Data Storage

- As mentioned, computer science involves the study of algorithms and getting machines to perform them — before we dive into the algorithm part, let's study the machines that we use today to do our bidding
- In the past, these machines were based on gears, punch cards, beads, or other mechanical artifact; from the vacuum tube onward, we've relied on *electronic* devices, and these devices have two main states: they're either *on* or *off*
- On/off…current/no current…true/false…one/zero

1s and **0**s

- At the lowest, physical level, computers only recognize two values: one or zero
- *Everything* you've experienced with today's computers are ultimately expressed as patterns of **1**s and **0**s
- Having only two fundamental values means that we have a *binary* system
- Compare this with our *decimal* system, where an individual numeric symbol, or *digit*, can have *10* possible values (0 to 9)

Bits and PCs

- A single **1** or **0** is said to occupy a *bit*, short for *binary digit* (as opposed to the *decimal digits* that we use)
- 4 bits is sometimes called a "nybble;" 8 bits is a *byte*
- Physical phenomena — light, colors, force, movement, sounds (vibrations) — are *analog* or *continuous*, falling into an infinitely precise range of values or intensities; to represent them in a computer, they need to be *digitized* — expressed as sequences of distinct bits
  - Devices that place data into a computer (mice, scanners, microphones) can be viewed as *analog-to-digital* (A/D) converters; devices that send data *from* a computer (monitors, printers, amps) perform the reverse, converting digital (bits) to analog (light, sound waves)

From Binary to Boolean

- Another well known set of concepts is binary in nature: *true* vs. *false* (yet another synonym: *dichotomy*)
- Thus, the concept of **1** vs. **0** is interchangeable with the concept of true vs. false: **1** corresponds to true while **0** corresponds to false
- Mathematical logic has defined well-known operations for combining true and false values — these are called *boolean operations*, in honor of George Boole (1815–1864), one of the subject's pioneers

And, *Or*, *Not*, But Not *But* (!)

- Mathematical (a.k.a. boolean) logic gives exact definitions for what *and*, *or*, and *not* mean
- In fact we get an additional operation called *xor* ("exclusive *or*") for which we don't have a one-word English equivalent
- To see the "truth tables" for *and*, *or*, and *xor*, see Brookshear Figure 1.1
- There is no boolean operation for "but" — does the above title make sense now?

Gates and Flip-flops

- So now we have the conceptual framework: values (true/false, **1**/**0**) and operations (*and*, *or*, *xor*, *not*)
- We've seen that boolean values "exist" in the real world via electronic signal or current
- Boolean operations also have real-world counterparts, in the form of *gates* and *flip-flops* — devices that take *input* signals, then produce an *output* signal
- The actual devices span a variety of technologies, so on paper they are represented with standard symbols — see the book for details

Hexadecimal Notation

- In the same way that our decimal system expresses numbers as a sequence of decimal digits ("0" to "9") multiplied by powers of 10…
  - $728 = 7(10^2) + 2(10^1) + 8(10^0)$
- …we can also interpret streams of bits as sequences of values multiplied by powers of 2:
  - $\mathbf{1001} = \mathbf{1}(2^3) + \mathbf{0}(2^2) + \mathbf{0}(2^1) + \mathbf{1}(2^0) = 9$
- This can get pretty long — it takes 4 bits to express 16 different values, where decimal needs only 2 digits

- Fortunately, our *Hindu-Arabic* system of notation enables easy translation from one base to another if the new base is an exact power of the old one
- With the binary system, we have powers of 2: $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, and so on
- Note how, if we "cluster" bits into subgroups, you can make their notation shorter without completely losing their underlying bit representation
- With computers, we use groups of 4 bits, thus expressing 16 distinct values per digit, as opposed to 2 in binary or 10 in decimal — this is *base 16*, or the *hexadecimal* system, where the digits are "0" to "9" and we co-opt "A" to "F" to represent the values 10 to 15

The Joy of Hex

With hexadecimal notation (or "hex" for short), conversion is easy (vs. converting from binary to decimal) — let's look at the binary stream "**1011011000**"

- Conversion to decimal requires adding the powers of 2 for which a bit is "**1**": $2^9 + 2^7 + 2^6 + 2^4 + 2^3 = 728$
- Conversion to hex requires that we cluster the bits into groups of 4, then just translate the groups:
  - **1011011000** clusters into **0010 1101 1000**
  - Each group of 4 becomes a digit: 2D8

