CS 31: Introduction to Computer Systems

17-18: Caching

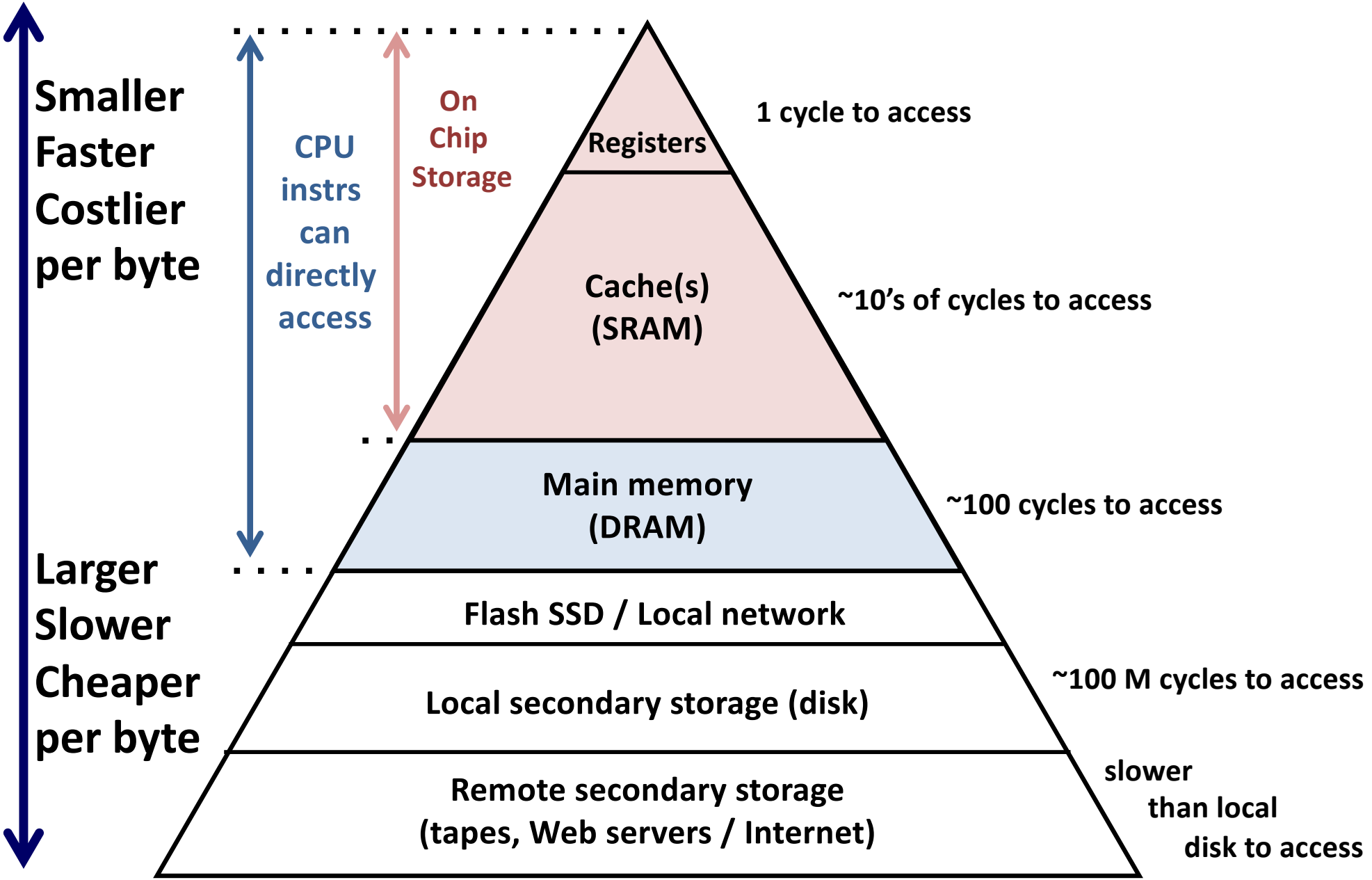
April 2-7 2020

Reading Quiz

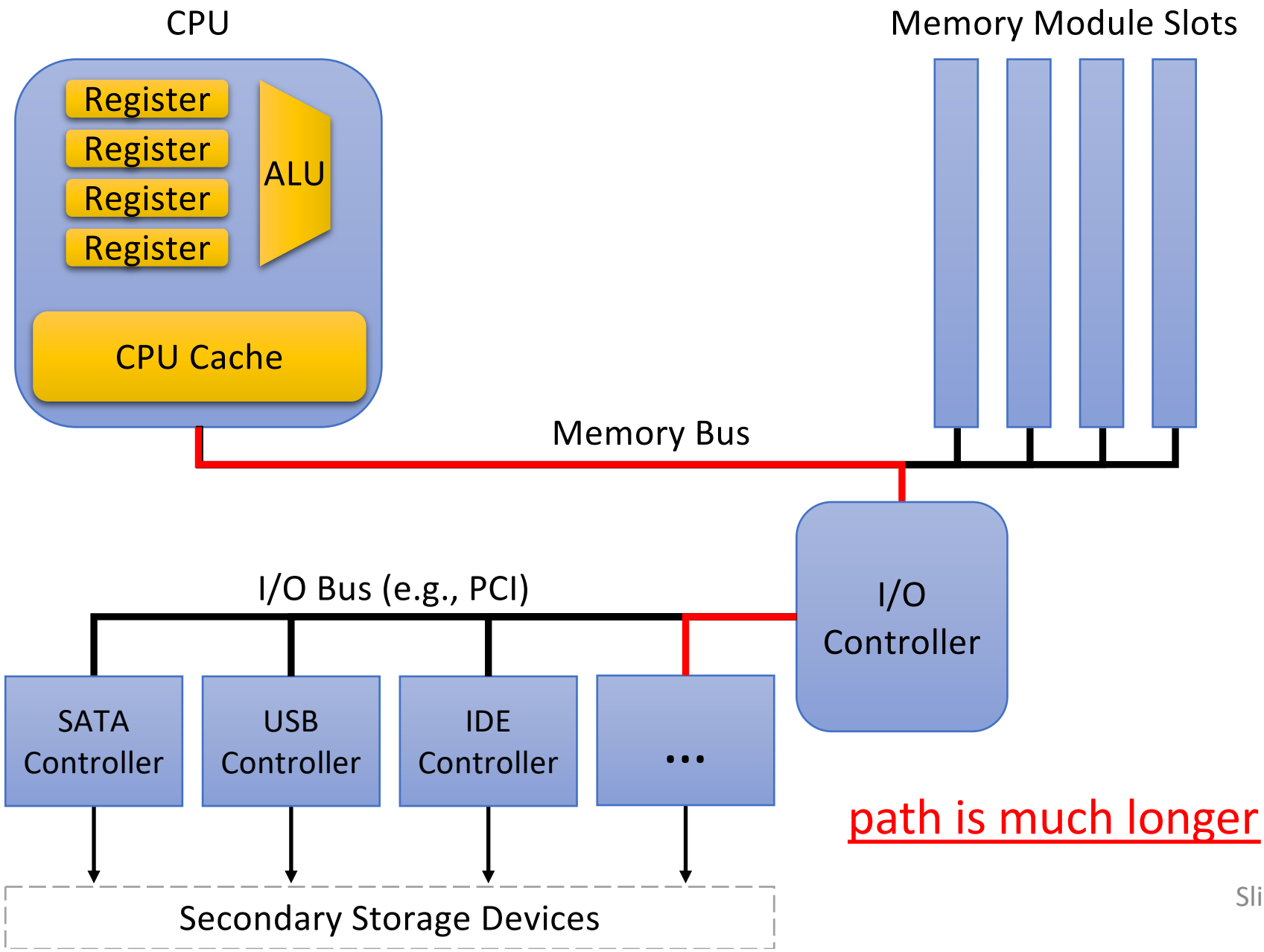
Abstraction Goal

- Reality: There is no one type of memory to rule them all!
- Abstraction: hide the complex/undesirable details of reality.
- Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.

