CSC 252: Computer Organization
Spring 2020: Lecture 17

Instructor: Yuhao Zhu

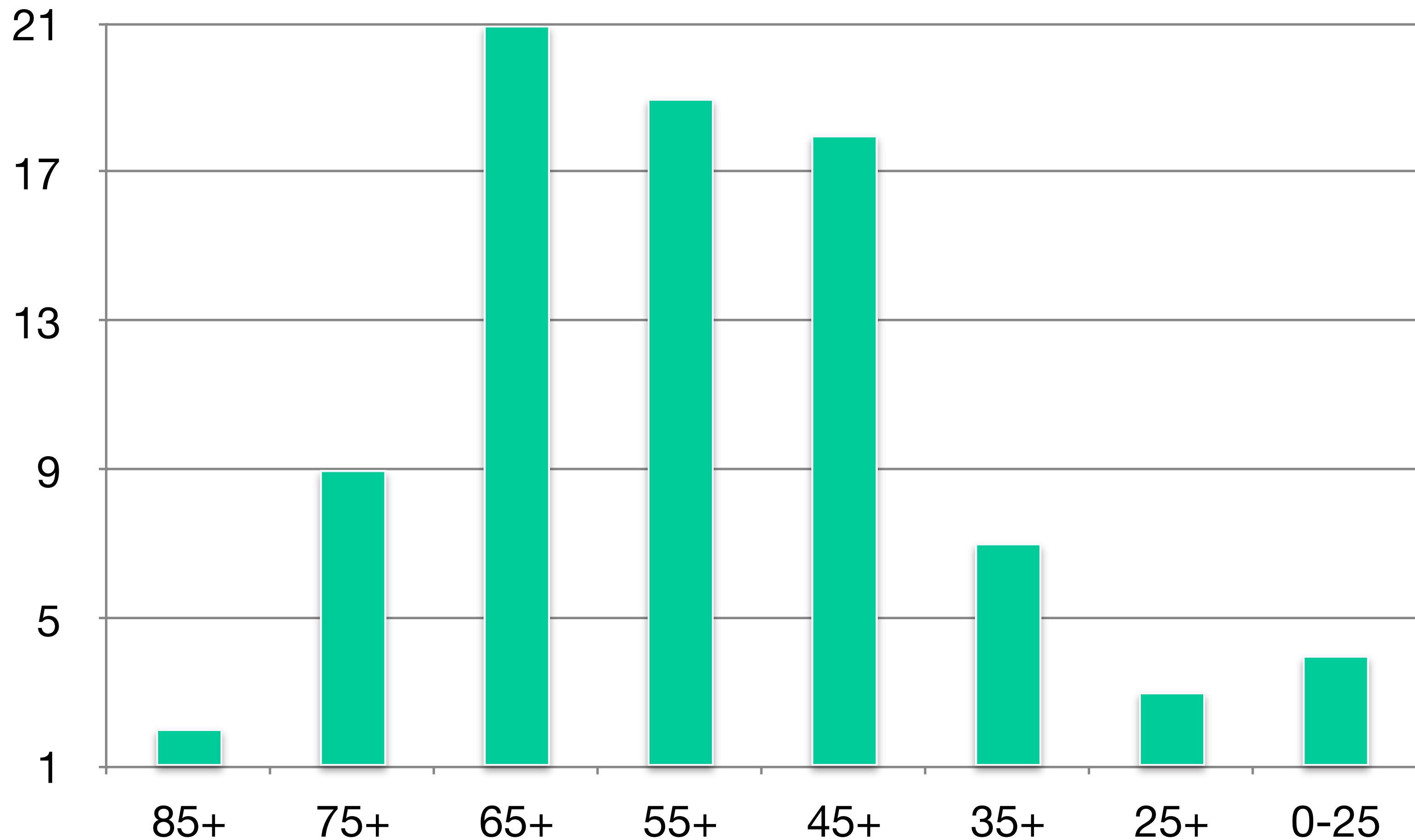
Department of Computer Science
University of Rochester

Announcement

- Recall: 75 full score + 20 extra credit
- Max: 90
- Min: 11.3
- Median: 58.25
- Mean: 57.71
- Standard Deviation: 16.23

Announcement

- Mid-term grade distribution



Announcement

Announcement

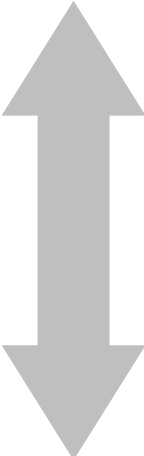
- Point distribution:
  - 8% per lab: 40% in total
  - 22% for mid-term
  - 38% for final

Announcement

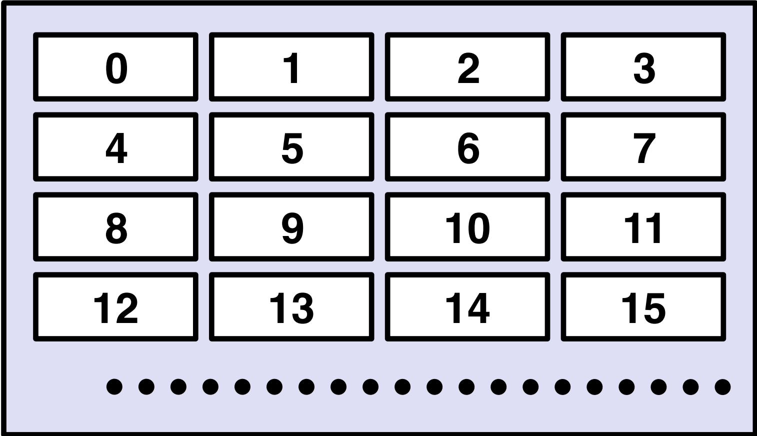
- Point distribution:
  - 8% per lab: 40% in total
  - 22% for mid-term
  - 38% for final
- If you can’t make an office hour (mine or TAs), feel free to email and/or schedule a different time at your convenience

Cache Illustrations

CPU

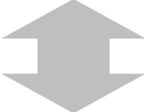


Memory
(big but slow)

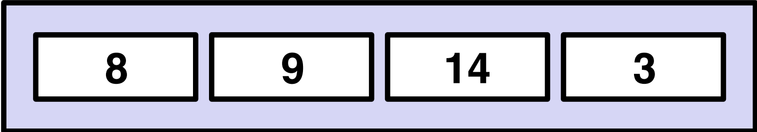


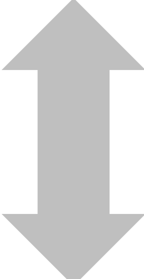
Cache Illustrations

CPU

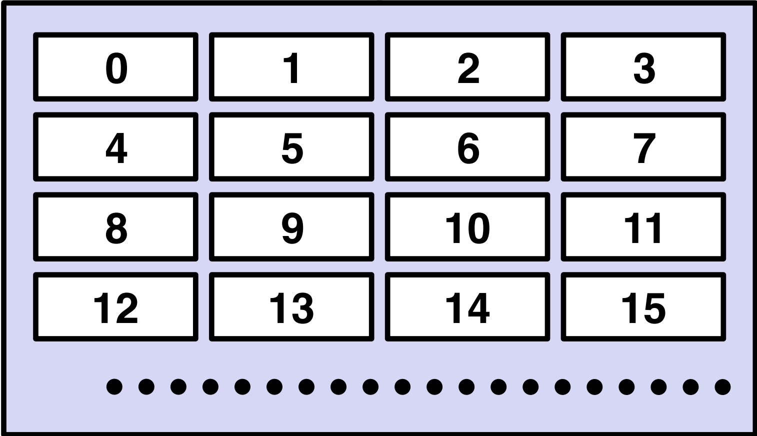


Cache
(small but fast)

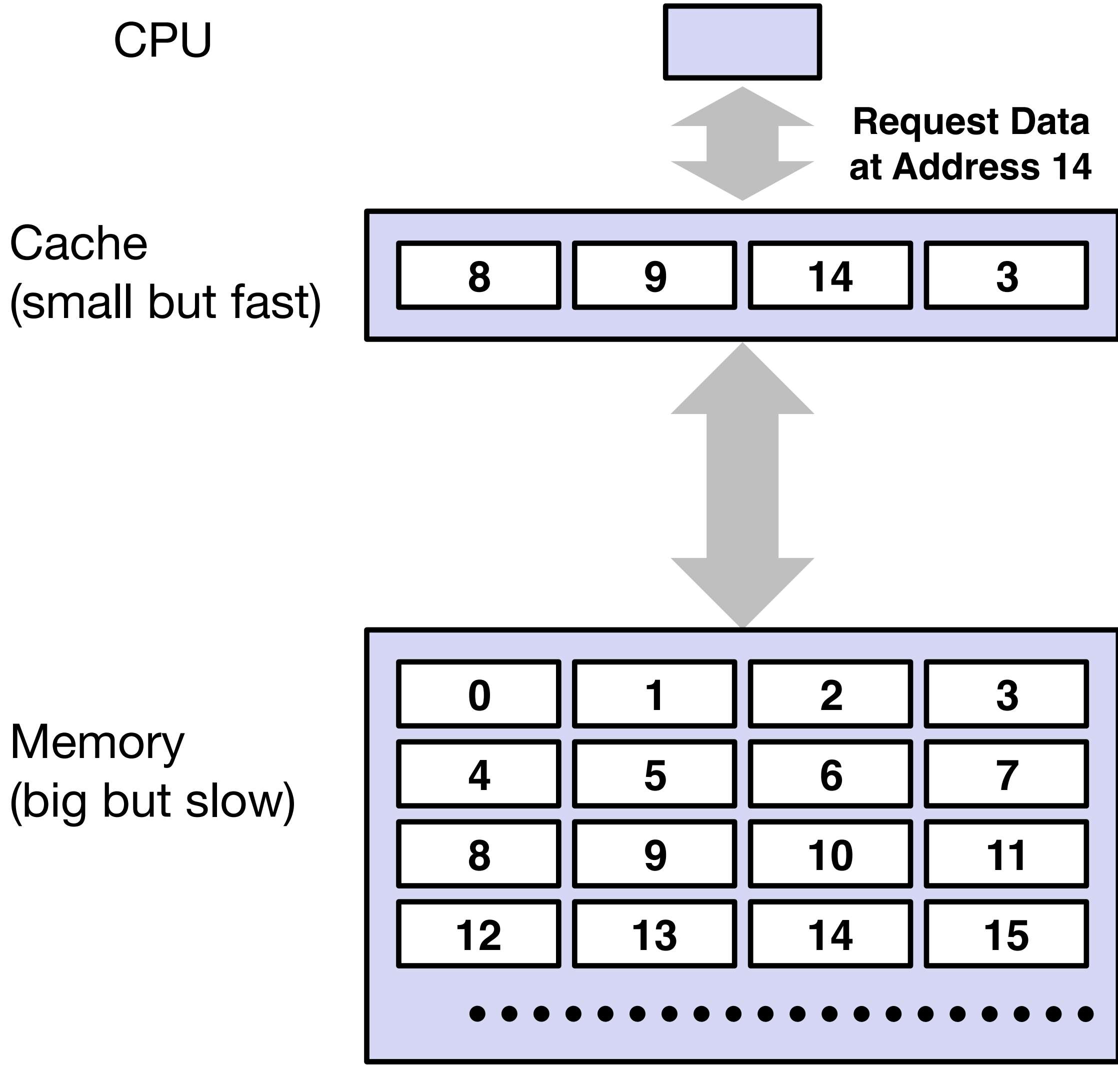




Memory
(big but slow)

