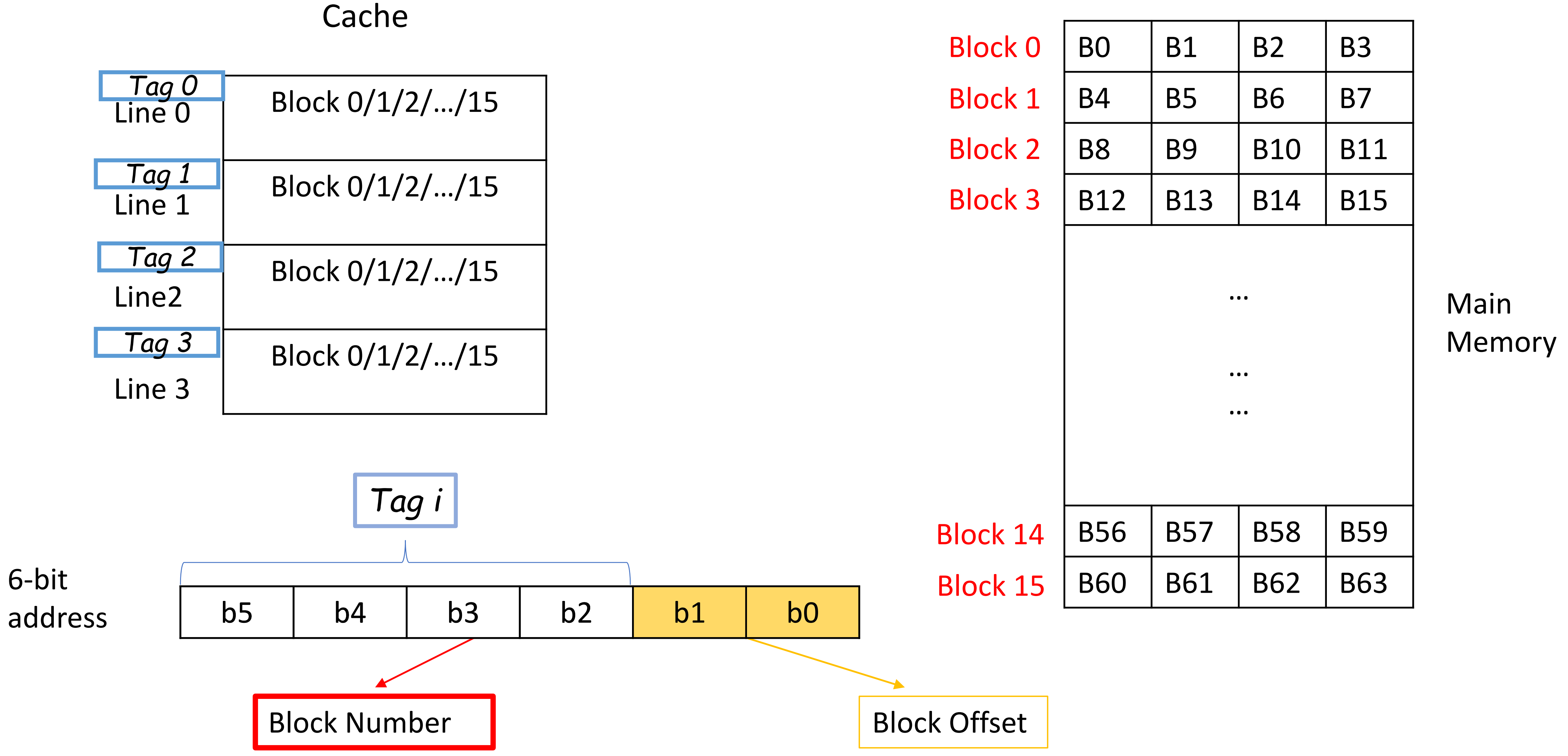
Cache Mapping, Cache Lab

Fully Associative Mapping

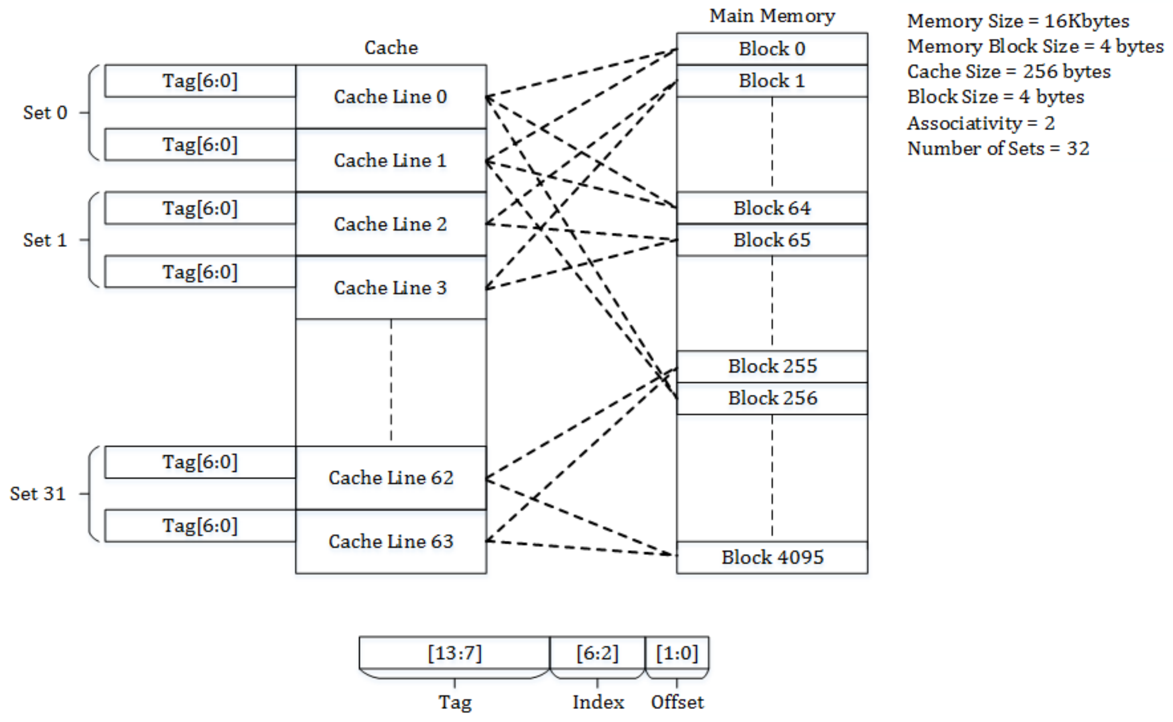
- A Main Memory Block can be mapped to any Cache line.
  - **Advantage**
    - Better Cache Hit Rate
  - **Disadvantages**
    - Slow because the valid bit and tag of every cache line has to be compared.
    - Expensive due to the high cost of associative-comparison hardware.

Fully Associative Mapping

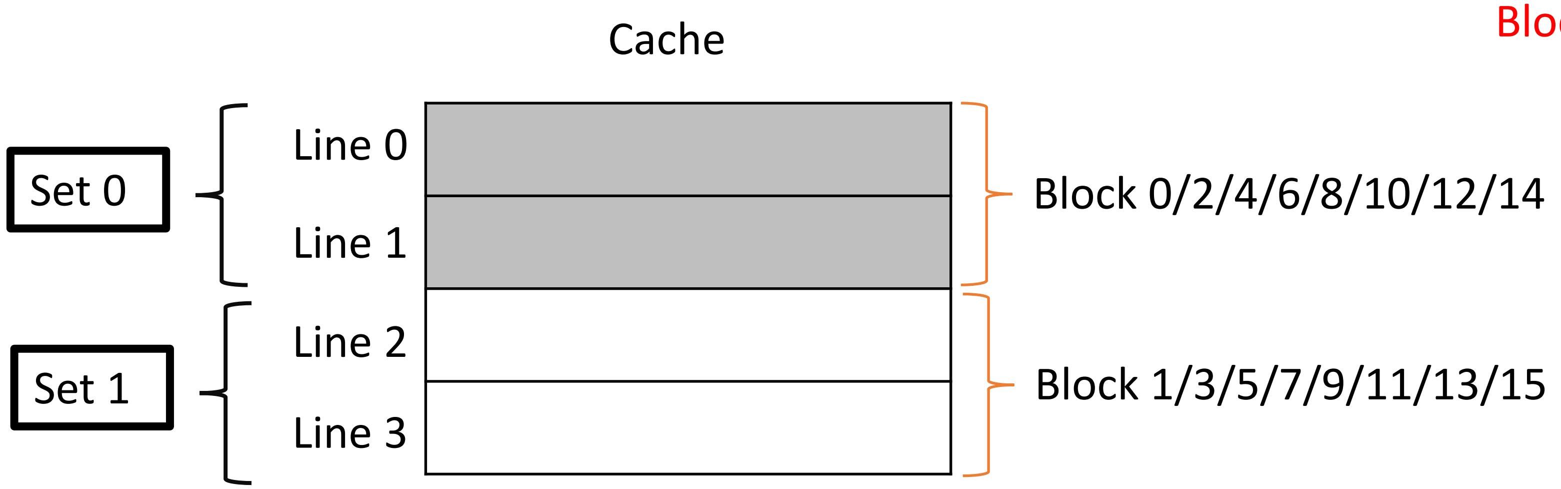


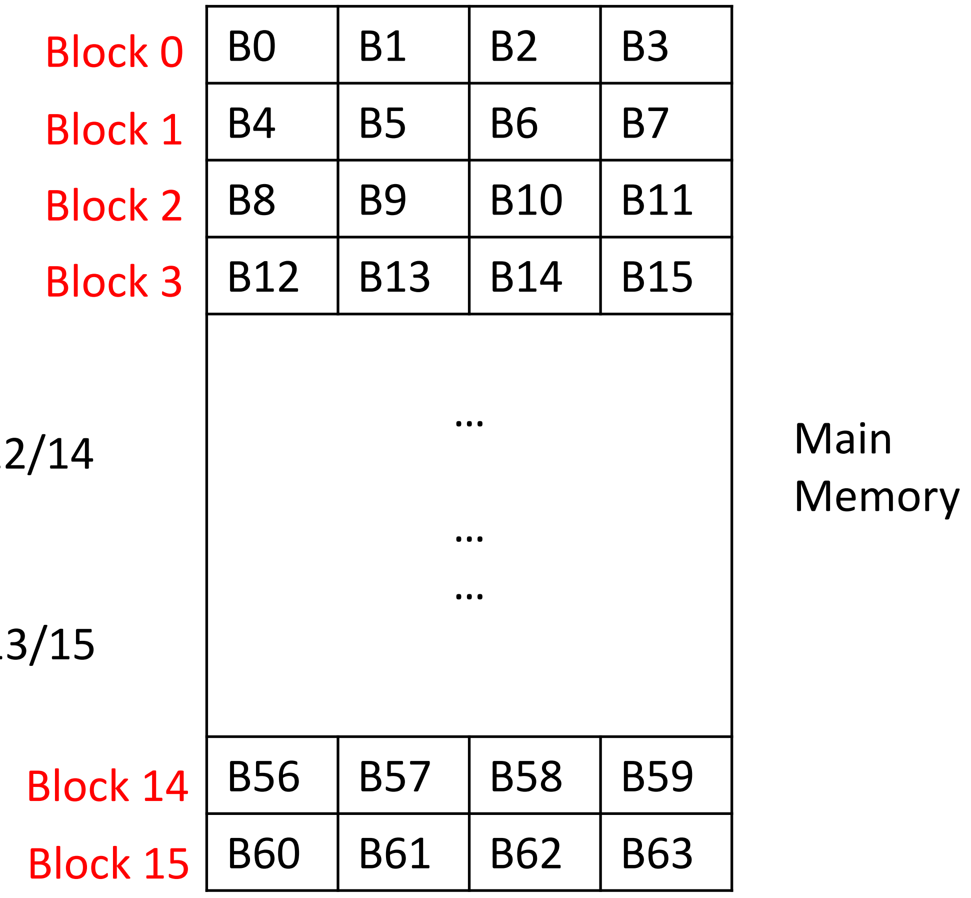
Set Associative Mapping

- Cache lines grouped into sets
- A main memory block is mapped to a set
  - Associative Mapping within a set
- Tradeoff between Direct Mapping and Fully Associative Mapping
- **Disadvantages**
  - Can still suffer from conflict miss.



