

# **CSC 252/452: Computer Organization**

## **Fall 2025: Lecture 16**

Instructor: Yanan Guo

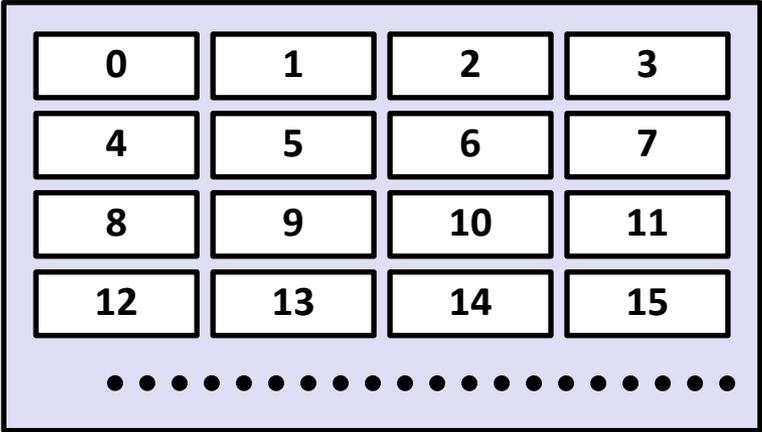
Department of Computer Science  
University of Rochester

# Cache Illustrations

CPU



Memory  
(big but slow)

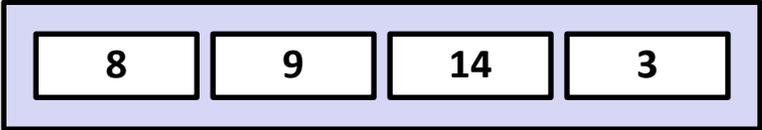


# Cache Illustrations

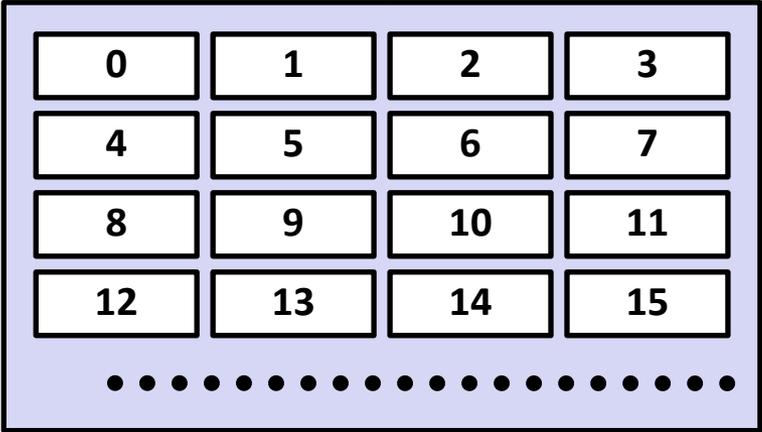
CPU



Cache  
(small but fast)



Memory  
(big but slow)



# Cache Illustrations

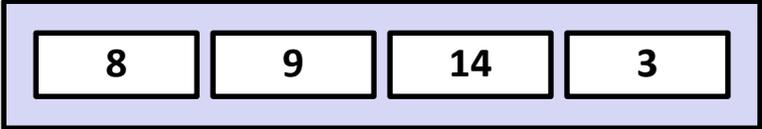
CPU



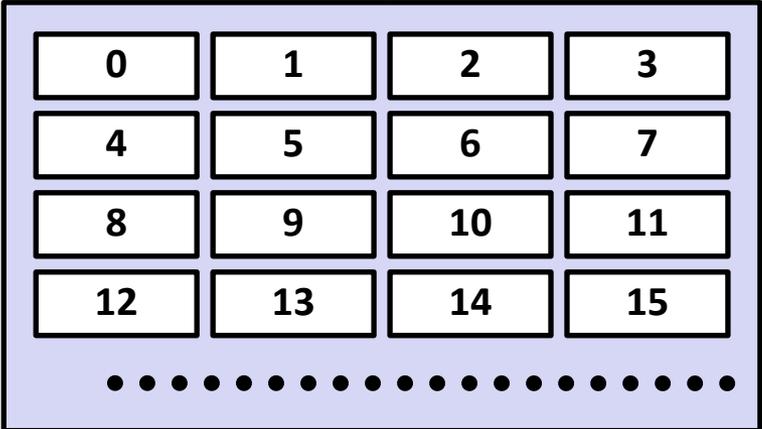
Request Data  
at Address 14

*Data in address b is needed*

Cache  
(small but fast)



Memory  
(big but slow)



# Cache Illustrations

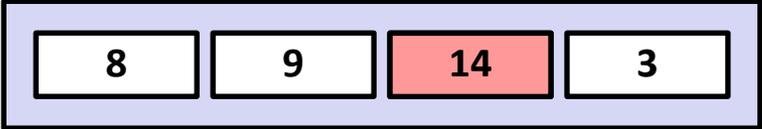
CPU



Request Data  
at Address 14

*Data in address  $b$  is needed*

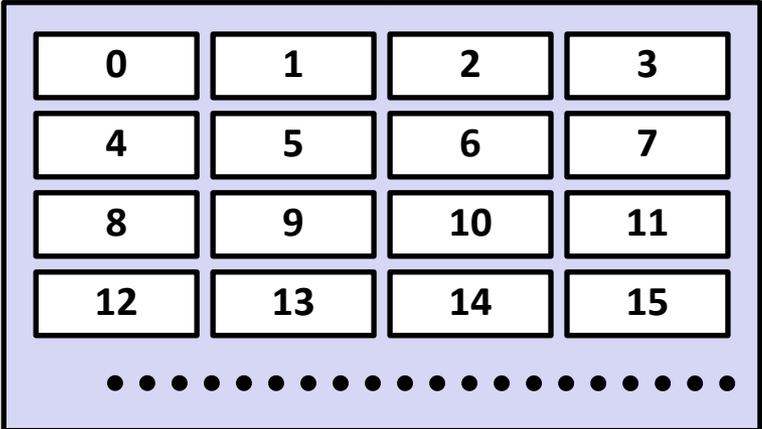
Cache  
(small but fast)



*Address  $b$  is in cache: **Hit!***



Memory  
(big but slow)

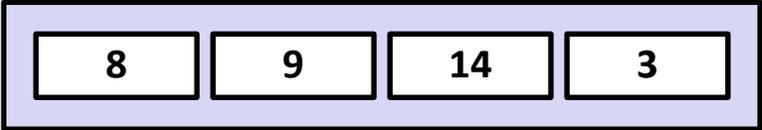


# Cache Illustrations

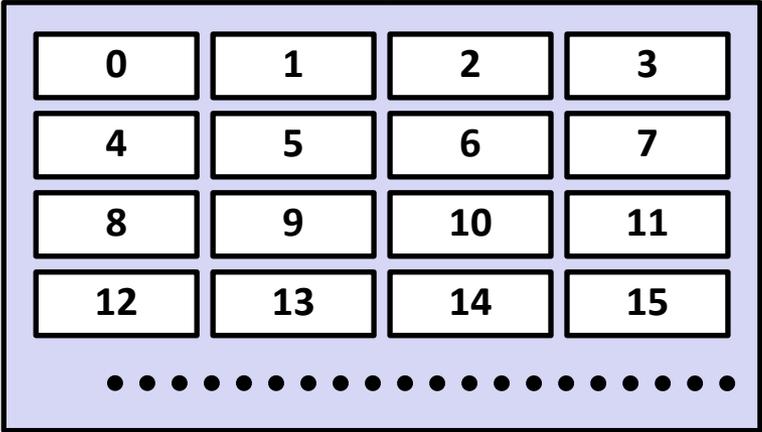
CPU



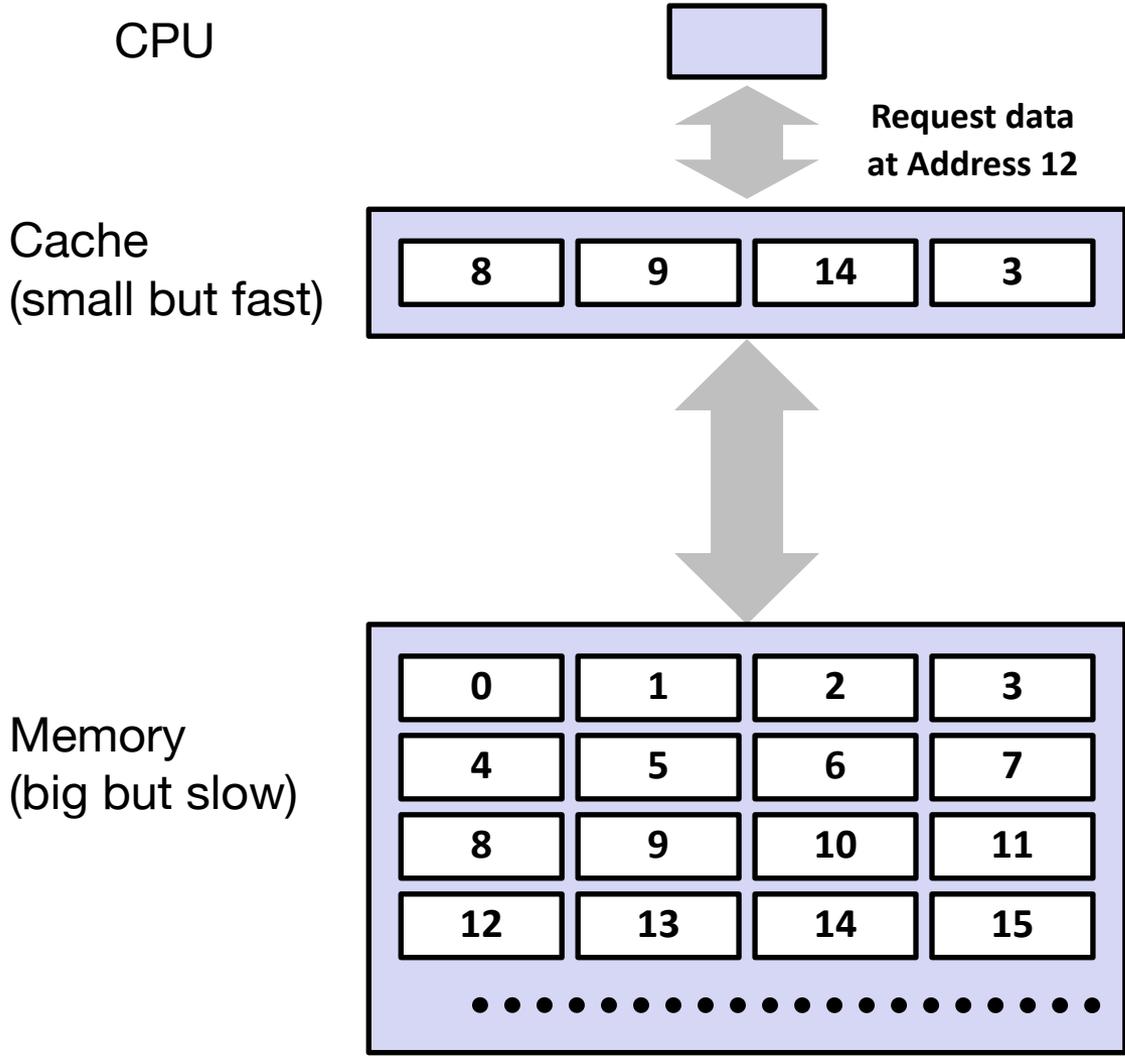
Cache  
(small but fast)



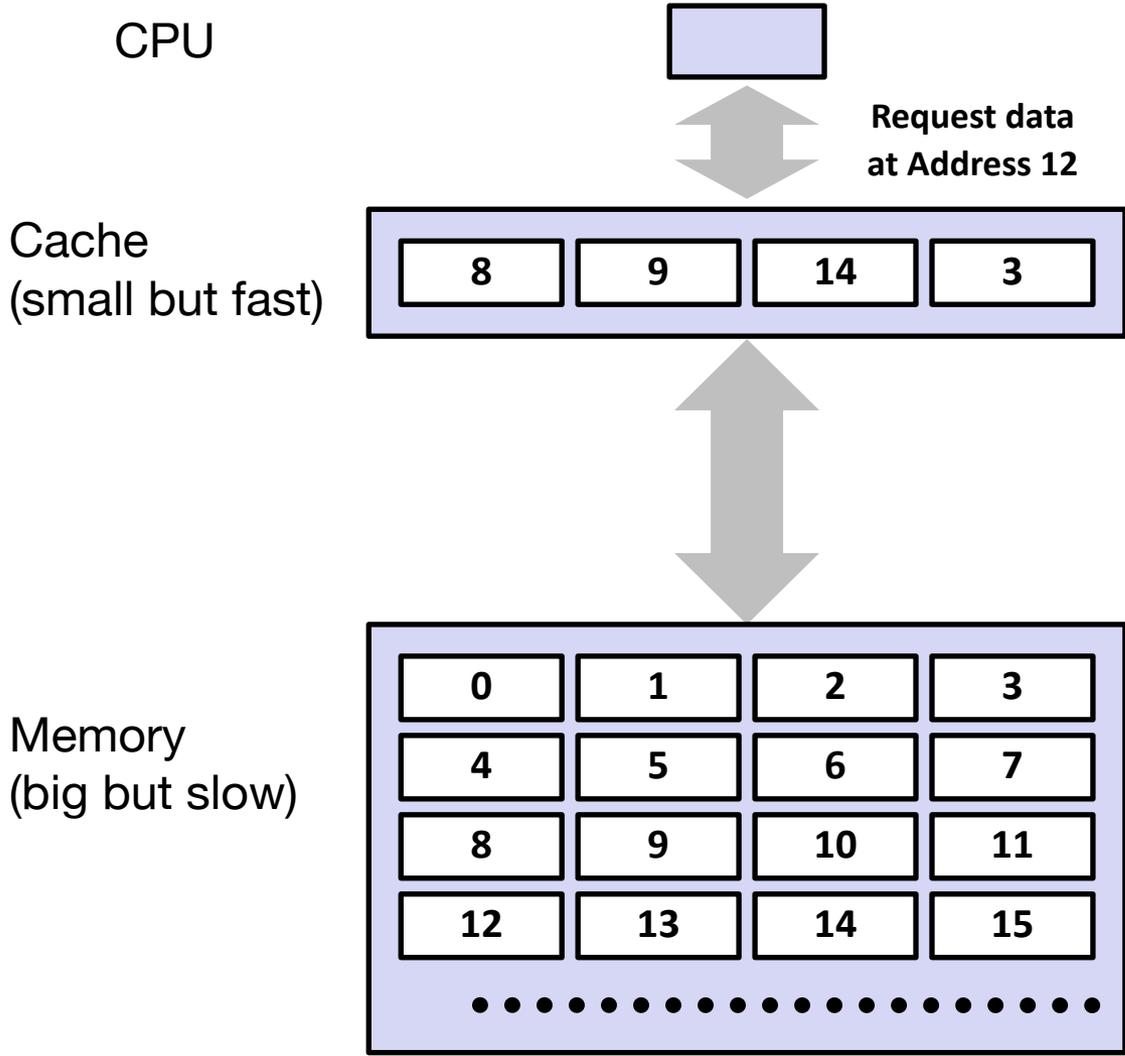
Memory  
(big but slow)



