CS61C

Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

UC Berkeley
Lecturer
Justin Yokota

Caches III: Direct Mapped

- EECS/CS **Undergraduate Town Hall**, Thursday April 11 4-5pm, Wozniak Lounge (tomorrow!)
- Data4All Kickoff event! **RSVP by today**. EdStem post #824 [link]

Warming up the Fully Associative Cache [review]

- Suppose the cache starts **cold**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0

$\underbrace{0101\ 1110\ 00}_{0x178}\underbrace{00}_{0x0}$

   a. **Cache hit**! Tag 0x178 is valid!
   b. Read byte at 0x0 **offset** and return to processor.

Val id	Tag	Data 11	10	01	00
1	0x10F	…	…	…	…
1	0x178	…	…	…	…
1	0x209	…	…	…	…
0	…	…	…	…	…

Memory address:

11 2	1 0
(10b)	(2b)
tag	offset

No access to main memory occurs on a **cache hit**.

Yan, Yokota

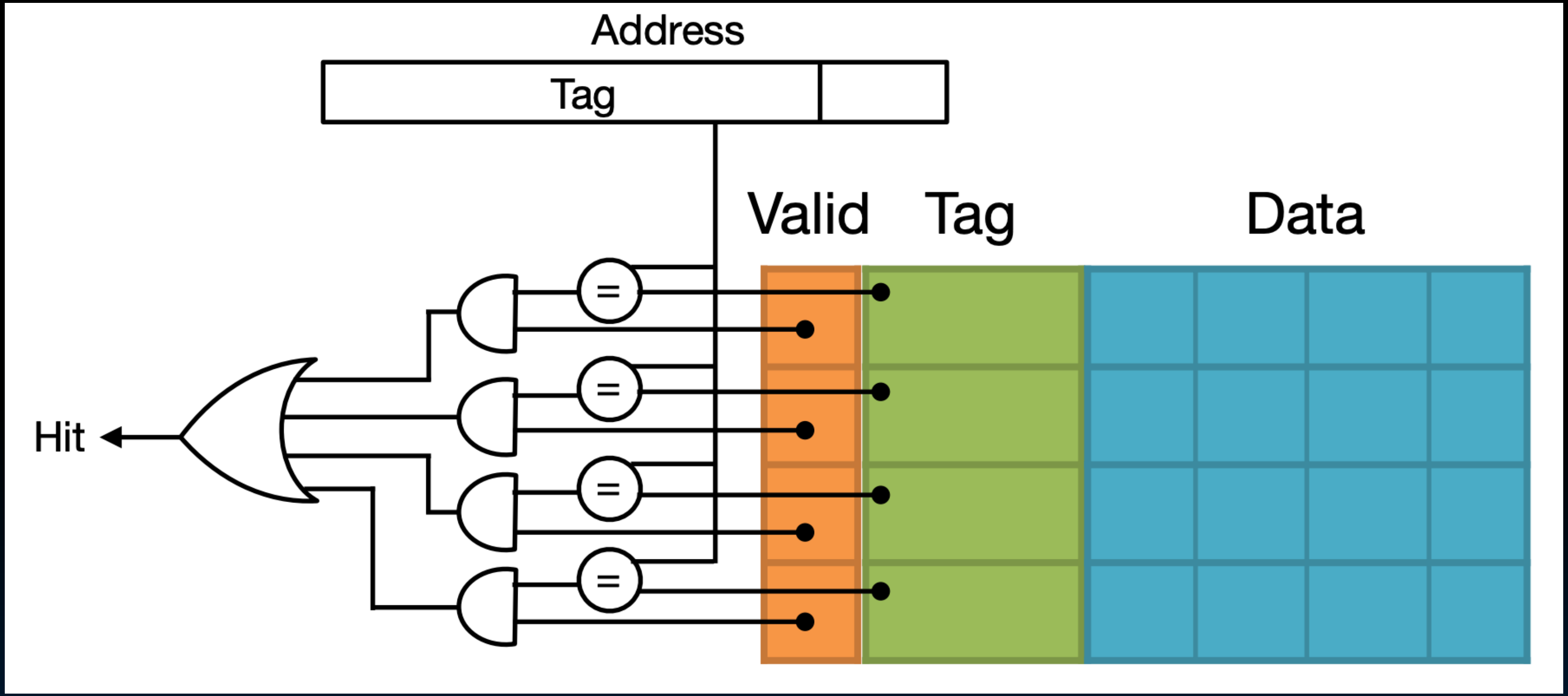
Cache Terminology

[review]

- Cache line (i.e., cache) block
  - The smallest unit of memory that can be transferred between the main memory and the cache.
  - Each line has its own entry in the cache.
  - Line size/block size: The number of bytes in each cache line.
- When we bring data from the main memory into the cache, it is done in the granularity of a cache line (or cache block).
  - Typically **cache lines are 64 bytes**.
  - Cache blocks (cache lines) helps us take advantage of **spatial locality**.
- **Capacity**:
  - The total number of data bytes that can be stored in a cache.
  - For fully associative cache, **capacity = # lines * line size**.

Hardware: Fully Associative Cache

- A Fully Associative Cache must compare tags in parallel:



Need hardware comparator for every single entry!

Sometimes infeasible; e.g., 1MB cache w/ 16B entries → $2^{20}/2^{4} = 2^{16}$ = 16K comparators.

Agenda

Block Replacement Policies

- Block Replacement Policies
- Write Policies
- Direct Mapped Cache
- Types of Misses

A Warmed up Cache Still Can Miss

- Suppose the cache starts **cold**.

1. Load byte 0x43F   0x10F, 0x3
2. Load byte 0x5E2   0x178, 0x2
3. Load word 0x824   0x209, 0x0
4. Load word 0x5E0   0x178, 0x0
5. Load word 0x524   0x149, 0x0

Fills up the cache

Val id	Tag	Data			
		11	10	01	00
1	0x10F	…	…	…	…
1	0x178	…	…	…	…
1	0x209	…	…	…	…
1	0x149	…	…	…	…

A Warmed up Cache Still Can Miss

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0
5. Load word 0x524 0x149, 0x0
6. Load byte 0x972 $\underbrace{1001\ 0111}_{\text{tag } 0x25C}\ \underbrace{0010}_{\text{offset } 0x2}$

Cache miss!

