

CSC 252 Computer Organization

The Memory Hierarchy Part II

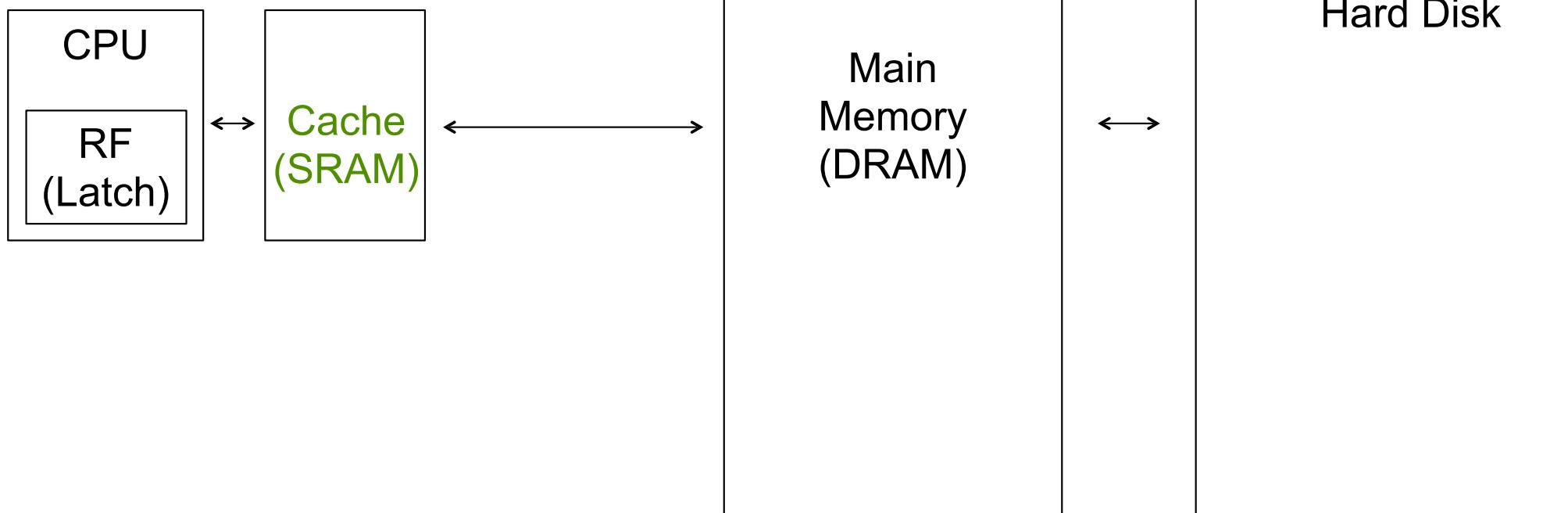
Chen Ding
Professor

Guest lecture
March 26, 2019

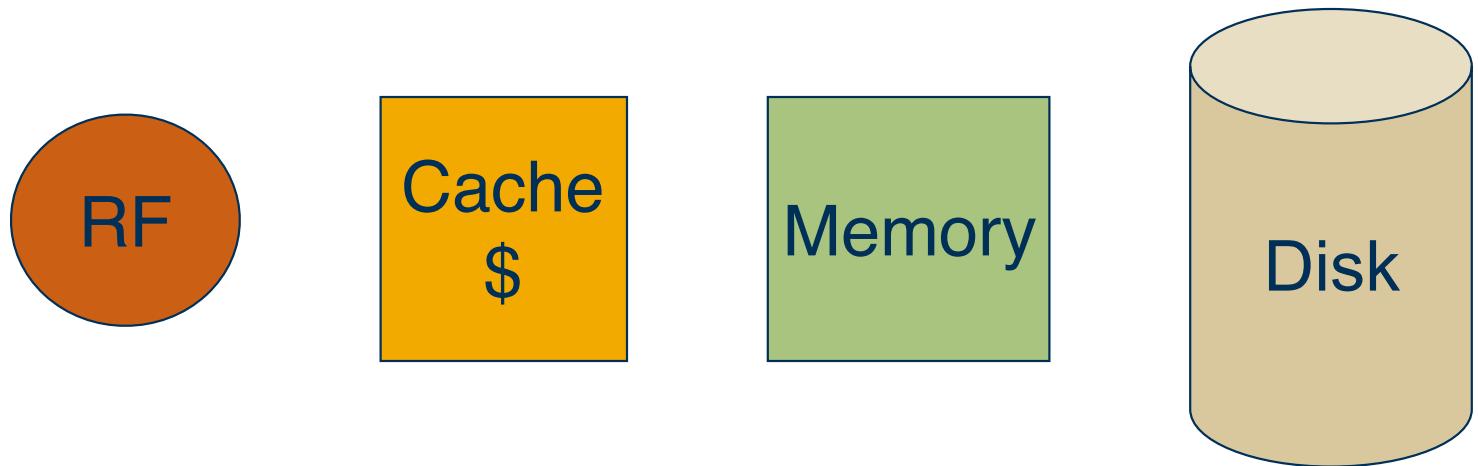
Required reading:
Sections 6.4 and 6.5

Last Lecture: Memory Hierarchy

- Fundamental tradeoff
 - Fast memory: small
 - Large memory: slow



How Things Have Progressed



1995 low-mid range <small>Hennessy & Patterson, Computer Arch., 1996</small>	200B 5ns	64KB 10ns	32MB 100ns	2GB 5ms
2009 low-mid range <small>www.dell.com, \$449 including 17" LCD flat panel</small>	~200B 0.33ns	8MB 0.33ns	4GB <100ns	750GB 4ms
2015 mid range	~200B 0.33ns	8MB 0.33ns	16GB <100ns	256GB 10us

3D XPOINT™ MEMORY MEDIA

Breaks the memory/storage barrier

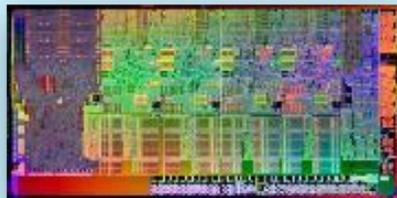
MEMORY

+

STORAGE

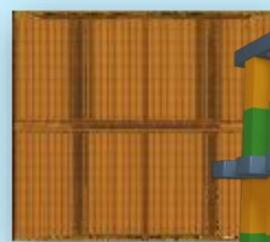
SRAM

Latency: 1X
Size of Data: 1X



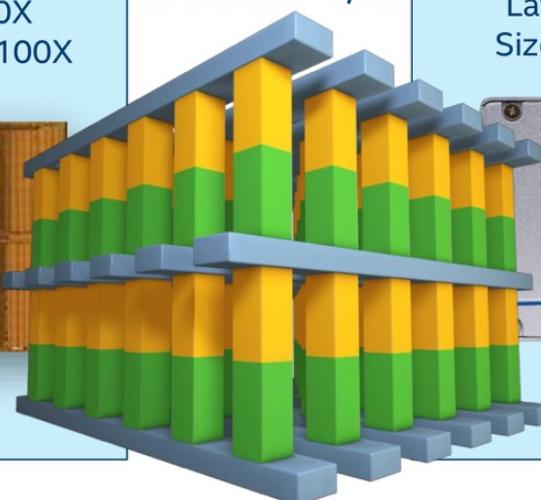
DRAM

Latency: ~10X
Size of Data: ~100X



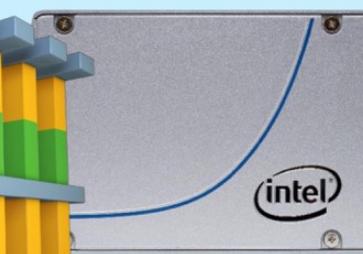
3D XPoint™

Latency: ~100X
Size of Data: ~1,000X



NAND SSD

Latency: ~100,000X
Size of Data: ~1,000X



HDD

Latency: ~10 Million X
Size of Data: ~10,000X



Technology claims are based on comparisons of latency, density and write cycling metrics amongst memory technologies recorded on published specifications of in-market memory products against internal Intel specifications.

NVM Solutions Group

Intel® Optane™ Technology Workshop



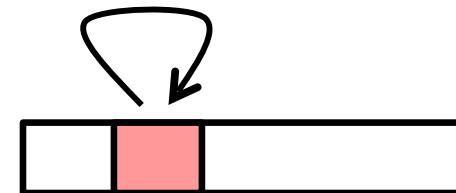
<https://cdn.arstechnica.net/wp-content/uploads/2017/03/IntelR-OptaneTM-Technology-Workshop-Analyst-and-Press-Slides-3-15...-4.jpeg>

Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

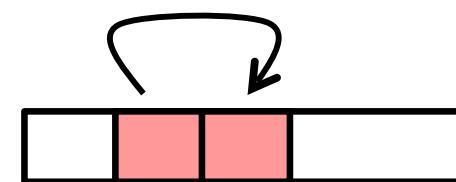
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time

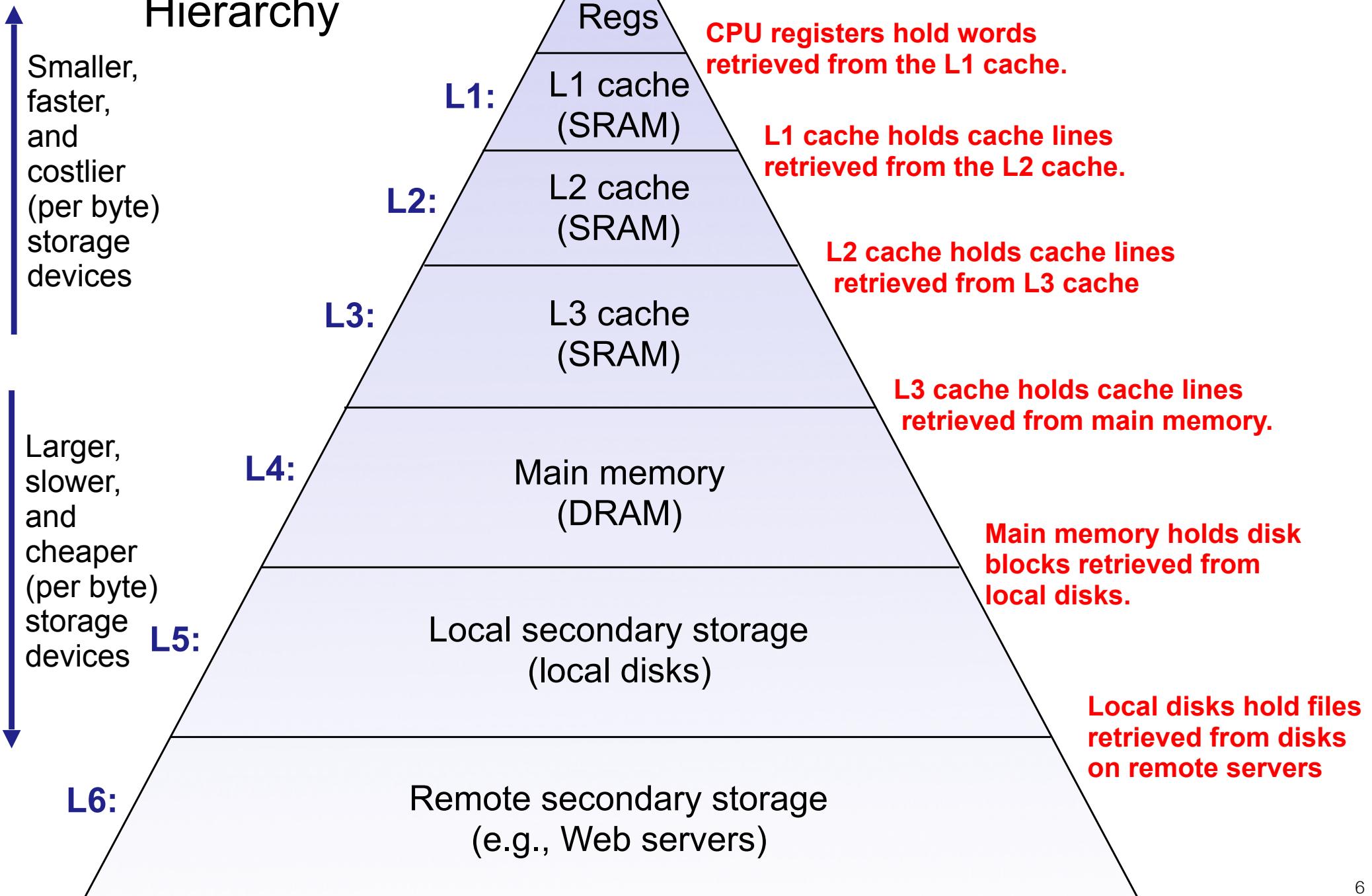


Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

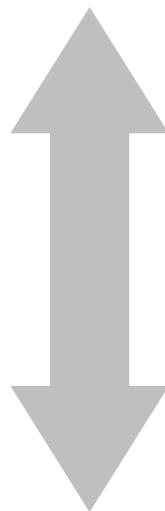
- Data references
 - **Spatial** Locality: Reference array elements in succession (stride-1 reference pattern)
 - **Temporal** Locality: Reference variable sum each iteration.
- Instruction references
 - **Spatial** Locality: Reference instructions in sequence.
 - **Temporal** Locality: Cycle through loop repeatedly.

Example Memory Hierarchy

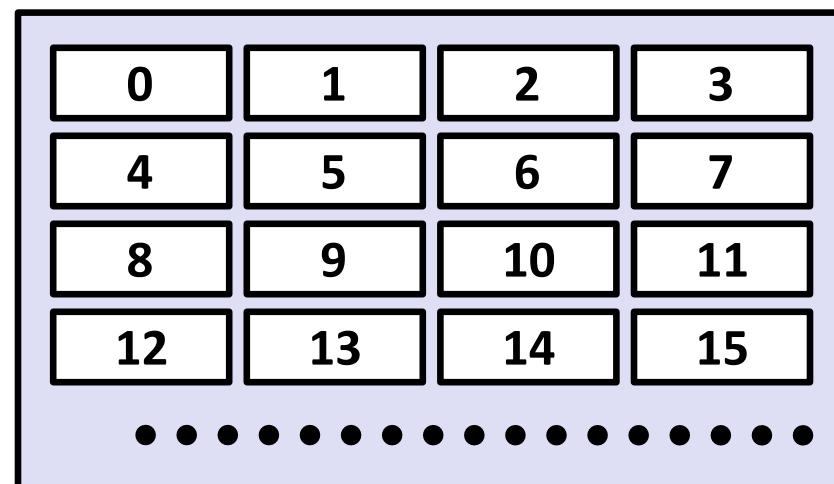


Cache Illustrations

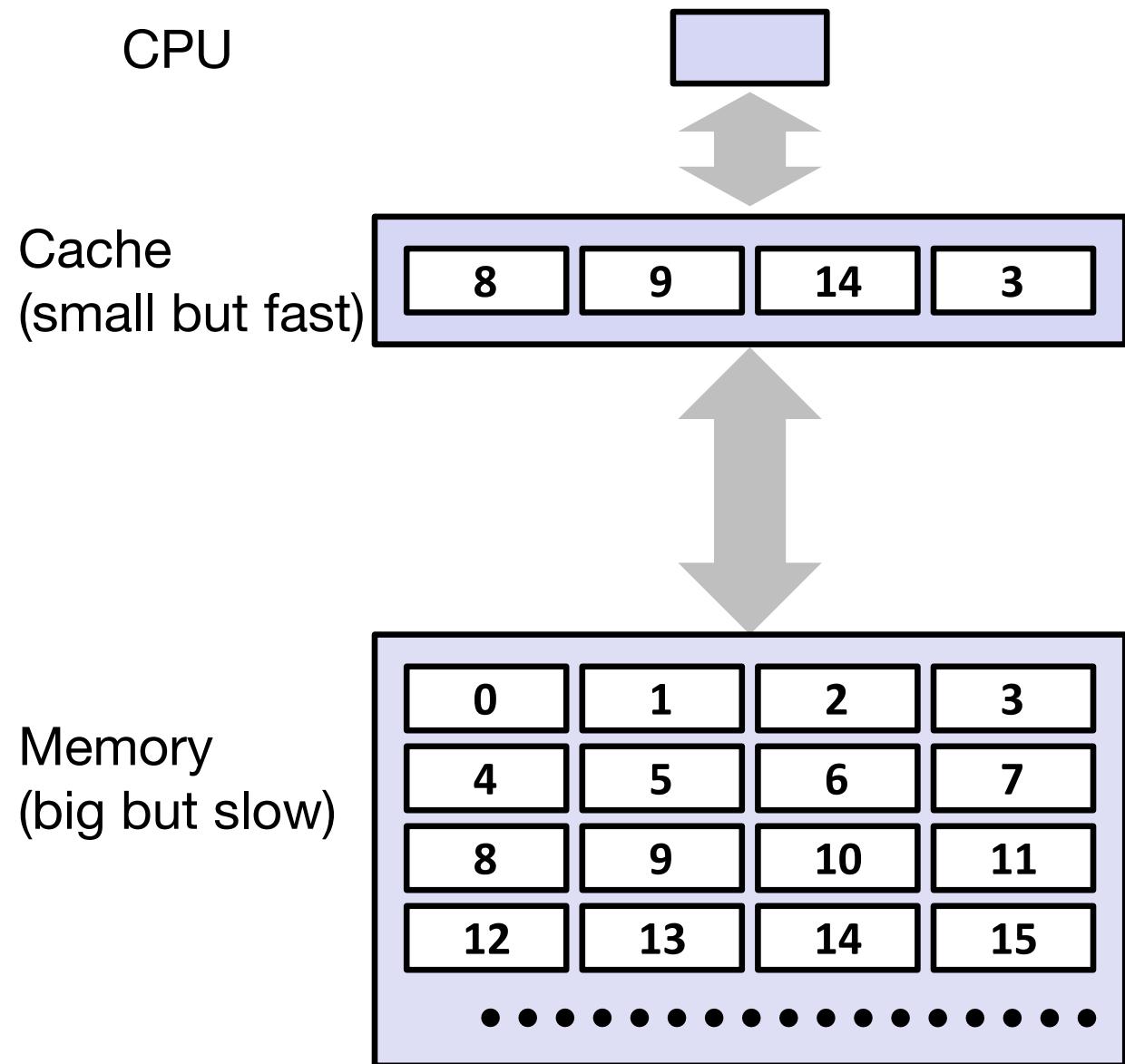
CPU



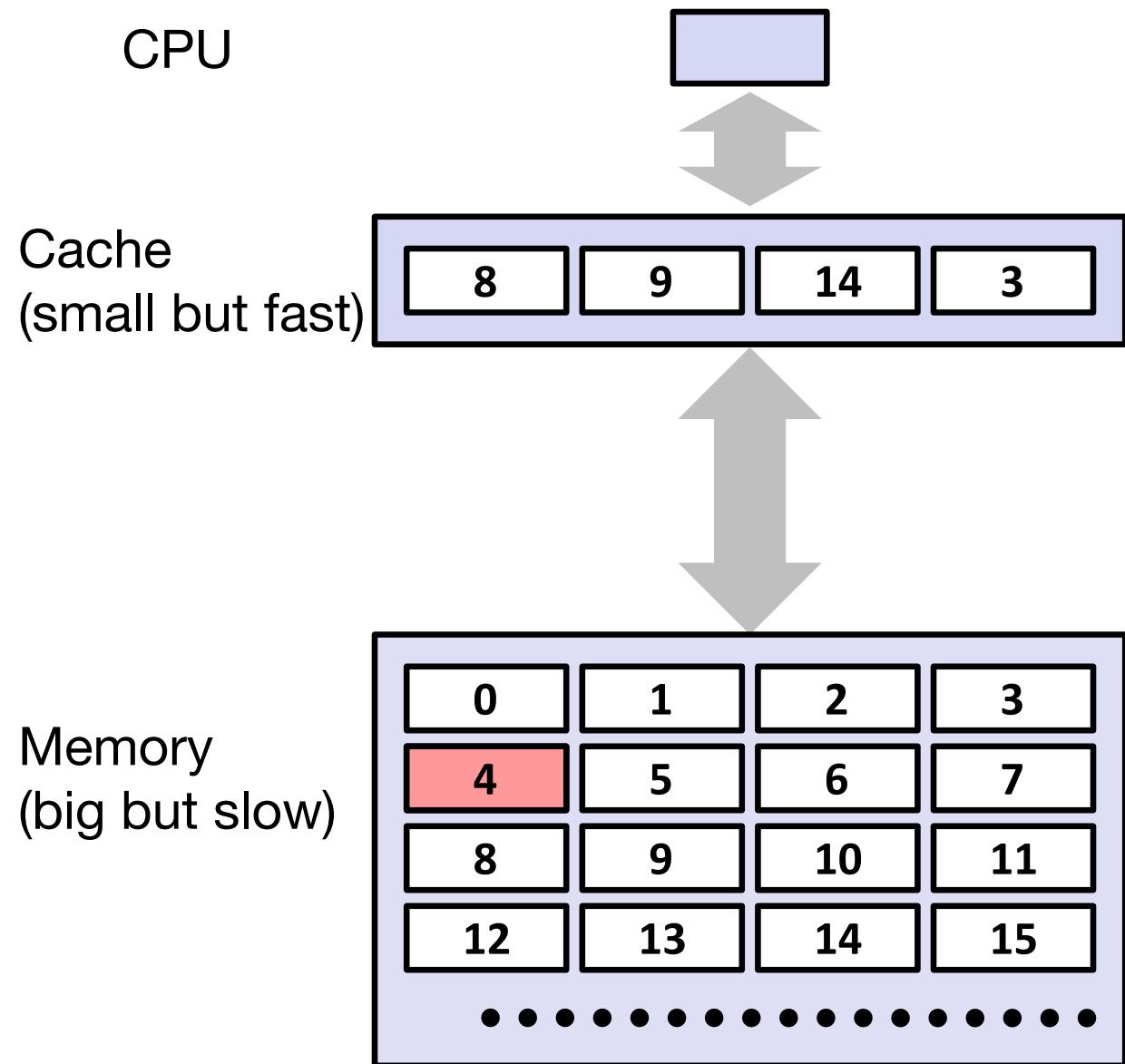
Memory
(big but slow)



