Cache Memory

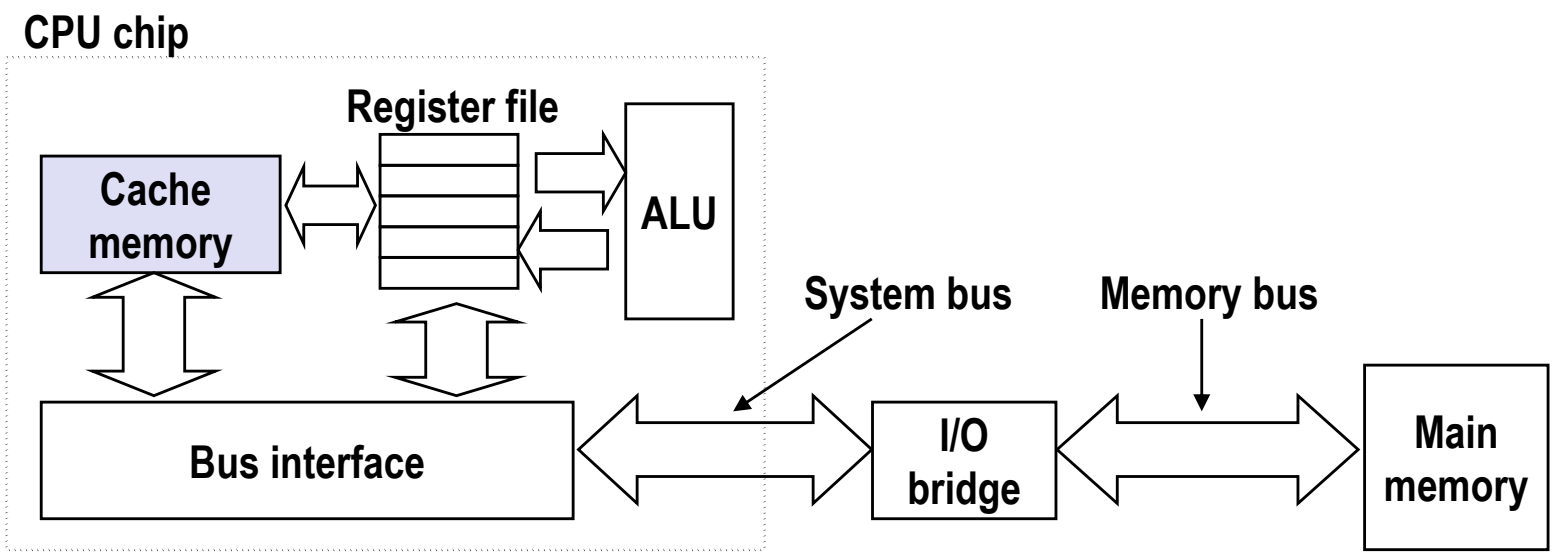
B&O Readings: 6.4-6.7

CSE 361: Introduction to Systems Software

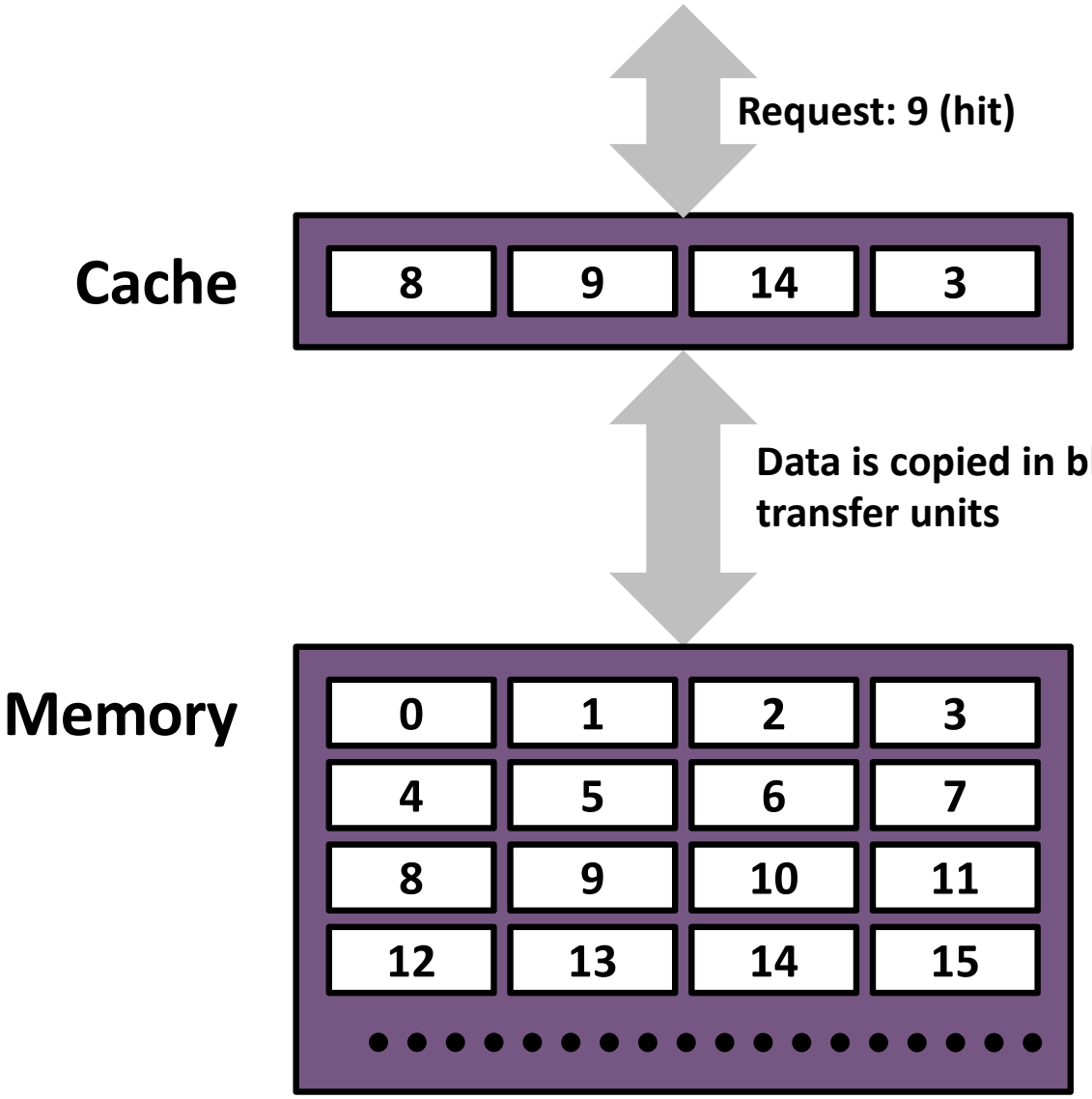
Instructor:
I-Ting Angelina Lee

Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache, then in main memory**
- **Typical system structure:**



General Cache Concepts

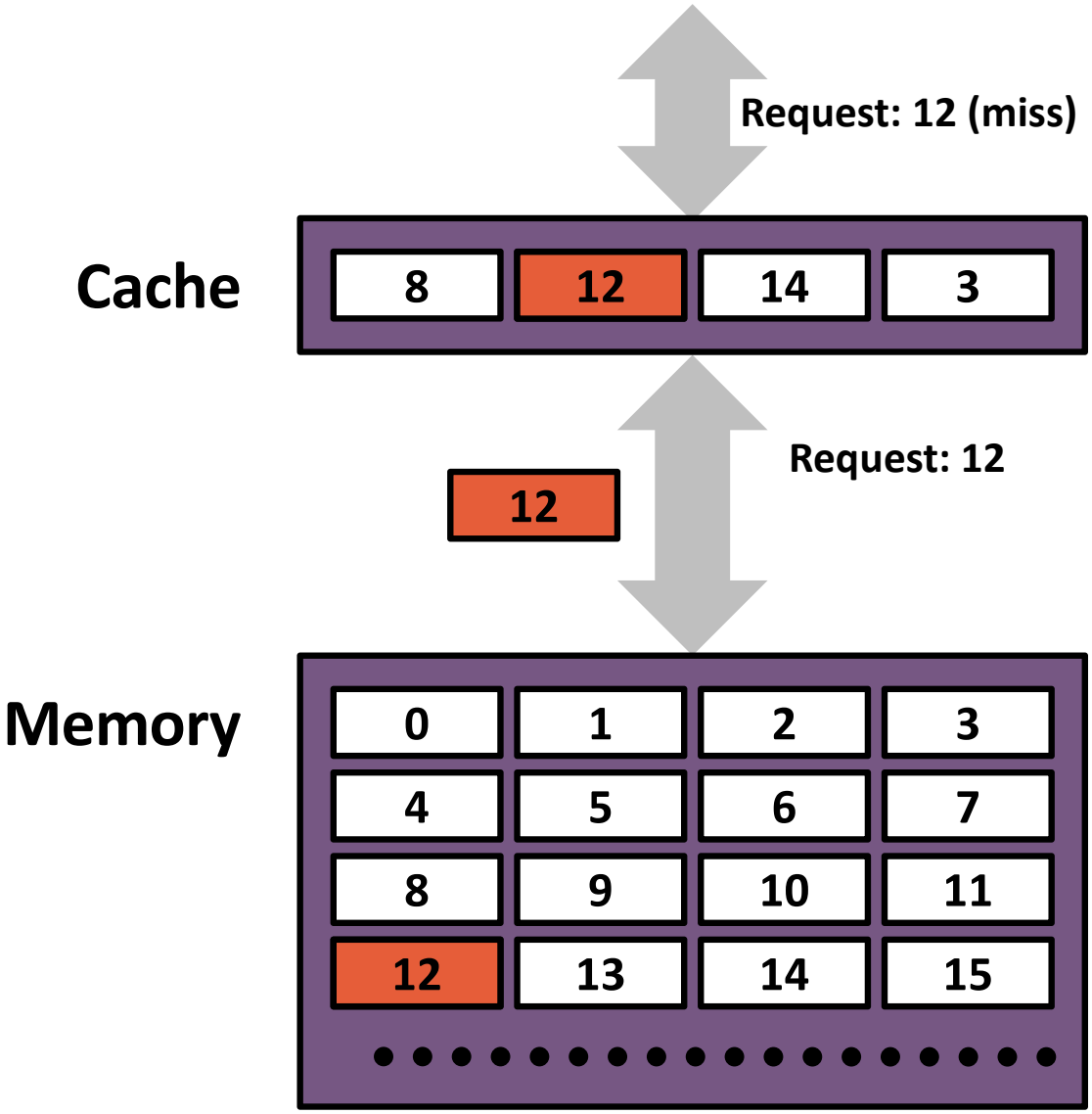


Smaller, faster, more expensive memory caches a subset of the blocks

Larger, slower, cheaper memory viewed as partitioned into "blocks"

- **Cache hit:** requested block is already in cache.
- **Cache miss:** requested block is not in cache; must bring it in from memory.

General Cache Concepts: Miss



Smaller, faster, more expensive memory caches a subset of the blocks

Block b is fetched from memory

Larger, slower, cheaper memory viewed as partitioned into "blocks"

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Spice Caches

this is your spice wall

Naz Shahrokh, http://www.artslant.com

this is your kitchen

http://www.cleverkitchen.com

You need a spice rack!

