15-213 Recitation
Caches & C Review

Your TAs

Friday, September 27th

Reminders

- **`attacklab`** was due *yesterday*.
- **`cachelab`** was released yesterday, and is due ***Thursday October 10th***.
- Written 4 due ***October 2nd***.
- Written 5 ("Midterm") coming up!
  - Roughly the length of two writtens, so make sure to plan your time accordingly.

Agenda

- Intro to `cachelab`
- Review: Cache Concepts
- Review: Programming in C
- Activity: Parsing Command-Line Arguments with `getopt()`
- Cache Practice Problems

cachelab

`cachelab`: Overview

- First project-based assignment:
  - You'll write a *cache simulator* in C. From scratch!
- Take in parameters defining the cache structure (**`s`**, **`E`**, **`b`**).
- Read a "trace file" of memory accesses and simulate each one.
- After simulating those accesses, return the number of hits, misses, evictions, etc.

Review: Cache Concepts

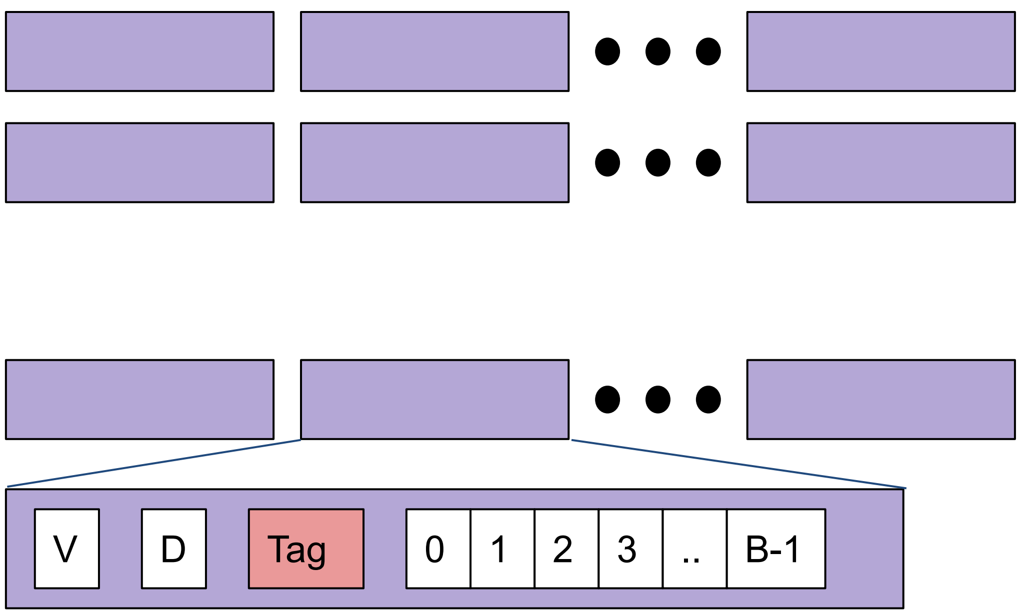
Cache Concepts: Configurations

- Your cache simulators will need to support *parameters* (**`s`**, **`E`**, **`b`**) that allow the user to configure the layout of the cache.
- But what do these parameters mean?
- Let's review how a cache is organized!

Cache Organization

Note: don't need to store data for `cachelab`.

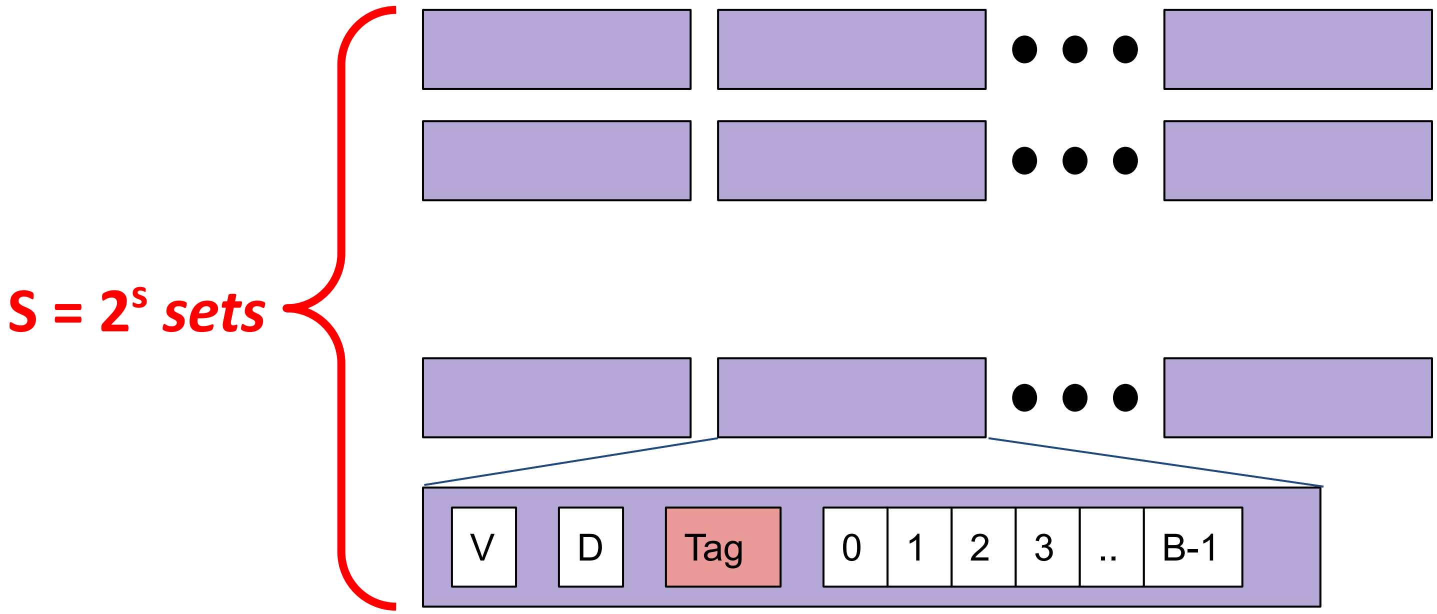
- A cache is composed of *sets*
- Each set is composed of some number of *lines*
- Each line stores the cached data itself, as well as information used by the cache.



Cache Organization

- A cache is composed of *sets*

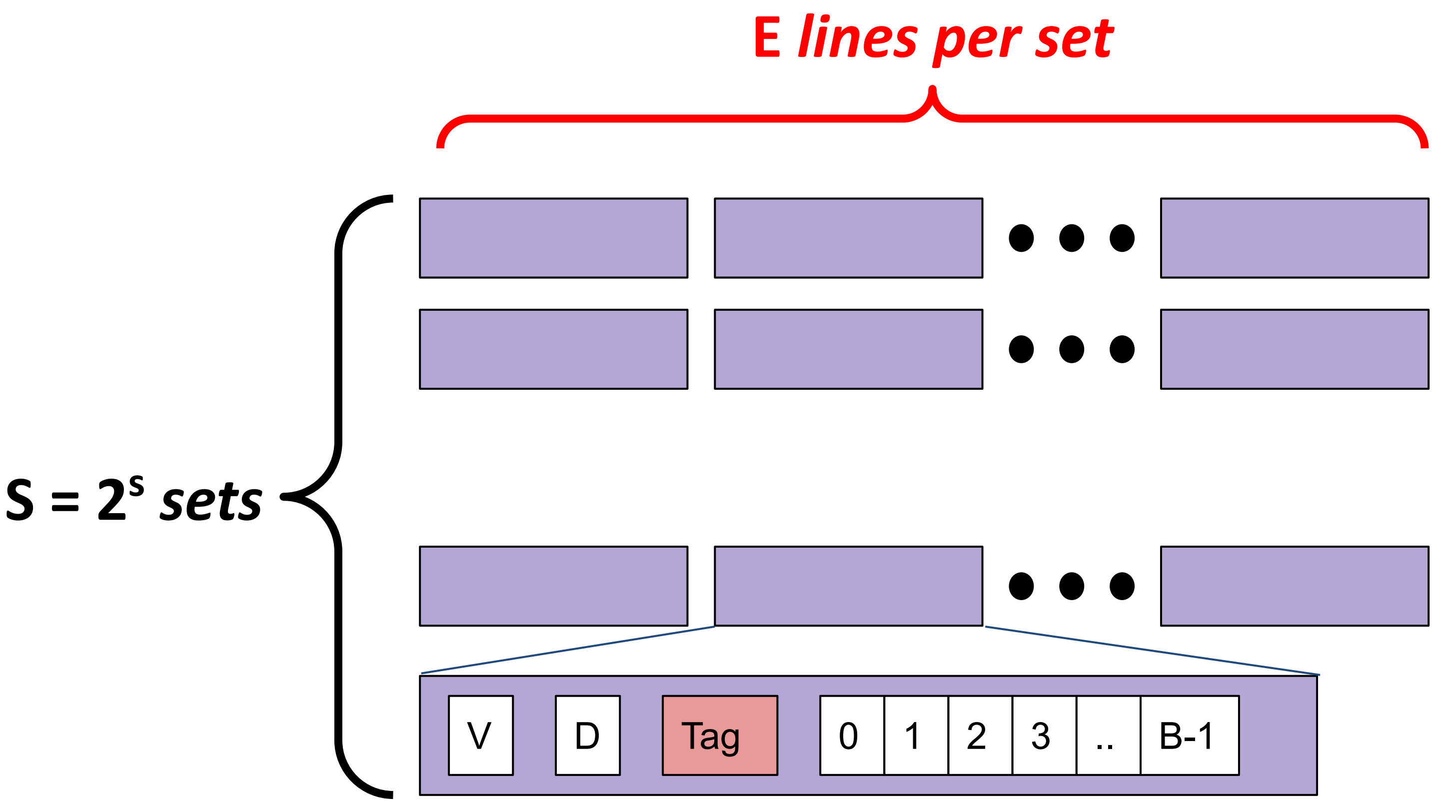
s – Number of set *bits*
$S = 2^s$ – Number of *sets*



Cache Organization

- Each set is composed of *lines*

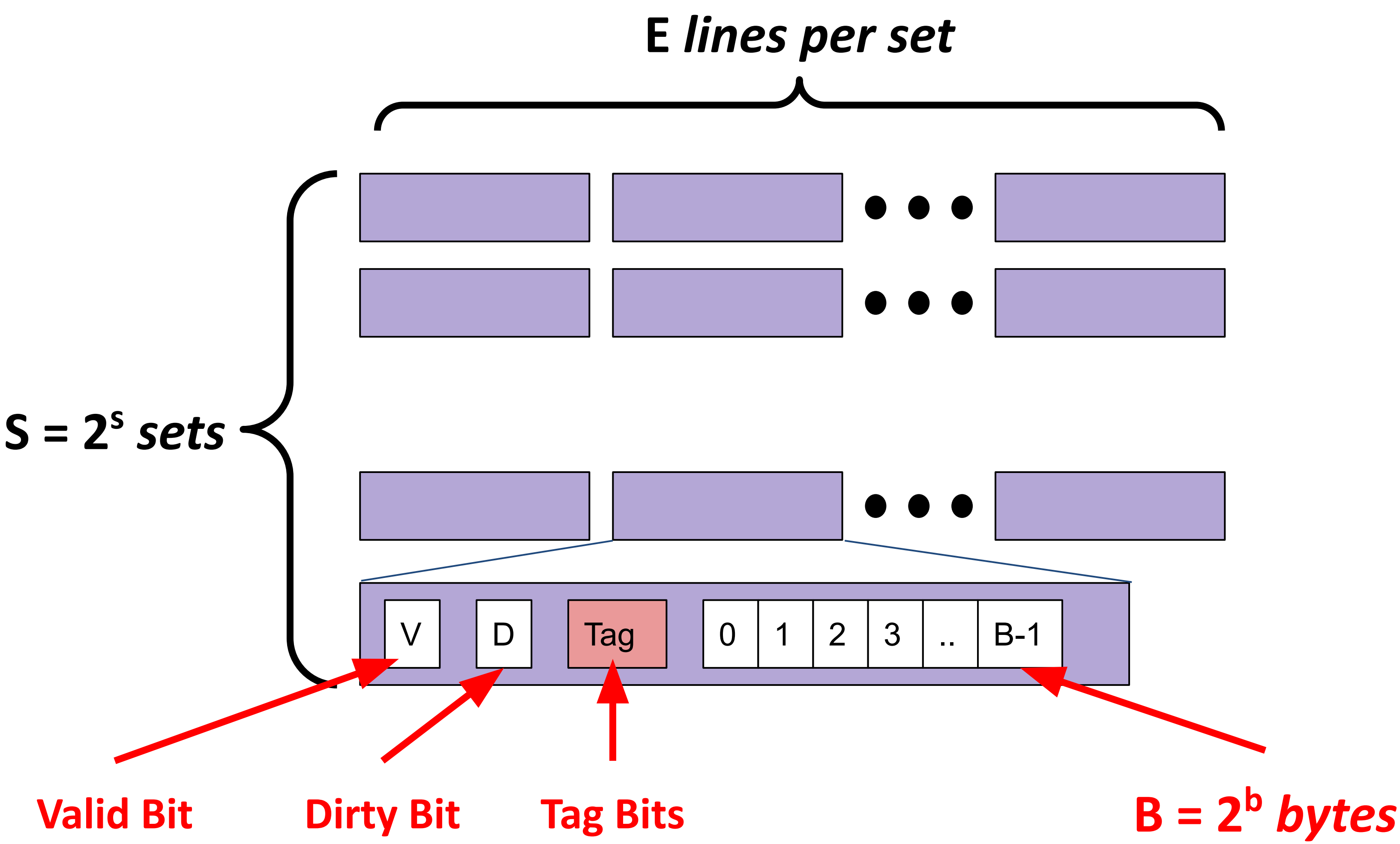
E – Number of *lines* per set



Cache Organization

- Each line stores data

b – Number of block offset *bits*
$B = 2^b$ – Block Size



Cache Concepts: Cache Read

- We have an address that we want to look up in our cache.

`0x00604420`

- How do we search for it? Which set? Which line?
- Our *parameters* (**`s`** and **`b`**) determine how we partition the bits of our address.

Cache Concepts: Cache Read

- Our *parameters* (**`s`** and **`b`**) determine how we partition the bits of our address.
- Suppose **`s`** = 6 and **`b`** = 6

`0b0000000011000000100010000100000`

Remaining bits are *tag bits*

6 bits for *set index*

6 bits for *block offset*

Cache Concepts: Cache Read

Tag: 000000001100000100

Set: 010000

Block Offset: 100000

- These bits now tell us how to do the lookup in our cache!
- Use set index (0b010000 = 16) to select the set
- Loop through lines in that set to find a matching tag (0b000000001100000100)
- If found and valid bit is set: ***Hit!***
  - Locate data starting at byte offset (0b100000)

Cache Concepts: Cache Miss

- But what happens if the cache doesn't have our data?
- We have a *cache miss*
- If we have a free line in the set, just load data into there
- Otherwise, the set is full!
  - We have to *evict* a line according to some *replacement policy*.
  - **`cachelab`**: LRU (Least Recently Used)
  - Other policies exist!
  - Finally, load our new line into the free slot.

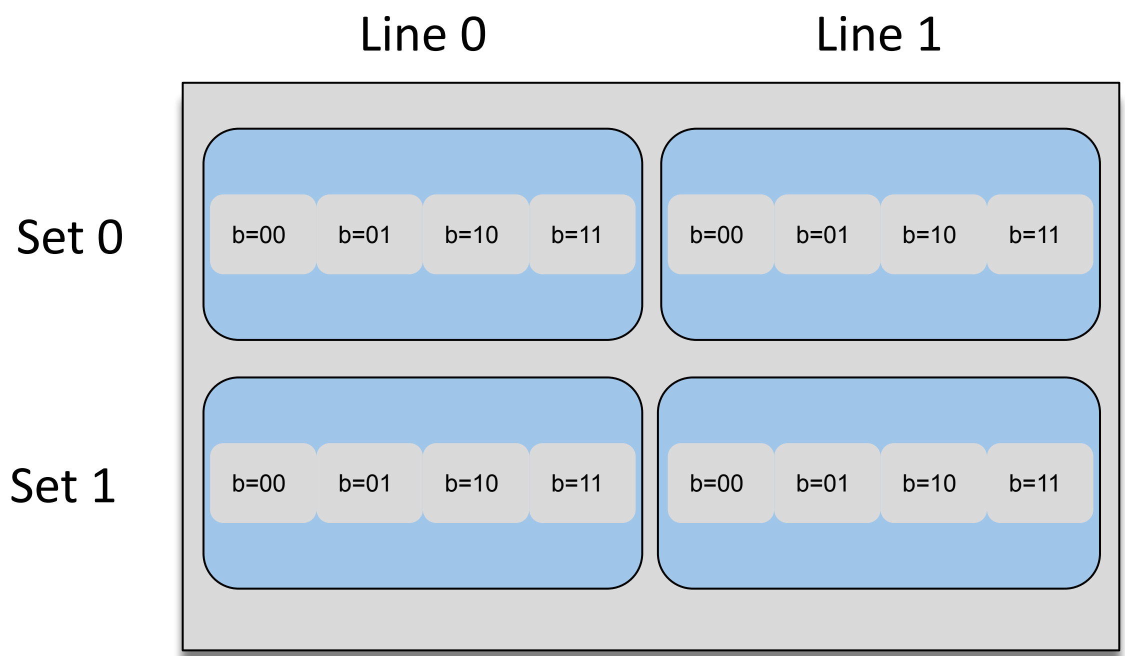
Cache Concepts: Dirty Bit

- You will implement a ***write-back, write-allocate*** policy for `cachelab`.
- *Write-Allocate:* Writes load the line into cache, update it in place.
- *Write-Back*: Defer writing updates to memory until line is evicted.
  - Expensive to flush every evicted line to memory.
  - ***Dirty bit*** indicates whether cache line has been written to, and needs to be flushed to memory.

Example Trace

Example Trace

- We will use the following configuration:
  - **s** = 1
  - **E** = 2
  - **b** = 2



Example Trace: Reading a Trace

```
bpr.trace
L 0,1
L 0,1
L 1,1
S 2,1
L 5,1
L 4,1
L 8,1
L 0,1
L 16,1
L 9,1
L 24,1
L 32,1
L 0,1
```

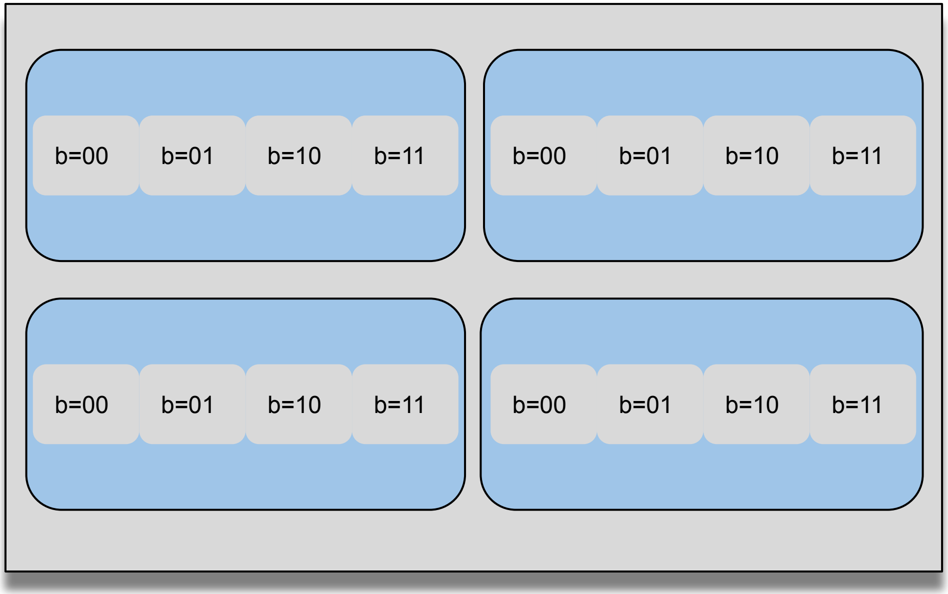
- `op <Addr>, <Size>`
- `op`:
  - `L` – Load
  - `S` – Store

Example Trace

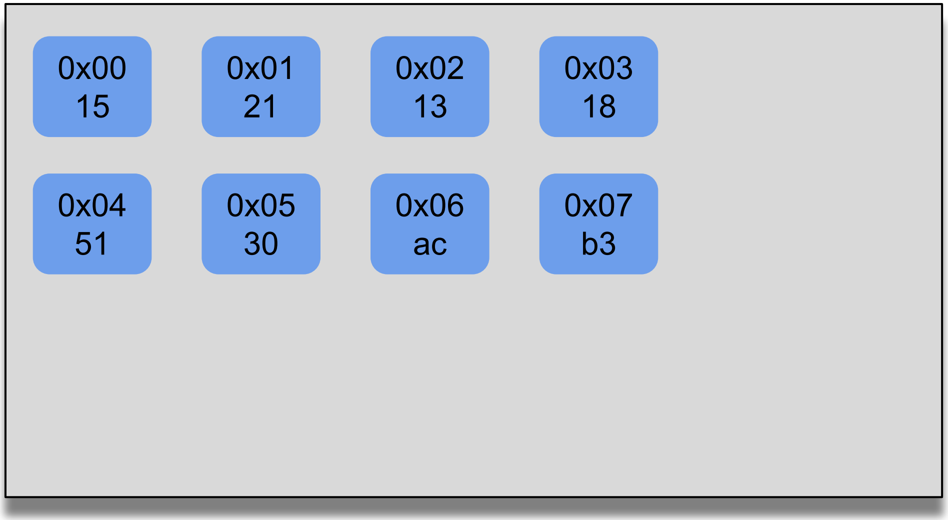
```
  . . .
→ L 0,1
  . . .
```

Will this instruction result in a hit or a miss?

