# CS 31: Introduction to Computer Systems

15-16: Storage and Memory

March 26-31, 2020

SWARTHMORE COLLEGE

## Transition

- First half of course: hardware focus
  - How the hardware is constructed
  - How the hardware works
  - How to interact with hardware / ISA
- Up next: performance and software systems
  - Memory performance
  - Operating systems
  - Standard libraries (strings, threads, etc.)

## Efficiency

- How to <u>Efficiently</u> Run Programs

- Good algorithm is critical…

- Many systems concerns to account for too!
  - The memory hierarchy and its effect on program performance
  - OS abstractions for running programs efficiently
  - Support for parallel programming

## Suppose you're designing a new computer architecture. Which type of memory would you use? Why?

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100's of GB), slow, cheap

D. something else (it must exist)

## Suppose you're designing a new computer architecture. Which type of memory would you use? Why?

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100's of GB), slow, cheap

D. something else (it must exist)

trade-off between capacity and speed

## Classifying Memory

- Broadly, two types of memory:
  1. Primary storage: CPU instructions can access any location at any time (assuming OS permission)
  2. Secondary storage: CPU can’t access this directly

# Random Access Memory (RAM)

- Any location can be accessed directly by CPU
  - Volatile Storage: lose power → lose contents
- Static RAM (SRAM)
  - Latch-Based Memory (e.g. RS latch), 1 bit per latch
  - Faster and more expensive than DRAM
    - “On chip”: Registers, Caches
- Dynamic RAM (DRAM)
  - Capacitor-Based Memory, 1 bit per capacitor
    - “Main memory”: Not part of CPU

## Memory Technologies

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 100ns, $20 – $75 per GB
    (Main memory, “RAM”)

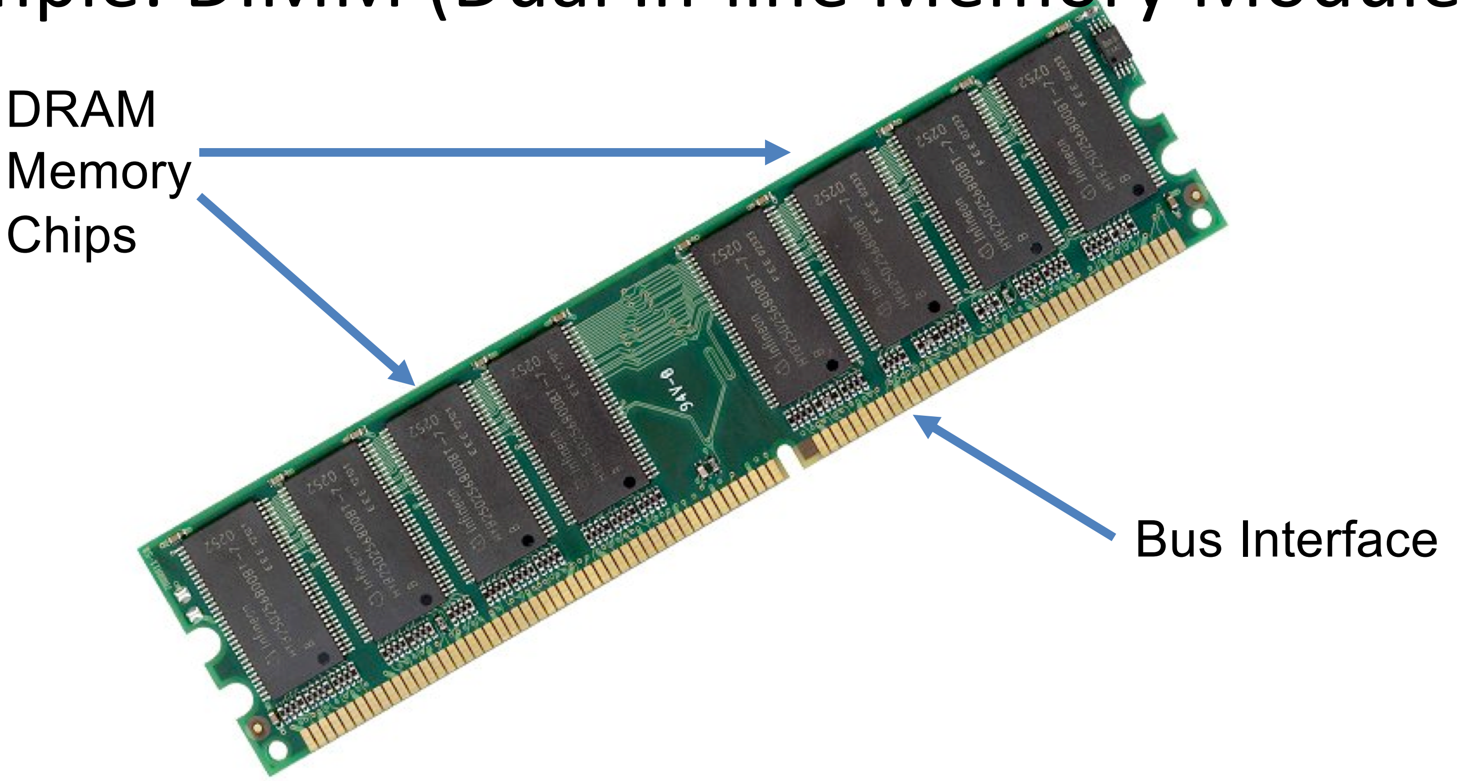
We’ve talked a lot about registers (SRAM) and we’ll cover caches (SRAM) soon.
Let’s look at main memory (DRAM) now.

## Dynamic Random Access Memory (DRAM)

Capacitor based:

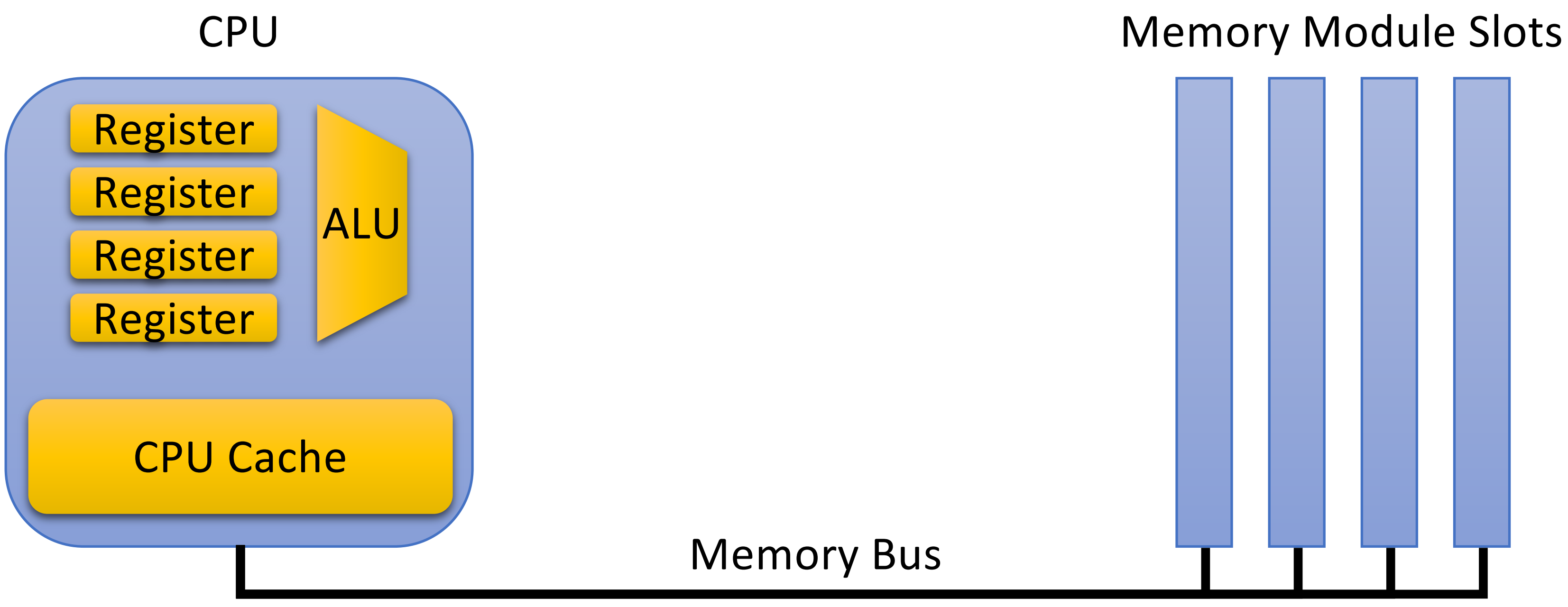
- cheaper and slower than SRAM
- capacitors are leaky (lose charge over time)
- Dynamic: value needs to be refreshed (every 10-100ms)

Example: DIMM (Dual In-line Memory Module):



# Connecting CPU and Memory

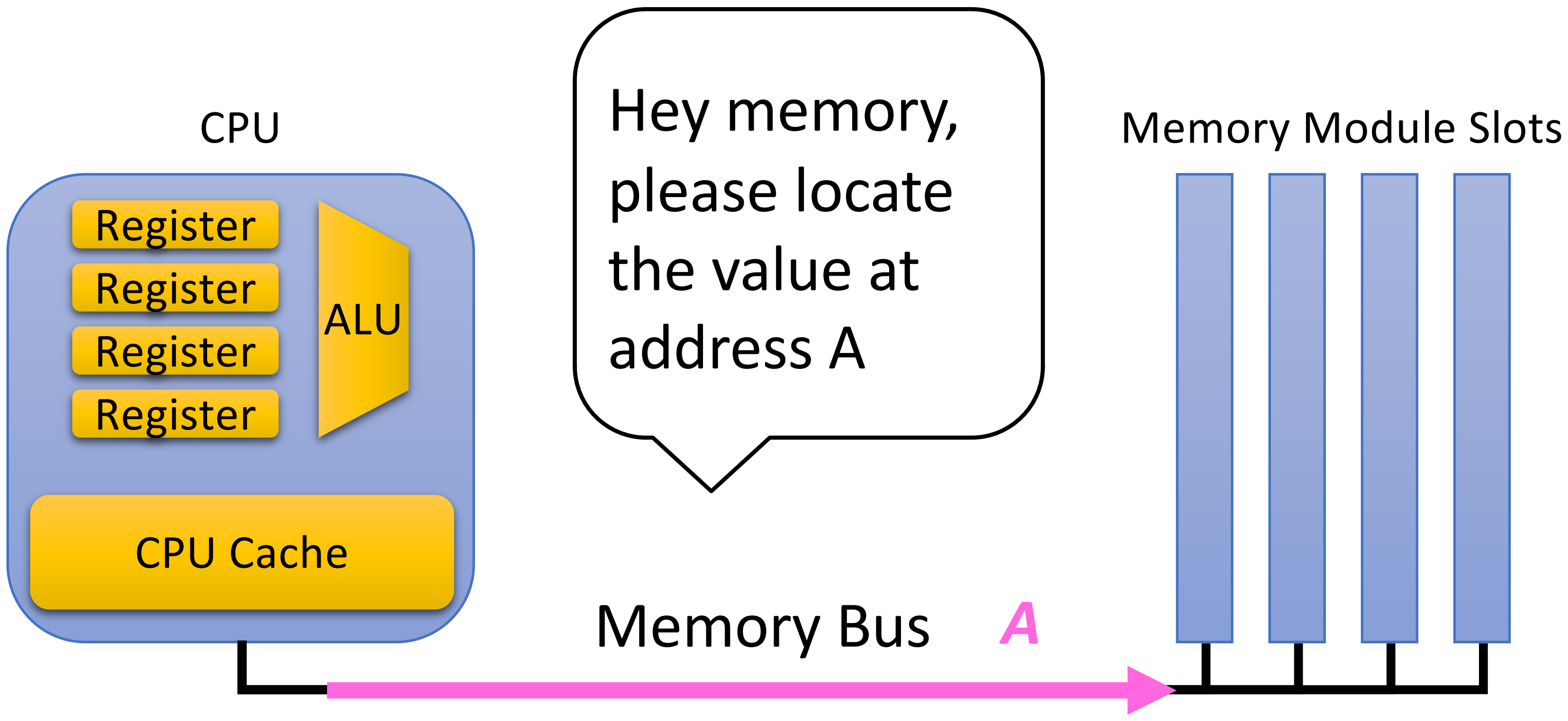
- Components are connected by a bus:
  - A bus is a collection of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.



# How Memory Read Works

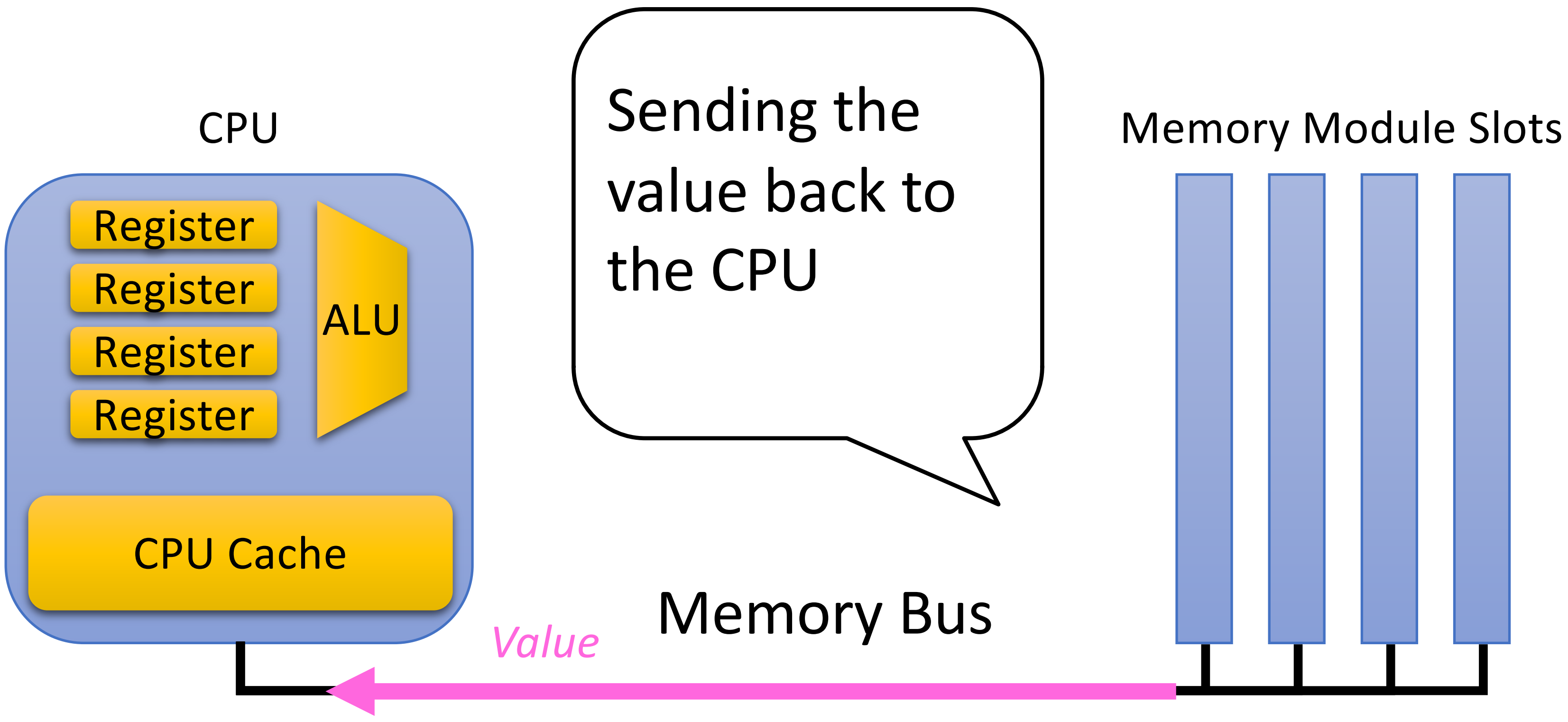
(1) CPU places address A on the memory bus.

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



# Read (cont.)

(2) Main Memory reads address A from memory, fetches value at that address and puts it on the bus

