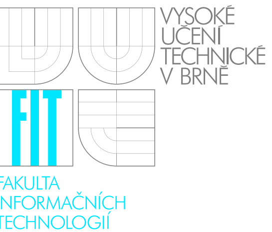


Computer Hardware

IPR

Study support

Doc. Ing. Jiří Kunovský, CSc. **November 19, 2008**

This learning text was supported by the project "Increase in competitive strength of IT specialists – alumni for the European employment market", reg. č. CZ .04.1.03/3.2.15.1/0003. This project is co-financed by the European social fund and by a state budget of Czech Republic.

Chapter 1

Introduction

There are in a lot of ways how to understand elements of the computer, how to design and analyse them and how to correctly follow their functions.

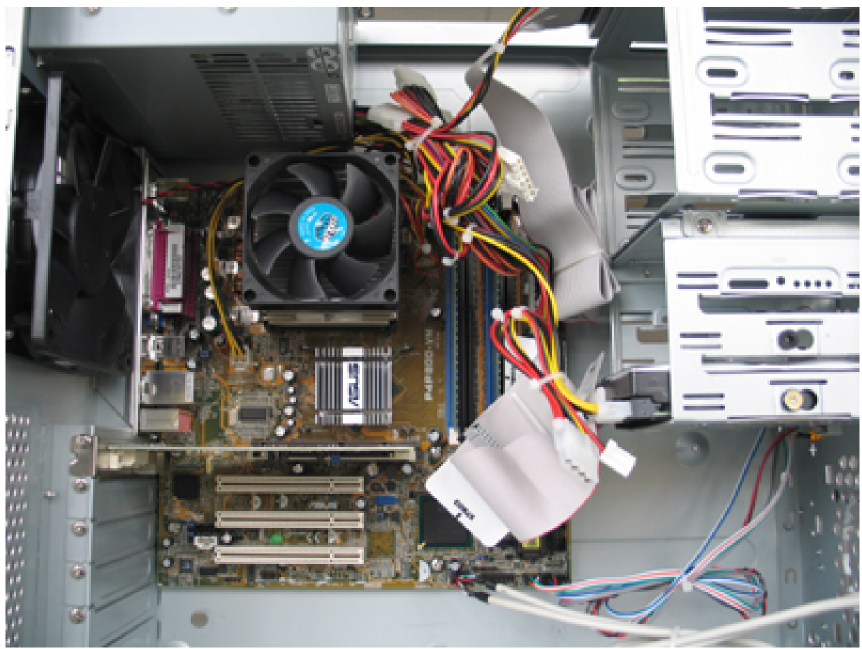


Figure 1.1: Computer inside

The study support uses the common method for a progressive interpretation of the function of computer elements (from basic circuits with resistors, diodes, transistors to combinational logic circuits or processors) and it is extended by concrete mathematical calculation examples.

At first we will talk about block and circuit solutions according to required operations of mathematical calculation. Later the individual solutions as particular elements of computer will be specified.

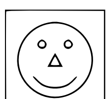
This text is characterised by a large visualisation (even if it does not describe very specific details). The emphasis lays on function principles, from which we can elicit even very complicated circuit solutions of computer elements according to initial requirements.

Some parts are marked by pictographs in the text. Pictographs show to the user the importance of each fraction of the text or other important information.

	Necessary time for study		Question, task to solve
	Target		Computer practice, lab example
DEF	Definition	x+y	Example
	Important part		Reference
	Extension information		Right result
1	Difficult part	Σ	Summary
2			Tutor's commentary
3			Interesting part

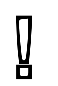
Table 1.1: Meaning of used pictographs

1.1 Organising information



The study support acts as the learning text for the course called Computer Hardware and it is the source of minimum knowledge for successful graduation. The text should be a help and that means it tries to fill up other study lectures and it does not replace the mimeographed or lecture notes. The subject consists of lectures and guided computer labs. An attendance at lectures is not compulsory. However experiences conclusively show that absence at lectures belongs to the most frequent failure in passing the mid-term exam and the final exam. The detailed information about the Computer Hardware with updates in every school year are situated on the website `http://www.fit.vutbr.cz/study/courses/IPR/private`, where the access for each student is granted.

1.2 Final exam and classification of the course

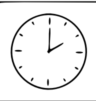


It is possible to obtain up to 25 points for specified activities in computer labs. Another 15 points students can obtain by writing the mid-term exam. The final written exam gives maximum 60 points.

Lecturers and supervisors of computer labs may appreciate extraordinary student's activities by extra points, but it is only a privilege.

The classification of successful students is managed by educational specifications of FIT VUT and principles of ECTS (European credit system). For successfully passing the course a student needs to get at least 50 points from the maximum of 100 points.

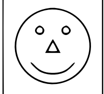
1.3 Estimation of time intensity



The approximate estimation of time intensity of the course (6 credits) is defined subsequently: 1 credit = 25 − 30 hours of work ⇒ 6 credits matches 150 − 180 hours of student's work, thereof:

- Lectures 26 hours.
- Exercises 13 hours.
- Computer labs 13 hours.
- Continuous study including preparation of examples for exercises about 70 hours.
- Preparation for the mid-term exam and the final exam about 40 hours.

1.4 Methodical information



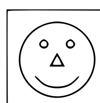
The Computer Hardware course is a compulsory subject in the second term of Bachelor degree, Programme Information Technology. This course is the introduction for wide and interesting problems which students could use to obtained knowledge in other subjects.

The knowledge of basic methods for solving non-linear circuits is a necessary presumption for a good work of every future specialist in the IT sphere.

We speculate about new and more arranged edition of study lectures, which would respect last changes in the course content coming from the three-year Bachelor study programme.

For readers of this study support we recommend to devote the time to think about introduced examples and also to solve all tasks in the important text parts.

1.5 Words of art

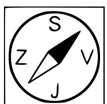


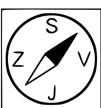
The rigorous thinking is the only presumption of each Msc. thesis and it has to be based on exact terms and right terminology. Technical dialect could be a common tool for every day communication. However it does not fit for publication actions or for advanced presentation activities. Therefore the text tries to present new terms in a precise and terminologically correct way.

1.6 Graphic layout

The text is written in the standard text type Times New Roman, 12 pt. Significant terms or text parts are written bold and/or underlined. Supplementary notes are represented by smaller type Times New Roman, 10 pt.

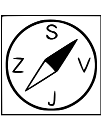
1.7 Learning outcomes and competences - specific





Students obtain an ability to analyse functions and to design non-linear electric circuits.

1.8 Learning outcomes and competences - generic



Students obtain the basics of selected methods for the description and solving non-linear electrical circuits, which belongs to elementary knowledge of each specialist in IT sphere.

1.9 Annotation

Analysis of transitional processes in electric circuits in a time area. TKSL simulation language. Formulation of circuit equations and possibilities of their solutions. Analysis of RC, RL and RLC circuits. Analysis of non-linear electric circuits. Parameters and characteristics of semiconductor elements. Graphic, numerical and analytical methods of non-linear circuit analysis. TTL and CMOS gates. Power supply units. Limiters and sampling circuits. Level translators, stabilizers. Astable, monostable and bistable flip-flops. Dissipationless and dissipation lines. Wave propagation on lines, reflections, adjusted lines.

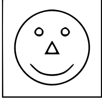
1.10 Syllabus of lectures

1. Analysis of transitional processes in electric circuits in a time area.
2. TKSL simulation language.
3. Formulation of circuit equations and possibilities of their solutions.
4. Analysis of RC, RL and RLC circuits.
5. Analysis of non-linear electric circuits.
6. Parameters and characteristics of semiconductor elements.
7. Graphic, numerical and analytical methods of non-linear circuit analysis.
8. TTL and CMOS gates.
9. Power supply units. Limiters and sampling circuits. Level translators, stabilizers.
10. Level translators, stabilizers.
11. Astable, monostable and bistable flip-flops.
12. Dissipationless and dissipation lines.
13. Wave propagation on lines, reflections, adjusted lines.

1.11 Study literature

1. Murina, M.: Teorie obvodu. Brno, VUTIUM 2000.

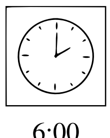
1.12 Note



The text of this brochure was made in a very short time and under time pressure. Thereby it contains some faults, mistakes or typing errors. The author will be very grateful for any tips and the author asks readers to send them to `kunovsky@fit.vutbr.cz`.

Chapter 2

Computer elements and numerical integration



6:00

2.1 Didactic intention

The aim is to introduce the function of computer elements as registers, adders and elementary microprocessors.

2.2 Numerical solution of differential equations

The numerical integration is a very clear and concrete mathematical example of calculation.

Let say that the numerical integration, or more precisely numerical calculation, of the differential equation $y' = f(t, y)$ with the initial value $y(0) = y_0$ means for us to establish values of required solution in points $y(t_1) = y_1$, $y(t_2) = y_2$, $y(t_3) = y_3, \cdots$, (see fig. 2.1).

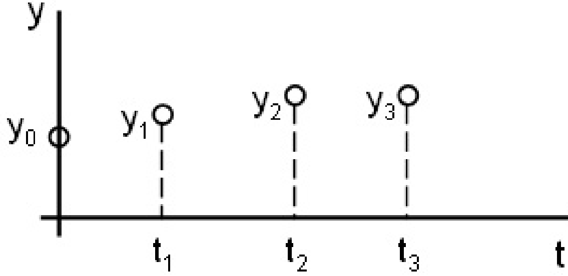


Figure 2.1: Graphical plotting of differential equation solution

Let's have a look at the simple example to clarify the definition.

a) Simple differential equation

$$y' = y \qquad y0 = y_0$$

b) Simple system of differential equations

$$\begin{aligned} y' &= z \qquad & y(0) = y_0 \\ z' &= -y \qquad & z(0) = z_0 \end{aligned}$$

One-step and multi-step methods are usually used for numerical calculation of differential equations.

One-step integration methods are suitable for calculations, where the method uses in the step $i + 1$ in time t_{i+1} the value from the previous step i in time t_i. These types of integration methods compute with the initial value $y[t = 0] = y(0) = y_0$ as a starting value on the beginning of computation.

Multi-step integration methods compute the step $i+1$ in time t_{i+1} using several values obtained from previous steps $i,\ i-1, i-2, \cdots$ in time $t_i,\ t_{i-1},\ t_{i-2}, \cdots$ We call them non-self starter methods.

2.2.1 Taylor series

We consider the Taylor series of the function y in the point i as a essential one-step integration method. The formula for the new value in the step $(i+1)$ is given by

$$y_{i+1} = y_i + hy'_i + \frac{h^2}{2!}\, y_i^{(2)} + \cdots + \frac{h^n}{n!} y_i^{(n)} \tag{2.1}$$

There has been published plenty of numerical methods for the solution of differential equations. Methods are distinguished by an accuracy and a speed of computing.

2.2.2 Euler method

Euler method is the simplest one-step method. It computes with only first two terms of Taylor series. The computing formula is defined by:

$$y_{i+1} = y_i + hy'_i \tag{2.2}$$

If the value h is small enough, the method is sufficiently accurate. The main advantage is that the method needs to know just one previous derivative.

2.2.3 Solution of simple differential equation

We apply the Euler method to the simple differential equation:

$$y' = y \qquad y(0) = y_0 \tag{2.3}$$

Using the formula 2.2 the equation is now given by

$$y_{i+1} = y_i + hy'_i$$

then replace by 2.3, we find

$$y_{i+1} = y_i + hy_i \tag{2.4}$$

The first step of computation is given by

$$y_1 = y_0 + hy_0$$

and next steps follow

$$\begin{aligned} y_2 &= y_1 + hy_1 \\ y_3 &= y_2 + hy_2 \\ &\vdots \end{aligned}$$

The equation 2.4 is the essential equation for the solution using Euler method. From the equation 2.4 is derived the request for the single-purpose arithmetic unit structure, also known as an elementary integration processor. Mathematical operations like addition, subtraction and multiplication are required for calculations.

Analytical solution of the differential equation is equal to

$$y = y_0 e^t \tag{2.5}$$

numerically		**analytically**	
step size	**y**	**t**	**y**
0	1,00000	0	1,00000
1	1,10000	0,1	1,10517
2	1,21000	0,2	1,22140
3	1,33100	0,3	1,34986
4	1,46410	0,4	1,49183
5	1,61051	0,5	1,64872
⋮	⋮	⋮	⋮
19	6,11591	1,9	6,68589
20	6,72750	2,0	7,38906

Table 2.1: Comparison of numerical and analytical solution

Now we calculate resultant values from equations 2.4 and 2.5 to obtain the comparison of numerical and analytical solution. We obtain numerical solution from the equation 2.4 and analytical solution from 2.5. We put up these resultant values into the table 2.1 for clarity.

The initial value $y_0 = 1$ and the step size $h = 0,1$ have been chosen.

Both results are plotted in the graph.

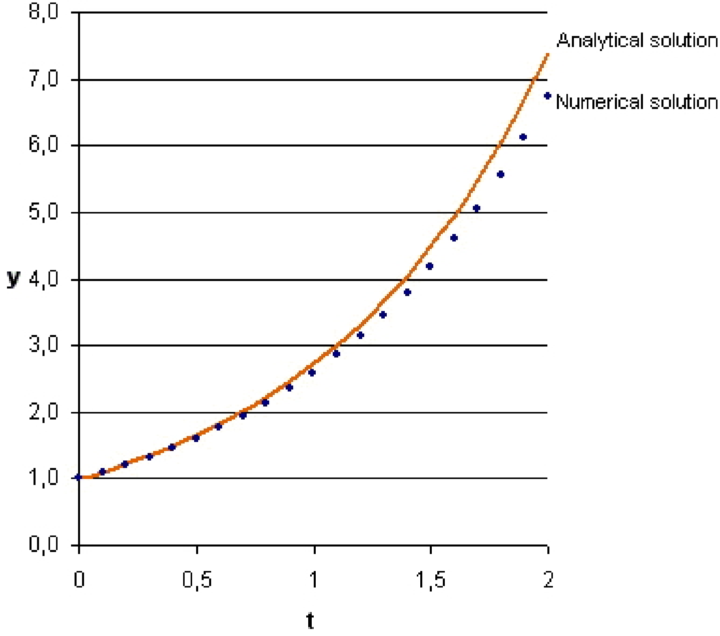


Figure 2.2: Comparison of numerical and analytical solution

2.2.4 Solution of differential equation order 2

We present the example of second order differential equation

$$y'' = -y \qquad y'(0) = y_{p1} \qquad y(0) = y_0 \tag{2.6}$$

Before solving the equation it is necessary to transform the second order differential equation to the equivalent system of first order differential equations. We substitute the equation 2.6 by $y' = z$ and we have

$$z' = -y \qquad z(0) = y_{p1} \tag{2.7}$$

$$y' = z \qquad y(0) = y_0 \tag{2.8}$$

We apply the Euler method to the system of differential equations

$$z_{i+1} = y_i + hy_i' \qquad z_{i+1} = z_i + hz_i'$$

and after substitution we obtain the essential equation for solving

$$y_{i+1} = y_i + hz_i \qquad z_{i+1} = z_i + h(-y_i) \tag{2.9}$$

First step of calculation is

$$y_1 = y_0 + hz_0 \qquad z_1 = z_0 + h(-y_0)$$

followed by next steps, which are

$$\begin{aligned} y_2 &= y_1 + hz_1 & z_2 &= z_1 + h(-y_1) \\ y_3 &= y_2 + hz_2 & z_3 &= z_2 + h(-y_2) \\ &\vdots & &\vdots \end{aligned}$$

Equations in this position are disposed for the parallel design of calculation. It is very important to know that all mathematical operations executed in parallel in both microprocessors are identical. Hence, microprocessors can be controlled by one unique unit.

We compare results with analytical solutions, which are

$$y = sin(t) \qquad z = cos(t) \tag{2.10}$$

For solving equations we set initial values $y_0 = 0,\ z_0 = 1$ and the step size $h = 0,1$. Resultant values from the equation 2.9 (numerical solution) and from the equation 2.10 (analytical solution) are put up in the table 2.2.

step size	y		z	
	numerically	**analytically**	**numerically**	**analytically**
0	0,00000	0,00000	1,00000	1,00000
1	0,10000	0,09983	1,00000	0,99500
2	0,20000	0,19867	0,99000	0,98007
3	0,29900	0,29552	0,97000	0,95534
4	0,39600	0,38942	0,94010	0,92106
5	0,49001	0,47943	0,90050	0,87758
6	0,58006	0,56464	0,85150	0,82534
7	0,66521	0,64422	0,79350	0,76484
⋮	⋮	⋮	⋮	⋮
81	1,46049	0,96989	-0,32534	-0,24355
82	1,42796	0,94073	-0,47138	-0,33916
83	1,38082	0,90217	-0,61418	-0,43138

Table 2.2: Comparison of numerical and analytical solution

Results are plotted in figures 2.3 and 2.4 for easy comparison. Solution of the equation $y = f(t), y(0) = 0$ is plotted in fig. 2.3 and solution of the equation $z = z(t), z(0) = 1$ is shown in fig. 2.4.

