CS250: Foundations of Computer Systems [Memory Hierarchy]

Caching and the Hierarchy
Memories fast, cut-rate, and copious
No such thing alas

Could we get by a lot more
With a lot less

Caches do so on their own
Nifty like its green-backed homophone

Powered by locality
Temporal and spatial
With the hierarchy
You can certainly get by

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Can RAM accesses get as fast as those to the CPU cache?
- Does main memory ever use SRAM?
- RAM, SRAM, DRAM, and main memory??
- Do computers always have L1, L2, L3 caches?
- Pins in a DIMM?

2

Topics covered in this lecture

- Main memory
- Memory addressing
- Speed differential across the memory hierarchy
- Midterm-I

Review: We looked at how many bits we need to address 64KB of memory

- We wish to know the value of n for $2^n = 65{,}536$
- The **inverse** of raising 2 to a certain power is the base-2 logarithm
- Therefore $\log_2(2^n) = n$ and $\log_2(65{,}536) = 16$
- Stated another way, $2^{16} = 65{,}536$
  - Therefore, a 16-bit memory address is needed to address 65,536 bytes

Why does the number of bits matter?

- The number of bits used to represent a memory address is a key part of a computer system's design
- It **limits the amount of memory** that a computer can access, and it impacts how programs deal with memory at a low level

Modern computing systems have 64-bit addressing

- A 64-bit address range should *suffice for a long time* as far as memory is concerned
- Unlikely you will ever put 2^{64} bytes of memory into a computer system and feel that you need more
  - 16,000,000 Terabytes or 16,000 Petabytes!
- Of course, people have made claims like this in the past
  - A few years ago, no one thought a computer would need 1GB of memory, yet computers with 64GB of memory (or more) are very common today

Let's say you end up needing even more addressing

- You would increase in powers of two
- You would first double to 128- and then …
  - 256- bits of addressing

Why 256-bits is effectively infinity …

- 2^{256} is effectively infinity for one simple reason
  - It's **physically impossible** to build that much memory based on estimates of the *current size* of the universe
    - About 2^{246} different elementary particles
- Unless you can attach 1 byte of memory to every elementary particle on the planet
  - You won't even come close to approaching 2^{256} bytes of memory on a given computer system
- Maybe we really will use whole planets as computer systems one day, as Douglas Adams predicted in *The Hitchhiker's Guide to the Galaxy*


MEMORY ADDRESSING
Suburbia is where the developer bulldozes out the trees, then names the streets after them.
— Bill Vaughan

9

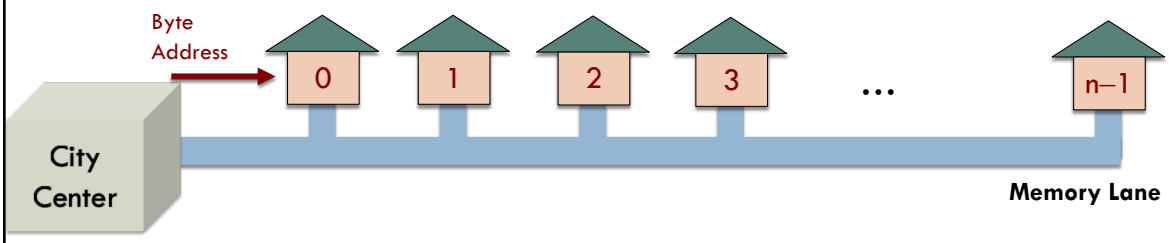
10

Memory is like a long street full of houses [1/2]

- Each house is exactly the same size and has room for a certain number of bits
- Building codes have pretty much settled on 1 byte per house
- And just like on a real street, each house has an address, which is just a number
- It's pretty common to refer to a **memory location**, which is just memory at a particular address, such as 3 Memory Lane

11

Memory is like a long street full of houses [2/2]



