



Operační systémy 1

# Řízení výpočtu, volání podprogramů

Petr Krajča



Katedra informatiky  
Univerzita Palackého v Olomouci



- standard IEEE 754
- čísla zakódovaná ve tvaru

$$\text{hodnota} = (-1)^{\text{znamenko}} \times \text{mantisa} \times 2^{\text{exponent}}$$

- několik variant s různou velikostí a přesností

## Poznámky

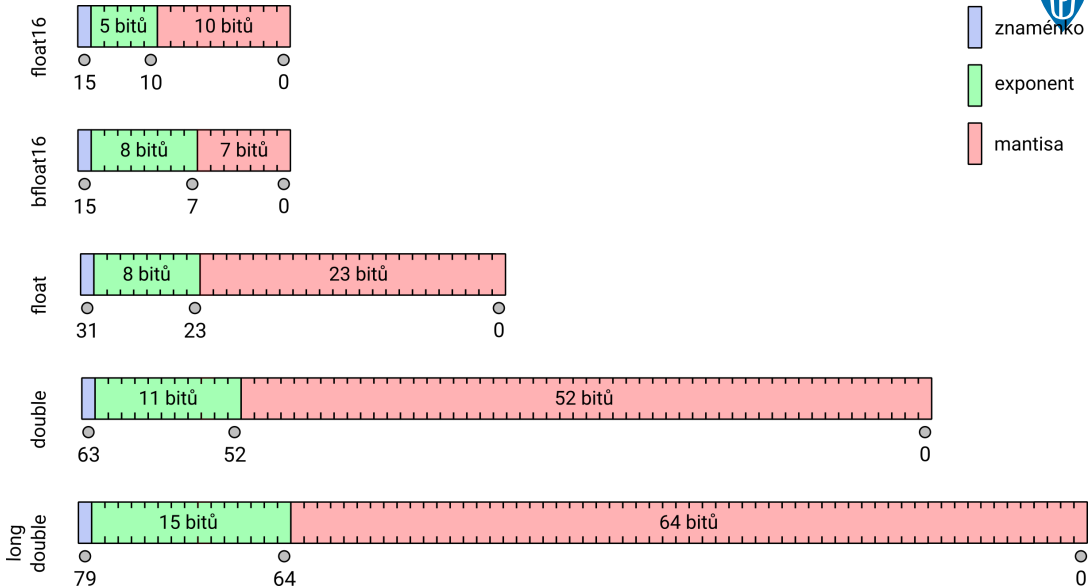
- existuje záporná nula
- existují nekonečna – maximální exponent + nulová mantisa
- existuje NaN (Not a Number) – maximální exponent + nenulová mantisa



označení	typ	velikost (bity)	exponent (bity)	mantisa (bity)
jednoduchá přesnost	float	32	8	23
dvojitá přesnost	double	64	11	52
rozšířená přesnost	long double	80	15	64
poloviční přesnost	float16, half (float)	16	5	10
brain floating point	bfloat16	16	8	7



- znaménko
- exponent
- mantisa





- podpora „multimédií“
- SIMD (single instruction multiple data)
- 128bitové registry XMM0 – XMM15
  - kapacita pro 4 FP hodnoty s jednoduchou přesností
  - kapacita pro 2 FP hodnoty dvojitou přesností
  - vektory celých čísel (16 8bitových hodnot, 8 16bitových, atd.); včetně saturace
- později přibylo rozšíření XMM0 – XMM15 na 256 bitů (registry YMM0 – YMM15)
- nejnověji AVX-512 rozšiřuje tyto registry na 512 bitů a přidává další (ZMM0 – ZMM31)
- operace:
  - aritmetické (sčítání, odčítání, násobení, dělení, druhá odmocnina, minimum, maximum)
  - porovnání hodnot
  - manipulace s vektory (uspořádání hodnot, konverze datových typů)
  - práce s cache