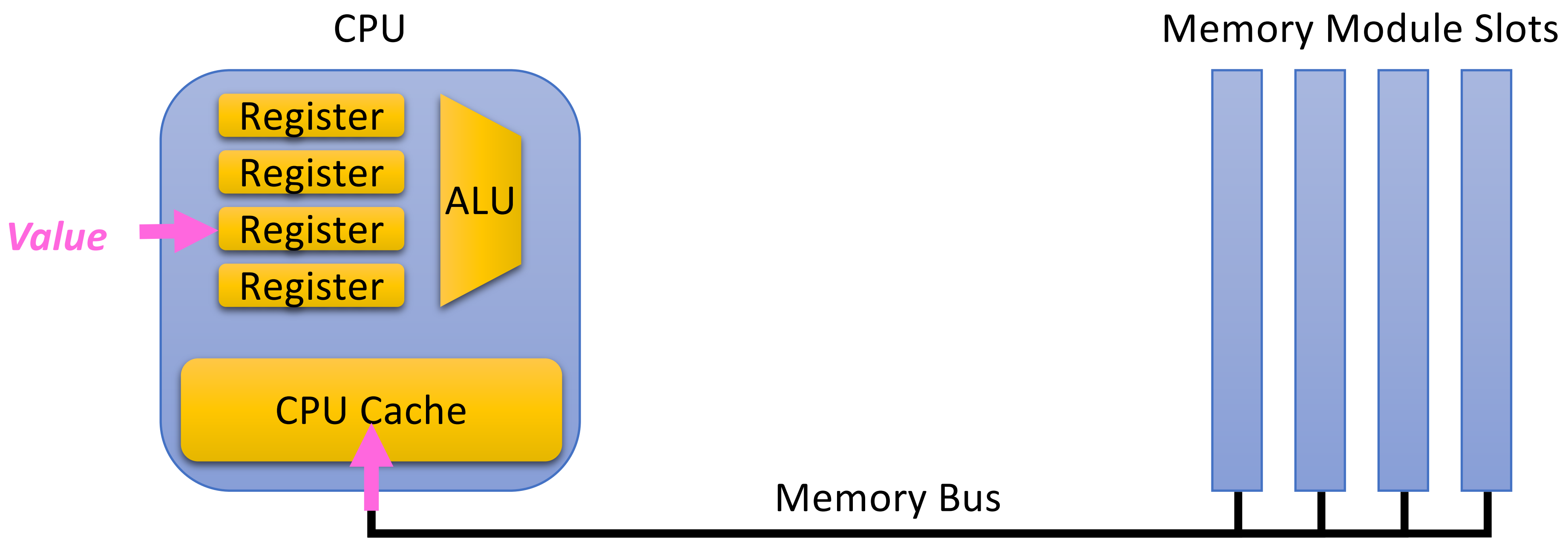


# Read (cont.)

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

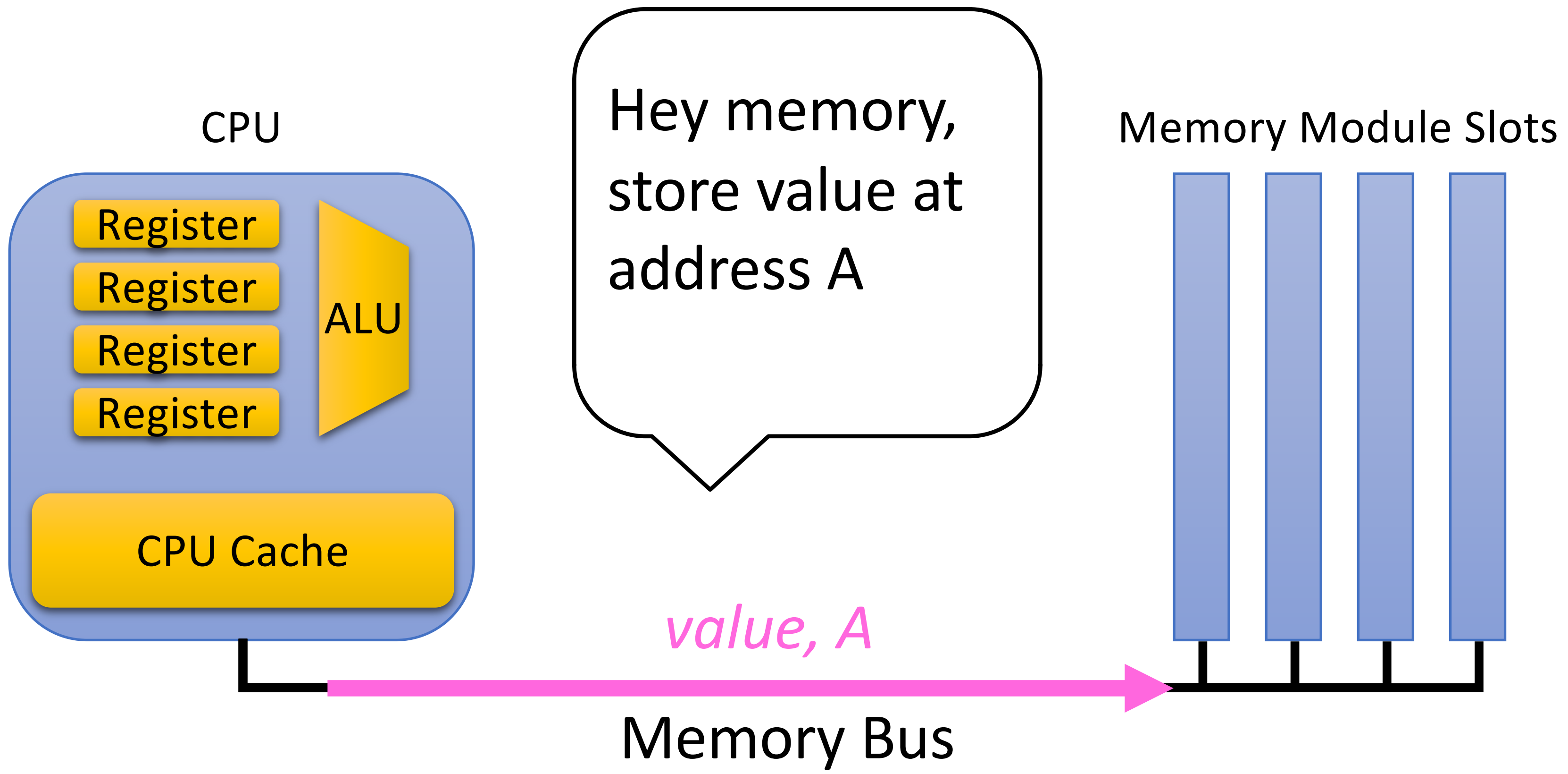
(3) CPU reads value from the bus,

- copies it into register %eax,
- a copy also goes into the on-chip cache memory



## Write

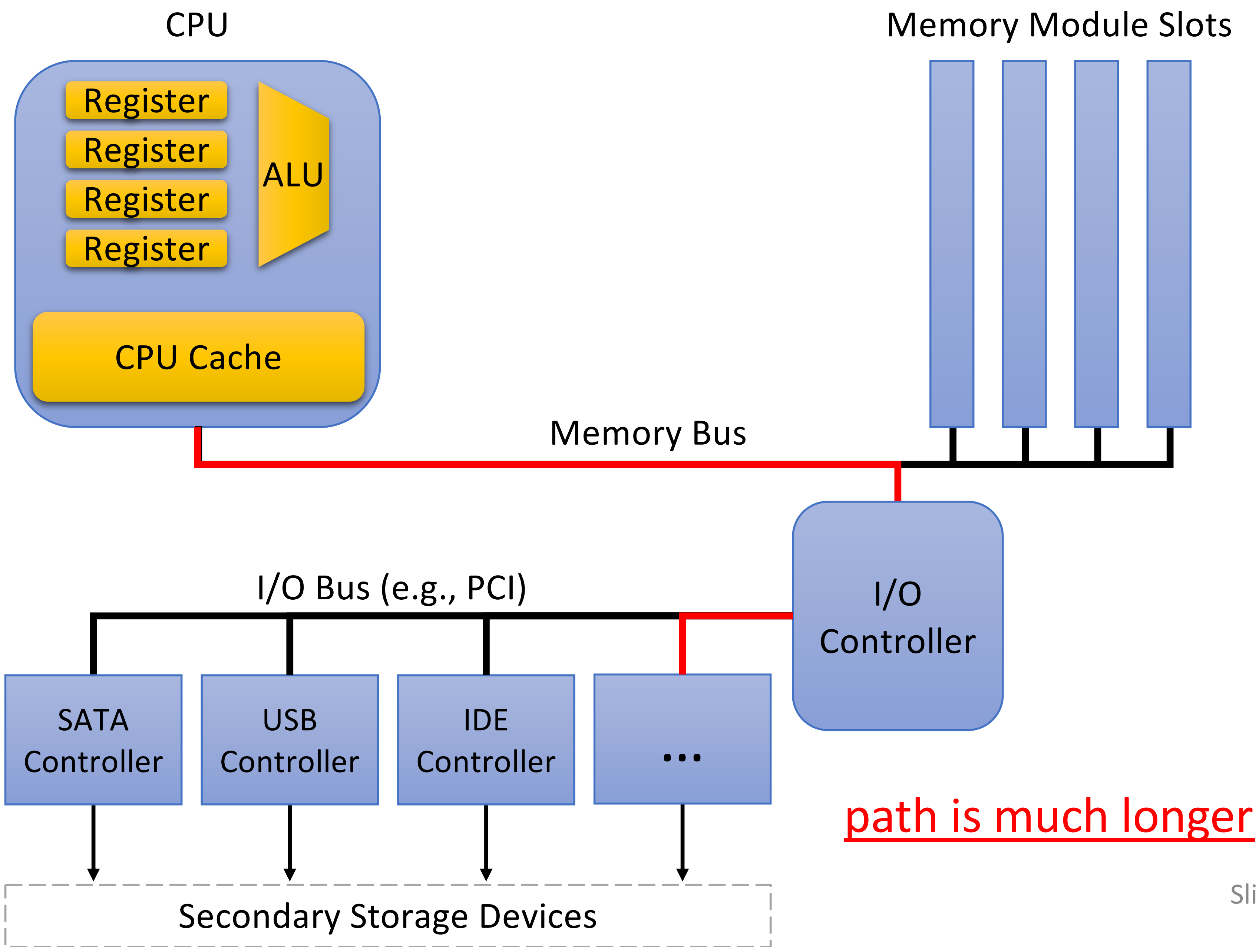
1. CPU writes A to bus, memory reads it
2. CPU writes value to bus, memory reads it
3. Memory stores value, y, at address A



## Secondary Storage

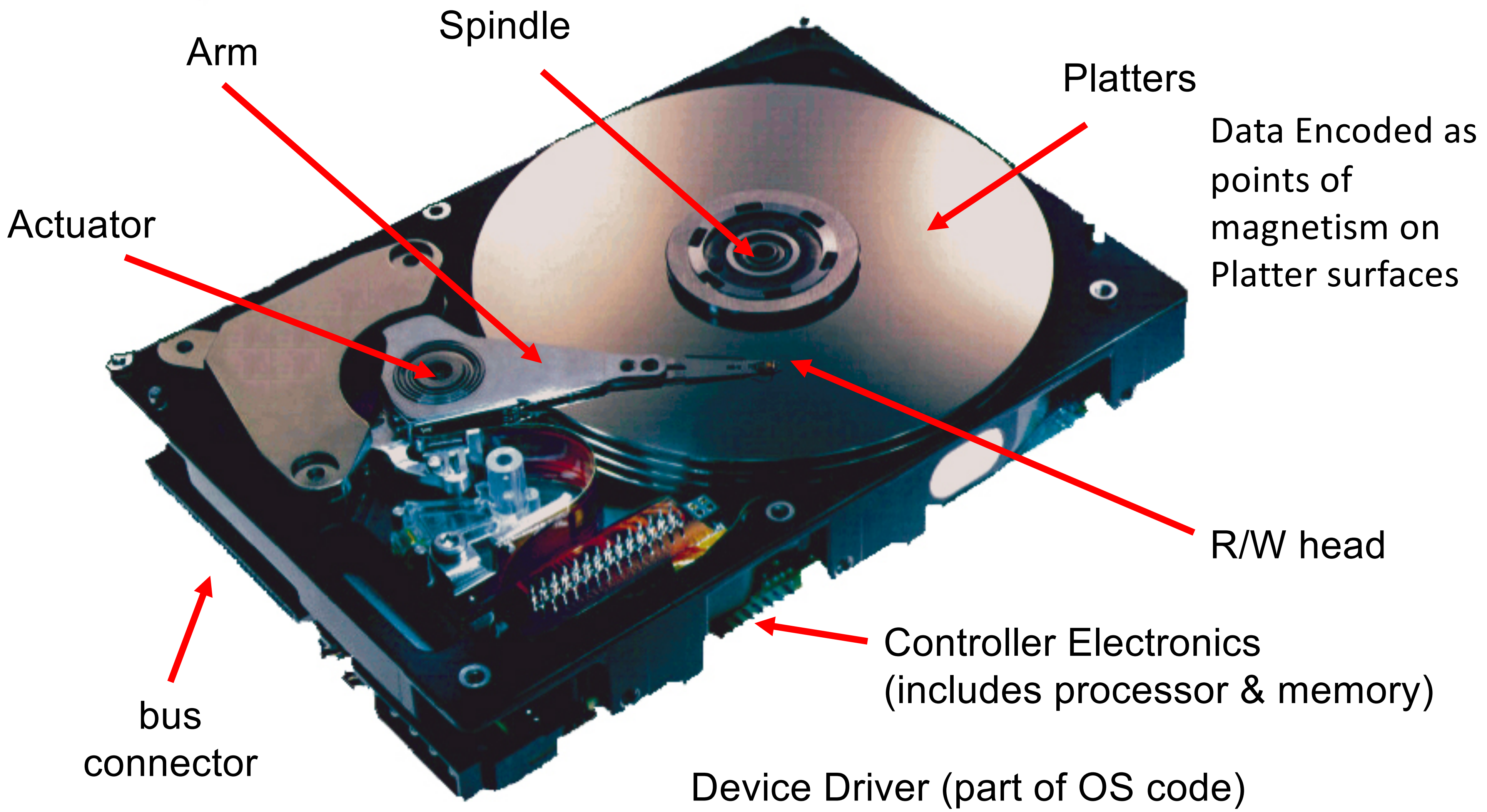
- Disk, Tape Drives, Flash Solid State Drives, …
- Non-volatile: retains data without a charge
- Instructions CANNOT directly access data on secondary storage
  - No way to specify a disk location in an instruction
  - Operating System moves data to/from memory

## Secondary Storage



path is much longer

## What’s Inside A Disk Drive?



Data Encoded as points of magnetism on Platter surfaces

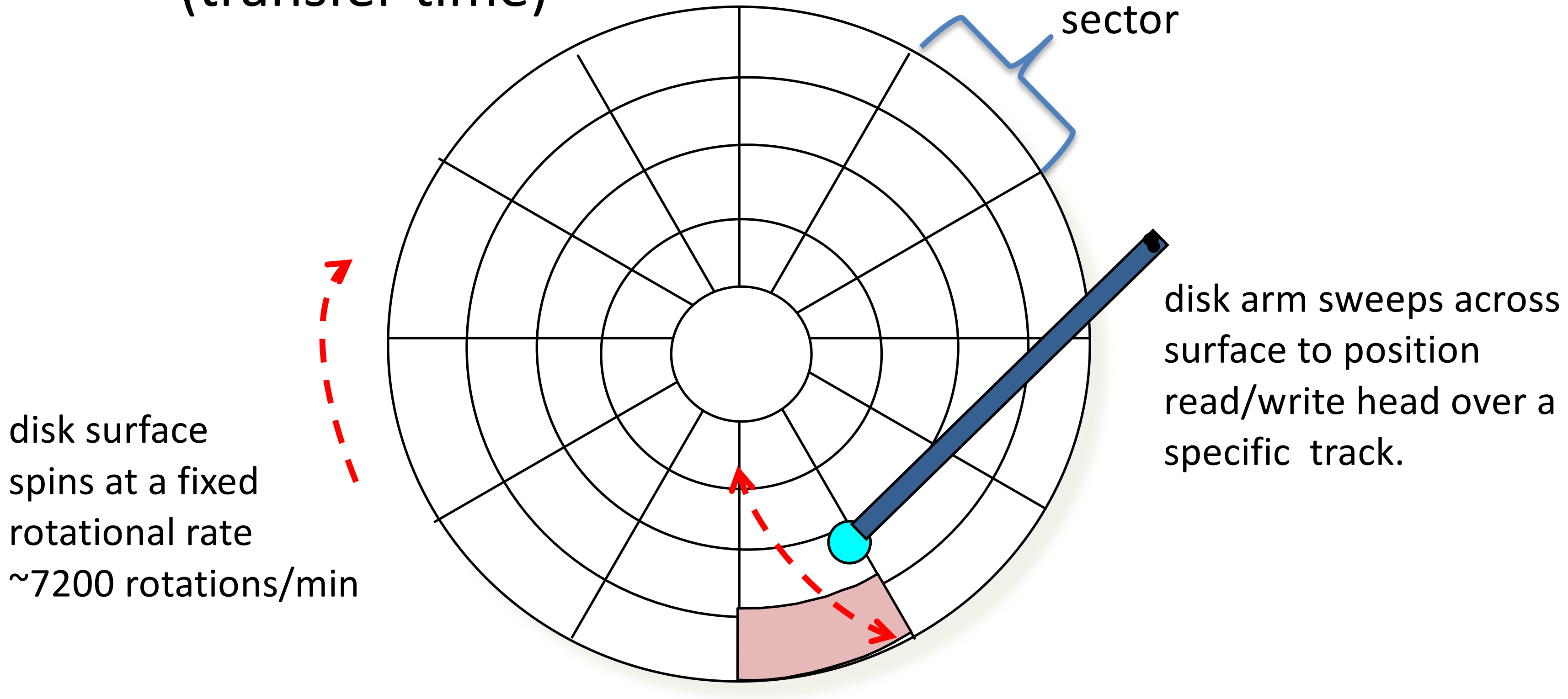
Device Driver (part of OS code) interacts with Controller to R/W to disk

*Image from Seagate Technology*

# Reading and Writing to Disk

Data blocks located in some Sector of some Track on some Surface

1. Disk Arm moves to correct track (seek time)
2. Wait for sector spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)


sector
disk arm sweeps across surface to position read/write head over a specific track.
disk surface spins at a fixed rotational rate ~7200 rotations/min

## Memory Technology

Like walking:

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

Down the hall

- Dynamic RAM (DRAM)
  - 50ns – 100ns, $20 – $75 per GB

Across campus

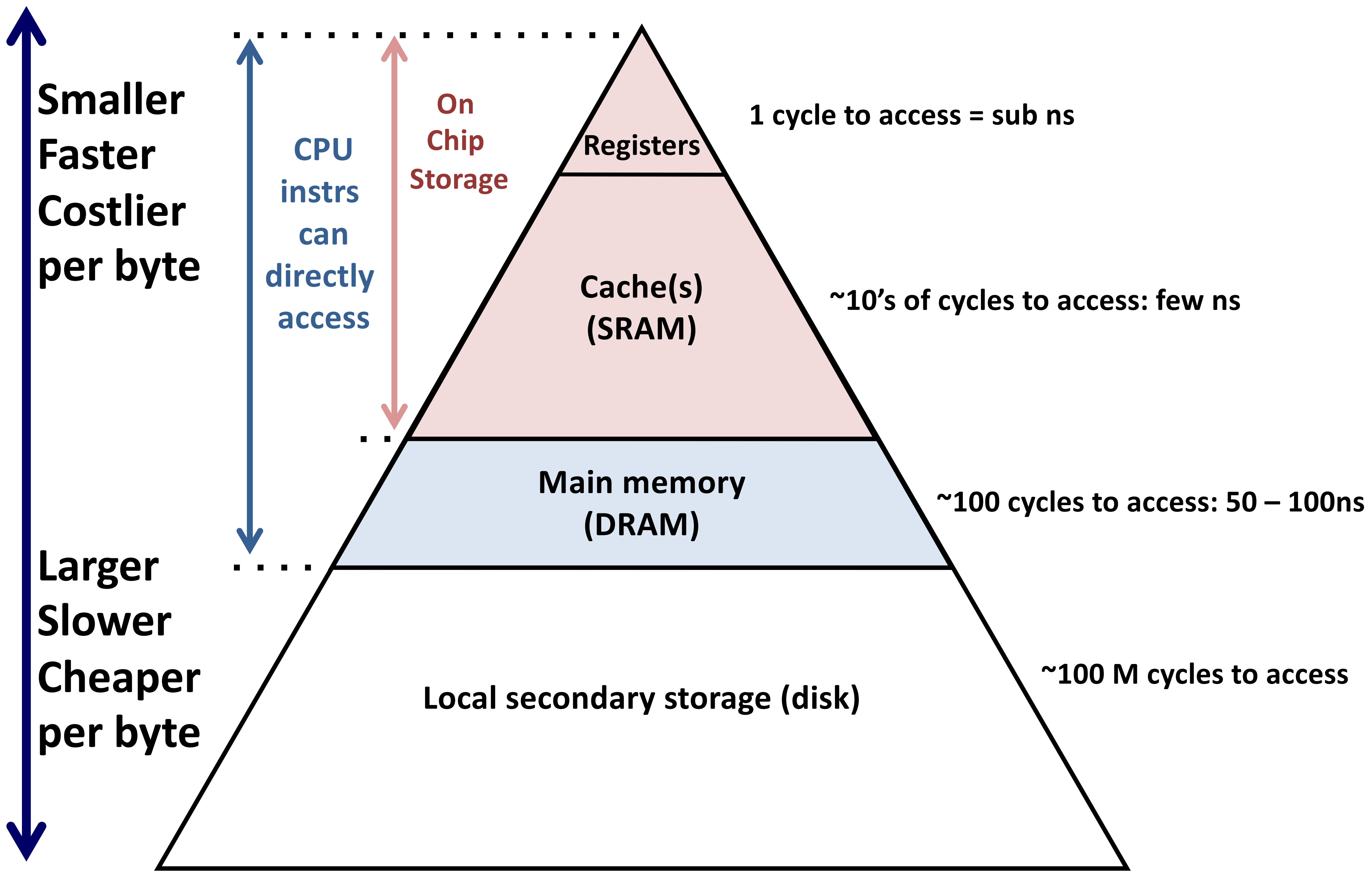
Solid-state disks (flash): 100 us – 1 ms, $2 - $10 per GB

- Magnetic disk
  - 5ms – 15ms, $0.20 – $2 per GB

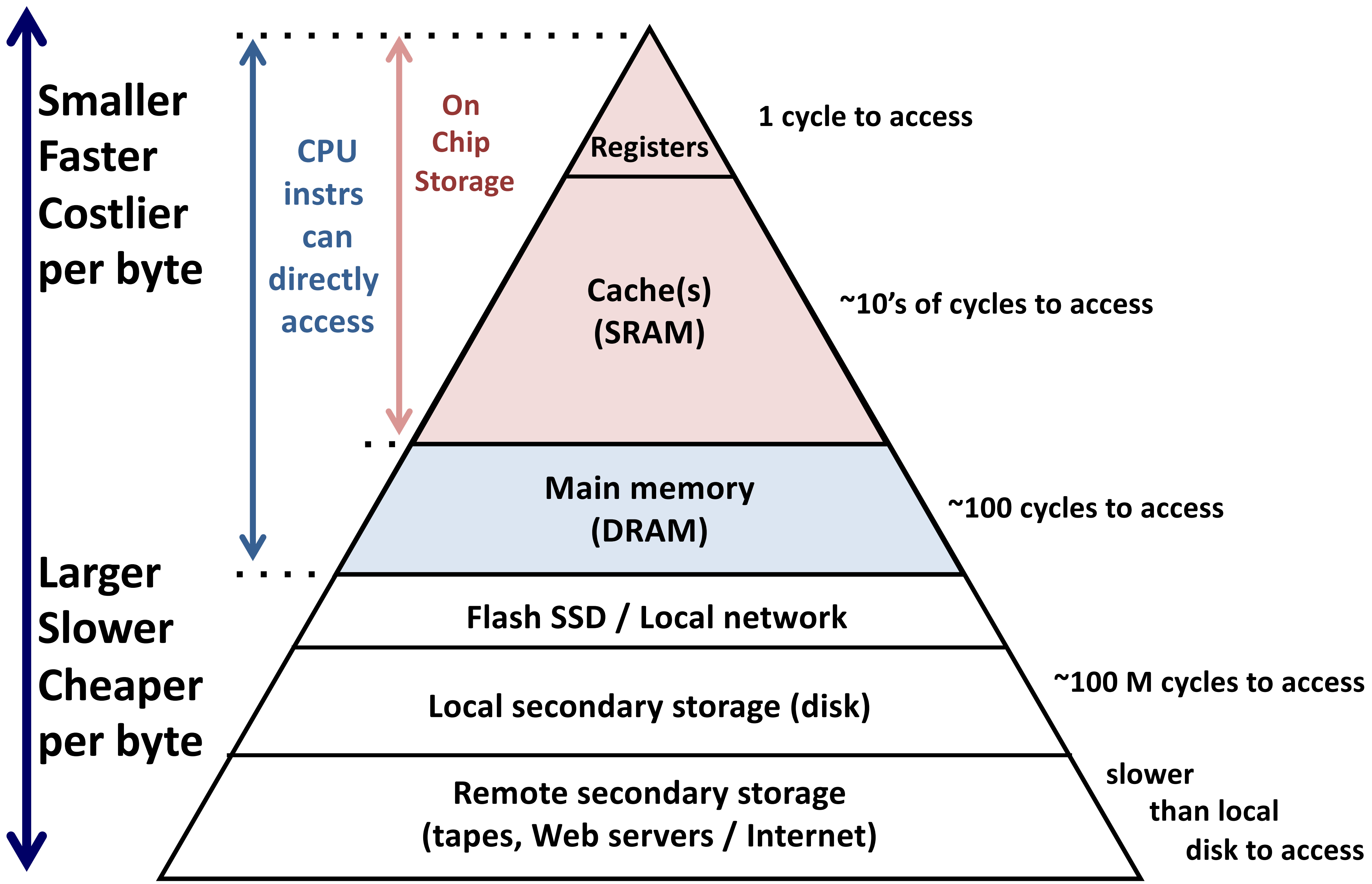
To Seattle

1 ms == 1,000,000 ns

## The Memory Hierarchy


Smaller
Faster
Costlier
per byte
Larger
Slower
Cheaper
per byte
CPU instrs can directly access
On Chip Storage
Registers
Cache(s) (SRAM)
Main memory (DRAM)
Local secondary storage (disk)
1 cycle to access = sub ns
~10's of cycles to access: few ns
~100 cycles to access: 50 – 100ns
~100 M cycles to access

# The Memory Hierarchy



Smaller
Faster
Costlier
per byte
CPU instrs can directly access
On Chip Storage
Registers
1 cycle to access
Cache(s) (SRAM)
~10's of cycles to access
Main memory (DRAM)
~100 cycles to access
Flash SSD / Local network
Local secondary storage (disk)
~100 M cycles to access
Remote secondary storage (tapes, Web servers / Internet)
slower than local disk to access
Larger
Slower
Cheaper
per byte

## Abstraction Goal

- Reality: There is no one type of memory to rule them all!

