CS250: Foundations of Computer Systems [Memory]

Caching and Locality

Access a data item once
Likely, then, you are to access it again
Later in time (temporality)

Access a data item once
The one close to it will likely be accessed next
Spatial proximity

Caching's secret sauce? locality
Spatial, temporal or
Some combination thereof

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- How much of the RAM (main memory) is used by the Operating System?
- What if something is deleted from main memory? Will the copy of that be deleted from the caches?
- How do large datasets (multi terabyte, for e.g.) get managed?
- Why is data that is missing from the cache put into it the first time that you access it?
  - Why not wait to see if it is actually being accessed more often?

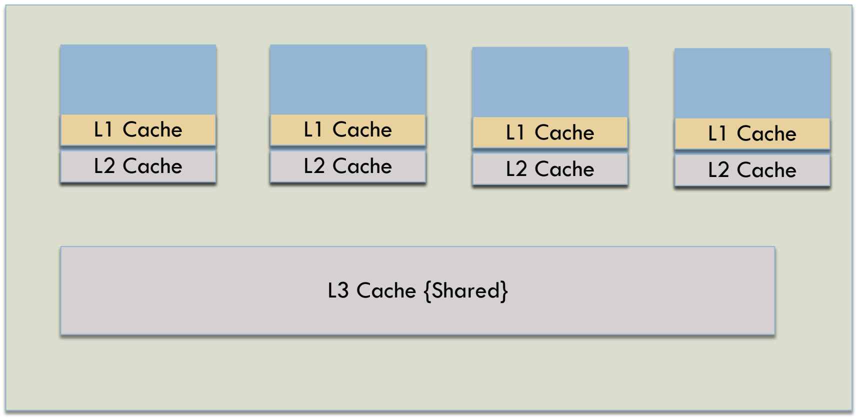
2

Topics covered in this lecture

- The memory hierarchy
- Enabling feature in caching
- Caching
  - Direct mapped
  - Associative
  - N-way associativity

3

Logical view of the Processor, Cores, and Caches



L1: Typically in the order of 4KB-32 KB **L2, L3**: Typically in the order of MBs

4

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

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.7

7

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

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.8

8

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

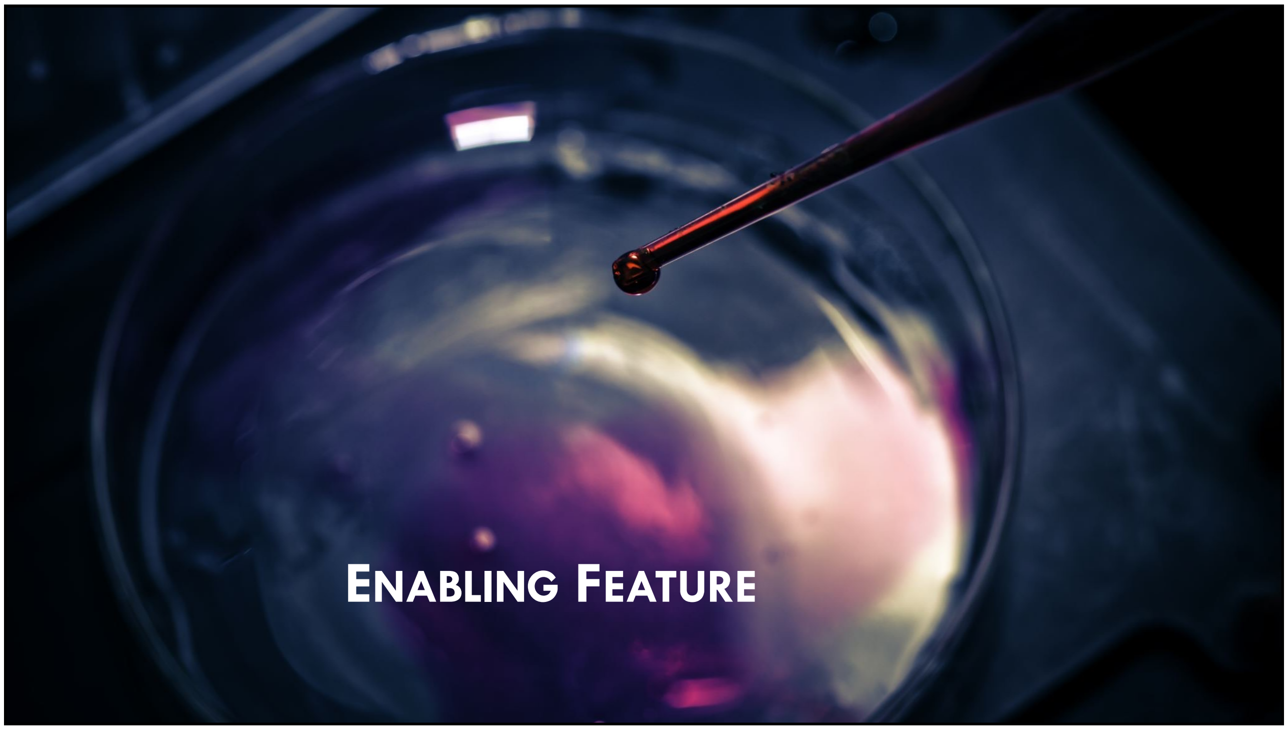
The main-memory subsystem comes below the cache system in the memory hierarchy

- 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 DRAM-based **main memory** for everything

