CS 31: Introduction to Computer Systems

16 Storage and Locality

03-25-2025

Reading Quiz

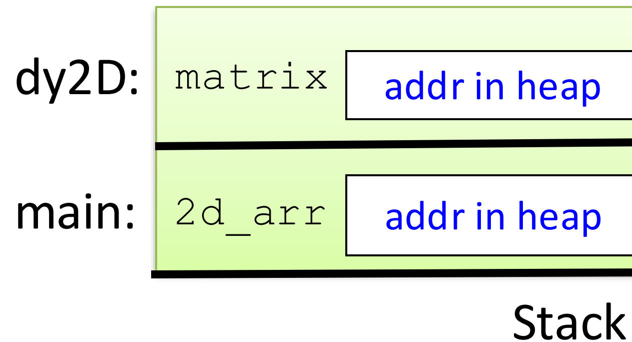
Today

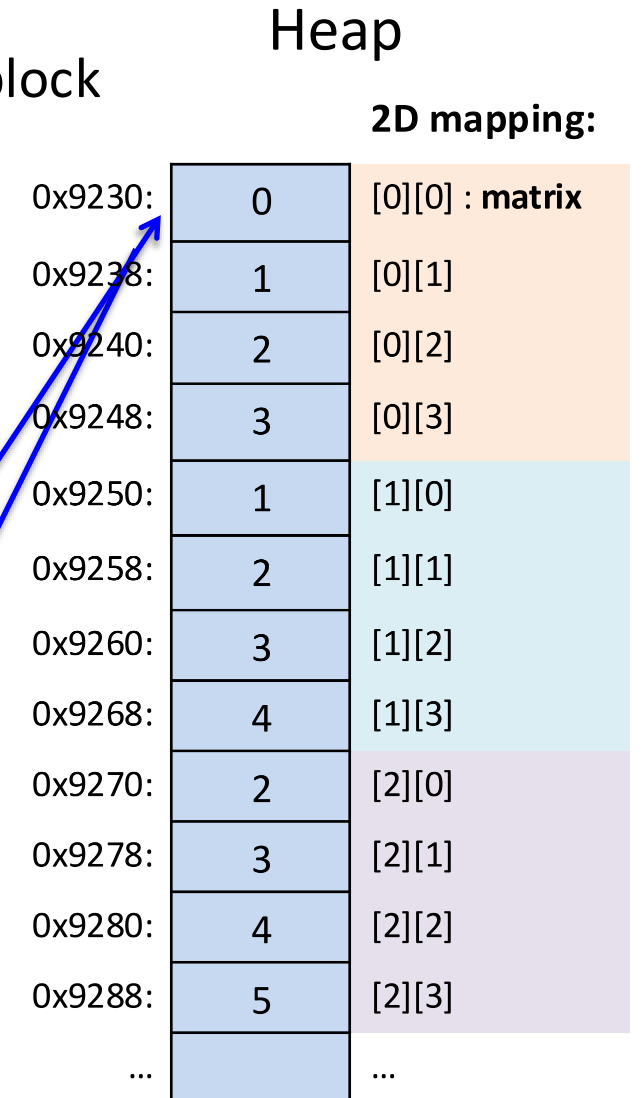
- Accessing *things* via an offset
  - Arrays, Structs, Unions
  - Connect accessing them in C with what we know about assembly
- How complex structures are stored in memory
  - Multi-dimensional arrays & Structs

Using Dynamically Allocated 2D Arrays as Parameters

- Parameter gets base address of contiguous memory in Heap
- Just like 1D arrays (almost). **Why?** It's just a pointer to a contiguous block of memory, only we (the programmer) know it represents a 2D array
- Pass *row* and *column* dimensions

```
void dy2D(int *matrix, int rows, int cols){
    int i, j;
    for(i=0; i < rows; i++) {
        for(j=0; j< cols; j++) {
            matrix[i*cols + j] = i*j;
        }
    }
}
int main() {
   long int *2d_arr = malloc(3 * 4 * sizeof(long int));
   dy2D(2d_arr, 3, 4);
}
```

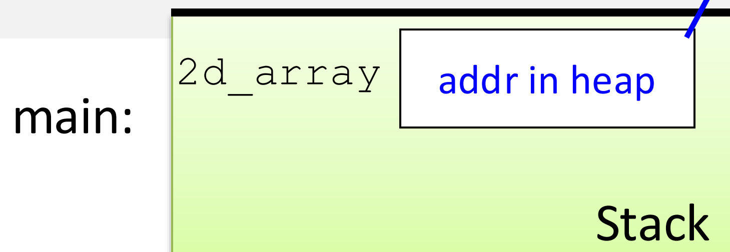


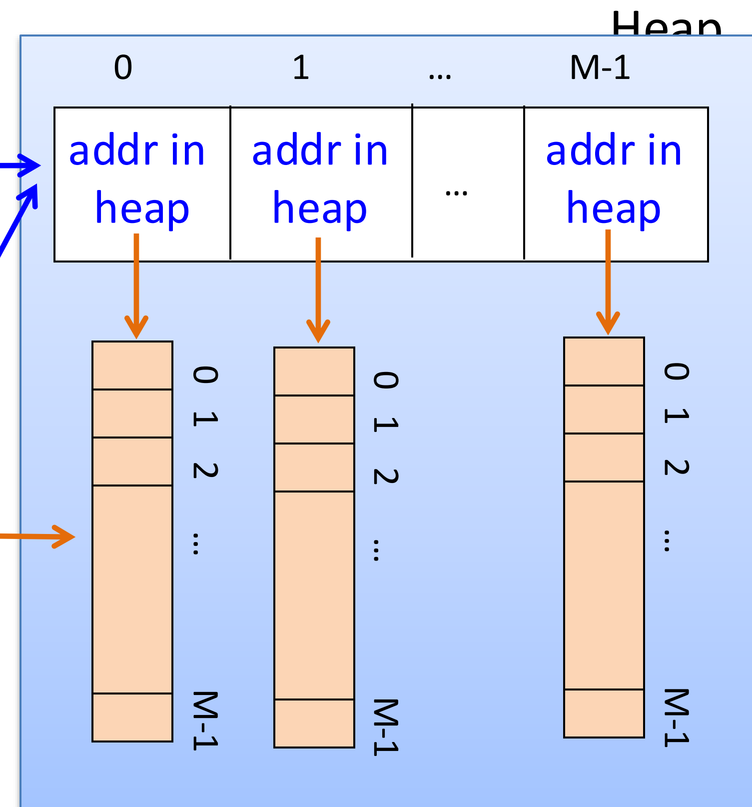


Dynamically Allocated 2D Array: Array of Pointers

- One malloc for an array of rows: an array of `int*`
- N mallocs for each row's column values: arrays of `int`
  - variable type is `int**`
  - stores address of rows array: an array of `int*`

```
int ** 2d_array;

// allocate a row of int pointers
2d_array = malloc (sizeof(int *) *M);


// for each int pointer in the row,
// allocate an array

for(i=0; i < M; i++) {
  2d_array[i] = malloc(sizeof(int)*N);
}
```



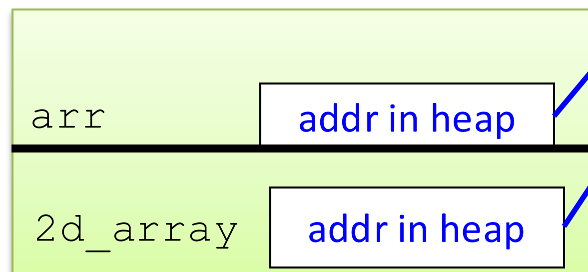


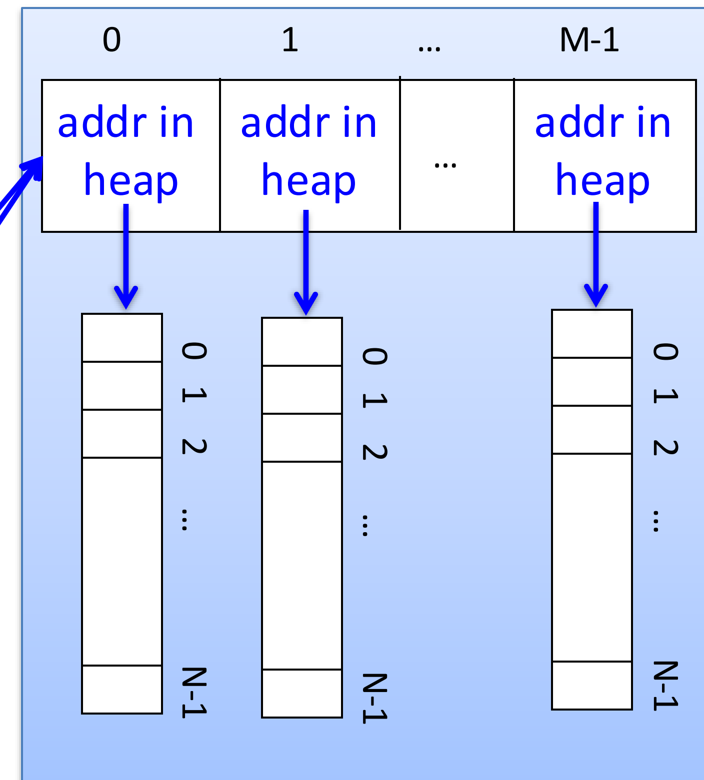
Using 2D Array (Array of Pointers) As Parameters

parameter gets base address of rows array of `int*`

- its type is `int**` : a pointer to `int*`: (with buckets of `int`)
- pass row and column dimension values
- Can use [i][j] to index into a specific location in 2D array.

```
void init2D(int **arr, int rows, int cols){
    int i, j;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            arr[i][j] = 0;
        }
    }
}
```



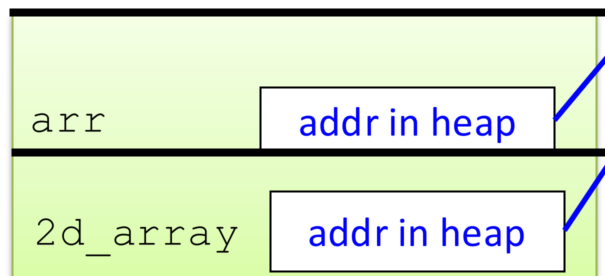


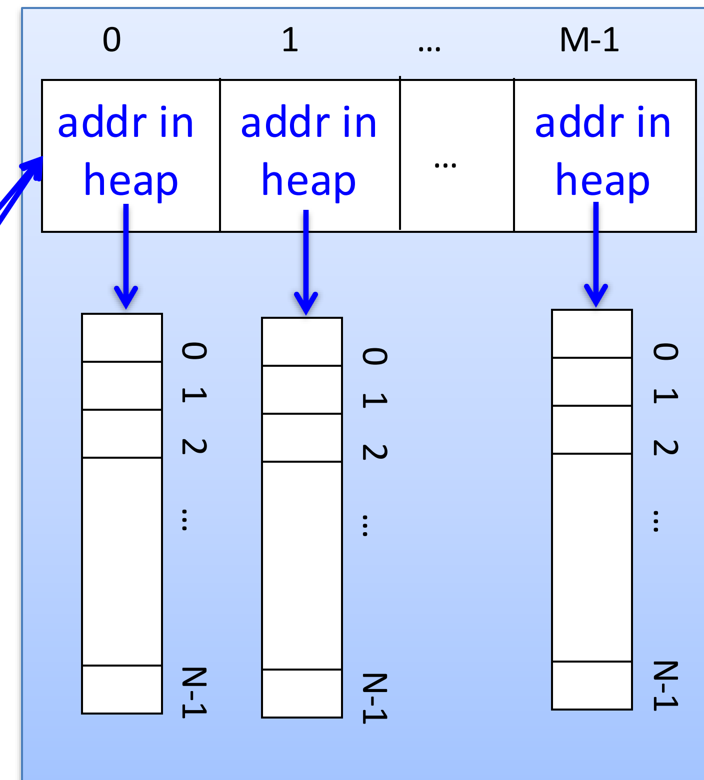
Using 2D Array (Array of Pointers): How about free-ing this memory?

parameter gets base address of rows array of `int*`

- its type is `int**` -> a pointer to an array of `int*`->
- each `int*` -> a pointer to an array of `ints`

```
void free(int **arr){
//TODO: decide which order to free memory

Option A: free the int ** array first
Option B: free the innner arrays (each int* array
first)

}
```





Two Ways for 2D Arrays

- We'll use BOTH methods in future labs:
  - **Lab 7**:
    - column-major, large chunk of memory that we treat as a 2D array,
    - use arr[index] where index = i * ROWSIZE + j to deference values
  - **Lab 8/9**:
    - array of integer pointers,
    - can use arr[N][M] to dereference values

Structs

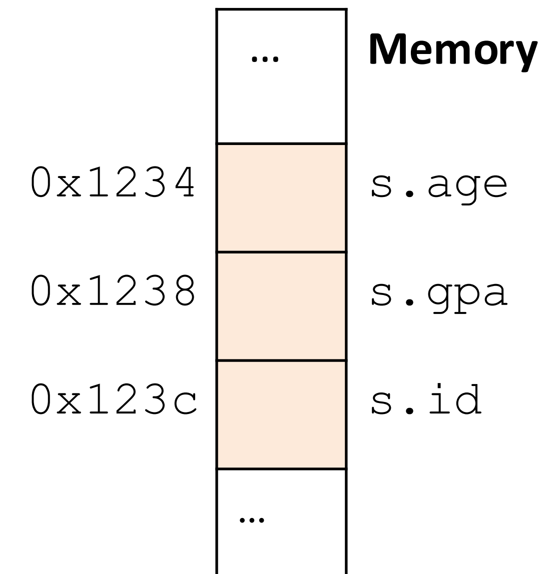
- Multiple values (fields) stored together
  - Defines a new type in C's type system
- Laid out contiguously by field (with a caveat we'll see later)
  - In order of field declaration.

Structs

Laid out contiguously by field (with a caveat we'll see later)

– In order of field declaration.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```

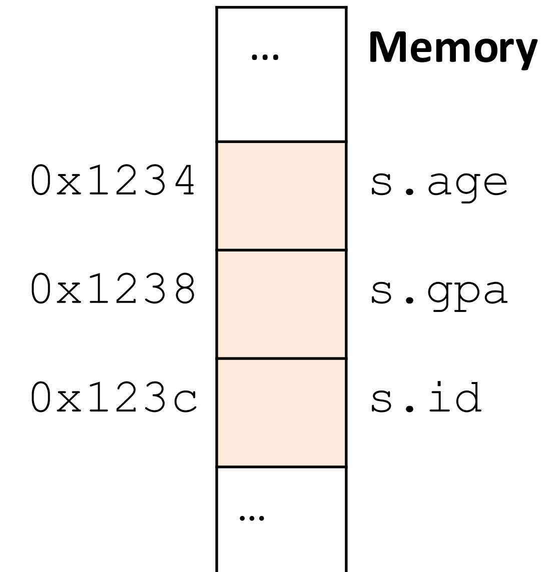


Structs

Struct fields accessible as a base + displacement

- Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Structs

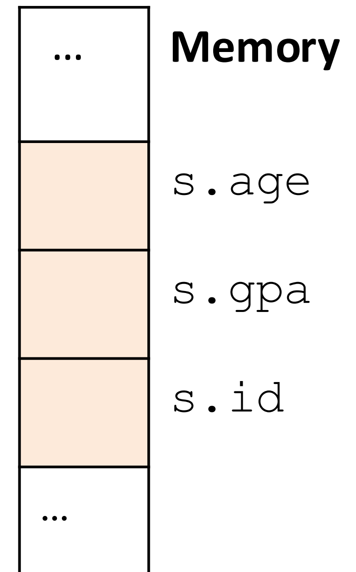
Struct fields accessible as a base + displacement

– Compiler knows (constant) displacement of each field

```
struct student{
   int age;
   float gpa;
   int id;
};

struct student s;
```

Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.