- Abstraction: hide the complex/undesirable details of reality.

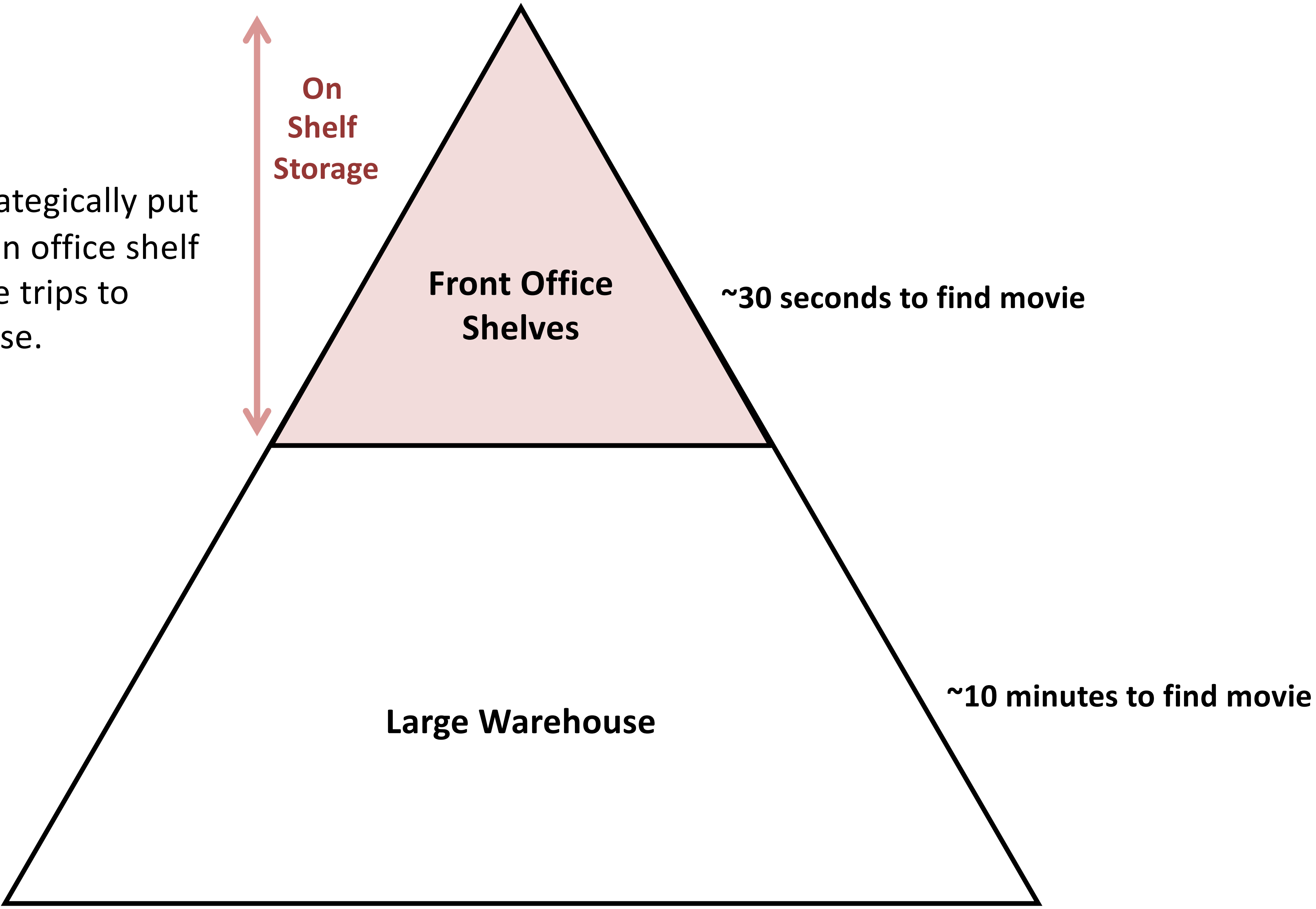
- Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.

## Motivating Story / Analogy

- You work at a video rental store (remember Blockbuster?)
- You have a huge warehouse of movies
  - 10-15 minutes to find movie, bring to customer
  - Customers don't like waiting…
- You have a small office in the front with shelves, you choose what goes on shelves
  - < 30 seconds to find movie on shelf

# The Video Store Hierarchy

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



## Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind

B. The Godfather

C. Rocky

D. Spirited Away

E. There’s no way for us to know.

## Problem: Prediction

- We can’t know the future…
- So… are we out of luck?
  What might we look at to help us decide?
- The past is often a pretty good predictor…

## Repeat Customer: Bob

- Has rented “Eternal Sunshine of the Spotless Mind” ten times in the last two weeks.
- You talk to him:
  - He just broke up with his girlfriend
  - Swears it will be the last time he rents the movie (he’s said this the last six times)

## Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind

B. The Godfather

C. Rocky

D. Spirited Away

E. There’s no way for us to know.

# Repeat Customer: Alice

- Alice rented Rocky a month ago
- You talk to her:
  - She's really likes Sylvester Stalone
- Over the next few weeks she rented:
  - Rocky II, Rocky III, Rocky IV

## Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind

B. The Godfather

C. Pulp Fiction

D. Rocky V

E. There’s no way for us to know.

## Critical Concept: Locality

- Locality: we tend to repeatedly access recently accessed items, or those that are nearby.
- Temporal locality: An item accessed recently is likely to be accessed again soon. (Bob)
- Spatial locality: We’re likely to access an item that’s nearby others we just accessed. (Alice)

## In the following code, how many examples are there of temporal / spatial locality? Where are they?

```
int i;
int num = read_int_from_user();
int *array = create_random_array(num);
for (i = 0; i < num; i++) {
  printf("At index %d, value: %d", i, array[i]);
}
```

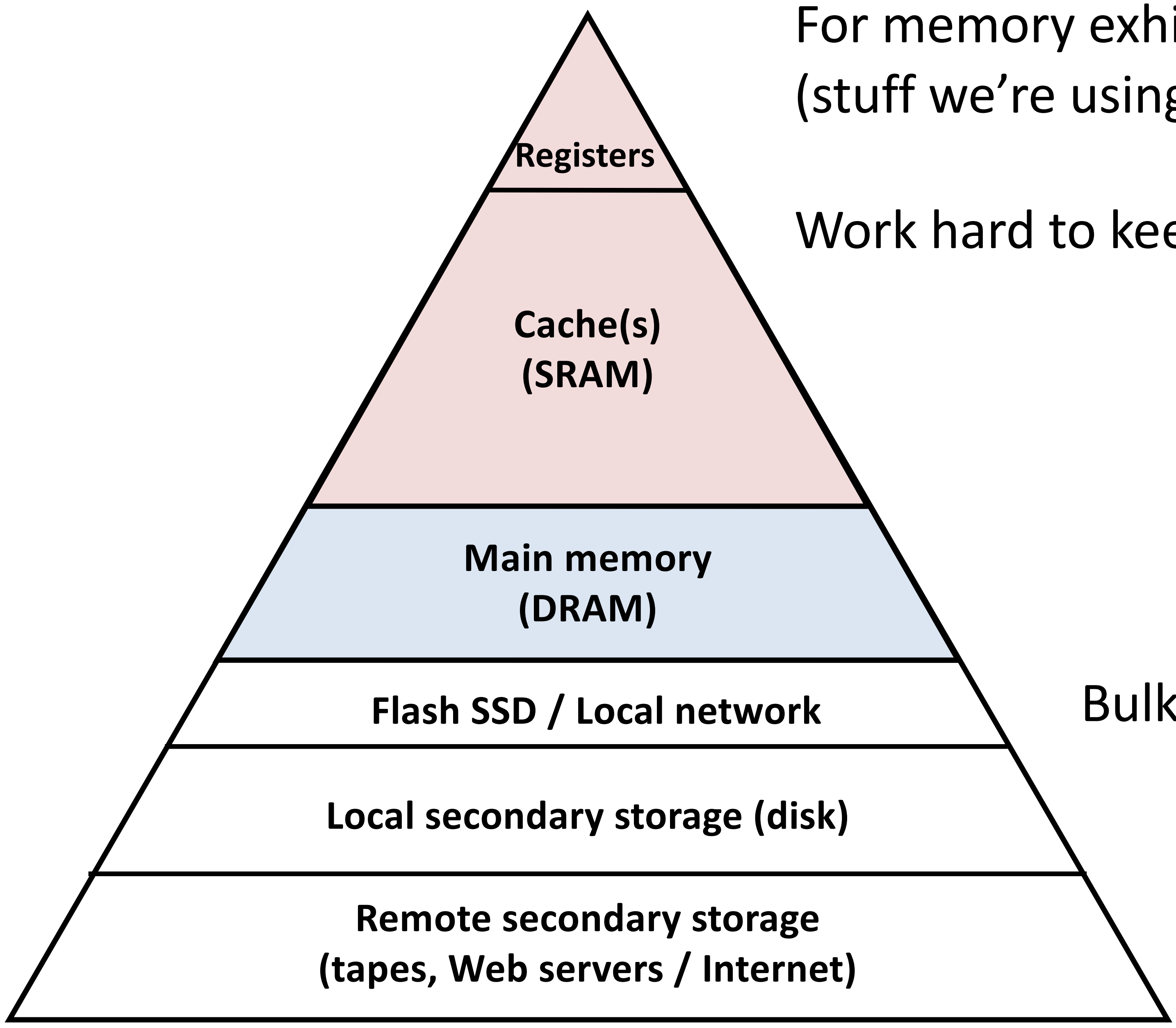
A. 1 temporal, 1 spatial
B. 1 temporal, 2 spatial
C. 2 temporal, 1 spatial
D. 2 temporal, 2 spatial
E. Some other number

## In the following code, how many examples are there of temporal / spatial locality? Where are they? (some of them)

```
int i;
int num = read_int_from_user();
int *array = create_random_array(num);
for (i = 0; i < num; i++) {
  printf("At index %d, value: %d", i, array[i]);
}
```

- Temporal
  - Array base access: for every iteration
  - i, num: access i and num on every iteration
  - printf: access the same instructions multiple times
  - printf: format string
- Spatial
  - printf: params to function call, and instructions come one after another
  - array elements
  - input parameters to a function call
  - instructions in the code above exhibit spatial locality

# Big Picture


Registers
Cache(s)
(SRAM)
Main memory
(DRAM)
Flash SSD / Local network
Local secondary storage (disk)
Remote secondary storage
(tapes, Web servers / Internet)

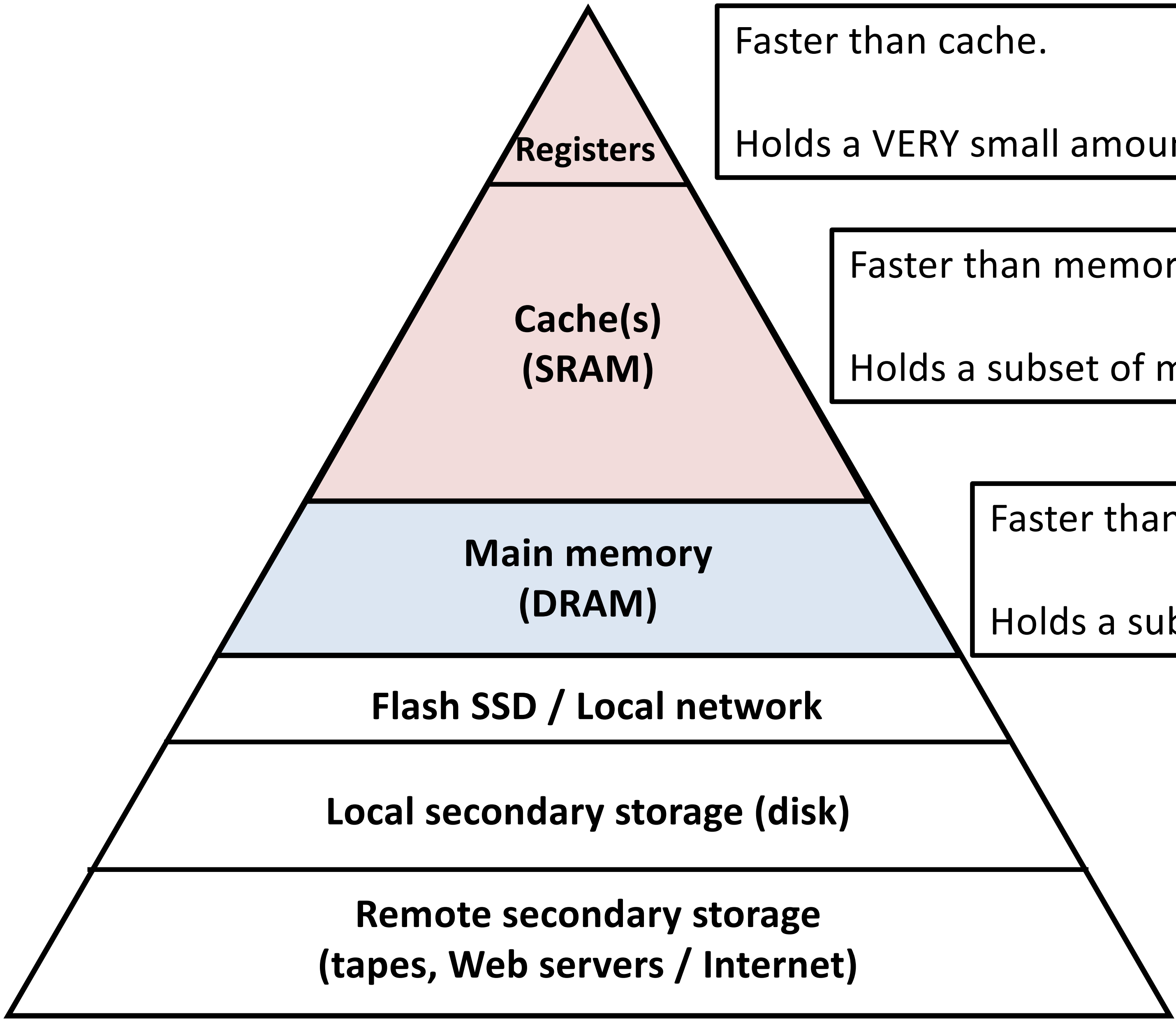
For memory exhibiting locality (stuff we're using / likely to use):

Work hard to keep them up here!

Bulk storage down here.

Move this up on demand.

## Big Picture


Registers
Cache(s)
(SRAM)
Main memory
(DRAM)
Flash SSD / Local network
Local secondary storage (disk)
Remote secondary storage
(tapes, Web servers / Internet)

Faster than cache.

Holds a VERY small amount.

Faster than memory. (On-chip hardware)

Holds a subset of memory.

Faster than disk.

Holds a subset of disk.

## Cache

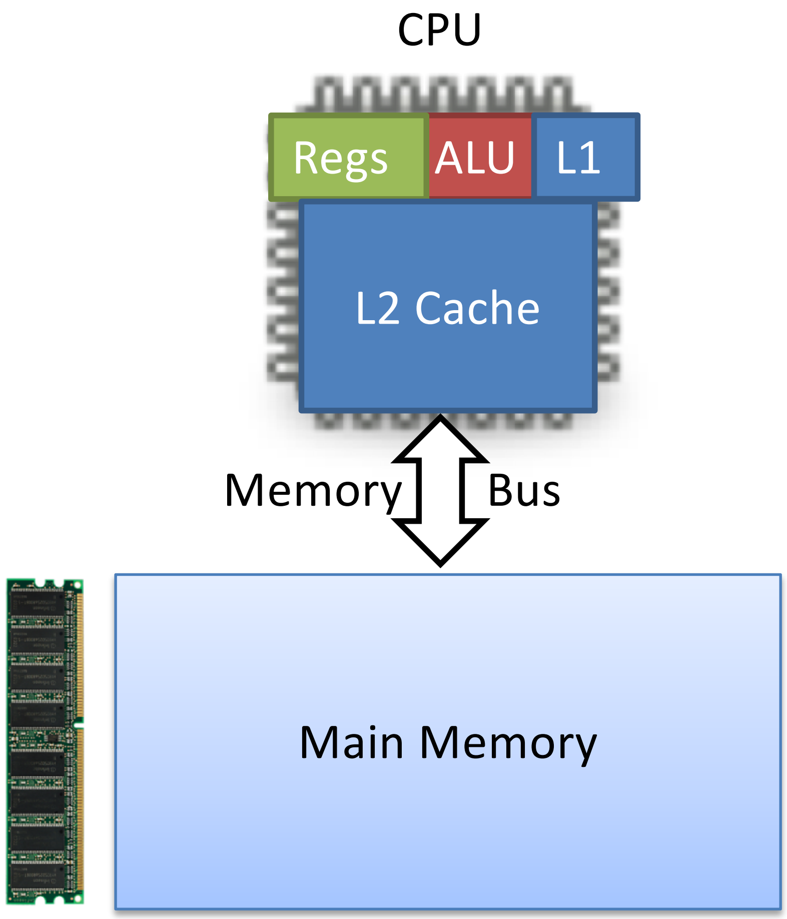
- In general: a storage location that holds a subset of a larger memory, faster to access

- CPU cache: an SRAM on-chip storage location that holds a subset of DRAM main memory (10-50x faster to access)

When I say "cache", assume this for now.

