Cache Designs

Computer Architecture

Wei Wang

Computer Science
University of Texas at San Antonio

Text Book Chapters

- “Computer Organization and Design,” Chapter 7.2 and 7.3.
- “Computer Architecture: A Quantative Approach,” Appendix B.

Road Map

- Cache Placement Policies
- Cache Replacement Policies
- Write Strategies

Cache Placement Policies

Cache Placement Policies

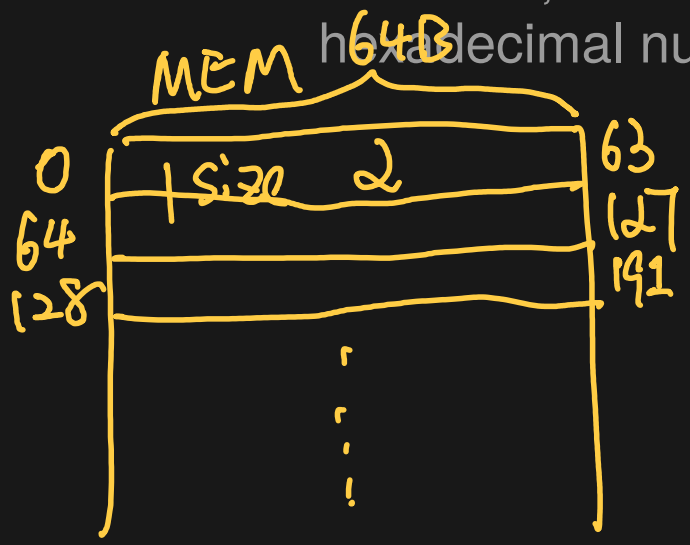
- For a cache line, which frame (slot) should it be stored?
- Three types of cache placement policies
  - Direct Mapped Caches
  - Fully Associated Caches
  - Set Associated Caches

Cache Line Addresses

- Consider a cache line with of size 2^s, the last s bits of its address (in binary number) will always be zero.
  - By definition, a new cache line always starts at an address which is multiple of 2^s.
  - For example, for cache line size of 64 (2^6) bytes, a cache line may start at address 0, 64, 128, ... etc. The binary representations for these addresses are $0b0000\ 0000$, $0b0100\ 0000$, $0b1000\ 0000$, ...
    - Note that the last six bits are always zero.
  - Note that, to save space, I will write address in hexadecimal numbers in the future.

64B = 2^6B ⇒ s = 6

32B = 2^5B ⇒ s = 5



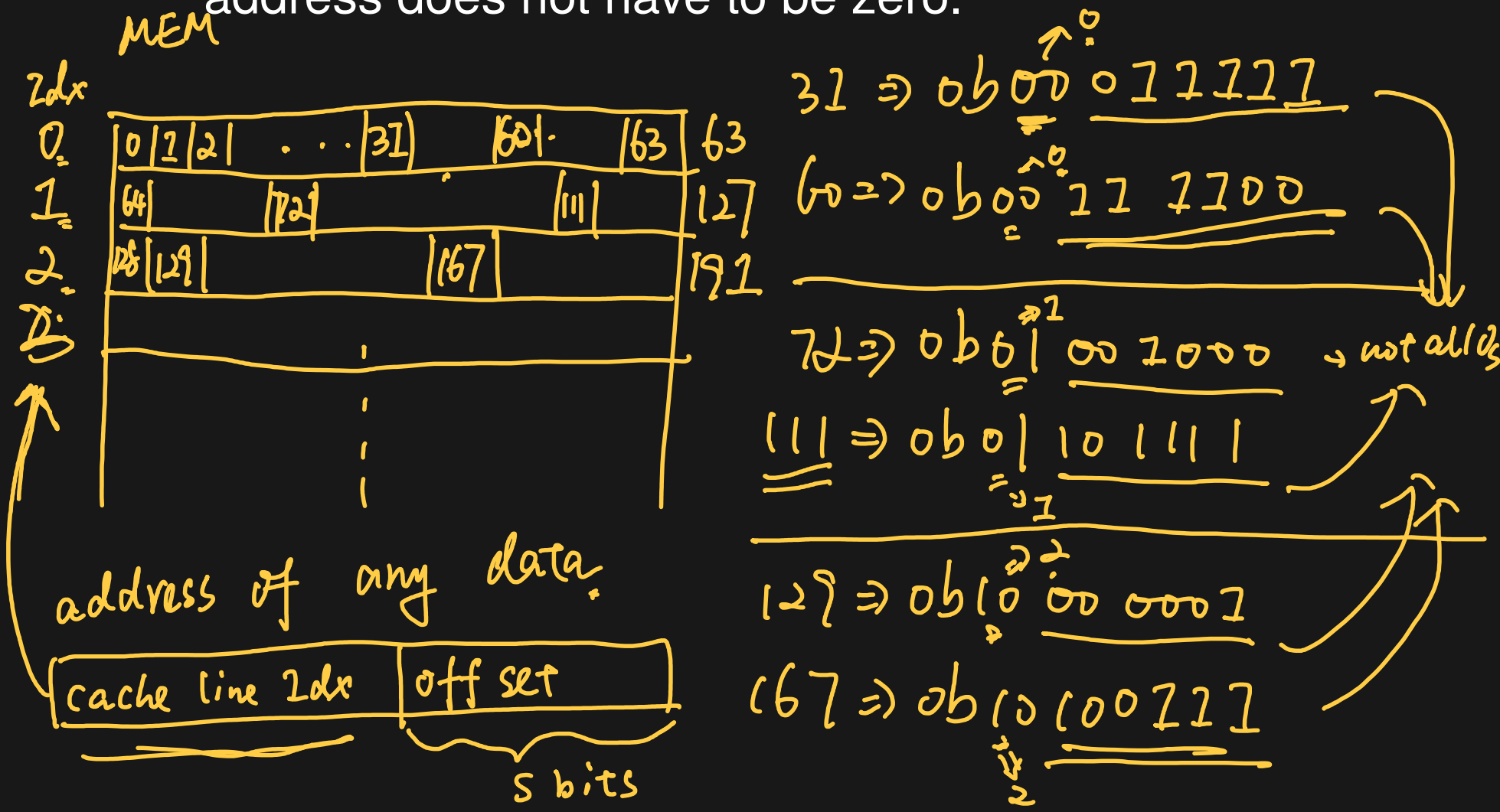
0 ⇒ 0b0000 0000 → all 0s

64 ⇒ 0b0100 0000 → all 0s

128 ⇒ 0b1000 0000 → all 0s

Memory Addresses for Data cont'd

- For an arbitrary piece of data, the last *s* bits of its address does not have to be zero.

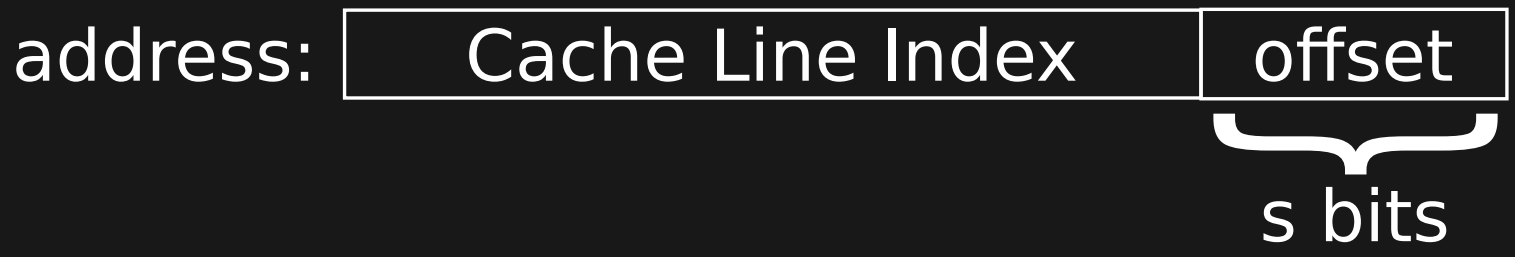


Memory Addresses for Data

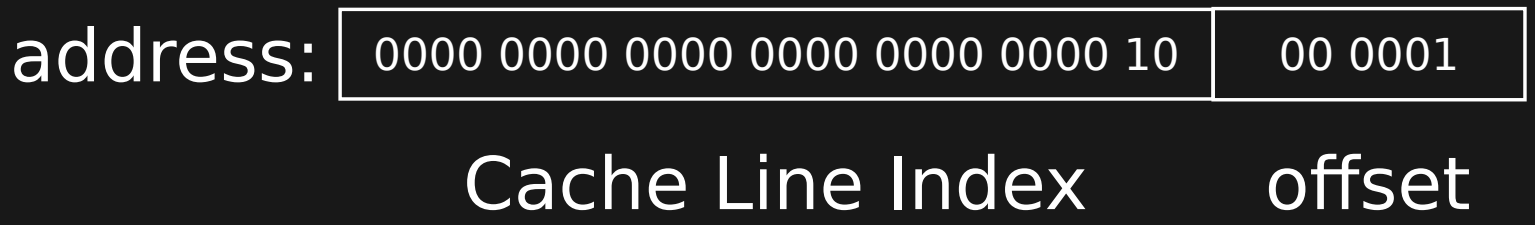
- For an arbitrary piece of data, the last *s* bits of its address does not have to be zero.
- If the last *s* bits of the memory address of a piece of data are not zero, then this data must be inside a cache line.
  - The last *s* bits gives the location of this data within its cache line.
  - We call the last *s* bits the offset (within the cache line) of this data.
  - E.g.,, for a piece of data at address $0x81$ (129 or $0b1000\ 0001$), its offset is $0x1$ ($0b00\ 0001$) for a 64-byte cache line.
- The rest bits of the address (i.e., non-offset bits) are the cache line index.
  - E.g., for address $0x81$, its cache line index is $0x2$ ($0b10$).

Memory Addresses for Data cont'd

- An illustration of memory address with offset and cache line index.



- E.g.„ for address $0x81$ with 64-byte cache lines, we have,



Direct Mapped Caches

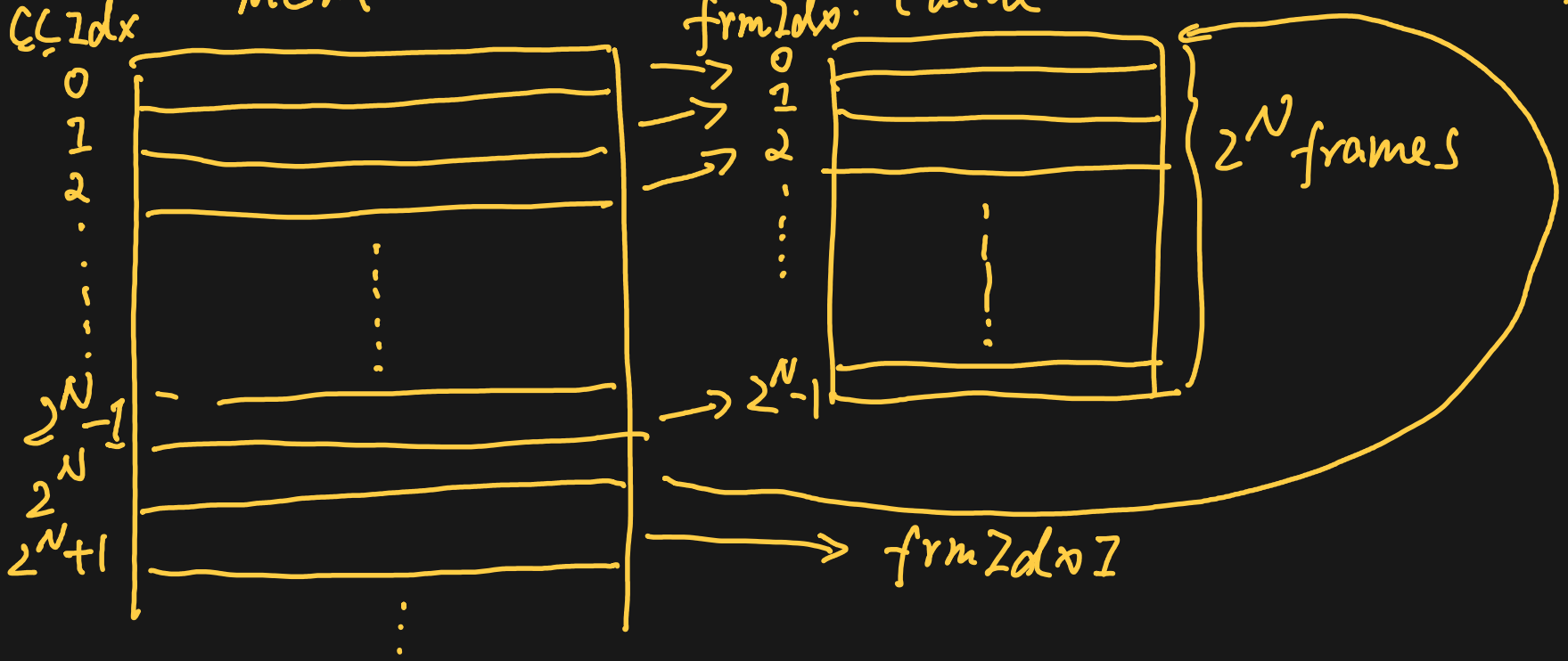
2^N frame x 2^S B = 2^{N+S} B

- The simple example we have seen in the "Introduction to Cache" lecture is a direct mapped cache.
- The placement policies for a cache with 2^N frames/slots with cache line size of 2^s is,

Map Algo

frm2dx = CL2dx % 2^N

CL in Mem ⇒ one & only one frame cache



Direct Mapped Caches

- The simple example we have seen in the “Introduction to Cache” lecture is a direct mapped cache.
- The placement policies for a cache with 2^N frames/slots with cache line size of 2^s is,
  - For a piece of data at address *a*, let its cache line index be *cidx*,
  - it will be mapped to the frame/slot at index $cidx \% 2^N$.
  - E.g., a cache line at address $0x81$ has a cache line index of $0x2$ if cache line size is 64 bytes. Therefore, if the cache has 2^4 slots,it will be mapped to the slot at $0x2 \% 2^4 = 2$.
- Note that, for a cache with 2^N slots, and each slot can store a cache line of 2^s bytes, the size of this cache is $2^N \times 2^s$ bytes.