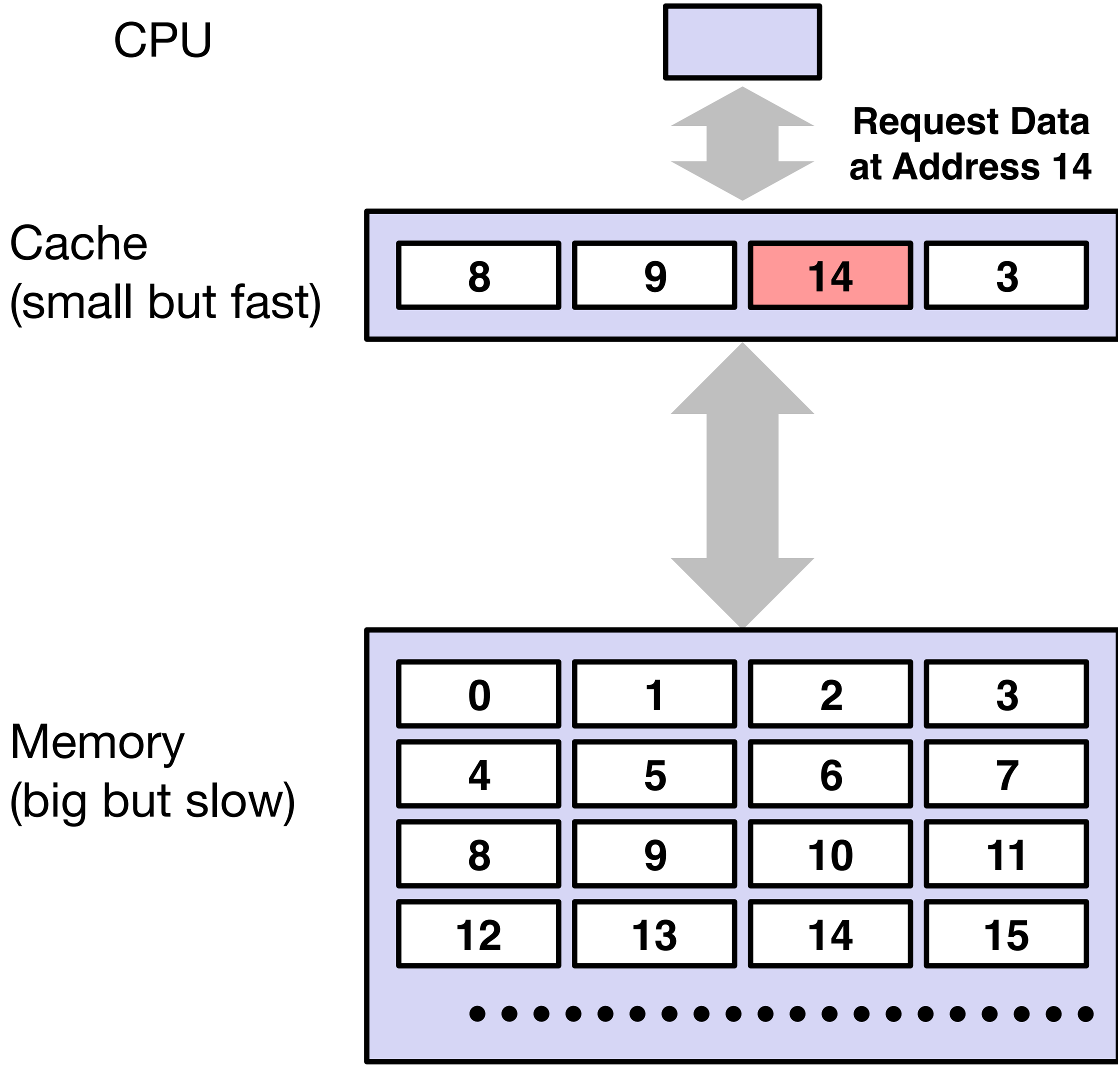


Cache Illustrations



Data in address b is needed

Cache Illustrations

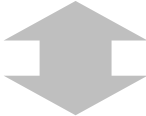


Data in address b is needed

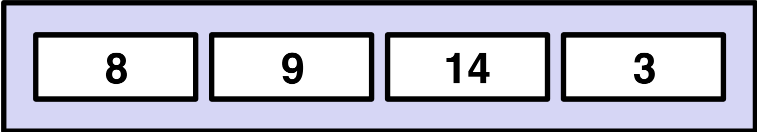
Address b is in cache: Hit!

Cache Illustrations

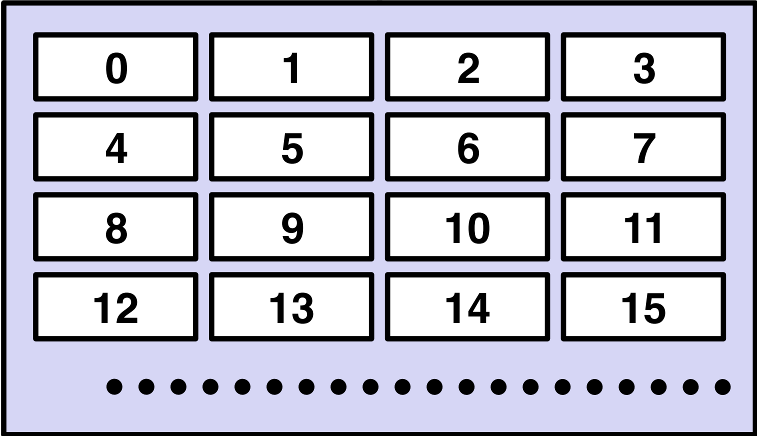
CPU



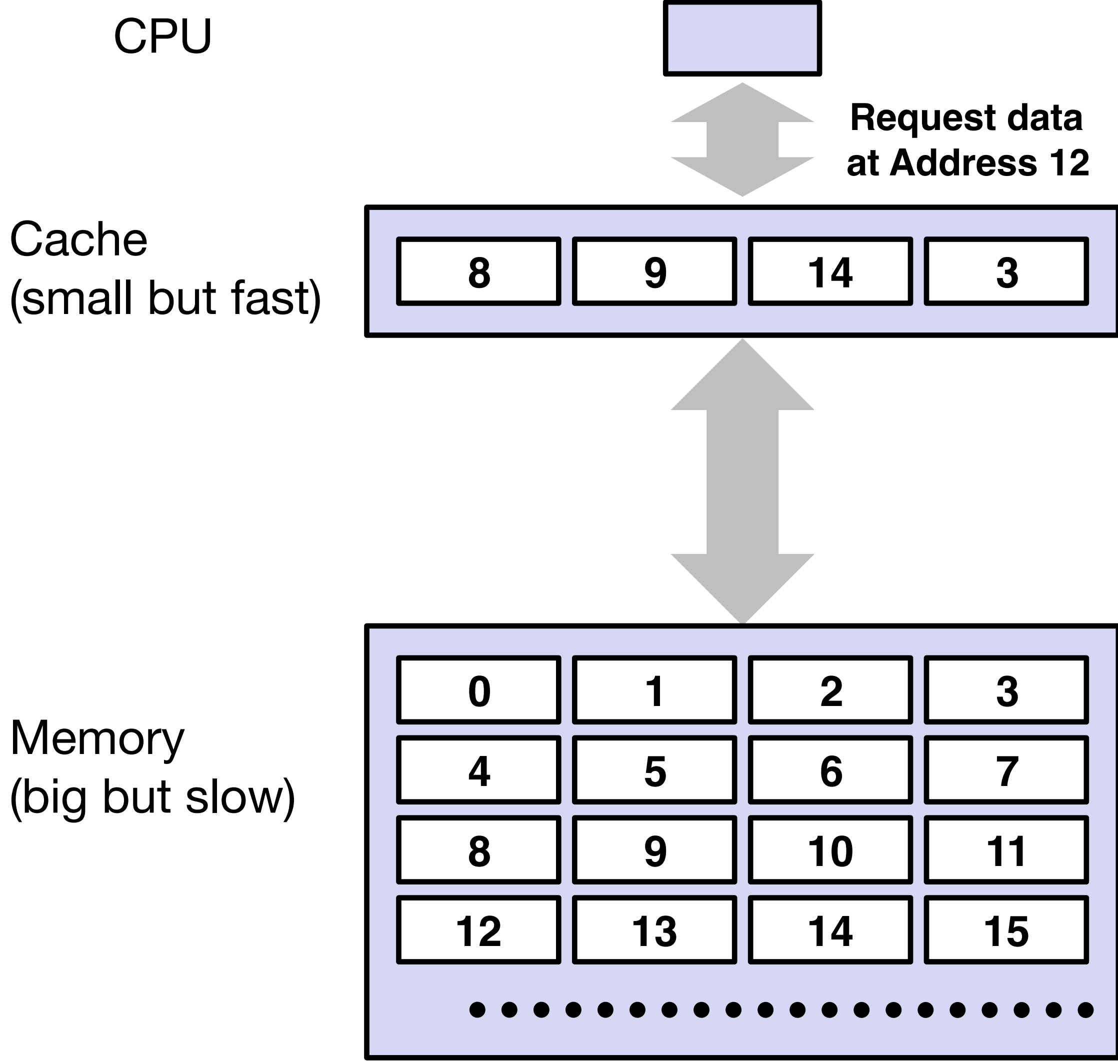
Cache
(small but fast)



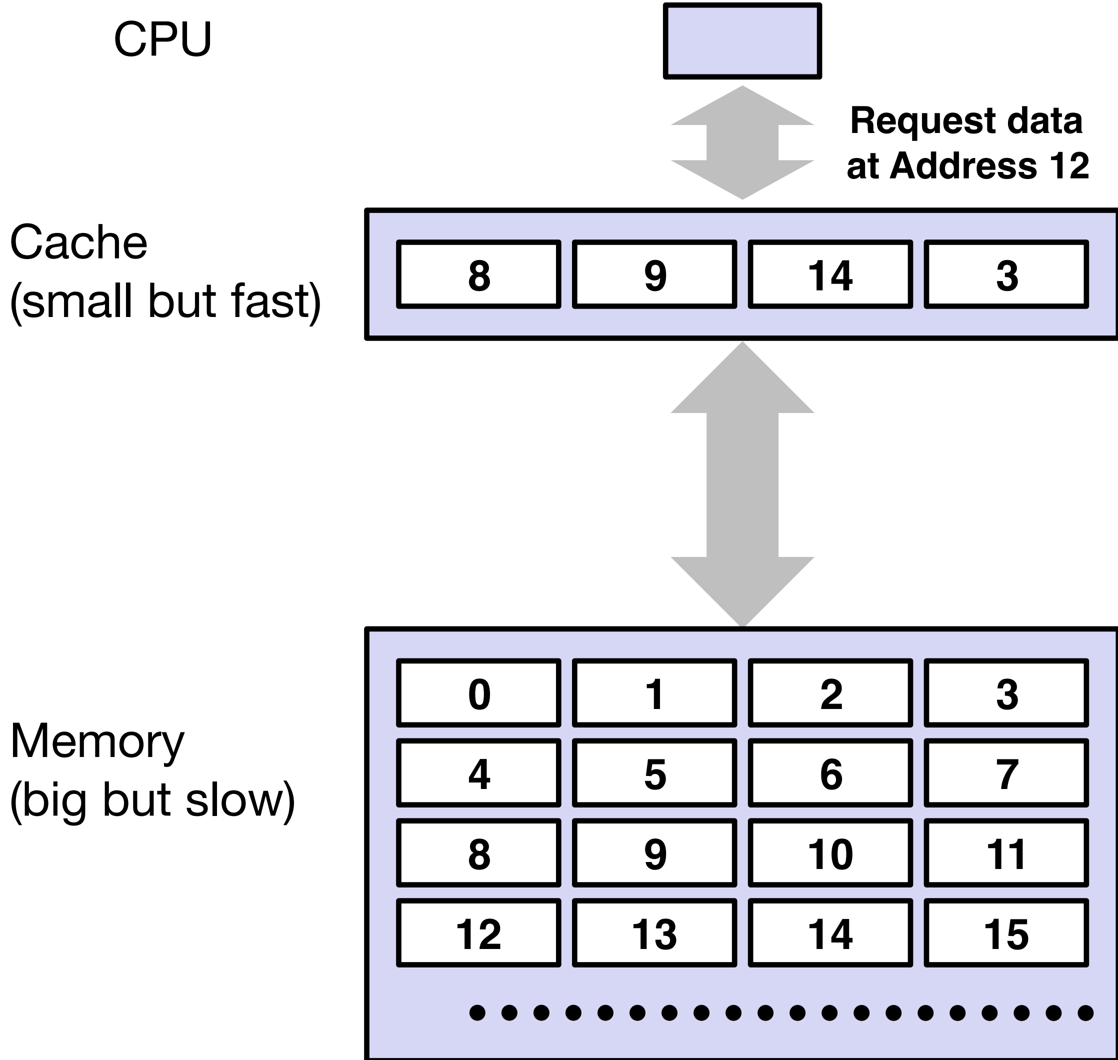
Memory
(big but slow)