Structs

Struct fields accessible as a base + displacement
In assembly: `mov reg_value, 8(reg_base)`

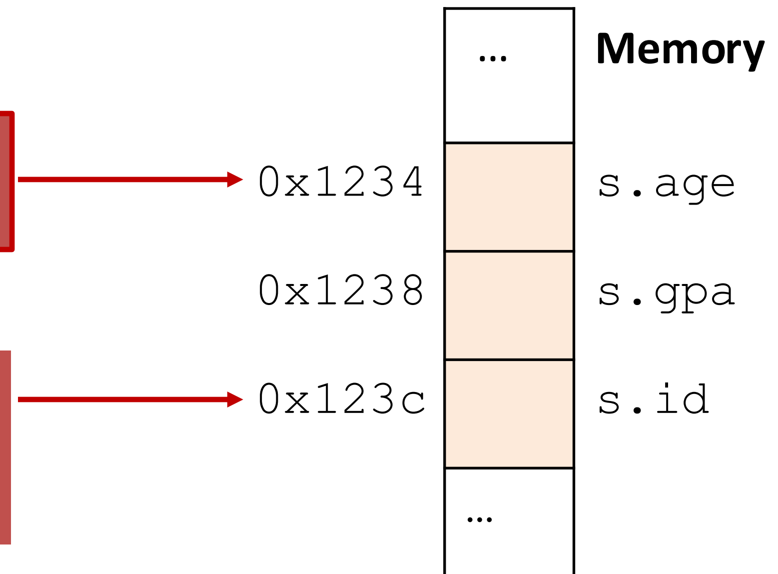
Where:

- `reg_value` is a register holding the value to store (say, 12)
- `reg_base` is a register holding the base address of the struct

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

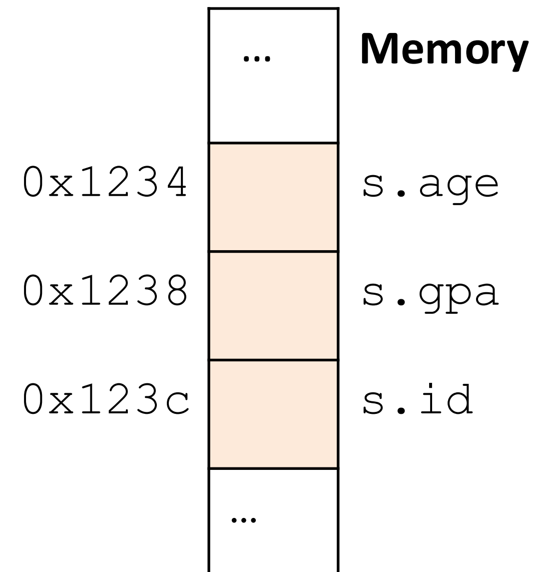
Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.



Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.



Data Alignment:

- Where (which address) can a field be located?

- char (1 byte): can be allocated at any address:

  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …

- short (2 bytes): must be aligned on 2-byte addresses:

  0x123**0**, 0x123**2**, 0x123**4**, 0x123**6**, 0x123**8**, …

- int (4 bytes): must be aligned on 4-byte addresses:

  0x123**0**, 0x123**4**, 0x123**8**, 0x123**c**, 0x124**0**, …

Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.

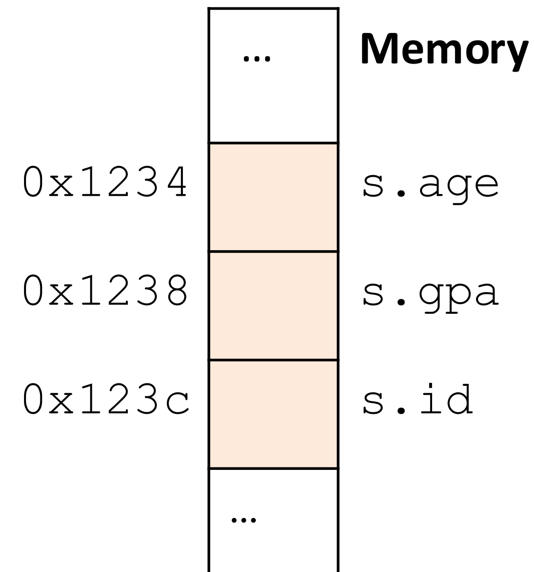
Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don't need to build wiring to access 4-byte chunks at any arbitrary location in hardware
- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)
- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```
struct student{
   int age;
   float gpa;
   int id;
};

struct student s;
```



Structs

```
struct student{
   char name[11];
   short age;
   int id;
};
```

How much space do we need to store one of these structures? Why?

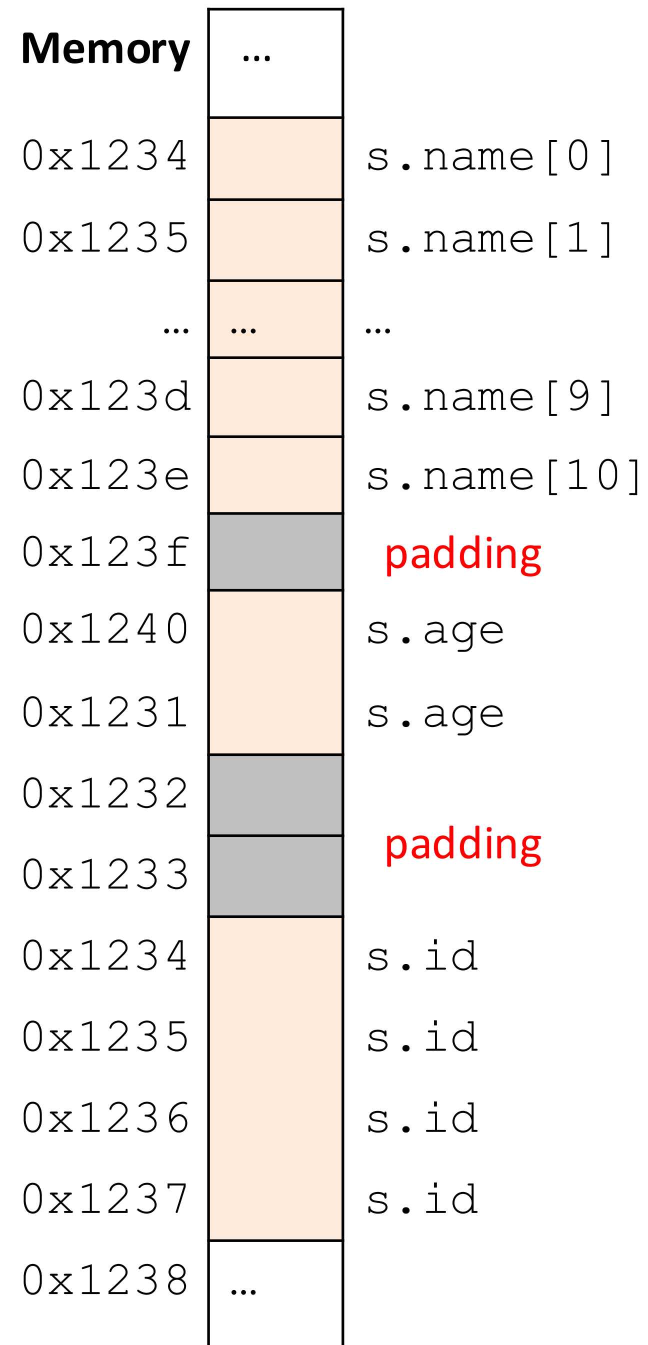
```
struct student{
  char name[11];
  short age;
  int id;
};
```

A. 17 bytes
B. 18 bytes
C. 20 bytes
D. 22 bytes
E. 24 bytes

Structs

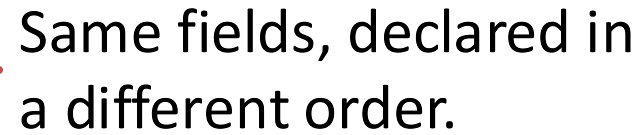
```
struct student{
  char name[11];
  short age;
  int id;
};

size of data:  17 bytes
size of struct: 20 bytes!
```

Use sizeof() when allocating structs with malloc()!



Alternative Layout

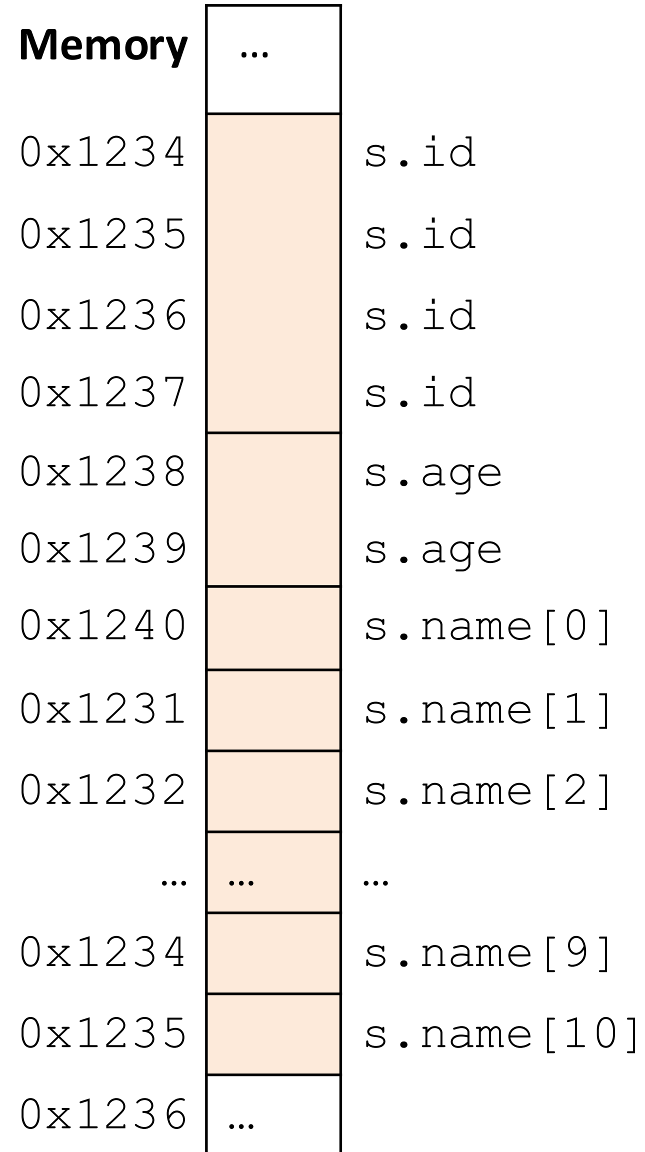
```
struct student{
   char name[11];
   short age;
   int id;
};
```



Alternative Layout

```
struct student{
  char name[11];
  short age;
  int id;
};

size of data:  17 bytes
size of struct: 17 bytes
```

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



Aside: Network Headers

- In networks, we attach metadata to packets
  - Things like destination address, port #, etc.
- Common for these to be a specific size/format
  - e.g., the first 20 bytes must be laid out like …
- Naïvely declaring a struct might introduce padding, violate format.

Cool, so we can get rid of this struct padding by being smart about declarations?

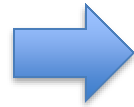
A. Yes (why?)

B. No (why not?)

Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.
- Rearranging helps, but often padding after the struct can't be eliminated.

```
struct T1 {
    char c1;
    char c2;
    int  x;
};
```

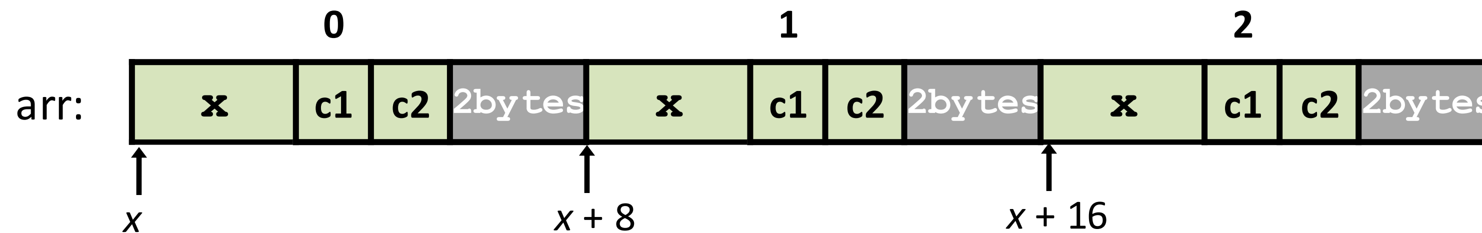


```
struct T2 {
    int x;
    char c1;
    char c2;
};
```

“External” Padding

Array of Structs: Field values in each bucket must be properly aligned:

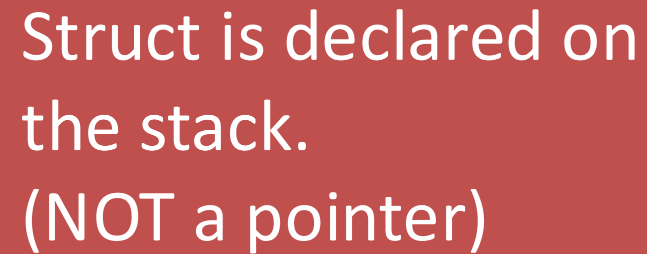
```
struct T2 arr[3];
```



Buckets must be on a 8-byte aligned address

Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, “Alice”);
```



Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Struct field syntax…

What about this?

```
struct student {
   int id;
   short age;
   char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Option 1: Works but ugly

```
(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, “Alice”);
```

Option 2: Use struct pointer dereference!

```
s->id = 406432;
s->age = 20;
strcpy(s->name, “Alice”);
```

Memory alignment applies elsewhere too!