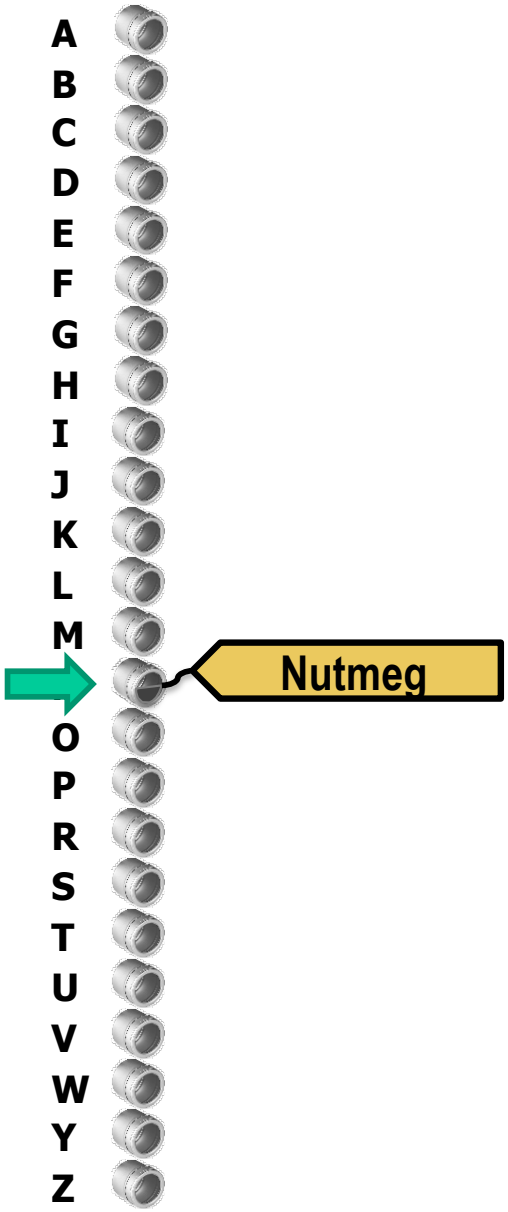
Designing the Perfect Spice Rack

http://www.bedbathandbeyond.com

- **24 spice entries, 1 for each letter (~~QX~~)**
- **Compared to your spice wall**
  - Smaller
  - Faster
  - More costly

Spice Wall

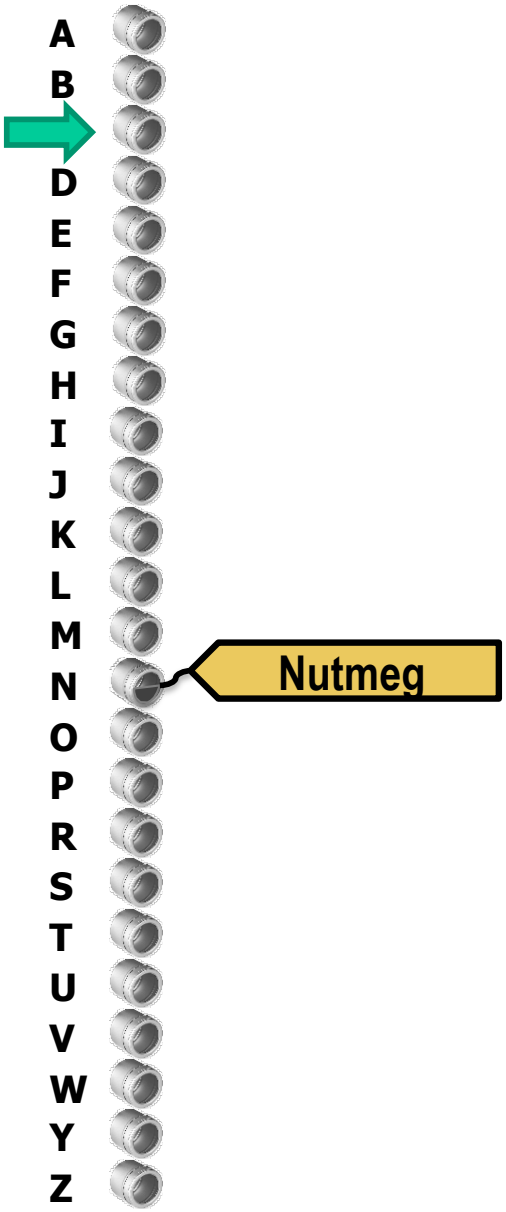
Finding a Spice (Hit)



Placement policy: look in the right row based on the 1st letter of spice

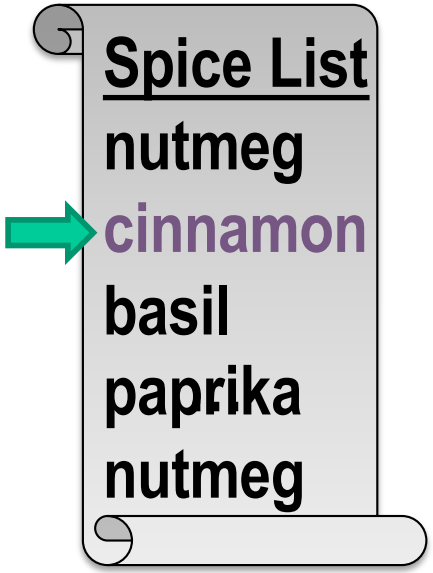
- **Hit:** ***yay!***

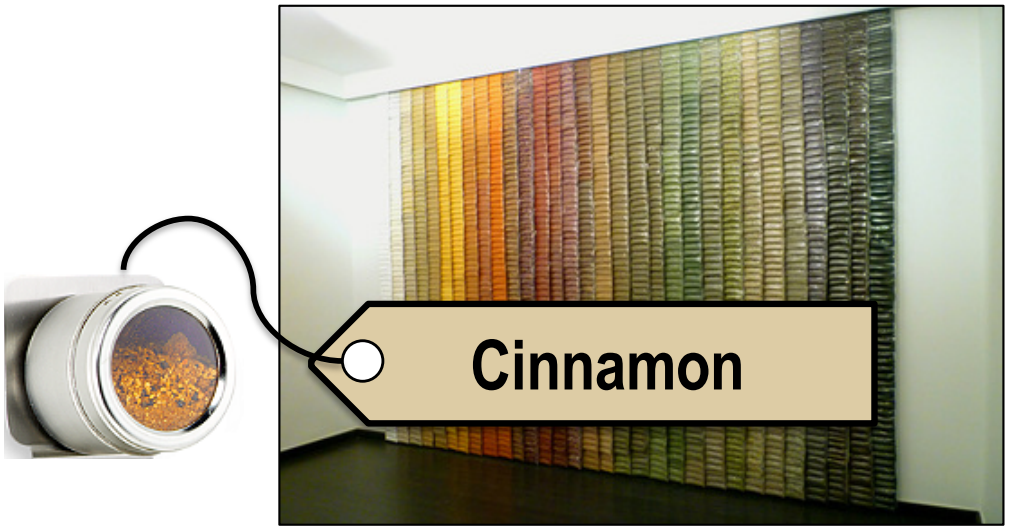
Not Finding a Spice (Miss)



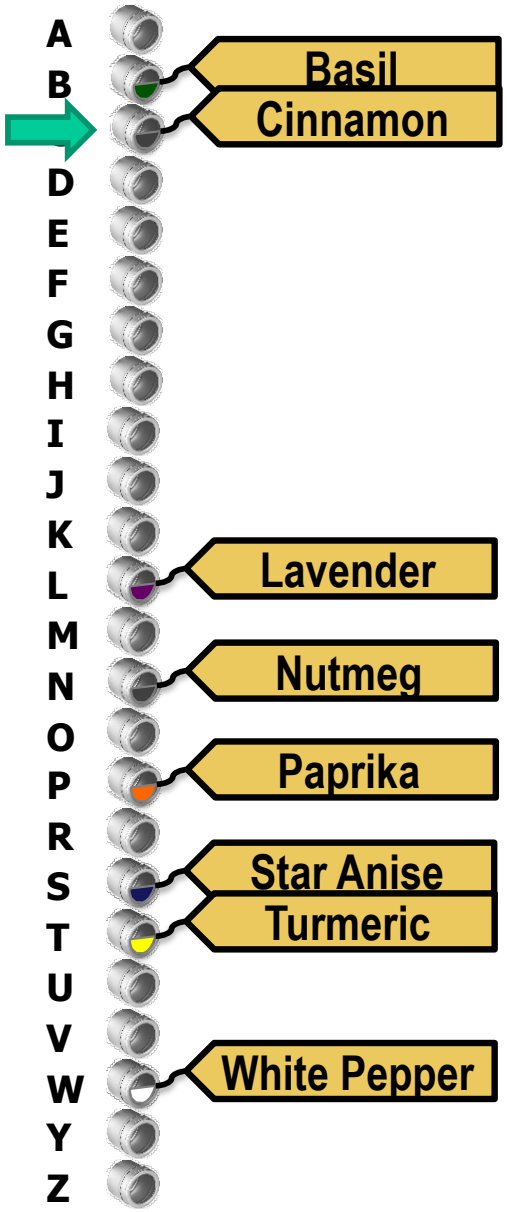
Placement policy: look in the right row based on the 1st letter of spice

- **Miss: *boo!***
  - retrieve from spice wall
  - put it in the spice rack





2 Refinements: Tags & Evictions

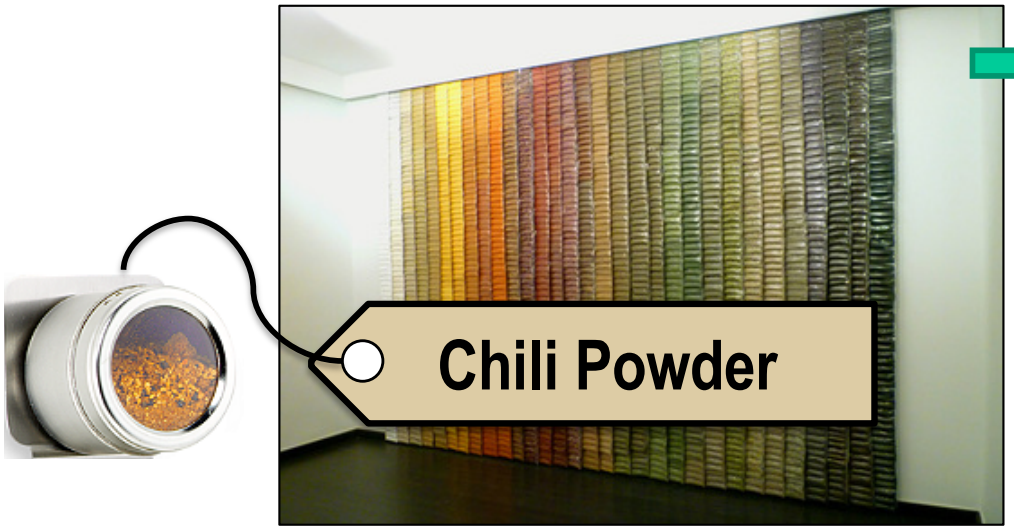


Placement policy: look in the right row

Replacement policy: only one choice, so simple

Read the tag!

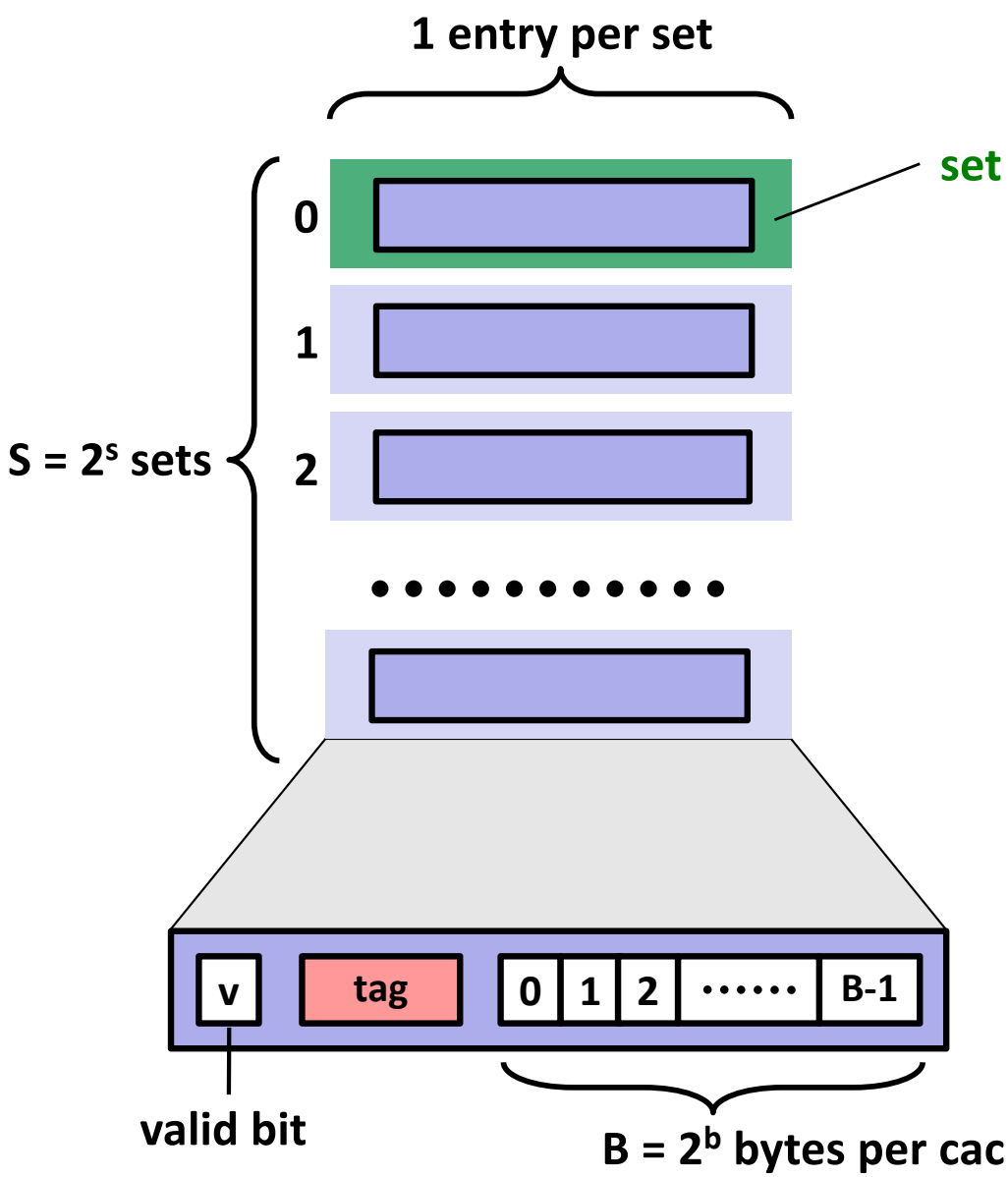
- **Miss:**
  - retrieve from spice wall
  - *always evict old spice*
  - put it in the spice rack