11



12

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

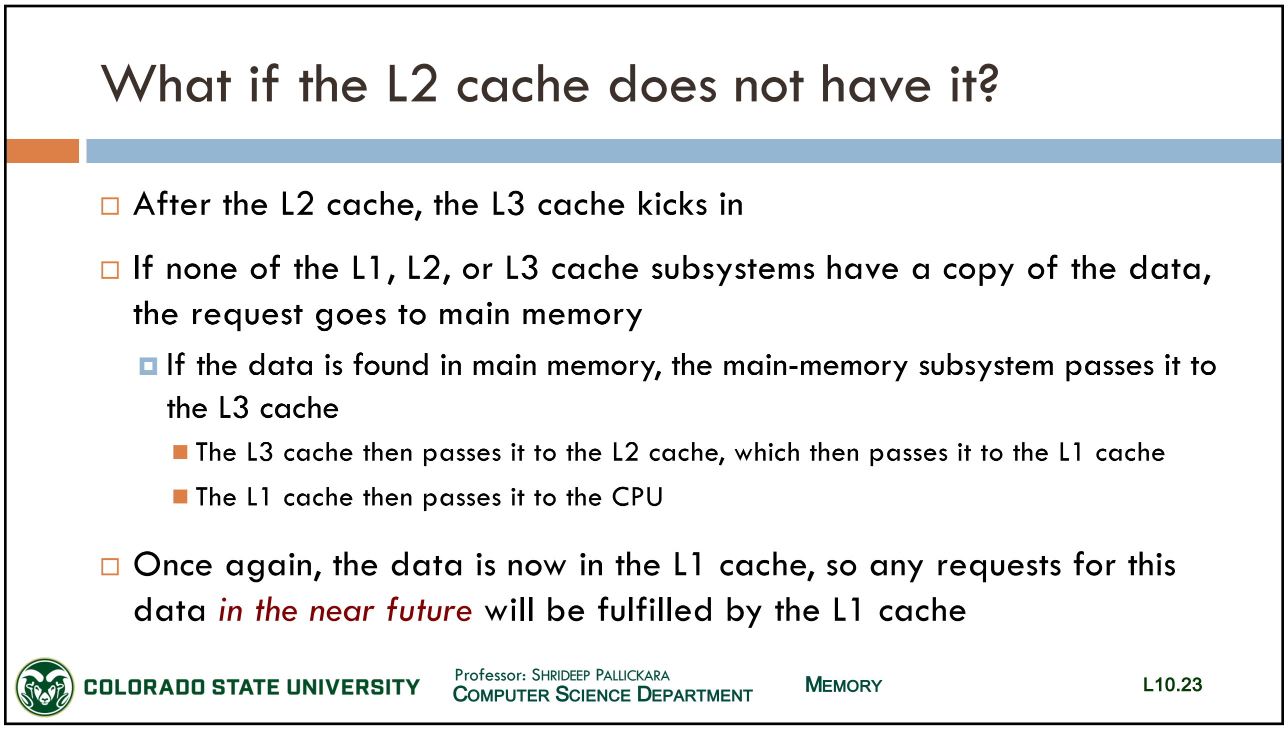
- With few exceptions, most memory subsystem accesses take place **transparently between**
  - One level of the memory hierarchy and
  - the *level immediately below or above it*

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 …

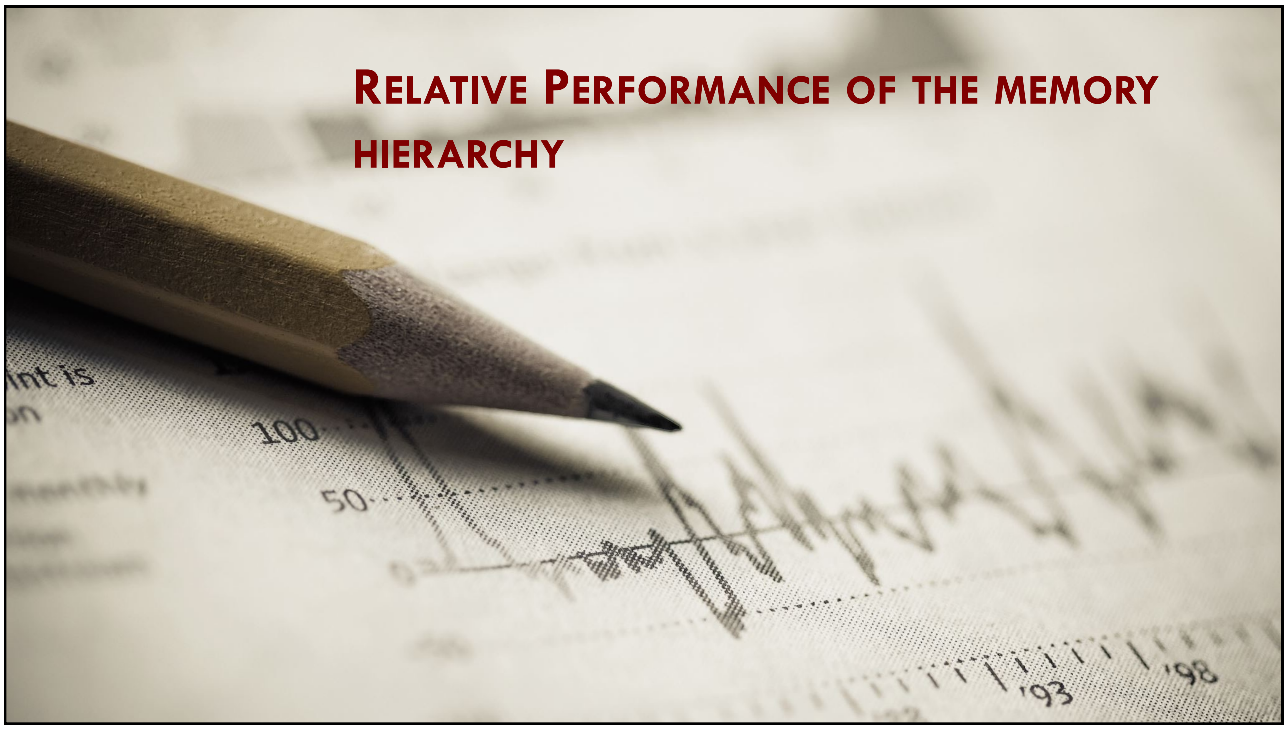
- The L1 cache subsystem passes the request down to the L2 cache subsystem
- If the L2 cache subsystem has the data?
  - The L2 Cache returns this data to the L1 cache, which then returns the data to the CPU
- Requests for the same data *in the near future* will be fulfilled by the L1 cache rather than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