12

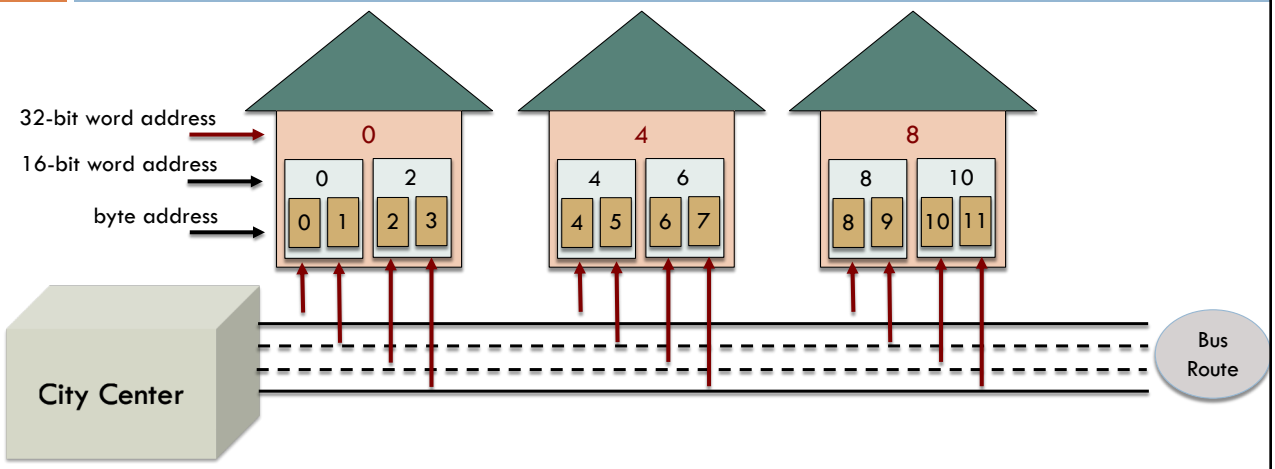
Just because the basic unit of memory is a byte doesn't mean we always look at it that way

- For e.g., 32-bit computers usually organize their memory in 4-byte **chunks**
  - While 64-bit computers usually organize their memory in 8-byte chunks
- Why does that matter?
  - It's like having a four- or eight-lane highway instead of a one-lane road
  - Huge, positive impact on **throughput**

More lanes …

- More lanes can handle more traffic because **more bits can get on the data bus**
- When we address memory, we need to know what we're addressing
- Addressing 32-bit words is different from addressing bytes because there are 4 bytes to that word on a 32-bit computer
  - And 8 bytes to a `long` word on a 64-bit computer

Another way to look at it [1/2]



Another way to look at it [2/2]

- The street in a 32-bit computer contains fourplexes, not single houses
- Each fourplex contains two duplexes
  - That means we can address an individual unit, a duplex, or a whole building

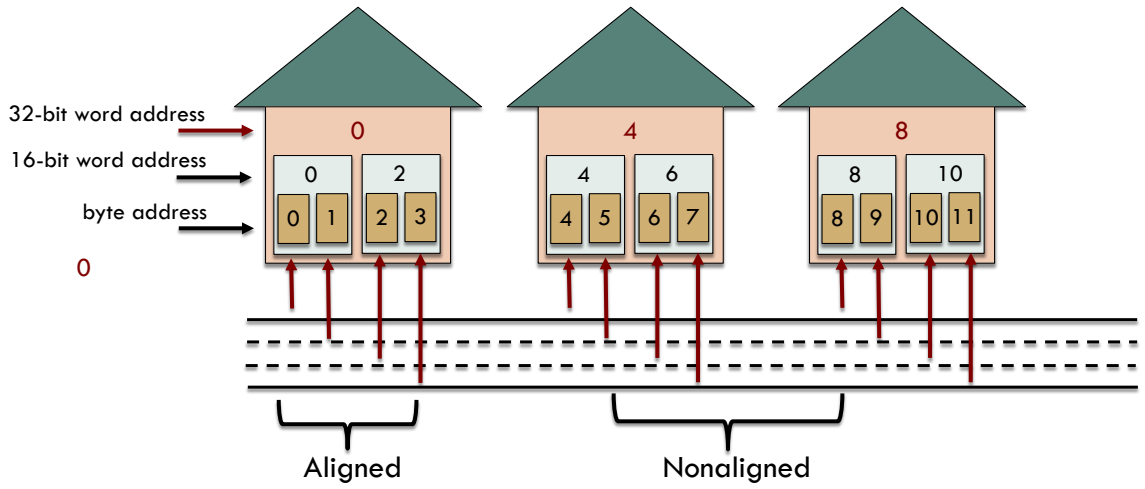
How commutes occur …

- Notice that each building straddles the highway such that **each byte has its own assigned lane**
  - And a 32-bit word takes up the whole road
- Bits commute to and from City Center on a bus that has four seats, *one for each byte*
- The doors are set up so that there's **one seat for each lane**
- On most modern computers, the bus stops only at one building on each trip from City Center

On most modern computers, the bus stops only at one building on each trip from City Center [1/2]

- This means we can't do things like form a word from bytes 5, 6, 7, 8
  - Because that would mean that the bus would have to make two trips: one to building 4 and one to building 8
- Older computers contained a complicated loading dock that allowed this, but planners noticed that it wasn't all that useful
  - Cut it out of the budget on newer models
- Trying to get into two buildings at the same time is called a **nonaligned access**