Set Associative Mapping



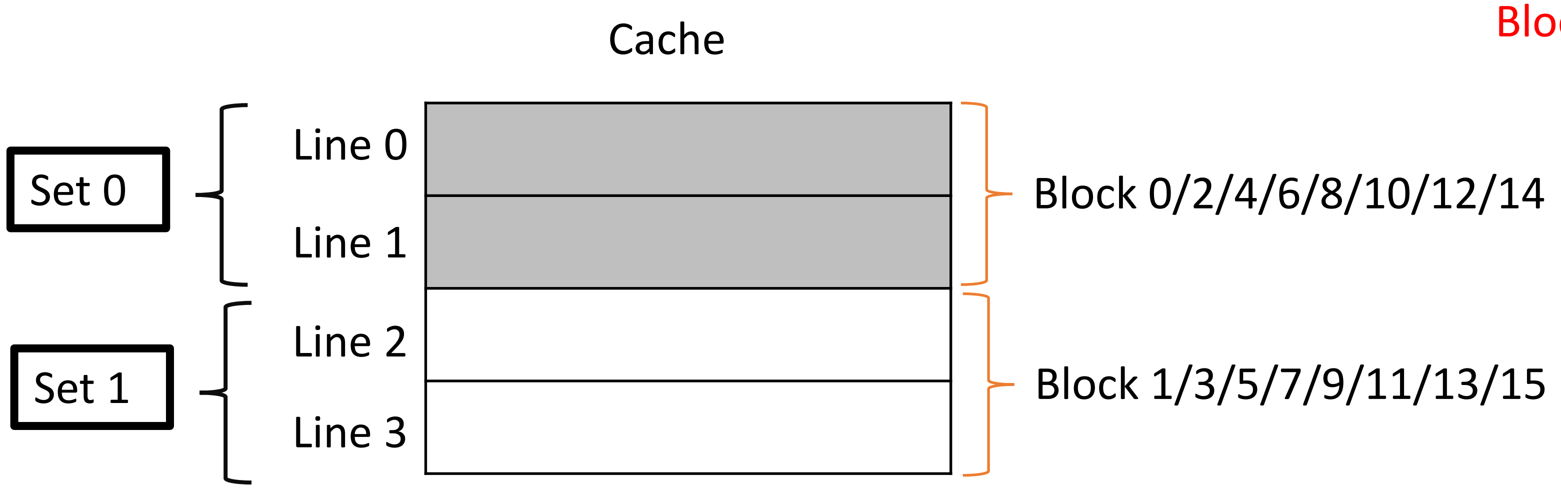


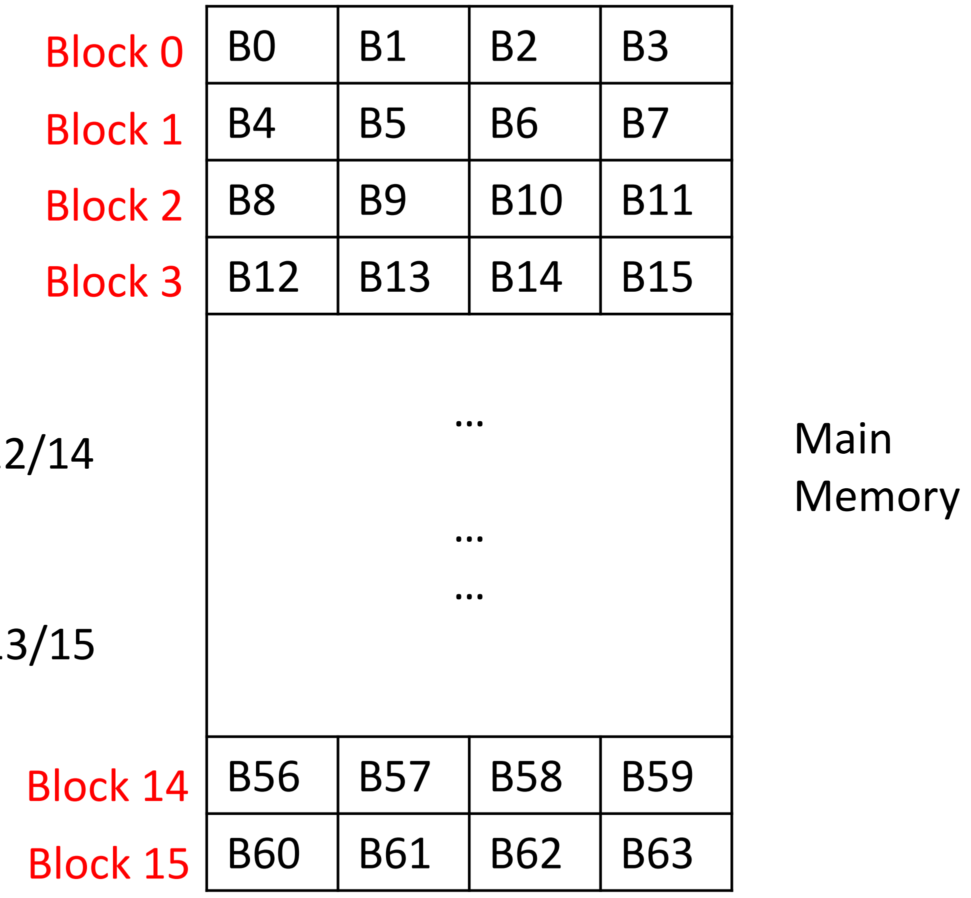
Block 0	B0	B1	B2	B3
Block 1	B4	B5	B6	B7
Block 2	B8	B9	B10	B11
Block 3	B12	B13	B14	B15
	…	…	…	…
Block 14	B56	B57	B58	B59
Block 15	B60	B61	B62	B63

- A MM Block is Direct Mapped to a Set
- Set Number/Index = **K** mod **s**
  - **K** = Block Number
  - **s** = Number of Sets

- Number of Bytes in Main Memory = 64 = 2^6
- How many addresses?
  - 64
- How many bits per address?
  - 6-bit address

Set Associative Mapping

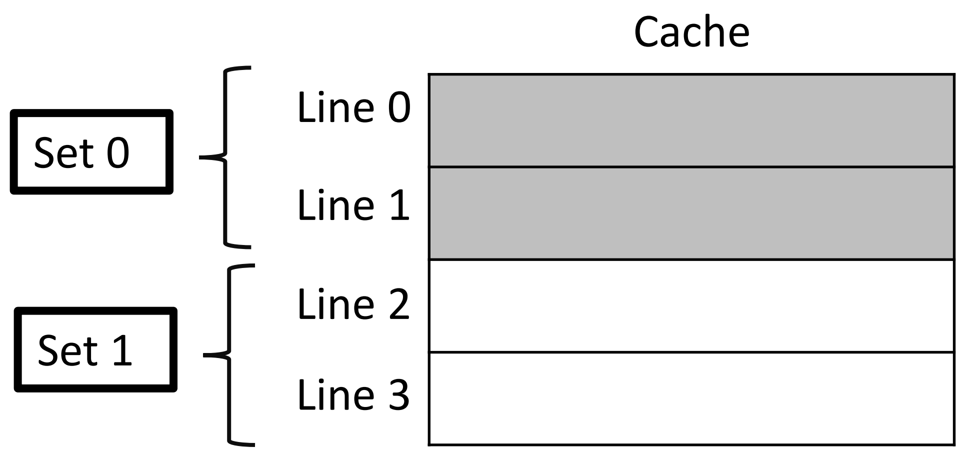


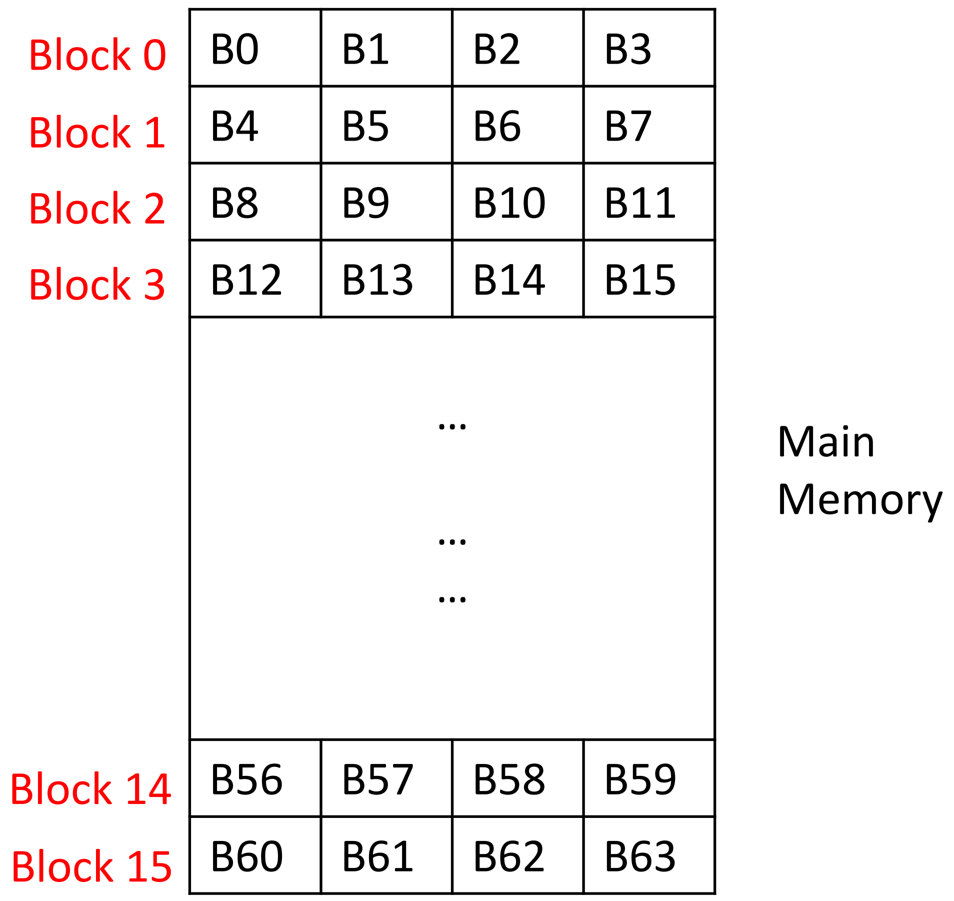


- How many bits for Set Index?
    - 1
- How many bits for Tag?
    - 3

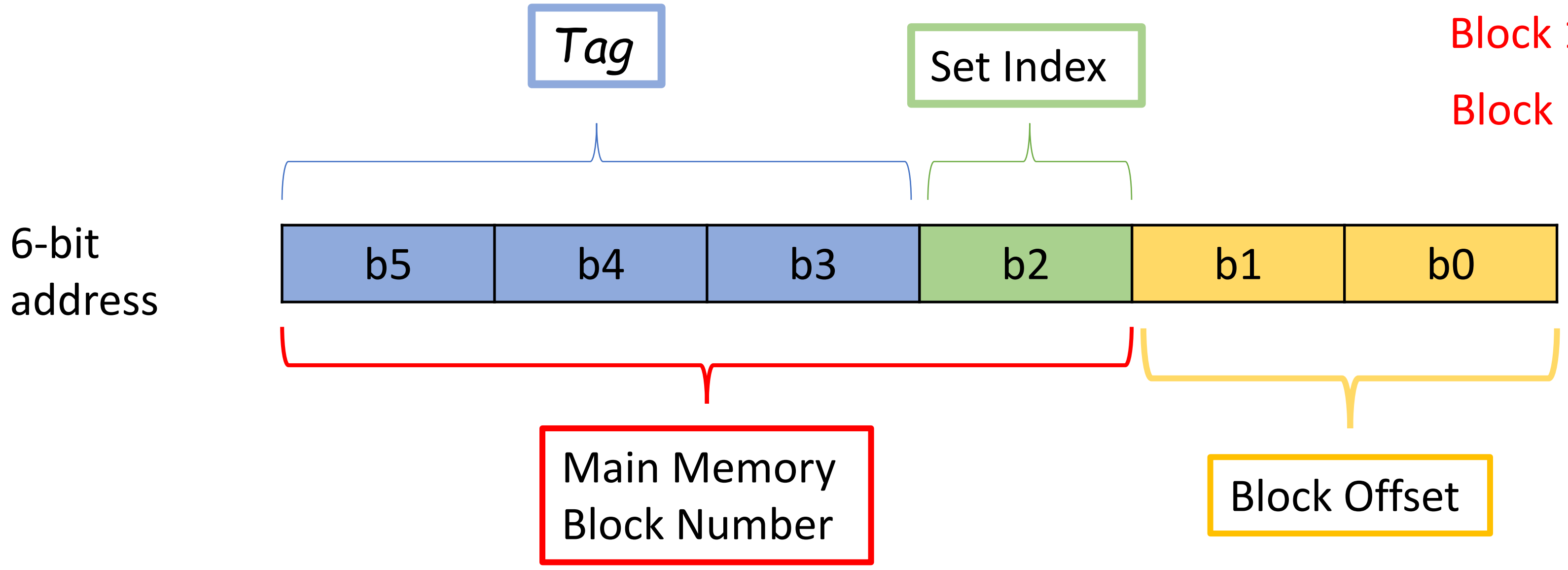
- Number of Bytes in Main Memory = 64 = 2^6
- How many addresses?
    - 64
- How many bits per address?
    - 6-bit address

Set Associative Mapping





Block 0	B0	B1	B2	B3
Block 1	B4	B5	B6	B7
Block 2	B8	B9	B10	B11
Block 3	B12	B13	B14	B15
	…	…	…	
Block 14	B56	B57	B58	B59
Block 15	B60	B61	B62	B63



Cache Tutorial

Accessing a Cache (Hit or Miss?)

