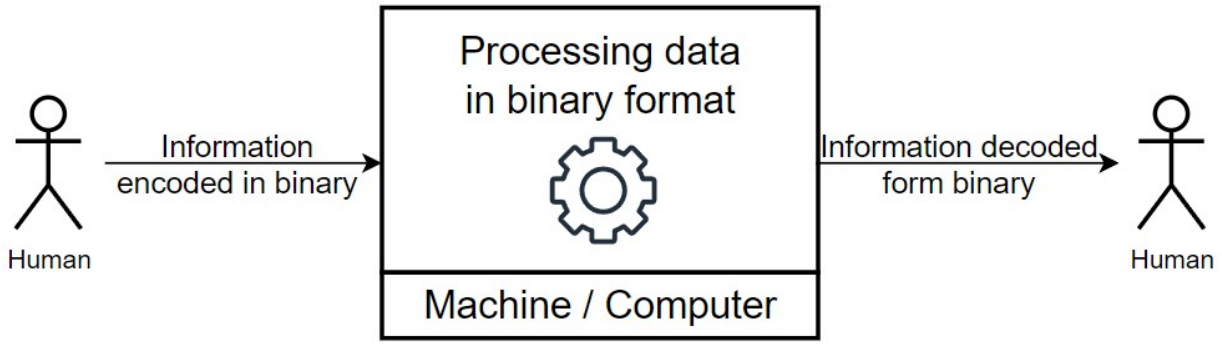
Chapter 3: Binary Encoding

1. Introduction to Data Representation

- We need a link between the abstract mathematics of number systems and the physical reality of a computer.
- A computer is fundamentally an electronic device. It operates based on electrical signals that are either present (high voltage) or absent (low voltage).
- These two distinct electrical states are the physical representation of the two binary digits: 1 ("on" or "high") and 0 ("off" or "low").

1.1. Encoding

- All forms of information that a computer handles, whether it is a simple number, a text character, an instruction, must be reduced to a simple binary format.
- Encoding is the systematic set of rules and conventions that defines how a specific piece of information is converted into a binary sequence for storage and processing.



- For example, the decimal number 83 is represented by the binary sequence 01010011 on 8 bits. The same binary sequence represent the character ‘S’.
- The same binary sequence can represent a number or a character, depending on the context or the encoding scheme being used by the computer program.
- Decoding is the opposite, it is the systematic set of rules and conventions that defines how a binary sequence is converted into a piece of information.

1.2. Basic Units of Data

This section defines the standard terminology and hierarchy used to describe data size, moving from the smallest unit upwards:

1.2.1. Bit (Binary Digit)

- The absolute smallest unit of data. A single 0 or 1.
- Can represent two states $2^1=2$.

1.2.2. Nibble

- A group of 4 bits. (Less common term, but useful for explaining Hexadecimal).
- Can represent $2^4=16$.

1.2.3. Byte

- A universally recognized group of 8 bits.
- It is the smallest addressable unit of memory in most modern computer architectures.
- This means the computer can assign a unique location (address) to every byte in memory.
- Can represent $2^8 = 256$.

1.2.4. Word

- A group of bits that is processed as a single unit by the Central Processing Unit (CPU), and normally used over all the architecture.
- The size of a word is specific to the computer's architecture (ex: a 32-bit CPU has a 32-bit word size; a 64-bit CPU has a 64-bit word size).
- Data within the CPU is in chunks the size of the computer's word.

1.3. Fixed-width restriction (Memory size)

- This concept is crucial as it imposes limits on what can be represented, leading to the necessary conventions for handling numbers outside that limit.
- Data represented inside a computer is limited by 4 characteristics: fixed-width (size), representable values, range, and the problem of Overflow.

1.3.1. Fixed-width (size or *N*)

- The purpose of fixed-wight for data is to be efficiently stored, addressed, and processed by hardware. Data must fit into fixed-size storage locations (registers, memory cells), and be transported on a fixed-size buses.
- Data types in programming (char, short, int, long) are simply names for fixed-width binary representations (8-bit, 16-bit, 32-bit, ...etc.).

1.3.2. Representable values

- For a given Fixed-Width we have a finite number of representable values.
- It represents all the possible probable combinations of 1s and 0s within N-bits.
- For instance, for N-bits we have:

N=1	N=2	N=3	N=4
0	00	000	0000
1	01	001	0001
	10	010	0010
	11	011	0011
		100	0100
		101	0101
		110	0110
		111	0111
			1000
			1001
			1010
			1011
			1100
			1101
			1110
			1111

- We get the formula **|values| = 2^N**

<u>Remark 1</u>: In mathematics, the notation |set| is called cardinality, it counts the number of elements in a set.

1.3.3. Range

- It describes the limits of an interval that encloses the represented values.
- For example, if we suppose in the table below the values represent unsigned numbers. For N=4, the smallest value is 0000 and the largest is 1111.
- Based on the table above, we can formulate an expression related to the unsigned numbers like so: **range = [0, 2^N-1]**
- The range depends on the format used to represent numbers.