Last class: The Memory Hierarchy



Last class: Memory Regions



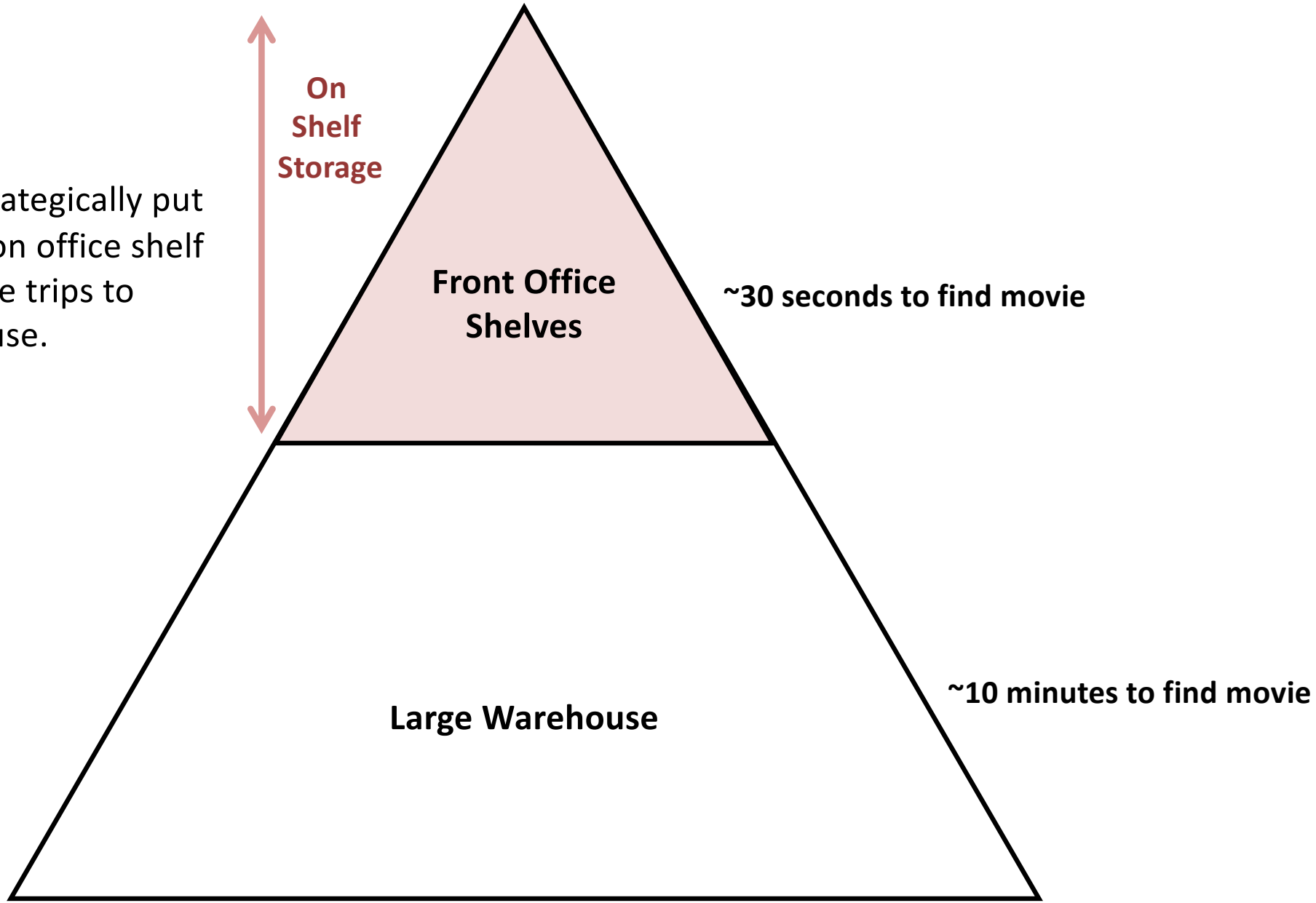
path is much longer

Last class: Caches, Locality

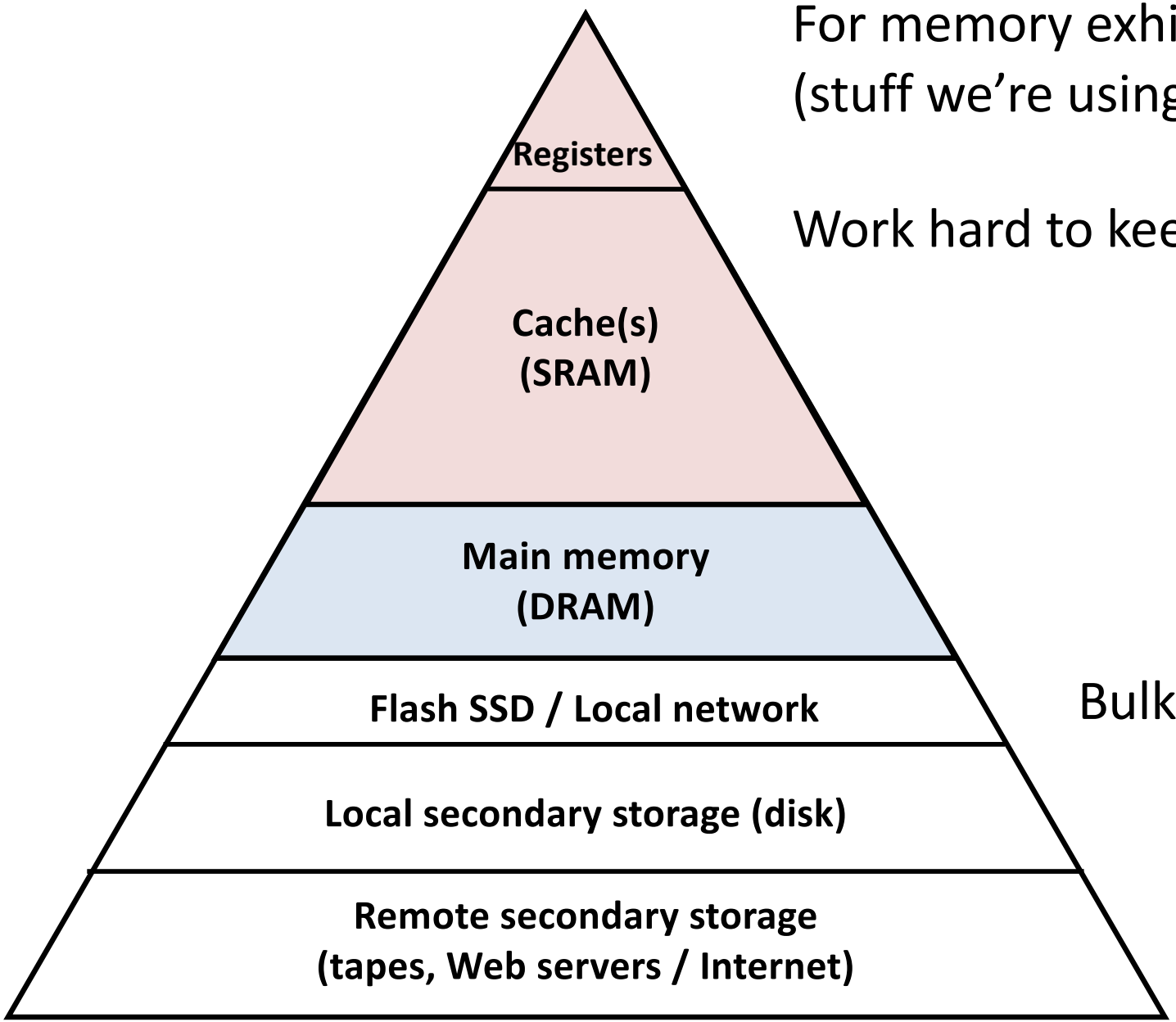
- A cache is a smaller, faster memory, that holds a subset of a larger (slower) memory
- We take advantage of locality to keep data in cache as often as we can!
- When accessing memory, we check cache to see if it has the data we're looking for.

The Video Store Hierarchy

Goal: strategically put movies on office shelf to reduce trips to warehouse.



Big Picture



For memory exhibiting locality (stuff we're using / likely to use):

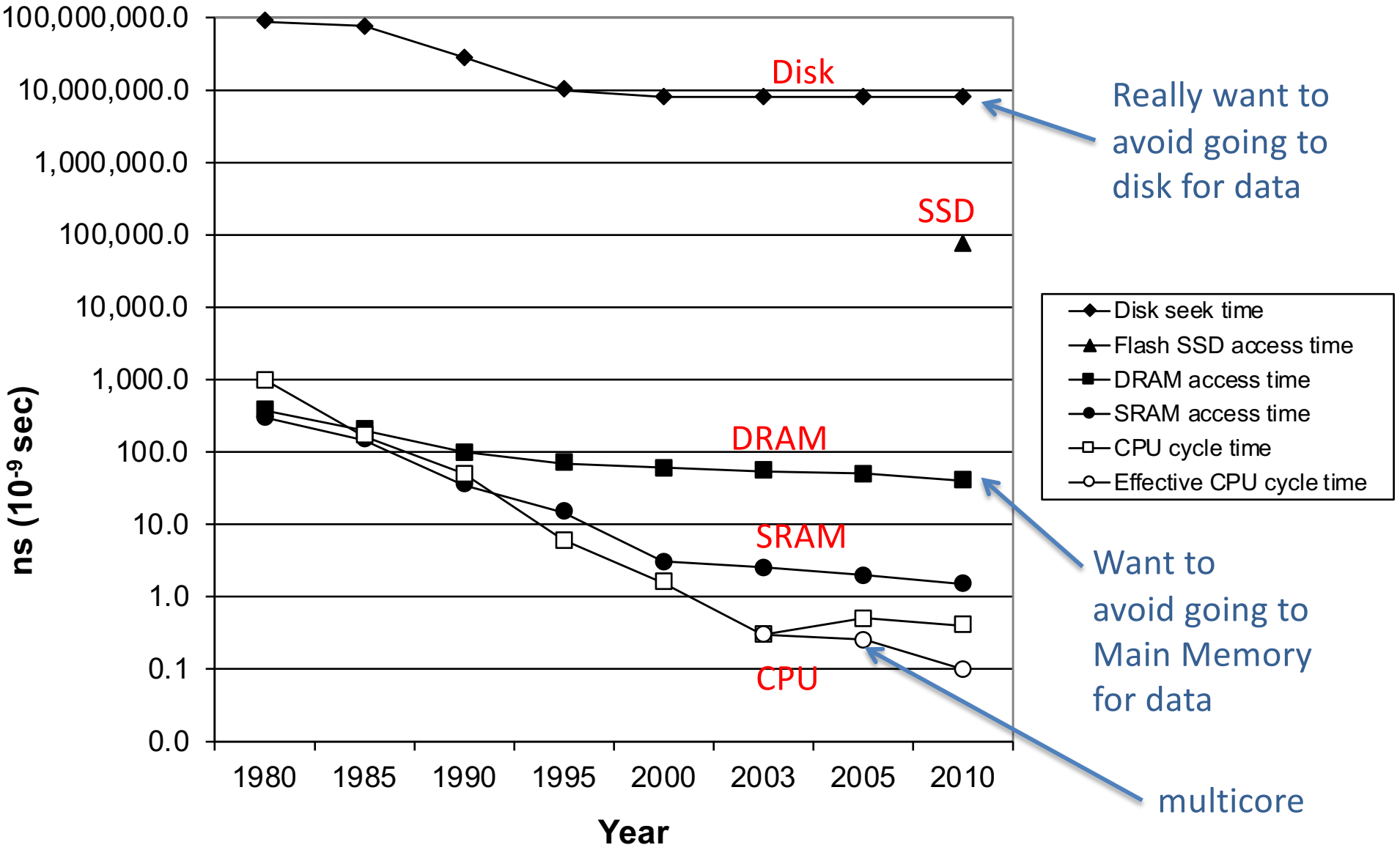
Work hard to keep them up here!

Bulk storage down here.

Move this up on demand.

Data Access Time over Years

Over time, gap widens between DRAM, disk, and CPU speeds.