Cache Illustrations



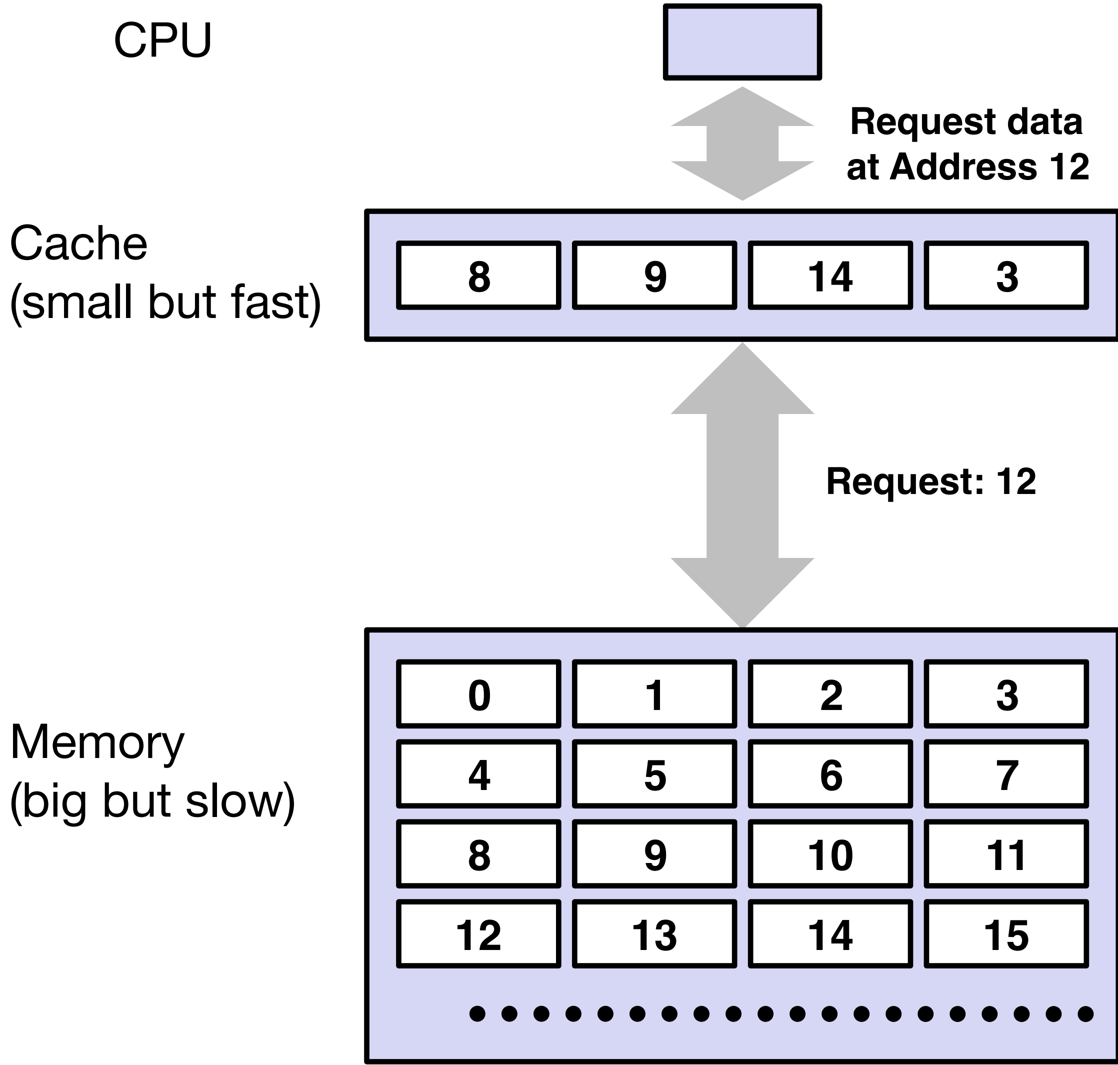
Cache Illustrations



Data in address b is needed

Address b is not in cache: Miss!

Cache Illustrations

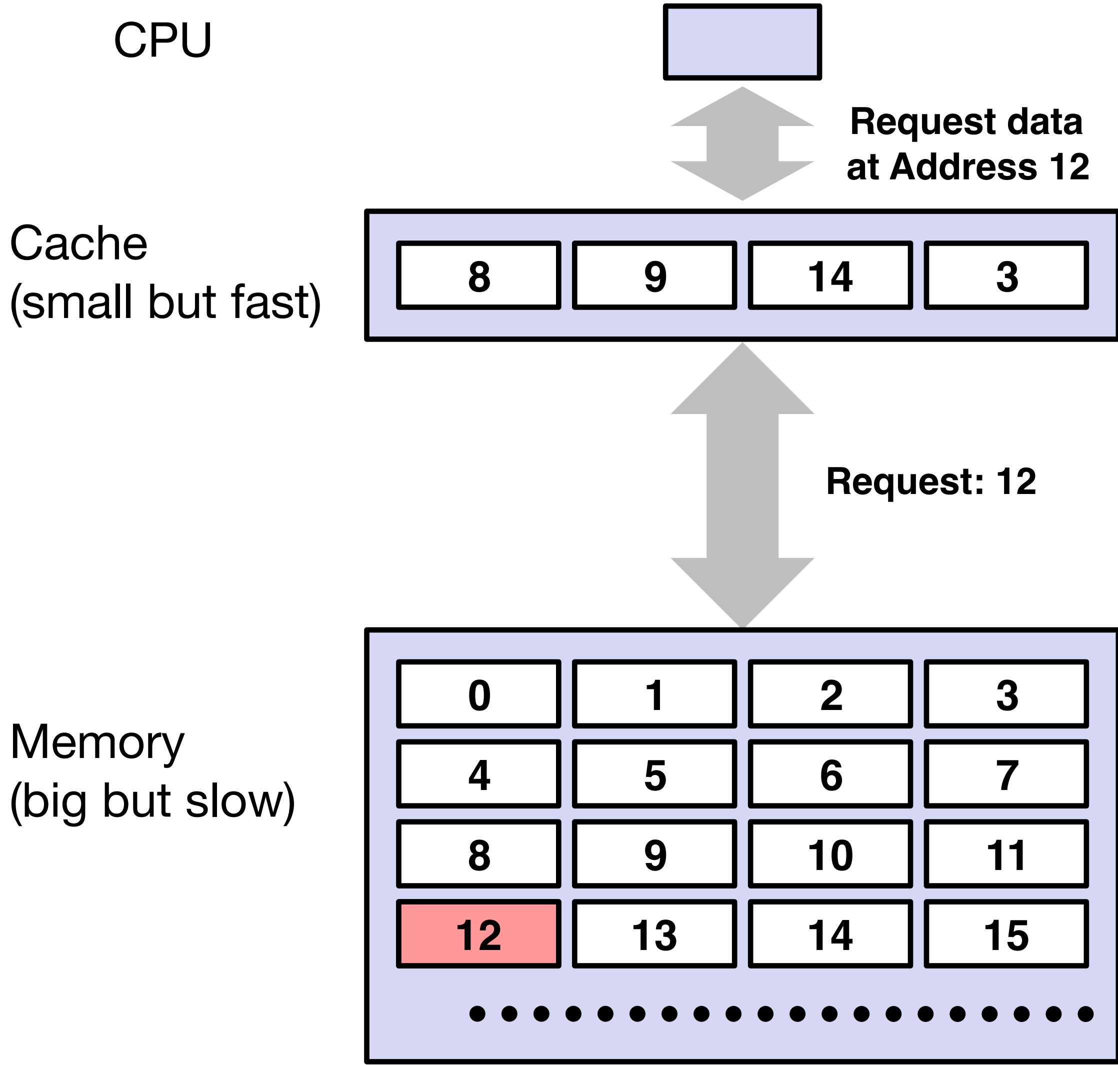


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations

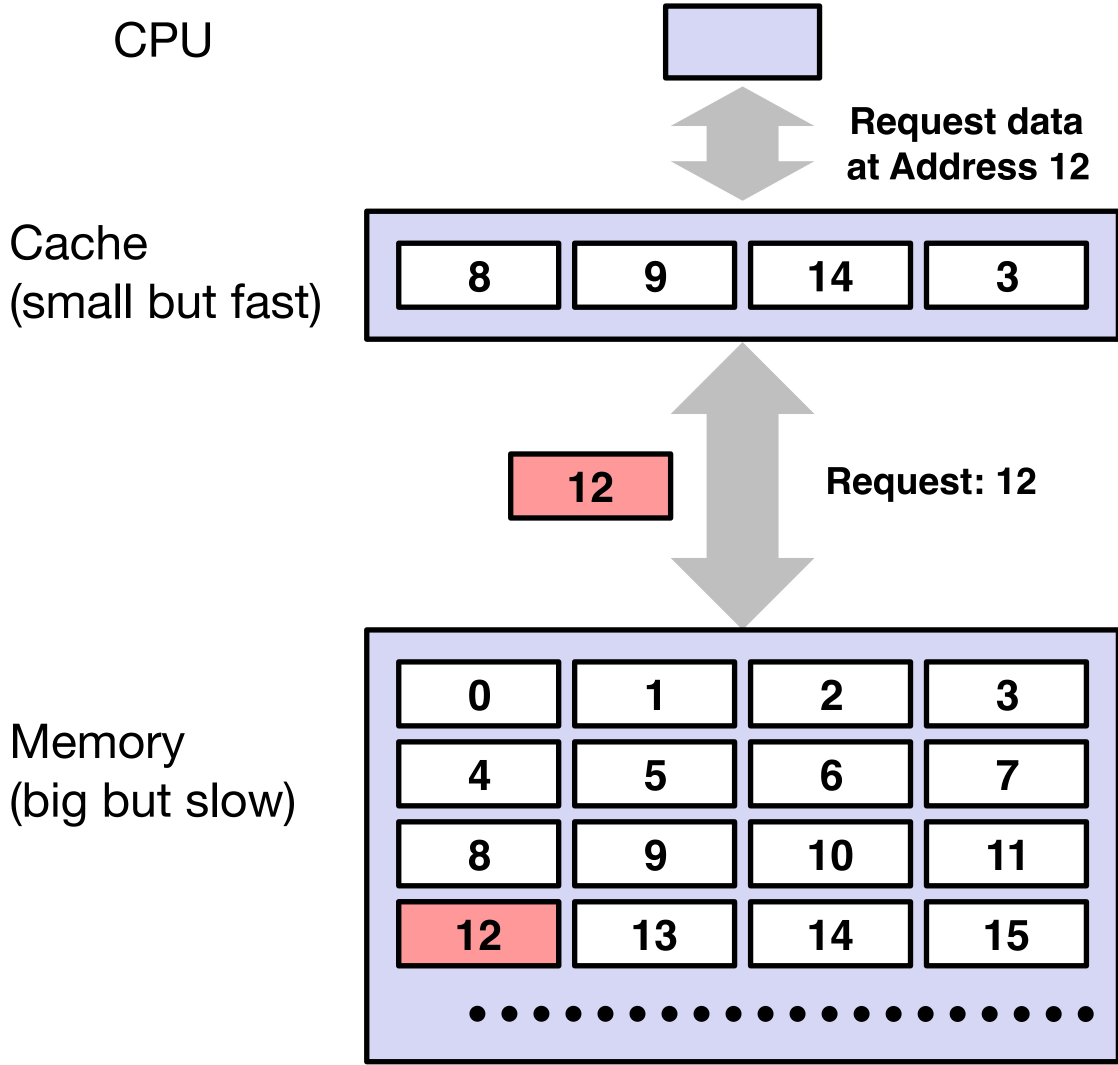


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations

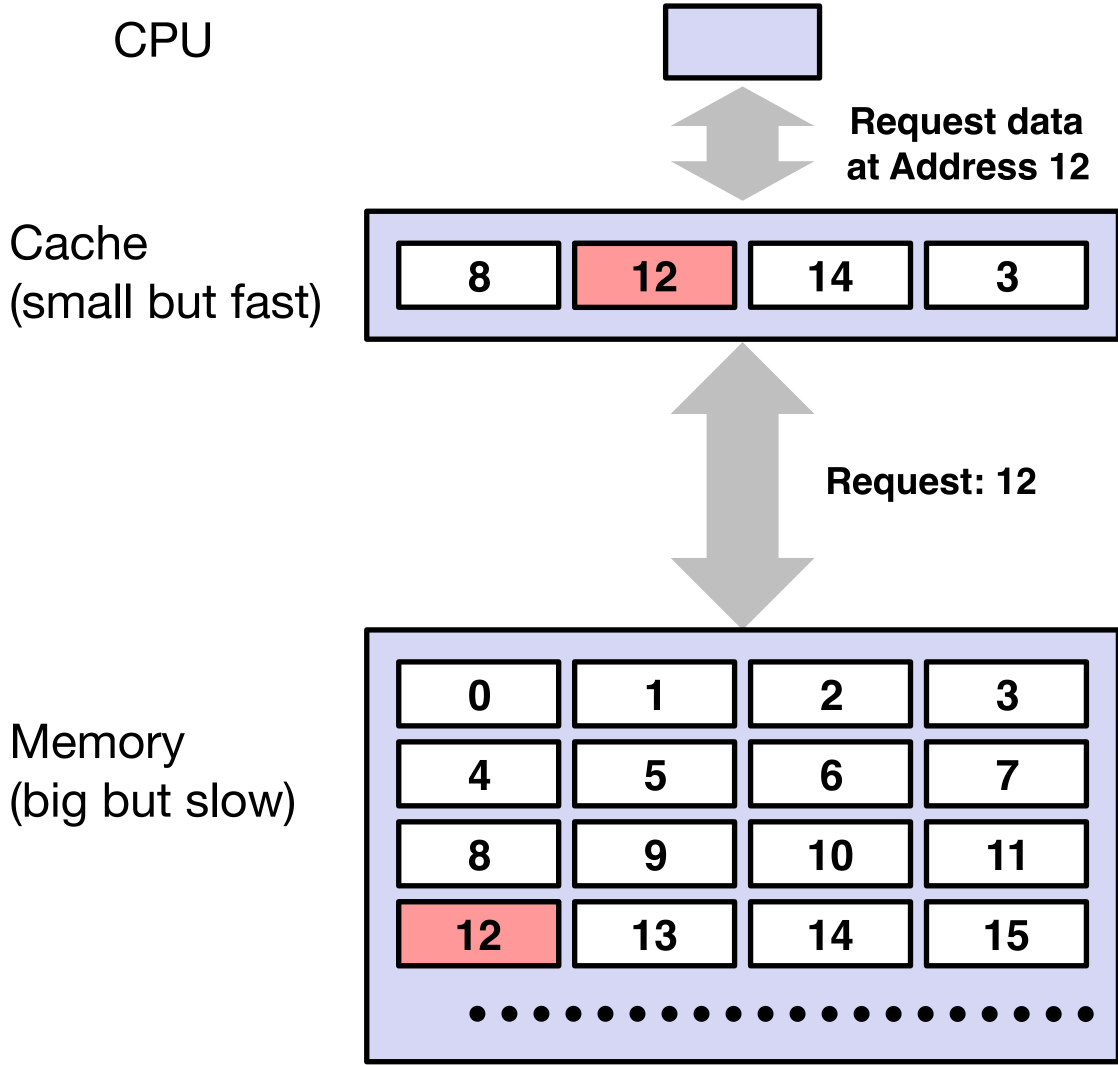


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations



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

Fully Associative Cache

Content Valid? Tag

- Every memory location can be mapped to any cache line in the cache.