Valid	Tag	Data 11	10	01	00
1	0x10F	…	…	…	…
1	0x178	…	…	…	…
1	0x209	…	…	…	…
1	0x149	…	…	…	…

When the FA cache **reaches capacity**, we can still miss.

Therefore we also need a policy for **block replacement**!

Yan, Yokota

Block Replacement Policies (i.e., Eviction)

- Least Recently Used (LRU)
  - Replace the entry that has not been used for the longest time, i.e., has the oldest previous access.
    - Pro: **temporal locality**!
      - recent past use implies likely future use
      - This is a very effective policy
    - Con: Complicated hardware to keep track of access history → performance hit

Fully Associative Cache with LRU policy

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F   0x10F, 0x3
2. Load byte 0x5E2   0x178, 0x2
3. Load word 0x824   0x209, 0x0
4. Load word 0x5E0   0x178, 0x0
5. Load word 0x524   0x149, 0x0
6. Load byte 0x972 $\underbrace{1001\ 0111\ 00}_{\text{tag}}\underbrace{10}_{\text{offset}}$

tag 0x25C   offset 0x2

Val id	LRU	Tag	Data 11	10	01	00
1		0x10F	…	…	…	…
1		0x178	…	…	…	…
1		0x209	…	…	…	…
1		0x149	…	…	…	…

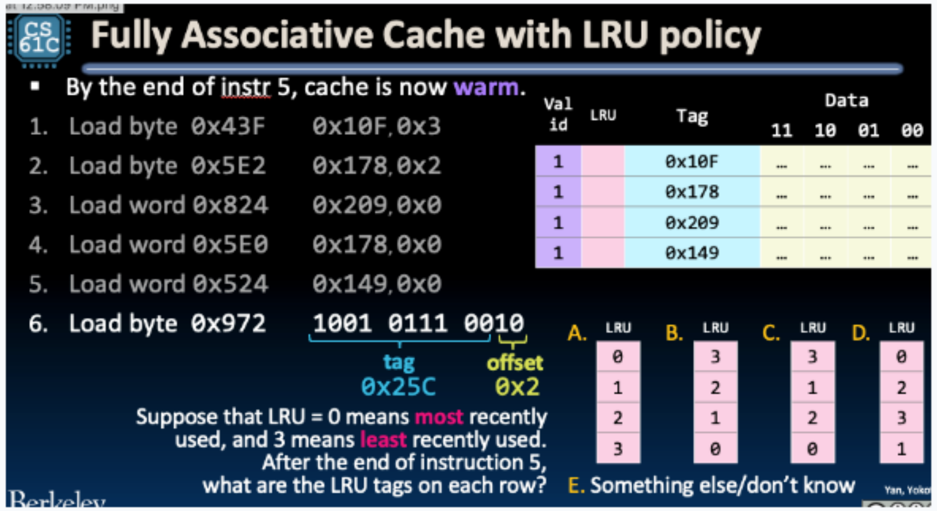
Suppose that LRU = 0 means **most** recently used, and 3 means **least** recently used. After the end of instruction 5, what are the LRU tags on each row?

A. LRU	B. LRU	C. LRU	D. LRU
0	3	3	0
1	2	1	2
2	1	2	3
3	0	0	1

E. Something else/don't know

Yan, Yokota

At the end of instruction 5, what are the LRU tags on each row?



Fully Associative Cache with LRU policy

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0
5. Load word 0x524 0x149, 0x0
6. Load byte 0x972 $\underbrace{1001\ 0111\ 00}_{\text{tag } 0x25C}\underbrace{10}_{\text{offset } 0x2}$

Valid	LRU	Tag	Data 11	Data 10	Data 01	Data 00
1		0x10F	...	...	...	...
1		0x178	...	...	...	...
1		0x209	...	...	...	...
1		0x149	...	...	...	...

Suppose that LRU = 0 means **most** recently used, and 3 means **least** recently used. After the end of instruction 5, what are the LRU tags on each row?

	A. LRU	B. LRU	C. LRU	D. LRU
	0	3	3	0
	1	2	1	2
	2	1	2	3
	3	0	0	1

E. Something else/don't know

Berkeley

Yan, Yoko

A

0%

B

0%

C

0%

D

0%

Something else/don't know

0%

Fully Associative Cache with LRU policy

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0
5. Load word 0x524 0x149, 0x0
6. Load byte 0x972

$$\underbrace{1001\ 0111\ 00}_{\substack{\text{tag} \\ \texttt{0x25C}}}\underbrace{10}_{\substack{\text{offset} \\ \texttt{0x2}}}$$

Val id	LRU	Tag	Data			
			11	10	01	00
1	3	0x10F	…	…	…	…
1	1	0x178	…	…	…	…
1	2	0x209	…	…	…	…
1	0	0x149	…	…	…	…

Suppose that LRU = 0 means **most** recently used, and 3 means **least** recently used. After the end of instruction 5, what are the LRU tags on each row?

Yan, Yokota

Fully Associative Cache with LRU policy

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0
5. Load word 0x524 0x149, 0x0
6. Load byte 0x972 0x25C, 0x2

Valid	LRU	Tag	Data 11	Data 10	Data 01	Data 00
1	0	0x25C	…	…	…	…
1	2	0x178	…	…	…	…
1	3	0x209	…	…	…	…
1	1	0x149	…	…	…	…

a. **Cache miss**! Tag 0x25C is not valid.

b. **Replace according to block replacement policy.**
Load into cache the **4-byte block** from 0x970 to 0x973. Mark **valid bit**. Update block replacement policy fields.

Suppose our cache uses LRU (Least Recently Used).

c. Read byte at 0x2 **offset** and return to processor.

Block Replacement Policies (i.e., Eviction)

- Least Recently Used (LRU)
  - Replace the entry that has not been used for the longest time, i.e., has the oldest previous access.
    - Pro: **temporal locality**!
      - recent past use implies likely future use
      - This is a very effective policy
    - Con: Complicated hardware to keep track of access history → performance hit

LRU is ideal for **temporal locality**, but the last three are used more commonly in practice.

