Chapter 1

Numbers

Figure 1.1: Unlike humans, computers can only understand binary

In the first module of this course, we will investigate one big question: *How does* ***everything*** *that a computer does get simplified into 0's and 1's?* It is a beautiful and well thought out process that will have us digging deep into how computers and programming languages were designed. But to understand any of that, we will need to begin at the very beginning: numbers.

We cannot begin our discussion of numbers in computers without stripping away 2 fundamental assumptions we already make about them in our everyday lives: we only have the digits 0-9 to work

with, and there is no limit to how high you can count. The first assumption leads us to a discussion of **bases** and the second, the topic of **representable ranges**.

1.1 Bases

Look at your hands. For most of you, you will see 10 fingers. For this reason alone, in our lives, we work in base 10. This means that any number can be broken up into scaled powers of ten. For example, the number 421 can be broken up into $(4 * 10^2) + (2 * 10^1) + (1 * 10^0)$. For any base b, we only have b digits to work with. Therefore, in computers, since we only have 2 digits to work with, $\{0, 1\}$, computers represent numbers in base 2.

The common bases we work with in this course are decimal (base 10), hexadecimal (base 16), octal (base 8), and binary (base 2).

1.2 Converting from Decimal

To gain a deeper understanding of numbers and their different bases, it is useful to look at numbers that we normally view in base 10, in another base. The process of converting from one base to another is quite tedious; the only way to get better and faster at it is to simply just practice. For the sake of keeping things exciting, I will illustrate this by converting from decimal to octal, though it is a worthwhile exercise to practice converting to other bases as well.

Say I wish to convert 421, which is in base 10, to base 8. As we just learned above, a number is just made up of its powers, so just like how 421 is made up of four 100's, two 10's, and one 1, to convert it into octal, we need to figure out how we can break down 421 into a 1's, b 8's, c 64's, and so on. The procedure for converting a number n in base 10 to base b is as follows:

1. Find the largest power of b that is $\leq n$, we'll call this number x
2. Keep subtracting x from n until you can't anymore without going negative
3. Keep a tally of how many times you subtracted x
4. Go back to step 1 and repeat until $n == 0$

Let's illustrate this with an example by converting 421 to its octal representation:

Firstly, we can note that the highest power of 8 that fits into 421 is 64.

Step	n	64's	8's	1's
0	421	0	0	0
1	421 - 64 = 357	1	0	0
2	357 - 64 = 293	2	0	0
3	293 - 64 = 229	3	0	0
4	229 - 64 = 165	4	0	0
5	165 - 64 = 101	5	0	0
6	101 - 64 = 37	6	0	0
7	37 - 8 = 29	6	1	0
8	29 - 8 = 21	6	2	0
9	21 - 8 = 13	6	3	0
10	13 - 8 = 5	6	4	0
11	5 - 1 = 4	6	4	1
12	4 - 1 = 4	6	4	2
13	3 - 1 = 4	6	4	3
14	2 - 1 = 4	6	4	4
15	1 - 1 = 0	6	4	5

So, 421 in base 10 becomes 645 in base 8! (This can also be denoted as 645_8). We can reverse-engineer the number to prove that this is correct:

$$(6 * 8^2) + (4 * 8^1) + (5 * 8^0) = 384 + 32 + 5 = 421$$

Cool! But this is kinda annoying and tedious, right? Well, luckily, converting to binary, which we do much more often, is more intuitive and straightforward.

1.3 Converting to Binary

We can repeat the process above for base 2 to convert to binary, and we would get the correct result (do it for practice pls). However, if you don't wish to think about powers of two and keeping tallies and what not, there is a nice handy trick that you can use:

1. Is n odd? If yes, write a 1, else 0
2. $n = n//2$
3. Repeat until $n == 0$

One thing to note is that when building up the number, we add digits onto the left side. Now, let's repeat the example above, but this time using this new trick:

Step	n	Building Binary
0	421	1
1	421 // 2 = 210	01
2	210 // 2 = 105	101
3	105 // 2 = 52	0101
4	52 // 2 = 26	00101
5	26 // 2 = 13	100101
6	13 // 2 = 6	0100101
7	6 // 2 = 3	10100101
8	3 // 2 = 1	110100101
9	1 // 2 = 0	0110100101

And so 421 in binary is 110100101_2. Let's verify this:

$$2^0 + 2^2 + 2^5 + 2^7 + 2^8 = 1 + 4 + 32 + 128 + 256 = 421$$

Great! Hopefully, this shortcut will alleviate some of tediousness of this conversion.