Fast Modulo Operation

frm2dx = CL2dx % 2^N

decimal

789 % 10 = 9

789 % 10^2 = 89

789 % 10^3 = 789

X % 10^N = last N digits of X

binary

Y % 2^N = last N bits of Y

45 % 2^4 = 45 % 16 = 13

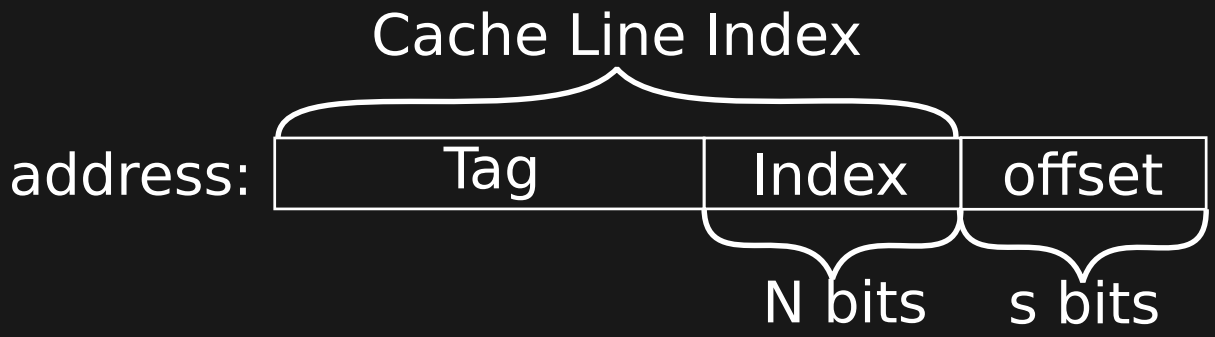
45_{10} % 2^4 = 1101_2

0b10 1101 % 2^4 = 1101

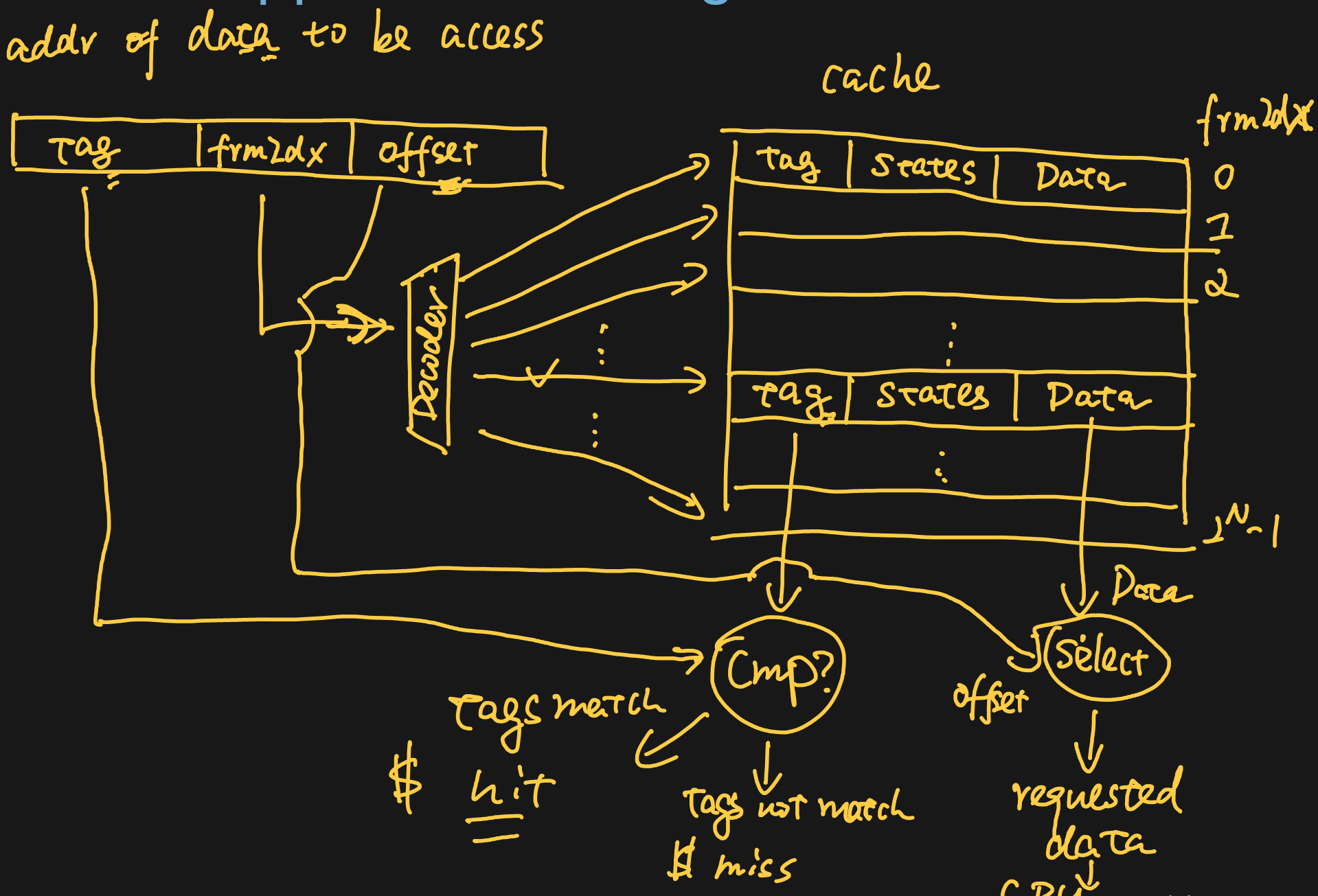
addr of a data

Fast Modulo Operation

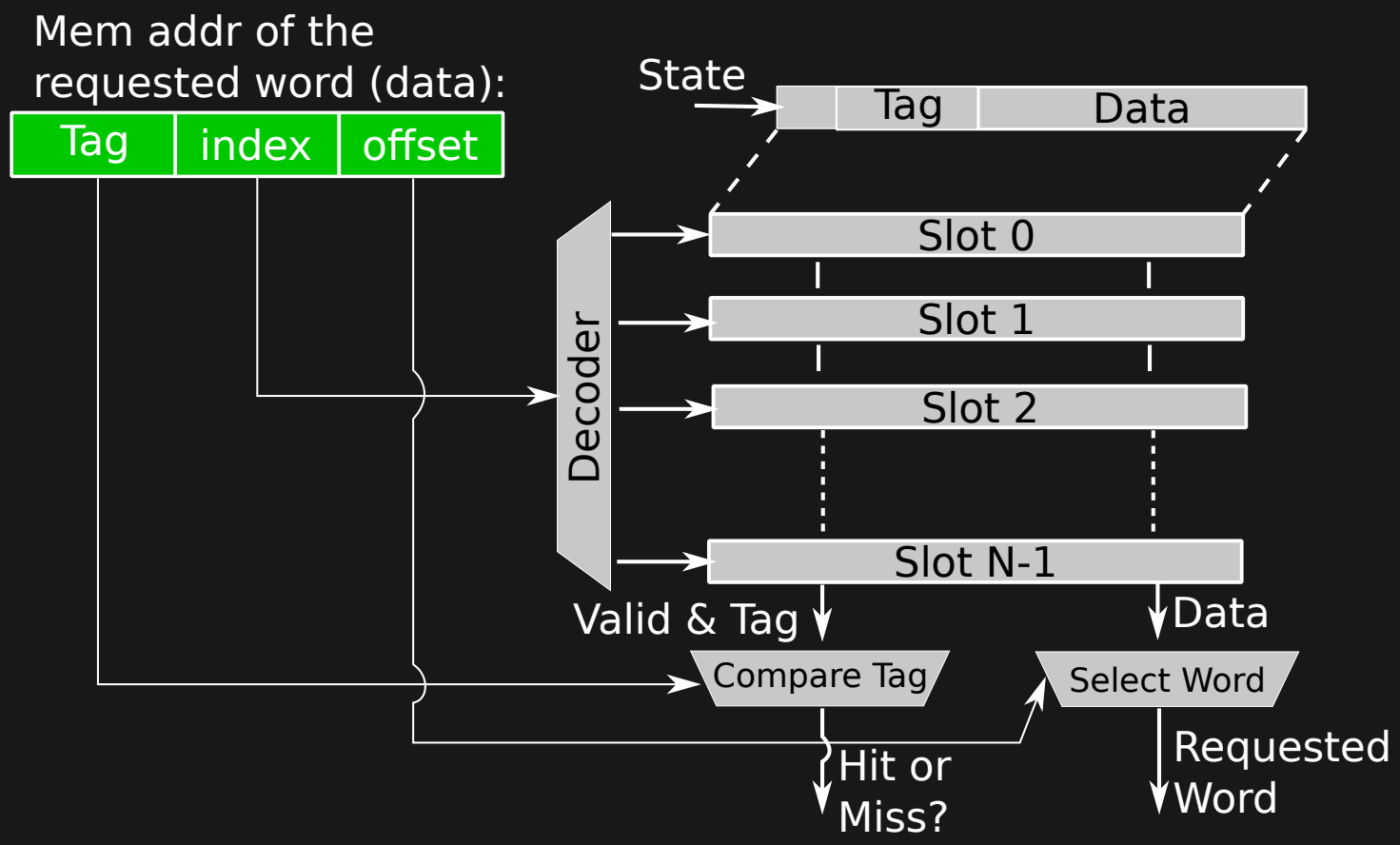
- A key benefit of Direct Mapped Cache is that frame index can be easily computed from memory addresses.
  - The cache line index *cidx*, can be easily acquired from a memory address by extracting the cache line index bits.
  - The modulo operation on the cache line index is especially extracting the last *N* bits from the
    - These *N* bits are called frame index bits or simply index bits.
    - The rest of the bits in cache line index are called tag. They are also THE tags used to identify a cache line.



Direct Mapped Cache Organization



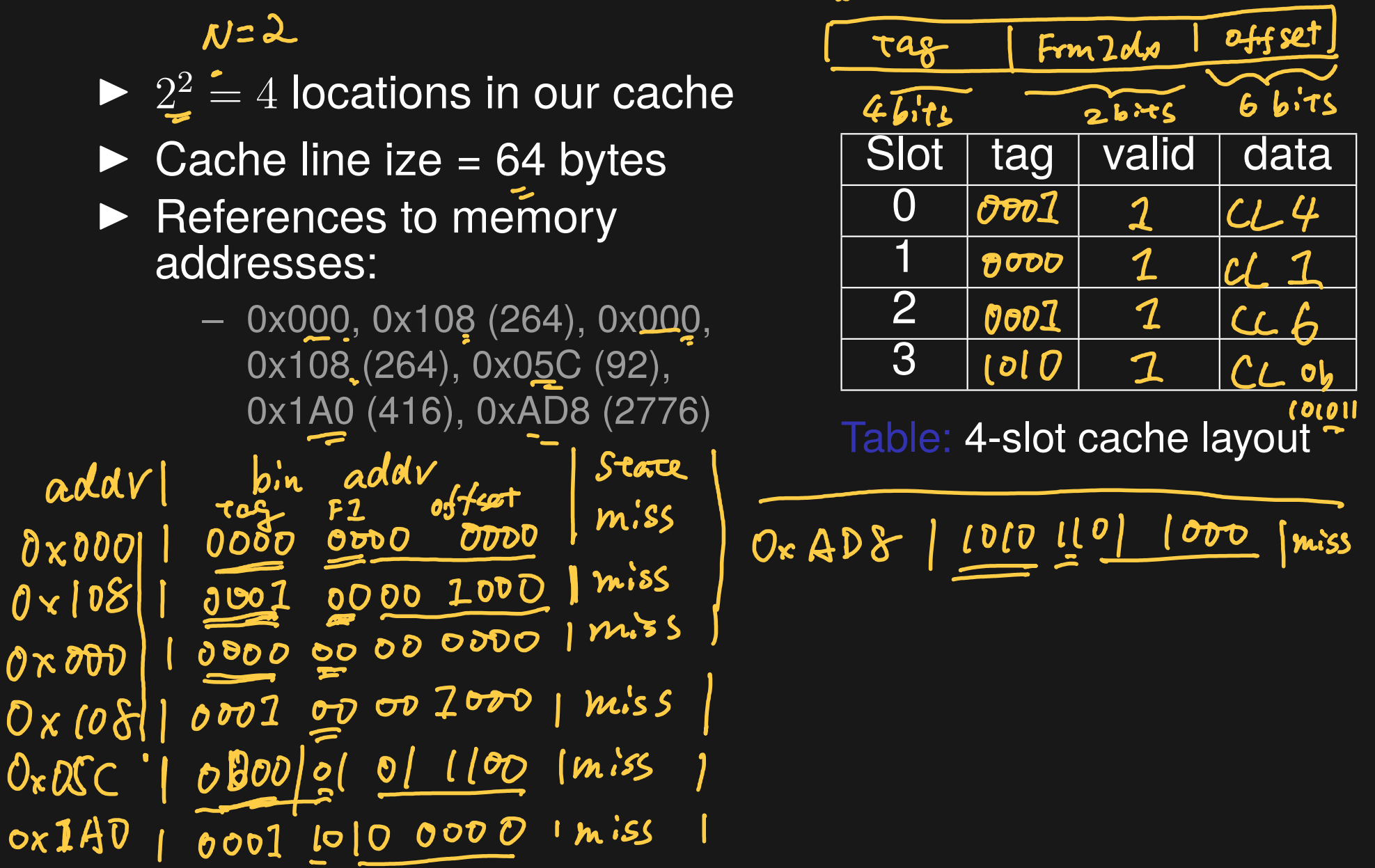
Direct Mapped Cache Organization



- The index bits are sent to decoder, which activates the corresponding frame/slot for tag comparison and word (data) accesses.

An Example of Direct Mapped Cache

- $2^2 = 4$ locations in our cache
- Cache line ize = 64 bytes
- References to memory addresses:
  - 0x000, 0x108 (264), 0x000, 0x108 (264), 0x05C (92), 0x1A0 (416), 0xAD8 (2776)



Slot	tag	valid	data
0	0001	1	CL 4
1	0000	1	CL 1
2	0001	1	CL 6
3	1010	1	CL 0

Table: 4-slot cache layout

addr	bin addr tag	F1	offset	State
0x000	0000	00	0000	miss
0x108	0001	00	1000	miss
0x000	0000	00	0000	miss
0x108	0001	00	1000	miss
0x05C	0000	01	0111 00	miss
0x1A0	0001	10	0000	miss
0xAD8	1010	11	1000	miss

An Example of Direct Mapped Cache cont'd

- Initially, the cache is empty and all valid bits are 0.

Slot	tag	valid	data
0		0	
1		0	
2		0	
3		0	

Table: 4-slot cache layout

An Example of Direct Mapped Cache cont'd

- First data accessed located at $0x000$.
- Binary address is $0b0000\ 0000\ 0000$.
  - Offset is $0b00\ 0000$
  - cache line beginning address is $0x000$.
  - Index is $0b00$, i.e., slot id is 0
  - Tag is $0b0000$ or $0x0$
- Slot 0 does not have data, a cache miss to slot 0.

Slot	tag	valid	data
0	$0x0$	1	64B from $0x000$
1		0	
2		0	
3		0	

Table: 4-slot cache layout

An Example of Direct Mapped Cache cont'd

- Second data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - cache line beginning address is $0x100$.
  - Index is $0b00$, i.e., slot id is 0
  - Tag is $0b0001$ or $0x1$
- Slot 0 has tag $0x0$, which does match tag $0x1$, thus a cache miss to slot 0.
  - Data from $0x000$ will be evicted and replaced with data from $0x100$.

Slot	tag	valid	data
0	$0x1$	1	64B from $0x100$
1		0	
2		0	
3		0	

Table: 4-slot cache layout

- Note that, the cache still have empty slots, but we are already seeing cache evictions.

An Example of Direct Mapped Cache cont'd

- 3rd data accessed locate at $0x0$, or $0b0000\ 0000\ 0000$, or 0.
  - Offset is $0b00\ 0000$
  - cache line beginning address is $0x000$.
  - Index is $0b00$, i.e., slot id is 0
  - Tag is $0b0000$ or $0x0$
- Slot 0 has tag $0x1$, which does match tag $0x0$, thus a cache miss to slot 0.
  - Data from $0x100$ will be evicted and replaced with data from $0x000$.

Slot	tag	valid	data
0	$0x0$	1	64B from $0x000$
1		0	
2		0	
3		0	

Table: 4-slot cache layout

- Again, a cache eviction despite empty slots.

An Example of Direct Mapped Cache cont'd

- 4th data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - cache line beginning address is $0x100$.
  - Index is $0b00$, i.e., slot id is 0
  - Tag is $0b0001$ or $0x1$
- Slot 0 has tag $0x0$, which does match tag $0x1$, thus a cache miss to slot 0.
  - Data from $0x000$ will be evicted and replaced with data from $0x100$.

