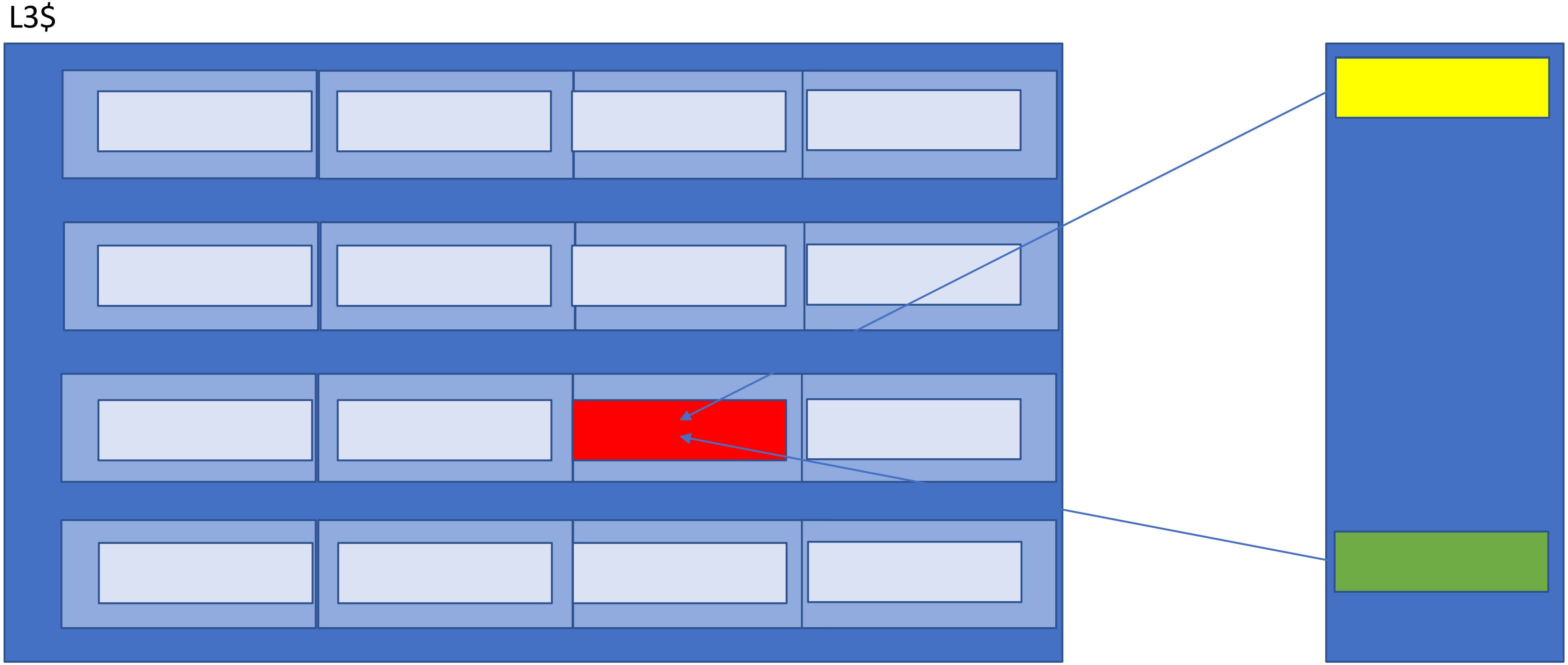
Why miss? Capacity misses

Working set of program contains more data than can be cached at one time.
By the **pigeonhole principle** caching all data requires missing at least once

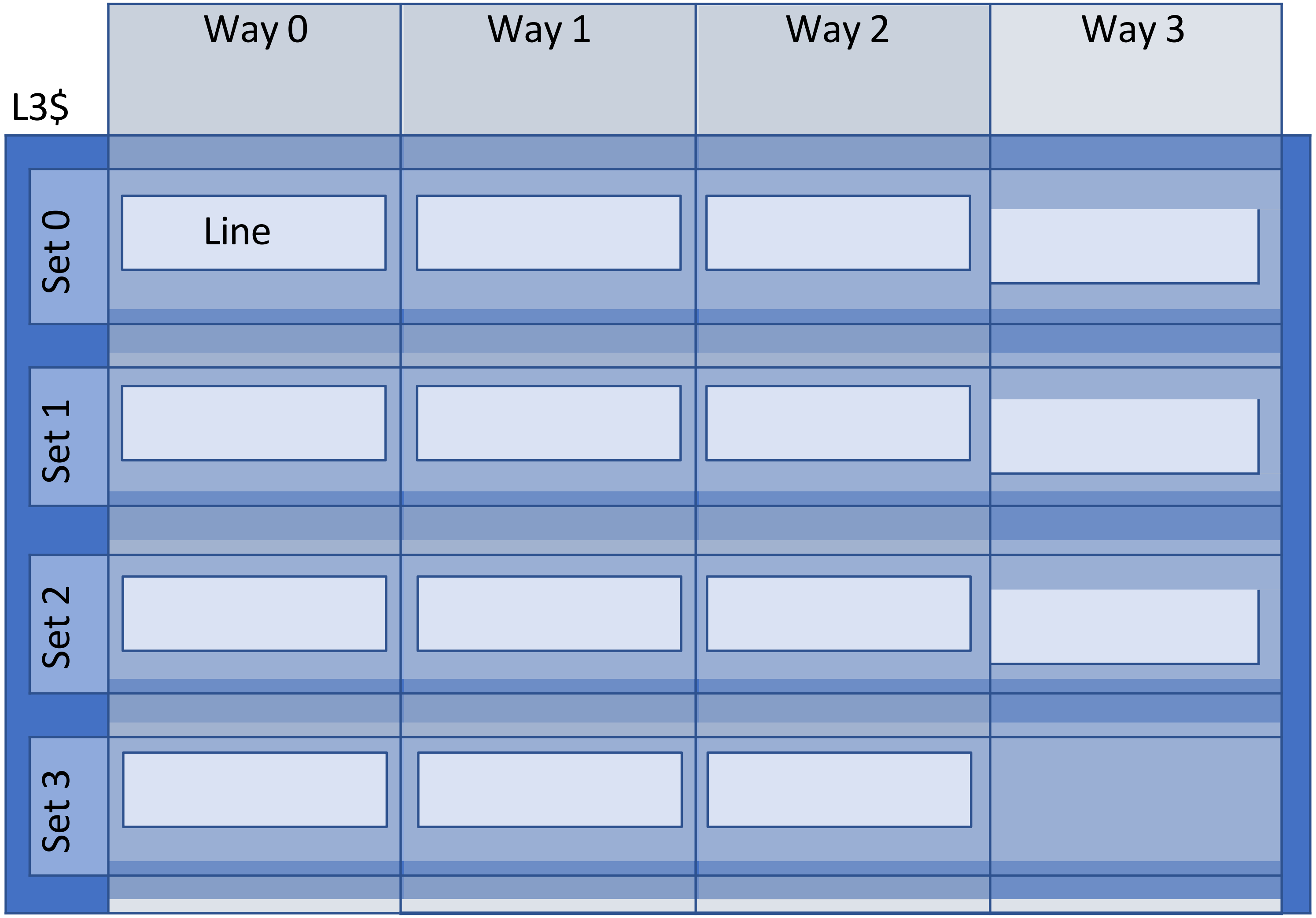


Why miss? Conflict misses

Multiple blocks of memory map to the same location in the cache and **conflict**, even if there is still some empty space in the cache



How many bits in tag/index/offset?



`lb x6 0x7fff0053`

set index

0x01111111111111110000000001010011

tag bits

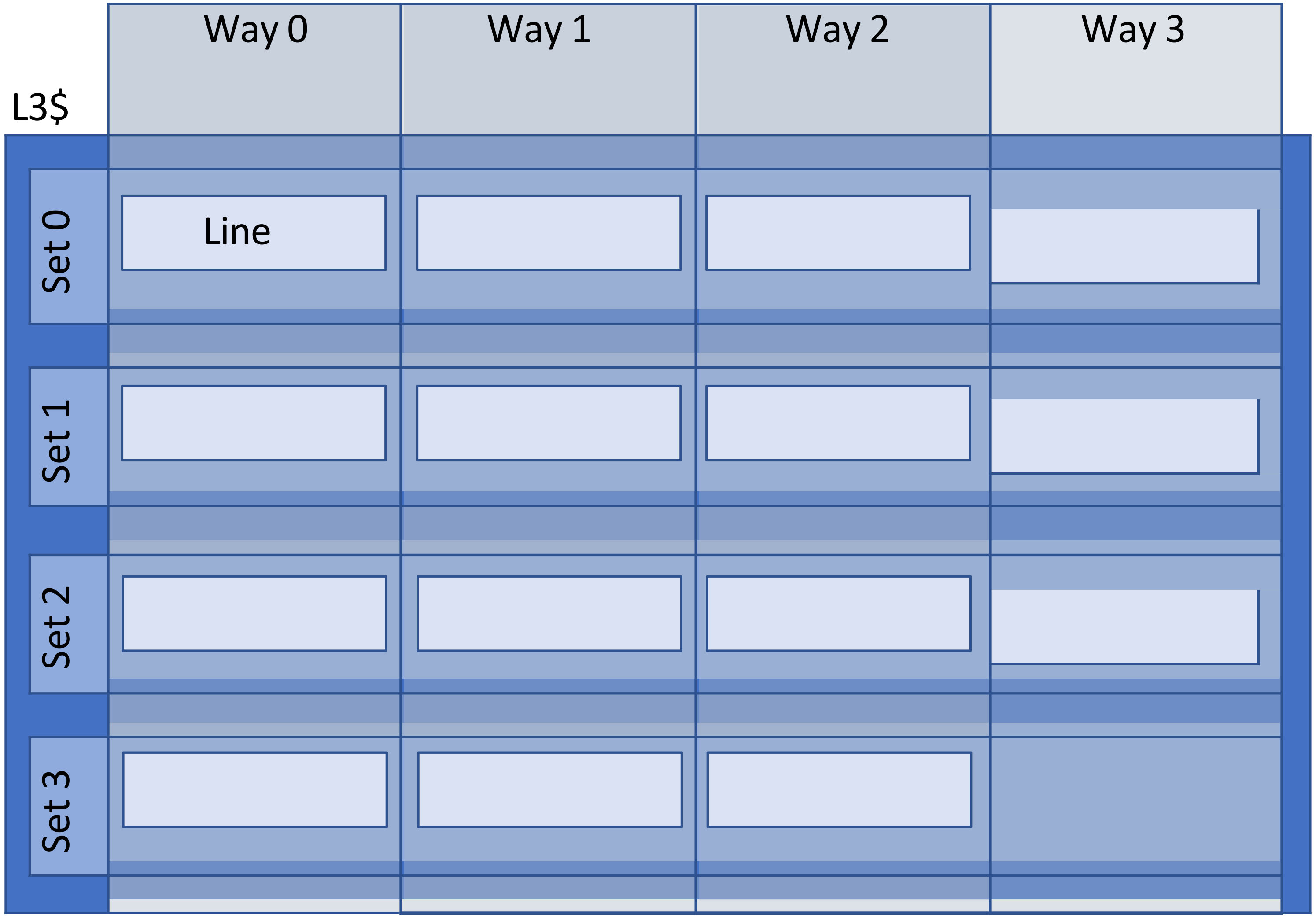
block offset

Why these numbers of bits?

Valid	Dirty	Tag	32 bytes data

Total cache size = 32B x 4 sets x 4 ways = 512B

How many bits in tag/index/offset?



`lb x6 0x7fff0053`

set index

`0x01111111111111110000000001010011`

tag bits

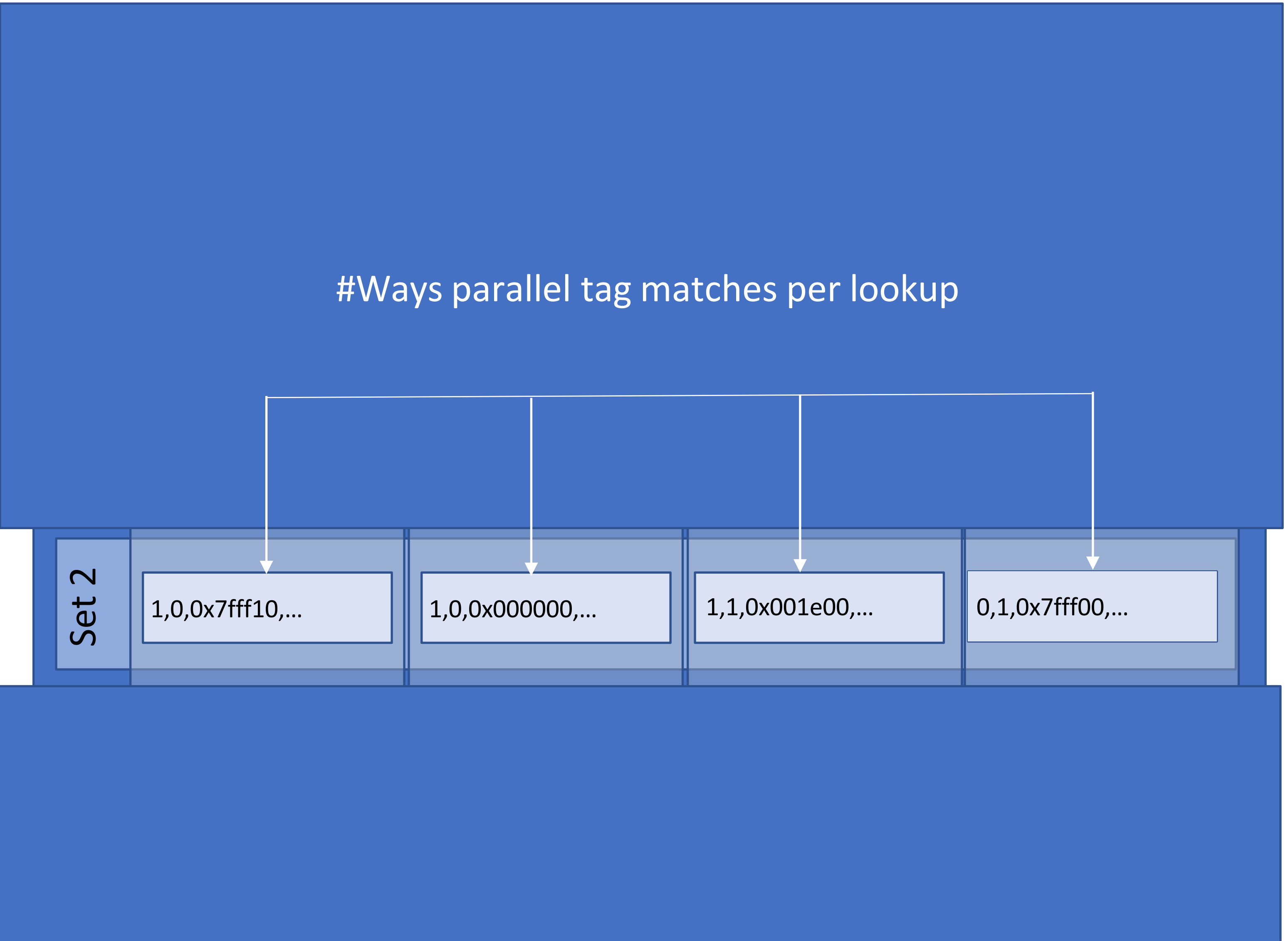
block offset

Enough **block offset** bits to count block bytes
Enough **set index** bits to count the sets
All left-over bits are **tag** bits
Question: what do tag bits mean?