1.3.4. Overflow

- Overflow is a problem that occurs when the result of an arithmetic operation exceeds the maximum representable value for the fixed number of bits.
- For example, if N=4 bits, the operation 1110 + 1001 = 10111 (14 + 9 = 23) would produce an overflow. Because the value 23 needs 5-bits to be represented, the fixed-width 4-bits is not sufficient.
- There exist some solutions to overcome this sort of problem.

2. Encoding Unsigned Integers (UI)(Positive)

- Numbers are the cornerstone for representing any other information in Computer Science.
- An Unsigned Integer is an integer value that is non-negative.
- The Unsigned Integers came to be the simplest representable form of numbers in the computer.
- The encoding of an Unsigned Integer is a direct application of the Binary Number System conversion methods.
- For example, the representation of the positive number $(18)_{10}$ with a size of 8-bits is $[000\underline{10010}]_{UI\text{-}8}$
- The value 10010 is calculated from the conversion Decimal-to-Binary. And the 3 left padded 0s are added to complete the 8-bits Fixed-Width.
- Brackets are the way to distinguish encoded numbers inside a machine from abstract mathematical numbers, like $(18)_{10}$.
- *UI-8*, means Unsigned Inter on 8-bits.
- Unsigned Integers are very common in programming languages. Ex: `unsigned int` type in C/C++ language.

Remark 2: Integers are not fractional numbers. The fractional number has a different way of representation.

2.1. Representable values

- Since hardware dictates a fixed data size (visible in the memory locations having a fixed size), the number of total values is known beforehand.
- The number of values is the total combinations possible in N-bits representation.
- This leads to the formula **$|values| = 2^N$**.
- For example, 8-bits unsigned integer size has 256 different values, $2^8 = 256$.

2.2. Range

- To determine the range is to find the smallest and the largest values in an encoding scheme.
- The smallest value in Unsigned Integers encoding is always 0. Because it is the smallest natural number.
- And the largest value is always: **$max = 2^N-1$**
- We can see the demonstration of the formula in the proof below.
- The range for Unsigned Integers on N-bits is: **$range = [0, 2^N-1]$**.

if we have:

$2^0 = 1$
$2^1 = 10$
$2^2 = 100$
$2^3 = 1000$
$2^4 = 10000$
$2^5 = 100000$

we conclude: $2^n = 1\underbrace{00\cdots\cdots00}_{n}$

then:

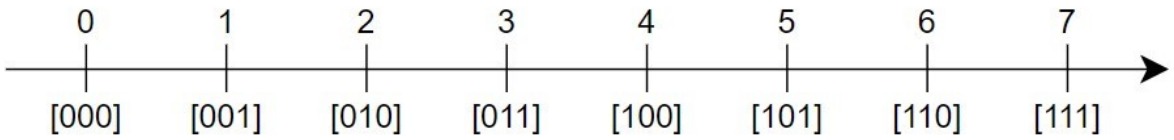
$$2^n - 1 = 1\underbrace{00\cdots\cdots00}_{n} - 1 = \underbrace{111\cdots\cdots11}_{n}$$

knowing $\underbrace{111\cdots\cdots11}_{n}$ is the maximum value in *n* bits.

this leads to: **max = 2^n-1**

2.3. Example N=3-bits

- For Unsigned Integer representation of N = 3-bits, the representable values, |values| = 2^3 = 8.
- And the range = [0,2^3-1] = [0, 7].
- The representation values are as follows:



3. Encoding Signed Integers (Positive and Negative)

- The unsigned scheme is simple and maximizes the positive range, but it cannot represent negative numbers.
- The need for computers to handle both positive and negative numbers requires an encoding scheme that dedicates one of the bits to representing the sign.
- We have 3 Signed Integers encoding schemes, Sign-Magnitude (SM), One's Complement (1C), and Two's Complement (2C).
- The 3 encoding schemes are a historic evolution, starting from Sign-Magnitude, then One's Complement and now the Two's Complement is the standard scheme of signed integers in modern machines.
- All the Signed Integer representations use the MSB as the sign bit. If it is 1, the number is negative, and it it is 0 the number is positive.
- If a number is positive, it is interpreted as an Unsigned Integer. If it is a negative number, depending on the encoding scheme, it may need some transformation to get its real value.
- Signed Integers are the default programming language numbers, like `int` or `long` in C/C++ language, for example.

3.1. Sign-Magnitude representation (SM)

- This is the most intuitive method, similar to how we typically write signed numbers in decimals by simply placing a minus sign.
- The total N-bits are split into two parts: a Sign Bit and a Magnitude.

- The Most Significant Bit (MSB), the leftmost bit, is designated as the Sign Bit. 0 indicates a positive number, and 1 indicates a negative number.
- The remaining N−1 bits encode the magnitude (absolute value) of the number in a standard unsigned binary fashion.
- For example, $(+5)_{10}$ = $[00000101]_{SM-8}$,and $(-5)_{10}$ = $[10000101]_{SM-8}$.

