PROGRAMMING CONCEPTS

Number Systems

The Computer Works in Binary,

or how I learned to think like a computer

Dan McElroy

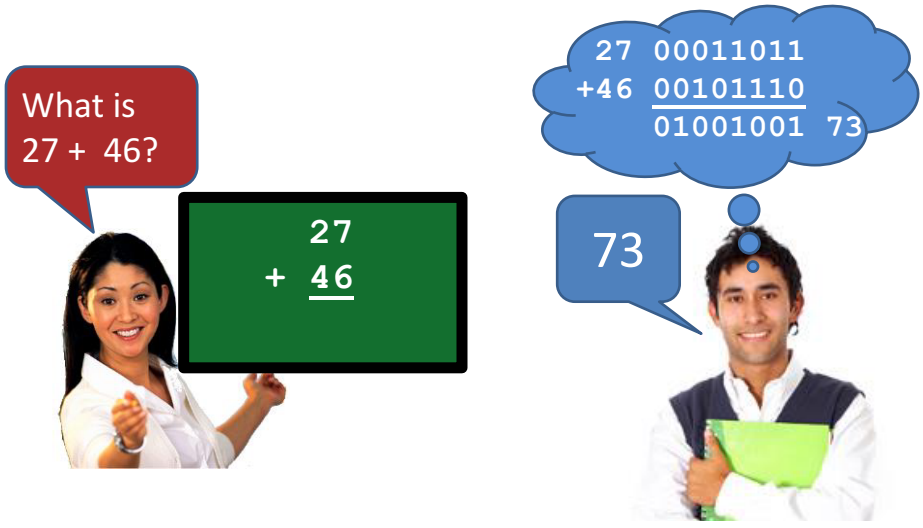
The computer's natural number system is binary not decimal.

For example, if you **input** 27 + 46, each of the numbers are converted to binary, the answer is **processed** in binary and then the result is converted back into a decimal 73 to be **output** on the screen.

```
27 →  00011011
+46 → +00101110
      =01001001 → 73
```

INPUT **PROCESS** **OUTPUT**

Suppose your brain worked in binary



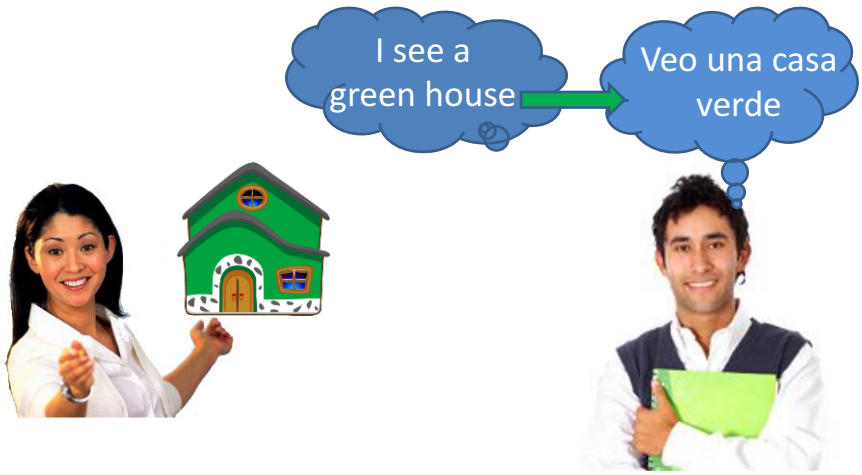
Suppose that I was learning Spanish and the teacher asks …