# Cache Illustrations



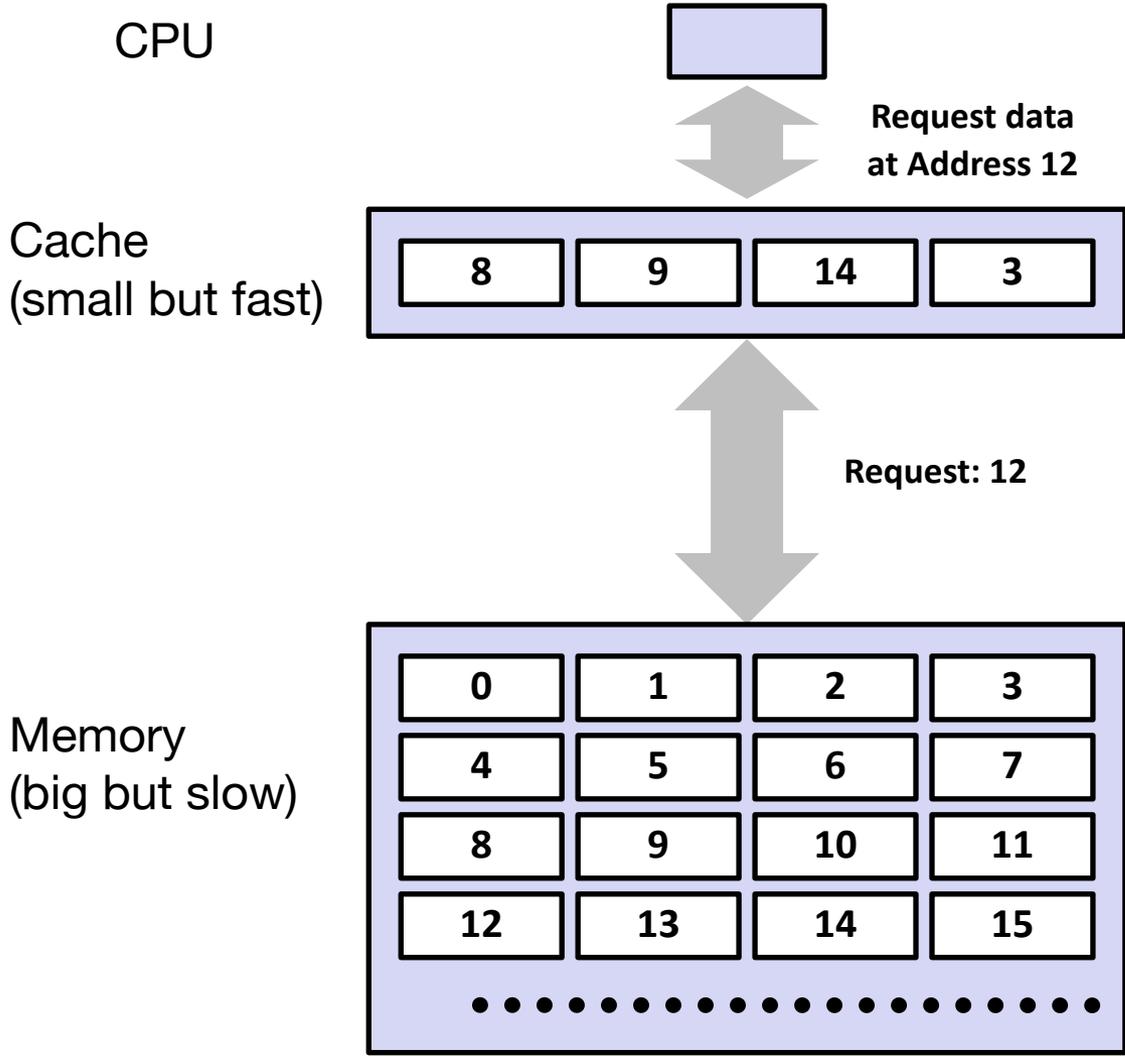
# Cache Illustrations



*Data in address  $b$  is needed*

*Address  $b$  is not in cache:  
**Miss!***

# Cache Illustrations

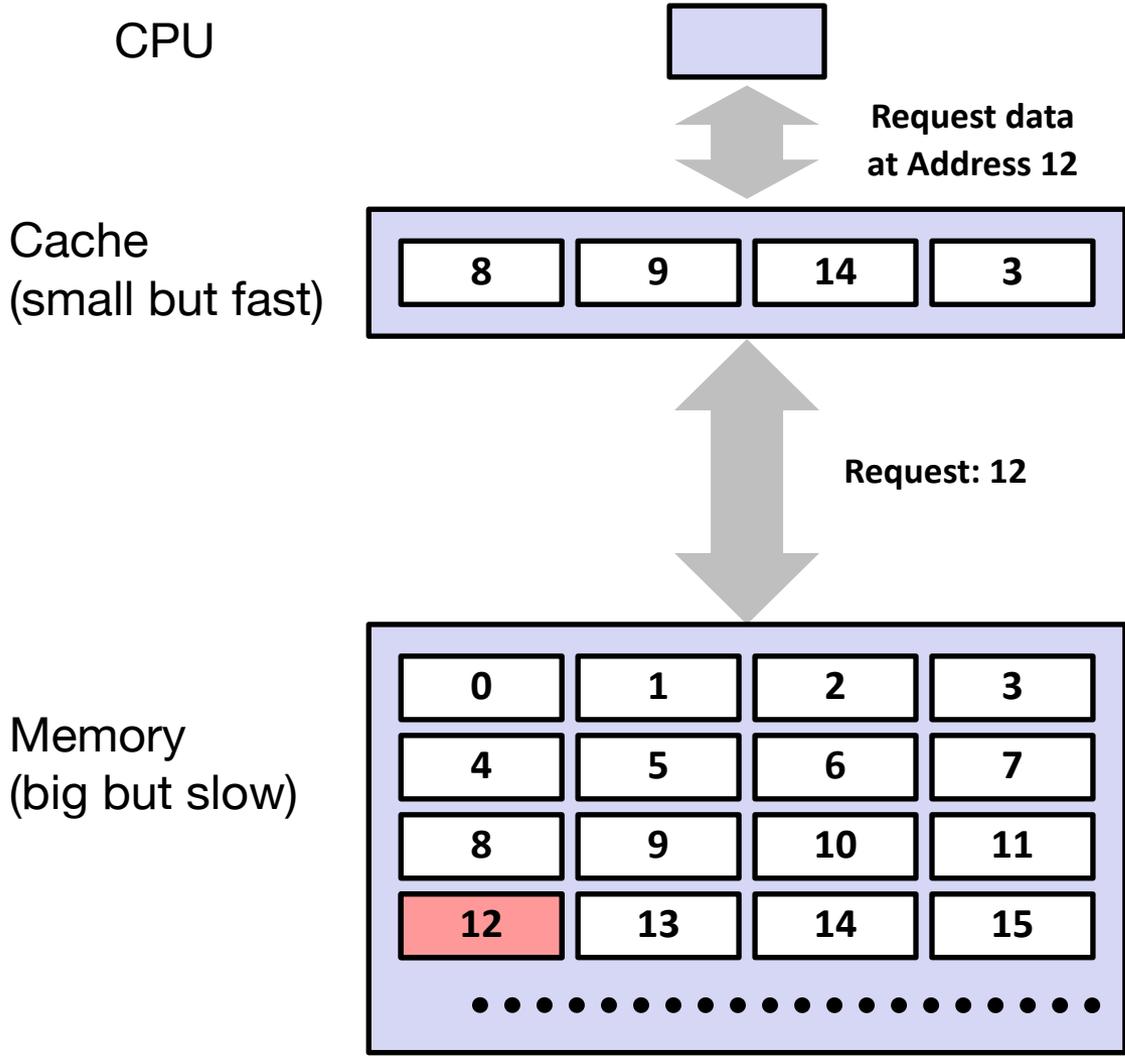


*Data in address b is needed*

*Address b is not in cache:  
**Miss!***

*Address b is fetched from  
memory*

# Cache Illustrations

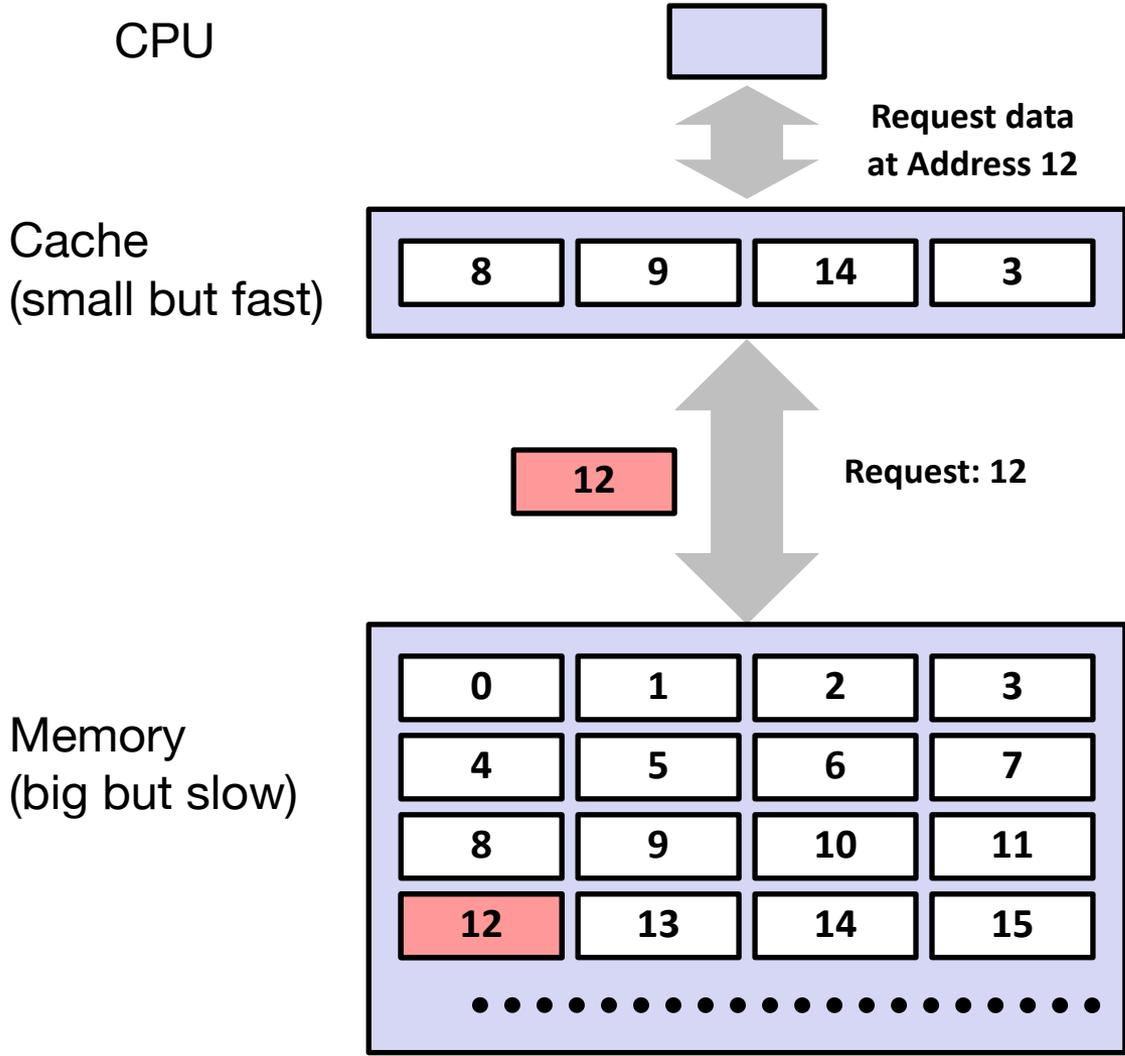


*Data in address  $b$  is needed*

*Address  $b$  is not in cache:  
**Miss!***

*Address  $b$  is fetched from  
memory*

# Cache Illustrations



*Data in address b is needed*

*Address b is not in cache:  
**Miss!***

*Address b is fetched from  
memory*

# Cache Illustrations

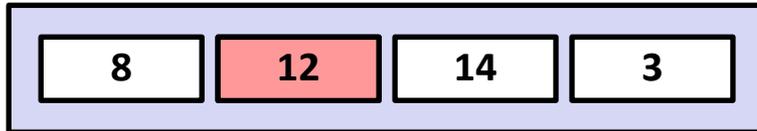
CPU



Request data  
at Address 12



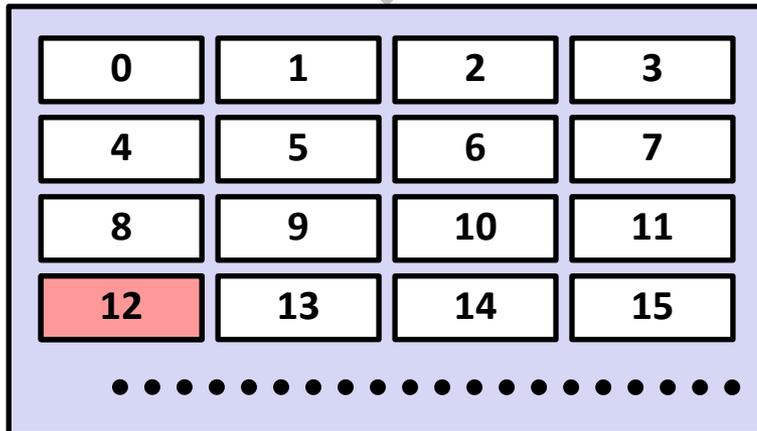
Cache  
(small but fast)



Request: 12



Memory  
(big but slow)



*Data in address  $b$  is needed*

*Address  $b$  is not in cache:  
**Miss!***

*Address  $b$  is fetched from  
memory*

*Address  $b$  is stored in cache*

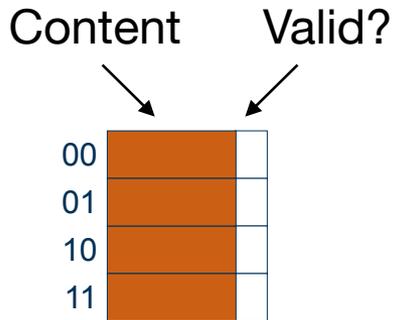
# Cache Hit Rate

- Cache hit is when you find the data in the cache
- Hit rate indicates the effectiveness of the cache

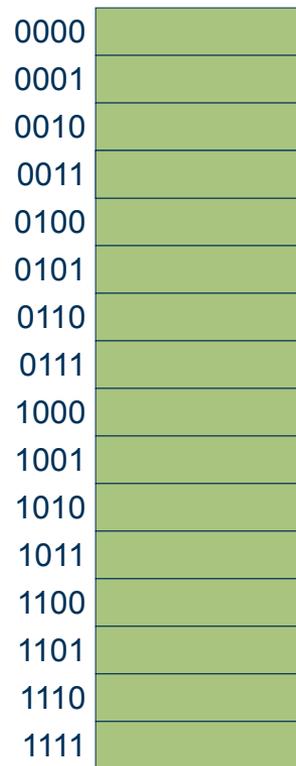
$$\text{Hit Rate} = \frac{\# \text{ Hits}}{\# \text{ Accesses}}$$

# A Simple Cache

Cache



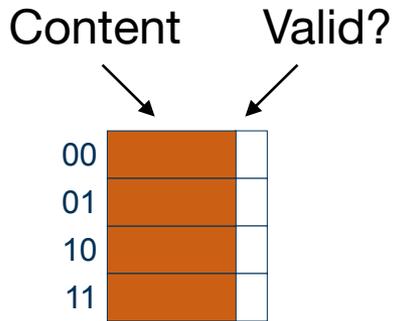
Memory



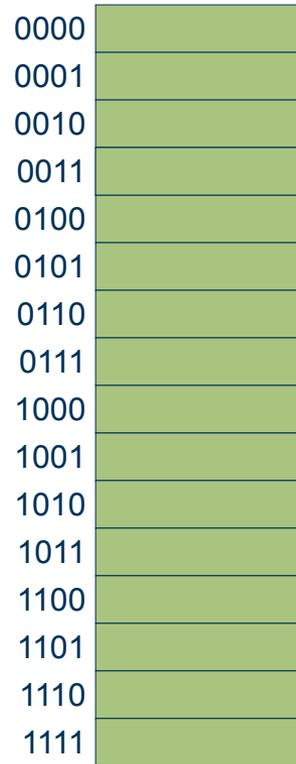
- 16 memory locations
- 4 cache locations

# A Simple Cache

Cache



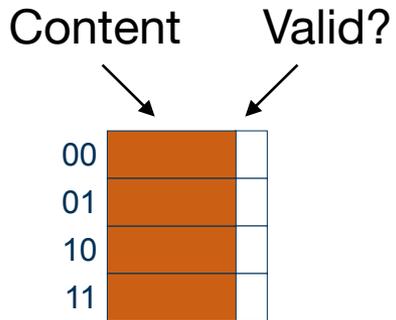
Memory



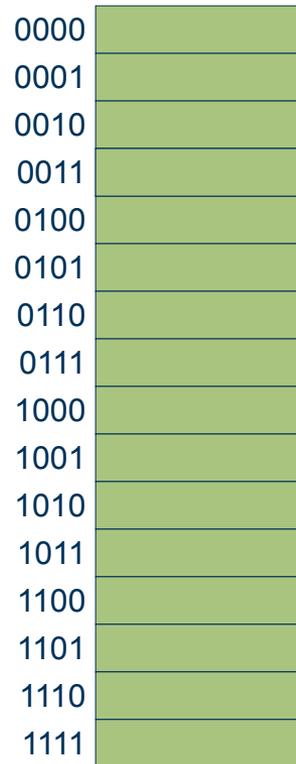
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**

# A Simple Cache

Cache



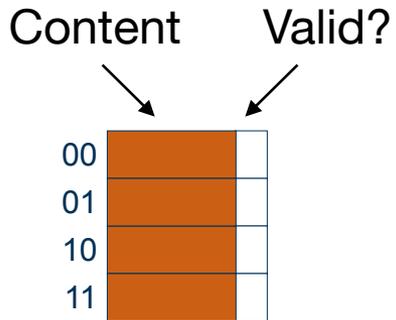
Memory



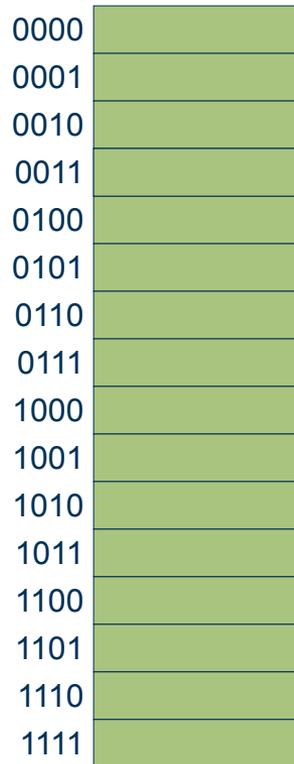
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.

# A Simple Cache

Cache



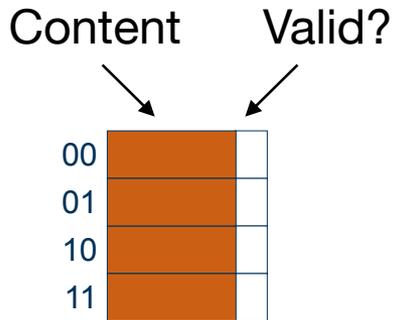
Memory



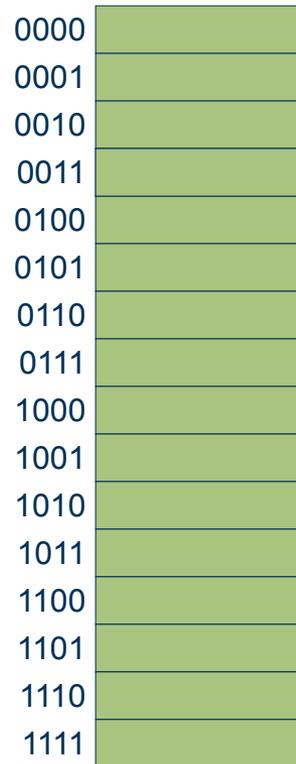
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B

# A Simple Cache

Cache



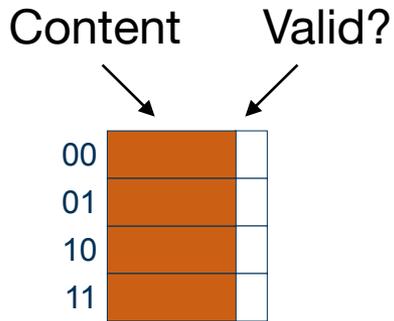
Memory



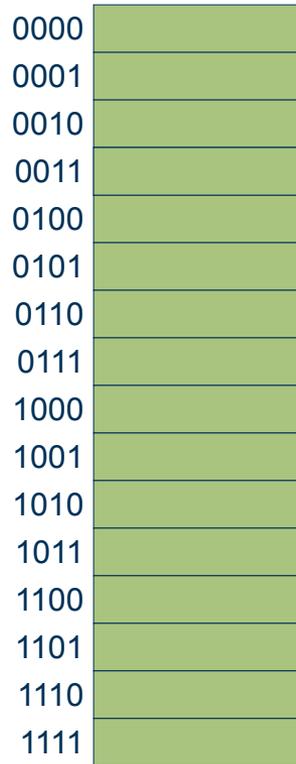
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Cache is smaller than memory (obviously)

# A Simple Cache

Cache



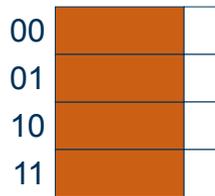
Memory



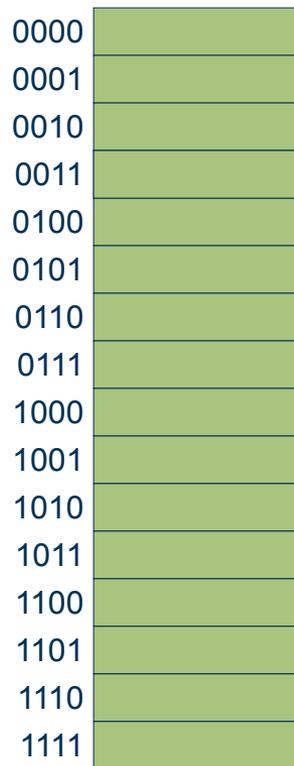
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Cache is smaller than memory (obviously)
  - Thus, not all memory locations can be cached at the same time

# Cache Placement

Cache



Memory



- Given a memory addr, say 0x0001, we want to put the data there into the cache; where does the data go?

