CSC 252: Computer Organization
Spring 2018: Lecture 16

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
- **Programming Assignment 4 is out**

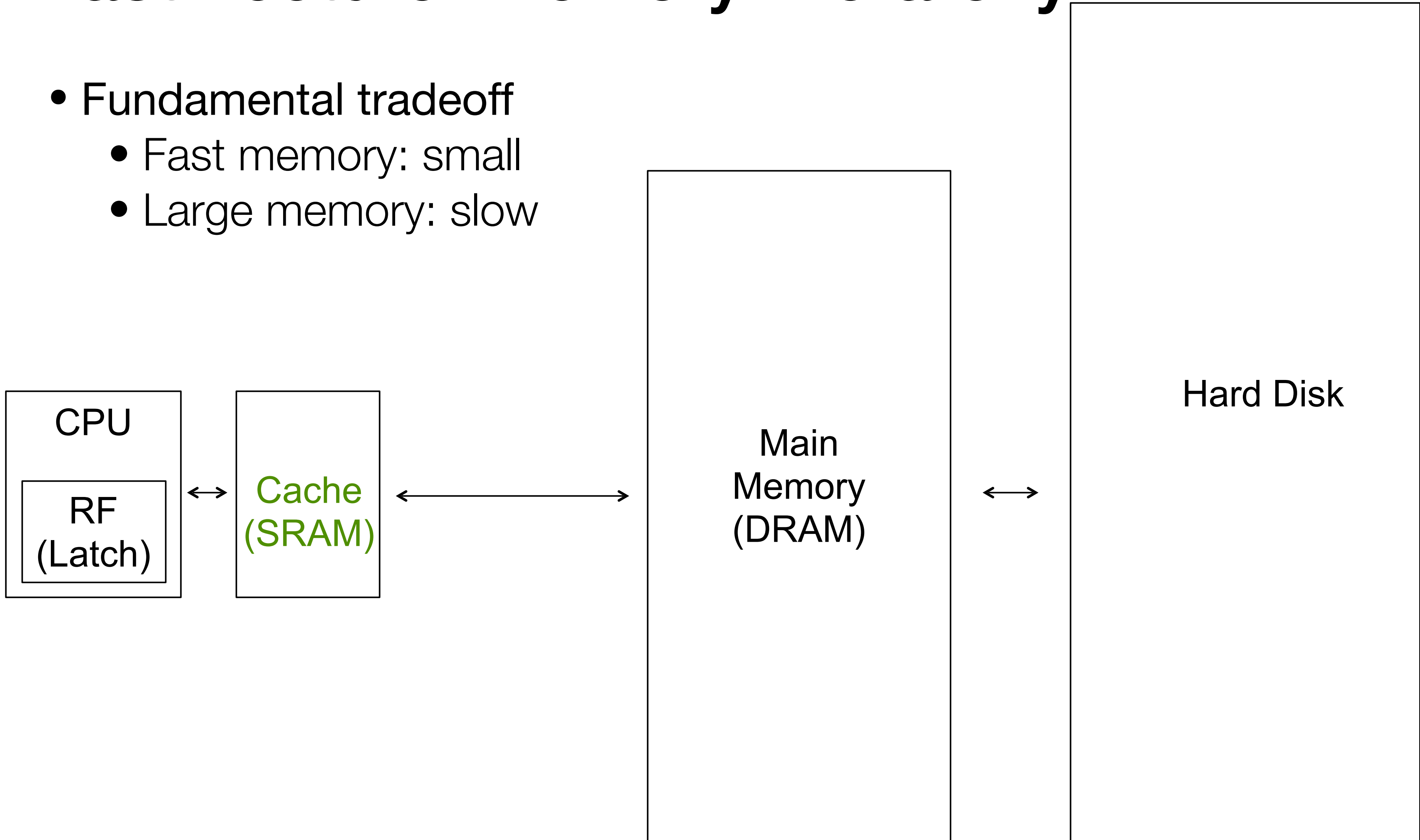
Announcement

- Programming Assignment 4 is out
  - Main assignment due on 11:59pm, **Monday, April 2**.

18	19	20	21	22	23	24
25	26	27	28	29	30	31
Sun Apr 1	Mon 2	Tue 3 Due	Wed 4	Thu 5	Fri 6	Sat 7

Last Lecture: Memory Hierarchy

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



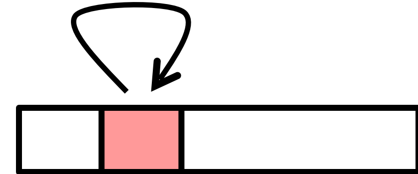
Locality

- 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

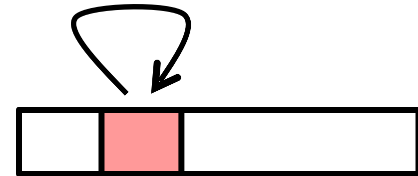
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



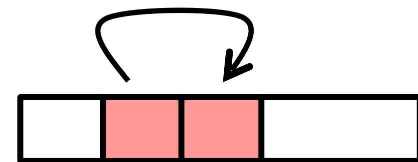
Locality

- 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

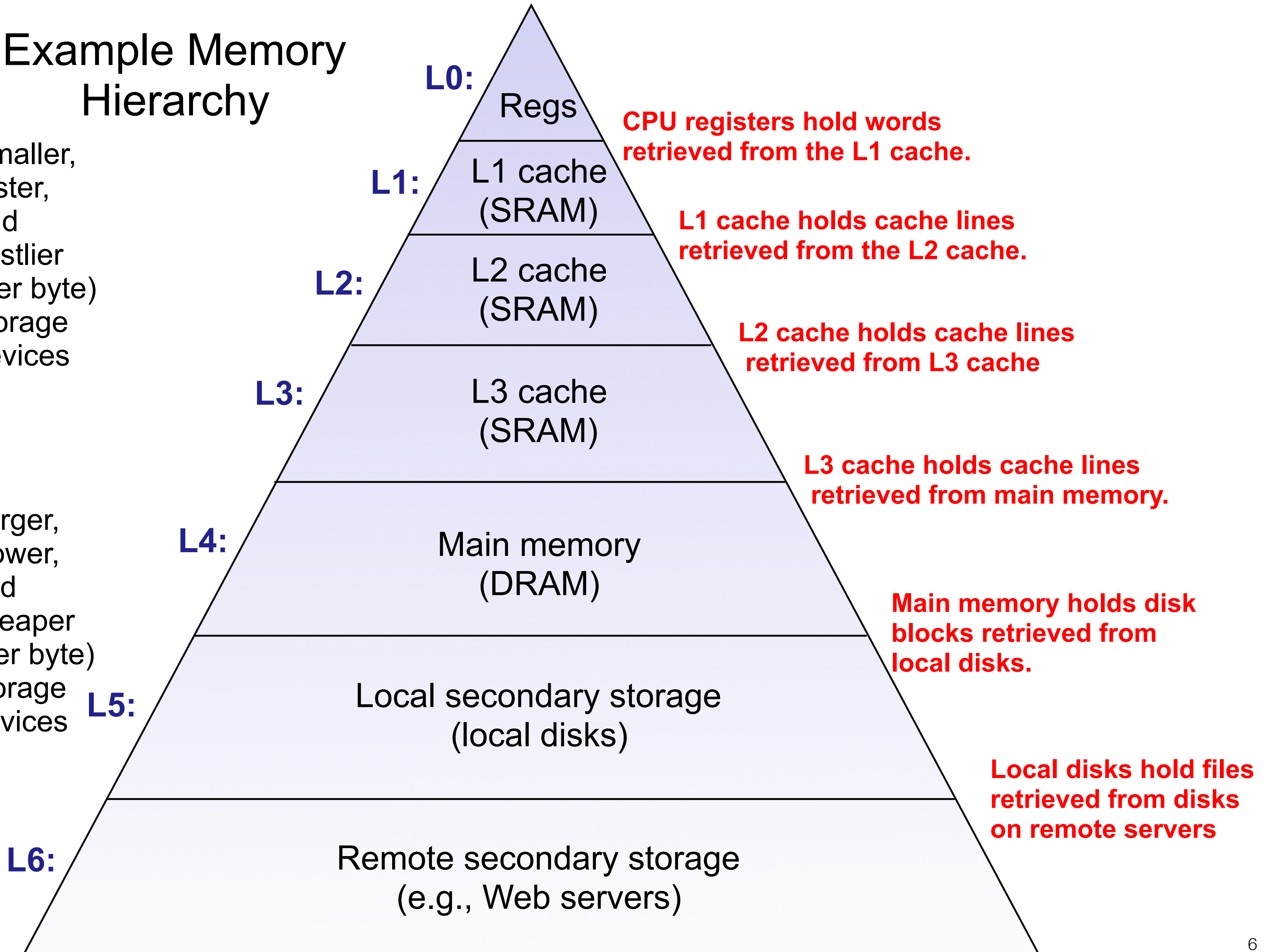


Locality Example

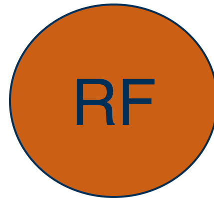
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

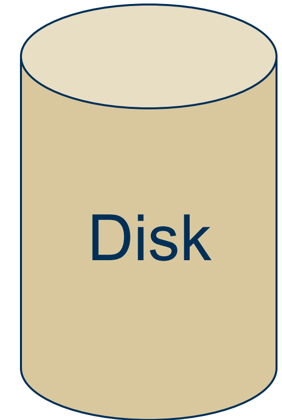
- Data references
  - **Spatial** Locality: Reference array elements in succession (stride-1 reference pattern)
  - **Temporal** Locality: Reference variable sum each iteration.

- Instruction references
  - **Spatial** Locality: Reference instructions in sequence.
  - **Temporal** Locality: Cycle through loop repeatedly.


Example Memory Hierarchy
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers

How Things Have Progressed

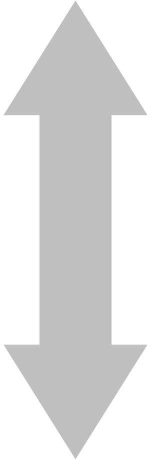




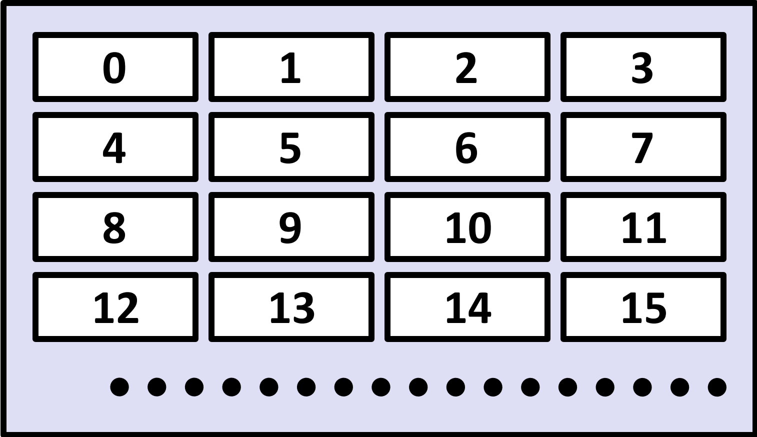
	RF	Cache $	Memory	Disk
1995 low-mid range Hennessy & Patterson, Computer Arch., 1996	200B 5ns	64KB 10ns	32MB 100ns	2GB 5ms
2009 low-mid range www.dell.com, $449 including 17" LCD flat panel	~200B 0.33ns	8MB 0.33ns	4GB <100ns	750GB 4ms
2015 mid range	~200B 0.33ns	8MB 0.33ns	16GB <100ns	**256GB** **10us**

Cache Illustrations

CPU

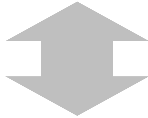


Memory
(big but slow)

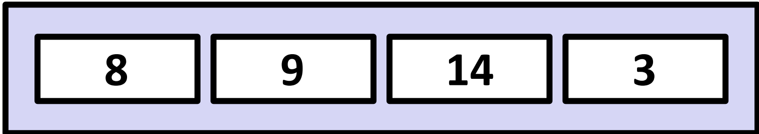


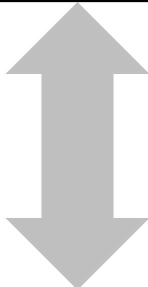
Cache Illustrations

CPU

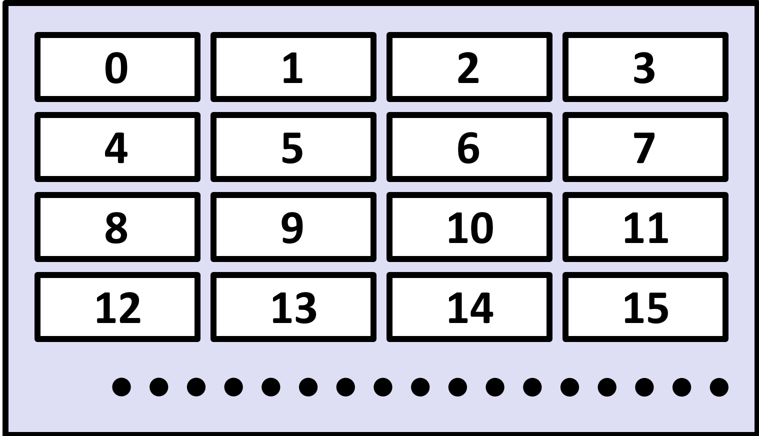


Cache
(small but fast)



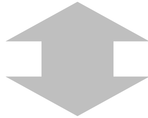


Memory
(big but slow)

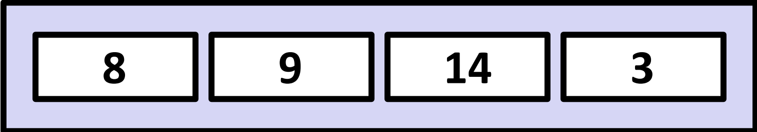


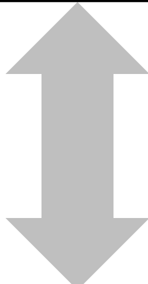
Cache Illustrations

CPU

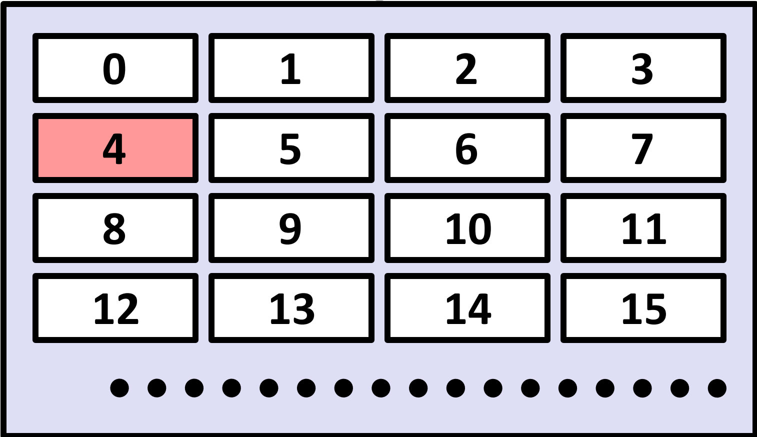


Cache
(small but fast)



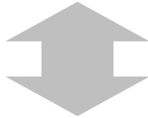


Memory
(big but slow)

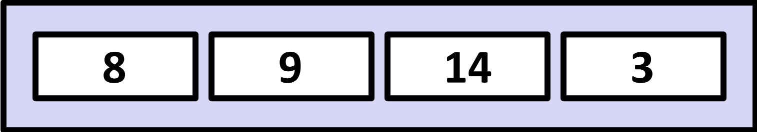


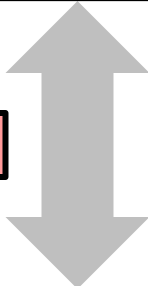
Cache Illustrations

CPU

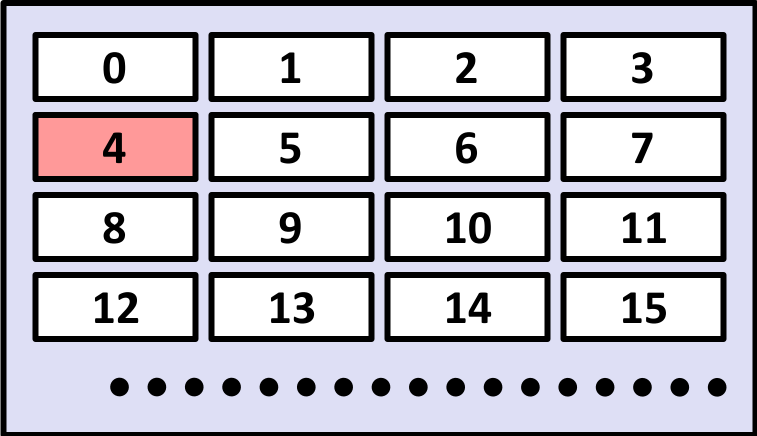


Cache
(small but fast)



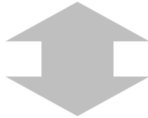


Memory
(big but slow)

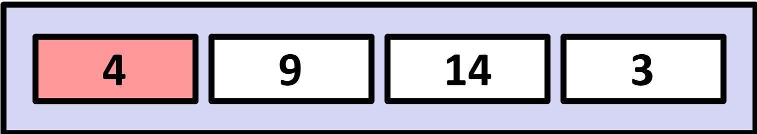


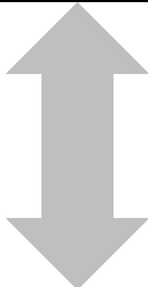
Cache Illustrations

CPU

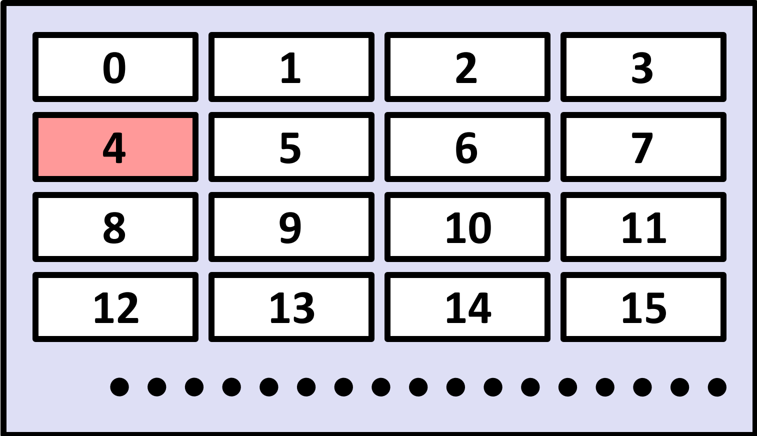


Cache
(small but fast)



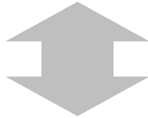


Memory
(big but slow)

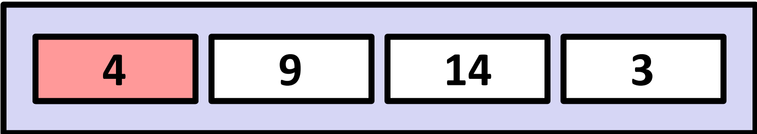


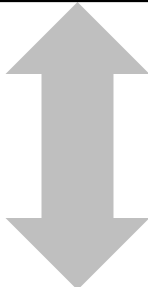
Cache Illustrations

CPU

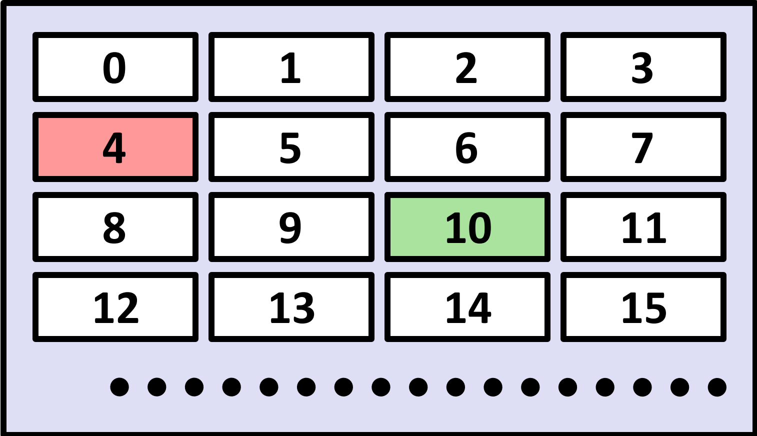


Cache
(small but fast)



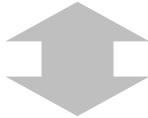


Memory
(big but slow)