```
int x;            vs.    double y;
char ch[5];              int x;
short s;                 short s;
double y;                char ch[5];
```

In nearly all cases, *you shouldn't stress about this.* The compiler will figure out where to put things.

Exceptions: networking, OS

Structs and Arrays

- Use Structs & Arrays to build complex data types
- Very important to think about type!

  from the outside in: (e.g.) a[3].age

  - type of a is a pointer to an array of student
  - can use [i] notation to access a bucket of this array
  - type of a[3] is a student struct
  - can use . to access a field in struct
  - type of a[3].age is an int
- Remember how different types are passed
  - semantics of passing an array vs. a struct
  - it is all pass by value, but what value is differs by type

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 Efficiently 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

Efficiency

- How to Efficiently 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)

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.

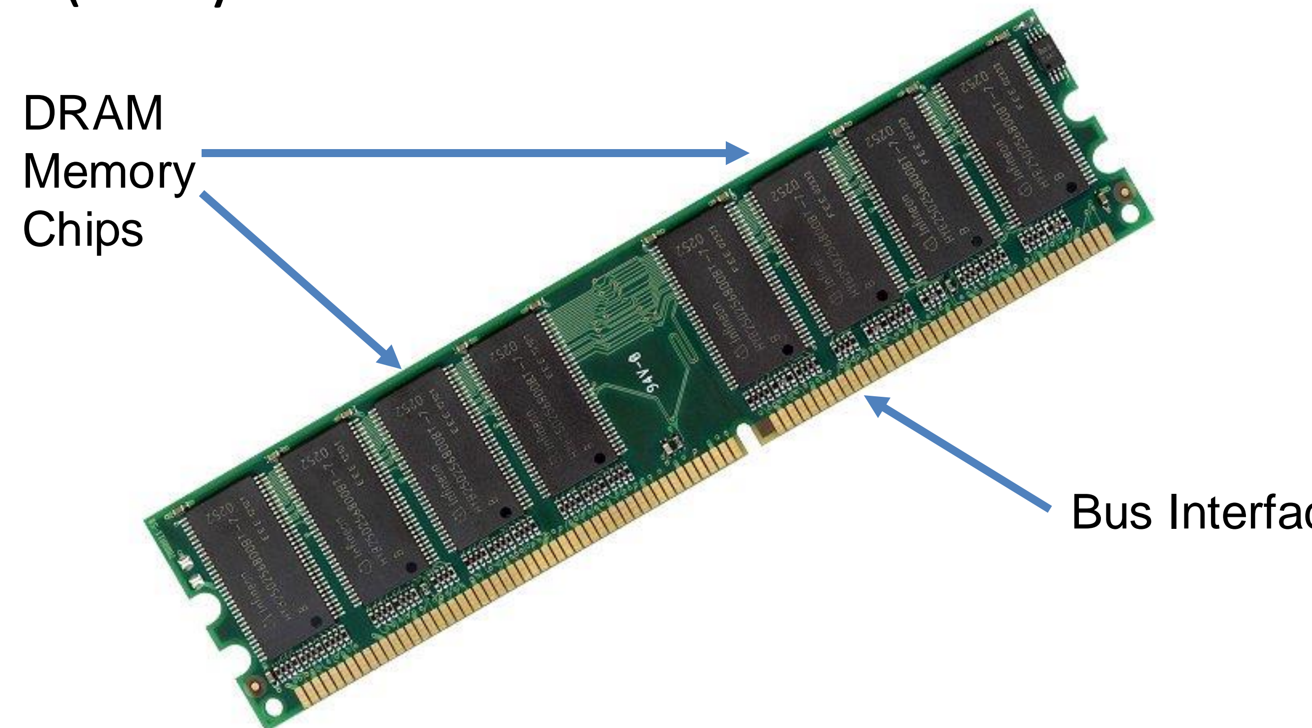
Dynamic Random Access Memory (DRAM)

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.

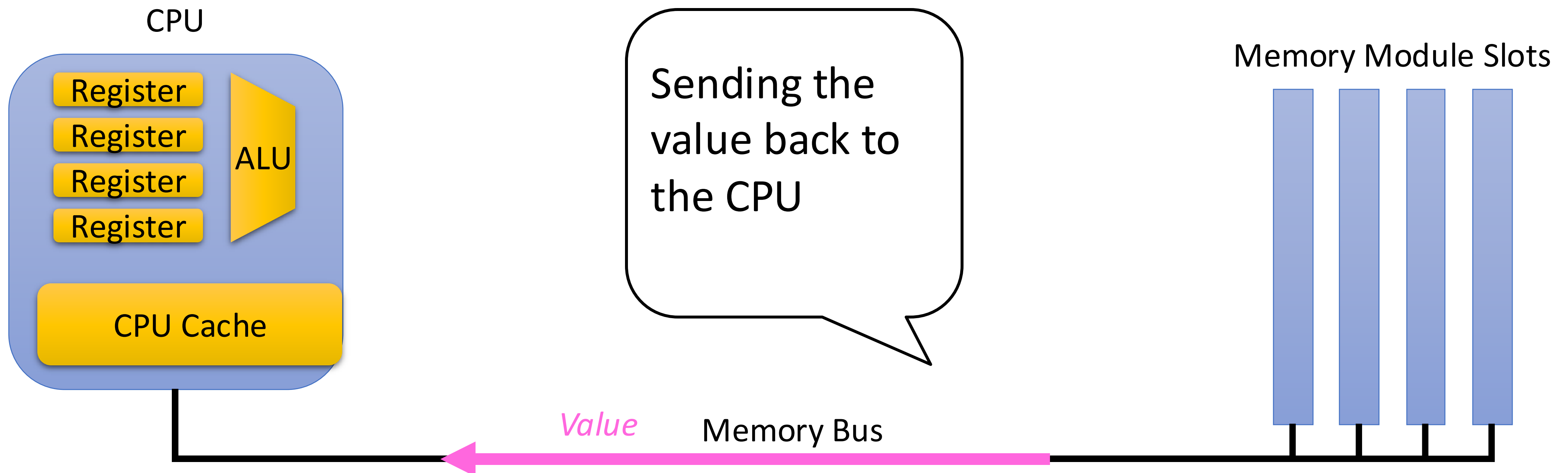
How A Memory Read Works

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

Load operation: `mov (Address A), %rax`

How A Memory Read Works

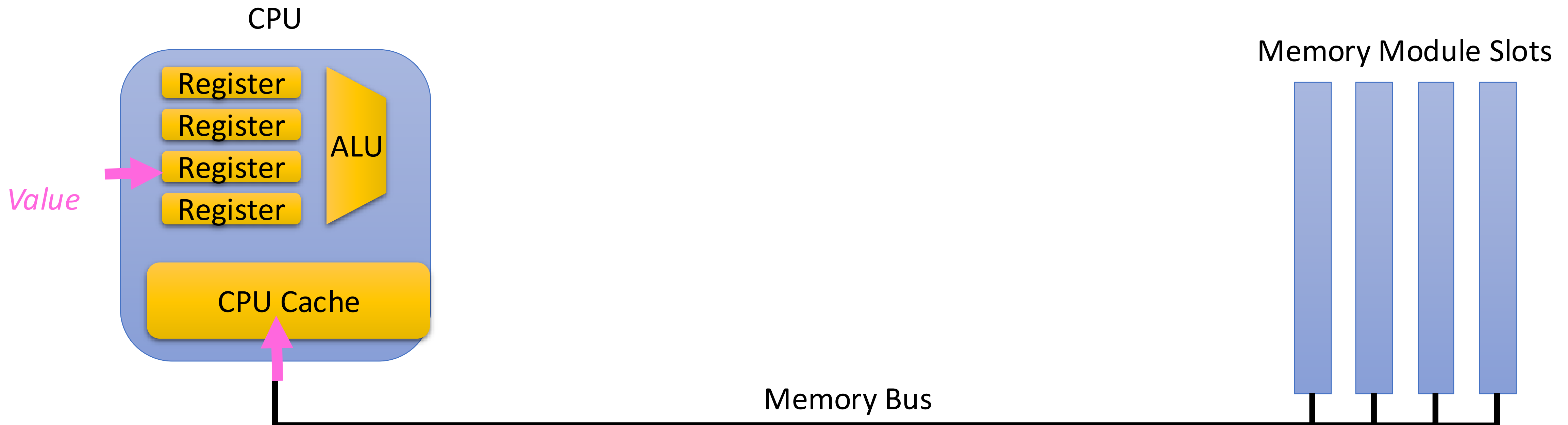
(2) Main Memory reads address A from memory, fetches value at that address and puts it on the bus



How A Memory Read Works

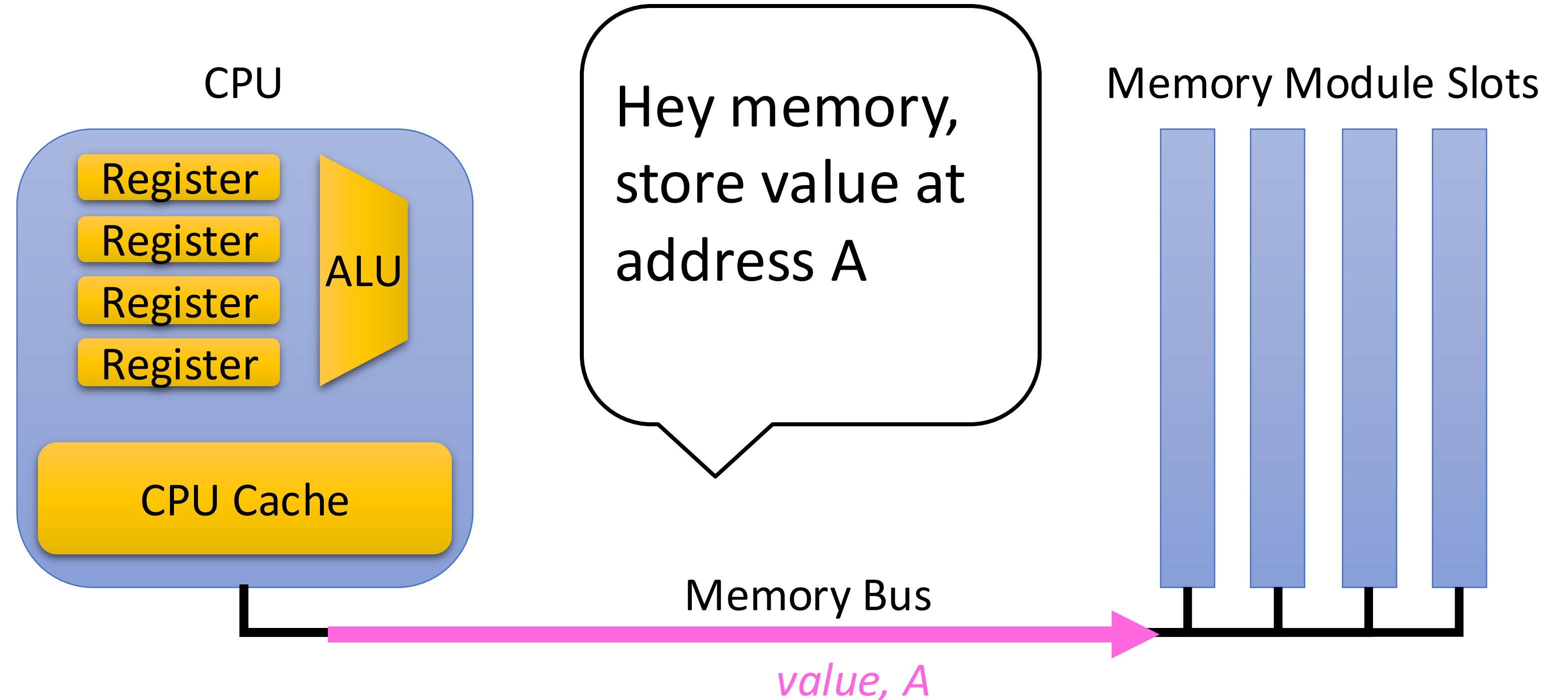
(3) CPU reads value from the bus, and copies it into register rax.

<u>a copy also goes into the on-chip cache memory</u>



How a Memory Write Works

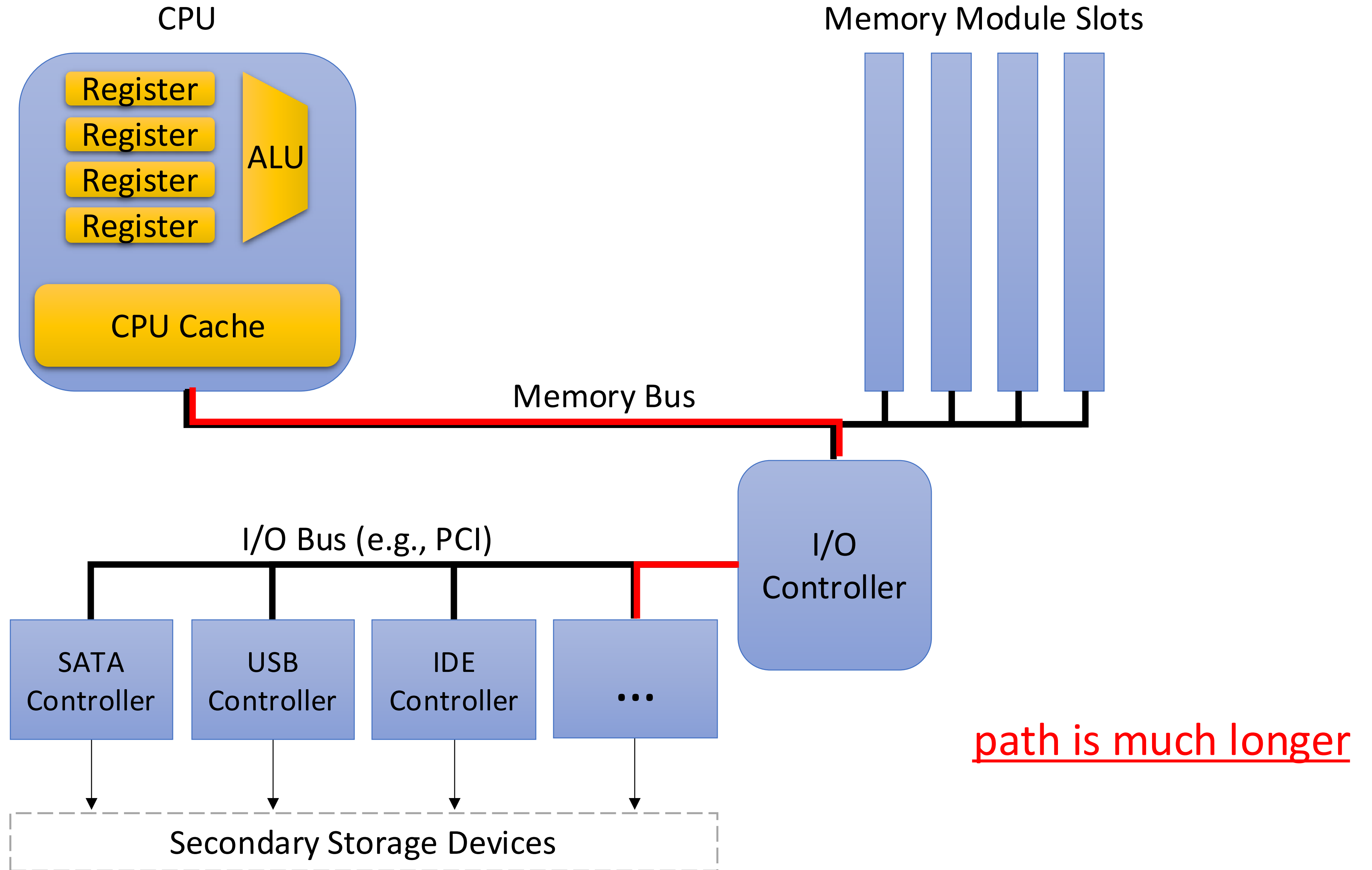
1. CPU writes A to bus, memory reads it
2. CPU writes value to bus, memory reads it
3. Memory stores value at address A



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



What’s Inside A Disk Drive?

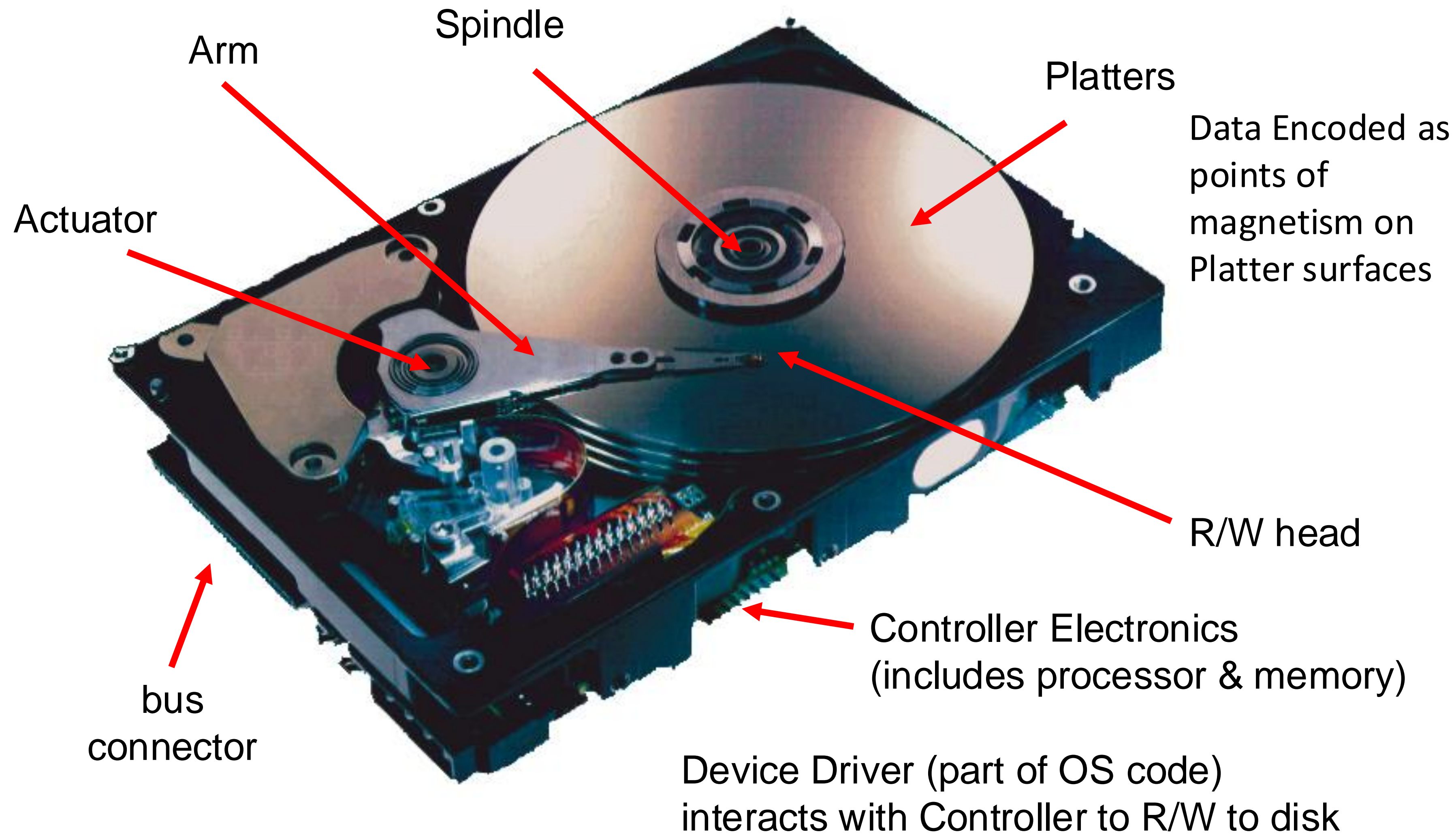
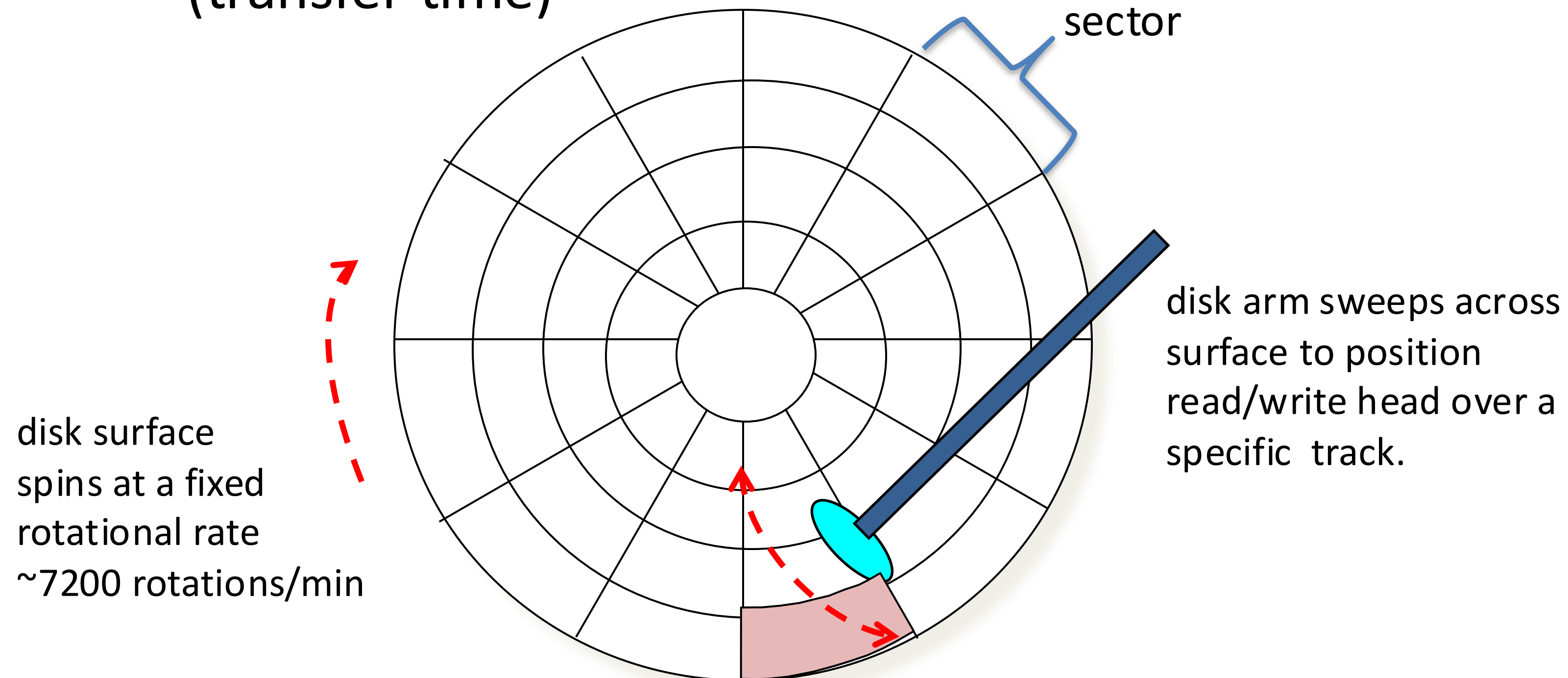


Image from Seagate Technology

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)



Memory Technology

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

Like walking:

Down the hall

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