Slot	tag	valid	data
0	$0x1$	1	64B from $0x100$
1		0	
2		0	
3		0	

Table: 4-slot cache layout

- Third eviction while the cache has empty slots. Acctually, the main problem with Direct Mapped Cache is that it may experiences high cache misses despite the cache still have many empty slots.

An Example of Direct Mapped Cache cont'd

- 5th data accessed locate at $0x05c$, or $0b0000\ 0101\ 1100$, or 92.
  - Offset is $0b01\ 1100$
  - cache line beginning address is $0x040$.
  - Index is $0b01$, i.e., slot id is 1
  - Tag is $0b0000$ or $0x0$
- Slot 1 does not has valid data, thus a cache miss to slot 1.

Slot	tag	valid	data
0	$0x1$	1	64B from $0x100$
1	$0x0$	1	64B from $0x040$
2		0	
3		0	

Table: 4-slot cache layout

An Example of Direct Mapped Cache cont'd

- 6th data accessed locate at $0x1a0$, or $0b0001\ 1010\ 0000$, or 416.
  - Offset is $0b10\ 0000$
  - cache line beginning address is $0x180$.
  - Index is $0b10$, i.e., slot id is 2
  - Tag is $0b0001$ or $0x1$
- Slot 2 does not has valid data, thus a cache miss to slot 2.

Slot	tag	valid	data
0	$0x1$	1	64B from $0x100$
1	$0x0$	1	64B from $0x040$
2	$0x1$	1	64B from $0x180$
3		0	

Table: 4-slot cache layout

An Example of Direct Mapped Cache cont'd

- 7th data accessed locate at $0xad8$, or $0b1010\ 1101\ 1000$, or 2776.
  - Offset is $0b01\ 1000$
  - cache line beginning address is $0xac0$.
  - Index is $0b11$, i.e., slot id is 3
  - Tag is $0b1010$ or $0xa$
- Slot 3 does not has valid data, thus a cache miss to slot 3.

Slot	tag	valid	data
0	$0x1$	1	64B from $0x100$
1	$0x0$	1	64B from $0x040$
2	$0x1$	1	64B from $0x180$
3	$0xa$	1	64B from $0xac0$

Table: 4-slot cache layout

Pros and Cons of Direct Mapped Cache

- Pros:
  - Very easy and fast to compute the frame/slot index given an address.
  - Fast index calculation allows quickly determining if there is a hit or miss.
  - Easy computation also simplify the hardware implementation.
- Cons:
  - Can have high cache miss rate even if there are still many empty slots in the cache.
    - In the previous example, we have 100% miss rate.
    - Certain memory access patterns may always experience cache misses.

Fully Associative Caches

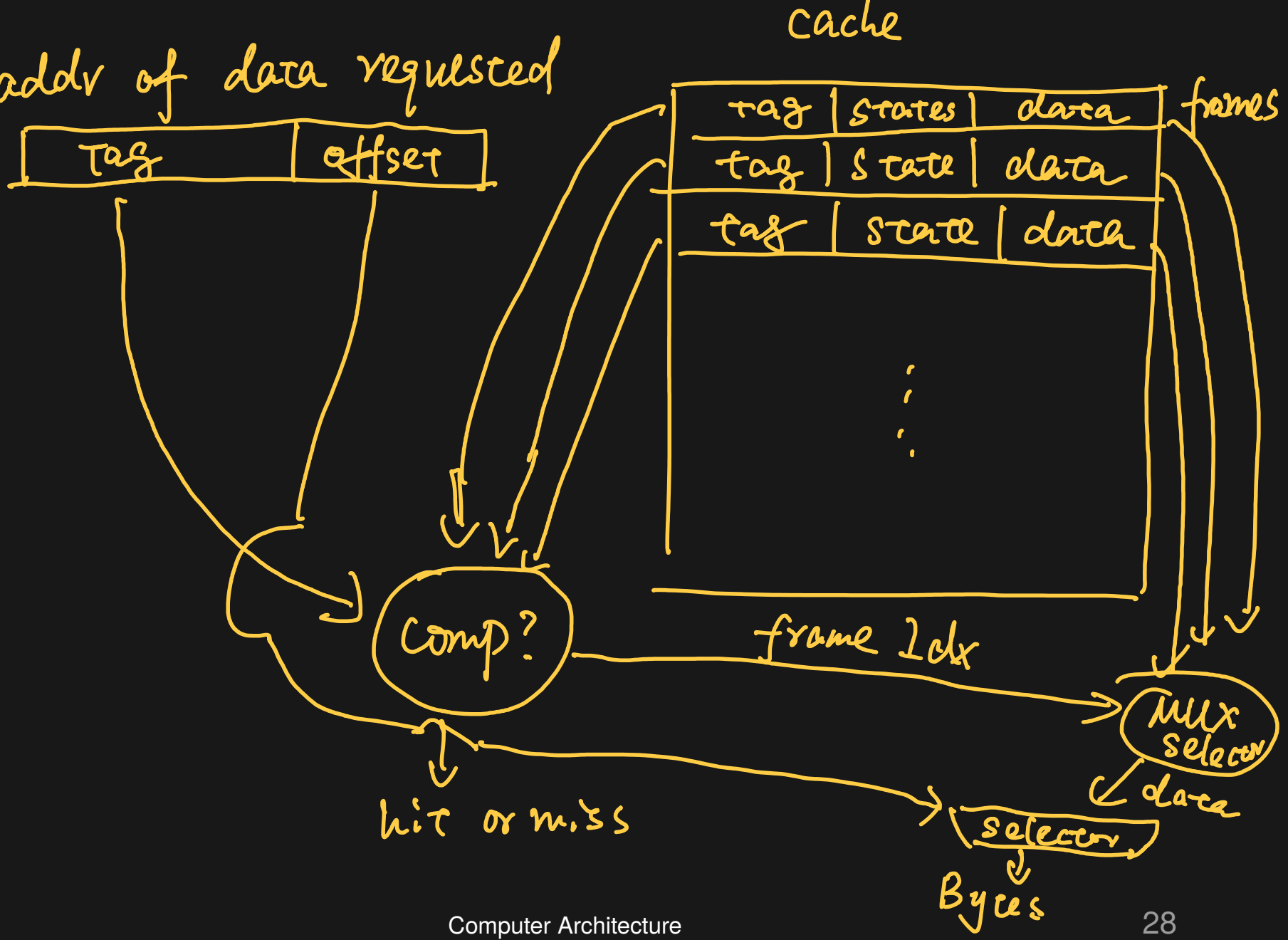
- In Fully Associative Caches, a cache line can be mapped into any slot. / frame
- When a cache miss happens for data at address *A* (with cache line index of *cidx*),
  - If there is empty slot in the cache, use the first empty slot to store cache line of *cidx*.
  - If there is no empty slot, a cache replacement policy is used to select a non-empty cache line to store the cache line of *cidex*.
    - The old data at the select non-empty slot will be evicted. This evicted cache line is called a victim.
    - Typical cache replacement policy will select the oldest or least used cache slot to store.
    - More on cache replacement policy will be discussed in this lecture.

Fully Associative Caches cont'd

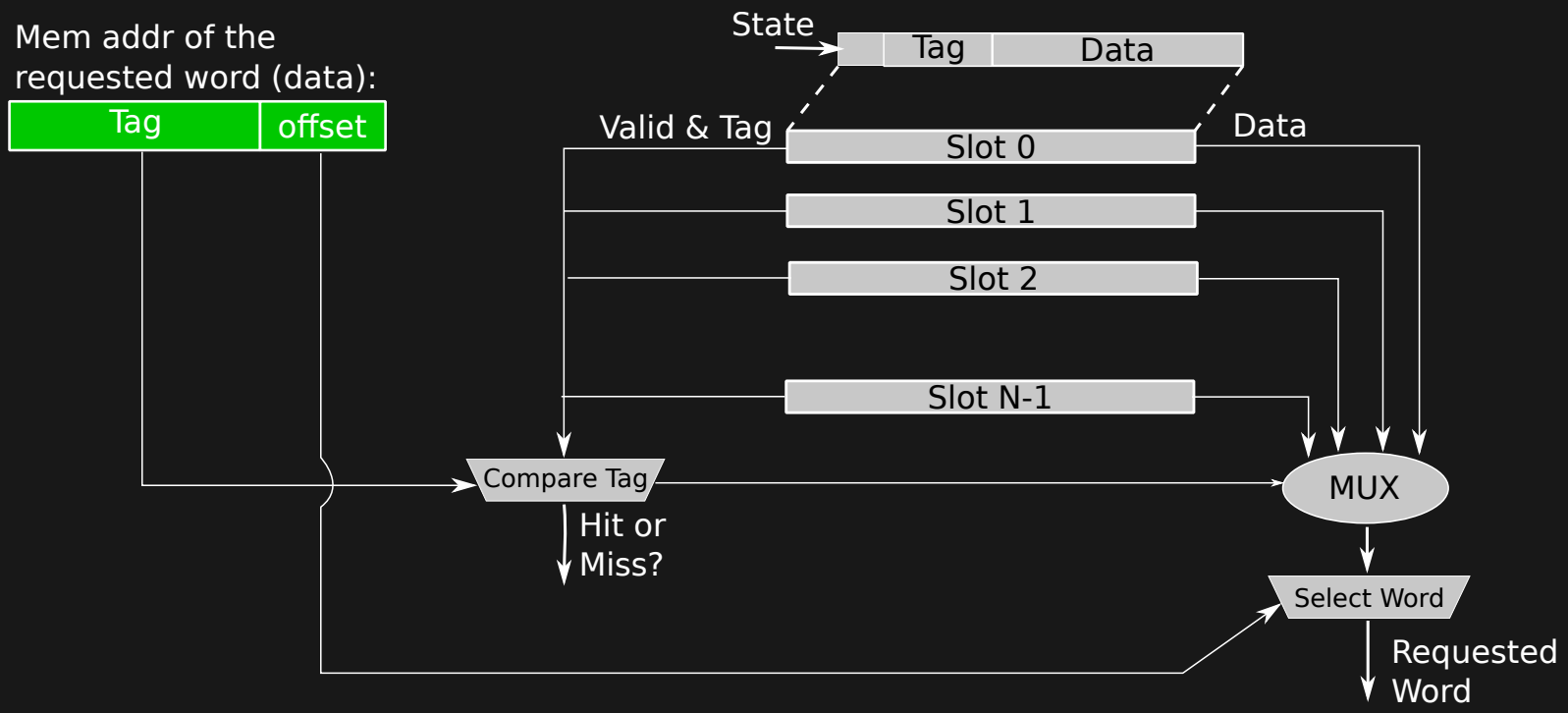
- Since a piece of data can be stored in any frame/slot, the memory is not used for calculating the frame/slot index.
- Consequently, a memory address is only partitioned into two parts: tag and offset.

- E.g., for address 0x081 ($0b0000\ 1000\ 00001$), the offset is $0b00\ 0001$, while the tag is $0b0000\ 00$.

Full Associative Cache Organization



Full Associative Cache Organization



- For each data access, all tags of all slots have to be read out to compare with the tag of the requested data's tag.

An Example of Fully Associative Cache

- $2^2 = 4$ locations in our cache (frames)
- Cache line size = 64 bytes
- References to memory addresses (same as the example for Direct Mapped Cache):
  - 0x000, 0x108 (264), 0x000, 0x108 (264), 0x05C (92), 0x1A0 (416), 0xAD8 (2776)

Slot	tag	valid	data
0	101011	1	Data CL 101011
1	000100	1	Data CL4
2	000001	1	Data CL1
3	000110	1	Data CL6

Table: 4-slot cache layout

Addr	binary	frame idx	hit/miss
0x000	0000 0000 0000	0	miss
0x108	0001 0000 1000	1	miss
0x000	0000 0000 0000	0	hit
0x108	0001 0000 1000	1	hit
0x05C	0000 0101 1100	2	miss

addr	bin	frm	hit/miss
0x1A0	0001 1010 0000	3	mis
0xAD8	1010 1101 1000	0	miss

An Example of Fully Associative Cache cont'd

- Initially, the cache is empty and all valid bits are 0.

Slot	tag	valid	data
0		0	
1		0	
2		0	
3		0	

Table: 4-slot cache layout

An Example of Full Associative Cache cont'd

- First data accesse located at $0x000$.
- Binary address is $0b0000\ 0000\ 0000$.
  - Offset is $0b00\ 0000$
  - cache line beginning address is $0x000$.
  - Tag is $0b0000\ 00$ or $0x00$
- Cache is empty, a cache miss occurs.
- Pick the first empty slot, slot 0, to store this cache line.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1		0	
2		0	
3		0	

Table: 4-slot cache layout

An Example of Fully Cache cont'd

- Second data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - cache line beginning address is $0x100$.
  - Tag is $0b0001\ 00$ or $0x04$
- No slot has tag $0x04$, a cache miss occurs.
- Pick the first empty slot, slot 1, to store this cache line.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1	$0x04$	1	64B from $0x100$
2		0	
3		0	

Table: 4-slot cache layout

An Example of Fully Associative Cache cont'd

- 3rd data accessed locate at $0x0$, or $0b0000\ 0000\ 0000$, or 0.
  - Offset is $0b00\ 0000$
  - cache line beginning address is $0x000$.
  - Tag is $0b0000\ 00$ or $0x00$
- Slot 0 has tag $0x0$, thus a cache hit to slot 0.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1	$0x04$	1	64B from $0x100$
2		0	
3		0	

Table: 4-slot cache layout

- Unless Direct Mapped Cache, this 3rd access is a cache hit to slot 0.

An Example of Fully Associative Cache cont'd

- 4th data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - cache line beginning address is $0x100$.
  - Tag is $0b0001\ 00$ or $0x04$
- Slot 1 has tag $0x04$, thus a cache hit to slot 1.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1	$0x04$	1	64B from $0x100$
2		0	
3		0	

Table: 4-slot cache layout

- Again, a cache hit, unlike Direct Mapped Cache.

An Example of Fully Associative Cache cont'd

- 5th data accessed locate at $0x05c$, or $0b0000\ 0101\ 1100$, or 92.
  - Offset is $0b01\ 1100$
  - cache line beginning address is $0x040$.
  - Tag is $0b0000\ 01$ or $0x01$
- No slot has tag $0x01$, a cache miss occurs.
- Pick the first empty slot, slot 2, to store this cache line.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1	$0x04$	1	64B from $0x100$
2	$0x01$	1	64B from $0x040$
3		0	

Table: 4-slot cache layout

An Example of Full Associative Cache cont'd

- 6th data accessed locate at $0x1a0$, or $0b0001\ 1010\ 0000$, or 416.
  - Offset is $0b10\ 0000$
  - cache line beginning address is $0x180$.
  - Tag is $0b0001\ 10$ or $0x06$
- No slot has tag $0x06$, a cache miss occurs.
- Pick the first empty slot, slot 3, to store this cache line.

Slot	tag	valid	data
0	$0x00$	1	64B from $0x000$
1	$0x04$	1	64B from $0x100$
2	$0x01$	1	64B from $0x040$
3	$0x06$	1	64B from $0x180$

Table: 4-slot cache layout

An Example of Full Associative Cache cont'd

- 7th data accessed locate at $0xad8$, or $0b1010\ 1101\ 1000$, or 2776.
  - Offset is $0b01\ 1000$
  - cache line beginning address is $0xac0$.
  - Tag is $0b1010\ 11$ or $0x2b$
- Not slot has tag $0x2b$, thus a cache miss.
- Pick the oldest slot, slot 0, to store this cache line.

