1. Počítačová reprezentace čísel a aritmetika

Rychlý náhled kapitoly: V této kapitole budou vysvětleny principy ukládání a manipulace s čísly v počítači, a to jak celých, tak reálných. Standardní počítačová reprezentace čísel a počítačová aritmetika obsahují důležité body, jichž bychom si při tvorbě počítačových modelů měli být vědomi. Ušetříme si tak dlouhé hodiny přemýšlení, proč něco funguje jinak, než se domníváme, že by mělo fungovat.

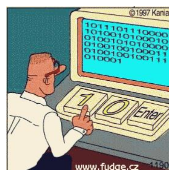
Cíle kapitoly:

- Naučit se pracovat s binární, oktalovou a hexadecimální reprezentací čísel.
- Porozumět reprezentaci celých čísel v počítači a vyhnout se případným problémům s přetečením nebo podtečením.
- Porozumět reprezentaci reálných čísel s pohyblivou desetinnou čárkou a její implementaci normou ANSI/IEEE 754-1985.
- Porozumět aritmetice v pohyblivou desetinné čárce a jejím úskalím.

Klíčová slova kapitoly: Celá čísla, znaménková, neznaménková; reálná čísla; bit; byte, bajt; binární, oktalová, hexadecimální reprezentace; pohyblivá desetinná čárka; FP čísla, normalizovaná, subnormální; norma ANSI/IEEE 754-1985; jednoduchá, dvojnásobná přesnost; strojové epsilon; zaokrouhlování; výjimky; stabilita

Motto: *10.0 × 0.1 se zřídkakdy rovná 1.0*

Celá čísla (*integers*) včetně znaků (*characters*) a reálná čísla (*real*) mají odlišný způsob počítačové reprezentace a provádění aritmetických operací. Počítač disponuje konečným prostorem pro uskladnění číslic; proto je nutné jak spočetnou množinu celých čísel, tak reálné kontinuum vhodně reprezentovat.



1.1. Binární čísla

Dekadická soustava – báze (*basis*) 10, deset číslic (*digits*) 0, . . . , 9:

$$(109.375)_{10} = 1 \times 10^2 + 0 \times 10^1 + 9 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$$

Technologie hardwaru předurčuje vyjádření čísel ve dvojkové soustavě (*binary system*) – báze 2, dvě číslice (*bit* = ***binary** digit*) 0, 1:

$$\begin{aligned}(109.375)_{10} &= 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\quad + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= (1101101.011)_2\end{aligned}$$

1	1	0	1	1	0	1	0	1	1	
MSB						LSB	MSB		LSB	LSB = least significant bit MSB = most significant bit

1.1.1. Aritmetické operace s binárními čísly

Analogicky jako v dekadické soustavě:

$1 + 0 = 1,\ 1 + 1 = (10)_2 = (2)_{10},\ (10)_2 + 1 = (11)_2 = (3)_{10}$, atd.

$$
\begin{array}{rrrrrrr}
 & & 1 & 1 & 1 & 1 & 0 \\
+ & & & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\qquad
\begin{array}{rrrrrrr}
 & & & & 1 & 1 & 1 \\
\times & & & & 1 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 \\
 & & & 1 & 1 & 1 & \\
 & & 1 & 1 & 1 & & \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

1.1.2. Konverze z dekadické do dvojkové soustavy

Celočíselná část:

109	Kvocient	54	27	13	6	3	1	0
: 2	Zbytek	1	0	1	1	0	1	1
		LSB						MSB

Zlomková část:

0.375	Zlomek	0.75	0.5	0
× 2	Celé č.	0	1	1
		MSB		LSB

Pozor! Opačné pořadí: LSB → MSB.

1.1.3. Oktalová a hexadecimální reprezentace

Oktalová: báze 8, osm číslic $0, \ldots, 7$, pokryto skupinou tří bitů. Binární číslo se rozdělí na skupiny tří bitů od tečky doleva a doprava (a přidají se nulové vycpávky, je-li třeba). Každá trojice se pak přepíše jako jediná oktalová číslice $0, \ldots, 7$:

$$(109.375)_{10} = (\boxed{001}\,\boxed{101}\,\boxed{101}\,.\,\boxed{011})_2 = (155.3)_8$$

Hexadecimální: báze 16, šestnáct číslic $0, \ldots, 9$, $\mathrm{A} \equiv 10$, $\mathrm{B} \equiv 11$, $\mathrm{C} \equiv 12$, $\mathrm{D} \equiv 13$, $\mathrm{E} \equiv 14$, $\mathrm{F} \equiv 15$, pokryto skupinou čtyř bitů. Binární číslo se rozdělí na skupiny čtyř bitů od tečky doleva a doprava (a přidají se nulové vycpávky, je-li třeba). Každá čtveřice se pak přepíše jako jediná hexadecimální číslice $0, \ldots, 9, \mathrm{A}, \ldots, \mathrm{F}$:

$$(109.375)_{10} = (\boxed{0110}\,\boxed{1101}\,.\,\boxed{0110})_2 = (6\mathrm{D}.6)_{16}$$

1.2. Reprezentace dat

Data a programy jsou v paměti ukládány v binárním formátu. Paměť je organizována do skupin po 8 bitech (b) – bajtů (*byte*, *bytes*, B) – nejmenší adresovatelná jednotka: 1 B = 8 b.

1 KB	1024 B	2^{10} B
1 MB	1024 KB = 1048576 B	2^{20} B
1 GB	1024 MB = 1073741824 B	2^{30} B
1 TB	1024 GB = 1099511627776 B	2^{40} B

Různé druhy fyzické paměti (hierarchie paměti):

Typ paměti	**Velikost**	**Přístupová doba**
Registry CPU	8 B	1 clock cycle
Cache, Level 1	126 KB–512 KB	1 ns
Cache, Level 2	512 KB–8 MB	10 ns
Hlavní paměť (RAM)	8 MB–4 GB	60 ns
Pevný disk (HDD)	2 GB–100 GB	10 ms

Několik bajtů (obvykle 4, tj. 32 b) tvoří slovo (*word*).

1.2.1. Znaky (characters)

Písmena abecedy (velká i malá), interpunkce, různé další symboly. ASCII (American Standard Code for Information Interchange): 7 bitů pro 1 znak ⇒ $2^7 = 128$ reprezentovatelných znaků (0–127: ‚dolní polovina ASCII tabulky‘).

Zbylé pozice do 1 B: 128–255 (příp. (-128)–(-1) v případě znaménkového znaku): ‚horní polovina‘, obvykle pro znaky národních abeced, není dodržován standard ISO 8859-2 ⇒ problémy s kódováním češtiny (další cp1250, PC-latin2, Kamenických, Mac OS, . . .).

Vyzkoušejte program `char2ascii` (zobrazí ASCII kód zadaného znaku).

Znaky v paměti Umístění znaku v RAM (chápané jako dlouhá sekvence bajtů): jednoznačně určeno adresou. Např. máme-li k dispozici 512 MB RAM, tj. 2^{30} B, adresy pokrývají $0, \dots, 2^{30} - 1 = (3FFFFFFF)_{16}$.

Vyzkoušejte program `charaddr` (zobrazí adresu, na níž je uložen zadaný znak).

Pro uložení posloupnosti znaků (znakového řetězce, *character string*) ‚Ahoj!' potřebujeme alokovat pět po sobě jdoucích bajtů, např.:

Adresa	**Obsah**	**Poznámka**
BFFFF260	'A'	první bajt
BFFFF261	'h'	
BFFFF262	'o'	
BFFFF263	'j'	
BFFFF264	'!'	poslední bajt

Pro přístup potřebujeme znát adresu prvního znaku (zde BFFFF260) a délku řetězce (zde 5).

Vyzkoušejte program `straddr` (zobrazí adresy, na nichž začíná a končí zadaný znakový řetězec).

1.2.2. Celá čísla (integers)

Neznaménková (unsigned) 1 B může reprezentovat celá čísla $0, \ldots, 2^8 - 1 = 255$, pro běžné aplikace málo. Standardní datové typy obvykle rezervují pro celé číslo 2, 4, 8 po sobě jdoucích bajtů. Závisí na platformě!! Např. v Unixu je `unsigned int` v jazyce C 4-bajtové celé číslo, v MS DOS 2-bajtové.

Příklad: 4-bajtové (32 bitové) neznaménkové celé číslo 37 je zobrazeno jako

00000000	00000000	00000000	00100101

Rozsah 32 bitových neznaménkových celých čísel je od 0

00000000	00000000	00000000	00000000

do $2^{32} - 1 = 4294967295$

11111111	11111111	11111111	11111111

Rozsah p-bitového neznaménkového celého čísla je $0, 1, \ldots, 2^p - 1$.

Pro demonstraci přetečení/podtečení (*overflow/underflow*) neznaménkových celých čísel vyzkoušejte programy `unsigover` a `unsigunder` [Sandu, 2001].

Znaménková (signed) Používají dvojkový doplněk (*two's complement*). p-bitové znaménkové celé číslo v rozsahu $-2^{p-1}, \ldots, -1, 0, 1, \ldots, 2^{p-1} - 1$ je reprezentováno nejmenším celým kladným číslem, s nímž je kongruentní modulo 2^p.

(číslo)$_{10}$	**(dvojkový doplněk)$_{10}$**	**(dvojkový doplněk)$_2$**
−2147483648	2147483648	10000000 00000000 00000000 00000000
−2147483647	2147483649	10000000 00000000 00000000 00000001
−2147483646	2147483650	10000000 00000000 00000000 00000010
⋮	⋮	⋮
−2	4294967294	11111111 11111111 11111111 11111110
−1	4294967295	11111111 11111111 11111111 11111111
0	0	00000000 00000000 00000000 00000000
1	1	00000000 00000000 00000000 00000001
2	2	00000000 00000000 00000000 00000010
⋮	⋮	⋮
2147483645	2147483645	01111111 11111111 11111111 11111101
2147483646	2147483646	01111111 11111111 11111111 11111110
2147483647	2147483647	01111111 11111111 11111111 11111111

Je-li k reprezentováno určitým bitovým vzorkem, pak $-(k+1)$ je reprezentováno inverzním bitovým vzorkem s $0 \to 1$, $1 \to 0$.

Důvod pro užívání dvojkového doplňku pro znaménková celá čísla: odčítání a operace se zápornými čísly jsou nahrazeny operacemi jen s kladnými čísly. Jiné způsoby (sign/magnitude a biased) viz [Sandu, 2001].

Pro demonstraci přetečení/podtečení znaménkových celých čísel vyzkoušejte programy `sigover` a `sigunder` [Sandu, 2001].

