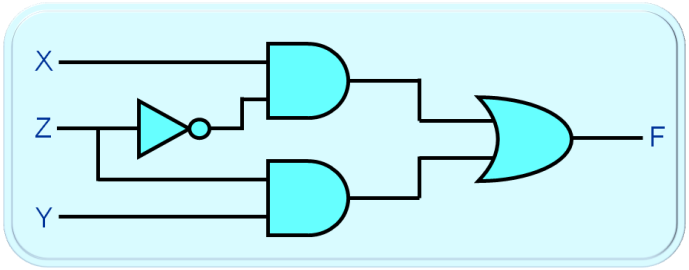

Binary Prerequisite

for Subjects

Computer Architectures

&

Logic Systems and Processors

Richard Šusta

Department of Control Engineering

CTU-FEE in Prague

Version 2.0 from September 25, 2023

Author:	Richard Susta, richard@susta.cz , http://susta.cz/
Figures:	* Figure 10 - ENIAC Electronic Numerical Integrator and Computer source Wikipedia, * the other figures, Richard Susta
Publisher:	Department of Control Engineering CTU-FEE in Prague, Technicka 2, 166 00 Prague 6 http://dce.fel.cvut.cz/
Datum of issue:	September, 2023
Length:	35 pages

Homepage of the document:

https://dcenet.fel.cvut.cz/edu/fpga/guides.aspx

Obsah

List of Figures

List of table

1 Summary of Prerequisite

A prerequisite is the minimum knowledge required to pass a course or subject that must be met before being allowed to study it. [Glossary of foreign words]

Overview

The prerequisite content could be well-known from programming courses. We will present the coding of binary integers as unsigned and signed numbers and explain their overflows during additions or subtractions. Furthermore, we will describe the logical and arithmetic shifts essential in logic circuits. In languages C, C #, and Java, they are partially included as shift operators << and >>.

Finally, we will briefly discuss the hexadecimal notation, necessary BCD coding, and ASCII characters.

Why was it created?

Many have certainly encountered binary numbers during their studies or have studied the subject themselves, but others have only experienced them fleetingly. Back in the day, when lectures started from the basics, the more knowledgeable students wrote in the course evaluation survey that they were pretty bored in the first few classes. After responding to their comments, the lecture focused more quickly on more interesting issues, but the less knowledgeable were blamed for not understanding the introductory passages.

I have written a prerequisite here to standardize the input knowledge to accommodate everyone. I have included only light terms. The complexities were left in the lectures.

How should we study the prerequisite?

Be sure to read the full text. If you think you understand a passage, don't skip it, but skim it to see if you can discover any new insights. However, slow down if you come across less familiar concepts or are not 100% sure about something, and study the topic carefully, including the procedures in the examples.

Language note

The textbook was translated from the original Czech language by a DeepL and then manually corrected with the aid of the Grammerly checker. Readers may encounter some sentence that escapes proofing. They will be purified in the following versions.

We do not use LaTeX. DeepL does not support its documents. We selected the MS Word editor, which does not allow inserting some cases of cross-references by a simple number. Some of them can also look clumsy.

History of the text

The prerequisite initially belonged to the APOLOS textbook, the name of which was created by combining the abbreviations of the course Computer Architecture (APO) and Logic Circuits and Systems (LOS), which was the predecessor of today's Logic Processor Systems (LSP). It contained a very brief overview of logic and binary numbers.

For the LOS, later LSP, I created other textbooks. In 2023, I completed the text Logic Circuits on FPGAs, greatly expanding the APOLOS passage on logic. I then removed the treatment of binary numbers from APOLOS and, after minor adaptations, saved it as a Binary Prerequisite.

Follow-up textbook for Logic Systems and Processors

Computer Architecture has an excellent textbook, "David A. Patterson and John L. Hennessy. 2013. Computer Organization and Design".

There are many materials on logic, but they often refer to older design techniques. Nothing comprehensive has been found that comprehensively summarizes the knowledge needed to design circuits correctly. And, after all, novice designers may find it more difficult to distinguish what is still valid today, what designs can still be applied after minor modifications, and what rampages are no longer allowed on modern means, even though they were once trendy and recommended, and so there are numerous diagrams of them on the web.

The binary prerequisite you are now reading is the entry portal.

The textbook **Logic Circuits on FPGAs** covers the main logic structures and principles. It contains general logic circuit knowledge but without descriptions in HDL languages. Schematics explain everything; for some, the circuit versions are given when implemented on FPGAs (Field Programmable Gate Arrays). There are widespread universal chips that offer a cheaper way for small series and a good solution for debugging monolithic integrated circuits' prototypes.

The textbook "Logic Circuits on FPGAs" serves as a springboard for design, which we can write in any HDL (Hardware Description Language). In our Control Engineering Department, we have chosen VHDL, which we consider more convenient for beginners. It is the focus of our other scripts, still under development, "**Circuit Design in VHDL 2008 for C Programmers**".

If somebody chooses to continue with Verilog or SystemVerilog, one will find several scholars from other authors.

2 Integers expressed in binary system

Maybe you have already heard somewhere the following joke or some of its modifications:

After a car accident, a programmer signed me the compensation of 1000 €.
He paid me ten euros with the note that he gave me two euros extra.

Its punchline is based on the fact that 1000 in a binary system is always decimal 8. It may or may not. The value depends on the method of coding numbers and the bit length of the numbers. Computers use different lengths of binary numbers, but usually only multiples of 8 bits (byte length). The lowest bit is located on the right, while the highest bit is on the left.

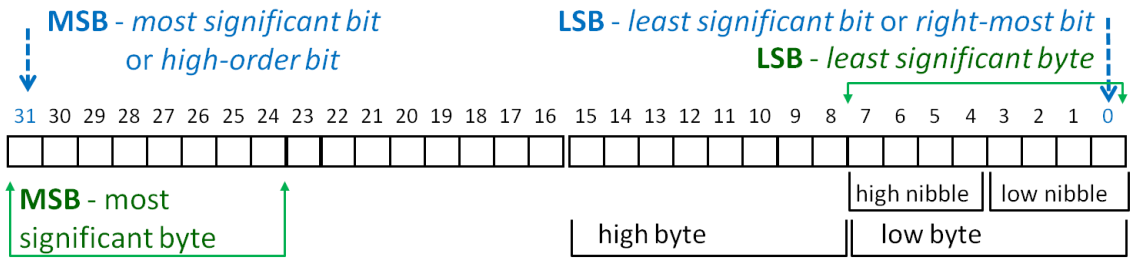

Figure 1 - Byte, bit, MSB, LSB

The word "**bit**" originally meant a small quantity of food. Four bits are sometimes called "***nibble***" (a negligible quantity of food). The size of 8 bits is known as **byte,** originating from a deliberate respelling of *bite* (a small amount of food). The size of 8 bits is also called "octet".

In logic circuits, a binary number may have an arbitrary positive length, i.e., a length greater than zero. There is no limit to the entire number of bits. The lowest bit can lie to the right (as in the picture above) and left. However, even circuits prefer the classical computer arrangement with the lowest bit to the right. The abbreviations mark the order of bits:

MSB "most significant bit" or "high-order bit". MSB is also used for specifying the ordering of bytes as "most significant byte".

LSB has the opposite meaning: "least significant bit" or "right-most bit". LSB is again used for specifying the ordering of bytes as "least significant byte".

We must distinguish from the context of a text whether MSB and LSB refer to a bit or a byte.

A length "**word**" indicates the native bit length of a processor. 32bit processors have "word" length 32-bits, 64-bit processors 64-bits. Word is not always and everywhere a 16-bit binary number, as it is sometimes mentioned mistakenly[1]. For example, "Apollo Guidance Computers" used in the flight to the Moon have 15-bit word.

The binary number is merely a sequence of 1 and 0, and its decimal value can be decided only by the selected method's specifications for encoding the decimal numbers. In the following text, we analyze the most frequently used ways.

[1] An exception of word-width can be found in industrial programming languages for PLCs (Programmable Logic Controllers), where the word is defined by standard IEC 1131-3. It introduces WORD type as a 16-bit length. Derived term DWORD (double word) defines 32-bit type and LWORD (long word) 64-bit type. The industrial standard IEC 1131-3, however, relates only to PLCs, it does not apply elsewhere.

2.1 Unsigned binary

By the whole name "binary encoded unsigned integers", **unsigned** binary is the base for binary numbers. Its principle is based on the mathematical fact that the sum of all previous members of 2^N series is always less by one than the following member. For example, the sum of the first four members of the series $2^0+2^1+2^2+2^3 = 1+2+4+8 = 15 = 2^4-1$. In general:

$$2^n - 1 = \sum_{k=0}^{n-1} 2^k \qquad (1)$$

We can express any nonnegative integer as the sum of selected members of 2^N series. If a member of the relevant powers is used, we write bit 1, otherwise 0. String $x \approx b_{m-1}\, b_{m-2} \ldots b_1\, b_0$ of m-bits is called a ***binary encoded unsigned integer***, starting now an **unsigned binary**, and it has a value:

$$x = \sum_{k=0}^{m-1} b_k 2^k \qquad (2)$$

For example, if we take string 1100100 as unsigned binary, we receive the decimal value 100:

$\mathbf{1}*2^6$	$+\ \mathbf{1}*2^5$	$+\ \mathbf{0}*2^4$	$+\ \mathbf{0}*2^3$	$+\ \mathbf{1}*2^2$	$+\ \mathbf{0}*2^1$	$+\ \mathbf{0}*2^0$	=
64	**+ 32**			**+ 4**			**=100**

The equation (1) ensures that there is exactly one combination of some members of 2^N series, whose sum gives the number, and each series member occurs in the sum at the most one time. In other words, the coding has bijective property, i.e., correspondence one-to-one.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

2.1.1 Changing bit-width of numbers

Bits '1' only determine the value of an unsigned binary number. We can insert any number of zeros before if without changing its value. For example, unsigned binaries 1100100, 01100100, 001100100, 0001100100, and so forth have the exact decimal value of 100. Here, we assume unlimited bit length. In practice, the width of binary numbers is limited, so we can add only as many zeroes to fit in a given limit.