23



RELATIVE PERFORMANCE OF THE MEMORY HIERARCHY

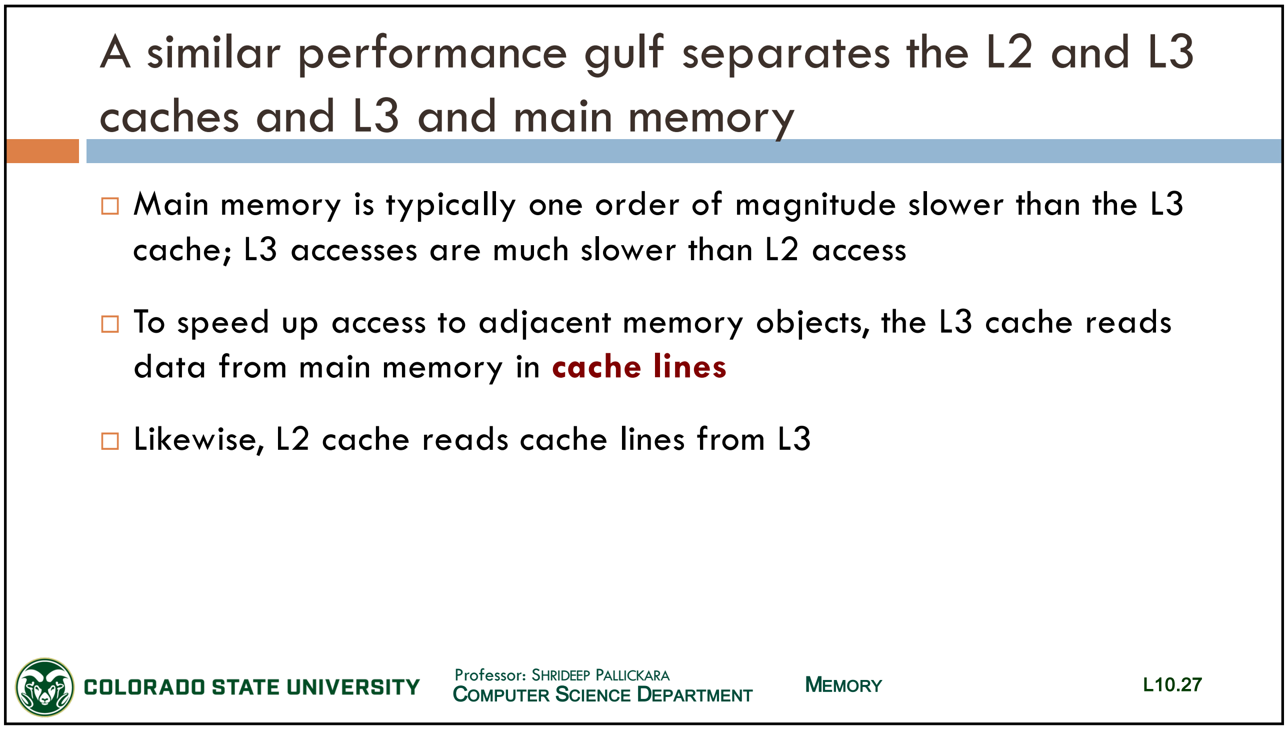
24

Relative performance of the memory system

- Registers are, unquestionably, the best place to store data you need to access quickly
  - Accessing a register **never requires any extra time**, and
  - Most machine instructions that access data can access register data
- The difference in speed between the L1, L2, and L3 cache systems isn't so dramatic unless the secondary or tertiary cache is not packaged together on the CPU

There are several reasons why L2 cache accesses are slower than L1 accesses

- It takes the CPU time to **determine** that the data it's seeking is not in the L1 cache
  - By the time it does that, the memory access cycle is nearly complete, and there's no time to access the data in the L2 cache
- The circuitry of the L2 cache may be **slower than the circuitry of the L1** cache in order to make the L2 cache less expensive
- L2 caches are usually 16 to 64 times larger than L1 caches
  - Larger memory subsystems **tend to be slower** than smaller ones
  - All this amounts to additional wait states for accessing data in the L2 cache



A similar performance gulf separates the L2 and L3 caches and L3 and main memory

- Main memory is typically one order of magnitude slower than the L3 cache; L3 accesses are much slower than L2 access
- To speed up access to adjacent memory objects, the L3 cache reads data from main memory in **cache lines**
- Likewise, L2 cache reads cache lines from L3