Valid	Dirty	Tag	32 bytes data

Total cache size = 32B x 4 sets x 4 ways = 512B

How many sets should your cache have?

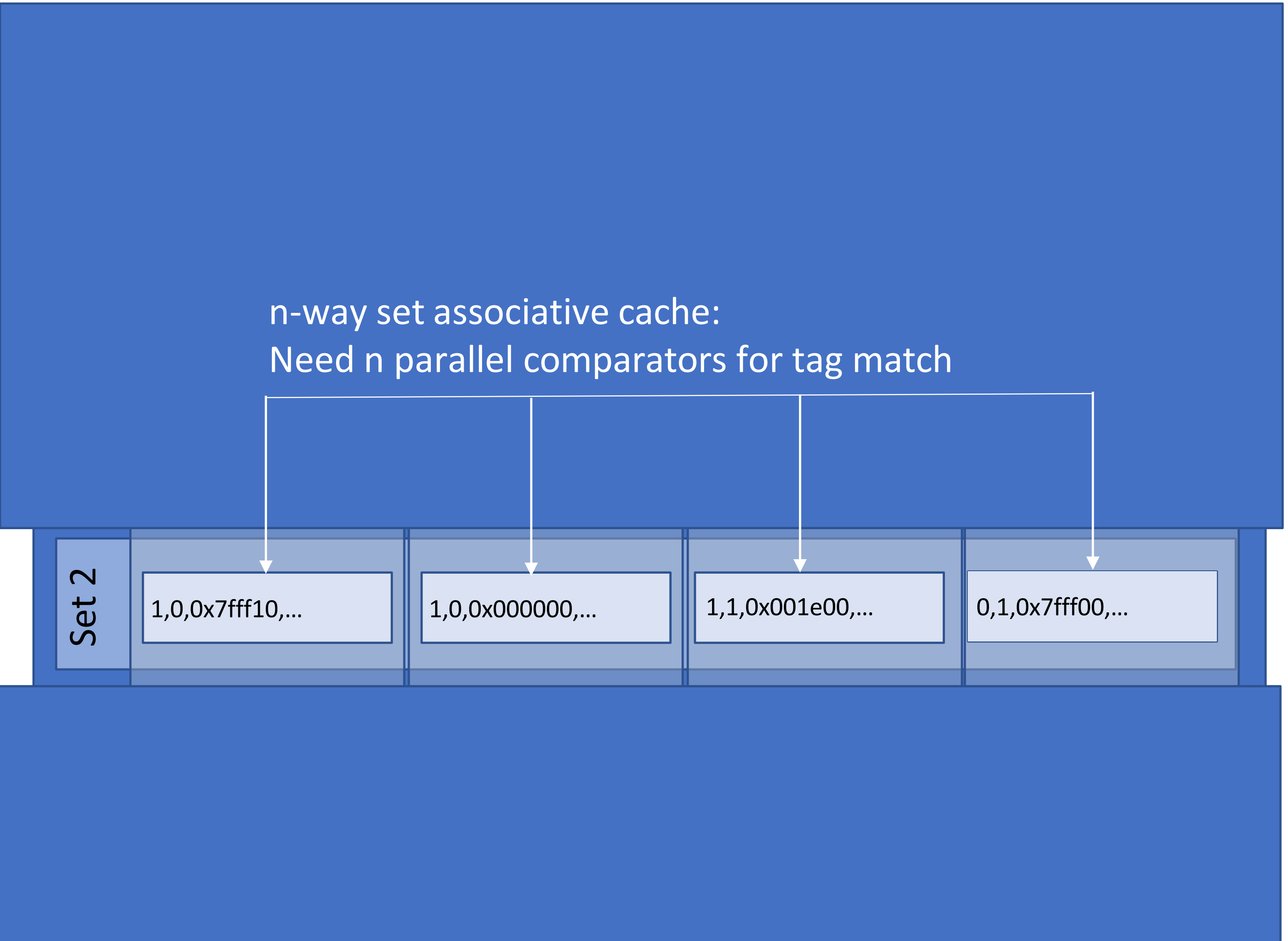


Set Associative Cache Design Procedure

1. Select total cache size
2. Select implementable #ways
3. cache size = #sets x #ways x #block_bytes
4. #sets = cache size / (#ways x #block_bytes)

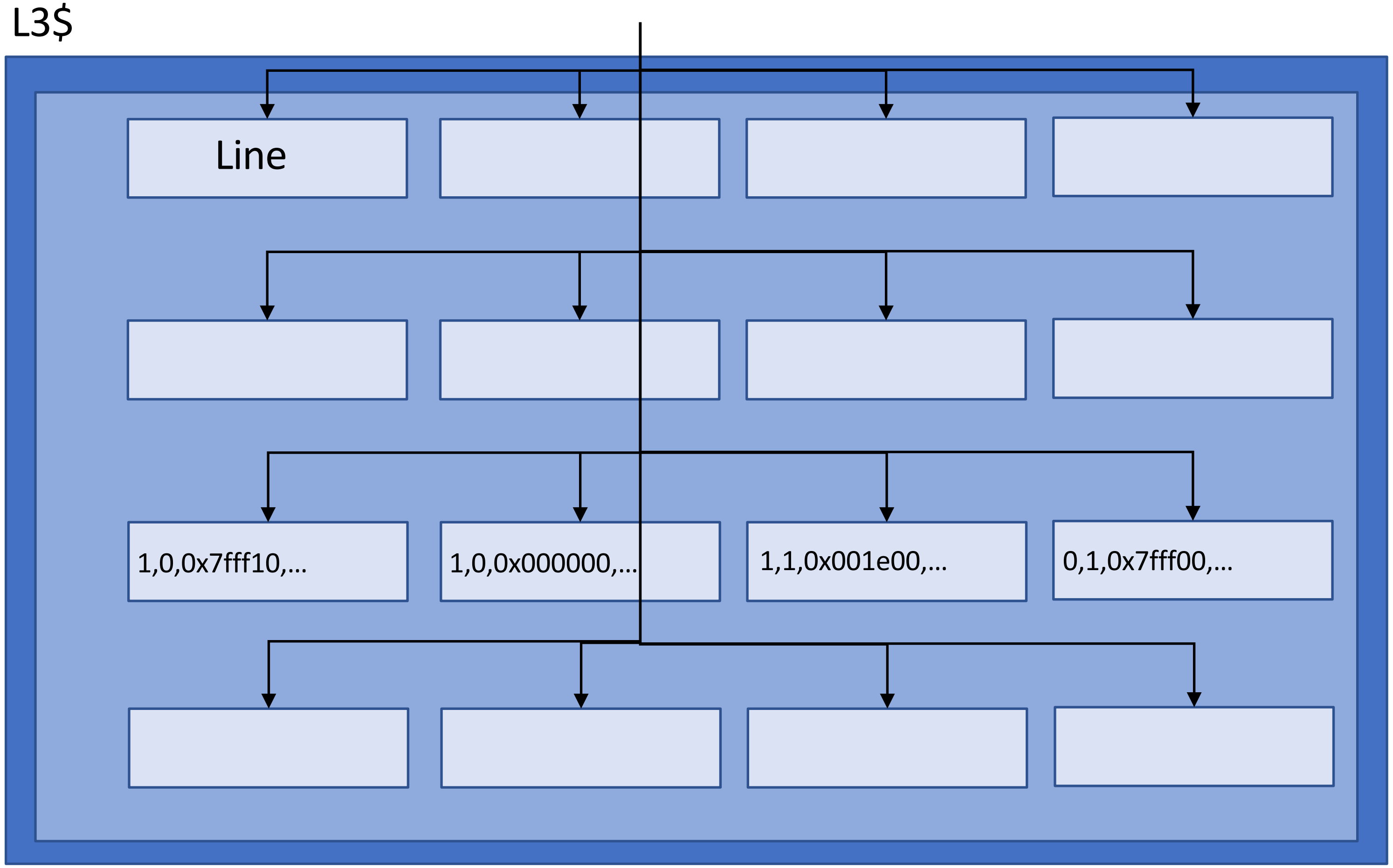
What is an implementable # of ways?

What is an implementable # ways?

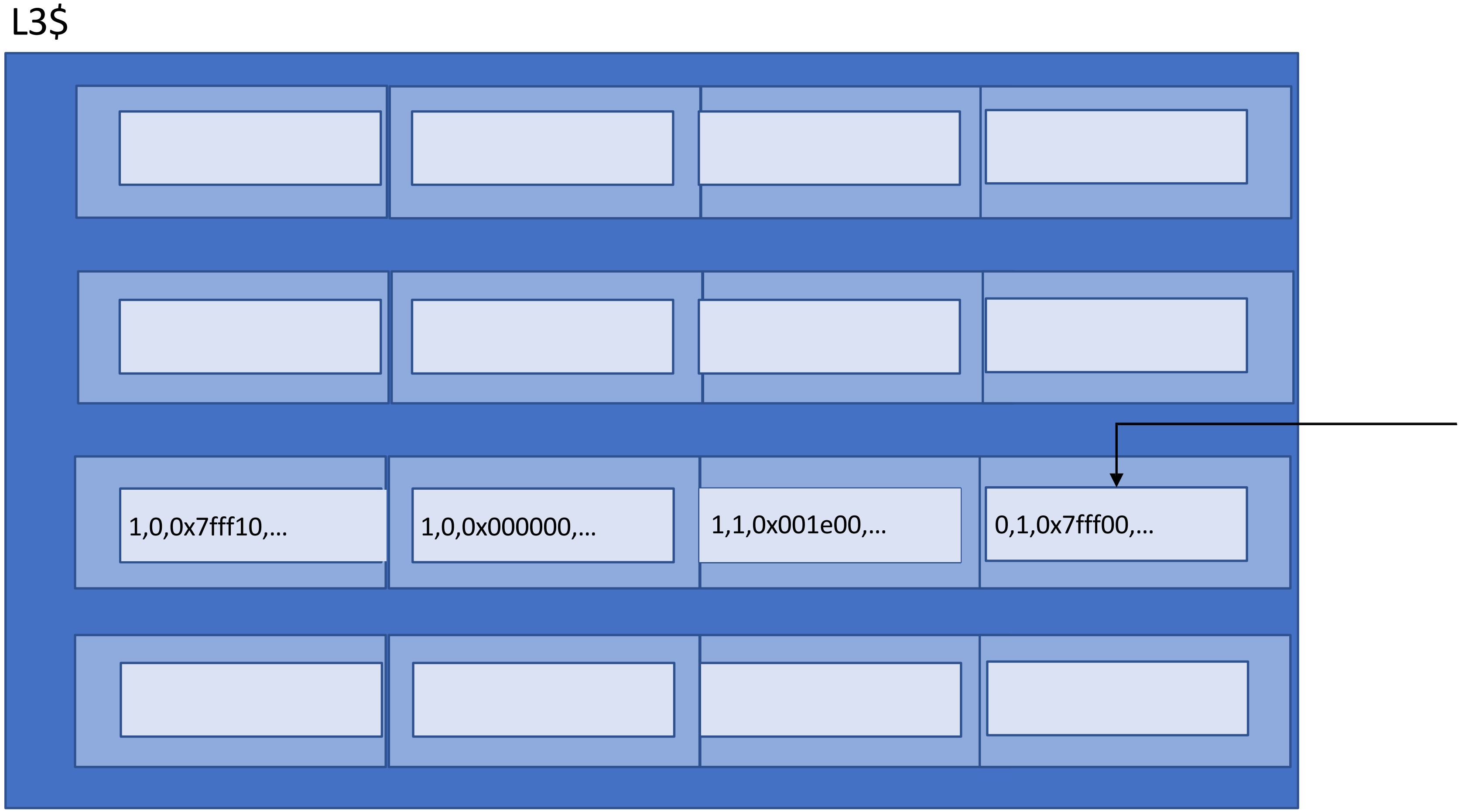


What is an implementable # ways?