1.4 Number Representation

Now we can move on to the opposite procedure: given a binary number, what is the appropriate decimal number? It turns out that this is much more nuanced. (Quick mental exercise: given our discussion so far, how could we represent negative numbers in binary using only 0's and 1's? Hard right?) Instead, we will zoom out to an abstract question: what can a number, specifically bits, represent? For the purposes of this course, to summarize the fundamental concepts of number representation into one sentence: **bits are just bits**. This seems like an obvious statement, but let's see what I mean by that:

If I ask you what the number 10101_2 is in decimal, anyone who had been exposed to binary before this course would automatically say, "Oh! That's 21 obviously! $1 + 4 + 16 = 21$". This *could* be right, but it also *could not*. The underlying assumption made here was that you saw that sequence of 5 bits and translated it using the **unsigned interpretation**. However, nowhere did I mention to interpret it that way; we all jump to that conclusion because that's how we've been exposed to bits before 61C.

What am I trying to say here? **Never assume anything.** If you see a binary number, the first question you should ask yourself is "how should I interpret this?" The 6 numeric interpretations we learn in class were:

1. Unsigned
2. Sign and Magnitude
3. One's Complement
4. Two's Complement
5. Biased
6. Floating Point

Floating point is like the weird cousin, so we will revisit that later. Below I will give a quick runthrough of the first five (assuming we're working in base 2):

Interpretation	Representable Range (n bit number)	Lowest	Highest
Unsigned	$[0, 2^n - 1]$	00..00	11...11
Sign and Magnitude	$[-(2^{n-1} - 1), 2^{n-1} - 1]$	11...11	01...11
One's Complement	$[-(2^{n-1} - 1), 2^{n-1} - 1]$	10...00	01...11
Two's Complement	$[-2^{n-1}, 2^{n-1} - 1]$	10...00	01...11
Biased	$[\text{bias}, 2^n - 1 + \text{bias}]$	00...00	11...11

To convert from each representation to decimal has its own special set of rules. Let's look at how to interpret the 5 bit number 10101_2. (Following the nomenclature of working with bits, when we refer to the **most significant bit (MSB)**, we are referring to the leftmost bit, and the **least significant bit (LSB)** is the rightmost. We also 0-index from right to left, so the LSB is bit 0 and the MSB is bit $n - 1$ for an n bit number):

1. Unsigned Rule: $(2^{n-1} * b_{n-1}) + (2^{n-2} * b_{n-2}) + \cdots + (2^1 * b_1) + (2^0 * b_0)$

   $$10101_2 : (2^4 * 1) + (2^3 * 0) + (2^2 * 1) + (2^1 * 0) + (2^0 * 1) = 16 + 0 + 4 + 0 + 2 = 21$$

   .

2. Sign and Magnitude Rule: Repeat process for unsigned on bottom $n - 1$ bits, if the MSB is 1, number is negative, else positive.

   $$10101_2 : (2^3 * 0) + (2^2 * 1) + (2^1 * 0) + (2^0 * 1) = 0 + 4 + 0 + 1 = 5 \rightarrow 5 * -1 = -5$$

   since the MSB is 1.

   Note: one of the weaknesses of sign and magnitude is the existence of 2 0's: $00\ldots00$ and $10\ldots00$.

3. One's Complement: A good trick from converting from one's complement to decimal is to look at the MSB. If it is a 0, do the method for unsigned interpretation. If it is a 1, simply flip the bits and interpret it using the unsigned method to get the magnitude, but keep in mind that the sign is negative.

   Note: similar to sign and magnitude, one's complement has the weakness of 2 0's: $00\ldots00$ and $11\ldots11$.

   $$10101 \rightarrow 01010 = 2^3 + 2^1 = 10 \rightarrow -10$$

4. Two's Complement: $(-2^{n-1} * b_{n-1}) + (2^{n-2} * b_{n-2}) + \cdots + (2^1 * b_1) + (2^0 * b_0)$

   Note the negative, which *only* applies to the MSB. Therefore, if the MSB is 0, the conversion would be the same as the unsigned method.

   $$10101 : (-2^4 * 1) + (2^2 * 1) + (2^0 * 1) = -11$$

If you don't want to deal with negatives, a trick for getting the magnitude of a two's complement number is flipping the bits and adding 1 (this is very similar to the one's complement method). Don't forget that if the MSB of the original number is a 1, slap on the negative.

$$10101 \rightarrow 01010 + 1 = 01011 = 2^3 + 2^1 + 2^0 = 11 \rightarrow -11$$

5. Bias: All bias does is shift the range of the unsigned interpretation. So, take the unsigned interpretation and add the bias to get the final result: actual number = unsigned + bias.