On most modern computers, the bus stops only at one building on each trip from City Center [2/2]



19

Who gets to sit in which seat when commuting on the bus? [1/2]

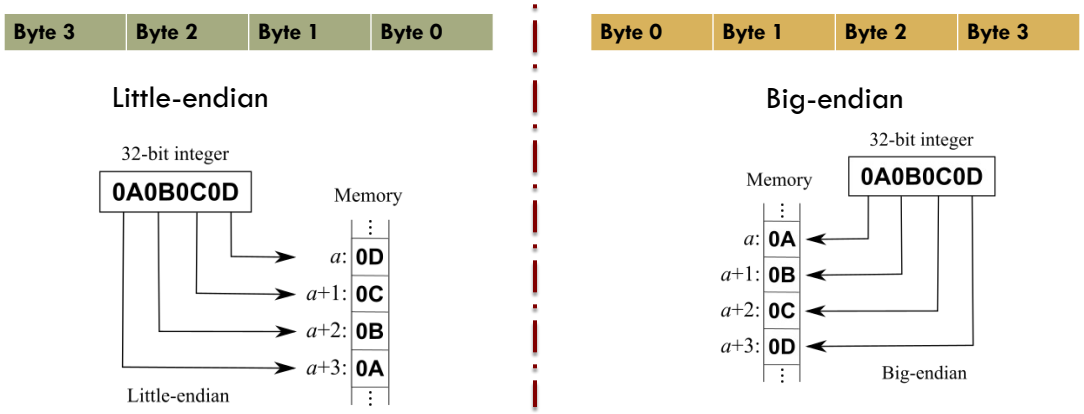
- Does byte 0 or byte 3 get to sit in the **leftmost** seat when a 32-bit word heads into town?
- It depends on the processor you're using, because designers have made them both ways
- Both work, so it's pretty much a theological debate
  - The **big-endian** and **little-endian** debate

20

Endian

- The term endian is based on the royal edicts in Lilliput and Blefuscu in Jonathan Swift's *Gulliver's Travels*
  - Regarding which was the proper end on which to crack open a soft-boiled egg
- Is used to describe the difference
  - Byte 0 goes into the rightmost seat in little-endian machines like Intel processors