Fully-associative cache:
comparators = # lines in entire cache

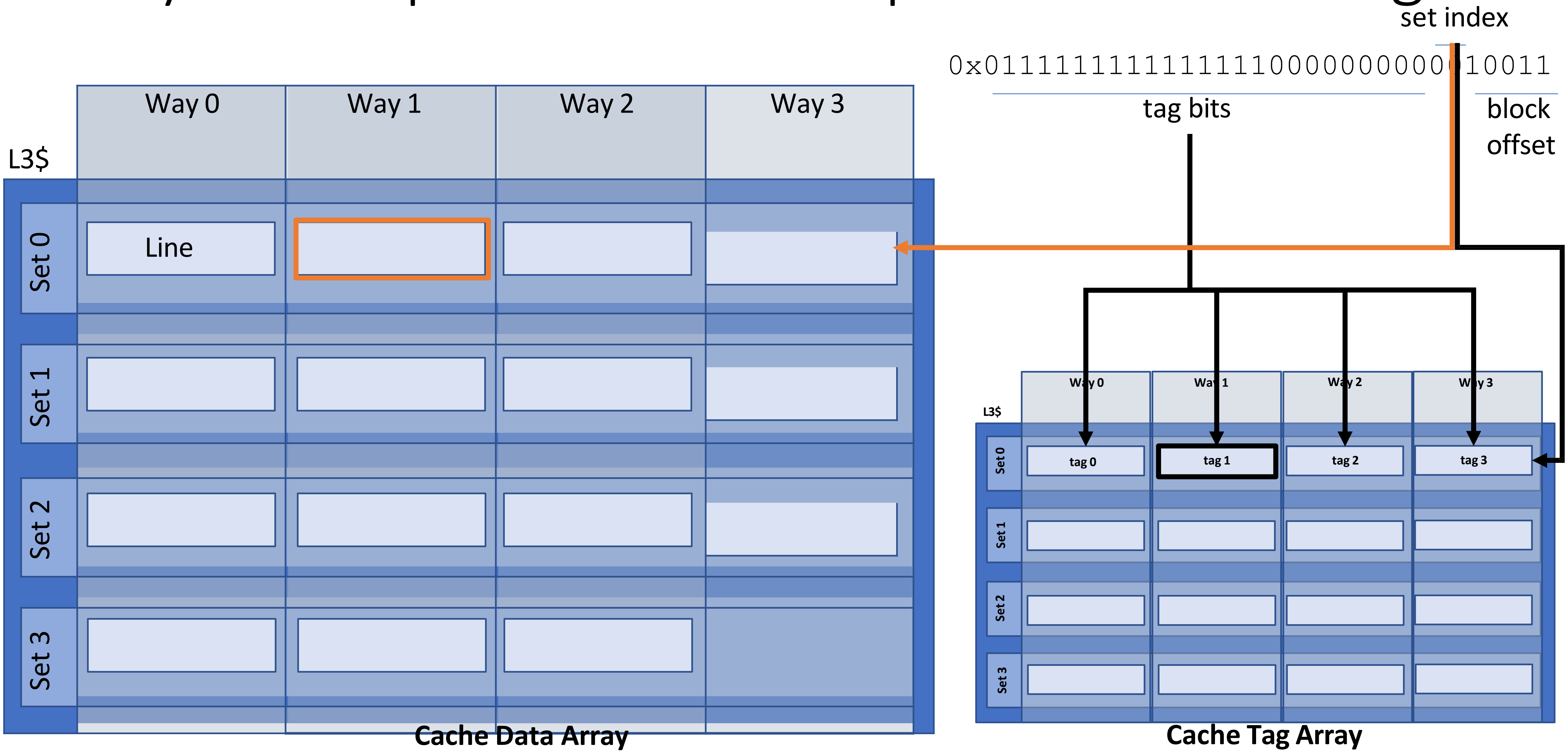


What is an implementable # ways?

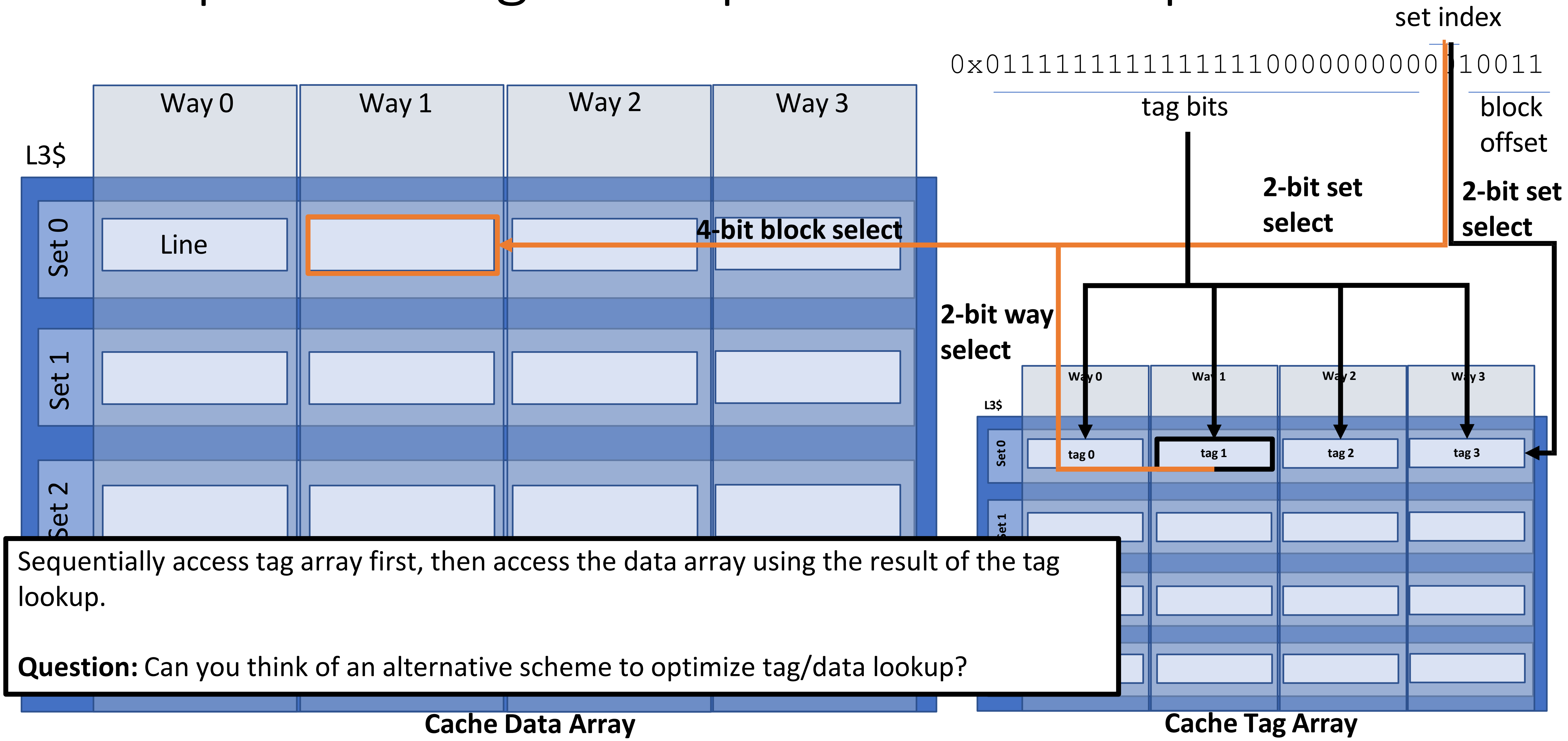


Direct mapped cache:
1 comparator because each set contains a single line

Physical implementation separates data & tags



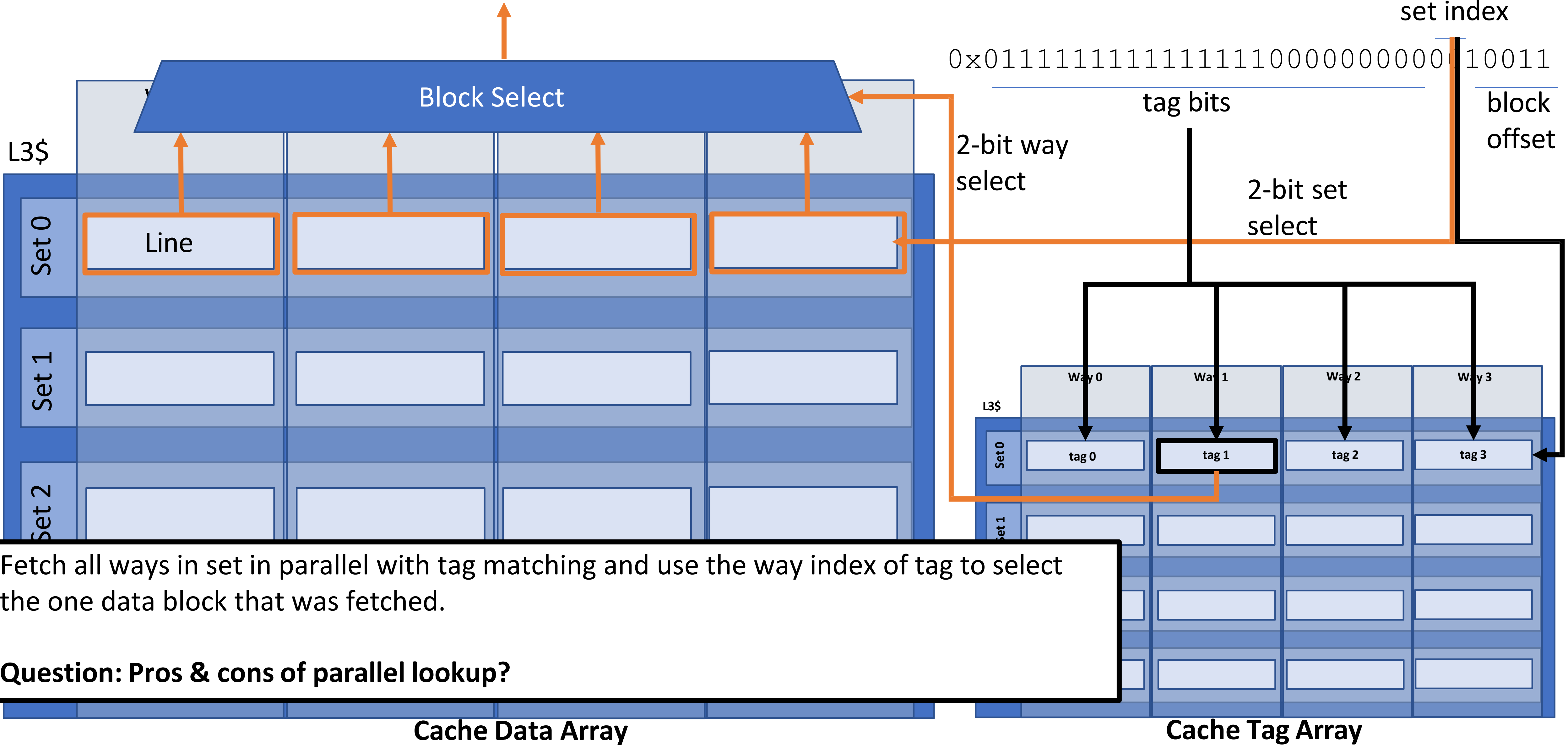
Sequential Tag Lookup & Data Lookup



Sequentially access tag array first, then access the data array using the result of the tag lookup.

Question: Can you think of an alternative scheme to optimize tag/data lookup?

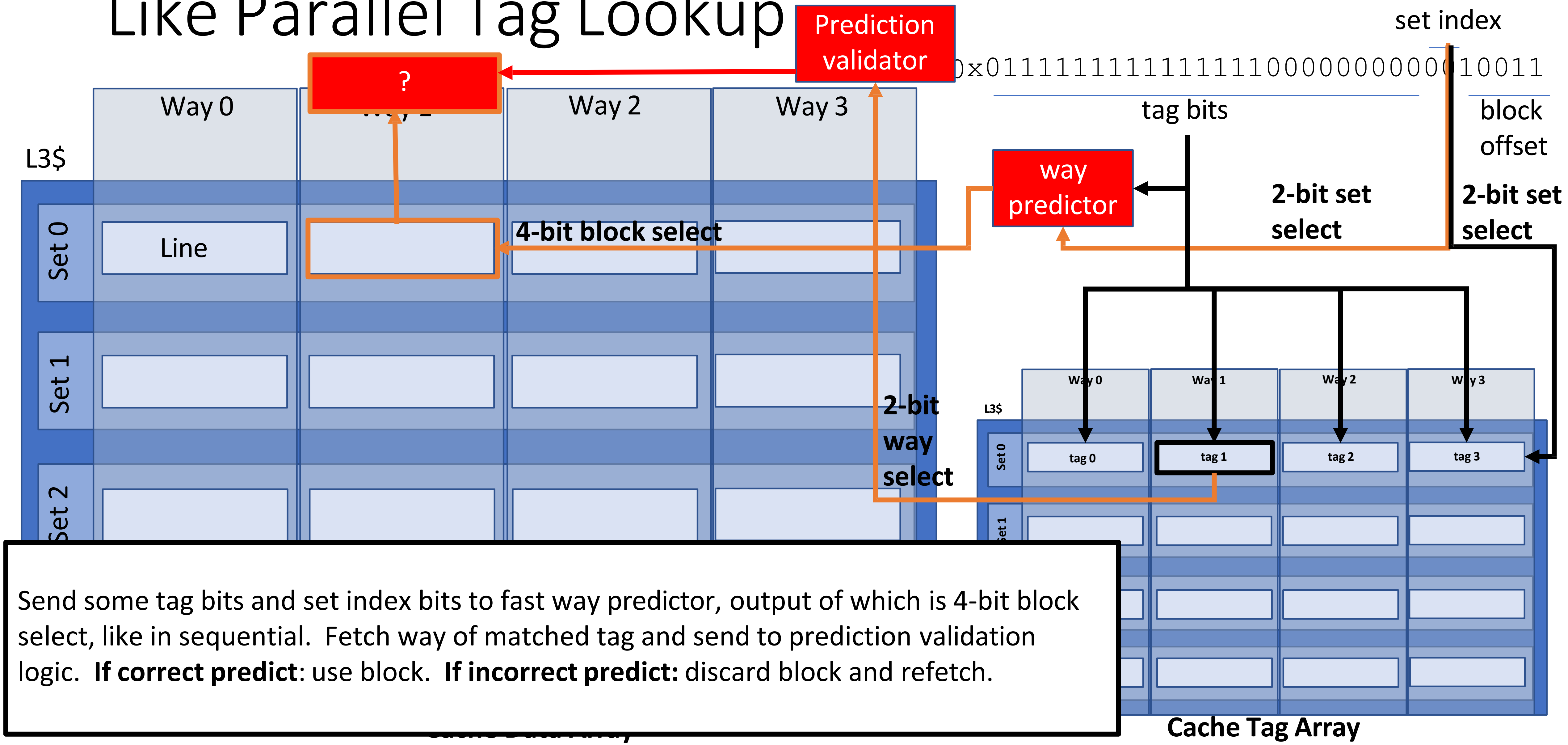
Parallel Tag Lookup & Data Lookup



Fetch all ways in set in parallel with tag matching and use the way index of tag to select the one data block that was fetched.

Question: Pros & cons of parallel lookup?

Way Prediction: Cost Like Sequential, Performance Like Parallel Tag Lookup



Send some tag bits and set index bits to fast way predictor, output of which is 4-bit block select, like in sequential. Fetch way of matched tag and send to prediction validation logic. **If correct predict**: use block. **If incorrect predict:** discard block and refetch.

Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20). Association for Computing Machinery, New York, NY, USA, 813–825. https://doi.org/10.1145/3320269.3384746

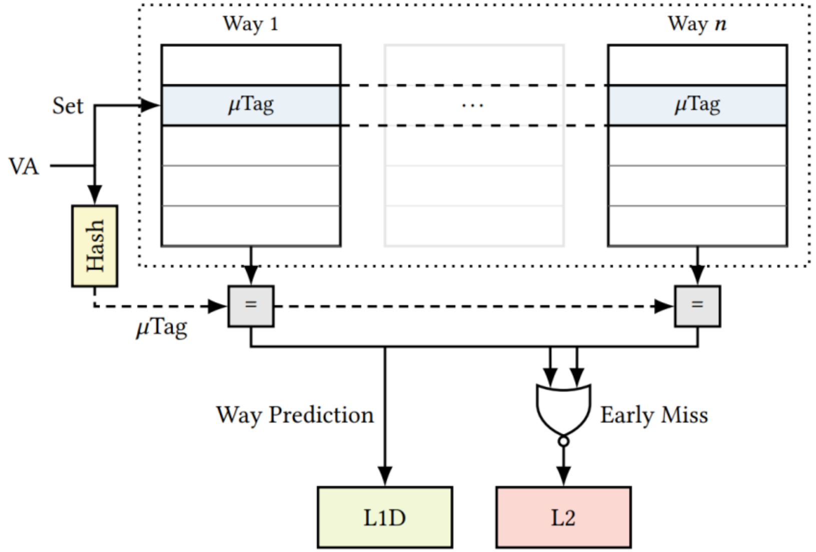


Figure 1: Simplified illustration of AMD's way predictor.

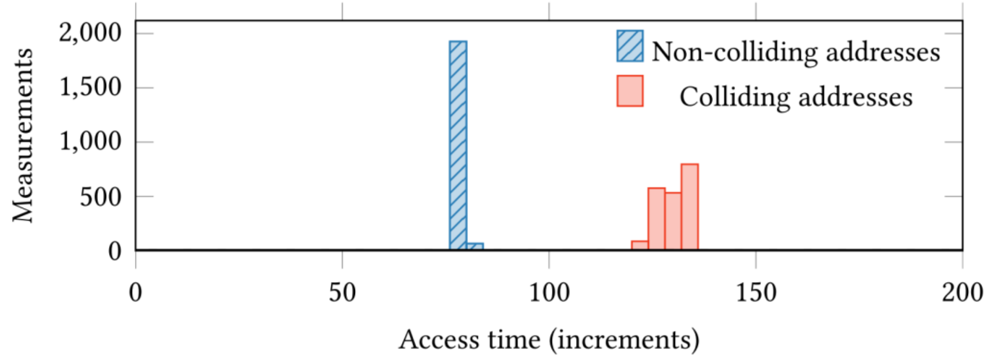
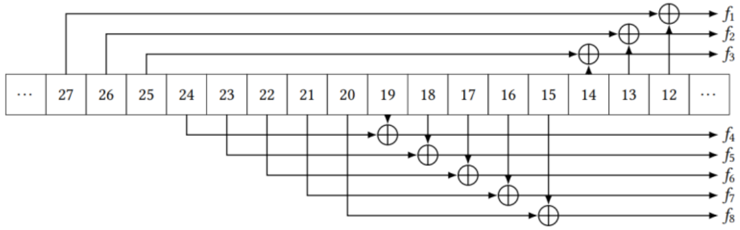
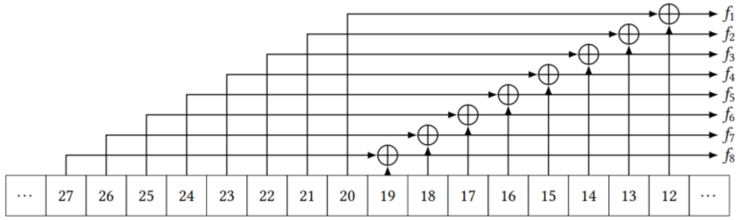


Figure 2: Measured duration of 250 alternating accesses to addresses with and without the same μTag.



(a) Zen, Zen+, Zen 2



(b) Bulldozer, Piledriver, Steamroller

Figure 3: The recovered hash functions use bits 12 to 27 of the virtual address to compute the μTag.

Cost of Associativity

512 Bytes, 256-bit (32B) lines, 1-way

```
$ ./destiny config/SRAM_512_1_256.cfg
```

Read Latency = 55.4943ps
Tag Read Latency = 277.84ps
Write Latency = 54.7831ps
Tag Write Latency = 212.575ps

Read Bandwidth = 674.493GB/s
Write Bandwidth = 633.944GB/s

Tag Read Dynamic Energy = 0.281324pJ
Tag Write Dynamic Energy = 0.222833pJ

512 Bytes, 256-bit (32B) lines, 4-way

```
$ ./destiny config/SRAM_512_4_256.cfg
```

Read Latency = 83.4307ps
Tag Read Latency = 293.516ps
Write Latency = 83.1343ps
Tag Write Latency = 226.518ps

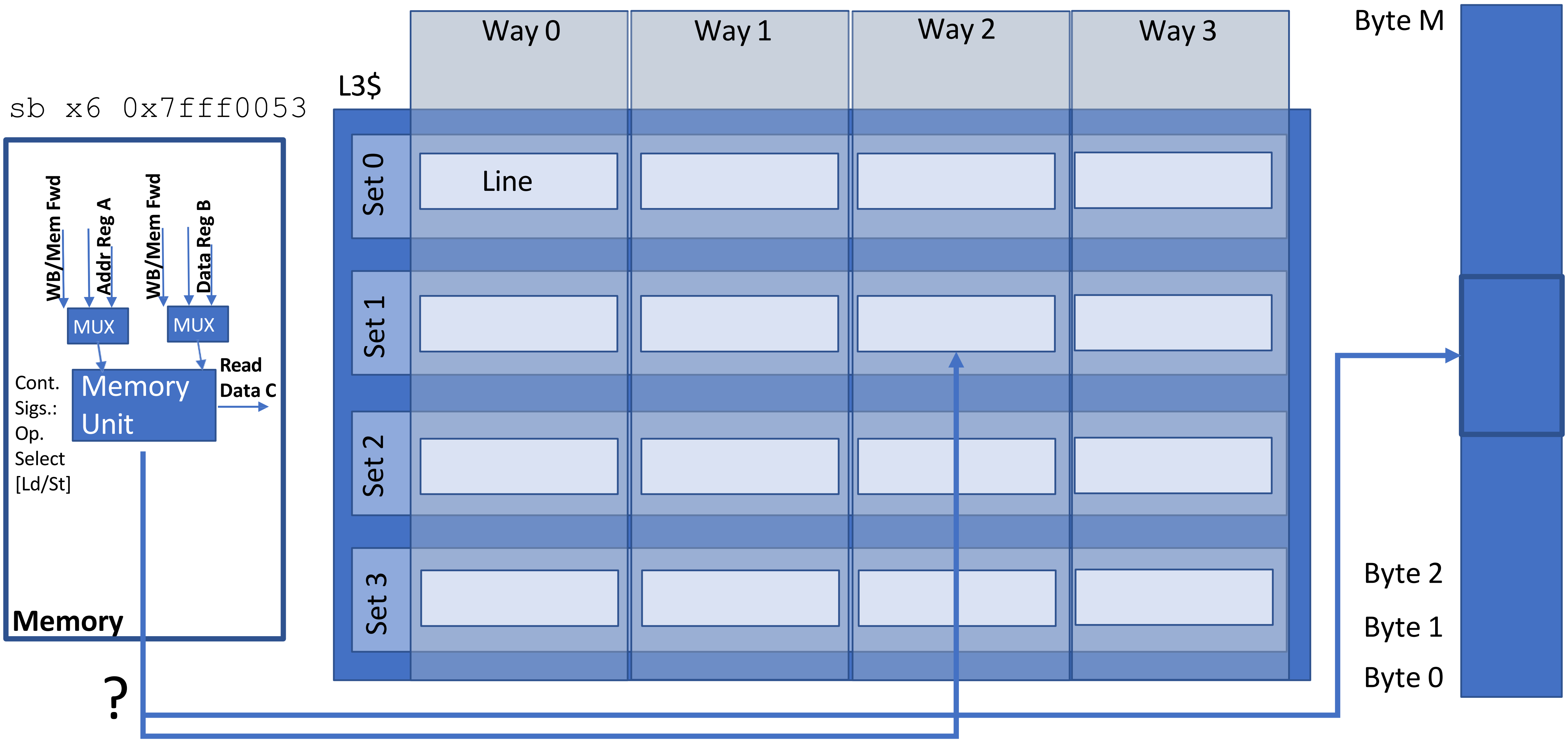
Read Bandwidth = 480.942GB/s
Write Bandwidth = 500.715GB/s

Tag Read Dynamic Energy = 1.01651pJ
Tag Write Dynamic Energy = 0.758075pJ

Higher associativity avoids conflict misses at an additional cost in hit latency & energy

Write Policies - Allocation

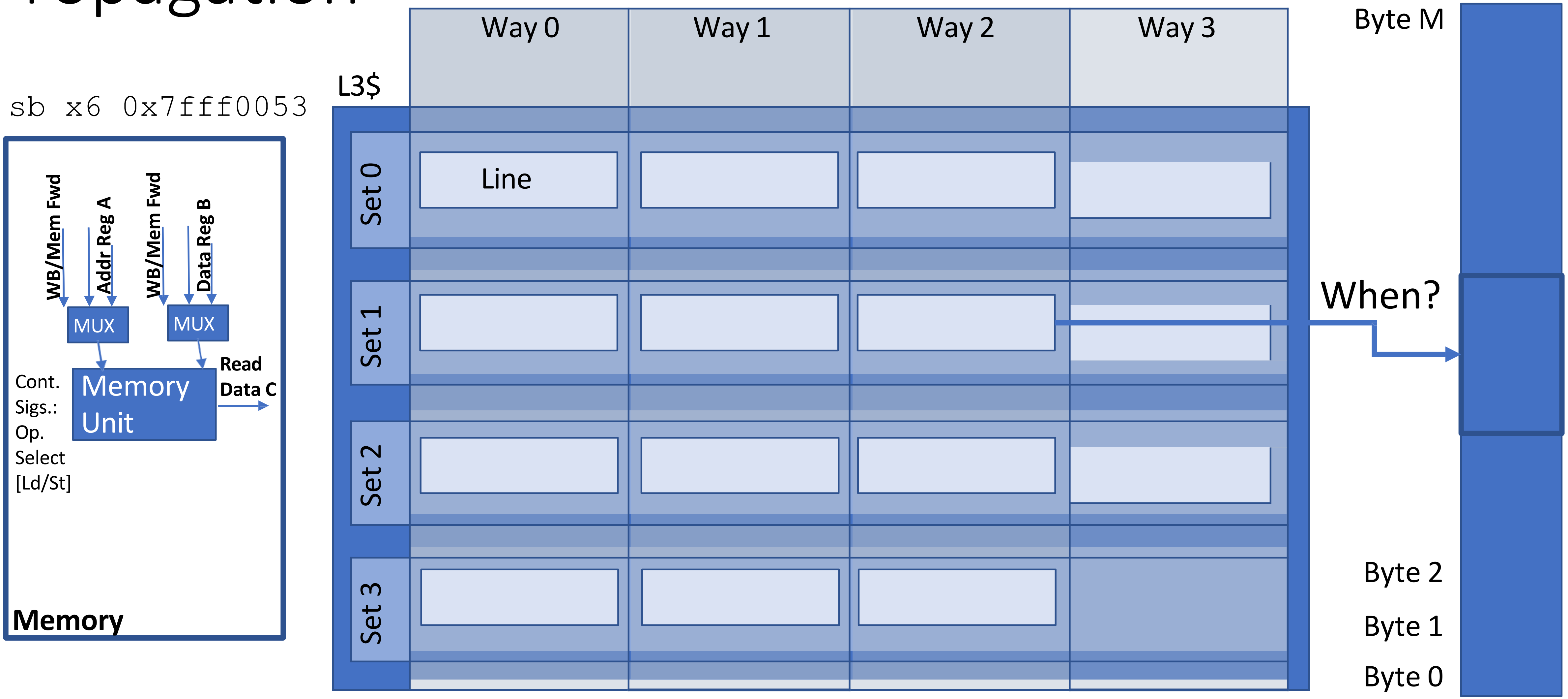
Write-Allocate: Stores go to cache
Write-No-Allocate: Stores do not go to cache



Write Policies - Propagation

Write-Back: Wait until line evicted to writeback
Write-Through: Writeback immediately on store