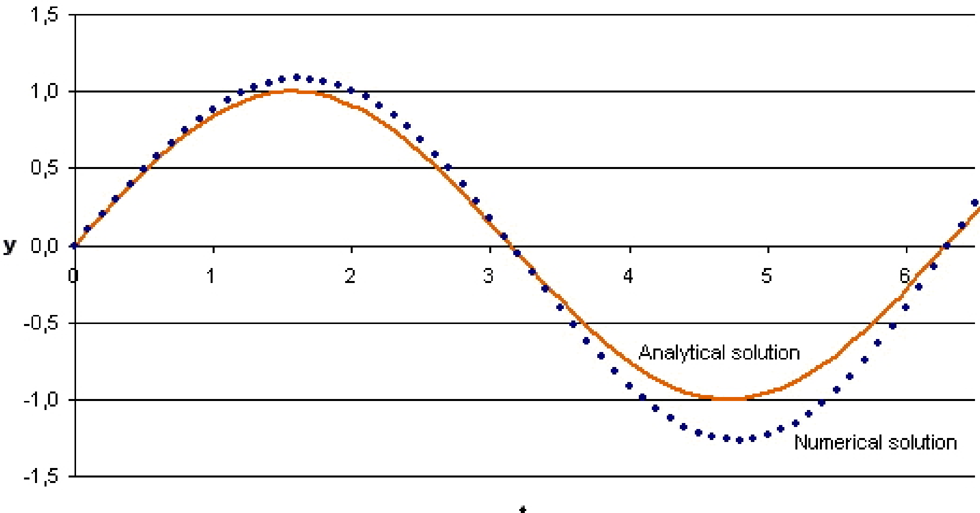


Figure 2.3: Solution of variable y

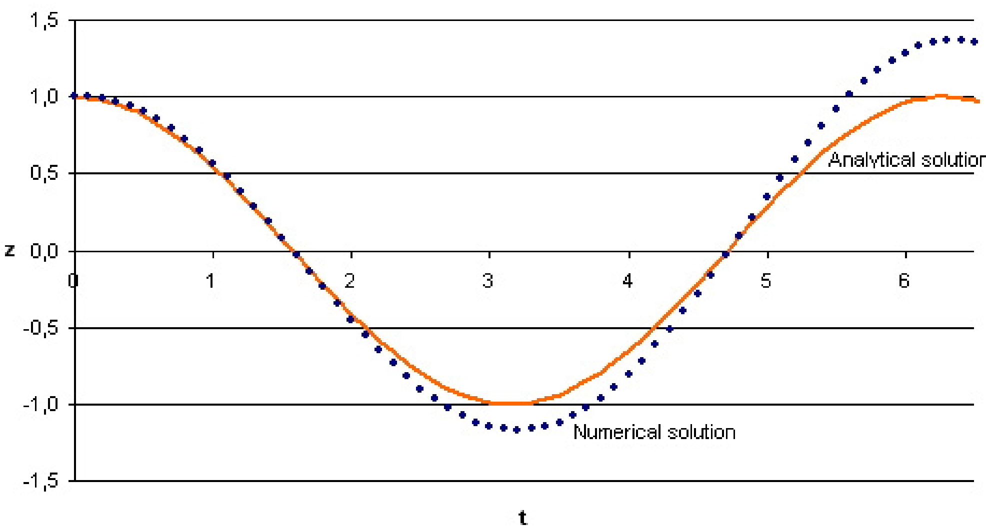


Figure 2.4: Solution of variable z

2.2.5 Conception of microprocessor

The design of microprocessor as a single-purpose arithmetic unit, which executes the numerical integration, is influenced by needful mathematical operations for numerical calculation of differential equations. From the view of microprocessor design, the simplest processing occurs, when the machine computes the homogeneous first order differential equation, e.g. example 1, especially the equation 2.4.

Solution of the homogeneous differential equation computed by Euler method gives individual desired operations of microprocessor:

- addition - realised by the combinatory adder,
- subtraction - is transferred into addition by changing subtrahend sign,
- multiplication - executed by serial-parallel method of partial totals and shifts based on Booth's algorithm.

Designed microprocessor computes the system of homogeneous first order differential equations with constant coefficients. Corresponded numerical integration is realized in the binary representation, in complement code in the fixed decimal point.

Theory of binary representation is the content of following chapter.

2.3 Binary representation

The concrete mathematical computation has been shown in the previous chapter. It is obvious that mathematical computation are realised in the binary code.

This chapter introduces conversions between decimal and binary number systems. The technique of data storage in the computer is shown, as well as the coding and organisation of individual bits.

2.3.1 Decimal and binary number systems conversion

The decimal number is divided by two and the remainder as a least-significant bit is stored. Then the (integer) result is divided by two repeatedly and remainders are remembered. This process repeats until the result of further division becomes a zero. The first digit of the number in the binary system is the remainder acquired from the last division.

Example: Conversion of integer number

division result	remainder
155 : 2 = 77	1
77 : 2 = 38	1
38 : 2 = 19	0
19 : 2 = 9	1
9 : 2 = 4	1
4 : 2 = 2	0
2 : 2 = 1	0
1 : 2 = 0	1

$$(155)_{10} = (10011011)_2$$

The binary number 10011011 could be written in the form:

$$1 \cdot 2^7 \quad + 0 \cdot 2^6 \quad + 0 \cdot 2^5 \quad + 1 \cdot 2^4 \quad + 1 \cdot 2^3 \quad + 0 \cdot 2^2 \quad + 1 \cdot 2^1 \quad + 1 \cdot 2^0 \quad =$$
$$1 \cdot 128 \quad + 0 \cdot 64 \quad + 0 \cdot 32 \quad + 1 \cdot 16 \quad + 1 \cdot 8 \quad + 0 \cdot 4 \quad + 1 \cdot 2 \quad + 1 \cdot 1 \quad = 155$$

Conversion of decimal number

To convert a number between 0 and 1 we must separate integer and decimal parts. The decimal fraction is repeatedly multiply by two and the integer fraction is stored. At each step, keep track of the integer part of the result but do not carry it along in subsequent multiplications. The first digit of number in the binary system is the first stored integer number after the multiplication.

0	625 x 2
1	250 x 2
0	500 x 2
1	000 x 2
0	000 x 2
0	000 x 2
0	000 x 2
0	000

$$(0,625)_{10} = (0,1010000)_2$$

The binary number 0,1010000 could be written in the form:

1	0,5	2^{-1}
0	0,25	2^{-2}
1	0,125	2^{-3}
0	0,0625	2^{-4}
$\vdots$	$\vdots$	$\vdots$

$$= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} = 0,625$$

2.3.2 Number storage

Numbers are stored bit after bit in the one-bit storage element in the binary system.

bits in memory

$(0,75)_{10} = (0,1100000)_2$ => 0 1 1 0 0 0 0 0

SIGN

Each storage element has an input (for the binary value writing) and an output (for the binary value reading) - see figure 2.5.

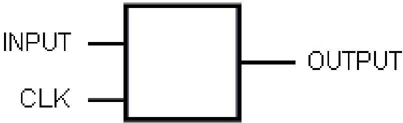


Figure 2.5: Storage element

The input value to the storage element is controlled by e.g. clock signal *CLK*.

The high frequency storage element is the D-type flip-flop (see fig. 2.6).

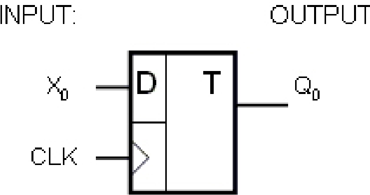


Figure 2.6: The symbol of D-type flip-flop

Important note: The input value (X_0) has been written into the storage element by the rising edge of the clock signal and it has been displayed on the output (Q_0). This process is shown in fig. 2.7.

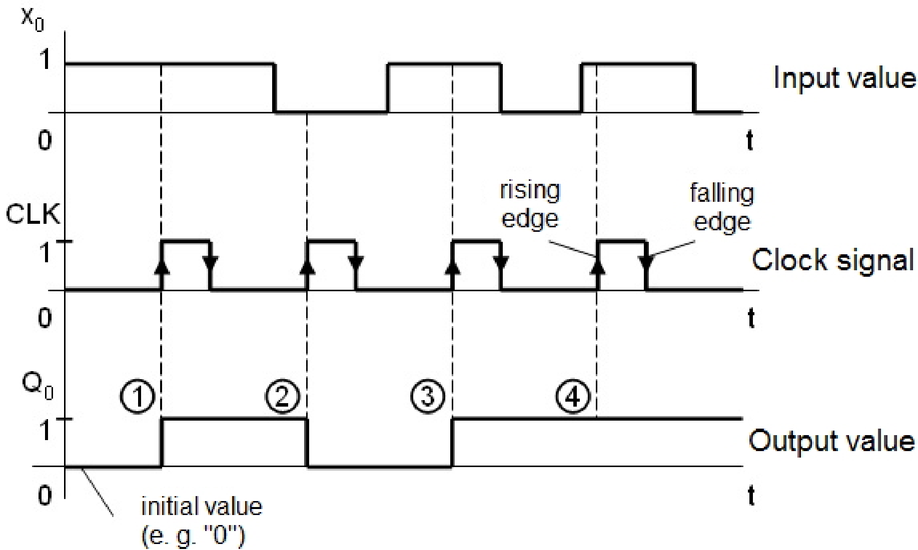


Figure 2.7: D flip-flop timing diagram

Explanatory text for fig. 2.7:

point **1** The arrival of the rising edge of the clock signal writes the input value to the flip-flop output. We say, the output is over set to the value $Q_0 = 1$. This output value Q_0 stays until the new rising edge of the clock signal comes.

point **2** The output value is set according to the input value again; that means the output is over set to the value $Q_0 = 0$. This state holds until the next clock signal comes.

point **3** The rising edge over sets the output to the value $Q_0 = 1$ according to the input.

point **4** Next clock signal does not change the output $Q_0 = 1$ (the input value is on the level 1, therefore the output stays on the same value 1). The input value has been changed on 0 for a moment between 3rd and 4th clock signal. But the output value is still 1, because for change the output value the rising edge of the clock signal is needed.

Register with parallel input and output

The register is built from the group of storage elements. The register is used for storage data (operands), which are used for mathematical calculations in the processor. It is a very fast storage unit.

The parallel register has to ensure:

- Parallel input data writing.
- Parallel output data reading.

The symbol of the register is in figure 2.8.

Detailed scheme of register is shown in figure 2.9. The register contains D-type flip-flop circuits. The rising edge of the clock signal *CLK* controls the writing of input data into the register.

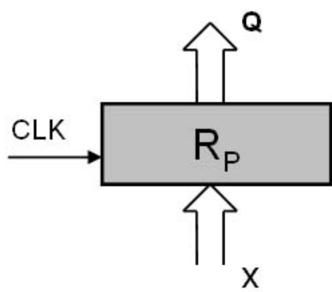


Figure 2.8: Symbol of parallel register

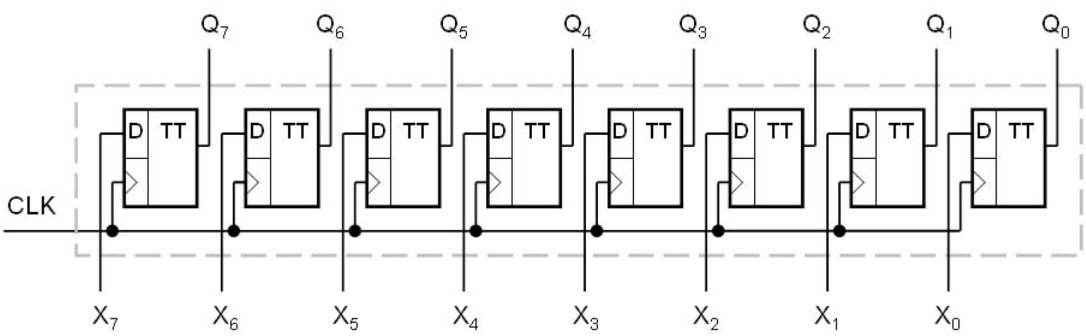


Figure 2.9: Internal connection of parallel register

Shift register

The shift register has to ensure:

- Parallel input for data writing.
- Serial input for data.
- Serial output for data.
- Right shift of data.

The symbol of the shift register SR is in figure 2.10.

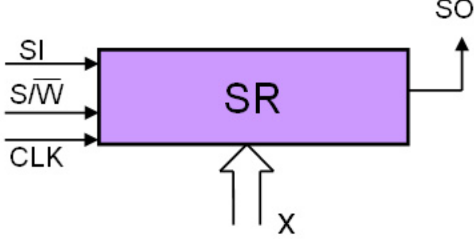


Figure 2.10: Shift register

Detailed scheme of internal connection of the shift register is shown in fig. 2.11. The function of the shift register after the clock signal arrival is described in the table of functions 2.3.

Input values are written into the shift register SR by the rising edge of the clock signal CLK and by logic level "0" on the input $S/\overline{W}$. The modification to logic level "1" on the input $S/\overline{W}$ causes the next clock signal CLK to shift the register content 1 bit to the right.

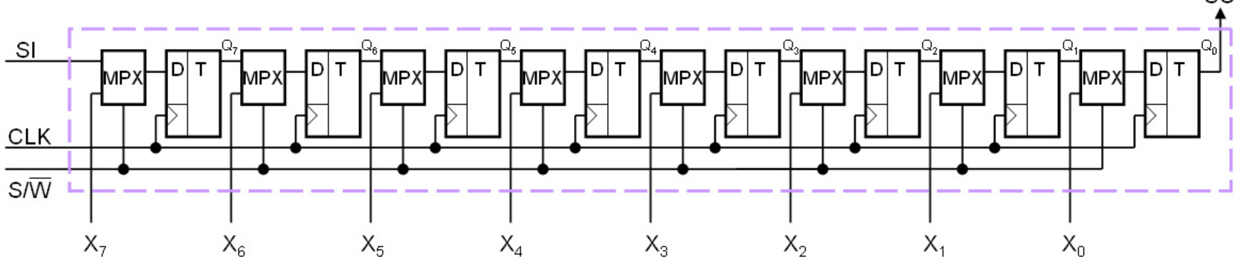


Figure 2.11: Internal connection of shift register

$S/\overline{W}$	Y_i
0	X_i
1	Q_{i-1}

Table 2.3: Function of the shift register

The circuit MPX is shown in fig. 2.12, it is a single-bit multiplexer, which switches the direct parallel input (writing) or the output from previous flip-flop circuit (shift).

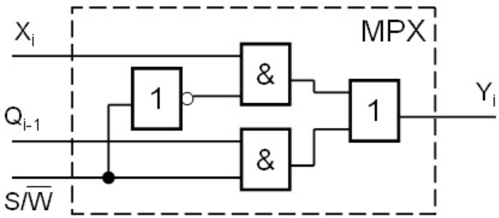


Figure 2.12: Circuit MPX - multiplexer

Arithmetic shift accumulator

Basic requirements of accumulator are:

- Parallel input data writing.
- Parallel output data reading.
- Arithmetic right shift.
- Insertion zeros.

The symbol of accumulator ACC is in fig. 2.13.

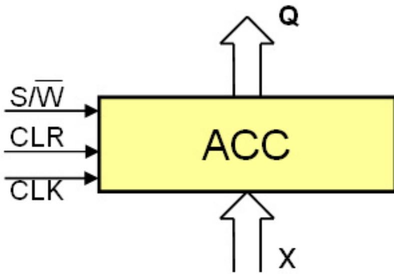


Figure 2.13: Accumulator

During the right arithmetic shift the bit "0" is not stored in the released position (as the logical shift), but the accumulator saves the bit, which has been on the position before shifting. This operation is called sign-propagation. The shift is important for calculation with numbers in codes with the sign in the most significant bit.

Detailed scheme of accumulator is in fig. 2.14.

