CSC 252: Computer Organization
Spring 2020: Lecture 16

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- By default 252/452 will be pass/fail
- 4/10 is the deadline to opt-out and request a letter grade
- Mid-term solution has been posted
- Mid-term grades will be posted this weekend
- Lectures will be recorded and posted online
- Office hours will be held through Zoom; links on the website

The Problem

- Bigger is slower
  - Flip-flops/Small SRAM, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec

- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte

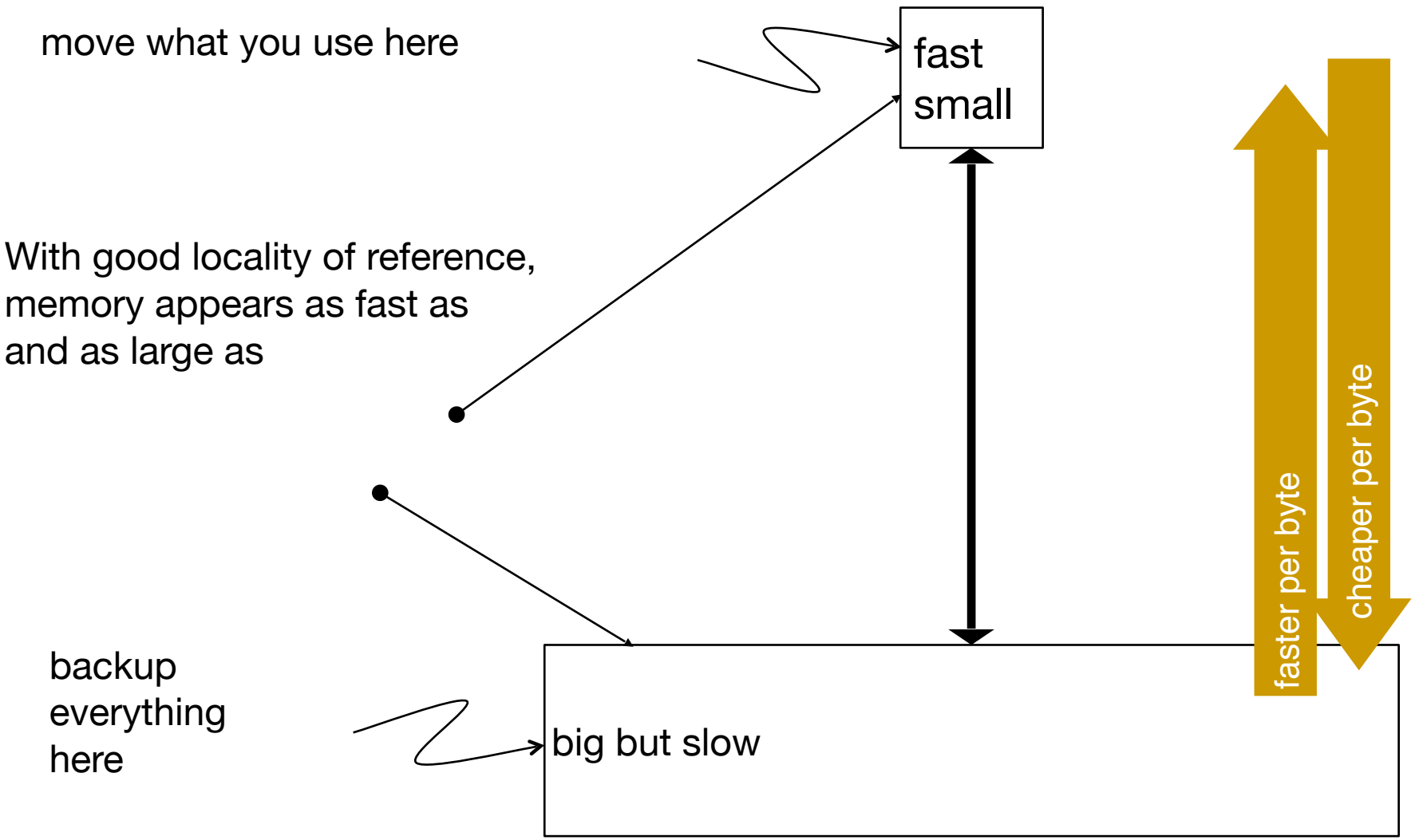
- Other technologies have their place as well
  - PC-RAM, MRAM, RRAM

We want both fast and large Memory

- But we cannot achieve both with a single level of memory

- Idea: Memory Hierarchy
  - Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor)
  - ensure most of the data the processor needs in the near future is kept in the fast(er) level(s)

Memory Hierarchy

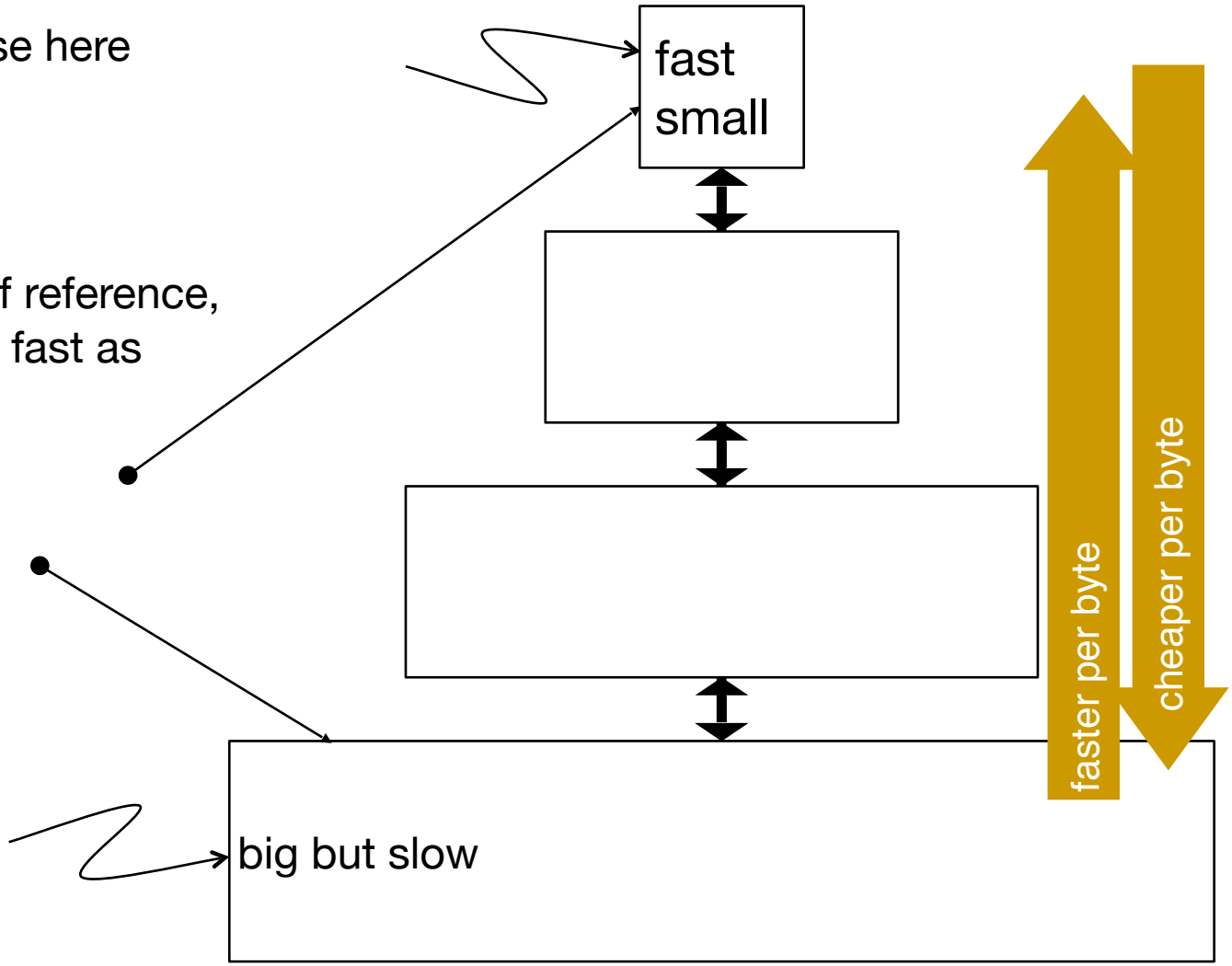


Memory Hierarchy

move what you use here

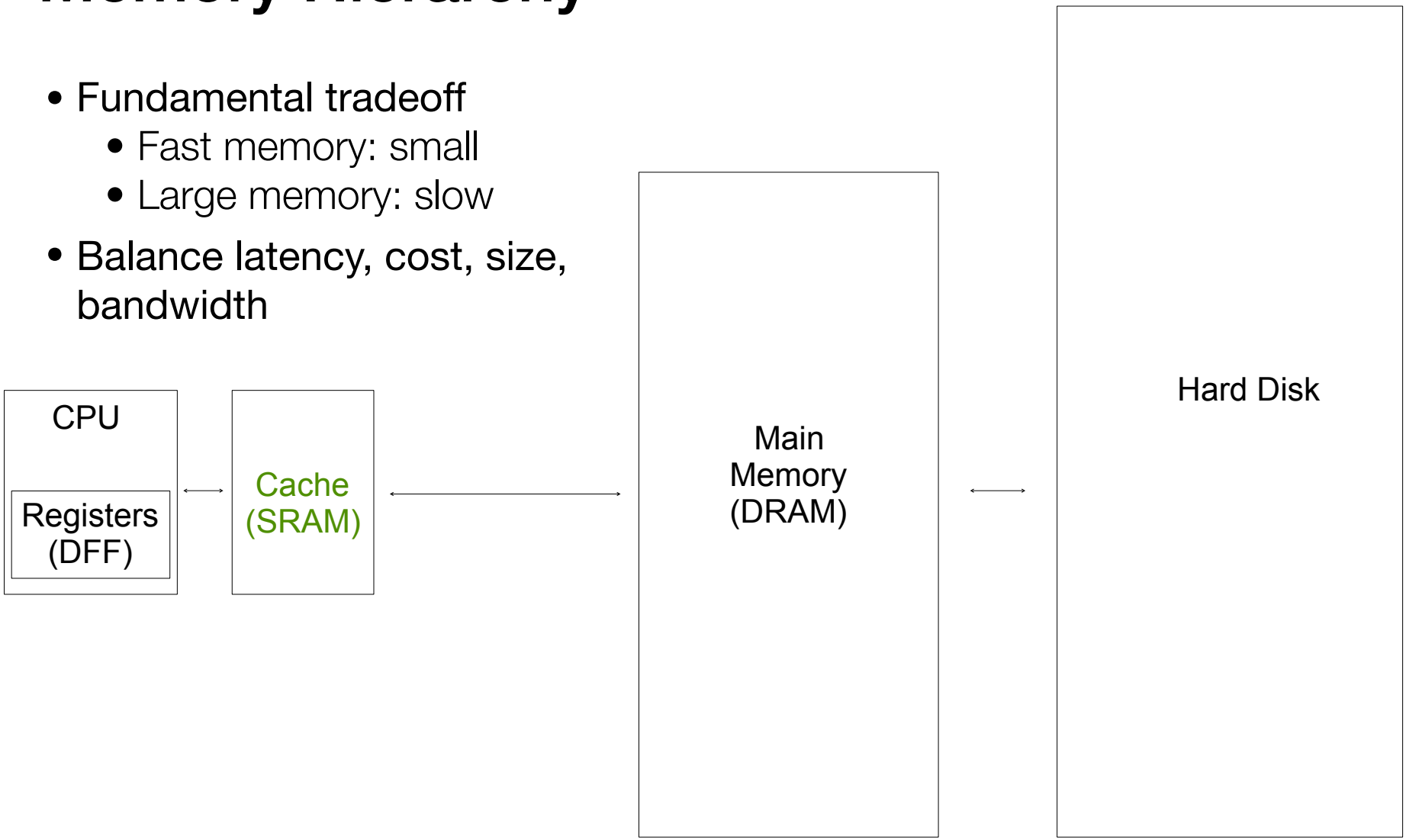
With good locality of reference, memory appears as fast as and as large as

backup everything here



Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Balance latency, cost, size, bandwidth

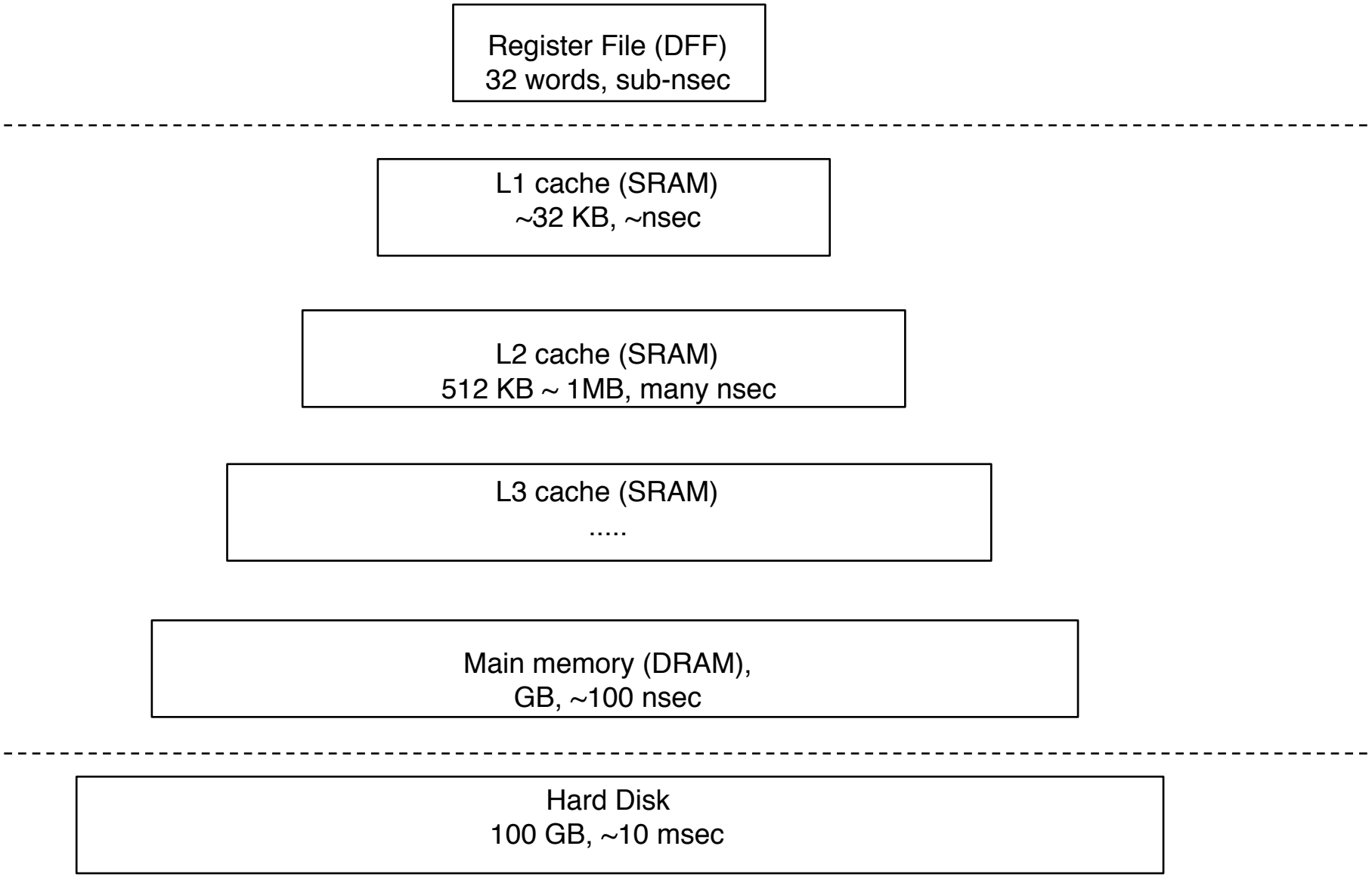


The Bookshelf Analogy

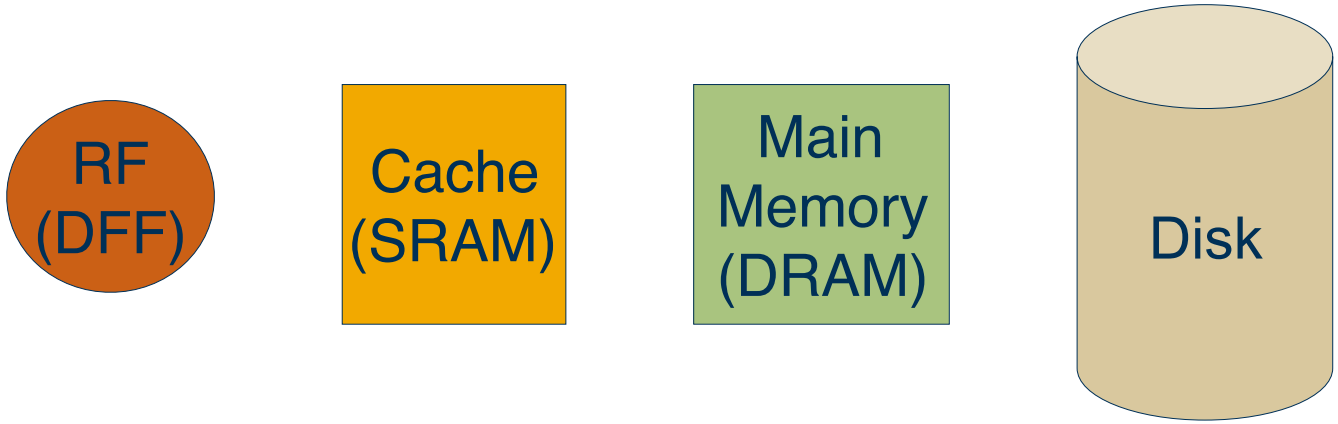
- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Org. books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time

A Modern Memory Hierarchy



How Things Have Progressed

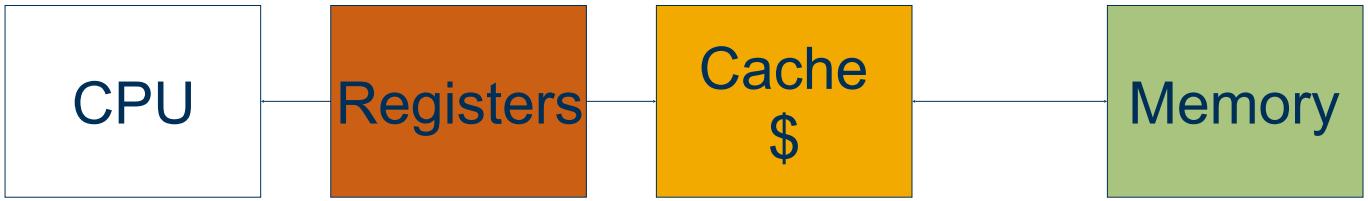


1995 low-mid range Hennessy & Patterson, Computer Arch., 1996	200B 5ns	64KB 10ns	32MB 100ns	2GB 5ms
2009 low-mid range www.dell.com, $449 including 17" LCD flat panel	~200B 0.33ns	8MB 0.33ns	4GB <100ns	750GB 4ms
2015 mid range	~200B 0.33ns	8MB 0.33ns	16GB <100ns	**256GB** **10us**

How to Make Effective Use of the Hierarchy

- Fundamental question: how do we know what data to put in the fast and small memory?
- Answer: ensure most of the data the processor needs in the near future is kept in the fast(er) level(s)
- How do we know what data will be needed in the future?
  - Do we know before the program runs?
    - If so, programmers or compiler can place the right data at the right place
  - Do we know only after the program runs?
    - If so, only the hardware can effectively place the data

How to Make Effective Use of the Hierarchy



- Modern computers provide both ways
- Register file: programmers explicitly move data from the main memory (slow but big DRAM) to registers (small, very fast)
  - `movq (%rdi), %rdx`
- Cache, on the other hand, is automatically managed by hardware
  - Sits between registers and main memory, “invisible” to programmers
  - The hardware automatically figures out what data will be used in the near future, and place in the cache.
  - How does the hardware know that??

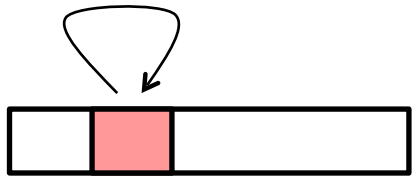
Locality

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

Locality

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

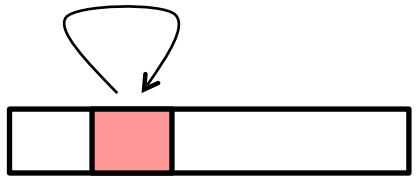
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



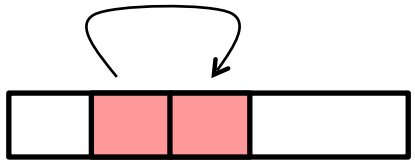
Locality

- 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

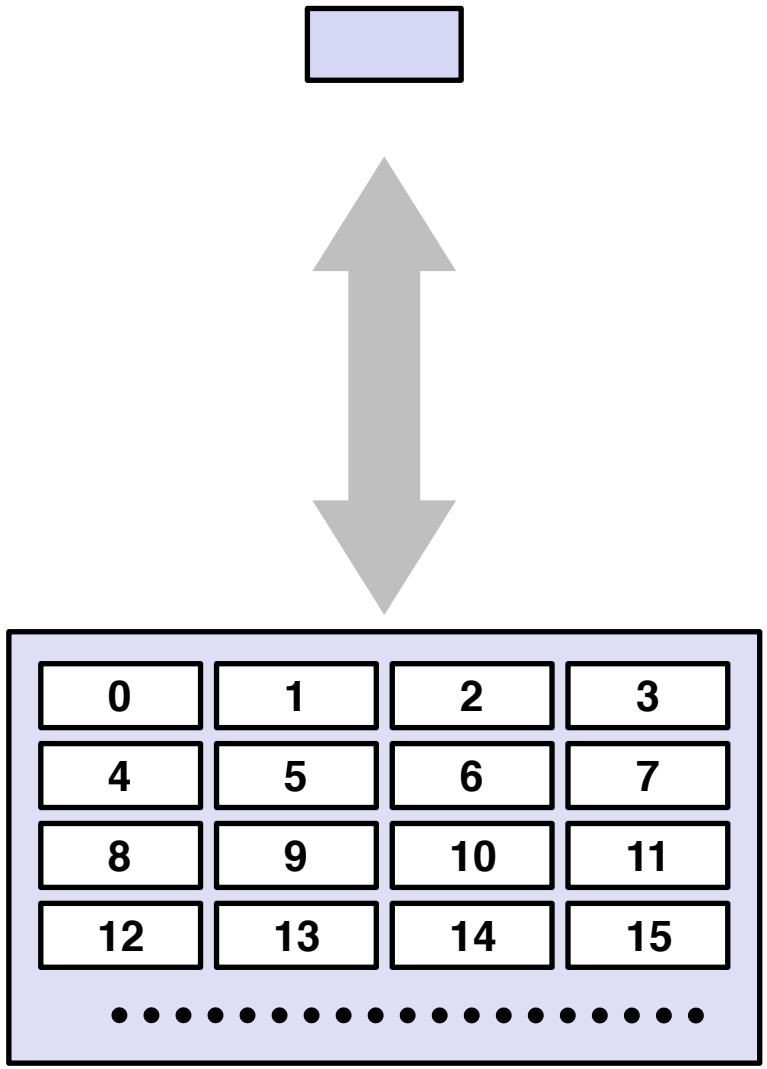
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- 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.