# Fully-Associative Cache

## Cache

	Content	Valid?
00	0xAA	
01	0xBB	
10	0xCC	
11	0xDD	

## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	0xAA
0111	
1000	0xBB
1001	
1010	
1011	0xCC
1100	0xDD
1101	
1110	
1111	

- Every memory location can be mapped to any cache line in the cache.
- Given a request to address A from CPU, detecting cache hit/miss requires:
  - Comparing address A with all four tags in the cache (a.k.a., associative search)
- Can we reduce the overhead:
  - of storing tags
  - of comparison

# Fully-Associative Cache

## Cache

	Content	Valid?
00	0xBB	
01	0xAA	
10	0xCC	
11	0xDD	

## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	0xAA
0111	
1000	0xBB
1001	
1010	
1011	0xCC
1100	0xDD
1101	
1110	
1111	

- Every memory location can be mapped to any cache line in the cache.
- Given a request to address A from CPU, detecting cache hit/miss requires:
  - Comparing address A with all four tags in the cache (a.k.a., associative search)
- Can we reduce the overhead:
  - of storing tags
  - of comparison

# Fully-Associative Cache

## Cache

	Content	Valid?	Tag
00	0xBB		
01	0xAA		
10	0xCC		
11	0xDD		

## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	0xAA
0111	
1000	0xBB
1001	
1010	
1011	0xCC
1100	0xDD
1101	
1110	
1111	

- Every memory location can be mapped to any cache line in the cache.
- Given a request to address A from CPU, detecting cache hit/miss requires:
  - Comparing address A with all four tags in the cache (a.k.a., associative search)
- Can we reduce the overhead:
  - of storing tags
  - of comparison

# Fully-Associative Cache

## Cache

	Content	Valid?	Tag
00	0xBB		addr [3:0]
01	0xAA		addr [3:0]
10	0xCC		addr [3:0]
11	0xDD		addr [3:0]

## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	0xAA
0111	
1000	0xBB
1001	
1010	
1011	0xCC
1100	0xDD
1101	
1110	
1111	

- Every memory location can be mapped to any cache line in the cache.
- Given a request to address A from CPU, detecting cache hit/miss requires:
  - Comparing address A with all four tags in the cache (a.k.a., associative search)
- Can we reduce the overhead:
  - of storing tags
  - of comparison

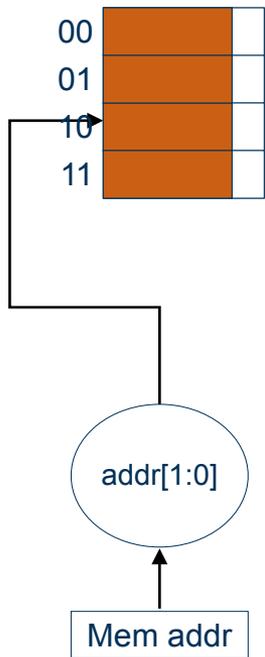
# A Few Terminologies

- A cache line: content + valid bit + tag bits
  - Valid bit + tag bits are “overhead”
  - Content is what you really want to store
  - But we need valid and tag bits to correctly access the cache

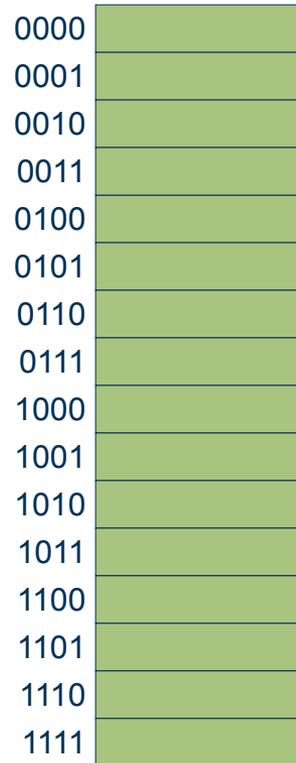
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	0xEF
1001	0xAC
1010	0x06
1011	
1100	
1101	0x70
1110	
1111	

# Direct-Mapped Cache

Cache



Memory

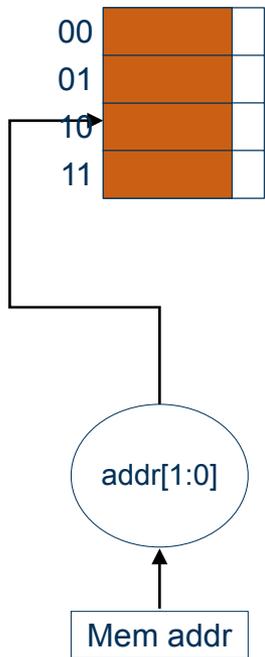


• Direct-Mapped Cache

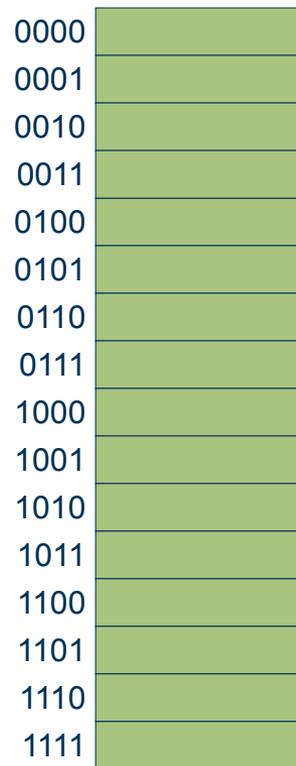
- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$

# Direct-Mapped Cache

Cache



Memory

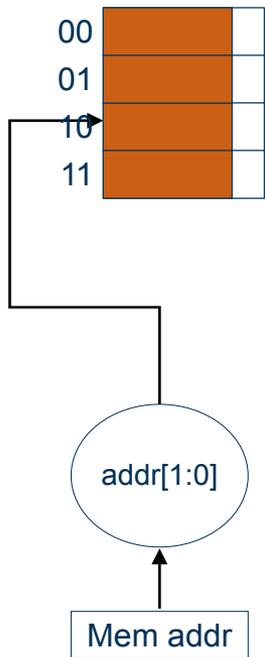


- Direct-Mapped Cache

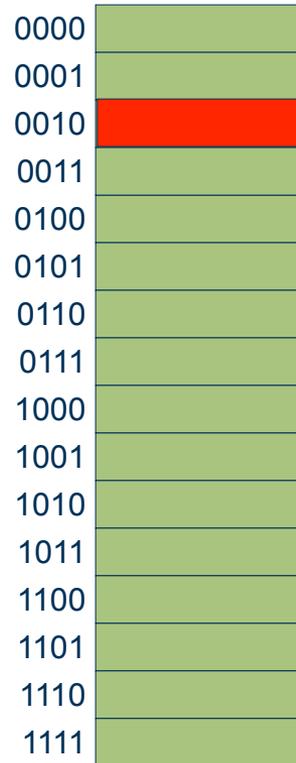
- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

# Direct-Mapped Cache

Cache



Memory

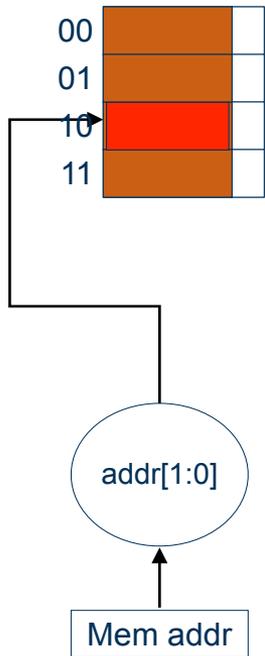


## • Direct-Mapped Cache

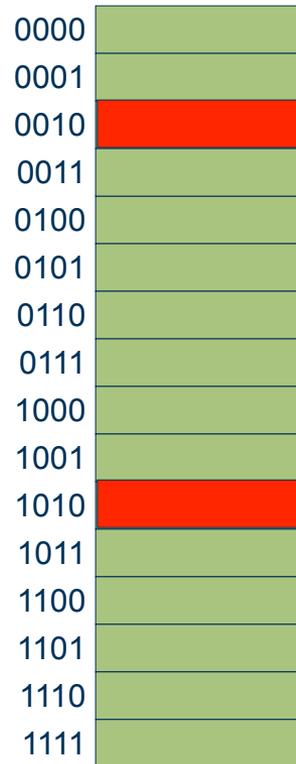
- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

# Direct-Mapped Cache

Cache



Memory



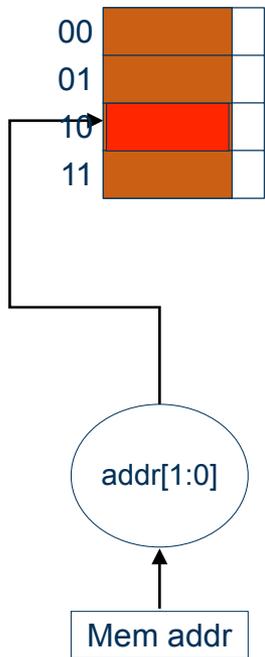
- Direct-Mapped Cache

- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

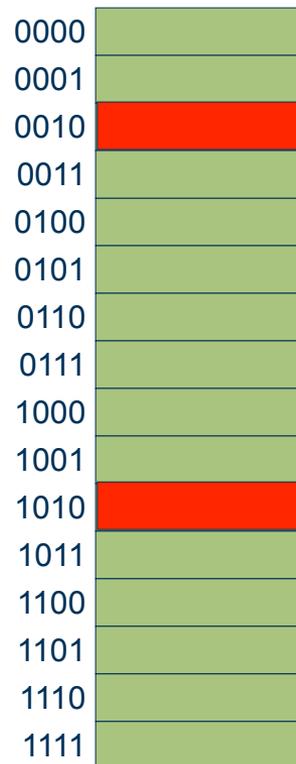
- Multiple addresses can be mapped to the same location

# Direct-Mapped Cache

Cache



Memory



- Direct-Mapped Cache

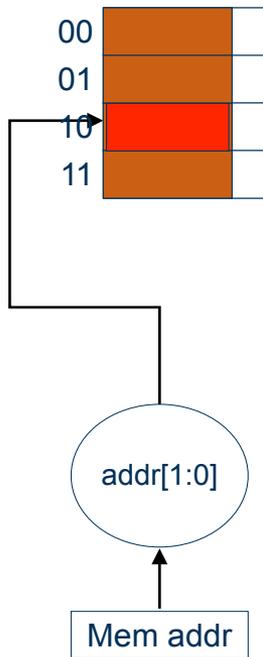
- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

- Multiple addresses can be mapped to the same location

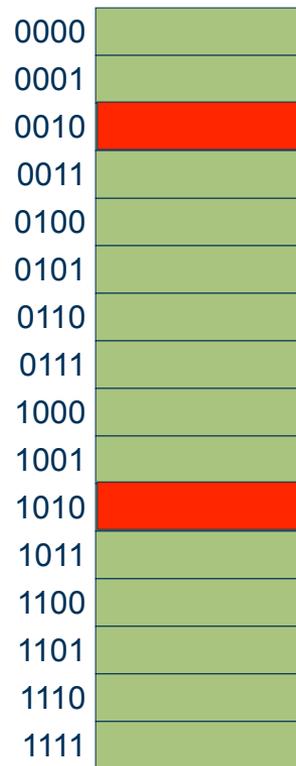
- E.g., 0010 and 1010

# Direct-Mapped Cache

Cache



Memory



- Direct-Mapped Cache

- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

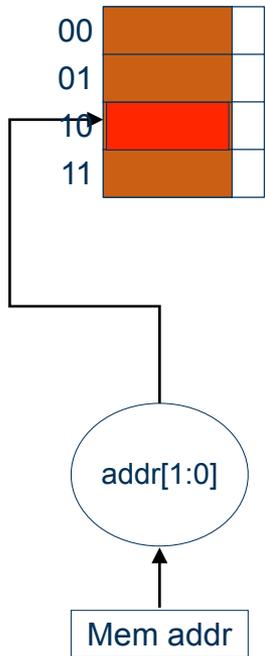
- Multiple addresses can be mapped to the same location

- E.g., 0010 and 1010

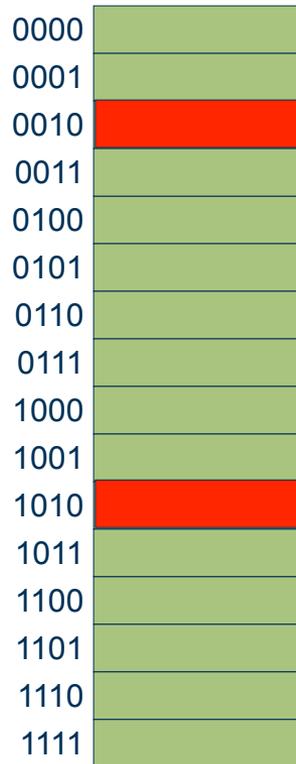
- How do we differentiate between different memory locations that are mapped to the same cache location?

# Direct-Mapped Cache

Cache



Memory



- Direct-Mapped Cache

- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

- Multiple addresses can be mapped to the same location

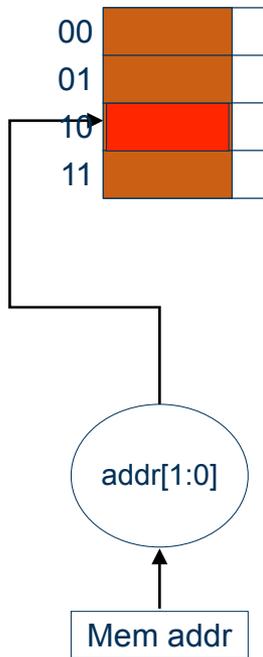
- E.g., 0010 and 1010

- How do we differentiate between different memory locations that are mapped to the same cache location?

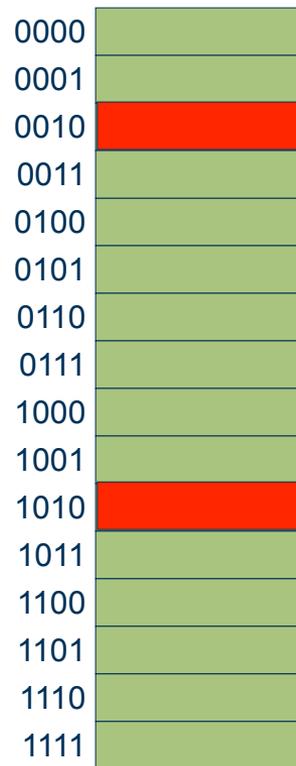
- Add a tag field for that purpose

# Direct-Mapped Cache

## Cache



## Memory

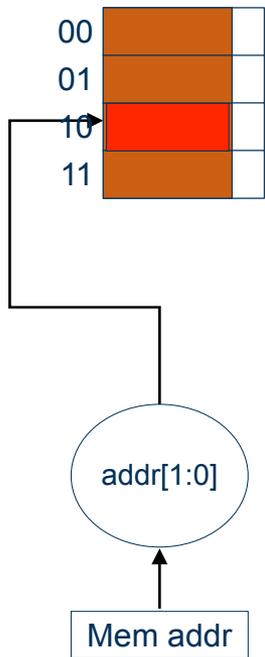


## • Direct-Mapped Cache

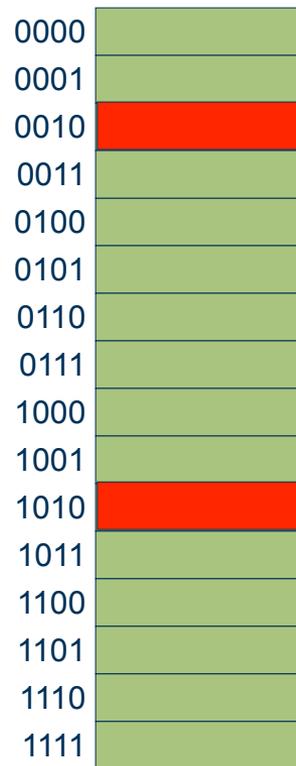
- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that purpose
  - What should the tag field be?

# Direct-Mapped Cache

Cache



Memory



- Direct-Mapped Cache

- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

- Multiple addresses can be mapped to the same location

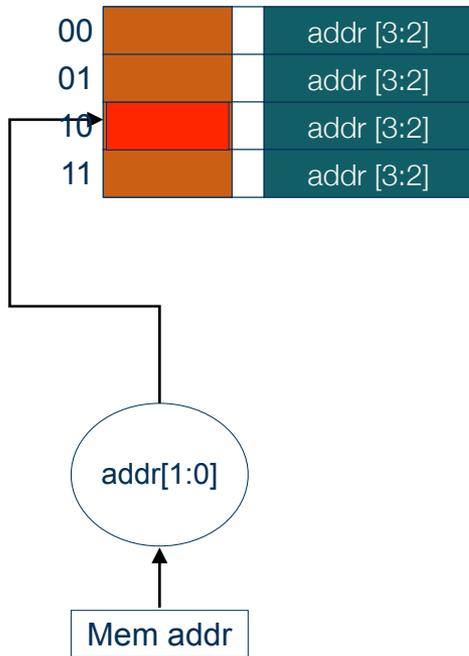
- E.g., 0010 and 1010

- How do we differentiate between different memory locations that are mapped to the same cache location?

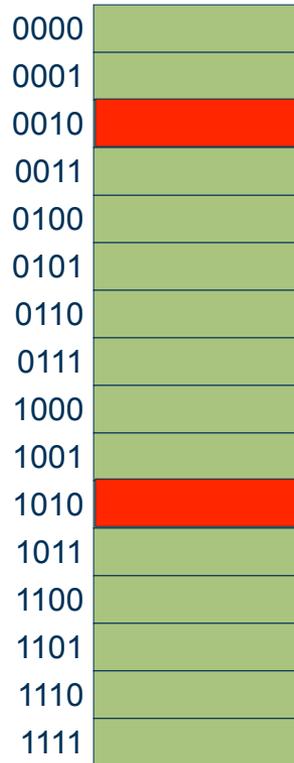
- Add a tag field for that purpose
- What should the tag field be?
- $ADDR[3]$  and  $ADDR[2]$  in this particular example

# Direct-Mapped Cache

## Cache



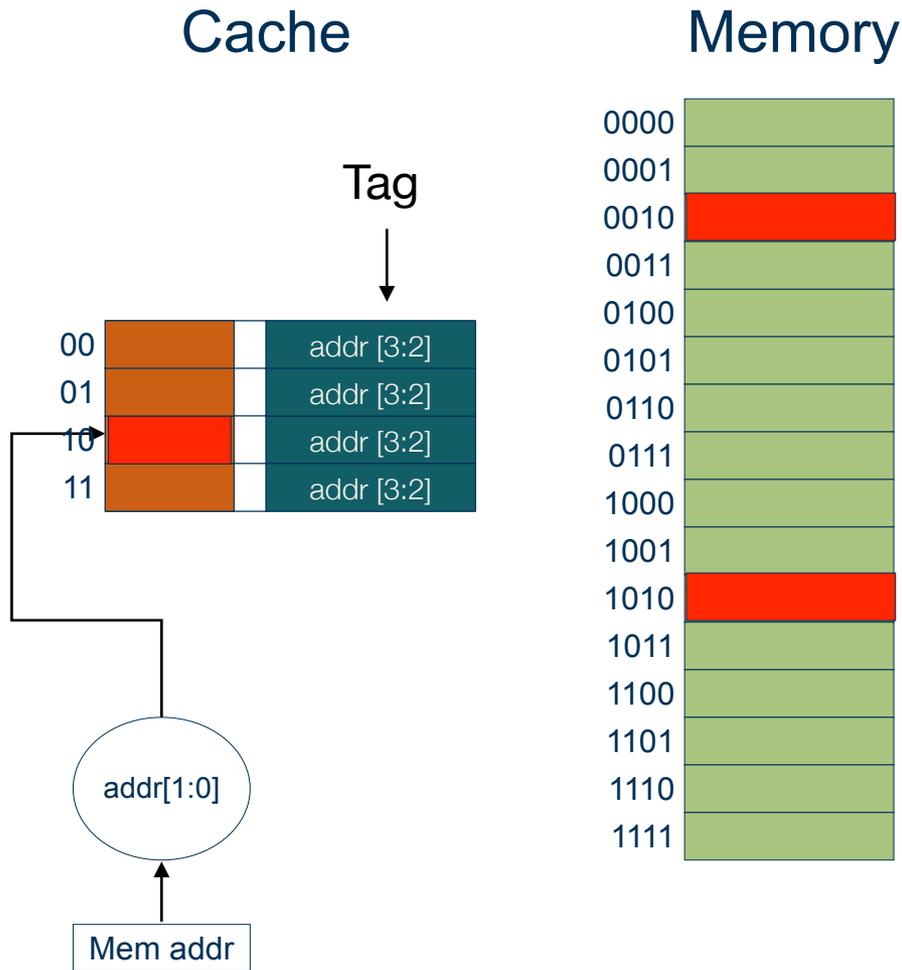
## Memory



## • Direct-Mapped Cache

- One address can only be mapped to one cache line
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- 
- Multiple addresses can be mapped to the same location
    - E.g., 0010 and 1010
  - How do we differentiate between different memory locations that are mapped to the same cache location?
    - Add a tag field for that purpose
    - What should the tag field be?
    - $ADDR[3]$  and  $ADDR[2]$  in this particular example