2.1.2 Logical shifts

The operation of logical left shift appends bit 0 after the binary number, i.e., to its right side. For example, unsigned binary 101, corresponding to decimal 5 (2^2+2^0), changes by left shift to 1010 which has double decimal value, i.e., 10. Each 1-bit was moved below the next member of 2^N series. Similarly, 10100 with appended two 0 bits has a quadruple decimal value, i.e., 20, and 101000 is eight times of the original value, i.e. 40.

		2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1
8*5=**40**	=32+8	1	0	1	0	0	0
4*5=**20**	=16+4		1	0	1	0	0
2*5 = **10**	=8+2			1	0	1	0
5	=4+1				1	0	1

If we have limited bit width, then the value of a number is doubled by logical left shifts as long as the leftmost bit in 1 reaches the end of storage for our number.

For example, if we have unsigned binary 101 stored in 8 bits, i.e., as 00000101, then its value is always doubled after its first 5 logical left shifts, i.e., to the value 10100000, which corresponds to the decimal number 5 * 25 = 5 * 32 = 160. Another logical shift left gives 01000000, which is decimal 64. The highest bit 1 was lost due to the limited 8-bit width. An arithmetic overflow occurred. We will discuss it more in Chapter 2.1.5.

Programming languages based on the C language defines << [2] followed by the length of the shift in bits. If we have variable byte x = 5; (in C language as *unsigned char x=5;*) then:
2 * x == (x << 1) , or 4*x == (x << 2), and so on up 32*x == (x << 5).

The logical right shift corresponds to the operation of integer dividing by 2, since bits 1 are shifted to previous members of 2^N series. The integer division gives a result and a remainder. If we shift unsigned binary 101 by one bit to the right, the result is 10, decimal 2. The lowest bit 1 has fallen out from 101 is the remainder.

		2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	
5	=4+1				**1**	**0**	**1**	
5/2 = **2**	=2					**1**	**0**	-> 1
the remainder 1								

In C language, operator >> only partially implements logical left shift because the operator does not give the remainder. For variable byte x = 5; it can be used only as an integer dividing by 2. It holds: (x% 2) == 1 , (x >> 1)==2 and x / 2 == 2.

Programming languages often translate integer multiplications and divisions by constants equal to 2^N powers with the aid of shifts because they are high-speed operations.

2.1.3 Conversion of unsigned binary to decimal number

Method 1: The decimal value of a binary string taken as an unsigned binary is equal directly to the sum of the corresponding member of 2^N series, where N is the number of the specific bit. If we have a binary string X = 10011, which has 1-bit on the 4th, 1st, and 0th position, then we can determine its value as the sum of corresponding members of the series:

$$X=10011 \rightarrow 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19$$

If the binary string is longer and with more 1-bits, such as Q = 11111110110, then its conversion by the sum would be challenging. Our 11bit Q has mostly 1-bits:

bit	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Q	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**

We can shorten the calculation by the property given in equation (1) that the value of the following member of 2^N series is always greater by 1 than the sum of all previous members. So we know that:

$$2^{11}-1 = 2048-1 = 2047 = 2^{10}+2^9+2^8+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$$

[2] In language C ++, bit shift operators << and >> are usually overloaded by some includes (as iostream.h) to reading from and writing to data streams but they still behave as shifts for number arguments.

If we compare 2047 with our Q, then Q corresponds to nearly the same sequence of 2^N series members, in which two members 2^3 and 2^0 are only missing. It means

$$Q = 11111110110 \rightarrow 2047 - 2^3 - 2^0 = 2047-8-1 = 2038$$

Method 2: We can also use logical left shift operations and calculate the value by a polynomial Horner scheme (see this name in Wikipedia).

We begin with the highest bit. We take the bit and write it as a result, 1 or 0. If there is the next bit, we multiply the result by 2 or add the value of this next bit (1 or 0). We repeat the process until the lowest bit is added.

10011

1 → 1*2+0=2 → 2*2+**0**=4 → 4*2+**1**=9 → 9*2+**1**=19

Another example, in abbreviated notation:

1111111**0**11**0**

1 → 2+**1**=3 → 6+**1**=7 → 14+**1**=15 → 30+**1**=31 → 62+**1**=63 → 126+**1**=127

→ 254+**0**=254 → 508+**1** → 1018+**1** = 1019 → 2038+**0**=2038

We cannot recommend method 2 for hand calculations based on our experience. The technique alternates operations multiplication and addition, so it is not entirely mechanical. We can easily make a numerical error. However, the method is very suitable for the algorithm that converts unsigned binary to BCD numbers, Chapter 2.5.2, page 27.

2.1.4 Conversion of decimal number to unsigned binary

For simple conversions, we can apply either repeated subtractions or division by 2

2.1.4.1 Repeated subtractions

First, we found the higher member of 2^N series less than the converted decimal. For example, if we have 35, we select $2^5 = 32$. We begin the subtractions from it.

decimal number	subtracted member	subtracted	binary result	
35	-32	3	**1**	MSB
3	-16	*no*	**0**	
3	-8	*no*	**0**	
3	-4	*no*	**0**	
3	-2	1	**1**	
1	-1	0 (end)	**1**	LSB

If the difference is positive, we write 1 bit and subtract the member of 2^N series. Otherwise, we write bit 0. Then, we try to lower member of the series. We repeat until we obtain 0. The result of the conversion of decimal number 35 is unsigned binary 100011.

Repeated subtractions require knowledge of 2^N series. We easily learn its several beginning members, but repeated divisions are more comfortable converting bigger decimals.

2.1.4.2 Repeated divisions by 2

The method is derived from logical right shifts, Chapter 2.1.2 on page 8. We divide the given decimal number by 2 until the quotient becomes zero. We write down the reminders after integer divisions from the least significant bit (LSB) to the most significant bit (MSB).

The algorithm ends after obtaining the quotient 0. If we convert a decimal number greater than 0, then the remainder of the last division is always 1, which is the leftmost bit of the obtained binary result.

35 / 2 = 17 *remainder of integer division* **1** - the least significant bit

17 / 2 = 8 *remainder of integer division* **1**

8 / 2 = 4 *remainder of integer division* **0**

4 / 2 = 2 *remainder of integer division* **0**

2 / 2 = 1 *remainder of integer division* **0**

1 / 2 = 0 *remainder of integer division* **1** - the most significant bit

We can write the algorithm more briefly. We divide a decimal by 2 and retrospectively determine remainders from intermediate results. Odd results had remainders 1.

For example, we convert the decimal number 1000 to unsigned binary. In the next line, symbol → indicates that the right number was derived as a quotient of dividing the left number by 2:

1000 → **500** → **250** → **125** → **62** → **31** → **15** → 7 → 3 → 1 → 0

If we write the odd numbers as bits 1 and 0, we get unsigned binary 1111101000 . We lined up bits from the lowest, i.e., in the reverse order of the row of the numbers.

It would not mind if we included the last → 0. In that case, we obtained a binary number 01111101000 with the same value, see paragraph 2.1.1.

2.1.5 Arithmetic overflow during additions and subtractions

Computers and digital circuits always store a finite number of bits. If we continue adding 1, the number eventually reaches its maximum value. For unsigned binary, the maximum contains only 1 bits. The used arithmetic representation gives their count.

	Carry	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
254		1	1	1	1	1	1	1	0
+1									1
255		1	1	1	1	1	1	1	1
+1									1
0	1	0	0	0	0	0	0	0	0
+1									1
1		0	0	0	0	0	0	0	1

Table 1 - Adding +1 to 8bit unsigned binary

The maximum unsigned 8-bit binary number is 11111111, representing decimal 255. If we add +1 to it, we get the unsigned binary=100000000, which correctly corresponds to decimal 2^8 = 256, but it has nine bits. In 8-bit binaries, we can save only its lower eight 0 bits. The highest bit must be thrown away, so our result equals 0.

The removed highest bit is called **Carry**; it carries value to a higher order. It announces an arithmetic overflow of unsigned binary numbers, i.e., exceeding the maximum value for the given bit length.

When subtracting 1, the overflow can also occur. We can imagine the subtraction as a progression from the bottom up in Table 1. Then, operation 0 minus 1 gives here decimal 255 as its result. In logic circuits, the overflow of the opposite direction from 0 to the maximum is sometimes called **Borrow** because it borrows value from a higher order.[3] However, it is frequently also called Carry.

If we subtract one from zero, the overflow always occurs, and the binary result is filled by bits 1. It is the maximum value of unsigned binary arithmetics. The bit width only determines the decimal value of this wrong result of the subtraction 0-1. For example:

for 8 bit unsigned binary, $0\text{-}1 = 2^8\text{-}1 = 255$,

for 9 bit unsigned binary, $0\text{-}1 = 2^9\text{-}1 = 511$

for 16 bit unsigned binary, $0\text{-}1 = 2^{16}\text{-}1 = 65535$, and so on.

The result of the subtraction 0-1 in decimal counting is -1. If we want to know its unsigned value, we add 2^m, where m represents bit width of binaries. When the result is less than zero, we add 2^m until we get a positive number. If the result is greater or equal to 2^m, then we subtract 2^m.

Question 1: In 4-bit unsigned binary arithmetic, what is the decimal value of the result for two decimal numbers when adding them 14 + 4 and when subtracting them 4-14?

Answer: We evaluate **14+4**=18. The result is over 2^4=**16**, so we correct it by 18-**16**=**2**.