- Most Recently Used (MRU)
  - Replace the entry that has the newest previous access.
- First In, First Out (FIFO)
  - Replace the oldest block in the set (queue).
- Last In, First Out (LIFO)
  - Replace the newest block in the set (stack).

Reasonable approximations to LRU and MRU without adding too much excess hardware.

Note both ignore order of accesses after (before) the first (last) access.

- Random

Works surprisingly okay (when given a low temporal locality workload)

Agenda

Write Policies

- Block Replacement Policies
- Write Policies
- Direct Mapped Cache
- Types of Misses

What about Stores?

- By the end of instr 5, cache is now **warm**.

1. Load byte 0x43F 0x10F, 0x3
2. Load byte 0x5E2 0x178, 0x2
3. Load word 0x824 0x209, 0x0
4. Load word 0x5E0 0x178, 0x0
5. Load word 0x524 0x149, 0x0
6. Load byte 0x972 0x25C, 0x2
7. Store byte 0x524 0x149, 0x0

Valid	LRU	Tag	Data 11	10	01	00
1	0	0x25C	…	…	…	…
1	2	0x178	…	…	…	…
1	3	0x209	…	…	…	…
1	1	0x149	…	…	…	…

Cache hit!...and then?

How do we handle stores?

Write-through vs. Write-back Policies

- **Store** instructions write to memory, which **changes values**.
- Hardware needs to ensure that cache and memory have consistent information.
- **Write-through**:
  - Write to the cache and memory at the same time.
  - (writes to memory take longer)

Write-through vs. Write-back Policies

[LRU column fixed post-class]

- **Store** instructions write to memory, which **changes values**.
- Hardware needs to ensure that cache and memory have consistent information.
- **Write-through**:
  - Write to the cache and memory at the same time.
  - (writes to memory take longer)
- **Write-back**:
  - Write data in cache and set a **dirty bit** to 1.
  - When this block gets evicted from the cache (and "back" to memory), write to memory.

Valid	Dirty	LRU	Tag	Data			
				11	10	01	00
1	0	1	0x25C	…	…	…	…
1	0	2	0x178	…	…	…	…
1	0	3	0x209	…	…	…	…
1	1	0	0x149	…	…	…	…

Store byte 0x524 0x149, 0x0

Cache hit w/**write-back**:
update data **within cache block**, and only write back to memory when this block is evicted.

Write-back policies require an additional **dirty bit** field.

Fully Associative Cache w/ LRU, Write-Back

Note: LRU also updates for all blocks on writes as well!! Because you are "using" blocks.
[LRU column fixed post-class]

Valid	Dirty	LRU	Tag	Data 11	Data 10	Data 01	Data 00
1	0	1	0x25C	…	…	…	…
1	0	2	0x178	…	…	…	…
1	0	3	0x209	…	…	…	…
1	1	0	0x149	…	…	…	…

Store byte 0x524 0x149, 0x0

Cache hit w/**write-back**:
update data **within cache block**, and only write back to memory when this block is evicted.

- **Write-back**:
  - Write data in cache and set a **dirty bit** to 1.
  - When this block gets evicted from the cache (and "back" to memory), write to memory.

Write-through vs. Write-back Policies

- **Store** instructions write to memory, which **changes values**.
- Hardware needs to ensure that cache and memory have consistent information.

- **Write-through**:
  - Write to the cache and memory at the same time.
  - (writes to memory take longer)

Very simple to implement

- **Write-back**:
  - Write data in cache and set a **dirty bit** to 1.
  - When this block gets evicted from the cache (and "back" to memory), write to memory.

(typically) lower traffic to memory, because you likely write multiple times before evicting from cache

Agenda

Direct Mapped Cache

- Block Replacement Policies
- Write Policies
- Direct Mapped Cache
- Types of Misses

Cache Design: Placement Policies

Fully Associative Cache

Put a new block anywhere

$\longleftrightarrow$

Set-Associative Cache

(out of scope)

Direct Mapped Cache

Put a new block in one specific place

(up next)

Fully associative caches need expensive hardware.