# Direct-Mapped Cache



- Direct-Mapped Cache

- One address can only be mapped to one cache line
- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

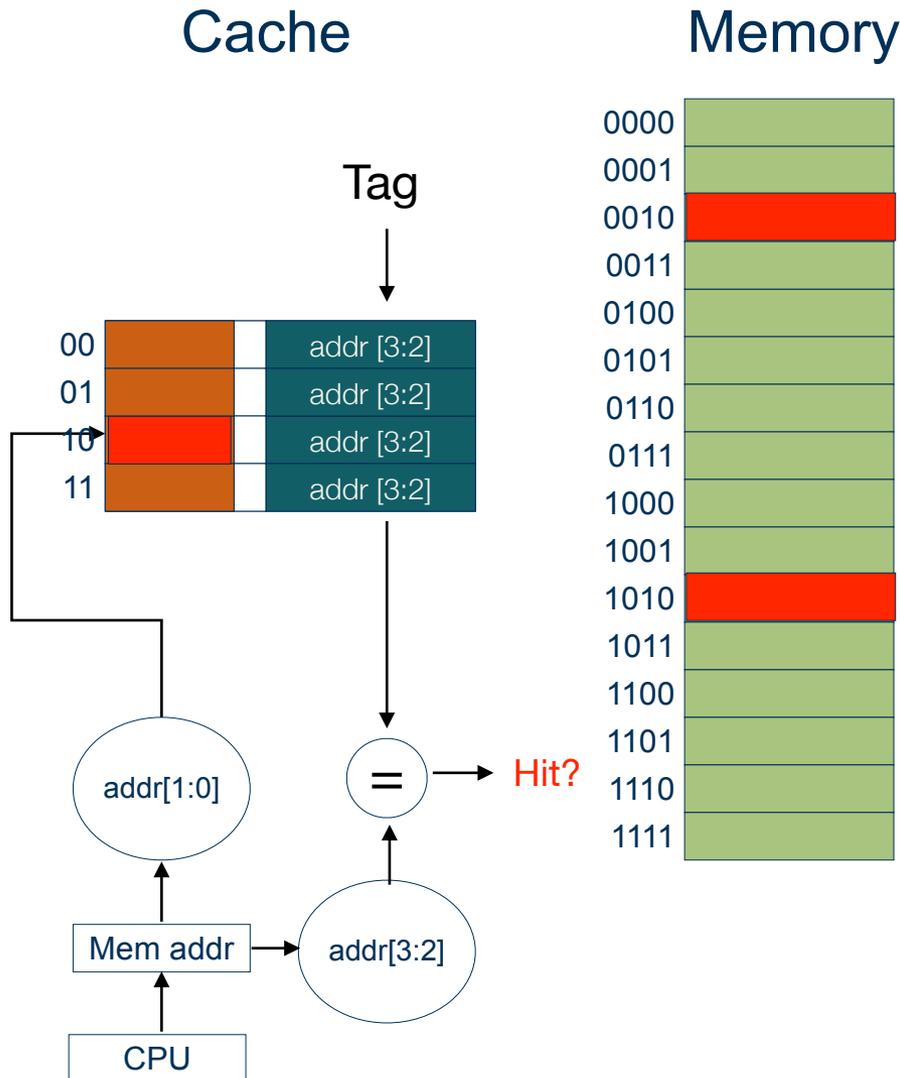
- Multiple addresses can be mapped to the same location

- E.g., 0010 and 1010

- How do we differentiate between different memory locations that are mapped to the same cache location?

- Add a tag field for that purpose
- What should the tag field be?
- $ADDR[3]$  and  $ADDR[2]$  in this particular example

# Direct-Mapped Cache



- Direct-Mapped Cache

- One address can only be mapped to one cache line
- CA = ADDR[1],ADDR[0]
- Always use the lower order address bits

- Multiple addresses can be mapped to the same location

- E.g., 0010 and 1010

- How do we differentiate between different memory locations that are mapped to the same cache location?

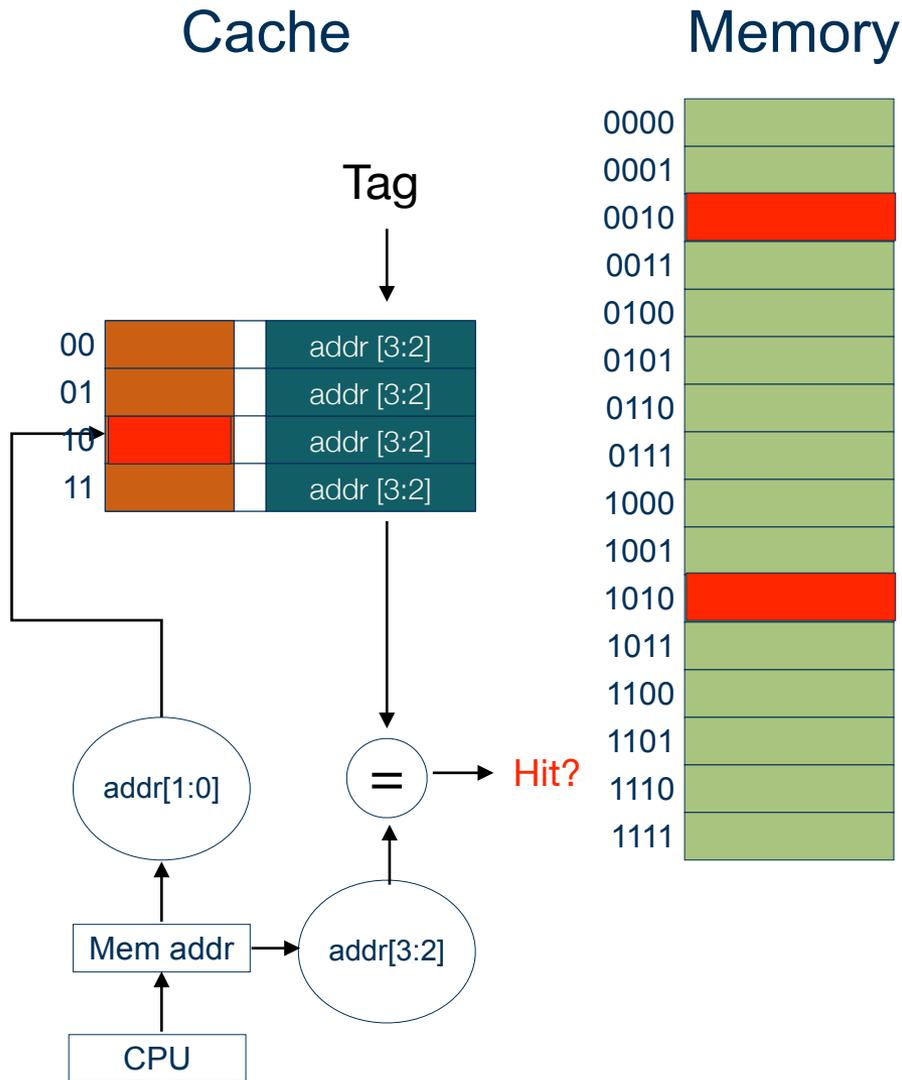
- Add a tag field for that purpose
- What should the tag field be?
- ADDR[3] and ADDR[2] in this particular example

# Direct-Mapped Cache



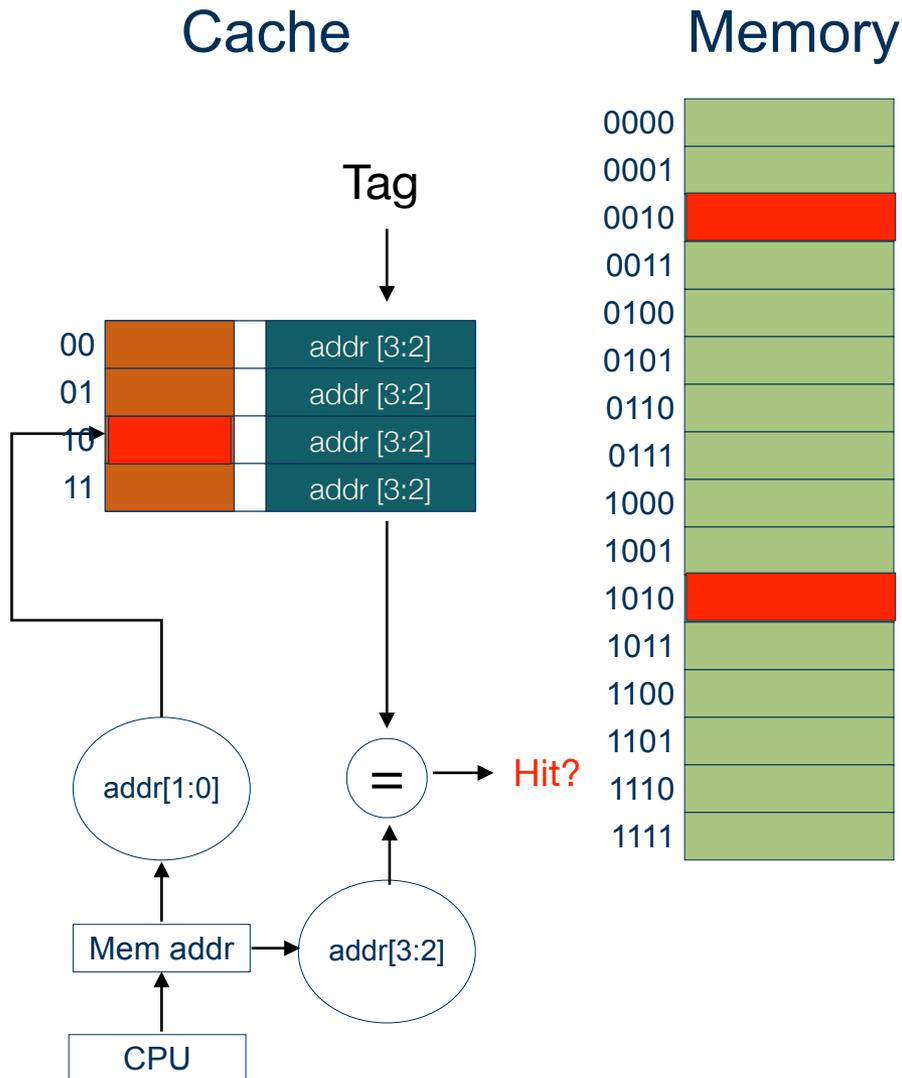
- Limitation: each memory location can be mapped to only one cache location.

# Direct-Mapped Cache



- Limitation: each memory location can be mapped to only one cache location.
- This leads to a lot of conflicts.

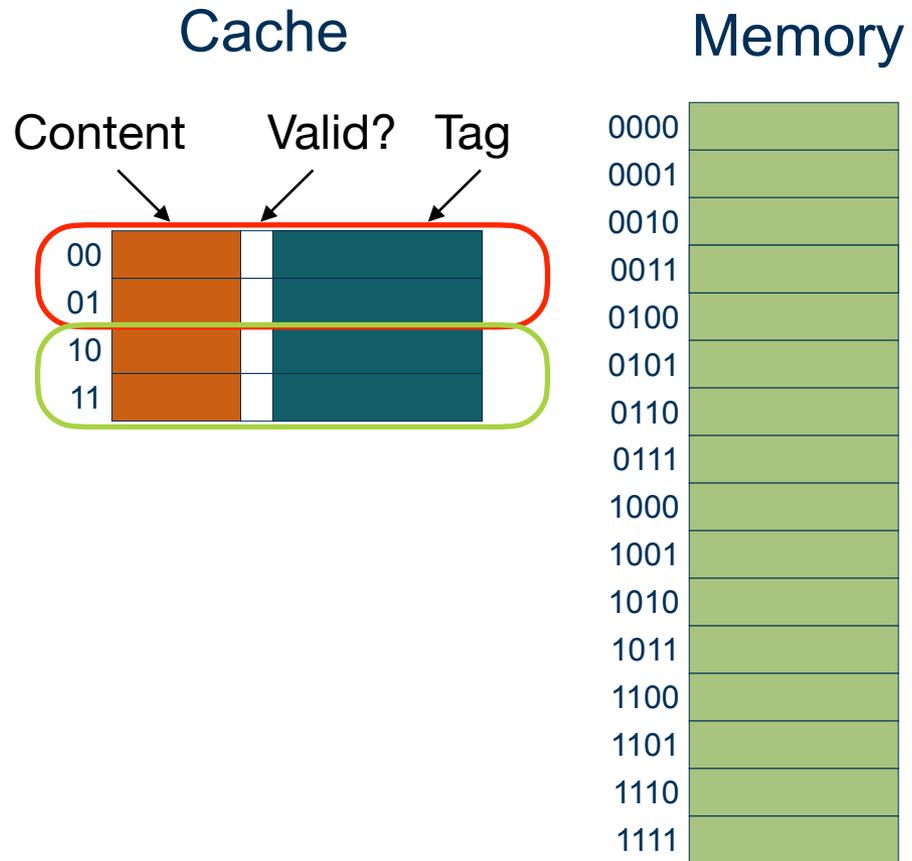
# Direct-Mapped Cache



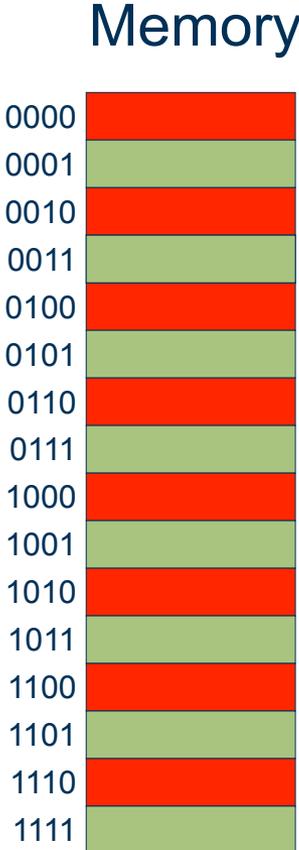
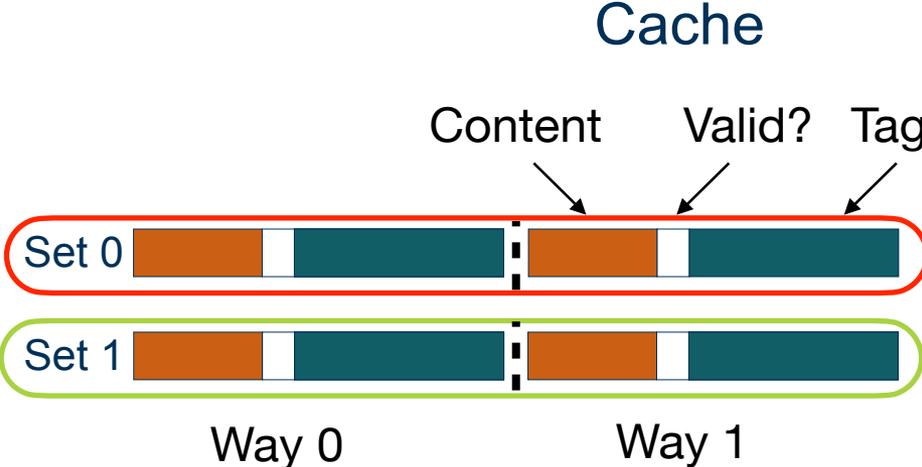
- Limitation: each memory location can be mapped to only one cache location.
- This leads to a lot of conflicts.
- How do we improve this?



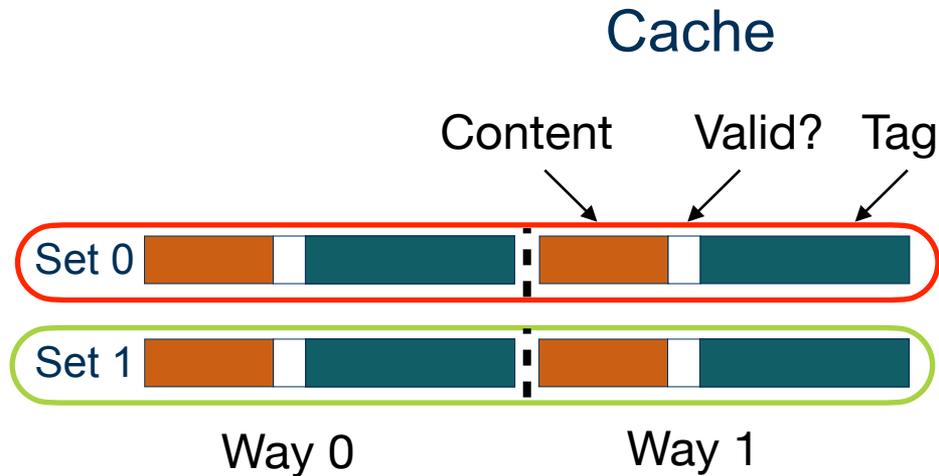
# Set-Associative Cache



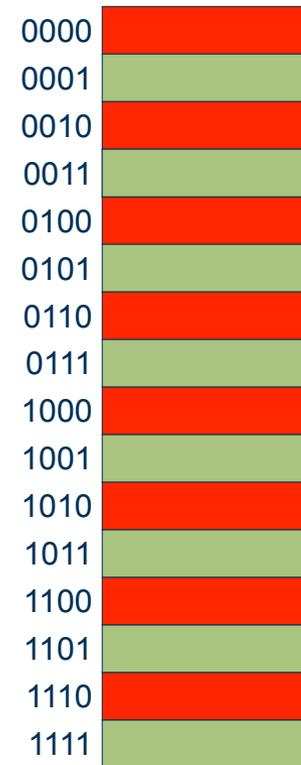
# Set-Associative Cache



# Set-Associative Cache



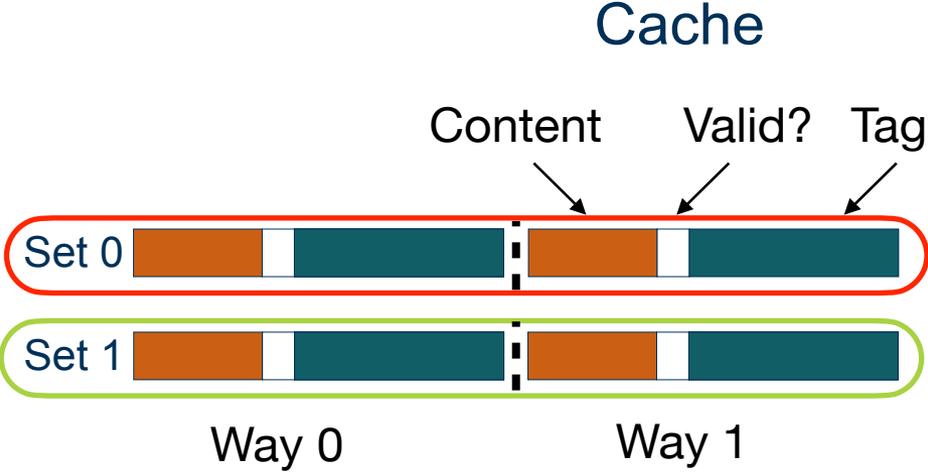
## Memory



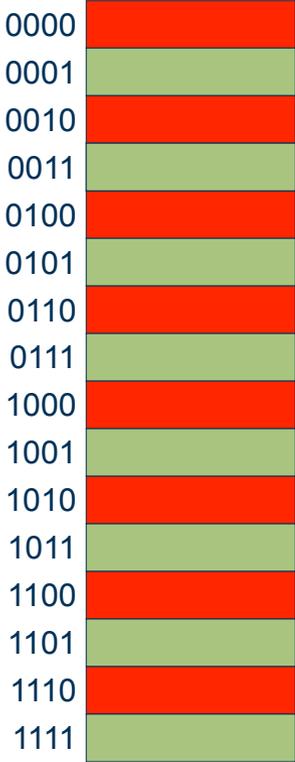
## 2-Way Set Associative Cache

- 4 cache lines are organized into two sets; each set has 2 cache lines (i.e., 2 ways)
- Lowest bit is used for cache index
  - Even address go to first set and odd addresses go to the second set
- Each address can be mapped to either cache line in the same set
  - Tag now stores the higher 3 bits instead of the entire address