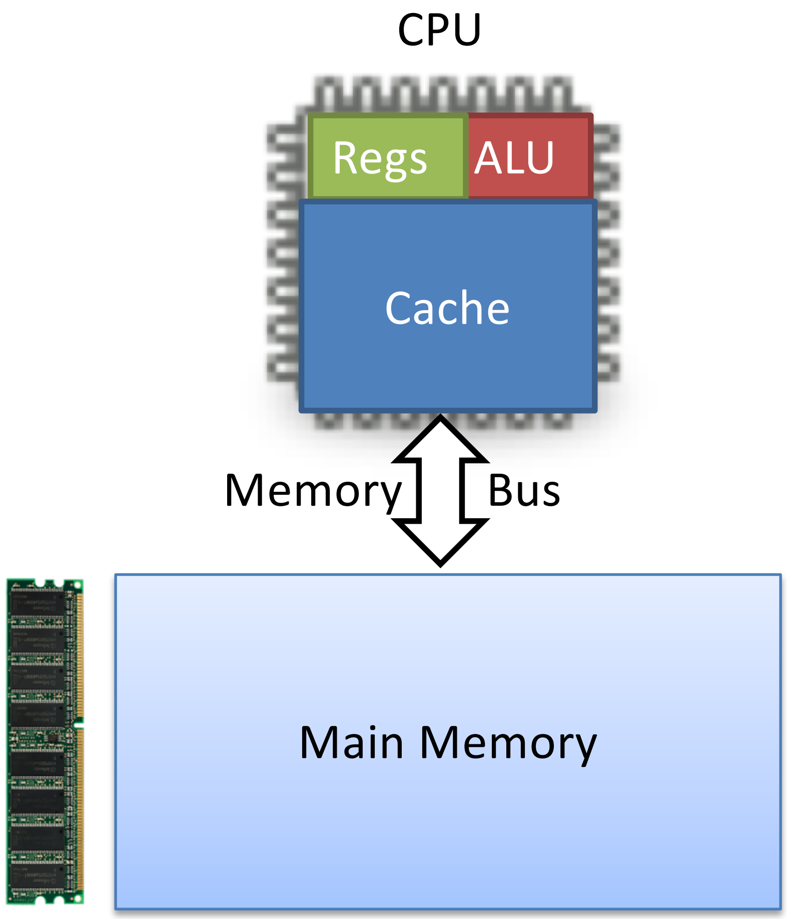
- Goal: choose the <u>right subset</u>, based on past locality, to achieve our abstraction

# Cache Basics



- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- Same rules apply:
  - L1 subset of L2

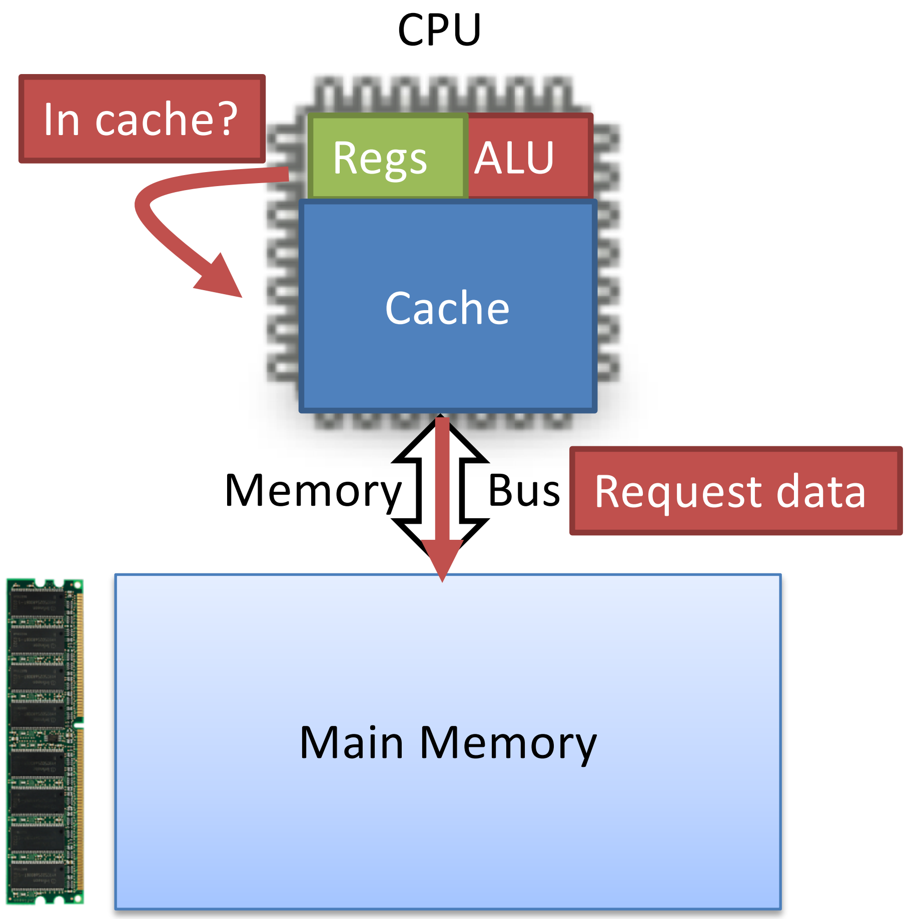
# Cache Basics



Cache is a subset of main memory.
(Not to scale, memory much bigger!)

- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- We'll assume one cache (same principles)

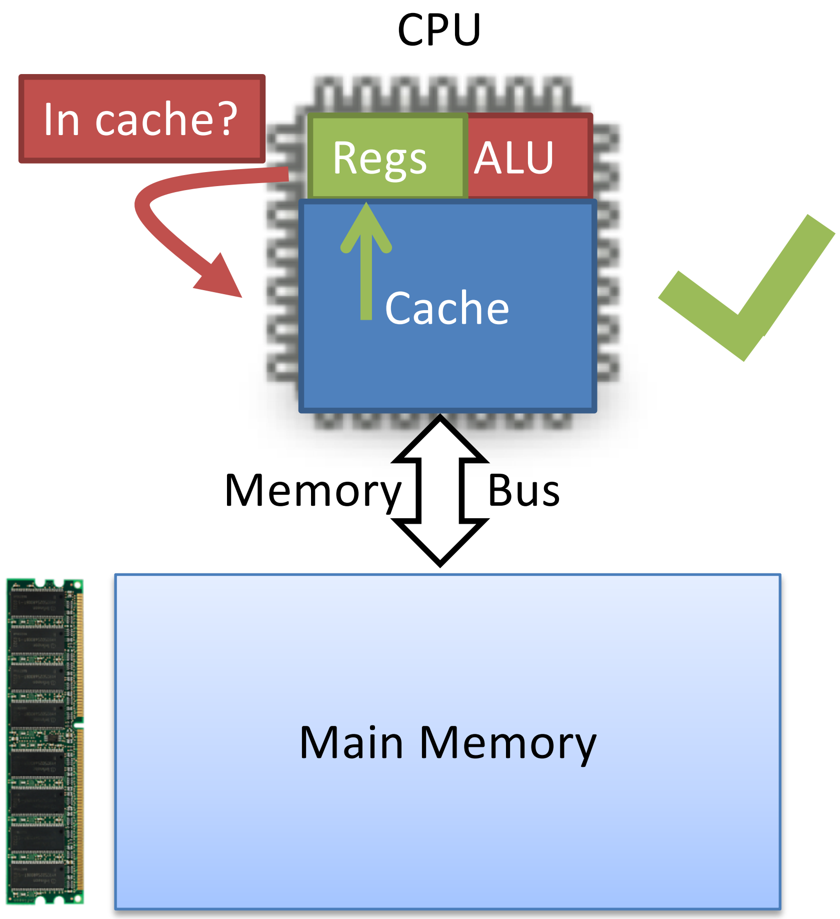
# Cache Basics: Read from memory



- In parallel:
  - Issue read to memory
  - Check cache

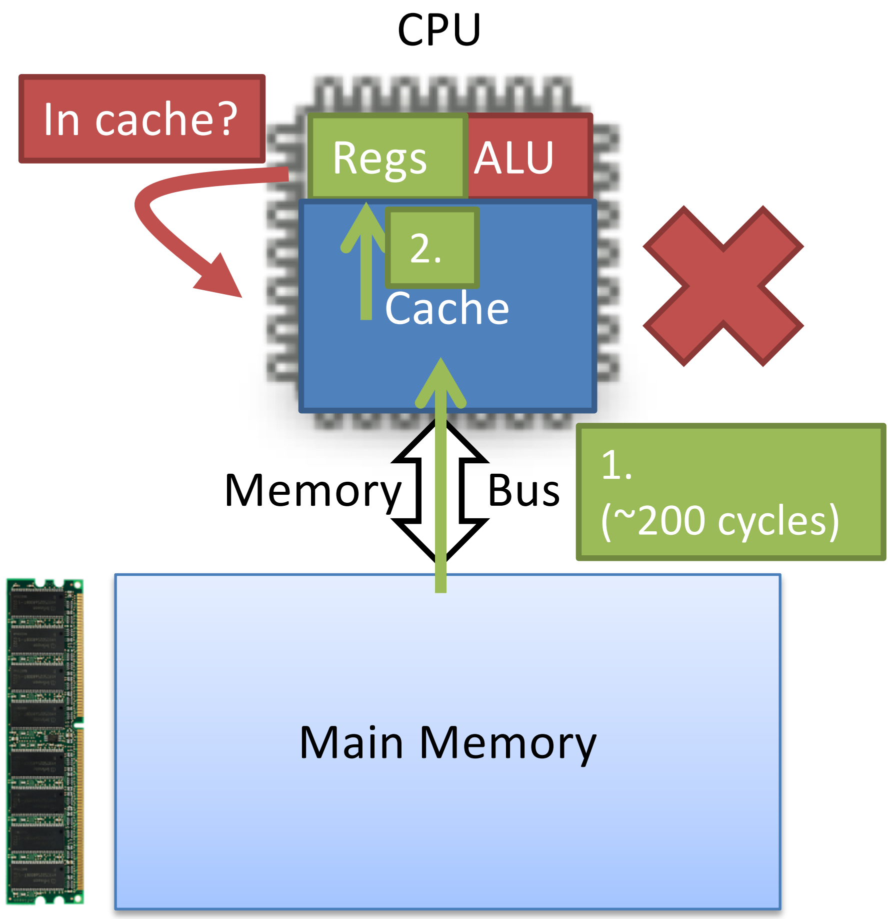
MB
BATTLESHIP
Combine Strategy And Luck
In This Exciting Naval Action Game
Ages 8 to Adult
2 Players
SET UP YOUR FLEET
CALL OUT THE SHOTS
MARK THE HITS
SINK YOUR OPPONENT'S FLEET

## Cache Basics: Read from memory



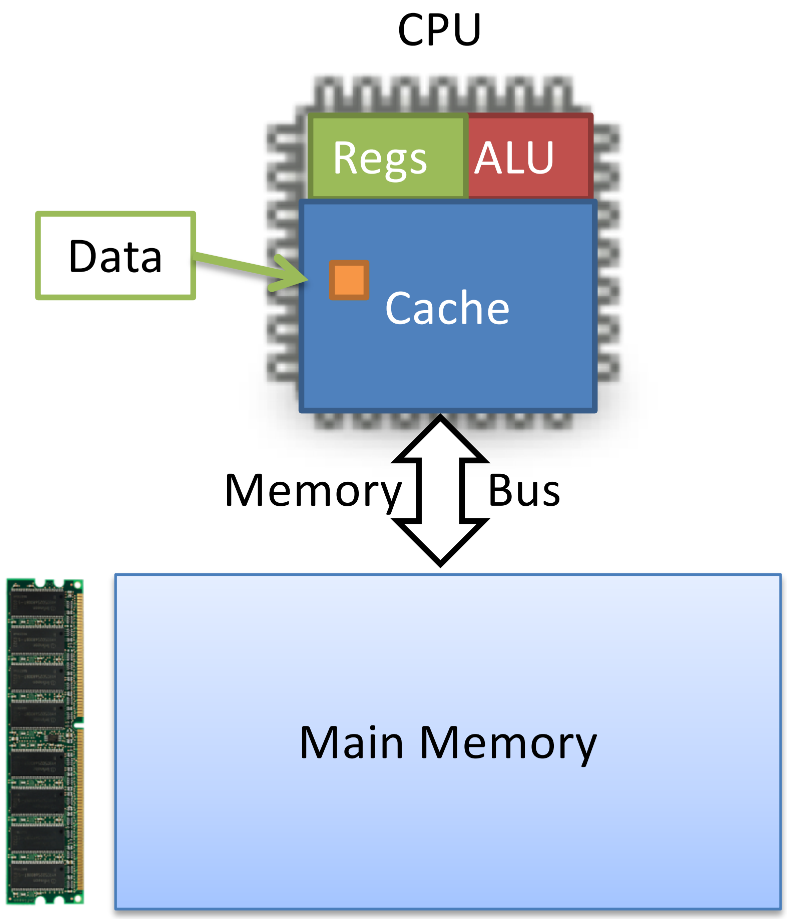
- In parallel:
  - Issue read to memory
  - Check cache
- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory

# Cache Basics: Read from memory



- In parallel:
  - Issue read to memory
  - Check cache
- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory
- Data not in cache (miss):
  1. Load cache from memory (might need to evict data)
  2. Send to register

# Cache Basics: Write to memory



- Assume data already cached
  – Otherwise, bring it in like read

1. Update cached copy.

2. Update memory?

## When should we copy the written data from cache to memory? Why?

A. Immediately update the data in memory when we update the cache.

B. Update the data in memory when we evict the data from the cache.

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)

## When should we copy the written data from cache to memory? Why?

A. Immediately update the data in memory when we update the cache. (“Write-through”)

B. Update the data in memory when we evict the data from the cache. (“Write-back”)

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)

## Cache Basics: Write to memory

- Both options (write-through, write-back) viable
- write-though: write to memory immediately
  - simpler, accesses memory more often (slower)
- write-back: only write to memory on eviction
  - complex (cache inconsistent with memory)
  - potentially reduces memory accesses (faster)

## Cache Basics: Write to memory

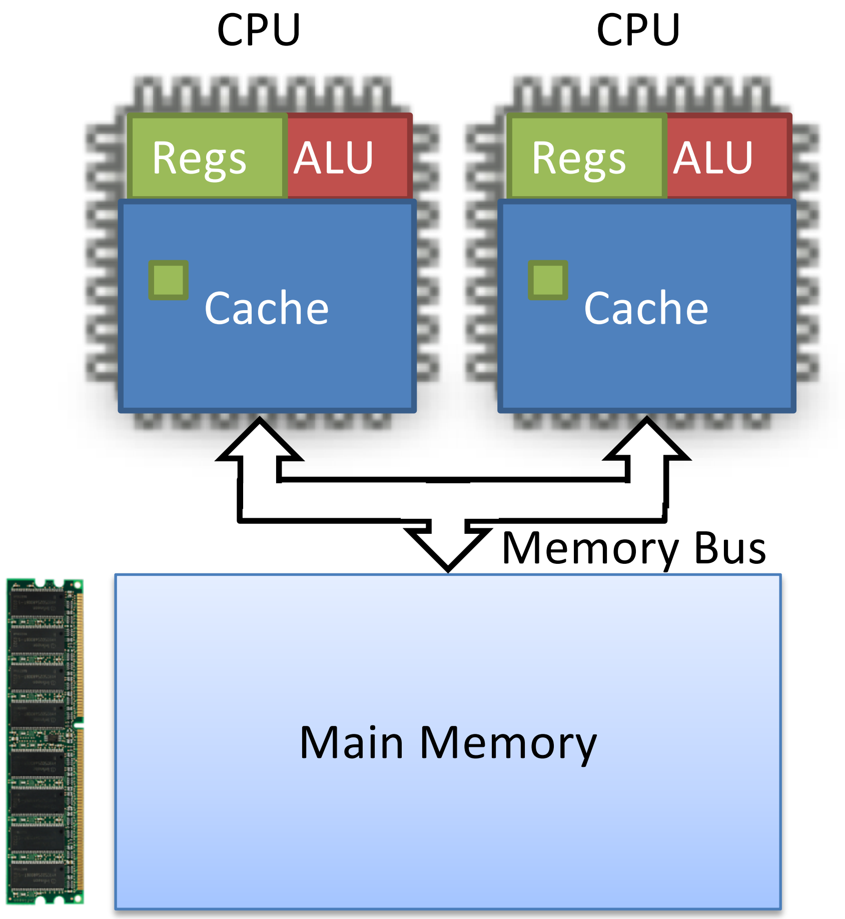
- Both options (write-through, write-back) viable

- write-though: write to memory immediately
  - simpler, accesses memory more often (slower)

- write-back: only write to memory on eviction
  - complex (cache inconsistent with memory)
  - potentially reduces memory accesses (<u>faster</u>)

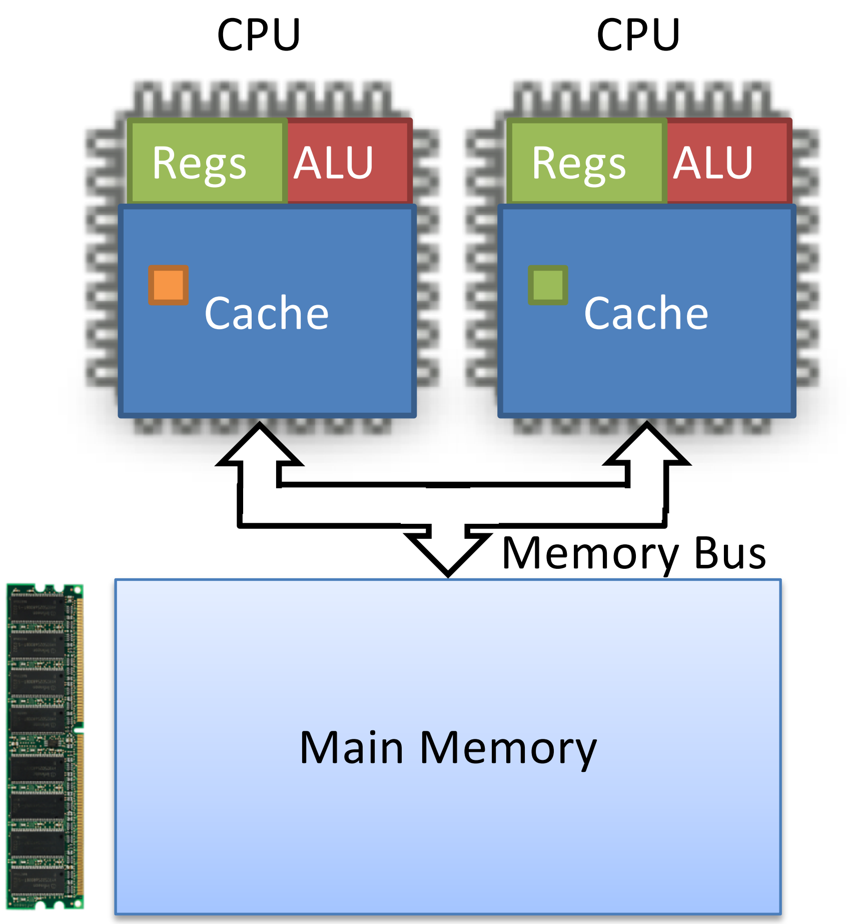
Sells better.
Servers/Desktops/Laptops ↗

# Bonus slides: Cache Coherence



- Keeping multiple cores' memory consistent

# Bonus slides: Cache Coherence



- Keeping multiple cores' memory consistent
- If one core updates data
  - Copy data directly from one cache to the other.
  - Avoid (slower) memory
- Lots of HW complexity here. (beyond 31)