Cache



Memory

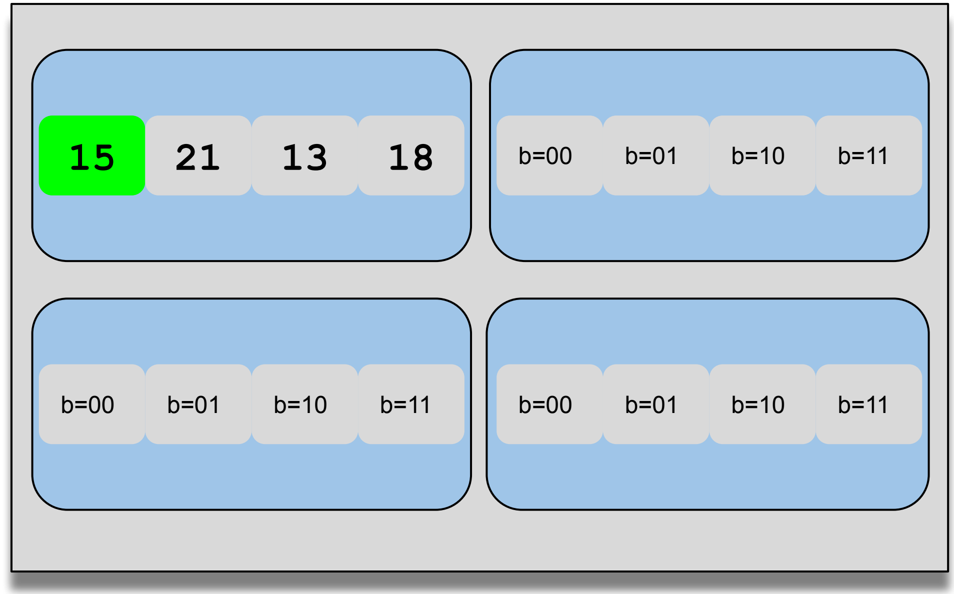


Example Trace

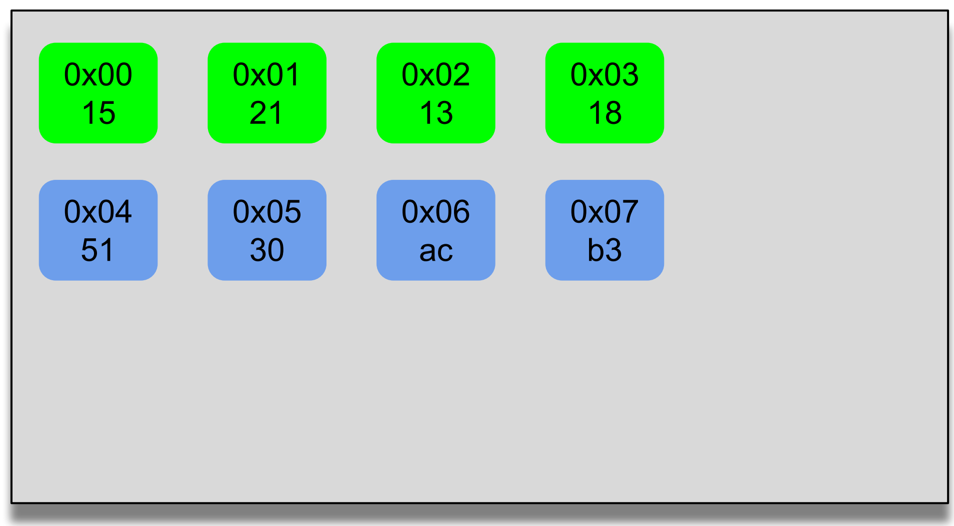
```
. . .
L 0,1
. . .
```

Miss

Cache



Memory



Why that line?

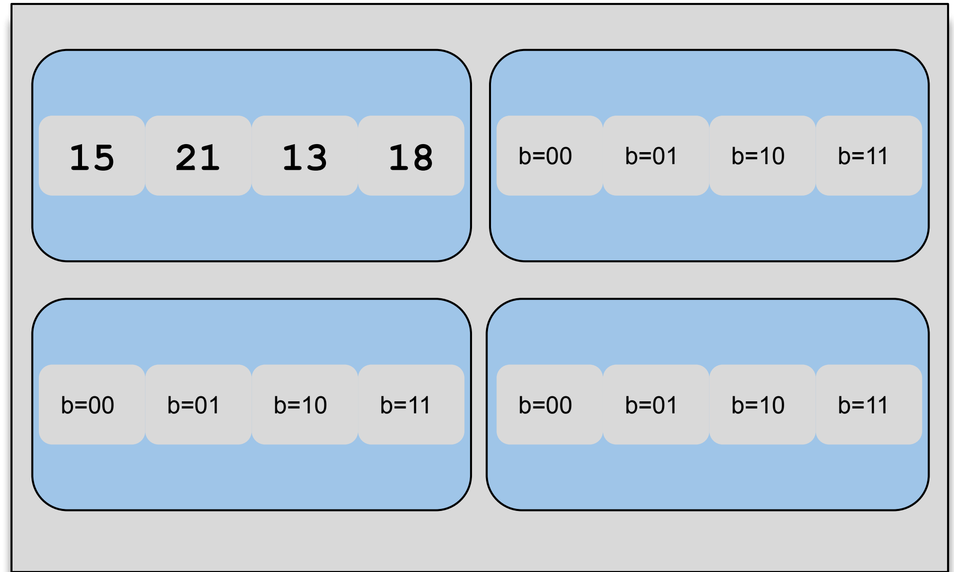
Where are those values from?

What kind of miss is this?

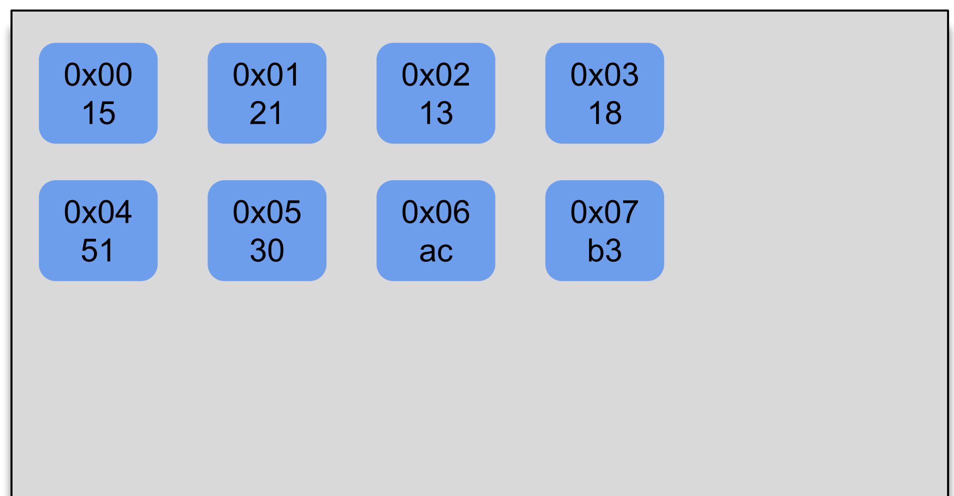
Example Trace

```
. . .
L 0,1    Miss
L 0,1    ???
. . .
```

Cache



Memory



Will this instruction result in a hit or a miss?

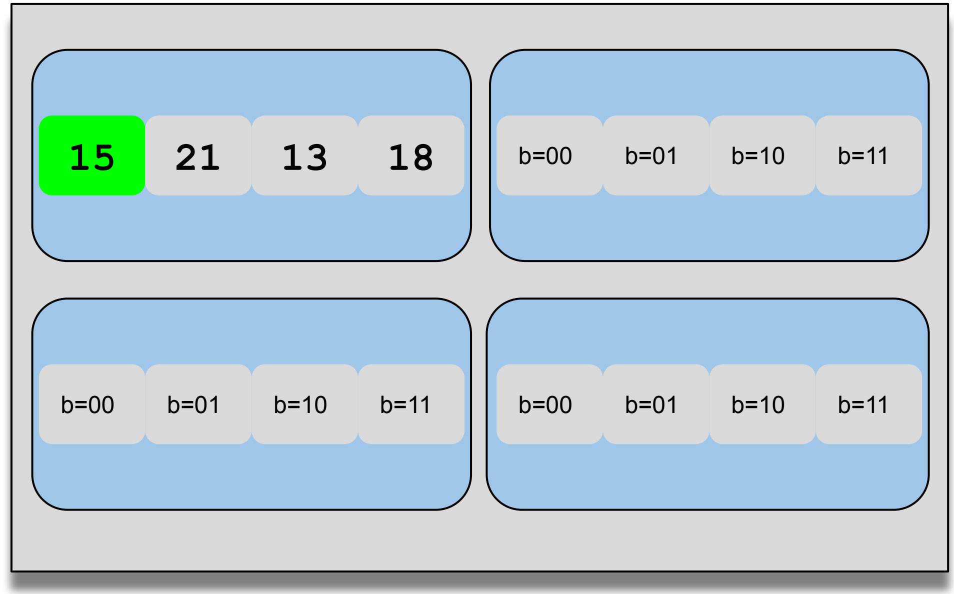
Example Trace

```
...
L 0,1
L 0,1
...
```

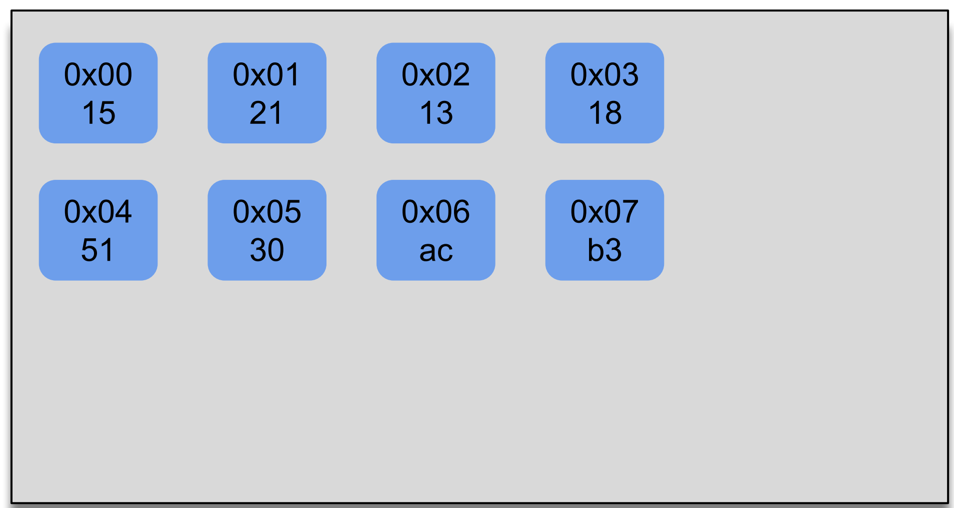
Miss

Hit!

Cache



Memory

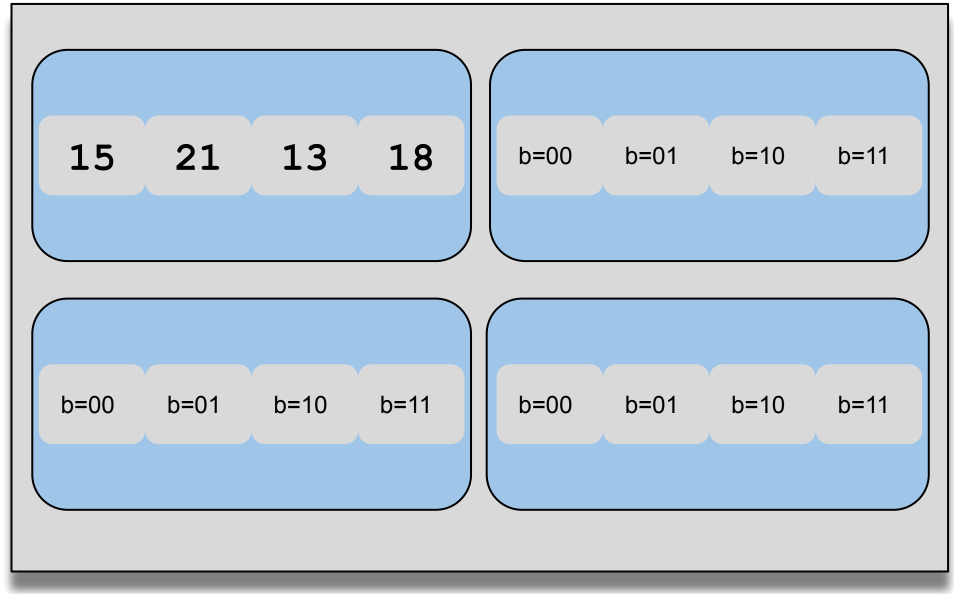


Example Trace

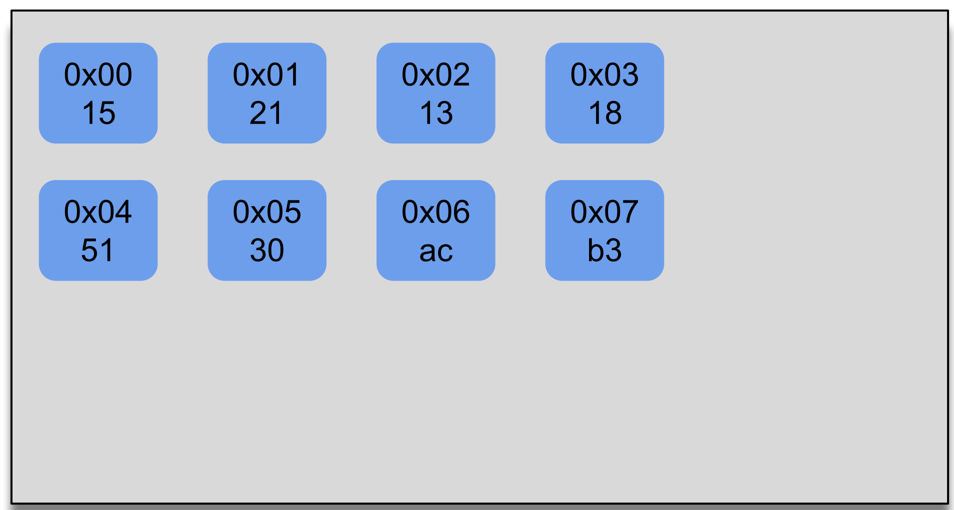
```
...
L 0,1    Miss
L 0,1    Hit!
L 1,1    ???
...
```

Will this instruction result in a hit or a miss?

Cache



Memory



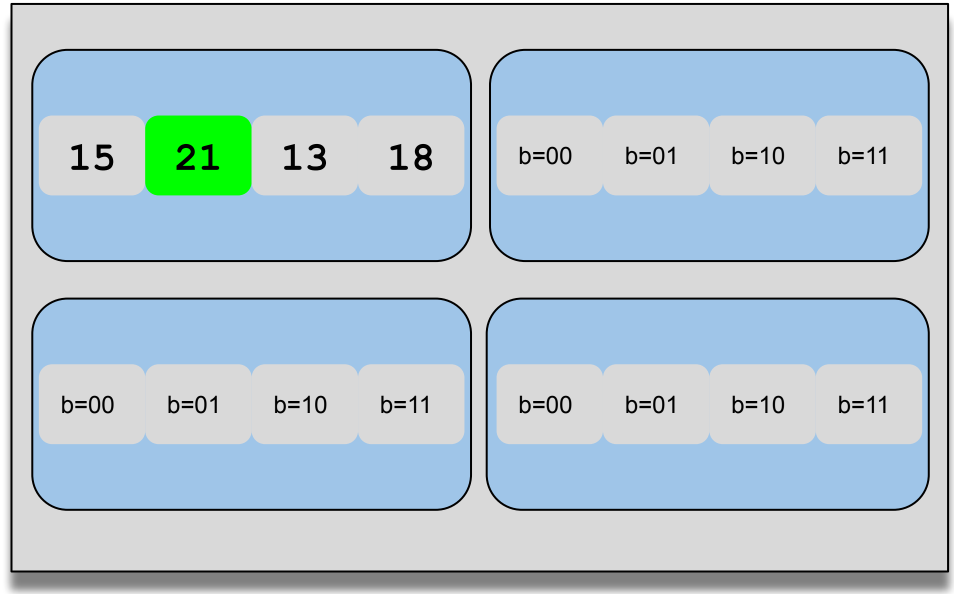
Example Trace

```
...
L 0,1    Miss
L 0,1    Hit!
L 1,1    Hit!
...
```

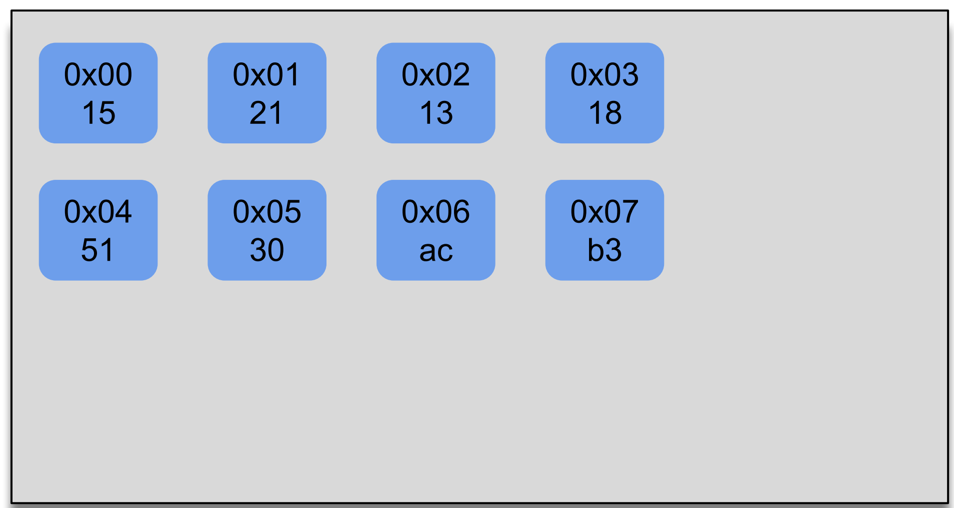
Not a miss!

We had already loaded all four bytes of the line into cache. Why?

Cache



Memory

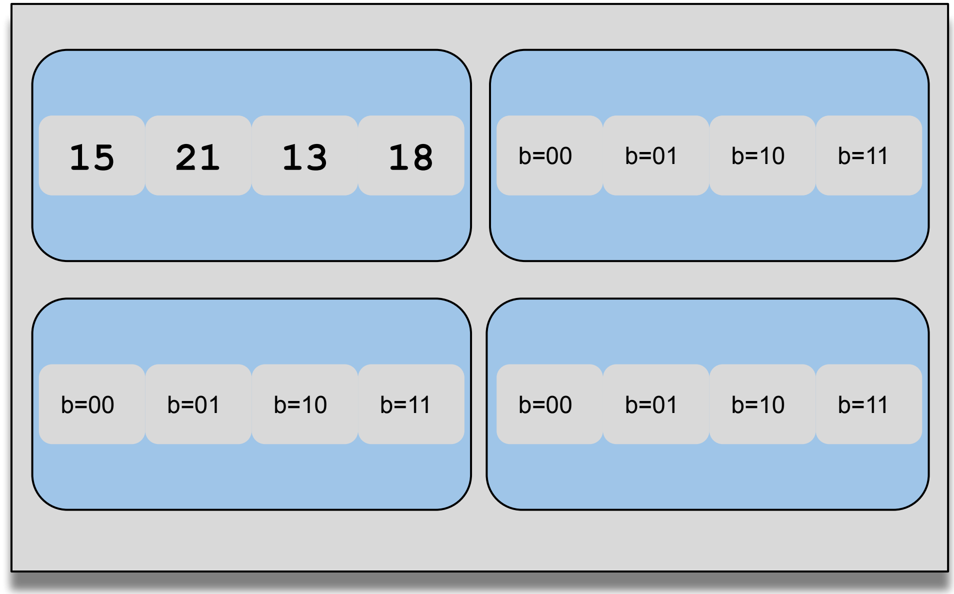


Example Trace

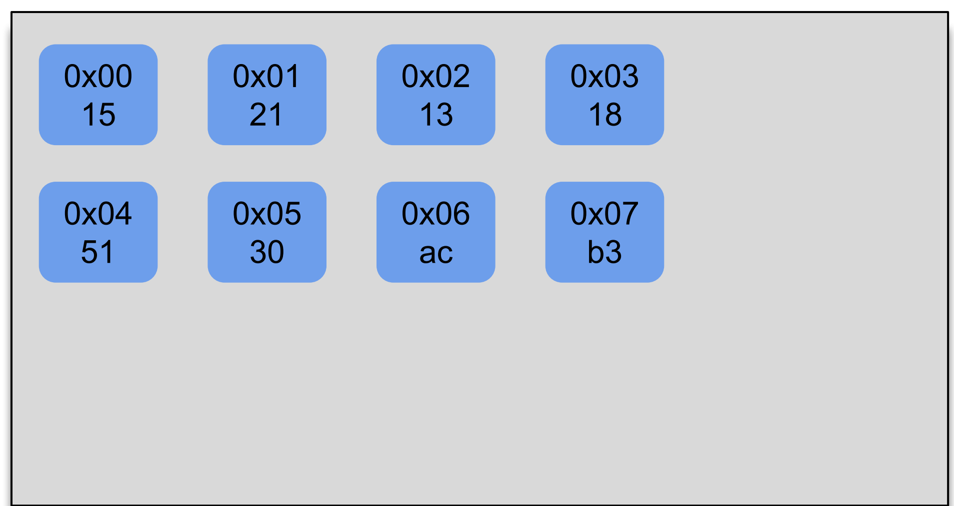
```
. . .
L 0,1    Hit!
L 1,1    Hit!
S 2,1    ???
. . .
```

Will this instruction result in a hit or a miss?