3.1.1. Representable values

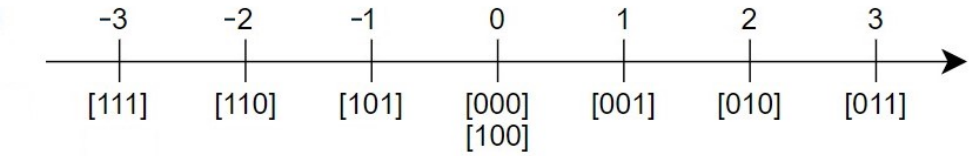
- By observation, we can see that we have two representations of zero: $(+0)_{10}$ = $[0000]_{SM-4}$, and $(-0)_{10}$ = $[1000]_{SM-4}$.
- This makes the representable values reduced by 1.
- Which makes the formula **$|values| = 2^N-1$**.

3.1.2. Range

- We saw from Unsigned Integer that the maximum value is 2^N-1.
- We know also that magnitude in the Sign-Magnitude scheme of positive values is on N-1 bits, which makes the maximum of positive numbers: **$max_{positive}=2^{N-1}-1$**.
- And the same for negative numbers, the magnitude is on N-1 bits, which makes the maximum of negative numbers: **$max_{negative}=-(2^{N-1}-1)$**.
- Those 2 limits enclose the range for Sign-Magnitude representation: **range = $[-(2^{N-1}-1), +(2^{N-1}-1)]$**.

3.1.3. Example N=3-bits

- For Sign-Magnitude representation of N = 3-bits, the representable values, $|values| = 2^3-1 = 7$.
- And the range = $[-(2^{3-1}-1),+(2^{3-1}-1)] = [-3, +3]$.
- The representation values are as follows:



3.2. One's Complement representation (1C)

- Historically, Sign-Magnitude was very readable to humans, but relatively difficult to implement in hardware (in ALU).
- One's Complement was an attempt to simplify the hardware by performing “subtraction by addition” (ex: 5-2 = 5+(-2)). Using only a single, simple addition operation. But require a second addition in case of overflow.
- In One's Complement scheme, positive numbers are represented the same as in the unsigned scheme.
- The representation of a negative number is by inverting all the bits (change all 0s to 1s, and all 1s to 0s) of its positive counterpart.
- For example, $(+5)_{10}$ = $[00000101]_{1C-8}$,and $(-5)_{10}$ = $[11111010]_{1C-8}$.

3.2.1. Representable values

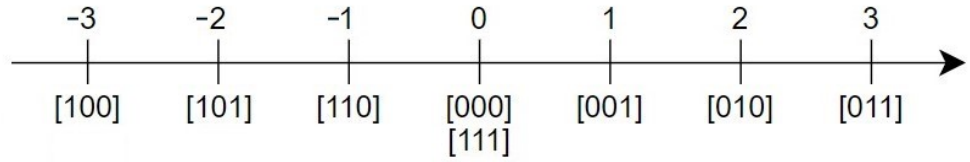
- Similar to Sign-Magnitude, we can see that we have two representations of zero: $(+0)_{10} = [0000]_{1C-4}$, and $(-0)_{10} = [1111]_{1C-4}$.
- This also makes the representable values reduce by 1.
- Which makes the formula **$|values| = 2^N-1$**.

3.2.2. Range

- Similar to Sign-Magnitude, positive and negative numbers are symmetric.
- And also similar to Sign-Magnitude, excluding the sign bit. N-1 is the number of bits to represent all positive numbers.
- Which makes the maximum of positive numbers: **$max_{positive} = 2^{N-1}-1$**.
- And the same for negative numbers, they are symmetric to positive, which makes the maximum of negative numbers: **$max_{negative} = -(2^{N-1}-1)$**.
- Those 2 limits enclose the range for One's Complement representation:
- **range = $[-(2^{N-1}-1), +(2^{N-1}-1)]$**.

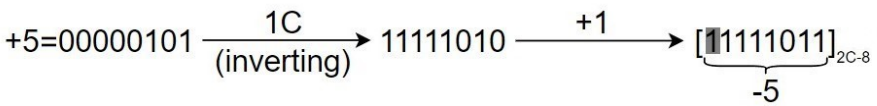
3.2.3. Example N=3-bits

- For One's Complement representation of N = 3-bits the representable values, $|values| = 2^3-1 = 7$.
- And the range = $[-(2^{3-1}-1),+(2^{3-1}-1)] = [-3, +3]$.
- The representation values are as follows:



3.3. Two's Complement representation (2C)(The Standard)

- Two's Complement is the dominant method used in practically all CPUs and ALUs today.
- Because it solves the problems of two zeros and complex arithmetic, simplifies hardware design significantly.
- Like the 2 previous schemes, positive numbers are represented the same as in the unsigned scheme.
- But for the negative numbers, it is more complex to calculate. The method we will use requires 2 steps.
- To convert a positive number to a Two's Complement negative representation. First step is to invert the number, it's just One's Complement transformation, then adding 1.
- For example $(-5)_{10} = [11111011]_{2C-8}$



- The opposite is to convert a negative Two's Complement number to its counterpart positive number. It is exactly the same steps, the first step is to invert the 2C number, using One's Complement, then adding 1.
- For example $[11111011]_{2C\text{-}8} = (-5)_{10}$

$$[\underbrace{11111011}_{-5}]_{2C\text{-}8} \xrightarrow[\text{(inverting)}]{1C} 00000100 \xrightarrow{+1} 00000101 = +5$$

Remark 3: Many methods exist to convert to and from Two's Complement numbers, in this lecture we only use the 2-step method: 1C followed by adding 1. In the 2 directions.