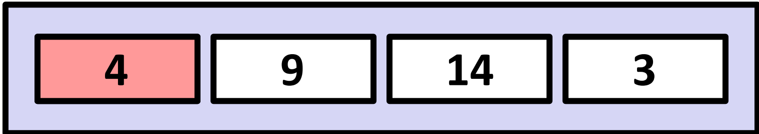


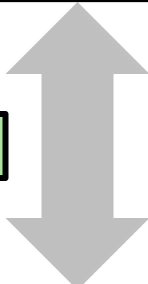
Cache Illustrations

CPU

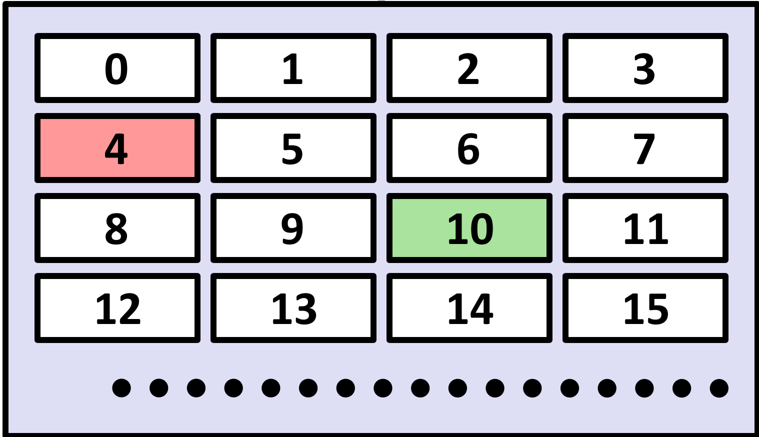


Cache
(small but fast)



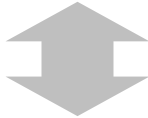


Memory
(big but slow)

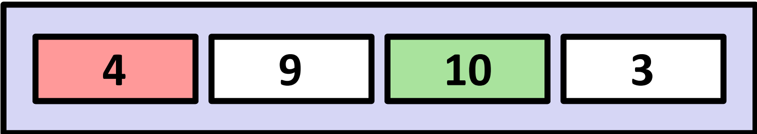


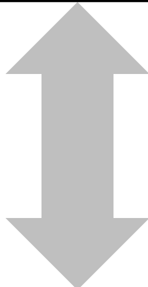
Cache Illustrations

CPU

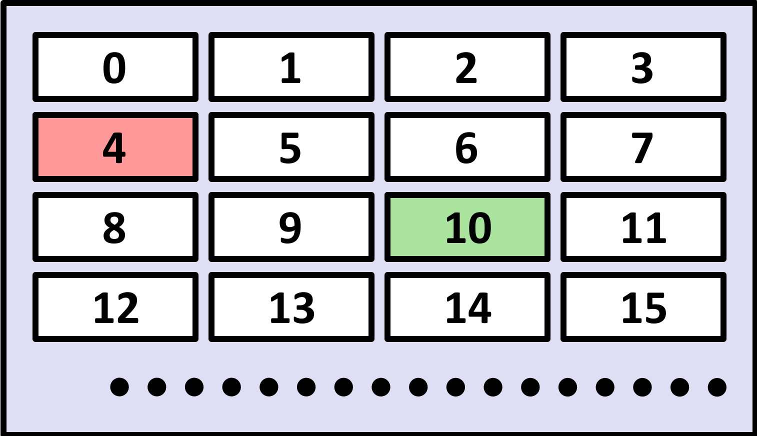


Cache
(small but fast)



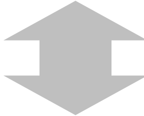


Memory
(big but slow)

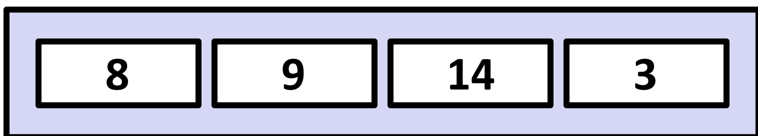


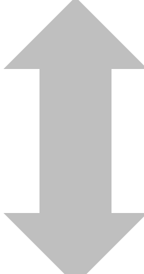
Cache Illustrations: Hit

CPU

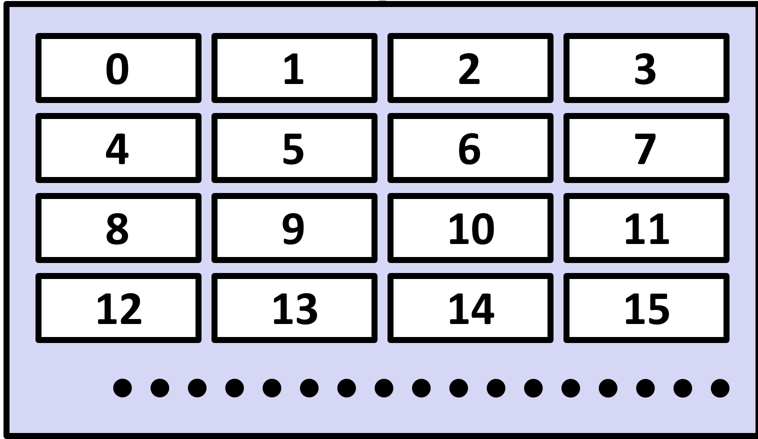


Cache
(small but fast)





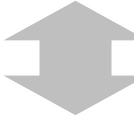
Memory
(big but slow)



Cache Illustrations: Hit

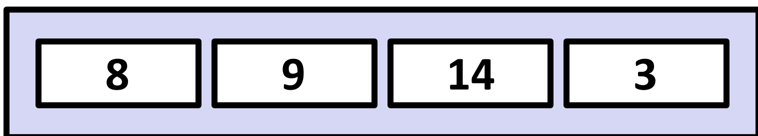
CPU

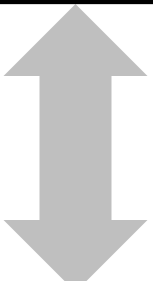
Data in address b is needed



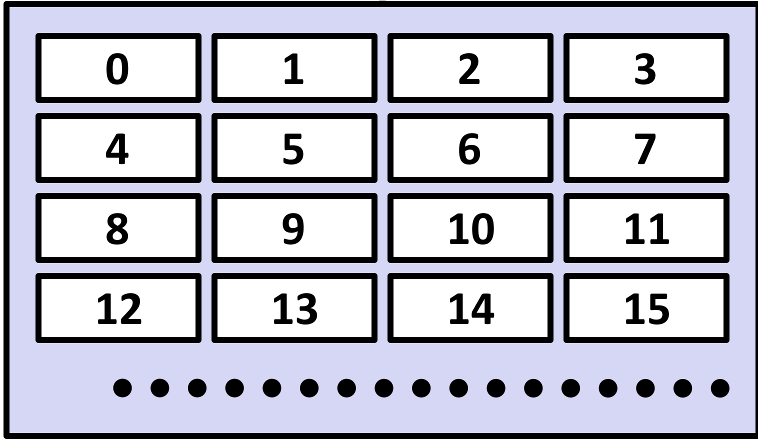
Request: 14

Cache
(small but fast)



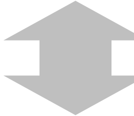


Memory
(big but slow)

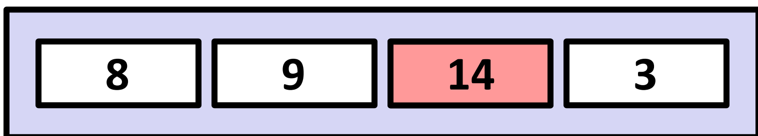


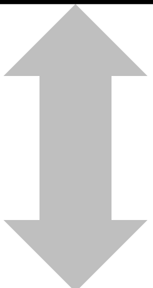
Cache Illustrations: Hit

CPU

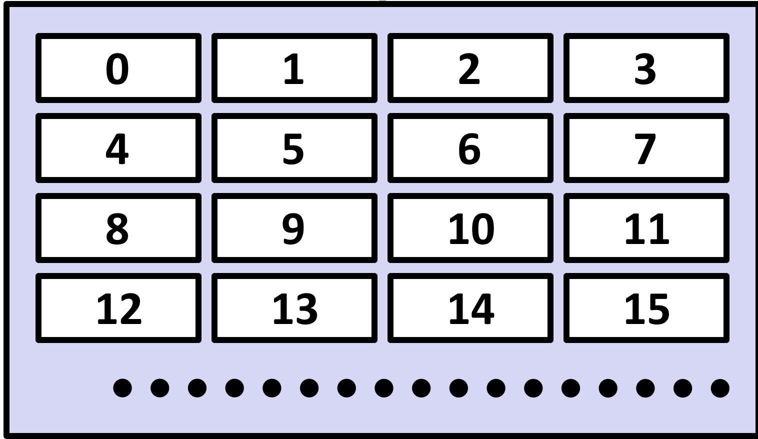


Cache
(small but fast)





Memory
(big but slow)

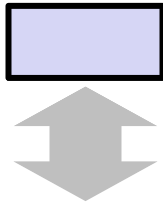


Data in address b is needed

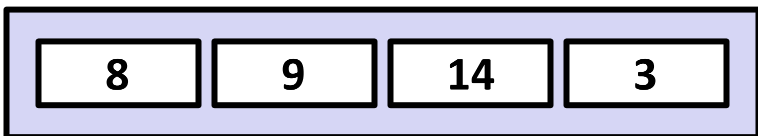
Address b is in cache: Hit!

Cache Illustrations: Miss

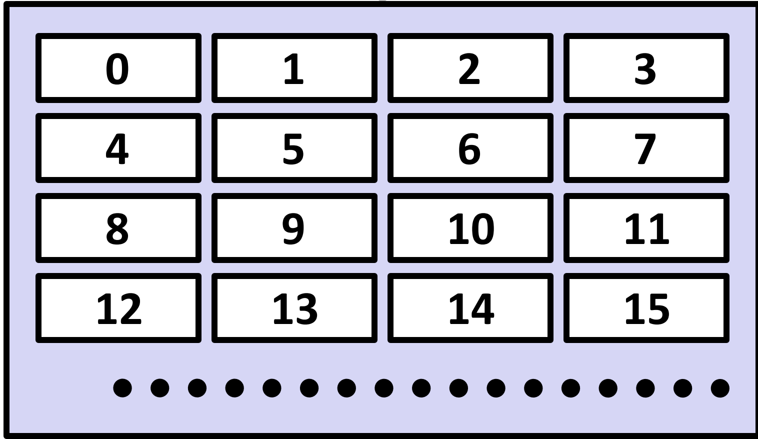
CPU



Cache
(small but fast)

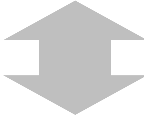


Memory
(big but slow)



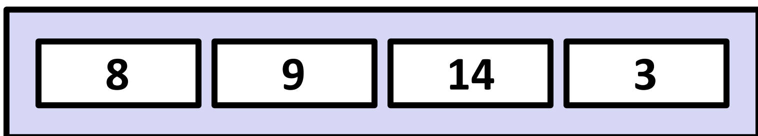
Cache Illustrations: Miss

CPU

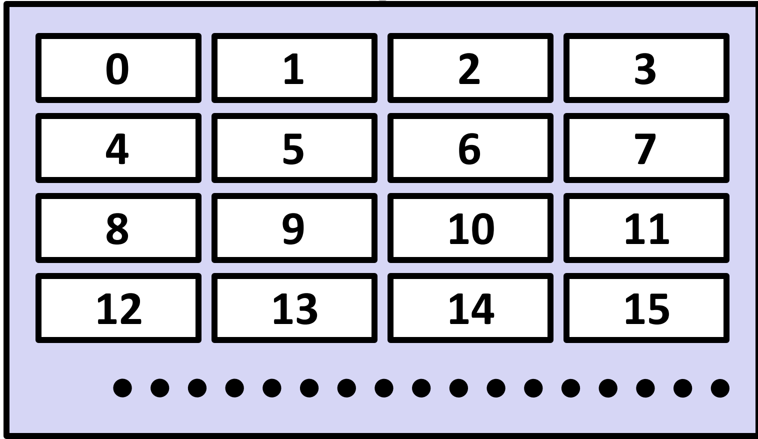


Request: 12

Cache
(small but fast)

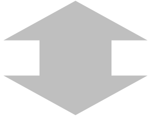


Memory
(big but slow)

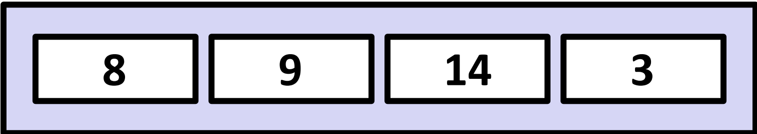


Cache Illustrations: Miss

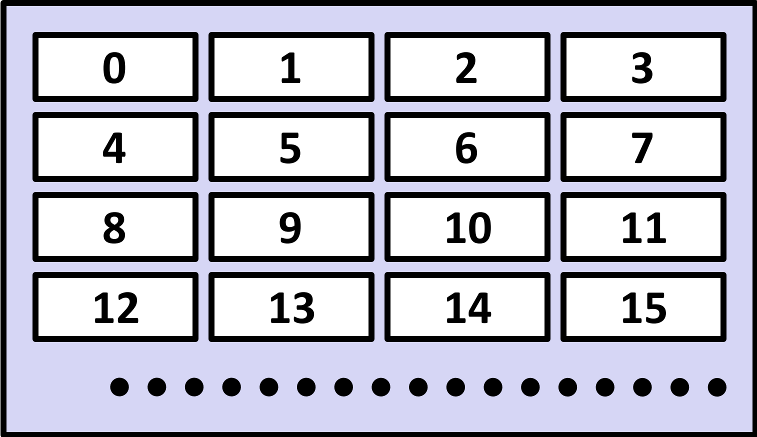
CPU



Cache
(small but fast)



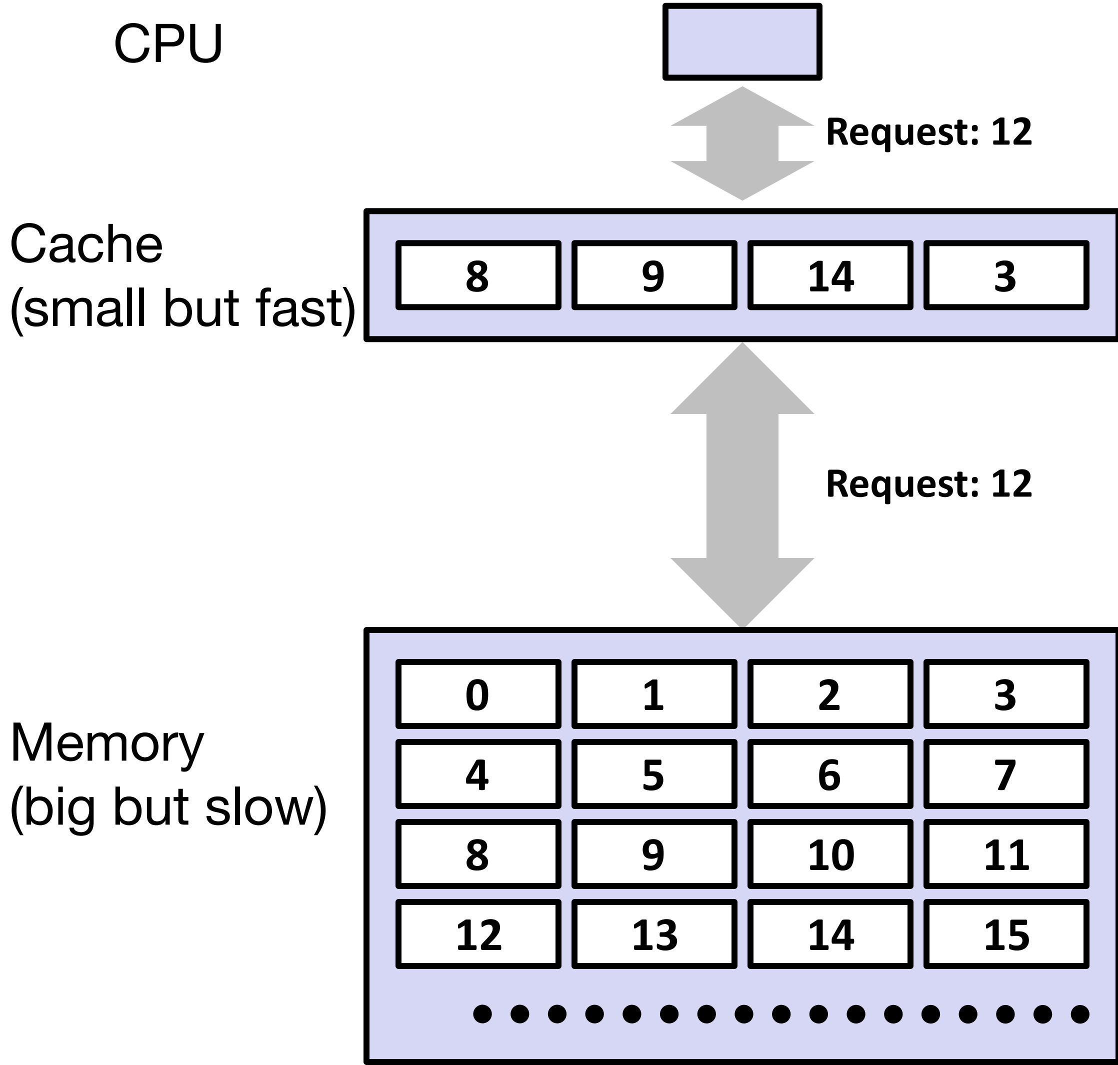
Memory
(big but slow)



Data in address b is needed

Address b is not in cache: Miss!

Cache Illustrations: Miss

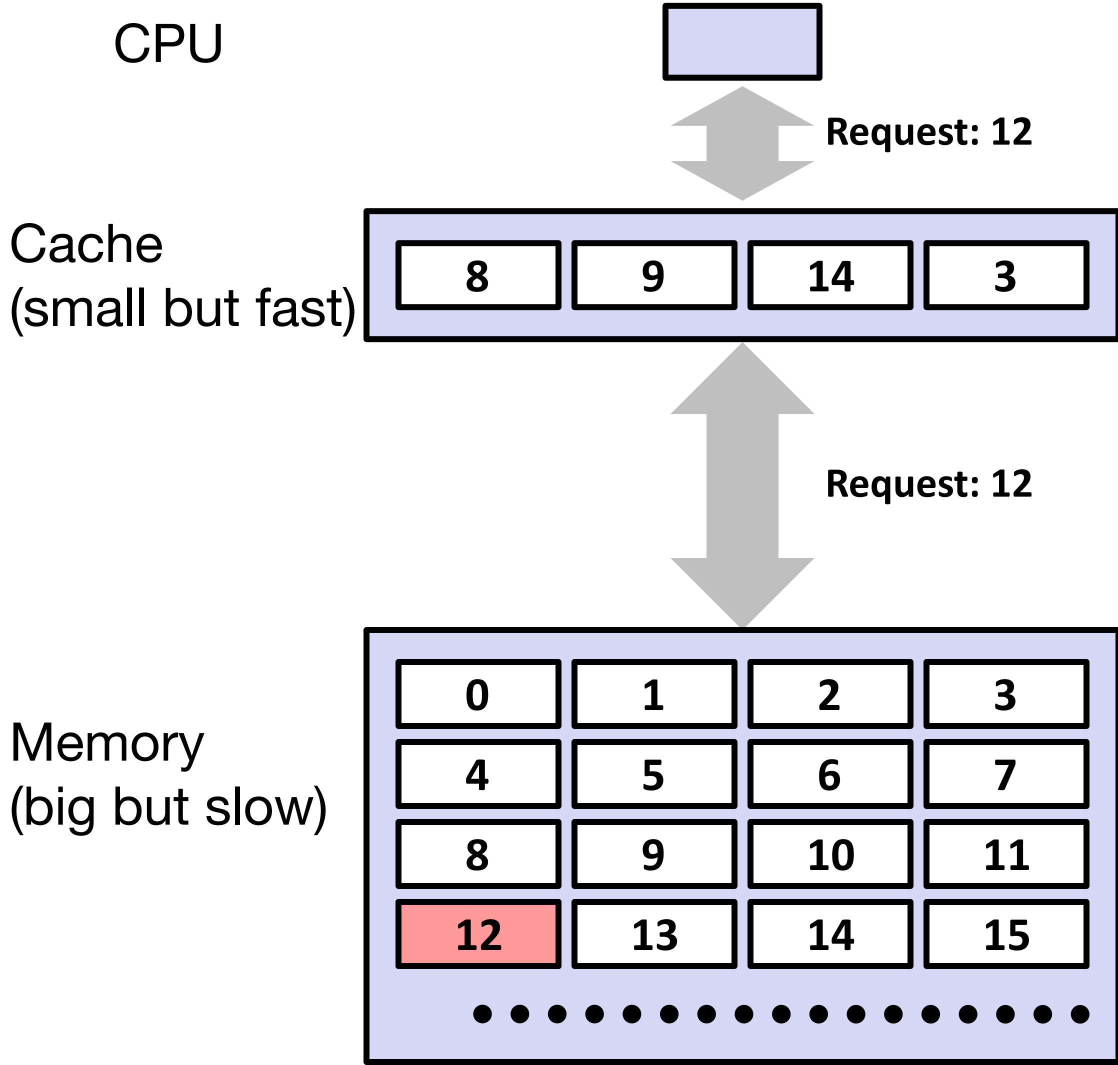


Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Cache Illustrations: Miss



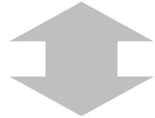
Data in address b is needed

Address b is not in cache: Miss!