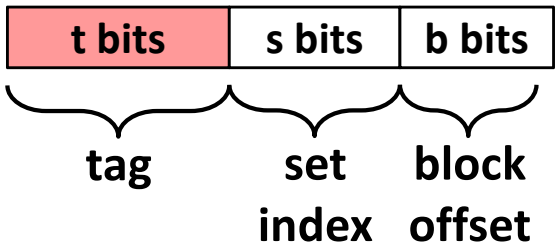
Spice Cache Example: Types of Cache Misses

- **Cold (compulsory) miss**
  - because the cache is empty (that block was never there before)
  - ex: the very first time we use a particular spice
- **Conflict miss**
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
  - ex: we are in trouble if the recipe calls for chili powder and cinnamon repeatedly
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
  - ex: the recipe calls for more than 24 spices (WHAT are you cooking??)

Cache Organization: Direct Mapped



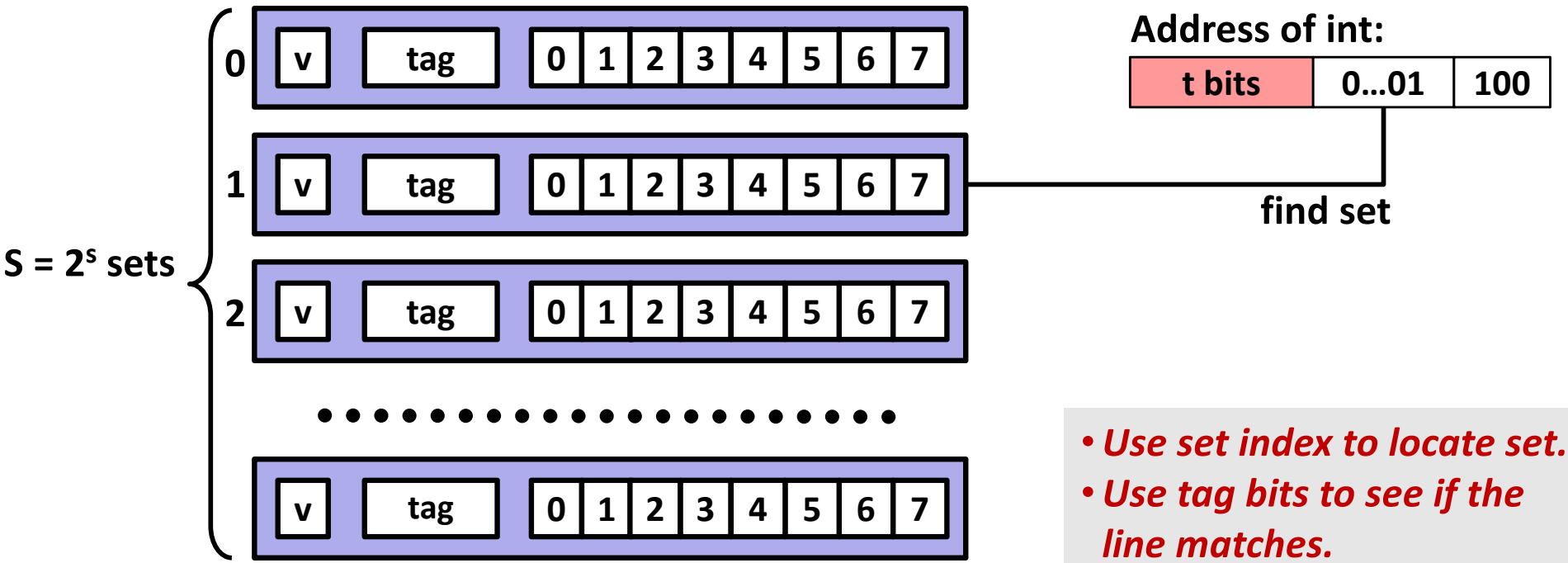
Address of word:



- *Block offset: index into the data block in cache line.*
- *Set index: index cache to locate set.*
- *Tag bits: disambiguate lines that maps to the same set.*
- *Valid bit: indicate if a line is valid.*

Example: Direct Mapped Cache

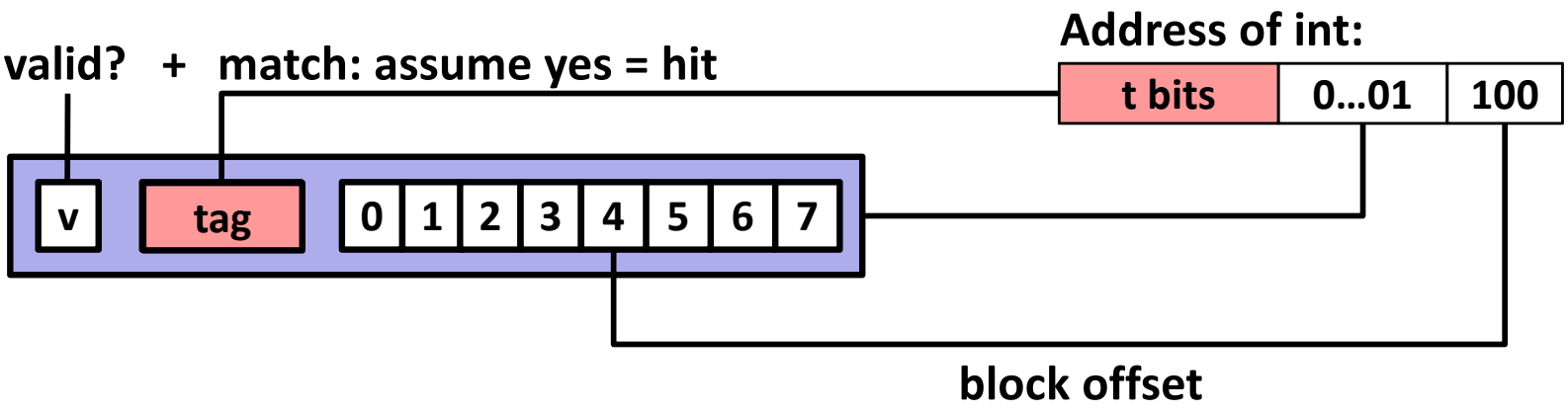
Direct mapped: One line per set
Assume: cache block size 8 bytes



- *Use set index to locate set.*
- *Use tag bits to see if the line matches.*
- *Valid bit indicates if the line is valid.*
- *If valid, locate data using block offset.*

Example: Direct Mapped Cache

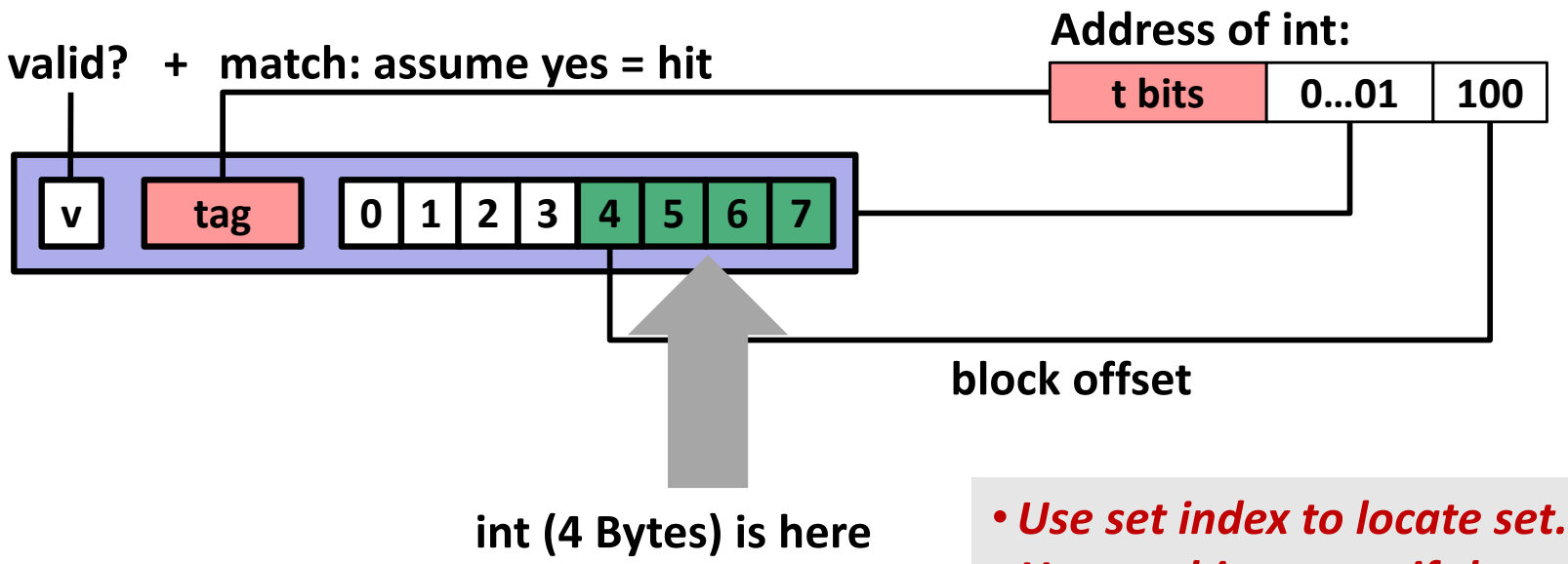
Direct mapped: One line per set
Assume: cache block size 8 bytes



- *Use set index to locate set.*
- *Use tag bits to see if the line matches.*
- *Valid bit indicates if the line is valid.*
- *If valid, locate data using block offset.*

Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



- *Use set index to locate set.*
- *Use tag bits to see if the line matches.*
- *Valid bit indicates if the line is valid.*
- *If valid, locate data using block offset.*

If no match: old line is evicted and replaced

Direct-Mapped Cache Simulation

Q: How many bits are in tags, set index and block offset?

t=1	s=2	b=1
x	xx	x

M=16 bytes, B=2 bytes/block, S=4 sets, E=1 block/set

Q: How many bits in the address?

