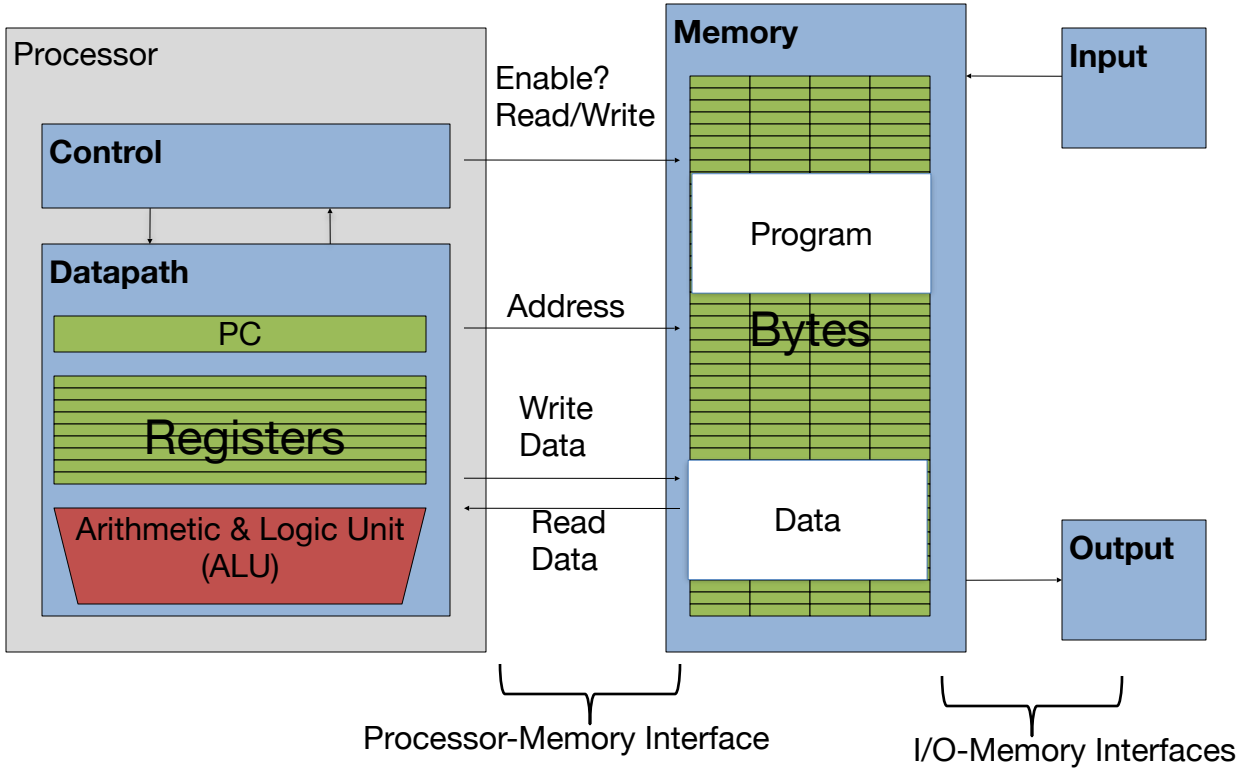
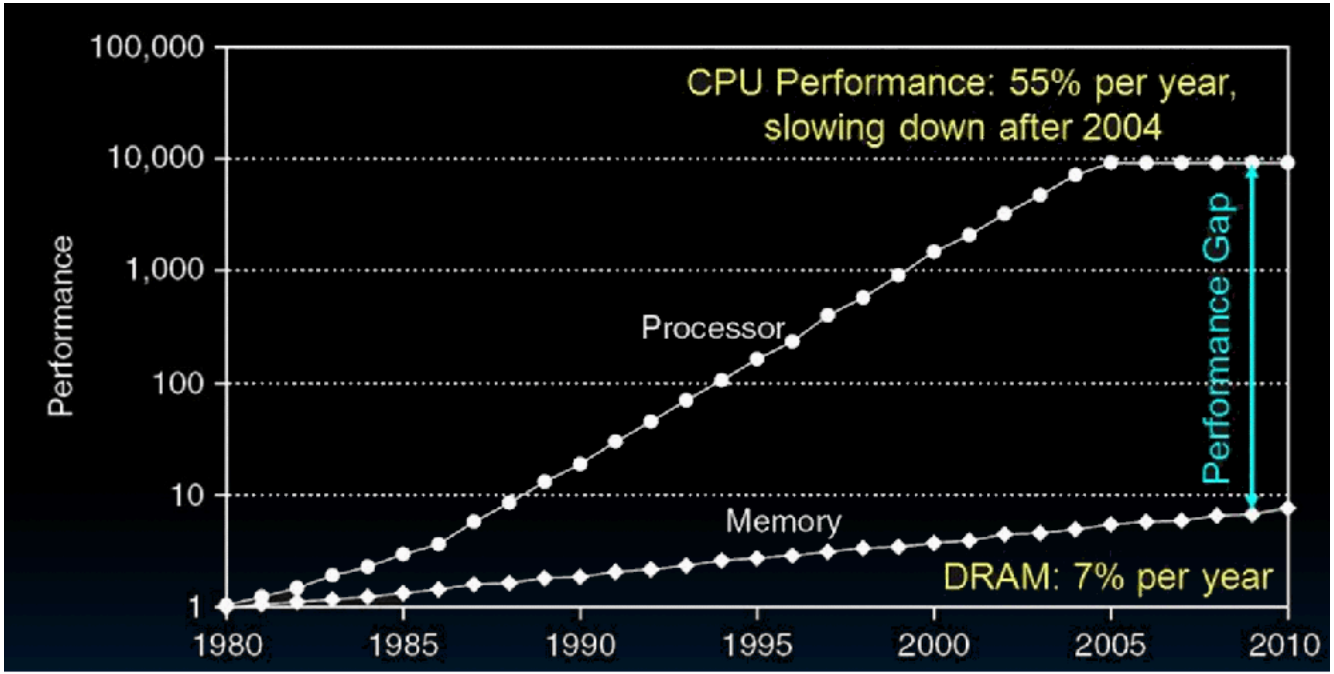
Caches

Components of a Computer



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!

Library Analogy

- Time to find a book in a large library
  - Search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book
- Larger libraries worsen both delays
- Electronic memories have same issue, plus the technologies used to store a bit slow down as density increases (e.g., SRAM vs. DRAM vs. Disk)

What to do: Library Analogy

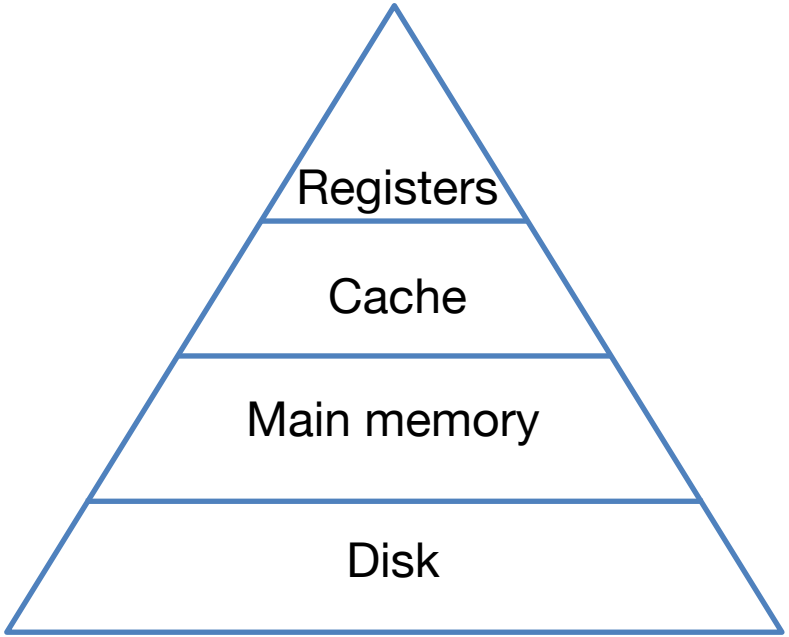
- Write a report using library books
- Go to library, look up books, fetch from stacks, and place on desk in library
- If need more, check out, keep on desk
  - But don't return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries

Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level…
  - Introducing a “memory cache”
- Usually on the same chip as the CPU
  - Faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory
- Most processors have separate caches for instructions and data.

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



Memory Hierarchy

- Programmer-invisible hardware mechanism
- Gives illusion of the speed of the fastest memory with the size of the largest memory
- How do we make it fast?
  - Hierarchy
- How do we make it appear large?
  - Keep the right data in the cache

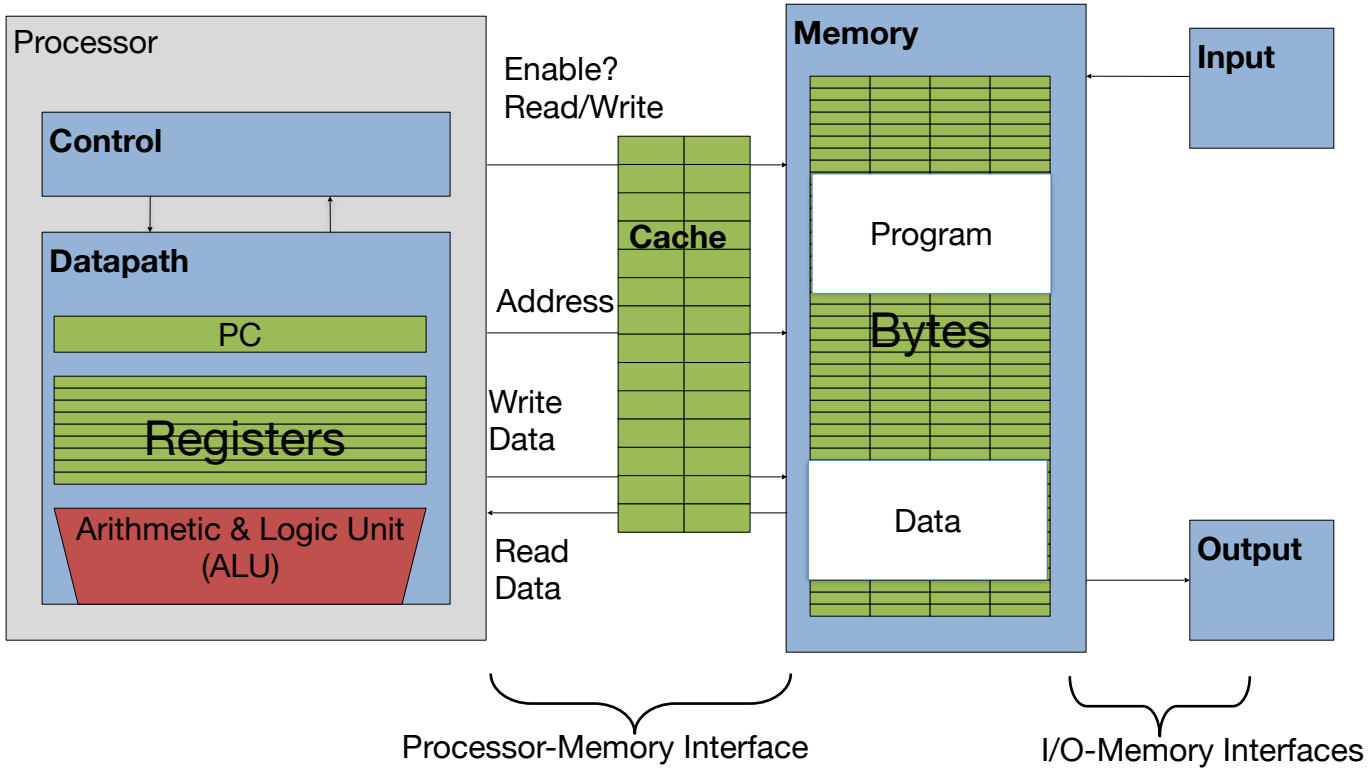
Memory Hierarchy Basis

- Cache contains copies of data that are being used
- Caches work on the principles of temporal and spatial locality.
  - Temporal locality (locality in time): If we use it now, chances are that we'll want to use it again soon.
  - Spatial locality (locality in space): If we use a piece of memory, chances are we'll use the neighboring pieces soon.

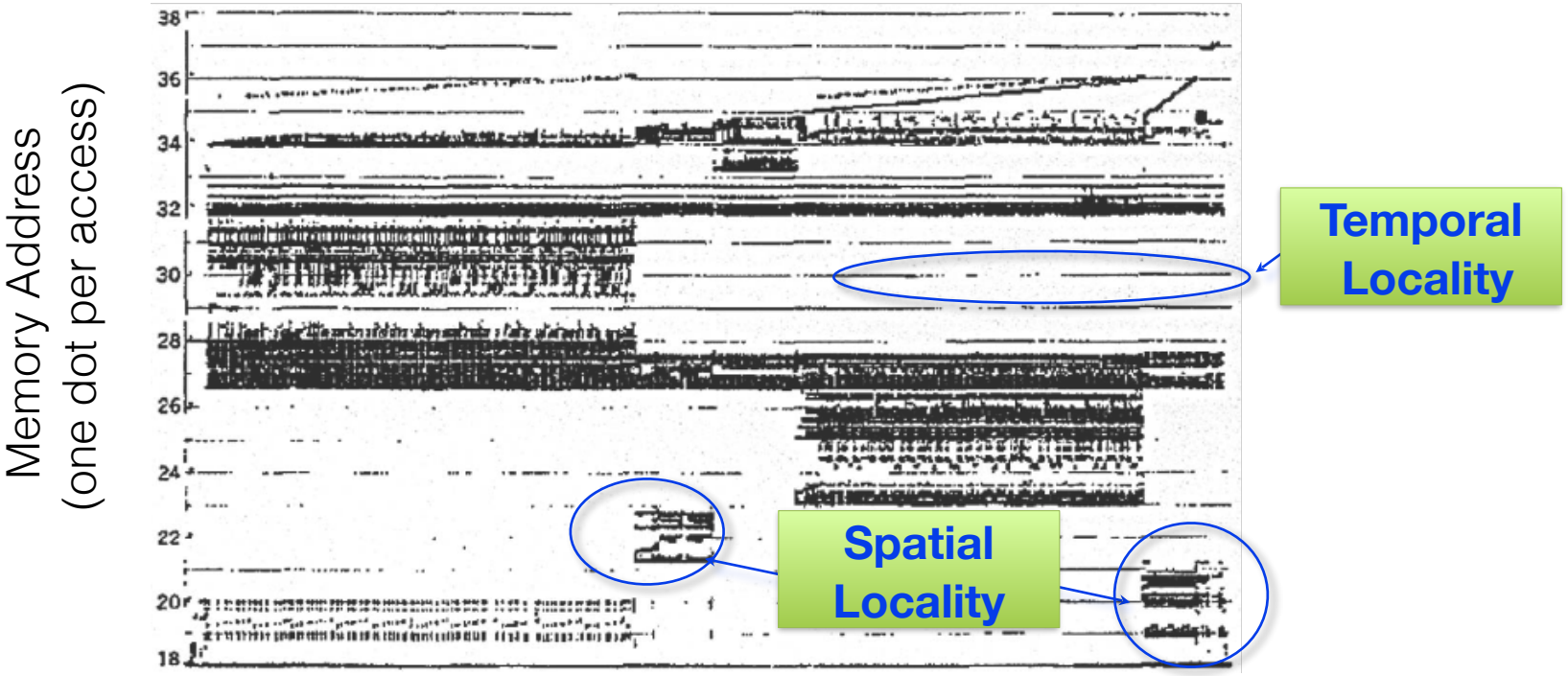
Taking Advantage of Locality

- 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

Adding Cache to the Computer

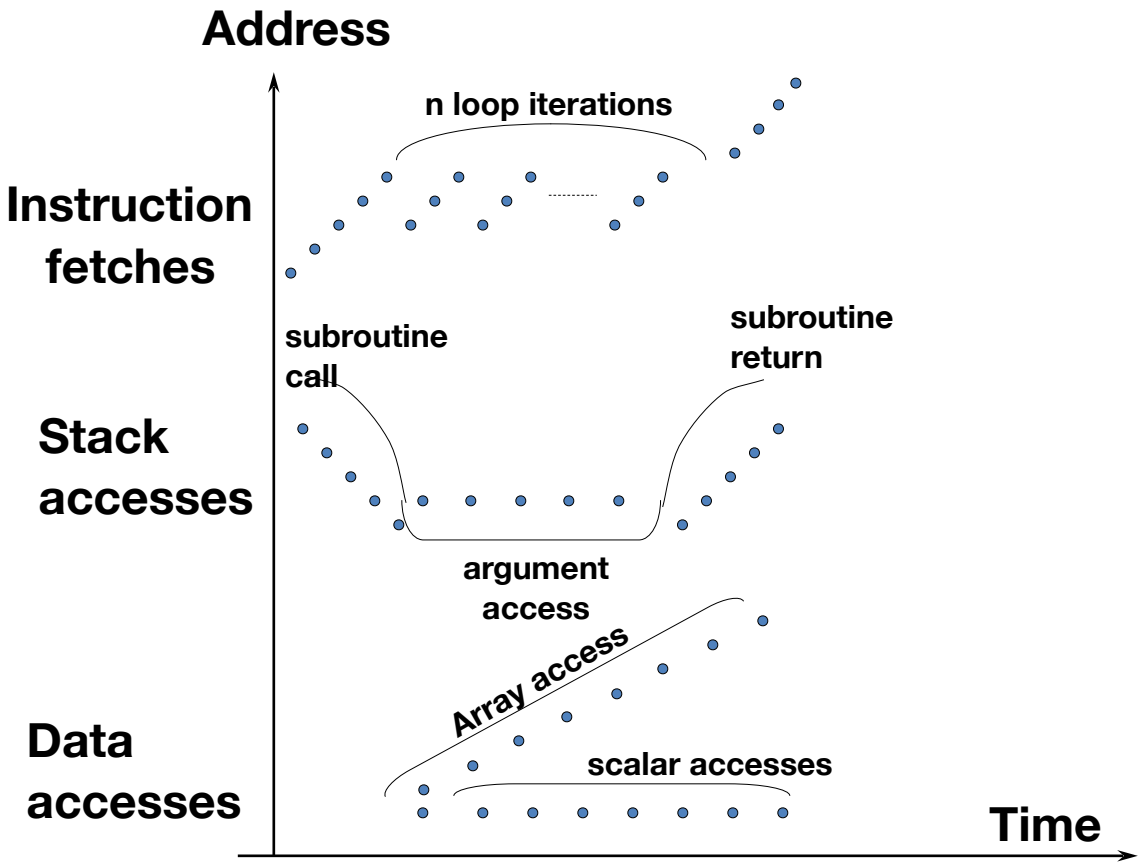


Memory Reference Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Good Memory Reference Patterns



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`

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

Cache Hit vs Cache Miss

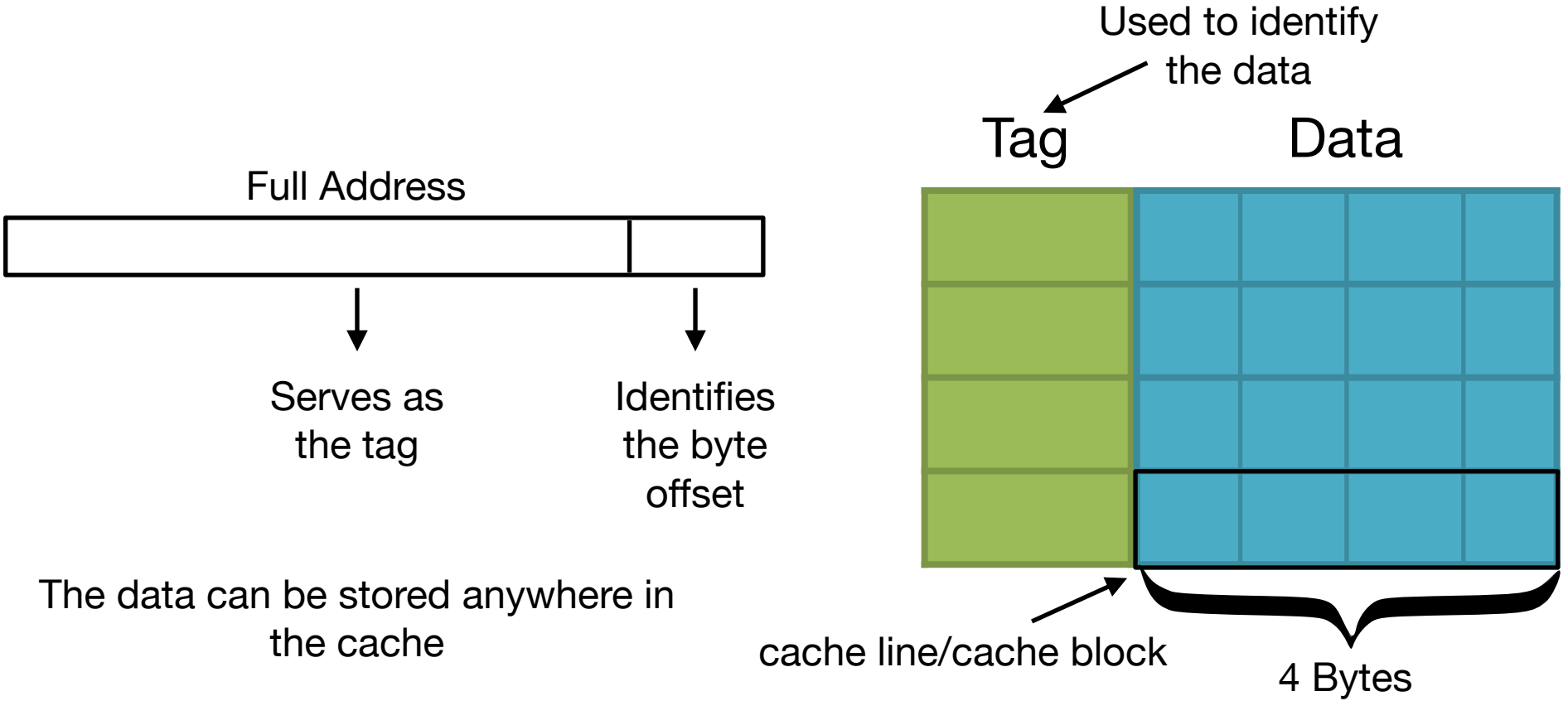
- Cache Hit
  - The data you were looking for is in the cache
  - Retrieve the data from the cache and bring it to the processor
- Cache Miss
  - The data you were looking for is not in the cache
  - Go to the memory to find the data, put the data in the cache, and bring it to the processor

How is our data stored in the cache?

