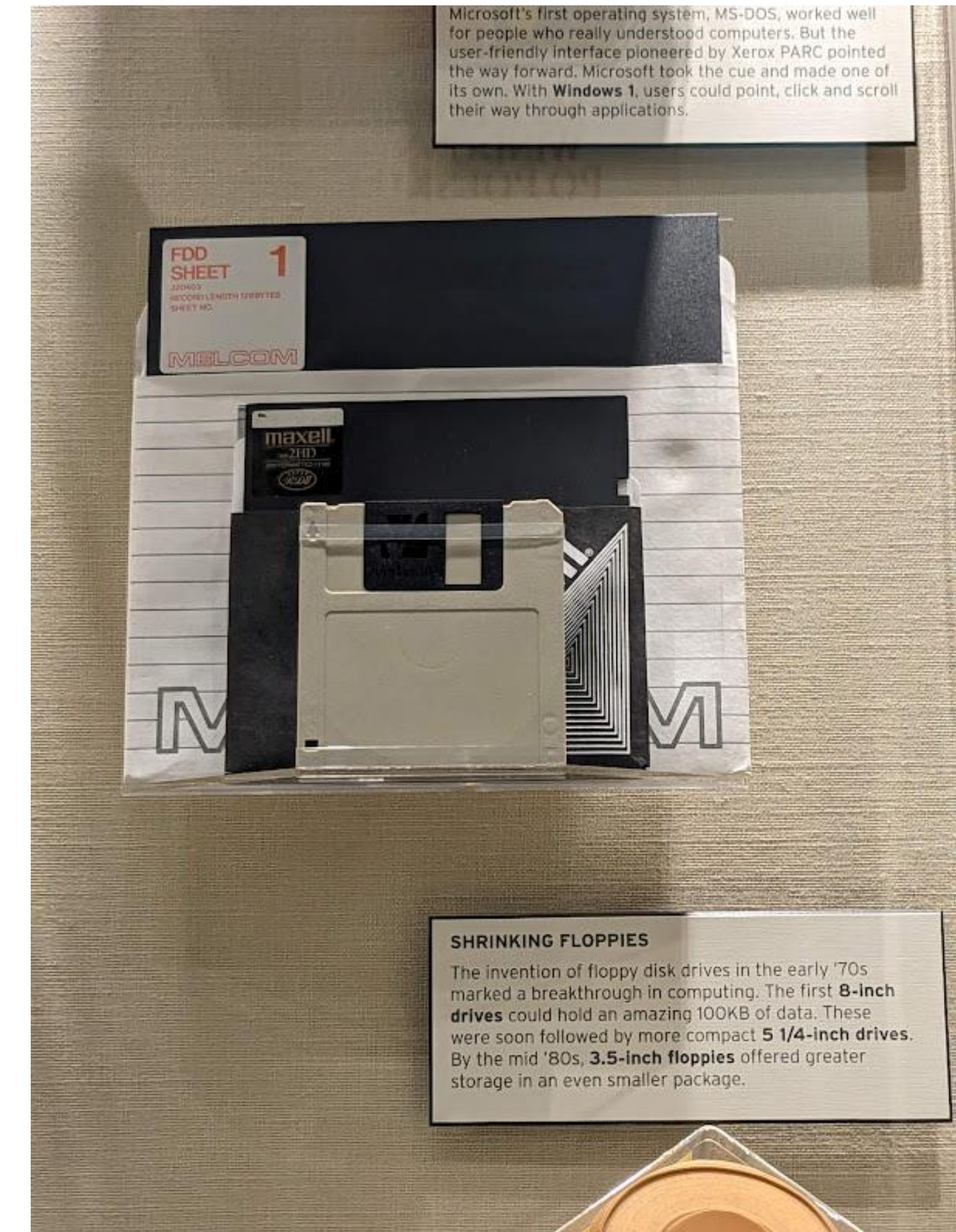


CSE240A – Lecture 9 (caches part 1)

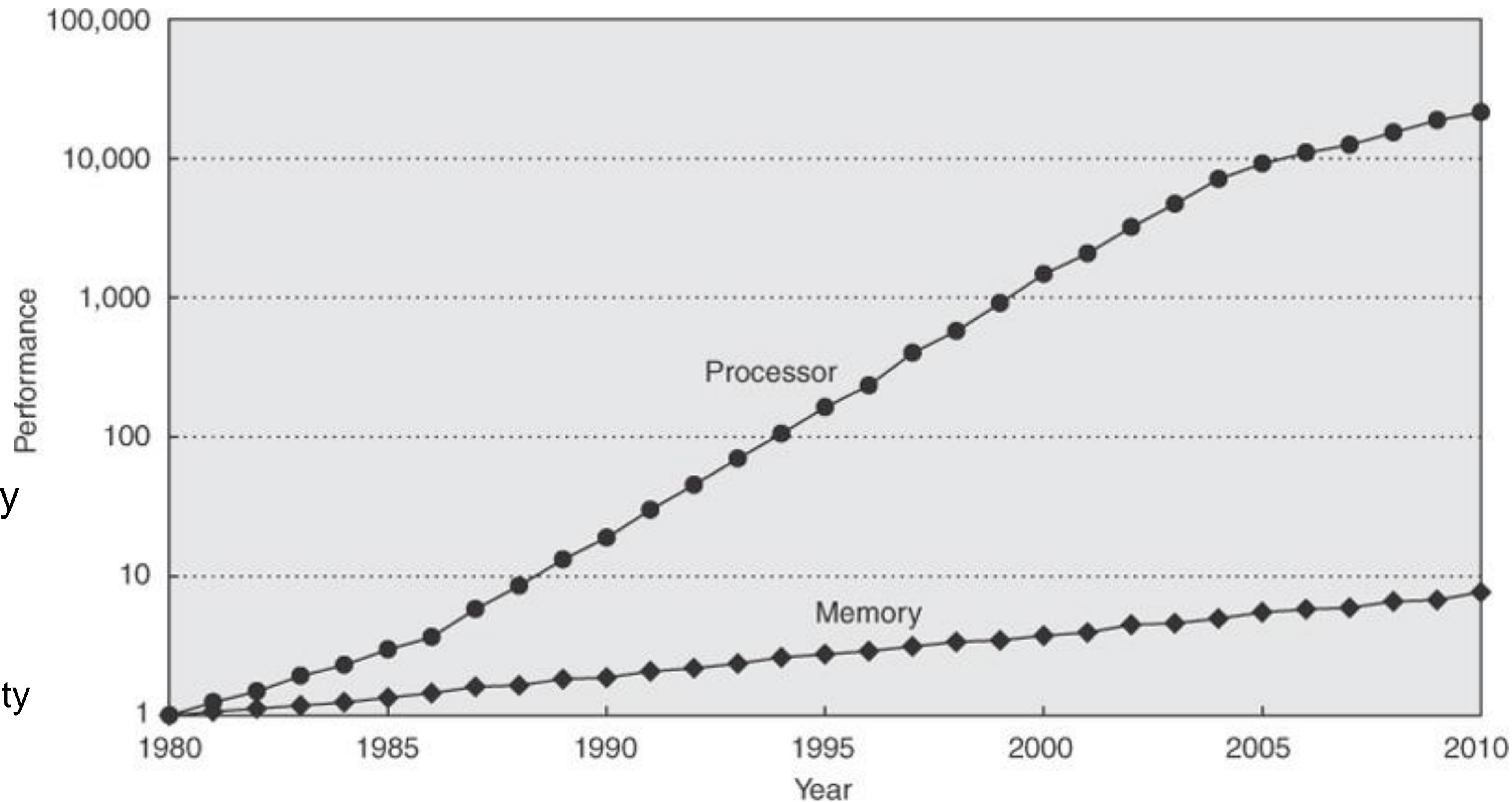


CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=635178>



Processor Memory Trends

- log graph – gap is large
- 1980 – no integrated cache
- 2000 – RM7000
Integrated L1's and L2s
- Computation vs Capacity
- More xtors and different outcomes
 - compute – more capability
 - memory – more storage



© 2007 Elsevier, Inc. All rights reserved.

Why Cache

- Compute wants fast memory
- Physics – bigger memory is slower
 - Bigger memory, longer wires
 - smaller transistors to get higher density (e.g. DRAM vs SRAM)
- Architects create hierarchy of memories to tradeoff speed and size
 - Amazon EC t2.micro : `cat /proc/cpuinfo | grep model`
model name : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
 - Xeon e5 (server class CPU), v3 – Haswell
12 cores, L1: 32K 8 way; L2: 256k 8 way; L3: 30MB 20 way shared

	registers	L1	L2	L3 – shared	DDR4 DRAM
latency	1 cyc	4 cyc	12 cyc	~60 cyc	~300 cyc
~bw		70-100 GB/sec	70 GB/sec	~25 GB/s	~15 GB/s

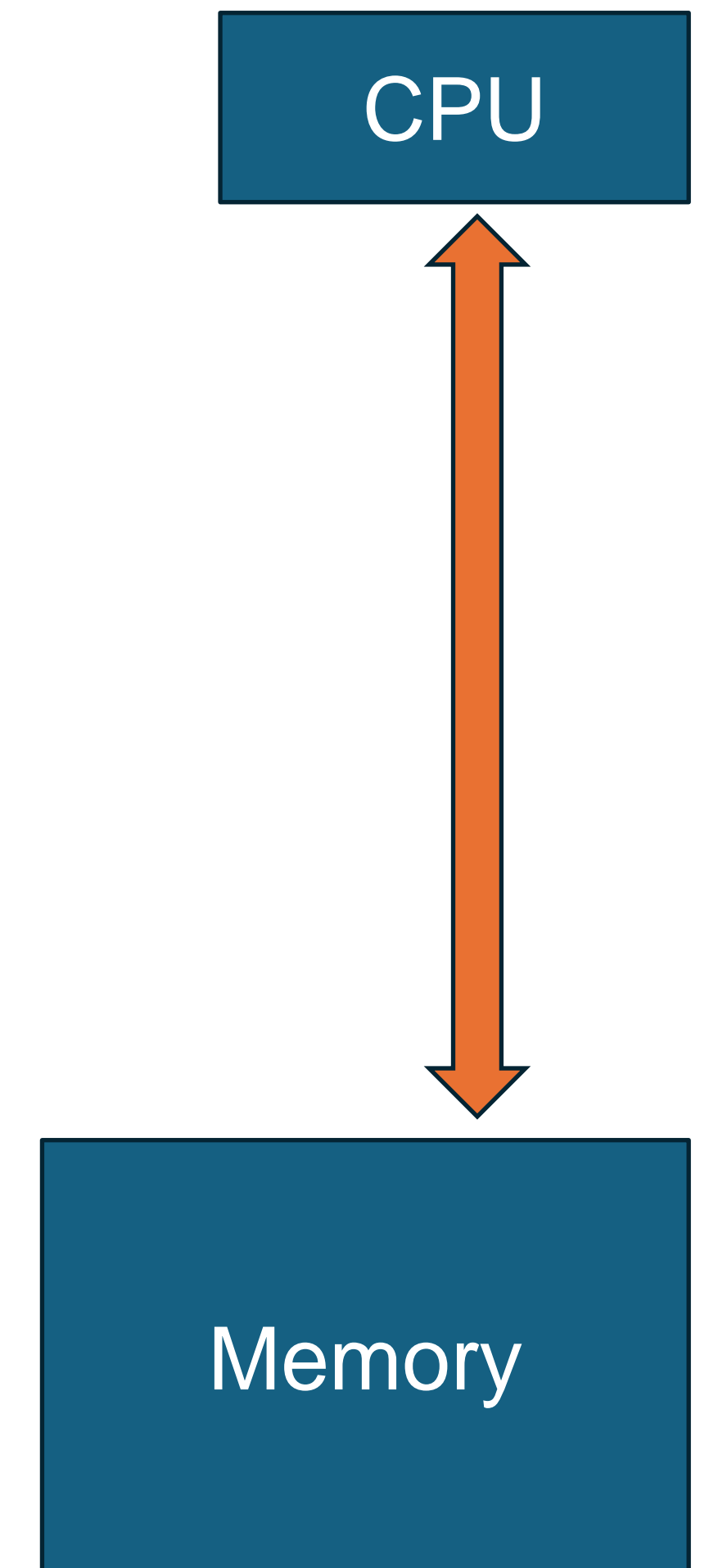
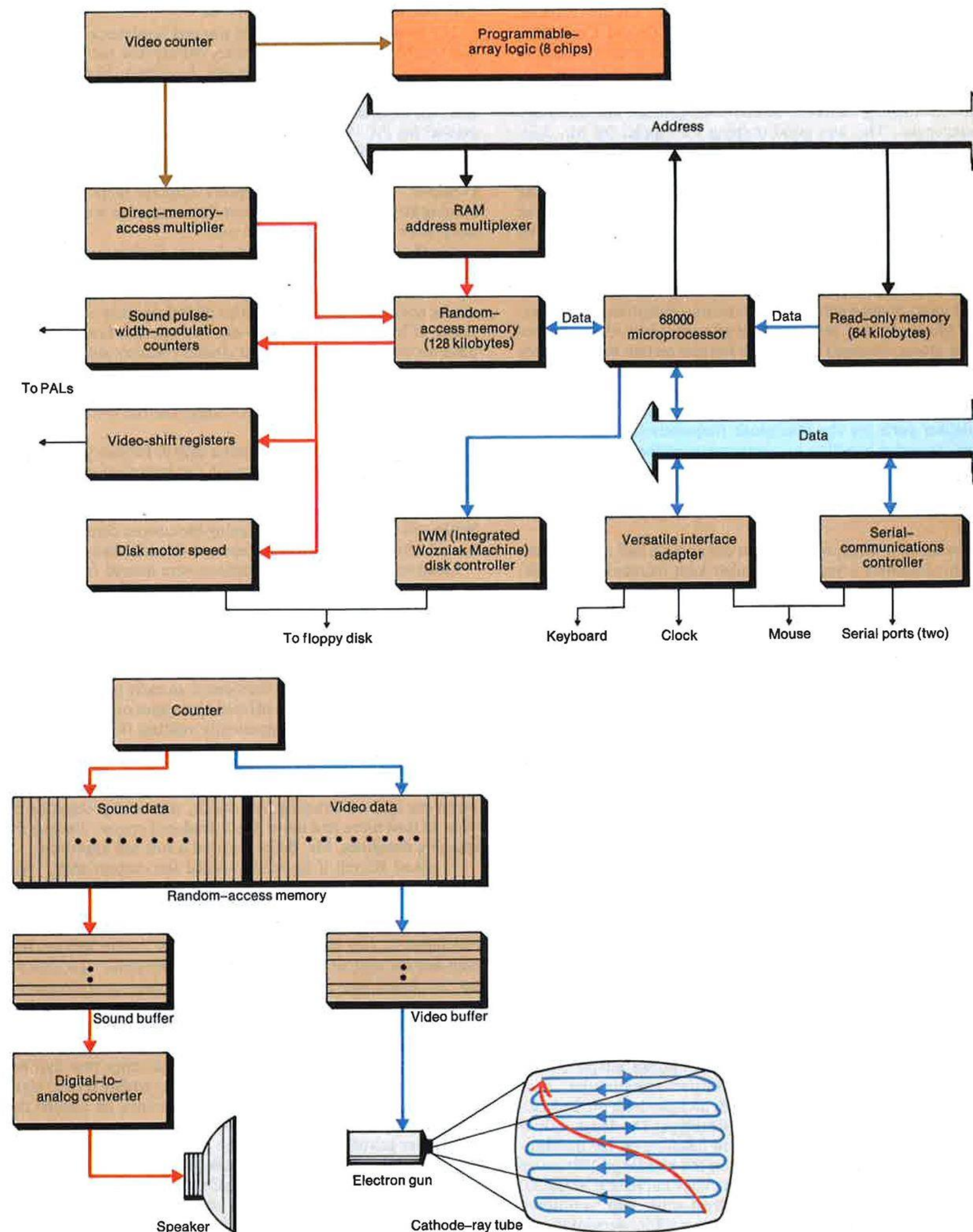
<http://www.7-cpu.com/cpu/Haswell.html>

