

A P O L O S

Prerequisite for Subjects

Computer Architectures

&

Logic Systems and Processors

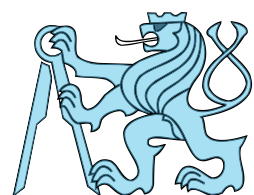
Richard Šusta



Department of Control

Engineering

CTU-FEE in Prague



Version 1.1 from September 25, 2017

Obsah

1	Introduction to Prerequisite.....	4
2	Logic functions	5
2.1	Description of logic functions by truth tables	5
2.2	Value X - don't care	7
2.3	Truth table defined by sets of values	9
2.4	Karnaugh map.....	11
2.4.1	Karnaugh maps for different sizes	13
2.5	Overview of the main logic functions	14
2.6	Operators and logic functions.....	16
2.7	Logical diagram.....	17
2.7.1	Bubbles of negations.....	19
2.7.2	Implementation of logical diagrams	19
2.7.3	Conversion of logical diagram to expression.....	20
2.8	Test from knowledge of Chapter 2	21
3	Integers expressed in binary system	22
3.1	Unsigned binary.....	23
3.1.1	Changing bit width of numbers.....	23
3.1.2	Logical shifts.....	23
3.1.3	Conversion of unsigned binary to decimal number	24
3.1.4	Conversion of decimal number to unsigned binary	25
3.1.5	Arithmetic overflow during additions and subtractions.....	26
3.2	Signed integers in two's complement	28
3.2.1	Important properties	29
3.2.2	Arithmetic negation by two's complement	29
3.2.3	Conversion of decimal number to signed binary	29
3.2.4	Conversion of signed binary to decimal	30
3.2.5	Change of bit length - sign extension.....	31
3.2.6	Logical and arithmetic shifts.....	32
3.2.7	Arithmetic overflow for addition and subtraction	34
3.3	Signed integer in straight binary and offset binary.....	35
3.4	Hexadecimal notation	36
3.4.1	Hexadecimal number	36
3.4.2	Numeral systems	37
3.5	BCD - Binary Coded Decimal.....	38
3.5.1	How to multiply BCD by 2.....	40
3.5.2	Conversion of unsigned binary to BCD.....	41
3.6	Character encoding standard ASCII.....	42
3.6.1	Extended ASCII.....	44
3.7	How much is 1000?	46
3.8	Test from knowledge of Chapter 3	47

4	Appendix.....	48
4.1	Alphabetical list of used terms and abbreviations.....	48
4.2	Solution of test from Chapter 2.....	49
4.3	Solution of test from Chapter 3.....	50

Seznam obrázků

Figure 1	- 7segment display.....	7
Figure 2	- Truth table in matrix form.....	11
Figure 3	- Genesis of Karnaugh map 4x4.....	11
Figure 4	- Dependences in Karnaugh map 4x4.....	12
Figure 5	- Some styles for drawing Karnaugh map 4x4.....	12
Figure 6	- Karnaugh maps for other sizes than 4x4.....	13
Figure 7	- Logic function of 1-input variable.....	14
Figure 8	- Logic functions of 2 input variables.....	14
Figure 9	- Karnaugh maps of main logic functions of two variables.....	15
Figure 10	- Symbols for logical operators.....	16
Figure 11	- Logical diagram and its logical expression.....	17
Figure 12	- Some possibilities for the evaluation of AND and OR.....	17
Figure 13	- Reducing number of inputs AND and OR gate.....	18
Figure 14	- Gates NAND and NOR.....	19
Figure 15	- Double negation.....	19
Figure 16	- Bubble of negations for inputs and outputs.....	19
Figure 17	- Byte, bit, MSB, LSB.....	22
Figure 18	- Adding and subtracting unsigned binary.....	27
Figure 19	- 4-bit unsigned and signed binaries.....	28
Figure 20	- Signed extension.....	31
Figure 21	- Logical and arithmetic right shifts.....	32
Figure 22	- Left shift of 8-bit binary.....	33
Figure 23	- Straight binary (Sign-magnitude).....	35
Figure 24	- Excess K with K=8.....	35
Figure 25	- BCD číslo 35.....	39
Figure 26	- ENIAC Electronic Numerical Integrator and Computer.....	39
Figure 27	- Principle of extended ASCII.....	44

List of table

Table 1	- Decoder "One-hot" - 1 from 8.....	9
Table 2	- Decoder "One-cold" - 1 from 8.....	10
Table 3	- Adding +1 to 8bit unsigned binary.....	26
Table 4	- Arithmetic overflow for addition of 8-bit signed binaries.....	34
Table 5	- Conditions for overflow of signed binaries.....	34
Table 6	- Some conventional radixes for numeral systems.....	37
Table 7	- ASCII table.....	43

1 Introduction to Prerequisite

A prerequisite *means a course that a student must complete as a sensible or arbitrary requirement for another course.* [Merriam-Webster's unabridged dictionary]

Its purpose

Many students met with logic circuits and binary numbers in high school or studied the topic by ourselves, but for others, it is a new concept. When I had started lectures from the basics, knowledgeable students wrote me in the ratings of subjects that they were bored in the early lectures. As my response to their comments, I had next year focused on the interesting questions faster, but less familiar listeners expressed complaints, in turn, that they did not understand the opening passages. To satisfy all, I wrote the prerequisite for unifying knowledge. I have included only simple concepts in it. Things that are more difficult remained in lectures.

How to study it

Read the entire text. If you understand to some part, do not skip it, but read it very briefly, maybe, you can find out some new evidence in it. However, slow your reading down if you encounter less familiar concepts, or you will not be too sure in some details and carefully study text including methods used in practical examples.

Overview of Chapters

Chapter 2

The chapter supposes familiarity with conversions of small numbers (from 0 to 15) to unsigned binary integers. If you have not learned it yet, first read the beginning of chapter 3.

Chapter 2 contains a minimal knowledge of logic functions. It starts by a college style \odot , i.e. by a mathematical definition that is necessary for following logical descriptions, but you need not be afraid, the next parts deal only with simple concepts.

We will enroll ways of specifying logic functions by a truth table. Instead of slave entering of all possible combinations, more concise methods can often be used. One of them, it is drawing Karnaugh map (KM) of a logic function, which is the most common procedure in practice for expressing of smaller logic functions. The description of KM remains at "cookbook level". Its theoretical background will be presented in lectures.

Finally, we will discuss logic functions NOT, AND, OR, and XOR. You probably know them from the programming languages C, C #, or Java as bit operators \sim , $\&$, $|$, and \wedge , and NOT, AND, or OR functions also as logical operators: $!$, $\&\&$, and $||$.

In the conclusion of the chapter, we will show the simple way for conversions between logical diagrams (schemas) and logical expressions.

Chapter 3

You could "theoretically" know its content from programming courses. We will present the coding of binary integers as unsigned and signed numbers, and we will explain their overflows during additions or subtractions. Furthermore, we will describe the logical and arithmetic shifts left and right that are very critical operations in logic circuits. In languages C, C # and Java, they are partially included as shift operators \ll and \gg .

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

2 Logic functions

Consider logical variables that take values only from a finite set B.

Completely Specified Logic Function of n variables $y = f(x_1, x_2, x_3, \dots, x_n)$ is the mapping:

$$B^n \rightarrow B, \text{ where } (x_1, x_2, x_3, \dots, x_n) \in B^n, x_i \in B, y \in B.$$

- If B set contains only 2 elements, i.e., it has *cardinality* $|B|=2$, then we have two-value logic¹. Set B can be written in that case as $B = \{ '0', '1' \}$, where '0' and '1' denote the logical zero (false) and the logical one (true).
- B^n denotes Cartesian product, i.e., the set of all possible n-tuples formed from B, and if $|B|=2$ then $|B^n|=2^n$.
- By *mapping* $B^n \rightarrow B$, we specify output values firmly assigned to elements from B^n . For n logical input variables, we can define 2^{2^n} different logic functions:
 - for $n=1$, there are $2^{2^1} = 2^2 = 4$ different logic functions,
 - for $n=2$, there are $2^{2^2} = 2^4 = 16$ different logic functions,
 - for $n=3$, there are $2^{2^3} = 2^8 = 256$ different logic functions.

Example: Let $B = \{ '0', '1' \}$. Logic function of 2 inputs can be written as $y = f(x_1, x_2)$. Its *Cartesian product* B^2 contains four 2-tuples, i.e. $B^2 = \{ ('0', '0'), ('0', '1'), ('1', '0'), ('1', '1') \}$.

We select one mapping from 16 possible that exist. If the inputs are different, we assign logical '1' to output, otherwise logical '0'. This logic function is known as *XOR* or non-equivalence. The following mapping defines our function $y = \text{xor}(x_1, x_2)$:

$$\begin{array}{l} \mathbf{xor: } B^2 \rightarrow B = ('0', '0') \rightarrow '0' \quad \text{simplified notation} \quad 00 \rightarrow 0 \\ ('0', '1') \rightarrow '1' \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 01 \rightarrow 1 \\ ('1', '0') \rightarrow '1' \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 10 \rightarrow 1 \\ ('1', '1') \rightarrow '0' \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 11 \rightarrow 0 \end{array}$$

2.1 Description of logic functions by truth tables

A mapping assigns just one value of B for each combination of input variables. The table is only another way how to write a mapping. XOR function, from the previous example, can be specified as the table that can have many formats, for example:

x1	x2	xor
0	0	0
0	1	1
1	0	1
1	1	0

alternatively:

x1	x2	xor
1	1	0
1	0	1
0	1	1
0	0	0

or:

x1	x2	xor
0	0	0
1	1	0
1	0	1
0	1	1

All three tables define the identical logic function. The order of rows in a truth table does not matter. We can write rows in any order if we satisfy the condition that we listed all of them. The definition of logical functions only requires that we must assign just one output value from B to each possible combination of input variables.

¹ For designing logic circuits, two-value logic with '0' and '1' logic is not sufficient. Even in this text, we will soon introduce 3-value logic by adding value X (don't care) because we will need it. For professional work, 9-value logic MVL-9 is frequently used. You will learn about it in specialized educational subjects.

A list of all the possible combinations is lengthy, so we often join several logic functions into one table. For example, we can write xor along with other common logic functions::

x1	x2	xor	and	or	nand	nor
0	0	0	0	0	1	1
0	1	1	0	1	1	0
1	0	1	0	1	1	0
1	1	0	1	1	0	0

Sometimes it is useful to reduce the number of rows. For example, the selection from more concurrent requests can be solved by priority logic functions. Their output p can specify the number of the highest order input xi that is in logical '1'.

For 3 inputs, a priority function can have right table:

- Output p3 = 00, only if all inputs are '0'.
- Output p3 = 01, if only input x1='1'.
- If x3='0' and x2='1' then p3=10 regardless of x1 input value, because we assigned have higher priority to x2 than to x1.
- Output p3=11, if the most priority input x3='1' regardless of values of remaining inputs.

x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

We can shorten the previous table by merging inputs. If the same output value is assigned to more rows for all possible values of some input or a group of inputs, we can replace that input or those inputs by "wildcard(s)", e.g. by '-' (hyphen).

x3	x2	x1	p	
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

→

x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	-	1	0
1	-	-	1	1

→

x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	-	1	0
1	-	-	1	1

The new table (top right) contains only 4 rows. Wildcards have reduced rows by merged inputs, i.e., we have replaced rows by some prescriptions how to generate them. Now we write easily even greater priority function for 10 inputs.

x10	x9	x8	x7	x6	x5	x4	x3	x2	x1	p10				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	0	1	-	0	0	1	0	0
0	0	0	0	0	0	0	1	-	-	0	0	1	1	0
0	0	0	0	0	0	1	-	-	-	0	1	0	0	0
0	0	0	0	0	1	-	-	-	-	0	1	0	1	0
0	0	0	0	1	-	-	-	-	-	0	1	1	0	0
0	0	0	1	-	-	-	-	-	-	0	1	1	1	0
0	0	1	-	-	-	-	-	-	-	1	0	0	0	0
0	1	-	-	-	-	-	-	-	-	1	0	0	0	1
1	-	-	-	-	-	-	-	-	-	1	0	1	0	0

We write only 11 rows instead of $2^{10}=1024$ rows needed for p10 full table: The last row of the table contains 9 wildcards, 1----- . In fact, it represents the prescription, which generates $2^9=512$ rows, because each wildcard takes two values as '0' and '1'. All created rows have the same output p10 = 1010.