- Fully Associative
- Direct Mapped
- Set-Associative

Fully Associative Cache

Fully Associative Cache



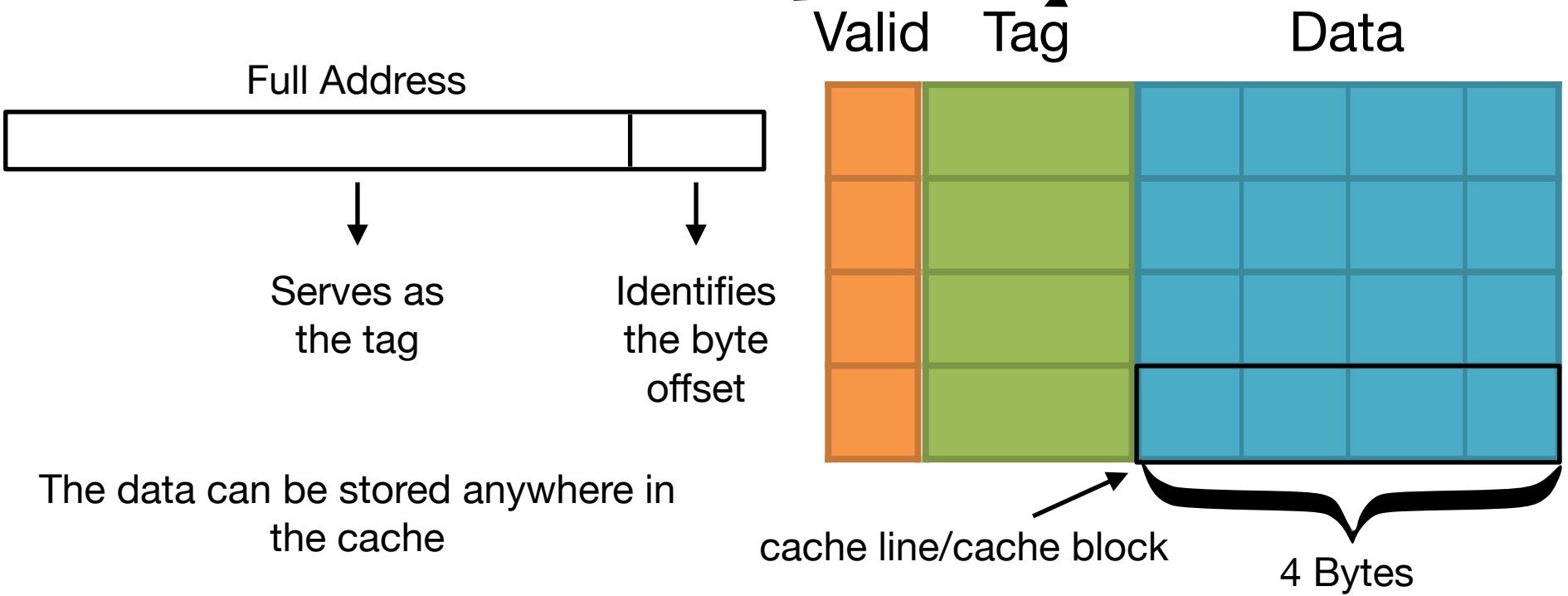
Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a “valid bit” to the cache tag entry
- 0 => cache miss, even if by chance, address = tag
- 1 => cache hit, if processor address = tag

Fully Associative Cache

Used to identify if the data is valid

Used to identify the data

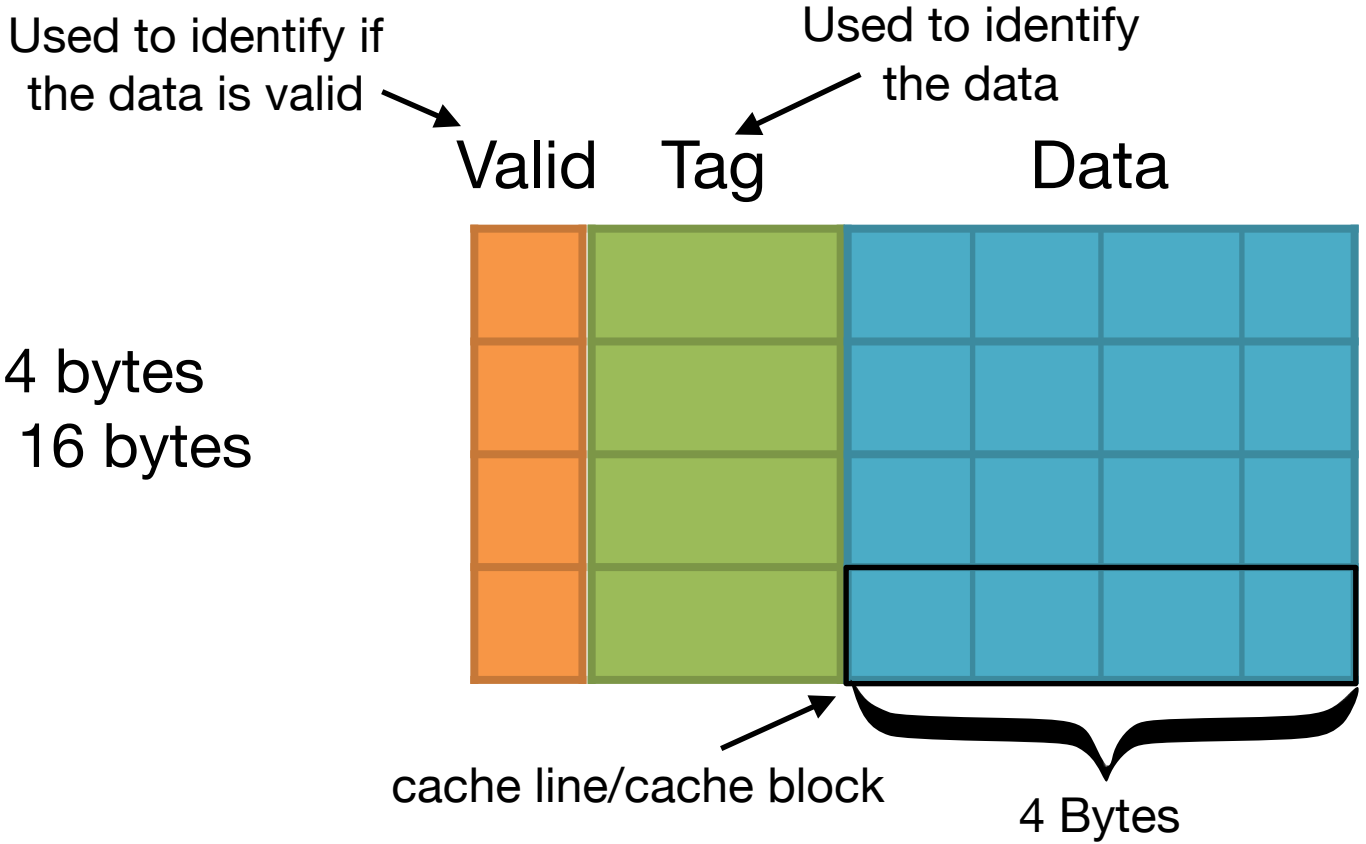


Terminology

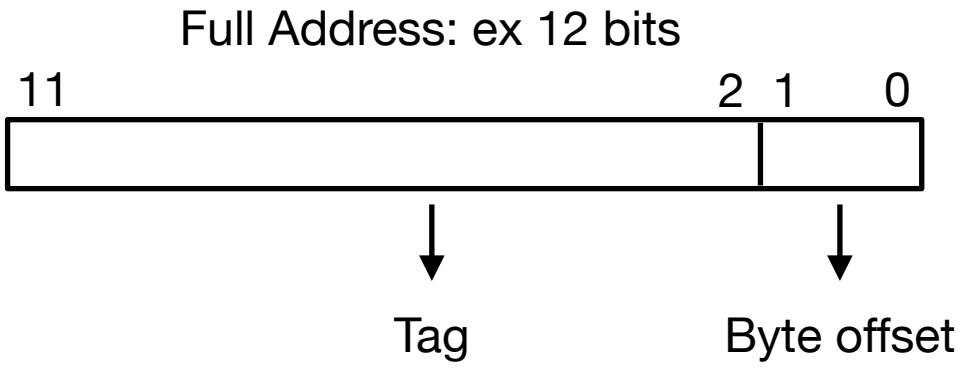
- Cache line/block
  - A single entry in the cache
- Line size / block size
  - The number of bytes in each cache line
- Tag
  - Identifies the data stored at a given cache line
- Valid bit
  - Tells you if the data stored at a given cache line is valid
- Capacity
  - The total number of data bytes that can be stored in a cache

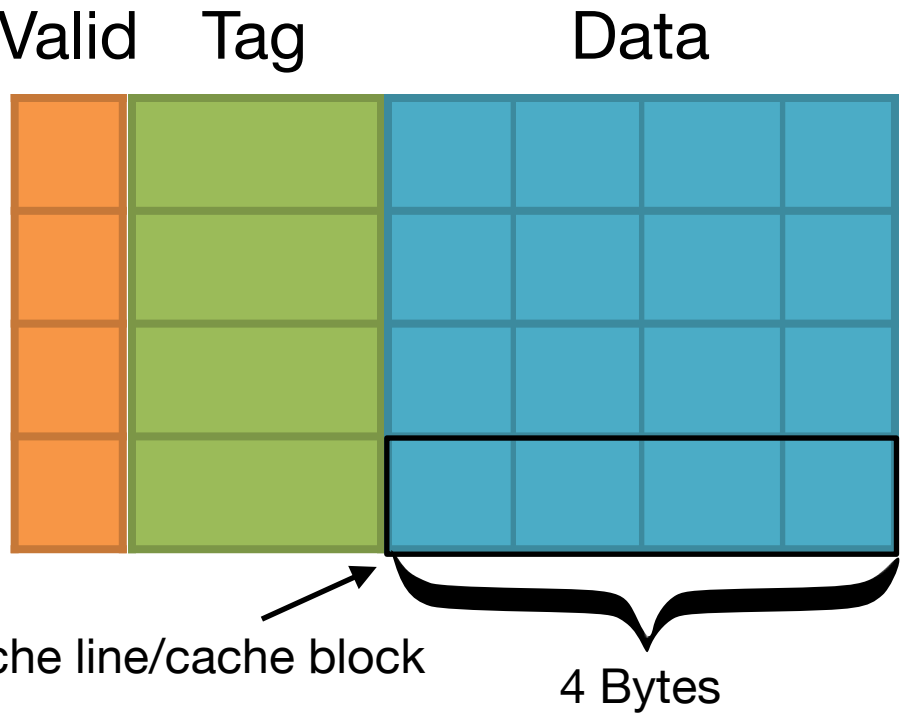
Fully Associative Cache

line size = 4 bytes
capacity = 16 bytes



Fully Associative Cache Address Breakdown

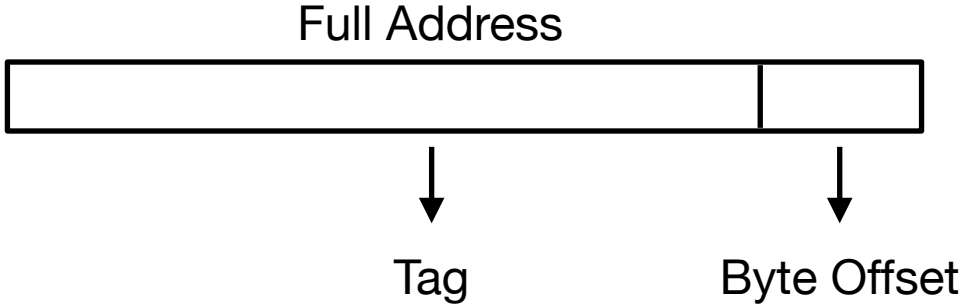




byte offset bits = $\log_2$(line size)
= $\log_2(4) = 2$

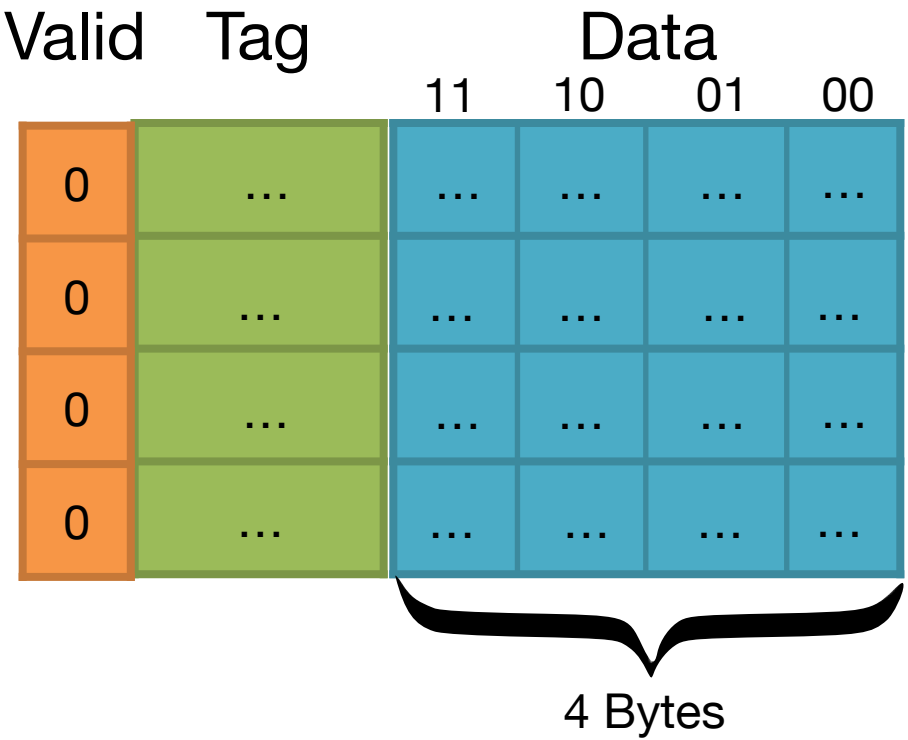
tag bits = # address bits - # offset bits
= 12 - 2 = 10

Fully Associative Cache

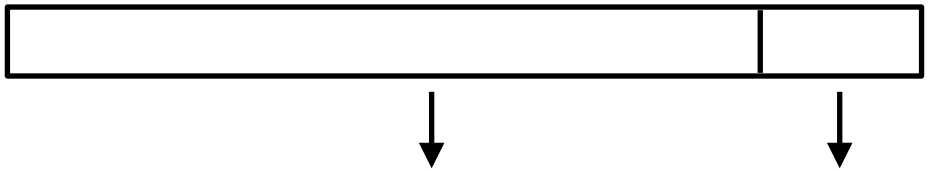


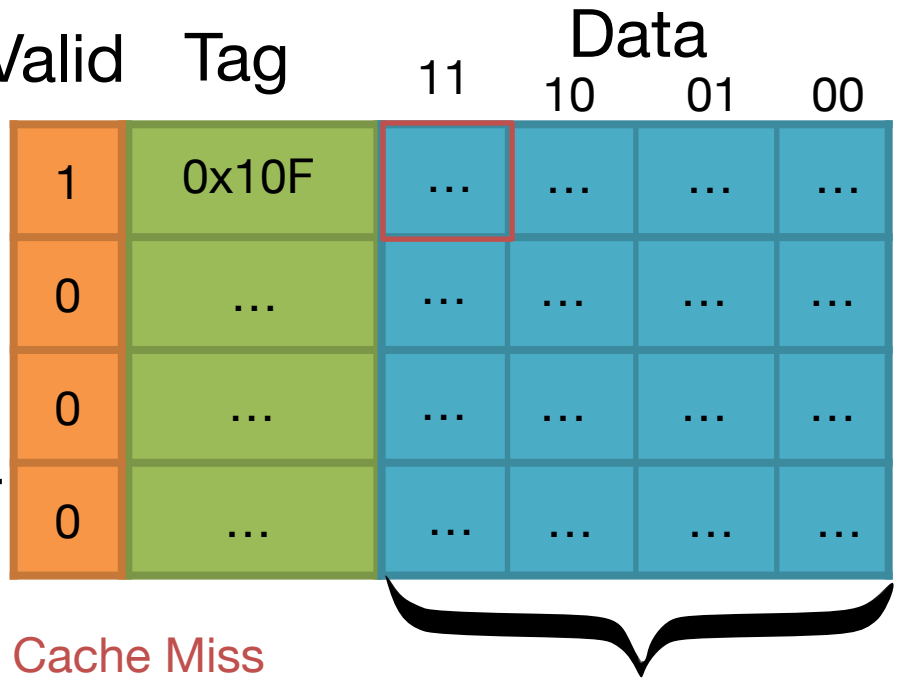
Ex: load byte at 0x43F

Tag = 10 bits Byte offset = 2 bits



Fully Associative Cache





Ex: load byte at 0x43F

0100 0011 1111

Tag = 10 bits Byte offset = 2 bits

Tag = 0x10F Byte offset = 0x3

When you load the value from memory, you'll load an entire block (in this case a block is 4 bytes) even if you are only reading one byte

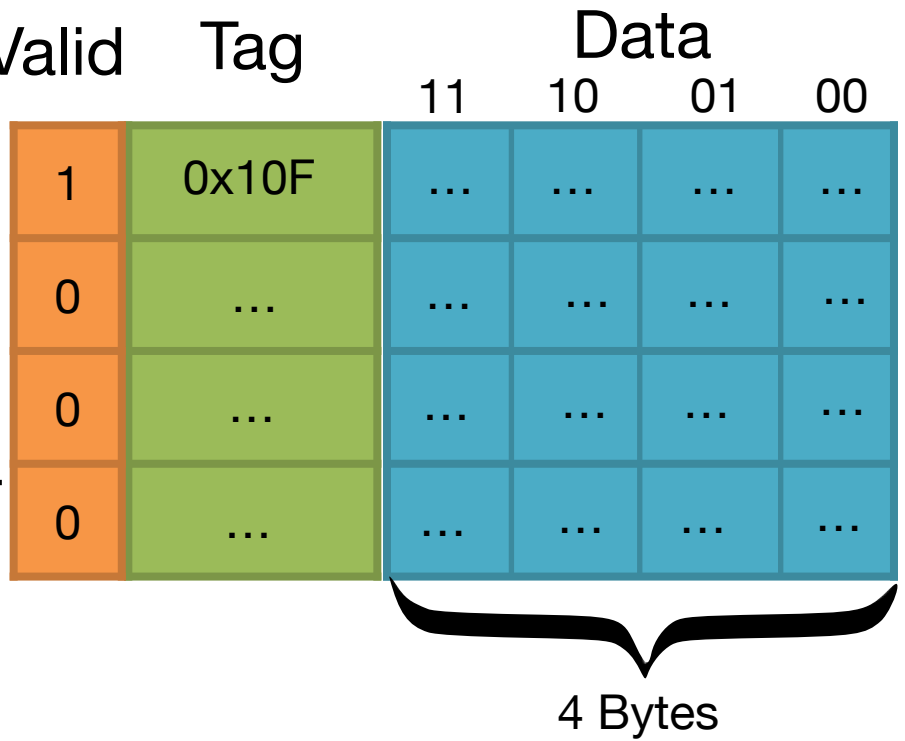
Locality

- Temporal locality
  - The data we access is saved in the cache for potential future use
- Spatial locality
  - We bring in a chunk of data at a time because there is a good chance that we will want to access other data within the chunk

