Chapter 3: Coding of Symbols and Non-Integer Numbers

1. Negative integers

Negative integers can be represented and encoded in various ways depending on the number of bits used and the specific encoding method employed. The three most common encoding methods are:

1.1. Sign-Magnitude representation

In sign-magnitude representation, a negative integer is represented by setting the most significant bit (MSB) as the sign bit. If the sign bit is 1, it indicates a negative number, and if it is 0, it represents a positive number. The remaining bits represent the magnitude of the number.

In this way, with a k-bit word, we can encode positive or negative integers N, such that N is within the range:

$$-(2^{k-1}-1) \leq N \leq +(2^{k-1}-1)$$

The disadvantage of this method is that 0 has two distinct representations: 000..0 and 10..0, representing +0 and -0, Also, arithmetic operations become complicated due to the sign bit, which needs to be handled separately.

1.2. The ones' complement and the two's complement.

1.2.1. Ones' complement:

In ones' complement representation, to get the negative value of a binary number, you flip all the bits (change 0s to 1s and 1s to 0s). This means that the negative representation of a number is obtained by taking the ones' complement of its positive representation. The ones' complement does not have a dedicated negative zero representation, and the range of representable numbers is:

$$-(2^{k-1}-1) \leq N \leq +(2^{k-1}-1),$$

1.2.2. Two's complement:

In two's complement representation, you flip all the bits and add 1 to the result to obtain the negative value. The two's complement method has a unique representation for zero and simplifies arithmetic operations since there is no negative zero. The range of representable numbers is asymmetric, with one more negative value than positive values:

$$-2^{k-1} \leq N \leq +(2^{k-1}-1)$$

Example: The representation of (-6) using 4 bits is as follows:

- In sign-magnitude: **1110**
- In ones' complement: **1001**
- In two's complement: **1001** + **1** = **1010**

Question: What is the range of signed numbers that can be represented using 4 bits for the 3 methods?

- We notice that the leftmost bit (sign bit) in all three methods is always 1 for negative numbers and 0 for positive numbers.
- In ones' complement and two's complement, arithmetic operations are advantageous because subtracting a number is equivalent to adding its complement. This means we can use just circuits performing the addition, and there is no special treatment required for the sign bit.
- In an addition using ones' complement, if there is a carry generated by the sign bit, it must be added to the obtained result. However, in two's complement, this carry is just ignored.

Example: subtracting 4-bit numbers:

Decimal	Sign-Magnitude	Ones' complement	Two's complement
+ 7	0 1 1 1	0 1 1 1	0 1 1 1
- 6	+ 1 1 1 0	+ 1 0 0 1	+ 1 0 1 0
= + 1	= ? 1 0 1	= 1 0 0 0 0	= 1 0 0 0 1

Note: In two's complement, an overflow only occurs if the carries generated just before the sign bit and by the sign bit itself are different.

Example: Two's complement addition on 3 bits:

Decimal	**Two's complement**
(- 4)	**1 0 0**
(- 1)	**1 1 1**
= - 5	**= ~~1~~ 0 1 1 ≠ - 5 , Therefore, the result is incorrect.**

2. The Fractional Numbers

2.1. Floating-Point Representation

2.1.1. The Standard Classic Representation

We know that it is necessary to store data in computers. Thus, the number 9,750 will be stored in the following form: 1001,11. However, this binary expression is not enough to fully define our data because there is no indication of the value of the binary weight assigned to different bits, hence the concept of the decimal point.

By using this concept of the decimal point, our number can be written as follows:

- $N = 1001,11 \times 2^0$
- $N = 100,111 \times 2^1$
- $N = 10,0111 \times 2^2$
- $N = 1,00111 \times 2^3$
- $N = 0,100111 \times 2^4$

This last expression has the advantage of representing the value as a number less than 1 multiplied by a power of 2. The exponent 4 (100 in binary) represents the position of the decimal point. Therefore, to fully define our information (9,750) in this representation system, two terms are required: the term 100111 called Mantissa and the term 100 called Exponent.

So, the floating-point representation consists of representing numbers N in the following form:

$N = M \times B^E$ with :

- B : Base (in our case, we are studying B=2)
- M : Mantissa
- E : Exponent

The exponent is an integer, and the mantissa is a purely fractional number (having no significant digits to the left of the decimal point). The mantissa is normalized, meaning it has the maximum number of significant digits: the first bit to the right of the decimal point is always 1 (e.g., 0.101110). Except for the value 0 (which is generally represented by the word 00…0), we always have:

$$(0.1)_2 \leq |M| < (1)_2 \text{ either } (0.5)_{10} \leq |M| < (1)_{10}$$

The exponent and the mantissa must be able to represent both positive or negative numbers, so they could be encoded in sign-magnitude, ones' complement, or two's complement form. Often, the mantissa is in ***sign-magnitude*** form, while the exponent is unsigned but ***biased*** (or shifted).

Example :

S	E	M

Where :

- **S** : is the sign of the mantissa,
- **E** : is the biased exponent
- **M** : is the mantissa.