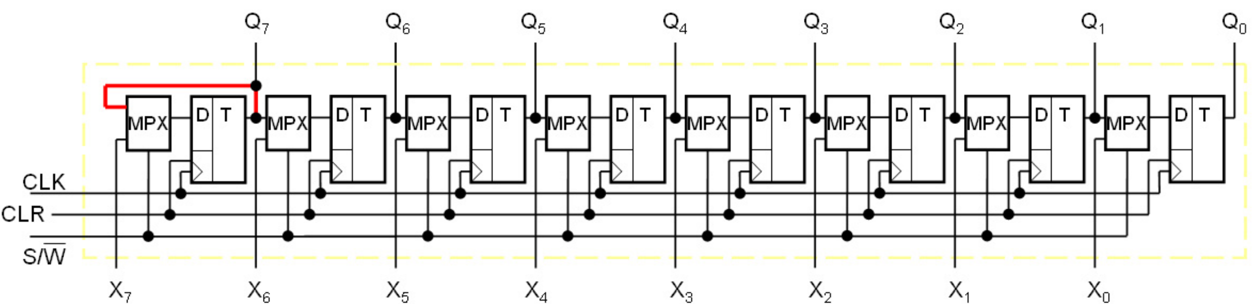


Figure 2.14: Internal connection of accumulator

The function of the accumulator is analogical to the shift register. The content of accumulator is reset by loading logic level "1" signal on the input CLR.

If the logic level "1" is on the input $S/\overline{W}$, the next clock signal CLK shifts the content of accumulator to the right. In the position of the most significant bit is written the previous logic level (before a clock signal CLK - sign propagation). If the logic level "0" on the input $S/\overline{W}$, the rising edge of the clock signal CLK loads input data into the accumulator.

2.4 Computational operations

In the chapter 2.2 operations like addition, subtraction, multiplication and division have been shown. This chapter describes how are these operations executed in the microprocessor. Fix-point numbers are used in ensuing text.

Used code is frequently represented by the complement code. It is possible to count up arbitrary positive and negative numbers. Potential carry from the sign bit is disregarded. The information about the sign is saved in the most significant bit – a sign bit.

2.4.1 Addition

See the simple example of addition two numbers $0,75 + 0,125$. In the binary system on 8 bits in complementary code: $(0,75)_{10} = (0,1100000)_2$, $(0,125)_{10} = (0,0010000)_2$. The result is $0,875$ in decimal system, i. e. $0,1110000$ in binary system.

	Decimal:		Binary:
x =	0,75		01100000
y =	+ 0,125	⇒	+ 00010000
z =	0,875		01110000

The process of addition works subsequently: operands are loaded to registers, which are connected with the input of adder (fig. 2.15) and operands are summed (fig. 2.16).

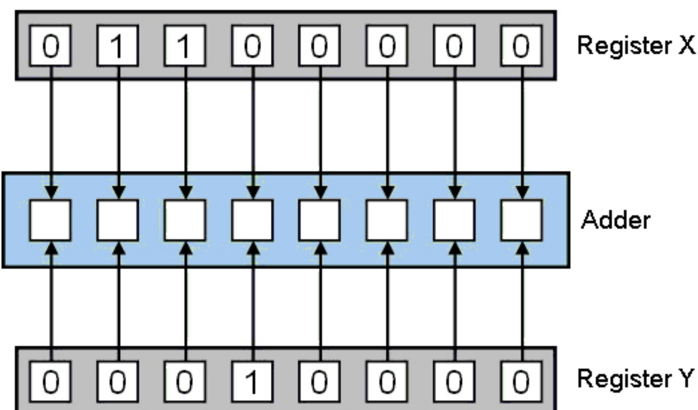


Figure 2.15: Operands on adder input

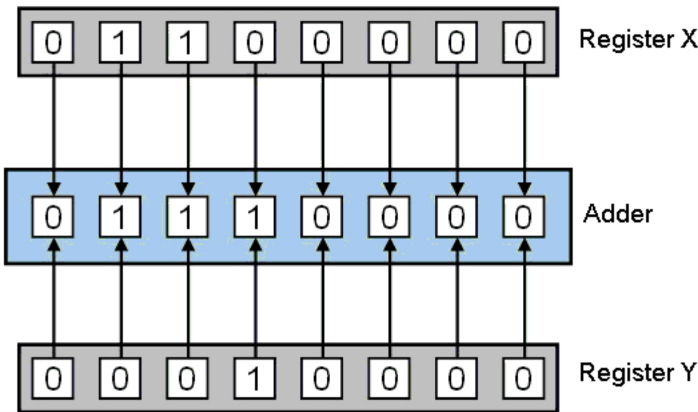


Figure 2.16: Operands sum

Previous example is the simplest possibility of computing. In other examples, the high order carry could happen (i.e. the adder computes two operands with a carry).

The high order carry could also happen in the case of addition of numbers $0,625$ and $0,125$. Binary numbers, the result are shown in fig. 2.17. The carry (logic value "1") is presented by the arrow and the symbol 1.

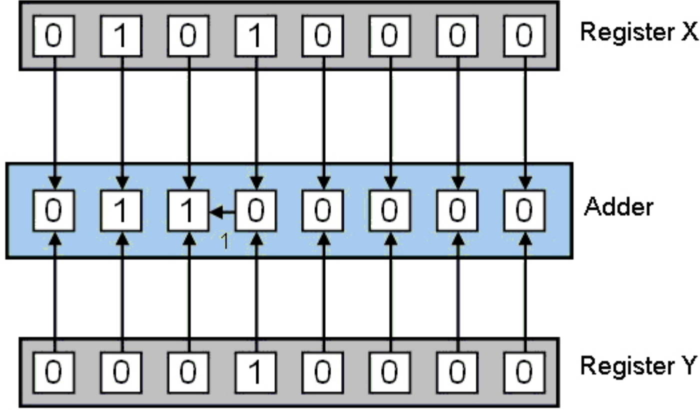


Figure 2.17: Addition with the carry

The connection diagram is shown in fig. 2.18. The operand X is stored in the register R_X, the operand Y is in the register R_Y.

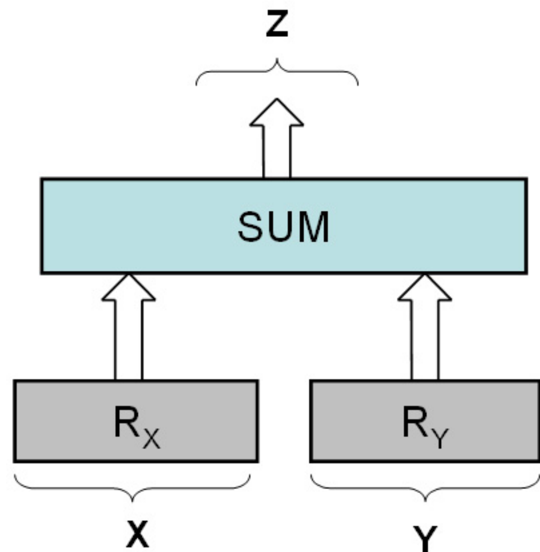


Figure 2.18: Principle of addition

The symbol of 8 bits combinational adder is in fig. 2.19.

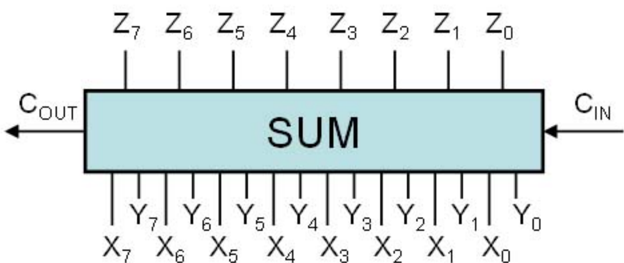


Figure 2.19: Combinational adder

Detailed scheme of the combinational adder is in fig. 2.20. The circuit ADD is a full one-bit adder. It is necessary to connect partial one-bit adders to each other (the output carry from one adder is the input for the second one, etc.).

It is important for a carry propagation, which could happen during the computation. Detailed scheme of one-bit full adder is in fig. 2.21.

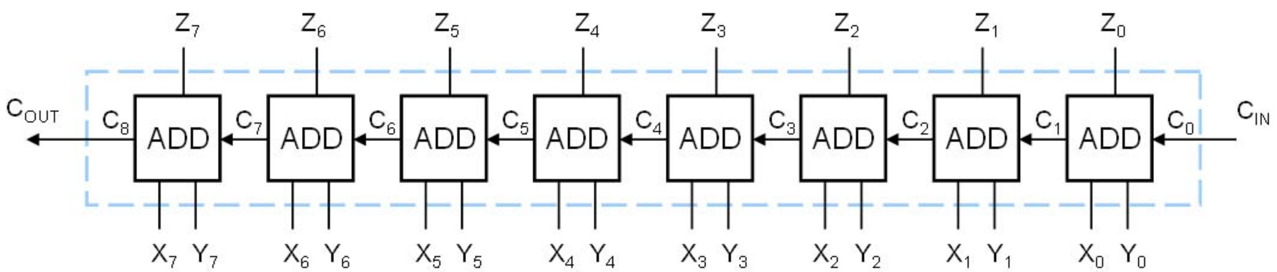


Figure 2.20: Combinational adder

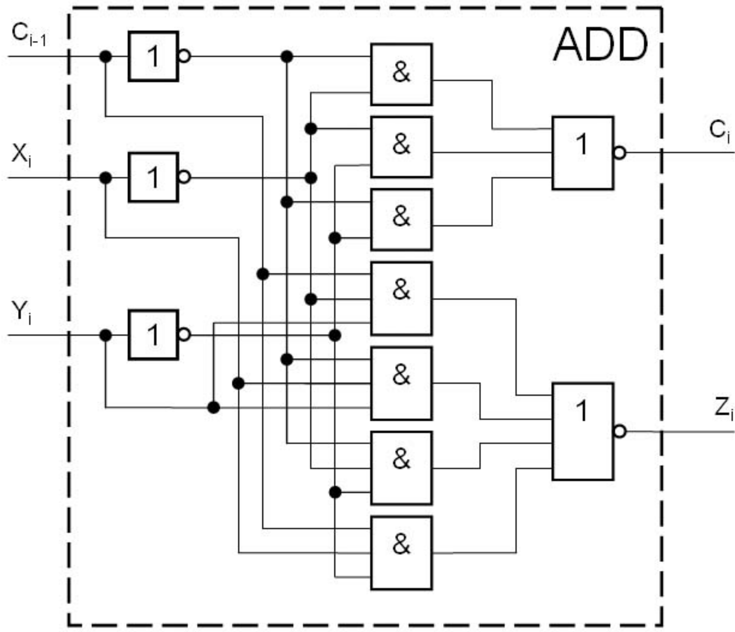


Figure 2.21: One-bit full adder

2.4.2 Subtraction

The operation of subtraction $(X - Y)$ is transferred to the addition with the subtrahend sign modification. The modification is realised by subtrahend conversion into the binary supplement and it follows described algorithm.

- Inverse all subtrahend orders (bits).
- Add the logic value "1" to the least significant bit.

The subtrahend is transferred to the supplementary system. Now the positive minuend and the negative subtrahend are added.

Let see the example with conversion into the binary supplement:

0,625	=	01010000	- convert into the binary system
		10101111	- invert of all bits
		1	- add 1
-0625	=	10110000	= negative result

Connection diagram of the example is in fig. 2.22.

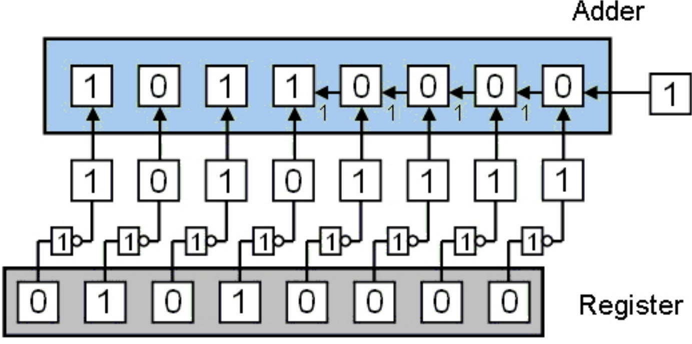


Figure 2.22: Conversion of negative number

Let's demonstrate the subtraction in the example $(0,625$ - $0,125)$. The number $(0,625)_{10}$ in the binary system is $(01010000)_2$ and the number $(0,125)_{10}$ is $(00010000)_2$. The number $(0,125)_{10}$ is transferred to the negative number in supplementary code and then added with the number $(0,625)_{10}$.

	Decimal:		Binary:
x =	0,625		01010000
y =	- 0,125	$\Rightarrow$	+ 11110000
z =	0,5		1|01000000

The technical realisation is more complicated.
We connect the first register $R\,Y$, where the subtrahend is stored, with inverters. Inverters negate the stored number and this new number is written into the input of the adder. On the second input $R\,X$ the minuend is stored. To complete the calculation in the complementary code, we have to add an extra value one to the least significant bit. It could be realised by third adder carry input C_{IN} with the value "1". The function of the technical realisation is shown in fig. 2.23. The completed result of the subtraction is shown in fig. 2.24.

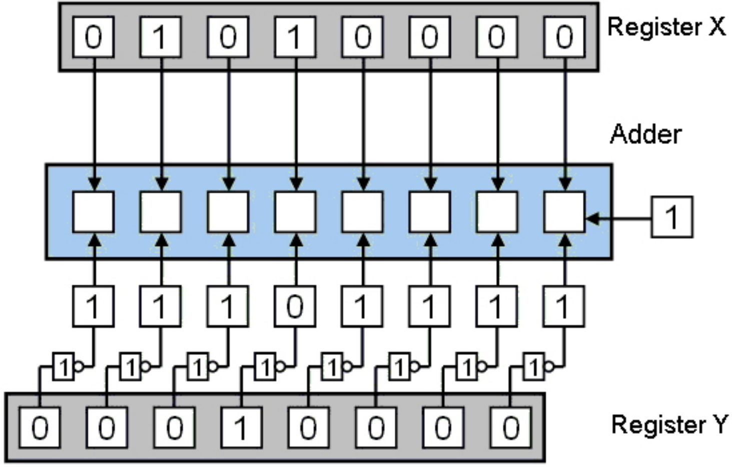


Figure 2.23: Operands writing

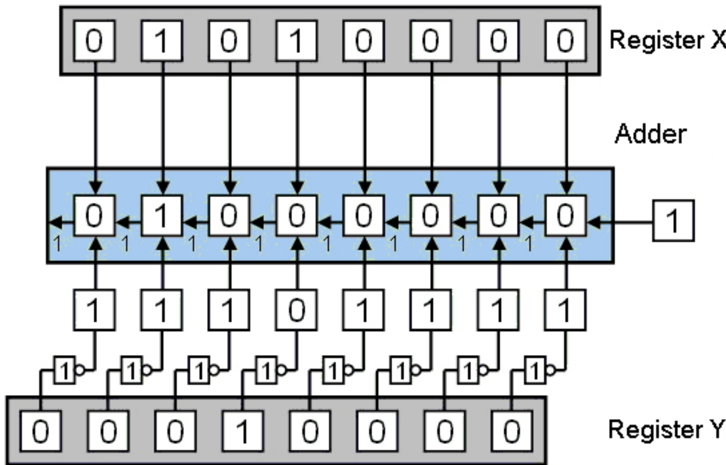


Figure 2.24: Completed computation

The most significant bit (sign bit) overflows during the computation. But this carry could easily be ignored. The result displayed on the given number of bits is correct.

The operation of addition or subtraction could be managed using a controlled negation.

Unit of controlled negation

The unit of controlled negation realises two functions:

- to carry input data X (on the level NEG="0"),
- to negate input data X (control signal NEG="1").

The symbol of the unit BNEG is shown in fig. 2.25.

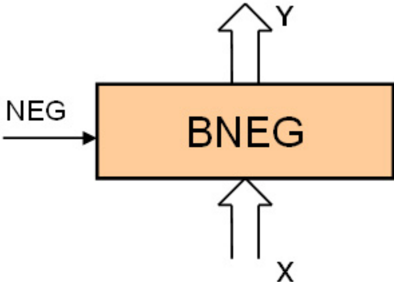


Figure 2.25: Unit of controlled negation BNEG

The function of the circuit is described in the table 2.4.

NEG	Y
0	X
1	$\overline{X}$

Table 2.4: Function of BNEG

The unit of controlled negation could be realized in two ways.

- Using non-equivalence terms (exclusive OR), see fig. 2.26.

