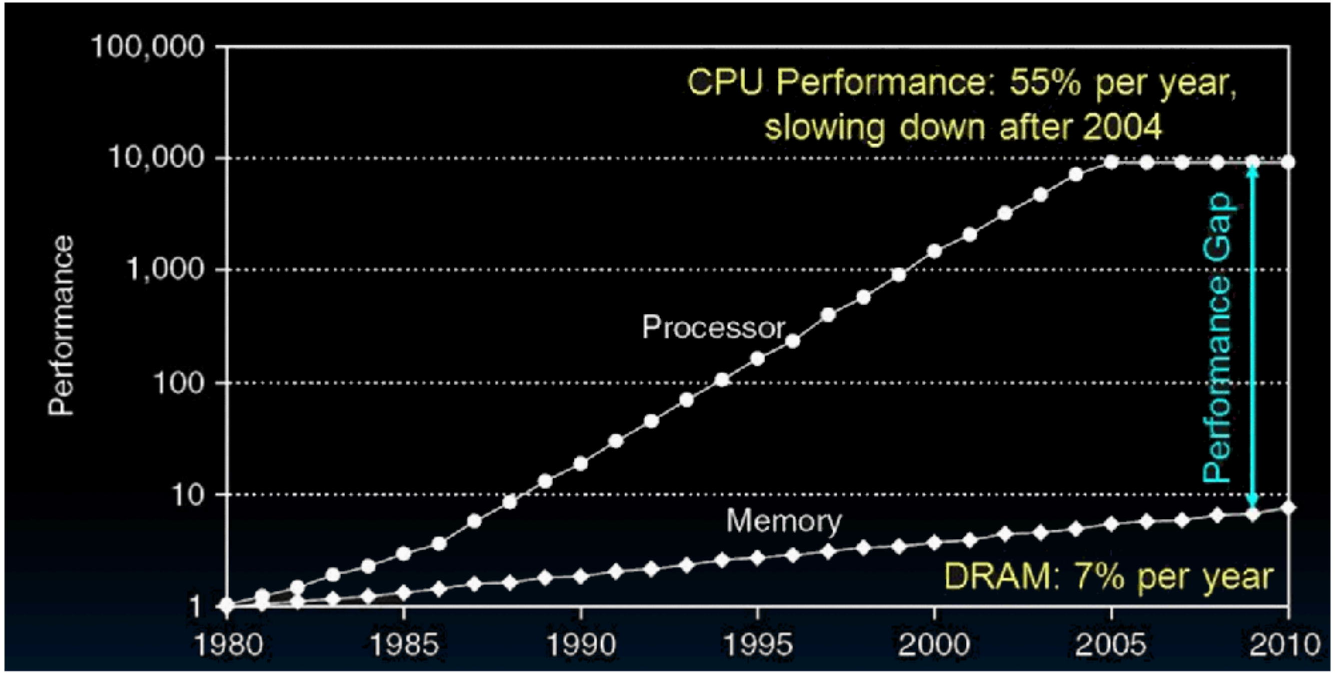
Caches

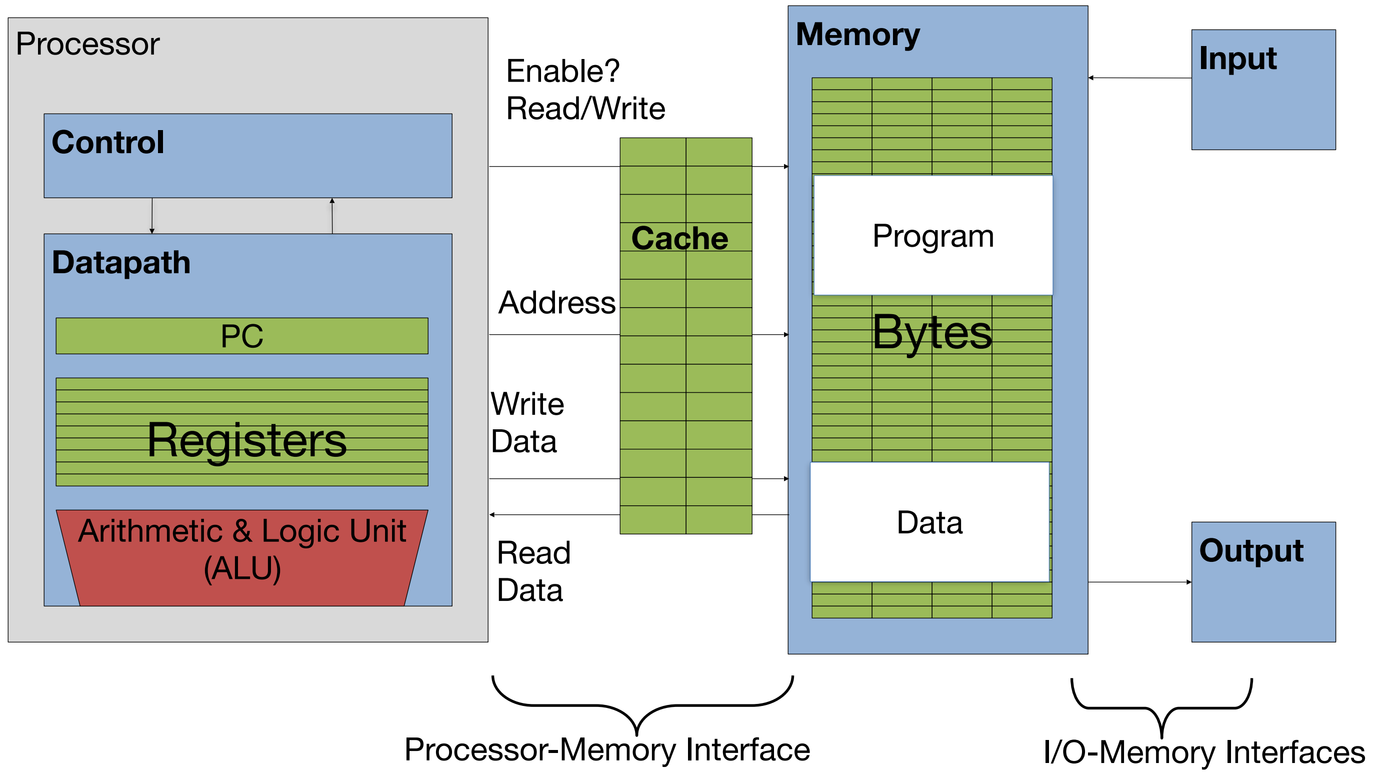
Recall: Processor-DRAM Latency Gap



1980 microprocessor executes ~one instruction in same time as DRAM access
2020 microprocessor executes ~1000 instructions in same time as DRAM access

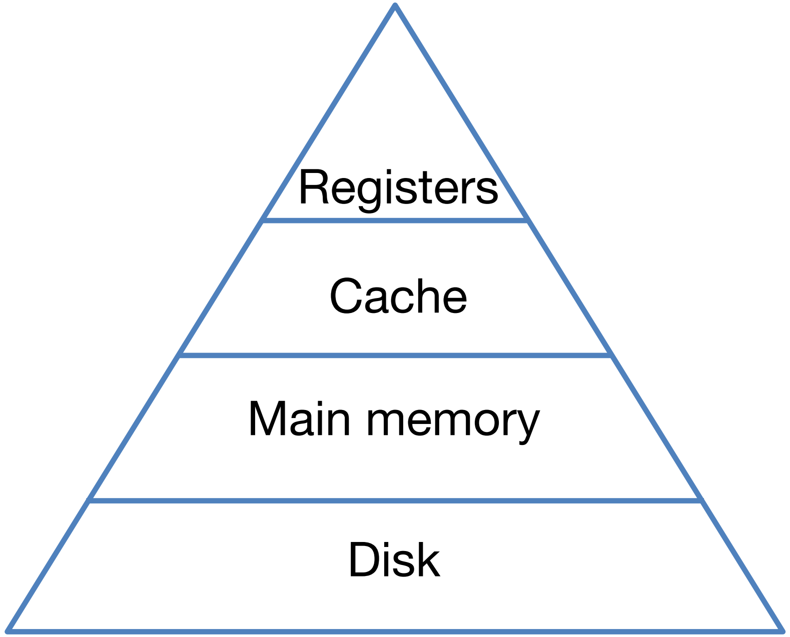
Slow DRAM access could have disastrous impact on CPU performance!

Recall: Adding Cache to the Computer



Recall: Memory Hierarchy

- If level closer to Processor, it is:
  - Smaller
  - Faster
  - More expensive
  - subset of lower levels (contains most recently used data)
- Lowest Level (usually disk=HDD/SSD) contains all available data
- Memory Hierarchy presents the processor with the illusion of a very large & fast memory



Recall: Taking Advantage of Locality

- Temporal Locality
  - If a memory location is referenced then it will tend to be referenced again soon
  - ⇒ Keep most recently accessed data items closer to the processor
- Spatial Locality
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
  - ⇒ Move blocks consisting of contiguous words closer to the processor

Recall: Memory Access without Cache

- Load word instruction: `lw t0 0(t1)`
- `t1` contains `0x12F0`, Memory[`0x12F0`] = `99`

1. Processor issues address `0x12F0` to Memory
2. Memory reads word at address `0x12F0` (`99`)
3. Memory sends `99` to Processor
4. Processor loads `99` into register `t0`

Recall: Memory Access with Cache

- Load word instruction: `lw t0,0(t1)`
- t1 contains 0x12F0, Memory[0x12F0] = 99
- With cache: Processor issues address 0x12F0 to Cache
  1. Cache checks to see if has copy of data at address 0x12F0
     2a. If finds a match (Hit): cache reads 99, sends to processor
     2b. No match (Miss): cache sends address 0x12F0 to Memory
        I. Memory reads 99 at address 0x12F0
        II. Memory sends 99 to Cache
        III. Cache replaces word which can store 0x12F0 with new 99
        IV. Cache sends 99 to processor
  2. Processor loads 99 into register t0

Recall: Terminology

- Cache Hit
  - The data you were looking for is in the cache
- Cache Miss
  - The data you were looking for is not in the cache
- Eviction
  - Removing an entry from the cache

Terminology

- Hit Rate
  - number of hits / number of accesses
- Miss Rate
  - 1 - hit rate
- Hit Time
  - The time that it takes for you to access an item on a cache hit
- Miss penalty
  - On a miss, the time it takes to access the block after discovering that its not in the cache

Recall: Terminology

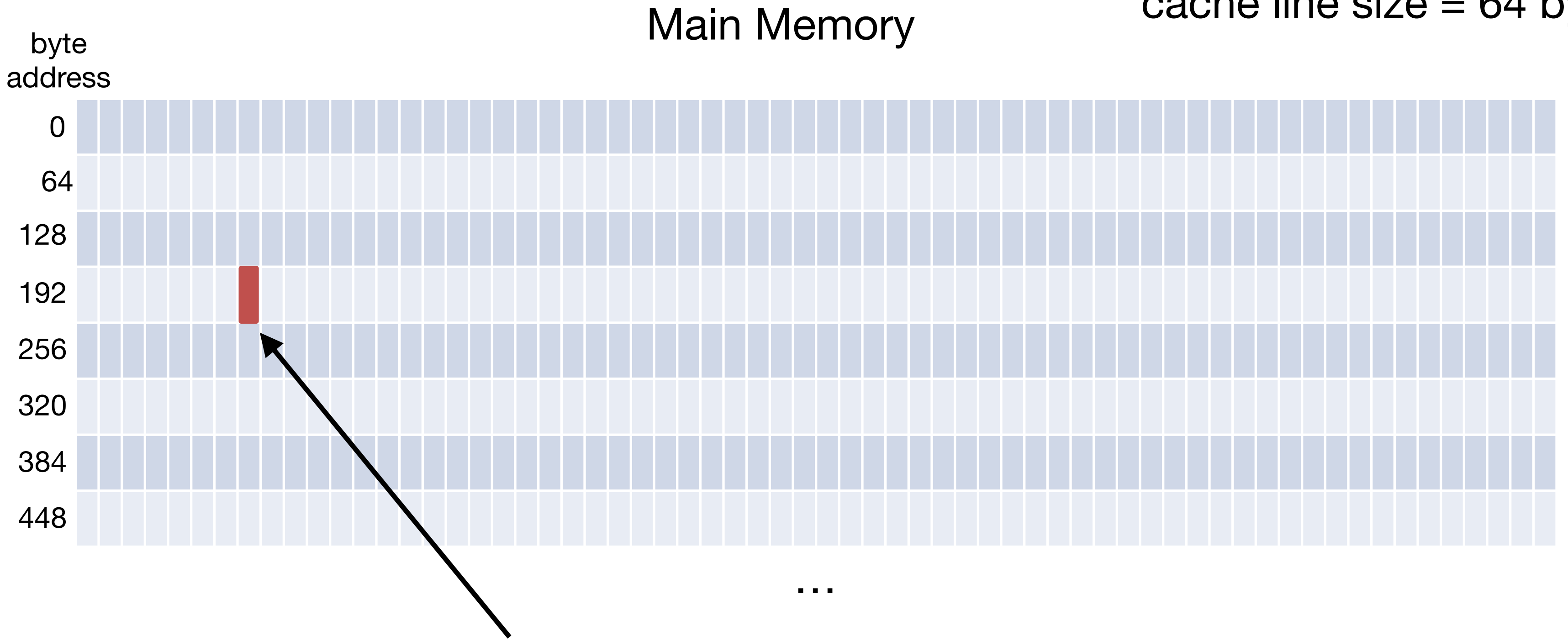
- Cache line/block
  - The smallest unit of memory that can be transferred between the main memory and the cache
  - Each line has its own entry in the cache
- Line size/block size
  - The number of bytes in each cache line
- Capacity
  - The total number of data bytes that can be stored in a cache
  - For fully associative cache, capacity = # lines * line size

Cache Lines

- 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
- This helps us take advantage of spatial locality

Cache Lines

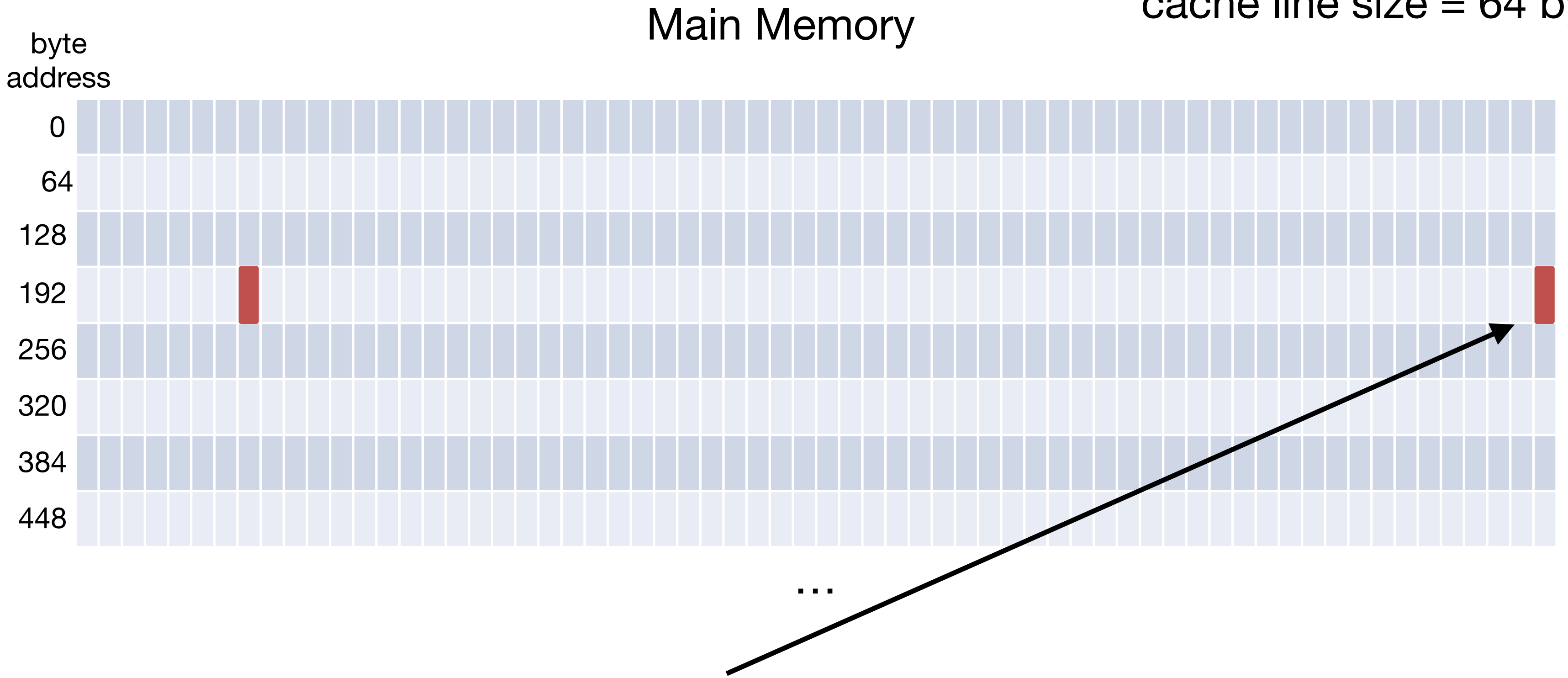
cache line size = 64 bytes



If we want to access byte 199_{10} and its not in the cache, we would bring in bytes 192_{10}-255_{10} into the cache

Cache Lines

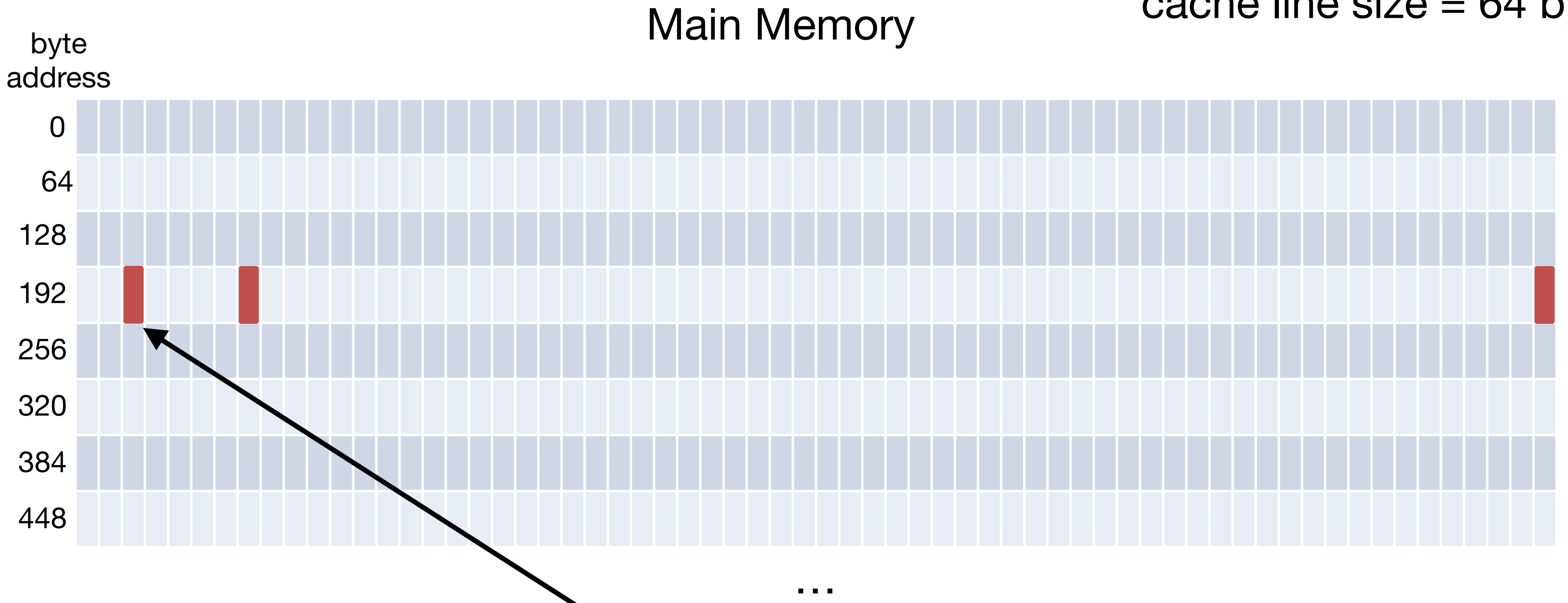
cache line size = 64 bytes



If we then wanted to access byte 255_{10}, we would get a cache hit because we just brought in the line that its in

Cache Lines

cache line size = 64 bytes



If we then wanted to access byte 194_{10}, we would get a cache hit because we just brought in the line that its in

Recall: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator to tell if each entry is valid for this program
- 1 = valid
- 0 = invalid

Recall: Tag

- Every line in the cache has a tag which helps us identify if the memory address that we are trying to access is stored in the cache
- To check if our address matches a given line, we need to verify that the valid bit of that line is one and check if the tags are equivalent

Recall: Write-through vs Write-back Policies

- Write-through
  - Write to the cache and the memory at the same time
  - The write to memory will take longer
  - Very simple to implement
- Write-back
  - Write data in cache and set the dirty bit to 1
  - When this line gets evicted from the cache, write it to memory
  - Typically lowers traffic to the memory because you might write to something multiple times before you evict it from the cache

additional metadata needed

Write-allocate vs No write-allocate

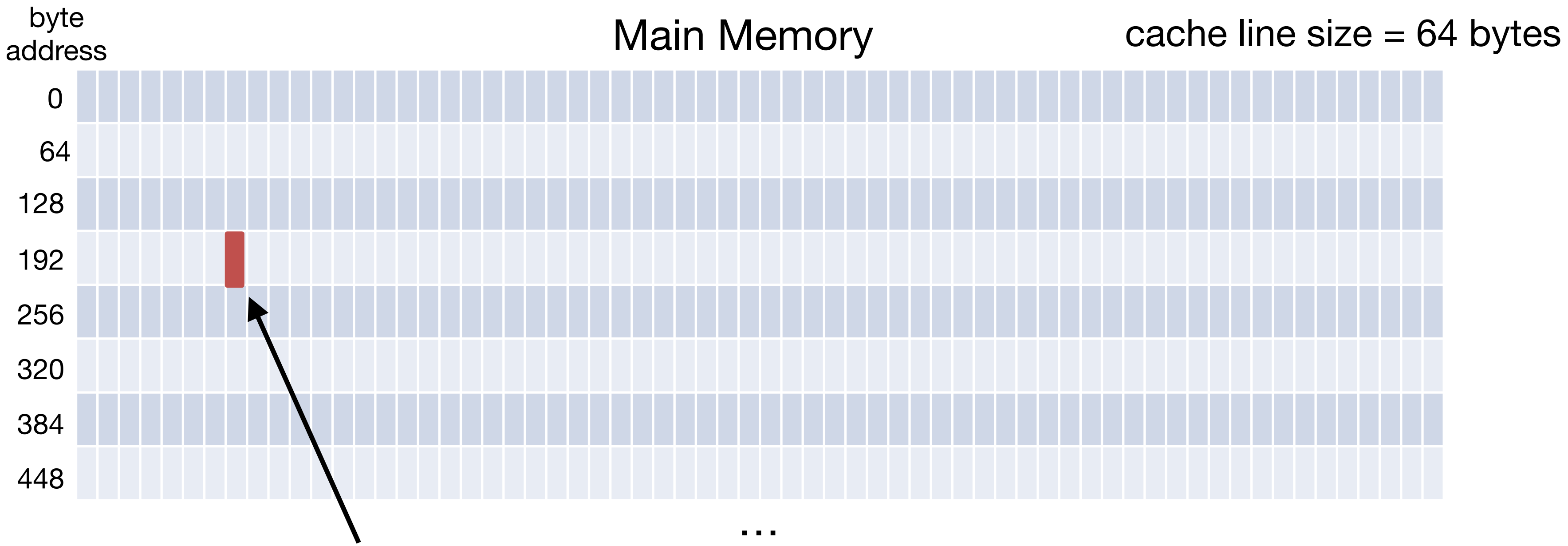
- Write-allocate
  - On a write miss, you bring the line into the cache and then update the line
- No write-allocate
  - On a write miss, don't bring the line into the cache, you only update memory
- For both, you always bring the line in on a read miss

Common Combinations

- Write through, no write-allocate
  - When there are write hits, the cache and main memory are updated
  - On write misses, the block is not brought into the cache, and main memory is updated
  - On read misses, the line is still brought into memory
- Write back, write-allocate
  - On read and write misses, the line is brought into the cache
  - On writes, you only update the cache and set the dirty bit to 1

Write allocate

- When you write a byte, you'll read the whole line in from memory, store it in the cache, and then update the byte



If we want to write to byte 199_{10} and its not in the cache, we would bring in bytes 192_{10}-255_{10} into the cache and then update the byte

Recall: Eviction Policies

- Least-Recently Used
  - Replace the entry that has not been used for the longest time
  - Hardware keeps track of access history
  - Requires additional metadata

Approximate LRU

- Most caches have 100s or 1000s of lines
- Keeping track of the order in which each entry was accessed consumes a lot of bits and requires that you update every entry on each accesses
- Instead of implementing a true LRU algorithm, we implement an approximate LRU algorithm

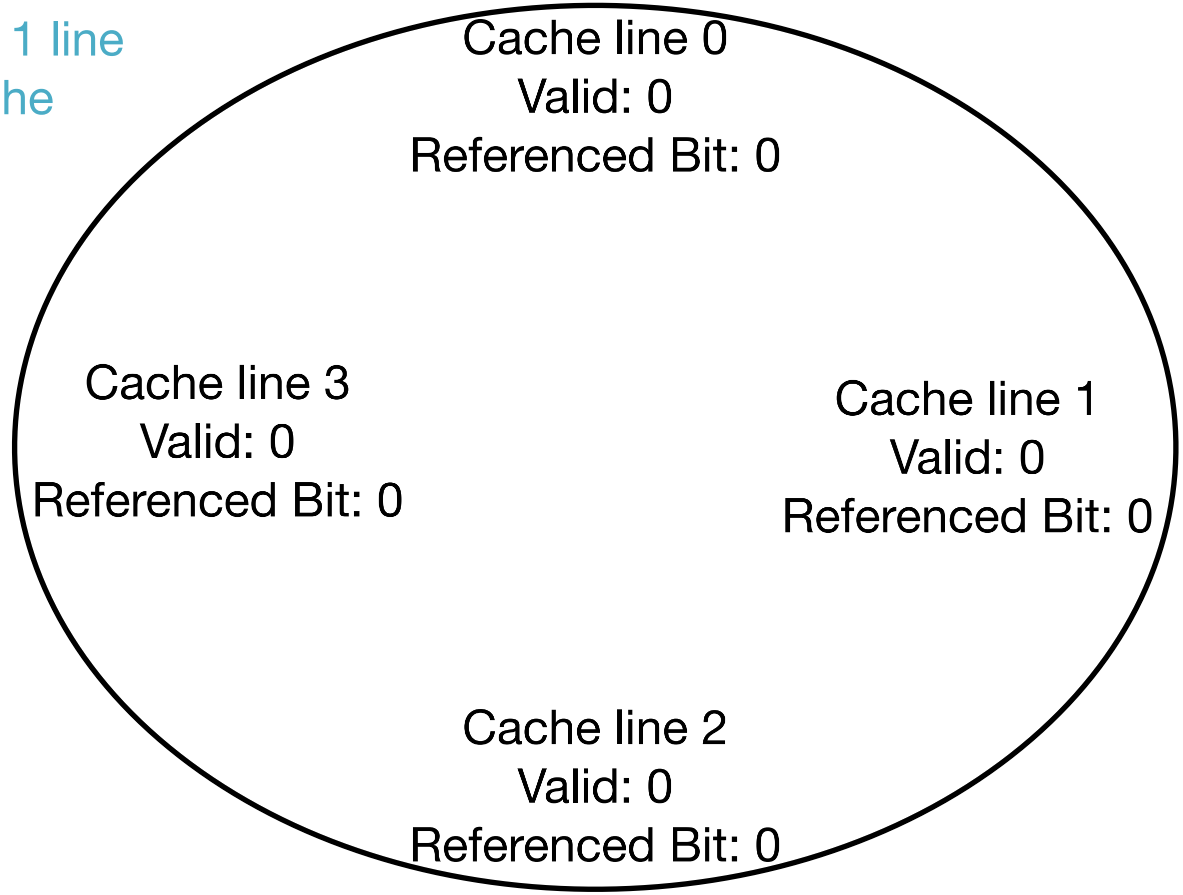
Approximate LRU (Clock Algorithm)

- Each cache line has 1 referenced bit
- This bit is set to 1 each time the line is accessed
- When a line needs to be evicted, we start at the first entry in the cache and check its referenced bit
  - If the bit is 1, we set the bit to 0 and move to the next line
  - If the bit is 0, we choose this line as the victim
- The next time we need to evict a line from the cache, we will start searching at the point where we left off
- (note that we don't start evicting lines until all entries in the cache are valid)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

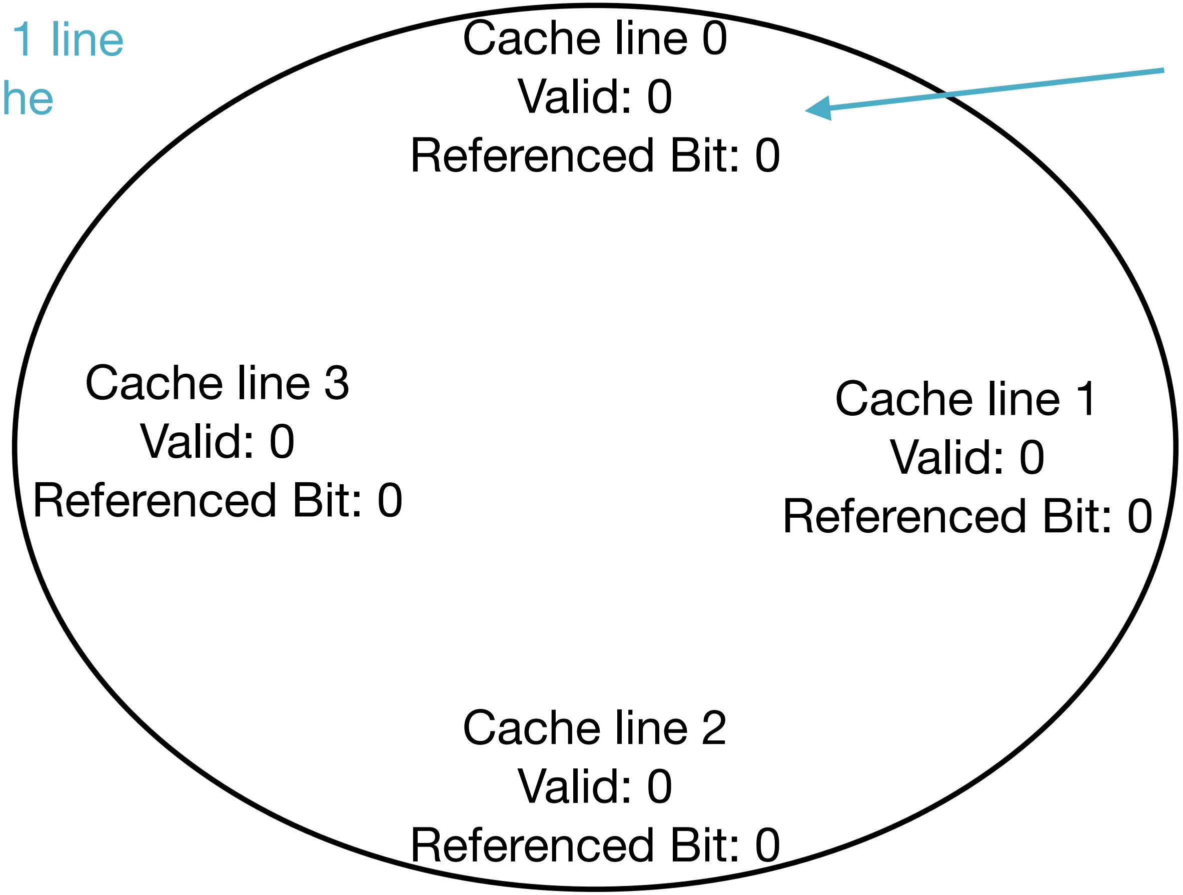
Action 1: Bring 1 line into the cache



Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 1: Bring 1 line into the cache