We evaluate **4-14** = -10. The result is less than 0, so we correct it by -10+**16** = **6**.

We can imagine the previous calculations on the wheel with numbers; see Figure 2. Addition operations correspond to rotating the wheel counterclockwise and subtractions to turning the wheel clockwise. A count of wheel cogs, for which the wheel turns, is determined by the number that we add or subtract. The figure shows that the number 14 is about 4 positions counterclockwise from the number 2 (14 + 4 = 2), and number 4 is about 14 positions clockwise from number 6 (4-14 = 6).

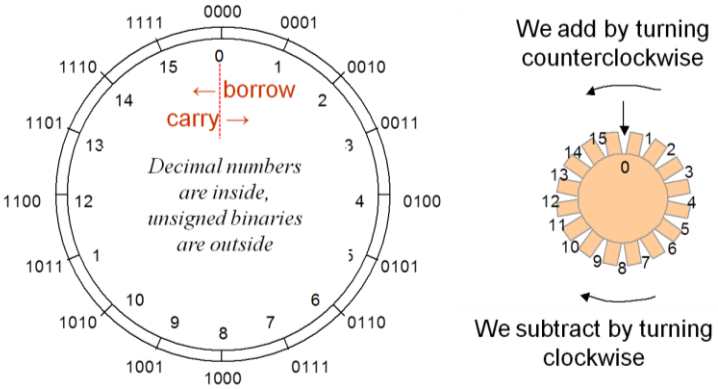

Figure 2 - Adding and subtracting unsigned binary

[3] Most processors do not distinguish overflow directions and their ALUs generate Carry in the both cases. Whether it was a Carry or a Borrow we can find out only according to executed assembler instruction. Addition - Carry, subtraction - Borrow.

Question 2: In 8-bit unsigned binary arithmetic, what is the decimal result of the addition of two decimal numbers 200 and 100?

Answer: We add numbers, 200+100 = 300. The result is over 2^8 = **256**. We subtract correction 2^8: 300-**256**=44. The result is 44.

Question 3: In 10-bit unsigned binary arithmetic, what is the decimal result of the subtraction of two decimal number 1000-1500?

Answer: We evaluate 1000-1500 = -500. The result is less than 0, so we add 2^{10} = **1024**, so -500+**1024**=524. The result is 524.

Question 4: In 5-bit unsigned binary arithmetic, what is the decimal result of the addition of two decimal numbers 10 a 20?

Answer: We add 10+20=30. The result is positive and less than 2^5=32. No correction is required.

2.2 Signed integers in two's complement

For integers with a sign, several different codes exist, of which the most used is two's complement based on the arithmetic overflow, Chapter 2.1.5.

If we have unsigned binary x stored in m-bits, we can create its **one's complement** χ by negating all its bits. The sum $x+\chi$ is unsigned binary with all bits in 1 because χ has bits 1 in such positions where x has bits 0.

The sum $x+\chi = 2^m-1$ is the maximum unsigned binary. If we add +1 to χ, we obtain $x+(\chi+1)=2^m$. The result 2^m has bit length $m+1$. We can store only m lower bits that are all equal to 0. For m-bit unsigned binary, therefore, the following holds $x+(\chi+1)=0$.

For this property, $(\chi+1)$ is called **two's complement of x.**

For example, if we have 4-bit unsigned binary, decimal 4 is coded as 0100. Its one's complement (the negation of all its bits) is 1011 (χ). If we add +1 (binary 0001) to χ, we get 1100 (χ+1); it is two's complement of 0100. The sum 0100+1100 =10000. The result 10000 (taken as unsigned binary) has a decimal value 16, but 10000 has 5-bit width. Into 4-bit binary, we can store only its lower bits 0. Thus, the result equals to 0000. The arithmetic overflow has occurred.

We define signed integers in **two's complement,** also known as **signed binary,** as**:**

- The negation of an unsigned binary number is its two's complement,
- Further, we specify that a m-bit binary, which has bit 1 in its most significant bit (i.e., in the bit with weight 2^{m-1}), encodes a negative decimal number.

For 4-bit arithmetic, signed binary are shown on the next page in Figure 3 to the right. Signed binary 1000, with decimal value -8, has a particular position. Its two's complement also exists, but it is a binary 1000 itself. Because 1000 has 1 in its upper bit, it represents the negative decimal number -8, to which no positive decimal counterpart exists in 4-bit arithmetic. This asymmetry is only one drawback of signed binaries (signed integers in two's complement).

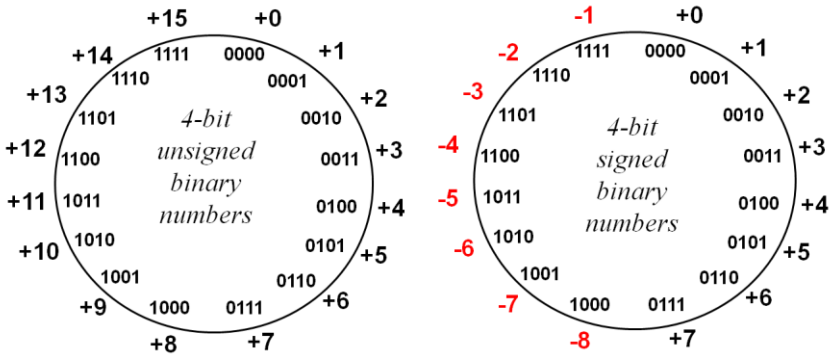

Figure 3 - 4-bit unsigned and signed binaries

Otherwise, signed binaries provide only benefits. We calculate their additions and subtractions in the same way as unsigned binaries. Therefore, we can use the same computing unit for both representations of numbers. It depends only on us whether we interpret the results of the operations as unsigned or signed binaries. Moreover, the coding of positive integers is the same. Two's complement is calculated only for negative integers.

For the advantages mentioned above, signed binaries (signed integers in two's complement) are the ultimate way for storing signed integer numbers (in C, signed int type).

2.2.1 Important properties

Signed binaries have significant properties worth remembering.

- Number 0 is always coded by all bits 0.
- If we denote the bit length as **m**, such binary encode decimals in the range from 2^{m-1} to 2^{m-1} -1, the other numbers are out of the range.
  For example, for m = 4, the range is from -2^3 to 2^3 - 1, thus from -8 to 7.
- A signed binary with 1 followed by m-1 bits 0 is always the least number, and its decimal value equals -2^{m-1}. This number is also the only anomaly; its positive counterpart does not exist in a given bit length. For example, if we have 8-bit signed binaries, their least binary is 10000000 with decimal value $-2^{8-1} = -2^7 = -128$.
- Signed binary with 0 followed by m-1 bits 1 is always the bigger number. Its decimal value equals to 2^{m-1} -1. For example, if we have 8-bit signed binaries, their greatest binary is 01111111 and has decimal value is 2^{8-1} -1 = 2^7 -1 = 127. *Note: Its counterpart, decimal -127, is coded as 10000001. It is greater by 1 than -128.*
- Signed binary with all bits 1, i.e., m bits 1, always equals to decimal -1. For example, if we have 8-bit signed binaries, then -1 is coded as 11111111. *Note: Decimal -2 is stored as 11111110 because it is less by one than -1.*

2.2.2 Arithmetic negation by two's complement

Let us have a signed binary (signed integer in two's complement) of known bit length m. We find its negative number (arithmetic negation) by **algorithm of two's complement**:

a) we logically negate all its bits (one's complement),
b) then, we add 1 to the result to obtain two's complement of the original binary.

Example 1: Calculate arithmetic negation of 8-bit signed binary 01100100, it has decimal value 100.

Answer: We create one's complement by negating all its bits 01100100→10011011. Finally, we add 1 to it, i.e., 10011011+00000001 = **10011100**.

Example 2: Calculate arithmetic negation of 10-bit signed binary 1000000000, (decimal value -512).

Answer: The example is a trick question. The correct answer is: "**we cannot",** see Important properties above.

2.2.3 Conversion of decimal number to signed binary

To convert a decimal number, we must always know the bit length of the desired signed binary number. We again denote the bit length as m. For bit length m, we can convert only integers that satisfy the range of signed binaries from 2^{m-1} to 2^{m-1} -1, paragraph 2.2.1

- We convert positive integers as unsigned binaries.
- We convert negative integers by any of the following methods, in which we denote an entered negative decimal number as -x
    a) We convert the absolute value -x, i.e. | -x |, as an unsigned integer and create its two's complement. The disadvantage of this method is the necessity of binary adding 1 when calculating the two's complement.
    b) To avoid binary addition, we can convert the absolute value of the decimal number reduced by 1, i.e., |-x|-1, to unsigned binary. Its one's complement (negation of all its bits) equals two's complement of |-x|.
    c) Alternatively, we can convert (2^m - x) to m-bit unsigned binary. If we take the result as m-bit signed binary, it is equal to -x. This method uses the direct how signed binaries in two's complement have been defined.

  How do we verify that we remember a method? We need to know one decimal and its correct conversion to signed binary. For example, decimal -1 is always converted to all bits 1. First, we try to convert our selected known number by any of the methods above. If we get the correct result, we remember the calculation process correctly.

  Remember: We can always verify our conversion of -x by the addition: -x+x=0

Example: Convert decimal -12 to 8-bit signed binary:

- Calculation by a) First, we convert absolute value |-12| = 12 to 8-bit unsigned binary as 00001100. We create one's complement of the result by the negation of its bits as 11110011. Then, we increment it by 1, so 11110011+00000001 = **11110100.**
- Calculation by b): |-12|-1 = 11. Decimal 11 as 8-bit unsigned binary is 00001011. Then, we perform one's complement of the result: 00001011 → **11110100.**
- Calculation by c): 2^8-12 = 256-12 = 244. Decimal 244 converted to 8-bit unsigned binary is **11110100**. This number, taken as 8-bit signed binary, has its decimal value -12.

