МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Gavrylenko S., Khatsko N.

FUNDAMENTALS OF COMPUTER SYSTEMS ARCHITECTURE

The study guide for the students of
121 – "Software Engineering" and 123 – "Computer Engineering"
for full-time and distance education

Харків
НТУ «ХПІ»
2019

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Gavrylenko S., Khatsko N.

FUNDAMENTALS OF COMPUTER SYSTEMS ARCHITECTURE

The study guide for the students of
121 – "Software Engineering" and 123 – "Computer Engineering"
for full-time and distance education

Затверджено
редакційно-видавничою
радою
НТУ «ХПІ», протокол № 2
від 17 травня 2019р.

Харків
НТУ «ХПІ»
2019

УДК 004.2; 004.6; 519.6
Г 12

Рецензенти:

О.С. Назаров, к.т.н., доцент, заступник декана факультету комп'ютерних наук Харківського національного університету радіоелектроніки;

Є.В. Танько, канд. пед. наук, доц. кафедри іноземних мов НТУ «ХПІ».

Розглянуто базові принципи архітектури компьютерних систем, питання подання інформації в різних системах числення, виконання логічних та арифметичних операцій. В кожному розділі наведена необхідна теоретична інформація, приклади представлення інформації та приклади виконання арифметичних та логічних операцій, приведені завдання для самостійного виконання та контрольні запитання.

Розраховано на студентів спеціальностей 121 – «Інженерія програмного забезпечення» та 123 – «Комп'ютерна інженерія» денної та дистанційної форм навчання.

Gavrylenko S., Khatsko N.

Г 12 Fundamentals of computer systems architecture / Gavrylenko S., Khatsko N. – Kharkiv : NTU "KhPI", 2019. – 75 p.

ISBN 978-966-8944-91-8

In the study guide "Fundamentals of computer systems architecture" the questions of presentation of information in different systems of calculation, execution of logical and arithmetic operations are considered. Each chapter provides the necessary theoretical information, examples of presentation of information and examples of execution of arithmetic and logical operations, given tasks for self-execution and control questions.

For the students of specialties 121 – "Software Engineering" and 123 – "Computer Engineering".

ISBN 978-966-8944-91-8

УДК 004.2; 004.6; 519.6

TABLE OF CONTENTS

INTRODUCTION

Nowadays, microelectronics, computer technologies and the entire computer science industry have become one of the main components of world scientific and technological progress. The influence of computer technology on all spheres of human activity continues to expand in breadth and depth. Computers are used not only for performing complex calculations, but also for managing production processes, education tasks, healthcare or environmental related issues etc. It's possible due to the fact that computers are able to process any type of information: digital, text, tabular, graphical, visual and audible.

Just like each building, each computer has a visible structure, referred to as its architecture. The architecture of a building can be examined at various levels of details, like the number of stories, the room size, the doors and windows location and so on. Take a look at a computer's architecture at similar levels of detail of basic hardware elements, which in turn depends on the type of the computer.

The methodical tutorial includes the basic questions of computer architecture, the arithmetic and logical foundations of the computer. In the first chapter of tutorial some important concepts related to computer architecture are introduced with special emphasis on those processor nodes that perform data conversion operations (simple arithmetic operations). The following chapters explain in detail the arithmetical foundations of computing. The study of various computer systems used in computing and arithmetical operations is very important for understanding how information processing is performed in computing machines.

The main aim of this methodical tutorial is to help students with the self-education. The examples and tasks discussed will help to effectively master the study material. Questions for self-monitoring are recommended for students to test knowledge.

1 INTRODUCTION TO COMPUTER ARCHITECTURE

Everybody knows what a computer is. The box that stands on your desk, runs your programs and, sometimes, crashes at the wrong time. Inside that box is the electronics that runs your software, stores your information, and helps to create a connection to the world. It's all about information processing.

Computer systems are divided into two separate categories. The first, and most obvious, is the desktop computer. The second category is the embedded computer, a computer that is integrated into another system with the purposes of controlling and/or monitoring. Embedded computers are far more numerous than desktop systems, but far less obvious. Each person on average has one or two computers. But, such a person may not know that he or she has 30 or more embedded computers, hidden inside TVs, VCRs, remote controls, washing machines, cell phones, air conditioners, game consoles, ovens, toys, and a lot of other devices.

This chapter describes computer architecture in general. It is applicable to both embedded and desktop computers, because the primary difference between an embedded machine and a general-purpose computer is its application. The basic principles of operation and the underlying architectures are fundamentally the same.

Both have a processor, memory, and, often, several forms of input and output. The primary difference lies in their intended use and in the user control degree over the loading and launching software. Desktop computers can run a variety of application programs, with system resources are orchestrated by an operating system. By running different application programs, the functionality of the desktop computer is changing. In contrast, the embedded computer is normally dedicated to a specific task. The embedded computer may or may not have an operating system, and rarely provide the user with the ability to arbitrarily install new software. Embedded hardware is often simpler than a desktop system, but it can also be far more complicated too. An embedded computer can be implemented in a single chip with just a few support components, and its purpose may be similar to a controller for a garden-watering system. Alternatively, the embedded computer may be a distributed

parallel machine with 150 processors, which is responsible for all the flight and control systems of a commercial jet. No matter how many diverse with embedded hardware excite, the underlying principles of design are the same.

This chapter introduces some important concepts related to computer architecture, with special emphasis on those processor nodes that perform data conversion operations (simple arithmetic operations).

Basically, a computer is a machine designed to process, store, and retrieve data. Data may be represented as numbers in a spreadsheet, characters in the text document, dots of color in an image, waveforms of sound, or the state of some system, such as an air conditioner or a player.

Everything that a computer does, from web browsing to printing, involves moving and processing numbers. The electronics of a computer is nothing more than a system designed to hold, move, and change the numbers.

A computer system is composed of many parts, both hardware and software. At the heart of the computer is the processor, the hardware that executes the computer programs. The computer also has memory, often several different types in one system. The memory is used to store programs while the processor is running them, as well as store the data that the programs are manipulating with. The computer also has devices for storing data, or exchanging data with the outside world. Such as allowing to input the text via a keyboard, to display information on a screen, or to move programs and data to or from a disk drive.

The processor is the most important part of a computer, the component around which everything else is centered. In essence, the processor is the computing part of the computer. The processor is an electronic device capable of manipulating data (information) in a way specified by a sequence of instructions. The instructions are also known as opcodes or machine code. This sequence of instructions may be altered to suit the application, and, therefore, computers are programmable.

The processor by itself is incapable to perform any task successfully. It requires memory (for program and data storage), support logic, and at least one input/output

device (I/O device) used to transfer data between the computer and the outside world. The basic computer system is shown in Figure 1.1.

Such computer architecture is known as a Von Neumann machine, named after John Von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form. Von Neumann computers can be named control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation.

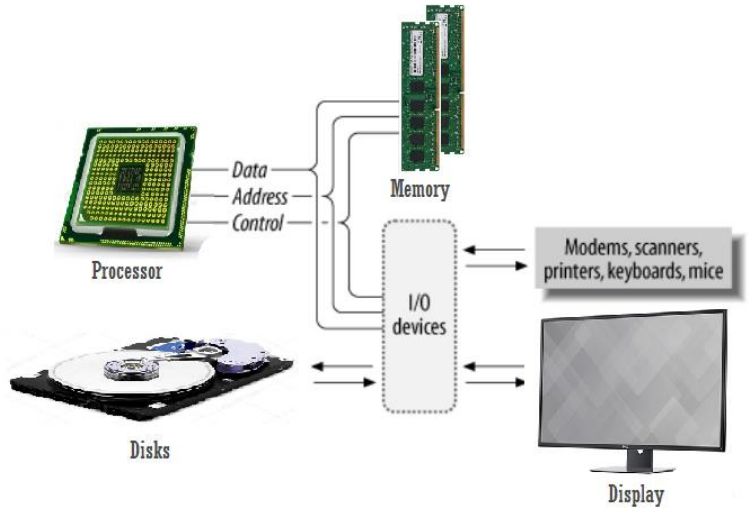


Figure 1.1 – Basic computer system

The memory of the computer system contains both the instructions that the processor executes and manipulates data. The memory of a computer system is never empty. It always contains something, whether it be instructions, meaningful data, or just the random garbage that appeared in the memory when the system powered up. Instructions are read (fetched) from memory, while data is both read from and written to memory, as shown in Figure 1.2.

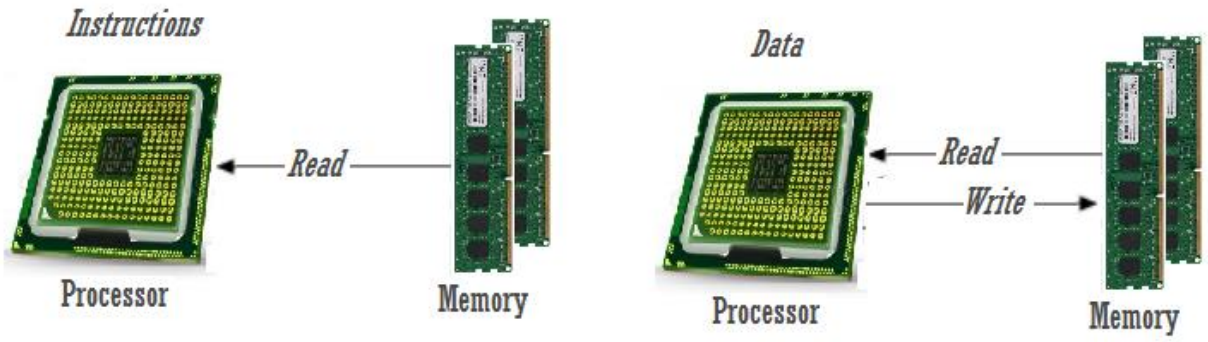


Figure 1.2 – Data flow

This form of computer architecture is known as a von Neumann machine, named after John von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form. Von Neumann computers are what can be termed control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation.

The classic von Neumann machine has several distinctive characteristics:

- there is no real difference between the data and instructions;
- the data has no intrinsic value;
- data and instructions share the same memory;
- memory is a linear (one-dimensional) array of storage locations.

There are some interesting non von Neumann architectures, such as the massively parallel Connection Machine [8], the dataflow architecture [5], the graph reduction machine [6] and the neural networks [7].

There are two main types of architectures: Princeton, often called the von Neumann architecture, and Harvard. The difference between them is that in the classic von Neumann architecture of the computer programs and data are stored in the common operating memory and transmitted to the processor on a single channel (data bus and control), while Harvard architecture requires the use of separate address spaces for storing commands and data, as well as separate transmission streams for commands and data (Fig. 1.2, 1.3).

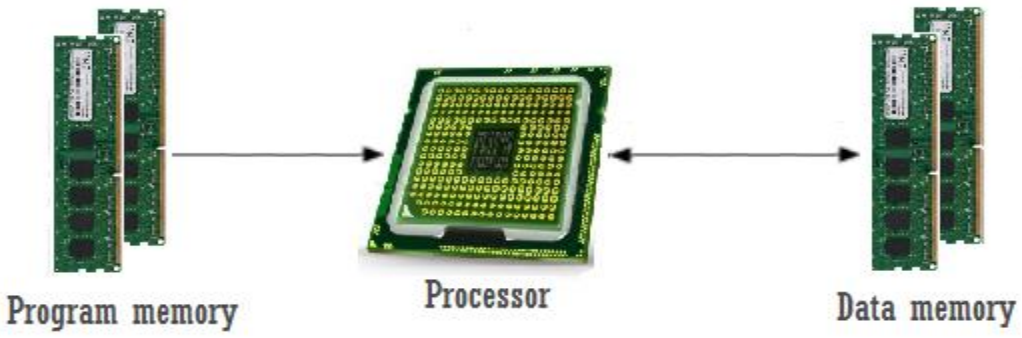


Figure 1.3 – Harvard architecture

The advantages of von Neumann architecture:

– simplification of the microprocessor device, since it only accesses common RAM;

– the use of a single memory area allows you to quickly redistribute resources between program and data areas, which significantly increases the flexibility of the microprocessor.

The advantages of Harvard architecture:

– the small areas use of data memory contributes to the acceleration of information retrieval in memory and increases the speed of the microprocessor;

– the presence of a separate data bus and command bus also allows you to increase the speed of the microprocessor;

– it is possible to organize parallel execution of programs (each memory is connected to the processor by a separate bus, which allows doing the current command simultaneously with reading/writing data while the current command performs the selection and decoding of the next command).

The disadvantage of the Harvard architecture is the complexity of the microprocessor architecture and the need to generate additional control signals for the instruction memory and data memory.

The modern approach suggests that von Neumann principles still underlie the construction of single-processor computers, although they have been significantly modified. Multiprocessor computer systems capable of parallel computing are based on the Harvard architecture.

The set of functional blocks (devices) and the connections between them are called the functional structure of the computer. More details about the functional structure of modern computers can be found in [1, 2, 4]. The functional structure of the computer determines its specific composition at a certain level of detail (devices, blocks, nodes, etc.) and describes all internal connections. However, the structure of the computer should be distinguished from its architecture. Under this term is understood a set of logical and physical organization principles of the computer (Fig. 1.4).

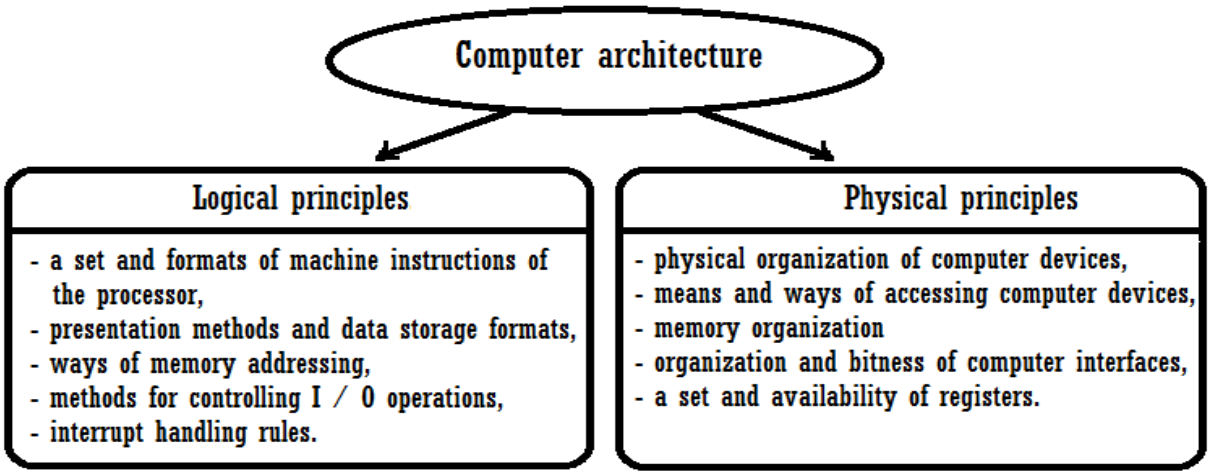


Figure 1.4 – The main components of computer architecture

The fundamental principles of computer logic are also formulated by von Neumann. Including:

– the use of a binary number system for encoding information in a computer,

– software management of the computer,

– memory uniformity,

– memory addressing.