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

Fully Associative Cache

Content Valid? Tag

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

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

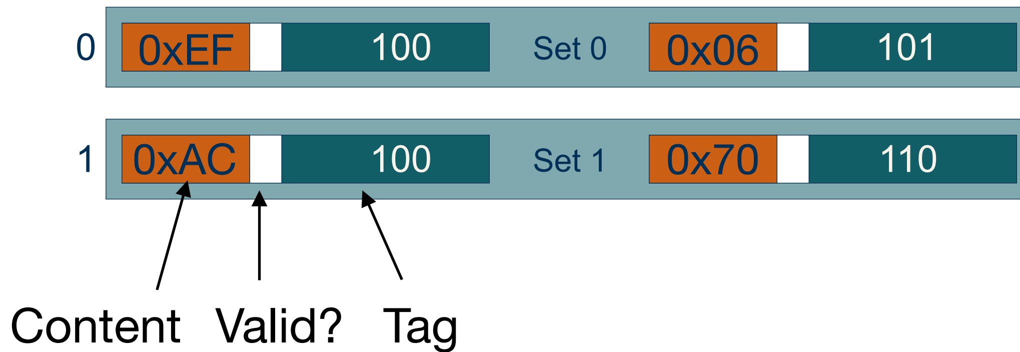
Fully Associative Cache

Content Valid? Tag

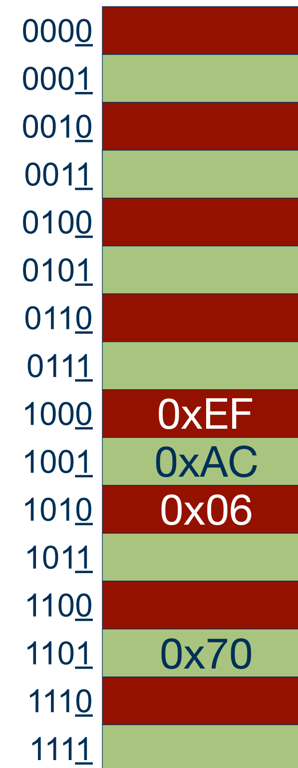
Address	Content
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	0xEF
1001	0xAC
1010	0x06
1011	
1100	
1101	0x70
1110	
1111	

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

2-Way Associative Cache

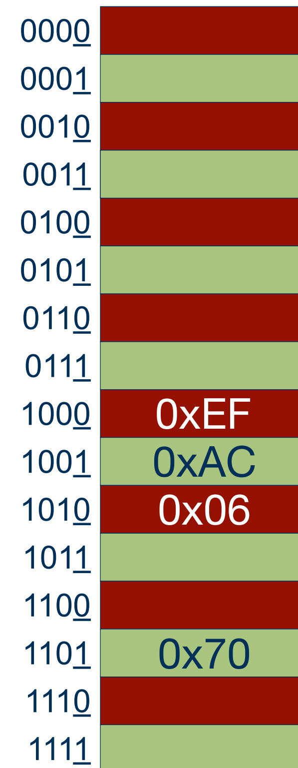


- 4 cache lines are organized into two sets; each set has 2 cache lines (i.e., 2 ways)



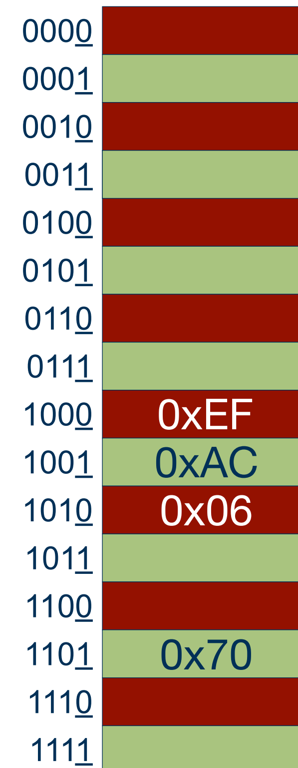
2-Way Associative Cache

- 4 cache lines are organized into two sets; each set has 2 cache lines (i.e., 2 ways)
- Even address go to first set and odd addresses go to the second set

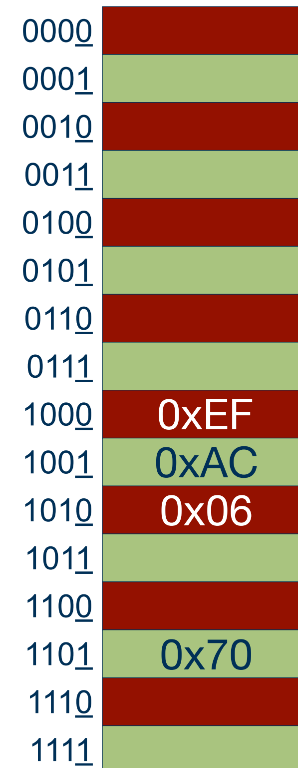


2-Way Associative Cache

- 4 cache lines are organized into two sets; each set has 2 cache lines (i.e., 2 ways)
- 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
  - Using the LSB to find the set (i.e., odd vs. even)
  - Tag now stores the higher 3 bits instead of the entire address



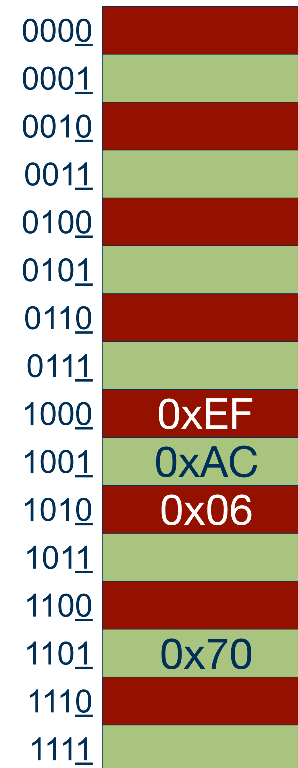
2-Way Associative Cache



- Given a request to address, say 101**1**, from the CPU, detecting cache hit/miss requires:

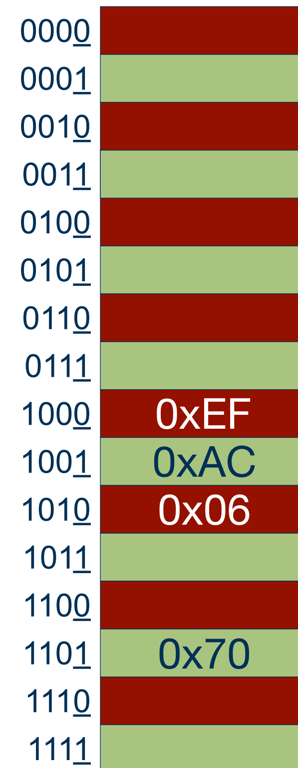
2-Way Associative Cache

- Given a request to address, say 101**1**, 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

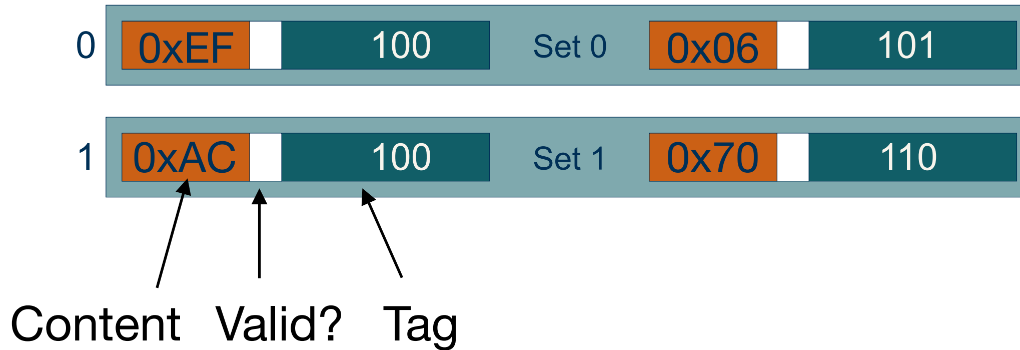


2-Way Associative Cache

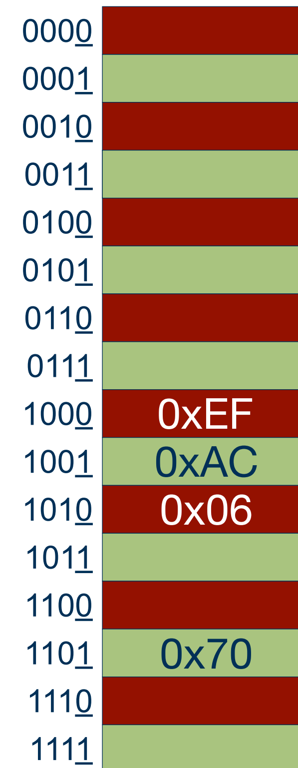
- Given a request to address, say 101**1**, from the CPU, detecting cache hit/miss requires:
  - Using the LSB to index into the cache and find the corresponding set, in this case set 1
  - Then do an associative search in that set, i.e., compare the highest 3 bits 101 with both tags in set 1



2-Way Associative Cache



- Given a request to address, say 101**1**, 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

Set	Content	Valid?	Tag
00	0xEF		10
01	0xAC		10
10	0x06		10
11			

Content Valid? Tag

- 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

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

Direct-Mapped (1-way Associative) Cache

	Content	Valid?	Tag
00	0xEF		10
01	0xAC		10
10	0x06		10
11			

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

- Given a request to address, say 11**01**, from the CPU, detecting cache hit/miss requires:

Direct-Mapped (1-way Associative) Cache

Set	Content	Valid?	Tag
00	0xEF		10
01	0xAC		10
10	0x06		10
11			

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

- Given a request to address, say 11**01**, from the CPU, detecting cache hit/miss requires:
  - Using the 2 LSBs to index into the cache and find the set, in this case set 01

Direct-Mapped (1-way Associative) Cache

	Content	Valid?	Tag
00	0xEF		10
01	0xAC		10
10	0x06		10
11			

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

- Given a request to address, say 11**01**, from the CPU, detecting cache hit/miss requires:
  - Using the 2 LSBs to index into the cache and find the set, in this case set 01
  - Then do an associative search in that set, i.e., compare the highest 2 bits 11 in the address with the tag in set 01 —> miss

Direct-Mapped (1-way Associative) Cache

	Content	Valid?	Tag
00	0xEF		10
01	0xAC		10
10	0x06		10
11			

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

- Given a request to address, say 11**01**, from the CPU, detecting cache hit/miss requires:
  - Using the 2 LSBs to index into the cache and find the set, in this case set 01
  - Then do an associative search in that set, i.e., compare the highest 2 bits 11 in the address with the tag in set 01 —> miss
  - Only one comparison required