2.2.4 Conversion of signed binary to decimal

To convert binary numbers, we must again know the bit length of the desired signed binary representation. We again denoted it as m.

- Signed binary numbers with 0 in their most significant bit (MSB) are converted by the same ways as unsigned binaries.
- Negative signed binary numbers, i.e. with MSB equal to 1, can be converted by one of the following ways, which are reversed versions of the previous methods in 2.2.3.
  - First, we calculate two's complement of the signed binary and converted it as unsigned binary to number x. Finally, we change its sign to minus, so -x.
  - We can perform only one's complement (logical negation of bits) of our signed binary. Then, we convert the result as unsigned binary to decimal that we denote as y. The required result -x is given by: -x = -y-1.
  - Alternatively, we can convert the signed binary as an unsigned binary to the number we denote as z. The required result -x is given by $-x = z-2^{m}$.

  Example: Convert 9-bit signed binary 000111000 to a decimal number.

  Solution: The most significant bit is 0. Therefore, we convert as an unsigned binary. For example, we apply adding of the weights of its 1-bits: $2^5+2^4+2^3 = 32+16+8=56$.

  Example: Convert 8-bit signed binary 11001100 to a decimal number.

  Solution: MSB is 1. Thus, we apply the methods for negative binaries:

  - Calculation by a) We evaluate two's complement of signed binary 11001100: 00110011+00000001=00110100. Then, we convert 00110100 as an unsigned binary to 52. The searched result is its negation, so **-52.**
  - Calculation by b): We evaluate one's complement of 11001100 → 00110011. Then, we convert it as unsigned binary: 00110011 → 51. The searched result is -51-1 = **-52.**
  - Calculation by c): We convert signed binary 11001100 as unsigned, e.g., by adding weights of 1-bits: 128+64+8+4=204. The result is $204-2^8= 204-256=$**-52**.

2.2.5 Change of bit length - sign extension

In front of an unsigned binary number, we can add 0 bits without changing its value, see 2.1.1. To preserve the value of a signed binary number, we must utilize the sign extension that maintains the most significant bit, the sign of our number.

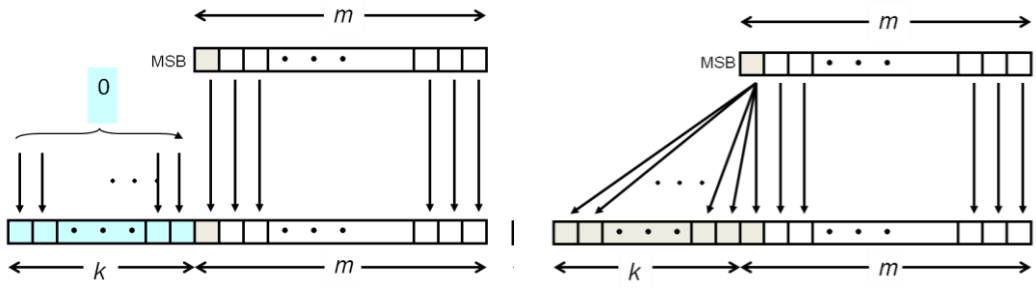

Figure 4 - Signed extension

Figure 4 shows the differences between binary numbers. While we always insert bits 0 for unsigned binary, we must copy the most significant bit (MSB) for signed binary.

Conversely, if we decrease the length of a binary number, we can remove all leftmost bits 0 of an unsigned binary. For a signed binary, we can remove either the most significant bits 0 and 1 in the case that we preserve the value of the original most significant bit.

Example 1: Extend 4-bit signed binary 0111 to 8 bits.

Solution: The most significant bit 0 is copied into inserted bits, so the result is 0000 0111.

Example 2: Extend 8-bit signed binary 1000 0010 to 16 bits.

Solution: We again copy MSB=1 into inserted bits, so the result is 1111 1111 1000 0010.

Example 3: What is the shortest possible bit length for 8-bit signed binary 1110 0100?

Solution: We can remove only 2 bits 1. The shortest length is 6 bits, so 10 0100.

Example 4: What is the shortest possible bit length for 8-bit signed binary 0000 0100?

Solution: The shortest length is 4 bits, so 0100.

Note: The type of a binary number determines whether we must perform its sign extension. Machine codes of processors contain different instructions for loading signed and unsigned data from smaller sizes to larger one. For example, RISC V CPU loads an unsigned byte to a 32-bit register by LBU instruction, but it has LB instruction that performs sign extension for byte containing a signed binary in two's complement. Processors of x86 family use MOV instruction and MOVSX for the same purposes. Compilers of higher languages select machine code instructions according to types of variables.

2.2.6 Logical and arithmetic shifts

Preserving sign bit requires different shift operations with binary numbers, so we use two shift types. Unsigned binaries require logical shifts and signed binaries use arithmetic shifts that respect the sign bits.

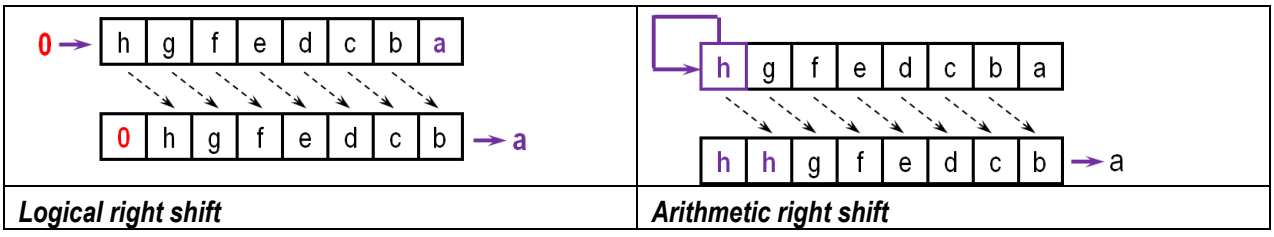

Figure 5 - Logical and arithmetic right shifts

Figure 5 depicts right shifts for 8-bit binary "hgfedcba" in which ´a´ is LSB and ´h´ is MSB. If a binary represents the unsigned format, we perform right shifts by inserting 0-bits. If we have a signed binary, we utilize arithmetic right shifts, in which the highest bit (MSB) remains firmly in its place, here ´h´.

For example, 8-bit 11101011 changes after the logical right shift to 01110101, while it changes to 11110101 after arithmetic right shift. In contrast for 8-bit string 01101010, whose MSB=0, the both logical and arithmetic shifts give the same result 00110101.

Input	Decimal value as	Left shift	After the shift	Decimal value as
11101011	unsigned= 235	logical	01110101	unsigned= 117
	signed= -21	arithmetic	11110101	signed= -11
01101010	unsigned= 106	logical	00110101	unsigned= 53
	signed= 106	arithmetic	00110101	signed = 53

From the table above, we can see that the logical right shift is suitable for unsigned binaries, for which corresponds to dividing by 2. The result is its quotient and the lowest bit (LSB) lost by the shift is its remainder.

Arithmetic right shifts are necessary for signed binaries to preserve their sign bits. For a positive signed binary, the result is the quotient of dividing by 2, and the remainder is given by the lowest lost bit. For a negative signed binary, the quotient is rounded towards the lower numbers, e.g. -21/2 = floor(-10.5) = -11, where floor() denotes rounding down.[4] Arithmetic right shifts interpreted as a division by 2 give contradictory results, for example -1/2 = -1.

If we replace integer division by 2 with the aid of arithmetic right shifts, we must sometimes correct the results of negative binary shifts to receive the expected numbers. The corrections are simple; we increment the results by 1 in selected cases.[5] Therefore, we should remember that the arithmetic right shift is not an exact analogy of integer division by 2 for signed binaries.

Left shifts of binary numbers are the same - the logical left shift is performed as arithmetic. Figure 6 shows the shifts for 8-bit binary "hgfedcba", in which ´a´ is LSB and ´h´ is MSB that is lost after left shifts.

If there is not arithmetic overflow, then a left shift corresponds to the multiplication by 2 for unsigned and signed binaries.

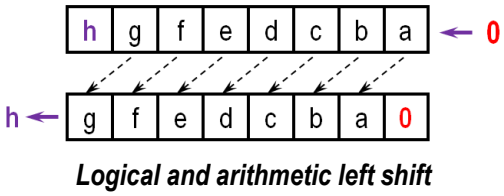

Figure 6 - Left shift of 8-bit binary

The signed and unsigned representations differ only when arithmetic overflow occurs during shifts. For an unsigned binary number, the arithmetic overflow occurs when the highest bit, lost

[4] In C, we have floor(...) function, floor(...) method in Java, and Math.Floor(...) method in C# .

[5] In general, processors can use the same arithmetic unit for unsigned and signed binary, which is the major advantage of these representations. Only in some cases, arithmetic operations with signed binaries require additional steps. Further, in a multiplication or division, when their operands are negative signed binary numbers, the result requires some corrections and negative signed binary numbers near zero have a lot of bits 1 and their multiplication could last too long. The processors can include a unit for the multiplications of negative signed binaries to speed up calculations, but frequently perform it with absolute values and adjust signs of results. Detailed descriptions are beyond the scope of this publication. For more information, search "Booth's algorithm" on Wikipedia.

during shifting, equals 1. However, signed arithmetic overflow occurs when the shift changes the value of MSB.

For example, if we have 8-bit signed binary 11101011 (decimal -21), then its first left shift gives the result 1101011**0** (-42) and the second 1010110**0** (-84). After performing third left shift, we obtain 0101100**0** (decimal 88). The overflow has occurred.

Another example: If we have 8-bit signed binary 00110010 (decimal 50), its first left shift result is 0110010**0** (decimal 100). The second shift gives 1100100**0**, which has as 8-bit signed binary decimal value -56; thus, we have the overflow.