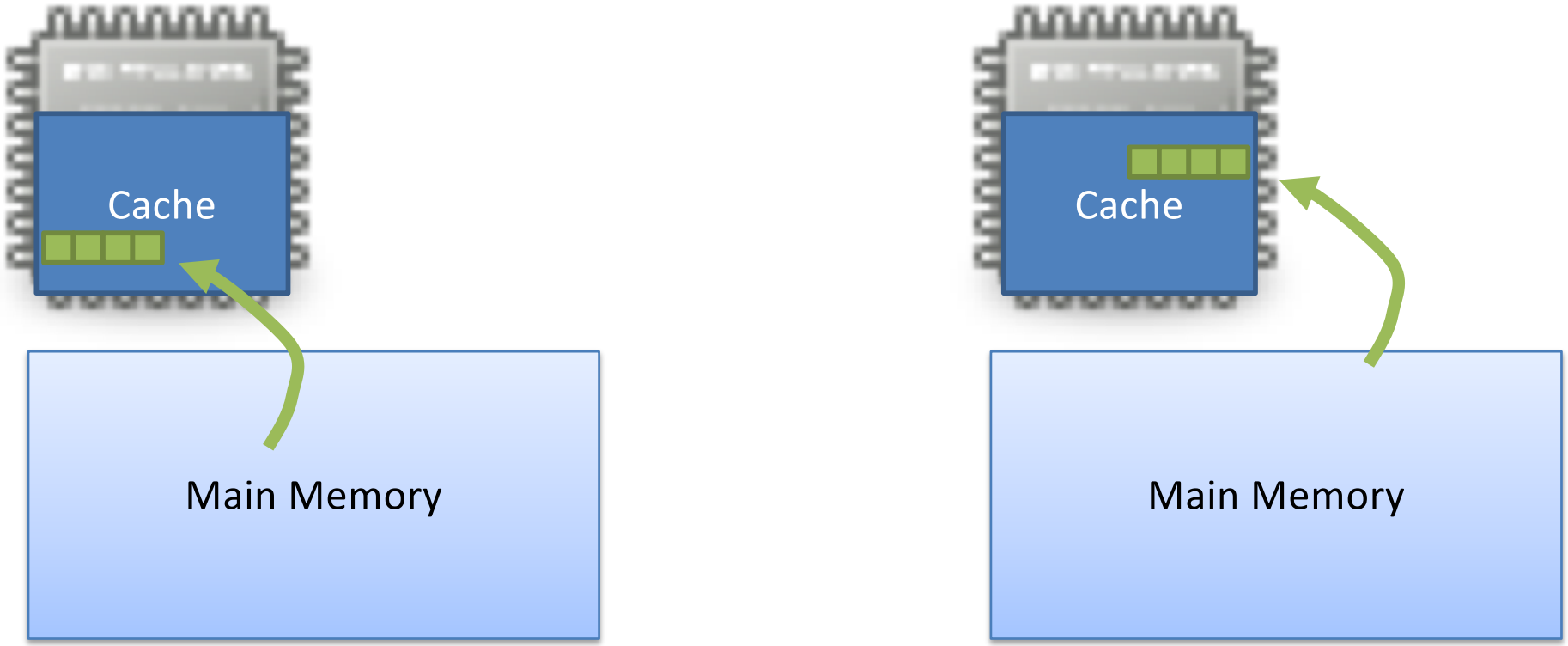


Why we miss…

- 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.

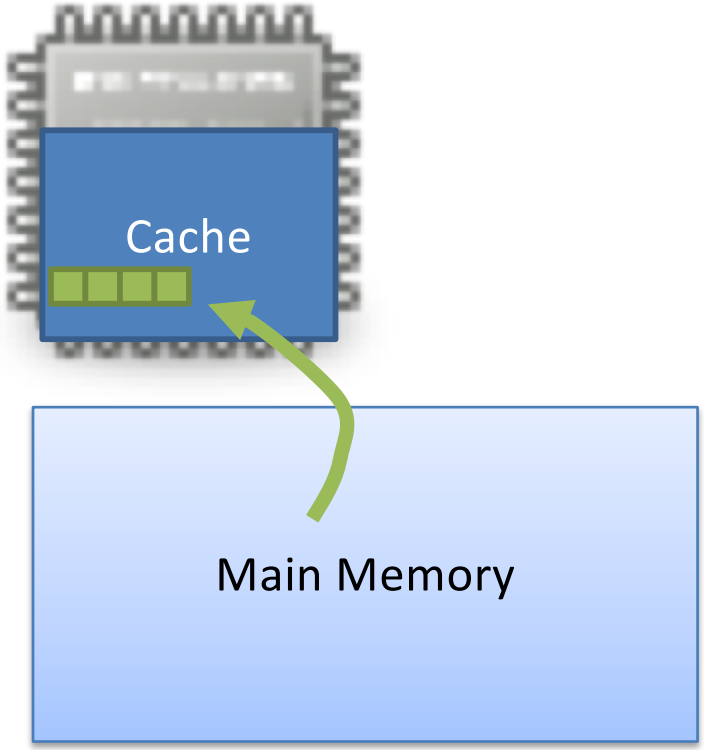
Cache Design

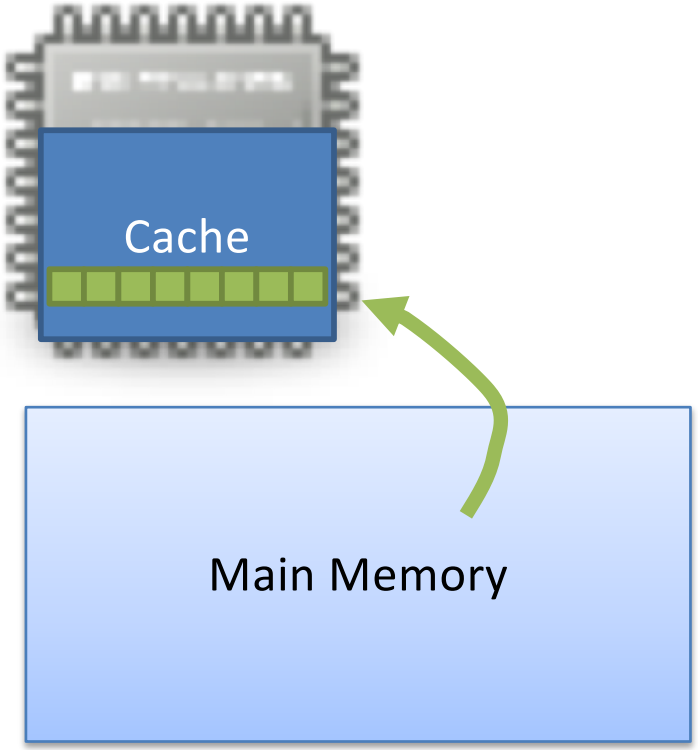
- Lot's of characteristics to consider:
  - Where should data be stored in the cache?



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

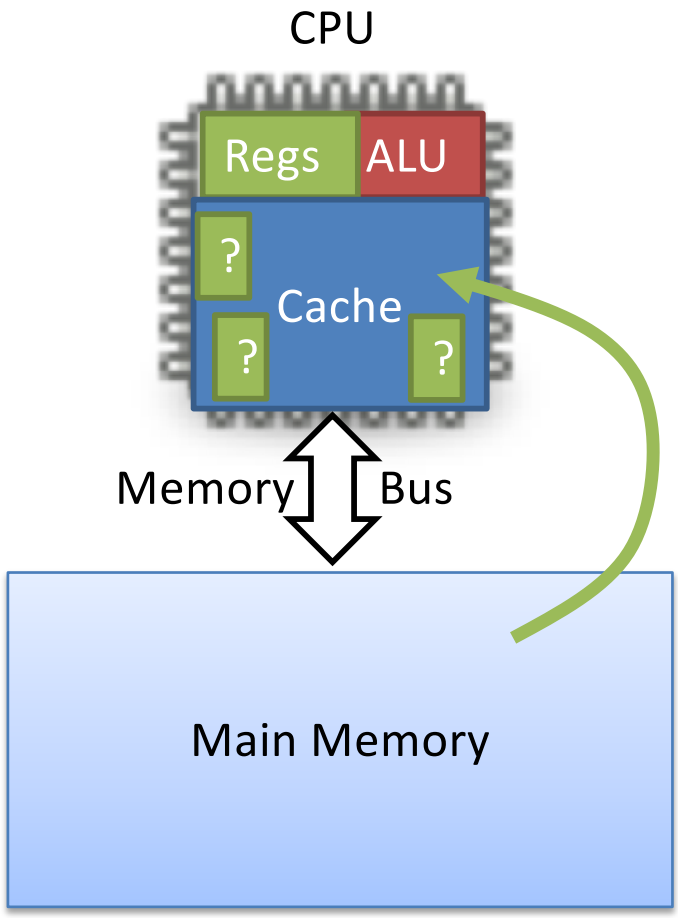
Suppose the CPU asks for data, it's not in cache. We need to move in into cache from memory. Where in the cache should it be allowed to go? trade-offs?

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.

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?)

