B33POS: Počítačové systémy

Lekce 04. Reálná čísla

Petr Štěpán
stepan@fel.cvut.cz

9. října, 2025

Obsah

Bitové operace

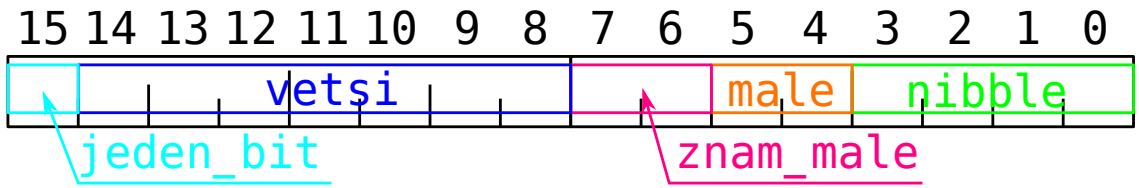
- V jazyce C a ve strojových instrukcích procesoru jsou definovány bitové operace – and, or, xor, negace.
- Pokud máte proměnnou `int a=-10` a chcete zjistit její nejméně významný bajt, pak můžete použít instrukci bitový and `unsigned int bajt = a & 0xff;`
- POZOR neplést s operací `&&`, toto je logický and.
- Jak instrukce pracuje, provede "logický"and mezi sobě odpovídajícími bity čísla:

```
   a = 11111111 11111111 11111111 11110110
       &&&&&&&& &&&&&&&& &&&&&&&& &&&&&&&&
0xff = 00000000 00000000 00000000 11111111
-------------------------------------------
a&0xff = 00000000 00000000 00000000 11110110
```

Bitové operace

Programovací jazyk C umí pracovat s čísly u nichž zadáte jejich bitovou velikost. Struktura s více bitovými položkami může být definována následujcím způsobem:

```
struct pole {
  int nibble:4;
  unsigned int male:2;
  int znam_male:2;
  unsigned int vetsi:7;
  unsigned int jeden_bit:1;
}
```



- položka nibble je číslo se znaménkem a má 4 bity
- položka male je malé číslo bez znaménka o rozsahu 0...3
- položka znam_male je číslo od -2 .. 1
- položka vetsi je číslo od 0 do 127
- položka jeden_bitje buď 0 nebo 1

Bitové operace

Jak bude skutečně definovaná struktura `struct pole` uložena v paměti záleží na překladači.

Struktura bude optimálně uložená ve dvou bajtech, ale může být také uložena ve více bajtech od 5 do 8.

Zarovnávání položek ve strukturách a třídách lze ovlivnit příkazem:
`#pragma pack(N)`,
kdy N definuje na kolik bajtů se jednotlivé položky zarovnávají.

Tento přístup není standardizován, proto je dobré, umět si naprogramovat ukládání do podobné struktury sami.

Bitové operace - zápis hodnoty

Jak uložit hodnotu (value), která má maximálně k-bitů na pozici l bitů od nejméně významného bitu?
Nejdříve je nutno nastavit k-bitů na 0.

```
unsigned short pole=0xffff;
usngigned mask_k = (1<<k)-1;
pole &= (~(mask_k<<l));
```

mask_k je číslo, které má na nejnižších k bitech 1 a jinde 0
mask_k<<l je číslo, které má na odpovídajících k bitech 1 a jinde 0
~ (mask_k<<l) je bitová negace předchozí hodnoty, na odpovídajících místech je k 0 jinde jsou 1
Nyní můžeme nastavit hodnotu value operací or.

```
pole |= (value & mask_k)<<l;
```

(value & mask_k) pro jistotu ořízneme hodnotu value pouze na k-bitů
(value & mask_k)<<l posuneme hodnotu na správnou pozici
pole |= (value & mask_k)<<l zapíšeme hodnotu na správnou pozici v poli

