Numeric codes. Byte orders. Signed numbers. Real numbers.

EDIT LAUFER

Numeric codes

Above we examined the transformation between the number systems. In the following, we'll see how to use them on a computer, i.e. when we would like to generate a code. We talk about numeric code in the case when numbers are represented only.

Binary code

Actually, the binary form of the number, but it is very important that we do not deal with the number of bits in the case of a simple conversion. However, when we want to produce a numeric code, we must agree about the code length and all numbers must be written at this specified size (e.g.: 8, 16, 32 bit)

Hexadecimal code

It isn’t a real number system, the computer does not work directly with it. It is used formally to describe the binary numbers in a shorter, more manageable form. The conversion between the binary and hexadecimal numbers systems is precise, which is also an important aspect of applicability.

Conversion: The hexadecimal value of the 4 bit – groups is generated.

Binary form:	0011	0110	0100	1010	1100	1100	0101	0111
Hexadecimal form:	3	6	4	A	C	C	5	7

BCD (Binary Coded Decimal)

The code name refers to adding a binary code to the digits of the decimal number based on a code table.

BCD is actually a code family, as there are several versions of it. We will look at these variations in the following.

BCD - 8421

Decimal value	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The name suggests that the 4-bit binary value of the digit is used to represent it during the coding.

E.g.: 519 = $\underbrace{0101}_{5}\ \underbrace{0001}_{1}\ \underbrace{1001}_{9}$

BCD – Excess-3

Decimal value	Excess-3
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

The name suggests that the 4-bit binary value of the digit increased by 3 is used to represent it during the coding.

It’s advantage is that 0,1 consistently, resulting in more consistent energy consumption and warming

E.g.: 519 = $\underbrace{1000}_{5}\ \underbrace{0100}_{1}\ \underbrace{1100}_{9}$

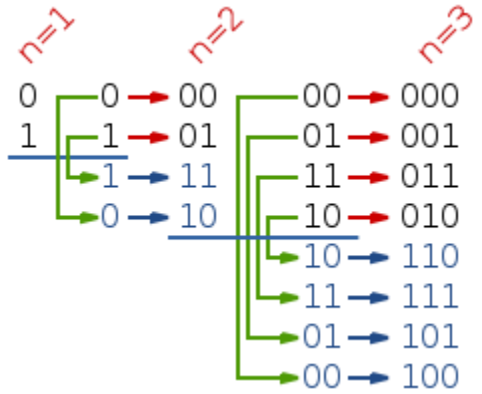
BCD - Gray

Decimal value	Gray
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101

It significance is that the adjacent digits differ only in one bit.

It can be used in logic circuits to encode stepping (e.g. numeric control, CNC).

Generation:



E.g.: 519 = 0111 0001 1101

The codes in the current level are horizontally reflected (blue line), then at the next level, above the line we write 0 before that, and nelow the line we write 1 before that.
The code of the current level (bit number) can be found in columns n=1, n=2, n=3, respectively.

Properties of BCD code

Advantages:

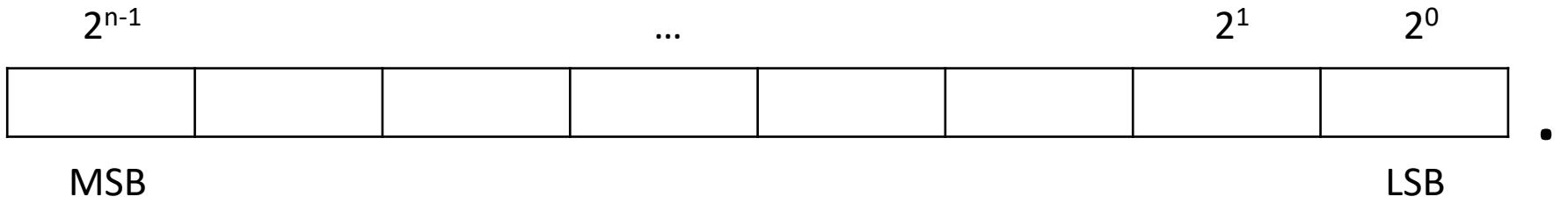
- Easy conversion, since the values can be read from a table
- Precise conversion back and forth, i.e. we can avoid the imprecision problem during the conversion from decimal number system to binary.

