## For total newbies

# 1. Number systems

## 6.1. Decimal system

## 6.2. Binary system

### 1.1.1. Example

1011 (in binary system) =

$1*(2^3) + 0*(2^2) + 1*(2^1) + 1*(2^0) =$

8 + 0 + 2 + 1 = 11 (in decimal system)

*Remark*

There are 10 kinds of people: those who understand Binary and those who do not.

### 1.1.2. Example

```
00110101    (= 53 decimal)
10001100    (=140)
---------------
11000001    (=193)  (in decimal)
```

## 6.3. **Hexadecimal number system**

*files\01-01 Hexadecimal.docx*

### 1.1.3. Example

Our goal is to convert $1234_8$ into hexadecimal system. First we convert the number into decimal system like so:

$1234_8 = 1x8^3 + 2x8^2 + 3x8^1 + 4x8^0 = 1x512 + 2x64 + 3x8 + 4x1 = 668_{10}$

And we go on into a hexadecimal system:

$668_{10} = 2x256 + 9x16 + 12x1 = 2x16^2 + 9x16^1 + 12x16^0 = 29C_{16}$

## 6.4. 2nd complement

In 2nd complement code, the zero is only 0000 0000, -1 is 1111 11111, -2 is 1111 1110, -3 is 1111 1101, and the lowest possible number -128 is 1000 0000. The largest number for byte with sign is 127 (0111 1111).

# 2. How computer works

## 6.5. Basic functional blocks

*<CZ>files\01-02 Computer CZ.png</CZ>files\01-02 Computer EN.png*

The basic functional blocks of a PC are **microprocessor** (µP), **memory** (RAM, ROM), peripheries (e.g. disc drives, …)

they are interconnected by 3 buses:

- **control** bus (only R/ Ŵ wire shown)
- **address** bus
- **data** bus

## 6.6.

One-directional: µP ->peripheries

width 32 bits: addressing space $2^{32}$

obsolette microprocessors: width only $2^{20}$ = 1MB, which is totally insufficient, memory must switch pages

## 6.7. Data Bus

Bi-directional

At a moment, it can be only one direction: µP ->peripheries or peripheries ->µP the data flow direction is controlled by the R/Ŵ wire of control bus

direction (*read/write*) is always seen from a perspective of the microprocessor

## 6.8. Microprocessor

Contains many specialized memories (= registers)

-

for us the imporant ones are:

- **accumulator**,
- **program counter** PC,
- **stack pointer** SP,

- status register (=Processor Status Word, PSW)

Microprocessor also contains **ALU** = arithmetic-logical unit that executes calculations (results generally stored in an accumulator)

## 3. Instruction cycle

We will start the desctiption of instruction cycle by fetching instruction code. Program counter (PC) contains address of an instruction that is to be executed. First, instruction must be fetched into the µP.

PC content spreads from via address bus into all peripheries

Bbut address is unique, so only 1 device can react to it

Let’s suppose the address is 100

ROM reacts to addess 100. It finds out that at address 100 there is a value of 37. Because the R/Ŵ wire is in its Read state the memory will send the value of 37 to the data bus.

the value from memory will spread via data bus throughout the computer, also into the µP.

Inside of a µP the value will be loaded into the instructions decoder.

The goal of instructions decoder is to understand what type instruction it is and to take care of its execution.

After execution the PC increments by +1

So on address bus new address appears (101) and the whole proces repeats.

Mostly the instructions aren’t so easy.

Were it for example an instruction *„add number that follows to the content of the capacitor*“, first „the number that follows“ would have to load.

That means that instruction cycle would continue: PC would add address 101 to address bus,

Memory would add the content of 101 address, or number 2 in our case, to the data bus

Number 02 would load into µP. But now the value wouldn’t load into instruction decoder but through switched internal switch would be sent to ALU

ALU will do the calculation. The result will most often stay in accumulator.

If an istruction would be to write data to the memory the proces would be similar, only by the R/Ŵ wire the address 101 would be written into instead of read from.

So: program is a sequence of numbers, which can mean either data or instructions, depending on their position (instruction code always goes first)

## 4. Unconditional jump

We described a proces that is completely linear. Such a program would be of no great use.

(Unconditional) jumps to other address are done by inserting a goal adress (often read from memory or calculated) into the program counter PC.

Program then contines from different address

## 5. Conditional jump

Conditional jumps are important – they execute only if certain condition is met.

µP contains set of one bit memories, arrayed into PSW register. After each intruction the bits in PSW are set according to the result of the instruction (ie. if the result is zero, negative, overflow occured etc.)

Conditional jumps will exeute if in PSW is an adequate value.

# 6. Functions

## 6.1. Function call

We often need to repeat some part of a program – e.g. to do a calculation of mathematical function.

Easy: it would be possible to jump at the start of the function from many places

But problem: how to make sure that the jump at the **end** of the function will lead where it is supposed to?!