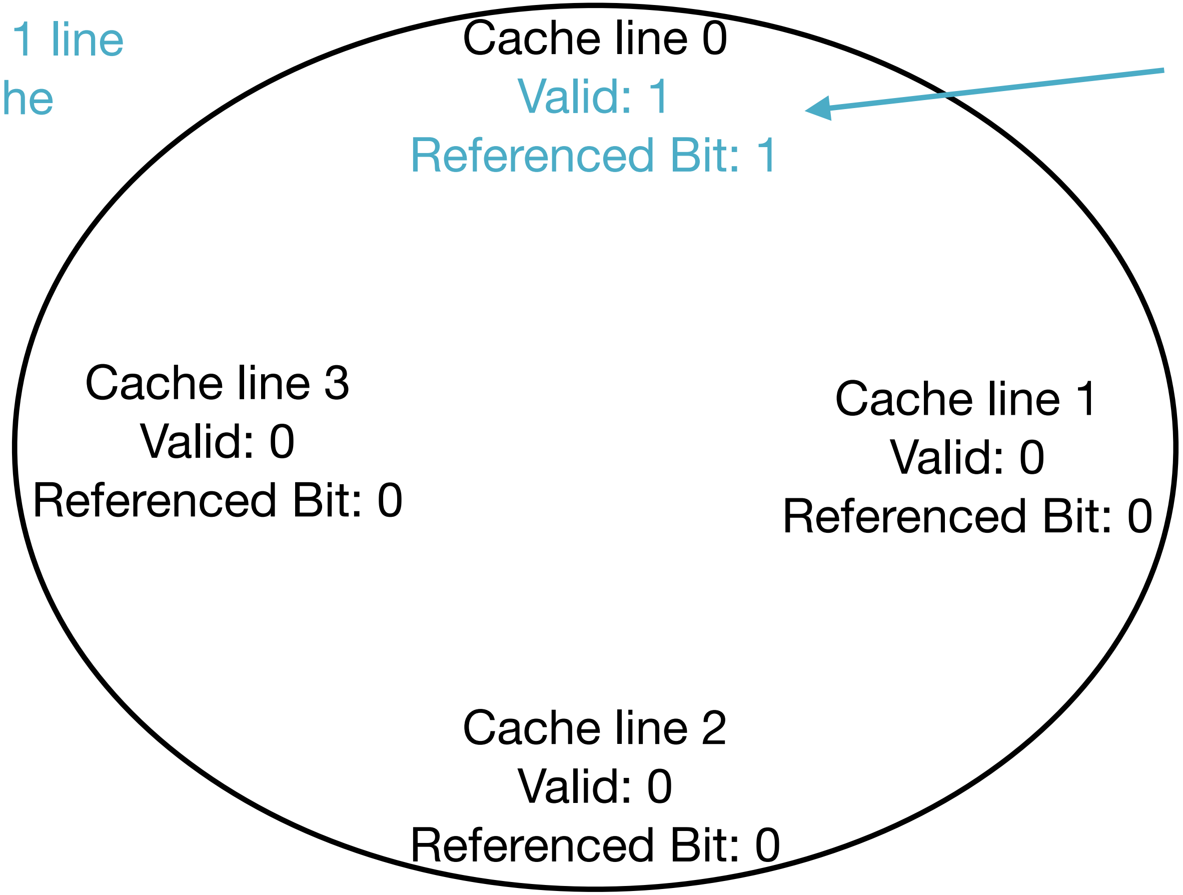
Cache line 0 is invalid, so we'll use this line

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 1: Bring 1 line into the cache

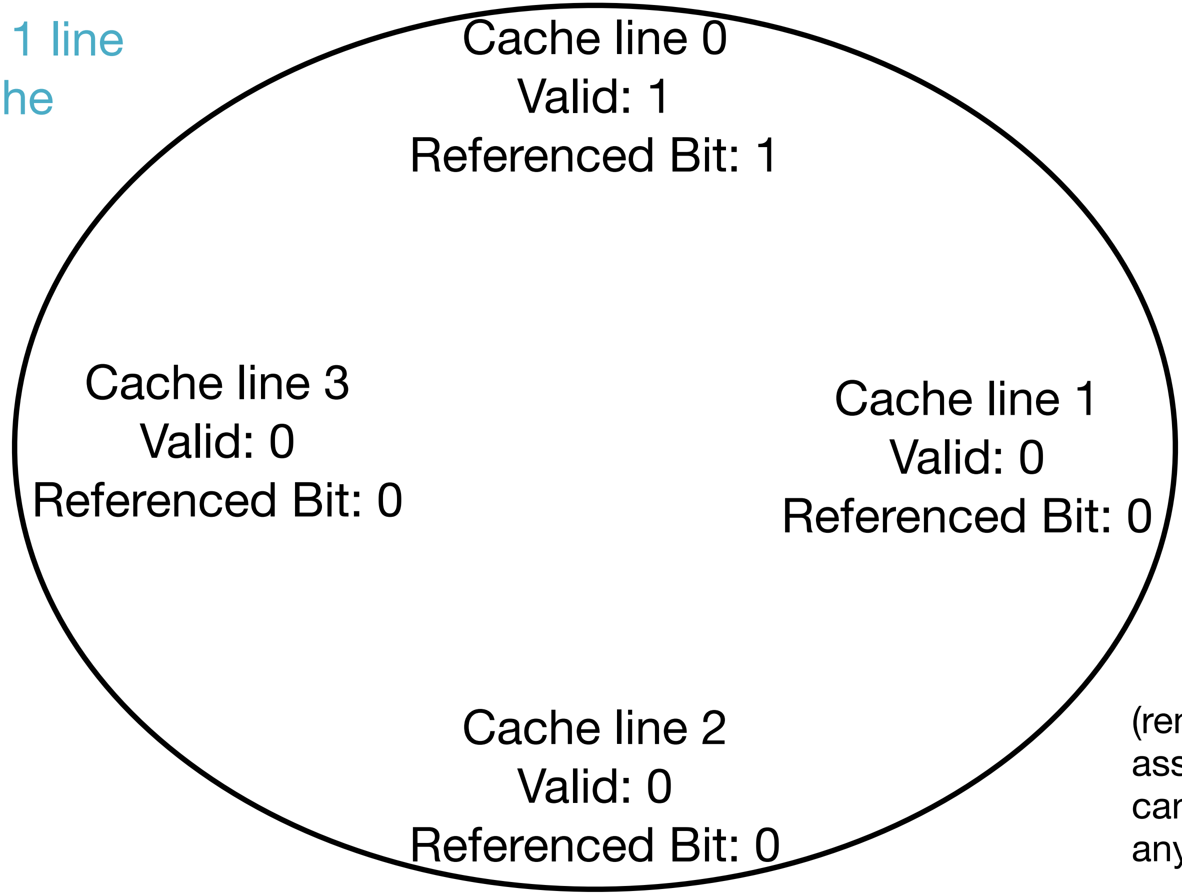
Cache line 0 is invalid, so we'll use this line



Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 2: Bring 1 line into the cache

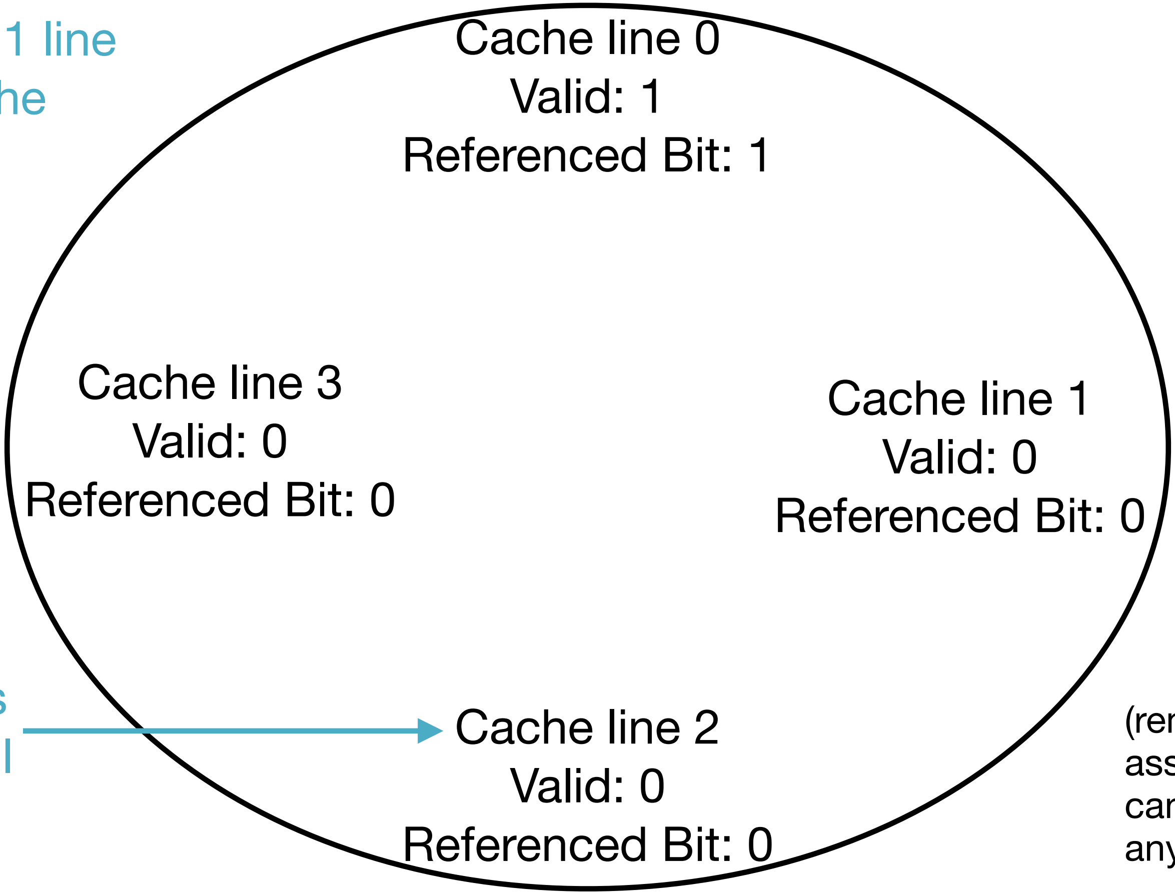


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 2: Bring 1 line into the cache



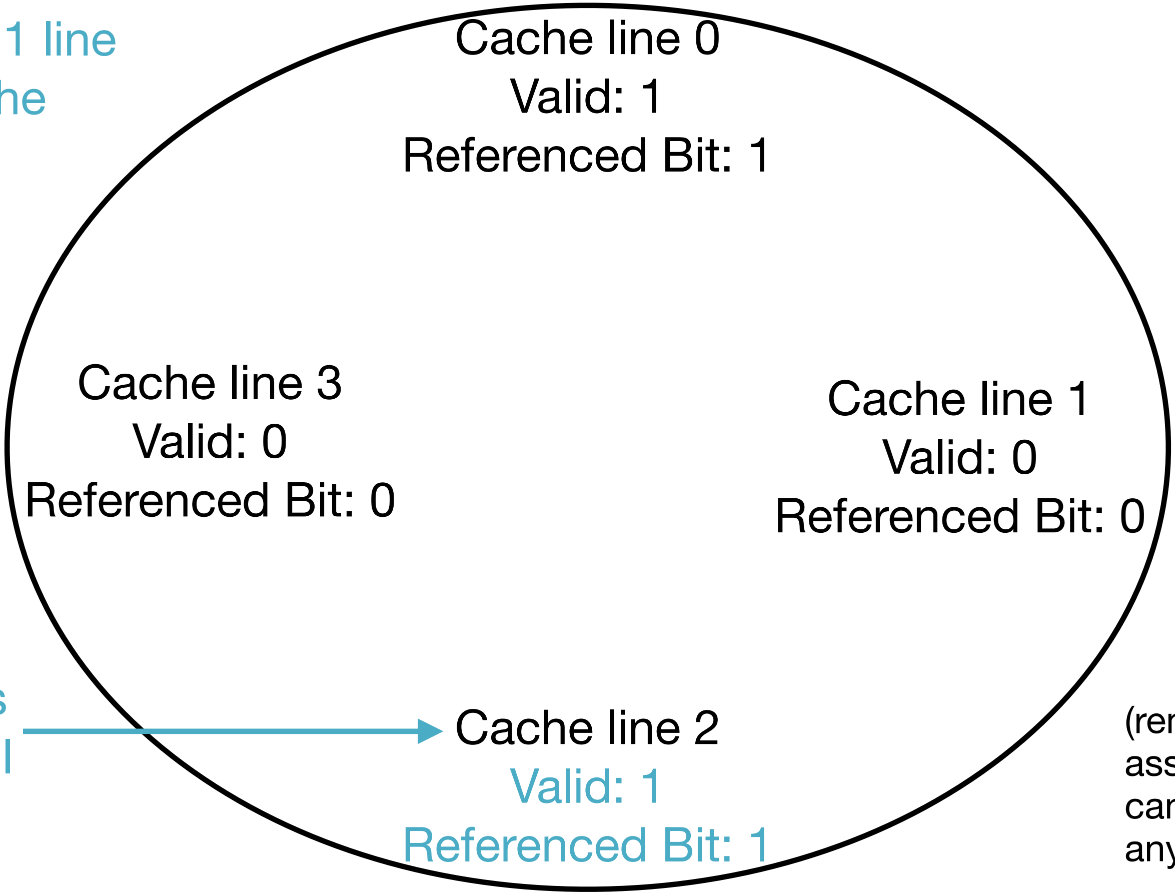
Cache line 2 is invalid, so we'll use this line

(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 2: Bring 1 line into the cache



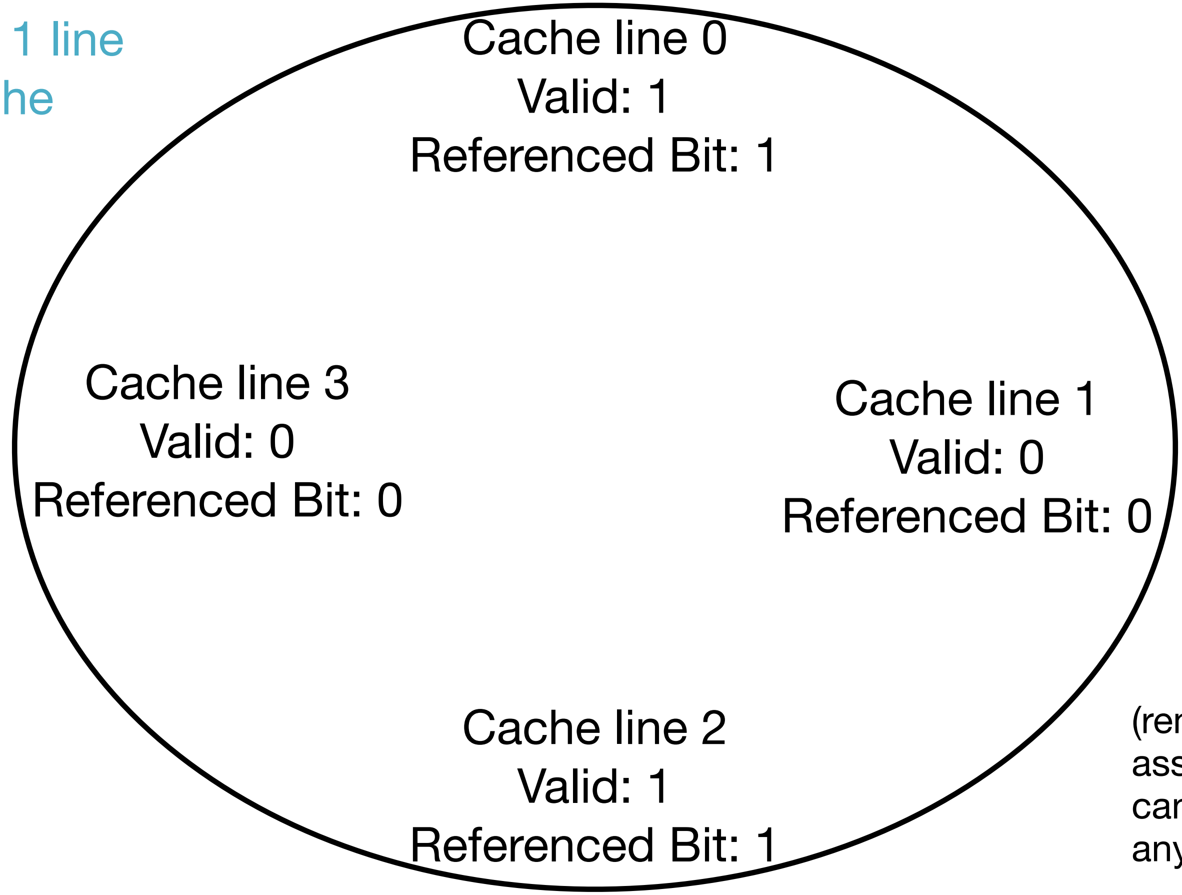
Cache line 2 is invalid, so we'll use this line

(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 3: Bring 1 line into the cache



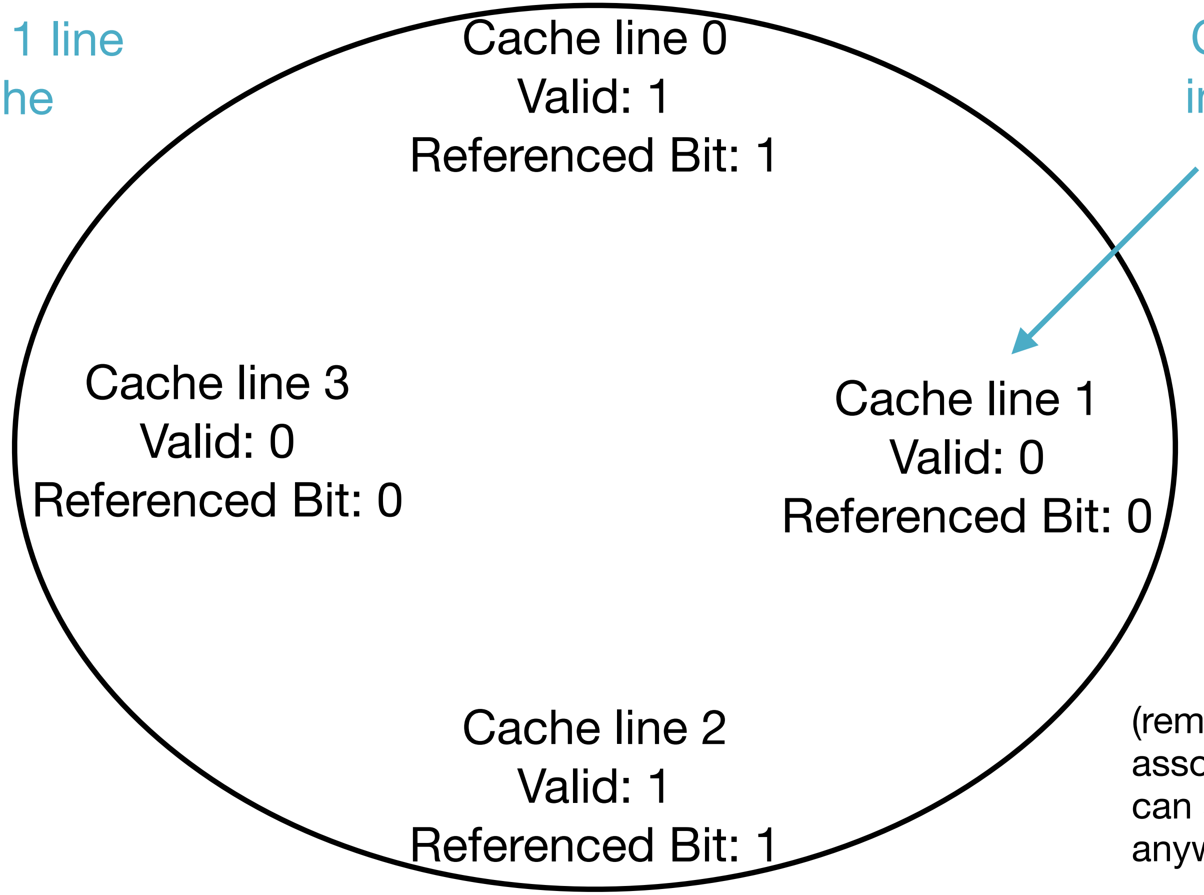
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 3: Bring 1 line into the cache

Cache line 1 is invalid, so we'll use this line



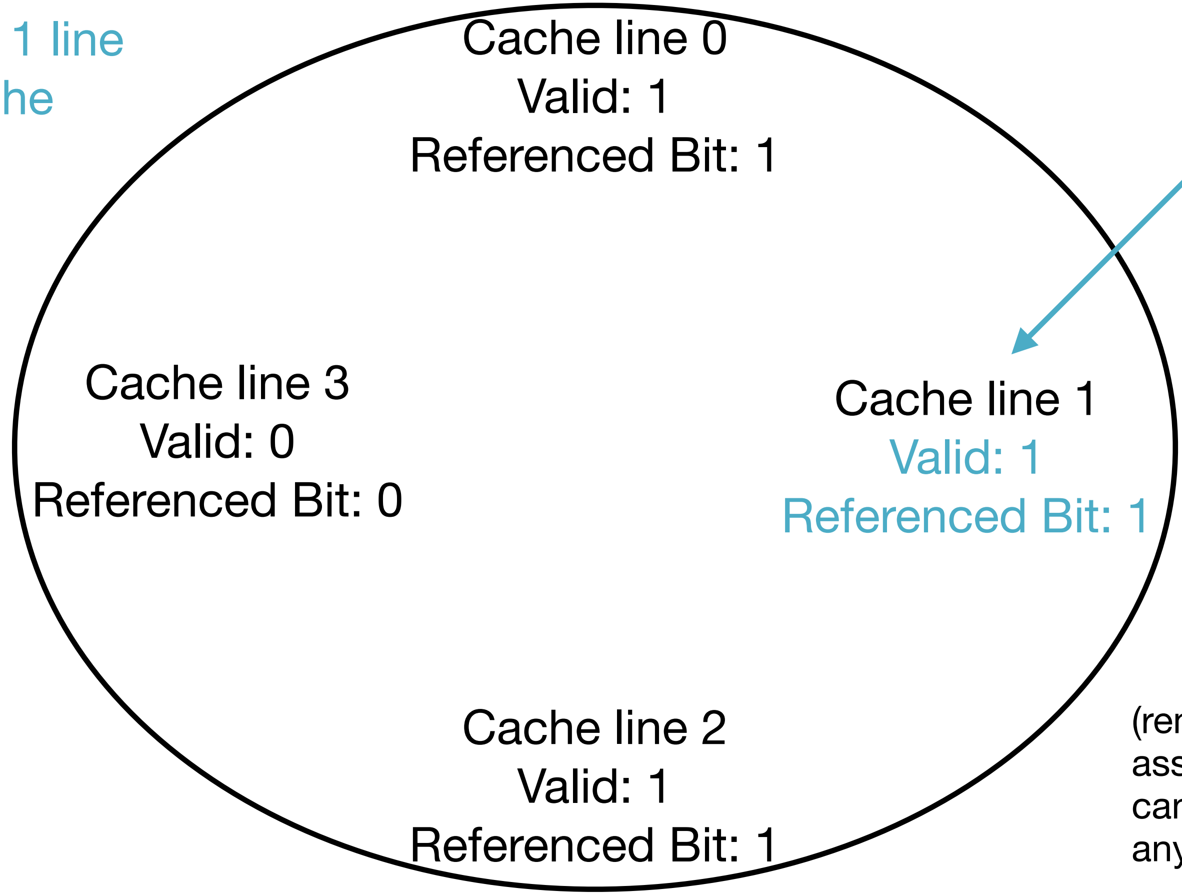
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 3: Bring 1 line into the cache

Cache line 1 is invalid, so we'll use this line

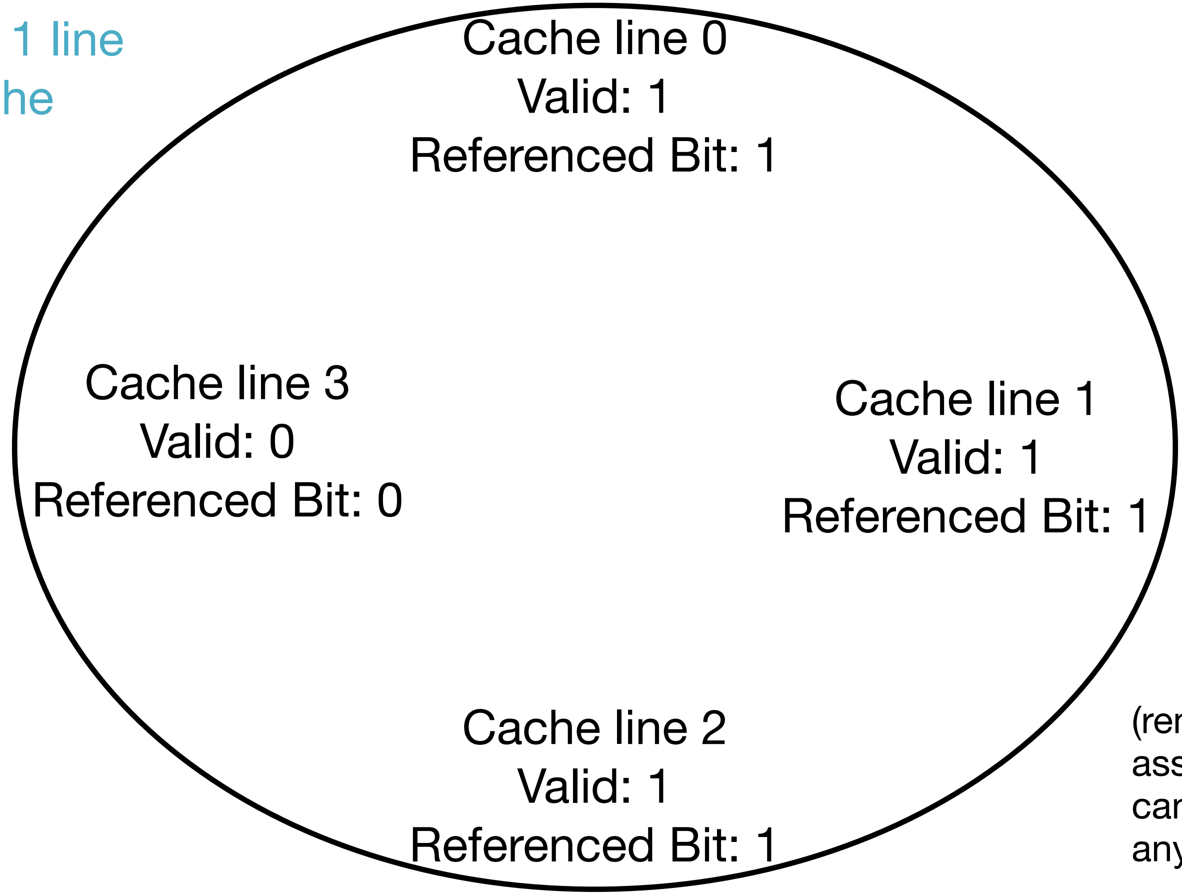


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 4: Bring 1 line into the cache

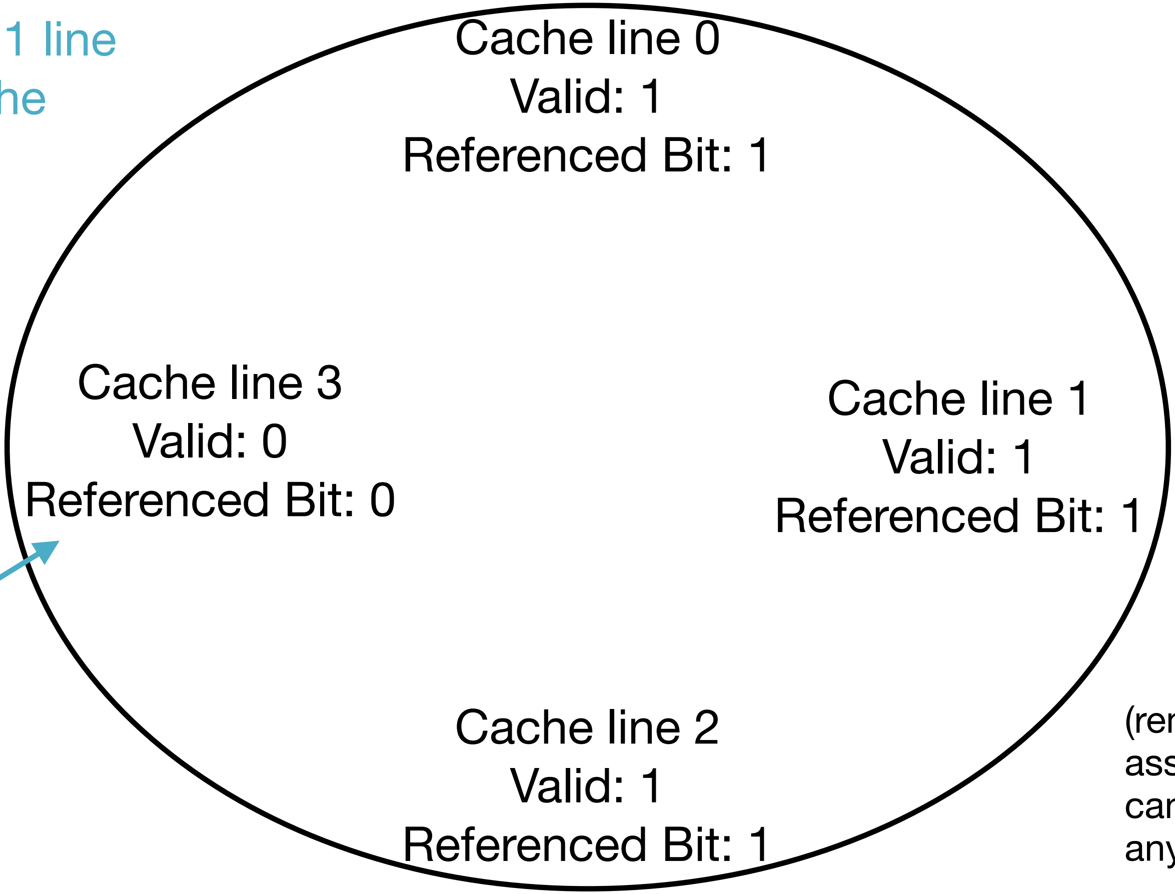


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 4: Bring 1 line into the cache