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

A. Better temporal locality (does not change how freq. we use a block)

B. Better spatial locality

C. Fewer misses (better hit rate)

D. More misses (worse hit rate)

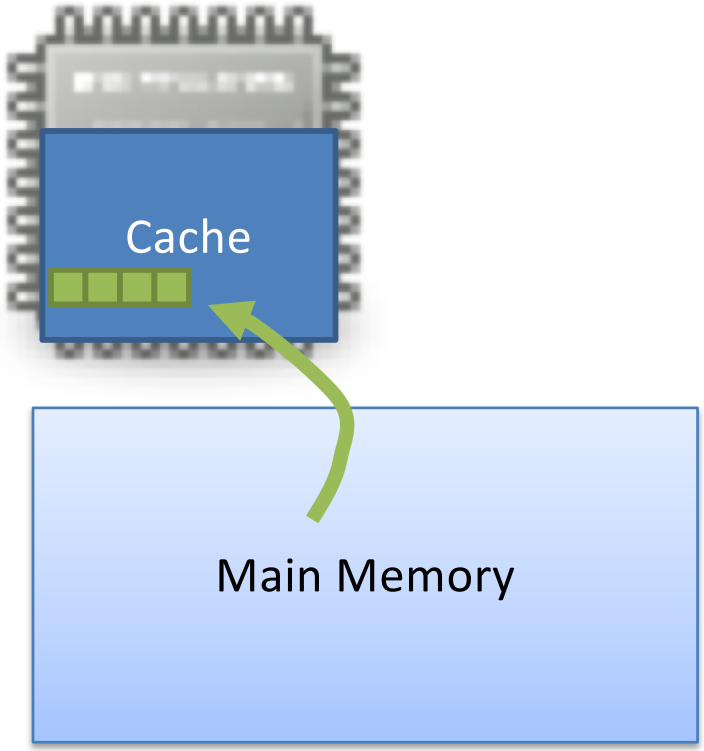
hard to make a determination

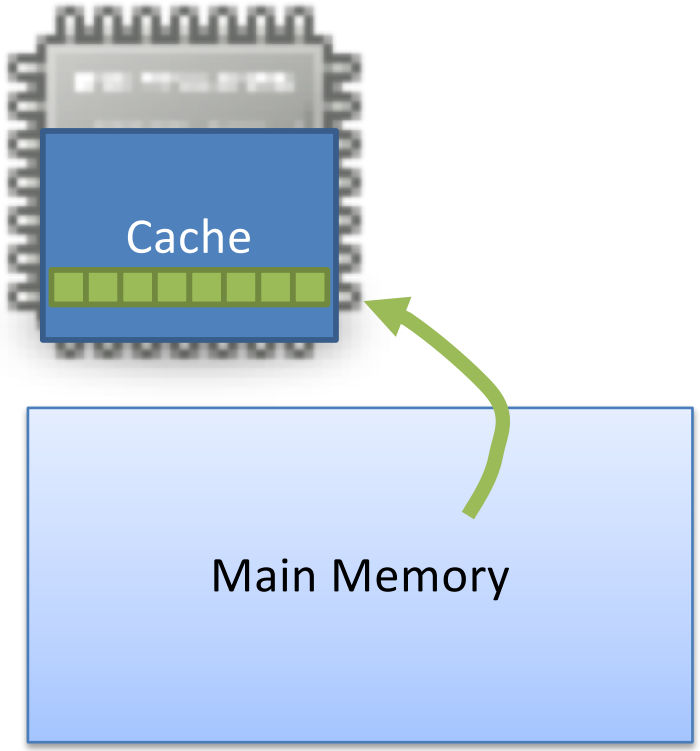
- don't know what the prog, is doing
- harmful if prog. does not exhibit good spatial locality

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

- Some associativity, medium size blocks:
  - 8-way associative (memory in one of eight places)
  - 16 or 32 or 64-byte blocks

What should we use to determine whether or not data is in the cache?

A. The memory address of the data.

B. The value of the data.

C. The size of the data.

D. Some other aspect of the data.

What should we use to determine whether or not data is in the cache?

A. The memory address of the data.
   - Memory address is how we identify the data.

B. The value of the data.
   - If we knew this, we wouldn't be looking for it!

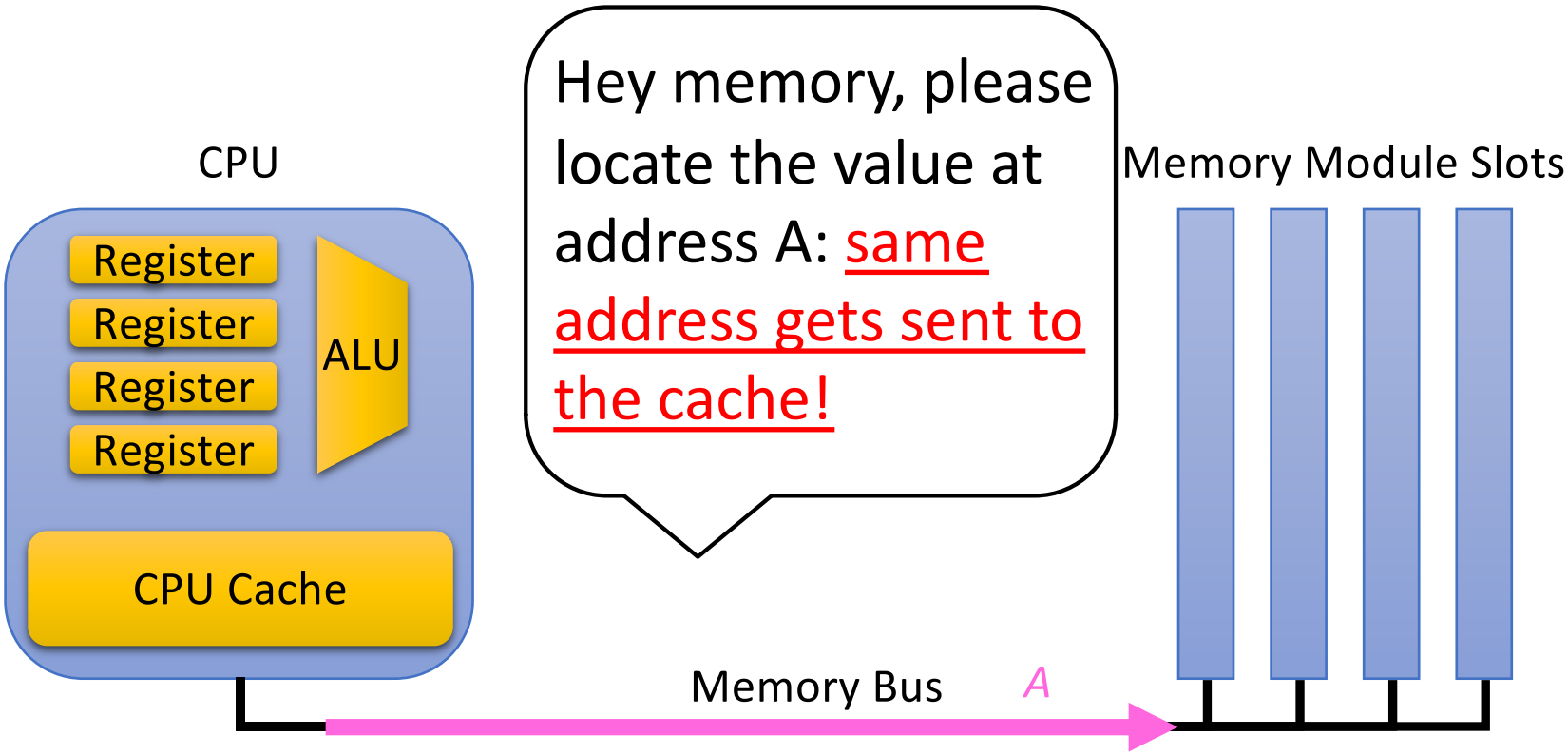
C. The size of the data.

D. Some other aspect of the data.

Recall: Memory Reads

CPU places address A on the memory bus.

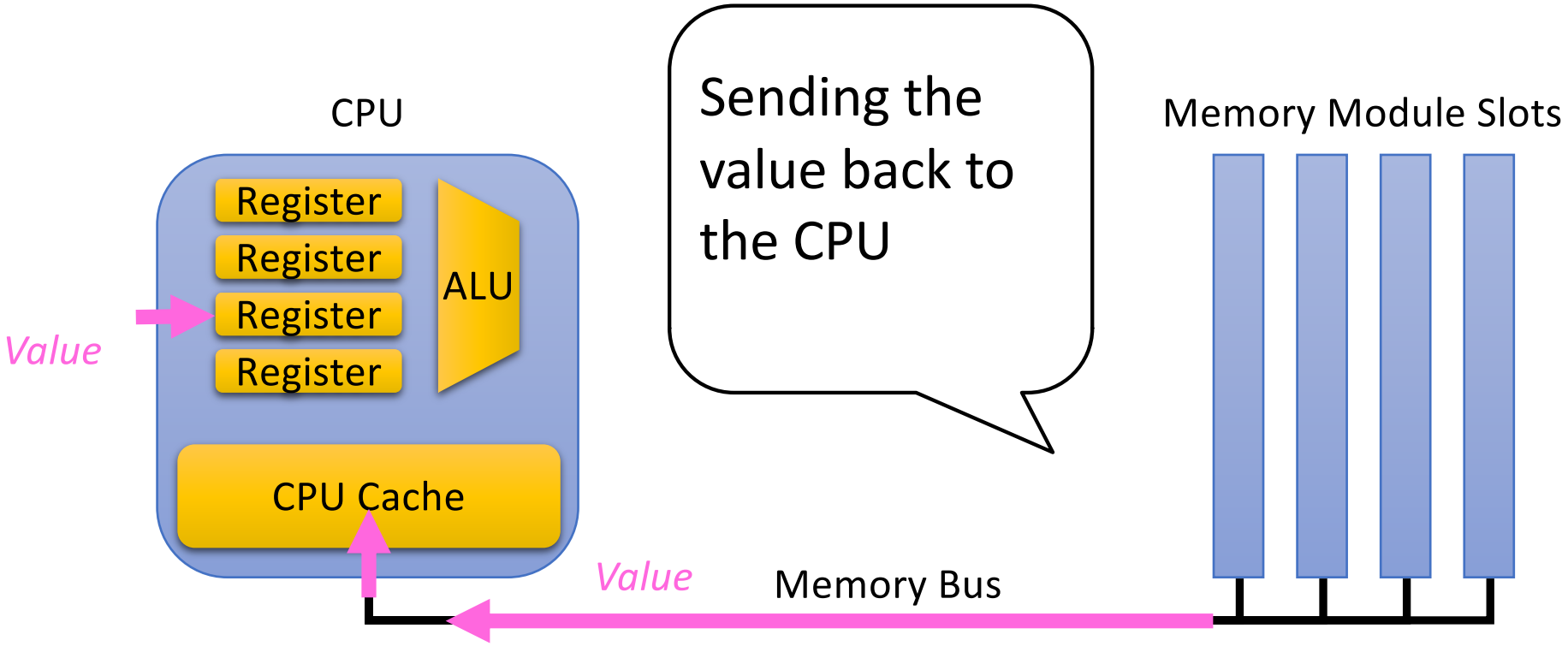
Load operation: `movl (A), %eax`



Recall: Memory Reads

Memory retrieves value and sends it across bus.

CPU reads value from the bus, and copies it into register %eax, a copy also goes into the on-chip cache memory.



Memory Address Tells Us…

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

Memory Addresses

- Like everything else: series of bits (32 or 64)
- Keep in mind:
  - N bits gives us 2^N unique values.
- 32-bit address:
  - 10110001011100101101010001010110

Divide into regions, each with distinct meaning.