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.27

27

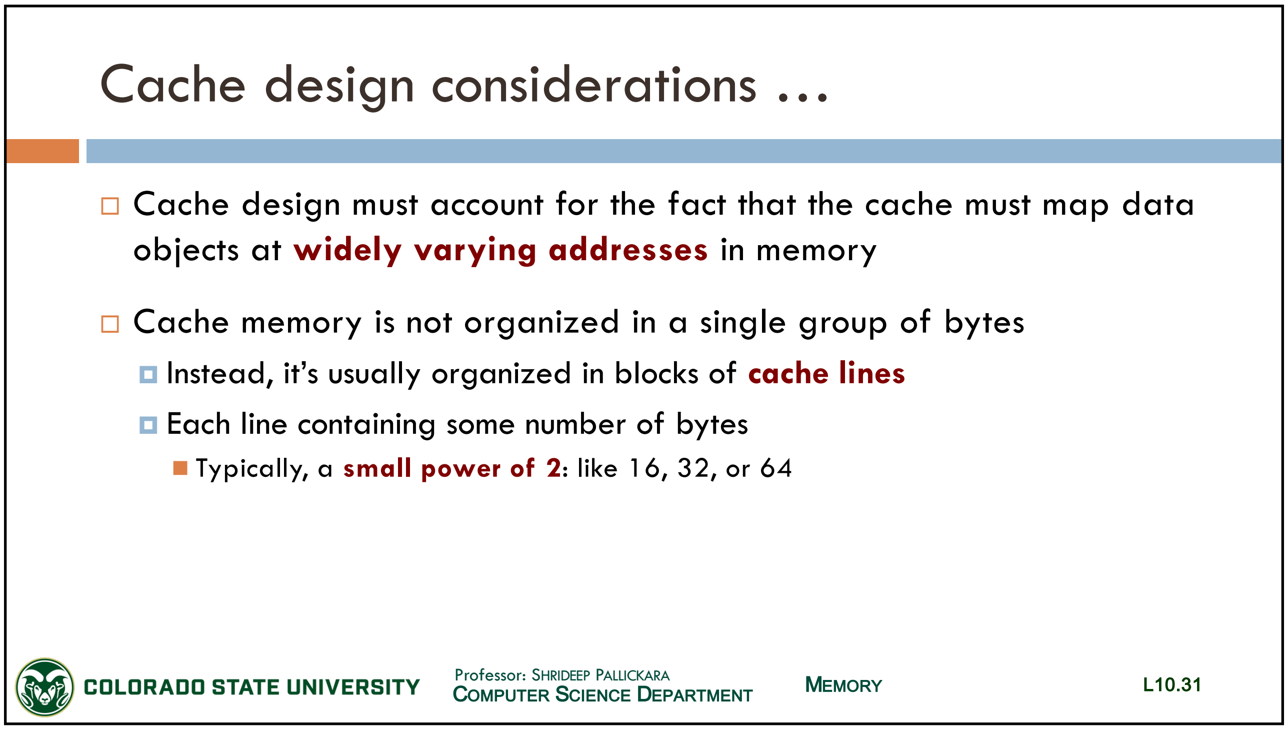
28

Up until this point

- We have treated the cache as a magical place that
  - Automatically stores data when we need it, perhaps fetching new data as the CPU requires it
- But **how** exactly does the cache do this?
  - And **what happens when it is full,** and the CPU is requesting additional data that's not there?

Programs access only a small amount of data at a given time

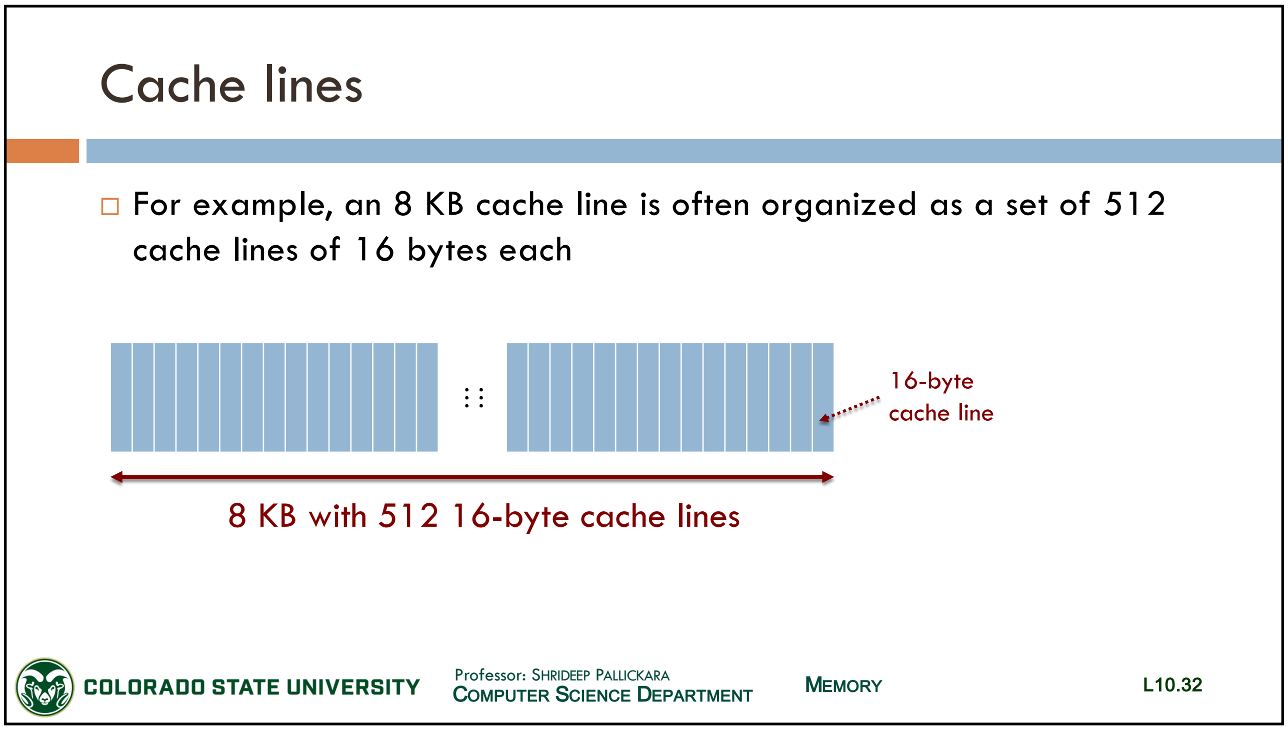
- A cache that is sized accordingly will improve their performance
- Unfortunately, the data that programs want **rarely sits in contiguous memory location**
  - It's usually *spread out* all over the address space