Evolution of Caches – 1984 (Apple Macintosh)

- No caches.
- 68K microprocessor talks directly to ROM and DRAM

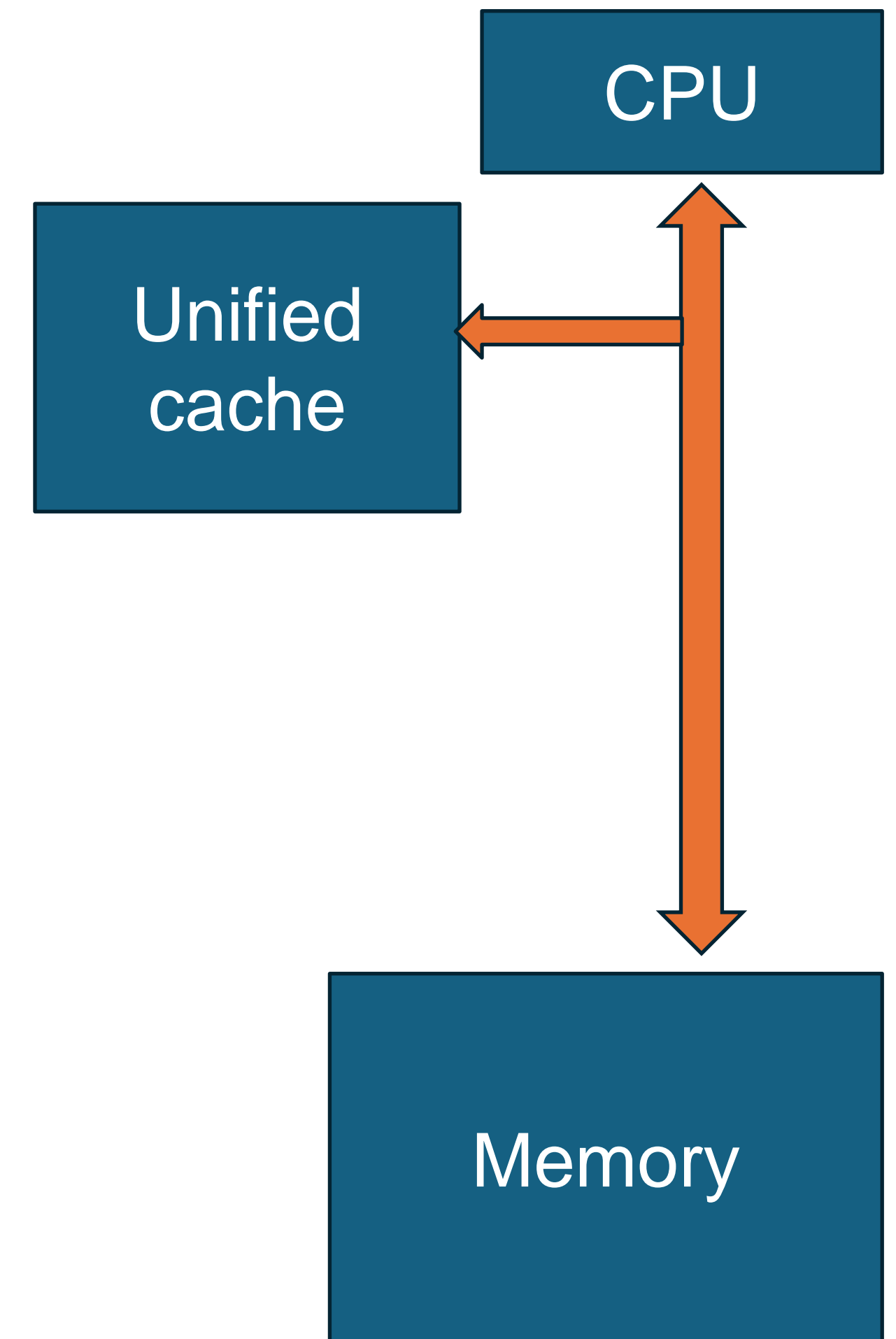


Sailko, CC BY 3.0 <<https://creativecommons.org/licenses/by/3.0/>>, via Wikimedia Commons



Evolution of Caches – (1985- Sun 2)

- 68K microprocessor talks to cache and memory
- What goes into the cache?
- Memory Hierarchy



Evolution of Caches (AMD Zen 4)

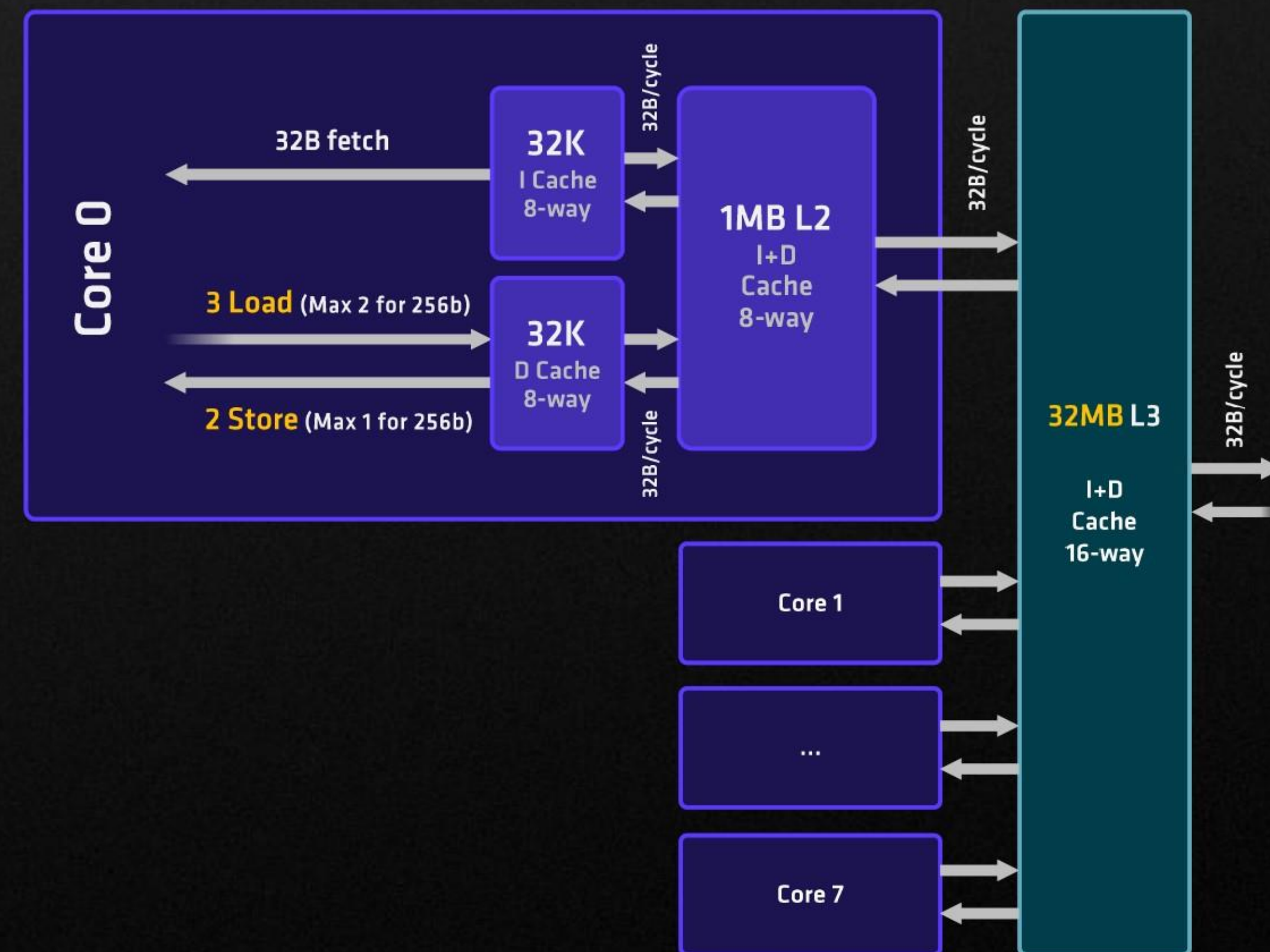
“Zen 4” Cache Hierarchy

New for Zen 4:

- Fast private 1M L2 cache
- More outstanding misses supported from L2 to L3 per core
- More outstanding misses supported from L3 to memory
- Improved L3 and L2 miss BW
- Higher BW enables prefetch improvements

Zen 3 and Zen 4:

- L3 is filled from L2 victims (i.e. mostly exclusive)
- L2 tags duplicated in L3 for probe filtering and fast cache transfer
- L3 shared among all 8 cores in the complex



Memory Locality

- *Memory locality* is the principle that future memory accesses are *near* past accesses.

- Memory Hierarchies take advantage of two types of memory locality

- temporal locality – something used recently is likely to be used again



- spatial locality – something used recently is likely to be near (in memory) something to be used soon.