The processor workflow should be described. There are six basic types of access that a processor can perform with external chips. The processor can write data to memory or write data to an I/O device, read data from memory or read data from an I/O device, read instructions from memory, and perform internal manipulation of data within the processor. In many systems, writing data to memory is functionally

identical to writing data to an I/O device. Similarly, reading data from memory constitutes the same external operation as reading data from an I/O device, or reading an instruction from memory. In other words, the processor makes no distinction between memory and I/O.

The internal data storage of the processor is known as its registers. The processor has a limited number of registers, and these are used to hold the current data/operands that the processor is manipulating.

The Arithmetic Logic Unit (ALU) performs the internal arithmetic manipulation of data in the processor. The instructions that are read and executed by the processor control the data flow between the registers and the ALU. The instructions also control the arithmetic operations performed by the ALU via the ALU's control inputs.

Whenever instructed by the processor, the ALU performs an operation (typically one of addition, subtraction, NOT, AND, OR, XOR, shift left/right, or rotate left/right) on one or more values. These values, called operands, are typically obtained from two registers, or from one register and a memory location. Thereafter, the result of the operation is placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero, negative, or if an overflow or carry occurred. Some processors have separate units for multiplication and division, and for bit shifting, providing faster operation and increased throughput.

Each architecture has its own unique ALU features, and this can vary greatly from one processor to another. However, there are just thematic variations, and each has the common characteristics described above.

2 NUMERAL SYSTEMS

2.1 Number systems used in computer calculations

A number system (NS) is a set of symbols and rules intended to represent numbers. The number systems may be classified into positional and nonpositional.

In ancient times, people used to count on their fingers. When the fingers become insufficient for counting, stones, pebbles or sticks were used to indicate the values. This method of counting is called the nonpositional number system. It was very difficult to perform arithmetic operations with such a number system, as it had no symbol for zero. The most common nonpositional number system is the "Roman number system". In this system, only a few characters are used to represent the numbers, for example, I, V, X, L (for fifty), C (for hundred) and so on. Moreover, since it is very difficult to perform the addition or any other arithmetic operations in this system, no logical or positional techniques are used in this system. An interesting analysis of nonpositional systems can be found in [10]. In computers, positioning systems are used.

A **positional system** is a system for numbers representation by an ordered set of numerals symbols (called digits) in which the value of a numeral symbol depends ***on its position***. For each position a unique symbol or a limited set of symbols is used.

In a positional system with a base ***N***, the number ***A*** is a sequence of digits:

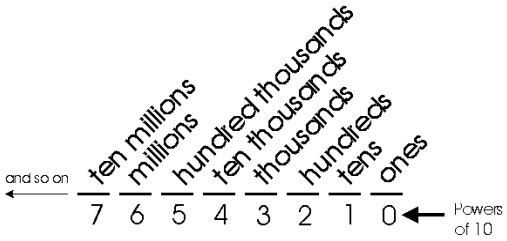
$$A_N = a_{m-1}a_{m-2}\ldots a_0\,.\,a_{-1}a_{-2}\ldots a_{-k} = a_{m-1}\cdot N^{m-1} + a_{m-2}\cdot N^{m-2} + \ldots + a_k\cdot N^{-k} = \sum_{i=-k}^{m-1} a_i \cdot N^i\,,$$

where a_i is the ***i***-th digit of the number ***A***; ***k*** is the number of digits in the fractional part of the number ***A***; ***m*** is the number of digits in the integer part of the number ***A***; ***N*** is the base of the number system.

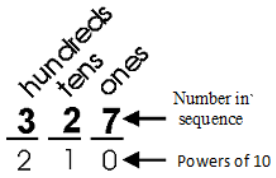
For any number system, if the number system's ***base*** (also called the ***radix***) is known, then can be discovered how many digits are used in creating written numbers in that system.

For any number system that has a base ***b***, the first ***b*** non-negative numbers are represented by the digits themselves. For base ***b***=10 (**decimal**), the first 10 numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Numbers beyond the first ***b*** numbers are represented by writing multiple digits, and ***associating*** each digit with a ***place value***. As you know, the number that follows 9 is 10, and writing 10 uses two digits: '1' and '0'. The '1' occupies the "tens" place and the '0' occupies the "ones" place. That's 1 "ten" and 0 "one". Counting continues with 11, 12, 13, ... , 18, 19, and finally 20, which is 2 "tens" and 0 "ones". After reaching the number 99 in the score, the next number in the sequence will require the addition of a third place, a place “one hundred” to the left, for the number 100: 1 “hundred”, 0 “ten” and 0 “one”.

Each place in a decimal number is associated with a power of ten, as we have seen. The right-most position is associated with "ones" or 10^0. The position to the left is associated with "tens" or 10^1. The position to the left of that is associated with "hundred" or 10^2. A value of a digit position is 10^n , where n – is a spot of each digit, as shown in this diagram:



Any number can be written as a sum of products of powers of 10. Consider the number '327.' Suppose we wrote it down in the diagram shown above, like this:

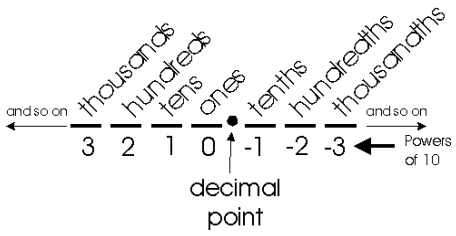


We can write the number '327' as a sum of products of powers of ten as follows:

$$327 = 3 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0 .$$

This is the number '327' written in an ***expanded notation***. It means exactly the same thing as writing '327' - it's just longer. This kind of notation is much more specific than just writing '327' because the base is given. When you see the number '327' written, you assume the base is 10. With computers, you can't always assume the base is 10. In fact, many times the base will be 2, 8, or even 16! The expanded notation is the only way to be sure which number is exactly considered.

Let's expand the diagram of place values and include a decimal point and places to the right of it. Here is how it would look:



Obviously, the place values become negative powers of ten, counting backwards from zero. This is logical, and easy to remember. The rule about writing any number in expanded notation still works, also, without any modification. For example, the number '43.57' would be written as:

$$43.57 = 4 \cdot 10^{1} + 3 \cdot 10^{0} + 5 \cdot 10^{-1} + 7 \cdot 10^{-2} .$$

In the **binary** number system, the base consists of 2 numbers: 0, 1. The weighted values for each position are determined as follows:

	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	
…	128	64	32	16	8	4	2	1	0.5	0.25	…

The binary system underlies modern technology of electronic digital computers. Computer memory comprises small elements that may only be in two states - ***off/on*** - that are associated with digits 0 and 1. Such an element is said to represent one ***bit*** - binary digit.

Counting in binary is just like counting in decimal. Each byte is comprised of four bits which, in binary notation, will describe adequately a number between 1 and 10 (the number 10 corresponding to the dialed zero). Real numbers can also be represented using binary notation by interpreting digits past the decimal point as negative powers of two.

In addition to the binary representation, digital computers use the **octal** and **hexadecimal** number systems. Considering an octal or hexadecimal number is much shorter than the respective binary number. Also binary code can be directly converted into **octal** or **hexadecimal** form, because the radices in the octal and hexadecimal notation are integer powers of two.

The following table shows the numbers from 1 to 16 which are recorded in different numeral systems.

Table 2.1 – Correspondence of numbers in different number systems.

Numeral system				
Decimal	Binary	Octal	Hexadecimal	Binary-decimal
0	0	0	0	0000 0000
1	1	1	1	0000 0001
2	10	2	2	0000 0010
3	11	3	3	0000 0011
4	100	4	4	0000 0100
5	101	5	5	0000 0101
6	110	6	6	0000 0110
7	111	7	7	0000 0111
8	1000	10	8	0000 1000
9	1001	11	9	0000 1001
10	1010	12	A	0001 0000
11	1011	13	B	0001 0001
12	1100	14	C	0001 0010
13	1101	15	D	0001 0011
14	1110	16	E	0001 0100
15	1111	17	F	0001 0101
16	10000	20	10	0001 0110
...	...	...	...	...
32	100000	40	20	0011 0010

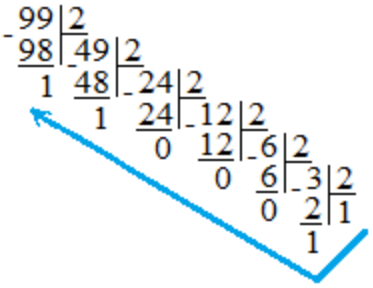
The sequence of actions for converting numbers from one number system to another is the same in all cases. In order to convert an integer N expressed in the radix p to a number system in the radix q, the number must consecutively be divided by the radix q until the last quotient is less than q. Then in the system to the radix q, the number N will be presented as an ordered set of residuals, the most significant digit in N being the last quotient.

The conversion form from number system to another is fully described in [11].

2.2 Conversion of a number from one number system to another. Examples

Example 2.1 Convert 99 decimal into the binary system. The decimal number is consecutively divided by 2:

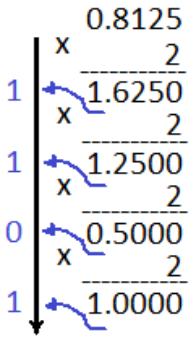
divide 99 by 2 – get the quotient 49 and the remainder 1;
divide 49 by 2 – get the quotient 24 and the remainder 1;
divide 24 by 2 – get the quotient 12 and the remainder 0;
divide 12 by 2 – get the quotient 6 and the remainder 0;
divide 6 by 2 – get the quotient 3 and the remainder 0;
divide 3 by 2 – get the quotient 1 and the remainder 1.



The arrow shows the direction in which the number should be read. Thus, the binary equivalent of 99 decimal is $99_{10} = 1100011_2$.

In converting a fraction expressed in the radix ***p*** to a number system in the radix ***q***, the fraction is consecutively multiplied by the radix q, only the fractional part being multiplied at each step. In the ***q***-nary system, the fraction will be represented by an ordered sequence of the integer parts of the products, where the most significant digit is the first digit of the product.

Example 2.2 Convert 0.8125 decimal to binary form. The fraction is consecutively multiplied by 2:



The arrow indicates the direction in which the number should be read. Thus, the binary equivalent of 0.8125 decimal is 0.1101. In the case of a mixed decimal number, it is first separated into the integer and fractional parts, and each part is then converted as already explained.

Example 2.3 Let's bring the number 150_{10} to the octal basis. We first find the largest power of 8 that is smaller than our number. Here, this is 8^2 or 64 (8^3 is 512). We count how many groups of 64 we can take from 150. This is 2, so the first digit in our base-8 number is 2. We have now accounted for 128 out of 150, so we have 22 left over. The largest power of 8 that is smaller than 22 is 8^1 (that is, 8). How many groups of 8 can be taken from 22? Two groups again, and thus our

second digit is 2. Finally, we are left with 6, and can obviously take 6 groups of one from this, our final digit. We end up with 226_8.

In fact, it can be made clearer with math steps:

1. $150/8^2 = 2$ remainder 22,

2. $22/8^1 = 2$ remainder 6,

3. $6/8^0 = 6$.

Example 2.4 Base-16 is also commonly used in computer programming, so it is very important to understand. The figure shows the correspondence of the symbols of the hexadecimal system and their values in the decimal system.

Except these extra digits, hexadecimal is just like any other base. For example, let's convert $3D_{16}$ to base-10. Following our previous rules, we have:

$$3D_{16} = 3 \cdot 16^1 + 13 \cdot 16^0 = 48 + 13 = 61_{10} \ .$$

So $3D_{16}$ is equal to 61_{10}. Notice how we use D's value of 13 in our calculation.

Example 2.5 We can convert from base-10 to base-16 similar to the way we did with base-8. Let's convert 696_{10} to base-16. First, we find the largest power of 16 that is less than 696_{10}. This is 16^2, or 296. Then:

1) $696/16^2 = 2$ remainder 184,

2) $184/16^1 = 11$ remainder 8,

3) $8/16^1 = 8$ remainder 0.

We have to replace 11 with its digit representation B, and we get $2B8_{16}$.

Feel free to try some more conversions for practice.

2.3 Exercises

2.3.1. Numbers in Table 1.2 in a given numeral system are represented in different numeral systems. Select a task from Table 2.2 according to your number in the group list. Present the results as shown in Table 2.3.

Table 2.2 – Individual tasks

Option	Numeral system				Option	Numeral system			
	Decimal	Binary	Octal	Hexadecimal		Decimal	Binary	Octal	Hexadecimal
1	234	1110111	234	A34	16	269	11001110	262	2C9
2	432	10010111	432	B32	17	805	11100011	705	80D
3	567	11001011	567	C67	18	675	11000110	675	2F5
4	543	10110011	543	543	19	931	10011001	731	9E1
5	786	10101011	756	7FF	20	873	11011011	773	87A
6	469	1111001	463	46A	21	764	11001001	764	764
7	897	10001110	736	A197	22	231	10010111	207	F31
8	438	11011010	437	4A38	23	345	11011101	345	34D
9	291	10010010	261	2B11	24	456	10101111	456	4F6
10	658	11101110	657	6E8	25	678	11101110	671	678
11	386	10000111	376	2B86	26	765	10111011	765	1A65
12	987	10111001	765	3E87	27	891	11111101	561	A91
13	876	11100010	654	A76	28	588	10000111	555	B88
14	564	01101100	564	B64	29	677	10011111	677	A177
15	368	10010011	366	3A8	30	483	10011000	443	1B83

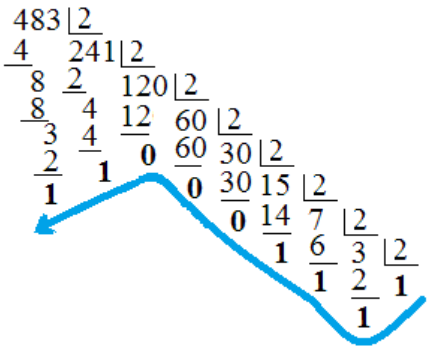
Table 2.3 - Example of a table to fill in the results of option 30 of Table 2.2

№	Decimal	Binary	Octal	Hexadecimal	Binary-decimal
1	**483**	?	?	?	?
2	?	**10011000**	?	?	?
3	?	?	**443**	?	?
4	?	?	?	**1B83**	?

2.3.2 Example of progress for option number 30 from Table 2.2.

a) Translation number 483 of their decimal system into binary NS.

Solution:



Answer: $483_{10} \rightarrow 111100011_2$

This result should be entered into Table 2.3 (at the intersection of line 1 and column 2).

Checking the result.

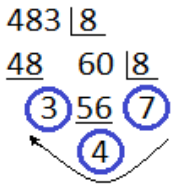
Ponderable coefficient	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Number	1	1	1	1	0	0	0	1	1

$$1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^1 + 1\cdot 2^0 = 256 + 128 + 64 + 32 + 1 + 1 = 483_{10}.$$

b) Translate number 483 in decimal NS into octal NS: 483_{10} → $?_8$.

Solution: let's show two ways:

1) We perform a division operation:



2) Divide binary number into triads from right to left and give each triad the corresponding octal number (see Table 2.1):

483_{10} → 111100011_2,

$111.100.011_2$ → **743_8**

Checking the result: $7*8^2+4*8^1+3*8^0=7*64+4*8+3=483_{10}$

Answer: **743_8**, this result should be entered into Table 2.3 (at intersection of line 1 and column 3).

c) Translate number 483_{10} into hexadecimal NS: 483_{10} → $?_{16}$.

