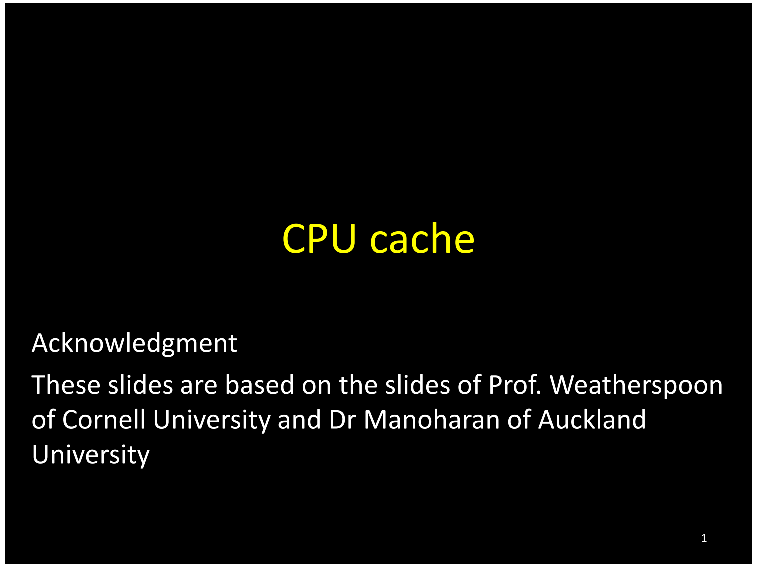
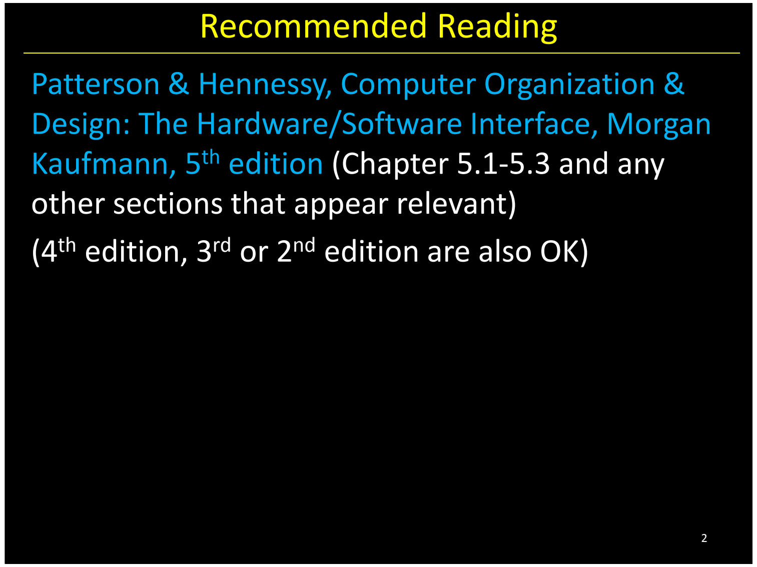


CPU cache

Acknowledgment

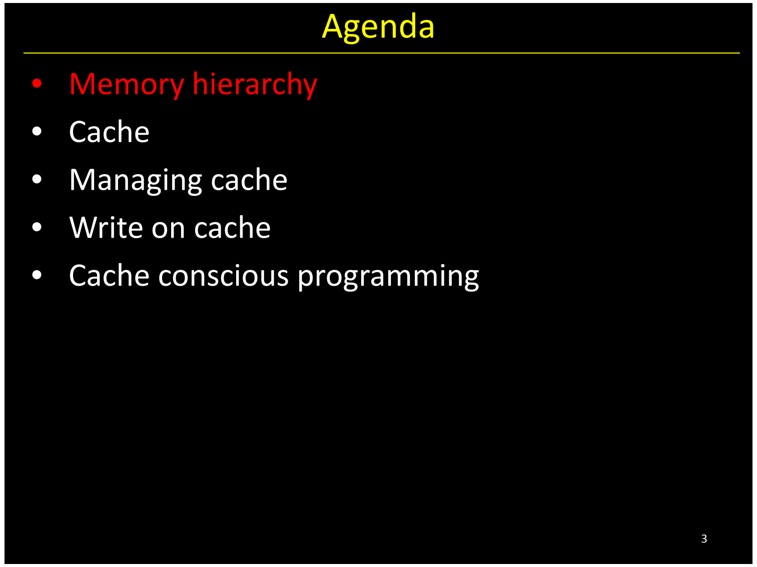
These slides are based on the slides of Prof. Weatherspoon of Cornell University and Dr Manoharan of Auckland University



Recommended Reading

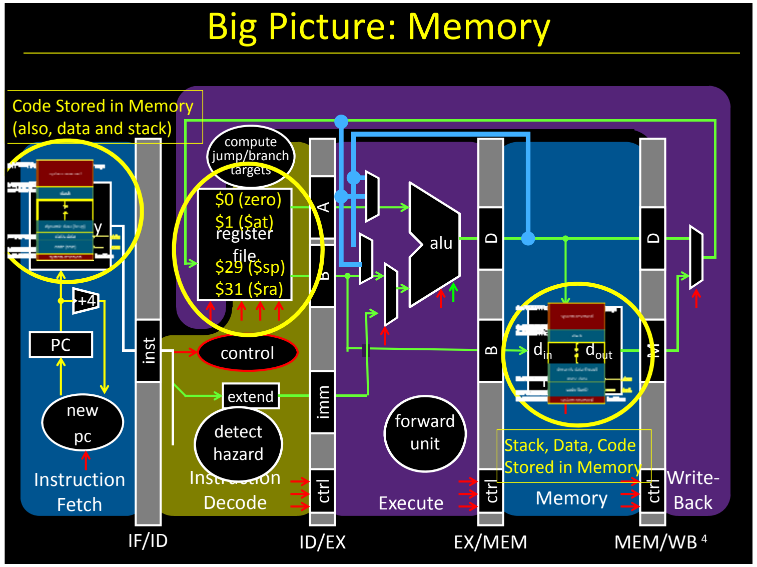
Patterson & Hennessy, Computer Organization & Design: The Hardware/Software Interface, Morgan Kaufmann, 5th edition (Chapter 5.1-5.3 and any other sections that appear relevant)

(4th edition, 3rd or 2nd edition are also OK)



Agenda

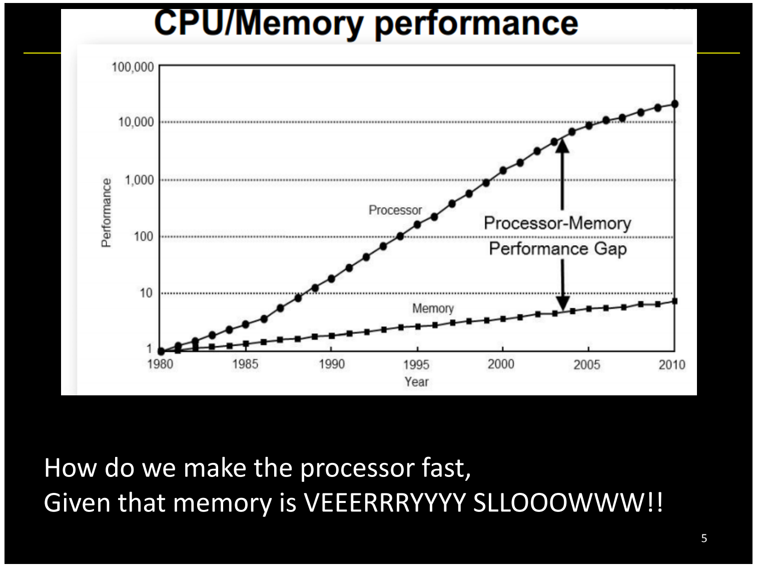
- Memory hierarchy
- Cache
- Managing cache
- Write on cache
- Cache conscious programming



Big Picture: Memory

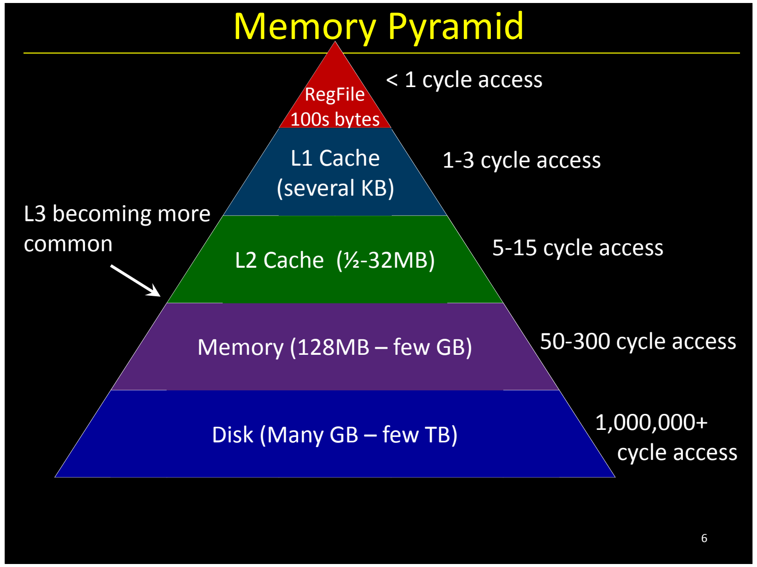
Code Stored in Memory (also, data and stack)

Stack, Data, Code Stored in Memory



CPU/Memory performance

How do we make the processor fast,
Given that memory is VEEERRRYYYY SLLOOOWWW!!



Memory Pyramid

RegFile
100s bytes
< 1 cycle access

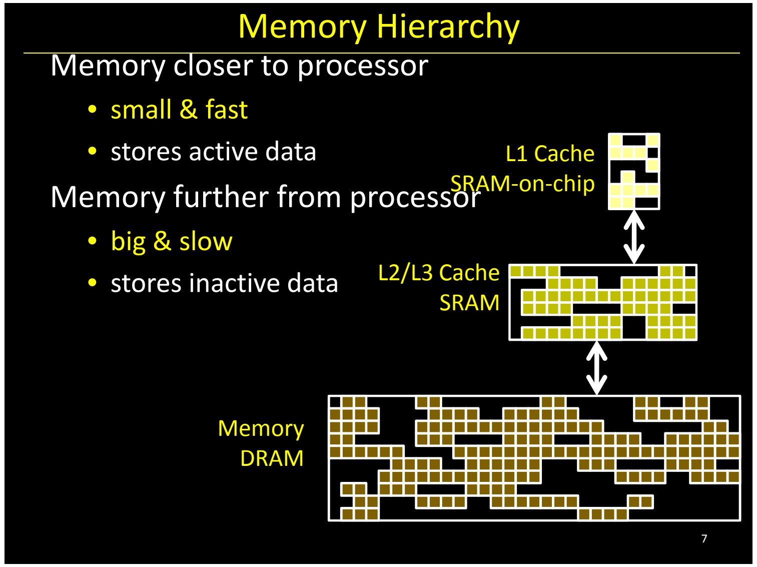
L1 Cache
(several KB)
1-3 cycle access

L3 becoming more
common

L2 Cache (½-32MB)
5-15 cycle access

Memory (128MB – few GB)
50-300 cycle access

Disk (Many GB – few TB)
1,000,000+
cycle access



Memory Hierarchy

Memory closer to processor

- small & fast
- stores active data

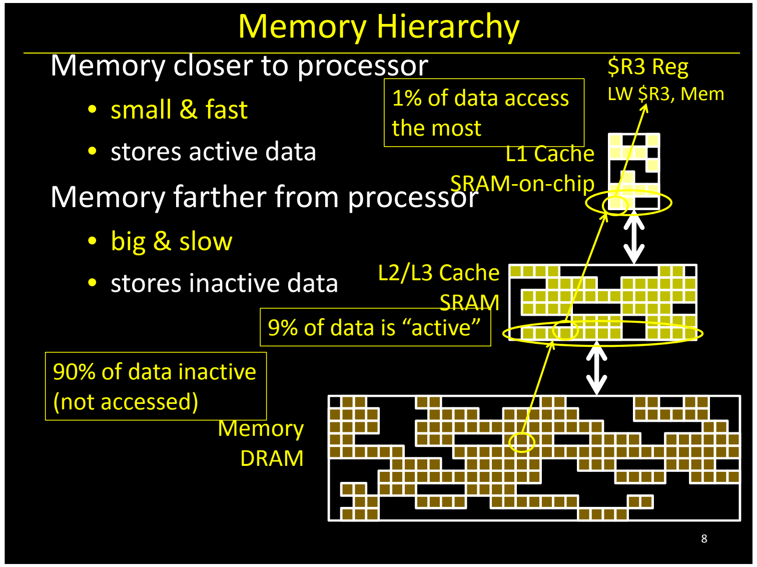
Memory further from processor

- big & slow
- stores inactive data

L1 Cache
SRAM-on-chip

L2/L3 Cache
SRAM

Memory
DRAM



Memory Hierarchy

Memory closer to processor

- small & fast
- stores active data

Memory farther from processor

- big & slow
- stores inactive data

$R3 Reg
LW $R3, Mem

1% of data access
the most

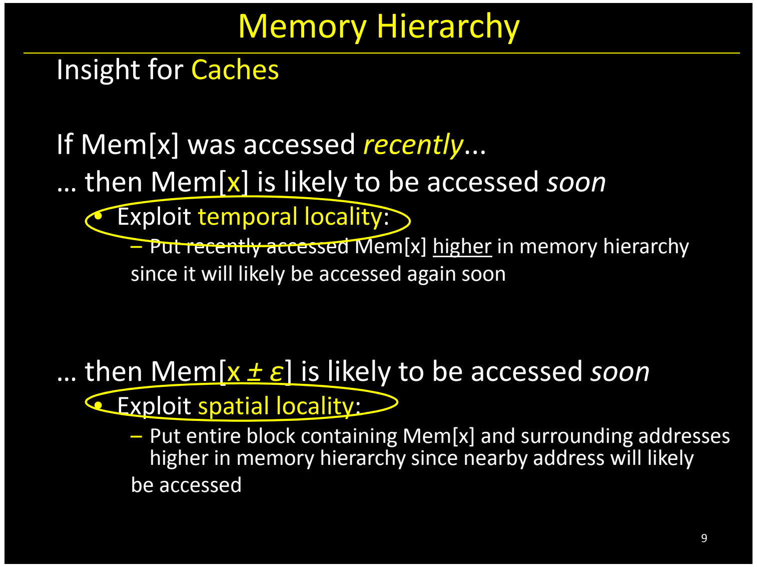
L1 Cache
SRAM-on-chip

L2/L3 Cache
SRAM

9% of data is "active"

90% of data inactive
(not accessed)

Memory
DRAM



Memory Hierarchy

Insight for Caches

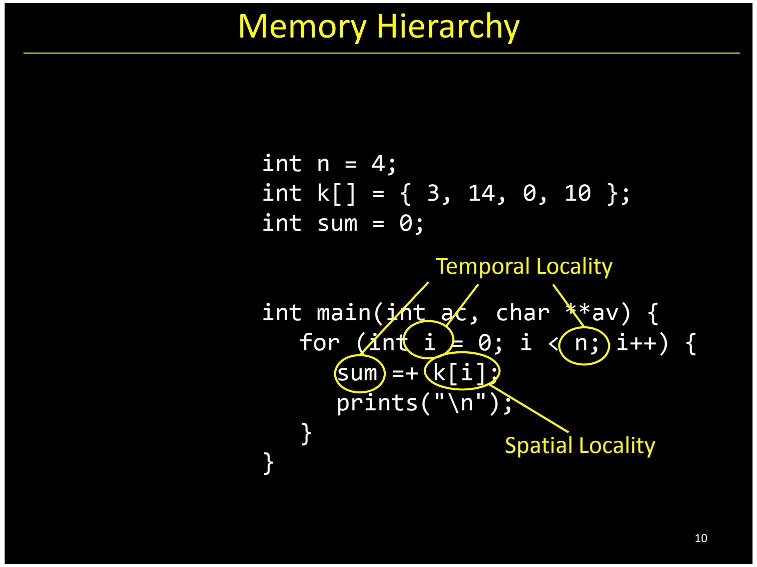
If Mem[x] was accessed *recently*...