3.3.1. Representable values

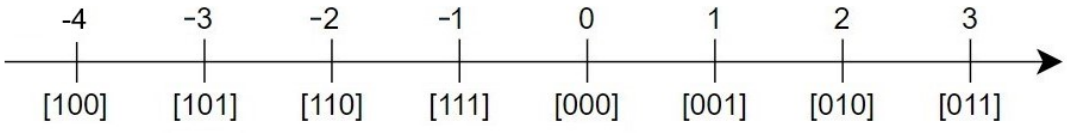
- We can see the conversion of the value $(-0)_{10}$ using Two's Complement, that it is the same as $(+0)_{10}$. Ex: $+0 = 0000 \xrightarrow{1C} 1111 \xrightarrow{+1} [0000]_{2C\text{-}4} = -0$
- That means there is only one representation of zero, which is a huge advantage for the scheme.
- Without a double zero the formula becomes **|values| = 2^N**.

3.3.2. Range

- Similar to Sign-Magnitude and One's Complement positive numbers. N-1 is the number of bits to represent all positive numbers.
- That makes the maximum of positive numbers: **$max_{positive} = 2^{N-1}-1$**.
- Counting the negative numbers in Two's Complement is different.
- We suppose the zero value is without a sign. We get one more negative value compared to positives. It is the counterpart of the value (+0) that should be (-0).
- Normally is a value in the form of $[100\cdots0]_{2C\text{-}N}$, knowing that negative values, similar to One's Complement are in an descending order. (-1) has the highest representation $[111\cdots1]_{2C\text{-}N}$. and the lowest is $[100\cdots0]_{2C\text{-}N}$.
- While converting $[100\cdots0]_{2C\text{-}N}$ in Two's Complement, we get the value:
  $[100\cdots0]_{2C\text{-}N} \xrightarrow{1C} 011\cdots1 \xrightarrow{+1} 100\cdots0 = 2^{N-1}$.(we have N-1 zeros)
- This additional value changes the maximum negative to: **$max_{negative} = -(2^{N-1})$**
- Which indicates clearly that the positive and negative are not symmetric like in One's Complement and Sign-Magnitude.
- **range = $[-(2^{N-1}), +(2^{N-1}-1)]$**.

3.3.3. Example N=3-bits

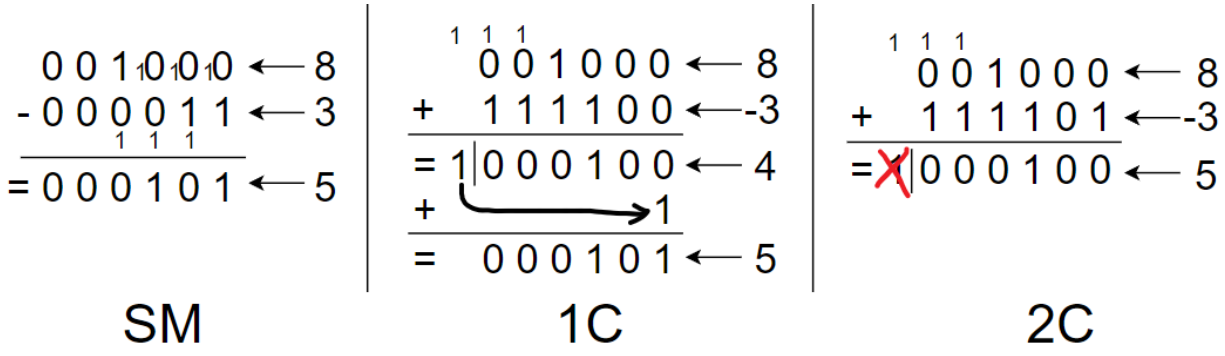
- For One's Complement representation of N = 3-bits, the representable values, $|values| = 2^3 = 8$.
- And the range = $[-(2^{3-1}),+(2^{3-1}-1)] = [-4, +3]$.
- The representation values are as follows:



3.4. Subtraction

In this section we will compare the 3 signed representations regarding arithmetic operations, especially addition and subtraction:

- Let us see how subtraction is performed in differently in the 3 representations with the example 8 - 3.
- For the Sign-Magnitude in the diagram below (SM) the subtraction is done directly ($[001000]_{SM-6}$ - $[100011]_{SM-6}$), this means the adder and the subtracter are 2 separate hardware.
- In the Sign-Magnitude the hardware needs to check the signs of the 2 operands to decide which is the appropriate operation. Which makes the hardware complex.
- In the One's Complement (1C) operation, the subtraction is done indirectly by using addition (8 + (-3)), but we need to add +1 if there is overflow.
- The subtraction $[001000]_{1C-6}$ + $[111100]_{1C-6}$ (8 + (-3)) needs 2 additions, but no specific hardware for subtraction. Which is simpler than Sign-Magnitude.*
- Like One's Complement, Two's Complement uses the "subtraction by addition" technique, and the 8 - 3 operation is performed with 8 + (-3) ($[001000]_{2C-6}$ + $[111101]_{2C-6}$).
- We can see in the diagram below that in the case of Two's Complement, only one addition is necessary to do subtraction. The overflow bit is ignored.



- We can clearly see that the Two's Complement is the simplest way to perform addition and subtraction, that needs only one adder.
- This simplification also applies to multiplication and division, knowing that those 2 require the use of addition and subtraction.

Remark 4: In this course, students only need to know subtraction in Two's Complement. The other operations have just been shown for the purpose of the demonstration.

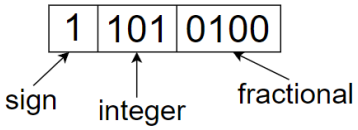
Remark 5: We didn't mention the Unsigned Integer in 3 operations shown above. But the addition done in Two's Complement remains valid for Unsigned Integers too. Only the interpretation is different. For example, $[1100]_{UI}$ + $[0010]_{UI}$ = $[1110]_{UI}$ is interpreted as 12 + 2 = 14. But in 2C the same $[1100]_{2C}$ + $[0010]_{2C}$ = $[1110]_{2C}$, is interpreted as -4 + 2 = -2.

4. Encoding Real/Fractional Numbers

Like negative numbers, the machine needs a way to represent functional numbers. The machine only recognizes binary format, and we need a way to represent the fractional dot. Two methods were developed for this job: Fixed-Point and Floating-Point.

4.1. Fixed-Point Representation

- It is very easy to represent fractional numbers using Fixed-Point, the N-bits are divided into 3 parts.
- The binary value of the number is distributed over 3 parts.
- The sign bit for the sign of the fractional number.
- The integer part, a group of bits to represent the integer part of the fractional number.
- And the fractional part, a group of bits to represent the fractional part of a number.
- For example, in 8-bits Fixed-Point representation the MSB is used for the sing, 3-bits are used for the integer part, and 4-bits are used for the fractional part.
- The value $(-5.25)_{10} = (-101.01)_2$ is represented as follows: $[1101|0100]_{Fix4.4}$



- The problem with Fixed-Point is a fundamental limitation in range and precision.
- It is not adequate for applications requiring both a vast range and fine precision.
- Nevertheless, it is still in use in some specific simple applications, but definitely not the standard scheme to represent real numbers.

4.2. Floating-Point Representation

- Many domains and scientific applications like cosmology and subatomic physics need big numbers or very fine precision numbers.
- The solution is to use a representation analog to Scientific Notation (Ex: 1.23×10^{27}).
- By definition, the Scientific Notation imposes one digit in the integer part, and the remain part is in the fraction. The decimal point is moved accordingly using the exponent
- The decimal point can be moved, this is why it is called floating point.
- With the Scientific Notation it is possible to get very large numbers or very small numbers, but not both at the same time, unfortunately.

4.2.1. Concept

- The way to represent Scientific Notation is:

$$\text{Fractional number} = \text{Sign} \times \text{Significand} \times \text{Base}^{\text{Exponent}}$$

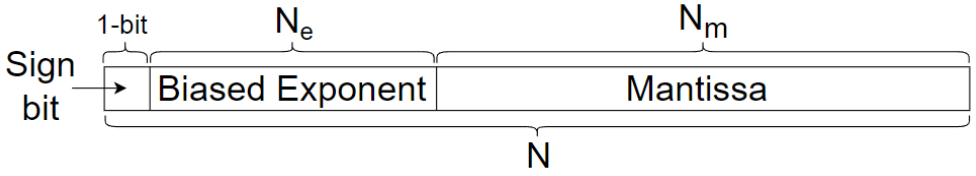
- In binary systems, this becomes:

$$\text{Floating Piont (FP)} = (-1)^{\text{Sign}} \times \text{Mantissa} \times 2^{\text{Exponent}}$$

- **Sign**: is Sign bit (0 for positive, 1 for negative).
- **Mantissa**: is the Significand, is the number.
- **Exponent**: the power of 2 that the mantissa is multiplied by.
- Example, the number 101.11 is transformed to 1.0111 x 2^2
- Moving the fractional point left is an increment of the Exponent, and moving right is a decrement.