Pozor na přiřazování např. hodnoty neznaménkové celočíselné proměnné do znaménkové celočíselné proměnné, jako např. v tomto fragmentu:

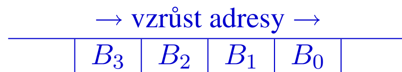
```
unsigned int u;
signed int s;
s=u;
```

Pro demonstraci konverze neznaménkové celočíselné proměnné do znaménkové vyzkoušejte program `unsig2sig`.

Celá čísla v paměti Jak je např. uloženo nějaké 32-bitové (ne)znaménkové celé číslo, třeba $(37)_{10} = (\underbrace{\boxed{00000000}}_{B_0}\underbrace{\boxed{00000000}}_{B_1}\underbrace{\boxed{00000000}}_{B_2}\underbrace{\boxed{00100101}}_{B_3})_2$?

Potřebujeme alokovat čtyři po sobě jdoucí bajty $B_0, \ldots, B_3$.

Ale pozor! Pořadí ukládání bajtů vícebajtových datových typů (tj. cokoli kromě znaků), tak řečená endianita (*endianity, byte sex*) závisí na systému. Např. IBM a Sun užívají adresování Big Endian, Intel adresování Little Endian.



Podle Gulliverových cest – Liliputáni se dělili na dvě skupiny, které se přely o to, na kterém konci se mají naklepávat vajíčka (Little Endians a Big Endians). Pro přístup potřebujeme znát nejnižší adresu (bajtu B_0 pro Big Endian, bajtu B_3 pro Little Endian) a velikost datového typu (zde 4 B).

Prostudujte zdrojový kód programu `byte_sex` [Kasprzak, 2004], jenž vypíše endianitu vašeho systému v době běhu.

Vyzkoušejte programy `uintaddr`/`sintaddr` (zobrazí počáteční a koncovou adresu zadaného (ne)znaménkového celého čísla).

Aritmetika počítačových celých čísel (s výjimkou celočíselného dělení) je, až na situace pře/podtečení, přesná, tj. numerický výsledek = matematický výsledek v $\mathbb{Z}$!! (Spočetná množina je ‚oseknuta' na konečnou.)

Reálná čísla $\mathbb{R}$ tvoří kontinuum (mohutnost $\aleph$), které musí v počítači být reprezentováno a vhodně aproximováno pomocí konečného počtu čísel. Lze reprezentovat jen čísla do určité meze, a v tomto rozsahu jen vhodně rozloženou konečnou podmnožinu $\mathbb{F} \subset \mathbb{R}$.

1.2.3. Reálná čísla

Obvykle reprezentována čísly v pohyblivé řádové čárce/tečce (*floating-point numbers*, FP). Historicky: čísla v pevné řádové čárce/tečce (*fixed-point numbers*) $\Rightarrow$ volba měřítka v rukou programátora (John von Neumann).

Motivace FP – hračkový model Pro jednoduchost volíme bázi $\beta = 10$. Každé číslo $x \in \mathbb{R}$ lze jednoznačně psát v normalizovaném tvaru

$$x = \underbrace{\sigma}_{\substack{\pm 1 \\ \text{znam., 1 b}}} \times \underbrace{m}_{\substack{1 \leq m < 10 \\ \text{mantisa} \in \mathbb{R},\ d_\mathrm{m}\ \text{č.}}} \times 10^{\overbrace{e}^{\text{exp.} \in \mathbb{Z},\ d_\mathrm{e}\ \text{č.}}} \tag{1.1}$$

Např. $109.375 = +1 \times 1.09375 \times 10^2$. Nerovnost omezující mantisu zajišťuje jedinou nenulovou číslici před její desetinnou tečkou (proto pohyblivá, $\Rightarrow$ jednoznačnost); nelze třeba $+1 \times 0.109375 \times 10^3$ nebo $+1 \times 0.00109375 \times 10^5$.

Mějme pro uložení 6 dekadických cifer: $\boxed{\sigma}\boxed{m_1 m_2 m_3}\boxed{e_1 e_2} = \boxed{+}\boxed{109}\boxed{02}$.
Zde $d_\mathrm{m} = 3$, $d_\mathrm{e} = 2$, $m_1 \neq 0$. Takových FP čísel je již konečně mnoho.

Lze v našem hračkovém modelu uložit $x = 0.000123$? Ne: | + | 000 | 00 |, úplná ztráta informace! Exponent je totiž nezáporný! Co s tím?

Použít bias: jsou-li jako exponent uloženy cifry e_1e_2, skutečný exponent je $e_1e_2 - 49$ (49 je bias). Skutečný exponent má rozsah -49 až 50, ale je uložen jako 00 až 99. Nemusíme pak ukládat znaménko exponentu, za cenu zmenšení rozsahu. Číslo $x = 0.000123 = +1 \times 1.23 \times 10^{-4}$ je tedy uloženo jako | + | 123 | 45 |. Obvykle se nejmenší/největší hodnota exponentu (zde 00 a 99) rezervuje pro reprezentaci speciální čísel jako $\pm\infty$, 0, subnormální čísla, `NaN` (viz sekci 1.2.3), tj. $e_1e_2 \in \{01, \ldots, 98\}$. Volba biasu je určena podmínkou $1/x_{\min} < x_{\max}$ a opačně.

Max. absolutní hodnota normalizovaného FP čísla: pro $m_1m_2m_3 = 999$, $e_1e_2 = 98$ je $x_{\max} = 9.99 \times 10^{49}$.

Min. absolutní hodnota normalizovaného FP čísla: pro $m_1m_2m_3 = 100$, $e_1e_2 = 01$ je $x_{\min} = 1.00 \times 10^{-48}$.

Nula je reprezentována $m_1m_2m_3 = 000$, $e_1e_2 = 00$. Obě (± 0) ekvivalentní.

Subnormální čísla: upustíme-li od normalizovaného tvaru ($m_1 \neq 0$), lze reprezentovat čísla menší než $x_{\min}$: $m_1m_2m_3 = 010$, $e_1e_2 = 00$ dává $x = 10^{-49}$, $m_1m_2m_3 = 001$, $e_1e_2 = 00$ dává $x = 10^{-50}$.

Subnormální čísla zlepšují přesnost v okolí 0: uvažujme fragment kódu `if (x!=y) z=1.0/(x-y);`, a $x = 1.02 \times 10^{-48}$, $y = 1.01 \times 10^{-48}$. Podmínka je v rámci normalizovaných FP splněna, ale rozdíl $x - y$ je pod rozsahem normalizovaných FP ($\Rightarrow$ dělení nulou!), nikoli však subnormálních FP, $x - y = 10^{-50}$.

Volíme-li bázi $\beta = 2$, máme normalizovaný tvar

$$x = \underbrace{\sigma}_{\substack{0 \equiv +, 1 \equiv - \\ \text{znam., 1 bit}}} \times \underbrace{m}_{\substack{1 \leq m < 2 \\ \text{mantisa} \in \mathbb{R},\, d_\mathrm{m} \text{ bitů}}} \times 2^{\overbrace{e}^{\text{exp.} \in \mathbb{Z},\, d_\mathrm{e} \text{ bitů}}} \tag{1.2}$$

Např. $(109.375)_{10} = (1101101.011)_2 = +1 \times 1.101101011 \times 2^6$. Mějme pro uložení 11 bitů: $d_\mathrm{m} = 6$, $d_\mathrm{e} = 4$, jeden bit pro znaménko. FP reprezentace je (zatím bez biasu) $\boxed{\sigma}\boxed{m_1 m_2 m_3 m_4 m_5 m_6}\boxed{e_1 e_2 e_3 e_4} = \boxed{0}\boxed{110110}\boxed{0110}$. Požadavek na normalizaci ve dvojkové soustavě ($m_1 \neq 0$) znamená $m_1 = 1$, je tedy zbytečné m_1 ukládat a místo toho uložit další, nejméně významný bit, navíc. Toto se nazývá technika skrytého bitu (*hidden bit technique*): FP reprezentace je pak $\boxed{0}\boxed{101101}\boxed{0110}$.

IEEE standard Standardizuje FP čísla, jak provádět aritmetické operace, a ošetření výjimek (*exceptions handling*). Vyvinut v 80 letech 20. století (IEEE, W. Kahan), nyní respektován všemi výrobci CPU.

ANSI/IEEE 754-1985 (pro $\beta = 2$), ANSI/IEEE 854-1987 (pro obecné β)
ISO standard: IEC 559: 1989

Číslo $x \in \mathbb{F}$ je binárně reprezentováno jako

$$x = \underbrace{\sigma}_{\text{1-bitové znam.}} \times \underbrace{m}_{d_\mathrm{m}\text{-bitová mantisa}} \times 2^{\overbrace{e}^{d_\mathrm{e}\text{-bitový exp.}} - \overbrace{E}^{\text{bias}}} \tag{1.3}$$

Jednoduchá přesnost (single precision)

Délka:	4 B (podléhají endianitě)	Znam.:	1 bit; 0 pro kladné, 1 pro záporné
Exponent:	$d_\mathrm{e} = 8$, bias $E = (127)_{10}$	Mant.:	$d_\mathrm{m} = 23$, technika skrytého bitu
$\lvert x_{\min,\max}\rvert$:	1.18×10^{-38}, 3.40×10^{38}	$\varepsilon, \varepsilon_-$:	1.1921×10^{-7}, 5.9605×10^{-8}

σ	$e_1e_2e_3\cdots e_8$	$m_1m_2m_3\cdots m_{23}$

$e_1e_2e_3\cdots e_8$	**Hodnota**	**Poznámka**
$(00000000)_2 = (0)_{10}$	$\pm(0.m_1\cdots m_{23})_2 \times 2^{-126}$ $\pm(0.m_1\cdots m_{23})_2 \times 2^{-126}$	$m_1 = \cdots = m_{23} = 0$: nula ($\pm 0$) aspoň jedno $m_i \neq 0$: subnormální
$(00000001)_2 = (1)_{10}$ ... $(01111111)_2 = (127)_{10}$ $(10000000)_2 = (128)_{10}$ ... $(11111110)_2 = (254)_{10}$	$\pm(1.m_1\cdots m_{23})_2 \times 2^{-126}$ ... $\pm(1.m_1\cdots m_{23})_2 \times 2^{0}$ $\pm(1.m_1\cdots m_{23})_2 \times 2^{1}$ ... $\pm(1.m_1\cdots m_{23})_2 \times 2^{127}$	normalizované číslo ... normalizované číslo normalizované číslo ... normalizované číslo
$(11111111)_2 = (255)_{10}$	‚`+inf`‘ a ‚`-inf`‘ ‚`NaN`‘	$m_1 = \cdots = m_{23} = 0$: $\pm\infty$ aspoň jedno $m_i \neq 0$: not a number

Dvojnásobná přesnost (double precision)