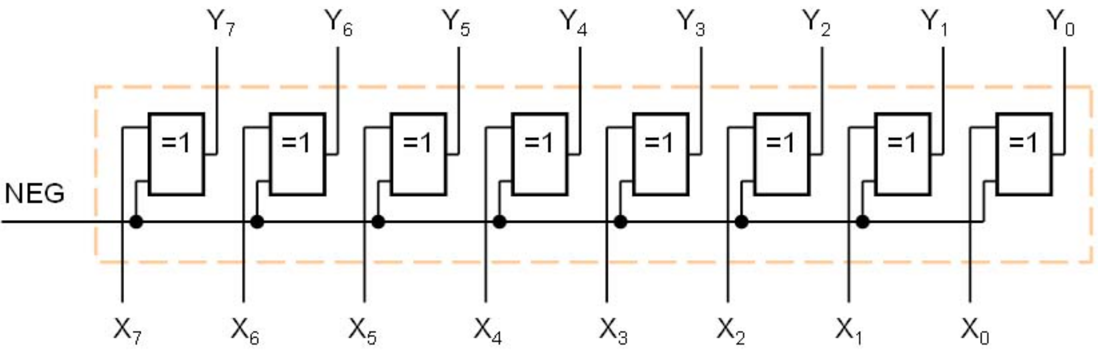


Figure 2.26: Realisation of BNEG using exclusive OR

The equation for values on the output Y_i:

$$Y_i = NEG \otimes X_i = \overline{NEG} \wedge X_i \vee NEG \wedge \overline{X_i}$$

It could be written in the truth table:

X_i	NEG	Y_i
0	0	1
1	1	1
0	1	1
1	1	0

- Using multiplexer (see fig. 2.28), which switches the direct and the inverse output from the attached register, see fig. 2.27.

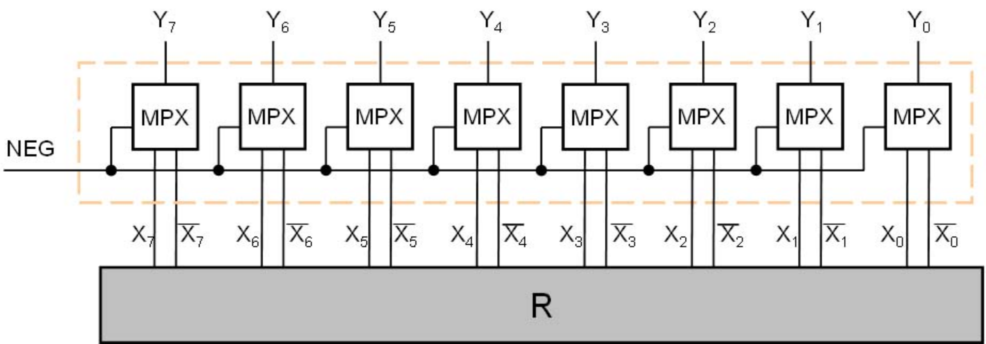


Figure 2.27: Realisation of BNEG using multiplexer

It is possible to work with the second realization, when the register with the direct (X_i) and also the inverse output ($\overline{X_i}$) is used in the connection. We connect the double output to the one-bit multiplexer (fig. 2.28) and the multiplexer switches the direct and the inverse output from the register using the controlled signal *NEG*. The function of the multiplexer is described in the following truth table.

NEG	Y_i
0	X_i
1	$\overline{X_i}$

Figure 2.28: Multiplexer

The subtracter integration, which uses the unit BNEG, is shown in figure 2.29.

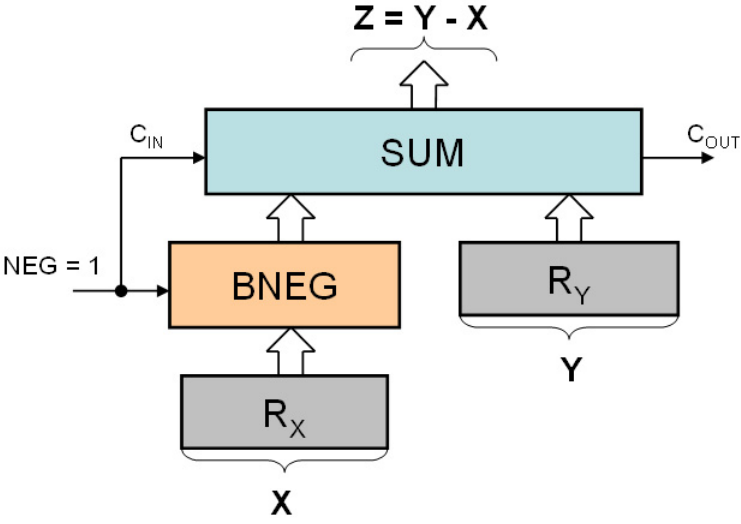


Figure 2.29: Subtraction principle

In a perspective view of the proposed processor the subtracter is used also as an adder. The function is specified by the control signal *NEG*. If the signal *NEG* is at the logic level 1, that means the unit is used as a subtracter. If the *NEG* is a the logic 0, the unit is an adder.

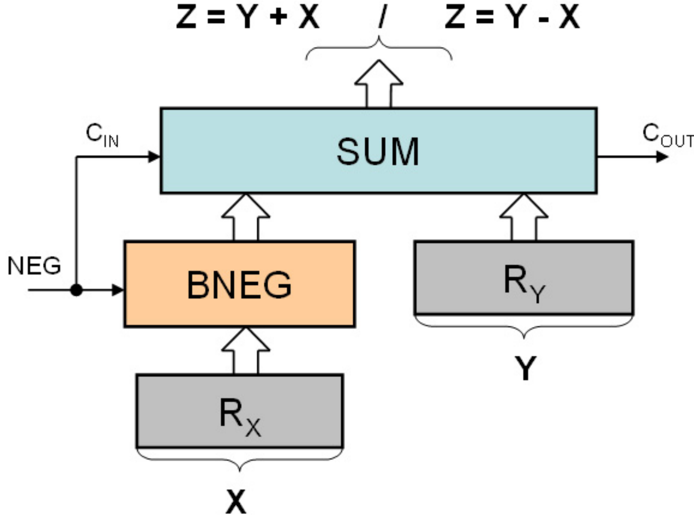


Figure 2.30: Function of adder-subtracter is given by the signal *NEG*

2.4.3 Multiplication

In this chapter, more examples of multiplication will be presented. The multiplication is more complicated operation then previous operations and we can realize it in many ways. The elementary computational method will be demonstrated in signed magnitude representation and the Booth's multiplication algorithm will be presented.

The following table shows elementary conversion to decimal numbers (weights for the transformation from binary representation to decimal representation):

2^{-1}	0,5
2^{-2}	0,25
2^{-3}	0,125
2^{-4}	0,0625
2^{-5}	0,03125
2^{-6}	0,015625
2^{-7}	0,0078125
2^{-8}	0,00390625

Multiplication in unsigned magnitude binary representation

The classical multiplication principle is based on sequential addition and multiplication. The principle is demonstrated in following examples:

- **0,625 * 0,5 = 0,3125:**

```
      0, 1 0 1 0 ≡ 0,625
    * 0, 1 0 0 0 ≡ 0,5
    -------------
         0 0 0 0 0
       0 0 0 0 0
     0 0 0 0 0                Result conversion:
   0 1 0 1 0                  0,25      2^-2
 0 0 0 0 0                    0,0625    2^-4
 -----------------            ------
 0, 0 1 0 1 0 0 0 0 ≡ 0,3125  0,3125
```

The result is now stored in the double number of bits.

- **0,625 * 0,75 = 0,46875:**

```
      0, 1 0 1 0 ≡ 0,625
    * 0, 1 1 0 0 ≡ 0,75       Result conversion:
    -------------
           0 0 0 0            0,25      2^-2
         0 0 0 0              0,125     2^-3
       1 0 1 0                0,0625    2^-4
     1 0 1 0                  0,3125    2^-5
 -----------------            -------
 0, 0 1 1 1 1 0 0 0 ≡ 0,46875 0,46875
```

The most significant bit is not needed in this example (the bit has any important information). This bit provides information about the location of the decimal separator.

- **0,9375 * 0,9375 = 0,87890625:**

Calculation	Result conversion:	
0, 1 1 1 1 ≡ 0,625		
* 0, 1 1 1 1 ≡ 0,75		
1 1 1 1		
1 1 1 1		
(1 0 1 1 0 1)	0,5	2^{-1}
1 1 1 1	0,25	2^{-2}
(1 1 0 1 0 0 1)	0,125	2^{-3}
1 1 1 1	0,00390625	2^{-8}
0, 1 1 1 0 0 0 0 1 ≡ 0,87890625	0,87890625	

If there is a problem with ones, it is easier to calculate the example step-by-step and to note partial results (rows in brackets). The partial result is good for computation clarity and to avoid the carry calculation.

If we want to multiply negative numbers as well, absolute values are needed for the calculation. After the computation we compute the sign of both numbers, for the multiplication of one positive and one negative number, the result is a negative number. If we multiply two positive, or two negative numbers, the result is always a positive number.

Booth's multiplication algorithm

The algorithm is based on repeatedly partial addition (subtraction) and partial shifts.

The function of algorithm executes the multiplier value using the two predetermined logic values of bits (a current one and a previous one) according to rules described in the table 2.5.

multiplier logic value		
b_i bit	b_{i-1} bit	action
0	0	fill remainder with zeros
0	1	add the multiplicand to the remainder
1	0	subtract the multiplicand from the remainder
1	1	fill remainder with zeros

Table 2.5: Principle of Booth's multiplication algorithm

We need the positive and also negative value of the multiplicand during the computation. Current value of multiplier bits decides the specific action in the step.

Notice the last multiplier bit, is compared with the value called "small zero" in the first step.

The algorithm principle is demonstrated in examples, where numbers $0,625$ and $0,5$ are multiplied. We describe four possibilities of computation, when number signs are different in each example. The algorithm will show us always the same correct result for any sign combination.

The truncation error is also shown in following examples. The error could arise, when the same number of bits for the result storage as operands number of bits is used.

- **0,625 * 0,5 = 0,3125:**

Multiplier: 0,625 = 0,1010
Multiplicand: 0,5 = 0,1000 – plus multiplicand
-0,5 = 1,1000 – minus multiplicand

$$
\begin{array}{rl l}
 & 0,\ 1\ 0\ 1\ 0 \equiv 0,\ 1\ 0\ 1\ 0 & \\
 * & 0,\ 1\ 0\ 0\ 0 & \\
\hline
\text{CLR} & 0\ \ 0\ 0\ 0\ 0 & \text{reset} \\
+ & 0\ \ 0\ 0\ 0\ 0 & 0\ 0\ \text{(plus zero)} \\
\sum & 0\ \ 0\ 0\ 0\ 0 & \\
\hline
\rightarrow & 0\ \ 0\ 0\ 0\ 0 & \\
+ & 1\ \ 1\ 0\ 0\ 0 & 1\ 0\ \text{(minus multiplicand)} \\
\sum & 1\ \ 1\ 0\ 0\ 0 & \\
\hline
\rightarrow & 1\ \ 1\ 1\ 0\ 0 & \\
+ & 0\ \ 1\ 0\ 0\ 0 & 0\ 1\ \text{(plus multiplicand)} \\
\sum & 0\ \ 0\ 1\ 0\ 0 & \\
\hline
\rightarrow & 0\ \ 0\ 0\ 1\ 0 & \\
+ & 1\ \ 1\ 0\ 0\ 0 & 1\ 0\ \text{(minus multiplicand)} \\
\sum & 1\ \ 1\ 0\ 1\ 0 & \\
\hline
\rightarrow & 1\ \ 1\ 1\ 0\ 1 & \\
+ & 0\ \ 1\ 0\ 0\ 0 & 0\ \ 1\ \text{(plus multiplicand)} \\
\sum & 0,\ 0\ 1\ 0\ 1 & \text{result}
\end{array}
$$

The result checking (conversion into the decimal system):

0,25	2^{-2}
0,0625	2^{-4}
0,3125	

We see that the algorithm is correct. It gives us the correct result and the result sign. The positive sign was correctly obtained from the multiplication of two positive numbers. Notice, the sign is a component part of calculation.

- **0,625 * (-0,5) = -0,3125:**

Multiplier:	0,625 = 0,1010	
Multiplicand:	(-0,5) = 1,1000	– plus multiplicand
	-(-0,5) = 0,1000	– minus multiplicand

	0, 1 0 1 0 ≡ 0, 1 0 1 0	
$*$	1, 1 0 0 0	
CLR	0 0 0 0 0	reset
+	0 0 0 0 0	0 0 (plus zero)
$\sum$	0 0 0 0 0	
→	0 0 0 0 0	
+	0 1 0 0 0	1 0 (minus multiplicand)
$\sum$	0 1 0 0 0	
→	0 0 1 0 0	
+	1 1 0 0 0	0 1 (plus multiplicand)
$\sum$	1 1 1 0 0	
→	1 1 1 1 0	
+	0 1 0 0 0	1 0 (minus multiplicand)
$\sum$	0 0 1 1 0	
→	0 0 0 1 1	
+	1 1 0 0 0	0 1 (plus multiplicand)
$\sum$	1, 1 0 1 1	result

The result checking (conversion into the decimal system):
We get a negative result. To convert it to the decimal representation, we have to negate the number and add an extra number one.

11011	
00100	- invert of all bits
1	- add an one
00101	= positive result

Now it is easy to check that the absolute value of number 0,3125 is the correct result of the example.

Conversion into the decimal system:

0,25	2^{-2}
0,0625	2^{-4}
0,3125	

- **-0,625 * 0,5 = -0,3125:**

  Multiplier: -0,625 = 1,0110

  Multiplicand: 0,5 = 0,1000 – plus multiplicand

  -0,5 = 1,1000 – minus multiplicand

$$
\begin{array}{rl}
 & 1,\ 0\ 1\ 1\ 0 \equiv 1,\ 0\ 1\ 1\ 0 \\
* & 0,\ 1\ 0\ 0\ 0 \\
\hline
\text{CLR} & 0\ \ 0\ 0\ 0\ 0 \quad \text{reset} \\
+ & 0\ \ 0\ 0\ 0\ 0 \quad 0\ 0\ \text{(plus zero)} \\
\sum & 0\ \ 0\ 0\ 0\ 0 \\
\rightarrow & 0\ \ 0\ 0\ 0\ 0 \\
+ & 1\ \ 1\ 0\ 0\ 0 \quad 1\ 0\ \text{(minus multiplicand)} \\
\sum & 1\ \ 1\ 0\ 0\ 0 \\
\rightarrow & 1\ \ 1\ 1\ 0\ 0 \\
+ & 0\ \ 0\ 0\ 0\ 0 \quad 1\ 1\ \text{(plus zero)} \\
\sum & 1\ \ 1\ 1\ 0\ 0 \\
\rightarrow & 1\ \ 1\ 1\ 1\ 0 \\
+ & 0\ \ 1\ 0\ 0\ 0 \quad 0\ 1\ \text{(plus multiplicand)} \\
\sum & 0\ \ 0\ 1\ 1\ 0 \\
\rightarrow & 0\ \ 0\ 0\ 1\ 1 \\
+ & 1\ \ 1\ 0\ 0\ 0 \quad 1\ 0\ \text{(minus multiplicand)} \\
\sum & 1,\ 1\ 0\ 1\ 1 \quad \text{result}
\end{array}
$$

- **-0,625 * (-0,5) = 0,3125:**

  Multiplier: -0,625 = 1,0110

  Multiplicand: (-0,5) = 1,1000 – plus multiplicand

  -(-0,5) = 0,1000 – minus multiplicand

$$
\begin{array}{rl}
 & 1,\ 0\ 1\ 1\ 0 \equiv 1,\ 0\ 1\ 1\ 0 \\
* & 1,\ 1\ 0\ 0\ 0 \\
\hline
\text{CLR} & 0\ \ 0\ 0\ 0\ 0 \quad \text{reset} \\
+ & 0\ \ 0\ 0\ 0\ 0 \quad 0\ 0\ \text{(plus zero)} \\
\sum & 0\ \ 0\ 0\ 0\ 0 \\
\rightarrow & 0\ \ 0\ 0\ 0\ 0 \\
+ & 0\ \ 1\ 0\ 0\ 0 \quad 1\ 0\ \text{(minus multiplicand)} \\
\sum & 0\ \ 1\ 0\ 0\ 0 \\
\rightarrow & 0\ \ 0\ 1\ 0\ 0 \\
+ & 0\ \ 0\ 0\ 0\ 0 \quad 1\ 1\ \text{(plus zero)} \\
\sum & 0\ \ 0\ 1\ 0\ 0 \\
\rightarrow & 0\ \ 0\ 0\ 1\ 0 \\
+ & 1\ \ 1\ 0\ 0\ 0 \quad 0\ 1\ \text{(plus multiplicand)} \\
\sum & 1\ \ 1\ 0\ 1\ 0 \\
\rightarrow & 1\ \ 1\ 1\ 0\ 1 \\
+ & 0\ \ 1\ 0\ 0\ 0 \quad 1\ 0\ \text{(minus multiplicand)} \\
\sum & 0,\ 0\ 1\ 0\ 1 \quad \text{result}
\end{array}
$$

- **0,6875 * 0,8125 = 0,55859375:**

  Multiplier: 0,6875 = 0,1011

  Multiplicand: 0,8125 = 0,1101 – plus multiplicand

  -0,8125 = 1,0011 – minus multiplicand

	0, 1 0 1 1 $\equiv$ 0, 1 0 1 1	
$*$	0, 1 1 0 1	
CLR	0 0 0 0 0	reset
+	1 0 0 1 1	1 0 (minus multiplicand)
$\sum$	1 0 0 1 1	
$\rightarrow$	1 1 0 0 1	
+	0 0 0 0 0	1 1 (plus zero)
$\sum$	1 1 0 0 1	
$\rightarrow$	1 1 1 0 0	
+	0 1 1 0 1	0 1 (plus multiplicand)
$\sum$	0 1 0 0 1	
$\rightarrow$	0 0 1 0 0	
+	1 0 0 1 1	1 0 (minus multiplicand)
$\sum$	1 0 1 1 1	
$\rightarrow$	1 1 0 1 1	
+	0 1 1 0 1	0 1 (plus multiplicand)
$\sum$	0, 1 0 0 0	result

The result checking (conversion into the decimal system): $(0,1000)_2 = (0,5)_{10}$

And for the twin check, we convert the expected result into the binary system:

0	55859375 x 2
1	1171875 x 2
0	234375 x 2
0	46875 x 2
0	9375 x 2
1	875 x 2
1	75 x 2
1	5 x 2
1	0

$(0,55859375)_{10} = (0,10001111)_2$

It is clear that we get the correct result on 4 bits. Unfortunately the truncation error arises, hence for the accurate result is needed the double number of bits. In this example operands have 4 bits and the accurate result must be displayed on 8 bits.
Hence we need to select the reasonable number of bits, which gives us required accuracy. The small number of bits, as used in the example above is not suitable for realistic calculation. Also the truncation error is normally much smaller than in our result.