Slot	tag	valid	data
0	$0x2b$	1	64B from $0xac0$
1	$0x04$	1	64B from $0x100$
2	$0x01$	1	64B from $0x040$
3	$0x06$	1	64B from $0x180$

Table: 4-slot cache layout

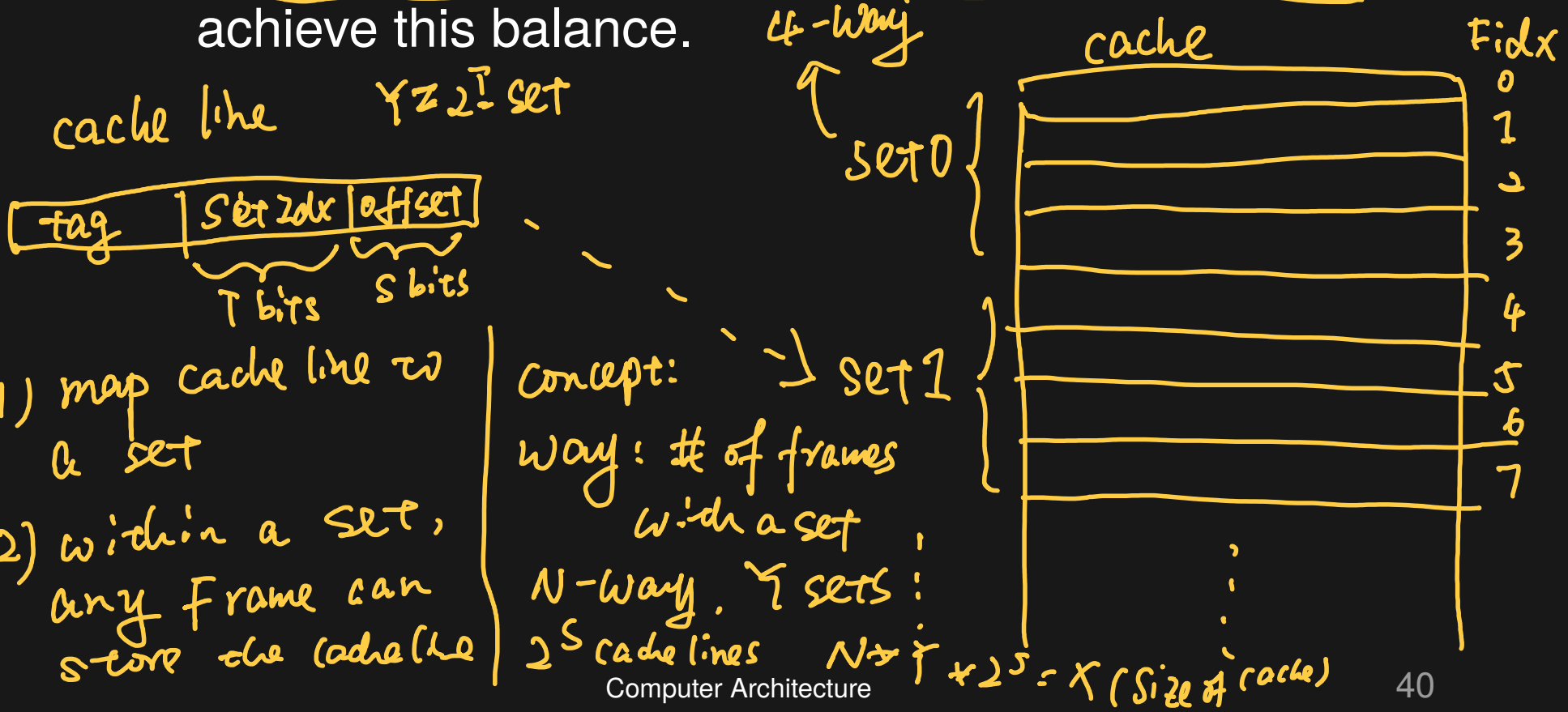
- Here we use a very simple replacement policy: choosing the slot with the oldest cache line.

Pros and Cons of Fully Associative Cache

- Pros:
  - Much fewer cache misses than other cache placement policies. A cache miss only occurs when the cache is not large enough to hold all of the working set.
    - In the previous example, there are two cache hits, whereas in the Direct Mapped Cache, there are no cache hits at all.
- Cons:
  - It takes a long time to compare the tags of all slots.
    - Tags can be checked in parallel, but it requires considerably more transistors to implement parallel tag comparison logics.
    - This extra time is required even for cache hits. The increased hit time would degrade the overall cache performance.
  - Control logic is more complex since all slots have to be activated and selected for every data access.

Set Associative Caches

- Directed Mapped cache has too many misses, Fully Associative caches are too slow. Can we strike a balance between miss rates and cache speeds?
- Set Associative caches combines the ideas from both directed mapped cache and fully associative cache to achieve this balance.

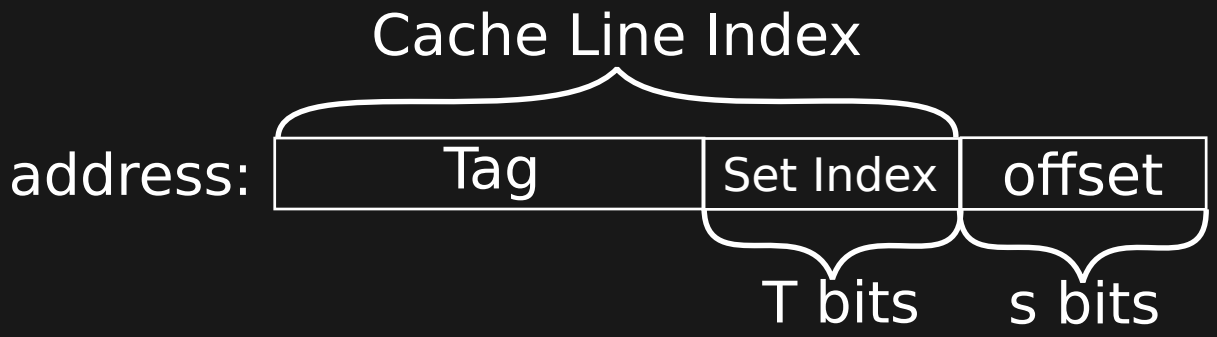


Set Associative Caches

- Directed Mapped cache has too many misses, Fully Associative caches are too slow. Can we strike a balance between miss rates and cache speeds?
- Set Associative caches combines the ideas from both directed mapped cache and fully associative cache to achieve this balance.
  - A cache is partition into sets of slots, and the set id for a memory address can be directly determined similarly as Direct Mapped cache.
  - Within a set, there are several fully associated slots. A cache line mapped to this set can be stored in any these slots.
- A *N*-way associative cache is a Set Associative cache with *N* slots/ways per set.
  - The number of sets depends on the cache size and *N*. For a cache with *X* bytes, there are $\frac{X}{2^s \cdot N}$ sets, where 2^s is the cache line size.

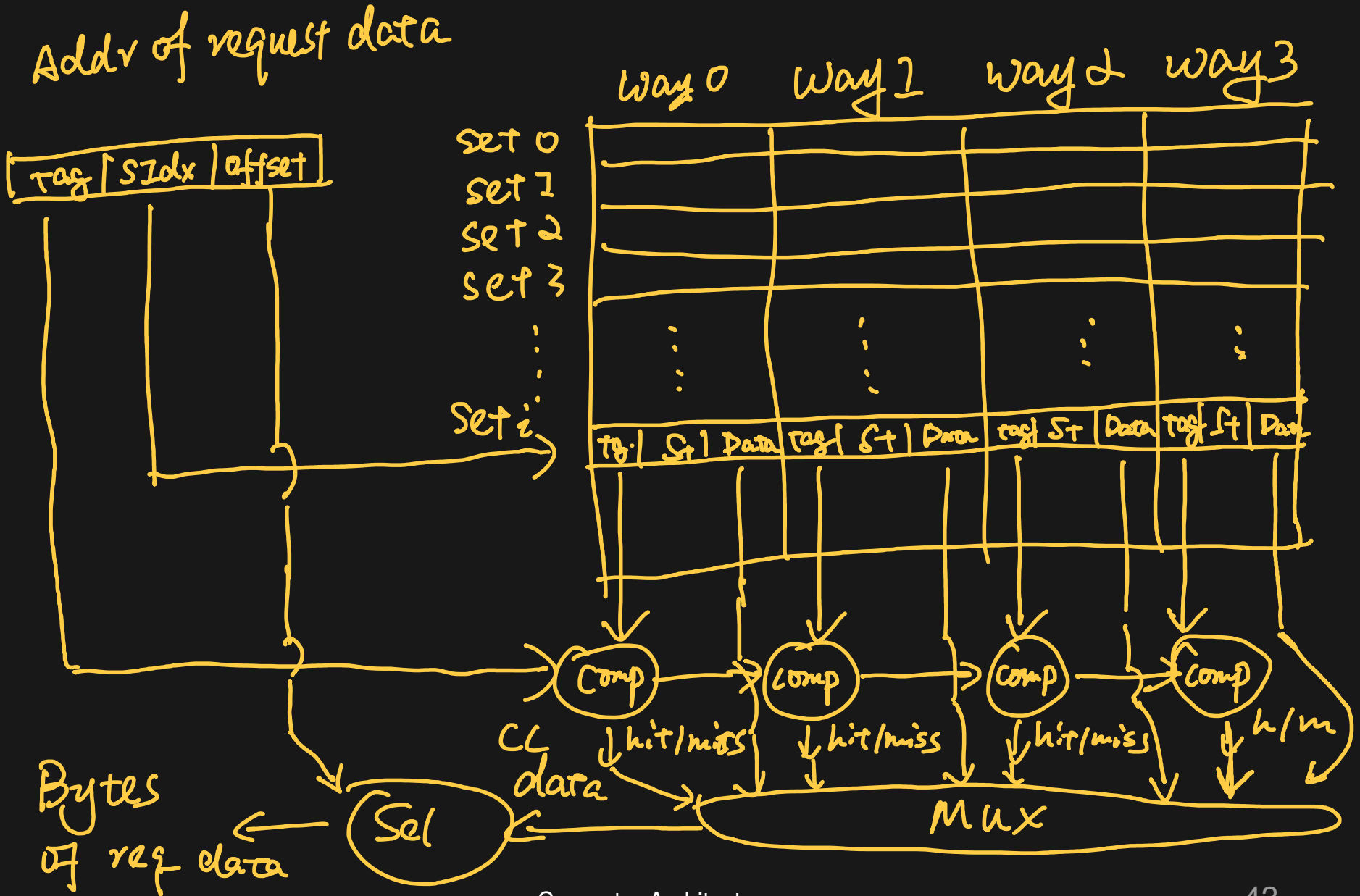
Set Associative Caches cont'd

- Similar to Direct Mapping cache, a few bits in the memory address is used to determine the corresponding set index.
- Consequently, a memory address is only partitioned into three parts: tag, set index and offset.
  - If there are 2^T sets, then there are T bits for set index.

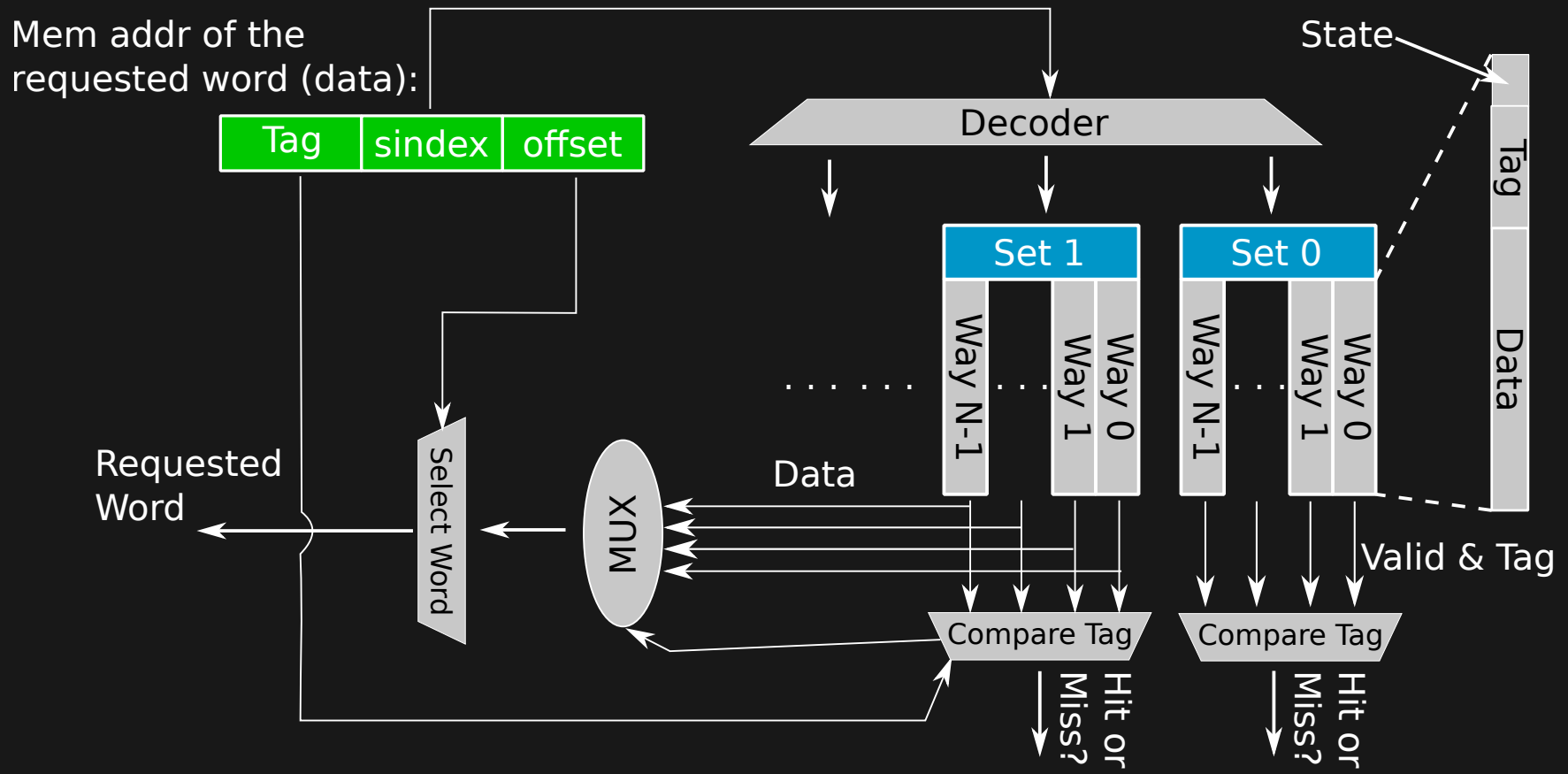


- E.g., If there are two (2^1) sets, there is only one bit for set index. For address 0x081 ($0b0000\ 1000\ 00001$), the offset is $0b00\ 0001$, the set index is 0, and the tag is $0b0000\ 00$.

Set Associative Cache Organization



Set Associative Cache Organization



Set Associative Cache Organization cont'd

- The set index of a memory address is used to identify the set that may storing the cache line of this address. The tags of all ways (slots) in this set will be compared to identify the actual slot with the request data.

