ESD - Elettronica dei Sistemi Digitali

Data Representation

Prof. Riccardo Berta

2025.10.08

Contents

1 Data Representation

Digital systems process and store information in a form that must ultimately be expressed as numbers. Whether it is text, images, audio, or other signals, all data inside a computer or an digital system is represented using **numbers**. Understanding how different types of numbers are expressed and manipulated is therefore a fundamental step in digital design.

1.1 Number Systems

A number system defines **how digits are combined to represent quantities**, much like the familiar decimal system that we use in everyday life.

While humans are most comfortable with the decimal (base-10) system, digital hardware relies on the **binary (base-2) system**, in which all values are expressed using only two symbols. This is because it is much easier and more reliable to distinguish between two electrical states (such as high and low voltages) than between ten or more. Humans, instead have ten fingers, which naturally leads to a base-10 counting system.

Beyond binary and decimal, other number systems such as **hexadecimal** (**base-16**) play an important role in digital design. Hexadecimal serve as convenient shorthand for binary numbers, allowing engineers to read and manipulate long binary strings more easily.

Every operation in a digital system (from arithmetic and logic to memory addressing and data transmission) relies on these representations.

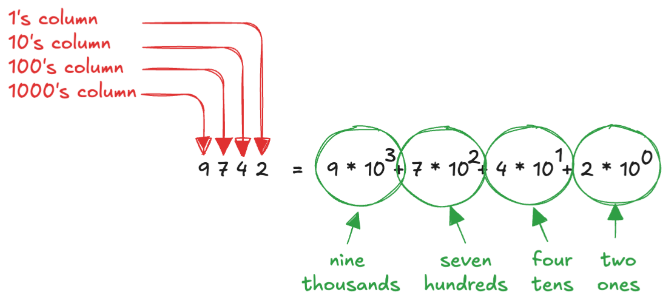
1.1.1 Decimal System

From an early age, we are taught to count and perform arithmetic in the decimal number system. This system is based on the number ten, a natural choice since humans have ten fingers, which historically guided the development of counting methods. The decimal system uses **ten distinct digits**:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Decimal digits can be combined to form larger numbers. The value of a digit depends not only on the digit itself but also on its position within the number (**positional notation**). Each position, or column, represents a power of ten.

For example, consider the number:



Here, the rightmost digit (2) represents two ones, the next digit (4) represents four tens, the next (7) represents seven hundreds, and the leftmost digit (9) represents nine thousands. Each step to the left increases the column weight by a factor of ten.

Because each column is weighted by a power of ten, decimal numbers are referred to as being in **base 10**. In general, a base indicates **how many distinct digits a number system uses** before carrying over to the next column. In decimal, the base is 10, so after the digit 9 comes the carry-over to the next column, forming 10.

An important property of the decimal system is that an N-digit decimal number can represent one of 10^N distinct values. For example:

- a 1-digit number ranges from 0 to 9, giving 10 possibilities.

- a 2-digit number ranges from 0 to 99, giving 100 possibilities.
- a 3-digit number ranges from 0 to 999, giving 1000 possibilities.

Thus, in general, an N-digit decimal number represents values in the range:

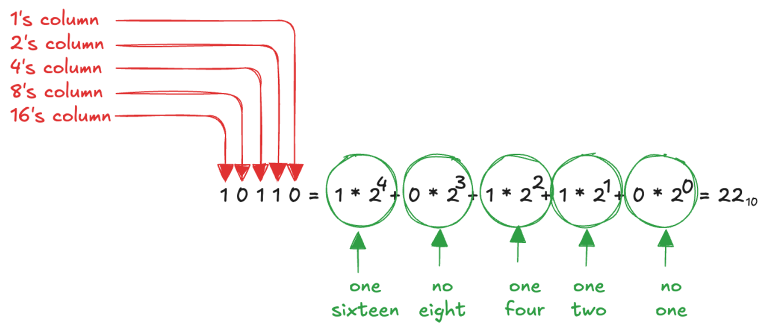
$[0\,,\ (10^N - 1)]$

By reviewing decimal numbers and their positional representation, we can smoothly transition to other number systems that are better suited to digital electronics.

1.1.2 Binary Numbers

A binary number is made up of **bits** (short for binary digits), each of which can take only one of two possible values: 0 or 1. This simplicity makes binary ideal for electronic systems, where **two distinct voltage levels** can be used to represent the two states reliably.

Just like decimal numbers use powers of 10, binary numbers **use powers of 2**. Each column of a binary number represents a power of 2, and each position has twice the weight of the previous one. The column weights in binary are therefore: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...



This shows how binary numbers can be directly converted to decimal by summing the values of the powers of two where there are 1s.