Cache



Memory



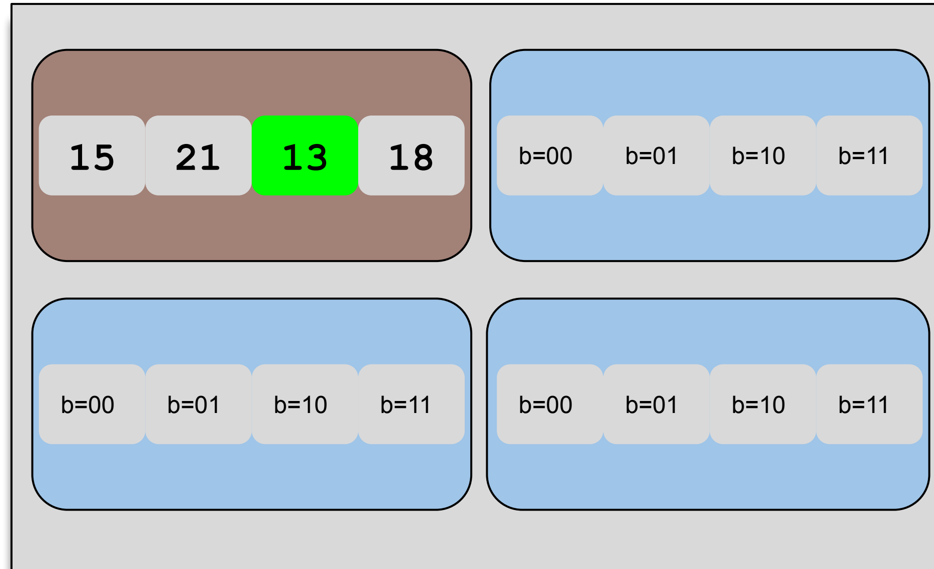
Example Trace

```
...
L 0,1    Hit!
L 1,1    Hit!
S 2,1    Hit!
...
```

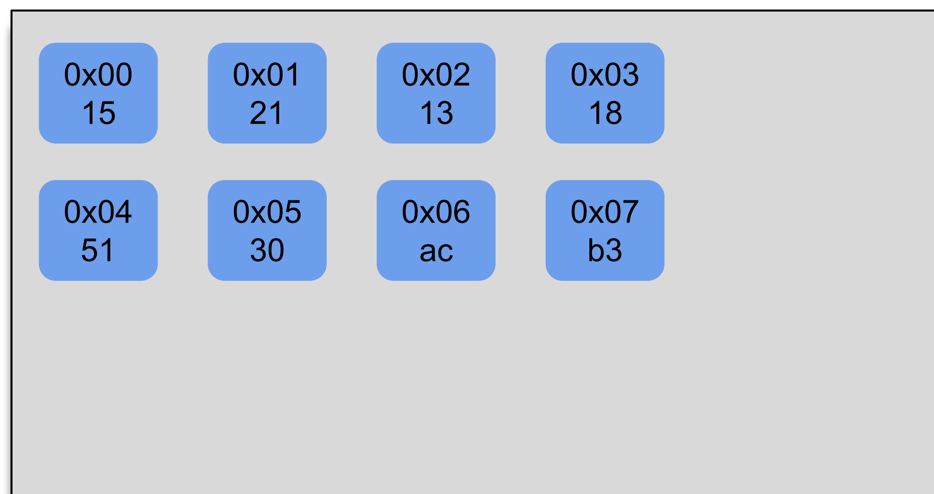
Write hit!

Set dirty bit.

Cache



Memory

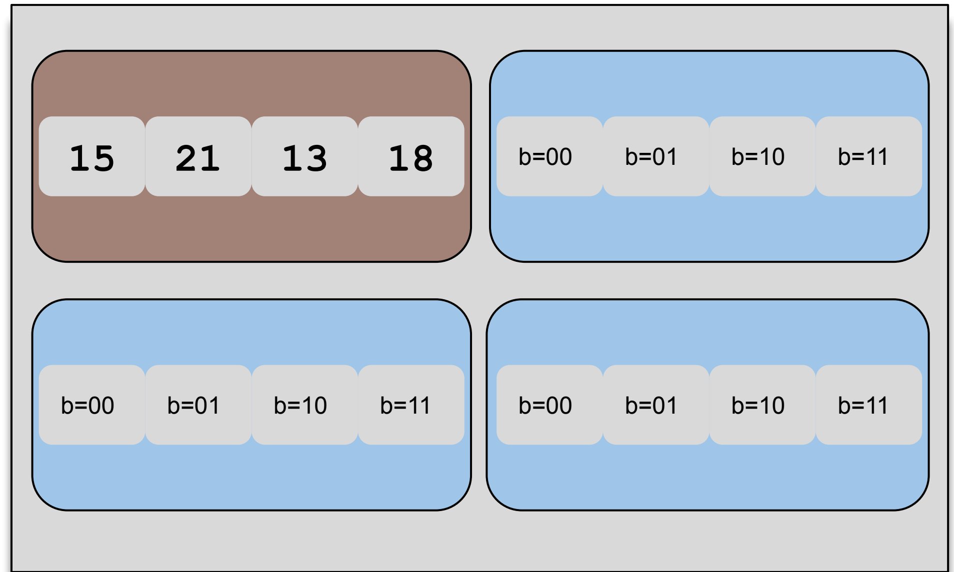


Example Trace

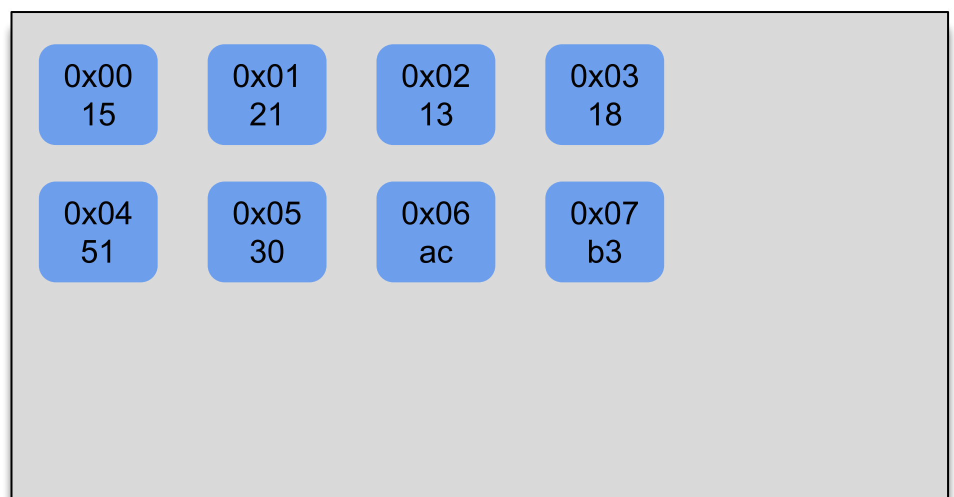
```
. . .
L 1,1    Hit!
S 2,1    Hit!
L 5,1    ???
. . .
```

Will this instruction result in a hit or a miss?

Cache



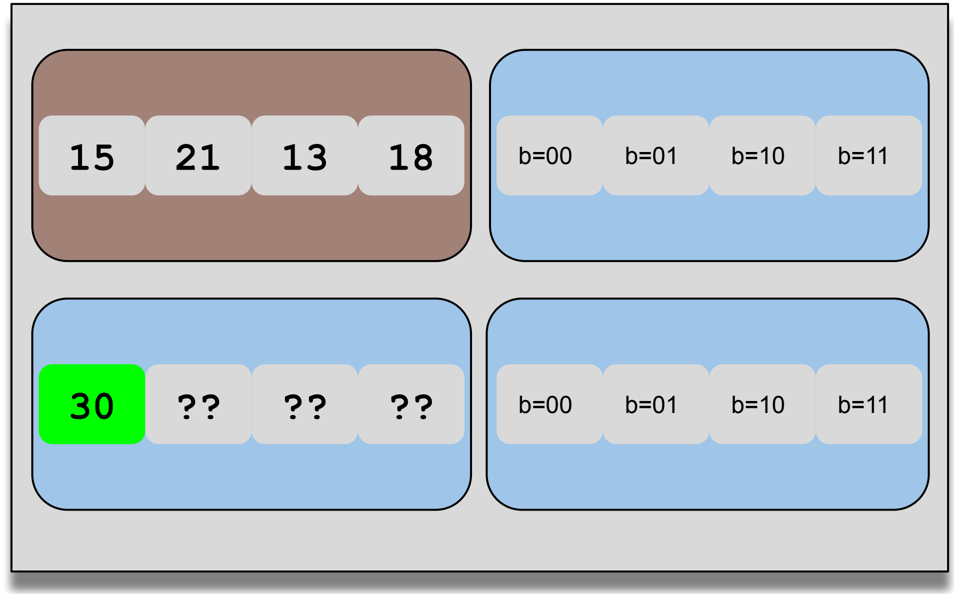
Memory



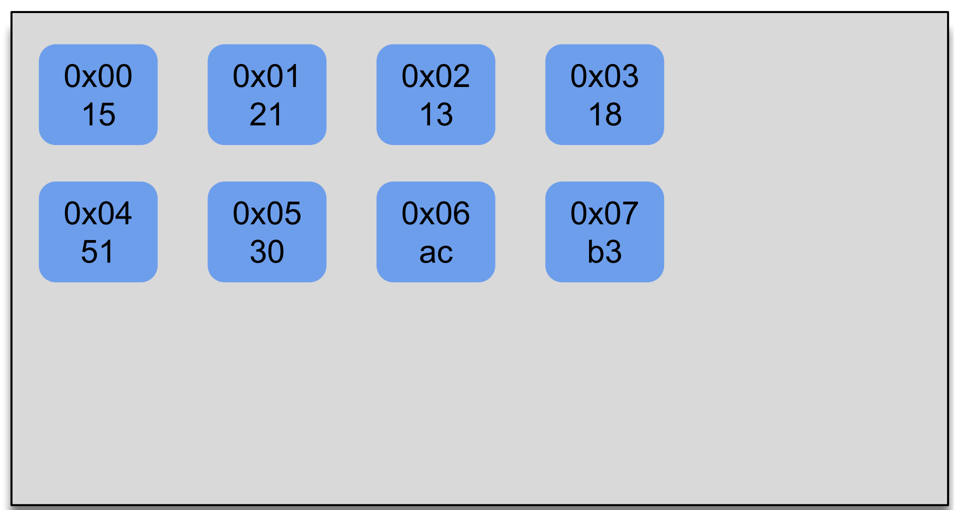
Example Trace

```
...
L 1,1    Hit!
S 2,1    Hit!
L 5,1    Miss
...
```

Cache



Memory



Do we load just one byte like this?

Example Trace

```
  . . .
  L 1,1    Hit!
  S 2,1    Hit!
→ L 5,1    Miss
  . . .
```

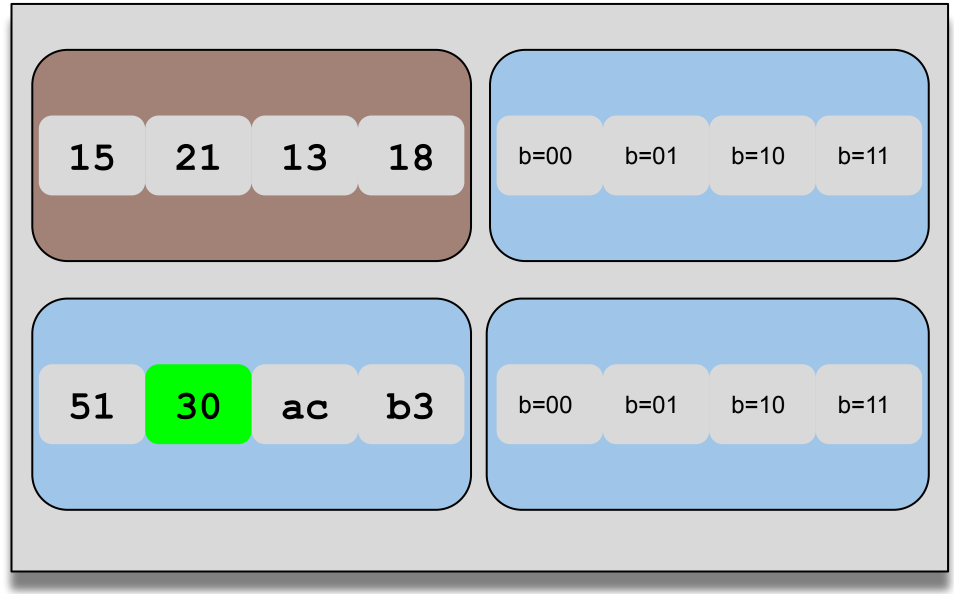
Do we load just one byte like this?

No!

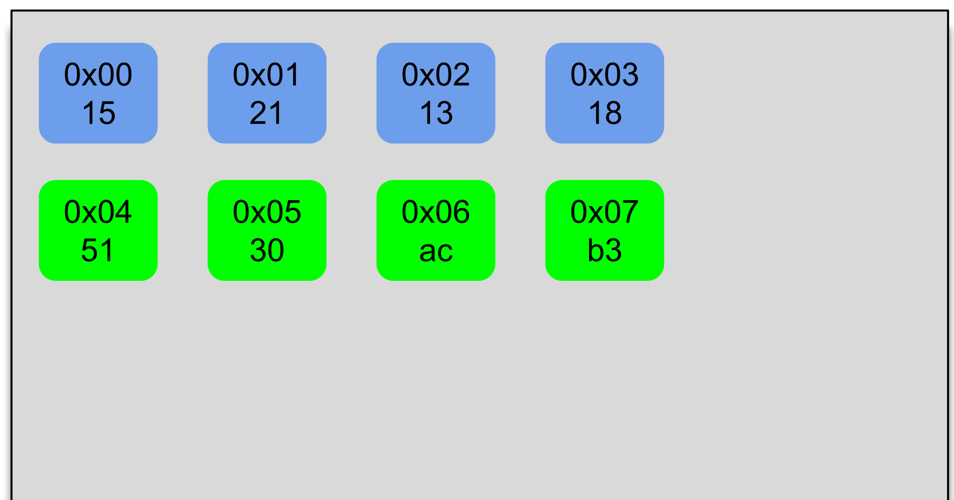
Example Trace

```
...
L 1,1    Hit!
S 2,1    Hit!
L 5,1    Miss
...
```

Cache



Memory

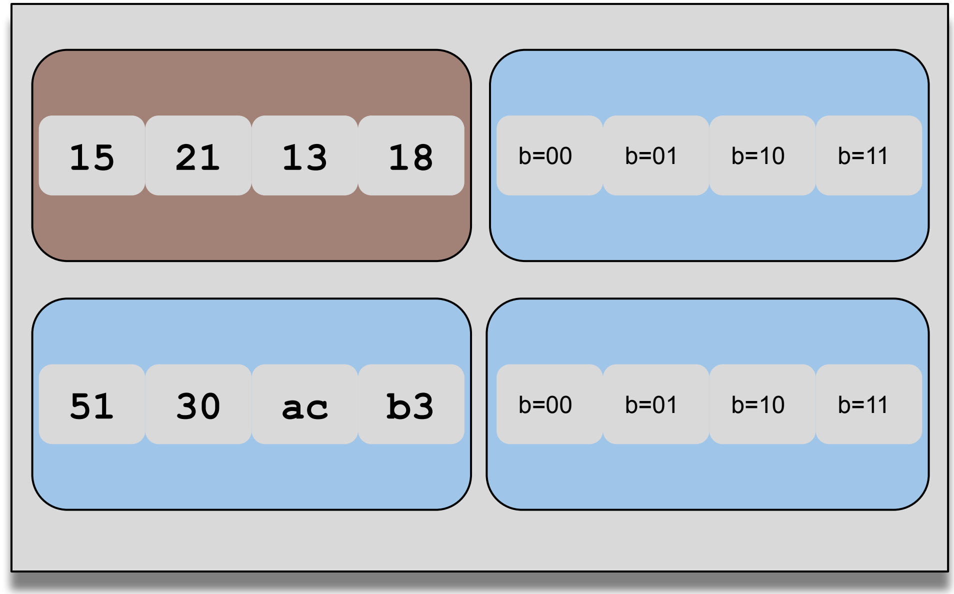


Why do we start with a byte from below address 5?

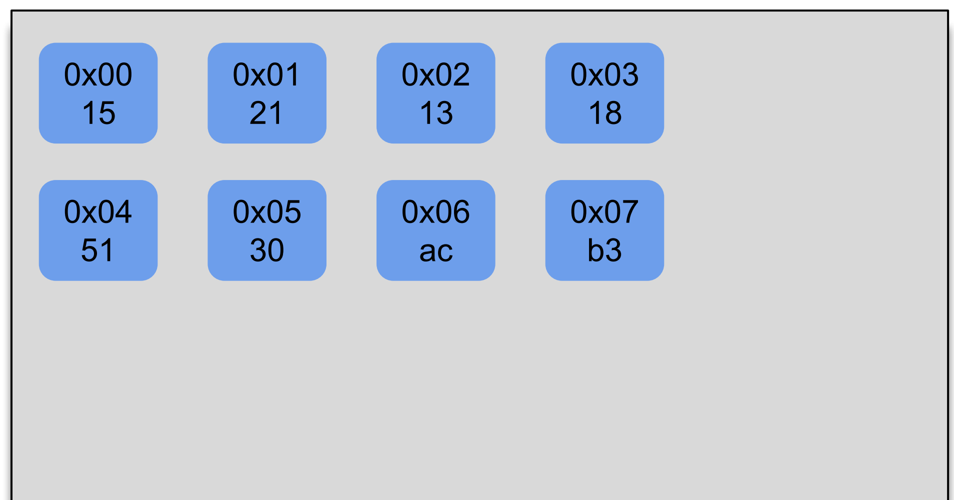
Example Trace

```
...
S 2,1    Hit!
L 5,1    Miss
L 4,1    ???
...
```

Cache



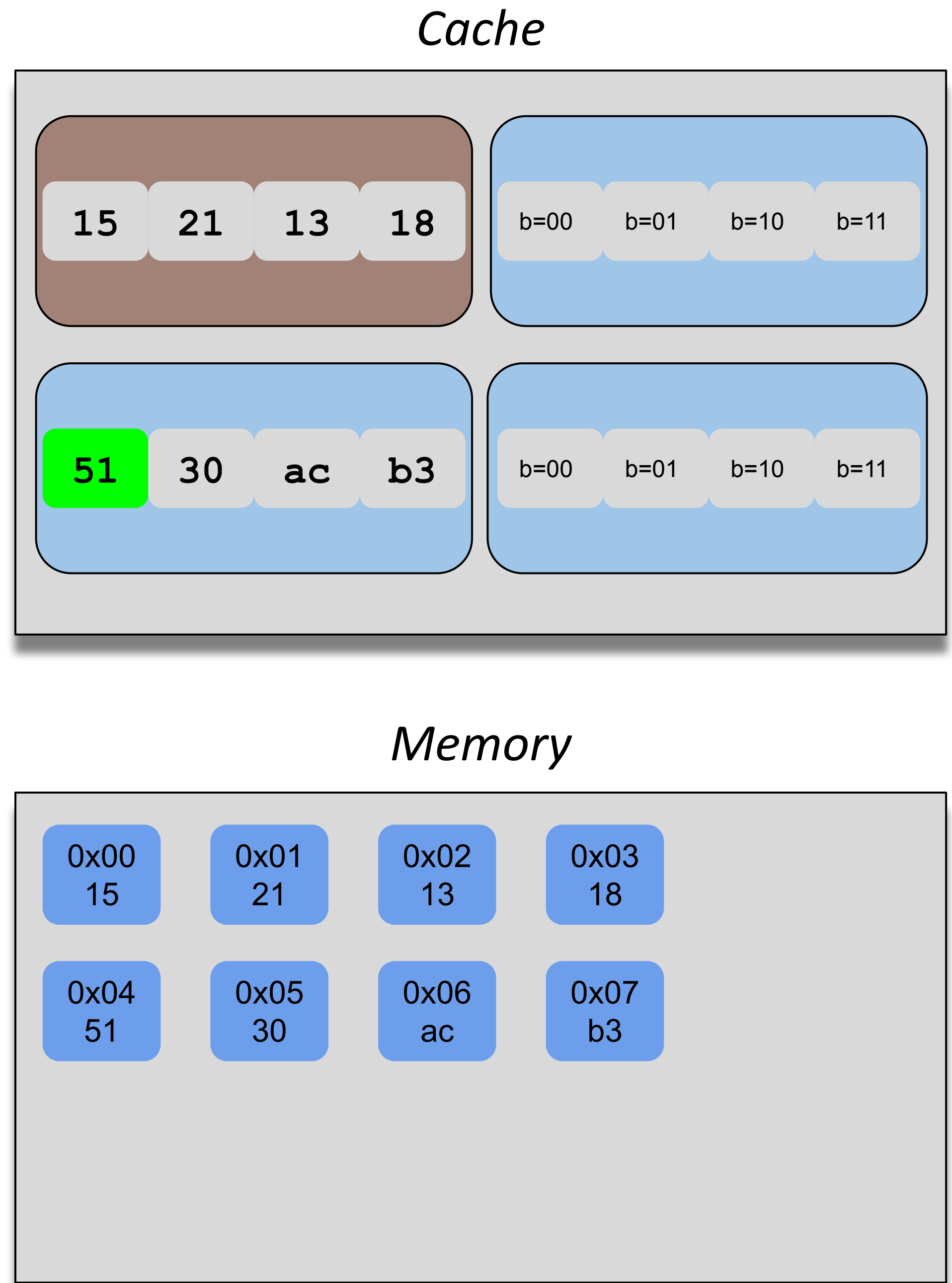
Memory



Will this instruction result in a hit or a miss?