4.2.2. IEEE 754 Standard (The Industry Standard)

- To ensure that real numbers are represented identically across all computers, the normalization Institute IEEE established the IEEE-754 Standard for Floating-Point encoding scheme.
- It is different from the simplified representation shown above.
- Two different versions well-known for the standard related to the Size (N) in bit of the real number are: 32-bits (Simple Precision) and 64-bits (Double Precision).
- The 32-bits Floating-Point in C/C++ is the type `float`, and 64-bits is `double`.
- Every IEEE-754 Floating-Point number is divided into 3 fixed-size fields:



- The size of the different fields is as in the following table:

Format	N	N_e	N_m
32-bit Floating-Point	32-bits	8-bits	23-bits
64-bit Floating-Point	64-bits	11-bits	52-bits

- The following formula gives the way to decode the value of a Floating-Point:

$$FP = (-1)^S \times (1.M) \times 2^{E_{Real}}$$

Where:

$$E_{real} = E_{biased} - Bias$$

- S: is the sign bit
- M: is the Mantissa
- E_{real}: is the Real Exponent
- E_{biased}: is the Biased Exponent
- Bias: is a delta to shift the range of the exponent including negative values.
- It is also possible to use a global version of the formula:

$$FP = (-1)^S \times (1.M) \times 2^{E_{Biased} - Bias}$$

- This formula is used to do the decoding operation, the encoding operation is done in reverse.
- 3 elements of this formula need more depth explanation, and the logic behind the way this formula is used to encode Floating-Point numbers.
- The first element is how the Sign bit is used. The calculation of $(-1)^S$ produces -1 if S=1 and +1 if S=0, this aligns perfectly with the interpretation of a sign bit.

- The second element, the Mantissa in the IEEE-754 scheme always represents the fractional part of the number.
- Knowing that only the first digit (different from zero) is assigned to the integer part in a Scientific Notation, and 1 is the only candidate in a binary number.
- That makes the first 1 in the representation implicit and doesn't need to be stored, saving this way a precious bit of information.
- This implicit leading 1 was called *normalization* of the number by IEEE, and guarantees the inequality $1 \leq 1.M < 2$.
- The third element is the exponent. On N_e-bits, it is required to represent positive and negative numbers. The IEEE didn't choose the usual representations, like SM, 1C, or 2C, they were judged complex. And chose a *biased* representation.
- The reason to choose the biased method is to make the comparison between real numbers faster by the hardware.
- The concept of bias is to delta shift (called bias) by half to the negative, the interval of Unsigned Integers in N_e-bits. Using the formula:

```
real_number = shifted_number - delta
```

- The demonstration below shows how it is done on N_e=4-bits:

Let pretend N_e=4-bits,
we have:

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

If we divide the values into 2 equal halves and take the value just below the middle (7=0111).

Let pretend (0111) is the new shifted 0, below it are negative numbers and above it are positive numbers:

8 = 1111
7 = 1110
6 = 1101
5 = 1100
4 = 1011
3 = 1010
2 = 1001
1 = 1000
0 = 0111
-1 = 0110
-2 = 0101
-3 = 0100
-4 = 0011
-5 = 0010
-6 = 0001
-7 = 0000

The formula to do the transformation is:

```
new_number = old_number - delta
```

Delta (Bias) is the new 0. and always has the form $(011\cdots\cdots11)_2$ on N_e-bits.

- A generalization of the Bias from the form $(011\cdots\cdots11)_2$ on N_e-bits can be deduced, knowing that the form contains N_e-1 one. The formula is: **Bias = 2^{Ne-1}-1**
- We can find the Biases for the 2 IEEE representations:

Format	N_e	Bias
32-bit Floating-Point	8-bits	2^{8-1}-1 = **127**
32-bit Floating-Point	11-bits	2^{8-1}-1 = **1023**

4.2.3. IEEE-754 32-bits Single-Precision example

Let's decode the 32-bit binary number: 0|10000001|00010000000000000000000

1. Extract Fields:

- S (Sign bit) = 0 → Positive number (+)
- E_{biased} (Biased Exponent) = 10000001 → $E_b = (129)_{10}$
- M (Mantissa) = 00010000000000000000000 → M = 0001

2. Calculate Real Exponent:

- Bias = $(127)_{10}$
- $E_{real} = E_{biased}$ - Bias = 129 - 127 = 2 → $E_r = 2$

3. Determine Significand:

- Since the number is normalized, the significand is 1.M
- Significand = $(1.0001)_2$

4. Calculate Final Value:

- The formula: $FP = (-1)^S \times 1.M \times 2^{Er}$

  $FP = +1.0001 \times 2^2 = (100.01)_2$
- Using FP.FPF: $(100.01)_2 = (1x2^2 + 1x2^{-2}) =$ **$(4.25)_{10}$**

Let do now use the opposite example, encoding the decimal number 21.125 into the 32-bit IEEE 754 Single-Precision format.