... then Mem[x] is likely to be accessed *soon*

- Exploit temporal locality:
  - Put recently accessed Mem[x] higher in memory hierarchy since it will likely be accessed again soon

... then Mem[x ± ε] is likely to be accessed *soon*

- Exploit spatial locality:
  - Put entire block containing Mem[x] and surrounding addresses higher in memory hierarchy since nearby address will likely be accessed



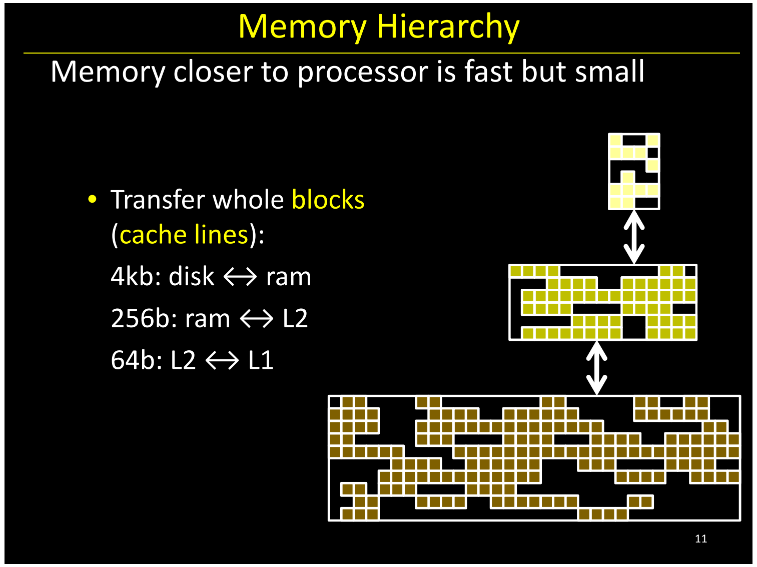
Memory Hierarchy

```
int n = 4;
int k[] = { 3, 14, 0, 10 };
int sum = 0;

int main(int ac, char **av) {
   for (int i = 0; i < n; i++) {
      sum =+ k[i];
      prints("\n");
   }
}
```

Temporal Locality

Spatial Locality



Memory Hierarchy

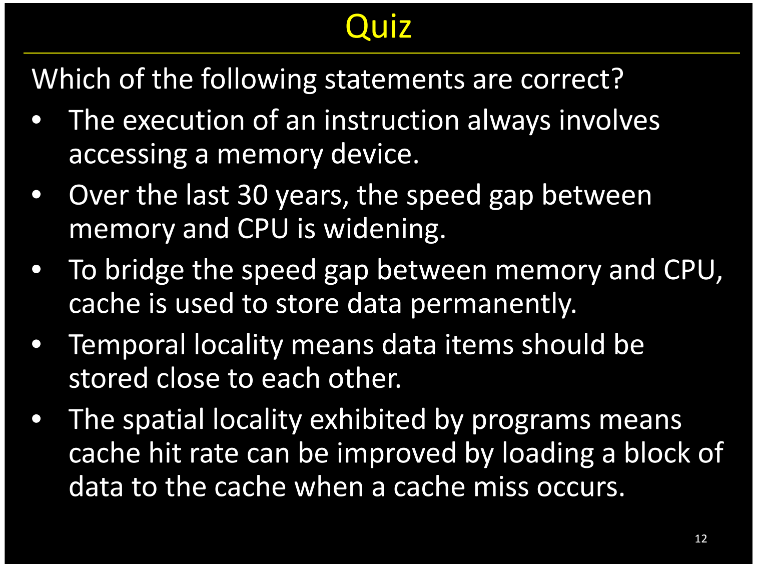
Memory closer to processor is fast but small

- Transfer whole blocks (cache lines):

  4kb: disk ↔ ram

  256b: ram ↔ L2

  64b: L2 ↔ L1



Quiz

Which of the following statements are correct?

- The execution of an instruction always involves accessing a memory device.
- Over the last 30 years, the speed gap between memory and CPU is widening.
- To bridge the speed gap between memory and CPU, cache is used to store data permanently.
- Temporal locality means data items should be stored close to each other.
- The spatial locality exhibited by programs means cache hit rate can be improved by loading a block of data to the cache when a cache miss occurs.

Agenda

- Memory hierarchy
- Cache
- Managing cache
- Write on cache
- Cache conscious programming

Cache Lookups

Processor tries to access Mem[x]

Check: is block containing Mem[x] in the cache?

- Yes: cache hit
  - return requested data from cache line
- No: cache miss
  - read block from memory (or lower level cache)
  - (evict an existing cache line to make room)
  - place new block in cache
  - return requested data

Cache look up example

- Assume a cache has 4 lines (slots) and each line can hold one item (byte).
  - There are 4 slots where we can keep numbered (i.e. addressable) items.
- We want to keep in the 4 slots some items that have been used in the recent past.
- What process should we follow to make our cache work?

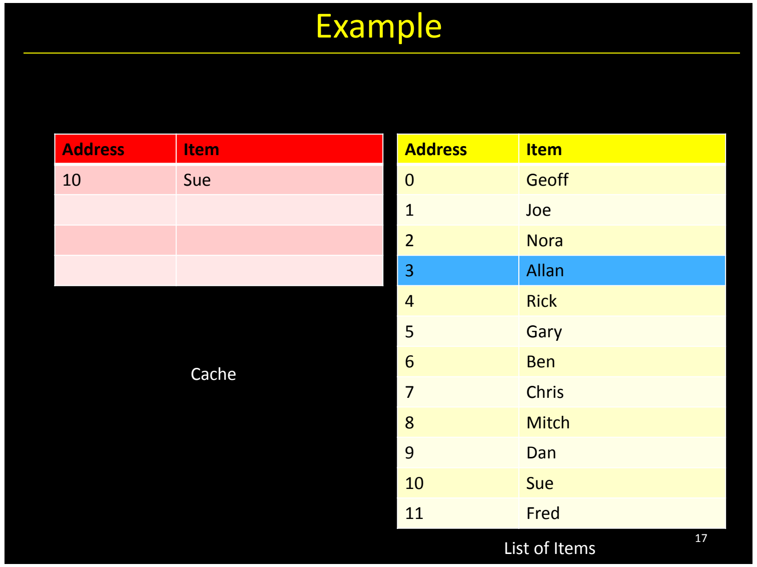
Example

address	Item

Cache

address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items



Example

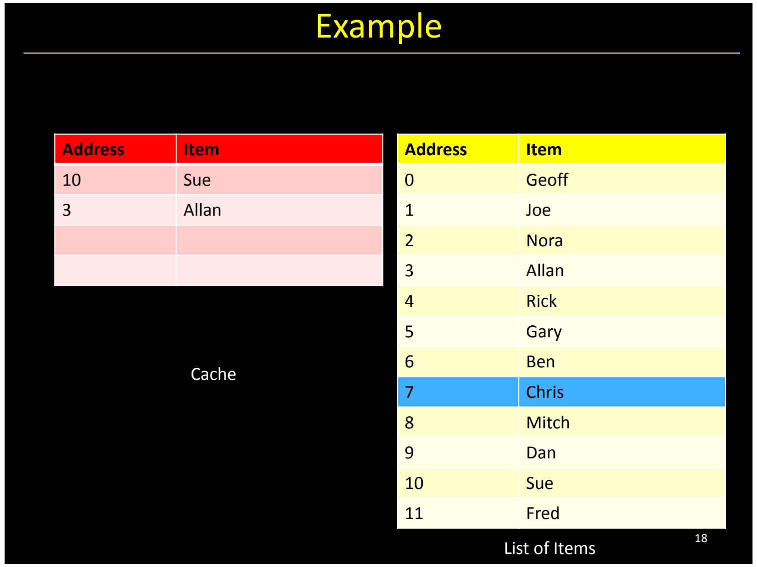
Address	Item
10	Sue

Cache

Address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

17



Example

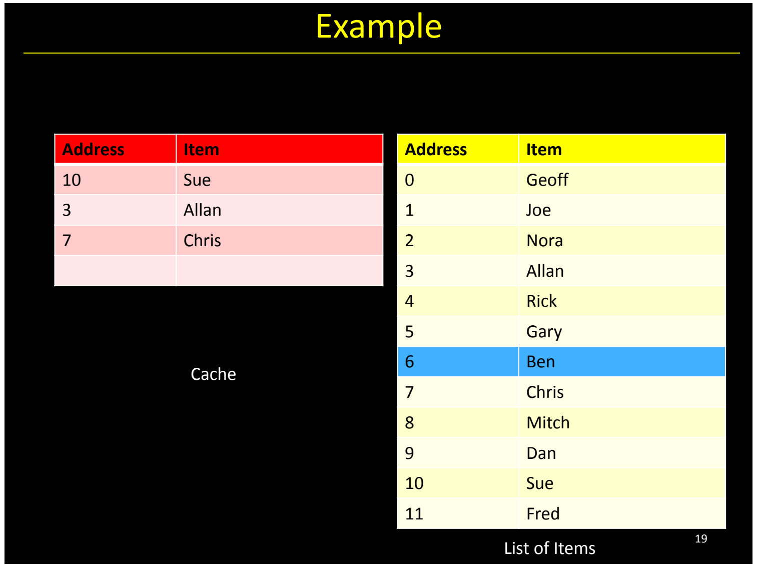
Address	Item
10	Sue
3	Allan

Cache

Address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

18



Example

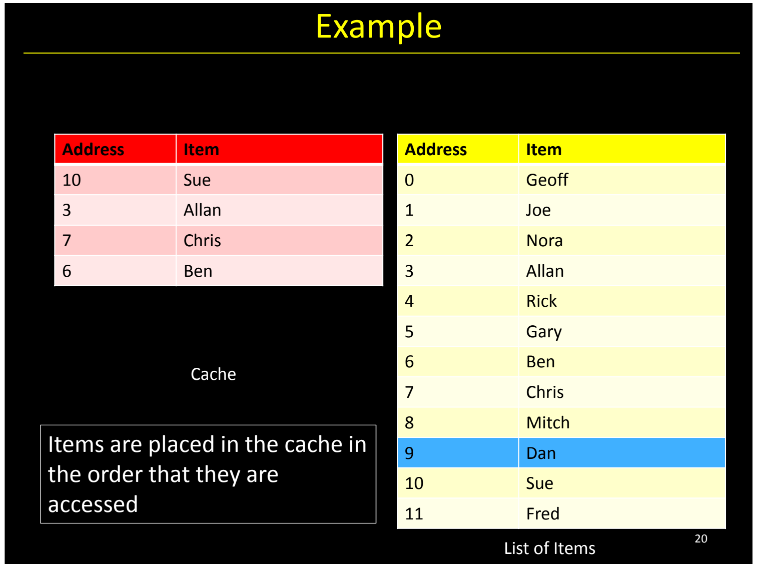
Address	Item
10	Sue
3	Allan
7	Chris

Cache

Address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

19



Example

Address	Item
10	Sue
3	Allan
7	Chris
6	Ben

Cache

Items are placed in the cache in the order that they are accessed

Address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

20

Issues with our Cache

Searching through the cache for a given address is expensive.

- What is the time complexity of the search, in terms of the size of the cache?

Issues with our Cache

Is there a way to reduce the complexity (i.e. speed up the search)?

Direct-Mapped Caches

In our first attempt, any address from the list can go to any slot in the cache – thus there was a need to search *every* slot in the cache, when we looked for an address.

Now, we have a variant of the cache where an address can go to *just one* predefined slot in the cache.

- Address 0 will go to slot 0 (and only slot 0); address 1 will go to slot 1 (and only slot 1); etc;
- Address 4 will go to slot 0 (and only slot 0); address 5 will go to slot 1 (and only slot 1); etc;
- This is called a direct-mapped cache. In contrast, our first cache is called a fully-associative cache (where an address can go to any slot).

Direct-Mapped Caches

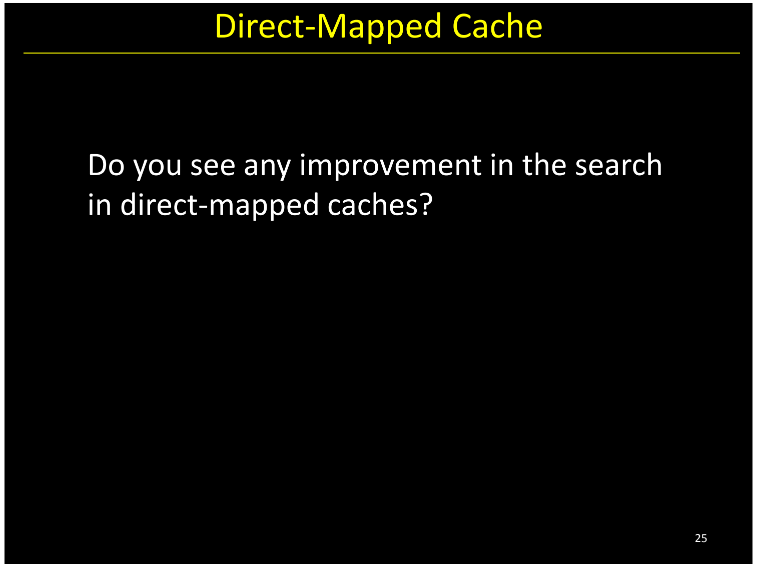
Address	Item

Cache

Which slot in the cache each address of the data will go to?

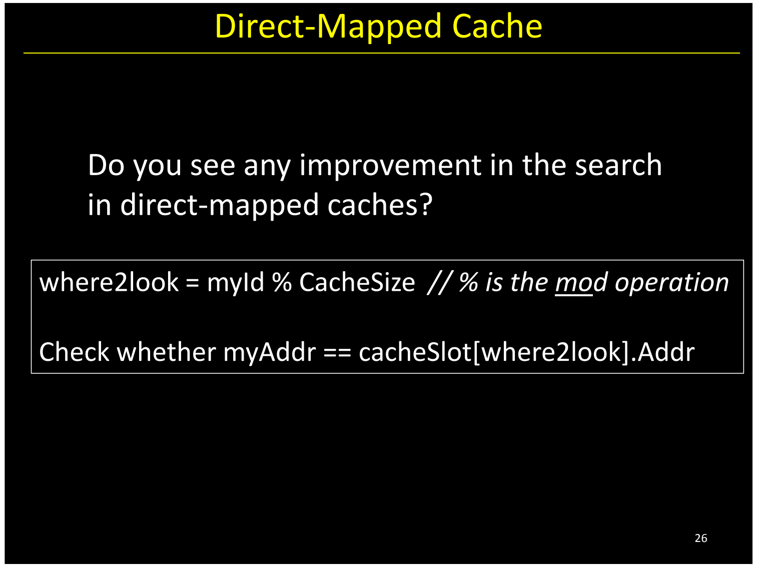
Address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items



Direct-Mapped Cache

Do you see any improvement in the search in direct-mapped caches?

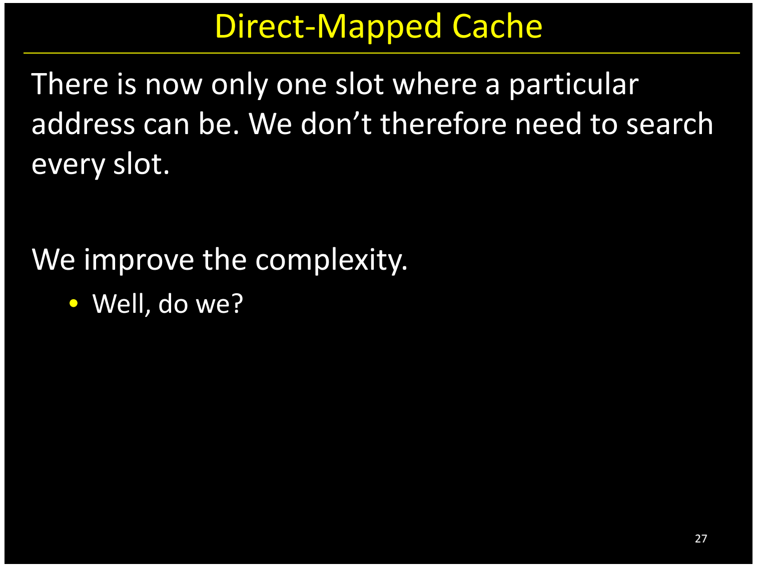


Direct-Mapped Cache

Do you see any improvement in the search in direct-mapped caches?

```
where2look = myId % CacheSize  // % is the mod operation

Check whether myAddr == cacheSlot[where2look].Addr
```

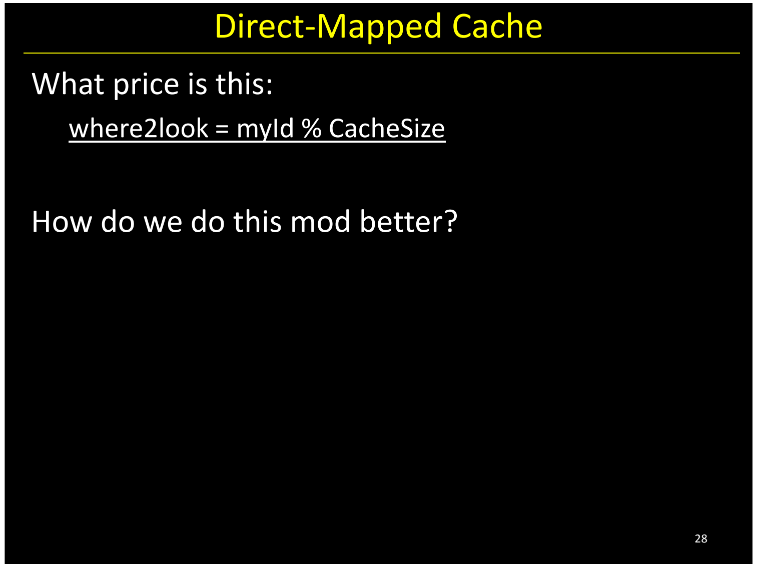


Direct-Mapped Cache

There is now only one slot where a particular address can be. We don’t therefore need to search every slot.