- jednotlivé operace nastavují hodnoty bitů v registru RF/EF
- záleží na operaci, které příznaky nastavuje
- **příznaky pro řízení výpočtu**
  - SF (sign flag) – podle toho, jestli výsledek je nezáporný (0) nebo záporný (1)
  - ZF (zero flag) – výsledek byl nula
  - CF (carry flag) – výsledek je větší nebo menší než největší/nejmenší možné číslo
  - OF (overflow flag) – příznak přetečení znaménkové hodnoty mimo daný rozsah
- **další příznaky**
  - AF (auxiliary carry flag) – přenos ze čtvrtého do pátého bitu (BCD čísla)
  - PF (parity flag) – nastaven na jedna při sudé paritě (pouze dolních 8 bitů)
- **řídící příznaky**
  - TF (trap flag) – slouží ke krokování
  - DF (direction flag) – ovlivňuje chování instrukcí blokového přesunu
  - IOPL (I/O privilege level) – úroveň oprávnění (2 bity, pouze jádro)
  - IF (Interrupt enable flag) – možnost zablokovat některá přerušení (pouze jádro)



- procesor zpracovává jednu instrukci za druhou (pokud není uvedeno jinak)  $\implies$  skok
- nepodmíněný skok
  - operace `JMP r/m/i` – ekvivalent `GOTO` (použití při implementaci smyček)
- není přítomná operace ekvivalentní `if`
- podmíněný skok je operace ve tvaru `Jcc`, provede skok na místo v programu, pokud jsou nastaveny příslušné příznaky
- např. `JZ i` (provede skok, pokud výsledek předchozí operace byl nula), dále `JNZ`, `JS`, `JNS`, ...

## Porovnávání čísel

- srovnání čísel jako rozdíl (operace `CMP r/m, r/m/i`, je jako `SUB`, ale neprovádí přiřazení)
- `JE` skok při rovnosti, `JNE`, při nerovnosti (v podstatě operace `JZ` a `JNZ`)
- a další operace

- příklad použití
- podmíněné skoky po porovnání neznaménkových hodnot

instrukce	alt. jméno	příznaky	podmínka
JA	JNBE	$(CF \text{ or } ZF) = 0$	$A > B$
JAE	JNB	$CF = 0$	$A \geq B$
JB	JNAE	$CF = 1$	$A < B$
JBE	JNA	$(CF \text{ or } ZF) = 1$	$A \leq B$

- podmíněné skoky po porovnání znaménkových hodnot

instrukce	alt. jméno	příznaky	podmínka
JG	JNLE	$(SF = OF) \ \& \ ZF = 0$	$A > B$
JGE	JNL	$(SF = OF)$	$A \geq B$
JL	JNGE	$(SF \neq OF)$	$A < B$
JLE	JNG	$(SF \neq OF) \text{ nebo } ZF = 1$	$A \leq B$



```
00000000 <main>:
0:   8b 4c 24 04      mov     ecx,DWORD PTR [esp+0x4]
4:   b8 01 00 00 00   mov     eax,0x1
9:   83 f9 00         cmp     ecx,0x0
c:   0f 8e 0a 00 00 00 jle     1c <main+0x1c>
12:  f7 e9           imul   ecx
14:  83 e9 01        sub     ecx,0x1
17:  e9 ed ff ff ff  jmp     9 <main+0x9>
1c:  c3             ret
```

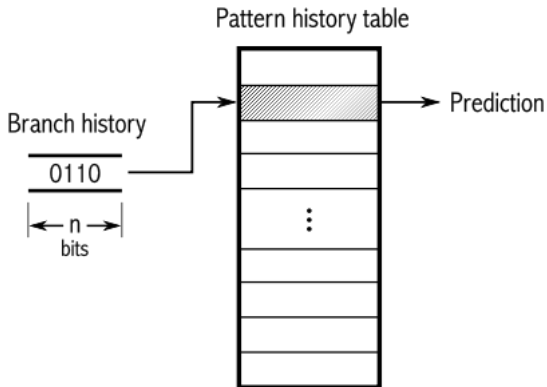
- pro snadnější implementaci cyklů byly zavedeny speciální operace
- JRCXZ, JECXZ, JCXZ – provede skok, pokud registr RCX/ECX/CX je nulový (není potřeba explicitně testovat registr \*CX)
- LOOP – odečte jedničku od RCX/ECX, a pokud v registru RCX/ECX není nula, provede skok