Assume the following caches all have block size $K = 4$ and are in the current state shown (you can ignore "–"). All values are shown in hex. Tag fields are NOT padded, while bytes of the cache blocks are shown in full. The word size for the machine with these caches is 12 bits (i.e. addresses are 12 bits long)

Direct-Mapped:

Set	Valid	Tag	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	–	–	–	–	–
2	0	–	–	–	–	–
3	1	D	DE	AF	BA	DE
4	0	–	–	–	–	–
5	0	–	–	–	–	–
6	1	13	31	14	15	93
7	0	–	–	–	–	–

Set	Valid	Tag	B0	B1	B2	B3
8	0	–	–	–	–	–
9	1	0	01	12	23	34
A	1	1	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	–	–	–	–	–
D	1	11	C0	04	39	AA
E	0	–	–	–	–	–
F	1	F	FF	6F	30	0

Offset bits: 2

Index bits: 4

Tag bits: 6

Cache Tutorial

	Hit or Miss?	Data returned
a) Read 1 byte at `0x7AC`	Miss	—
b) Read 1 byte at `0x024`	Hit	0x01
c) Read 1 byte at `0x99F`	Miss	—

Address: 011110101100

Block Offset : 0

Set/Line Index = 11 = 0xB

Tag = 30 = 0x1E

Set	Valid	Tag	B0	B1	B2	B3
8	0	–	–	–	–	–
9	1	0	01	12	23	34
A	1	1	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	–	–	–	–	–
D	1	11	C0	04	39	AA
E	0	–	–	–	–	–
F	1	F	FF	6F	30	0

Cache Tutorial

	Hit or Miss?	Data returned
a) Read 1 byte at `0x7AC`	Miss	—
b) Read 1 byte at `0x024`	Hit	0x01
c) Read 1 byte at `0x99F`	Miss	—

Address: 000000100100

Block Offset : 0

Set/Line Index = 0x9

Tag: 0x0

Set	Valid	Tag	B0	B1	B2	B3
8	0	—	—	—	—	—
9	1	0	01	12	23	34
A	1	1	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	F	FF	6F	30	0

Cache Tutorial

2-way Set Associative:

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	0	—	—	—	—	—
2	1	3	4F	D4	A1	3B
3	0	—	—	—	—	—
4	0	6	CA	FE	F0	0D
5	1	21	DE	AD	BE	EF
6	0	—	—	—	—	—
7	1	11	00	12	51	55

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	1	2F	01	20	40	03
2	1	0E	99	09	87	56
3	0	—	—	—	—	—
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	37	22	B6	DB	AA
7	0	—	—	—	—	—

Offset bits: 2

Index bits: 3

Tag bits: 7

Cache Tutorial

	Hit or Miss?	Data returned
a) Read 1 byte at `0x435`	Hit	0xAD
b) Read 1 byte at `0x388`	Miss	—
c) Read 1 byte at `0x0D3`	Miss	—

Address: 01000011 0101

Block Offset = 1

Set Index = 5 = 0x5

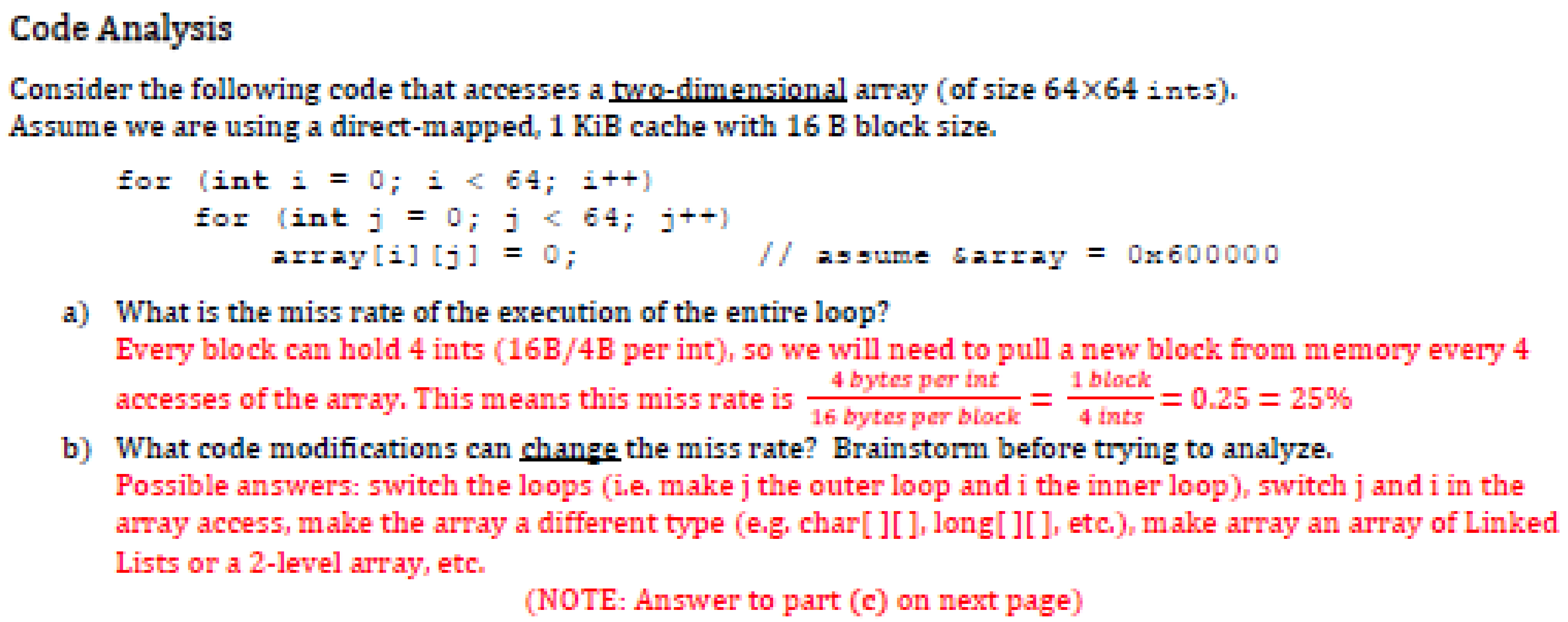
Tag = 0x21

2-way Set Associative:

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	0	—	—	—	—	—
2	1	3	4F	D4	A1	3B
3	0	—	—	—	—	—
4	0	6	CA	FE	F0	0D
5	1	21	DE	$\boxed{\text{AD}}$	BE	EF
6	0	—	—	—	—	—
7	1	11	00	12	51	55

Set	Valid	Tag	B0	B1	B2	B3
0	0	—	—	—	—	—
1	1	2F	01	20	40	03
2	1	0E	99	09	87	56
3	0	—	—	—	—	—
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	37	22	B6	DB	AA
7	0	—	—	—	—	—

Cache Tutorial



Code Analysis

Consider the following code that accesses a two-dimensional array (of size 64×64 `ints`).
Assume we are using a direct-mapped, 1 KiB cache with 16 B block size.

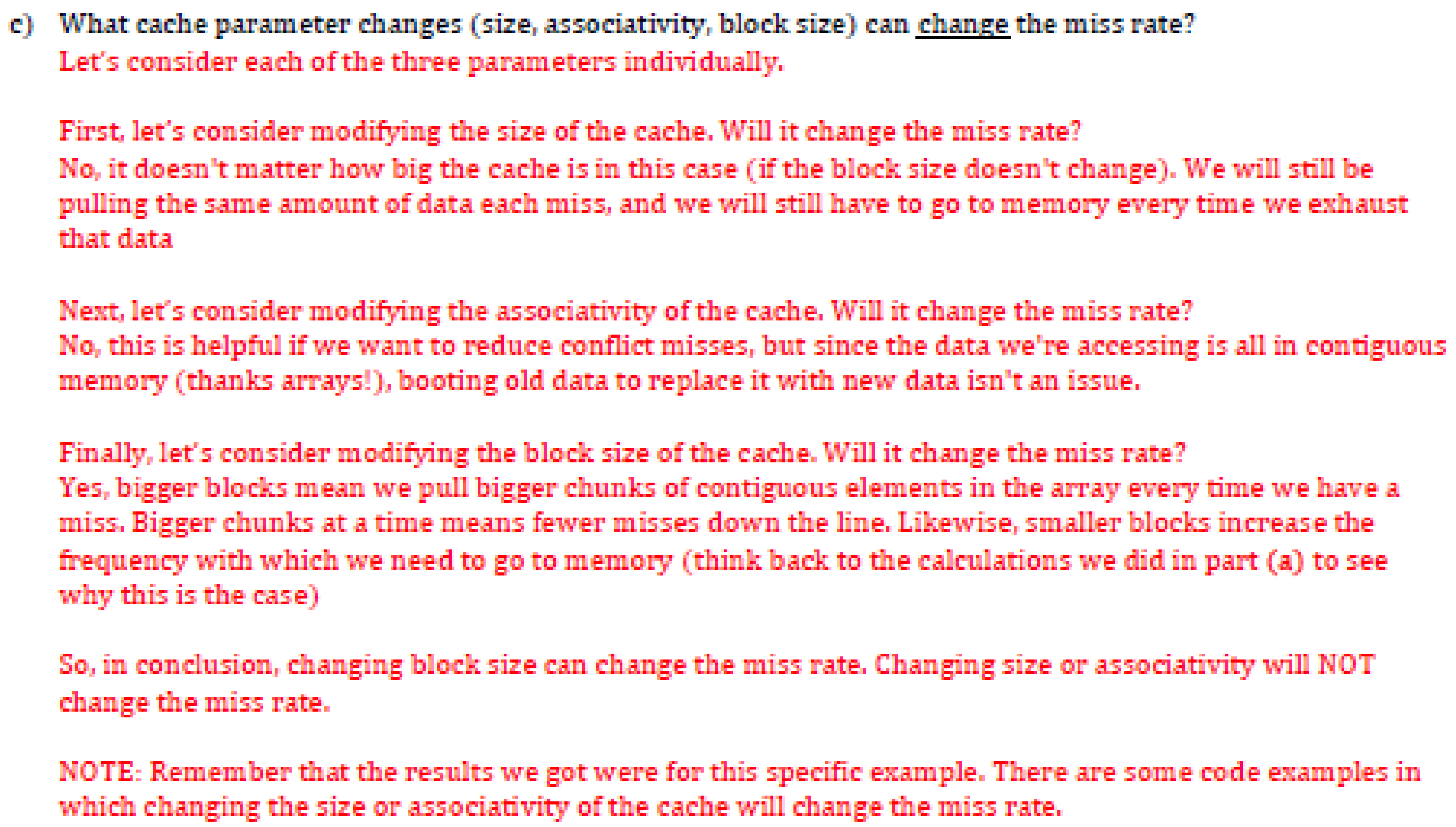
```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        array[i][j] = 0;            // assume &array = 0x600000
```

a) What is the miss rate of the execution of the entire loop?
Every block can hold 4 ints (16B/4B per int), so we will need to pull a new block from memory every 4 accesses of the array. This means this miss rate is $\frac{4 \text{ bytes per int}}{16 \text{ bytes per block}} = \frac{1 \text{ block}}{4 \text{ ints}} = 0.25 = 25\%$

b) What code modifications can change the miss rate? Brainstorm before trying to analyze.
Possible answers: switch the loops (i.e. make j the outer loop and i the inner loop), switch j and i in the array access, make the array a different type (e.g. char[ ][ ], long[ ][ ], etc.), make array an array of Linked Lists or a 2-level array, etc.

(NOTE: Answer to part (c) on next page)

Cache Tutorial



c) What cache parameter changes (size, associativity, block size) can change the miss rate?

Let's consider each of the three parameters individually.

First, let's consider modifying the size of the cache. Will it change the miss rate?
No, it doesn't matter how big the cache is in this case (if the block size doesn't change). We will still be pulling the same amount of data each miss, and we will still have to go to memory every time we exhaust that data

Next, let's consider modifying the associativity of the cache. Will it change the miss rate?
No, this is helpful if we want to reduce conflict misses, but since the data we're accessing is all in contiguous memory (thanks arrays!), booting old data to replace it with new data isn't an issue.

Finally, let's consider modifying the block size of the cache. Will it change the miss rate?
Yes, bigger blocks mean we pull bigger chunks of contiguous elements in the array every time we have a miss. Bigger chunks at a time means fewer misses down the line. Likewise, smaller blocks increase the frequency with which we need to go to memory (think back to the calculations we did in part (a) to see why this is the case)

So, in conclusion, changing block size can change the miss rate. Changing size or associativity will NOT change the miss rate.

NOTE: Remember that the results we got were for this specific example. There are some code examples in which changing the size or associativity of the cache will change the miss rate.

Cache Simulator Demo

Let's get some practice with the cache simulator! First, go to:

http://www.cs.pitt.edu/~vinicius/cachesim/

At the top you'll see 4 boxed regions:

- System Parameters † This lets you play around with the structure/format of the cache
- Manual Memory Access † This is where you actually make reads and writes to memory
- History An interactive log of executed accesses. You can type/paste accesses here, too!
- Simulation Messages Describes the most recent actions made by the simulator.

† These include "Explain" toggles that walk you through execution step-by-step.

a) Set the following System Parameters (but *don't* generate the system yet):

Address Width → 6, Cache Size → 16, Block Size → 4, Associativity → 2, leave the rest at default values.

Based on just the system parameter numbers above shown, predict the following:

i) Highest memory address: 0b 0011 1111 ii) Number of sets in cache: 2

[*Click "Generate System" to verify your responses*]

b) We are about to **READ** the byte at the address **0x2A**. Predict the following:

i) This block will be placed in set #: 0 ii) The stored tag bits will be: 0b 101

iii) The 4 bytes of *data* in this block are (in order): 0xe9, 0x36, 0xae, 0x32

[*Enter "2a" into the Read Addr and click "Read" to verify your responses*]

c) We are about to **WRITE** the byte **0xB1** to the address **0x1B**. Predict the following:

i) This block will be placed in set #: 0 ii) The stored tag bits will be: 0b 011

[*Enter "1b" into the Write Addr and "b1" into the Write Byte and then click "Write" to verify your responses*]

iii) Notice that the value of the byte at address 0x1B is different in the cache and memory.

What indicates this disparity in the cache? The dirty bit

What would have happened if our write miss policy were "**No Write-Allocate**" instead?