`sb x6 0x7fff0053`



Recall 18x13: Snoopy Caches

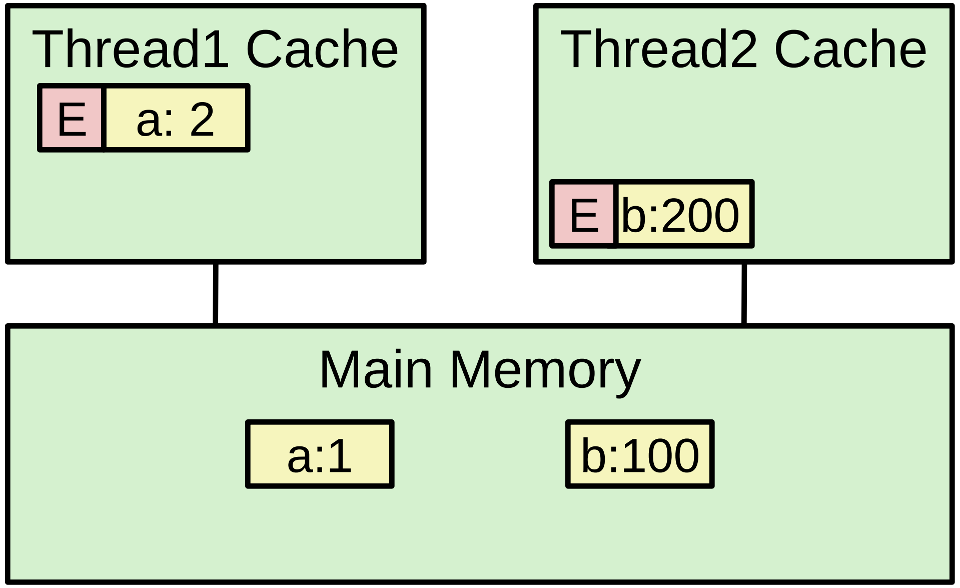
```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

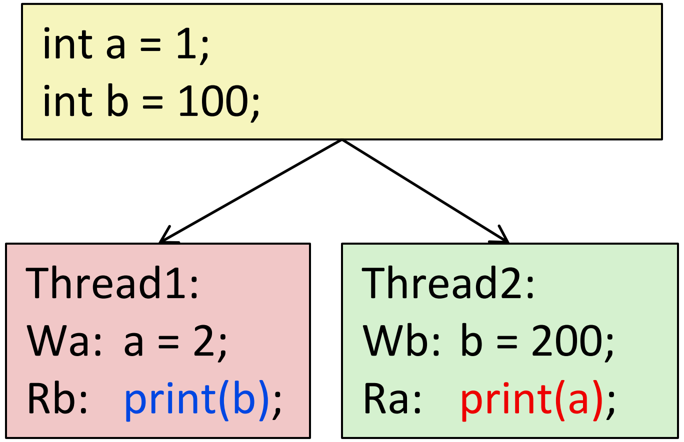
```
Thread2:
Wb:  b = 200;
Ra:  print(a);
```

Tag each cache block with state

Invalid	Cannot use value
Shared	Readable copy
Exclusive	Writeable copy

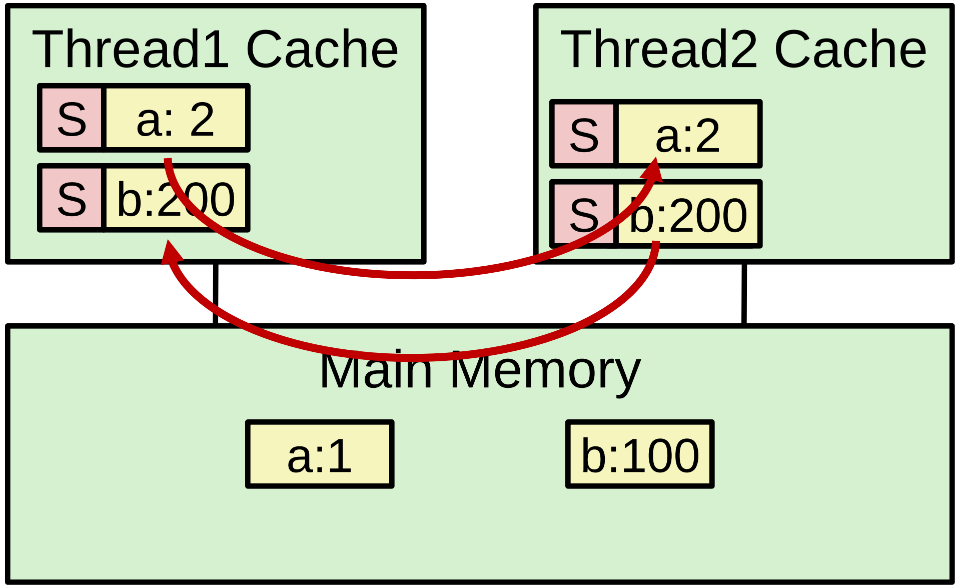


Recall 18x13: Snoopy Caches



Tag each cache block with state

Invalid	Cannot use value
Shared	Readable copy
Exclusive	Writeable copy

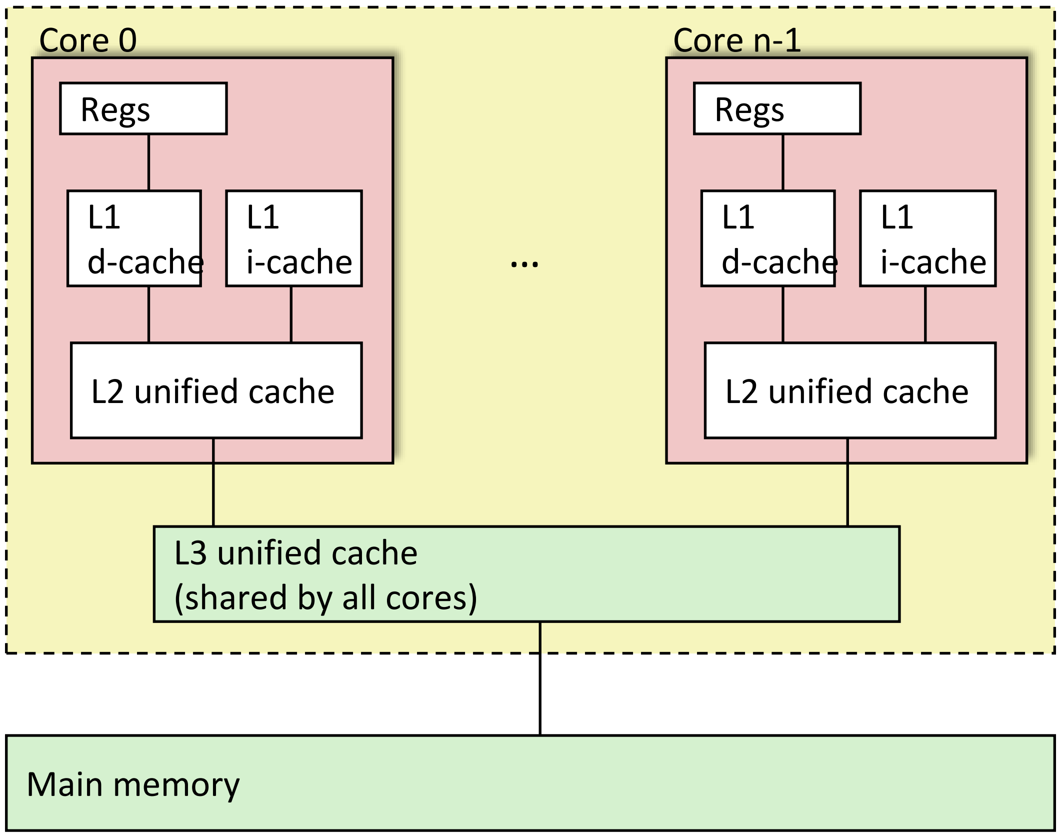


print 2

print 200

- When cache sees request for one of its E-tagged blocks
  - Supply value from cache (Note: value in memory may be stale)
  - Set tag to S

Recall 18x13: Typical Multicore Processor

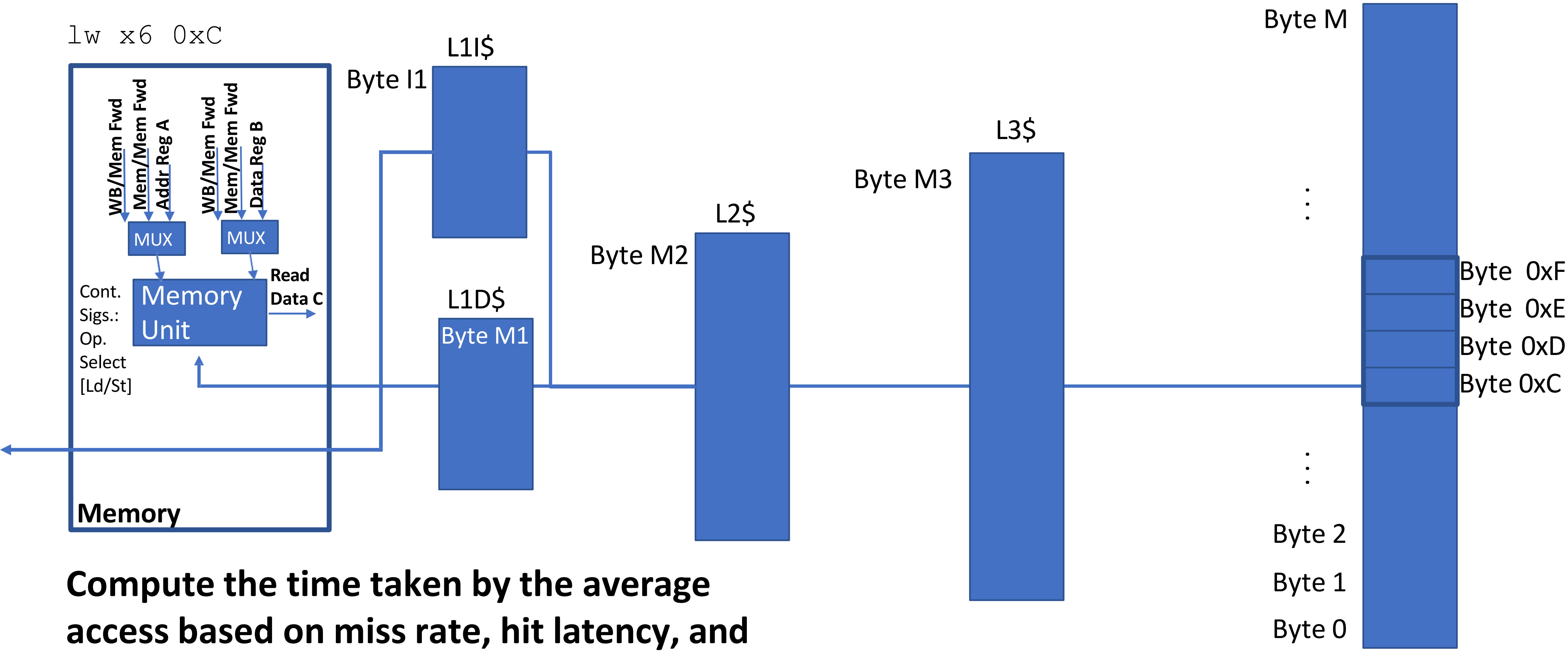


Propagation Policy v. Multicore Cache Coherency

- What is required for a snooping?
- How does propagation policy facilitate or impede this?
- What does this suggest about cache policy by level?

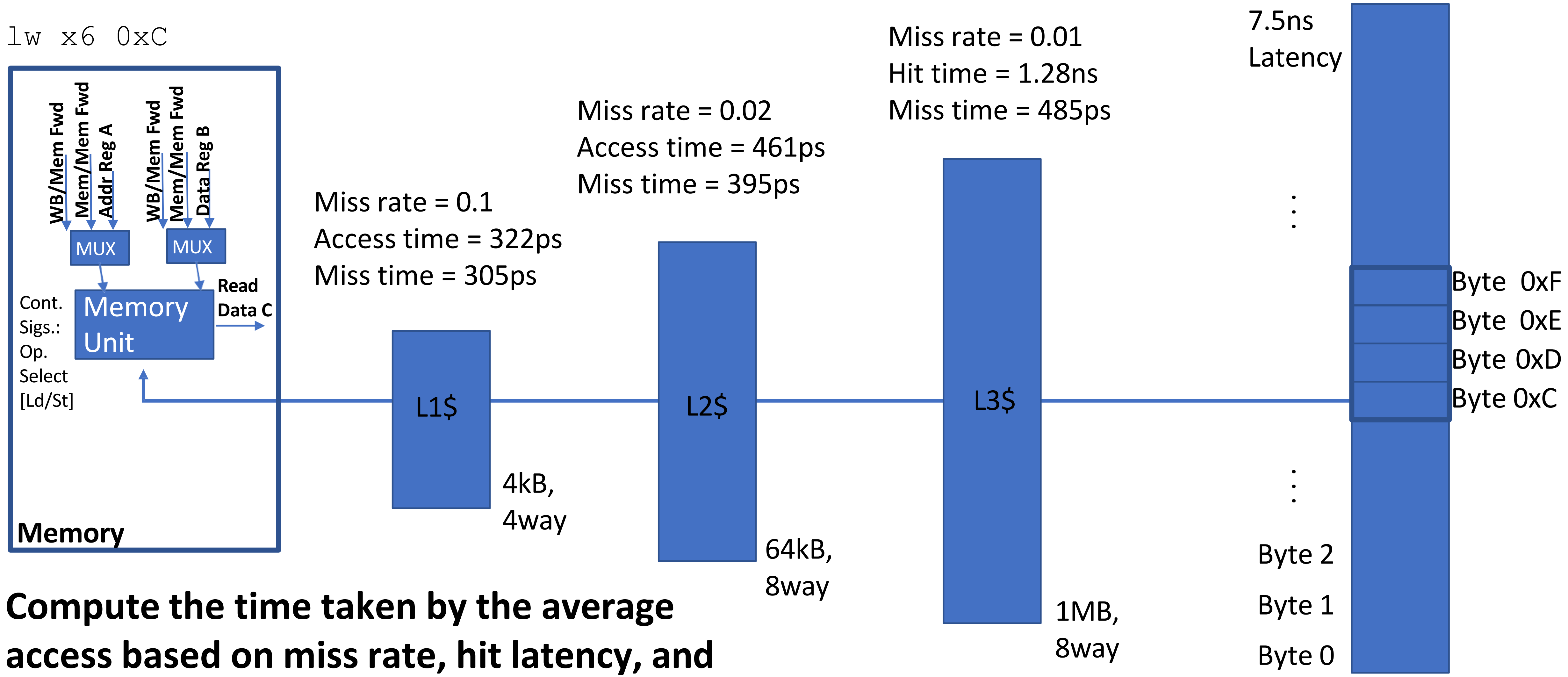
Cache Hierarchy Performance Measurement

Average Memory Access Time (AMAT): Measuring the performance of a memory hierarchy



Compute the time taken by the average access based on miss rate, hit latency, and miss penalty at each level

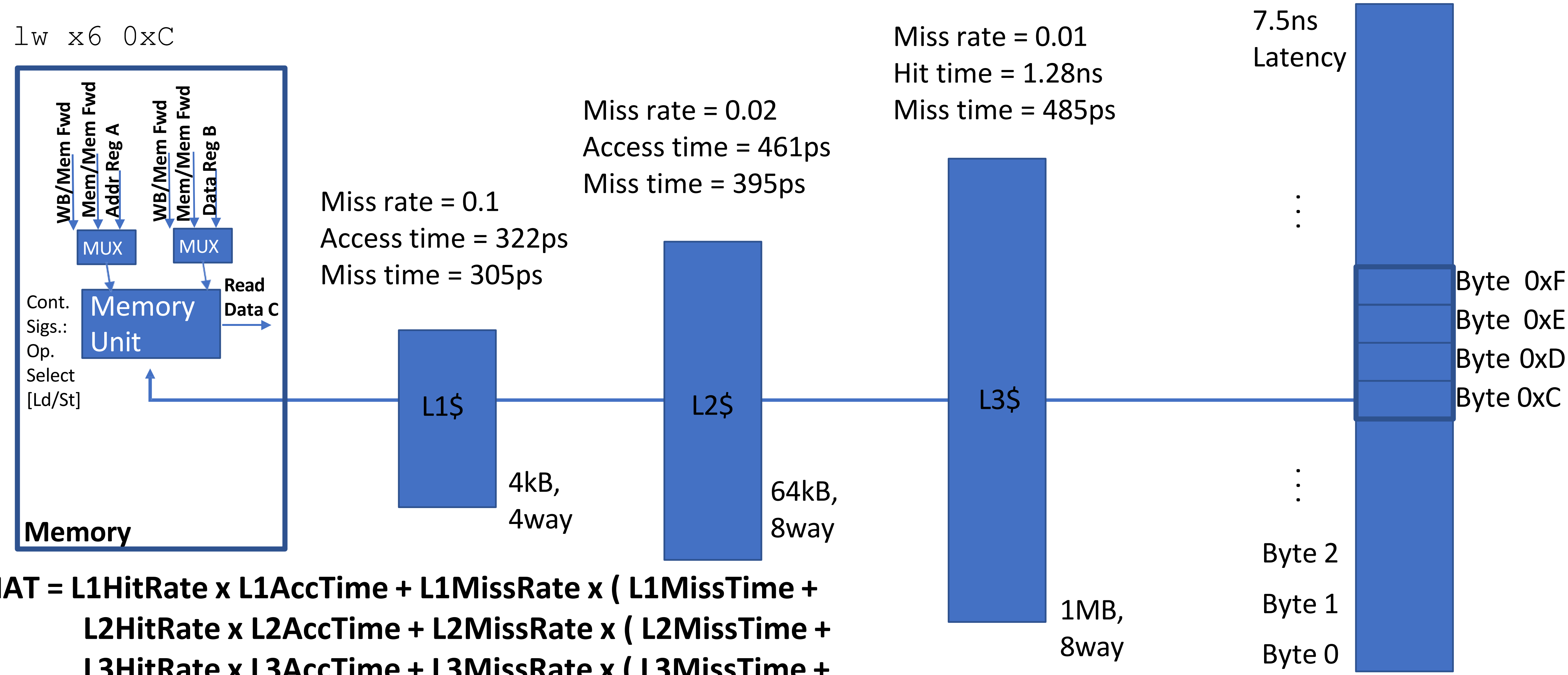
Average Memory Access Time (AMAT): Measuring the performance of a memory hierarchy



Compute the time taken by the average access based on miss rate, hit latency, and miss penalty at each level

Average Memory Access Time (AMAT): Measuring the performance of a memory hierarchy

`lw x6 0xC`



AMAT = L1HitRate x L1AccTime + L1MissRate x (L1MissTime +
L2HitRate x L2AccTime + L2MissRate x (L2MissTime +
L3HitRate x L3AccTime + L3MissRate x (L3MissTime +
DRAM Latency)))

Computing the AMAT 1/2/4/23 90% hits

Miss rate = 0.1
Access time = 322ps
(1 cycle @ 3GHz)
Miss time = 305ps

Miss rate = 0.02
Access time = 461ps
(2 cycles @ 3GHz)
Miss time = 395ps

Miss rate = 0.01
Hit time = 1.28ns
(4 cycles @ 3GHz)
Miss time = 485ps

DRAM Latency
7.5ns (CAS latency)
(23 cycles @ 3GHz)

0.322ns x 0.9 + 0.1 x (0.305ns +
0.461ns x 0.98 + 0.02 x (0.395ns +
1.28ns x 0.99 + 0.01 x (0.485ns +
7.5ns)))

AMAT in Seconds

1 x 0.9 + 0.1 x (1 +
2 x 0.98 + 0.02 x (2 +
4 x 0.99 + 0.01 x (2 +
23)))

AMAT in Cycles

Computing the AMAT

Miss rate = 0.1
Access time = 322ps
Miss time = 305ps

Miss rate = 0.02
Access time = 461ps
Miss time = 395ps

Miss rate = 0.01
Hit time = 1.28ns
Miss time = 485ps

DRAM Latency
7.5ns (CAS latency)

0.322ns x 0.9 + 0.1 x (0.305ns + 0.461ns x 0.98 + 0.02 x

All Shopping Images News Maps More Tools

About 0 results (0.52 seconds)

(0.322 ns x 0.9) + (0.1 x ((0.305 ns) + (0.461 ns x 0.98) + (0.02 x ((0.395 ns) + (1.28 ns x 0.99) + (0.01 x ((0.485 ns) + (7.5 ns))))))) =

0.3689621 nanoseconds

1 x 0.9 + 0.1 x (1 + 2 x 0.98 + 0.02 x (2 +

All Shopping Images News Videos More Tools

About 5,550,000 results (1.24 seconds)

(1 x 0.9) + (0.1 x (1 + (2 x 0.98) + (0.02 x (2 + (4 x 0.99) + (0.01 x (2 + 23)))))) =

1.20842

cycles

Computing the AMAT – 2/5/10/30 90% hits

Miss rate = 0.1
Access time = 2 cycles
Miss time = 2 cycles

Miss rate = 0.02
Access time = 5 cycles
Miss time = 5 cycles

Miss rate = 0.01
Hit time = 10 cycles
Miss time = 10 cycles

DRAM Latency
30 cycles

2 x 0.9 + 0.1 x (2 +
5 x 0.98 + 0.02 x (5 +
10 x 0.99 + 0.01 x (10 +
30))) = 2.52 cycles = **3 cycles**

AMAT in cycles

Computing the AMAT – 2/5/10/30 80% hits

Miss rate = 0.2
Access time = 2 cycles
Miss time = 2 cycles

Miss rate = 0.02
Access time = 5 cycles
Miss time = 5 cycles

Miss rate = 0.01
Hit time = 10 cycles
Miss time = 10 cycles

DRAM Latency
30 cycles

2 x 0.8 + 0.2 x (2 +
5 x 0.98 + 0.02 x (5 +
10 x 0.99 + 0.01 x (10 +
30))) = 3.04 cycles **= 4 cycles = 2 x L1 latency!**

AMAT in cycles

The ABCs of Optimizing a Cache

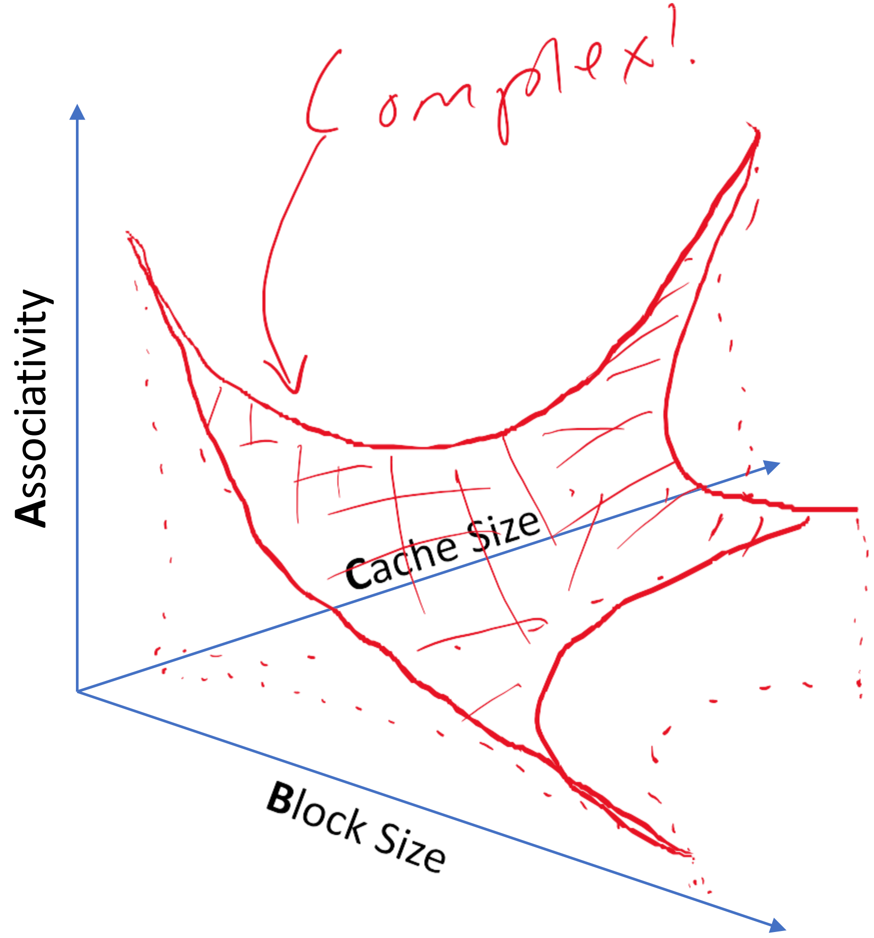
Associativity vs. Block Size vs Cache Size

Many complex inter-dependent factors determine cache performance

- Associativity
- Block Size
- Cache Size
- Replacement Policy
- Write allocation policy
- Write propagation policy

Best option depends on workload!

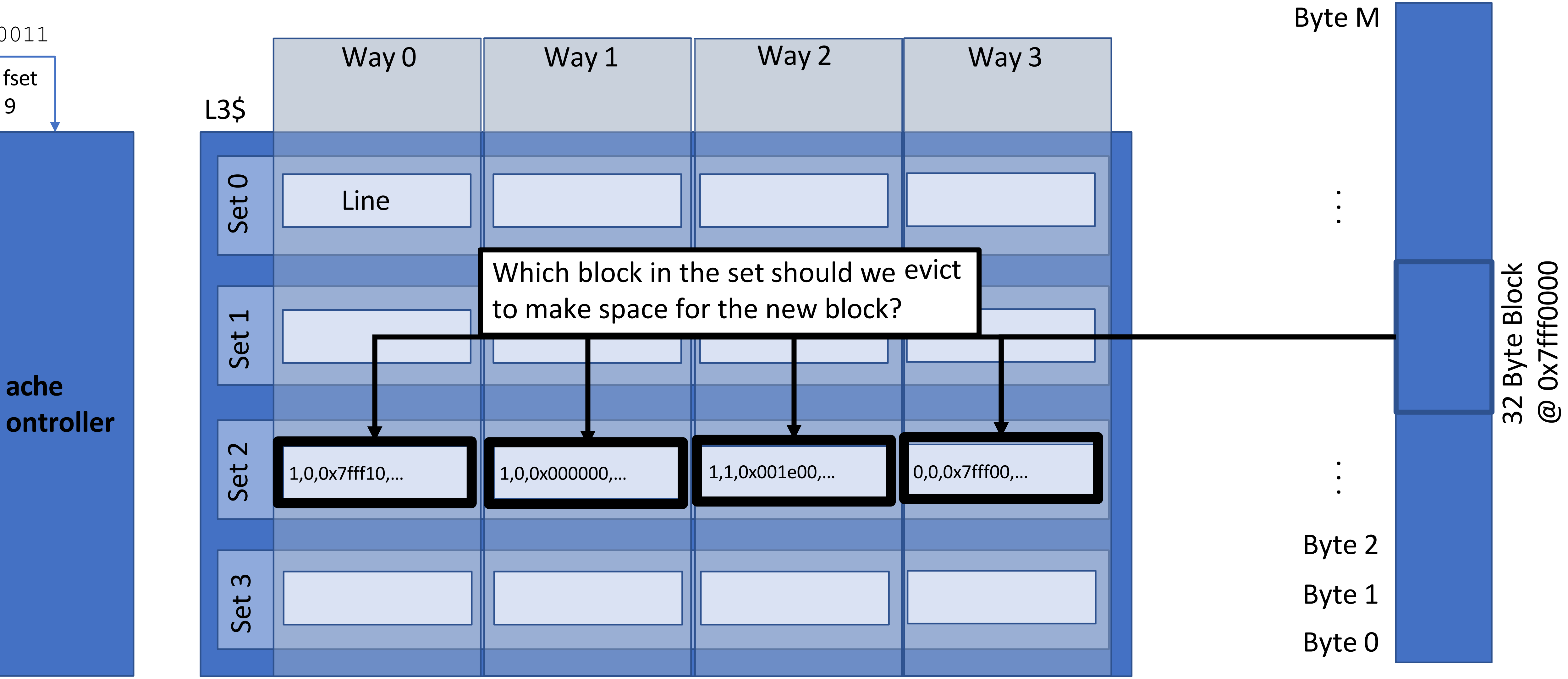
- Factors will sometimes work *against* one another, where improving degrades another. (we will study this next week)



Replacement Policies

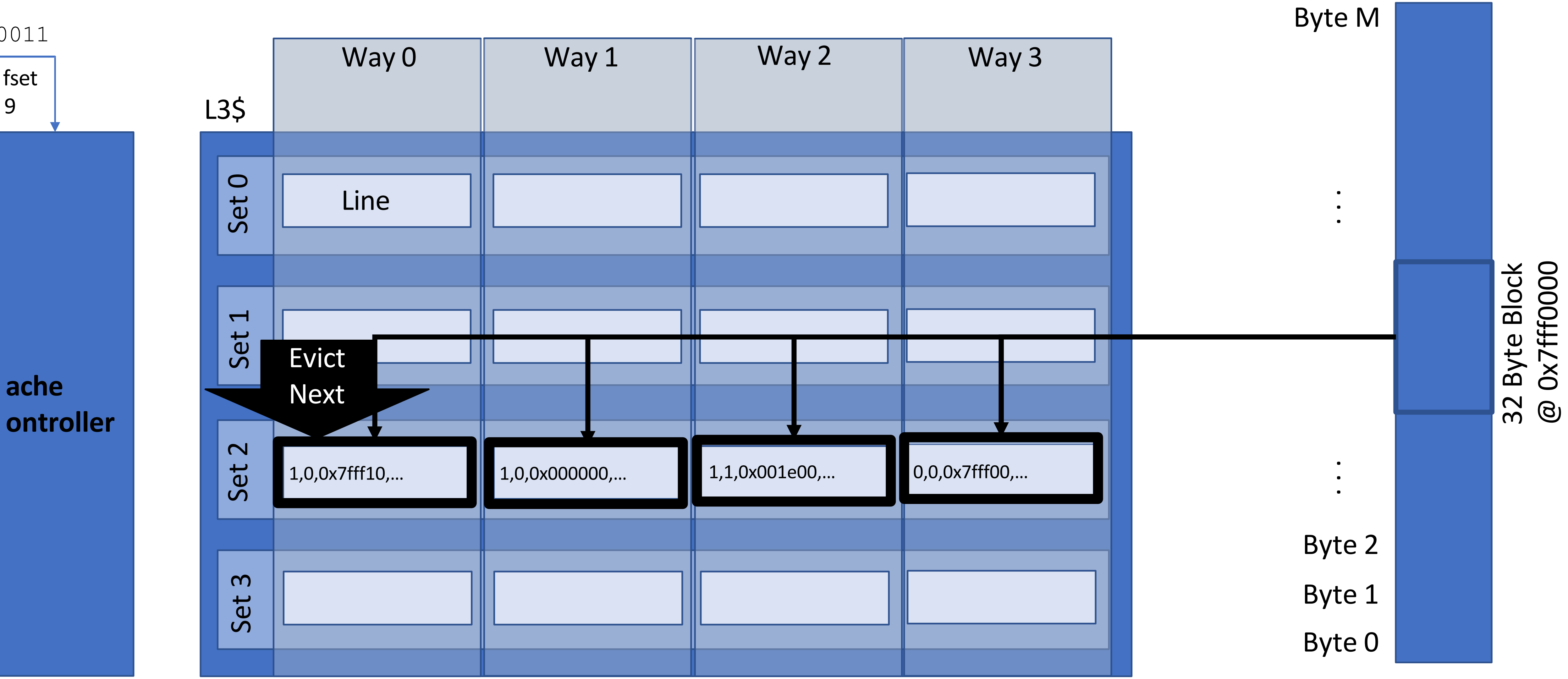
Replacement Policies