The number of distinct values that can be represented depends on the number of bits. An N-bit binary number can represent exactly 2^N different values.The table below shows this relationship:

1-Bit	2-Bit	3-Bit	Decimal
0	00	000	0
1	01	001	1
	10	010	2

1-Bit	2-Bit	3-Bit	Decimal
	11	011	3
		100	4
		101	5
		110	6
		111	7

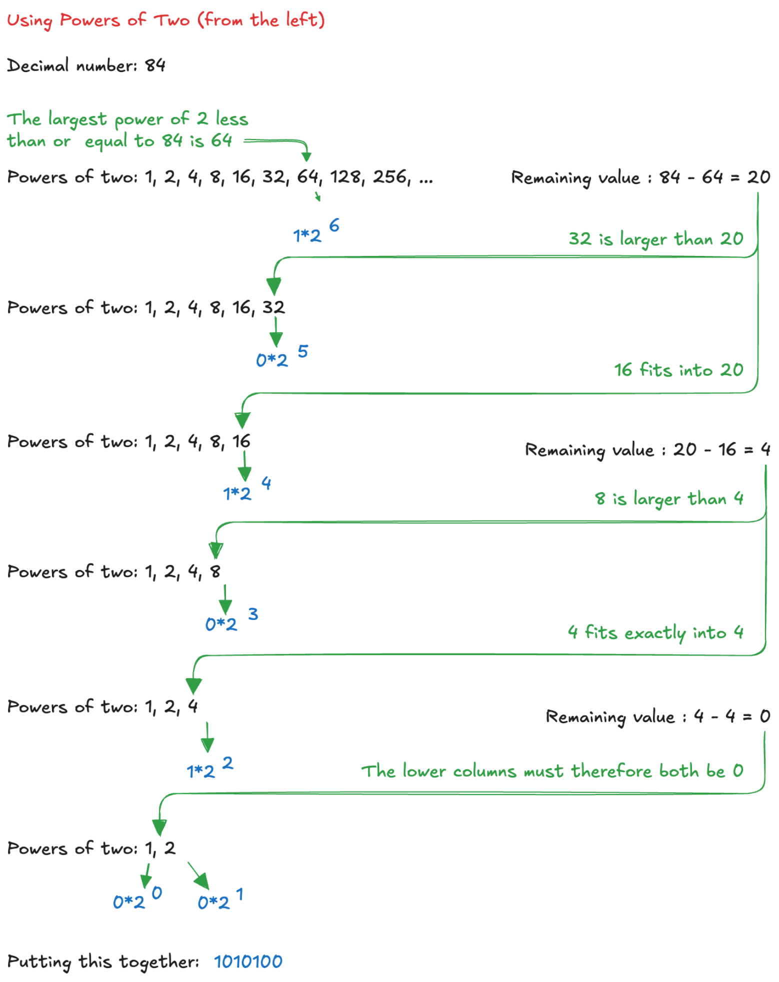
This illustrates how increasing the number of bits exponentially increases the number of representable values. For instance, with 8 bits, we can represent 256 distinct values, ranging from 0 to 255.

Binary numbers are the foundation of all digital electronics. Every operation inside a digital circuit relies on binary representation. By understanding how binary numbers work, we can begin to understand how larger and more complex data structures are built and manipulated within digital systems.

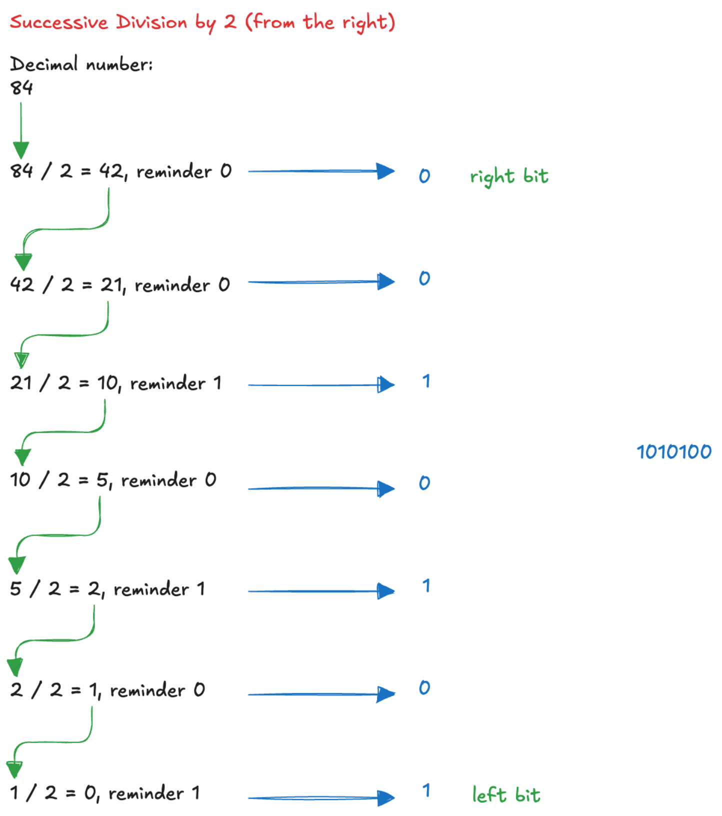
1.1.3 Decimal to Binary Conversion

Digital systems operate in binary, however humans have problems working with binary numbers directly. Therefore, it is essential to understand **how to convert numbers** from the decimal system (base 10) into binary (base 2). There are two common approaches to perform this conversion.