Another example: The table below left is, in fact, shorter specification of the table right:

c	b	a	y
⋮	0	⋮	1
⋮	1	0	0
0	1	1	1
1	1	1	0

→

c	b	a	y
0	0	0	1
0	0	1	1
1	0	0	1
1	0	1	1
0	1	0	0
1	1	0	0
0	1	1	1
1	1	1	0

Certain functions are difficult to define without wildcards, as the previous priority function of p10 for 10 inputs. When we write truth tables by hand, however, the excessive usage of wildcards reduces clarity, as it is evident from the table above left. At first glance, we do not know whether we have specified all possible combinations of inputs.

The wildcards are widely applied in computer processing of truth tables, e.g. during the processes of minimization of logical functions.

2.2 Value X - don't care

If we want to write the truth table for the decoder that converts decimal digits to a 7-segment display, then we can easily create the table for input values 0-9 (binary *unsigned* from 0000 to 1001, see Chapter 3.1, page 23).

However, what do we assign to inputs **10-15** (from 1010 to 1111)? There are not required in the entry. We can select something for them, but at the time of the table creation, we do not know if our randomly assigned output values do not impede subsequent operations, such as further minimizing of our logic function.

A wiser solution is to postpone our decision. As a sign of the delayed decision, we use "**don't care**" mark, which specifies that the output value does not matter to us. This mark is often written as X.

With the aid of X a wildcards '-', we easily write table for the conversion of decimal digits to the 7-segment display. We suppose that segments are lighting on logical '1' input.

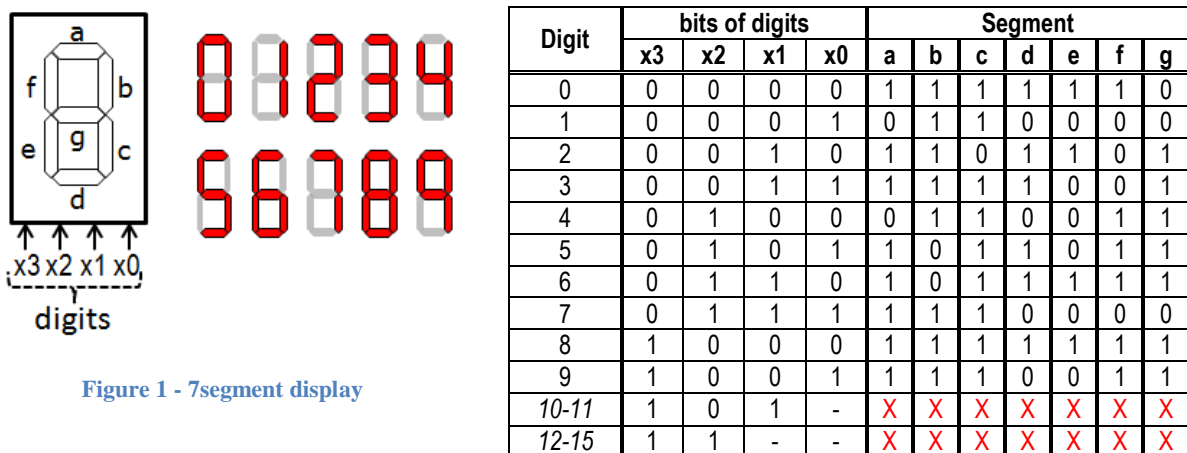


Figure 1 - 7segment display


The table of 7-segment display, right in Figure 1, is nearly professional except dividing inputs and outputs to columns. In more concise notations, logical values are often combined in sequences, or vectors respectively, which significantly reduce the table.

For example, instead of

x3	x2	x1	x0
0	0	0	0

we write only 0000 and we add the order of the variables in the sequences into the header of the table. Table in Figure 1 can be reduced to a more concise version below right:

Digit	bits of digits				Segment						
	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X



Digits	binary x: 3210	Segment abcdefg
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1110011
10-11	101-	XXXXXXXX
12-15	11--	XXXXXXXX

The sequences (vectors) of logical '0' and '1' also have practical significance. They can faster specify logic functions in professional development tools for designing circuits. Logical values are processed here often in the form of vectors to shorten programming statements. In contrast, there are almost not used longer definitions of logic functions by filling tables divided into individual columns.

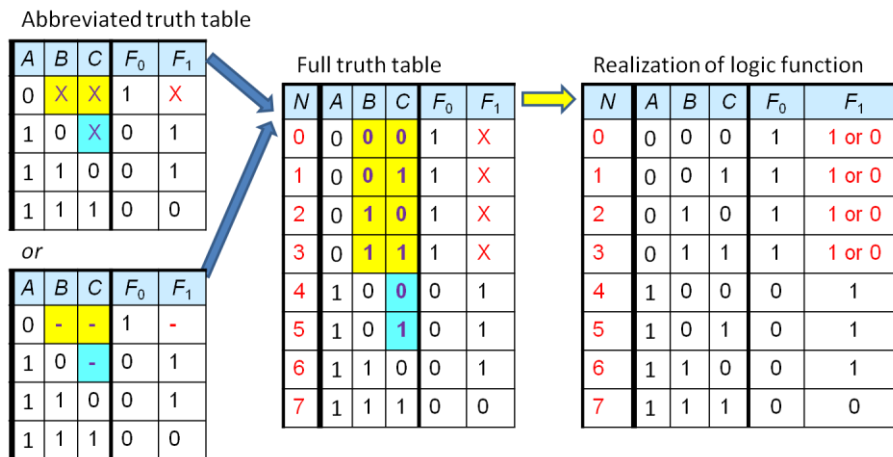
More about "don't-care"

- "don't-care" specifies only a designer's note that an output value will be assigned latter, i.e., during the next steps depending on what latter appears more useful. It is, therefore, a sign of a delayed decision (something like to-do a comment, or ToDo respectively).
- "don't-care" can be used only for outputs. In designs, we cannot, in any case, defer a decision of input values. We must know them in advance.
- "don't care" has no meaning of "an unknown output". It corresponds more to "not assigned yet", "unspecified yet", or "anything can be here, we do not care about value".
- "don't-care" cannot be physically realized in the circuits, and eventually, all "don't-care" must be replaced by exact logical values, e.g., by logical '0' or logical '1'.²

Unfortunately in many publications, wildcards for merged inputs and "don't care" symbols for outputs are often denoted by the same symbols, frequently as characters 'X' or '-'. Then, we must distinguish their exact meanings according to their position in truth tables, whether symbols are placed in an input part or an output part.

² In pursuit of a maximum precision here, we avoid the claim that X (don't care) is always and everywhere replaced by either logic '0' or '1'. Usually it happens but there are other options. For example, the output may go into high impedance state, i.e., to be disconnected, which is widely used in computer buses, as you will see later in lectures.

- Input of a logic function: If we see a group "0 - -" or "0 X X" (according to used notation), then it generates 4 rows of inputs 000, 001, 010 a 011 with the same output value. Character 'X', or '-' respectively', has here meaning of wildcard, *i.e., it is the prescription for generating of input values.*
- Output of a logic function: For example, a group "1X" or "1-" means that a decision about output value has been delayed. Here, the symbol always specifies "don't care". We cannot use any generation by wildcards for outputs - each output must have only one fixed value in the table that will be finally used for physical realization of a logic function. We can only temporarily postpone our decision about assigned value.



2.3 Truth table defined by sets of values

Table 1 below describes 8 logic functions whose outputs F0 to F7 become '1' only for one logical combination of input values, the functions reports its presence. Together, the functions make up one-hot decoder that is a crucial element which forms the basis of many other logical constructions.

N	C	B	A	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0	0	0	0
3	0	1	1	0	0	0	1	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0
5	1	0	1	0	0	0	0	0	1	0	0
6	1	1	0	0	0	0	0	0	0	1	0
7	1	1	1	0	0	0	0	0	0	0	1

Table 1- Decoder "One-hot" - 1 from 8

Neither of the previous methods is suitable for the elegant brief description of this decoder. However, we may specify its logic function by sets of input values with outputs equaled to logical '1', because the count of logical '1's is here less than '0's. This set is called **Onset**. We encode inputs as unsigned binaries (*see Chapter 3.1 page 23*).

Table 1 is now reduced to one line, to the following lists of onsets.

$$F0^{on} = \{0\}, F1^{on} = \{1\}, F2^{on} = \{2\}, F3^{on} = \{3\}, F4^{on} = \{4\}, F5^{on} = \{5\}, F6^{on} = \{6\}, F7^{on} = \{7\}$$

Outputs of functions F0 to F7 are equal to logical '0' for all unspecified inputs.

Table 2 describes another analogous decoder, which is called the **one-cold decoder** because the outputs of F0 to F7 functions are in '0' for exactly one input combination.

N	C	B	A	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1	1	1	1	1
2	0	1	0	1	1	0	1	1	1	1	1
3	0	1	1	1	1	1	0	1	1	1	1
4	1	0	0	1	1	1	1	0	1	1	1
5	1	0	1	1	1	1	1	1	0	1	1
6	1	1	0	1	1	1	1	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1	0

Table 2- Decoder "One-cold" - 1 from 8

Here, the definition by onsets is not advantages. We use now **offsets**³ that are the sets of input values, in which outputs of a logic function are equal to logical '0'.

One-cold decoder is easily defined by offsets:

$$F0^{off} = \{0\}, F1^{off} = \{1\}, F2^{off} = \{2\}, F3^{off} = \{3\}, F4^{off} = \{4\}, F5^{off} = \{5\}, F6^{off} = \{6\}, F7^{off} = \{7\}$$

Here, the unspecified values are again in the second set, i.e. they are equal to logical '1'.

If we add don't care set, we can use onsets and offsets *also for* logic functions that contain don't care marks.

N	C	B	A	X	Y
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	X	0
3	0	1	1	X	0
4	1	0	0	1	0
5	1	0	1	1	X
6	1	1	0	1	1
7	1	1	1	1	1

We can write logic functions X(C,B,A) a Y(C,B,A), specified by the table above, as:

$$X: X^{off} = \{0,1\}, X^{dc} = \{2,3\}; Y: Y^{on} = \{6,7\}, Y^{dc} = \{5\},$$

For each function, we chose a method that gave us the least work. We write function X by offset and don't care set, because X outputs contain less '0' than '1', and we select for Y function onset a don't care set.

Note: Mathematical notation for onset, offset and don't care *set can* differ according to customs of an author. Symbols F^{on} , F^{off} a F^{dc} used here are not a general rule. We can encounter other notations. For example, onset is frequently written by lowercase m (from minterm) and offset as uppercase M (from Maxterm). Thus, the previous functions X and Y could be specified in other texts as:

$$X: M(0,1), dc(2,3); Y: m(6,7), dc(5)$$

The names "minterm" and "Maxterm" follow from the minimization of logic functions whose explanation is beyond the scope of this publication. You will learn about it in lectures.

³ Name 'offset' is quite misleading, since this term more frequently indicates a displacement, shift or steady-state error in mathematics and science papers, but 'offset' is really used in logic circuits.

2.4 Karnaugh map

In technical practice, logic functions with a smaller number of inputs are usually specified by Karnaugh map (KM). We present its detailed derivation with the aid of the truth table of logic function $Y = f(d,c,b,a)$ with four inputs (d,c,b,a) and one output Y. We use constant y00 to y15 (equaled to some logical values '0', '1' or X) for its outputs to transparently show their order.

d	c	b	a	Y
0	0	0	0	y00
0	0	0	1	y01
0	0	1	0	y02
0	0	1	1	y03
0	1	0	0	y04
0	1	0	1	y05
0	1	1	0	y06
0	1	1	1	y07
1	0	0	0	y08
1	0	0	1	y09
1	0	1	0	y10
1	0	1	1	y11
1	1	0	0	y12
1	1	0	1	y13
1	1	1	0	y14
1	1	1	1	y15

		0		1		b
d	c	0	1	0	1	a
0	0	y00	y01	y02	y03	
0	1	y04	y05	y06	y07	
1	0	y08	y09	y10	y11	
1	1	y12	y13	y14	y15	

Figure 2 - Truth table in matrix form

The truth table left has 16 rows, but the inputs c and d have the same value for groups of 4 rows. Such table can be advantageously abbreviated by rewriting in a matrix form 4x4, shown at right, where each output has input values specified by its column and row.