```
lb x6 0x7fff0053
```



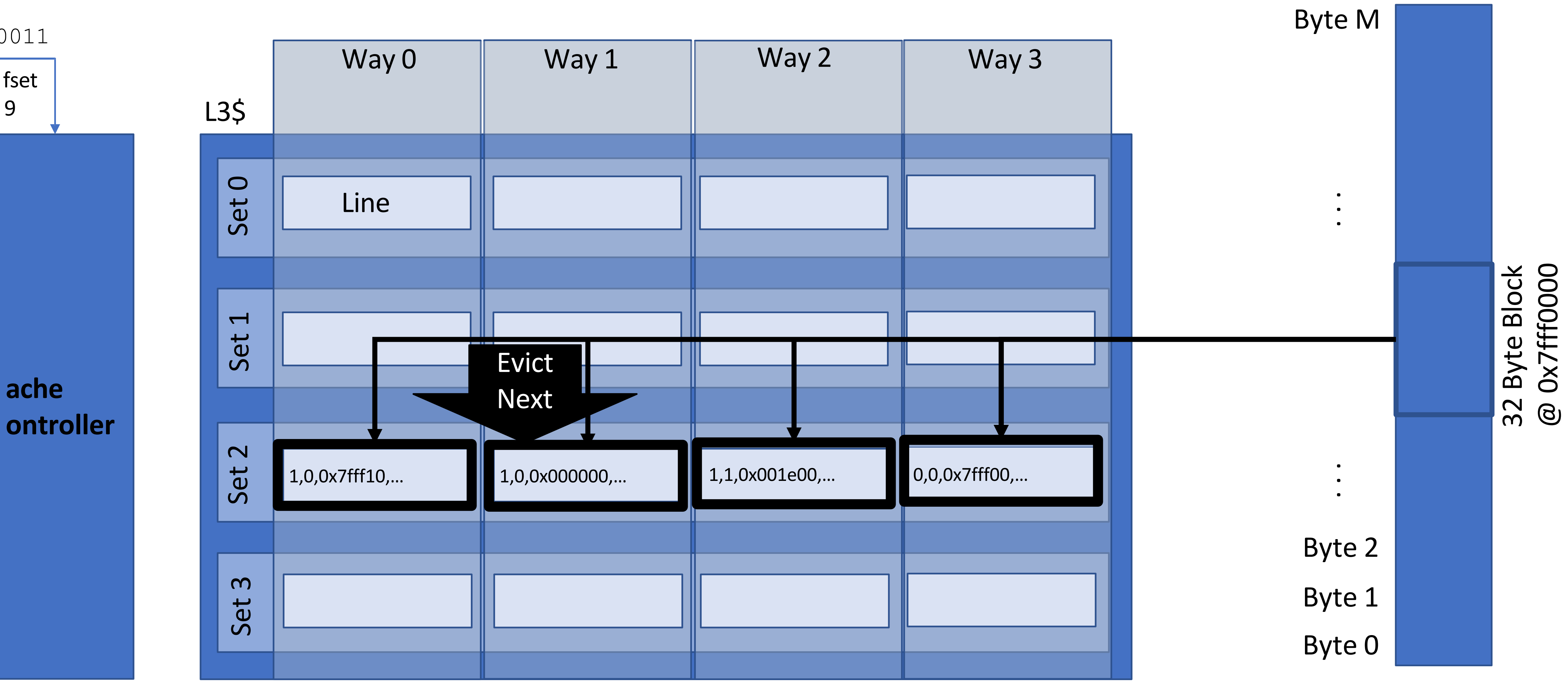
Replacement Policies – Round Robin