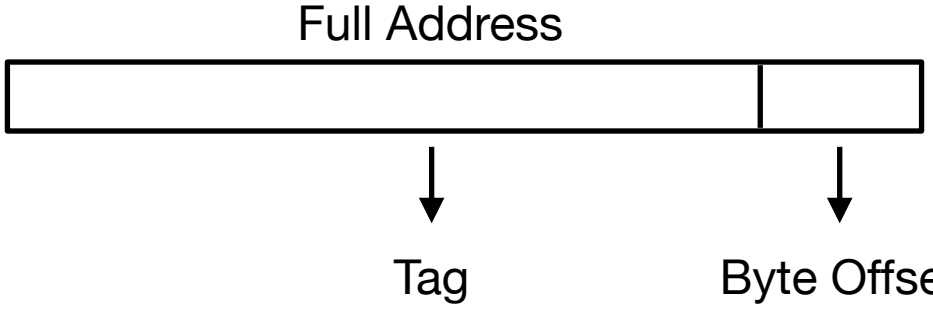
Fully Associative Cache

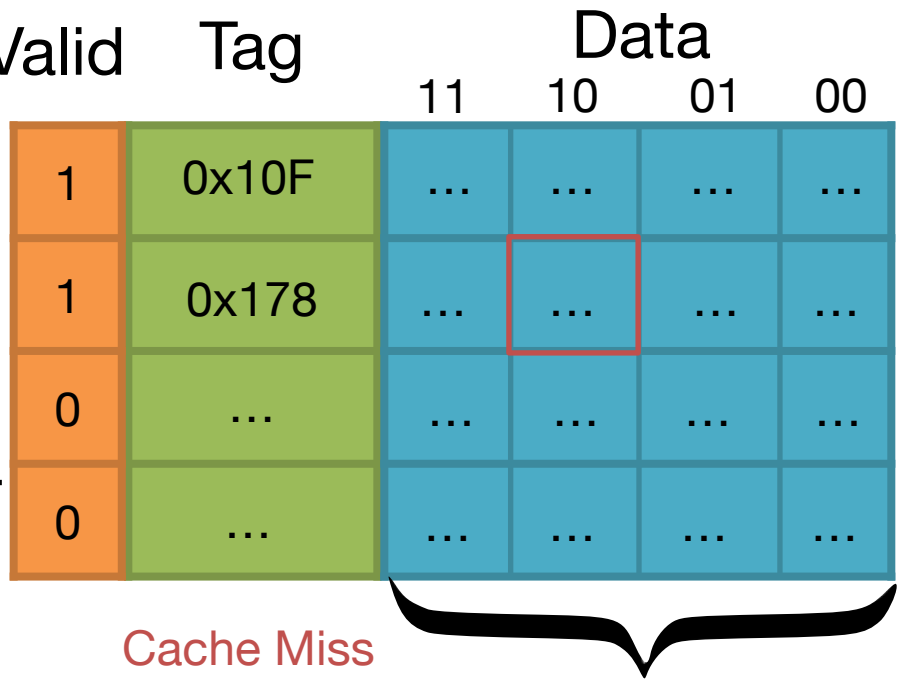
Ex: load byte at 0x5E2

Tag = 10 bits Byte offset = 2 bits



Fully Associative Cache





Ex: load byte at 0x5E2

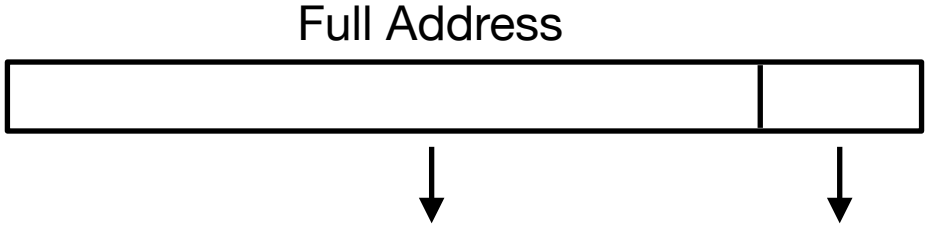
0101 1110 0010

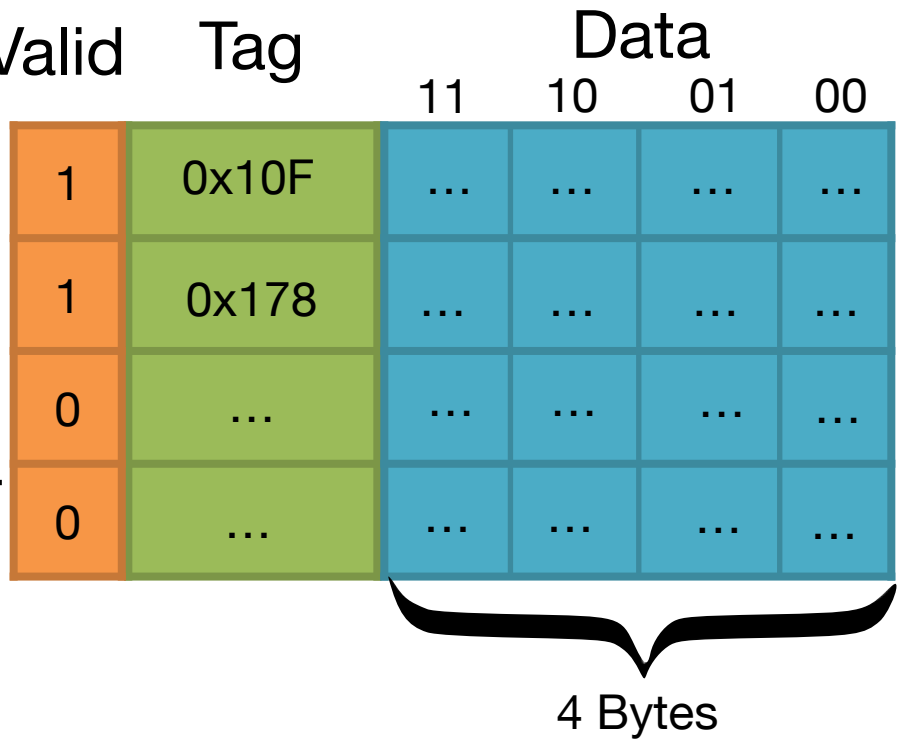
Tag = 10 bits Byte offset = 2 bits

Tag = 0x178 Byte offset = 0x2

We are only reading one byte, but we bring in an entire line (in this case the line is 4 bytes)

Fully Associative Cache

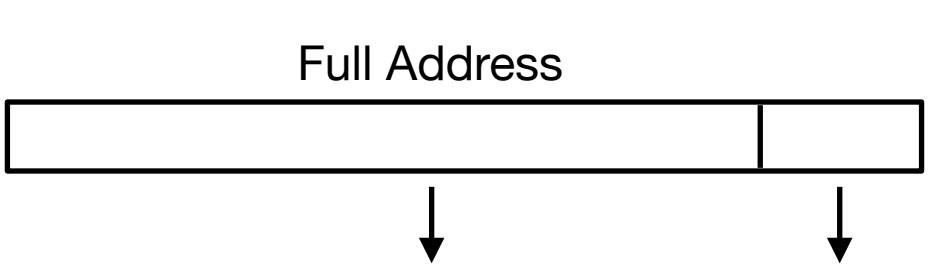


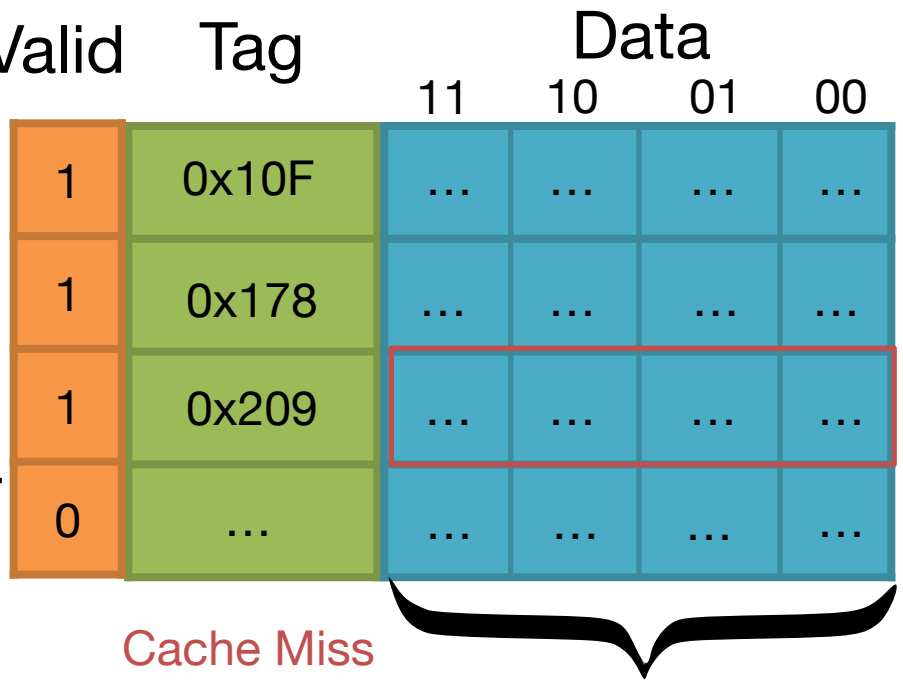


Ex: load word at 0x824

Tag = 10 bits Byte offset = 2 bits

Fully Associative Cache





Ex: load word at 0x824

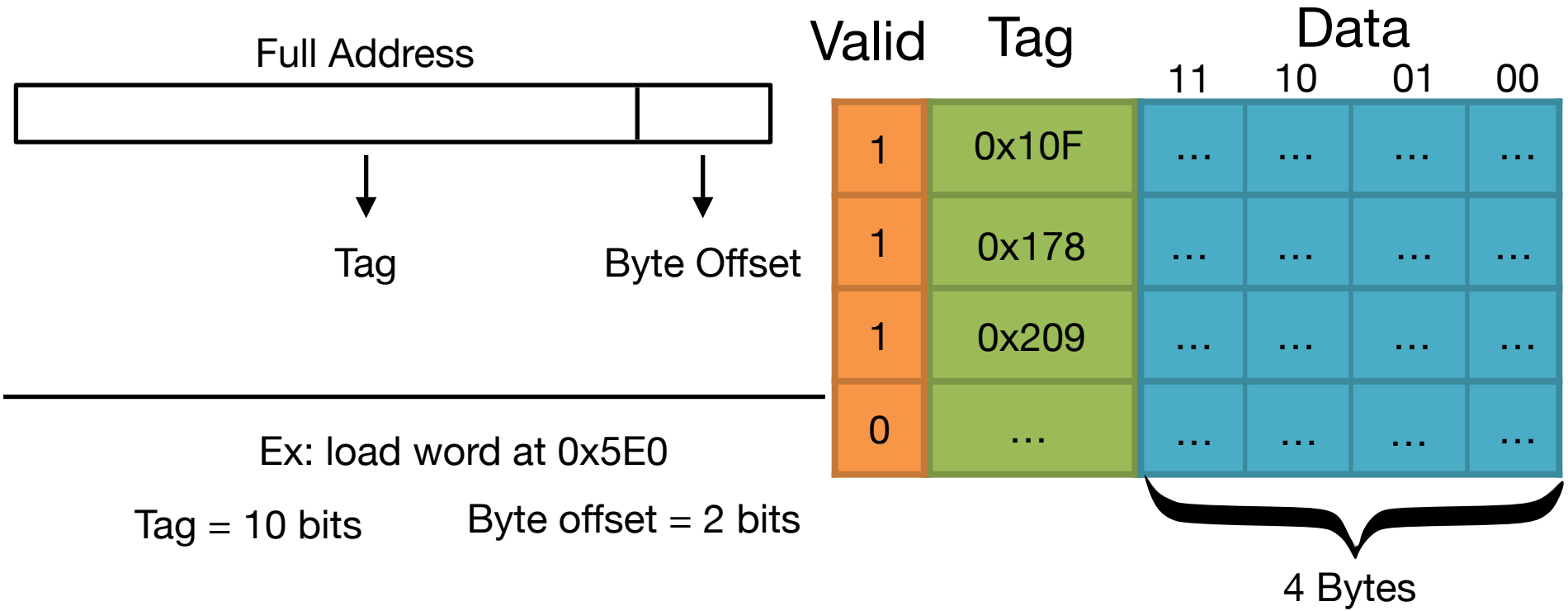
1000 0010 0100

Tag = 10 bits Byte offset = 2 bits

Tag = 0x209 Byte offset = 0x0

We are reading 4 bytes and bringing in one line (4 bytes)

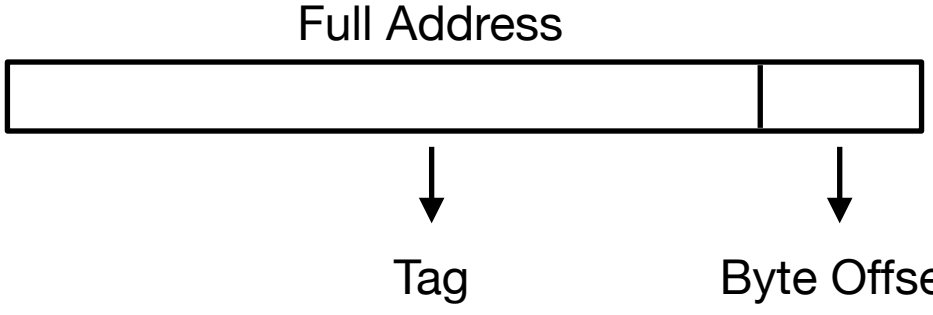
Fully Associative Cache

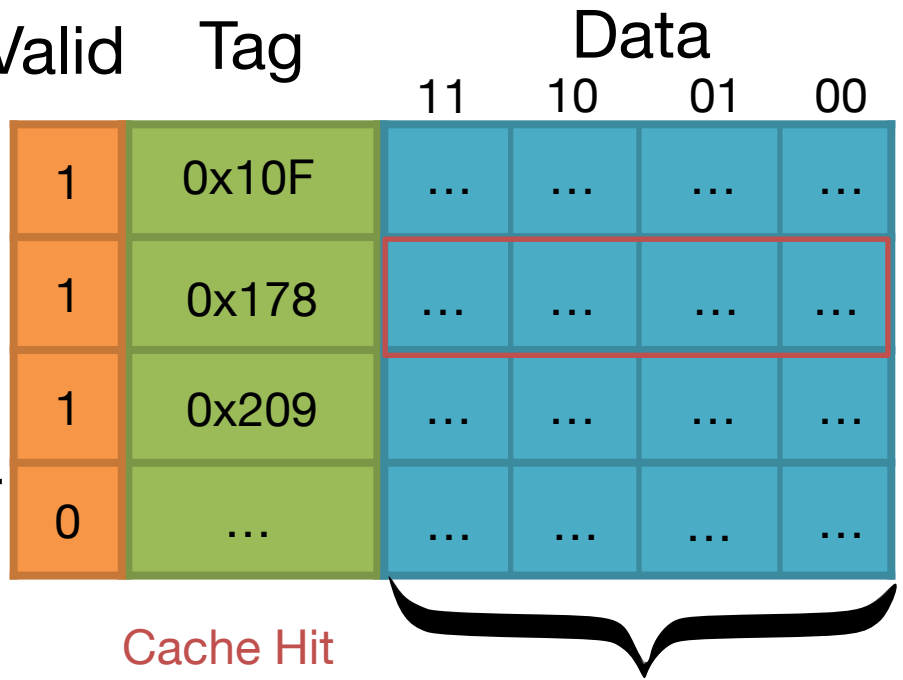


Ex: load word at 0x5E0

Tag = 10 bits Byte offset = 2 bits

Fully Associative Cache





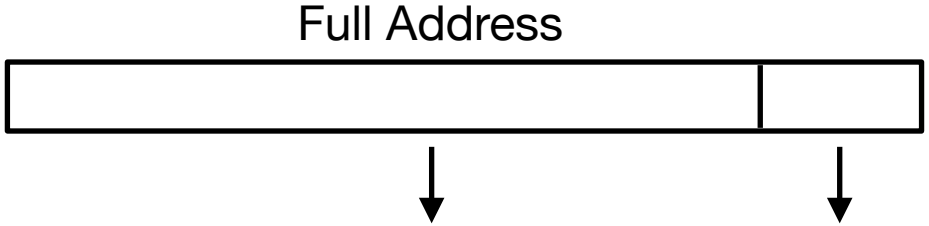
Ex: load word at 0x5E0

0101 1110 0000

Tag = 10 bits Byte offset = 2 bits

Tag = 0x178 Byte offset = 0x0

Fully Associative Cache



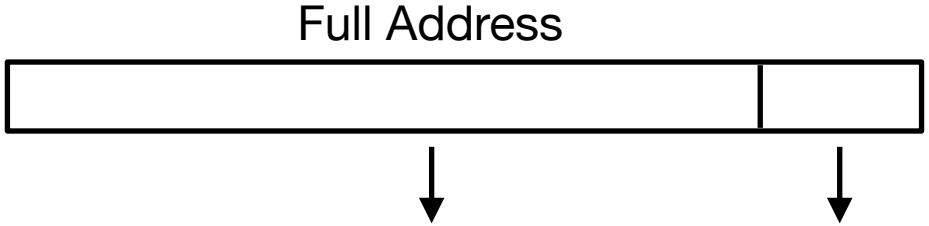
Ex: load word at 0x524

Tag = 10 bits Byte offset = 2 bits

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

4 Bytes

Fully Associative Cache



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

4 Bytes

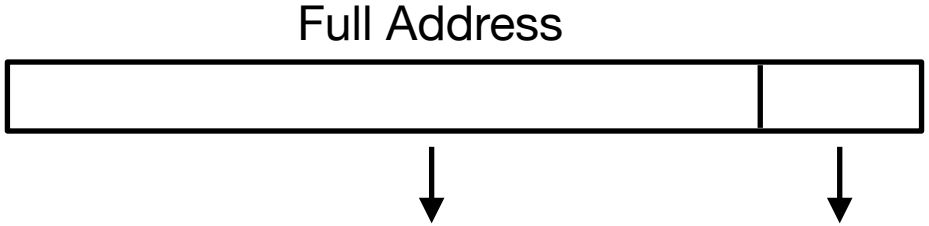
Ex: load word at 0x524

0101 0010 0100

Tag = 10 bits Byte offset = 2 bits

Tag = 0x149 Byte offset = 0x0

Fully Associative Cache



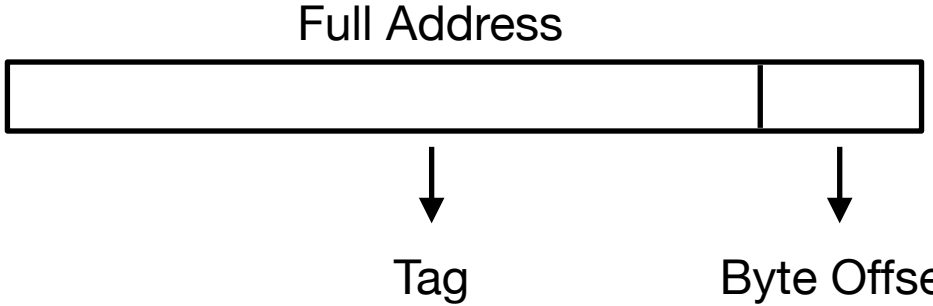
Ex: load byte at 0x972

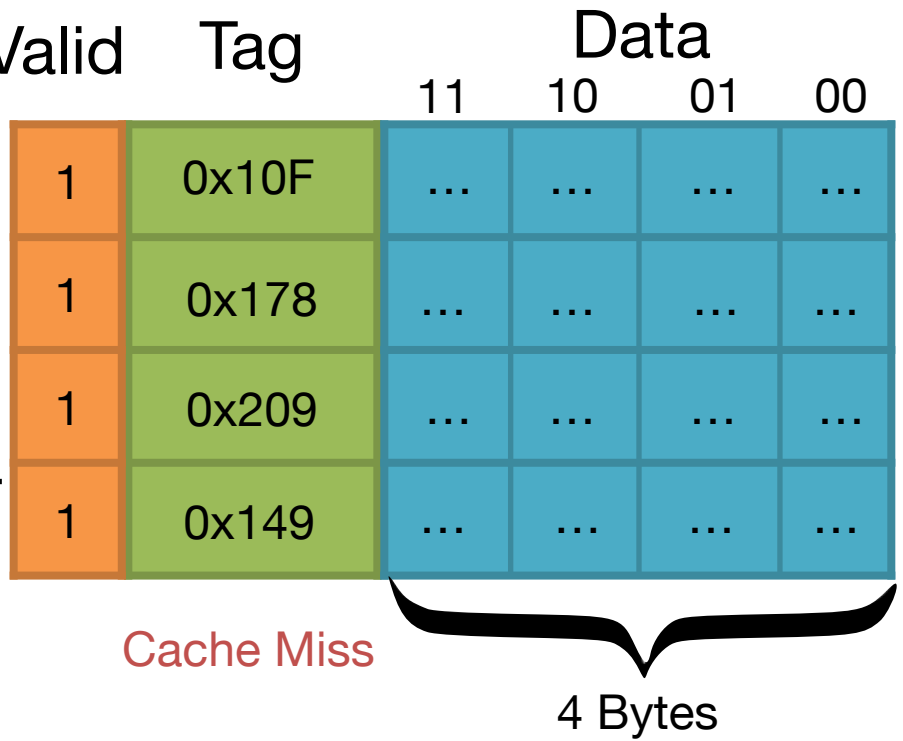
Tag = 10 bits Byte offset = 2 bits

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

4 Bytes

Fully Associative Cache





Ex: load byte at 0x972

1001 0111 0010

Tag = 10 bits Byte offset = 2 bits

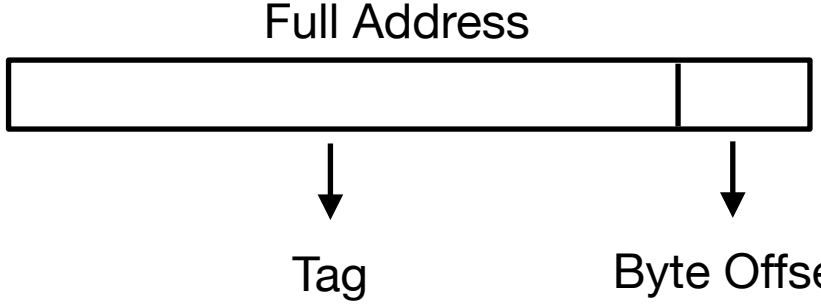
Tag = 0x25C Byte offset = 0x2

Eviction Policies

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

Fully Associative Cache

0 = Most recently used (MRU)
3 = Least recently used (LRU)