Solution: Divide binary number into tetrads (4) from right to left and transform each tetrad into the corresponding hexadecimal number:

483_{10} → 111100011_2, 0001.1110.0011 → **$1E3_{16}$**

Checking the result: $1E3_{16}$ –> $1*16^2 + 14*16^1 + 3*16^0 = 256+ 224 +3 = 483_{10}$

Answer: **$1E3_{16}$,** this result should be entered into Table 2.3 (at the intersection of line 1 and column 4).

d) Translate number 483_{10} into binary-decimal NS: 483_{10} → $?_{2\text{-}10}$.

Solution: Translate each decimal figure into the corresponding binary tetrad:

4 8 3 → 0100 1000 0011.

Answer: 100100000112-10 – this result should be entered into Table 1.3 (at the intersection of line 1 and column 5).

2.4 Control questions

1. Explain non-positional and positional numeral systems, give examples of non-positional and positional numbers.

2. Define the notion of "numerical system", "the basis of the numerical system".

3. What is base 10? Binary? Hexadecimal?

4. How can you convert from one base to another?

5. Determine the maximum positive number that can still be placed in one, two, four, or eight bytes.

6. What operation must be performed to convert an integer part from one s / h to another:

a) dividing;

b) subtraction;

c) multiplication;

d) adding?

3 REPRESENTATIONS IN OPERATIONAL MEMORY. LOGICAL OPERATIONS

3.1 Representation of information in a computer

Information (command and data: numeric, textual, graphical, etc.) is encoded with binary digits 0 and 1. Therefore, various types of information **placed** in computer memory are practically undetectable, identification is possible only with the execution of the program, according to its logic and to the context. For example, the phrase “Hello world!” In binary code encoded in Windows-1251 looks like this:

1100111111 1100001110 1000111000 1011100101 1111001000
1011000010 0000111011 0011101000 1111000000 100001

Each type of information has formats – units-structural information encoded with binary digits 0 and 1. The basic unit is 1 **bit**. A bit is the smallest piece of information obtained when choosing between two equiprobable events.

The minimum unit of information in the computer is 1 **byte**, which is equal to 8 adjacent bits. The string of bits making up a byte is processed as a unit by a computer. A byte can represent the equivalent of a single character, such as the letter 'B', a comma, or a percentage sign, or it can represent a number from 0 to 255.

Usually, all data formats used in the computer are multiples of a byte, i.e. consist of an integer number of bytes.

A computer **word**, like a byte, is a group of fixed number of bits processed as a unit, which varies from computer to computer but is fixed for each computer. The length of a computer word is 2 bytes for 16-bit operating systems, 4 bytes for 32-bit operating systems, and 8 bytes for most 64-bit operating systems (in some cases 4 is also possible considering the difference in data models).

Special types of memory units are used to measure large amounts of information. The table shows the different sizes of memory:

Table 3.1 – Special types of memory units

Name	Short name	Number of bytes	Number of bytes approx
Bit	-	1 bit = 1/8b	
Nibble	-	4 bit = 1/2b	
Byte	b	2^0 = 1b	1b
Kilo Byte	K or Kb	2^{10} = 1024b	$1024 \approx 10^3$ b
Mega Byte	Mb	2^{20} = 1024Kb	$1\ 048\ 576 \approx 10^6$ b
Giga Byte	Gb	2^{30} = 1024Mb	$\approx 10^9$ b
Tera Byte	Tb	2^{40} = 1024Gb	$\approx 10^{12}$ b
Peta Byte	Pb	2^{50} = 1024Tb	$\approx 10^{15}$ b
Exa Byte	Eb	2^{60} = 1024Pb	$\approx 10^{18}$ b
Zetta Byte	Zb	2^{70} = 1024Eb	$\approx 10^{21}$ b
Yotta Byte	Yb	2^{80} = 1024Zb	$\approx 10^{24}$ b

3.2. Typical computer registers

The processor handles all information in the computer. Information from the memory enters processor registers.

The registers are intended for storing multi-digit codes of data commands, addressing and processing arithmetic and logical operations on them. The microprocessor contains the following groups of registers: general registers, segment registers, a command pointer and flags. For example, microprocessors Intel 8086 contain twelve programmable addressable registers with a width of 16 bits, which are combined into three groups: data registers; register-pointers and segment registers. The data registers and index registers are grouped under the general name “General Purpose Registers” (fig. 2.1). In addition to these registers, there are two other registers (status and control registers): an instruction pointer and a flag register.

The register consists of two parts – the high-order half and the lower half – each its has of 8 bits. For example, AH – high AX – high-order half of 8 bits, AL – low AX – lower half of 8 bits.

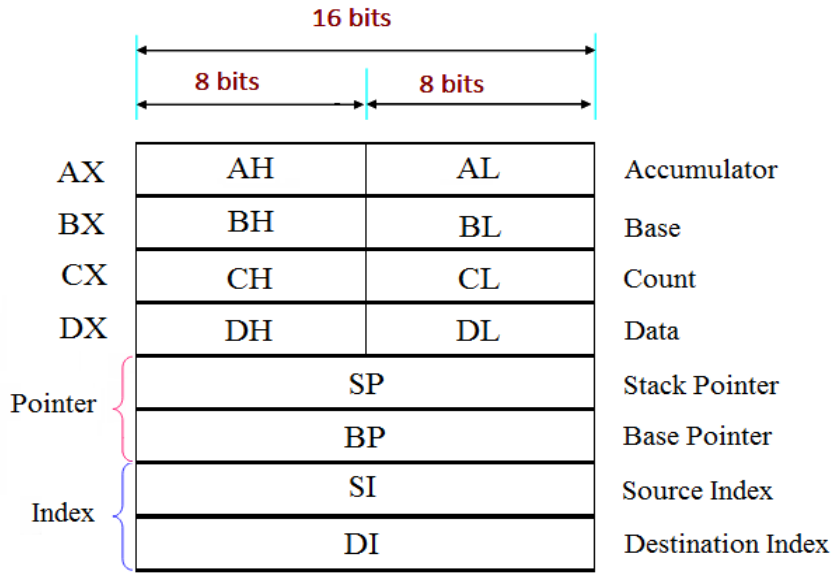


Figure 3.1 – General Purpose Registers [13]

Four data registers (AX, BX, CX and DX) can be used by a programmer for temporary storing any objects: operands of logical and arithmetic operations; address components; pointers on the main memory cells.

Some of these registers have a strict functional purpose, which are indicated in the right column in the fig 2.1. This provides compact coding and memory savings.

Each bit of register information is stored in a memory device with a storage that is called **a trigger**. When a signal is applied to the trigger input, it is set to 0 or 1. This state is stored until a new signal is inputted to the port or the voltage is disconnected. The union of eight, sixteen or more triggers is a functional node, called **a register**. Except triggers, the register includes additional schemes that ensure:

– setting the register to zero;

– reception of the transfer of codes;

– shift the code left and right to the required number of digits;

– boolean logical operations, etc.

A counter is a combination node that calculates the number of input signals received at its port. There are adding or subtracting counters.

The operating node performing the arithmetic addition of the number codes is called **the adder**. The addition is performed bitwise, taking into consideration the carry bit from the previous digit.

A combinational logic circuit that converts the input code into a signal on only one of the outputs is called **a decoder.** Usually decoders are used to convert binary code to decimal.

More information can be found in the book "Structured Computer Organization" [12] and many other books.

3.3 Representation of numerical information in a computer

The computer uses three types of numbers: fixed point, floating point and binary-decimal representation for integers. Integers are the simplest numeric data that a computer operates with. There are two representations for integers: **unsigned** (only for non-negative integers) and **signed**.

For **unsigned numbers**, the range of number values is determined by the inequation:

$$0 \le A_2 \le 2^n - 1,$$

where *n* is the number of bits assigned to the representation of the number. For example, for n = 8, unsigned integers can be represented from the range [0 - 255].

For **numbers with a sign**, the range of values change is determined by the inequation:

$$-\left(2^{n-1}-1\right) \le A_2 \le +\left(2^{n-1}-1\right).$$

When ***n*** = 8, you can represent signed integers from the range: [-127; +127], since one bit is assigned to the sign.

Positive numbers have “0” value of sign digit, negative ones have “1” value.

The length of the standard integer type most commonly coincides with the size of the computer word on the targeted platform. Many computer languages offer a choice between short, long and a standard length (int).

The representation of the whole type on different data models may change. ***Short int*** is 1-2 bytes, ***int*** is 2-4 bytes and ***longint*** is 4-8 bytes. Also in some languages, the data type is a ***longlong***, which is 8 bytes long.

Consider the representation of an integer in one byte. The bits are numbered in a byte from right to left (as in the figure below).

In computing the least significant bit (LSB) is the bit position in a binary integer, which giving the units value. The LSB is sometimes referred to as the low-order bit or right-most bit, due to the convention in positional notation of writing less significant digits further to the right. It is similar to the least significant digit of a decimal integer, which is the digit in the position one (right-most). The most significant bit (MSB, also called the high-order bit) is the bit, positioned in a binary number, which is having the greatest value. The MSB is sometimes referred to as the high-order bit or left-most bit due to the convention in positional notation of writing more significant digits further to the left. In conventional Intel bit ordering, MSB is numbered 7 and LSB is numbered 0:

MSB							LSB
7	6	5	4	3	2	1	0

Unsigned numbers have only one representation: significant digits are stored in all bits, from the low-order bit to the high-order bit. For example, the number “99” is written as follows:

Bit’s number:	7	6	5	4	3	2	1	0
number	0	1	1	0	0	0	1	1

An n-bit signed binary number consists of two parts: one part denoting the sing of the number and another part denoting the magnitude of the number. The MSB is always a sign bit, which denotes the sign of the number and the convention is that 0 and 1 denote “+” and “-“, respectively. The remaining (n-1) bit denotes the magnitude of the number. For example, the number "+99" has the representation:

	sing							
	MSB							LSB
Bit’s number:	7	6	5	4	3	2	1	0
number	0	1	1	0	0	0	1	1

The three best-known methods of extending the binary numeral system to represent signed numbers are: sign-and-magnitude, ones' complement and two's complement.

In "sign and magnitude" approach or signed magnitude representation (SMR), a number's sign is represented with a sign bit: setting MSB to 0 for a positive number or positive zero, and setting it to 1 for a negative number or negative zero. The remaining bits in the number indicate the magnitude (or absolute value). For example, in an eight-bit byte, only seven bits represent the magnitude can be ranged from 0000000 (0_{10}) to 1111111 (127_{10}). Thus, number “-99” can be represented once the sign bit (the eighth bit) is added as:

Bit’s number:		7	6	5	4	3	2	1	0
Positive number	“99”	0	1	1	0	0	0	1	1
Negative number	“-99”	1	1	1	0	0	0	1	1

The complement of a number is the number, which is added to the original and will make it equal to a multiple of the base number system.

The complement of a number can be used as a representation of that number as a negative and positive number that represents a negative. It is a method, which can be used to make substruction easier for machines. Consequently, complements are used in digital computers for simplifying the substruction operation and for the logical operation.

For every base r system, there are two types of complements: *r*s complement and (*r* - 1)s complement. For decimal *r* = 10, we have 9s and 10s complement. For binary *r* = 2, we have 1’s and 2’s complement. For octal *r* = 8, we have 7s and 8s complement. For hexadecimal *r* = 16, we have 9’s and 15’s complements.

The **1's complement**. Positive numbers are the same in both sequence, but we need to define the negative numbers in the system. All the negative numbers have the binary MSB = 1, which is helpful in identifying the sign of the number. In addition, the sign bit allows to divide the counting sequence evenly between positive and negative numbers.

To form the negative of any number, first of all complement all the bits of that number. This result is known as the one's complement of the original number. This requires changing every logic 1 bit in a number to logic 0, and every logic 0 bit to logic 1. For instance, let us find the 1's complement of 0011 0110 in binary:

Bit's number	7	6	5	4	3	2	1	0
Unsigned number	0	0	1	1	0	1	1	0
1's complement	1	1	0	0	1	0	0	1

The **2's complement**. We do not just place 1 in the MSB of a binary number to make it negative. We must take the 2's complement of the number. Taking the 2's complement of the number will cause the MSB to become 1.

To obtain the 2's complement of a number, there is a two-step process:

1. Take the 1's complement of the number by changing every logic 1 bit in the number to logic 0 bit, and change every logic 0 bit to logic 1 bit.

2. Add 1 to the 1's complement of the binary number. Now, we have the 2's complement of the original number. Here, we can notice that the MSB has become 1.

Both 1's and 2's complements of '00110110_2' are shown in the following table:

Bit's number	7	6	5	4	3	2	1	0
Unsigned number	0	0	1	1	0	1	1	0
1's complement	1	1	0	0	1	0	0	1
2's complement	1	1	0	0	1	0	1	10

What is the difference between 1's complement and 2's complement? The main difference is that 1's complement has two representations of zero: '00000000', which is a positive zero (+0) and '11111111', which is a negative zero (-0); whereas in 2's

complement, there is only one representation for zero – '00000000' (+0) because if 1 is added to '11111111' (-1), it's '00000000' which is the same as positive zero. This is the reason why 2's complement is generally used. Another difference is that while adding numbers using 1's complement, first of all binary addition is done and then added an end-around carry value. Thus, 2's complement has only one value for zero and doesn't require carry values.

3.4 Bitwise operations. Logical operations

Logical commands are mainly used to manipulate binary values. Logical commands are called logical because they act according to the rules of formal logic, rather than arithmetic. They change bit values in registers or memory cells.

The logical operations in the computer: inverting (NOT), logical addition OR, logical multiplication AND, exclusive XOR. These operations are performed on bits.

Table 3.1 – Bitwise Logical Operations

Operation	Description	Bit1	Bit2	Result bit
Not	Inverting	0		1
		1		0
And	Multiplication	0	0	0
		0	1	0
		1	0	0
		1	1	1
Or	Addition	0	0	0
		0	1	1
		1	0	1
		1	1	1
Xor	Exclusive	0	0	0
		0	1	1
		1	0	1
		1	1	0

Logical operations AND, OR, XOR are performed on two operands, and the NOT operation is performed on one operand.

The **AND** operation is useful for **filtering, masking, or setting to nil**. It sets the result bit to 1 for each position where both operands contain "1", otherwise the result bit is reset to 0.

The **OR** operation sets the result bit to 1 for each position where at least one of the two operands contains 1. In positions, where both operands contain 0, the result bit is 0. This command is usually used **to set bit to 1.**

The **XOR** (OR exclusive) operation is a modification of the OR operation. This operation sets a result bit to 0 when both operands are equal. This name of this operation is given because it excludes combination of 1&1 bits. This command is usually **used to invert a bit**.

The **NOT** command is used if it is necessary to invert all bits of a number.

The commands for manipulating bits (bitwise operations) are divided into two groups: logical commands and shift commands. The shift commands are in turn applied to linear and rotate shift instructions.

3.5 Examples of problem solving

Example 3.1 Invert number $A_{10} = 27$

Bit's number	7	6	5	4	3	2	1	0
$A_{10} = 27$	0	0	0	1	1	0	1	1
Operation	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*
Result	1	1	1	0	0	1	0	0

Example 3.2 Set the value of the 2nd and 3rd bits of the number "27" to 1.