and I hear the words in Spanish and **input** them into my brain

I then **convert** the Spanish words into English …

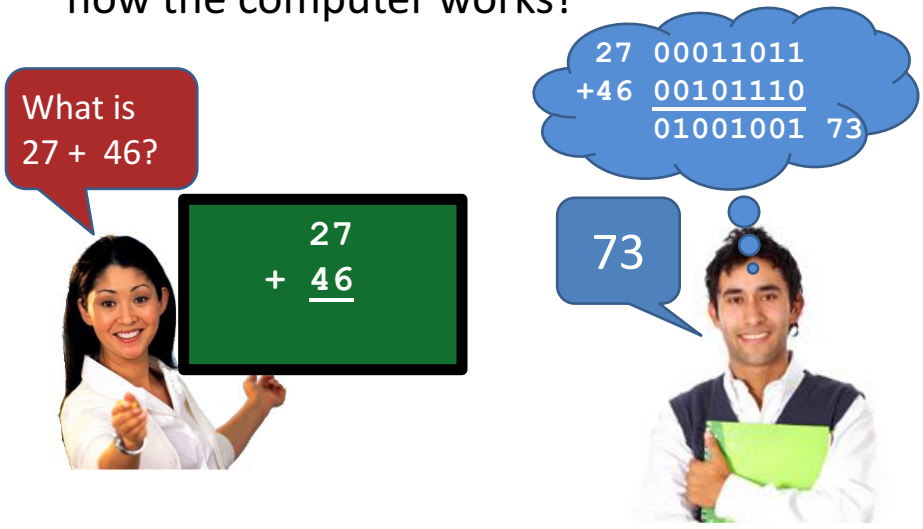
Process the answer using English …

Convert the answer back into Spanish…



and **output** the answer back in Spanish

Now does it make sense how the computer works?



For the Computer, it is Decimal as a Second Language

Assembly Language	High Level Language
You may need to provide the code to convert from decimal to binary, and back from binary to decimal.	Conversion to and from binary is already built into the input and output routines.

```
 27 ⟶  00011011
+46 ⟶ +00101110
      =01001001 ⟶ 73
```

INPUT **PROCESS** **OUTPUT**

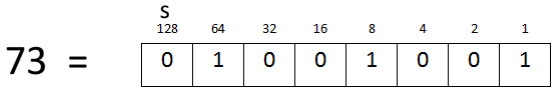
Integers

Integers are whole numbers. The values of integers can be positive, negative or zero.

Examples:

79
-85
0

Binary Representation of an Integer

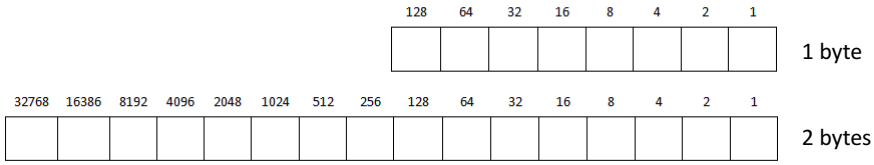


64 + 8 + 1 = 73

An unsigned decimal value from 0 to 255 can be stored in 8 bits for 256 combinations, including 0.

For signed numbers, the most significant bit is used as a **sign bit** to indicate a positive or negative number leaving one less bit for the value. Then, 8 bits can be used to store a value from -128 to +127, still 256 combinations.

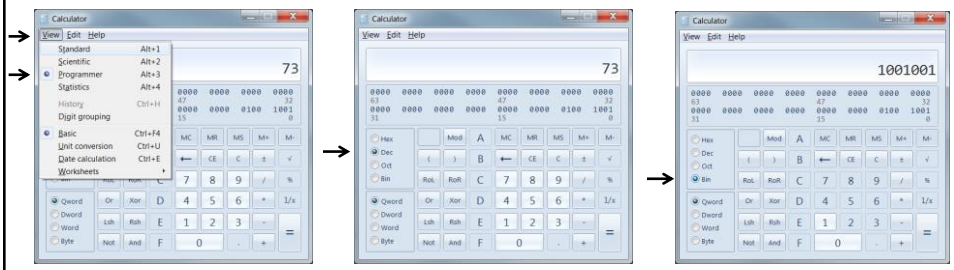
More Bits for Larger Numbers



C C++	Java	Visual Basic	Bytes	Range
byte	byte	Byte	1	0 to 255
short	short	Short	2	-32,768 to +32,767
unsigned short			2	0 to 65,536
int	int	Integer	4	-2,147,483.648 to +2,147,483,647
unsigned int			4	0 to 4,294,967,295
	long	Long	8	-9,223,372,036 854,775,808 to +9,223,372,036 854,775,807

Converting to Binary

The easiest way to convert from decimal to binary and back again is to use a calculator designed for programmers. Most graphical operating systems already include this type of calculator. Here is the calculator from Windows 7.