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

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

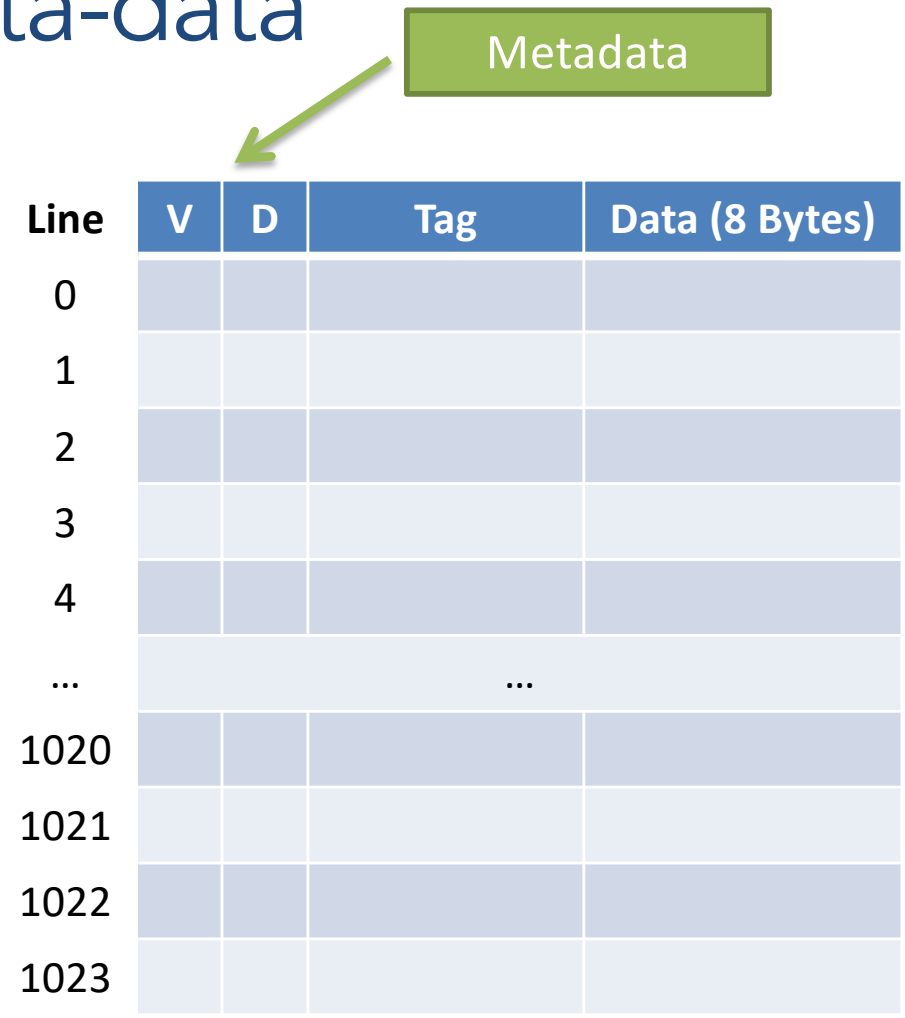
Cache meta-data

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



Address division: Direct-Mapped

- Identify byte in block
  - How many bits do we need to represent each byte uniquely?
- Identify which row (line)
  - How many bits do we need to represent each line uniquely?

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

A. Block 8 bits Row 1024 bits B. Block 3 bits Row 10 bits

C. Block 10 bits Row 10 bits D. Block 32 bits Row 32 bits

Address division: Direct-Mapped

- 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:

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

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

Index:
Which line (row) should we check?
Where could data be?

Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
	4	

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

Index:
Which line (row) should we check?
Where could data be?

Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	

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

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).

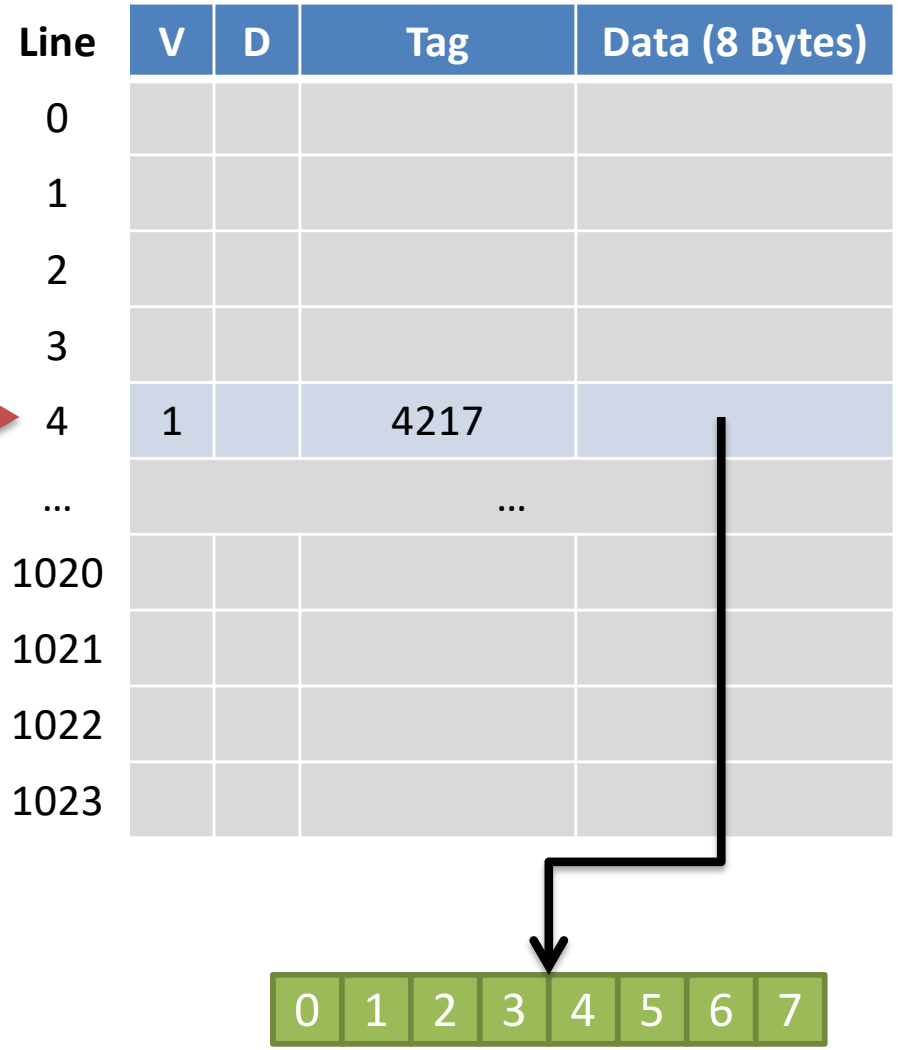
If tag doesn't match,
or line is invalid, it's a miss!

Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	

Byte offset tells us which subset of block to retrieve.



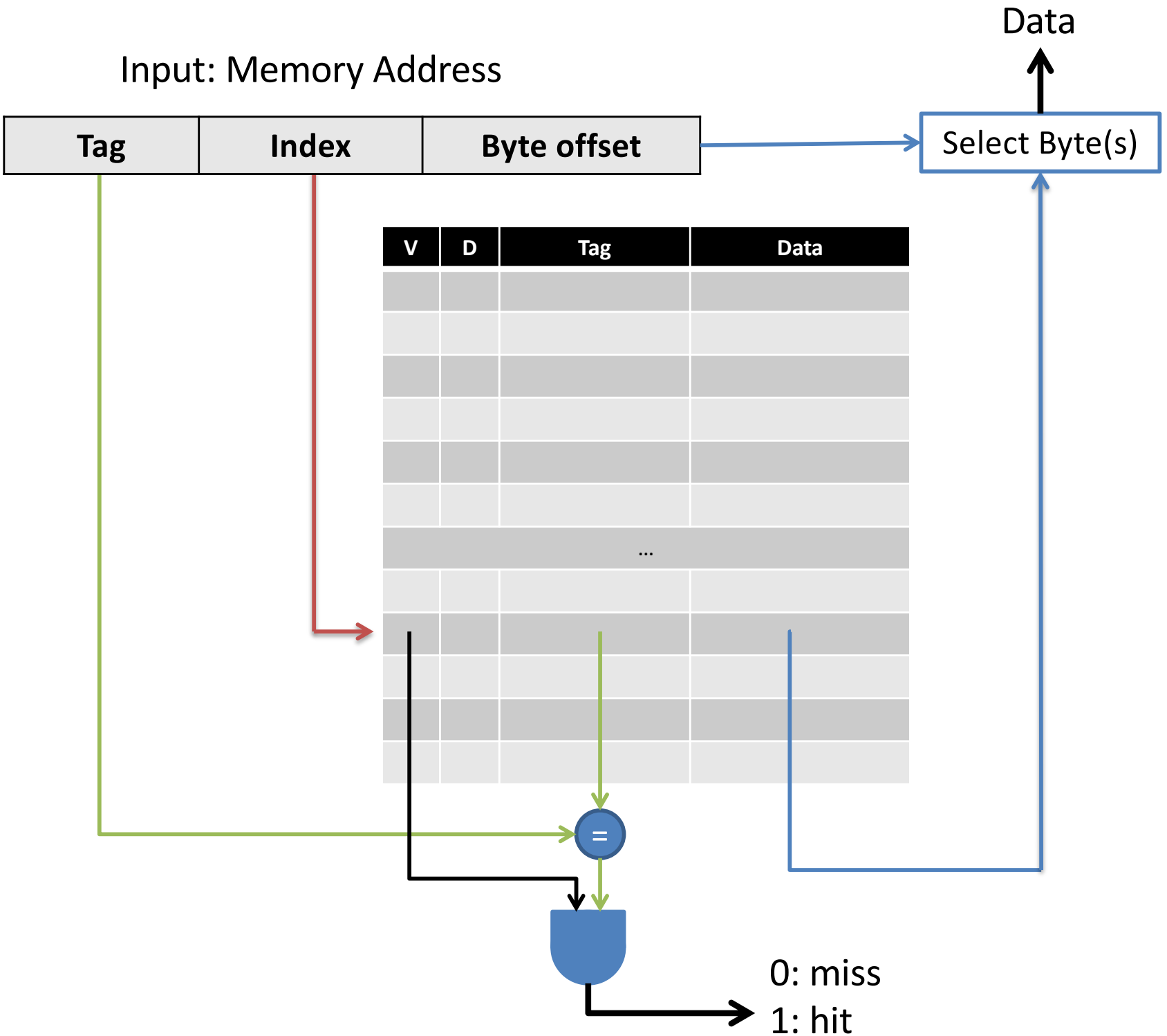
Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	2

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

Byte offset tells us which subset of block to retrieve.