Délka:	8 B (podléhají endianitě)	Znam.:	1 bit; 0 pro kladné, 1 pro záp.
Exponent:	$d_\mathrm{e} = 11$, bias $E = (1023)_{10}$	Mant.:	$d_\mathrm{m} = 52$, technika skrytého bitu
$\lvert x_{\min,\max}\rvert$:	2.23×10^{-308}, 1.79×10^{308}	ε, ε_-:	2.22×10^{-16}, 1.11×10^{-16}

σ	$e_1e_2e_3\cdots e_{11}$	$m_1m_2m_3\cdots m_{52}$

$e_1e_2e_3\cdots e_{11}$	**Hodnota**	**Poznámka**
$(00000000000)_2 = (0)_{10}$	$\pm(0.m_1\cdots m_{52})_2 \times 2^{-1022}$ $\pm(0.m_1\cdots m_{52})_2 \times 2^{-1022}$	$m_1 = \cdots = m_{52} = 0$: nula ($\pm 0$) aspoň jedno $m_i \neq 0$: subnormální
$(00000000001)_2 = (1)_{10}$ ... $(01111111111)_2 = (1023)_{10}$ $(10000000000)_2 = (1024)_{10}$ ... $(11111111110)_2 = (2046)_{10}$	$\pm(1.m_1\cdots m_{52})_2 \times 2^{-1022}$ ... $\pm(1.m_1\cdots m_{52})_2 \times 2^{0}$ $\pm(1.m_1\cdots m_{52})_2 \times 2^{1}$ ... $\pm(1.m_1\cdots m_{52})_2 \times 2^{1023}$	normalizované číslo ... normalizované číslo normalizované číslo ... normalizované číslo
$(11111111111)_2 = (2047)_{10}$	‚`+inf`‘ a ‚`-inf`‘ ‚`NaN`‘	$m_1 = \cdots = m_{52} = 0$: $\pm\infty$ aspoň jedno $m_i \neq 0$: not a number

Jaké číslo reprezentuje v jedn. přes. |0|10000000|00110000000000000000000|? $\frac{19}{8} = 2^1 \times (1 + 2^{-3} + 2^{-4})$

Jak je zobrazeno $x = 5$ jako 4 B integer? |00000000000000000000000000000101|

Jak je zobrazeno $x = 5.0$ jako single prec.? $(1 + 2^{-2}) \times 2^2 \equiv$ |0|10000001|01000000000000000000000|

Rozšířená dvojnásobná přesnost (double-extended precision)

Doporučeno normou ANSI/IEEE 754-1985 (mají registry mikroprocesorů Intel).

Délka:	10 B (podléhají endianitě)	Znam.:	1 bit; 0 pro kladné, 1 pro záp.
Exponent:	$d_\mathrm{e} = 15$, bias $E = (16383)_{10}$	Mantisa:	$d_\mathrm{m} = 64$, bez skrytého bitu

σ	$e_1e_2e_3\cdots e_{15}$	$m_1m_2m_3\cdots m_{64}$

$e_1e_2e_3\cdots e_{15}$	**Hodnota**	**Poznámka**
$(000000000000000)_2 = (0)_{10}$	$\pm(m_1.m_2\cdots m_{64})_2 \times 2^{-16362}$	$m_1 = \cdots = m_{64} = 0$: nula ($\pm 0$)
$(000000000000001)_2 = (1)_{10}$	$\pm(m_1.m_2\cdots m_{64})_2 \times 2^{-16362}$	normalizované číslo
...	...	...
$(011111111111111)_2 = (16383)_{10}$	$\pm(m_1.m_2\cdots m_{64})_2 \times 2^{0}$	normalizované číslo
$(100000000000000)_2 = (16384)_{10}$	$\pm(m_1.m_2\cdots m_{64})_2 \times 2^{1}$	normalizované číslo
...	...	...
$(111111111111110)_2 = (32766)_{10}$	$\pm(m_1.m_2\cdots m_{64})_2 \times 2^{16363}$	normalizované číslo
$(111111111111111)_2 = (32767)_{10}$	‚`+inf`' a ‚`-inf`'	$m_1 = \cdots = m_{64} = 0$: $\pm\infty$
	‚`NaN`'	aspoň jedno $m_i \neq 0$: not a number

Užitečné pro dočasné uchování mezivýsledků (v registrech). Existuje také rozšířená jednoduchá přesnost (*single-extended precision*). Podrobně viz [Goldberg, 1991].

Co představuje |0|10000000|10000000000000000000000| jako single prec. FP číslo a co jako 4 B integer?
3.0 a 1077936128
Jak jsou v single prec. zobrazena čísla 2^{-126} a $(2 - 2^{-23}) \times 2^{127}$?
|0|00000001|00000000000000000000000| a |0|11111110|11111111111111111111111|

Strojové epsilon Jak probíhá např. sečtení dvou FP čísel? Pokud nejsou exponenty stejné, zvětší se exponent menšího čísla na hodnotu většího čísla, a zároveň se zmenšuje mantisa posunem cifer o potřebný počet míst doprava (tj. dělením β, *right-shifting*).

Např. v hračkovém modelu sčítáme $1 = 1.00 \times 10^0$ a $0.01 = 1.00 \times 10^{-2}$. Upravíme druhý sčítanec na 0.01×10^0 a sečteme: $1.00 \times 10^0 + 0.01 \times 10^0 = 1.01 \times 10^0$. OK!

Kdyby byl druhý sčítanec nejbližší menší FP číslo než 0.01, tj. 9.99×10^{-3}, dostaneme srovnáním jeho exponentu a posunem mantisy o tři pozice $1.00 \times 10^0 + 0.00999 \times 10^0 = 1.00 \times 10^0$! Ztráta signifikantní informace!

Číslo 0.01 je nejmenší kladné FP číslo ε, pro které platí $1 + \varepsilon > 1$, nebo jinak, $1 + \varepsilon$ je nejbližší FP číslo větší než 1. Nazývá se strojové epsilon (*machine epsilon*). V našem hračkovém modelu $\varepsilon = 0.01$.

Analogicky v binární IEEE aritmetice je strojové epsilon $\varepsilon = 2^{-d_\mathrm{m}}$ (při použití skrytého bitu) a $\varepsilon = 2^{1-d_\mathrm{m}}$ (bez použití skrytého bitu). (Lze definovat rovněž strojové epsilon ε_- pro odečítání od 1.)

Demonstrace pro single precision:

$$1.0 \equiv |0|01111111|00000000000000000000000|$$

$$\varepsilon \equiv |0|\underbrace{01101000}_{(104)_{10}}|00000000000000000000000|$$

Zvětšíme-li exponent ε o 1, musíme mantisu dělit dvojkou, čímž se skrytý bit 1 posune na první místo v mantise (teď už ale žádný další skrytý bit u ε není!):

$$\varepsilon = |0|\underbrace{01101001}_{(105)_{10}}|10000000000000000000000|$$

Dalším zvětšením exponentu o 1 a posunem mantisy doprava máme:

$$\varepsilon = |0|\underbrace{01101010}_{(106)_{10}}|01000000000000000000000|$$

Po $127 - 104 = 23 = d_{\mathrm{m}}$ takových operacích je mantisa srovnána a skrytý bit vytlačen na pravou krajní pozici:

$$\varepsilon = |0|\underbrace{01111111}_{(127)_{10}}|00000000000000000000001|$$

Vezmeme-li místo čísla 1.0 libovolné FP číslo x s obecným exponentem $e - E$ a mantisou m nejvýše $|111\dots10|$, bude příslušné strojové epsilon mít exponent menší o d_{m} a v mantise samé nuly, tj. nejbližší FP číslo bude o $\varepsilon \times 2^{e-E}$ větší. Pro $1 \le x < 2$ je $e - E = 0$, pro $2 \le x < 4$ je $e - E = 1$ (mezera mezi FP čísly 2ε), pro $0.5 \le x < 1$ je $e - E = -1$ (mezera mezi FP čísly $\varepsilon/2$), apod. Čím větší je FP číslo, tím větší je mezera mezi nimi; FP čísla mají vysokou hustotu v blízkosti nuly a čím více se od nuly vzdalujeme, tím se stávají řidšími – viz obrázek na následující straně.

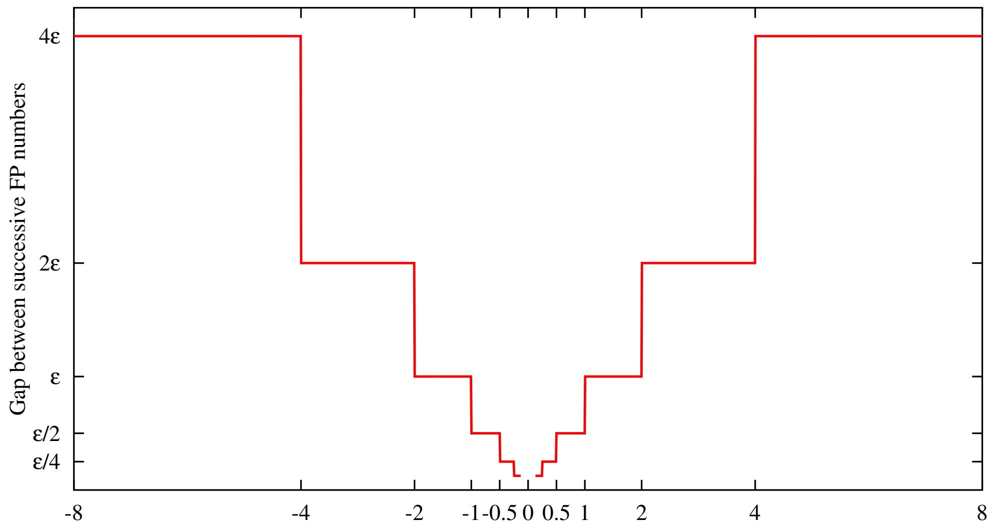
Jaká je mezera mezi 1024 a nejbližším větším IEEE single precision FP číslem?
$1024\varepsilon \approx 0.00012$

Uvažujte číslo $(0.1)_{10}$. Jaká je jeho single precision FP reprezentace?

$$= |0|\overbrace{01111011}^{(123)_{10}}|10011001100110011001100| =$$
$$(1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + \cdots) \times 2^{-4}$$

Má technika skrytého bitu v tomto případě za důsledek přesnější reprezentaci? Ne.

Program `epsilon` [Sandu, 2001] po zadání exponentu p sečte $1 + 2^{-p}$ a od výsledku odečte 1 (pro single i double precision). Je-li $2^{-p} < \varepsilon$, výsledek bude nula. Spusťte program pro $p \in \{20, \dots, 60\}$ a experimentálně zjistěte strojové epsilon pro jednoduchou i dvojnásobnou přesnost.



Grafické znázornění rozložení FP čísel. V každém intervalu $\langle 2^n, 2^{n+1})$ leží stejný počet FP čísel, takže mezera mezi nimi se zvětšuje a hustota zmenšuje.