With 4 bits, for example, we can represent $2^4 = 16$ values of E, ranging from 0 to 15. We can match the first 8 values (from 0 to 7) to a negative exponent and the next 8 values (from 8 to 15) to a non-negative exponent. A zero exponent is represented by the value 8, an exponent equal to +1 by the value 9, and an exponent equal to -1 by the value 7. This means that the bias is equal to 8. The bias value is subtracted from the biased exponent (ranging from 0 to 15) to obtain the effective exponent (ranging from -8 to +7)

Example: Representation of signed integers on 3 bits (exponent coded on 3 bits →8 possible values between 0 and 7)

Decimal	Sign-Magnitude	One's complement	Two's complement	Biased representation
+ 3	0 1 1	0 1 1	0 1 1	1 1 1
+ 2	0 1 0	0 1 0	0 1 0	1 1 0
+ 1	0 0 1	0 0 1	0 0 1	1 0 1
0	0 0 0 // 1 0 0	0 0 0 // 1 1 1	0 0 0	1 0 0
- 1	1 0 1	1 1 0	1 1 1	0 1 1
- 2	1 1 0	1 0 1	1 1 0	0 1 0
- 3	1 1 1	1 0 0	1 0 1	0 0 1
- 4	------	------	1 0 0	0 0 0

One can observe that the biased representation is identical to the two's complement, except for the sign bit, which is reversed. In biased representation, the sign bit being 1 corresponds to values greater than or equal to 0, and when the sign bit is 0, it corresponds to values less than 0.

The exponent determines the range of representable numbers, and the size of the mantissa determines the precision of these numbers.

2.1.2. The IEEE 754 standard in single precision

A fractional number, according to this standard, is represented using 32 bits divided into 3 parts

- **S** is the sign of the mantissa. S ← 0 IF M ≥ 0. S ← 1 If M < 0.
- $\mathbf{E_b}$ is the biased exponent, encoded using 8 bits and calculated using the following formula:

$$Eb = E_{real} + (2^{8-1} - 1) = E_{real} + 127$$

- **M** is the mantissa, encoded using 23 bits. In the IEEE 754 standard, the decimal point is placed after the bit set to 1 with the highest weight (not like the classic representation).

For example :

$(11011,01)_2 = (0,1101101 \times 2^5)$ ← according to the classical standard

$= (1,101101 \times 2^4)$ ← according to IEEE 754 standard

$= (1,M \times 2^4)$

Where M = 101101

Example : $(100011,01)_2 = (0,10001101 \times 2^6) = (1,0001101 \times 2^5)$

The number is positive So S = 0

Eb = $E_{real} + 127 = 5 + 127 = (132)_{10} = (10000100)_2$

M = 10001101

The representation of the number $(11011,01)_2$ in floating point according to the IEEE 754 standard in single precision is:

0	10000100	10001101000000000000000

2.1.3. The IEEE 754 standard in double precision:

A fractional number, according to this standard, is represented using 64 bits divided into 3 parts:

- **S :** encoded using 1 bit.
- **Eb** : encoded using 11 bits and calculated using the following formula :

$$Eb = E_{real} + (2^{11-1} - 1) = E_{real} + 1023$$

- **M** : encoded using 52 bits.

Example : $(-100011,01)_2 = (-0,10001101 \times 2^6) = (-1,0001101 \times 2^5)$

The number is negative, so S = 1

$Eb = E_{real} + 1023 = 5 + 1023 = (1028)_{10} = (10000000100)_2$

M = 10001101

The floating-point representation of the number $(-100011,01)_2$ in IEEE 754 double precision is :

1	10000000100	1000110100000000000000000000000000000000000000000000

2.1.4. Arithmetic operations in floating-point representation

For **multiplication**, you need to add the exponents, multiply the mantissas, and re-normalize the result if necessary.

Example : $(0.2 \times 10^{-3}) \times (0.3 \times 10^{7}) = ?$

- Add the exponents: $-3 + 7 = 4$
- Multiply the mantissas: $0.2 \times 0.3 = 0.06$
- Result before normalization: 0.06×10^{4}
- Normalized result: 0.6×10^{3}

For **division**, you need to subtract the exponents, divide the mantissas, and re-normalize the result if necessary

For **addition**, the exponents must have the same value. Therefore, you may need to de-normalize the smaller value to bring its exponent to the same value as the larger number. After adding the mantissas, normalization may be required.

Example : $(0.300 \times 10^{4}) + (0.998 \times 10^{6}) = ?$

- Denormalization: $0.300 \times 10^{4} \rightarrow 0.003 \times 10^{6}$
- Add the mantissas : $0.003 + 0.998 = 1.001$
- Normalization of the result: $1.001 \times 10^{6} \rightarrow 0.1001 \times 10^{7}$

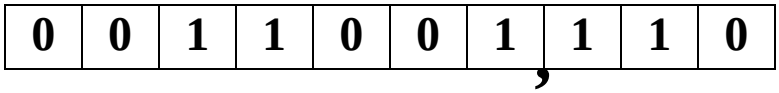
Subtraction is performed similarly to addition, but you need to perform subtraction of the mantissas instead of addition.