A: 4

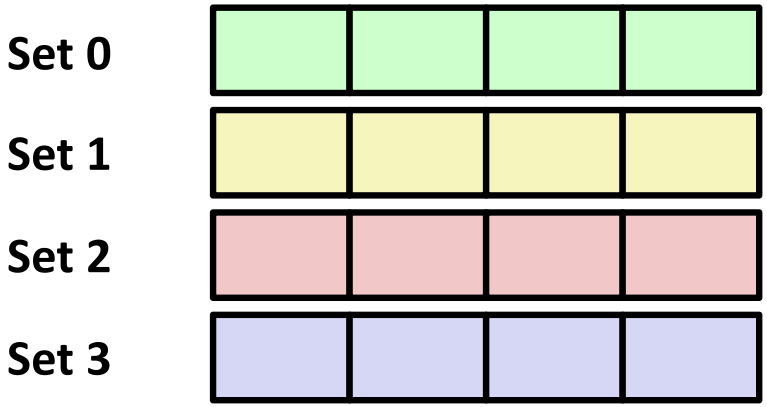
Address trace (reads, one byte per read):

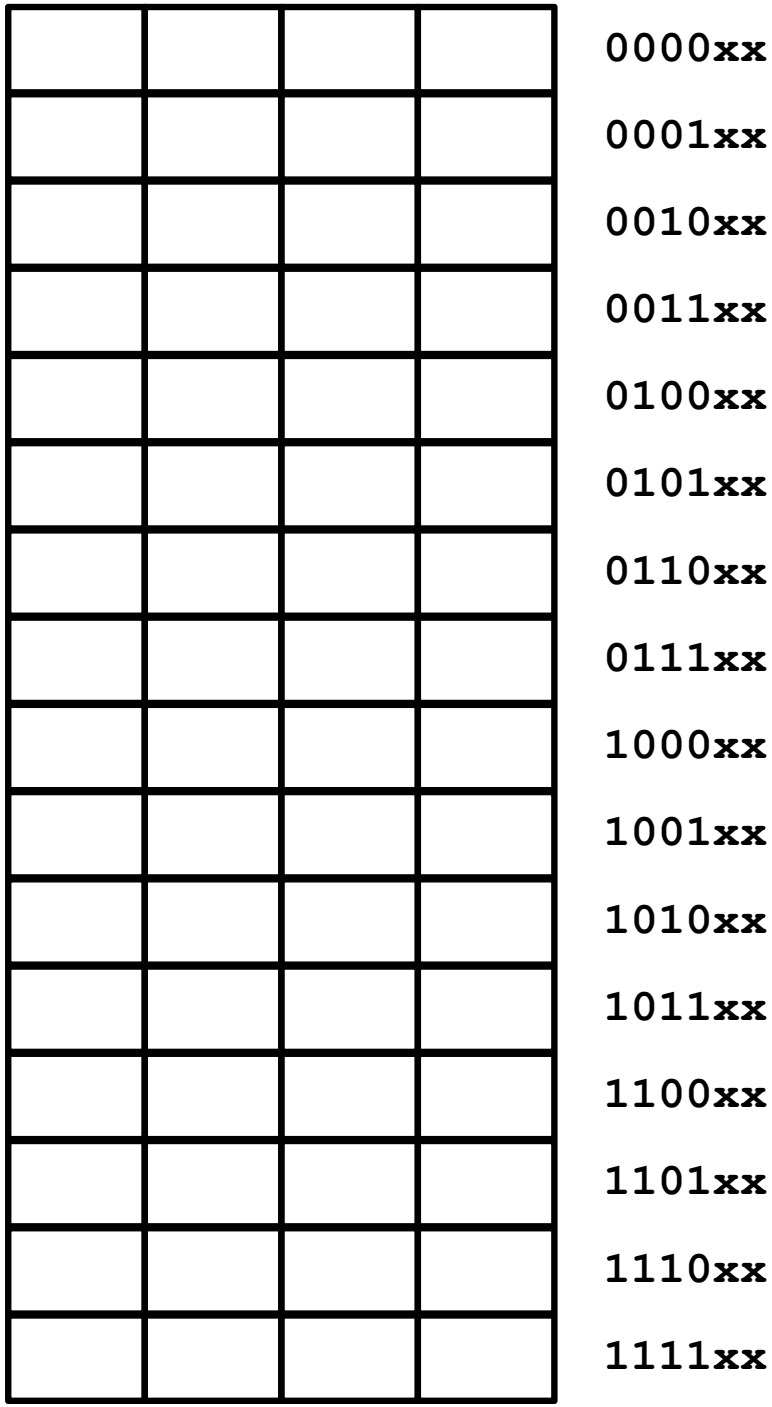
0	**[0000$_2$],**	miss (cold)
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss (cold)
8	**[1000$_2$],**	miss (conflict)
0	**[0000$_2$]**	miss (conflict)

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Illustration: Why Use Middle Bits as Set Bits?

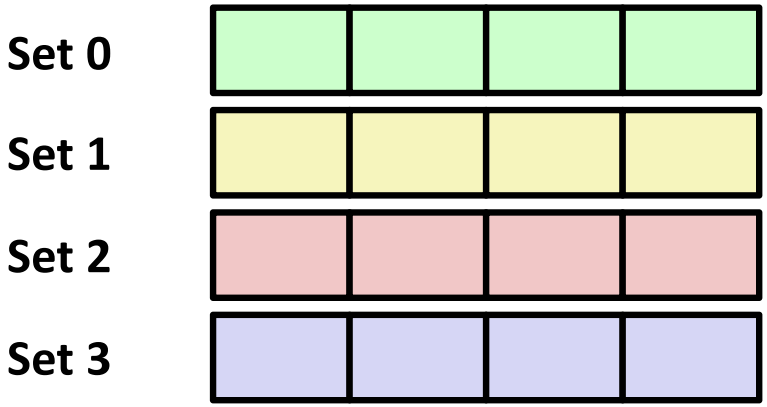
- **64-byte memory**
  - 6-bit addresses
- **16 byte, direct-mapped cache**
- **Block size = 4 (4 sets)**
- **2 bits tag, 2 bits index, 2 bits offset**

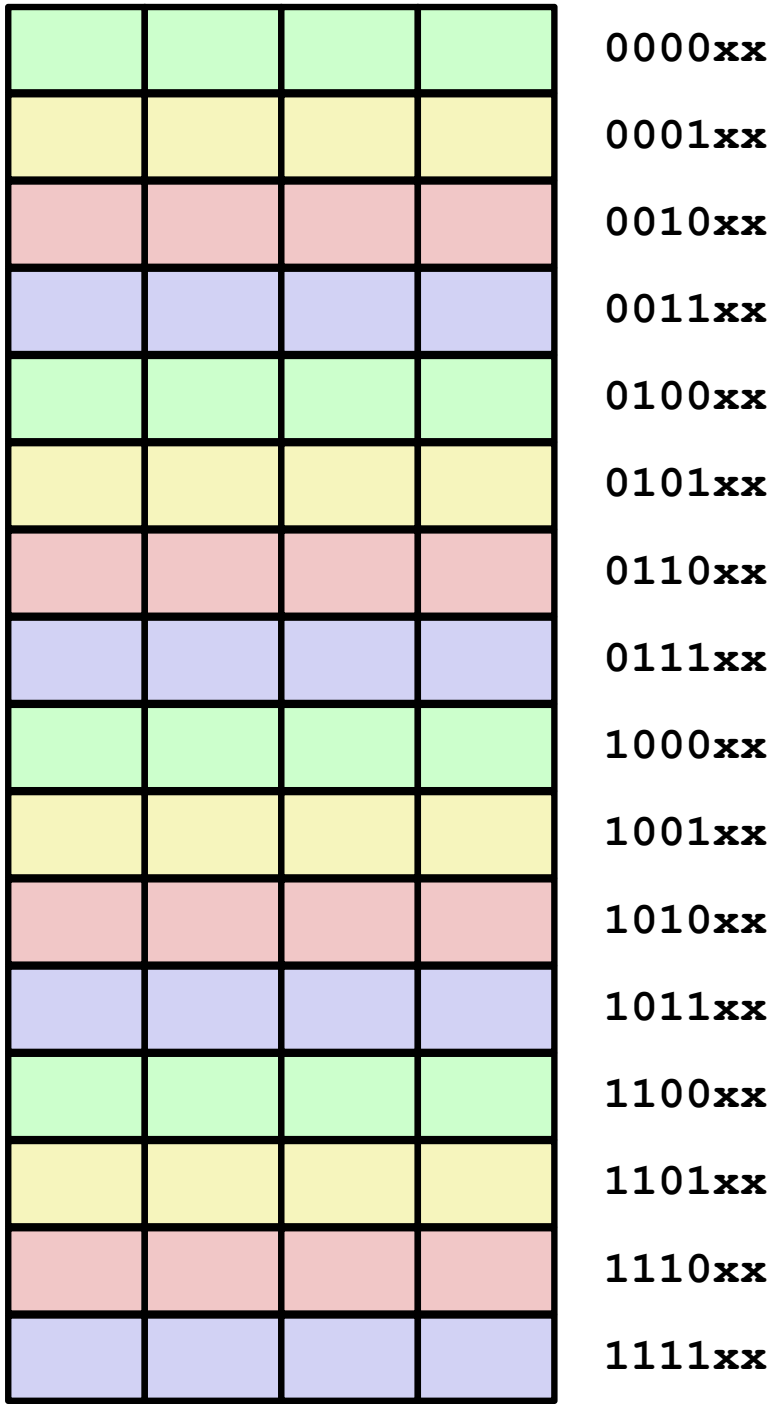




Middle Bit Indexing

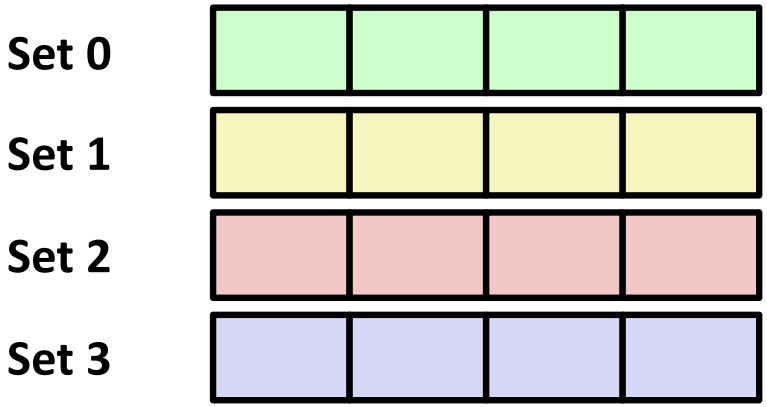
- **Addresses of form TTSSBB**
  - TT Tag bits
  - SS Set index bits
  - BB Offset bits
- **Makes good use of spatial locality**

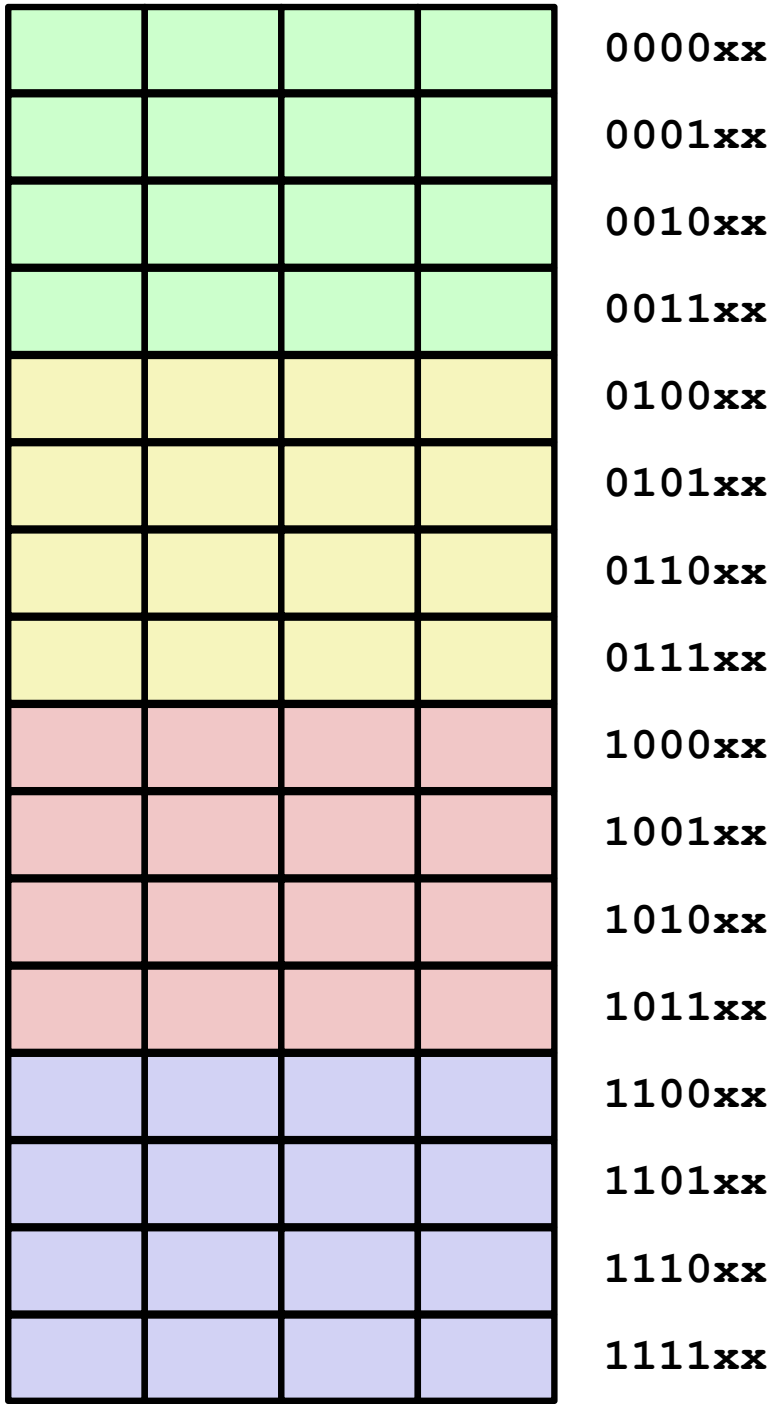




High Bit Indexing

- **Addresses of form SSTTBB**
  - SS Set index bits
  - TT Tag bits
  - BB Offset bits
- **Program with high spatial locality would generate lots of conflicts**