We modify table right in Figure 2. We swap the last two columns, and then the last two rows. We have obtained the middle table in Figure 3 — that is already Karnaugh map of logic function Y. Logical '1' of inputs are placed side by side in it, so instead of writing 0 and 1 we can draw lines symbolizing value of '1', see the table at right in Figure 3.

		0		1		b
d	c	0	1	0	1	a
0	0	y00	y01	y02	y03	
0	1	y04	y05	y06	y07	
1	0	y08	y09	y10	y11	
1	1	y12	y13	y14	y15	

		0		1		b
d	c	0	1	1	0	a
0	0	y00	y01	y03	y02	
0	1	y04	y05	y07	y06	
1	1	y12	y13	y15	y14	
1	0	y08	y09	y11	y10	

		0		1		b
d	c	0	1	1	0	a
		y00	y01	y03	y02	
		y04	y05	y07	y06	
		y12	y13	y15	y14	
		y08	y09	y11	y10	

Figure 3 - Genesis of Karnaugh map 4x4

The most important property of Karnaugh maps, the necessary condition for any Karnaugh map, is the following. One input variable is only changed if we performed any movement in it by one cell vertically or horizontally.

For example, y00 output is assigned to inputs dcba=0000 and its neighbor y04, in the row below, to dcba = 0100. The movement from y00 to y04 changed only input c from '0' to '1'.

The rule also applies to the edge map, for example, when moving from the first to the fourth row in the same column. If we take the last column for the show - output y02 has inputs dcba=0010 the output y10 has inputs dcba = 1010. Input d has only changed from '0' to '1'.

Another example, output y14 at the end of the 3rd row has inputs dcba=1110, and y12 in the beginning of the same row has inputs dcba = 1100. Input b changed only from '1' to '0'.

The ordering of values by such way that two successive values differ only in one bit is called Gray code, rarely known also as reflected binary code. The code is a vital concept, used not only in minimizing logic functions but also in sensors for position and transfers information, such as error correction in digital television.

Indexes i (of y_i outputs in Karnaugh maps) are not successive. If we compare numbers of indexes on two columns in any row, that the 2nd column has its indexes greater by +1 than the 1st column, the third column by +3 and the 4th column by +2 than the 1st column.

Property follows from the input variables. Each input bit has weight given by power two series, 2^n . If you arrange inputs as dcb a, then input a has a weight of $1 = 2^0$, b input has weight $2 = 2^1$, c input has weight $4 = 2^2$ and d input has weight $8 = 2^3$. The sum (row+column) gives an input index value that corresponds to an output written inside Karnaugh map.

The 2nd column has its indexes +1 higher than the 1st column because 2nd column has $a=1$ and a represent weight 1. The 3rd column has two input variables in '1', a and b , thus it has indexes $+3=(+b+a)$ higher than the 1st column. Analogously, the last column has only input $b=1$, so, its weight=2, thus, the 4th column indexes are +2 higher.

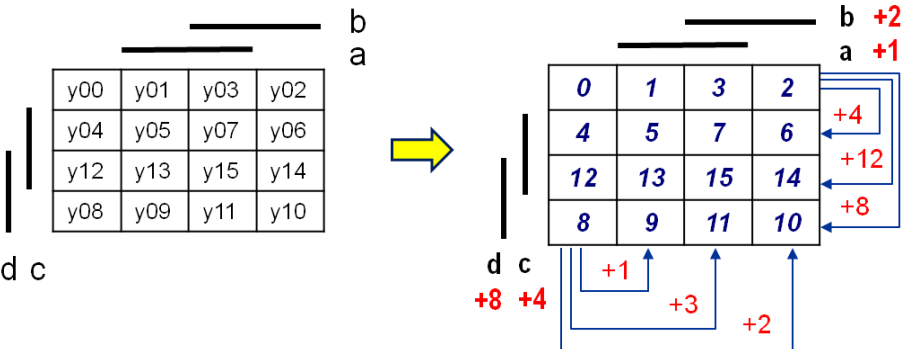


Figure 4 - Dependences in Karnaugh map 4x4

From the above properties, we can derive the differences between the indexes in a single column. The 2nd row has an index value always by +4 greater than the corresponding output of the same column of the 1st row ($a + c = 4$). The 3rd row has higher indexes by +12 ($+ d + c$) from the 1st row and the 4th row by +8 ($+ d$). If we have correctly allocated indexes to the 1st row, then the rest can be mechanically derived.

Karnaugh map, abbreviated KM, which Figure 4 depicts left, of course, is not the only way to draw it or how to organize the input variables. KM styles depend on customs of authors. Several different KM drawings, of many possible, are shown in Figure 5.

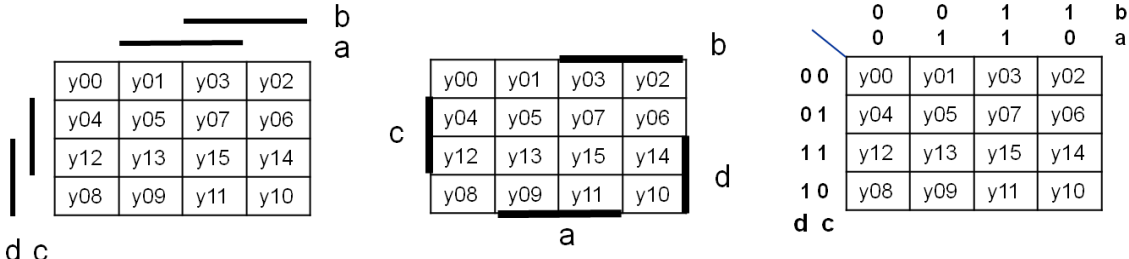


Figure 5 - Some styles for drawing Karnaugh map 4x4

If variables are organized in different orders, they have certainly different weights, and then, we also obtain a different order of indexes. However, any KM must always meet the previously mentioned property of Gray code. One input variable (a bit) only changes by any movement, either in a column or a row, including crossing over the edges of a map.

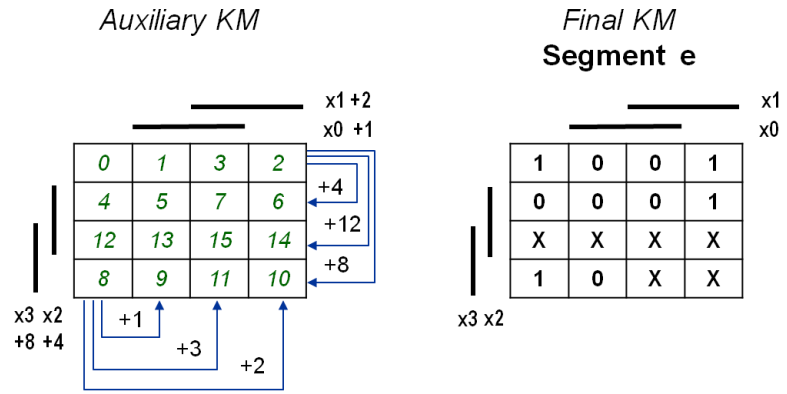
Example: Draw the Karnaugh map (KM) of segment e of 7-segment display.

Solution: Figure 1 on page 7 depicts truth table of 7-segment display. We pick from it the values for segment e.

Given that we have not much experience with drawing KM we proceed through an intermediate step to prevent mistakes ☺.

First, we draw an auxiliary KM, in which we enter indexes of our ordering of input variables. According to them, we fill the values of the truth table of segment e.

N	x: 3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	X
12-15	11--	X



2.4.1 Karnaugh maps for different sizes

Karnaugh maps are not suitable for processing in computers. They used exclusively for hand minimizing or writing logic functions with a small number of inputs. Although KMs can be theoretically built for any logical function of any size, their clarity decreases exponentially with increase in the number of variables. It also demonstrates Figure 6, where are depicted some selected possibilities how to draw Karnaugh maps for other sizes than 4x4, including the final ordering of input indexes.

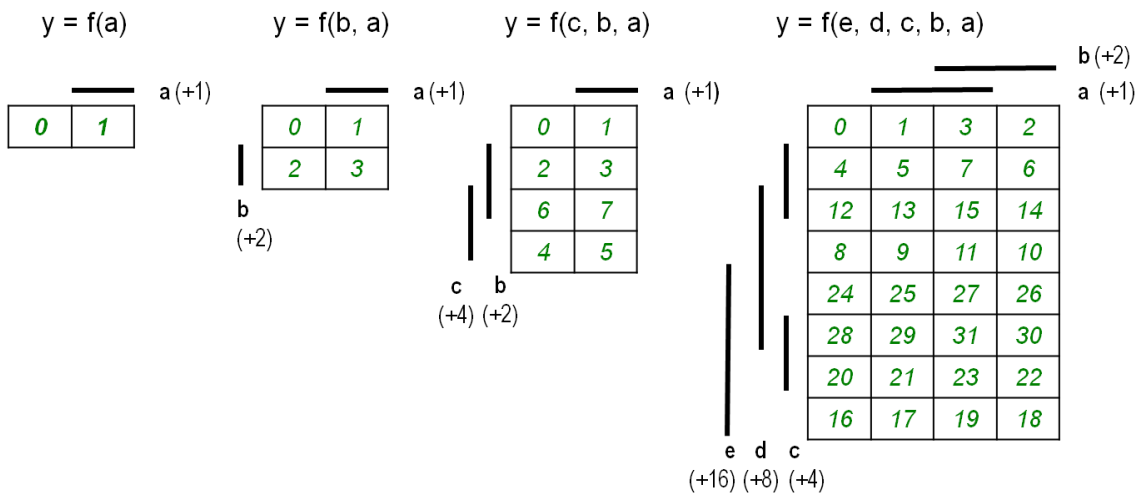


Figure 6 - Karnaugh maps for other sizes than 4x4

Fortunately, it is possible to reduce the number of variables in logic functions by various handy decompositions and expansions, which will be the topics of lectures, so you can always manage with maps to 4x4 for your manual designs ☺.

2.5 Overview of the main logic functions

Figure 7 shows all the logical functions of **one variable input**, including their schematic symbols. Two of these are constant because their output remains constant regardless of the value of the input - F0 has output permanently logical '0' and F3 logical '1'.

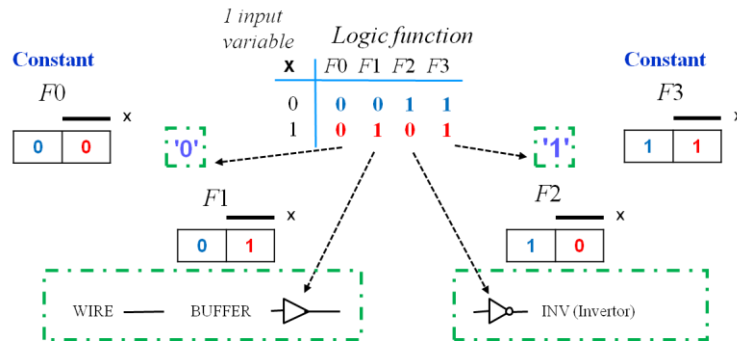


Figure 7 - Logic function of 1-input variable

F1 function has its output value equals the input. If it is a direct connection, then F1 is called WIRE. When F1 is realized by an electronic element, either for an electrical isolation or to obtain higher output current or different signal level, in this case, F1 is called BUFFER to emphasize this fact.

F2 logic function changes the input logical '0' to '1' and '1' to '0'. It is called inverter or negation, sometimes also complement and denoted as INV or NOT.

Figure 8 shows all **logical functions of two input variables**, again including their schematic symbols. When you look at it, you notice, it is present 16 functions, but 6 of them, marked in blue, can be replaced by proper logical functions of one variable.

The first two of these are functions F0 and F15 are not dependent on inputs. We know them as constants from Figure 7. Other 4 functions BUFX, BUFFS, INVX, and INVY have their output values dependent only on one input. If we connect their input that affects output to logic function BUFFER, or inverter (INV) respectively, we can replace them.