To do this let's execute the bitwise logical addition (the logical operation OR) of the given number and the number, in which "1" only in the second and third bits:

Bit's number	7	6	5	4	3	2	1	0
$A_{10} = 27$	0	0	0	1	1	0	1	1
Operation	*OR*	*OR*	*OR*	*OR*	*OR*	*OR*	*OR*	*OR*
Mask	0	0	0	0	**1**	**1**	0	0
Result	0	0	0	1	1	1	1	1

Example 3.3 Reset the 3rd bit of the number "27" to 0.

To do this it's needed to perform the logical multiplication operation on a number containing zero in third bit.

Bit's number	7	6	5	4	3	2	1	0
A_{10} = 27	0	0	0	1	1	0	1	1
Operation	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*
Mask	1	1	1	1	**0**	1	1	1
Result	0	0	0	1	0	0	1	1

Example 3.4 Define the value of the 0-th bit of the number "38".

To do this it's needed to perform the bitwise logical multiplication (logical AND) of a given number and a number whose only 1 in the first bit.

Bit's number	7	6	5	4	3	2	1	0
A_{10} = 38	0	0	1	0	0	1	1	0
Operation	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*	*AND*
Mask	0	0	0	0	0	0	0	**1**
Result	0	0	0	0	0	0	0	0

Example 3.5 Invert the 5-th bit of the number "38".

To invert the 5-th bit the mask is overlay: in the 5-th bit is 1, and in the other bits – 0; then it's needed to perform the XOR operation.

Bit's number	7	6	5	4	3	2	1	0
D = 38	0	0	1	0	0	1	1	0
Operation	*XOR*	*XOR*	*XOR*	*XOR*	*XOR*	*XOR*	*XOR*	*XOR*
Mask	0	0	**1**	0	0	0	0	0
Result	0	0	0	0	0	1	1	0

3.6 Exercises

3.6.1 Select a task from table 1 according to your number in the group list.

Table 3.2 – Individual task

Option number.	Decimal number	Bit number (n) to be seted.	Bit number (k) to be determined.	Bit number (m) to be reseted	Bit number (p) to be inversed
1	2	3	4	5	6
1	65	1	4	0	5
2	30	2	5	1	6
3	16	3	6	2	7
4	25	4	7	3	0
5	55	5	0	4	1
6	63	6	1	5	2

Continue Tab.3.2

1	2	3	4	5	6
7	94	7	2	6	3
8	83	0	3	7	4
9	43	1	4	0	5
10	64	2	5	1	6
11	97	3	6	2	7
12	84	4	7	3	0
13	11	5	0	4	1
14	26	6	1	5	2
15	68	7	2	6	3
16	12	0	3	7	4
17	33	1	4	0	5
18	44	2	5	1	6
19	28	3	6	2	7
20	46	4	7	3	0
21	35	5	0	4	1
22	56	6	1	5	2
23	34	7	2	6	3
24	48	0	3	7	4
25	67	1	4	0	5
26	81	2	5	1	6
27	96	3	6	2	7
28	98	4	7	3	0
29	37	7	0	5	1
30	23	5	1	4	6

1) For a decimal number (positive and negative), it’s needed to get its binary representation in three formats: the sign-magnitude representation (SMR) or direct code, 1’s complement notation and 2’s complement notation. Use a field with a length of 1 byte to represent the code.

2) Perform bitwise logical operations to the sign–magnitude representation of a positive number in such order:

– invert the received binary number;

– set the selected bit;

– determine the value of the selected bit;

– reset the selected bit;

– invert the selected bit.

3) Perform the same logical operations for a negative number in the 2's complement code.

3.6.2 Process of obtaining results for variant 30 from table 3.2

a) For number '23_{10}' we have:

	+23								-23							
Bit's number	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
SMR	0	0	0	1	0	1	1	1	1	0	0	1	0	1	1	1
1′s complement	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0
2′s complement	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	1

For a positive number, SMR, 1's complement and 2's complement codes are the same.

b) We perform the operation NOT to invert a binary number (flip all the bits):

Bit's number	7	6	5	4	3	2	1	0
A_{10} = +23	0	0	0	1	0	1	1	1
Operation	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*	*NOT*
Result	1	1	1	0	1	0	0	0

c) Set the value of the 5-th bit of the number 23 to 1.

Perform logic operation OR for this number and number, that have 1 only at 5-th bit – $2^5 = 32_{10}$.

		7	6	5	4	3	2	1	0
A_{10} = +23		0	0	0	1	0	1	1	1
Operation	***OR***								
Mask = 32		0	0	**1**	0	0	1	0	0
Result = 55		0	0	**1**	1	0	1	1	1

d) Determine the value of the first bit of the number 23.

Perform logic operation AND for this number and the number, that have 1 only in bit 1 – $2^1 = 2_{10}$

		7	6	5	4	3	2	1	0
A_{10} = +23		0	0	0	1	0	1	1	1
Operation	***AND***								
Mask = 2		0	0	0	0	0	0	**1**	0
Result = 2		0	0	0	0	0	0	**1**	0

Let's go through another example for a better understanding. Define the 5-th bit in the binary number. To do this it's needed to perform logic operation AND with a mask – $2^5 = 32_{10}$

		7	6	5	4	3	2	1	0
A_{10} = +23		0	0	0	1	0	1	1	1
Operation	***AND***								
Mask = 32		0	0	**1**	0	0	0	0	0
Result = 0		0	0	**0**	0	0	0	0	0

e) Reset the 4-th bit of number 23.

Perform logic operation AND for this number and the number that have 0 only in 4-th bit.

		7	6	5	4	3	2	1	0
A_{10} = +23		0	0	0	1	0	1	1	1
Operation	***AND***								
Mask = 237		1	1	1	**0**	1	1	1	1
Result = 7		0	0	0	**0**	0	1	1	1

f) Inverse 6-th bit of number 23.

Perform logic operation XOR for this number and number that have 1 only in 6-th bit – $2^6 = 64_{10}$.

		7	6	5	4	3	2	1	0
A_{10} = +23		0	0	0	1	0	1	1	1
Operation	***XOR***								
Mask = 64		0	**1**	0	0	0	0	0	0
Result = 87		0	1	0	1	0	1	1	1

3.7 Control questions

1. Are there sign–magnitude, 1’s complement, and 2’s complement codes for a positive and a negative number?

2. Rules of performing bitwise logical operations that are used in this laboratory work.

3. Why masks are needed when performing bitwise logical operations?

4 Use 2’s complement code for number ‘-5’. Reset the 7-th bit. What number is it?

5. Why registers of microprocessor are divided into groups? List functional destinations of: ALU registers; segment registers.

6. Arrange the following units of memory in ascending order of their capacity: terabyte, petabyte, megabyte and gigabyte.

4 ADDITION OF INTEGER BINARY NUMBERS

4.1 Adding binary numbers

All modern computers are equipped with a well-developed command system, the implementation of which may include tens and hundreds of machine operations. However, the implementation of any operation is based on the use of simple micro-operations, such as addition and displacement. This allows you to have a single arithmetic-logical device for performing any operations related to information processing.

Previously we talked about different representations of numbers in binary code: the sign-magnitude representation or direct code, ones' complement notation and two's complement notation.

The sign-magnitude representation is used to represent numbers in the computer's memory, as well as when performing operations of multiplication and division. The ones' complement and two's complement codes are used to perform the subtraction operation, which is replaced by the addition of numbers with different signs: $a - b = a + (-b)$.

The direct code of a binary number coincides in the image with the record of the number itself. The sign bit value for positive numbers is 0, and for negative numbers – 1. The one's complement code for a positive number is the same as the direct code. For a negative number, all digits of the number are replaced by the opposite ones (1 by 0, 0 by 1), and unit is entered in the sign bit. In the future, when writing the code, we agree to separate the signed bit from the digital ones with a comma.

For example, for the number +1101 the direct code is 0,0001101, for the number -1101 the direct code is 1,0001101.

The 1's complement code for a positive number is the same as SMR code. For a negative number, all digits of the number are replaced by the opposite ones (1 by 0, 0 by 1), and unit is entered in the sign bit.

For example: for the number +1101 – SMR code is 0,0001101; the 1's complement code is 0,0001101. For the number -1101, SMR code is 1,0001101; the 1's complement code is 1,1110010.

The two's complement code of positive number matches the ones' complement code. For a negative number, the calculation of the 2's complement is formed in two steps: take the ones' complement of number and add one to the result.

For example: for the number +1101 :

SMR	1's complement	2's complement
0,0001101	0,0001101	0,0001101

.

For the number -1101 :

SMR	1's complement	2's complement
1,0001101	1,1110010	1,1110011

.

Computers perform addition in the binary number system. Table 4.1 shows the rules for adding binary digits $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ of the same order, taking into consideration the possible displacement of $\boldsymbol{P}_{i-1}$ from the previous lower order and the possible displacement of $\boldsymbol{P}_i$ to the next higher order.

Table 4.1 - Rules for adding binary digits

The value of i-th bits of numbers A, B and shifting from the previous order P_{i-1}			Order of the sum S_i	Shifting to the next order P_i
a_i	b_i	P_{i-1}		
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Consider addition of the two numbers:

$$\begin{array}{l} 100 \quad \leftarrow -carries \\ 0100_2 = 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0 = 4_{10} \\ + \\ 0110_2 = 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0 = + 8_{10} \\ \hline 1010_2 = 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 = 12_{10} \end{array}$$

The subtraction operation is reduced to the addition operation by converting numbers to the reverse or additional code according to Table 4.2.

Table 4.2 – Rules of conversion the subtraction operation into add operation

Required Operation	Required Conversion
A+B	A+B
A-B	A+(-B)
-A+B	(-A)+B
-A-B	(-A)+(-B)

Consider another example of addition:

$$\begin{array}{l} 100 \quad \leftarrow -carries \\ 0100_2 = 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0 = 4_{10} \\ + \\ 1110_2 = 1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0 = + 14_{10} \\ \hline 0010_2 = 0\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 = 2_{10} \end{array}$$

The result, 2, is arithmetically incorrect. The problem here is that the addition has produced carry beyond the fourth bit. Since this is not taken into account in the result, the answer is wrong. The concept of overflow arises.

Overflow of the bit grid can lead to the transfer of a unit to a sign bit, which will lead to an incorrect result. A positive number resulting from an arithmetic operation can be perceived as negative, since "1" appears in the sign digit and vice versa. For example:

$$X = 0{,}11011110 \qquad Y = 0{,}1101100 \qquad X + Y = 1{,}1001010\ ,$$

X and Y are codes of positive numbers, but in the process of addition, “1” appeared in the sign digit, which means the code of a negative number.

To recognize the overflow of the bit grid in a computer, modified sign-magnitude (direct), 1's complement (reverse), and 2's complement codes are used. In these codes, the sign is encoded in two digits: the combination of '00' corresponds to the plus sign and the combination of '11' to the minus sign. The indicator of overflow in the modified code is the difference between digits in sign digits. It is '01' or '10' in the sign bits.

To cope with overflow, it's needed to normalize the number. Addition of numbers in modified codes is no different from addition in ordinary forms of 1's or 2's complement codes. However, when adding numbers in the additional code, the resulting carry unit in the sign digit is discarded, and when adding numbers in the reverse code, the resulting carry unit in the sign bit is added to the least significant bit of the code sum.

Consider example of addition: add two numbers: A_{10} = +16, B_{10} = -7 in 1's and 2's complement codes.

According to the table, the transformation A + (- B) is necessary, in which the second term is converted taking into consideration the sign. Use byte to represent numbers.

1's complement form of B: 11 111000;

2's complement form of B: 11 001000.

Addition in 1's complement code	Addition in 2's complement code	Check up
$A_{m1c} = 00\ 010000$	$A_{m2c} = 00\ 010000$	
$+$	$+$	
$B_{m1c} = 11\ 111000$	$B_{m2c} = 11\ 111001$	
$C_{m1c} = 1\ 00\ 001000$	$C_{m2c} = 1\ 00\ 001000$	$C_{10} = 1001_2 = 1 \cdot 2^3 + 1 \cdot 2^0 = 9$
$+$		
$C_{m1c} = 00\ 001001$	$C_{m2c} = 00\ 001001$	

Addition of binary numbers is carried out sequentially, bitwise in accordance with the rules (Table 4.1).

When adding numbers, the following rules must be observed:

1) The subtraction operation is replaced by the addition operation with a negative number.

2) The terms must have the same number of bits. To align the bit grid of terms, insignificant zeros is added to the left of the integer.

3) Sign bits participate in addition in the same way as significant ones.

4) The necessary code transformations are made with the change of signs of numbers. The assigned insignificant zeros change their value during transformations as a general rule.

5) When converting a transfer unit from a high-order sign bit, in the case of using 1's complement form code, this unit is added to the lowest bit. When using 2's complement form code, the transfer unit is lost. The sign of the result is generated automatically, the result is presented in the code in which the original terms are presented.

6) In order to check the result, it is converted into the SMR format.

4.2 Exercises

From Table 4.3, according to your number in the group list, select the values *A* and *B*, which are given: one – in the two's complement, and the second – in the ones' complement code, for the representation use a field of 1 byte (8 bits). When writing a code, we agree on separating the sign digit of the number with a comma from other digits.

Table 4.3 – Individual tasks

Variant	*C* (2's complement)	*D* (1's complement)	Variant	*C* (2's complement)	*D* (1's complement)
1	2	3	4	5	6
1	1,110111	0,110111	16	0,1101010	1,110110
2	1,1010111	0,10111	17	0,000111	1,1110001
3	1,1110110	0,101011	18	1,11111	0,110000
4	0,101011	1,1011001	19	0,1001011	1,100110
5	0,1111101	1,10111	20	0,01011	1,1011011
6	0,100111	1,0111101	21	1,1011001	0,1100100
7	1,1001111	0,100010	22	0,11011	1,1001011
8	0,1001100	1,1101101	23	1,0111100	0,1101

Continue Tab.4.3

1	2	3	4	5	6
9	1,0111	0,100010	24	0,101110	1,1010111
10	0,10111	1,101110	25	0,1101101	1,110
11	0,1101011	1,100011	26	1,1111	0,1011101
12	1,100011	0,111001	27	0,1001011	1,111101
13	1,01110	0,111010	28	0,111011	1,100111
14	0,110001	1,0110110	29	1,111	0,101111
15	1,10111	0,10011	30	0,1011	1,1100000

For your variant, get decimal, binary, sign-magnitude, ones' complement and two's complement codes for + *C*, -*C*, + *D* and -*D*, and fill in the Table, as shown in Table 4.4 for variant 30.

Table 4.4 – Example codes for variant 30 from Table 4.1

Codes	+*C*	–*C*	+*D*	–*D*
Decimal	+11	–11	+31	–31
Binary	+1011	–1011	+11111	–11111
SMR (Direct)	0,0001011	1,0001011	0,0011111	1,0011111
1's complement (Inverse)	0,0001011	1,1110100	0,0011111	1,1100000
2's complement (Complement)	0,0001011	1,1110101	0,0011111	1,1100001

Over the numbers *C* and *D* manually perform the addition operations in the modified one's compliment and two's complement codes, alternately using the "+" and "-" characters before each of these numbers, namely:

$$+D + (+C),\ +D + (-C),\ -D + (+C),\ -D + (-C).$$

Convert the result of the calculations in a sing-magnitude code, and then translate it into decimal code and perform a check. Obtained addition results are demonstrated as shown for variant 30 in Table 4.2.