Recap: What We Learned Thus Far

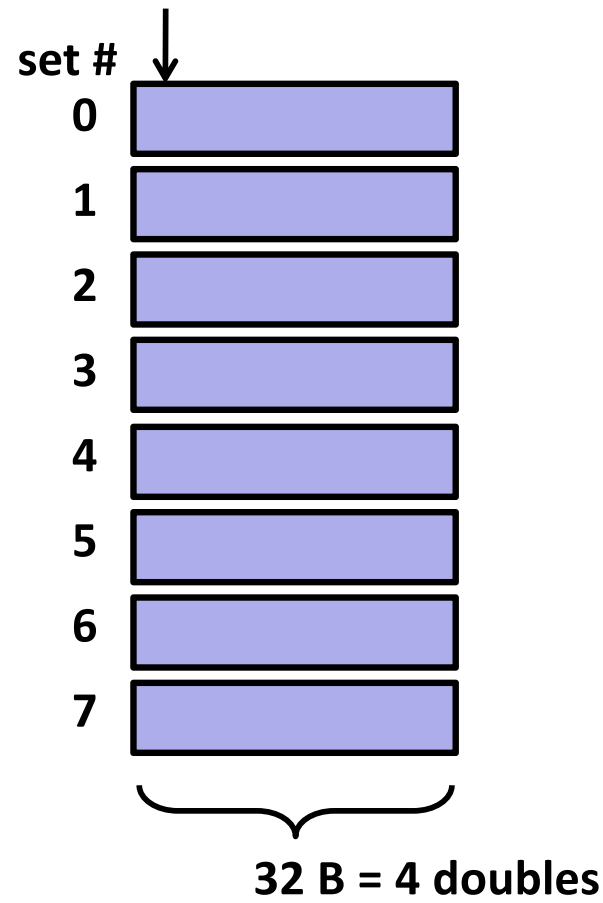
- Different types of cache misses: cold miss, capacity miss, conflict miss.
- How a direct mapped cache is organized
- How to look up a cache line in a direct mapped cache
- Why we use the middle bits as set index instead of the high bits.

A High-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache:
- Size of this array:
- Number of bits used for block offset:
- Number of bits used for indexing sets:
- Number of bits for tags:
- Number of reads performed:
- Number of misses incurred:

blackboard

A High-Level Example

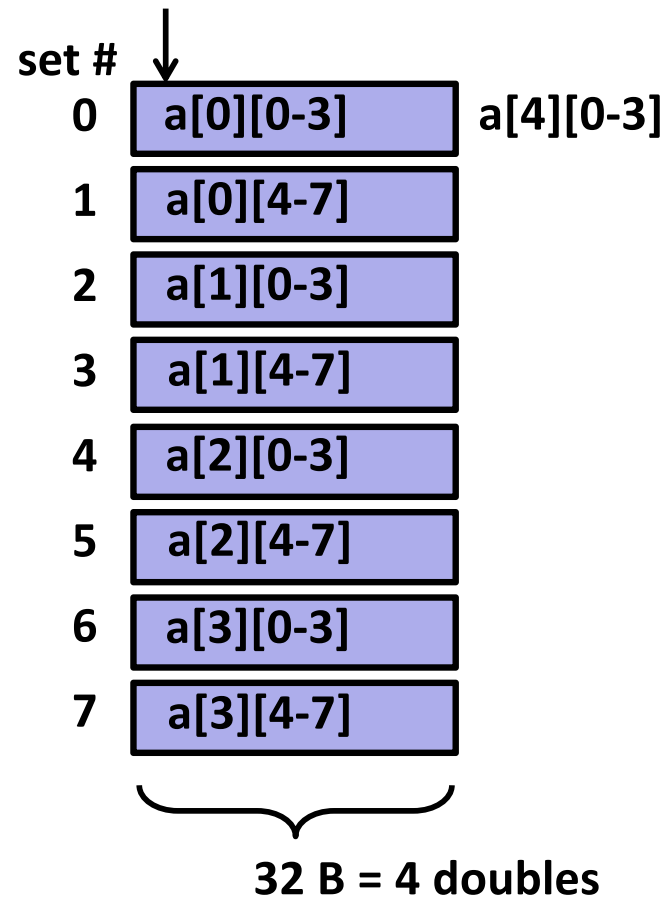
```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            sum += a[i][j];
    return sum;
}
```

Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache: 256 bytes
- Size of this array: 512 bytes
- Number of bits used for block offset: 5 bits
- Number of bits used for indexing sets: 3 bits
- Number of bits for tags: 24 bits
- Number of reads performed: 64
- Number of misses incurred:

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here



blackboard

A High-Level Example

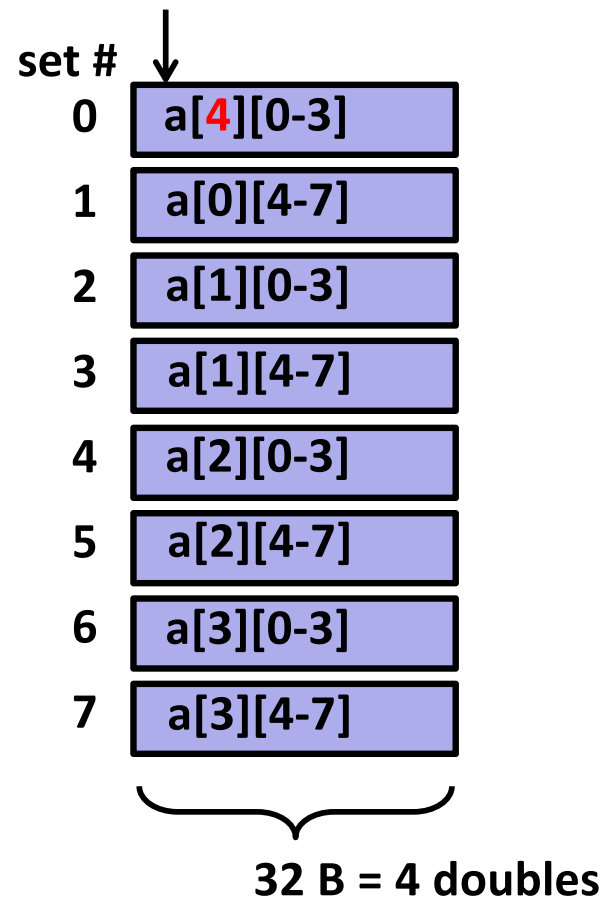
```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            sum += a[i][j];
    return sum;
}
```

Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache: 256 bytes
- Size of this array: 512 bytes
- Number of bits used for block offset: 5 bits
- Number of bits used for indexing sets: 3 bits
- Number of bits for tags: 24 bits
- Number of reads performed: 64
- Number of misses incurred:

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here



blackboard

A High-Level Example

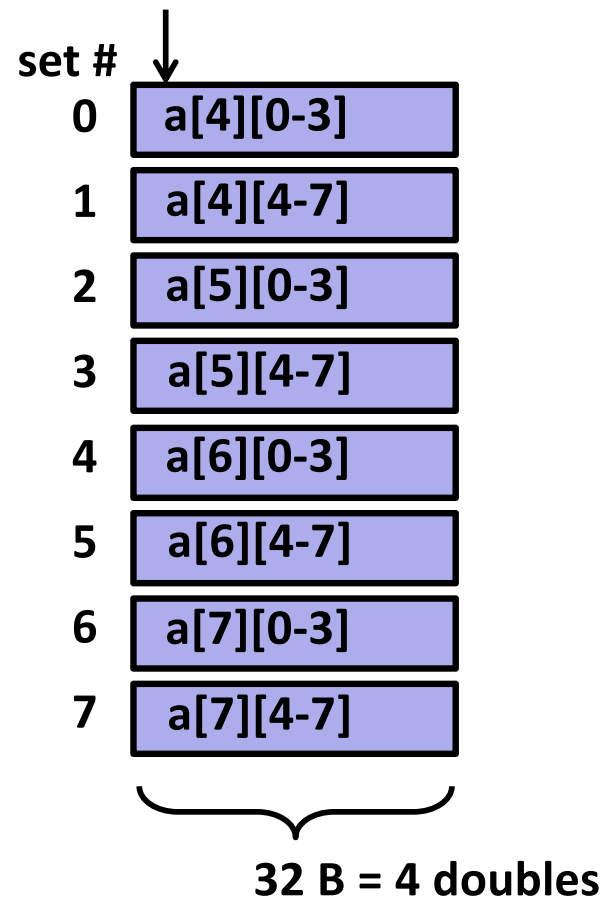
```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            sum += a[i][j];
    return sum;
}
```

Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache: 256 bytes
- Size of this array: 512 bytes
- Number of bits used for block offset: 5 bits
- Number of bits used for indexing sets: 3 bits
- Number of bits for tags: 24 bits
- Number of reads performed: 64
- Number of misses incurred: 16

Ignore the variables sum, i, j

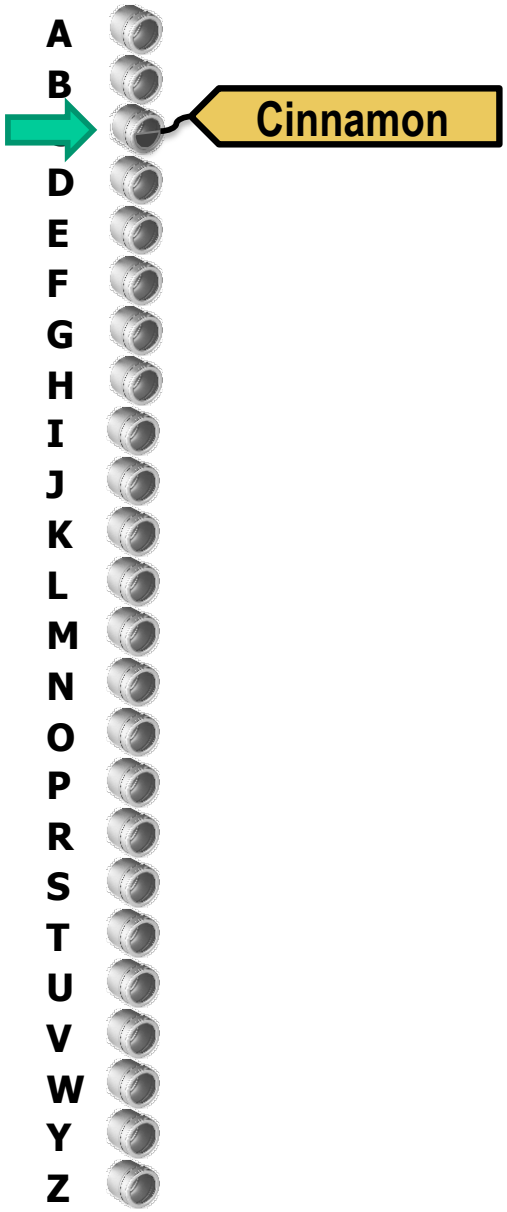
assume: cold (empty) cache, a[0][0] goes here