An Example of Set Associative Cache

2 set ⇒ 2^1 set ⇒ 1 bit for set idx

- 2-Way set associative cache with 2 sets. So there are 4 slots in total
- Cache line size = 64 bytes
- References to memory addresses (same as the example for Direct Mapped Cache):
  - 0x000, 0x108 (264), 0x000, 0x108 (264), 0x05C (92), 0x1A0 (416), 0xAD8 (2776)

Set	Way	tag	val	data
0	0	00011	1	DataCL6
0	1	00010	1	DataCL4
1	0	00000	1	DataCL1
1	1	10101	1	DataCL 101011

Table: 4-slot cache layout

addr	binary	set / way	hit / miss
0x000	0000 0000 0000	0 / 0	miss
0x108	0001 0000 1000	0 / 1	miss
0x000	0000 0000 0000	0 / 0	hit
0x108	0001 0000 1000	0 / 1	hit
0x05C	0000 0101 1100	1 / 0	miss
0x1A0	0001 1010 0000	0 / 0	miss
0xAD8	1010 1101 1000	1 / 1	miss

An Example of Set Associative Cache cont'd

- Initially, the cache is empty and all valid bits are 0.

Set	Way	tag	valid	data
0	0		0	
0	1		0	
1	0		0	
1	1		0	

Table: 4-slot cache layout

An Example of Set Associative Cache cont'd

- First data accesse located at $0x000$.
- Binary address is $0b0000\ 0000\ 0000$.
  - Offset is $0b00\ 0000$
  - cache line starts at $0x000$.
  - Set index is 0. Tag is $0b0000\ 0$ or $0x00$
- Cache is empty, a cache miss occurs.
- Pick the first empty slot, way 0, in set 0 to store this cache line.

Set	Way	tag	valid	data
0	0	$0x00$	1	64B@$0x000$
0	1		0	
1	0		0	
1	1		0	

Table: 4-slot cache layout

An Example of Set Associative Cache cont'd

- 2nd data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - Cache line starts at $0x100$.
  - Set index is 0, Tag is $0b0001\ 0$ or $0x02$
- No slot in set 0 as tag $0x02$, a cache miss occurs.
- Pick the first empty slot, way 1, in set 0 to store this cache line.

Set	Way	tag	valid	data
0	0	$0x00$	1	64B@ $0x000$
0	1	$0x02$	1	64B@$0x100$
1	0		0	
1	1		0	

Table: 4-slot cache layout

An Example of Set Associative Cache cont'd

- 3rd data accessed locate at $0x0$, or $0b0000\ 0000\ 0000$, or 0.
  - Offset is $0b00\ 0000$
  - Cache line starts at $0x000$.
  - Set index is 0; tag is $0b0000\ 0$ or $0x00$
- Way 0 of set 0 has tag $0x0$, thus a cache hit.

Set	Way	tag	valid	data
0	0	$0x00$	1	64B@$0x000$
0	1	$0x02$	1	64B@$0x100$
1	0		0	
1	1		0	

Table: 4-slot cache layout

- Unless direct mapped cache, this 3rd access is a cache hit to set 0 way 0. However, locating the set index is as fast as direct mapped cache.

An Example of Set Associative Cache cont'd

- 4th data accessed locate at $0x108$, or $0b0001\ 0000\ 1000$, or 264.
  - Offset is $0b00\ 1000$
  - cache line starts at $0x100$.
  - Set index is 0; tag is $0b0001\ 0$ or $0x02$
- Set 0 way 1 has tag $0x04$, thus a cache hit.

Set	Way	tag	valid	data
0	0	$0x00$	1	64B@$0x000$
0	1	$0x02$	1	64B@$0x100$
1	0		0	
1	1		0	

Table: 4-slot cache layout

- Again, a cache hit like fully associative cache. And a fast set index determination as direct mapped cache.

An Example of Set Associative Cache cont'd

- 5th data accessed locate at $0x05c$, or $0b0000\ 0101\ 1100$, or 92.
  - Offset is $0b01\ 1100$
  - Cache line starts at $0x040$.
  - Set index is 1, tag is $0b0000\ 0$ or $0x00$
- No ways/slots in Set 1 has tag $0x01$, a cache miss occurs.
- Pick the first empty slot, way 0, in set 1 to store this cache line.

Set	Way	tag	valid	data
0	0	$0x00$	1	64B@$0x000$
0	1	$0x02$	1	64B@$0x100$
1	0	$0x00$	1	64B@$0x040$
1	1		0	

Table: 4-slot cache layout

- Note that way 0 of set 0 also has tag $0x00$. As long as they are in different set, they will never be accessed at the same time.

An Example of Set Associative Cache cont'd

- 6th data accessed locate at $0x1a0$, or $0b0001\ 1010\ 0000$, or 416.
  - Offset is $0b10\ 0000$
  - Cache line starts $0x180$.
  - Set index is 0; tag is $0b0001\ 1$ or $0x03$
- No slot in set 0 has tag $0x06$, a cache miss occurs.
- Pick the oldest slot, way 0, in set 0 to store this cache line.

Set	Way	tag	valid	data
0	0	$0x03$	1	64B@$0x180$
0	1	$0x04$	1	64B@$0x100$
1	0	$0x01$	1	64B@$0x040$
1	1		0	

Table: 4-slot cache layout

- Unlike fully associative cache, this 6th access caused a cache eviction when there is still empty space in the cache.

An Example of Full Associative Cache cont'd

- 7th data accessed locate at $0xad8$, or $0b1010\ 1101\ 1000$, or 416.
  - Offset is $0b01\ 1000$
  - Cache line starts at $0xac0$.
  - Set index is 1; tag is $0b1010\ 1$ or $0x15$
- Not slot in set 1 has tag $0x15$, thus a cache miss.
- Pick the empty slot, way 1, in set 1, to store this cache line.

Set	Way	tag	valid	data
0	0	$0x03$	1	64B@$0x180$
0	1	$0x04$	1	64B@$0x100$
1	0	$0x01$	1	64B@$0x040$
1	1	$0x15$	1	64B@$0xac0$

Table: 4-slot cache layout

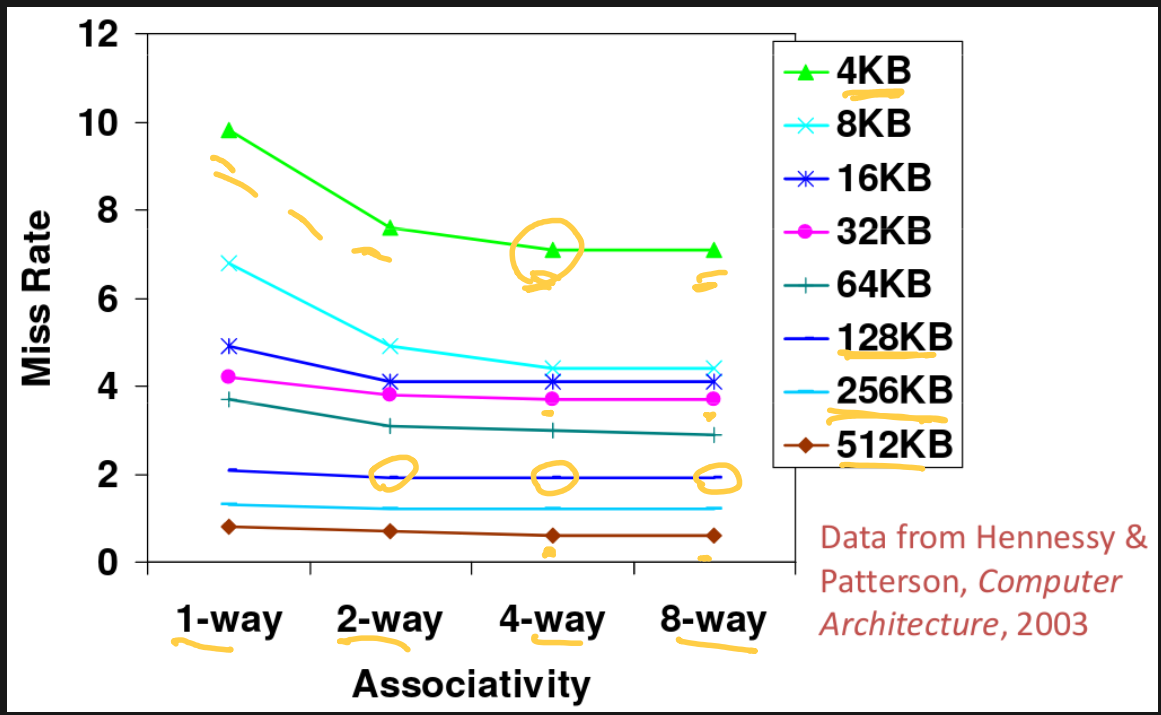
Pros and Cons of Set Associative Cache

- Pros:
  - Fewer cache misses than direct mapped caches.
    - In the previous example, there are two cache hits, similar as fully associative cache.
  - Simpler implementation and Faster than fully associative cache.
  - Set associative cache is a trade off between cache miss rate and implementation complexity/latency.
    - More ways => lower cache miss rates, but higher complexity and higher latency.
    - A direct mapped cache is essentially a 1-way set associative cache. A fully associative cache is essentially a 1-set set associative cache with all lines as its ways.
- Cons:
  - No much of a con, but it is different to determine the best number of ways.

# of ways # of sets

Determining the Best Number of Ways

- Like all engineering parameters, there is no one number works well for all applications.
- Benchmarks can help us to determine the averagely best number of ways for a special use case.
- Cache miss rates for different numbers of ways for SPEC2000 benchmarks,



Determining the Best Number of Ways cont'd

- Raw data for the above figure:

Cache Size (KB)	Associativity			
	1-Way	2-Way	4-Way	8-Way
4	0.098	0.076	0.071	0.071
8	0.068	0.049	0.044	0.044
16	0.049	0.041	0.041	0.041
32	0.042	0.038	0.037	0.037
64	0.037	0.031	0.030	0.029
128	0.021	0.019	0.019	0.019
256	0.013	0.012	0.012	0.012
512	0.008	0.007	0.006	0.006

AMAT

Determining the Best Number of Ways cont'd

- Note in the above figure, the largest gains are from direct mapped (1-way) cache to 2-way set associative. The improvement is limited after 4 ways.
  - Therefore, we usually don't need a very high number of ways to achieve nearly lowest miss rates.
- Also, cache miss rates also depend heavily on cache size (as expected).
  - Consequently, for a new CPU design with a new cache size, we need to conduct new experiments with benchmarks to determine the number of ways that gives the lowest miss rates.

Determining the Best Number of Ways cont'd

- Another limitation of high number of ways is that more ways means slower cache latency, i.e., higher cache hit latency.
- Recall that Average Memory Access Time (AMAT) depends on both hit latency and miss rates.

$$
\begin{aligned}
AMAT &= Lat_{hit} \times Rate_{hit} + Lat_{miss} \times Rate_{miss} \\
&= Lat_{hit} \times Rate_{hit} + (Lat_{hit} + Miss\_Penalty) \times Rate_{miss} \\
&= Lat_{hit} \times (Rate_{hit} + Rate_{miss} + Miss\_Penalty \times Rate_{miss} \\
&= Lat_{hit} \times 100\% + Miss\_Penalty \times Rate_{miss} \\
&= Lat_{hit} + Miss\_Penalty \times Rate_{miss}
\end{aligned}
\tag{1}
$$

- A higher number of ways reduces $Rate_{miss}$ but increases Lat_{hit}, and may eventually increase the overall $AMAT$.

Determining the Best Number of Ways cont'd

- Another example from Computer Architecture: A Quantitative Approach (Figure B.13):
  - Assume a case where one more way means slower cache
    - $Lat_{hit,2-way} = 1.36 \times Lat_{hit,1-way}$
    - $Lat_{hit,4-way} = 1.44 \times Lat_{hit,1-way}$
    - $Lat_{hit,8-way} = 1.52 \times Lat_{hit,1-way}$
  - Also, using the miss rates from slide 56.
  - The AMATs for various cache sizes and ways are,

Cache Size (KB)	Associativity			
	1-Way	2-Way	4-Way	8-Way
4	3.44	3.25	3.22	3.28
8	2.69	2.58	2.55	2.62
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

Determining the Best Number of Ways cont'd

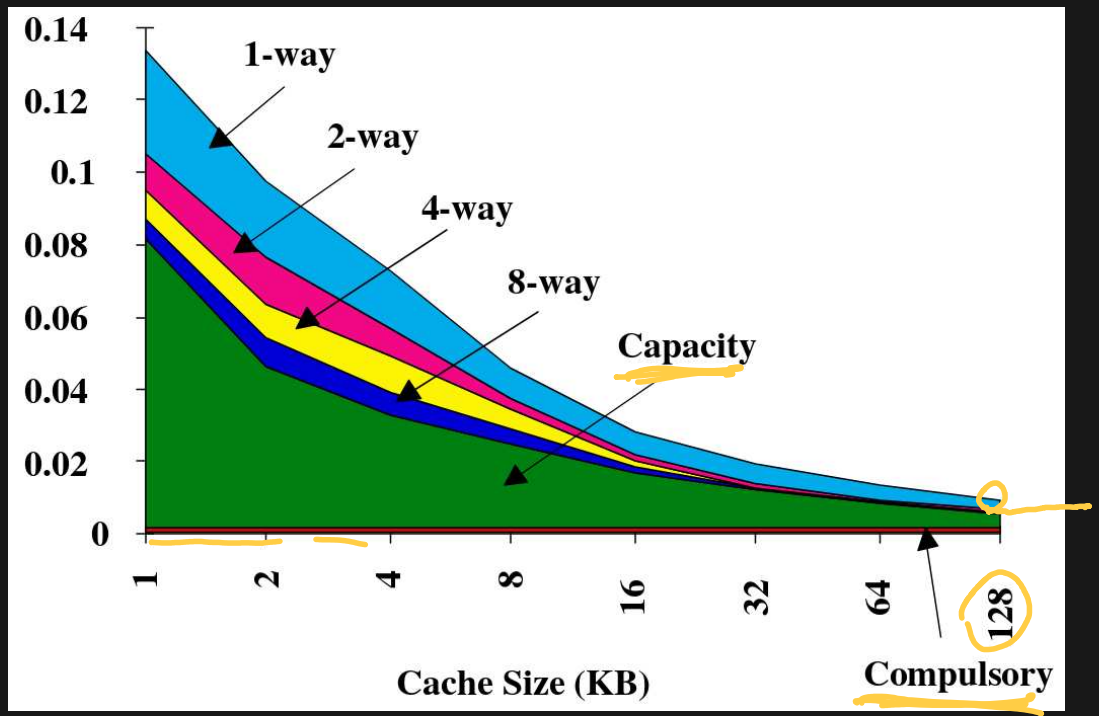
- Note how AMAT deteriorates with more ways. Some times it gets worse than 1-way (direct mapped cache) in the cells with red numbers.
- One last note, these numbers are fairly old. Modern processors usually use 16-way caches as the hit latency does not deteriorates that fast now.

Types of Cache Misses: The Three C's

- Compulsory: On the first access to a cache line; the cache line must be brought into the cache; also called cold start misses, or first reference misses.
- Capacity: Occur because cache lines are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).
- Conflict: In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.
  - The idea is that hits in a fully associative cache that becomes misses in an *N*-way set associative cache are due to more than *N* distinctive requests to the same set.
  - Think the case where the cache has empty slots but there are still evictions.

