Diskrečioji matematika

www.mif.vu.lt/~algis

Basic structures
Introduction

```pascal
program euclid (input, output);
var x,y: integer;
function gcd (u,v: integer): integer;

  var t: integer; begin
  repeat
    if u<v then begin t := u; u := v; v := t end;
    u := u-v;
  until u = 0;
  gcd := v end;

begin while not eof do
  begin readln (x, y); if (x>0) and (y>0) then writeln (x, y, gcd (x, y)) end;
end.
```
Introduction

- This algorithm has some exceptional features
  - it is applicable only to numbers;
  - it has to be changed every time when something of the environment changes, say if numbers are very long and does not fit into a size of variable (numbers like 1000!).

- For algorithms of applications in the focus of this course, like databases, information systems, etc., they are usually understood in a slightly different way:
  - is repeated many times;
  - in a various circumstances;
  - with different types of data.
Introduction
Von Neuman computing model
At the onset, computers required hardware changes to work on new problems; some historians say that this early stage of “programming” was wiring.

Clearly, requiring hardware changes with each new programming operation was time-consuming, error-prone, and costly.

Von Neumann’s proposal was to store the program instructions right along with the data.

This may sound trivial, but it represented a profound paradigm shift.

The stored program concept was proposed about fifty years ago; to this day, it is the fundamental architecture that fuels computers.
Four Sub-Components

There are four sub-components in von Neumann architecture:

- Memory
- Input/Output (called “IO”)
- Arithmetic-Logic Unit
- Control Unit

While only 4 sub-components are called out, there is a 5th, key player in this operation: a bus, or wire, that connects the components together and over which data flows from one sub-component to another.

As you already know, there are several different flavors of memory. Each type of memory represents cost/benefit tradeoffs between capability and cost …
The Von Neumann Architecture

- Memory: Store data and program, Execute program
- Processor (CPU): Control Unit, ALU: Do arithmetic/logic operations requested by program
- Input-Output: Communicate with "outside world", e.g.
  - Screen
  - Keyboard
  - Storage devices
  - ...

Bus
Memory Types: RAM

- RAM is typically volatile memory (meaning it doesn’t retain voltage settings once power is removed)
- RAM is an array of cells, each with a unique address
- A cell is the minimum unit of access. Originally, this was 8 bits taken together as a byte. In today’s computer, word-sized cells (16 bits, grouped in 4) are more typical.
- RAM gets its name from its access performance. In RAM memory, theoretically, it would take the same amount of time to access any memory cell, regardless of its location with the memory bank (“random” access).
Memory Types: ROM

- It gets its name from its cell-protection feature. This type of memory cell can be read from, but not written to.

- Unlike RAM, ROM is non-volatile; it retains its settings after power is removed.

- ROM is more expensive than RAM, and to protect this investment, you only store critical information in ROM.

- There is a third, key type of memory in every computer – registers.

- Register cells are powerful, costly, and physically located close to the heart of computing.
Memory Operations

- Two basic operations occur within this subcomponent: a *fetch operation*, and a *store*.

- The fetch operation:
  - A cell address is loaded into the MAR.
  - The address is decoded, which means that thru circuitry, a specific cell is located.
  - The data contents contained within that cell is copied into another special register, called a *Machine Data Register (MDR)*.
  - Note that this operation is non-destructive – that is, the data contents are copied, but not destroyed.
Memory Operations

- The second memory operation is called a *store*.
  - The fetch is like a read operation; the store is like a write operation
  - In the store, the address of the cell into which data is going to be stored is moved to the MAR and decoded.
  - Contents from yet another special register, called an *accumulator*, are copied into the cell location (held in the MAR).
  - This operation is destructive, meaning that whatever data was originally contained at that memory location is overwritten by the value copied from the accumulator.
I/O: Input and Output

- There is both a human-machine interface and a machine-machine interface to I/O.
  - Examples of the human-machine interface include a keyboard, screen or printer.
  - Examples of the machine-machine interface include things like mass storage and secondary storage devices.
- Input and output devices are the least standardized of the various sub-components, which means that you have to pay extra special attention to make certain that your input or output devices are compatible with your machine.
The ALU

- The third component in the von Neumann architecture is called the Arithmetic Logic Unit.
- This is the subcomponent that performs the arithmetic and logic operations for which we have been building parts.
- The ALU is the “brain” of the computer.
The ALU

- It houses the special memory locations, called registers, of which we have already considered.