Notice that we will have no overflow after the second shift if we take the same binary entry as an unsigned number, because its decimal value is 200. The overflow would appear here after its third left shift, resulting in 1001000**0** with the decimal value 144.

Note: The term of arithmetic overflow strictly depends on how we interpret a binary number. If a binary is taken as a common chain (vector) of bits with no specified numerical representation, then shifts just change positions of bits moving them to the left or the right. The shifts behave as an analogy of some conveyor belt that moves by one position, and the bit that lays at the end of the conveyor falls out.

2.2.7 Arithmetic overflow for addition and subtraction

In Chapter 2.1.5 on page 11, where we have discussed the addition and subtraction for unsigned binary, we have detected the overflow by Carry. However, the Carry has no practical meaning for signed binaries since it is frequently generated for them because they are based on the overflow of unsigned binaries, Chapter 2.2 on page 13.

For signed binary numbers, signed arithmetic overflow occurs when crossing the border between their largest and smallest numbers. For example, if we have 8-bit signed binaries, then their greatest number is decimal 127 coded as 0111 1111. When we increment it by 1, i.e. 0111 1111 + 0000 0001, then we obtain 1000 0000, i.e., their least number, decimal -128. We have a negative result of the sum of two positive numbers. Analogously, when subtracting 1 from -128, we receive the positive result 127.

	Overflow	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
126		0	1	1	1	1	1	1	0
+1		0	0	0	0	0	0	0	1
127	0	0	1	1	1	1	1	1	1
+1		0	0	0	0	0	0	0	1
-128	1	1	0	0	0	0	0	0	0
+1		0	0	0	0	0	0	0	1
-127	0	1	0	0	0	0	0	0	1

Table 2 - Arithmetic overflow for addition of 8-bit signed binaries

ALU of a processor detects overflow situations by a table of sign bits of operands and results. If the result is, of course, incorrect, flag Overflow is generated.

operand 1	operation	operand 2	result
positive	+	positive	negative
negative	+	negative	positive
negative	-	positive	positive
positive	-	negative	negative

Table 3 - Conditions for overflow of signed binaries

To be exact, ALU sets after each addition and subtraction of numbers, at least four basic arithmetic flags[6] to 0 or 1, to 0 or to 1, depending on the situation that has occurred

- Carry - carry or borrow from the most significant ALU bit position[7];
- Overflow - overflow for signed binary numbers;
- Sign - the result of the operation is negative;
- Zero - the result was zero.

Additions and subtractions of signed and unsigned binary numbers are performed the same way, so the majority of ALUs always assign all of these flags. Compiled machine instructions must test the proper flag, on which a result depends, according to the format considered as input operands.

2.3 Signed integer in straight binary and offset binary

Other coding methods exist for signed integers. For completeness, we mention two general methods: straight and offset binaries.

We can store a signed integer by encoding its absolute value as an unsigned binary and add a sign bit. This method is called a straight binary or sign or magnitude representation, abbreviated as sign-magnitude.

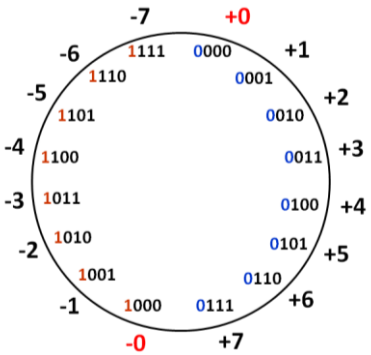

Figure 7 - Straight binary (Sign-magnitude)

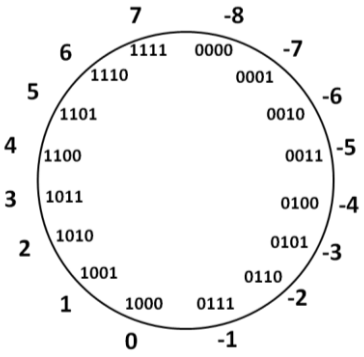

Figure 8 - Excess K with K=8

Figure 7 shows straight binaries for 4-bit length. In the code, there are two zeros, negative and positive. For example, if our result reaches zero during the iterations, we know how calculations

[6] Processors have flags stored in a special register called "flag register" or "status register" together with other flags. Many machine instructions influence these flags, not only arithmetic operations.

[7] ALUs set Carry also for shifts, but higher programming languages do not implement possibility to use their Carry results.

approach it. We can distinguish +0 or -0, which is essential in many numerical algorithms. The +0 and -0 are stored in float IEEE75 formats as unique values. Another advantage of this code is very fast arithmetic negation by only changing the highest bit.

Example: Encode decimal -15 as 8-bit straight binary.

Solution: 8-bit straight binary has 1 sign bit and 7 bits of its absolute value. We encode |-15| to 7 bits as unsigned binary to 000 1111 and we add sign bit, here 1, in front of it, because -15 is negative. The result is 1000 1111.

The next popular encoding is **Excess-K** or **offset binary** or **biased representation**. We first convert signed integers to nonnegative numbers by adding a fixed constant K to them, which is chosen so that the results are always positive. Then, we encoded them as an unsigned binary number.

If we choose m-bit length binaries and a fixed constant K, we obtain the range of decimals that we can convert from $-K$ to 2^m-1-K.

For example, if we select 4-bit binaries and $K=8$, we have the range from -8 to 7, see Figure 8. We can select the range by any value of K. For example, if we take K=30, then 4-bit binaries have a range from -30 to -15.

Example: Encode -15 as 5-bit Excess-K binary **+16**.

Solution: -15 **+16** = 1. Decimal 1 as 5-bit unsigned binary is 00001.

Excess-K additive code performs arithmetic operations with (x + K) and (y + K) numbers instead of the numbers x and y. For example, the result of x + y is (x + y) + 2 * K. We must always correct it by subtracting K. The corrections of multiplications are more complicated, and we cannot perform division directly in this code.

Straight and K-excess codes are unsuitable for immediate calculations because conventional processors cannot work with them. However, they are used for transmissions, as internal codes, and for composed numbers.

They are also fundamental for floating point numbers encoded according to the **IEEE 754** standard for types float, double, and extended used by most modern computers. IEEE 754 stores mantissa in straight code and exponent in K-excess code[8].

[8] With numbers IEEE 754, we usually do not calculate directly, but before arithmetic operations are decomposed into mantissa and exponent, which are processed separately. Finally, the result is again composed. However, some operations can be performed directly with composed numbers. Saving mantissa in straight code allows quick arithmetic negation of a number, merely by changing one bit. You can also compare two IEEE 754 number directly in composed form, i.e., as quickly as two integer numbers.

2.4 Hexadecimal notation

Hexadecimal notation is a shorthand way of writing binary strings. We encode 4-bit groups as 4-bit unsigned binary. The values from 10 to 15 are replaced by letters from A to F to maintain a single-character representation of groups.

Example 1: Write binary vector 10100111 in hexadecimal notation.

Solution: We divide the vector into 4-bit groups 1010 0111, and we encode each group, so the result **A7**

Example 2: Write 11100110101011 in hexadecimal notation:

Solution: First, we divide the string into 4-bit groups from its LSB to 11 1001 1010 1011. We extend the leftmost group with 2 bit to 4 bits 0011 1001 1010 1011. We encode them as **39AB**

Example 3: Convert hexadecimal notation 1F to 6-bit binary string.

Solution: By direct conversion, we obtain 0001 1111. We take the lower 6 bits, so **01 1111**

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

The hexadecimal notation leads implicitly to the number of bit lengths divisible by 4, but it is possible to use it for other lengths if we add the bit length specification.

2.4.1 Hexadecimal number

A hexadecimal number means that the bit string written in hexadecimal notation is interpreted as an unsigned binary number. There are several different ways for specifying thus format. We show some of them for the values of A7 and 39AB from examples 1 and 2 above.

a) 0A7H , 39ABH Hexadecimal notation ends with the suffix H, and if it begins with a letter, we add the prefix 0 to highlight the numeric value.

b) 0xA7, 0x39AB We add prefix 0x in front of the number.

c) X"A7", X"39AB" Notation in VHDL language.

d) 16#A7, 16#39AB Notation in PostScript language.

..... and many other formats, see Wikipedia, keyword Hexadecimal

In programming languages, however, a hexadecimal constant (literal) can also be taken as a signed binary number if assigned to a variable of a signed type. The situation is demonstrated in the following C ++ code compiled so that the variable int has 4 bytes.[9]

[9] The size of int depends on a compiler. In 64-bit environment, it could be 8 bytes, i.e. 64 bits, but many compilers select here also 4 bytes, i.e. 32 bits, for backward compatibility of programs.

```
int intsize = sizeof(int);       // intsize = 4 (byty)
unsigned char uc = 0xFF;         // uc = 255
char sc = 0xFF;                  // sc = -1
unsigned short int usi = 0xFFFF; // usi = 65535
short int ssi = 0xFFFF;          // ssi = -1
unsigned int ui = 0xFFFFFFFF;    // ui = 4294967295
int si = 0xFFFFFFFF;             // si = -1
```

As the example shows, 0xFF constant may not always be equal to 255, see variable sc.