Cache design considerations …

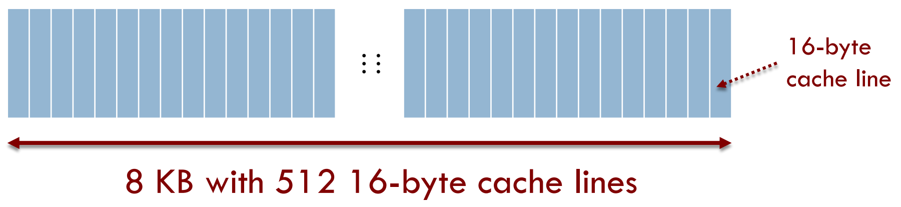
- Cache design must account for the fact that the cache must map data objects at **widely varying addresses** in memory
- Cache memory is not organized in a single group of bytes
  - Instead, it's usually organized in blocks of **cache lines**
  - Each line containing some number of bytes
    - Typically, a **small power of 2**: like 16, 32, or 64

31



Cache lines

- For example, an 8 KB cache line is often organized as a set of 512 cache lines of 16 bytes each



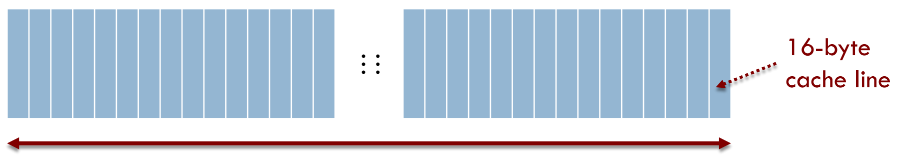
32

We can attach a different noncontiguous address to each of the cache lines

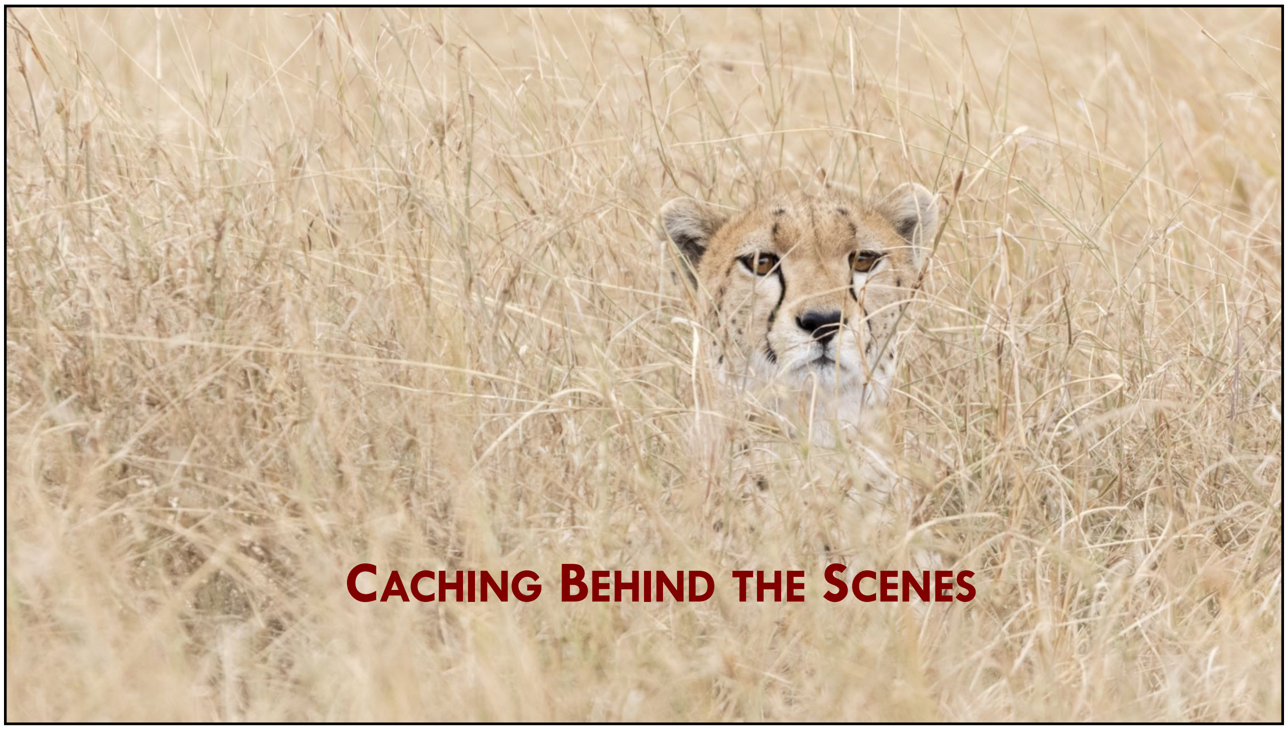
- Cache line 0 might correspond to addresses 0x10000 through 0x1000F
- Cache line 1 might correspond to addresses 0x21400 through 0x2140F