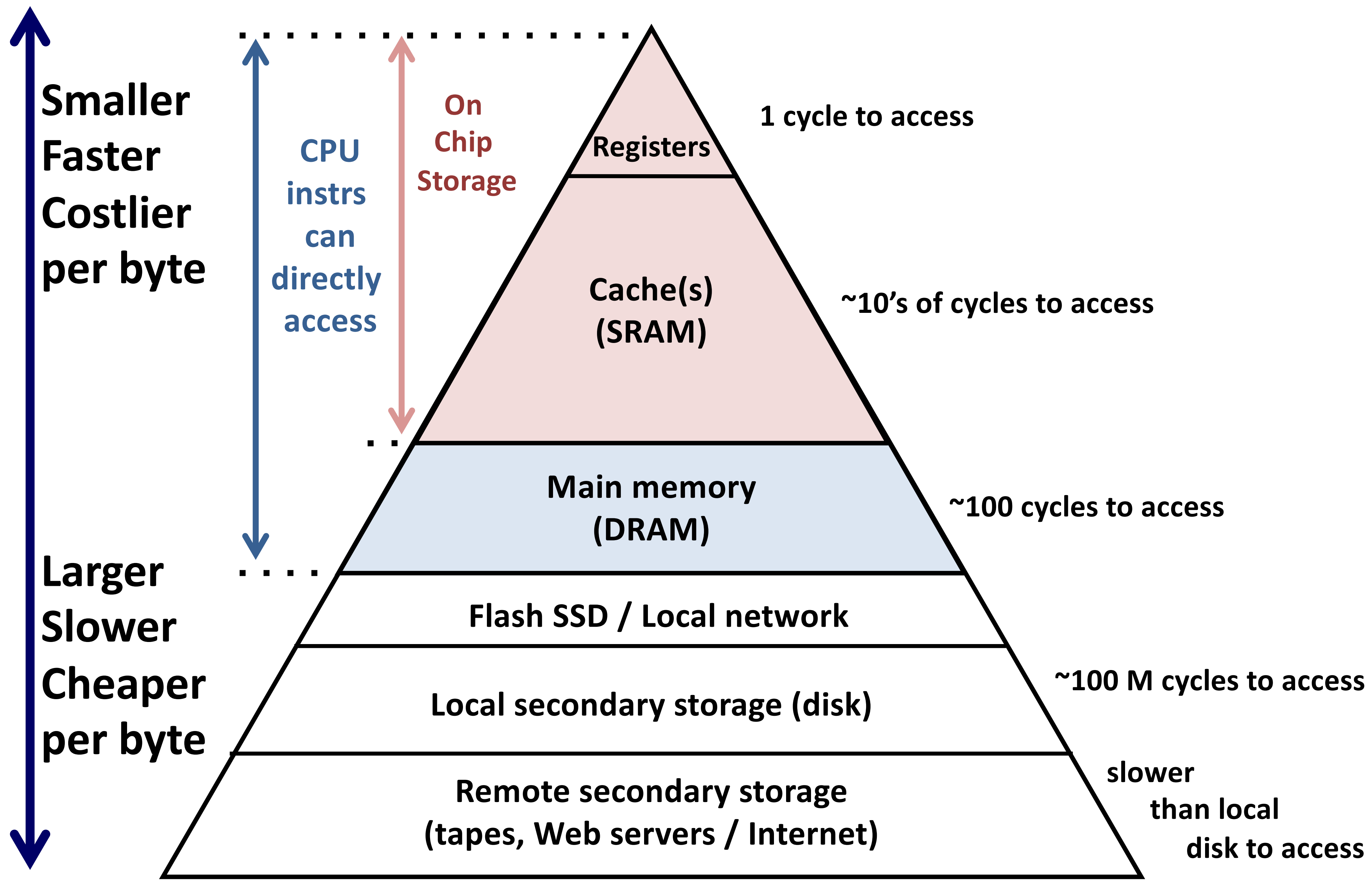
## Up next:

- Cache details
- How cache is organized
  - finding data
  - storing data
- How cached subset is chosen (eviction)

## Abstraction Goal

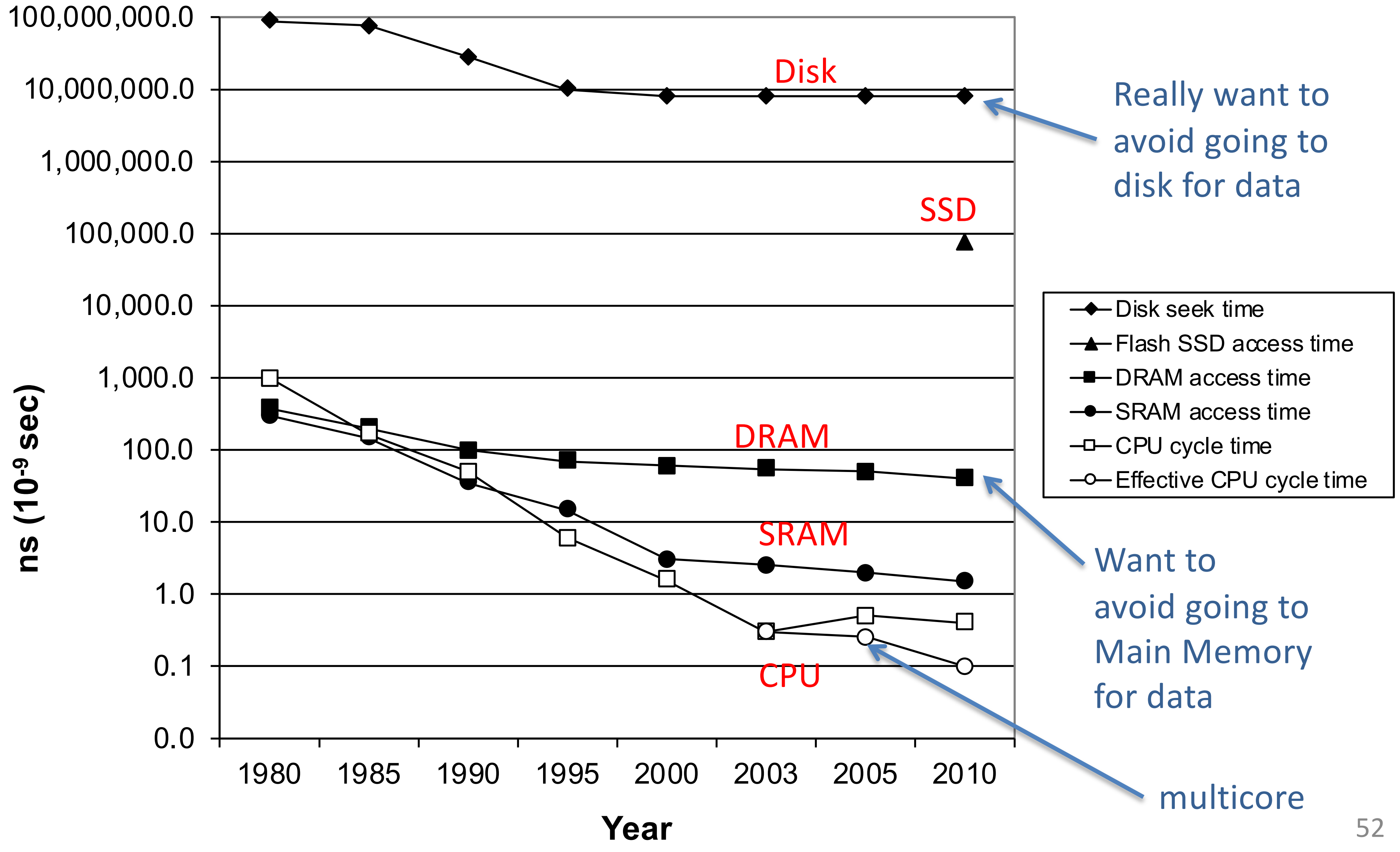
- Reality: There is no one type of memory to rule them all!
- Abstraction: hide the complex/undesirable details of reality.
- Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.

# The Memory Hierarchy


Smaller
Faster
Costlier
per byte
CPU instrs can directly access
On Chip Storage
Registers
1 cycle to access
Cache(s) (SRAM)
~10's of cycles to access
Main memory (DRAM)
~100 cycles to access
Larger
Slower
Cheaper
per byte
Flash SSD / Local network
Local secondary storage (disk)
~100 M cycles to access
Remote secondary storage (tapes, Web servers / Internet)
slower than local disk to access

# Data Access Time over Years

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



## Recall

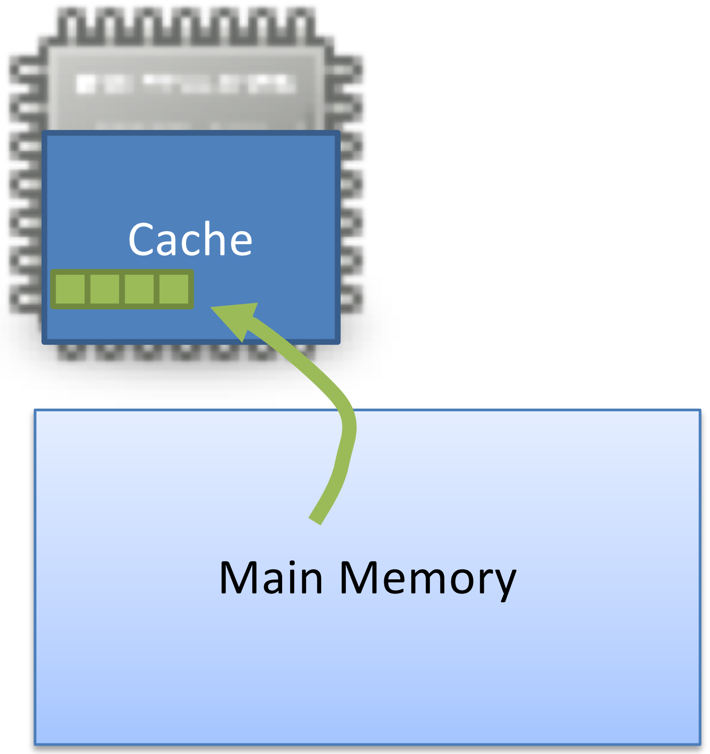
- A cache is a smaller, faster memory, that holds a subset of a larger (slower) memory
- We take advantage of <u>locality</u> to keep data in cache as often as we can!
- When accessing memory, we check cache to see if it has the data we’re looking for.

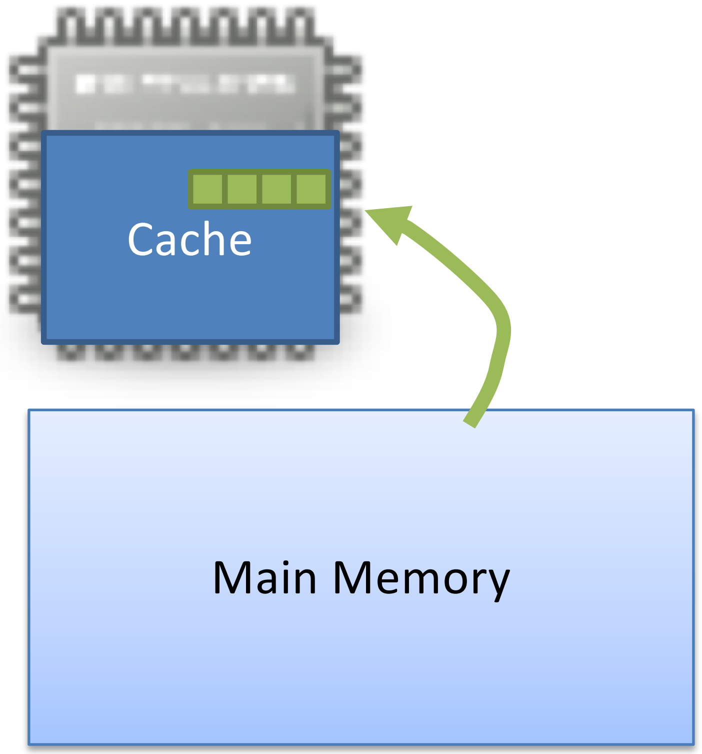
## Why we miss…

- 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

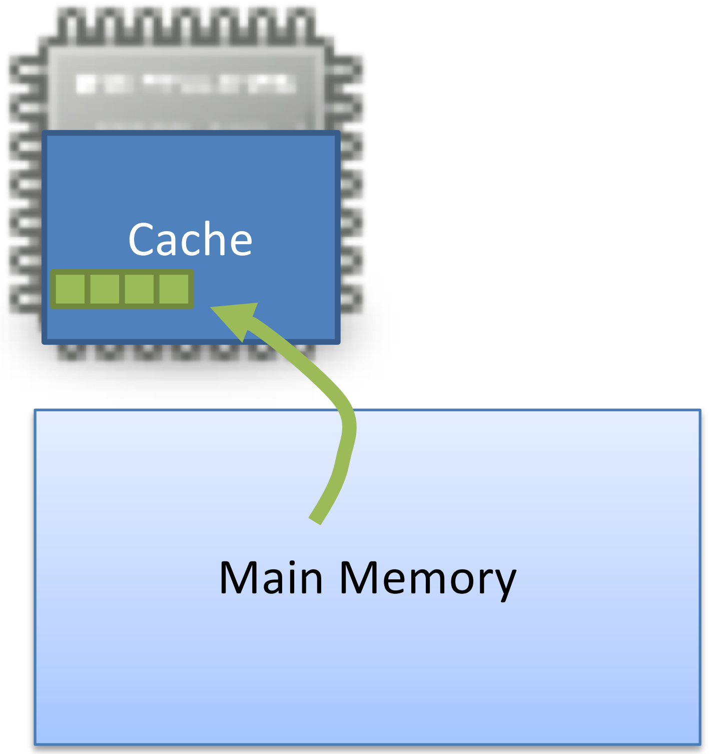
- Lot’s of characteristics to consider:
  - Where should data be stored in the cache?

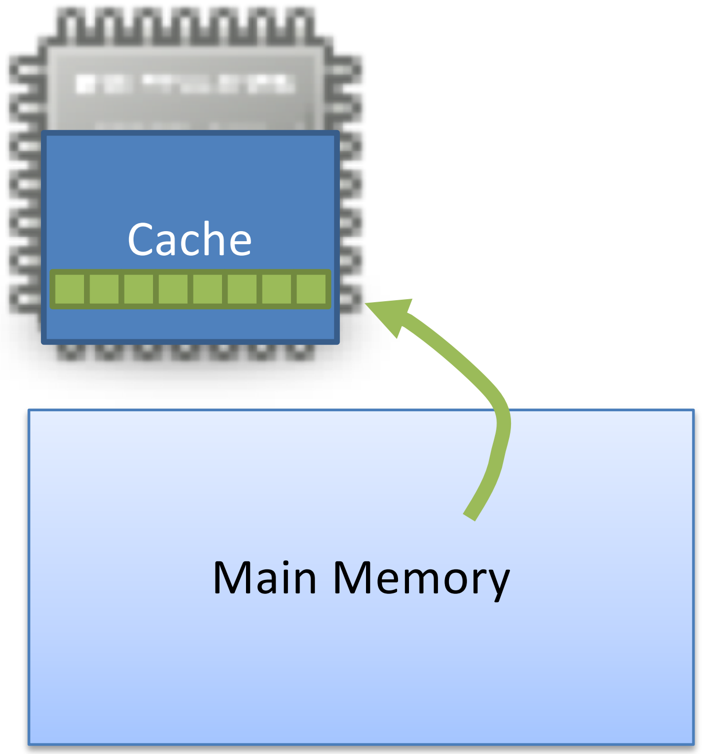




## Cache Design

- 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

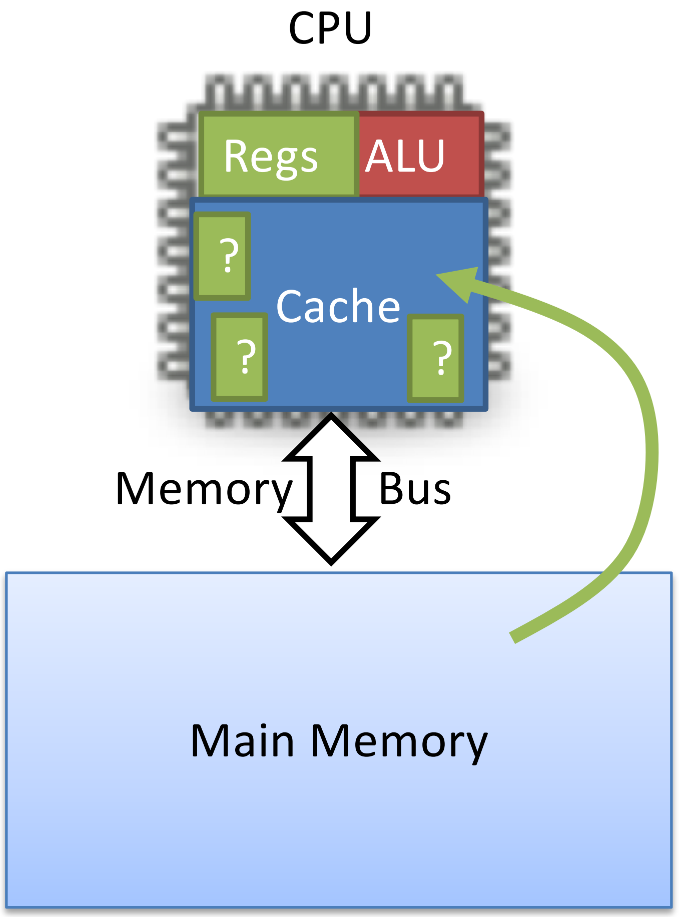
## Suppose the CPU asks for data, it's not in cache.We need to move in into cache from memory. Where in the cache should it be allowed to go?

A. In exactly one place.

B. In a few places.

C. In most places, but not all.

D. Anywhere in the cache.



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

A. Better temporal locality

B. Better spatial locality

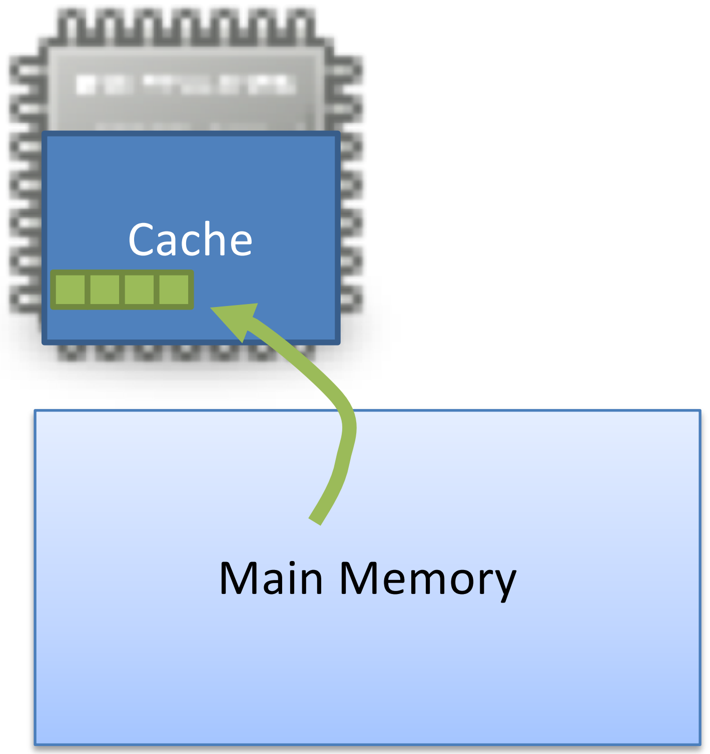
C. Fewer misses (better hit rate)

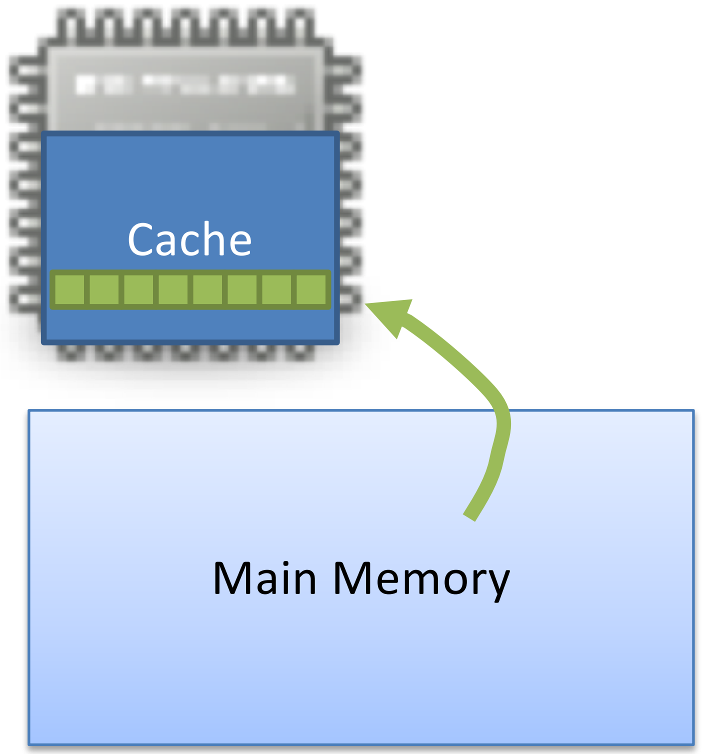
D. More misses (worse hit rate)

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

- 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.
   - <u>If we knew this, we wouldn’t be looking for it!</u>

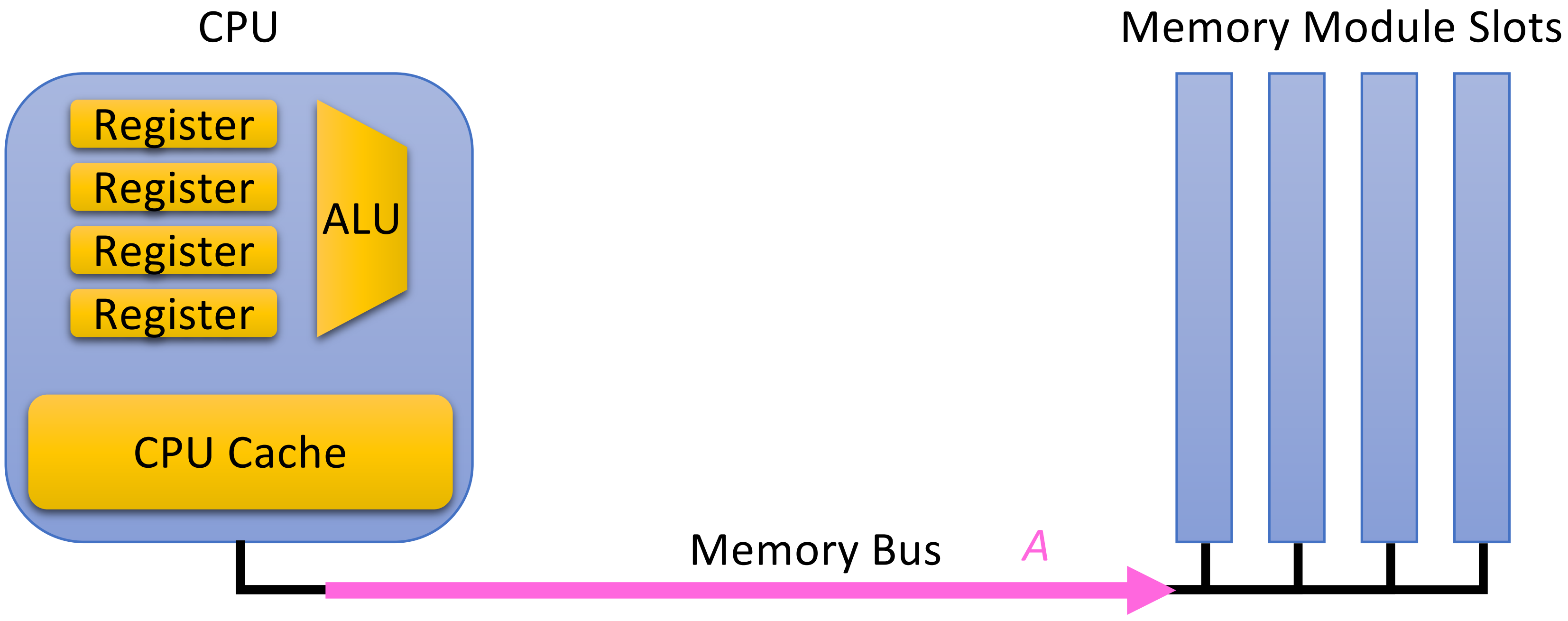
C. The size of the data.