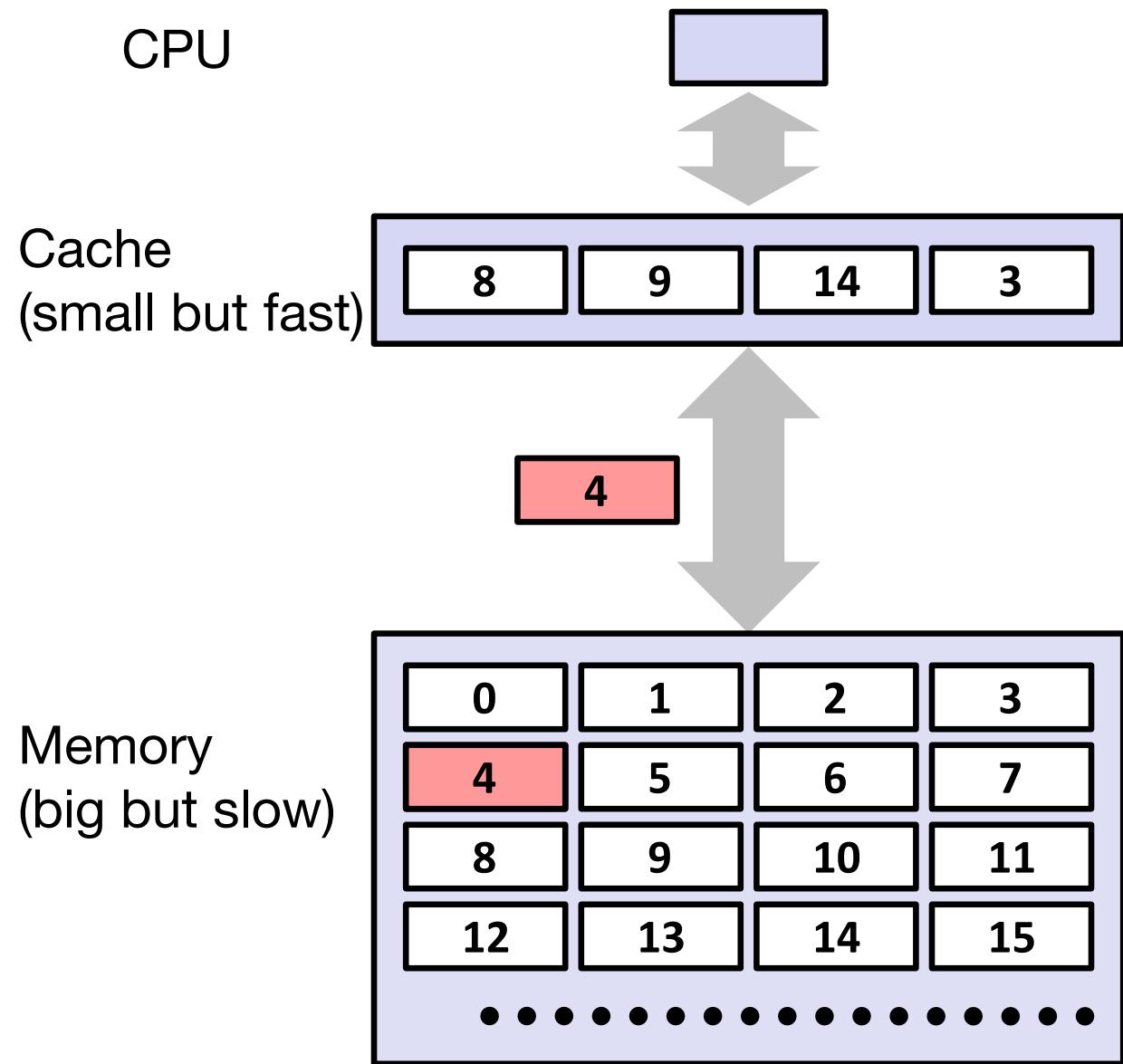
Cache Illustrations



Cache Illustrations

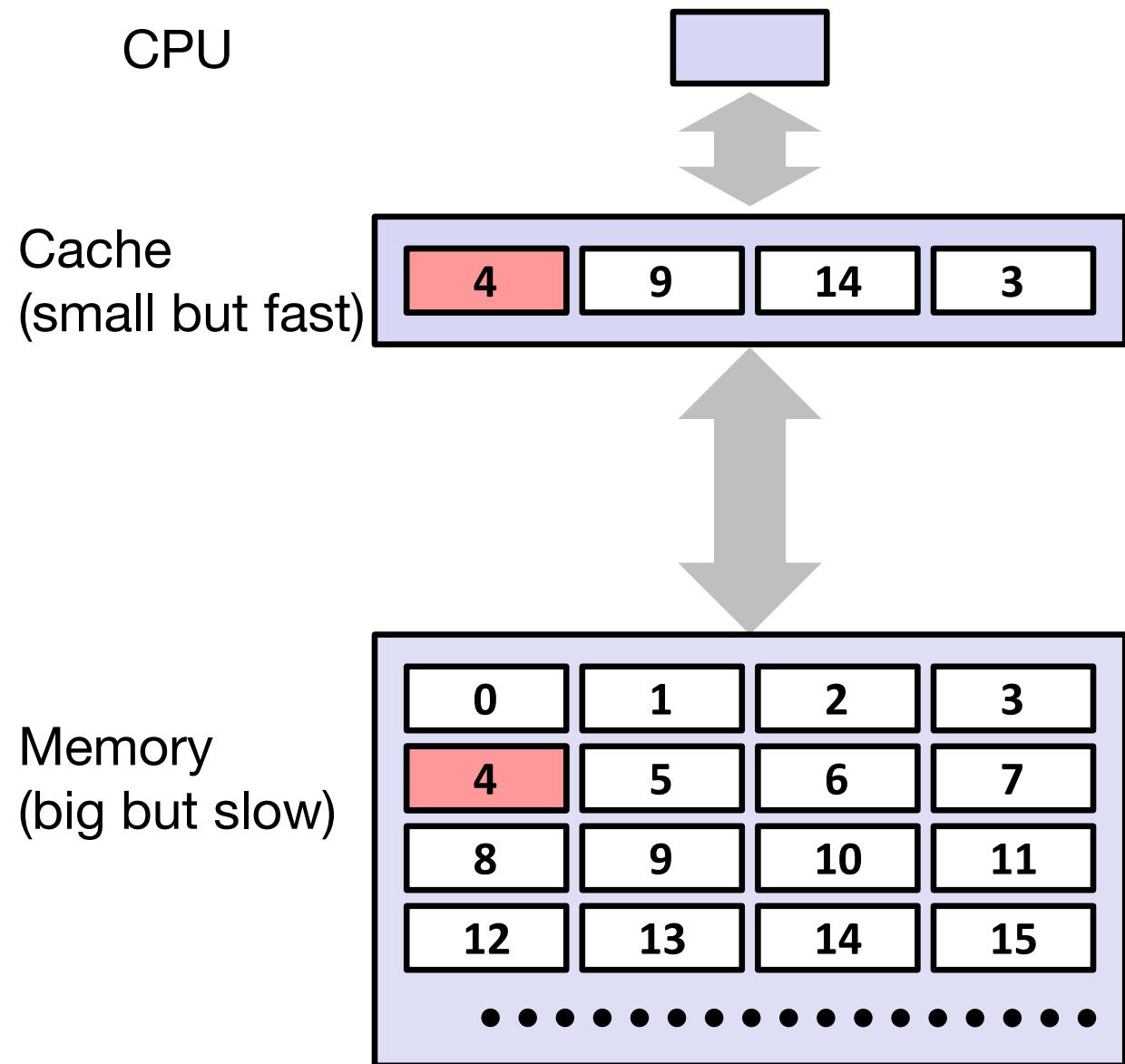


Cache Illustrations

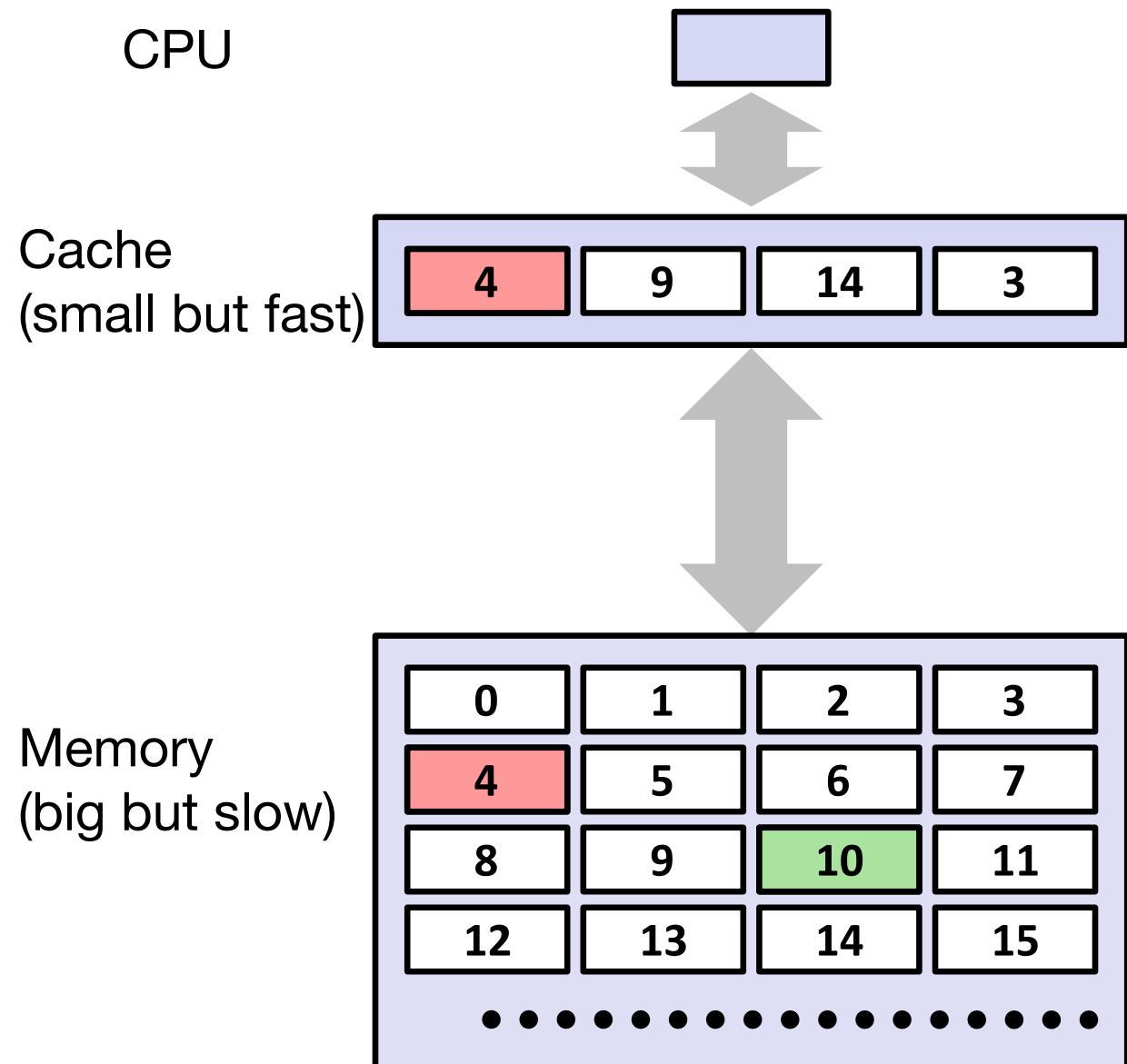


Placement
problem:
Where does
block 4 go?