Types of Cache Misses: The Three C's cont'd

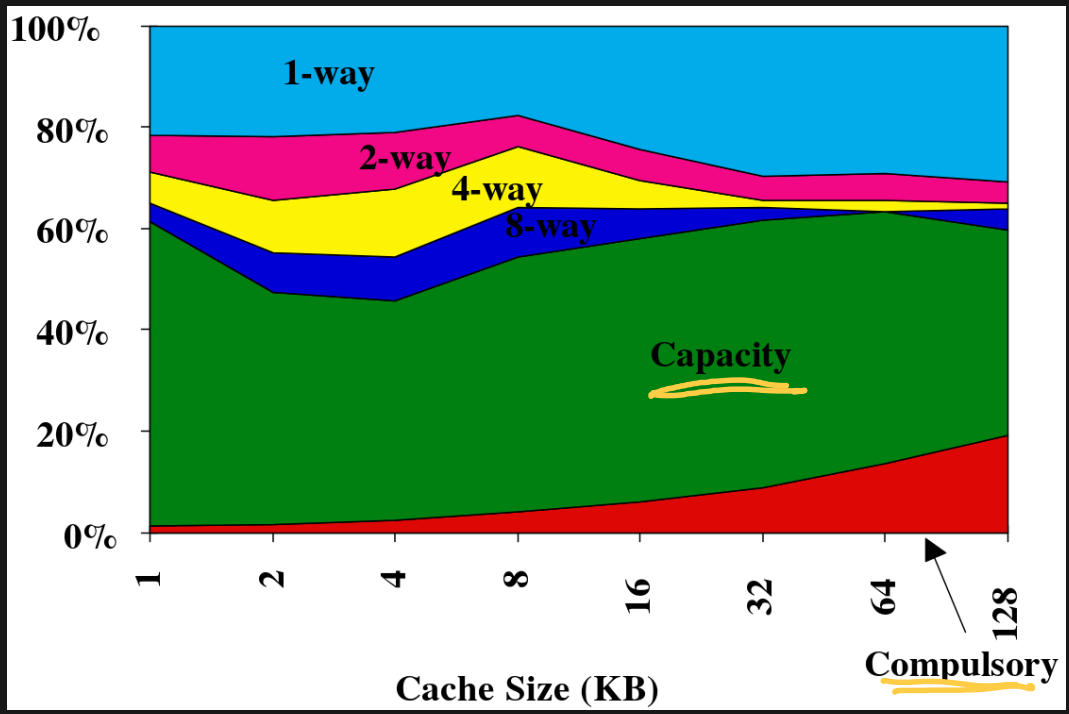
- An illustration of 3C Misses (using SPEC92):



- Based on the figure:
  - Compulsory misses are relatively low, comparing to other two types of misses.
  - Bigger cache => less capacity miss.
  - More ways => less conflict misses.

Types of Cache Misses: The Three C's cont'd

- An other illustration of 3C Misses (using SPEC92):



- Based on the figure:
  - Majority misses are capacity misses.
  - Except for 1-way, conflict misses are relatively less when cache is 32KB or larger.
  - As cache sizes grow, compulsory misses become more important (since other capacity misses are reducing).

Cache Replacement Policies

Cache Replacement Policies

- If a cache miss occurs when the target set is full, one of the ways need to be evicted to store the cache line.
- A cache replacement policy determine which way should e evicted.
- Common cache replacement policies:
  - FIFO, first in first out. We have been using this policy in our examples so far.
  - LRU, least recently used: evict the way with least-recently-used cache line. LRU assumes if a cache line is not frequently accessed, then it is probably not needed in the future.
  - MRU, most recently used: evict the way with the most-recently-used cache line. MRU assumes if a cache line is frequently accessed, then it is probably not needed in the future.
  - And many more.

Cache Replacement Policies cont'd

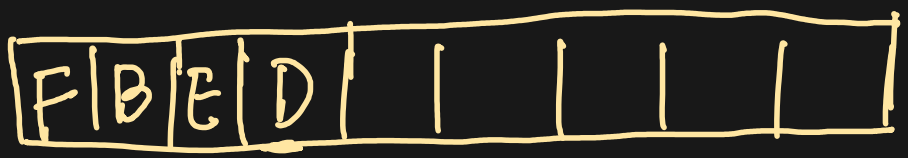
- Which replacement policy works better depends on the application's memory access pattern.
  - Again, the goal here is to minimize cache miss rates while do not increase cache latency and cache complexity.
- Modern processors typically use a policy based on LRU.
  - Non-LRU policies typically do not work well. LRU's assumption fits the memory behaviors of most real applications.
  - However, LRU is hard to implement in hardware.
  - Therefore, modern processors usually use a policy roughly based on LRU. Computer architects also tend to tweak the LRU policy to make it perform better.

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.

m m m m m h m

LRU Q:



Way	Data
0	E
1	B
2	F
3	D

Queue keep track of which cache line has been used recently

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Before accesses start.

LRU Q: 

Way	Data
0	
1	
2	
3	

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access A, assign A to way 0, and add it to LRU queue.

LRU Q:

Way	Data
0	A
1	
2	
3	

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access B, assign B to way 1, and add it to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

B	A		

Way	Data
0	A
1	B
2	
3	

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access C, assign C to way 2, and add it to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

C	B	A	

Way	Data
0	A
1	B
2	C
3	

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access D, assign D to way 3, and add it to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

D	C	B	A

Way	Data
0	A
1	B
2	C
3	D

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, assign E to the least used way. Since A is a the end of the list, A's slot is the least used. Therefore, assign E to way 0. Also, drop A from LRU Q and add E to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

E	D	C	B

Way	Data
0	E
1	B
2	C
3	D

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access B, a hit to way 2. Move B to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

B	E	D	C

Way	Data
0	E
1	B
2	C
3	D

LRU Policy

- Typically LRU policy requires maintaining a queue recording the recent access history of the ways within a set.
  - A queue is required for each cache set.
- For example, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access F, assign F to the least used way. Since C is a the end of the list, A's slot is the least used. Therefore, assign F to way 2. Also, drop C from LRU Q and add F to the beginning of the LRU queue and shift other records towards the end.

LRU Q:

F	B	E	D

Way	Data
0	E
1	B
2	F
3	D

Not-Recently-Used Poilcy and Clock Algorithm

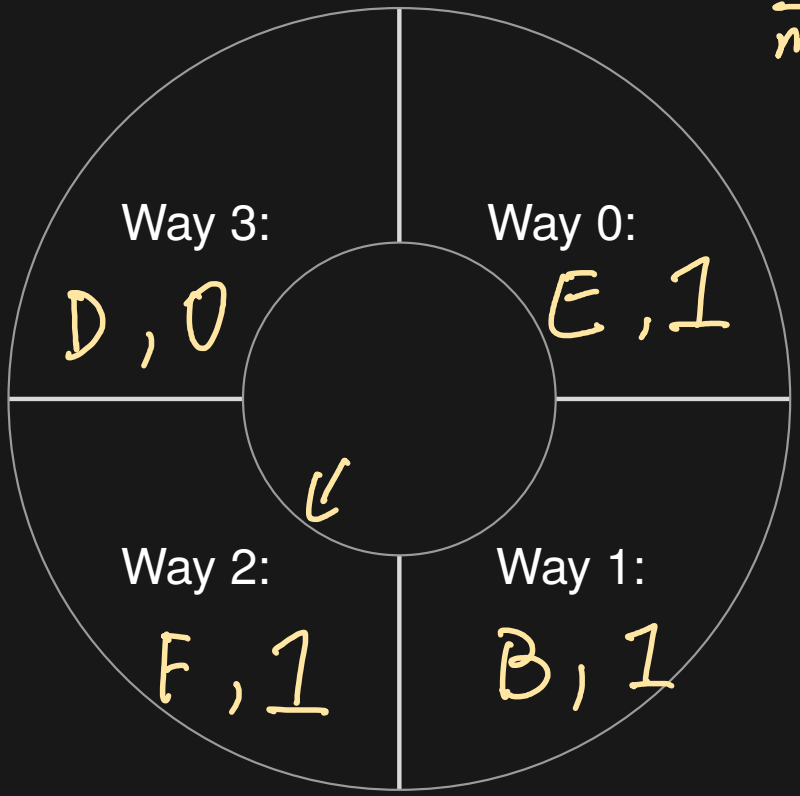
- Implementing an LRU queue with shifting and inserting in the hardware is fairly complex. Therefore, computer architects look for simple implementations that are similar to the behavior of the LRU policy.
- Not-Recently-Used (NRU) policy is approximation of LRU. The Clock Adaptive Replacement (CAR) algorithm is a typical implementation of NRU policy.
  - In CAR, the ways of a set are checked circularly similar to a clock.
  - Each way maintains a used-bit to indicate if it is accessed recently.
  - For a cache hit, set the used bit of the way that gets hit to 1.
  - For a cache miss, visit each way clockwise to find a way that has a used-bit of 0. This way is then the victim for eviction and replacement. During this visit, set every used-bit to 0 if it is 1.

An Example of CAR Algorithm

CRU

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.

A	B	C	D	E	B	F
m	m	m	m	m	H	m

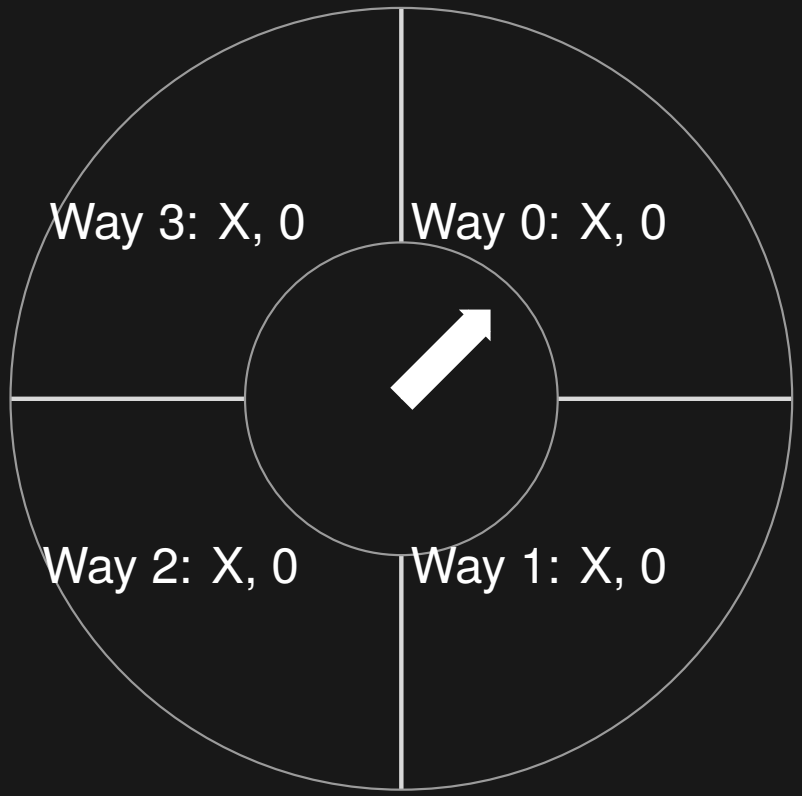


tag | Stat | data | used bit

- 1: used recently
- 0: not-used recently

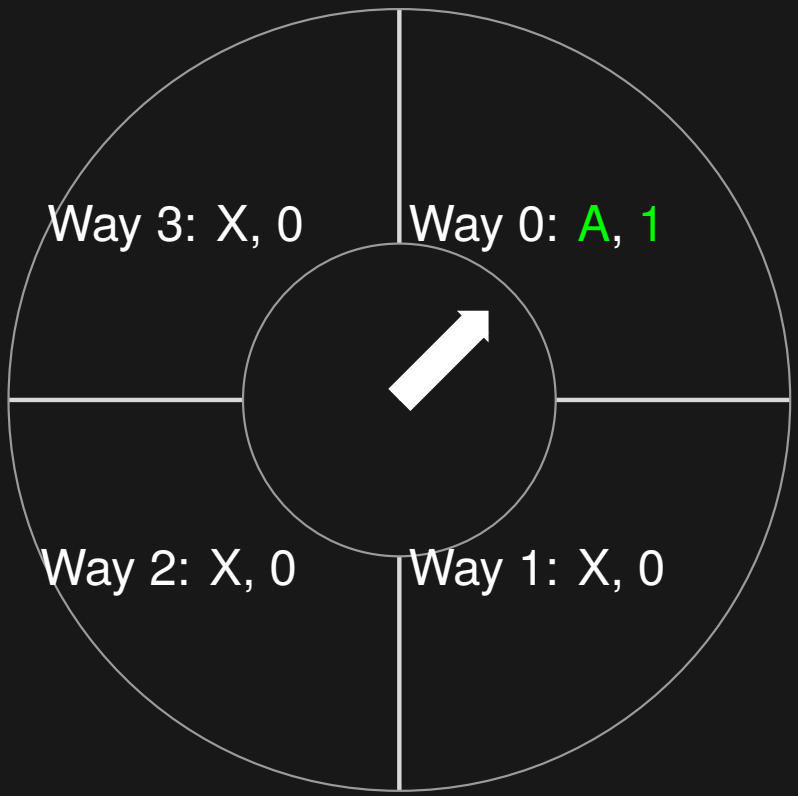
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Before accesses start, clock hand points to way 0. All ways have used-bits of 0.



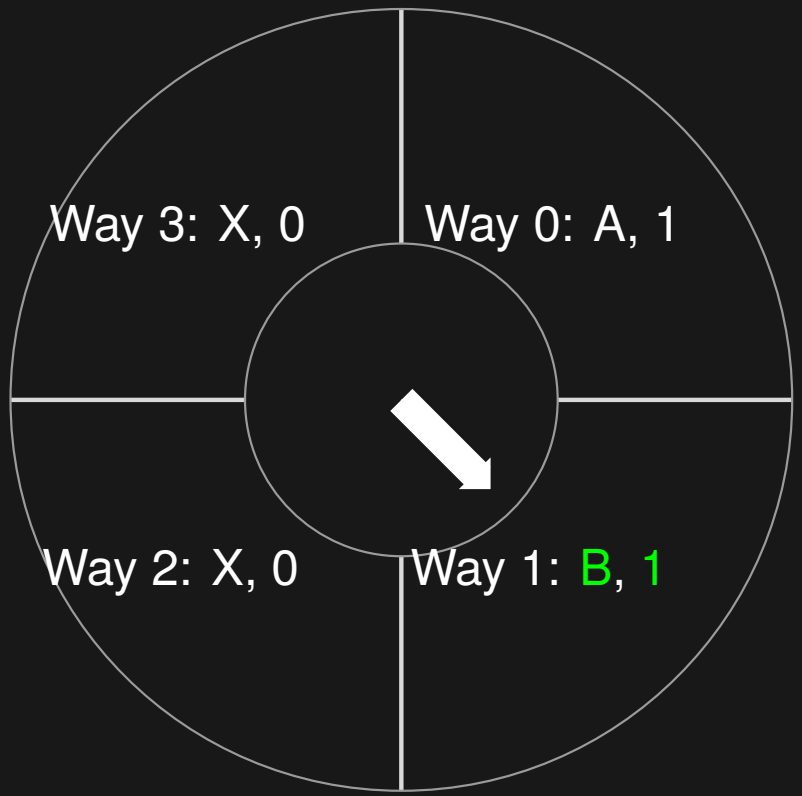
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access A. Find the first way with a used-bit of 0, which is way 0 there. Assign A to way 0, update its used-bit to 1.