## Poznámky

- uvádí se, že složené operace jsou pomalejší než jednotlivé kroky
- (obecně) podmíněné skoky zpomalují běh programu  $\implies$  zrušení výpočtu v pipeline
- procesory implementují různé heuristiky pro odhad, jestli daný skok bude proveden
  - statický přístup (např. u skoků zpět se předpokládá, že budou provedeny)
  - dynamický přístup (na základě historie skoků se rozhodne)
  - náповěda poskytnutá programátorem (příznak v kódu)



- procesory používají kombinace výše zmíněných metod (hlavně dynamický odhad); různé metody
- čtyřstavové počítadlo se saturací:
- při každém průchodu procesor ukládá do Branch Prediction Buffer (2b příznak, jestli byl skok proveden, nebo ne) a postupně přechází mezi čtyřmi stavy:
  - 11 – strongly taken
  - 10 – weakly taken
  - 01 – weakly not taken
  - 00 – strongly not taken
- až na stav 00 předpokládá, že skok bude proveden
- velikost BPB a počáteční stav počítadla se mezi procesory liší
- problém: pravidelné střídání úspěšnosti  $\implies$  dvouúrovňový odhad (vzor chování)



- pro každý vzor existuje odhad založený na výše zmíněném přístupu
- velikost vzoru závisí na procesoru
- globální vs. lokální tabulka

- procesor má vyčleněný úsek paměti pro zásobník (LIFO)  $\implies$  pomocné výpočty, návratové adresy, lokální proměnné, ...
- vyšší prog. jazyky obvykle neumožňují přímou manipulaci se zásobníkem (přesto má zásadní úlohu)
- procesory i386/AMD64 mají jeden zásobník, který roste shora dolů
- registr RSP ukazuje na vrchol zásobníku (`mov rax, [rsp]` načte hodnotu na vrcholu zásobníku)

- uložení/odebrání hodnot pomocí operací:

```
push r/m/i           ;; sub rsp, 8  
                     ;; mov [rsp], op1
```

```
pop r/m              ;; mov op, [rsp]  
                     ;; add rsp, 8
```

- registr RSP musí vždy obsahovat číslo, které je násobek osmi



- k volání podprogramu se používá instrukce `call r/m/i`  $\implies$  uloží na zásobník hodnotu registru IP a provede skok

```
push rip                ;; tato operace neexistuje
jmp <addr>
```

- k návratu z funkce se používá instrukce `ret`  $\implies$  odebere hodnotu ze zásobníku a provede skok na adresu danou touto hodnotou

```
add rsp, 8
jmp [rsp - 8]
```

- použití zásobníku umožňuje rekurzi



- předání parametrů
- vytvoření prostoru pro lokální proměnné
- provedení podprogramu/funkce
- odstranění proměnných, příp. argumentů
- návrat z podprogramu/funkce, předání výsledku



- způsob, jakým jsou předávány argumenty funkcím, jsou jen konvence (specifické pro překladač, i když často součástí specifikace ABI OS)
- předávání pomocí registrů (dohodnou se urč. registry), příp. zbývající argumenty se uloží na zásobník
- předávání argumentů čistě přes zásobník
- kdo odstraní předané argumenty ze zásobníku? (volaná funkce nebo volající?)

## Rámec funkce (stack frame)

- při volání funkcí se na zásobníku vytváří tzv. rámec (stack frame)
- obsahuje předané argumenty, adresu návratu, příp. lokální proměnné
- k přístupu k tomuto rámci se používá registr RBP



## i386: konvence C (cdecl)

- argumenty jsou předané čistě přes zásobník
- zprava doleva
- argumenty ze zásobníku odstraňuje volající
- umožňuje funkce s proměnlivým počtem parametrů



## Volání funkce

- 1 na zásobník jsou uloženy parametry funkce zprava doleva (`push <arg>`)
- 2 zavolá se funkce (`call <adresa>`), na zásobník se uloží adresa návratu
- 3 funkce uloží obsah registru EBP na zásobník (adresa předchozího rámce)
- 4 funkce uloží do registru EBP obsah ESP (začátek nového rámce)
- 5 vytvoří se na zásobníku místo pro lokální proměnné
- 6 na zásobník se uloží registry, které se budou měnit (`push <reg>`)