Bitové operace - zápis hodnoty

příklad, jak to bude vypadat pro k=5 a l=4

```
1<<5    = 0b0000000000100000
1<<5-1  = 0b0000000000011111 = mask_k

 mask_k<<4 0b0000000111110000
~ mask_k<<4 0b1111111000001111
pole       = 0b1111111111111111
pole & (~ (mask_k<<4))
           = 0b1111111000001111
value      = 0b0000010000001101
value & mask_k =
           = 0b0000000000001101
(value & mask_k)<<4
           = 0b0000000011010000
pole |= (value & mask_k)<<4
           = 0b1111111011011111
```

červená je chyba, value má mít jen 5 bitů – oprava chyby

Bitové operace - čtení hodnoty

Jak přečíst hodnotu (value) bez znaménka, která má maximálně k-bitů na pozici l bitů od nejméně významného bitu?
Nejdříve je nutno nastavit k-bitů na 0.

```
value = ((pole>>l) & mask_k);
```

mask_k je číslo, které má na nejnižších k bitech 1 a jinde 0
(pole>>l) je číslo, které má odpovídající bity posunuty na novou správnou pozici
(pole>>l) & mask_k je výsledná hledaná hodnota

příklad:

```
pole                = 0b1111111011011111
pole >>4            = 0b0000111111101101
pole >>4 & mask_k   = 0b0000000000001101
```

Bitové operace - čtení hodnoty se znaménkem

Jak přečíst hodnotu (value) se znaménkem, která má maximálně k-bitů na pozici l bitů od nejméně významného bitu?
Podobně jako u čísle bez znaménka, ale nakonec musíme znamínkový bit rozšířit va všechny bit vlevo výsledného typu:

```
mex_bit = 1<<(k-1)
value = (((pole>>l) & mask_k)^max_bit)-max_bit;
```

((val) ^ max_bit) - max_bit je znaménkové rozšíření na plnou hodnotu typu value

příklad:

```
value = -1              = 0b1111111111111111

pole                    = 0b1111111111111111
pole >>4                = 0b0000111111111111
pole >>4 & mask_k       = 0b0000000000011111
(pole >>4 & mask_k) ^  max_bit
                        = 0b0000000000001111
((pole >>4 & mask_k) ^  max_bit) - max_bit
                        = 0b1111111111111111
```

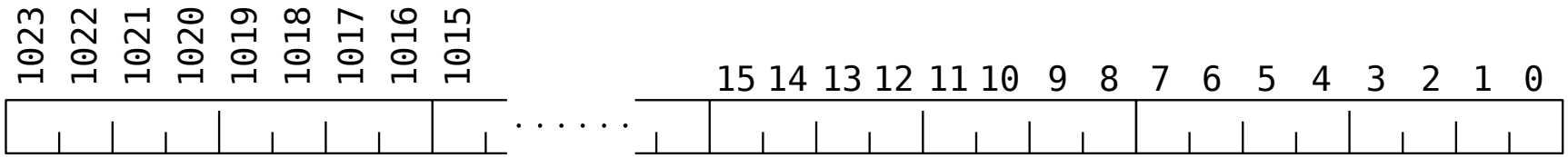
Bitové operace

```
unsigned short pole=0xffff;

pole &= (~0xf);
pole |= (nibble &0xf);

pole &= (~0x30);
pole |= ((male &0x3)<<4);

pole &= (~0xc0);
pole |= ((znam_male &0x3)<<6);

pole &= (~0x7F00);
pole |= ((vetsi &0x7F)<<8);

pole &= (~0x8000);
pole |= ((jeden_bit &0x1)<<15);
```

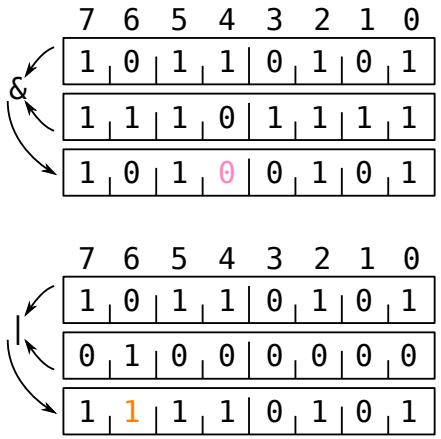
```
printf("nibble = %u\n",
        pole&0x0f);

printf("male = %u\n",
        (pole>>4)&0x3);

printf("znam_male = %d\n",
  (((pole>>6)&0x3)^0x2)-2);

printf("vetsi = %u\n",
        (pole>>8)&0x7f);

printf("jeden bit = %u\n",
        (pole>>15)&0x1);
```