Little-endian vs Big-endian

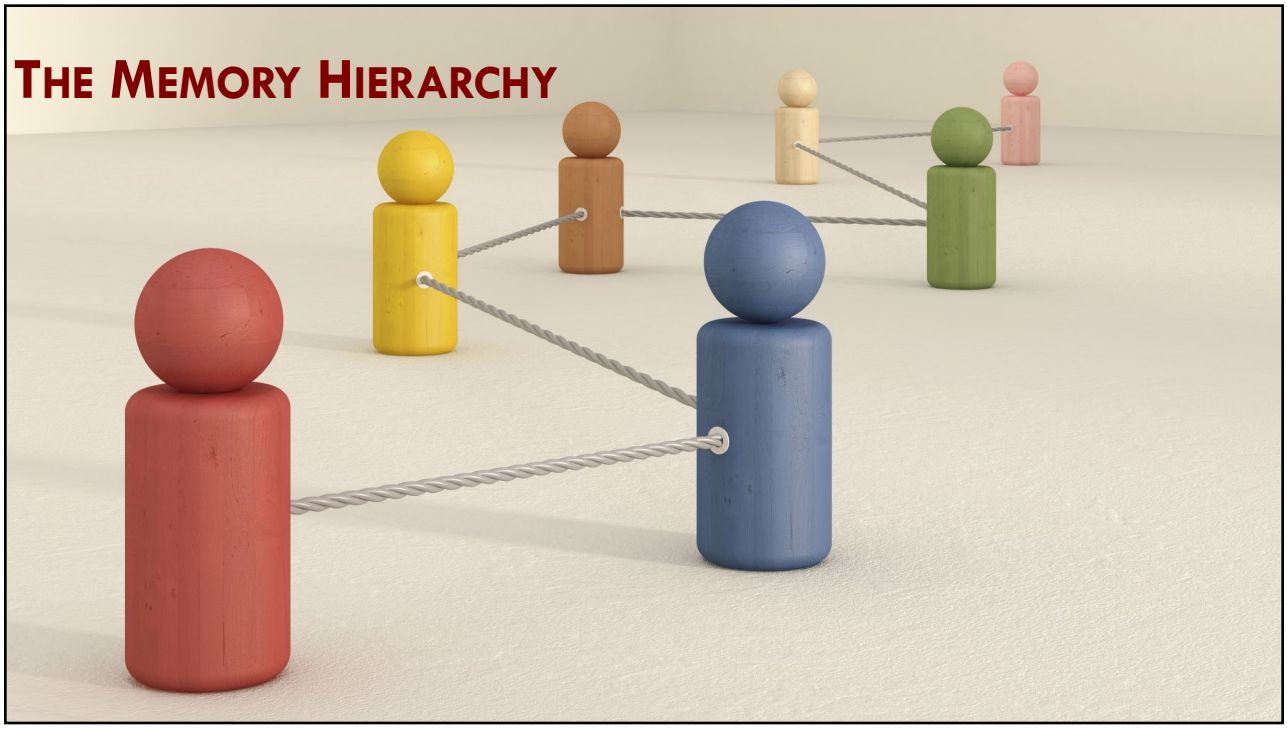


Endianness is something to keep in mind when you're transferring information from one device to another

- You don't want to *inadvertently* **shuffle** the data
- A notable instance of this occurred when the UNIX operating system was ported from the PDP-11 to an IBM Series/1 computer
  - A program that was supposed to print out "Unix" printed out "nUxi" instead, as the bytes in the 16-bit words got *swapped*
- This was sufficiently humorous that the term **nuxi syndrome** was coined to refer to byte-ordering problems

23

THE MEMORY HIERARCHY

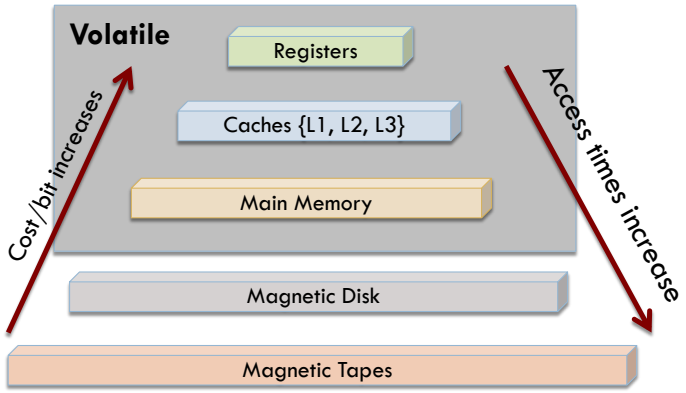


24

Most modern programs benefit by having a large amount of very fast memory

- Unfortunately, as a memory device gets **larger**, it tends to be *slower*
- Cache memories are **very fast**, but they are also *small and expensive*
- Main memory is inexpensive and large, but it is slow
- The **memory hierarchy** provides a way to compare the cost and performance of different types of memory

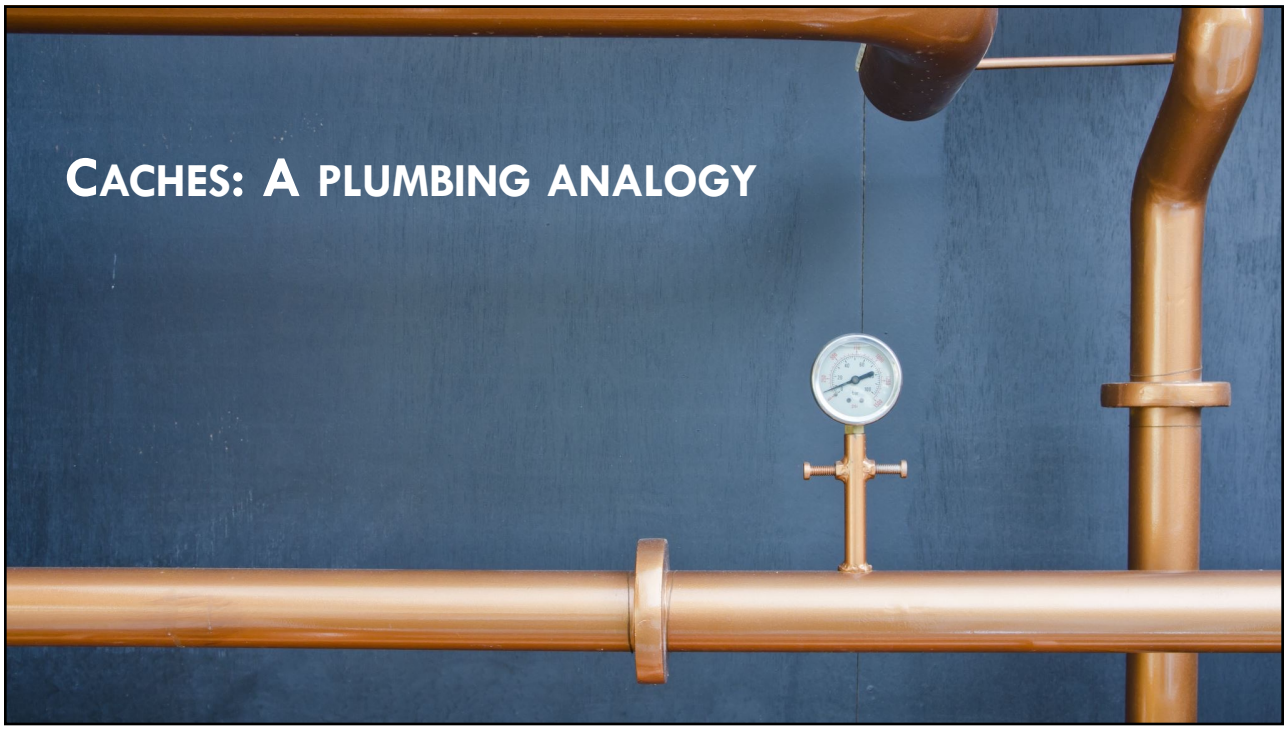
A high-level view of the memory hierarchy based on speed, cost, size and volatility