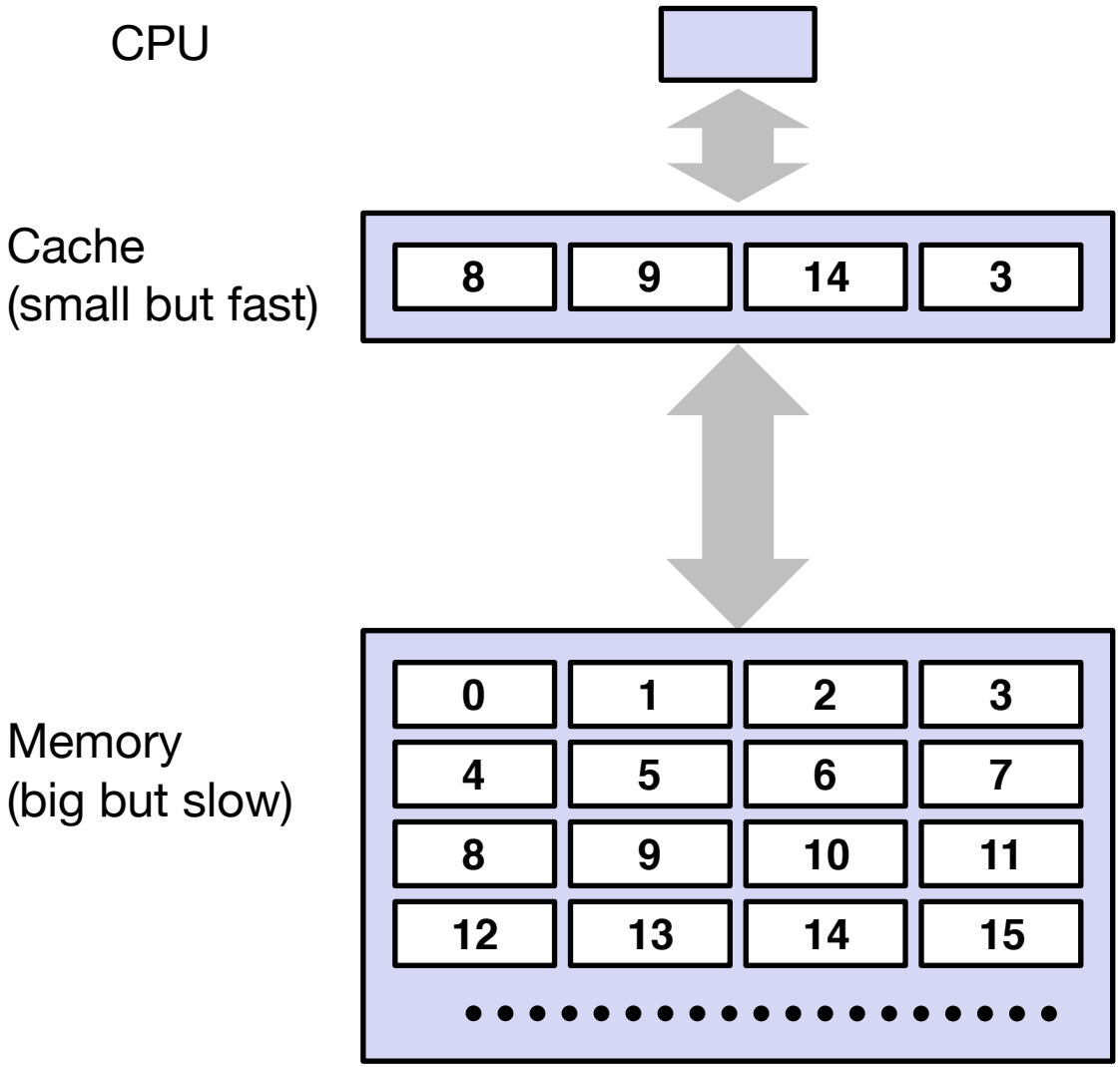
Cache Illustrations

CPU

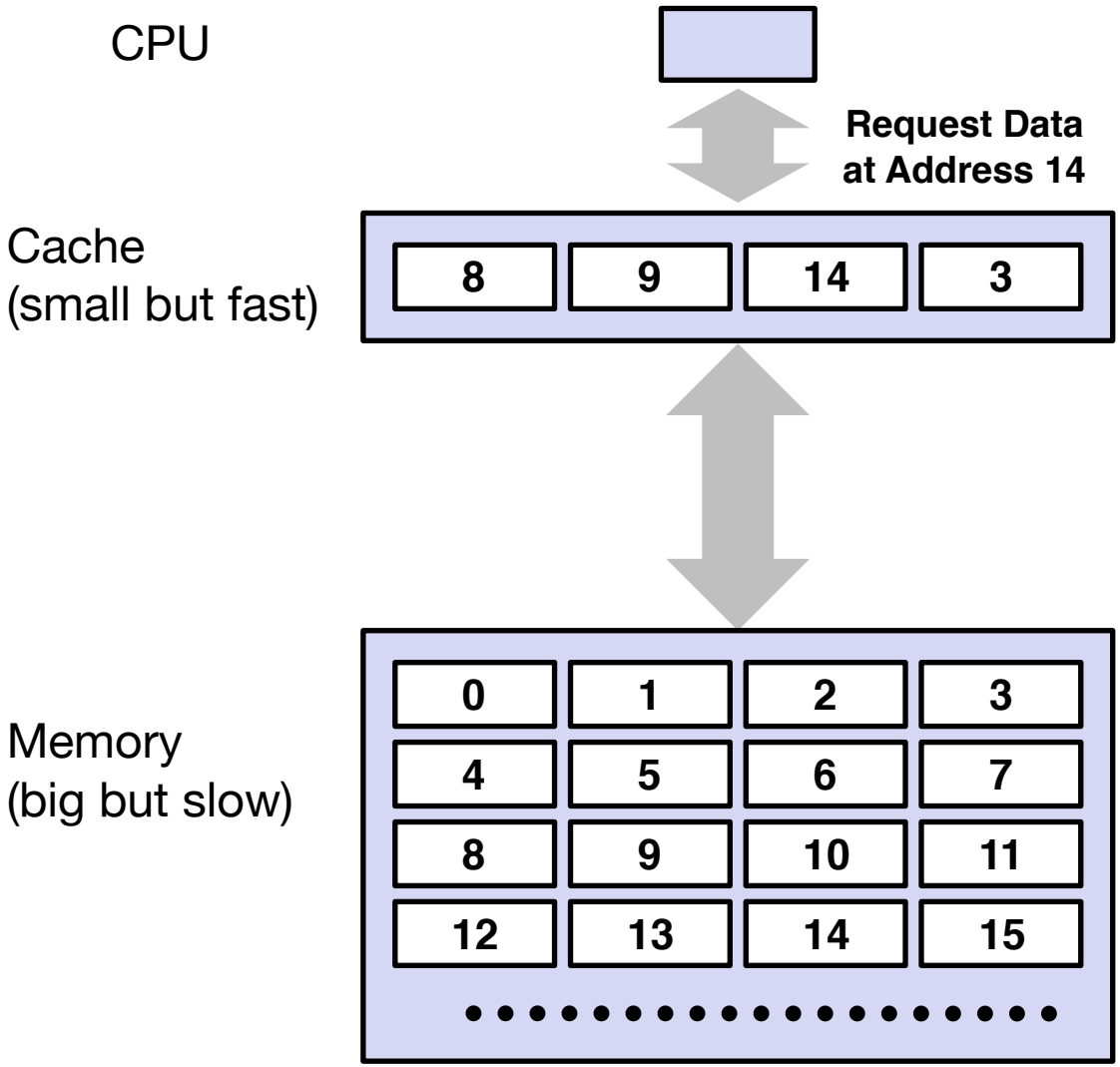


Memory
(big but slow)

Cache Illustrations

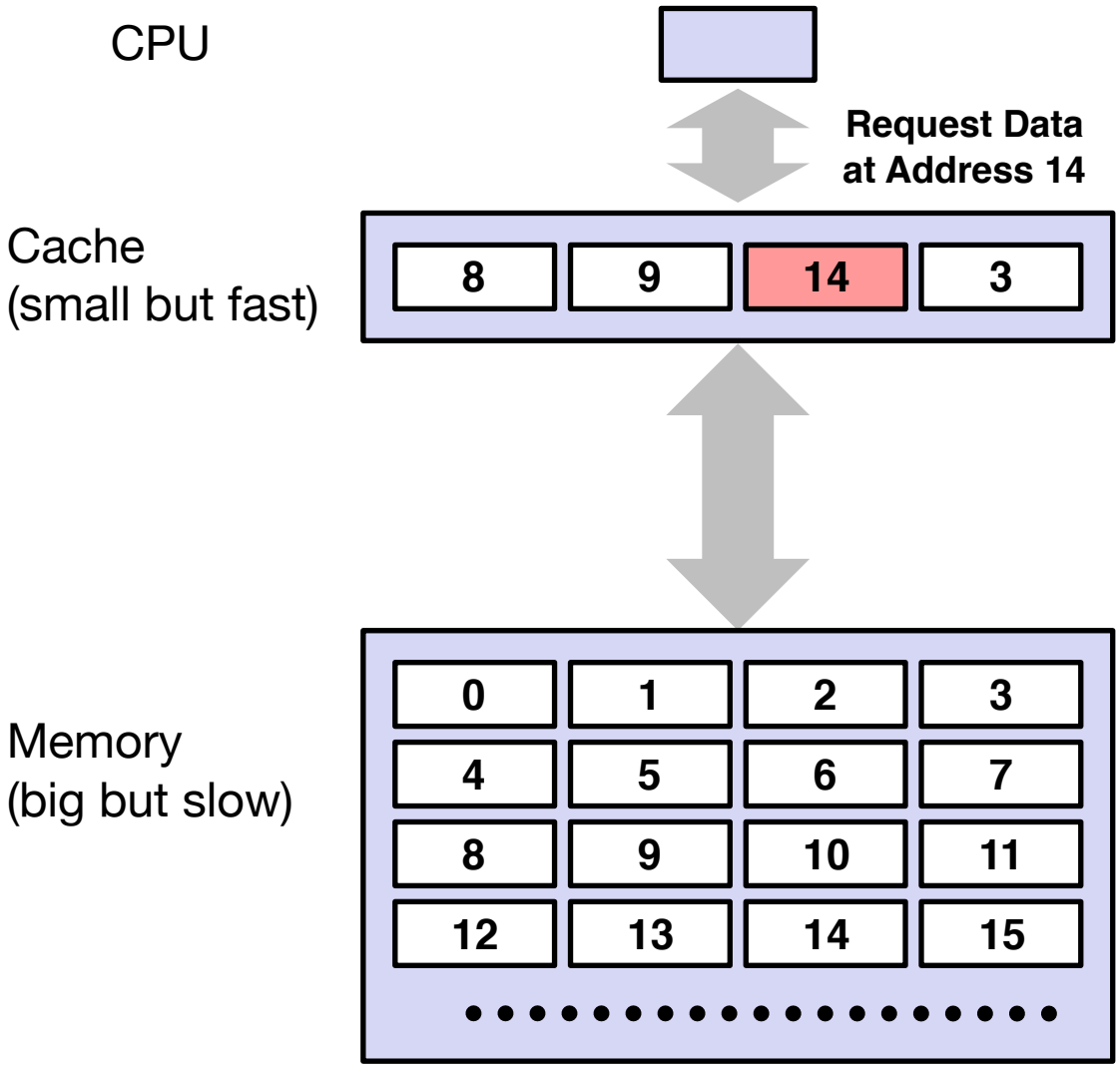


Cache Illustrations



Data in address b is needed

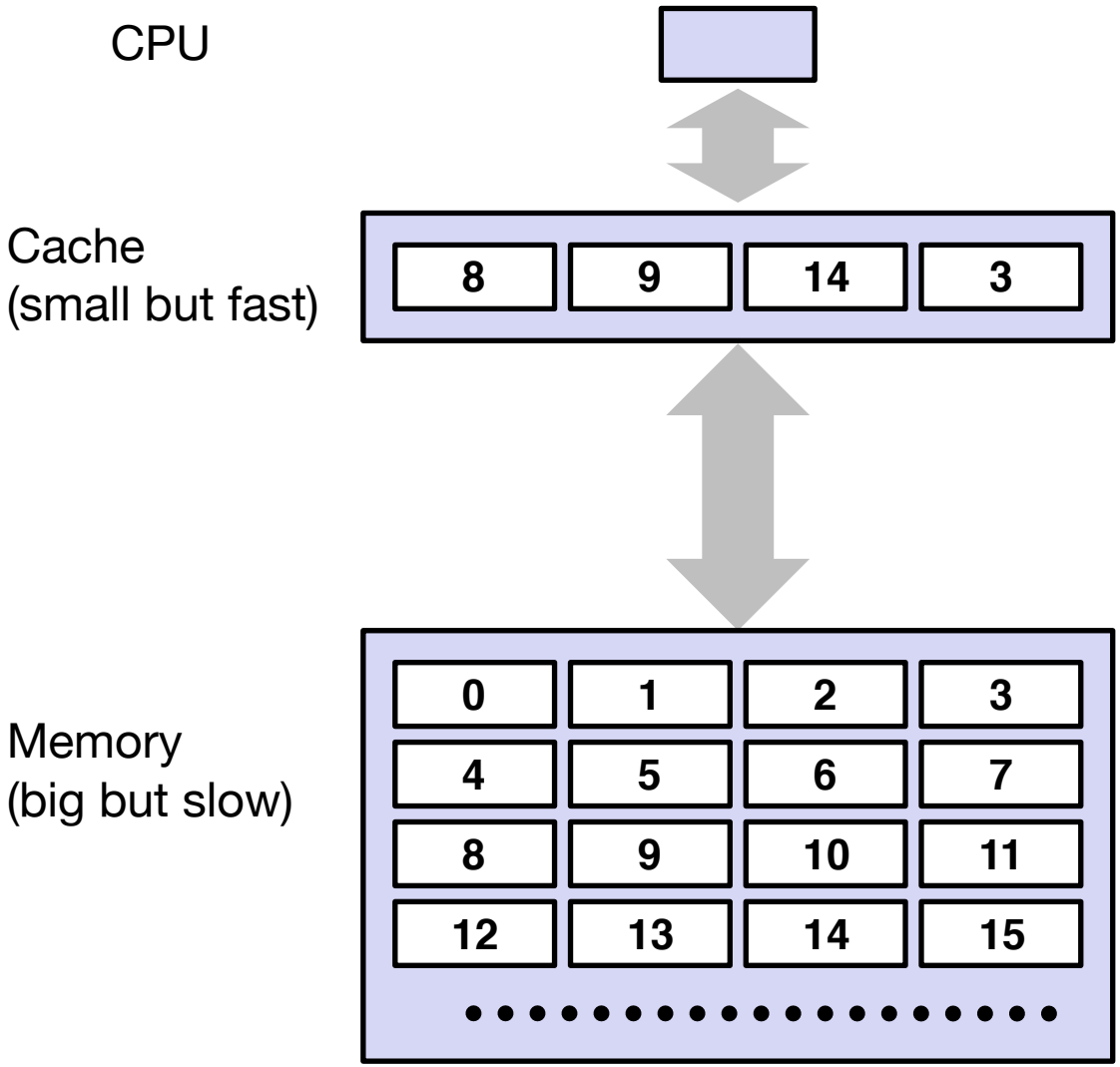
Cache Illustrations



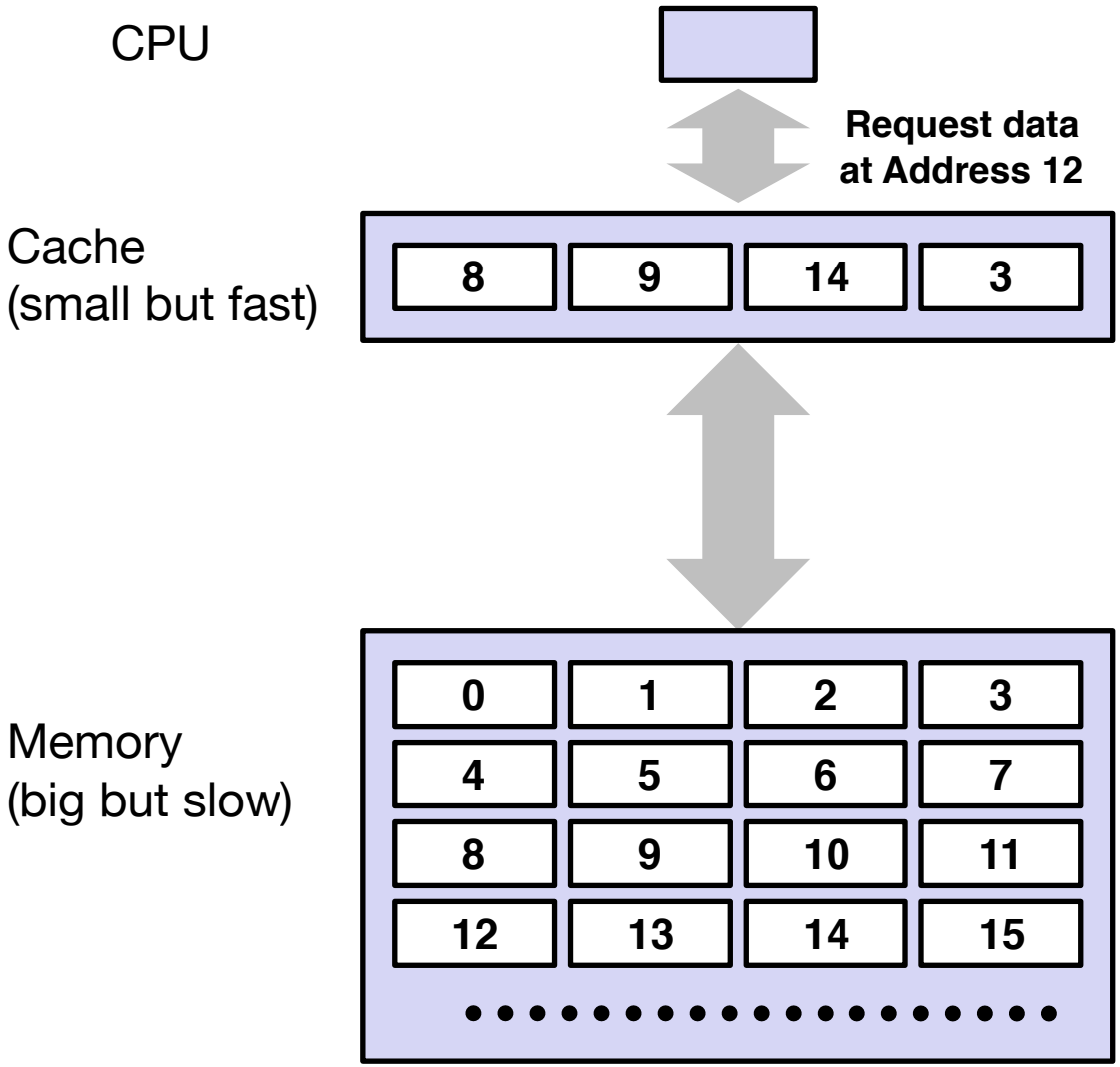
Data in address b is needed

Address b is in cache: Hit!

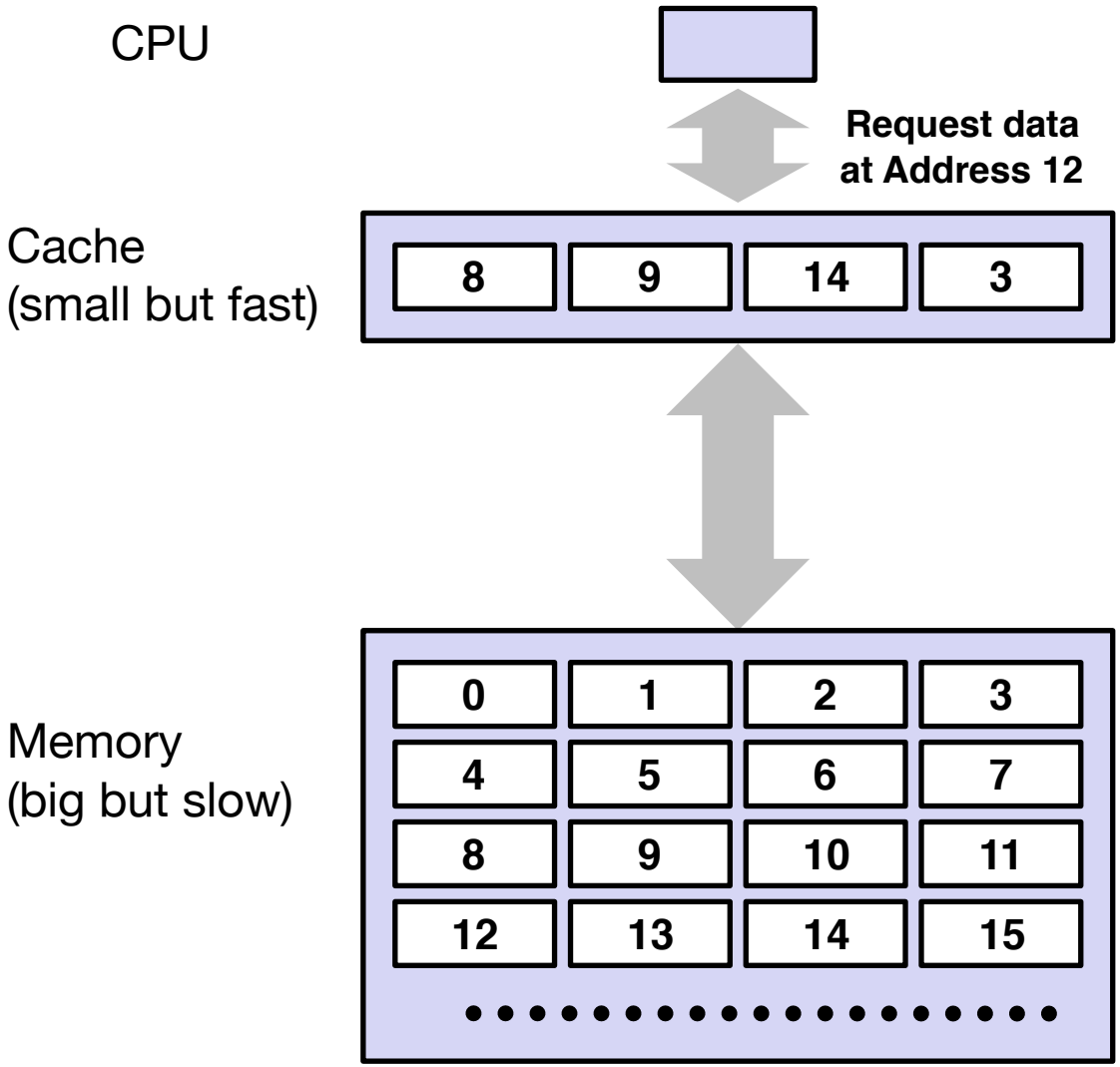
Cache Illustrations



Cache Illustrations



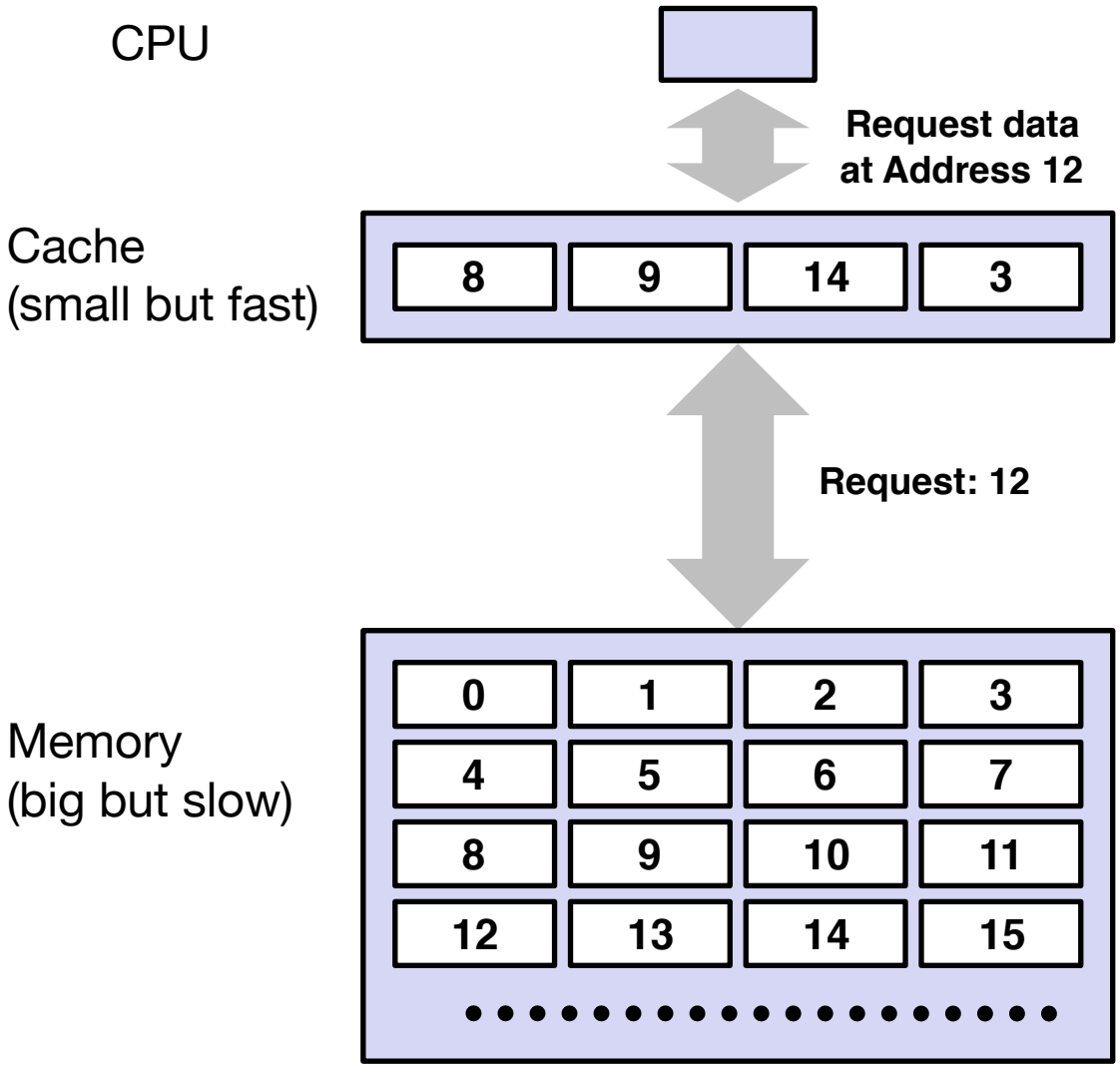
Cache Illustrations



Data in address b is needed

Address b is not in cache: Miss!