Bitové pole

Bitové pole se používá často jako nejefektivnější způsob zjištění, zda jsou některé objekty použité nebo nepoužité.




```
unsigned char array[128];
// 1024 bitů uložených v poli 128 bajtů
void set_bit(int index, unsigned char bit) {
  if (bit==0) {
    array[index>>3] &= (~(1<<(index&7)));
  } else {
    array[index>>3] |= (1<<(index&7));
  }
}
 unsigned char get_bit(int index) {
   return (array[index>>3]>>(index&7))&1u;
 }
```



Bitové pole

Použití funkcí je pak již jednoduché:

```
int main() {
  int i;

  for (i=0; i<1024; i++) {
    set_bit(i, (i%5)==0);
  }

  for (i=0; i<1024; i++) {
    printf("bit %i = %u\n", i , get_bit(i));
  }
  return 0;
}
```

Obsah

Reálná čísla

- Celé číslo X v binární soustavě je součet k bitů b_i vynásobených mocninami 2, tedy $X = \sum_{i=0}^{k-1} b_i 2^i$
- Reálné číslo X v binární soustavě je obdobný součet $k + j$ bitů b_i vynásobených mocninami 2, ale začneme již v záporných mocninách: $X = \sum_{i=-j}^{k} b_i 2^i$
- jistě si všichni pamatujete ze střední školy, že $2^{-j} = \frac{1}{2^j}$

$$\begin{array}{cccccccccccccc} & & & & & & & \frac{1}{2} & \frac{1}{4} & & & & & \\ & & & & & 4 & 2 & 1 & \frac{1}{2} & \frac{1}{4} & & & & \\ & & & & & \| & \| & \| & \| & \| & & & & \\ 2^{i} & 2^{i-1} & 2^{i-2} & \ldots & 2^{2} & 2^{1} & 2^{0} & 2^{-1} & 2^{-2} & \ldots & 2^{-j+1} & 2^{-j} \\ b_i & b_{i-1} & b_{i-2} & \ldots & b_2 & b_1 & b_0 . & b_{-1} & b_{-2} & \ldots & b_{-j+1} & b_{-j} \end{array}$$

Reálná čísla s pevnou desetinnou čárkou

Čísla s pevnou desetinnou čárkou (fixed point numbers):

- obdoba čísel s posunutou 0
- reálné číslo je reprezentované k-bitovým celým číslem se znaménkem, dále zvolíme pevný počet desetinných míst s bitů, kdy $0 \leq s \leq k$
- reprezentace – kód čísla X je $A(X) = \lceil X \cdot 2^s \rceil$
- rozkódování je funkcí $D(A) = \frac{A}{2^s}$
- rozsah čísel reprezentovaných je $< -\frac{2^{k-1}}{2^s}, \frac{2^{k-1}-1}{2^s} >$
- přesnost reprezentace čísel je $\pm\frac{1}{2^s}$.

Speciální případ:

- pokud chcete reprezentovat reálná čísla z intervalu $< 0, 1)$
- pak můžete nastavit $s = k$ a použít bezznaménkovou reprezentaci celých čísel
- dosáhnete lepší přesnosti než odpovídající `float`, nebo `double`

Některé SIMD instrukce používají pevnou desetinnou čárku.