1. Convert the Number to Binary:

- First, convert the integer part $(21)_{10}$ to binary using SED:

Division	Remainder
21 ÷ 2 = 10	1
10 ÷ 2 = 5	0
5 ÷ 2 = 2	1
2 ÷ 2 = 1	0
1 ÷ 2 = 0	1

$(12)_{10} = (10101)_2$

- Next, convert the fractional part $(0.125)_{10}$ to a binary fraction using SM:

Multiplication	Integer part
0.125 x 2 = 0.25	0
0.25 x 2 = 0.5	0
0.5 x 2 = 1.0	1

$(0.125)_{10} = (0.001)_2$

- Combine the integer and fractional parts: **$(21.125)_{10} = (10101.001)_2$**

2. Determine the Sign bit (S):

- The number 21.125 is positive → **S=0**

3. Normalize the Binary Number:

- The binary number must be converted to the format $1.M \times 2^{Ereal}$.

- We move the binary point to the left until only one '1' remains before it:

  $(10101.001)_2 = 1.0101001 \times 2^4$

- From normalization, we can determine the Real Exponent: **$E_r = 4$**
- And we can also determine the Mantissa: **M = 0101001**

4. Calculate the Biased Exponent (E_b):

- We use the Exponent Bias for Single-Precision, which is **Bias = 127**.
- We use the Exponent formula:

  $E_r = E_b$ - Bias $\rightarrow E_b = E_r$ + Bias

  $E_b = 4 + 127 = (131)_{10}$
- Now, we convert E_b to its 8-bit binary representation using SED:

Division	Remainder
131 ÷ 2 = 65	1
65 ÷ 2 = 32	1
32 ÷ 2 = 16	0
16 ÷ 2 = 8	0
8 ÷ 2 = 4	0
4 ÷ 2 = 2	0
2 ÷ 2 = 1	0
1 ÷ 2 = 0	1

  $(131)_{10} = (10000011)_2$
- We have **E_b = [10000011]**

5. Assemble the Final 32-bit Representation:

- We have the 3 fields in a 32-bits FP representation organized as follows: Sing bit (1-bit) | Biased Exponent (8-bits) | Mantissa (23-bits)
- We can form the final binary representation (M is right padded by 0s): **[0|100 0001 1|010 1001 <u>0000 0000 0000 0000</u>]$_{FP-32}$**
- Still this format is hardly readable by humans, we need to convert it to hexadecimal: **[41A90000]$_{FP-32}$**

<u>Remark 5</u>: FP-32 means 32-bits Floating-Point Single-Precision real number.

4.2.3. Single-Precision and Double-Precision range and precision

Ranges and precision for 32-bits and 64-bits Floating-Point in decimals are listed as follows:

Format	Approximate Decimal Range	Approximate Decimal Precision
32-bit Floating-Point	$\pm 10^{-38}$ to $\pm 10^{+38}$	7 digits
64-bit Floating-Point	$\pm 10^{-308}$ to $\pm 10^{+308}$	15-16 digits

- The decimal range and precision should be seen as related to the Scientific Notation.
- Approximate Decimal Range is the range of the exponent in Scientific Notation representation. And Approximate Decimal Precision is the number of digits supported in the Significand, or the number of digits in the number.

4.2.4. Special Values (Reserved for Errors and Edge Cases)

The IEEE 754 standard reserves the minimum and maximum stored exponent values for specific cases to handle exceptions in a standard way, the table below lists these special cases:

Biased Exponent (E_b)	Mantissa (M)	Represents
All 0s	All 0s	**zero (±0):** The only way to represent an exact zero.
All 0s	Non-Zero	**Denormalized Numbers:** Used for numbers smaller than the normal minimum.
All 1s	All 0s	**Infinity (±∞) :** Result of division by zero
All 1s	Non-Zero	**Not-a-Number (NaN):** Result of undefined operations (ex: 0/0,∞−∞).

- By observation of the table, the rule is simple, if E_b is all 0s or all 1s, the norm gives a special representation, and those cases should not be decoded as usual.
- This also gives a limit for the E_b, its max = 111⋯10, and its min = 000⋯01.

4.2.5. Denormalized Numbers (Values near zero)

- Denormalized numbers are used to represent numbers that are too small to be normalized, but not exactly zero.
- They are identified by reserving a specific pattern in the Bias Exponent field. E_b = 0, and the Mantissa is different from 0.
- The denormalized formula is slightly different from the normalized formula:

$$PF = (-1)^S \times 0.M \times 2^{E_{min}}$$

where

$$E_{min} = \text{minimum\_biased\_exponent} - Bias = 1 - Bias$$

- We can see that the Significand = 0.M, because we are dealing with smaller values than the Normalized representation can provide.
- And we have the fixed value of E_{min}, that represents the smallest value represented by E_b=1, knowing that E_b=0 is the special value.
- It is possible to apply the Exponent formula to get E_{min}. E_{min} = E_b-Bias = 1-Bias.
- E_{min} represent the real minimum Exponent.
- In Single-Precision Floating-Point, E_{min} = 1-127 = -126.
- We can apply an example on a Simple-Precision Denormalized number like so:
  [0|00000000|11000000000000000000000]$_{FP-32}$

  The formula is: $(-1)^0 \times 0.11 \times 2^{-126} = 0.75 \times 1.17549 \times 10^{-38}$
  $= \mathbf{8.8162 \times 10^{-39}}$

Remark 6: IEEE-754 defines other less-known fractional number schemes, like Half-Precision, a 16-bits Floating-Point (N_e=5, and N_m=10), or Quadruple-Precision with 128-bits (N_e=15, and N_m=112). The same methods and rules apply like for the Single and Double-Precision.