2.4.2 Numeral systems

Hexadecimal (also base 16, or hex) numbers are often introduced as a positional numeral system with a radix, or base, of 16. However, such definition directly induces that numbers are unsigned binaries, so we have had so far avoided.

$$x_{16} = \sum_{k=0}^{m-1} a_k 16^k ; \quad 16 > a_k \geq 0 \qquad (3)$$

We evaluate the value of hex number x_{16}=0xA7 as the sum $x_{16}=10*16^1+7*16^0 = 167$. From a mathematical point of view, we can select any integer greater than 1 as the radix. If we choose, for example, r = 10 then we get decimal numbers. If we denote the radix of a numeral system as **r**, then the equation gives number x_r as:

$$x_r = \sum_{k=0}^{m-1} a_k \mathbf{r}^k ; \quad \boldsymbol{r} > a_k \geq 0, \ \boldsymbol{r} > 1 \qquad (4)$$

The values a_k are numbers, but we write them by single characters in hex notation. The circuits and computers prefer numbers with radixes equal to $r=2^m$ because they allow fast conversions to binaries and efficient storage. The exponent m determines the length of a bit group coded by one character. The table below shows conventional systems:

Name of number	**Base (radix) r**	**Length of bit group**
Binary	2	1
Octal	8	3
Hexadecimal	16	4
Base32	32	5
Base64 nebo **Radix64**	64	6
Decimal	*10*	*- no possibility to convert by bit groups-*

Table 4 - Some conventional radixes for numeral systems

In the following paragraphs, we briefly summarize yet unlisted codes.

2.4.2.1 Octal numbers

If we select r=8 in equation (4), we obtain coding by 3-bit groups known as **octal** numeral system or **oct** for short.

For example, hexadecimal number 0xA7, with decimal value 167, is encoded as unsigned binary 1010 0111. If we split it from the right side to 3-bit groups as 10 100 111 and we encode each group as an unsigned binary number, then obtain octal number 247. Suffix Q is sometimes appended, i.e., 247Q, to emphasize octal encoding.

In the past, octals were widely used, mainly in telecommunications. Now, they are applied only rarely. In Unix versions, they remain in commands *chown* and *chmod* (abbreviation for *change owner* and *change mode*), whose arguments are octals (without suffixes Q).

2.4.2.2 Base32 and Base64

If we select radix r=32, we encode 5-bit groups to Base32. If we take radix r=64, then we encode 6-bit groups to Base64. We frequently encounter both encodings. They are suitable for efficient transmissions of long binary strings in text form as encryption of keys. Activation keys of programs are also often encoded as Base32 numbers.

Base32 and Base64 encoding are less transparent than hexadecimal numbers because they rarely represent numerical values according to the formula (4). Encoded bit groups with a length of 5, or respectively 6 bits, are not commensurable with the standard byte sizes of numbers, i.e., 1, 2, 4, or 8 bytes. We do not achieve efficient compression into text if we encode separate numbers with byte sizes. Therefore, all numbers are often joined into one long binary chain, i.e., one many-bit number encoded in a text string. After we decode the string to the original binary chain, we divide it into numbers.

We divide very long binary chains to bit groups, usually from the leftmost bit. Several code tables exist, put in place by manufacturers, so numbers a_k from equation (4) can be represented by different symbols. According to a selected table, we write characters for values from 0 to 31 for Base32, respectively, from 0 to 63 for Base64. The character tables are usually designed to avoid confusion with similar symbols, such as the lowercase letters l and numeral 1. Base32 only uses numbers and letters, and it is case-insensitive. Base64 needs more characters, so it is case-sensitive. An uppercase letter has a different value than its lowercase counterpart.

To the end of Base32 and Base64 codes, the padding is appended. The sequence of '=' indicates the length of the last group. It is also helpful as an end mark when the text result is stored in more lines. See Wikipedia, keys Base32 and Base64.

The comparison of some possible encoding to Base32 and Base64 with other codes:

Decimal number 1234567890

encoded as	encoded to text string
• decimal	`1234567890`
• (un)signed binary	`0100 1001 1001 0110 0000 0010 1101 0010`
• hexadecimal	`499602D2`
• octal	`11145401322`
its bit groups	`01 001 001 100 101 100 000 001 011 010 010`

as unsigned binary encoded in

• Base32 RFC4648	**`JGLAFUQ=`**
• *its bit groups*	`01001 00110 01011 00000 00101 10100 10+000`
• Base64 original	**`SZYC0g==`**
• -	`010010 011001 011000 000010 110100 10+0000`

2.5 BCD - Binary Coded Decimal

BCD encoding represents the oldest used method. Each its digit is encoded as a single unsigned binary number and stored in four bits, so we directly write a decimal number. For example, we store decimal 35 as 0011 0101 in BCD.

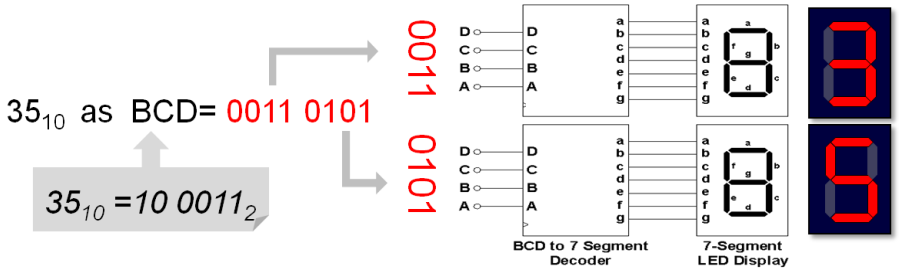

Figure 9 - BCD číslo 35

BCD coding offers the advantage of good readability for large binary numbers because we can directly see a decimal number from its binary encoding. Display devices preferably use BCD. For example, to display a number on the 7-segment display, we must first convert it to BCD, see Figure 9. Similarly, when we print numbers by calling printf () the function, it first converts numbers to BCDs and then writes characters of digits.

BCD contains 4-bit groups similar to hexadecimal numbers, but BCD does not use the whole range of 4-bit unsigned binaries from 0 to 15, but only the values from 0 to 9. If some 4-bit BCD group contains a value outside this range, then it is not a valid BCD number. When we encode a decimal 9876543210 to BCD number, we obtain 4 * 10 bytes, i.e. 40 bits:

Decimal number	9	8	7	6	5	4	3	2	1	0
BCD digit	**1001**	**1000**	**0111**	**0110**	**0101**	**0100**	**0011**	**0010**	**0001**	**0000**

The earlier computers used BCD numbers directly, precisely for their easy readability by human operators, such as the first electronic computer, ENIAC (Electronic Numerical Integrator and Computer) made in 1946, see Figure 10 [photograph from Wikipedia, key ENIAC].

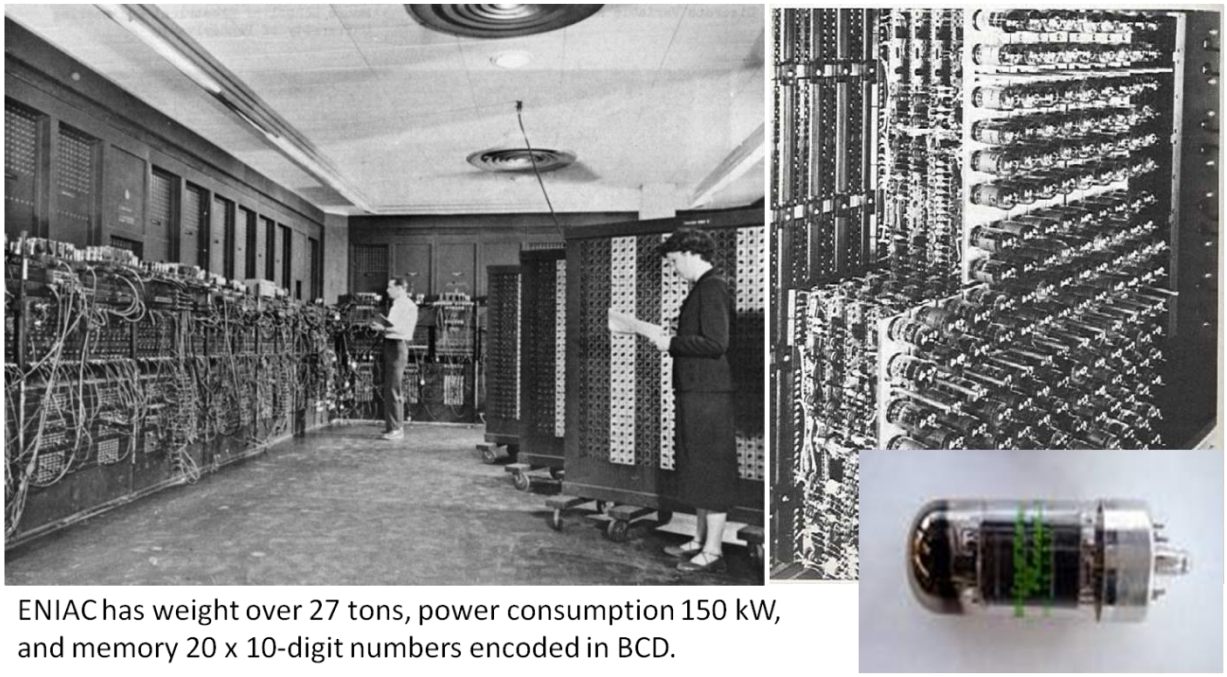

Figure 10 - ENIAC Electronic Numerical Integrator and Computer

Regarding storing BCD numbers in computer memories, there are two formats of BCD numbers. Unpacked BCD format stores each BCD digit in a single separate byte, which is suitable for numerical operations. Packed BCD format stores in one byte two BCD digits, and it is efficient for storing numbers. Microprocessors can usually perform arithmetic operations only with unpacked BCD numbers.

Today, direct counting with BCD numbers is rarely performed because it is about 2 to 3 times slower than binary numbers and requires more memory access. However, BCD is necessary for each printing of numbers.

We can convert a binary number to BCD by dividing by ten and counting remainders, but such an approach is unnecessarily slow. There is a much faster method based on the left shifts, previously mentioned as Horner scheme; see Method 2 in Chapter 2.1.3, beginning on page 9. Moreover, there is a method that can directly convert packed BCD numbers, which is suitable for parallel realization in hardware. For it, we need only the operation of BCD multiplication by 2.