Generally, if a cache line is n bytes long

- It will hold n bytes from main memory that fall on an ***n*-byte boundary**
- In our example of 16-byte cache lines, a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory
  - i.e., the least-significant 4 bits of the address of the first byte in the cache line are always 0



8 KB with 512 16-byte cache lines



CACHING BEHIND THE SCENES

35

Types of caches

- Direct mapped caches
- Fully associative caches
- N-way associative caches

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.36

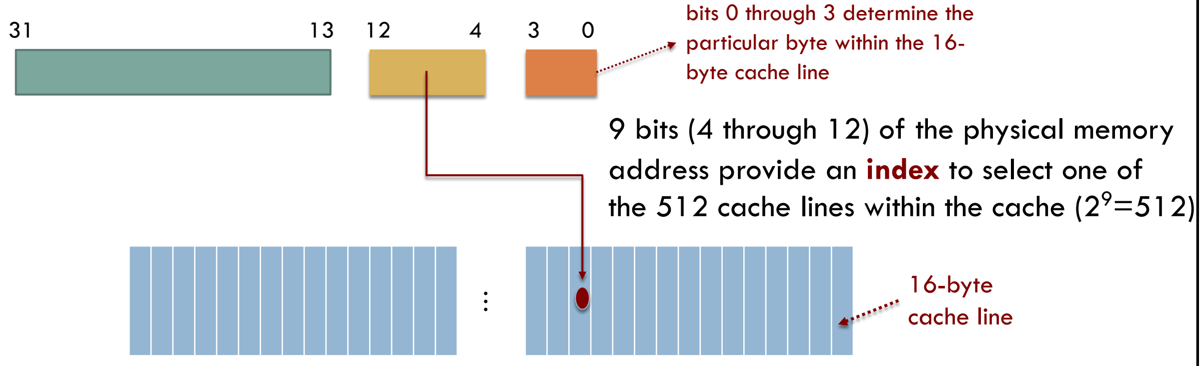
36

A **direct mapped cache** is also known as a one-way associative cache

- In a direct-mapped cache, a particular block of main memory is always loaded into—mapped to—the exact same cache line
- This mapping is determined by a small number of bits in the data block's memory address

37

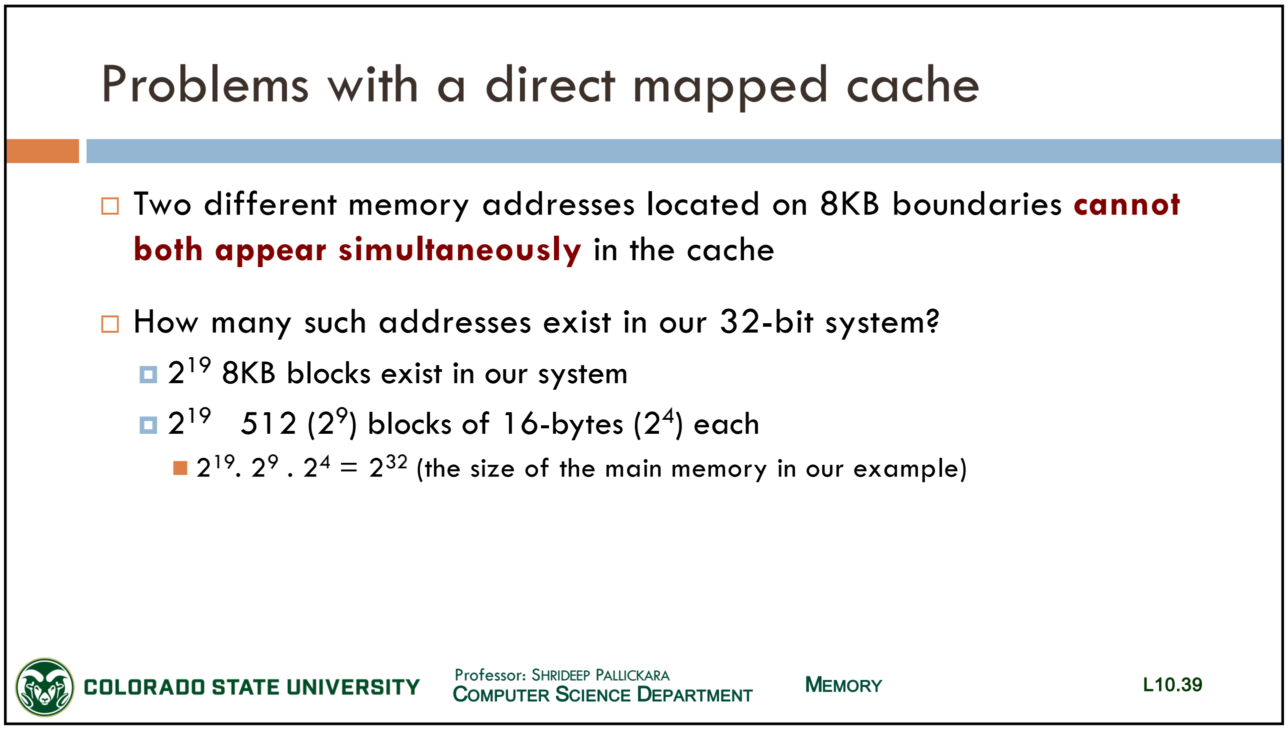
Direct-mapped Cache


31
13
12
4
3
0
bits 0 through 3 determine the particular byte within the 16-byte cache line
16-byte cache line