D. Some other aspect of the data.

## Recall: Memory Reads

CPU places address A on the memory bus.

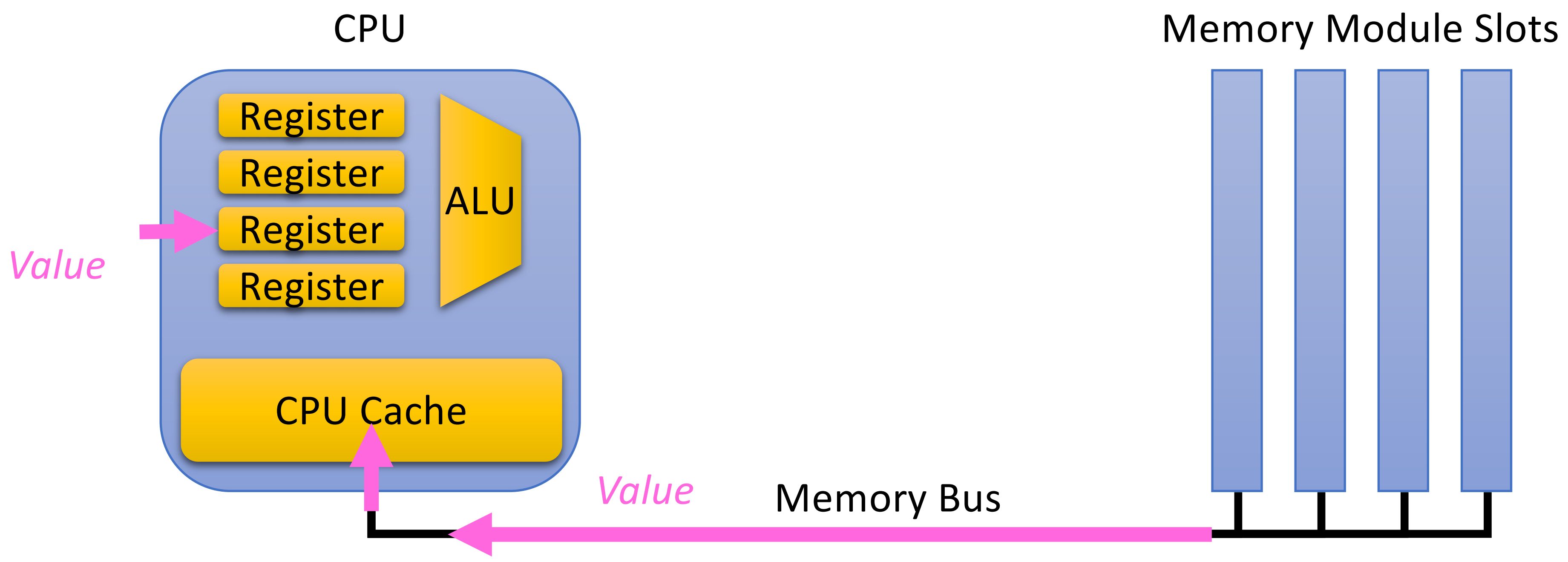
Load operation: `movl (A), %eax`



## 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

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 <u>not</u> 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 | | | | |

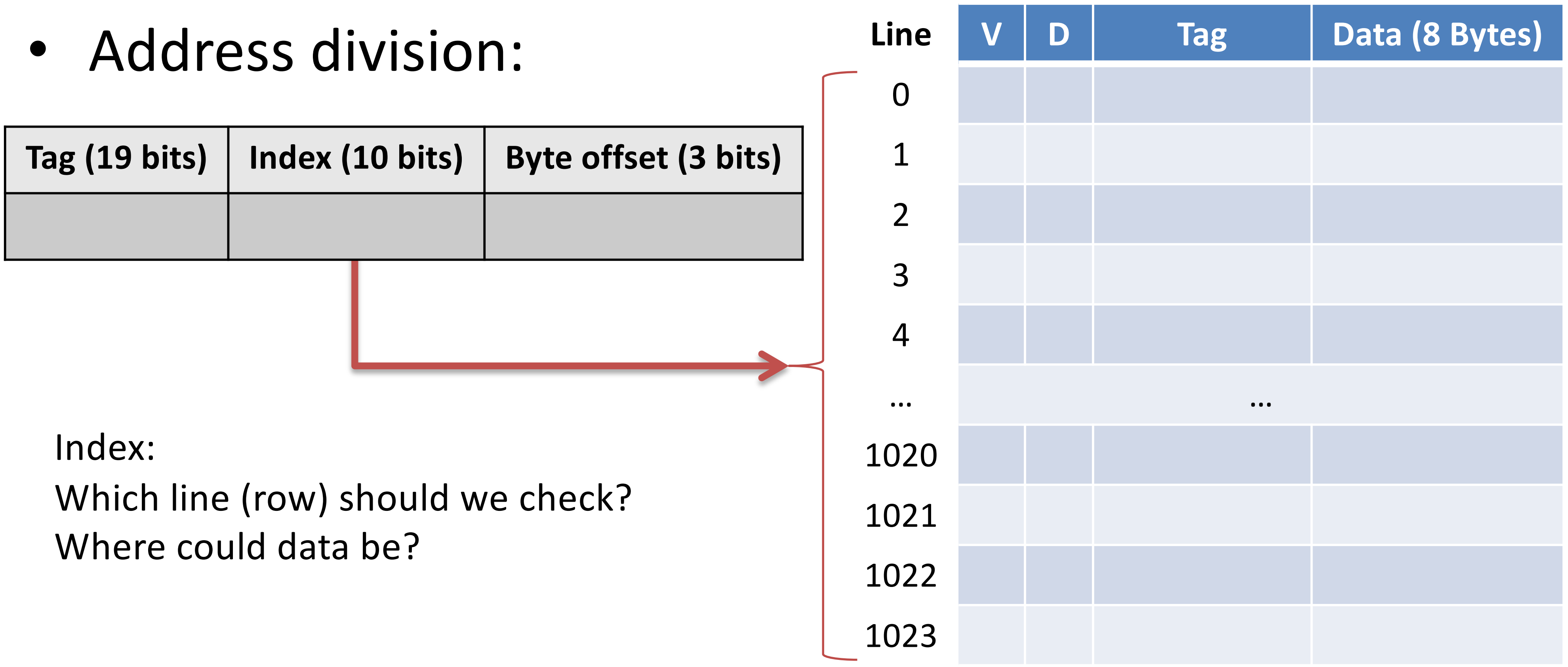
# Direct-Mapped

- 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

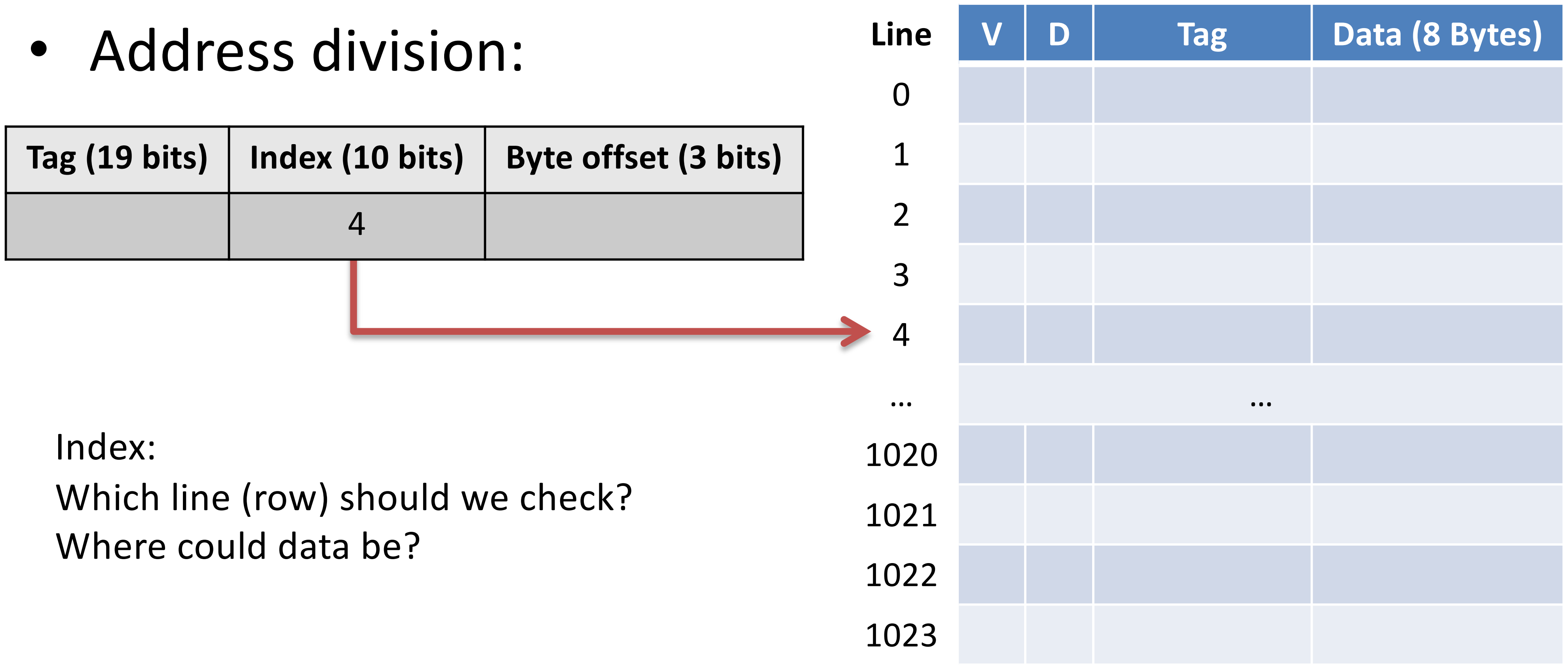
- Address division:



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

# Direct-Mapped

- Address division:



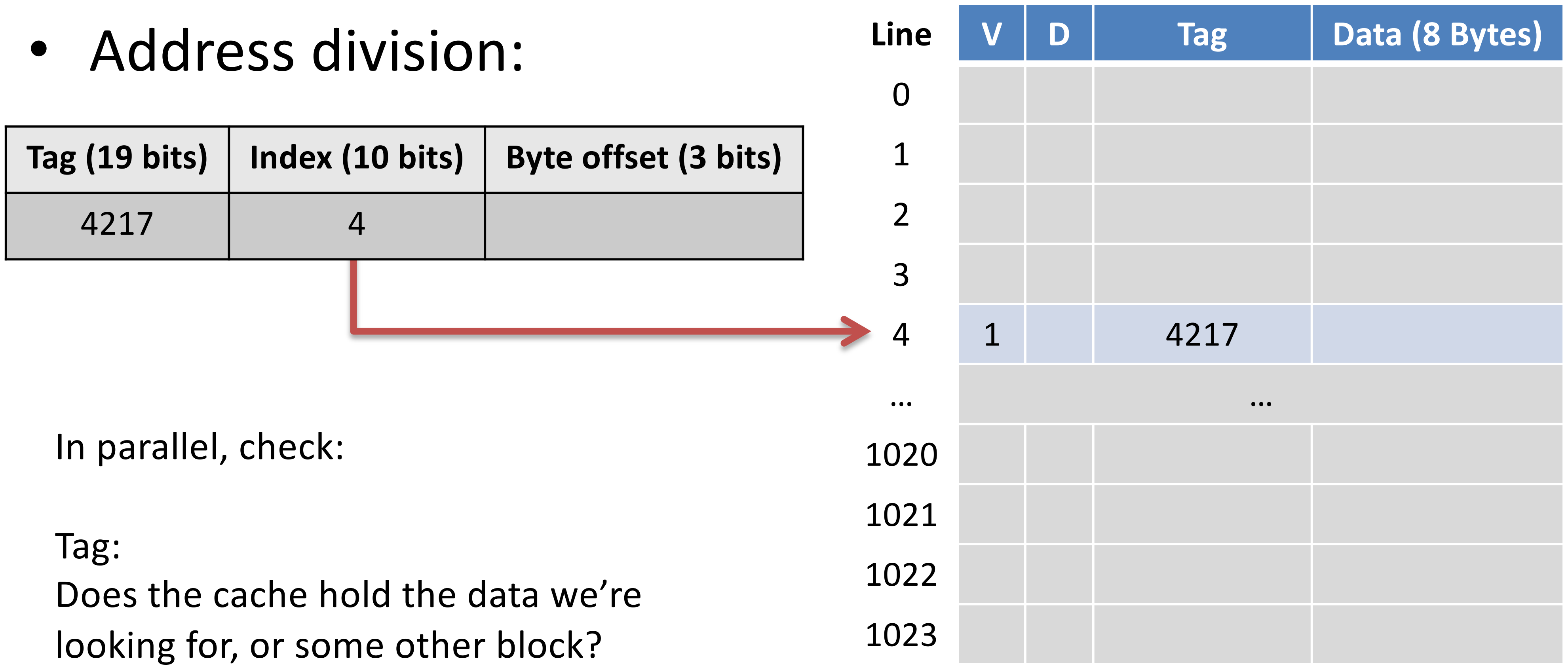
| Tag (19 bits) | Index (10 bits) | Byte offset (3 bits) |
|---|---|---|
|  | 4 |  |

| 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).

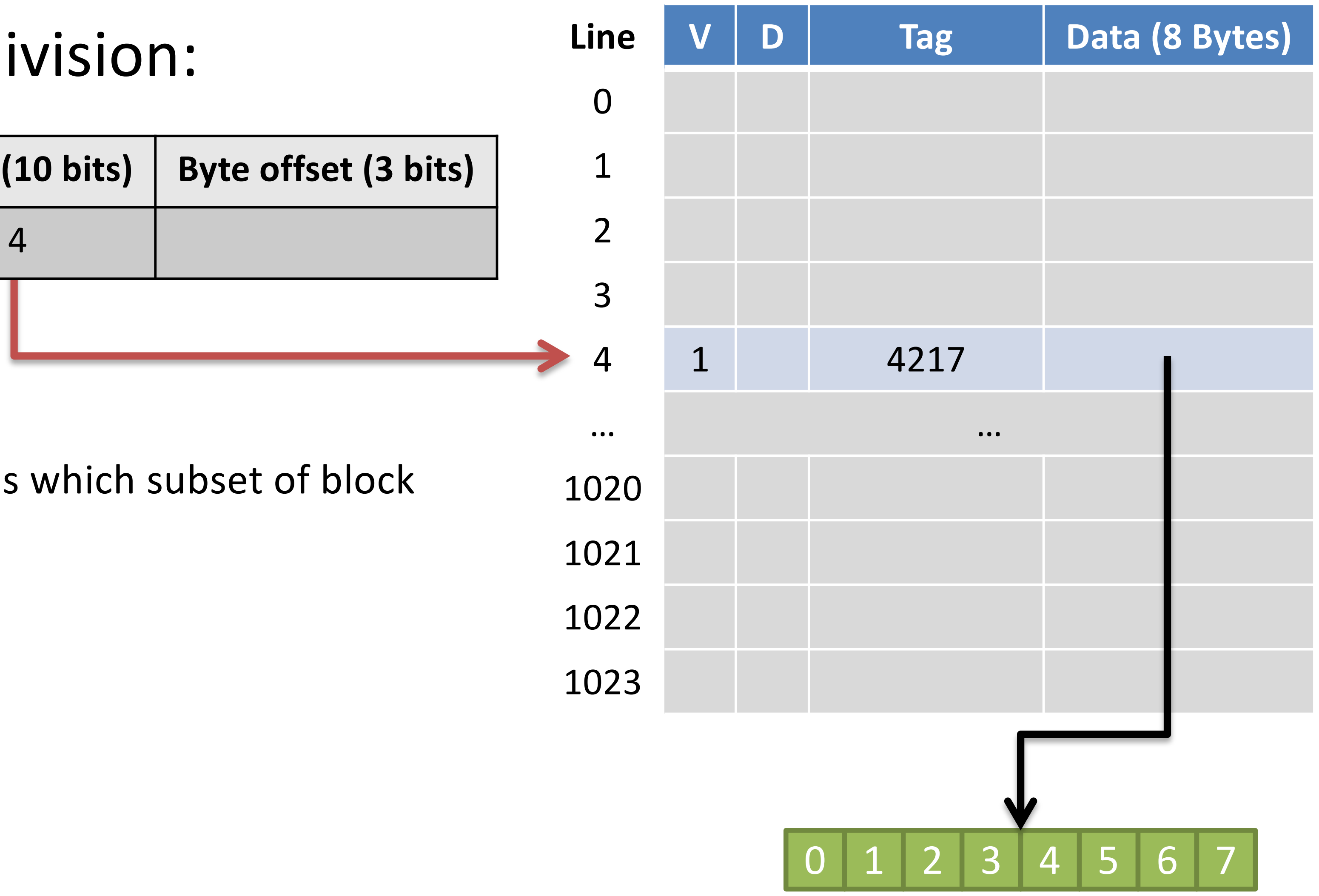
If tag doesn't match,
or line is invalid, it's a miss!