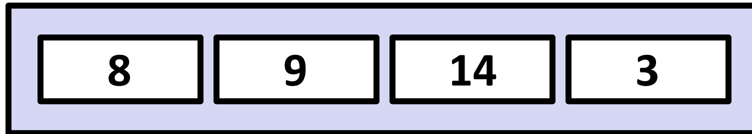
Address b is fetched from memory

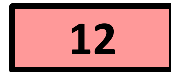
Cache Illustrations: Miss

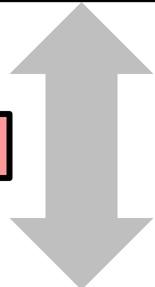
CPU



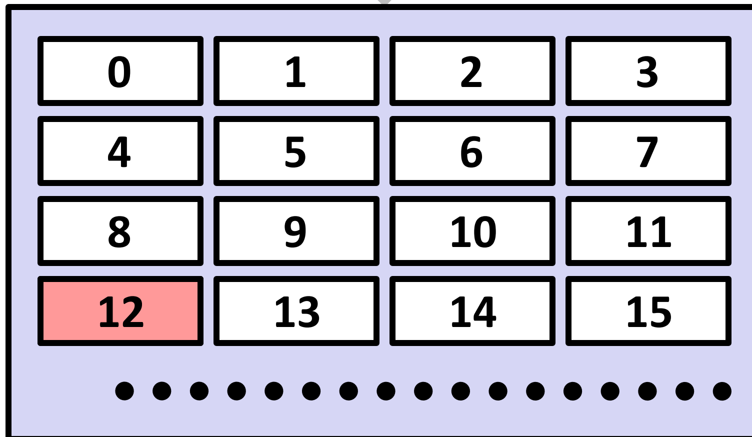
Cache
(small but fast)







Memory
(big but slow)



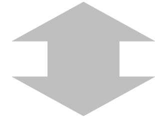
Data in address b is needed

Address b is not in cache: Miss!

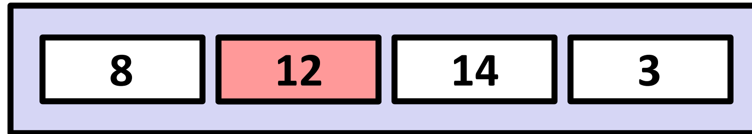
Address b is fetched from memory

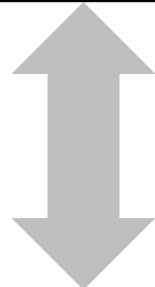
Cache Illustrations: Miss

CPU

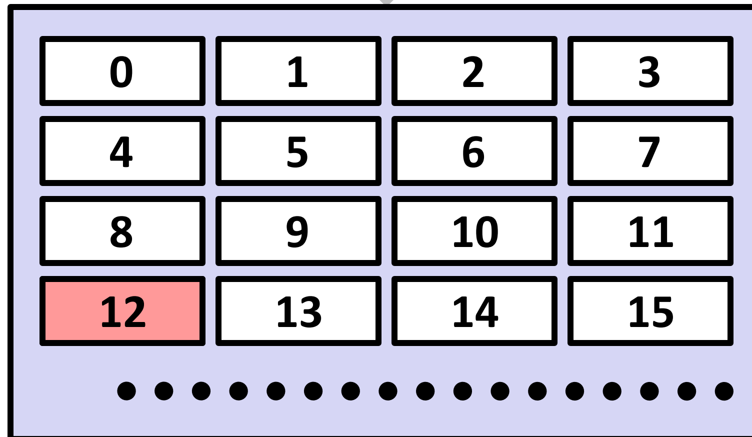


Cache
(small but fast)





Memory
(big but slow)



Data in address b is needed

Address b is not in cache: Miss!

Address b is fetched from memory

Address b is stored in cache

Cache Hit Rate

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

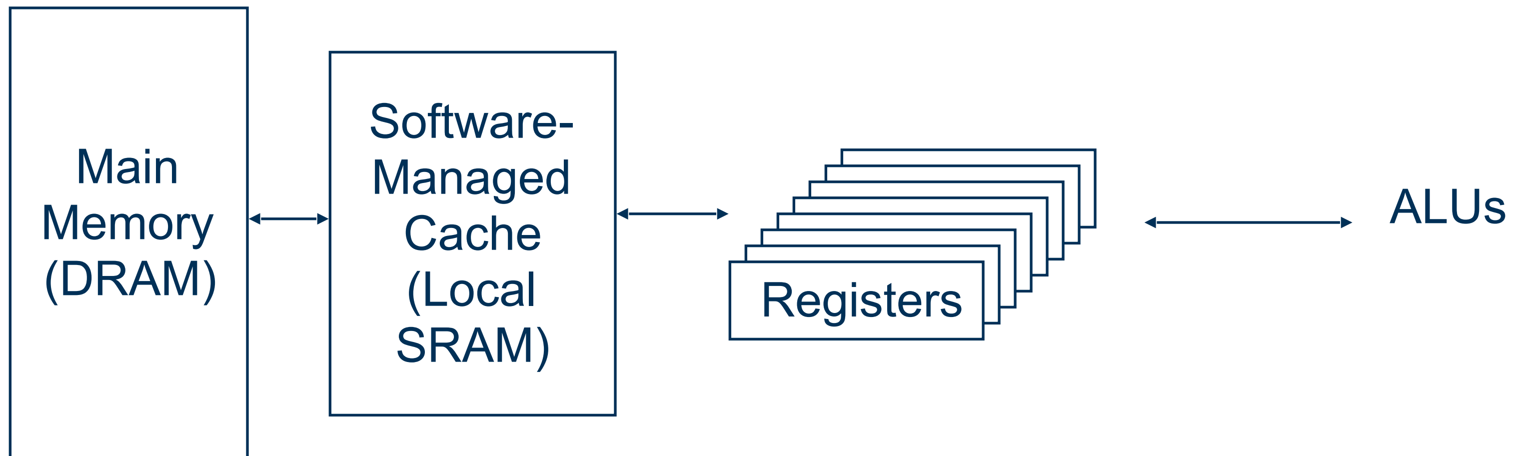
$$\text{Hit Rate} = \frac{\text{\# Hits}}{\text{\# Accesses}}$$

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?

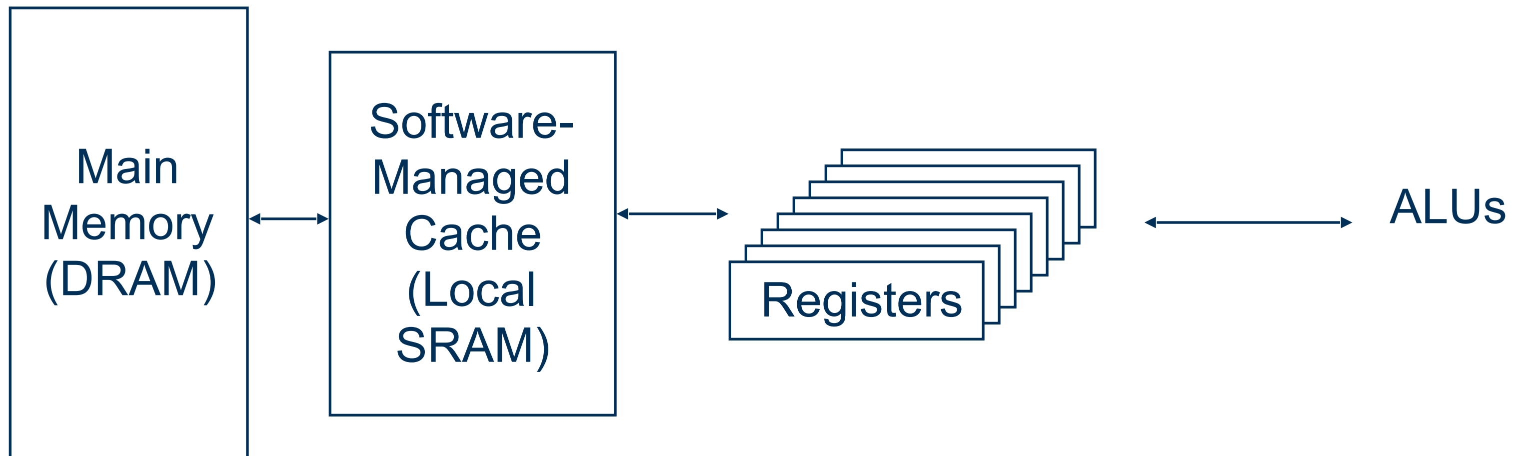
Cache Management: Explicit

- 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



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



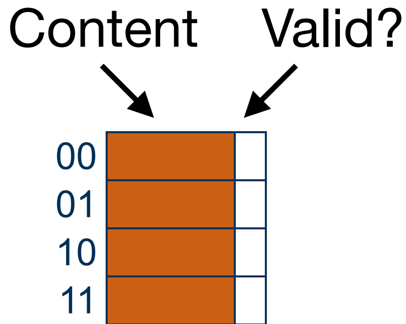
- Examples of software-managed cache
  - Sony Cell Broadband Engine (PS3): **Local store**
  - DSPs: **Scratchpad memory**
  - GPUs: “**Shared memory**”
  - Stream Processors: **Stream register file**

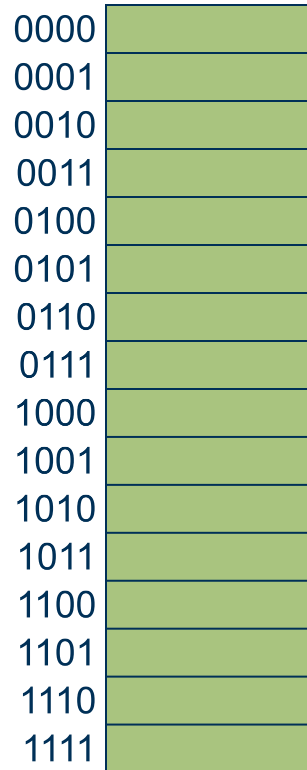
Want Automatic Management

- 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



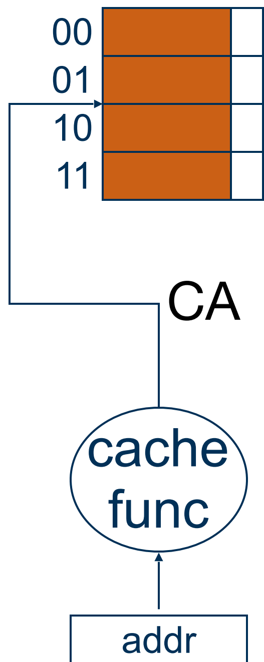


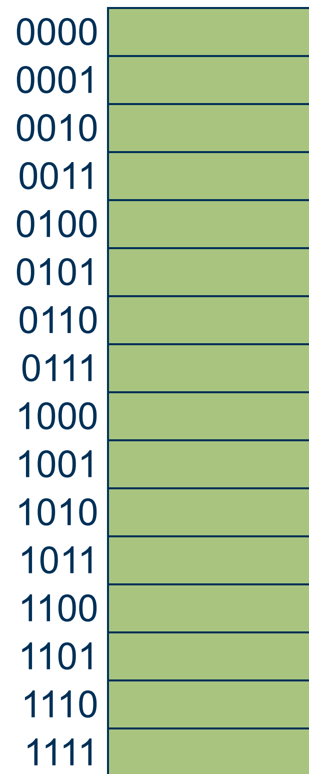
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit
- 16 memory locations
- For now, assume cache location size == memory location size
- Assume each memory location can only reside in one cache-line
- Cache is smaller than memory (obviously)
  - Thus, not all memory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

Cache Placement

Cache

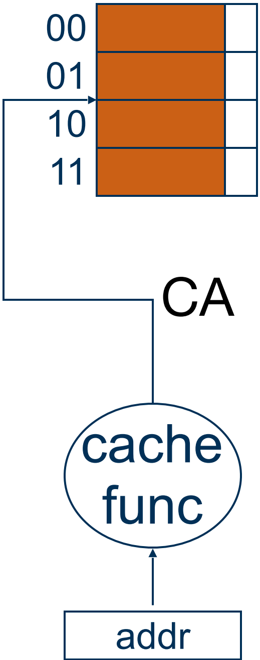
Memory



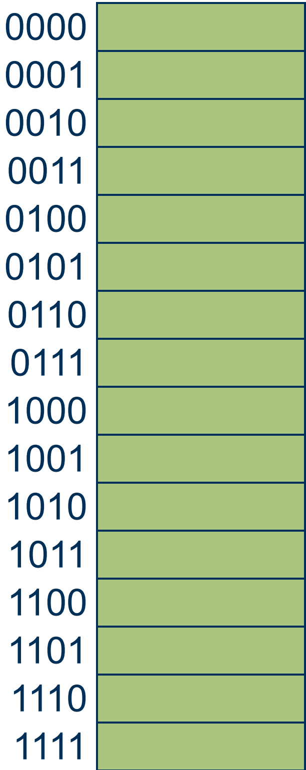


Cache Placement

Cache

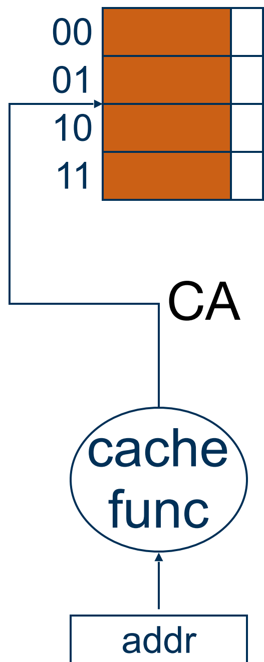


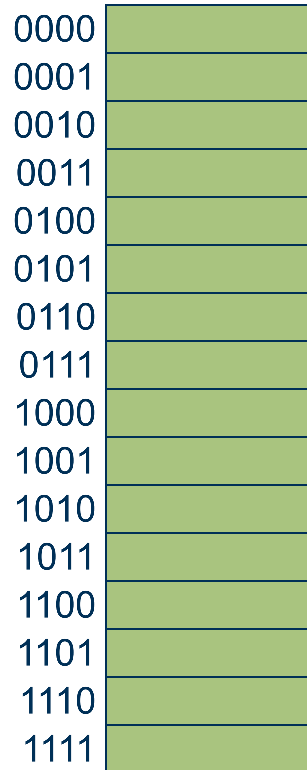
Memory



- Use memory address as a name
- Apply a function to the name to generate a cache address to access
- What are reasonable functions?

Function to Address Cache



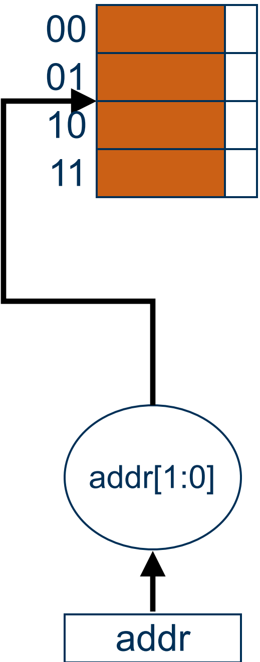


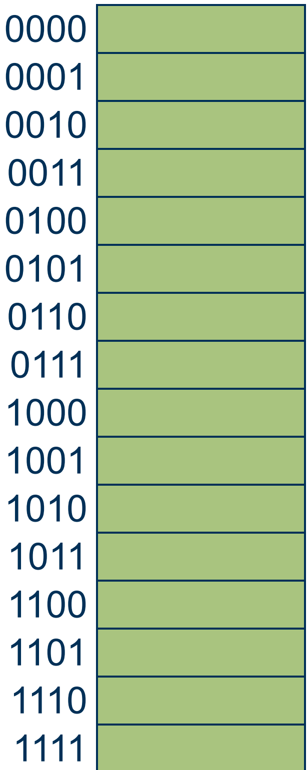
- Simplest function is a subset of address bits
- Six combinations in total
  - CA = ADDR[3],ADDR[2]
  - CA = ADDR[3],ADDR[1]
  - CA = ADDR[3],ADDR[0]
  - CA = ADDR[2],ADDR[1]
  - CA = ADDR[2],ADDR[0]
  - CA = ADDR[1],ADDR[0]
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]

Direct-Mapped Cache

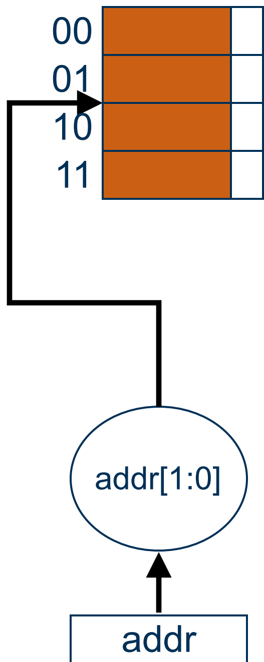
Cache

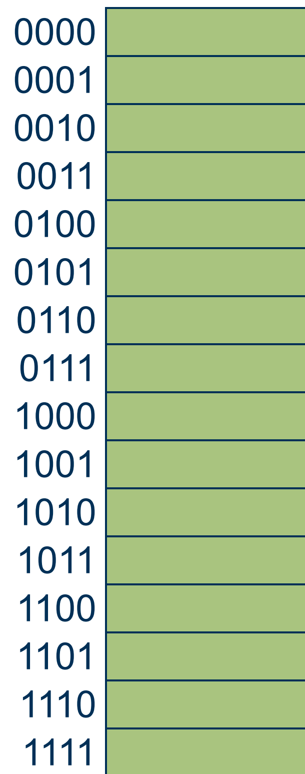
Memory





Direct-Mapped Cache

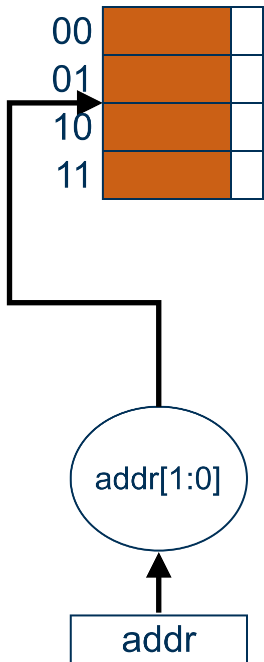




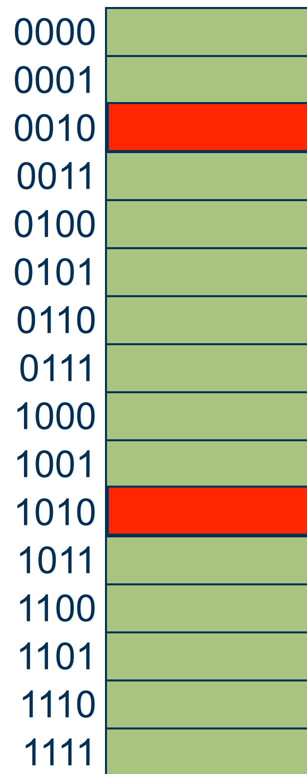
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits

Direct-Mapped Cache

Cache



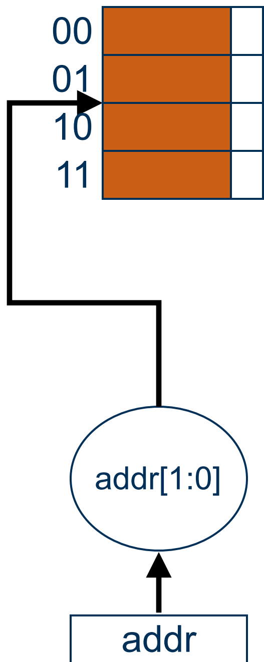
Memory



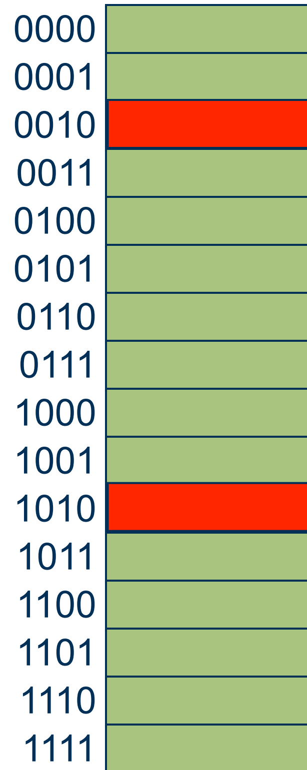
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010