# Set-Associative Cache

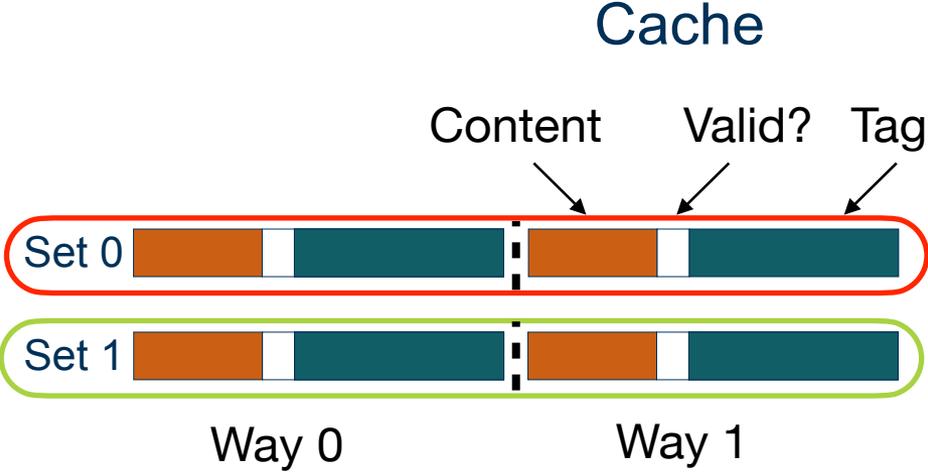


## Memory

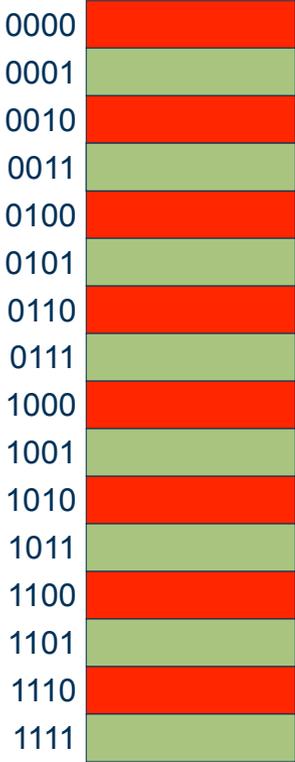


- Given a request to address, say 1011, from the CPU, detecting cache hit/miss requires:

# Set-Associative Cache

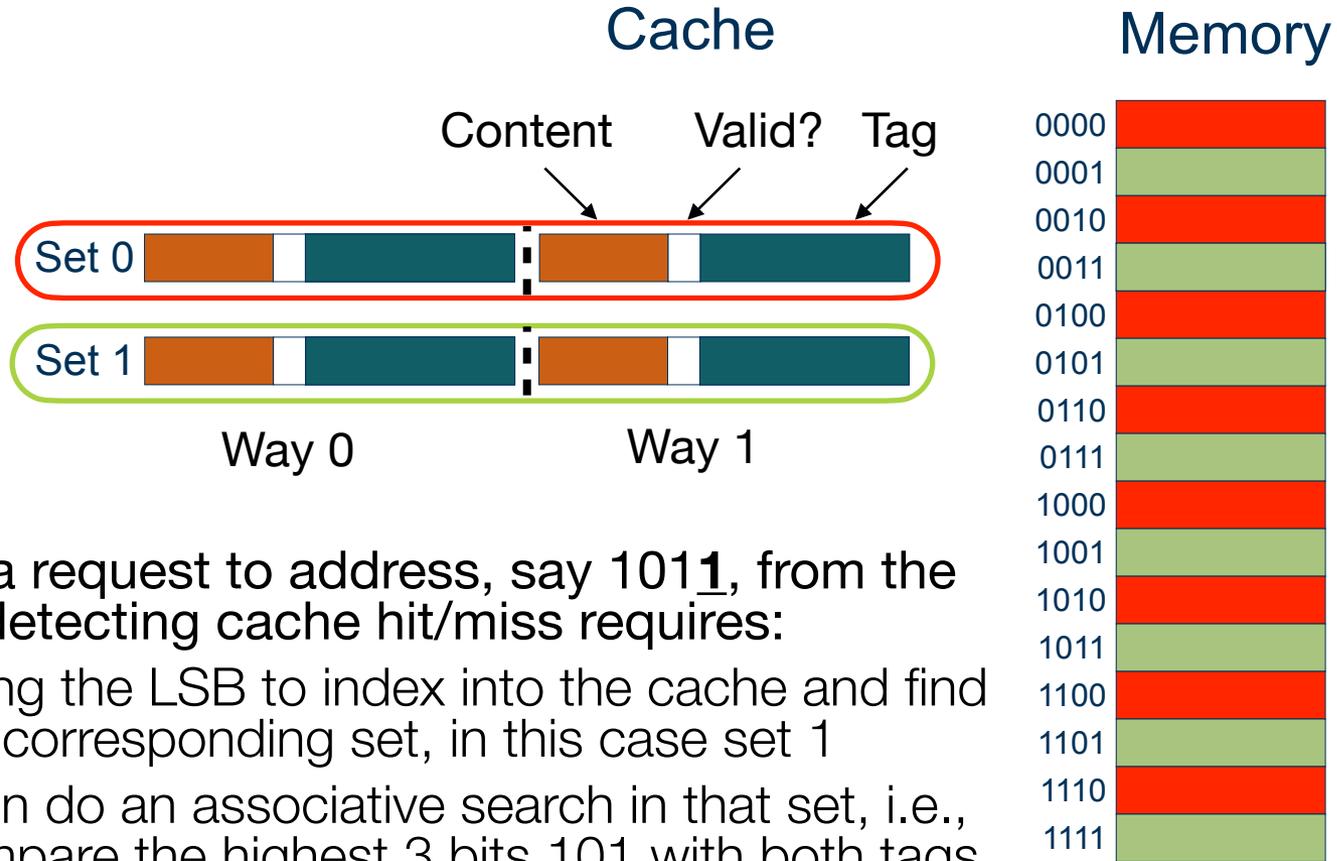


## Memory



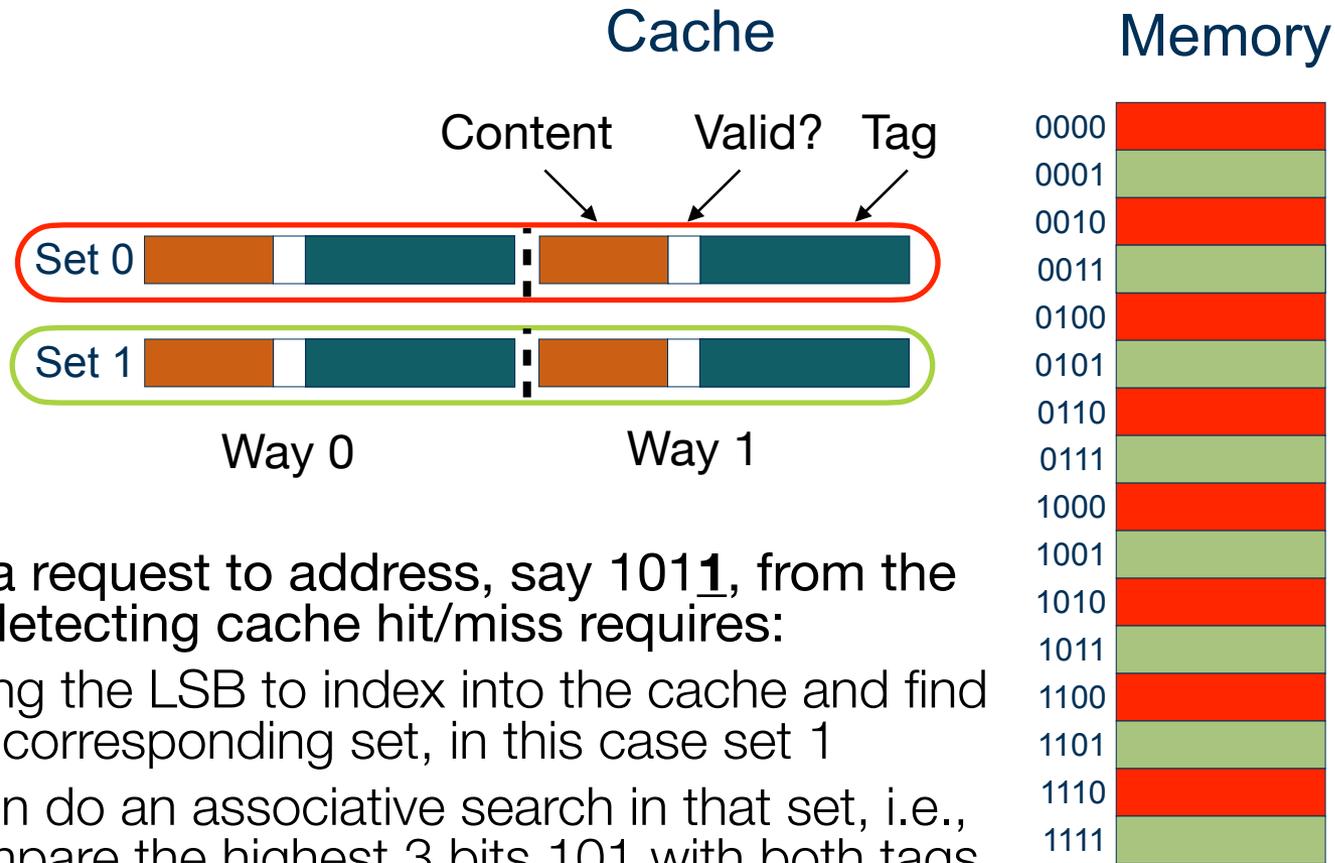
- Given a request to address, say 1011, from the CPU, detecting cache hit/miss requires:
  - Using the LSB to index into the cache and find the corresponding set, in this case set 1

# Set-Associative Cache



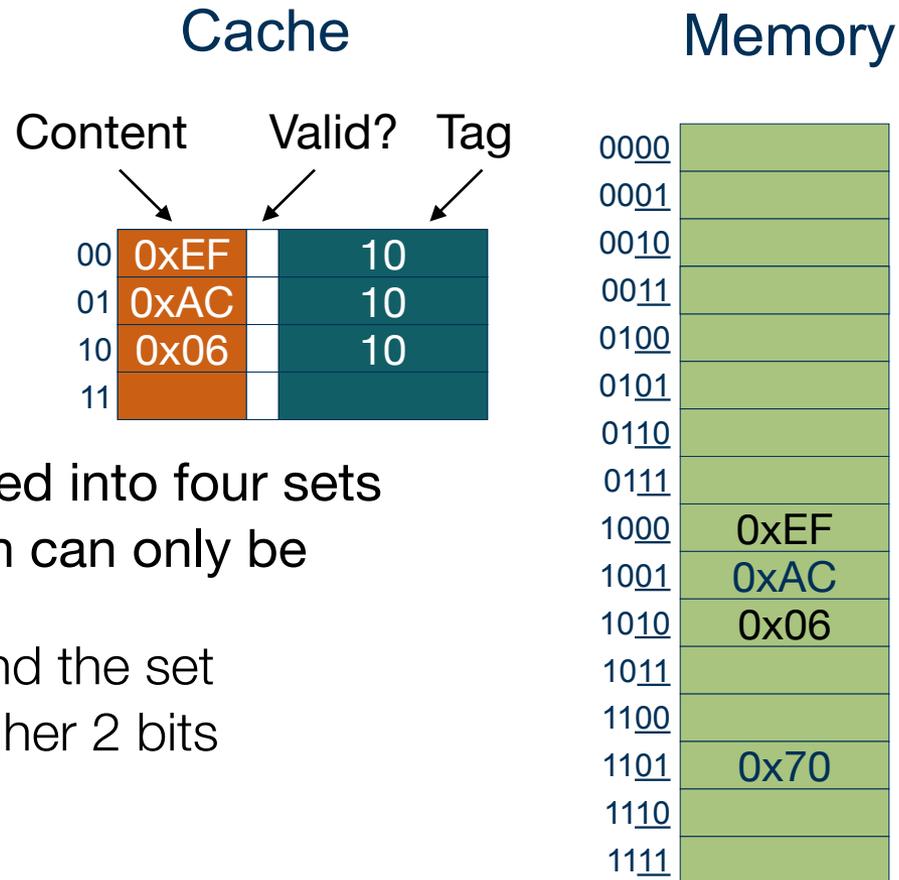
- Given a request to address, say 1011, from the CPU, detecting cache hit/miss requires:
  - Using the LSB to index into the cache and find the corresponding set, in this case set 1
  - Then do an associative search in that set, i.e., compare the highest 3 bits 101 with both tags in set 1

# Set-Associative Cache



- Given a request to address, say 1011, from the CPU, detecting cache hit/miss requires:
  - Using the LSB to index into the cache and find the corresponding set, in this case set 1
  - Then do an associative search in that set, i.e., compare the highest 3 bits 101 with both tags in set 1
  - Only two comparisons required

# Direct-Mapped (1-way Associative) Cache



- 4 cache lines are organized into four sets
- Each memory localization can only be mapped to one set
  - Using the 2 LSBs to find the set
  - Tag now stores the higher 2 bits

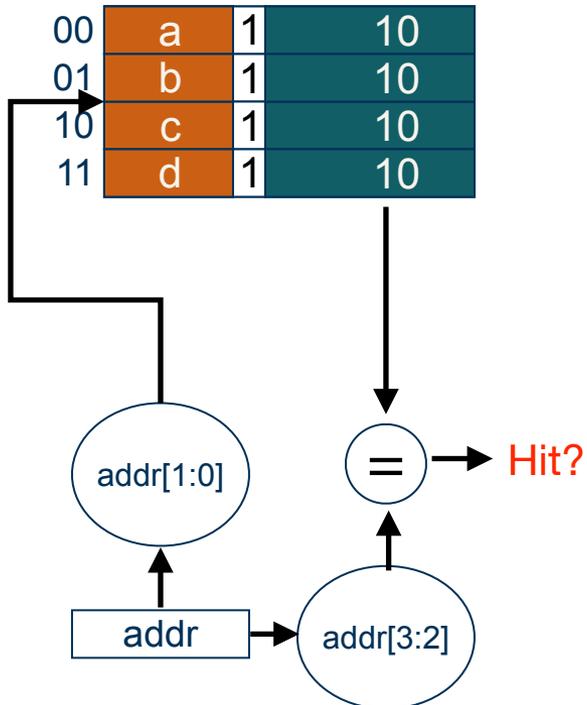
# Associative verses Direct Mapped Trade-offs

# Associative verses Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster

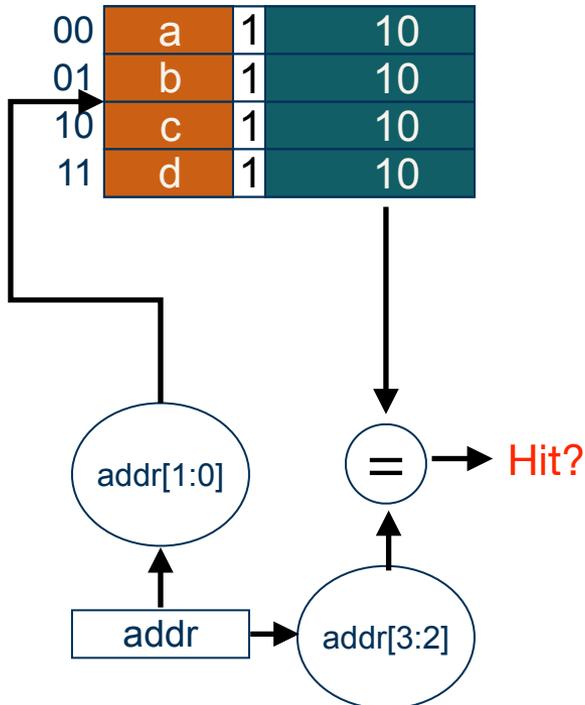
# Associative verses Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster



# Associative versus Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Set Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?