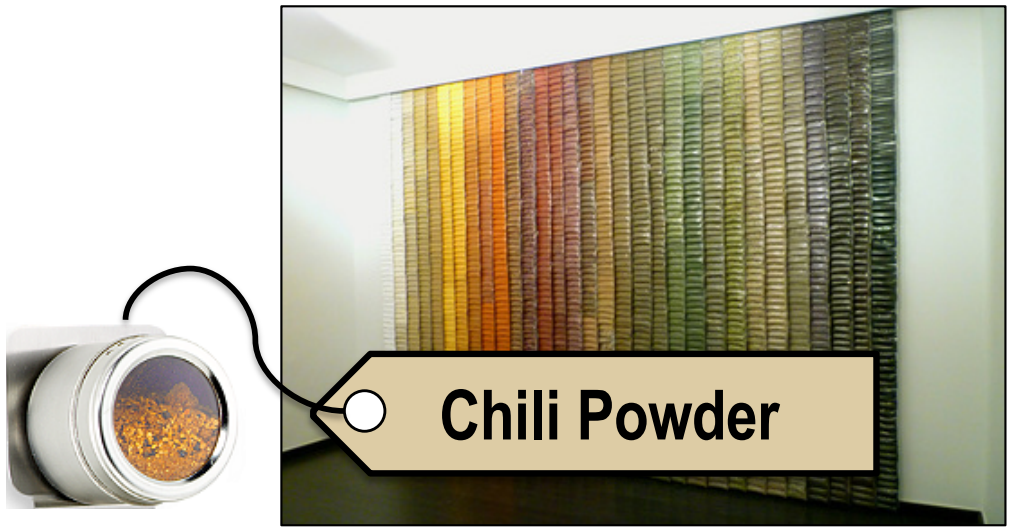
For every miss, we get 3 hits!

blackboard

Spicy Cache: Cincinnati Chili



- **Expensive!**
  - retrieve each spice from wall
  - *evict old spice*
- **Poor Utilization of Rack *C***



Utilizing Your Spice Rack

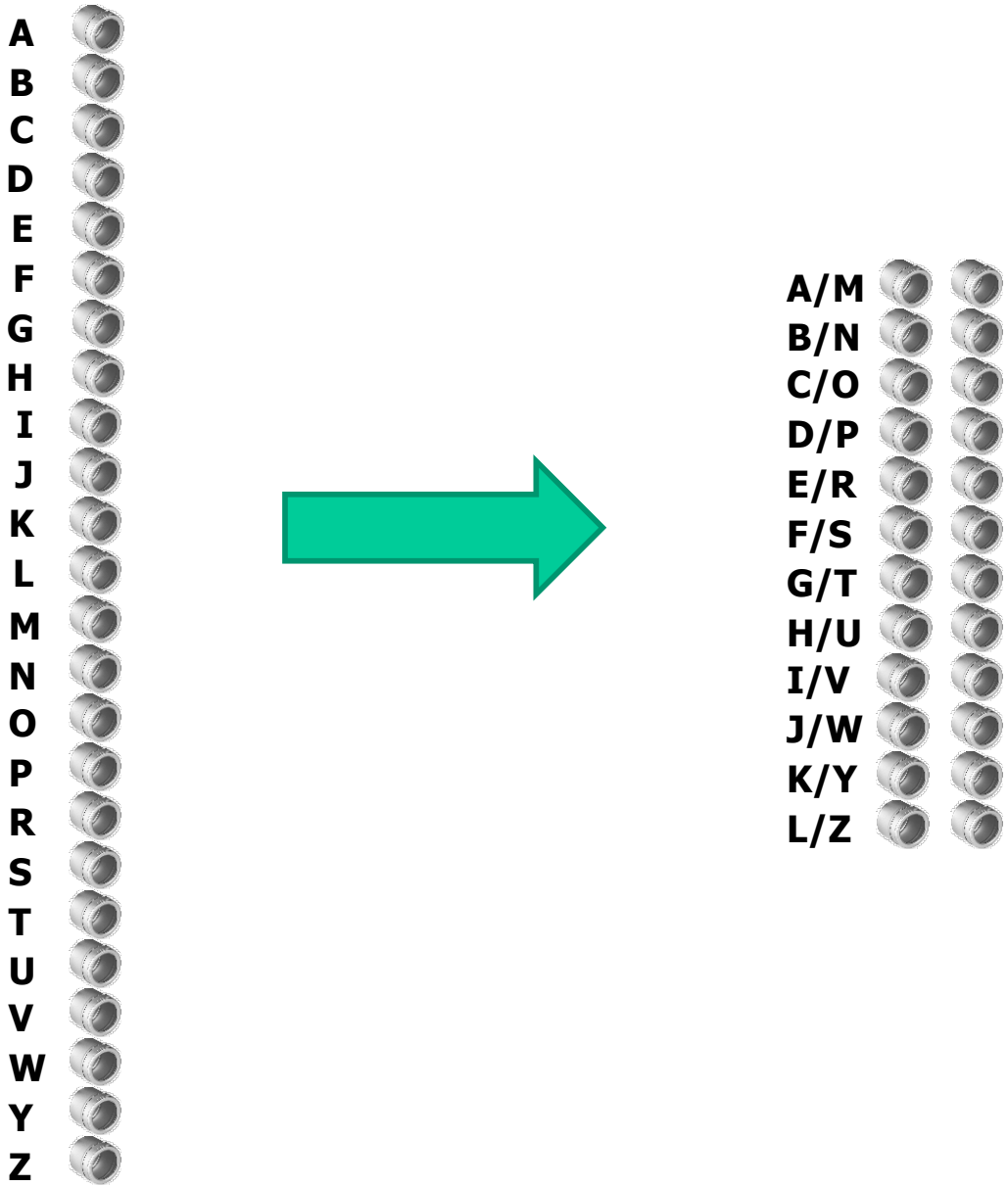
A	Allspice, Apple Pie Spice
B	Basil, Bay Leaves, Black Peppercorn
C	Cayenne, Chili Powder, Cinnamon, Cloves, Cumin, Curry Powder
D	Dill Weed
E	
F	
G	Garlic Powder, Ginger
H	
I	
J	
K	
L	
M	
N	Nutmeg
O	Onion Powder, Oregano
P	Paprika
R	Red Pepper Flakes, Rosemary
S	Saffron, Sage
T	Tarragon, Thyme
U	
V	Vanilla
W	
Y	
Z	

Do you notice anything?

I can't cook with cumin and curry powder? ☹

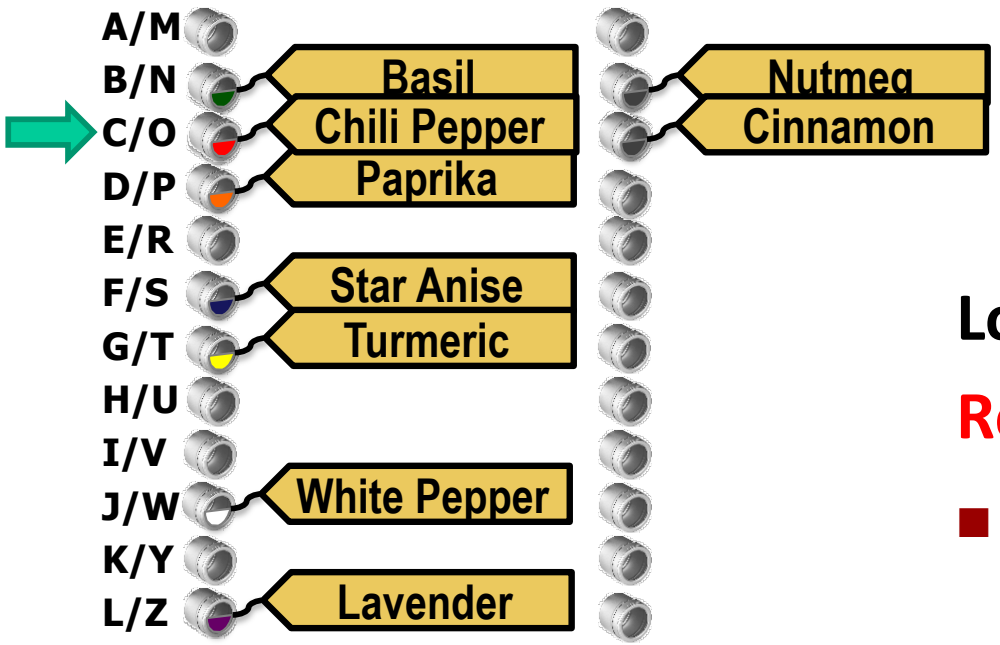
12 unused spice jars!

Re-arranging Your Spice Rack



World's first 2-way set associative spice rack!

2-way set associative spice rack

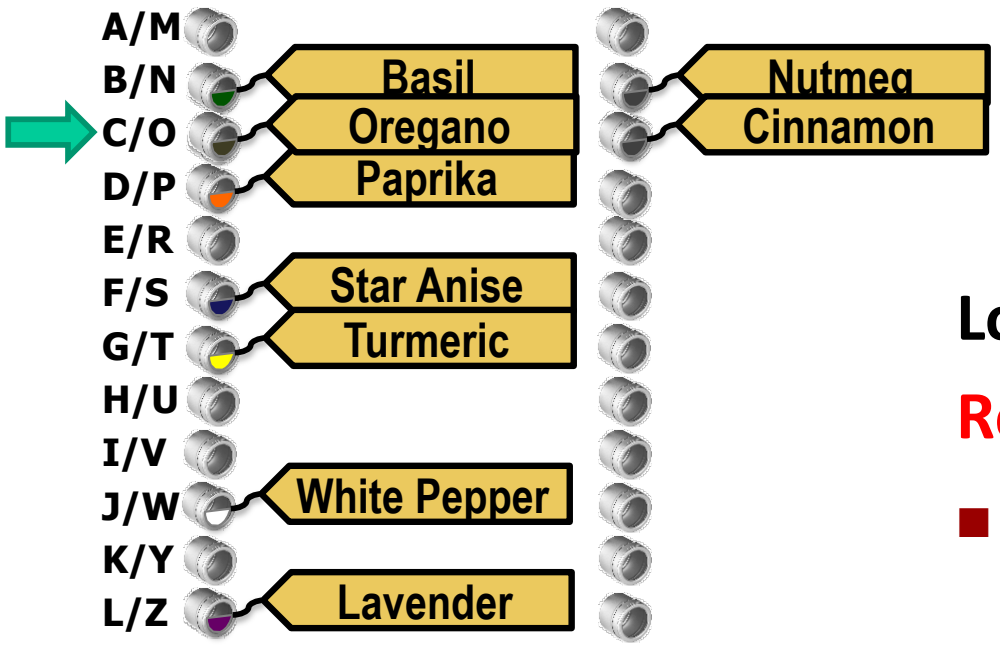


Look in the right row

Read 2 tags

- **Miss:**
  - retrieve from spice wall
  - evict one old spice (replacement policy dictates which one; typically least-recently used)
  - put it in the spice rack

2-way set associative spice rack

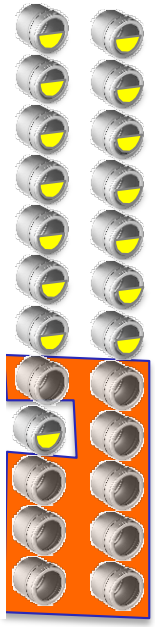


Look in the right row

Read 2 tags

- **Miss:**
  - retrieve from spice wall
  - evict one old spice (replacement policy dictates which one; typically least-recently used)
  - put it in the spice rack

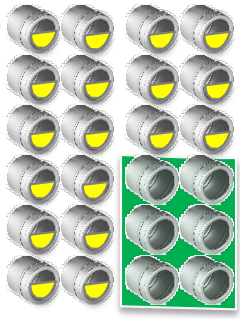
4-way Set Associative?