```
lb x6 0x7fff0053
```



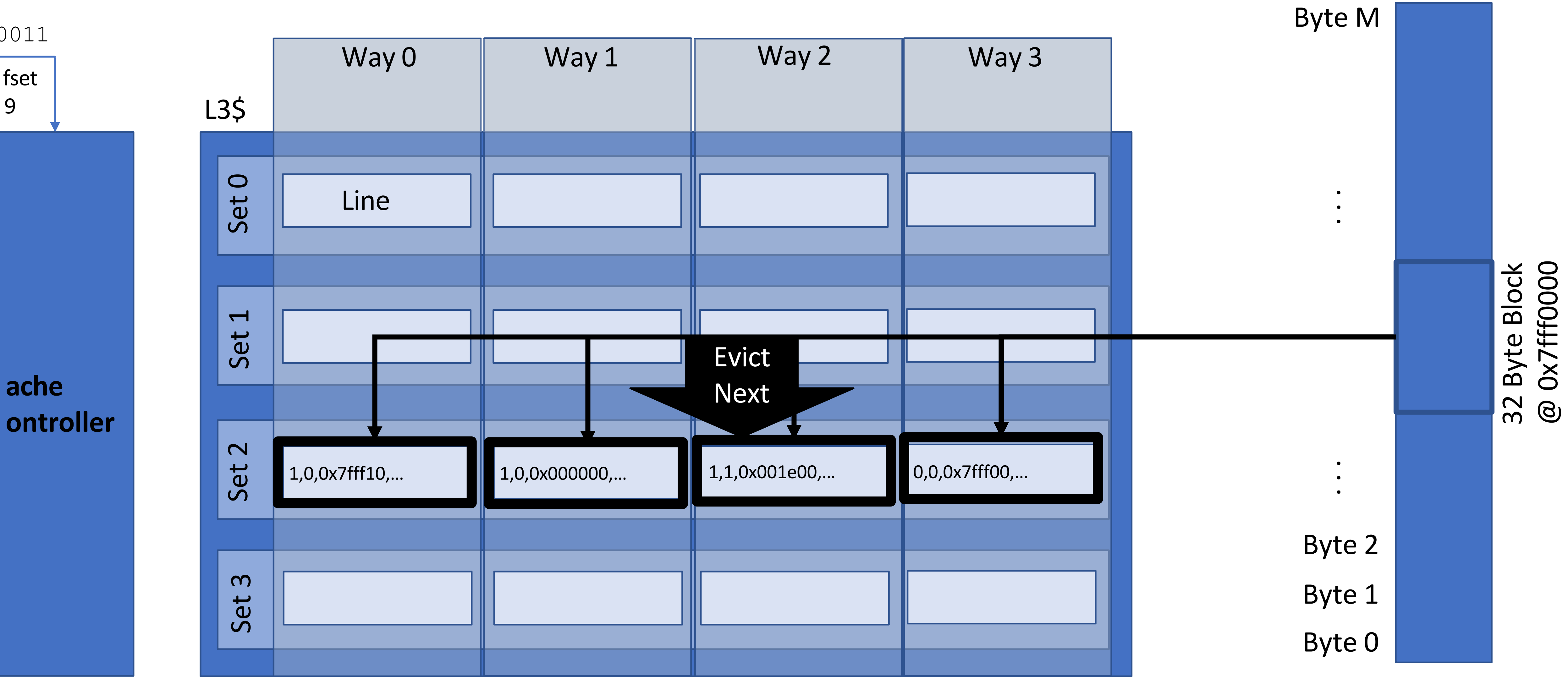
Replacement Policies – Round Robin

```
lb x6 0x7fff0053
```



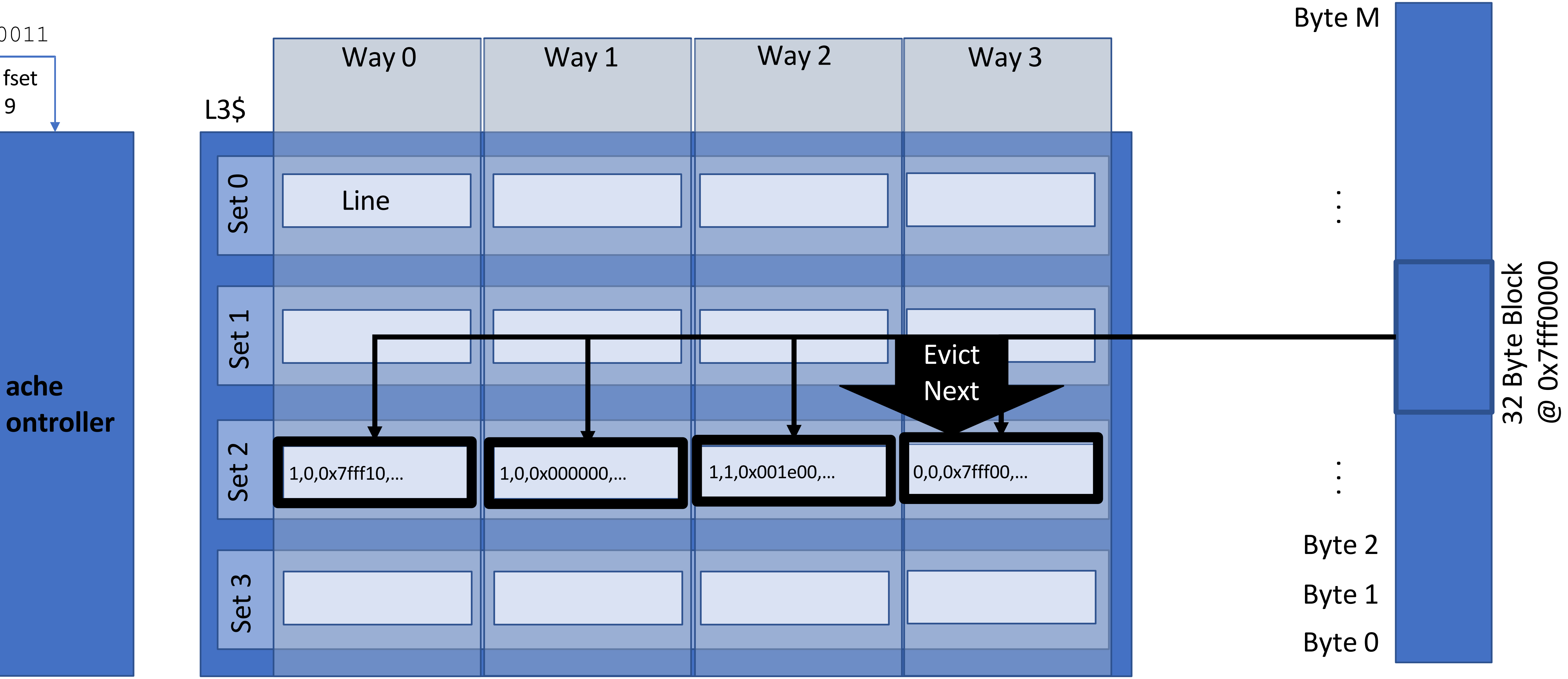
Replacement Policies – Round Robin

```
lb x6 0x7fff0053
```



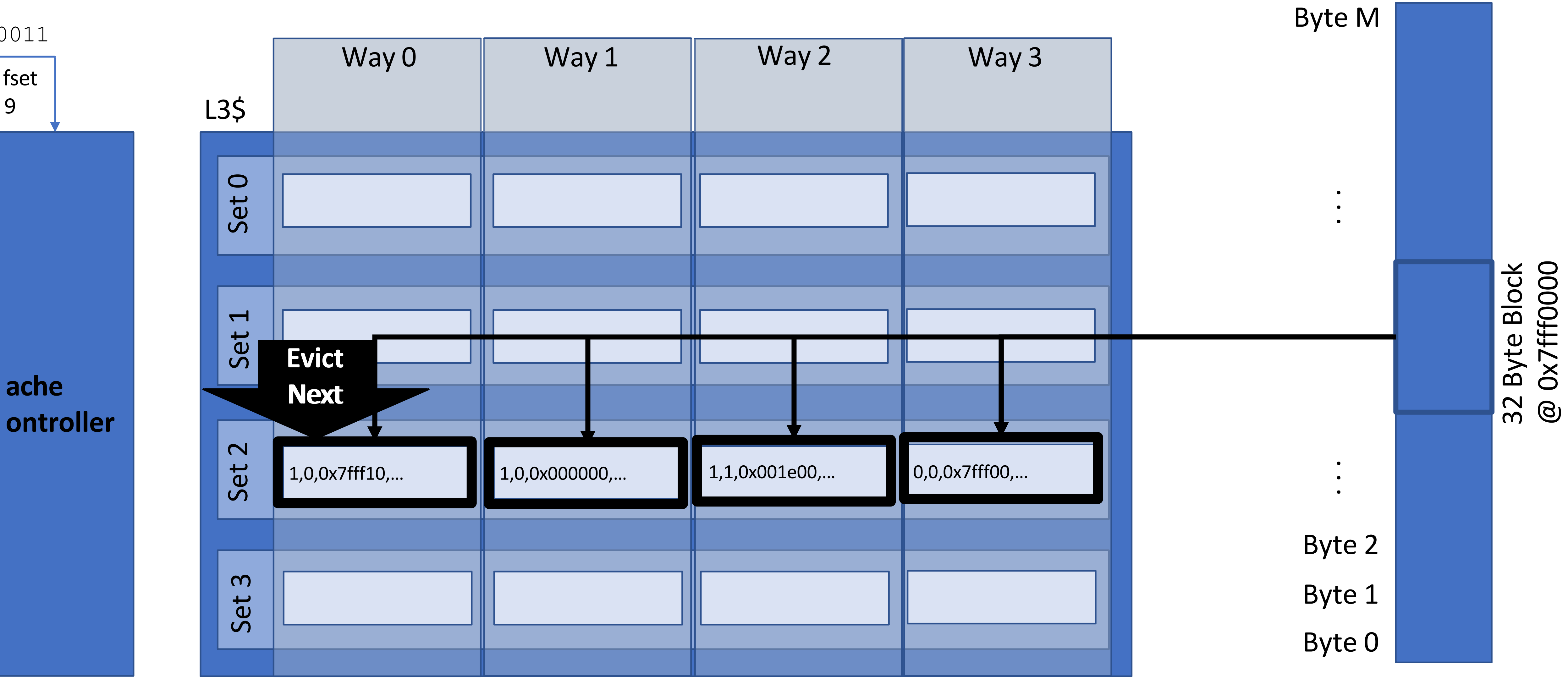
Replacement Policies – Round Robin