Cache Illustrations

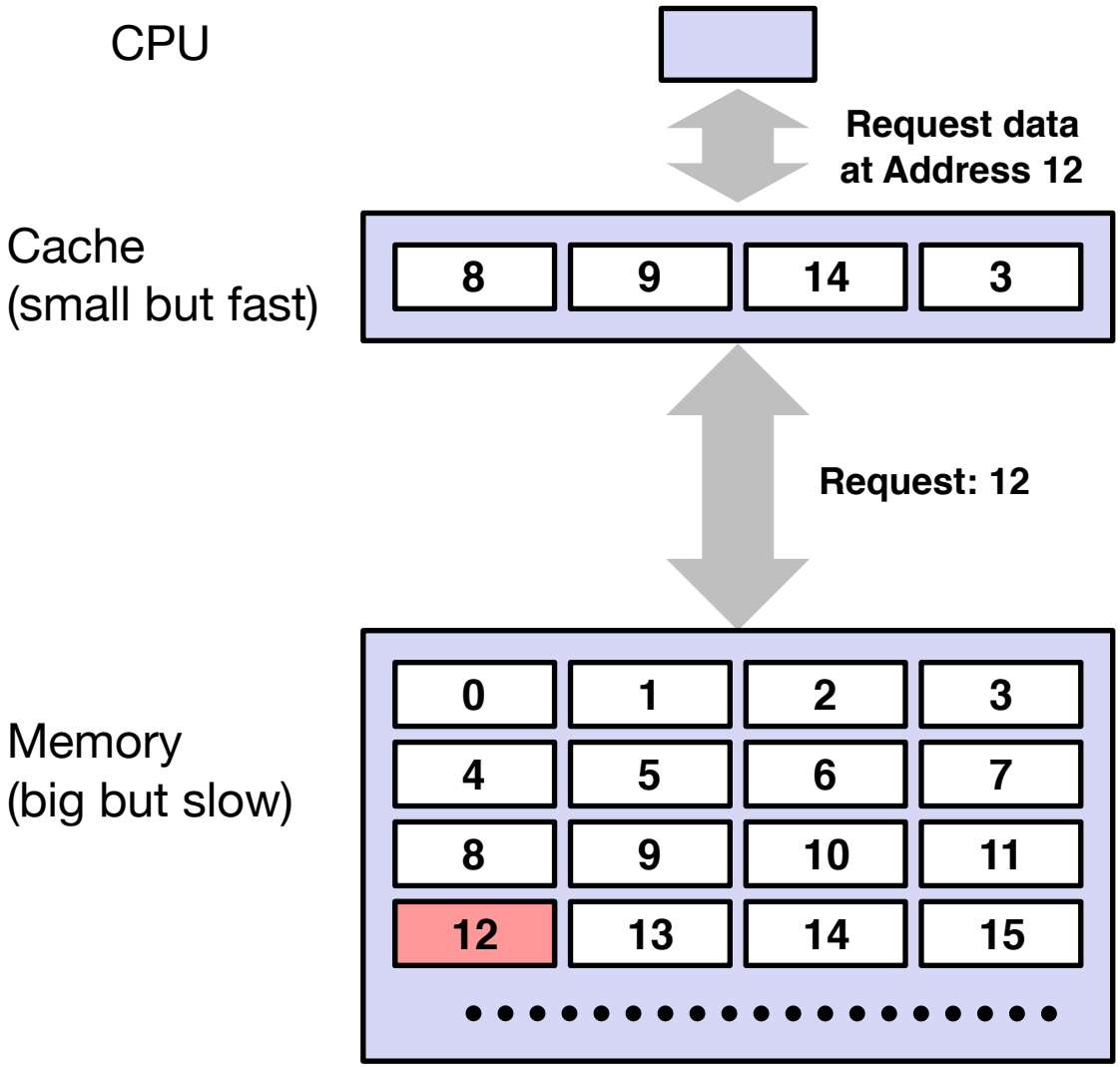


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations

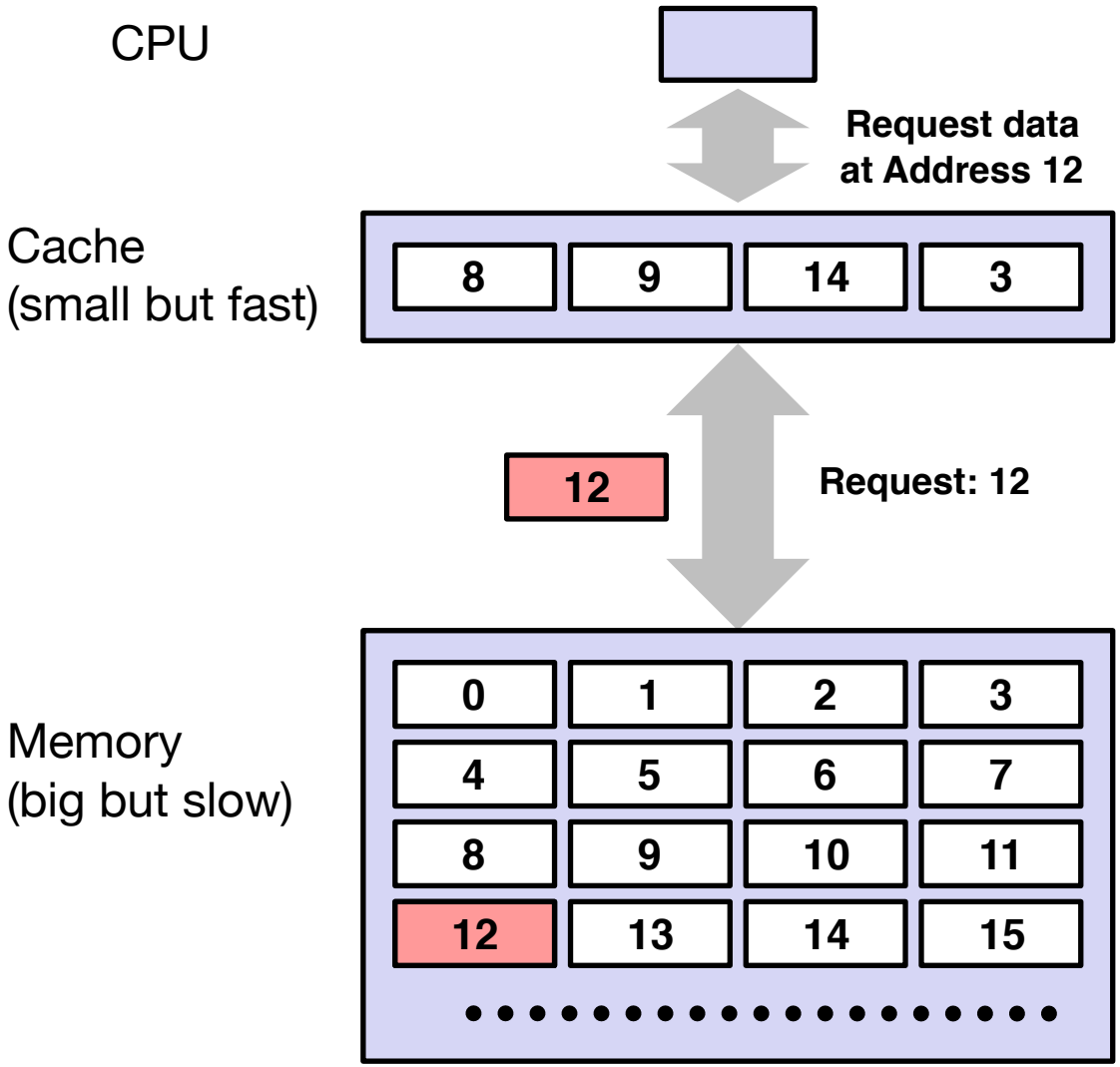


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations

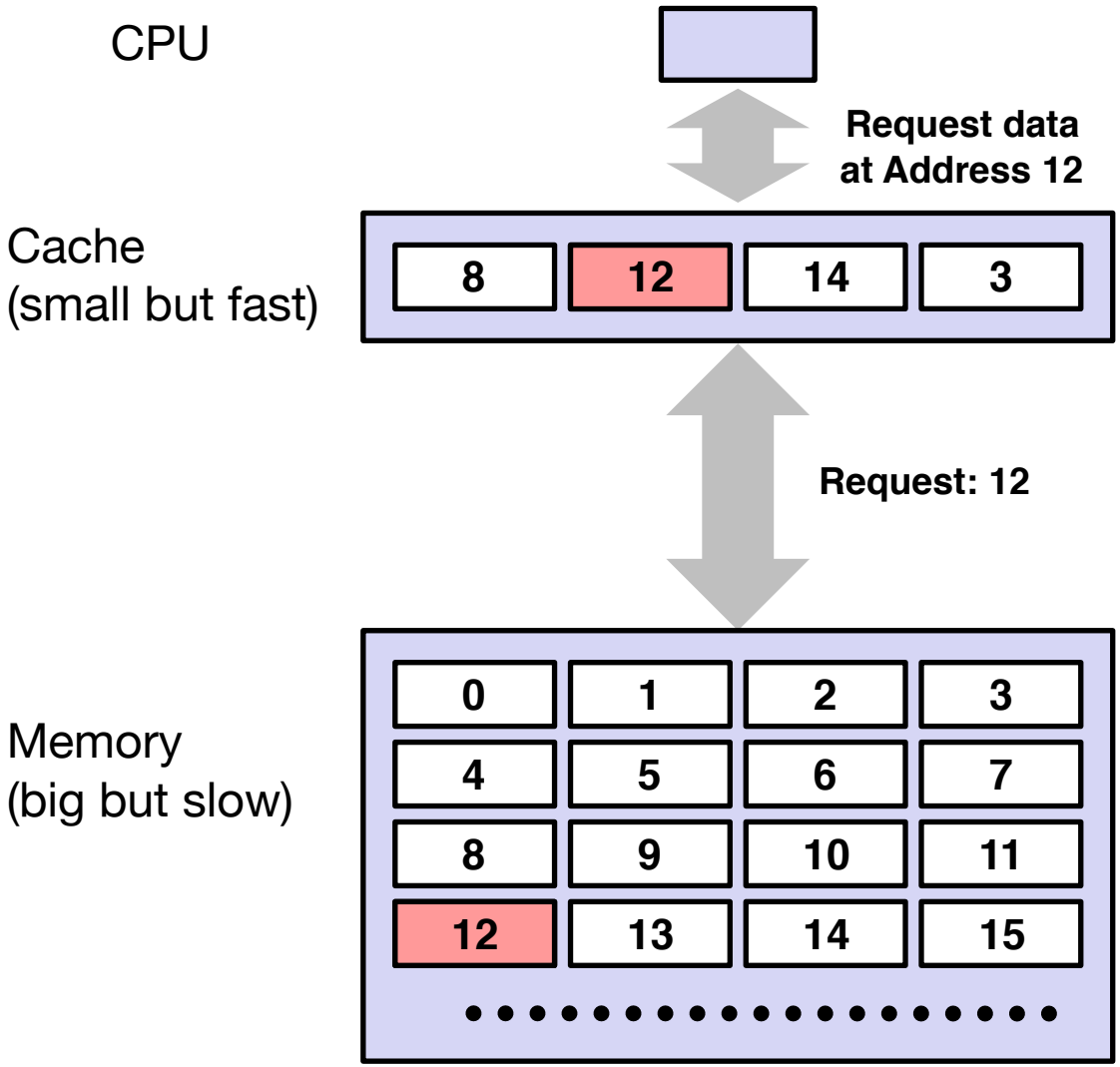


Data in address b is needed

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

Cache Hit Rate

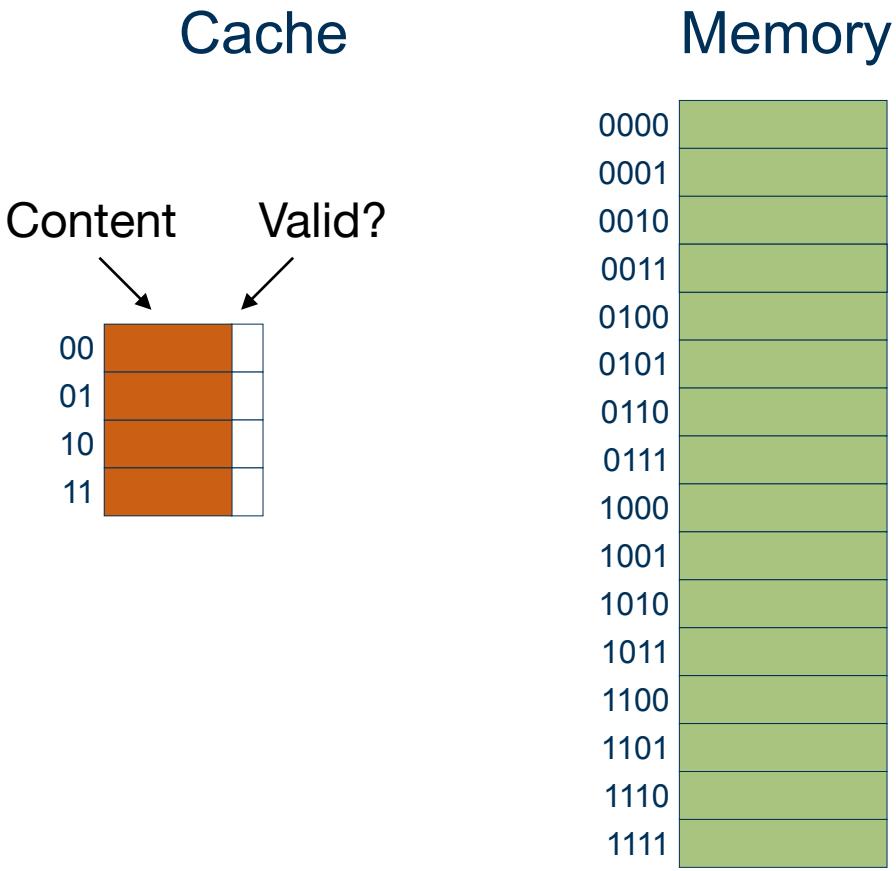
- Cache hit is when you find the data in the cache
- Hit rate indicates the effectiveness of the cache

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

Two Fundamental Issues in Cache Management

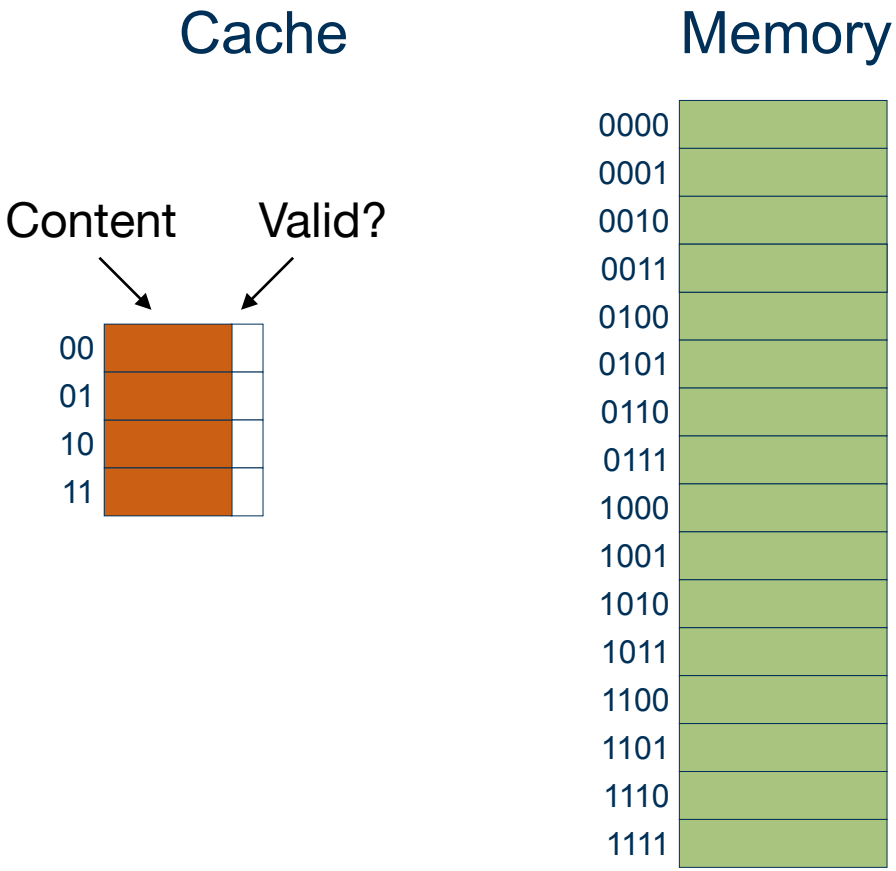
- Finding the data in the cache
  - Given an address, how do we decide whether it's in the cacher not?
- Kicking data out of the cache
  - Cache is small than memory, so when there's no place left in the cache, we need to kick something out before we can put new data into it, but who to kick out?

A Simple Cache



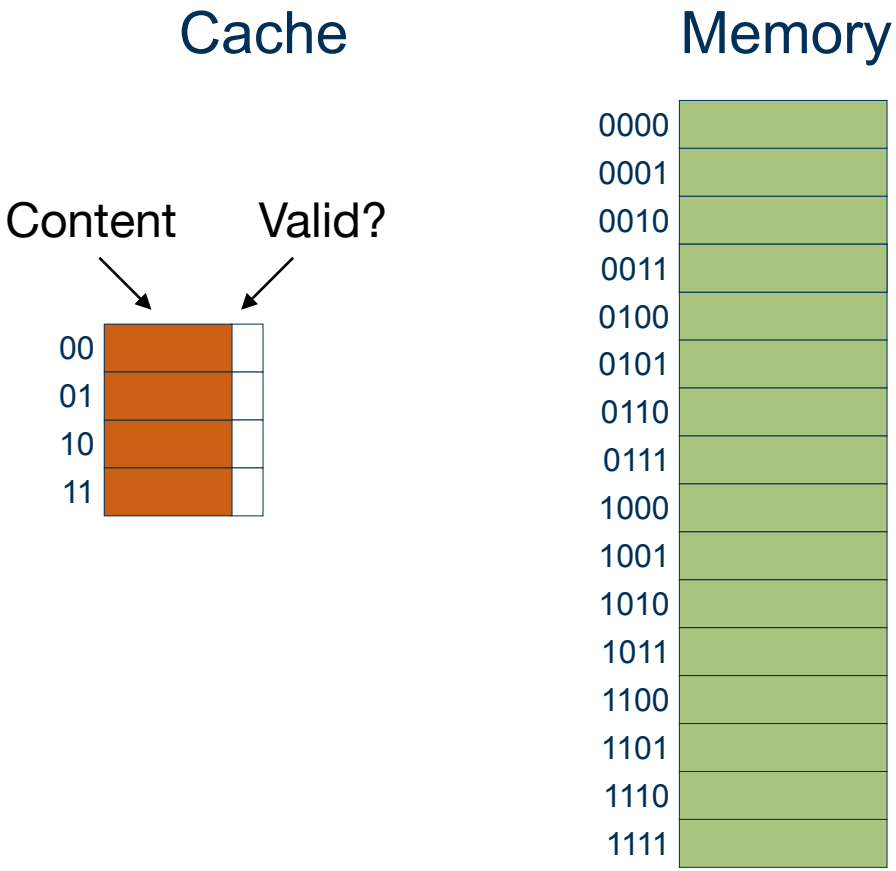
- 16 memory locations
- 4 cache locations

A Simple Cache



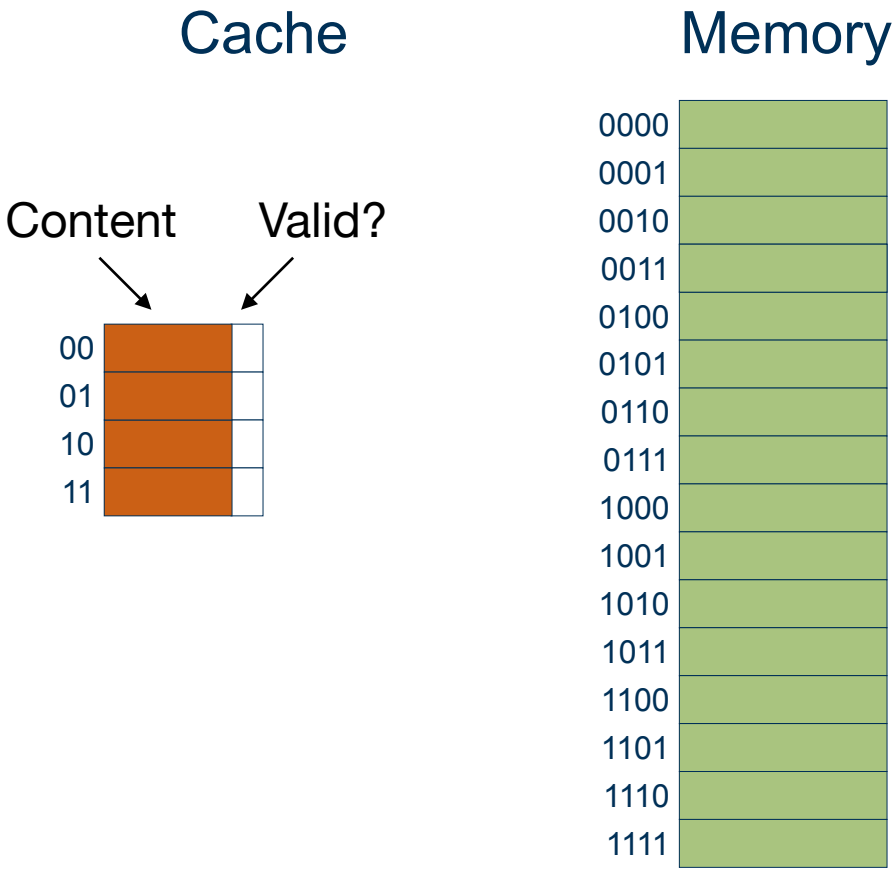
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**

A Simple Cache



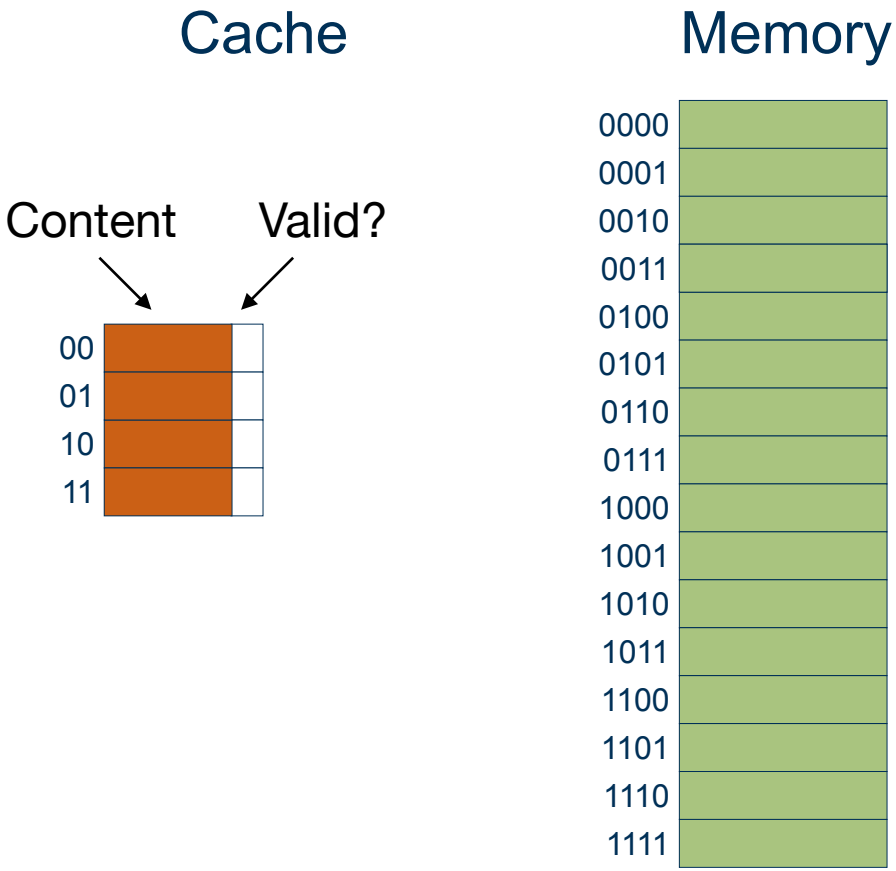
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.