Disadvantages:

- Bigger size, since 4 bit is needed to represent ine digit
- Redundant, since we can represent 2^4=16 different values in 4 bit, but we have only 10 digits.
- Slower operation due to transmission

Number representation - Integer

As mentioned in the previous part, when we would like to generate a numeric code, we should define the code length and all numbers must be written at the specified size (e.g.: 8, 16, 32 bit). Its general form:



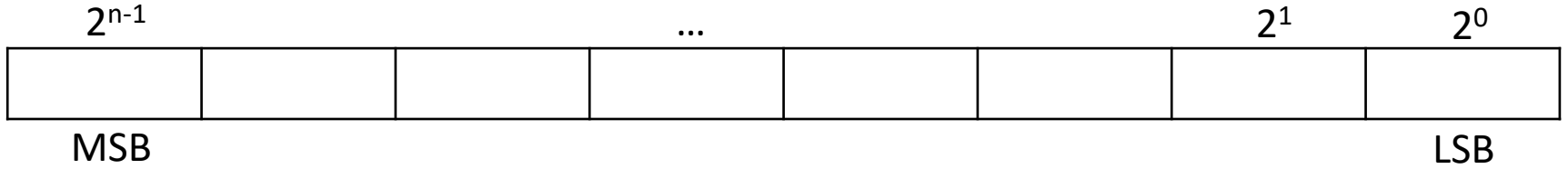
MSB: Most Significant Bit, generally the sign bit

LSB: Least Significant Bit, legkisebb helyiértékű bit, jobboldalon, mert az arab számokat így írjuk

The biggest positive number: 2^n-1

Number representation – Signed integer

If we would like to represent a signed number, we shoulr store also the sign of the number, so the previous form is modified to store the sign of the number in the first bit. Sign bit (0 – positive, 1 – negative), then the absolute value of the number. This is the so called signed absolute value form:



The biggest positive number: $2^{n-1} - 1$

(The exponent decreases by one, compared to the signless numbers, since there are fewer bits available to represent the number)

The biggest negative number: 2^{n-1}

(This is greater by one, than the positive number, because there is no minus zero)

Number representation – Negative numbers

The aforementioned signed absolute value form is managable for us, but not for the computer. Operation is complicated.

Formerly the computers were huge in size, vacuum tube, they were expensive, consumed a lot of energy, so we had to pay attention to using as little energy as possible. Addition is the most important operation among the basic operations, since multiplication is a repeated addition, division is a repeated subtraction, and the subtraction can be traced back to the addition applying the complement form. For this reason it is enough if we have an adder in the computer, no separate subtraction machine is required. Accordingly, hereafter we will see how can we create the complement form, and how this can be used during subtraction.

Negative numbers – complement creation

During the complement creation, we start from the signed absolute value form.

1. Signed absolute value form: sign bit + absolute value of the number
2. 1st complement: the number, which supplements the absolute value of the number (without the sign bit) to $2^{n-1}-1$. This number is obtained by changing all bits of the signed absolute value form into the opposite, except the sign bit.

Negative numbers – complement creation

One's complement is just a tool, an intermediate step for to create 2nd complement, we will not use it directly.

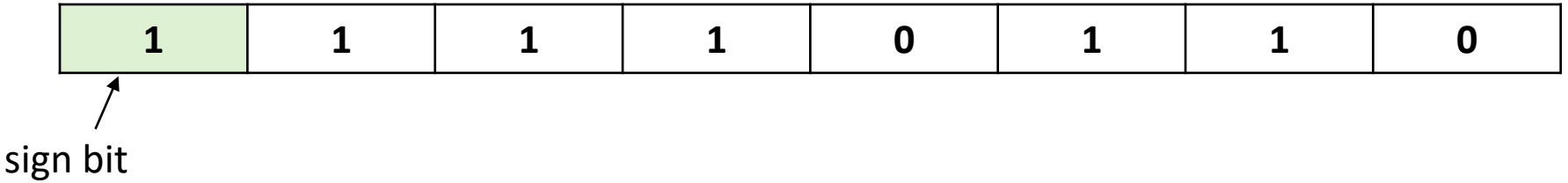
3. 2nd complement form: 1st complement + 1

   Negative numbers are used in this form during the operations. It is important to note, however, that in case of positive numbers, the signed absolute value is used.

Negative numbers – complement creation - Example

Create the 2nd complement of -118.

1. The signed absolute value form:



2. Creation of the 1st complement (changing the bits to the opposite, except the sign bit):

Negative numbers – complement creation - Example

Create the 2nd complement of -118.

3. Creation of the 2nd complement (1st complement+1) :

1	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1st complement**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**+1**
1	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**2nd complement**

4. That is, the resulting 2nd complement form:

1	**0**	**0**	**0**	**1**	**0**	**1**	**0**

Operations with signed numbers

It was mentioned earlier that the computer is doing the subtraction by transforming the number to be extracted to 2nd complement form, while the positive number is given in the signed absolute value form, then perform the addition. As it can be seen in the following part.