We can **work from the left** using powers of two. This method relies on identifying the largest power of 2 less than or equal to the decimal number and then subtracting it until the number is reduced to zero. For example, to convert 84 into binary:


Using Powers of Two (from the left)
Decimal number: 84
The largest power of 2 less than or equal to 84 is 64
Powers of two: 1, 2, 4, 8, 16, 32, 64, 128, 256, ...
Remaining value : 84 - 64 = 20
1*2 6
32 is larger than 20
Powers of two: 1, 2, 4, 8, 16, 32
0*2 5
16 fits into 20
Powers of two: 1, 2, 4, 8, 16
Remaining value : 20 - 16 = 4
1*2 4
8 is larger than 4
Powers of two: 1, 2, 4, 8
0*2 3
4 fits exactly into 4
Powers of two: 1, 2, 4
Remaining value : 4 - 4 = 0
1*2 2
The lower columns must therefore both be 0
Powers of two: 1, 2
0*2 0
0*2 1
Putting this together: 1010100

We can also **work from the right**. This method repeatedly divides the decimal number by 2 and records the remainders, which correspond to the binary digits from right to left. Using the same example:



Both methods give the same answer. The powers of two approach highlights the positional weights of binary digits, while the successive division method is straightforward and often used in manual conversions or algorithms.

1.1.4 Hexadecimal Numbers

Writing binary numbers is often cumbersome. Long strings of 0s and 1s are not only **tedious** to read and write, but also **prone to error**. However, conversion between binary and decimal number is not straightforward. For this reason, engineers commonly use the **hexadecimal number system** as a shorthand for binary.

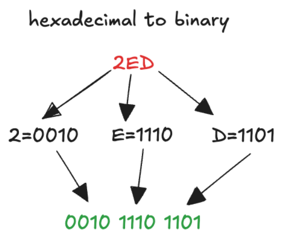
The hexadecimal system is **base 16**, meaning that each digit represents a value from 0 to 15. To cover all 16 possibilities, hexadecimal uses the decimal digits 0 through 9 together with the letters A to F:

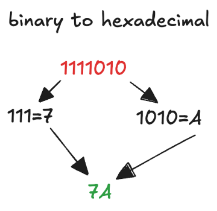
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Each hexadecimal digit **corresponds exactly to a group of four binary bits**. Since four bits can represent 2^4 = 16 different values, this mapping is natural and efficient:

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
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

This direct correspondence makes conversion between hexadecimal and binary particularly easy. To convert hexadecimal to binary, we simply **replace each hexadecimal digit with its four-bit binary equivalent**. To convert binary to hexadecimal, we **group the binary digits into groups of four, starting from the right, and replace each group with the corresponding hexadecimal digit**. For example:





Hexadecimal notation dramatically reduces the length of binary numbers and makes them easier to read and debug. For instance, the 32-bit binary value:

11010110100111101010110101011001

can be written much more compactly as:

D69EAD59

This compactness is why hexadecimal is widely used in programming, digital design, and computer architecture.

1.2 Data Units and Scaling

Data is not only represented in single bits, but also **grouped into larger units** that make storage and processing more efficient.

1.2.1 Bytes, nibble and words

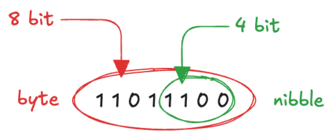
A **byte** is a group of **eight bits**. Since each bit can take two values, a byte can represent one of 2^8 = 256 different possibilities. For this reason, the size of objects stored in memory is **usually measured in bytes** rather than individual bits.

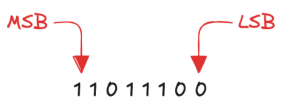
A smaller grouping of four bits is called a **nibble**. A nibble can represent one of 2^4 = 16 different values, which corresponds exactly to a single hexadecimal digit. Although the term nibble is less commonly used today, it remains useful when working with hexadecimal representations.

Microprocessors usually handle data in **larger chunks called words**. The size of a word depends on the **architecture** of the processor. For example, most modern computers use 64-bit processors, meaning that the processor can handle 64-bit words in a single operation. In the past, 32-bit processors were standard, and even earlier architectures supported 16-bit or 8-bit words.

Within a binary number, not all bits carry the same importance.

- the bit in the **rightmost position** is the **least significant bit** (**LSB**), since it represents the smallest weight (the ones place);
- the bit in the **leftmost position** is the **most significant bit** (**MSB**), since it carries the highest weight.





The same terminology applies at the byte level: within a multi-byte word, the **least significant byte** (**LSB**) is the one containing the lowest-order bits, while the **most significant byte** (**MSB**) contains the highest-order bits.

1.2.2 Kilo, Mega and Giga

Data sizes are often expressed in **multiples of bytes** using familiar prefixes such as **kilo**, **mega**, and **giga**. Although these prefixes originate from the decimal metric system, in computer science they are usually associated with powers of two:

- a **kilobyte** (**KB**) is defined as **2^10 = 1024 bytes**. This value is close to 10^3=1000, so the prefix kilo is used.
- a **megabyte** (**MB**) corresponds to **2^20 = 1.048.576 bytes**, which is approximately equal to one million (10^6) bytes.
- a **gigabyte** (**GB**) is defined as **2^30 = 1.073.741.824 bytes**, which is close to one billion (10^9) bytes.

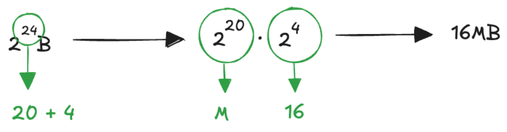
This works because the powers of two line up nicely with powers of ten. However, formally, the **IEC** (**International Electrotechnical Commission**) distinguish between **kibibyte** (KiB = 1024 B), **mebibyte** (MiB = 1024^2 B), **gibibyte** (GiB = 1024^3 B) and **kilobyte** (KB = 1000 B), **megabyte** (MB = 1000^2), **gigabyte** (GB = 1000^3). In this way the distinction between base 2 and base 10 is clear. However, in common usage KB, MB, and GB are still often used with base 2.