- So, does "10" stand for the decimal value 2, 10, or 16? Let us count the ways…
  - One notation standard is to use a subscript for the base; thus 10_2 is 2, 10_{10} is 10, 10_{16} is 16, and so on
  - But one can't really write subscripts in computer programs (as you'll see), so alternative notations include prefixes (%10, *0x*10) and suffixes (10*h*)
  - In general, the context of a number will tell you the base that is being used
- 2 hex digits (and thus 8 bits) form a *byte*
- Terms like "32-bit" or "64-bit" typically refer to how many bits a machine, processor, or device can handle in a single operation
- Most importantly — hex notation is *primarily for human convenience*: at the level of the machine, it's still all bits
  - Sometimes, *octal* notation (3 clustered bits per cluster = base 8) is also used as shorthand

Main Memory

- As mentioned, all information on a computer — whether documents, images, sound, video — is ultimately represented as sequences or patterns of bits
- The circuitry that keeps track of these bits while you (and thus the computer) are actively working on them is called *main memory*
- Main memory can be thought of as an incredibly long sequence of bits — in practice, we group the bits 8 at a time, so we typically think of, measure, and count main memory in terms of bytes instead of bits

- Within a byte, we imagine the bits arranged from left to right; typically, the leftmost bit is viewed as the *high-order* or *most-significant* bit, because if the byte were interpreted as a binary (base 2) number, then that bit is the one with the highest power, 2^7 or 128
- To indicate which byte in main memory we're talking about, we count them off from the beginning; the number of required "steps" from the beginning in order to reach a particular byte is called its *address*
- You may have noticed that the book starts with Chapter Zero; this is a common convention in computer science — we think of the first item of most lists as the "zero-th" item instead of the "1st" one, because it is "zero steps from the first item"

Memory Terms and Measurements

- Physical memory devices consist of circuitry that can store these billions of bytes; the circuitry is set up so that the time it takes to get to any byte is independent of its address — thus, such memory is called *random access memory* or RAM
  - Contrast this with camcorder or VCR tapes, where it takes longer to get to the end than the beginning (assuming that the tape starts out being fully rewound)
- Some types of memory require an electronic charge the flow through the circuitry in order to maintain the bit patterns; when you turn off the power, the data goes away — this is *volatile* or *dynamic* RAM
- Memory that can retains its data without power is called *non-volatile* RAM (NVRAM) — examples include the RAM in some music players and cell phones
  - Sometimes the data is "burned" into the memory; you can't change it because it is actually part of the circuitry — this is *read-only memory* or ROM
- Just as the metric system uses prefixes to cluster (or divide) units of measure (e.g., kilometers, milligrams), we also group bytes into larger units using the same prefixes: *kilo-*, *mega-*, *giga-*, *tera-*, *peta-*, *exa-*, *zetta-*, *yotta-*
  - But there's a twist: instead of an exact power of 10 (e.g., 10^3 for *kilo-*), these prefixes have historically referred to the power of 2 that is *nearest* to that power of 10 — so *kilobyte* is actually 1024 (2^{10}) bytes, *megabyte* is really 1048576 (2^{20}) bytes, and so on
  - Needless to say, this can get very confusing, especially since other units of measure do use the exact power of 10 (e.g., 1 kilogram is 1000 grams, period)
  - To address this, there is a movement to change these prefixes back to exact powers of 10; however, for the "binary versions" of these prefixes, the last syllable is to be replaced with "bi" (for "binary," natch) — *kibi-*, *mebi-*, *gibi-*, *tebi-*, *pebi-*, *exbi-*, *zebi-*, *yobi-*
  - Clearly though, this hasn't gone mainstream yet, so stay tuned :)

Mass Storage

- If main memory is where computers do much of their algorithmic work, then *mass storage* is where this work is "kept" for later use or even posterity
- Mass storage is ultimately just like main memory: it is intended to house sequences of bits
- But, unlike main memory, mass storage technology:
  - Is non-volatile (i.e., loss of power ≠ loss of data)
  - Stores *much* more information
  - Costs *much* less per unit of data
  - But tends to be *much* slower