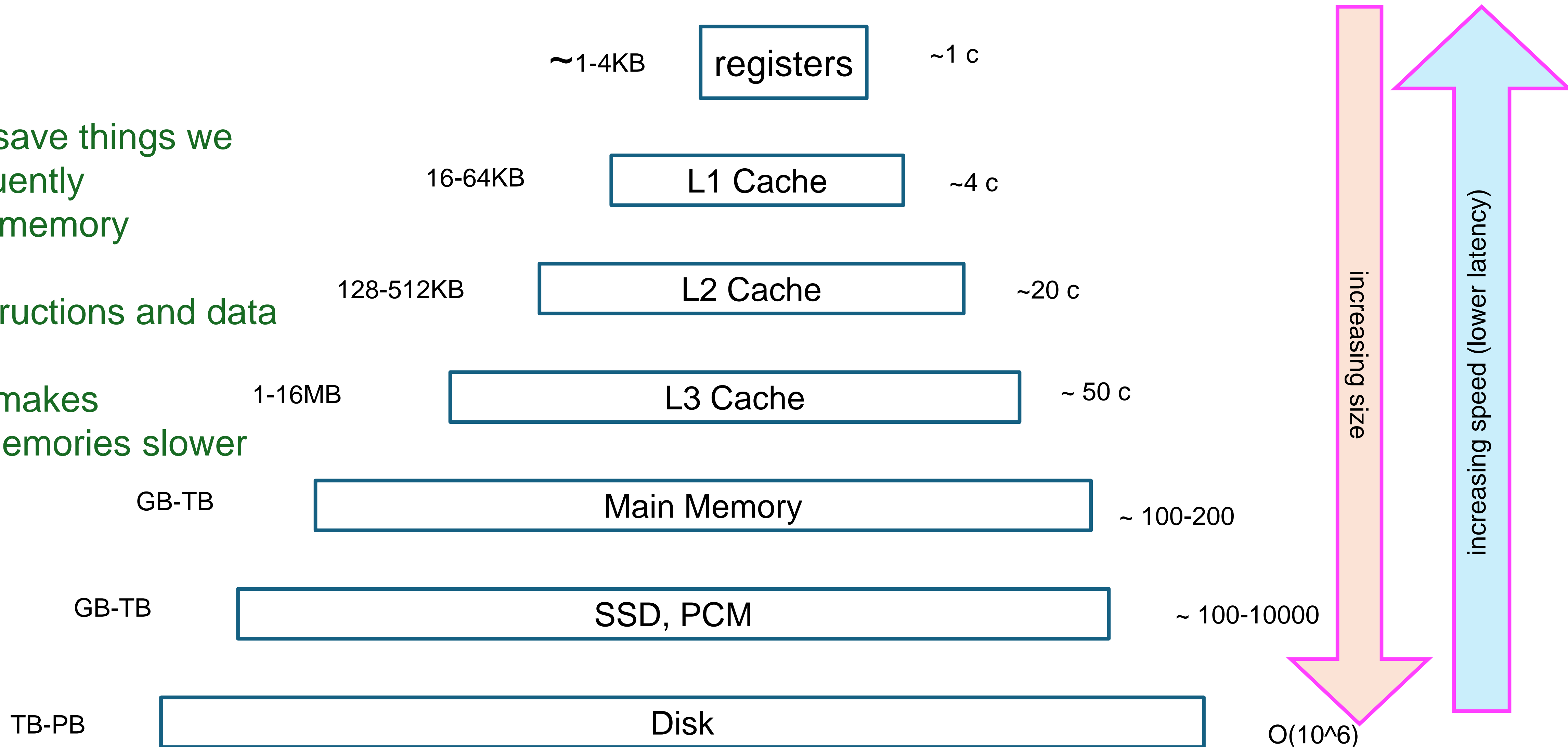


Memory System Hierarchy

Want to save things we
use frequently
In faster memory

Both instructions and data

Physics makes
Larger memories slower



Data Latency – real world perspective : ns -> m

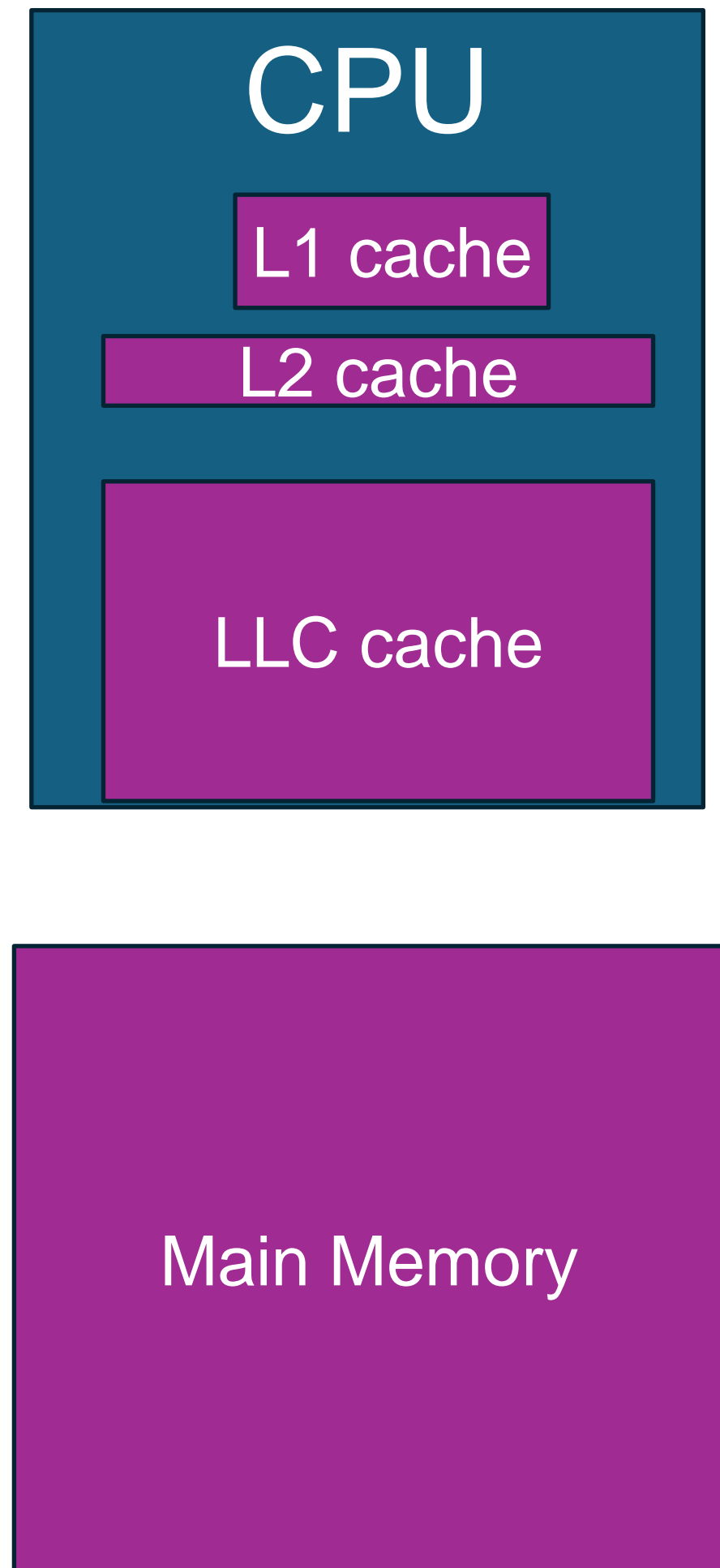


Layer	Latency (ns)	Distance (m)	From UCSD Center 115
Register	1	1	
Cache	10	10	
DRAM	100	100	
Fabric DRAM	400	400	
3D Xpoint	1000	1 km	CSE building
Flash SSD	100000	100 km	UC Irvine
HDD	5000000	5,000 km	Boston

Source – Michael Woodacre, HPE, cse240b spring 2017 lecture-11

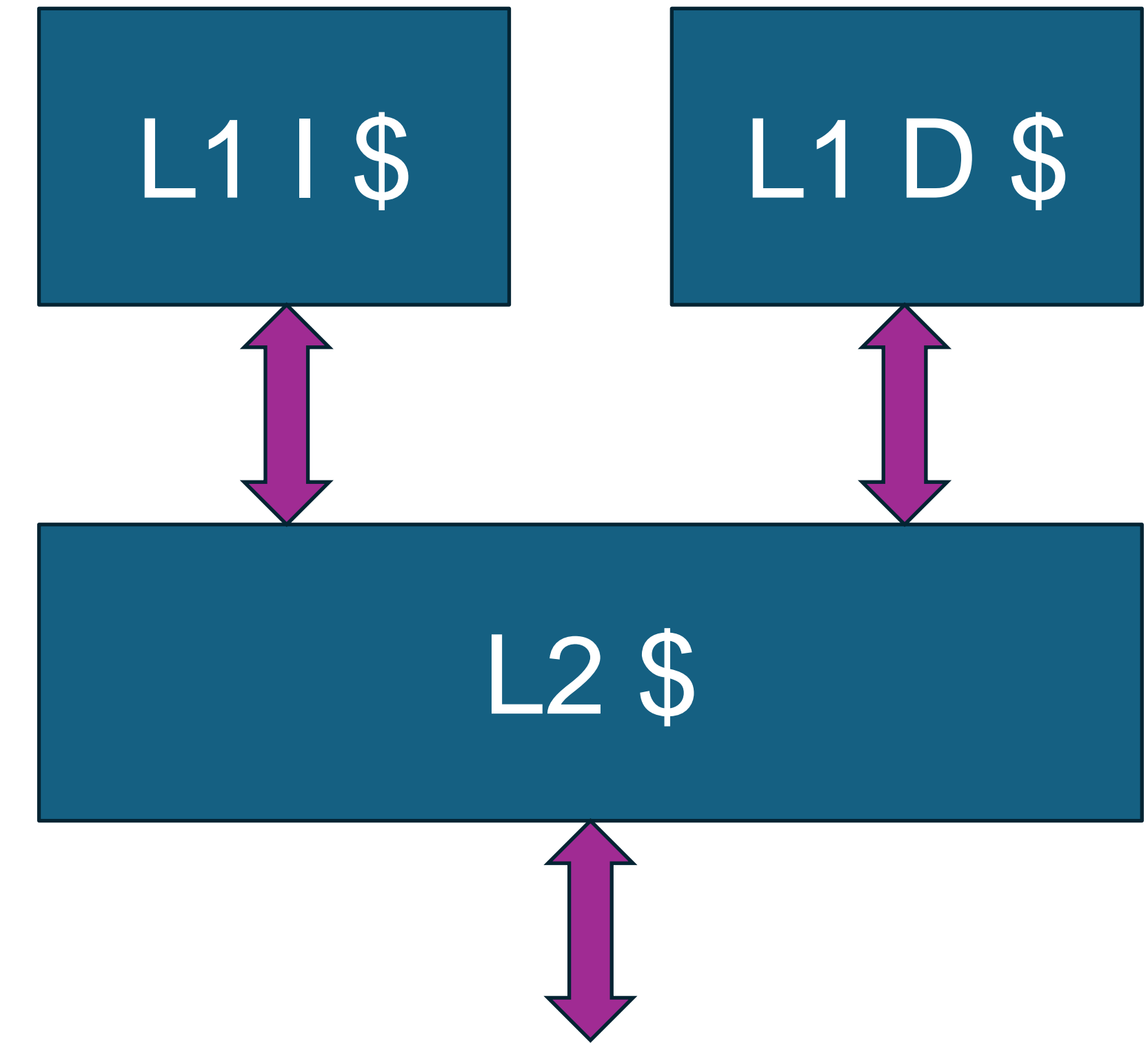
Cache Fundamentals

- **Cache Hit** – what we are looking for is in the cache
- **Cache Miss** – cache does not contain what we are looking for
- **Hit time** – how long does access take if cache hit (latency)
- **Miss time** – how long does it take to refill cache and get data to CPU (latency)
- **Hit Ratio** - # of hits / # of accesses
- **Miss Ratio** (1 – Hit Ratio)
- tip: Hit ratio and Miss ratio can be deceiving.
 - what if # of accesses is very low
 - **MPKI** – misses per thousand instructions



Cache Terminology

- **Cache Line Size** or **Cache Block Size** - basic cache unit (bytes or words)
- **Instruction cache**
- **Data cache**
- **Unified cache**
- 3 C's of misses
 - **Cold (compulsory)**
 - **Capacity**
 - **Conflict**



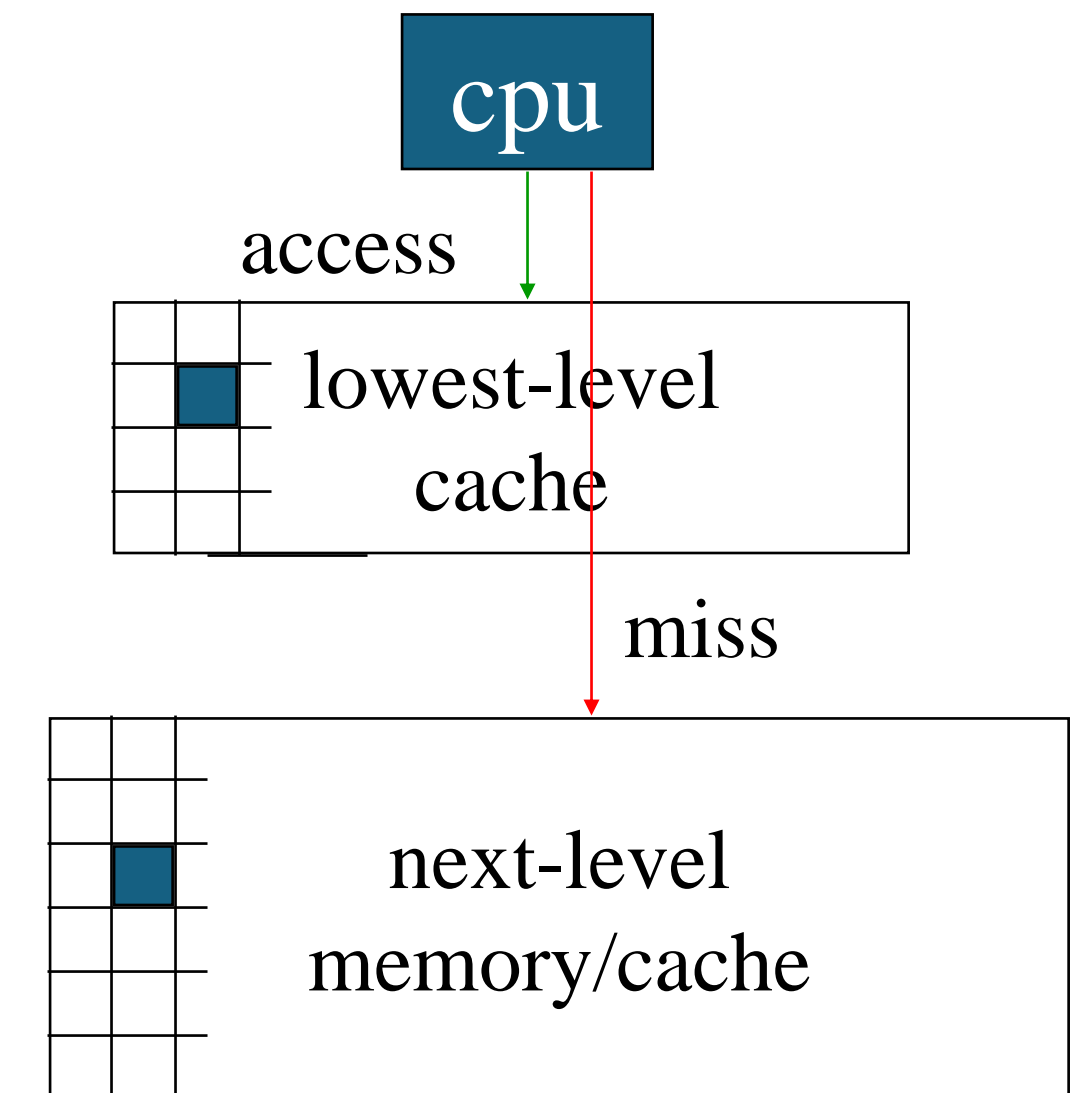
Caching Issues

On a memory access -

- How do I know if this is a hit or miss?

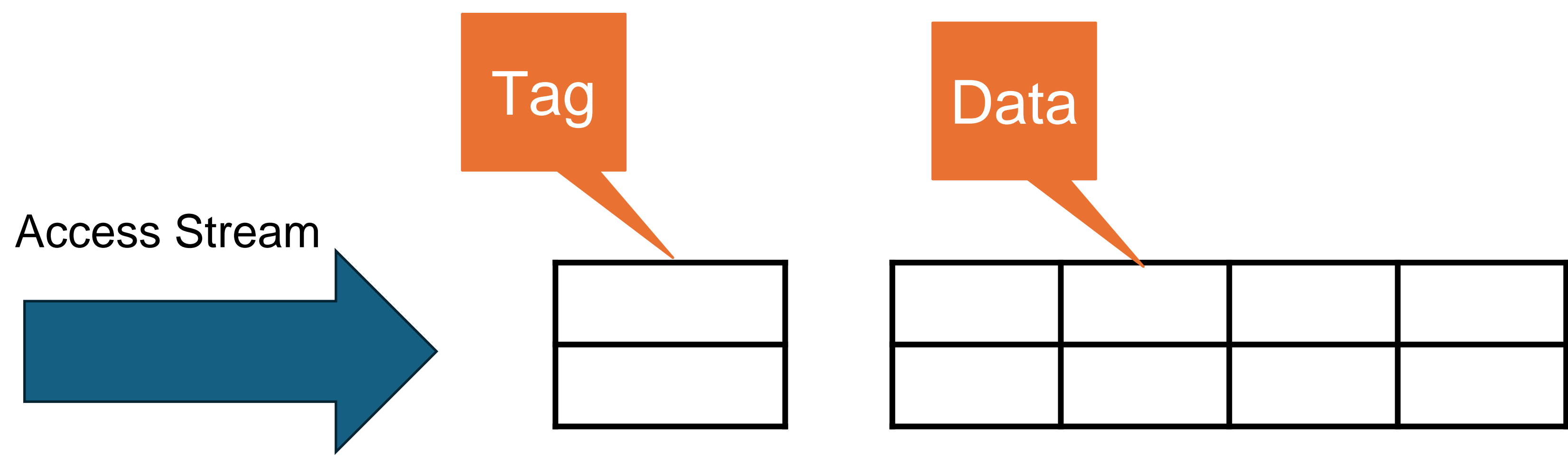
On a cache miss -

- where to put the new data?
- what data to throw out?
- how to identify (remember) what data these are?