Množina FP čísel: $\mathbb{F} = \{\pm 0,$ normální, subnormální, $\pm\infty\}$.

Pomocí programů machar_single a machar_double [Press et al., 1997a] diagnostikujte FP parametry vašeho systému. Položky na výstupu znamenají:

ibeta $\equiv \beta$, báze používané číselné soustavy.

it $\equiv d_\mathrm{m}+1$ (při použití skrytého bitu), d_m bez něj; počet cifer mantisy.

irnd Vrací $0,\ldots,5$ informující o způsobu zaokrouhlování při sčítání a jak je ošetřeno podtečení. 2, 5: váš systém zaokrouhluje podle IEEE standardu. 1, 4: váš systém provádí zaokrouhlování, ale ne podle IEEE standardu. 0, 3: nezaokrouhluje, usekává. 0, 1, 2: bez subnormálních čísel, 3, 4, 5: užívá subnormálních čísel.

ngrd Počet ‚guard digits'.

machep Nejmenší (nejzápornější) exponent $e-E$ báze β takový, že $1+\beta^{e-E}>1$.

negep Nejmenší (nejzápornější) exponent $e-E$ báze β takový, že $1-\beta^{e-E}<1$.

iexp $\equiv d_\mathrm{e}$, počet bitů exponentu.

minexp Nejmenší (nejzápornější) exponent $e-E$ báze β konzistentní s normalizovanými čísly.

maxexp Největší (kladný) exponent $e-E$ báze β.

eps $\equiv \varepsilon$, strojové epsilon.

epsneg $\equiv \varepsilon_-$, strojové epsilon ε_-.

xmin $\equiv \beta^{\mathtt{minexp}}$, nejmenší kladné normalizované FP číslo.

xmax $\equiv (1-\varepsilon_-)\beta^{\mathtt{maxexp}}$, největší normalizované FP číslo.

Zaokrouhlování Není-li $x \in \mathbb{R}$ a zároveň $x \notin \mathbb{F}$, musíme je reprezentovat vhodným FP číslem – zaokrouhlování (*rounding*). ANSI/IEEE definuje čtyři typy zaokrouhlování: nahoru, dolů, k nule, k nejbližšímu FP číslu.

Nahoru/dolů (up or down) Označme x_- (x_+) nejbližší menší nebo rovné (větší nebo rovné) FP číslo, a definujme

$$[x] \equiv \begin{cases} x_- & \text{pro zaokrouhlení dolů} \\ x_+ & \text{pro zaokrouhlení nahoru} \end{cases} \qquad [0] \equiv 0 \tag{1.4}$$

Zaokrouhlení reprezentuje určitou zaokrouhlovací chybu (*roundoff error*). Relativní zaokrouhlovací chybu δ definujeme

$$[x] = x(1+\delta) \tag{1.5}$$

Protože $|[x] - x| \leq |x_+ - x_-|$ a zároveň $x \geq x_-$, je

$$|\delta| \leq \frac{|x_+ - x_-|}{|x_-|} = \frac{\varepsilon 2^{e-E}}{m 2^{e-E}} \leq \varepsilon \tag{1.6}$$

Nalezněte v našem hračkovém FP modelu příklad čísla takového, že při zaokrouhlení dolů $\delta \approx \varepsilon$.
$\cdots 6666\dot{}0 = x$

K nule (chopping, toward zero) Zahození (odseknutí) bitů nenulových které se nevejdou do mantisy. Zřejmě $0 \le x_- \le x$ pro kladná x a $x \le x_+ \le 0$ pro záporná x. Zaokrouhlené číslo není dále od nuly než zaokrouhlované číslo. Zaokrouhlovací chyba

$$-\varepsilon \le \delta_{\text{chopping}} \le 0 \tag{1.7}$$

K nejbližšímu FP číslu (to nearest) Používá většina procesorů; pod zaokrouhlováním (*rounding*) budeme dále rozumět tento způsob.

$$[x] \equiv \begin{cases} x_- & \text{je-li} \quad x_- \le x < \frac{1}{2}(x_+ - x_-) \\ x_+ & \text{je-li} \quad x_+ \ge x > \frac{1}{2}(x_+ - x_-) \end{cases} \tag{1.8}$$

Zřejmě

$$-\frac{1}{2}\varepsilon \le \delta_{\text{rounding}} \le \frac{1}{2}\varepsilon \tag{1.9}$$

Chyby mají obě znaménka, takže se pravděpodobně zruší a nebudou se akumulovat jako u zaokrouhlení k nule. Je-li $x = \frac{1}{2}(x_+ + x_-)$, IEEE standard vyžaduje vybrat zaokrouhlení se sudým (nulovým) posledním bitem. Např. pro šestibitovou mantisu $x = 1.0000001$ může být zaokrouhleno na $x_- = 1.000000$ nebo na $x_+ = 1.000001$; v tomto případě x_-. To garantuje polovinu zaokrouhlení nahoru a polovinu dolů. Podrobně viz [Goldberg, 1991, Práger a Sýkorová, 2004].

FP čísla jsou přesná až na faktor $(1 \pm \varepsilon/2)$. Pro jednoduchou přesnost to znamená 1 ± 10^{-7}, tj. cca 7 desetinných míst pro čísla řádu jednotek, u čísel řádu 10^7 jsou nepřesností ‚postiženy' jednotky, u čísel menších než 1 je nepřesností ‚postižena' sedmá cifra po první nenulové číslici.

Podobně pro dvojnásobnou přesnost je přesných cca 16 desetinných míst pro čísla řádu jednotek.

Jaké jsou v IEEE single precision zaokrouhlené hodnoty těchto čísel:

$x = 4 + 2^{-20}$ — $x = 2^2(1 + 2^{-22})$, je FP číslo, $[x] = x$

$x = 8 + 2^{-20}$ — $x = 2^3(1 + 2^{-23})$, je FP číslo, $[x] = x$

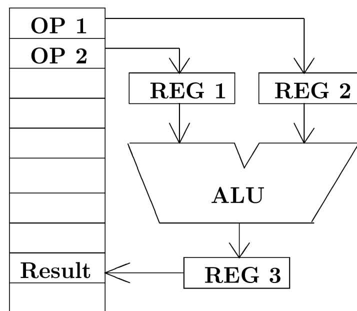
$x = 16 + 2^{-20}$ — $x = 2^4(1 + 2^{-24})$, není FP, $[x] = x_- = 2^4$, $x_+ = 2^4(1 + 2^{-23})$, (sudý posl. bit)

$x = 32 + 2^{-20}$ — $x = 2^5(1 + 2^{-25})$, není FP, x je blíže k x_-, takže $[x] = x_-$

Vyzkoušejte program `round` s čísly $1.0, 2.0, 1.1, 0.5, 0.0625, 0.6, 0.62$ a podle vlastního výběru. Vysvětlete získané výsledky.

1.3. Aritmetické operace

Hodnoty operandů jsou načteny z paměti do registrů CPU, Arithmetic and Logic Unit (ALU) provede operaci, výsledek vrátí do třetího registru, poté je hodnota uložena do paměti.



Operandy: FP čísla. Ale výsledek aritmetické operace nemusí být FP číslo! Bude zaokrouhleno ⇒ výsledek aritmetické operace je obecně poškozen zaokrouhlovací chybou.

V našem hračkovém modelu mějme dvě FP čísla $a = 97.2 = 9.72 \times 10^1$ a $b = 6.43 = 6.43 \times 10^0$. Mantisa druhého se posune tak, aby se exponenty rovnaly.

$$\begin{array}{rl} 9.72 & \times 10^1 \\ 0.643 & \times 10^1 \\ \hline 10.363 & \times 10^1 \end{array}$$

Výsledek 1.0363×10^2 není FP číslo! Bude zaokrouhlen na $1.04 \times 10^2 = 104$. Matematický výsledek se liší od numerického! Označme FP aritmetické operace $\oplus, \ominus, \otimes, \oslash$; obecně např. $a \oplus b \neq a + b$.

FP sčítání není asociativní! Pro $c = 0.999 = 9.99 \times 10^{-1}$ je $(a \oplus b) \oplus c = 1.04 \times 10^2 = 104$, ale $a \oplus (b \oplus c) = 1.05 \times 10^2 = 105$.

??? Ukažte, že FP sčítání je komutativní.

1.3.1. IEEE aritmetika

Nejprve exaktně, pak zaokrouhlit na nejbližší:

$$a \oplus b \equiv [a + b] \quad a \ominus b \equiv [a - b] \quad a \otimes b \equiv [a \times b] \quad a \oslash b \equiv [a/b] \tag{1.10}$$

Totéž platí pro druhou odmocninu, zbytek, a konverzi mezi celočíselným a FP typem. Operace prováděné tímto způsobem se nazývají exaktně zaokrouhlené (*exactly rounded*) nebo správně zaokrouhlené (*correctly rounded*).

Hardware kompatibilní s IEEE FP aritmetikou zlepšuje portabilitu programů.

Požadavek na exaktně zaokrouhlené operace se jeví jako přirozený, ale je obtížné hardwarově jej implementovat. Důvodem je, že pro implementaci korektní IEEE aritmetiky potřebujeme dodatečné hardwarové zdroje (registry dostatečné šířky).

Truncation error – na rozdíl od zaokrouhlovací chyby plně pod kontrolou programátora.

1.3.2. Speciální aritmetické operace

Znaménková nula Obě legální, ale ne zcela ekvivalentní. Je-li $x = +0$, $y = -0$ (ve smyslu FP reprezentace single, double a extended double), pak `(x==y)` vrací logickou pravdu.

Konzistence s oběma nekonečny: podle IEEE $1/(+0) = +\infty$ a $1/(-0) = -\infty$. Kdyby byla jen jedna nula a $1/0 = +\infty$, pak $1/(1/(-\infty)) = 1/0 = +\infty$.

Např. pro $\mathrm{tg}(\pi/2 - x)$ můžeme konzistentně definovat funkční hodnotu pro $x = \pm 0$ jako $\mp\infty$.

Nevýhoda: pro $x = +0$, $y = -0$ sice `(x==y)` vrací logickou pravdu, ale `(1/x==1/y)` logickou nepravdu.