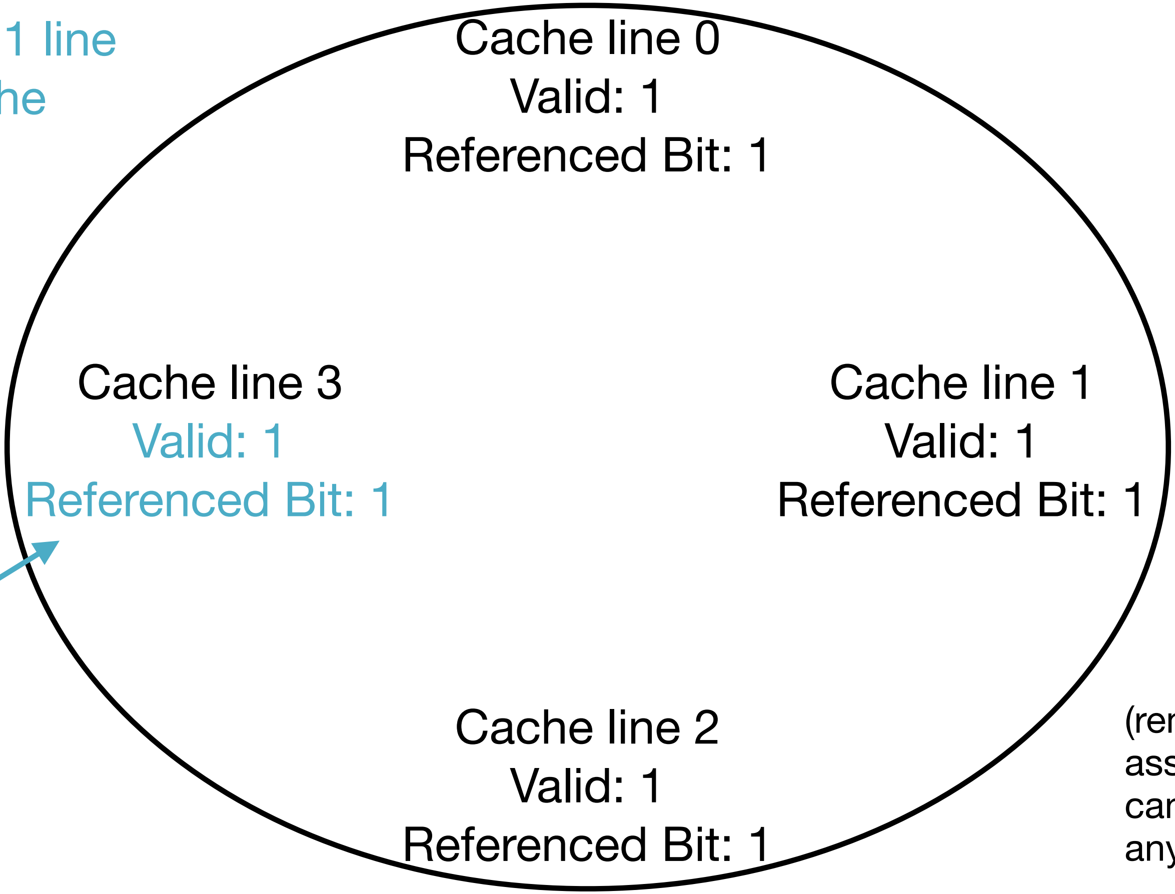
Cache line 3 is invalid, so we'll use this line

(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 4: Bring 1 line into the cache



Cache line 3 is invalid, so we'll use this line

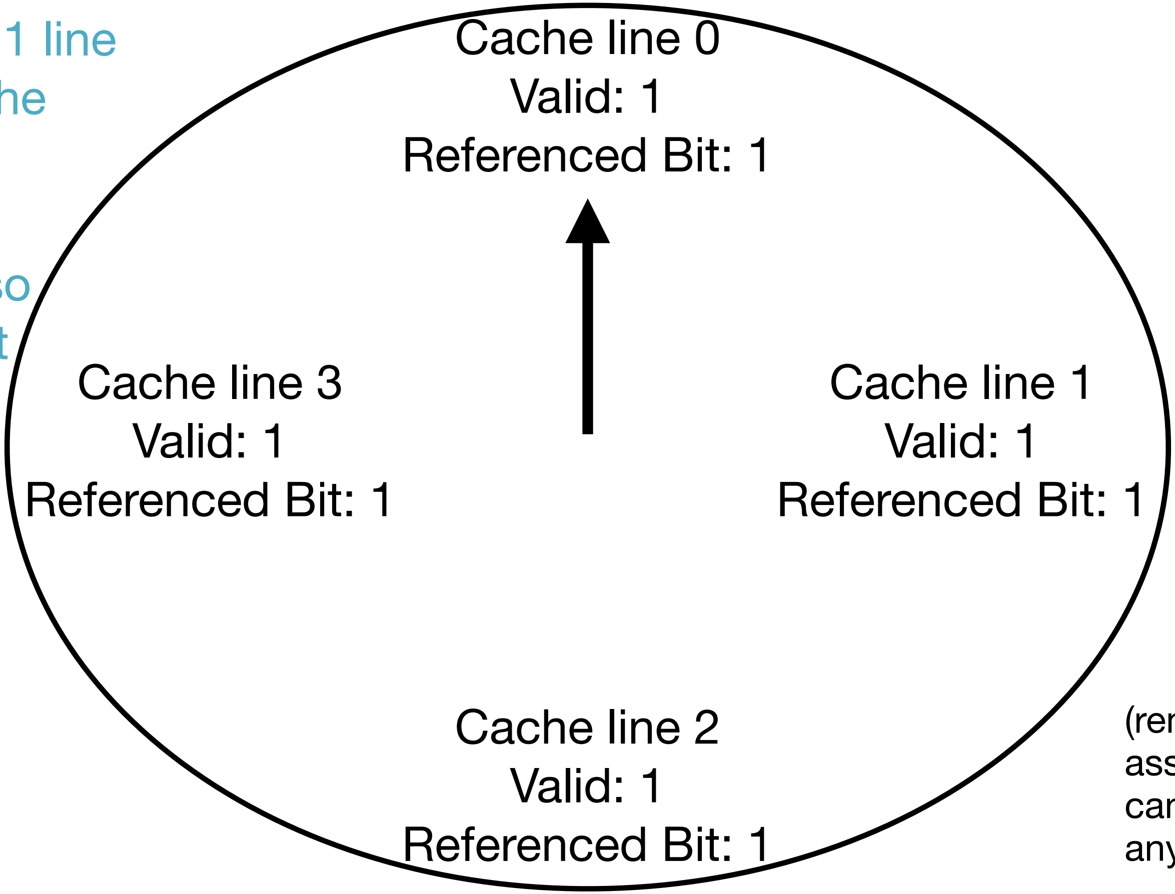
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



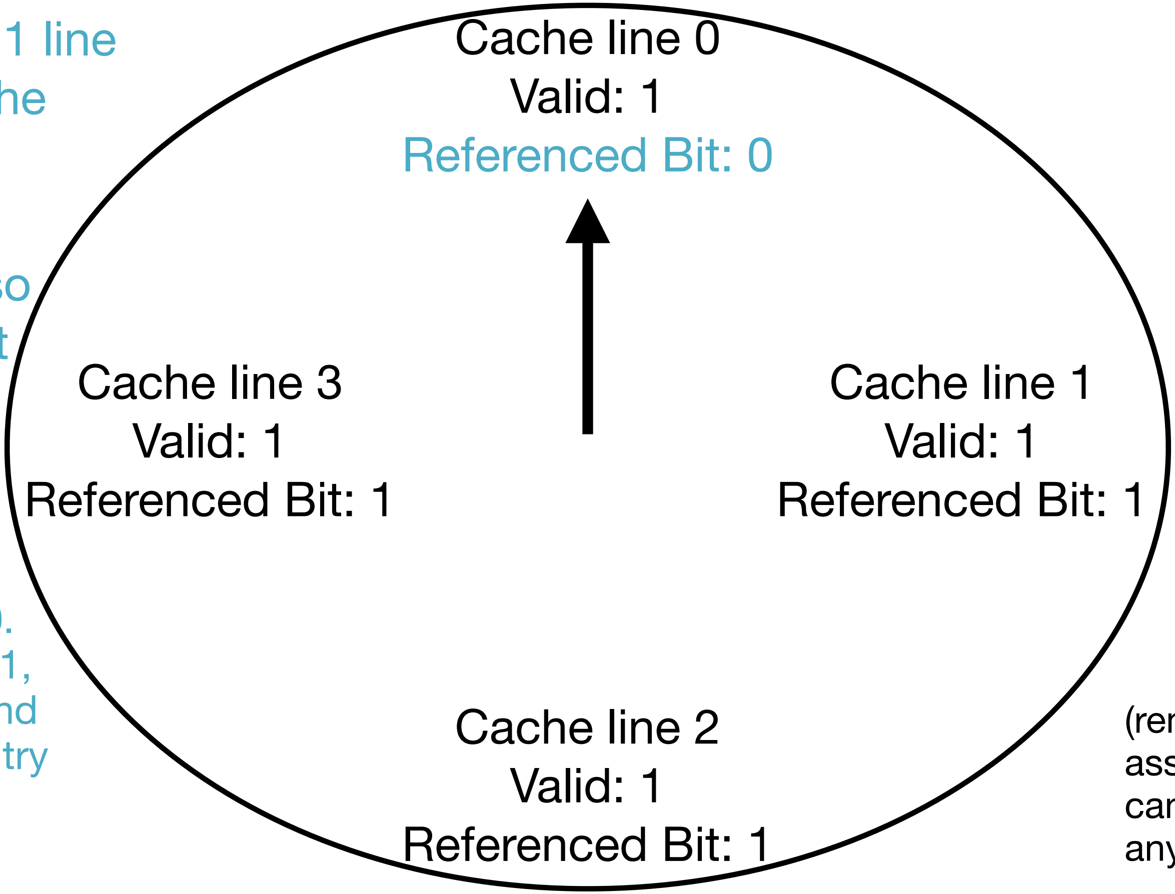
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



Start at cache line 0. The reference bit is 1, so we'll set it to 0 and move to the next entry

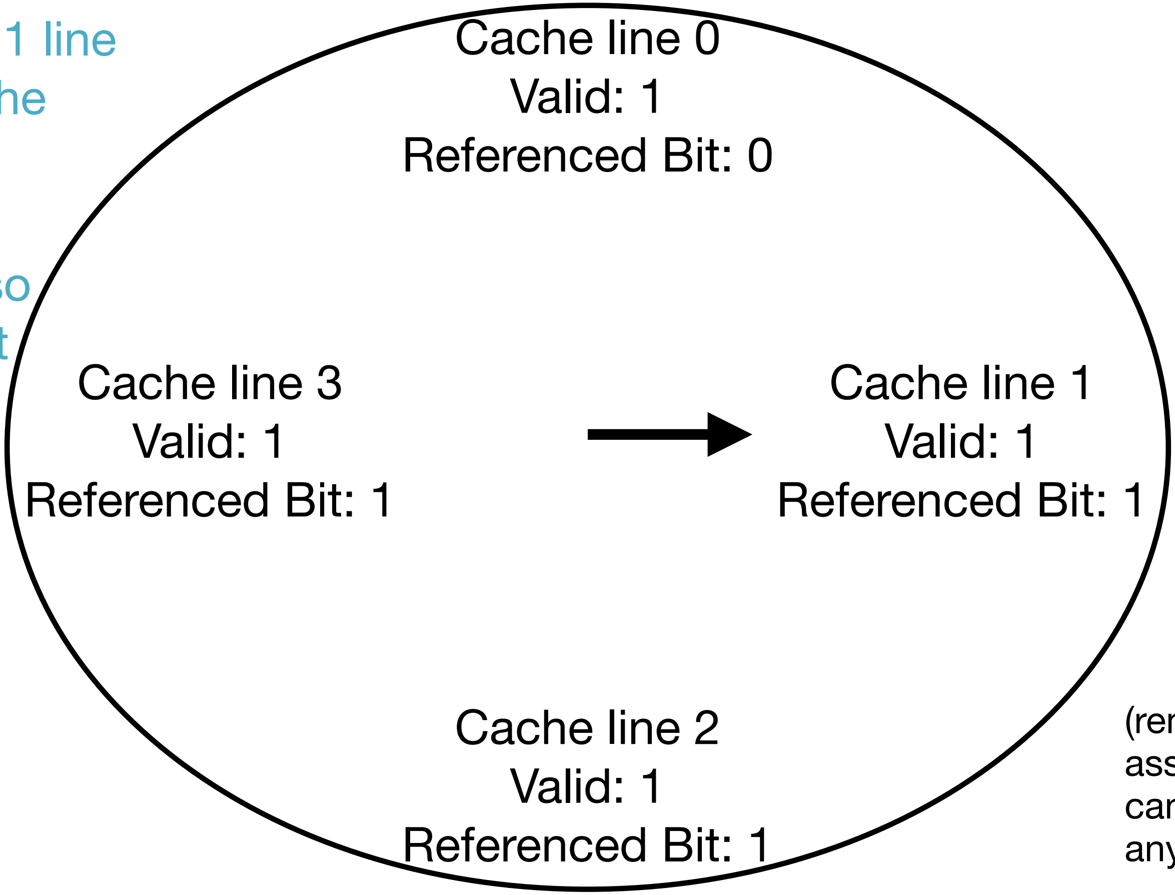
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



(remember its fully associative, so we can store the data anywhere)

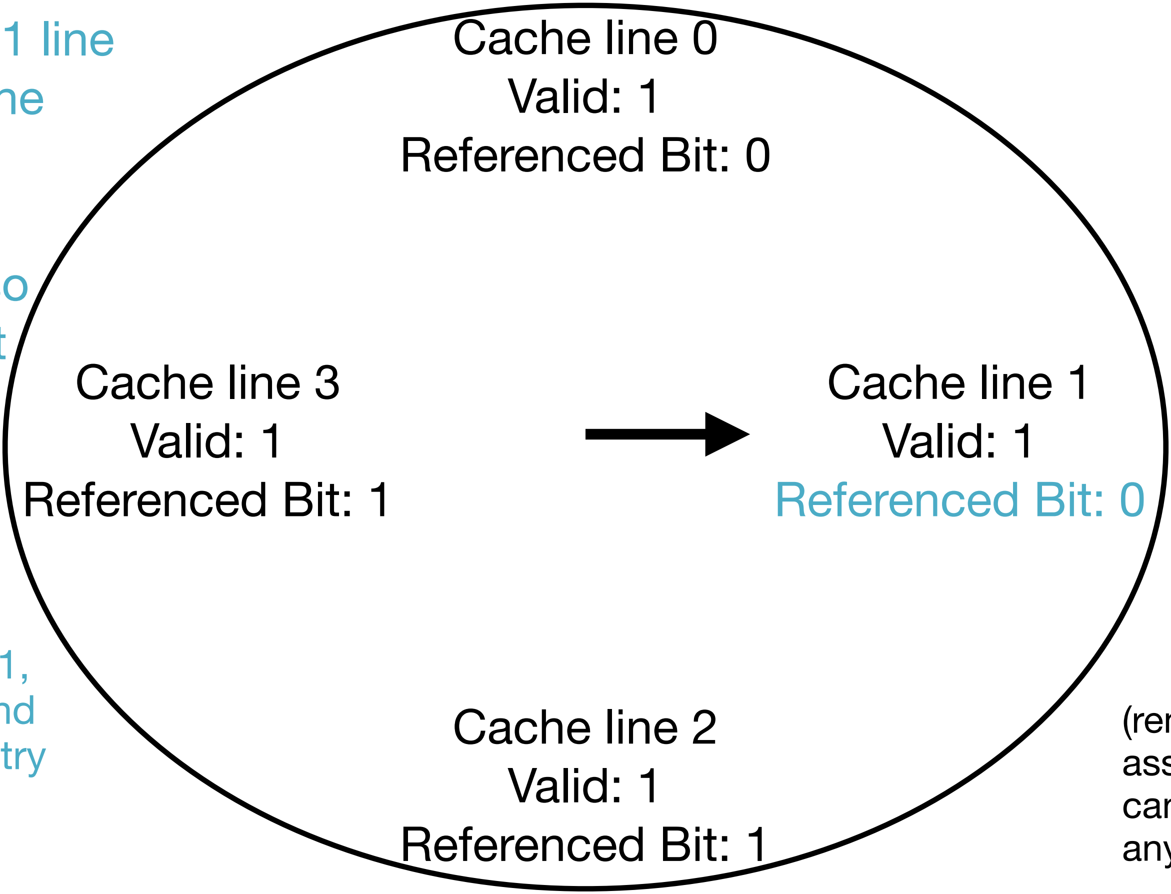
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one

Now we're at line 1. The reference bit is 1, so we'll set it to 0 and move to the next entry



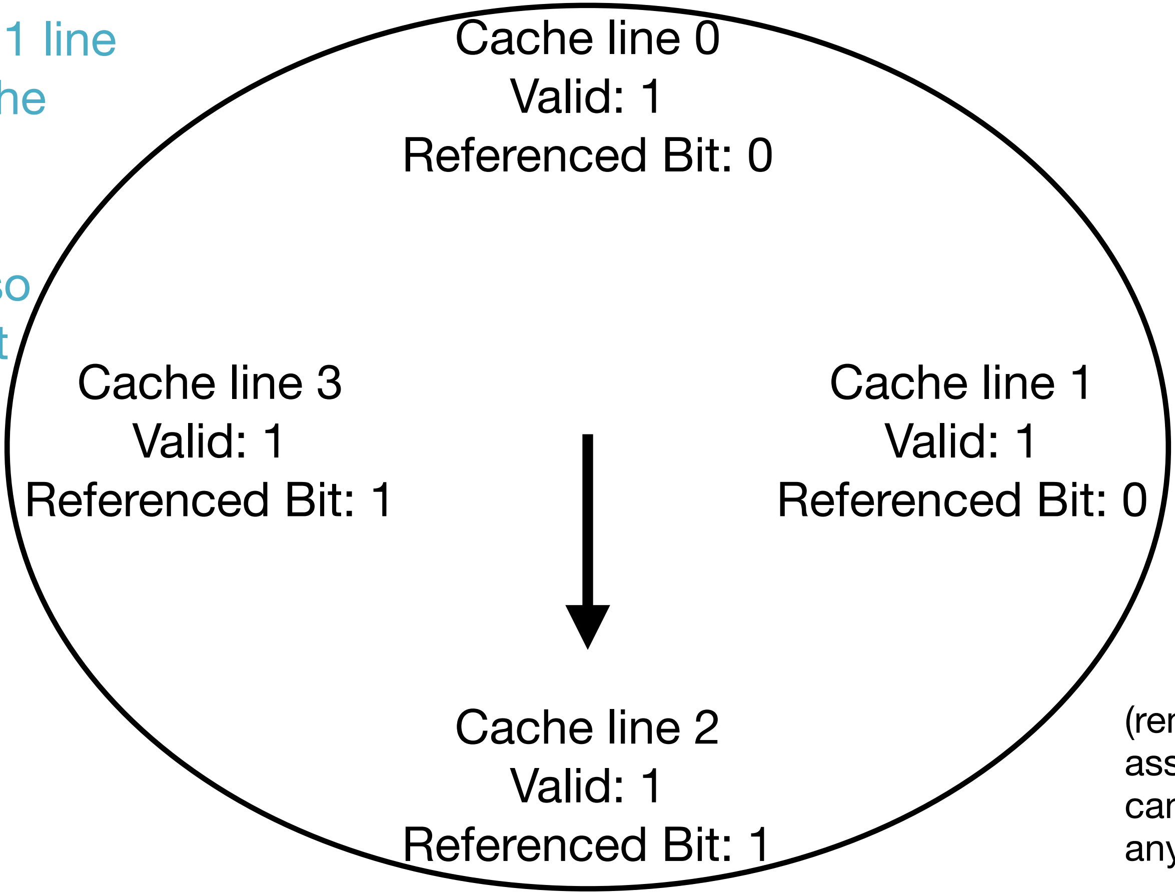
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



(remember its fully associative, so we can store the data anywhere)

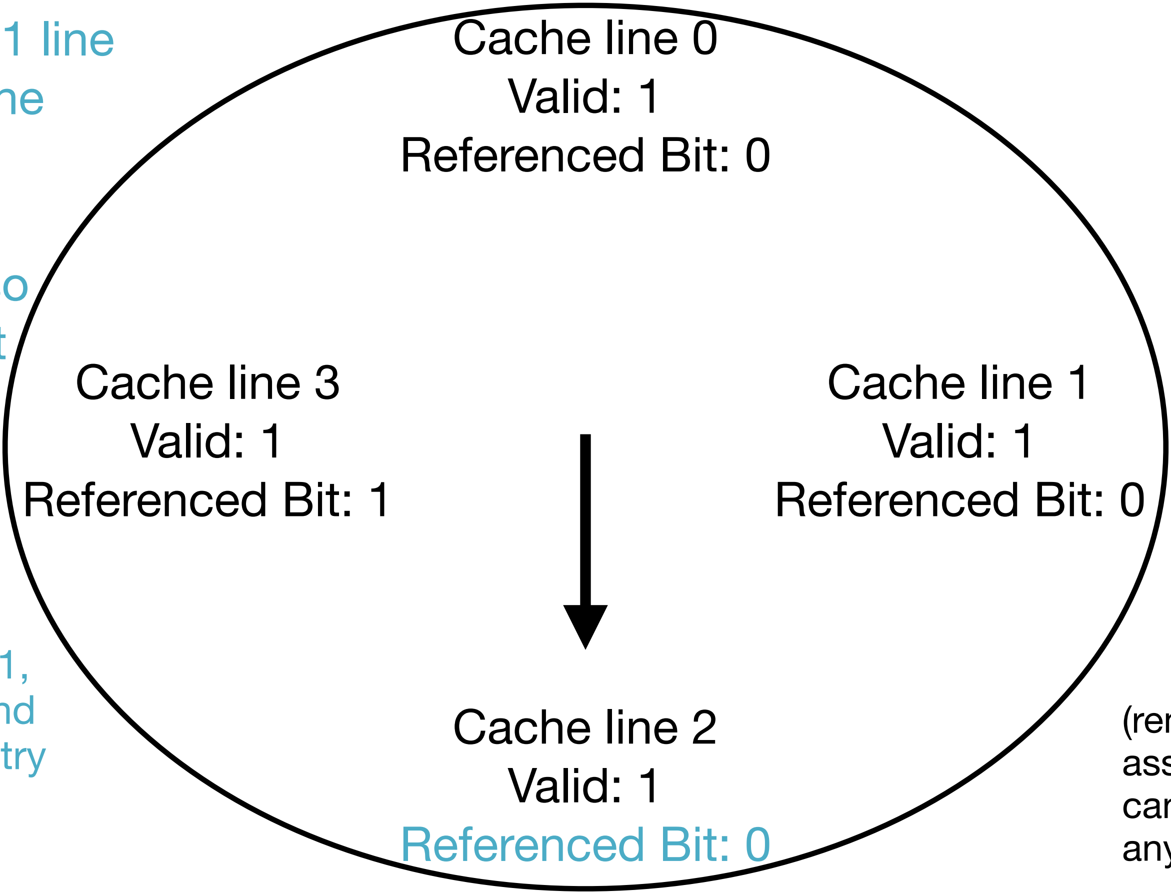
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one

Now we're at line 2. The reference bit is 1, so we'll set it to 0 and move to the next entry



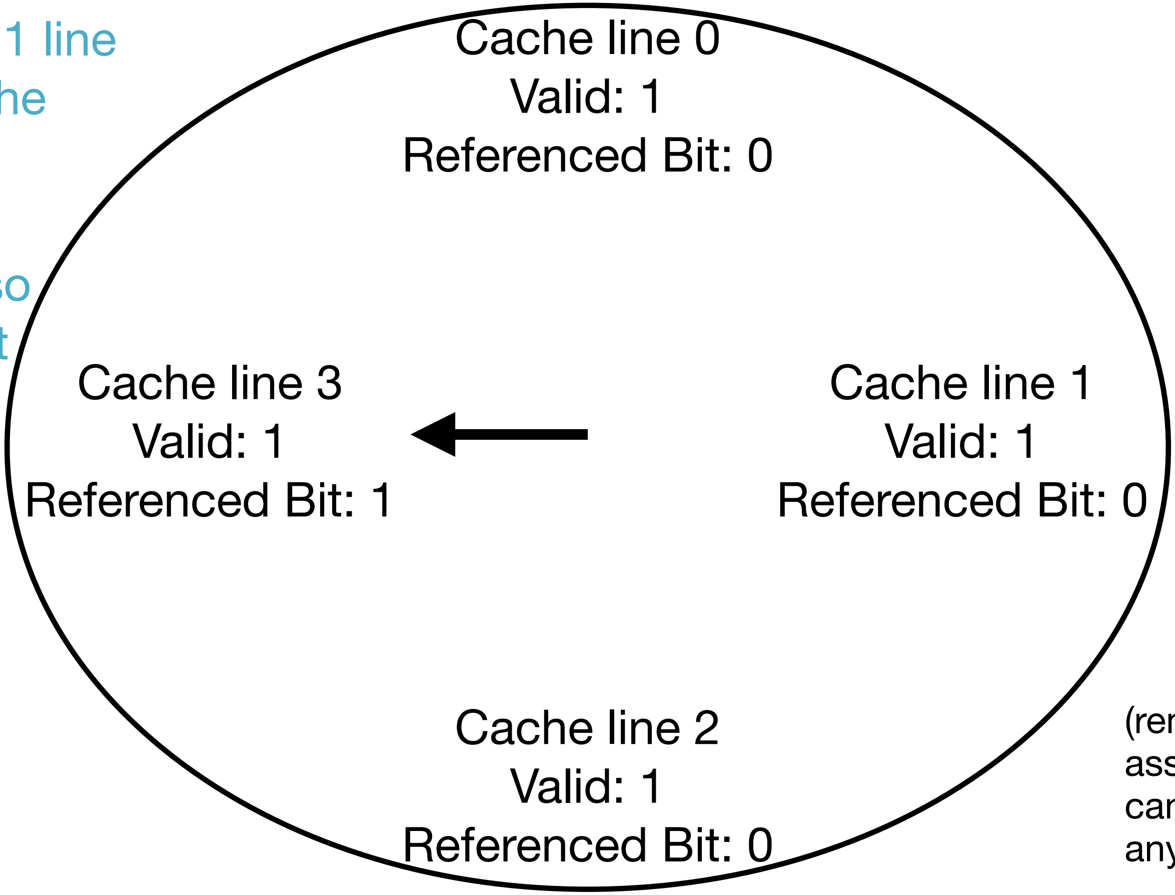
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



(remember its fully associative, so we can store the data anywhere)

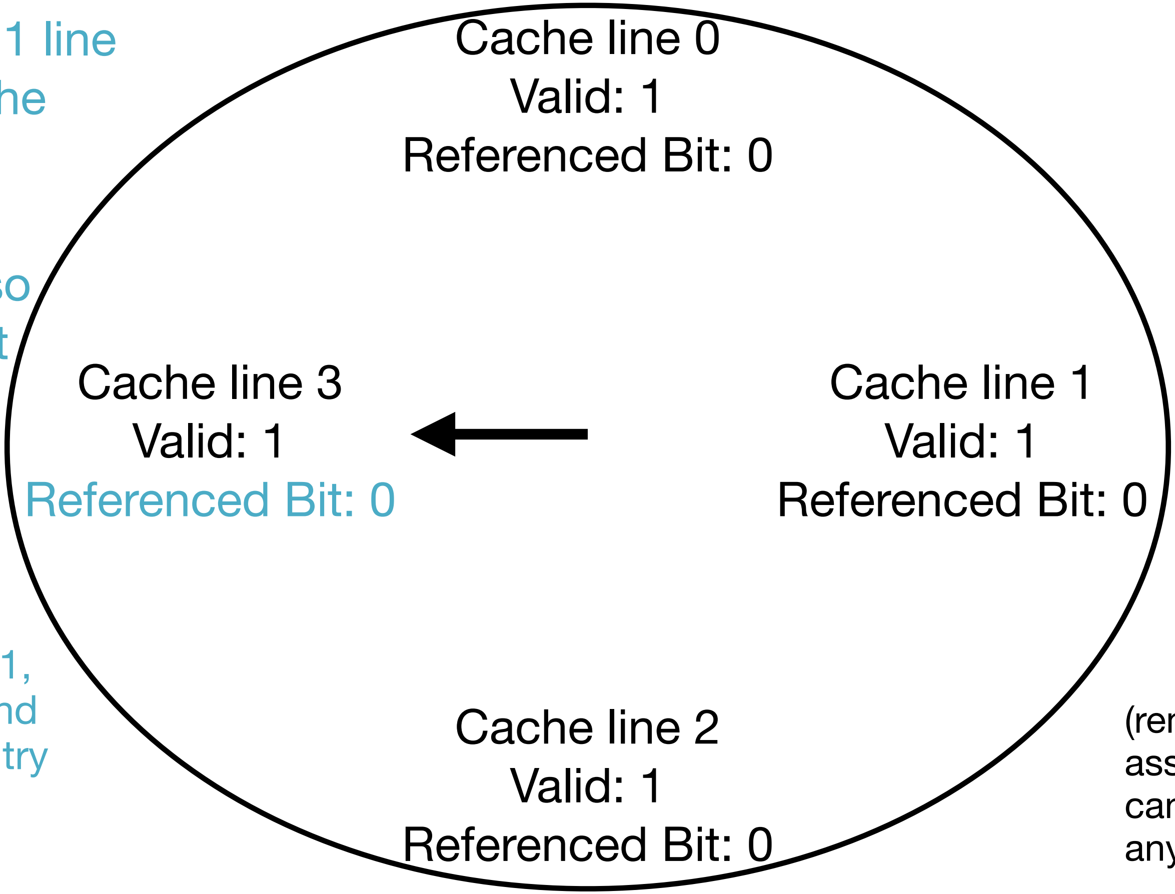
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one

Now we're at line 3. The reference bit is 1, so we'll set it to 0 and move to the next entry



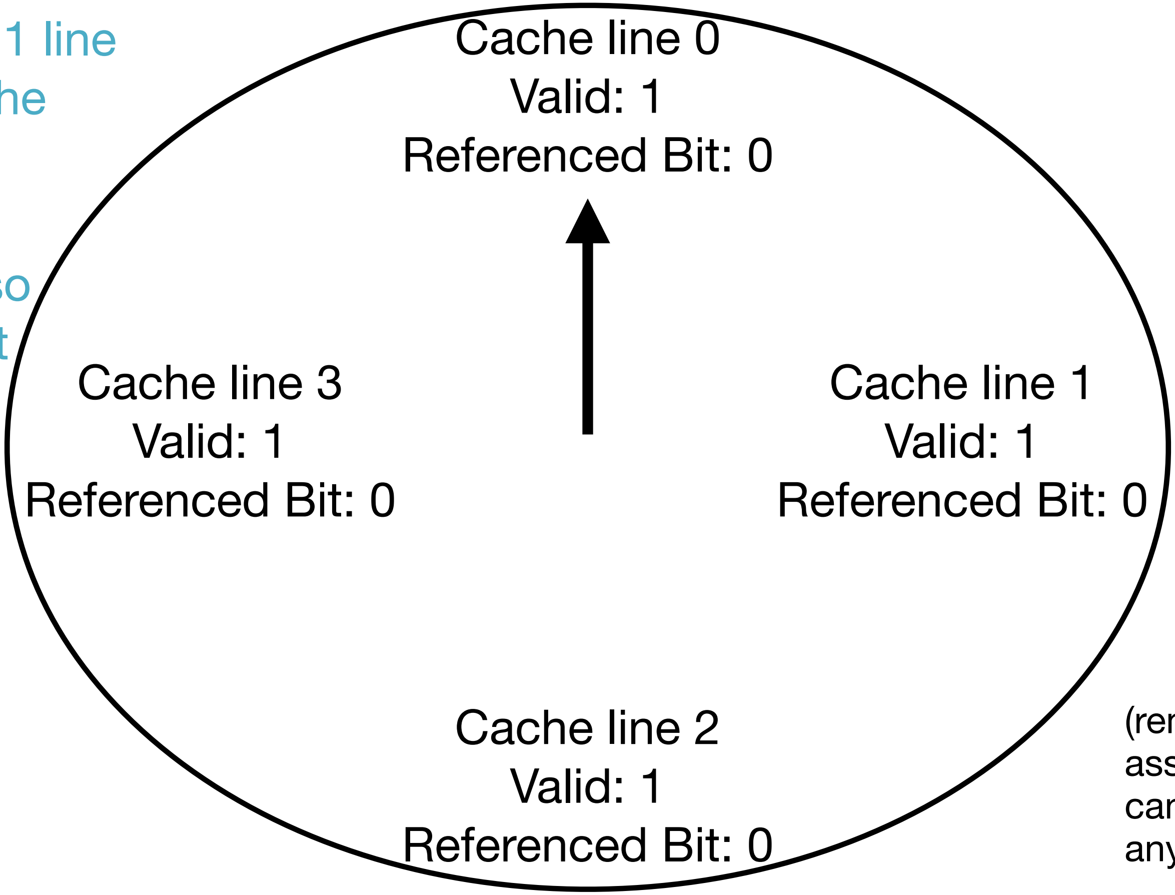
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one



