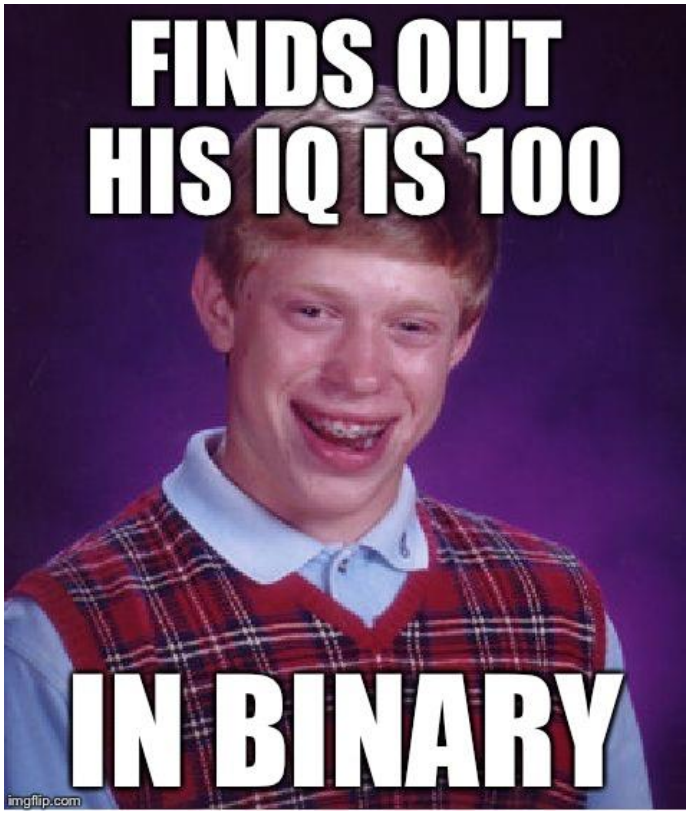
Binary and Programming

By Alexander Wang

42 Silicon Valley



FOREWORD

Welcome back for the next project. This project will be significantly shorter and is more of a history lesson unlike the previous one which had you doing some admittedly boring things but rather important things to learn and understand. This time we will be applying what you learned in the first chapter which was admittedly all theory to programming.

So, why numbers again? This time we will be discussing how a computer would see numbers and introduce some terminology that is relevant to programmers in general and especially to those who want to reverse engineering things.

This project will be building an intuition to some of the things that will be introduced when you can finally get to do some real Assembly Programming. Shall we get started?

Chapter 1 – Ugh Numbers Again:

Introduction:

Yes, yes, I know, numbers… again. Why? We just did it last project. Is the author insane for forcing it onto me again? Yes, yes he is. He's kind of a sadist that way. But don't worry! This time it's only one chapter long and introducing some terminology that will get repeated again for later on for now it's just intuition building.

Numbers for Computers:

Binary is the name of the game. Except we can't represent anywhere near the infinite number of numbers there are instead we are limited to a fairly small number (32 binary digits or 64 binary digits depending).

Let's introduce some terms immediately:

- **Binary Digit** will now mean **Bit**, which has a **value of 0 or 1**. And those two words are interchangeable.
- **Byte** is a group of **8 binary digits** or **8 bits**
- **Kilobyte** is a group of **1024 bytes** or **8192 bits** or **8192 binary digits**.
- **Gigabyte** is a group of **1024 kilobytes**
- **Most Significant bit** – is the leftmost bit (usually).
- **Least Significant bit** – is the rightmost bit (usually).

As I said in the first paragraph, we are limited to 32 binary digits or 32 bits. So, what's the *range* of numbers we can represent? Let's quickly calculate it in binary. 32 bits looks like the following (grouped into 4s):

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$$

And to see the maximum value we flip all the 0's into 1's:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

Maximum value is use our fun formula from the start:

$$(1 * 2^{31}) + (1 * 2^{30}) + \cdots + (1 * 2^{0})$$

Which is 4294967295 in decimal or in hexadecimal FFFFFFFF.

This number is a far cry from infinity but for the most part it is good enough for most modern uses.