Table 4.5 – Example results (for variant 30 from Table 4.2)

Numbers in different codes	Addition of numbers with different signs		Result	Result in SMR	Result in decimal code
C in decimal code (+11)	In two's complement code	+*C* + (–*D*)	1,1101100	1,0010100	–20
D in decimal code (–31)		+*C* + (+*D*)	0,0101010	0,0101010	+42
C in SMR (0,0001011)		–*C* + (–*D*)	1,1010110	1,0101010	–42
D in SMR (1,0011111)		–*C* + (+*D*)	0,0010100	0,0010100	+20
C in 1's complement code (0,0001011)	In ones' complement code	+*C* + (–*D*)	1,1101011	1,0010100	–20
D in 1's complement code (1,1100000)		+*C* + (+*D*)	0,0101010	0,0101010	+42
C in 2's complement code (0,0001011)		–*C* + (–*D*)	1,1010101	1,0101010	–42
D in 2's complement code (1,1100001)		–*C* + (+*D*)	0,0010100	0,0010100	+20

4.3 Process of obtaining results

The operation of adding two numbers using a modified 1's complement and a modified 2's complement code for numbers $D = 38_{10}$ and $C = 17_{10}$ is performed.

Calculate the values of the following expressions:

$$+D+(C),\ +D+(-C),\ -D+(C),\ -D+(-C).$$

The following terms are used in the text below: *SMR* – sign-magnitude (direct) representation; *1c* – ones' complement code; *m1c* – modified ones' complement code; *2c* – two's complement code; *m2c* – modified two's complement code.

Perform the above operations using a modified **ones' compliment code.**

1) $A = +D + (+C) = 38 + 17 = 55$

Both terms are positive, therefore their SMR and 1's complement code coincide.

$$\begin{array}{r} D_{m1c} = 00\,100110 \\ + \\ C_{m1c} = 00\,010001 \\ \hline A_{m1c} = 00\,110111 \end{array}$$

Two zeros in signed digits indicate that the number is positive, and the 1's complement code of representation of answer is coincided with the SMR. The received answer is converted to the decimal number system and the correctness

of the answer is checked: $A_{10} = 2^0 + 2^1 + 2^2 + 2^4 + 2^5 = 55$

2) $A = +D + (-C) = 38 - 17 = 21$

Note that the 1's complement code of the second term is different from its SMR.

$$\begin{array}{l} D_{m1c} = \ 00\,100110 \\ \quad + \\ C_{m1c} = \ 11\,010001 \\ \hline A_{m1c} = 100\,010100 \\ \quad\quad\quad\ \searrow + \nearrow \\ \hline\hline A_{m1c} = \ 00\,010101 \end{array}$$

We got the overflow of the bit grid and correction needed. The resulting '1' of carry in the sign digit is added to the least significant bit of the sum of codes to correct overflow.

In the received answer, two zeros are in signed digits and indicate that the number is positive, so the SMR code coincides with the 1's complement code.

Let's check up: $A_{10} = (2^0 + 2^2 + 2^4) = 21$

3) $A = -D + (+C) = -38 + 17 = -21$

Note that the 1's complement code of the first term is different from its SMR.

$$\begin{array}{l} D_{m1c} = \ 11\ 011001 \\ \quad + \\ C_{m1c} = \ 00\ 010001 \\ \hline A_{m1c} = \ 11\ 101010 \end{array}$$

Two **units** in sign digits indicate that the number is negative. To convert it into SMR, we invert the significant digits of the number *A*: $A_{mSMR} = \ 11\ 010101$.

Let's check up: $A_{10} = -(2^0 + 2^2 + 2^4) = -21$.

4) $A = -D + (-C) = -38 - 17 = -55$

$$\begin{array}{l} D_{m1c} = \ 11\,011001 \\ \quad + \\ C_{m1c} = \ 11\,101110 \\ \hline A_{m1c} = 111\,000111 \\ \quad\quad\quad\ \searrow + \nearrow \\ \hline\hline A_{m1c} = \ 11\,001000 \end{array}$$

We got the overflow of the bit grid and correction needed. The '1' was added to the least significant bit of the sum of codes to correct overflow. Two units in sign digits indicate that the number is negative. We invert the significant digits of the number *A*: $A_{mSMR} = \ 11\ 110111$

Let's check up: $A_{10} = -(2^0 + 2^1 + 2^2 + 2^4 + 2^5) = -55$.

Let’s perform an operation in **a modified two’s complement code**:

5) $A = +D + (+C) = 38 + 17 = 55$

Both terms are positive, therefore their SMR and 2’s complement code coincide.

$$\begin{array}{r} D_{m2c} = 00\,100110 \\ + \\ C_{m2c} = 00\,010001 \\ \hline A_{m2c} = 00\,110111 \end{array}$$

Two zeros in signed digits indicate that the number is positive, the 2’s complement code of the answer matches the SMR, it is $A_{mSMR} = \ 00\ \ 110111$.

Let's check up: $A_{10} = 2^0 + 2^1 + 2^2 + 2^4 + 2^5 = 55$.

6) $A = +D + (-C) = 38 - 17 = 21$

Note that the 2’s complement code of the second term is different from its SMR.

$$\begin{array}{r} D_{m2c} = 00\,100110 \\ + \\ C_{m2c} = \ 11\,101111 \\ \hline A_{m2c} = 00\,010101 \end{array}$$

Two zeros in sign digits indicate that the number is positive, SMR coincides with the 2’s complement code, it is $A_{mSMR} = \ 00\ \ 010101$.

Let's check up: $A_{10} = 2^0 + 2^2 + 2^4 = 21$.

7) $A = -D + (+C) = -38 + 17 = -21$

Note that the 2’s complement code of the first term is different from its SMR.

$$\begin{array}{r} D_{m2c} = 11\,011010 \\ + \\ C_{m2c} = 00\,010001 \\ \hline A_{m2c} = 11\,101011 \end{array}$$

Two units in sign digits indicate that the number is negative. To convert it into SMR, we invert the significant digits of the number and add 1 to the least significant bit:

$$\begin{array}{r} 11\,010100 \\ +1 \\ \hline\hline A_{mSMR} = 11\,010101 \end{array}$$

Let's check up: $A_{10} = -(2^0 + 2^2 + 2^4) = -21$.

8) $A = -D + (-C) = -38 - 17 = -55$

$$\begin{array}{r} D_{m2c} = 11\,011010 \\ + \\ C_{m2c} = 11\,101111 \\ \hline A_{m2c} = 11\,001001 \end{array}$$

Two units in sign digits indicate that the number is negative. To convert it into SMR, we invert the significant digits of the number and add 1 to the least significant bit:

$$\begin{array}{r} 11110110 \\ +1 \\ \hline\hline A_{mSMR} = 11110111 \end{array}$$

Let's check up: $A_{10} = -(2^0 + 2^1 + 2^2 + 2^4 + 2^5) = -55$.

4.4 Control questions

1. What is the difference between sign-magnitude representation, ones' complement code and two's complement code in computer for positive and negative numbers?

2. What happens to the unit of transfer from a higher character digit when performing the addition operation in ones' complement form and two's complement form codes?

3. What is the sign of a bit grid overflow when performing an add operation?

4. How is a digit grid overflow corrected?

5 ADDITION AND SUBTRACTION OF FLOATING-POINT NUMBERS

5.1 Floating-point representation

In floating-point representation, the computer must be able to represent the numbers and has an ability to operate on them in such a way, that the position of the binary point is variable and automatically adjusted as computation proceeds, for the accommodation of very large integers and very small fractions. In this case, the binary point is said to be the float, and the numbers are called the floating point numbers.

Numbers are too large for standard integer representations or have fractional components, which are usually represented in scientific notation, such a form commonly used by scientists and engineers. When we use Scientific Notation in a decimal, we write numbers in the following form:

$$+/- \, mantissa \times 10^{exponent}.$$

In this form, there is an optional sign which indicates whether the overall number is positive or negative, followed by a mantissa (also known as a significand). It is a real (fractional) number, which in turn is multiplied by a number base (or radix) raised by an exponent. As we know, in decimal this number base is 10. Examples: 4.25×10^{1}, -3.35×10^{3}, 10^{-3}, -1.42×10^{2}.

Floating-point representation is essentially scientific notation applied to binary numbers. In binary, the only real difference is that the number base equals 2 instead of 10. We would therefore write floating-point numbers in the following form:

$$+/- \, mantissa \times 2^{exponent}.$$

When we write numbers in scientific notation (whether they be binary or decimal) we can write them in various ways. In decimal we could write $1.5 \times 10^{2}, 15 \times 10^{1}$ and 150×10^{0}, but still all these numbers have exactly the same value. This provides flexibility, which also, unfortunately leads to confusion. In order to try and resolve this confusion a common set of rules known as **normalized**

scientific notation are used to define how numbers are normally written in scientific notation.

In the normalized form we have a single key rule: we choose an exponent so that the absolute value of the mantissa remains greater than or equal to 1, but less than the number base. Let's look at a couple of examples!

If we had the decimal number 500 and wanted to write it in scientific notation we can write it as either 500×10^0 or 50×10^1. In normalized form though, we would apply the rule above and move the radix point so that only a single digit, greater than or equal to 1 and less than (in this case) 10 were to the left of the radix point. This case means moving our radix point two places to the left so we had: 5.0×10^2. Examples of normalized floating-point numbers: $+1.23456789 \times 10^1$, $-9.987654321 \times 10^{12}$, $+5.0 \times 10^0$. These are not normalized examples: $+11.3 \times 10^1$ (significand is greater than radix (=10); -0.00003×10^7 (significand < radix); $-4.0 \times 10^{1/2}$ (exponent not integer).

Let's look at a slightly more complicated example, this time in binary. What if we had the binary number 10.1_2? Again, we apply the rules: we need to have a mantissa that is greater than or equal to 1 and less than our number base (which this time is 2). That would mean our mantissa would need to be 1.01×2^1. To get back to our original number we would need to move our radix point 1 place to the right. What does a right side mean? That means the exponent is positive.

The following example is a little more tricky: we want to write the number '0.111_2' in normalized scientific notation. Again, we apply the rules. We need a mantissa greater than or equal to 1 and less than 2: $1.11 \times 2^?$. Now, to go back to our original number we need to move our radix point 1 place to the left. That means our exponent is negative. That gives us: 1.11×2^{-1}.

5.2 Converting from decimal to binary (repetition of previous training)

Like a slight reminder, check out a couple of examples of converting floating-point numbers from decimal to binary radix. In order to make it, please, follow a simple iterative procedure:

1) multiply the number by exponent;

2) separately select the integer part of the number and translate it into binary code;

3) start with the decimal fraction: the fractional part is multiplied by 2;

4) separate the integer part of the intermediate result (it is '0' or '1');

5) stop if the fractional part of the intermediate result is 0 (terminated binary) or a result you've seen before (repeating binary);

6) repeat from step 3 with the fractional part of the result until the result reaches 0 or starts to repeat;

7) record the integer part and the fractional part of the result.

Example 5.1. Let's convert the number 3.4625×10^1 follow the steps below.

- Let's write the number without exponent: $3.4625 \times 10^1 = 34.625$.
- Let's convert separately with the 34_{10}, which equals 100010_2 .
- Separately convert the fractional part:

$2 \times .625 = 1.25$ (save the integer part),

$2 \times .25 = 0.5$ (no integer part to save),

$2 \times .50 = 1.00$ (save the integer part).

Let's write them left to right in order: $34.625_{10} = 100010.101_2$.

Example 5.2. Let's convert the number 1.23125×10^1.

- $1.23125 \times 10^1 = 12.3125$
- $12_{10} = 1100_2$
- $2 \times .3125 = 0.625$
  $2 \times .625 = 1.25$
  $2 \times .25 = 0.5$
  $2 \times .5 = 1.0$

Let's write the binary number: $12.3125_{10} = 1100.0101_2$.

5.3 IEEE 754 Representations

Floating-point numbers are used everywhere in modern computing. Whether it be the percentage of the market that have been upgraded to the latest version of iOS, the current position and orientation of your iPhone in space or the amount of money flowing into your bank account. As a result of such a wide use, the format, storing floating-point numbers in memory, has been standardized by the Institute of Electrical and Electronic Engineers. It is called the IEEE 754 standard.

This standard defines a number of different binary representations that can be used to store floating-point numbers in memory:

– half precision – uses 16-bits of storage in total;

– single precision – uses 32-bits of storage in total;

– double precision – uses 64-bits of storage in total;

– quadruple precision – uses 128-bits of storage in total.

In each of these cases, their basic structure is, as follows:

$$(-1)^{sign} \times mantissa \times 2^{exponent}.$$

When it comes to a storedge of floating-point numbers in memory, only three critical parts of that basic structure are stored: sign, exponent, mantissa. The fig 5.1 below shows how these parts are stored in memory. The most significant bit is the sign bit.

Figure 5.1 – Floating-point representation

All four binary representations defined in the IEEE 754 standard and have the most significant bit, as a sign bit and use it to store the **sign** of the overall number. If the sign bit is clear (a value of 0) the overall number is positive. If the bit is set (a value of 1) the number is negative.

The **exponent** represents the power to which the mantissa will be raised. There is always a fixed number of exponent bits when storing a floating point representation in memory, and the exact number of bits which use is defined by the particular IEEE 754 representation (single precision, double precision etc).

In all cases, the exponents in each of these representations need to be able to represent both positive exponents (in order to represent very large numbers) and negative exponents (in order to represent very small numbers). To avoid the complications of the need to store the exponents in two's complement format, something called an exponent bias is used.

Exponent bias is where the value stored for the exponent is offset from the actual exponent value by a bias. The bias is simply a number that is added to the exponent to ensure that the value that is stored is always positive. The table below shows the number of bits used for the exponent in each of the formats, the allowed range of values the different exponents which can be applied the bias along with the allowed values after applying the bias:

Table 5.1 – Characteristics of standard formats

Representation	Bits	Normal Range (Pre Bias)	Bias	Modified Range (Post Bias)	Notes
Half Precision (2 bytes)	5	-14 to +15	+15	+1 to +30	Biased values of 0 (all bits clear) and 31 (all bits set) have special meaning.
Single Precision (4 bytes)	8	-126 to +127	+127	+1 to +254	Biased values of 0 (all bits clear) and 255 (all bits set) have special meaning.
Double Precision (8 bytes)	11	-1022 to +1023	+1023	+1 to +2046	Biased values of 0 (all bits clear) and 2047 (all bits set) have special meaning.
Quadruple Precision (16 bytes)	15	-16382 to +16383	+16383	+1 to +32766	Biased values of 0 (all bits clear) and 32767 (all bits set) have special meaning.

Exponent bias can be calculated by the formula: $\left(2^{m-1}-1\right)+p$, m – the number of bits allocated to represent the order, p – actual order of normalized number.

Examples of exponents in short floating point which uses 8-bits for the exponent, which we want to range from -128 to +127 (single precision):

– the exponent is 135, then 135 = 135-127 = 8, therefore we have 2^8 ;

– the exponent is 120, then 120 = 120-127 = -7, therefore we have 2^{-7} .

In the IEEE 754 representations, the **mantissa** is expressed in normalized form. The formats follow the same rules for normalization as we saw with Scientific Notation, and put the radix point after the first non-zero digit.