(remember its fully associative, so we can store the data anywhere)

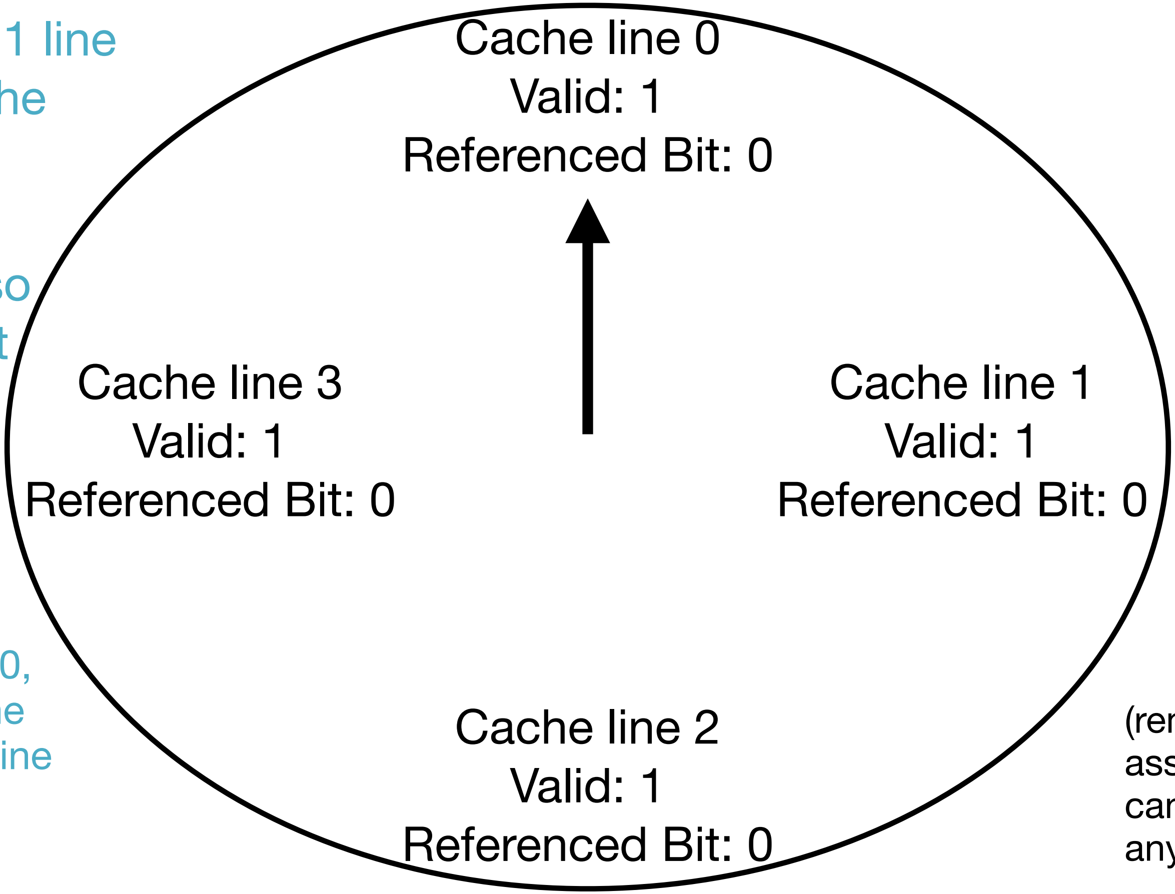
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one

Now we're at line 0. The reference bit is 0, so we'll evict this line to bring in the new line



(remember its fully associative, so we can store the data anywhere)

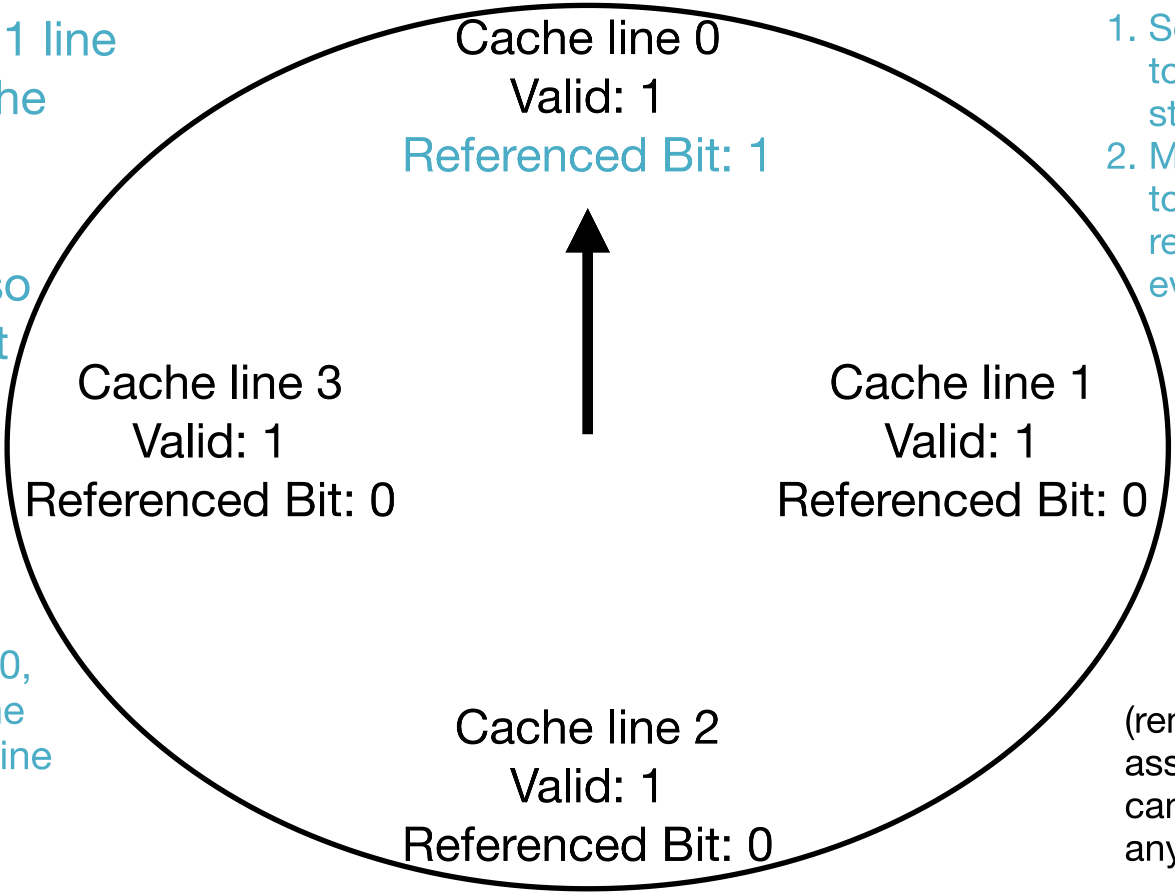
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 5: Bring 1 line into the cache

All lines are full, so we need to evict one

Now we're at line 1. The reference bit is 0, so we'll evict this line to bring in the new line



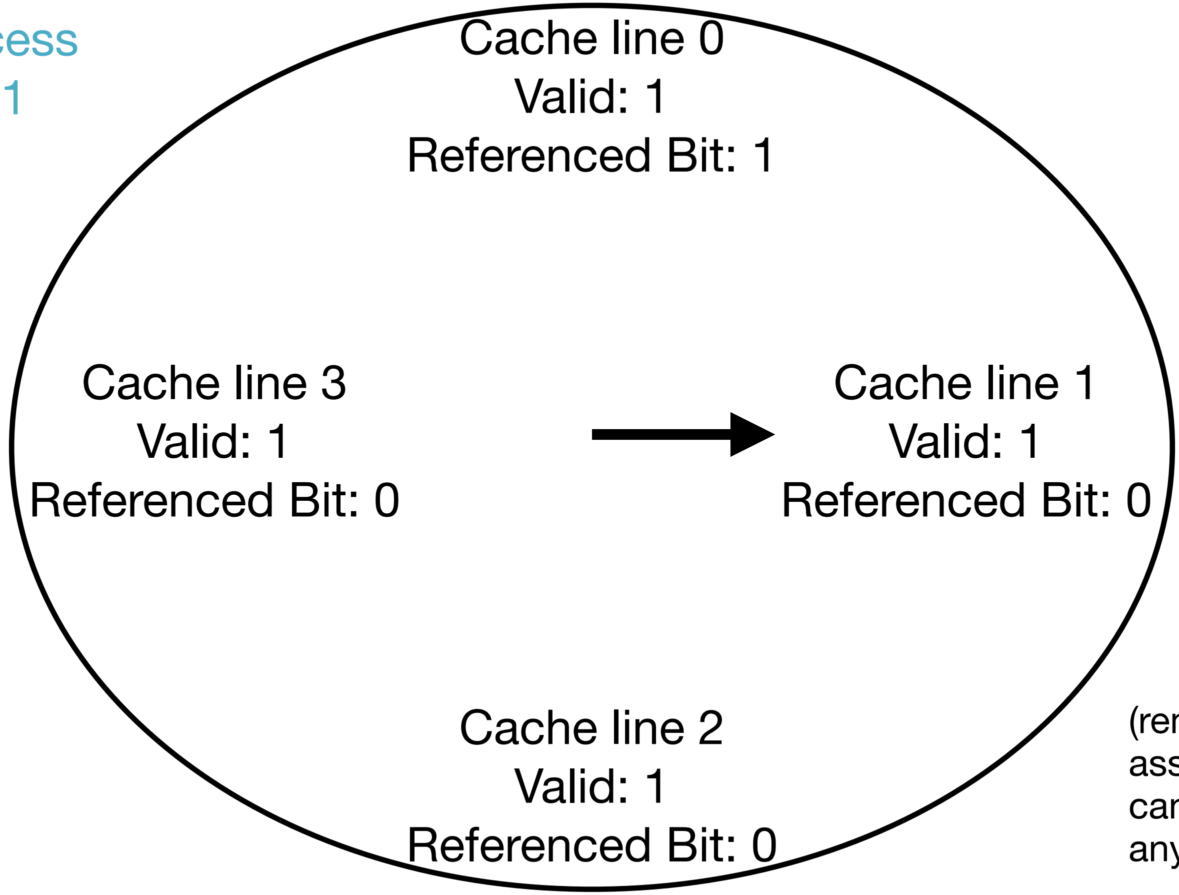
1. Set the referenced bit to one since we just stored a new line here
2. Move the clock hand to the next entry so it's ready for the next eviction

(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 6: Access cache line 1

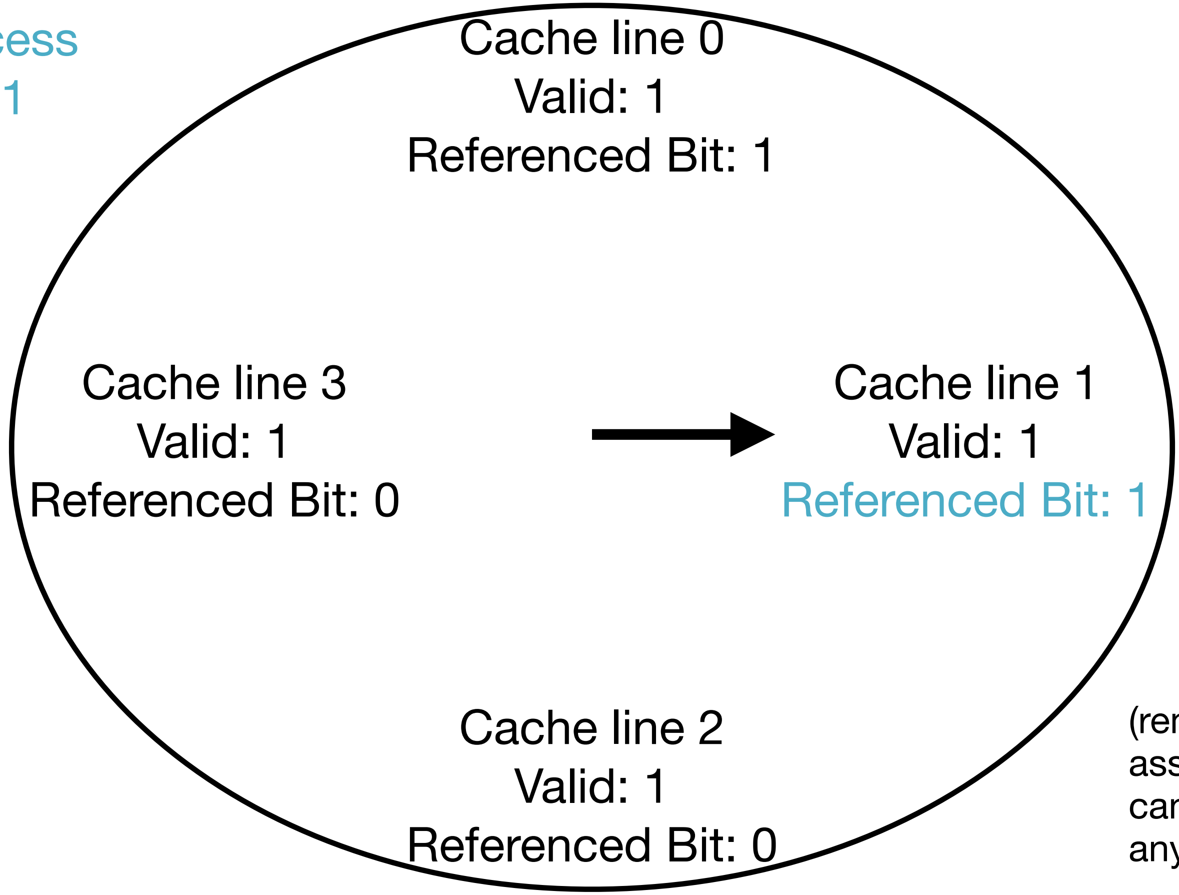


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 6: Access cache line 1

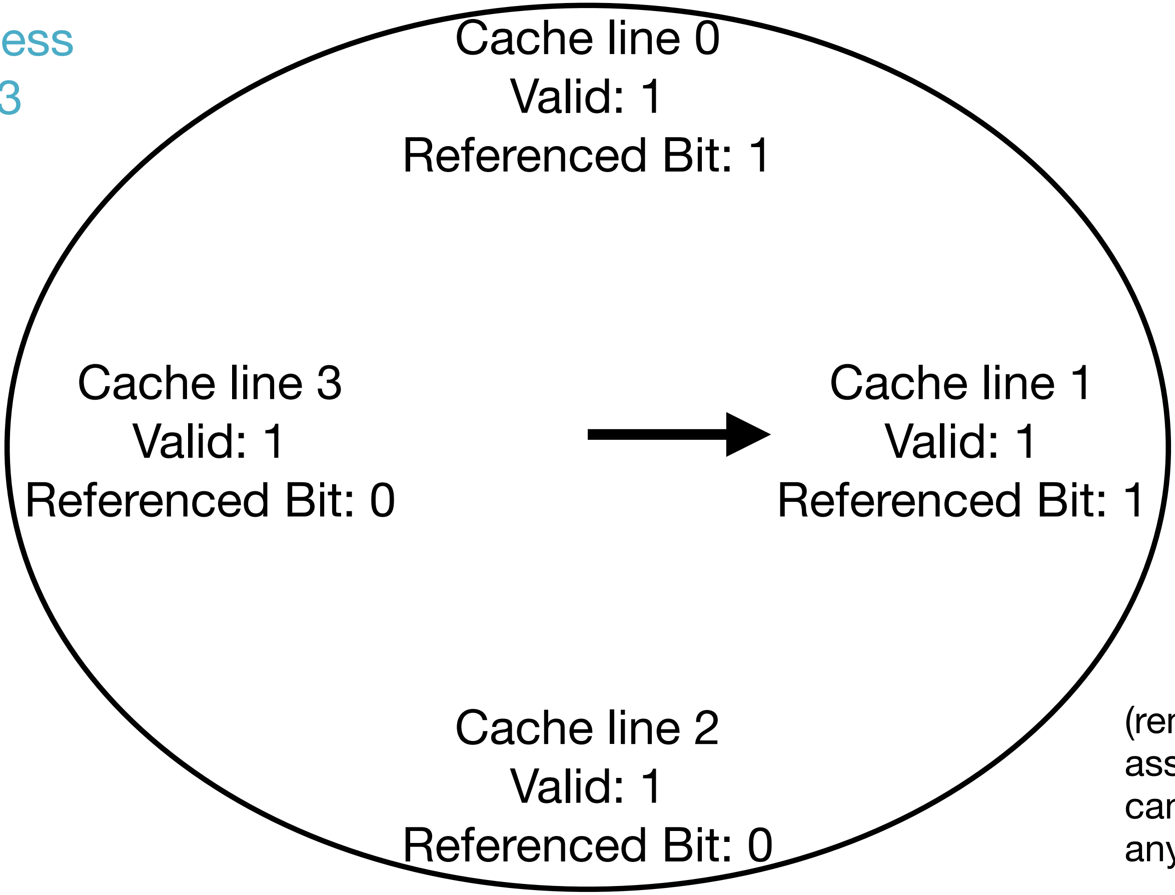


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 7: Access cache line 3

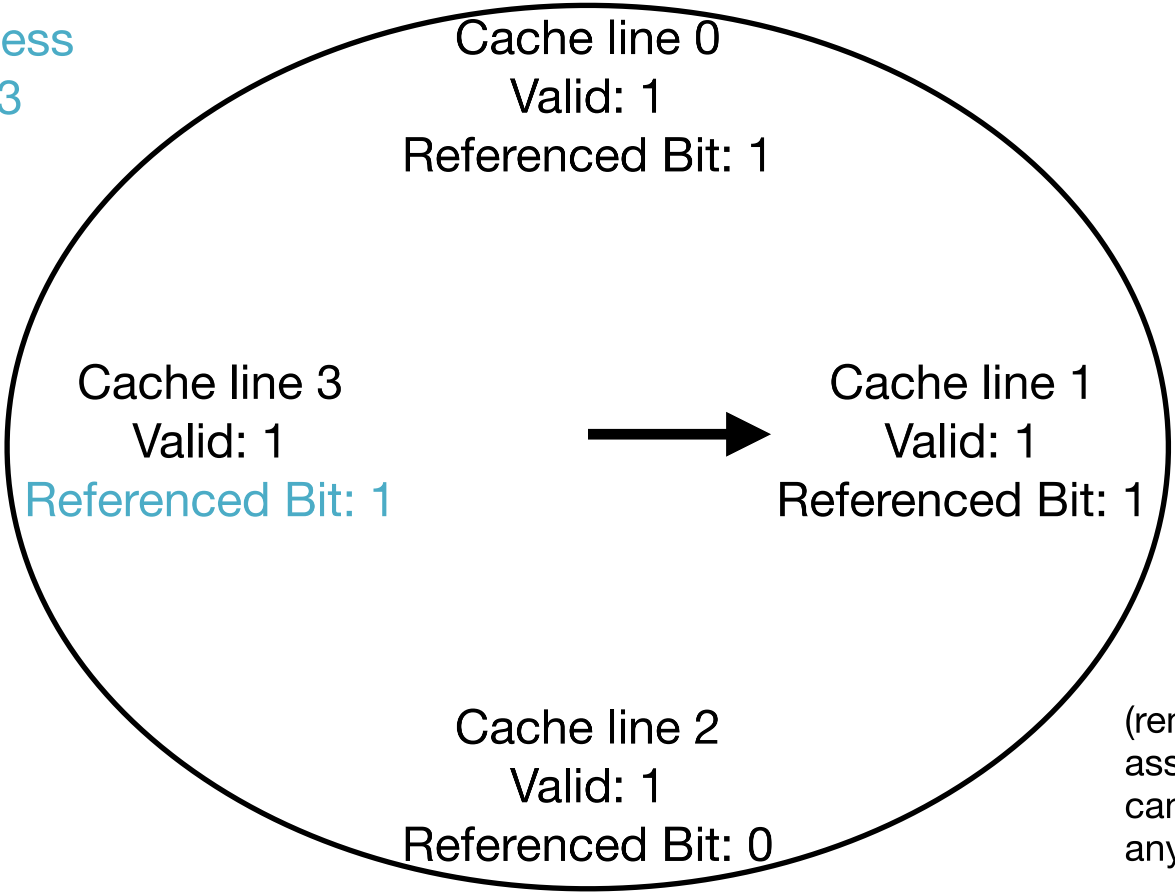


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 7: Access cache line 3

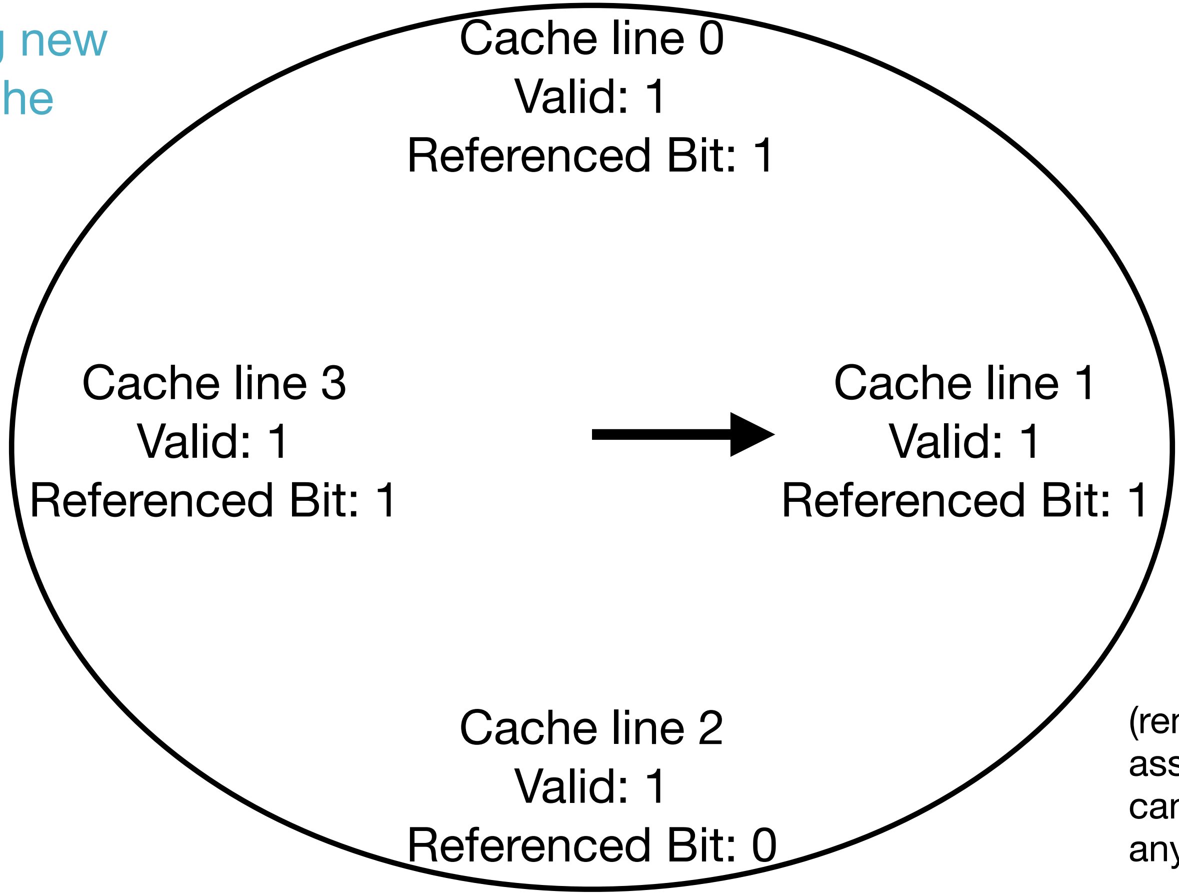


(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 8: Bring new line into cache



(remember its fully associative, so we can store the data anywhere)

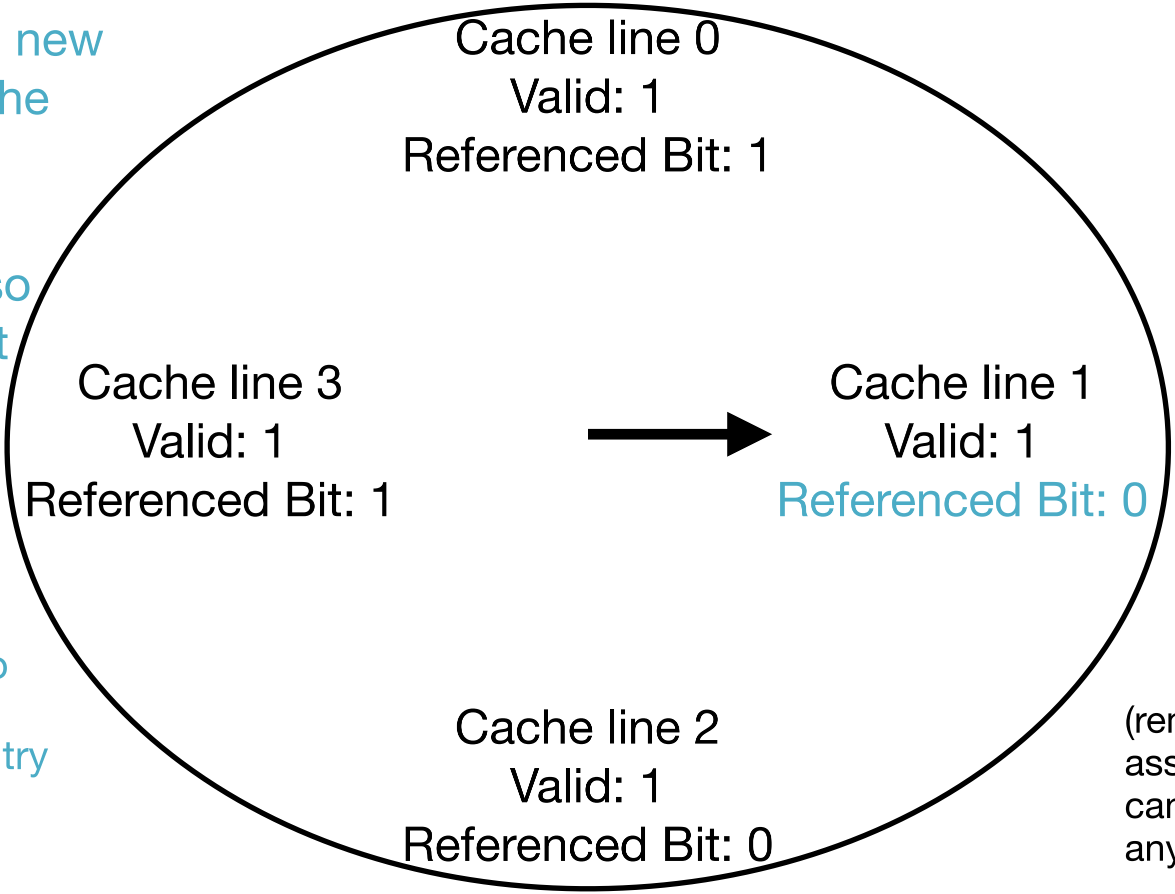
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 8: Bring new line into cache

All lines are full, so we need to evict one

We're at line 1. The reference bit is 1, so we'll set it to 0 and move to the next entry



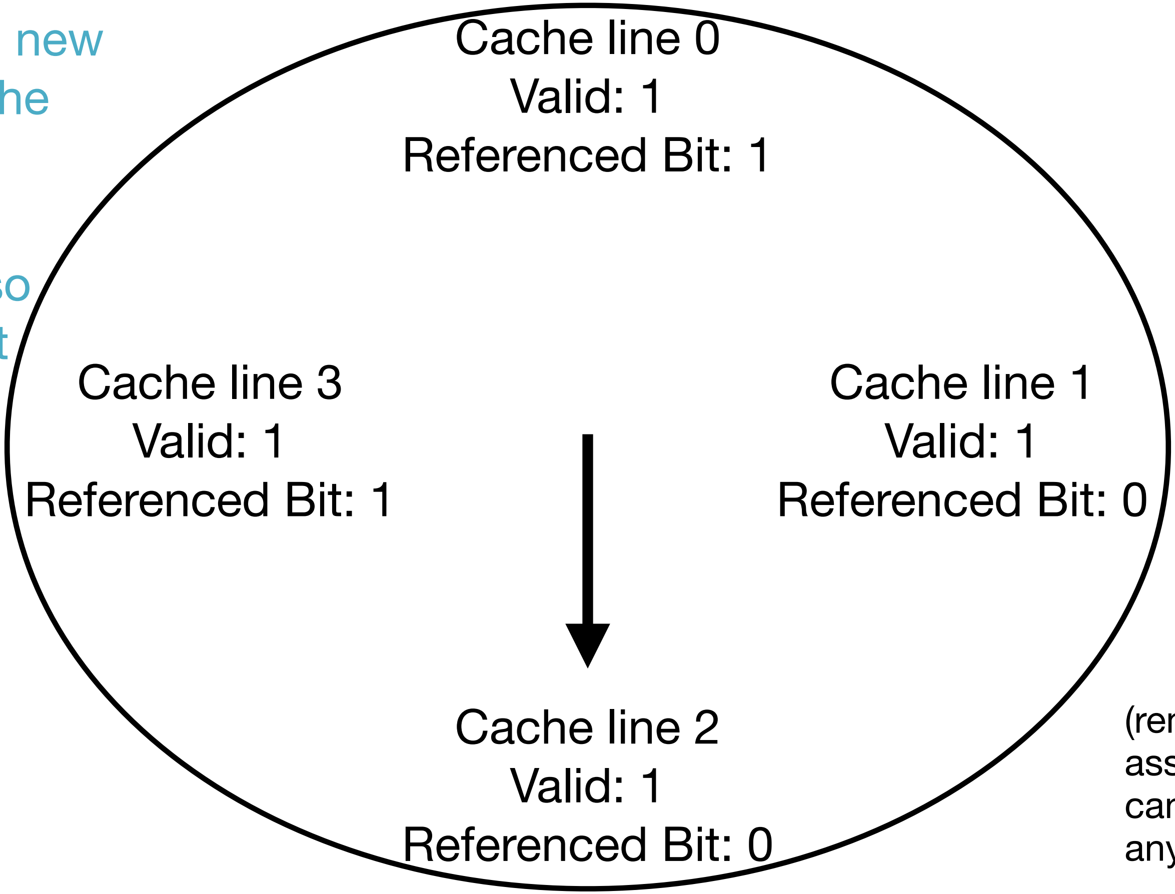
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 8: Bring new line into cache

All lines are full, so we need to evict one



(remember its fully associative, so we can store the data anywhere)