Example Trace

```
. . .
S 2,1    Hit!
L 5,1    Miss
L 4,1    Hit!
. . .
```

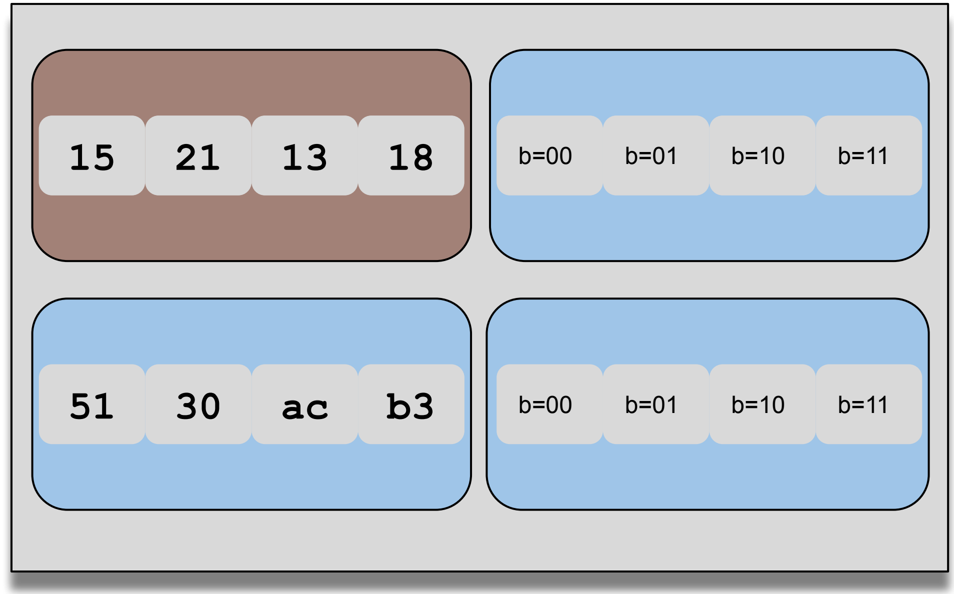


Example Trace

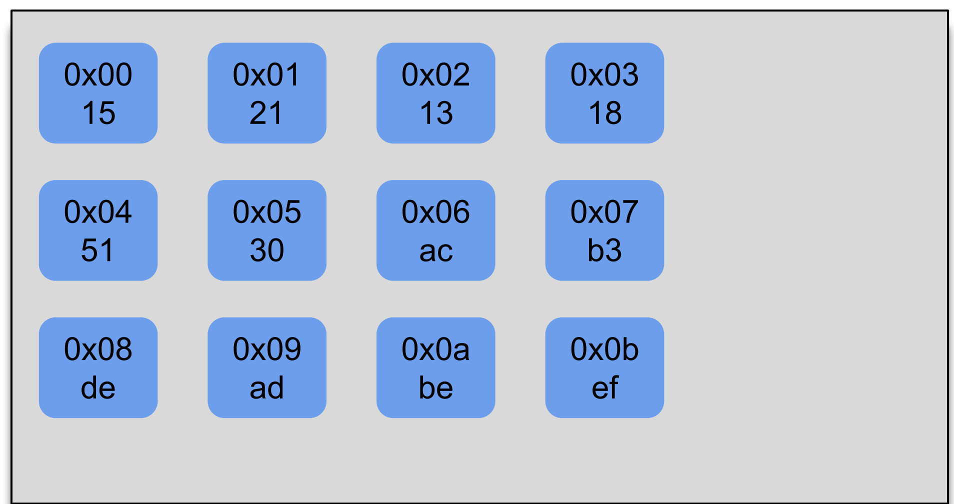
```
...
L 5,1    Miss
L 4,1    Hit!
L 8,1    ???
...
```

Will this instruction result in a hit or a miss?

Cache



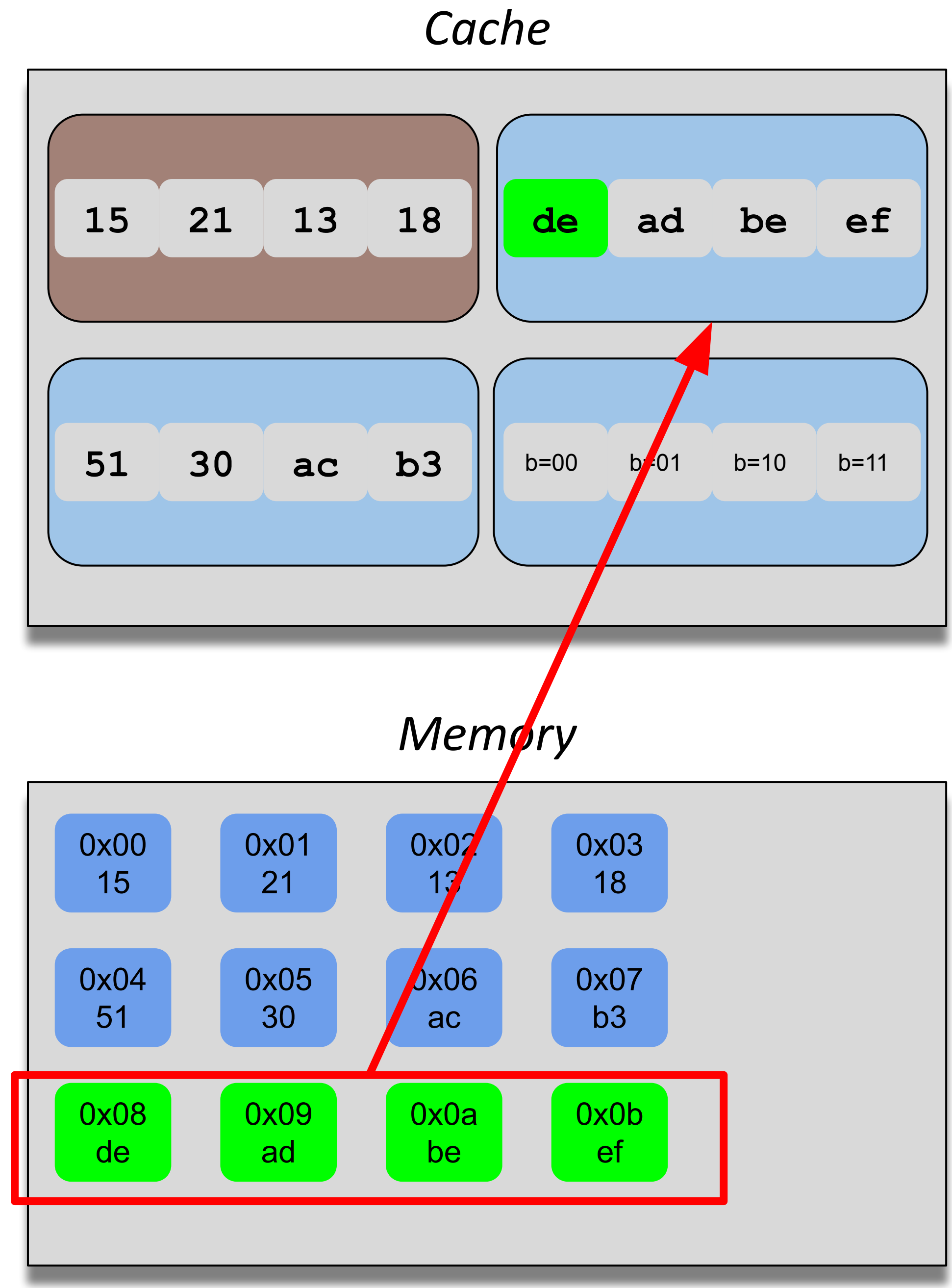
Memory



Example Trace

```
...
L 5,1
L 4,1
L 8,1
...
```

```
Miss
Hit!
Miss
```



Miss!

We had a free line, so just load the data into there.

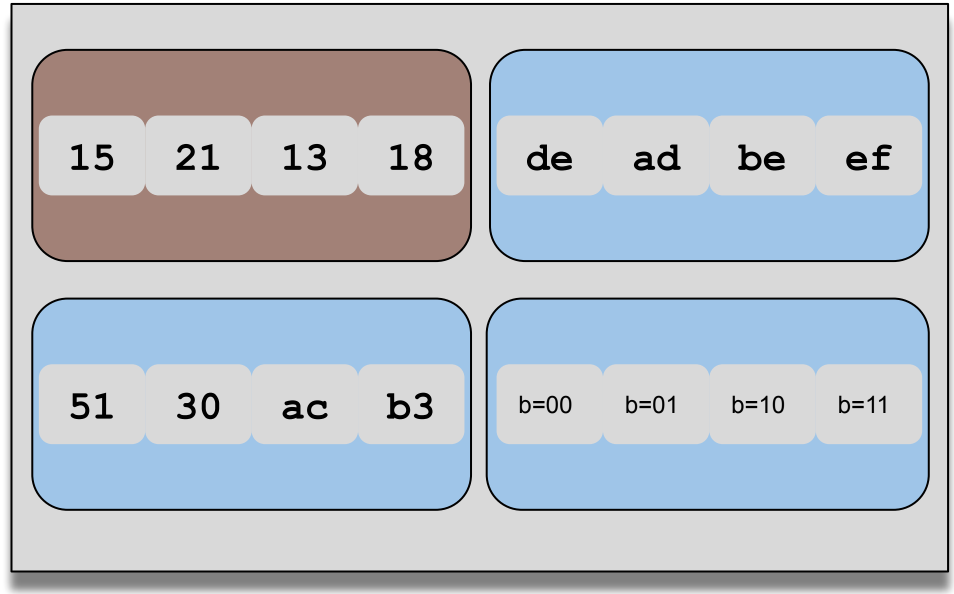
Example Trace

```
...
L 4,1
L 8,1
L 0,1
...
```

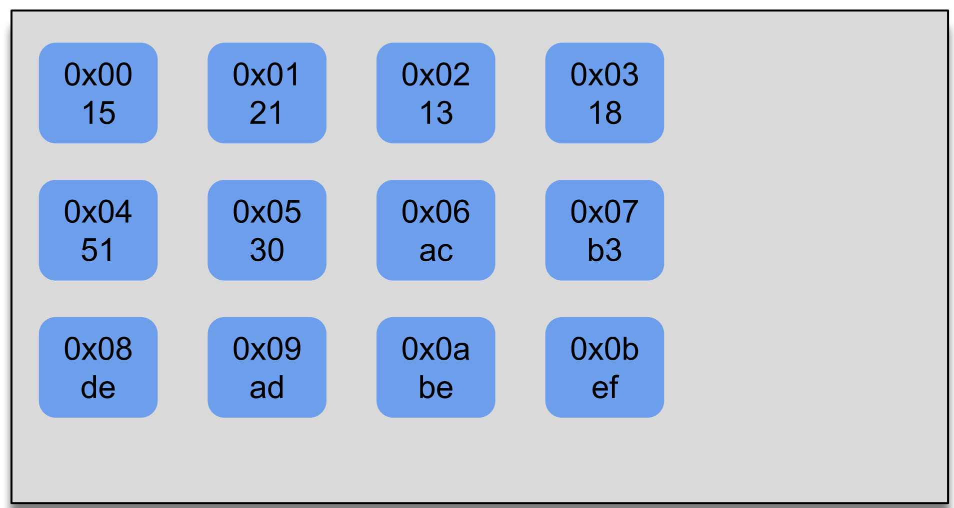
Hit!
Miss
???

Will this instruction result in a hit or a miss?

Cache



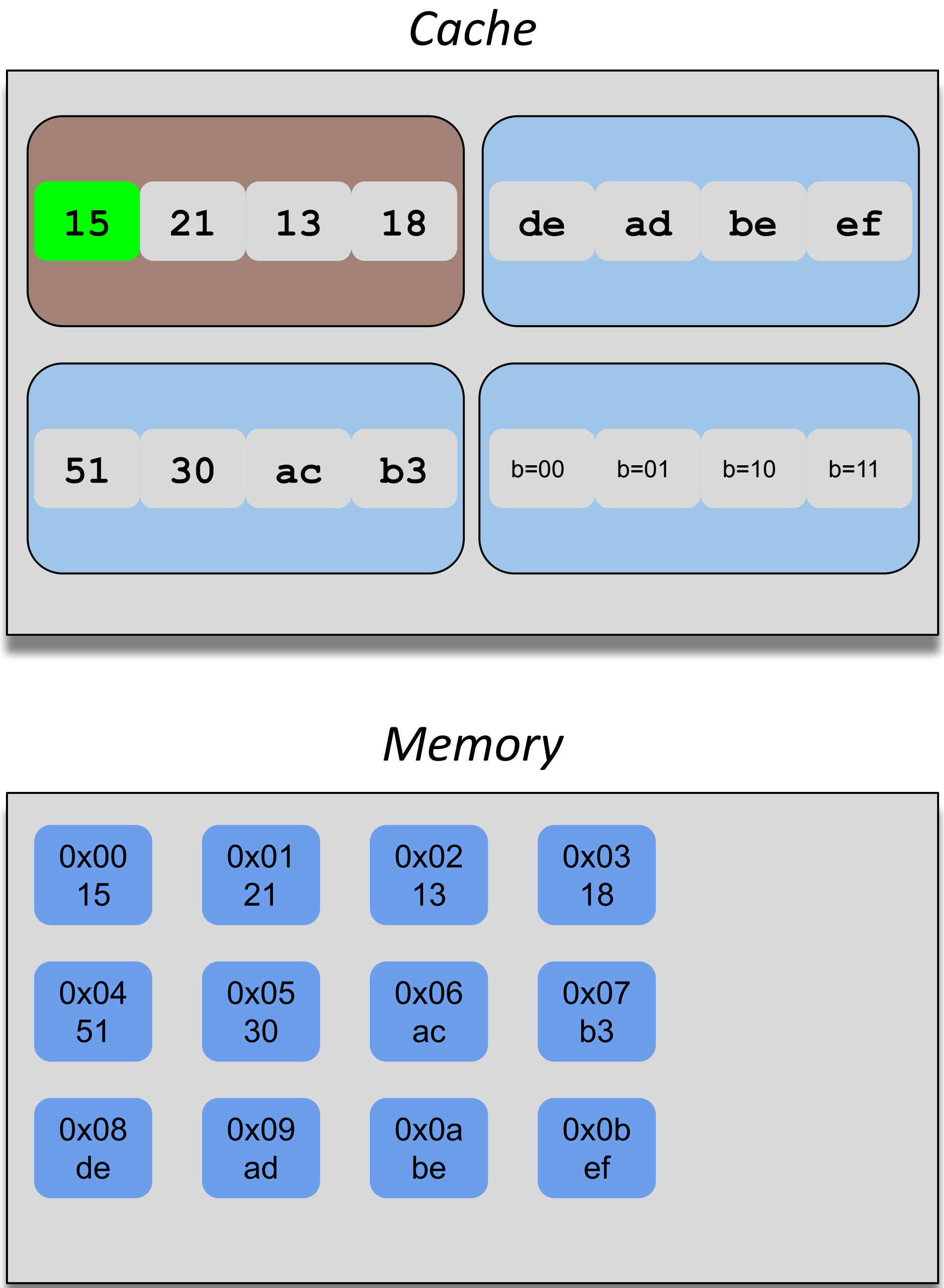
Memory



Example Trace

```
. . .
L 4,1
L 8,1
L 0,1
. . .
```

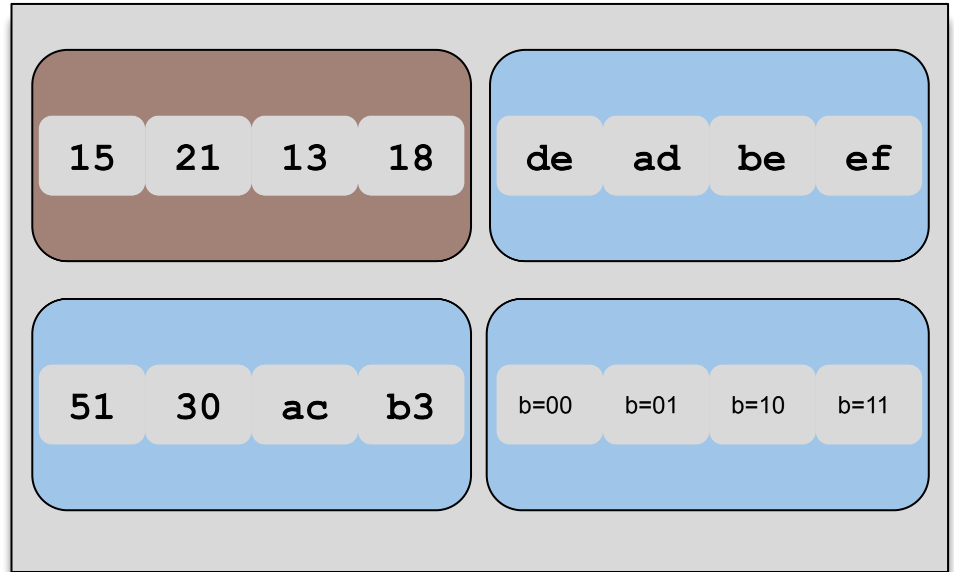
Hit!
Miss
Hit!



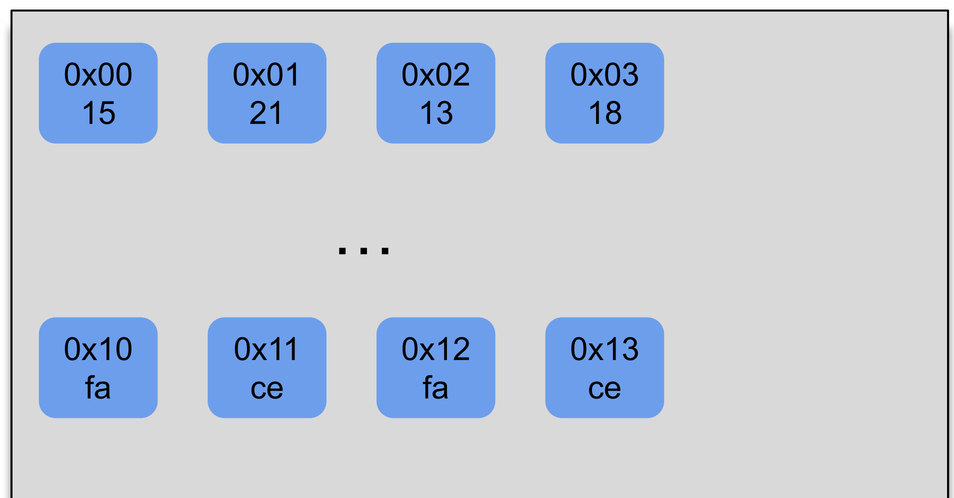
Example Trace

```
...
L 8,1     Miss
L 0,1     Hit!
L 16,1    ???
...
```

Cache



Memory



Will this instruction result in a hit or a miss?