*<CZ>..\files\01-03 Funkce CZ.png</CZ>..\files\01-03 Funkce EN.png*

Idea: during the jump will the subprogram make a note where it came from

Disadvantage: if µP was to note the return addresses into registers, it could only process as many levels of subprograms as number of registers it has ☹

Solution: in RAM memory it creates a **stack**, dimensioned as needed

Register SP (=Stack Pointer) will **allways point to the top of stack**

At the beginning e.g. SP=5000

*<CZ>..\files\01-04 Stack 1 počáteční.png</CZ>..\files\01-04 Stack 1 počáteční.png*

We will introduce a new instruction, subprogram **call.**

If in the main program at the address 100 were **call** of a function beginning at 2000:

- **return address** (in our case 101) will store into a stack at the place where SP points (meaning the address 5000)
- **SP will move** so that it points to a free space (meaning at 5001)
- **jump to the subprogram will be executed** (in our example to the address 2000)

If inside a function (say at the address 2200) there were call to another subprogram at 3000, this call would be executed in a quite similar fashion.

*..\files\01-06 Stack 3 CALL 3000.png*

## 6.2. Return from the subprogram

At the end of a subprogram a special instruction **RET** = return from subprogram, must be specified.

Stack operations are performed in an inverted fashion to the calls, which means:

- SP will return to the preceding position (the number 2201 is there)

  ..\files\01-07 Stack 4 RET 2201.png

- content of the position (number 2201) will load
- and will be used as a return address.

- So the jump to the address 2201 means that we returned just before the last subprogram call
- When we find RET at the end of subprogram, the procedure repeats.
- So return to the address 101 follows, meaning behind funciton call.

Note that numbers at stack are not cleaned. It is not necessary as they are overwritten by new data during next call. Figure shows situation when function 2000 is called again, from other address.

### 1.1.4. Theory and practices

The procedure described above was just a scatch. Reality can be very different:

- stack grows towards **smaller addresses**, meaning it grows upwards,
- return addresses have multiple bytes, therefore the stacks grows/shrinks several bytes at a time,
- many other information  are stored in a stack.

## 6.3. Interrupt

Typical scenario where we work with interruption is servicing slow devices (printer)

Interrupt works in a similar fashion like a procedure call, but it is hardware invoked

Unlike the CALL the interruption is asynchronous, it can occure at any time ⇨ we must take care of PSW ⇨ PSW is also saved at the stack PSW ⇨ for return we need a special instruction **IRET**

## 6.4. Computer power-on

When the power is turned on, the microprocessor is prepared to work and as soon as it is permitted (hardware signal on the *reset* wire), it will work just described way from the starting address.

The starting address is located in the ROM. The program is stored in the ROM on the hardware level: it cannot be changed, it needs no initialization and it is available immediately after power-up.

The program in the ROM performs basic computer start-up: finds equipment and configuration of the motherboard, performs a RAM memory test, finds out from where to load operating system, etc.. This is the primitive program you communicate with when you chooste from where the operating system should be loaded: e.g. from the disc, from *flash* memory or from the network.

Program finalizes its job by loading the operating system. The operating system is long and complex system, in addition there are a number of different operating systems. But there is a certain unification: on the disk, there is an agreed track, which holds a short *booting progr*am. It is for all operating systems at the same place and doing the same activity: loads and runs the rest of the operating system.

The operating system will take care of all the other work computer.

## 7. Von Neumann‘s architecture

Described computer architecture is called von Neumann’s

Another classical architecture is harvard architecture (it has separated bus for transmission of data from a bus carrying a program – higher speed, especially that of data processing)

Modern development focuses on output upswing ⇨ mainly multiprocessor solutions

Example: An interesting multiprocessor architecture has developed Google corporation.

## 8. Multiprocessor computers

Main problem: even small amount of sequential intructions can slow the computations down considerably

Several conceptions, e.g..

- identical instruction x N processors (operate on varying data)
- matrix processors (hypercube)
- RISC (=*Reduced Instruction Set Computer*)

## 9. RISC

simplified instruction set: all instructions have same number of tacts (ie. 5)

several (ie. 5) instructions are executed simultaneously, but each instruction is in a different stage

effect: during each stage 1 instruction is finished

*..\files\01-09 RISC EN.docx*

## Vývoj programovacích prostředků...</CZ> Evolution of programming instruments...

## 1.

At the very beginning there was a „hot wiring“ of a program into the hardware

Discovery that the program can be saved into a memory similarly to the data:

- 
- assembler (=language of symbolic addresses)
- higher programming languages (FORTRAN, BASIC, COBOL,...)