- Mass storage devices come in many technological flavors, each of which we encounter almost daily now:
  - *Magnetic systems* use magnetic media, such as coated disks or tape, to hold information; a *head* with a magnetic field and sensor uses the magnetic properties of the medium's coating to store data
  - *Optical systems* use reflective properties instead of magnetism to read/write bits; their design tends to make them more suited for reading looooooong sequences of data such as music or video rather than random access of data anywhere on the medium
  - *Flash memory*, like main memory stores information completely electronically, without the need for moving parts; however, they cost more, and are not yet suitable for the dynamic, constant read/write activity seen by main memory
- Unlike computers, people don't see information as strings of "physical" bits, but instead as "logical" units of information that make sense together
  - Examples include documents, images, applications, addresses, appointments, etc.
  - To bridge this gap, storage technology is typically "presented" as an *abstraction* (there's that word again) such as a *file system* — instead of "bits at this location," we get *files*, *folders*, and other ideas that correspond better to how we view information

From Bits to Text, Numbers, Images, Sounds, Etc.

- Speaking of abstractions…even within the level of an individual file, we still don't see bits; instead, we see words, numbers, or color, and in some cases we don't see data but *hear* it
- But we now know that all of these items are really all sequences of bits — how is it done? Two key factors:
  - You need an *encoding/decoding* scheme that allows you to unambiguously translate bits back and forth
  - And you need *standardization* so that everyone agrees to use the same scheme

Representing Text

- Text (words, numbers, punctuation) can be broken up into "atomic" units called *characters*: 'a' 'b' '3' '!' etc.
  - Thus, representing text is largely a translation from a specific bit pattern to a character and back, say 'A' is **0000 0001**, '!' is **0000 0010**, etc.
- As mentioned, this is useless unless everyone is using the same bit-to-character code, so early on, the *ASCII* standard was established, using 7 bits per character
- Eventually we needed more characters (accents, other languages, etc.), so now we have *Unicode* and related *ISO* standards that use 16 or more bits per character

Representing Numbers

- You might look at this and go "but wait — aren't binary values *already* numbers, but just in base 2?"
- This would be partially correct: given n bits, you can express 2^n distinct values, and these can stand in for the integers 0 to $2^n - 1$
- But there are some wrinkles to this scheme:
  - How do you express negative numbers?
  - How about fractions?
  - How about infinity?
- It turns out that addressing these issues results in significant differences in bit representation schemes

- For negative numbers, we need to use one bit as the *sign bit*, changing the representable range of values for n bits to $[-2^{n-1} \ldots 2^{n-1} - 1]$
  - Even with a "sign bit," the devil is in the details; such details result in schemes such as *two's complement* and *excess notation*, but explained further in the textbook
- Fractions require a bit-level equivalent to the *radix point*, or the position past which we are representing a number whose value is between zero and one
  - *Fixed point* representation resembles our decimal notation the most, while *floating point* can handle a wider range of numbers at the cost of decreased precision
- As to infinity…well, the truth is that we can't create bits out of thin air; we do get *overflow*, which occurs when calculations exceed our allocated bit space
  - But not to worry — first, when programmers know that their numbers need "special handling," they can create ways to deal with that; second, with today's 32-bit and 64-bit systems, we have enough room for all but the most demanding numeric tasks

Representing Images, Audio, and Video

- If you thought text and numbers were tricky, wait till you try turning more complicated media — images, audio, video — into bit patterns
  - Not surprisingly, the effective representation of these items in digital form came along much later than for text and numbers
- The full details are quite daunting, but the overall principle lies in *quantifying* the sensory elements (light, sound, time) of these types of information
  - For images, core values include *position* or *location* and *color*
  - For sound, we can measure the *amplitude* of a sound wave
  - Temporal or time-based media such as video and sound add a notion of *frequency* — or, how "quickly" do individual images or amplitude values have to be "displayed"

Other Twists: Compression and Error Correction

- Additional issues in data representation arise due to practical considerations, such as:
  - Storage and transmission speed aren't infinite — what if the data involved is *really* large (particularly compelling for images, audio, and video)?
  - Technology isn't perfect — how do we know that my data doesn't get corrupted?
- To address these concerns, additional "layers" of bit representation are added to the methods that we've already covered
  - *Data compression* algorithms seek ways to use *fewer* bits to store the *same* information
  - *Error correction* codes use additional bits to "double-check" whether a bit pattern may be corrupt or inconsistent