We would write directly to memory and not cache the block starting at 0x18

d) We are about to **READ** the byte at address **0x01**. Predict the following:

i) This block will be placed in set #: 0 ii) The stored tag bits will be: 0b 000

iii) Will this access cause a conflict/replacement? (circle one) [✓ Yes] No

iv) If yes, which block will be evicted? (circle one) [✓ Read from (b)] Write from (c)

[*Enter "01" into the Read Addr and click "Read" to verify your responses*]

e) We are about to **WRITE** the byte **0xE9** to the address **0x1C**. Predict the following:

i) This block will be placed in set #: 1 ii) The stored tag bits will be: 0b 011

iii) Will this access cause a conflict/replacement? (circle one) Yes [✓ No]

iv) If yes, which block will be evicted? Read from (b) Write from (c) Read from (d)

[*Enter "1c" into the Write Addr and "e9" into the Write Byte and then click "Write" to verify your responses*]

f) At this point, your **History** should show:

```
  R(0x2a) = M
  W(0x1b, 0xb1) = M
  R(0x01) = M
  W(0x1c, 0xe9) = M
>
```

Append the bolded text below so that your History looks like:

```
  R(0x2a) = M
  W(0x1b, 0xb1) = M
  R(0x01) = M
  W(0x1c, 0xe9) = M
> W(0x03, 0xff)
  R(0x27)
  R(0x10)
  W(0x1d, 0x00)
```

[*Click "Load." You'll notice that " = ?" is appended to each of these new memory accesses*]

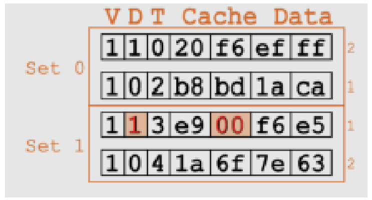
Predict if '?' will resolve to Hit (H) or Miss (M) for each of the new accesses:

i) W(0x03, 0xff) = H ii) R(0x27) = M

iii) R(0x10) = M iv) W(0x1d, 0x00) = H

[*Click the down arrow (↓) to verify your responses for each access*]

g) The cache, after the 8 executions detailed above, should look like this:



	V	D	T	Cache	Data			
Set 0	1	1	0	20	f6	ef	ff	2
	1	0	2	b8	bd	1a	ca	1
Set 1	1	1	3	e9	00	f6	e5	1
	1	0	4	1a	6f	7e	63	2

The small numbers on the right (outside of the sets) indicate how recently used each line is within the set, with smaller numbers being *more recently* used).

i) An LRU replacement policy will evict which block on the next conflict in set 0? [✓ Line 1] Line 2

ii) What is one benefit of using **LRU** over **Random**?

Favors temporal locality, as local variables usually are reused frequently.

iii) What is one benefit of using **Random** over **LRU**?

Cheaper and faster to use as there is no need to maintain record of the most recently used block.

h) If we were to flush the cache right now (don't actually) how many bytes in memory would change? 3

How many bytes would change if we were using **Write Through** instead of **Write Back**? 0

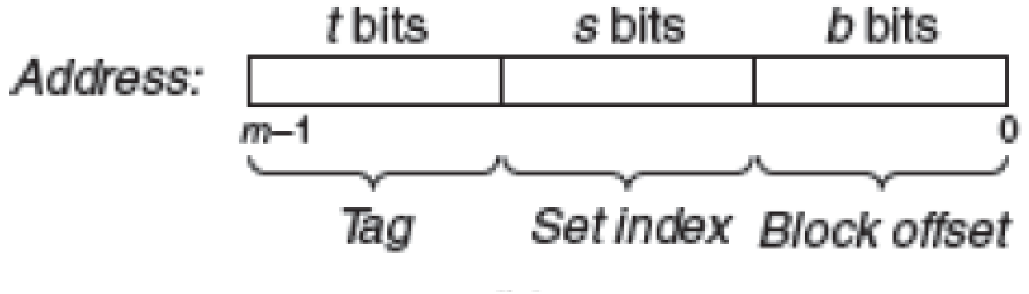
Can you explain why these numbers are the same/different? (if not, try changing the write hit policy and re-running using the history above).

Write Back won't write the new value to memory directly but will instead cache it and mark its block as dirty. When any dirty block is removed from the cache the memory corresponding to that block will be updated.

Write Through will write any new value to memory directly, thus meaning that no block in the cache will be dirty and no values in memory need to be updated when flushing the cache.

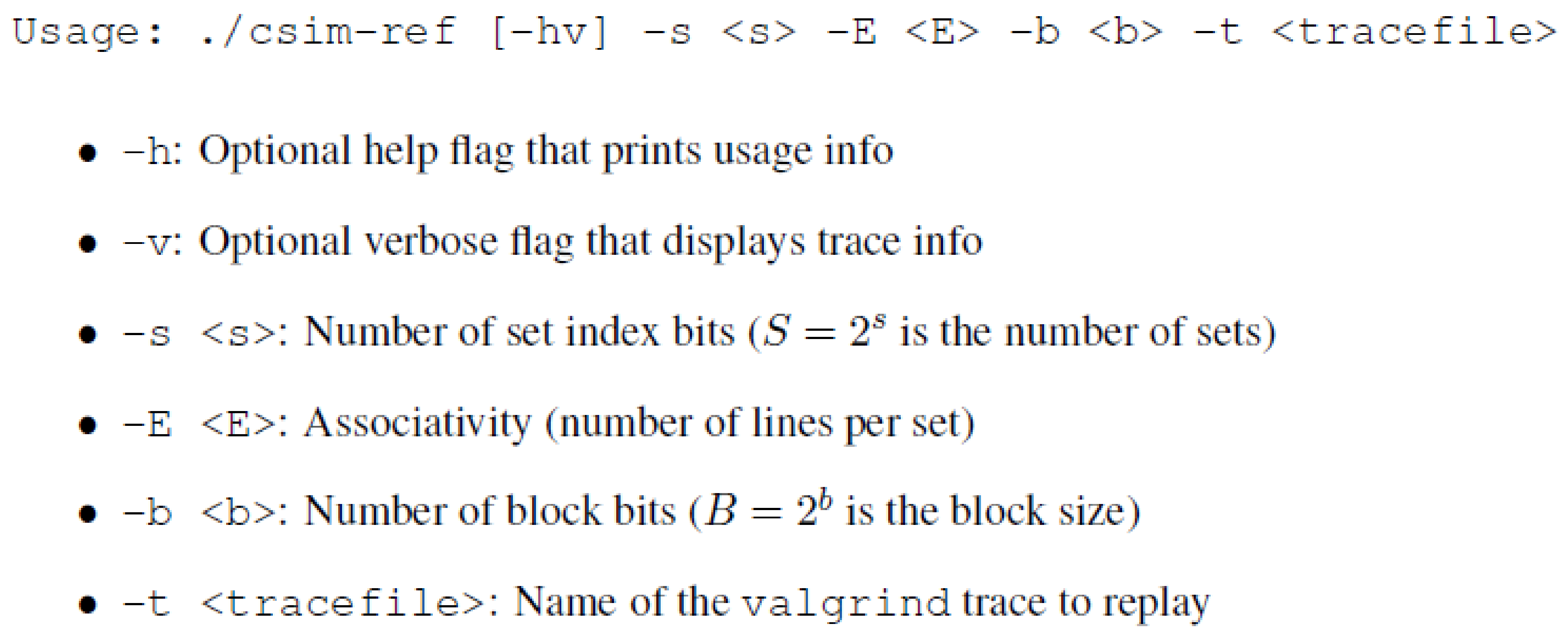
Cache Lab

Cache Lab



- A cache simulator is NOT a cache!
  - Main Memory contents are NOT stored
  - Block offsets are NOT used – don’t worry about the ***b*** bits in your address.
  - Simply count hits, misses, and evictions

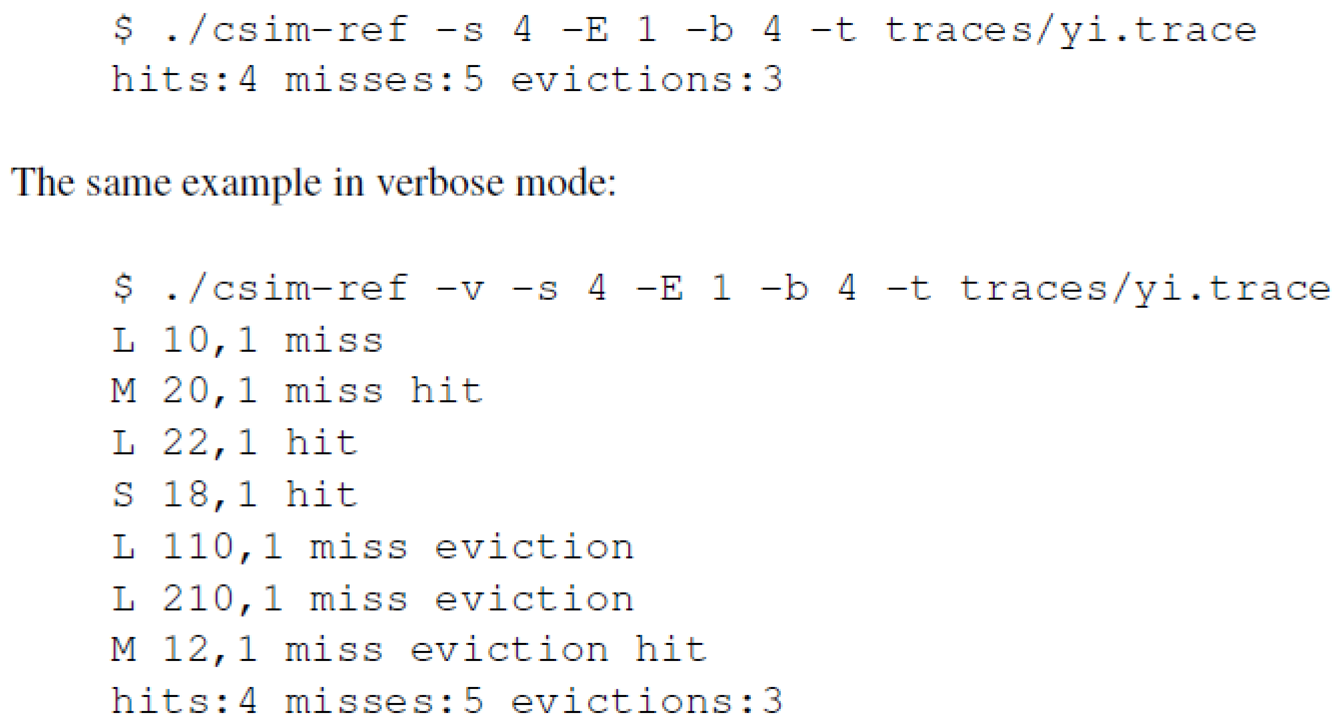
Step 1: Understanding the Command Line Arguments



```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

Step 1: Understanding the Command Line Arguments



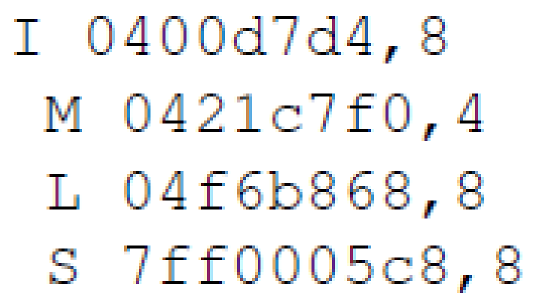
```
$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

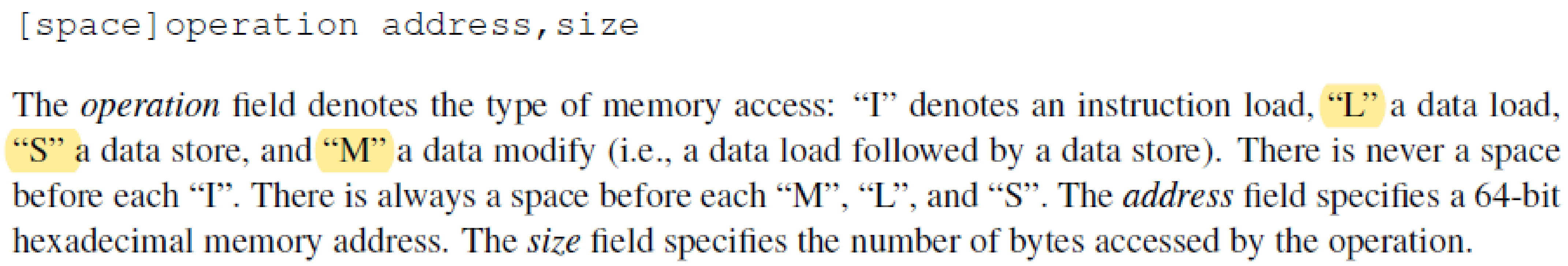
```
$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Cross-check your results with the results obtained by the reference simulator *csim-ref*

Step 2 : Understanding the Trace files



```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```



```
[space]operation address,size
```

The *operation* field denotes the type of memory access: “I” denotes an instruction load, “L” a data load, “S” a data store, and “M” a data modify (i.e., a data load followed by a data store). There is never a space before each “I”. There is always a space before each “M”, “L”, and “S”. The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

Step 2 : Understanding the Trace files

- Instruction loads (I) are **ignored**, since we are interested in evaluating data cache performance.

- Each load/store means **1 memory access.**
    - Can cause at most one cache miss.

- Data modify (M) is treated as a load followed by a store to the same address. **2 Memory Accesses.**
    - Hence, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.

- Assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can **ignore the request sizes** in the valgrind traces.