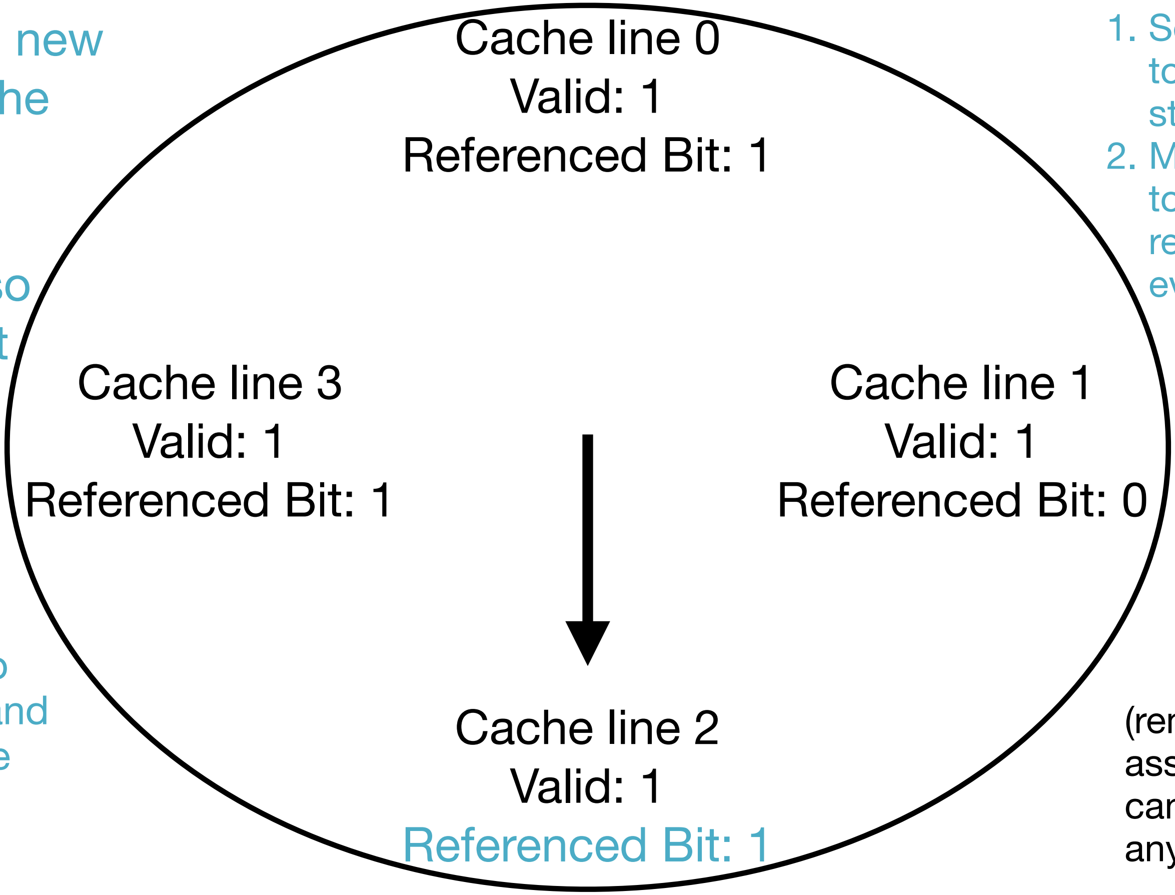
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 8: Bring new line into cache

All lines are full, so we need to evict one

We're at line 2. The reference bit is 0, so we'll evict this line and bring in the new one



1. Set the referenced bit to one since we just stored a new line here
2. Move the clock hand to the next entry so it's ready for the next eviction

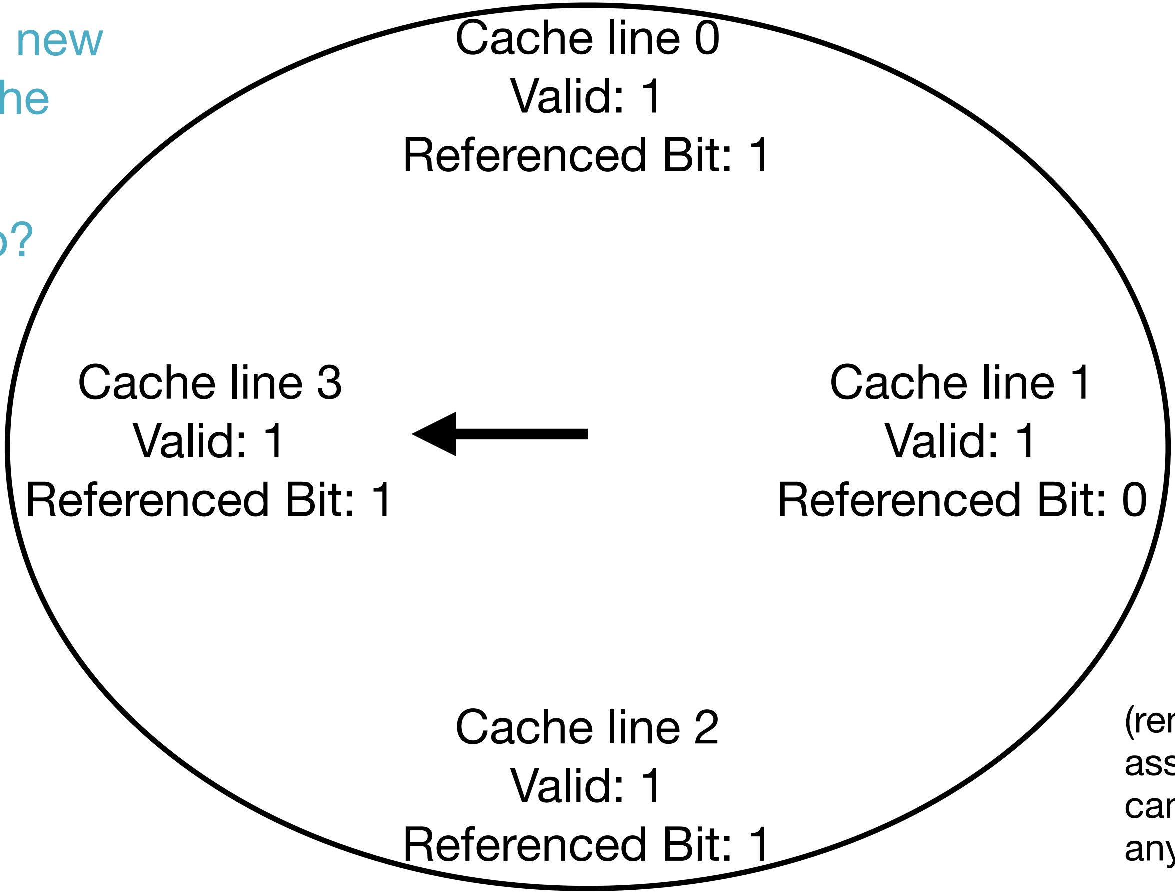
(remember its fully associative, so we can store the data anywhere)

Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 9: Bring new line into cache

Where will it go?



(remember its fully associative, so we can store the data anywhere)

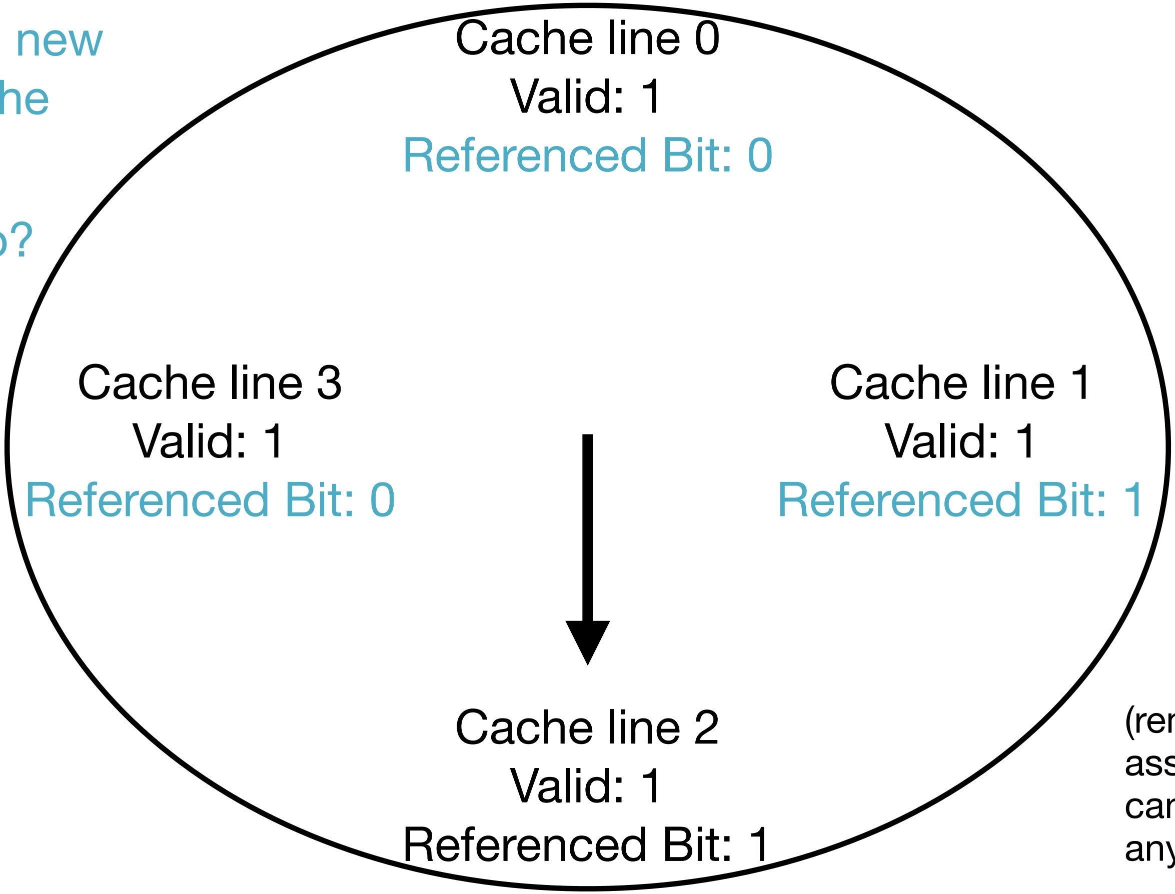
Approximate LRU (Clock Algorithm)

4-line Fully Associative cache

Action 9: Bring new line into cache

Where will it go?

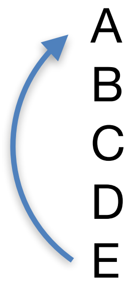
Cache line 1!



(remember its fully associative, so we can store the data anywhere)

More Eviction Policies

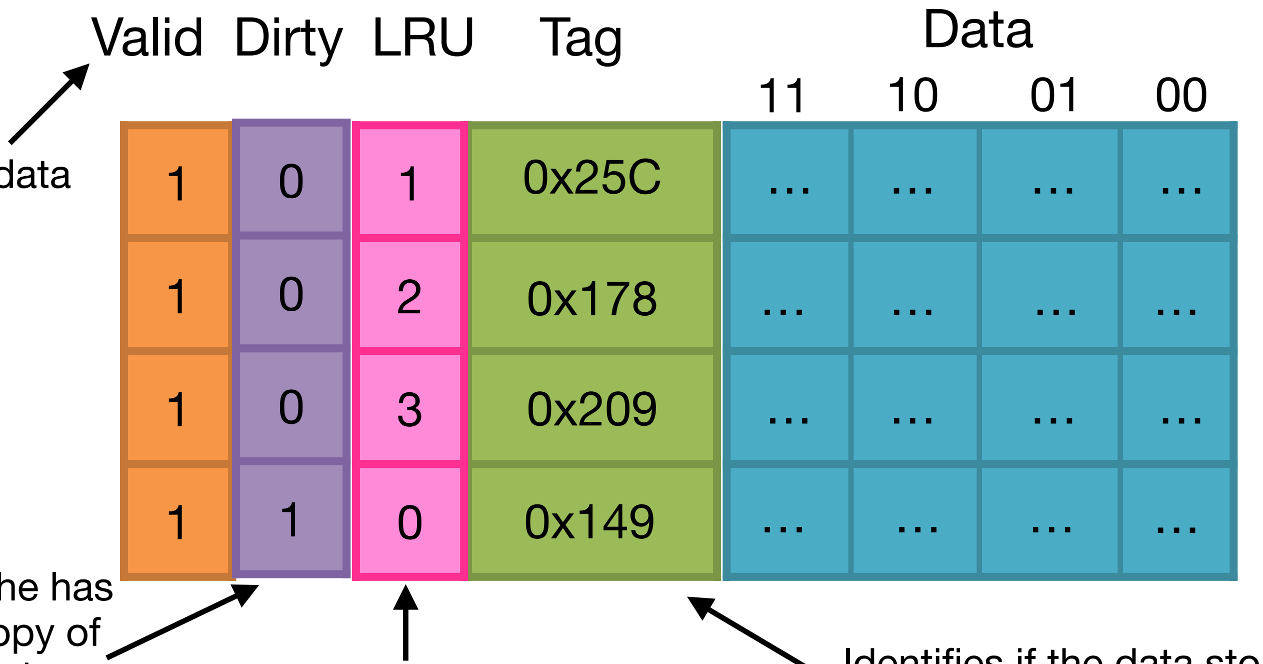
- Most recently used (MRU)
  - Evict the line that was most recently used
- Random
  - Chooses a random line to evict
  - Benefit: don’t have to keep track of additional metadata
- Example where these are better than LRU:
  - 4 cache lines, iterating through an array where you access elements A, B, C, D, and E, where each element maps to a different line in the cache
  - LRU would have a 0% hit rate
  - MRU would have a 75 % hit rate
  - Random would have a non-zero hit rate



Fully Associative Cache

Fully Associative Cache (write-back)

- The data can be stored anywhere



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	…	…	…	…

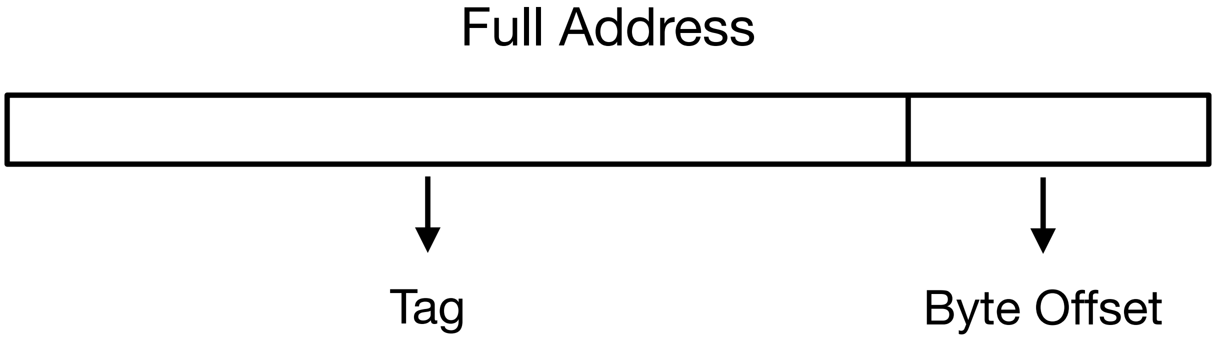
Identifies if the data is valid

Identifies if the cache has a more updated copy of the data than main memory

Identifies which line should be kicked out if we run out of room

Identifies if the data stored at this line corresponds to the data that we are looking for

Address Breakdown



# byte offset bits = $\log_2$(line size)

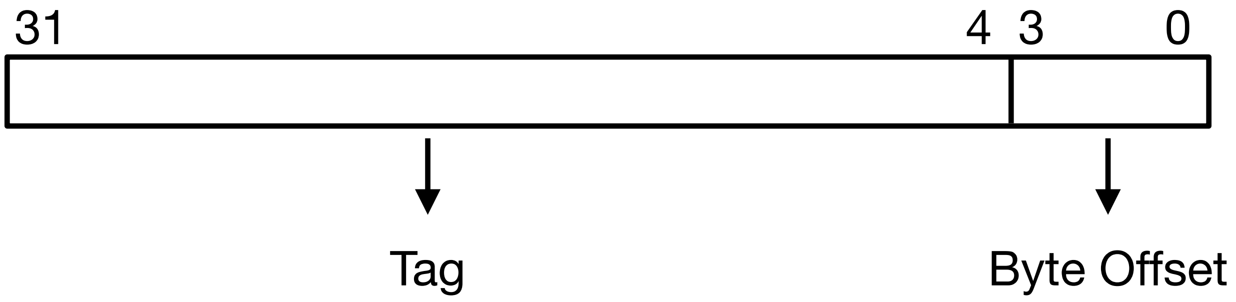
# tag bits = # address bits - # offset bits

Example of Address Breakdown

- Ex1: 64 KB fully associative cache with 16B blocks, 32-bit address space

# byte offset bits = $\log_2$(line size) = $\log_2$(16) = 4

# tag bits = # address bits - # offset bits = 32 - 4 = 28



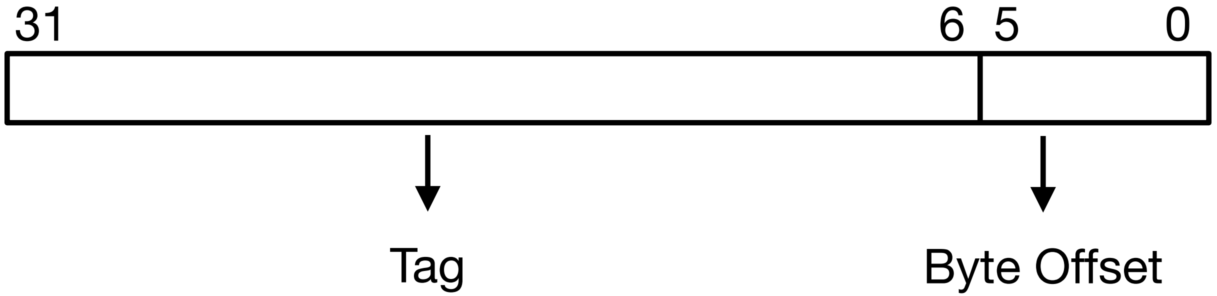
Example of Address Breakdown

- Ex2: 64 KB fully associative cache with 1K lines, 32-bit address space

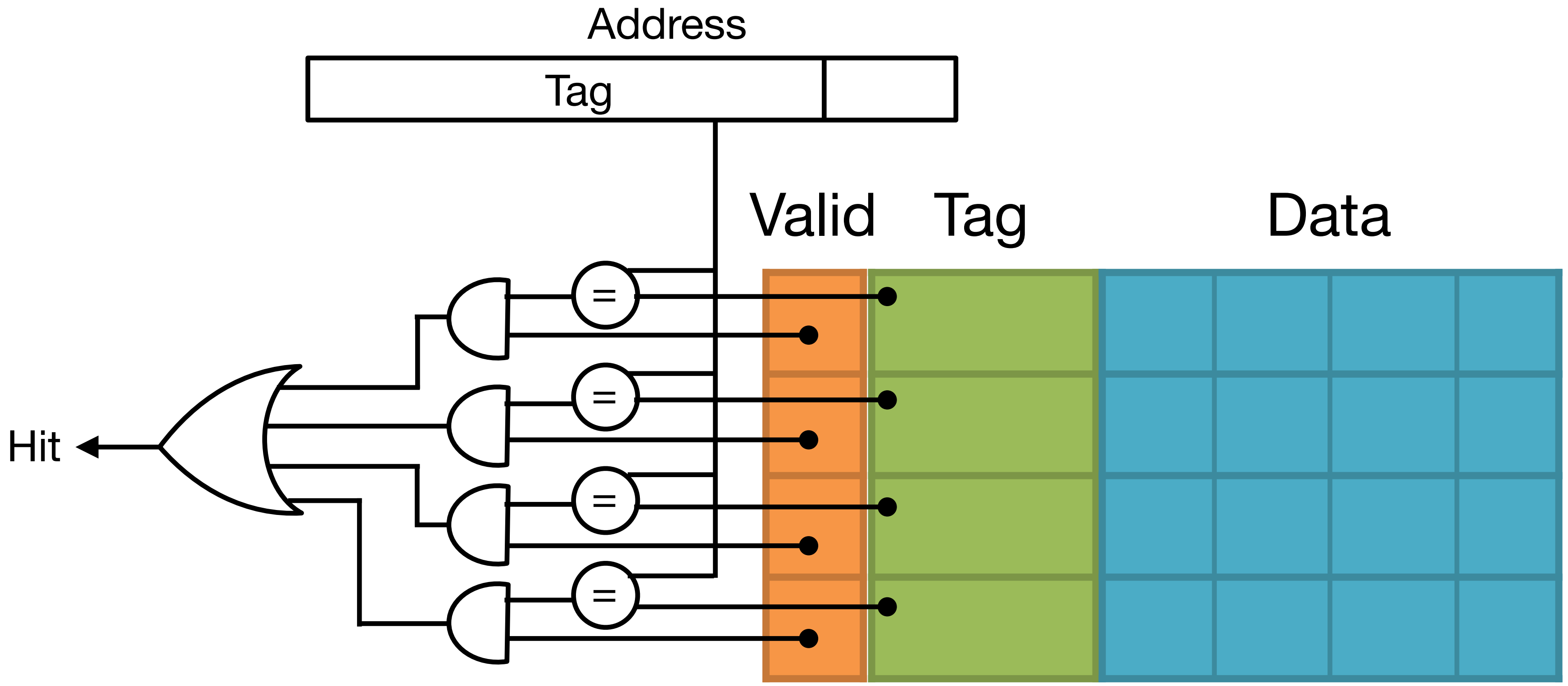
line size = cache size / num lines = 64 KB / 1K = 64B

# byte offset bits = $\log_2$(line size) = $\log_2$(64) = 6

# tag bits = # address bits - # offset bits = 32 - 6 = 26

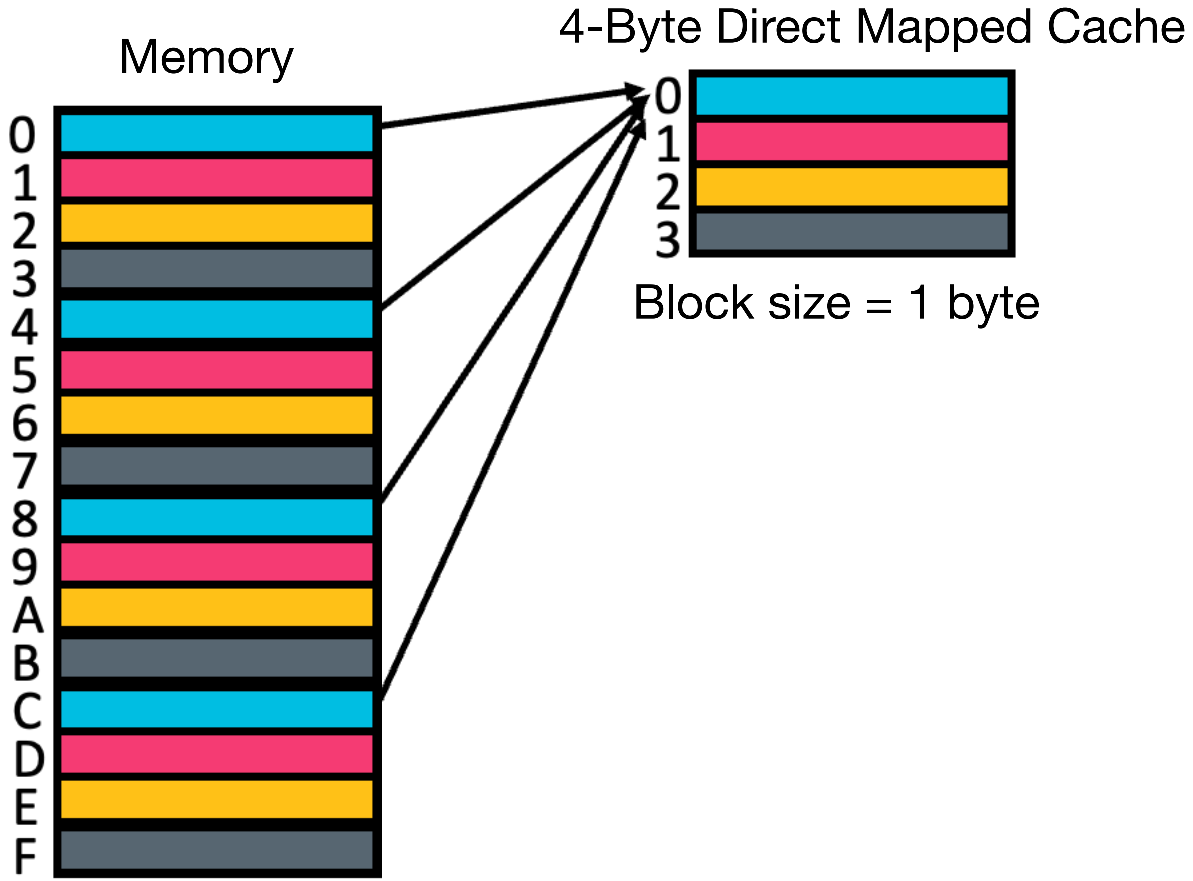


Hardware Required to Check for Hit

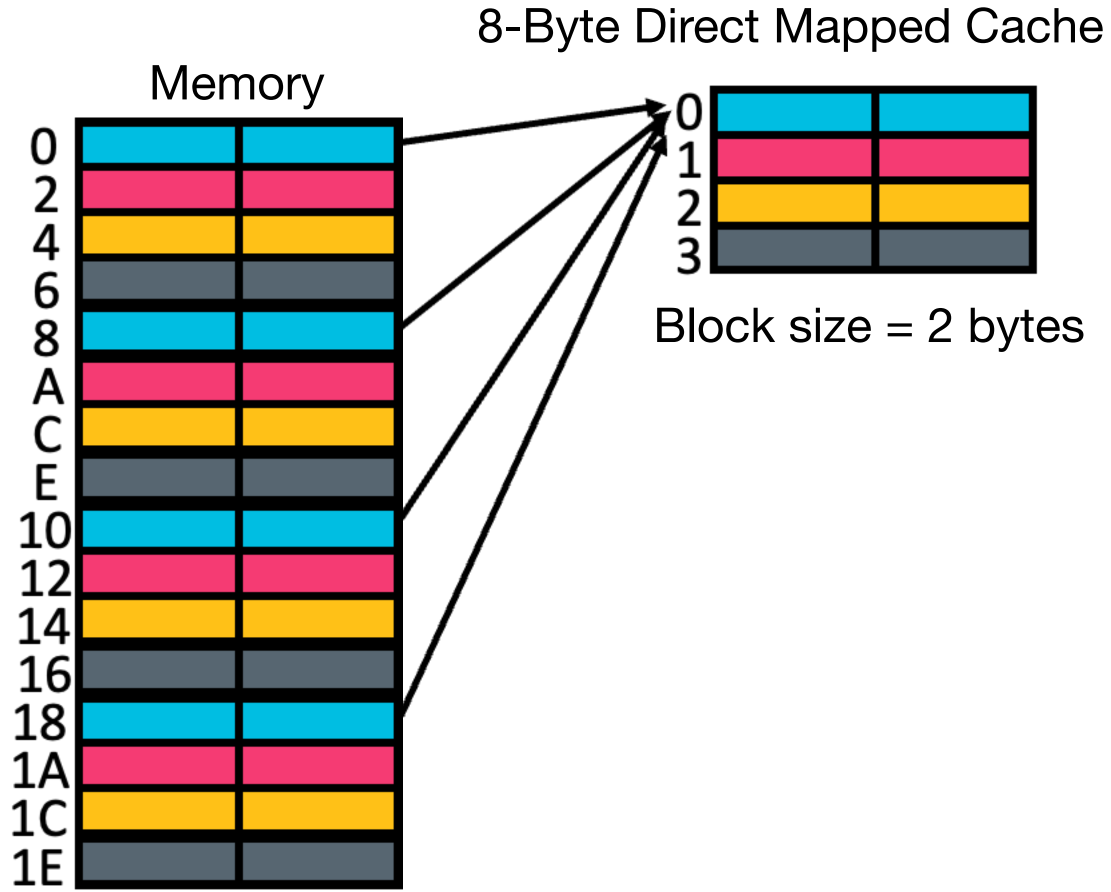


Direct Mapped Caches

Direct Mapped Cache



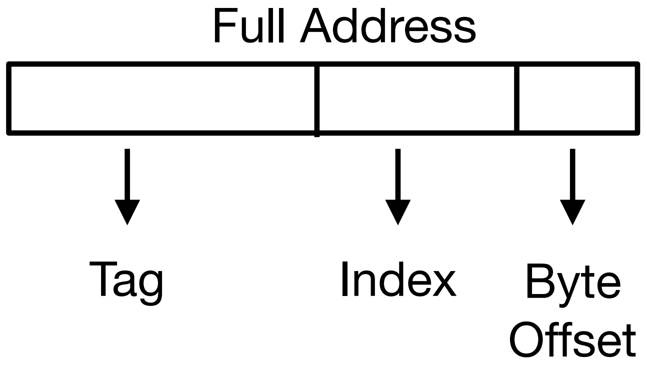
Direct Mapped Cache



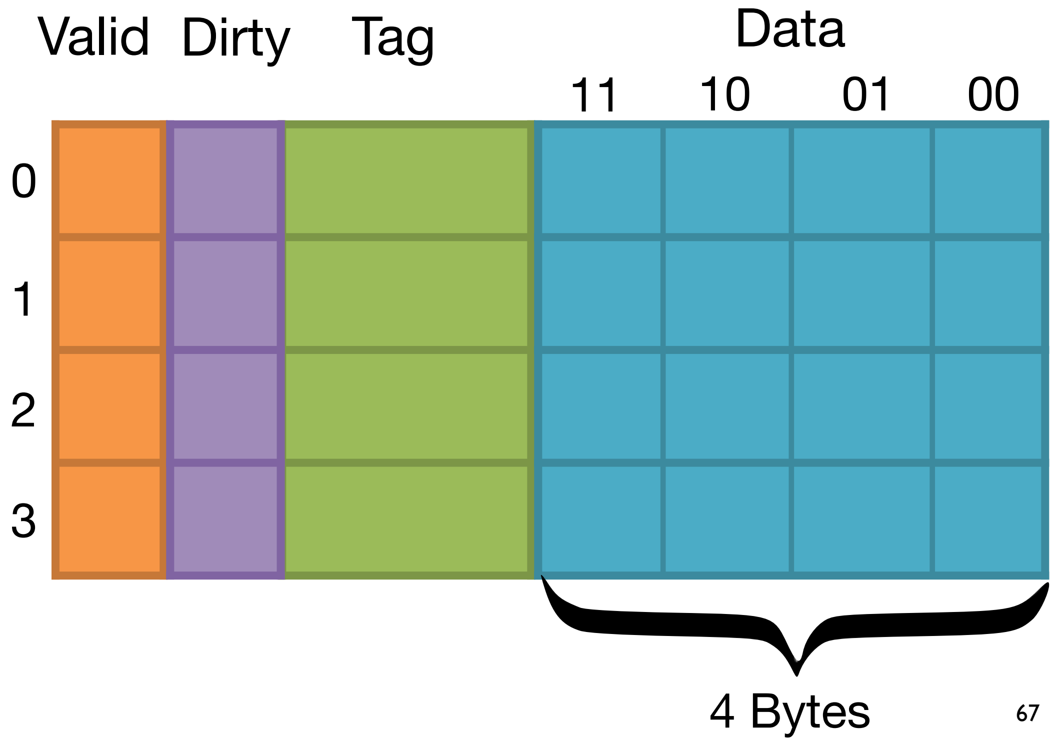
Note that this memory and cache are twice as large as the previous slide

Direct Mapped (write-back)