1.2.3 Estimate

When working with digital systems, big numbers come up all the time (like memory sizes or data limits). Instead of always reaching for a calculator, we can **estimate them quickly** if we remember just the smaller powers of two:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512

These are easy to remember because each number is just double the previous one. Now suppose we want to know the value of 2^24. We can break the exponent into parts that are easier to handle:



Here's the trick, 2^20 is a MB and 2^4 is 16, this gives 16 MB. The exact answer is 16,777,216, but the estimate is close enough for most practical purposes.

From memory addressing to data transfer, digital systems rely on these standardized groupings to ensure efficiency and compatibility across hardware and software.

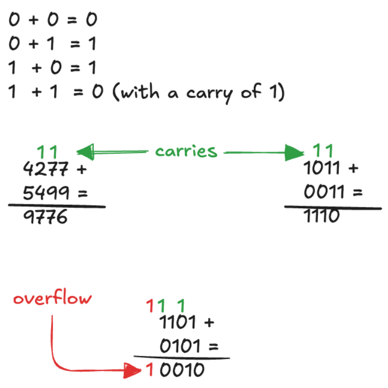
1.3 Arithmetic with Binary Numbers

Digital systems perform **arithmetic** using binary numbers, where simple rules for addition, subtraction, and multiplication must be adapted to handle the representation of both positive and negative values.

1.3.1 Binary Addition

Binary addition works according to the same principles as decimal addition, but it is actually simpler because there are only two digits. Just as in decimal addition, if the sum of two digits is greater than what a single digit can hold, we **carry over** to the next column.

However, in digital system we cannot store an infinite number of digits. Numbers are represented using a **fixed number of bits**, such as 4 bits, 8 bits, 32 bits, or 64 bits. This limitation means that sometimes the result of an addition is **too large** to fit in the available number of bits. This situation is called **overflow**. For example, a 4-bit number can represent values from 0 to 15, if the result of an addition is larger than 15, it cannot be stored correctly in 4 bits. The extra bit is discarded, producing an incorrect result within the 4-bit limit.



Overflow can be **detected** by checking for a carry out of the most significant column. If such a carry exists but cannot be stored, it means the result has exceeded the representable range.

This is an important point. Keep in mind that we are no longer in the purely mathematical realm; we are dealing with real systems that have specific limitations, and overlooking these constraints can lead to serious problems:

The $7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed. The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

1.3.2 Signed binary numbers

Of course, we can also consider **subtraction**, which works in a way similar to decimal subtraction. However, our focus is on how these operations are **implemented in hardware**. If we were to support both addition and subtraction directly, we would need **two separate circuits**: an adder and a subtractor. Instead, we can observe that subtraction can be expressed as the addition of a positive number and a negative number, thus reducing the two operations to a single one in hardware. This requires a representation for negative numbers that allows us to reuse the same addition circuitry.

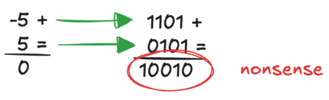
The most straightforward idea is the **sign/magnitude representation**:

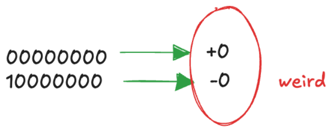
- the most significant bit (MSB) is used as the sign bit
- a sign bit of 0 indicates a positive number
- a sign bit of 1 indicates a negative number
- the remaining N-1 bits are used for the magnitude, or absolute value

This system is intuitively appealing because it **resembles our usual way of writing numbers**: a minus sign followed by the magnitude. For example a 4-bits numbers:

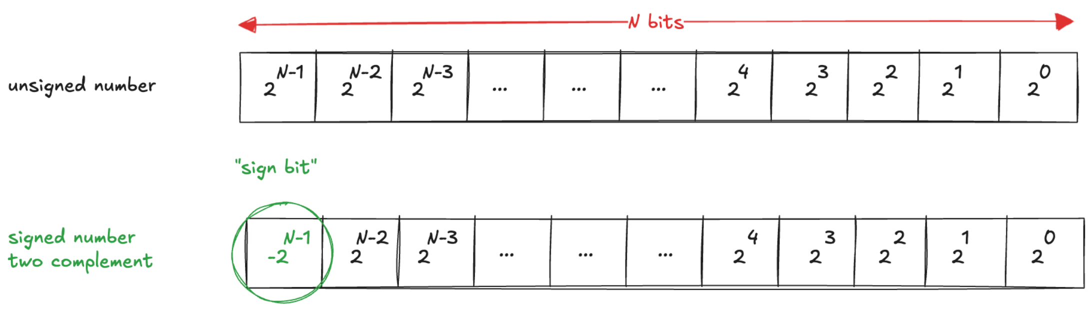
- 0101 = +5
- 1101 = -5

However, with this **ordinary binary addition does not work** and we have **two representations of zero**, which is a undesirable property:





Because of these issues, sign/magnitude is rarely used in digital systems. The most common solution is the **two's complement representation**. It works exactly like unsigned binary numbers, except that **the most significant bit (MSB) has a negative weight**:


N bits
unsigned number
2^{N-1} 2^{N-2} 2^{N-3} ... ... 2^4 2^3 2^2 2^1 2^0
"sign bit"
signed number two complement
-2^{N-1} 2^{N-2} 2^{N-3} ... ... 2^4 2^3 2^2 2^1 2^0

The following table show a 8-bit explanation:

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
...	...	...
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
...	...	...
11111110	254	-2
11111111	255	-1

The table illustrates how the same 8-bit binary pattern can be interpreted in two different ways: as an unsigned number (ranging from 0 to 255) or as a signed number (ranging from –128 to +127). If the MSB is 0, the number is non-negative, and the representation is the same as the unsigned case.