We improve the complexity.

- Well, do we?

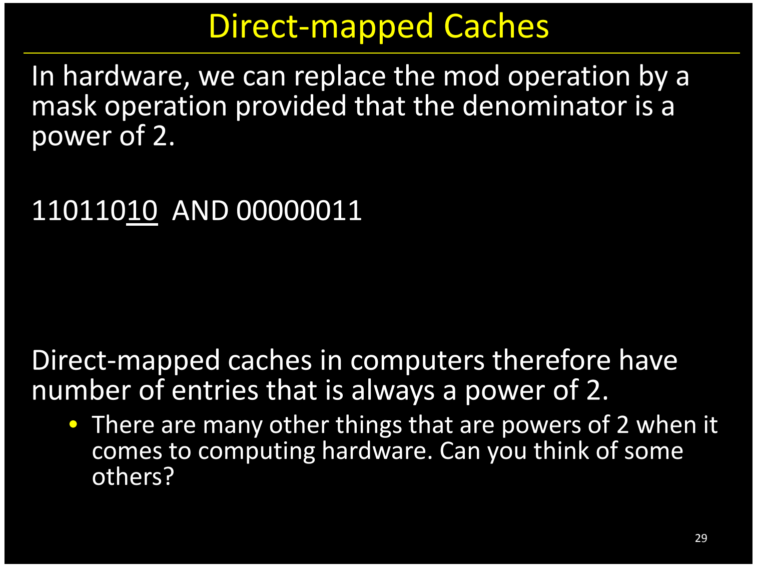


Direct-Mapped Cache

What price is this:

where2look = myId % CacheSize

How do we do this mod better?



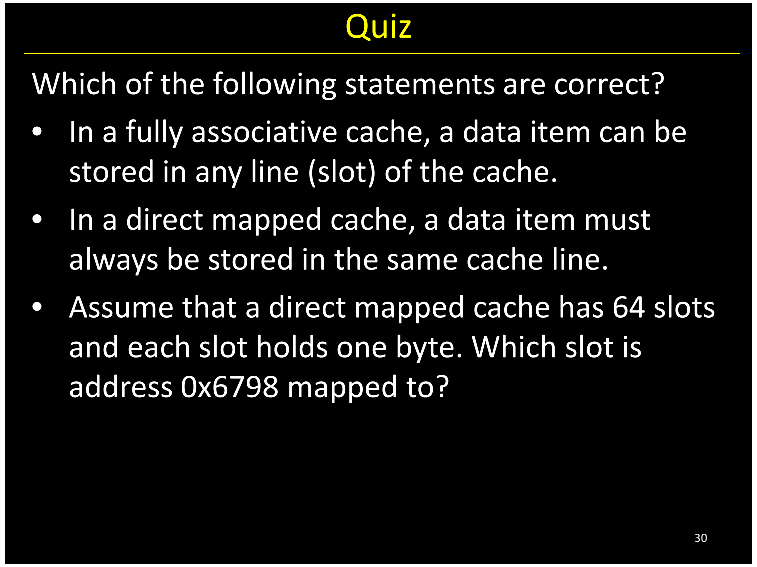
Direct-mapped Caches

In hardware, we can replace the mod operation by a mask operation provided that the denominator is a power of 2.

11011010 AND 00000011

Direct-mapped caches in computers therefore have number of entries that is always a power of 2.

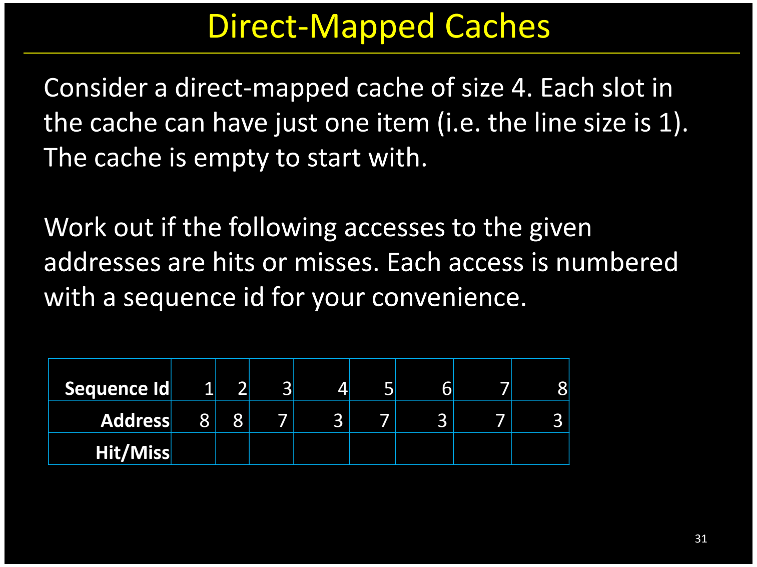
- There are many other things that are powers of 2 when it comes to computing hardware. Can you think of some others?



Quiz

Which of the following statements are correct?

- In a fully associative cache, a data item can be stored in any line (slot) of the cache.
- In a direct mapped cache, a data item must always be stored in the same cache line.
- Assume that a direct mapped cache has 64 slots and each slot holds one byte. Which slot is address 0x6798 mapped to?

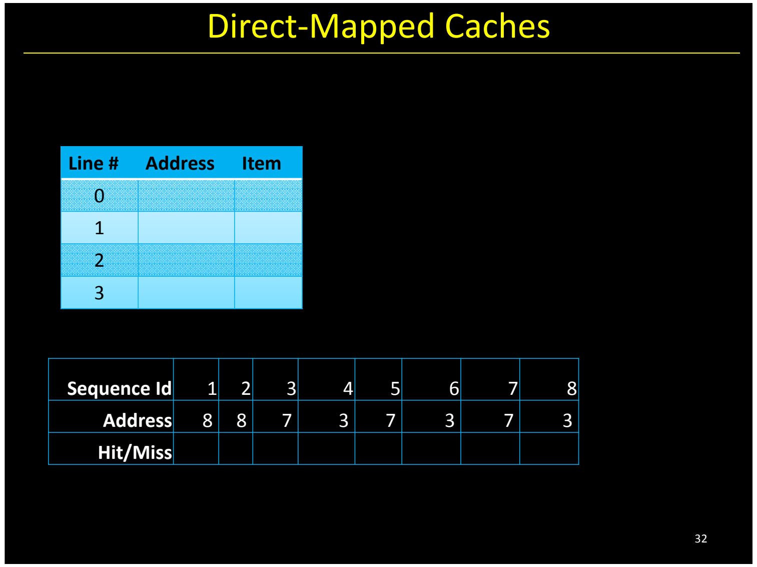


Direct-Mapped Caches

Consider a direct-mapped cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1). The cache is empty to start with.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

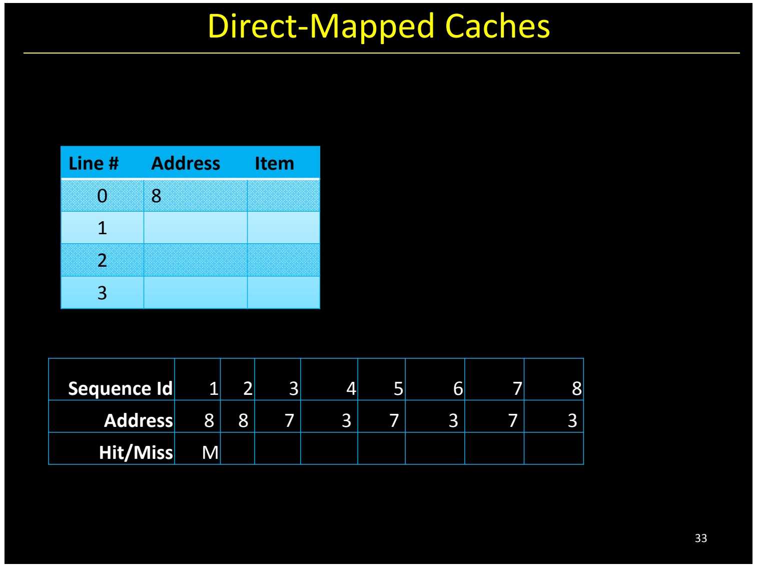
Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss								



Direct-Mapped Caches

Line #	Address	Item
0		
1		
2		
3		

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss								

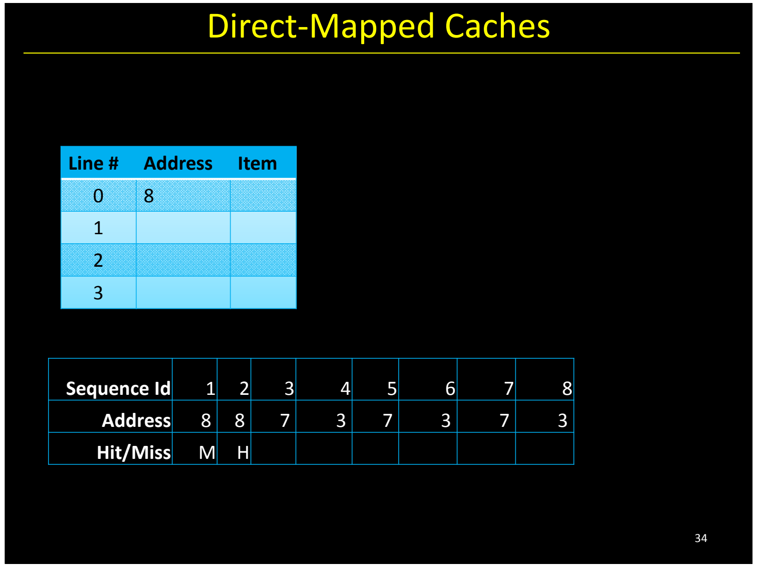


Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3		

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M							

33

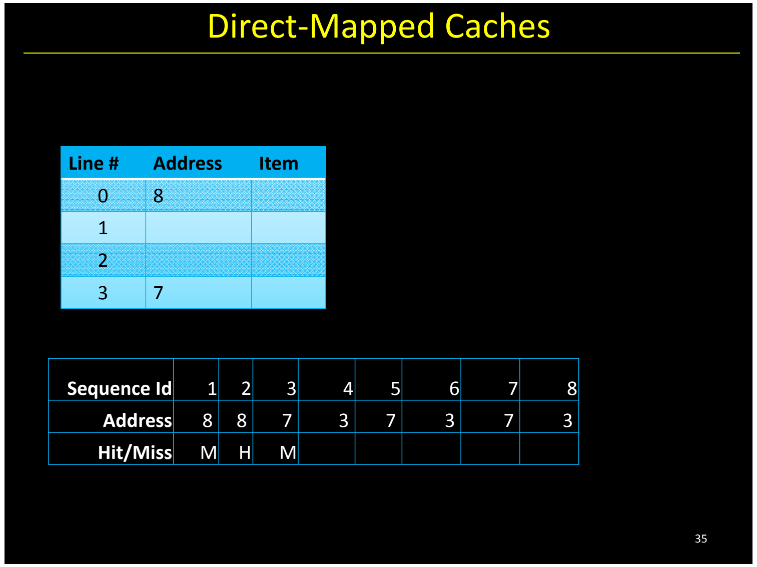


Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3		

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M	H						

34

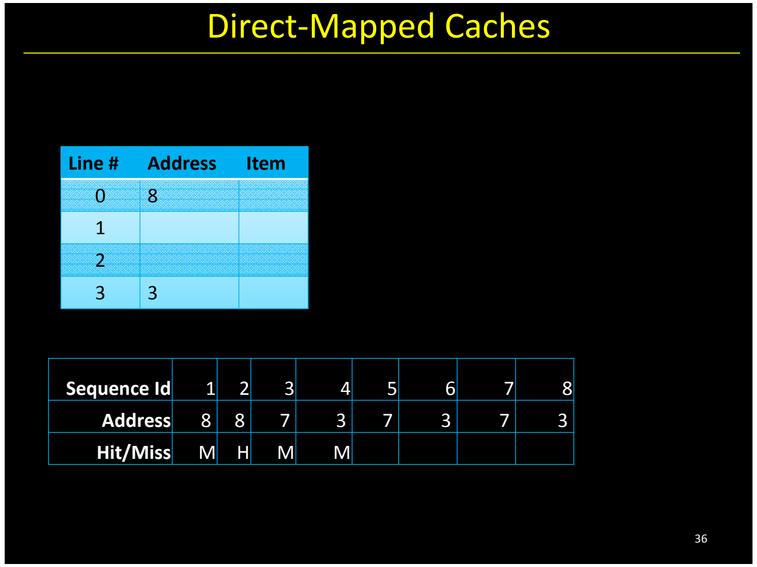


Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M	H	M					

35

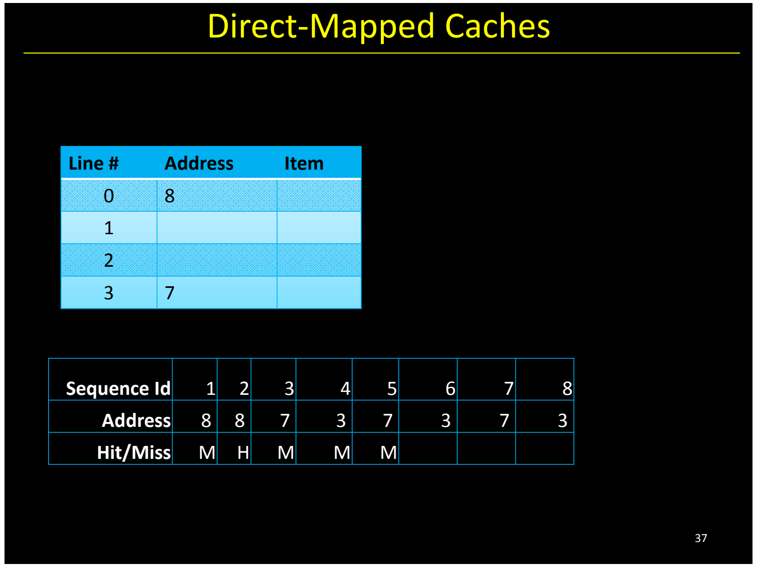


Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3	3	

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M	H	M	M				

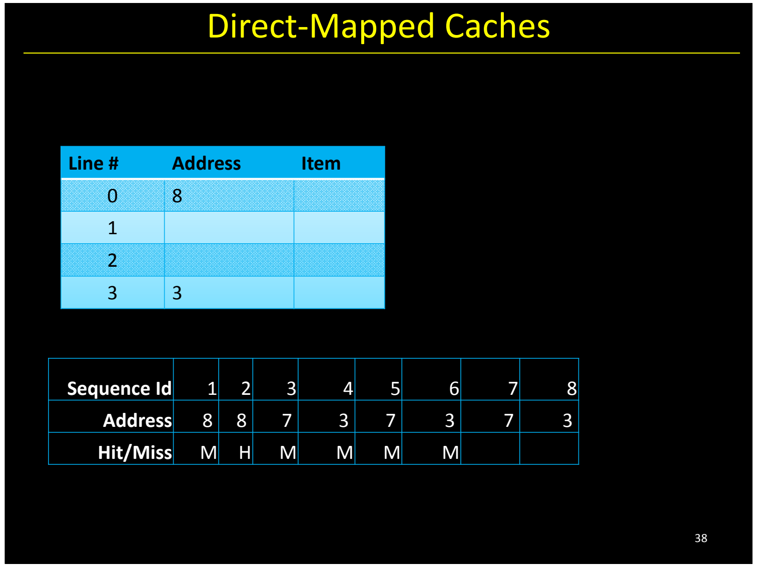
36



Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3	7	

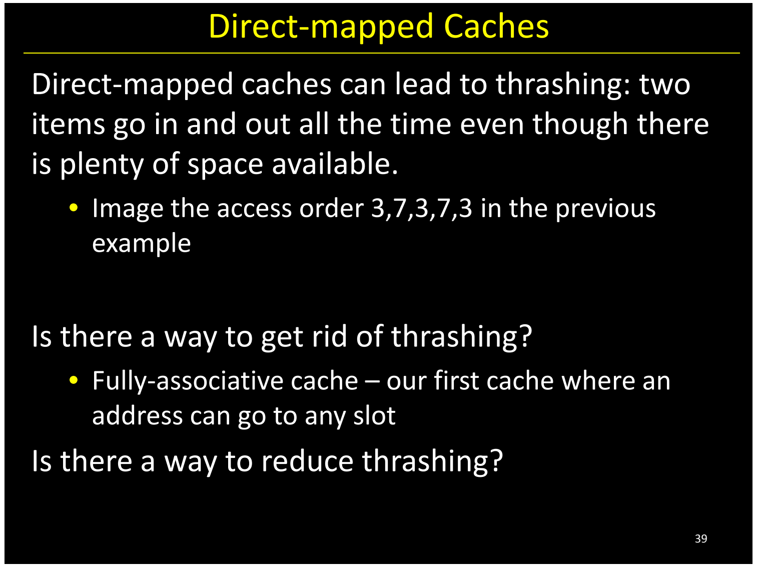
Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M	H	M	M	M			



Direct-Mapped Caches

Line #	Address	Item
0	8	
1		
2		
3	3	

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss	M	H	M	M	M	M		



Direct-mapped Caches

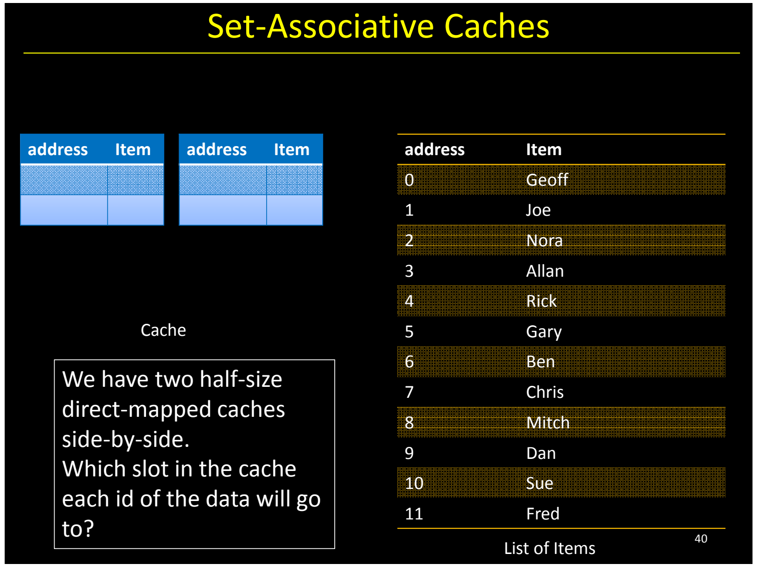
Direct-mapped caches can lead to thrashing: two items go in and out all the time even though there is plenty of space available.