Example Trace

```
...
L 8,1     Miss
L 0,1     Hit!
L 16,1    Miss
...
```

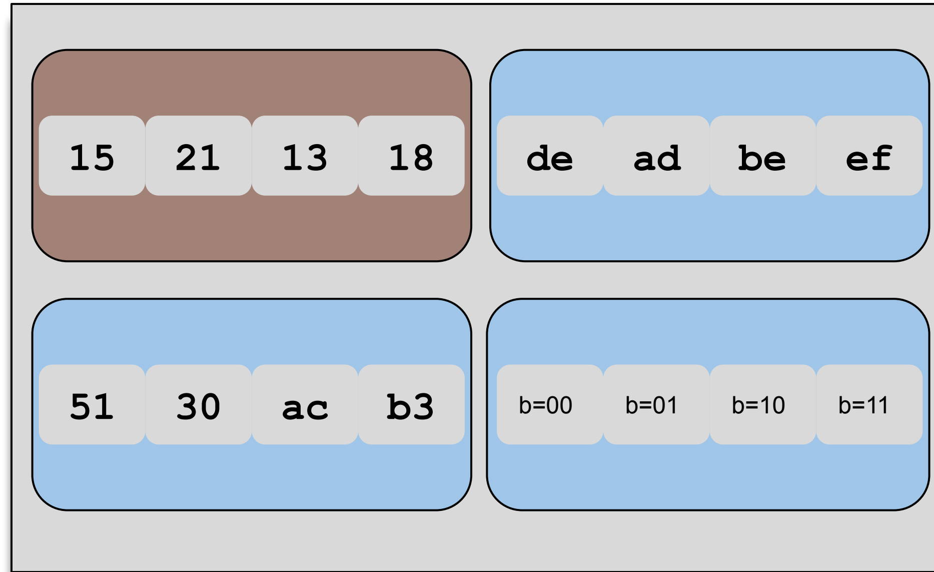
What kind of miss is this?

16 = 0b10000

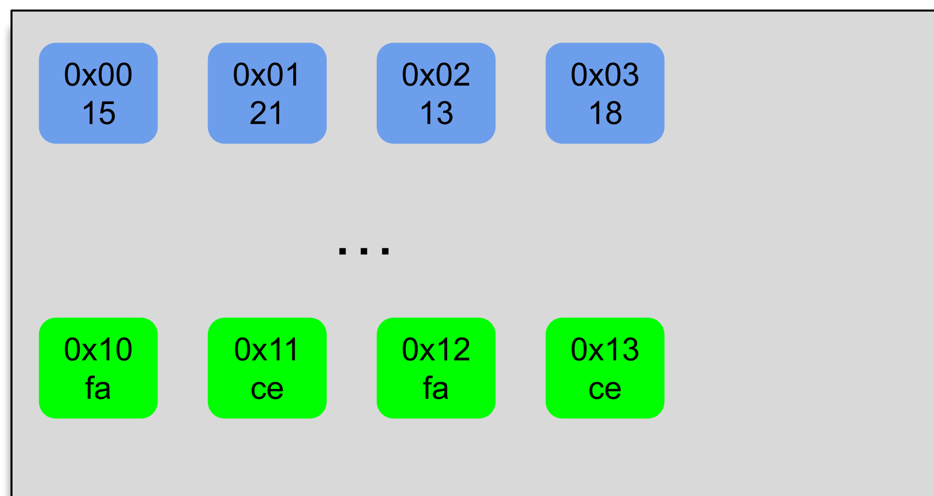
=> Set Index 0

=> Have to evict!

Cache

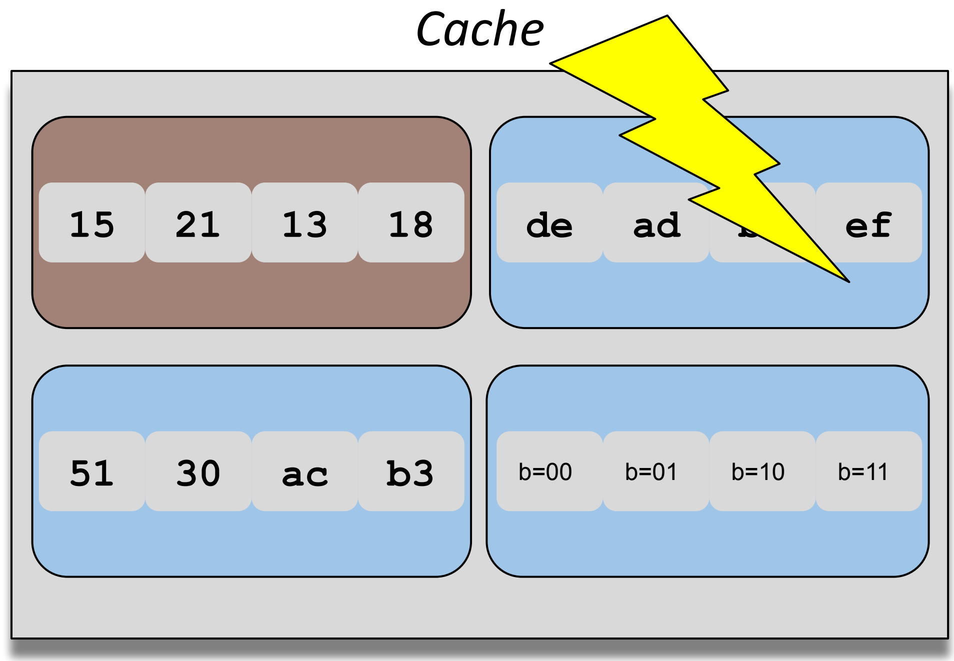


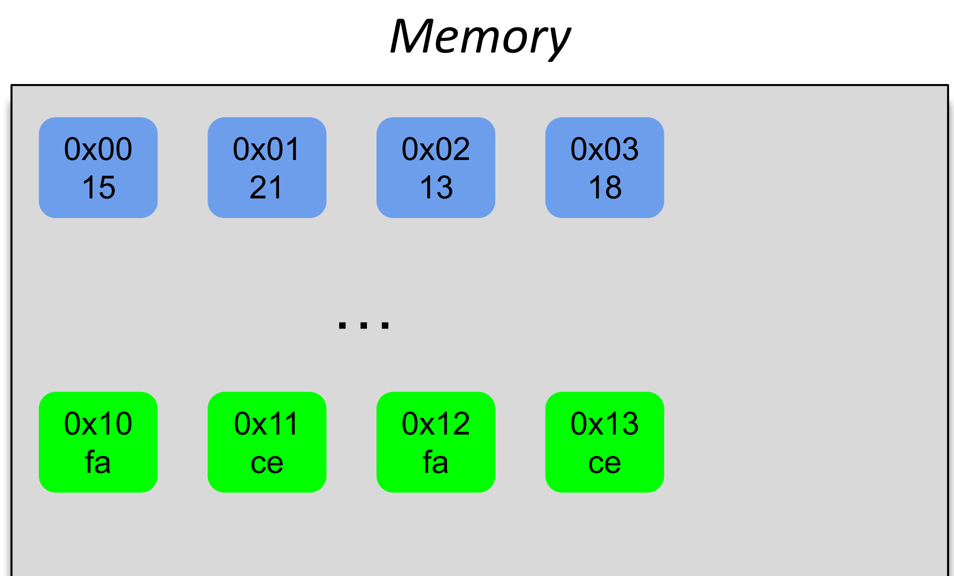
Memory



Example Trace

```
...
L 8,1     Miss
L 0,1     Hit!
L 16,1    Miss
...
```





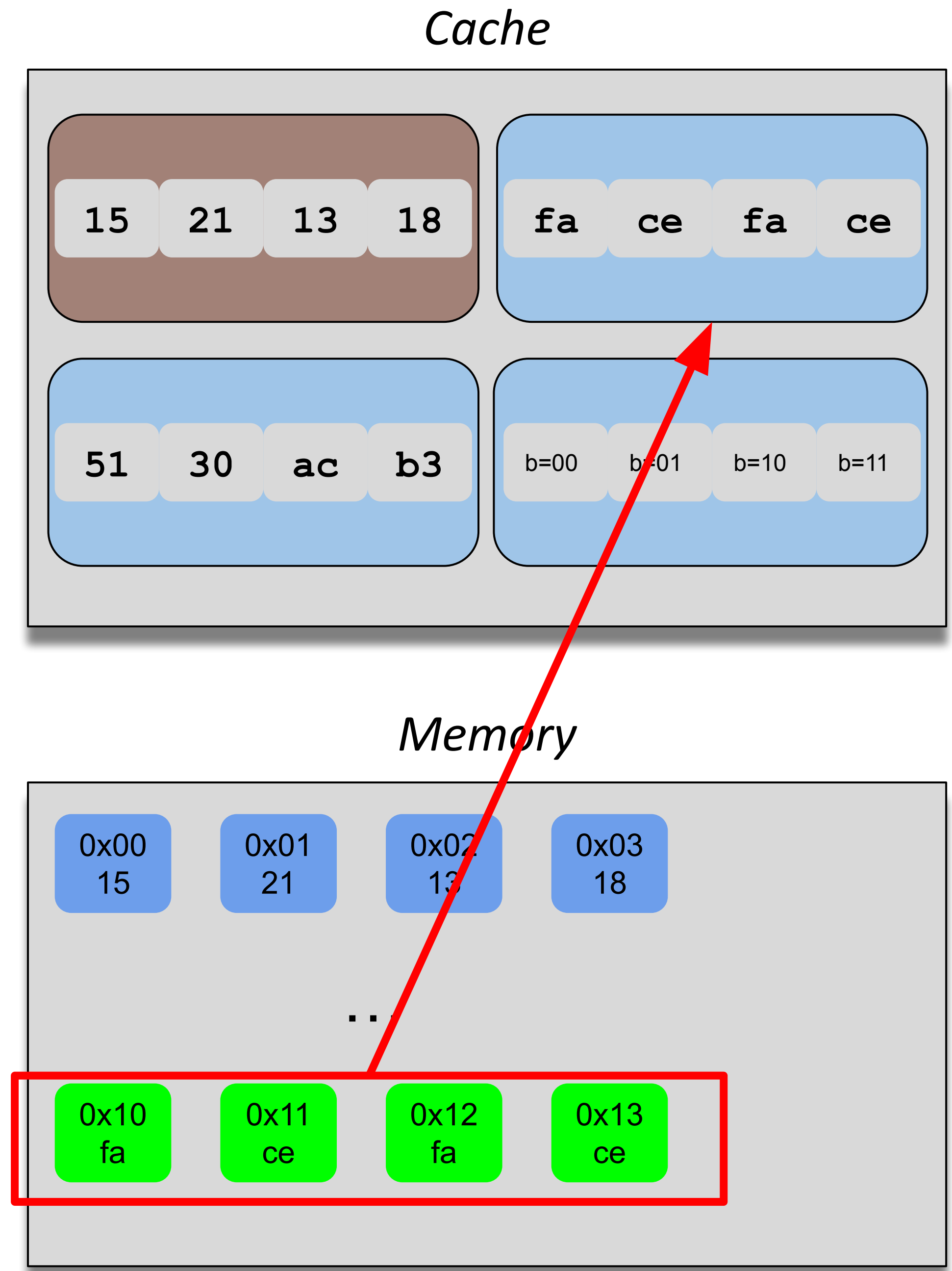
Cold Miss (first time seeing this block)

Evict LRU (Least Recently Used) line from set 0

Example Trace

```
...
L 8,1     Miss
L 0,1     Hit!
L 16,1    Miss
...
```

Load new data into line

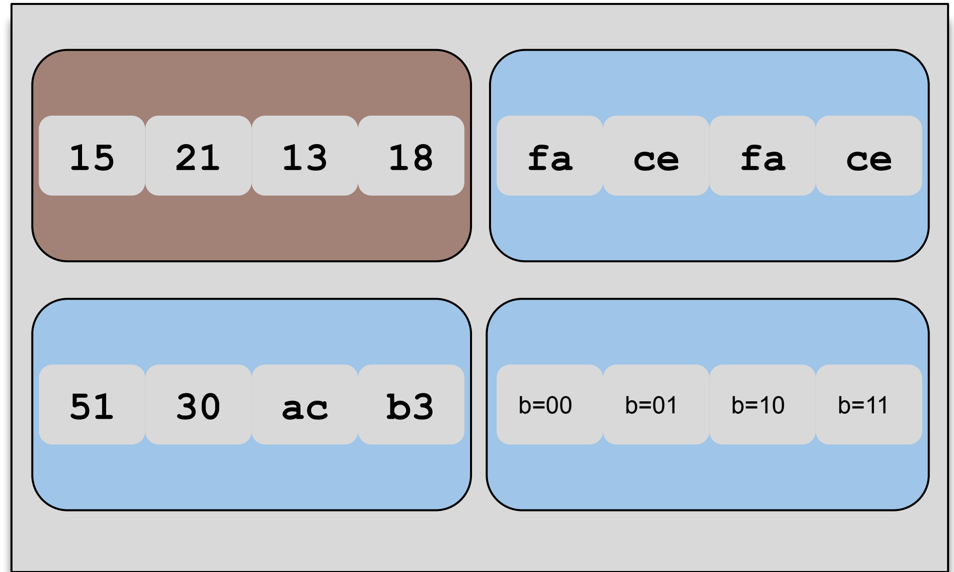


Example Trace

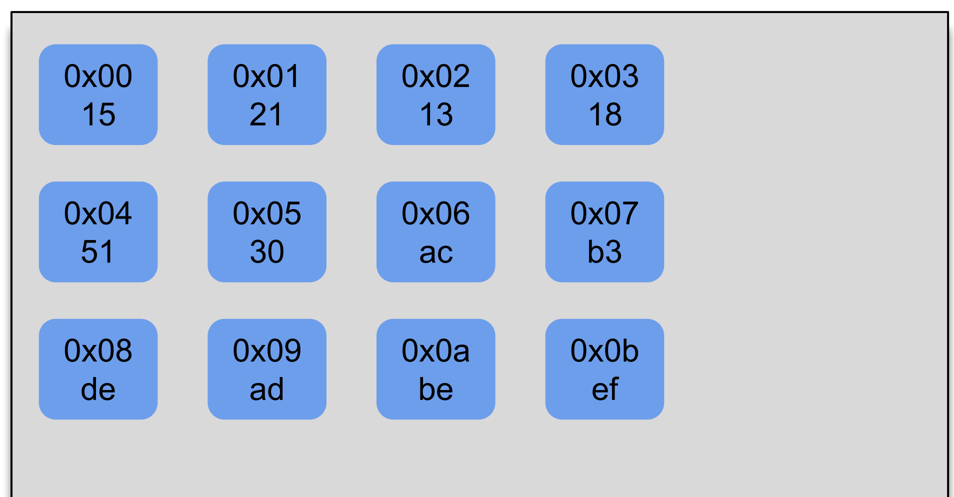
```
...
L 0,1
L 16,1
L 9,1
...
```

Hit!
Miss
???

Cache



Memory



Will this instruction result in a hit or a miss?

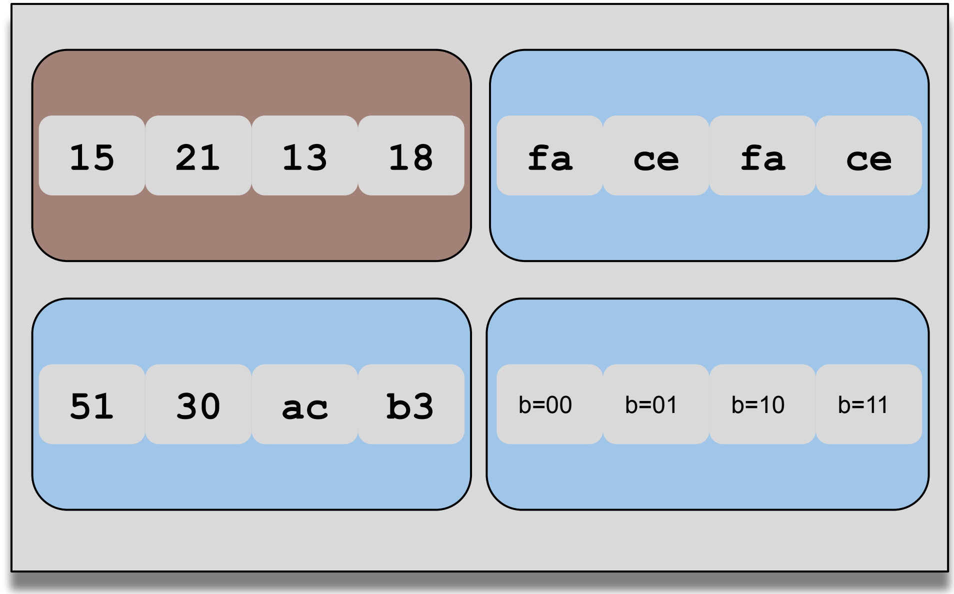
Example Trace

```
. . .
L 0,1     Hit!
L 16,1    Miss
L 9,1     Miss
. . .
```

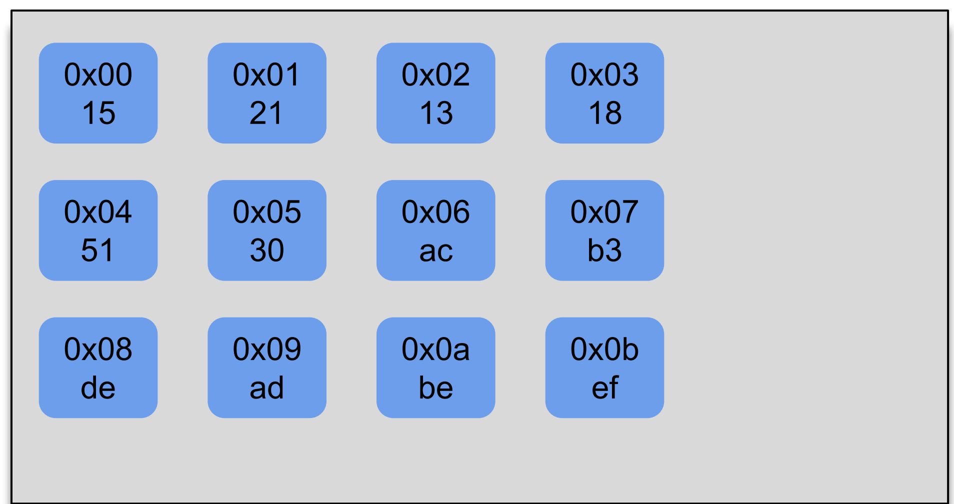
What kind of miss is this?

Has the block been in the cache before?

Cache



Memory



Cache Concepts: Conflict/Capacity Misses

```
L 0,1
L 0,1
L 1,1
S 2,1
L 5,1
L 4,1
L 8,1
L 0,1
L 16,1
L 9,1
...
```

- Has this block been in the cache before?
- Yes!
- If we've seen the block before:
  - Not a cold miss
  - Either a *conflict miss* or a *capacity miss*.

Cache Concepts: Conflict/Capacity Misses

```
L 0,1
L 0,1
L 1,1
S 2,1
L 5,1
L 4,1
L 8,1

L 0,1
L 16,1

L 9,1
...
```

How to distinguish between the two:

1. Find the last reference to that block in the trace.
2. Count the number of *unique* blocks referenced *in-between*:
   a. If the number is greater than or equal to the total number of lines in the cache: ***Capacity Miss***
   b. Otherwise: ***Conflict Miss***

Cache Concepts: Conflict/Capacity Misses

```
L 0,1
L 0,1
L 1,1
S 2,1
L 5,1
L 4,1
L 8,1

L 0,1
L 16,1

L 9,1
...
```

- In this case:
  - *Two* unique blocks in between current reference and last reference.
  - But we have *four* total lines in the cache
  - So we have a ***Conflict Miss***.

