Why do we choose binary for our computers? 0 and 1 are the alphabet of computer's language. At the hardware level, everything is either 0 or 1. 0 means switch off/no signal and 1 means switch on/signal is present. It's easier to store, process and transmit with only 0 and 1.

But, our human world is decimal, that means a base-10 number system. If you want to implement a base-10 system with computers, then you need to keep track of 10 different signal levels. That will be very complex. Therefore, we go with the base-2 number system with our computers.

See below for different number systems:

Human World

Base-10 : Decimal number system (from digit 0 to 9; 10 digits; Base-10)

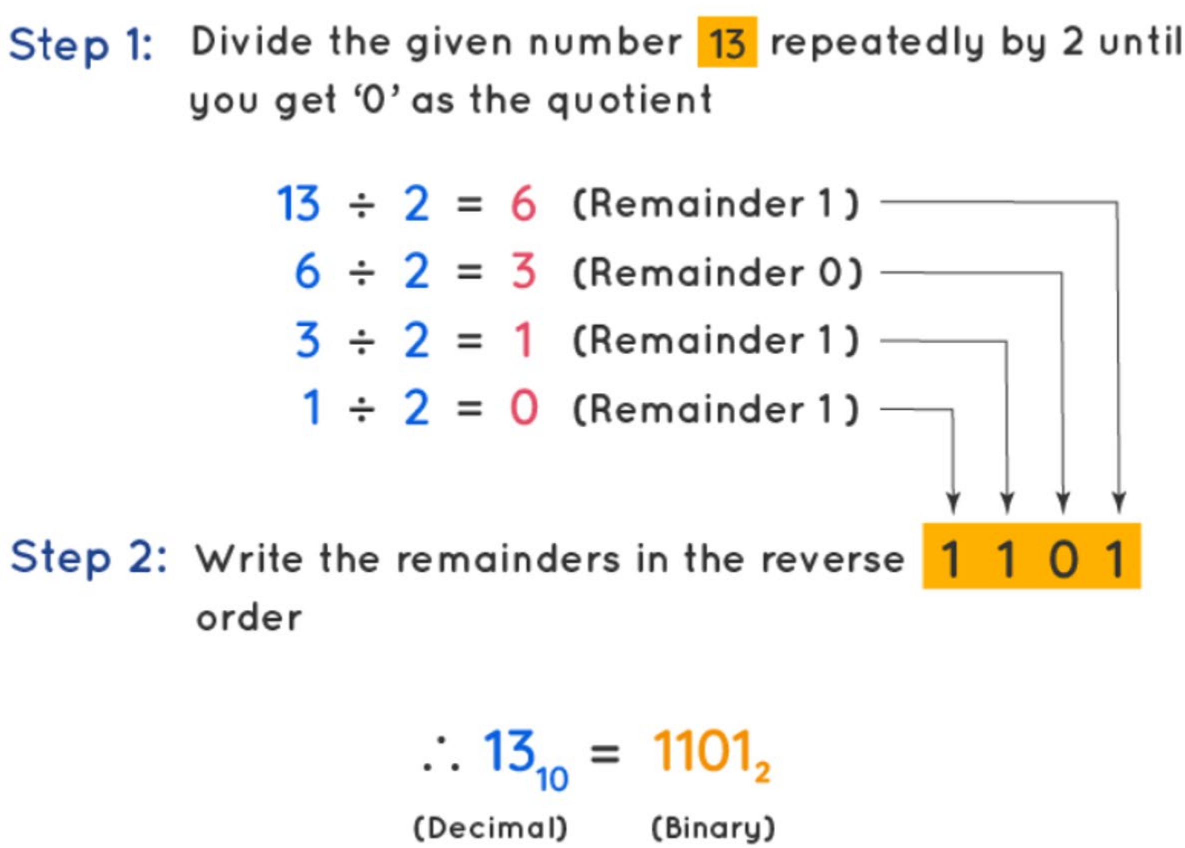
Base-16: Hexadecimal number system (from digit 0 to 9 and A to F; 16 digit; Base-16)

Computer World

Base-2: Binary number system (It's only two digits, 0 and 1; Base-2)

All our numbers in the human world are base-10 numbers. We need to convert those numbers to binary when you are in the computer world.

Practice on decimal-to-binary conversion and binary-to-decimal conversion



See this link below for binary-to-decimal conversion:

https://www.wikihow.com/Convert-from-Binary-to-Decimal

See below an example for conversion from base 8 to decimal. But, here instead of 2^0, 2^1 etc., we are doing 8^0, 8^1 ….., because the base is 8.