Data
Input: Memory Address
Tag
Index
Byte offset
Select Byte(s)
V
D
Tag
Data
...
=
0: miss
1: hit

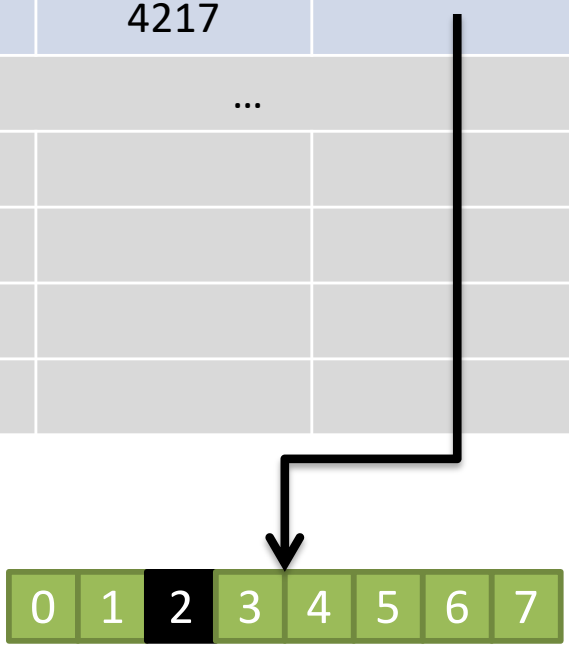
Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	2

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

Byte offset tells us which subset of block to retrieve.

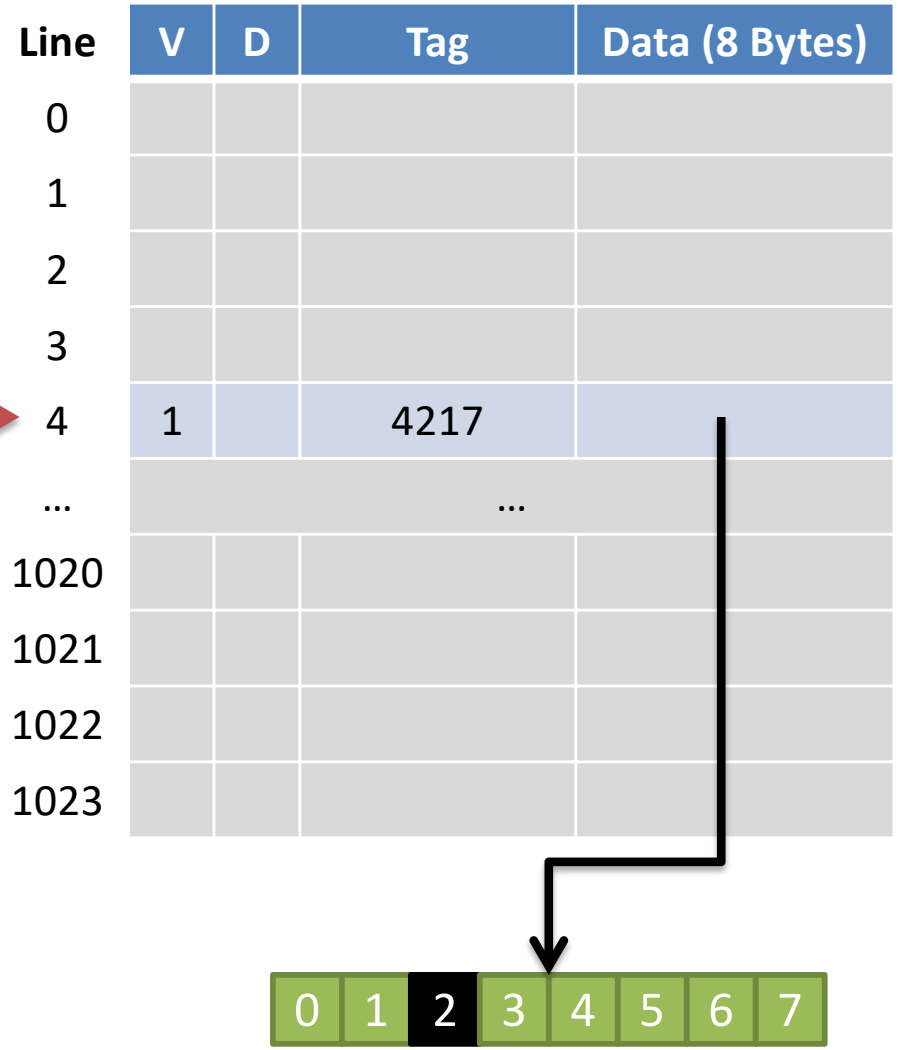


Can one read of a variable straddle multiple cache blocks?

Direct-Mapped

- Address division:

Tag (19 bits)	Index (10 bits)	Byte offset (3 bits)
4217	4	2



Byte offset tells us which subset of block to retrieve.

Can one read of a variable straddle multiple cache blocks?

No, recall mem. alignment!

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				

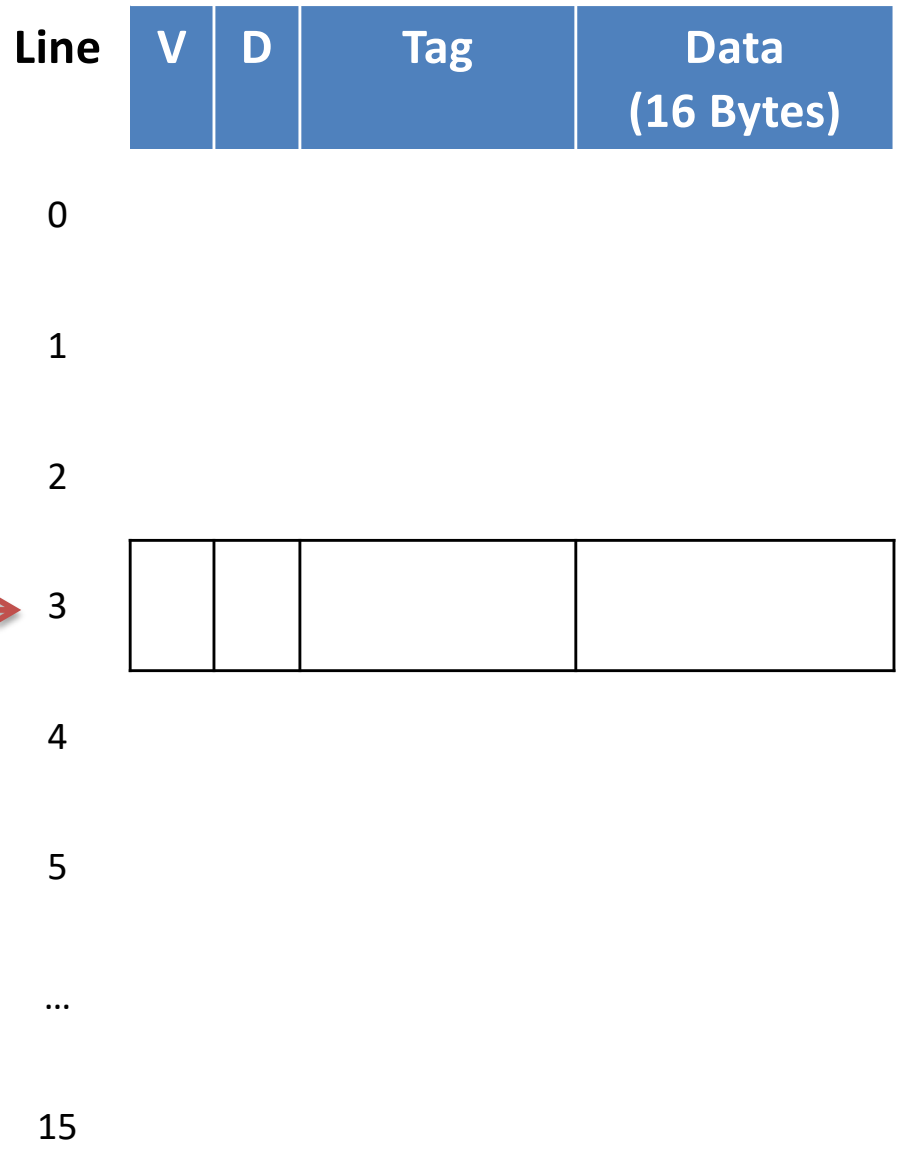
Direct-Mapped Example

- 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				

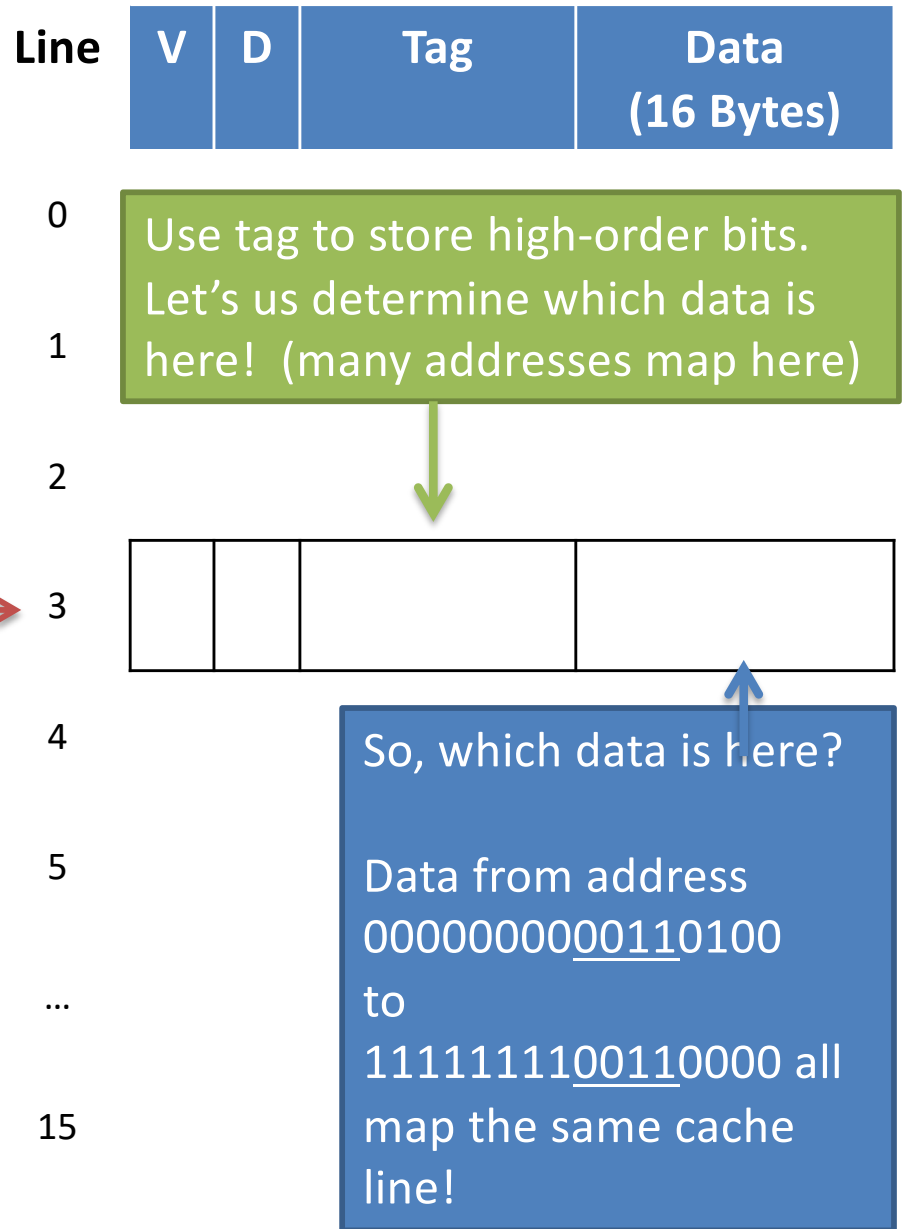
Direct-Mapped Example

- Let's say we access memory at address:
  - 01101011 0011 0100
- Step 2:
  - Use index to find line (row)
  - 0011 -> 3