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

Ex: load byte at 0x972

1001 0111 0010

Cache Miss

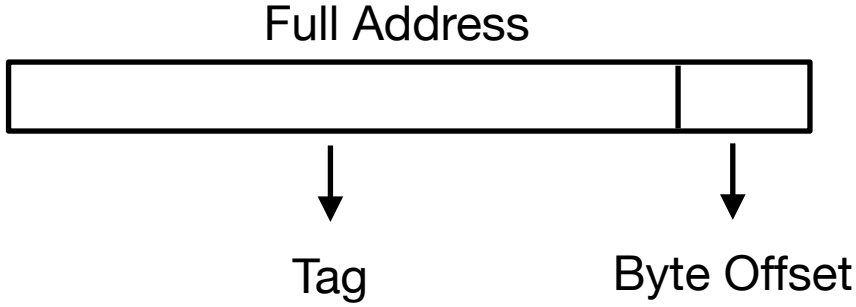
4 Bytes

Tag = 10 bits Byte offset = 2 bits

Tag = 0x25C Byte offset = 0x2

Fully Associative Cache

0 = Most recently used (MRU)
1 = Least recently used (LRU)



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

Ex: load byte at 0x972

1001 0111 0010

Cache Miss

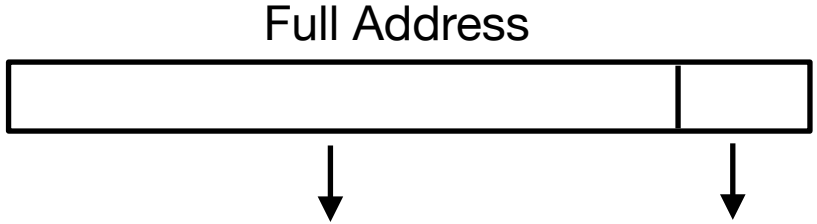
4 Bytes

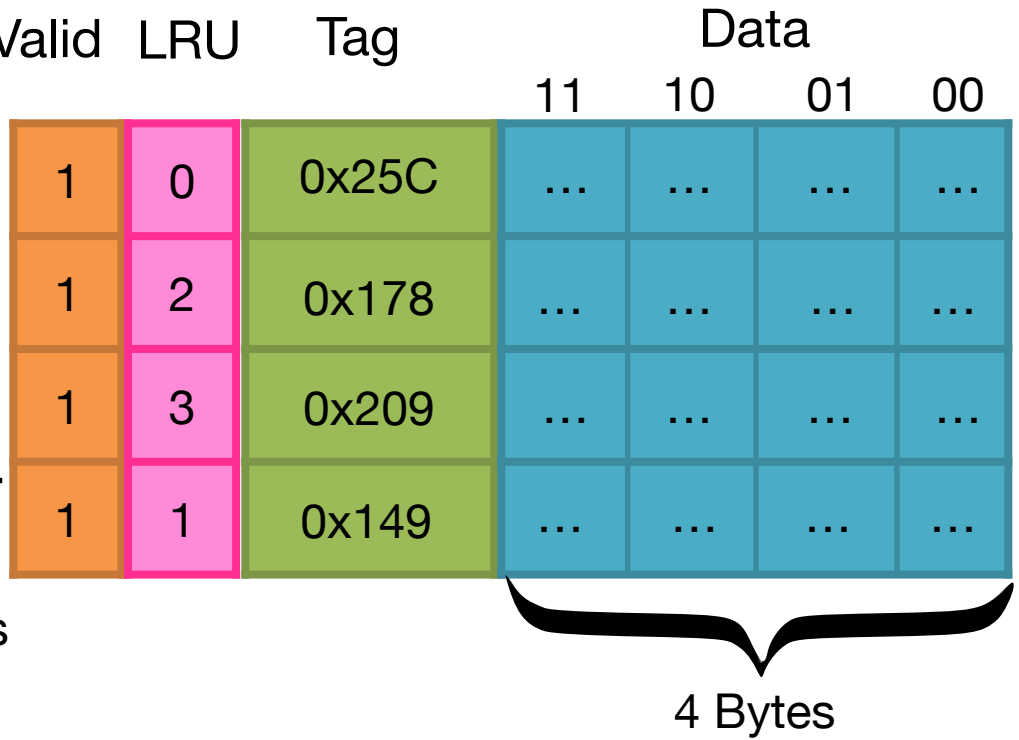
Tag = 10 bits Byte offset = 2 bits

Tag = 0x25C Byte offset = 0x2

Fully Associative Cache

0 = Most recently used (MRU)
1 = Least recently used (LRU)



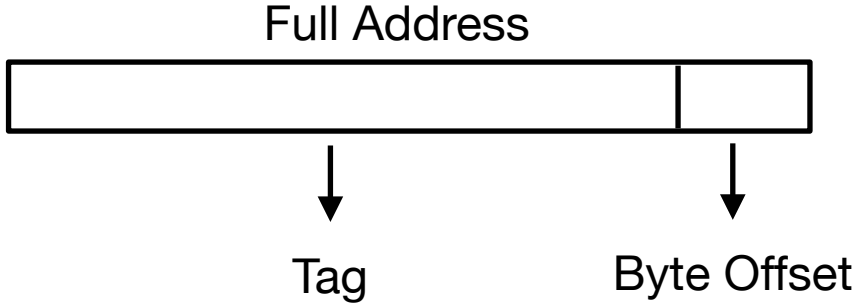


Ex: store byte at 0x524

Tag = 10 bits Byte offset = 2 bits

Fully Associative Cache

0 = Most recently used (MRU)
1 = Least recently used (LRU)



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

Ex: store byte at 0x524

0101 0010 0100

Cache Hit

4 Bytes

Tag = 10 bits Byte offset = 2 bits

Tag = 0x149 Byte offset = 0x0

Handling Stores

- Store instructions write to memory, changing values
- Need to make sure cache and memory have consistent information

Write-through vs Write-back Policies

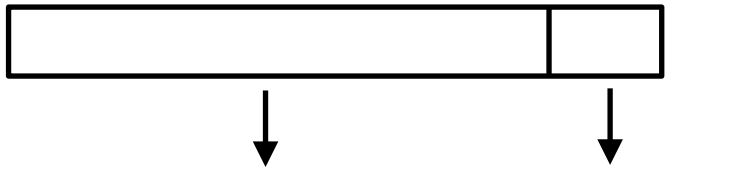
- Write-through
  - Write to the cache and the memory at the same time
  - The write to memory will take longer
- Write-back
  - Write data in cache and set the dirty bit to 1
  - When this line gets evicted from the cache, write it to memory

Write-through vs Write-back Policies

- Write-through
  - Very simple to implement
- Write-back
  - typically lowers traffic to the memory because you might write to something multiple times before you evict it from the cache

Fully Associative Cache (write-back)

0 = Most recently used (MRU)
1 = Least recently used (LRU)



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	…	…	…	…

Ex: store byte at 0x524

0101 0010 0100

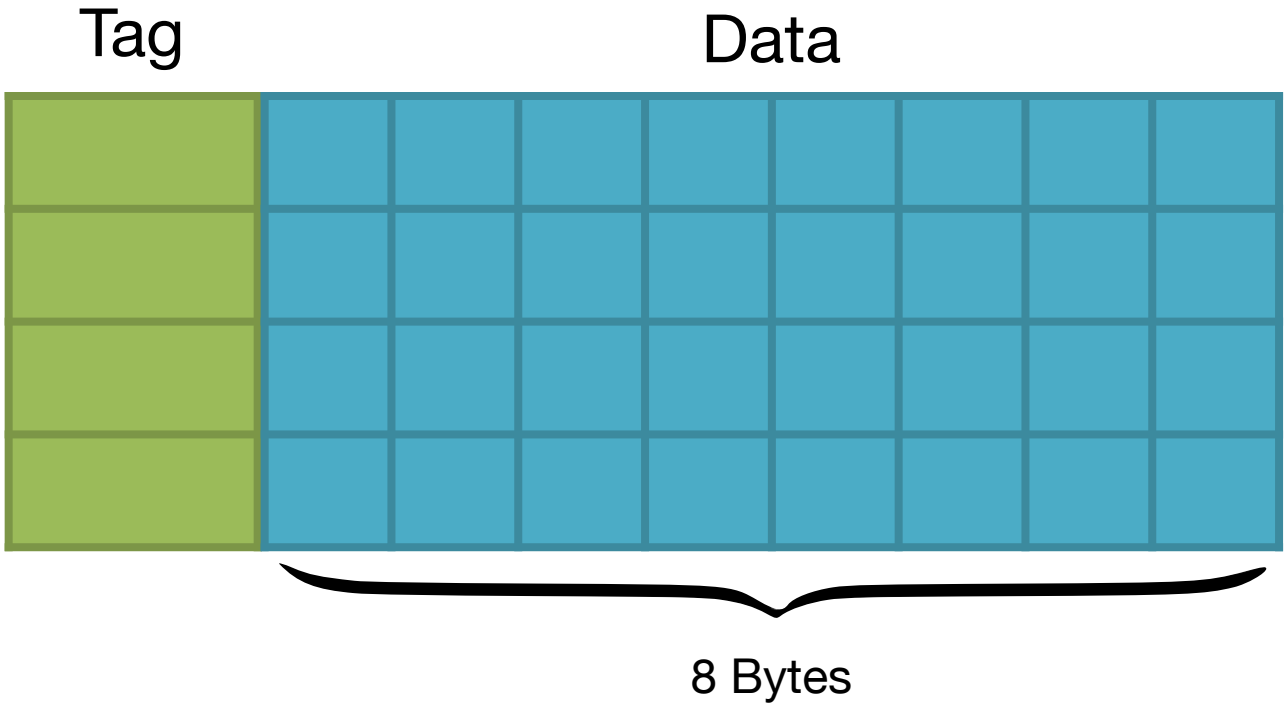
Cache Hit

4 Bytes

Tag = 10 bits Byte offset = 2 bits

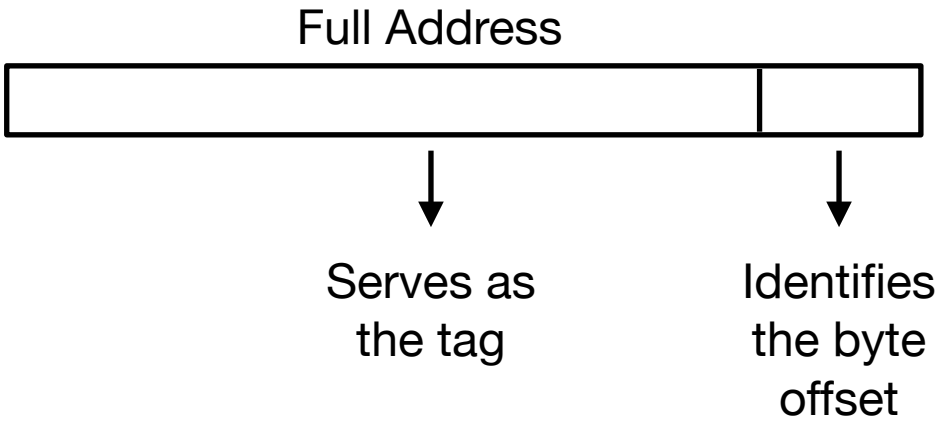
Tag = 0x149 Byte offset = 0x0

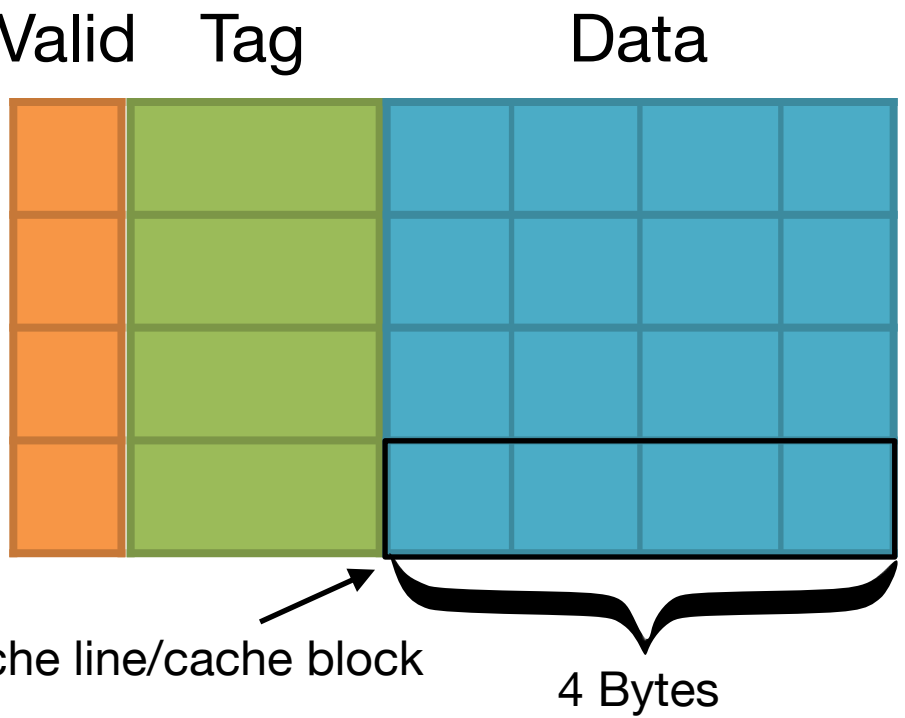
Fully Associative Cache



Hardware Required for Fully Associative Cache

- Need a comparator for each row in the cache to check the tag

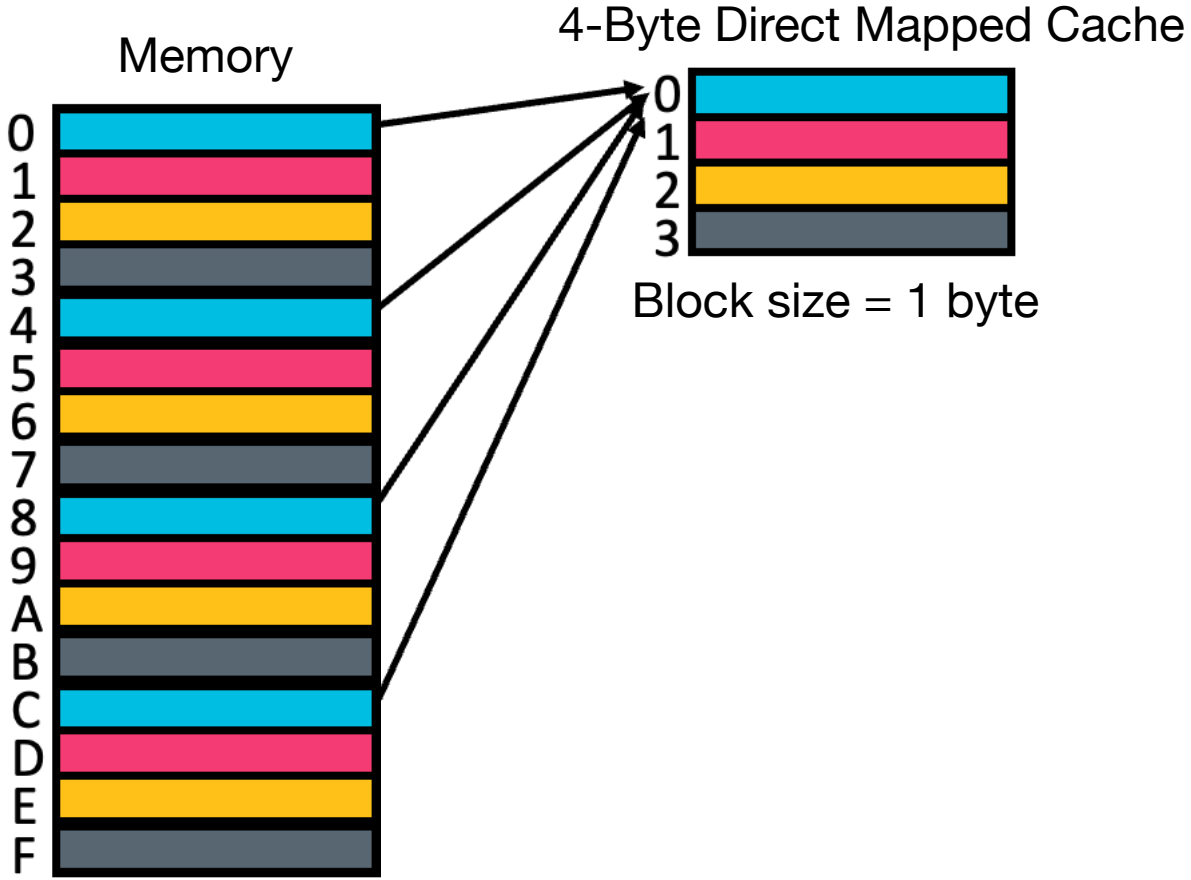




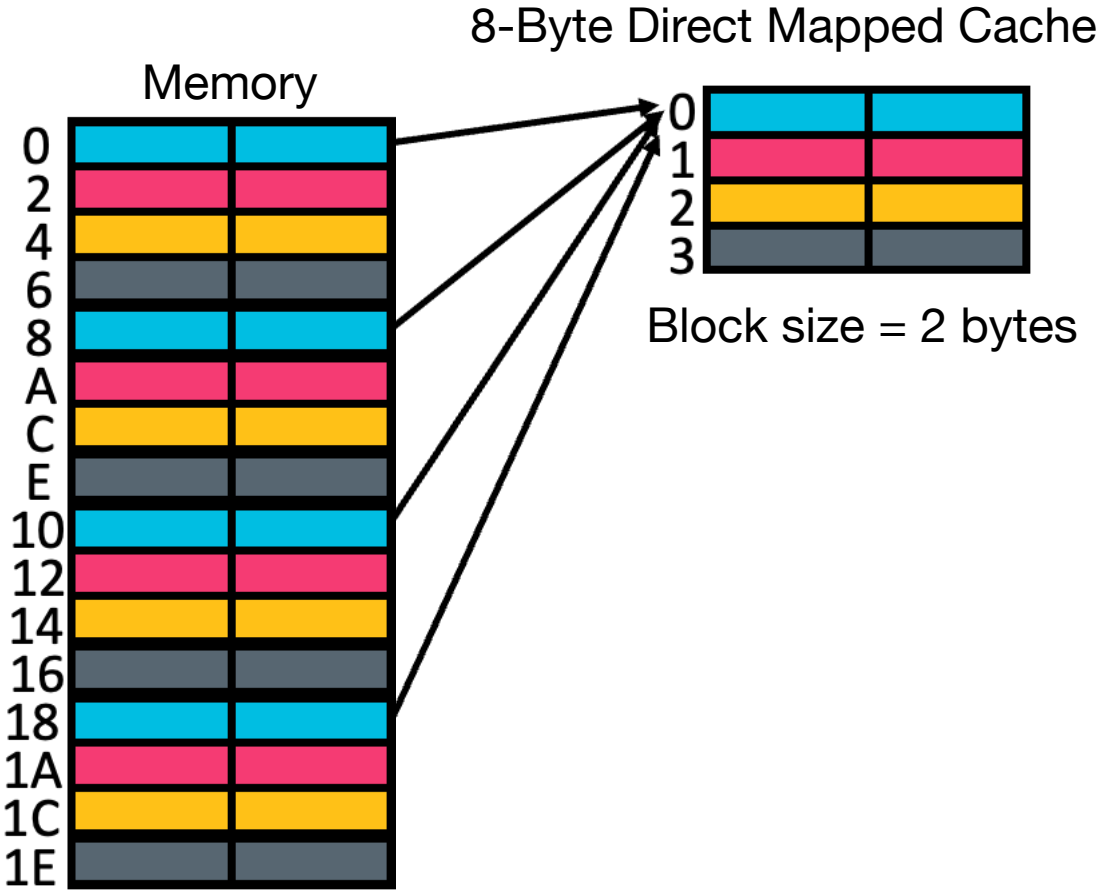
The data can be stored anywhere in the cache