- Image the access order 3,7,3,7,3 in the previous example

Is there a way to get rid of thrashing?

- Fully-associative cache – our first cache where an address can go to any slot

Is there a way to reduce thrashing?



Set-Associative Caches

address	Item

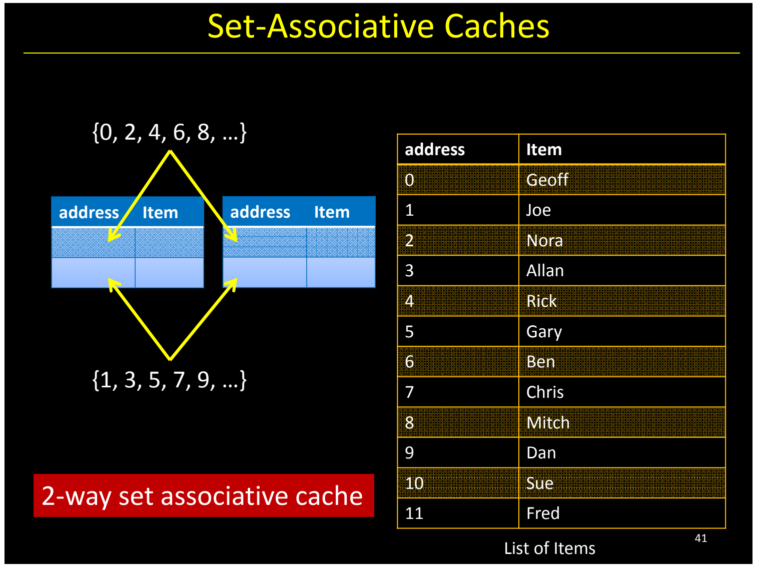
address	Item

Cache

We have two half-size direct-mapped caches side-by-side.
Which slot in the cache each id of the data will go to?

address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

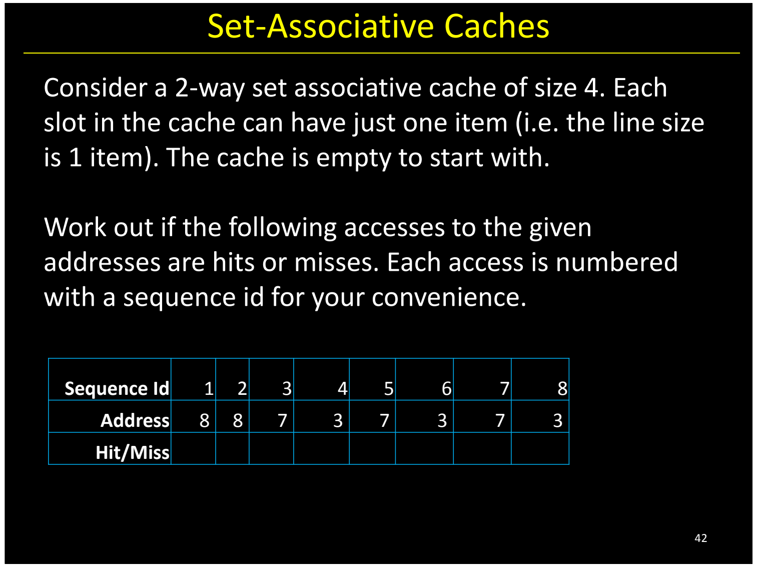


Set-Associative Caches

2-way set associative cache

address	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

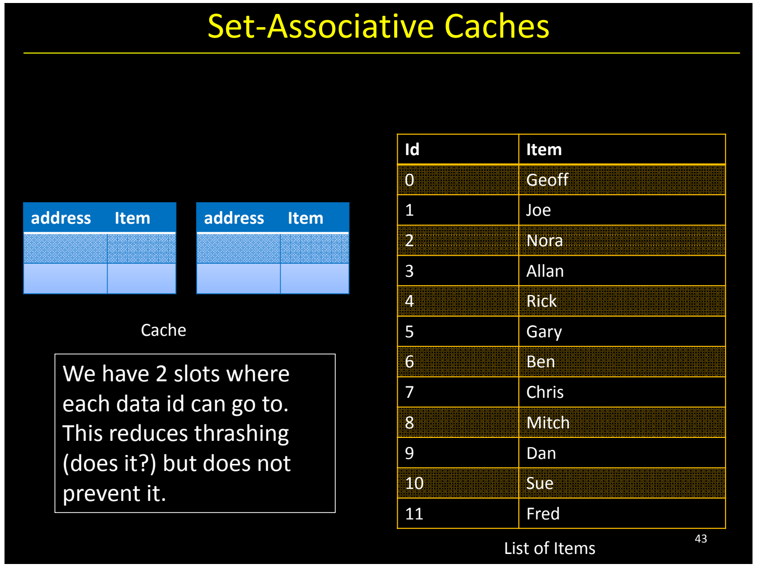


Set-Associative Caches

Consider a 2-way set associative cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8
Address	8	8	7	3	7	3	7	3
Hit/Miss								



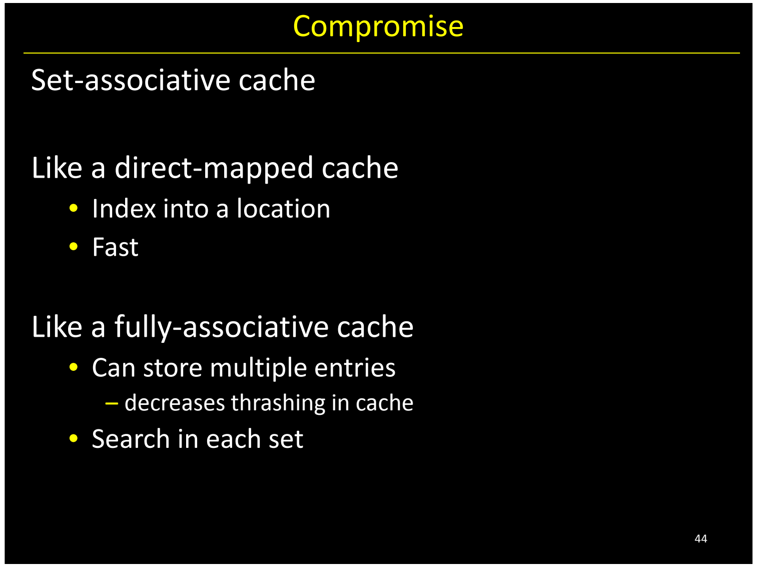
Set-Associative Caches

Cache

We have 2 slots where each data id can go to. This reduces thrashing (does it?) but does not prevent it.

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items



Compromise

Set-associative cache

Like a direct-mapped cache
- Index into a location
- Fast

Like a fully-associative cache
- Can store multiple entries
  - decreases thrashing in cache
- Search in each set

Misses

Three types of misses

- Compulsory
  - The line is being referenced for the first time
- Capacity
  - Miss due to the cache not being big enough to hold the current data set. If the number of active lines is more than the cache can contain, capacity misses take place.
- Conflict
  - The required line was previously there in the cache, but was replaced by another line which happened to map to the same cache location.
  - Conflict misses take place because of limited or zero associativity when lines must be discarded in order to accommodate new lines which are mapped to the same line in the cache.
  - A miss occurs when the replaced line needs to be accessed again.

Misses

Q: How to reduce…

Compulsory Misses

- Unavoidable? The data was never in the cache…
- Prefetching!

Capacity Misses

- Buy more SRAM

Conflict Misses

- Use a more flexible cache design

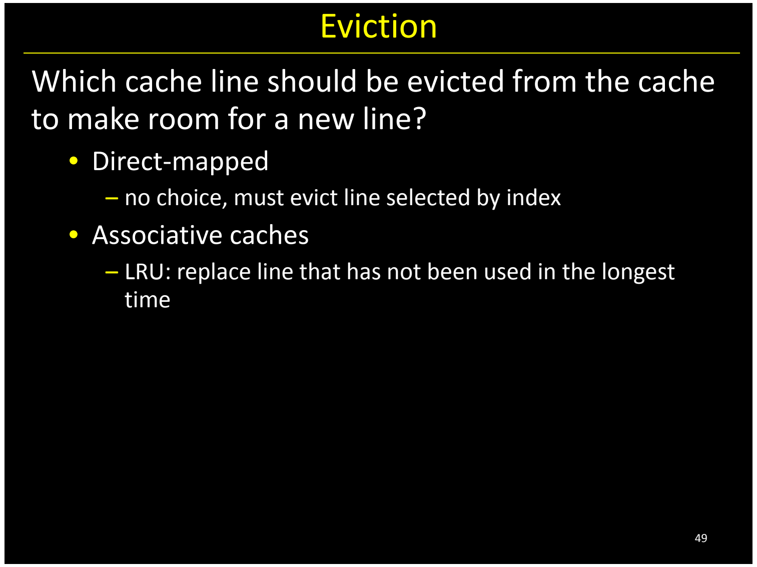
Quiz

Which of the following statements are correct?

- Cache thrashing happens when all slots in the cache are occupied.
- Fully associative cache does not suffer from thrashing.
- In a 2-way set associative cache, an item in the memory can be mapped to 2 slots in the cache.
- Compared with a fully associative cache of the same size, the access speed of a 2-way set associative cache is faster.
- Compulsory miss can be avoided by increasing the size of a cache.
- Conflict miss can be avoided by using 2-way set associative cache.

Agenda

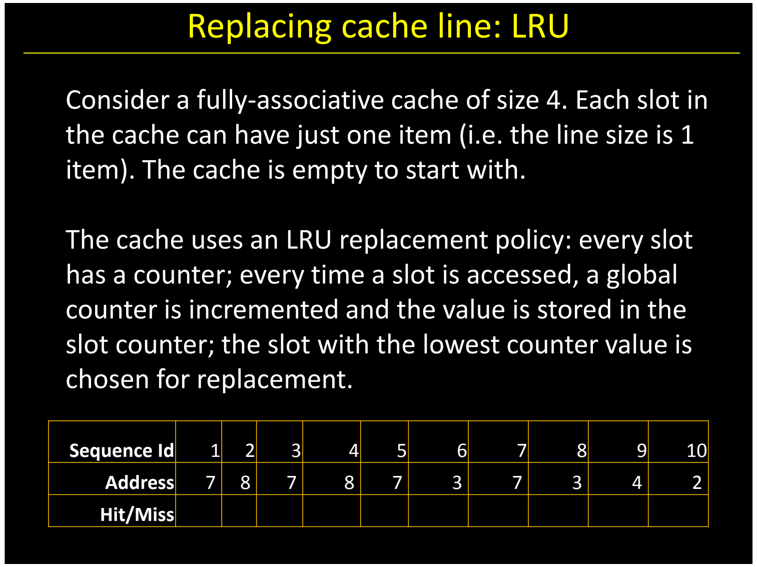
- Memory hierarchy
- Cache
- Managing cache
- Write on cache
- Cache conscious programming



Eviction

Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped
  - no choice, must evict line selected by index
- Associative caches
  - LRU: replace line that has not been used in the longest time

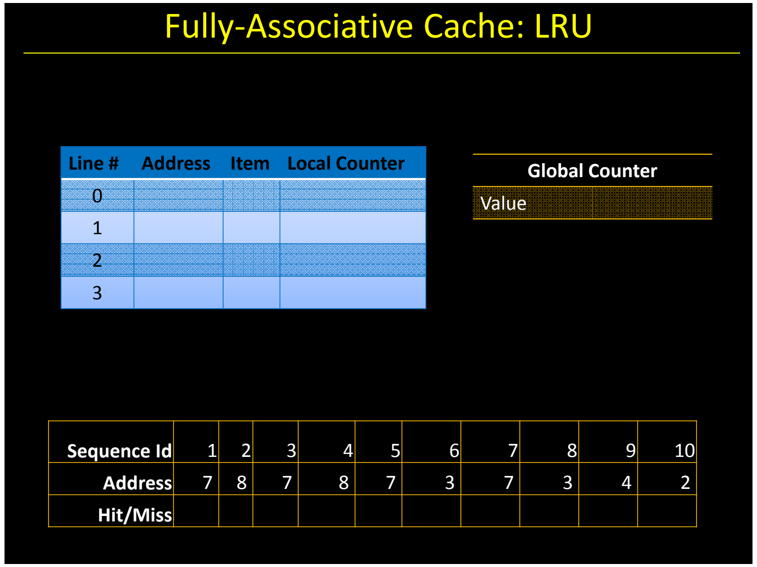


Replacing cache line: LRU

Consider a fully-associative cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

The cache uses an LRU replacement policy: every slot has a counter; every time a slot is accessed, a global counter is incremented and the value is stored in the slot counter; the slot with the lowest counter value is chosen for replacement.

Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss										

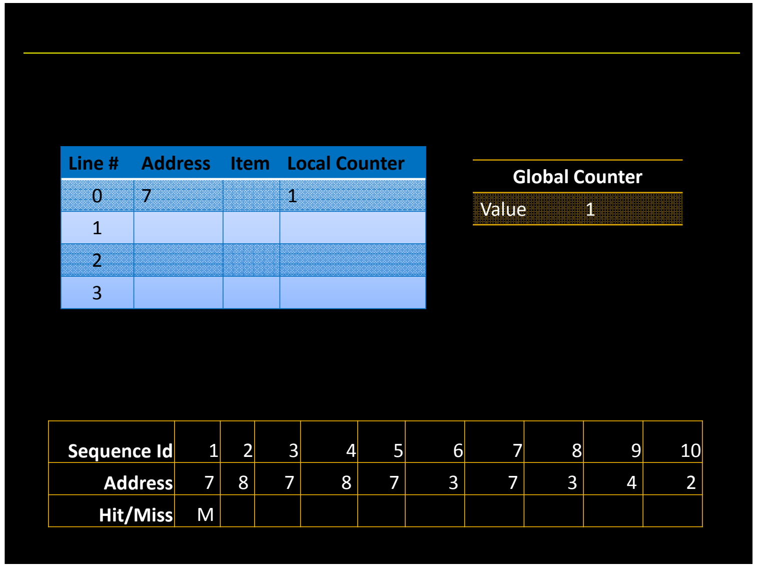


Fully-Associative Cache: LRU

Line #	Address	Item	Local Counter
0			
1			
2			
3			

Global Counter	
Value	

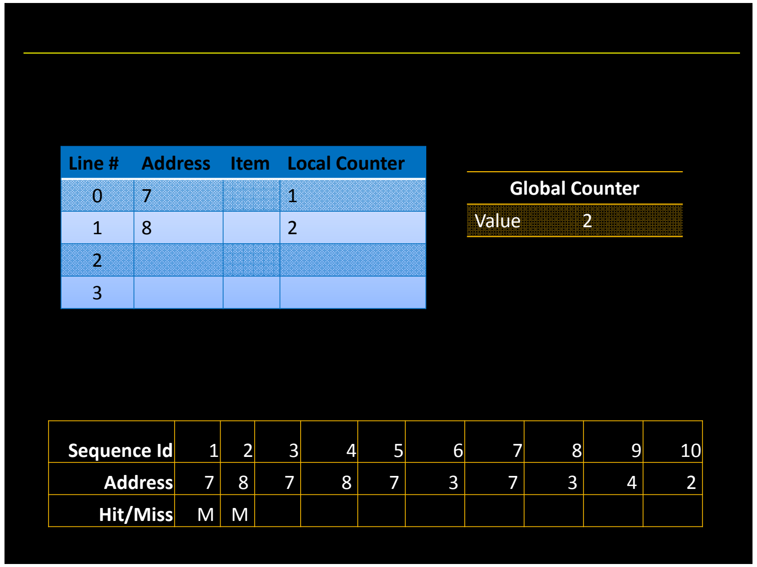
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss										



Line #	Address	Item	Local Counter
0	7		1
1			
2			
3			

Global Counter	
Value	1

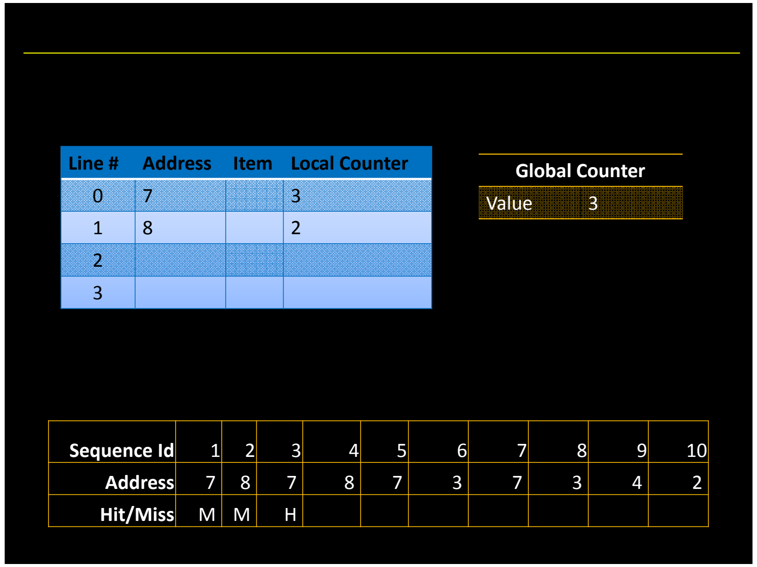
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M									



Line #	Address	Item	Local Counter
0	7		1
1	8		2
2			
3			

Global Counter	
Value	2

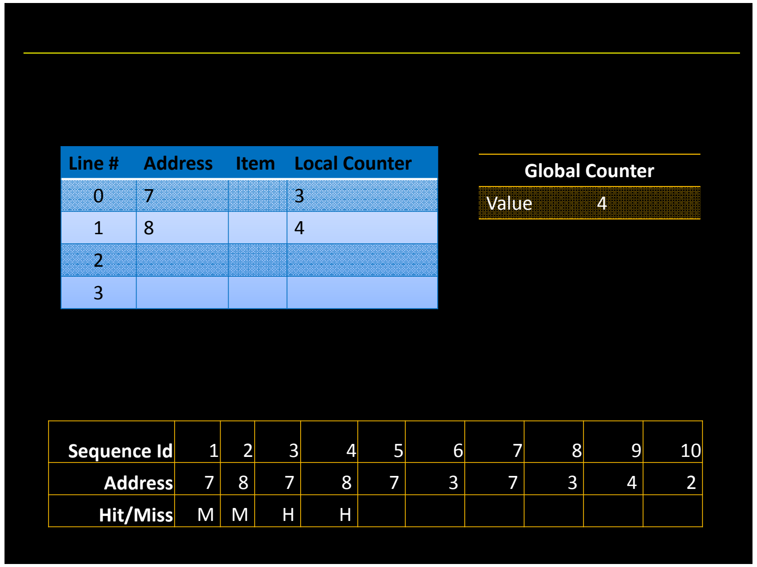
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M								



Line #	Address	Item	Local Counter
0	7		3
1	8		2
2			
3			

Global Counter	
Value	3

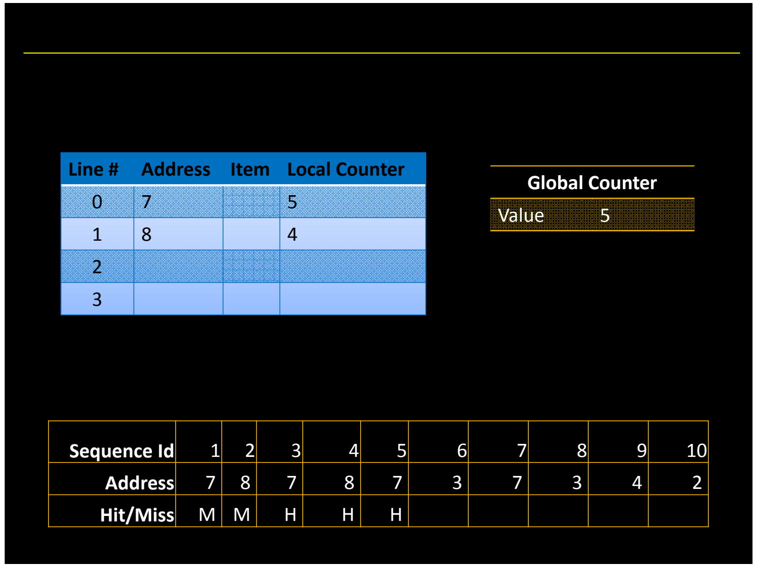
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H							



Line #	Address	Item	Local Counter
0	7		3
1	8		4
2			
3			

Global Counter	
Value	4

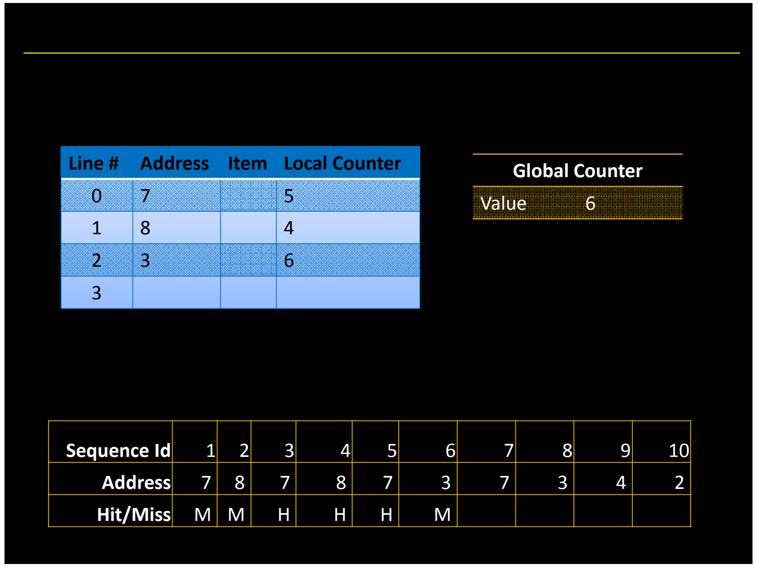
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H						



Line #	Address	Item	Local Counter
0	7		5
1	8		4
2			
3			

Global Counter	
Value	5

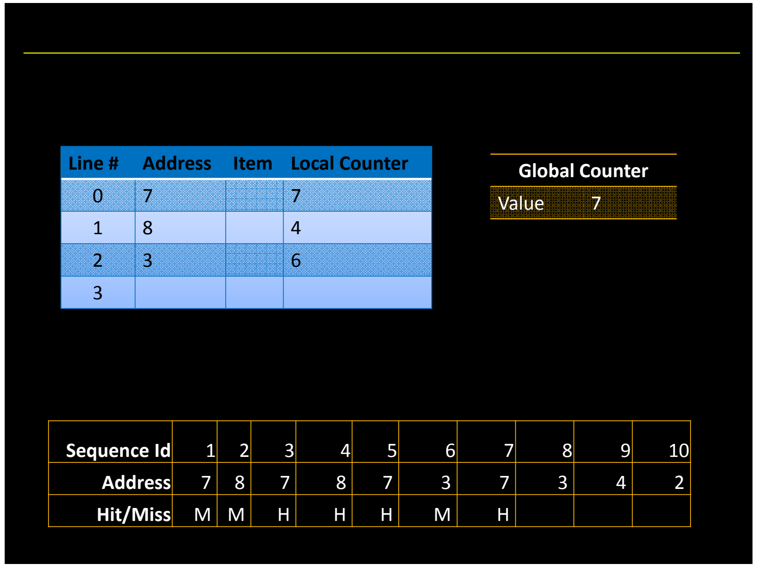
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H					



Line #	Address	Item	Local Counter
0	7		5
1	8		4
2	3		6
3			

Global Counter	
Value	6

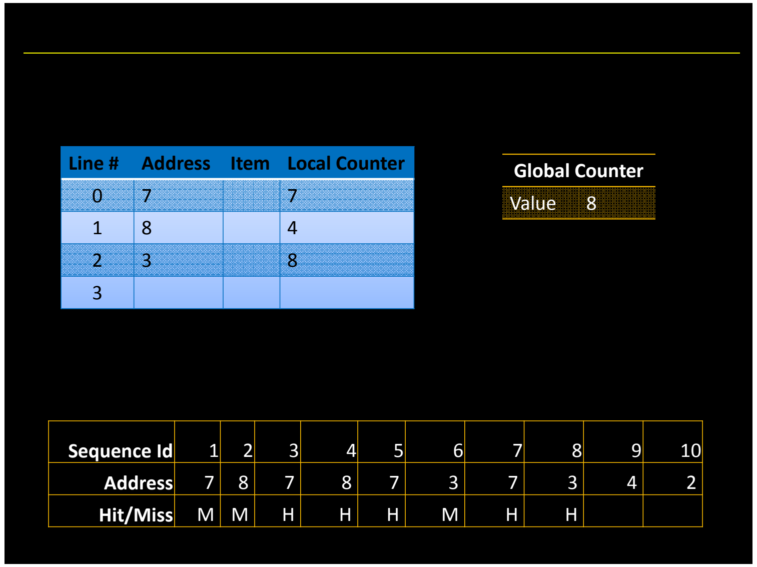
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H	M				



Line #	Address	Item	Local Counter
0	7		7
1	8		4
2	3		6
3			

Global Counter	
Value	7

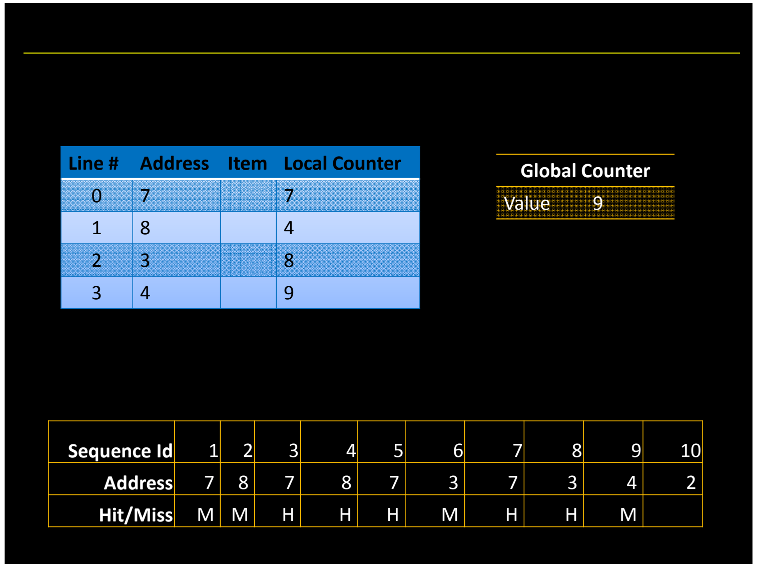
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H	M	H			



Line #	Address	Item	Local Counter
0	7		7
1	8		4
2	3		8
3			

Global Counter	
Value	8

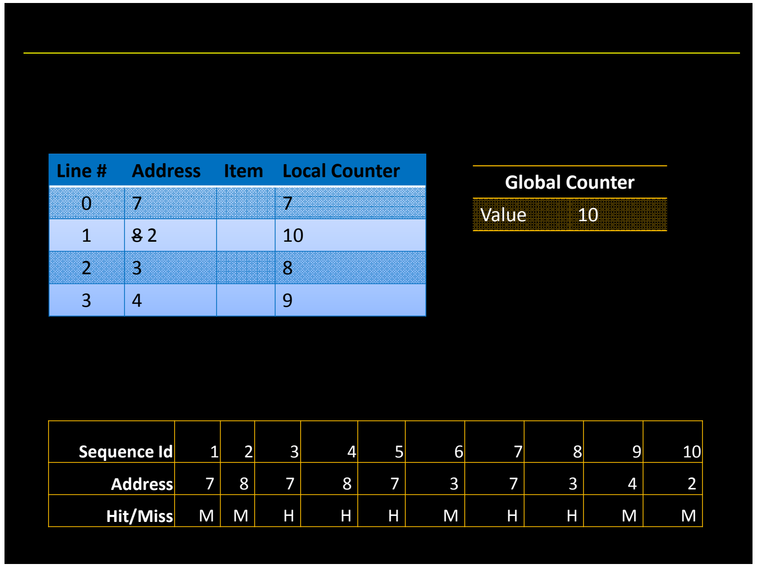
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H	M	H	H		



Line #	Address	Item	Local Counter
0	7		7
1	8		4
2	3		8
3	4		9

Global Counter	
Value	9

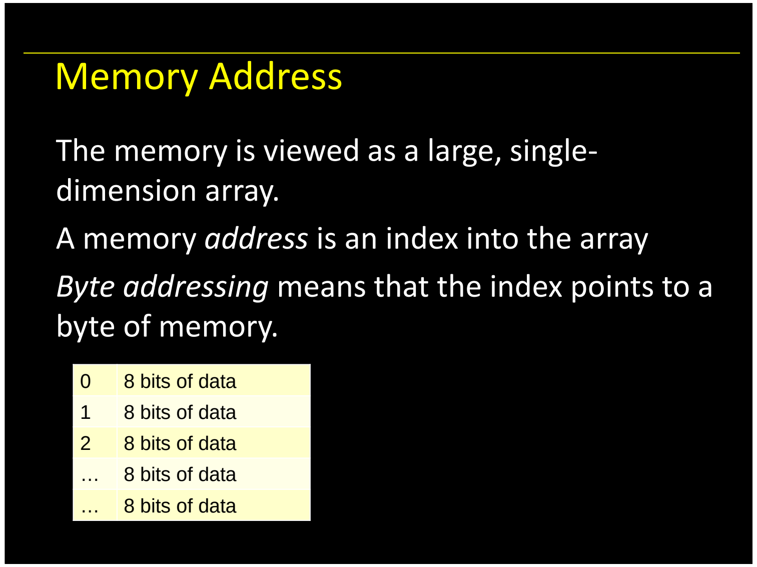
Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H	M	H	H	M	



Line #	Address	Item	Local Counter
0	7		7
1	~~8~~ 2		10
2	3		8
3	4		9

Global Counter	
Value	10

Sequence Id	1	2	3	4	5	6	7	8	9	10
Address	7	8	7	8	7	3	7	3	4	2
Hit/Miss	M	M	H	H	H	M	H	H	M	M



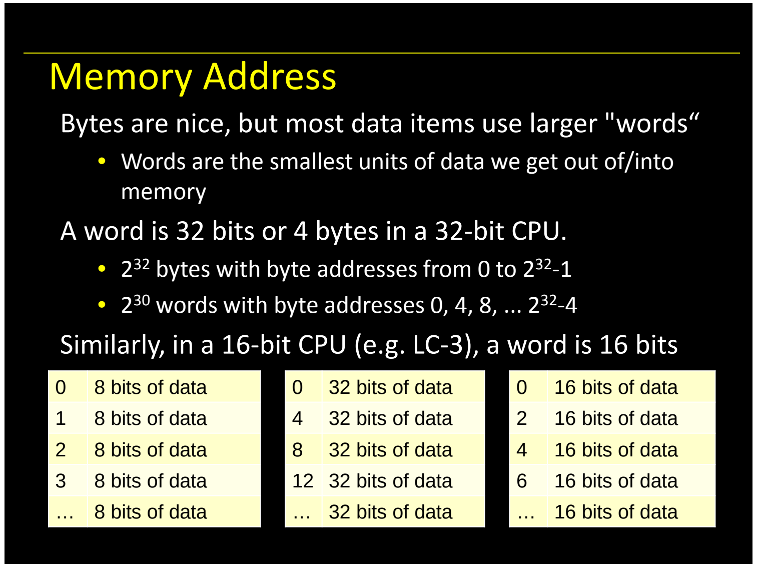
Memory Address

The memory is viewed as a large, single-dimension array.