# 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.

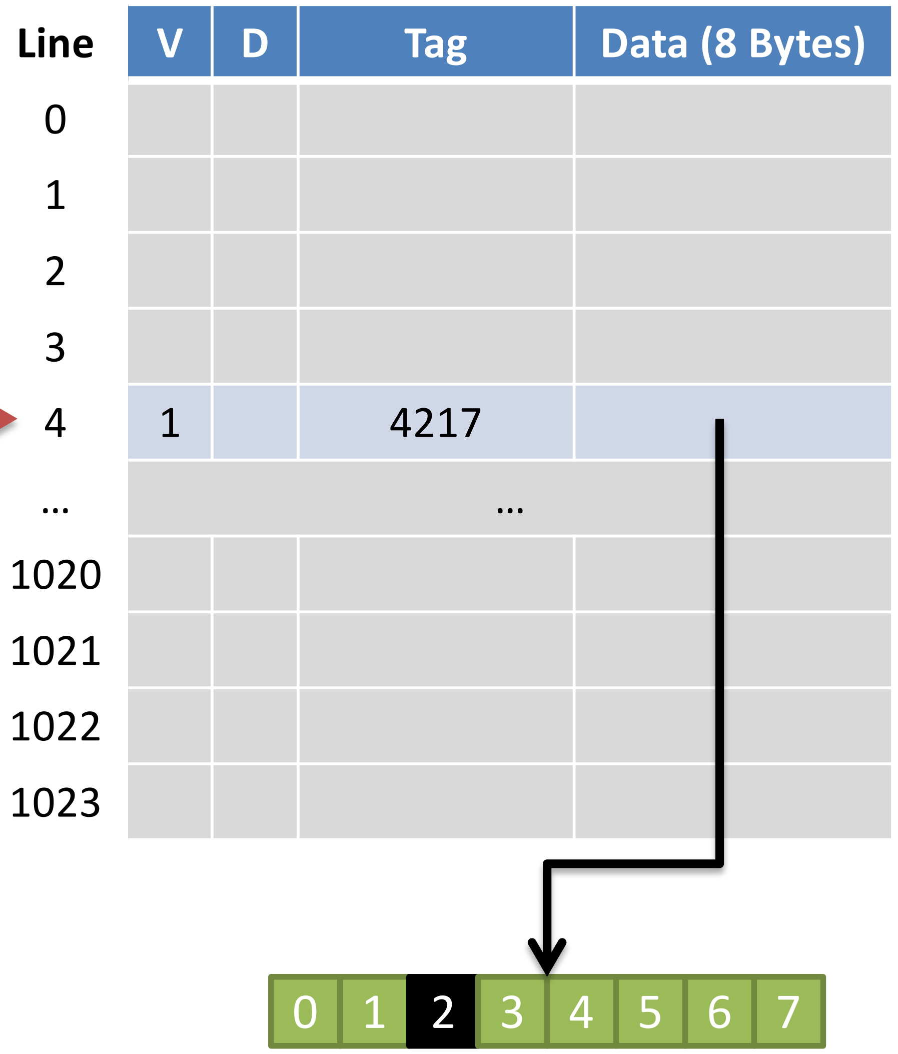


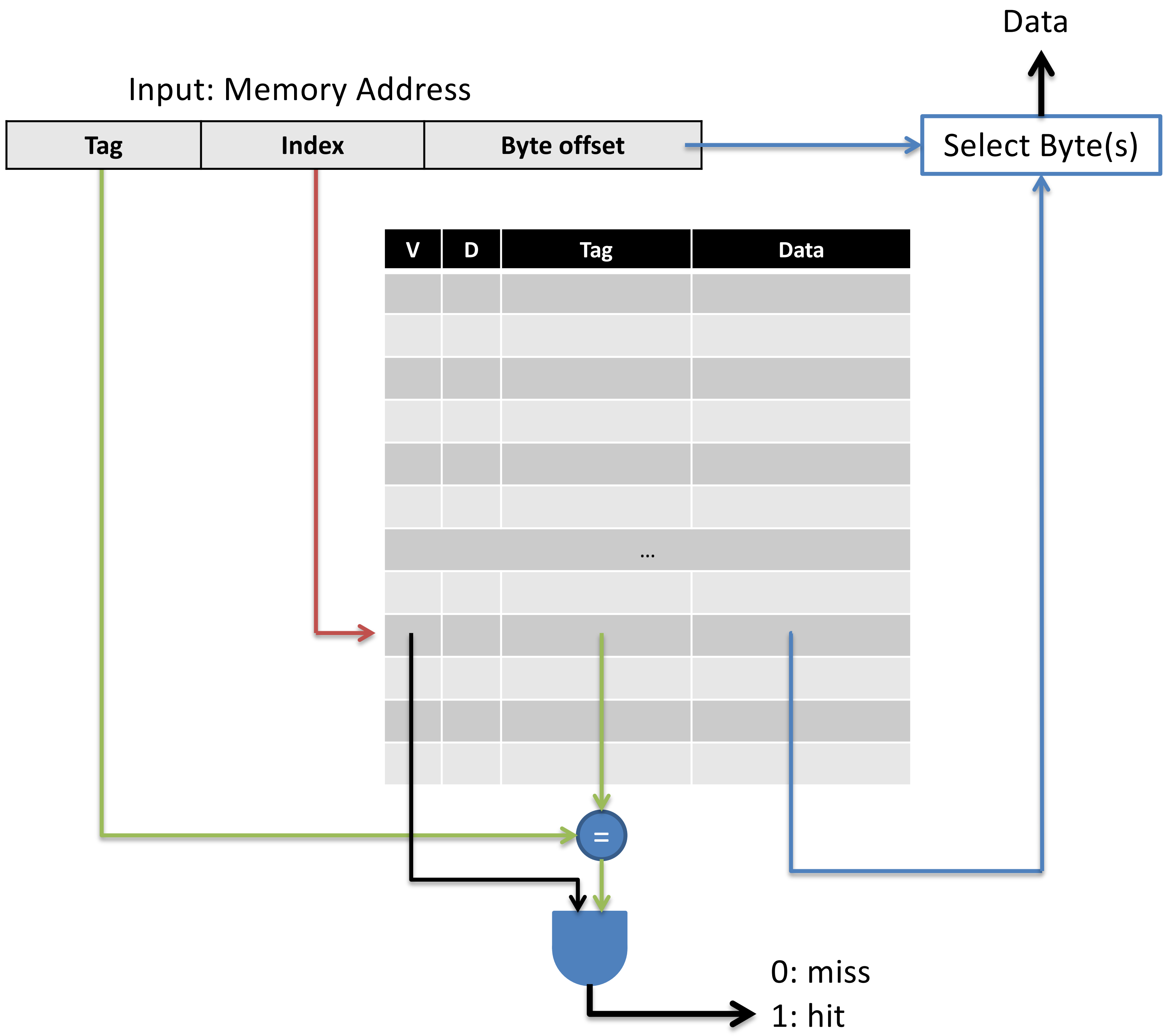
# Direct-Mapped

- Address division:

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

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 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 | | | | |

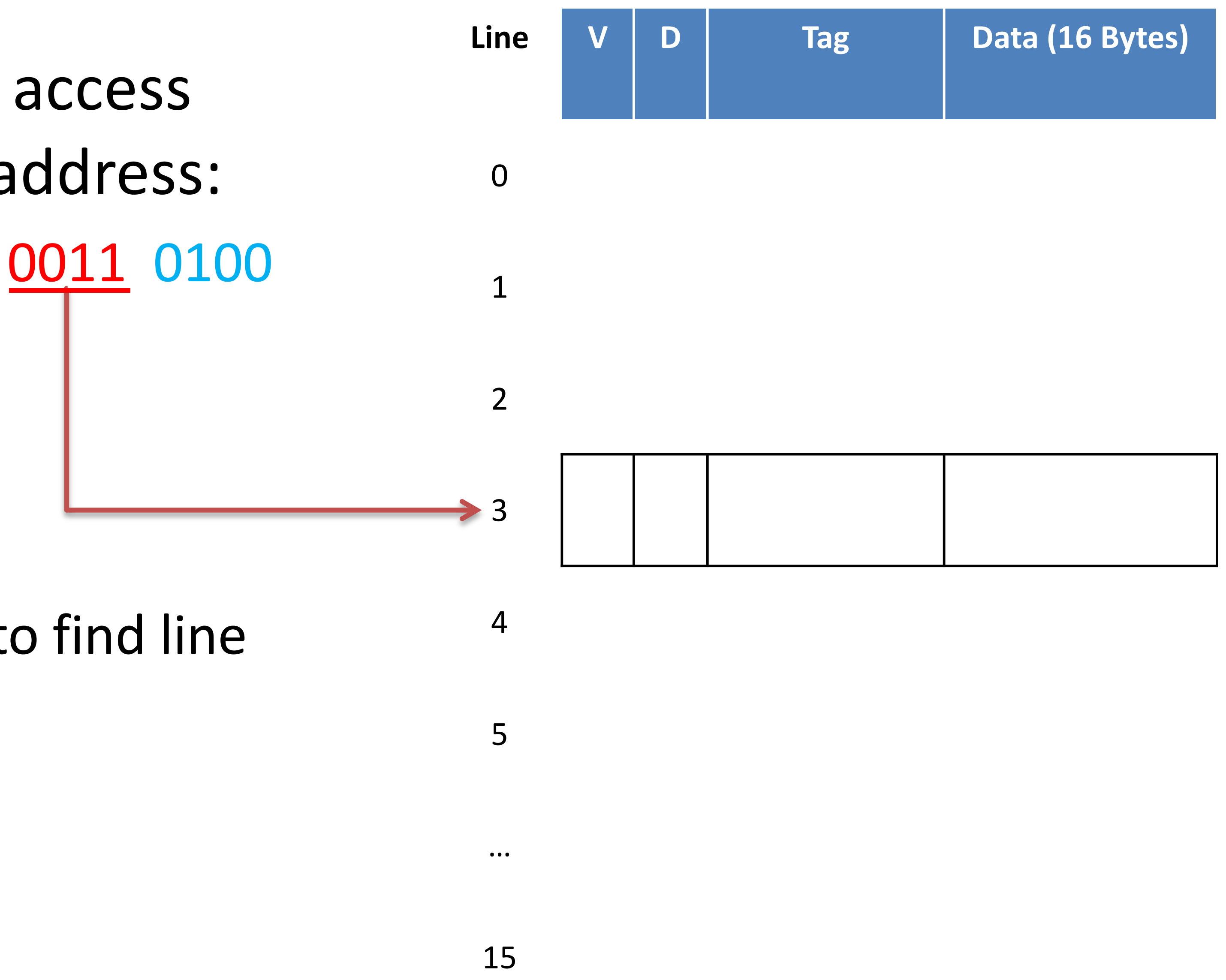
# Direct-Mapped Example

- Let's say we access memory at address:
  - 01101011 <u>0011</u> 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 | | | | |

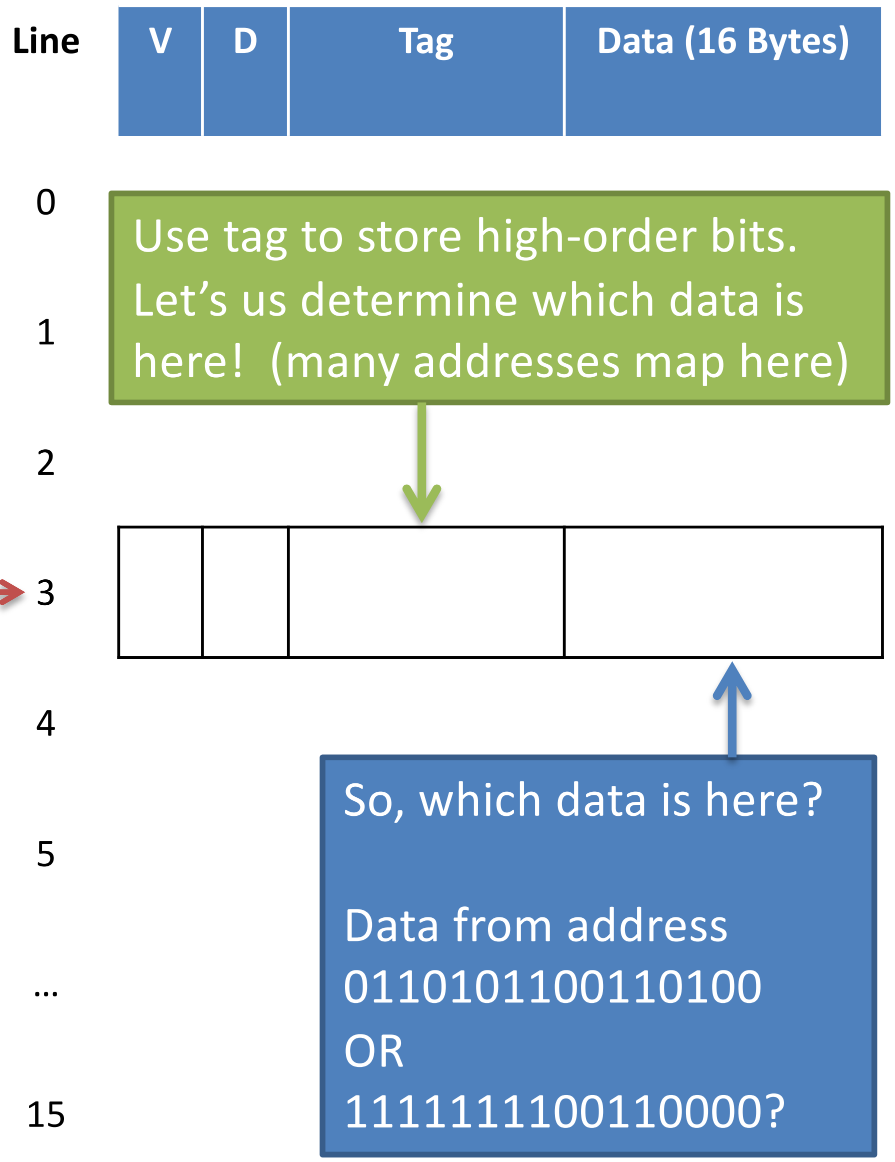
# Direct-Mapped Example

- 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



Use tag to store high-order bits. Let’s us determine which data is here! (many addresses map here)

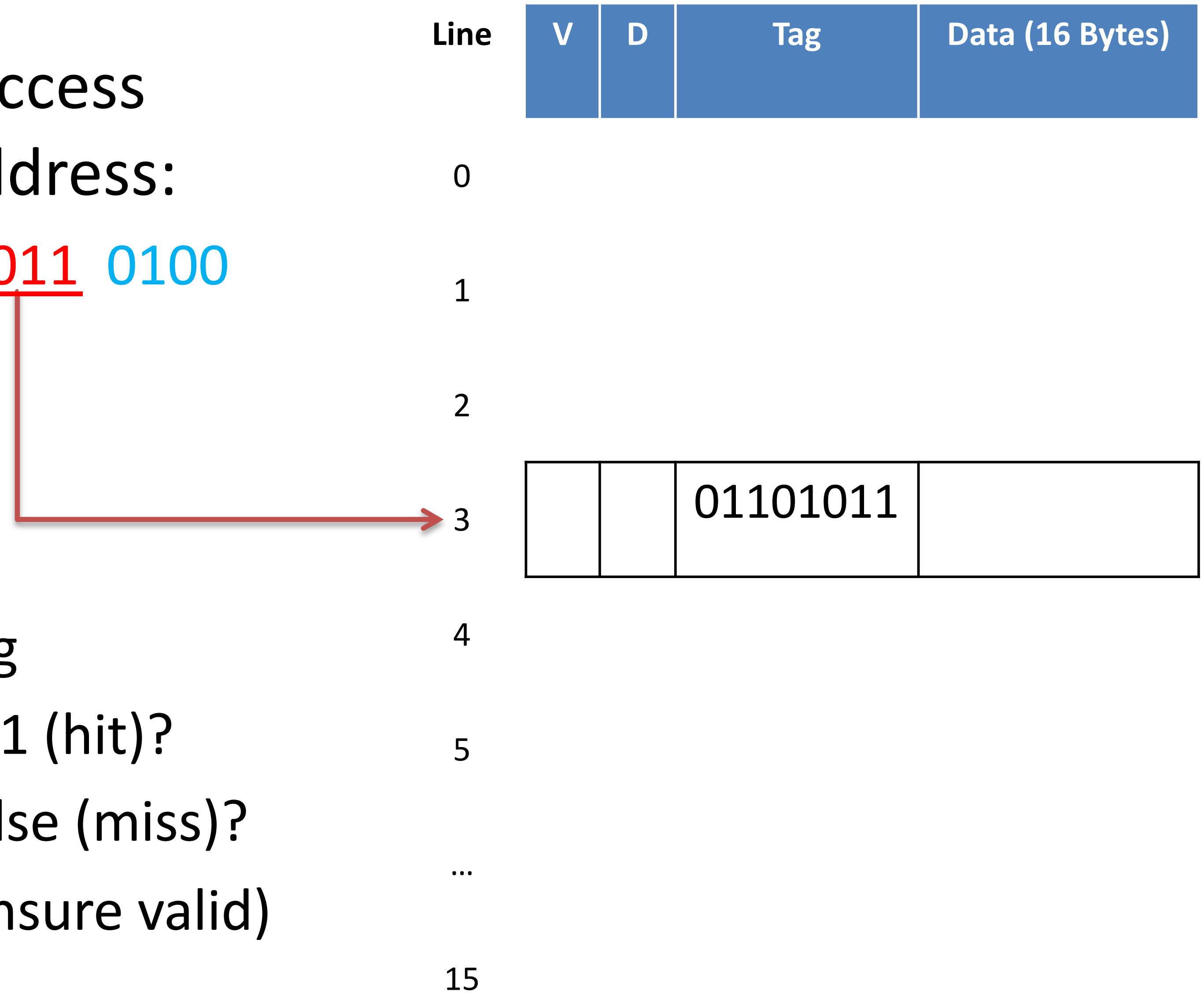
So, which data is here?

Data from address 0110101100110100 OR 1111111100110000?

# 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

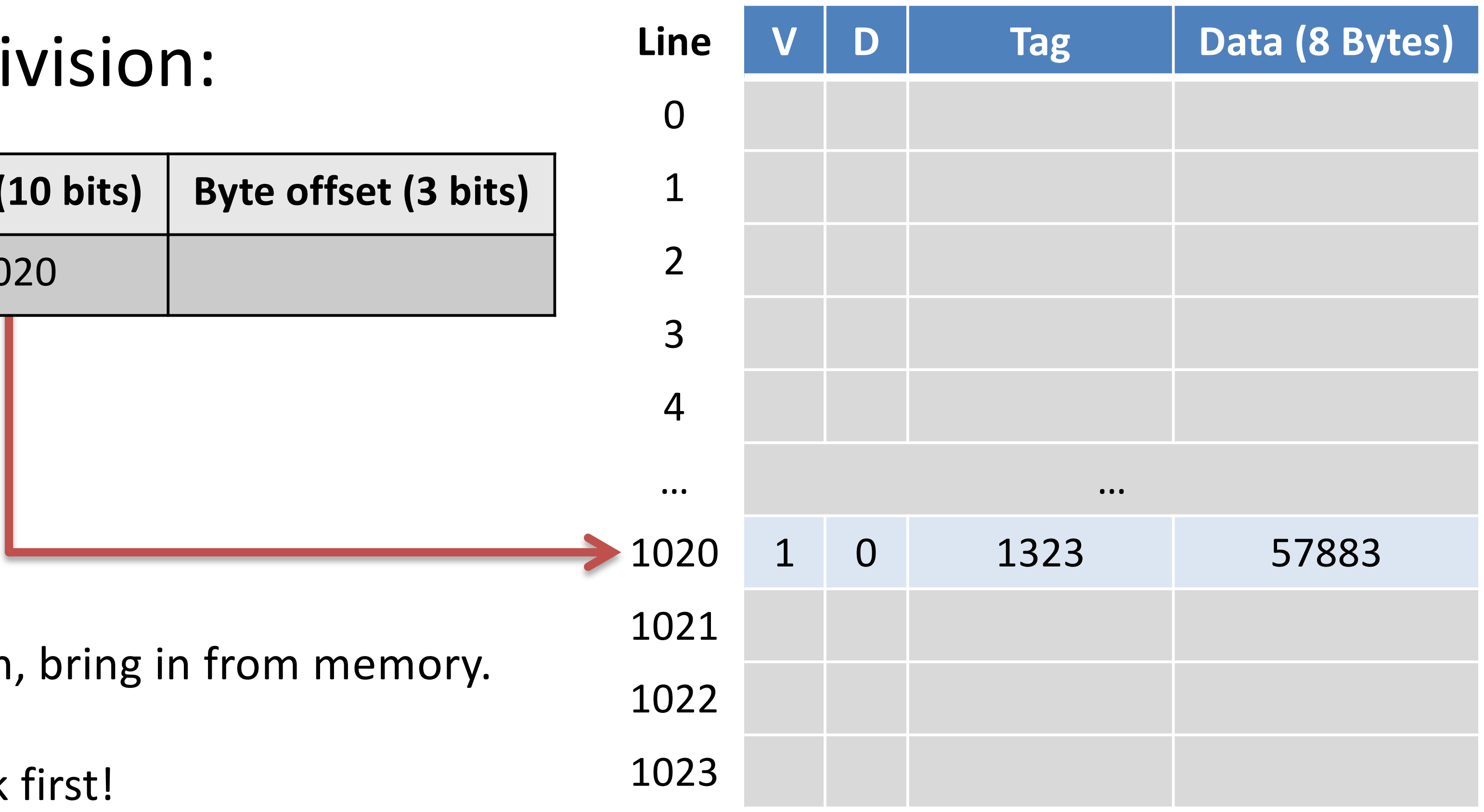
- Address division:

| 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!