Realization of multiplication in the circuit diagram

The multiplier is based on the adder-subtracter unit (fig. 2.30) and the multiplication is executed by Booth's algorithm. From the table 2.5, where the principle of Booth's algorithm is described, we obtain other circuit requirements. We add the control logic to the adder-subtracter where the control unit generates signals as a *NEG* and C_{IN}. These signals are necessary for the operation of subtraction. Then we create a signal for zero counting, we denote it as a the zero counter signal *BL* and this signal controls another new unit called blocking (inhibit) unit.

Blocking unit
makes for prevention of addition the multiplicand to the partial result in Booth's algorithm (adjacent bits 0, 0 or 1, 1). In fact it fills the multiplicand with zeros. The schematic symbol of the blocking unit is shown in fig. 2.31 and the function of the unit is described in the functional table 2.6.

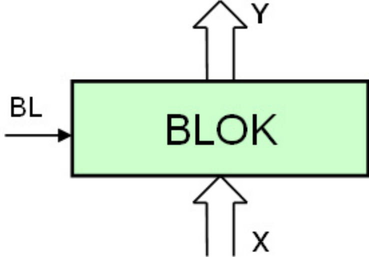


Figure 2.31: Blocking unit

BL	Y
0	0
1	X

Table 2.6: Function of blocking unit

The detailed unit scheme of internal connection is characterised in fig. 2.32.

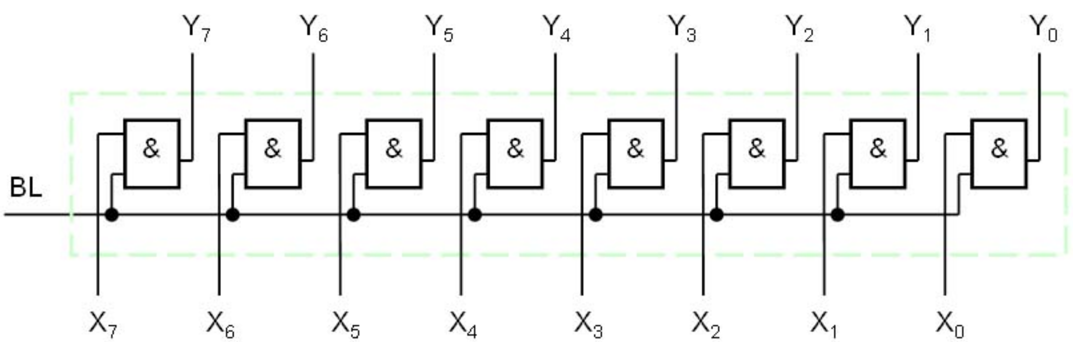


Figure 2.32: Internal connection of blocking unit

In the beginning of the calculation using Booth's multiplication algorithm the multiplicand and the multiplier are in two's complement representation. In the next step of the computation the arithmetic shift of total partial result is required. This partial result is stored in the accumulator (the accumulator theory is described in the chapter 2.3.2). Simultaneously the right shift of the multiplier is required. The multiplier is stored in the shift register (the shift register theory is charactered in the chapter 2.3.2). These two general operations are executed until all bit of multiplier are perused.

The multiplier, which executes the Booth's multiplication algorithm, contains:

- The adder SUM - computes progressive totals during the multiplication.
- Control units for data transfer BNEG, BLOK - execute controlled negation or data blocking (zeroing).
- The memory - it contains registers as memory units for multiplier storage SR (shift register) and the multiplicand storage RN,
- The accumulator ACC - stores the result (or partial results) from the adder.
- The "small zero" unit - stores the $i-1$ bit (required for multiplier register SR extension, in the beginning of calculation this unit is cleared).
- Sub-units of non-equivalence and logical conjunction - generate control signals for multiplication: NEG (data inverse), BL (data blocking) and C_{IN} (add an one to the least significant bit).

The total multiplier scheme is in fig. 2.33.

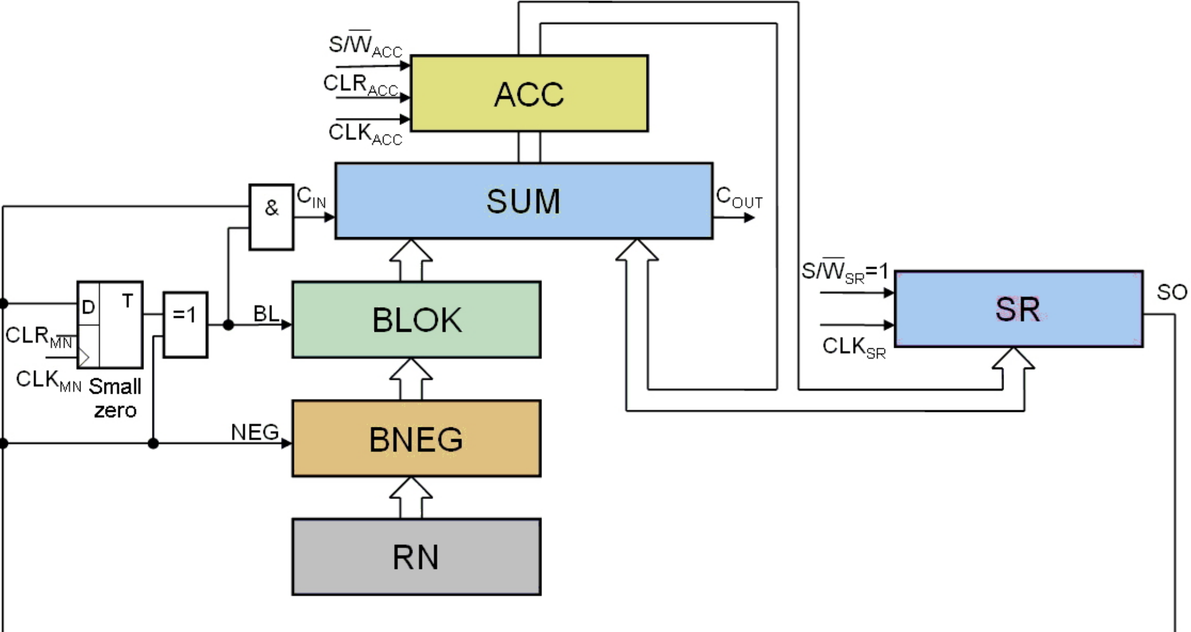


Figure 2.33: Multiplier

Brief description of multiplier function

1. The accumulator AC is cleared, the "small zero" flip-flop is cleared (with the signal $CLR_{ACC} = 1$, $CLR_{MN} = 1$) and control signals are set: $SF/\overline{WR}_{SR} = 1$, $CLK_{ACC} = 0, CLK_{SR} = 0$.
2. Accumulator resetting is ended ($CLR_{ACC} = 0$).
   Rem. In the adder output is the result:
   $<$blocking unit output$>$ + $<$accumulator output$>$ + NEG (C_{IN})
3. The accumulator ACC is set in the writing mode for input data (with the signal $SF/\overline{WR}_{ACC} = 0$).
4. The clock signal is activated CLK_{ACC} and partial result is written into the accumulator ($CLK_{ACC} = 1$).
5. The clock signal is deactivated CLK_{ACC} and it writes data into the accumulator ($CLK_{ACC} = 0$).
6. The accumulator ACC is set to the right shift mode (with the signal $SF/\overline{WR}_{ACC} = 1$).
7. The control shift signal $CLK_{ACC} = 1$ is activated, which shifts the content of accumulator. Simultaneously the control signal $CLK_{SR} = 1$ is activated, which shifts the multiplier shift register.
   Rem. The right shift is executed in the accumulator and also in the shift register.
8. Signals CLK_{ACC} ($CLK_{ACC} = 0$) and CLK_{SR} ($CLK_{SR} = 0$) are deactivated.
9. GOTO step 3, if the total content of the multiplier shift register is not right shifted.

2.5 Microprocessor design

The basis of microprocessor is in the multiplier described in the previous chapter. For required exact calculations of the equation 2.4 ($y_1 = h * y_0 + y_0$), the microprocessor needs to be extended with a multiplexer and a register for result:

- Multiplexer MPX - switches initial values into the adder.
- Result register RV - stores the initial value y_0 for the final addition.
- Multiplicand register RN - stores the step size h.
- Shift register SR - stores the initial value y_0.

Multiplexer

The basic requirement for the multiplexer MPX is:

- Parallel data inputs switching.

The table 2.7 described the circuit function.

ARV	Z
0	X
1	Y

Table 2.7: The function of multiplexer

The schematic symbol of multiplexer is in fig. 2.34.

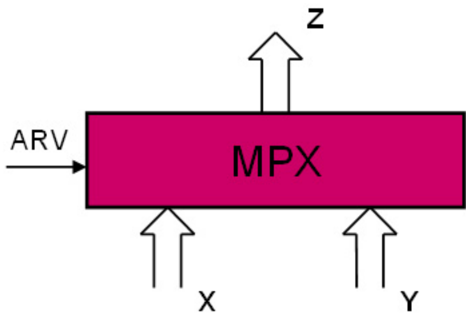


Figure 2.34: Multiplexer

The signal ARV switches data inputs. When the control signal ARV has the logic level "1", on the output Z is loaded data from the data bus Y. When the signal has the logic level "0", on the output is data from the data bus X.

The detailed scheme of multiplexer internal connection is in fig. 2.35.

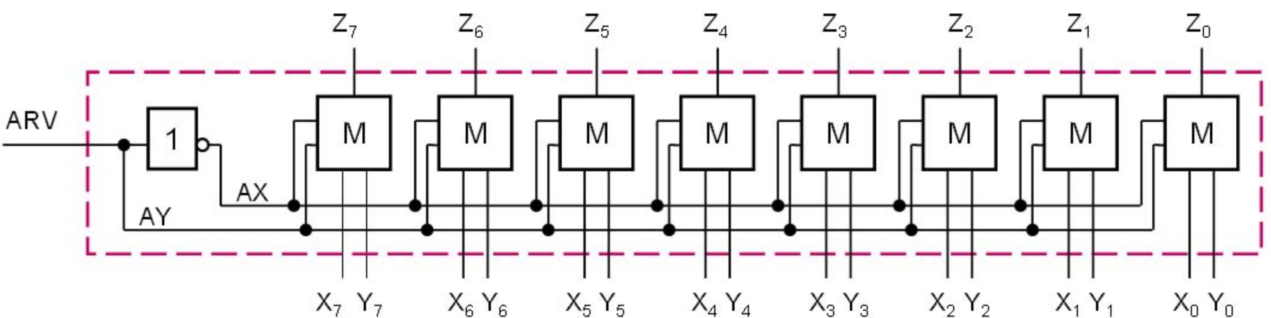


Figure 2.35: Internal connection of multiplexer

The circuit M is in fig. 2.36 and its function is described in the functional table 2.8.

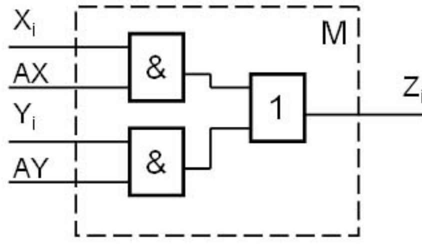


Figure 2.36: Circuit M of multiplexer

AX	AY	Z_i
1	0	X_i
0	1	Y_i

Table 2.8: Functional table of multiplexer

2.5.1 Arithmetic logic unit

The final connection of the microprocessor is in fig. 2.37.

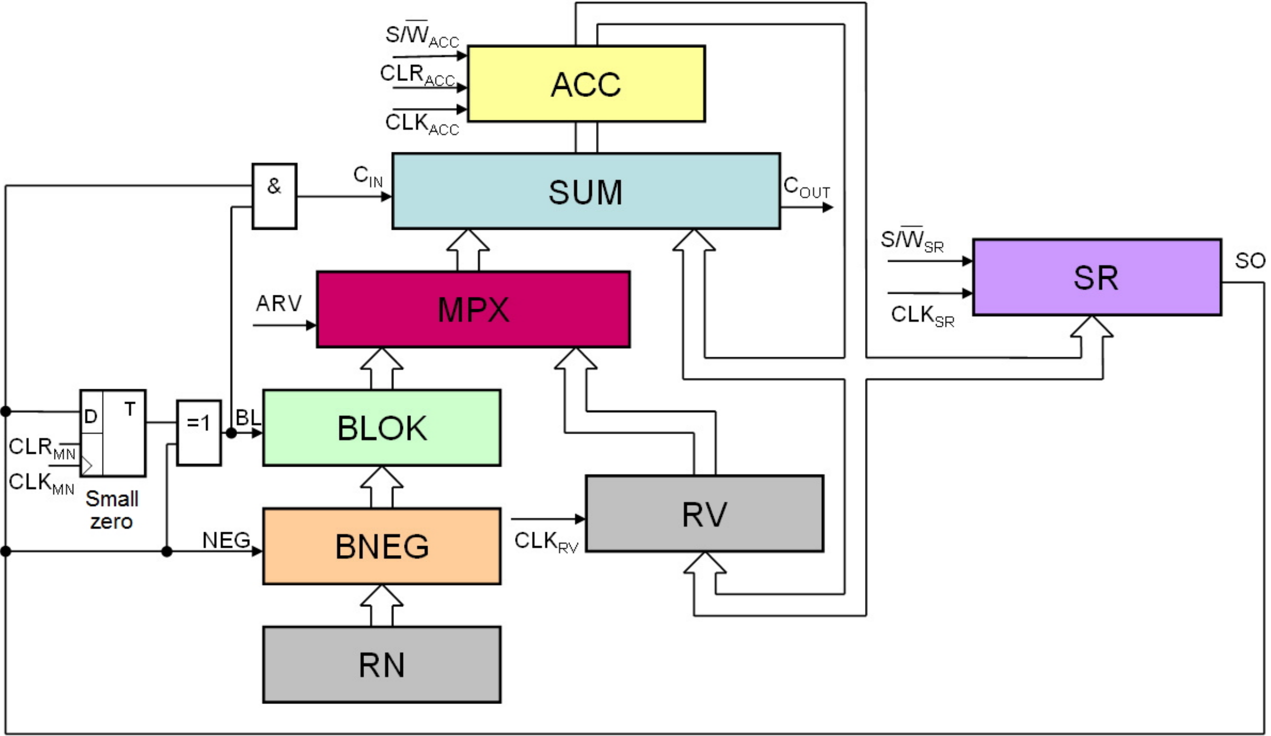


Figure 2.37: Internal connection of microprocessor

Functions of the microprocessor:

1. Registers are cleared, in registers SR and RV the initial value y_0 is loaded and in the register RN is loaded the step size h. D-type flip-flop has a level 0. This flip-flop state is called "small zero" state. It stores the logic value of the multiplier $i-1$ bit (see Booth's algorithm). Since we want to solve the differential equation 2.3, we connect the input and the output of our designed processor. After the value is loaded in the serial input, the control signals *NEG*, *BL* and C_{IN} are set.