Example

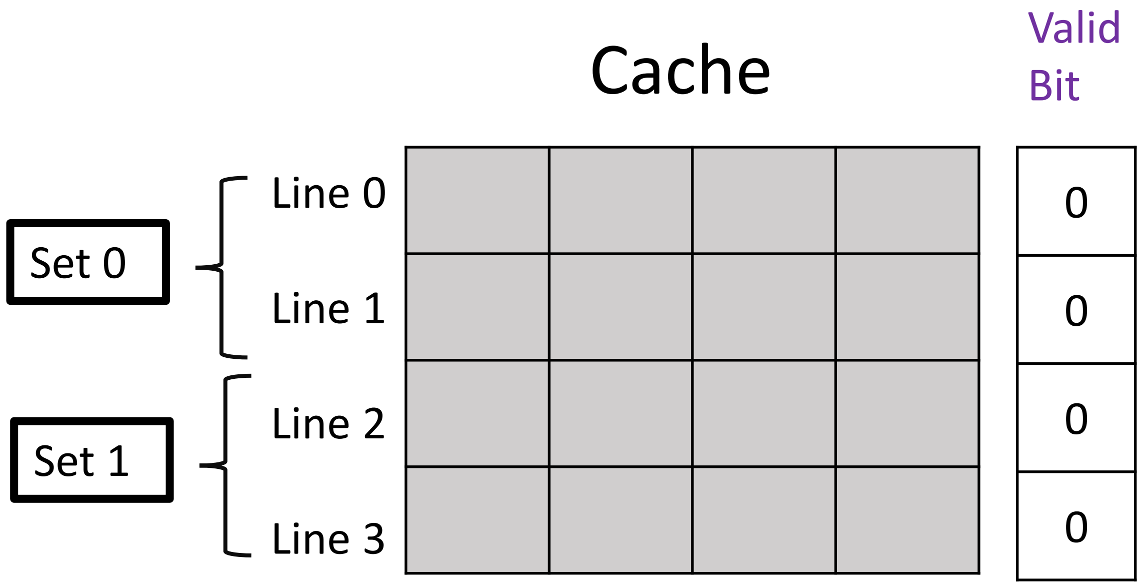
- **Command Line Argument**

```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

Number of bits for set Index = $s = 1$
Number of sets (S)= $2^s = 2^1 = 2$

Number of lines per set = $E = 2$
Total Number of Lines = $S * 2 = 2 * 2 = 4$

Number of bits for byte offset = $b = 2$
Block Size/Line Size = $2^b = 2^2 = 4$



Example

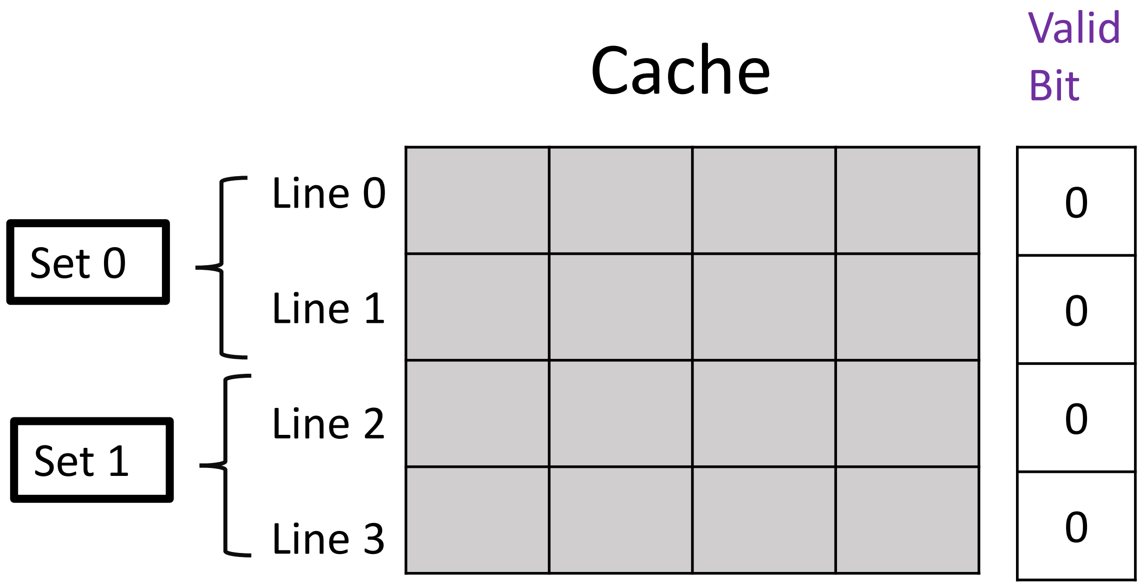
- **Command Line Argument**

```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```

Ignore I operations



Example

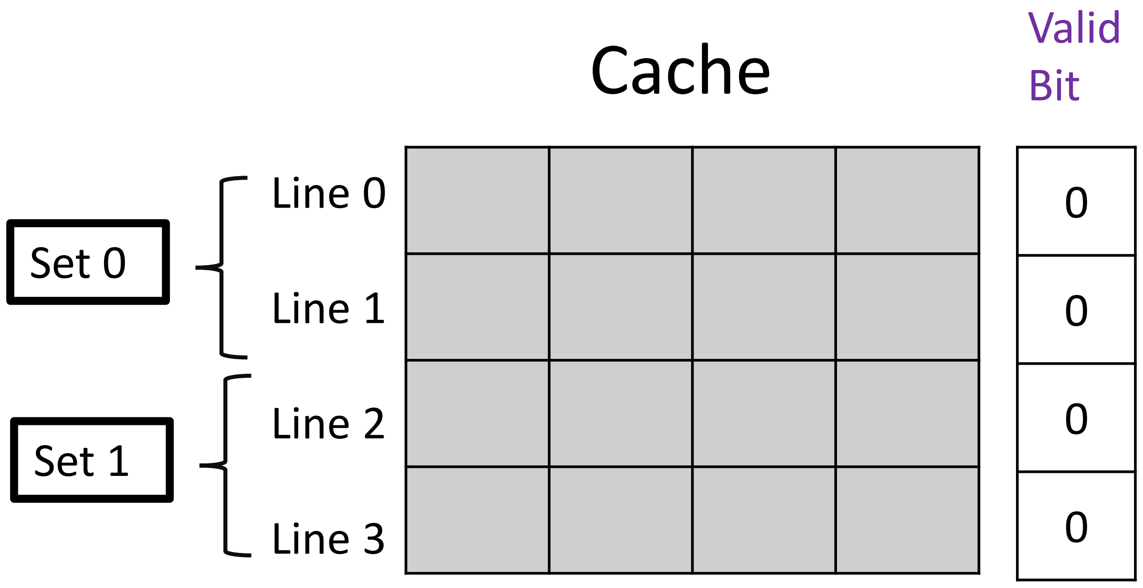
Cache MISS!

- **Command Line Argument**

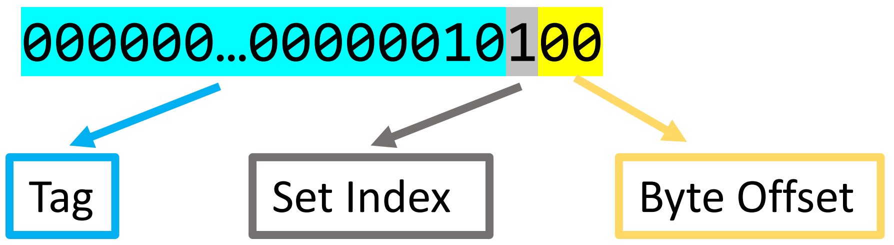
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



64-bit Address = 20 = 000000…00000010100



Example

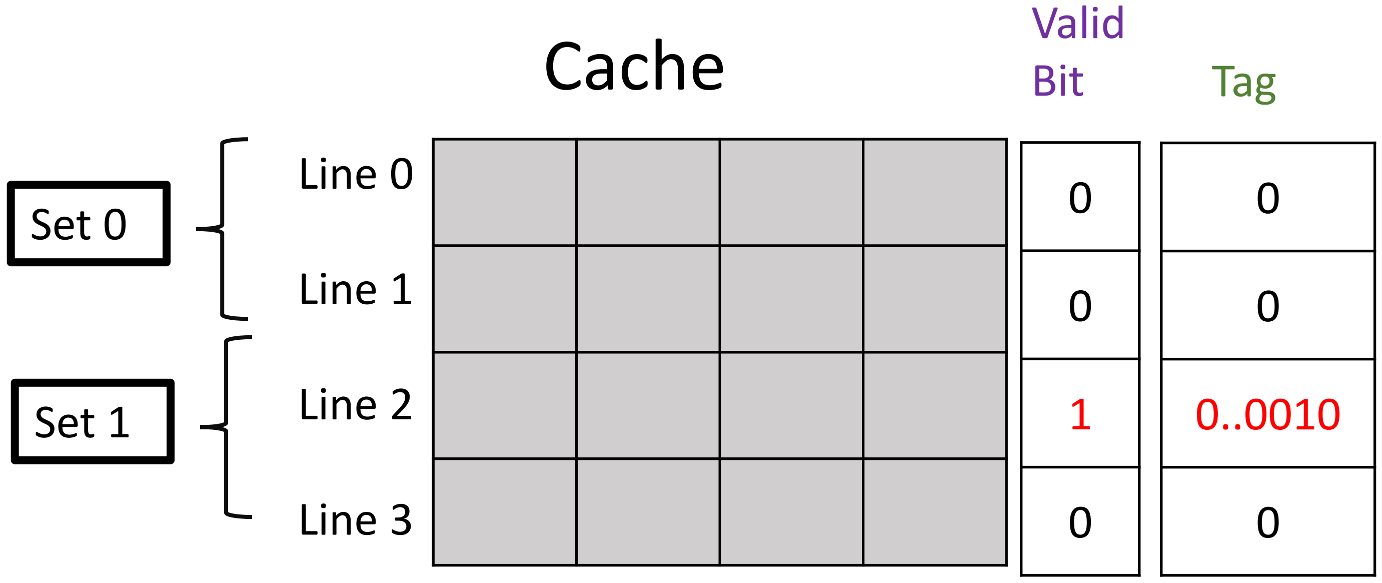
Update the Cache

- **Command Line Argument**

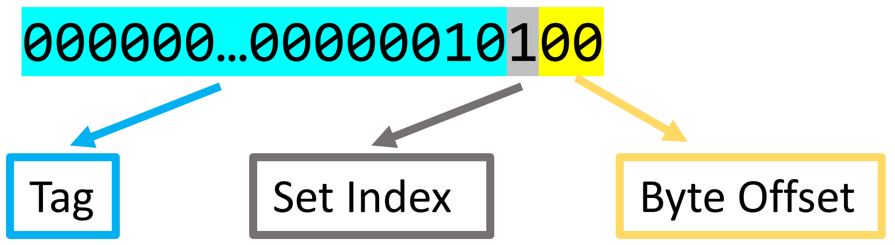
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 20 = 000000…000000010100
```



Example

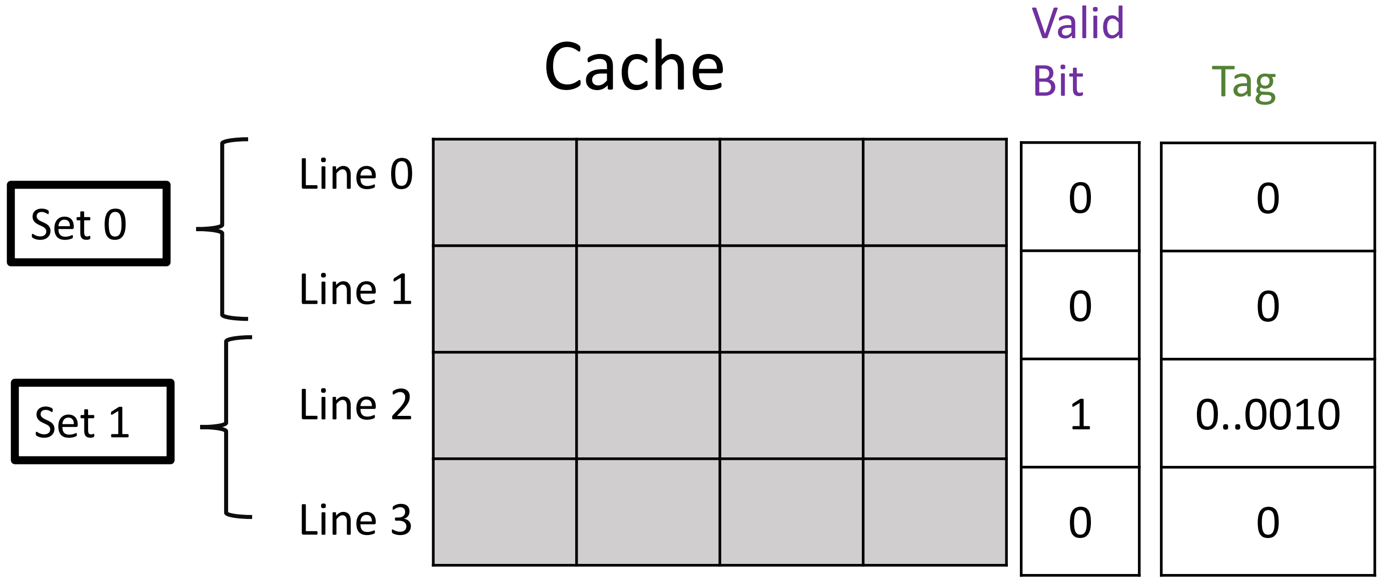
- Operation M means 2 memory accesses
  - 1st access: Cache MISS!

- **Command Line Argument**

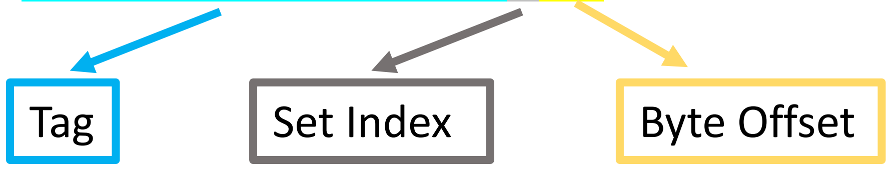
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 25 = 000000…00000011001
```