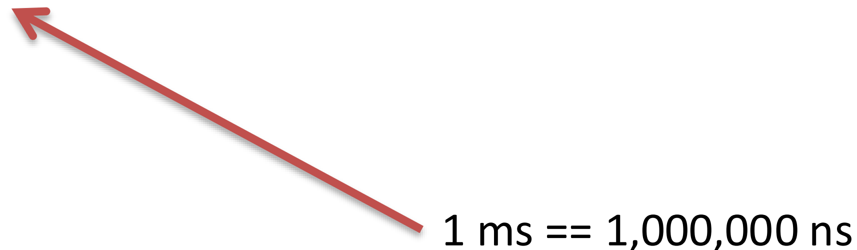
Across campus

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

(to Cleveland / Indianapolis)

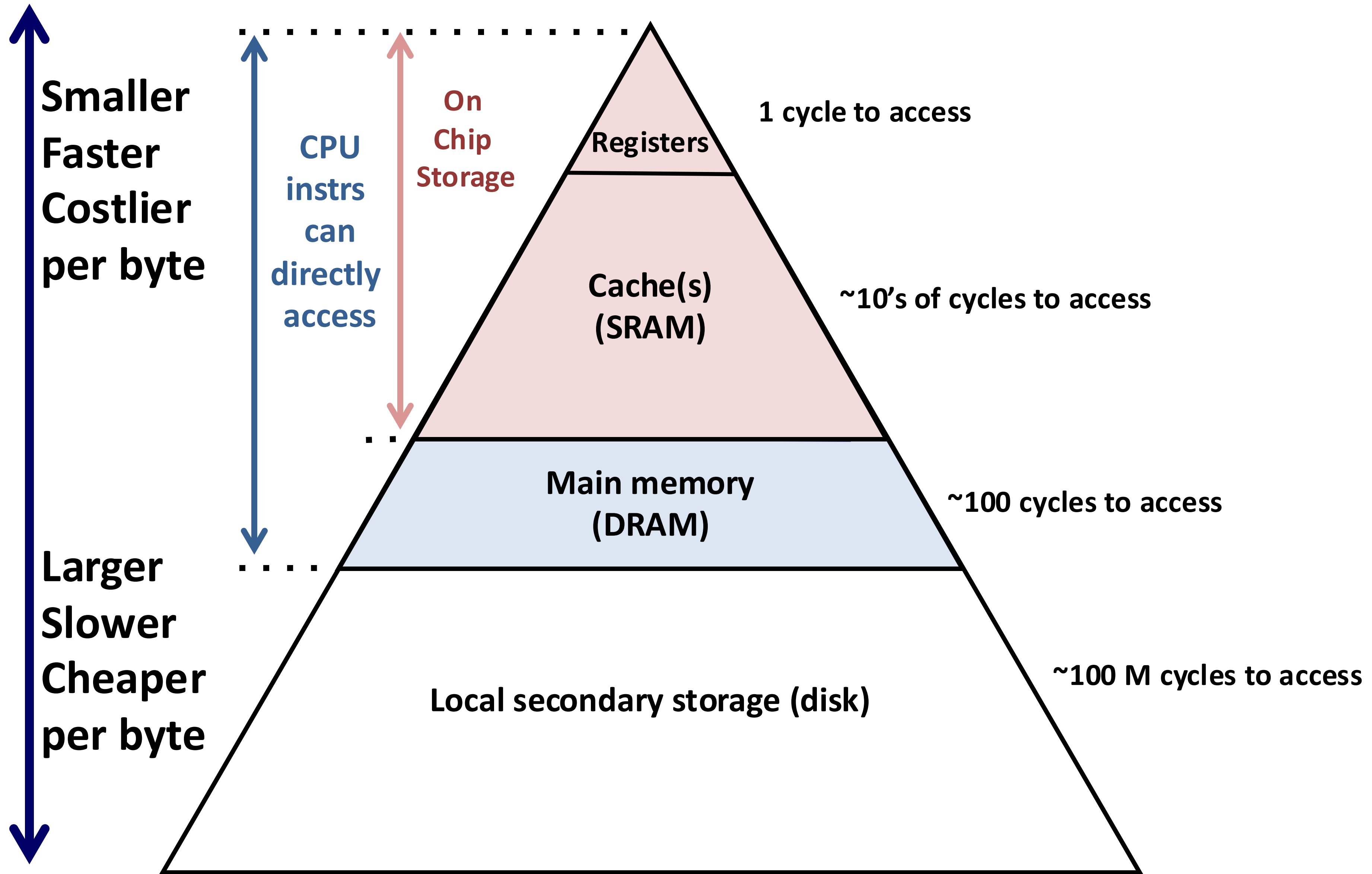
- Magnetic disk
  - 5ms – 15ms, $0.20 – $2 per GB

To Seattle

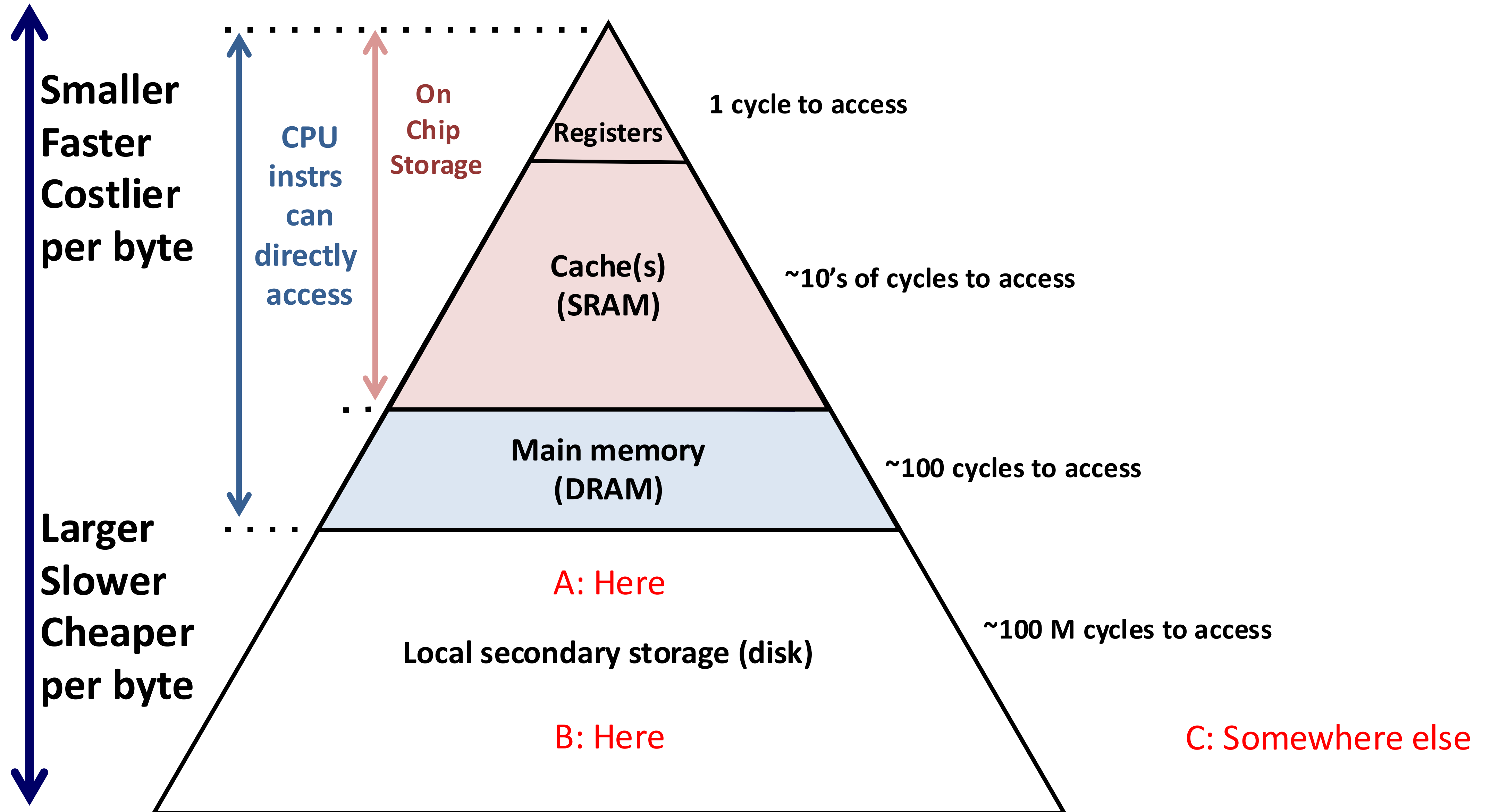


1 ms == 1,000,000 ns

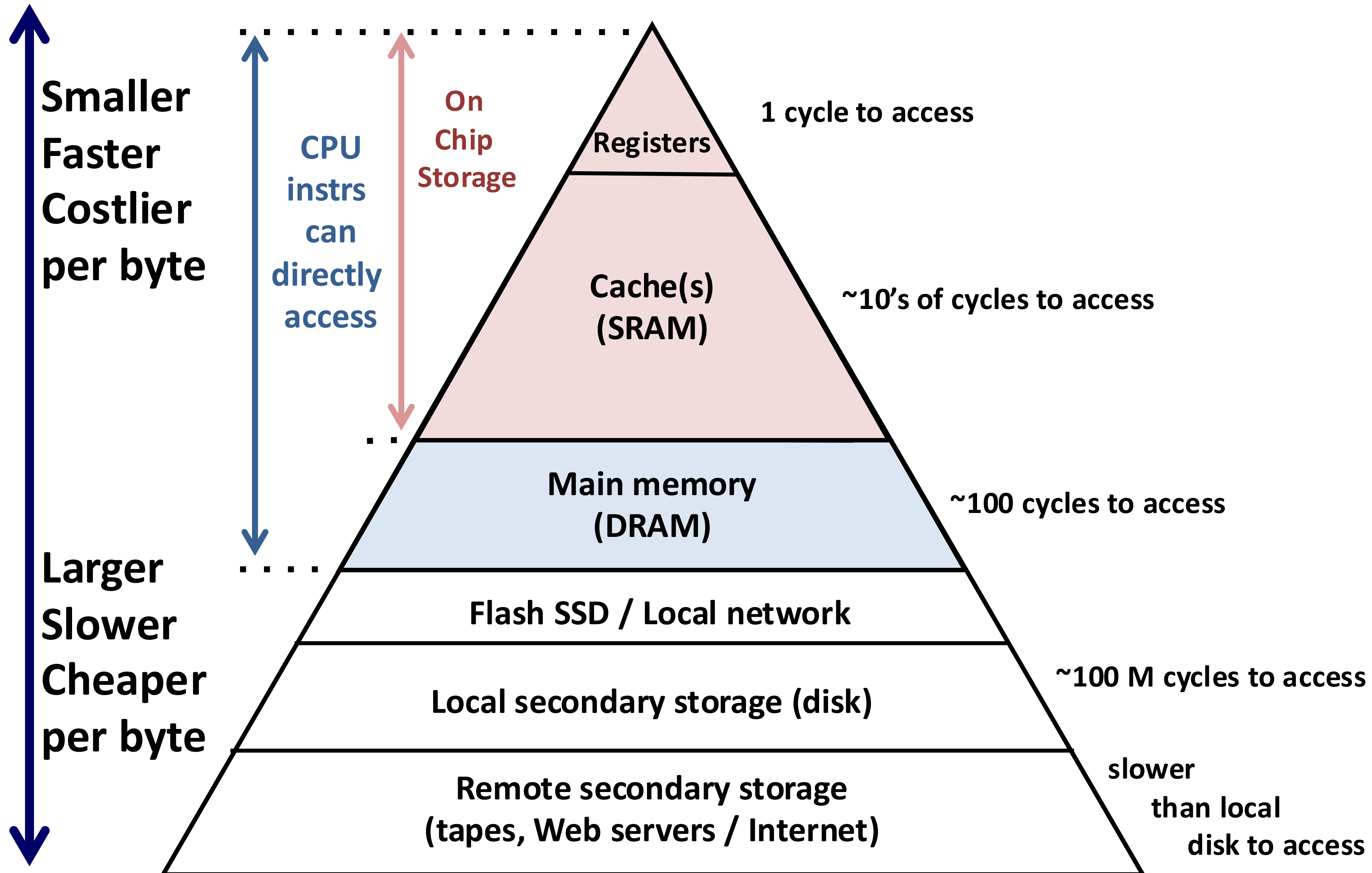
The Memory Hierarchy



Where does accessing the network belong?



The Memory Hierarchy



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

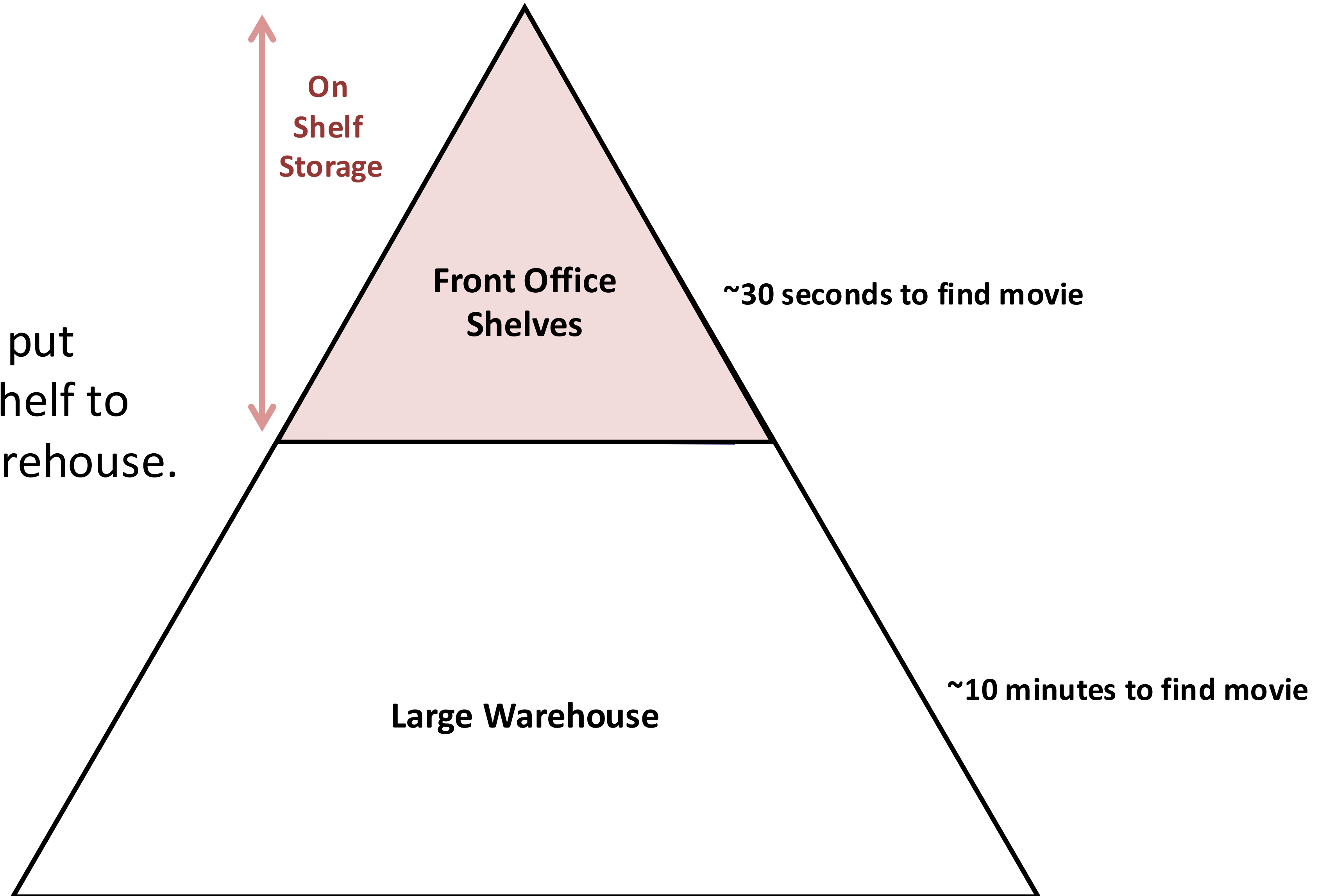
- You work at a library

- You have a huge library
  - 10-15 minutes to find a book and 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 a book on the front shelf

The Library Hierarchy

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



Quick vote: Which book should we place on the shelf ?

A. To Kill a Mockingbird

B. One Hundred Years of Solitude

C. Brave New World

D. Pride and Prejudice

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 borrowed “The Hobbit” ten times in the last two weeks.

- You talk to him:
  - He is looking forward to his tenth Renaissance Fair meet-up.

Quick vote: Which book should we place on the shelf today?

A. To Kill a Mockingbird

B. One Hundred Years of Solitude

C. The Hobbit

D. Pride and Prejudice

E. There is no way for us to knpw

Quick vote: Which book should we place on the shelf today?

A. To Kill a Mockingbird

B. One Hundred Years of Solitude

C. The Hobbit

D. Pride and Prejudice

E. There is no way for us to know

Repeat Customer: Alice

- Alice read “Dune” a month ago.
- You talk to her:
  - She’s really likes science fiction!
- Over the next few weeks she borrowed:
  - Dune Messiah (Dune-II), Children of Dune (Dune-III)

Quick vote: Which book should we place on the shelf today?

A. To Kill a Mockingbird

B. One Hundred Years of Solitude

C. God Emperor of Dune (Dune-IV)

D. Pride and Prejudice

E. There is no way for us to know

Quick vote: Which book should we place on the shelf today?

A. To Kill a Mockingbird

B. One Hundred Years of Solitude

C. <u>God Emperor of Dune (Dune-IV)</u>

D. Pride and Prejudice

E. There is 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 that has been 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?