2. The multiplicand register output RN is connected at the input BNEG. It guarantees that the value RN is inverted, when the control signal *NEG* comes in the processor input. But for complete inversion the value one - the signal C_{IN} (Carry In) is added. If the zero value is on the output (it means also *NEG=0*), data are not inverted.

3. Now the data from BNEG are on the input of blocking unit (BLOK). BLOK blocks (zeroing) of data. According the table 2.5 it adds only zeros (it does not change values), if adjacent multiplier bits are identical. This circuit is controlled by the signal BL.

4. Using the multiplexer, which is still set on the input X, data are written into the adder SUM. There data is added with the content of the accumulator ACC. And when the clock signal is activated, new data is restored in the accumulator. The result of the multiplication is stored step-by-step in the accumulator .

5. Now it is required to execute the right shift of bits in SR a ACC, but the last left bit is hold. It is according the multiplication algorithm and it is called the arithmetic shift.

6. Computation of progressive totals and shifts is executed step-by-step.

7. When the process of multiplication ended, the multiplexer has to switch the input.

8. The final result is written into the register RV from ACC.

From step 1 to step 4 the process is described as a data throughput in the combinational circuit. It means that the values passage is not controlled by the clock signal, but only by actual levels on inputs of units BNEG, BLOK and SUM. Other steps are controlled from the control unit and a microprogram which executes the function of the whole microprocessor.

2.5.2 Controller – generator of control signals

There are many types of controllers. For clarity the microprogrammable controller is used (see fig. 2.38).

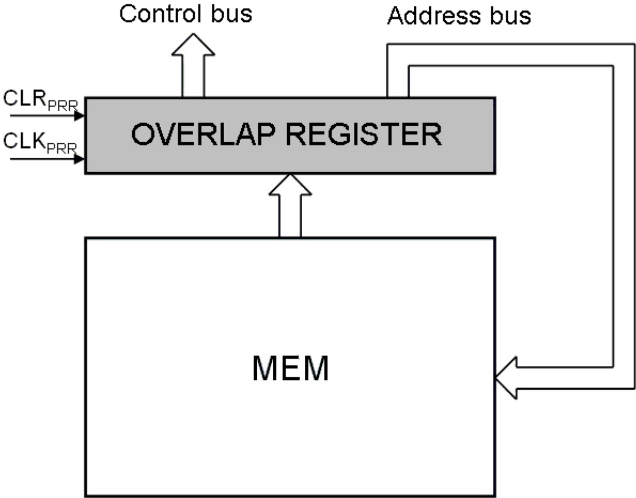


Figure 2.38: Microprogrammable controller

The microprogrammable controller consists of the memory of microprogram and from the overlap register. The overlap register works similarly as a normal register type "parallel in, parallel out". Control signals of overlap register are CLR_{PRR} (zeroing of all flip-flop registers) and CLK_{PRR} (the rising edge of this signal cases the input data write into the overlap register).

The overlap register is partly based on the unit, which generates the control signals - the control bus, and the unit, which generates the address of the microprogram memory - the address bus.

The most complicated part - from the view of the controller design - is the microprogram memory. The format of the instruction word is following:

the field of control signals	the address of next instruction

The possible execution of each microinstruction (the content of each memory address) is described in the table 2.9. It is the microprogram for the operation control, which is the multiplication in multiplier (from fig. 2.33). Signals CLK_{MN} and CLK_{SR} are identical ($CLK_{MN} = CLK_{SR}$), hence we use only one description CLK_{SR}.

CLR_{ACC}	CLR_{MN}	$S/\overline{W}_{ACC}$	$S/\overline{W}_{SR}$	CLK_{ACC}	CLK_{SR}	INSTR	note
1	1	0	1	0	0	0001	Zeroing ACC and MN.
0	0	0	1	0	0	0010	Finish of zeroing.
0	0	0	1	0	0	0011	ACC is in the writing mode.
0	0	0	1	1	0	0100	Write subtotal into ACC.
0	0	0	1	0	0	0101	End of writing.
0	0	1	1	0	0	0110	Set ACC for shifting.
0	0	1	1	1	1	0111	Right shift of ACC and SR.
0	0	1	1	0	0	1000	End of shifting.
if (all content of multiplier register is right shifted) end else instruction=0011							Test end of the multiplication.

Table 2.9: Microprogram of multiplication

Principle description

We are interested in the control bus, that are values of control signals CLR_{ACC}, CLR_{MN}, $S/\overline{W}_{ACC}$, $S/\overline{W}_{SR}$, CLK_{ACC}, CLK_{SR}. Each signal represents one bit of the control bus.

1. The overlap register (with the signal $CLR_{PRR} = 1$) is cleared.
2. The control signal CLR_{PRR} ($CLR_{PRR} = 0$) is deactivated.
   Rem. 1: New zero bits on the address bus pointed to the 1st row of the microprogram memory (in the address 0000 is the first microinstruction <110100 0001> for controlling of multiplication).
   Rem. 2: We suppose, the memory is realized in the way, that the content of the address ($\equiv$ corresponds to the microinstruction) is known on the output immediately (and it generates data inputs of the overlap register).

Function of the controller is managed by the signal CLK_{PRR}:

1. Signals CLK_{PRR} ($CLK_{PRR} = 1$) are activated (rising edge).
   Rem.: The content of data input is written into the overlap register. On the output of the register are bits from the control bus with values $CLR_{ACC} = 1$, $CLR_{MN} = 1$, $S/\overline{W}_{ACC} = 0$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 1st step of brief description of multiplier function above).
2. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated (falling edge).
   Rem.: Simultaneously, address bits are generated (following instruction INSTR - the column in the table 2.9): the address <0001> points to the next microinstruction, which is (after the address is written) transferred into the memory output and it is arranged in data inputs of the overlap register at the same time.
3. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and the microinstruction in the data inputs is written into the overlap register. Corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 0$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 2nd step of brief description above).
4. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
   Rem.: Simultaneously, address bits are generated (of the next instruction): the address <0010> points to the next instruction, which is (after the address is loaded) transferred to the memory output and it is arranged in data inputs of the overlap register.

5. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 0$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 3$^{\text{rd}}$ step of brief description above).
6. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
   Rem.: Simultaneously, address bits are generated (of the next instruction): the address <0011> points to the next instruction.
7. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 0$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 1$, $CLK_{SR} = 0$ (presented in the 4$^{\text{th}}$ step of brief description above).
8. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
   Rem.: Simultaneously address bits are generated (of the next instruction): the address <0100> points to the next instruction.
9. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 0$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 5$^{\text{th}}$ step of brief description above).
10. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
    Rem.: Simultaneously, address bits are generated (of the next instruction): the address <0101> points to the next instruction.
11. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 1$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 6$^{\text{th}}$ step of brief description above).
12. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
    Rem.: Simultaneously, address bits are generated (of the next instruction): the address <0110> points to the next instruction.
13. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 1$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 1$, $CLK_{SR} = 1$ (presented in the 7$^{\text{th}}$ step of brief description above).
14. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
    Rem.: Simultaneously, address bits are generated (of the next instruction): the address <0111> points to the next instruction.
15. The signal CLK_{PRR} ($CLK_{PRR} = 1$) is activated and corresponding bits of the control bus have values $CLR_{ACC} = 0$, $CLR_{MN} = 0$, $S/\overline{W}_{ACC} = 1$, $S/\overline{W}_{SR} = 1$, $CLK_{ACC} = 0$, $CLK_{SR} = 0$ (presented in the 8$^{\text{th}}$ step of brief description above).
16. The signal CLK_{PRR} ($CLK_{PRR} = 0$) is deactivated.
    Rem.: Simultaneously, address bits are generated (of the next instruction): the address <1000> points to the next instruction.

It is important that the rising edge CLK_{PRR} generates new bits of the control bus (CLR_{ACC}, CLR_{MN}, $S/\overline{W}_{ACC}$, $S/\overline{W}_{SR}$, CLK_{ACC}, CLK_{SR}), which control the multiplication.

A very important process is the changeover from one to the next instructions – each microinstruction has stored addresses of the following microinstructions in its address part.

It is naturally possible to optimize the microprogram.

2.5.3 The microprogram of the arithmetic logic unit

Arithmetic-logic unit (in fig. 2.37) is based on the multiplier (in fig. 2.33). The multiplier is extended with the MPX (for switching of adder data input) and with the result register RV (where initial values are stored y_0 in the end of calculation). Hence the microprocessor is extended also with other control signals *ARV* (for controlling the multiplexer function) and CLK_{RV} (the result storage in the result register).

The function of ALU is similar to the function of the multiplier. As soon as the multiplier bits are executed, the multiplexer switches the other input and the initial value is added to the result of multiplication. The whole process is controlled by microprogram, which is based on multiplication microprogram. The structure of microprogram is written in the table 2.10.

CLR_{ACC}	CLR_{MN}	$S/\overline{W}_{ACC}$	$S/\overline{W}_{SR}$	CLK_{ACC}	CLK_{SR}	CLK_{RV}	ARV	$INSTR$
1	1	0	1	0	0	0	0	0001
0	0	0	1	0	0	0	0	0010
0	0	0	1	0	0	0	0	0011
0	0	0	1	1	0	0	0	0100
0	0	0	1	0	0	0	0	0101
0	0	1	1	0	0	0	0	0110
0	0	1	1	1	1	0	0	0111
0	0	1	1	0	0	0	0	1000
if (all content of multiplier register is right shifted) instruction=1001 else instruction=0011								
0	0	0	1	0	0	0	0	1001
0	0	0	1	0	0	0	1	1010
0	0	0	1	1	0	0	1	1011
0	0	0	1	0	0	1	1	1100

Table 2.10: Microprogram of arithmetic-logic unit

Four last grey rows realise following functions:

- Set the accumulator into the writing mode.
- Switch the multiplexer.
- Write the result into the accumulator.
- Write the result into the result register.

2.5.4 Programmatic realization of the elementary processor simulator

Now the program is presented, which allows to simulate all the previous situations and calculations. It was developed for the illustration of described units functions. The capability of the program to simulate contents of all registers in every step is very useful. The simulation is executed in each step described in the chapter 2.5.1, especially in the part where the function of processor is detailed.

Run of simulation

In the main menu, under the item *Simulace* (Simulation), is the main simulation control. There are different choices:

- *Procesor* (Processor): It runs the simulation of the one-processor system which computes the problem using Taylor series method (the calculation of the first term corresponds to Euler method).
- *Rychlost* (Speed): It sets the speed of the animation.
- *Spustit* (Run): It starts the automatic computation of simulation steps.
- *Zastavit* (Stop): It terminates the automatic computation.
- *Krok* (Step): It executes one step of the simulation.

Setting of initial values of simulation

After start up the simulation (*Simulace* → *Procesor*) screen with information about an animation is shown. We can disable to display the screen in every start of the simulation. Then we fill up initial values for simulation in the next screen, see fig. 2.39. These are values: Počáteční podmínka (Initial step) y_0, Integrační krok (Step size) h and Přesnost (Accuracy), which gives us the accuracy of the Taylor series.

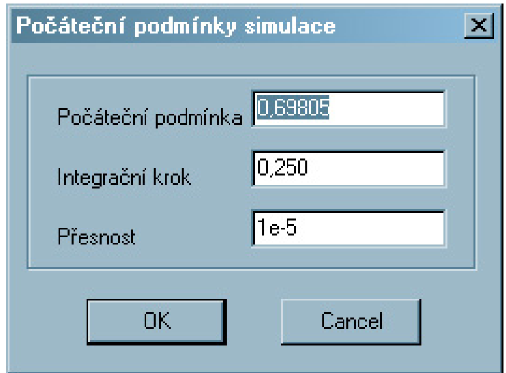


Figure 2.39: Initial values of simulation

Notice that an initial value and a step size should be written in the number with fixed decimal point. That gives the interval for required numbers ($< 1, -1 >$).

Behaviour of simulation

In the case of the setting that $y_0 = 0,69805$ and $h = 0,25$, as in fig. 2.39, the simulation screen is displayed (fig. 2.40).