Porovnání

`(a<b).OR.(a==b).OR.(a>b)`	True, jsou-li $a, b \in \mathbb{F}$. False, je-li jedno `NaN`.
`+0==-0`	True
`+inf==-inf`	False

Operace s nekonečnem Nekonečna poskytují možnost pokračovat ve výpočtu, objeví-li se přetečení rozsahu normalizovaných FP čísel.

$$a/\infty = \begin{cases} 0 & a \text{ konečné FP} \\ \texttt{NaN} & a = \infty \end{cases}$$

$$a \times \infty = \begin{cases} +\infty & a > 0 \\ -\infty & a < 0 \\ \texttt{NaN} & a = 0 \end{cases}$$

$$\pm\infty + a = \begin{cases} \pm\infty & a \text{ konečné FP} \\ \pm\infty & a = \pm\infty \\ \texttt{NaN} & a = \mp\infty \end{cases}$$

Operace s NaN Jakákoli operace, jejímž aspoň jedním operandem je `NaN`, má výsledek `NaN`. Navíc je `NaN` výsledkem: $\infty + (-\infty)$, $0 \times \infty$, $0/0$, ∞/∞, $\sqrt{-|x|}$, $x \bmod 0$, $\infty \bmod x$.

1.3.3. Výjimky, návěští, zachycování

(*exceptions, flags, exception trapping*)

IEEE definuje 5 výjimek: dělení nulou, přetečení (*overflow*), podtečení (*underflow*), neplatná operace (*invalid operation*), nepřesná operace (*inexact operation*).

Trap handlers jsou užitečné pro zpětnou kompatibilitu programů.

Dělení nulou Je-li $a \in \mathbb{F}$, IEEE vyžaduje:

$$a/0.0 = \begin{cases} +\infty & \text{je-li} \quad a > 0 \\ -\infty & \text{je-li} \quad a < 0 \\ \texttt{NaN} & \text{je-li} \quad a = 0 \end{cases} \tag{1.11}$$

Přetečení Když je výsledek aritmetické operace konečný, ale větší co do velikosti než největší reprezentovatelné FP číslo. Standardní IEEE ošetření: výsledek $\pm\infty$ (round to nearest) nebo největší reprezentovatelné FP číslo (round toward 0). Některé kompilátory přetečení zachytí a ukončí vykonávání programu s chybovou hláškou.

Příklad eliminace přetečení: [Press et al., 1997a] poskytuje knihovnu `complex.c`, v němž jsou definovány funkce pro práci s komplexními čísly, mj. funkci `Cabs` pro výpočet absolutní hodnoty. Pro datový typ `fcomplex` definovaný jako

```
typedef struct FCOMPLEX {float r,i;} fcomplex;
```

Chybná implementace:

```
float Cabs(fcomplex z)
{
    return sqrt(z.r*z.r+z.i*z.i);
}
```

Zde mohou kvadráty přetéci! Špatně!

Tento výpočet nepřeteče! Tak to má být!

Správná implementace:

```
float Cabs(fcomplex z)
{
    float x,y,ans,temp;
    x=fabs(z.r);
    y=fabs(z.i);
    if (x == 0.0)
        ans=y;
    else if (y == 0.0)
        ans=x;
    else if (x > y) {
        temp=y/x;
        ans=x*sqrt(1.0+temp*temp);
    } else {
        temp=x/y;
        ans=y*sqrt(1.0+temp*temp);
    }
    return ans;
}
```

Podtečení Když je výsledek aritmetické operace menší než nejmenší normalizované FP číslo. Podle IEEE je výsledek subnormální číslo (tzv. postupné podtečení, *gradual underflow*) nebo 0, je-li výsledek dostatečně malý. Subnormální čísla mají menší přesnost než normalizovaná, takže vedou ke ztrátě přesnosti. To je ale lepší než bez jejich použití.

Příklad ztráty přesnosti: v našem hračkovém modelu $x = 1.99 \times 10^{-40}$, $y = 1.00 \times 10^{-11}$, $z = 1.00 \times 10^{11}$ spočtěme $(x \times y) \times z$. Matematický výsledek je roven $t_{\text{exact}} = x$. Podle teorie zaokrouhlovacích chyb očekáváme

$$t_{\text{FP}} = (x \otimes y) \otimes z = (1 + \delta) t_{\text{exact}}, \qquad |\delta| \approx \varepsilon \tag{1.12}$$

(Pro každé FP násobení je relativní chyba $|\delta_{\otimes}| \leq \varepsilon/2$.) V hračkovém modelu je $\varepsilon = 0.01$, takže očekáváme

$$t_{\text{FP}} \in \langle (1 - 2\varepsilon) t_{\text{exact}}, (1 + 2\varepsilon) t_{\text{exact}} \rangle = \langle 1.98 \times 10^{-40}, 2.00 \times 10^{-40} \rangle \tag{1.13}$$

Avšak součin $x \otimes y = 1.99 \times 10^{-51}$ podtéká, a má v subnormálních číslech reprezentaci 0.01×10^{-49}, což vynásobeno se z dá $t_{\text{FP}} = 1.00 \times 10^{-40}$. Relativní chyba je pak $|\delta_{\text{subnorm}}| = 0.99 = 99\varepsilon$.

Neplatná operace Výsledkem je `NaN`.
Nepřesná operace Výsledkem je hodnota zaokrouhlená podle IEEE.

Vyzkoušejte program `messy-c`.

1.3.4. Systémové aspekty

Design počítačových systémů vyžaduje hluboké znalosti FP čísel. Moderní procesory mají speciální FP instrukce, kompilátory musí generovat takové FP instrukce, a operační systémy musí ošetřit výjimky generované FP instrukcemi.

Zde uvedeme pouze vybrané aspekty, podrobná diskuse je uvedena např. v [Goldberg, 1991, Sandu, 2001].

Optimalizace Následující kódy jsou matematicky ekvivalentní:

eps1single

```
#include <stdio.h>
int main(void)
{
    float eps=1.0;
    do {
        eps /= 2.0;
    } while (1.0+eps>1.0);
    printf("\n%.12e\n",eps);
    return 0;
}
```

eps2single

```
#include <stdio.h>
int main(void)
{
    float eps=1.0;
    do {
        eps /= 2.0;
    } while (eps>0.0);
    printf("\n%.12e\n",eps);
    return 0;
}
```

eps3single

```
#include <stdio.h>
int main(void)
{
    float eps=1.0,x;
    do {
        eps /= 2.0; x=1.0+eps;
    } while (x>1.0);
    printf("\n%.12e\n",eps);
    return 0;
}
```

Spouštějte demonstrační programy `epsXsingle`, `X = 1,2,3` a optimalizovanou verzi `eps3single-optcpu`, a pokuste se vysvětlit výsledky. (Double varianty `epsXdouble` a `eps3double-optcpu` se liší jen záměnou `float` za `double`.)

1.3.5. Dlouhé sumace

Problém s dlouhými sumacemi: každá individuální sumace vnáší do částečného součtu určitou chybu, takže celková suma může být značně nepřesná.

Položíme-li v sumě $\sum_{j=1}^{n} x_j$: $s_1 = (1+\delta_1)x_1$, $s_2 = (1+\delta_2)(s_1+x_2) = (1+\delta_2)[(1+\delta_1)x_1 + x_2]$, $s_3 = (1+\delta_3)(s_2+x_3) = (1+\delta_3)\{(1+\delta_2)[(1+\delta_1)x_1+x_2]+x_3\} = (1+\delta_1+\delta_2+\delta_3)x_1 + (1+\delta_2+\delta_3)x_2 + (1+\delta_3)x_3 + \mathcal{O}(\varepsilon^2)$, atd.

- Výpočet ve vyšší přesnosti (např. žádáme-li s_n v single, počítáme v double). Problém nastává, žádáme-li výsledek v double.
- Vhodně seřadit členy – zřejmě je výhodnější seřadit sčítance vzestupně.
- Použít Kahanovu sumační formuli [Goldberg, 1991]:

```
sum=x(1)
c=0.0
DO j=2,n
  y=x(j)-c
  t=sum+y
  c=(t-sum)-y
  sum=t
END DO
```

Pak vypočítaná suma

$$s_n = \sum_{j=1}^{n} x_j(1+\delta_j) + \mathcal{O}(n\varepsilon^2)\sum_{j=1}^{n}|x_j| \quad (1.14)$$

kde $|\delta_j| \le 2\varepsilon$. Důkaz viz [Goldberg, 1991].

Uvažujme sumu ($b \in \mathbb{N}$)

$$s = 1 + \underbrace{\frac{1}{b} + \ldots + \frac{1}{b}}_{b \text{ členů}} + \underbrace{\frac{1}{b^2} + \ldots + \frac{1}{b^2}}_{b^2 \text{ členů}} + \ldots + \underbrace{\frac{1}{b^{p_{\max}}} + \ldots + \frac{1}{b^{p_{\max}}}}_{b^{p_{\max}} \text{ členů}} \quad (1.15)$$

Každý ze členů ve svorkách je roven jedné, exaktní matematická suma je proto $p_{\max} + 1$. Počet sčítanců v sumě je

$$1 + b + b^2 + \ldots + b^{p_{\max}} = \frac{b^{p_{\max}} - 1}{b - 1} \quad (1.16)$$

Vyzkoušejte program longsum, který vykonává sumaci (1.15) pro $b = 10$, $p_{\max} = 7$ třemi způsoby: v sestupném pořadí, ve vzestupném pořadí a s použitím Kahanovy sumační formule. Jak se chovají optimalizované verze longsum-01 a longsum-02?

Typický výstup v single precision vypadá (GNU/Linux 2.4.20, GCC 3.3 20030226):

```
basis = 10,  maximum_power = 7
number_of_terms_in_the_sum = 11111111
exact:      8.00000000,  rel_error =   0.00000000 %
sort down: 6.95631695,  rel_error = -13.04603815 %
sort up:    8.01876831,  rel_error =   0.23460388 %
Kahan:      8.00000000,  rel_error =   0.00000000 %
```

U optimalizované verze je poslední řádek
`Kahan: 6.95631695, rel_error = -13.0460377 %` Proč?

1.3.6. Patologie, nástrahy a pasti

Konverze mezi binárním a dekadickým formátem

Vyzkoušejte program storeprt, jehož první část pouze uloží číslo $x = 1.0 \times 10^{-4}$ a poté jej vytiskne na terminál. Vysvětlete pozorovanou ‚anomálii'.

Porovnávání FP čísel

Vyzkoušejte opět program storeprt, jehož druhá část se větví podle výsledku srovnání čísel $1.0 \times 10^8 \times x^2$ a 1.0. Vysvětlete, proč se skutečnost liší od očekávání.

Vyzkoušejte upravený program storeprt2, jenž má upravenou podmínku větvení. Proč tento program pracuje v souladu s očekáváním?

Vyzkoušejte programy quiz_cor a quiz_inc. Proč první z nich vybere ‚správnou' a druhý ‚nesprávnou' větev?

Zvláštní konverze Někdy je nepřesnost v FP přenesena prostřednictvím konverzí do celočíselných typů.

Vyzkoušejte program `convers` a jeho optimalizovanou verzi `convers-optcpu`. Vysvětlete výsledky.

Jiný problém nastává při konverzi single do double precision. Taková konverze nezlepšuje přesnost – bity registru jsou ‚vycpány‘ nulami a číslo zůstává jinak stejné jako v single.