```
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

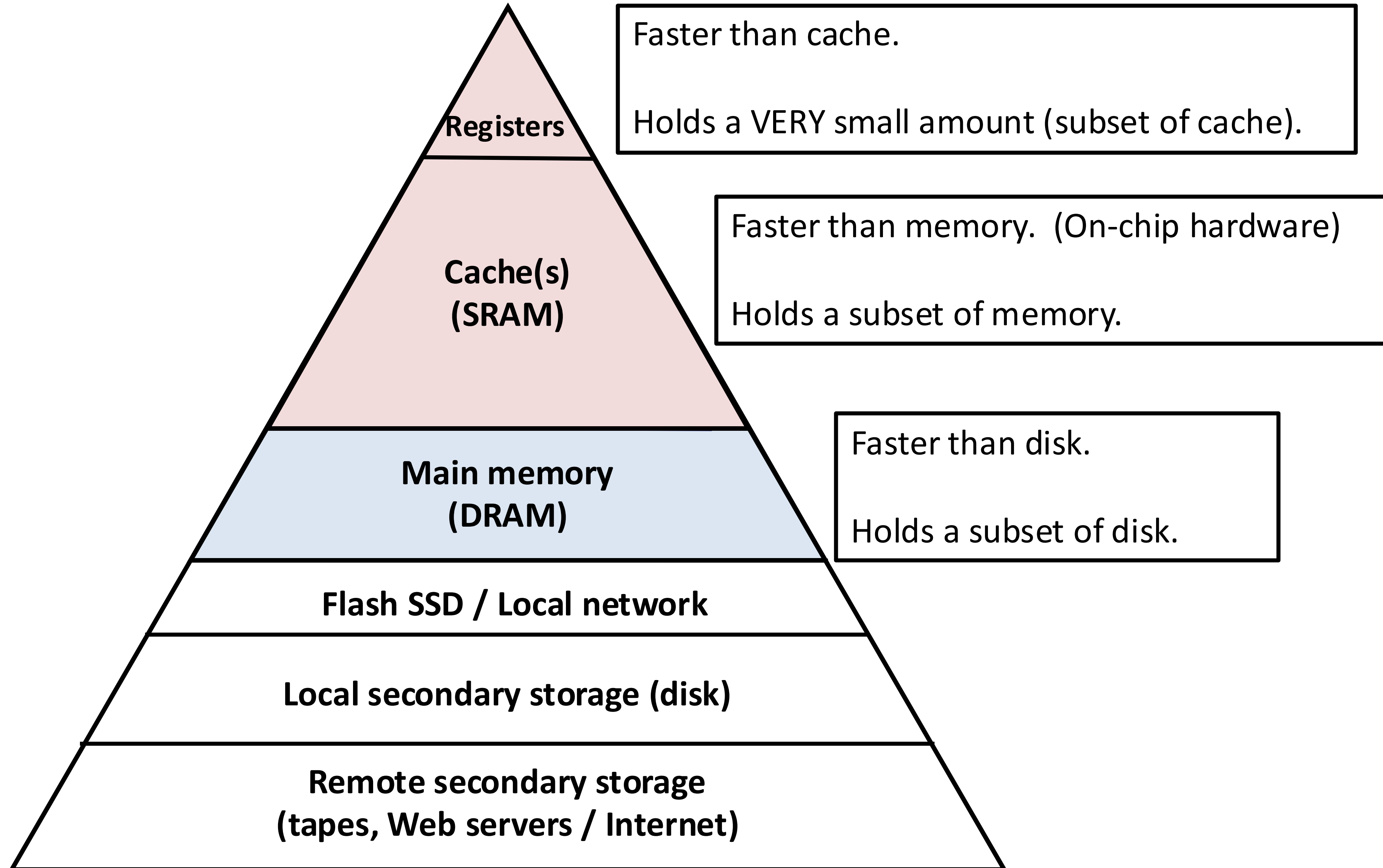
Temporal

- 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



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.

Cache

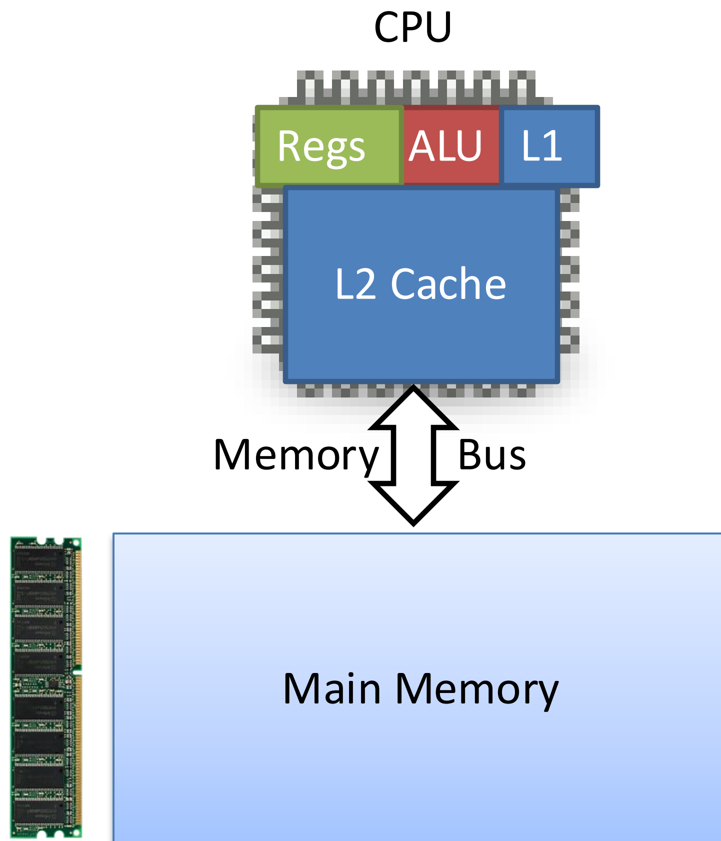
- In general: a storage location that holds a subset of a larger memory, faster to access

When we say “cache”, assume we’re referring to CPU cache from now on, unless we say otherwise.

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

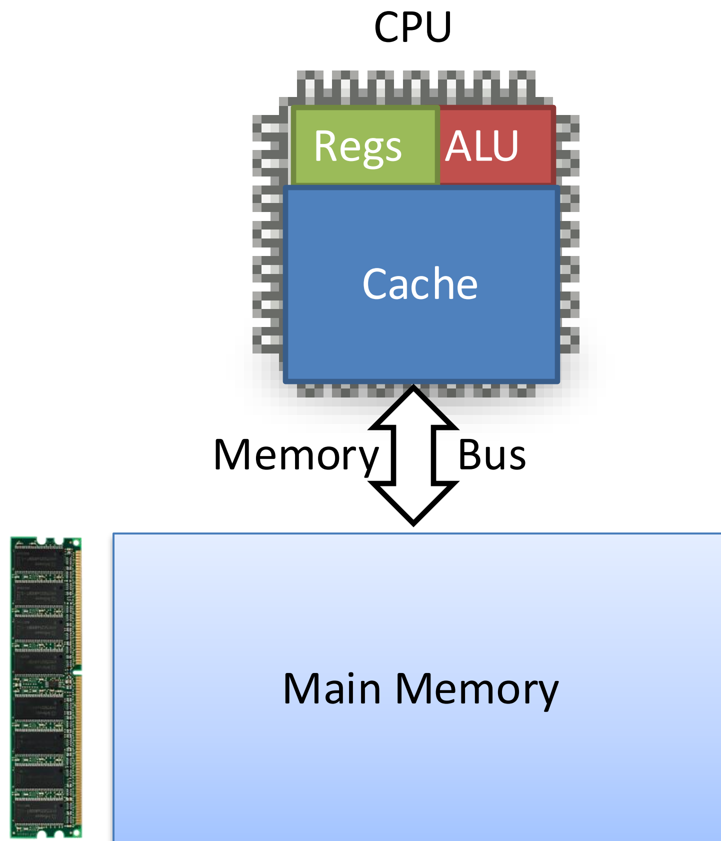
- Goal: choose the <u>right subset</u>, based on past locality, to achieve our abstraction