Example

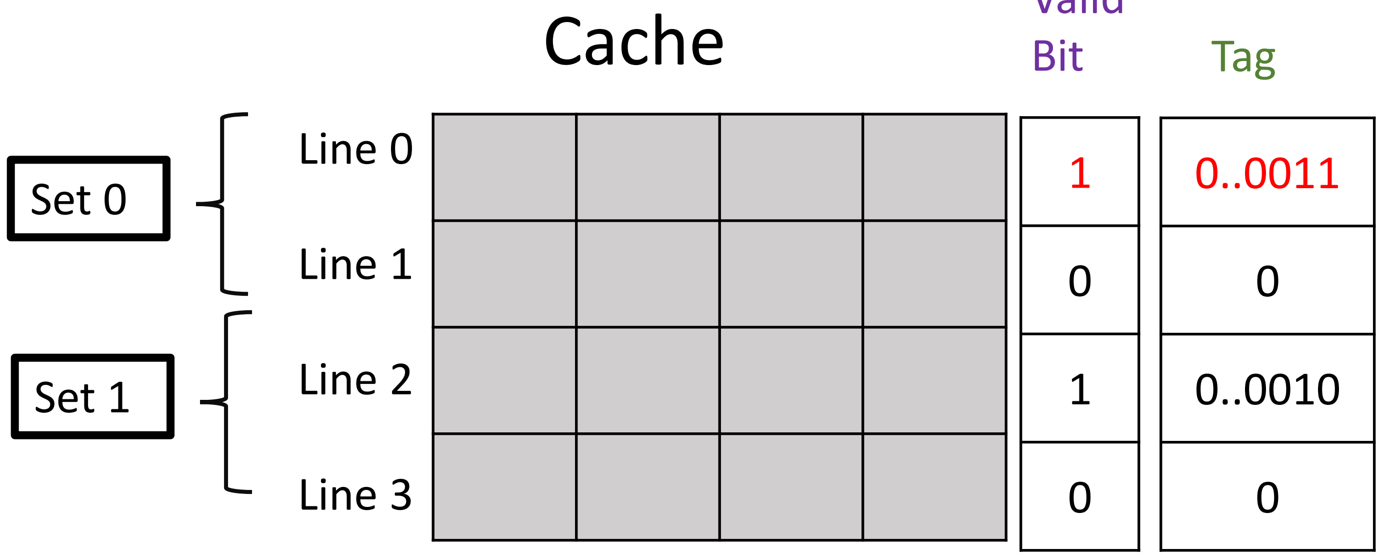
- Operation M means 2 memory accesses
  - 1st access: Cache MISS!
    - Update the Cache
  - 2nd access: Cache HIT!

- **Command Line Argument**

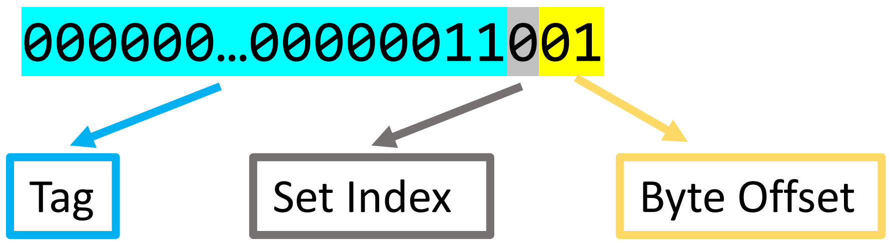
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 25 = 000000…00000011001
```



Example

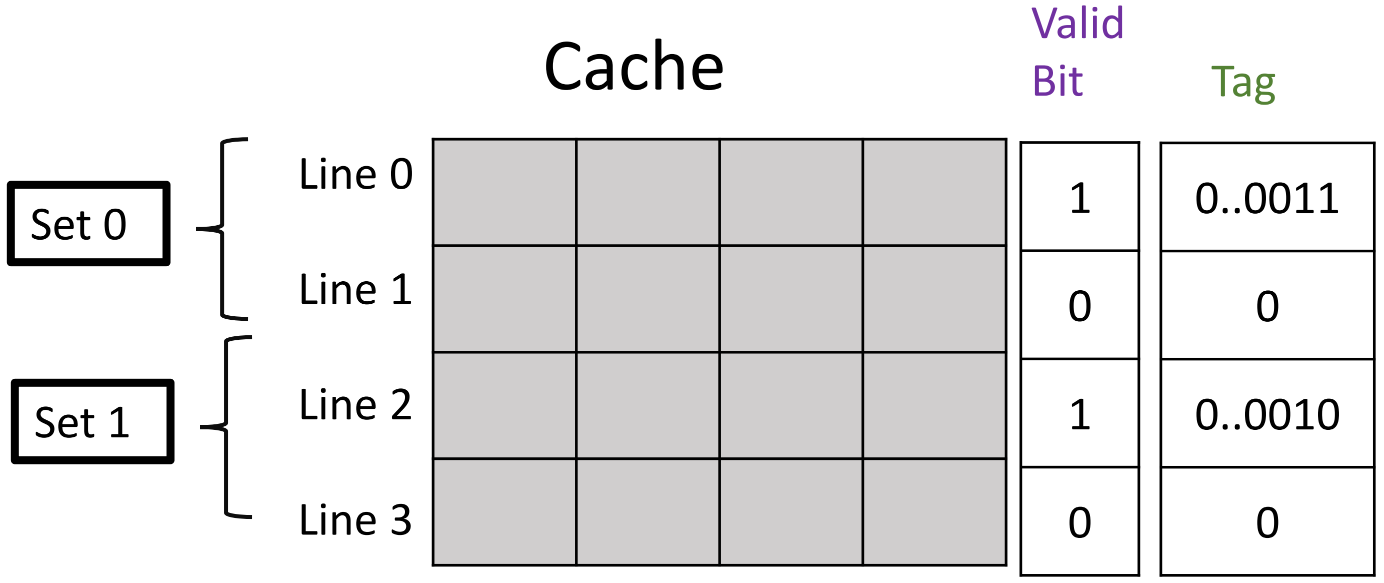
Cache MISS!

- **Command Line Argument**

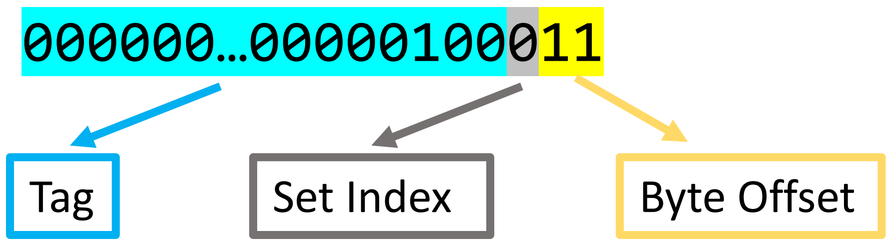
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 35 = 000000…000001000 11
```



Example

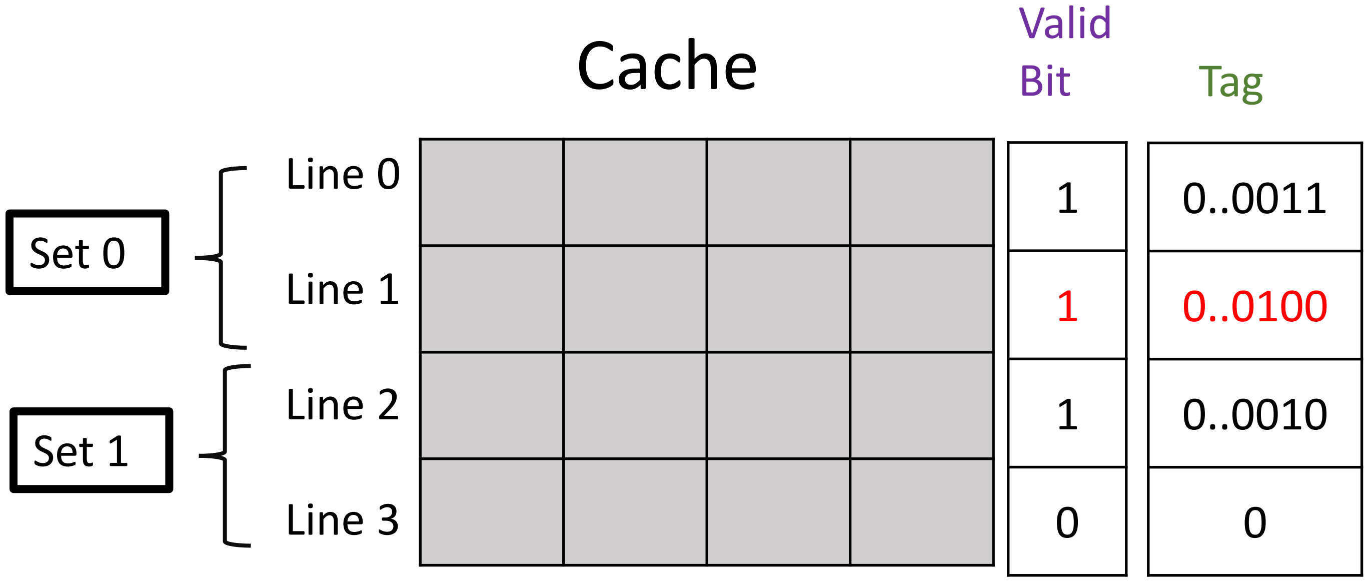
Update the cache

- **Command Line Argument**

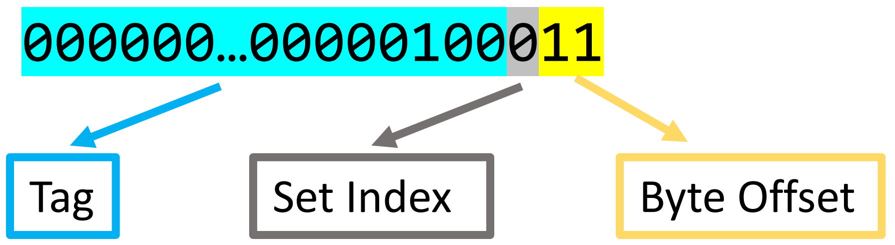
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 35 = 000000…000001000 11
```



Example

Cache MISS!

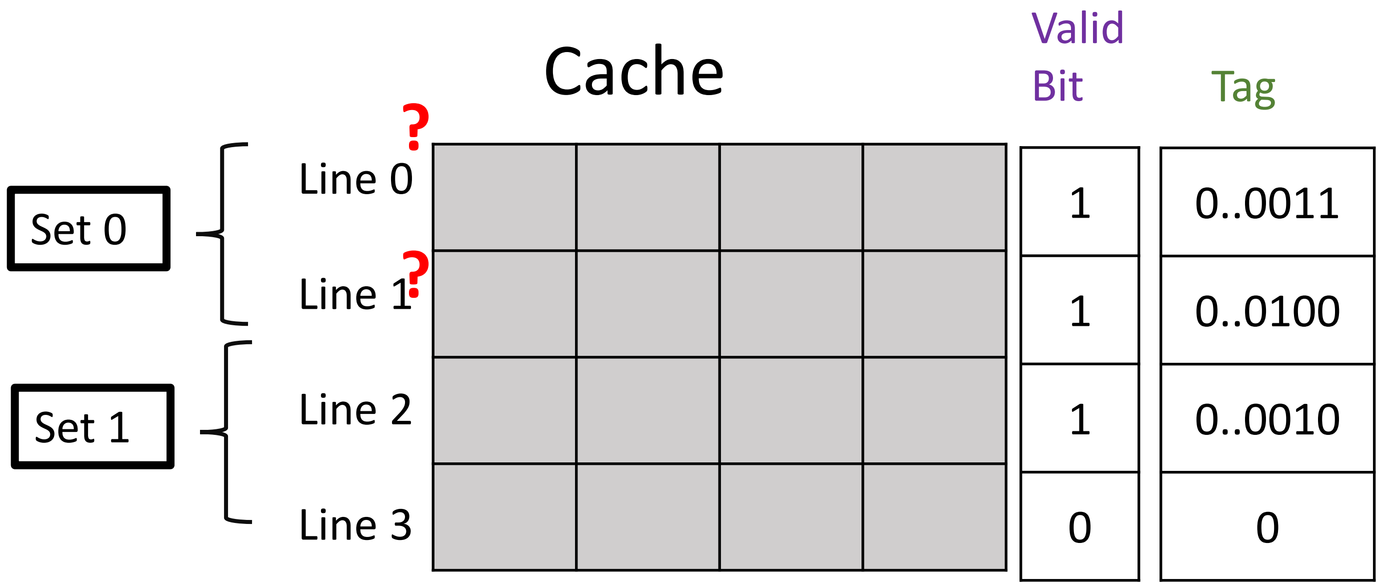
Which cache line to be replaced/evicted??

- **Command Line Argument**

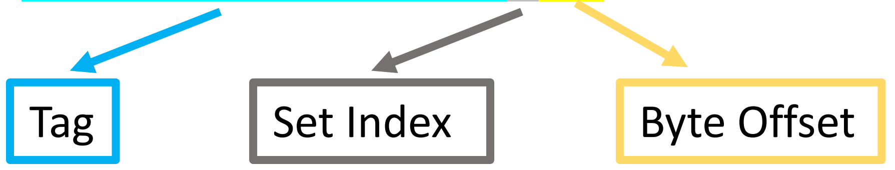
```
$./csim -s 1 -E 2 -b 2 -t traces/tr1.trace
```

tr1.trace

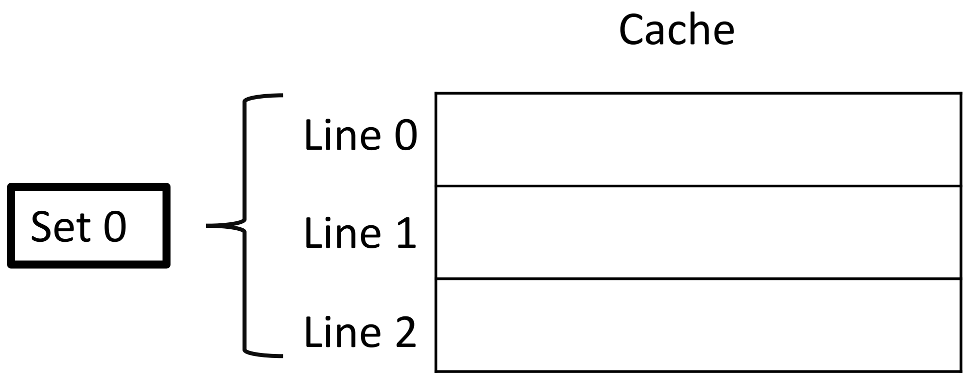
```
I 10, 1
 L 20, 1
 M 25, 1
 S 35, 1
 L 50, 1
```