Example Trace

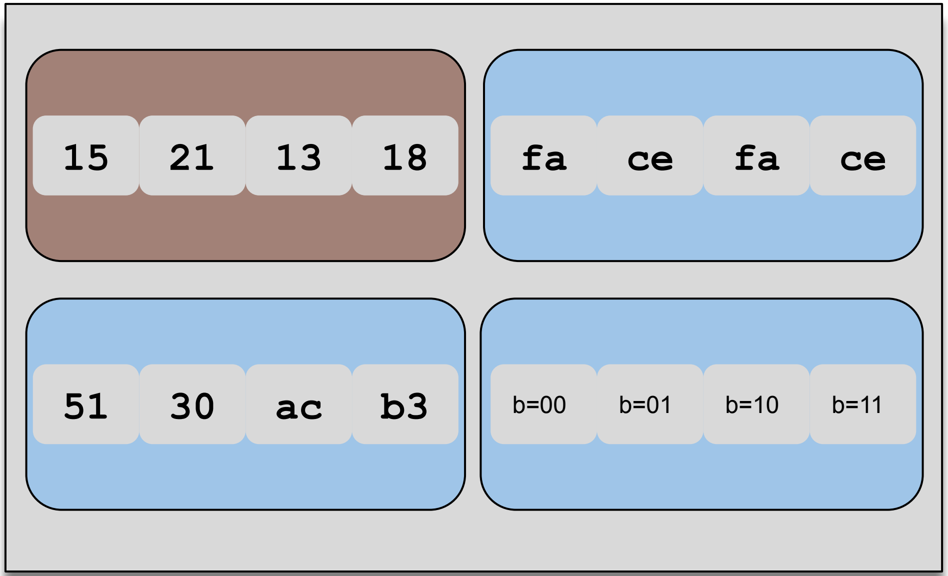
```
...
L 0,1     Hit!
L 16,1    Miss
L 9,1     Miss
...
```

9 = 0b1001

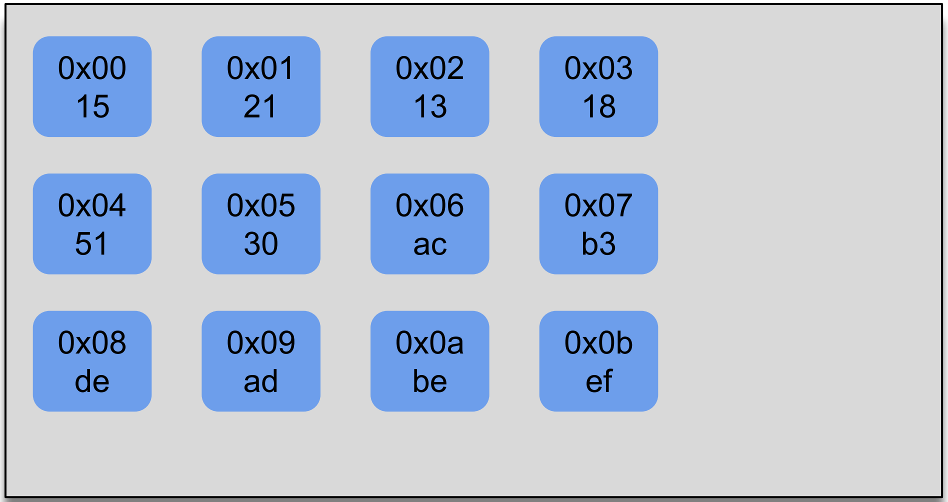
=> Set Index 0

=> Have to evict!

Cache



Memory

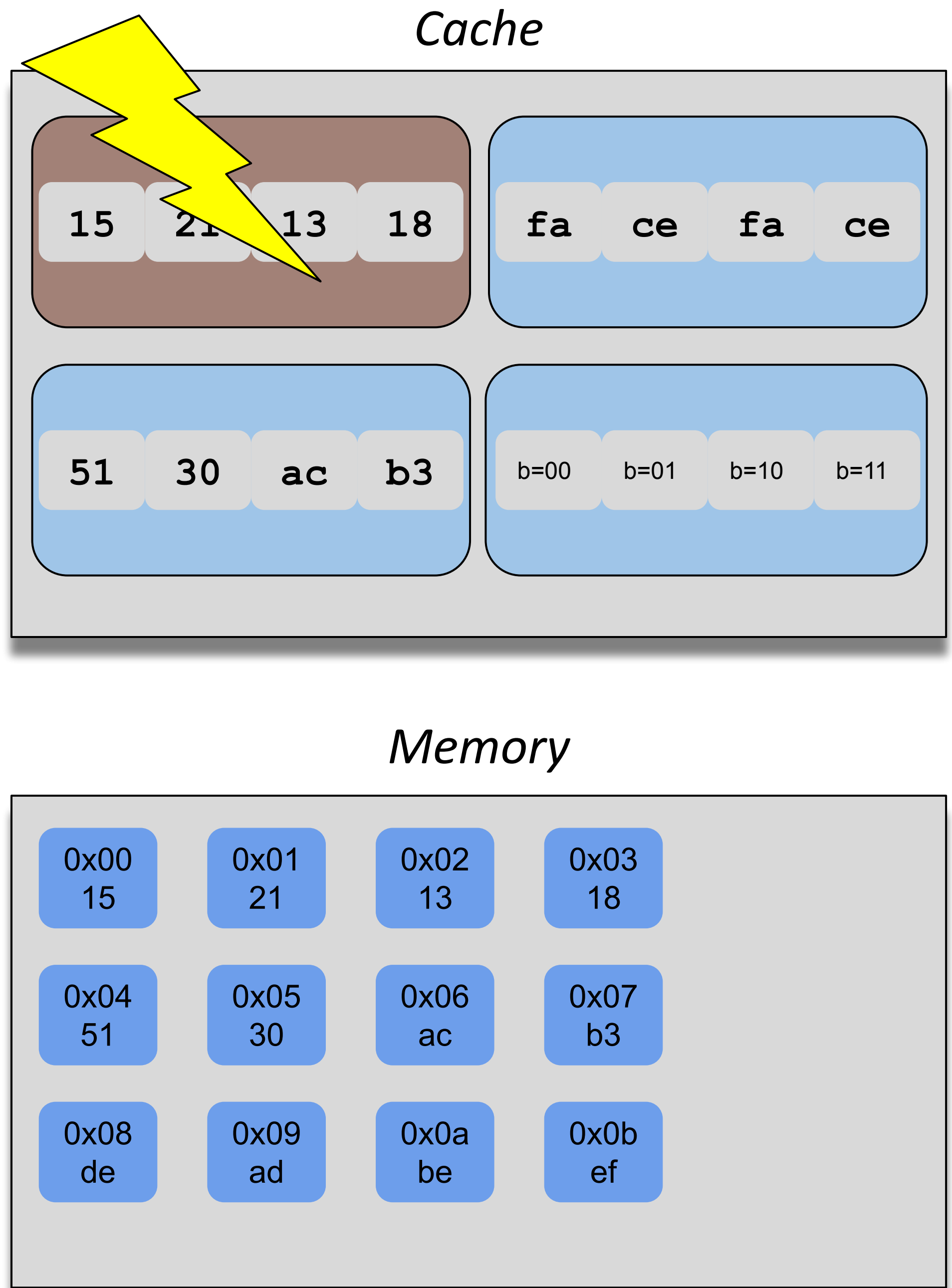


Example Trace

```
...
L 0,1      Hit!
L 16,1     Miss
L 9,1      Miss
...
```

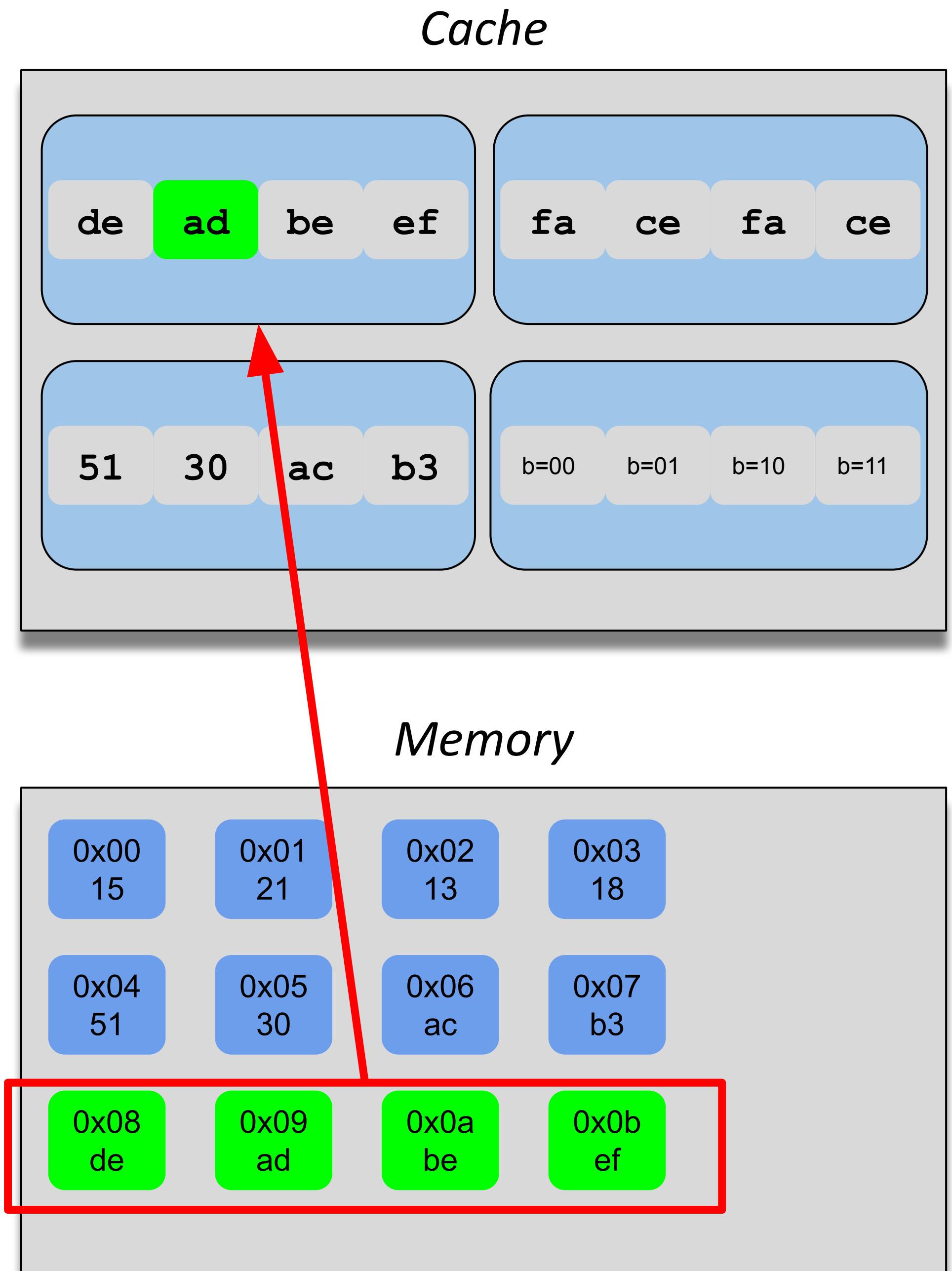
Evict least recently used line

Dirty bit set => Dirty Eviction



Example Trace

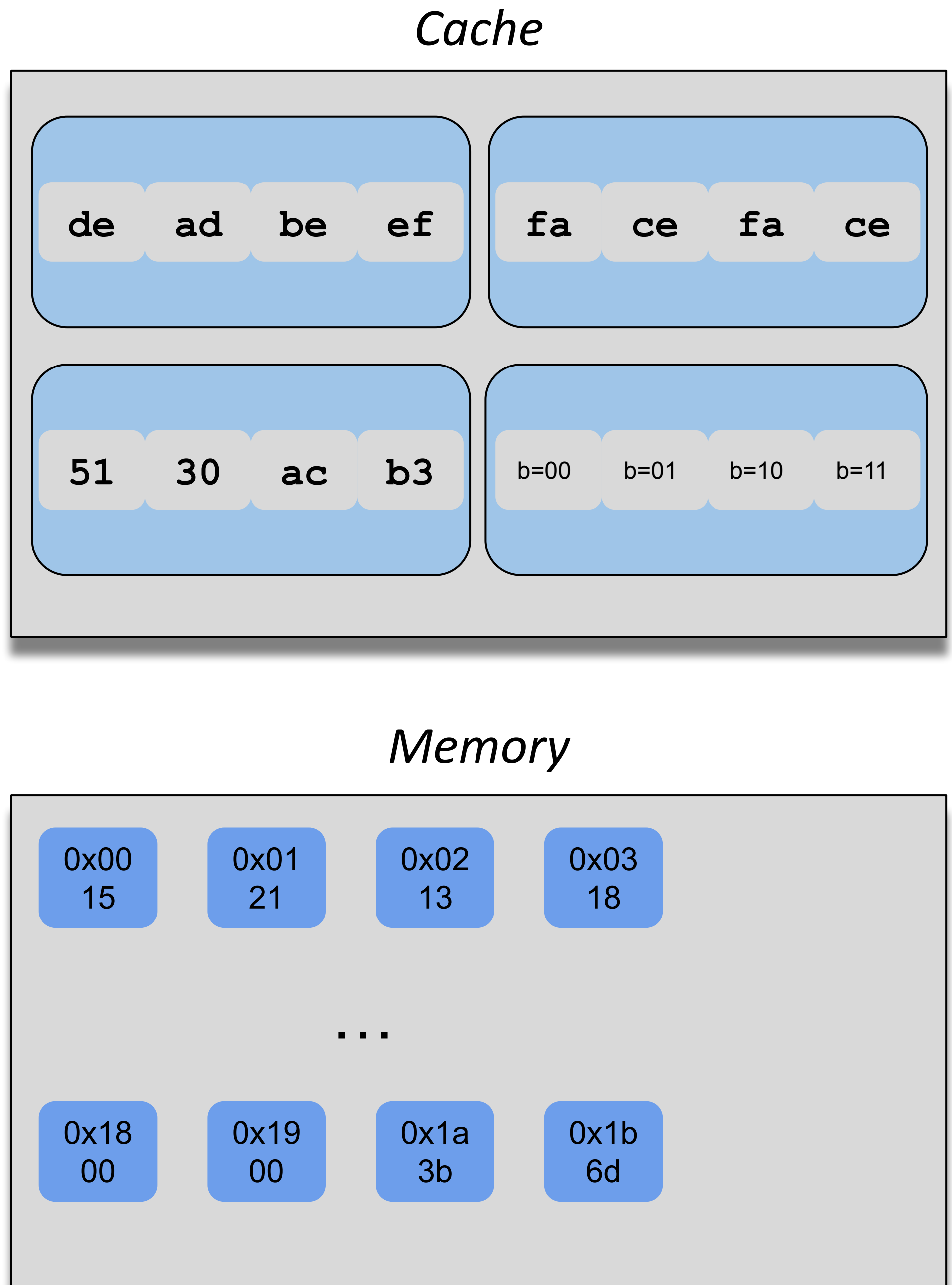
```
...
L 0,1     Hit!
L 16,1    Miss
L 9,1     Miss
...
```



Load new value into line

Example Trace

```
...
L 16,1    Miss
L 9,1     Miss
L 24,1    ???
...
```



Will this instruction result in a hit or a miss?

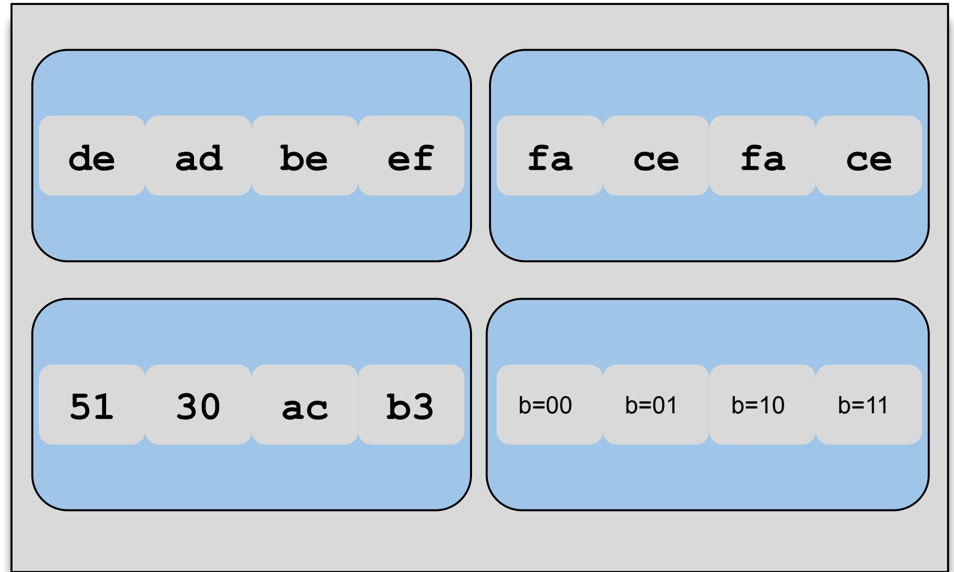
Example Trace

```
...
L 16,1    Miss
L 9,1     Miss
→ L 24,1  Miss
...
```

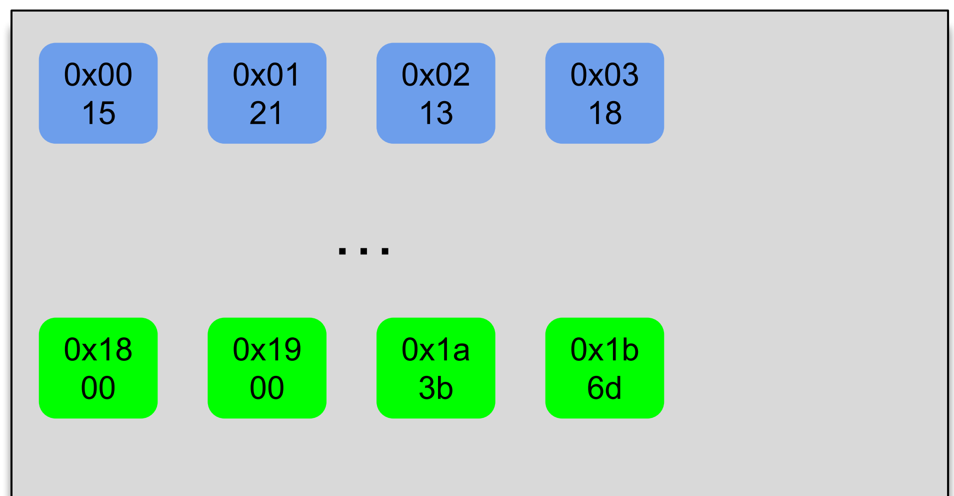
What type of miss is this?

Which line will get evicted?