A Simple Cache



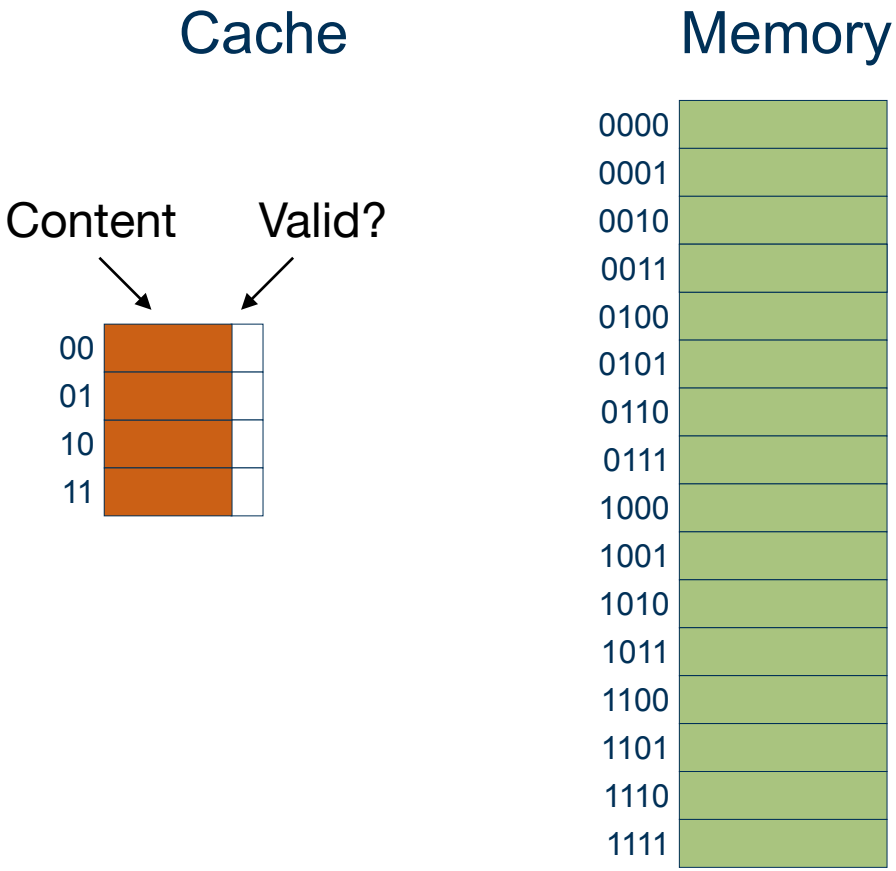
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B

A Simple Cache



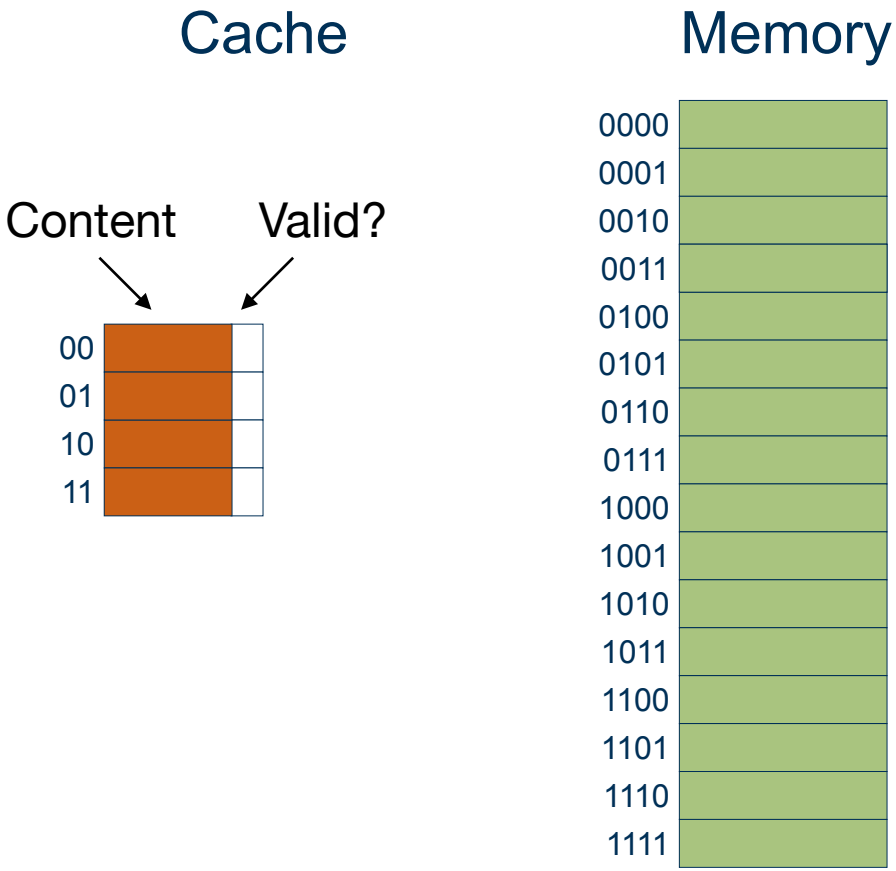
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line

A Simple Cache



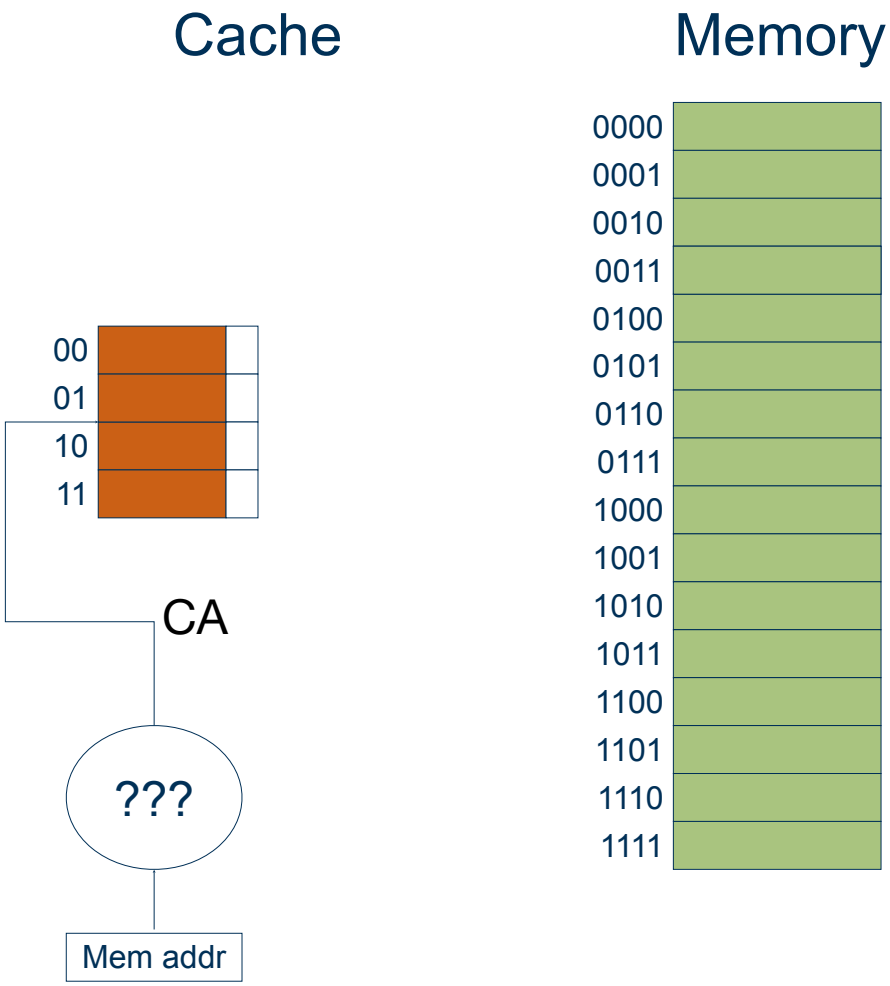
- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line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

A Simple Cache

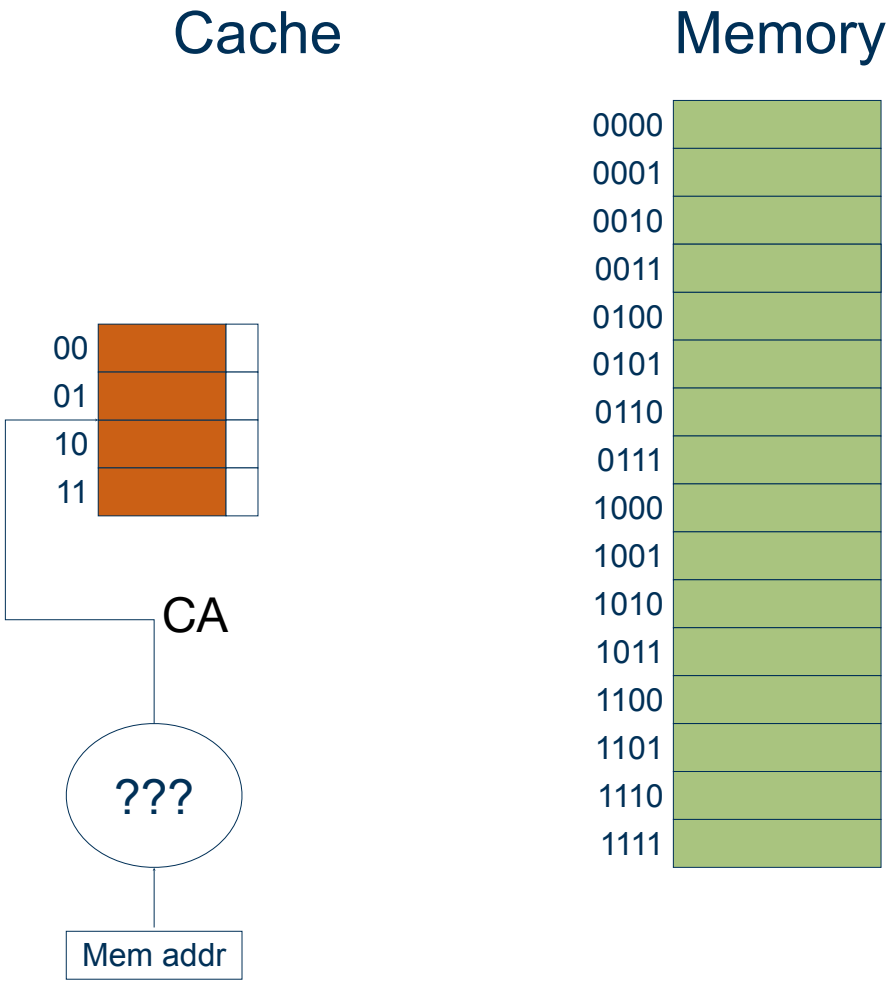


- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line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

Cache Placement

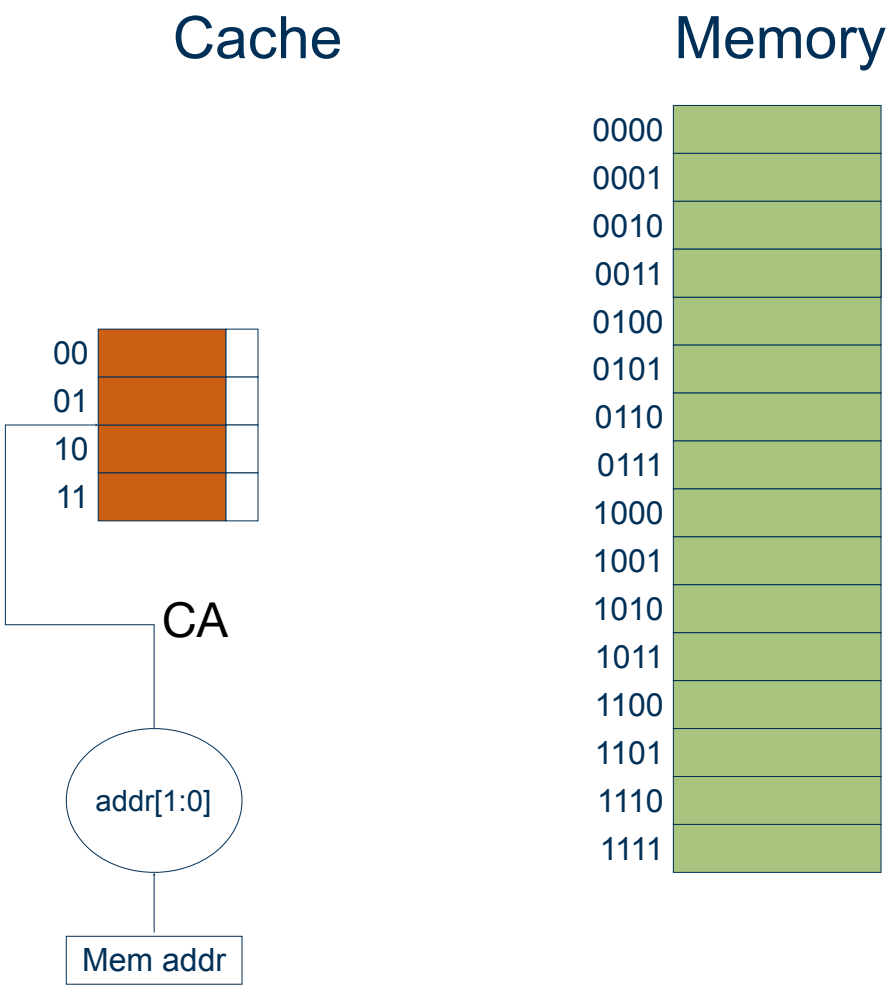


Cache Placement



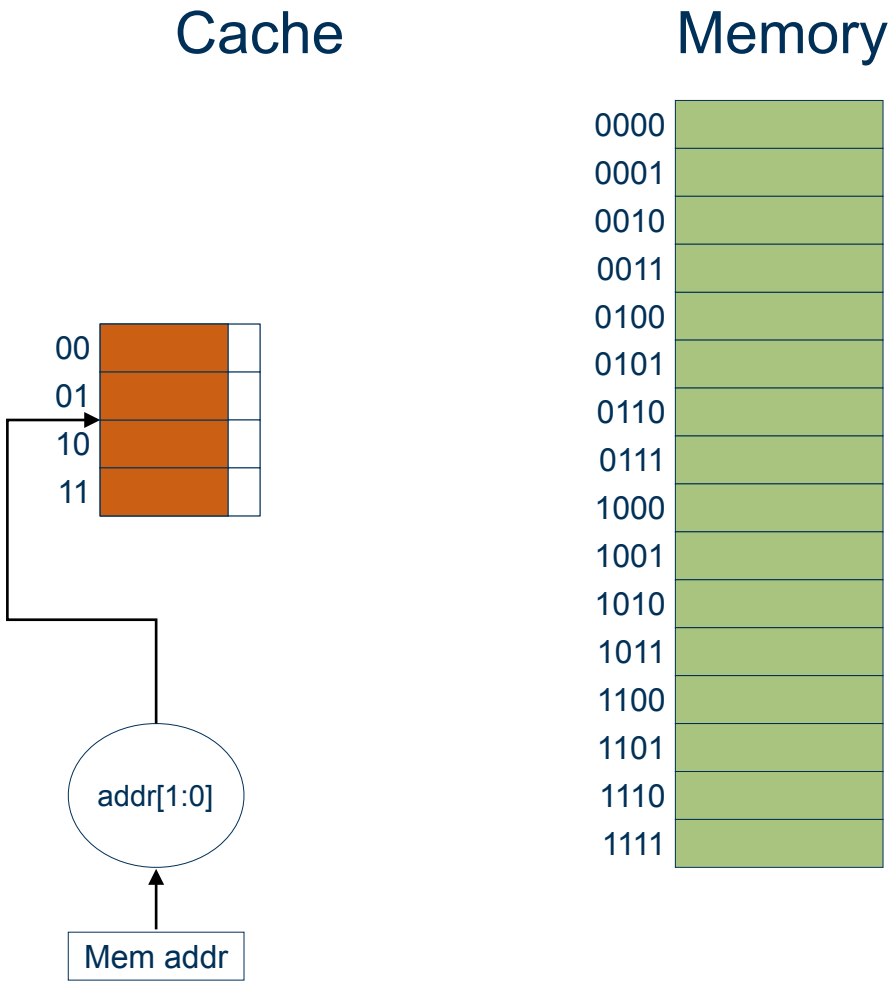
- Given a memory addr, say 0x0001, we want to put the data there into the cache; where does the data go?

Function to Address Cache



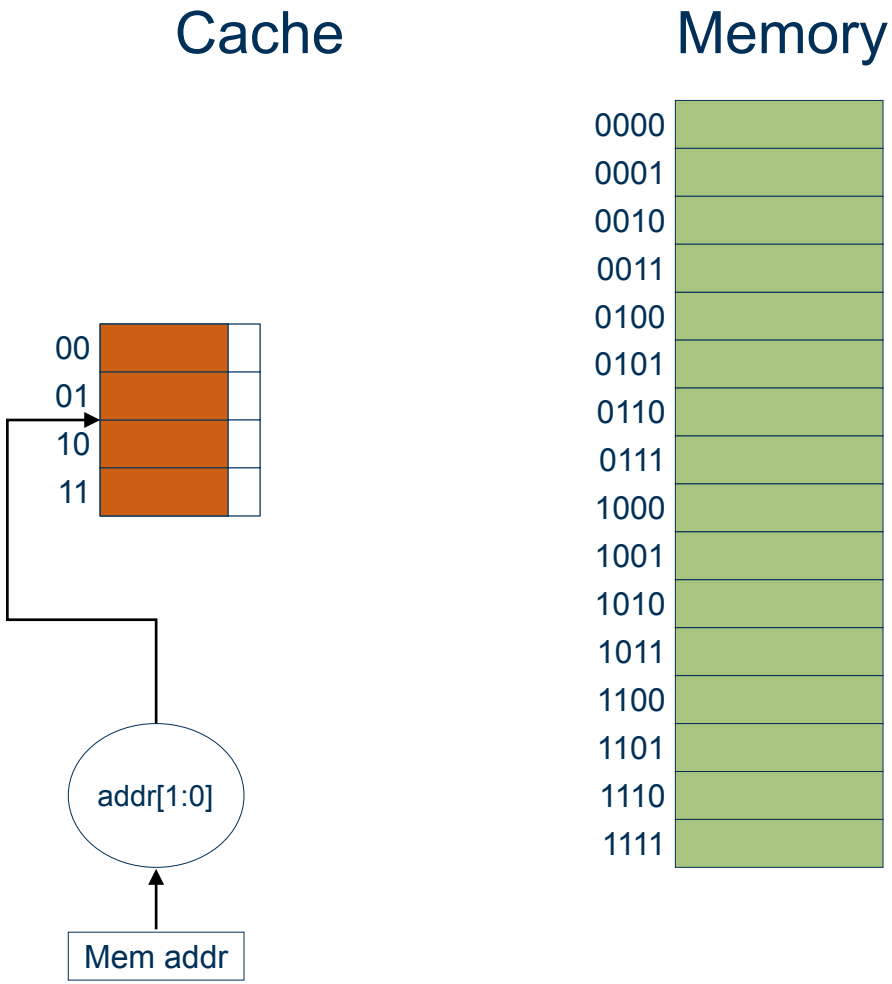
- Simplest way is to take a subset of address bits
- Six combinations in total
  - CA = ADDR[3],ADDR[2]
  - CA = ADDR[3],ADDR[1]
  - CA = ADDR[3],ADDR[0]
  - CA = ADDR[2],ADDR[1]
  - CA = ADDR[2],ADDR[0]
  - CA = ADDR[1],ADDR[0]
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]

Direct-Mapped Cache



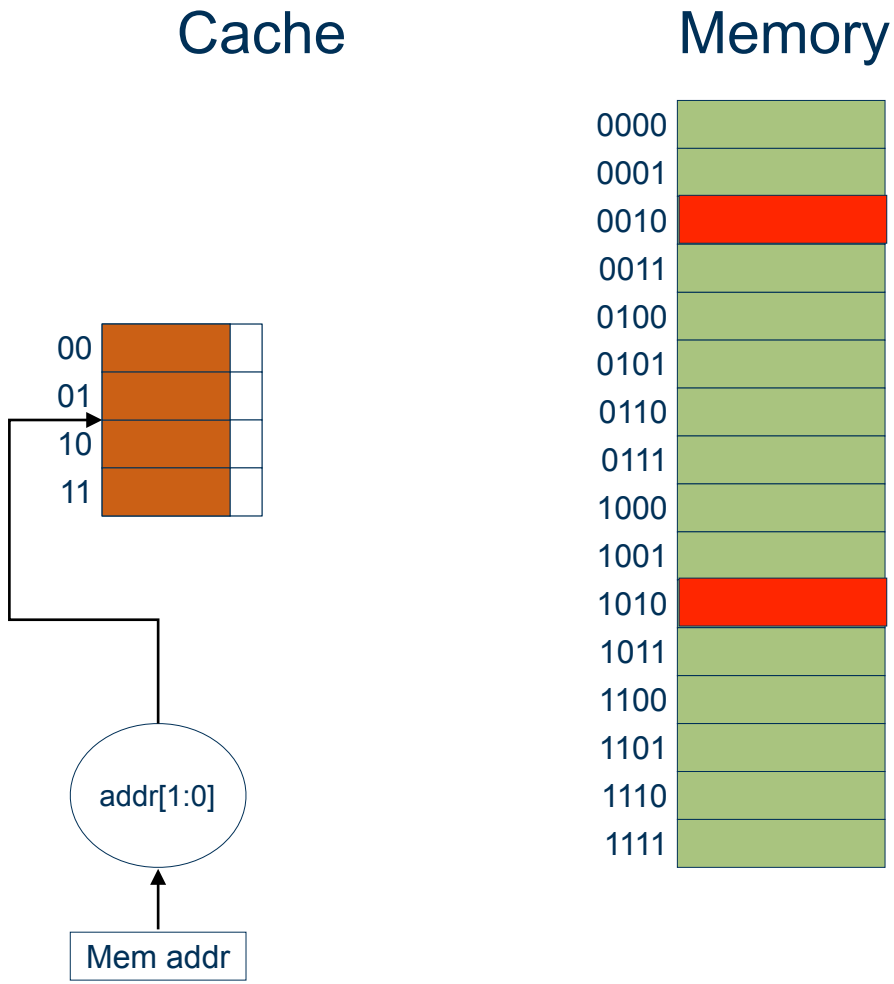
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits

Direct-Mapped Cache



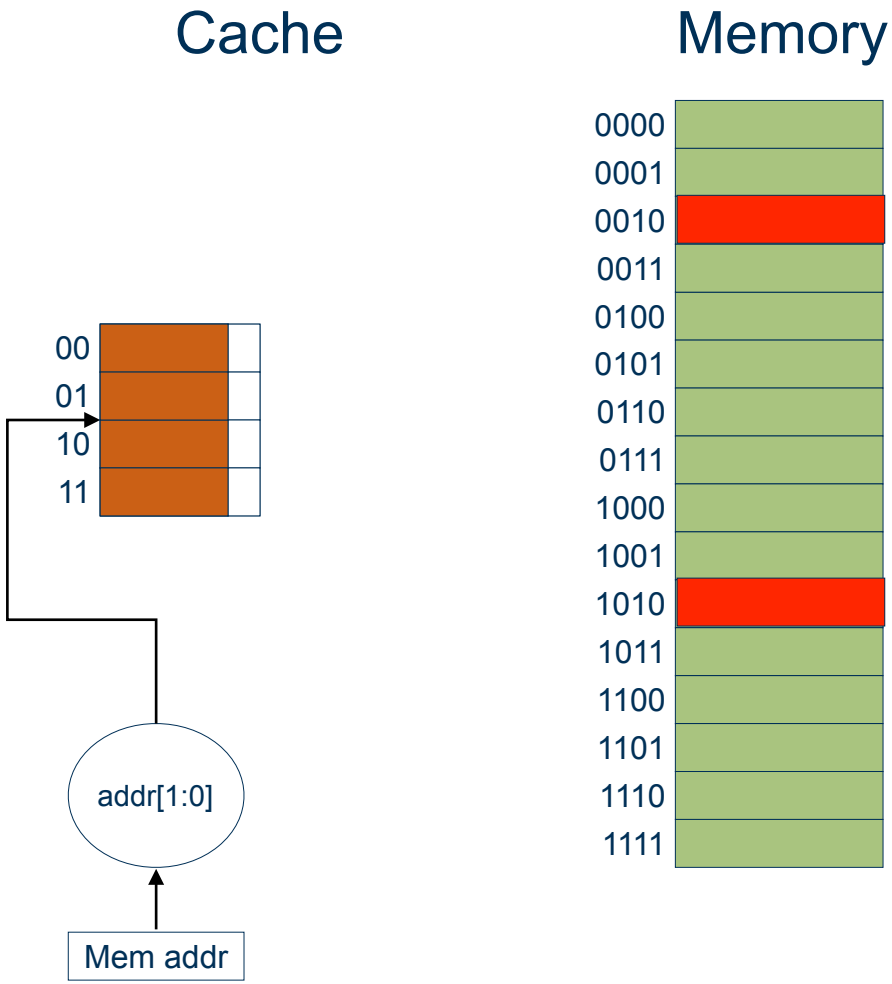
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location

Direct-Mapped Cache



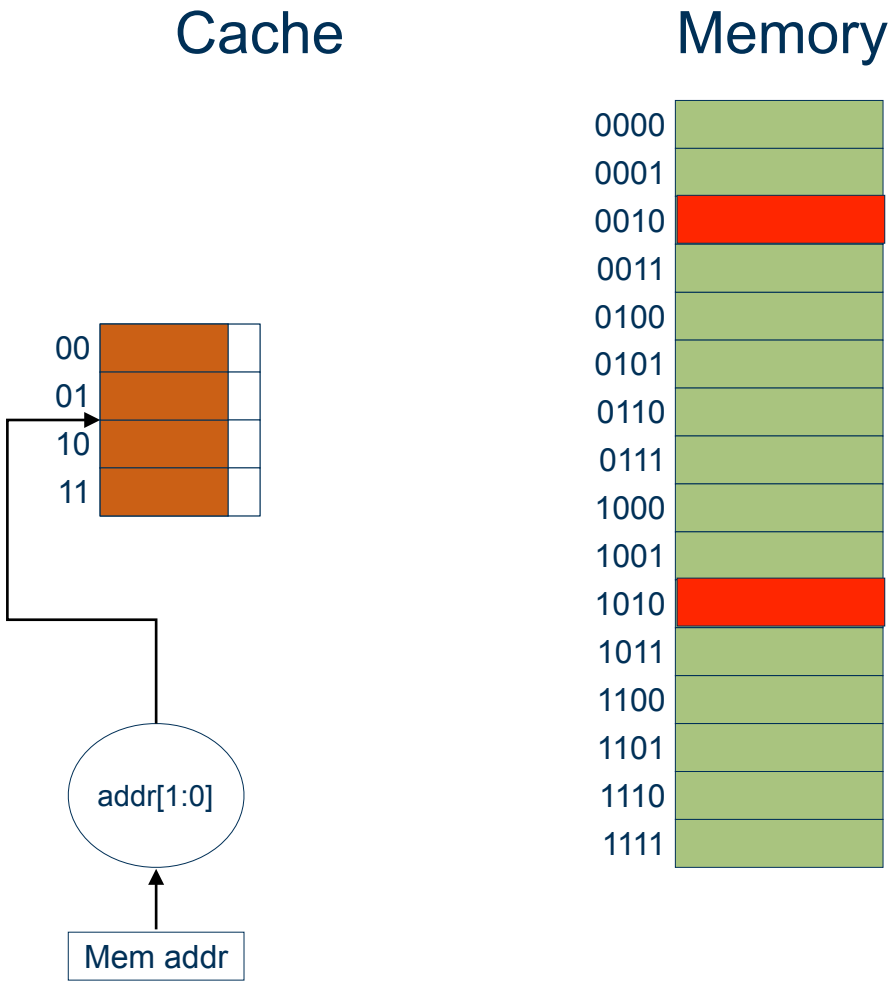
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010

Direct-Mapped Cache



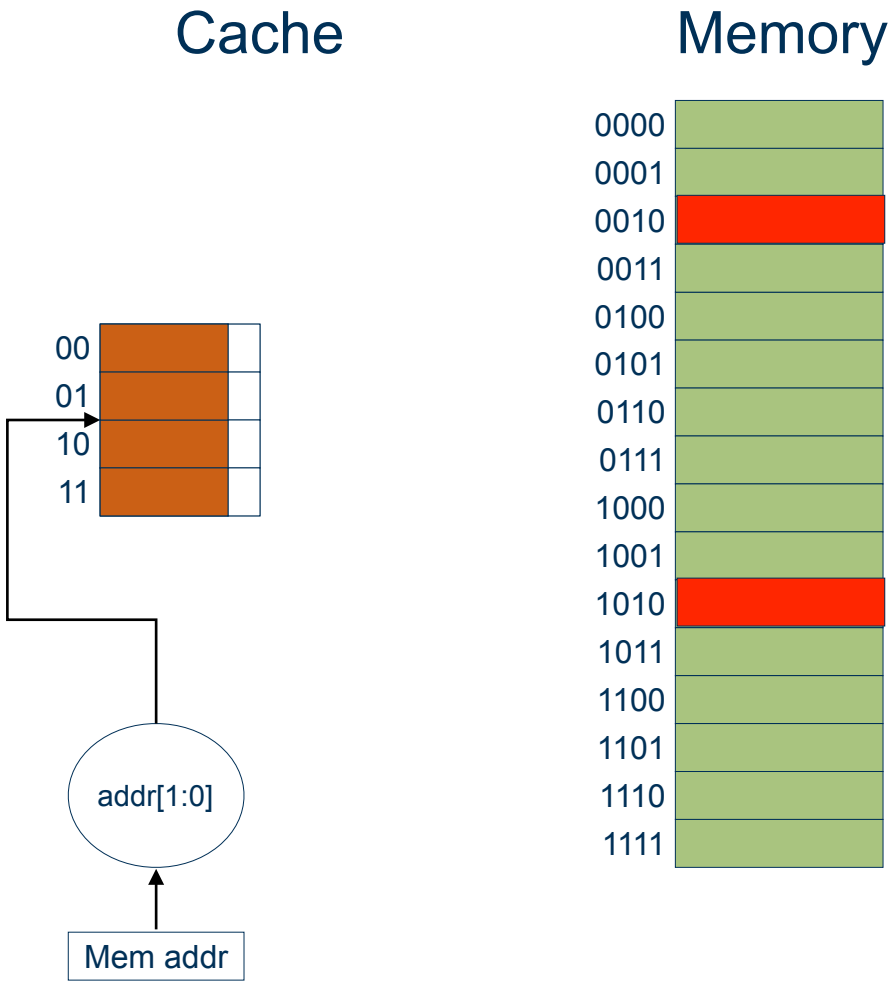
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?

Direct-Mapped Cache



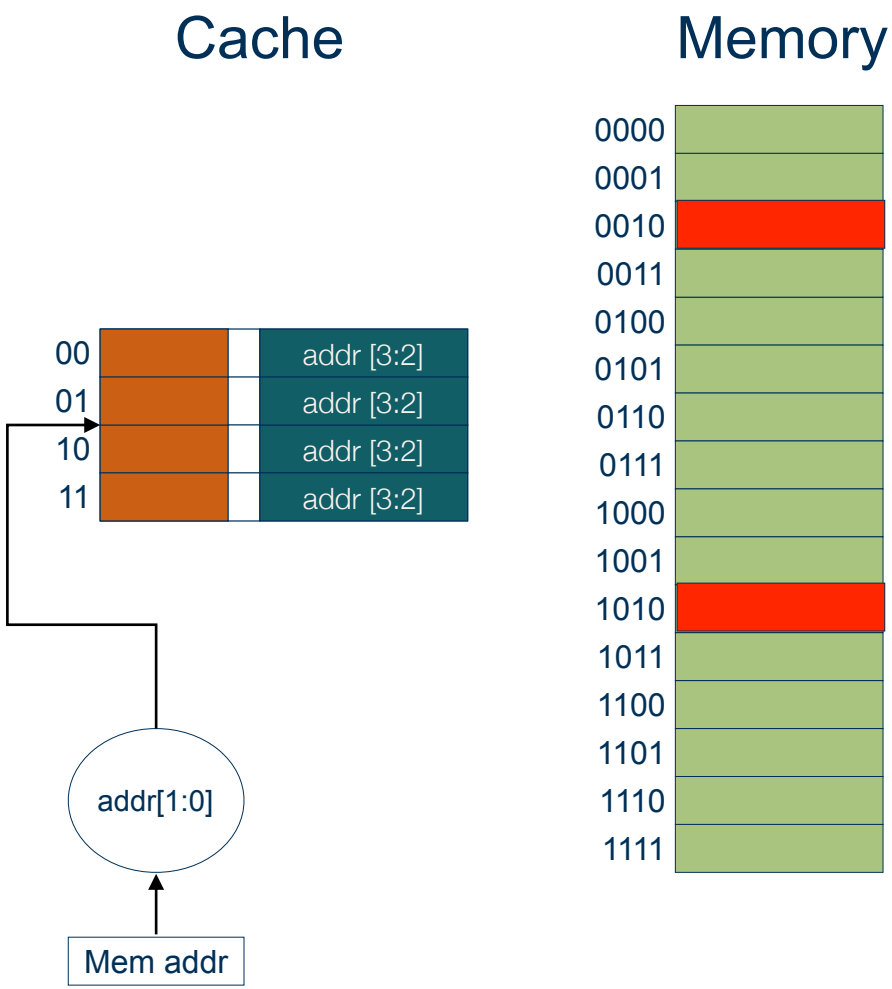
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

Direct-Mapped Cache



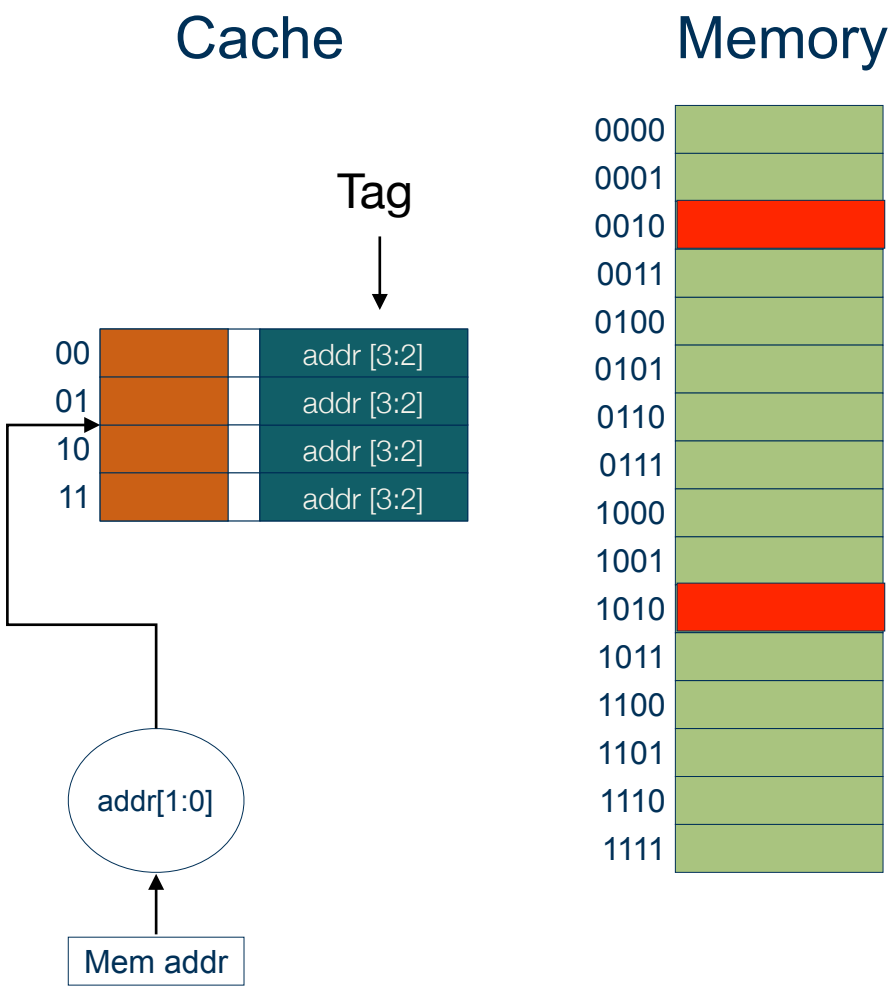
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

Direct-Mapped Cache



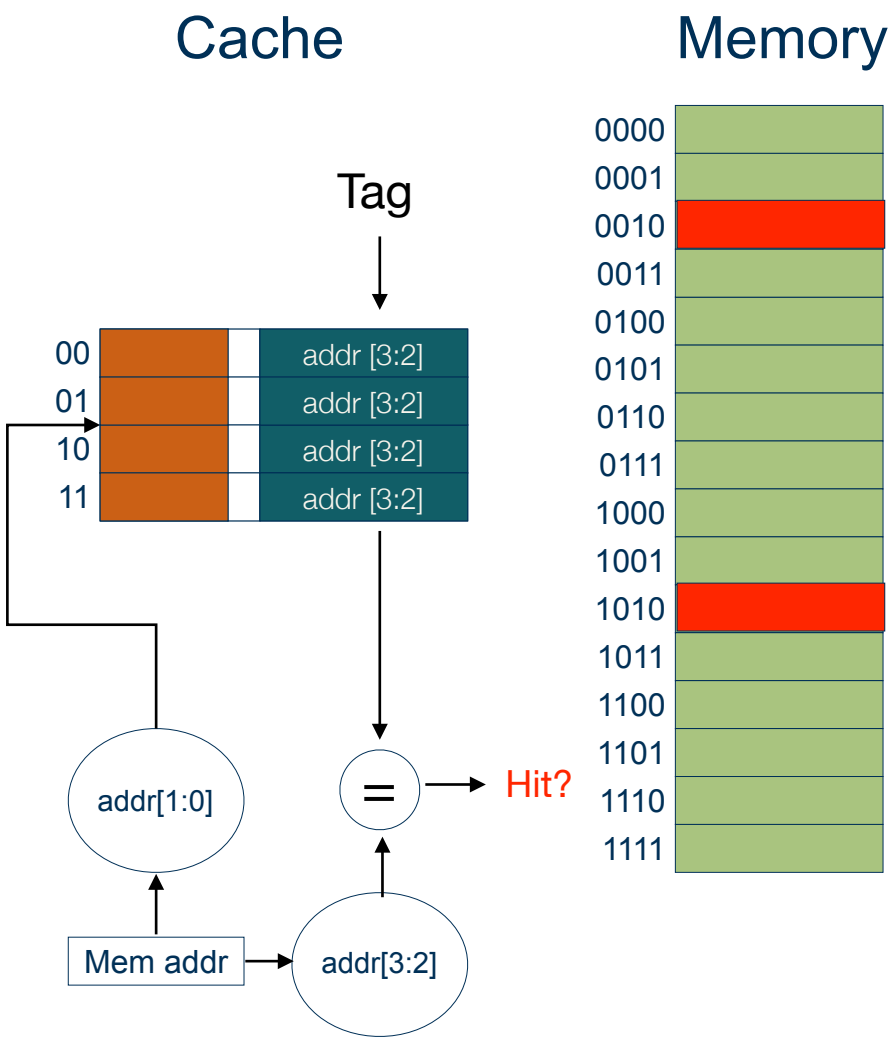
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

Direct-Mapped Cache



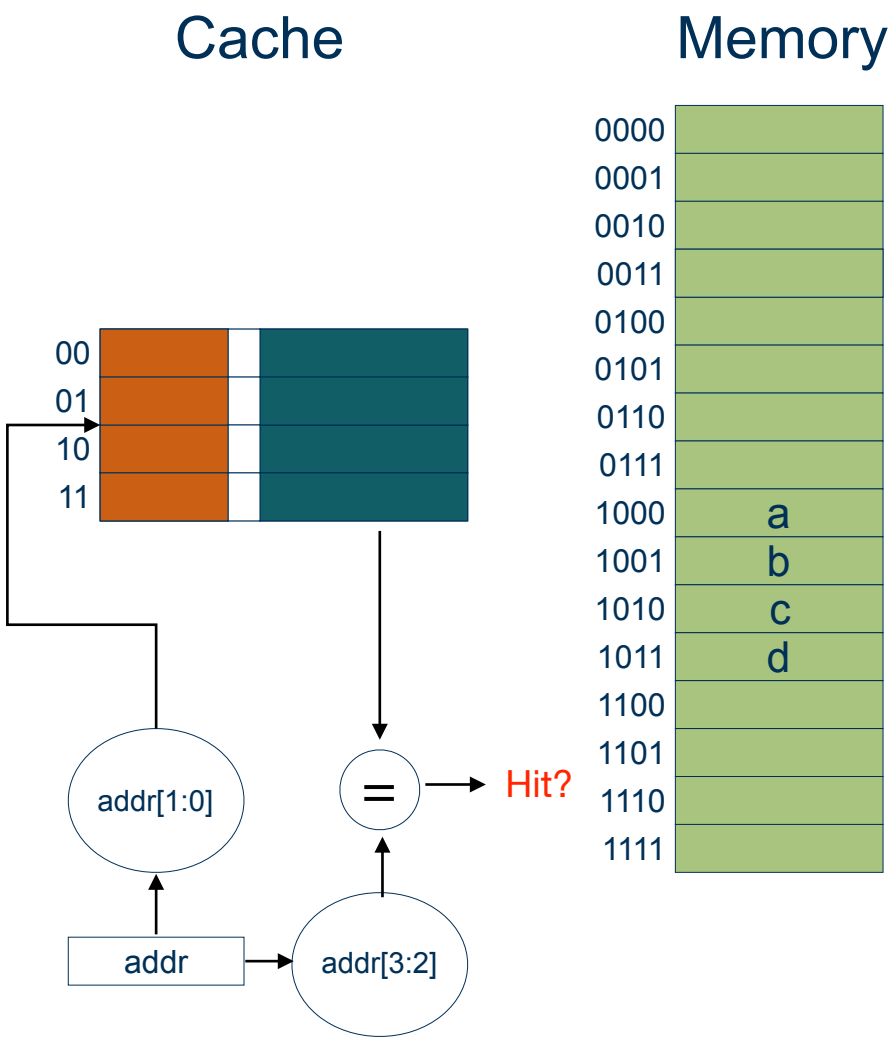
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

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

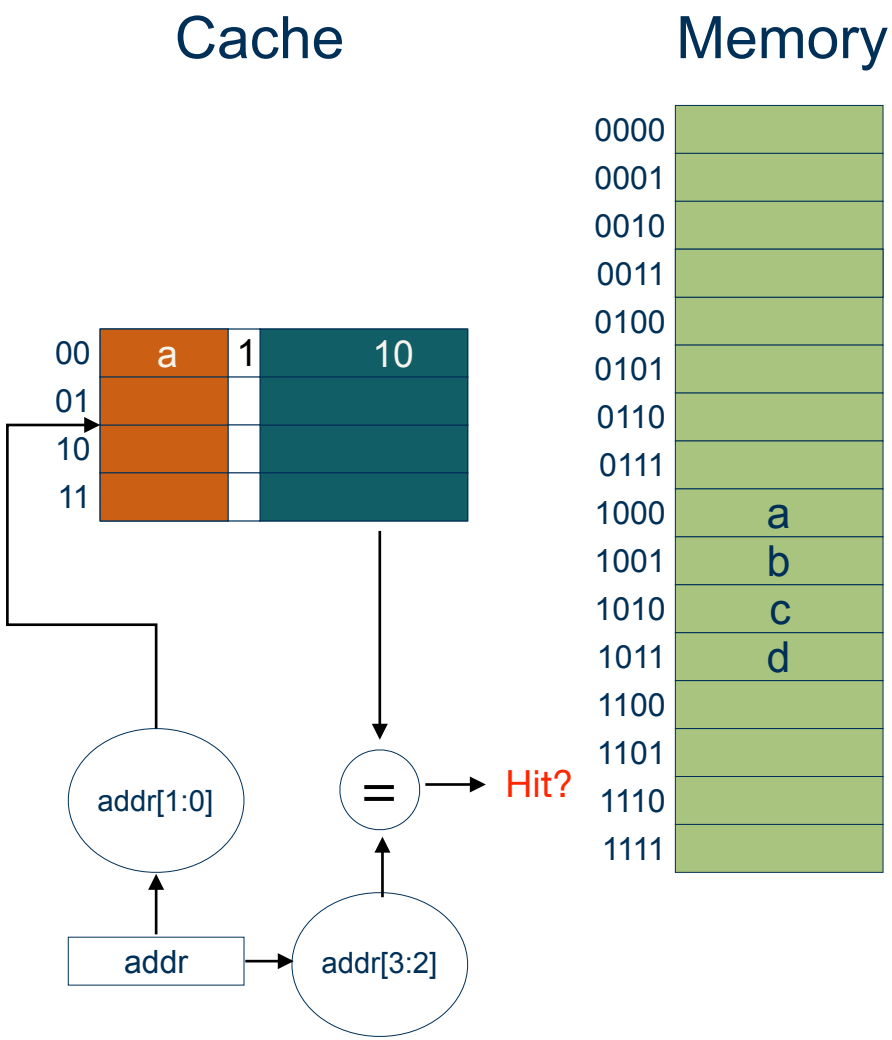
Example: Direct-Mapped Cache




```
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?

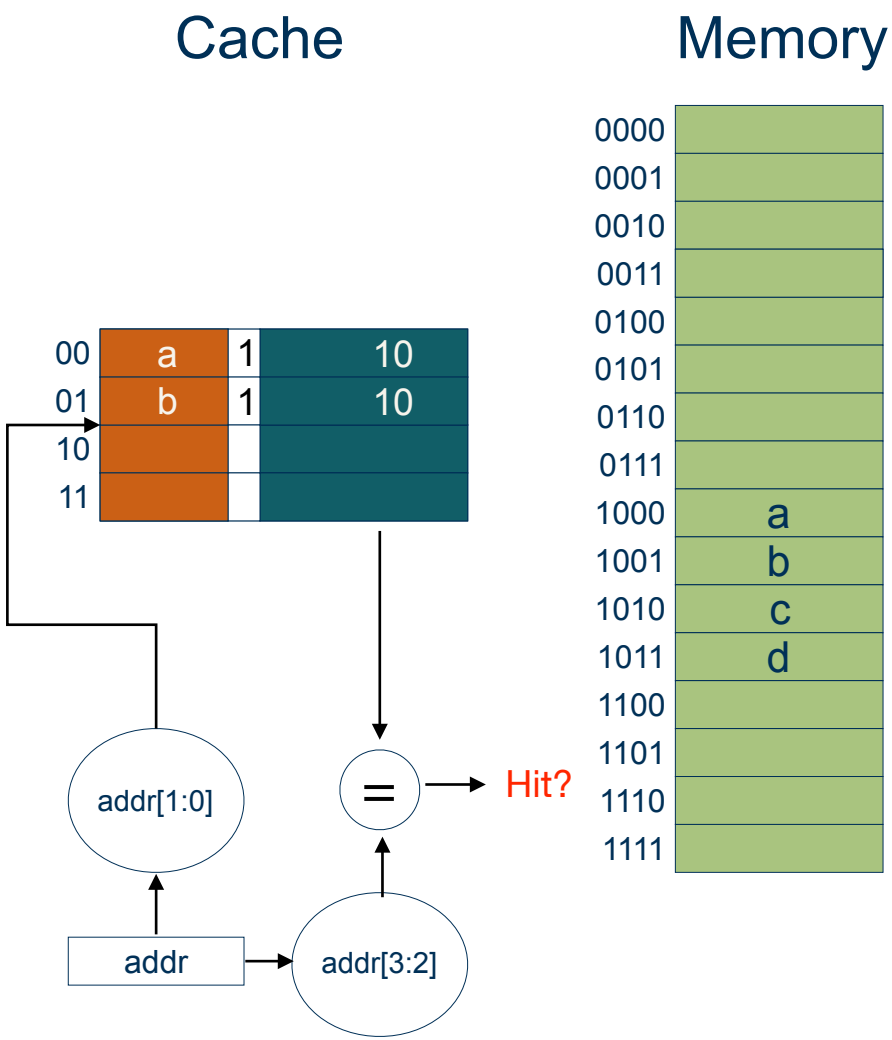
Example: Direct-Mapped Cache




```
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?