Example: 00000010 = +2.

If the MSB is 1, the number is negative, however the MSB now carries a weight of –128 (for the 8-bit case), while the other bits contribute with their normal positive weights. This means that 10000000 is interpreted as –128, 10000001 as –127, and so on, until 11111111, which represents –1.

Notice that the two's complement representation **covers the range from –128 up to +127**. This range is not symmetric: there is one more negative number than positive numbers. The reason is that the MSB has been reassigned a negative weight, allowing us to cover an extra value on the negative side, it is called the **weird number**, because it has no positive counterpart.

Notice that the range of an N-bit two's complement number is

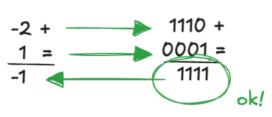
$[-2^{N-1}, 2^{N-1} - 1]$

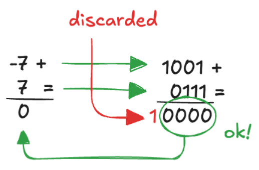
For example, the range of a 8-bit two's complement representation covers the range from –128 up to +127. This range is **not symmetric**: there is one more negative number than positive numbers, since there is no negative zero. The most negative value is therefore special, because it has no corresponding positive counterpart. This is sometimes referred to as the **weird number**.

The main advantage of two's complement representation is that **binary addition and subtraction work correctly for both positive and negative numbers**. This makes it possible to implement all arithmetic with a single adder circuit in hardware.

When adding two N-bit numbers in two's complement:

- if the result requires N+1 bits, **the extra carry out of the most significant bit is simply discarded**
- the remaining N bits represent the correct result, provided that no overflow occurs. Consider the following example with 4-bits words:



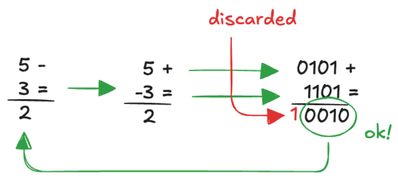


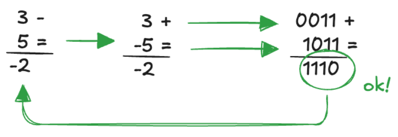
Since only 4 bits are stored, the fifth bit is discarded, leaving 0000, which correctly represents 0.

Subtraction is performed by converting it into addition:

A - B = A + (-B)

This works because negative numbers are already represented in a way that makes addition valid. Consider the following example using 4-bits words:

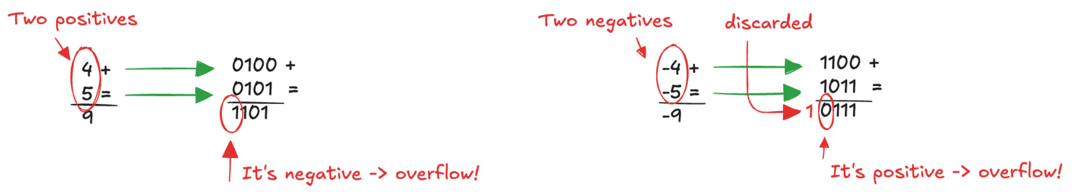




Another advantage is that **zero has only one unique encoding**, which avoids the redundancy present in other signed number systems.

However, we need to be careful: **overflow cannot be detected simply by checking the final carry bit** as with unsigned numbers. Fortunately, there is still a reliable way to detect it. The overflow can occur only if we are adding two numbers with the same sign, otherwise the result will be in the correct range. Notice that if we add two numbers with the same sign (both positive or both negative), the result should logically have the same sign:

- adding two positives should never give a negative
- adding two negatives should never give a positive If that happens, it means the true result was too large (for positives) or too small (for negatives) to fit in the available number of bits. The sign "flips" incorrectly because the representation has wrapped around, and this is exactly what we call overflow. For example:

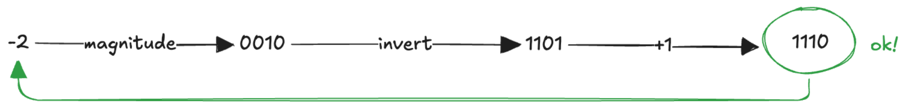


If the same operations had been performed using 5 bits, the correct results would have been obtained.

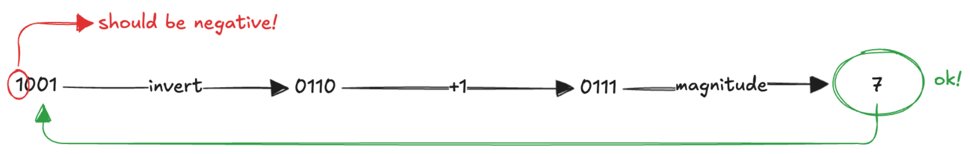
A fast procedure to obtain the two's complement representation of a negative number is straightforward:

- take the binary representation of the positive number (the magnitude)
- invert all the bits (this is called the one's complement)
- add 1 to the result.