```
64-bit Address = 50 = 000000…000000110010
```



Least Recently Used (LRU) Eviction

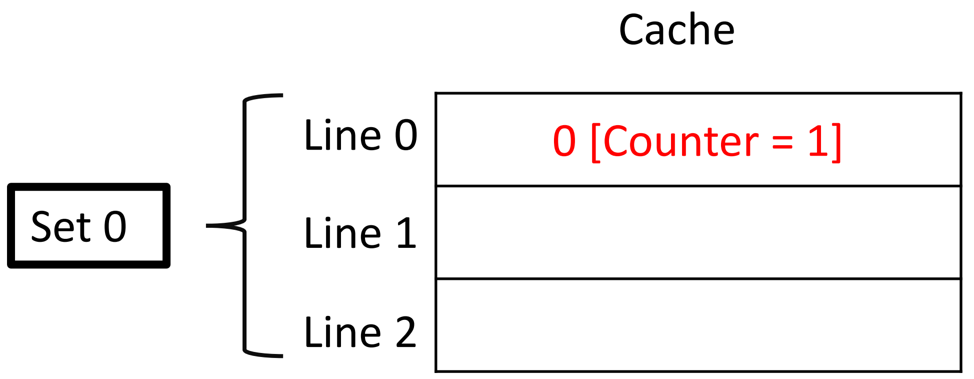


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 0

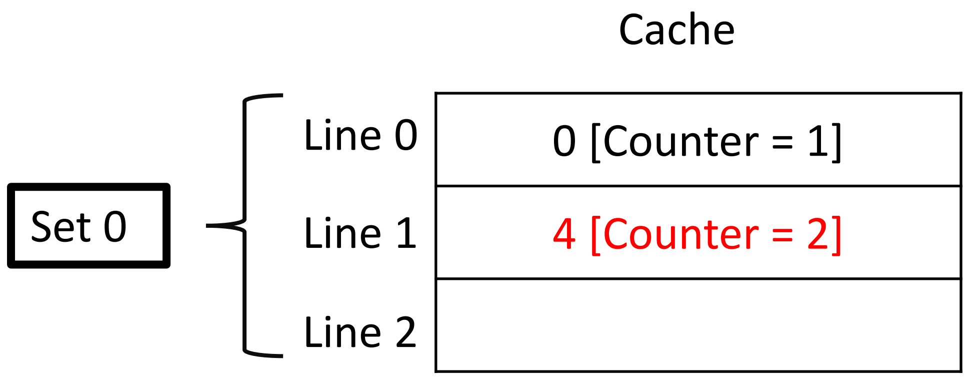
Least Recently Used (LRU) Eviction



Tag Requests
0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 1

Least Recently Used (LRU) Eviction

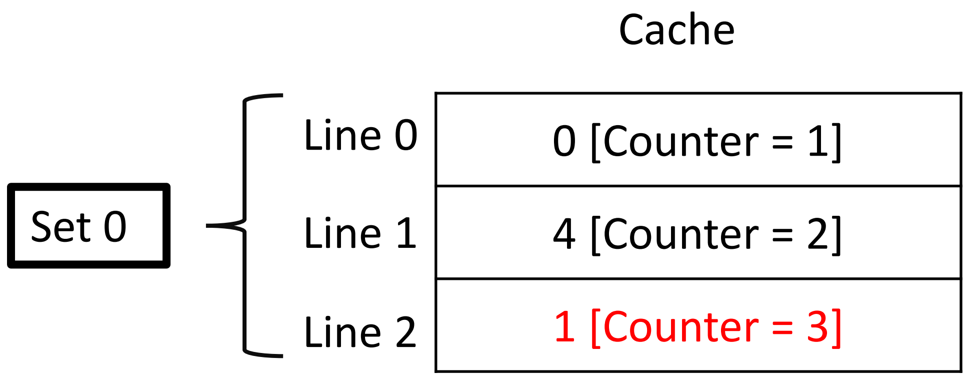


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 2

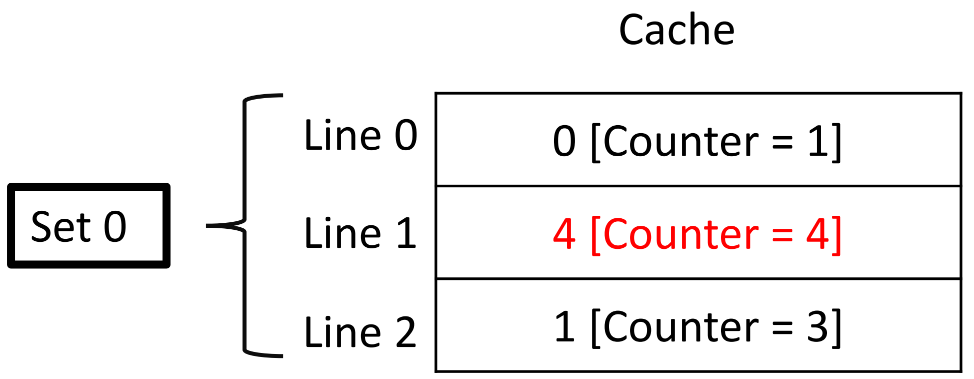
Least Recently Used (LRU) Eviction

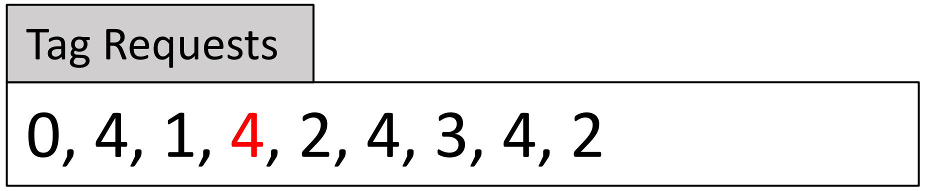


Tag Requests
0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 3

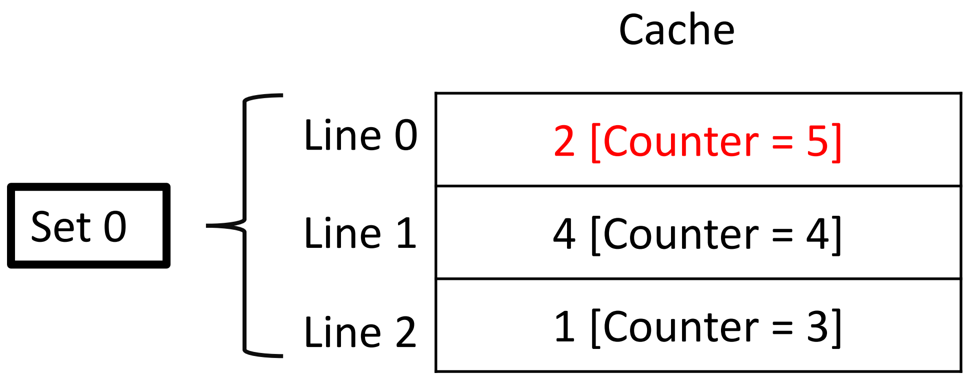
Least Recently Used (LRU) Eviction





Global Counter = 4

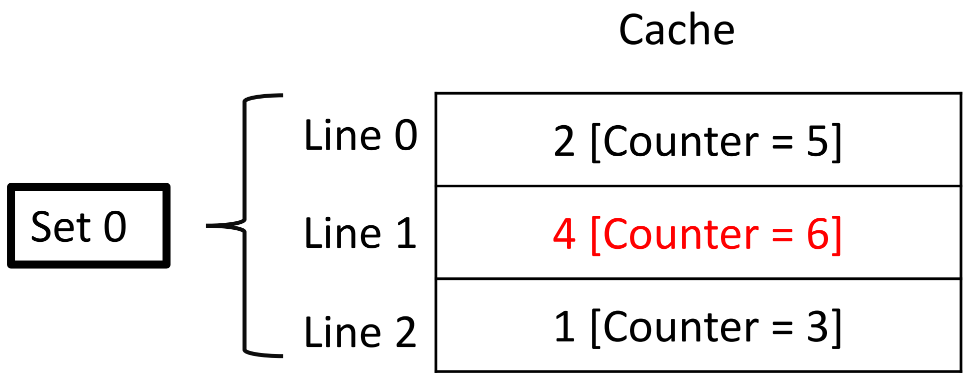
Least Recently Used (LRU) Eviction



Tag Requests
0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 5

Least Recently Used (LRU) Eviction

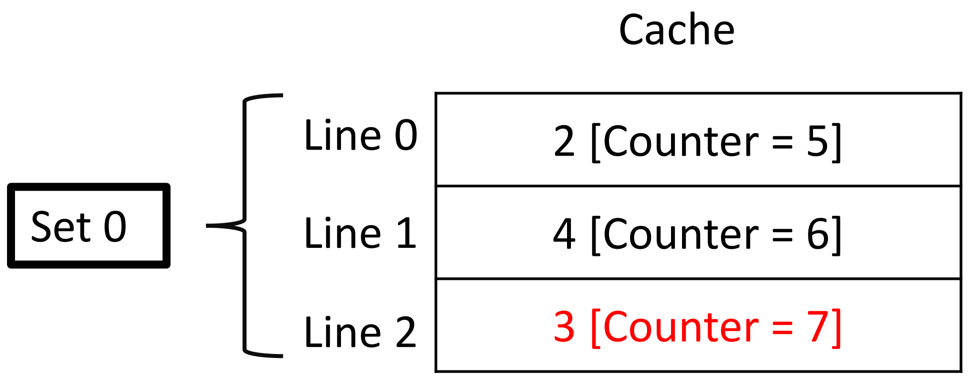


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 6

Least Recently Used (LRU) Eviction

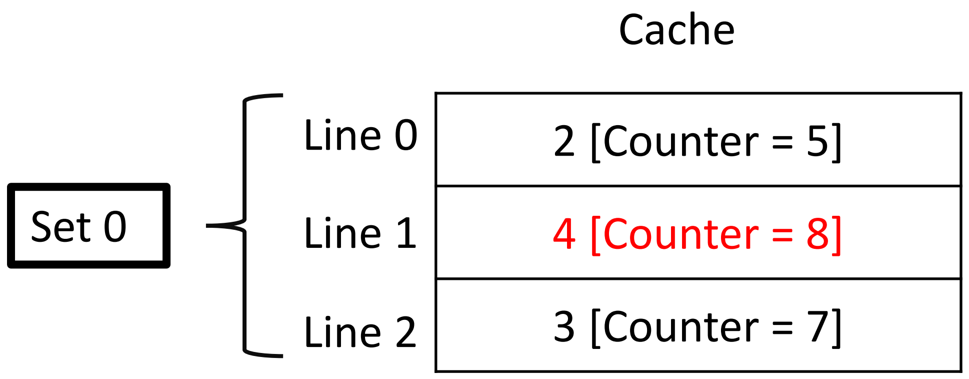


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 7

Least Recently Used (LRU) Eviction

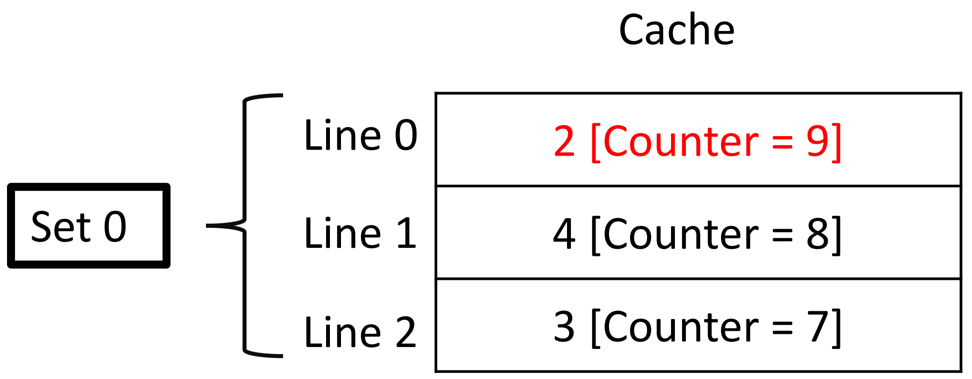


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 8

Least Recently Used (LRU) Eviction

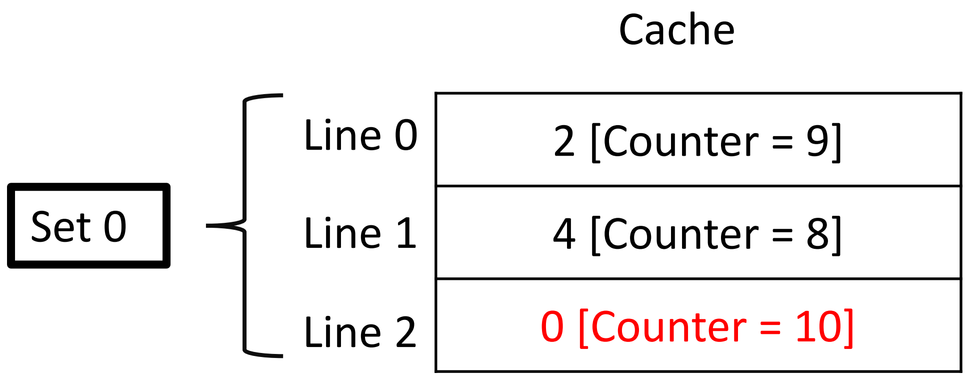


Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2

Global Counter = 9

Least Recently Used (LRU) Eviction



Tag Requests

0, 4, 1, 4, 2, 4, 3, 4, 2, 0

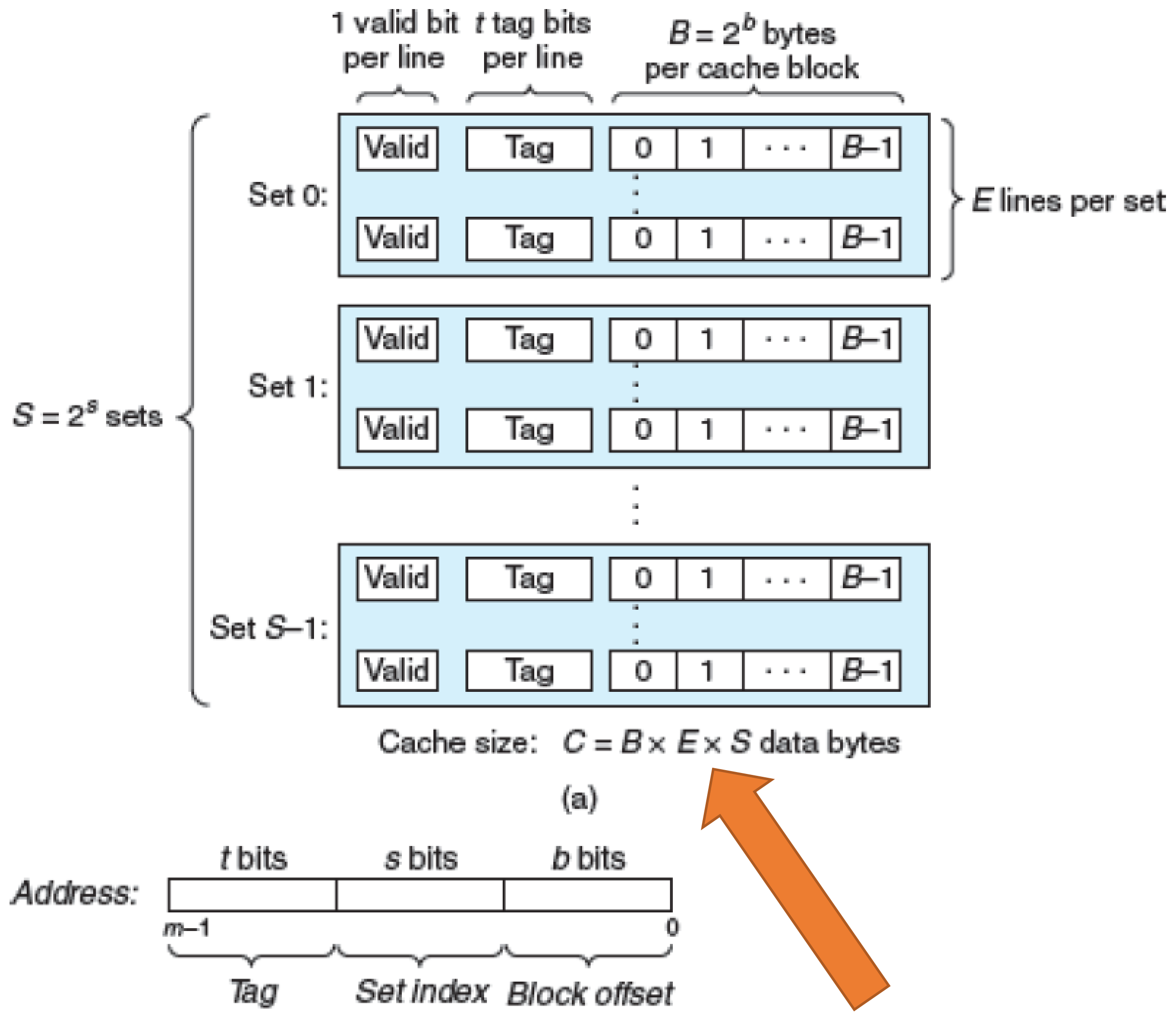
Global Counter = 10

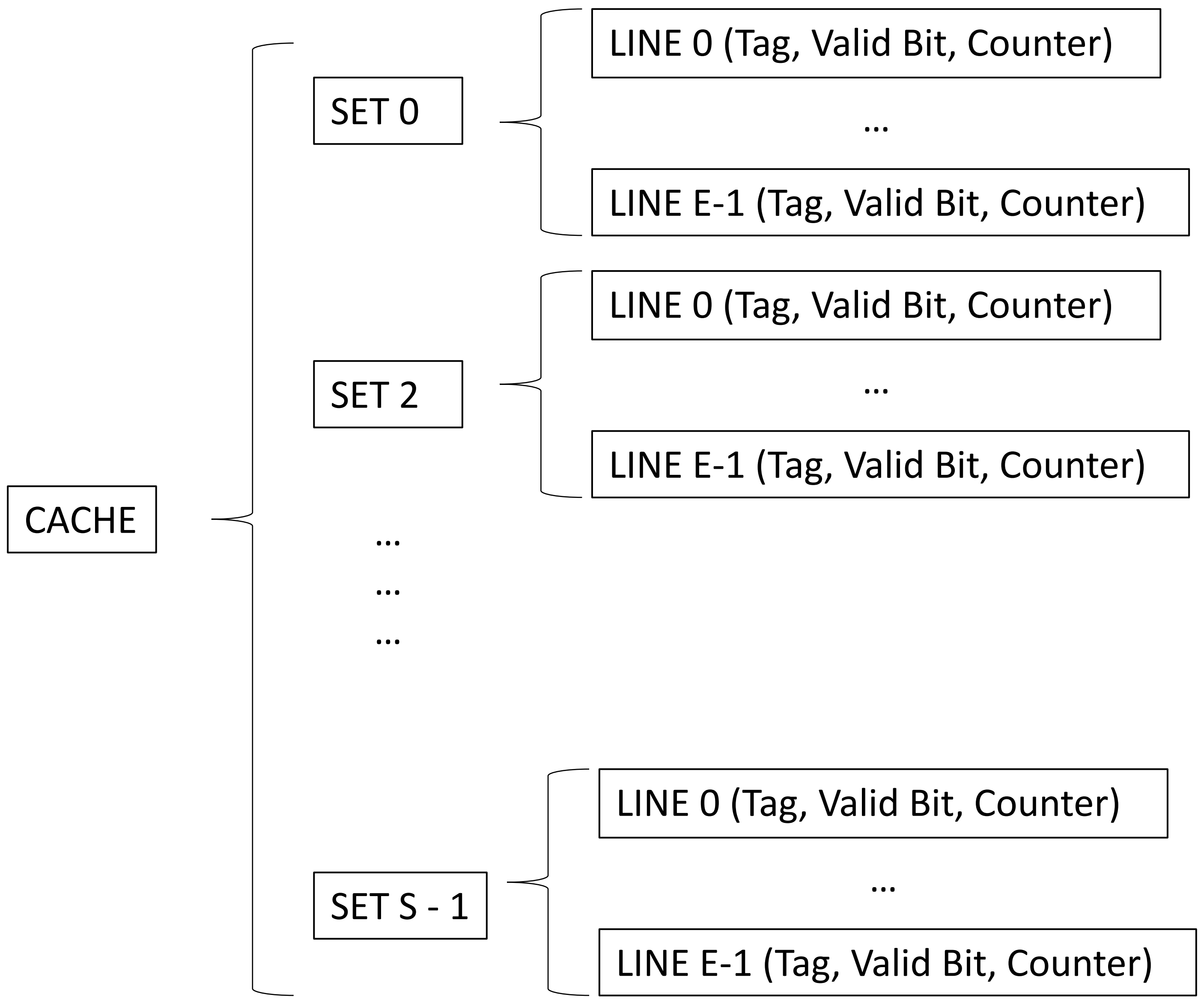
Step 3: Build Your Cache

- Design your cache
  - Define your own data structures
  - Types: cache, cache line, cache set
  - You can use a counter to implement LRU replacement policy

Step 3: Build Your Cache

- How **big is your Cache**?
  - Depends on user inputs of *s*, *E* and *b*.
- So, use malloc() for allocating space for
  - Each of the sets
  - Lines per set
  - Initialize valid bit, tag and LRU counter
- Deallocate (with free) the cache data structures of each set and line.




CACHE
SET 0
LINE 0 (Tag, Valid Bit, Counter)
…
LINE E-1 (Tag, Valid Bit, Counter)
SET 2
LINE 0 (Tag, Valid Bit, Counter)
…
LINE E-1 (Tag, Valid Bit, Counter)
…
…
…
SET S - 1
LINE 0 (Tag, Valid Bit, Counter)
…
LINE E-1 (Tag, Valid Bit, Counter)

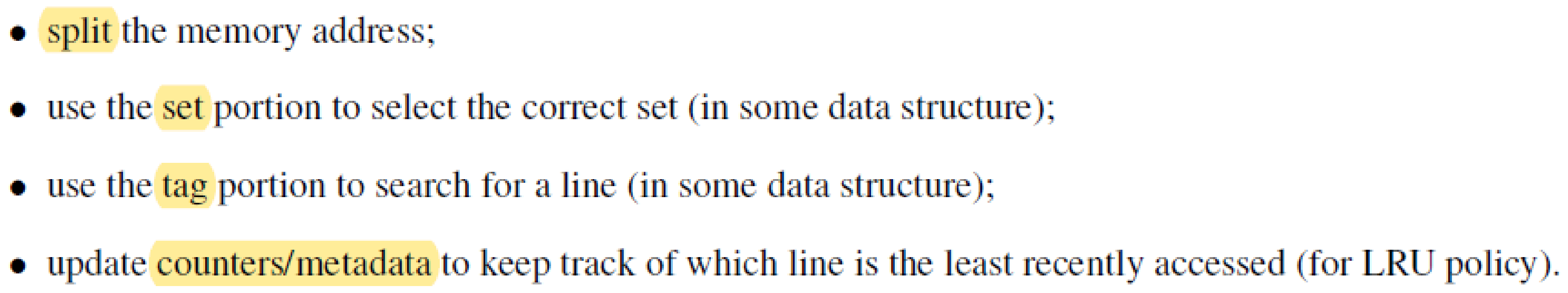
Example of a 2D array of integers