Cache Basics



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

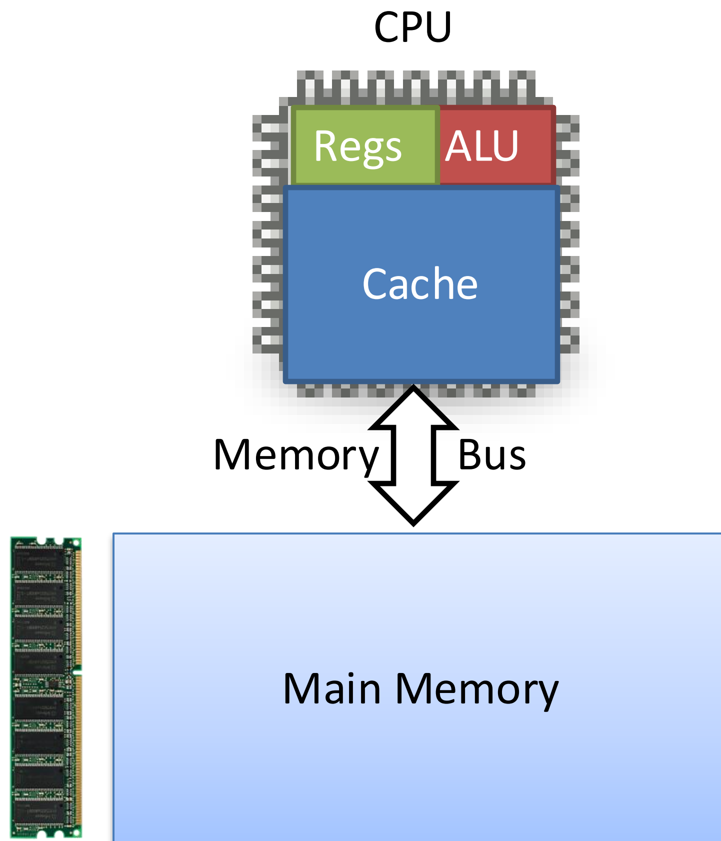
Cache Basics



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

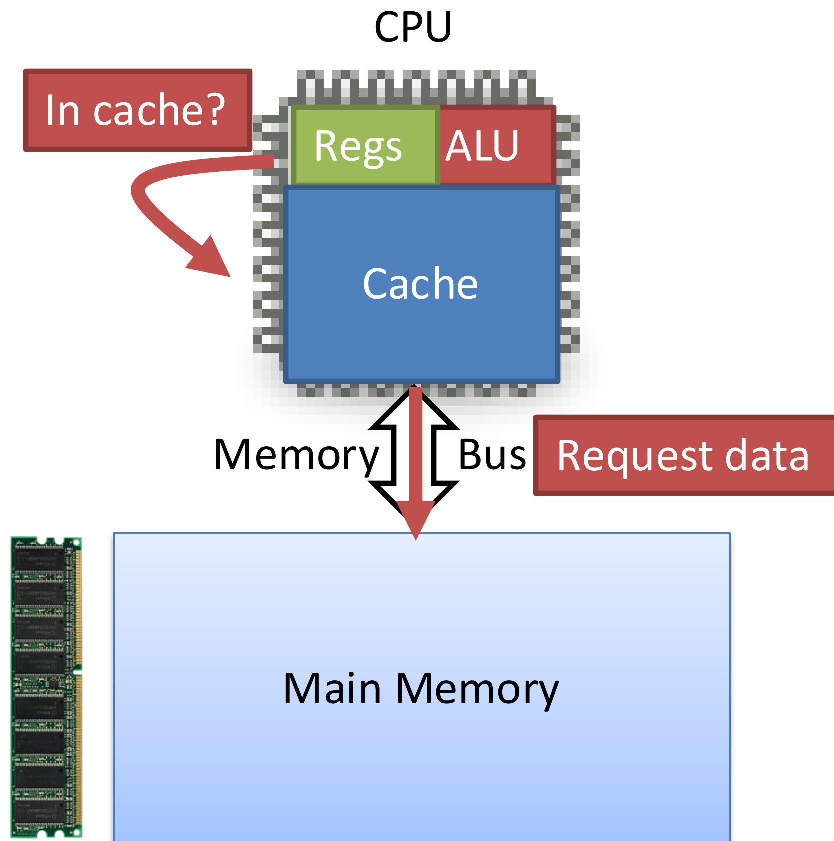
- CPU real estate dedicated to cache
- Usually two levels:
  - L1: smallest, fastest
  - L2: larger, slower
- We'll assume one cache (same principles)

Cache Basics: Read from memory

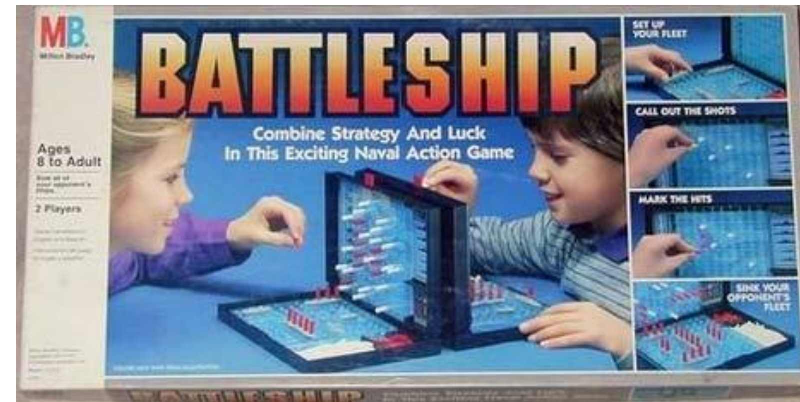


- In parallel:
  - Issue read to memory
  - Check cache

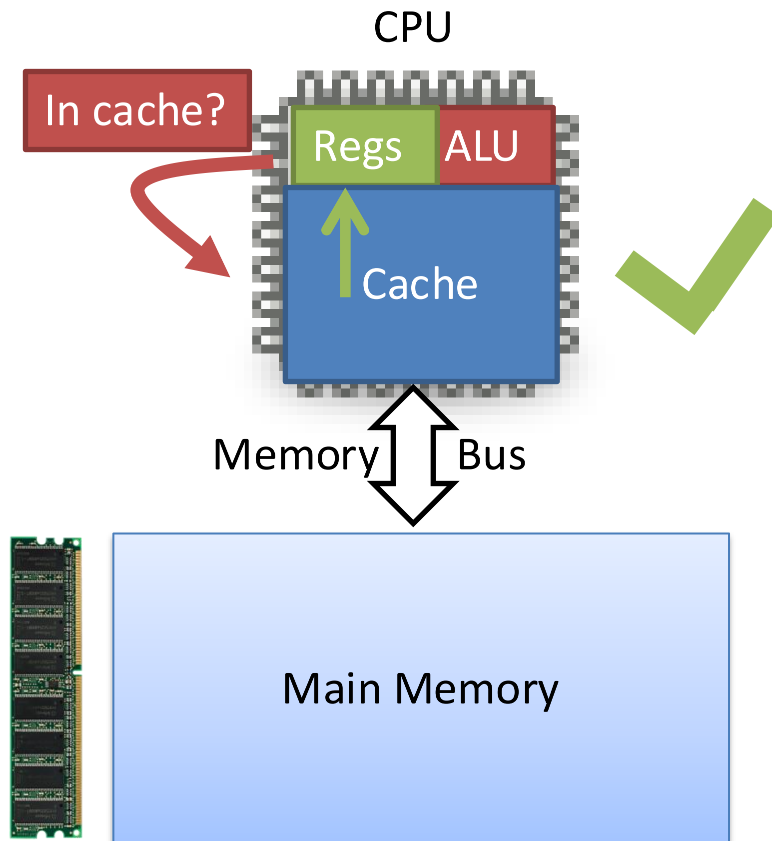
Cache Basics: Read from memory



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

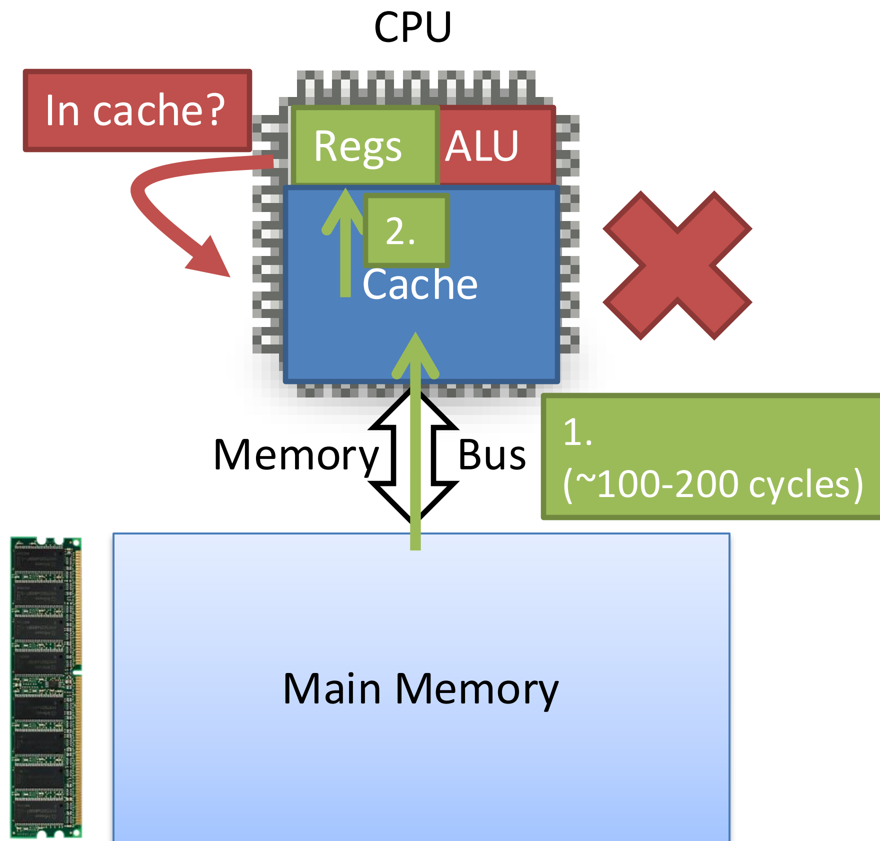


Cache Basics: Read from memory



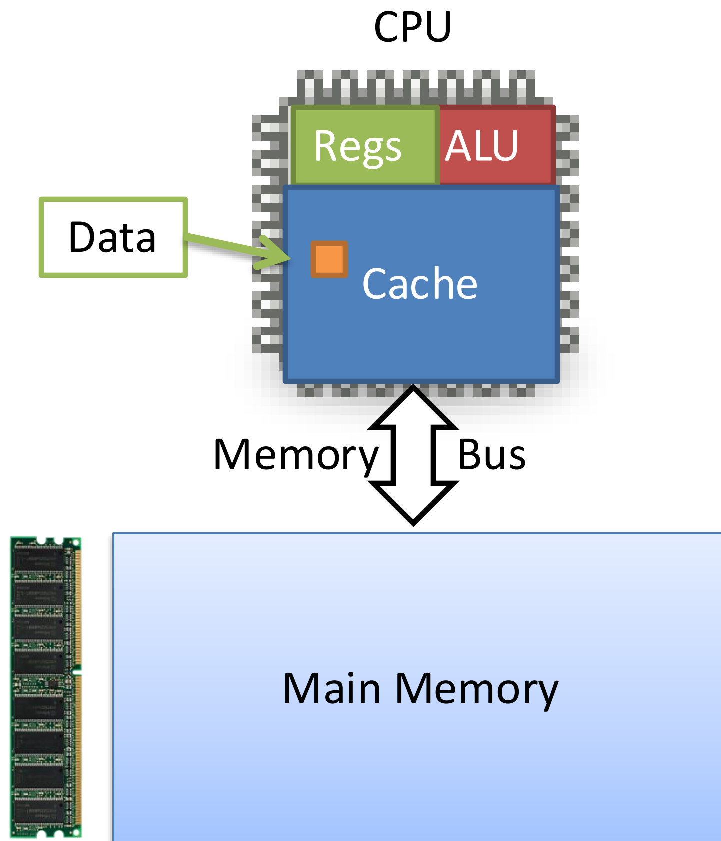
- In parallel:
  - Issue read to memory
  - Check cache
- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory

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 remove ("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. Write-through: Immediately update the data in memory when we update the cache.

B. Write-back: Update the data in memory when we remove ("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?)

Cache Basics: Write to memory

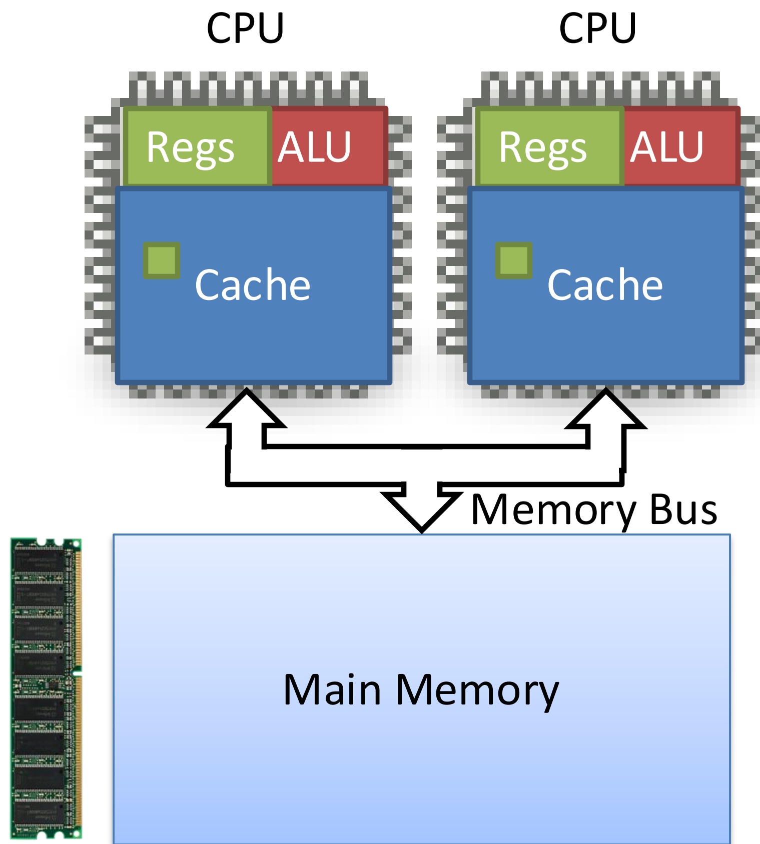
- Both options (write-through, write-back) viable

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

- Write-back: only write to memory on eviction
  + potentially reduces memory accesses (faster)
  – complex (cache inconsistent with memory)

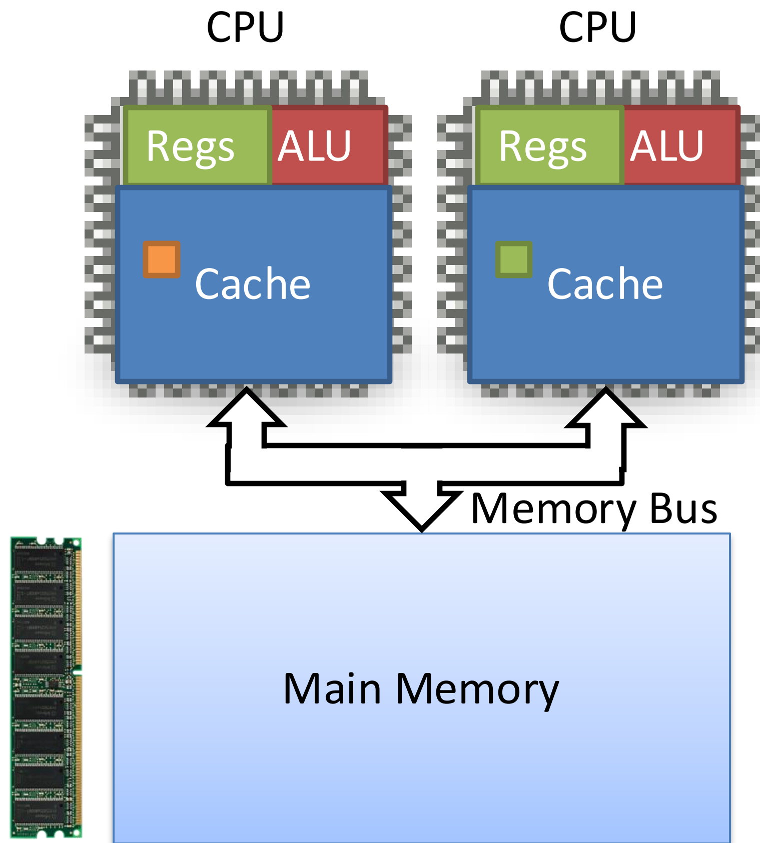
Sells better.
Servers/Desktops/Laptops

Cache Coherence



- Keeping multiple cores' memory consistent

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. We might discuss towards end of semester.