Direct-Mapped Cache

Cache

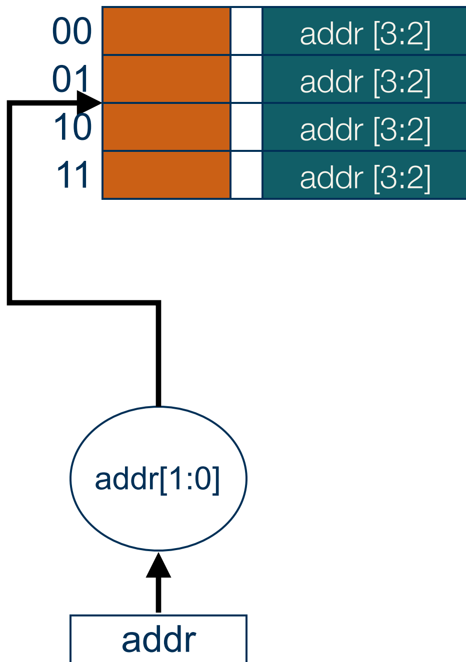


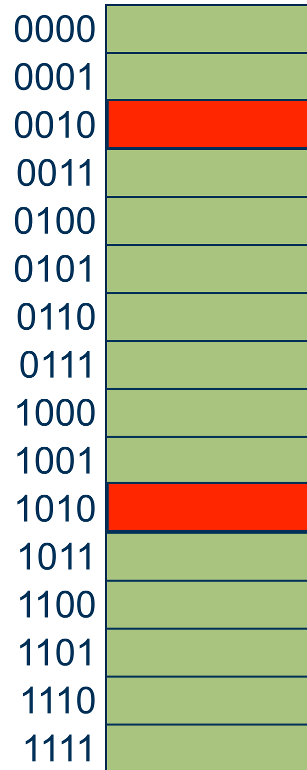
Memory



- 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

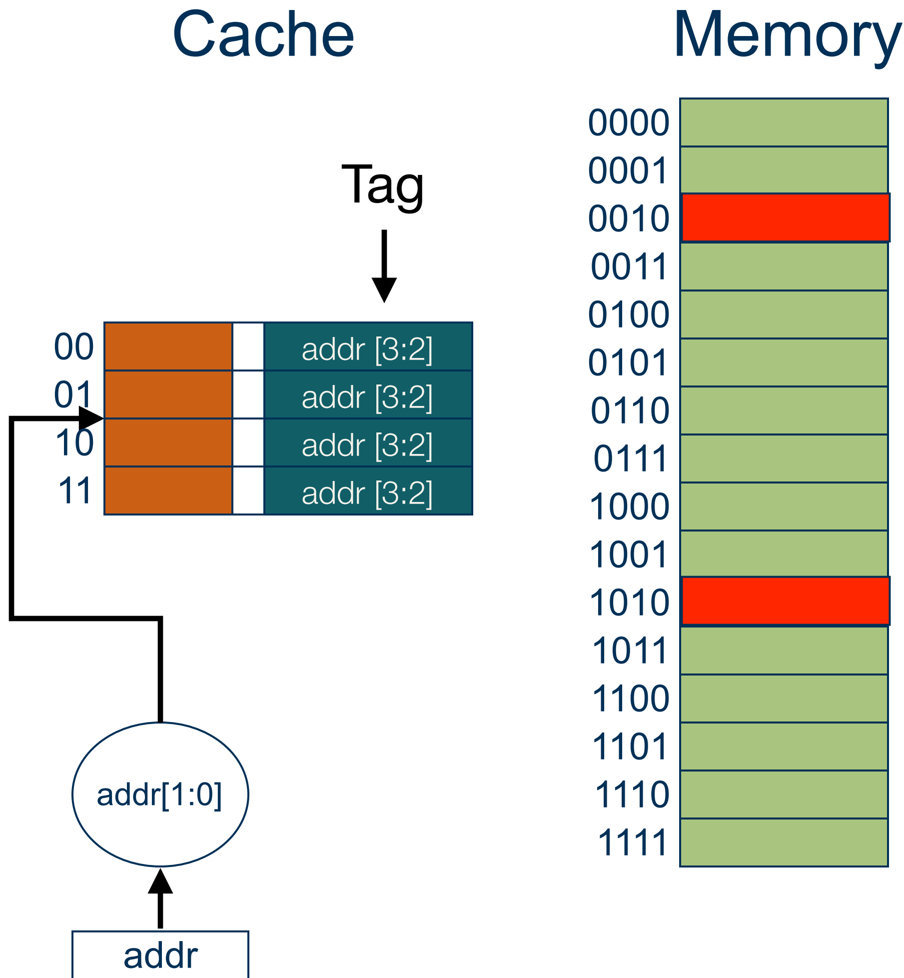
Direct-Mapped Cache





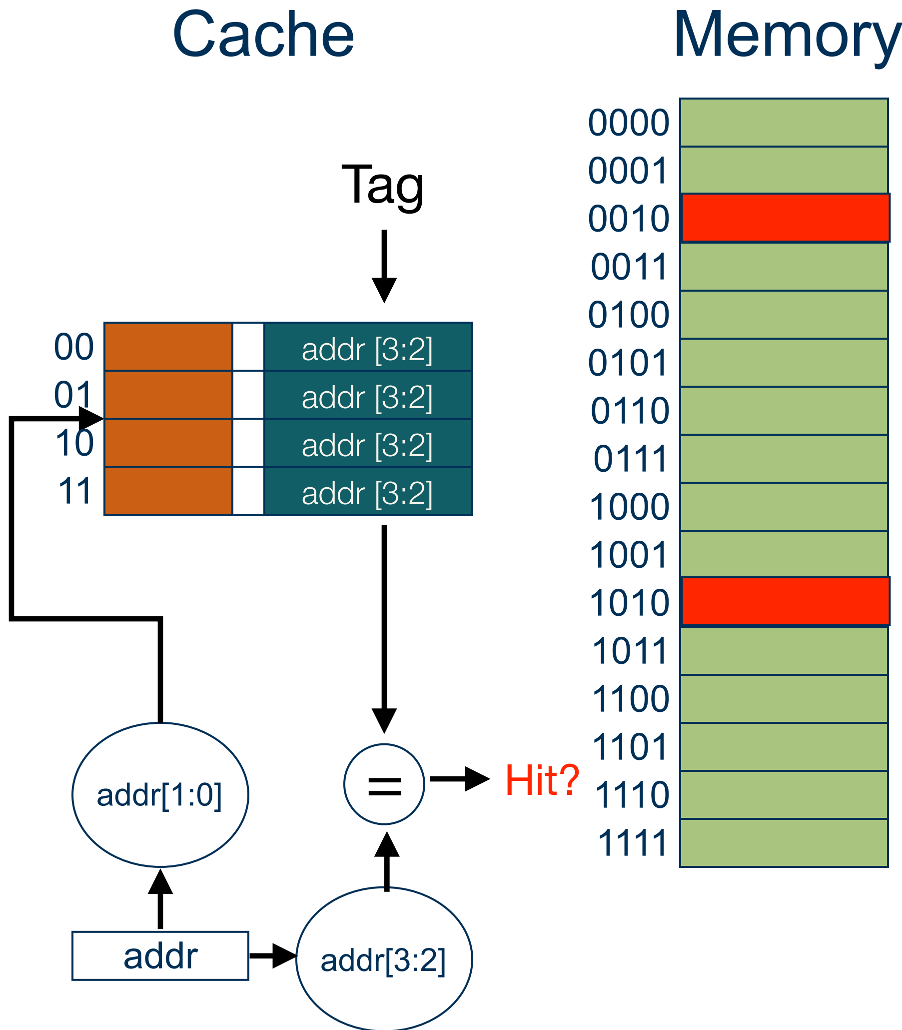
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

Direct-Mapped Cache



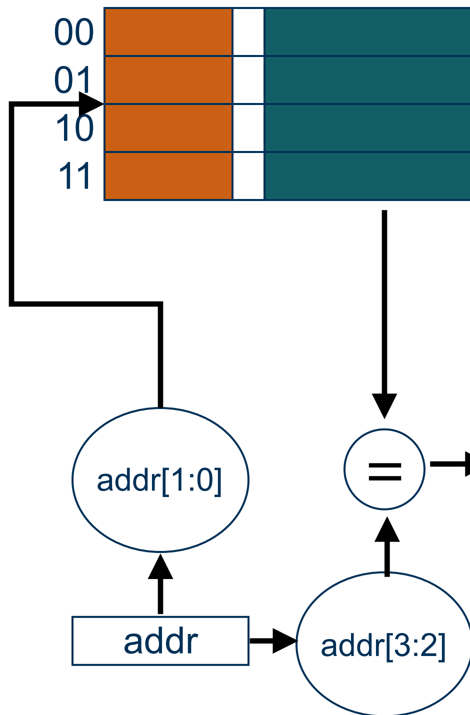
- Direct-Mapped Cache
  - CA = ADDR[1],ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that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

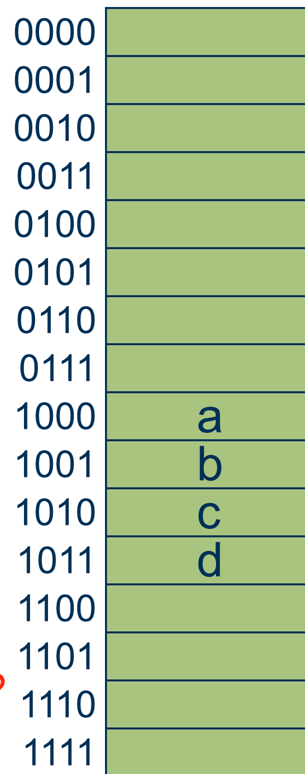
Direct-Mapped Cache



- 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



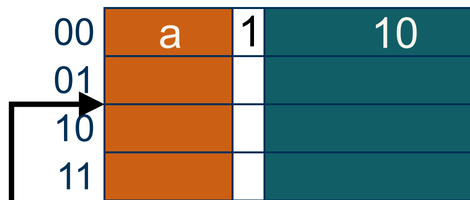


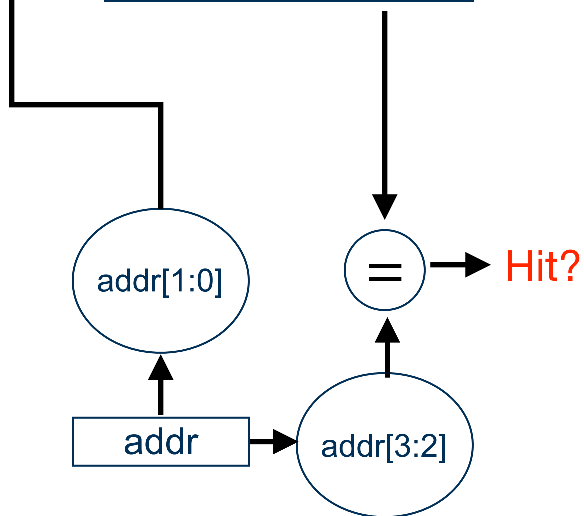

```
for (i = 0; i < 4; ++i) {
    A += mem[i];
}
for (i = 0; i < 4; ++i) {
    B *= (mem[i] + A);
}
```

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

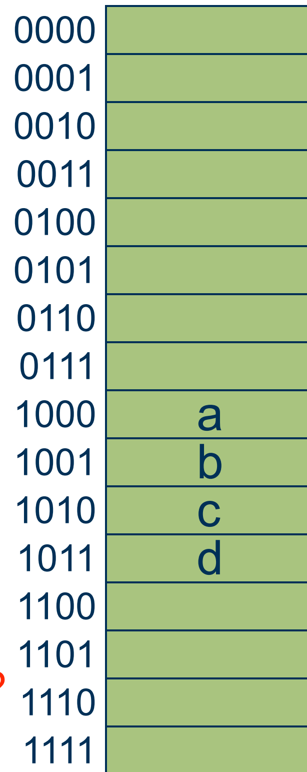
Example: Direct-Mapped Cache

Cache

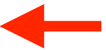




Memory

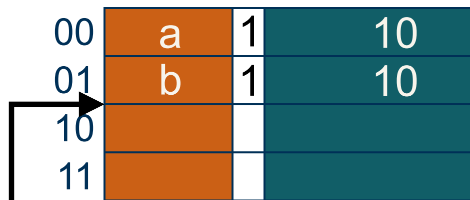


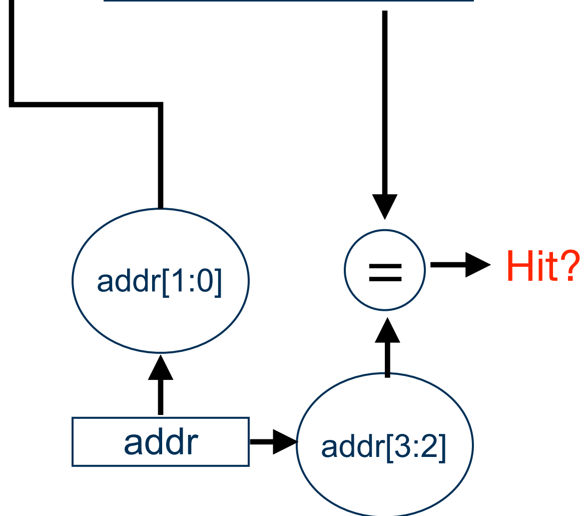

```
for (i = 0; i < 4; ++i) {
    A += mem[i];
}
for (i = 0; i < 4; ++i) {
    B *= (mem[i] + A);
}
```

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

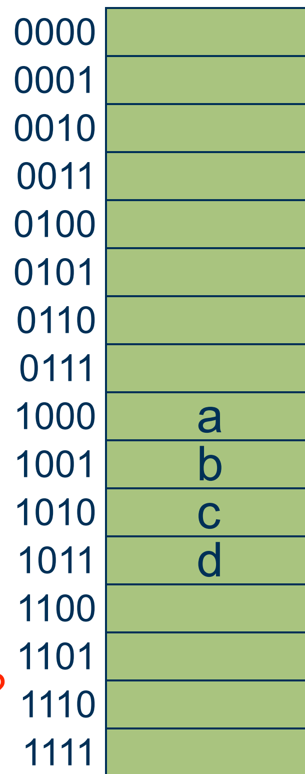
Example: Direct-Mapped Cache

Cache

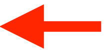




Memory

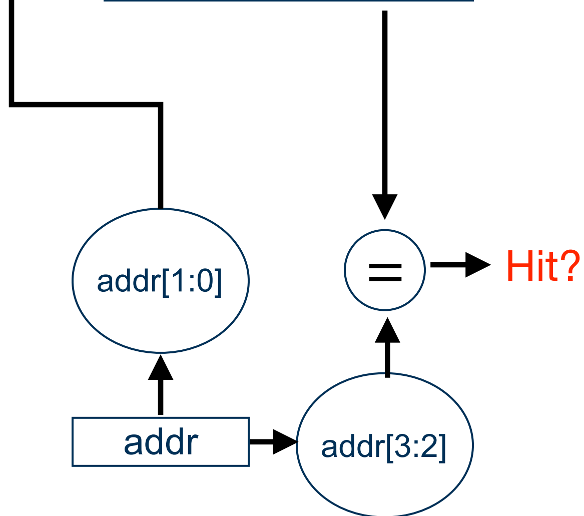



```
for (i = 0; i < 4; ++i) {
    A += mem[i];
}
for (i = 0; i < 4; ++i) {
    B *= (mem[i] + A);
}
```

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

Example: Direct-Mapped Cache

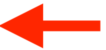
Cache



Memory

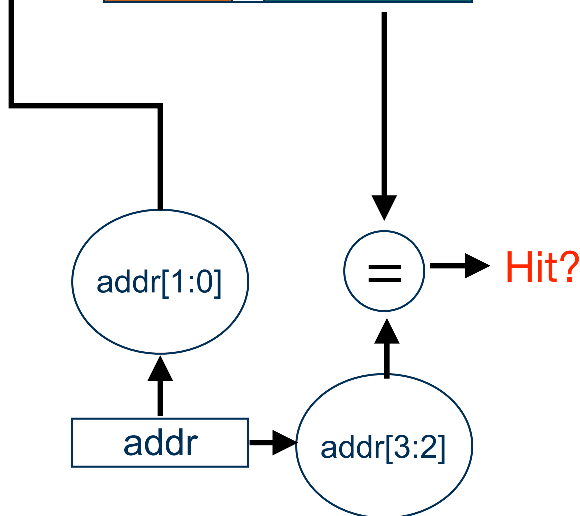
Address	Data
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

```
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

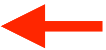
Cache



Memory

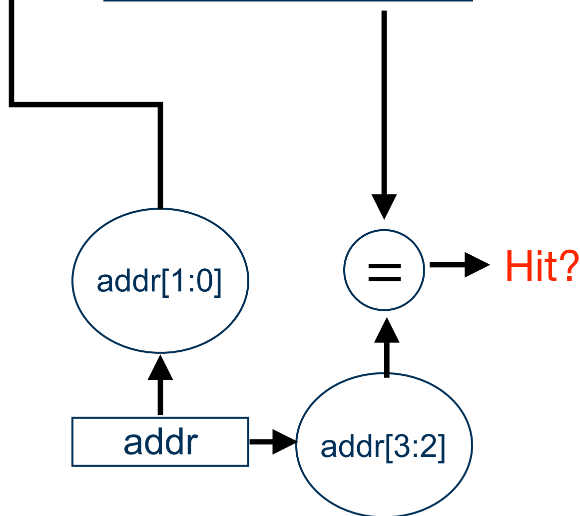
Address	Content
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

```
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

Cache



Memory

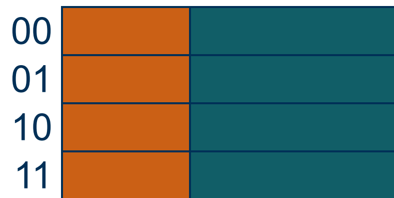
Address	Memory
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	d
1100	
1101	
1110	
1111	

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

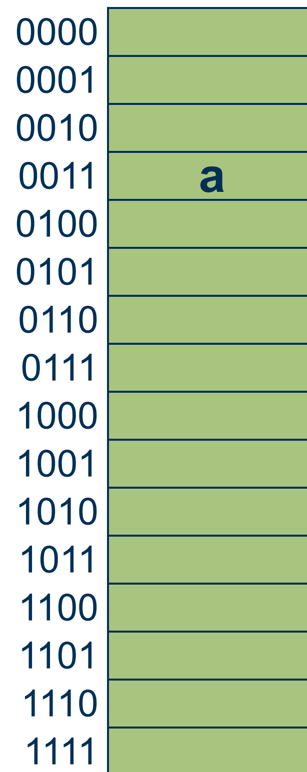
- Assume mem == 0b1000
- Read 0b1000
- Read 0b1001
- Read 0b1010
- Read 0b1011
- Read 0b1000; cache hit? ←

One Possible Direct-Mapped Cache Implementation

Cache



Memory

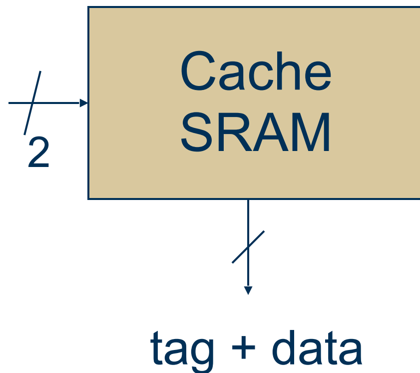


- Cache state is RAM!
- Implement cache as a single SRAM
- Need appropriate comparators