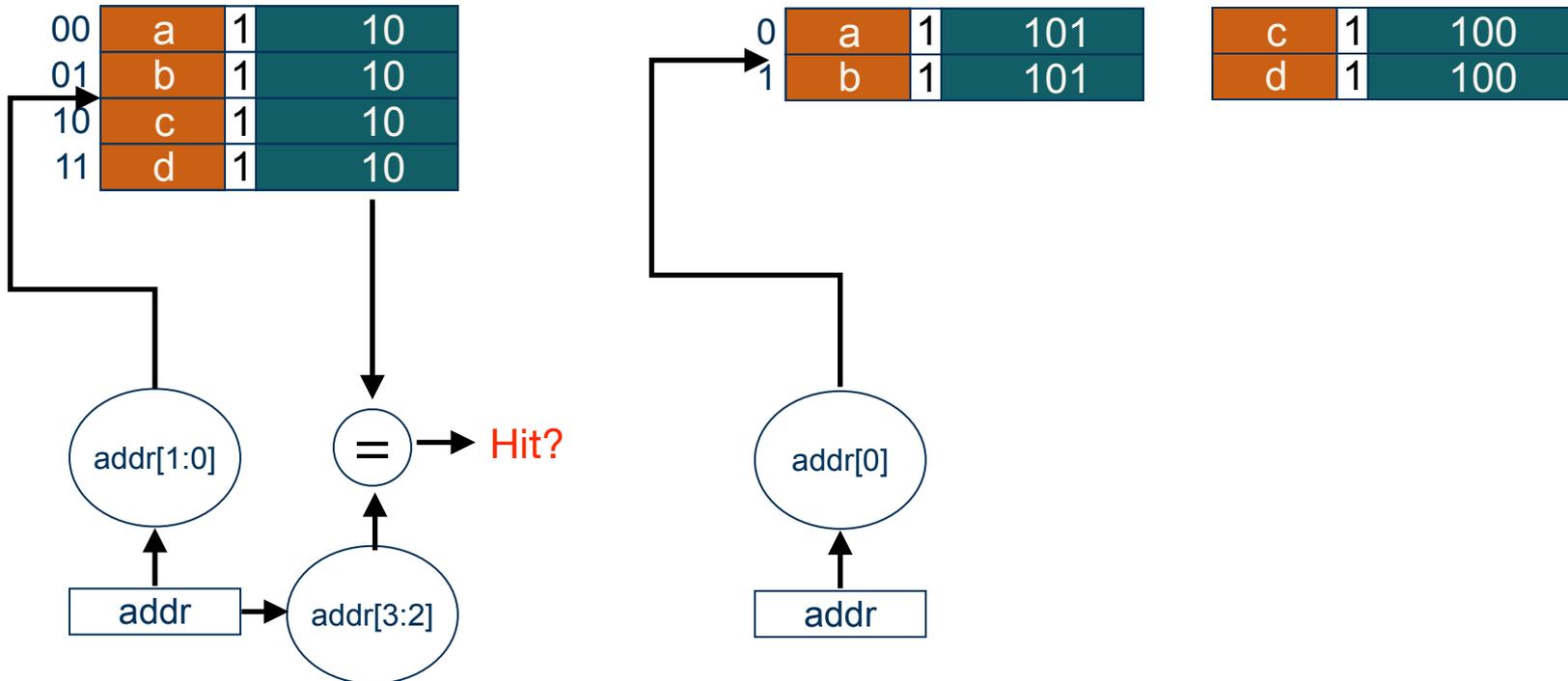


0	a	1	101
1	b	1	101

c	1	100
d	1	100

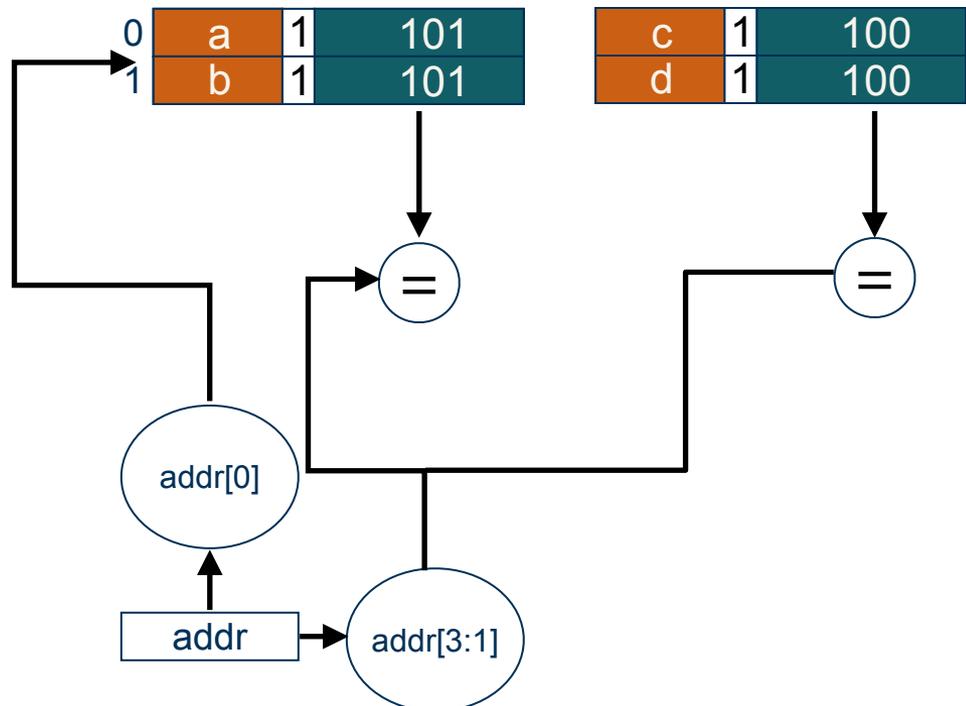
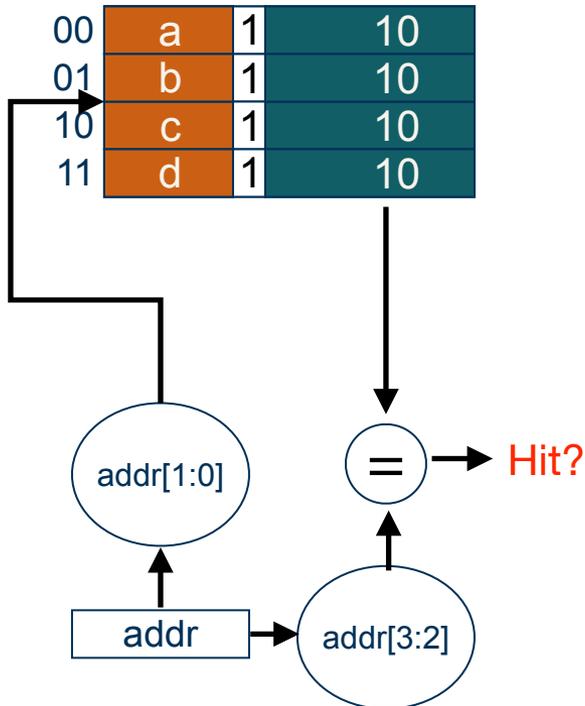
# Associative versus Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Set Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?



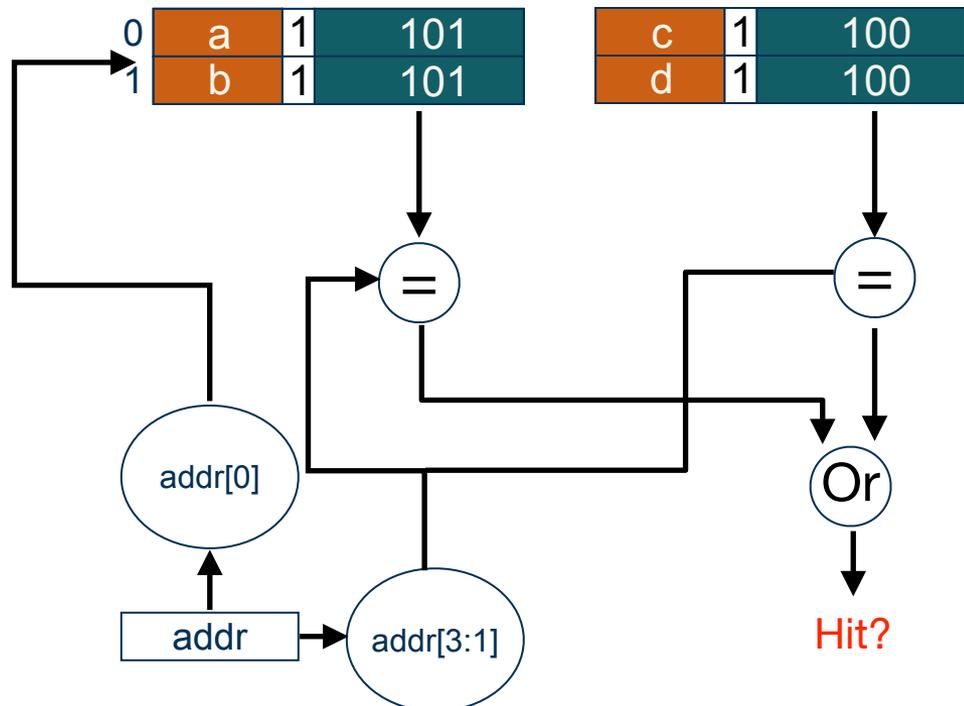
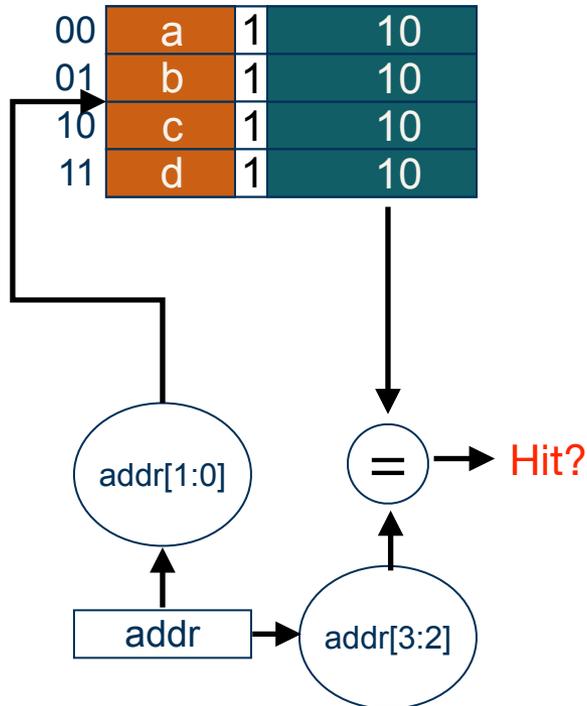
# Associative versus Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Set Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?

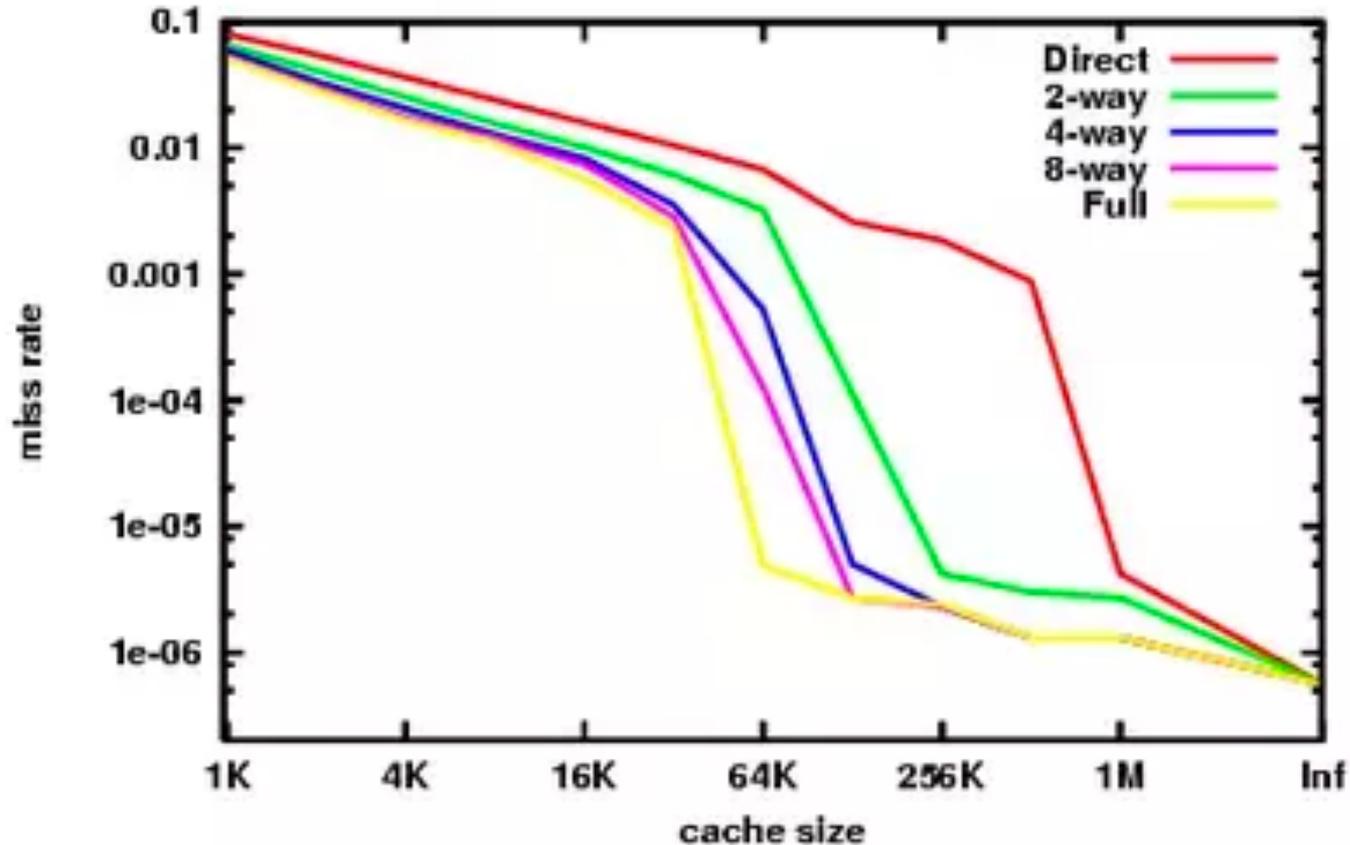


# Associative versus Direct Mapped Trade-offs

- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Set Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?



# Associative versus Direct Mapped Trade-offs



Miss rate versus cache size on the Integer portion of SPEC CPU2000

# Cache Organization

- Finding a name in a roster
- If the roster is completely unorganized
  - Need to compare the name with all the names in the roster
  - Same as a fully-associative cache
- If the roster is ordered by last name, and within the same last name different first names are unordered
  - First find the last name group
  - Then compare the first name with all the first names in the same group
  - Same as a set-associative cache

# Cache Access Summary (So far...)

- Assuming  $b$  bits in a memory address
- The  $b$  bits are split into two halves:
  - Lower  $s$  bits used as index to find a set. Total sets  $S = 2^s$
  - The higher  $(b - s)$  bits are used for the tag
- Associativity  $n$  (i.e., the number of ways in a cache set) is **independent** of the the split between index and tag

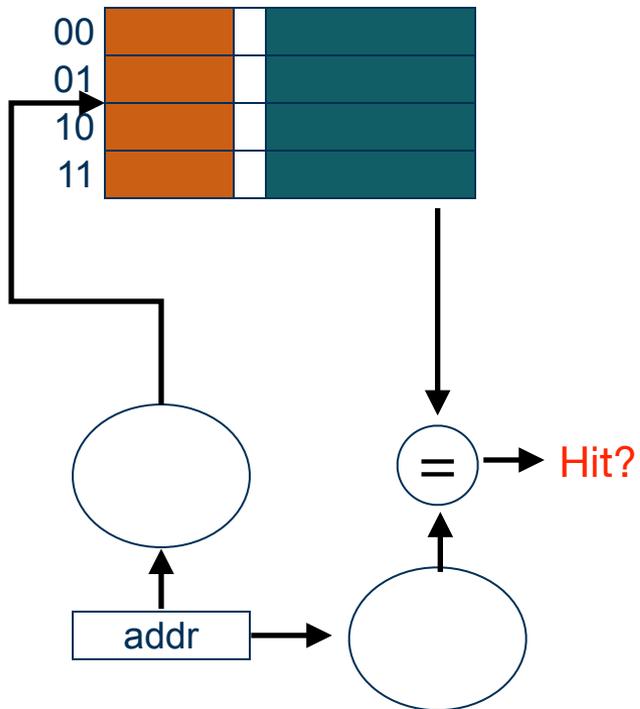


# Locality again

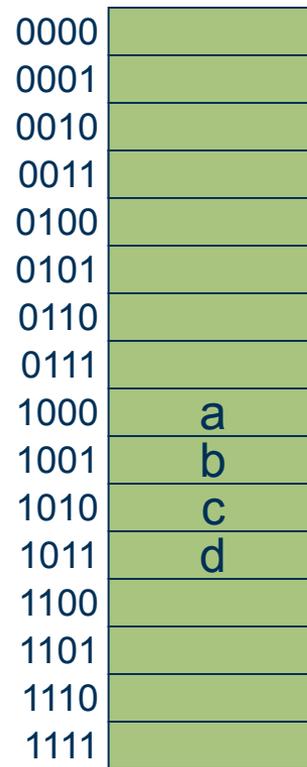
- So far: temporal locality
- What about spatial?
- Idea: Each cache location (cache line) store multiple bytes