As we are expressing our numbers in binary, we knowing that the first non-zero digit will always be a 1 (after all we can only have 1's or 0's). Taking this into consideration, we are able to drop that first bit, simply assuming it is there, and instead gain an additional (implicit) bit of precision. When numbers are stored, we only store the part of the mantissa that represents the fractional part of the number, the part to the right of the radix point. This is provided by the IEEE 754 standard.

In accordance with this standard, the highest binary digit (the whole part) of the mantissa of a real number in the normalized form is always ′1′. In this case, the entire mantissa is unnecessary to be stored in the memory.

Let's look at some examples.

1) Number $A_{10} = -30.125$ must be written in 4 bytes:

$A_2 = -11110.001 = -1.1110001 \times 2^{+4}$.

Mantissa = 1.1110001, frac = 1110001, $p = 4_{10} \Rightarrow$

bias is $\left(2^{8-1} - 1\right) + 4 = 127 + 4 = 131_{10} = 10000011_2$.

Binary float-point representation is

$$\underbrace{0}_{sign}\ \underbrace{100\ 0001}_{\exp}\ \underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 0000}_{frac}.$$

2) Given the bit string: 0 100 0000 0 101 0000 0000 0000 0000 0000.

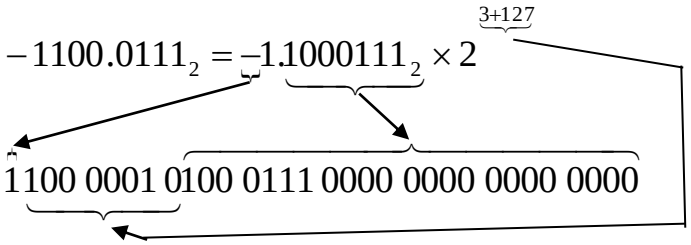
What floating point number does it represent?

We see that this is a positive, normalized number.

$$10000000_2 = 128_{10} \Rightarrow \exp = 128 - 127 = 1.$$

So, this number is: $1.101_2 \times 2^1 = 11.01_2 = 3.25_{10}$.

3) Representing values $-12.4375_{10} = -1100.0111_2$



5.4 Rules for adding a floating-point format numbers

The basic rules for adding (subtracting) numbers of a floating-point format can be formulated as follows:

• to add two floating-point values, they have to be aligned so that they have the same exponent and the same grid of the mantissa:

– shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1;

– shifting the mantissa RIGHT by 1 bit INCREASES THE EXPONENT by 1.

• after addition, the sum may need to be normalized;

• potential errors include overflow, underflow and inexact results.

Decimal example: $3.25 \times 10^3 + 2.63 \times 10^{-1} = ?$

The first step – aligning decimal points, the second step – adding:

$$3.25 \times 10^3 + 0.000263 \times 10^3 = 3.250263 \times 10^3.$$

The third step – normalizing the result: already normalized!

The binary example: $0.25 + 100 = ?$

1. Convert into binary in short representation (2 bytes)

$$0.25_{10} = 0.01_2 = 1.0_2 \times 2^{-2} \Rightarrow 0\ \underbrace{01111101}_{-2+127=125}\ \underbrace{0000000000\ 0000000000\ 000}_{\text{fraction}}$$

$$100_{10} = 1100100_2 = 1.1001 \times 2^6 \Rightarrow 0\ \underbrace{10000101}_{6+127=133}\ \underbrace{1001000000\ 0000000000\ 000}_{\text{fraction}}$$

2. Align radix points, it's needed to shift the mantissa from the first term to the right, because its exponent is smaller. Choose to shift the 0.25, in order to increase its exponent. Shift by ($10000101_2 - 01111101_2 = 1000_2 = 8_{10}$) 8 places.

$0\underbrace{0111110}_{125}1\underbrace{000\ 0000\ 0000\ 0000\ 0000\ 0000}$ (original value)

$0\underbrace{0111111}_{126}0\underbrace{100\ 0000\ 0000\ 0000\ 0000\ 0000}$ (shifted 1 place)

Note that hidden bit is shifted into MSB (most significant bit) of mantissa.

$0\underbrace{0111111}_{127}1\underbrace{010\ 0000\ 0000\ 0000\ 0000\ 0000}$ (shifted 2 places)

$0\underbrace{100\ 0000}_{128}0\underbrace{001\ 0000\ 0000\ 0000\ 0000\ 0000}$ (shifted 3 places)

$0\underbrace{100\ 0000}_{129}0\underbrace{000\ 1000\ 0000\ 0000\ 0000\ 0000}$ (shifted 4 places

$0\underbrace{100\ 0000}_{130}1\underbrace{000\ 0100\ 0000\ 0000\ 0000\ 0000}$ (shifted 5 places)

$0\underbrace{100\ 0001}_{131}0\underbrace{000\ 0010\ 0000\ 0000\ 0000\ 0000}$ (shifted 6 places)

$0\underbrace{100\ 0010}_{132}0\underbrace{000\ 0001\ 0000\ 0000\ 0000\ 0000}$ (shifted 7 places)

$0\underbrace{100\ 0010}_{133}1\underbrace{000\ 0000\ 1000\ 0000\ 0000\ 0000}$ (shifted 8 places)

3. Add of mantissas

$$\begin{array}{ll} & 0.000\ 0000\ 1000\ 0000\ 0000\ 0000 \quad (0.25_{10}) \\ + & 1.100\ 1000\ 0000\ 0000\ 0000\ 0000 \quad (100_{10}) \\ \hline & 1.100\ 1000\ 1000\ 0000\ 0000\ 0000 \end{array}$$

4. Normalize the result – get the "hidden bit" to be a 1. The example is already normalized. Result is

$$0\underbrace{10000101}_{133}\underbrace{10010001000\ 00...00} = 1.10010001_2 \times 2^{133-127} = 1100100.01_2 = 100.25_{10}$$

Check out the four-step algorithm (fig. 5.2) and, afterwards, the example.

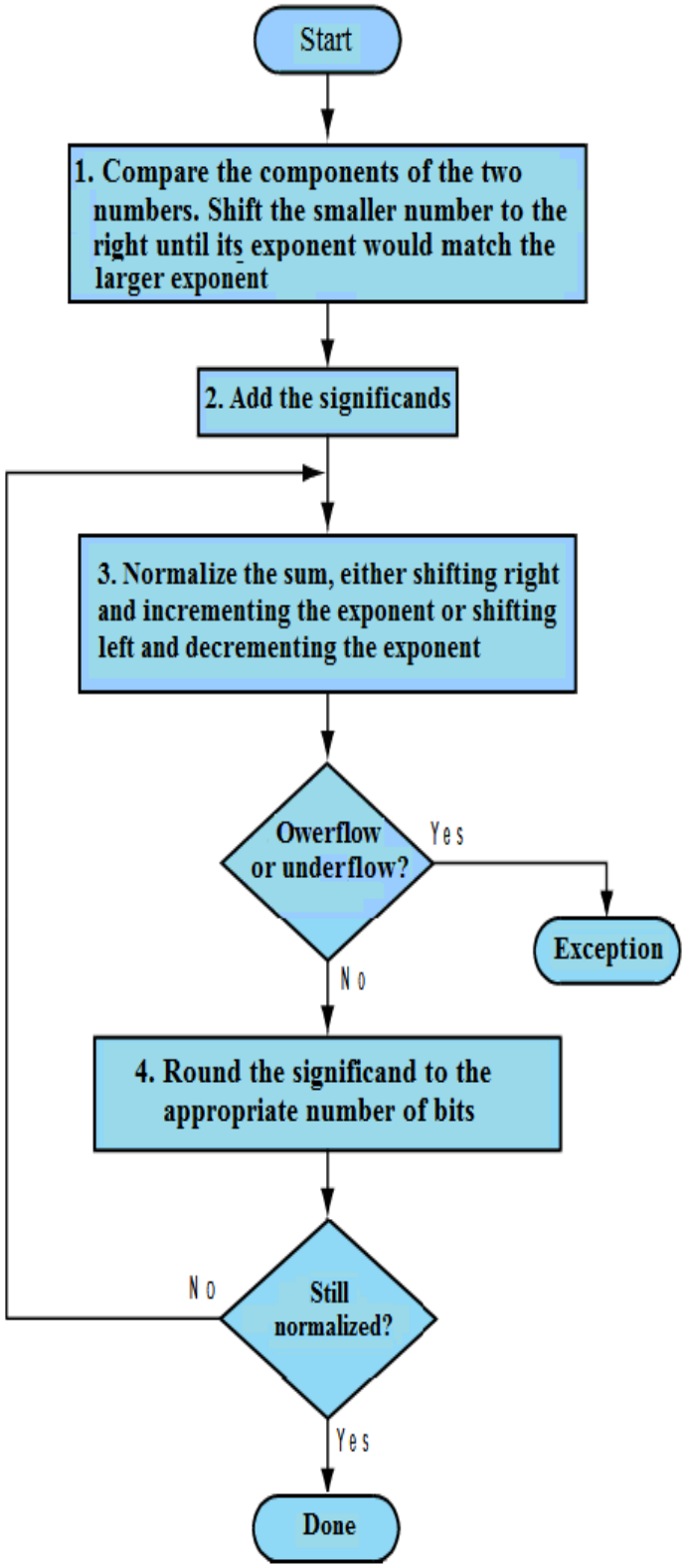


Figure 5.2 – Floating-point addition

Consider the example with 4-digit precision: $0.5 + (-0.4375)$

and binary form: $1.000_2 \cdot 2^{-1} + (-1.110_2 \cdot 2^{-2})$.

1. Align binary points, shift number with smaller exponent:

$$1.0_2 \cdot 2^{-1} + \left(-0.111_2 \cdot 2^{-1}\right).$$

2. Add significands:

$$1.000_2 \cdot 2^{-1} + \left(-0.111_2 \cdot 2^{-1}\right) = 0.001_2 \cdot 2^{-1}$$

3. Normalize result and check for over/underflow:

$$1.0_2 \cdot 2^{-4} - \text{with no over/underflow.}$$

4. Round and renormalize if necessary:

$$1.0_2 \cdot 2^{-4} = 0.0625_{10}$$

5.5 Floating-point precision and rounding

Often due to the small amount of binary digits allocated for storing the mantissa, there is a fractional part representation error of the output number in the binary system. It is calculated by the formulas:

$$\underset{\text{absolute error } \Delta}{\Delta = X_{input} - X_{output},} \qquad \underset{\text{relative error } \delta}{\delta = \left(\Delta / X_{output}\right) \cdot 100\%,}$$

where X_{input} – input number , X_{output} – output number.

All fraction bits are significant for relative precision. Relative precision for floating-point numbers of a single precision format is approx 2^{-23} because we have 23 bits in the mantissa. That is equivalent to 6 decimal digits of precision: $23 \times \log_{10} 2 \approx 23 \cdot 0.3 \approx 6$.

For double precision we have 52 bits in the mantissa and approx 16 decimal digits of precision: $52 \times \log_{10} 2 \approx 52 \cdot 0.3 \approx 16$.

The floating-point addition associative is not associative. Its need to beware! Look at the example:

Associativity law for the addition: $A + (B + C) = (A + B) + C$.

Let $A = -2.7 \cdot 10^{23}$, $B = 2.7 \cdot 10^{23}$ and $C = 1.0$ in single precision format.

$$A + (B + C) = -2.7 \cdot 10^{23} + (2.7 \cdot 10^{23} + 1.0) = -2.7 \cdot 10^{23} + 2.7 \cdot 10^{23} = 0.0$$

$$(A + B) + C = (-2.7 \cdot 10^{23} + 2.7 \cdot 10^{23}) + 1.0 = 0.0 + 1.0 = 1.0$$

The result is approximate... Why the smaller number disappeared?

Arithmetic operations on floating-point values compute results that cannot be represented in the given amount of precision. So, the result should be rounded. There are many ways of rounding. They each have "correct" uses, and exist for different reasons. The goal is to compute the round result, which is as "correct" as possible. There are even arguments about what is really correct.

Rounding in binary system is similar, but it still may cause some difficulties. The biggest challenge is rounding fractions. For example, it may not be obvious right away why the fraction 0.11101 when rounded to 2 places after the decimal point results in the integer 1.

There are five rounding methods, as defined by the IEEE-754 standard, and most of them are pretty straightforward. The first two rounds to a nearest value (ties to even and ties away from zero); the others are called directed roundings: towards zero, towards positive infinity and towards negative infinity. These rules are easier to understand using decimal system as an example (Table 5.2).

Table 5.2 – Example of rounding to integers using the IEEE 754 rules

Mode/Example value	+11.5	+12.5	-11.5	-12.5
to nearest, ties to even	+12.0	+12.0	-12.0	-12.0
to nearest, ties away from zero	+12.0	+13.0	-12.0	-13.0
toward 0	+11.0	+12.0	-11.0	-12.0
toward +∞	+12.0	+13.0	-11.0	-12.0
toward -∞	+11.0	+12.0	-12.0	-13.0

During rounding towards 0 (also called truncation) it's needed to figure out how many bits (digits) are available and take that majority bits for the result and throw away the rest. This influences on making the represented value closer to 0. Example: the number is '0.7783'. If 3 decimal places available then we have '0.778'; if 2 decimal places available then we have '0.77'.

Round up – regardless of the value, round towards $+\infty$. Rounded result is close, but not less than true result. Example: rounding '1.23' if 2 decimal places gives '1.3'; '-2.86' if 2 decimal places – '-2.8'.

Round down – regardless of the value, round towards $-\infty$. Rounded result is close, but not bigger than true result. Example: rounding '1.23' if 2 decimal places gives '1.2'; '-2.86' if 2 decimal places – '-2.9'.

Examples for binary numbers, rounding to 2 digits after radix point:

• Round towards zero (truncate): $1.1101_2 \Rightarrow 1.11_2$, $1.001_2 \Rightarrow 1.00_2$,

$$-1.1101_2 \Rightarrow -1.11_2,\ -1.001_2 \Rightarrow -1.00_2;$$

• Round up: $1.1101_2 \Rightarrow 10.00_2$, $1.001_2 \Rightarrow 1.01_2$;

• Round down: $1.1101_2 \Rightarrow 1.11_2$, $1.001_2 \Rightarrow 1.00_2$.

Directed roundings are pretty straightforward! 'The round to the nearest' and 'ties to even' rules usually is pretty hard to understand as directed roundings. The general rules of rounding come from comparing the original number and the middle between two rounding options.

The general rule during rounding binary fractions to the **n**-th place prescribes to check the digit following the **n**-th place in the number. If it's a **0**, then the number should always be rounded down. If, instead, the digit is **1** and any of the following digits are also **1**, then the number should be rounded up. If, however, all of the following digits are **0**s, then a tie breaking rule must be applied and usually it's the 'ties to even'. This rule says that the number should be rounded to the number that has **0** at the **n**-th place.

To demonstrate those rules in action let's round some numbers to 2 places after the radix point:

• 0.11001 – rounds down to 0.11, because the digit at the 3-rd place is 0;

• 0.11101 – rounds up to 1.00, because the digit at the 3-rd place is 1 and there are following digits of 1 (5-th place);

• 0.11100 – rounds to 0.11 – apply the 'ties to even' tie breaker rule and round up because the digit at 3-rd place is 1 and the following digits are all 0's;

• $1.1111_2 \Rightarrow 10.00_2$, $1.1101_2 \Rightarrow 1.11_2$, $1.001_2 \Rightarrow 1.00_2$;

• $-1.1101_2 \Rightarrow -1.11_2$ (1/4 of the way between), $-1.001_2 \Rightarrow -1.00_2$.