At the top level of the memory hierarchy are the CPU's general-purpose **registers**

- Registers provide the **fastest possible access** to data on the CPU
- The register is also the **smallest** memory object in the hierarchy
  - For example, the 32-bit 80x86 has just 8 general-purpose registers, and the x86-64 variants have up to 16 general-purpose registers
- Impossible to add more registers to a CPU
- CPUs have a very **limited number of registers**, and the cost per byte of register memory is quite high

27



28

Caches: A plumbing analogy

- Consider a plumber with a *toolbox* (a **cache**) that can hold four tools
- A number of the jobs being attended to are similar
  - So, the same four tools are *repeatedly used* (a **cache hit**)
- However, a significant number of jobs require additional tools; if the plumber does not know in advance what the job will entail?
  - Arrives and starts work
  - Partway through the job, the plumber needs an additional tool
  - Since it's not in the *toolbox* (**L1 cache**), plumber retrieves the item from the *van* (**L2 cache**)

Some jobs are tougher for the plumber

- Occasionally the plumber needs a special tool
  - Must leave the job, drive to the local *hardware store* (**global memory**), fetch the needed item, and return
- Plumber does not know *how long* (the latency) this operation will take
  - There may be *congestion* on the freeway and/or *queues* at the hardware store (other processes **competing for main memory access**)
  - Clearly, this is not a very efficient use of the plumber's time

While fetching new tools the plumber is not working on the problem at hand

- Each time a different tool or part is needed?
  - It needs to be fetched by the plumber from either the van or the hardware store
- While this might seem bad, fetching data from an HDD or an SSD is even worse, akin to **ordering an item** at the hardware store
  - In comparative form, data arrives by *snail mail*

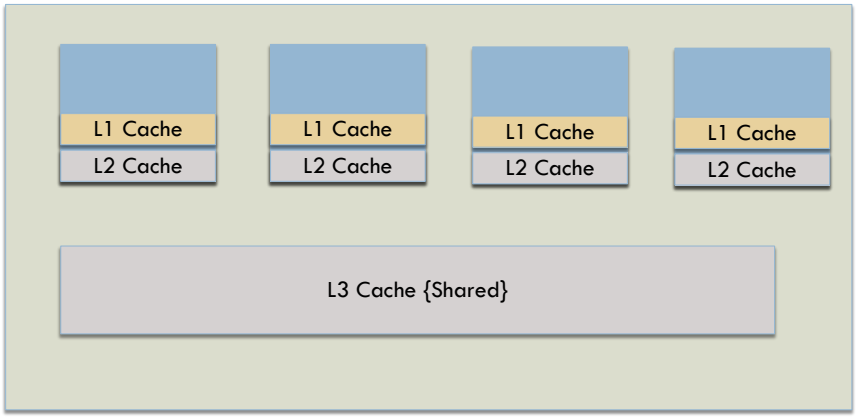
Typical latencies to global memory are in the order of hundreds of clocks

- Increasingly, the answer to this problem from traditional CPU processor design has been to increase the size of the cache
  - In effect, **arrive with a bigger van** so *fewer trips* to the hardware store are necessary
- There is, however, an **increasing cost** to this approach
  - Both in terms of **capital outlay** for a larger van and
  - The time it takes to **search** a bigger van for the tool/part

The approach taken by most CPU designs today?