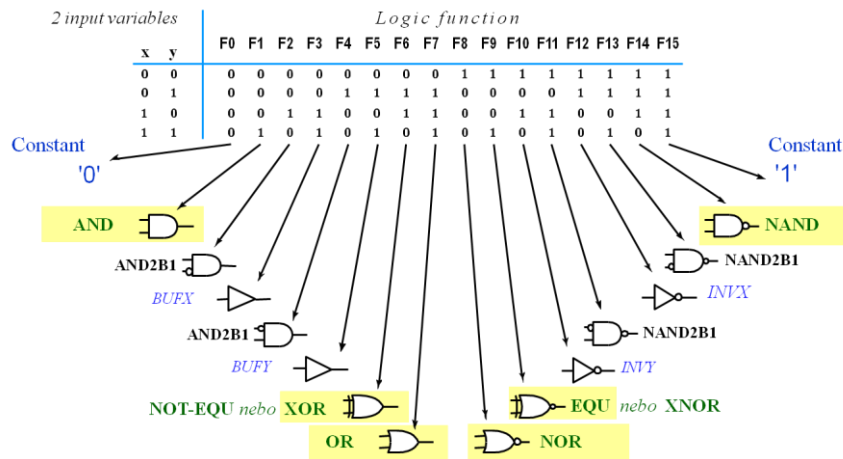


Figure 8 - Logic functions of 2 input variables

Of the remaining 10 logic functions, only 6 of them are practically used: AND, XOR, OR, NOR, EQU, NAND. They are in yellow highlighting. Figure 9 depicts their Karnaugh maps. As we can see from it, these selected logical function of two variables are easy to remember for its symmetry - their output is not dependent on order of inputs, by the other words, their

outputs do not change, if we swap x and y . Additionally, three logical functions in the bottom row of Figure 9 (NAND, EQU, and NOR) are just negated logical functions of the first row, so in fact, we just need to be familiar only with three logical functions of two variables, namely AND, XOR, and OR.

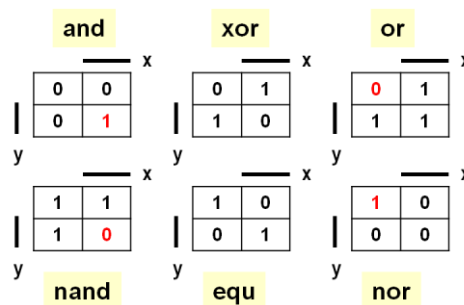


Figure 9 - Karnaugh maps of main logic functions of two variables

For the following text, we define the ordering of logical values as '0' < '1', by the words, logical '0' is less than logical '1'.

- **Logic function AND** gives as its output logical '1' only when the both its inputs are at logic '1'. Therefore, it can be considered as the selection of a minimum value of its inputs. If any input is in logical '0', then the **minimum** is '0'.
- **Logic function OR** is in some ways mirrored AND function. Logic function OR gives as its output logical '0' only when the both its inputs are at logic '0'. Therefore, it can be considered as the selection of a **maximum** value of its inputs. If any input is in logical '1', then the maximum is '1'.

The used analogy of minimum and maximum selection allows us expand the logical AND and OR functions to functions with an arbitrary number of inputs.

- **Logic function AND with n inputs**, $Z = \text{and}(x_{n-1}, \dots, x_1, x_0)$, selects minimum from all its inputs - Z is in logical '1' only when all its inputs $x_i = '1'$. If one or more inputs are '0', then minimum $Z = '0'$.
- **Logic function OR with n inputs**, $Z = \text{or}(x_{n-1}, \dots, x_1, x_0)$, selects maximum from all its inputs - Z is in logical '0' only when all its inputs $x_i = '0'$. If one or more inputs are '1', then maximum $Z = '1'$.

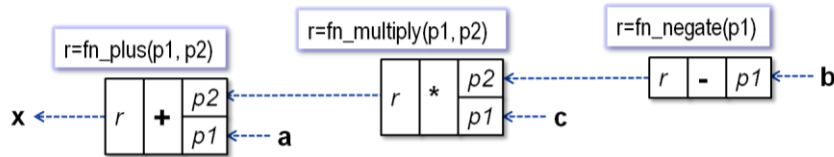
Logic XOR and EQU functions perform the comparison of their inputs:

- **Logic function XOR**, eXclusive OR gives as its output logical '1' when only one its input is in logical '1'. XOR is often described by another way that its output is in logical '1' when the values of its inputs are different. It is sometimes also called NOT-EQU or NON-EQU, but XOR name is much more common in programs or logical systems.
- **Logic function EQU**, EQUivalence, gives as its output logical '1' when the both its inputs have equal values. Sometimes it is also known as XNOR, eXclusive NOT OR. XNOR suggests that this function is negated XOR, but EQU name is more frequent.

XOR logic function can also be extended to multi-input XOR, but the similar extension has not greater practical meaning. Multi-input XOR has a negligible usage, unlike to widely implemented multi-input AND and OR functions. Two-input XOR is enough for the vast majority of circuits; on the other hand, it uses to be a critical component here.

2.6 Operators and logic functions

Taking ordinary mathematical expression, such as $x = a + (-b) * c$, then the evaluating of this expression is performed by the chain of function calls that can be drawn as an expression tree:



or it can be written as: $x = \text{fn_plus}(a, \text{fn_multiply}(c, \text{fn_negate}(b)))$. Function negation, corresponding to a unary operator, has one input $p1$ and returns r (result), while the remaining functions have two inputs, $p1$ a $p2$ because they evaluate binary operators.

Logic functions are also more likely written with the aid of operators than in functional forms. Figure 10 depicts an overview of some possible symbols. The highlights emphasize the most frequent symbols whose choice follows primarily from computer keyboards.

	NOT	AND	OR	XOR
Other alternative operators	x'	$x \cdot y$	$x + y$	$x \oplus y$
	$\neg x$ or \bar{x}	$x \wedge y$	$x \vee y$	$x \neq y$
	$-x$	$x \times y$, xy	$x + y$	
Bit.oper. C,C#,Java	$\sim x$	$x \& y$	$x y$	$x \wedge y$
Log.oper.C,C#,Java	$!x$	$x \&\& y$	$x y$	
Pascal, VHDL	<i>not</i> x	<i>x and y</i>	<i>x or y</i>	<i>x xor y</i>
Graphical symbols				

Figure 10 - Symbols for logical operators

The most appropriate symbols for AND and OR operators would have been probably symbols \wedge a \vee , used by predicate logic, but they are troublesome for entering on standard keyboards. We prefer symbols '+' and '.' that denote additions and multiplications in arithmetic. However, '+' and '.' have entirely different properties as Boolean operators!

The arithmetic addition is not distributive over multiplication, but logical OR function is distributive over logical AND function, see yellow highlighted cell in the table below. Moreover, the addition and multiplication are not idempotent operations, but AND function and OR function have this property, see green highlighted cells.

	<i>commutative</i>	<i>associative</i>	<i>distributive</i>	<i>idempotency</i>
AND '.'	$a.b = b.a$	$a.(b.c) = (a.b).c$	$a.(b+c) = (a.b)+(a.c)$	$a.a = a$
OR '+'	$a+b = b+a$	$a+(b+c) = (a+b)+c$	$a+(b.c) = (a+b).(a+c)$	$a+a = a$

Higher priority (precedence) of multiplication before addition misleadingly induces precedence of AND before OR, for which no real foundation exists. The both logic operations have equal status. However, the higher precedence of AND is frequently introduced for reducing the number of necessary brackets, e.g. in languages C or Java. To the contrary, AND precedence does not exist for example in VHDL language for designing of circuits.

Exercise: Try to prove the laws in the table above with the aid of the fact that we can understand AND and OR functions as selections of a minimum and a maximum, see Chapter 2.5.

2.7 Logical diagram

Logical diagram or logical schema of simple logic functions is, in fact, a process of its evaluation that creates syntactic analyzer (a parser) from a logic expression.

We take the example of $Y = (\text{not } (A \text{ and } B)) \text{ or } (C \text{ and } D)$. Since unary operations have higher precedences in general than binary operations, we can skip braces marked in red. We write the function abbreviated as $Y = \text{not } (A \text{ and } B) \text{ or } (C \text{ and } D)$, respectively, with the aid of operators "+", ".", and "' also as $Y = (A \cdot B)' + (C \cdot D)$.

Its evaluation can begin by the left operation AND: $\lambda_0 = A \cdot B$ [$\lambda_0 = A \text{ and } B$], where λ_0 denotes its intermediate result. Its negation is: $\lambda_1 = \lambda_0' = (A \cdot B)'$ [$\lambda_1 = \text{not } (A \text{ and } B)$]. Then, we evaluate the next operation AND: $\lambda_2 = C \cdot D$ [$\lambda_2 = C \text{ and } D$]. Finally, we join both intermediate results λ_1 and λ_2 by operator OR to $Y = \lambda_1 + \lambda_2 = (A \cdot B)' + (C \cdot D)$ [$Y = \text{not } (A \text{ and } B) \text{ or } (C \text{ and } D)$].

Figure 11 depicts its evaluation. Operations labeled by their names are on the top, but below, the same scheme is drawn by the far more common method, with the aid of symbols. For historical reasons, the graphical symbols for logical operation are often called **logic gates**.

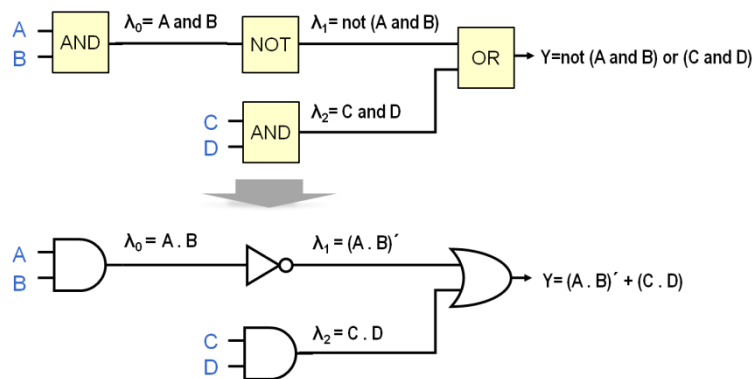


Figure 11 - Logical diagram and its logical expression

Logic diagram, like the tree expression, describes the exact procedure of evaluation. If we have, for example, functions $N = A \cdot B \cdot C \cdot D$ and $R = A + B + C + D$ with more AND or OR operations, we can evaluate them by several ways. Figure 12 depicts some of them.

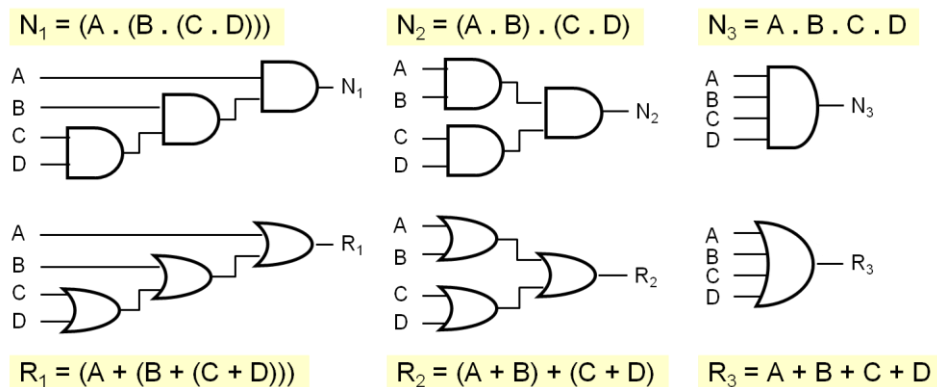


Figure 12 - Some possibilities for the evaluation of AND and OR

Results of N_1 and R_1 are evaluated by concatenating logical operations. We use here associative properties of AND and OR functions. Results of N_2 and R_2 are calculated by another chaining of operations into a tree. Last N_3 and R_3 utilize multi-input AND and OR functions introduced in Chapter 2.5, which compute multiple logical operations in a single step.

From a mathematical point of view, $N=N_1=N_2=N_3$ a $R=R_1=R_2=R_3$, because the logical result does not depend here on the order in which we evaluate logic function. During its realization, it usually also does not matter, because we ask in many cases only correct logical outcome,⁴ so we choose the method that we have liked.

In schemes, logic gates AND and OR sometimes have more inputs than we need. It can happen for some structural reasons, or even because a used graphical editor does not offer appropriate gate ☹.

With the multiple elements, we can reduce the number of inputs in several ways. For AND and OR operations, we can connect inputs of a multi-input logic gate to the same signal due to idempotent law, see Chapter 2.5 on page 14, by this way, we reduce the number of inputs, as shown in Figure 13. If we connection all inputs of multi-input logic gates together, we obtain buffer, which was previously described, see Figure 7 on page 14.