9 bits (4 through 12) of the physical memory address provide an **index** to select one of the 512 cache lines within the cache (2^9=512)

8 KB with 512 16-byte cache lines

38

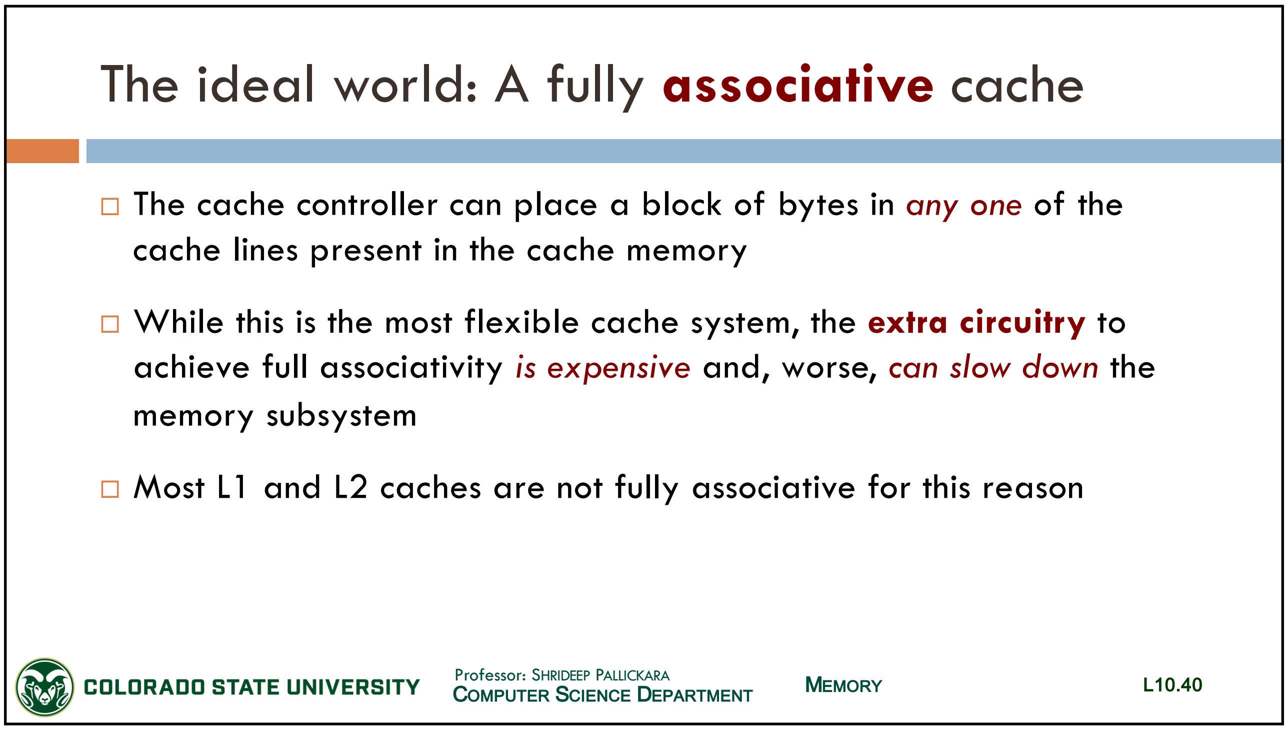


Problems with a direct mapped cache

- Two different memory addresses located on 8KB boundaries **cannot both appear simultaneously** in the cache
- How many such addresses exist in our 32-bit system?
  - 2^{19} 8KB blocks exist in our system
  - 2^{19} 512 (2^9) blocks of 16-bytes (2^4) each
    - $2^{19}. 2^9 . 2^4 = 2^{32}$ (the size of the main memory in our example)

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.39

39



The ideal world: A fully **associative** cache

- The cache controller can place a block of bytes in *any one* of the cache lines present in the cache memory
- While this is the most flexible cache system, the **extra circuitry** to achieve full associativity *is expensive* and, worse, *can slow down* the memory subsystem
- Most L1 and L2 caches are not fully associative for this reason

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT MEMORY L10.40

40

Trade-off space

- A fully associative cache is too complex, too slow, and too expensive to implement
- But a direct-mapped cache is too inefficient

A compromise: the **n-way associative cache**

- In an n-way set associative cache, the cache is broken up into sets of n cache lines
- The CPU determines the **particular set to use** based on
  - Some subset of the memory address bits, just as in the direct-mapping scheme, and …
  - The cache controller uses a fully associative mapping algorithm to determine which one of the n cache lines within the set to use

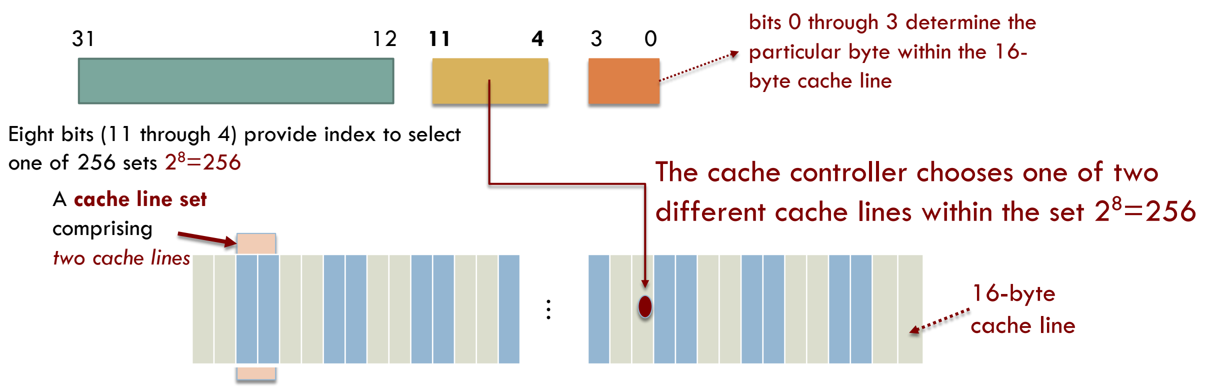
For example, an 8KB two-way set associative cache subsystem with 16-byte cache lines [1/2]