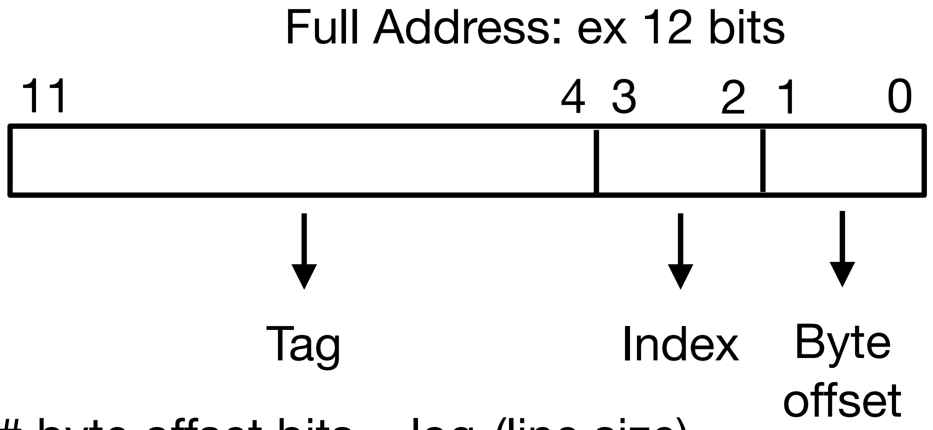
- The data can only be stored in one location



TIO



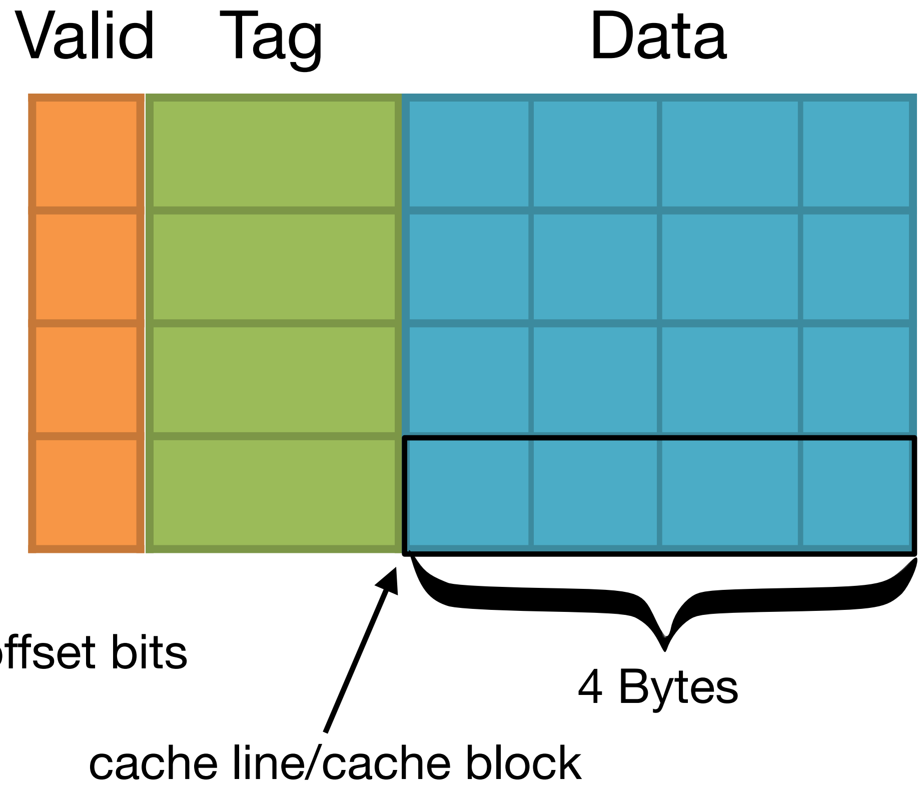
Direct Mapped Cache Address Breakdown



# byte offset bits = $\log_2$(line size) = $\log_2(4) = 2$

# index bits = $\log_2$(# lines) = $\log_2(4) = 2$

# tag bits = # address bits - # index bits - # offset bits = 12 - 2 - 2 = 8



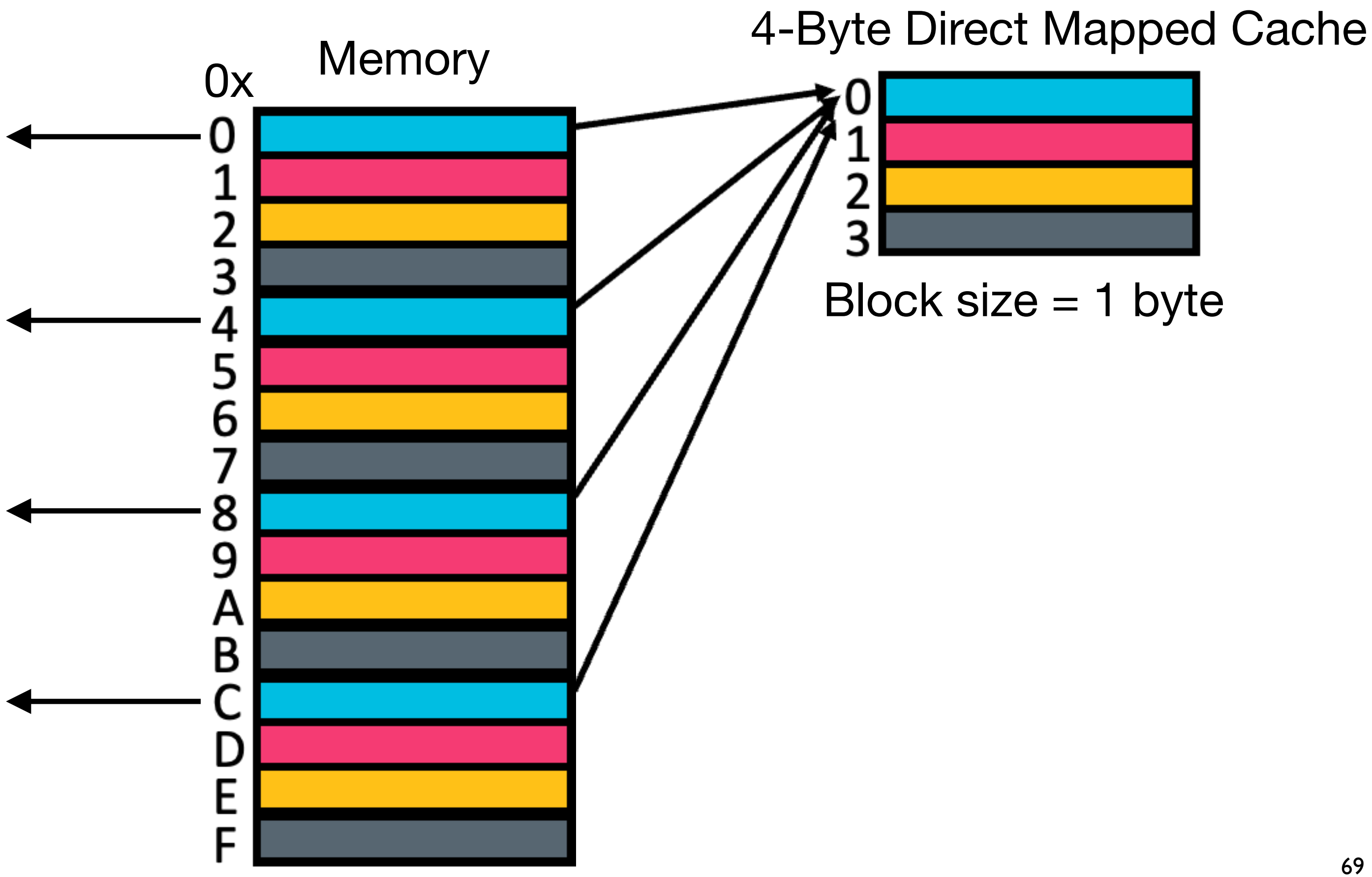
Direct Mapped Cache

Address: 0b 0000
Offset = None
Index = 0b 00
Tag = 0b 00

Address: 0b 0100
Offset = None
Index = 0b 00
Tag = 0b 01

Address: 0b 1000
Offset = None
Index = 0b 00
Tag = 0b 10

Address: 0b 1100
Offset = None
Index = 0b 00
Tag = 0b 11



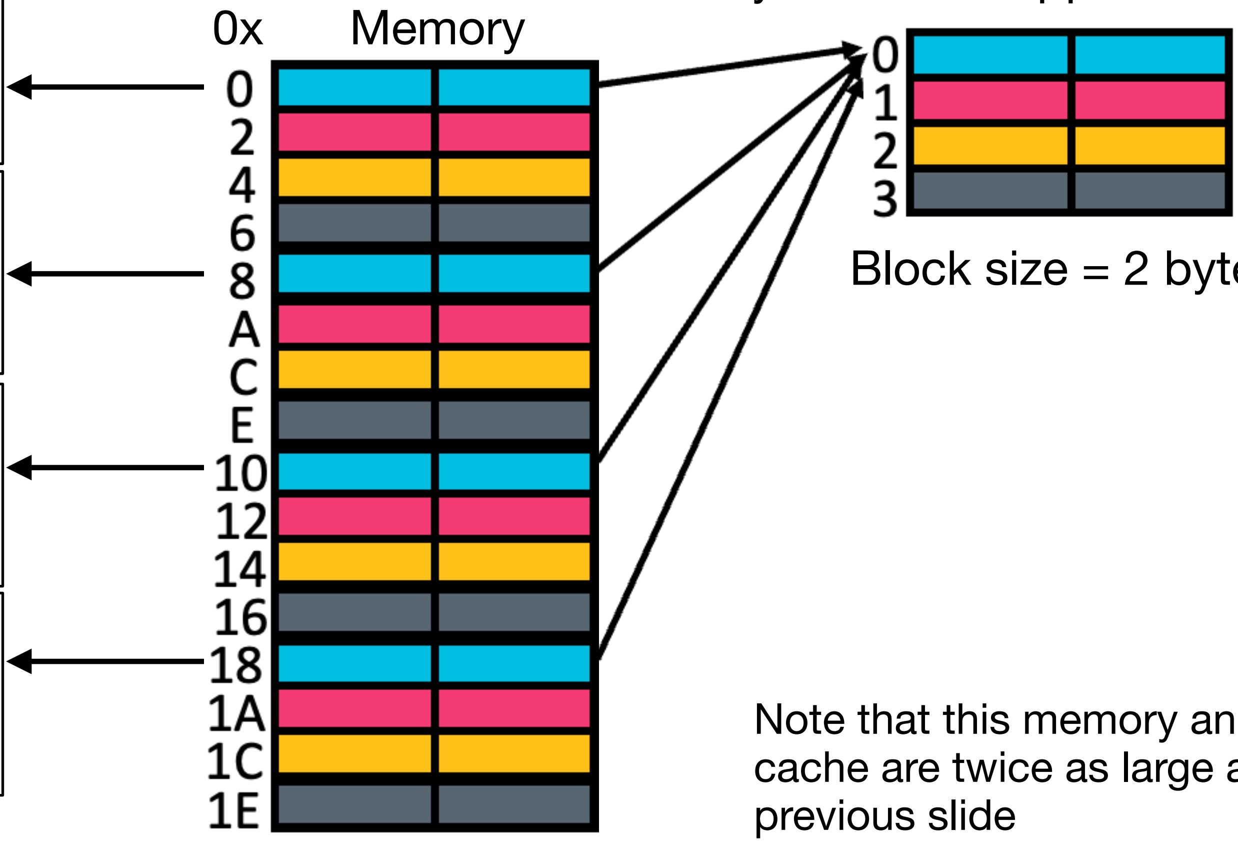
Direct Mapped Cache

Address: 0b 0 0000
Offset = 0b 0
Index = 0b 00
Tag = 0b 00

Address: 0b 0 1000
Offset = 0b 0
Index = 0b 00
Tag = 0b 01

Address: 0b 1 0000
Offset = 0b 0
Index = 0b 00
Tag = 0b 10

Address: 0b 1 1000
Offset = 0
Index = 0b 00
Tag = 0b 11



8-Byte Direct Mapped Cache

Block size = 2 bytes

Note that this memory and cache are twice as large as the previous slide

Direct Mapped Cache Address Breakdown

# byte offset bits = $\log_2$(line size)

# index bits = $\log_2$(# lines)

# tag bits = # address bits - # index bits - # offset bits

Ex: Direct Mapped Cache with 32 bit address, 4B blocks and 4KB data

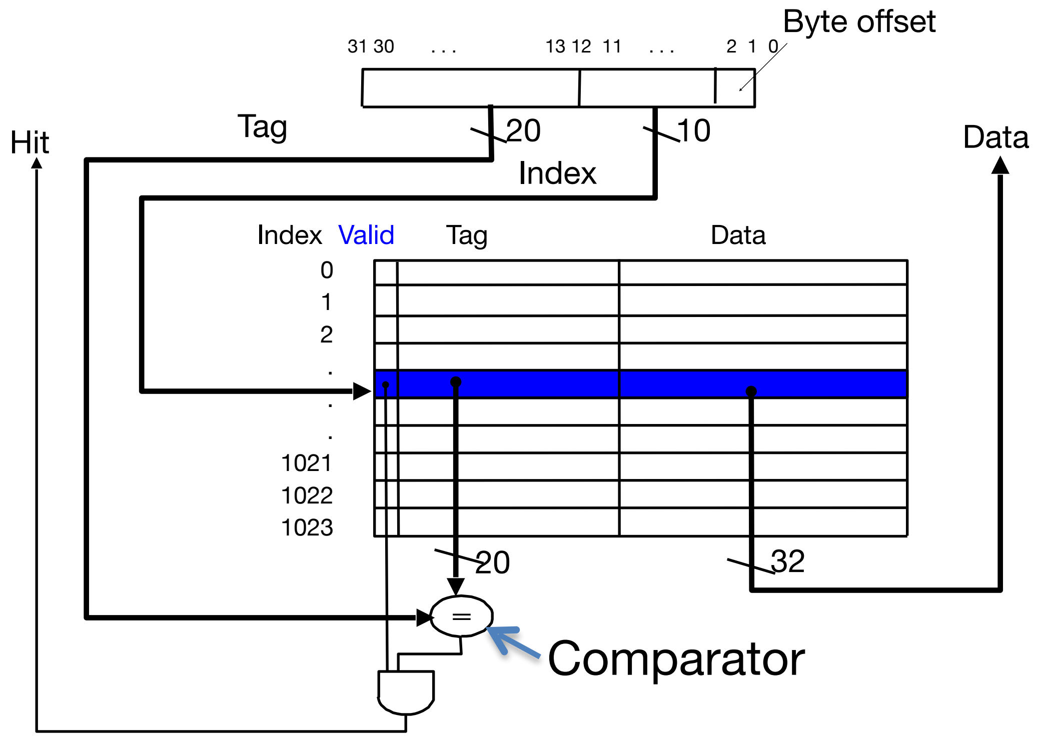
# byte offset bits = $\log_2$(line size) = $\log_2(4) = 2$

# lines = cache size / line size = 4KB / 4B = 1K

# index bits = $\log_2$(# lines) = $\log_2$(1K) = 10

# tag bits = # address bits - # index bits - # offset bits = 32 - 10 - 2 = 20

Direct-Mapped Cache Hardware: 4B blocks, 4KB data



Direct Mapped Cache Address Breakdown

# byte offset bits = $\log_2$(line size)

# index bits = $\log_2$(# lines)

# tag bits = # address bits - # index bits - # offset bits

Ex: Direct Mapped Cache with 32 bit address, 16B blocks and 4KB data

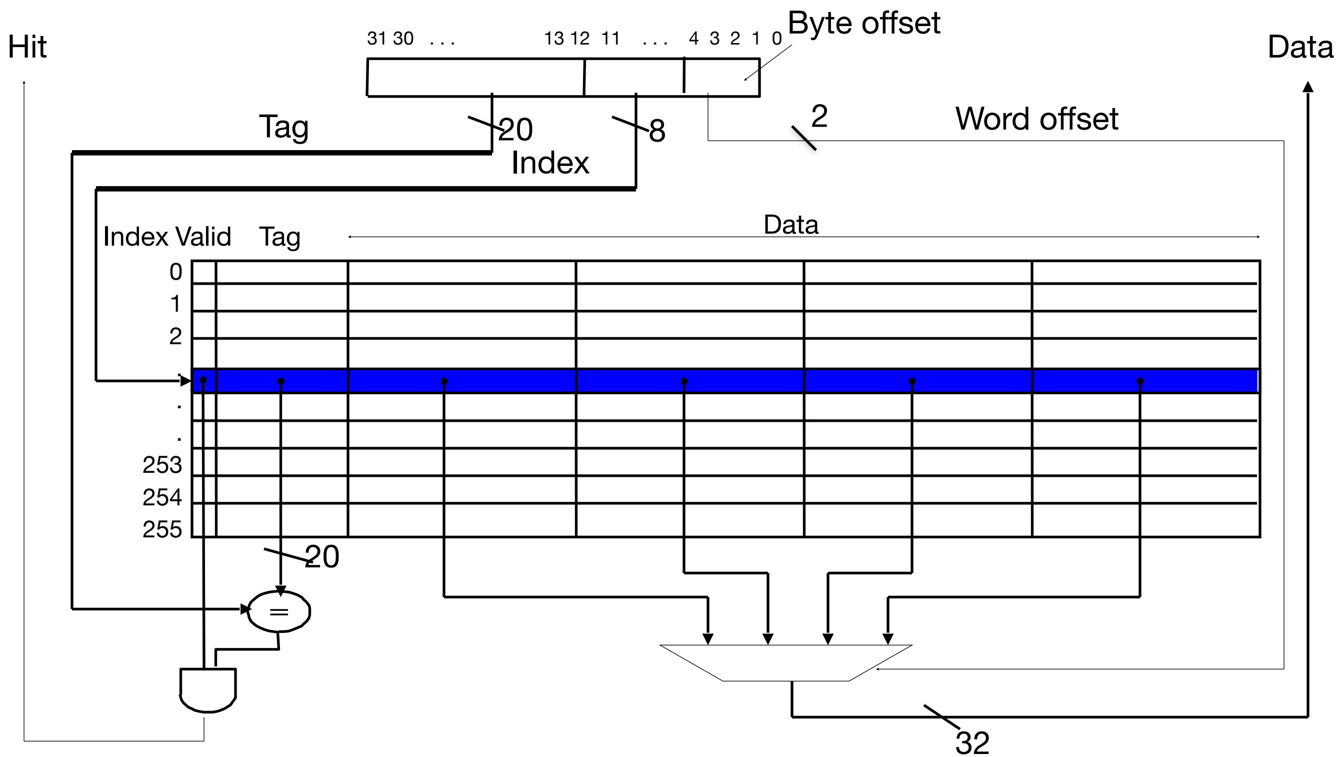
# byte offset bits = $\log_2$(line size) = $\log_2$(16) = 4

# lines = cache size / line size = 4KB / 16B = 256

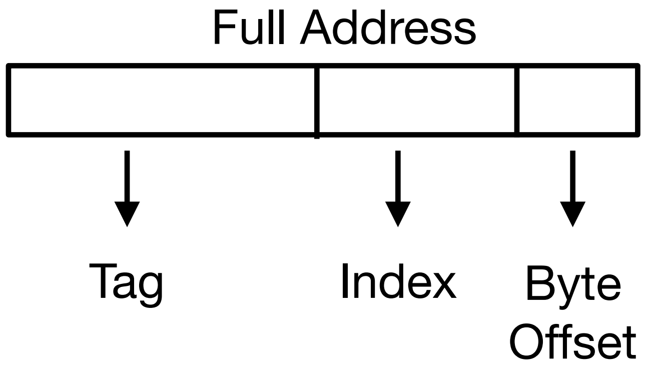
# index bits = $\log_2$(# lines) = $\log_2$(256) = 8

# tag bits = # address bits - # index bits - # offset bits = 32 - 8 - 4 = 20

Multiword-Block Direct-Mapped Cache: 16B block size, 4 kB data

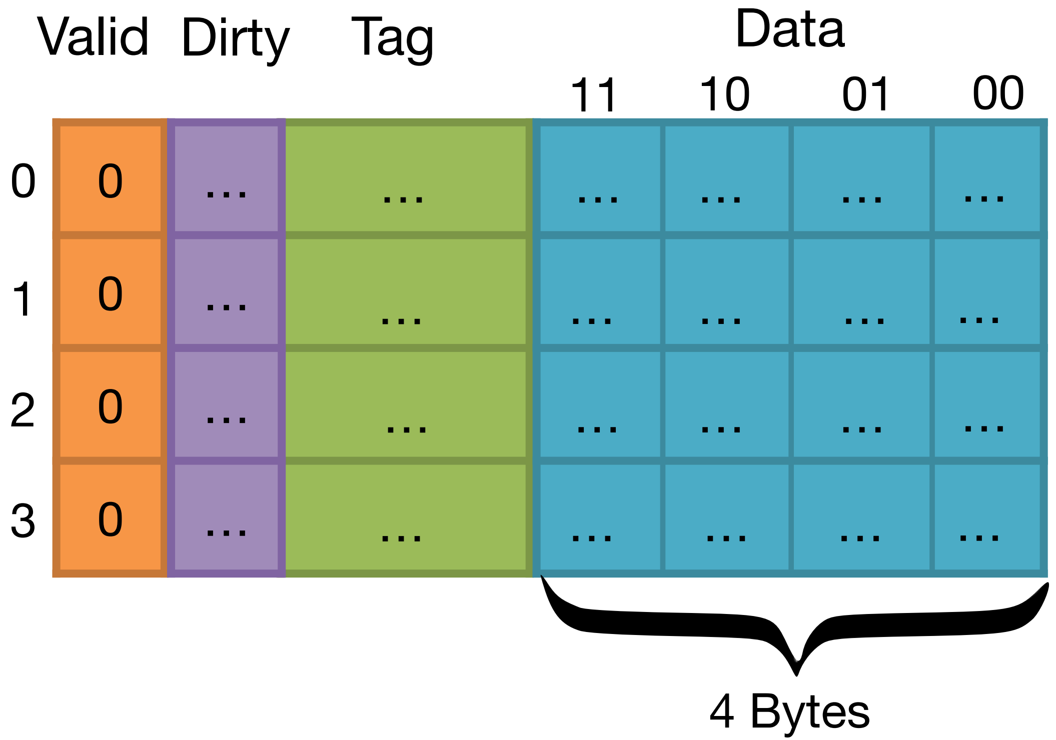


Direct Mapped Cache (write-back)

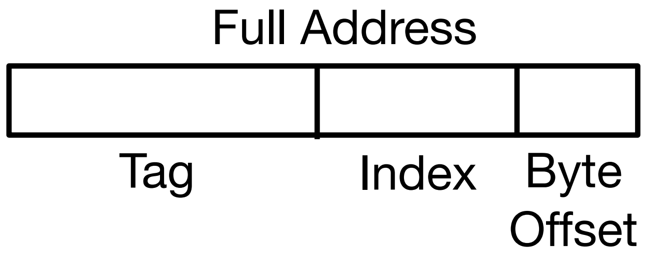


Read byte 0xFE2

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



Direct Mapped Cache (write-back)



Read byte 0xFE2

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

0b 1111 1110 0010

T = 0xFE

I = 0x0

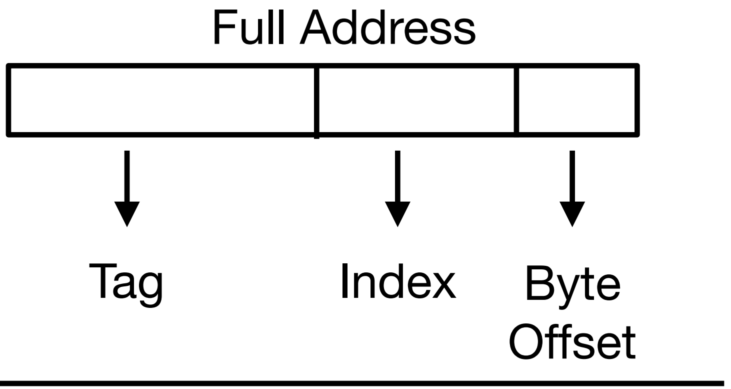
O = 0x2

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

4 Bytes

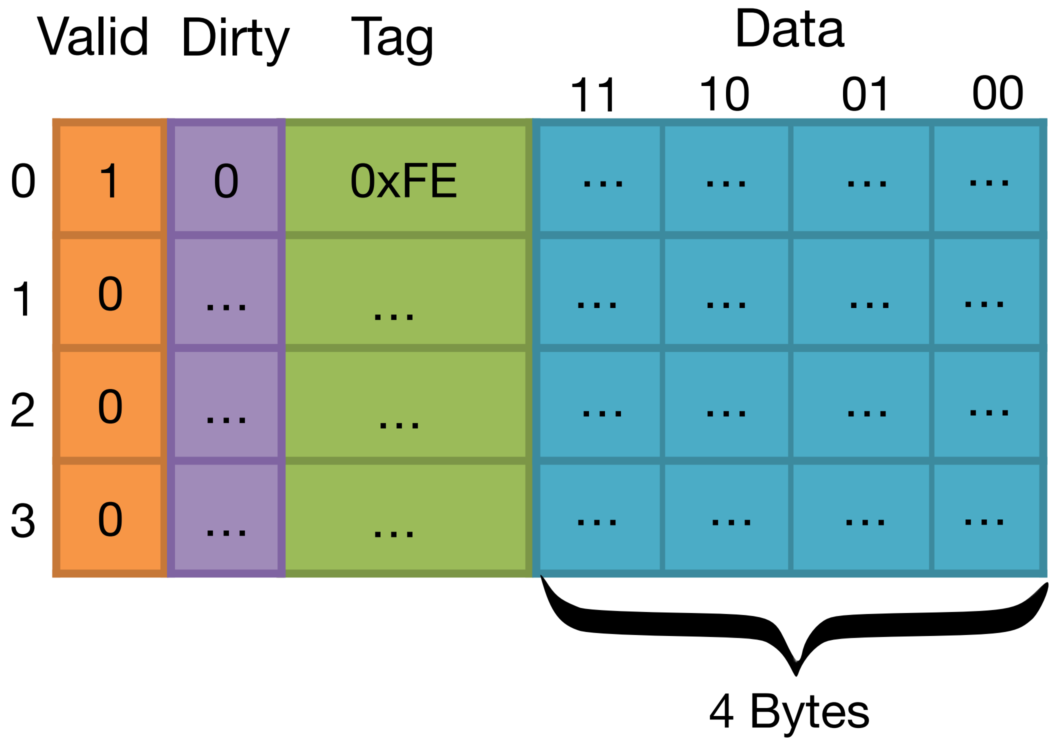
Cache Miss => bring in entire line

Direct Mapped Cache (write-back)



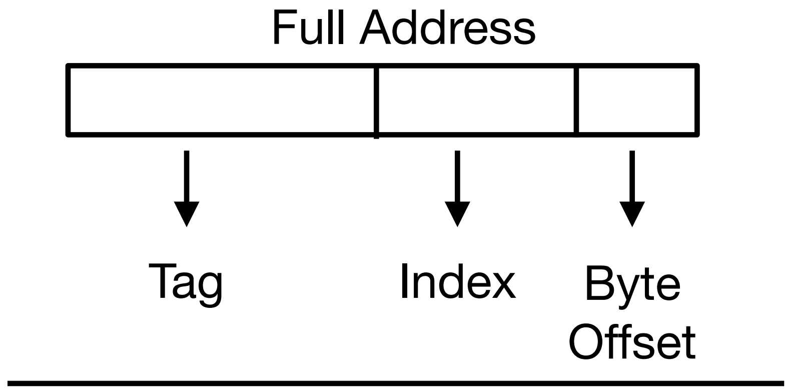
Read byte 0xFE8

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



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

Direct Mapped Cache (write-back)



Read byte 0xFE8

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

0b 1111 1110 1000

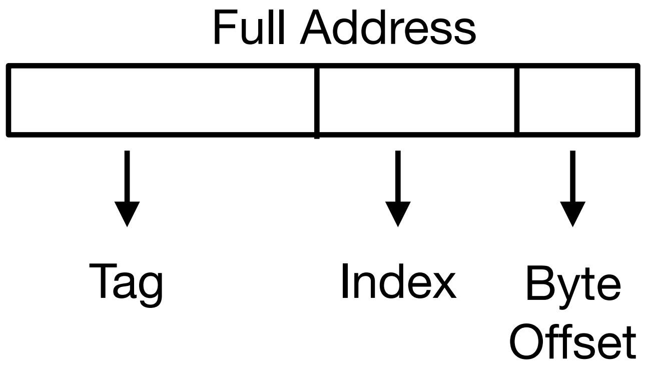
T = 0xFE
I = 0x2
O = 0x0

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

4 Bytes

Cache Miss => bring in entire line

Direct Mapped Cache (write-back)



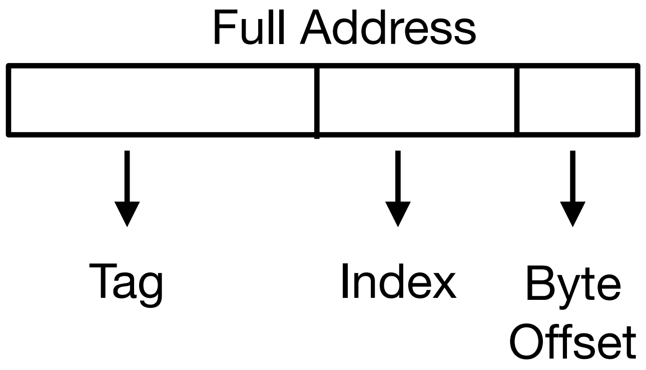
Read byte 0xFE9

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

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

4 Bytes

Direct Mapped Cache (write-back)



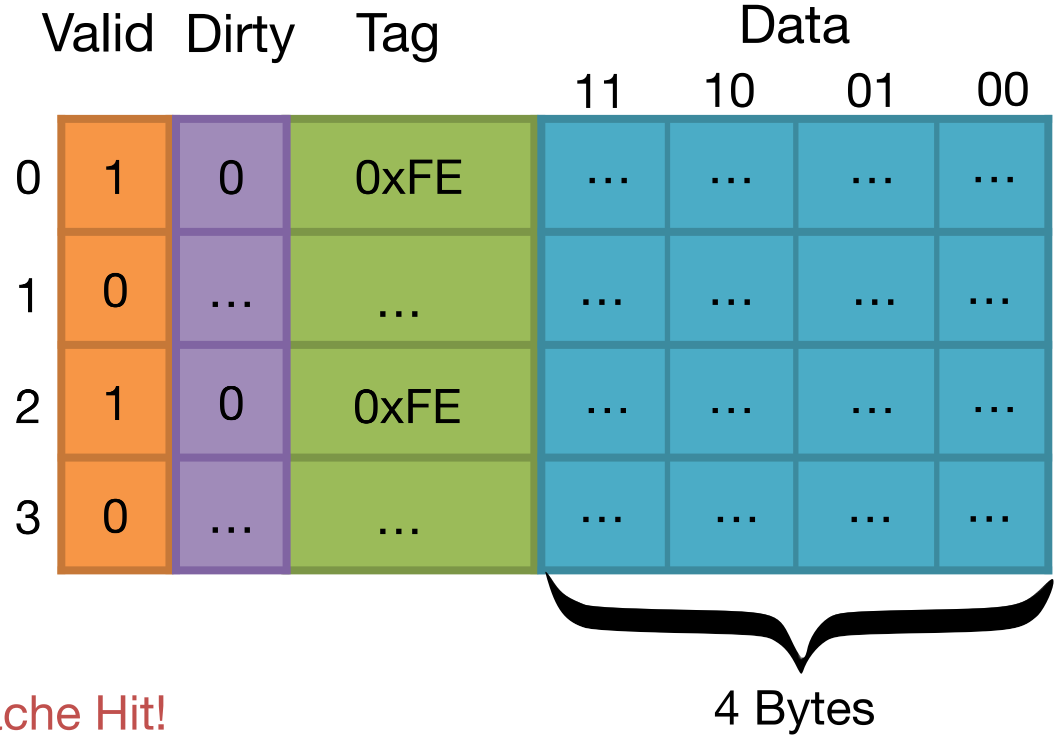
Read byte 0xFE9

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

0b 1111 1110 10 01

T = 0xFE
I = 0x2
O = 0x1

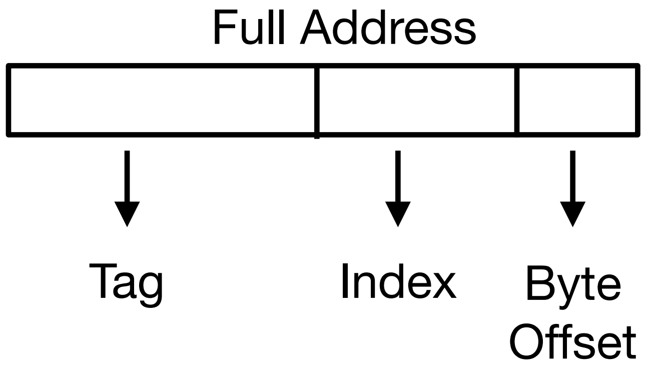
Cache Hit!