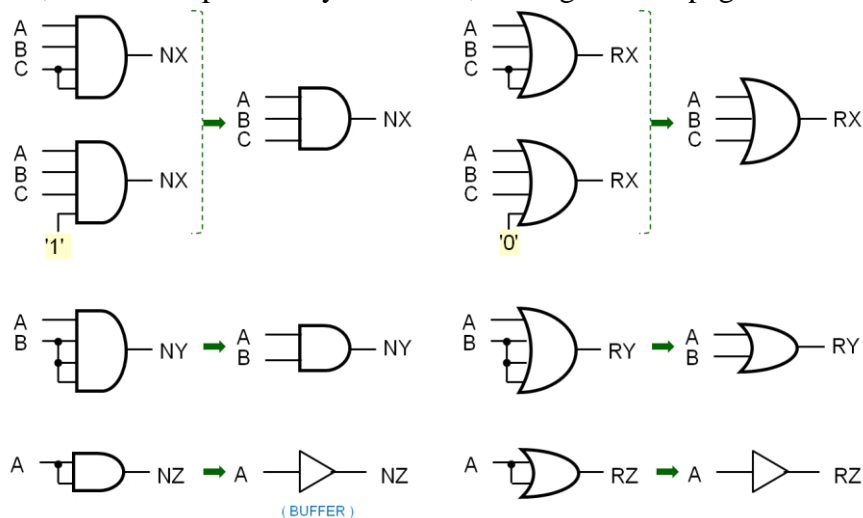


Figure 13 - Reducing number of inputs AND and OR gate

We can also connect to AND and OR gate inputs that we do not need to constants.

- AND corresponds to the minimum of inputs, so we can connect its unused inputs to the maximum, i.e., to logical '1' that does not affect the result, see function NX.
- On the contrary, OR corresponds to the maximum of inputs, so we can connect its unused inputs to the minimum, i.e., to logical '0' that does not affect the result, see function RX.

Important note: If you leave any unused input unconnected (floating), then it is always a serious error in your design⁵. Development tools for circuits try to correct such omissions. They announce floods of warnings, and they automatically connect floating inputs to '1' or '0'. However, they may not always hit the appropriate values, and the following search for such unconnected errors can be greedily "time-consuming". ☹

⁴ Evaluations of individual logic functions N_x or R_x have different maximum length of paths between input and output, so we get the results with different time delays. Usually, it does not matter. We are interested only in correct results. Situations when we must take into account times of evaluations are beyond the scope of this publication. They will be discussed in lectures.

⁵ Even if unconnected (floating) input does not affect the operation of a circuit, it increases electrical noise in it. The phenomenon will be explained in very advanced lectures.

2.7.1 Bubbles of negations

In logical diagrams, inverters are often not inserted as full schematic symbols, but they are reduced to small circle or bubble. Symbols for logic functions NAND and NOR, which were shown in Figure 9 on page 15, consist of functions AND and NOT, respectively OR and NOT, but they are abbreviated. Instead of drawing entire inverters (NOT), we append only bubbles to outputs of AND, or OR respectively, as negation signs, see Figure 14.

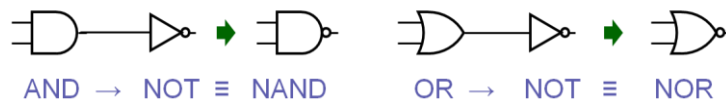


Figure 14 - Gates NAND and NOR

Since the following law holds: $\text{not}(\text{not } a) = a$, in the words, two negations cancel each other out, then two subsequent bubbles of negations are also cleared as Figure 15 shows.

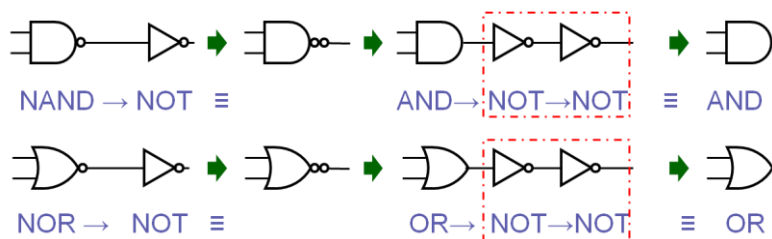


Figure 15 - Double negation

Bubbles of negations are also used as outputs and inputs, see Figure 16 bottom right, where is drawn equation $X = (A'.B)'$ by one gate with bubbles:

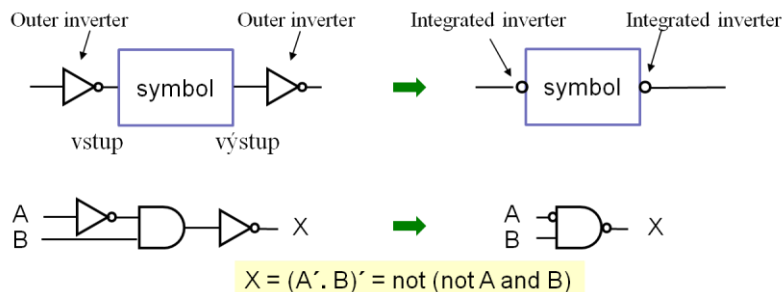


Figure 16 - Bubble of negations for inputs and outputs

2.7.2 Implementation of logical diagrams

In earlier times, logical diagrams also served for the direct implementation of logic functions. Graphical symbols were implemented by circuits called logic gates, which performed operations directly OR, AND, NOT, NAND, NOR, XOR, and others. Logical diagrams had represented a sort of construction plans of entire circuits.

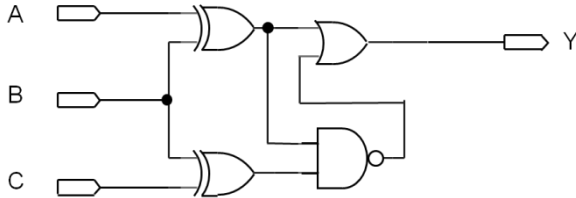
The development of modern circuits have moved this way to history, so it slowly became almost as rare as, for example, circuits with vacuum tubes. Today, logic is overwhelmingly implemented by programmable logic arrays, and the name "gate" remains mainly as a synonym for logical operations.

Logical diagrams are still frequently used for their clarity. For smaller logic function, they easier display dependencies of outputs on inputs and the cooperation of logic functions with advanced logic circuits.

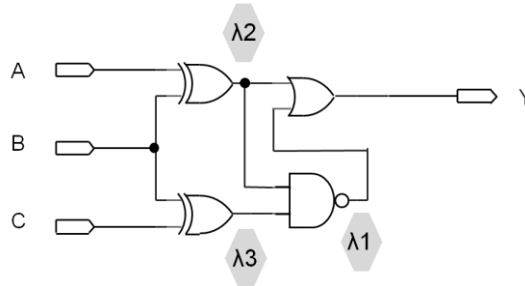
2.7.3 Conversion of logical diagram to expression

To conclude this section, we convert a logical diagram to a logical expression. Because the diagrams show the evaluation process of a logic function, so we can just follow the paths and write performed operations as logical operations. We demonstrate it on an example.

Example - Write logical expression corresponding to the logical diagram below.



Solution: We can begin either from the left side of the diagram, in the direction of its calculation, or vice versa from the end. We show the second method, which is more versatile. It can also convert very complex circuits with internal loops. First, we label outputs of the blocks.



The diagram contains XOR gates so that we write operators by words. Output Y is:

$$Y = \lambda 2 \text{ or } \lambda 1 \quad (\text{eq1})$$

Intermediate result $\lambda 1$ consists of AND operation with output bubble of negation, so $\lambda 1 = \text{not } (\lambda 2 \text{ and } \lambda 3)$. We substitute $\lambda 1$ into (eq1):

$$Y = \lambda 2 \text{ or not } (\lambda 2 \text{ and } \lambda 3) \quad (\text{eq2})$$

Now, we evaluate $\lambda 2 = A \text{ xor } B$ and substitute it into (eq2) instead of two $\lambda 2$

$$Y = (A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } \lambda 3) \quad (\text{eq3})$$

Finally, we get $\lambda 3 = B \text{ xor } C$ and substitute it into equation (eq3), which gives the result:

$$Y = (A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } (B \text{ xor } C)) \quad (\text{eq4})$$

We can rewrite the equation (eq4) with the aid of other symbols for operators AND, OR, and NOT. We leave XOR written by its name

$$Y = (A \text{ xor } B) + ((A \text{ xor } B) \cdot (B \text{ xor } C))'$$

~o~

Here, we close the basics of logic functions. Try to solve the test in the next part.

However, for the understanding of more complex operations such as adders or counters, you still need to know the basics of internal coding of integers, types of signed and unsigned integer, and hexadecimal notation, BCD numbers, and ASCII character encoding. Most students have certainly met with these concepts in programming courses, but we rather repeat them in the next chapter.

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.

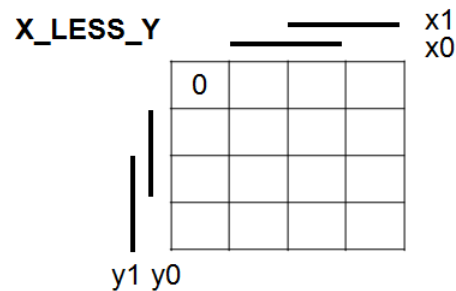
Question 1: Fill unfinished truth tables of logic functions:

x3	x2	x1	AND(x1,x2,x3)	OR(x1,x2,x3)	NAND(x1,x2,x3)	NOR(x1,x2,x3)
0	0	0				
0	0	1				
0	1	0	0			
0	1	1		1		
1	0	0			1	
1	0	1				0
1	1	0				
1	1	1				

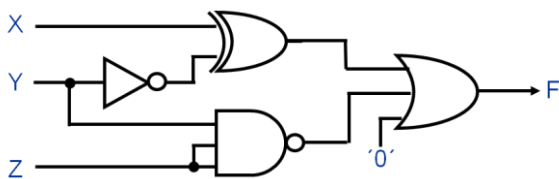
x2	x1	XOR(x1,x2)	EQU(x1,x2)
1	0		
1	1		
0	0	0	
0	1		0

Question 2: Rewrite the table left as Karnaugh map.

x1	x0	y1	y0	X_LESS_Y
0	0	0	0	0
0	0	0	1	1
0	0	1	-	1
0	1	0	-	0
0	1	1	-	1
1	0	0	-	1
1	0	1	0	0
1	0	1	1	1
1	1	-	-	0



Question 3: Write the logical expression that is realized by the following logic diagram:



$F(X,Y,Z)=\dots\dots\dots$

Question 4: Draw the logical diagram of the following logic function:

$$G(X,Y,Z) = \text{not}(\text{not } X \text{ xor } Y) \text{ and } \text{not}(\text{not } Y \text{ or } Z)$$

3 Integers expressed in binary system

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

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

Its punchline is based on the fact that 1000 in a binary system is always decimal 8. 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 to the left.

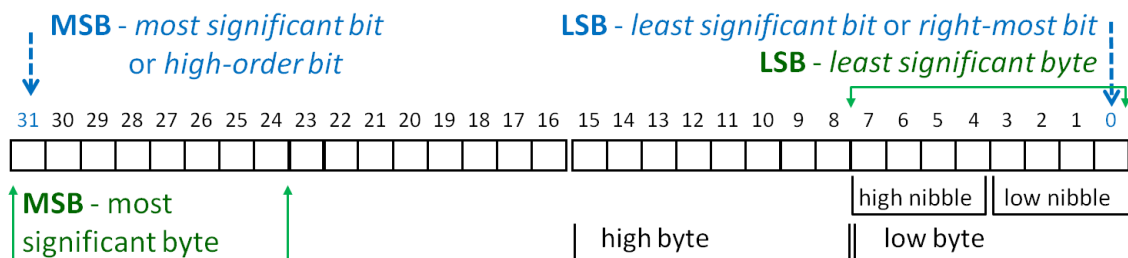


Figure 17 - Byte, bit, MSB, LSB

Word "**bit**" originally means a small quantity of food. 4 bits are sometimes called "**nibble**" (a trifling quantity of food). The size of 8 bits is known as "**byte**" whose origin comes 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 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 mistakenly⁶. For example, "Apollo Guidance Computers" used in the flight to the Moon have 15-bits word.

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

⁶ 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 standard IEC 1131-3, however, relates only to PLCs, it does not apply elsewhere.