Ověřte si uvedené tvrzení pomocí programu `sing2dbl`.

Operandy v paměti a v registrech Je-li tentýž výpočet prováděn s operandy v registru (FP registry CPU Pentium mají rozšířenou dvojnásobnou přesnost, 80 bitů) a s operandy v paměti, může se výsledek lišit.

Tento jev demonstrujte pomocí programu `mem_reg-00` a jeho optimalizované verzi (stupeň -O2) `mem_reg-02`.

Neplatné číslice Odečteme-li v single precision $100000.1 - 100000.0$, očekáváme výsledek 0.1.

Jaký výsledek poskytne program `insignid`? Jak jej vysvětlíte?

Single precision zaručuje 7 decimálních číslic, a výše uvedené odečítání zruší nejvýznamnějších 6; výsledek obsahuje jen jednu. Připojené ‚smetí' jsou nesignifikantní číslice, které pocházejí z nepřesné binární reprezentace operandů.

Na pořadí operací záleží! Matematicky ekvivalentní výrazy mohou dávat odlišné výsledky podle toho, v jakém pořadí se provádí vyčíslování v FP.

Demonstrujte tuto skutečnost pomocí programu `ordofop`, případně jeho optimalizované verze `ordofop-optcpu`.

Proč dává optimalizovaná verze jiné výsledky než neoptimalizovaná?

Katastrofické vyrušení (*catastrophic cancellation*, *loss-of-significance errors*). Odečítáme-li velmi blízká čísla, nejvýznamnější číslice operandů se rovnají a vzájemně se zruší. Pokud jsou operandy poškozeny zaokrouhlovací chybou, může dojít k úplnému zamaskování výsledku.

Řešíme-li kvadratickou rovnici

$$ax^2 + bx + c = 0 \tag{1.17}$$

s takovými koeficienty, že $4ac \ll b^2$, bude jeden z kořenů

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.18}$$

podléhat katastrofickému vyrušení. Pro $b > 0$ to bude kořen x_1 odpovídající $+$. V takovém případě rozšíříme rovnici (1.18) výrazem $-b - \sqrt{b^2 - 4ac}$:

$$x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{1.19}$$

Vyzkoušejte program `cancerr`, který řeší kvadratickou rovnici s koeficienty $a = 1.0$, $b = c = 1.0 \times 10^8$. Podmínka $4ac \ll b^2$ je splněna a kořen (1.18) odpovídající znaménku $+$ trpí katastrofickým vyrušením. Použitím (1.19) dostaneme podstatně přesnější výsledek.

1.3.7. Stabilita

Někdy je zaokrouhlovací chyba ‚vmixována'do výpočtu v jeho rané fázi a zvětšuje se tak dlouho, až zcela překryje správný výsledek.

Příklad [Press et al., 1997a]: Kladným řešením kvadratické rovnice

$$x^2 = 1 - x \tag{1.20}$$

je ‚Zlatý řez' (*Golden Mean*) $\varphi = \frac{1}{2}(\sqrt{5} - 1) \approx 0.61803398875$. Celočíselné mocniny φ^n lze počítat pomocí $\varphi^{n+1} = \varphi\varphi^n$, $n = 0, 1, \ldots$, nebo z rekurentní formule

$$\varphi^{n+1} = \varphi^{n-1} - \varphi^n, \quad \varphi^0 = 1, \quad \varphi^1 = 0.61803398875, \quad n = 1, 2, \ldots \tag{1.21}$$

jen pomocí odčítání. (Důkaz (1.21): $\varphi^{n+1} - \varphi^{n-1} = \varphi^{n-1}(\varphi^2 - 1) = -\varphi^n$, protože podle (1.20) je výraz v závorce roven $-\varphi$.)

Rovnice (1.20) má ještě jedno řešení: $\psi = -\frac{1}{2}(\sqrt{5} + 1) \approx -1.61803398875$. Toto řešení splňuje stejnou rekurzivní relaci (1.21). Jelikož (1.21) je lineární, a $|\psi| > 1$, malá příměs zavlečená zaokrouhlovací chybou bude exponenciálně růst.

Tento příklad nestability je implementován v programech `unstable_single` a `unstable_double` (varianty pro jednoduchou a dvojnásobnou přesnost).

1.4. Havárie způsobené chybným použitím FP čísel

- Kolaps řízení obranného protiraketového systému Patriot ve válce v Zálivu r. 1991. Systém fungoval efektivně, ale v jednom případě selhal: Identifikace střel Scud probíhala tak, že radarový systém po zachycení podezřelého objektu předpověděl jeho budoucí polohu po uplynutí nějakého časového intervalu, a jestliže v tom místě v příslušnou dobu objekt zjistil, dal signál k jeho zničení. Právě v předpovědi nové polohy docházelo k nepřesnostem, a to tím větším, čím déle systém pracoval. Po 20 hodinách byla chyba v určení polohy zhruba 137 metrů, po 100 hodinách 687 metrů. Příčina: vnitřní hodiny počítače uchovávaly čas v celých násobcích 0.1 sec a program je převáděl na FP číslo v sekundách. Při tom se dekadické číslo 0.1, které má v binární soustavě nekonečný rozvoj 0.0001100110011 . . ., zaokrouhlovalo.
- Převod FP čísla na celé číslo způsobil destrukci rakety Ariane 5 v ceně $\sim$ 1 G$ v červnu 1996. Příčina: třicet sedm vteřin po startu se program pokusil konvertovat horizontální komponentu rychlosti rakety z double precision na krátké (16bitové) celé číslo, které přeteklo, počítačový program ohlásil neplatnou operaci a řídicí systém provedl samodestrukci rakety.

Více viz [Práger a Sýkorová, 2004] a
http://www.fas.org/starwars/gao/im92026.htm
http://www.siam.org/siamnews/general/patriot.htm
http://www.ima.umn.edu/~arnold/disasters/

Shrnutí kapitoly: První kapitola měla úvodní charakter a seznámila vás s principy využívaných ve výpočetní technice při reprezentaci a manipulaci s čísly. Naučili jste se rozumět problémům celočíselné aritmetiky a hlavně aritmetiky reálných čísel. Seznámili jste se s důležitými pojmy jako jsou FP čísla, normalizovaná a subnormální čísla; norma ANSI/IEEE 754-1985; jednoduchá a dvojnásobná přesnost, strojové epsilon, zaokrouhlování výjimky a stabilita výpočtu

Další zdroje:

- Výklad nevyžadující zvláštní předběžné znalosti je uveden např. v [Práger a Sýkorová, 2004].
- Úvod do Fortranu 95 a numerických metod [Sandu, 2001]. Obsahuje kapitolu o reprezentaci čísel a počítačové aritmetice. Dostupné na http://www.cs.mtu.edu/~asandu/Courses/CS2911/fortran_notes/main.html
- Podrobnou analýzu s mnoha odkazy na původní prameny podává např. článek [Goldberg, 1991].
- Sekce 1.3 (1.2) v [Press et al., 1997a]. Pro jazyk C dostupné na http://www.library.cornell.edu/nr/bookcpdf/c1-3.pdf, pro Fortran na http://www.library.cornell.edu/nr/bookfpdf/c1-2.pdf.

7. Poznámky

7.1. Proč nesignalizuje první bit mantisy speciální číslo?

V případě báze $\beta = 10$ by mohla být použit pro signalizaci nuly nebo subnormálních čísel nulovost prvního bitu mantisy. Tuto techniku však nelze použít v případě $\beta = 2$ kvůli skrývání prvního bitu normalizovaných čísel, který může být roven pouze 1. Příznakem speciálního čísla je vyhrazená speciální hodnota exponentu, tj. $00\ldots0$ nebo $11\ldots1$.

7.2. Strojové ε_-

Strojové ε_- je definováno jako nejmenší kladné FP číslo, pro které platí $1 - \varepsilon_- < 1$, nebo jinak, $1 - \varepsilon_-$ je nejbližší FP číslo menší než 1. V binární IEEE aritmetice je strojové epsilon $\varepsilon_- = 2^{-d_\mathrm{m}-1} = \varepsilon/2$ (při použití skrytého bitu) a $\varepsilon_- = 2^{-d_\mathrm{m}} = \varepsilon/2$ (bez použití skrytého bitu).

Analogicky strojové epsilon na intervalu $\langle 2^n, 2^{n+1})$ je $2^n\varepsilon$, speciálně pro $n = -1$ je na intervalu $\langle 1/2, 1)$ $\varepsilon_- = \varepsilon/2$.

9. Popisy a zdrojové texty příkladů

Krátké zdrojové texty jsou zobrazeny přímo; předchází jim stručný popis a ‚neonový box' s názvem, z něhož lze program kliknutím spustit, např. char2ascii níže. Někdy je vypuštěn popis, spouští se stejným způsobem, např. storeprt. Dlouhé zdrojové texty nejsou zobrazeny, lze je prohlížet kliknutím na ‚neonový box' s názvem a spouštět kliknutím na symbol ➶ vlevo od názvu, např. longsum. Jakýkoli název v popisu v ‚neonové barvě' je prohlížecí link.

9.1. Ke kapitole 1

char2ascii Program zobrazí ASCII kód zadaného znaku v decimální, oktalové a hexadecimální soustavě.

```
#include <stdio.h>
int main(void)
{
    char a;
    printf("%s","\nPlease input character: ");
    scanf("%c",&a);
    printf("ASCII code of character \'%c\' is %3u (decimal)\n",a,a);
    printf("                                     %3o (octal)\n",a);
    printf("                                      %2X (hexadecimal)\n",a);
    return 0;
}
```

charaddr Program zobrazí adresu (v hexadecimálním tvaru), na kterou v době běhu umístil zadaný znak.

```
#include <stdio.h>
int main(void)
{
    char a;
    printf("%s","\nPlease input character: ");
    scanf("%c",&a);
    printf("Character \'%c\' is located at address %p\n",a,&a);
    return 0;
}
```

straddr Program zobrazí hexadecimální adresy prvního a posledního znaku zadaného řetězce. Je-li zadaný řetězec delší než 8 znaků, bude oříznut.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char a[16];
    unsigned int i;
    printf("%s","\nPls input string of chars (will be truncated to 8 chars): ");
    scanf("%8s",a);
    printf("String length is:          %u\n",(i=strlen(a)));
    printf("Address of 1st/last char:  %p / %p\n",a,a+i-1);
    return 0;
}
```

unsigover Program přičte zadané kladné celé číslo i k maximálnímu neznaménkovému celému číslu zmenšenému o 3. Pro $i > 3$ dojde k přetečení.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
    unsigned int k=UINT_MAX-3,i;
    printf("\nMaximum unsigned int is: m = %u\n",k+3);
    printf("The same - 3 is:         k = %u\n",k);
    printf("Input small pos. integer i = ");
    scanf("%u",&i);
    printf("Sum                  k + i = %u\n",k+i);
    return 0;
}
```

