2 Časová a prostorová složitost

V minulé kapitole jsme zkusili navrhnout algoritmy pro několik jednoduchých úloh. Zjistili jsme přitom, že pro každou úlohu existuje algoritmů více. Všechny fungují, ale jak poznat, který z nich je nejlepší? A co vlastně znamenají pojmy „lepší“ a „horší“? Kritérií kvality může být mnoho. Nás v této knize budou zajímat časové a paměťové nároky programu, tzn. rychlost výpočtu a velikost potřebné operační paměti počítače.

Jako první srovnávací metoda nás nejspíš napadne srovnávané algoritmy naprogramovat v nějakém programovacím jazyce, spustit je na větší množině testovacích dat a měřit se stopkami v ruce (nebo alespoň s těmi zabudovanými do operačního systému), který z nich je lepší. Takový postup se skutečně v praxi používá, z teoretického hlediska je však nevhodný. Kdybychom chtěli svým kolegům popsat vlastnosti určitého algoritmu, jen stěží nám postačí „na mém stroji doběhl do hodiny“. A jak bude fungovat na jiném stroji, s odlišnou architekturou, naprogramovaný v jiném jazyce, pod jiným operačním systémem, pro jinou sadu vstupních dat?

V této kapitole vybudujeme způsob, jak měřit dobu běhu algoritmu a jeho paměťové nároky nezávisle na technických podrobnostech – konkrétním stroji, jazyku, operačním systému. Těmto mírám budeme říkat *časová a prostorová složitost* algoritmu.

2.1 Jak fungují počítače uvnitř

Definice pojmu „počítač“ není samozřejmá. V současnosti i v historii bychom jistě našli spoustu strojů, kterým by se tak dalo říkat. Co mají společného? My se přidržíme všeobecně uznávané definice, kterou v roce 1946 vyslovil vynikající matematik John von Neumann.

Von neumannovský počítač se skládá z pěti funkčních jednotek:

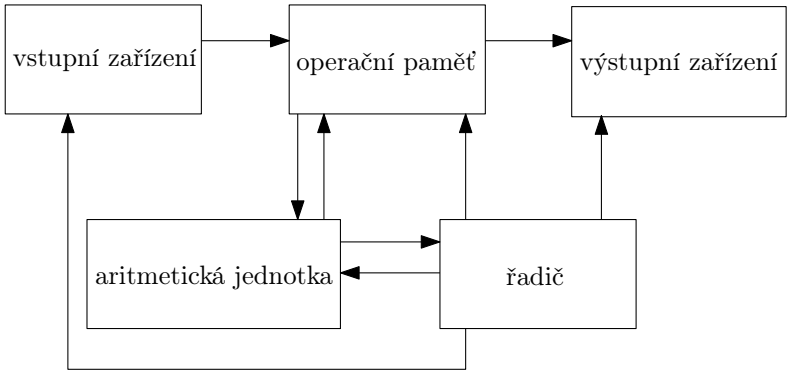
- *řídicí jednotka (řadič)* – koordinuje činnost ostatních jednotek a určuje, co mají v kterém okamžiku dělat,
- *aritmeticko-logická jednotka (ALU)* – provádí numerické výpočty, vyhodnocuje podmínky, ...,
- *operační paměť* – uchovává data a program,
- *vstupní zařízení* – zařízení, odkud se do počítače dostávají data ke zpracování,
- *výstupní zařízení* – do tohoto zařízení zapisuje počítač výsledky své činnosti.

Struktura počítače je nezávislá na zpracovávaných problémech, na řešení problému se musí zvenčí zavést návod na zpracování (program) a musí se uložit do paměti; bez tohoto programu není stroj schopen práce.

Programy, data, mezivýsledky a konečné výsledky se ukládají do téže paměti.[1] Paměť je rozdělená na stejně velké *buňky,* které jsou průběžně očíslované; pomocí čísla buňky (*adresy*) se dá přečíst nebo změnit obsah buňky. V každé buňce je uloženo číslo. Všechna data i instrukce programu kódujeme pomocí čísel. Kódování a dekódování zabezpečují vhodné logické obvody v řídicí jednotce.