Fixed point – implementace v C

```
#define NUM_POINTS 10
#define NUM_MASK  ((1<<NUM_POINTS)-1)

int fixed_p1, fixed_p2;

fixed_p1 = (15 << NUM_POINTS) + ((21<< NUM_POINTS)/100);
// fixed_p = 15.21 = 15 + 21/100
fixed_p2 = (3 << NUM_POINTS) +
          (((long long int)141592<< NUM_POINTS)/1000000);
// fixed_p2 = 3.141592 = 3 + 141592/1000000

printf("%08x %08x\n", fixed_p1, fixed_p2);
// vytiskne  0000f35c 00003243
printf("%lf %lf\n", fixed_p1/(double)(1<<NUM_POINTS),
                    fixed_p2/(double)(1<<NUM_POINTS));
// vytiskne  15.209961 3.141357
// spravne   15.21     3.141592
```

Reálná čísla s pevnou desetinnou čárkou

Počítání s čísly s pevnou desetinnou čárkou:

- součet a rozdíl je součtem a rozdílem celočíselné reprezentace čísel
- násobení je složitější:
  - $A(X \cdot Y) = (X \cdot Y) \cdot 2^s = \frac{(X \cdot 2^s) \cdot (Y \cdot 2^s)}{2^s} = \frac{A(X) \cdot A(Y)}{2^s}$
- dělení je obdobné:
  - $A(\frac{X}{Y}) = (\frac{X}{Y}) \cdot 2^s = \frac{(X \cdot 2^s) \cdot 2^s}{(Y \cdot 2^s)} = \frac{A(X) \cdot 2^s}{A(Y)}$
- Nejedná se o výrazné zesložitění, protože násobení číslem 2^s je posun o s bitů doleva a dělení číslem 2^s je posun o s bitů doprava.

Fixed point – matematické operace

```
int fixed_sum, fixed_sub;

fixed_sum = fixed_p1 + fixed_p2;
fixed_sub = fixed_p1 - fixed_p2;
printf("sum %08x sub %08x \n", fixed_sum, fixed_sub);
// vytiskne  sum 0001259f sub 0000c119
printf("%lf %lf\n", fixed_sum/(double)(1<<NUM_POINTS),
                    fixed_sub/(double)(1<<NUM_POINTS));
// vytiskne  18.351318 12.068604
// spravne   18,351592 12,068408

int fixed_mul, fixed_div, fixed_div2;
fixed_mul = ((long long int)fixed_p1*fixed_p2)>>NUM_POINTS;
fixed_div = (fixed_p1/fixed_p2)<<NUM_POINTS;                    // ztrata presnos
fixed_div2 = ((long long int)fixed_p1<<NUM_POINTS)/fixed_p2;

printf("mul %x div1 %x div2 %x\n", fixed_mul, fixed_div, fixed_div2);
// vytiskne mul 2fc7a div1 4000 div2 4d78
printf("mul %lf div1 %lf div2 %lf\n", fixed_mul/(double)(1<<NUM_POINTS),
  fixed_div/(double)(1<<NUM_POINTS), fixed_div2/(double)(1<<NUM_POINTS));
// vytiskne mul 47.779785   div1 4.000000 div2 4.841797
// spravne  mul 47,78361432 div 4,841494376
```

Obsah

Plovoucí desetinná čárka

Plovoucí desetinná čárka (floating point numbers)

Obdoba vědeckého zápisu reálných čísel:
$-123000000000000.0 = -1.23 \cdot 10^{14} = -1.23\text{E}14$
$0.000000000000123 = -1.23 \cdot 10^{-13} = -1.23\text{E} - 13$

Binární reprezentace používá obdobně mocninu 2:
$110110000000000.0 = 1.1011 \cdot 2^{14} = 1.1011\text{E}14 = 29696_{10}$
$-0.0000000000000011101 = -1.1101 \cdot 2^{-16} = -1.1101\text{E} - 16 \approx$
$\approx 0.00002765_{10}$

Všimněte si, že každé reálné číslo (kromě 0) začíná v binárním vědeckém zápisu jedničkou.