2.2. Fixed-Point Representation

The representation of numbers in floating-point format is not the only option available. There is also the representation of numbers in fixed-point format. The difference is that the number of digits after the decimal point (the rightmost position) is always the same, thus providing a fixed precision for the represented fractional numbers.

Example :

Let's take $(25,75)_{10}$ as an example, represented in binary as $(11001,110)_2$



In this configuration, the position of the decimal point is **fixed** (between the 3rd and 4th bit). Since the decimal point is not explicitly shown or represented, by default, the rightmost bit represents the weight of $\mathbf{2^0}$, which is incorrect in this case. This representation assumes an implicit multiplication of this number by $\mathbf{2^{-3}}$ to obtain the exact value. The term **-3** represents the fixed position of the decimal point and must be stored in the machine to interpret the number correctly.

3. The different types of data encoding

3.1 BCD (Binary Coded Decimal) Code:

This type of encoding seeks to combine the advantages of the decimal system and the binary code. It is commonly used for displaying decimal data, such as in calculators. Each decimal digit is represented by a binary word (usually four bits).

To encode a decimal number in BCD, each digit of the decimal base number is separately encoded into binary.

Example : (BCD sur 4 bits) : $1985 = 0001\ 1001\ 1000\ 0101_{(BCD)}$

Note:

- The BCD representation of a number is not equivalent to the natural binary representation of the decimal number. BCD coding is simple, but mathematical operations cannot be directly performed on it.
- There are several types of BCD codes, but the one presented in this section is the most commonly known.

3.2 EBCDIC (Extended Binary Coded Decimal Interchange) Code:

This code is mainly used by IBM. It is represented using 8 bits and is used for character encoding, where each character is associated with its EBCDIC code.

Example :

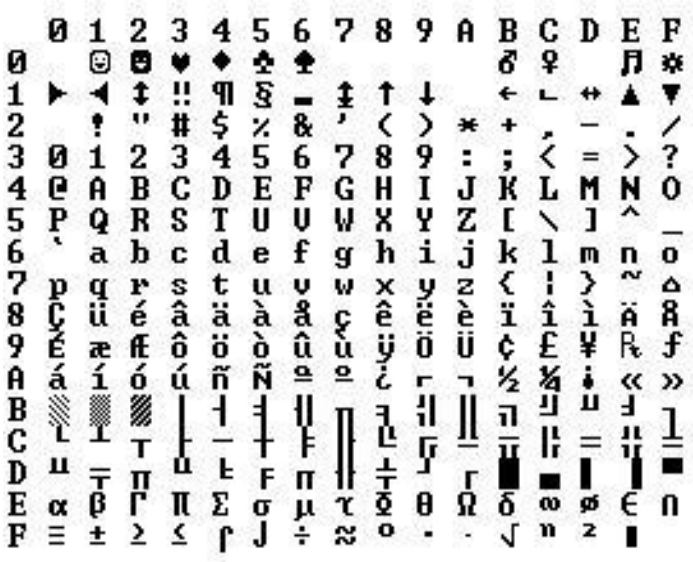
- The code of the uppercase character '**A**' in **EBCDIC** = **11000001**
- The code of the character '**0**' in **EBCDIC = 11110000**

3.3 The ASCII Code: *American Standard Code for Information Interchange*

This code is a widely used character encoding standard. It was developed in the early 1960s to standardize the representation of characters and control characters in computers and communication devices.

The ASCII code offers different extensions depending on the "code page". For example, the Code page **850** is a "multilingual" character set, while code page **864** defines the Arabic character set, and code page **437** defines the French character set, etc...

La table des codes ASCII



Note: The ASCII code table above displays printable characters and not control codes. Indeed, the characters with decimal codes 10, 13, and 27 respectively represent Line Feed (LF, which moves the cursor to the next line), Carriage Return (CR, which moves the cursor to the beginning of the current line), and Escape (ESC, used for various purposes like initiating special sequences in terminal control)

3.4 Gray Code :

Gray code is a type of binary encoding that ensures that only one-bit changes at a time when a number is incremented by one. This property is important for several applications.

To construct the Gray code of a number N, you can simply calculate the exclusive OR (XOR) between N expressed in binary and the same binary number shifted one position to the right.

The **exclusive OR (⊕)** is a logical operation that returns 0 if the two operands are the same and 1 otherwise.

A	B	A ⊕B	
0	0	0	⟹ A and B identical
0	1	1	
1	0	1	⟹ A and B different
1	1	0	

Exemples : Let's represent 10 in Gray code. $(10)_{10} = (1010)_2$.

1 0 1 0 ⟹ 10 in binary

⊕ 0 1 0 1 ⟹ 10 in binary shifted by one place to the right

1 1 1 1

Finally, we find that **10** in decimal is represented by **1111** in Gray code.