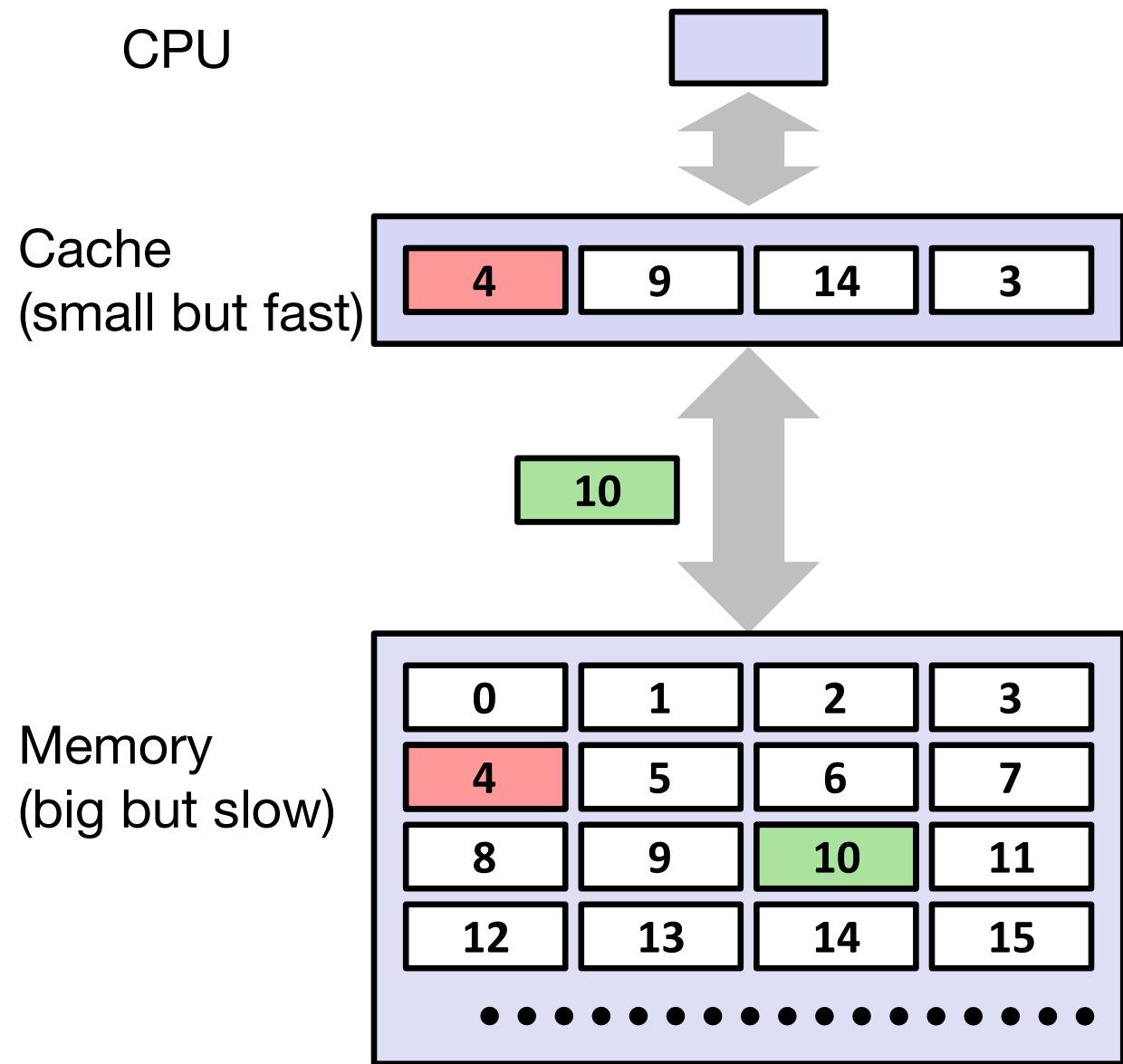
Cache Illustrations



Cache Illustrations

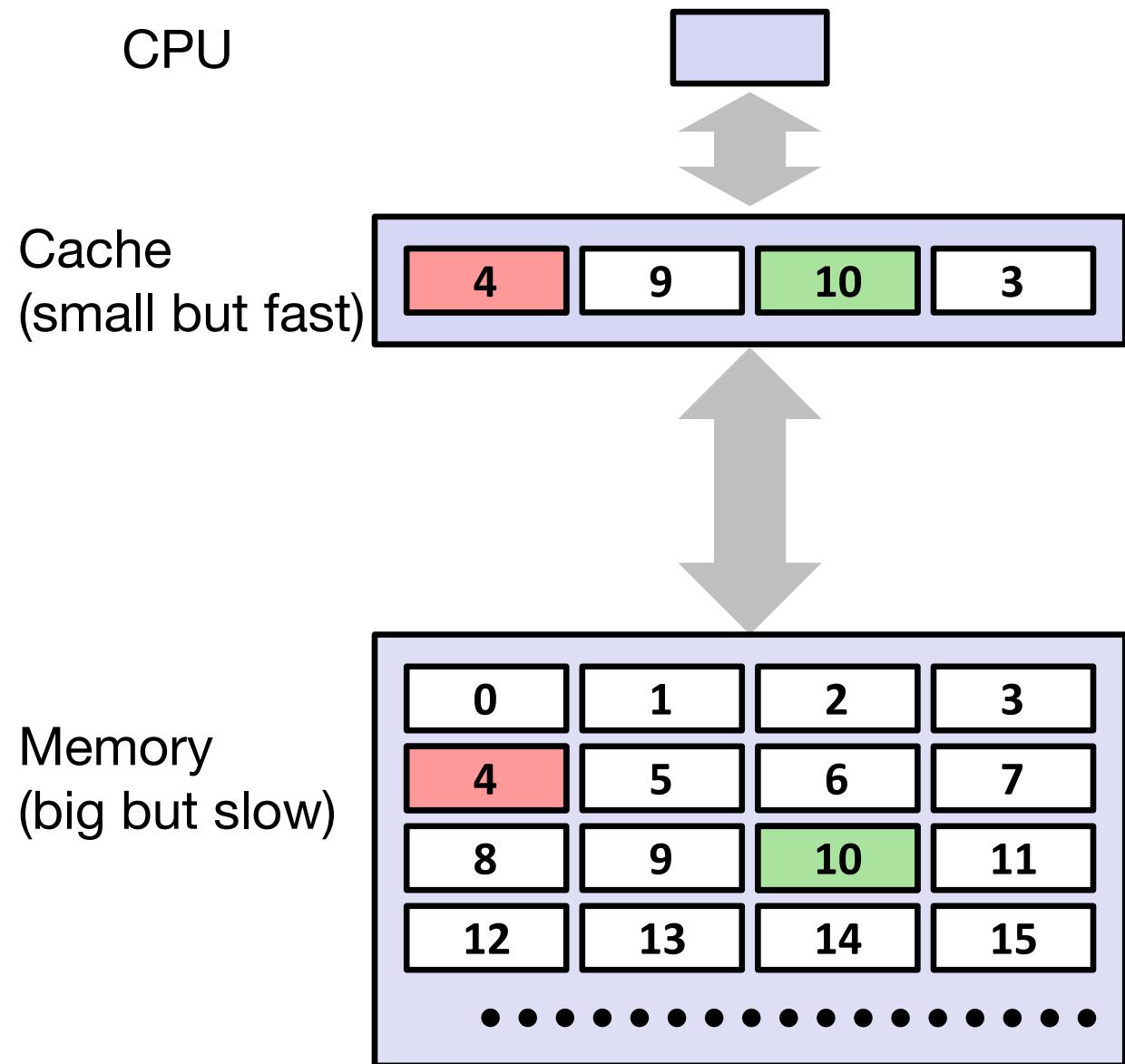


Cache Illustrations

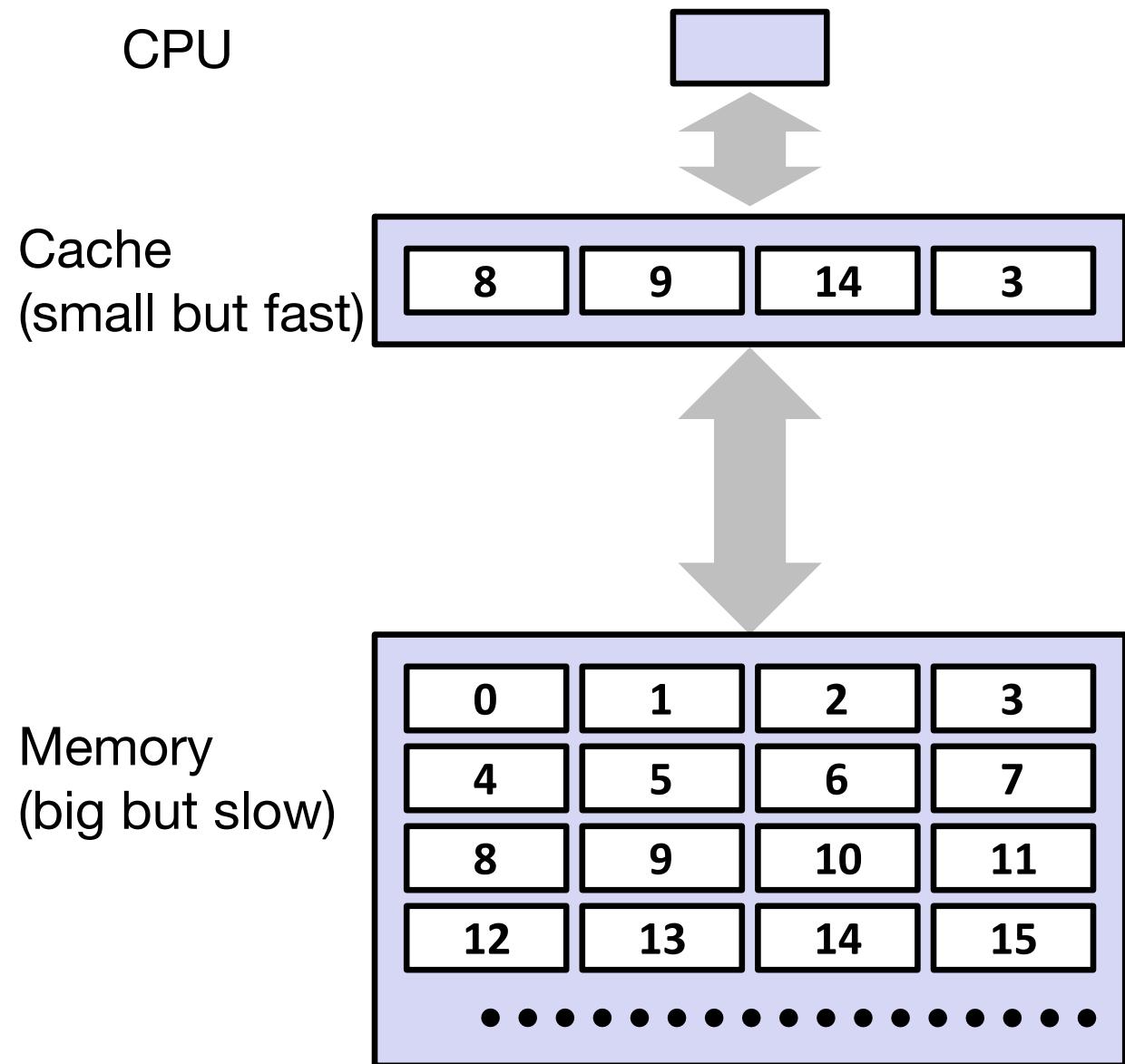


Where does
10 go?

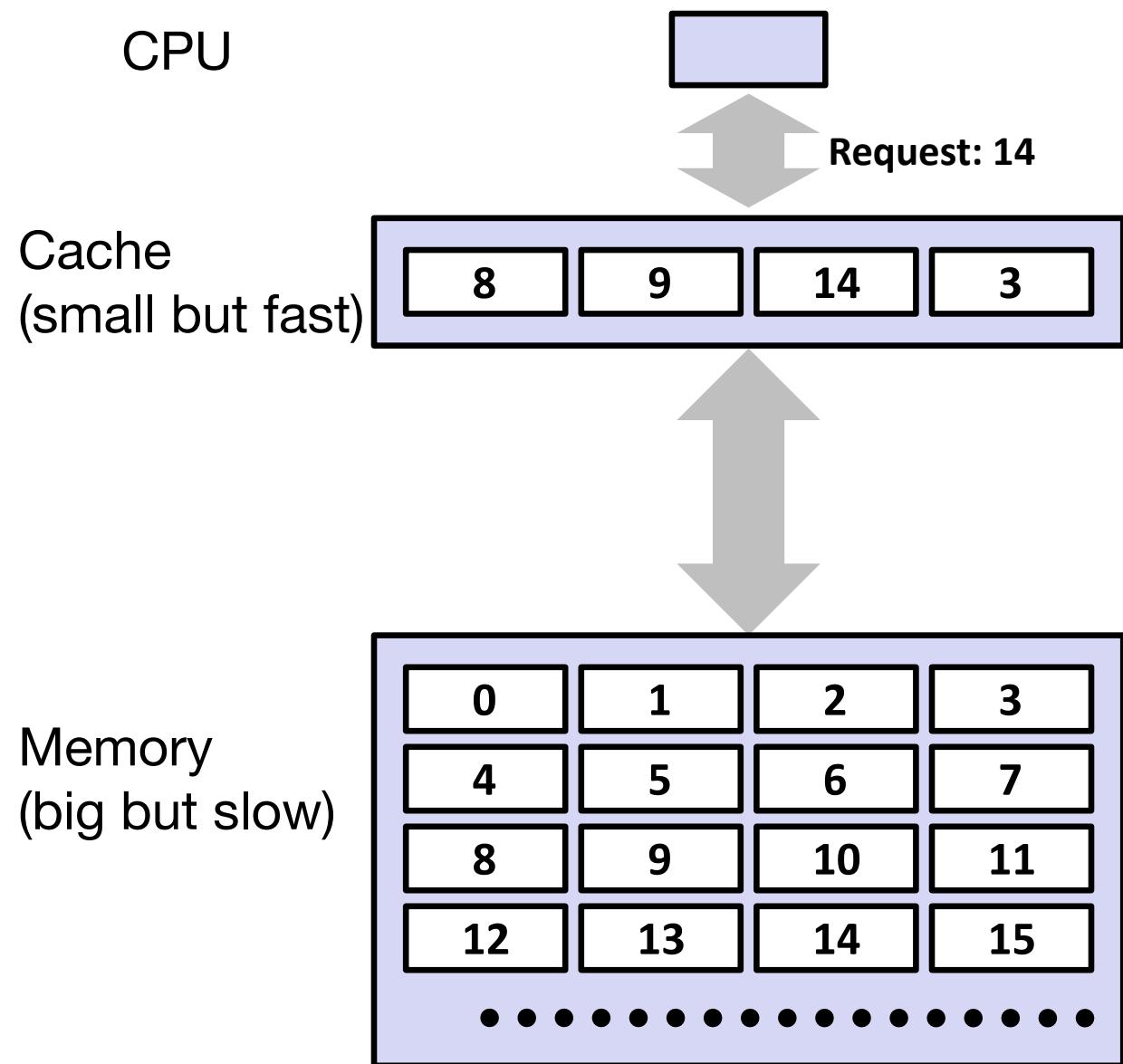
Cache Illustrations



Cache Illustrations: Hit



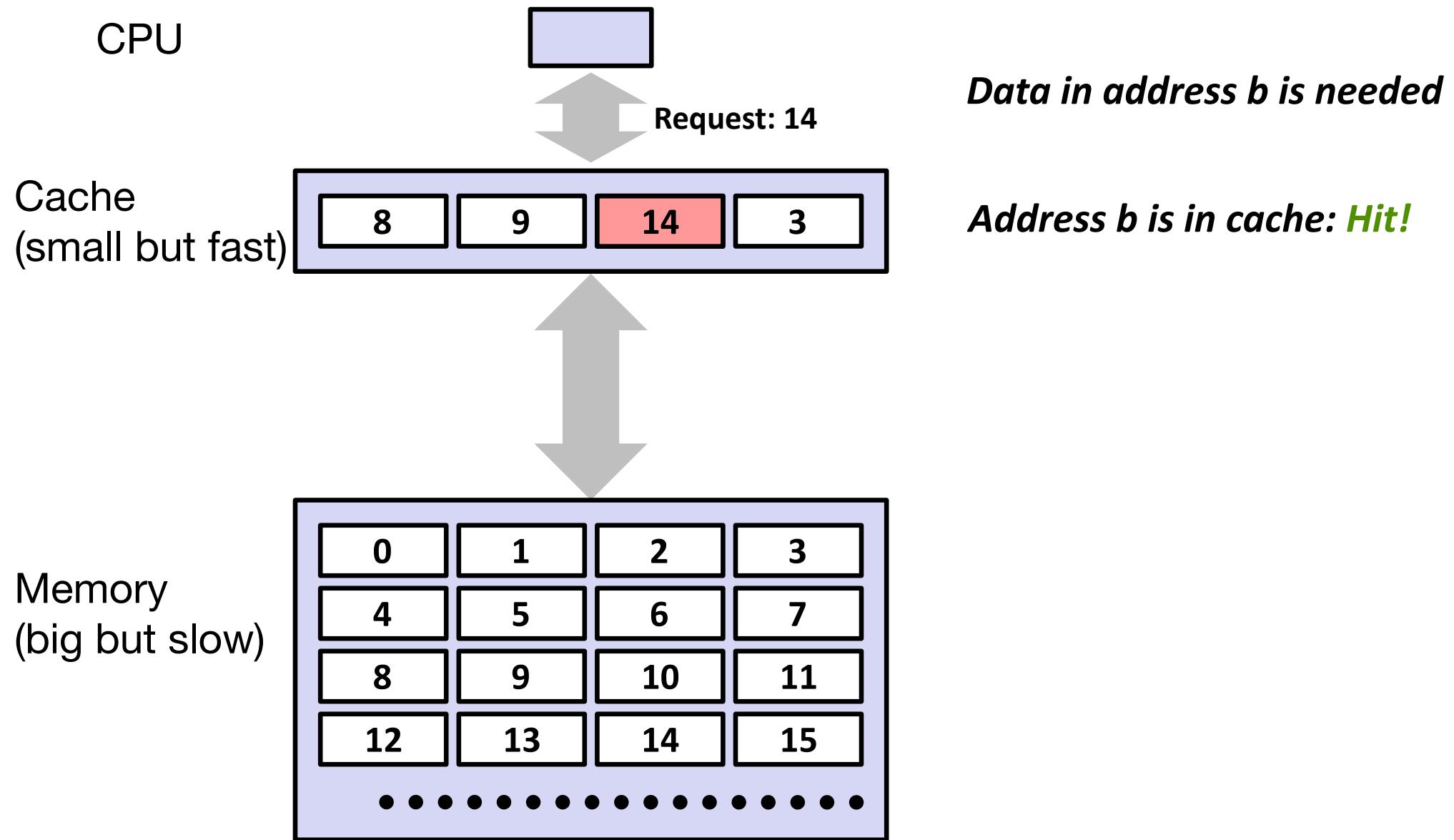
Cache Illustrations: Hit



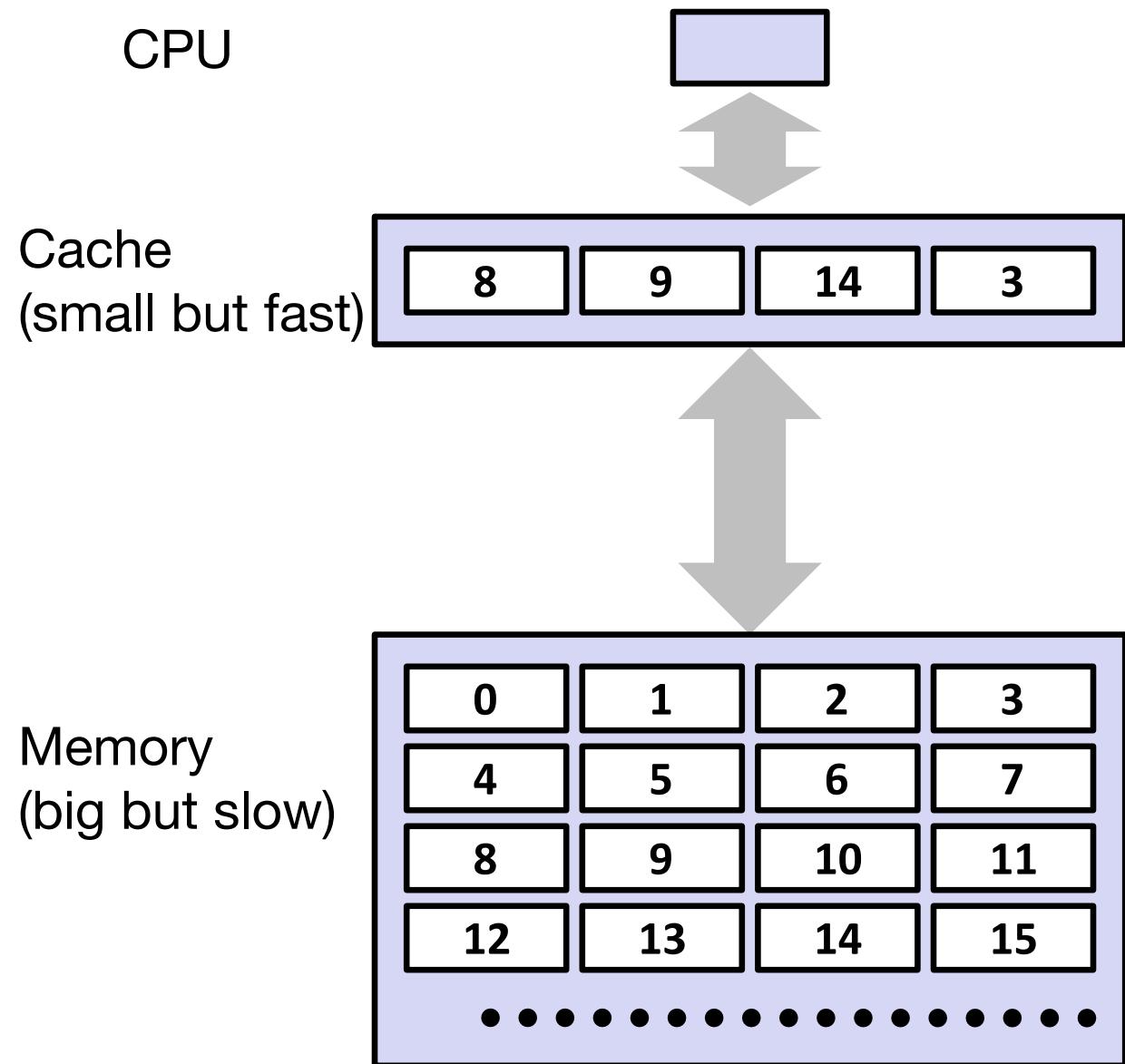
Data in address b is needed

Lookup
problem:
where to look
for block 14?