5.6 Exercises

Select a task from Table 5.3 according to your number in the group list.

Table 5.1 – Individual task

Option	A_{10}	B_{10}	Option	A_{10}	B_{10}
1	16.53	–36.29	16	36.63	–57.75
2	23.47	53.67	17	–59.97	37.73
3	–65.38	15.74	18	38.12	68.59
4	42.89	33.52	19	–71.17	39.93
5	–76.74	17.44	20	40.14	–56.65
6	–32.19	–80.88	21	–72.27	43.36
7	99.57	–15.97	22	44.51	–69.96
8	51.68	28.79	23	63.85	46.64
9	–35.49	77.39	24	–48.84	–18.81
10	64.25	–19.43	25	61.16	50.15
11	–47.31	29.34	26	–52.29	28.82
12	20.52	45.78	27	26.63	54.47
13	30.32	–55.56	28	58.85	–24.42
14	67.76	31.41	29	22.72	60.76
15	34.54	–87.77	30	–70.73	21.12

1) Convert numbers A_{10} and B_{10} into SMR form of binary numbers. Consider representation precision with 5 numbers after a comma.

2) Represent obtained binary numbers in a normalized floating-point format in 2 bytes (fig. 5.1 and top examples).

3) Determine which exponent is the smaller exponent. Align the grids of the mantissas.

4) Get the 1's complement and 2's complement codes for $+A, -A, +B, -B$.

5) Manually calculate the values of the following expressions in 2's complement forms:

$$+A+(+B),\ +A+(-B),\ -A+(+B),\ -A+(-B).$$

6) Normalize the answer to normal exponential form and convert the results to decimal.

7) Check up result.

5.7 Control questions

1. How many bits are in half precision, single precision double precision and quadruple precision formats?

2. How does the transfer unit from the most significant sign digit taken into account when adding in the two's complement code?

3. Represent the number $A_{10} = 4.5 + N$ (N is your number in the group list) in the single precision format (4 bytes).

4. Operation features of addition in a two's complement modified code during overflow of the bit grid.

6 MULTIPLICATION OF BINARY NUMBERS

6.1 Commonly rules

Multiplication of two fixed point binary number in signed-magnitude representation (SMR) is done by the process of successive shift and add operation.

In the multiplication process bits of the multiplier are considered and the least significant bit considers first. If the multiplier bit is 1, the multiplicand is copied or 0 is copied. The numbers copied down in successive lines are shifted one position to the left from the previous number. Afterwards, when numbers are added, their sum form the product. The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, sign of the product is positive, otherwise - negative.

For example 1, let's multiply +12 by +15, which in binary will be 1100 by 1111.

$$\begin{array}{rcccccccc} & & & & 1 & 1 & 0 & 0 & (Multiplicand) \\ & & & \times & 1 & 1 & 1 & 1 & (Multiplier) \\ \hline & & & & 1 & 1 & 0 & 0 & \\ & & & 1 & 1 & 0 & 0 & & \\ & & 1 & 1 & 0 & 0 & & & \\ & 1 & 1 & 0 & 0 & & & & \\ \hline 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \end{array} ;$$

$$10110100_{2} = 180_{10}; \; 12_{10} \times 15_{10} = 180_{10}.$$

The answer is correct.

So. When multiplying numbers **in SMR**:

1. Sign and significant bits are processed separately. To determine the sign of the result, the sign bits of the operands are summed using the operation XOR.

2. Multiplication of numbers is done step by step adding the shifted value of the multiplicand and partial sum with a non-zero bit of the corresponding bit of the multiplier.

When we are multiplying numbers in 2’s complement form:

1. The number of bits of a multiplicand is increased by the number of the bits of a multiplier. Insignificant zeros are added to the left of the multipliers.

2. Multiplicand and multiplier are represented in two’s complement codes.

3. The result sign is obtained automatically.

4. Multiplication of numbers is made step by step by adding the shifted value of the multiplicand and partial sum if there is a non-zero bit in the corresponding bit of the multiplier.

5. If multiplier is a negative number, we need to do a correction. And when multiplying the multiplicand *A* by the sign bit of the multiplier *B*, we must sum with negative *A* (*-A*) in 2’s complement code (but don't positive *A*).

Example 2. Multiply *A * (-B)* in the 2’s complement code, *A*=10, *B*=13.

Number	***+A***	***-A***	***+B***	***-B***
Decimal	10	-10	13	-13
Binary	1010	-1011	1101	-1101
SMR (1 byte)	0,1010	1,1010	0,1101	1,1101
2’s complement	0,1010	1,0110	0,1101	1,0011

When multiplied by the sign bit, we write the multiplicable *A* as *-A* is written in 2’s complement form (step 5)

$$
\begin{array}{cccccccccc}
0, & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & (Multiplicand\ A) \\
 & & & & 1, & 0 & 0 & 1 & 1 & (Multiplier\ B) \\
 & & & & & 1 & 0 & 1 & 0 & \\
 & & & & 1 & 0 & 1 & 0 & & \\
 & & & 0 & 0 & 0 & 0 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
1, & 0 & 1 & 1 & 0 & & & & & \\
\hline
1, & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 &
\end{array}
\quad ,
$$

Inversion: 1,10000001 +1 = 1,10000010

Check up: $0,10000010_2=130_{10}$

6.2 Floating-point multiplication

• Multiplying floating point values does not requires re-alignment - realigning may lead to loss of significance.

• After multiplication, the product may need to be normalized.

• Potential errors include overflow, underflow and inexact results.

The figure 6.1 shows the multiplication algorithm.

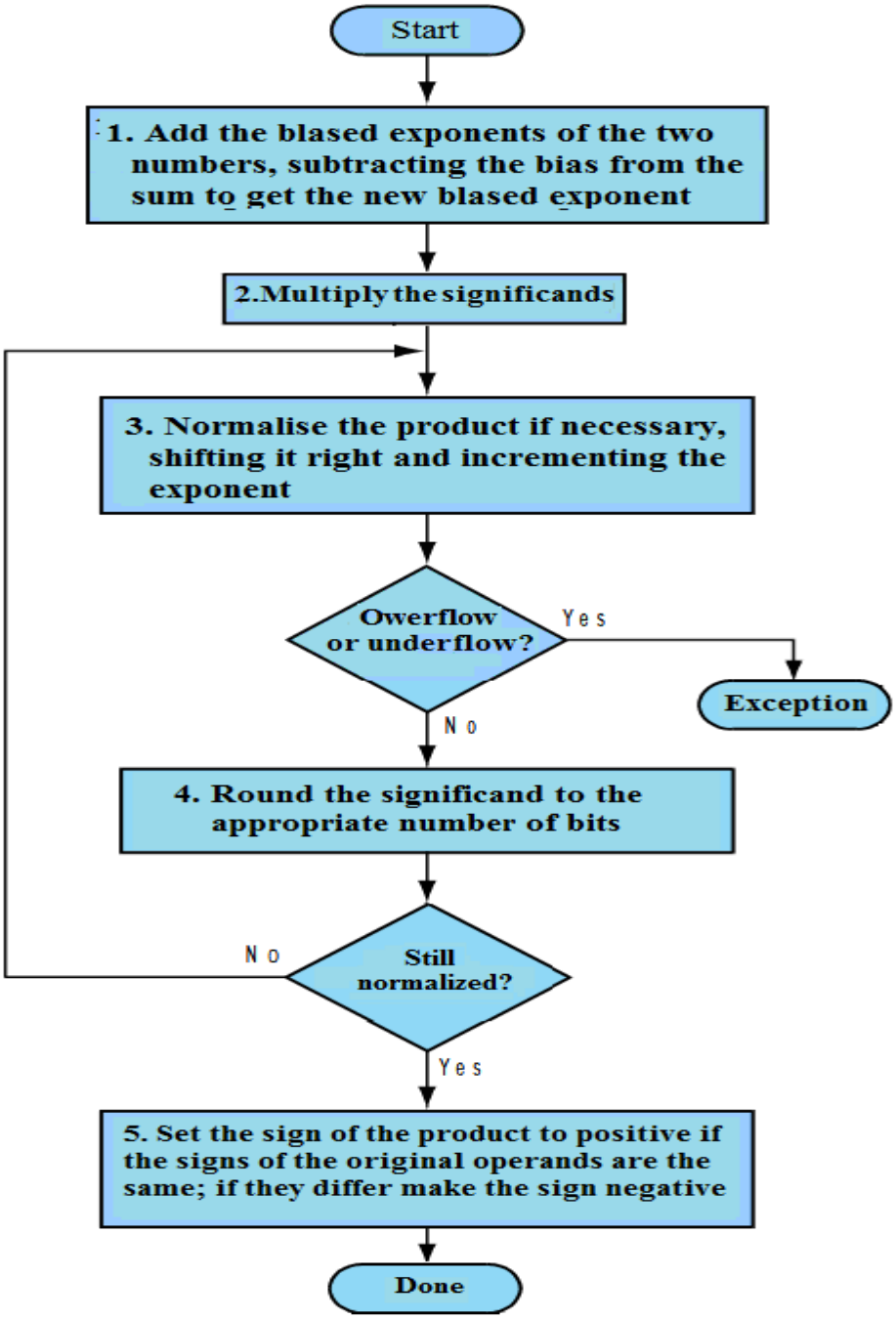


Figure 6.1 – Multiplication algorithm

Example 6.1 on decimal values given in scientific notation:

$$3.0\times10^{-1}\times0.5\times10^{-2}=?$$

It's the simple algorithm: multiply mantissas and add exponents.

$$3.0\times10^{-1}\times0.5\times10^{-2}=3.0\times0.5\times10^{-1+(-2)}=1.5\times10^{-3}$$

Example 6.2 in binary. $A=4\times10^{5}$, $B=12\times10^{60}$.

Multiply A_2=0 10000100 0100 and B_2=1 00111100 1100.

The used mantissa is only 4 bits. Also, in the floating-point format the mantissa and the exponent are separate numbers.

1. Add the blased exponents of the two numbers:

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \quad (=5+127)\\ + & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \quad (=-67+127)\\ \hline & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \quad (=5-67+127) \end{array},$$

subtracting the bias from the sum to get the new exponent:

$$\begin{array}{rcccccccc} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \quad (=5-67+127)\\ + & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \quad (1's\ complement\ to\ 127)\\ \hline & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \quad (decimal\ result=65) \end{array}$$

2. Multiply the significands: mantissas multiplication and don't forget the hidden bit

$$\begin{array}{rcccccccc} & & & & & 0 & 1 & 0 & 0\\ \times & & & & & 1 & 1 & 0 & 0\\ \hline & & & & & 0 & 0 & 0 & 0\\ & & & & 0 & 0 & 0 & 0 & \\ & & & 0 & 1 & 0 & 0 & & \\ & & 0 & 1 & 0 & 0 & & & \\ & 0 & 0 & 0 & 0 & & & & \\ \hline & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}, \quad \text{becomes} \quad 00110000.$$

Put the result back together (and add sign bit): 1 110 0000 0 011 0000

3. No need to normalize the product.

4. Round the significand to the appropriate number of bits. No need to round!

5. This is the value stored: 1 11000000 0110000

6.3 Exercises

1) Choose the task from the Table 6.1 according to your number in group list.

Table 6.1 – Individual task

Variant	multiplicand A_{10}	Multiplier B_{10}	Variant	multiplicand A_{10}	Multiplier B_{10}
1	16	–36	16	3	–27
2	28	33	17	–5	17
3	–25	15	18	8	20
4	12	33	19	–4	19
5	–36	17	20	4	–26
6	–32	–10	21	–17	13
7	9	–15	22	14	–19
8	11	28	23	23	6
9	–35	27	24	–8	–5
10	4	–19	25	21	2
11	–7	29	26	–5	28
12	20	15	27	3	35
13	30	–5	28	9	–12
14	6	14	29	13	6
15	34	–8	30	–7	21

2) Convert numbers A_{10} and B_{10} into binary code in SMR and 2's complement codes. Use numbers $+A_2$, $+B_2$, $–A_2$, and $–B_2$.

3) Perform multiplication of binary numbers A_2 and B_2. Use 2's complement code for different combinations: ($+A_2$, $+B_2$), ($+A_2$, $–B_2$), ($–A_2$, $+B_2$) and ($–A_2$, $–B_2$).

6.4 Control questions

1. Features of multiplying integer binary numbers with a sign in SMR and 2's complement form.

2. What does the term "partial sum" mean?

3. If the multiplicand or multiplier is a negative number, what should in mind when performing a multiplication operation?

7 DIVISION OF BINARY NUMBERS

Division algorithms are divided into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration. Newton-Raphson and Goldschmidt algorithms fall into this category. We consider the simplest algorithms.

7.1 Division of fixed-point numbers

The division operation is performed by shifting operations and adding 2's complement code to the adder. Division schemes with remainder recovery (restoring division) and without remainder recovery (non-restoring division) are distinguished.

The **non-restoring division** is performed according to the following scheme:

1. Equalize the bit grid of the dividend and divider (divisor).

2. If the dividend $A > 0$, then $A = A - B$. If the resulting difference $A > 0$, then increase by one the value of the integer part of the quotient C and go to step 3, otherwise finish the formation of the integer part of the quotient and go to step 4.

3. Repeat step 2 until the resulting difference A is less than 0.

4. Perform a linear shift of A by one digit to the left.

5. If $A > 0$, then find $A = A - B$, otherwise $A = A + B$.

6. If $A > 0$, then assign 1 to the i-th order of fractional part, otherwise assign 0.

7. If the number of fractional digits is less than the required number, then go back to step 4.

8. Done. Doing decimal conversion.

Example 7.1 Find the result of dividing $C = A/B$ according to the algorithm without reconstruct a remainder: A = 10, B = 3. Find the three digits of the fractional part.

Step 1. Get binary values and aligned the bit grid.

Number	**+*A***	**+*B***	**-*B***
Decimal	10	3	-3
Binary	1010	0011	-0011
SMR (1 byte)	0,1010	0,0011	1,0011
Modified 2's complement	00,1010	00,0011	11,1101

Steps 2-3. Find the integer part of the quotient C. Perform the current subtraction operation until the result becomes negative or zero:

$$C_2 = 00 \quad 2)\ \begin{array}{r} 00\ \ 1010 \\ 11\ \ 1101 \\ \hline 00\ \ 0111 \end{array} > 0 \quad \text{increment C} \quad 3)\ C_2 = 01$$

$$2)\ \begin{array}{r} 00\ \ 0111 \\ 11\ \ 1101 \\ \hline 00\ \ 0100 \end{array} > 0 \quad \text{increment C} \quad 3)\ C_2 = 10$$

$$2)\ \begin{array}{r} 00\ \ 0100 \\ 11\ \ 1101 \\ \hline 00\ \ 0001 \end{array} > 0 \quad \text{increment C} \quad 3)\ C_2 = 11$$

$$2)\ \begin{array}{r} 00\ \ 0001 \\ 11\ \ 1101 \\ \hline 11\ \ 1110 \end{array} < 0 \quad \text{fix point} \quad 3)\ C_2 = 11.$$

Steps 4-6. We proceed to the search for the fractional part.

4|) Shift one bit to the left of the last value of A. Now $A_{2'sc} = 11\ 1100$.