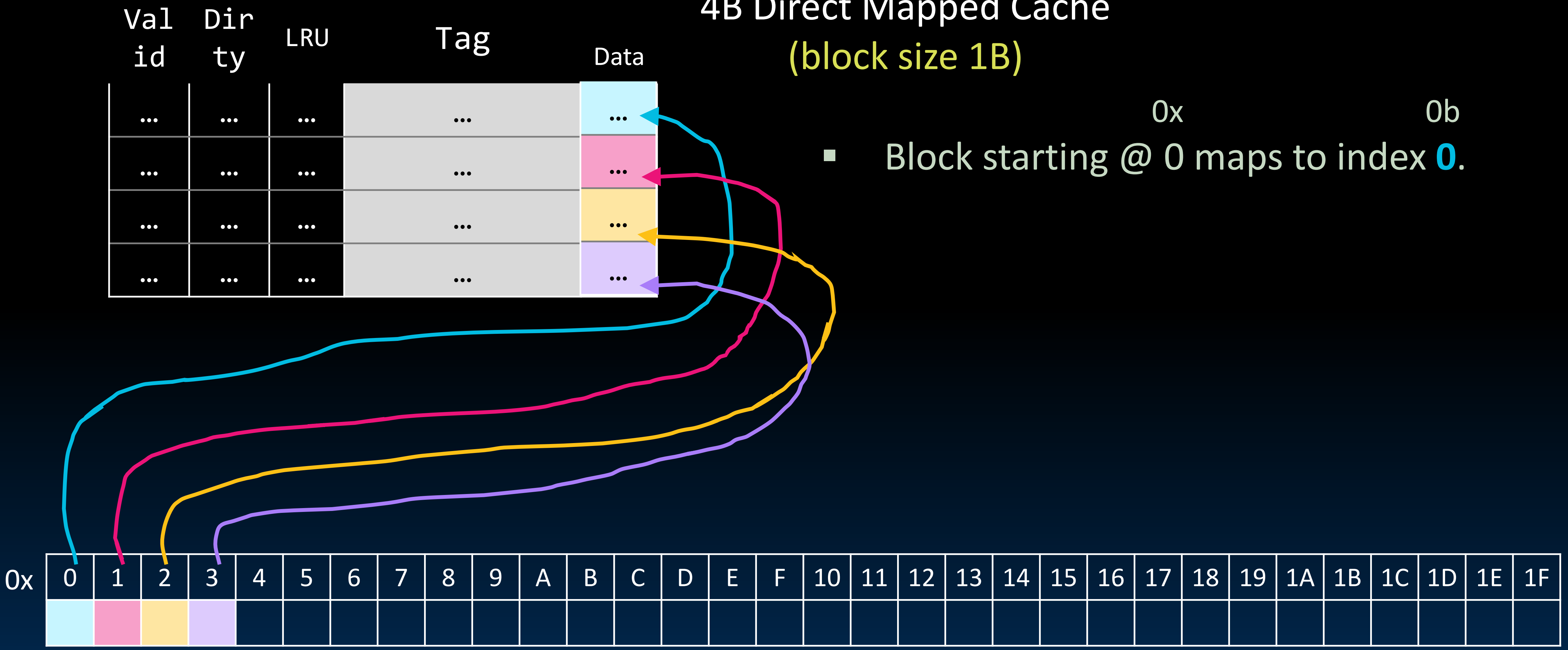
Direct Mapped Cache

- **Placement policy**: Each memory address is associated with exactly one possible block in the cache.
  - To check for existence in the cache, we only need to look in a single location in the cache.

Valid	Dirty	Tag	Data			
			11	10	01	00
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…

How do we ensure this?

(1/2) The Idea behind Direct Mapped Caches



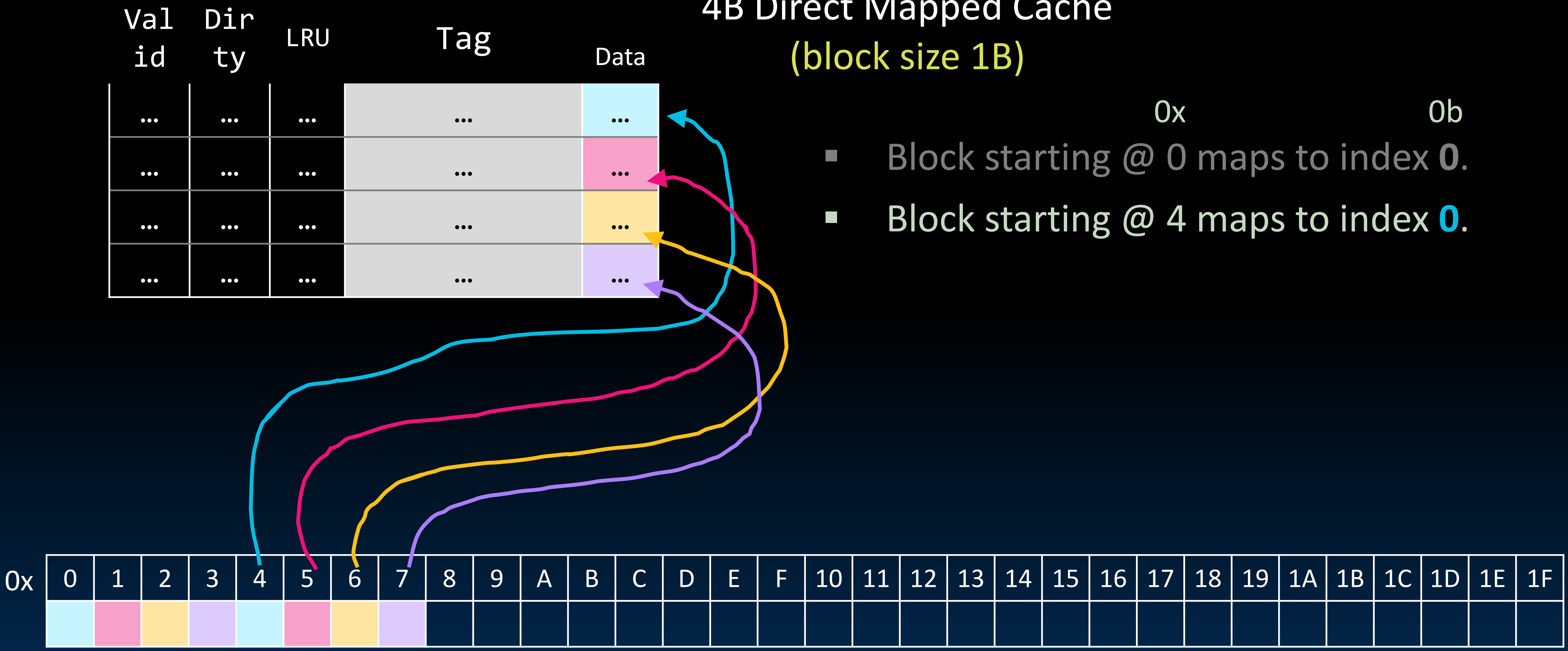
4B Direct Mapped Cache
(block size 1B)

0x 0b

- Block starting @ 0 maps to index **0**.

Memory

(1/2) The Idea behind Direct Mapped Caches



Valid
Dirty
LRU
Tag
Data
0x
0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
Memory

Valid	Dirty	LRU	Tag	Data
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

4B Direct Mapped Cache
(block size 1B)

0x 0b

- Block starting @ 0 maps to index **0**.
- Block starting @ 4 maps to index **0**.

Yan, Yokota

(1/2) The Idea behind Direct Mapped Caches

Valid	Dirty	LRU	Tag	Data
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

4B Direct Mapped Cache
(block size 1B)

0x 0b

- Block starting @ 0 maps to index **0**.
- Block starting @ 4 maps to index **0**.
- Block starting @ 8 maps to index **0**.
- Block starting @ C maps to index **0**.
- etc.

Different addresses, get **same block index** but different **tags**.