2.5.1 How to multiply BCD by 2

When a BCD number contains digits from 0 to 4, we multiply them by 2 as unsigned binaries by the logical left shift. The values of the result will be in the valid BCD range from 0 to 8. The problem arises for BCD digits in the range from 5 to 9. After multiplying them by 2, we obtain the results from 10 to 18 outside the valid BCD range.

We can correct them by the **double daddle** algorithm. Before performing the left shift of a BCD, we add 3 to all BCD digits with values from 5 to 9.

Why are 3? It is half of the length range missing in BCD coding. BCD encoding utilizes only the values 0 to 9 of the full range of 4-bit unsigned binaries (i.e., from 0 to 15). BCD omits its 6 values from 10 to 15. Therefore, before we left shift a BCD, we added +3 to its BCD digits greater than 4 as their corrections. Then, these corrected digits will skip the missing 6 values during the following shift left (multiplication by 2). Thus, we obtain the required result.

BCD	Correction	Before left shift	After left shift
0000 0000 [0 | **0**]		0000 0000 [0 | 0]	0000 0000 [0 | **0**]
0000 0001 [0 | **1**]		0000 0001 [0 | 1]	0000 0010 [0 | **2**]
0000 0010 [0 | **2**]		0000 0010 [0 | 2]	0000 0100 [0 | **4**]
0000 0011 [0 | **3**]		0000 0011 [0 | 3]	0000 0110 [0 | **6**]
0000 0100 [0 | **4**]		0000 0100 [0 | 4]	0001 0100 [0 | **8**]
0000 0101 [0 | **5**]	+[0 | 3]	0000 1000 [0 | 8]	0001 0000 [**1** | **0**]
0000 0110 [0 | **6**]	+[0 | 3]	0000 1001 [0 | 9]	0001 0010 [**1** | **2**]
0000 0110 [0 | **7**]	+[0 | 3]	0000 1010 [0 | 10]	0001 0100 [**1** | **4**]
0000 0110 [0 | **8**]	+[0 | 3]	0000 1011 [0 | 11]	0001 0110 [**1** | **6**]
0000 0110 [0 | **9**]	+[0 | 3]	0000 1100 [0 | 12]	0001 1000 [**1** | **8**]

The correction can cause the temporary creation of invalid values for BCD digits, like 1010, 1011, and 1100, in the last rows of the table. Still, the logical left shift immediately converts them to valid BCD values.

2.5.2 Conversion of unsigned binary to BCD

We explain the algorithm by the example of converting 8-bit unsigned binary 01110011, hex 0x73, with decimal value 115.

Steps:

a) At the beginning of each step, we first examine all individual 4-bit BCD digits. If a digit is greater than the number 4 (0100), we add binary 3 (0011). We perform these additions separately for each corrected BCD digit, i.e., as the operation with an isolated 4-bit unsigned binary. Intermediate results of the correction can be over 9 (1001), but such invalid values are automatically corrected in the next step b).

b) Then, we perform the logical left shift of entire BCD number joined with converted unsigned binary, i.e., we shift all numbers as a single continuous chain of bits.

For 8-bit unsigned binary, we repeat the steps a) and b) by 8 times.

Operation	packed **BCD number**		**Unsigned binary**
Initialization	**[0 | 0 | 0]**	0000 0000 0000	**01110011**
after 1st joined left shift BCD and binary	**[0 |0 | 0]**	0000 0000 0000	**11100110**
after 2nd joined left shift BCD and binary	**[0 |0 | 1]**	0000 0000 0001	**11001100**
after 3rd t joined left shift BCD and binary	**[0 | 0 | 3]**	0000 0000 0011	**10011000**
after 4th t joined left shift BCD and binary	**[0 | 0 | 7]**	0000 0000 0111	**00110000**
BCD digit 0111 > 0100	+[0 | 0 | 3]	+0000 0000 0011	
result of the correction	[0 | 0 | 10]	0000 0000 1010	**00110000**
after 5th joined left shift BCD and binary	**[0 | 1 | 4]**	0000 0001 0100	**01100000**
after 6th joined left shift BCD and binary	**[0 | 2 | 8]**	0000 0010 1000	**11000000**
BCD digit 1000 > 0100→ correction	+[0 | 0 | 3]	+0000 0000 0011	
result of the correction	[0 | 2 | 11]	0000 0010 1011	**11000000**
after 7th joined left shift BCD and binary	**[0 | 5 | 7]**	0000 0101 0111	**10000000**
BCD digits 0101 a 0111 *>= 0100→ correction*	+[0 | 3 | 3]	+0000 0011 0011	
result of the correction	**[0 | 8 | 10]**	0000 1000 1010	**10000000**
after 8th joined left shift BCD and binary	**[1 | 1 | 5]**	0001 0001 0101	**00000000**
the end - BCD contains the final result			

The conversion algorithm can be extended to longer numbers. It is discussed in more detail in the follow-up textbook **Logic Circuits on FPGA**, where we preset its C-code and the circuit that implements it.

2.6 Character encoding standard ASCII

The character encoding standard ASCII (American Standard Code for Information Interchange*)* was developed in 1963 and revised. Its most recent update in 1986 is utilized until today. It is also incorporated into the newer codes, as Unicode or UTF8, which assigns the same numeral values to characters defined in ASCII for backward compatibility.

ASCII contains 128 valid characters with decimal values from 0 to 127; see Table 5 on p.29. We can store this range into 8-bit unsigned or signed binary.

Example: Covert text "Hello, Logic!" to ASCII bytes.

Solution: We find out text symbols in ASCII table and write down their ASCII codes, e.g. as:

Symbol	**H**	**e**	**l**	**l**	**o**	**,**		**L**	**o**	**g**	**i**	**c**	**!**
Hexacimal	48	65	6c	6c	6f	2c	20	4c	6f	67	69	63	21
Decimal value	72	101	108	108	111	44	32	76	111	103	105	99	33

Important properties of ASCII

- Control characters - The characters with codes from 0 to 31 are reserved for controlling devices, like printers or teletypewriters. Language C contains escape code for the frequently used control characters with values from 7 to 13. In C, the escape codes begin by a backslash. Table 5 emphasizes them in red color. We mention here only BS - backspace ('\b'), TAB - tabulator ('\t'), LF - linefeed ('\n'), and CR - carriage return - go to the beginning of a line of text ('\r'). We can find the complete overview of control characters on Wikipedia, under ASCII.
- Digits 0-9 have decimal codes from 48 to 57 (hexadecimal from 0x30 to 0x39). Therefore, we can easily convert between a digit character and its numerical value by subtracting or adding 48 (0x30), the ASCII value of the character '0'.
- Letters are stored in two contiguous blocks in alphabetical order. Uppercase letters occupy positions from 65 to 90 (0x41 to 0x5A) and lowercase from 97 to 122 (0x61 to 0x7A), so we can quickly test whether a character is a letter and sort them alphabetically.
- Lowercase and uppercase character codes have difference to each other about 32 decimal, hexadecimal 0x20, so that the conversions between uppercase and lowercase letters are fast, we just add, respectively subtract, 32 (0x20) from the value of the character code.

Table 5 - ASCII table

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0x0	NUL	32	0x20	(mezera)	64	0x40	@	96	0x60	`
1	0x1	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x2	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x3	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x4	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x5	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x6	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x7	\a BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x8	\b BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x9	\t TAB	41	0x29	)	73	0x49	I	105	0x69	i
10	0xA	\n LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0xB	\v VT	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0xC	\f FF	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0xD	\r CR	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0xE	SO	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0xF	SI	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	

2.6.1 Extended ASCII

Basic ASCII uses 7 bits, ending at 127 (0x7f). 8-bit unsigned binary has its maximum value 255 (0xFF). The values from 0x80 to 0xFF were later used for extending ASCII (**extended ASCII**) with national characters, mainly various accented characters.

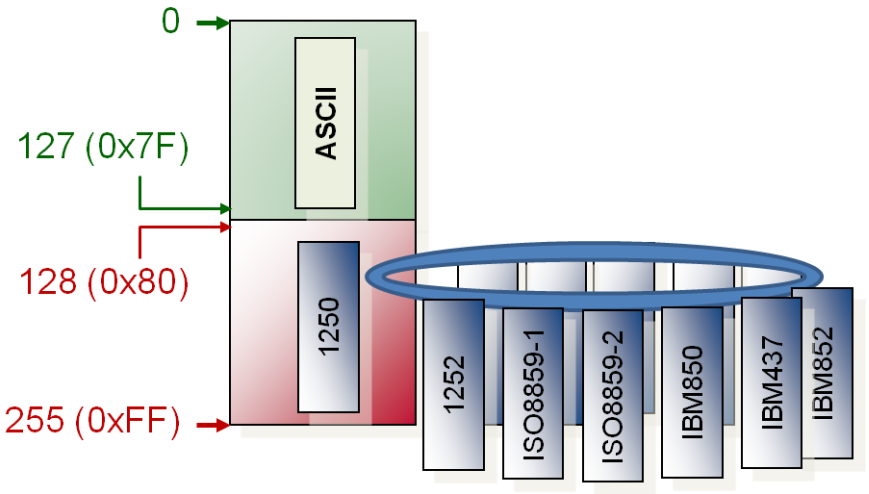

Figure 11 - Principle of extended ASCII

Figure 11 demonstrates the principle of the extension. While the lower decimal values from 0 to 127 remain constant, as defined by the ASCII standard, the upper half from 128 to 255 changed according to national needs.

Many different extending coding options exist. IBM OEM specification contains 81 of them. IANA (Internet Assigned Numbers Authority) has registered 257 of them, which still did not include all code pages used. For example, it does not contain code page 895 (Brothers Kamenický encoding), formerly very popular in the Czech Republic.