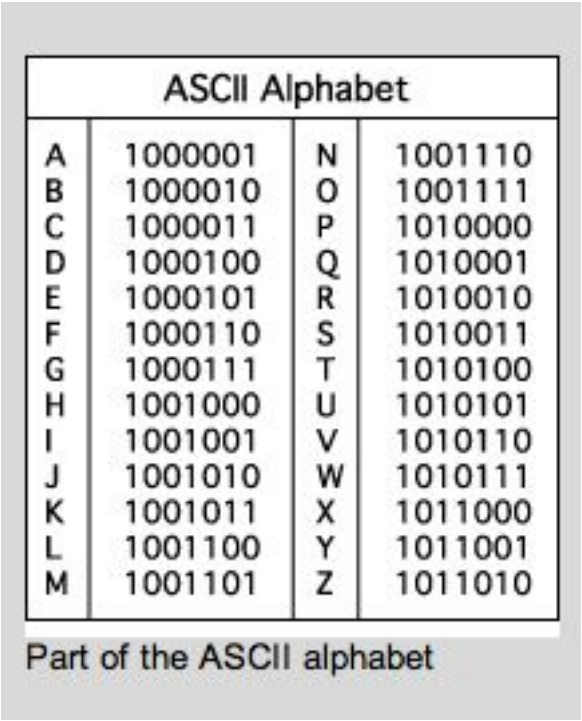
   You may notice that sometimes, we choose very specific biases. This can be understood through the lens of representable ranges. The unsigned range of 5 bits is $[0, 2^n - 1] = [0, 31]$. It is useful, in many scenarios, to shift this range so we have an almost equal number of positive and negative values, so let's shift our range by a number that will allow this: -15! If we say our bias is -15, then our new range becomes $[0 - 15, 31 - 15] = [-15, 16]$. Generally, for an n bit number, we choose the bias to be $-2^{n-1} - 1$ to even out the range of positive and negative numbers.

1.4.1 ASCII

Following the theme of bits just being bits, it is worth noting that since everything in your code has to reduce down to bits, how do we convert things that are not integers to numbers? To take it even one step further, not only does all data need to become binary, but your code itself is data that needs to be converted as well! Stay tuned, we will get to all of this, but for now, let's talk about how some people decided to resolve this problems for strings as an interesting use case.

Essentially, some people got together and decided, "Hey, let's just create a mapping from letters to numbers that everyone can agree on!" That's it. The ASCII table is just a chosen mapping from characters to numbers. There are other encodings as well, such as Unicode, but the core idea is the same: we just needed a way to convert from strings to numbers and visa versa.

So at the lowest levels, when your computer sees strings, it's actually just seeing bits, not able to know if those bits were originally a number or a letter. Yet everything works beautifully. Such is the power of *AbStRaCtIoN*.



ASCII Alphabet

A	1000001	N	1001110
B	1000010	O	1001111
C	1000011	P	1010000
D	1000100	Q	1010001
E	1000101	R	1010010
F	1000110	S	1010011
G	1000111	T	1010100
H	1001000	U	1010101
I	1001001	V	1010110
J	1001010	W	1010111
K	1001011	X	1011000
L	1001100	Y	1011001
M	1001101	Z	1011010

Part of the ASCII alphabet

Figure 1.2: A portion of the ASCII table

1.5 Bringing Everything Together

By now, I hope some questions are floating in your mind. Who handles all this interpretation? Which interpretation does my computer actually do? What is happening?

Well, the intricacies of this delve more into how processors work and are designed, which we will delve into later, but one fascinating thing to note is that 2's complement is the standard for a beautiful reason: **Regardless of the interpretation, doing simple binary addition or subtraction will give us the correct result**. Let's see what I mean:

I want to add 2 binary numbers: 10011_2 and 01001_2. This gives me 11100_2. Ok, great. Now, I will tell you that you should interpret it as unsigned numbers. Let us check to see if the math makes sense:

$$10011_2 + 01001_2 = 11100_2 \rightarrow 19 + 9 = 28$$

Thank you unsigned interpretation, very cool! Moving on to 2's complement:

$$10011_2 + 01001_2 = 11100_2 \rightarrow -13 + 9 = -4$$

Wow! This nice property does not work for other interpretations such as sign and magnitude or one's complement, and this simplicity is why all integers on modern computers are stored as 2's complement, signed numbers.