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪	🟦	🟥	🟨	🟪

Memory

(2/2) The Idea behind Direct Mapped Caches

Valid	Dirty	LRU	Tag	Data 01	Data 00
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…

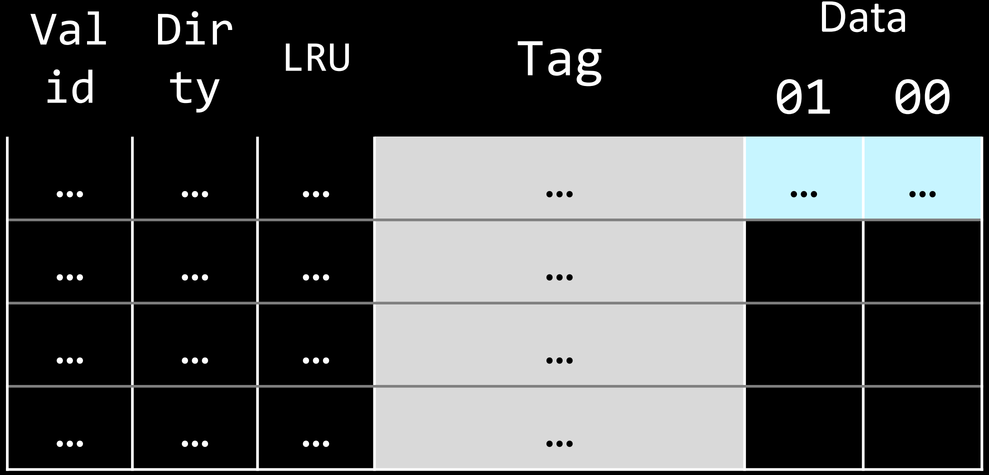
8B Direct Mapped Cache
(block size **2B**)

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F

Memory

Yan, Yokota

(2/2) The Idea behind Direct Mapped Caches



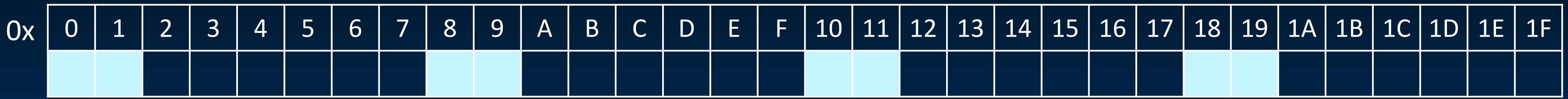
Valid	Dirty	LRU	Tag	Data 01	Data 00
…	…	…	…	…	…
…	…	…	…		
…	…	…	…		
…	…	…	…		

8B Direct Mapped Cache
(block size **2B**)

0x 0b

- Block starting @ 0 maps to index **0**.
- Block starting @ 8 maps to index **0**.
- Block starting @ 10 maps to index **0**.
- Block starting @ 18 maps to index **0**.
- etc.

Different addresses, get **same block index** but different **tags**.



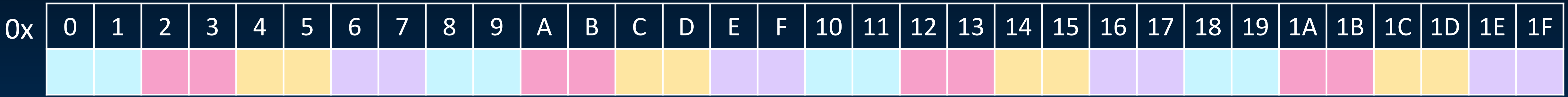
0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	■	■							■	■							■	■							■	■						

Memory

(2/2) The Idea behind Direct Mapped Caches

8B Direct Mapped Cache
(block size **2B**)

Valid	Dirty	LRU	Tag	Data 01	Data 00
…	…	…	…	… 🟦	… 🟦
…	…	…	…	… 🟥	… 🟥
…	…	…	…	… 🟨	… 🟨
…	…	…	…	… 🟪	… 🟪



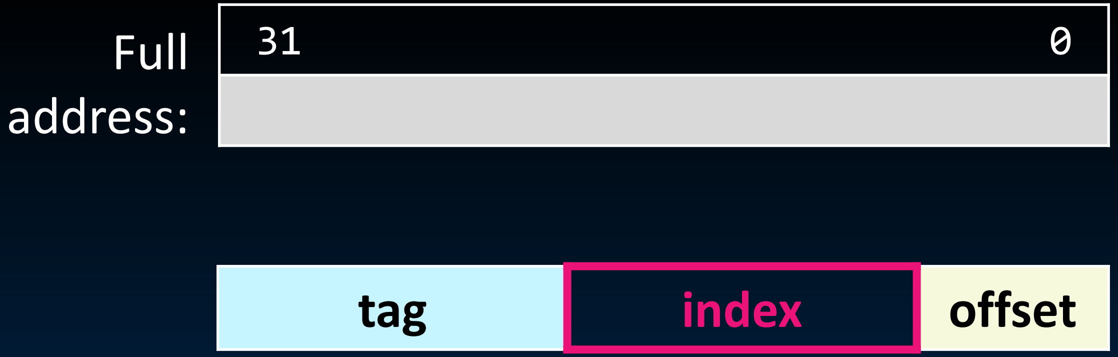
Memory

Yan, Yokota

Direct Mapped Cache

- **Placement policy**: Each memory address is associated with exactly one possible block in the cache.
  - To check for existence in the cache, we only need to look in a single location in the cache.

Val id	Dir ty	Tag	Data 11	10	01	00
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…



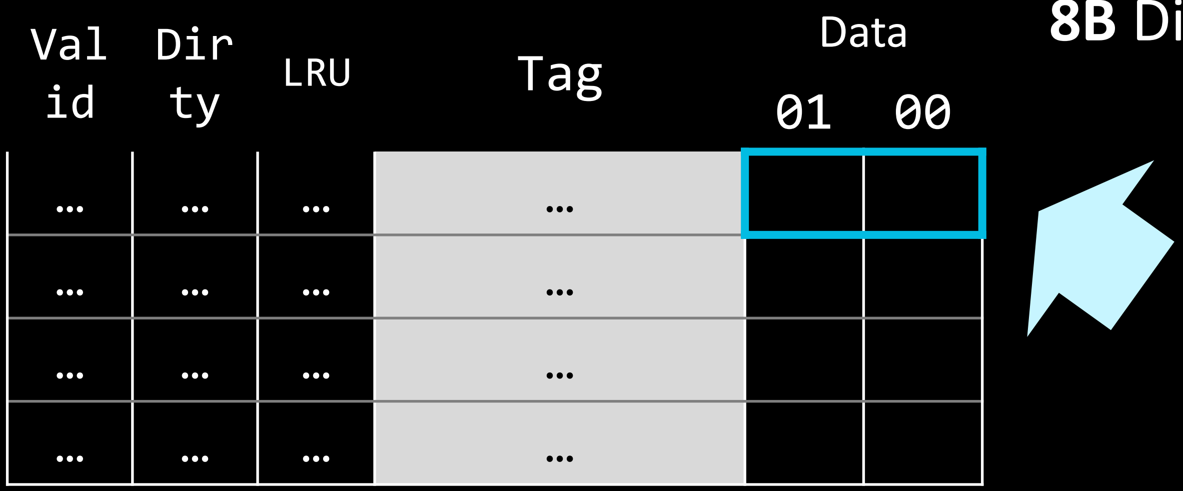
tag to check if have correct block

index to select block

byte offset within block

In direct mapped caches, split address to also get **index** of cache block to directly map to.