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

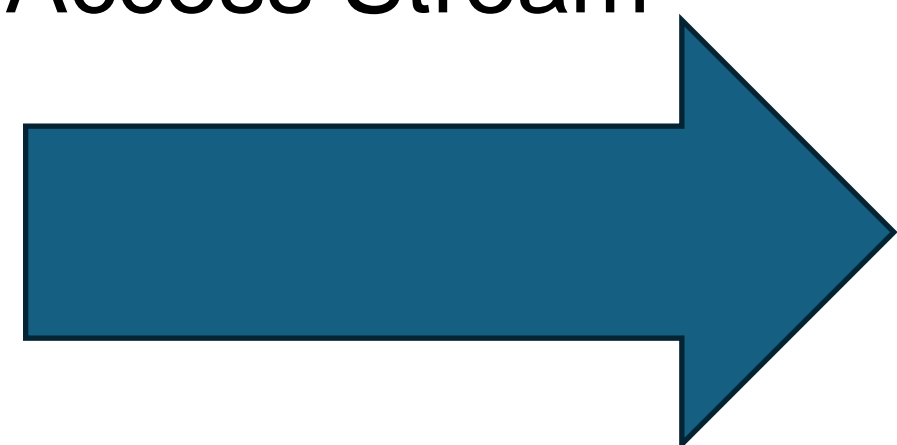


0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P

A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

Access Stream

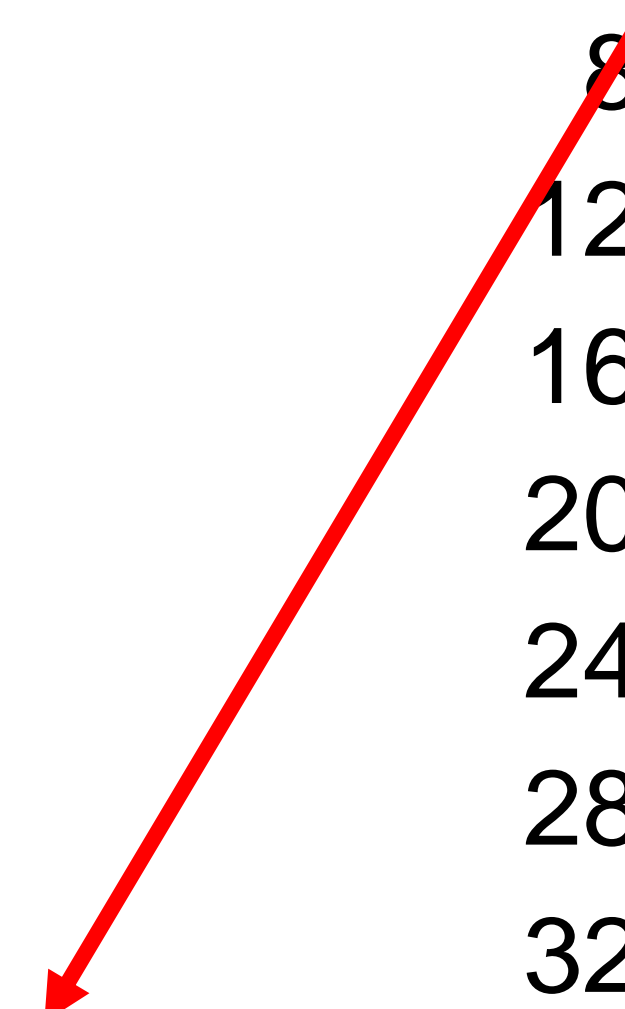


0

A	C	E	F

4 (miss)

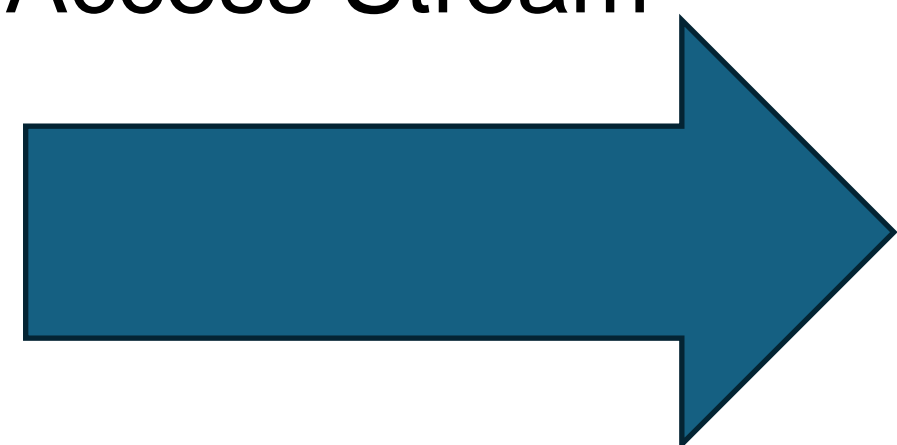
0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

Access Stream

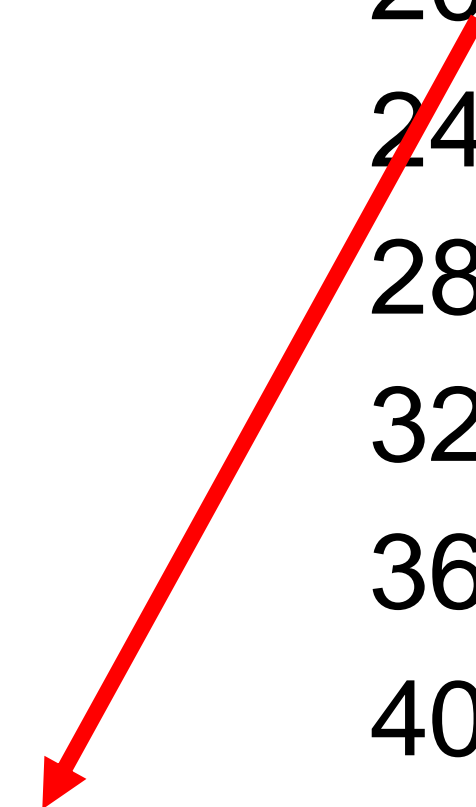


0
16

A	C	E	F
A	D	X	Y

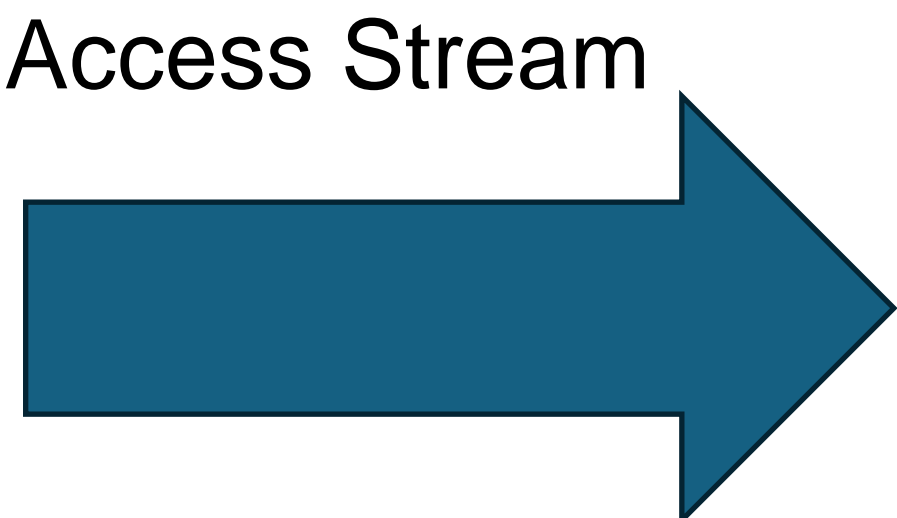
4, 24 (miss)

0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative
- Cache Hit



4, 24, 8 (hit)

0
16

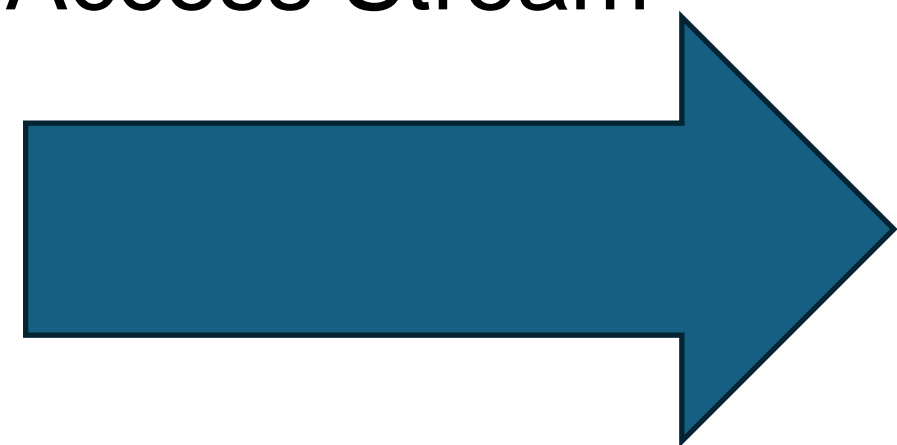
A	C	E	F
A	D	X	Y

0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P

A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

Access Stream

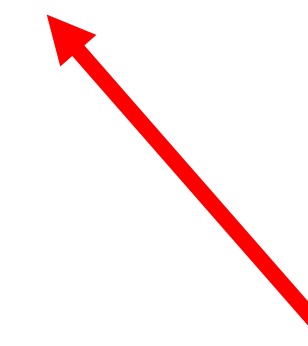


0
48

A	C	E	F
5	U	H	P

4, 24, 8, 48 (miss)

0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

Access Stream



0 d
48

A	XXc	E	F
5	U	H	P

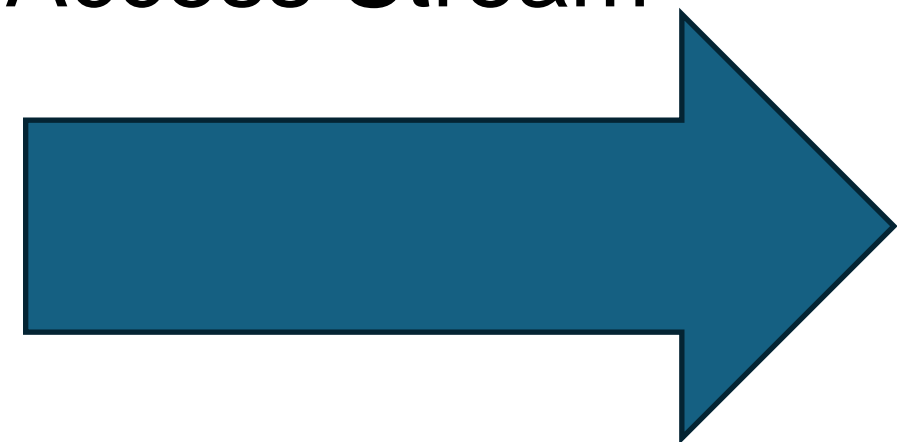
4, 24, 8, 48, 4 -> XXc (hit)

0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P

A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative

Access Stream



0 d
32

A	XXc	E	F
W	Q	55	W

4, 24, 8, 48, 4 -> XXc, 40 (miss)

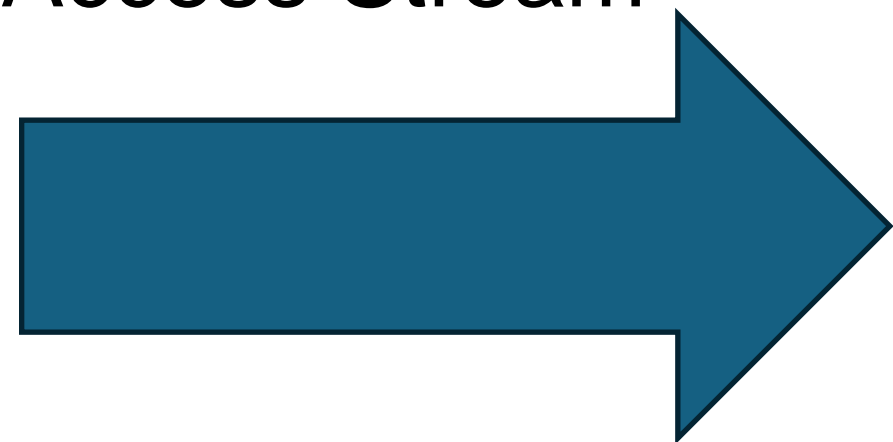
0	A
4	C
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative
- Miss (dirty line evicted)
 - Capacity miss since this line (16) was previously in the cache

Access Stream



16
32

A	D	X	Y
W	Q	55	W

4, 24, 8, 48, 4 -> XXc, 40, 16 (wb 0)

0	A
4	XXc
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P



A simple cache

- block size = 16 bytes (4 words), 2 entries, fully associative
- Store miss
 - capacity miss because cache line (48) was previously in the cache