A memory *address* is an index into the array

Byte addressing means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
…	8 bits of data
…	8 bits of data



Memory Address

Bytes are nice, but most data items use larger "words“

- Words are the smallest units of data we get out of/into memory

A word is 32 bits or 4 bytes in a 32-bit CPU.

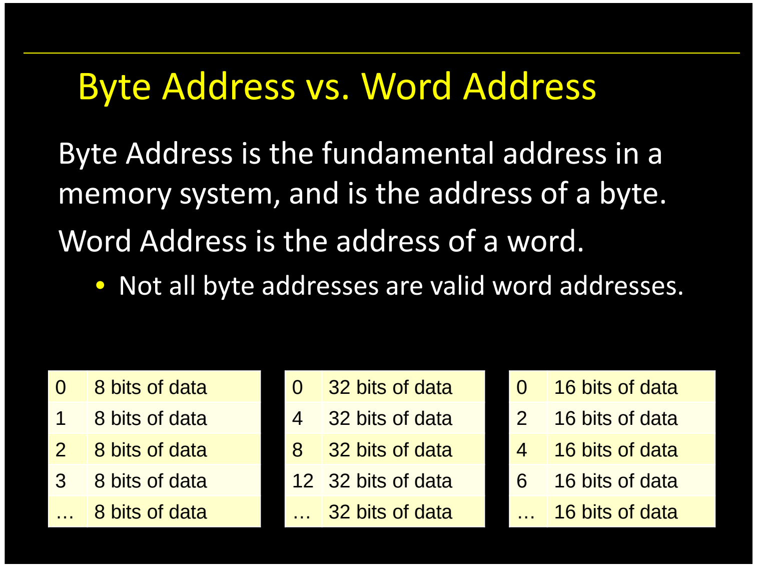
- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

Similarly, in a 16-bit CPU (e.g. LC-3), a word is 16 bits

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
…	8 bits of data

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
…	32 bits of data

0	16 bits of data
2	16 bits of data
4	16 bits of data
6	16 bits of data
…	16 bits of data



Byte Address vs. Word Address

Byte Address is the fundamental address in a memory system, and is the address of a byte.

Word Address is the address of a word.

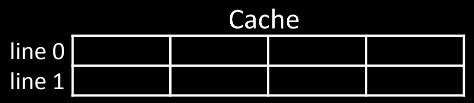
- Not all byte addresses are valid word addresses.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
…	8 bits of data

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
…	32 bits of data

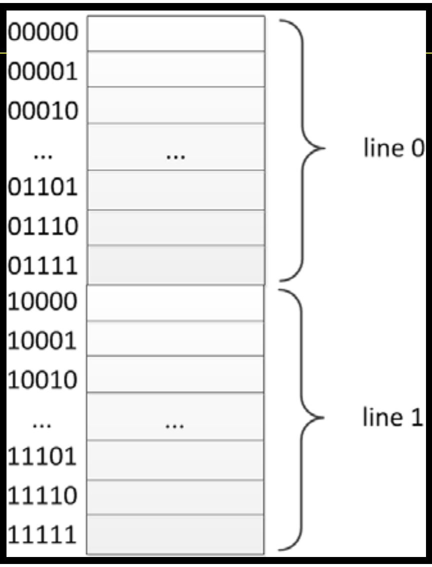
0	16 bits of data
2	16 bits of data
4	16 bits of data
6	16 bits of data
…	16 bits of data

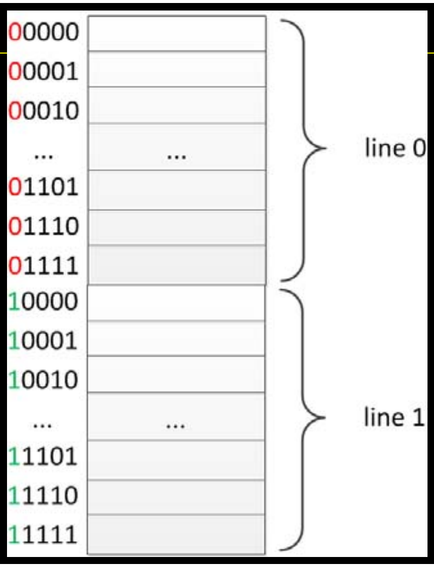
- A cache is organised as a collection of lines.
- Each line consists of one or several bytes.



- A cache with 2 lines where each line has four words and each word has four bytes. How do we map a memory location to a cell in the cache (assume the cache is a direct mapped cache)?

- The total number of bytes in the cache is
  - num of lines × num of words in a line × num of bytes in a word
  - 2 × 4 × 4 = 32
- To determine the cell in the cache, we carry out a MOD operation
  - address MOD 32
  - look at the last five bits of the address





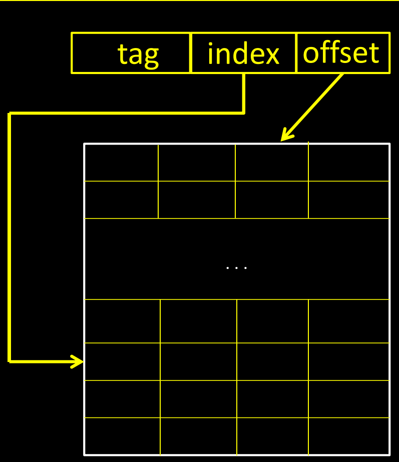
- In the 5 least significant bits of the address, the left most bit determines the cache line and the rest of the bits decide the position of a byte in a line.
- In general, if there are 2^n cache lines and each line has 2^m bytes,
  - There are 2^{m+n} bytes in the cache.
  - The m+n least significant bits of a memory address are used to determine where a byte should be stored in a cache.
  - Among the m+n bits, the left-most n bits decides which cache line a byte is stored, and the other m bits indicate position of the byte with a cache line.

Mapping memory address to cache

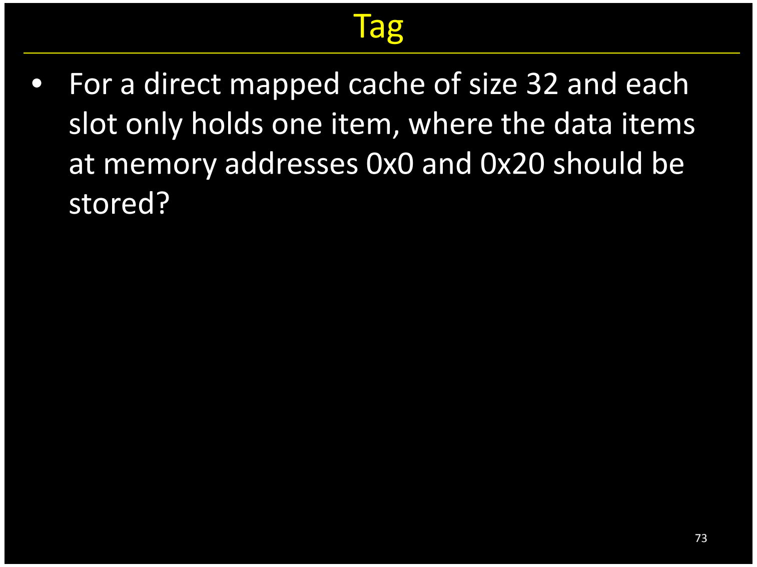
Generally speaking, for direct mapped and set associative cache, when deciding which location in the cache a memory location should be stored, the memory address is partitioned into three fields.

- The offset field determines the position in a cache line.
- The index field decides the cache line.
- The tag field is used to distinguish different memory locations that are mapped to the same cache location.

tag	index	offset

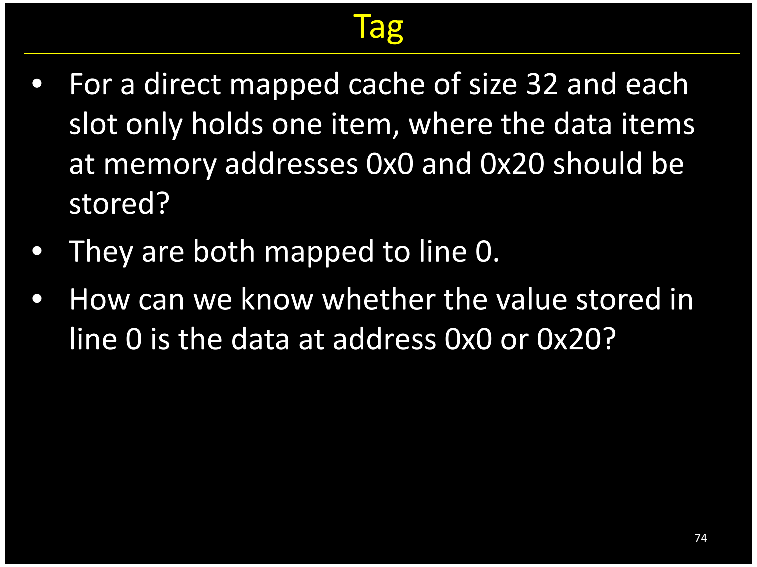


- If each cache line only has one byte, the offset field is not needed.
  - Each line only has one location.
  - The address consists of two fields, i.e. tag and index



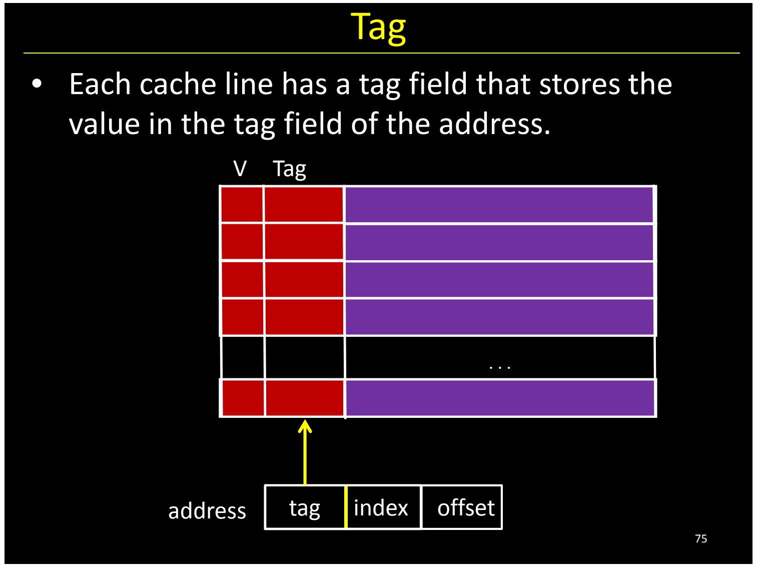
Tag

- For a direct mapped cache of size 32 and each slot only holds one item, where the data items at memory addresses 0x0 and 0x20 should be stored?



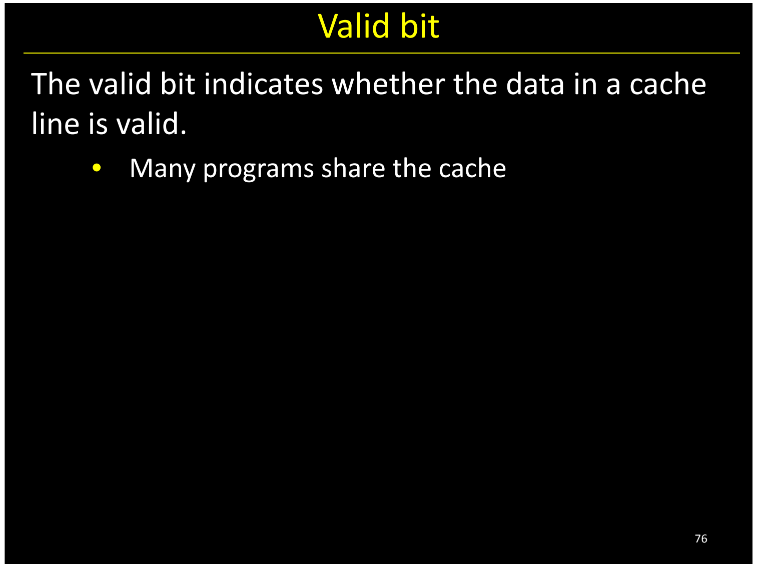
Tag

- For a direct mapped cache of size 32 and each slot only holds one item, where the data items at memory addresses 0x0 and 0x20 should be stored?
- They are both mapped to line 0.
- How can we know whether the value stored in line 0 is the data at address 0x0 or 0x20?



Tag

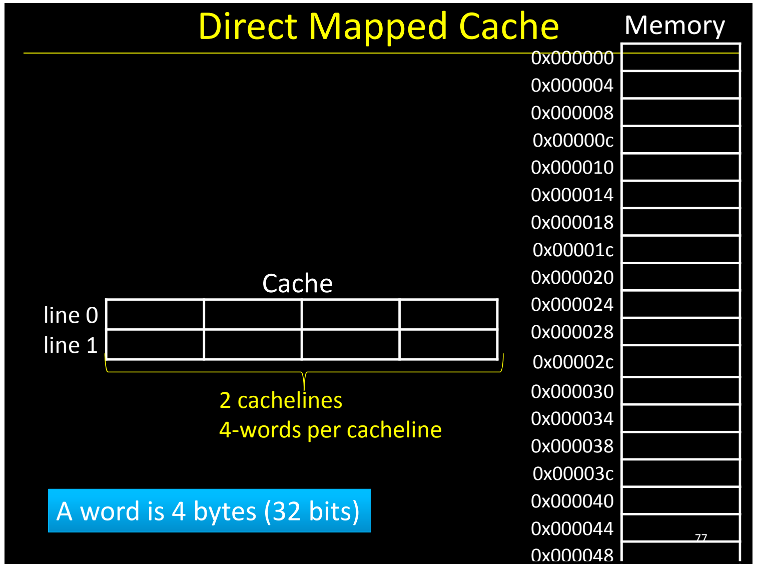
- Each cache line has a tag field that stores the value in the tag field of the address.



Valid bit

The valid bit indicates whether the data in a cache line is valid.

- Many programs share the cache



Direct Mapped Cache

Memory

0x000000
0x000004
0x000008
0x00000c
0x000010
0x000014
0x000018
0x00001c
0x000020
0x000024
0x000028
0x00002c
0x000030
0x000034
0x000038
0x00003c
0x000040
0x000044
0x000048

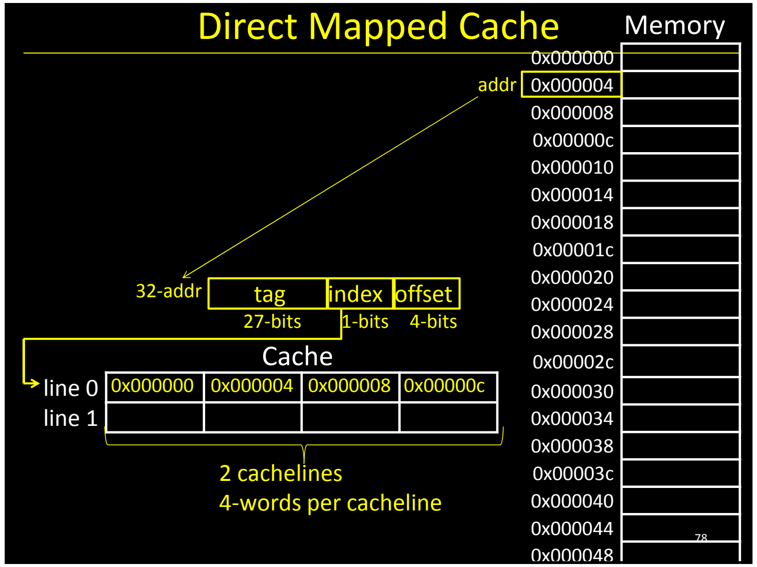
Cache

line 0
line 1

2 cachelines
4-words per cacheline

A word is 4 bytes (32 bits)

77



Direct Mapped Cache

Memory

addr 0x000004

0x000000
0x000004
0x000008
0x00000c
0x000010
0x000014
0x000018
0x00001c
0x000020
0x000024
0x000028
0x00002c
0x000030
0x000034
0x000038
0x00003c
0x000040
0x000044
0x000048

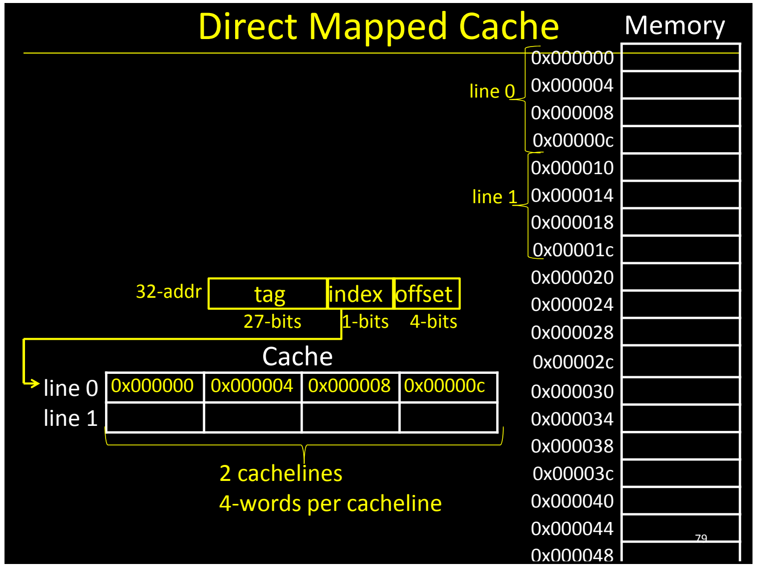
32-addr	tag	index	offset
	27-bits	1-bits	4-bits