- Arrive with a *van* (**L2 cache**) and a *truck* (**L3 cache**)
- In the extreme case of the server processors?
  - A huge 18-wheeler is brought in to try to ensure that the plumber is kept busy for just that little bit longer

Logical view of the Processor, Cores, and Caches



** Intel Meteor Lake class of CPUs have an L4 cache

L1: Typically in the order of 4KB-32 KB

L2, L3: Typically in the order of MBs

Working our way down from registers: L1 Cache

- The level-one (L1) cache system is the **next highest performance** subsystem in the memory hierarchy
- As with registers, the CPU manufacturer usually provides the L1 cache **on the chip**, and you **cannot expand it**
- Its size is **usually small**
  - Typically, between 4KB and 32KB
  - Though this is much larger than the register memory available on the CPU

More about the L1 cache

- The L1 cache size is **fixed** on the CPU
- The cost per cache byte is much lower than the cost per register byte
  - Because the cache contains more storage than is available in all the registers combined

Level-two (L2) cache is present on some CPUs, but not all [1/2]

- For example, Intel i3, i5, i7, and i9 CPUs include an L2 cache as part of their package
  - But some of Intel's older Celeron chips do not
- The L2 cache is generally **much larger than the L1 cache**
  - For e.g., 256KB to 1MB as compared with 4KB to 32KB

Level-two (L2) cache is present on some CPUs, but not all [2/2]

- On CPUs with a built-in L2 cache, the cache is **not expandable**
- It still costs less than the L1 cache
  - Because we **amortize** the cost of the CPU across all the bytes in the two caches, and the L2 cache is larger

The level-three cache (or the L3 cache)

- Level-three (L3) cache is present on all but the oldest Intel processors
- The L3 cache is **larger** still than the L2 cache
  - Typically shared across all the cores
  - Usually, 1-8 MB

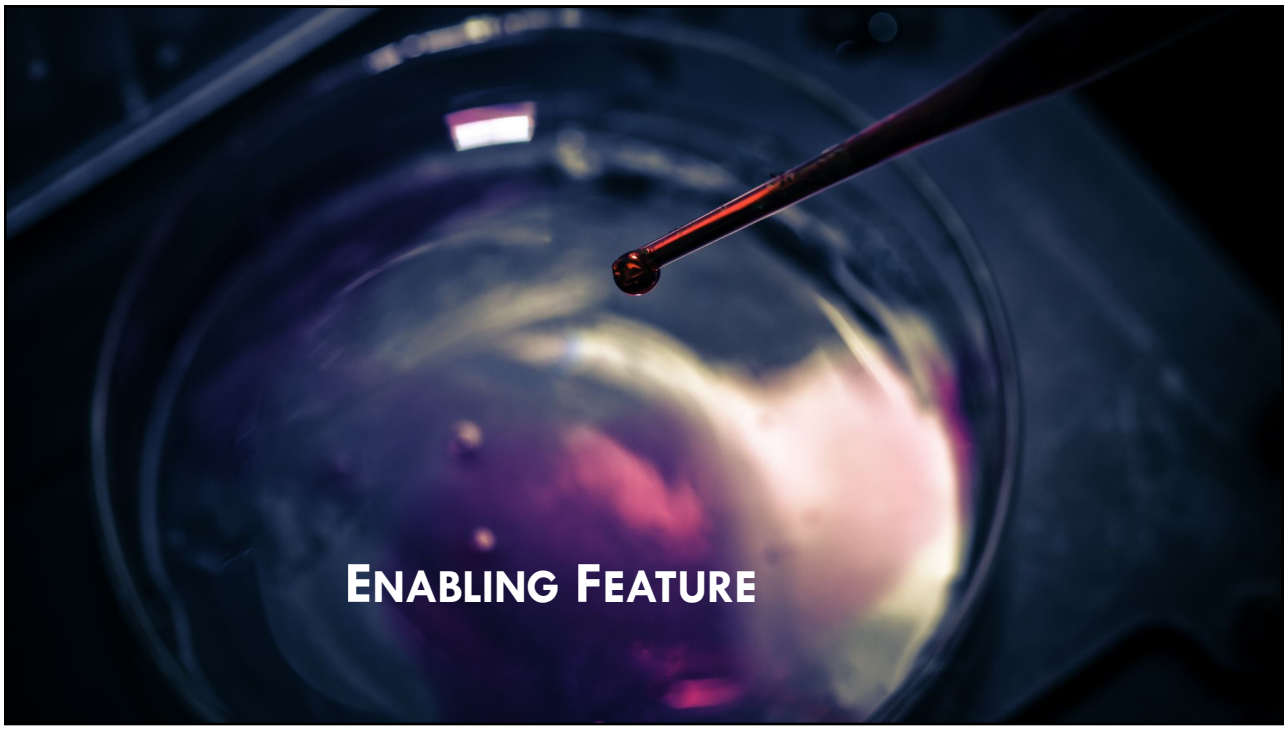
The main-memory subsystem comes below the cache system in the memory hierarchy