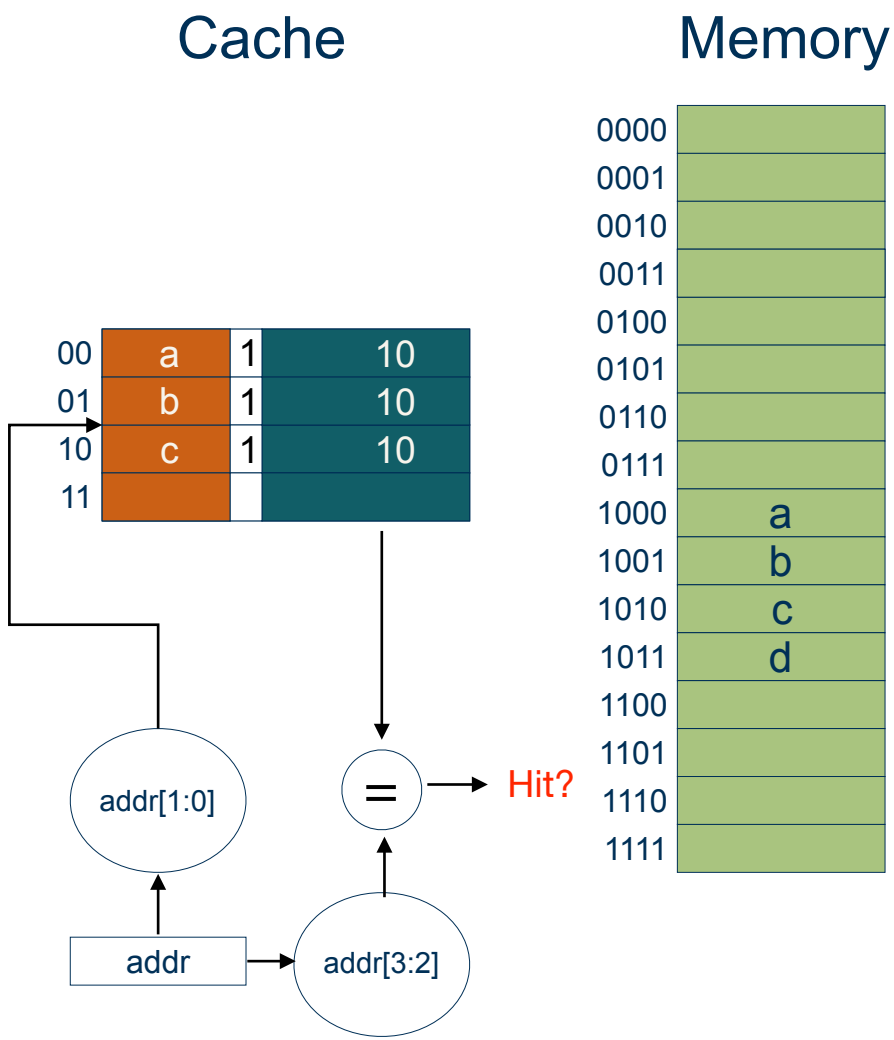
Example: Direct-Mapped Cache




```
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?

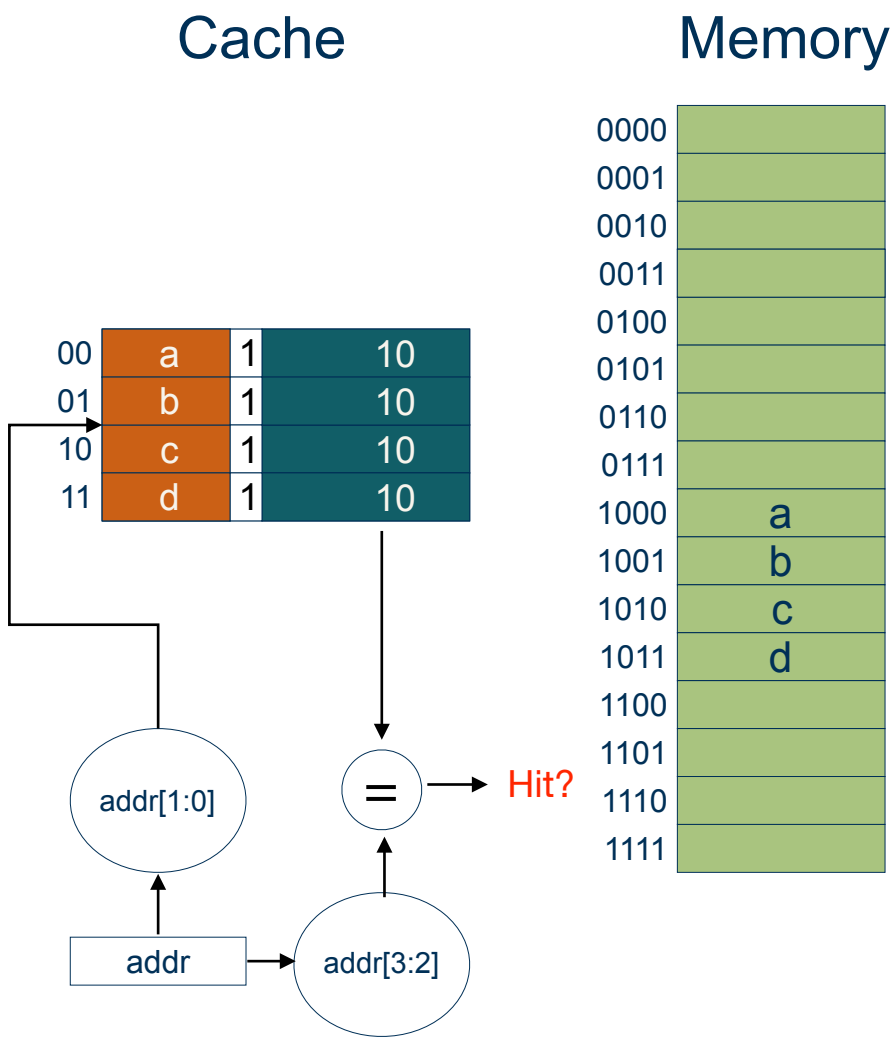
Example: Direct-Mapped Cache




```
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?