Access Stream



16
52 d

A	D	X	Y
5	XXu	H	P

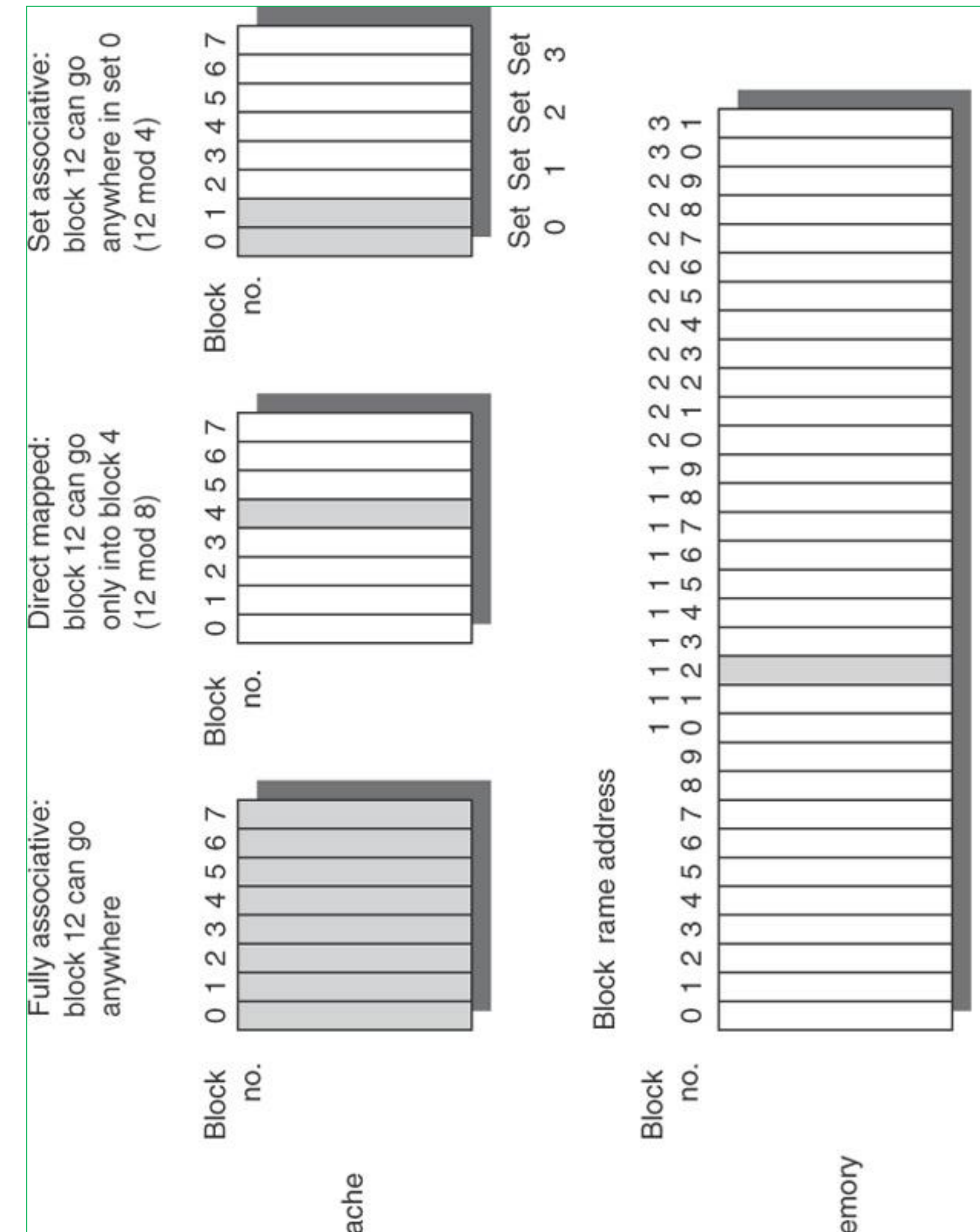
0	A
4	XXc
8	E
12	F
16	A
20	D
24	X
28	Y
32	W
36	Q
40	55
44	W
48	5
52	U
56	H
60	P

4, 24, 8, 48, 4 -> XXc, 40, 16 (wb 0), 52 -> Xxu (store miss, capacity)

Cache Organization: associativity

Where can a block be placed in the cache?

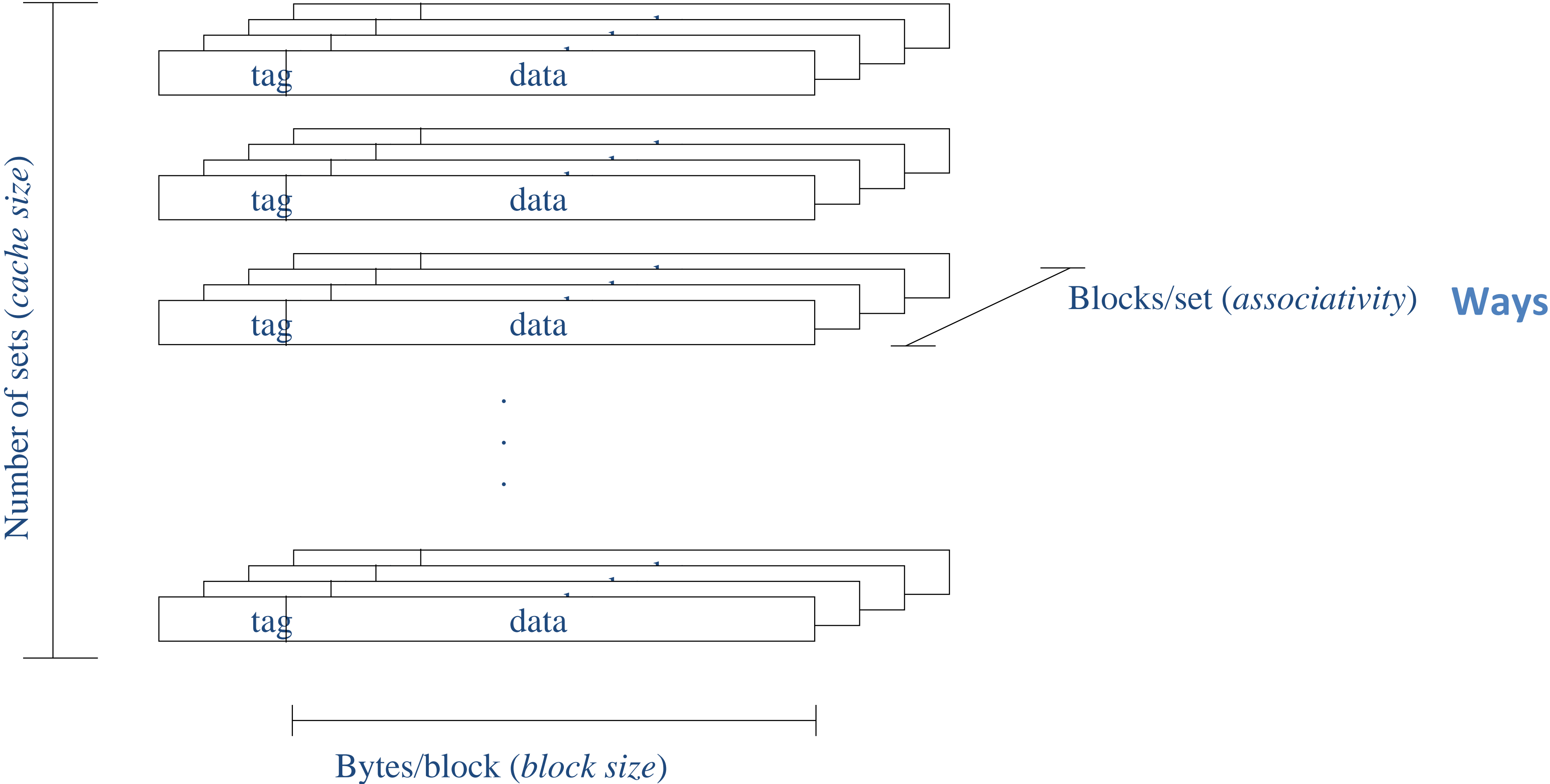
- Block 12 (memory) placed in 8-block cache:
 - Fully associative, direct mapped, n-way set associative
 - index = pointer to the set in the cache where a memory location might be cached
- (associativity = degree of freedom in placing a particular block of memory)
- (set = a collection of cache blocks with the same cache index)
- Conflict misses cannot happen in Fully associative cache



Cache Organization -- Overview

- A typical cache has three dimensions

tag	index	block offset
-----	-------	--------------



How big

Cache size is

cache block size (line size) * # sets * associativity (# ways)

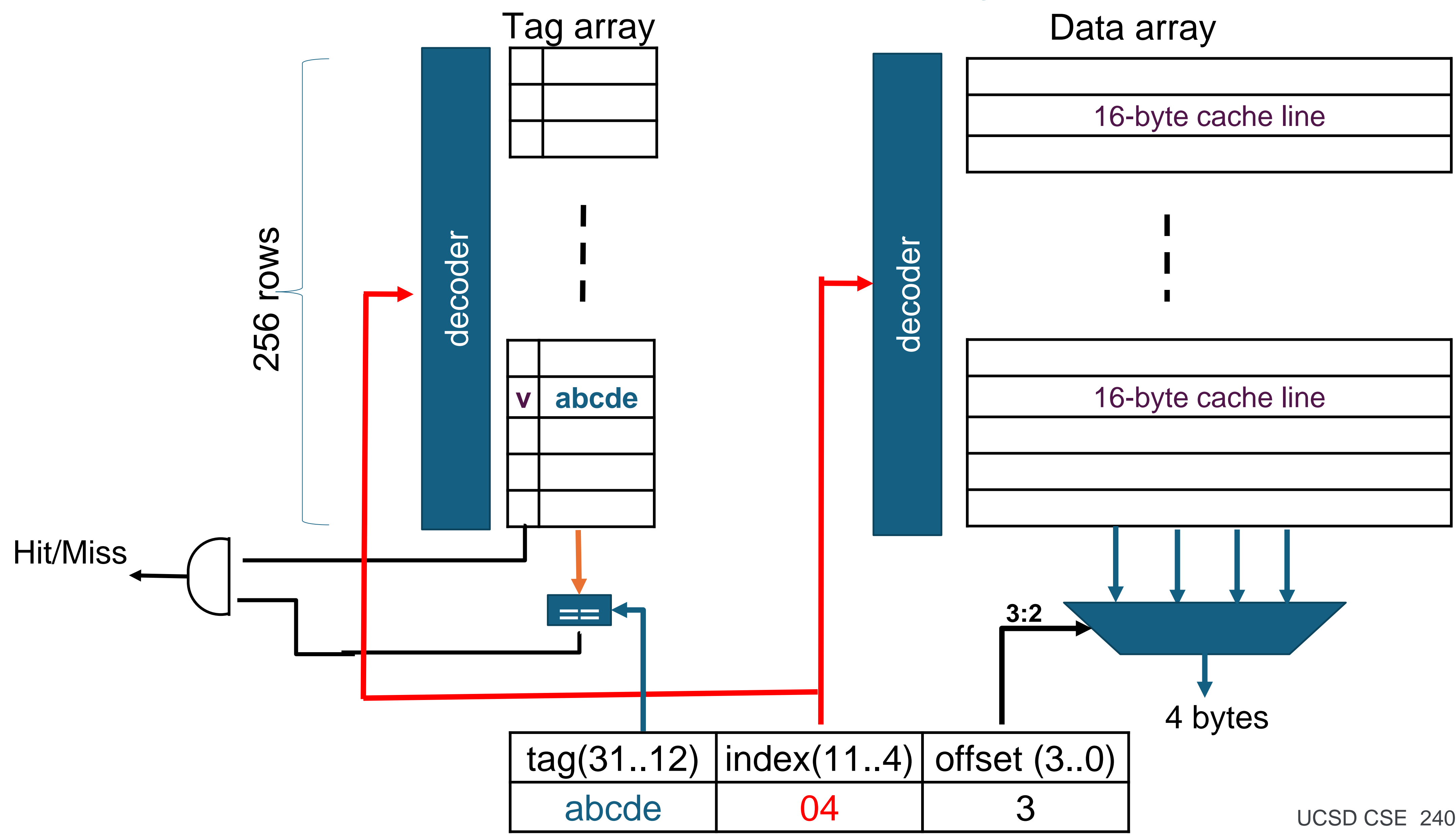
OR

cache block size (line size) * # blocks

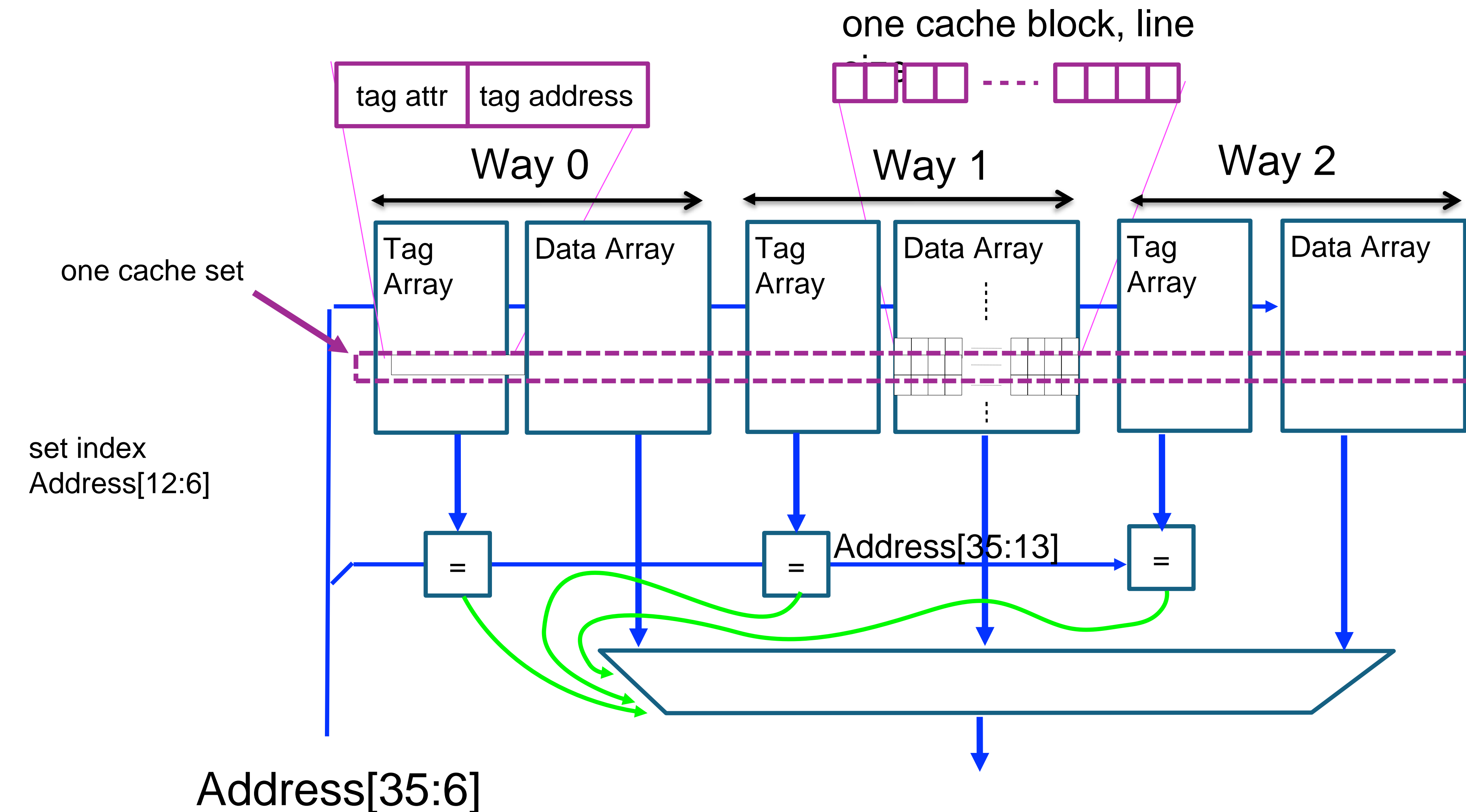
=> # sets = #blocks / associativity

Size does not include tags, status bits, etc. Size refers only to the data storage capacity.