# Cache-Line Size of 2

Cache

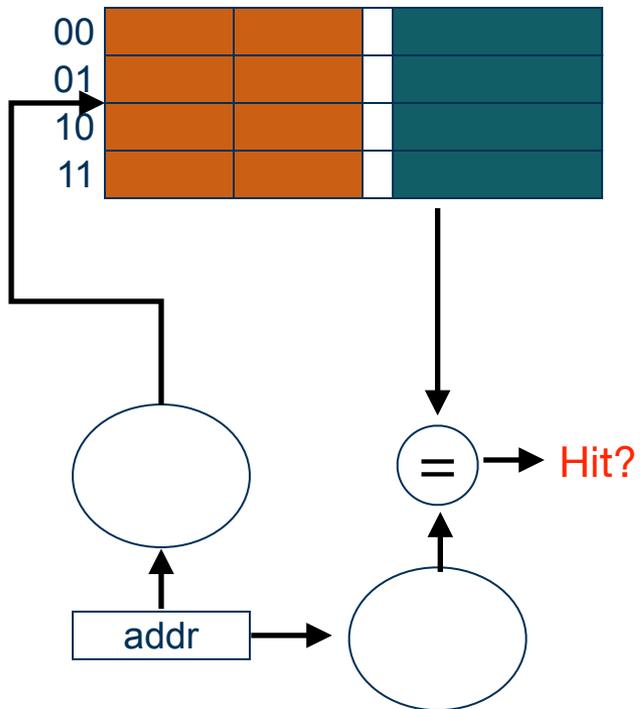


Memory

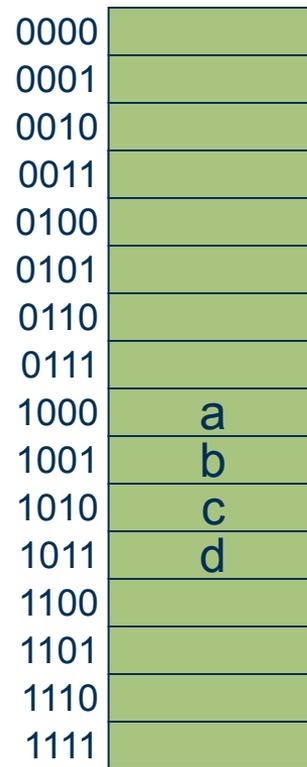


# Cache-Line Size of 2

Cache

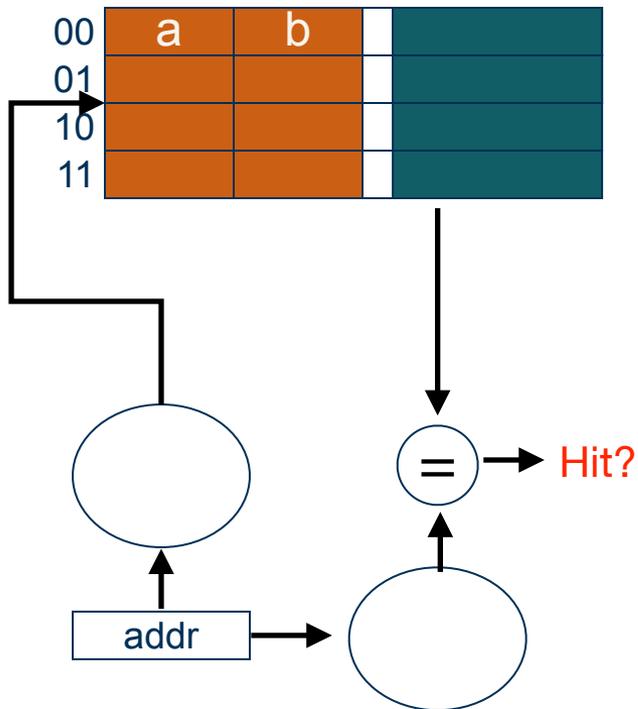


Memory

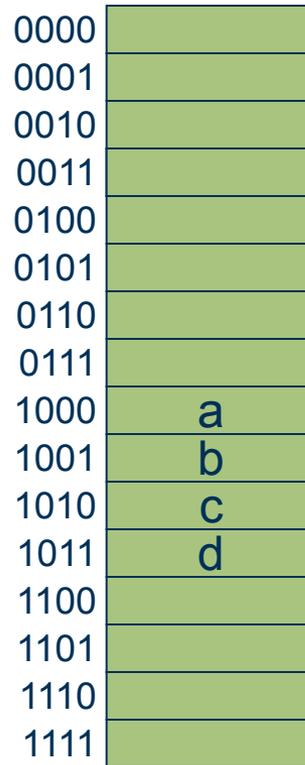


# Cache-Line Size of 2

Cache



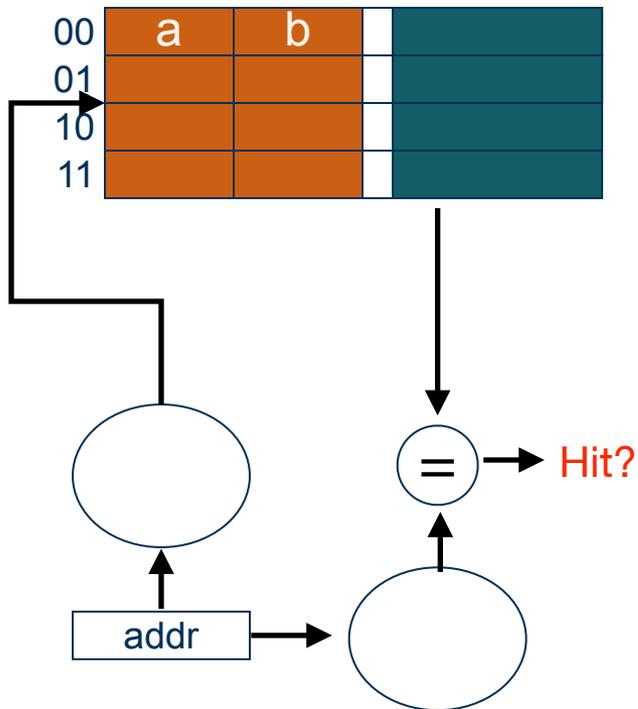
Memory



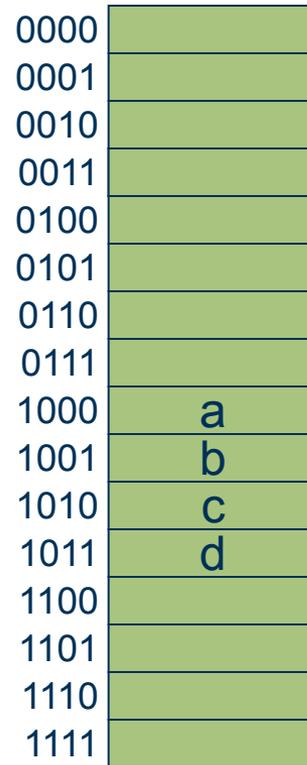
- Read 1000

# Cache-Line Size of 2

Cache



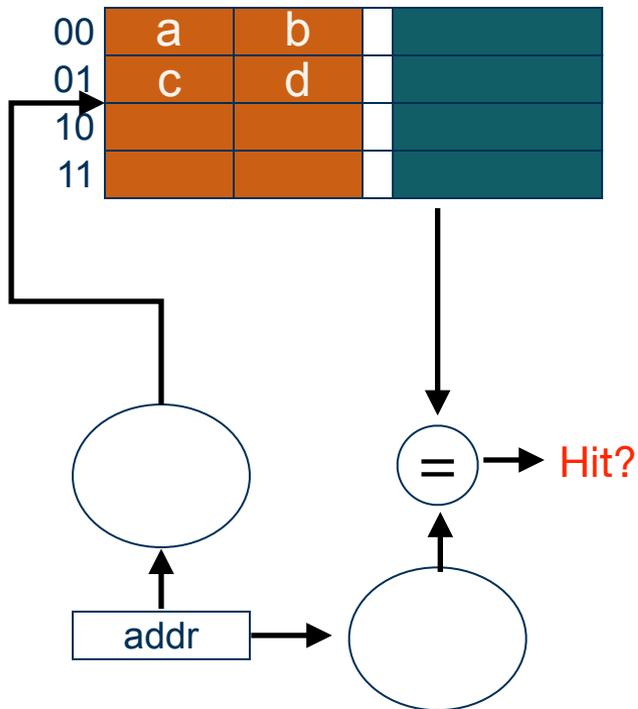
Memory



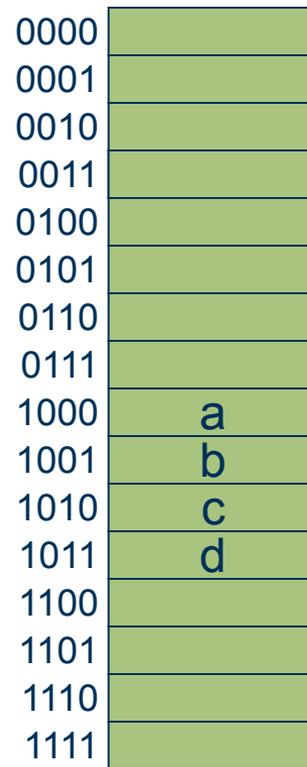
- Read 1000
- Read 1001 (**Hit!**)

# Cache-Line Size of 2

Cache



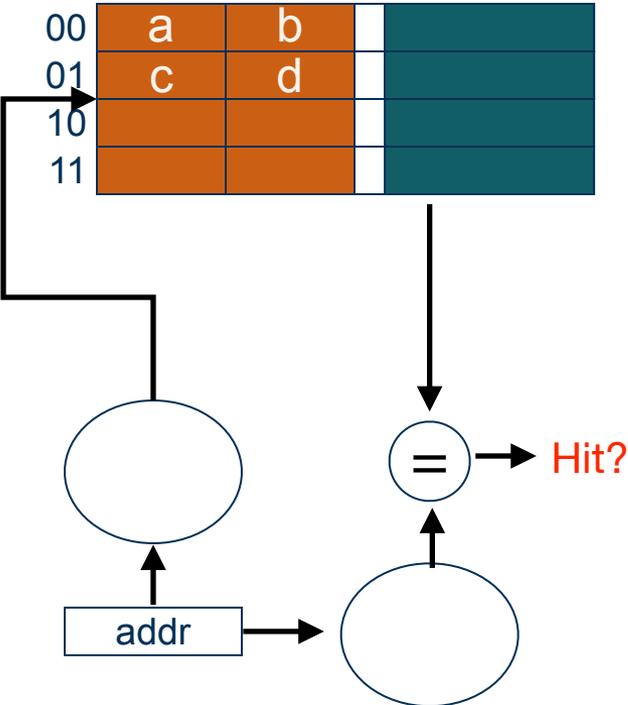
Memory



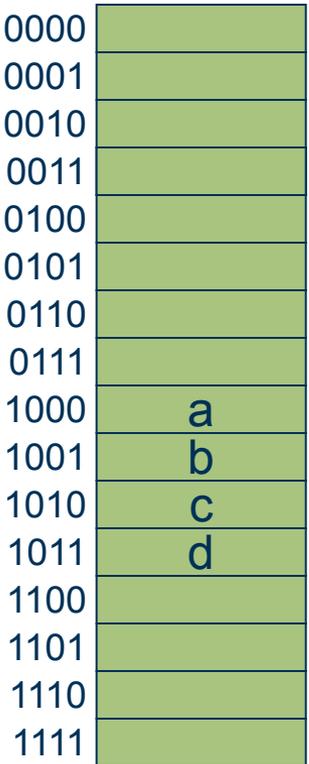
- Read 1000
- Read 1001 (Hit!)
- Read 1010

# Cache-Line Size of 2

Cache



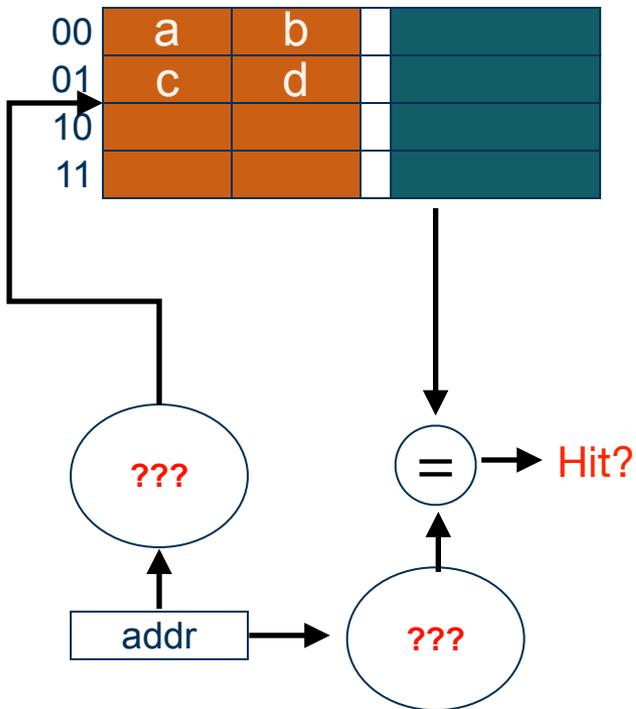
Memory



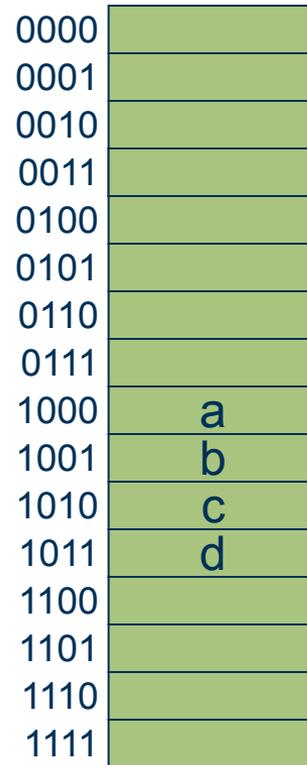
- Read 1000
- Read 1001 (Hit!)
- Read 1010
- Read 1011 (Hit!)

# Cache-Line Size of 2

Cache



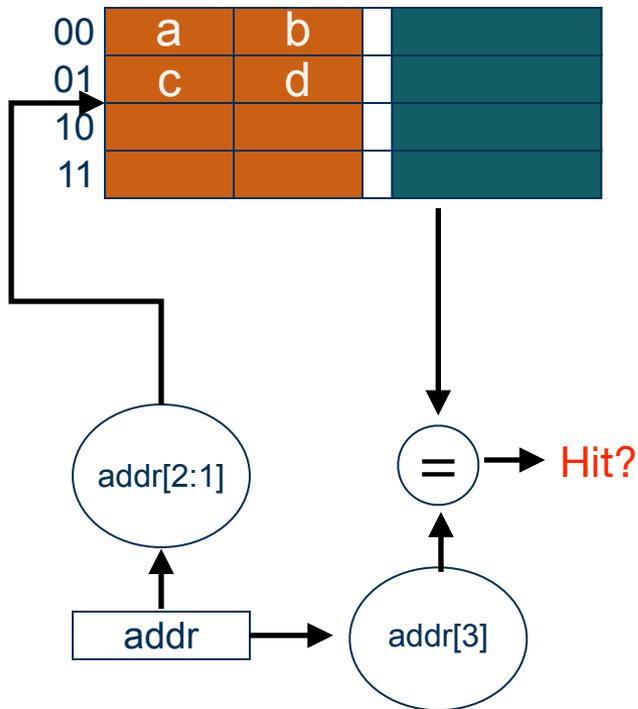
Memory



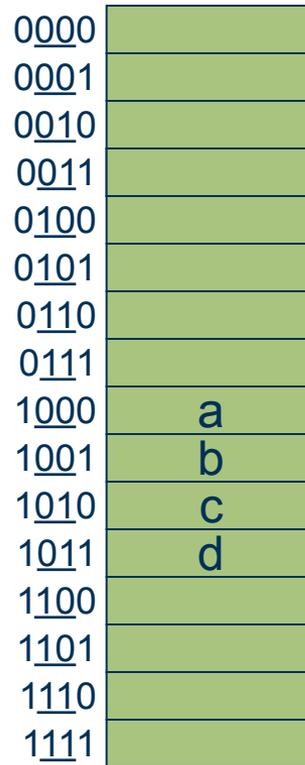
- Read 1000
- Read 1001 (Hit!)
- Read 1010
- Read 1011 (Hit!)
- How to access the cache now?

# Cache-Line Size of 2

Cache



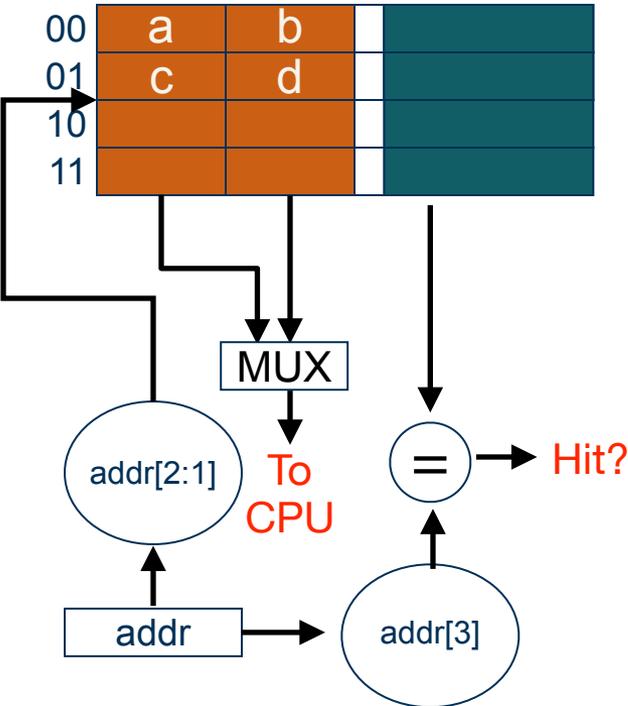
Memory



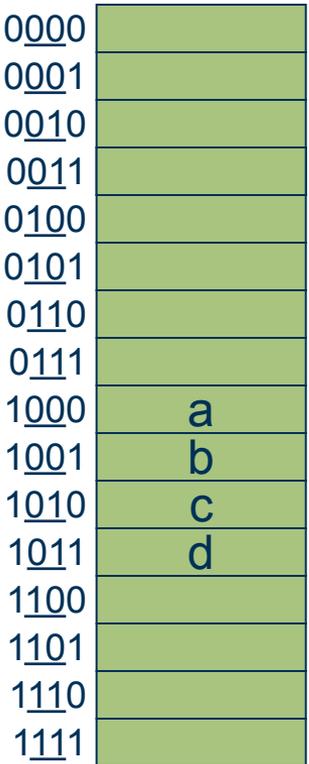
- Read 1000
- Read 1001 (Hit!)
- Read 1010
- Read 1011 (Hit!)

# Cache-Line Size of 2

Cache



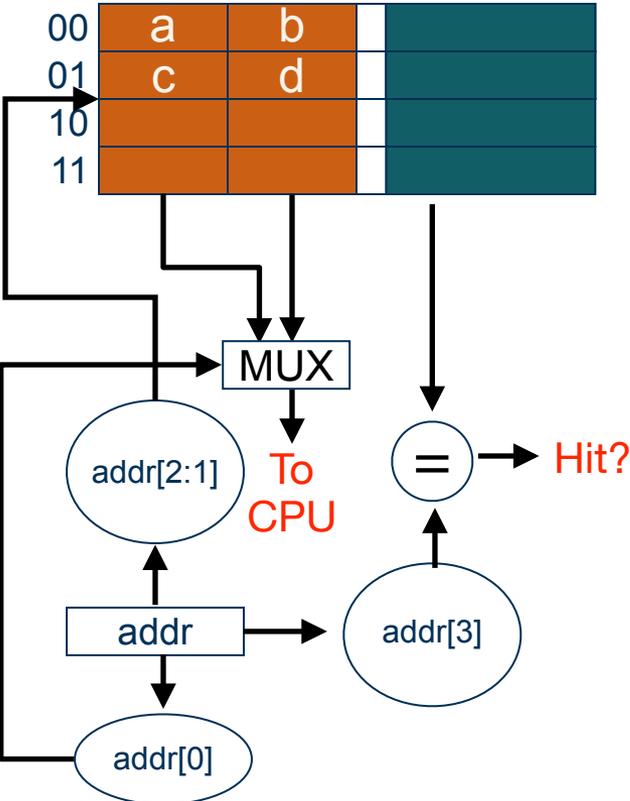
Memory



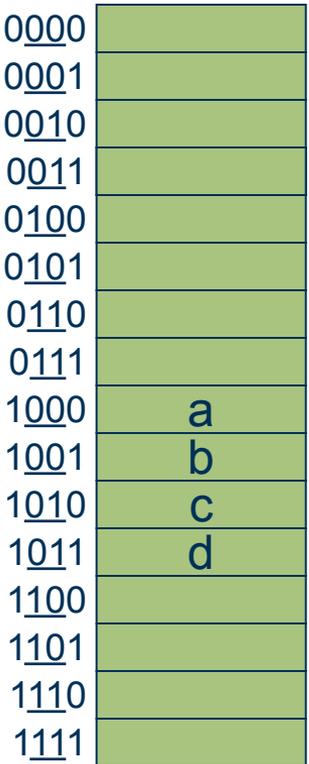
- Read 1000
- Read 1001 (Hit!)
- Read 1010
- Read 1011 (Hit!)

# Cache-Line Size of 2

Cache



Memory



- Read 1000
- Read 1001 (Hit!)
- Read 1010
- Read 1011 (Hit!)



# Handling Reads

# Handling Reads

- Read miss: Put into cache

# Handling Reads

- Read miss: Put into cache
  - Any reason not to put into cache?

# Handling Reads

- Read miss: Put into cache
  - Any reason not to put into cache?
  - What to replace? Depends on the replacement policy. More on this later.

# Handling Reads

- Read miss: Put into cache
  - Any reason not to put into cache?
  - What to replace? Depends on the replacement policy. More on this later.
- Read hit: Nothing special. Enjoy the hit!

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”
- Write-through

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”
- Write-through
  - + Simpler

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”
- Write-through
  - + Simpler
  - + Memory is up to date

# Handling Writes (Hit)

- Intricacy: data value is modified!
- Implication: value in cache will be different from that in memory!
- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the cache line is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction. Potentially saves bandwidth between cache and memory + saves energy
  - - Need a bit in the tag store indicating the block is “dirty/modified”
- Write-through
  - + Simpler
  - + Memory is up to date
  - - More bandwidth intensive; no coalescing of writes

# Handling Writes (Miss)

- Do we allocate a cache line on a write miss?
  - **Write-allocate**: Allocate on write miss
  - **Non-Write-Allocate**: No-allocate on write miss
- Allocate on write miss

# Handling Writes (Miss)

- Do we allocate a cache line on a write miss?
  - **Write-allocate**: Allocate on write miss
  - **Non-Write-Allocate**: No-allocate on write miss
- Allocate on write miss
  - + Can consolidate writes instead of writing each of them individually to memory

# Handling Writes (Miss)

- Do we allocate a cache line on a write miss?
  - **Write-allocate**: Allocate on write miss
  - **Non-Write-Allocate**: No-allocate on write miss
- Allocate on write miss
  - + Can consolidate writes instead of writing each of them individually to memory
  - + Simpler because write misses can be treated the same way as read misses

# Handling Writes (Miss)

- Do we allocate a cache line on a write miss?
  - **Write-allocate**: Allocate on write miss
  - **Non-Write-Allocate**: No-allocate on write miss
- Allocate on write miss
  - + Can consolidate writes instead of writing each of them individually to memory
  - + Simpler because write misses can be treated the same way as read misses
- Non-allocate
  - + Conserves cache space if locality of writes is low (potentially better cache hit rate)

# Instruction vs. Data Caches

- Separate or Unified?