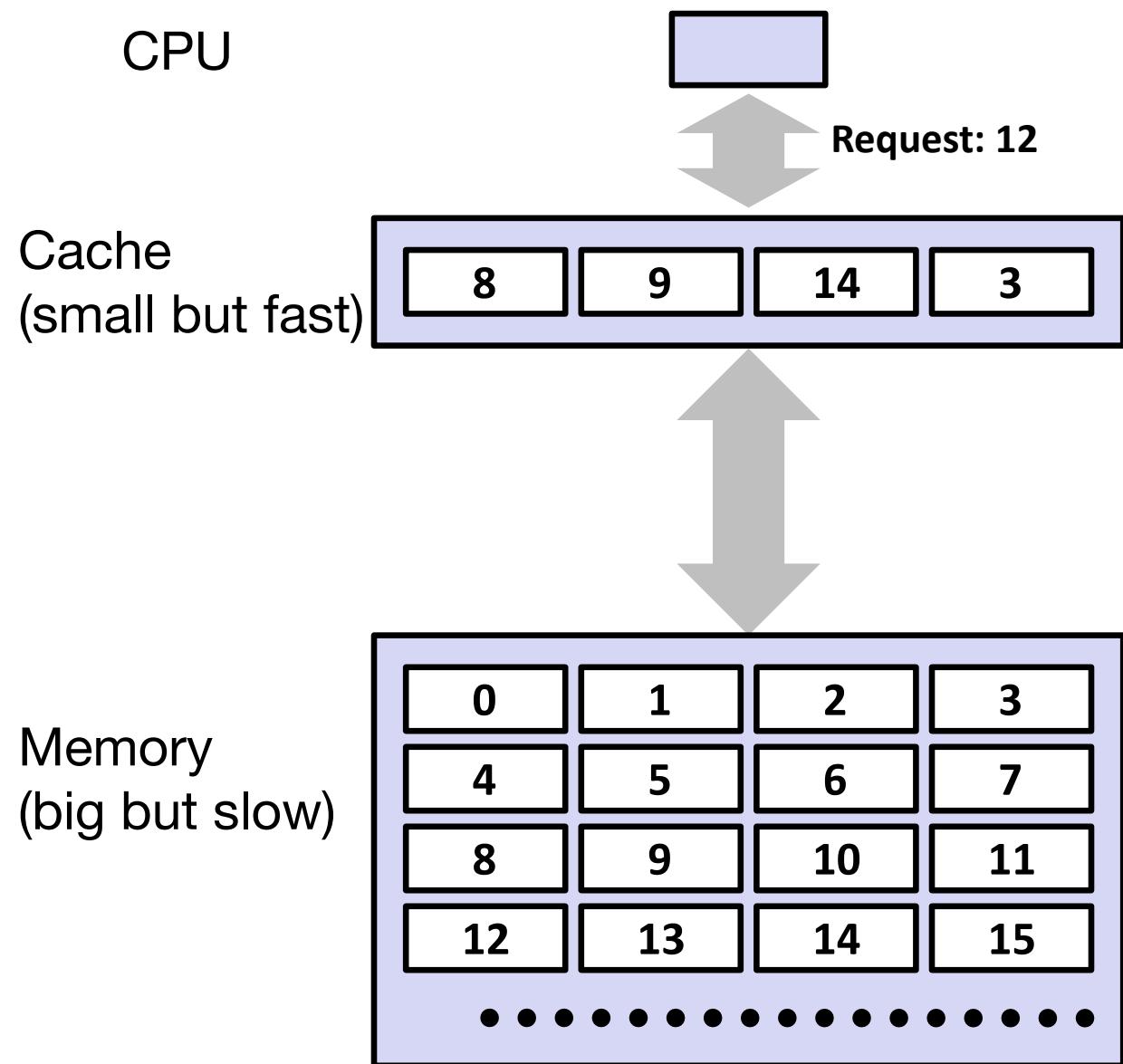
Cache Illustrations: Hit



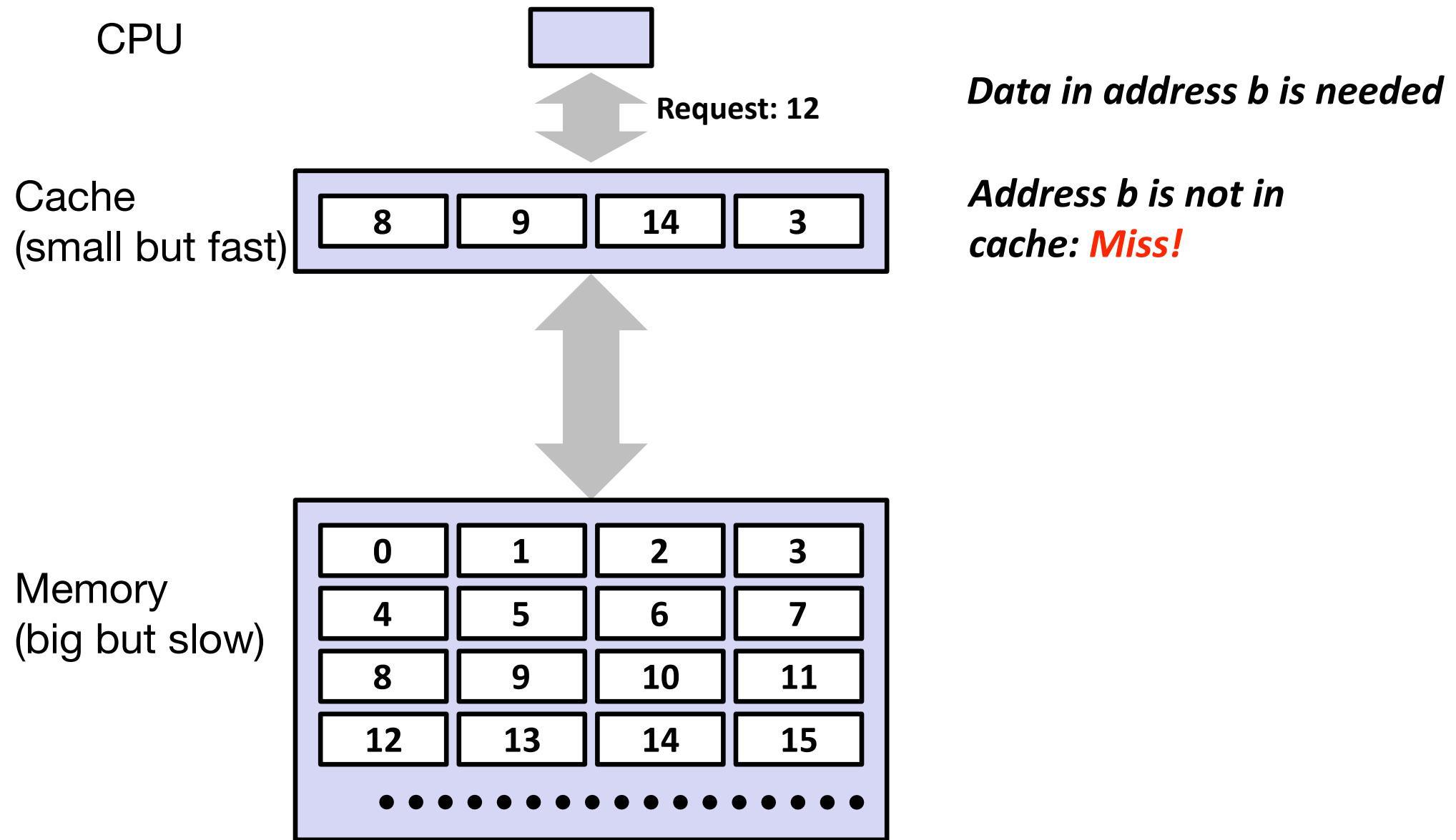
Cache Illustrations: Miss



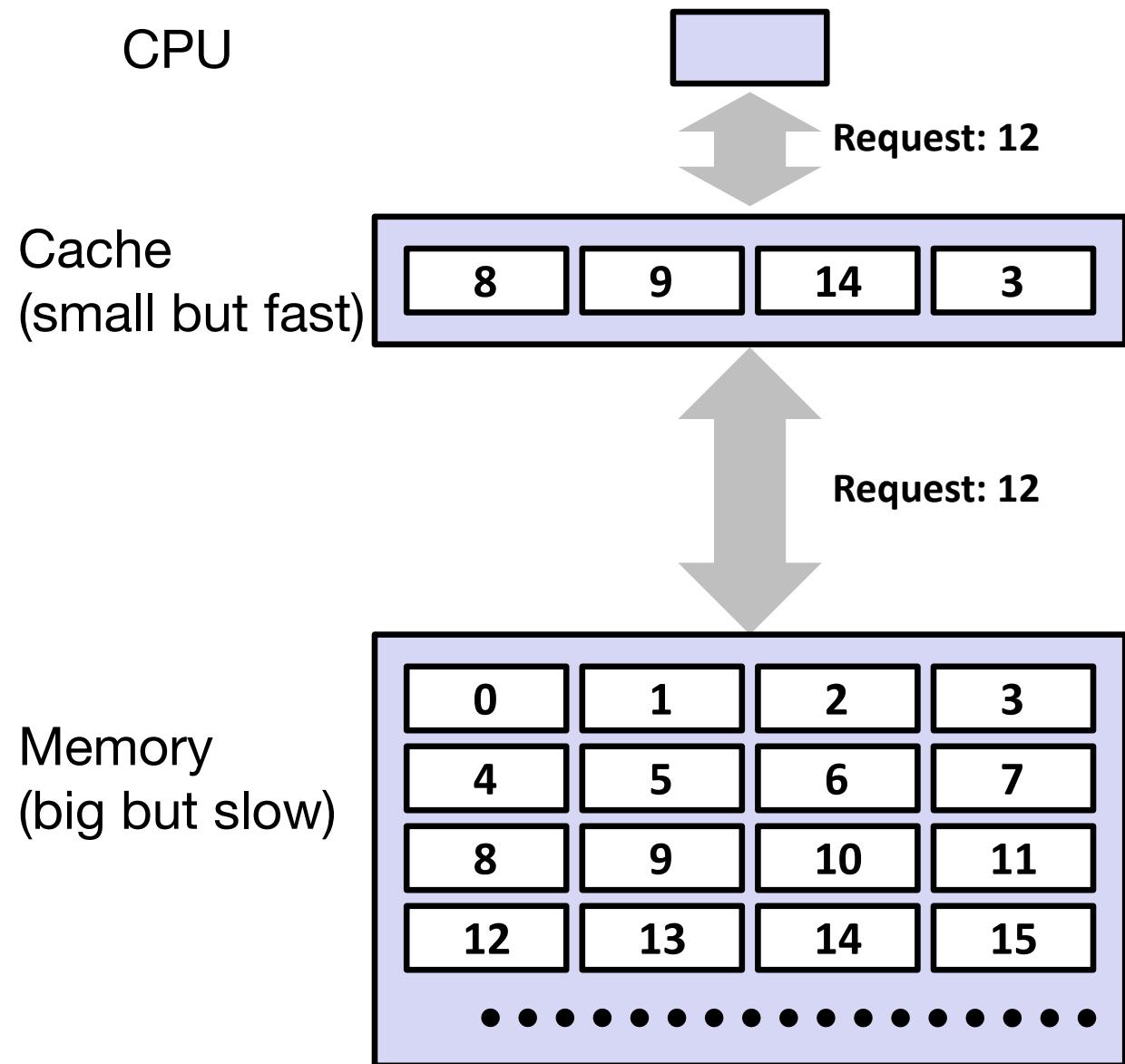
Cache Illustrations: Miss



Cache Illustrations: Miss



Cache Illustrations: Miss

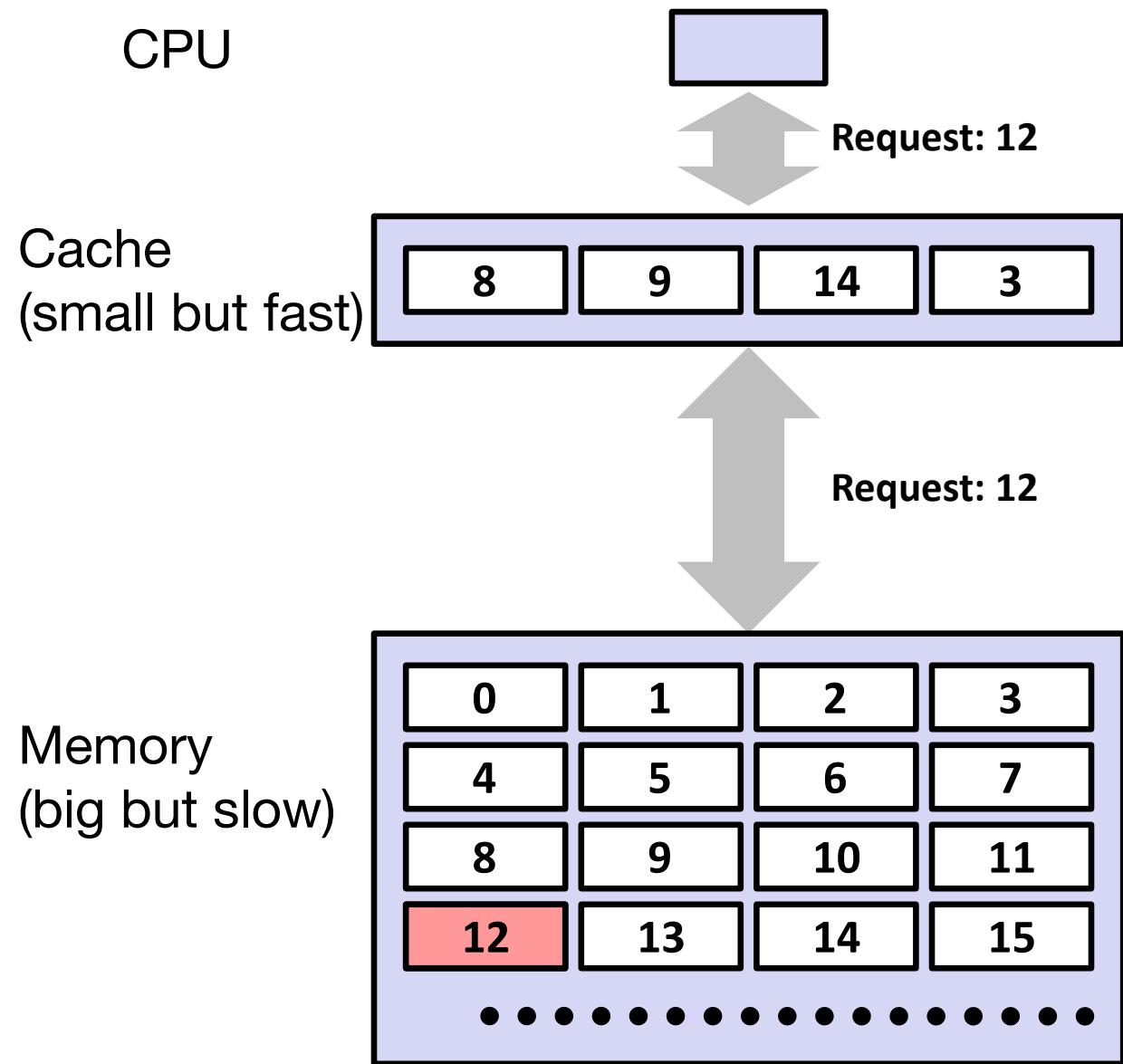


Data in address b is needed

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

Address b is fetched from memory

Cache Illustrations: Miss

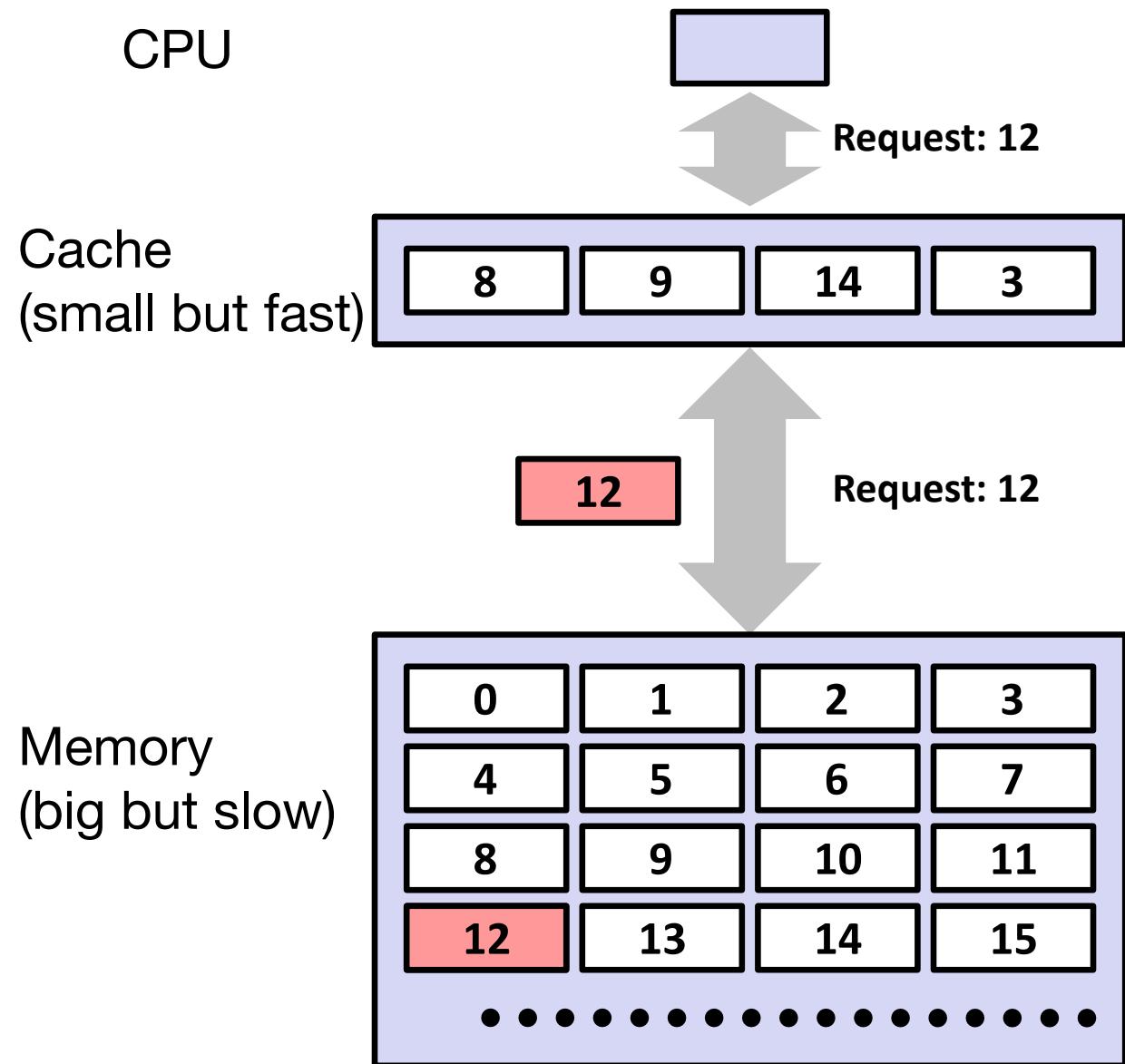


Data in address b is needed

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

Address b is fetched from memory

Cache Illustrations: Miss



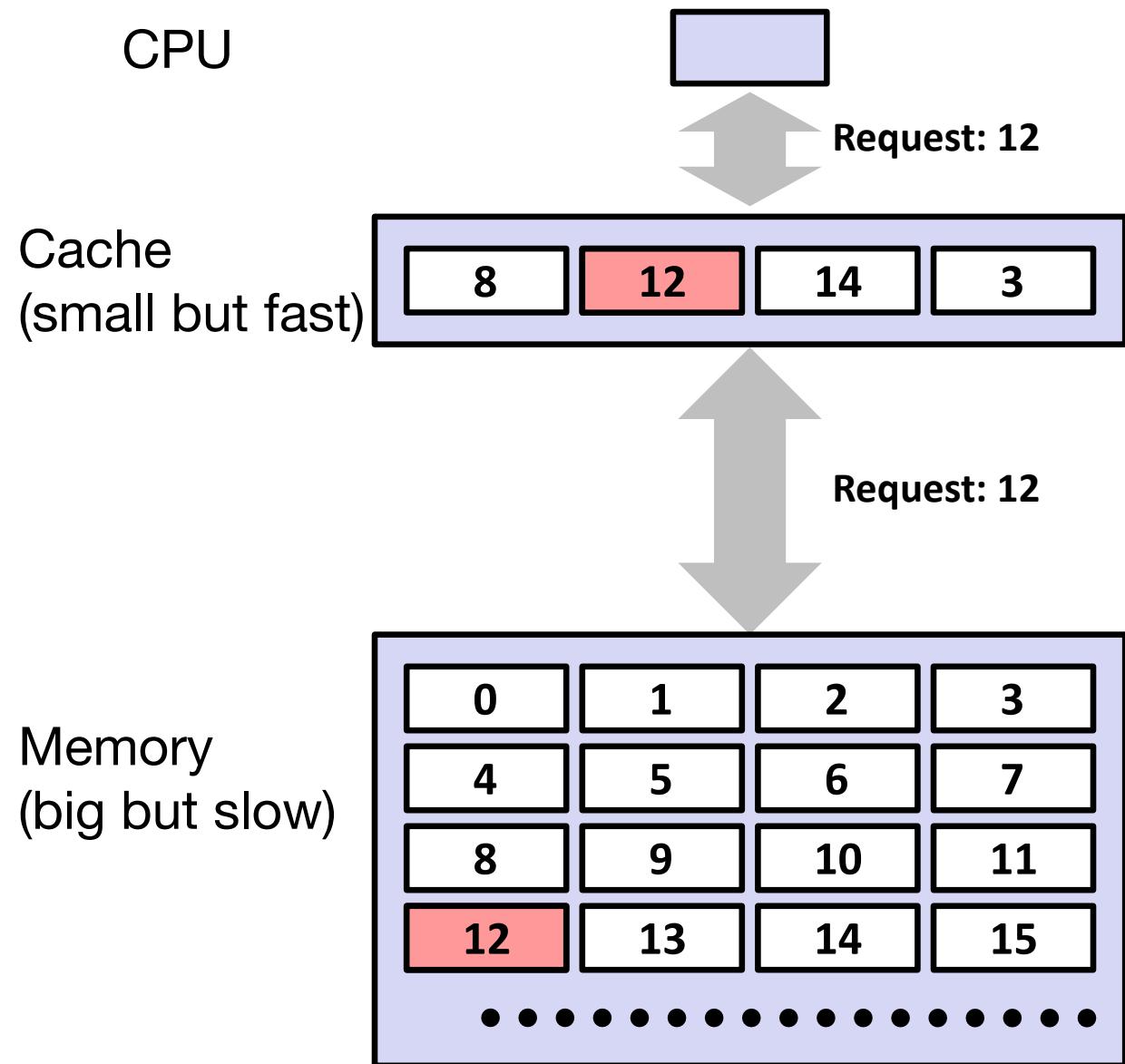
Data in address b is needed

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

Address b is fetched from memory

**Where does
12 go?**

Cache Illustrations: Miss



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

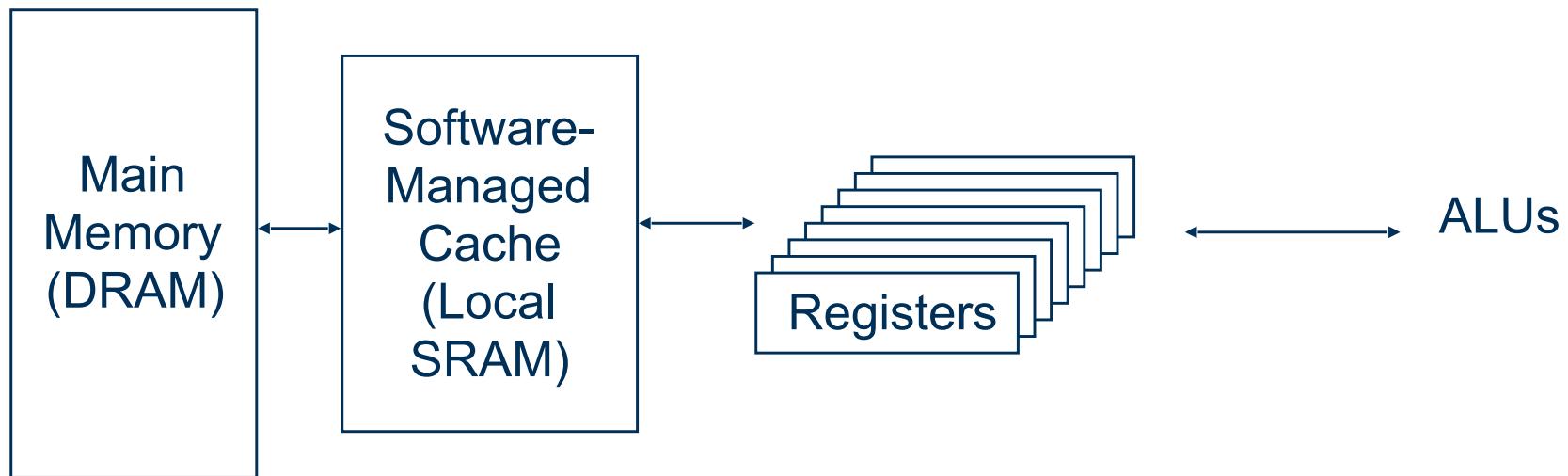
$$\text{Hit Rate} = \frac{\# \text{ Hits}}{\# \text{ Accesses}}$$

Two Fundamental Issues in Cache Management

- Placement
 - Where in the cache is data placed?
 - Or more importantly, how can I find my data?
 - Random placement? Pros vs. cons
- Replacement
 - Given more than one location to place, where is data placed?
 - Or, what to kick out?