IEEE-754

Standard IEEE-754 definuje, jakým způsobem zakódovat reálné číslo do 32, 64 bitů.
Reálné 32-bitové číslo obsahuje:

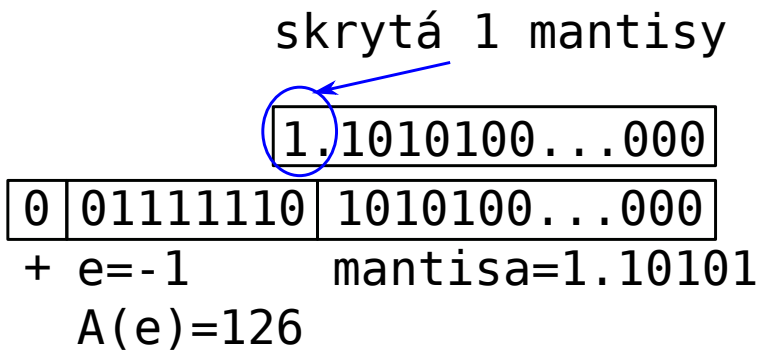
- 1 bit znaménko (pozn. existuje 0 i -0)
- 8 bit hodnota exponentu s posunutou nulou o 127
- 23 bitů hodnota mantisy, tedy cifer reprezentujících hodnotu

Reálné 64-bitové číslo obsahuje:

- 1 bit znaménko (pozn. existuje 0 i -0)
- 11 bitů hodnota exponentu s posunutou nulou o 1023
- 52 bitů hodnota mantisy, tedy cifer reprezentujících hodnotu

IEEE-754

Příklad: Reálné číslo $0.828125_{(10)} = 0.5 + 0.25 + 0.0625 + 0.015625 = 2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} = 0.110101_{(2)}$.
Číslo převedeme do vědecké notace: $0.110101 = 1.10101\mathrm{E} - 1$.
Exponent $e = -1$, reprezentace s posunutou nulou o 127 je $A(-1) = -1 + 127 = 126$.
Abychom neplýtvali zbytečně bity, pro normalizovaná čísla se první 1 mantisy neukládá do binární reprezentace čísla:



Zkuste si
`https://www.h-schmidt.net/FloatConverter/IEEE754.html`

IEEE-754

Normalizované číslo, je každé číslo, které lze zapsat jako `1.XXXXX E exp`, kde exp je číslo od -126 do 127, tedy jehož reprezentace je od 1 do 254.

Denormalizovaná čísla jsou čísla, která lze zapsat jako `0.XXXXX E -126`, tedy například 0.0, nebo všechna čísla v intervalu $(-1.17549\text{E}-38, 1.17549\text{E}-38)$, tedy od $(-2^{-126}, 2^{-126})$.

Pokud je reprezentace exponentu nastavena na samé 1, tedy 255, tzn. hodnota exponentu je 128, pak se jedná o speciální čísla.

- pokud je mantisa 0, pak se jedná o nekonečno, podle znaménka *-inf*, nebo *inf*.
- pokud je mantisa nenulová, pak se jedná o speciální výraz *NaN* – Not a Number, tedy chybná hodnota čísla, například po výpočtu odmocniny záporného čísla

IEEE-754

Přehled reálných čísel:

Exponent	Mantisa	Hodnota
00000000	0	0.0 – čistá nula
00000000	nenulová	Denormalizovaná čísla blízká 0
00000001	0	nejmenší normalizované číslo se skrytou 1 v mantise
1 až 254	cokoliv	normalizovaná čísla, skrytá 1 v mantise
11111111	0	nekonečno
11111111	nenulová	NaN chybná hodnota

Denormalizované číslo s nejmenší absolutní hodnotou různé od 0 je:

- exponent = 0 (-126), mantisa=000...0001, hodnota = $2^{-23+(-126)} \approx 1.4E-45$

Normalizované číslo s nejmenší absolutní hodnotou:

- exponent = 1 (-126), mantisa=000...0000, hodnota = $2^{-126} \approx 1.17E-38$

Normalizované číslo s největší absolutní hodnotou:

- exponent = 255 (127), mantisa=111...1111, hodnota = $(2-2^{-23})2^{127} \approx 3.4E38$