Operations by binary numbers

Adder table

+	0	1
0	0	1
1	1	10

Multiplication table

*	0	1
0	0	0
1	0	1

Note: when the result is a 2-bit number, we write down the lower bit, and the higher bit is displayed as a transmission at the previous bit (e.g. in the case of 10, we write down the 0, and 1 is the transmission)

Operations with signed numbers

1. Create the signed absolute value form of the number.
2. The positive numbers are left in the signed absolute value form, the negative numbers are transformed to 2nd complement form (subtrahend).
3. Perform the addition.
4. We examine the sign of the result and proceed accordingly. (See next slide)

The result of the operation for signed numbers

1. If the result is positive (sign bit 0):
   - We obtained the signed absolute value form, we are ready.
2. If the result is negative (sign bit 1):
   - The result is in 2nd complement form, so we need to revert to the signed absolute value form, to convert it into a decimal number system
   - Change all bits to the opposite, except the sign bit.
   - Add 1 to the result.

Operations with signed numbers – Example 1.

The operation to be performed: 46-111

1. Create the signed absolute value form of the numbers:

46:	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**
-111:	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**

2. Create the 2nd complement form of the negative number:

1	**0**	**0**	**1**	**0**	**0**	**0**	**1**

Operations with signed numbers – Example 1.

3. Perform the operation:

	0	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**46 (signed abs. value)**
+	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**-111 (2nd complement)**
	1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**Result**

4. The result is negative, i.e. it is in 2nd complement form:

1	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**Opposite bits**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**+1**
1	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**Signed abs. Value form of the result**

Transformation to the decimal number system: $-(2^6+2^0)=-65$

Operations with signed numbers – Example 2.

The operation to be performed : 118-116

1. Create the signed absolute value form of the numbers :

118:	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**0**
-116:	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

2. Create the 2nd complement form of the negative number :

1	**0**	**0**	**0**	**1**	**1**	**0**	**0**

Operations with signed numbers – Example 2.

3. Perform the operation :

	0	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**118 (signed abs. value)**
+	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**-116 (2nd complement)**
1	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**result**

Overflow due to transmission. We don’t deal with it, because it hasn't got space in the available 8 bits, it will be lost. The sign bit is 0.

4. The result is positive, i.e. it is in signed absolute value form:

5. Transformation to the decimal number system : $2^1=2$

Fixed-point representation – real numbers

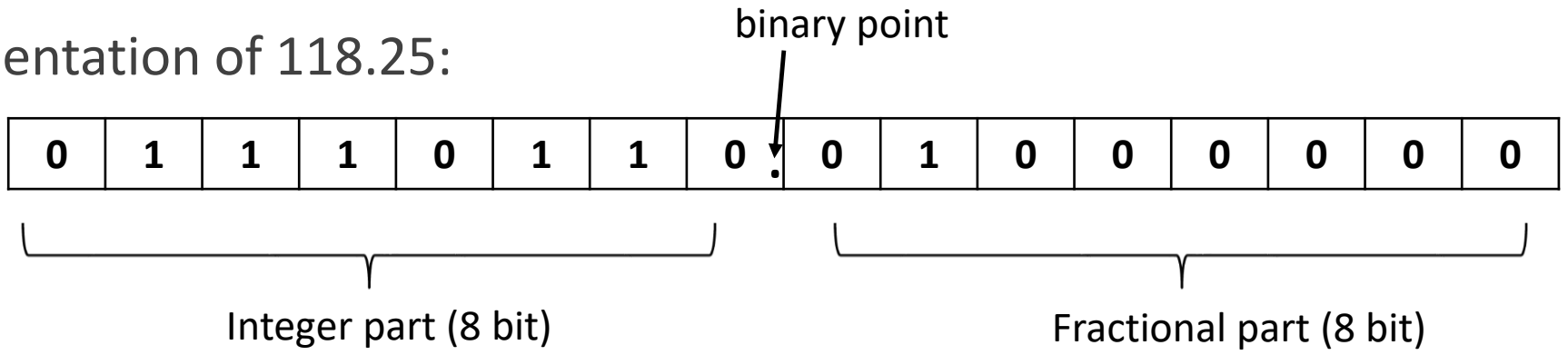
The position of the binary point is fix, i.e. the size of the integer part and of the fractional part are fixed.