- Main memory is the general-purpose, relatively low-cost memory found in most computer systems
  - Typically, DRAM or something similarly inexpensive
  - **Many differences in main-memory technology** with speed variations
- The main-memory types include standard DRAM, synchronous DRAM (SDRAM), double data rate DRAM (DDRAM), DDR3, DDR4, and so on
- Generally, you **won't find a mixture** of these technologies in the same computer system

The whole point of the memory hierarchy is to allow reasonably fast access to a large amount of memory

- If only a little memory were necessary, we'd use fast static RAM (the circuitry that cache memory uses) for everything
- If speed wasn't an issue, we'd use **main memory** for everything

41



42

Enabling feature: Locality [1/2]

- Caches work on the principle of either spatial (close in the address space) or temporal (close in time) locality
- Thus, data that has been *accessed before*, will likely be accessed again (**temporal locality**)
- Data that is *close to the last accessed data* will likely be accessed in the future (**spatial locality**)
- Caches work well where the task is repeated many times

Enabling feature: Locality [2/2]

- The memory hierarchy enables us to take advantage of the principles of **locality**
  - Move *frequently referenced* data **into fast memory** and
  - Leave *rarely referenced* data in **slower memory**
- Unfortunately, during the course of a program's execution, **the sets** of oft-used and seldom-used data *change*
  - This is often referred to as the **working-set** model

The sets of oft-used and seldom-used data change

- We can't simply distribute our data throughout the various levels of the memory hierarchy when the program starts and then leave the data alone as the program executes
  - No fill-it-once and forget-about-it-forever solution
- Instead, the different memory subsystems need to be able to **accommodate changes** in spatial locality or temporality of reference
  - During the program's execution by **dynamically moving data** between subsystems

Moving data between the registers and memory is strictly a program function

- The program loads data into registers and stores register data into memory using machine instructions like `mov`
- It is the programmer's or **compiler's responsibility** to keep heavily referenced data in the registers as long as possible
  - The CPU will not automatically place data in general-purpose registers in order to achieve higher performance

Program responsibility and obliviousness [1/2]

- Programs explicitly control access to registers, main memory, and memory-hierarchy subsystems that are at the file storage level or lower
- Programs are **largely unaware** of the memory *hierarchy between* the register level and main memory

Program responsibility and obliviousness [2/2]

- In particular, cache accesses are **transparent** to the program
  - Access to these levels of the memory hierarchy usually occurs *without any intervention* on a program's part
- Programs simply access main memory, and the hardware and operating system take care of the rest
  - Really? Yes!!!

Responsibility for data movements

- If a program always accesses main memory, it will run slowly
  - Modern DRAM main-memory subsystems are **much slower** than the CPU
- Cache memory subsystems and the CPU's cache controller **move data** between main memory and the L1, L2, and L3 caches
  - So that the CPU can quickly access oft-requested data
- Likewise, it is the **virtual memory** subsystem's responsibility to move oft-requested data from hard disk to main memory

Mechanics of transparent data accesses

- With few exceptions, most memory subsystem accesses take place **transparently between**
  - One level of the memory hierarchy and
  - the *level immediately below or above it*