## Návrat z funkce

- 1 obnovíme hodnoty registrů (které byly umístěny na zásobník `pop <reg>`)
- 2 odstraníme lokální proměnné (lze k tomu použít obsah EBP)
- 3 obnovíme hodnotu EBP
- 4 provedeme návrat z funkce `ret`
- 5 odstraníme argumenty ze zásobníku (lze použít přičtení k ESP)



```

    ...
    argument n
    ...
EBP + 12 --> argument 2
EBP + 8  --> argument 1
          návratová adresa
          původní EBP
EBP - 4  --> první lokální proměnná
EBP - 8  --> druhá lokální proměnná
          ...
ESP      --> poslední lokální proměnná
```



## Volání funkce

```
push arg2      ;; druhý argument
push arg1      ;; první argument
call func
add esp, 8     ;; odstraní oba argumenty ze zásobníku
```

## Tělo funkce

```
push ebp
mov ebp, esp
sub esp, n     ;; vytvoří místo pro lokální proměnné
push ...      ;; uloží obsah používaných registrů
...           ;; tělo funkce
pop ...       ;; vrátí hodnoty registrů do původního stavu
mov esp, ebp  ;; odstraní lokální proměnné
pop ebp
ret
```



- první argument leží na adrese  $[EBP + 8]$ , druhý na  $[EBP + 12]$ , atd.
- první lokální proměnná na  $[EBP - 4]$ , druhá na  $[EBP - 8]$ , atd.

### Uchovávání registrů

- uchovávání všech použitých registrů na začátku každé funkce nemusí být efektivní
- používá se konvence, kdy se registry dělí na
  - *callee-saved* – o uchování hodnot se stará volaný (EBX, ESI, EDI)
  - *caller-saved* – o uchování hodnot se stará volající (EAX, ECX, EDX)
- po návratu z funkce mohou registry EAX, ECX a EDX obsahovat cokoliv



- prvních 6 argumentů: RDI, RSI, RDX, RCX, R8, R9
- čísla s plovoucí řádovou čárkou přes: XMM0-XMM7 (počet použitých XMM registrů musí být v registrů AL)
- zbytek přes zásobník (zprava doleva)
- návratové hodnoty přes RAX nebo XMM0
- pod vrcholem zásobníku oblast 128 B (červená zóna) pro libovolné použití

```
// a -> RDI, b -> XMM0, c -> RSI, d -> XMM1; 2 -> a1
```

```
void foo(int a, double b, int c, float d);
```

- caller-saved: RAX, RDI, RSI, RDX, RCX, R8, R9, R10, R11
- callee-saved: RBX, RSP, RBP, R12, R13, R14, R15



Pozice	Obsah	Rámec
RBP + 16 + 8 * (n - 1)	n-tý argument na zásobníku	předchozí
...	...	
RBP + 16	první argument na zásobníku	
RBP + 8	návratová adresa	aktuální
RBP	původní RBP	
RBP - 8	lok. proměnné a nespecifikovaná data	
...	...	
RSP	...	
RSP - 128	červená zóna	



- první 4 argumenty: RCX, RDX, R8, R9
- čísla s plovoucí řádovou čárkou přes: XMM0-XMM3
- na zásobníku se vytváří stínové místo pro uložení argumentů
- zbytek přes zásobník
- návratové hodnoty přes RAX nebo XMM0

```
// a -> RCX, b -> XMM1, c -> R8, d -> XMM3
```

```
void foo(int a, double b, int c, float d);
```

```
sub    rsp, 0x28                ; (0x20 + 0x08 -- kvůli zarovnání po call)
```

```
movabs rcx, <addr: msg>
```

```
call   printf
```

```
add    rsp, 0x28
```

- caller-saved: RAX, RCX, RDX, R8, R9, R10, R11
- callee-saved: RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15



Pozice	Obsah	Rámec
...	...	předchozí
RBP + 56	6. argument	
RBP + 48	5. argument	
...	...	
RBP + 16	prostor pro uložení registrů	
RBP + 8	návratová adresa	aktuální
RBP	původní RBP	
RBP - 8	lok. proměnné a nespécifikovaná data	
...	...	
RSP	...	