Po sobě jdoucí instrukce programu se nacházejí v po sobě jdoucích paměťových buňkách. Instrukcemi skoku se dá odklonit od zpracování instrukcí v uloženém pořadí. Existují následující typy instrukcí:

- aritmetické instrukce (sčítání, násobení, ukládání konstant, ...)
- logické instrukce (porovnání, AND, OR, ...)
- instrukce přenosu (z paměti do ALU a opačně, na vstup a výstup)
- podmíněné a nepodmíněné skoky
- ostatní (čekání, zastavení, ...)



Obrázek 2.1: Schéma von Neumannova počítače (po šipkách tečou data a povely)

Přesné specifikaci počítače, tedy způsobu vzájemného propojení jednotek, jejich komunikace a programování, popisu instrukční sady, říkáme *architektura.* Nezabíhejme do detailů fungování běžných osobních počítačů, čili popisu jejich architektury. V každém z nich se však jednotky chovají tak, jak popsal von Neumann. Z našeho hlediska bude nejdůležitější podívat se co se děje, pokud na počítači vytvoříme a spustíme program.

[1] Tím se liší von Neumannova architektura od architektury zvané *harvardská,* jež program a data důsledně odděluje. Výhodou von Neumannova počítače je kromě jednoduchosti i to, že program může modifikovat sám sebe. Výhodou harvardské pak možnost přistupovat k programu i datům současně.

Algoritmus zapíšeme obvykle ve formě vyššího programovacího jazyka. Zde je příklad v jazyce C.

```
#include <stdio.h>

int main(void)
{
   static char s[] = "Hello world\n";
   int i, n = sizeof(s);
   for (i = 0; i < n; i++)
      putchar(s[i]);
   return 0;
}
```

Aby řídicí jednotka mohla program provést, musíme nejdříve spustit *kompilátor* neboli *překladač*. To je nějaký jiný program, který náš program z jazyka C přeloží do takzvaného *strojového kódu*. Tedy do posloupnosti jednoduchých instrukcí kódovaných pomocí čísel, jimž už počítač přímo rozumí. Na rozdíl od původního příkladu, který na všech počítačích s překladačem jazyka C bude vypadat stejně, strojový kód se bude lišit architekturu od architektury, operační systém od operačního systému, dokonce překladač od překladače.

Ukážeme příklad úseku strojového kódu, který vznikl po překladu našeho příkladu v operačním systému Linux na architektuře AMD64. Tato architektura kromě běžné paměti pracuje ještě s *registry* – těch jsou řádově jednotky, také se do nich ukládají čísla a jsou přístupné rychleji než operační paměť. Můžeme si představit, že jsou uložené uvnitř aritmeticko-logické jednotky.

Aby se lidem jednotlivé instrukce lépe četly, mají přiřazeny své symbolické názvy. Tomuto jazyku symbolických instrukcí se říká *assembler*. Kromě symbolických názvů instrukcí dovoluje assembler ještě pro pohodlí pojmenovat adresy a několik dalších užitečných věcí.

```
MAIN:   pushq   %rbx            # uschovej registr RBX na zásobník
        xorl    %ebx, %ebx      # vynuluj registr EBX
LOOP:   movsbl  str(%rbx), %edi # ulož do EDI RBX-tý znak řetězce
        incq    %rbx            # zvyš RBX o 1
        call    putchar         # zavolej funkci putchar z knihovny
        cmpq    $13, %rbx       # už máme v RBX napočítáno 13 znaků?
        jne     LOOP            # pokud ne, skoč na LOOP
        xorl    %eax, %eax      # vynuluj EAX: nastav návratový kód 0
        popq    %rbx            # vrať do RBX obsah ze zásobníku
        ret                     # vrať se z podprogramu
STR:    .string "Hello world\n"
```