

# Computer Systems

CS107

Cynthia Lee

Today's materials adapted from Kevin Webb  
at Swarthmore College

# Today's Topics

## TODAY'S LECTURE:

- › Caching

## ANNOUNCEMENTS:

- › **Assign6 & Assign7** due Friday!
  - 6 & 7 NO late days allowed!
  - **Submit early and often!** That way if a catastrophe happens at deadline time and you can't submit, at least you have some points from your earlier submissions!
- › Today's lecture about cache \*is\* fair game for final exam, some practice problems in lab this week (rest of lab will be final review)
- › Final exam practice exams to be posted tomorrow
- › I'll have extra office hours this week
- › Good luck everyone! <3 <3

# Cache Performance Goal

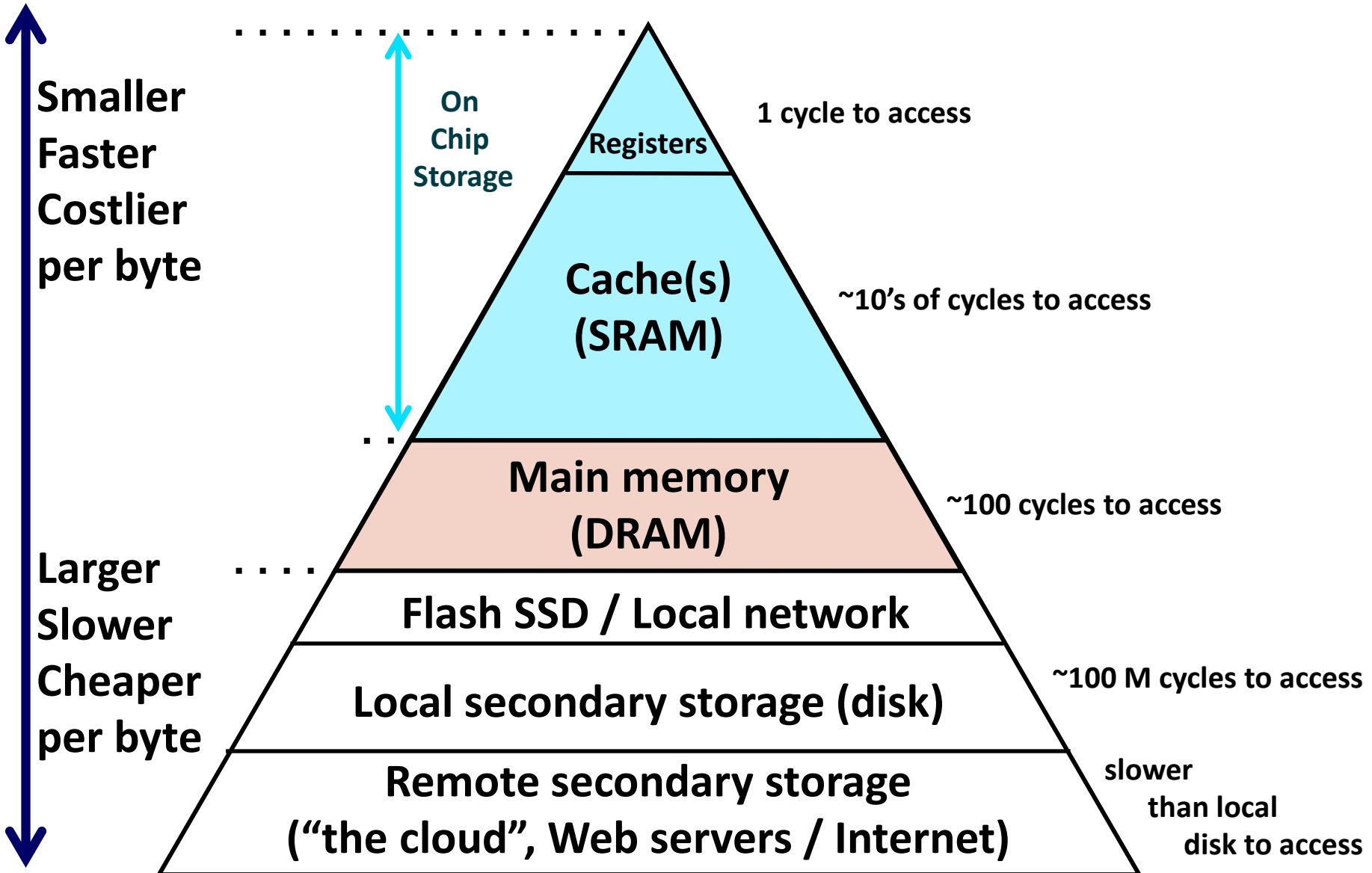
- Goal:
  - › We want to have the entirety of main memory available to the ALU to perform operations at the speed of register access
- Reality says: LOL / “We all want things”
- It’s partly the cost that is a barrier (smaller/faster memory is more expensive), but there are also physical limitations

Aside: But wait, why does everyone have to work in one building, what if there are smaller employers scattered throughout?