Cache Management: Explicit

- Under explicit software control.
- Requirements:
 - Cache and memory have different address spaces
 - Different memory access instructions for cache and memory
- Often hard/impossible to get right



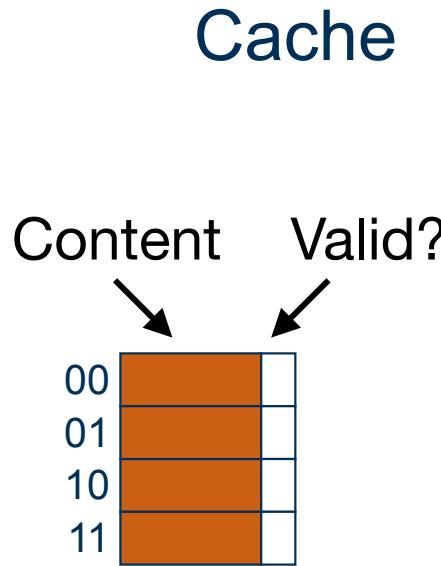
- Examples of software-managed cache
 - Sony Cell Broadband Engine (PS3): **Local store**
 - DSPs: **Scratchpad memory**
 - GPUs: “**Shared memory**”
 - Stream Processors: **Stream register file**

Want Automatic Management

- Software-managed cache is nice, but
 - explicit management is painful
 - often cannot tell statically what will be reused
 - code portability suffers too
- Caches are thus mostly hardware-managed
 - When we say cache today, it almost always means hardware-managed cache
 - Software-managed cache is often called scratchpad memory
 - Cray never believed in hardware-managed cache (“Caches are for wimps!”)

Cray: <https://en.wikipedia.org/wiki/Cray>

Baseline Cache & Memory

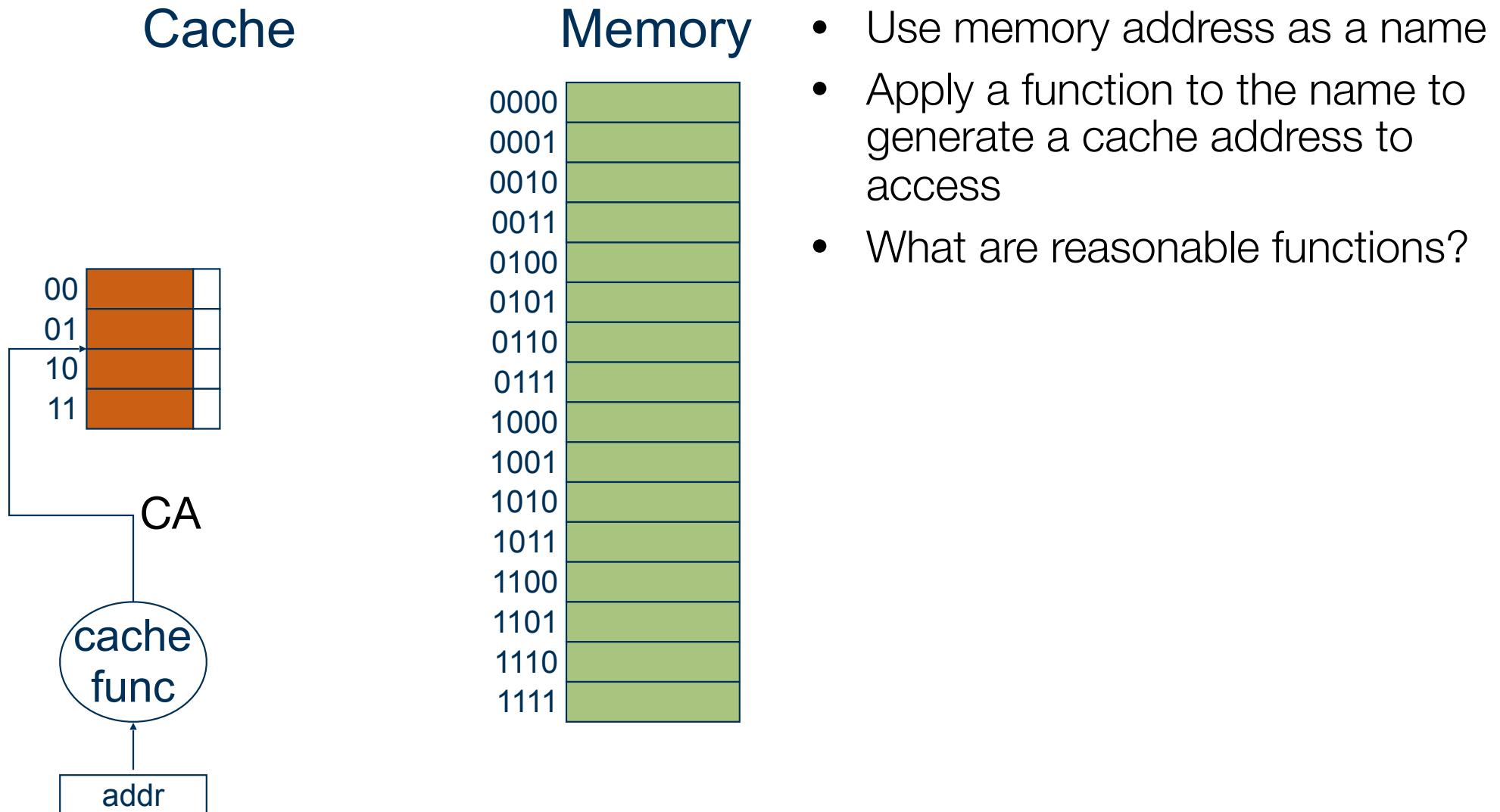


Memory

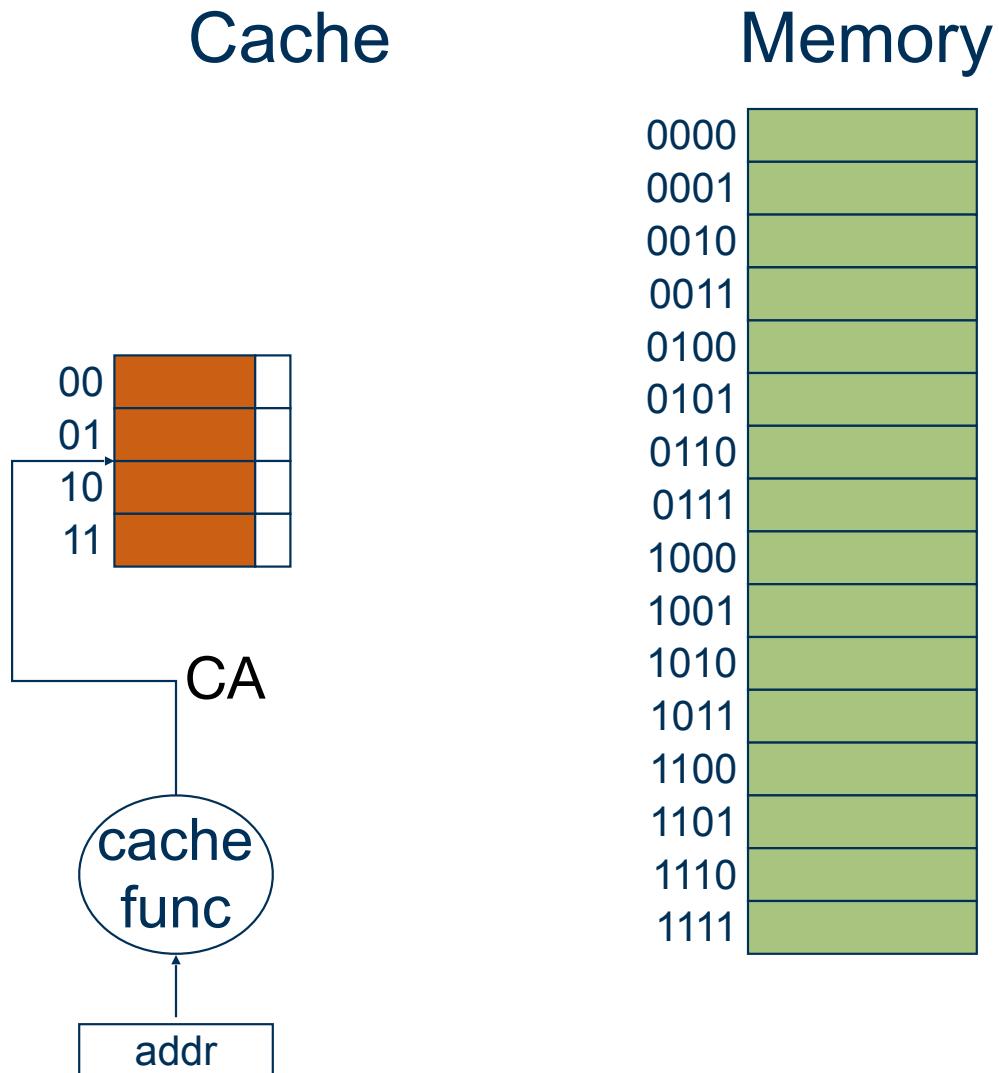
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 4 cache locations
 - Also called **cache-line**
 - Every location has a valid bit
- 16 memory locations
- For now, assume cache location size == memory location size
- Assume each memory location can only reside in one cache-line
- Cache is smaller than memory (obviously)
 - Thus, not all memory locations can be cached at the same time

Cache Placement



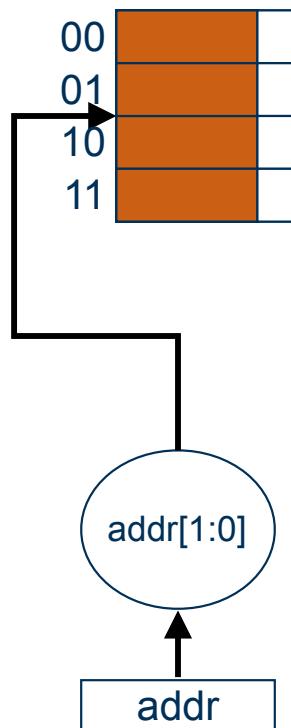
Function to Address Cache



- Simplest function is a subset of address bits
- Six combinations in total
 - $CA = ADDR[3], ADDR[2]$
 - $CA = ADDR[3], ADDR[1]$
 - $CA = ADDR[3], ADDR[0]$
 - $CA = ADDR[2], ADDR[1]$
 - $CA = ADDR[2], ADDR[0]$
 - $CA = ADDR[1], ADDR[0]$
- Direct-Mapped Cache
 - $CA = ADDR[1], ADDR[0]$

Direct-Mapped Cache

Cache

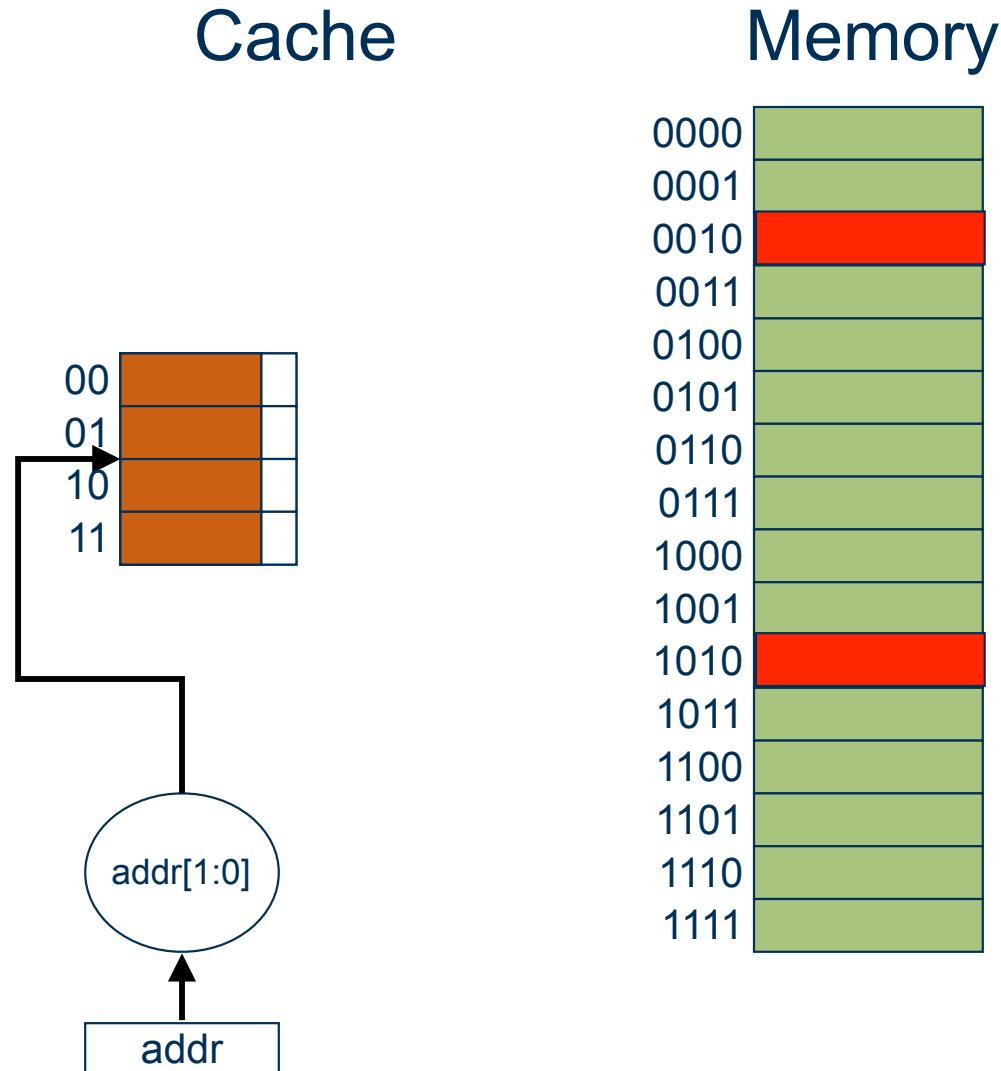


Memory

- Direct-Mapped Cache

- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

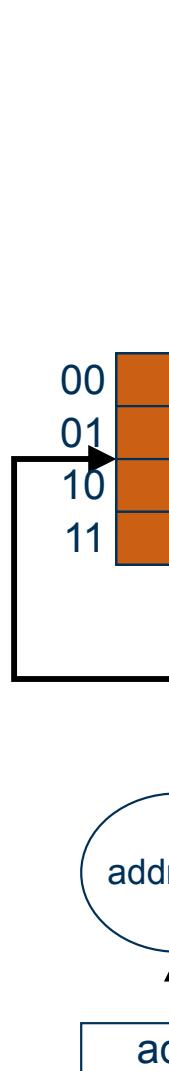
Direct-Mapped Cache



- Direct-Mapped Cache
 - $CA = ADDR[1], ADDR[0]$
 - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
 - E.g., 0010 and 1010

How many / which blocks map to the 2nd cache block, i.e. $CA=01$?

Direct-Mapped Cache



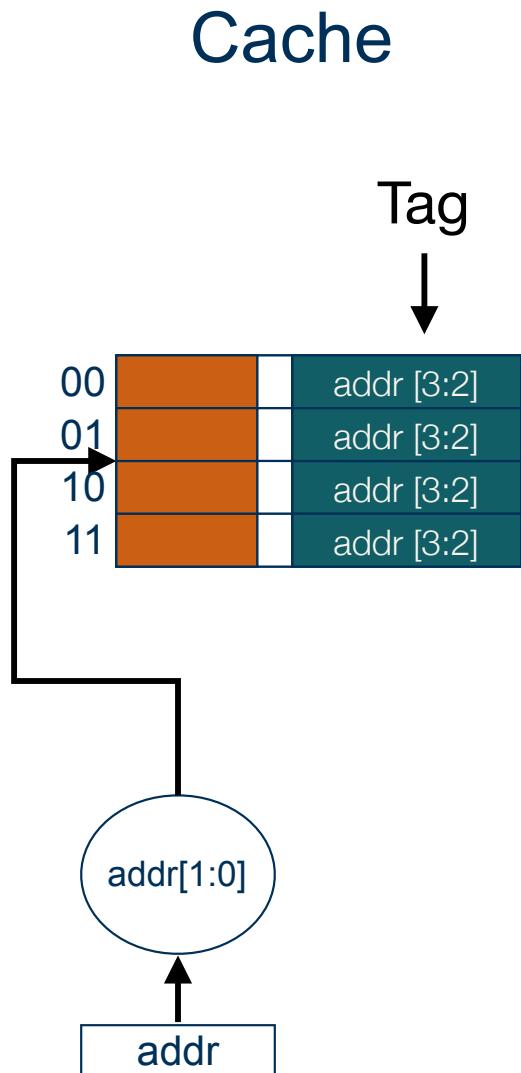
Memory

0000	green
0001	green
0010	red
0011	green
0100	green
0101	green
0110	green
0111	green
1000	green
1001	green
1010	red
1011	green
1100	green
1101	green
1110	green
1111	green

- Direct-Mapped Cache

- $CA = ADDR[1], ADDR[0]$
- 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
 - $ADDR[3]$ and $ADDR[2]$ in this particular example

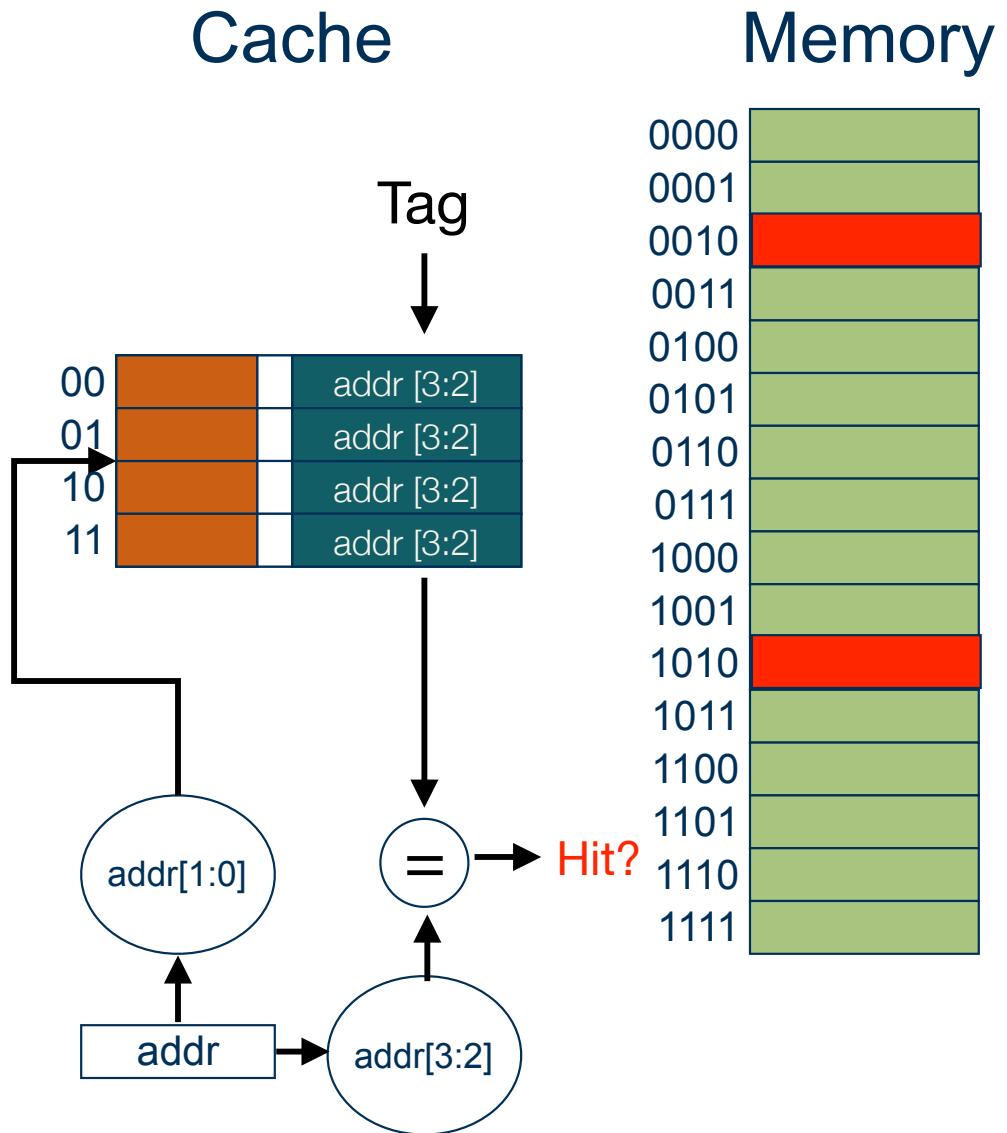
Direct-Mapped Cache



Memory

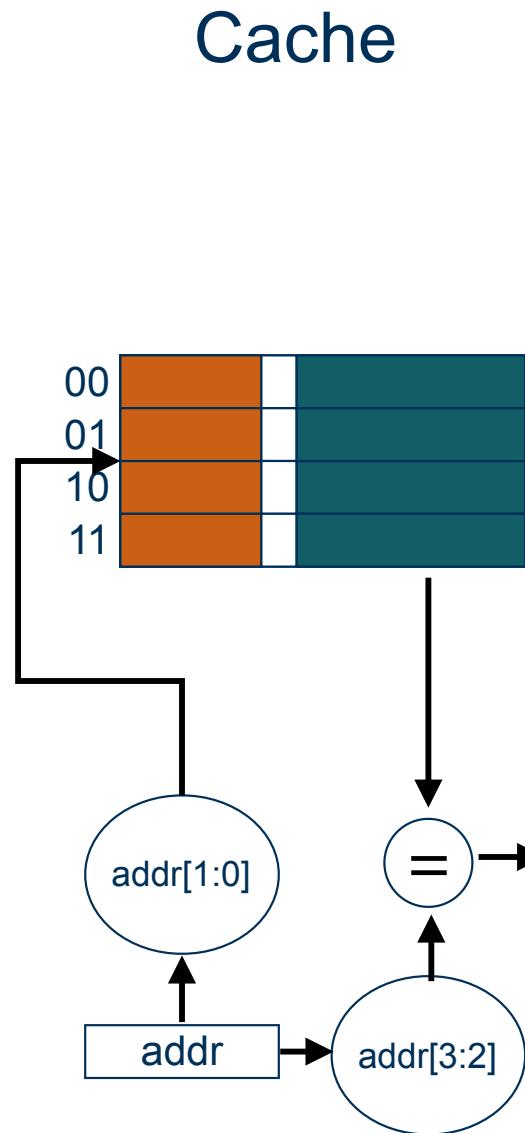
- Direct-Mapped Cache
 - $CA = ADDR[1], ADDR[0]$
 - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
 - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
 - Add a tag field for that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