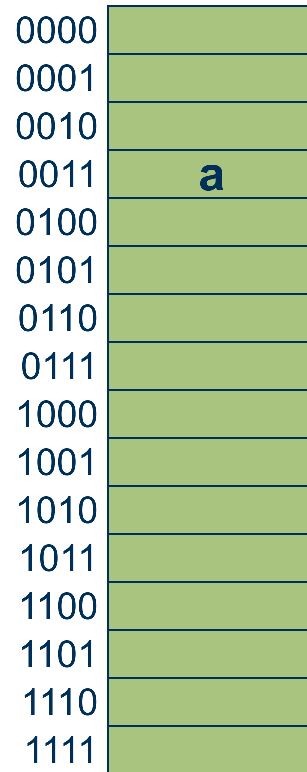
- Memory is implemented as a DRAM

One Possible Direct-Mapped Cache Implementation

Cache



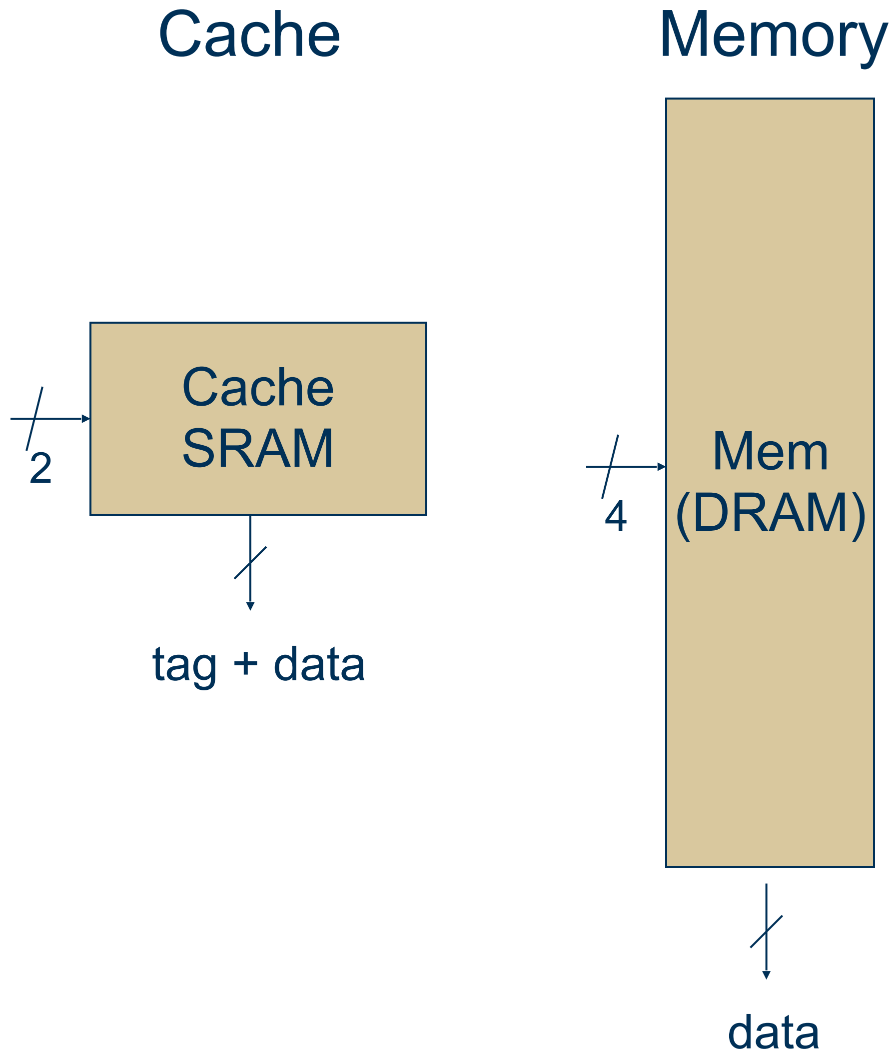
Memory



- Cache state is RAM!
- Implement cache as a single SRAM
- Need appropriate comparators

- Memory is implemented as a DRAM

One Possible Direct-Mapped Cache Implementation



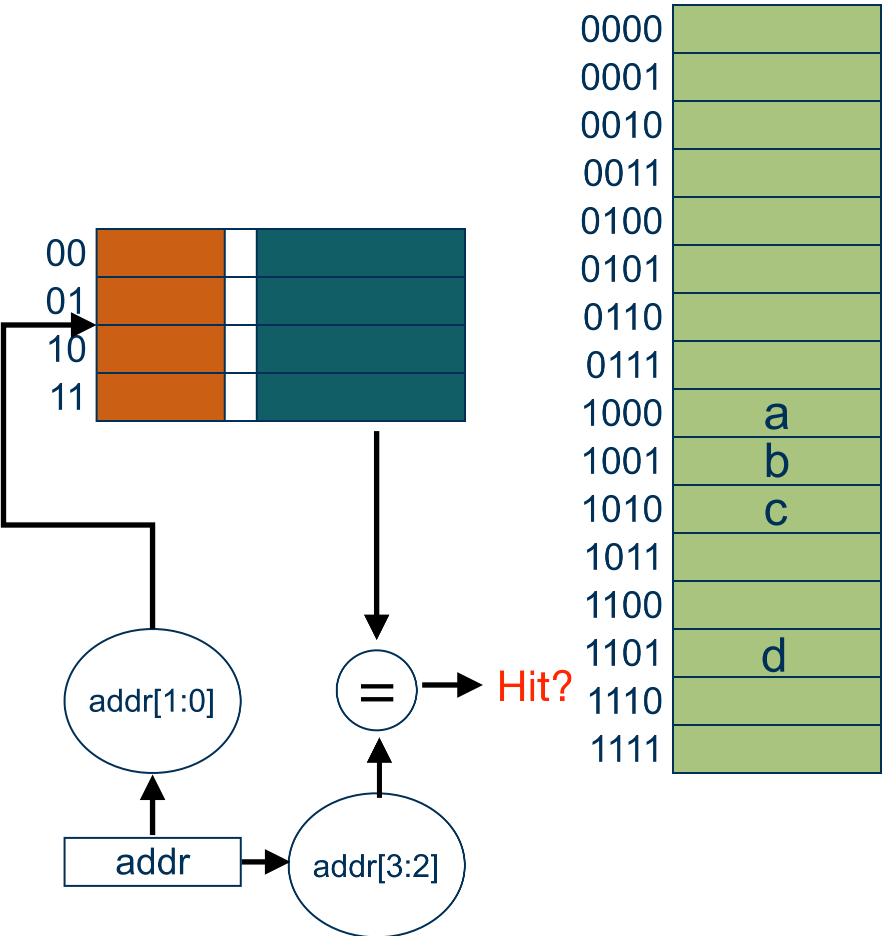
- Cache state is RAM!
- Implement cache as a single SRAM
- Need appropriate comparators

- Memory is implemented as a DRAM

Conflicts

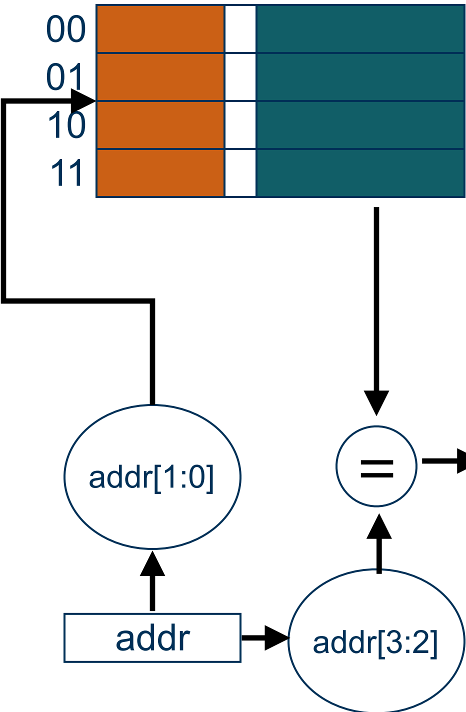
Cache

Memory



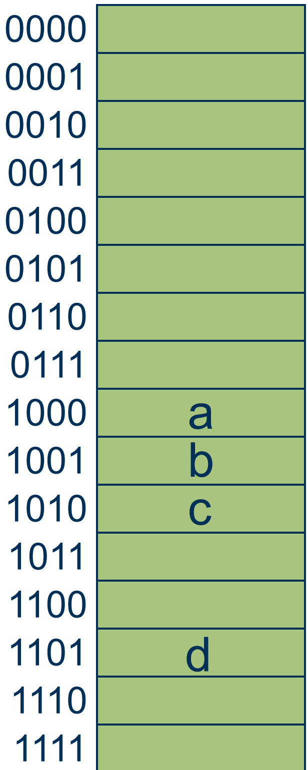
Conflicts

Cache



Hit?

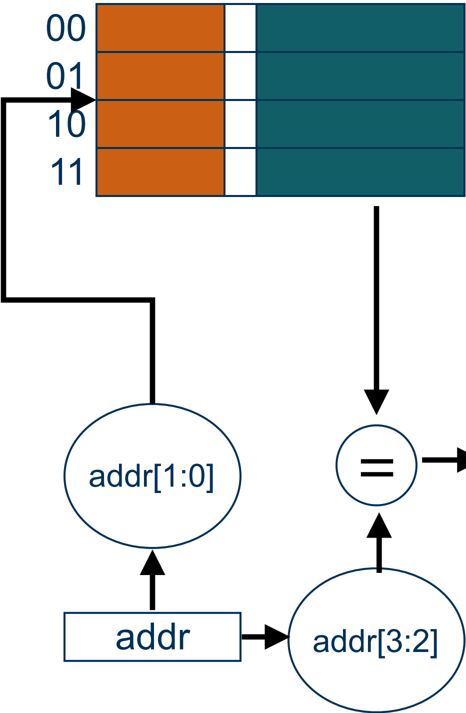
Memory



Assume the following memory access stream:

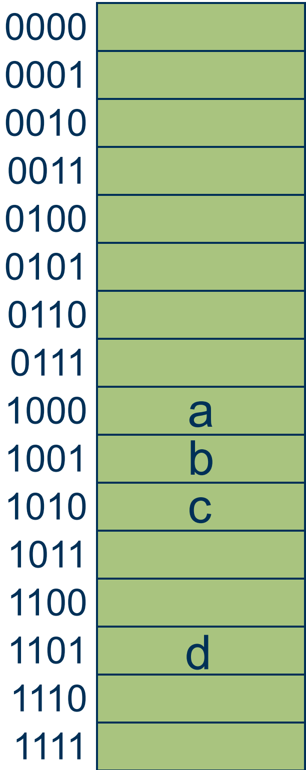
Conflicts

Cache



Hit?

Memory

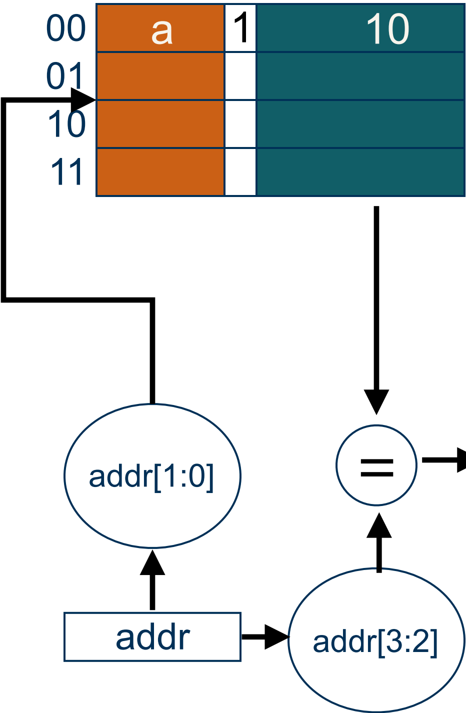


Assume the following memory access stream:

- Read 1000

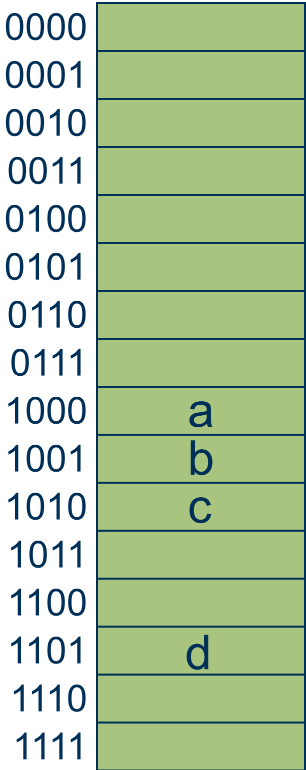
Conflicts

Cache



Hit?

Memory

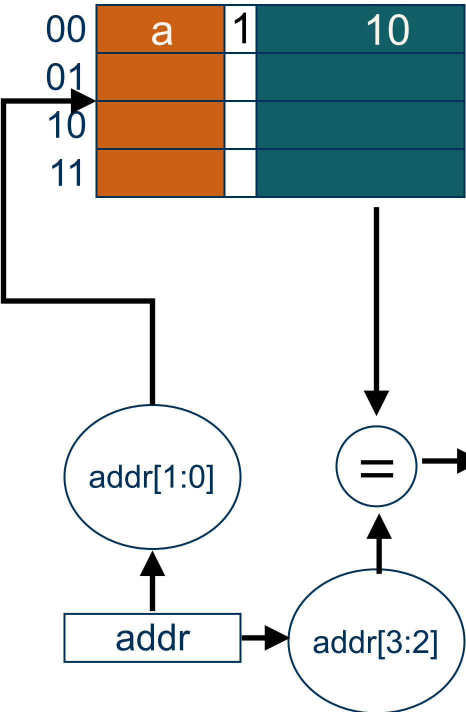


Assume the following memory access stream:

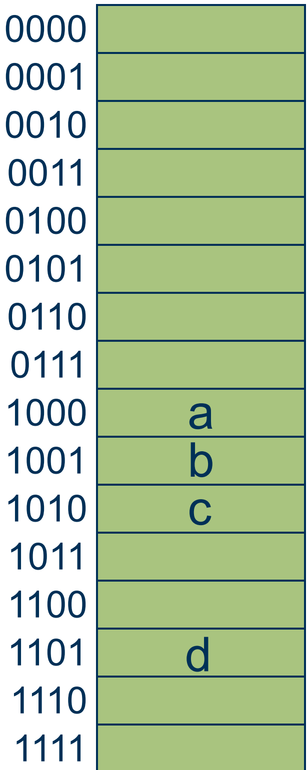
- Read 1000

Conflicts

Cache



Memory



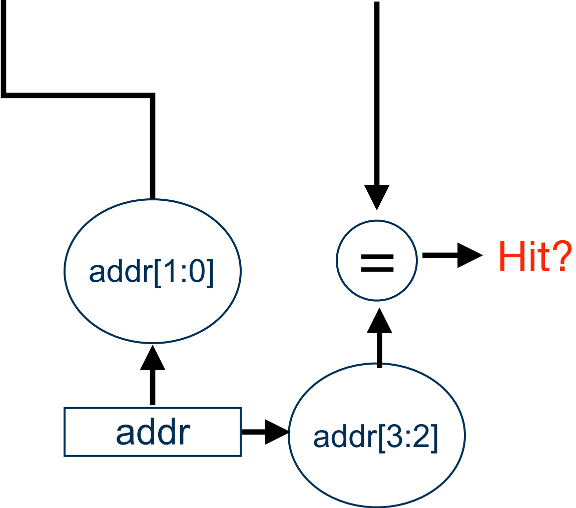
Assume the following memory access stream:

- Read 1000
- Read 1001

Conflicts

Cache

00	a	1	10
01	b	1	10
10			
11			



Memory

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

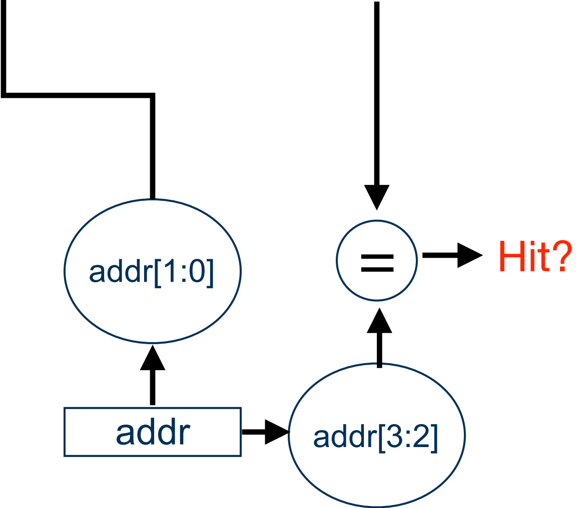
Assume the following memory access stream:

- Read 1000
- Read 1001

Conflicts

Cache

00	a	1	10
01	b	1	10
10			
11			



Memory

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

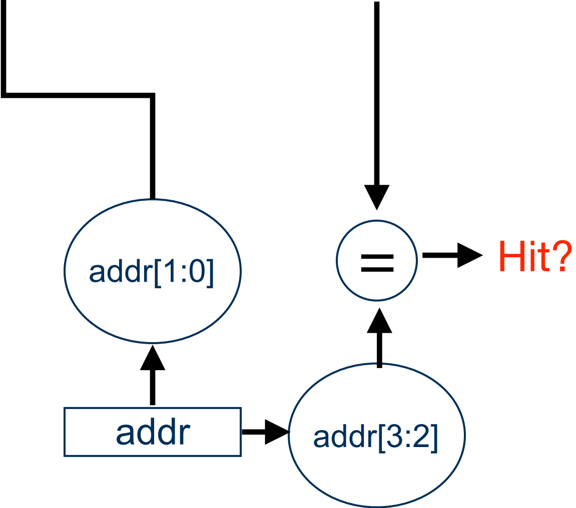
Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

Conflicts

Cache

00	a	1	10
01	b	1	10
10	c	1	10
11			



Memory

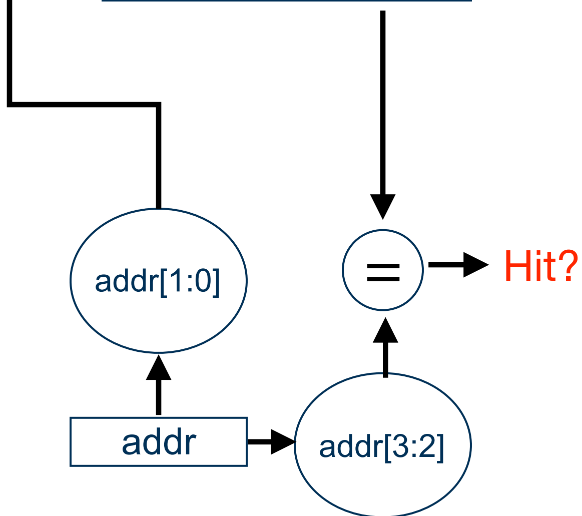
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

- Read 1000
- Read 1001
- Read 1010

Conflicts

Cache



Hit?

Memory

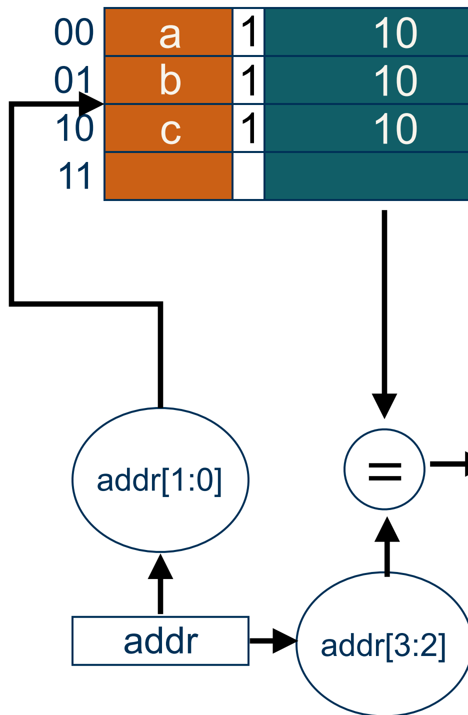
Address	Data
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

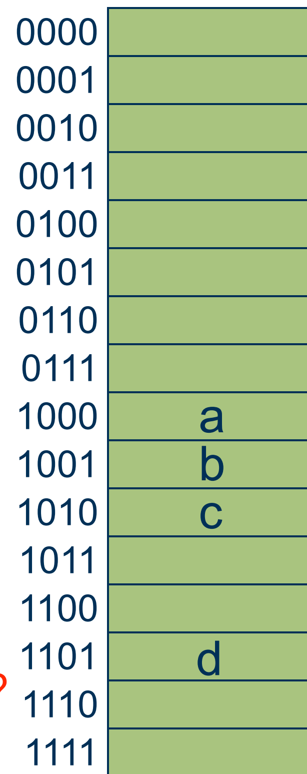
- Read 1000
- Read 1001
- Read 1010
- Read 1101 **(kick out 1001)**

Conflicts

Cache



Memory

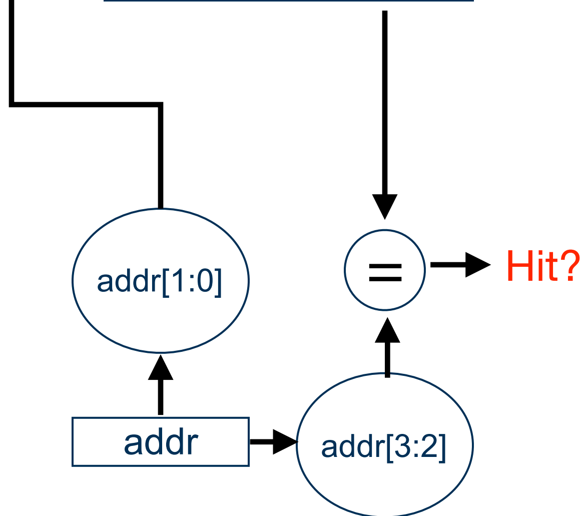


Assume the following memory access stream:

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

Conflicts

Cache



Memory

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

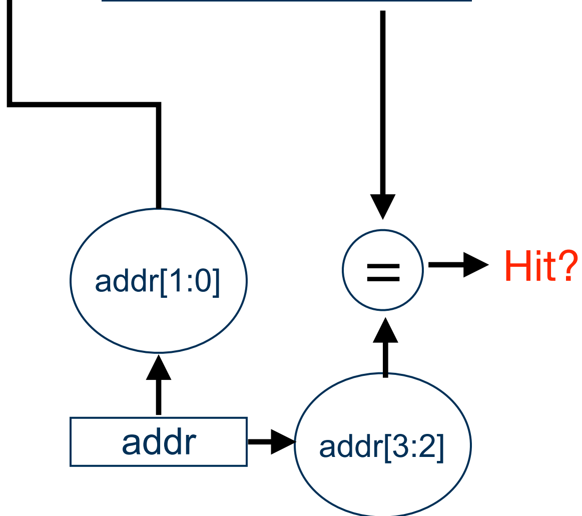
Assume the following memory access stream:

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

Conflicts

Cache

00	a	1	10
01	d	1	11
10	c	1	10
11			



Memory

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

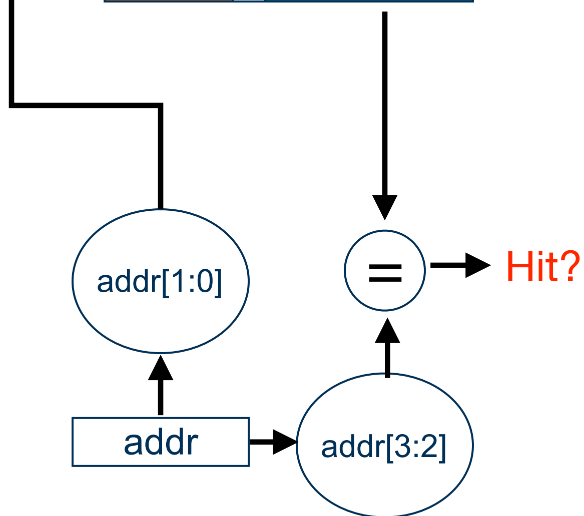
Assume the following memory access stream:

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

Conflicts

Cache

00	a	1	10
01	d	1	11
10	c	1	10
11			



Memory

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

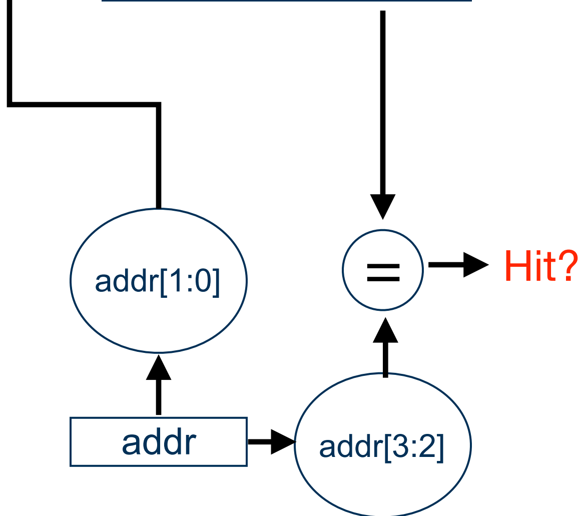
Assume the following memory access stream:

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

Conflicts

Cache

00	a	1	10
01	d	1	11
10	c	1	10
11			



Memory

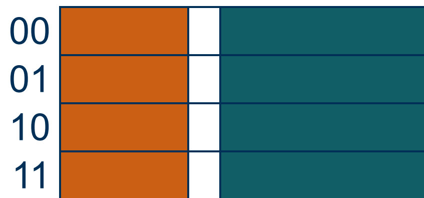
Address	Data
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	a
1001	b
1010	c
1011	
1100	
1101	d
1110	
1111	

Assume the following memory access stream:

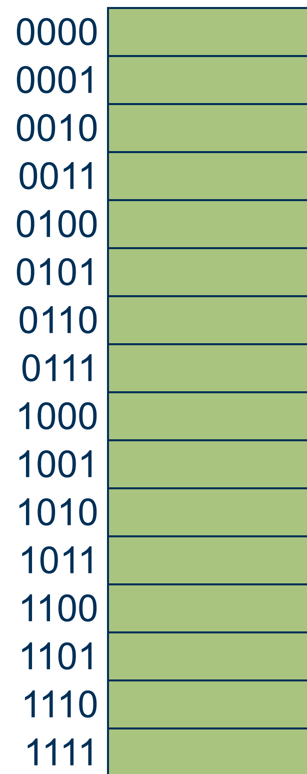
- Read 1000
- Read 1001
- Read 1010
- Read 1101 **(kick out 1001)**
  - addr[1:0]: 01
  - addr[3:2]: 11
- Read 1001 -> **Miss!**
- Why? Each memory location is mapped to only one cache location

Sets

Cache

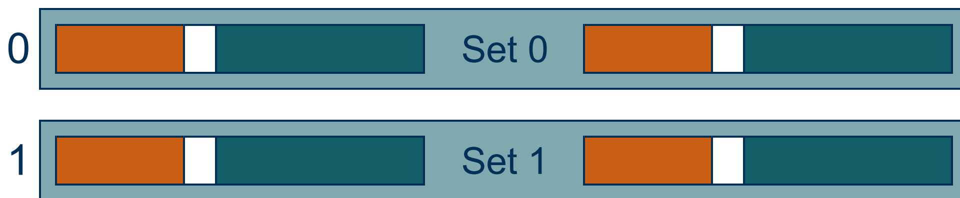


Memory



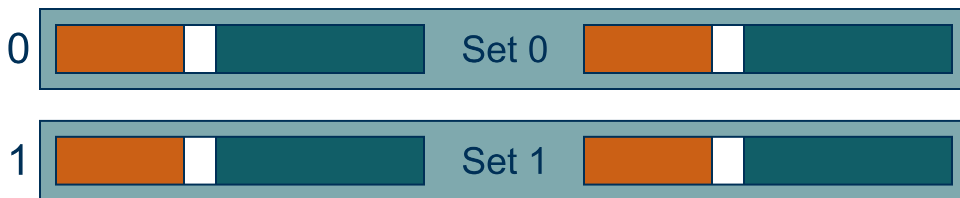
- Each cacheable memory location is mapped to **a set of cache locations**
- A set is one or more cache locations
- Set size is the number of locations in a set, also called **associativity**

2 Way Set Associative



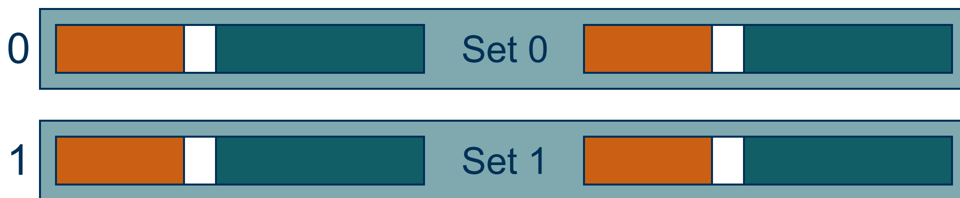
2 Way Set Associative

- 2 sets, each set has two entries



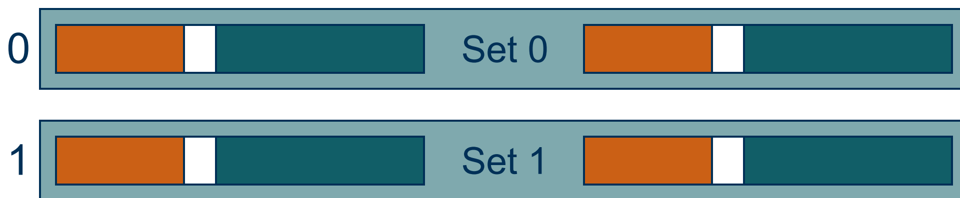
2 Way Set Associative

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



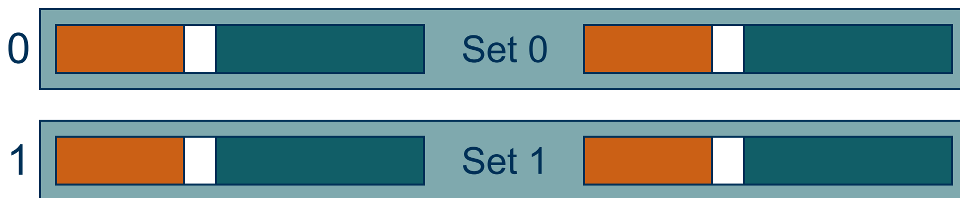
2 Way Set Associative

- 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]

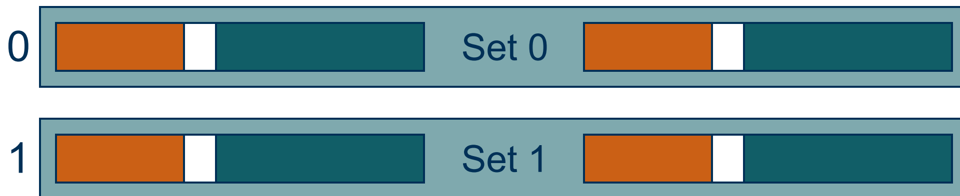


2 Way Set Associative

- 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

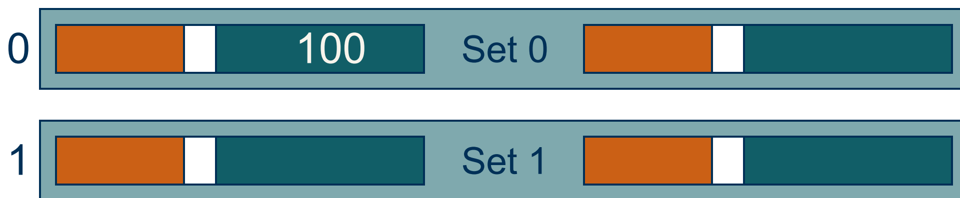


2 Way Set Associative



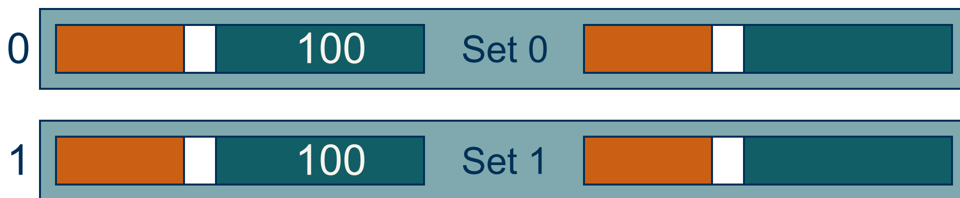
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:

2 Way Set Associative



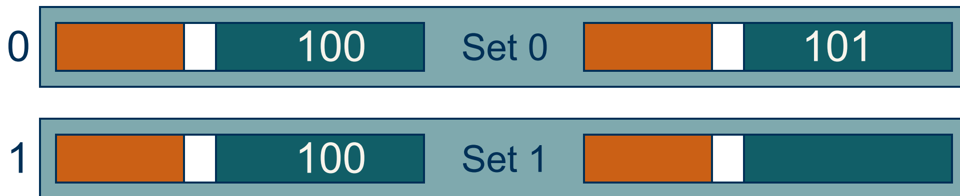
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
  - Read 1000

2 Way Set Associative



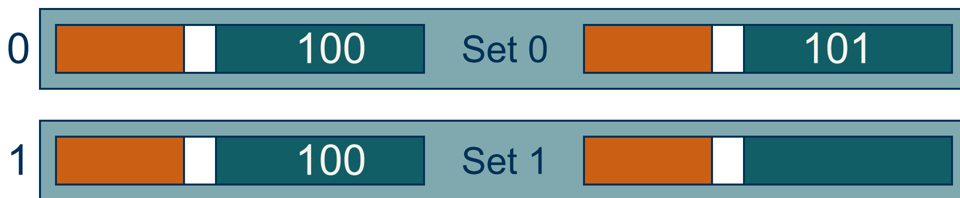
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
  - Read 1000
  - Read 1001

2 Way Set Associative



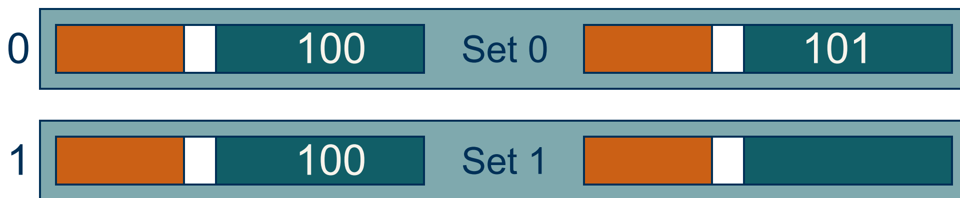
- 2 sets, each set has two entries
- Only need one bit, addr[0] to index into the cache now
- Correspondingly, the tag needs 3 bits: Addr[3:1]
- Either entry can store any address that gets mapped to that set
- Now with the same access stream:
    - Read 1000
    - Read 1001
    - Read 1010

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!**

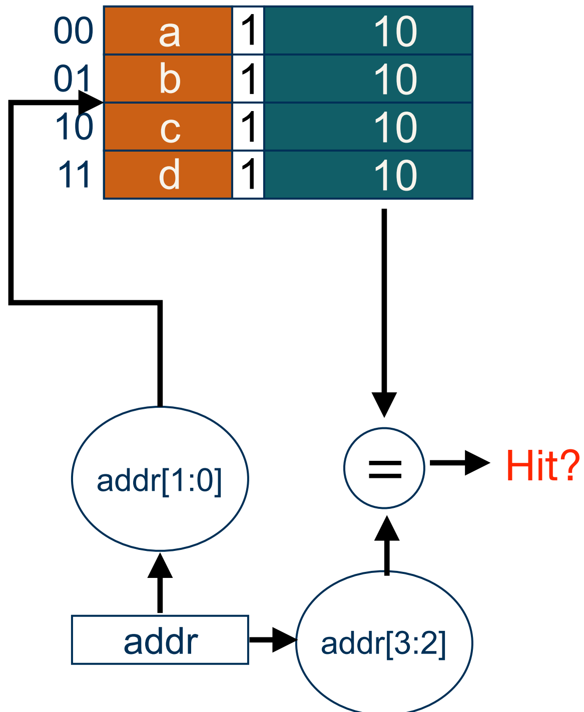
Associative verses Direct Mapped Trade-offs

Associative verses Direct Mapped Trade-offs

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

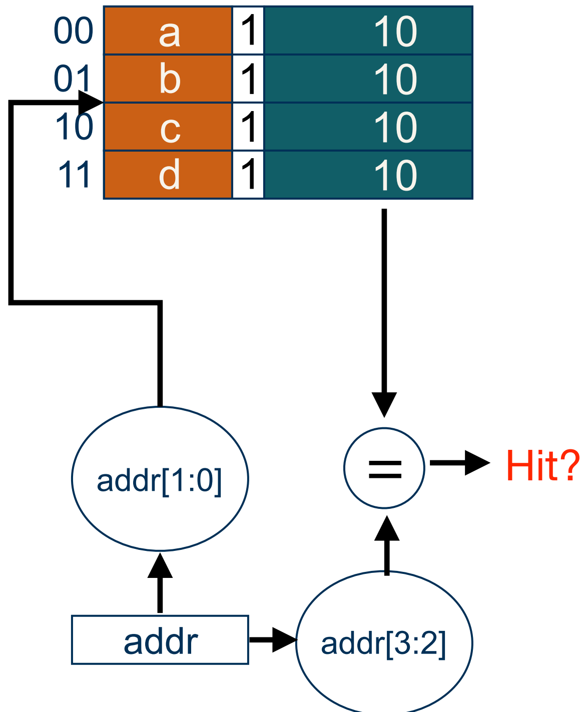
Associative verses Direct Mapped Trade-offs

- 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?

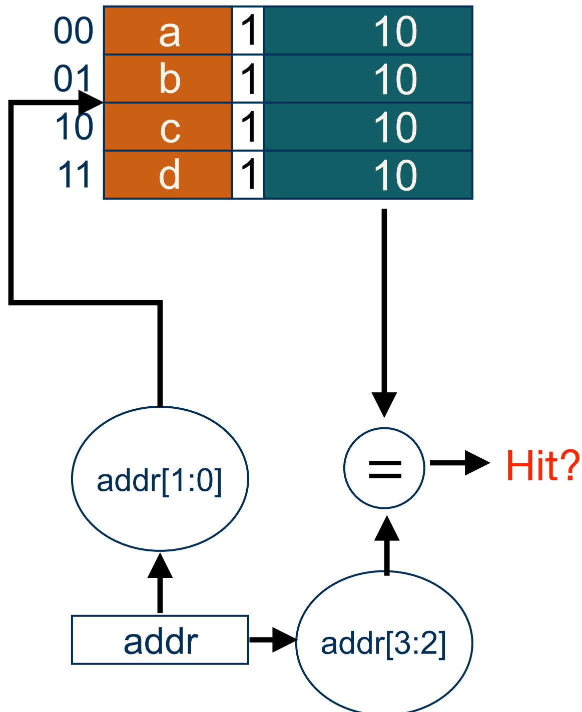


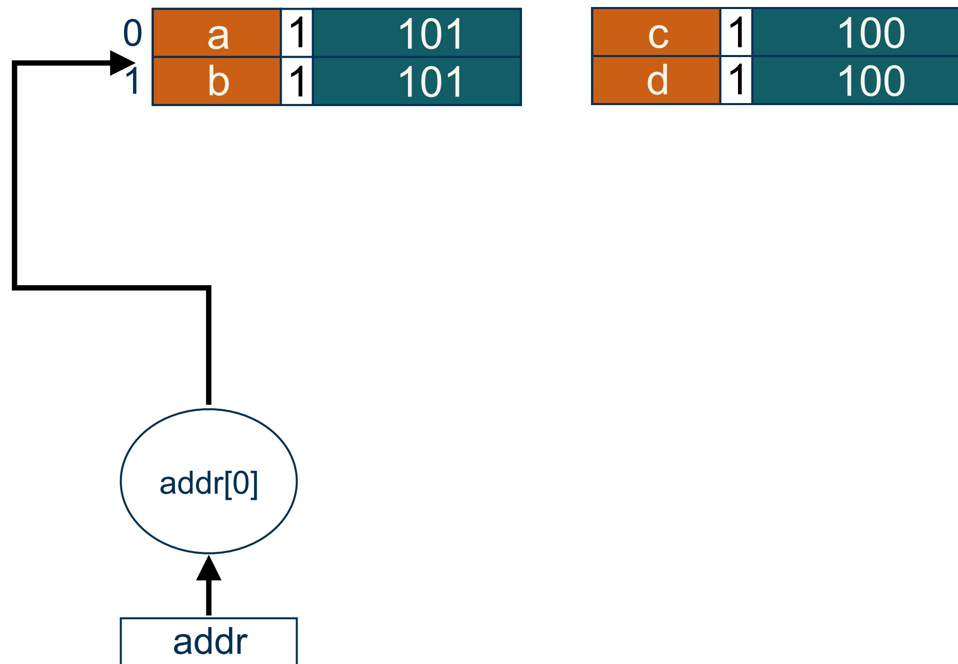
0	a	1	101
1	b	1	101

c	1	100
d	1	100

Associative verses Direct Mapped Trade-offs

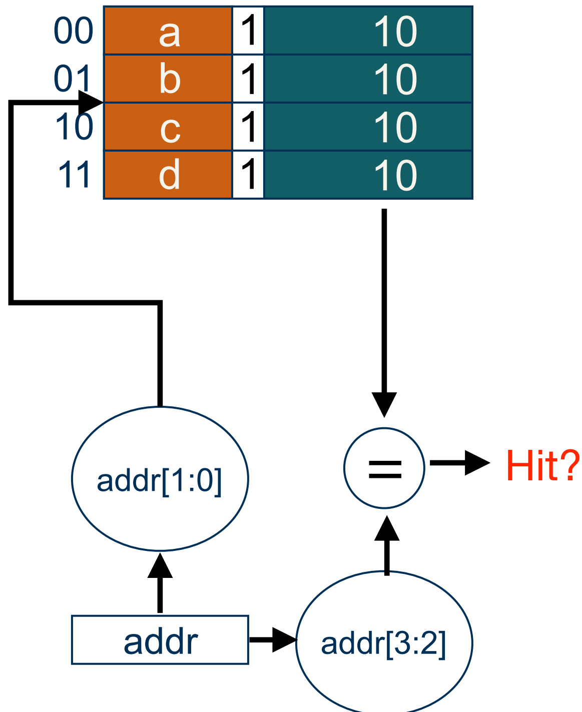
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower. Why?