- Effort to remove restrictions that programming language pose on a programmer.
- generation of languages allowing "almost anything" (Fortran 77, PL/1, ADA)
- Disillusion: programmer only writes a code 20% of a time, only 10% of a time does he come up with algorithms, and whole 70% of a time he debugs !
- Effort to define languages that would allow authomated debugging:
- languages of a 3rd. generation (PASCAL, ...)
- Problem: these languages don’t support cooperation of multiple programmers on a single project!

## 2. Attempts for solution:

- CASE TOOLs a CASE – appliances for organizational support of programmer’s work
- modular programming (MODULA, SIMULA62)
- nonalgorithmic programming (languages 4GL, programming by example, AI methods, declarative programming).

## 3. Programming crisis (for over 50 years)

- 2% of programs are used as they were produced,
- 2-3% are used after modest finish work, not exceeding 10-15% of source texts,
- 20% needed substantial changes,
- 20% returned and remade (generally at the basis of new contracts),
- **Customer has never used this program**

- **Program found unuseable**

Results are graphically expressed on plot.

# 4. Consequences

## 6.1. Finding a way out of the crisis

Finding ways from programming crisis leads in two directions, by revolutionary technology advancements (multiprocessors, neuron networks,…), and by object oriented computer architecture. These architectures solve 2 basic problems]:

- problem of memory primitivism (can’t distingusih what memory cell contains, or we can interpret its content in multiple ways: byte x char x 1/2 word etc.),

- problem of memory globality (even the smallest change in memory changes overal memory state => hard to locate range of change).

# 5. Program = data + algorhitms

## 6.1. Data a data types

Example: with numbers we work entirely differently than with letters. Numbers can be multiplied (3*7=21), letters can’t. Letters can be grouped into words, numbers hardly.

Modern programming languages introduce „strick type checking“. The aim is to detect incorrect data manipulation as soon as possible. And to do so authomatically, during translation.

Example: when compiler discovers that we are trying to multiply two test strings, there is most likely something wrong ☺

**Program = data + algorithms**

Algorithm = operating sequence

Algorithm definition

Algorithm is an exact sequence that solves certain class of tasks, or so called problem area, which is represented by a final instruction sequence. Algorithm guarantees that after a final number of steps on a set of valid entry data the demanded result will be generated.

More academical definition ;-)

Algorithm can be understood as an implementation of a transformational function f, which executes a display from a set of entry data (A) into a set of output data (B). Written in mathematical fashion:

$f: A \rightarrow B.$

## 6.2. History

first references about algorithm and algorithmic solution date back to around 810 A.D when they appeared in works of Uzbek mathematician Al-Chórezmí.

**Example** – quadratic equation

We start solving the equation in a form ax2 + bx +c = 0 (while a≠0) in the domain of real numbers by calculating a discriminant (D), at the base of which we specify a set of roots.

D=b2-4ac.

If D<0 the set of roots (K) is in the domain of real numbers (R) disjoint (empty). Algorithm of calculation is therefore leading us to the result K = Ø.

In case the diskriminant equals to zero there exists one (double) root, which we find with following formula: K= - b/2a

, K = {x}.

If the diskriminant is positive there are exactly two roots, which satisfy the quaratic equation solution. To find them is a matter of solving the equations decribed below:

$x1= (-b+ \sqrt{D})/2a$

$x2= (-b- \sqrt{D})/2a$

For a set of roots applies that $K = \{x1, x2\}$.

## 6.3. Properties of algorithms

Criteria for Algorithms

1. Generality
2. Expliciteness

3. Finiteness
4. Ability to have the result
5. Correctness
6. Repeatability
7. Effectivity

## 6.

Time efficiency of algorithms.

What good would be an algorithm that would fulfill the criterion of final number of realized steps if the time of evaluation would take 1 year?

## 6.4. Memory efficiency of algorithms

Relevance of memory efficiency of the algorithms is especially connected with the past when operating memory was truly scarce commodity.

## 6.5. **Other properties**

One goal (problem) can usually be solved by many various algorithms.

Example: prime numbers can be found by multiple means (brute force x sieve of Eratosthen)

For certain problems there are no algorithmic solutions.

Example: there is no algorithm to write "beautiful" music

## 6.6.

Fundamental theorem of arithmetic that was enounced already by Euclides, states that every natural number can be expressed in form of multiplication of exacly given prime numbers.

(The primes and can be seen as the building blocks, which are a base of all other natural numbers.)

Factorization of any natural number is virtually impossible in cases when this number is a product of too large prime numbers.

This is a mathematical base for cryptography.

## 6.7. **Example of factorization**

Your goal is to break a number 33729211 into a product of two prime numbers.

For this probleme we only know a single algorithm – a „brute force“. It means to try all the combinations.

*Question to ponder: where is the upper boundary of the cycle?*

The result is here

## Excercise

Run and answer tests in Moodle.

Download the appliance for VirtualBox.

Install latest Virtual Box and import the  appliance.

Look at multi-computer and multiprocessor architecture by Google.

## Quiz

# Content