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!

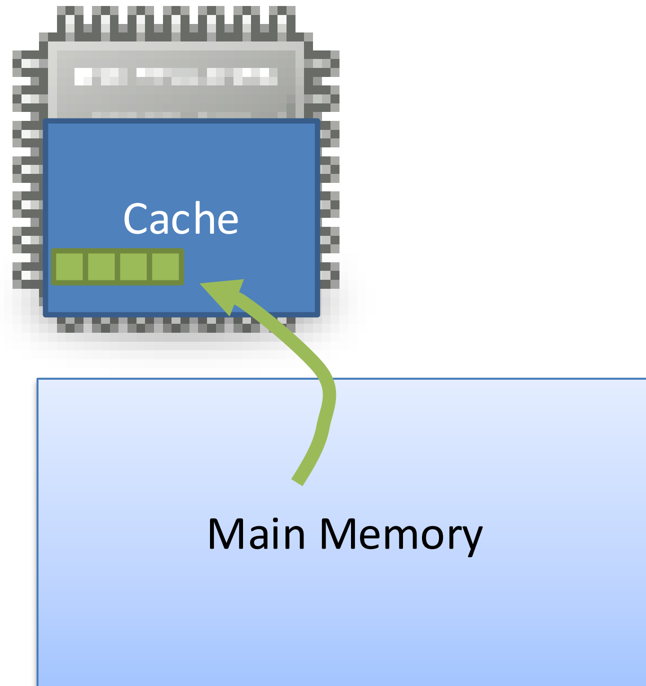
- When accessing memory, we check cache to see if it has the data we're looking for.

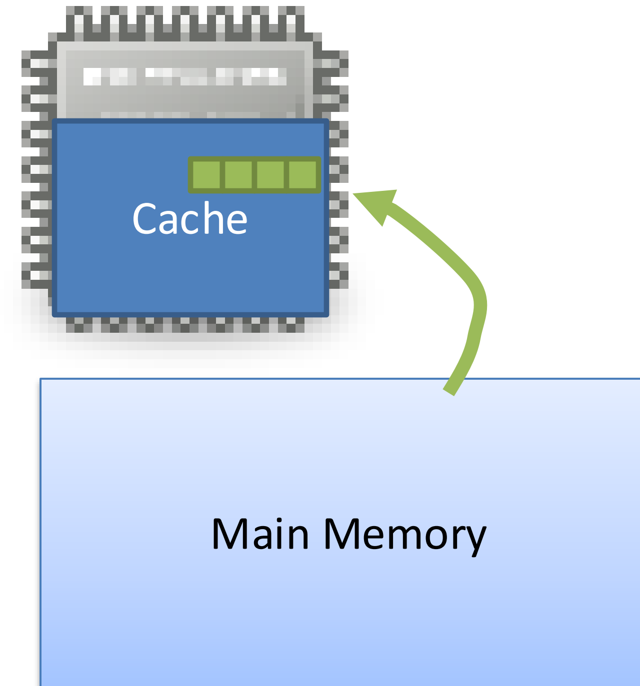
Why we miss…

- 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

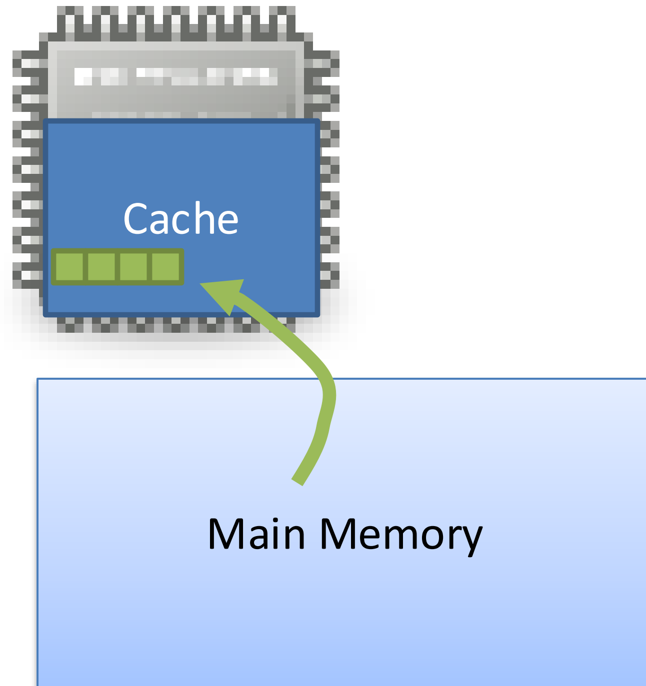
- Lot’s of characteristics to consider:
  – Where should data be stored in the cache?

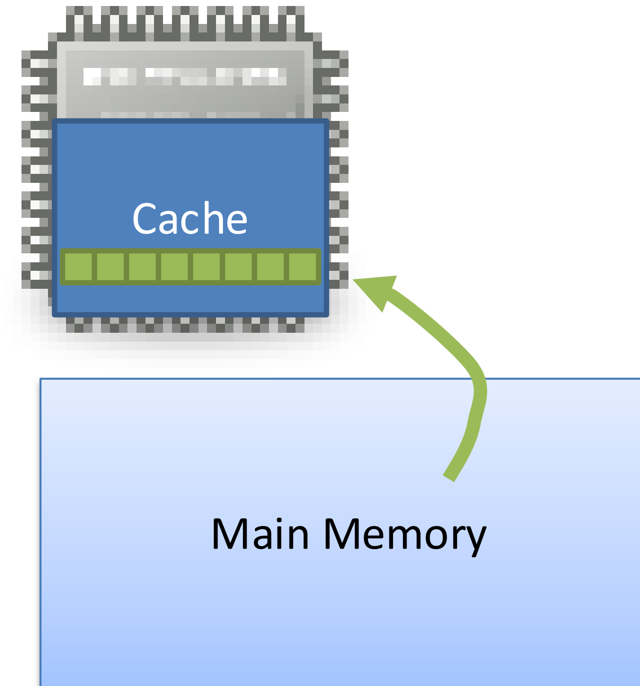




Cache Design

- 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

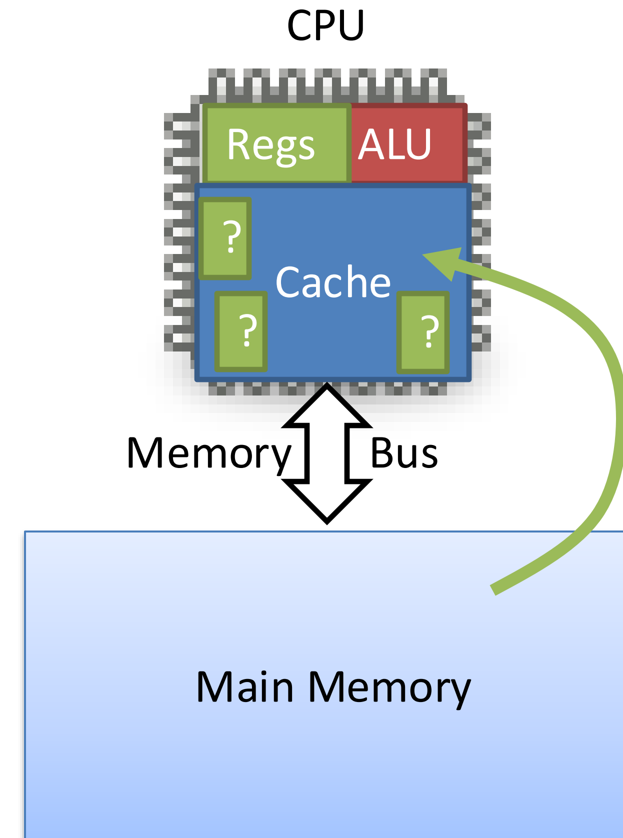
Suppose the CPU asks for data, it's not in cache.
We need to move in into cache from memory. Where in the cache should it be allowed to go?

A. In exactly one place.

B. In a few places.

C. In most places, but not all.

D. Anywhere in the cache.



A. Direct-Mapped: In exactly one place

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

B. Set-Associative: In a few places.

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

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

D. “Fully associative”: Anywhere in the cache.

- 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)

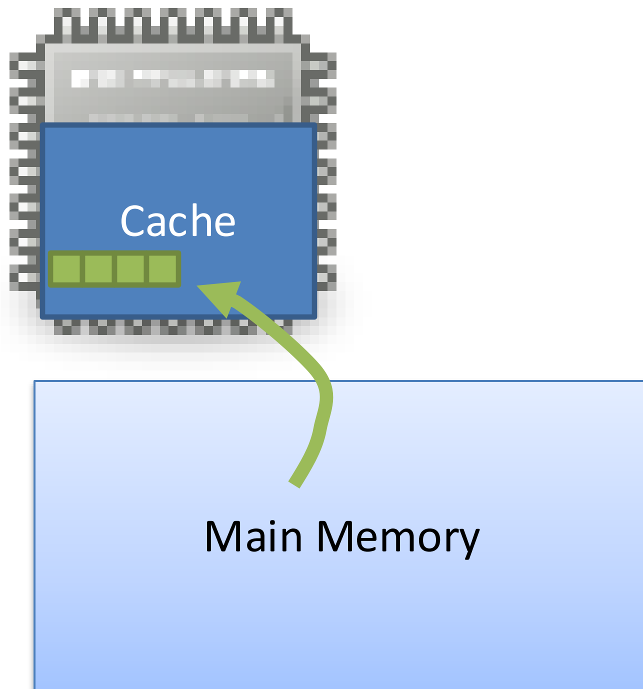
E. More than one of the above. (Which?)

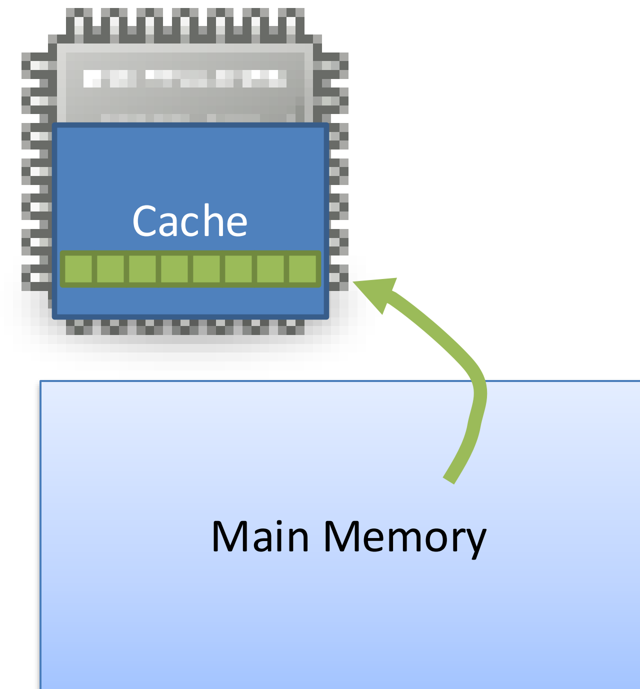
hard to make a determination
- don't know what the prog, is doing
- harmful if prog. does not exhibit good spatial locality

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!

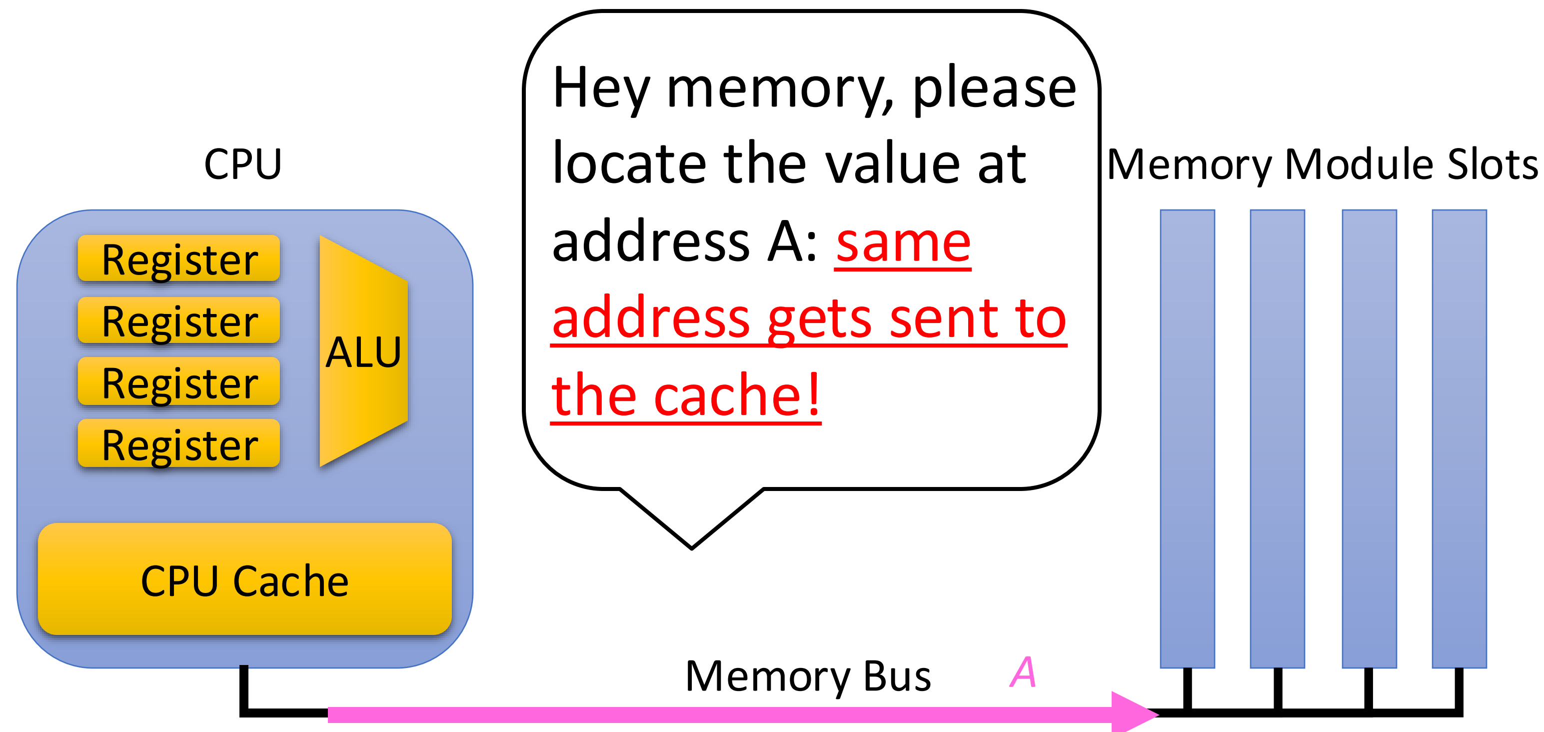
C. The size of the data.

D. Some other aspect of the data.

Recall: Memory Reads

CPU places address A on the memory bus.