- **Problem: Convert $(5377)_8$ to decimal.**

- **Solution:**
  - **$(5377)_8 = 5 * 8^3 + 3 * 8^2 + 7 * 8^1 + 7 * 8^0$**
    $= 5 * 512 + 3 * 64 + 7 * 8 + 7 * 1$
    $= 2560 + 192 + 56 + 7$
    $= 2815$

In this example below, you are dividing by 16, because the base is 16.

Example: Convert $(743)_{10}$ to hexadecimal

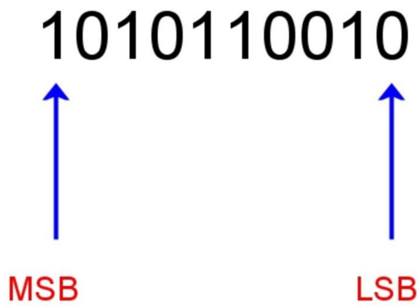
Note: Since remainders can have values from 0 to b - 1, that is, 0 to 15, we use F for 15, E for 14, D for 13, C for 12, B for 11, and A for 10

	Quotient	Remainder
16	743	7
16	46	E
16	2	2
	0	

Therefore, if we write the remainders in order from the last one obtained to the first one obtained, we have:

$(743)_{10} = (2E7)_{16}$

Always pay attention to your MSB and LSB



MSB: most significant bit/the leftmost digit
LSB: least significant bit/the rightmost digit

Another quick way to convert from HEX to binary or vice versa is to by following the chart below:

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

****Pick up some examples and practice conversion from one base to another base**

When you are representing everything at bit-level, you need to know about bit-level operations. We call them as bit-level logic. These logics are implemented with gates (made with transistors) at hardware level. Few basic gates are AND, OR, NOT, XOR etc.

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

******Pick up some examples and practice Boolean algebra like below**

- Operations applied bitwise

```
   01101001     01101001     01101001
 & 01010101   | 01010101   ^ 01010101   ~ 01010101
 ----------   ----------   ----------   ----------
   01000001     01111101     00111100     10101010
```

C language supports bit-level operations. Operations &, |, ~, ^ Available in C. See some examples below

- ~0x41 -> 0xBE
  - ~01000001_2 -> 10111110_2
- ~0x00 -> 0xFF
  - ~00000000_2 -> 11111111_2
- 0x69 & 0x55 -> 0x41
  - 01101001_2 & 01010101_2 -> 01000001_2
- 0x69 | 0x55 -> 0x7D
  - 01101001_2 | 01010101_2 -> 01111101_2

See has some logical operators too. There's difference between logical and bit-level. The logical operators are &&, ||, !

pay attention to your bit-level operators(&,|, ~) and logical operators and how they are different

Logical operators (&&, ||, !) : View 0 as "False"; Anything nonzero as "True"; Always return 0 or 1

Examples:

- !0x41 -> 0x00
- !0x00 -> 0x01
- !!0x41 -> 0x01

- 0x69 && 0x55 -> 0x01
- 0x69 || 0x55 -> 0x01

******Pick up some examples and practice the difference bit-level and logical operators**

There are two types of shift operations: left shift and right shift. Right shift is two types: logical and arithmetic. Left shift is only logical.

Left Shift: x << y

- Shift bit-vector **x** left **y** positions
- Throw away extra bits on left
- Fill with 0's on right

Right Shift: x >> y

- Shift bit-vector **x** right **y** positions
- Throw away extra bits on right
- Logical shift
- Fill with 0's on left
- Arithmetic shift
- Replicate most significant bit on left

Example:

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

****Pick up some examples and practice shift operations**

Big-endian and little-endian

In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable x of type int has address 0x100; that is, the value of the address expression &x is 0x100. Then (assuming data type int has a 32-bit representation) the 4 bytes of x would be stored in memory locations 0x100, 0x101, 0x102, and 0x103. Note that in the word 0x01234567 the high-order byte has hexadecimal value 0x01, while the low-order byte has value 0x67. Where the least significant byte comes first—is referred to as little endian. Where the most significant byte comes first—is referred to as big endian.