Accuracy and magnitude of the numbers depend on the position of the binary point

- If you push the point to the left, the accuracy increases, while the magnitude decreases
- Moving the point to the right the magnitude increases and the accuracy decreases

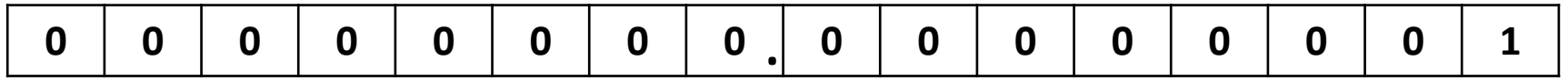
Fixed-point representation - example

Representation of 118.25:



Uneconomical (many bits are used for very small and very big numbers).

E.g.:



Floating-point representation – real numbers

In order to eliminate problems with fixed point representation, there is a need for another representation mode. Its basic principle is to store the normal form of the number:

$$N = \pm M \cdot 2^{\pm k}$$

M: mantissa (significand) – accuracy depends on its size

k: characteristics (exponent) – magnitude depends on its size

General form:

Mantissa sign bit	Characteristic including its sign	Mantissa

Steps of the floating point representation

1. Prescribing the number binary
2. Transform to normal form, i.e. shifting the binary point to obtain 0 as the integer part and the forst bit of the fractional part be 1. Of course, this number must be multiplied by the appropriate power of two, so that its value does not change.

Floating point representation - example

The number: 118.25

1. Binary form of the number: 1110110.01
2. Normal form of the number: $0.111011001*2^7$
3. Mantissa: 111011001, Characteristics: 111 (binary form of 7)

Note: The representation of the number is described in more detail below.

Characteristics of floating point representation

- The small numbers can be represented with great precision
- The big numbers can be represented with small precision, but it isn’t so important
- The size of the mantissa and characteristics are defined in IEEE standards (The following slide shows the characteristics of some basic types used in C #.)

Floating-point number types

Type	Domain	Storage	Mantissa	Characteristics
Single	$1{,}5 \cdot 10^{-45} \ldots 3{,}4 \cdot 10^{38}$	4 byte	23 bit	8 bit
Real	$2{,}939 \cdot 10^{-39} \ldots 1{,}701 \cdot 10^{38}$	6 byte	39 bit	8 bit
Double	$5{,}0 \cdot 10^{-324} \ldots 1{,}797 \cdot 10^{308}$	8 byte	52 bit	11 bit
Extended	$3{,}4 \cdot 10^{-4932} \ldots 1{,}1 \cdot 10^{4932}$	10 byte	63 bit	15 bit

Operations with binary real numbers

Addition

1. Shift the characteristics towards the larger
2. Perform the addition
3. Transform the result to normal form

Multiplication

General form of the numbers: $N_1 = M_1 \cdot 2^{k_1}, N_2 = M_2 \cdot 2^{k_2}$

Multiplication: $N_1 \cdot N_2 = m_1 \cdot m_2 \cdot D^{k_1 + k_2}$

Transform the result to normal form

Addition - example

Perform the following operation in binary number system using floating point representation:

16.5 + 7.5

1. Transform the binary form of the numbers to normal form:

   16.5 = 10000.1 → $0.100001*2^5$

   7.5 = 111.1 → $0.1111*2^3$

2. Shift the characteristics towrds the larger:

   $0.100001*2^5$ (we leave it unchanged, because it is the bigger characteristics)

   $0.001111*2^5$ (shift the binary point to the left by two, because we should increase the characteristics by 2)

Addition - example

3. Perform the operation:

$$\begin{array}{r} 0.100001 \\ +\ 0.001111 \\ \hline 0.110000 \end{array}$$

   The result: $0.110000*2^5$

4. The result is in normal form, so no further conversion is necessary

5. Chack the result: $11000 = 2^4 + 2^3 = 24$

Multiplication - example

Perform the following operation in binary number system using floating point representation : 4.25 * 2.5

1. Transform the binary form of the numbers to normal form:

   $4.25 = 100.01 \rightarrow 0.10001*2^3$

   $2.5 = 10.1 \rightarrow 0.101*2^2$

2. Perform the operation:

$$
\begin{array}{r}
0.10001 * 0.101 \\
\hline
10001 \\
00000 \\
10001 \\
\hline
0.01010101
\end{array}
$$

Multiplication - example

3. The obtained mantissa: 0.01010101

   The characteristics: $k_1 + k_2 = 3 + 2 = 5$

   The result: $0.01010101*2^5$

4. Trensform the result to normal form:

   $0.01010101*2^5 = 0.1010101*2^4$

5. Chack the result: $1010.101 = 2^3 + 2^1 + 2^{-1} + 2^{-3} = 10.625$