# Instruction vs. Data Caches

- Separate or Unified?
- Unified:

# Instruction vs. Data Caches

- Separate or Unified?
- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split Inst and Data caches)

# Instruction vs. Data Caches

- Separate or Unified?
- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split Inst and Data caches)
  - - Instructions and data can thrash each other (i.e., no guaranteed space for either)

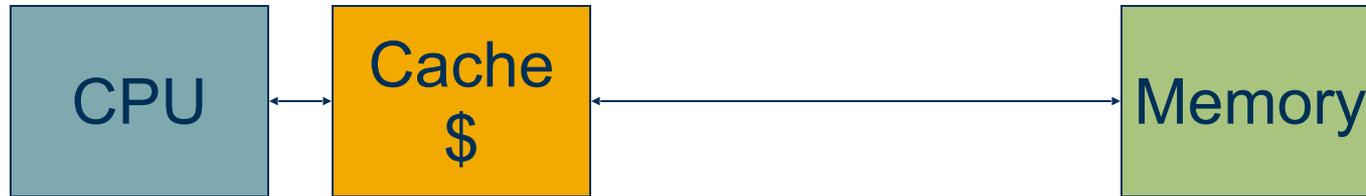
# Instruction vs. Data Caches

- Separate or Unified?
- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split Inst and Data caches)
  - - Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - - Inst and Data are accessed in different places in the pipeline.  
Where do we place the unified cache for fast access?

# Instruction vs. Data Caches

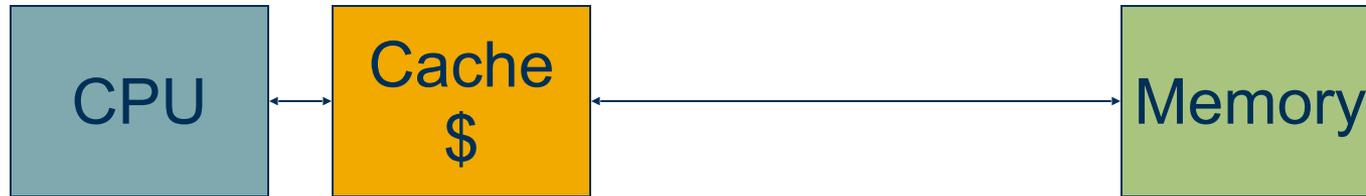
- Separate or Unified?
- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split Inst and Data caches)
  - - Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - - Inst and Data are accessed in different places in the pipeline.  
Where do we place the unified cache for fast access?
- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

# General Rule: Bigger == Slower



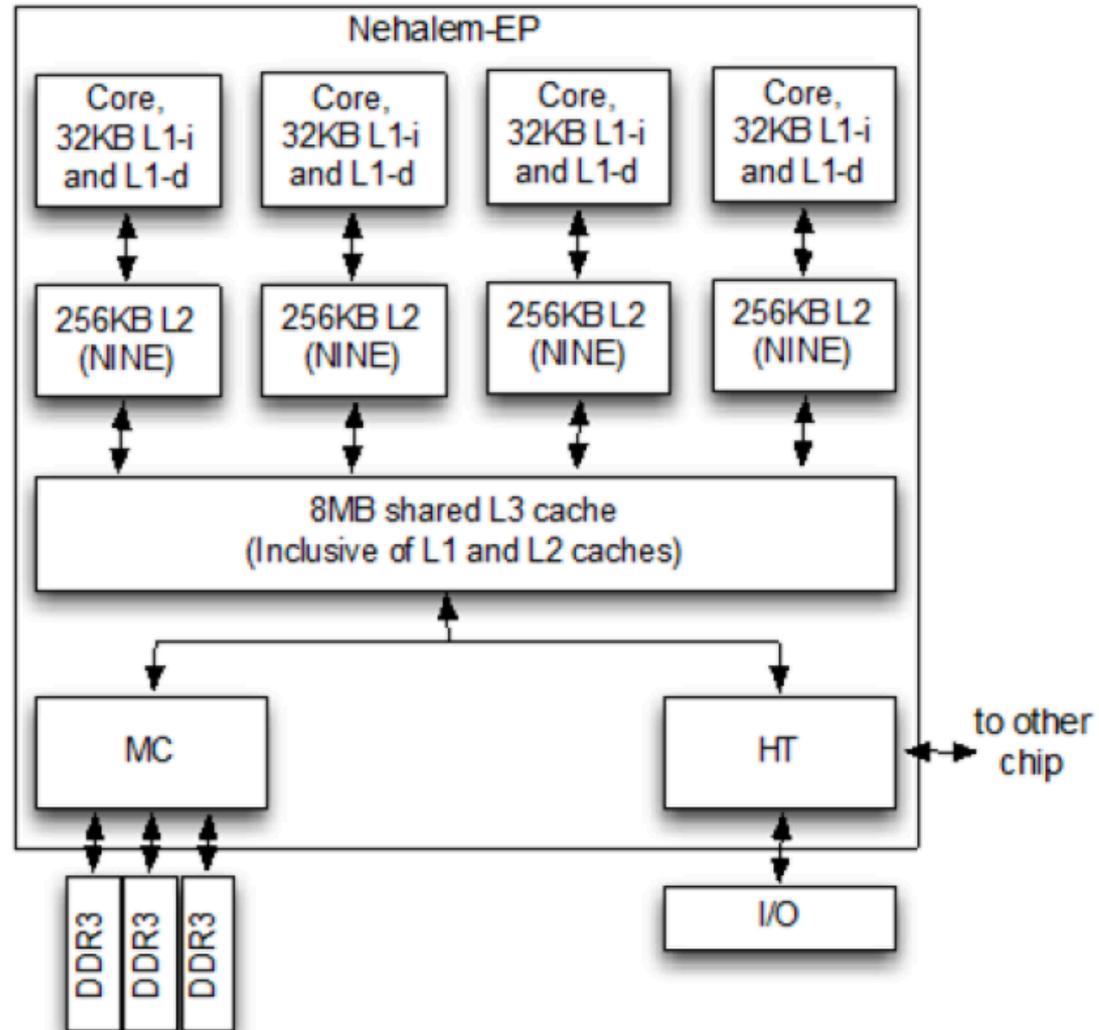
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)

# General Rule: Bigger == Slower



- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

# A Real Intel Processor



# Eviction/Replacement Policy

- Which cache line should be replaced?

# Eviction/Replacement Policy

- Which cache line should be replaced?
  - Direct mapped? Only one place!

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first
  - If all are valid, consult the **replacement policy**

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first
  - If all are valid, consult the **replacement policy**
  - Randomly pick one???

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first
  - If all are valid, consult the **replacement policy**
  - Randomly pick one???
  - Ideally: Replace the cache line that's least likely going to be used again

# Eviction/Replacement Policy



- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first
  - If all are valid, consult the **replacement policy**
  - Randomly pick one???
  - Ideally: Replace the cache line that's least likely going to be used again
    - Approximation: Least recently used (LRU)

# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?

Cache Lines

0

1

LRU index (1-bit)



# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?

Cache Lines



LRU index (1-bit)

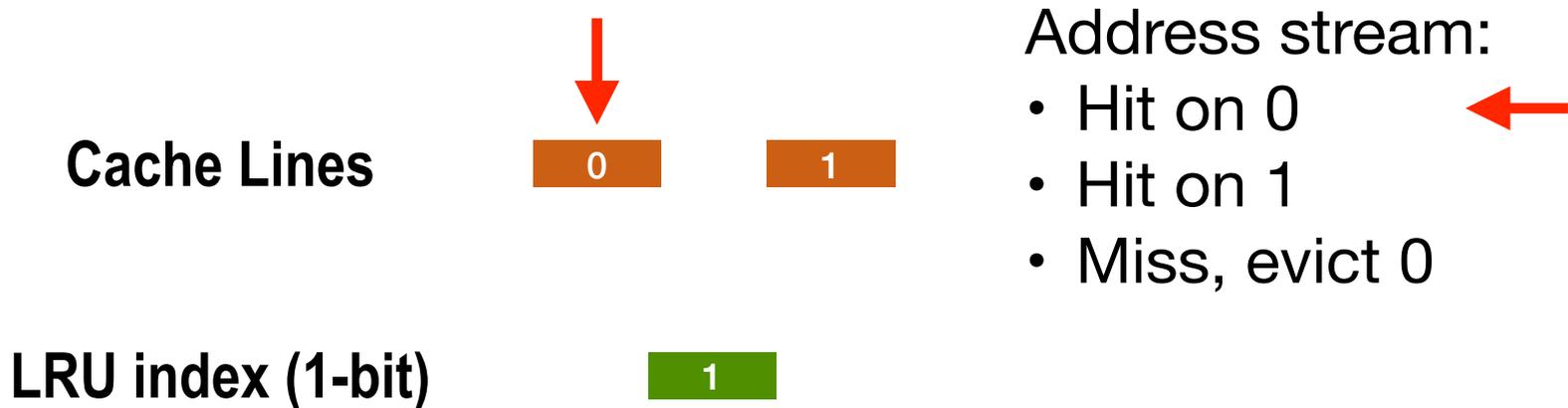


Address stream:

- Hit on 0
- Hit on 1
- Miss, evict 0

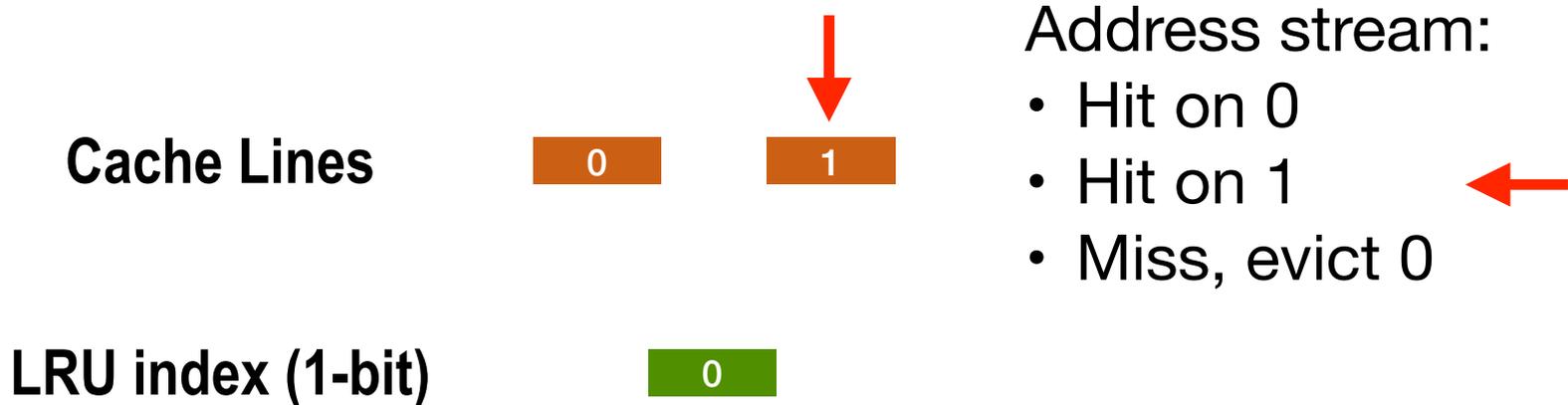
# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?



# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?



# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?

Cache Lines



LRU index (1-bit)



Address stream:

- Hit on 0
- Hit on 1
- Miss, evict 0 ←

# Implementing LRU

- Idea: Evict the least recently accessed block
- Challenge: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly? One bit?

Cache Lines



LRU index (1-bit)

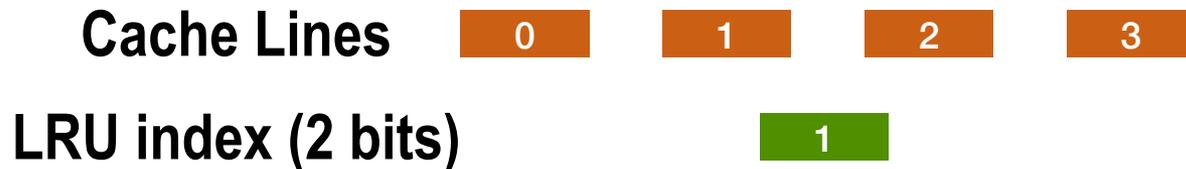


Address stream:

- Hit on 0
- Hit on 1
- Miss, evict 0 ←

# Implementing LRU

- Question: 4-way set associative cache:



Address stream:

- Hit on 0
- Hit on 2
- Hit on 3
- Miss, evict 1

# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?

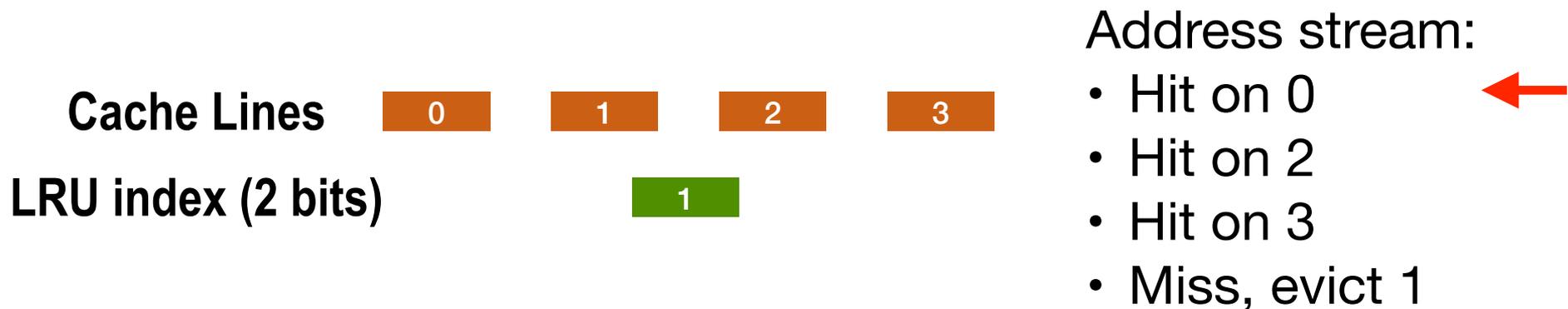


Address stream:

- Hit on 0
- Hit on 2
- Hit on 3
- Miss, evict 1

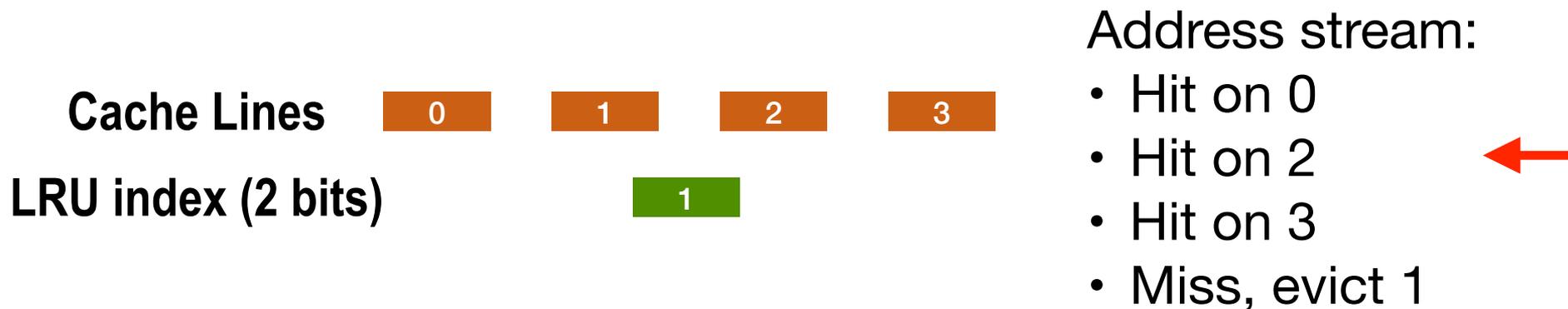
# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?



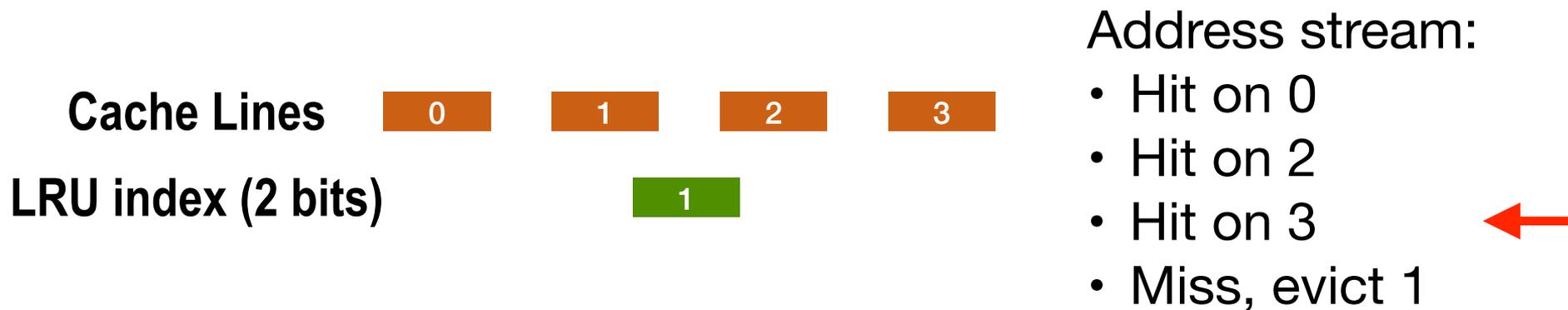
# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?



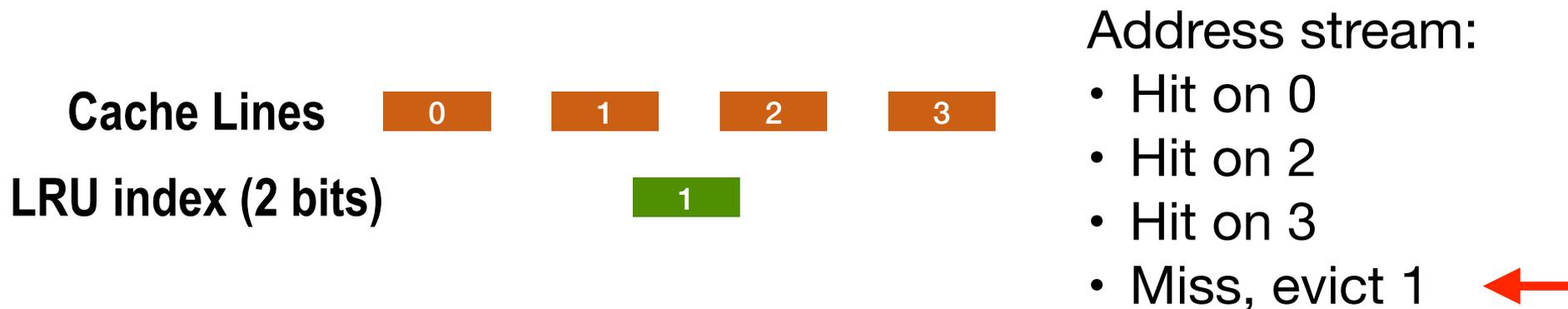
# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?
  - Essentially have to track the ordering of all cache lines



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?
  - Essentially have to track the ordering of all cache lines
  - What are the hardware structures needed?



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?
  - Essentially have to track the ordering of all cache lines
  - What are the hardware structures needed?
  - In reality, true LRU is never implemented. Too complex.



# Implementing LRU

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly? Will the same mechanism work?
  - Essentially have to track the ordering of all cache lines
  - What are the hardware structures needed?
  - In reality, true LRU is never implemented. Too complex.
  - “Pseudo-LRU” is usually used in real processors.