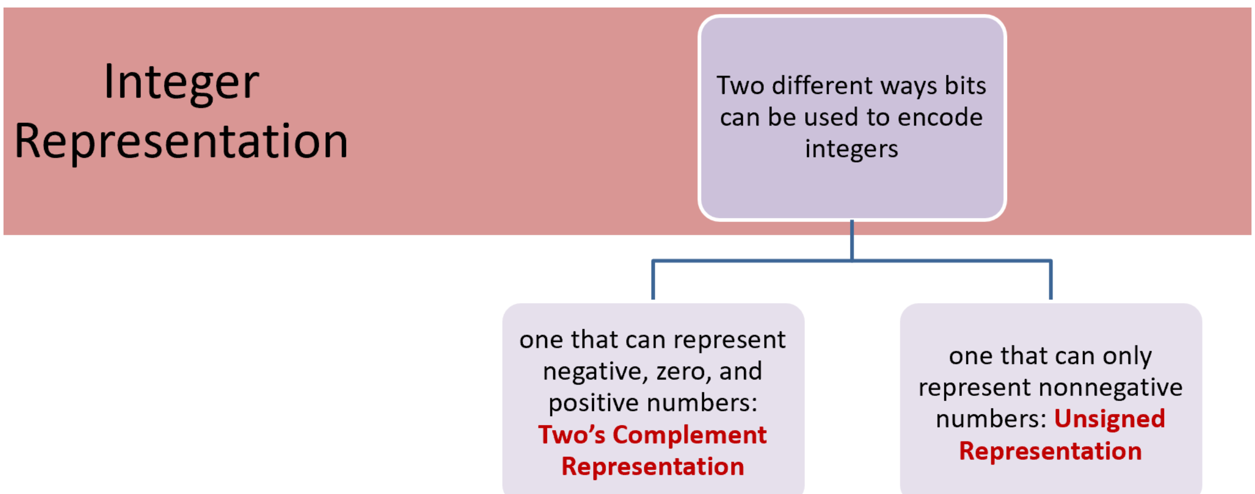
Big endian

	0x100	0x101	0x102	0x103	
. . .	01	23	45	67	. . .

Little endian

	0x100	0x101	0x102	0x103	
. . .	67	45	23	01	. . .

****Pick up some examples and practice big-endian and little-endian (practice problem)



When you have 4 bits, you can represent 2^4 numbers, that means you have 16 slots. That means you can represent 16 integers. If you go with the unsigned representation, then you can represent from number 0 to 15, that's your 16 numbers.

X	B2U(X)
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Here, B2U means Binary to Unsigned. See below for the conversion:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

$$\begin{array}{rclclcl}
B2U_4([0001]) & = & 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & 0+0+0+1 & = & 1 \\
B2U_4([0101]) & = & 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & 0+4+0+1 & = & 5 \\
B2U_4([1011]) & = & 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & 8+0+2+1 & = & 11 \\
B2U_4([1111]) & = & 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & 8+4+2+1 & = & 15
\end{array}$$

Now, if you want to represent negative numbers, then you need to do the Two's complement encoding. Therefore, out of your 16 slots, you use 8 slots for the positive integers and 8 slots for the negative integers. Here, 0 considered as a positive integer. See below for the representation:

X	**B2T(X)**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	–8
1001	–7
1010	–6
1011	–5
1100	–4
1101	–3
1110	–2
1111	–1

Here, B2T means binary to Two's Complement Conversion. See below for the conversion:

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

$$\begin{array}{rclclcl}
B2T_4([0001]) & = & -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 & = & 0+0+0+1 & = & 1 \\
B2T_4([0101]) & = & -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 & = & 0+4+0+1 & = & 5 \\
B2T_4([1011]) & = & -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 & = & -8+0+2+1 & = & -5 \\
B2T_4([1111]) & = & -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 & = & -8+4+2+1 & = & -1
\end{array}$$

Note that the MSB bit is negative, that's the sign bit. If you are representing a negative integer, then you will have your MSB as 1 in binary.

Now, if you try to see your binary values, unsigned integers values and signed integer values side-by-side, you will see that the bit-value is same, but the integer value is different (depending on what kind of representation you are going for).

If you consider only 4-bits, you will see that all the positive values are same for unsigned and signed. But, it gets different for negative value. Let's say you have a computer that only works with unsigned values, and you have a computer that only works with signed values, then when you try to convert from one computer to another (let's say from signed to unsigned), all your negative values become positive values.

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

→ T2U →

← U2T ←

Again, at the bit-level, 1101 remains 1101, but for an unsigned system 1101 means 13, but for a signed system, 1101 means -3.

There's quick way where you can go from signed to unsigned or vice versa.