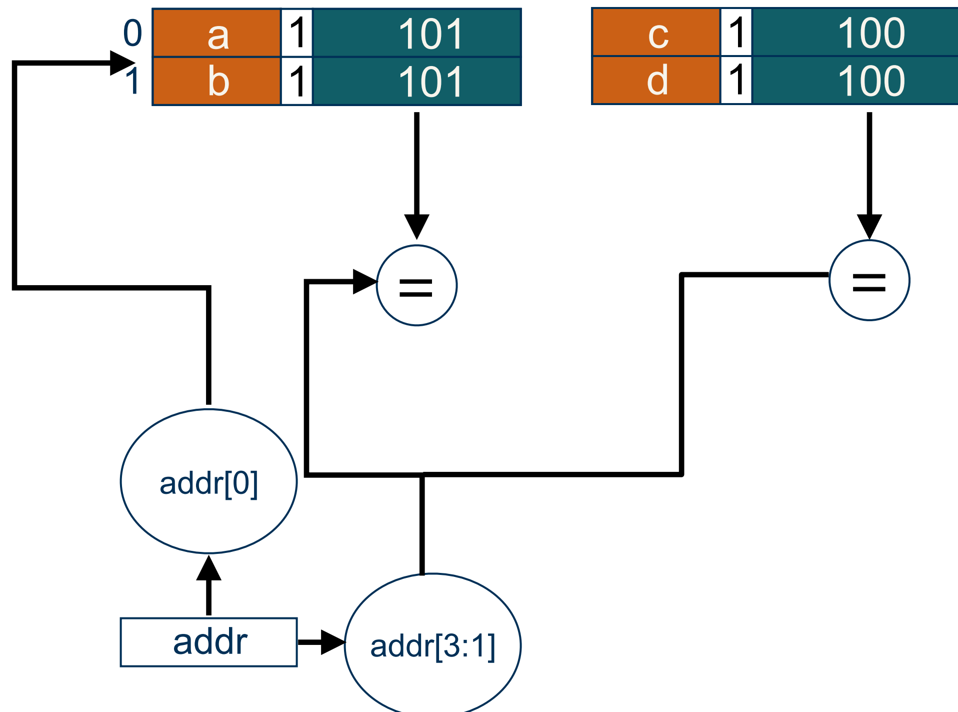




Associative verses Direct Mapped Trade-offs

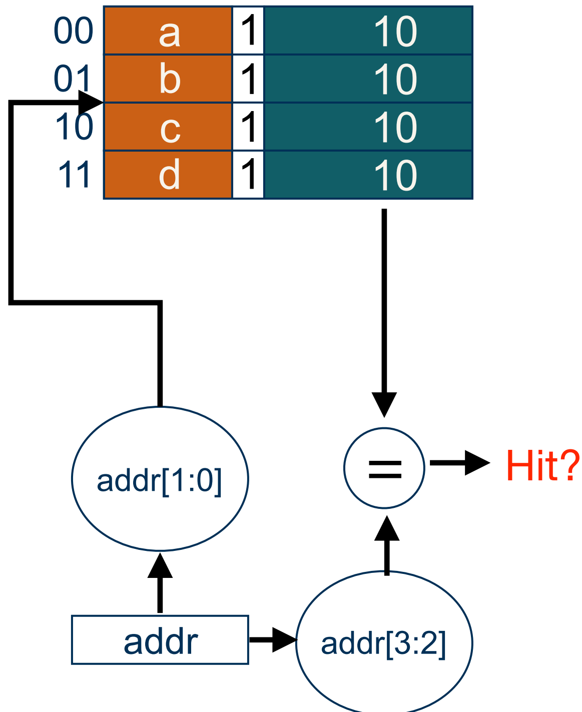
- Direct-Mapped cache
  - Generally lower hit rate
  - Simpler, Faster
- Associative cache
  - Generally higher hit rate. Better utilization of cache resources
  - Slower. Why?

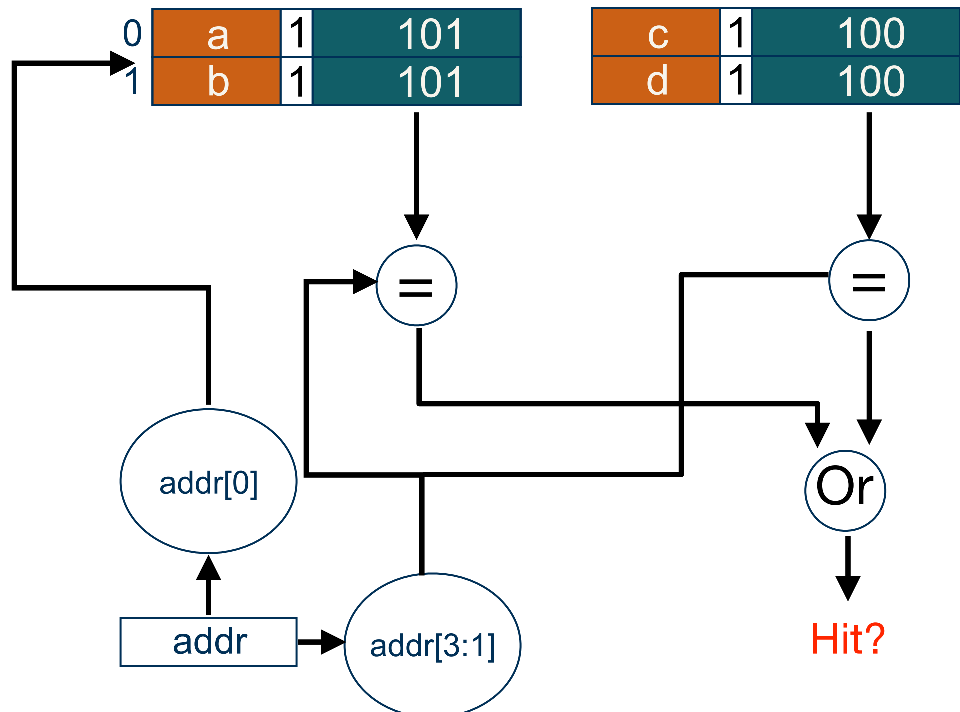




Associative verses Direct Mapped Trade-offs

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





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

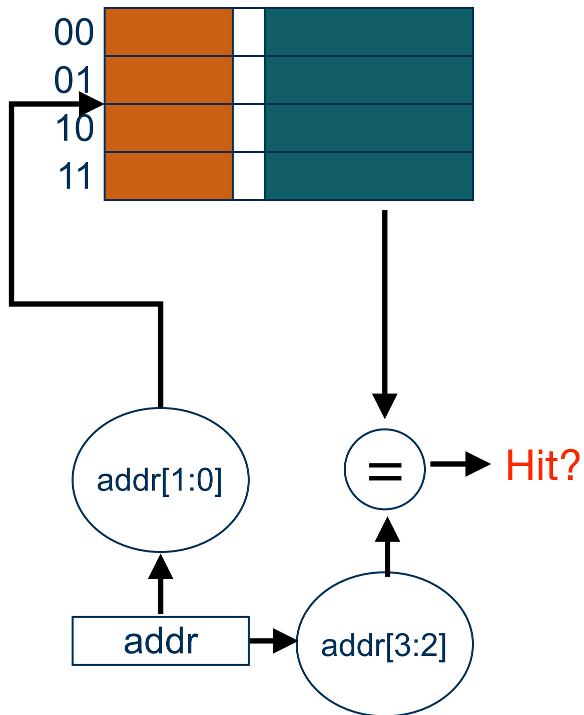
Locality again

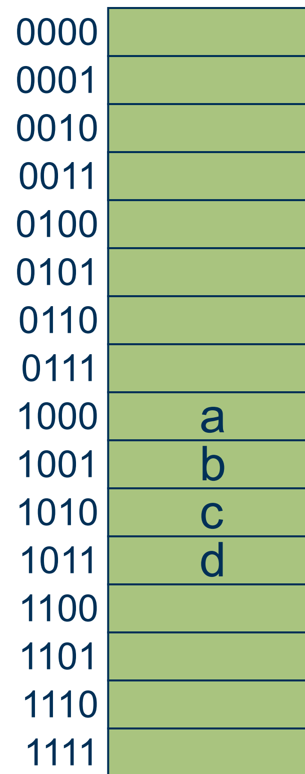
- So far: temporal locality
- What about spatial?
- Idea: Each cache location (cache line) store multiple bytes

Cache-Line Size of 2

Cache

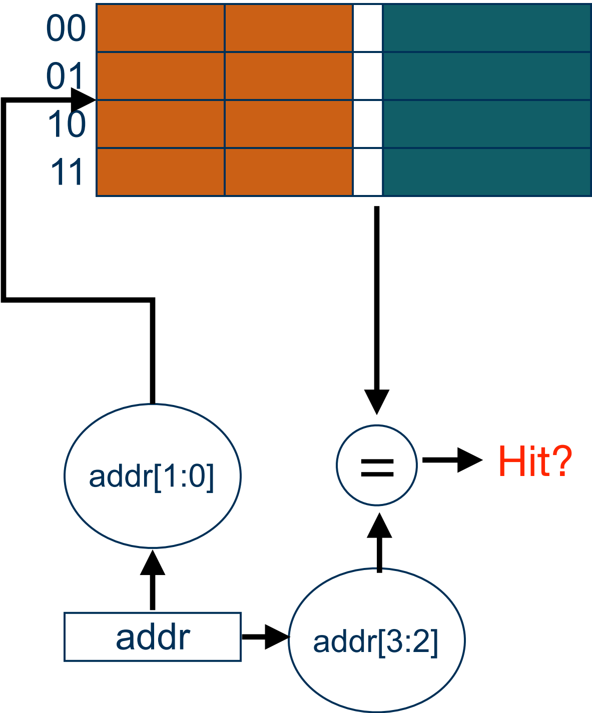
Memory



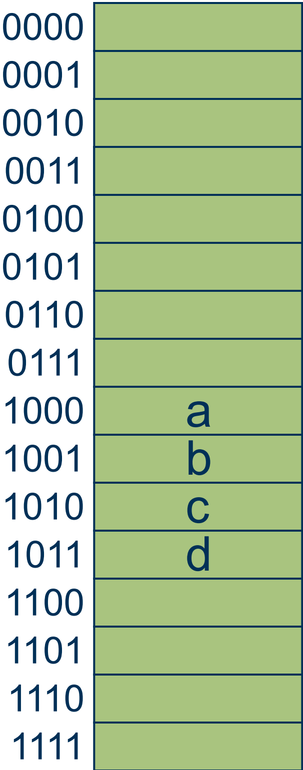


Cache-Line Size of 2

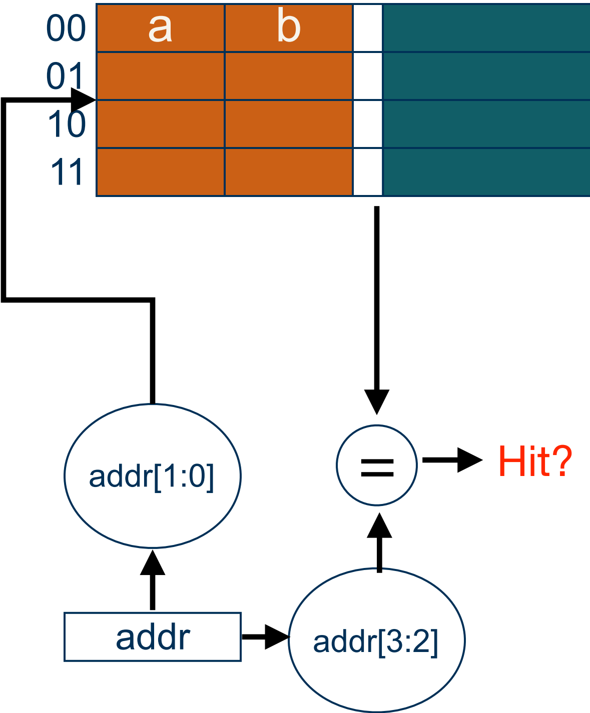
Cache

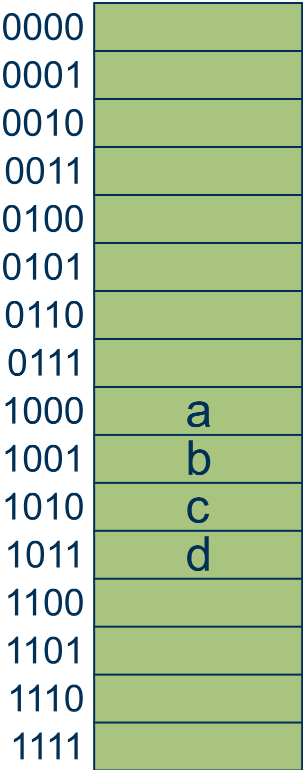


Memory



Cache-Line Size of 2

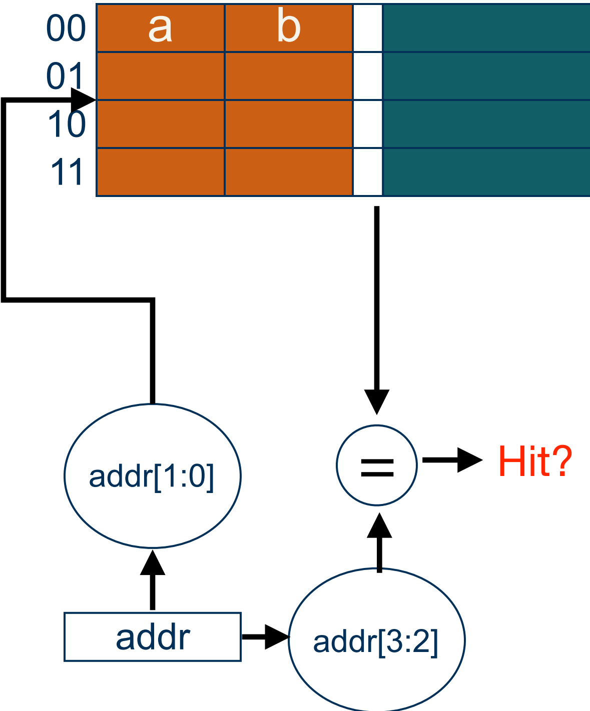




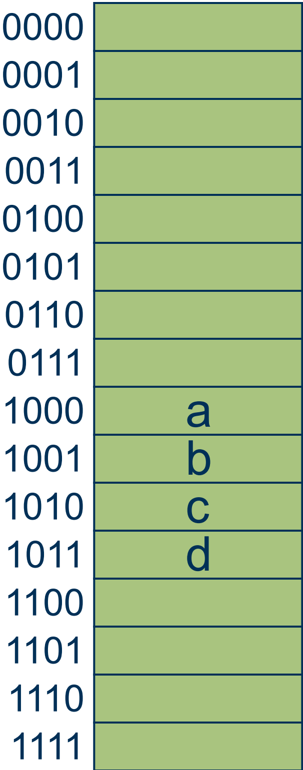
- Read 1000

Cache-Line Size of 2

Cache



Memory

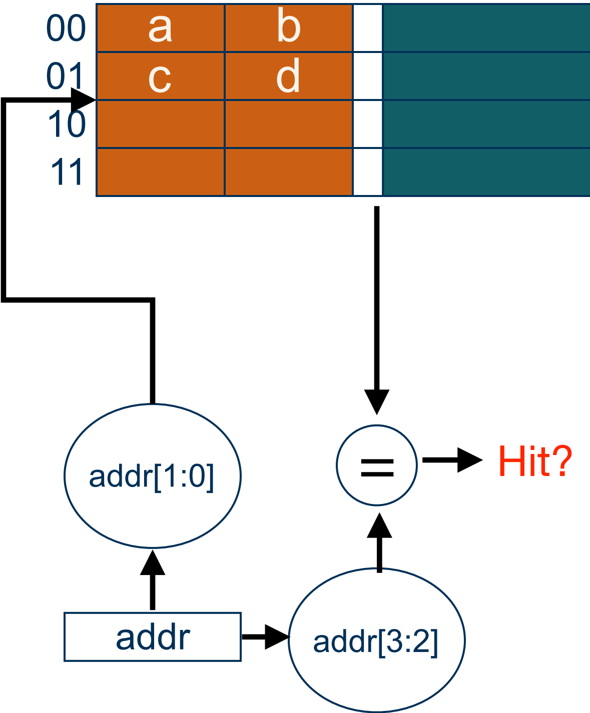


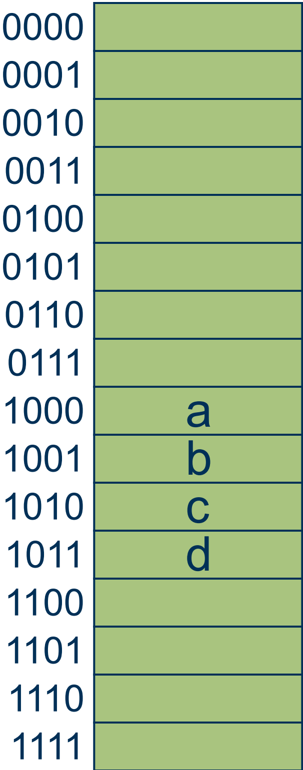
- Read 1000
- Read 1001 (Hit!)

Cache-Line Size of 2

Cache

Memory

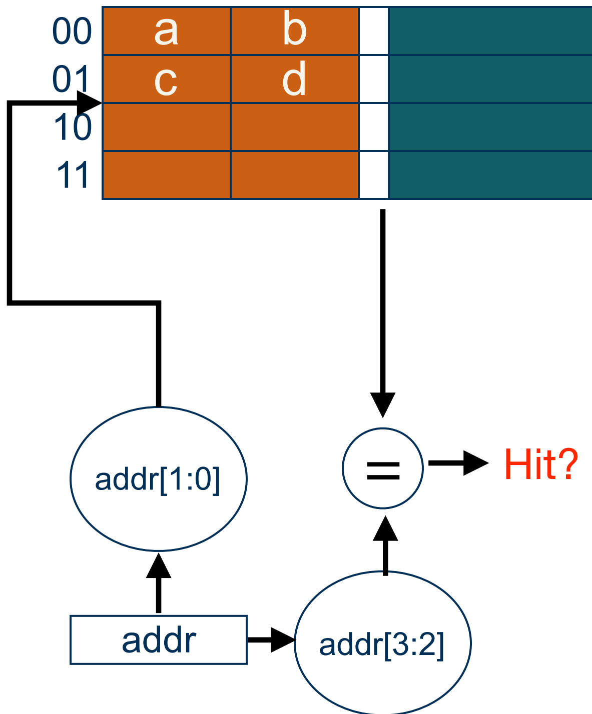




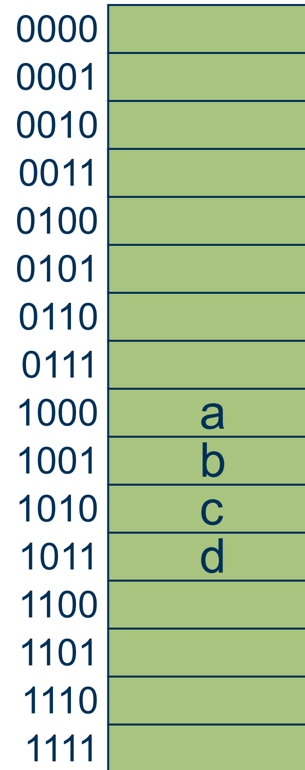
- Read 1000
- Read 1001 (Hit!)
- Read 1010

Cache-Line Size of 2

Cache



Memory



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

Cache 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

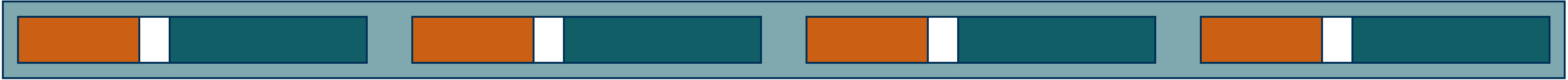
Eviction/Replacement Policy

- Which cache line should be replaced?

Eviction/Replacement Policy

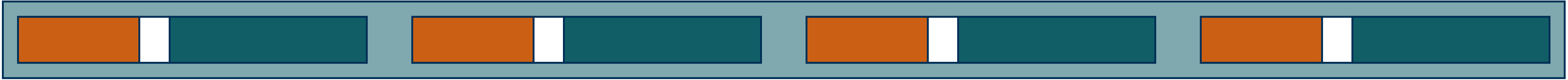
- Which cache line should be replaced?
  - Direct mapped? Only one place!

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?