Direct Mapped Cache – One Way Set Associative



Cache Terminology: Example 3-way cache



- **set** - set of addresses with identical set index bits
max # of aliased addresses we can store w/o conflict is number of ways
- Direct mapped cache (1 way)
- fully associative (ways = # cache lines)

Given a 4 KB cache 2 way-associative cache with 32-byte line size and a 32-bit address, how many bits are needed for address in the tag?



- A. 19
- B. 22
- C. 20
- D. 21
- E. None of these

webclicker.web.app NUTCOM

Cache Replacement Policies

- When a cache miss happens, which cache block should we replace. This called the cache *victim*
- If a miss occurs on cache line X and X was previously evicted (victimized) from the cache, this is called a *Conflict* miss

What is the relative frequency of conflict misses, in general from least to most in these cache (assume all are the same capacity (size))?

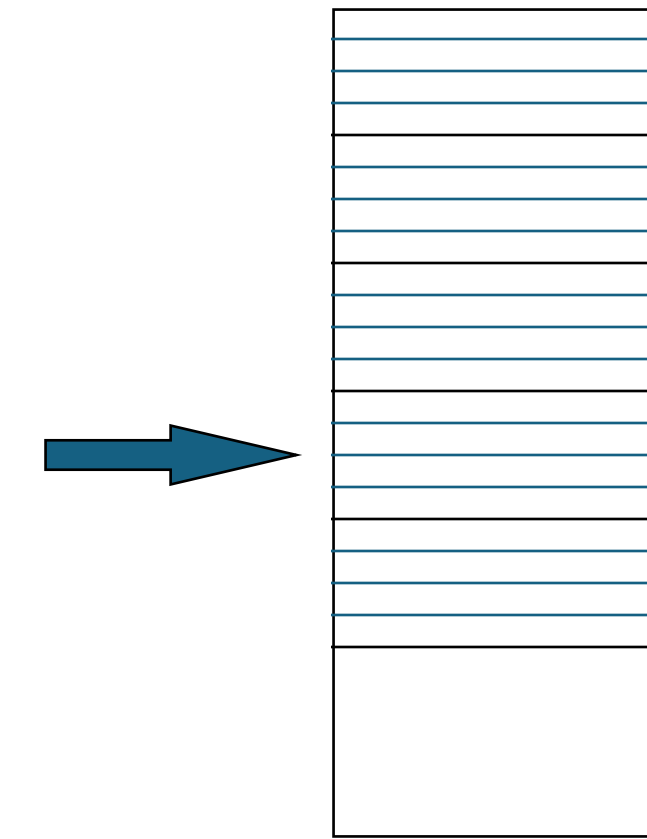
- A. direct, set associative, fully associative
- B. set associative, direct, fully associative
- C. fully associative, direct, set associative
- D. fully associative, set associative, direct

Cache Types

	FA	Set Associative	Direct Mapped
Capacity Miss	y	y	y
Conflict Miss	never	can happen	most likely
Cold Miss	y	y	y

Replacement Policy Performance

- Direct Mapped is Easy – only one way, no choice to make
- Set associative or fully associative:
 - longest till next use (ideal, impossible)
 - least recently used (a practical approximation)
 - pseudo-LRU (e.g., NMRU, NRU)
 - random (easy)
- how many bits for LRU?



Disclaimer – old results.

Associativity	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Implementing LRU

- Associativity (# of ways) n
- We need to store all possible orderings of Least recently used to Most Recently used.
- each way is represented by $\log_2(n)$
- n ways

~ queue of $\log_2(n) * n$ #SETS

Theoretical bound is: $\text{ceil}(\log_2(n!)) * \text{\#sets}$

64 KB 8 way-set associative cache with 64 byte cache lines

How many bits do we need to store LRU?

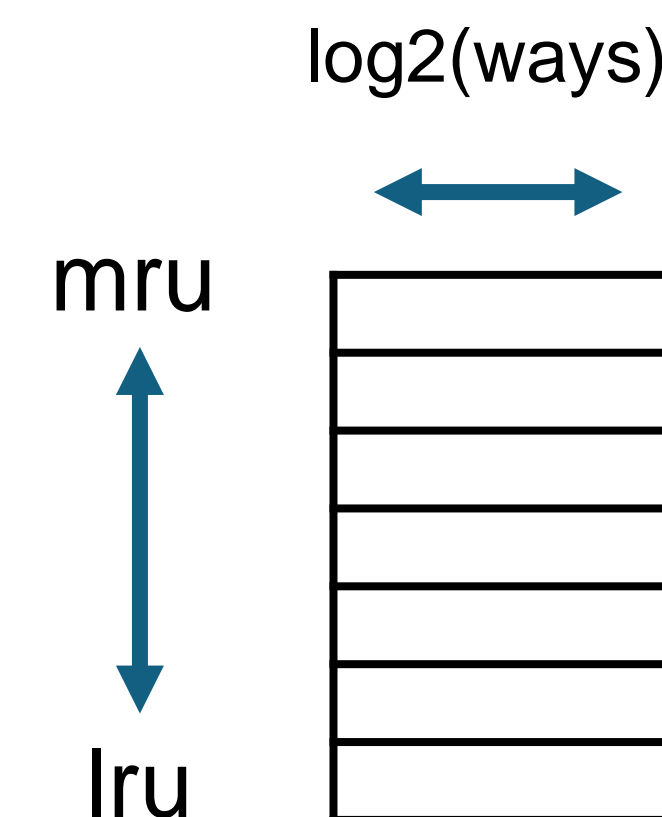
$$\log_2(8) = 3$$

$$8 \text{ ways} * 3 = 24 \text{ bits per set}$$

$$\text{\#blocks} = 64\text{K}/64 = 1024$$

$$\text{\#sets} = 1024/8 = 128$$

$$128 * 24 = 3072 = 384\text{B}$$

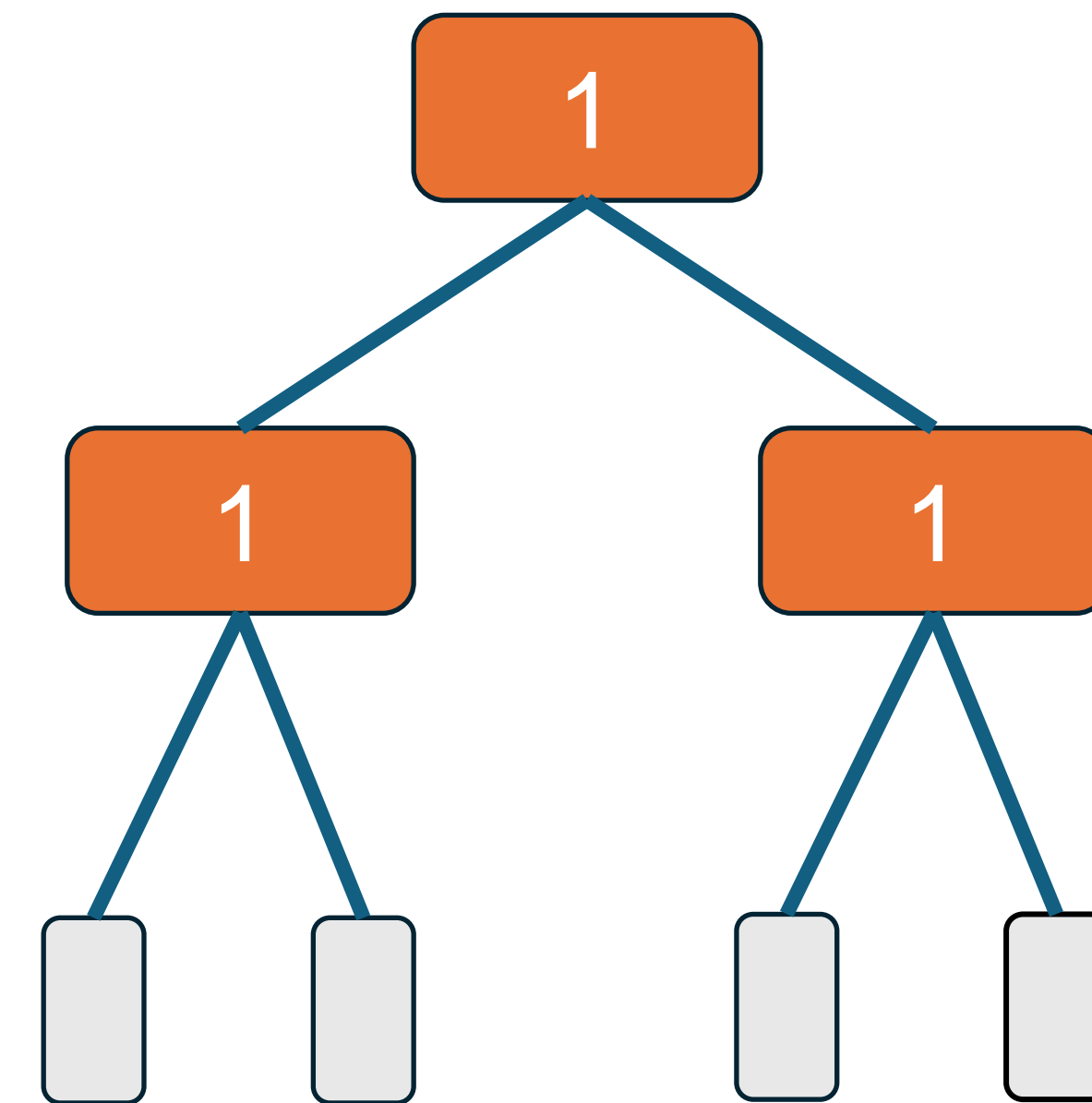


Implementing LRU complexities

- When cache miss
 - **read** the LRU bits, pick the LRU set
 - on refill **write** the LRU bits with the new LRU ordering
- When cache hits
 - **read** the LRU bits,
 - **write** the updated LRU bits
- Where should this state be stored?