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

4 Bytes

Direct Mapped Cache (write-back)



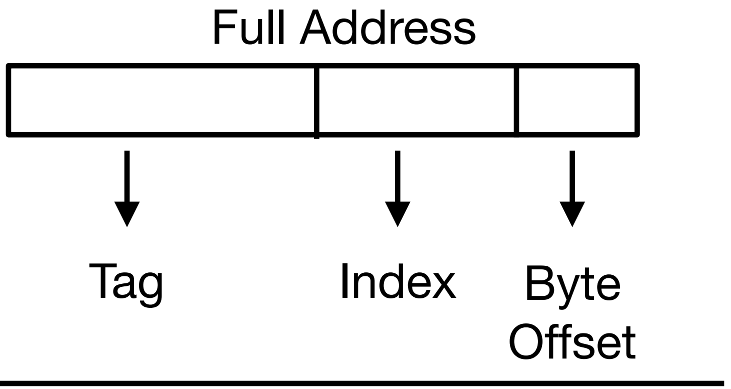
Read byte 0xDF9

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

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

4 Bytes

Direct Mapped Cache (write-back)



Read byte 0xDF9

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

0b 1101 1111 1001

T = 0xDF

I = 0x2

O = 0x1

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

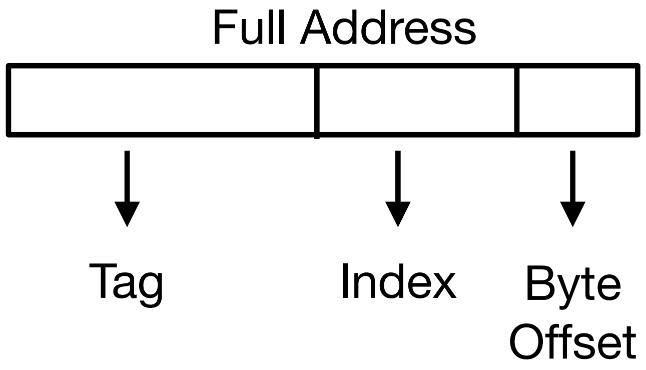
4 Bytes

Cache Miss => bring in entire line

Direct Mapped Cache Replacement

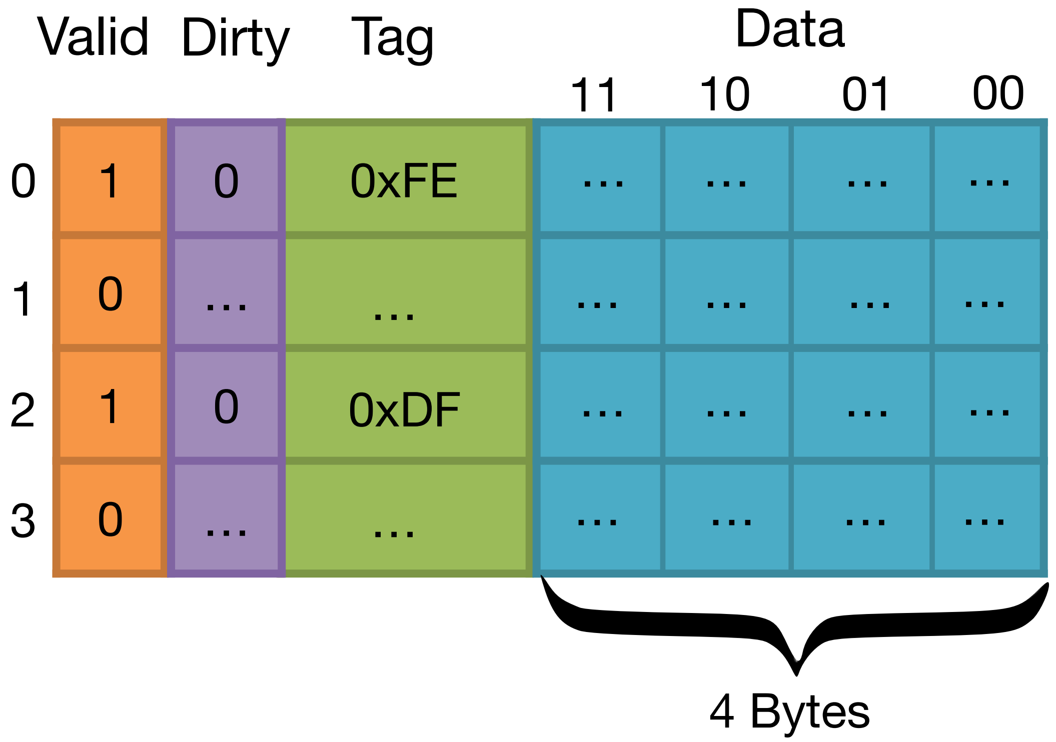
- Every address can only be stored at one location in the cache
- If there is already something else stored at our index, we must evict it

Direct Mapped Cache (write-back)



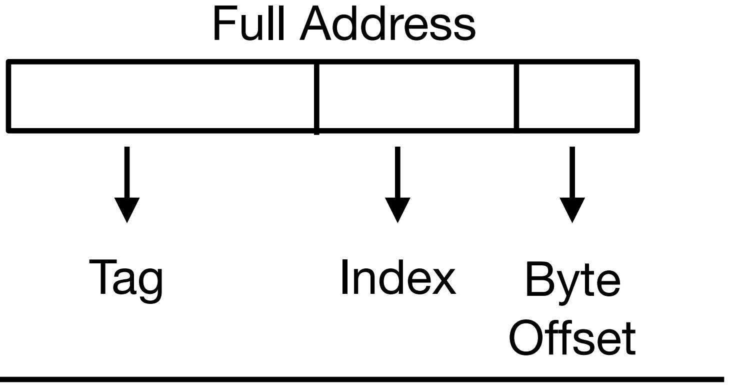
Read byte 0xFE8

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



	Valid	Dirty	Tag	11	10	01	00
0	1	0	0xFE	…	…	…	…
1	0	…	…	…	…	…	…
2	1	0	0xDF	…	…	…	…
3	0	…	…	…	…	…	…

Direct Mapped Cache (write-back)



Read byte 0xFE8

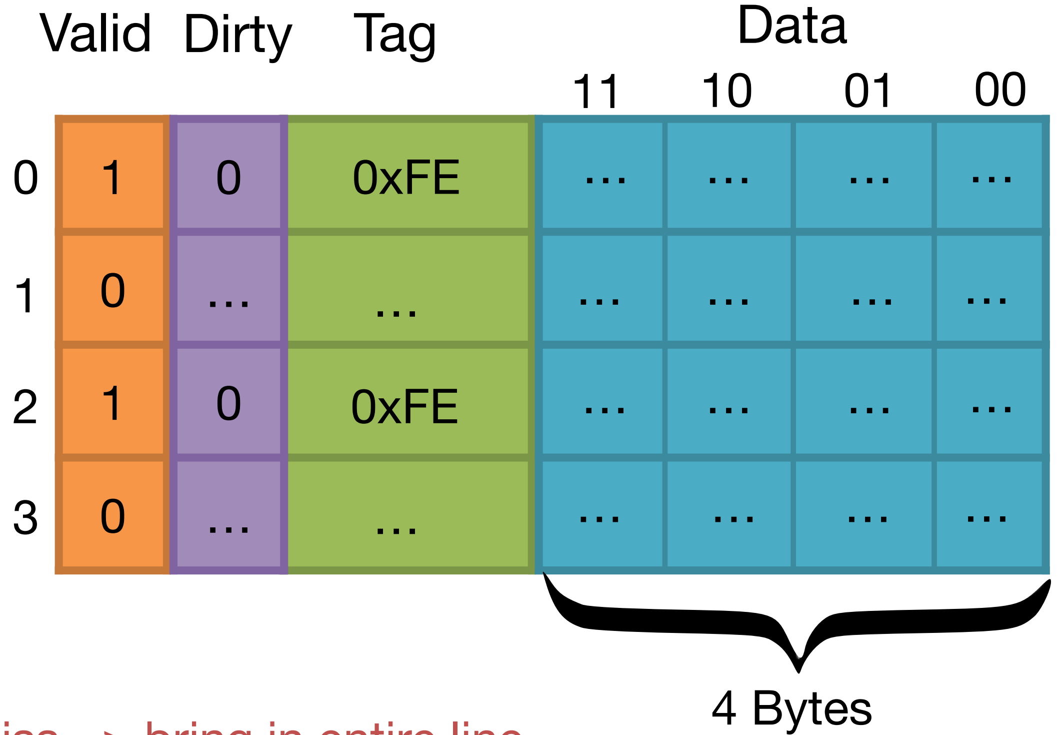
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 1000

T = 0xFE

I = 0x2

O = 0x0



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

Cache Miss => bring in entire line

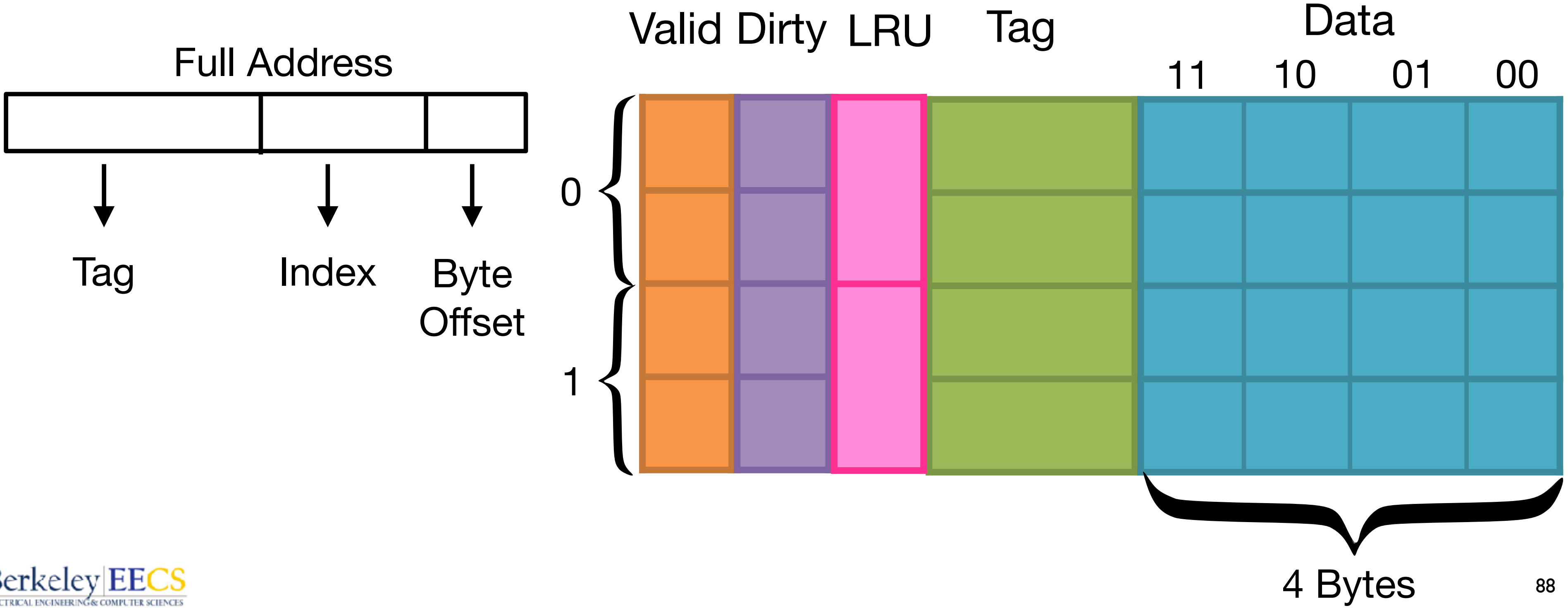
Direct Mapped

- If we had used a fully associative cache, the previous access would have been a hit!
- Direct Mapped leads to more conflicts than fully associative
- Compromise: Set Associative

Set Associative Caches

Set Associative (2-way, write-back)

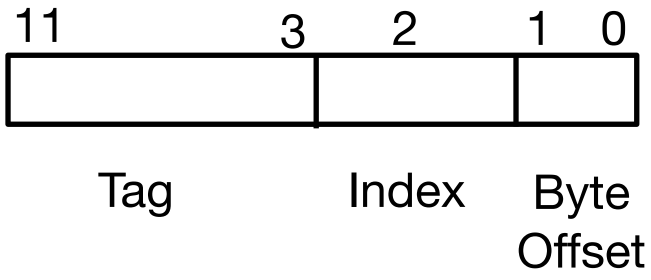
- The data can only be stored at one index, but there are multiple slots to store it in



Set Associative (2-way, write-back)

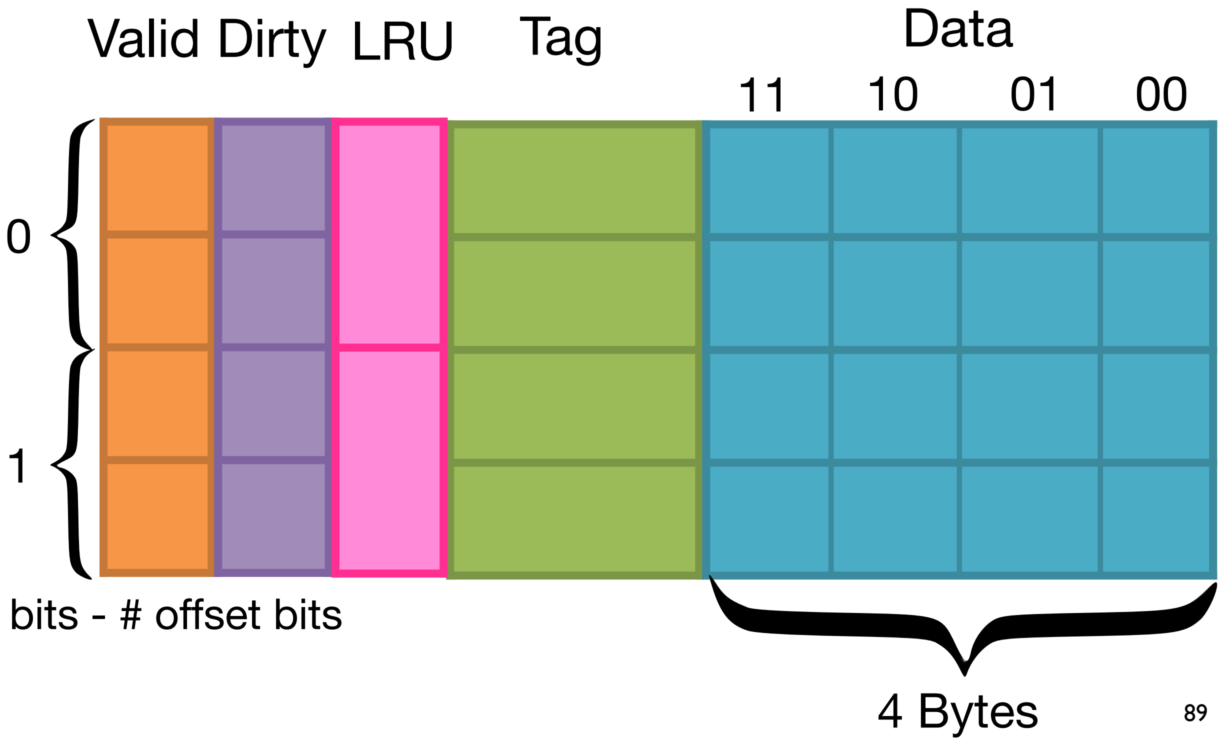
- The data can only be stored at one index, but there are multiple slots to store it in

Full Address: ex: 12 bits



# byte offset bits = $\log_2$(line size) = $\log_2(4) = 2$

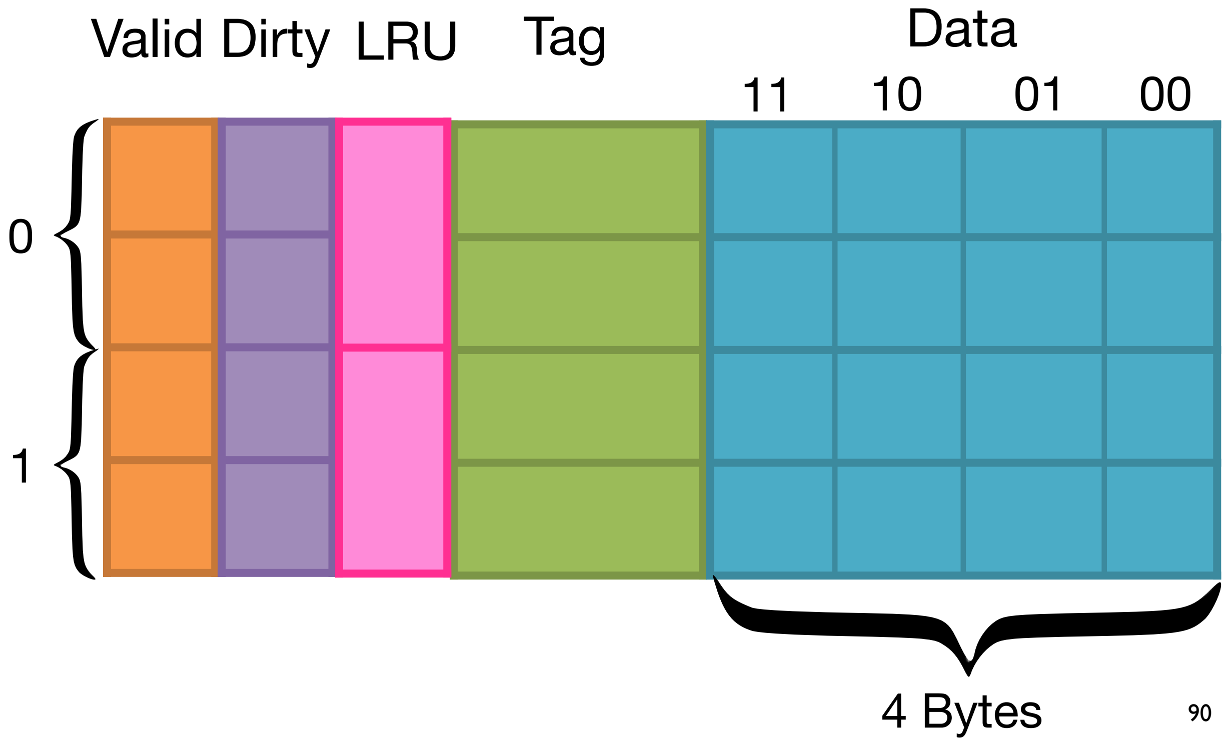
# index bits = $\log_2$(# sets) = $\log_2(2) = 1$



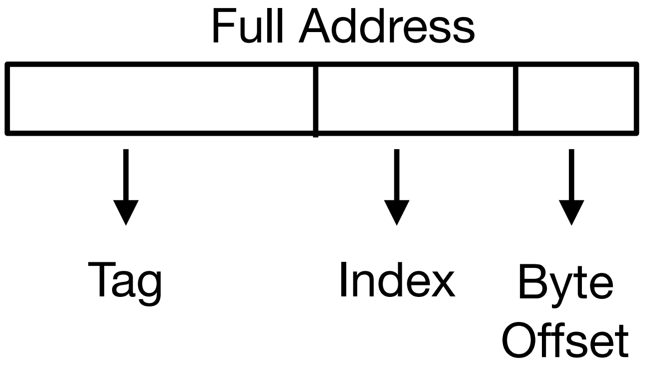
# tag bits = # address bits - # index bits - # offset bits = 12 - 1 - 2 = 9

Set Associative (2-way, write-back)

- The data can only be stored at one index, but there are multiple slots to store it in
- The LRU bit indicates which set has was least recently used

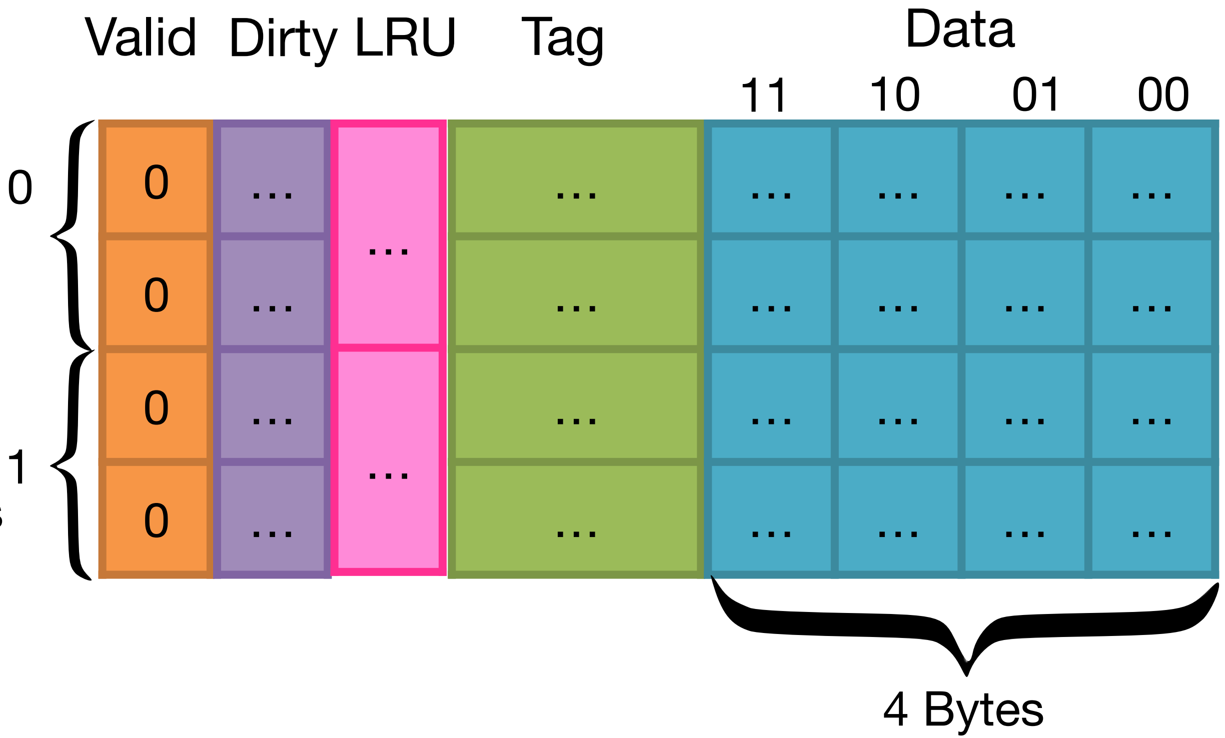


Set Associative (2-way, write-back)

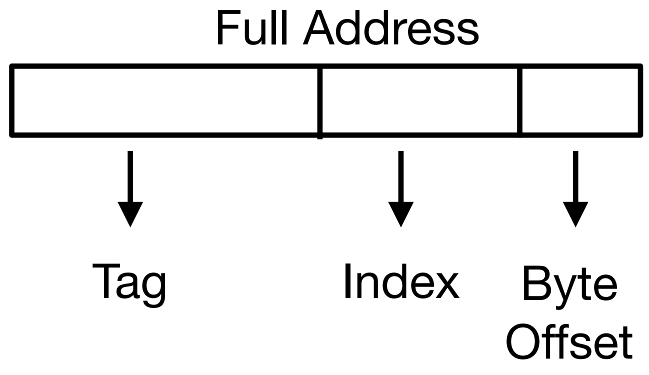


Read Byte 0x8E2

T = 9 bits I = 1 bit O = 2 bits



Set Associative (2-way, write-back)

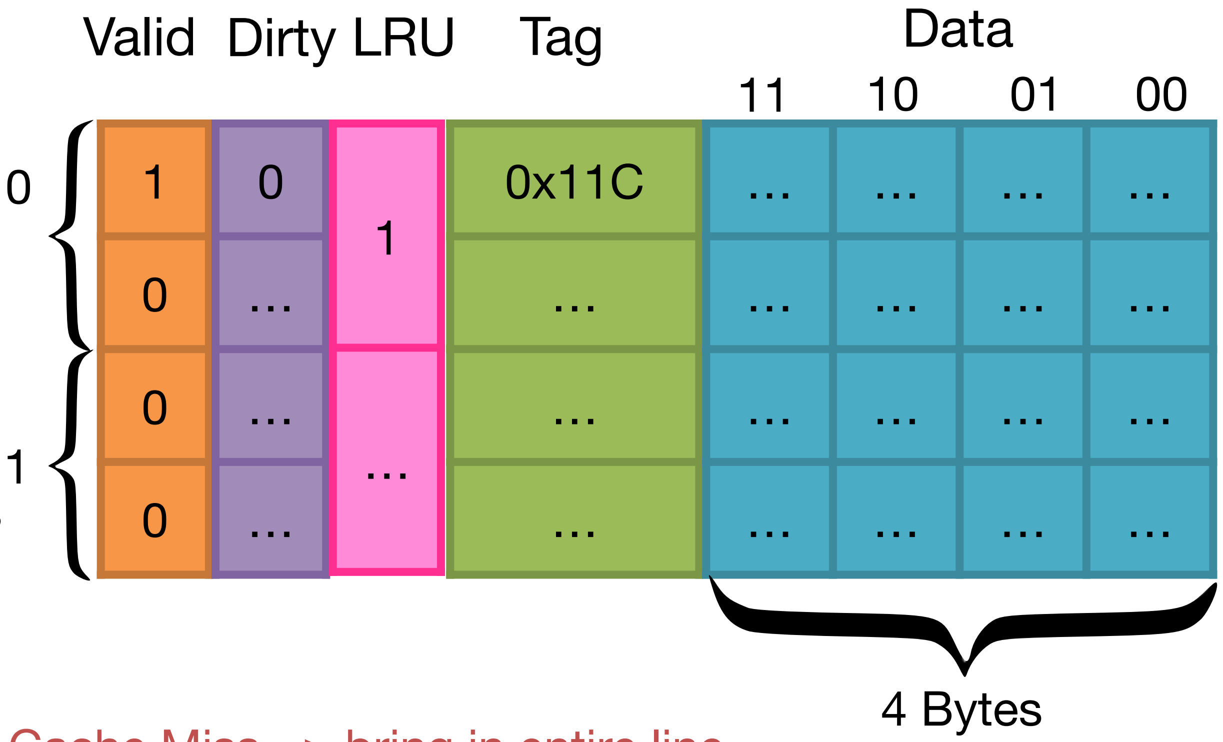


Read Byte 0x8E2

T = 9 bits I = 1 bit O = 2 bits

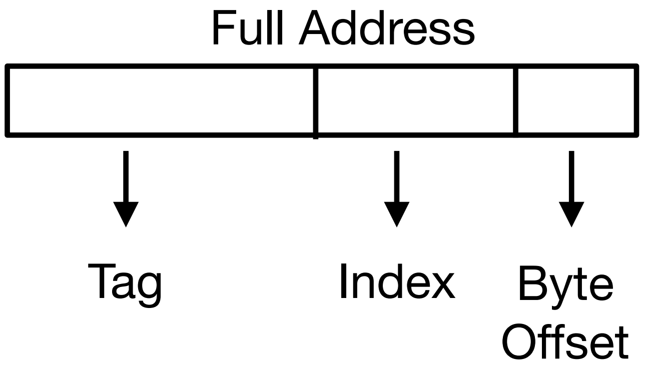
0b 1000 1110 0010

T = 0x11C
I = 0x0
O = 0x2



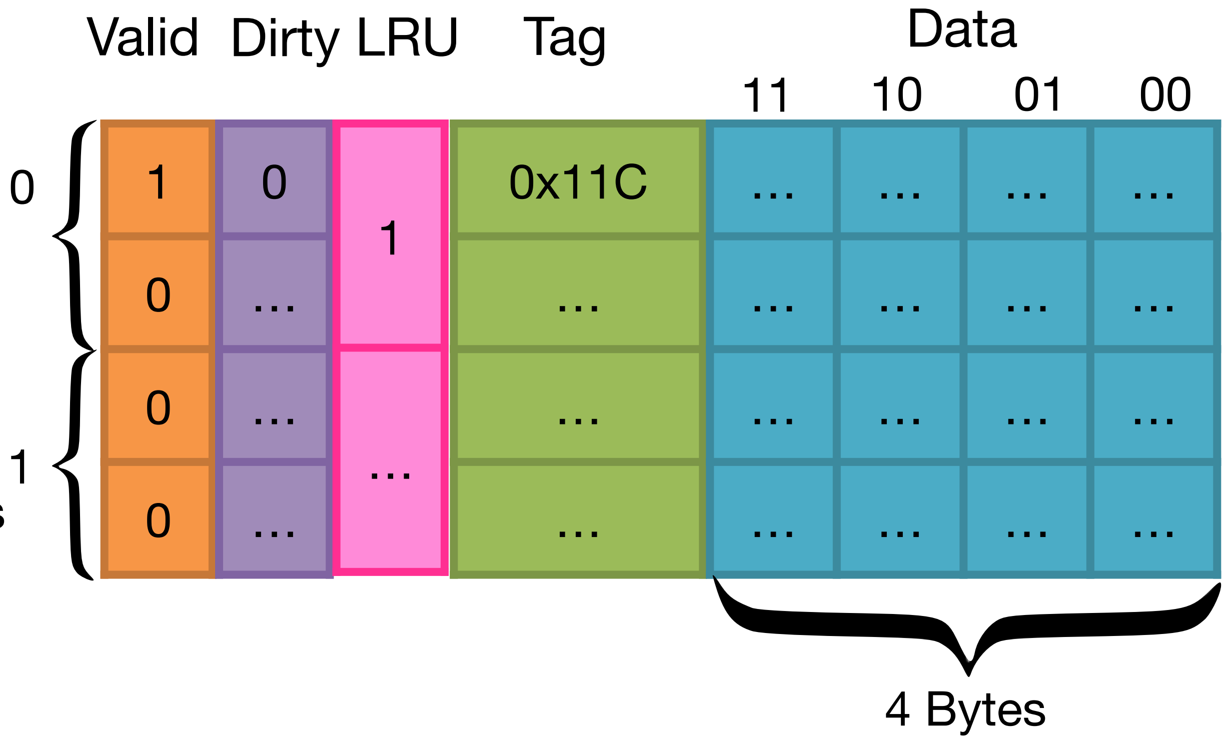
Cache Miss => bring in entire line

Set Associative (2-way, write-back)