Direct Mapped Caches

Direct Mapped Cache



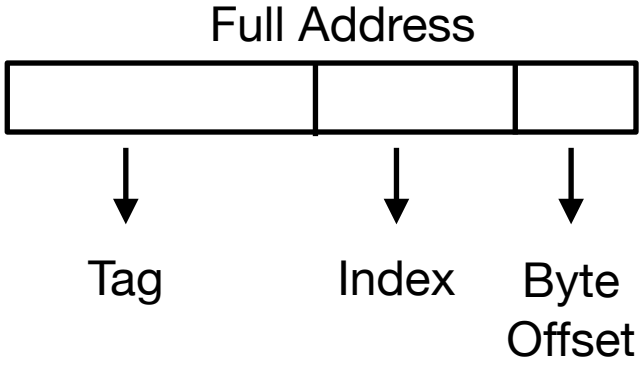
Direct Mapped Cache



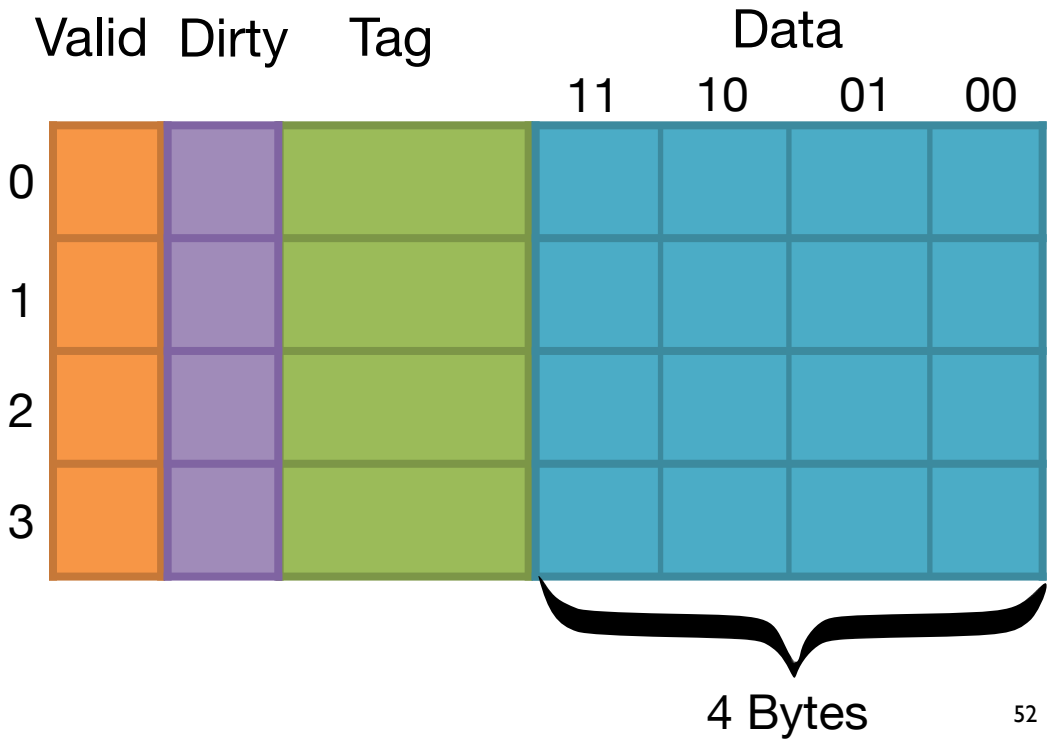
Note that this memory and cache are twice as large as the previous slide

Direct Mapped (write-back)

- The data can only be stored in one location

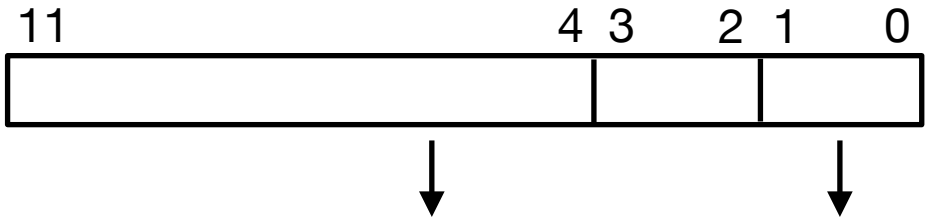


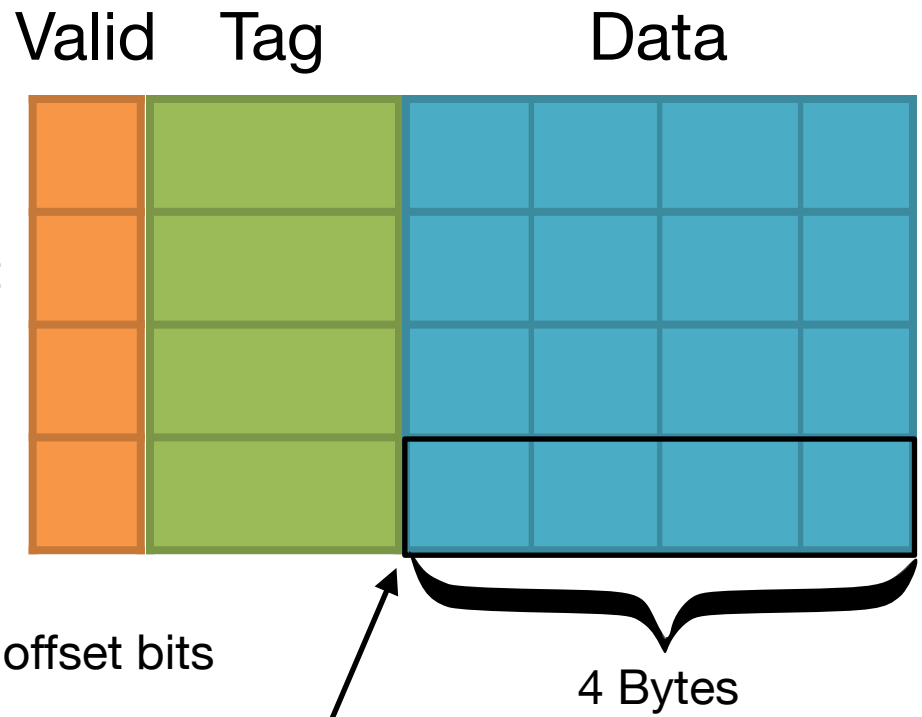
TIO



Direct Mapped Cache Address Breakdown

Full Address: ex 12 bits



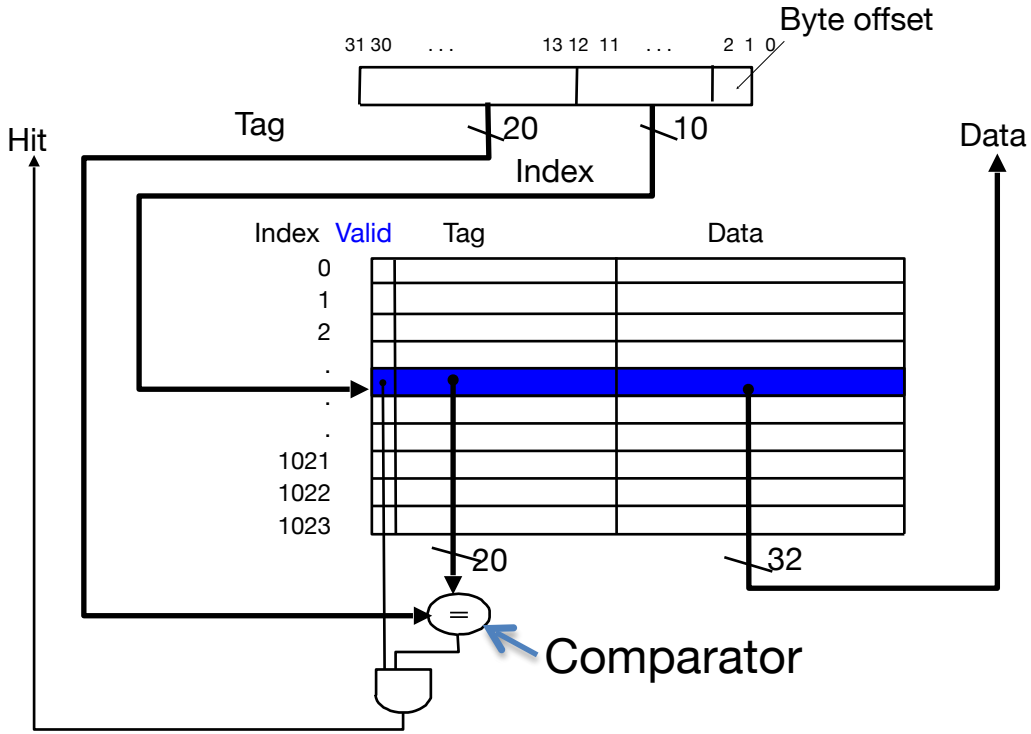


byte offset bits = $\log_2$(line size)
= $\log_2(4) = 2$

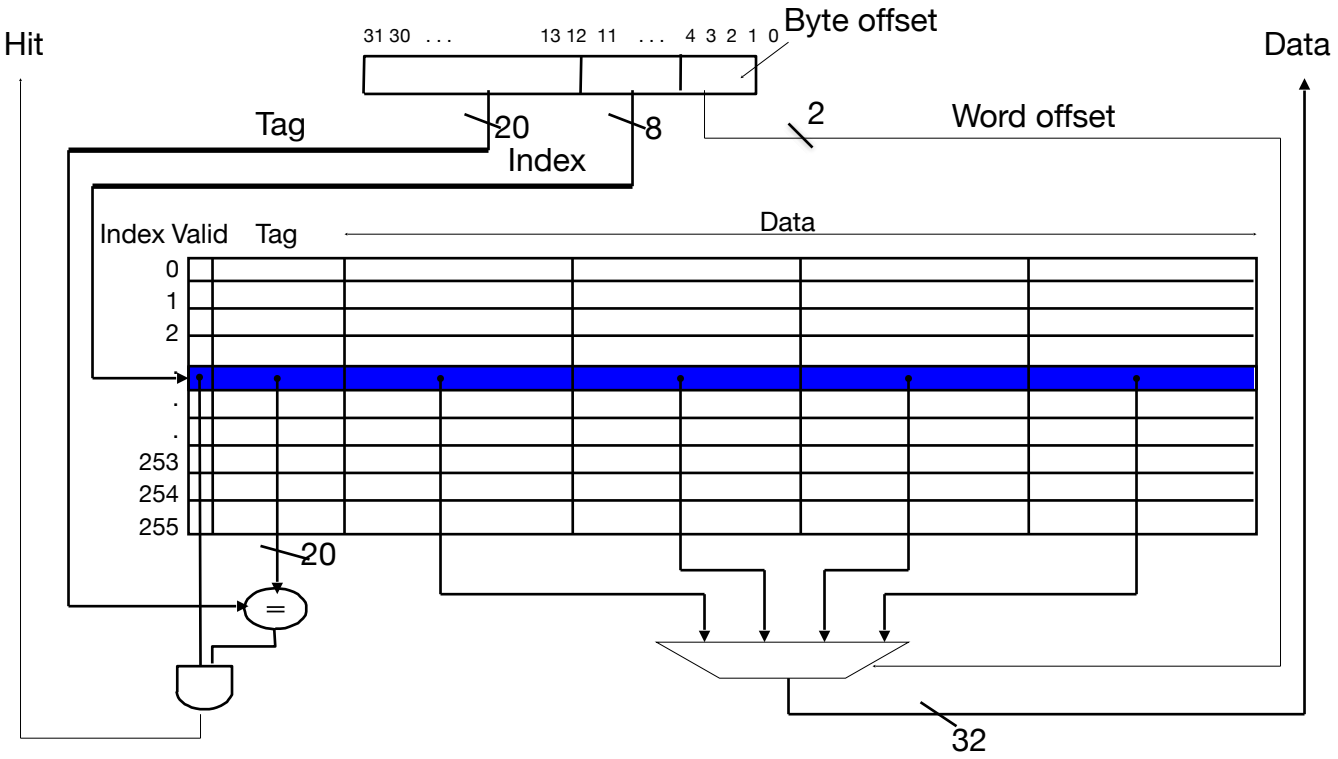
index bits = $\log_2$(# lines)
= $\log_2(4) = 2$

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

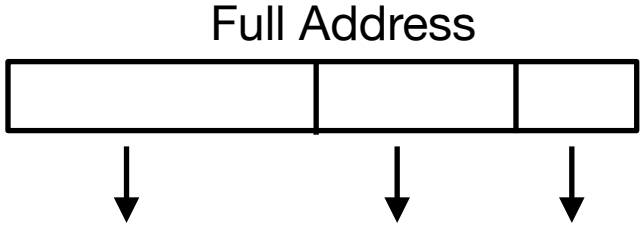
Direct-Mapped Cache Hardware: 1 word blocks, 4KB data



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

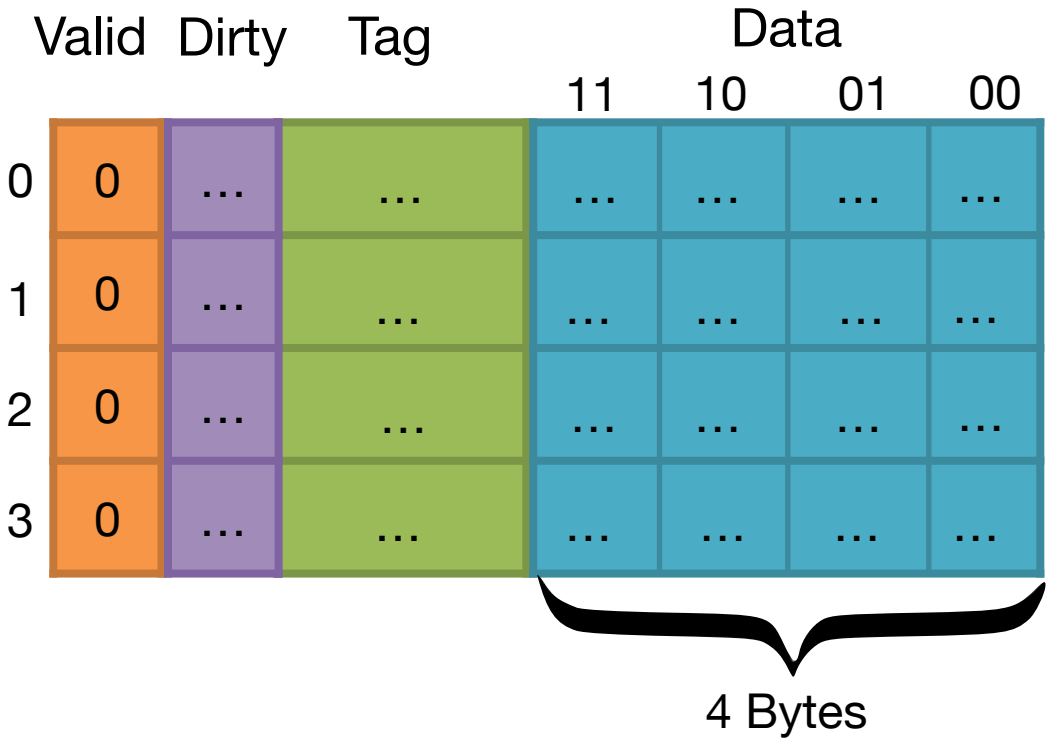


Direct Mapped

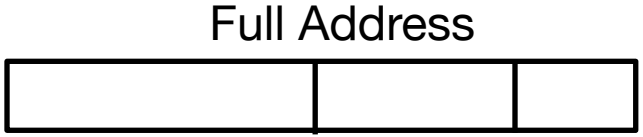


Read byte 0xFE2

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



Direct Mapped



Read byte 0xFE2

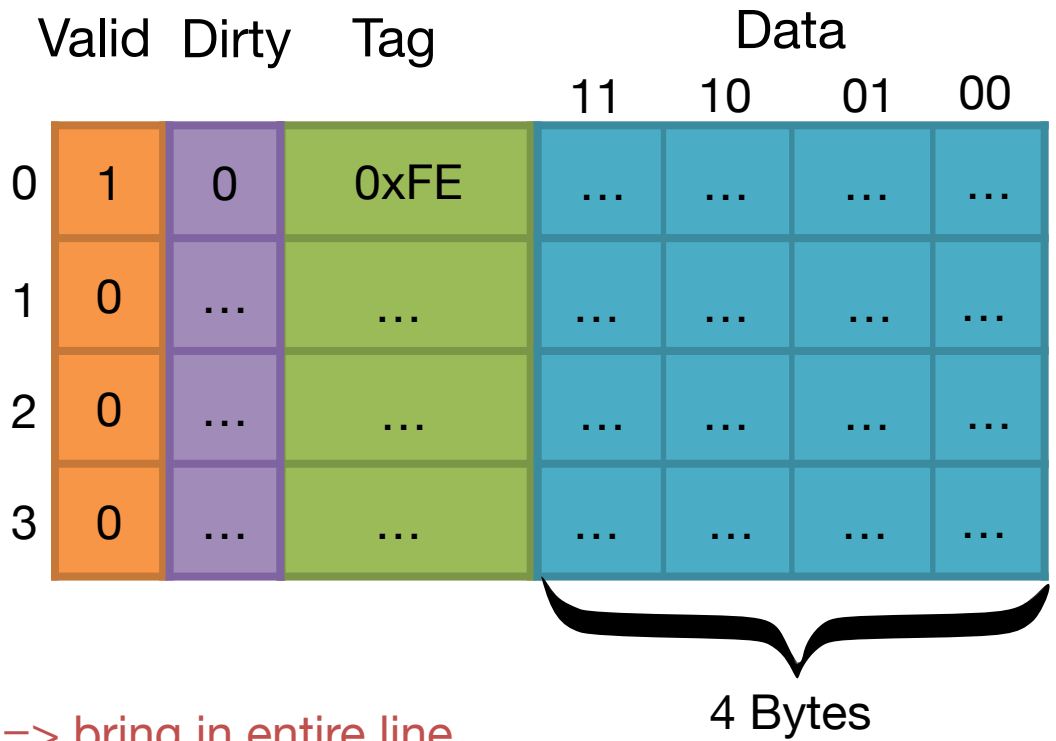
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 0010

T = 0xFE

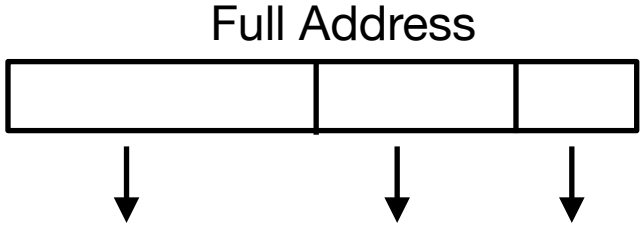
I = 0x0

O = 0x2



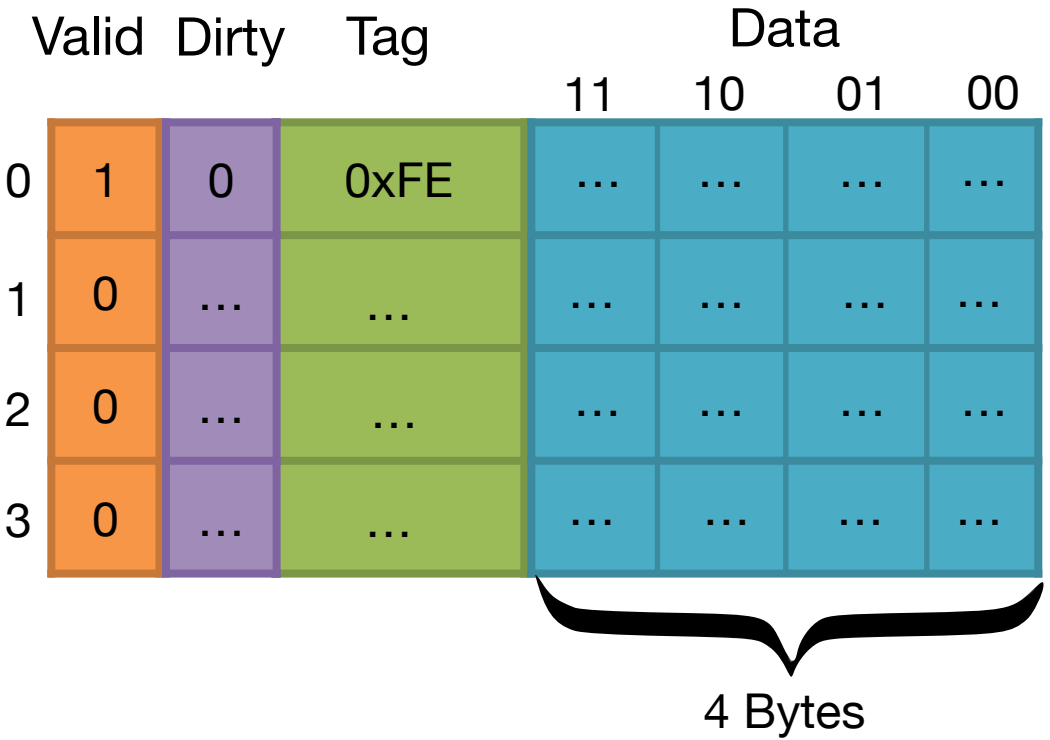
Cache Miss => bring in entire line

Direct Mapped

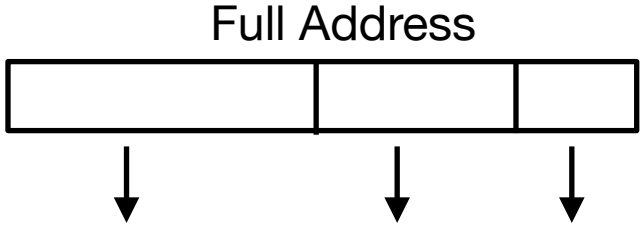


Read byte 0xFE8

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



Direct Mapped



Read byte 0xFE8

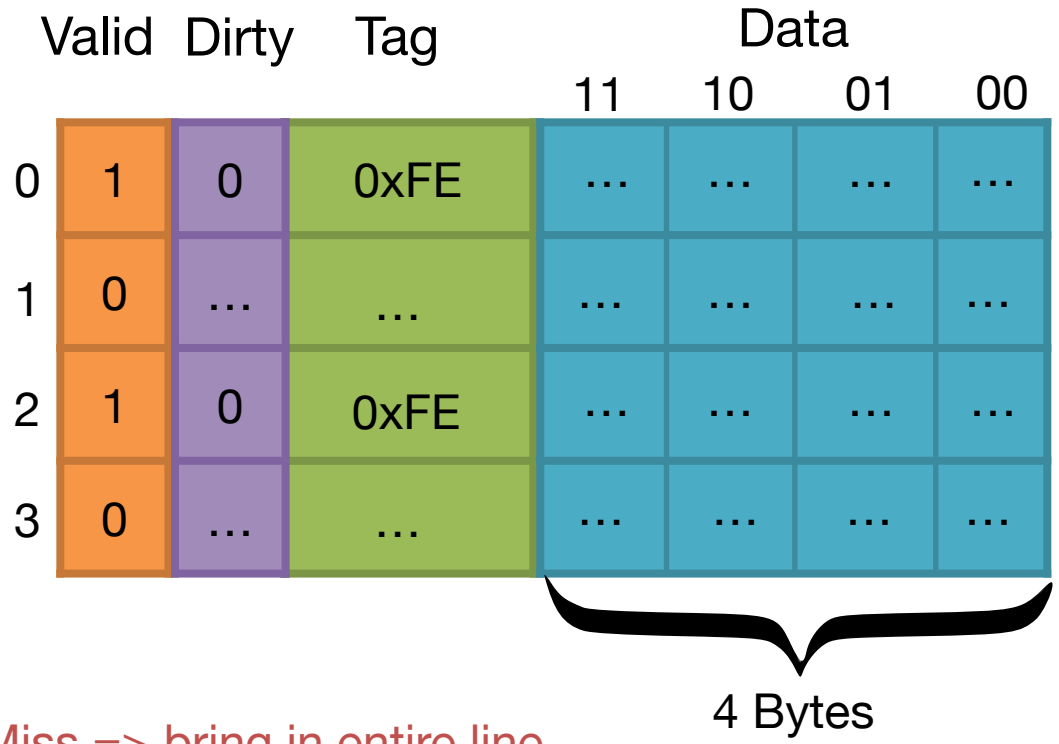
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 1000