We could encounter countless compatibility issues if we did not know the used code page. However, extended ASCII is still used today because it has a significant advantage of storing each symbol in 1 byte.

If we work with extended ASCII characters in language C, we must remember that the extension code page uses values from 128 to 255 (from 0x80 to 0xFF). These are signed 8-bit numbers from -128 to -1 because C language takes char type just assigned char type (signed is here the default), i.e., as an 8-bit signed binary number.

The widespread mistake is skipping whitespace characters with values 0x8 from 0x20 by comparing with the character ' ' = 0x20.

The following program was compiled in v C++, where sizeof(char)=1 (1 byte):

```
char * line = " \n\t à la mode";
// wrong program for skipping of whitespaces
int i=0; while (line[i]!=0 && line[i] <= ' ') i++;
char c = line[i]; // c='1'
```

The program jumped over newline character '\n' (0xA) and tabulator '\t' (0x9), but it also skipped accented character à, because it has negative values in an extended ASCII table.

We correct the code above by using unsigned char types. The characters from 0x80 to 0xFF range in the extended ASCII are now converted to decimal values of 128 to 255, so that only whitespaces are less than or equal to the space character ' '.

```
unsigned char * line = (unsigned char*)" \n\t à la mode";
int i=0; while (line[i]!=0 && line[i] <= ' ') i++;
char c = line[i]; // c='à'
```

In programs, extended ASCII coding is now usually replaced by Unicode. Its basic coding plane has 16-bit characters. All Unicode planes contain codes from 0 to 0x10ffff that store all the world's national characters, including historical scripts and most symbols. To values 0 to 0x7f, Unicode assigns the same characters as ASCII.

UTF-8 (Unicode Transformation Format) is increasingly utilized for storing texts and websites, which has maximum compatibility with ASCII, because its character values from 0 to 0x7f codes are identical to ASCII. UTF8 uses codes from 0x80 to 0xFF to indicate the beginnings of more byte sequences of Unicode characters. Their length can be up to 6 bytes, but UTF8 modern standard RFC 3629 limits sequence to 4 bytes.

The above program that skips whitespaces will not be simplified if we use Unicode type characters, for example, C++ language type wchar_t. The opposite is valid.

In Unicode, we must check not only 6 white characters (less than or equal space ' ' in ASCII), but we should add further tests recognizing at least **19 new characters**[10] added to Unicode for different typographic spacings and line spacings. Many simple programs for text processing apply Unicode and ignore its additional characters. ☺ Silently assume that input texts do not contain them.

In any case, ASCII remains the primary encoding for hardware and small display devices.

2.7 How much is 1000?

We have begun Chapter 2 with the following joke:

After a car accident, a programmer signed me the compensation of 1000 €.
He paid me ten euros with the note that he gave me two euros extra.

Thus, how much is 1000 in different numeral systems? Now, we know that its value depends on used binary encoding :-) We can convert four digit text "1000" to decimal number:

- = **-8** from 4-bit binary signed,
- = **-0** from sign-magnitude,
- = **8** from unsigned binary or signed binary longer than 4 bits,
- = **512** from octal number,
- = **1000**, if we take the digits as decimal,
- = **4096** from hexadecimal number,
- = **to an arbitrary integer** from K-excess code according to its chosen K offset value.

[10] Totally, Unicode adds 25 new whitespaces, but 6 of them are rarely used. We can find complete list of whitespaces on Wikipedia, key "Whitespace character".

2.8 Test from knowledge of Chapter 2

Try to answer 4 questions from memory, i.e., without any aids. The correct solution is in the appendix.

Question 1 - Fill in missing values in tables.

	8-bit unsigned binary		BCD number	
Decimal number	Binary	Hexadecimal	Binary	Hexadecimal
100	0110 0100	64	0001 0000 0000	100
150				
50				
300				

	10-bit signed binary		Straight binary - sign-magnitude	
Decimal number	Binary	Hexadecimal	Binary	Hexadecimal
-100	11 1001 1100	39C	10 0110 0100	264
-10				
	11 1111 1110			
		200		
			10 0000 0100	
511				

Question 2 - Consider the operands written as decimal numbers. Fill decimal values of the results of arithmetic additions and subtractions, if the numbers are stored in given format.

Format of binary number	Length in bits	Operation with decimal values	gives the result with decimal value
unsigned	8	100+200=	44
unsigned	10	100+200=	300
unsigned	8	200+200=	
unsigned	9	200+200=	
unsigned	8	127+1=	
signed	8	127+1=	
signed:	8	100-150=	
signed:	12	100-150=	

Question 3 - Consider 8-bit binary operands. Write the results of the operations:

string (vector) of bits	The left shift by 1 bit		The right shift by 1 bit	
	arithmetic	logical	arithmetic	logical
1000 0001	0000 0010			
1111 1111				
0101 0101				
1010 1010				

Question 4 - What are the final values of iresult and cresult of the program in C language?

```
char c1 = 'A';
char c2 = 'b';
int iresult = c2 - c1;      //  iresult =.............
char cresult = '0' + 5;     //  cresult =..............
```

3 Appendix

3.1 Alphabetical list of used terms and abbreviations

ALU Arithmetic Logic Unit - ALU processor is an essential component of the processor that performs all arithmetic and logical operations.

ASCII American Standard Code for Information Interchange - coding of characters, see Chapter 2.6 on page 28.

BCD Binary Coded Decimal - encoding of a decimal number for good comprehensibility of human and easy visualization, see Chapter 2.5 on page 25.

biased representation - the other name of ***Excess-K*** encoding of numbers, see Chapter 2.6.1 on page 30.

Borrow it means borrowing a bit from higher order during arithmetic overflow in the direction down, see Chapter 2.1.5 on page 11. However, processors do not usually distinguish the direction of overflow and Carry denotes the both direction.

Carry arithmetic overflow of binary number range in the direction up, see Chapter 2.1.5 on page 11. However, processors do not usually distinguish the direction of overflow and Carry denotes the both directions, and it is the main status flags generated by ALUs of processors.

Excess-K coding of numbers, see Chapter 2.1.5 on page 20.

Extended ASCII - the extension of ASCII code, see Chapter 2.6.1 on page 30.

logic gate it was originally designated as an electronic logical element. Today, it often denotes the schematic symbol of a logic operation, see the following textbook Logical Circuits on FPGA.

LSB "*least significant bit*" or "*right-most bit*". LSB also means a byte order as "least significant byte". We distinguish whether LSB refers to byte or bit from the contextual description, see Figure 1 on page 7.

MSB "*most significant bit*" or "high-order bit". MSB also means a byte order as "most significant byte". We distinguish whether MSB refers to byte or bit from the contextual description, see Figure 1 on page 7.

offset binary - the other name for K-excess coding of numbers, see Chapter 2.3 on page 20.

overflow the term refers to arithmetic overflow that occurs when the result is outside of a range of used binary numbers. In processors, this term mainly denotes overflow flag that is related to arithmetic overflow with signed binaries, and it means that the result of the operation is meaningless - as a negative result of the addition of two positive numbers.

straight binary - coding of numbers, see Chapter 2.6.1 on page 30.

sign–magnitude - the other name of straight binary coding of numbers, see Chapter 2.6.1 on page 30.

signed binary - the abbreviation for the binary coding of signed integers in two's complement, see Chapter 2.2 on page 13.

unsigned binary - the abbreviation for the binary coding of positive integers, see Chapter 2.1 on page 8.

3.2 Solution of test from Chapter 2

The solution is related to test on page 32.

Question 1 - Fill in missing values in tables.

	8-bit unsigned binary		BCD číslo	
Decimal number	Binary	Hexadecimal	Binary	Hexadecimal
100	0110 0100	64	0001 0000 0000	100
150	1001 0110	96	0001 0101 0000	150
50	0011 0010	32	0000 0101 0000	050
300	impossible	impossible	0011 0000 0000	300

	10-bit signed binary		Straight binary - sign-magnitude	
Decimal number	Binary	Hexadecimal	Binary	Hexadecimal
-100	11 1001 1100	39C	10 0110 0100	264
-10	11 1111 0110	3F6	10 0000 1010	20A
-2	11 1111 1110	3FE	10 0000 0010	202
-512	10 0000 0000	200	out of the range	out of the range
-4	11 1111 1100	3FC	10 0000 0100	204
511	01 1111 1111	1FF	01 1111 1111	1FF

Question 2 - Consider the operands written as decimal numbers. Fill decimal values of the results of arithmetic additions and subtractions, if the numbers are stored in given format.

Format of binary number	Length in bits	Operation with decimal values	gives the result with decimal value
unsigned	8	100+200=	44
unsigned	10	100+200=	300
unsigned	8	200+200=	144
unsigned	9	200+200=	400
unsigned	8	127+1=	128
signed	8	127+1=	-128
signed:	8	100-150=	impossible, 150 is out of 8-bit signed range
signed:	12	100-150=	-50

Question 3 - Consider 8-bit binary operands. Write the results of operations:

string (vector) of bits	The left shift by 1 bit		The right shift by 1 bit	
	arithmetic	logical	arithmetic	logical
1000 0001	0000 0010	0000 0010	1100 0000	0100 0000
1111 1111	1111 1110	1111 1110	1111 1111	0111 1111
0101 0101	1010 1010	1010 1010	0010 1010	0010 1010
1010 1010	0101 0100	0101 0100	1101 0101	0101 0101

Question 4 - What are the final values of iresult and cresult of the program in C language?

```
char c1 = 'A';
char c2 = 'b';
int iresult = c2 - c1;      //  iresult = 33 (0x21)
char cresult = '0' + 5;     //  cresult = '5'
```

3.3 GNU Free Documentation License

Version 1.3, 3 November 2008

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to

text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section.

You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single

copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version,

but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.