- Organizes the cache into **256** cache-line sets with **two cache** lines each
- Eight bits from the memory address determine which one of these 256 different sets will contain the data
  - $2^8 = 256$
- Once the cache-line set is determined, the cache controller maps the block of bytes to one of the two cache lines within the set

For example, an 8KB two-way set associative cache subsystem with 16-byte cache lines [2/2]

- This means **two different memory addresses** located on 8KB boundaries (addresses having the same value in bits 4 through 11) can both appear simultaneously in the cache
- However, a **conflict** will occur if you attempt to access a *third* memory location at an address that is an even multiple of 8KB

An 8 KB two-way set associative cache subsystem with 16-byte cache lines


31
12
11
4
3
0
bits 0 through 3 determine the particular byte within the 16-byte cache line
Eight bits (11 through 4) provide index to select one of 256 sets 2^8=256
A **cache line set** comprising *two cache lines*
The cache controller chooses one of two different cache lines within the set 2^8=256
16-byte cache line

8 KB with 2-way set associate cache with 256 sets of two (16-byte) cache lines each

45

What if we have a 4-way associative cache

- A four-way set associative cache puts four associative cache lines in each cache-line set
- In our example, 8KB cache, a four-way set associative caching scheme would have **128 cache-line sets** with **four cache lines** each
- This would allow the cache to maintain up to four different blocks of data without a conflict, each of which would map to the same cache line in a direct-mapped cache

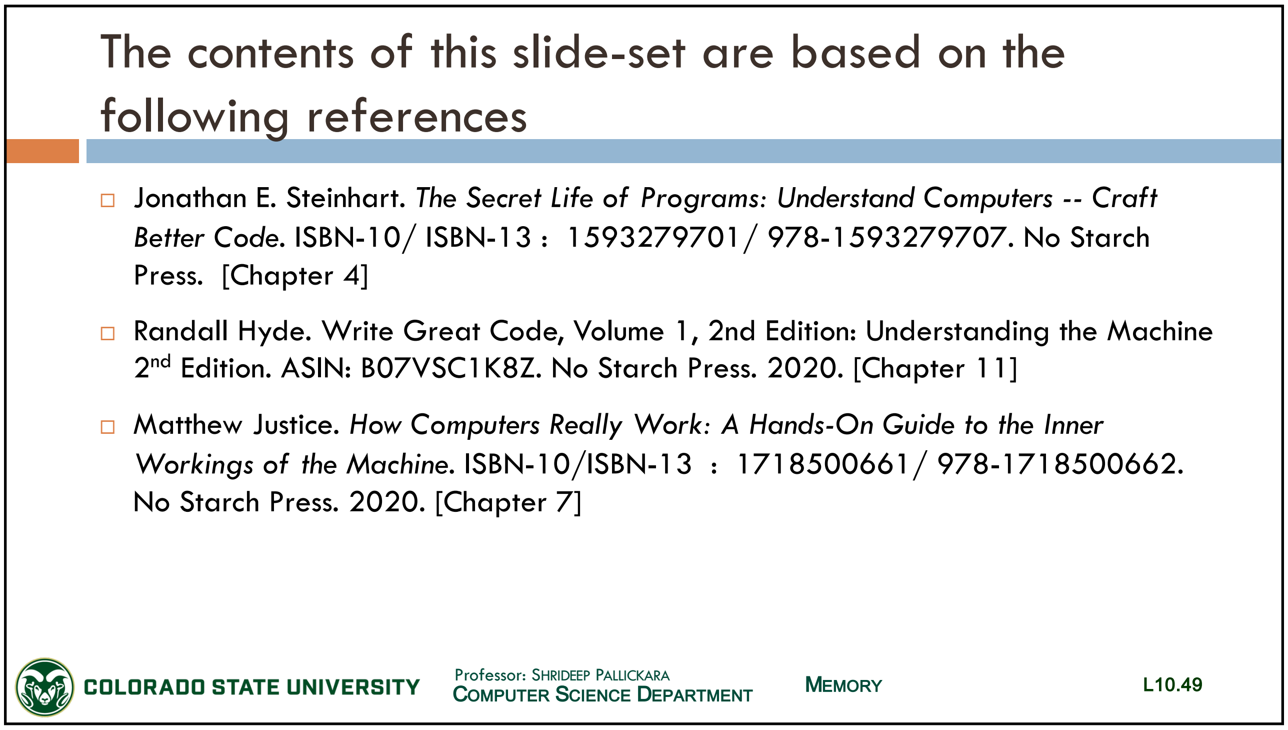
46

2-/4-way set associative vs direct mapped

- A 2- or 4-way set associative cache is
  - Much better than a direct-mapped cache and
  - Considerably less complex than a fully associative cache

Can we keep increasing the number of lines in each cache-line set?

- The more cache lines we have in each cache-line set, the closer we come to creating a fully associative cache
  - With all the attendant problems of complexity and speed
- Most cache designs are direct-mapped, two-way set associative, or four-way set associative
  - The various members of the 80x86 family make use of all three



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]