3.1 Unsigned binary

Unsigned binary, by the whole name "binary encoded unsigned integers", represents 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:

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

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 $x \approx b_{m-1} b_{m-2} \dots b_1 b_0$ of m -bits is called a **binary encoded unsigned integer**, hereinafter an **unsigned binary**, and it has a value:

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

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

$$\begin{array}{ccccccc} 1*2^6 & + & 1*2^5 & + & 0*2^4 & + & 0*2^3 & + & 1*2^2 & + & 0*2^1 & + & 0*2^0 & = \\ \mathbf{64} & & \mathbf{32} & & & & & & \mathbf{4} & & & & & =\mathbf{100} \end{array}$$

The equation (1) ensures that there is exactly one combination of some members of 2^N series, whose sum gives the number, and each member of the series 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

3.1.1 Changing bit width of numbers

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

3.1.2 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 its logical left shift to 1010 that has double decimal value, i.e. **10**. Each 1-bit was in fact moved below the next member of 2^N series with a double value. Similarly, 10100 with appended two 0 bits has quadruple decimal value, i.e. **20**, and 101000 is eight times of the original value, i.e. **40**.

		$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
$8*5=\mathbf{40}$	$=32+8$	1	0	1	0	0	0
$4*5=\mathbf{20}$	$=16+4$		1	0	1	0	0
$2*5=\mathbf{10}$	$=8+2$			1	0	1	0
$\mathbf{5}$	$=4+1$				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 $5 * 2^5 = 5 * 32 = 160$. Another logical shift left gives result 01000000 that 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 3.1.5.

Programming languages based on the C language defines bit left shift operator \ll ⁷ that is 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: $2 * x == (x \ll 1)$, or $4 * x == (x \ll 2)$, and so on up $32 * x == (x \ll 5)$.

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 that 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	-> 1
	the remainder	1						

In C language, operator \gg is only a partial implementation of logical left shift because the operator does not give the remainder. For variable byte $x = 5$; it can be used only as integer dividing by 2. It holds: $(x \% 2) == 1$, $(x \gg 1) == 2$ and $x / 2 == 2$.

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

3.1.3 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 $Q = 1111110110$, then its conversion by the sum would be more challenging. Our Q has length of 11 bits, and the most of them are 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 by 1 greater than the sum of all previous members. So we know that:

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

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

$$Q = 1111110110 \rightarrow 2047 - 2^3 - 2^0 = 2047-8-1 = 2038$$

⁷ In language C ++, bit shift operators \ll and \gg 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.

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 our result that is 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) to the result. We repeat the process until the lowest bit is added.

$$1 \rightarrow 1*2+0=2 \rightarrow 2*2+0=4 \rightarrow 4*2+1=9 \rightarrow 9*2+1=19$$

Another example, in abbreviated notation:

11111110110

$$1 \rightarrow 2+1=3 \rightarrow 6+1=7 \rightarrow 14+1=15 \rightarrow 30+1=31 \rightarrow 62+1=63 \rightarrow 126+1=127 \\ \rightarrow 254+0=254 \rightarrow 508+1 \rightarrow 1018+1 = 1019 \rightarrow 2038+0=2038$$

We cannot recommend method 2 for hand calculations, based on our experience. The method 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 3.5.2, page 41.

3.1.4 Conversion of decimal number to unsigned binary

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

3.1.4.1 Repeated subtractions

We found the largest member of 2^N series that is still 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	new decimal	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 decimal is bigger than the member of the series, so we write 1 bit, we subtract the member. Otherwise, we write bit 0 and we try the 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 for converting of large decimals.

3.1.4.2 Repeated divisions by 2

The method is derived from logical right shifts, Chapter 3.1.2 on page 23. We divide given decimal number by 2 until the quotient becomes zero. We write down the remainders after integer divisions from the least significant bit (LSB) to the most significant bit (MSB).

The algorithm ends after obtaining quotient 0. In the case that we are converting a decimal number greater than 0 then the remainder of the last division is always 1, which is the leftmost bit of 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 algorithm more briefly. We just divide a decimal by 2, and we retrospectively determine remainders from intermediate results. Odd results have remainders 1.

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

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

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

Note that it would not mind if we have also included the last $\rightarrow 0$. In that case, we have obtained a binary number **01111101000** that has the same value, see paragraph 3.1.1.

3.1.5 Arithmetic overflow during additions and subtractions

Computers and digital circuits always store a finite number of bits. If we are adding 1 to a number, then the number eventually reaches its maximum value. For unsigned binary, the maximum contains only 1 bits. Their count is given by bit length of a binary number.

	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 3 - 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, in fact, equals to 0.

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

When we are subtracting 1, the overflow can also occur. We can imagine the subtraction as a progression from the bottom up in Table 3. 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 a bit from a higher order.⁸ However, the both directions are frequently called as Carry.

⁸ 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.

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. The bit width only determines the decimal value of this wrong result of the subtraction zero minus one, (0-1). For example:

for 8 bit unsigned binary, $(0-1) = 2^8 - 1 = 255$,

for 9 bit unsigned binary, $(0-1) = 2^9 - 1 = 511$

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

The result of the subtraction 0-1 in decimal counting is -1. The value of overflow result for unsigned binary numbers of limited length is determined by correcting 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 18. Addition operations correspond to rotating the wheel counter-clockwise 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 ($4-14 = 6$).

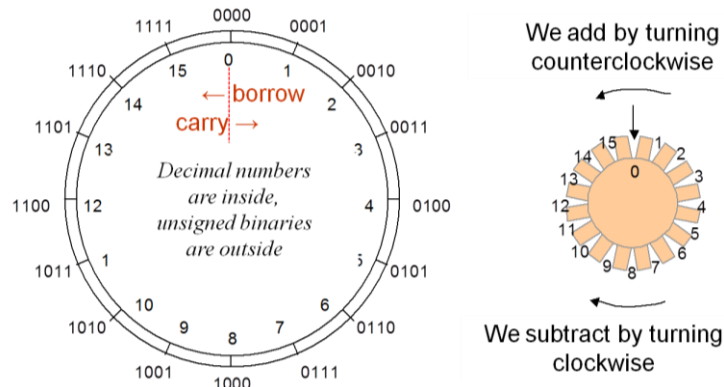


Figure 18 - Adding and subtracting unsigned binary

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.

3.2 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 3.1.5.

If we have unsigned binary x stored in m -bits, we can create its **one's complement** χ by negating of 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 , then 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, then decimal number 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 then, we get 1100 ($\chi + 1$) which is two's complement of 0100. The sum $0100 + 1100 = 10000$. The result 10000 considered as unsigned binary has 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**, hereinafter referred to **signed binary**, as:

- The negation of 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 in Figure 19 to the right. Signed binary 1000, with decimal value -8, has here 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 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).

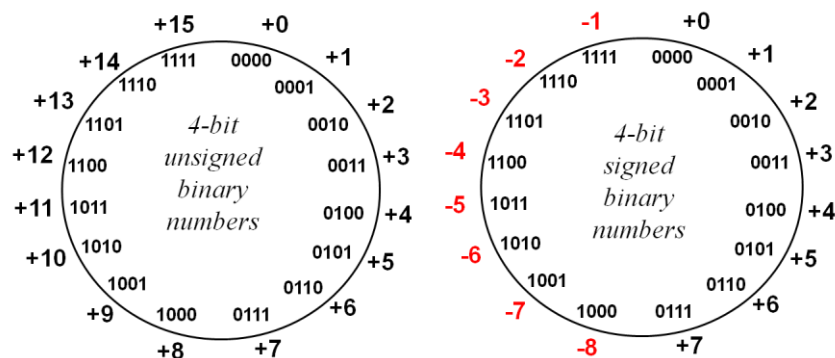


Figure 19 - 4-bit unsigned and signed binaries

Otherwise, signed binaries provide only benefits. We calculate their additions and subtractions by 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 integer in two's complement) are the ultimate way for storing signed integer numbers (in C, signed int type).

3.2.1 Important properties

Signed binary have significant properties worth remembering.

- Number 0 is always coded by all bits 0.
- If we denote the bit length as m , then we can convert decimal numbers 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 having 1 followed by $m-1$ bits 0 is always the least number, and its decimal value is equal to -2^{m-1} . This number is also the only anomaly - its positive counterpart does not exist in given bit length. For example, if we have 8-bit signed binaries, then their least binary is 10000000 with decimal value $-2^{8-1} = -2^7 = -128$.
- Signed binary having 0 followed by $m-1$ bits 1 is always the greatest number and its decimal value equals to $2^{m-1} - 1$. For example, if we have 8-bit signed binaries, then their greatest binary is 01111111 and its 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 having all bits 1, i.e., m bits 1, is always equal 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 1 than -1.*

3.2.2 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, having decimal value 100.

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

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.

3.2.3 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 3.2.1

- 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) is equal to 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 the way in which 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 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 $11110011+00000001 = \mathbf{11110100}$.
- Calculation by b): $|-12|-1 = 11$. Decimal 11 as 8-bit unsigned binary is 00001011. Then, we perform one's complement of the result: $00001011 \rightarrow \mathbf{11110100}$.
- Calculation by c): $2^8-12 = 256-12 = 244$. Decimal 244 converted to 8-bit unsigned binary is 11110100. The same number **11110100** taken as 8-bit signed binary has decimal value -12.

3.2.4 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 3.2.3.
 - a) First, we calculate two's complement of the signed binary, and we converted it as unsigned binary to number, which we denote x . Finally, we change its sign to minus, so $-x$.
 - b) 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: $-x = -y-1$.
 - c) Alternatively, we can convert the entire signed binary as an unsigned binary to the number that 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 the binary by the same ways 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 unsigned binary to 52. The searched result is its negation, so **-52**.
- Calculation by b): We evaluate one's complement of 11001100 \rightarrow 00110011. Then, we convert it as unsigned binary: $00110011 \rightarrow 51$. The searched result is $-51 - 1 = -52$.
- Calculation by c): We convert signed binary 11001100 as unsigned binary, for example, by adding weights of 1-bits: $128 + 64 + 8 + 4 = 204$. The searched result is $204 - 2^8 = 204 - 256 = -52$.

3.2.5 Change of bit length - sign extension

In front of an unsigned binary number, we can add 0 bits without changing its value, see 3.1.1. To preserve the value of a signed binary number must utilize sign extension which maintains the most significant specifying whether the number is positive or negative.

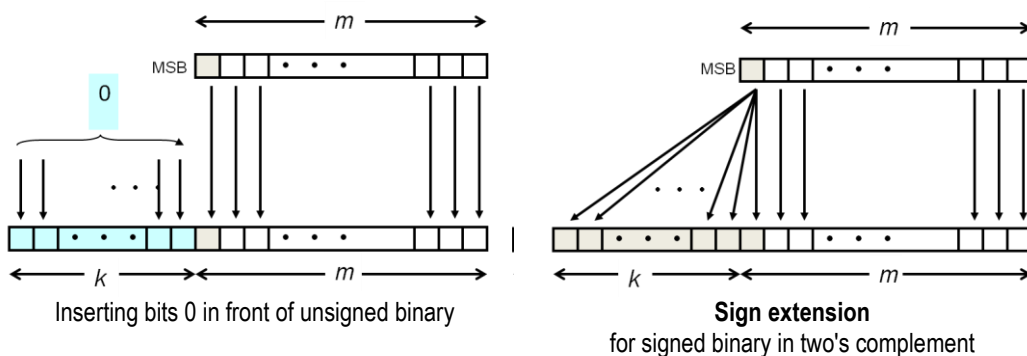


Figure 20 - Signed extension

Figure 20 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 size to larger one. For example, MIPS CPU loads 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.

3.2.6 Logical and arithmetic shifts

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

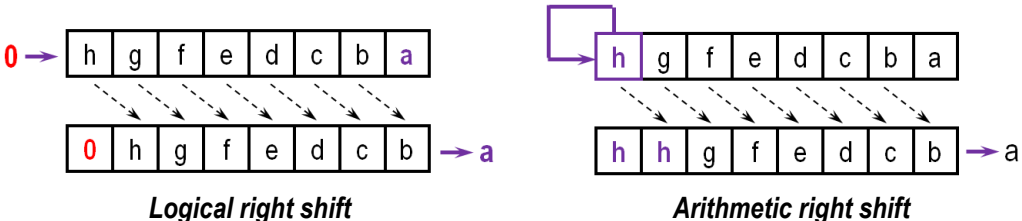


Figure 21 - Logical and arithmetic right shifts

Figure 21 depicts right shifts for 8-bit binary "hgfedcba" in which 'a' is LSB and 'h' is MSB. If a binary represents unsigned format, we perform right shifts by inserting 0-bits. If we have a signed binary, then we utilize arithmetic right shifts, in which the highest bit (MSB) remains firmly in its place, here 'h'.