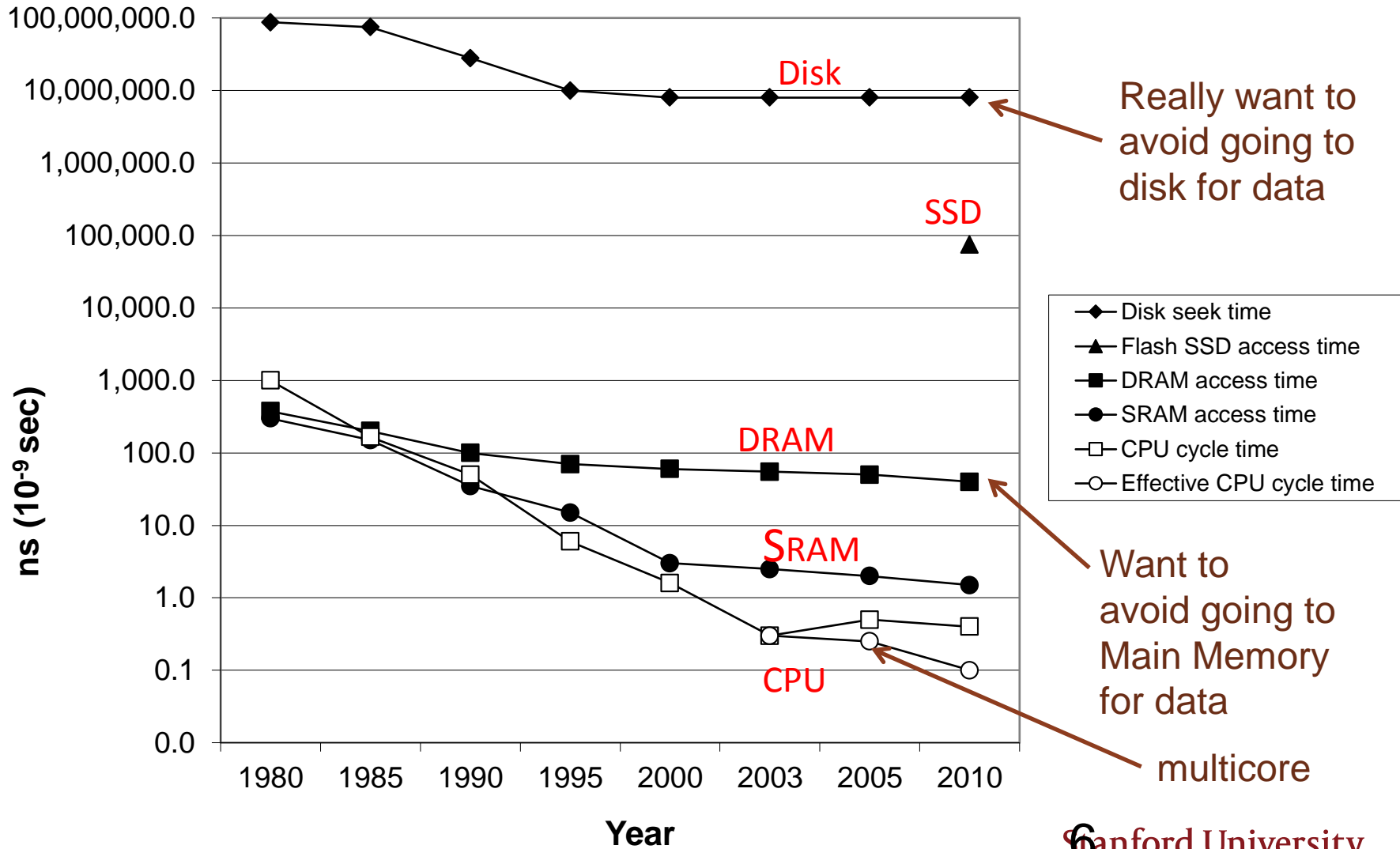
- That's essentially what distributed systems are (e.g. large datacenters)



# The Memory Hierarchy



# Growing gap between processor speed and memory access speed



# Recall

- A cache is a smaller, faster memory, that holds a subset of a larger (slower) memory
- We take advantage of locality (spatial and temporal) to keep data in cache as often as we can!
- When accessing memory, we first check cache to see if it has the data we're looking for.

## Breaking it down: why we miss...

### **COMPULSORY (COLD-START) MISS:**

- First time we use data, load it into cache.

### **CAPACITY MISS:**

- Cache is too small to store all the data we're using.

### **CONFLICT MISS:**

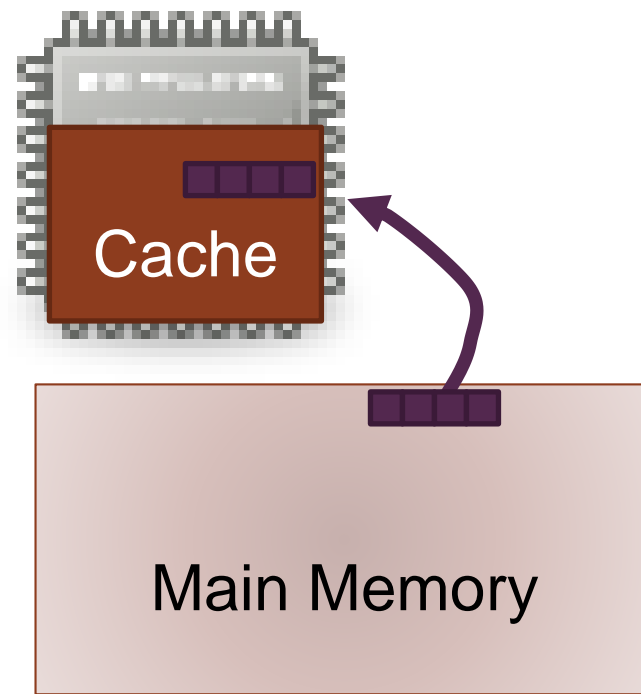
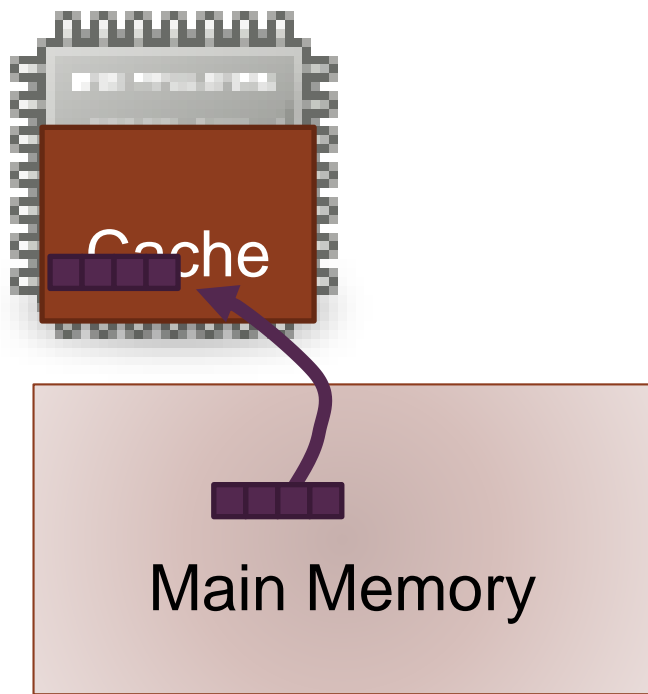
- To bring in new data to the cache, we evicted other data that we're still using.



# Cache Design

LOT'S OF CHARACTERISTICS TO CONSIDER:

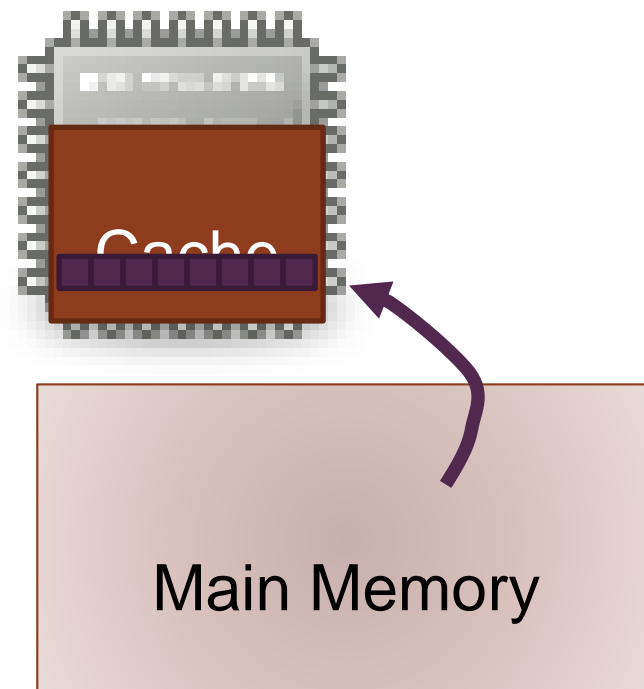
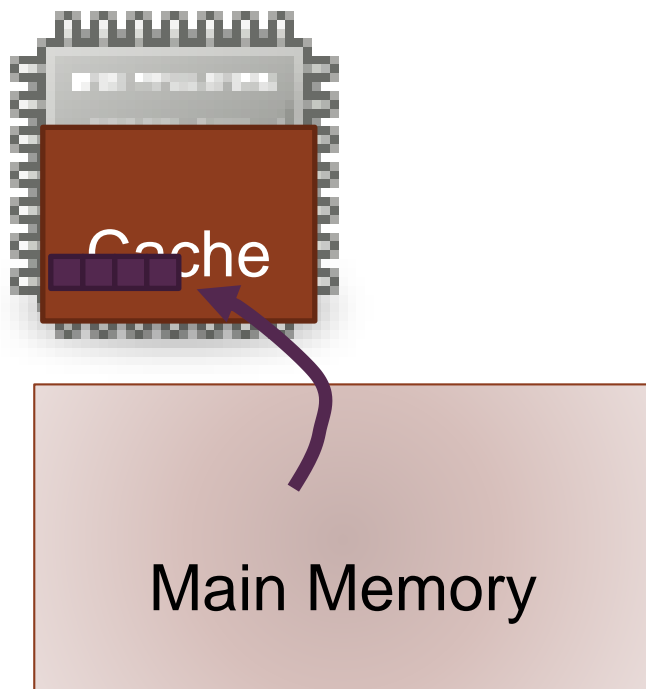
- Where should data be stored in the cache?



# Cache Design

LOT'S OF CHARACTERISTICS TO CONSIDER:

- Where should data be stored in the cache?
- What size data chunks should we store? (block size)



# Cache Design

LOT'S OF CHARACTERISTICS TO CONSIDER:

- Where should data be stored in the cache?
- What size data chunks should we store? (block size)

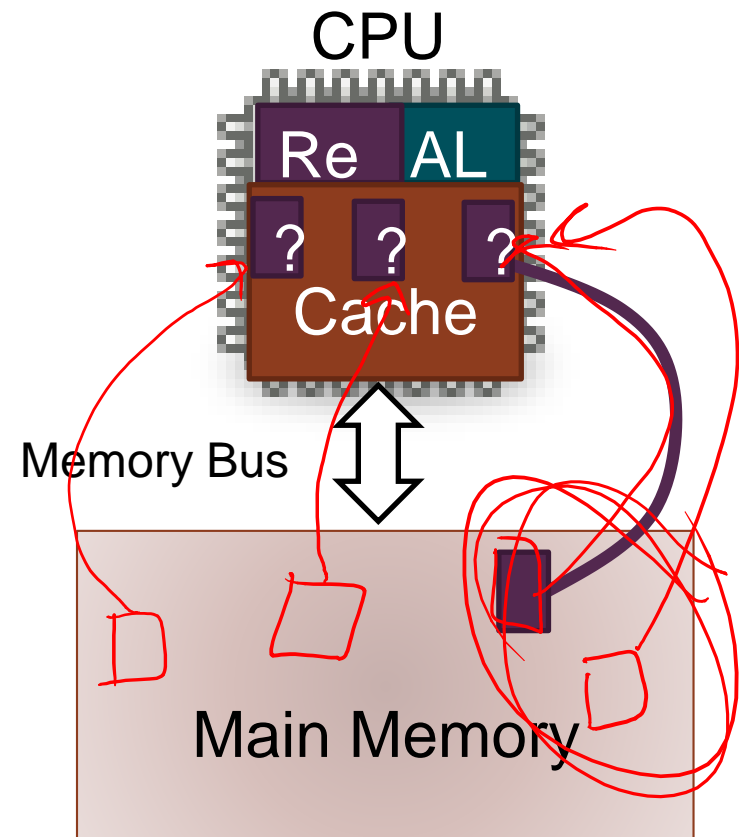
GOALS:

- Maximize hit rate
- Maximize (temporal & spatial) locality benefits
- Reduce cost/complexity of design

## Design discussion:

Suppose the CPU asks for data, it's not in cache. We need to move in into cache from memory. Where in the cache should it be allowed to go?

- A. In exactly one place.
- B. In a few places.
- C. In most places, but not all.
- D. Anywhere in the cache.



A. In exactly one place. (“Direct-mapped”)

- Every location in memory is directly mapped to one place in the cache. Easy to find data.

B. In a few places. (“Set associative”)

- A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

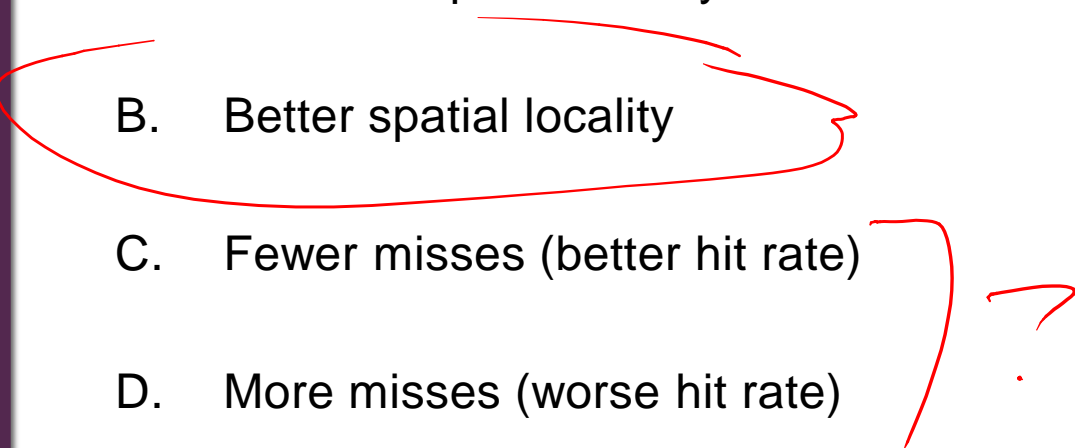
~~C. In most places, but not all.~~

D. Anywhere in the cache. (“Fully associative”)

- No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.