A/M Allspice, Apple Pie Spice
B/N Basil, Bay Leaves, Black Peppercorn, Nutmeg
C/O Cayenne, Chili Powder, Cinnamon, + 5 more
D/P Dill Weed, Paprika
E/R Red Pepper Flakes, Rosemary
F/S Saffron, Sage
G/T Garlic Powder, Ginger, Tarragon, Thyme
H/U
I/V
J/W Vanilla
K/Y
L/Z

2-way set associative:
8 unused spice jars

4-way set associative:
6 unused spice jars



A/G/M/T Allspice, Apple Pie Spice, Garlic Powder, Ginger, Tarragon, Thyme
B/H/N/U Basil, Bay Leaves, Black Peppercorn, Nutmeg
C/I/O/V Cayenne, Chili Powder, Cinnamon, + 5 more, Vanilla
D/J/P/W Dill Weed, Paprika
E/K/R/Y Red Pepper Flakes, Rosemary
F/L/S/Z Saffron, Sage

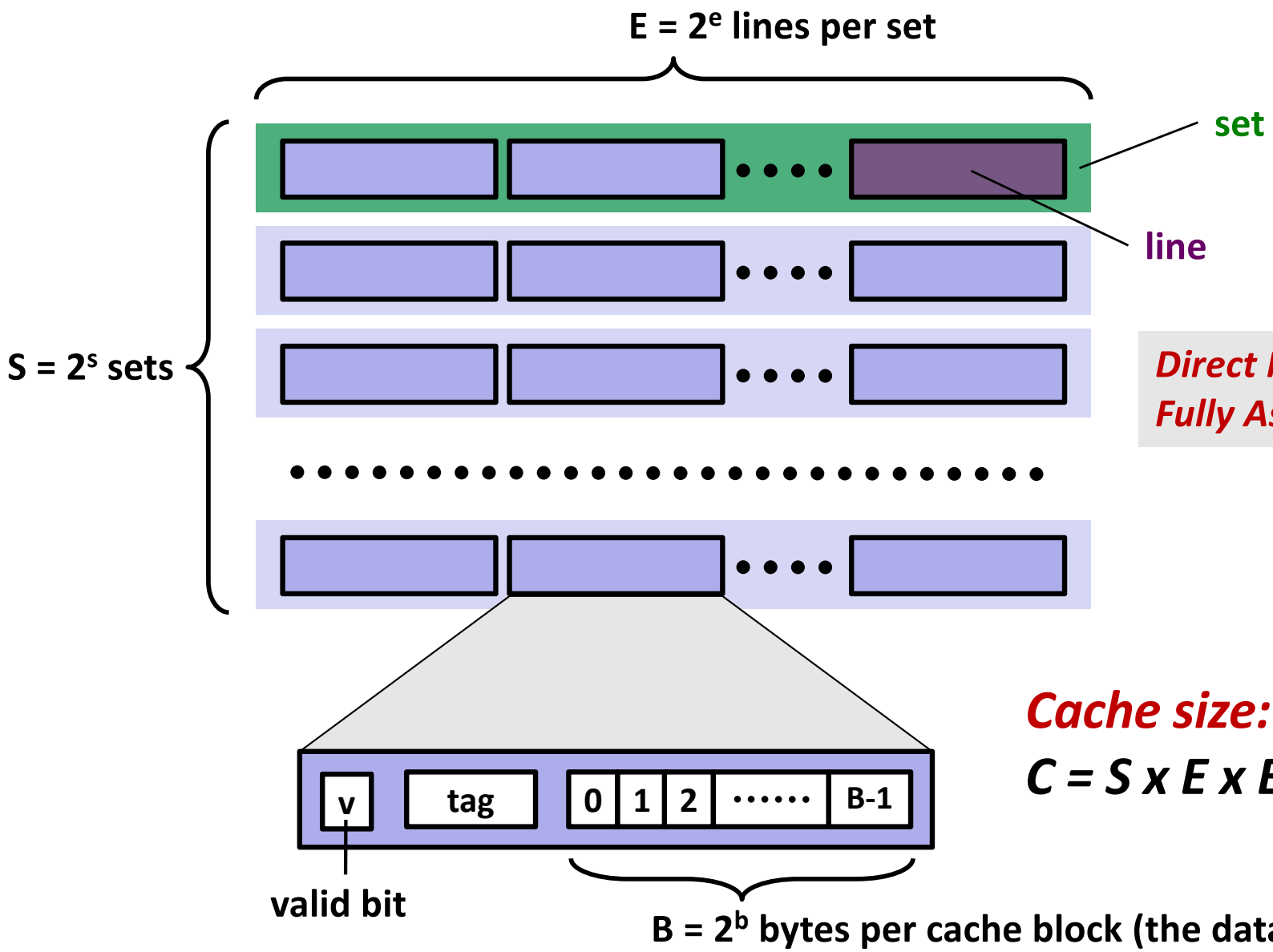
Fully-Associative!

A-Z

Allspice, Apple Pie Spice, Basil, Bay Leaves, Black Peppercorn, Cayenne, Chili Powder, Cinnamon, Cloves, Cumin, Curry Powder, Dill Weed, Garlic Powder, Ginger, Nutmeg, Onion Powder, Oregano, Paprika, Red Pepper Flakes, Rosemary, Saffron, Sage, Tarragon, Thyme, Vanilla

- **25 spices in 24 spice jars**
- **No unused spice jars!**
  - Yes!
- **No such thing as a free lunch**
  - Read 24 tags

General Cache Organization (S, E, B)

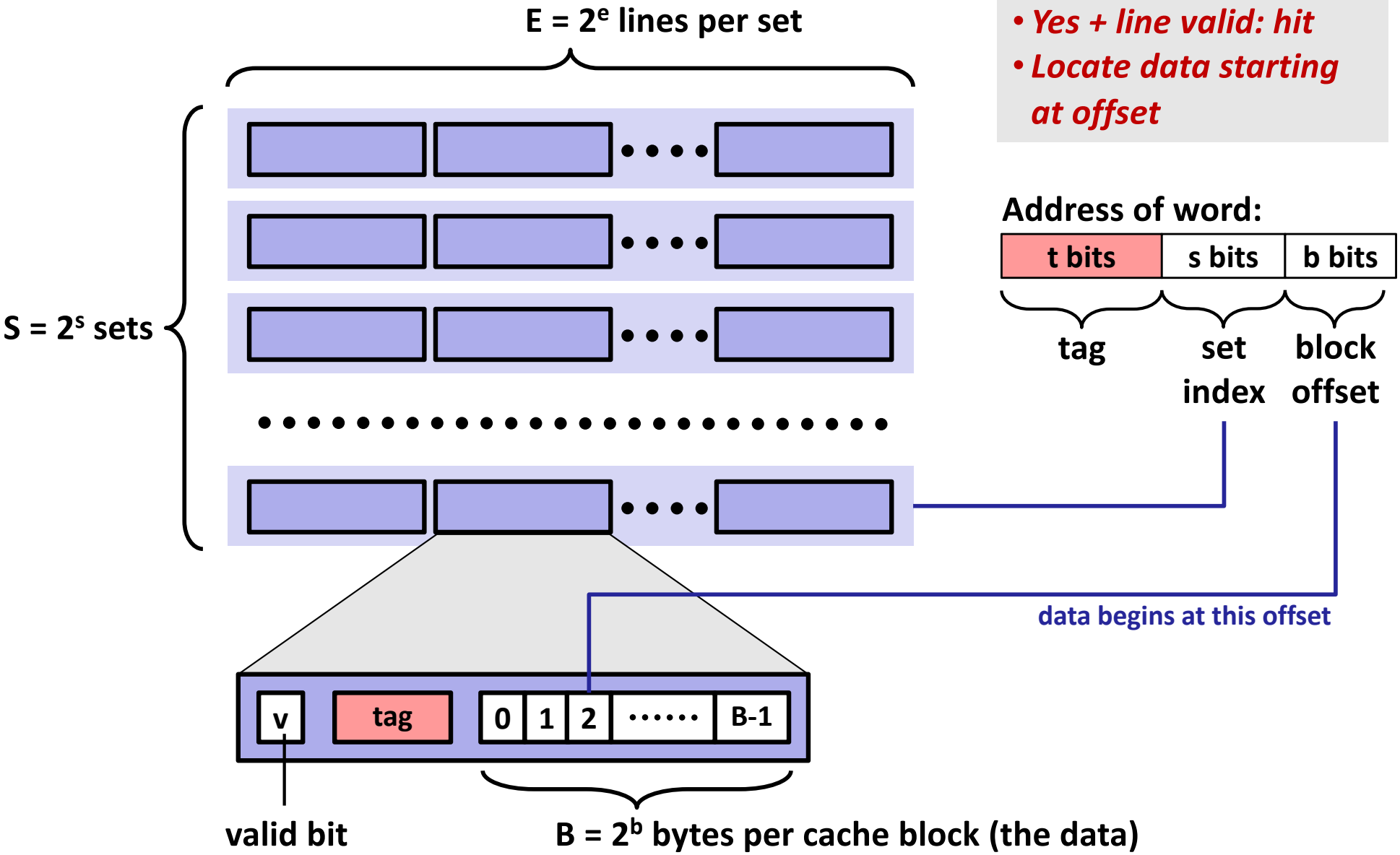


Direct Mapped: E=1
Fully Associative: S=1

Cache size:
C = S x E x B data bytes

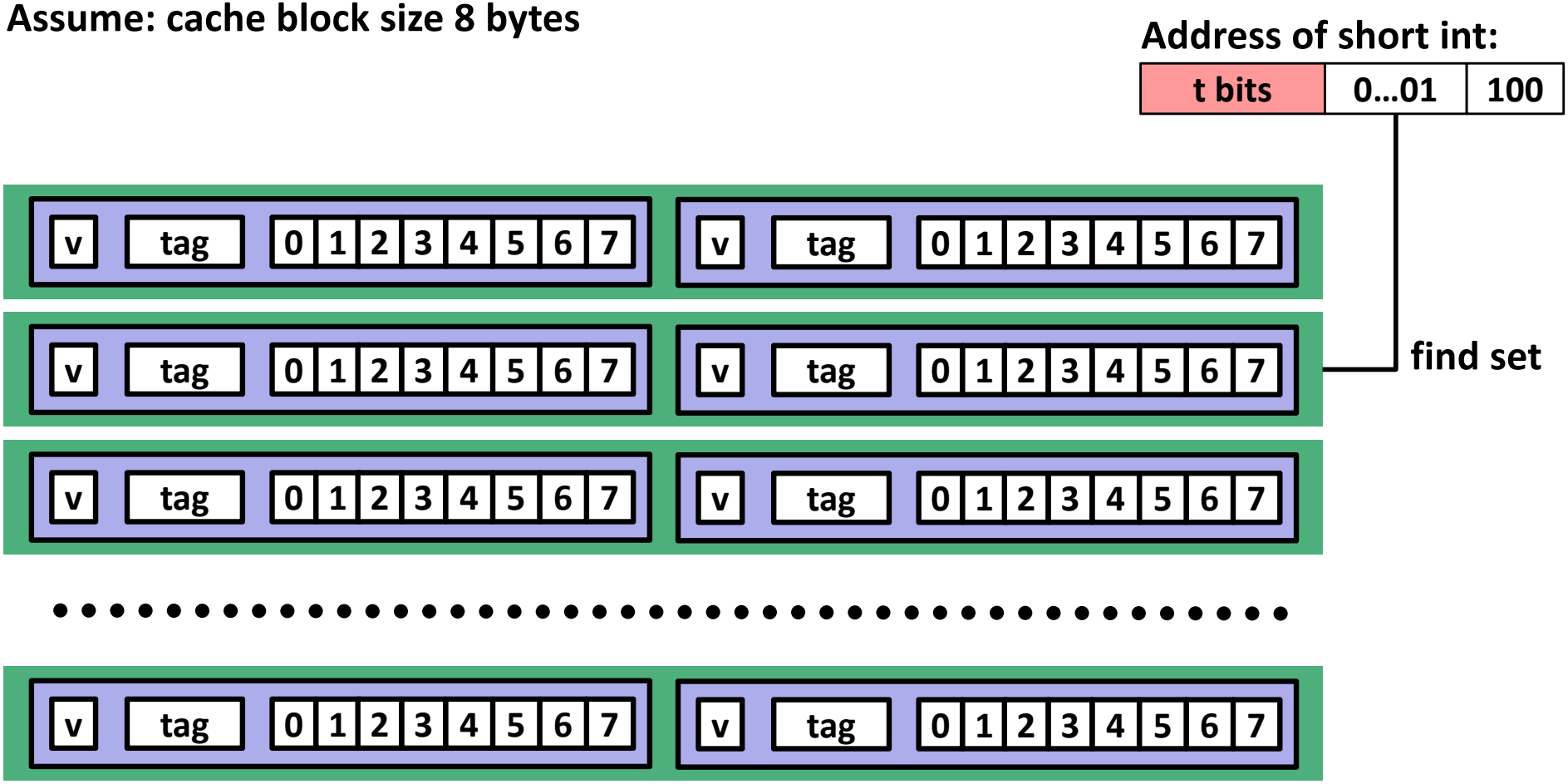
Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



E-way Set Associative Cache (Here: E = 2)