(2/2) The Idea behind Direct Mapped Caches

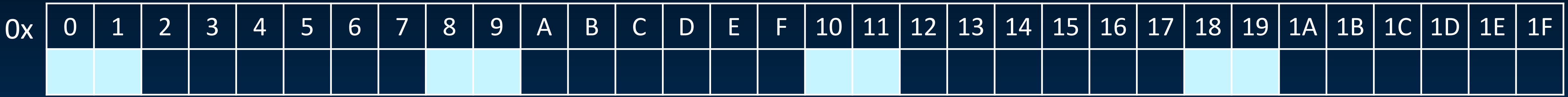


8B Direct Mapped Cache
(block size **2B**)

	0x		0b	0b
▪	Block starting @ 0 maps to index **0**.			Tag 00
▪	Block starting @ 8 maps to index **0**.			Tag 01
▪	Block starting @ 10 maps to index **0**.			Tag 10
▪	Block starting @ 18 maps to index **0**.			Tag **11**

$$\underbrace{\texttt{1 100x}}$$

Different addresses, get **same block index** but different **tags**.



0x
0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F

Memory

Fill in the blank

Valid	Dirty	Tag	Data			
			11	10	01	00
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…
…	…	…	…	…	…	…

[activity slide]

Suppose we have the following **direct mapped** cache, for **12b** addresses.

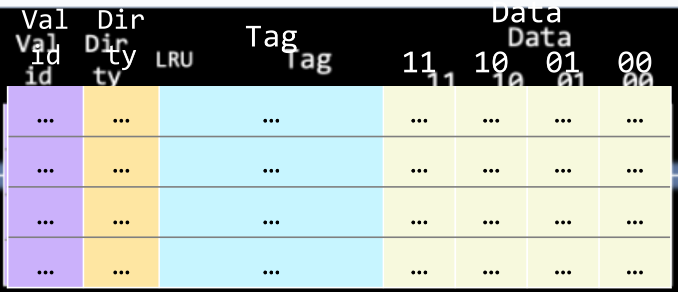
	A.	B.	C.	D.	E.	F.
1. What is the block size / line size, in bytes?	2	4	8	10	16	Other
2. What is the capacity, in bytes?	2	4	8	10	16	Other
3. For a 12b address, how many bits is the byte offset?	2	4	8	10	16	Other
4. For a 12b address, how many bits is the index ?	2	4	8	10	16	Other
5. For a 12b address, how many bits is the tag?	2	4	8	10	16	Other

Berkeley UNIVERSITY OF CALIFORNIA

Yan, Yokota

Put a pin on the answers for **direct mapped caches.** 0

Fill in the blank



[activity slide]

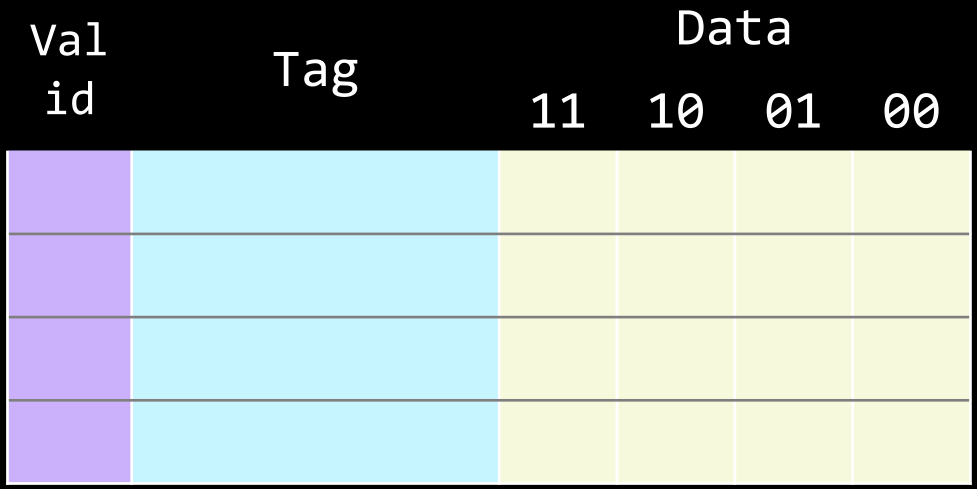
Suppose we have the following **direct mapped** cache, for 12b addresses.

	A.	B.	C.	D.	E.	F.
1. What is the block size / line size, in bytes?	2	4	8	10	16	Other
2. What is the capacity, in bytes?	2	4	8	10	16	Other
3. For a 12b address, how many bits is the byte offset?	2	4	8	10	16	Other
4. For a 12b address, how many bits is the index ?	2	4	8	10	16	Other
5. For a 12b address, how many bits is the tag?	2	4	8	10	16	Other

Terminology, Part 1 (for 12b addresses!)