Side Note:

While there is a reason for being limited to 32-bits or 64-bits it will not be discussed in this project and will be discussed in the project following this one. For now, this is strictly about representing various things in a computer.

What happened to Negative Numbers?

But the above is only the positive numbers. So how do negative numbers get represented? There were two developments I won't discuss them and instead give the end result and what they decided on:

1000 0000 0000 0000 0000 0000 0000 0000

See that only **1** in the last spot on the right (**most significant bit**)? It is called the **sign bit** – **says if a number is negative or not**.

Thus, the sign bit **effectively cuts the maximum value in half**. However, the halves aren't equal:

-2,147,483,648 to 2,147,483,647

As we can see the negative numbers have one extra value compared to the positive side.

To simplify it down we will assume a 3 bit digit and how all the numbers are represented using this:

Binary	Value Decimal
111	-4
110	-3
101	-2
100	-1
000	0
001	1
010	2
011	3

Similarly, if you were to have 32-bits instead of 3-bits. Now let's build some intuition.

Building Intuition for Later Things:

Using the above representation and table, we can see that if we go over our maximum positive value it'll loop into the negatives. This is a very bad thing and it is what is called **overflow – number loops around to either positive or negative as a result of an operation like adding or multiplying**. So we should be able to detect this, and thankfully the computer does check for this kind of thing.

But what about when we add two numbers together in decimal and we have to **carry** a number over – this is also something we should be checking for.

Python Data Types and How They Are Represented:

Let's look at the humble "character" which in Python has the following syntax:

'a', 'b', 'c', 'd' ….

Which means we want the literal character 'a' or whatever character we want. These are represented in binary and have a specific format.

Side Note:

Every character that you can type on a standard keyboard with a few exceptions has a numerical value associated with it. For example, all the letters of the alphabet (including lowercase) have the values of: 65-90 for uppercase, and 97-122 for lowercase.

What about a whole number? We covered that up above.

Now what about Booleans? Similar to how a binary number act it's either True or False and as the previous chapter said: we can represent True or False in binary as 1 or 0 respectively.

Tl;dr: every single data type that is used in every programming language is represented internally as binary.

Critical that you understand that statement fully.

Floating Points or Irrational Numbers:

These guys deserve their own special section. How do we represent something like ½ or one third in binary? We can't. At least not for 99% of the numbers. What we can do is approximate it.

I will not explain the reasoning behind the equation as that will require a very long discussion about the history but this is how they are represented mathematically:

$$-1^{S} * (1.0 + 0.M) * 2^{E-127}$$

S = Sign bit (negative or positive), 31st bit

M = Mantissa (23rd bit to 30th bit)

E = Exponent (0th bit to 22nd bit)

Steps to Represent a Floating Point Number Given the Fraction Number:

1. Convert the absolute value of the number to binary.
    a. Convert the integral and fractional part separately
    b. Fractional Part can be converted with multiplication (repeatedly multiplying by 2 and harvest each bit as it appears left of the decimal)
2. Append x 2^0 to the end of the binary number.
3. Normalize the number. Move the point so that it is one bit from the left. Adjust the exponent of two so that the value does not change.
4. Place the mantissa in the mantissa field of the number. Omit the leading one and fill with zeroes on the right.
5. Add the bias to the exponent from step 3.
6. Set the sign bit.

Example Problem

Convert 2.625_{10} to Floating Point Format:

Step 1:

$$Integral\ Part: 2_{10} = 10_2$$

$$Fractional\ Part: 0.625_{10}$$

0.625 * 2 = 1.25 → 1
0.25 * 2 = 0.5 → 0
0.5 * 2 = 1.0 → 1

So the number is then:

$$10.101_2$$

Step 2: Add x 2^0

$$10.101_2 * 2^0$$

Step 3: Normalize:

$$1.0101_2 * 2^1$$

Step 4: Mantissa: 0101_2

Step 5: Exponent: 1 + 127 = 128

Step 6: Sign bit is 0

So it will look like the following based off of equation mathematically above:

$$1^0 * (1 + 0.10100000 \ldots) * 2^{128-127}$$