5) Since, A<0 then $A_{2'sc} = -A_{2'sc} + B_{2'sc}$.

$$+\ \begin{array}{r} 11\ \ 1100 \\ 00\ \ 0011 \\ \hline 11\ \ 1111 \end{array} < 0$$

6) The result is negative, so the next bit of the fractional part is zero: $C_2 = 11.0$

Continue from step 4.

4) Shift one bit to the left, now $A_{2'sc} = 11\ 1110$.

5) Since, A<0 then $A_{2'sc} = -A_{2'sc} + B_{2'sc}$.

$$+\begin{array}{r} 11\ 1110 \\ 00\ 0011 \\ \hline 00\ 0001 \end{array} > 0 .$$

6) The result is positive, so the next bit of the fractional part is one: $C_2 = 11.01$ Continue from step 4.

4) Shift one bit to the left, now $A_{2'sc} = 000\ 0010$.

5) Since, A>0 then $A_{2'sc} = +A_{2'sc} + (-B_{2'sc})$.

$$+\begin{array}{r} 00\ 0010 \\ 11\ 1101 \\ \hline 11\ 1111 \end{array} < 0 .$$

6) The result is negative, so the next bit of the fractional part is 0: $C_2 = 11.010$

Step 7-8. After conversion to decimal code, the result is $C_{10} = 3.2510$.

The result contains the absolute error Δ:

$\Delta = C_{\text{expected}} - C_{fact} = 3.33333 - 3.2510 = 0.08233$.

Error occurred due to the small number of digits in the quotient.

The **restoring division** is performed according to the following scheme:

1. Equalize the bit grid of the dividend and divisor.

2. If the dividend $A > 0$, then $A = A - B$. If the resulting difference $A > 0$, then increase by one the value of the integer part of the quotient C and go to step 3, otherwise finish the formation of the integer part of the quotient and go to step 4.

If the value $A = 0$ is obtained, then increase the value of the integer part of the quotient C by 1, complete the formation of the quotient, fixing the position of the comma, which separates the integer part of the quotient from its fractional part equal to 0 in this case, and proceed to the step 9.

3. Repeat step 2 until the resulting difference A is less than 0.

4. Recover the remainder $A = A + B$.

5. Perform a linear shift of A by one digit to the left.

6. Find $A = A - B$.

7. If $A < 0$, then assign 0 to the i-th digit of the fractional part and go to step 4. Otherwise assign 1 the i-th fractional part and go to step 5.

8. If the number of fractions is sufficient or $A = 0$, then the formation of the fractional part is completed.

9. Done.

Example 7.2 Find the result of dividing $C = A/B$ according by the algorithm of division with recovery of the remainder: $A = 12$, $B = 5$. Find the three digits of the fractional part.

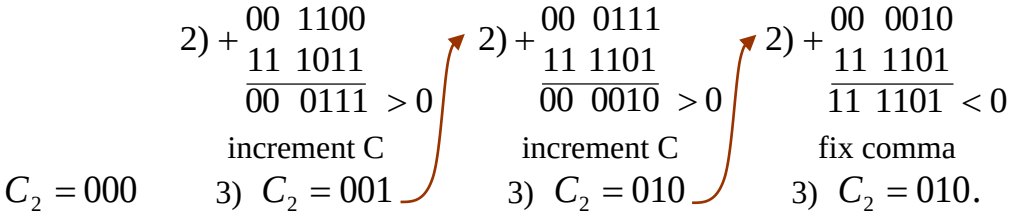
Let's put in that $C_{\text{expected}} = 2.4$.

Step 1. Get binary values and aligned the bit grid.

Number	**+*A***	**+*B***	**-*B***
Decimal	12	5	-5
Binary	1100	0101	-0101
SMR (1 byte)	0,1100	0,0101	1,0101
Modified 2's complement	00,1100	00,0101	11,1011

Steps 2-3. Find the integer part of the quotient C.

Perform the current subtraction operation until the result becomes negative or zero:



The calculation of the integer part is completed.

Steps 4-6.

4) Reconstruct the remainder: $A_{2'sc} = A_{2'sc} + B_{2'sc}$, $A_{2'sc} = 11\ 1101$.

$$+\begin{array}{r} 11\ \ 1101 \\ 00\ \ 0101 \\ \hline 00\ \ 0010 \end{array}$$

5) Shift one bit to the left in *A*, now $A_{2'sc} = 00\ \ 0100$.

6) Since, A>0 then we do $A_{2'sc} = +A_{2'sc} + (-B_{2'sc})$.

$$+\begin{array}{r} 00\ \ 0100 \\ 11\ \ 1011 \\ \hline 11\ \ 1111 \end{array} < 0$$

7) The result is negative, so the next bit of the fractional part is zero: $C_2 = 10.0$.

Repeat the steps from 4-th step.

4) Reconstruct the remainder: $A_{2'sc} = A_{2'sc} + B_{2'sc}$,.

$$+\begin{array}{r} 11\ \ 1111 \\ 00\ \ 0101 \\ \hline 00\ \ 0100 \end{array}, \qquad A_{2'sc} = 00\ \ 0100.$$

5) Shift one bit to the left in *A*, now $A_{2'sc} = 00\ \ 1000$.

6) Since, A>0 then we do $A_{2'sc} = +A_{2'sc} + (-B_{2'sc})$.

$$+\begin{array}{r} 00\ \ 1000 \\ 11\ \ 1011 \\ \hline 00\ \ 0011 \end{array} > 0$$

7) The result is positive, so the next bit of the fractional part is one: $C_2 = 10.01$

Repeat the steps from 5-th step.

5) Shift one bit to the left in *A*, now $A_{2'sc} = 00\ \ 0110$.

6) Since A>0 then we do $A_{2'sc} = +A_{2'sc} + (-B_{2'sc})$.

$$+\begin{array}{r} 00\ \ 0110 \\ 11\ \ 1011 \\ \hline 00\ \ 0001 \end{array} > 0$$

7) The result is positive, so the next bit of the fractional part is one: $C_2 = 10.011$

8) The number of digits after the point is enough. Done.

After conversion to decimal code, the result is $C_{10\,fact} = 2.34510$.

The result contains the absolute error Δ:

$$\Delta = C_{\text{expected}} - C_{fact} = 2.4 - 2.34510 = 0.0549 .$$

Error occurred due to the small number of digits in the quotient.

7.2 Division of binary floating-point numbers

If *A* and *B* are given in normalized form:

$$A = M_A \times 2^{pA},\ B = M_B \times 2^{pB},$$

where M_A, M_B are mantissas, and *pA* and *pB* are orders of numbers *A* and *B*, then their quotient will be equal to

$$C = M_A \times 2^{pA} \div M_B \times 2^{pB} = \left(M_A \div M_B\right) \times 2^{pA-pB}.$$

In the division of floating-point numbers, their mantissa is divided as a fixed-point number, and the orders are subtracted. Potential errors of the float-point division include overflow, underflow, inexact results and attempts to divide by zero.

Afterwards, the division operation for floating-point numbers is performed in five stages.

1 stage. Definition of the quotient by adding modulo two signed digits of the operands.

2 stage. Division of the mantissa operand modules according to the rules for dividing fixed-point numbers.

3 stage. Determining the order of the quotient by subtracting the order of the divisor from the order of the dividend.

4 stage. Normalization of the result and its rounding.

5 stage. Signing the mantissa result.

The first two stages completely coincide with the rules for dividing numbers with a fixed point. The third step is the usual addition in 1’s complement codes.

When dividing normalized numbers, denormalization of the result is possible only to the left and only by one digit. And due to the fact, the mantissa of any normalized number lies within: $2^{-1} \le |M| < 1 - 2^{-n}$.

Then the smallest and largest possible value of the quotient of the mantissa are equal, respectively:

$$|M_{\min C}| = \frac{2^{-1}}{1-2^{-n}} > 2^{-1},\ |M_{\max C}| = \frac{1-2^{-n}}{2^{-1}} = 2 - 2^{-(n-1)} < 2,$$

i.e., the mantissa of the quotient lies within $2^{-1} < |M_C| < 2$.

Therefore, at the fourth stage, it may be necessary to normalize the mantissa of the quotient by shifting it to the right by one bit and increasing the order of the quotient by one. If, before division, shift the dividend by one digit to the right, then at the fourth stage, it may be necessary to normalize the results to the left by one bit.

Example 7.3. Get $A = -5 \times 10^{-6}$, $B = 8 \times 10^{-5}$

$$A_{SMR} = \underbrace{1}_{sign}\ \underbrace{011\ 1100}_{\exp}\ \underbrace{1\ 101}_{mantissa},\quad B_{SMR} = \underbrace{0}_{sign}\ \underbrace{0111101\ 0}_{\exp}\ \underbrace{110}_{mantissa}$$

1 stage. Definer of quotient' sign: $1 \oplus 0 = 1$

2 stage. Mantissas division by one of the methods shown above. It turned out:

$$|M_C| = |M_A| \div |M_B| \approx 00{,}1101$$

3 stage. Determining the order of quotient:

1	1	1	1	0	0	1	*SMR*
0	0	0	0	1	0	1	1'*s comp*
1	1	1	1	1	1	0	

Convert into true exponent: $1111110_2 = 126_{10}$, 126-127=-1

4 stage. The quotient is received the normalized, so only the rounding of the mantissa is performed: $|M_C| = 0{,}111$

5 stage. Result: $M_{SMR} = \underset{sign}{1}\ \underbrace{0111111\ 0}_{\exp}\ \underbrace{111}_{mantissa}$, in decimal it is -7.0×10^{-1}

.

7.3 Exercises

1) Perform a manual division of the two given pairs of numbers, using the scheme restoring and non-restoring division.

Table 7.1 – Individual tasks

	restoring division method		non-restoring division method			restoring division method		non-restoring division method			restoring division method		non-restoring division method	
Variant	Dividend A_{10}	Divisor B_{10}	Dividend A_{10}	Divisor B_{10}	Variant	Dividend A_{10}	Divisor B_{10}	Dividend A_{10}	Divisor B_{10}	Variant	Dividend A_{10}	Divisor B_{10}	Dividend A_{10}	Divisor B_{10}
1	23	7	33	10	11	17	4	27	8	21	28	9	22	6
2	24	5	34	11	12	18	5	28	9	22	29	7	43	14
3	25	7	35	8	13	19	6	29	8	23	30	8	44	13
4	26	8	36	10	14	20	6	30	9	24	31	7	25	6
5	27	6	37	9	15	21	6	31	8	25	32	6	26	7
6	28	6	38	8	16	22	5	32	9	26	33	9	27	7
7	29	8	39	9	17	23	8	17	5	27	34	8	12	3
8	30	9	40	12	18	24	7	18	4	28	35	9	13	4
9	31	8	41	13	19	25	8	19	5	29	36	8	14	4
10	32	7	42	12	20	26	6	20	5	30	37	9	19	4

2) The result of the division must be shown in SMR and decimal codes.

7.4 Control questions

1. Why does the quotient value obtained after division according to the above schemes not always coincide with the expected one?

2. What is the difference of the division scheme without restoration and division scheme with the restoration?

3. Draw a diagram of the algorithm performing a division during the formation of the integer part of the quotient.

4. Draw a diagram of the algorithm performing a division without recovery, as well as with recovery during the formation of fractional part of the quotient.

BIBLIOGRAPHY

1 Murdocca M. J., Heuring V. P. Principles of computer architecture. – NY : Prentice Hall, 1999. – 640 p.

2 Марченко А.Г., Смикодуб Т.Г. Електронно-обчислювальні машини та мікропроцесорні системи: Навчальний посібник. – Миколаїв : НУК, 2007. – 176 с.

3 Dumas J. D. Computer Architecture: Fundamentals and Principles of Computer Design. – Boca Raton : CRC Press, 2016. – 462 p.

4 Catsoulis J. Designing Embedded Hardware. : O'Reilly, 2005, – 400 p.

5 Veen A. H. Dataflow machine architecture. // ACM Computing Surveys. – 18 (4), 1986. pp. 365–396. doi:10.1145/27633.28055

6 Clarke T. J. W., Gladstone P., MacLean C., Norman A. C. SKIM – The S, K, I Reduction Machine. // LISP Conference, 1980:– pp. 128-135

7 Hopfield J. J. Neural networks and physical systems with emergent collective computational abilities. // Proceedings of the National Academy of science of the USA. 1982. 79 (8). : –pp. 2554–-2558.

8 Blaise Barney Introduction to Parallel Computing. / Lawrence Livermore National Laboratory [Електронний ресурс]. – Режим доступу: https://computing.llnl.gov/tutorials/parallel_comp/

9 Irabashetti P. S. Parallel processing in processor organization // International Journal of Advanced Research in Computer and Communication Engineering -Vol. 3, Issue 1, 2014. ,– pp. 5150-5153.

10 Hollings Ch. An Analysis of Nonpositional Numeral Systems // The Mathematical Intelligencer – 2009, Volume 31, pp 15–23.

11 Sargunar J. Introduction to Computer Science. – ITL Education Solutions Limited. New Delhi : Pearson, - 2011. – 483 p.

12 Tanenbaum A. S., Austin T., Structured Computer Organization. – NJ : Prentice Hall, 2013. – 800 p.

13 Plantz Robert G. Introduction to Computer Organization with x86-64 Assembly Language & GNU/Linux 2015. – 568p.

14 Dandamudi S.P. Fundamentals of Computer Organization and Design, - Springer Verlag, 2003. –1061 p.

15 Tarnoff D. Computer Organization and Design Fundamentals.– 2007.– 408 p. [Електронний ресурс]. – Режим доступу: http://www.cuc.ucc.ie/CS1101 /David%20Tarnoff.pdf

16 Stallings W. Computer Organization and Architecture, Design for Performance - New Jersey : Pearson Education, Inc., 2016. – 864 p. [Електронний ресурс]. – Режим доступу: http://home.ustc.edu.cn/~louwenqi/reference_books_ tools/Computer%20Organization%20and%20Architecture%2010th%20-20William% 20Stallings.pdf

17 Chalk B.S. Computer Organization and Architecture: an introduction. - Basingstoke : Palgrave Macmillan, 2003.– 350 p.

Навчальне видання

ГАВРИЛЕНКО Світлана Юріївна
ХАЦЬКО Наталія Євгенівна

ОСНОВИ АРХІТЕКТУРИ КОМП'ЮТЕРНИХ СИСТЕМ

Навчально-методичний посібник з курсів «Основи комп'ютерної інженерії», «Основи архітектури програмних систем», «Комп'ютерна математика» для студентів спеціальностей «Інженерія програмного забезпечення» та «Комп'ютерна інженерія» денної та дистанційної форм навчання

Англійською мовою

Відповідальний за випуск Годлевський М.Д.
Роботу до видання рекомендував Горілий О.В.

В авторській редакції

План 2019 р., поз. 92

Підп. до друку 08.11.19. Формат 60×84 1/16. Папір офсетний.
Riso-друк. Гарнітура Times New Roman. Ум. друк. арк.4.7
Наклад 50 прим. Зам. № 45/6 Ціна договірна.

Видавець Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5473 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Надруковано у Видавництві "Курсор"
м. Харків, вул. Дмитрівська, 5
т. (057) 706-31-73
Свідоцтво про внесення до Державного реєстру суб'єктів видавничої справи
серія № 21 від 24.03.2000 р.