Float point decode – implementace v C

Dekódování normalizovaných čísel v jazece C

```
#include <stdio.h>

#define SIGN(x)
              ((((*(unsigned int*)&x)>>31)&0x1)?'+':'-')
#define EXP(x)
              (((*(unsigned int*)&x)>>23)&0xff)
#define MANTISA(x)
              ((*(unsigned int*)&x)&0x7FFFFF)

int main() {
  float a = 0.125;
  float b = -0.1;

  printf("Hodnota %f znamenko %c exp %i(%i) mantisa %x\n", a,
     SIGN(a), EXP(a)-127, EXP(a), MANTISA(a));
  printf("Hodnota %f znamenko %c exp %i(%i) mantisa %x\n", b,
     SIGN(b), EXP(b)-127, EXP(b), MANTISA(b));
}
```

Rozumíte definovaným makrům?

IEEE-754 revize 2008

Standard IEEE-754 navíc definuje reálné číslo s 16 bity (half precision) a s 128 bity (quad precision).
Reálné 16-bitové číslo obsahuje:

- 1 bit znaménko
- 5 bitů hodnota exponentu s posunutou nulou o 15
- 10 bitů hodnota mantisy, (plus 11tý bit skrytý neukládaný)

Reálné 128-bitové číslo obsahuje:

- 1 bit znaménko
- 15 bitů hodnota exponentu s posunutou nulou o 16383
- 112 bitů hodnota mantisy, (plus 113 bit skrytý)

IEEE-754

Porovnání reálných čísel na velikost:

- Kladná čísla jsou vždy větší než záporná
- Po odstranění znamének, lze absolutní hodnoty čísel porovnat tak, jak jsou uložené v paměti jako celé číslo bez znaménka
  - To je možné díky zvolené reprezentaci exponentu jako čísla s posunutou nulou
  - Větší exponent větší číslo, při rovnosti exponentů větší mantisa znamená větší číslo

IEEE-754

Sčítání nebo odčítání dvou reálných čísel ve vědecké notaci

1. Převést mantisy čísel na společný exponent, který má hodnotu největšího exponentu (porušíme notaci)
2. Provést součet, nebo rozdíl mantis i se skrytými jedničkami
3. Výsledné číslo normalizovat
   - při sčítání se může exponent zvětšit
   - při odčítání se exponent může zmenšit

IEEE-754

Příklad: Sčítání dvou reálných čísel 31.5+0.75

$31.5_{(10)} = 11111.1_{(2)} = 1.11111\text{E}4 \qquad 0.75_{(10)} = 0.11_{(2)} = 1.1\text{E}-1$

Obě čísla převedeme na stejný exponent 4 a sečteme:

```
  1.11111
  0.000011
-----------
 10.000001
```

Výsledné číslo musíme znormalizovat zvýšením exponentu na 5 a tím posunutím desetinné čárky vlevo:

$10.000001\text{E}4 = 1.0000001\text{E}5$

Toto číslo je reprezentací čísla 32.25.

IEEE-754

Příklad: Sčítání dvou reálných čísel 31.5+0.75

$31.5_{(10)} = 11111.1_{(2)} = 1.11111\text{E}4 \qquad 0.75_{(10)} = 0.11_{(2)} = 1.1\text{E}-1$

Obě čísla převedeme na stejný exponent 4 a sečteme:

```
  1.11111
  0.000011
-----------
 10.000001
```

Výsledné číslo musíme znormalizovat zvýšením exponentu na 5 a tím posunutím desetinné čárky vlevo:

$10.000001\text{E}4 = 1.0000001\text{E}5$

Toto číslo je reprezentací čísla 32.25.