T = 0xFE

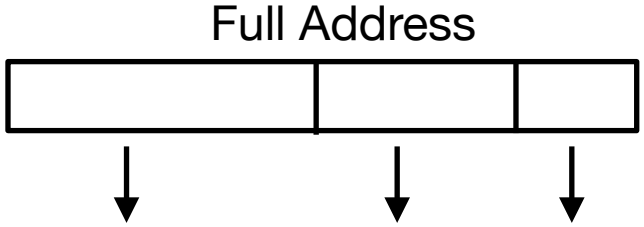
I = 0x2

O = 0x0



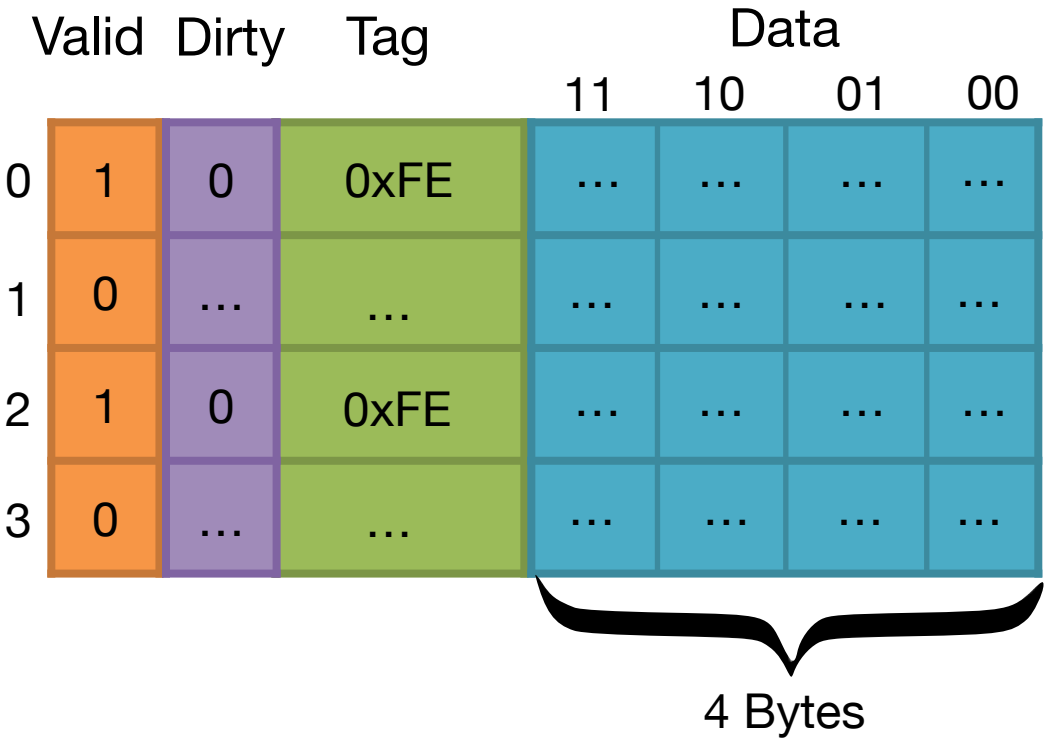
Cache Miss => bring in entire line

Direct Mapped

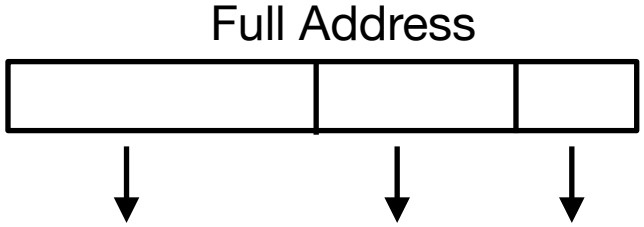


Read byte 0xFE9

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



Direct Mapped



Read byte 0xFE9

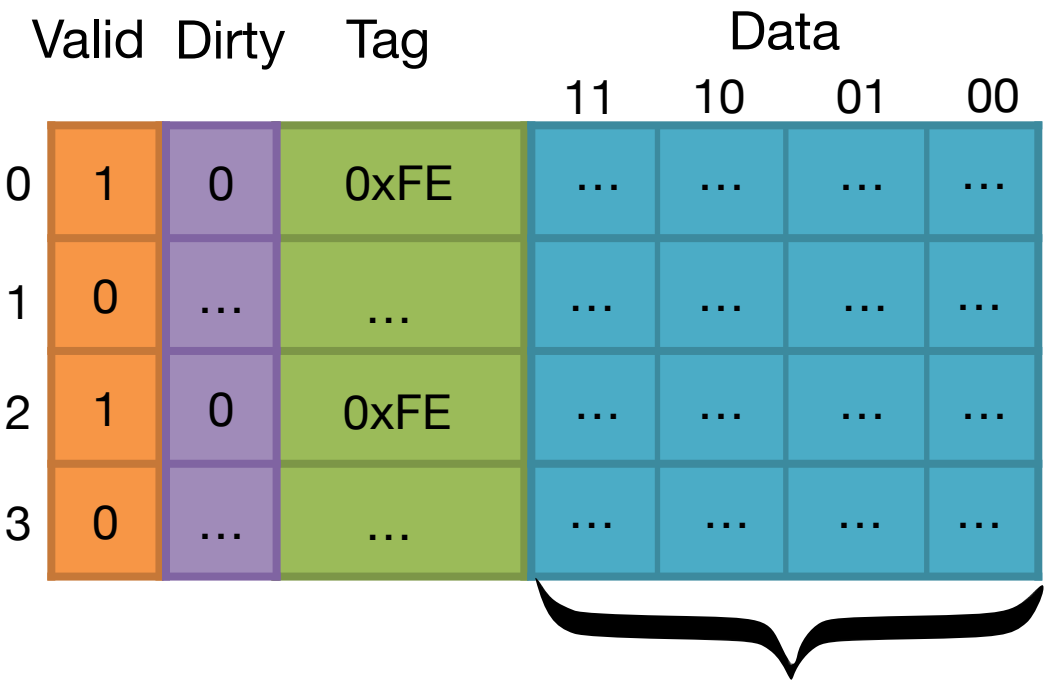
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 1001

T = 0xFE

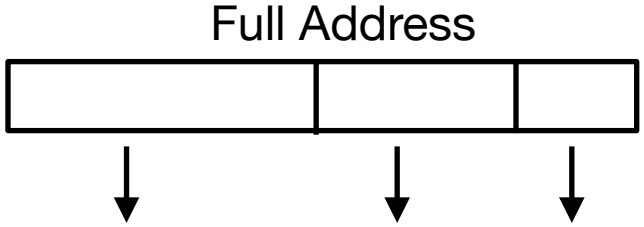
I = 0x2

O = 0x1



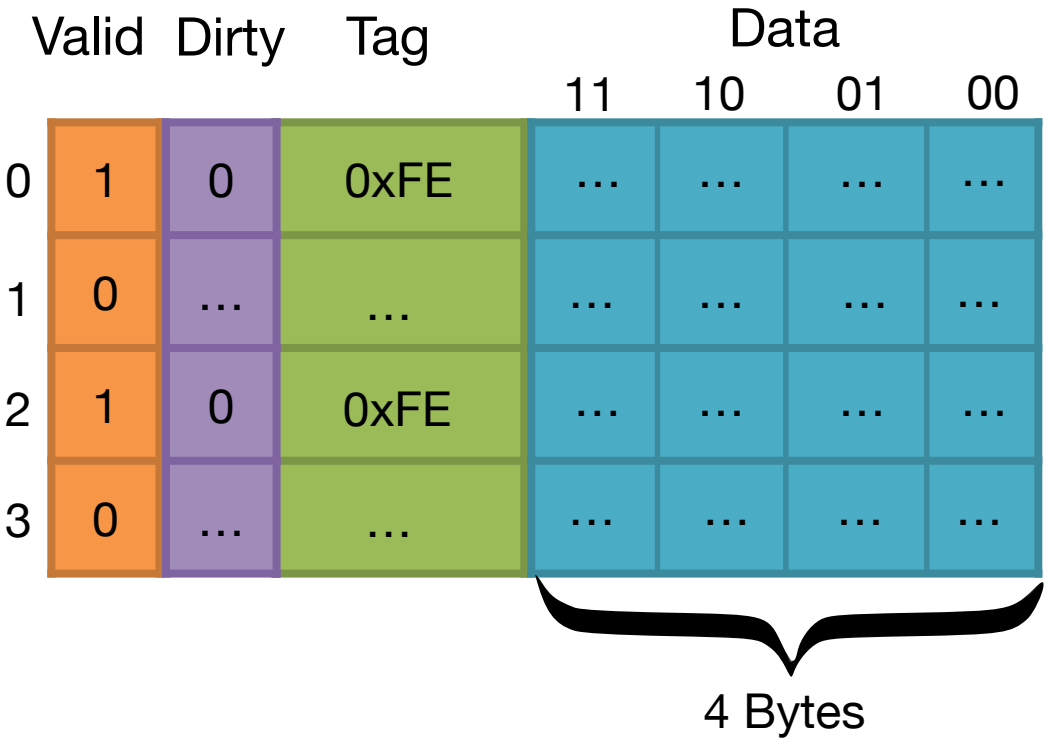
Cache Hit!