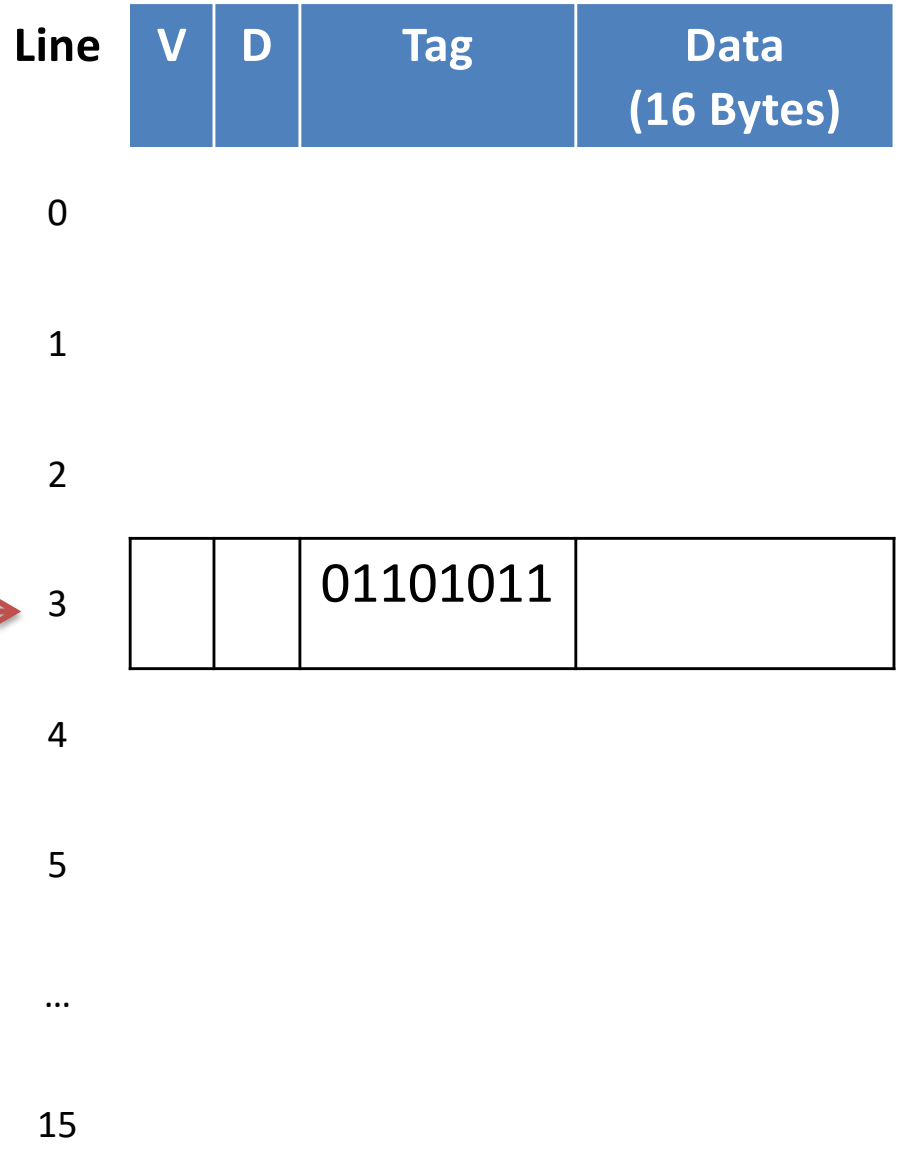
Direct-Mapped Example

- 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)



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

- For direct-mapped cache:
  - Mechanism: overwrite bits in cache line, updating
    - Valid bit
    - Tag
    - Data
  - Policy: not many options for direct-mapped
    - Overwrite at the only location it could be!

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.

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				

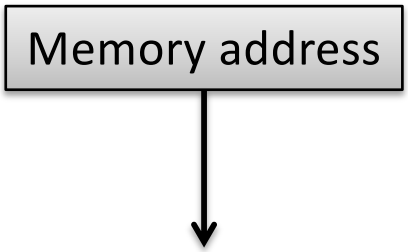
Main Memory

2. Copy data from memory. Update tag.

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?

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



Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

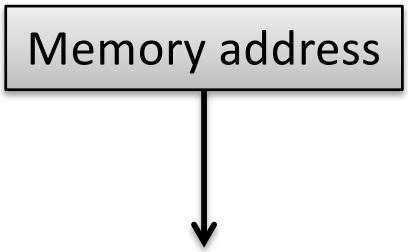
Write 01110000 (Value: 7)

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?



Read 010001**00** (Value: 5)

Read 11100010 (Value: 17)

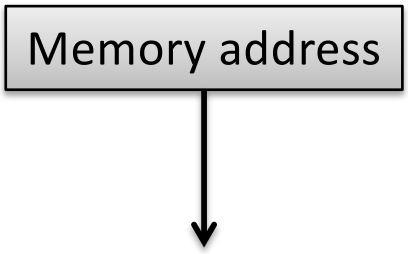
Write 01110000 (Value: 7)

Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 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?



Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01110000 (Value: 7)

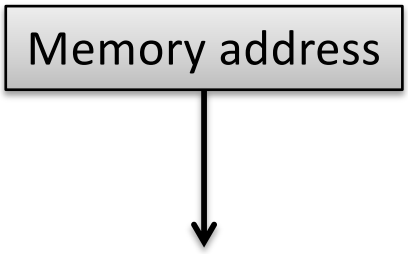
Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

No change necessary.

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 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?



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~~ 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	~~011~~ 010	~~9~~ 5
2	~~0~~ 1	0	~~101~~ 101	~~15~~ 12
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)

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	~~011~~ 010	~~9~~ 5
2	~~0~~ 1	0	~~101~~ 101	~~15~~ 12
3	1	~~1~~ 1	~~001~~ 011	~~8~~ 2
4	1	~~0~~ 1	011	~~4~~ 7
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

Associativity

- Problem: suppose we're only using a small amount of data (e.g., 8 bytes, 4-byte block size)

- Bad luck: (both) blocks map to same cache line
  - Constantly evicting one another
  - Rest of cache is going unused!

- Associativity: allow a set blocks to be stored at the same index. Goal: reduce conflict misses.

Comparison

Direct-mapped

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- Middle bits (index) tell you which 1 line to check.

- (+) Low complexity, fast.
- (-) Conflict misses.

N-way set associative

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- Middle bits (set) tell you which N lines to check.

- (+) Fewer conflict misses.
- (-) More complex, slower, consumes more power.

Comparison: 1024 Lines

(For the same cache size, in bytes.)

Direct-mapped

- 1024 indices (10 bits)

2-way set associative

- 512 sets (9 bits)
  - Tag slightly (1 bit) larger.

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4								
…			…				…	
508								
509								
510								
511								

2-Way Set Associative

Tag (20 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

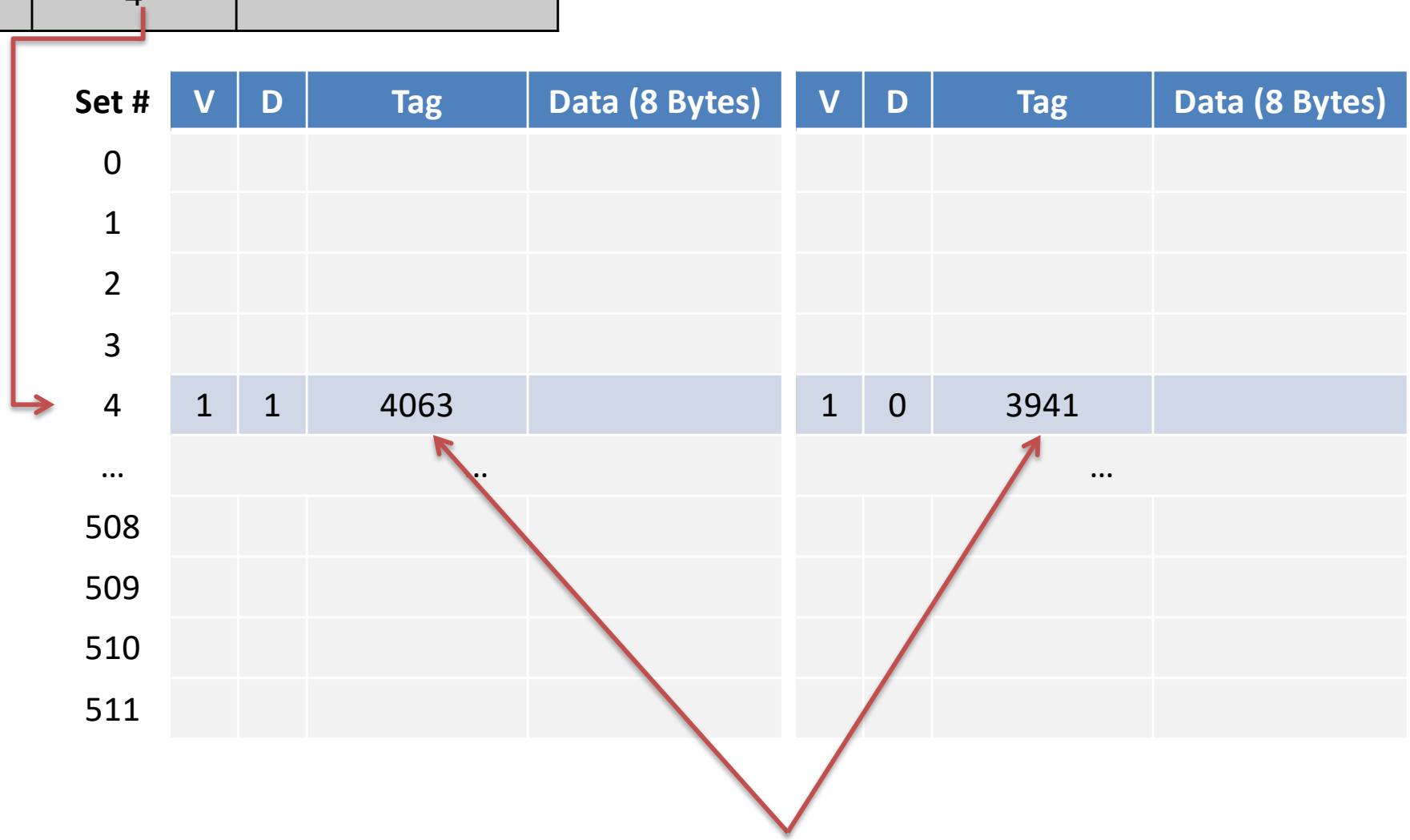
Same capacity as previous example:
1024 rows with 1 entry vs.
512 rows with 2 entries