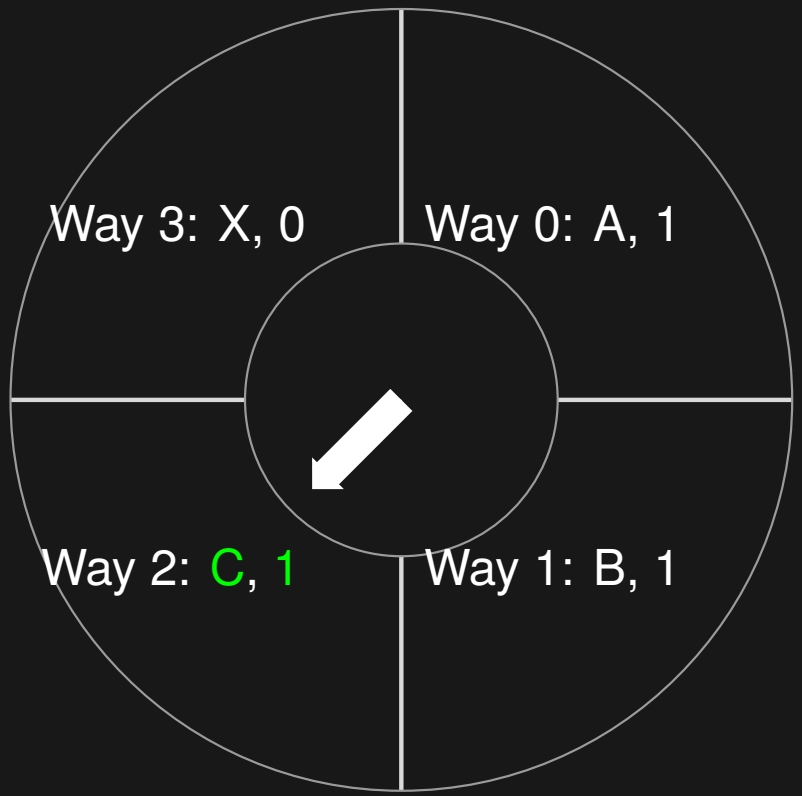
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access B. Find the first way with a used-bit of 0 by moving the clock hand clockwise. Here, way 1 has 0 used-bit. Assign B to way 1, update its used-bit to 1.



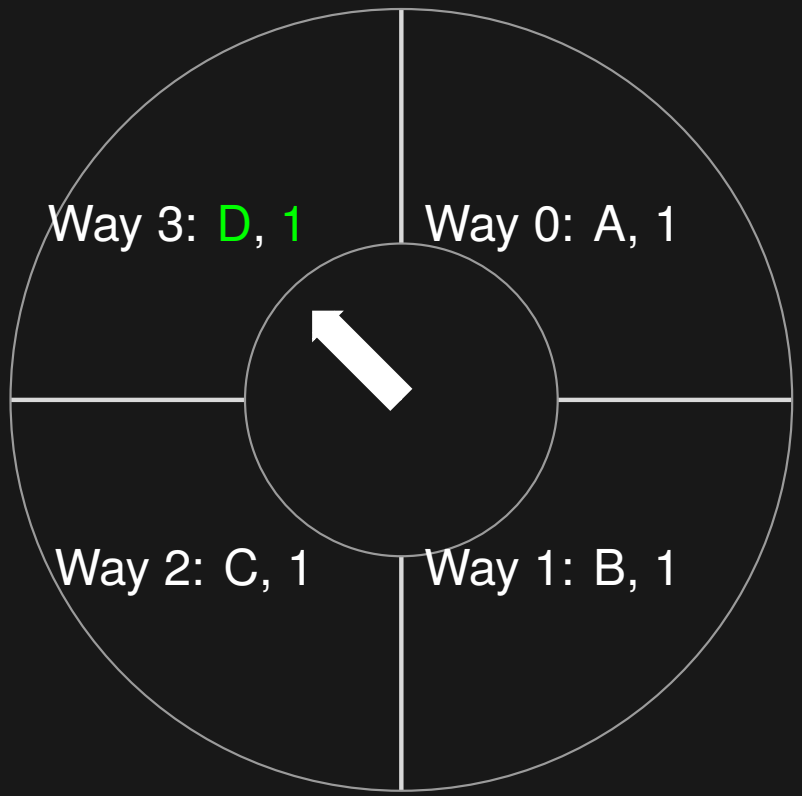
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access C. Find the first way with a used-bit of 0 by moving the clock hand clockwise. Here, way 2 has 0 used-bit. Assign C to way 2, update its used-bit to 1.



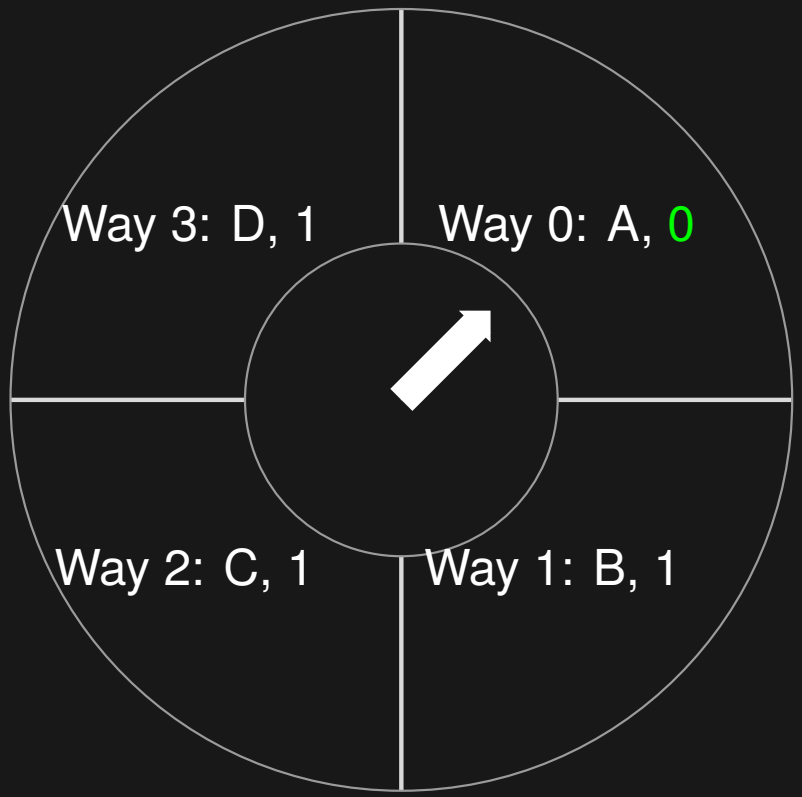
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access D. Find the first way with a used-bit of 0 by moving the clock hand clockwise. Here, way 3 has 0 used-bit. Assign D to way 2, update its used-bit to 1.



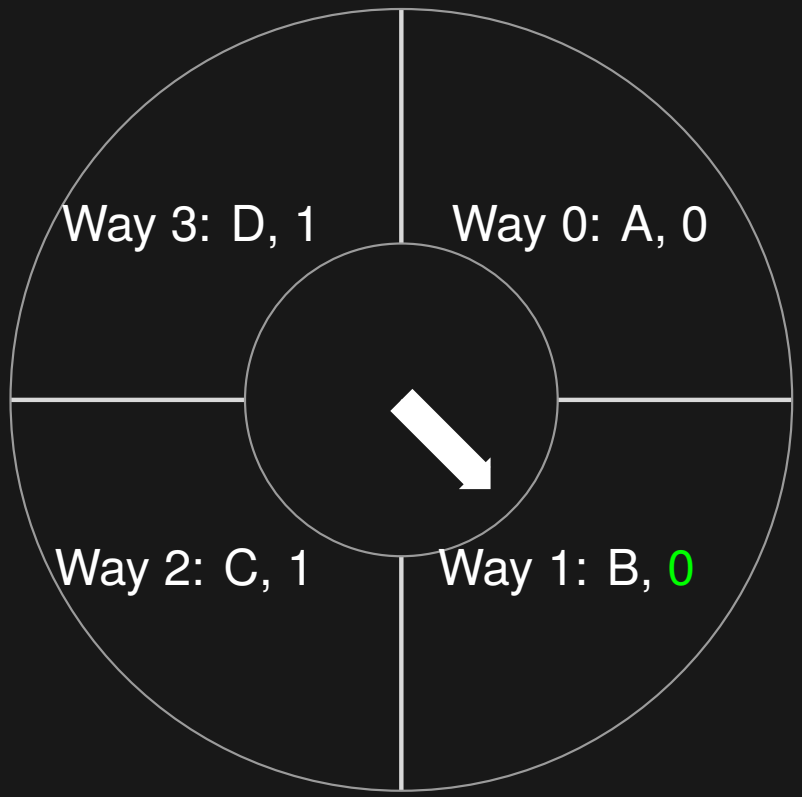
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 0 first. Used-bit is 1, reset it to 0



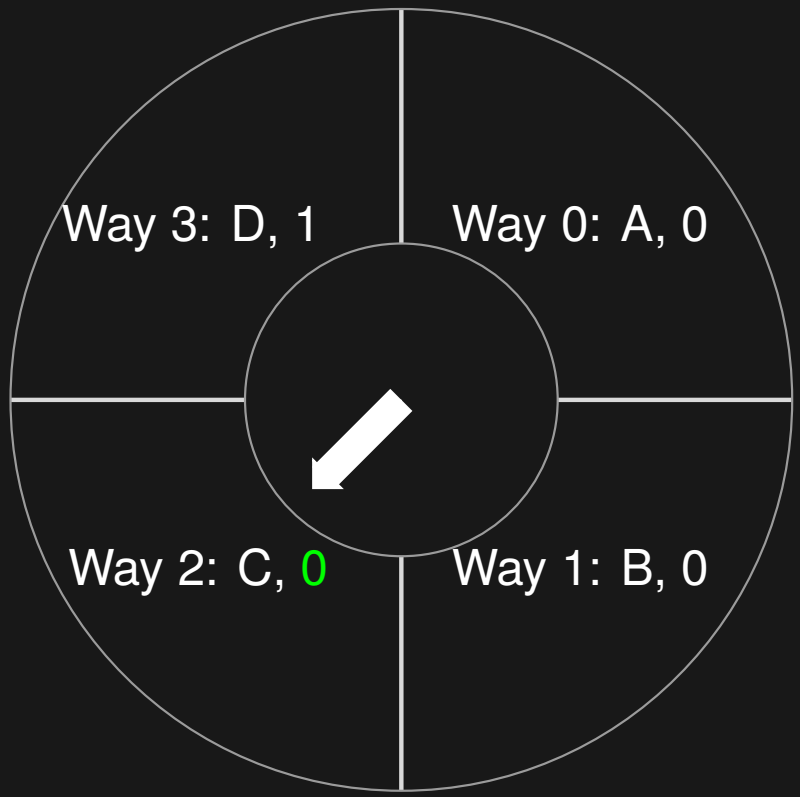
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 1. Used-bit is 1, reset it to 0



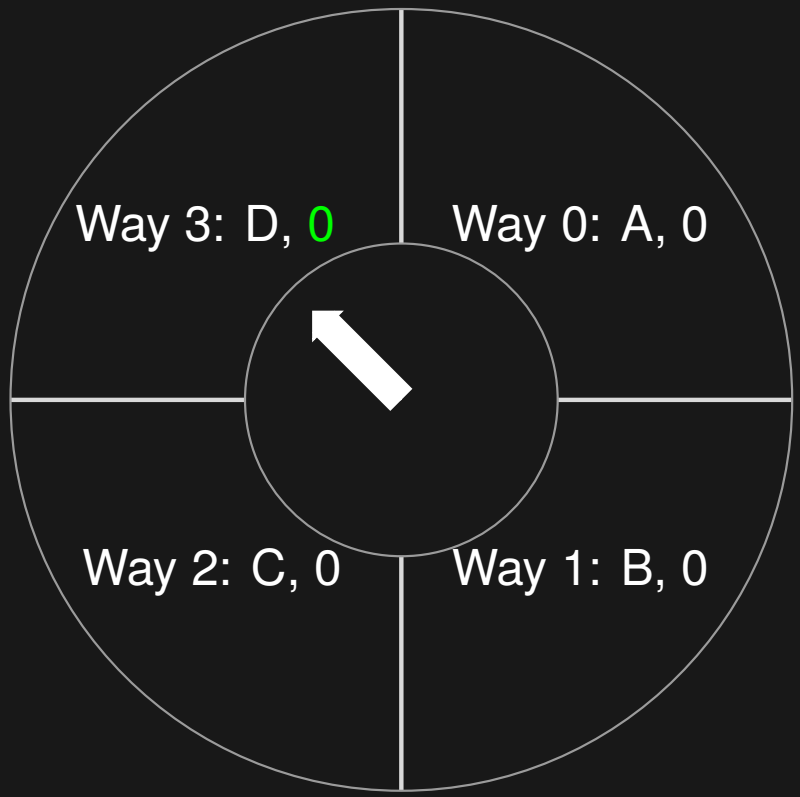
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 2. Used-bit is 1, reset it to 0



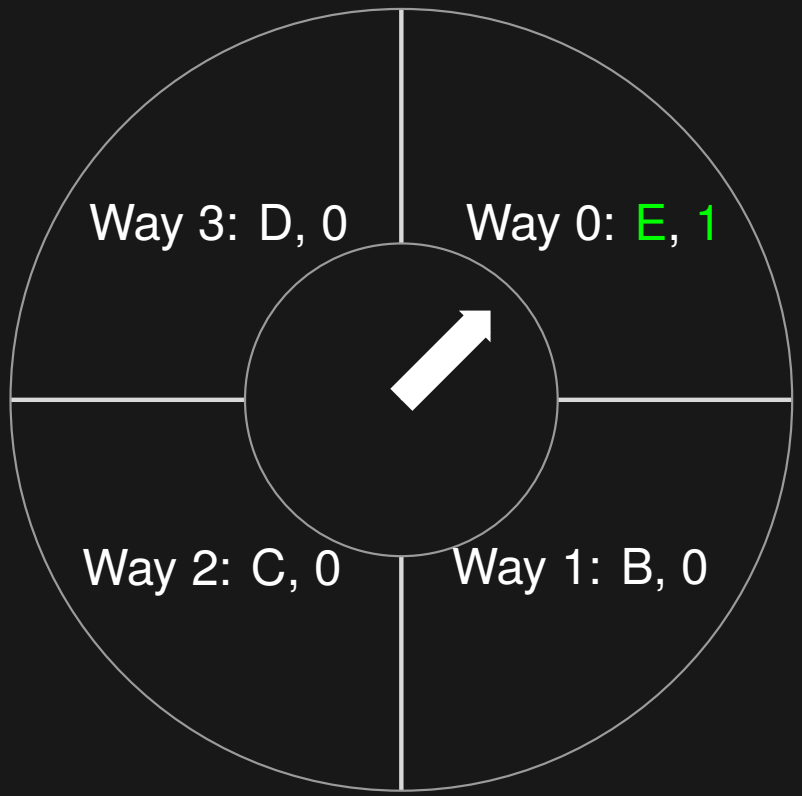
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 3. Used-bit is 1, reset it to 0



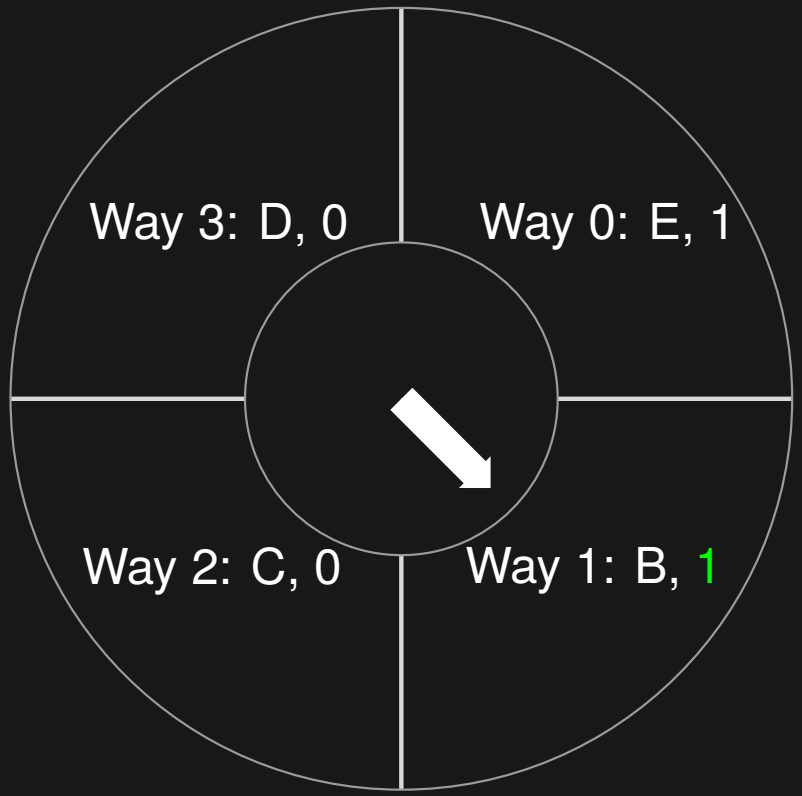
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access E, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 0. Used-bit is 0, find the victim. Assign E to way 0, and set the used-bit to 1