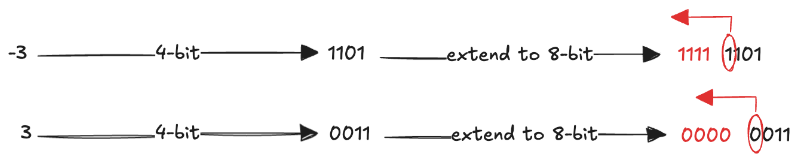
This process converts a positive number into its negative counterpart. Consider the following example, representing –2 in 4-bit two's complement word



We can reverse the procedure to get the magnitude of a negative number. For example, consider the 4-bit binary number 1001. We want to know its decimal value:

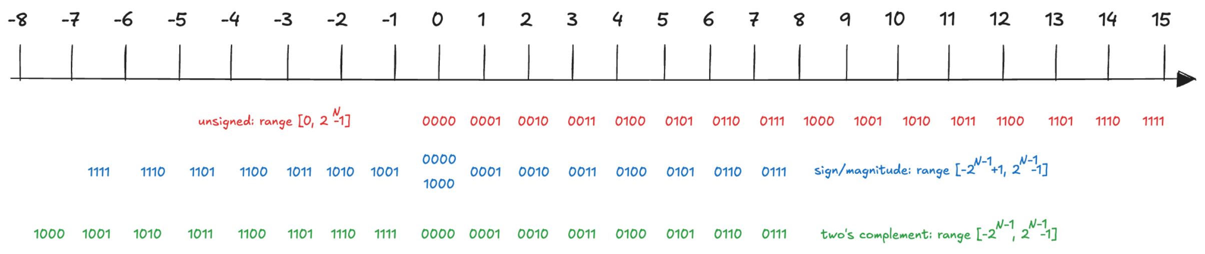


When a two's complement number is represented **with more bits**, its value must remain unchanged. To achieve this, the sign bit (the MSB) is copied into the newly added higher-order positions. This procedure is called **sign extension**. For instance:



Notice that the additional bits simply repeat the original sign bit, ensuring that the value is preserved.

Finally, we can provide a comparison between unsigned, sign/magnitude and two's complement representations in terms of ranges and value:

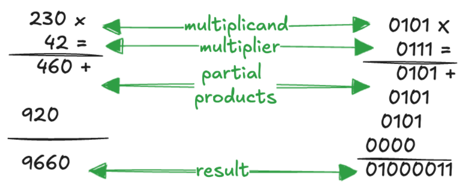


Two's complement is the standard in digial systems because it combines **efficiency** (one circuit for both addition and subtraction), **uniqueness** (only one zero), and **correctness** (arithmetic works seamlessly).

1.3.3 Binary Multiplication

Multiplication in binary follows the same principles as multiplication in decimal. The only difference is that binary numbers involve only two digits and this greatly simplifies the process of forming partial products. When multiplying two binary numbers:

- each bit of the multiplier is considered in turn
- if the bit is 1, the partial product is the entire multiplicand
- if the bit is 0, the partial product is simply 0
- each partial product is shifted according to the position of the multiplier bit, just as in decimal multiplication



In decimal multiplication, partial products may be multiples of the multiplicand, in binary multiplication, **each partial product is either the multiplicand or zero**, making the process simpler. The result of multiplying an M-bit number by an N-bit number may require up to **M+N bits**.

1.4 Fractional Numbers

We have focused on how digital systems represent integers, either signed or unsigned. However, digital systems must also handle fractions (or rational numbers), since many real-world value (such as temperature, voltage, or probabilities) cannot always be expressed as whole numbers. To represent fractions, we can use two main approaches: fixed-point numbers and floating-point numbers.

1.4.1 Fixed-point numbers

In fixed-point representation, numbers are expressed with an **implied binary point** between the integer part and the fractional part. This is directly analogous to the decimal point in ordinary decimal numbers. For example:

$0110.1100 = 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Here, the digits to the left of the binary point represent integer values, while the digits to the right represent fractional values.

Just as with integers, fixed-point numbers can also represent negative values. We ca use the **sign/magnitude notation**, in which the MSB is reserved for the sign of the number and the remaining bits represent the magnitude of the number. For example:

-2.375 is written as 1010.0110

where the leading 1 denotes the negative sign and the rest of the bits encode the magnitude. Or we can adopt the **two's complement notation**. For example:

-2.375 is written as 1101.1010

Notice that even though any integer number N can be written as a sum of powers of two:

$78 = 64 + 8 + 4 + 2 = 2^6 + 2^3 + 2^2 + 2^1$

decimal fractions are **rarely exact in binary**. For example, consider 0.78; in binary it is represented as an infinite series:

$0.78 = 0.11000100100001111110\ldots$

This is an infinite repeating expansion in base 2, just like $1/3 = 0.3333\ldots$ in base 10.

The reason is that:

$0.78 = 78/100 = 39/50 = 39/(2 \times 5^2)$.

In base 2, the denominator includes the factor 5, which is not a power of 2, and therefore the binary expansion does not terminate. Only fractions of the form

$\frac{k}{2^n}$

can be represented exactly in binary.

It is important to stress that fixed-point numbers are, at the hardware level, **nothing more than a sequence of bits**. The binary point is not physically present in the representation. Instead, its position is established only by convention. This means that, without prior agreement on the format, there is no way to determine where the binary point should be placed, and thus no way to correctly interpret the number. The interpretation always depends on the chosen representation scheme.