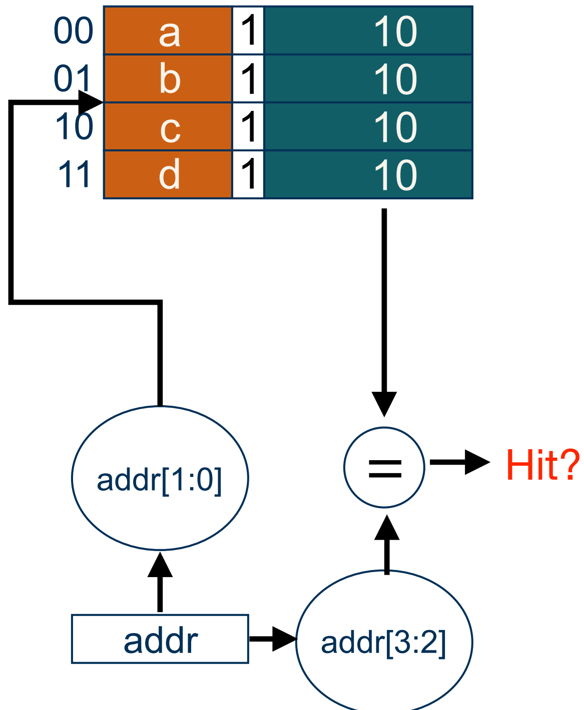
Associative verses Direct Mapped Trade-offs

Associative verses Direct Mapped Trade-offs

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

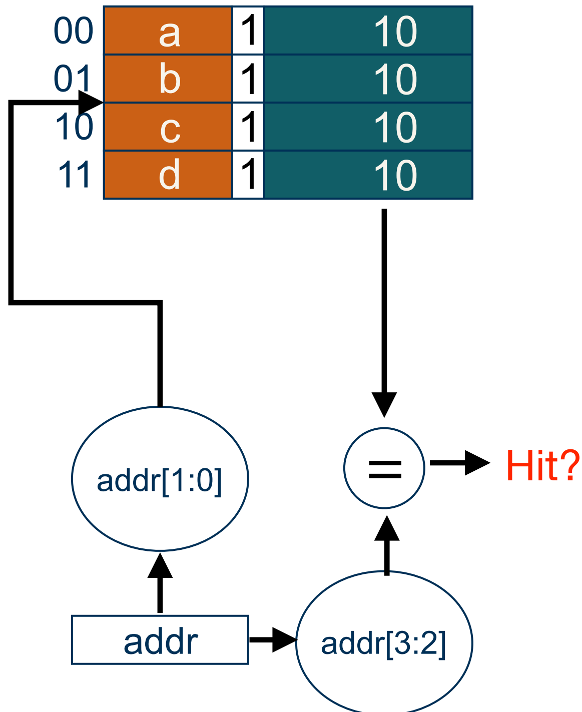
Associative verses Direct Mapped Trade-offs

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



Associative verses Direct Mapped Trade-offs

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

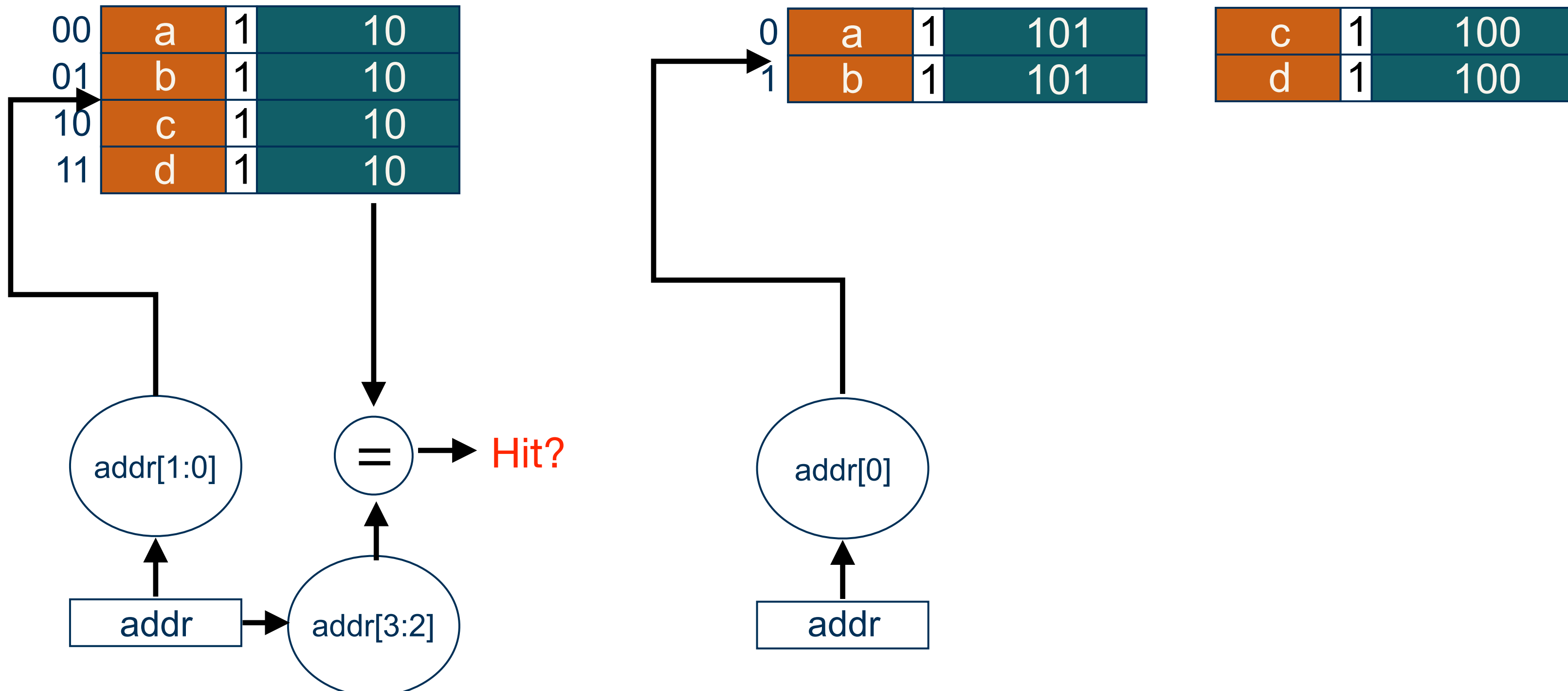


0	a	1	101
1	b	1	101

c	1	100
d	1	100

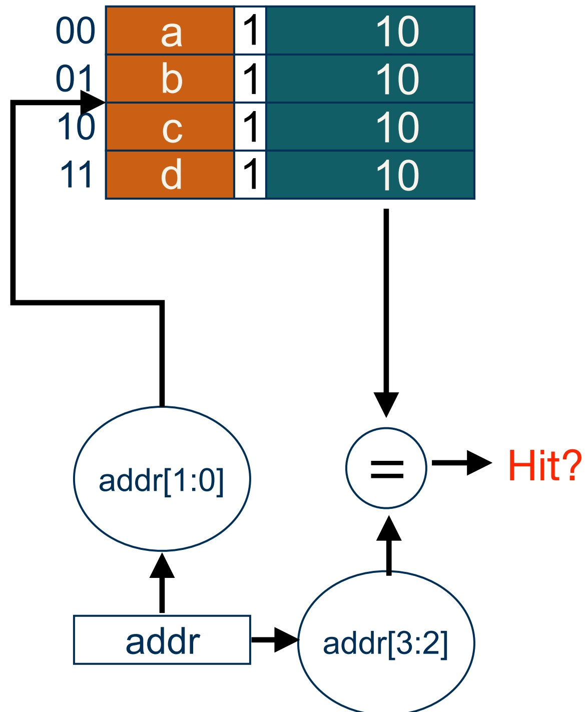
Associative verses Direct Mapped Trade-offs

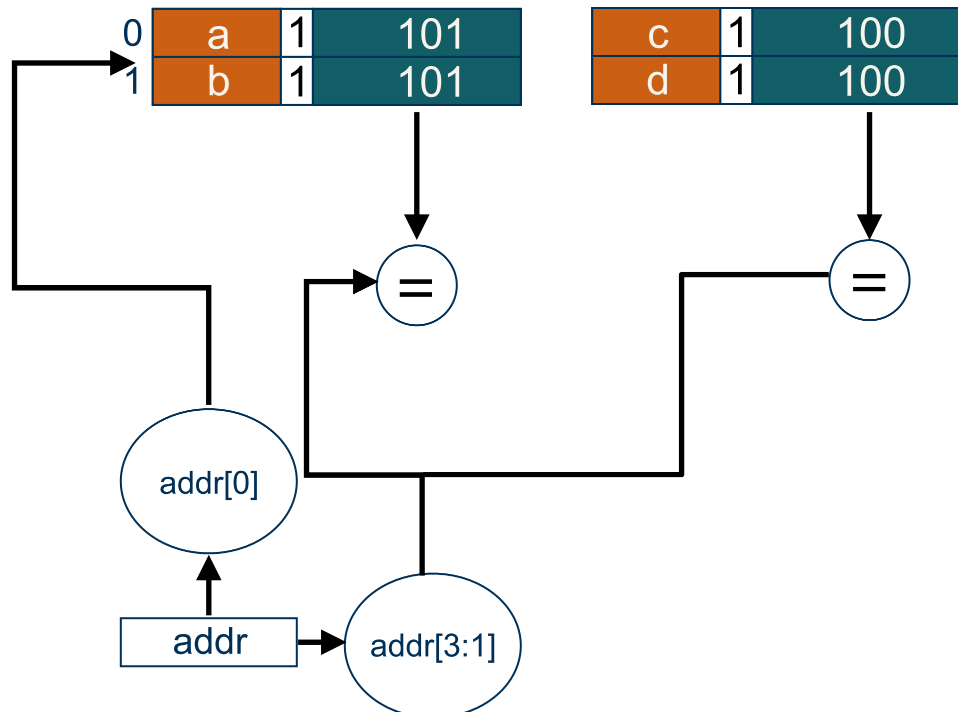
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?



Associative verses Direct Mapped Trade-offs

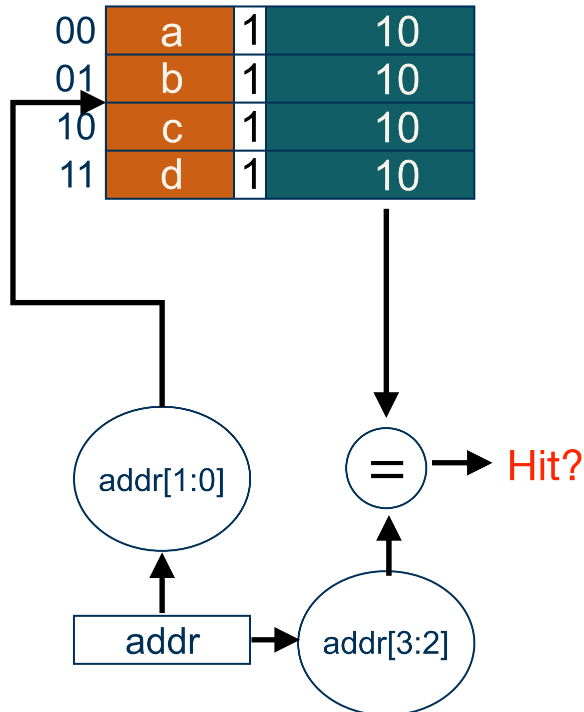
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower and higher power consumption. Why?