For example, 8-bit string 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. $-21/2 = \text{floor}(-10.5) = -11$, where floor() denotes rounding down.⁹ Arithmetic right shifts interpreted as a division by 2 give contradictory results, for example $-1/2 = -1$.

⁹ In C, we have floor(...) function, floor(...) method in Java, and Math.Floor(...) method in C# .

If we replace integer dividing by 2 with the aid of arithmetic right shifts, then we must sometimes correct results of negative binaries to receive expected number. The corrections are simple; we increment the results by 1 in selected cases.¹⁰ Therefore, we should remember that the arithmetic right shift is not exact analogy integer division by 2 for signed binaries.

Left shifts of binary numbers are the same - the logical left shift is performed as arithmetic. Figure 22 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 no an arithmetic overflow, then a left shift corresponds to the multiplication by 2 for unsigned and signed binaries.

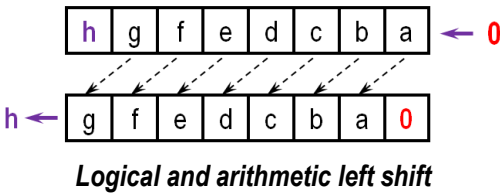


Figure 22 - Left shift of 8-bit binary

The two representations of binaries differ only in the condition where arithmetic overflow occurs during shifts, so the result value does not correspond to the original decimal value multiplied by 2. For an unsigned binary number, the arithmetic overflow occurs at the moment when the highest bit, which is lost when shifting, equals to 1. However, signed arithmetic overflow occurs at moment 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 11010110 (-42) and the second 10101100 (-84). After performing third left shift, we obtain 01011000 (decimal 88). The overflow has occurred.

Another example: If we have 8-bit signed binary 00110010 (decimal 50), then its first left shift result is 01100100 (decimal 100). The second shift gives 11001000, 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 in the case of the same entry taken as an unsigned binary, because its decimal value is 200. The overflow would appear here after its third left shift whose result will be 10010000 with 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.

¹⁰ 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, we sometimes must perform some corrections. However, the processors often include a special unit for the multiplications of negative signed binaries to speed up calculations. Negative signed binary numbers close to zero tend to have a lot of bits 1 and their multiplication could last too long. Detailed descriptions are beyond the scope of this publication. For more information, search "Booth's algorithm" on Wikipedia

3.2.7 Arithmetic overflow for addition and subtraction

In Chapter 3.1.5 on page 26, where we have discussed the addition and subtraction for unsigned binary, we have detected the overflow by Carry, i.e. the carry to a higher order. 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 3.2 on page 28.

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 we subtract 1 from -128, we receive 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 4 - Arithmetic overflow for addition of 8-bit signed binaries

ALU of a processor detects similar situation 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 5 - Conditions for overflow of signed binaries

To be exact, ALU sets after each addition and subtraction of numbers, at least four basic arithmetic flags¹¹ 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 position¹²;
- Overflow - overflow for signed binary numbers;
- Sign - the result of operation is negative;
- Zero - the result was zero.

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

¹¹ 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.

¹² ALUs set Carry also for shifts, but higher programming languages do not implement possibility to use Carry with shifts.

3.3 Signed integer in straight binary and offset binary

Other coding methods exist for signed integers. For completeness, we mention two very general methods, namely 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 straight binary or sign and magnitude representation abbreviated as sign-magnitude.

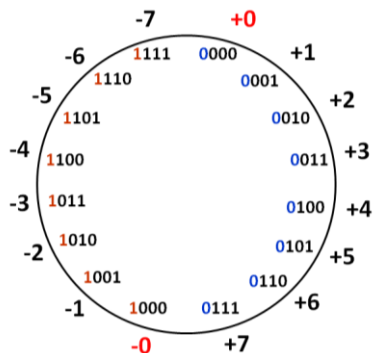


Figure 23 - Straight binary (Sign-magnitude)

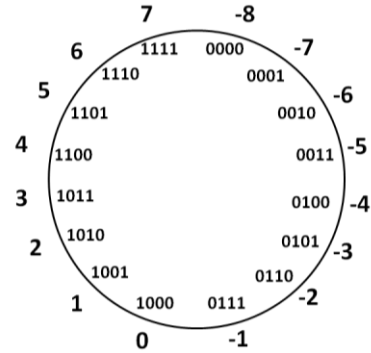


Figure 24 - Excess K with K=8

Figure 23 shows straight binaries for 4-bit length. In the code, there are two zeros, negative and positive. For example, if our result reaches to zero during the iterations, we know the direction of approaching zero. Another advantage of the code is very fast arithmetic negation, which we create it only by 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 by such way that the results are always positive numbers. Then, we encoded them as an unsigned binary number.

If we choose of m -bit length binaries and a fixed constant K , then 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 $K=8$, then we have the range from -8 to 7, see Figure 24. We can select the range of by any values of K . For example if we take $K=30$, then 4-bit binaries have the 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 $(x + y) + 2 * K$. It means that 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 en-

coded according to the IEEE 754 standard used for types float, double and extended by overwhelming majority of modern computers. IEEE 754 stores mantissa in straight code and exponent in K-excess code¹³.

3.4 Hexadecimal notation

Hexadecimal notation is a shorthand way of writing binary strings. Each 4 bits are encoded as 4-bit unsigned binary. To maintain single character representation for each group, values from 10 to 15 are replaced by letters from A to F.

Example 1: Write binary string 10100111 in hex. notation.

Solution: We divide the string 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 length divisible by 4, but it is possible to use it for other lengths if we add the bit length specification.

3.4.1 Hexadecimal number

Hexadecimal number means that the bit string written in hexadecimal notation is interpreted as an unsigned binary number. There are several different ways for writing such numbers. 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 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) as can also be taken as a signed binary number, if it is assigned to a variable of a signed type. The situation is demonstrated on the following C ++ code compiled so that the variable `int` has 4 bytes.¹⁴

¹³ 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.

¹⁴ 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`.

3.4.2 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.

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

The value of hex number $x_{16}=0xA7$, we evaluate by (3) 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:

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

The values a_k are numbers, but we write them by single characters, as well as in hex notation. The circuits and computers prefer numbers with radices 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
<i>Decimal</i>	<i>10</i>	<i>- no possibility to convert by bit groups-</i>

Table 6 - Some conventional radices for numeral systems

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

3.4.2.1 Octal numbers

If we select $r=8$ in equation (4), then 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).

3.4.2.2 *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 the both encoding. They are suitable for efficient transmissions of long binary strings in text form, as encryptions keys. Activation keys of programs are also often encoded as Base32 numbers.

Base32 and Base64 encoding and are not as transparent as hexadecimal numbers because they rarely represent numerical values according to the formula (4). Encoded bit groups with the 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. If we encoded separate numbers that have byte sizes, we do not achieve efficient compression into text. Therefore, all numbers are often joined into one long binary chain, i.e. to one many-bit number that is 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 for avoiding possible confusion with similar symbols, such as the lowercase letter l and the 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 that the length of the last group and it is also useful 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
• <i>its bit groups</i>	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

3.5 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 number 0011 0101 in BCD.

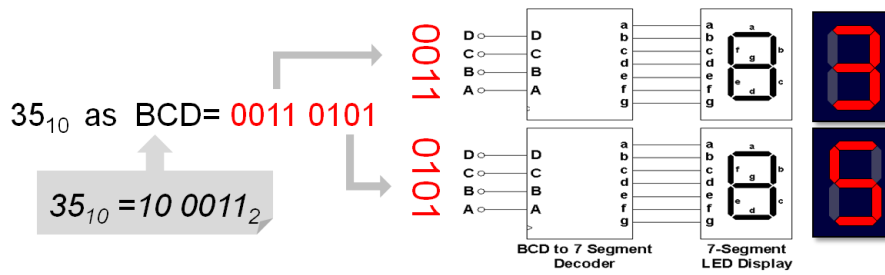


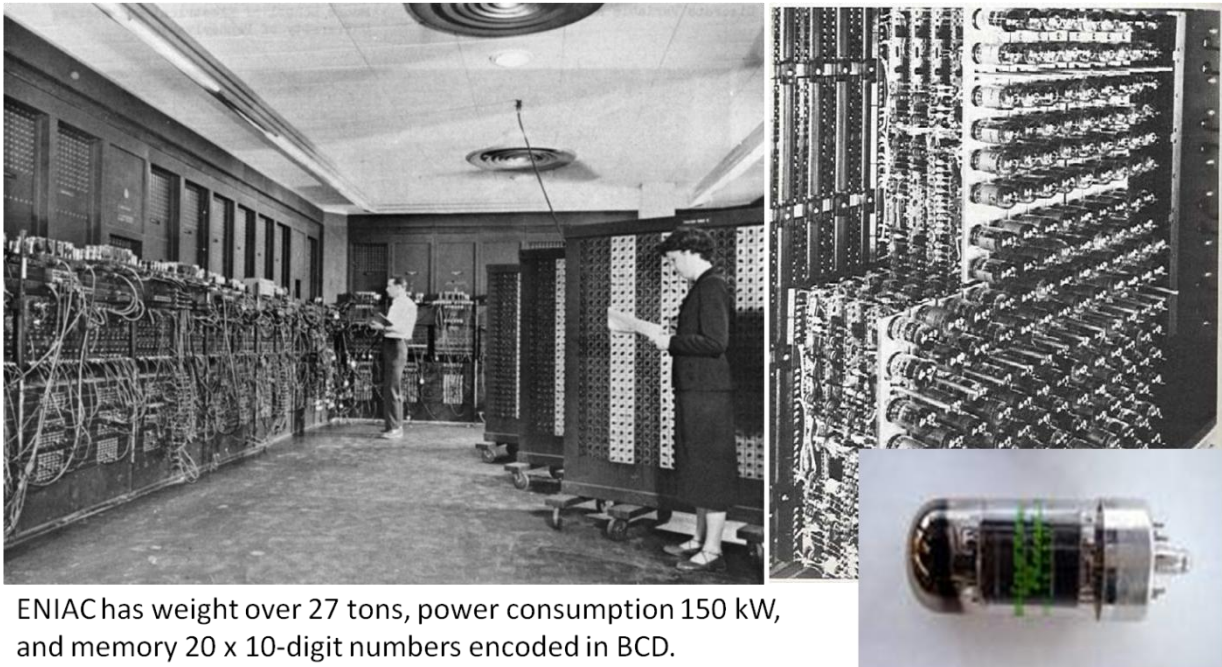
Figure 25 - BCD číslo 35

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 25. Similarly, when we print numbers by calling printf () the function, then it first converts numbers to BCDs and then, it writes characters of digits.

BCD contains 4-bit groups, as well as hexadecimal numbers, but unlike them, 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 group of BCD contains a value that is outside this range, then it is not valid BCD number. When we encode a decimal number 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 26 [photograph from Wikipedia, key ENIAC].



ENIAC has weight over 27 tons, power consumption 150 kW, and memory 20 x 10-digit numbers encoded in BCD.

Figure 26 - 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, and it is suitable for numerical operations. Packed BCD format stores in one byte two BCD digits and it is effi-

cient for storing numbers. Microprocessors can usually perform arithmetic operations only with unpacked BCD numbers.

Today, the direct counting with BCD numbers is performed only rarely, because it is about 2 to 3 times slower than binary numbers and it also requires more accesses into memories. However, BCD is necessary for each printing of numbers.

We can convert a binary number to BCD of course by divisions by ten and by counting remainders, but such 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 3.1.3, beginning on page 24. Moreover, the method can directly convert packed BCD numbers, and it is suitable for parallel realization in hardware. For it, we need only the operation of BCD multiplication by 2.

3.5.1 How to multiply BCD by 2

When a BCD number contains digits from 0 to 4, then 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 we multiply them by 2, we obtain the results from 10 to 18 that are outside of the valid BCD range.