- The ALU is important enough that we will come back to it later. For now, just realize that it contains the circuitry to perform addition, subtraction, multiplication and division, as well as logical comparisons (less than, equal to and greater than).
The last of the four subcomponents is the Control Unit.
The control unit is the work horse that drives the fetch and execute cycle.
Remember we said that in memory, a cell address is loaded into the MAR – it is the control unit that figures out which address is loaded, and what operation is to be performed with the data moved to the MDR.
We will come back and look in detail at how the Control Unit performs this task.
Memory

- $2^k \times m$ array of stored bits
- **Address**
  - unique ($k$-bit) identifier of location
- **Contents**
  - $m$-bit value stored in location

**Basic Operations:**
- **LOAD**
  - read a value from a memory location
- **STORE**
  - write a value to a memory location
Boolean data

Data values: \{false, true\}

In C/C++: false = 0, true = 1 (or nonzero)

Could store 1 value per bit, but usually use a byte (or word)

Operations:

- and  
- or  
- not

\[
\begin{array}{c|c|c}
\text{&&} & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{||} & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{c|c}
x & \neg x \\
0 & 1 \\
1 & 0 \\
\end{array}
\]
Character Data

Store numeric codes (ASCII, EBCDIC, Unicode)
1 byte for ASCII and EBCDIC,
2 bytes for Unicode (see examples on p. 35).

Basic operation: comparison of chars to determine if $==$, $<$, $>$, etc.
uses their numeric codes (i.e. uses their ordinal values)
Integer Data

**Nonnegative (unsigned) integer:**

Type `unsigned` (and variations) in C++

Store its base-two representation in a fixed number $w$ of bits

(e.g., $w = 16$ or $w = 32$)

$$88 = 0000000001011000_2$$

**Signed integer:** Type `int` (and variations) in C++

Store in a fixed number $w$ of bits using one of the following representations:
Sign-magnitude representation

Save one bit (usually most significant) for sign

\[ 0 = +, \quad 1 = - \]

Use base-two representation in the other bits.

\[ 88 \rightarrow 0\_000000001011000 \]
\[ \uparrow \text{sign bit} \]
\[ -88 \rightarrow 1\_000000001011000 \]

1. Cumbersome for arithmetic computations
2. 2 0’s in this scheme
3. Incrementing by one results in \textit{subtraction} of one, not addition!
Complement representation

For non-negative $n$:
Use ordinary base-two representation with leading (sign) bit 0

For negative $n$ ($-n$):
(1) Find $w$-bit base-2 representation of $n$
(2) Complement each bit.
(3) Add 1

Example: $-88$
1. $88$ as a 16-bit base-two number $0000000001011000$
2. Complement this bit string $1111111110100111$
3. Add 1 $1111111110101000$

Same as sign mag.
Good for arithmetic computation

(see p. 38)

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

These work for both + and – integers

5 + 7:

\[
\begin{align*}
\text{0000000000000101} \\
+\text{0000000000000111} \\
\overline{\text{00000000000001100}}
\end{align*}
\]

111 carry bits

5 + –6:

\[
\begin{align*}
\text{0000000000000101} \\
+\text{11111111111111010} \\
\overline{\text{1111111111111111}}
\end{align*}
\]
Problems with Integer Representation

- Limited Capacity -- a finite number of bits
- Overflow and Underflow:
  
  **Overflow**
  - addition or multiplication can exceed largest value permitted by storage scheme

  **Underflow**
  - subtraction or multiplication can exceed smallest value permitted by storage scheme

- Not a perfect representation of (mathematical) integers
  - can only store a finite (sub)range of them
How is Real Data represented?

Types `float` and `double` (and variations) in C++

Single precision (IEEE Floating-Point Format)

1. Write binary representation in floating-point form:
   \[ b_1.b_2b_3 \ldots \times 2^k \] with each \( b_i \) a bit and \( b_1 = 1 \) (unless number is 0)

   mantissa \quad exponent

Example: \( 22.625 = 10110.101 \) (see p.41)

Floating point form: \( 1.0110101_2 \times 2^4 \)

```
+ 127
```

\[ \text{sign} \quad \text{exponent} \quad \text{mantissa} \]

\[ 010000001 101101 010100000000000000000000000000000 \]

\textbf{double:}
- Exp: 11 bits,
- bias 1023
- Mant: 52 bits