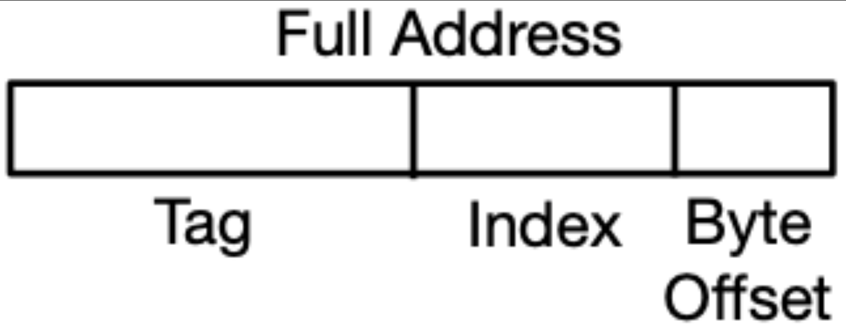
- Cache block/line: A single entry in the cache

1. Block size / line size: # bytes per cache block
   4 bytes
2. Capacity 4 x 4 bytes = **16 bytes**
   - Total # data bytes that can be stored in a cache
3. Offset $\log_2$(block size) = **2 bits**
   - Identifies byte offset of data stored within a given cache block
4. Index $\log_2$(# lines) = **2 bits**
   - Selects block
5. Tag # address bits - # offset bits - # index bits = **8 bits**
   - Identifies (checks) if given cache block is correct blocks



Yan, Yokota

Direct Mapped Example



Read byte 0xFE2

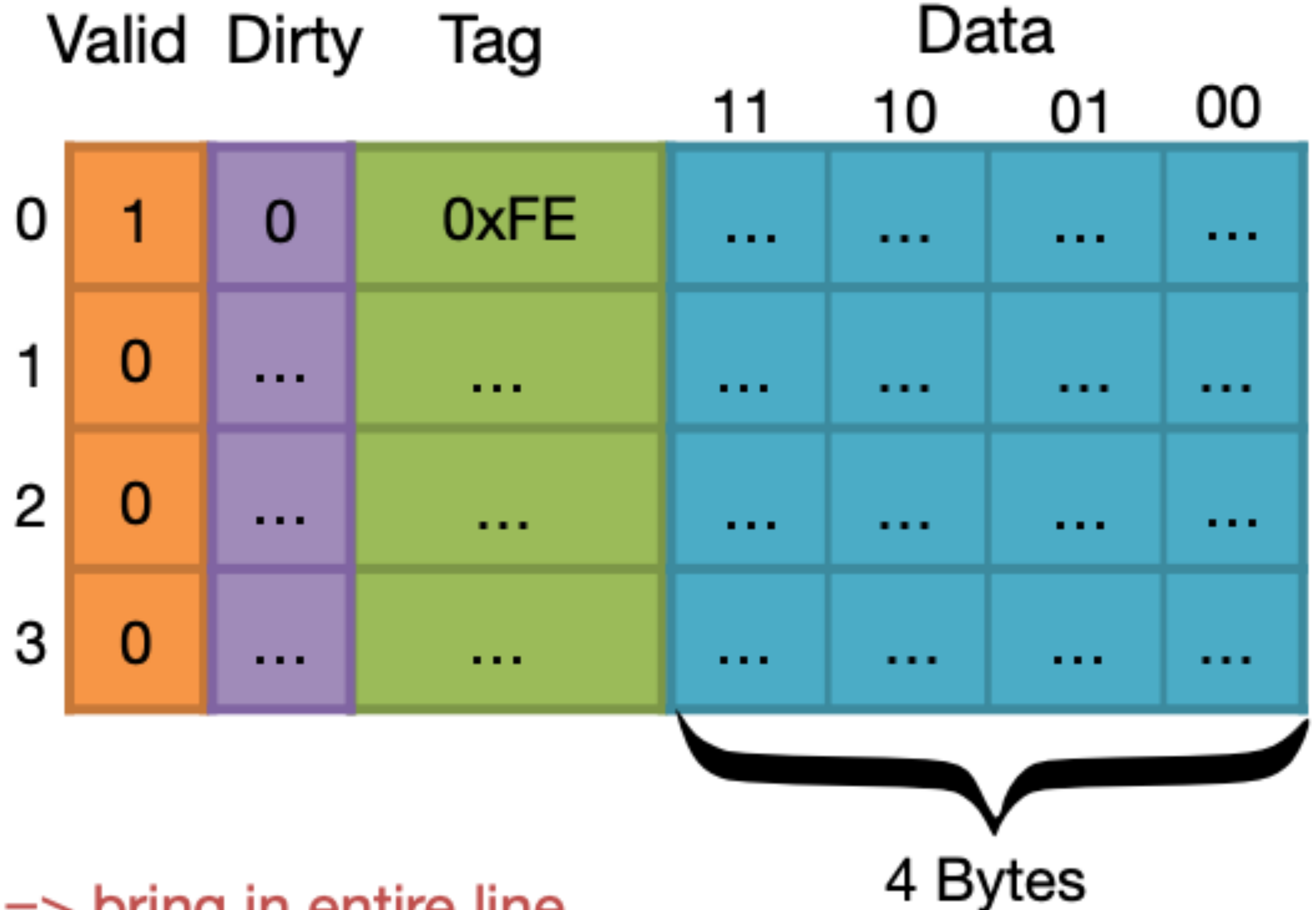
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 0010