```
//ALLOCATE
int** matrix = malloc(10 * sizeof(int*));
for (int i=0; i<10; ++i)
{
        matrix[i] = malloc(10 * sizeof(int));
}

//DEALLOCATE
for (int i=0; i<10; ++i)
{
  free(matrix[i]);
}

free(matrix);
```

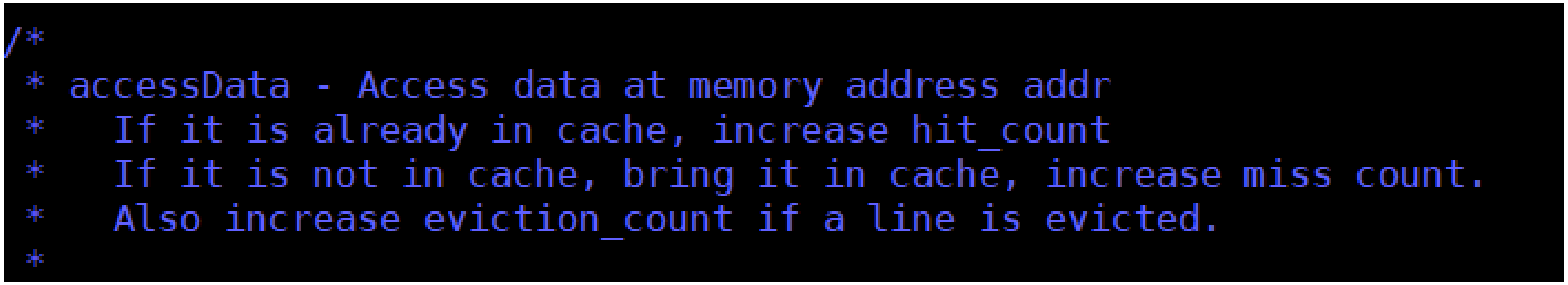
Step 4: Parsing

- Use *getopt* to parse command-line arguments.
- Open/close file *trace_fn* for reading (using fopen/fclose)
- Read lines (using *fgets, fscanf,* or *getline*) from the file handle (may name `trace_fp` variable)
- Use sscanf to parse fields within a string
  - Skip lines not starting with ` S`, ` L` or ` M`
  - Parse the memory address (unsigned long, in hex) and len (unsigned int, in decimal) from each input line
- **Most of Parsing has already been done for you. Check the code and comments in csim.c.**

Step 5: Access Data



- split the memory address;
- use the set portion to select the correct set (in some data structure);
- use the tag portion to search for a line (in some data structure);
- update counters/metadata to keep track of which line is the least recently accessed (for LRU policy).



```
/*
 * accessData - Access data at memory address addr
 *   If it is already in cache, increase hit_count
 *   If it is not in cache, bring it in cache, increase miss count.
 *   Also increase eviction_count if a line is evicted.
 *
```

References

- https://www.youtube.com/watch?v=3eriC-pIQKg
- https://www.youtube.com/watch?v=mCF5XNn_xfA
- https://www.youtube.com/watch?v=j5PUJllPPVE
- https://en.wikipedia.org/wiki/Cache_placement_policies
- https://www.cs.cmu.edu/afs/cs/academic/class/15213-s15/www/recitations/recitation-7.pdf
- Locality
- http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/9-virtual-mem/LRU-replace.html