unsigunder Varianta předchozího příkladu: program odečte zadané kladné celé číslo i od neznaménkové 3. Pro $i > 3$ dojde k podtečení.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
    unsigned int k=3,i;
    printf("\nMinimum unsigned int is: m = %u\n",k-3);
    printf("The same + 3 is:         k = %u\n",k);
    printf("Input small pos. integer i = ");
    scanf("%u",&i);
    printf("Difference           k - i = %u\n",k-i);
    return 0;
}
```

sigover Program přičte zadané kladné celé číslo i k maximálnímu znaménkovému celému číslu zmenšenému o 3. Pro $i > 3$ dojde k přetečení.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
    signed int k=INT_MAX-3,i;
    printf("\nMaximum signed int is:   m = %d\n",k+3);
    printf("The same - 3 is:         k = %d\n",k);
    printf("Input small pos. integer i = ");
    scanf("%d",&i);
    printf("Sum                  k + i = %d\n",k+i);
    return 0;
}
```

sigunder Varianta předchozího příkladu: program odečte zadané kladné celé číslo i od minimálního znaménkového celého čísla zvětšeného o 3. Pro $i > 3$ dojde k podtečení.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
    signed int k=INT_MIN+3,i;
    printf("\nMinimum signed int is:   m = %d\n",k-3);
    printf("The same + 3 is:         k = %d\n",k);
    printf("Input small pos. integer i = ");
    scanf("%d",&i);
    printf("Difference           k - i = %d\n",k-i);
    return 0;
}
```

unsig2sig Program přiřadí hodnotu zadaného neznaménkového celého čísla v proměnné u do znaménkové celočíselné proměnné s. Pozorujte chování programu v závislosti na tom, jestli je v u hodnota nejvýše rovná nebo větší než maximální znaménkové celé číslo.

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
    unsigned int u;
    signed int s;
    printf("\nInput pos. integer le/gt %d: u = ",INT_MAX);
    scanf("%u",&u);
    s=u;
    printf("After conversion to signed:          s = %d\n",s);
    return 0;
}
```

byte_sex Vypíše endianitu systému v době běhu. Upraveno podle `bytesex.c`, © 2001 Jan „Yenya“ Kasprzak, ¡kas@fi.muni.cz¿).

```
#include <stdio.h>
int main(void)
{
    union {int i; char c[sizeof(int)];} u;
    u.i = 0; u.c[0] = 1;
    puts(u.i == 1 ? "\nLittle Endian" : "\nBig Endian");
    return 0;
}
```

uintaddr Program vypíše adresy, na něž v době běhu umístil neznaménkové celé číslo.

```
#include <stdio.h>
int main(void)
{
    unsigned int a;
    printf("%s","\nPlease input nonnegative number: ");
    scanf("%u",&a);
    printf("%u-bit number \'%u\' occupies\naddresses "
     "from %p to 0x%x\n",
     8*sizeof(a),a,&a,((unsigned int)(&a+1))-1);
    return 0;
}
```

sintaddr Program vypíše adresy, na něž v době běhu umístil znaménkové celé číslo.

```
#include <stdio.h>
int main(void)
{
    signed int a;
    printf("%s","\nPlease input number: ");
    scanf("%d",&a);
    printf("%u-bit number \'%d\' occupies\naddresses "
     "from %p to 0x%x\n",
     8*sizeof(a),a,&a,((unsigned int)(&a+1))-1);
    return 0;
}
```

epsilon Pro zadané kladné celé číslo p vypočte a zobrazí $(1.0 \oplus 2^{-p}) \ominus 1.0$ pro jednoduchou a dvojnásobnou přesnost. Pro dostatečně veliké p je $2^{-p} < \varepsilon$ a uvedený výraz bude nulový. Metodou pokusu a omylu umožňuje stanovit strojové epsilon pro obě přesnosti.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    float a;
    double b;
    int p;
    printf("\n%s","Please provide exponent: ");
    scanf("%d",&p);
    a = 1.0f + (float)pow(2.0,-p);
    printf("Single: %E\n",a-1.0);
    b = 1.0 + pow(2.0,-p);
    printf("Double: %E\n",b-1.0);
    return 0;
}
```

round Zadané číslo uloží do reálné proměnné v jednoduché přesnosti. Obsah proměnné pak zobrazí. Vyzkoušejte čísla 1.0, 2.0, 1.1, 0.5, 0.0625, 0.6, 0.62 a podle vlastního výběru a pozorujte zaokrouhlování.

```
#include <stdio.h>
int main(void)
{
    float x;
    printf("\n%s","Please provide a number: ");
    scanf("%f",&x);
    printf("Rounded to nearest FP:   %.12f\n",x);
    return 0;
}
```

messy-c Ilustrace některých aritmetických operací s výsledky $\pm\infty$ a NaN; verze v C.

```
#include <stdio.h>
int main(void)
{
    float a,b,c,d;
    a=0.0;
    b=-0.0;
    printf("\na = %f, b = %f\n",a,b);
    c=1.0/a;
    d=1.0/b;
    printf("1.0/a = %f,  1.0/b = %f\n",c,d);
    printf("1.0/a + 1.0/b   = %f\n",c+d);
    printf("1.0/a - 1.0/b   = %f\n",c-d);
    printf("1.0/b - 1.0/a   = %f\n",d-c);
    printf("(1.0/a)/(1.0/b) = %f\n",c/d);
    printf("(1.0/a)*(1.0/b) = %f\n",c*d);
    printf("a*(1.0/a)       = %f\n",a*c);
    return 0;
}
```

messy-f Ilustrace některých aritmetických operací s výsledky $\pm\infty$ a NaN; verze ve Fortranu 90.

```
PROGRAM messy
  IMPLICIT NONE
  REAL :: a,b,c,d
  a=0.0
  b=-0.0
  PRINT*
  PRINT '(a5,f9.6,a6,f9.6)', &
       " a = ", a, ", b = ", b
  c=1.0/a
  d=1.0/b
  PRINT '(a9,f9.6,a10,f10.6)', &
       " 1.0/a = ", c, ", 1.0/b = ", d
  PRINT*, "1.0/a + 1.0/b   = ", c+d
  PRINT*, "1.0/a - 1.0/b   = ", c-d
  PRINT*, "1.0/b - 1.0/a   = ", d-c
  PRINT*, "(1.0/a)/(1.0/b) = ", c/d
  PRINT*, "(1.0/a)*(1.0/b) = ", c*d
  PRINT*, "a*(1.0/a)       = ", a*c
  STOP
END PROGRAM messy
```

eps3single-optcpu Optimalizovaná verze (-O3) programu eps3single.

```
/* Code identical to eps3single.c. */
```

eps1double Verze programu eps1single ve dvojnásobné přesnosti.

```
/* Same as eps1single.c except of 'float' -> 'double'. */
```

eps2double Verze programu eps2single ve dvojnásobné přesnosti.

```
/* Same as eps2single.c except of 'float' -> 'double'. */
```

eps3double Verze programu eps3single ve dvojnásobné přesnosti.

```
/* Same as eps3single.c except of 'float' -> 'double'. */
```

eps3double-optcpu Optimalizovaná verze (-O3) programu eps3double.

```
/* Same as eps3single.c except of 'float' -> 'double'. */
```

machar_single Program pro diagnostiku FP parametrů systému v jednoduché přesnosti podle [Press et al., 1997a]. Význam položek výstupu je uveden v rámečku s popisem programu.

machar_double Program pro diagnostiku FP parametrů systému ve dvojnásobné přesnosti podle [Press et al., 1997a]. Význam položek výstupu je uveden v rámečku s popisem programu.

longsum Demonstrace dlouhé sumace při sestupném, vzestupném uspořádání a při použití Kahanovy sumační formule. Neoptimalizovaná verze (`-O0`).

longsum-O1 Demonstrace dlouhé sumace při sestupném, vzestupném uspořádání a při použití Kahanovy sumační formule. Stupeň optimalizace `-O1`.

longsum-O2 Demonstrace dlouhé sumace při sestupném, vzestupném uspořádání a při použití Kahanovy sumační formule. Stupeň optimalizace `-O2`.

storeprt

```
PROGRAM storeprt
  IMPLICIT NONE
  REAL :: x=1.0E-4
  PRINT*
  PRINT*, x
  IF ((1.0E+8 * x**2)==1.0) THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Incorrect'
  END IF
  STOP
END PROGRAM storeprt
```

storeprt2

```
PROGRAM storeprt2
  IMPLICIT NONE
  REAL :: x=1.0E-4, epsilon=1.0E-7, w
  PRINT*
  PRINT*, x
  w=1.0E+8 * x**2
  IF (ABS(w-1.0) .LE. 0.5*epsilon*(ABS(w)+ABS(1.0))) THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Incorrect'
  END IF
  STOP
END PROGRAM storeprt2
```

quiz_cor

```
PROGRAM quiz_cor
  IMPLICIT NONE
  REAL :: x=1.0/2.0
  PRINT*
  IF ((2.0*x) .EQ. 1.0) THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Incorrect'
  END IF
  STOP
END PROGRAM quiz_cor
```

quiz_inc

```
PROGRAM quiz_inc
  IMPLICIT NONE
  REAL :: x=1.0/3.0
  PRINT*
  IF ((3.0*x) .EQ. 1.0) THEN
     PRINT*, 'Correct'
  ELSE
     PRINT*, 'Incorrect'
  END IF
  STOP
END PROGRAM quiz_inc
```

convers

```
PROGRAM convers
  IMPLICIT NONE
  REAL :: x=1.0E-4
  INTEGER :: i
  i=10000*x
  PRINT*
  PRINT*, i
  STOP
END PROGRAM convers
```

convers-optcpu

```
PROGRAM convers
  IMPLICIT NONE
  REAL :: x=1.0E-4
  INTEGER :: i
  i=10000*x
  PRINT*
  PRINT*, i
  STOP
END PROGRAM convers
```

sing2dbl

```
#include <stdio.h>
int main(void)
{
    float x=1.234567f;
    double y,z=1.234567;
    y=x;
    printf("\nx = %.12f\n",x);
    printf("y = %.12f\n",y);
    printf("z = %.12f\n",z);
    return 0;
}
```

insignid

```
PROGRAM insignid
  IMPLICIT NONE
  REAL :: x=100000.0,y=100000.1,z
  z=y-x
  PRINT*
  PRINT*, 'z = ', z
  STOP
END PROGRAM insignid
```

mem_reg-O0

```
SUBROUTINE ratio(r,s,t)
  IMPLICIT NONE
  REAL :: r,s,t
  t=s/r
END SUBROUTINE ratio