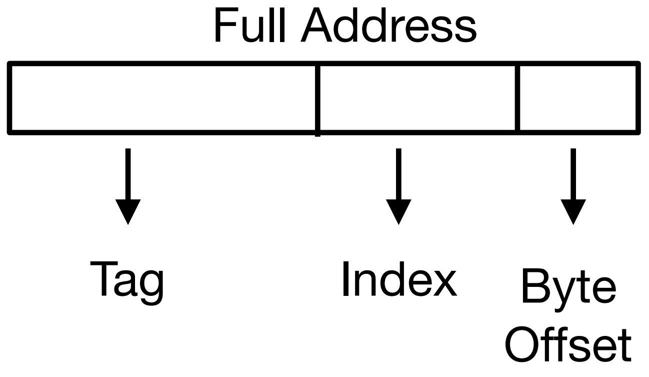


Read Byte 0x8E8

T = 9 bits I = 1 bit O = 2 bits



Set Associative (2-way, write-back)

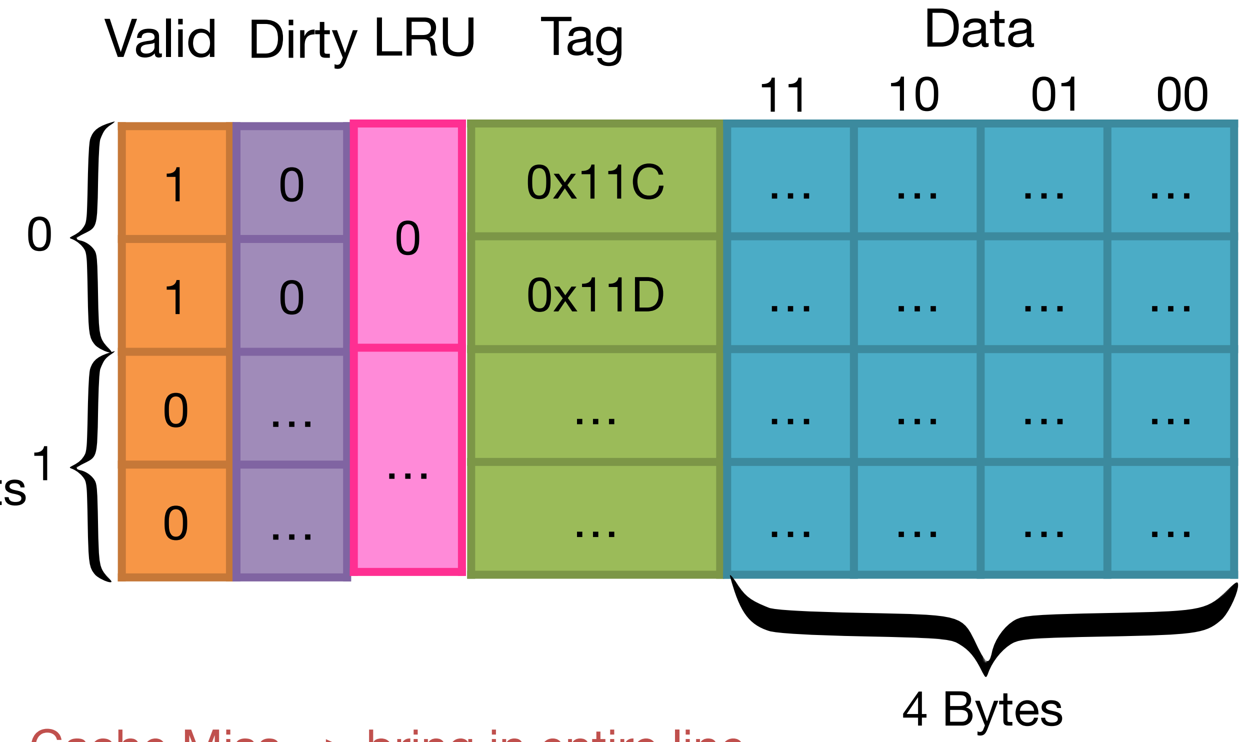


Read Byte 0x8E8

T = 9 bits   I = 1 bit   O = 2 bits

0b 1000 1110 1000

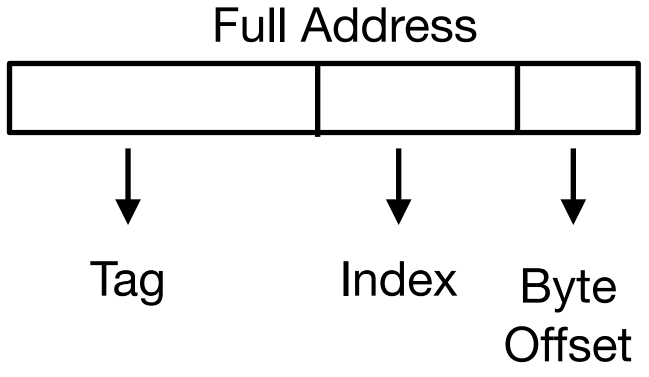
T = 0x11D
I = 0x0
O = 0x0



	Valid	Dirty	LRU	Tag	11	10	01	00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	0	…	…	…	…	…	…	…
	0	…		…	…	…	…	…

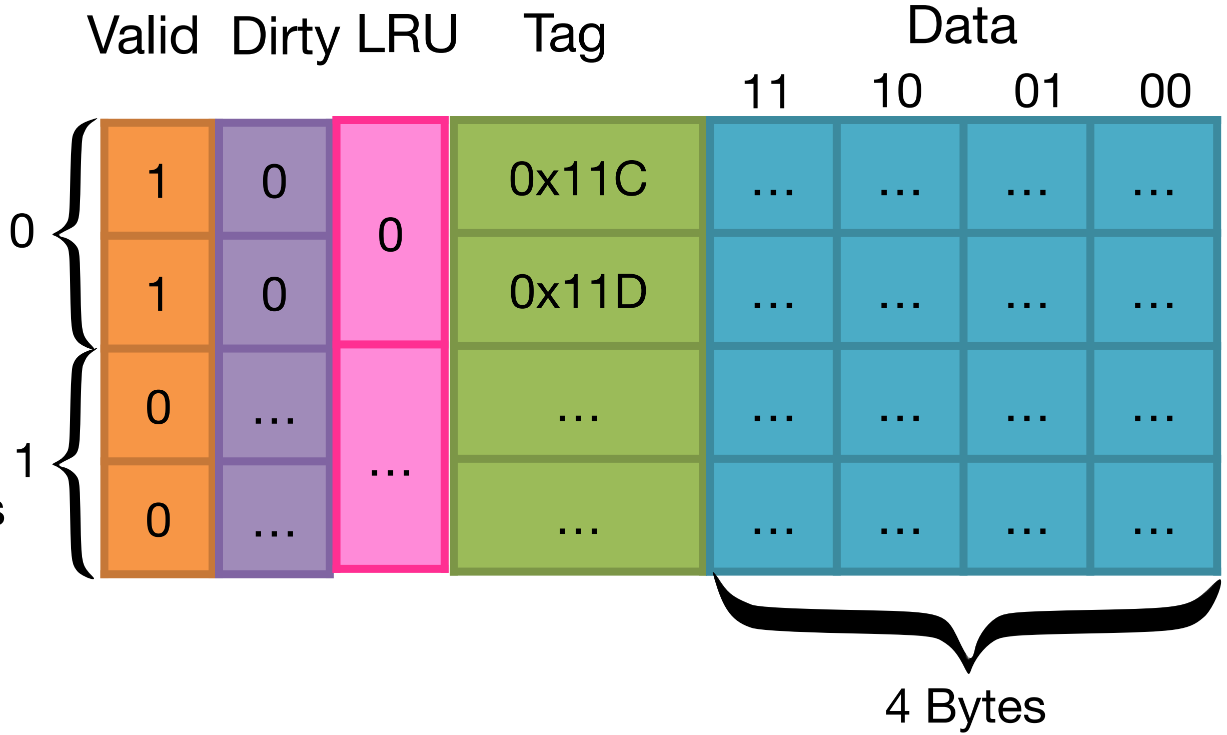
Cache Miss => bring in entire line

Set Associative (2-way, write-back)



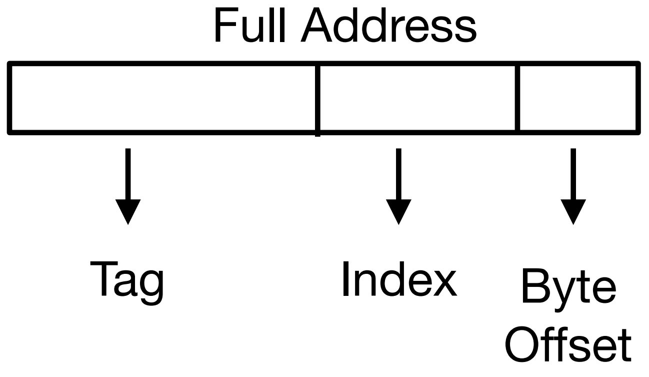
Read Byte 0x8E9

T = 9 bits I = 1 bit O = 2 bits



Set	Valid	Dirty	LRU	Tag	11	10	01	00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	0	…	…	…	…	…	…	…
	0	…		…	…	…	…	…

Set Associative (2-way, write-back)

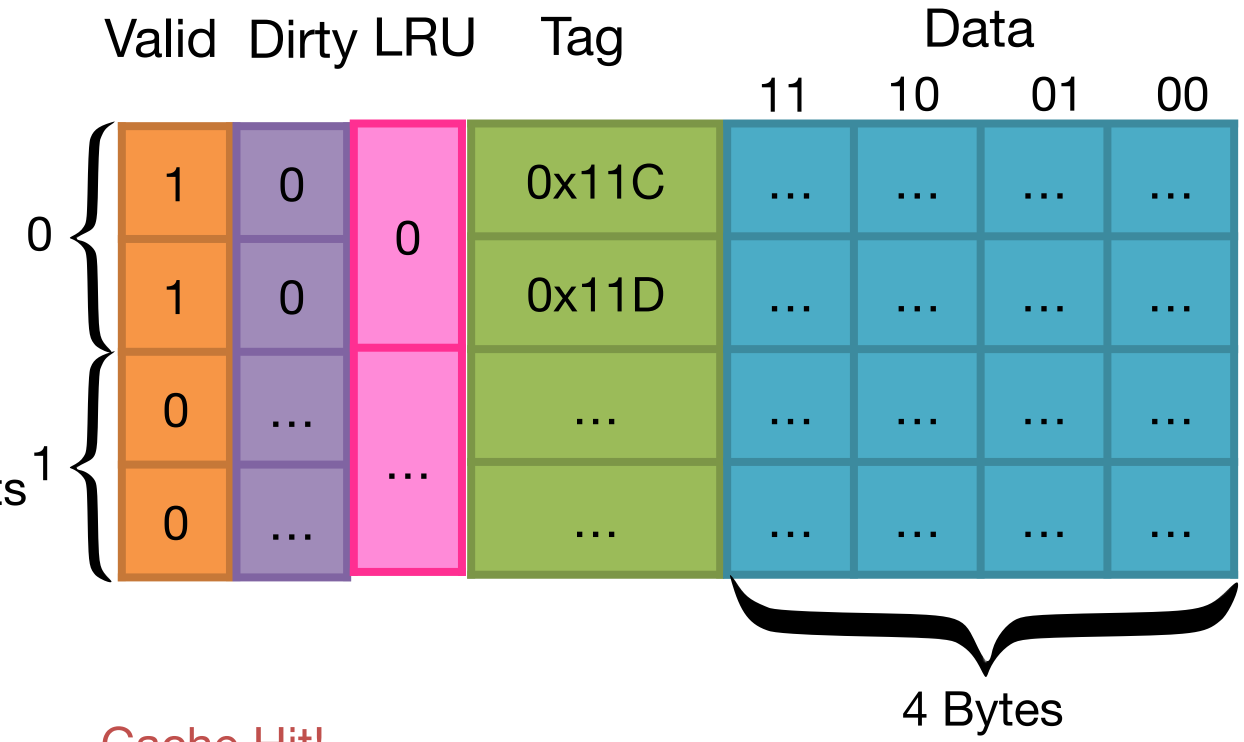


Read Byte 0x8E9

T = 9 bits I = 1 bit O = 2 bits

0b 1000 1110 1001

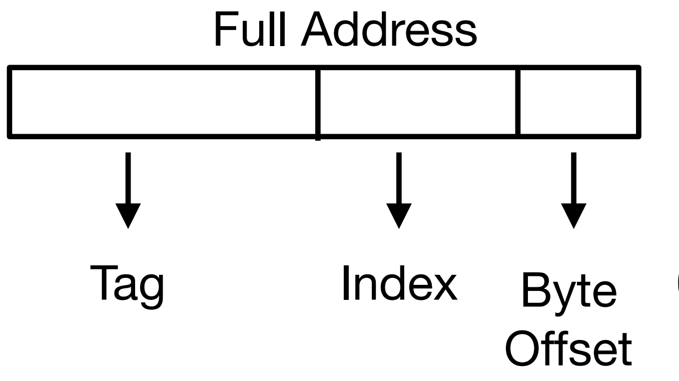
T = 0x11D
I = 0x0
O = 0x1



	Valid	Dirty	LRU	Tag	11	10	01	00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	0	…	…	…	…	…	…	…
	0	…		…	…	…	…	…

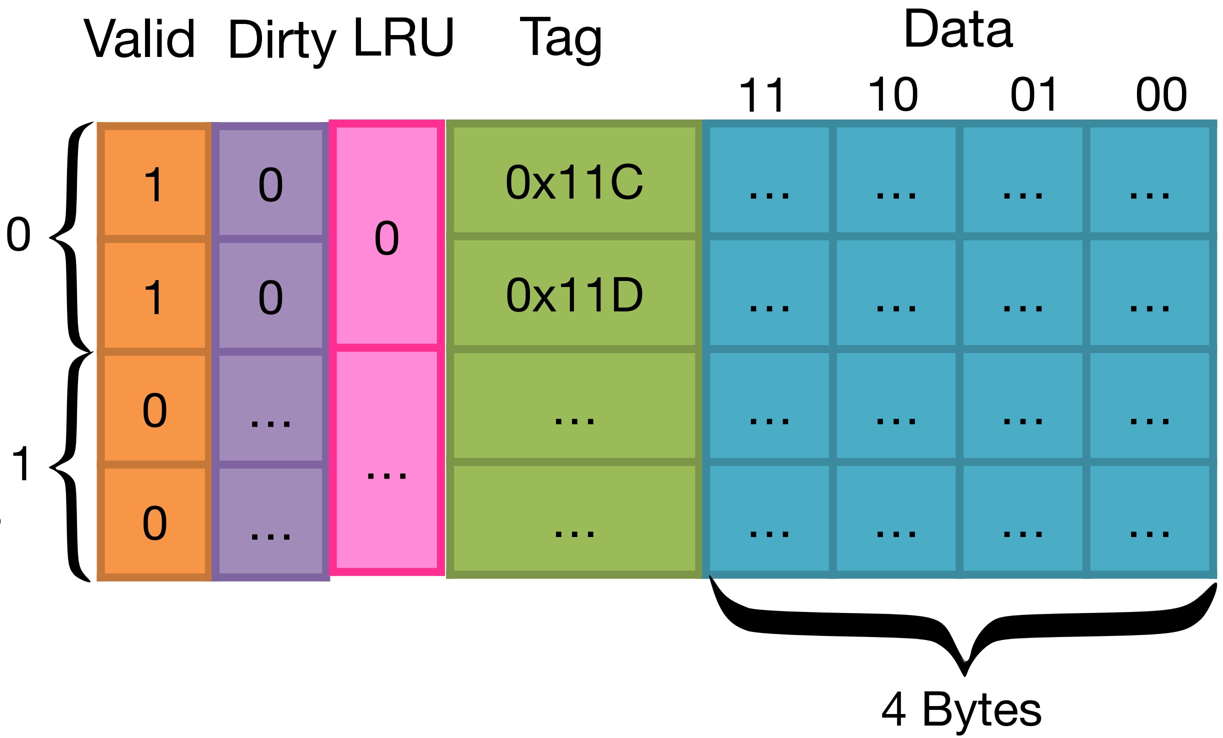
Cache Hit!

Set Associative (2-way, write-back)



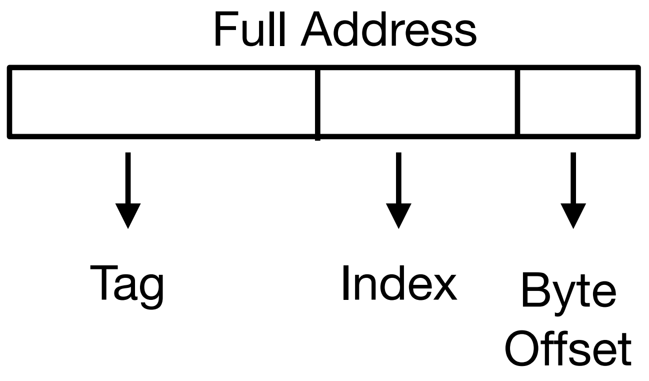
Read Byte 0xDF7

T = 9 bits   I = 1 bit   O = 2 bits



Set	Valid	Dirty	LRU	Tag	11	10	01	00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	0	…	…	…	…	…	…	…
	0	…		…	…	…	…	…

Set Associative (2-way, write-back)

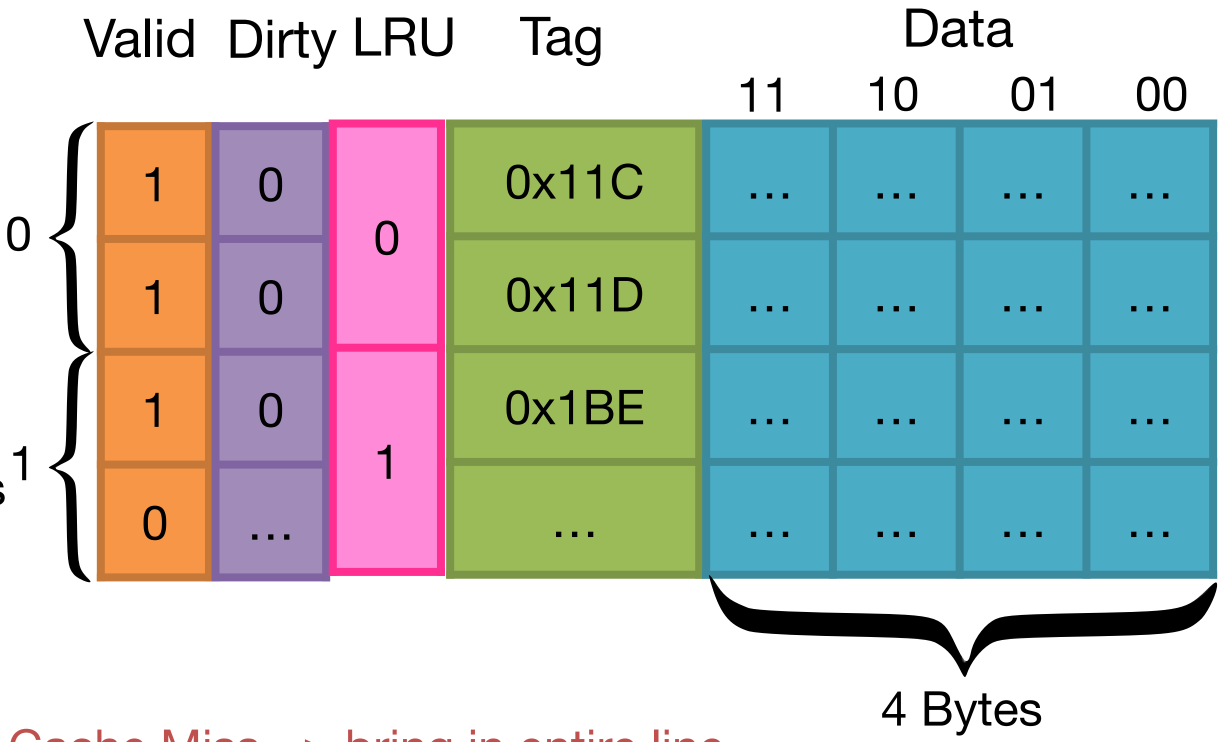


Read Byte 0xDF7

T = 9 bits I = 1 bit O = 2 bits

0b 1101 1111 0111

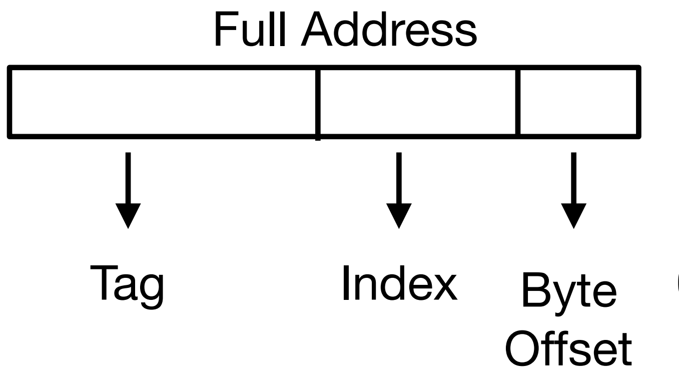
T = 0x1BE
I = 0x1
O = 0x3



Set	Valid	Dirty	LRU	Tag	Data 11	Data 10	Data 01	Data 00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	1	0	1	0x1BE	…	…	…	…
	0	…		…	…	…	…	…

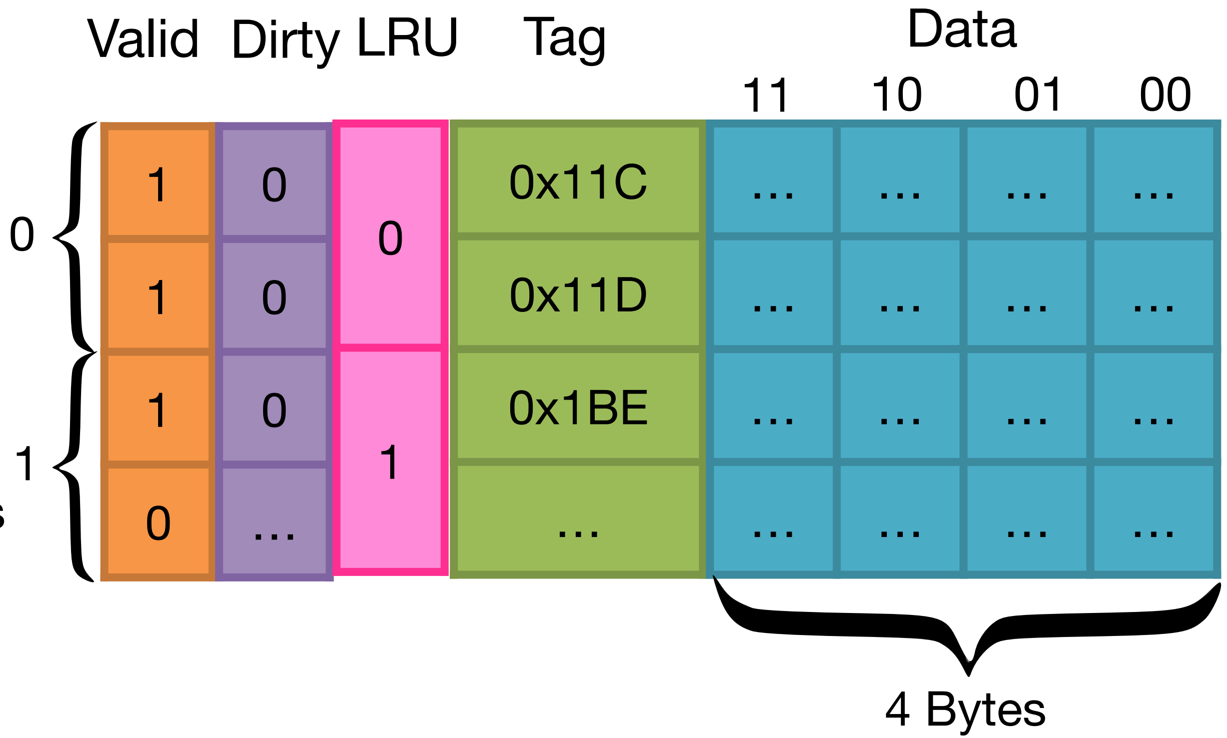
Cache Miss => bring in entire line

Set Associative (2-way, write-back)



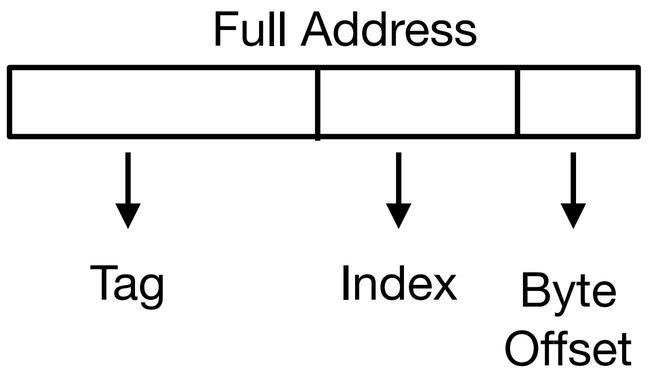
Read Byte 0xAB8

T = 9 bits I = 1 bit O = 2 bits



Set	Valid	Dirty	LRU	Tag	11	10	01	00
0	1	0	0	0x11C	…	…	…	…
	1	0		0x11D	…	…	…	…
1	1	0	1	0x1BE	…	…	…	…
	0	…		…	…	…	…	…

Set Associative (2-way, write-back)

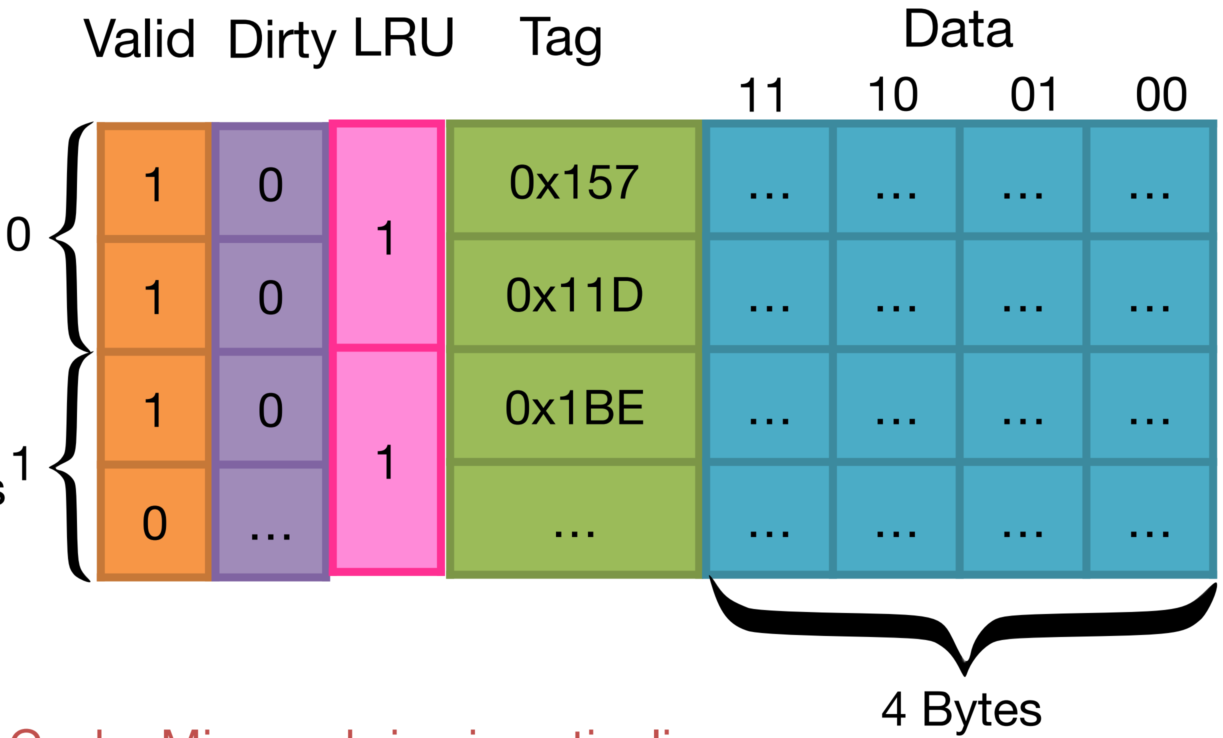


Read Byte 0xAB8

T = 9 bits I = 1 bit O = 2 bits

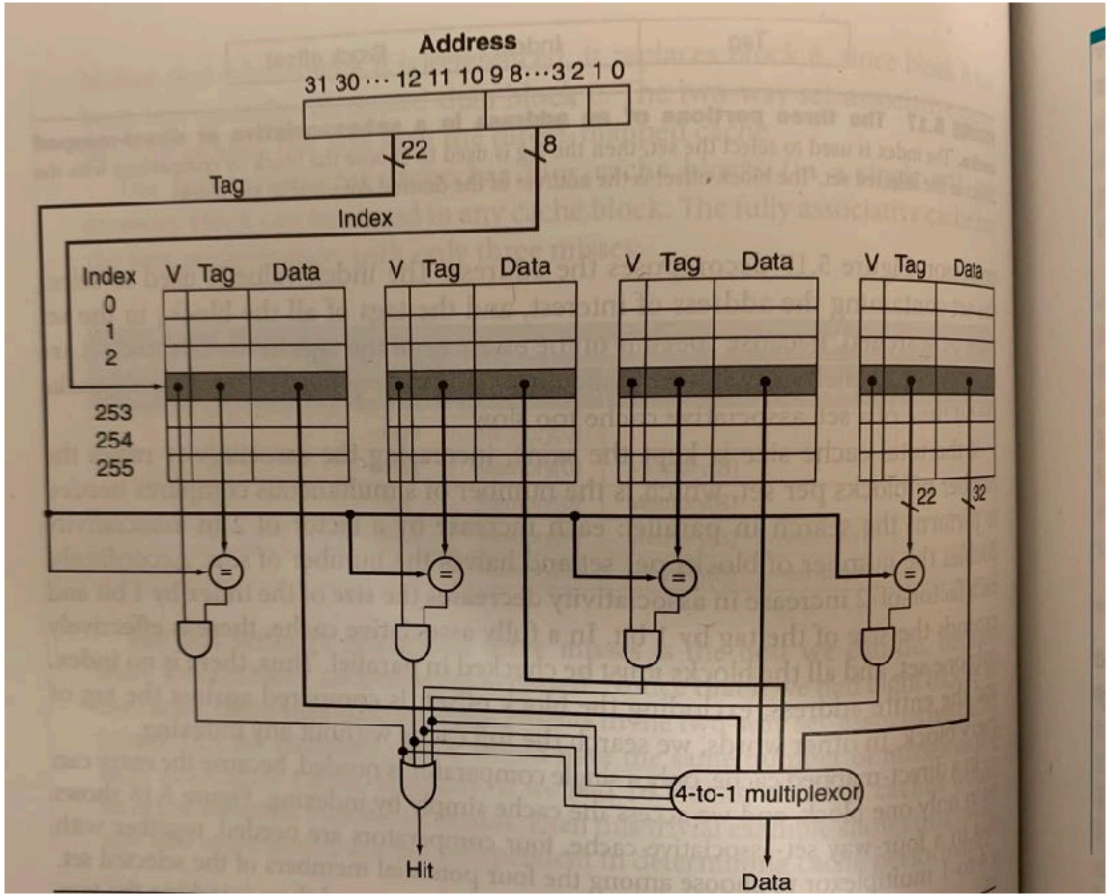
0b 1010 1011 1000

T = 0x157
I = 0x0
O = 0x0



Cache Miss => bring in entire line

Implementation of 4-way Set Associative Cache



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
  - Occur in set-associative or direct mapped caches when multiple blocks compete for the same set
  - Misses of this type would not occur in a fully associative cache of the same type

Comparisons

Fully Associative

- A specific line of data can be stored in any line of the cache
- No index
- Need to choose replacement 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

Set Associative

- A specific line of data can be stored at only one index of the cache (but multiple lines can be in each index)
- Has index
- Need to choose replacement policy

Next Lecture

- Cache Performance
- Multilevel Caches