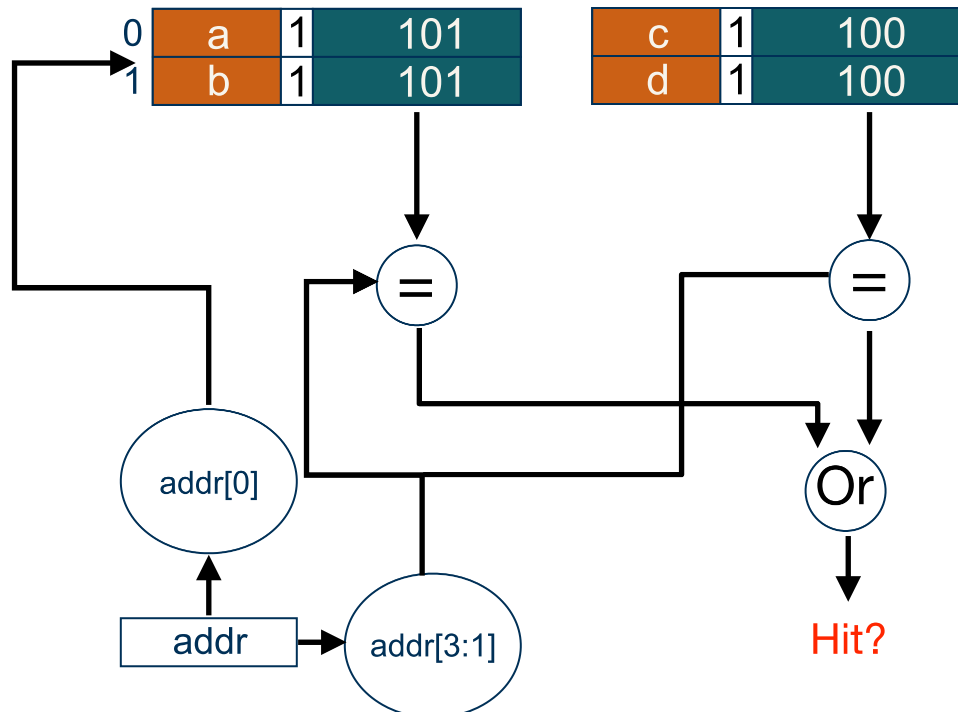




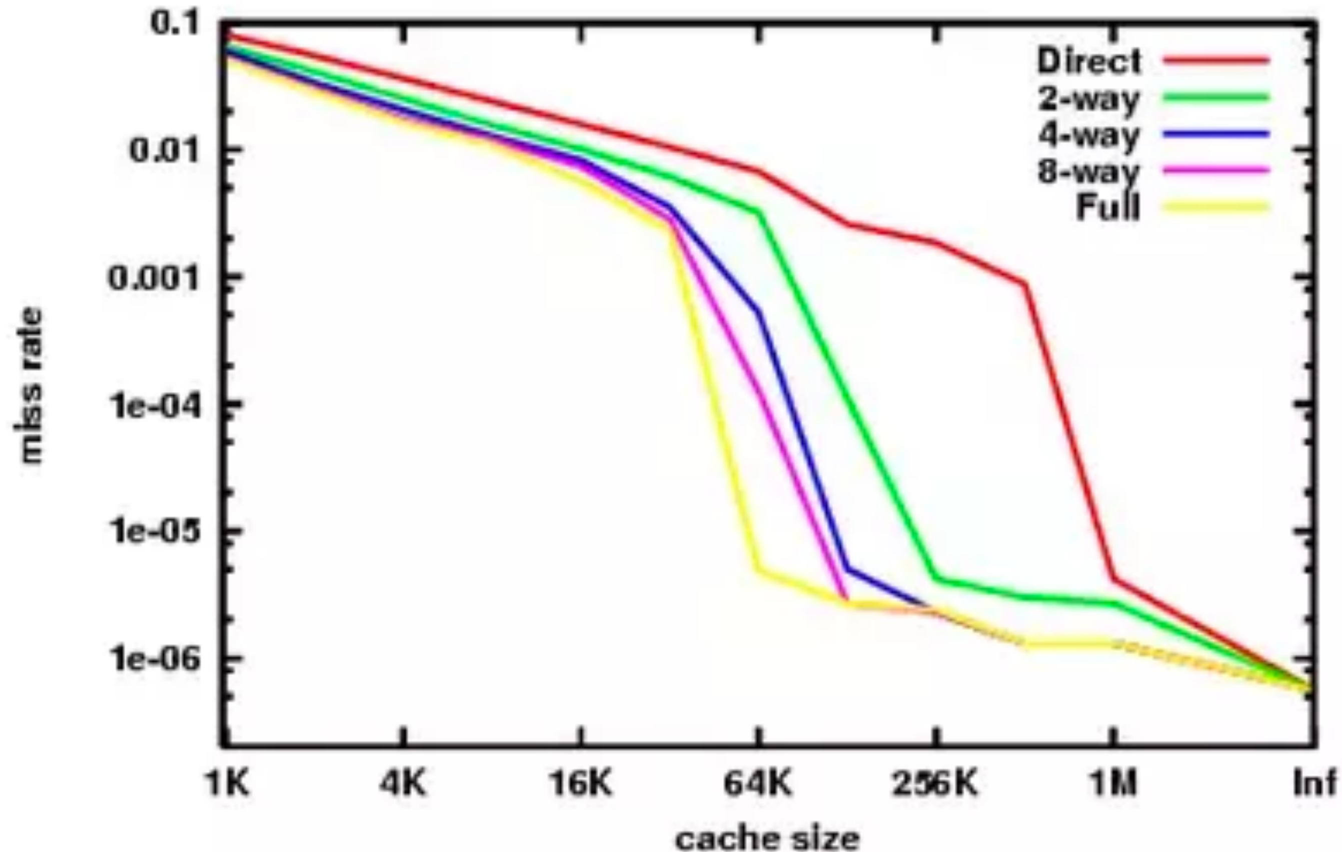
Associative verses Direct Mapped Trade-offs

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





Associative verses 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

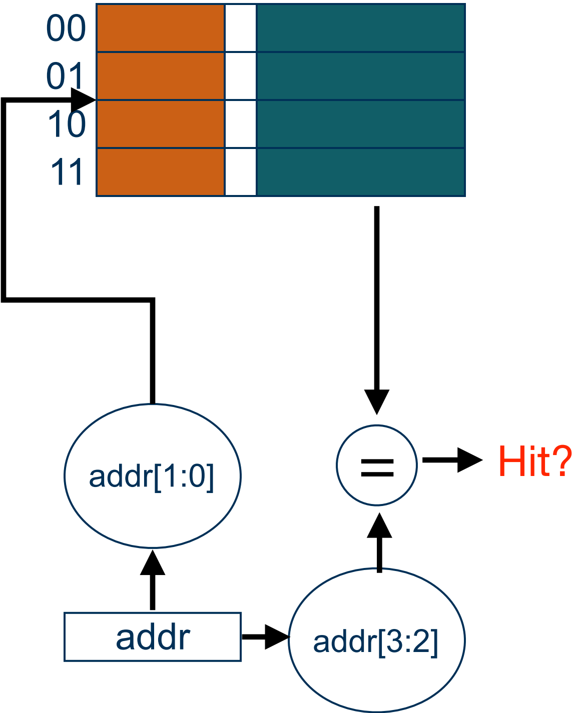
Locality again

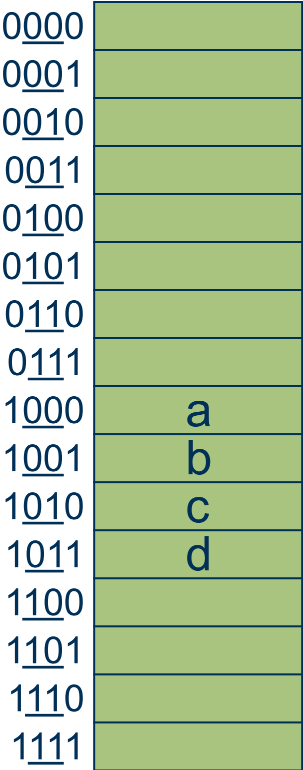
- So far: temporal locality
- What about spatial?
- Idea: Each cache location (cache line) store multiple bytes