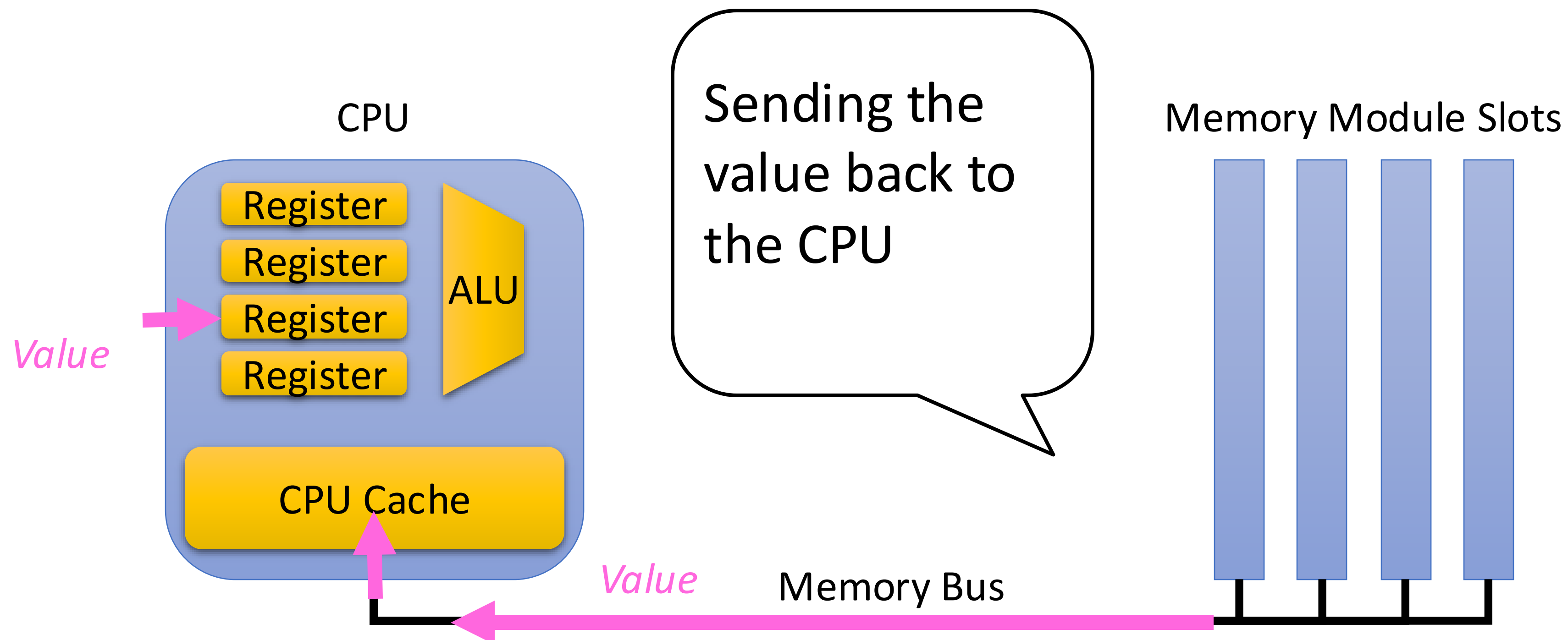
Load operation: `movl (A), %eax`



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.

- 64-bit address:

10110001 01110010 11010100 01010110 10110001 01110010 11010100 01010110

Divide into regions, each with distinct meaning.

Address Division

- First section: Tag
  - Of all the addresses that map to this location, which one is here?
  - Number of bits for this section is any bits left over after index and offset.

- Second section: Index
  - Which location(s) in the cache should we check for the data with this address?
  - Number of bits for this section depends on the number of cache locations.

- Third section: Offset
  - If we find a block of bytes in the cache (on a hit) which byte offset within the block do we actually want?
  - Number of bits for this section depends on the block size – must be able to uniquely identify every byte in the block.

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.

A. Anywhere in the cache. ("Fully associative")
   - No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.

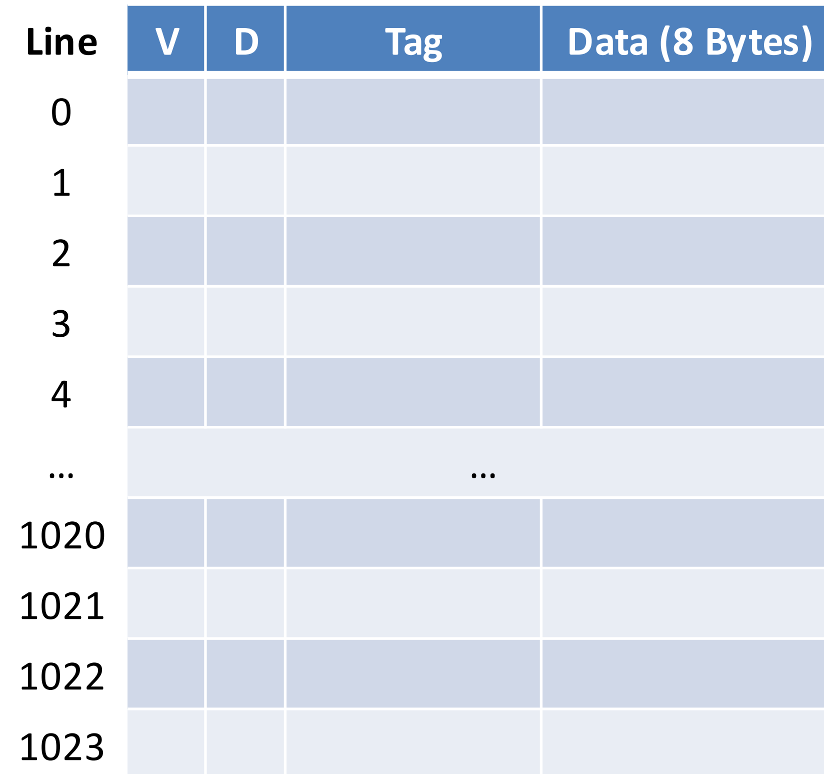
Direct-Mapped

- 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

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



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

Metadata

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

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
- Tag:
  - 64 - 13: 51 bits

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)

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)

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	

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 (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

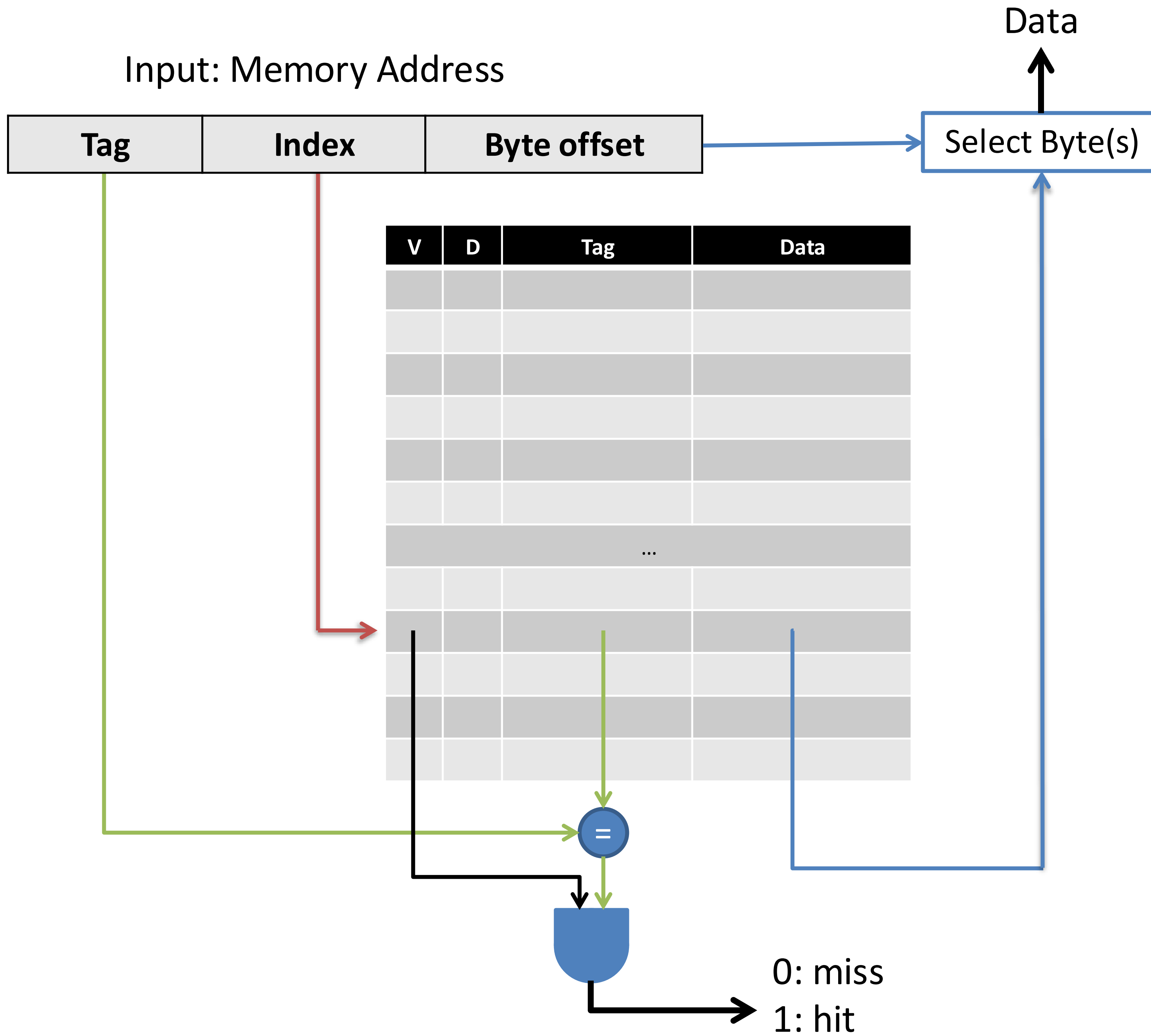
Address division:

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


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 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

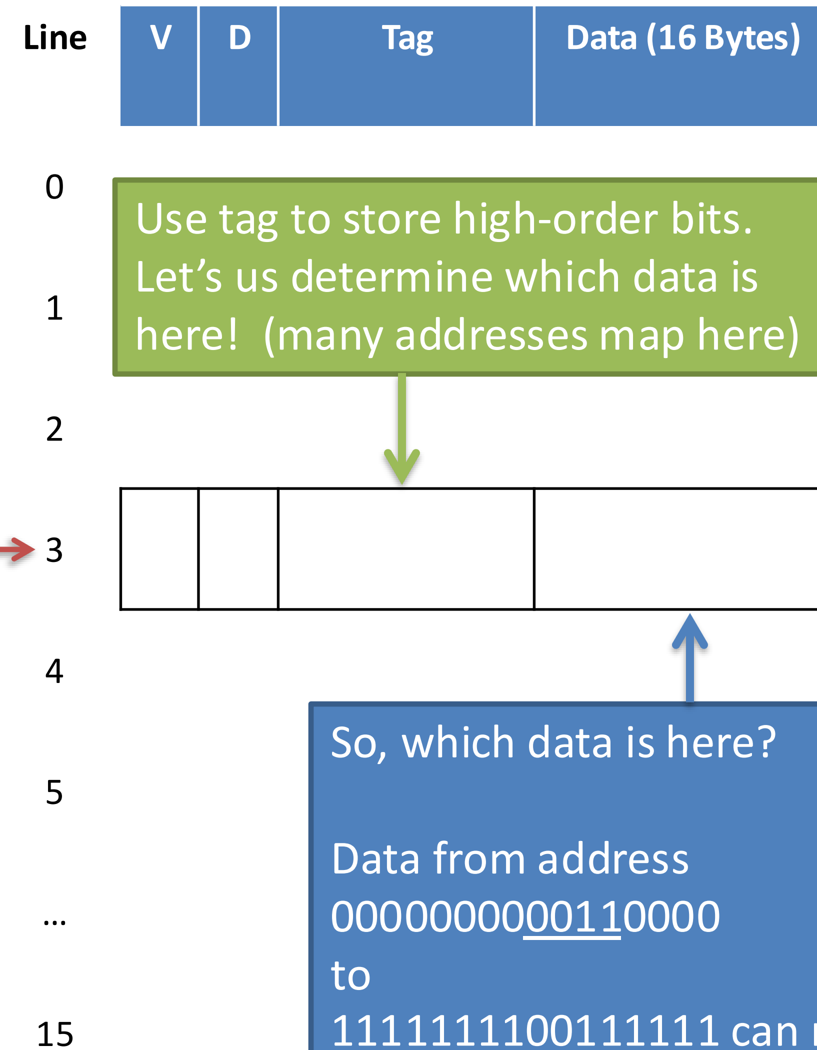
- Let's say we access memory at address:
  - 01101011 0011 0100
- Step 2:
  - Use index to find line (row)
  - 0011 -> 3

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

- 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)

Line	V	D	Tag	Data (16 Bytes)
0				
1				
2				
3			01101011	
4				
5				
...				
15				

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 <u>can</u> do.
  - Policy: Governs the decisions making for using the mechanism. What you <u>should</u> 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 (51 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	

Find line:

Tag doesn't match, bring in from memory.

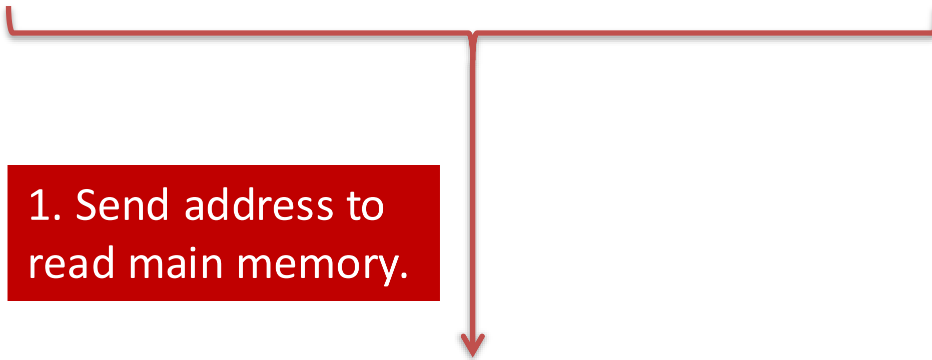
If dirty, write back first!

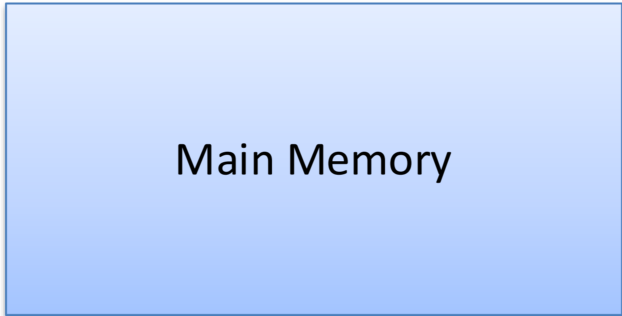
Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
…			…	
1020	1	0	1323	57883
1021				
1022				
1023				

Eviction: Direct-Mapped

- Address division:

Tag (51 bits)	Index (10 bits)	Byte offset (3 bits)
3941	1020	





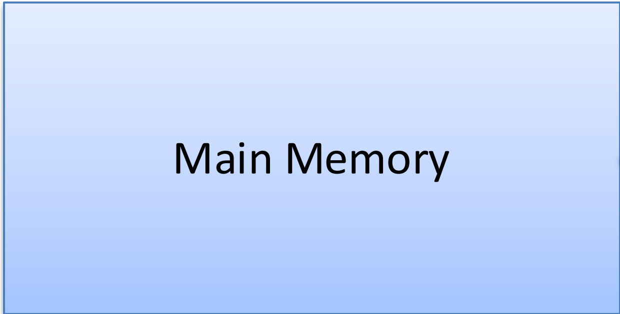
Line	V	D	Tag	Data (8 Bytes)
0				
1				
2				
3				
4				
…	…			
1020	1	0	1323	57883
1021				
1022				
1023				

Eviction: Direct-Mapped

- Address division:

Tag (51 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.

Direct-Mapped

- Address division:

Tag (51 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 (51 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?

No, recall mem. alignment!

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?

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

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)

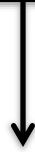
Read 10101010 (Value: 12)

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	~~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)

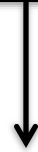
Read 10101010 (Value: 12)

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