Cache



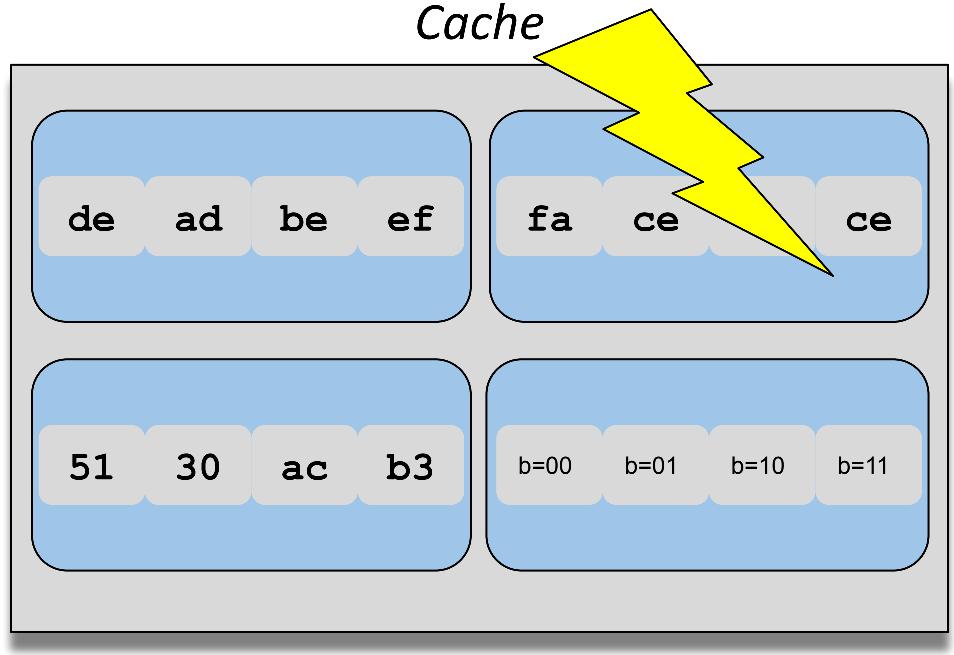
Memory



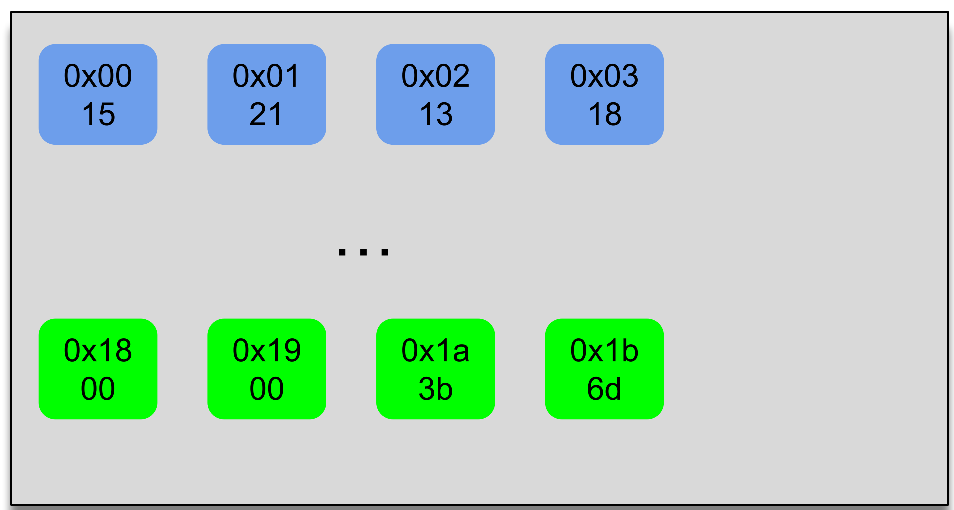
Example Trace

```
. . .
L 16,1    Miss
L 9,1     Miss
L 24,1    Miss
. . .
```

Cache



Memory

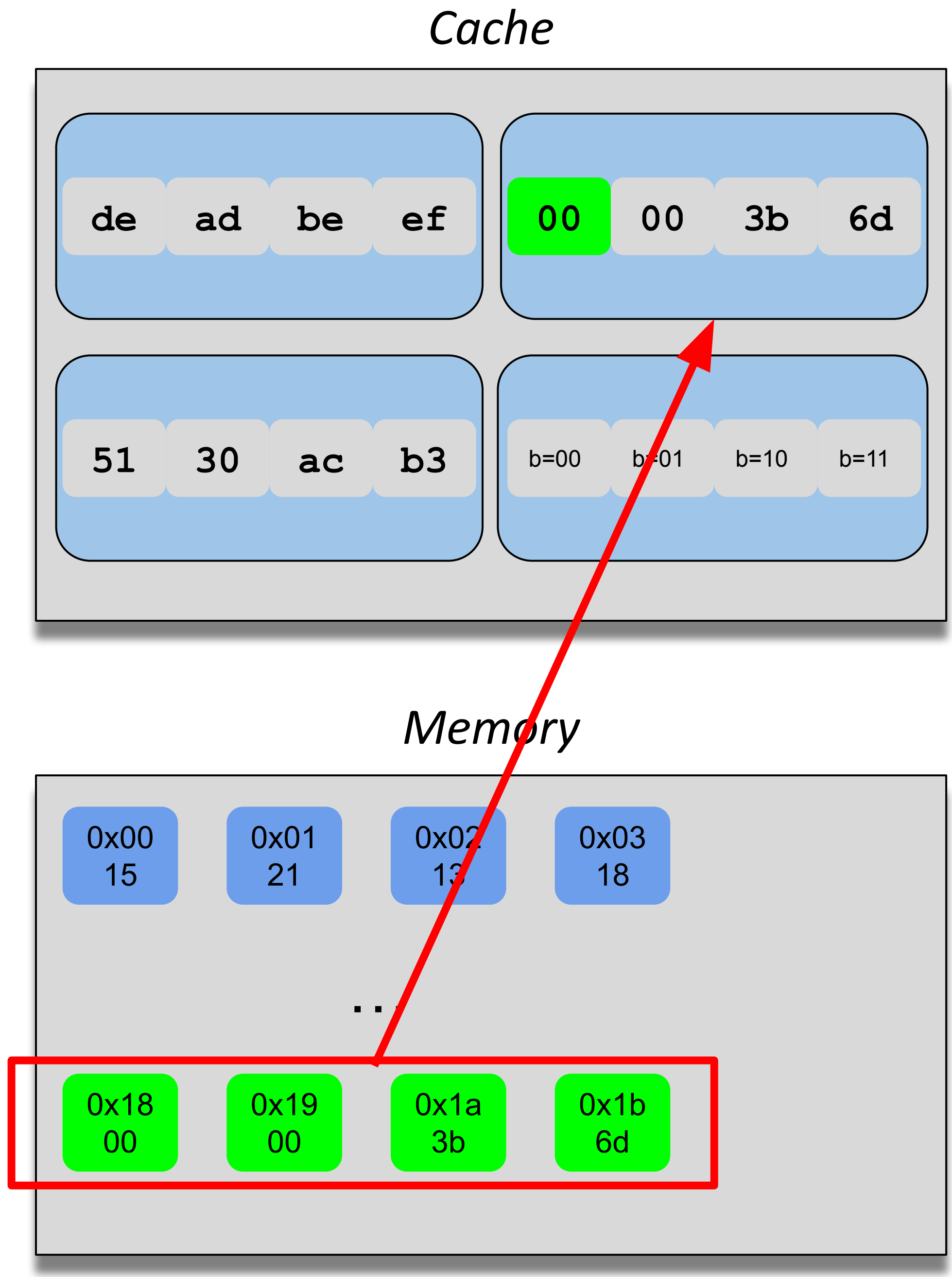


Evict least recently used line

Example Trace

```
...
L 16,1     Miss
L 9,1      Miss
L 24,1     Miss
...
```

Load new value into line

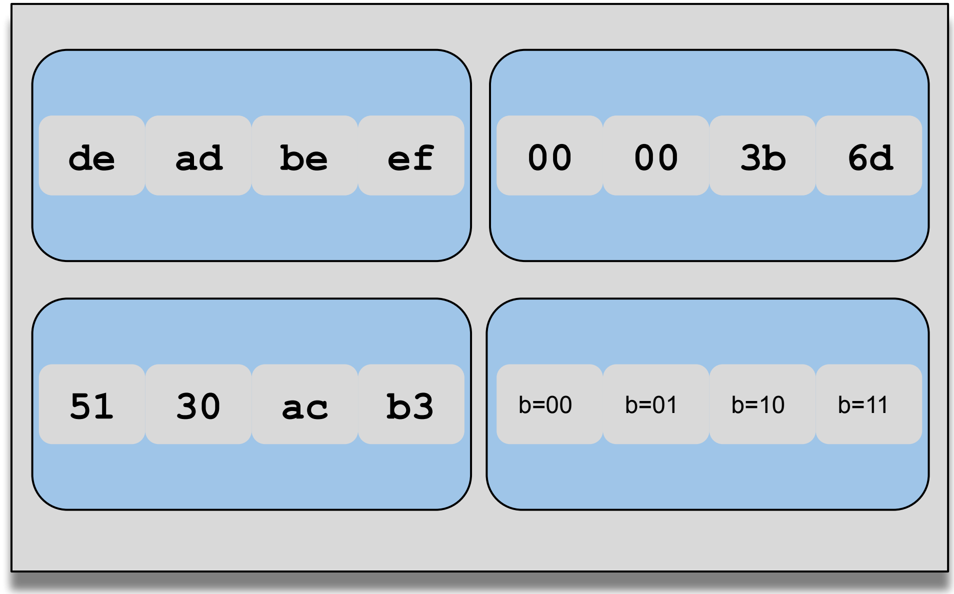


Example Trace

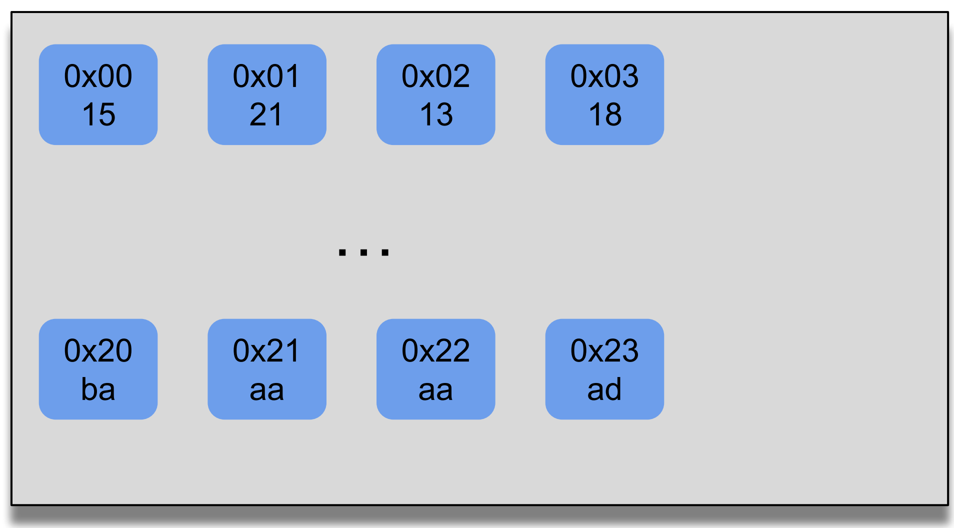
```
...
L 9,1     Miss
L 24,1    Miss
L 32,1    ???
...
```

Will this instruction result in a hit or a miss?

Cache



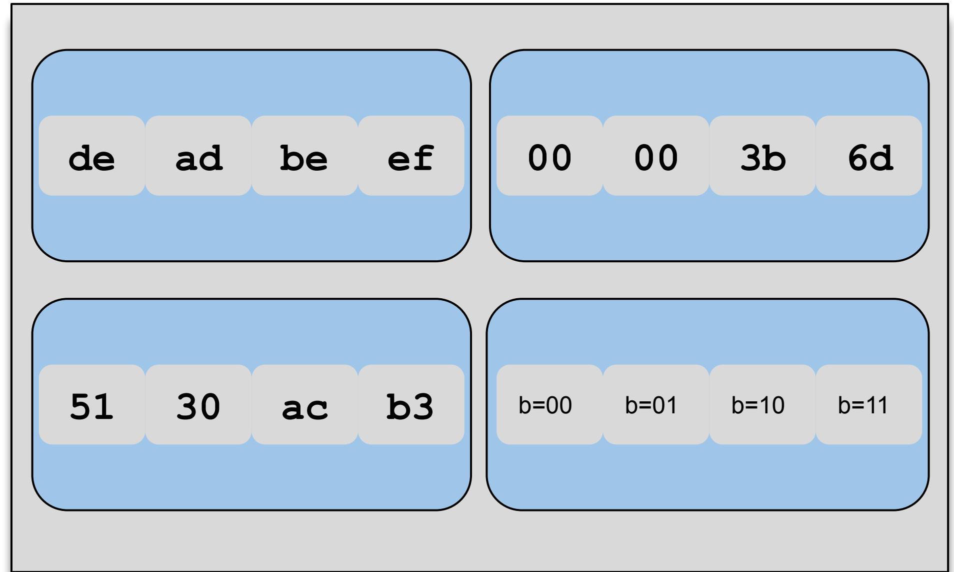
Memory



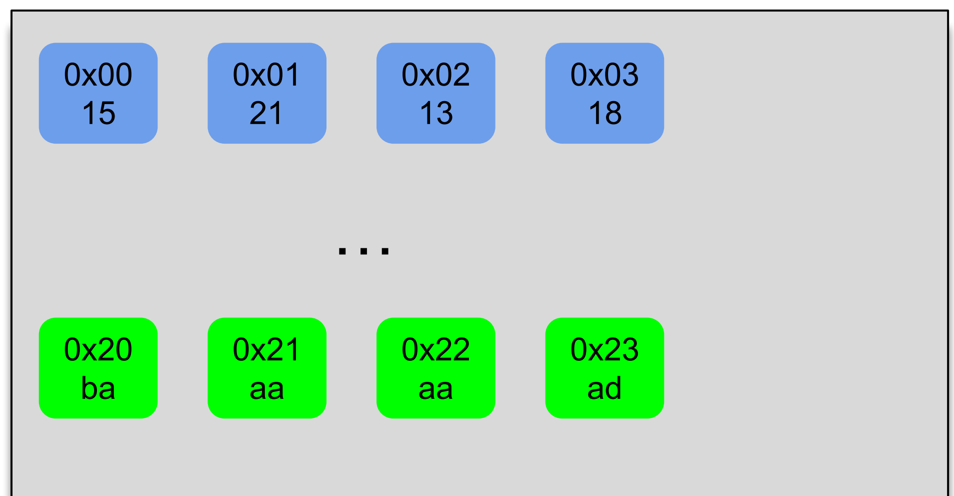
Example Trace

```
...
L 9,1     Miss
L 24,1    Miss
L 32,1    Miss
...
```

Cache



Memory

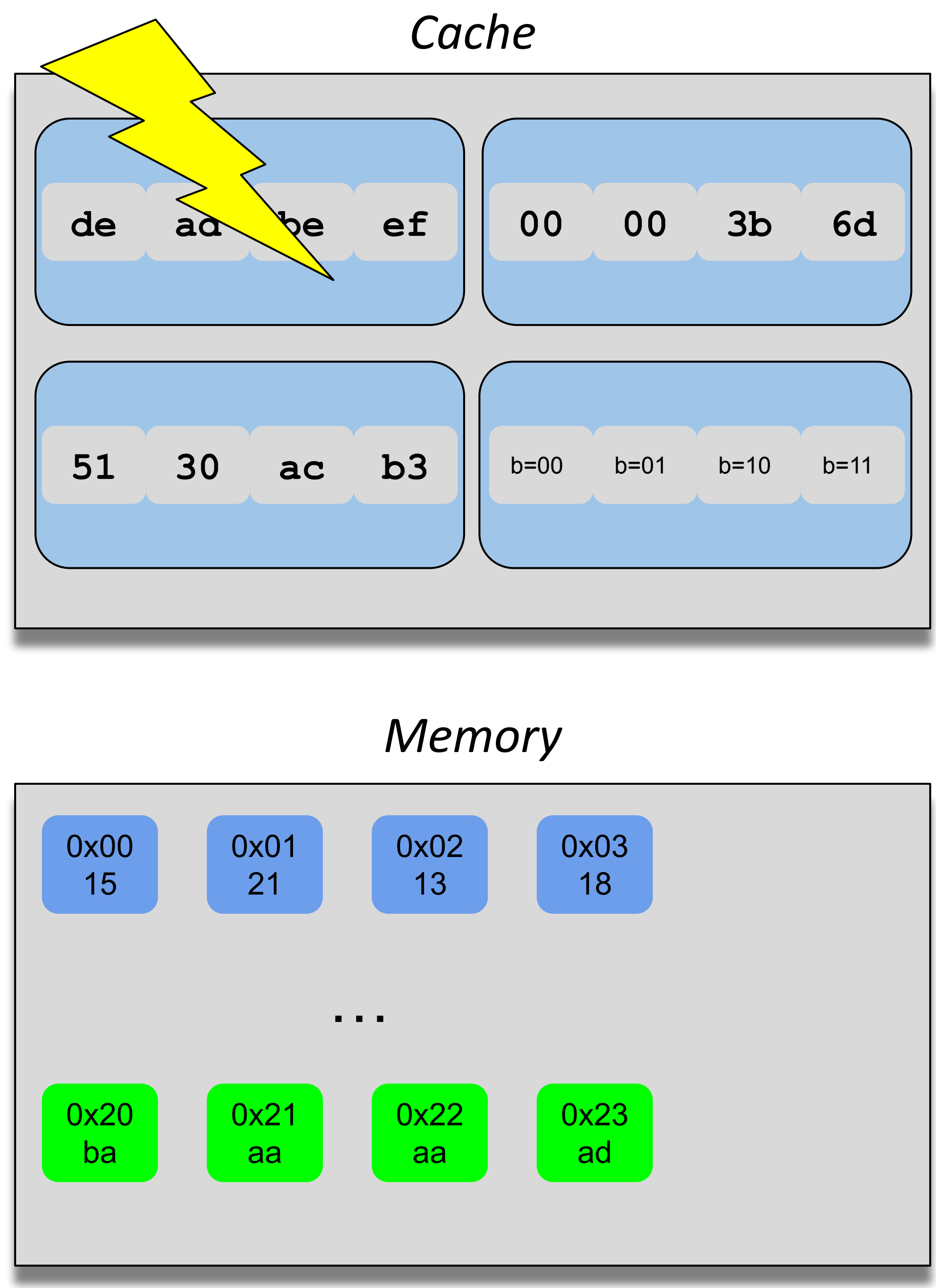


What type of miss is this?

Which line gets evicted?

Example Trace

```
...
L 9,1      Miss
L 24,1     Miss
L 32,1     Miss
...
```

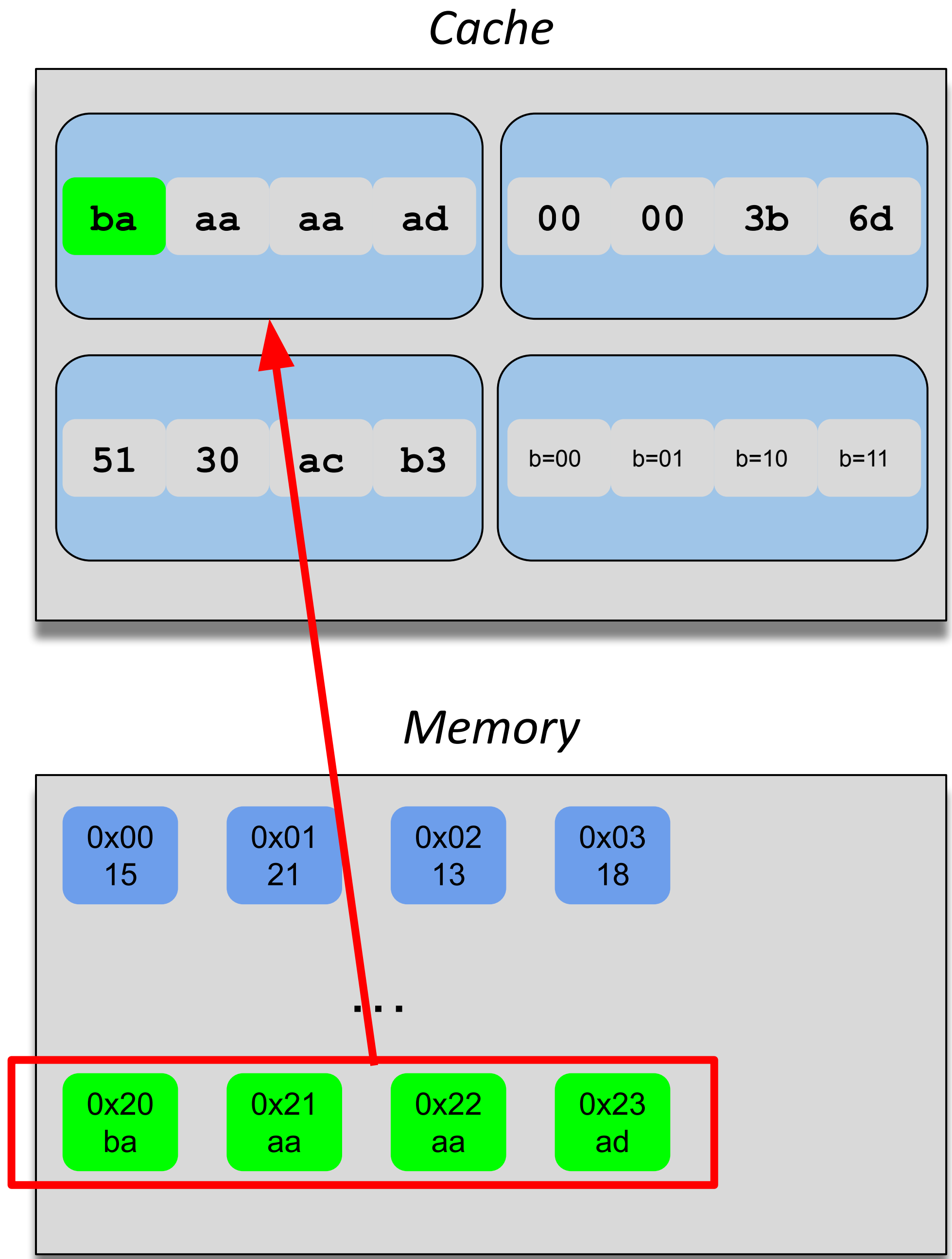


Evict least recently used line

Example Trace

```
...
L 9,1     Miss
L 24,1    Miss
L 32,1    Miss
...
```

Load new value into line

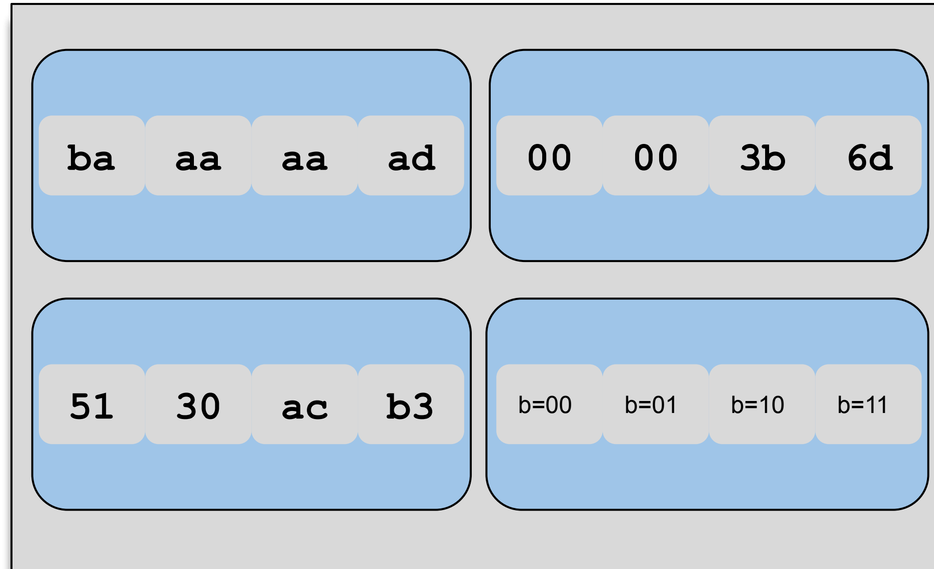


Example Trace

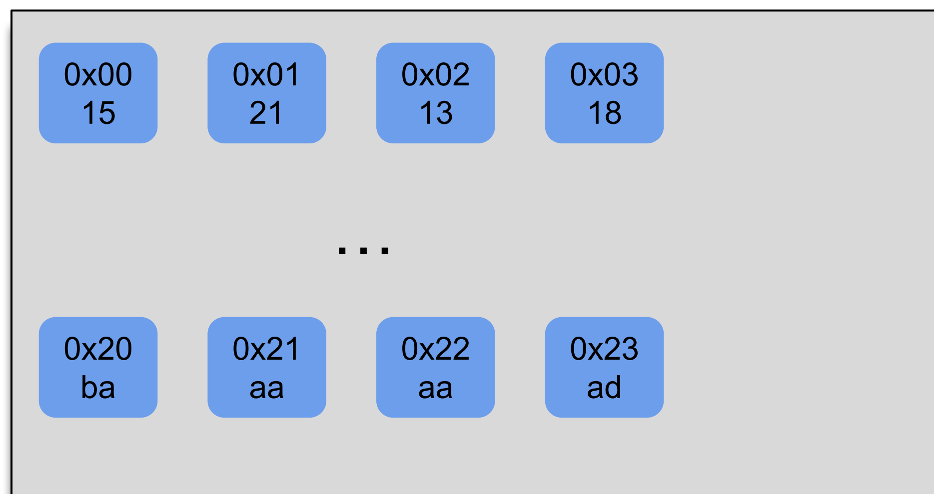
```
...
L 24,1    Miss
L 32,1    Miss
L 0,1     ???
...
```

Will this instruction result in a hit or a miss?