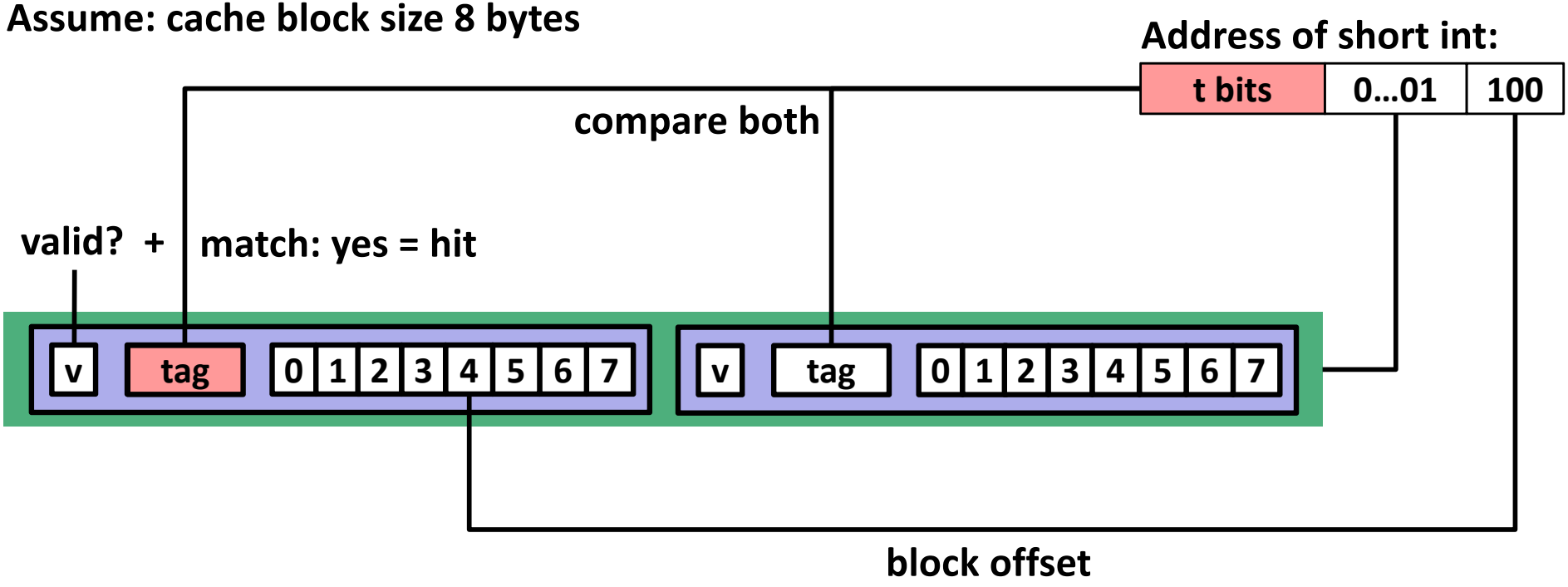
E = 2: Two lines per set
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

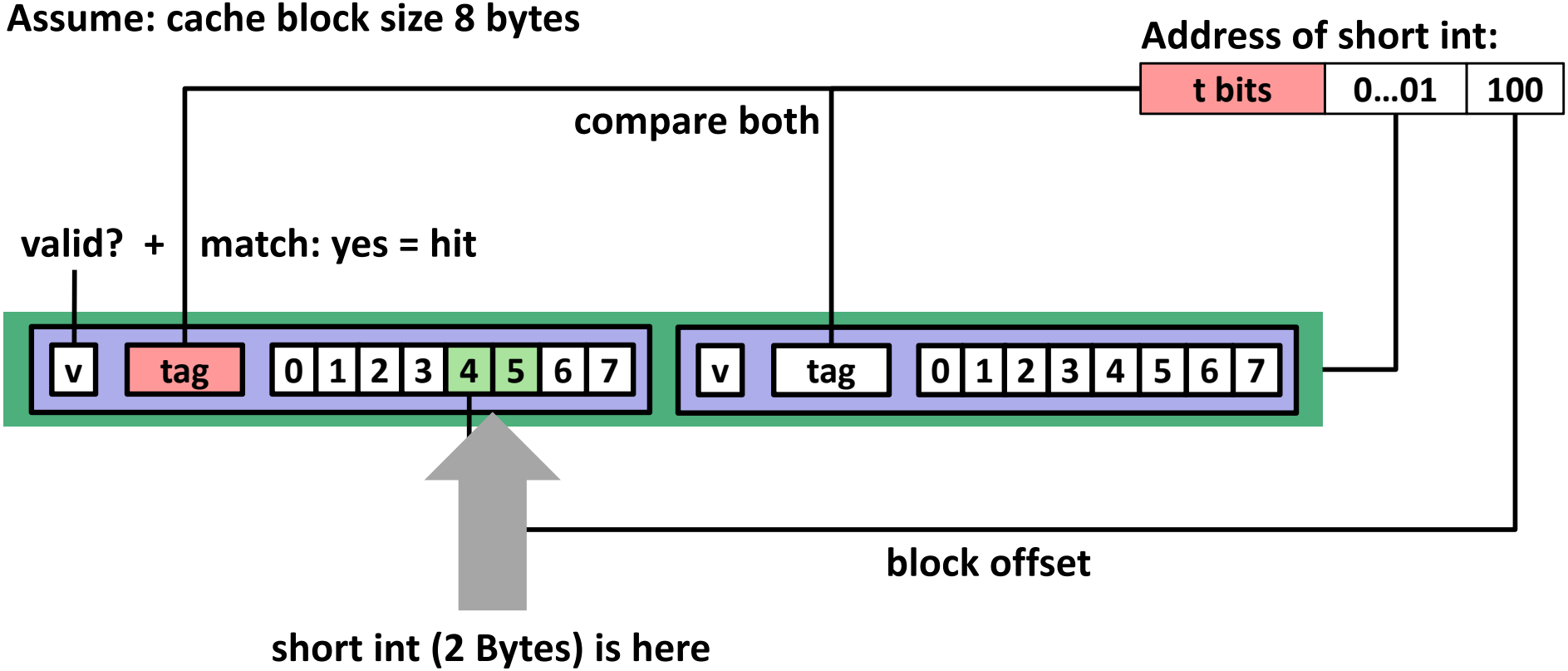
E = 2: Two lines per set

Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 block/set

Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss (cold)
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss (cold)
8	**[1000$_2$],**	miss (cold)
0	**[0000$_2$]**	hit

Compared to the direct-mapped cache, we avoided the conflict miss between block 0 and 8!

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Recap: What We Learned Thus Far

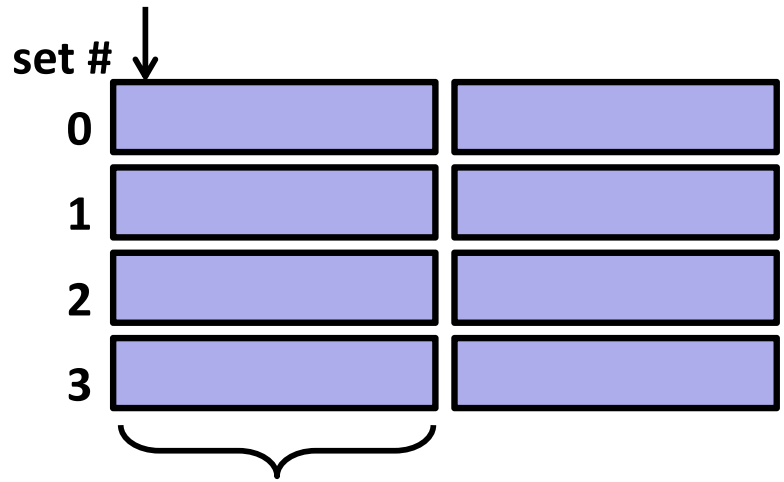
- How a multi-way set associative cache is organized
- A multi-way set associative cache can avoid conflict misses better than a direct mapped cache.
- How to look up a cache line in a multi-way set associative cache.

A High-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (j = 0; j < 8; j++)
        for (i = 0; i < 8; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache:
- Size of this array:
- Number of bits used for block offset:
- Number of bits used for indexing sets:
- Number of bits for tags:
- Number of reads performed:
- Number of misses incurred:

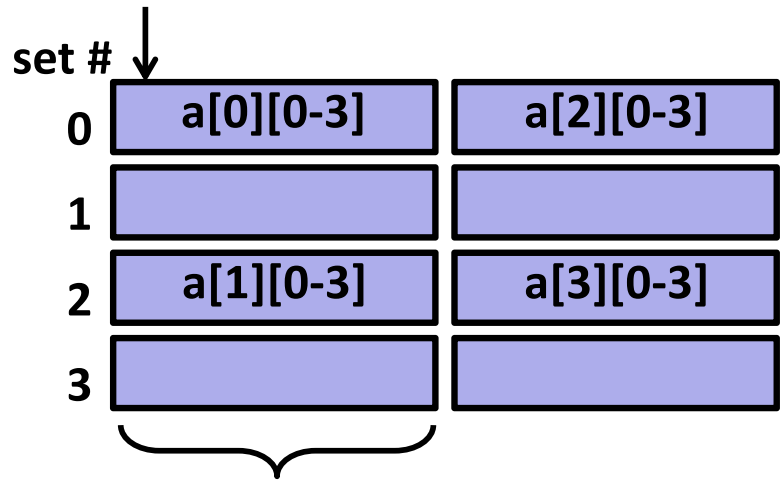
blackboard

A High-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (j = 0; j < 8; j++)
        for (i = 0; i < 8; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache: 256 bytes
- Size of this array: 512 bytes
- Number of bits used for block offset: 5 bits
- Number of bits used for indexing sets: 2 bits
- Number of bits for tags: 25 bits
- Number of reads performed: 64
- Number of misses incurred:

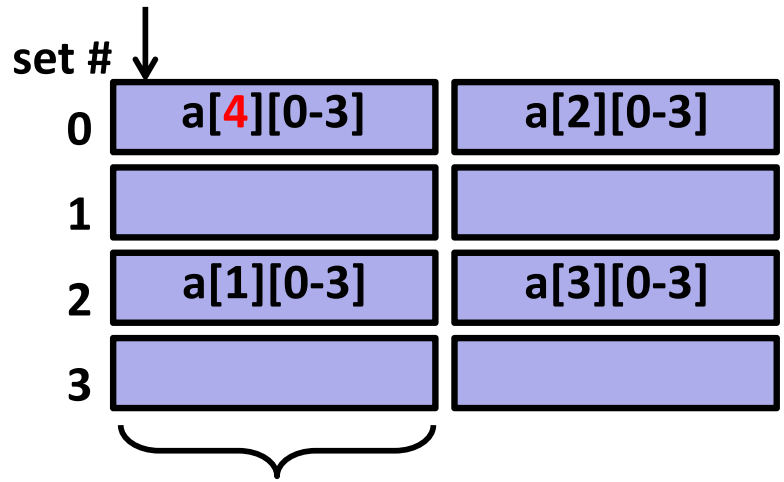
blackboard

A High-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[8][8]) {
    int i, j;
    double sum = 0;

    for (j = 0; j < 8; j++)
        for (i = 0; i < 8; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache, a[0][0] goes here



Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache: 256 bytes
- Size of this array: 512 bytes
- Number of bits used for block offset: 5 bits
- Number of bits used for indexing sets: 2 bits
- Number of bits for tags: 25 bits
- Number of reads performed: 64
- Number of misses incurred: 64

The cache is *thrashing* between a[i][*] and a[i+4][*]!

Every read is a miss!

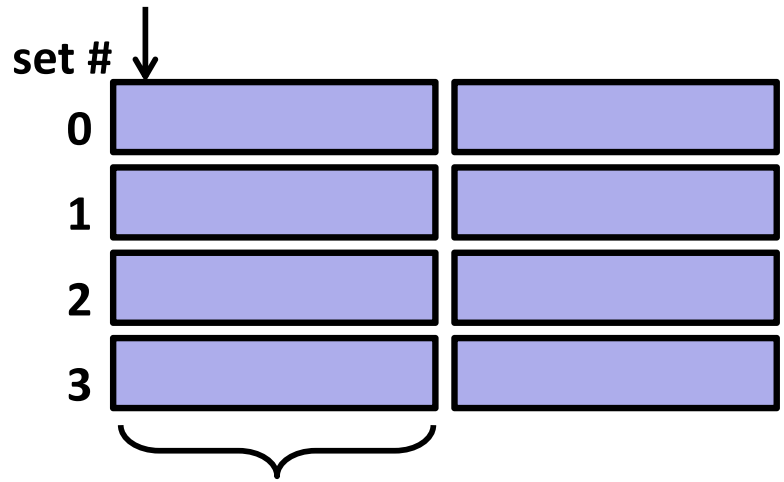
blackboard

A High-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[8][12]) {
    int i, j;
    double sum = 0;

    for (j = 0; j < 12; j++)
        for (i = 0; i < 8; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



Assume M = 2^{32} (32-bit addresses)

- Capacity of this cache:
- Size of this array:
- Number of bits used for block offset:
- Number of bits used for indexing sets:
- Number of bits for tags:
- Number of reads performed:
- Number of misses incurred:

blackboard