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								

2-Way Set Associative

Tag (20 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								

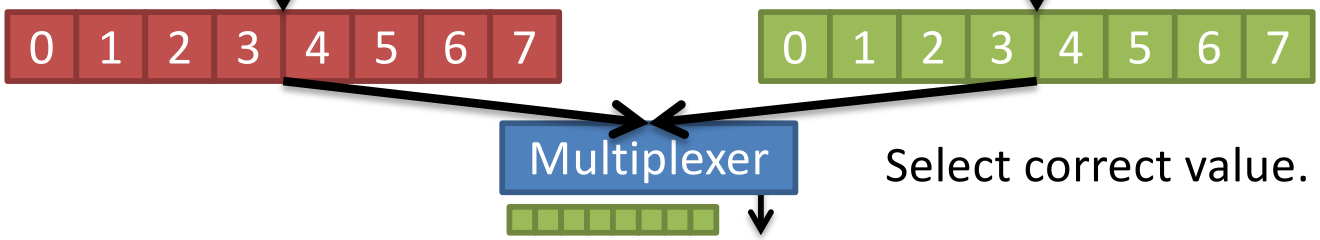


Check all locations in the set, in parallel.

2-Way Set Associative

Tag (20 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

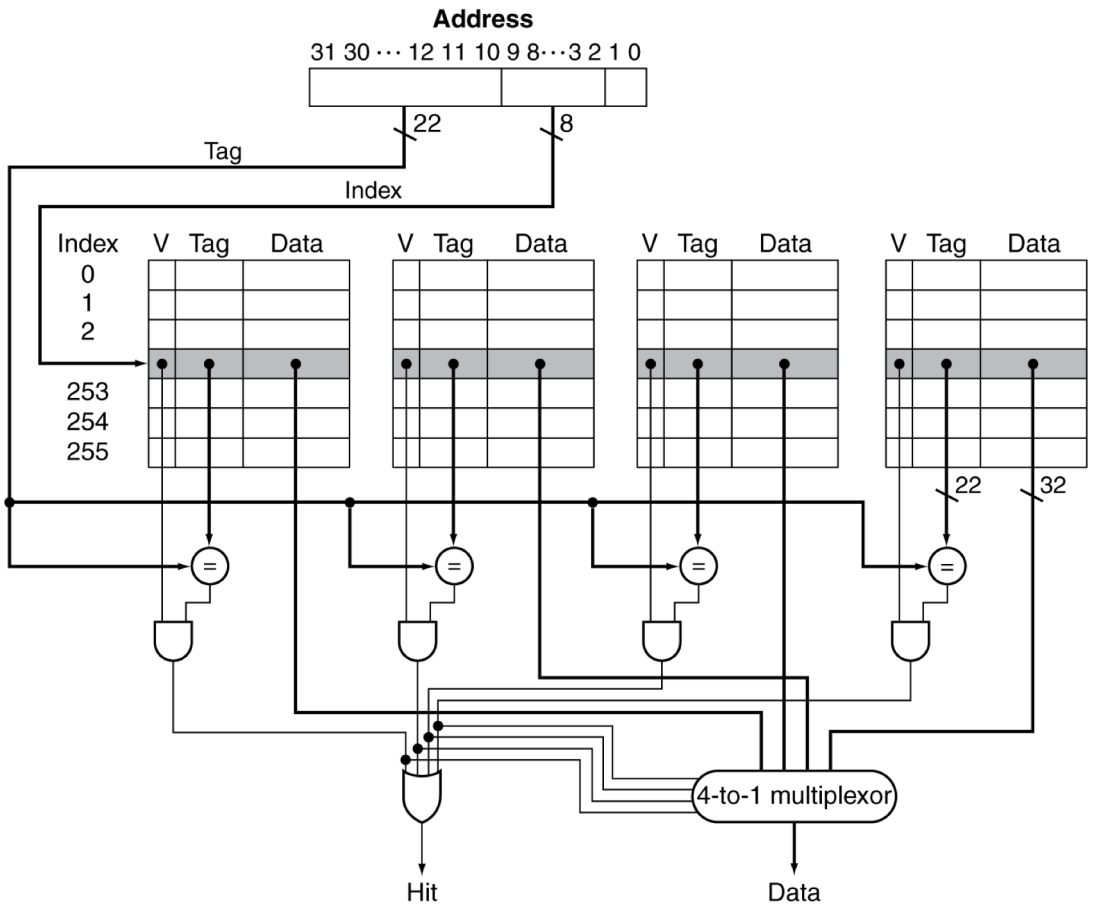
Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								



Select correct value.

4-Way Set Associative Cache

Clearly, more complexity here!



Eviction

- Mechanism is the same…
  - Overwrite bits in cache line: update tag, valid, data

- Policy: choose which line in the set to evict
  - Pick a random line in set
  - Choose an invalid line first
  - Choose the least recently used block
    - Has exhibited the least locality, kick it out!

Common combo in practice.

Least Recently Used (LRU)

- Intuition: if it hasn't been used in a while, we have no reason to believe it will be used soon.

- Need extra state to keep track of LRU info.

Set #	LRU	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0	0								
1	1								
2	1								
3	0								
4	1	1	1	4063		1	0	3941	
…		…				…			

Least Recently Used (LRU)

- Intuition: if it hasn't been used in a while, we have no reason to believe it will be used soon.

- Need extra state to keep track of LRU info.

- For perfect LRU info:
  - 2-way: 1 bit
  - 4-way: 8 bits
  - N-way: N * $\log_2$ N bits

Another reason why associativity often maxes out at 8 or 16.

These are metadata bits, not "useful" program data storage.

(Approximations make it not quite as bad.)

How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01100100 (Value: 7)

Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
0	1	0	0	111	4	1	0	001	17
1	0	1	1	111	9	1	0	010	5
2				…	…			…	…
3									
4									
5									
6									
7									

Cache Conscious Programming

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

  (ex) 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

(ex) 2D array accesses

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

A. is faster.

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

B. is faster.

Algorithmically, both O(N * M).

Is one faster than the other?

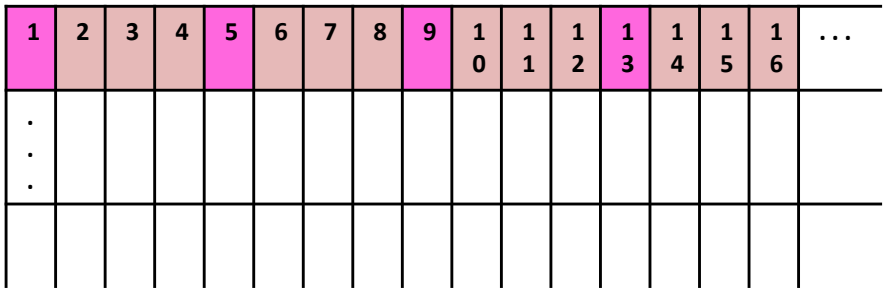
C. Both would exhibit roughly equal performance.

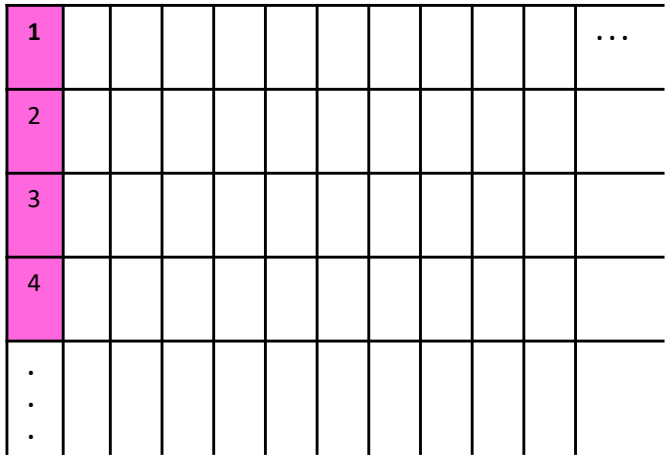
Cache Conscious Programming

The first 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];
}}
```





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

Program Efficiency and Memory

- Be aware of how your program accesses data
  - Sequentially, in strides of size X, randomly, …
  - How data is laid out in memory
- Will allow you to structure your code to run much more efficiently based on how it accesses its data
- Don’t go nuts…
  - Optimize the most important parts, ignore the rest
  - “Premature optimization is the root of all evil.” -Knuth

Amdahl's Law

Idea: an optimization can improve total runtime at most by the fraction it contributes to total runtime

> If program takes 100 secs to run, and you optimize a portion of the code that accounts for 2% of the runtime, the best your optimization can do is improve the runtime by 2 secs.

Amdahl's Law tells us to focus our optimization efforts on the code that matters:

> Speed-up what is accounting for the largest portion of runtime to get the largest benefit. And, don't waste time on the small stuff.

Up Next:

- Operating systems, Processes
- Virtual Memory