T = 0xFE

I = 0x0

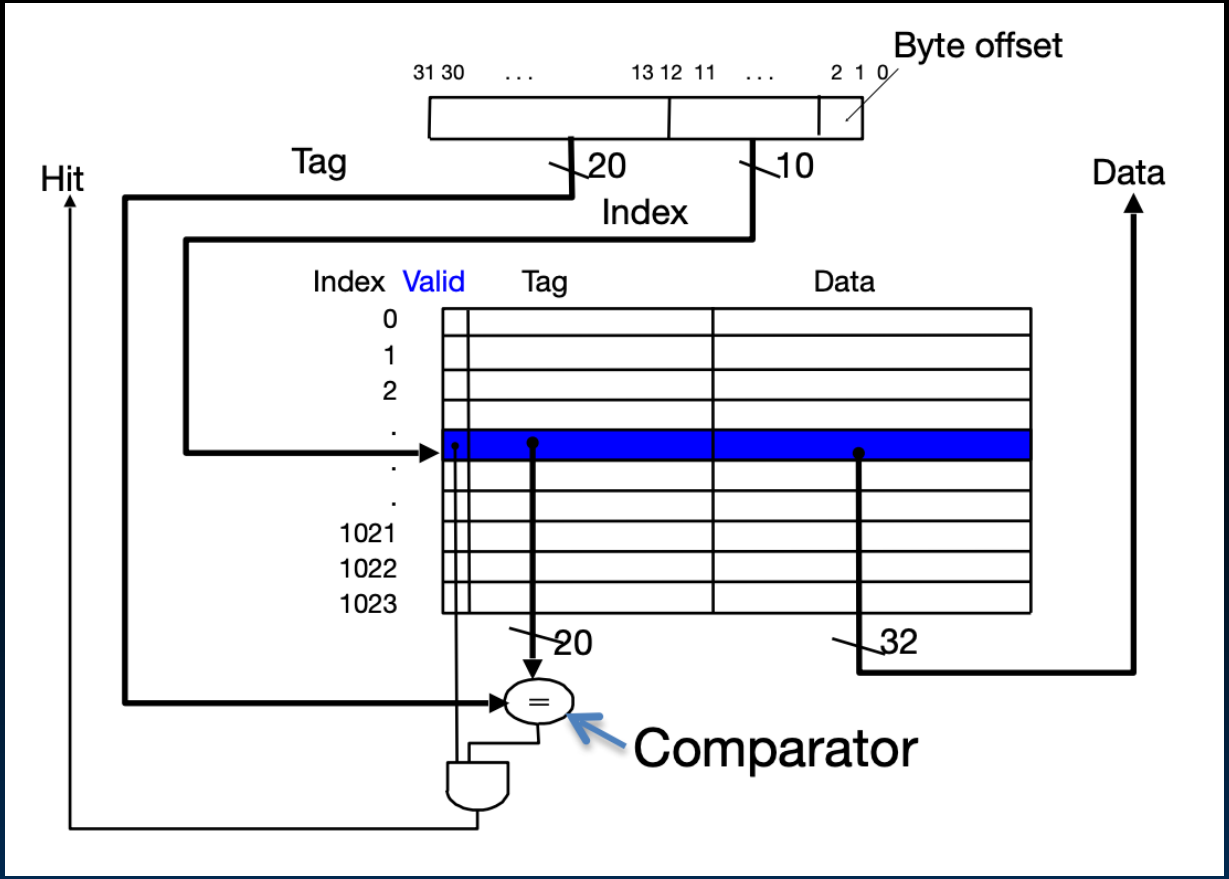
O = 0x2



	Valid	Dirty	Tag	Data 11	Data 10	Data 01	Data 00
0	1	0	0xFE	...	...	...	...
1	0	...	...	...	...	...	...
2	0	...	...	...	...	...	...
3	0	...	...	...	...	...	...

Cache Miss => bring in entire line

Hardware: Direct Mapped Cache



4B blocks, 4KB data

Much simpler than Fully Associative! Just check one tag.

Yan, Yokota

Agenda

Types of Misses

- Block Replacement Policies
- Write Policies
- Direct Mapped Cache
- Types of Misses

Direct Mapped: What is Possible?

- What policies can be implemented for a **direct-mapped cache**? Select all that apply.

1. Block Replacement (Eviction) Policy
   - A. Least Recently Used
   - B. Most Recently Used
   - C. FIFO
   - D. Random
   - E. None of the above

2. Write-Back Policy
   - A. Write-through (memory access per write)
   - B. Write-back (dirty bit and write on eviction)

What policies can be implemented for a direct-mapped cache? Select all that apply.

Replacement: LRU	0%
Replacement: MRU	0%
Replacement: FIFO	0%
Replacement: Random	0%
Replacement: None of the above	0%
Write: Write-through	0%
Write: Write-back	0%

Direct Mapped: What is Possible?

- What policies can be implemented for a **direct-mapped cache**? Select all that apply.

1. Block Replacement (Eviction) Policy
   - A. Least Recently Used
   - B. Most Recently Used
   - C. FIFO
   - D. Random
   - [✓ E.] None of the above

2. Write Policy
   - [✓ A.] Write-through (memory access per write)
   - [✓ B.] Write-back (dirty bit and write to memory on eviction)

In direct mapped caches, there is only ever one block to evict—the existing block with matching index.

Types of Misses

- Compulsory Miss
  - Caused by the first access to a block that has never been in the cache
- Capacity Miss
  - Caused when the cache cannot contain all the blocks needed during the execution of a program
  - Occur when blocks were in the cache, replaced, and later retrieved
- Conflict Miss
  - Multiple blocks compete for the same location in the cache, even when the cache has not reached full capacity.

Types of Misses

- Compulsory Miss
  - Caused by the first access to a block that has never been in the cache
- Capacity Miss
  - Caused when the cache cannot contain all the blocks needed during the execution of a program
  - Occur when blocks were in the cache, replaced, and later retrieved
- Conflict Miss
  - Multiple blocks compete for the same location in the cache, even when the cache has not reached full capacity.

Which types of misses can occur for Fully Associative caches (FA)? For Direct Mapped caches (DM)? Select all that apply.

A. Compulsory
B. Capacity
C. Conflict
D. None of the above
E. All of the above

Which types of misses can occur for Fully Associative caches (FA)? For Direct Mapped caches (DM)? Select all that apply.

FA: Compulsory	0%
FA: Capacity	0%
FA: Conflict	0%
FA: None of the above	0%
DM: Compulsory	0%
DM: Capacity	0%
DM: Conflict	0%
DM: None of the above	0%

Types of Misses

- Compulsory Miss
  - Caused by the first access to a block that has never been in the cache
- Capacity Miss
  - Caused when the cache cannot contain all the blocks needed during the execution of a program
  - Occur when blocks were in the cache, replaced, and later retrieved
- Conflict Miss
  - Multiple blocks compete for the same location in the cache, even when the cache has not reached full capacity.
  - Occurs in direct mapped caches, as well as in (out of scope) set associative caches.
  - Misses of this type would **not occur in a fully associative cache** with similar specs.

Summary: Cache Comparisons

- Fully Associative
  - A specific line of data can be stored in **any line** of the cache.
  - No index.
  - Need to choose replacement policy.
  - Need to choose write policy.
- Direct Mapped
  - A specific line of data can only be stored in **one index** of the cache.
  - Has index.
  - If the line you want to store the data in is occupied, you kick out that line.
  - Need to choose write policy.