Cache-Line Size of 2

Cache

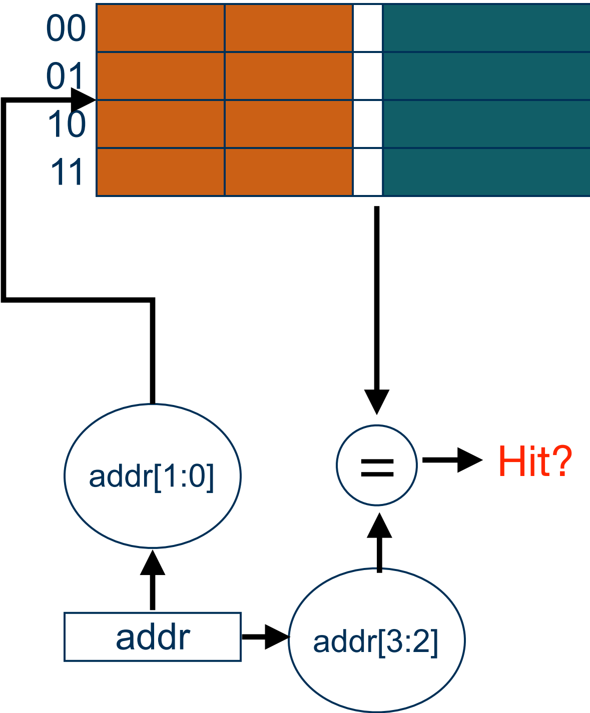
Memory



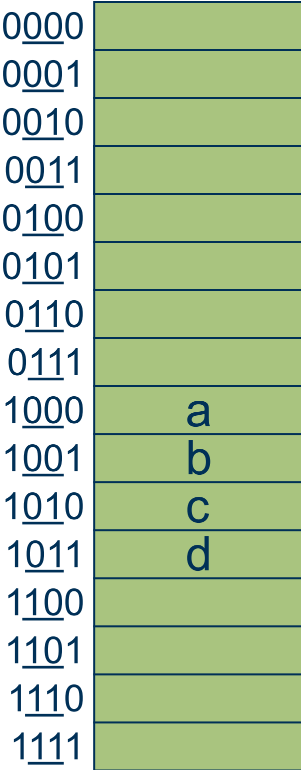


Cache-Line Size of 2

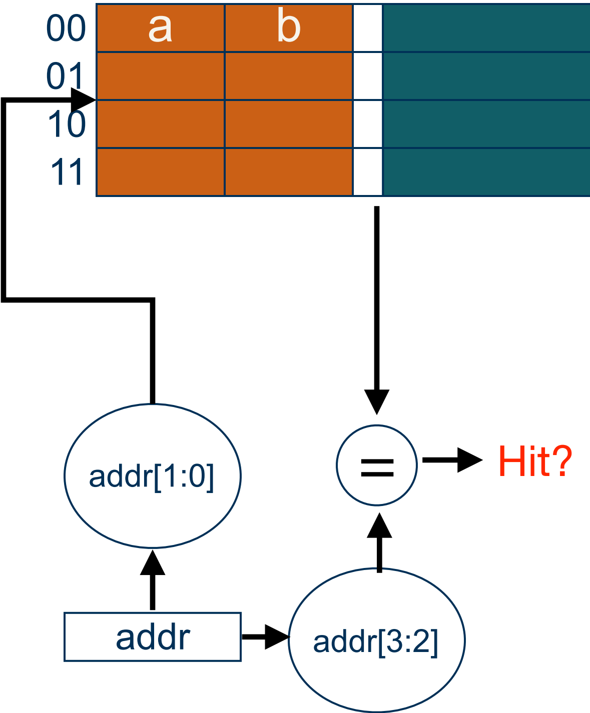
Cache

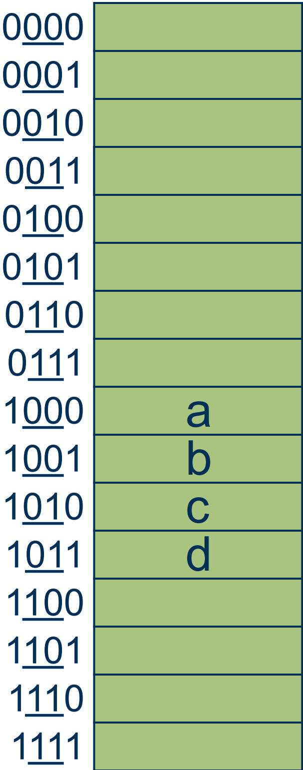


Memory



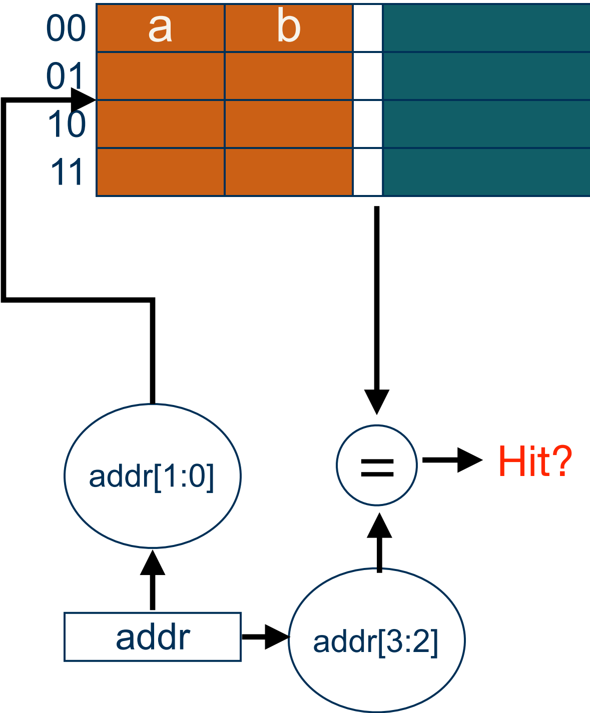
Cache-Line Size of 2

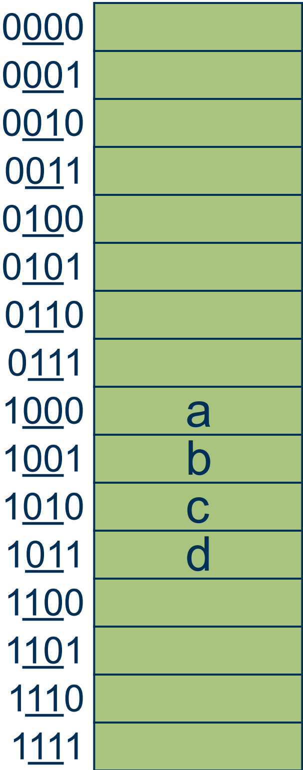




- Read 1000

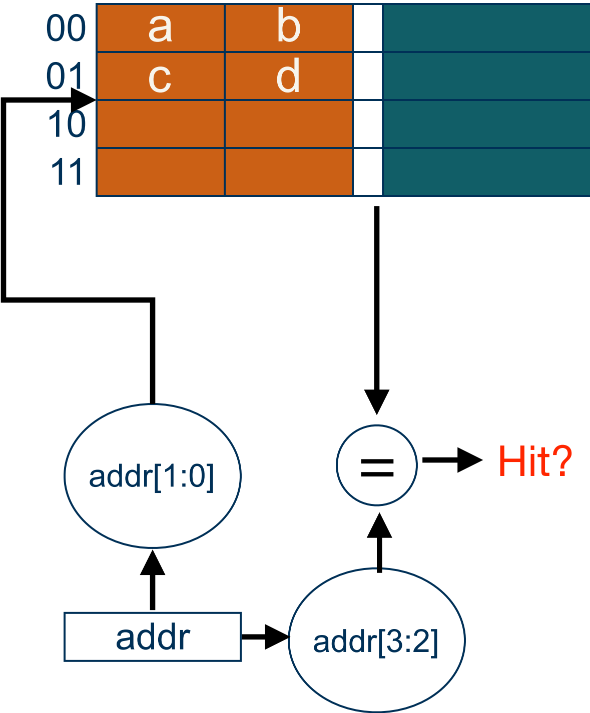
Cache-Line Size of 2

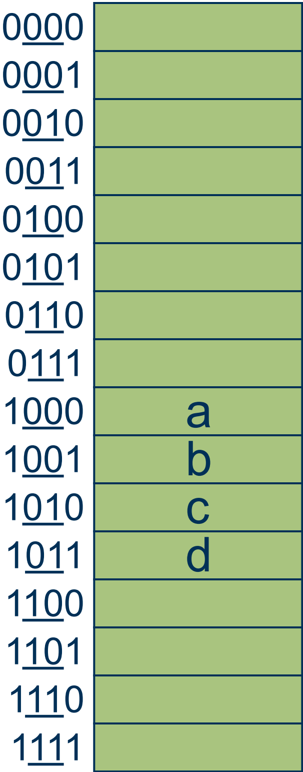




- Read 1000
- Read 1001 (Hit!)

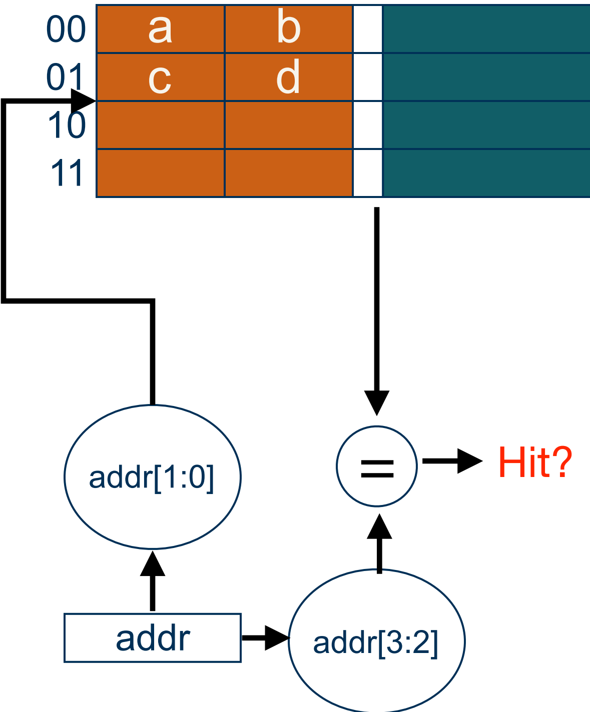
Cache-Line Size of 2

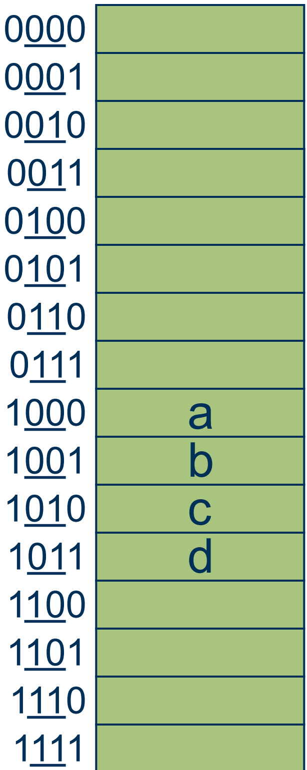




- Read 1000
- Read 1001 (Hit!)
- Read 1010

Cache-Line Size of 2





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

Cache Access Summary

- Assuming *b* bits in a memory address
- The *b* bits are split into three fields:
  - Lower *l* bits are used for byte offset within a cache line. Cache line size $L = 2^l$
  - Next *s* bits used as index to find a set. Total sets $S = 2^s$
  - The higher (*b* - *l* - *s*) bits are used for the tag
- Associativity *n* is independent of the the split between index and tag

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?
- 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 (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
  - - Requires transfer of the whole cache line (although only one byte might have been modified)

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

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!

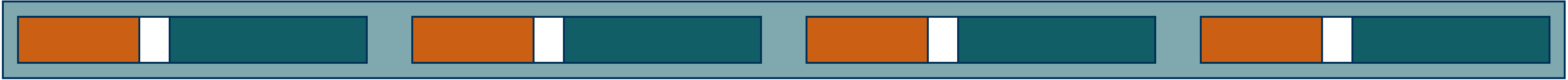
Eviction/Replacement Policy

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

Eviction/Replacement Policy

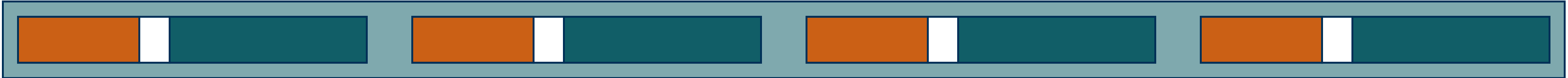
- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first

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

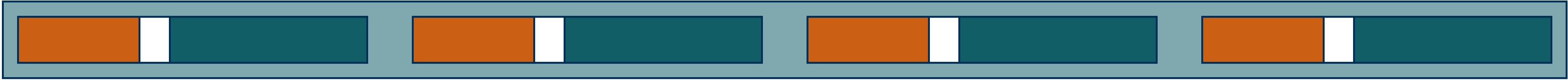


- Which cache line should be replaced?
  - Direct mapped? Only one place!
  - Associative caches? Multiple places!
- For associative cache:
  - Any invalid cache line first
  - If all are valid, consult the replacement policy
  - Randomly pick one???

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?

Address stream:

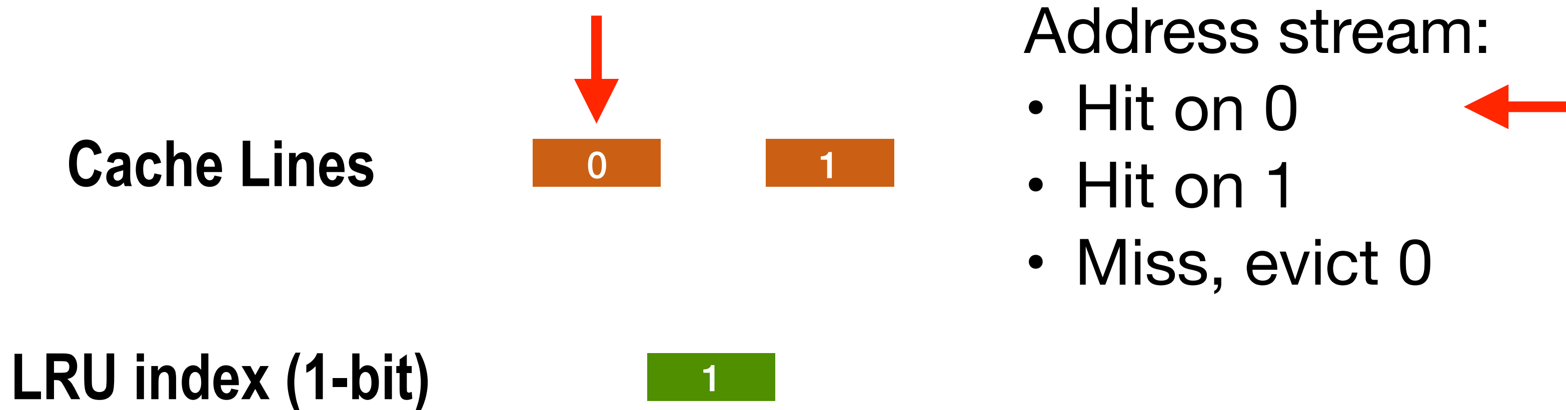
- Hit on 0
- Hit on 1
- Miss, evict 0

Cache Lines

LRU index (1-bit)

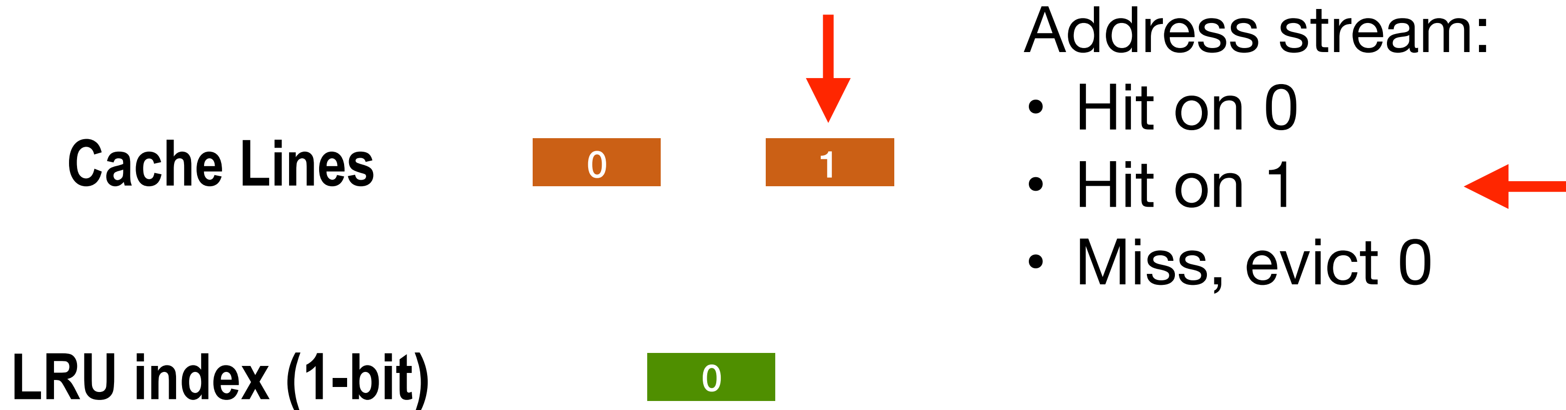
Implementing LRU

- 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?



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?

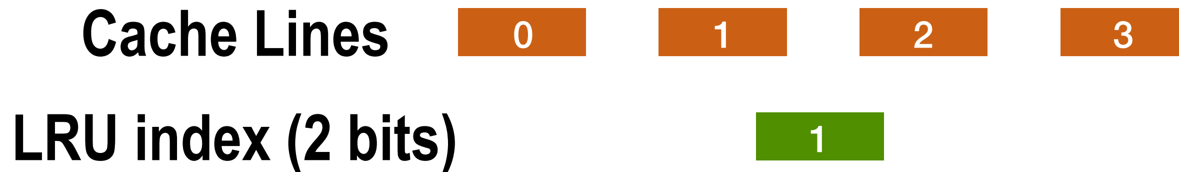
Cache Lines

LRU index (1-bit)

Address stream:

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

Implementing LRU

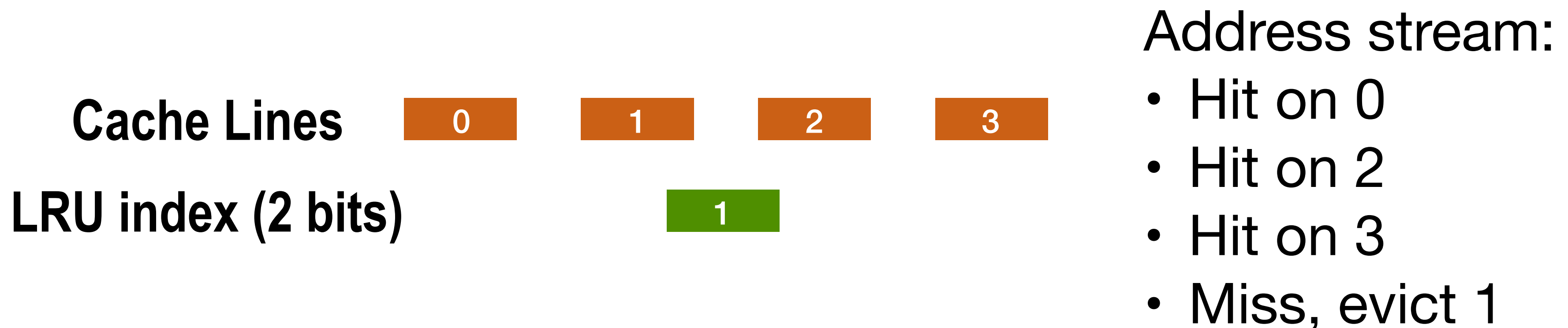


Address stream:

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

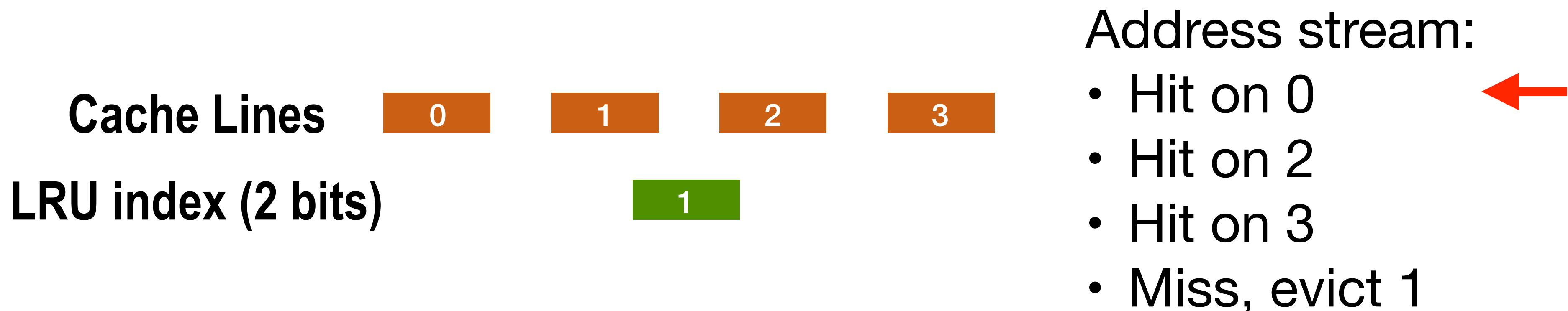
Implementing LRU

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



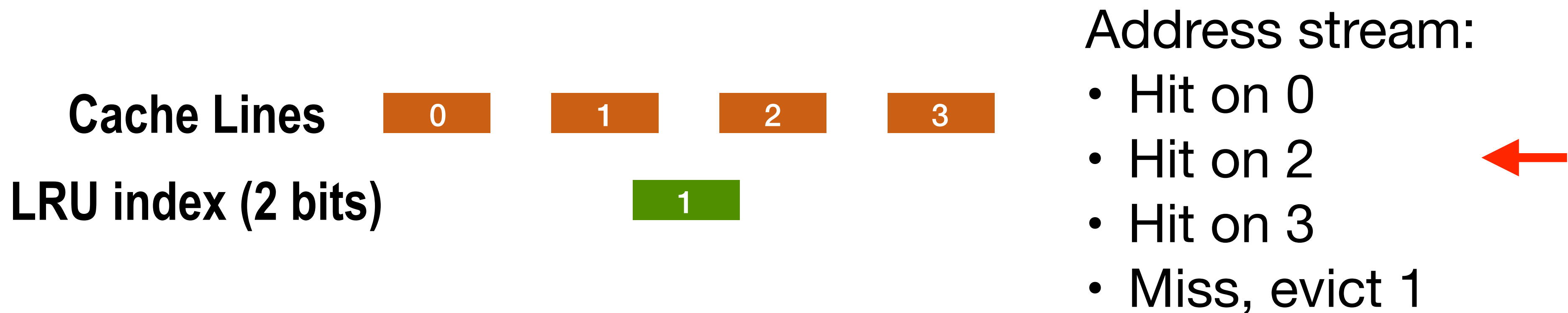
Implementing LRU

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



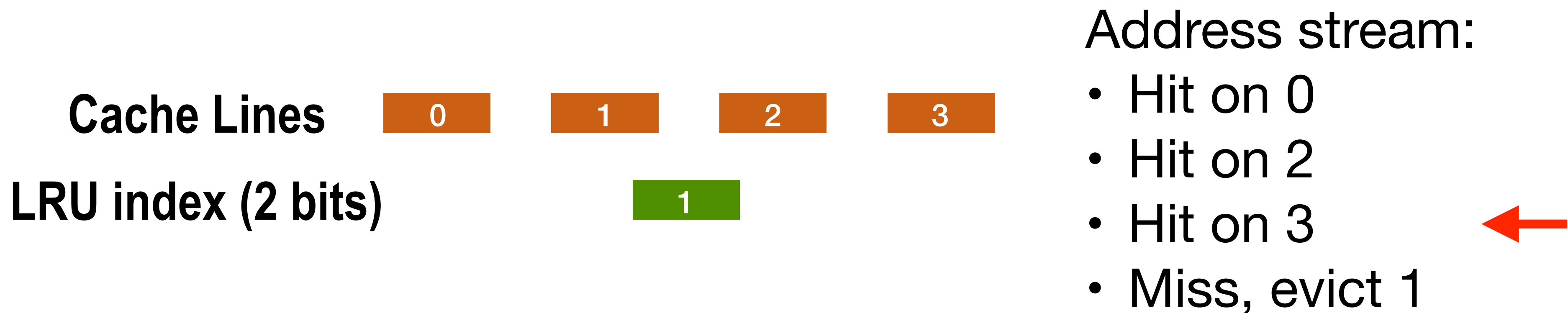
Implementing LRU

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



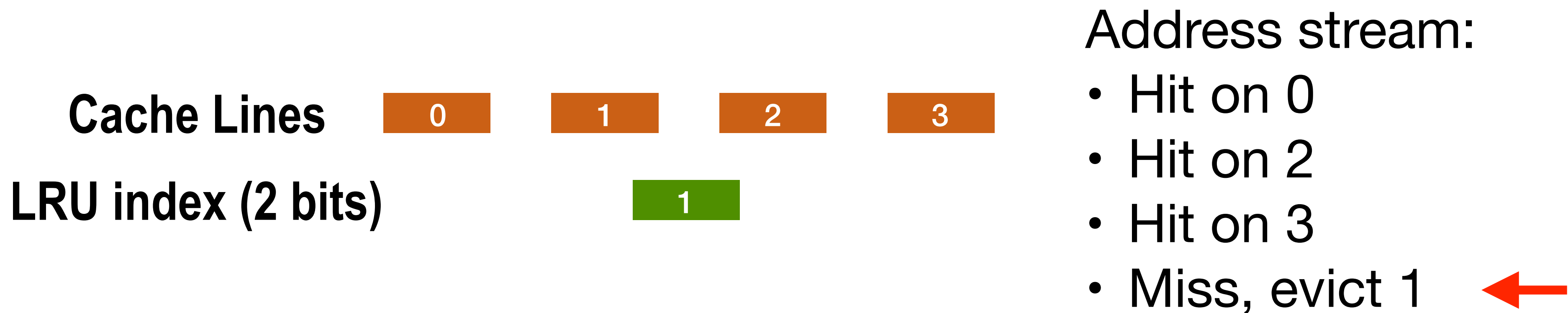
Implementing LRU

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



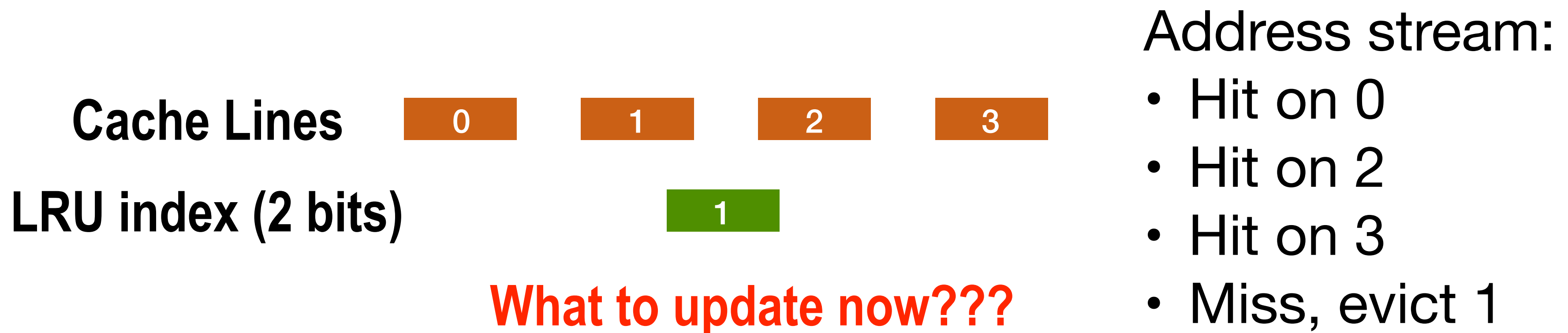
Implementing LRU

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



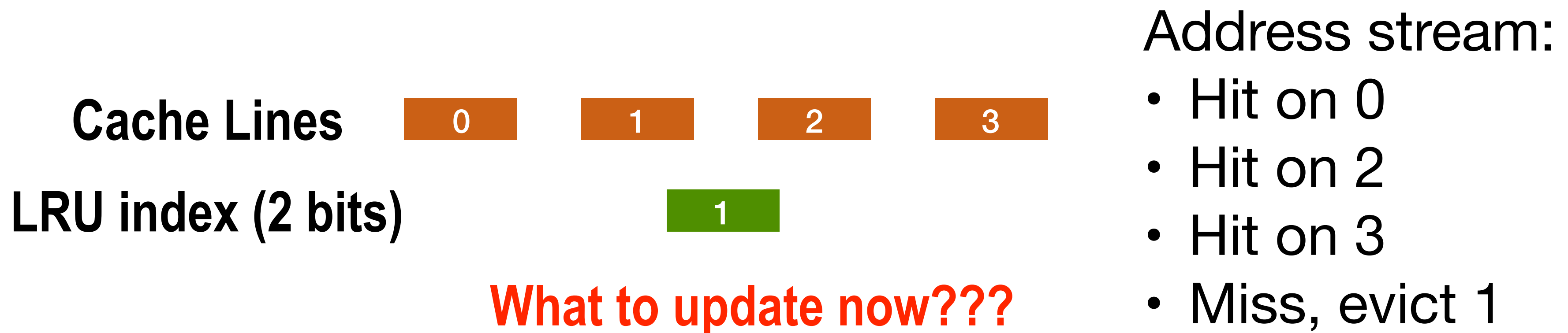
Implementing LRU

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



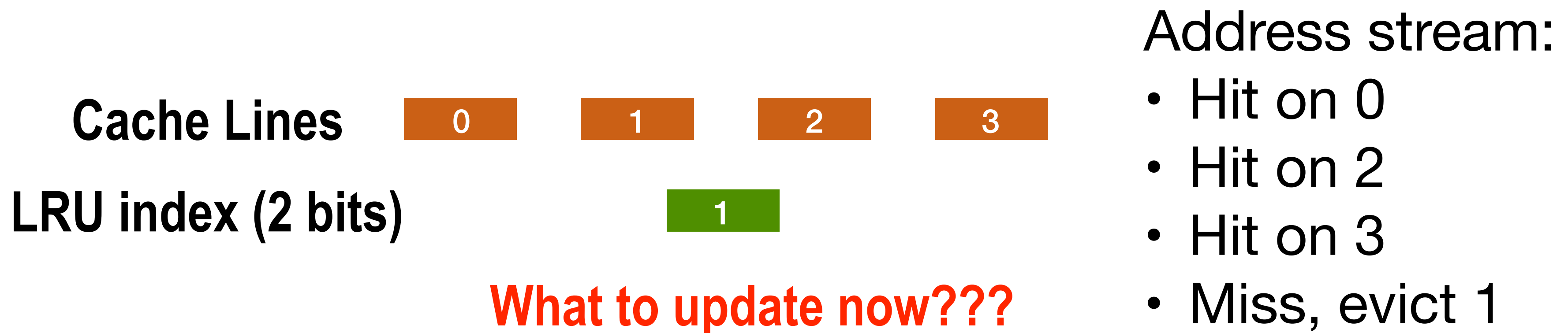
Implementing LRU

- 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
  - How many possible orderings are there?



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
  - How many possible orderings are there?
  - What are the hardware structures needed?

What to update now???

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?
  - Essentially have to track the ordering of all cache lines
  - How many possible orderings are there?
  - What are the hardware structures needed?
  - In reality, true LRU is never implemented. Too complex.

What to update now???

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?
  - Essentially have to track the ordering of all cache lines
  - How many possible orderings are there?
  - What are the hardware structures needed?
  - In reality, true LRU is never implemented. Too complex.
  - Google Pseudo-LRU

Cache Lines 0 1 2 3

LRU index (2 bits) 1

What to update now???

Address stream:

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