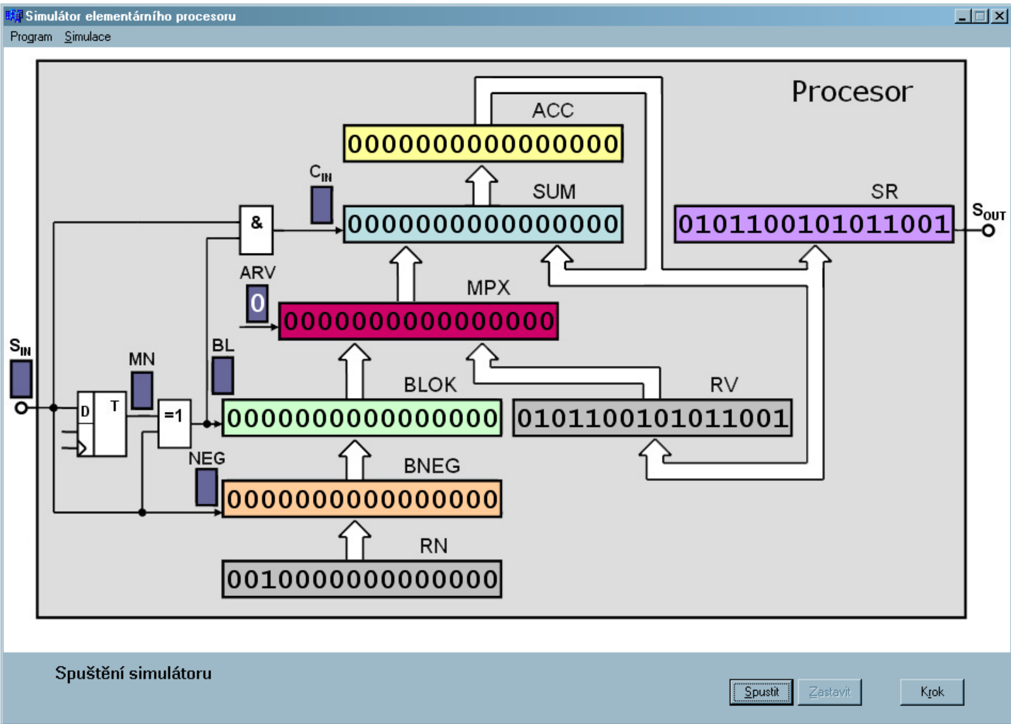


Figure 2.40: Processor schema

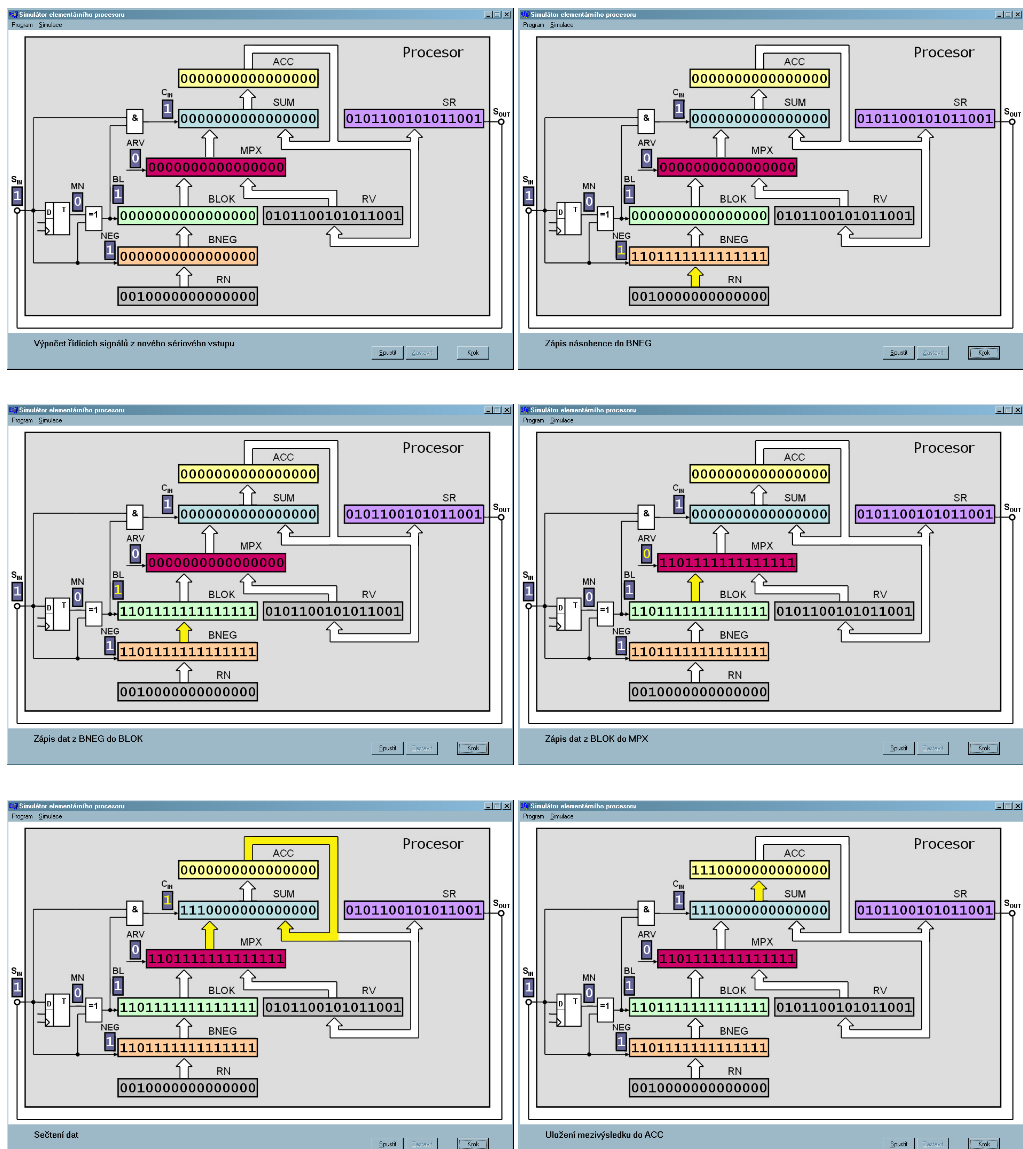
To control the simulation we can press the button *Krok* (Step), which executes one step of the calculation and then the simulation waits for next command. Effectively we can use the button *Spustit* (Run), which computes the whole example and shows the result, and the button *Zastavit* (Stop), which pauses the simulation. Notice that the purpose of functions is not to play some technique, but to simulate the process influenced by initial settings. To continue the paused process, we press the button *Zastavit* (Stop), as well as *Krok* (Step) or *Spustit* (Run).

Notice the text under the picture of schema. There we can find the information about the state, in which is the processor during the simulation. The window *Hodnota výsledku* (Resultant value) gives us a value stored in the register RV, which is the result of all simulation.

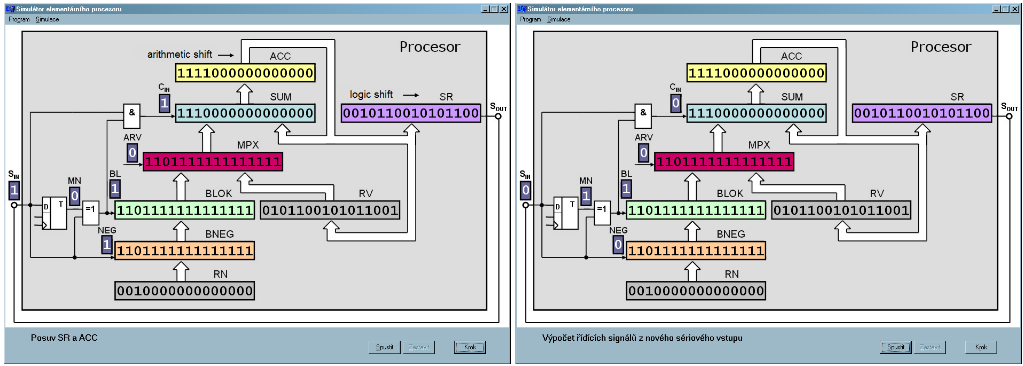
Example of function

The following sequence of simulator displays calculation technique in chosen moments (described below).

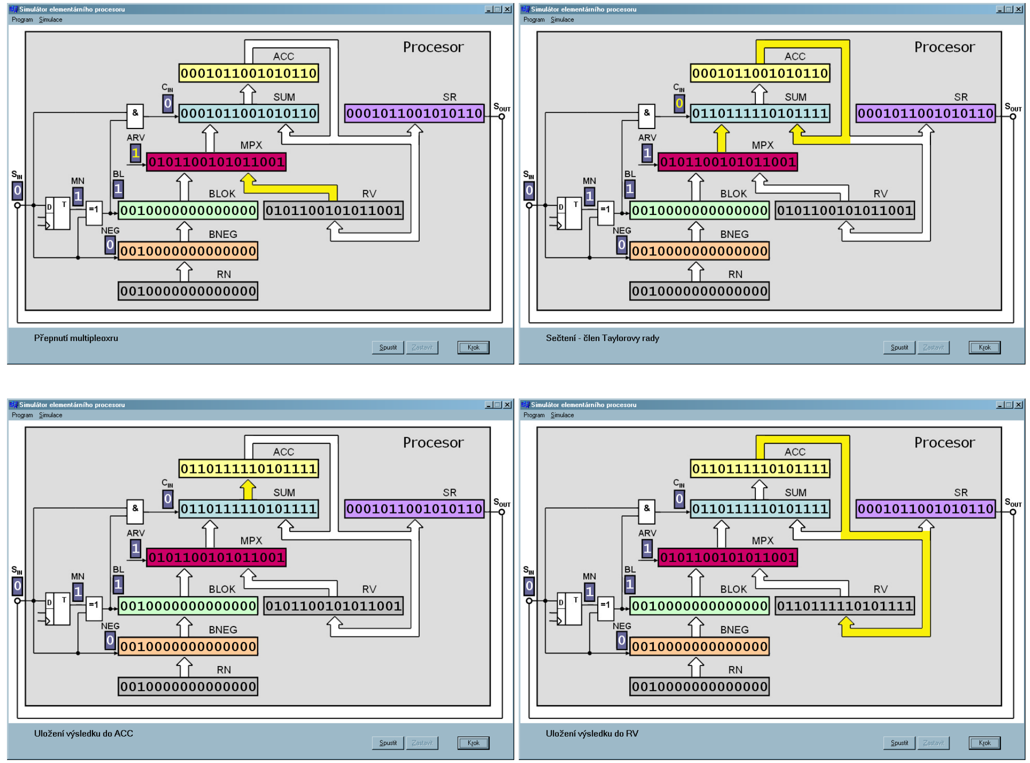
A) The first figure shows the small zero setting (zeroing of the flip-flop) and the computation of control signals for the Booth's multiplication algorithm. The data transfer from the multiplicand register through the inversion unit, the blocking unit, the multiplexer MPX, the adder SUM, which adds the value one to the content of the accumulator ACC, and finally to storage the partial value into the accumulator ACC.



B) It is executed the right shift of the content of accumulator and also the right shift of the shift register (the previous value of the least significant bit of the shift register SR is written into the "small zero" flip-flop). Then the previous sequence of calculation is executed (according A) until all multiplier bits are processed from the register SR.



C) When the multiplication is finished, the result is written into the accumulator ACC. The multiplexer MPX is switched and the content of the accumulator is added with the initial value, which is stored in the result register RV. This result is then rewritten in ACC and finally into the result register RV.



Serial input and serial output

With assumption that the processor in fig. 2.41 has a serial output (S_{OUT}) connected directly into the serial input (S_{IN}), we solve here the Euler differential equation.

$$y' = y \qquad y(0) = y_0$$

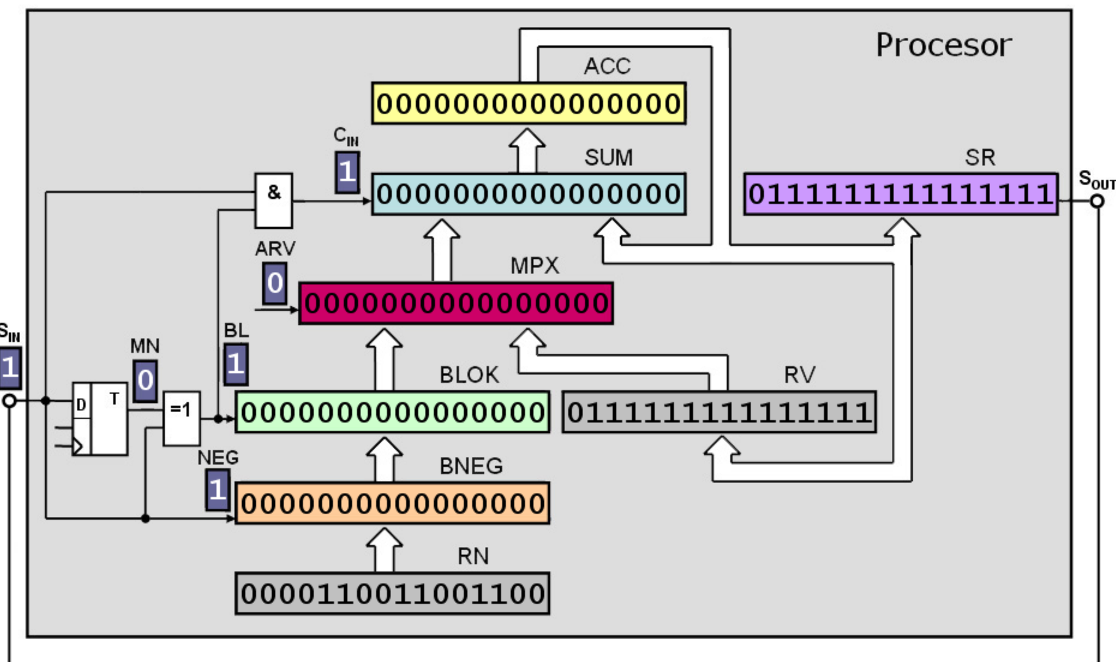


Figure 2.41: Connection for differential equation solving

With assumption that the serial output of the top processor in fig. 2.42 is connected into the serial input of the bottom processor and the serial output of the bottom processor is connected into the serial output of the top processor, we solve the second order Euler differential equation.

$$y'' = -y \qquad y'(0) = y_{p1} \qquad y(0) = y_0$$

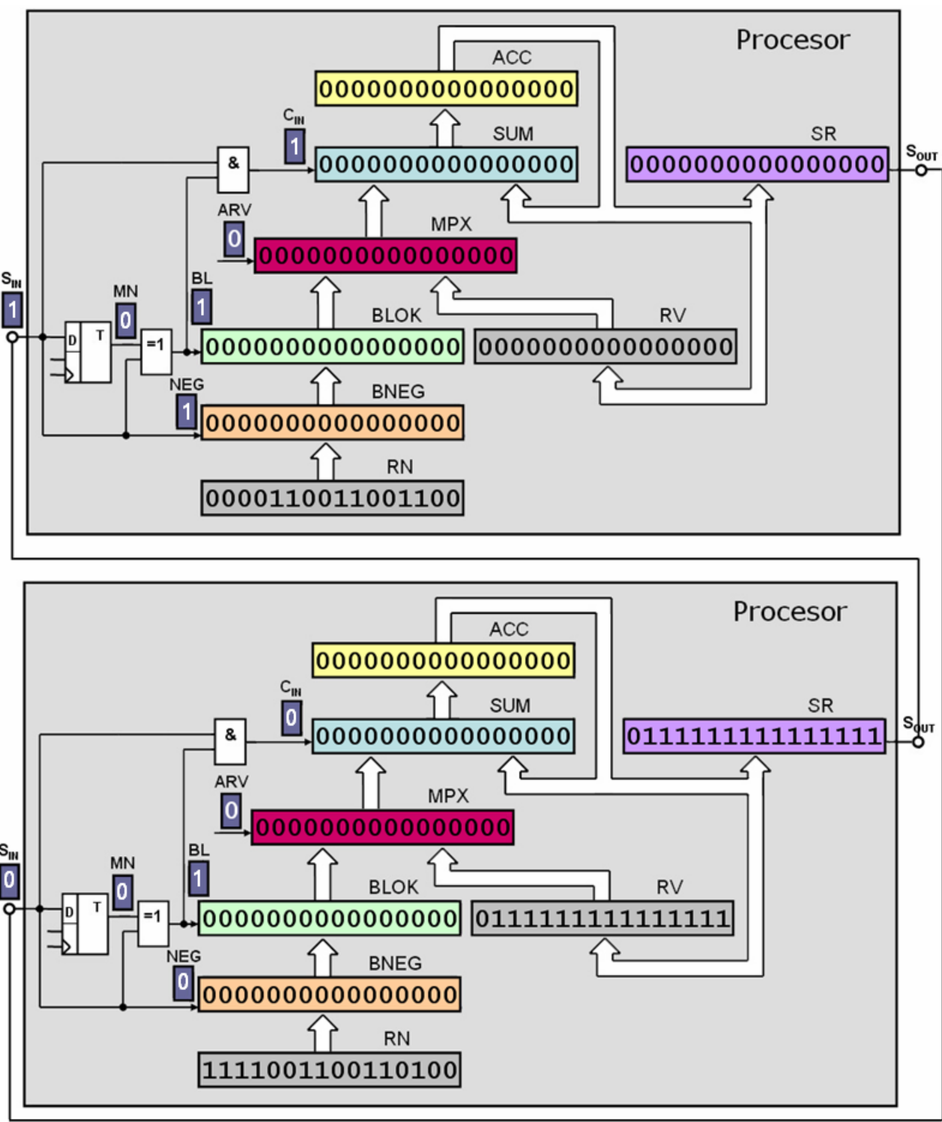


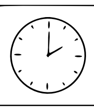
Figure 2.42: Connection for second order differential equation solving

2.6 Resume

The function of register, adder, controller, multiplexer and elementary processor was presented on the concrete mathematical computation in detail. The described theory relates with the numerical integration.

Chapter 3

Pipeline



1:30

3.1 Didactic intention

The aim is to introduce the important method for computation acceleration of computer elements, which is the chain of data-processing technique called pipeline.

3.2 Pipeline principle

We have four identical tasks (yellow, red, green and blue), which comes into the system and the system executes tasks one-by-one.

(Example: Four cars are waiting for the process of wheels mounting on the assembly-line. To each car is fixed the left front wheel - in the section S1, right front wheel - in the section S2, left back wheel - in the section S3 and right back wheel - in the section S4.)

From the technical view, especially from computer elements design, the example represents four instruction in the computer. At first the yellow instruction is executed:

- The instruction selection is made from the memory (I - Instruction Fetch = "the left front wheel is mounted").
- The instruction is decoded (ID - Instruction Decoding = "the right front wheel is mounted").
- Operands are chosen from the memory according to decoded instruction, e.g. the multiplication (OF - Operand Fetch = "the left back wheel is mounted").
- Operands are computed (EX - Execution = "the right back wheel is mounted").

Then the system executes the second (red), third (green) and fourth (blue) instruction (car) in successive steps.