Cache Lines

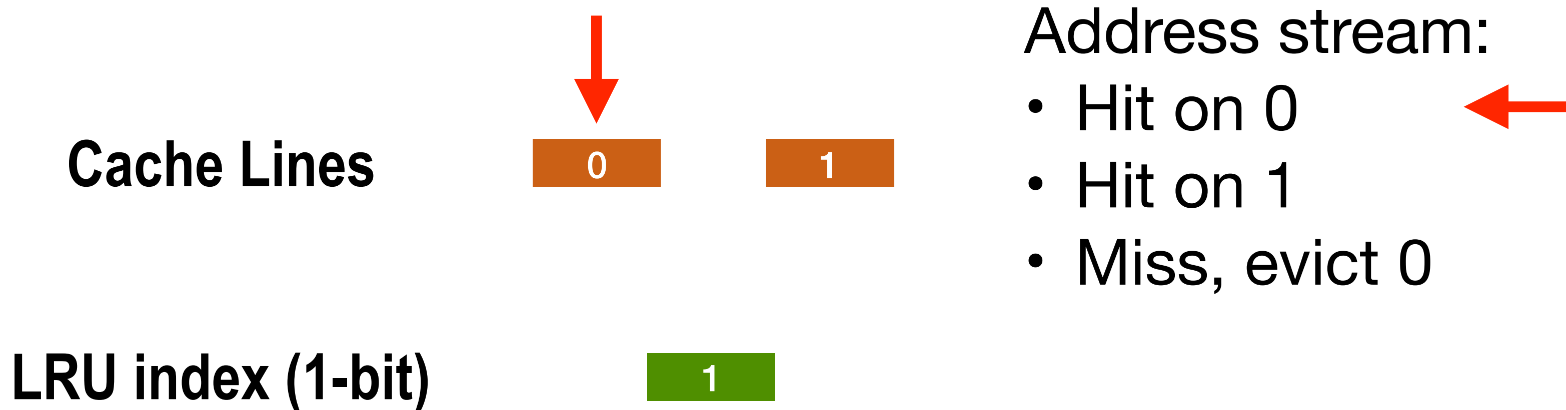
LRU index (1-bit)

Address stream:

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

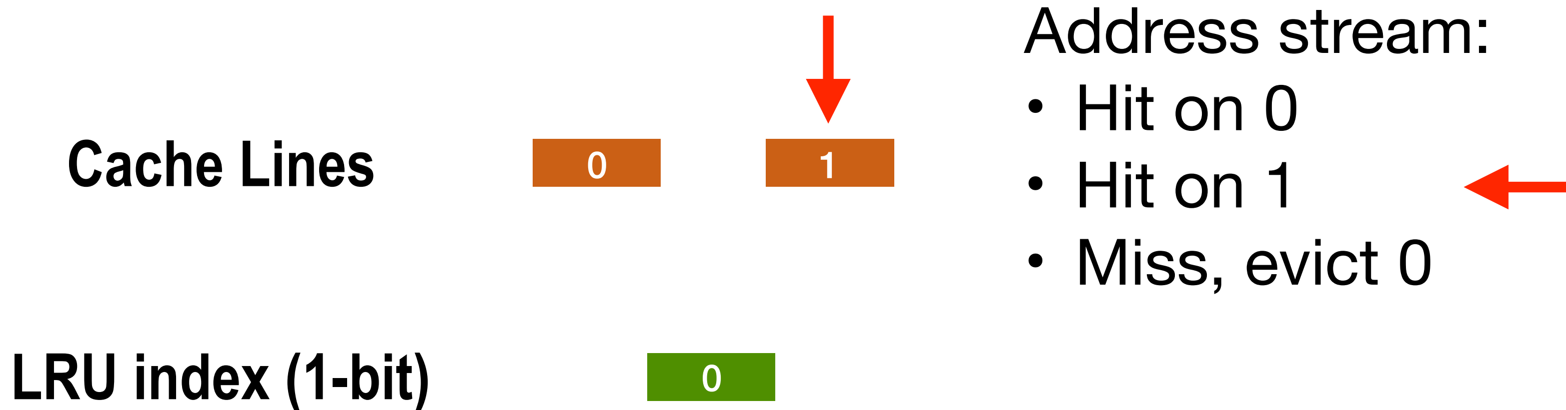
Implementing LRU

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



Implementing LRU

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



Implementing LRU

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

Cache Lines

LRU index (1-bit)

Address stream:

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

Implementing LRU

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

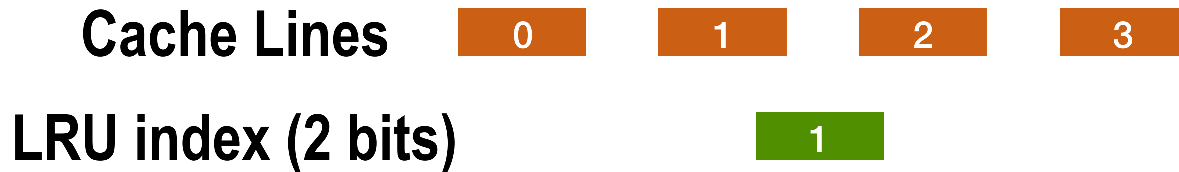
Cache Lines

LRU index (1-bit)

Address stream:

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

Implementing LRU

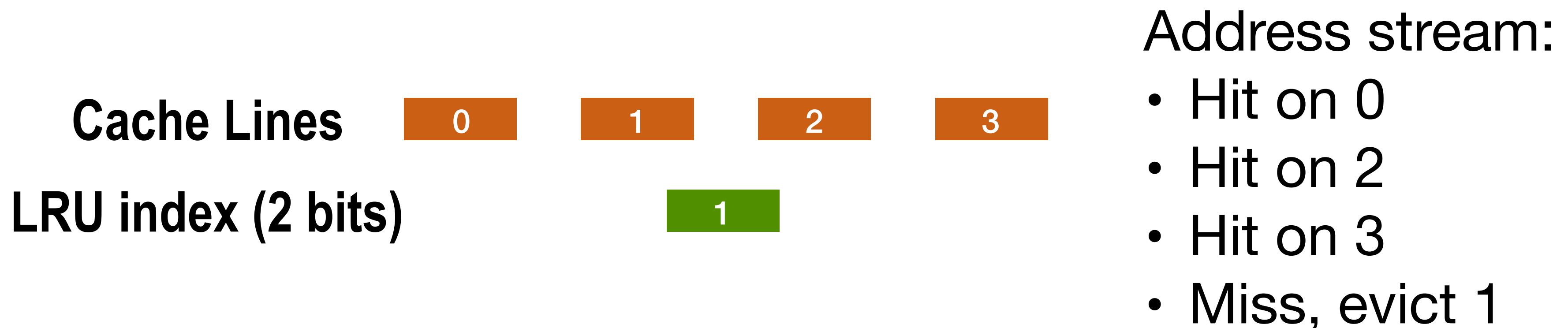


Address stream:

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

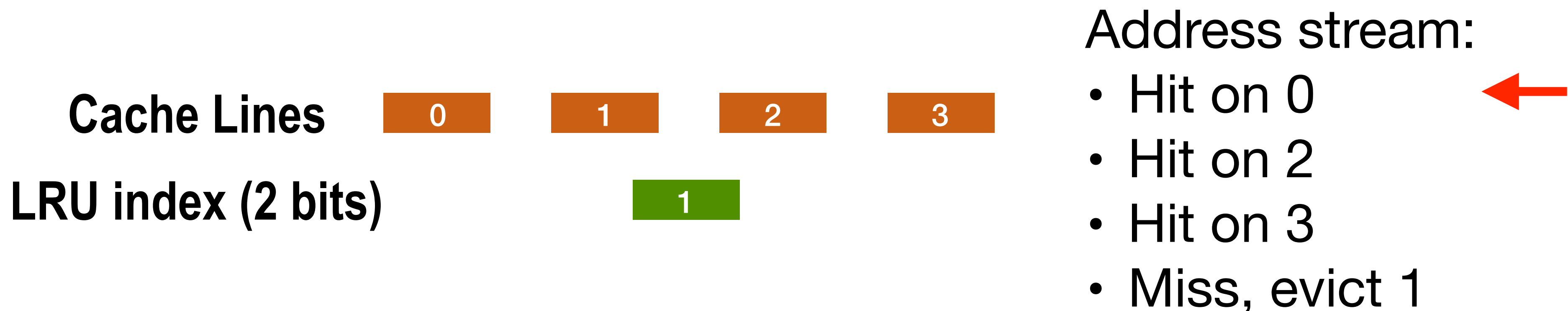
Implementing LRU

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



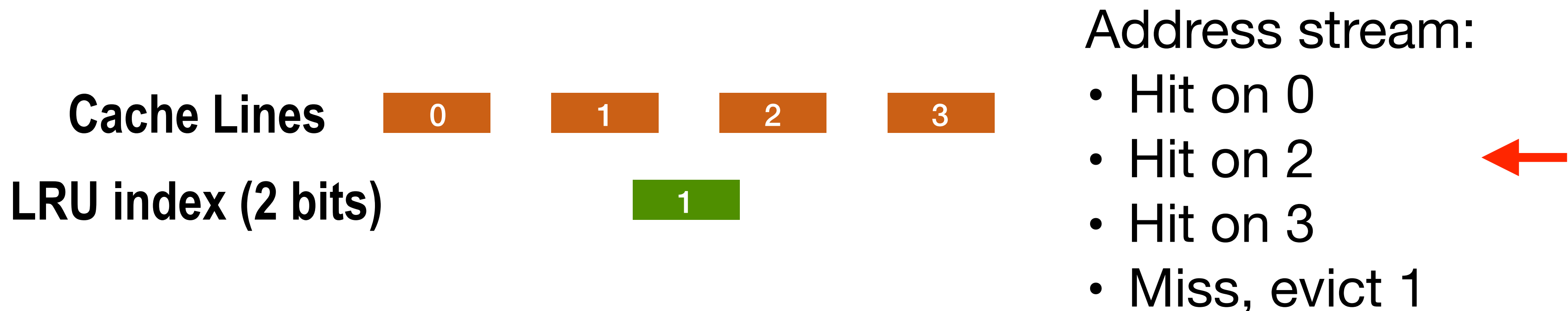
Implementing LRU

- 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

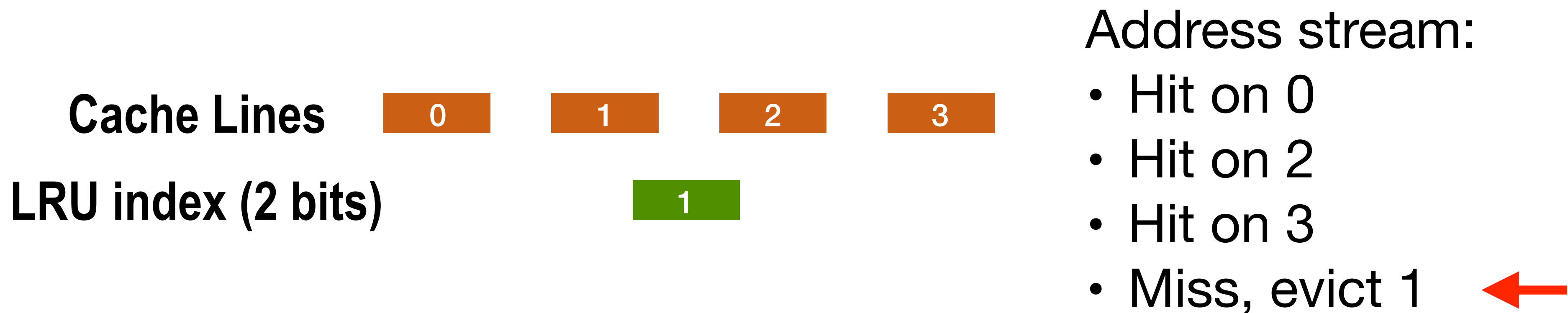
- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - Will the same mechanism work?

Address stream:

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

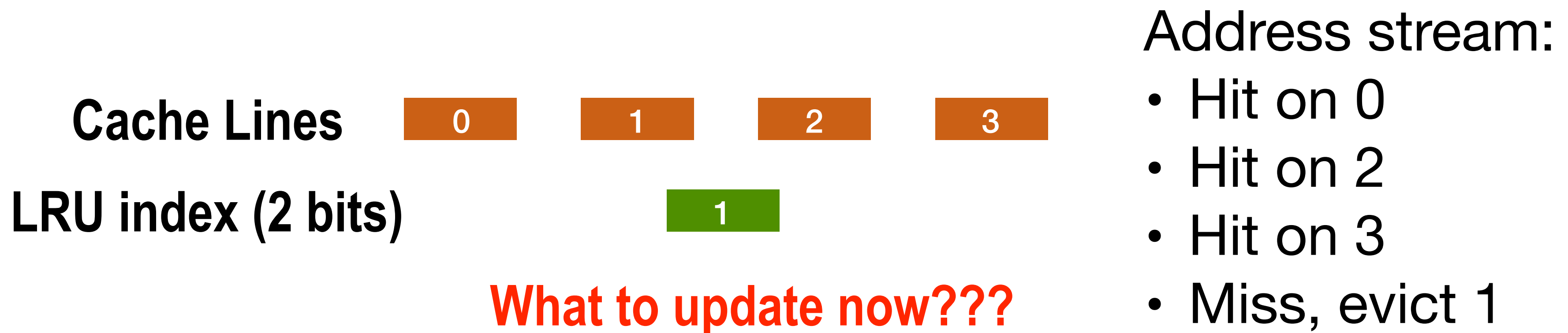
Implementing LRU

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



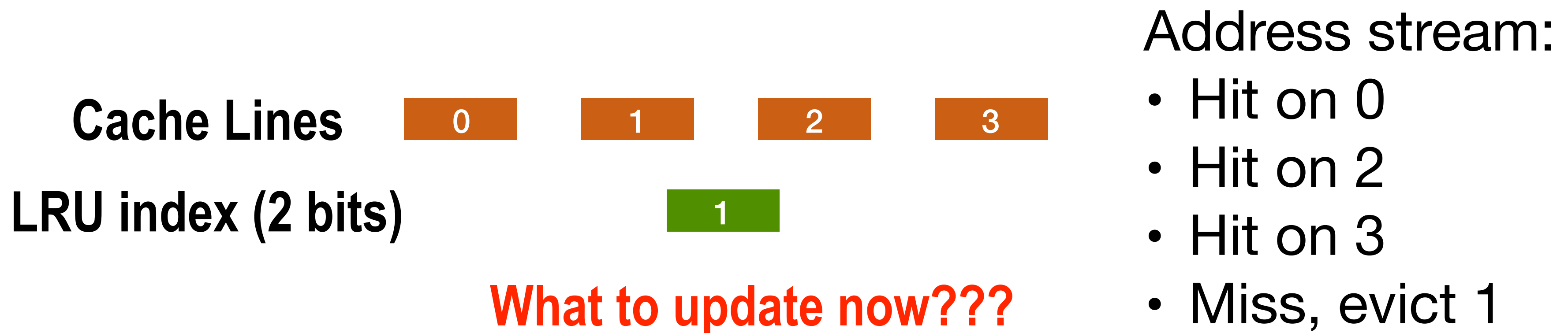
Implementing LRU

- 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?
  - Essentially have to track the ordering of all cache lines



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?

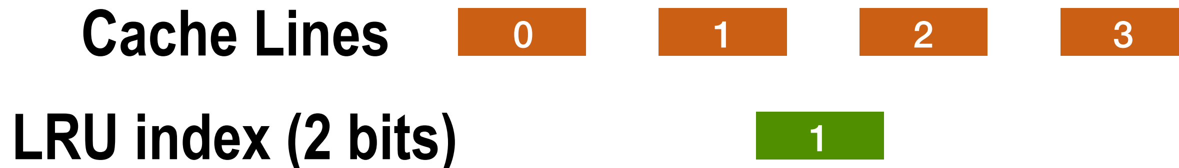
What to update now???

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?



Address stream:
- Hit on 0
- Hit on 2
- Hit on 3
- Miss, evict 1

What to update now???

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

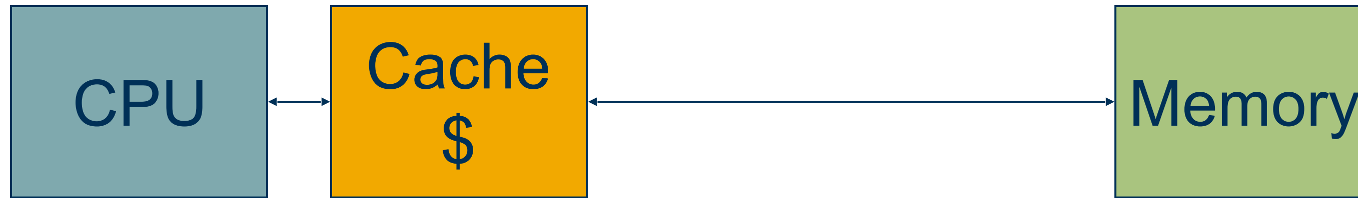
LRU index (2 bits) 1

What to update now???

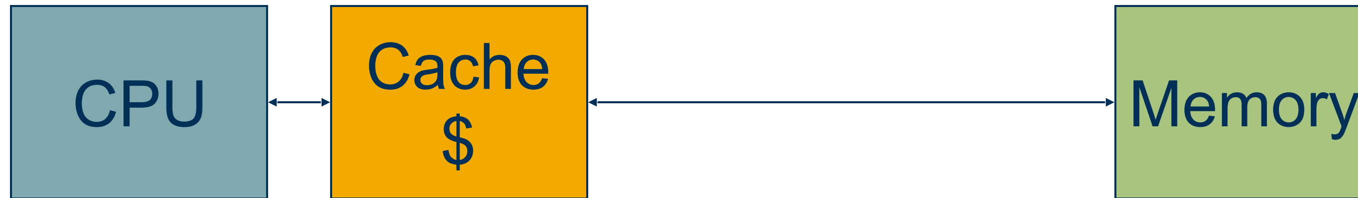
Address stream:

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

General Rule: Bigger == Slower

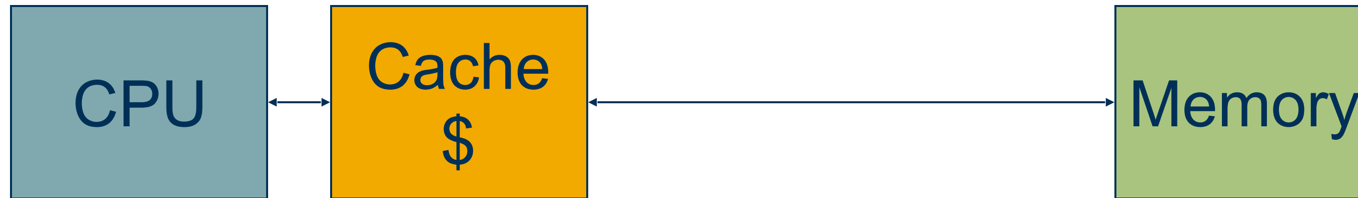


General Rule: Bigger == Slower



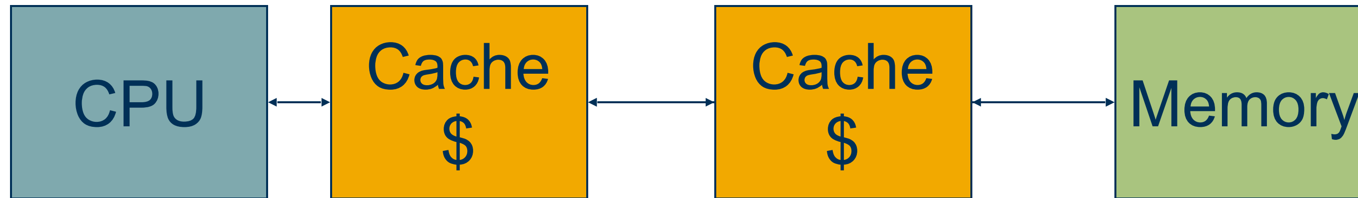
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)

General Rule: Bigger == Slower



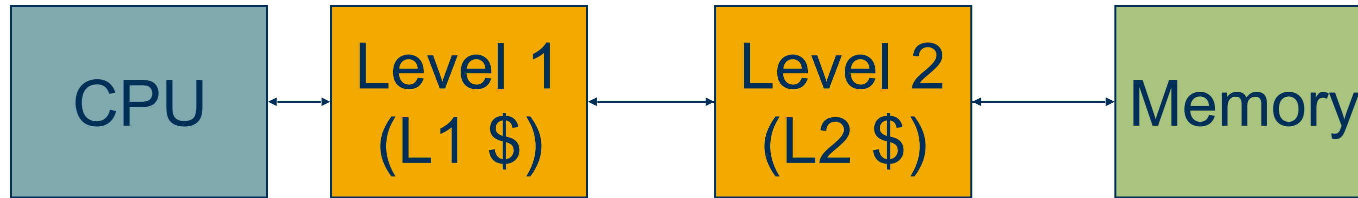
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

General Rule: Bigger == Slower



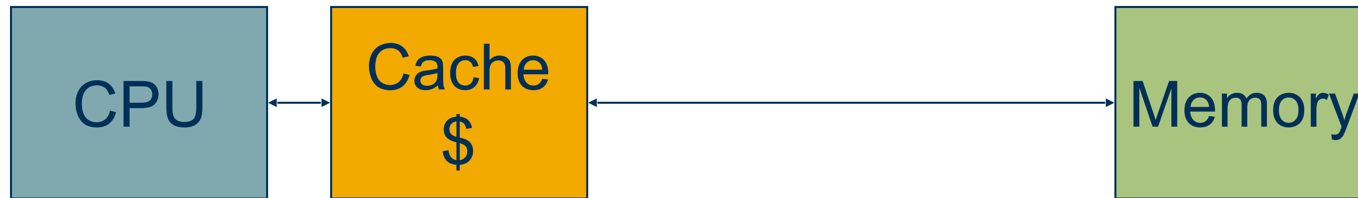
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

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