A common way to describe fixed-point formats is the **Q notation**, which makes explicit how many bits are assigned to the integer part and how many bits are assigned to the fractional part. In this notation:

- **Ua.b** denotes an unsigned fixed-point number, where a bits represent the integer part and b bits represent the fractional part;
- **Qa.b** denotes a signed fixed-point number in two's complement, with a bits for the integer part and b bits for the fractional part.

Fixed-point formats are widely used in practice, particularly in **digital signal processing** (DSP), **computer graphics**, and **machine learning** applications. These domains often require real-time computations on embedded hardware platforms and fixed-point arithmetic offers a good compromise between efficiency, precision, and hardware complexity. To give some concrete examples:

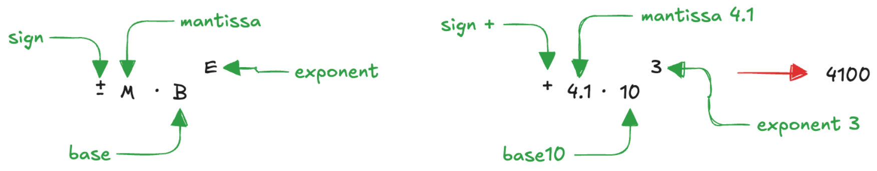
- Q1.15 uses 1 bit for the sign and 15 bits for the fractional part. It is the most common format in DSP applications because it offers high precision for fractional values.
- Q1.31 has 1 sign bit and 31 fractional bits. It is used when extremely high precision is required, for instance in intermediate stages of filtering or other numerical algorithms.

- U8.8 has 8 integer bits and 8 fractional bits in an unsigned representation. This format is frequently employed in sensor readings, where both a reasonable integer range and fractional precision are necessary.

However, fixed-point formats have some limitations. They **require a fixed agreement** on where the binary point lies, and **their range and precision are limited** by the number of bits allocated to the integer and fractional parts.

1.4.2 Floating-point numbers

Floating-point numbers are best understood by drawing an analogy with **scientific notation** in decimal arithmetic. In scientific notation, a number is expressed in the form:

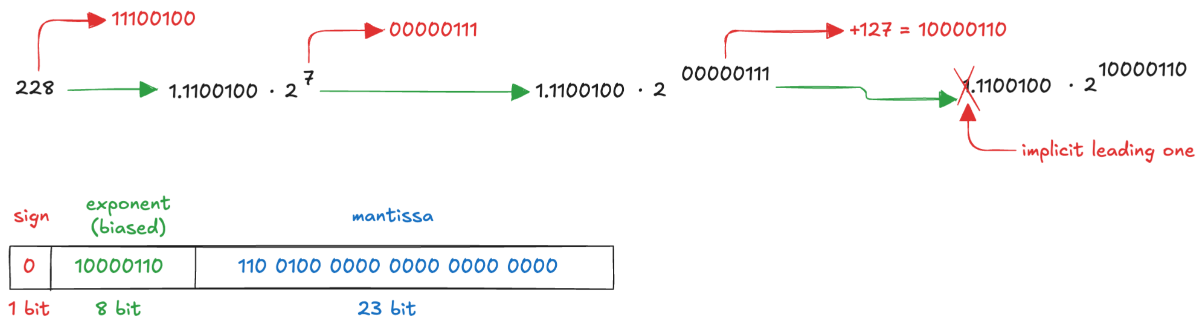


where:

- M is the **mantissa**, which contains the significant digits of the number,
- B is the **base** (10 in the decimal system, 2 in binary systems),
- E is the **exponent**, which shifts the decimal (or binary) point,
- and the leading *sign* indicates whether the number is positive or negative.

In the example, the mantissa is 4.1, the base is 10, and the exponent is 3. Notice that in this form the **decimal point "floats"** so that it always follows the most significant digit of the mantissa.

This idea extends naturally to binary floating-point numbers. By adjusting the exponent, the binary point can be shifted left or right, which makes it possible to represent both very large numbers and very small numbers using the same format. However, there are many reasonable ways to store the informatoin about mantissa, sing and exponent, and historically computer manufacturers developed their own formats independently. This lack of compatibility meant that results produced on one digital system could not easily be interpreted by another, which represented a significant barrier to portability and interoperability. To address this issue, the **Institute of Electrical and Electronics Engineers** (IEEE) introduced the **IEEE 754 Floating-Point Standard** (1985). This standard defines how floating-point numbers are represented and manipulated, and it has since become the universally adopted convention in modern computing. Today, virtually all processors and programming languages rely on IEEE 754, ensuring that floating-point calculations are consistent across platforms. For example, in the 32-bit single-precision format, one bit is allocated for the sign, eight bits for the exponent, and twenty-three bits for the fraction. The use of a bias (127 in the single-precision case) allows the exponent to represent both positive and negati value. For example, the number 228 can be represented as:



Notice that, in normalized floating-point numbers, the first bit of the mantissa is always 1. This bit, known as the **implicit leading one**, does not need to be stored explicitly, which allows one additional bit of precision to be represented within the available storage.

Notice that floating-point numbers can represent **a much wider range of values** than fixed-point numbers with the same number of bits. However, this flexibility comes at the cost of **increased complexity** in both hardware and software implementations. Floating-point arithmetic is more computationally intensive than fixed-point arithmetic, and it can introduce rounding errors and precision issues that must be carefully managed.