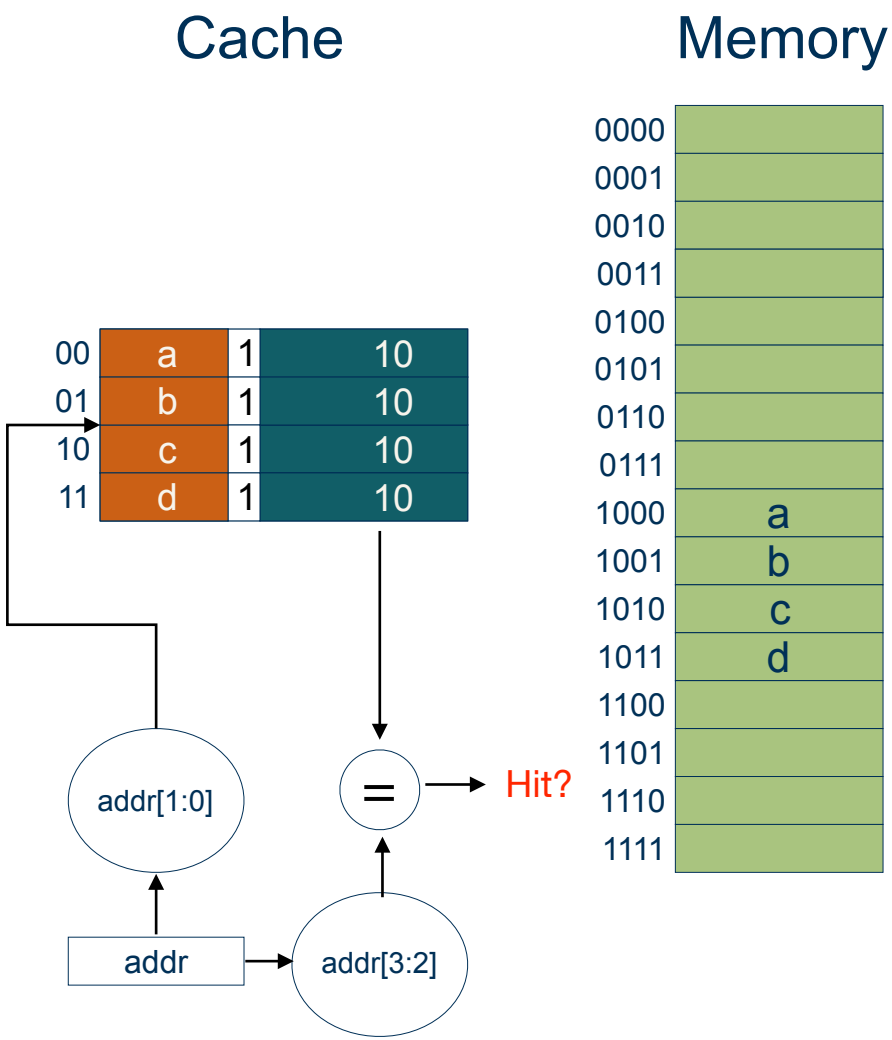
Example: Direct-Mapped Cache




```
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?

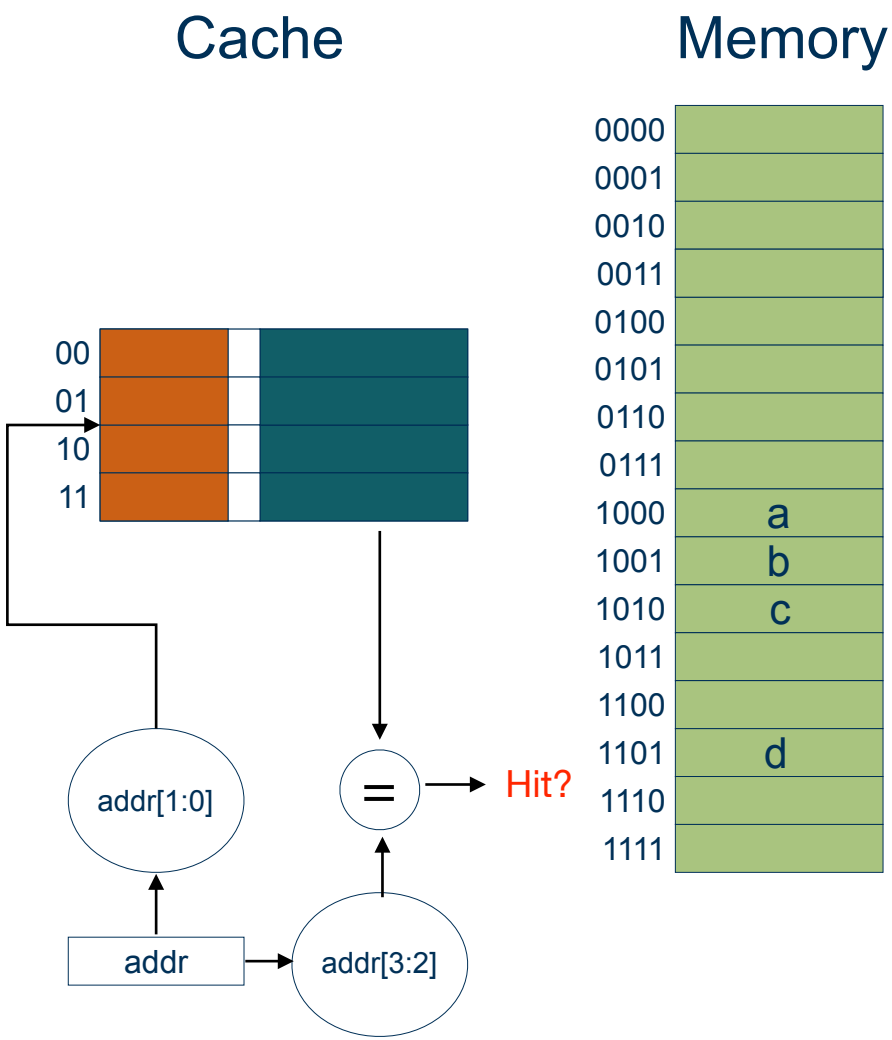
Example: Direct-Mapped Cache




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

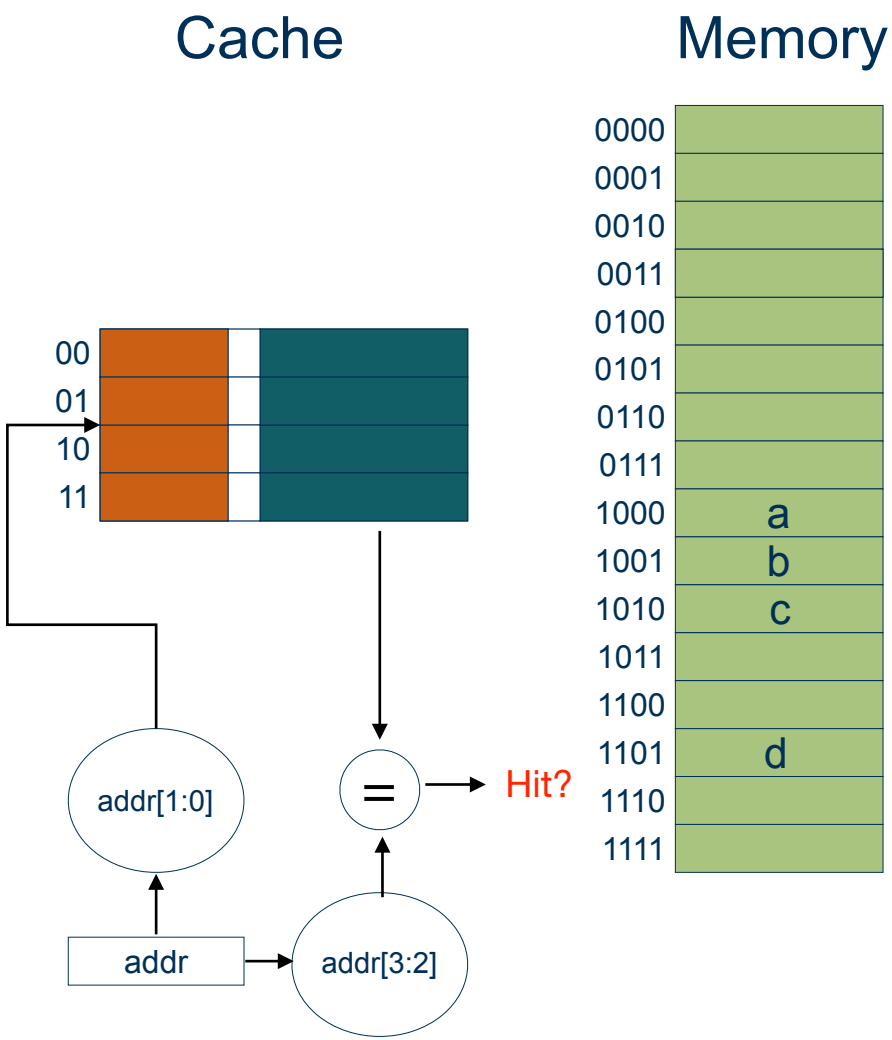
- Assume mem == 0b1000
- Read 0b1000
- Read 0b1001
- Read 0b1010
- Read 0b1011
- Read 0b1000; cache hit?

Conflicts



Assume the following memory access stream:

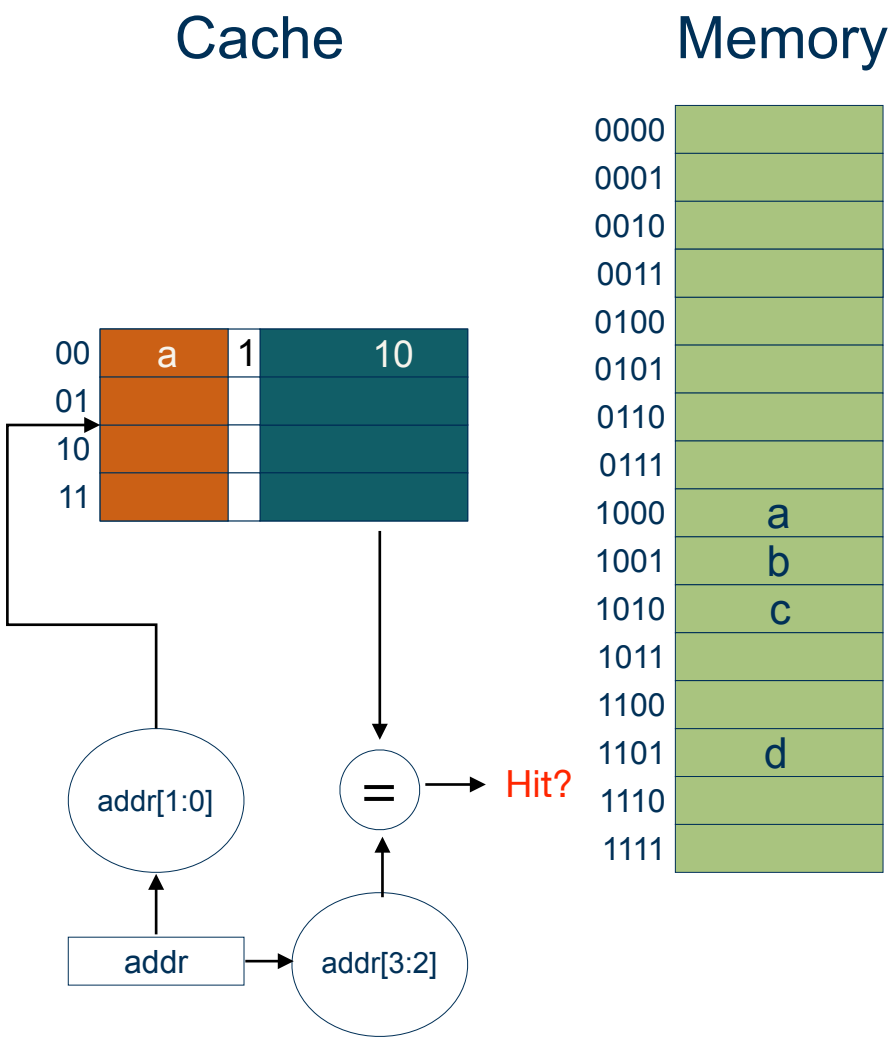
Conflicts



Assume the following memory access stream:

- Read 1000

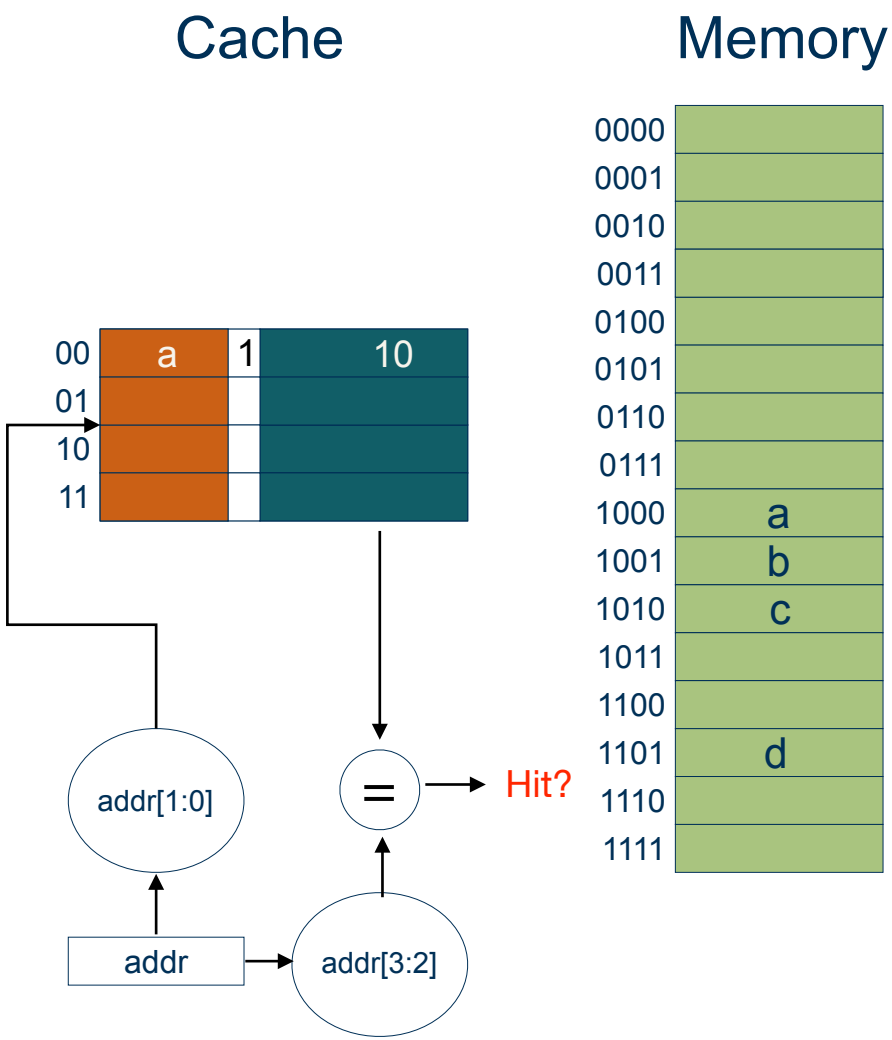
Conflicts



Assume the following memory access stream:

- Read 1000

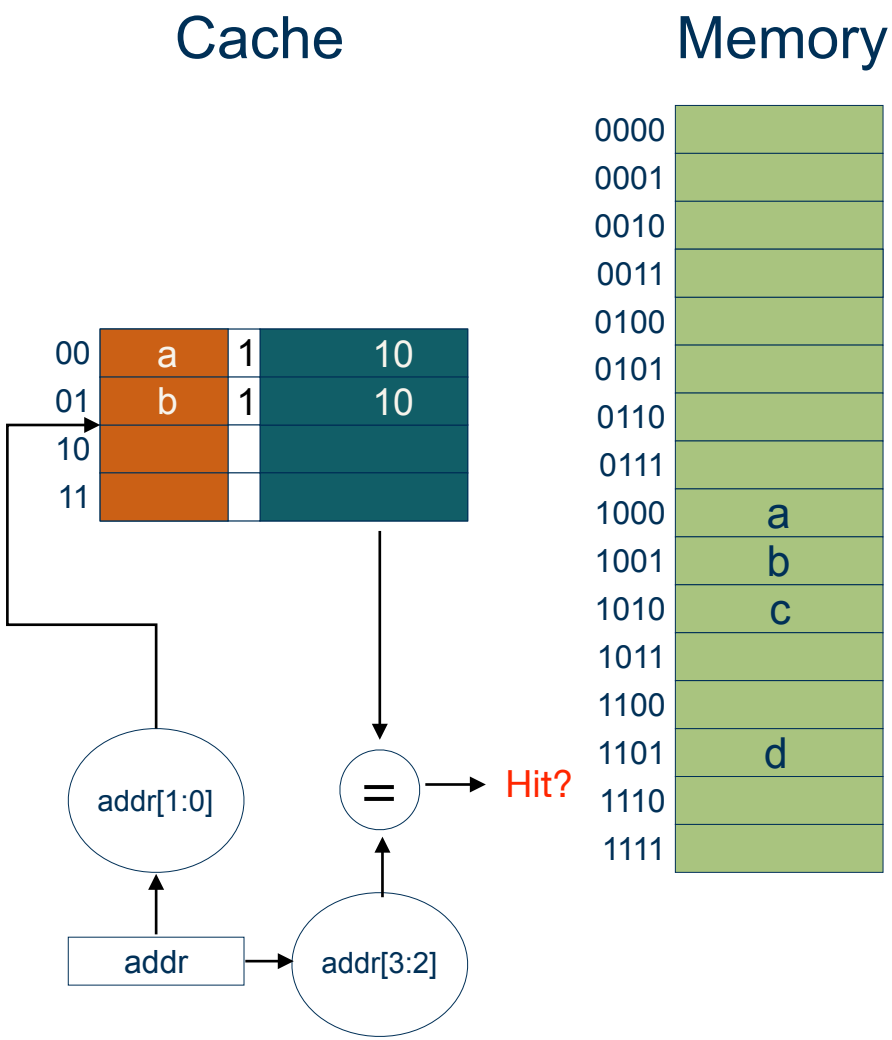
Conflicts



Assume the following memory access stream:

- Read 1000
- Read 1001

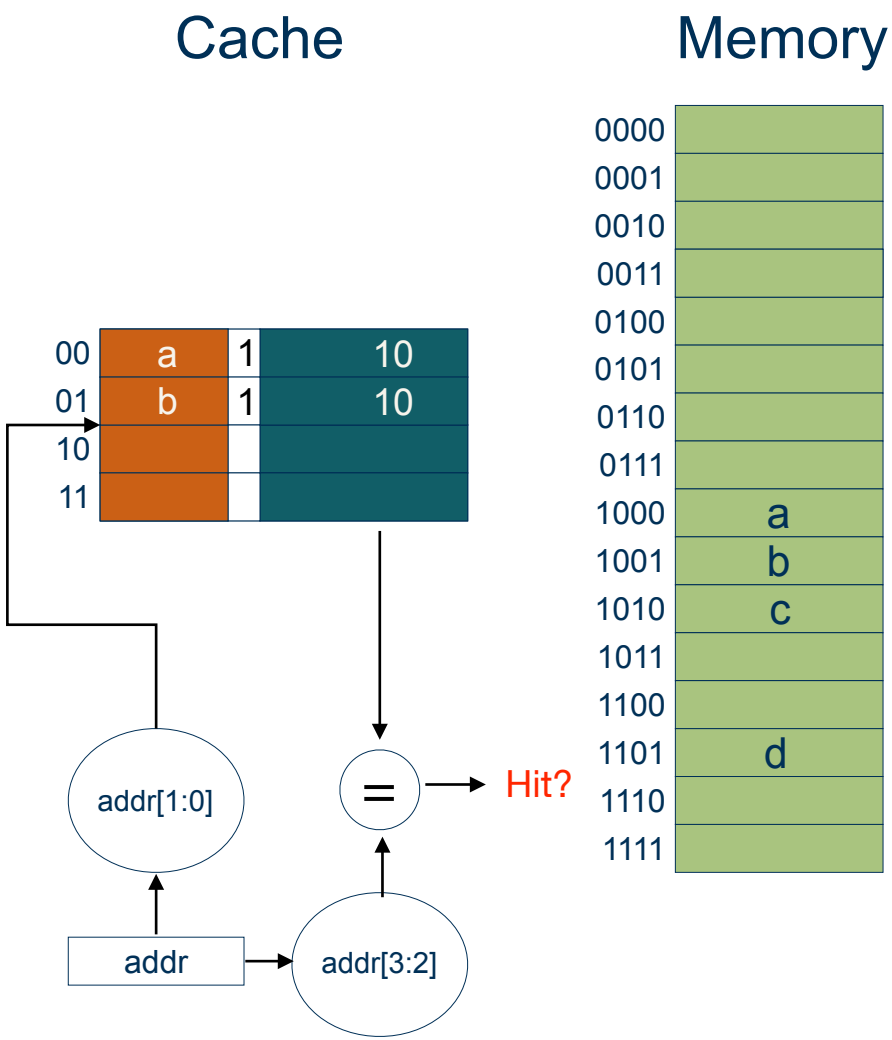
Conflicts



Assume the following memory access stream:

- Read 1000
- Read 1001

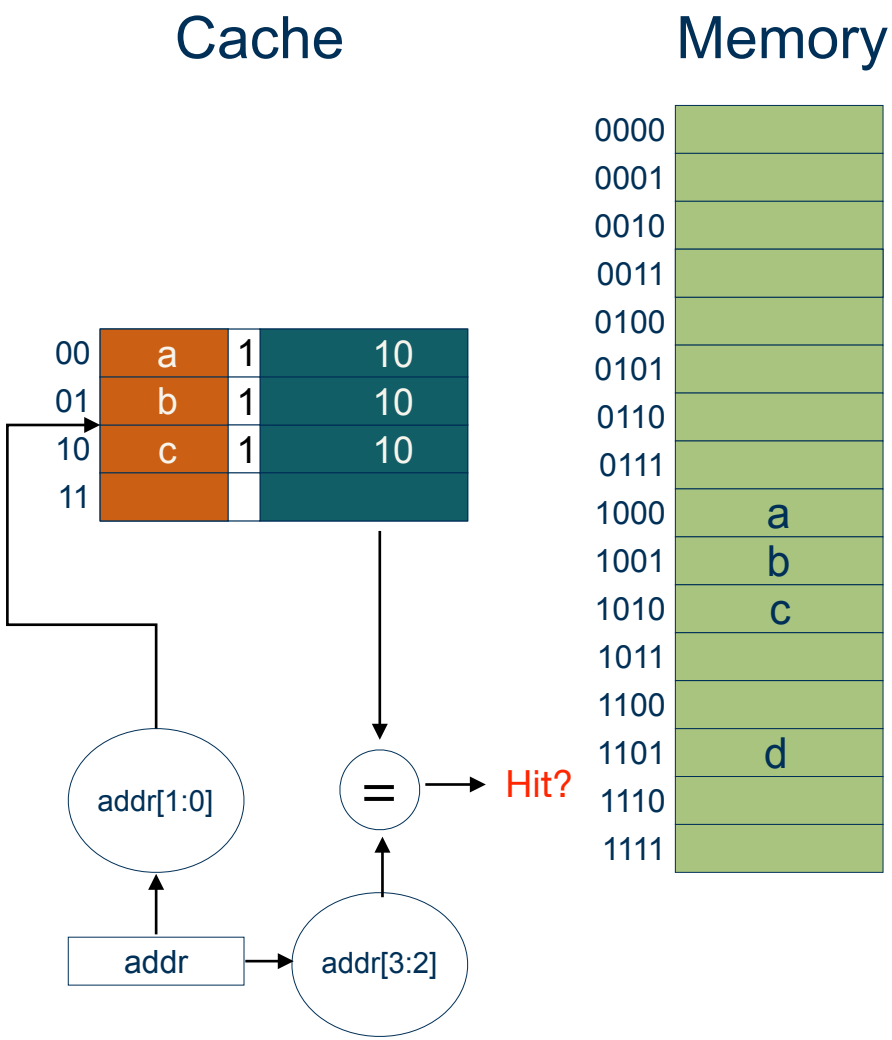
Conflicts



Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

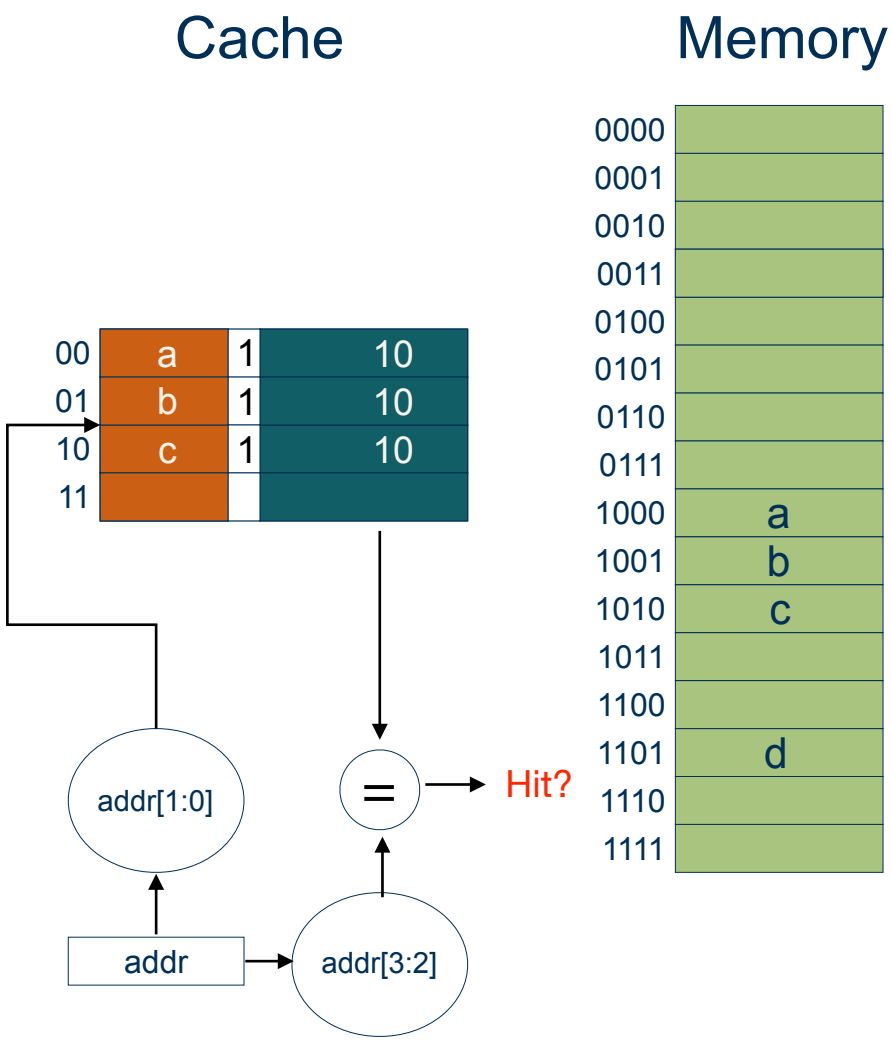
Conflicts



Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

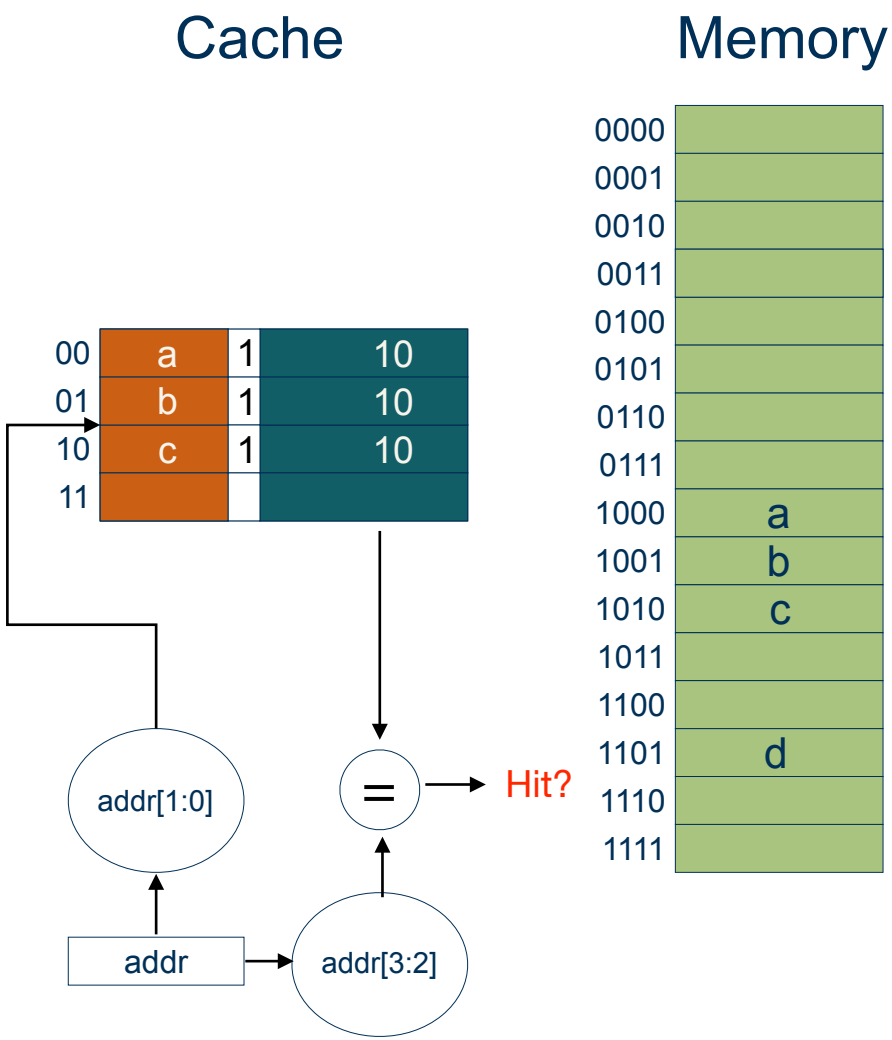
Conflicts



Assume the following memory access stream:

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

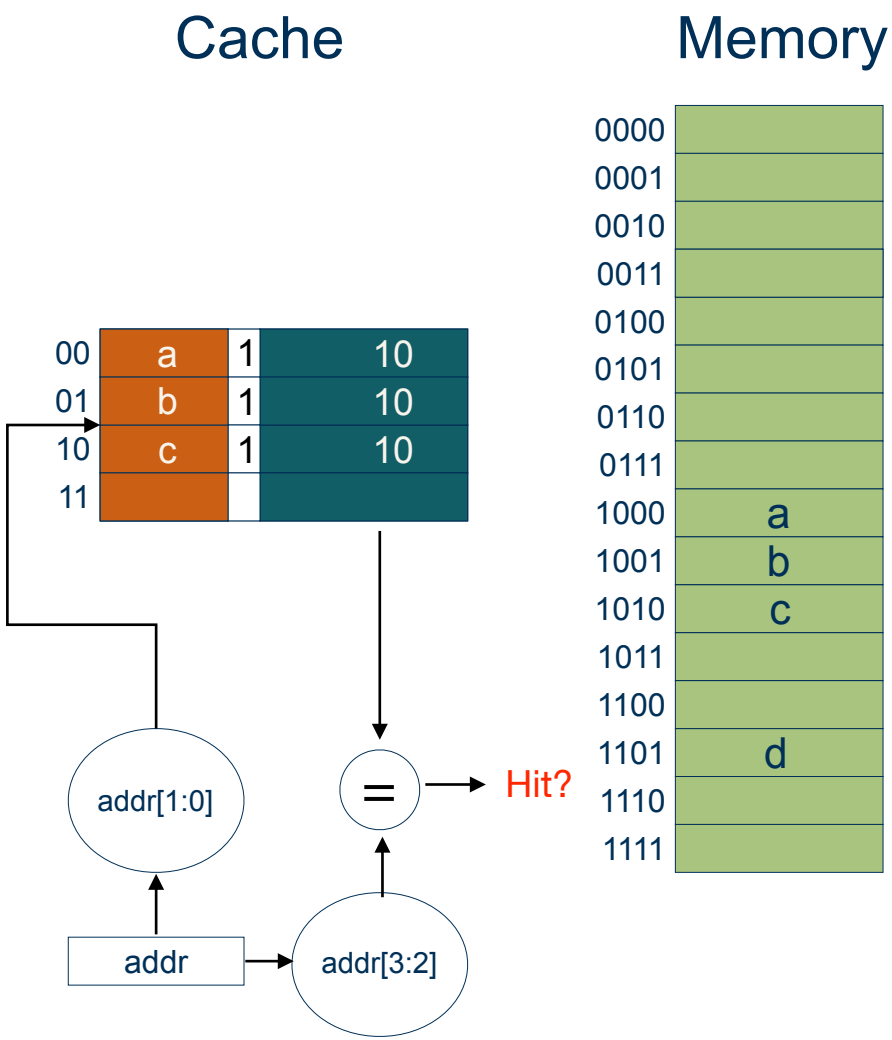
Conflicts



Assume the following memory access stream:

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

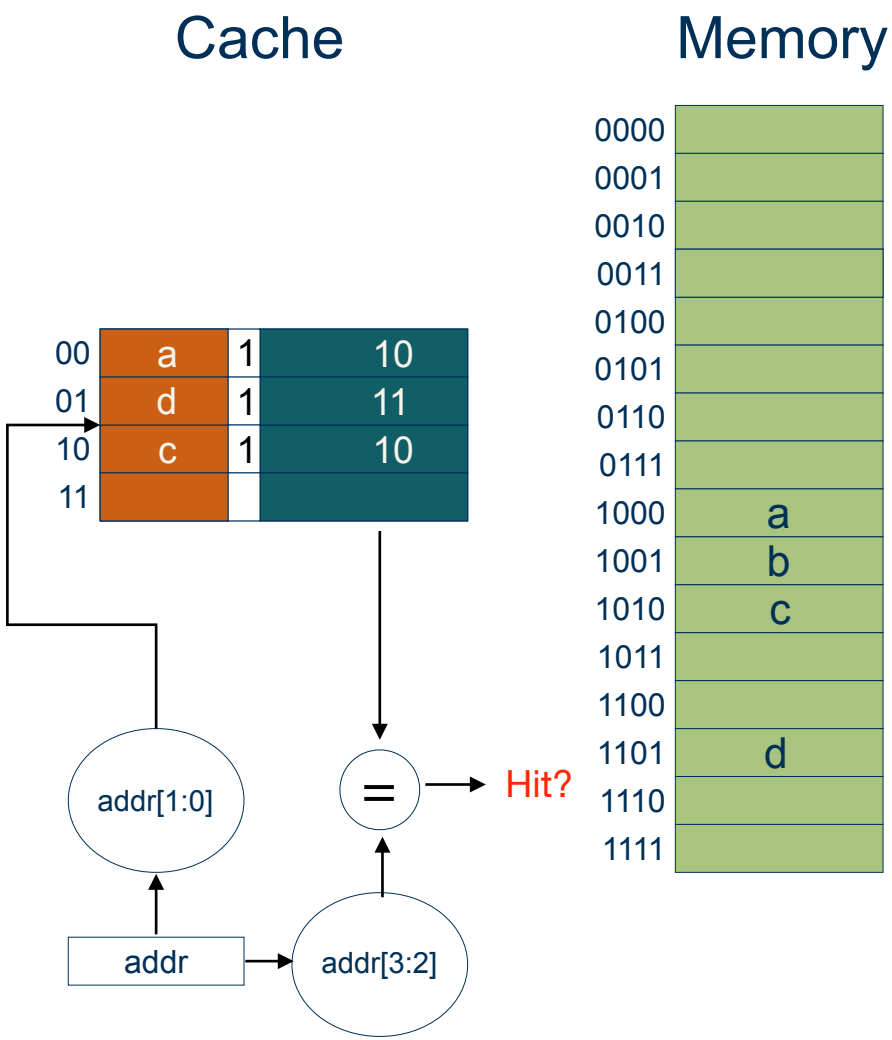
Conflicts



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

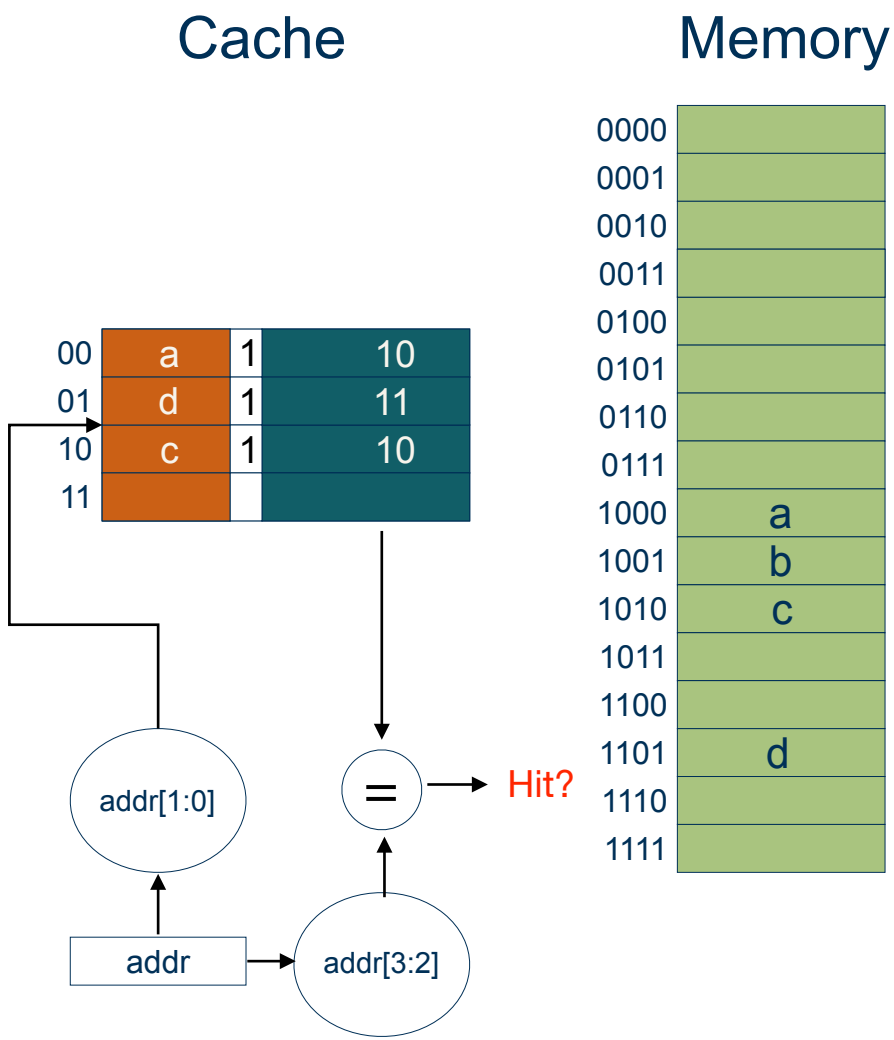
Conflicts



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

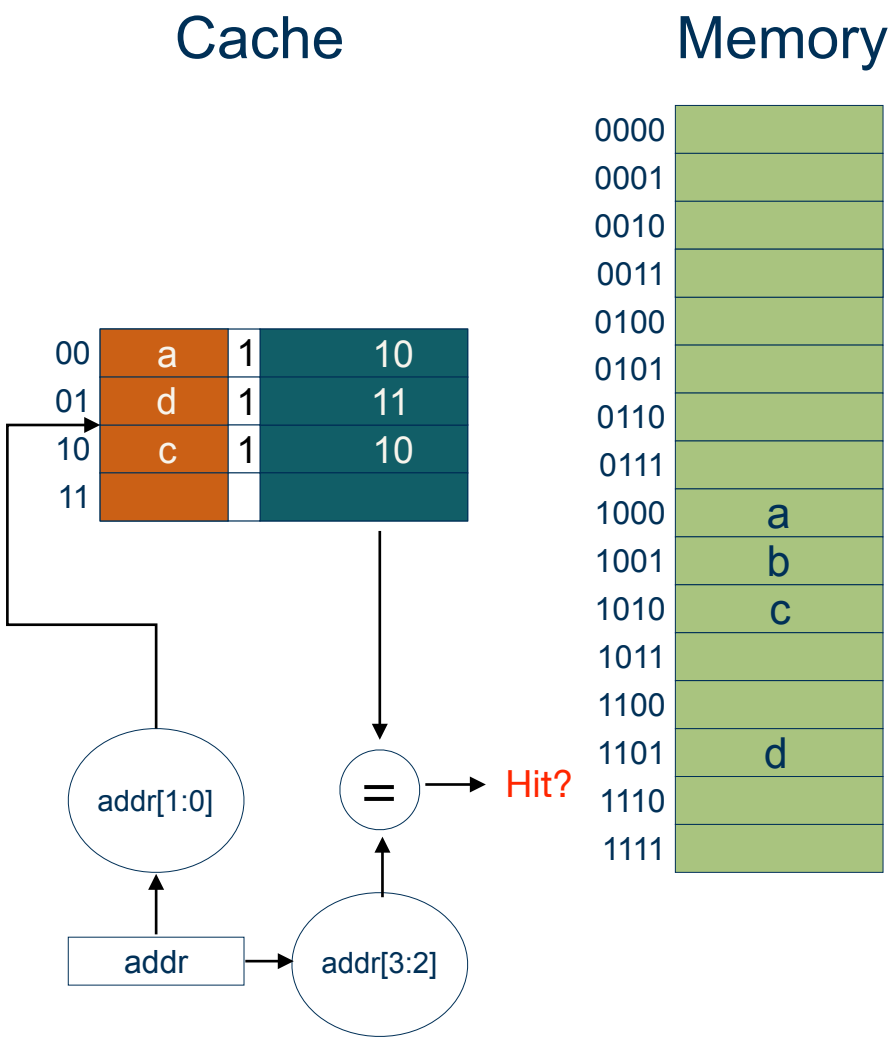
Conflicts



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!**

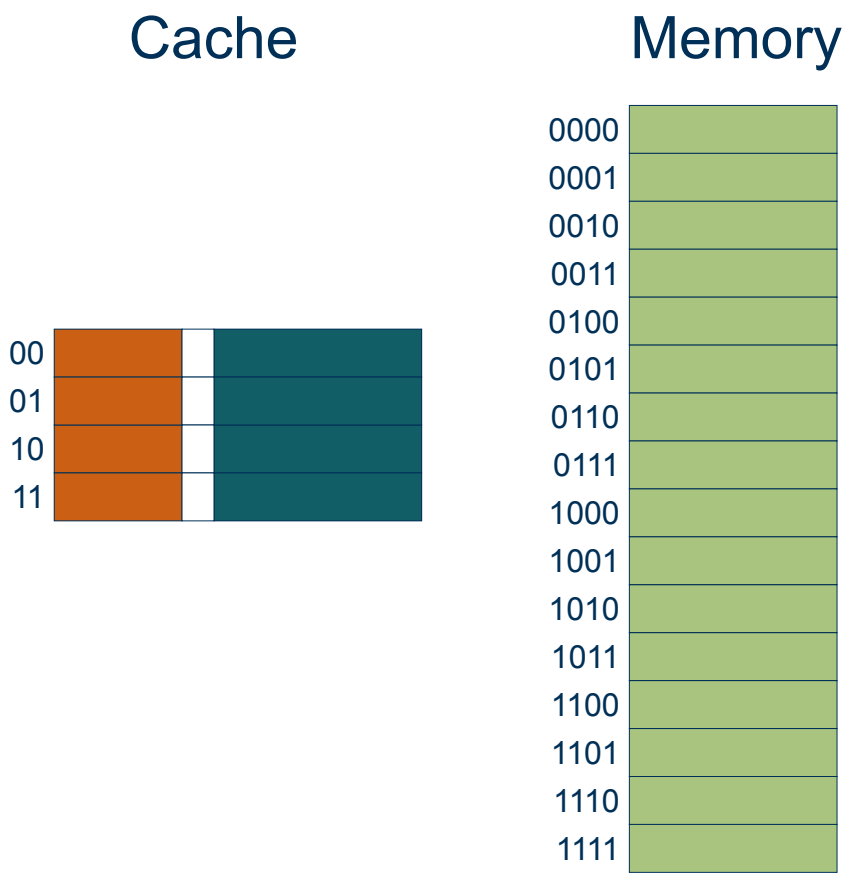
Conflicts



Assume the following memory access stream:

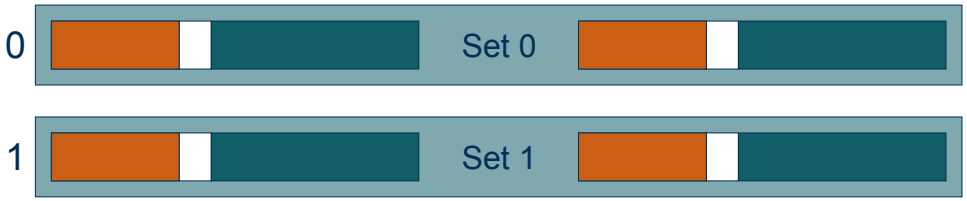
- Read 1000
- Read 1001
- Read 1010
- Read 1101 **(kick out 1001)**
  - addr[1:0]: 01
  - addr[3:2]: 11
- Read 1001 -> **Miss!**
- Why? Each memory location is mapped to only one cache location

Sets



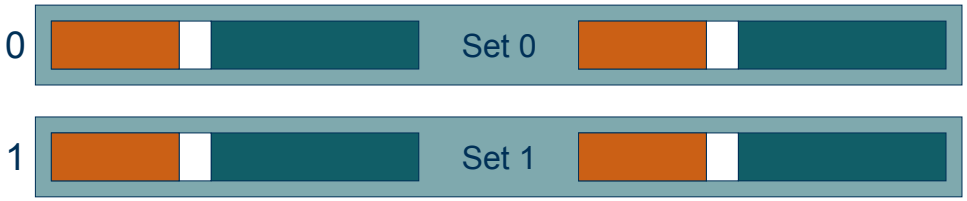
- Each cacheable memory location is mapped to **a set of cache locations**
- A set is one or more cache locations
- Set size is the number of locations in a set, also called **associativity**

2 Way Set Associative



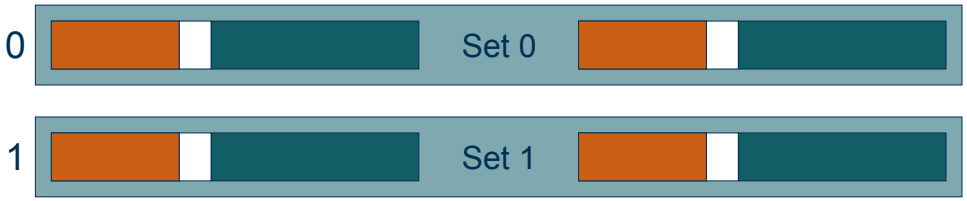
2 Way Set Associative