You may have seen different primitive types of numbers in your coding career: long, short, int, and the unsigned versions of all of these. Now you can understand what these mean: long is just more bits so we can store larger numbers, short is less bits, and int is the standard 32 bits in most machines. If you precede any of these types with "unsigned", the computer will interpret it via unsigned interpretation instead of 2's complement!

1.6 Bitwise Operations

Built into many modern programming languages are bitwise operations. There are 2 properties worth mentioning:

1. $0 \,\&\, b$ will "mask" the b by always making the result 0. We can use this to isolate bits:
   $111111 \& 0010000 = 0010000$
2. $0 \mid b$ will allow b to fall through by always making the result b:
   $111111 | 0010000 = 1111111$

In addition, let's briefly go over shifting:
If we shift a number left, we pad it on the right with 0's:
$11...11 << 1 = 11...10$
On the other hand, if we shift a number to the right, what we pad it with is dependent on the type of shift:

1. Arithmetic shift: pad the left with the MSB: $11...11 >> 1 = 11...11$
2. Logical shift: pad the left with 0s: $11...11 >> 1 = 01...11$

In other words, arithmetic shifting preserves the sign while logical shifting does not. For a review of the relationship between sign extending and preserving the sign of a number, run through an example, interpreting the numbers as two's complement numbers. It's an excellent exercise with a nice result.

1.7 Extra Section (not in scope)

These are dumb things I thought of while writing this to keep me entertained. Enjoy:

1. The old school rapper: Notorious B.I.T
2. The newest mumble rapper: Lil' Bit
3. "How are you so optimistic and positive all the time? You remind me of an unsigned number..."
4. Lil Pump's new catchphrase: ASCII-tet
5. Breakup phrase: "I think I can describe our current relationship in 16 bits, cuz I'mma cut it SHORT right now"

6. Pickup phrase: "Ey bb I wanna declare us with 64 bits cuz I can see us together for a LONG time"
7. I just liked this meme I found:

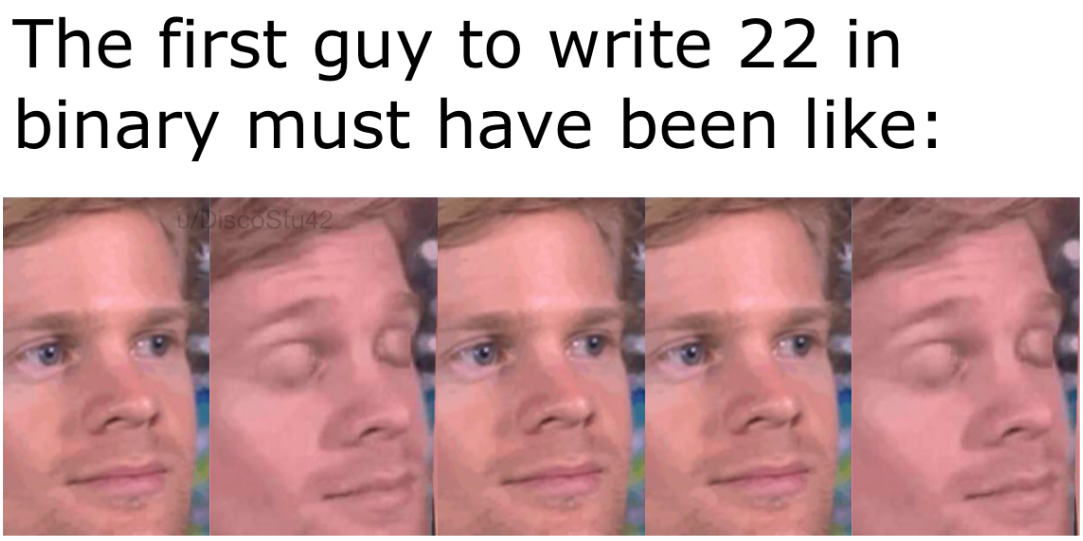


Figure 1.3: What interpretation did this meme creator assume?