## Design discussion:

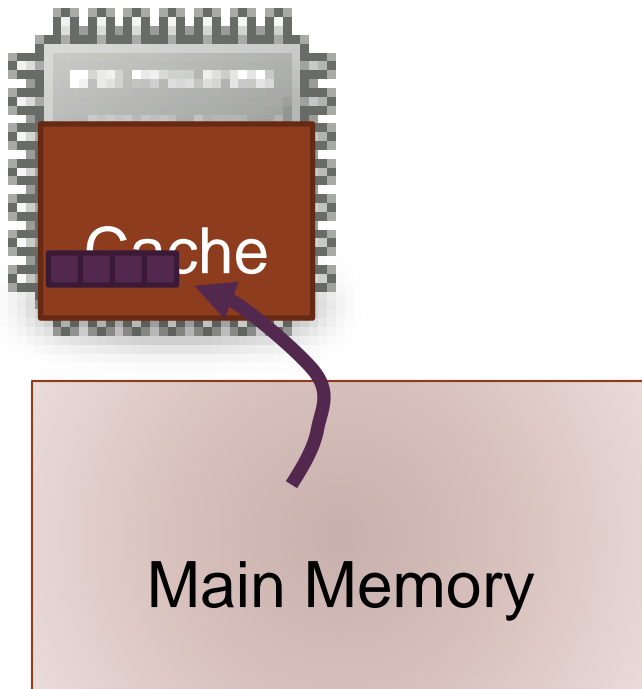
A larger *block size* (caching memory in larger chunks) is likely to exhibit...

- A. Better temporal locality
  - B. Better spatial locality
  - C. Fewer misses (better hit rate)
  - D. More misses (worse hit rate)
  - E. More than one of the above. (Which?)
- 

# Block Size Implications

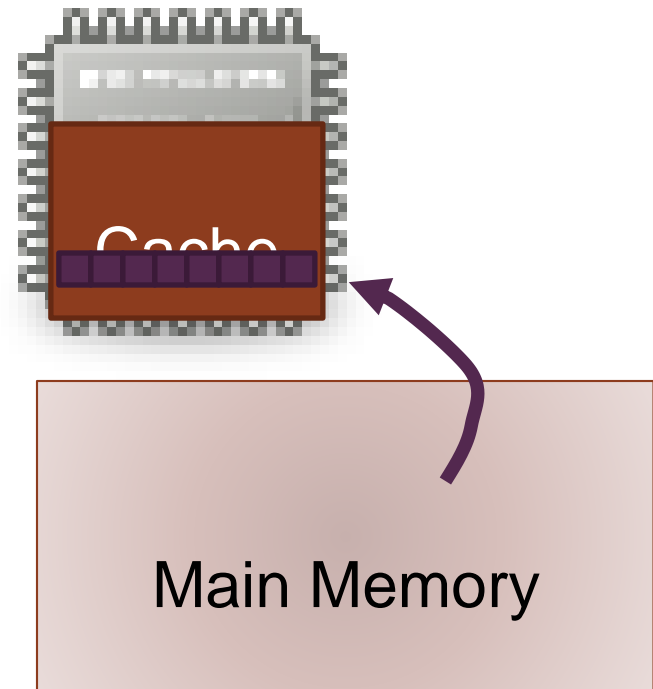
## SMALL BLOCKS

- Room for more blocks
- Fewer conflict misses



## LARGE BLOCKS

- Fewer trips to memory
- Longer transfer time
- Fewer cold-start misses



# Trade-offs

There is no single best design for all purposes!

**Common systems question:** which point in the design space should we choose?

Given a particular scenario:

- Analyze needs
- Choose design that fits the bill



# Real CPUs

Goals: general purpose processing

- balance needs of many use cases
- middle of the road: jack of all trades, master of none

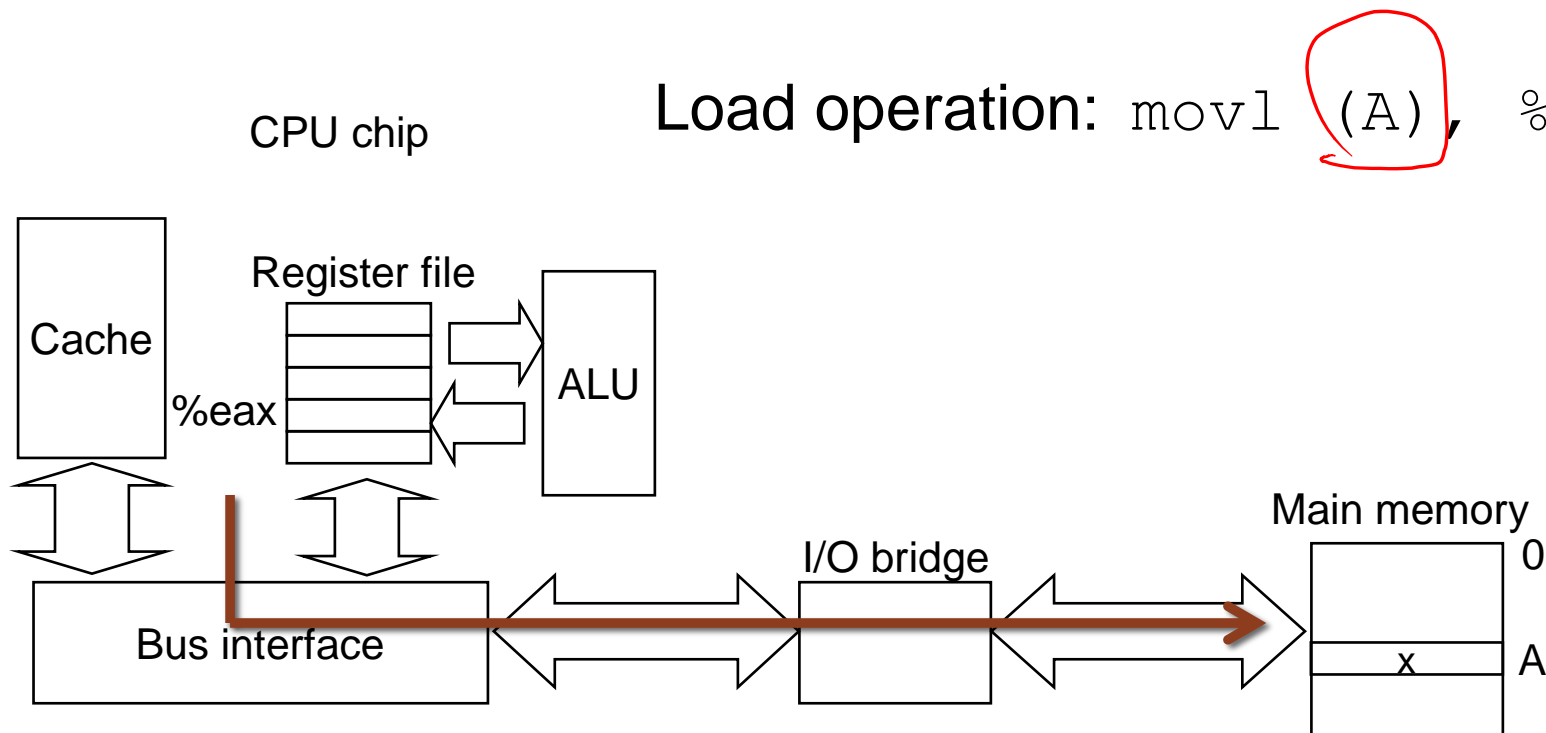
Typical: some associativity, medium size blocks:

- 8-way associative (memory in one of eight places)
- 16 or 32-byte blocks

## Recall: How Memory Read Works

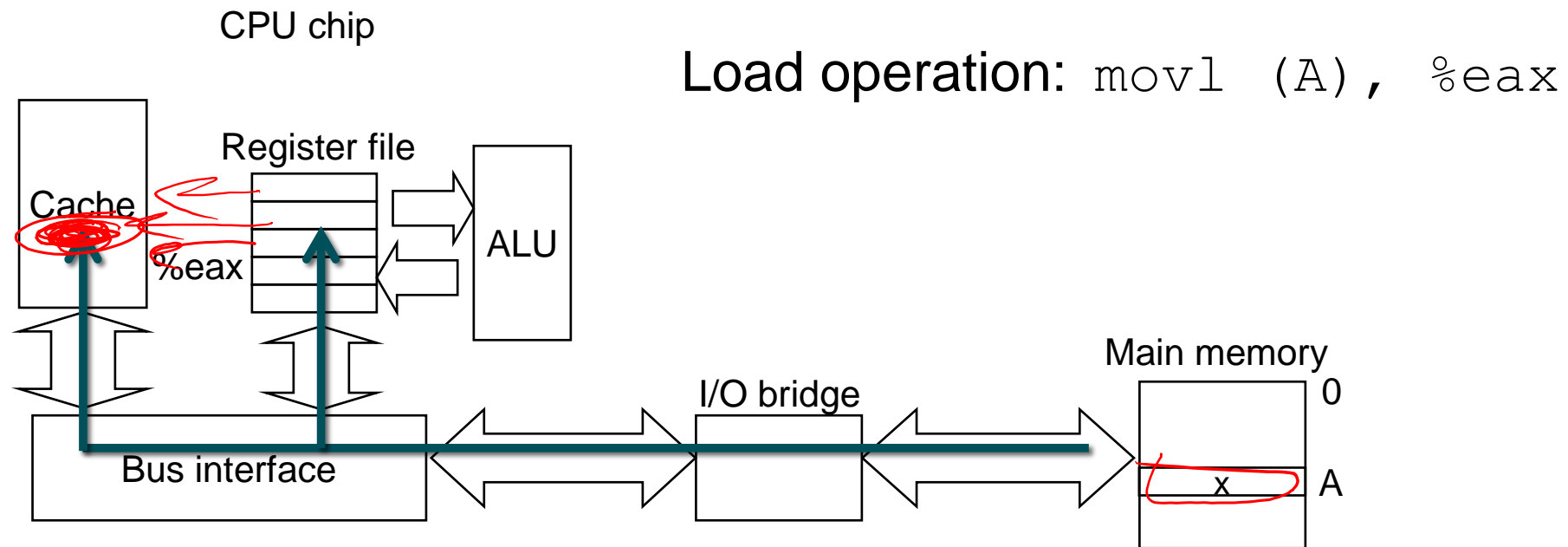
(1) CPU places address  $A$  on the memory bus.

Load operation: `movl (A), %eax`



## Recall: How Memory Read Works

- (1) CPU places address A on the memory bus.
- (2) Memory sends back the value



## Memory Address is the key to telling us a few things:

1. Is the block containing the byte(s) you want already in the cache?
2. If not, where should we put that block?
  - › Do we need to kick out (“evict”) another block?
3. Which byte(s) within the (multi-byte) block do you want?

**A. In exactly one place. (“Direct-mapped”)**

- **Every location in memory is directly mapped to one place in the cache. Easy to find data.**

**B. In a few places. (“Set associative”)**

- A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

~~C. In most places, but not all.~~

**D. Anywhere in the cache. (“Fully associative”)**

- No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.

# Direct-Mapped

One place data can be.

Example: let's assume some parameters:

- 1024 cache locations (every block mapped to one)
- Block size of 8 bytes

# Direct-Mapped

Metadata

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020				
1021				
1022				
1023				

# Cache Metadata

Valid bit: is the entry valid?

- If set: data is correct, use it if we 'hit' in cache
- If not set: ignore 'hits', the data is garbage

Dirty bit: has the data been written?

- Used by "write-back" caches
- If set, need to update memory before eviction



# Direct-Mapped

## ADDRESS DIVISION:

- Identify byte in block
  - › How many bits?
  
- Identify which row (line)
  - › How many bits?

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020				
1021				
1022				
1023				

# Direct-Mapped

## ADDRESS DIVISION:

- Identify byte in block
  - › How many bits? 3
  
- Identify which row (line)
  - › How many bits? 10

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020				
1021				
1022				
1023				

# Direct-Mapped

ADDRESS DIVISION (64-BIT ADDRESS):

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
	1111100000	010

1111 00000 010

Index:

Which line (row) should we check?

Where could data be?

Line

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020				
1021				
1022				
1023				

# Direct-Mapped

ADDRESS DIVISION:

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
	4	



Index:

Which line (row) should we check?

Where could data be?

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020				
1021				
1022				
1023				

# Direct-Mapped

ADDRESS DIVISION:

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	



In parallel, check:

Tag:

Does the cache hold the data we're looking for, or some other block?

Valid bit:

If entry is not valid, don't trust garbage in that line (row).

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4	1		4217	
...			...	
1020				
1021				
1022				
1023				

If tag doesn't match,  
or line is invalid, it's a  
miss!

# Direct-Mapped

ADDRESS DIVISION:

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	

Byte offset tells us which subset of block to retrieve.

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4	1		4217	
...			...	
1020				
1021				
1022				
1023				

0 1 2 3 4 5 6 7

# Direct-Mapped

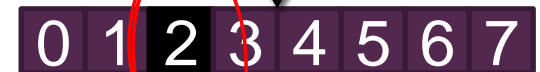
ADDRESS DIVISION:

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	2 <i>010</i>



Byte offset tells us which subset of block to retrieve.

Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4	1		4217	
...			...	
1020				
1021				
1022				
1023				

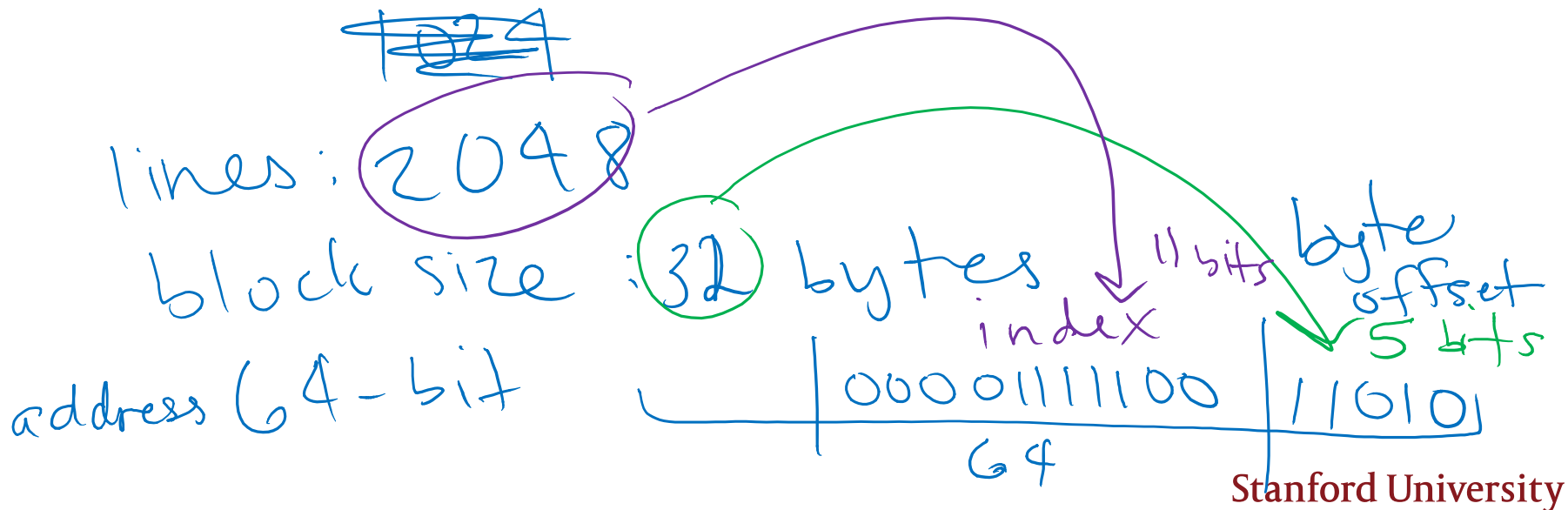


# Direct-Mapped Example

SUPPOSE OUR ADDRESSES ARE 16 BITS LONG.

OUR CACHE HAS 16 ENTRIES, BLOCK SIZE OF 16 BYTES

- 4 bits in address for the index
- 4 bits in address for byte offset
- Remaining bits (8): tag





# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT  
ADDRESS:

- 0110101100110100

STEP 1:

- Partition address into tag, index, offset

Line	V	D	Tag	Data (16 Bytes)
0				
1				
2				
3				
4				
5				
...				
15				

# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT  
ADDRESS:

- 01101011 0011 0100

STEP 1:

- Partition address into tag, index, offset

Line	V	D	Tag	Data (16 Bytes)
0				
1				
2				
3				
4				
5				
...				
15				

# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT  
ADDRESS:

- 01101011 0011 0100

STEP 2:

- Use index to find line (row)
- 0011 -> 3



Line	V	D	Tag	Data (16 Bytes)
0				
1				
2				
3				
4				
5				
...				
15				

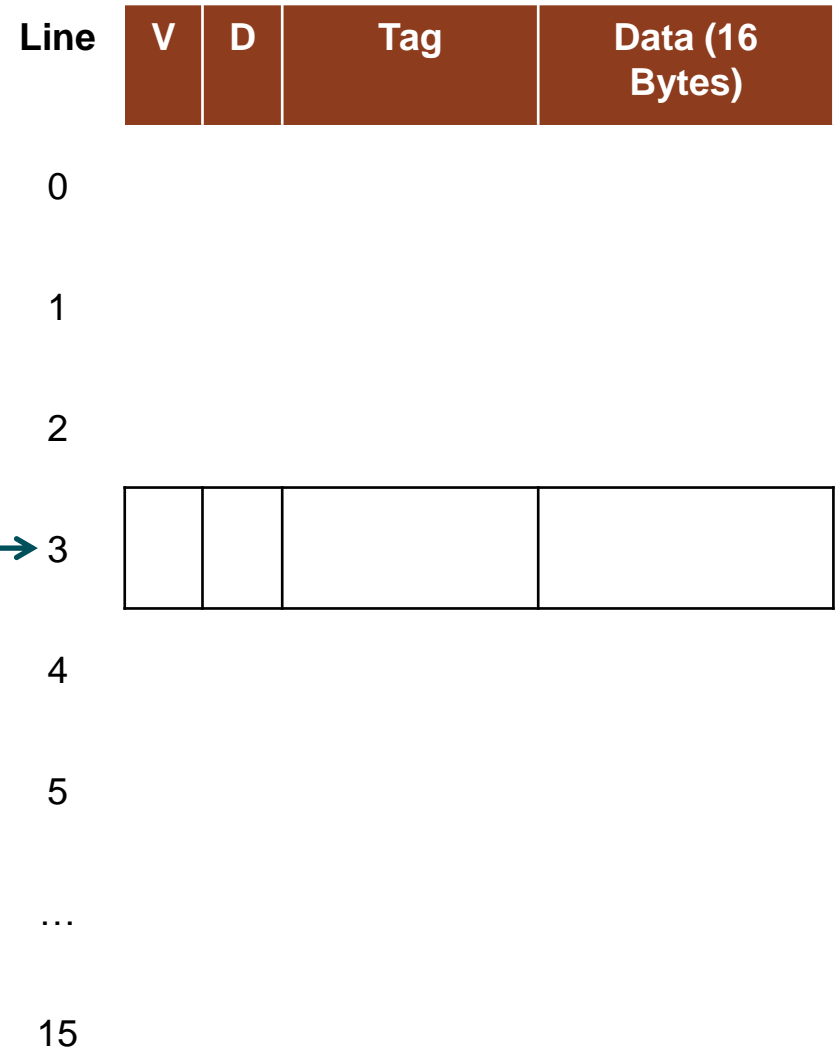
# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT ADDRESS:

- 01101011 0011 0100

STEP 2:

- Use index to find line (row)
- 0011 -> 3



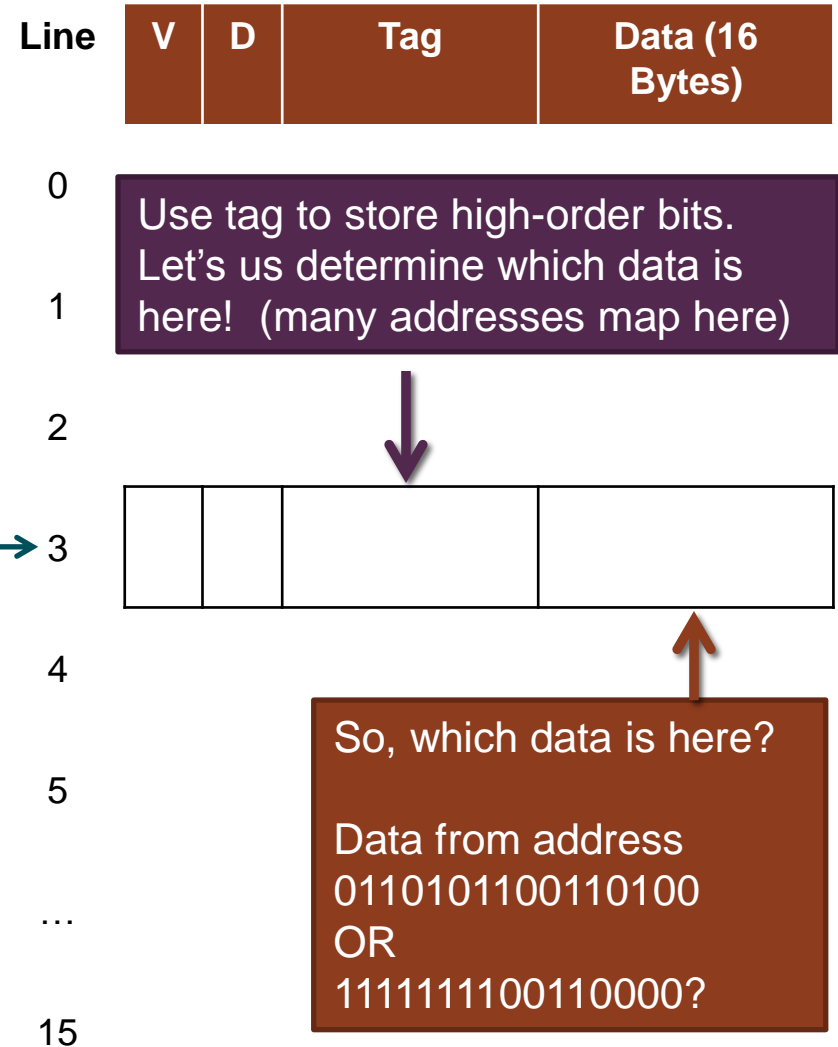
# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT ADDRESS:

- 01101011 0011 0100

NOTE:

- ANY address with 0011 (3) as the middle four index bits will map to this cache line.
- e.g. 11111111 0011 0000



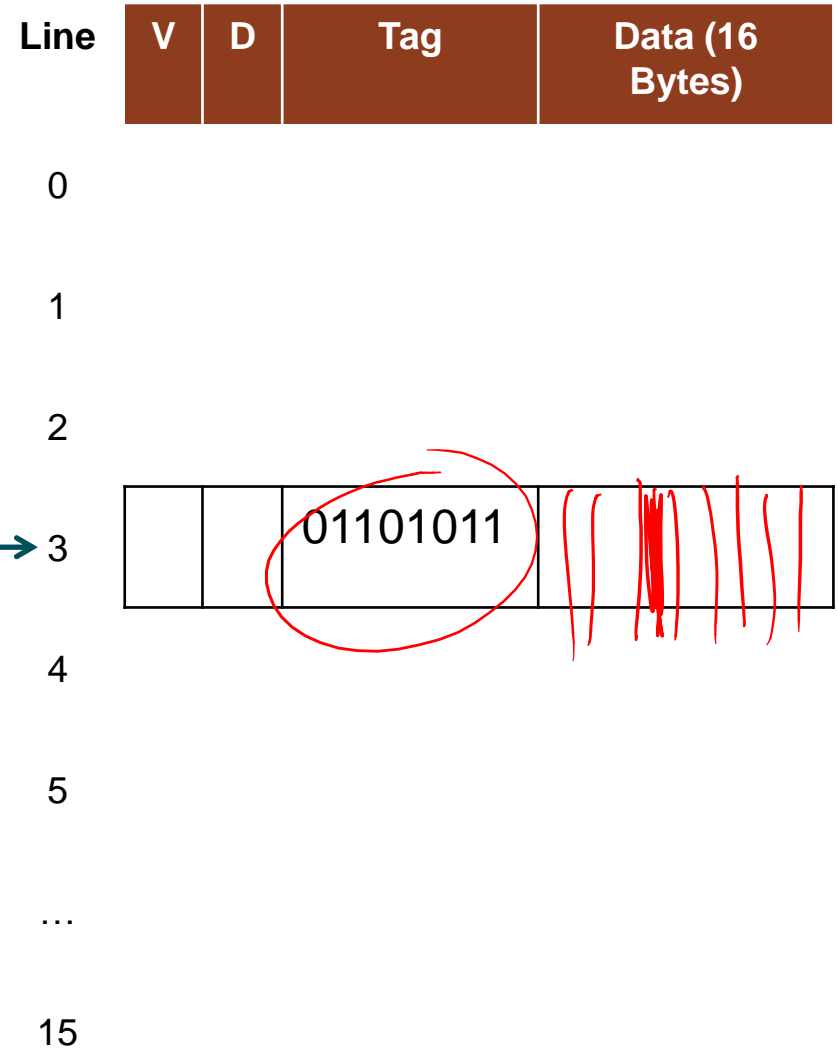
# Direct-Mapped Example

LET'S SAY WE ACCESS MEMORY AT ADDRESS:

- 01101011 0011 0100

STEP 3:

- Check the tag
- Is it 01101011 (hit)?
- Something else (miss)?
- (Must also ensure valid)



# Extra Slides

MORE PRACTICE

We didn't cover these in lecture since everyone seemed overwhelmed by heap allocator work, but the sample problems covered might be helpful review for the final exam (no new concepts here, just new practice).

# Extra Example 1



# Eviction

If we don't find what we're looking for (miss), we need to bring in the data from memory.

Make room by kicking something out.

- If line to be evicted is dirty, write it to memory first.

Another important systems distinction:

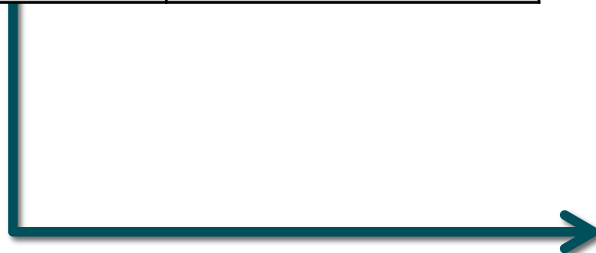
- Mechanism: An ability or feature of the system. What you can do.
- Policy: Governs the decisions making for using the mechanism. What you should do.