```
lb x6 0x7fff0053
```

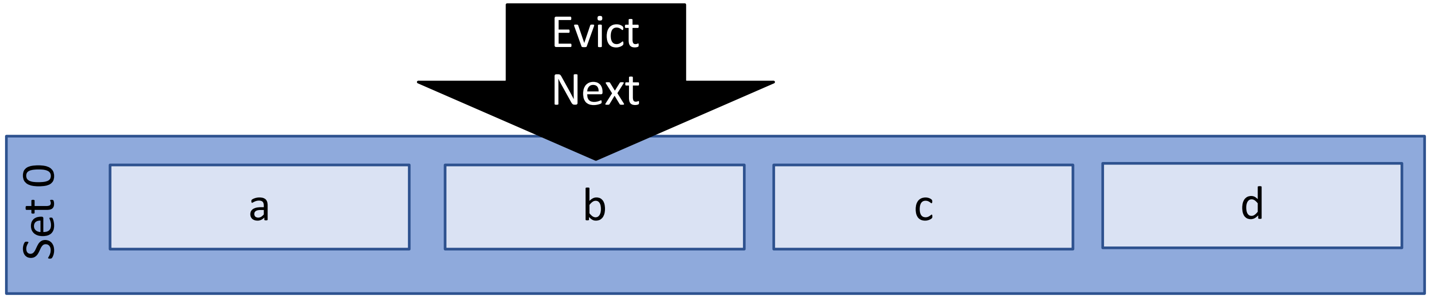


Replacement Policies – Round Robin

```
lb x6 0x7fff0053
```



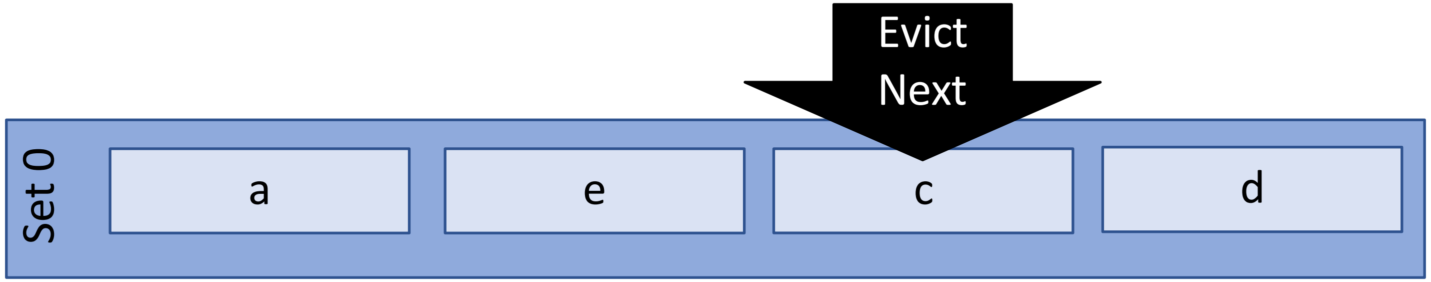
Replacement Policies – Round-Robin Analysis




```
lb x6 0xe
lb x6 0xb
lb x6 0xc
lb x6 0xd
lb x6 0xa
```

Advantage: Simple to implement and understand
Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed…

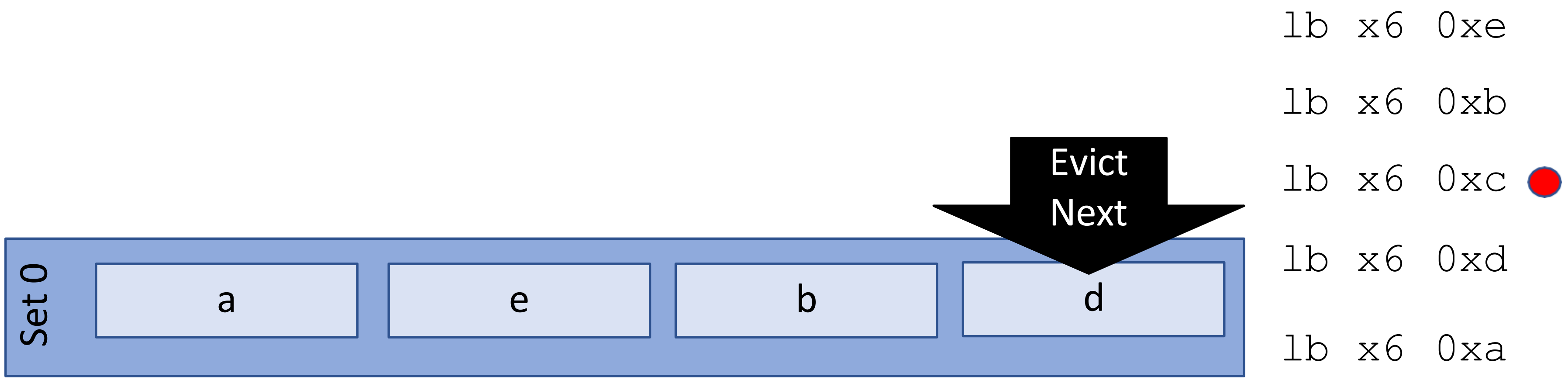
Replacement Policies – Round-Robin Analysis




```
lb x6 0xe
lb x6 0xb
lb x6 0xc
lb x6 0xd
lb x6 0xa
```

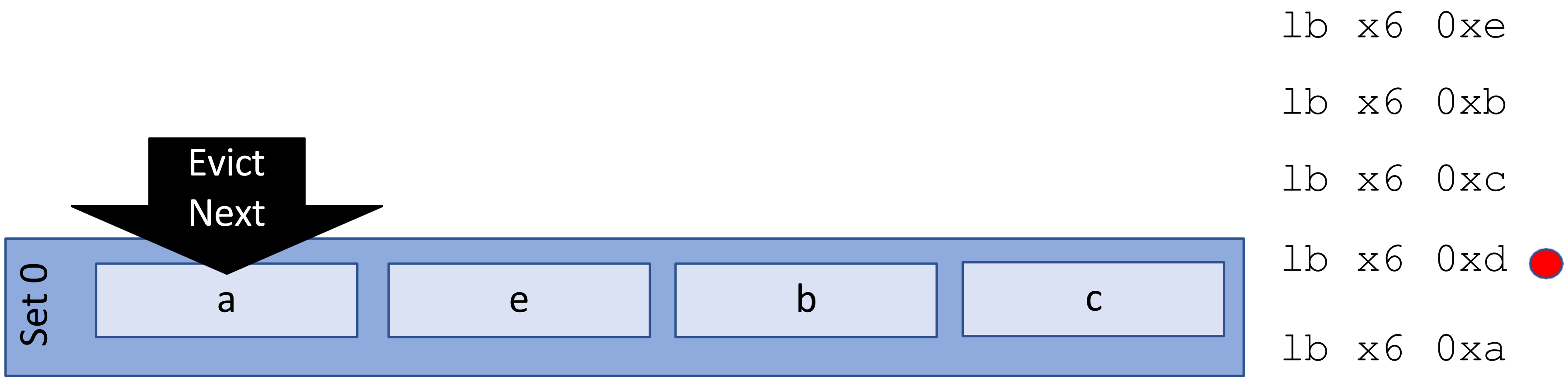
Advantage: Simple to implement and understand
Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed…

Replacement Policies – Round-Robin Analysis



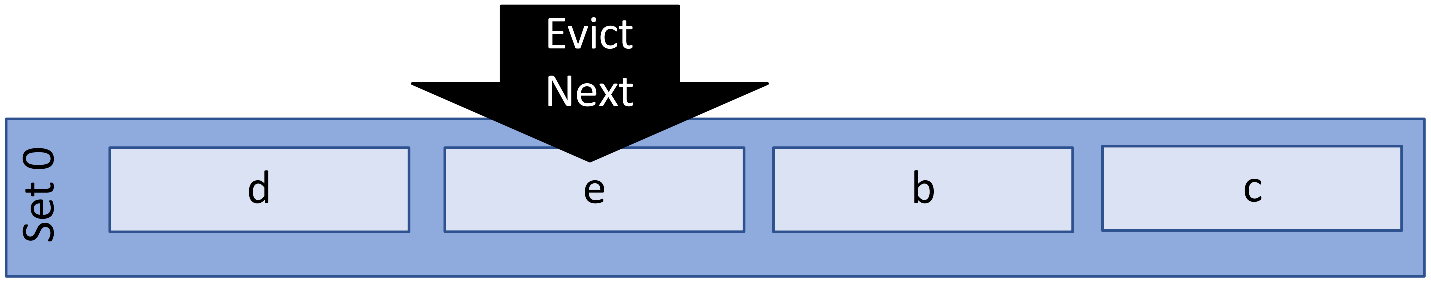
Advantage: Simple to implement and understand
Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed…

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…

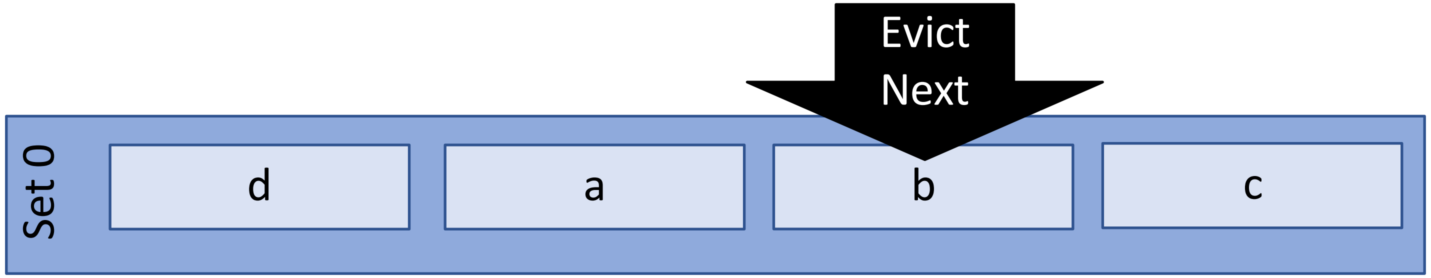
Replacement Policies – Round-Robin Analysis




```
lb x6 0xe
lb x6 0xb
lb x6 0xc
lb x6 0xd
lb x6 0xa
```

Advantage: Simple to implement and understand
Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed…

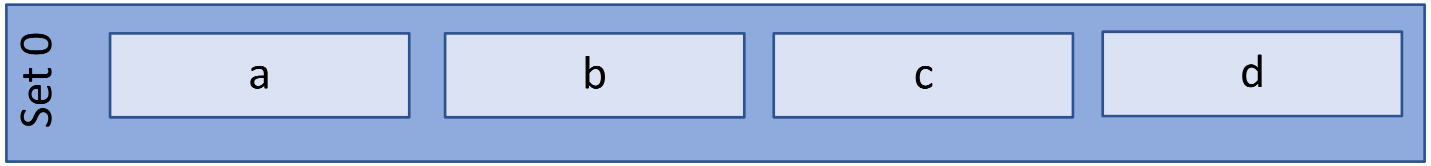
Replacement Policies – Round-Robin Analysis




```
lb x6 0xe
lb x6 0xb
lb x6 0xc
lb x6 0xd
lb x6 0xa
```

Advantage: Simple to implement and understand
Disadvantage: Hopefully the next to evict isn't going to be the next to be accessed…

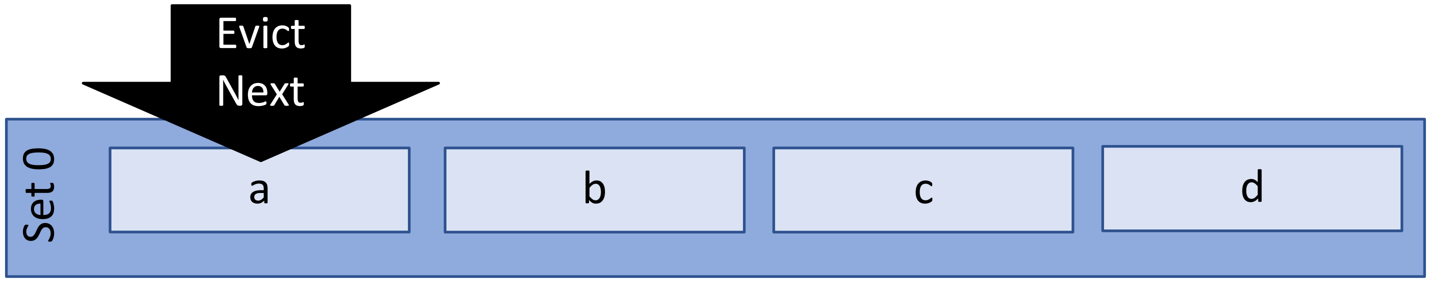
Minimum Number of Misses?




```
lb x6 0xe

lb x6 0xb

lb x6 0xc

lb x6 0xd

lb x6 0xa
```

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

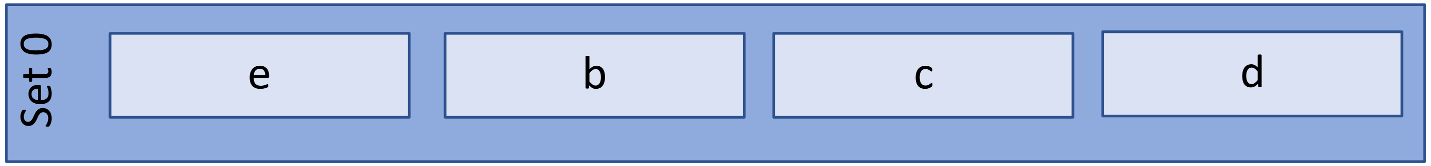
Minimum Number of Misses?




```
lb x6 0xe
lb x6 0xb
lb x6 0xc
lb x6 0xd
lb x6 0xa
```

When are we going to re-use cached data?

```
lb x6 0xe    Miss
lb x6 0xb    Hit
lb x6 0xc    Hit
lb x6 0xd    Hit
lb x6 0xa    Miss
```



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?

What did we just learn?

- Memory has a high access cost; memory hierarchy mitigates that cost
- Caches make locality exploitable to optimize for data reuse
- Review of the basics of cache operation, address decomposition, set associative caches
- Miss types
- The costs of associativity & tag storage arrays
- What to do about writes?
- The replacement problem

What to think about next?

- More caches (next time)
  - Replacement from the ground up
  - Caching optimizations: victim caches, write buffers & lockup-free caches, prefetching, way partitioning, banking & bank conflicts
  - Scratchpads vs. Caches & their relation to the HW/SW interface
- Performance Evaluation (next next time)
  - Design spaces, Pareto Frontiers, and design space exploration
- Miscellaneous (micro)architectural tricks & optimizations (future)
  - Vector processors, SIMD/SIMT, dataflow

Replacement Policies – Not Most Recently Used

Replacement Policies - PLRU

Replacement Policies - SRRIP

Replacement Policies – Belady Optimal

Replacement Policies – Hawkeye

Victim Caches

Banking & Bank Conflicts

Bank Mapping Function

NUCA, SNUCA, DNUCA, RNUCA

Cache Partitioning

Prefetching

Non-temporal Stores

Scratchpads

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