Direct-Mapped Cache



- Direct-Mapped Cache
 - $CA = ADDR[1], ADDR[0]$
 - 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
 - $ADDR[3]$ and $ADDR[2]$ in this particular example

Example: Direct-Mapped Cache

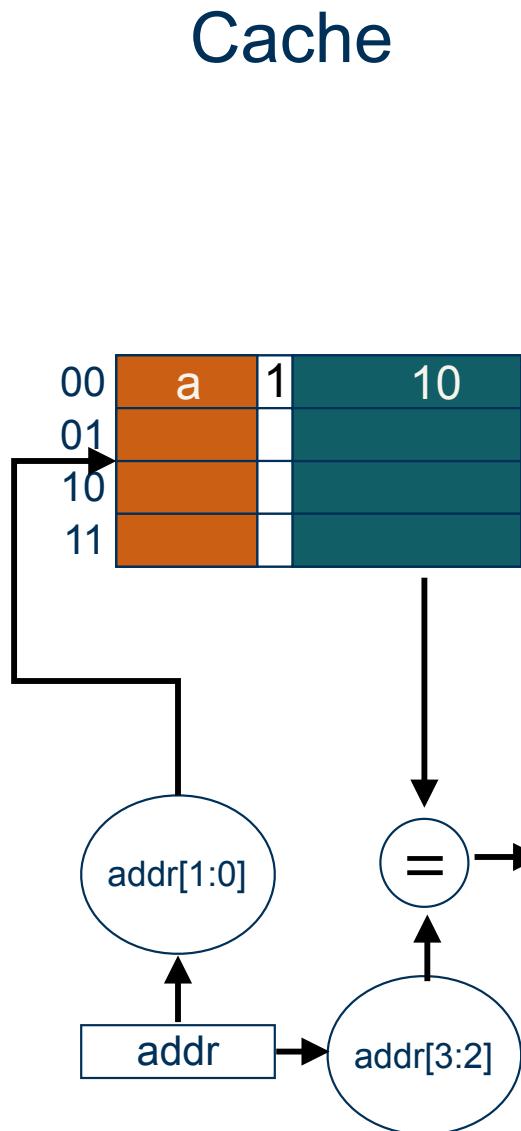


Memory Address	Content
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

```
for (i = 0; i < 4; ++i) {  
    A += mem[i];  
}  
  
for (i = 0; i < 4; ++i) {  
    B *= (mem[i] + A);  
}
```

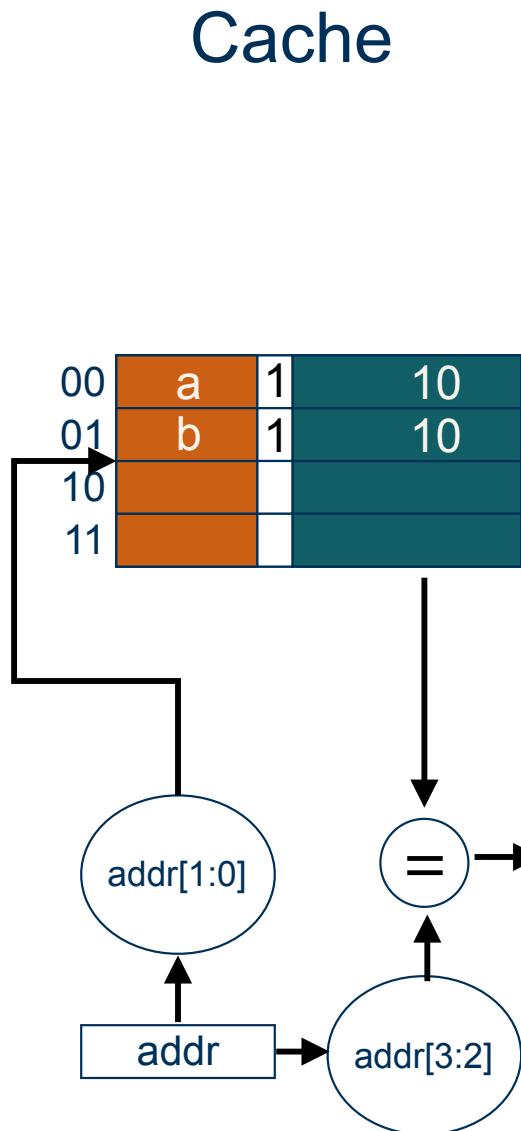
- Assume mem == 0b1000
- Read 0b1000
- Read 0b1001
- Read 0b1010
- Read 0b1011
- Read 0b1000; cache hit?

Example: Direct-Mapped Cache





Example: Direct-Mapped Cache



Memory

```
for (i = 0; i < 4; ++i) {
```

```
    A += mem[i];
```

```
}
```

```
for (i = 0; i < 4; ++i) {
```

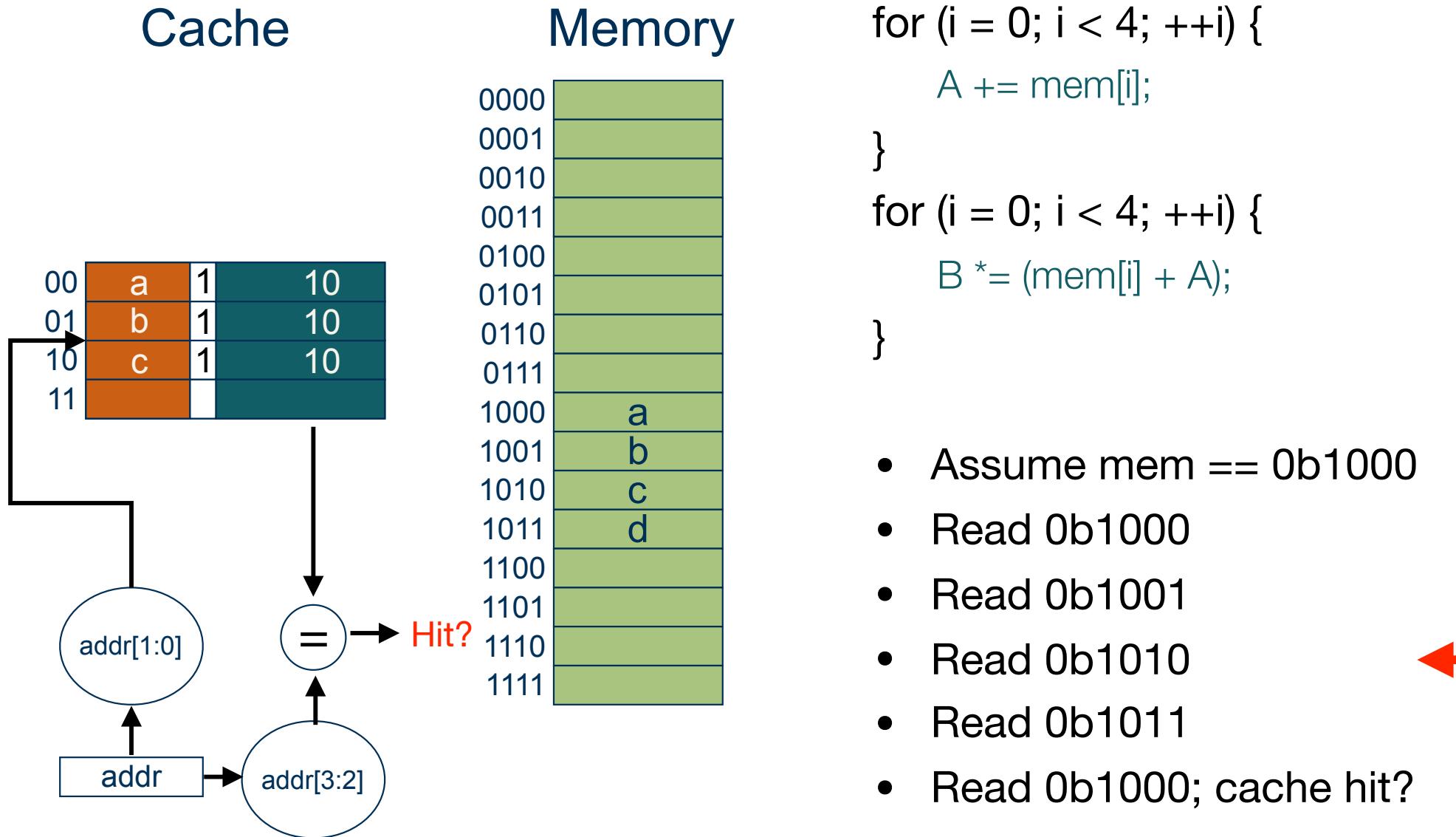
```
    B *= (mem[i] + A);
```

```
}
```

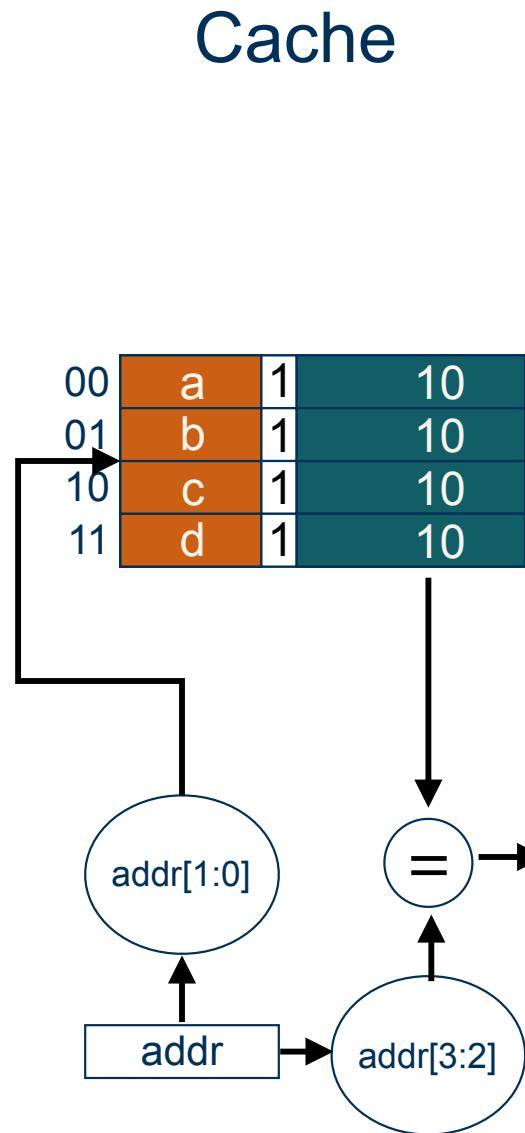
- Assume $mem == 0b1000$
- Read 0b1000
- Read 0b1001
- Read 0b1010
- Read 0b1011
- Read 0b1000; cache hit?



Example: Direct-Mapped Cache



Example: Direct-Mapped Cache



Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

```
for (i = 0; i < 4; ++i) {
```

```
    A += mem[i];
```

```
}
```

```
for (i = 0; i < 4; ++i) {
```

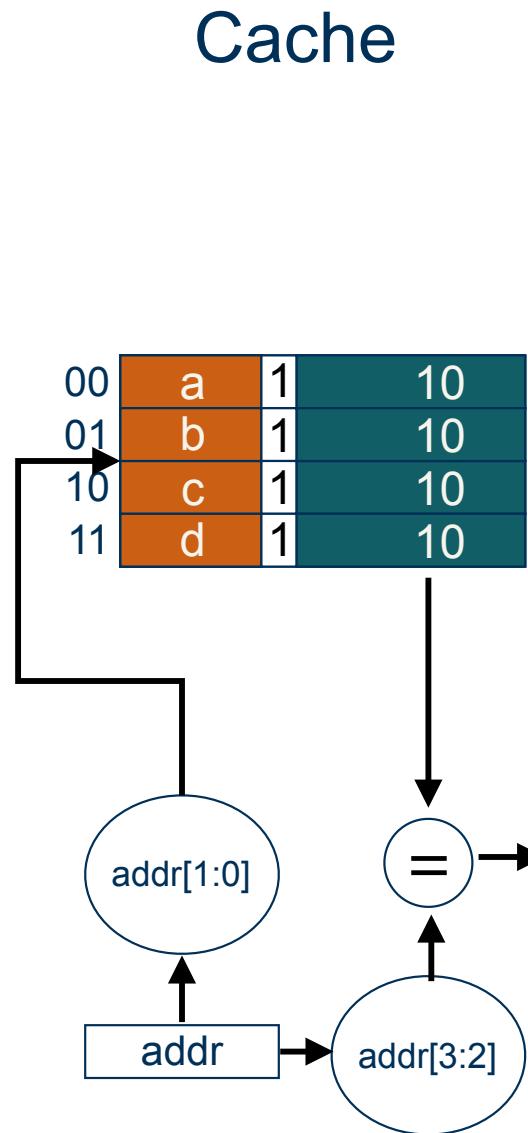
```
    B *= (mem[i] + A);
```

```
}
```

- Assume mem == 0b1000
- Read 0b1000
- Read 0b1001
- Read 0b1010
- Read 0b1011
- Read 0b1000; cache hit?



Example: Direct-Mapped Cache



Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

```
for (i = 0; i < 4; ++i) {
```

```
    A += mem[i];
```

```
}
```

```
for (i = 0; i < 4; ++i) {
```

```
    B *= (mem[i] + A);
```

```
}
```

- Assume mem == 0b1000

- Read 0b1000

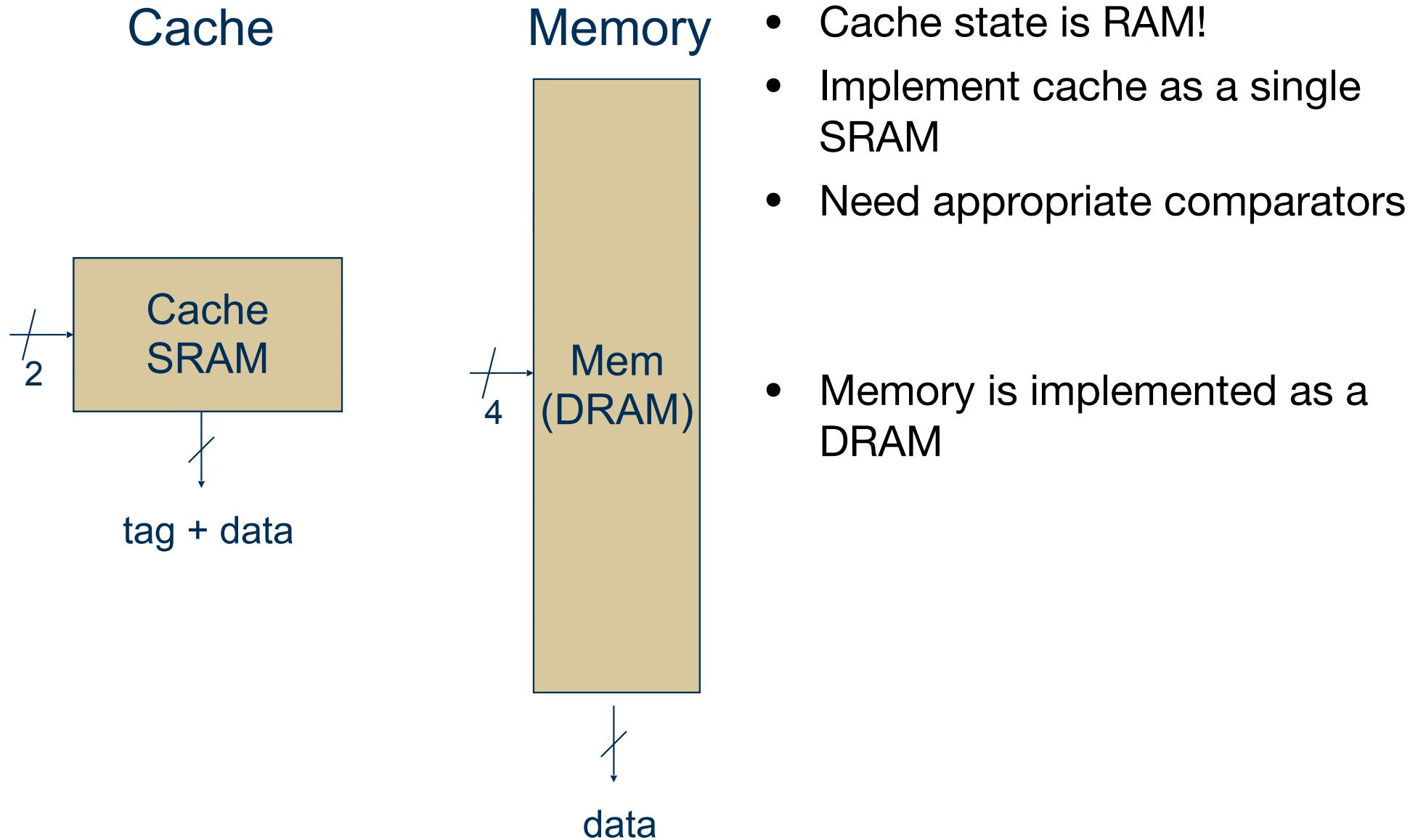
- Read 0b1001

- Read 0b1010

- Read 0b1011

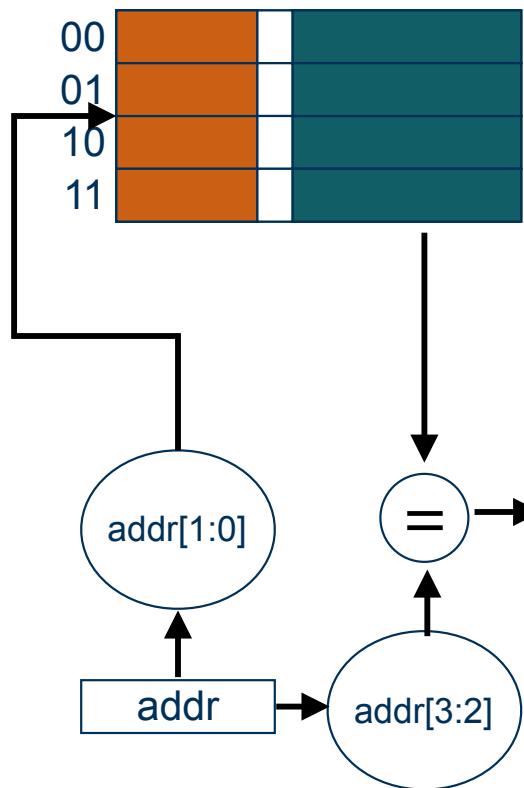
- Read 0b1000; cache hit? ←

One Possible Direct-Mapped Cache Implementation



Conflicts

Cache



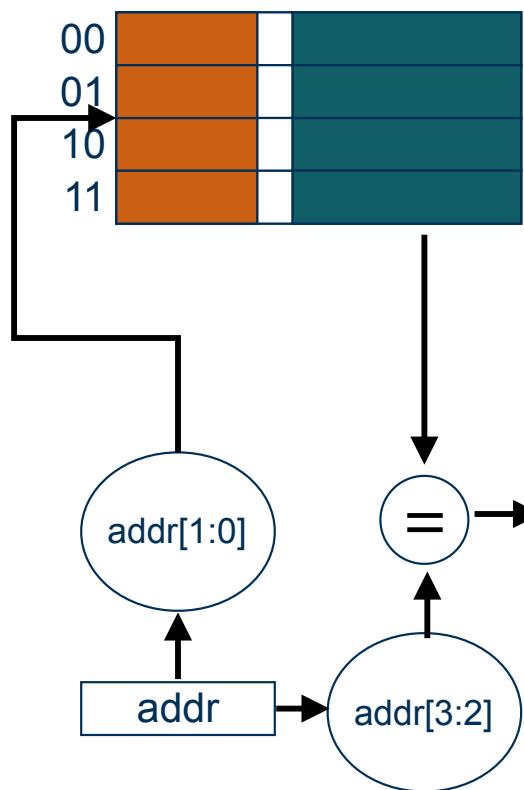
Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