- 2 sets, each set has two entries



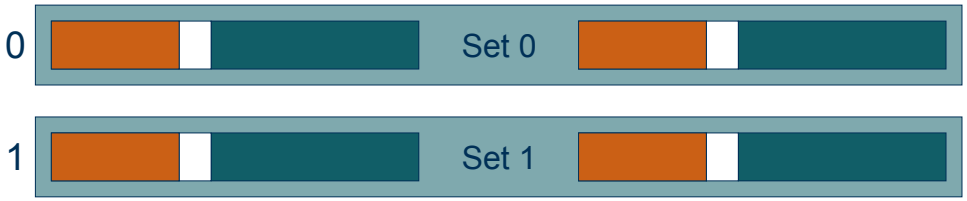
2 Way Set Associative

- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now



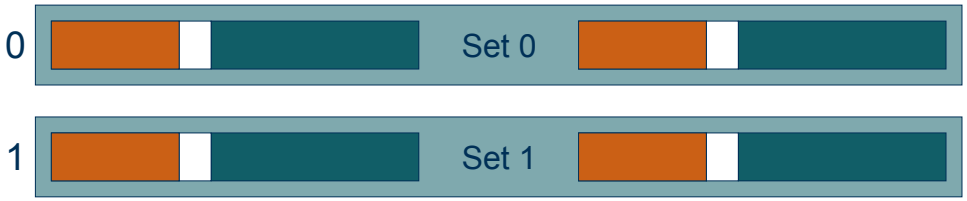
2 Way Set Associative

- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]

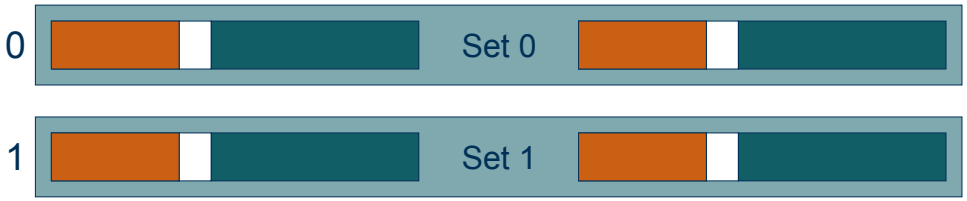


2 Way Set Associative

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

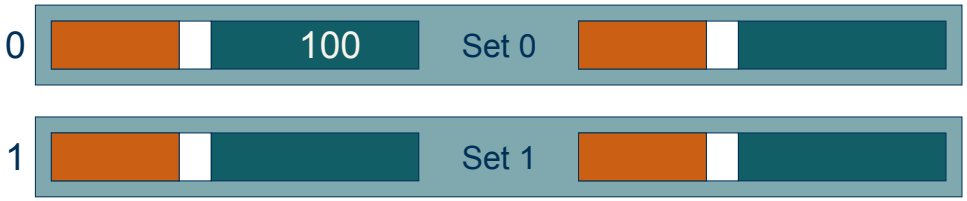


2 Way Set Associative



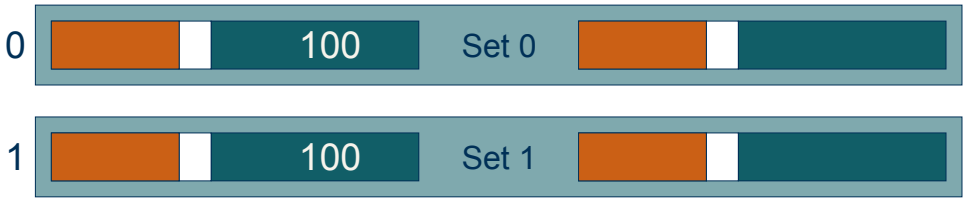
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:

2 Way Set Associative



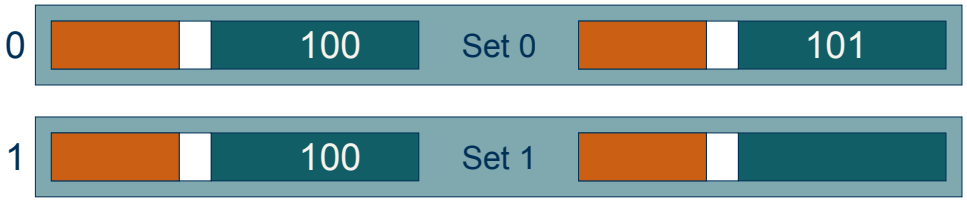
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
  - Read 1000

2 Way Set Associative



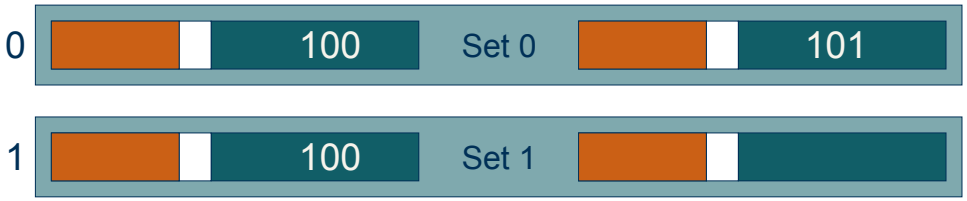
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
  - Read 1000
  - Read 1001

2 Way Set Associative



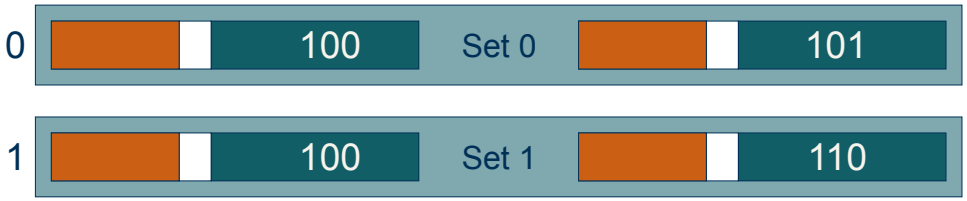
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
  - Read 1000
  - Read 1001
  - Read 1010

2 Way Set Associative



- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: 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)**

2 Way Set Associative



- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: 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

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

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

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

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

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!**

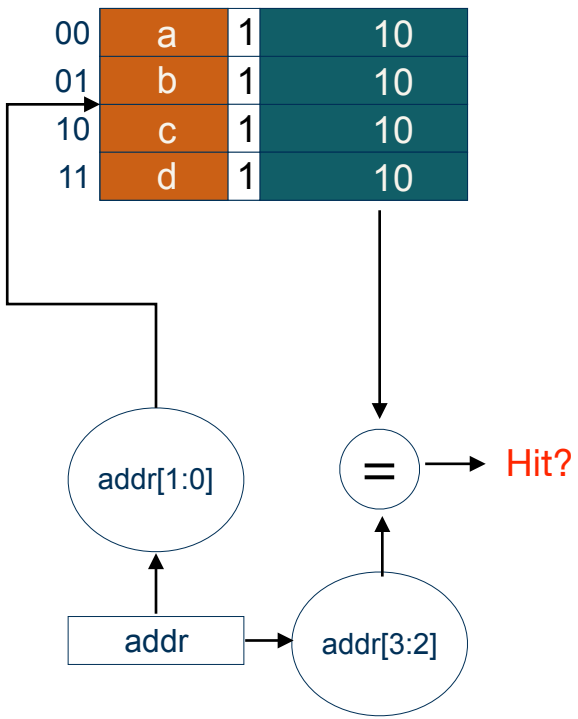
Associative verses Direct Mapped Trade-offs

Associative verses Direct Mapped Trade-offs

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

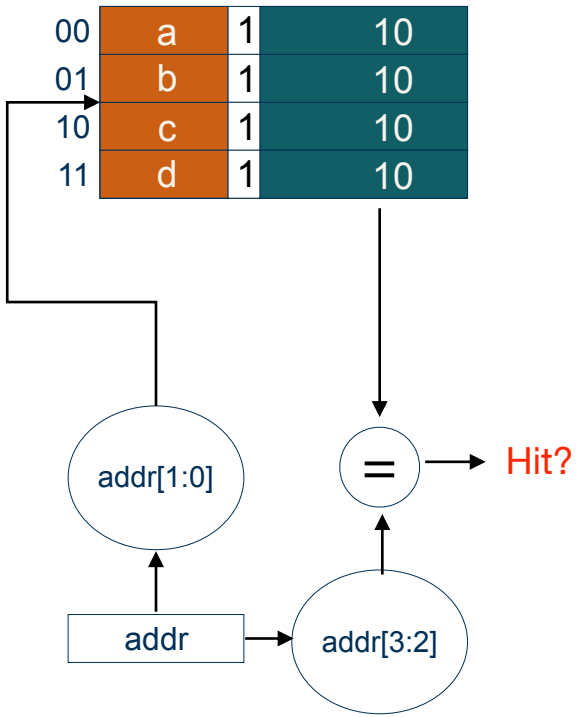
Associative verses Direct Mapped Trade-offs

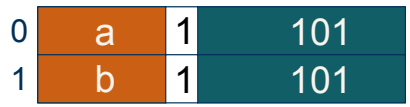
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster

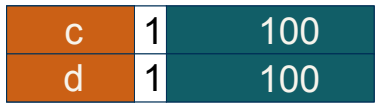


Associative verses Direct Mapped Trade-offs

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

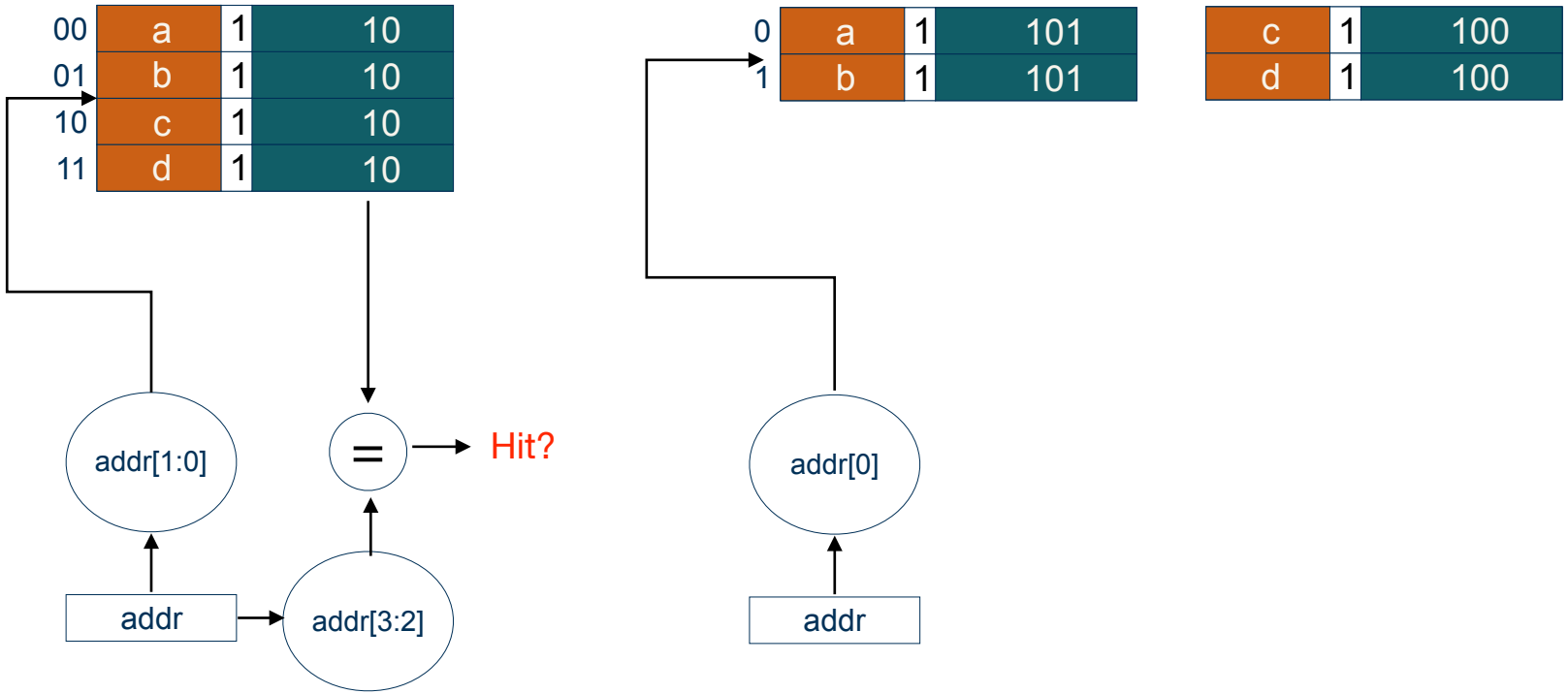






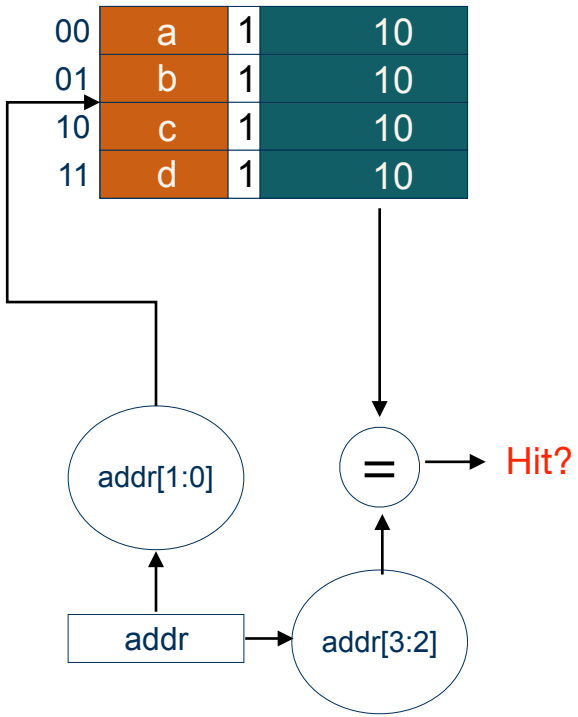
Associative verses Direct Mapped Trade-offs

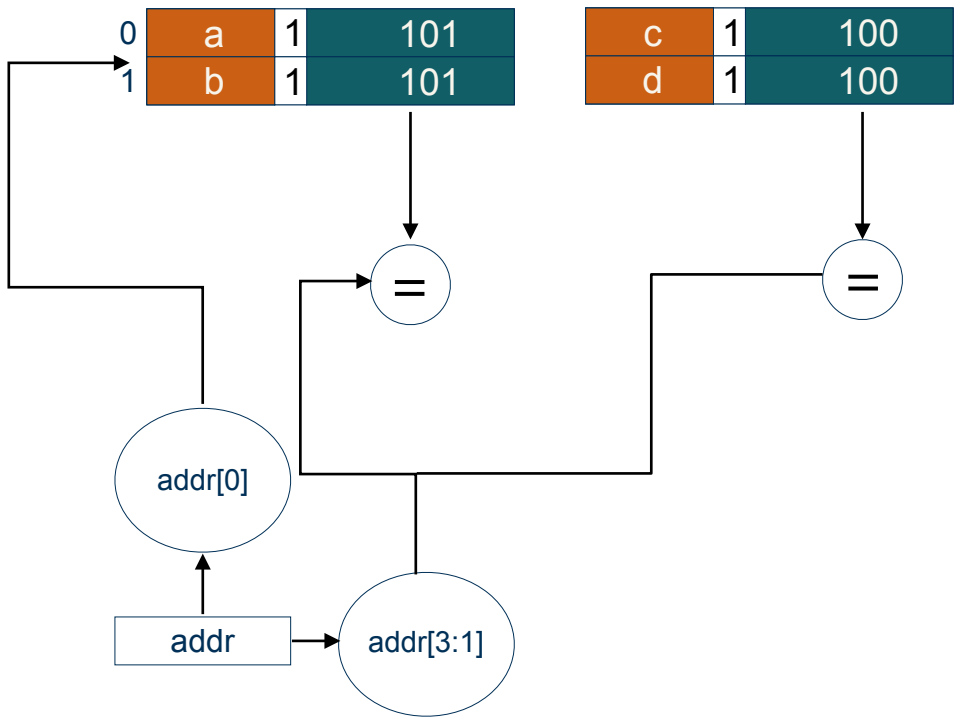
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower. Why?



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. 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. Why?