Cache



Memory



Cache Concepts: Conflict/Capacity Misses

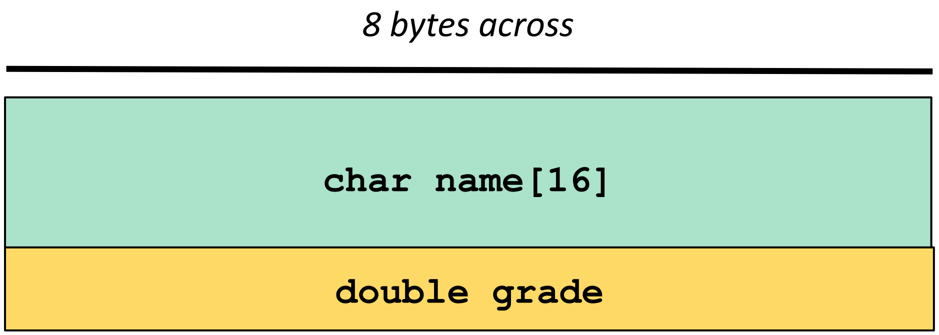
```
L 0,1
L 0,1
L 1,1
S 2,1
L 5,1
L 4,1
L 8,1
L 0,1

L 16,1
L 9,1
L 24,1
L 32,1

L 0,1
```

- In this case:
  - Number of unique blocks *in-between* current reference and most recent reference: 4
  - Our cache has 4 total lines
  - So: ***Capacity Miss***
- Note: the cache is **not** full!

Review: Programming in C

Programming in C: Structs

```
struct student {
   char name[16];
   double grade;
};
```



- Group multiple related fields under one block of memory, at one address.
- Will probably be useful for **`cachelab`**!

Programming in C: Style

- ***Code Reviews***: **`cachelab`** will be the first lab graded for style by your TAs.
  - Comments
  - File Header
  - Modularity
  - Correctness:
    - **`malloc()`** can fail! Library functions can fail!
    - Memory leaks, File Descriptor leaks

Activity: Parsing Command-Line Arguments with `getopt()`

Activity: `getopt()`

- Split up into groups of 2-3 people
- One person needs a laptop
- On a Shark Machine, type:

```
$ wget https://www.cs.cmu.edu/~213/activities/rec6.tar
$ tar -xvf rec6.tar
$ cd rec6
```

- Before getting started, you'll need to learn what getopt() does.
- Read the man pages!
  - **`man getopt`**
  - Or https://linux.die.net/man/3/getopt

Activity: `getopt_example.c`

```
$ make
$ ./getopt_example <Your arguments here>
```

- Try running the program with some arguments:
  - e.g. `./getopt_example -v -n 5`
  - What do you see?
- Look at the source code, and see if you can answer the following:
  - How does the program process its arguments?
  - What does the `-v` argument do? What does the `-n` argument do?

Activity: getopt_example.c

```
while ((opt = getopt(argc, argv, "vn:")) != -1) {
    switch (opt) {
        case 'v':
            verbose = 1;
            break;
        case 'n':
            n = atoi(optarg);
            break;
        default:
            fprintf(stderr, "usage: …");
            exit(1);
    }
}
for (int i = 0; i < n; i++) {
    if (verbose) printf("%d\n", i);
}
printf("Done counting to %d\n", n);
```

Count up to **-n** argument

If **-v** argument is set, print all numbers before n

Activity: `getopt_example.c`

Returns **`-1`** when done parsing!

```
while ((opt = getopt(argc, argv, "vn:")) != -1) {
    <-- Omitted -->
}
```

- Arguments are **`-v`** and **`-n`**
- Colon indicates option **`-n`** has required argument, which will get parsed into **`optarg`**.

Cache Practice Problems

Cache Practice Problems

- We'll work through a series of questions together.
- Write down your answer to each question.
- Discuss with classmates!

Cache Practice Problem: Locality

- The following function exhibits which type of locality?

  Consider ***only <u>array accesses</u>***.

```
void who(int *arr, int size) {
    for (int i = 0; i < size-1; ++i)
        arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

C. Both spatial and temporal

D. Neither

Cache Practice Problem: Locality

- The following function exhibits which type of locality?

  Consider ***only <u>array accesses</u>***.

```
void who(int *arr, int size) {
    for (int i = 0; i < size-1; ++i)
        arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

C. Both spatial and temporal

D. Neither

- ***Spatial:*** **Items with nearby addresses tend to be referenced close together in time.**
- ***Temporal:*** **Recently accessed addresses tend to be accessed again in the near future.**

Cache Practice Problem: Locality

- The following function exhibits which type of locality?

  Consider ***only <u>array accesses</u>***.

```
void coo(int *arr, int size) {
    for (int i = size-2; i >= 0; --i)
        arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

C. Both spatial and temporal

D. Neither

Cache Practice Problem: Locality

- The following function exhibits which type of locality?

  Consider ***only array accesses***.

```
void coo(int *arr, int size) {
    for (int i = size-2; i >= 0; --i)
        arr[i] = arr[i+1];
}
```

A. Spatial

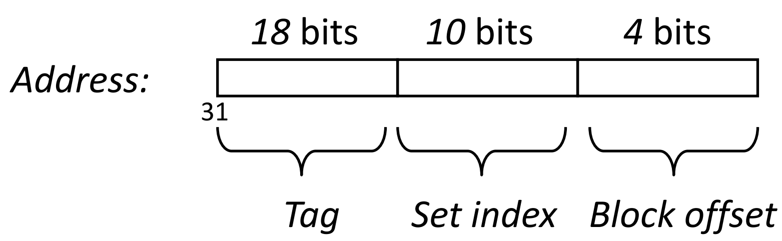
B. Temporal

*C. **Both spatial and temporal***

D. Neither

Cache Practice Problem: Cache Parameters

- Given the following address partition, how many int values fit in each block?



A. 0

B. 1

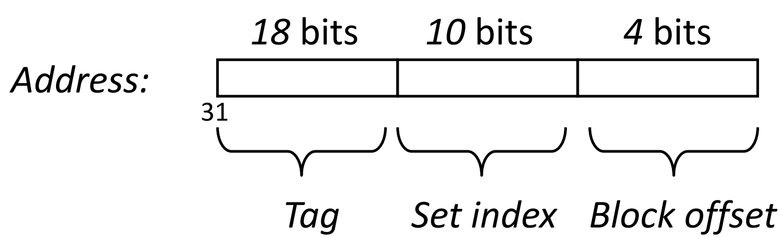
C. 2

D. 4

E. Not enough information to determine

Cache Practice Problem: Cache Parameters

- Given the following address partition, how many int values fit in each block?



A. 0

B. 1

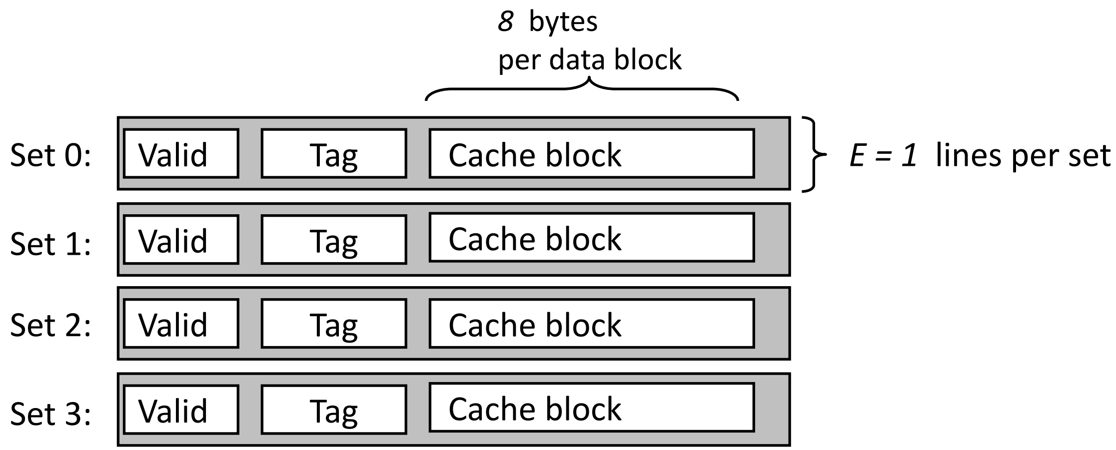
C. 2

D. 4

E. Not enough information to determine

- (**b** = 4) Four Block Offset Bits
- So block size is 2^4 = 16 bytes
- Integers are 4 bytes
- So we can fit four integers in each block.

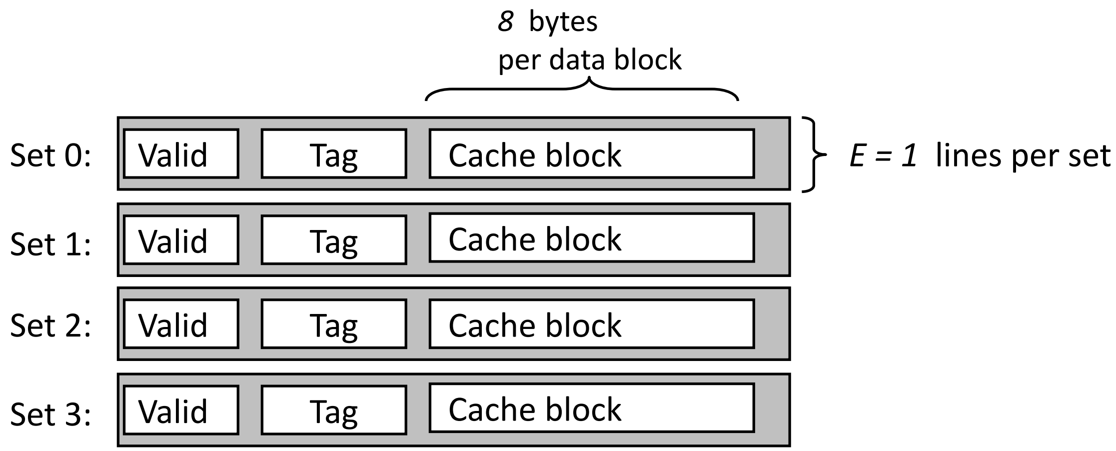
Cache Practice Problem: Cache Parameters



- What are the parameters corresponding to this cache organization?

Option	t (# Tag Bits)	s	b
A	1	2	3
B	27	2	3
C	25	4	3
D	1	4	8
E	20	4	8

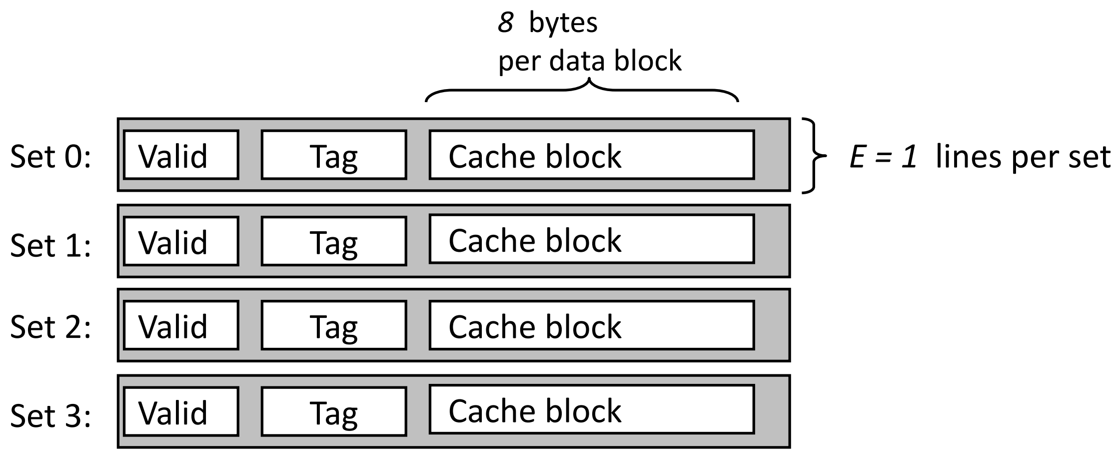
Cache Practice Problem: Cache Parameters



- What are the parameters corresponding to this cache organization?

Option	`t` (# Tag Bits)	`s`	`b`
A	1	2	3
B	27	2	3
C	25	4	3
D	1	4	8
E	20	4	8

Cache Practice Problem: Which Set?



Which *set* does the address **`0xfa1c`** map to?

A. 0

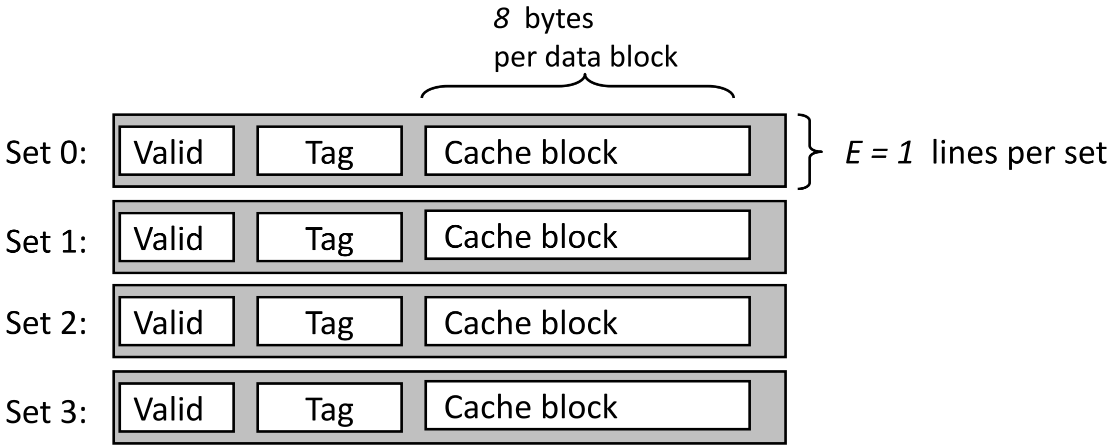
B. 1

C. 2

D. 3

E. None of the above

Cache Practice Problem: Which Set?



Which *set* does the address **`0xfa1c`** map to?

A. 0

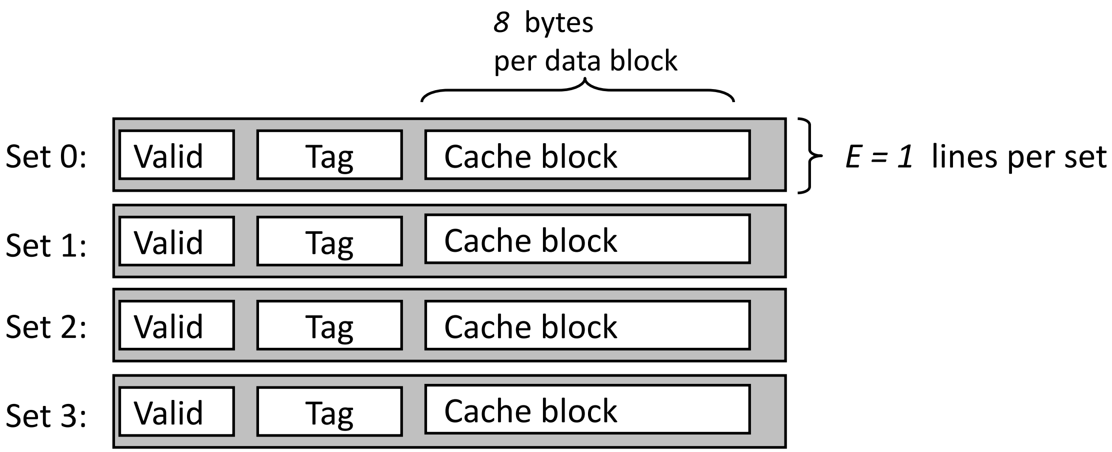
B. 1

C. 2

D. 3

E. None of the above

Cache Practice Problem: Range



Which range of addresses will be in the same block as **`0xfa1c`**?

A. **`0xfa1c`**

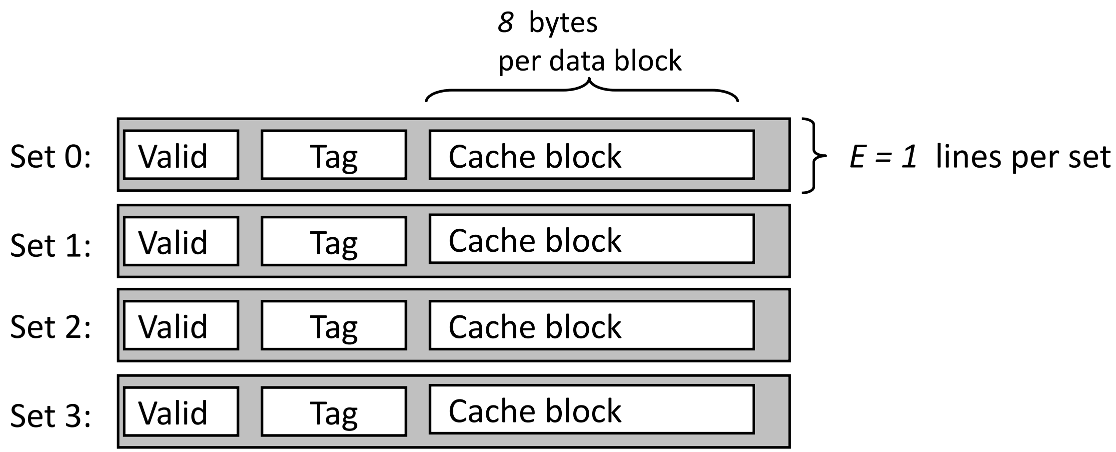
B. **`0xfa1c-0xfa23`**

C. **`0xfa1c-0xfa1f`**

D. **`0xfa18-0xfa1f`**

E. It depends on the access size

Cache Practice Problem: Range



Which range of addresses will be in the same block as **`0xfa1c`**?

A. **`0xfa1c`**

B. **`0xfa1c-0xfa23`**

C. **`0xfa1c-0xfa1f`**

D. `0xfa18-0xfa1f`

E. It depends on the access size

Cache Practice Problem

- If **N** = 16, how many bytes does the loop access of **a**?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

A. 4

B. 16

C. 64

D. 256

Cache Practice Problem

- If **N** = 16, how many bytes does the loop access of **a**?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

A. 4

B. 16

C. 64

D. 256

Wrapping Up

- `cachelab` tips:
  - Review Lectures
  - Start early! This lab can be challenging!
  - Don’t get discouraged!
- C Programming Review materials:
  - Piazza @254, @503
  - Keep an eye on Piazza for *Bootcamp 4: C Programming*.

The End