Float point sčítání – implementace v C

Algoritmus sčítání v jazyce C (bez uvážení znamének)

```
#include <stdio.h>
#define SIGN(x)
  ((((*(unsigned int*)&x)>>31)&0x1)?'+':'-')
#define EXP(x)
  (((*(unsigned int*)&x)>>23)&0xff)
#define MANTISA(x)
  ((*(unsigned int*)&x)&0x7FFFFF)

int main() {
  float a = 31.5;
  float b = 0.75;
  float c = a+b;
  int e_a, e_b, e_c;
  unsigned int man_a, man_b, man_c;
  int shift;

  e_a = EXP(a);
  e_b = EXP(b);

  man_a = MANTISA(a)|0x800000;
  man_b = MANTISA(b)|0x800000;

  if (e_a>e_b) {
    shift = e_a-e_b;
    e_c = e_a;
    man_b=(man_b>>shift);
  } else {
    shift = e_b-e_a;
    e_c = e_b;
    man_a=(man_a>>shift);
  }
  man_c = man_a+man_b;

  if ((man_c&0x1000000)!=0) {
    man_c = man_c>>1;
    e_c++;
  }
  man_c = (man_c & 0x7FFFFF);
  printf("C %i %06x, spocteno %i %06x\n",
         EXP(c), MANTISA(c), e_c, man_c);
```

Uměli byste přidat vzít v úvahu znaménko?

IEEE-754 – Násobení

Násobení dvou reálných čísel:

1. Exponent výsledku je součet exponentů čísel
2. Mantisa výsledku je součin mantis čísel (i se skrytými jedničkami)
3. Výsledné číslo normalizovat
   - podle výsledku násobení mantis je někdy nutné zvýšit exponent o 1 a vyrotovat mantisu doprava

IEEE-754 – Násobení

Příklad: Vynásobíme čísla $0.375 \cdot 1.25$

$0.375_{(10)} = 0.011_{(2)} = 1.1\mathrm{E}-2$ $\quad$ $1.25_{(10)} = 1.01_{(2)} = 1.01\mathrm{E}0$

Součin mantis je:

```
   11 odpovídá 1.1
 *101 odpovídá 1.01
-------------------
   11
  00
 11
-------------------
 1111
```

odpovídá `1.111`, výsledek má 3 desetinná čísla.

```
  375 odpovídá 0.375
 *125 odpovídá 1.25
-------------------
 1875
 750
375
-------------------
46875
```

odpovídá `0.46875`, výsledek má 5 desetinná čísla

Exponent vychází $-2+0=-2$ a hodnota čísla je tedy $1.111E-2$ tedy $0.01111_{(2)} = 0.25 + 0.125 + 0.0625 + 0.03125 = 0.46875_{(10)}$

Float point násobení – implementace v C

Algoritmus násobení v jazyce C (bez uvážení znamének)

```
#include <stdio.h>
#define SIGN(x)
  ((((*(unsigned int*)&x)>>31)&0x1)?'+':'-')
#define EXP(x)
  (((*(unsigned int*)&x)>>23)&0xff)
#define MANTISA(x)
  ((*(unsigned int*)&x)&0x7FFFFF)

int main() {
  float a = 0.375;
  float b = 1.25;
  float c = a*b;
  int e_a, e_b, e_c;
  uint32_t man_a, man_b, man_c;
  int shift;

  e_a = EXP(a);
  e_b = EXP(b);
  e_c = e_a + e_b - 127;
```

```
  man_a = MANTISA(a)|0x800000;
  man_b = MANTISA(b)|0x800000;
  man_c = (((uint64_t)man_a)*man_b)>>23;

  while ((man_c&0x1000000)!=0) {
    man_c = man_c>>1;
    e_c++;
  }
  man_c = (man_c & 0x7FFFFF);
  printf("C %i %06x, spocteno %i %06x\n",
         EXP(c), MANTISA(c), e_c, man_c);
```

Uměli byste přidat vzít v úvahu znaménko?

IEEE-754

Reálná čísla shrnutí:

- reálná čísla s plovoucí čárkou umí reprezentovat čísla ve velmi velkém rozsahu:
  - float – absolutní hodnota čísel od $1.175494351E-38$ do $3.402823466E+38$
  - double – absolutní hodnota čísel od $2.2250738585072014E-308$ do $1.7976931348623158E+308$
- přesnost čísla se udává na počet validních cifer:
  - float – v desítkové soustavě 6-7 platných cifer
  - double – v desítkové soustavě 15-16 platných cifer

POZOR: Následující `while` cyklus neskončí:

```
float a=1.0, step=5e-8;
while (a*a<1.01) {
  a+=step;
}
```