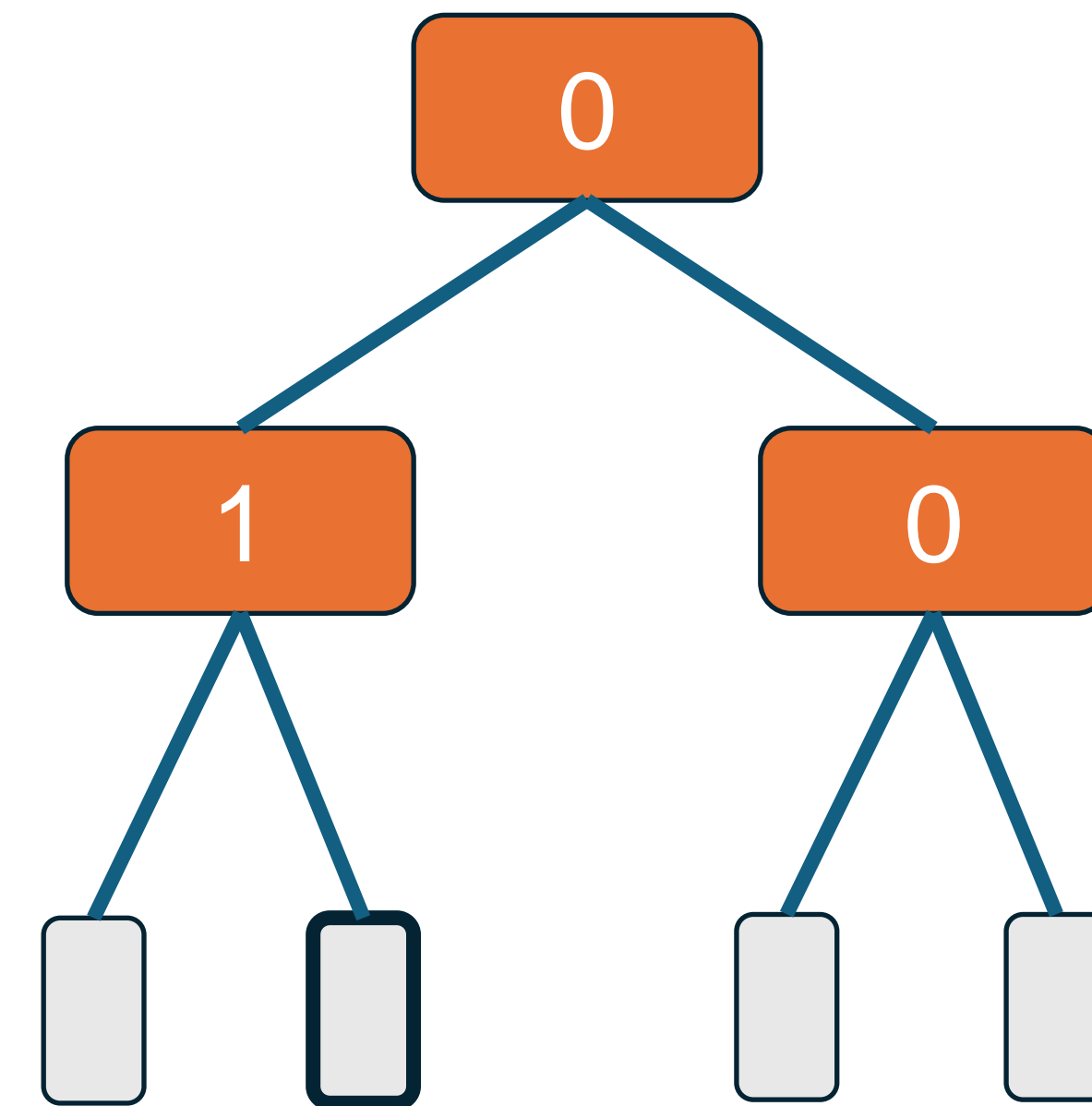
Alternatives – Psuedo LRU

- Approximate LRU
- Keep track of most recent access



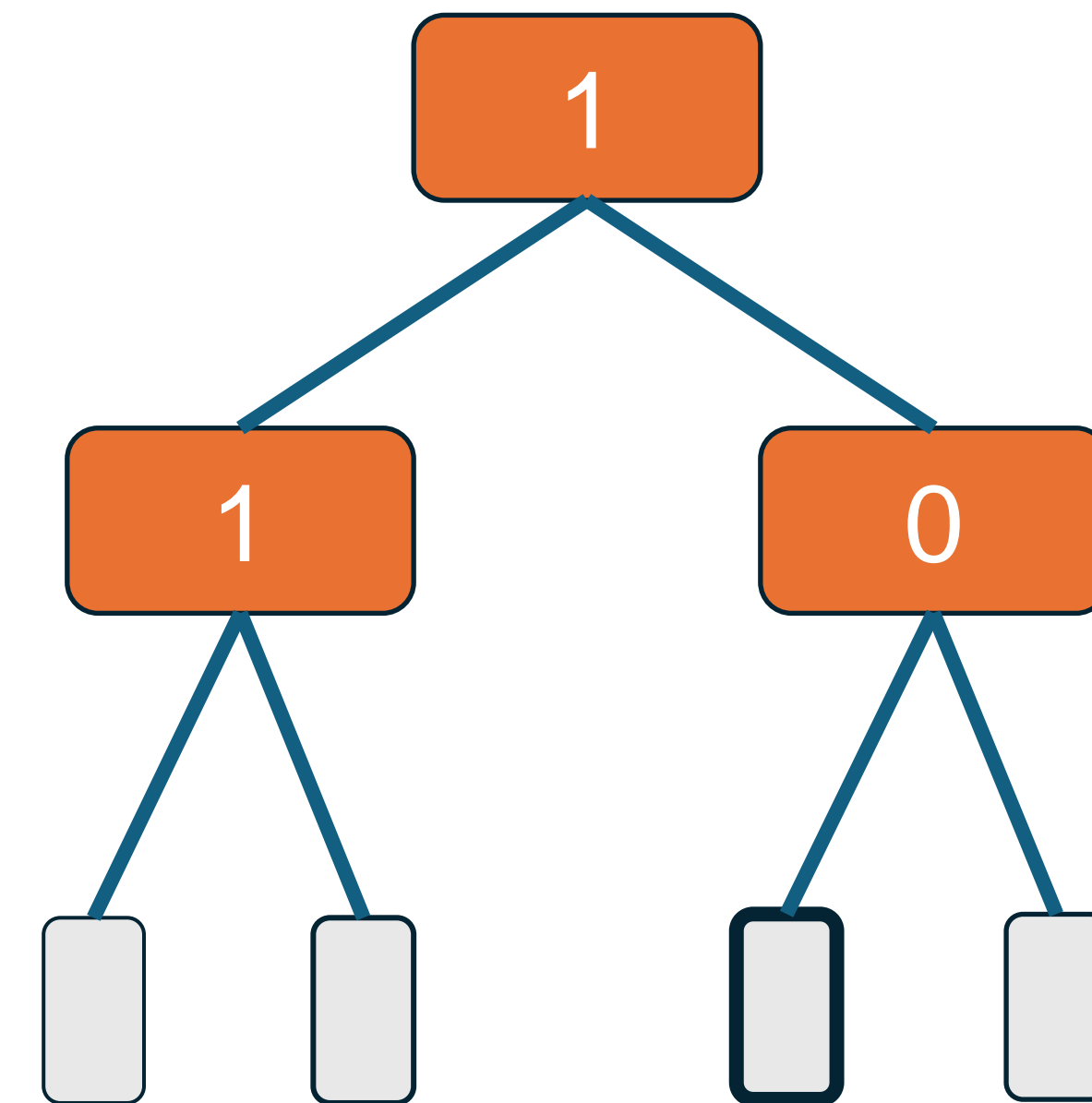
Alternatives – Psuedo LRU

- Approximate LRU
- Keep track of most rece0
- Hit on way 3



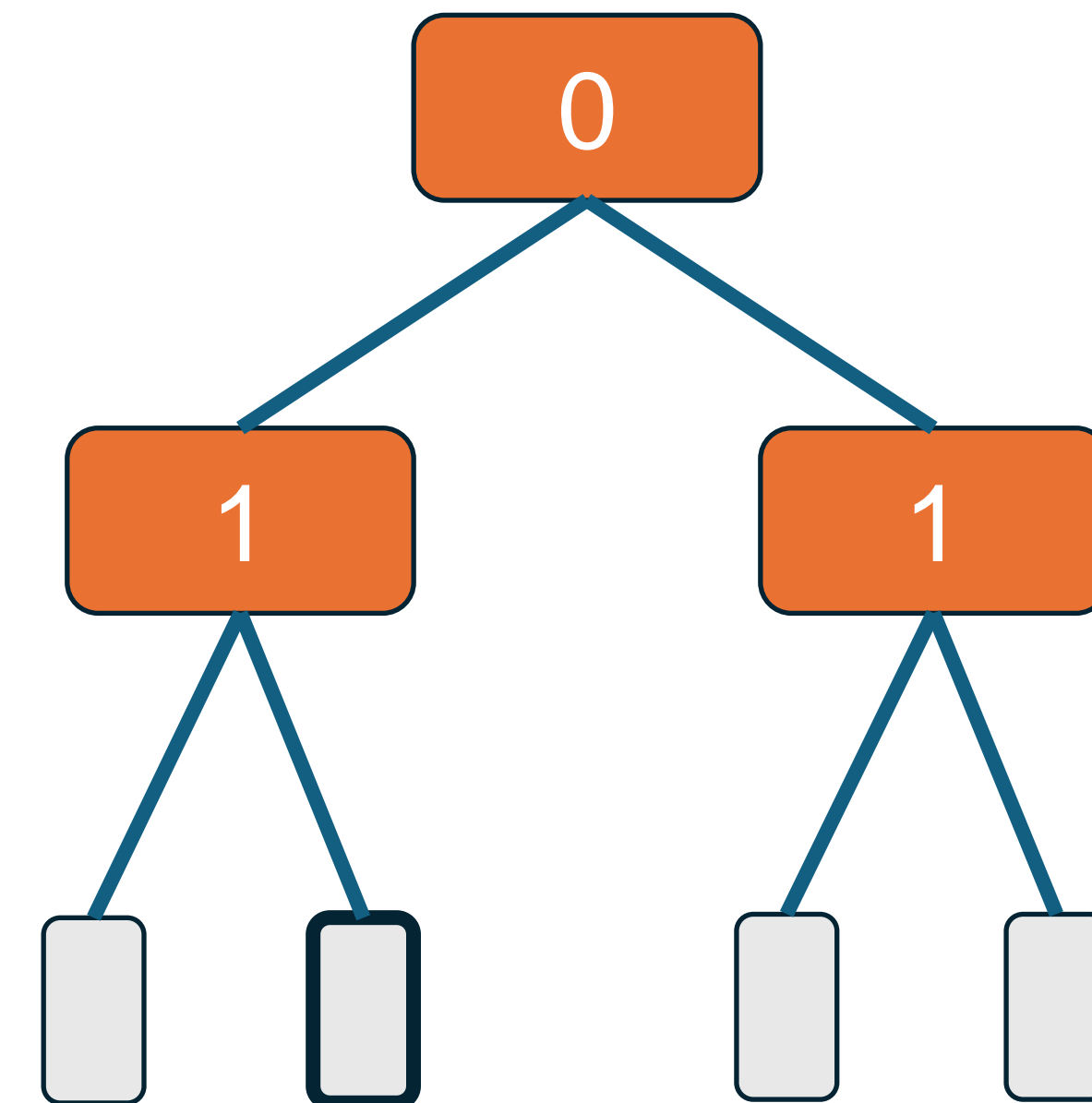
Alternatives – Psuedo LRU

- Approximate LRU
- Keep track of most rece1
- Hit on way 3
- Hit on way 0



Alternatives – Psuedo LRU

- Approximate LRU
- Keep track of most rece0
- Hit on way 3
- Hit on way 0
- Miss – fill way 2



Write happens on a store

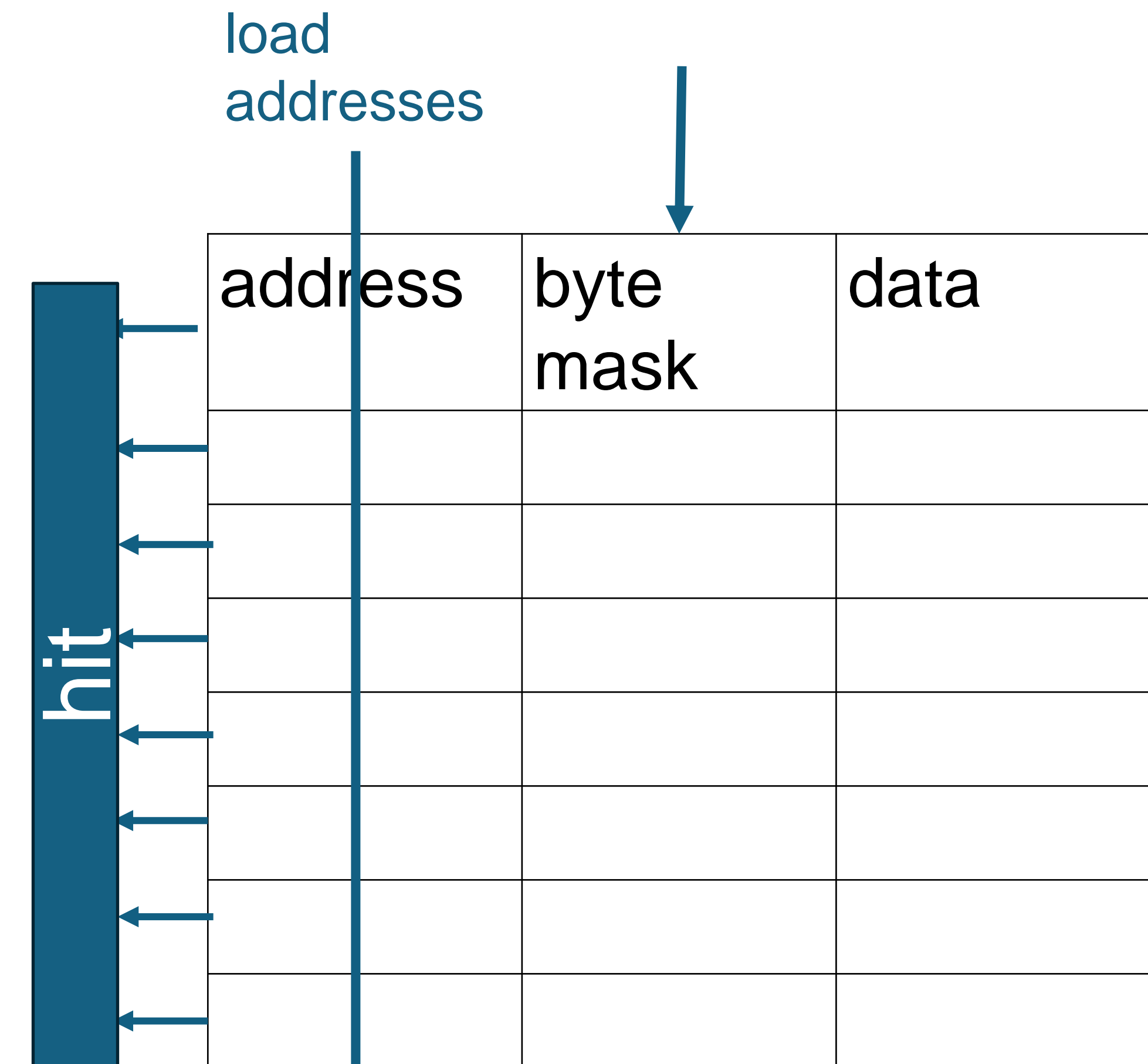
- Load – just load the cache with the missing cache line
- Stores – more decisions to make
 - Store hit
 - *writeback* – cache line is updated, next level cache/memory is not
 - *writethru* – cache line is updated and next level cache is updated as well

Writeback vs Writethru

- **writeback** – store go to cache
 - tag needs to maintain a "D" bit (dirty bit).
 - When cache line is replaced, it needs to be written back
 - "Ownership" – cache is said to own that cache line and is responsible for making sure that new value is visible at some point
- **writethru / write around**– stores go to cache and to next level in the memory hierarchy
 - no need for "D" bit – line is always clean.
 - Writes go to "writebuffer" so processor doesn't need to wait for writes to propagate to slower memory
- Pros/Cons
 - writethru - misses don't need to write data back to memory
 - writeback – high bandwidth. Stores to the same cache line don't generate more traffic

Writebuffer

- memory systems are often optimized for certain data sizes
- Generally, cache line transactions
- Smaller than 64-bits usually means read-modify-write because error correction is done in a 64-bit basis (read the data, modify it, generate new error checking code, write it back to memory)
- Write combining buffers try to gather larger chunks of data
 - merge redundant or overlapping writes (maybe)
- Need to *snoop* the writebuffer if processor writes data and then reads it back
 - can forward from the writebuffer
 - can flush the writebuffer and refill from next level memory (or main memory)



Store Miss

Two choices

- Allocate a cache line for the store – write allocate
- Don't allocate a cache line for the store – write no allocate