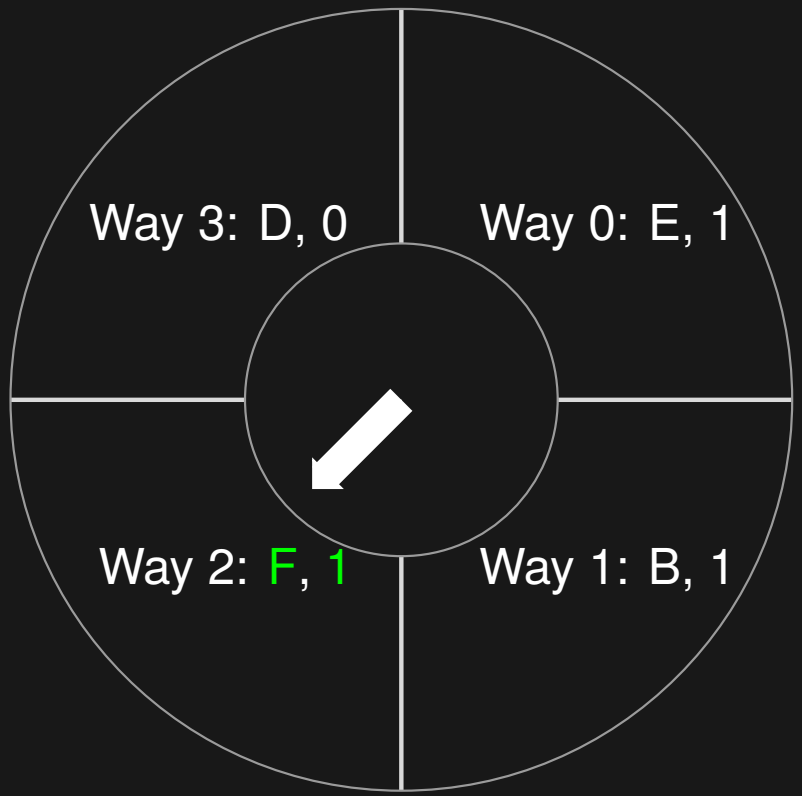
An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access B, a cache hit to way 1. Update the way 1's used-bit to 1.



An Example of CAR Algorithm

- Again, consider the following accesses to the same set in a 4-way cache: A, B, C, D, E, B, F.
  - Access F, a cache miss, need to find replacement by visiting ways clock wise to find a way with a used-bit of 0.
    - Visit way 2 first. Used-bit is 0, find the victim. Assign F to way 0, and set the used-bit to 1.



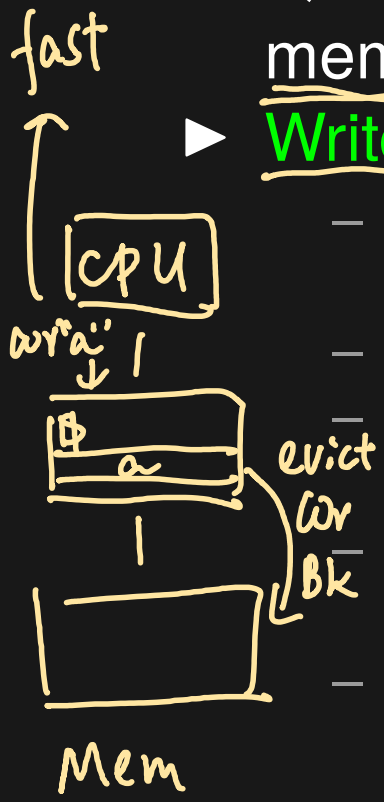
Write Strategies

Write Policies

- Writes are only about 21% of data cache traffic
- Optimize cache for reads, do writes "on the side"
  - Reads can do tag check/data read in parallel
  - Writes must be sure we are updating the correct data and the correct amount of data (1-8 byte writes)
  - Writes have to be serial to ensure correctness. However, serial process is slow.
- Two cases to consider,
  - What to do on a write hit?
  - What to do on a write miss?

Write Hit Policies

- Write hits: write requests hit data in the cache. Writes can be directed to cache instead of memory.
- Question: When to propagate new values to memory?
- Write back – Information is only written to the cache:
  - Higher level caches and memory are only updated when it is evicted (dirty bits say when data has been modified)
  - Can write at speed of cache
  - Caches become temporarily inconsistent with higher levels of cache and memory.
  - Uses less memory bandwidth/power (multiple consecutive writes may require only 1 final write to memory at eviction)
  - Multiple writes within a cache line can be merged into one write
  - Evictions are longer latency now (requires a memory write)

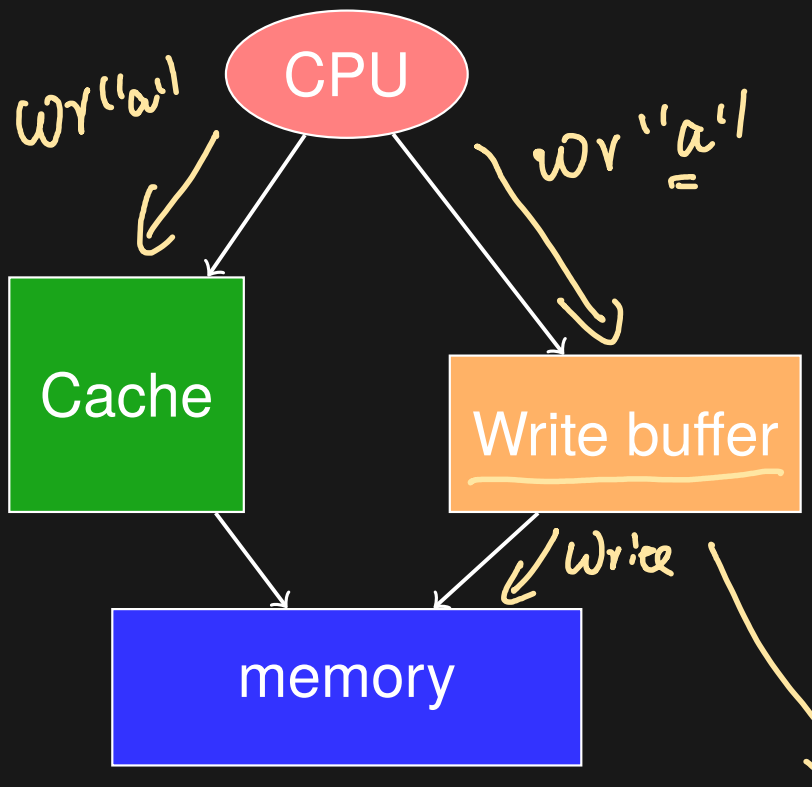


write "a" x 3 to cache ⇒ write "a" x 1 to main mem

Write Hit Policies cont'd

- Write hits: write requests hit data in the cache. Writes can be directed to cache instead of memory.
- Question: When to propagate new values to memory?
- Write Through – Information is written to cache and to the lower-level memory:
  - Main memory is always "consistent/coherent"
  - Easier to implement – no dirty bits
  - Reads never result in writes to lower levels (cheaper)
  - Higher bandwidth needed as every write needs to be sent to memory
  - Processors may need to stalls to wait for the finish of slow memory writes.
    - Write buffers used to avoid write stalls
  - Required for memory with I/O data (e.g., video meory) to ensure consistent between CPU and I/O devices.

Write Buffers ⇒ Wr Through

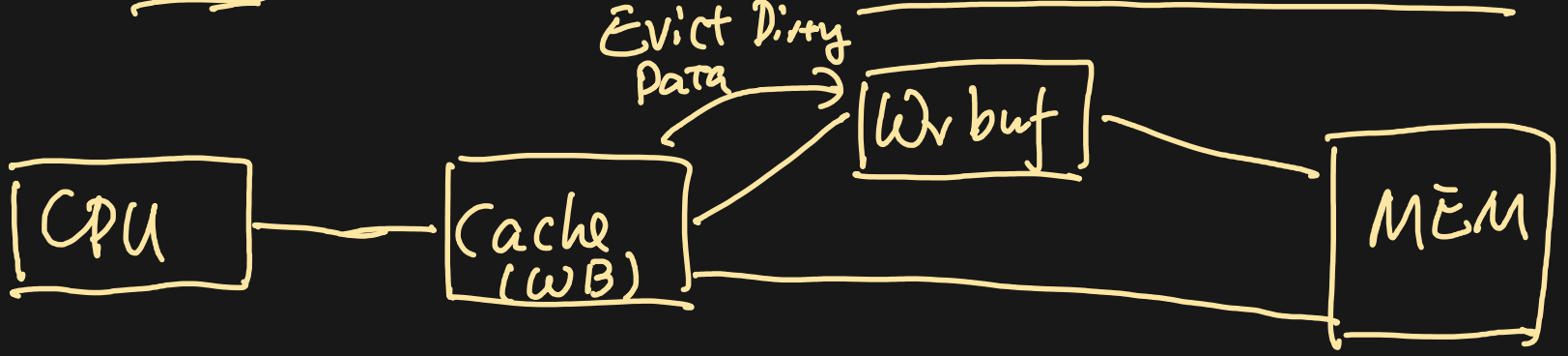


- Small chunks of memory to buffer outgoing writes.
- Instead of memory, processor writes to write buffer and return.
- The memory controller sends data in write buffers to memory without processor management.

like write cache

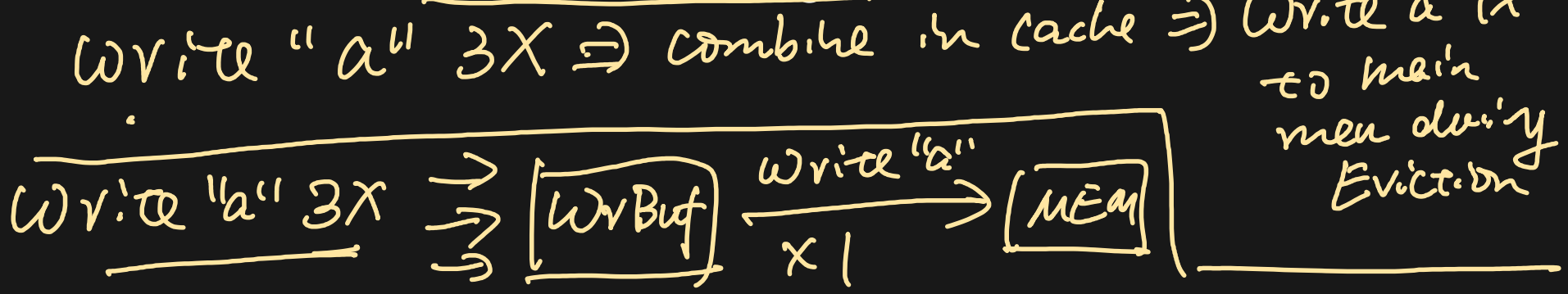
Write Buffers cont'd

- Processor can continue when data written to buffer, allowing overlap of processor execution with memory update
- Full write buffers will still cause processors to stall.
- Write buffers can also serve read requests: if a cache miss requests some data happens to be in the write buffer, the data can be served from write buffer.



Write Combining at Write Buffers

- If there are multiple writes to different parts of the same cache line, they can be combined in the write buffers to save memory bandwidth.
- After all writes to the cache line is done, one memory write request is issued to update the whole cache line in the memory.
  - Th memory read/write transaction size is typically the cache lines size.
- Modern CPU usually implements write combining instead of pure write-through.



Write Buffer Flush Policy

- When to flush?
  - Aggressive flushing => Reduce chance of stall cycles due to full write buffer
  - Conservative flushing => Write combining more likely (entries stay around longer) => reduces memory traffic
- What to flush?
  - Selective flushing of particular entries?
  - Flush everything below a particular entry
  - Flush everything
- Again, benchmarks can help us pick a good policy.

Actually CPU Implementation

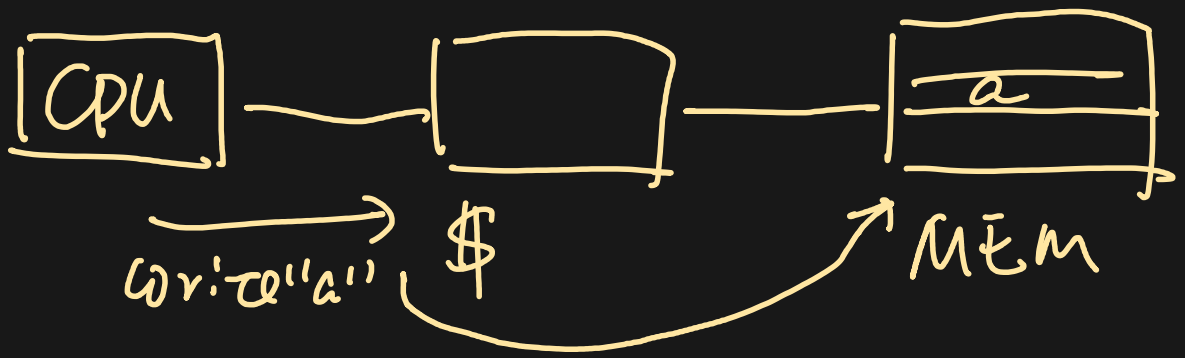
- Users can specify a part of memory to be write-back, write-combined or un-cached.
- Un-cached memory behaves like pure write-through for memory writes.
- Write-combined memory has weak coherence, which may affect program correctness if used for normal data accesses (i.e., non-I/O data).

W.B. Wr Buf

WC buffer (write cache)

Write misses?

- Write Allocate
  - A cache block/slot is allocated on a write miss
  - Standard write hit actions follow the cache block/slot allocation
  - Write misses = Read Misses
  - Goes well with write-back → work well if the written is accessed/read soon
- No-write Allocate
  - Write misses do not allocate a block
  - Only update lower-level memory
  - Cache blocks/slots only allocated on read misses!
  - Goes well with write-through



Case Study: Intel Kabylake

- L1I Cache 32 KiB/core, 8-way set associative
- L1D Cache 32 KiB/core, 8-way set associative
- L2 Cache 256 KiB/core, 4-way set associative
- L3 Cache 2 MiB/core, 16-way set associative

private private shared

4 core ⇒ 8 MB L3 cache

accessed by all cores ~~processors~~

Case Study: AMD Zen (Ryzen)

- L1I Cache 64 KiB/core, 4-way set associative
- L1D Cache 32 KiB/core, 8-way set associative
- L2 Cache 512 KiB/core, 8-way set associative
- L3 Cache 2 MiB/core, 16-way set associative

Case Study: ARM Coretex A72

- L1I Cache 48 KiB/core, 3-way set associative
- L1D Cache 32 KiB/core, 2-way set associative
- L2 Cache 512 KiB to 4 MiB, 16-way set associative

Shared

Acknowledgment

This lecture is partially based on the slides from Dr. David Brooks.