For example, the CPU rarely accesses main memory directly

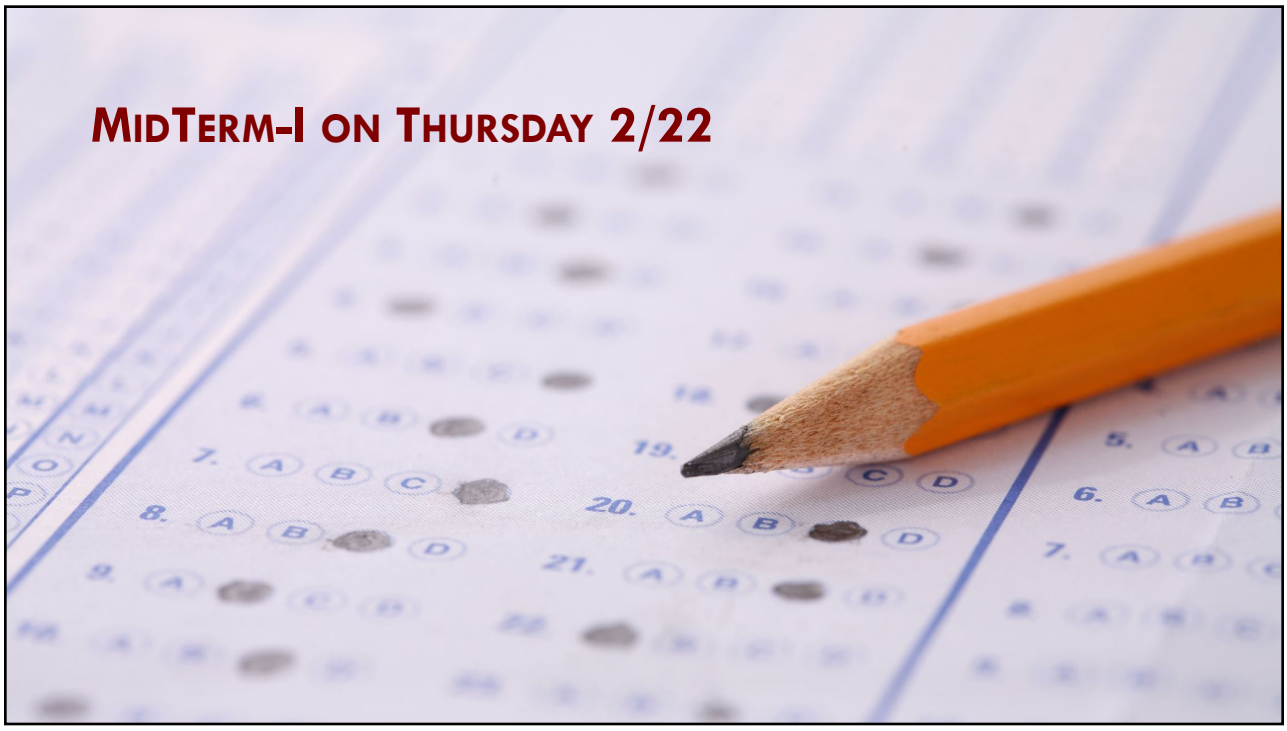
- Instead, when the CPU requests data from memory, the L1 cache subsystem takes over
- If the requested data is in the cache
  - **Cache hit**
  - The L1 cache subsystem returns the data to the CPU, and that concludes the memory access
- If the requested data is not in the cache?
  - **Cache miss**!

If the requested data isn't present in the L1 cache …

- The L1 cache subsystem passes the request down to the L2 cache subsystem
- If the L2 cache subsystem has the data?
  - The L2 Cache returns this data to the L1 cache, which then returns the data to the CPU
- Requests for the same data *in the near future* will be fulfilled by the L1 cache rather than the L2 cache
  - Because the L1 cache **now has a copy** of the data

What if the L2 cache does not have it?

- After the L2 cache, the L3 cache kicks in
- If none of the L1, L2, or L3 cache subsystems have a copy of the data, the request goes to main memory
  - If the data is found in main memory, the main-memory subsystem passes it to the L3 cache
    - The L3 cache then passes it to the L2 cache, which then passes it to the L1 cache
    - The L1 cache which then passes it to the CPU
- Once again, the data is now in the L1 cache, so any requests for this data *in the near future* will be fulfilled by the L1 cache



Will account for 15% of your course grade

- Part 1: Number representations
- Part 2: Boolean Logic and Boolean Algebra
- Part 3: Memory basics and the hierarchy [all topics incl. lecture on 2/20]
- Each accounts for 25 points
  - The exam itself is for 75 points
- If you have SDC accommodations, please ensure that they are aware that you will take the test at the University Testing Center

The contents of this slide-set are based on the following references

- Jonathan E. Steinhart. *The Secret Life of Programs: Understand Computers -- Craft Better Code*. ISBN-10/ ISBN-13 : 1593279701/ 978-1593279707. No Starch Press. [Chapter 4]
- Randall Hyde. Write Great Code, Volume 1, 2nd Edition: Understanding the Machine 2nd Edition. ASIN: B07VSC1K8Z. No Starch Press. 2020. [Chapter 11]
- Matthew Justice. *How Computers Really Work: A Hands-On Guide to the Inner Workings of the Machine*. ISBN-10/ISBN-13 : 1718500661/ 978-1718500662. No Starch Press. 2020. [Chapter 7]
- Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of GPU Computing)*. ISBN-10/ISBN-13: 0124159338/978-0124159334. 1st Edition. Morgan Kaufmann. [Chapters 2,3]