Conflicts

Cache



Memory

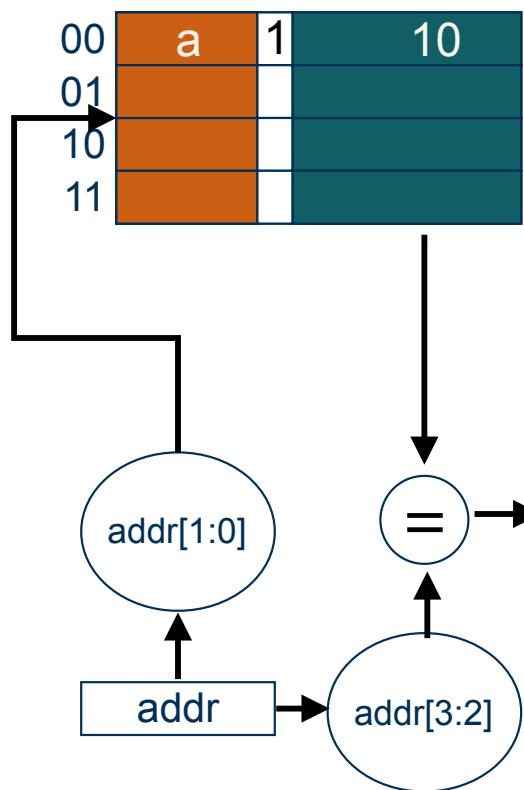
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000

Conflicts

Cache



Memory

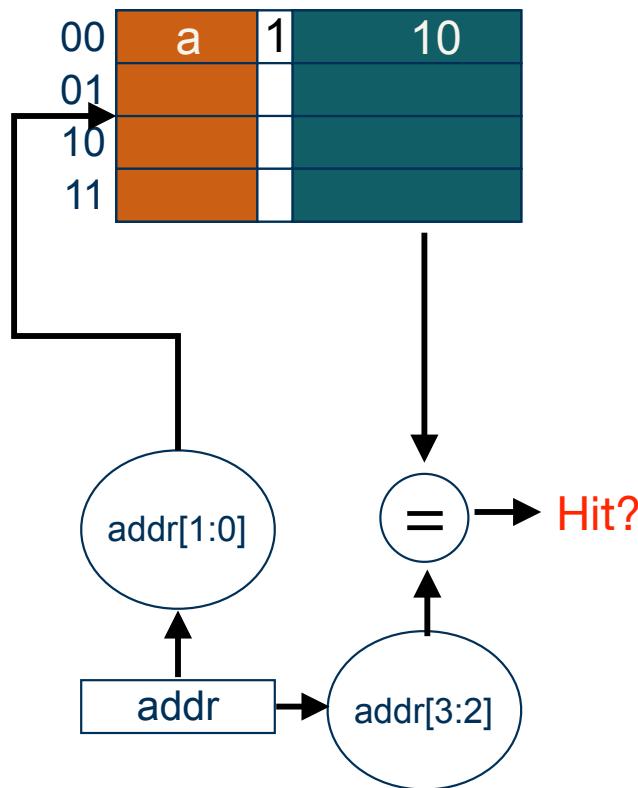
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000

Conflicts

Cache



Memory

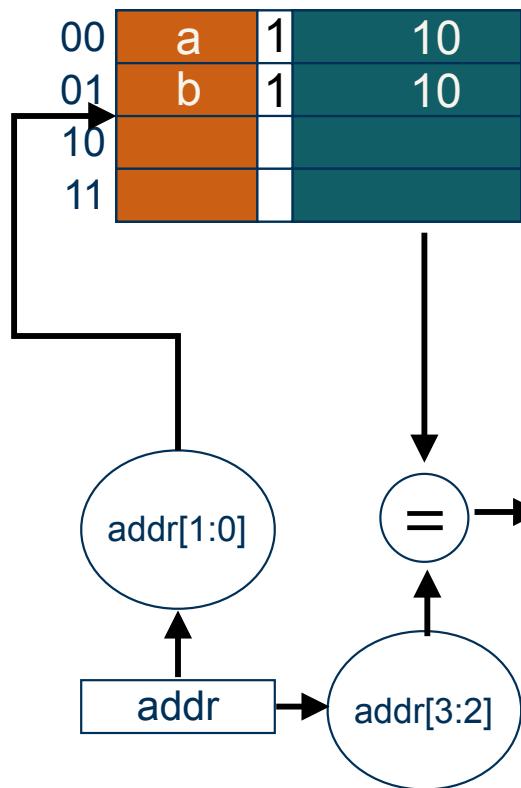
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000
- Read 1001

Conflicts

Cache



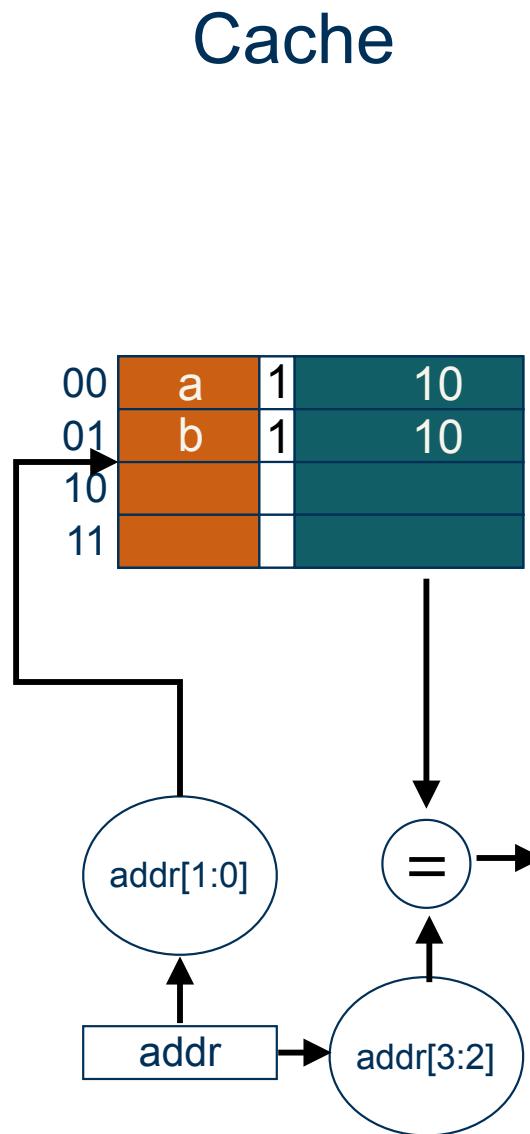
Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000
- Read 1001

Conflicts



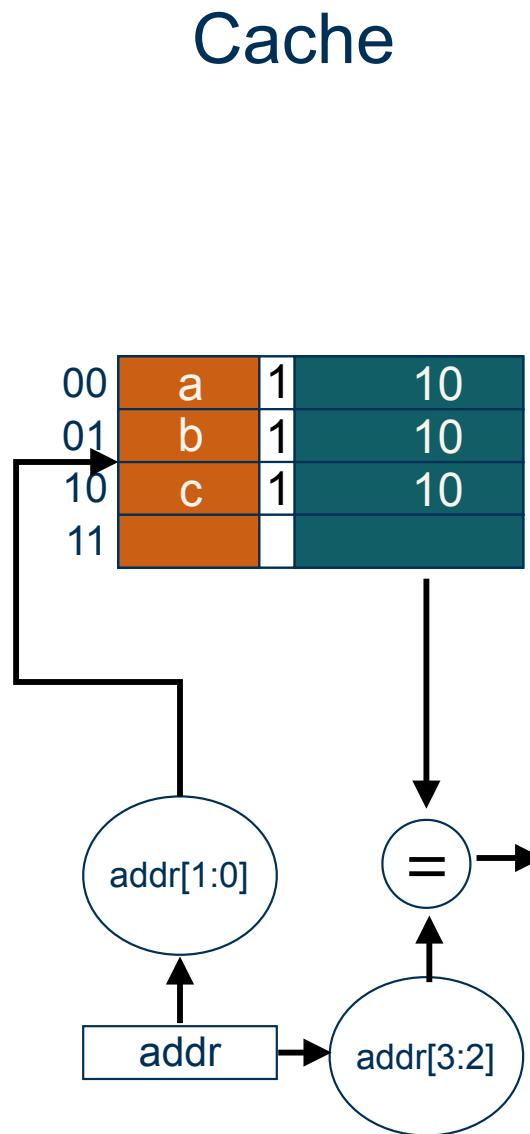
Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

Conflicts

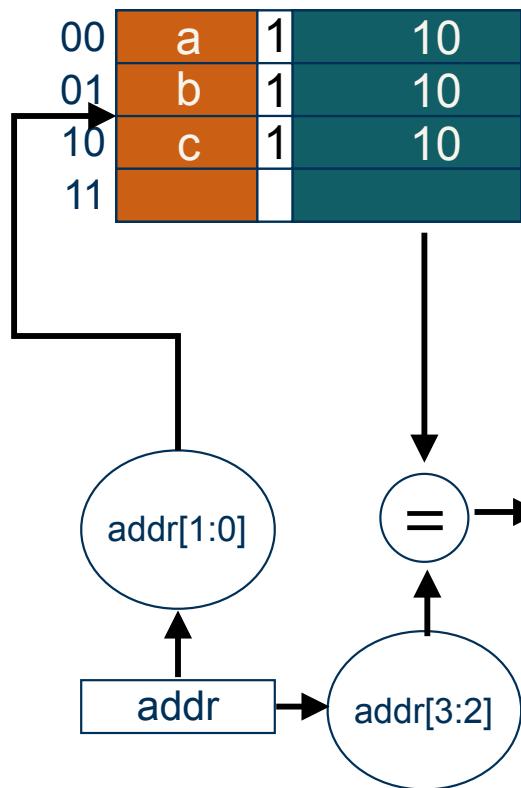


Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

Conflicts

Cache



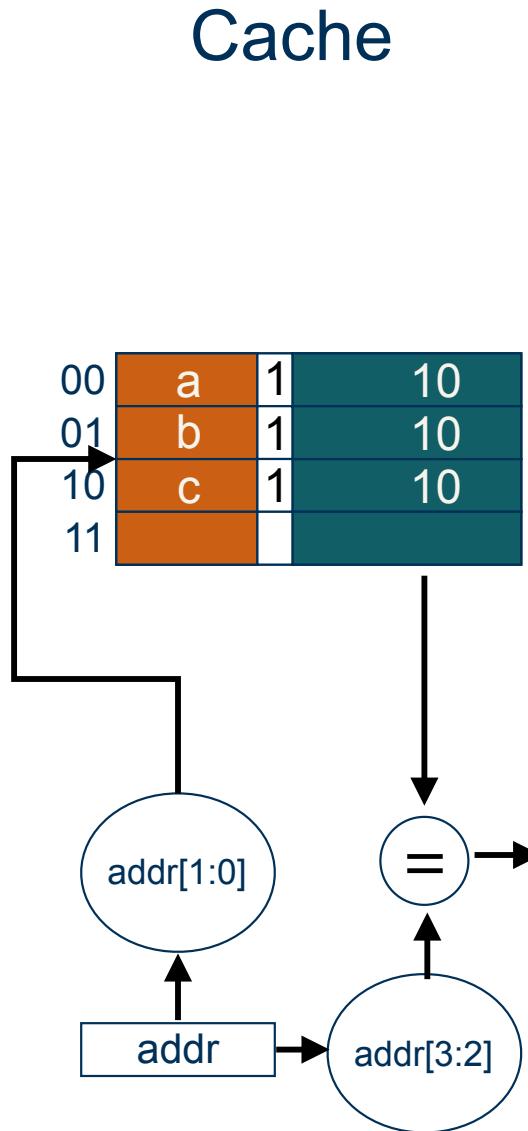
Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010
- Read 1101 (**kick out 1001**)

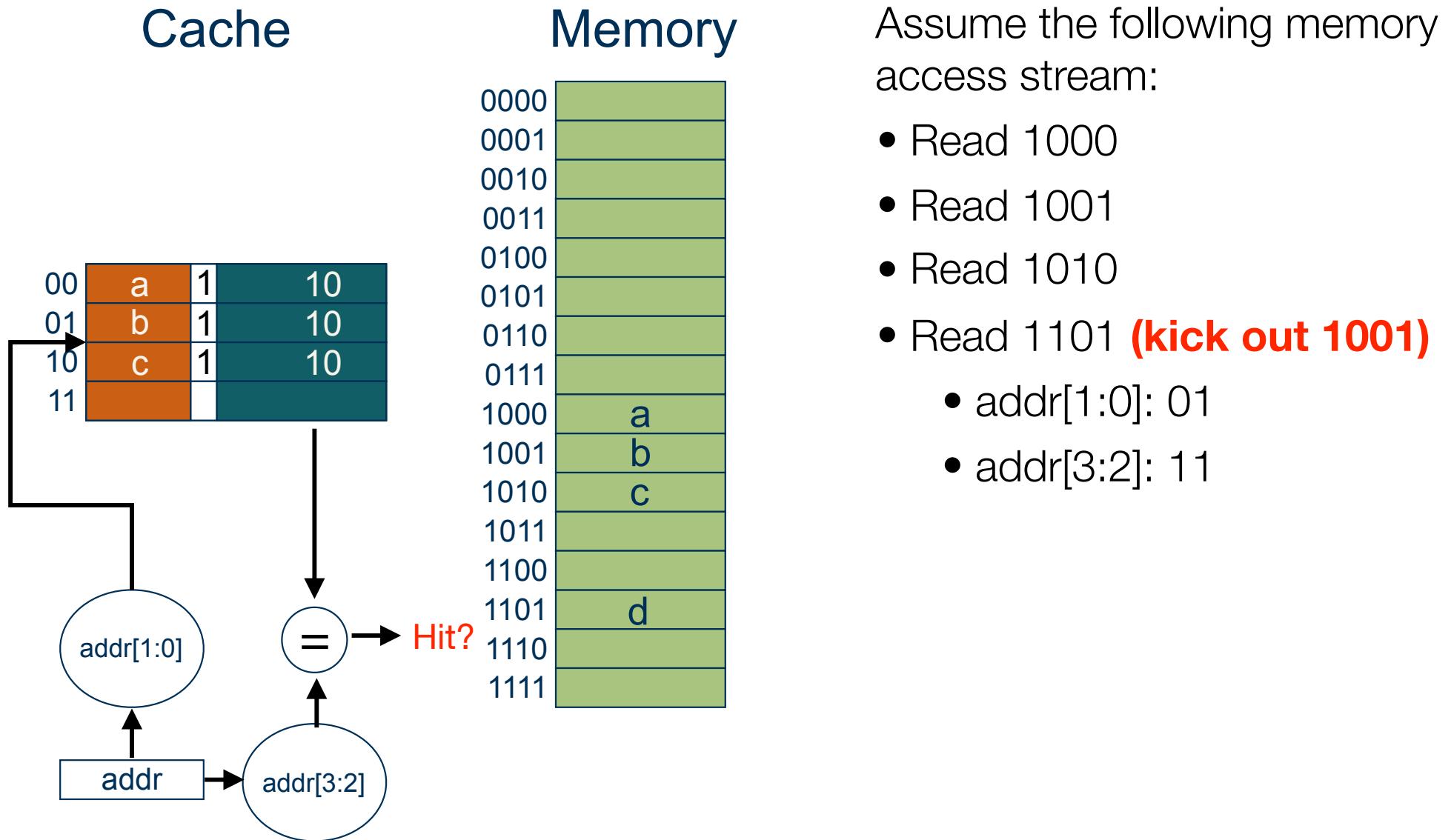
Conflicts



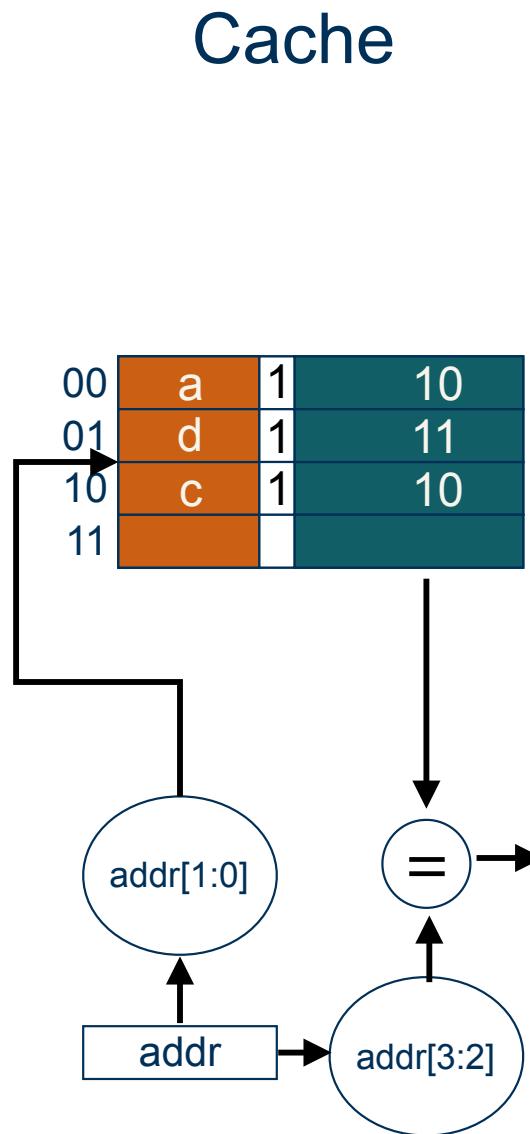
Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010
- Read 1101 (**kick out 1001**)
 - $addr[1:0]: 01$

Conflicts



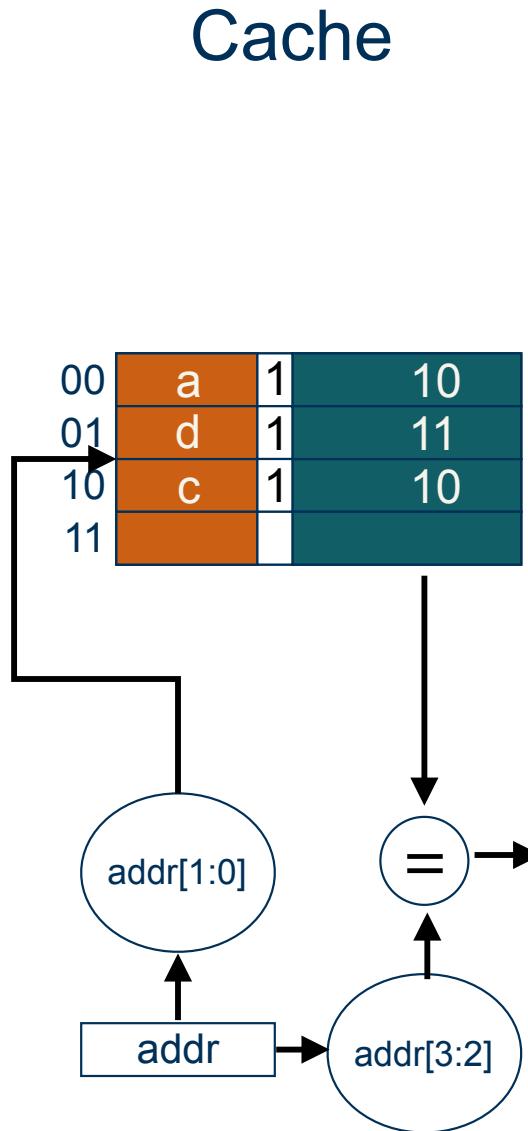
Conflicts



Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010
- Read 1101 (**kick out 1001**)
 - $addr[1:0]: 01$
 - $addr[3:2]: 11$