5. Encoding Alphanumeric Data (Text and Characters)

- While binary encoding schemes like Two's Complement and IEEE-754 are essential for representing numbers, computers must also process text.
- *Character Encoding* is the systematic method used to translate these non-numeric symbols into their binary representations.
- A computer only understands the binary digits 0 and 1. For a machine to store or transmit text data, each character must be assigned a unique numerical code.
- Character encoding is a mapping (a look-up table) that assigns a specific, fixed-length binary sequence (usually one or more bytes) to every character.
- That includes letters: 'A', 'B', 'a', 'b'…etc. Digits as characters: '0', '1', '2',…etc. Punctuation: ',', '!', '?', …etc. Control codes: non-printable codes used for formatting or device control, such as CR (Carriage Return) or LF (Line Feed).
- A universal standard is necessary for data portability. That prevents the same character getting from many representations. This ensures the data is compatible over different hardware machines and Operating Systems.

Remark 7: It is important to make a clear distinction between a digit like 2 and its text character ’2’. The apostrophe generally makes this distinction.

5.1. ASCII (American Standard Code for Information Interchange)

- ASCII is the foundational character encoding standard, developed in the 60s, and is still the basis for much of modern text processing.
- Historically, the first version was 7-bit ASCII, that uses 7 bits to represent each character.
- With 7 bits, it can represent 2^7=128 unique characters coded from 0 to 127.
- This set includes the uppercase and lowercase English alphabet, the digits 0-9, common punctuation, and 33 non-printable control codes.
- For example: The code for the letter 'A' is $(65)_{10}$, which is $(1000001)_2$.
- An extended ASCII 8-bits version was introduced when computers began operating on bytes (8-bits) as the minimum unit.
- The 8^{th} bit in ASCII was initially unused, when added the set expanded to 2^8=256 characters adding codes from 128 to 255.
- This space was used to add common Western European characters (like letters with diacritics: é, à, ü) and basic graphic symbols.
- Unfortunately, these extended versions were often non-standard and varied by country or application, leading to code page issues.
- We have below an example of an 8-bits extended ASCII table:

Hex		00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80	128																
90	144																
A0	160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
B0	176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

5.2. Unicode (Global Standards)

- ASCII is limited to 256 characters, which is enough for English but cannot accommodate non-Latin scripts like Chinese, Japanese, Korean, Arabic…etc.
- There are also thousands of historical or mathematical symbols, and non-official languages, that need a way to be represented in computers.
- This led to the creation of the universal character set, called *Unicode*.
- Unicode is not an encoding scheme itself like ASCII, but a vast, universal mapping that assigns a unique, platform-independent number, called a *Code Point*, to every character.
- A Code Point is simply an identifier or fixed number, written in the format U+XXXX (in hexadecimal), assigned for every character in the catalog.
- For example, Latin capital letter A is U+0041. Or the Greek letter alpha (α) U+03B1.
- That includes all the well-known human languages, covering the total of 149000 characters.
- The main misconception is that Unicode is not an encoding scheme. It doesn't fix how those Code Points should be encoded inside the computer.
- But this role is affected by a different group of related encoding schemes, named UTF-8, UTF-16, and UTF-32.

5.3. Unicode Encoding Scheme (UTF-8, UTF-16, UTF-32)

- An encoding scheme is a set of rules that dictates how a Unicode Code Point will be converted into a sequence of binary digits (bytes). To be stored on a hard drive or transmitted over a network.
- 3 well-known encoding schemes exist, they are described in the table below:

Encoding Scheme	Description
UTF-8	Variable-length: Uses 1, 2, 3, or 4 bytes to represent a Code Point, depending on the size of the Code Point.(Most common).
UTF-16	Variable-length: Uses 2 bytes for most common characters, and 4 bytes for less common ones. Common in Windows and Java systems.
UTF-32	Fixed-length: Uses 4 bytes for every Code Point, regardless of the character. Simplest for programming, but inefficient for storage or transmission.

- For example, the Greek letter alpha (α), with the Code Point U+03B1, is encoded $[03B1]_{UTF-8}$, $[03B1]_{UTF-16}$, and $[000003B1]_{UTF-32}$.

5.4. UTF-8 (Unicode Transformation Format 8-bits)

- It is the most common encoding for the internet because it is backward-compatible with ASCII. Standard ASCII characters are encoded in a single byte.
- Meaning all ASCII characters are valid UTF-8, while also being able to represent any character from any writing system.
- It is an encoding that is an encoding that uses a variable number of bytes per character, making it highly efficient.
- It successfully bridges the need for universal character support with storage and transmission efficiency.