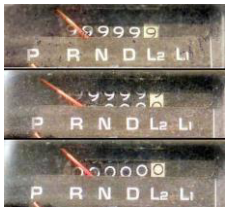


Signed vs. Unsigned

Internally, computers only use binary, 0's and 1's, also known as base 2. As a human, I learned to count in decimal, 0 through 9, also known as base 10.

Before starting a discussion on how computers implement signed numbers (+ and –), first let's see how it could be implemented in decimal without using a '+' or a '–' symbol.

Ten's Complement



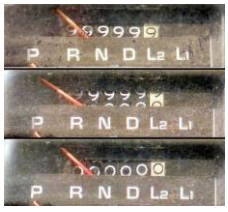
Think about an odometer in a car. As the mileage reaches 99999, one more mile causes the odometer to roll over to 00000. See the chart under the picture.

I don't know if you ever saw the movie, "Ferris Bueller's Day Off", but Ferris and his friend Cameron 'borrowed' Cameron's Dad's red Ferrari and some dude took it and ran up a lot of extra miles on the car before the kids got it back. They jacked up the car, ran it in reverse hoping to take off the miles. Unfortunately, this didn't work.

Adding Miles

99997
+ 1 = 99998
+ 1 = 99999
+ 1 = 00000
+ 1 = 00001
+ 1 = 00002
+ 1 = 00003

Ten's Complement



We could use half of the values from 00000 up to 49999 to represent positive numbers, and 99999 down to 50000 to represent negative numbers. All of the math would work for addition and subtraction. For example, +3 – 5 would give 99998 which is –2.

We would just need to convert the negative values into a human readable format using the '–' symbol before displaying the numbers on the display.

Subtracting Miles

00003 is +3
– 1 = 00002 is +2
– 1 = 00001 is +1
– 1 = 00000 is 0
– 1 = 99999 is –1
– 1 = 99998 is –2
– 1 = 99997 is –3

Two's Complement

By default, integers are 'signed' numbers. C and C++ can also declare an integer to be 'unsigned' `unsigned int x;`
The charts below use only **eight** bits, but show the decimal equivalent value as the binary value **rolls over** past `00000000`.

Unsigned integers
11111101 is 253
11111110 is 254
11111111 is 255
00000000 is 0
00000001 is 1
00000010 is 2
00000011 is 3

Signed integers
11111101 is –3
11111110 is –2
11111111 is –1
00000000 is 0
00000001 is +1
00000010 is +2
00000011 is +3

Two's Complement

For both signed and unsigned, when the least significant bit = 0 the number is even, and when it = 1, the number is odd.

For signed numbers when the most significant bit = 0 the number is positive, and when it = 1, the number is negative.

Unsigned integers
11111101 is 253
11111110 is 254
11111111 is 255
00000000 is 0
00000001 is 1
00000010 is 2
00000011 is 3

Sign bit

Signed integers
11111101 is –3
11111110 is –2
11111111 is –1
00000000 is +0
00000001 is +1
00000010 is +2
00000011 is +3

Why Is Zero a Positive Number?

Actually, zero is not a positive number. But in binary, all zeros represents the number zero. There was confusion in the early days of computers of what to do with 10000000 which had a 1 in the sign bit, but all zeros for the number. Could this be considered negative 0? One time, a customer received a bill for -0.00 but since the balance was not all zero, they kept receiving bills and threatened with collections.

Now, the number zero uses up one of the possible combinations of positive numbers (sign bit = 1), so a Short Integer has the range of -32,768 to +32,767

Groups of Bits

Although computers only work in binary, it can be somewhat confusing to write down a large number of 1's and 0's.

Octal and hexadecimal are a convenient shorthand way of representing binary numbers.

Octal numbers are used to represent groups of three bits and hexadecimal numbers are used to represent groups of four. Hexadecimal usually goes by its nickname "hex".

Octal and Hex

There are eight combinations when using three bits, and sixteen combinations when using four bits. The choice of whether to use octal or hex depends on how the bits are logically grouped on the computer or in the program.

Memory and character codes are organized in groups of eight bits (a byte). Two hex characters easily represent a byte.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Hexadecimal Numbers

For example, 101110100101001 is separated into groups of four bits 101 1101 0010 1001 starting from the least significant bit (LSB). Assume leading zeros if there are less than four bits on the left side. Binary is then converted into the hex digits 5D29 using the table. As a programmer, you should memorize the hex table.