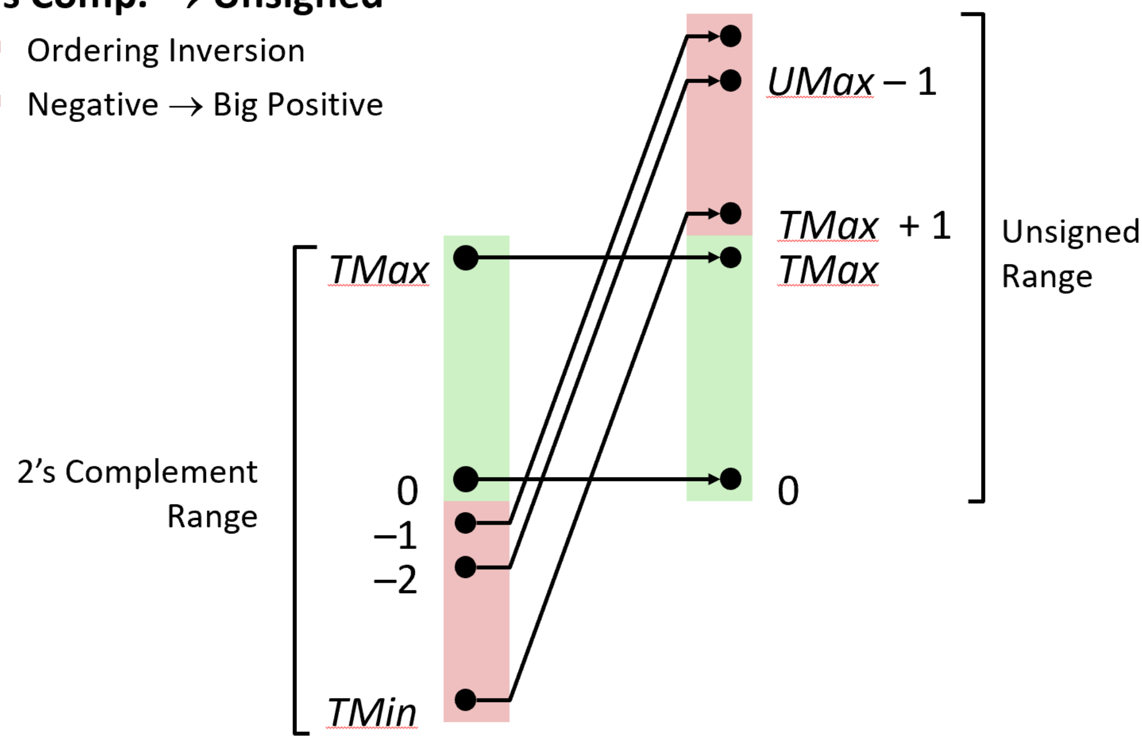
Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4	=	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

As it's 4-bits, therefore +/- 2^4 or +/- 16. If it's 5 bits, then it will be 2^5 or 32…. And continues

You also need to know about Tmax, Tmin, Umax and Umin. See below for that:

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



****For 4-bits, how many integers can be represented? What will be the range for unsigned? What will be the range for signed? Mention your Tmax, Tmin, Umax, Umin

****For 5-bits, how many integers can be represented? What will be the range for unsigned? What will be the range for signed? Mention your Tmax, Tmin, Umax, Umin

****For 8-bits, how many integers can be represented? What will be the range for unsigned? What will be the range for signed? Mention your Tmax, Tmin, Umax, Umin

****For 32-bits, how many integers can be represented? What will be the range for unsigned? What will be the range for signed? Mention your Tmax, Tmin, Umax, Umin

Casting Surprise!

If there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned. Including comparison operations <, >, ==, <=, >=

- **Bit pattern is maintained**
- **But reinterpreted**
- **Can have unexpected effects: adding or subtracting 2^w**

****** Work on practice problems (6 and 9) on casting**

Asymmetry!

- A few points are worth highlighting about these numbers. First, as observed, the two's-complement range is asymmetric: |TMin| =|TMax| + 1; that is, there is no positive counterpart to TMin. As we shall see, this leads to some peculiar properties of two's-complement arithmetic and can be the source of subtle program bugs.

This asymmetry arises because half the bit patterns (those with the sign bit set to 1) represent negative numbers, while half (those with the sign bit set to 0) represent nonnegative numbers. Since 0 is nonnegative, this means that it can represent one less positive number than negative.

Second, the maximum unsigned value is just over twice the maximum two's-complement value: UMax = 2TMax + 1. All of the bit patterns that denote negative numbers in two's- complement notation become positive values in an unsigned representation.

For 4-bits numbers, you have -8 (Tmin), but you don't have +8, because of this asymmetry. The maximum positive value you can represent is +7.

Why do we need to have unsigned representation?

For some tasks, we only need positive numbers, for example memory addresses. Why should we waste half of our slots representing negative numbers, when we don't need negative numbers? See below for more tasks:

Unsigned Representation

Unsigned values are very useful when we want to think of words as just collections of bits with no numeric interpretation.

Addresses are naturally unsigned, so systems programmers find unsigned types to be helpful.

Unsigned values are also useful when implementing mathematical packages for modular arithmetic and for multiprecision arithmetic, in which numbers are represented by arrays of words.