Direct Mapped

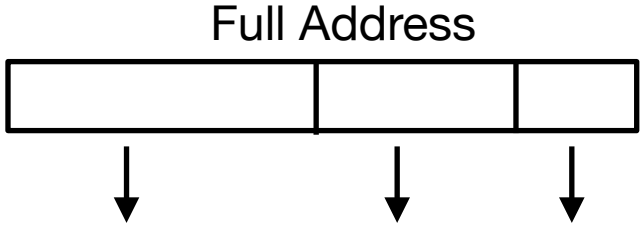


Read byte 0xDF9

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



Direct Mapped



Read byte 0xDF9

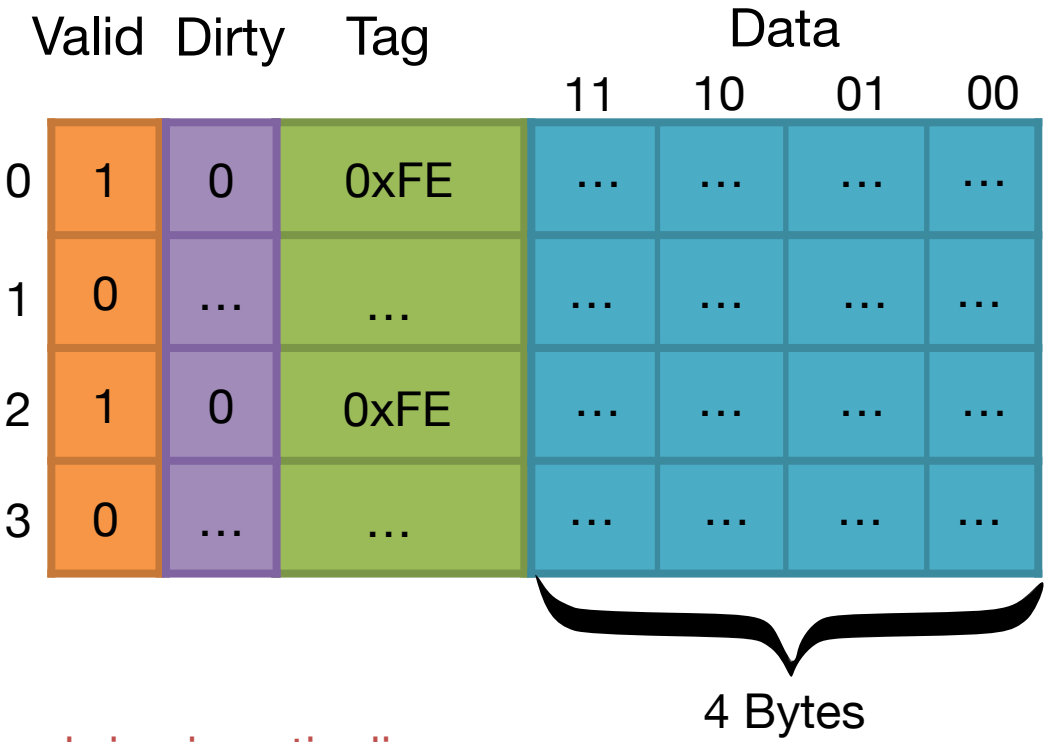
T => 8 bits I => 2 bits O => 2 bits

0b 1101 1111 1001

T = 0xDF

I = 0x2

O = 0x1

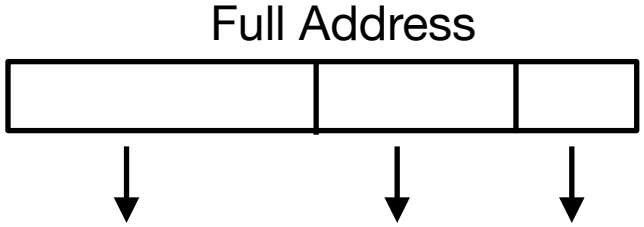


Cache Miss => bring in entire line

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



Read byte 0xDF9

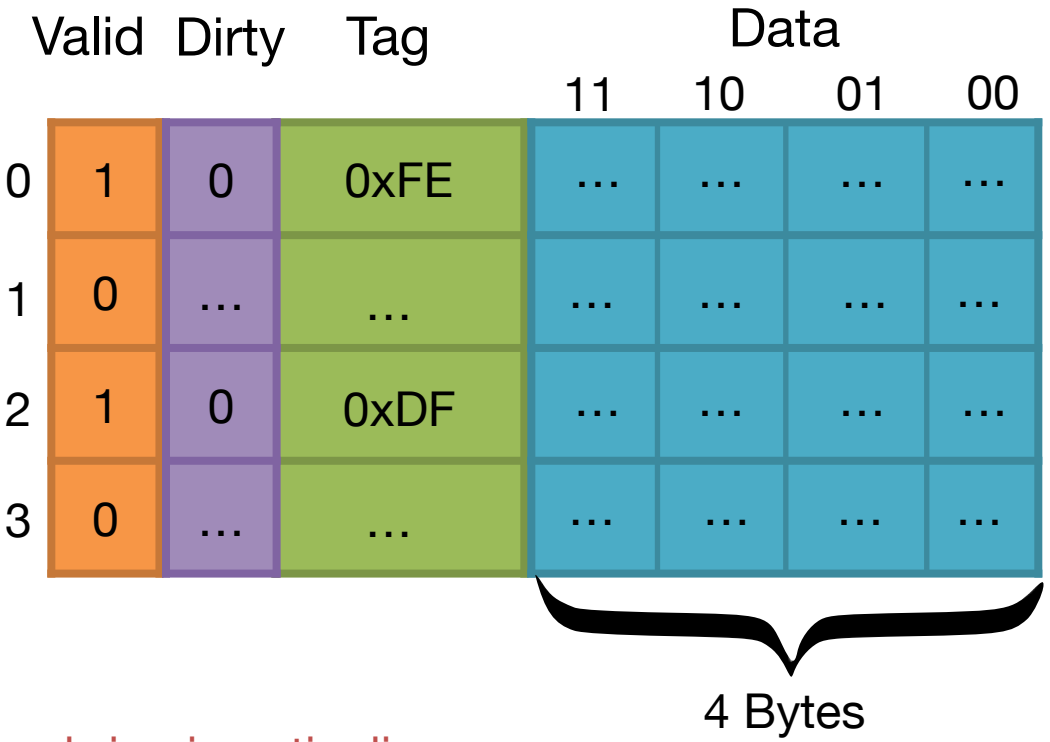
T => 8 bits I => 2 bits O => 2 bits

0b 1101 1111 1001

T = 0xDF

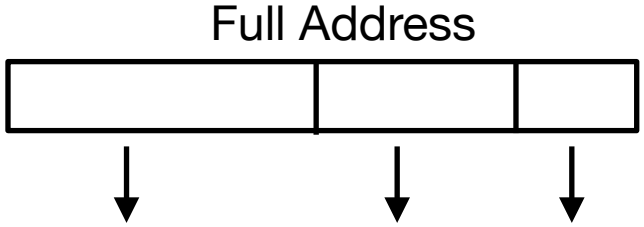
I = 0x2

O = 0x1



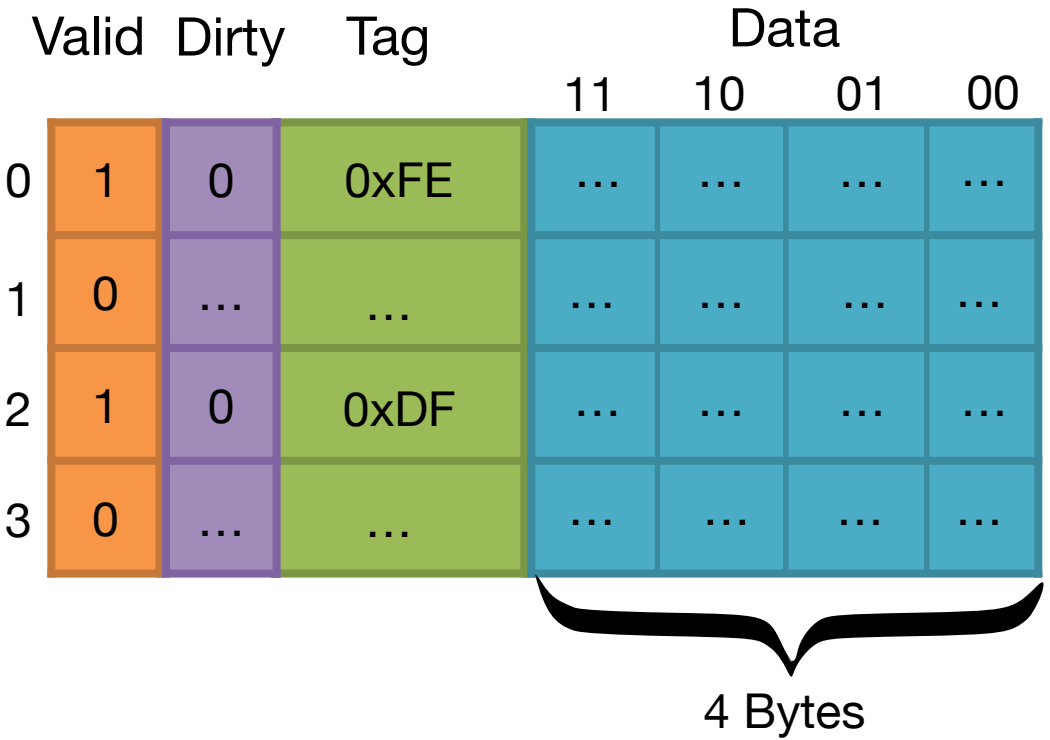
Cache Miss => bring in entire line

Direct Mapped

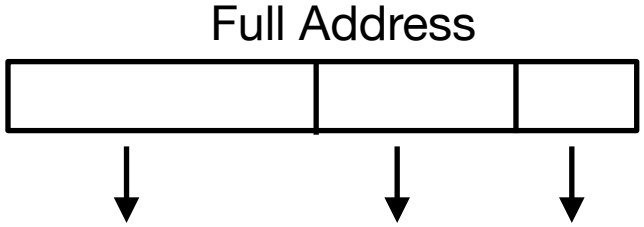


Read byte 0xFE8

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



Direct Mapped



Read byte 0xFE8

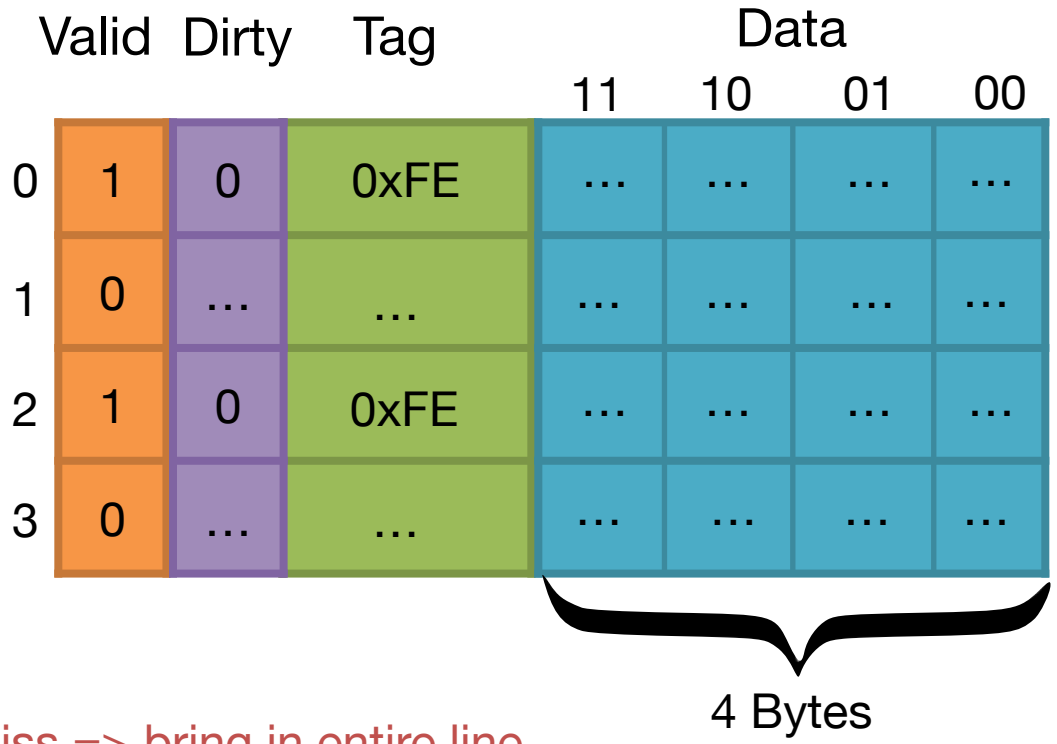
T => 8 bits I => 2 bits O => 2 bits

0b 1111 1110 1000

T = 0xFE

I = 0x2

O = 0x0



Cache Miss => bring in entire line

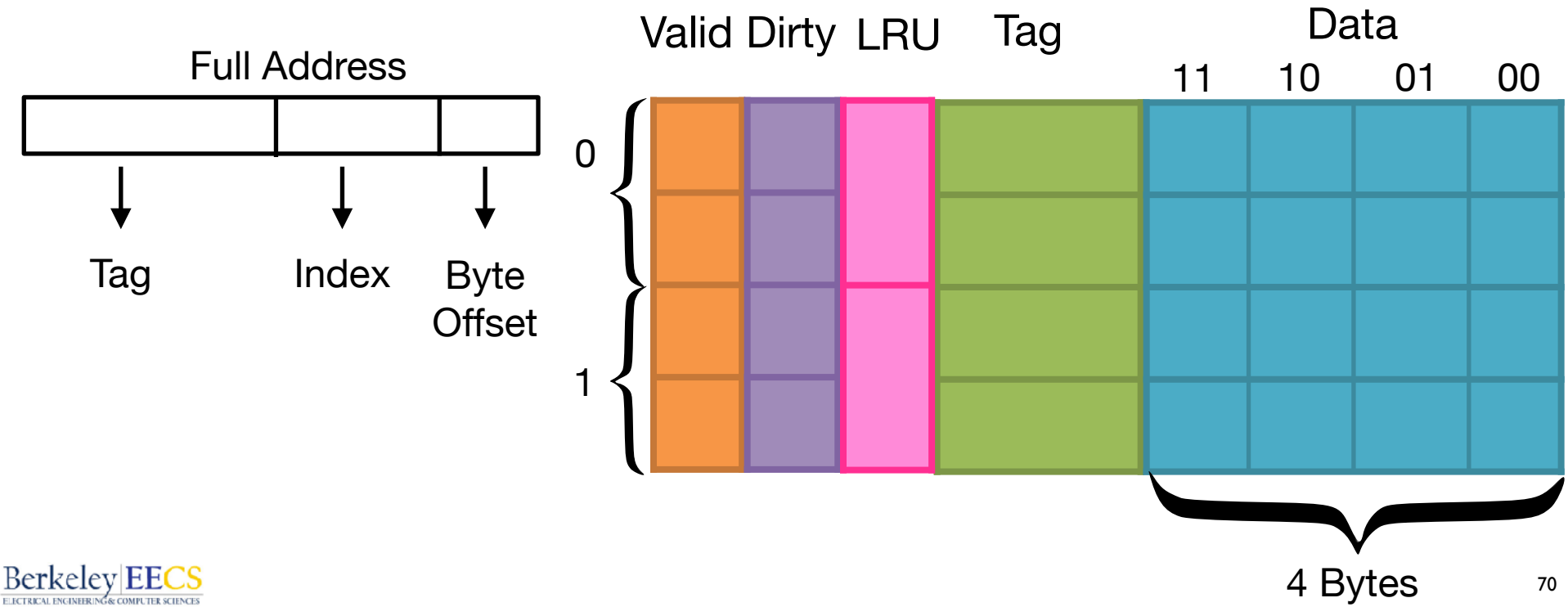
Direct Mapped

- 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 (write-back)

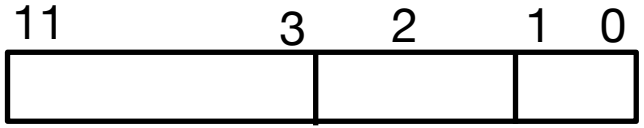
- The data can only be stored at one index, but there are multiple slots to store it in

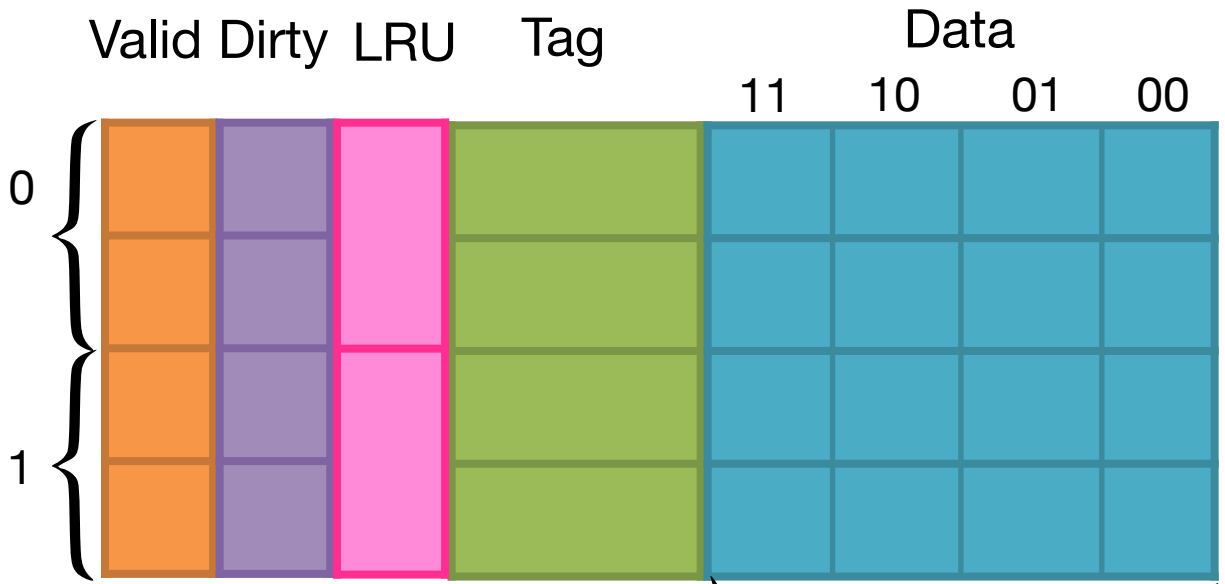


Set Associative (write-back)

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

Full Address: ex: 12 bits



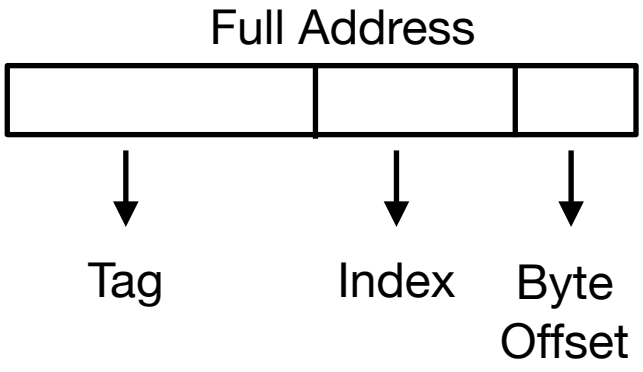


byte offset bits = $\log_2$(line size)
= $\log_2(4) = 2$

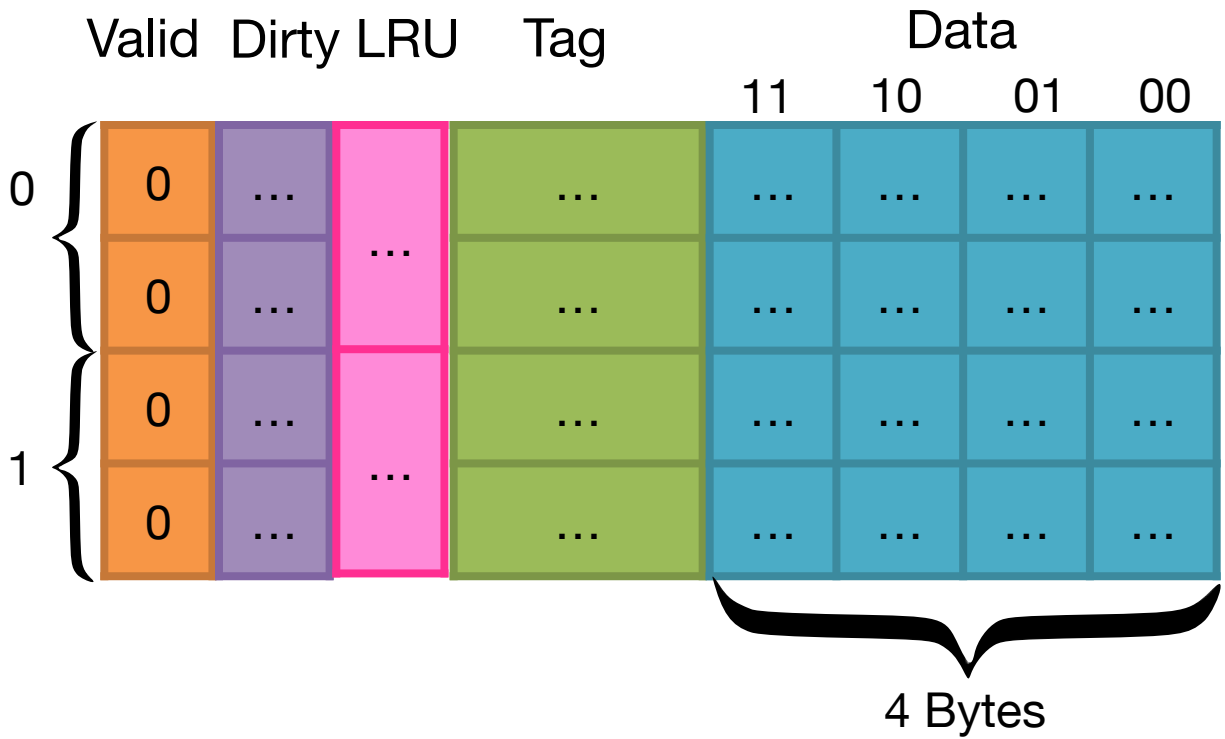
index bits = $\log_2$(# sets)
= $\log_2(2) = 1$

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

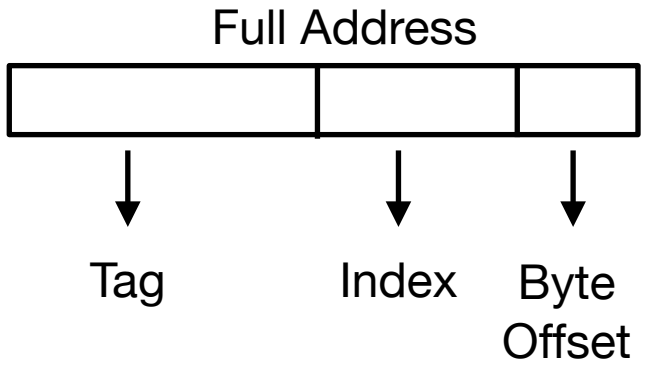
Set Associative (write-back)



Read Byte 0x8E2



Set Associative (write-back)



Read Byte 0x8E2

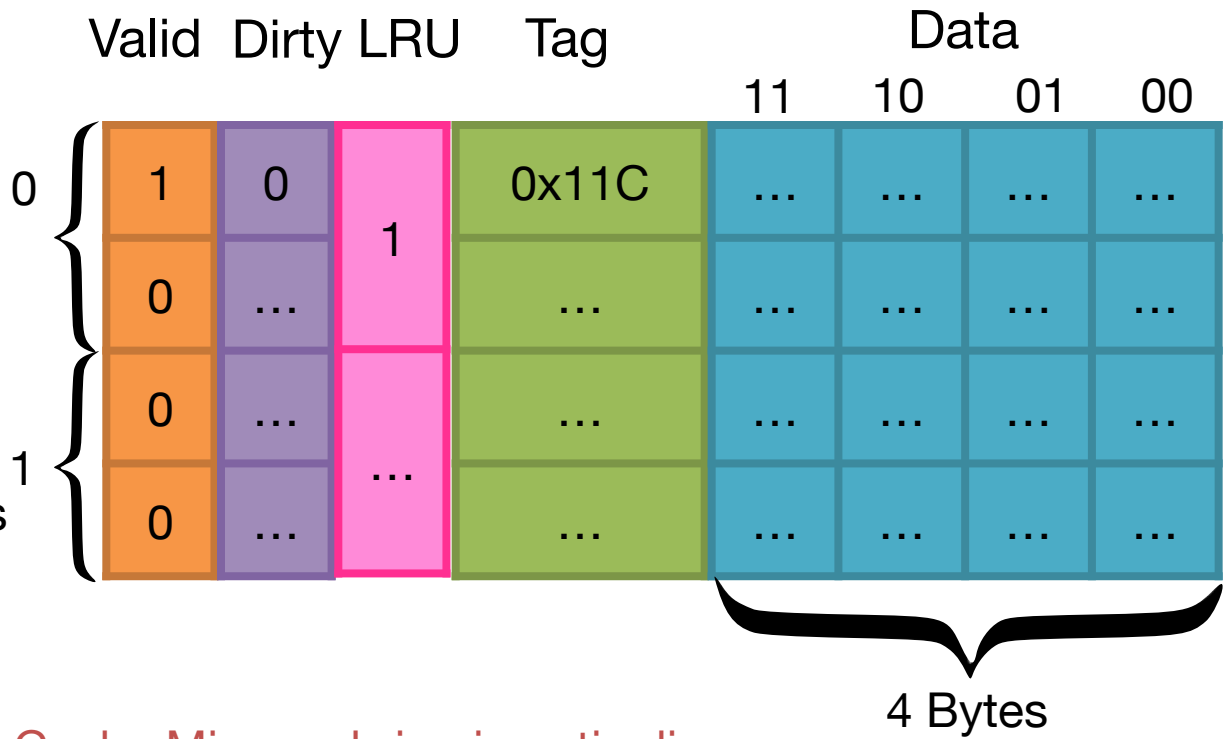
T = 9 bits I = 1 bit O = 2 bits

0b 1000 1110 0010

T = 0x11C

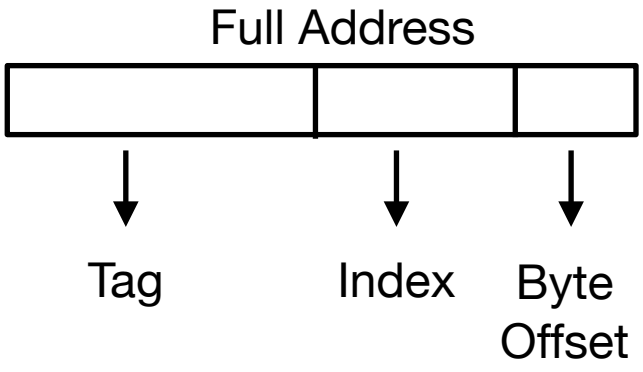
I = 0x0

O = 0x2

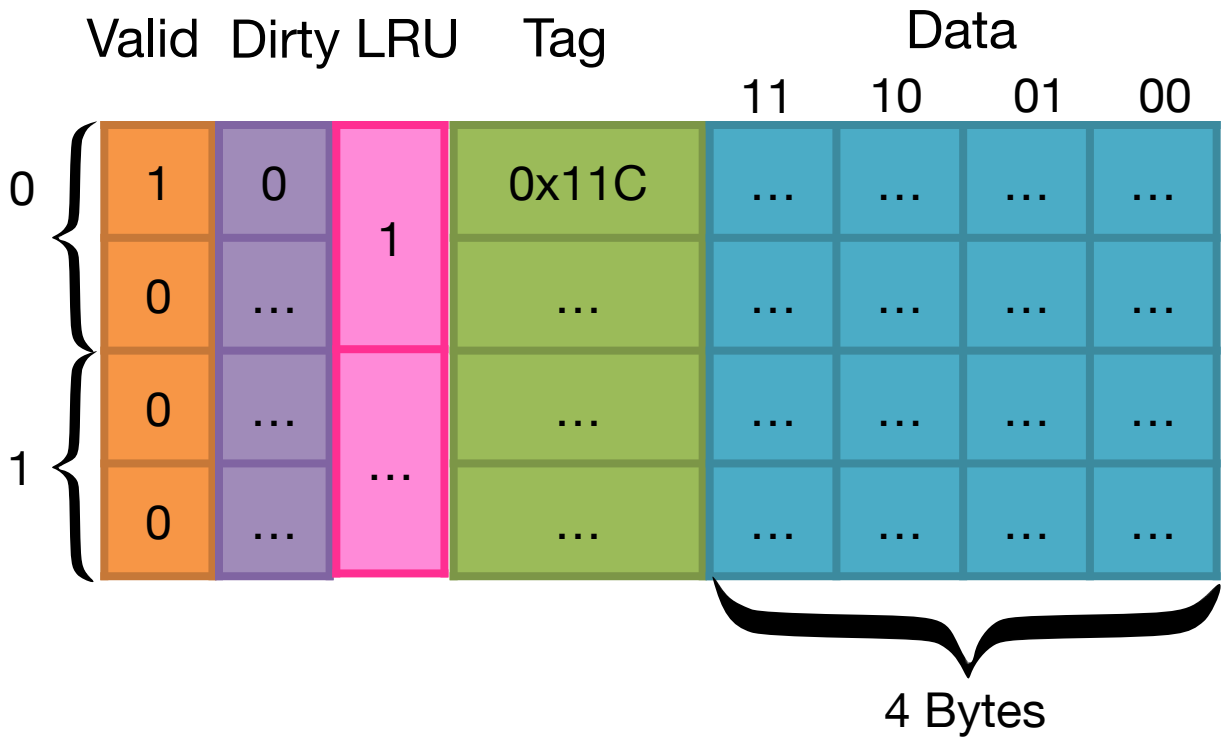


Cache Miss => bring in entire line

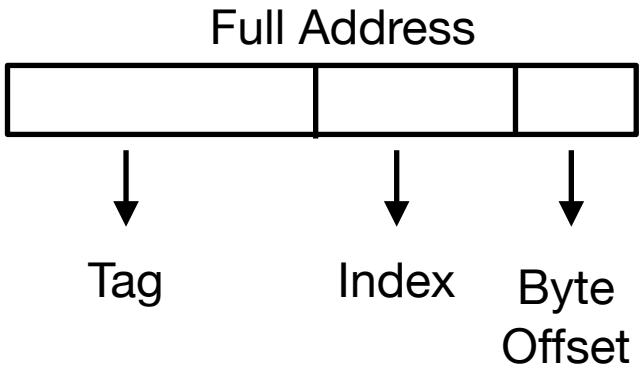
Set Associative (write-back)



Read Byte 0x8E8



Set Associative (write-back)