# Eviction: Direct-Mapped

ADDRESS DIVISION:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	

Find line:



Tag doesn't match, bring in from memory.

If dirty, write back first!

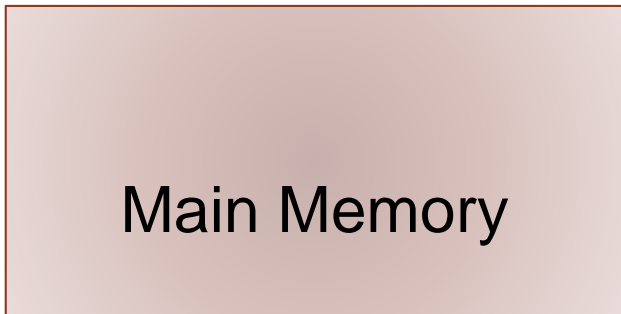
Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020	1	0	1323	57883
1021				
1022				
1023				

# Eviction: Direct-Mapped

ADDRESS DIVISION:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	

1. Send address to read main memory.



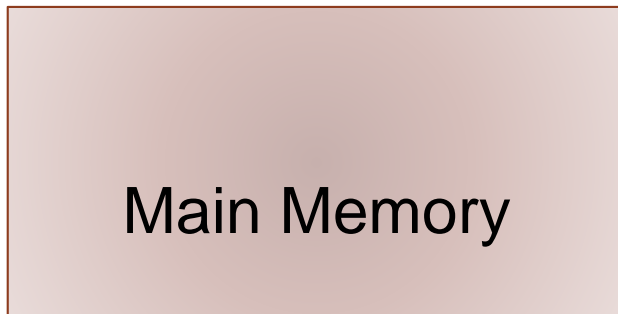
Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020	1	0	1323	57883
1021				
1022				
1023				

# Eviction: Direct-Mapped

ADDRESS DIVISION:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	

1. Send address to read main memory.



Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
...			...	
1020	1	0	3941	92
1021				
1022				
1023				

2. Copy data from memory.  
Update tag.

# Extra Example 2

Your turn: Suppose we had 8-bit addresses, a cache with 8 lines, and a block size of 4 bytes.

How many bits would we use for:

- Tag?
- Index?
- Offset?

Your turn: Suppose we had 8-bit addresses, a cache with 8 lines, and a block size of 4 bytes.

How many bits would we use for:

- Tag?  $8 - 3 - 2 = 3$  bits
- Index? 3 bits
- Offset? 2 bits

# Extra Example 3



# How would the cache change if we performed the following memory operations?

Memory  
address



READ 01000100 (VALUE: 5)

READ 11100010 (VALUE: 17)

WRITE 01110000 (VALUE: 7)

READ 10101010 (VALUE: 12)

WRITE 01101100 (VALUE: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	011	9
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# How would the cache change if we performed the following memory operations?

Memory address



READ 01000100 (VALUE: 5)  
READ 11100010 (VALUE: 17)  
WRITE 01110000 (VALUE: 7)  
READ 10101010 (VALUE: 12)  
WRITE 01101100 (VALUE: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	<del>011</del> 010	9 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# How would the cache change if we performed the following memory operations?

Memory address



READ 01000100 (VALUE: 5)  
READ 11100010 (VALUE: 17)  
WRITE 01110000 (VALUE: 7)  
READ 10101010 (VALUE: 12)  
WRITE 01101100 (VALUE: 2)

No change necessary.

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	<del>011</del> 010	9 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# How would the cache change if we performed the following memory operations?

Memory address



- READ 01000100 (VALUE: 5)
- READ 11100010 (VALUE: 17)
- WRITE 01110000 (VALUE: 7)
- READ 10101010 (VALUE: 12)
- WRITE 01101100 (VALUE: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	<del>011</del> 010	9 5
2	0	0	101	15
3	1	1	001	8
4	1	0 1	011	4 7
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# How would the cache change if we performed the following memory operations?

Memory address



- READ 01000100 (VALUE: 5)
- READ 11100010 (VALUE: 17)
- WRITE 01110000 (VALUE: 7)
- READ 10101010 (VALUE: 12)
- WRITE 01101100 (VALUE: 2)

Note: tag happened to match, but line was invalid.

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	<del>011</del> 010	9 5
2	<del>0</del> 1	0	<del>101</del> 101	15 12
3	1	1	001	8
4	1	<del>0</del> 1	011	4 7
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# How would the cache change if we performed the following memory operations?

Memory address



READ 01000100 (VALUE: 5)  
 READ 11100010 (VALUE: 17)  
 WRITE 01110000 (VALUE: 7)  
 READ 10101010 (VALUE: 12)  
WRITE 01101100 (VALUE: 2)

1. Write dirty line to memory.
2. Load new value, set it to 2, mark it dirty (write).

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	<del>011</del> 010	9 5
2	<del>0</del> 1	0	<del>101</del> 101	15 12
3	1	<del>1</del> 1	<del>001</del> 011	8 2
4	1	<del>0</del> 1	011	4 7
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

# Extra Example 4

# Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

Example: 2D array accesses

```
for(i=0; i < N; i++) {  
    for(j=0; j < M; j++) {  
        sum += arr[i][j];  
    }  
}
```

```
for(j=0; j < M; j++) {  
    for(i=0; i < N; i++) {  
        sum += arr[i][j];  
    }  
}
```

- Algorithmically, both  $O(N * M)$
- Is one faster than the other?



# Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

Example: 2D array accesses

```
for(i=0; i < N; i++) {  
    for(j=0; j < M; j++) {  
        sum += arr[i][j];  
    }  
}
```

```
for(j=0; j < M; j++) {  
    for(i=0; i < N; i++) {  
        sum += arr[i][j];  
    }  
}
```

- Algorithmically, both  $O(N * M)$
- Is one faster than the other?
  - A. Left one is faster
  - B. Right one is faster
  - C. Same

# Cache Conscious Programming

The first (left) nested loop is more efficient if the cache block size is larger than a single array bucket (for arrays of basic C types, it will be).

```
for(i=0; i < N; i++) {  
  for(j=0; j < M; j++) {  
    sum += arr[i][j];  
  }  
}
```

```
for(j=0; j < M; j++) {  
  for(i=0; i < N; i++) {  
    sum += arr[i][j];  
  }  
}
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
.																
.																
.																

1									...
2									
3									
4									
.									
.									
.									

(ex) 1 miss every 4 buckets vs. 1 miss every bucket