Cache

line 0	0x000000	0x000004	0x000008	0x00000c
line 1				

2 cachelines
4-words per cacheline

78



Direct Mapped Cache

Memory

line 0
line 1

0x000000
0x000004
0x000008
0x00000c
0x000010
0x000014
0x000018
0x00001c
0x000020
0x000024
0x000028
0x00002c
0x000030
0x000034
0x000038
0x00003c
0x000040
0x000044
0x000048

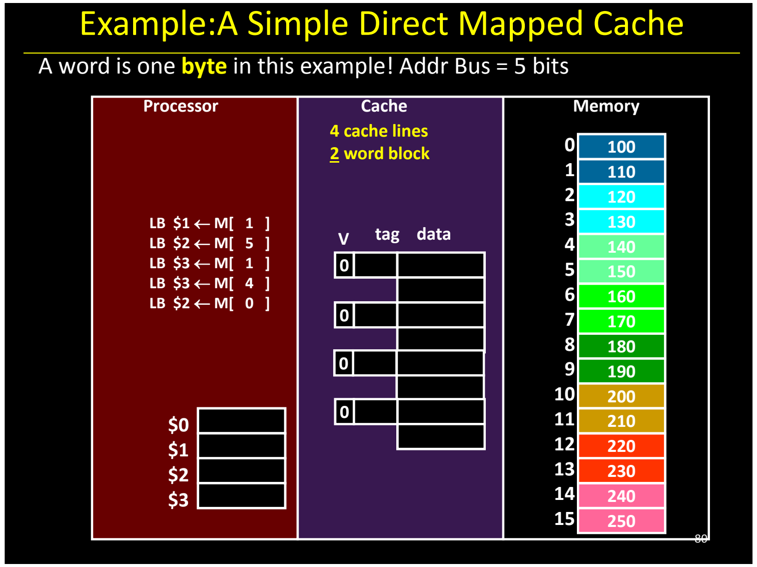
32-addr	tag	index	offset
	27-bits	1-bits	4-bits

Cache

line 0	0x000000	0x000004	0x000008	0x00000c
line 1				

2 cachelines
4-words per cacheline

79



Example:A Simple Direct Mapped Cache

A word is one **byte** in this example! Addr Bus = 5 bits

Processor

```
LB  $1 ← M[  1  ]
LB  $2 ← M[  5  ]
LB  $3 ← M[  1  ]
LB  $3 ← M[  4  ]
LB  $2 ← M[  0  ]
```

$0
$1
$2
$3

Cache

4 cache lines
2 word block

V	tag	data
0		
0		
0		
0		

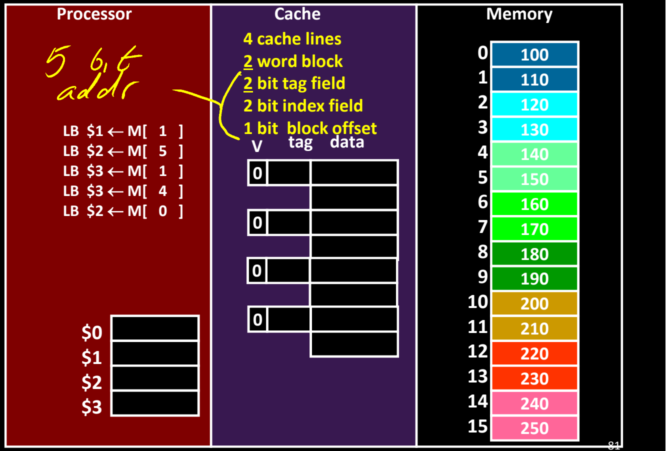
Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

80

Example:A Simple Direct Mapped Cache

Using **byte addresses** in this example! Addr Bus = 5 bits



Processor

5 bit addr

```
LB  $1 ← M[  1  ]
LB  $2 ← M[  5  ]
LB  $3 ← M[  1  ]
LB  $3 ← M[  4  ]
LB  $2 ← M[  0  ]
```

Register	Value
$0	
$1	
$2	
$3	

Cache

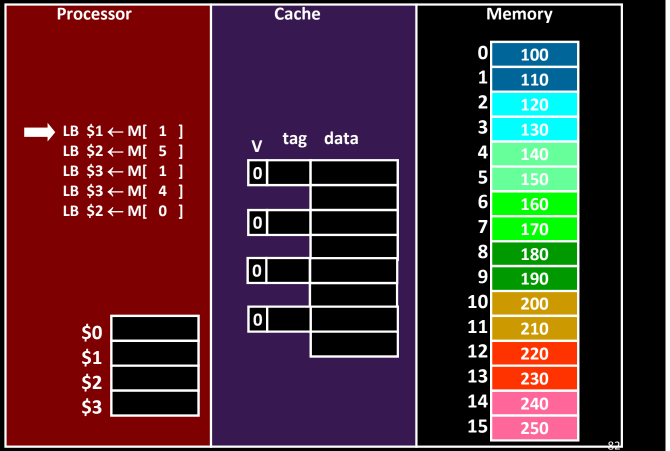
4 cache lines
2 word block
2 bit tag field
2 bit index field
1 bit block offset

V	tag	data
0		
0		
0		
0		

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

1st Access



Processor

```
➡ LB  $1 ← M[  1  ]
  LB  $2 ← M[  5  ]
  LB  $3 ← M[  1  ]
  LB  $3 ← M[  4  ]
  LB  $2 ← M[  0  ]
```

Register	Value
$0	
$1	
$2	
$3	

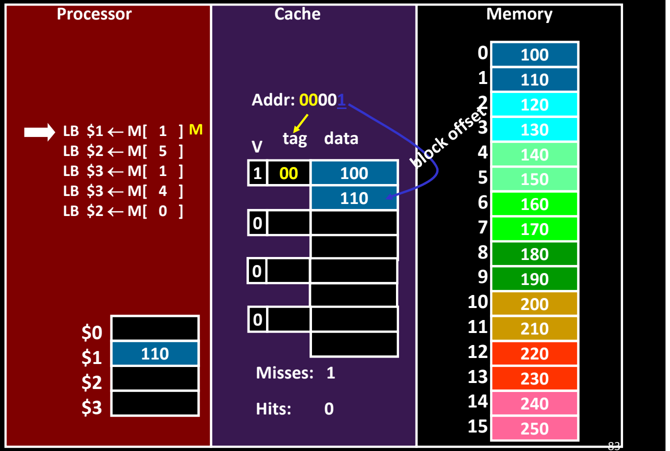
Cache

V	tag	data
0		
0		
0		
0		

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

1st Access



Processor

```
➡ LB  $1 ← M[  1  ] M
  LB  $2 ← M[  5  ]
  LB  $3 ← M[  1  ]
  LB  $3 ← M[  4  ]
  LB  $2 ← M[  0  ]
```

Register	Value
$0	
$1	110
$2	
$3	

Cache

Addr: 00001

V	tag	data
1	00	100
		110
0		
0		
0		

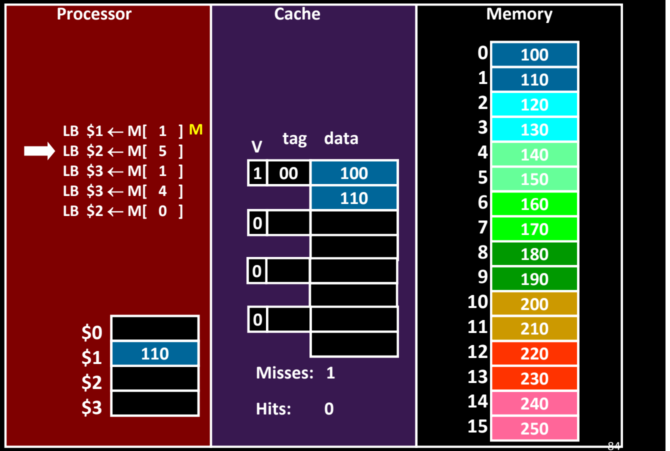
Misses: 1

Hits: 0

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

2nd Access



Processor

```
  LB  $1 ← M[  1  ] M
➡ LB  $2 ← M[  5  ]
  LB  $3 ← M[  1  ]
  LB  $3 ← M[  4  ]
  LB  $2 ← M[  0  ]
```

Register	Value
$0	
$1	110
$2	
$3	

Cache

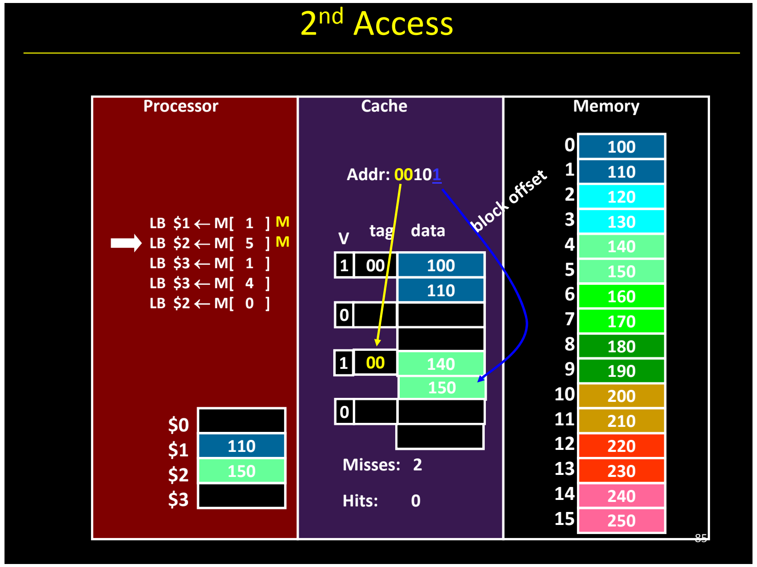
V	tag	data
1	00	100
		110
0		
0		
0		

Misses: 1

Hits: 0

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250



2nd Access

Processor

LB $1 ← M[1] M
⇨ LB $2 ← M[5] M
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]

$0	
$1	110
$2	150
$3	

Cache

Addr: 00101

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

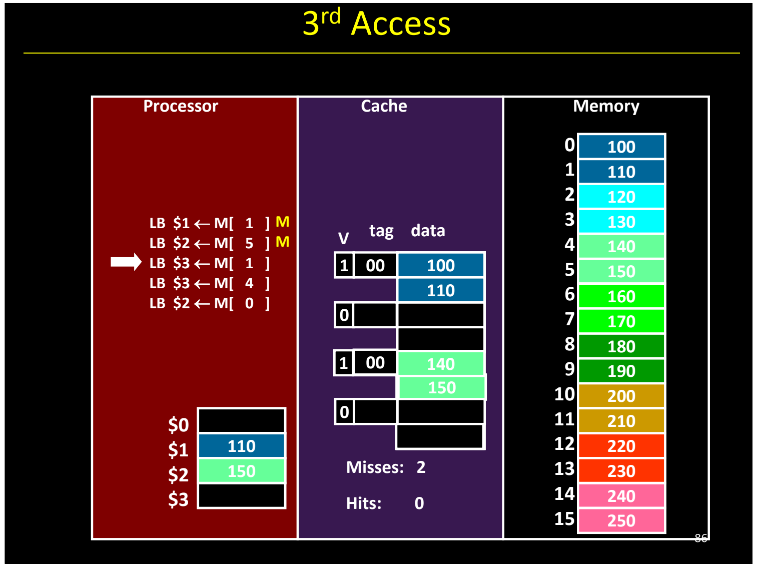
Misses: 2

Hits: 0

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

85



3rd Access

Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
⇨ LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]

$0	
$1	110
$2	150
$3	

Cache

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

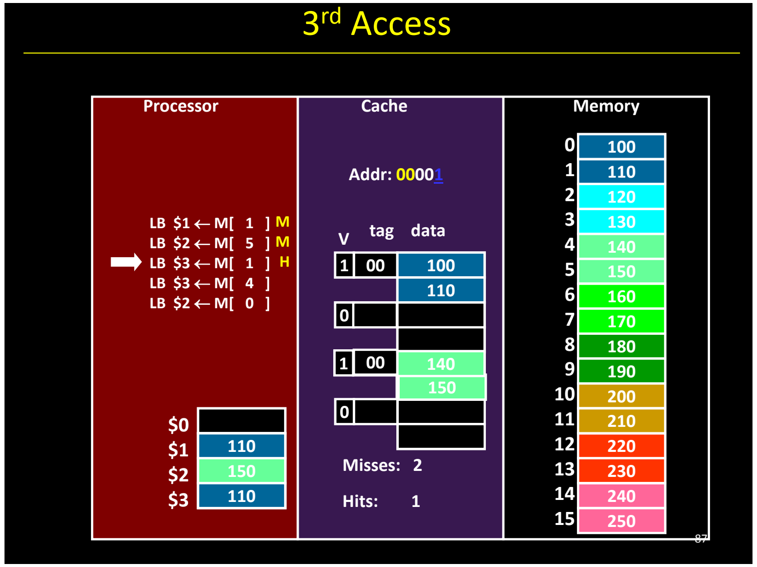
Misses: 2

Hits: 0

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

86



3rd Access

Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
⇨ LB $3 ← M[1] H
LB $3 ← M[4]
LB $2 ← M[0]

$0	
$1	110
$2	150
$3	110

Cache

Addr: 00001

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

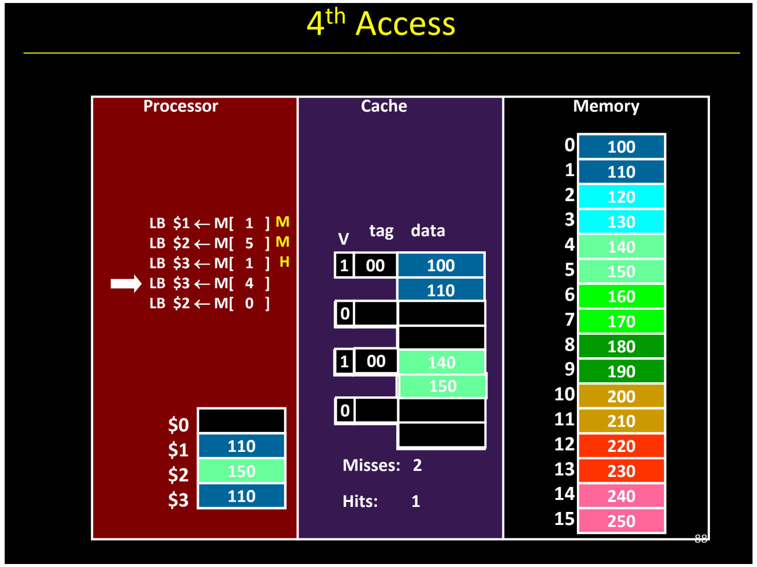
Misses: 2

Hits: 1

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

87



4th Access

Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
⇨ LB $3 ← M[4]
LB $2 ← M[0]

$0	
$1	110
$2	150
$3	110

Cache

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

Misses: 2

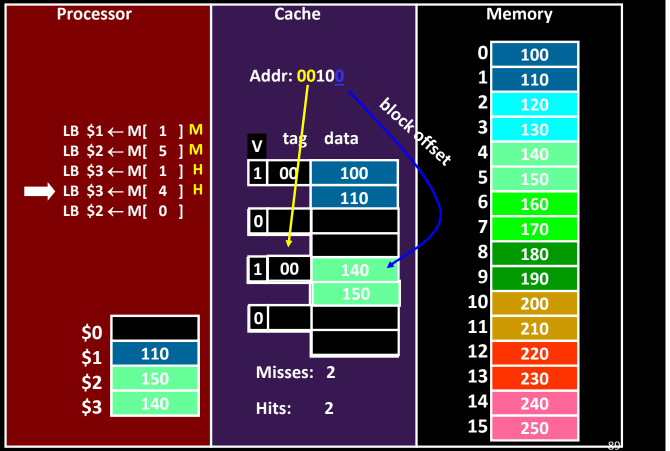
Hits: 1

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

88

4th Access



Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
➡ LB $3 ← M[4] H
LB $2 ← M[0]