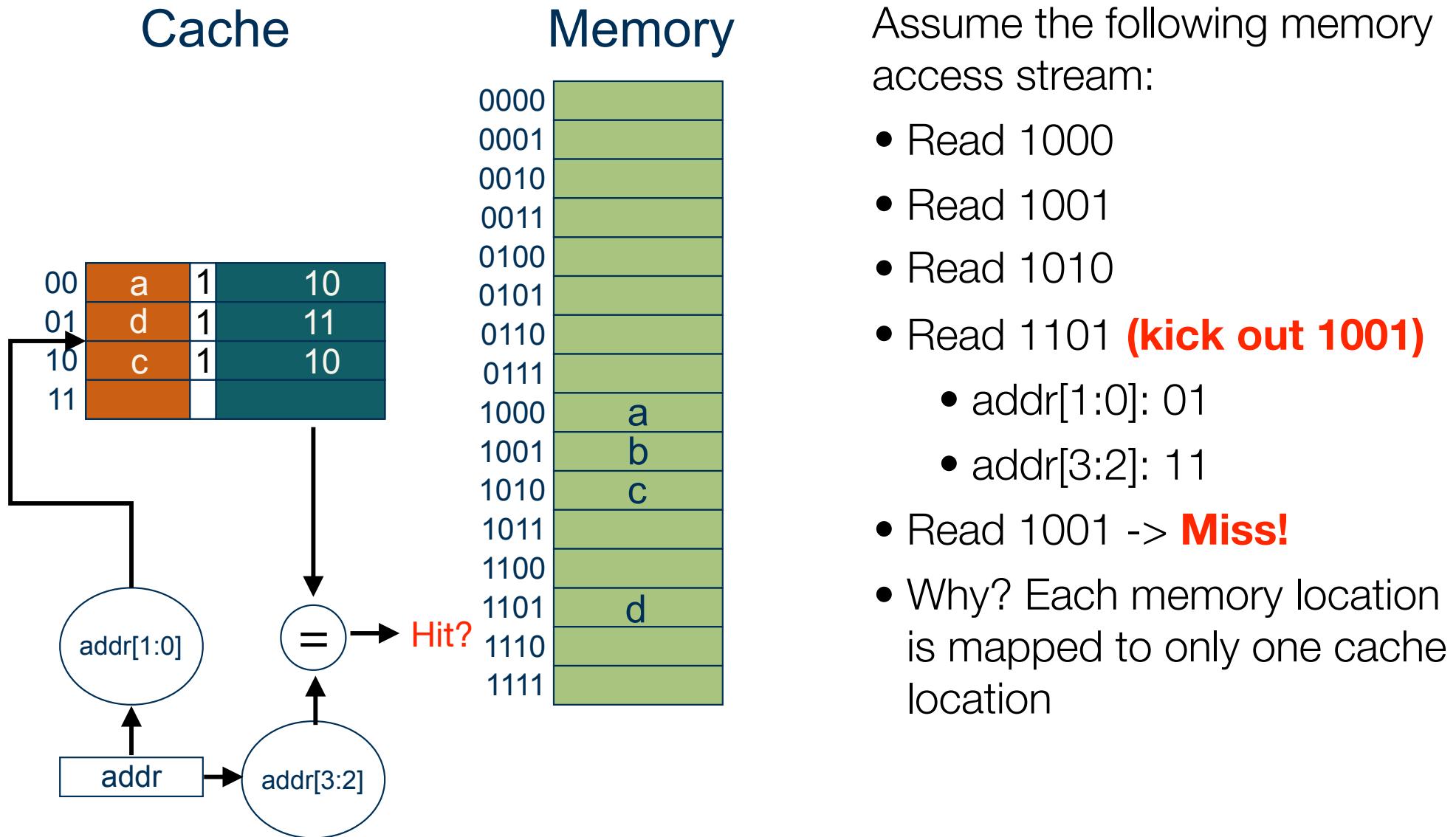
Conflicts



Assume the following memory access stream:

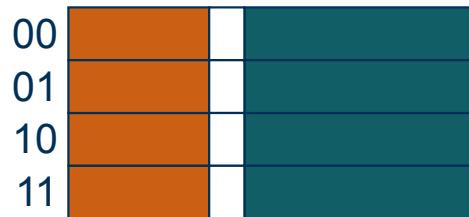
- Read 1000
- Read 1001
- Read 1010
- Read 1101 (**kick out 1001**)
 - $addr[1:0]: 01$
 - $addr[3:2]: 11$
- Read 1001 -> **Miss!**

Conflicts

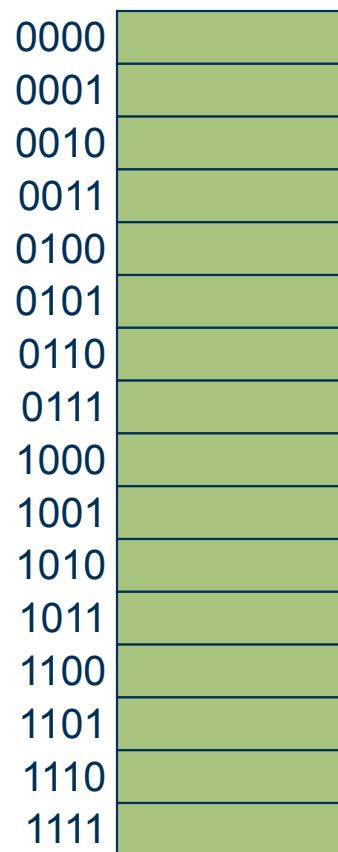


Sets

Cache



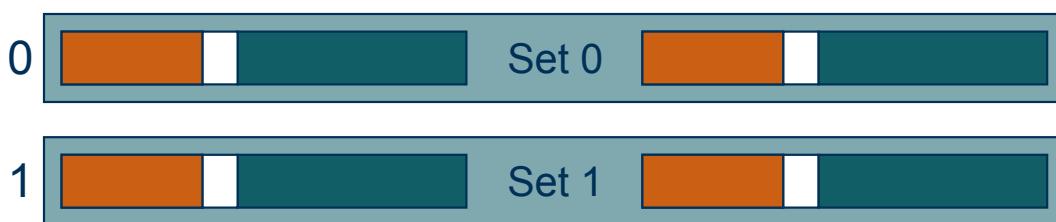
Memory



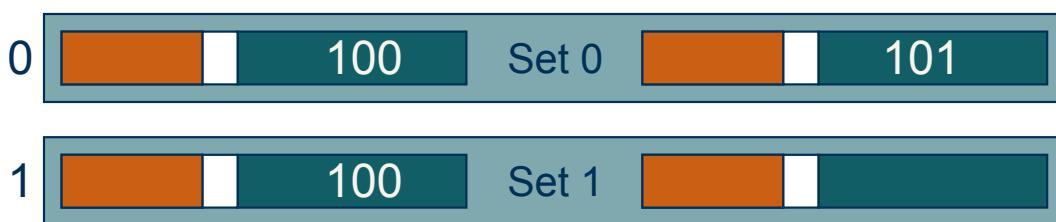
- Each cacheable memory location is mapped to **a set of cache locations**
- A set is one or more cache locations
- Set size is the number of locations in a set, also called **associativity**

2 Way Set Associative

- 2 sets, each set has two entries
- Only need one bit, $\text{addr}[0]$ to index into the cache now
- Correspondingly, the tag needs 3 bits: $\text{Addr}[3:1]$
- Either entry can store any address that gets mapped to that set



2 Way Set Associative



- 2 sets, each set has two entries
- Only need one bit, $\text{addr}[0]$ to index into the cache now
- Correspondingly, the tag needs 3 bits: $\text{Addr}[3:1]$
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
 - Read 1000
 - Read 1001
 - Read 1010
 - Read 1101 (**1001 can still stay**)
 - **Read 1001 -> Hit!**

4 Way Set Associative

- One single set that contains all the cache locations
- Also called Fully-Associative Cache
- Every entry can store any cache-line that maps to that set



4 Way Set Associative

- One single set that contains all the cache locations
- Also called Fully-Associative Cache
- Every entry can store any cache-line that maps to that set

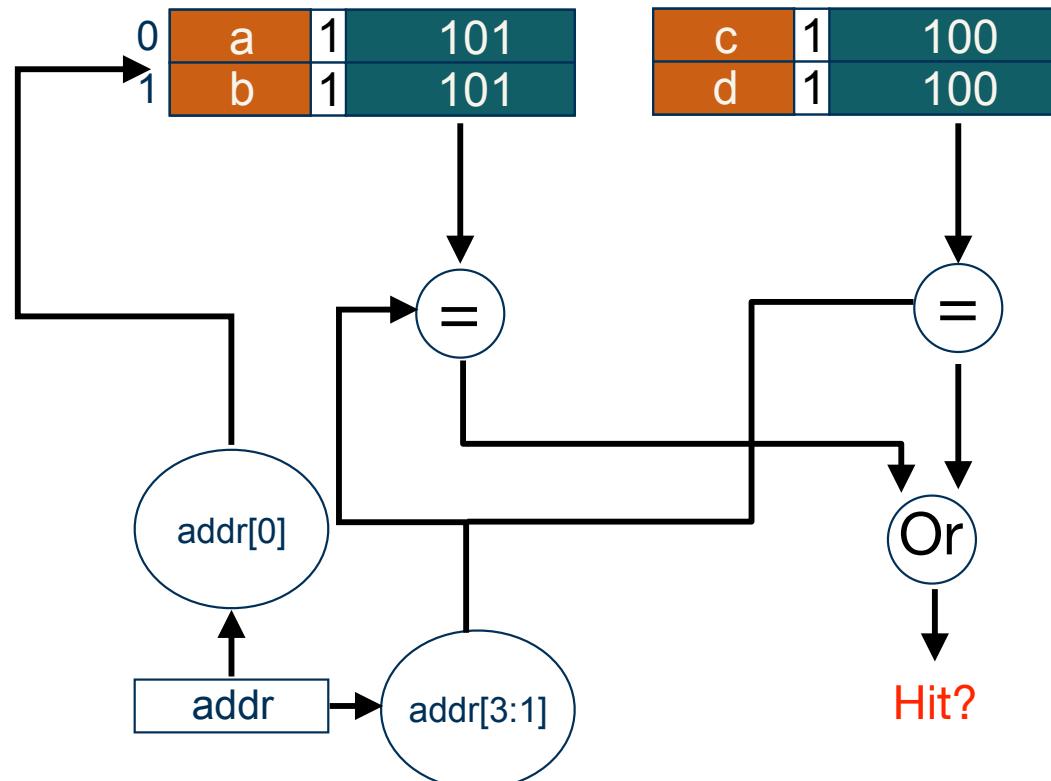
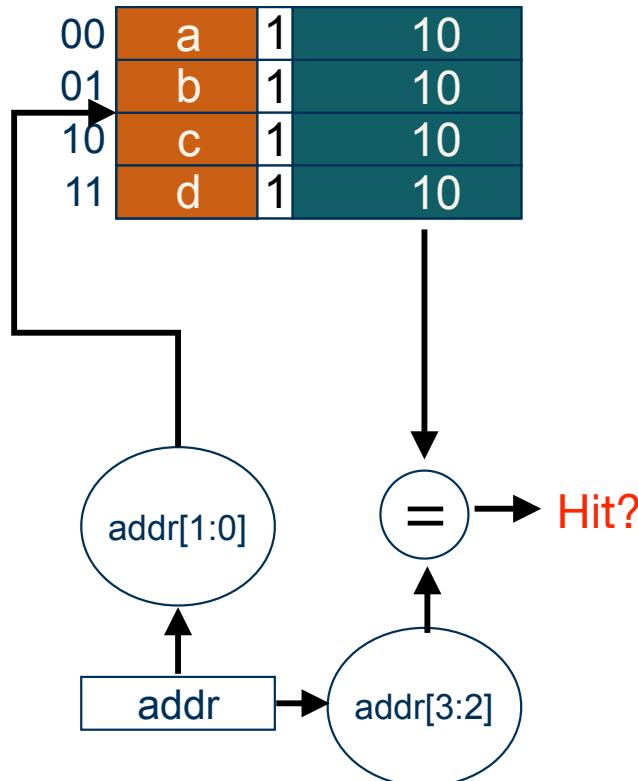


Assuming the same access stream

- Read 1000
- Read 1001
- Read 1010
- Read 1101
- Read 1001 -> **Hit!**

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. Why?



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 is independent of the the split between index and tag



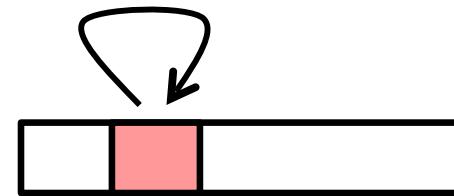
Virtual or physical memory address?

Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

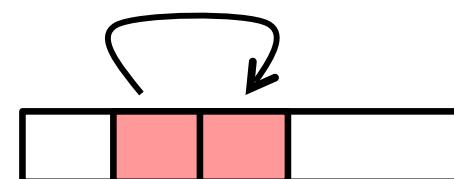
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future

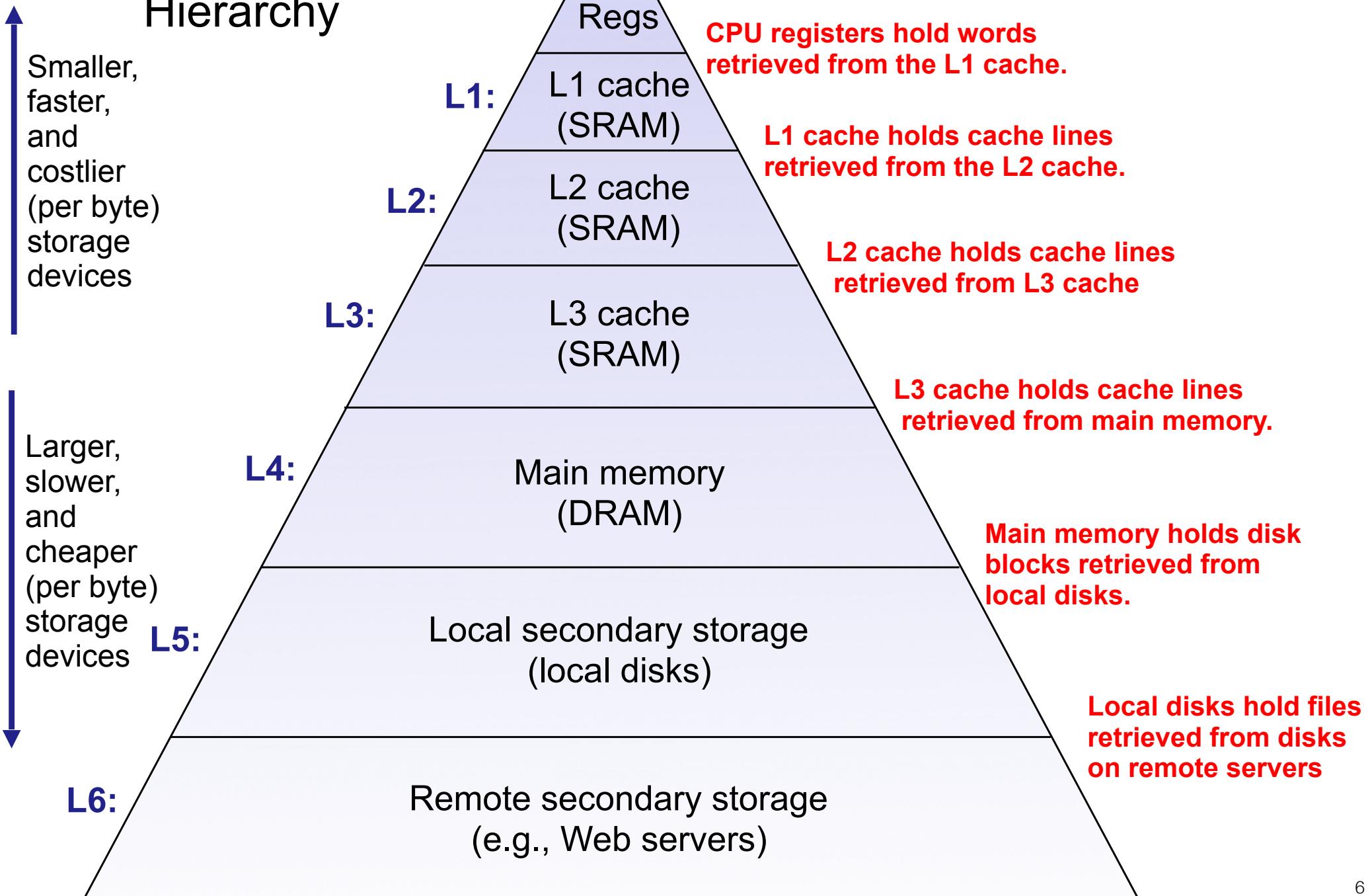


- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time



Example Memory Hierarchy

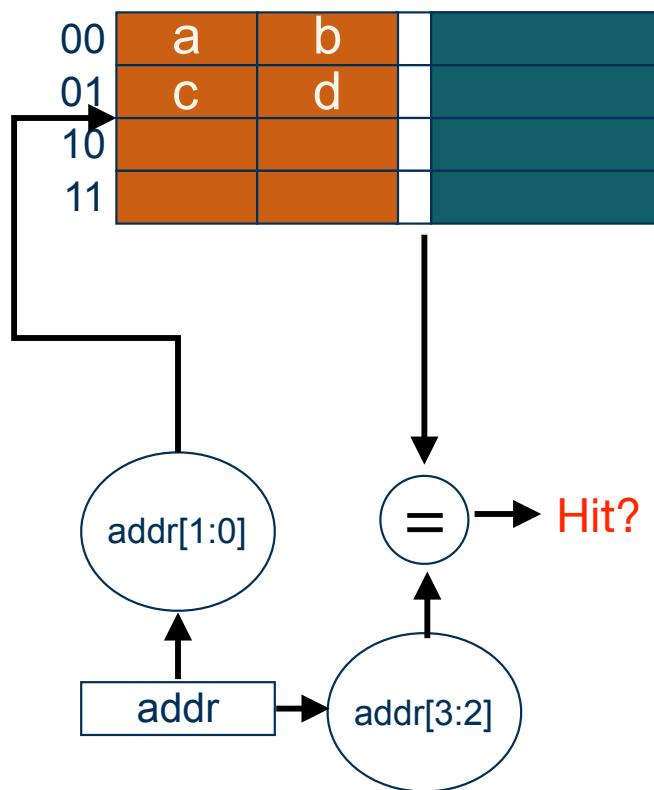


Locality again

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

Cache-Line Size of 2

Cache



Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

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

What's the
hit ratio?

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

- 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
 - 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
 - + 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?
- 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?
 - 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

- 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



LRU index (2 bits)

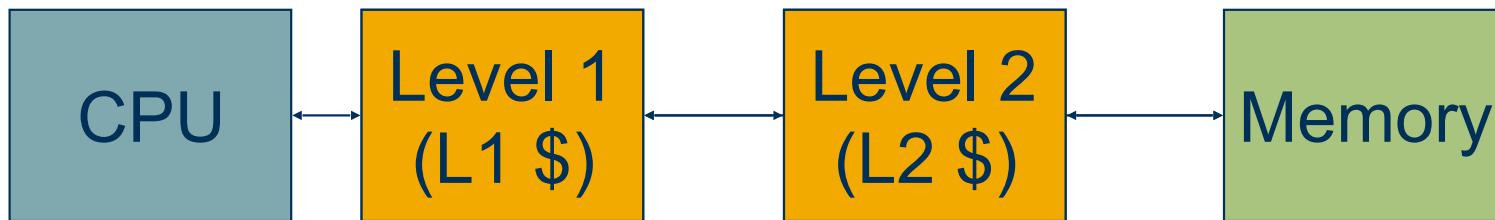


What to update now???

Address stream:

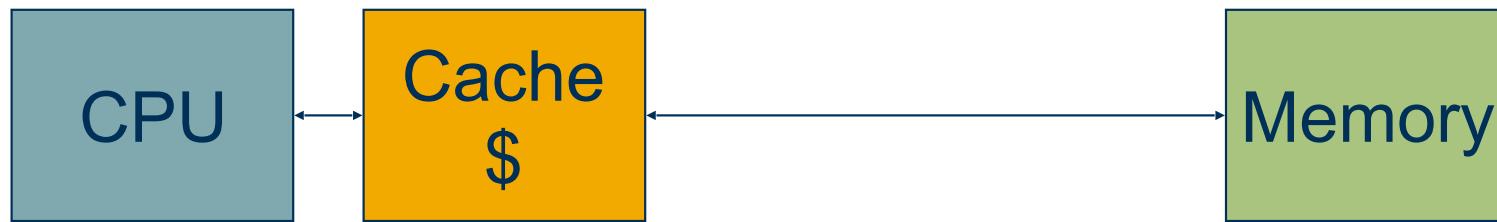
- Hit on 0
- Hit on 2
- Hit on 3
- Miss, evict 1

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

Summary



- Assumptions:
 - memory access (~100ns) >> cache access (~1ns)
 - cache smaller, faster, more expensive than memory
 - Programs exhibit ***temporal locality***
 - if an item is referenced, it will tend to be referenced again soon
 - Programs exhibit ***spatial locality***
 - If an item is referenced, the next item in memory is likely to be accessed soon