The IEEE 754 standard also defines several special cases:

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

The IEEE 754 standard, defines additional formats:

By providing a common and precise representation, IEEE 754 makes it possible to carry out floating-point arithmetic in a consistent way across different architectures. This consistency is essential for scientific computing, graphics, and any domain where reproducibility of numerical results is critical.

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits	Bias
Single	32	1	8	23	127
Double	64	1	11	52	1023
Quad	128	1	15	112	16363

In recent years, the growing importance of machine learning has driven the development of **new floating-point formats**. These formats **trade some precision for reduced storage and faster computation**, which is acceptable in many applications where approximate results are sufficient. Examples include:

- Google BFloat16: 16-bit format with 8 exponent bits and 8 mantissa bits.
- IBM DLFloat16: 16-bit format with 6 exponent bits and 10 mantissa bits.
- NVIDIA FP8: 8-bit format with 4 exponent bits and 4 mantissa bits.
- Tesla CFloat8: 8-bit format with 5 exponent bits and 4 mantissa bits.
- NVIDIA FP4: 4-bit format introduced in the NVIDIA Blackwell GPUs (2024).

These compact formats are not designed for general-purpose arithmetic, but they provide significant performance and memory advantages.

1.5 Exercises

1 - The Babylonians developed the sexagesimal (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the decimalnumber 4000 in sexagesimal?

- Information per digit:
  $I = \log_2(60) \approx 5.91$ bits
- Convert 4000 to base 60:
  $4000/60 = 66$ remainder 40
  $66/60 = 1$ remainder 6
  $1/60 = 0$ remainder 1
- Result:
  $4000 = 1 \times 60^2 + 6 \times 60^1 + 40 \times 60^0$

2 - How many different numbers can be represented with 16 bits?

- Each bit can take two values (0 or 1)

- With 16 bits, the total number of distinct patterns is:
  $2^{16} = 65.536$
- If the numbers are unsigned, the range is
  0 to 65.535
- If they are signed two's complement, the range is
  -32.768 to $+32.767$

3 - What is the largest unsigned 32-bit binary number?

- An unsigned 32-bit binary number uses all 32 bits for magnitude.
- The largest value occurs when all bits are 1:
  11111111 11111111 11111111 11111111
- In decimal, this equals:
  $2^{32} - 1 = 4.294.967.295$
- Usando i multipli:
  $2^{32} = 2^{30} \cdot 2^{2} = 4\ GB$

4 - What is the largest 16-bit binary number that can be represented with (a) unsigned numbers? (b) two's complement numbers? (c) sign/magnitude numbers?

(a) Unsigned numbers

- All 16 bits are used for the magnitude.
- Largest value:
  11111111 11111111 $= 2^{16} - 1 = 65.535$

(b) Two's complement numbers

- Range for N-bit two's complement:
  $[2^{n-1}\ ,\ (2^{n-1} - 1)]$
- For n = 16:
  $[-32{,}768\ ,\ +32{,}767]$
- Largest value: 32.767

(c) Sign/magnitude numbers

- 1 bit for sign, 15 bits for magnitude.
- Largest positive magnitude:
  0111111111111111 $= 2^{15} - 1 = 32.767$

5 - Convert the unsigned binary number 11110000 to decimal and to hexadecimal

- Binary to decimal:
  $11110000 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 = 128 + 64 + 32 + 16 = 240$
- Binary to hexadecimal:
  Group into 4-bit chunks: $1111\ 0000_2 = F0_{16}$

6 - Convert the hexadecimal 3B16 number to decimal

- $3B = 3 \times 16^1 + 11 \times 16^0 = 48 + 11 = 59$

7 - Convert the two's complement binary numbers 011100002 to decimal

- 01110000_2 has MSB = 0, so the number is non-negative.
- $01110000 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 = 64 + 32 + 16 = 112$

8 - Convert the decimal numbers −63 and −132 to 8-bit two's complement number or indicate that the decimal number would overflow the range

- For 8-bit two's complement, the representable range is -128 to $+127$.
- -63:
  $63 = 00111111$
  Invert bits: 11000000_2
  Add 1: 11000001_2
  Final: $-63_{10} = 11000001_2$
- -132_{10}:
  Since $-132 < -128$, it falls outside the 8-bit two's complement range.
  Result: **overflow**

9 – Convert the decimal number -5.75 into IEEE 754 single-precision format

- Sign bit: 1 (negative number)
- Convert integer part: $5 = 101$
- Convert fractional part: $0.75 = .11$
- Combined binary: 101.11
- Normalize: 1.0111×2^2

- Exponent: 2 + 127 = 129 = 10000001
- Mantissa: 0111000 … (fill to 23 bits)
- Final: 1 10000001 01110000000000000000000

10 – Convert the IEEE 754 single-precision binary number
0 10000001 $10011000000000000000000_2$ **into decimal.**

- Sign = 0, so it is positive
- Exponent = 10000001 = 129
- Unbiased exponent = $129 - 127 = 2$
- Fraction bits = 1001100 …
- Ssignificand = 1.10011_2
- $1.10011 = 1 + 2^{-1} + 2^{-4} + 2^{-5} = 1.59375$
- Value = $1.59375 \times 2^2 = 6.375$