$0	
$1	110
$2	150
$3	140

Cache

Addr: 00100

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

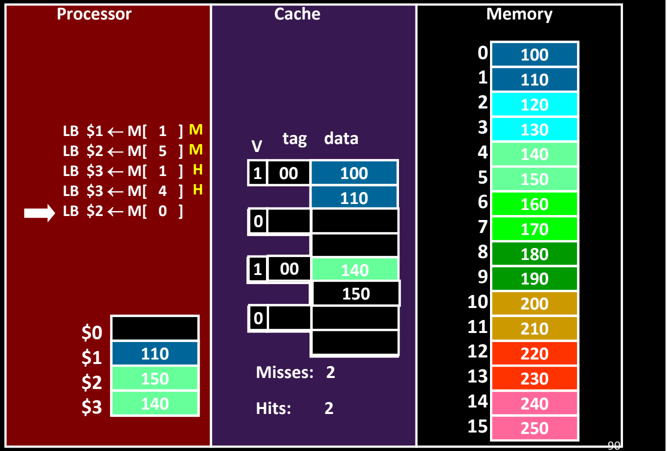
Misses: 2

Hits: 2

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

5th Access



Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
➡ LB $2 ← M[0]

$0	
$1	110
$2	150
$3	140

Cache

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

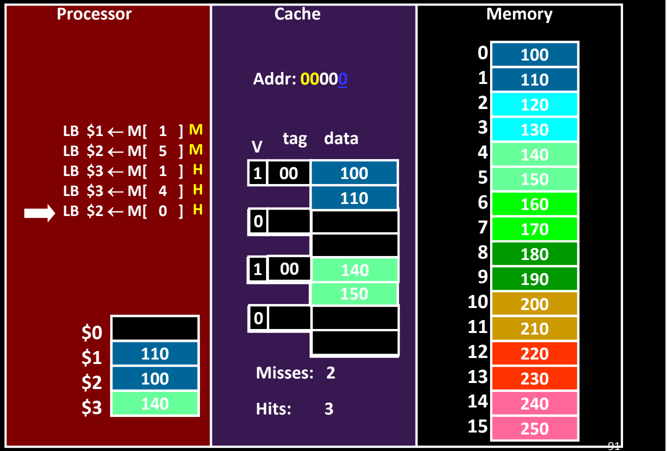
Misses: 2

Hits: 2

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

5th Access



Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
➡ LB $2 ← M[0] H

$0	
$1	110
$2	100
$3	140

Cache

Addr: 00000

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

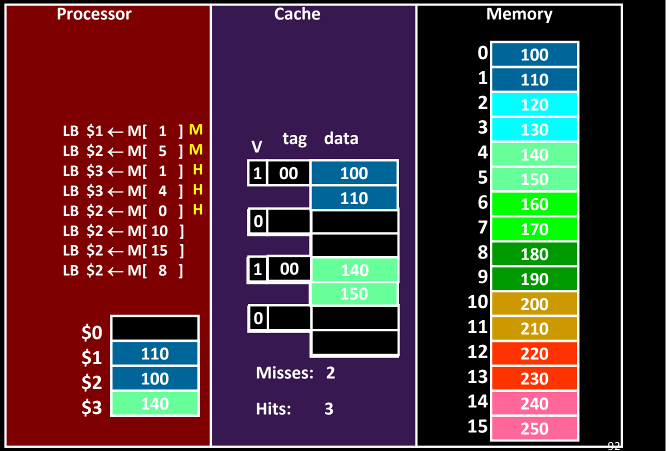
Misses: 2

Hits: 3

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Exercise



Processor

LB $1 ← M[1] M
LB $2 ← M[5] M
LB $3 ← M[1] H
LB $3 ← M[4] H
LB $2 ← M[0] H
LB $2 ← M[10]
LB $2 ← M[15]
LB $2 ← M[8]

$0	
$1	110
$2	100
$3	140

Cache

V	tag	data
1	00	100
		110
0		
1	00	140
		150
0		

Misses: 2

Hits: 3

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Quiz

Which of the following statements are correct?

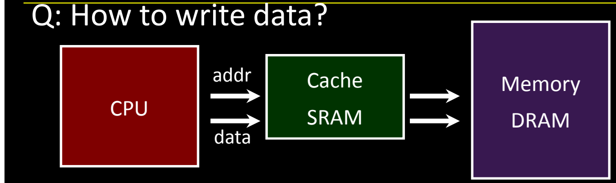
- LRU is used to select the line to be evicted in direct mapped cache.
- When using a global and local counter to implement LRU cache eviction scheme, the local counter of a slot is incremented when a new value is written to the slot.
- In modern computer architecture, bytes are the smallest units of data that we get out of/into memory in one read/write operation.
- The size of a word is always 32 bits (i.e. four bytes).
- Assume that (a) a cache has 8 lines, (b) each line has 2 words, and (c) each word is 32 bits. The cache has 64 bytes.
- The tag field of a cache line stores the address of a memory location.
- The valid bit of a cache line indicates whether the contents of the line correspond to values stored in the memory.
- A direct mapped cache has 4 cache lines. Each cache line consists of 2 words, and each word is one byte. The address bus consists of 6 bits. What is the number of bits for the tag, index and offset fields respectively?

Agenda

- Memory hierarchy
- Cache
- Managing cache
- Write on cache
- Cache conscious programming

Write Policies

Q: How to write data?



If data is already in the cache…

Write-Through

- writes go to main memory and cache

Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

Write Policies

- If it is not in the cache?
  - Allocate the line (put it in the cache)

  (write allocate policy)
  - Write it directly to memory without allocation in the cache

  (no write allocate policy)

write-through

A cache with a write-through and write-allocate policy

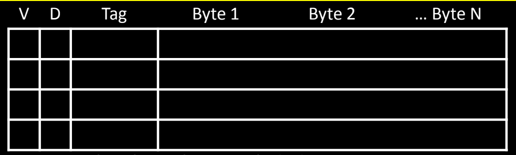
- reads an entire block (cache line) from memory on a cache miss
- writes only the updated item to memory for a store
- evictions do not need to write to memory

write-back

NOT write all stores immediately to memory

- Keep the most current copy in cache, and update memory when that data is ***evicted*** (write-back policy)

Write-Back Meta-Data



V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

Write-Back

Each miss reads a block

Each evicted dirty cache line writes a block

No write is needed if the cache line is not dirty

Performance Tradeoffs

In a cache hit, write-through is slower than write-back on writes.

In a cache miss, write-back might be slower than write-through if a dirty line has to be evicted.

Quiz

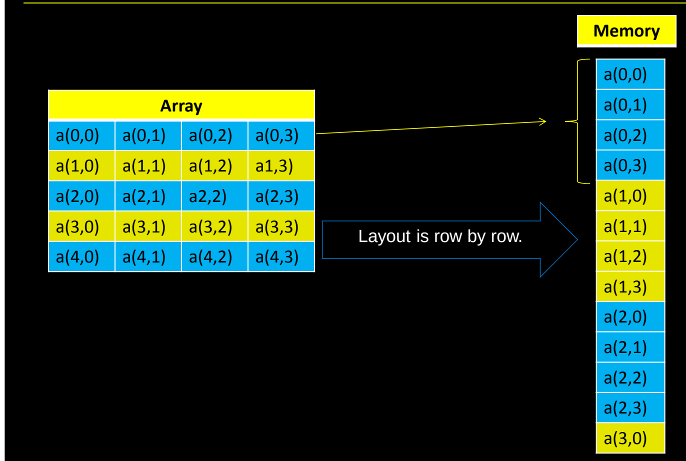
Which of the following statements are correct?

- The write-through policy modifies both the contents of the cache and the memory in a write operation.
- If a write operation has a cache hit, the write-back policy would delay the write operation on the cache until the cache line is evicted.
- When a cache line is evicted, the write-back policy always writes a complete cache line to the memory.
- Compared with the write-through policy, the write-back policy always gives better performance to the cache.

Agenda

- Memory hierarchy
- Cache
- Managing cache
- Write on cache
- Cache conscious programming

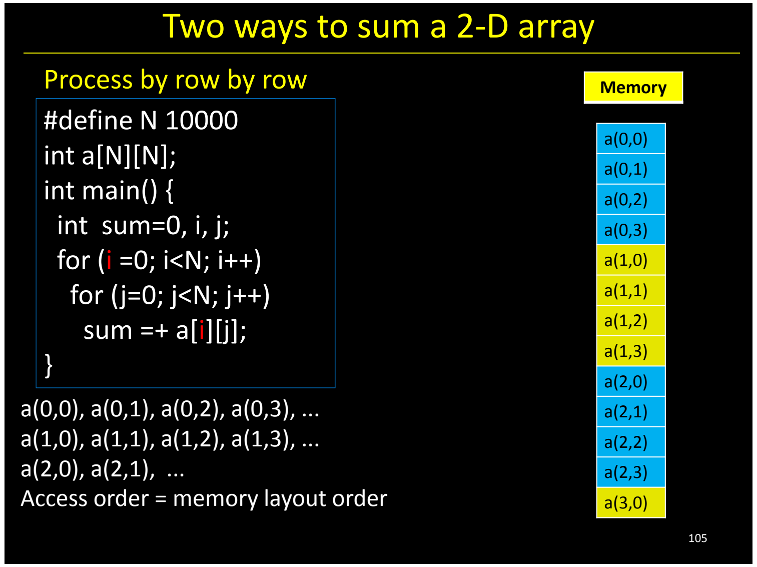
C's Array Layout in Memory



Array			
a(0,0)	a(0,1)	a(0,2)	a(0,3)
a(1,0)	a(1,1)	a(1,2)	a1,3)
a(2,0)	a(2,1)	a2,2)	a(2,3)
a(3,0)	a(3,1)	a(3,2)	a(3,3)
a(4,0)	a(4,1)	a(4,2)	a(4,3)

Layout is row by row.

Memory
a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)



Two ways to sum a 2-D array

Process by row by row

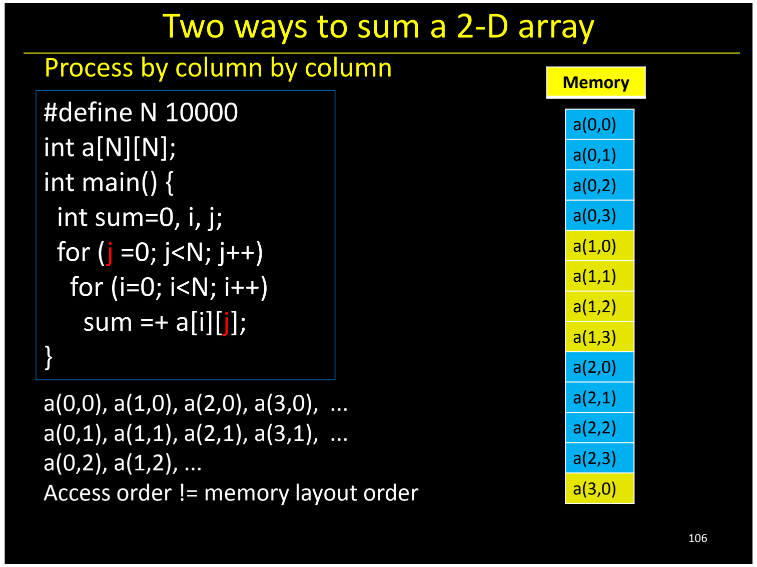
```
#define N 10000
int a[N][N];
int main() {
  int  sum=0, i, j;
  for (i =0; i<N; i++)
    for (j=0; j<N; j++)
      sum =+ a[i][j];
}
```

a(0,0), a(0,1), a(0,2), a(0,3), …
a(1,0), a(1,1), a(1,2), a(1,3), …
a(2,0), a(2,1), …
Access order = memory layout order

Memory

a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

105



Two ways to sum a 2-D array

Process by column by column

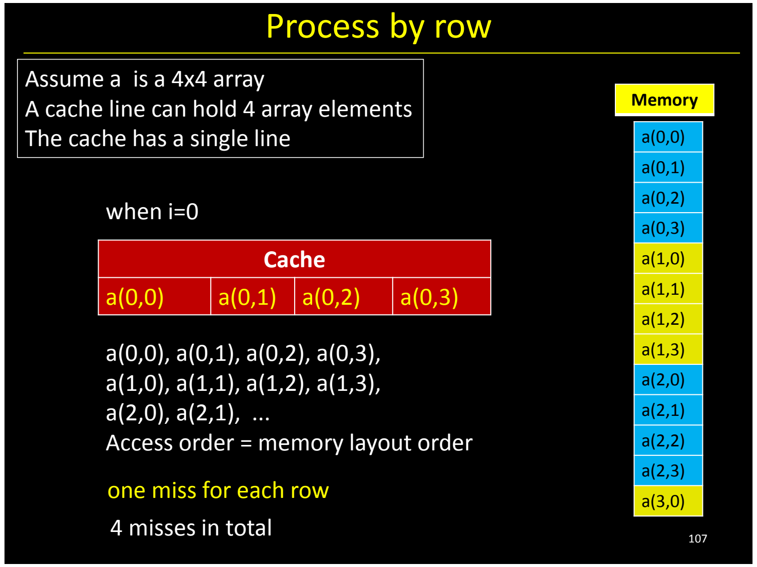
```
#define N 10000
int a[N][N];
int main() {
  int sum=0, i, j;
  for (j =0; j<N; j++)
    for (i=0; i<N; i++)
      sum =+ a[i][j];
}
```

a(0,0), a(1,0), a(2,0), a(3,0), …
a(0,1), a(1,1), a(2,1), a(3,1), …
a(0,2), a(1,2), …
Access order != memory layout order

Memory

a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

106



Process by row

Assume a is a 4x4 array
A cache line can hold 4 array elements
The cache has a single line

when i=0

Cache			
a(0,0)	a(0,1)	a(0,2)	a(0,3)

a(0,0), a(0,1), a(0,2), a(0,3),
a(1,0), a(1,1), a(1,2), a(1,3),
a(2,0), a(2,1), …
Access order = memory layout order

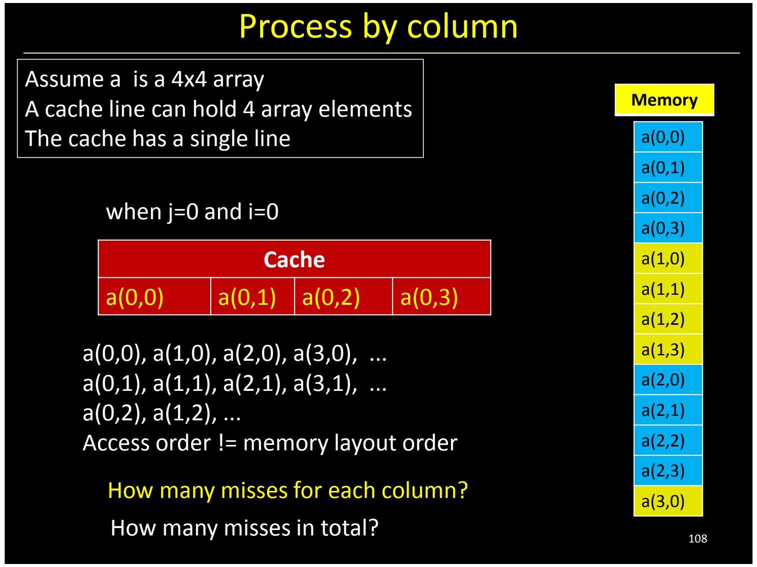
one miss for each row

4 misses in total

Memory

a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

107



Process by column

Assume a is a 4x4 array
A cache line can hold 4 array elements
The cache has a single line

when j=0 and i=0

Cache			
a(0,0)	a(0,1)	a(0,2)	a(0,3)

a(0,0), a(1,0), a(2,0), a(3,0), …
a(0,1), a(1,1), a(2,1), a(3,1), …
a(0,2), a(1,2), …
Access order != memory layout order

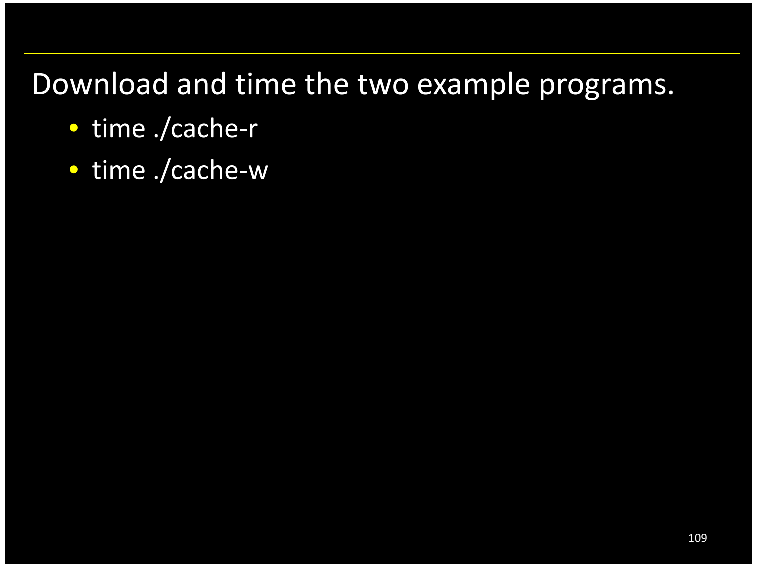
How many misses for each column?

How many misses in total?

Memory

a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

108



Download and time the two example programs.

- time ./cache-r
- time ./cache-w