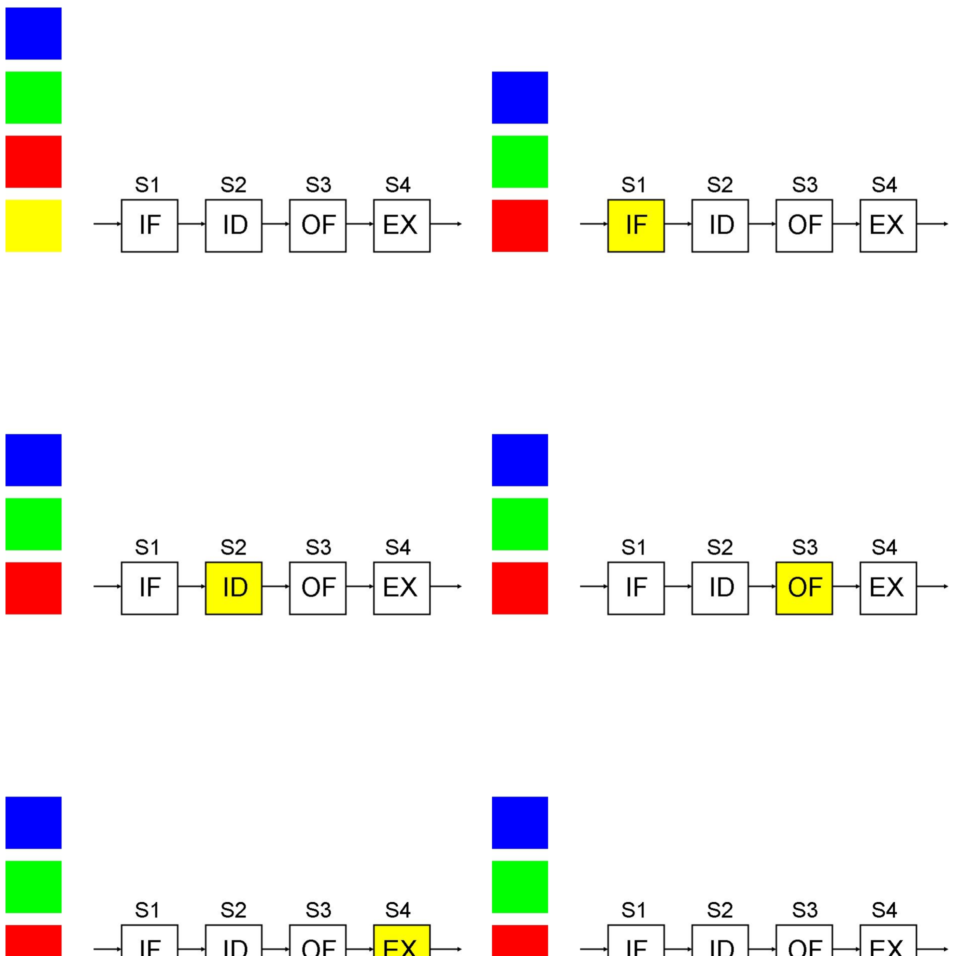
The previous classical processing is often presented in the space–time diagram, that means the section of executing task is marked in specific time intervals.
Rem. This classical processing is easily analogically described as an assembly shop, where one technician mounts the left front wheel, then the right front wheel, the left back wheel and finally the right back wheel to the car.

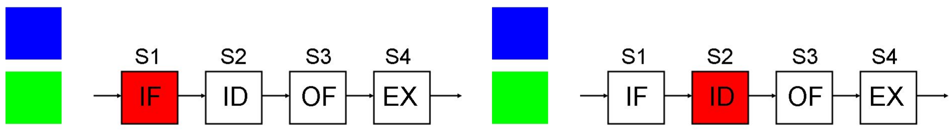
In chained processing it is necessary to provide separate functions for sections S1, S2, S3 and S4. And it is also needed for each section to execute tasks in the same time. The pipeline processing is described in the space–time diagram.

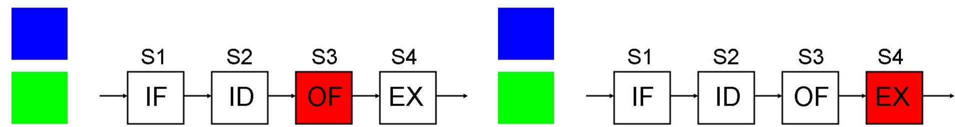
At first the yellow task's Instruction Fetch (IF) is executed in the section S1. After finished processing the first (yellow) task moves to the section S2 and the Instruction Decoding (ID) is started to execute. Simultaneously the second (red) task entries the released section S1 (and IF starts to execute). After processing is finished, in the section S2, the first (yellow) task comes into the section S3. The second (red) task takes the released place in the section S2 and simultaneously the third (green) task entries the free section S1.

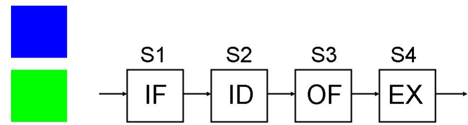
The final operation is executed with the first (yellow) task in the section S4. The second (red) task is allowed to entry into the section S3, the third (green) task moves to the released section S2 and the last fourth (blue) task comes into the section S1, etc.

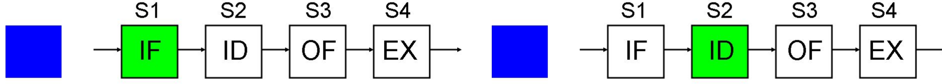
Space–time diagrams represents the acceleration in the pipeline processing. Rem. Adders in the processor Pentium computes similarly.

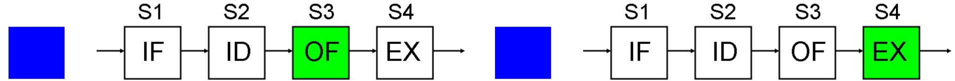


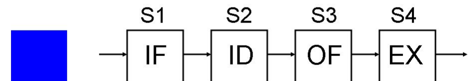

S1
S2
S3
S4
IF
ID
OF
EX
S1
S2
S3
S4
IF
ID
OF
EX

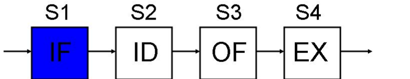

S1
S2
S3
S4
IF
ID
OF
EX
S1
S2
S3
S4
IF
ID
OF
EX

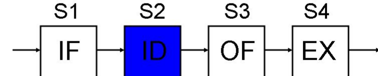

S1
S2
S3
S4
IF
ID
OF
EX

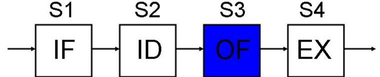

S1
S2
S3
S4
IF
ID
OF
EX
S1
S2
S3
S4
IF
ID
OF
EX

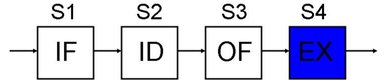

S1
S2
S3
S4
IF
ID
OF
EX
S1
S2
S3
S4
IF
ID
OF
EX

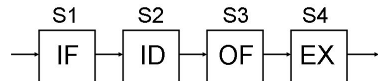

S1
S2
S3
S4
IF
ID
OF
EX

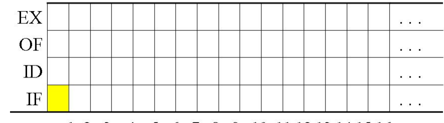

S1
S2
S3
S4
IF
ID
OF
EX

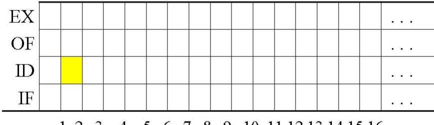

S1
S2
S3
S4
IF
ID
OF
EX

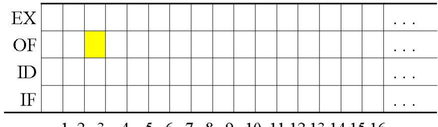

S1
S2
S3
S4
IF
ID
OF
EX

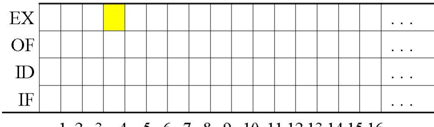

S1
S2
S3
S4
IF
ID
OF
EX

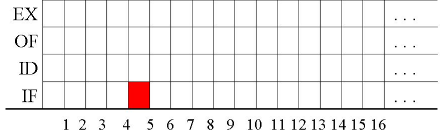

S1
S2
S3
S4
IF
ID
OF
EX

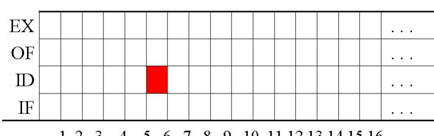

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

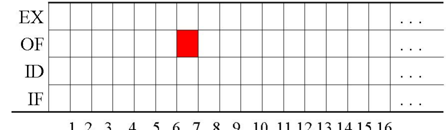

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

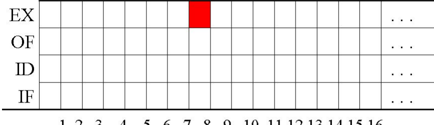

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

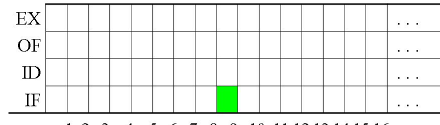

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

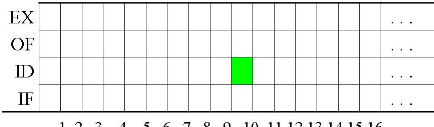

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

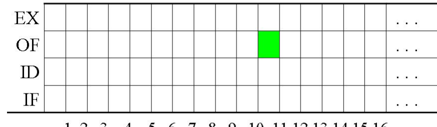

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

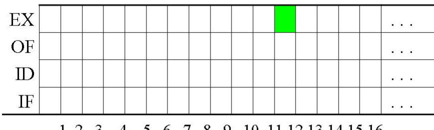

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

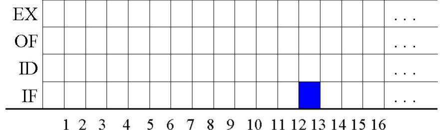

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

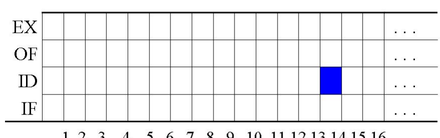

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

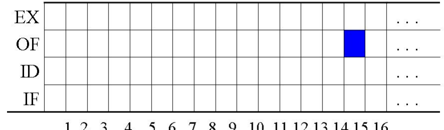

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

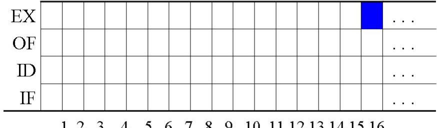

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

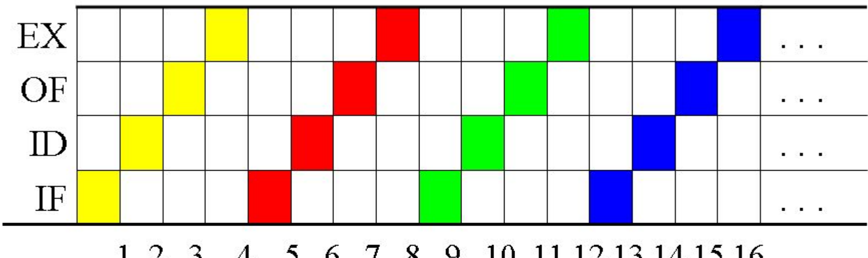

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

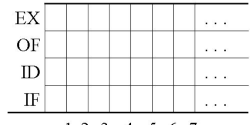

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

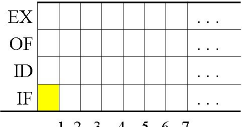

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

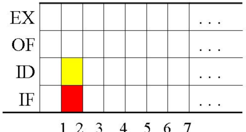

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

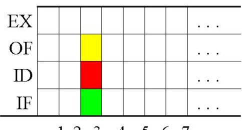

EX
OF
ID
IF
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
. . .

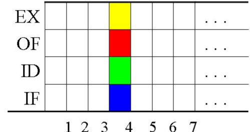

EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

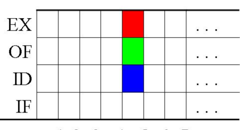

EX
OF
ID
IF
1 2 3 4 5 6 7
...

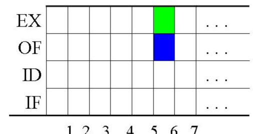

EX
OF
ID
IF
1 2 3 4 5 6 7
...

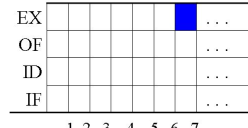

EX
OF
ID
IF
1 2 3 4 5 6 7
...

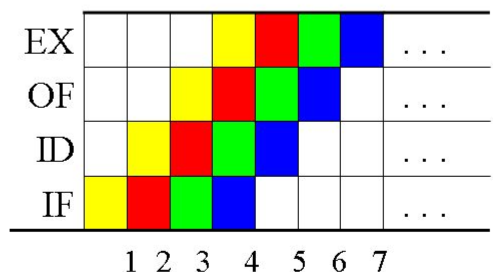

EX
OF
ID
IF
1 2 3 4 5 6 7
...

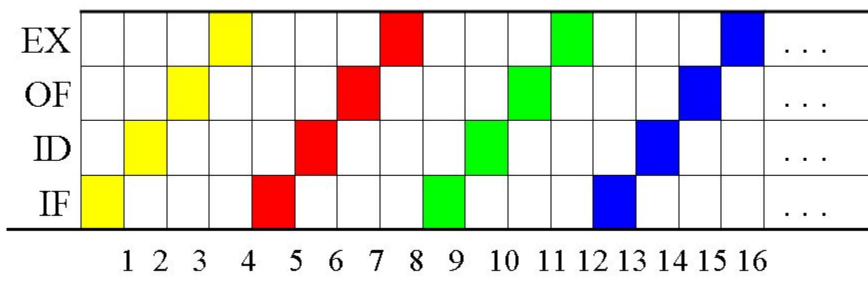

EX
OF
ID
IF
1 2 3 4 5 6 7
...

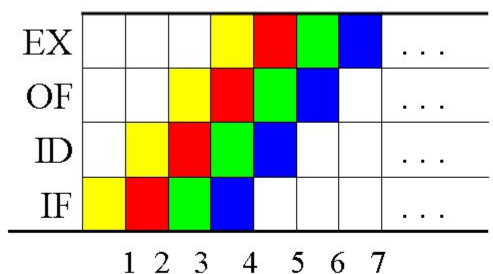

EX
OF
ID
IF
1 2 3 4 5 6 7
...


EX
OF
ID
IF
1 2 3 4 5 6 7
...

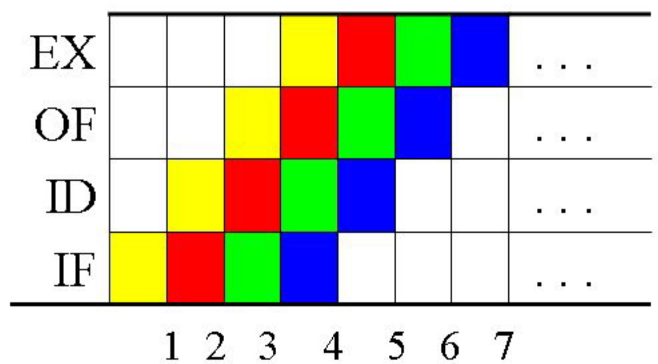

EX
OF
ID
IF
1 2 3 4 5 6 7
...


EX
OF
ID
IF
. . .
1 2 3 4 5 6 7


EX
OF
ID
IF
. . .
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


EX
OF
ID
IF
. . .
1 2 3 4 5 6 7

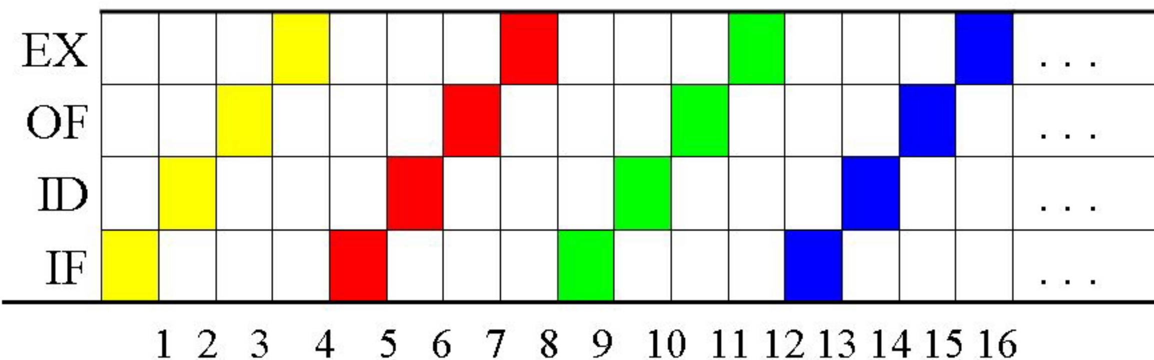
Pipeline



Execution time $= n + k - 1$

n ... sections k ... tasks

Classic processing (not pipeline)



Execution time $= n \cdot k$

n ... sections $\quad$ k ... tasks

Coefficient increase transmittance

$$q = \frac{n.k}{n+k-1}$$
$$k \gg n \qquad k \rightarrow \infty$$

3.3 Resume

The pipeline processing belongs to important methods for computation acceleration of computer elements. The necessary condition of the acceleration in pipeline processing is a repeated calculation of big amount of identical tasks.

The engineering analogy:
The idea of pipelining was firstly used on assembly manufacturing in machine works (serial assembly-lines for car production). Work on the assembly-line is split in each (time identical) operation, which continues immediately. Operations are executed in separate sections.