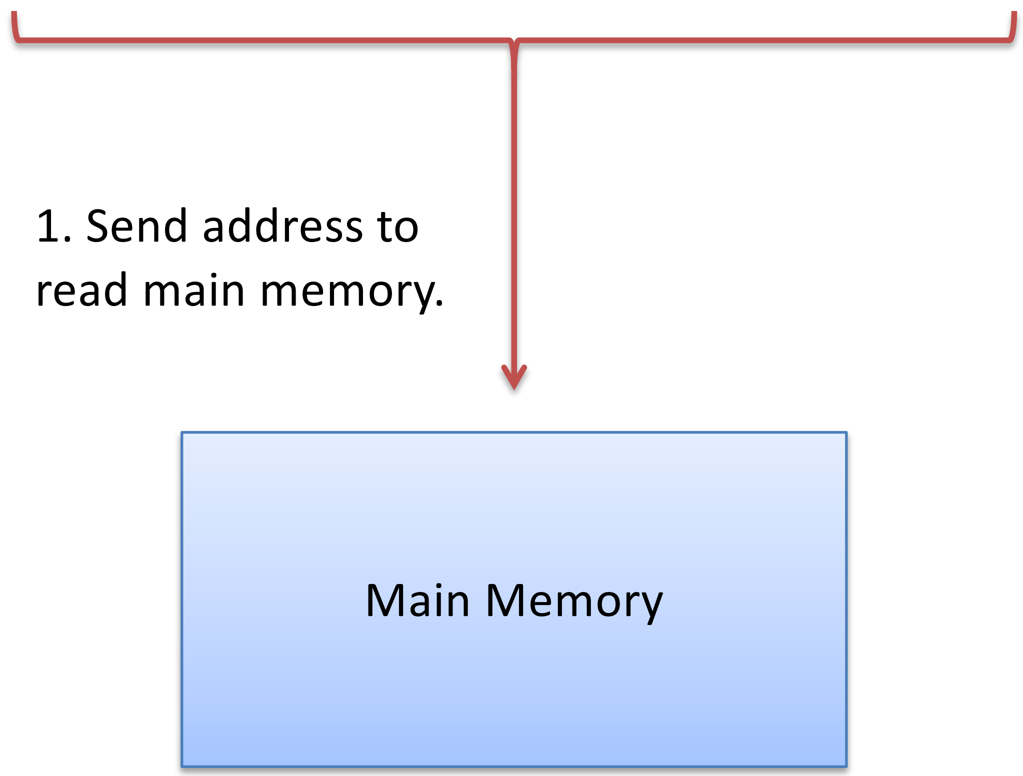


| Line | V | D | Tag | Data (8 Bytes) |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| … | … | | | |
| 1020 | 1 | 0 | 1323 | 57883 |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

# Eviction: Direct-Mapped

- Address division:

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



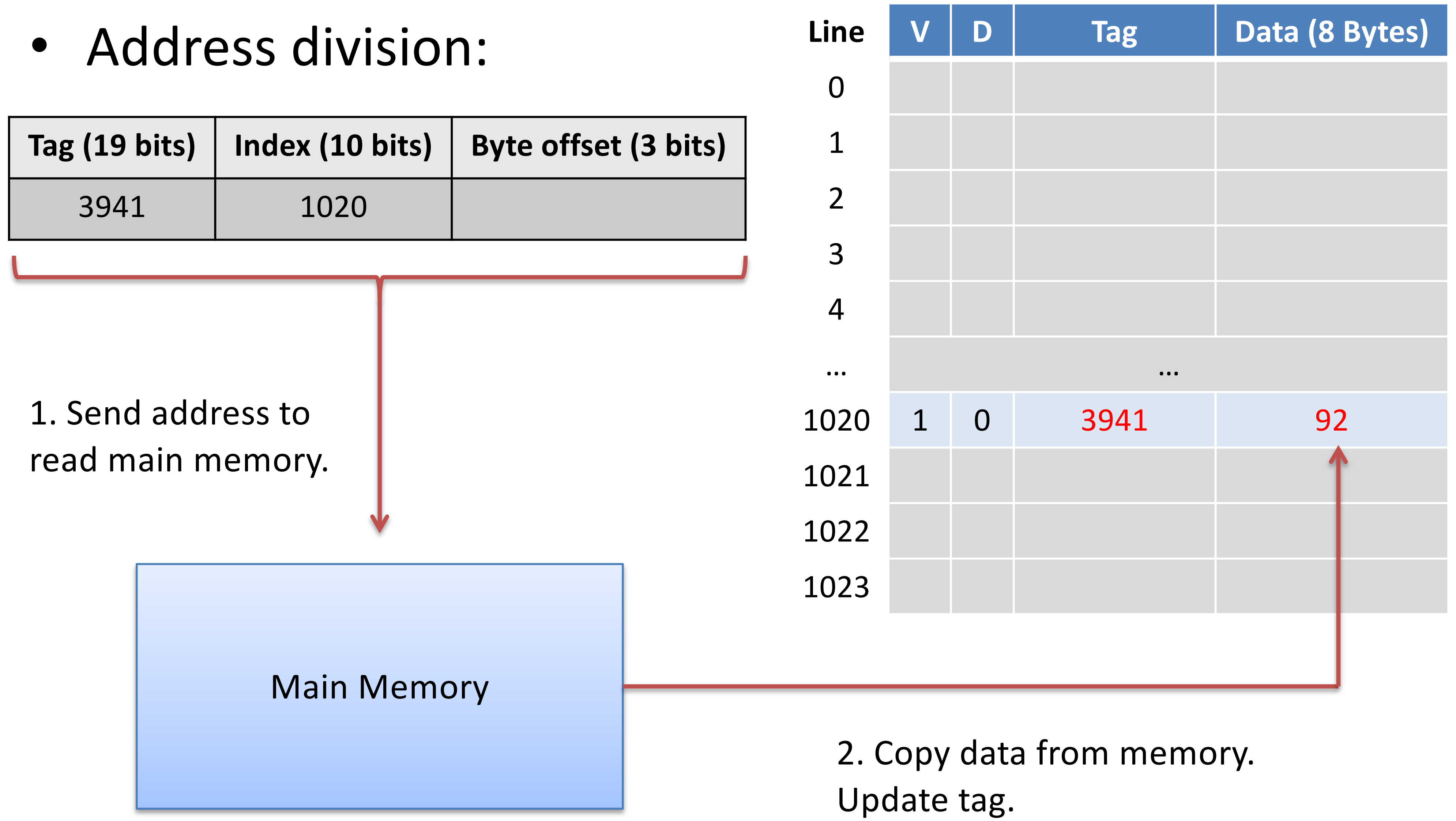
1. Send address to read main memory.

| 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
2. Copy data from memory. Update tag.

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

## 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?

Memory address

$\downarrow$

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 | ~~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?

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 | ~~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?

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~~ 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 <u>1</u> 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 <u>N</u> 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

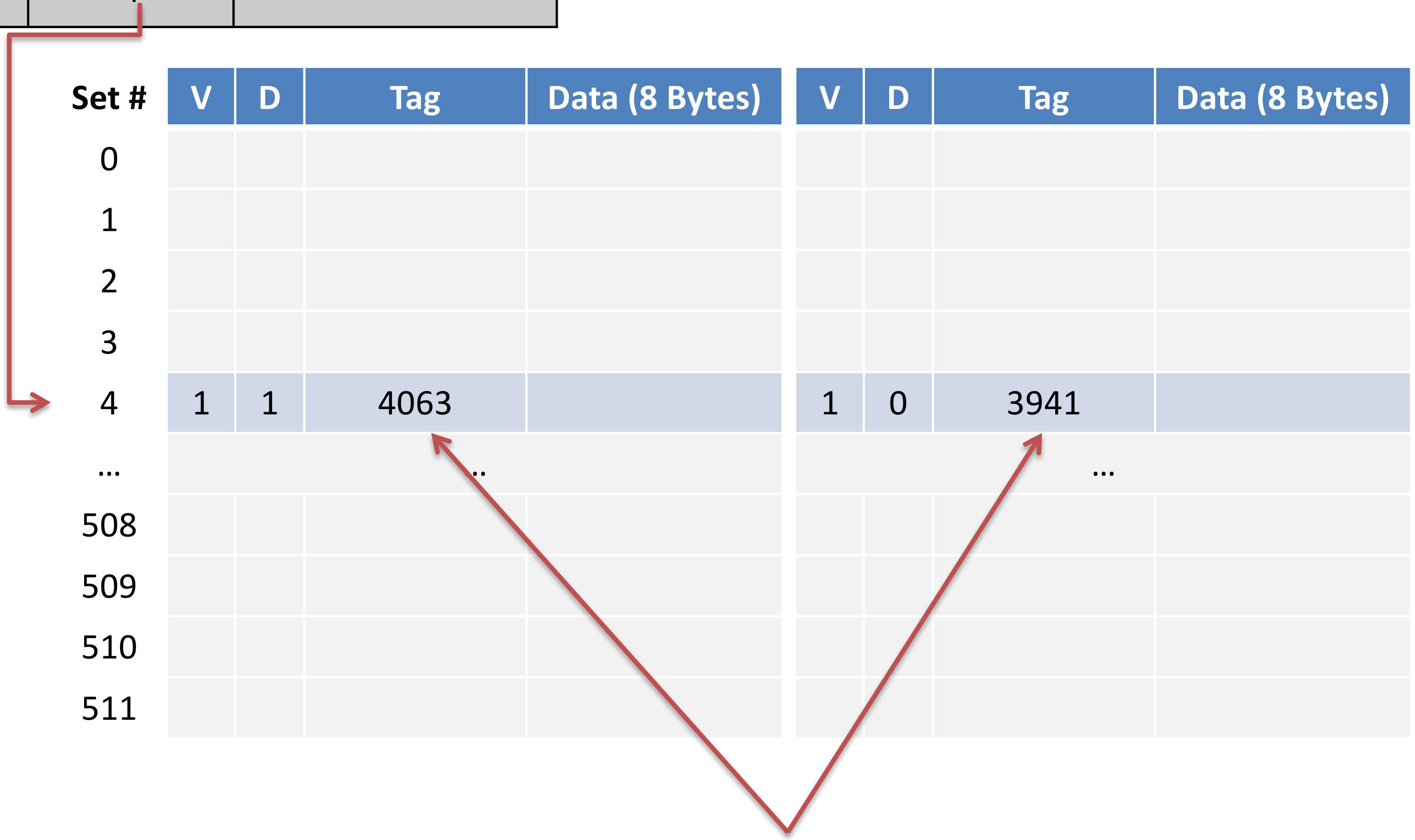
| Tag (20 bits) | Set (9 bits) | Byte offset (3 bits) |
|---|---|---|
| 3941 | 4 | |

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 | |

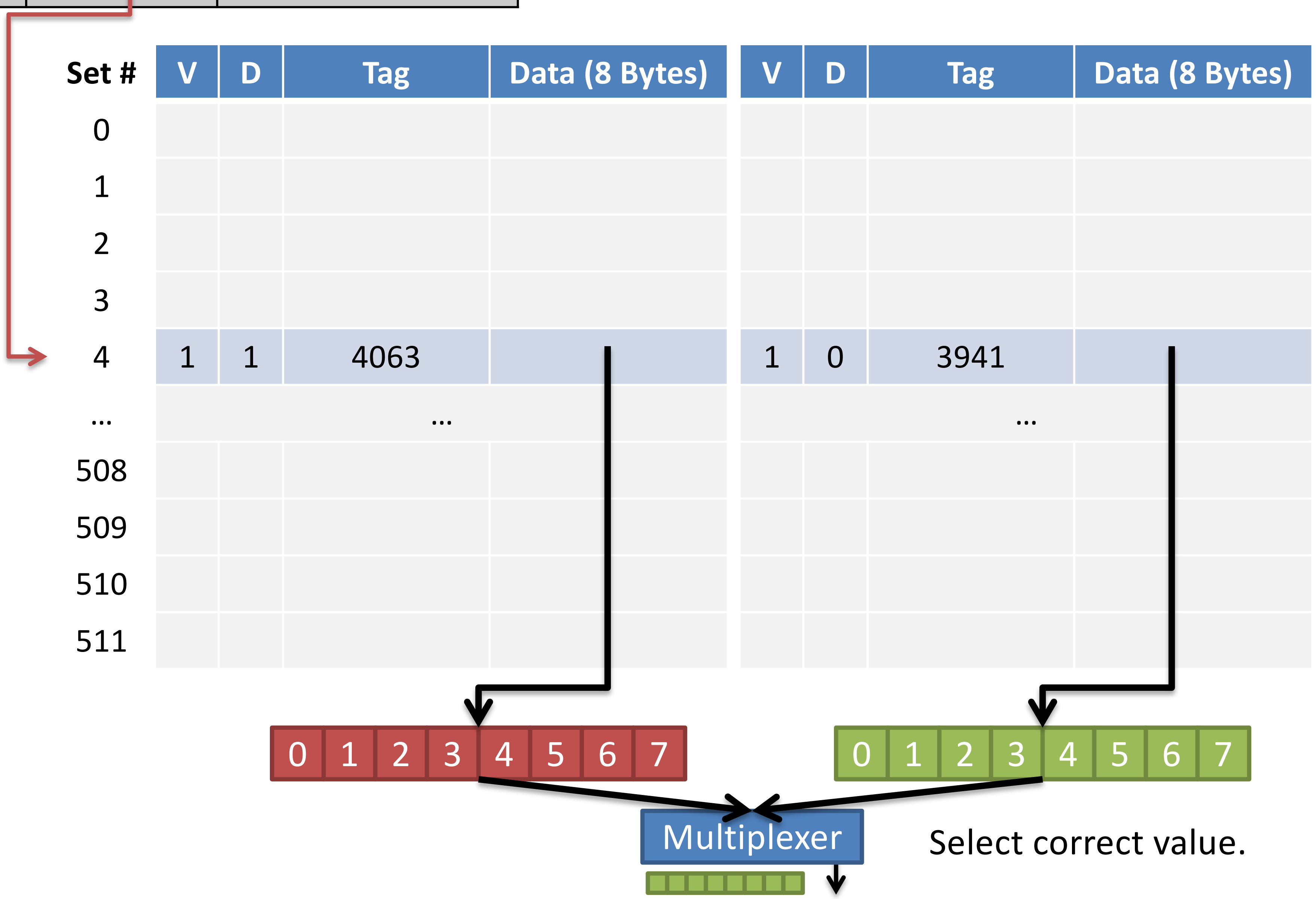


| Set # | V | D | Tag | Data (8 Bytes) | V | D | Tag | Data (8 Bytes) |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | 1 | 1 | 4063 | | 1 | 0 | 3941 | |
| … | | | … | | | | … | |
| 508 | | | | | | | | |
| 509 | | | |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

Check all locations in the set, in parallel.

# 2-Way Set Associative

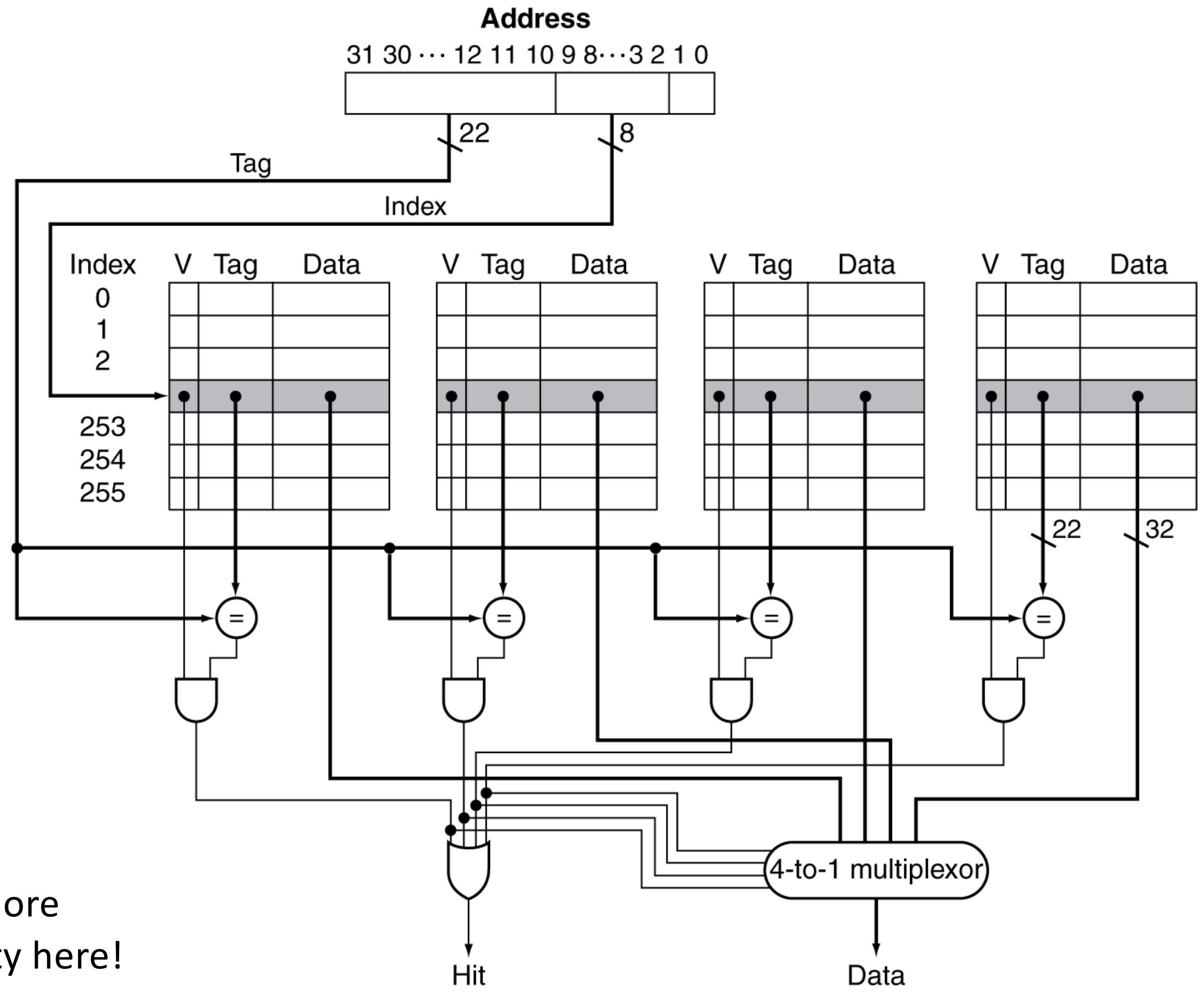
| Tag (20 bits) | Set (9 bits) | Byte offset (3 bits) |
|---|---|---|
| 3941 | 4 | |



| 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!

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ 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).

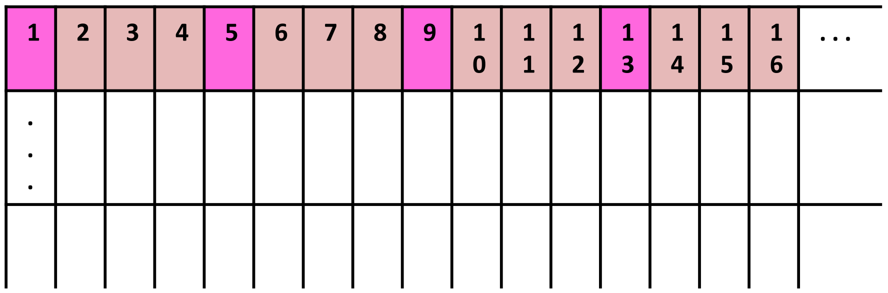
Is one faster than the other?

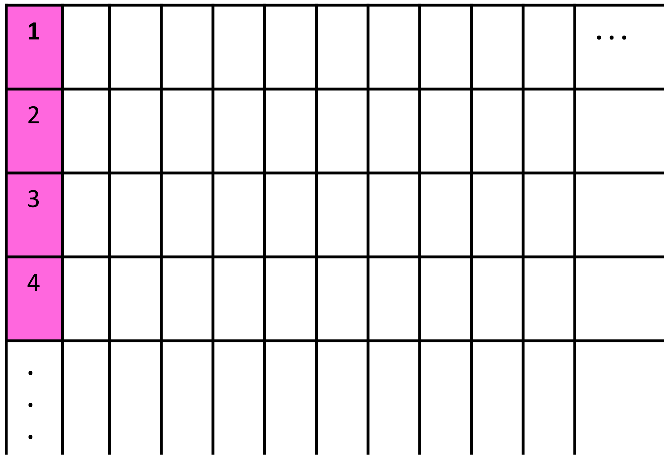
**C. Both would exhibit roughly equal performance.**

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





(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