We can correct them by the following way. Before we perform the left shift of a BCD, we add 3 to all BCD digits that have values from 5 to 9.

Why are we adding 3? It is a half of the length range that is 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 add +3 to its BCD digits that are 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, but these are immediately converted by the logical left shift to valid BCD values.

3.5.2 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), then we add binary 3 (0011) to it. 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 the both 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 1 st joined left shift BCD and binary	[0 0 0]	0000 0000 0000 11100110
after 2 nd joined left shift BCD and binary	[0 0 1]	0000 0000 000 1 11001100
after 3 rd joined left shift BCD and binary	[0 0 3]	0000 0000 00 11 10011000
after 4 th joined left shift BCD and binary	[0 0 7]	0000 0000 0111 00110000
<i>BCD digit 0111 > 0100</i>	<i>+[0 0 3]</i>	<i>+0000 0000 0011</i>
<i>result of the correction</i>	[0 0 10]	0000 0000 1010 00110000
after 5 th joined left shift BCD and binary	[0 1 4]	0000 0001 010 0 01100000
after 6 th joined left shift BCD and binary	[0 2 8]	0000 0010 <u>1000</u> 11000000
<i>BCD digit 1000 > 0100 → correction</i>	<i>+[0 0 3]</i>	<i>+0000 0000 0011</i>
<i>result of the correction</i>	[0 2 11]	0000 0010 1011 11000000
after 7 th joined left shift BCD and binary	[0 5 7]	0000 <u>0101</u> <u>0111</u> 10000000
<i>BCD digits 0101 a 0111 ≥ 0100 → correction</i>	<i>+[0 3 3]</i>	<i>+0000 0011 0011</i>
<i>result of the correction</i>	[0 8 10]	0000 1000 1010 10000000
after 8 th joined left shift BCD and binary	[1 1 5]	0001 0001 0101 00000000
the end - BCD contains the final result		

The algorithm is relatively straightforward and mechanical, and we can extend it to longer binaries. Unfortunately, in higher programming languages, we cannot easily shift several joined numbers. Machine code instructions allow these operations because ALUs store a bit that falls out during a shift into processor Carry flag. The Carry can be immediately used as the input for the next shift, so we can easily shift arbitrarily long chain of numbers.

In C language or other similar high-level programming languages, shifts are implemented only partially without the possibility of shifting multiple variables. Processors can perform, but we cannot simply write.

The most efficient way for their programming leads through the placement of BCD and unsigned binary into a single *unsigned int* or *unsigned long* type, according to the required length. Then, we have a little bit complicated tests of BCD digits, but our program is still more efficient than a code with shifts of separated variables.

```

typedef unsigned char byte;
struct BCD_t { byte digit[3]; };
void byte2bcd(byte bin, BCD_t & bcd)
{
    // 3 BCD digits are stored in bits 19 down to 8; binary input bin in bits 7 down to 0
    unsigned int BCD_plus_bin = bin;
    for (int i = 1; i <= 8; i++) // counter of bit shifts
    {
        //For 8bits input bin, the correction is only performed for lower BCD digits [1] and [0],
        // because the upper BCD digit [2] can contain only values from 0 to 2
        if ((BCD_plus_bin & 0xF00) > 0x400) BCD_plus_bin += 0x300;
        if ((BCD_plus_bin & 0xF000) > 0x4000) BCD_plus_bin += 0x3000;
        BCD_plus_bin <<= 1; // shift left together BDC + bin
    }
    // unpacking BCD digits to obtain result
    bcd.digit[0] = (BCD_plus_bin & 0xF00) >> 8;
    bcd.digit[1] = (BCD_plus_bin & 0xF000) >> 12;
    bcd.digit[2] = (BCD_plus_bin & 0xF0000) >> 16;
}
int main(int argc, char * argv[]) // a test of byte2bcd function
{
    byte VSTUP = 0x73; // bin=01110011
    BCD_t bcd;

    byte2bcd(VSTUP, bcd);

    for (int j = 2; j >= 0; j--) // direct print of characters
        putchar((char)(bcd.digit[j] + '0'), stdout);

    putchar('\n', stdout); // end of line

    return 0;
}

```

-
- Note 1: The code can run faster because the corrections for the lowest BCD digit [0] occur only when $i \geq 4$, and for the following BCD digit [1] when $i \geq 7$.
- Note 2: We explain the converting a number to character by adding '0' in the next section about ASCII characters, in paragraph "Important properties of ASCII".
-

3.6 Character encoding standard ASCII

The character encoding standard ASCII (American Standard Code for Information Interchange) was developed in 1963, then, it was revised. Its most recent update during 1986 is utilized until today. It is also incorporated into the newer codes, as Unicode or UTF8, which assigned 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 7 on page 43. 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	!
Hexadecimal	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

Table 7 - 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	

Important properties of ASCII

- Control characters - The character with codes from 0 to 31 are reserved for controlling devices, like printers or teletypewriters. Language C contains escape code for the main control characters with values from 7 to 13 frequently used until today. In C, the escape codes begin by a backslash. Table 7 emphasizes them by 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 and its numerical value by subtracting or adding 48 (0x30), which is 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 easily test whether a character is a letter and sorted them alphabetically.
- Lowercase and uppercase character codes are always shifted 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.

3.6.1 Extended ASCII

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

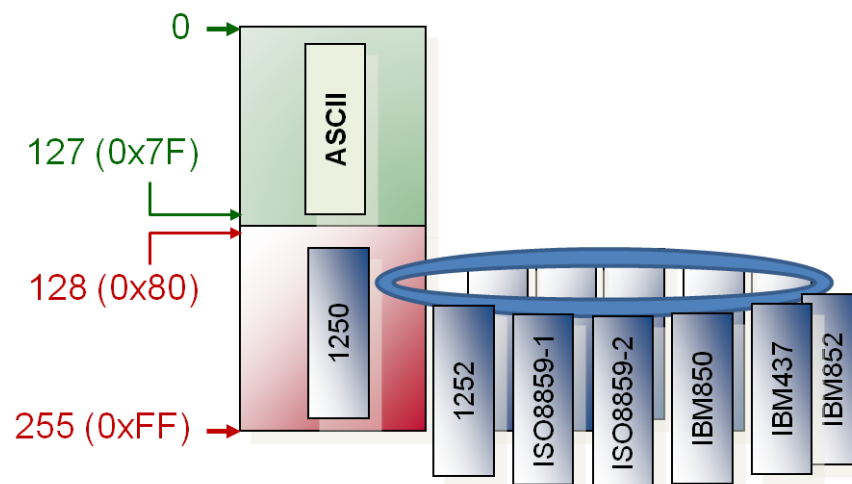


Figure 27 - Principle of extended ASCII

Figure 27 demonstrates the principle of the extension. While the lower decimal values from 0 to 127 remains constant, defined by the ASCII standard, so 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, and it still far

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.

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

In language C, if we work with extended ASCII characters, we have to remember that the extension code page uses values from 128 to 255 (from 0x80 to 0xFF). These are signed 8-bit number 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.

Very frequent mistake is the skipping whitespaces characters with values 0x8 from 0x20 by their comparing with 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='l'
```

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

We correct the code above by using unsigned char types. The, accented 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 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 allow storing all the world's national characters including historic scripts and the most of symbols. To values 0 to 0x7f, Unicode assigns the same characters as ASCII.

For storing texts and websites, it is increasingly utilized UTF-8 (Unicode Transformation Format) that has maximum compatibility with ASCII, because its character values from 0 to 0x7f codes are identical to ASCII. UTF8 utilizes codes from 0x80 to 0xFF for the indication of the beginnings of more byte sequence 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 characters¹⁵ added to Unicode for different typographic spacings and line spacings. Many simple programs for processing texts freely apply Unicode, but they ignore its additional characters and silently ☺ assume that input texts do not contain them.

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

¹⁵ 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".

3.7 How much is 1000?

We have begun Chapter 3 by the following joke:

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

Thus, how much it is 1000 in different numeral systems? Everything depends on used 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 are decimal number,
- = **4096** from hexadecimal number,
- = **to an arbitrary integer** from K-excess code according to its K offset value.

3.8 Test from knowledge of Chapter 3

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.

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

Decimal number	10-bit signed binary		Straight binary - sign-magnitude	
	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 `creresult` of the program in C language?

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


4 Appendix

4.1 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 3.6 on page 42.
- BCD** Binary Coded Decimal - encoding of a decimal number for good comprehensibility of human and easy visualization, see Chapter 3.5 on page 38.
- biased representation** - the other name of *Excess-K* encoding of numbers, see Chapter 3.6.1 on page 44.
- Borrow** it means borrowing a bit from higher order during arithmetic overflow in the direction down, see Chapter 3.1.5 on page 26. 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 3.1.5 on page 26. 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 3.1.5 on page 20.
- Extended ASCII** - the extension of ASCII code, see Chapter 3.6.1 on page 44.
- logic gate** it was originally designated as an electronic logical element. Today, it often denotes the schematic symbol of a logic operation, see Chapter 2.7.2 on page 19.
- 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 17 on page 22.
- 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 17 on page 22.
- offset binary** - the other name for K-excess coding of numbers, see Chapter 3.3 on page 35.
- 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 3.6.1 on page 44.
- sign-magnitude** - the other name of straight binary coding of numbers, see Chapter 3.6.1 on page 44.
- signed binary** - the abbreviation for the binary coding of signed integers in two's complement, see Chapter 3.2 on page 28.
- unsigned binary** - the abbreviation for the binary coding of positive integers, see Chapter 3.1 on page 23.

4.2 Solution of test from Chapter 2

The solution is related to the test on page 21.

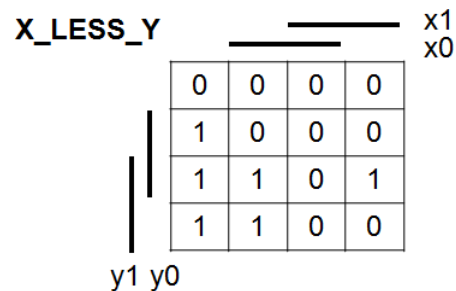
Question 1: Fill unfinished truth table logic functions::

x3	x2	x1	AND(x1,x2,x3)	OR(x1,x2,x3)	NAND(x1,x2,x3)	NOR(x1,x2,x3)
0	0	0	0	0	1	1
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	0	0

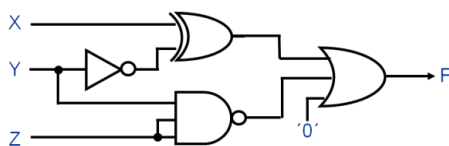
x2	x1	XOR(x1,x2)	EQU(x1,x2)
1	0	1	0
1	1	0	1
0	0	0	1
0	1	1	0

Question 2: Rewrite the table left as Karnaugh map.

x1	x0	y1	y0	X_LESS_Y
0	0	0	0	0
0	0	0	1	1
0	0	1	-	1
0	1	0	-	0
0	1	1	-	1
1	0	0	-	0
1	0	1	0	0
1	0	1	1	1
1	1	-	-	0



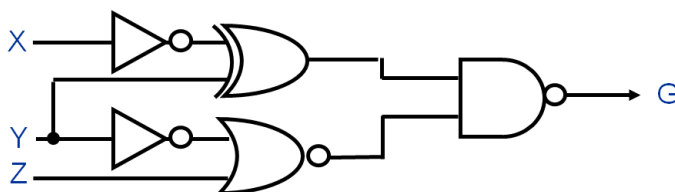
Question 3: Write the logical expression that is realized by the following logic diagram:



$$F(X,Y,Z) = (X \text{ xor not } Y) \text{ or not } (Y \text{ and } Z)$$

Question 4: Draw the logical diagram of the following logic function:

$$G(X,Y,Z) = \text{not} ((\text{not } X \text{ xor } Y) \text{ and not } (\text{not } Y \text{ or } Z))$$



4.3 Solution of test from Chapter 3

The solution is related to test on page 47.

Question 1 - Fill in missing values in tables.

Decimal number	8-bit unsigned binary		BCD číslo	
	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

Decimal number	10-bit signed binary		Straight binary - sign-magnitude	
	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 `creult` of the program in C language?

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

Author: Richard Susta
<http://susta.cz/>

Figures: * Figure 26 - 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, 2016

Length: 50 pages

Homepage of document:
http://dcenet.felk.cvut.cz/edu/fpga/doc/LogicFunctions_Prerequisite_V21.pdf