SUBROUTINE subtr(r,s,t)
  IMPLICIT NONE
  REAL :: r,s,t
  t=s-r
END SUBROUTINE subtr

PROGRAM mem_reg
  IMPLICIT NONE
  REAL :: a,b,x,y,z,c,direct, bycall
  DATA a /3.0/, b /10.0/
  DATA x /3.0/, y /10.0/
  direct=(y/x)-(b/a)
  CALL ratio(x,y,z)
  CALL ratio(a,b,c)
  CALL subtr(c,z,bycall)
  PRINT*
  PRINT*, direct-bycall
  STOP
END PROGRAM mem_reg
```

mem_reg-O2

```
SUBROUTINE ratio(r,s,t)
  IMPLICIT NONE
  REAL :: r,s,t
  t=s/r
END SUBROUTINE ratio

SUBROUTINE subtr(r,s,t)
  IMPLICIT NONE
  REAL :: r,s,t
  t=s-r
END SUBROUTINE subtr

PROGRAM mem_reg
  IMPLICIT NONE
  REAL :: a,b,x,y,z,c,direct, bycall
  DATA a /3.0/, b /10.0/
  DATA x /3.0/, y /10.0/
  direct=(y/x)-(b/a)
  CALL ratio(x,y,z)
  CALL ratio(a,b,c)
  CALL subtr(c,z,bycall)
  PRINT*
  PRINT*, direct-bycall
  STOP
END PROGRAM mem_reg
```

ordofop

```
PROGRAM ordofop
  IMPLICIT NONE
  REAL :: x=1.23456789,y=4.56789123,z=9.999999,r1,r2
  r1=(x*y)/z
  r2=(1.0/z)*x*y
  PRINT*
  PRINT* 'r1 - r2 = ',r1-r2
  r1=(x*y)
  r1=r1/z
  r2=1.0/z
  r2=r2*x
  r2=r2*y
  PRINT* 'r1 - r2 = ',r1-r2
  STOP
END PROGRAM ordofop
```

ordofop-optcpu

```
PROGRAM ordofop
  IMPLICIT NONE
  REAL :: x=1.23456789,y=4.56789123,z=9.999999,r1,r2
  r1=(x*y)/z
  r2=(1.0/z)*x*y
  PRINT*
  PRINT* 'r1 - r2 = ',r1-r2
  r1=(x*y)
  r1=r1/z
  r2=1.0/z
  r2=r2*x
  r2=r2*y
  PRINT* 'r1 - r2 = ',r1-r2
  STOP
END PROGRAM ordofop
```

unstable_single

```
#include <stdio.h>
#include <math.h>
#define DBL 0
#if (DBL==1)
#define MAX 46
#define REAL double
#else /* DBL */
#define MAX 26
#define REAL float
#endif /* DBL */
int main(void)
{
    REAL g=(sqrt(5.0)-1.0)/2.0,m=g,s=g,prevs=1.0,ss;
    int i;
    printf("\n%s\n",
           " i     s_(i+1) = s_1 * s_i    s_(i+1) ="
           " s_(i-1) - s_i    frac.err.[%]");
    printf("%s\n",
           "--------------------------------------"
           "-----------------------------");
    for (i=0;i<=MAX;i++) {
  if (i==0)
      printf("%2d    %+.16f    %+.16f     %+9.2f\n",
       i,1.0,1.0,0.0);
  else {
      printf("%2d    %+.16f    %+.16f     %+9.2f\n",
       i,m,s,100.*(s-m)/m);
      m=m*g;
      ss=s;
      s=prevs-s;
      prevs=ss;
  }
    }
    return 0;
}
```

unstable_double

```
#include <stdio.h>
#include <math.h>
#define DBL 1
#if (DBL==1)
#define MAX 46
#define REAL double
#else /* DBL */
#define MAX 26
#define REAL float
#endif /* DBL */
int main(void)
{
    REAL g=(sqrt(5.0)-1.0)/2.0,m=g,s=g,prevs=1.0,ss;
    int i;
    printf("\n%s\n",
           " i     s_(i+1) = s_1 * s_i    s_(i+1) ="
           " s_(i-1) - s_i    frac.err.[%]");
    printf("%s\n",
           "--------------------------------------"
           "-----------------------------");
    for (i=0;i<=MAX;i++) {
  if (i==0)
      printf("%2d    %+.16f    %+.16f     %+9.2f\n",
                   i,1.0,1.0,0.0);
  else {
      printf("%2d    %+.16f    %+.16f     %+9.2f\n",
       i,m,s,100.*(s-m)/m);
      m=m*g;
      ss=s;
      s=prevs-s;
      prevs=ss;
  }
    }
    return 0;
}
```

cancerr

```
PROGRAM cancerr
  IMPLICIT NONE
  REAL :: a=1.0,b=1.0E+8,c=1.0E+8,d,x1,x2
  d=SQRT(b**2-4.0*a*c)
  x1=(-b-d)/(2.0*a) ! this is ok
  x2=(-b+d)/(2.0*a) ! suffers from cancellation
  PRINT*
  PRINT*, 'x1 = ', x1, ', x2 = ', x2
  x2=(2.0*c)/(-b-d) ! this is much better
  PRINT*, 'x1 = ', x1, ', x2 = ', x2
  STOP
END PROGRAM cancerr
```

Reference

[Abramowitz a Stegun, 1964] Abramowitz, M. a Stegun, I. A. (1964).
Handbook of Mathematical Functions, svazek 55 *Applied Mathematics Series*.
National Bureau of Standards, Washington.
Reprinted 1968 by Dover Publications, New York.

[Bratley et al., 1983] Bratley, P., Fox, B. L., a Schrage, E. L. (1983).
A Guide to Simulation.
Springer-Verlag, New York.

[Burrus et al., 1998] Burrus, C. S., Gopinath, R. A., a Guo, H. (1998).
Introduction to Wavelets and Wavelets Transforms. A Primer.
Prentice Hall, New Jersey.

[Dahlquist a Bjorck, 1974] Dahlquist, G. a Bjorck, A. (1974).
Numerical Methods.
Prentice Hall, Englewood Cliffs, NJ.

[Došlá et al., 2002] Došlá, Z., Plch, R., a Sojka, P. (2002).
Matematická analýza s programem Maple. 2. Nekonečné řady.
Masarykova univerzita v Brně, Brno.
CD-ROM.

[Forsythe et al., 1977] Forsythe, G. E., Malcolm, M. A., a Moler, C. B. (1977).
Computer Methods for Mathematical Computations.
Prentice-Hall, Englewood Cliffs, NJ.

[Frigo a Johnson, 0000] Frigo, M. a Johnson, S. G. (0000).
FFTW – the Fastest Fourier Transform in the West.

[Goldberg, 1991] Goldberg, D. (1991).
What every computer scientist should know about floating-point arithmetic.
ACM Comput. Surveys, 23(1):5–48.

[Grega et al., 1974] Grega, A., Kluvanec, D., a Rajčan, E. (1974).

Matematika pre fyzikov.
Slovenské pedagogické nakladateľstvo, Bratislava, 1. vydání.

[Hardy a Rogosinski, 1971] Hardy, G. H. a Rogosinski, W. W. (1971).
Fourierovy řady.
SNTL/Alfa, Praha, 1. vydání.

[Hledík, 2007] Hledík, S. (2007).
FoSpATOX — Fourier and Spectral Applications Tools/Toys for Experiments.
collections of programs for experiments in 1D spectral and wavelet analysis.

[Kasprzak, 2004] Kasprzak, J. (2004).
Unix.
http://uf.fpf.slu.cz/.

[Knuth, 1981] Knuth, D. E. (1981).
Seminumerical Algorithms, svazek 2 *The Art of Computer Programming*.
Addison-Wesley, Reading, MA, 2. vydání.

[Kopeček a Kučera, 1989] Kopeček, I. a Kučera, J. (1989).
Programátorské poklesky.
Mladá fronta – program Start, Praha.

[Metcalf a Reid, 1999] Metcalf, M. a Reid, J. (1999).
Fortran 90/95 Explained.
Oxford University Press, New York, 2nd vydání.

[Percival a Walden, 2000] Percival, D. B. a Walden, A. T. (2000).
Wavelet Methods for Time Series Analysis.
Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge.

[Press et al., 1997a] Press, W. H., Teukolsky, S. A., Vetterling, W. T., a Flannery, B. P. (1997a).
Numerical Recipes in C: The Art of Scientific Computing.
Cambridge University Press, Cambridge, 2nd vydání.

[Press et al., 1997b] Press, W. H., Teukolsky, S. A., Vetterling, W. T., a Flannery, B. P. (1997b).
Numerical Recipes in Fortran 77: The Art of Scientific Computing.

Cambridge University Press, Cambridge, 2nd vydání.
[Fortran Numerical Recipes, Vol 1].

[Press et al., 1997c] Press, W. H., Teukolsky, S. A., Vetterling, W. T., a Flannery, B. P. (1997c).
Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing.
Cambridge University Press, Cambridge.
[Fortran Numerical Recipes, Vol 2, with foreword by M. Metcalf].

[Prudnikov et al., 1981] Prudnikov, A. P., Bryčkov, J. A., a Maričev, O. I. (1981).
Integraly i rjady. Elementarnyje funkcii.
Nauka, Moskva.

[Práger a Sýkorová, 2004] Práger, M. a Sýkorová, I. (2004).
Jak počítače počítají.
Pokroky Mat. Fyz. Astronom., 49(1):32–45.

[Ralston, 1978] Ralston, A. (1978).
Základy numerické matematiky.
Academia, Praha.
2. vydání.

[Rektorys, 1995] Rektorys, K. (1995).
Přehled užité matematiky.
Prometheus, Praha.

[Sandu, 2001] Sandu, A. (2001).
Lecture Notes: Introduction to Fortran 95 and Numerical Computing. A Jump-Start for Scientists and Engineers.
http://www.cs.mtu.edu/~asandu/Courses/CS2911/fortran_notes/main.html.

[Schwartz, 1972] Schwartz, L. (1972).
Matematické metody ve fyzice.
SNTL – Nakladatelství technické literatury, Praha.

[Segeth, 1998] Segeth, K. (1998).
Numerický software.
Karolinum – nakladatelství Univerzity Karlovy, Praha.

[Stroud a Secrest, 1966] Stroud, A. H. a Secrest, D. (1966).
Gaussian Quadrature Formulas.
Prentice Hall, Englewood Cliffs, NJ.

[Vetterling et al., 1993] Vetterling, W. T., Teukolsky, S. A., Press, W. H., a Flannery, B. P. (1993).
Numerical Recipes Example Book (C).
Cambridge University Press, Cambridge, 2nd vydání.

[Walker, 1999] Walker, J. S. (1999).
A Primer on Wavelets and Their Scientific Applications.
Chapman & Hall/CRC Press, Boca Raton.