The letters A–F are used so that only one character is required for all sixteen combinations.

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Converting

Below are samples converting between binary, octal and hex.

```
Bin  010  110  101  001  011
Oct   2    6    5    1    3

Bin 0100 0110 1101 0010 1111 1110
Hex   4    6    D    2    F    E
```

Try converting:

```
Bin  011  111  000  110  101  100  010
Oct
Bin 1110 0011 1011 1110 0000 0101
Hex
```

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

FLOATING POINT NUMBERS

Floating point numbers can contain digits past the decimal point. Floating point numbers are called REAL NUMBERS in the math department and in some computer languages

Examples:

7.0	0.0	
3.1418579	0.00000000578	positive
-85.3357	-0.00000000578	negative

Scientific Notation

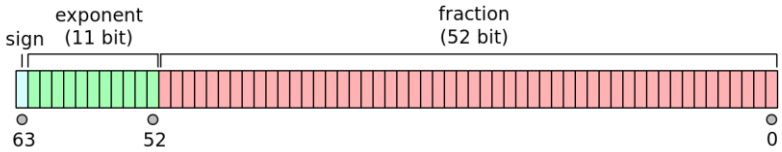
If you ever took a science class, one of the first things the teacher introduced was scientific notation. Only one digit is placed before the decimal point and then an exponent of base 10 to indicate where the decimal point should be.

The number has two parts, the mantissa (numbers), and the exponent (power of 10).

Value	Scientific Notation	Computer Display
398445	3.98445×10^5	3.98445e5
-175.6	-1.756×10^2	-1.756e2
3.0	3.0×10^0	3.0e0
0.0000000582	5.82×10^{-8}	5.82e-8

Floating Point in Binary

The IEEE (Institute of Electrical and Electronics Engineers) has defined a standard format for representing floating point numbers in binary.



Just like the scientific notation, floating point numbers are stored in two parts when they are converted to binary.

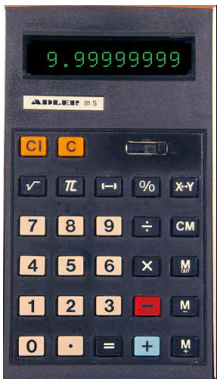
Double Precision Floating Point Numbers Are Extremely Accurate

- but not always perfect. In some cases the computer approximates the number as close as is possible within the limits of the number of bits available to store the number.

Floating point numbers are GREAT for scientific work, but occasionally the extremely small errors cause grief in the financial community.

Problems with Floating Point

Using 64 bits, the results of computing with floating point math are very accurate, but sometimes very small inaccuracies occur. For example, using a plain four-function calculator, if you take 10, divide by 3, and then multiply by 3 again, you would expect to get 10. But this is what happens on some four-function calculators is:



10 / 3 = 3.3333333
3.3333333 x 3 = 9.9999999

Some of the calculators will round the display to show a 10, but still keep 9.9999999 internally.

Problems with Floating Point

I tried the 10 / 3 x 3 and ended up with 10 on several compilers. But when I tried 29 / pi x pi, even though the display showed 29, the answer was not 29 internally. I needed to display 20 digits past the decimal to see what number the internal memory actually held. The internal value was actually 29.00000000000000400000

When I compared the original value to the result, they were not equal.

You can sometimes have a bad result if you test to see if a floating point number is equal to another number.

```
#include <iostream>          // sample C++ program to demonstrate errors
#include <iomanip>           // in floating-point computations
using namespace std;

int main(int argc, char* argv[])
{
    double value = 29;
    double result = 0;

    result = value / 3.14159265358979323846  * 3.14159265358979323846;

    if (result == value)          // see if the computed result is the same as the original value
        cout << "OK" << endl;     //  if they are equal, display OK on the screen
    else
        cout << "BAD" << endl;    //  otherwise display BAD on the screen

    cout << result << endl;       // show the result without a bunch of digits past the decimal
    cout << showpoint << setprecision(20) << fixed << result << endl;

    return 0;
}
```

ACKNOWLEDGEMENTS

IMAGES

Wikimedia® Commons

Microsoft® ClipArt