Tradeoffs

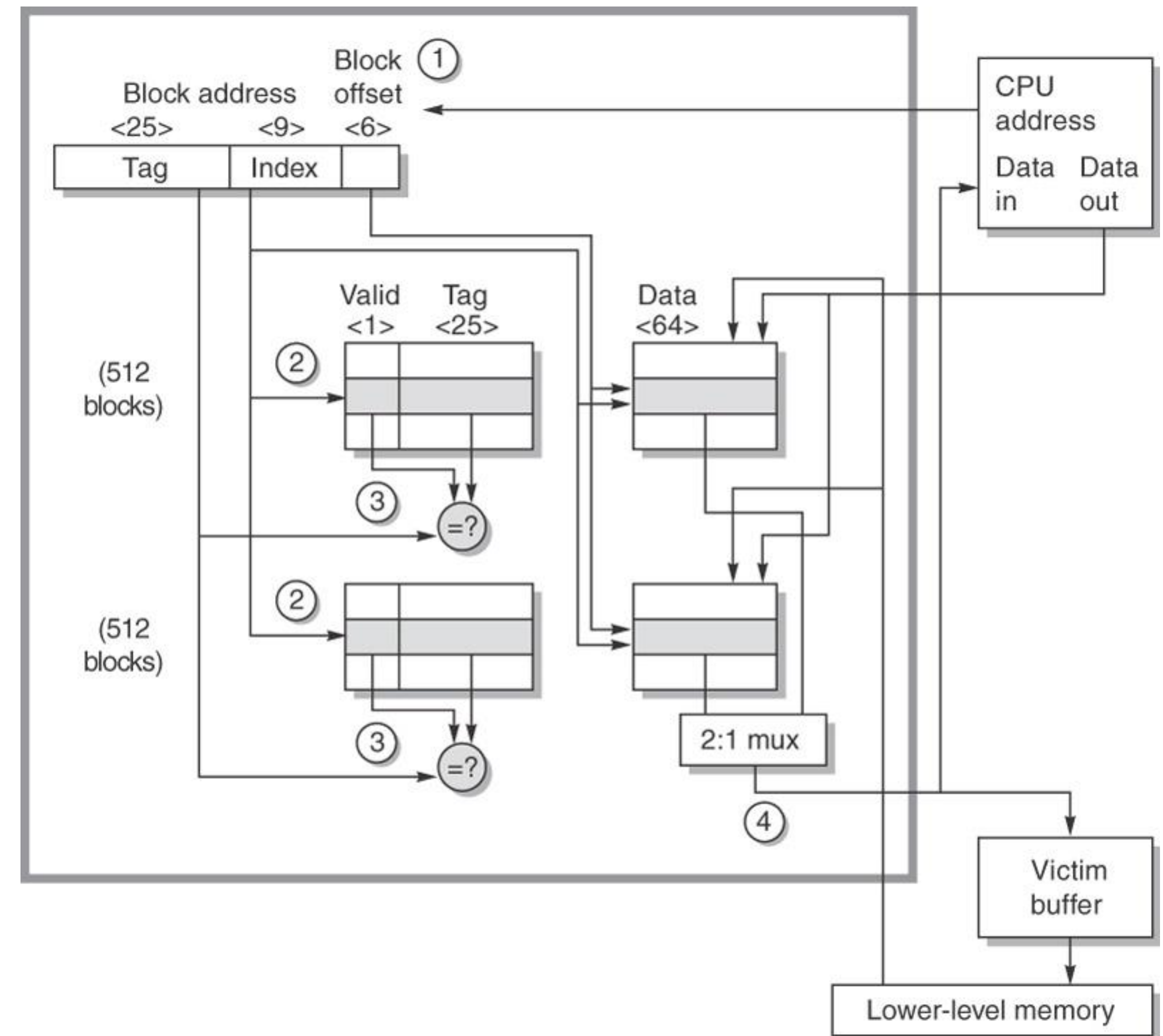
- **Store Allocate** : generally - when storing data, processor is done with it for a while. Otherwise it would keep the data in registers. Allocating cache lines for stores might evict data that is needed
- **Store No allocate** – need to put data into the Writebuffer. But lower levels of memory hierarchy are generally lower bandwidth (bigger in size). So what if we can't keep up up with store rate. Writebuffers generally high bandwidth but low capacity.

Which policy store allocate or store-noallocate makes sense dealing with I/O devices? (e.g writing to a serial device)

- A. store allocate
- B. store no allocate
- C. not sure

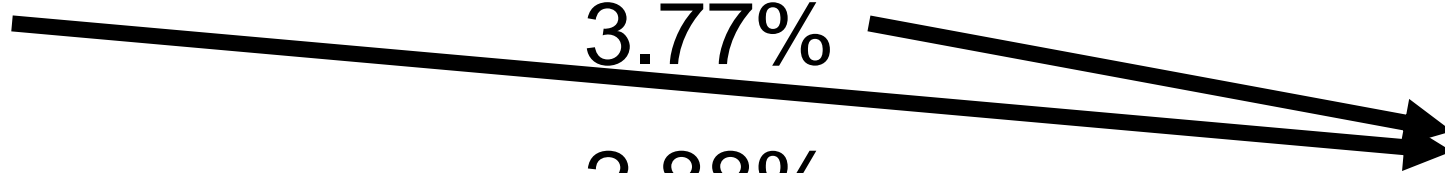
Example – Alpha 21264 D cache

- 64 KB, 2 way set associative



Cache Organization: Separate Instruction and Data Caches?

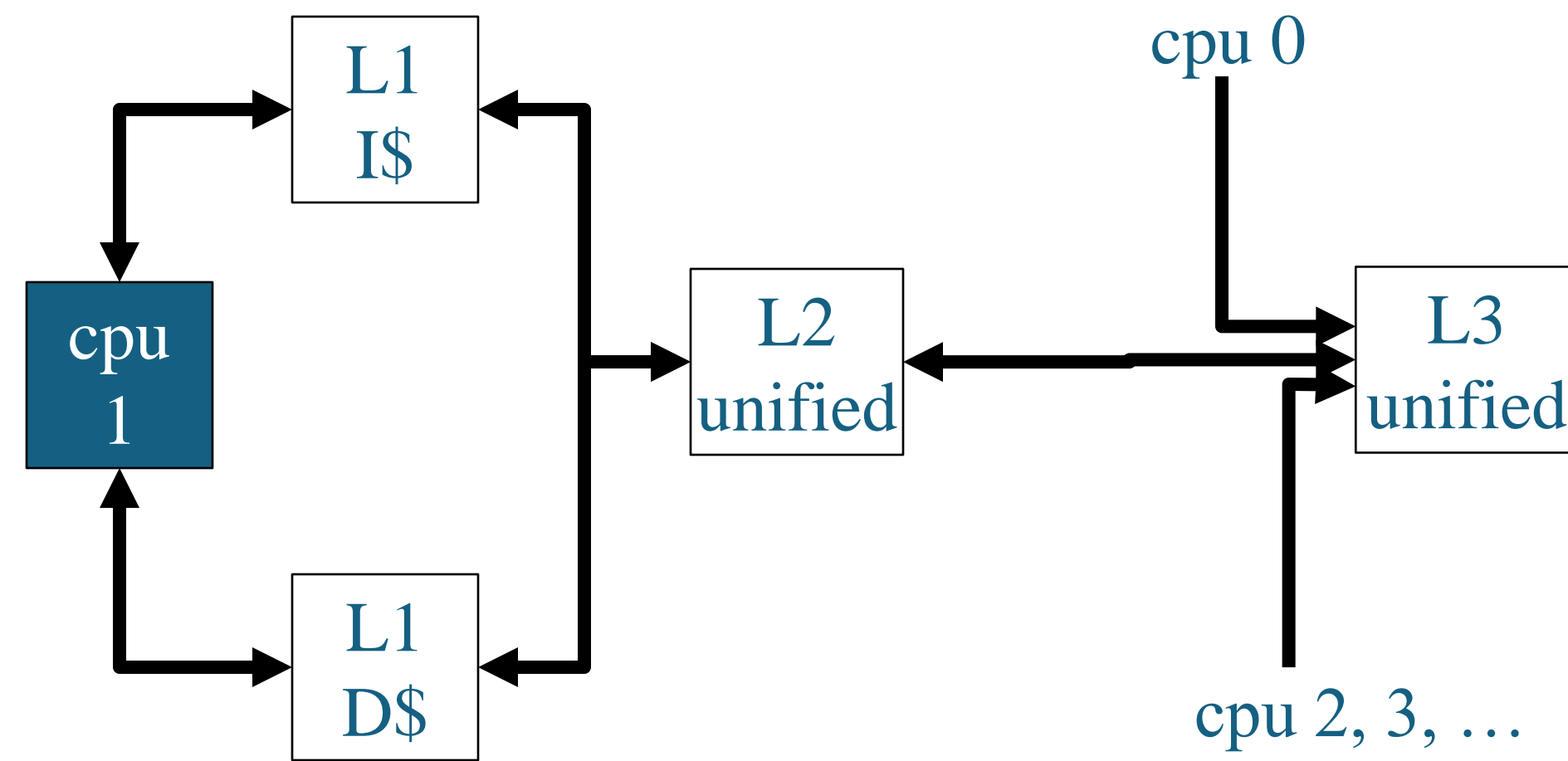
Size	Instruction Cache	Data Cache	Unified Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%



if 75% of accesses are instructions?

Other reasons to separate?

Split I and D



Cache Performance

$$\text{CPUtime} = \text{IC} \times (\text{CPI}_{\text{execution}} + \text{Memory stalls per instruction}) \times \text{Clock cycle time}$$

Cache Performance

$$ET = IC \times (CPI_{\text{execution}} + \text{Memory stalls per instruction}) \times \text{Clock cycle time}$$

$$ET = IC \times (CPI_{\text{execution}} + \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}) \times \text{Clock cycle time}$$

(includes hit time as part of CPI)

(note, I will typically call “memory stalls (cycles) per instruction” **MCPI**)

Cache Performance

- $ET = IC \times (BCPI + MCPI) \times \text{clock cycle time}$
- $MCPI = \text{Memory accesses/inst} \times \text{Miss rate} \times \text{Miss penalty}$
 - Or, if you have more detail...
- $MCPI = (\text{Loads/inst} \times \text{Load miss rate} \times \text{Load miss penalty} + \text{Stores/inst} \times \text{Store miss rate} \times \text{Store miss penalty})$
- Or...
- $\text{Memory stall clock cycles} = (\text{Inst Cache Reads/inst} \times \text{IC miss rate} \times \text{IC miss penalty} + \text{Data Cache accesses/inst} \times \text{DC miss rate} \times \text{DC miss penalty})$
- Etc.

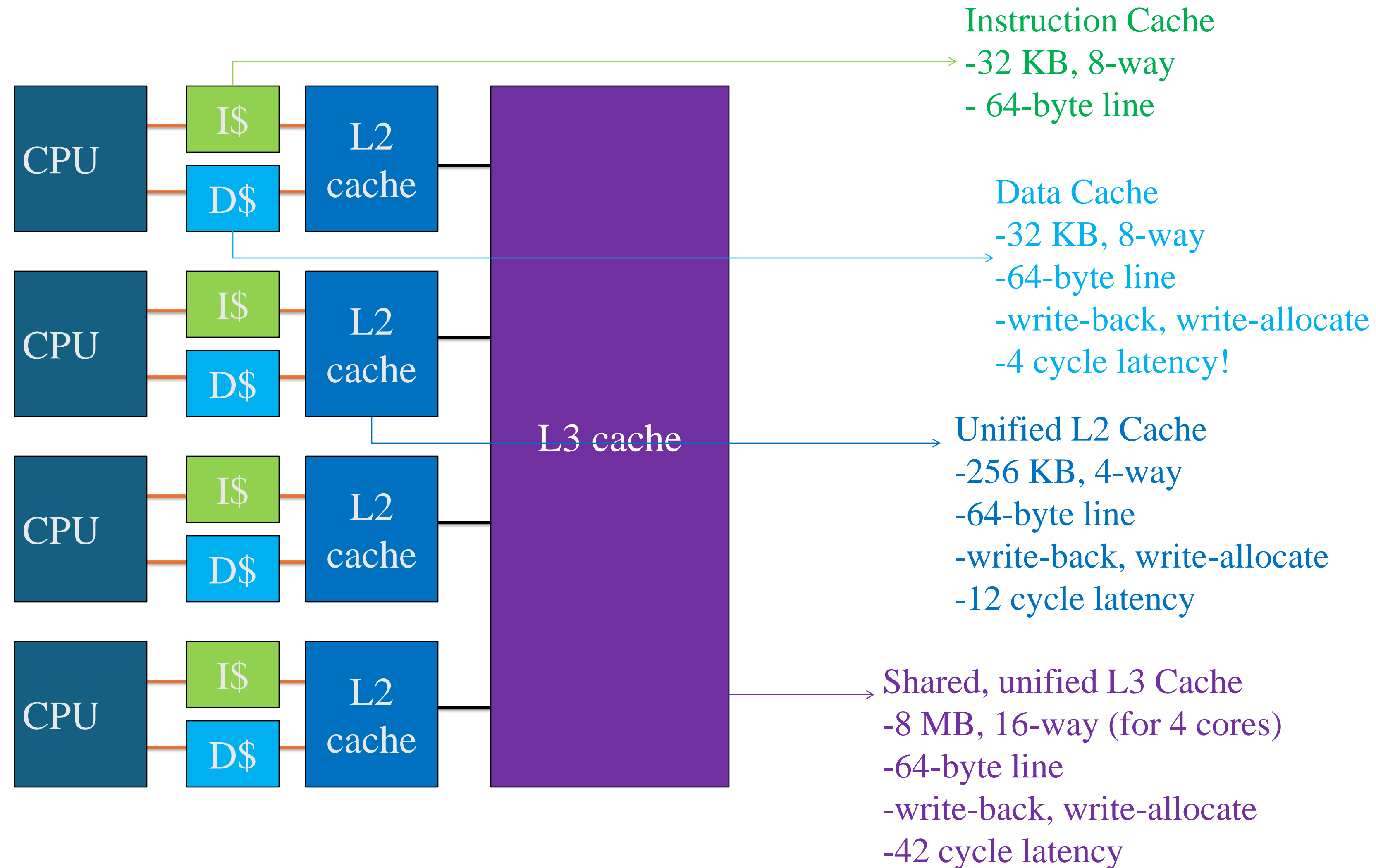
Units???

$MCPI = \text{mem acc/inst} * \text{cache misses/mem access} * \text{cycles/cache miss}$

Cache Performance

- Instruction cache miss rate of 4%, data cache miss rate of 9%, Base CPI (no memory stalls) = 1.1, 20% of instructions are loads and stores, miss penalty = 12 cycles, CPI = ?
- A. 1.70
- B 1.80
- C. 2
- D. None of these

Intel Skylake



Memory latency = 42 cycles (L3 hit time) + 51ns
 = 246 cycles at 4 GHz

Another way to think about performance

- I like MCPI, because it puts it in the context of the full ET equation.
- But sometimes you want to just focus on the performance of the memory hierarchy and ignore everything else.
- So sometimes, you see in the literature:
- **AMAT** – Average Memory Access Time
 - When you do a memory access, how long on average does it take?

Cache Performance - AMAT

(Alternate metric of memory performance used in a lot of research – ignores non-memory cpu components)

Average memory access time = Hit time + Miss rate x Miss penalty (ns or clocks)

Improving Cache Performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

How are we going to improve cache performance??

1.

2.

3.

Caches, pt I: Key Points

- CPU-Memory gap is a major performance obstacle
- Caches take advantage of program behavior: locality
- Designer has lots of choices -> cache size, block size, associativity, replacement policy, write policy, ...
- Time of program still only reliable performance measure