Read Byte 0x8E8

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

0b 1000 1110 1000

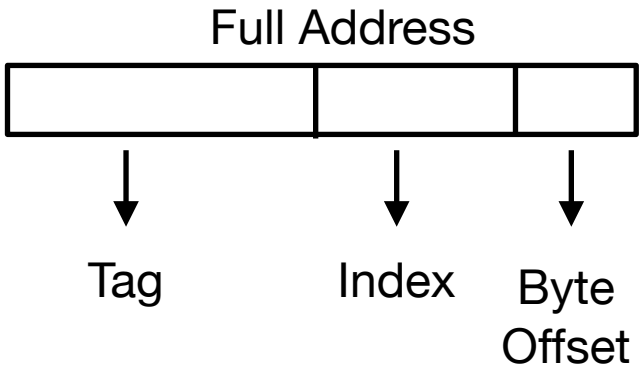
T = 0x11D

I = 0x0

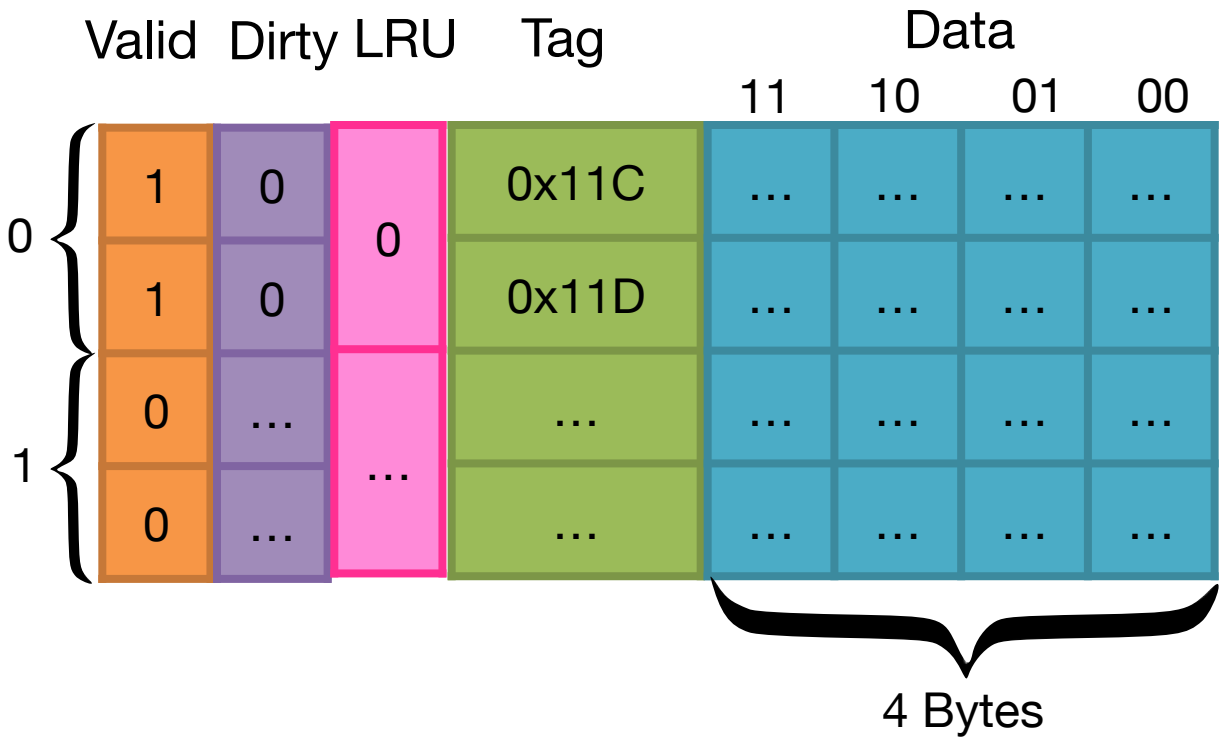
O = 0x0

Cache Miss => bring in entire line

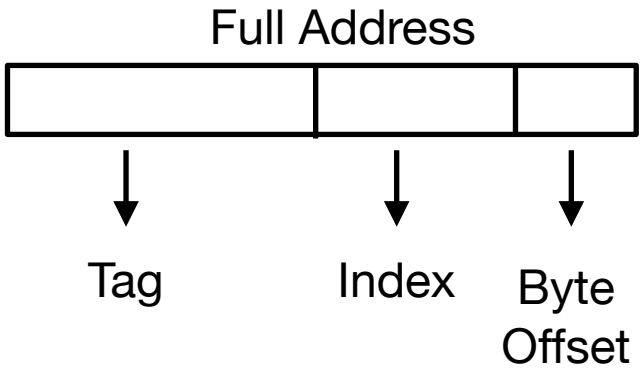
Set Associative (write-back)



Read Byte 0x8E9



Set Associative (write-back)



Read Byte 0x8E9

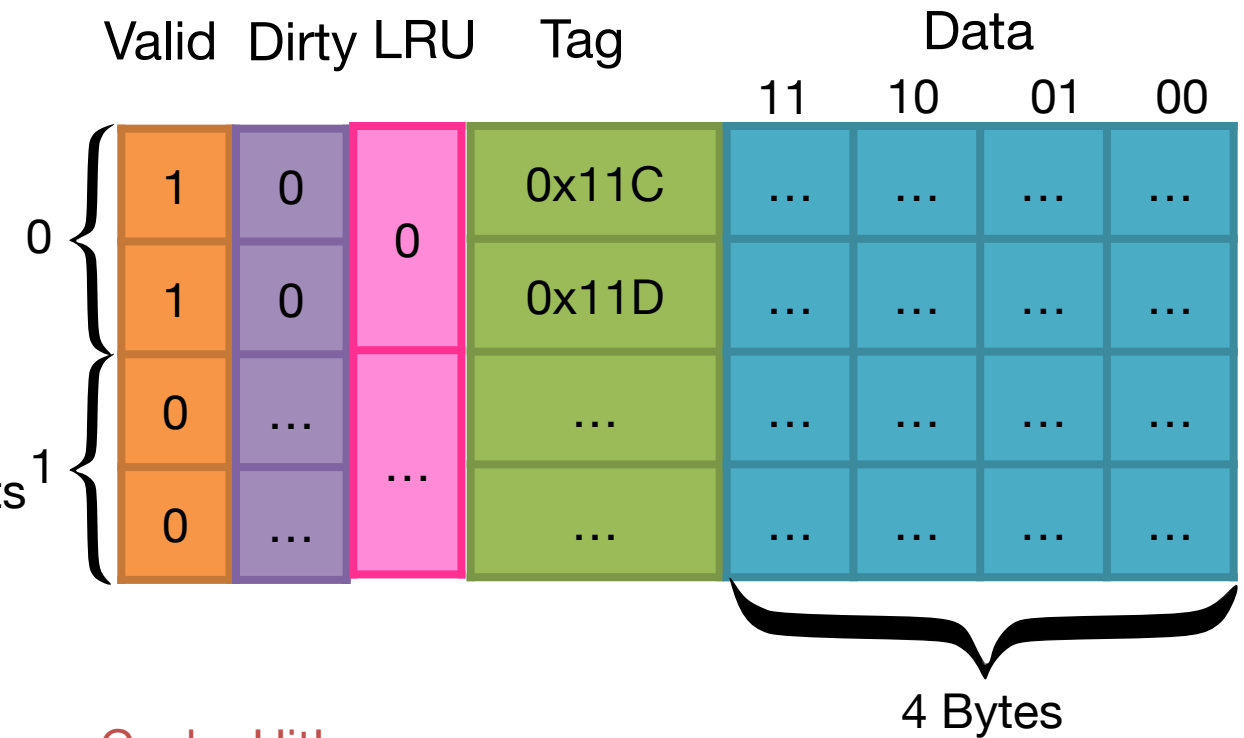
T = 9 bits I = 1 bit O = 2 bits

0b 1000 1110 1001

T = 0x11D

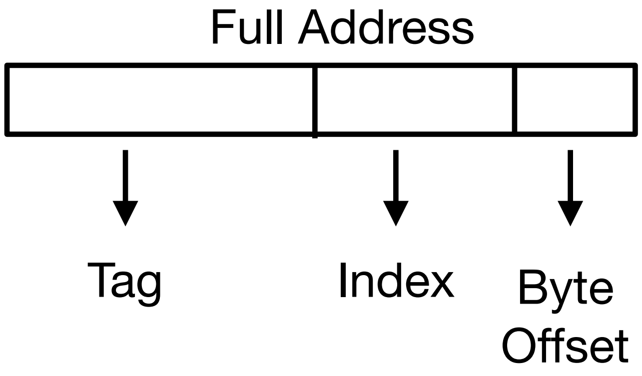
I = 0x0

O = 0x1

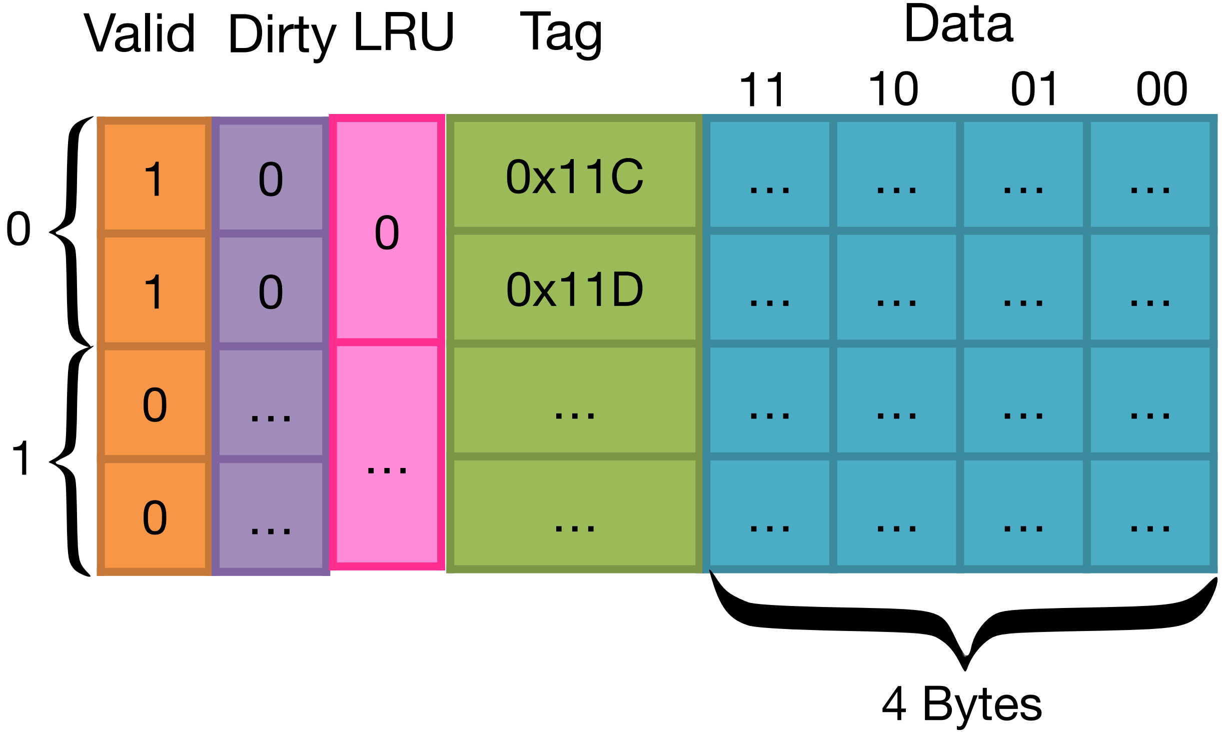


Cache Hit!

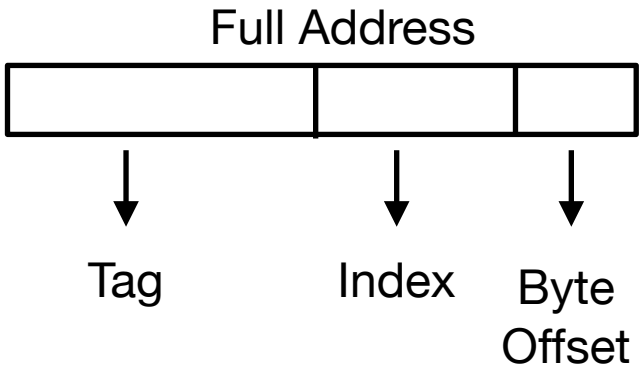
Set Associative (write-back)



Read Byte 0xDF7



Set Associative (write-back)



Read Byte 0xDF7

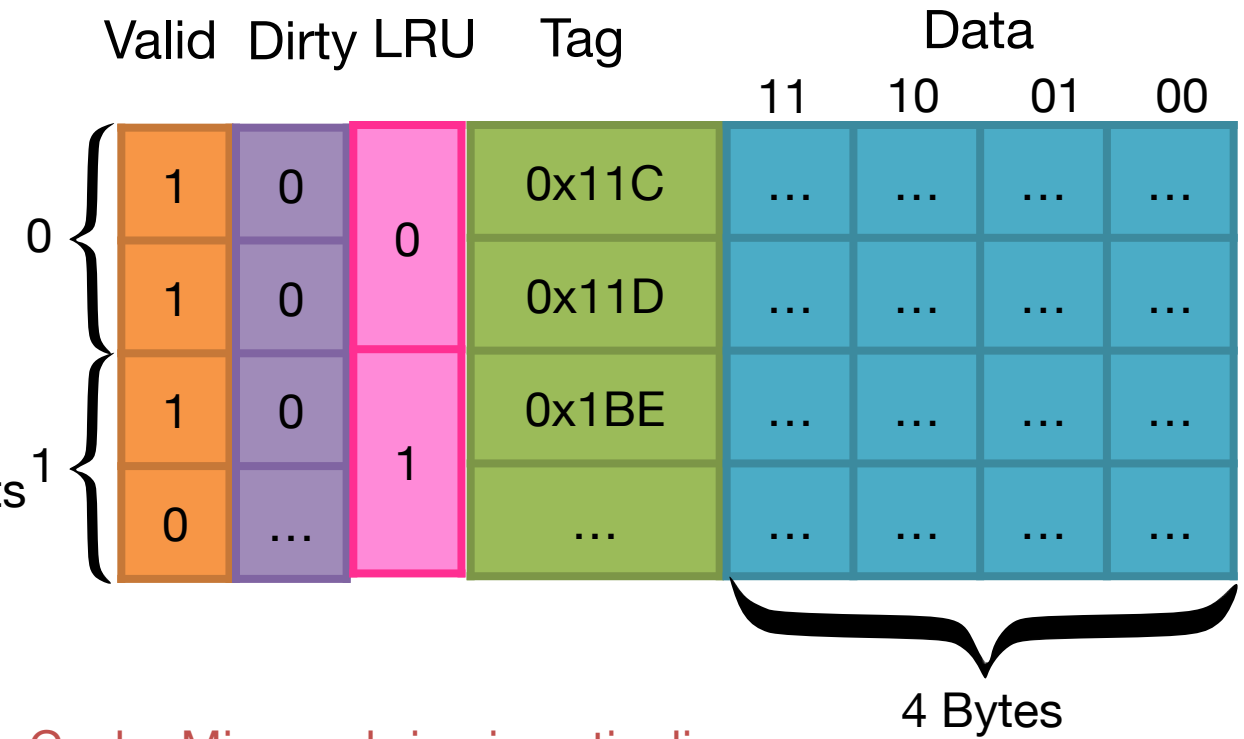
T = 9 bits I = 1 bit O = 2 bits

0b 1101 1111 0111

T = 0x1BE

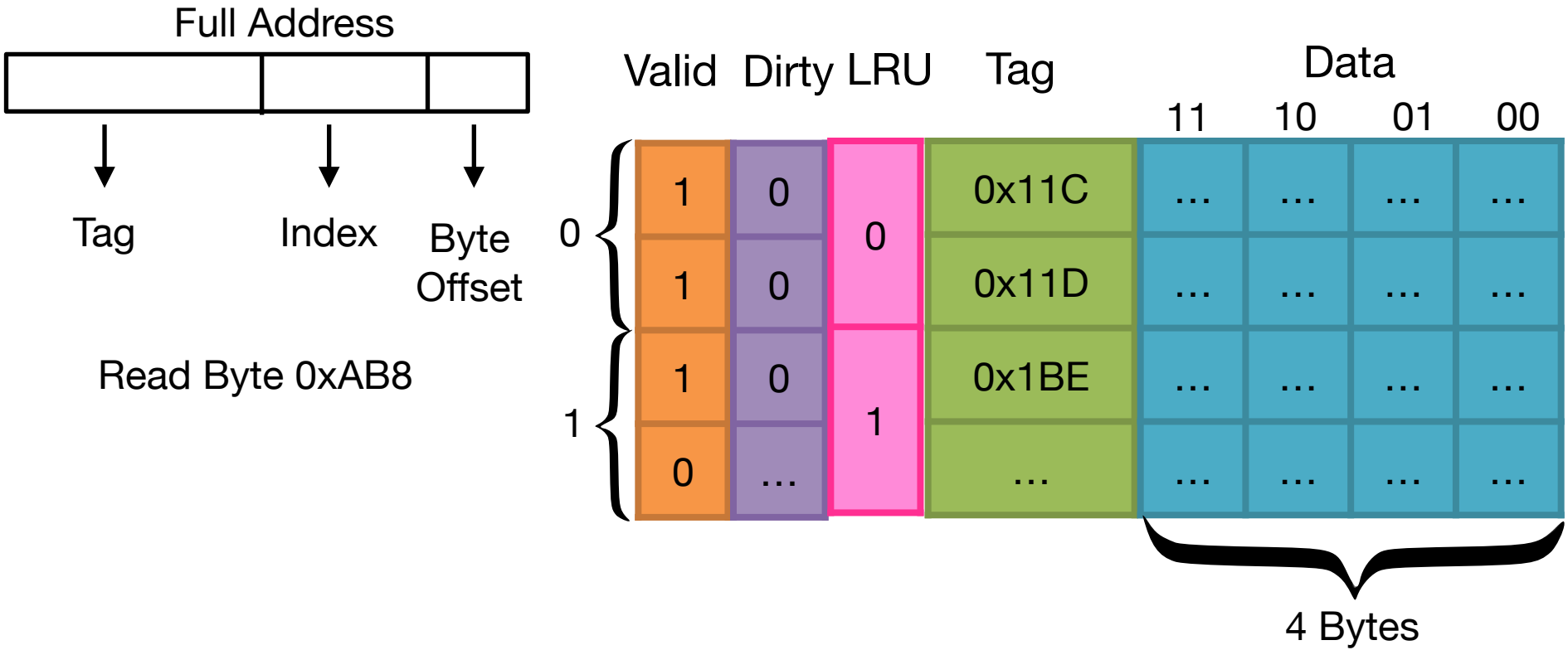
I = 0x1

O = 0x3

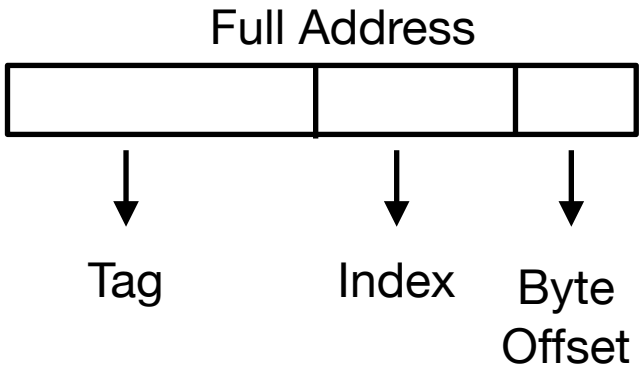


Cache Miss => bring in entire line

Set Associative (write-back)



Set Associative (write-back)



Read Byte 0xAB8

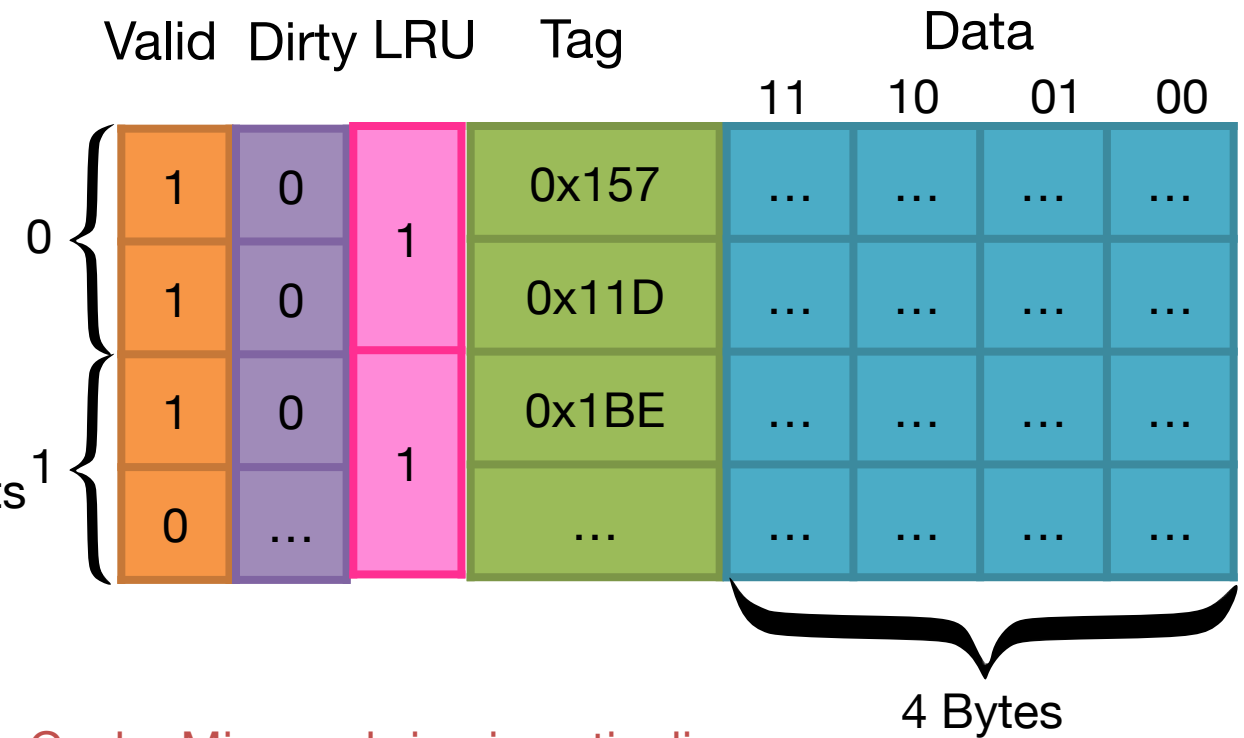
T = 9 bits I = 1 bit O = 2 bits

0b 1010 1011 1000

T = 0x157

I = 0x0

O = 0x0



Cache Miss => bring in entire line

Next Lecture

- Cache Performance
- Multilevel Caches