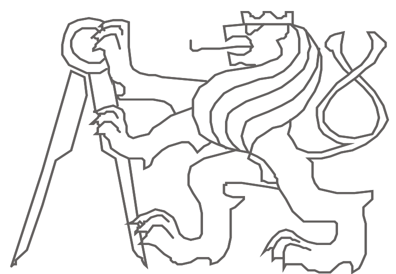
Architektura počítačů

Počítačová aritmetika a úvod

Richard Šusta, Pavel Píša

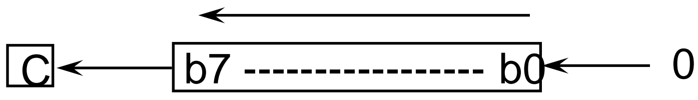


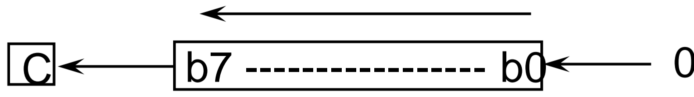
České vysoké učení technické, Fakulta elektrotechnická

Rychlost integer operací

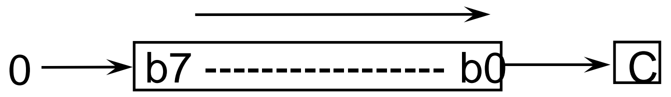
<table>
<tr><th>Operace</th><th>Jazyk C</th></tr>
<tr><td>Bitová negace</td><td>~x</td></tr>
<tr><td>Násobení či dělení 2^n</td><td>x<<n , x>>n</td></tr>
<tr><td>Increment, decrement</td><td>++x, x++, --x, x--</td></tr>
<tr><td>Negace čísla <- bitová negace + increment</td><td>-x</td></tr>
<tr><td>Sčítání</td><td>x+y</td></tr>
<tr><td>Odčítání <- negace čísla + sčítání</td><td>x-y</td></tr>
<tr><td>Násobení na hardwarové násobičce</td><td rowspan="2">x*y</td></tr>
<tr><td>Násobení na sekvenční násobičce</td></tr>
<tr><td>Dělení</td><td>x/y</td></tr>
</table>

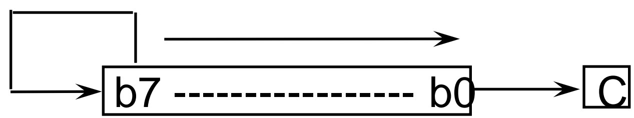
Logical Shift   Arithmetic Shift





násobení 2 x





dělení 2 unsigned

dělení 2 signed

Přetečení - unsigned

- V tomto případě sledujeme **přenos z nejvyššího řádu**
- Opět situace, kdy výsledek operace není správný, protože se nevešel do zobrazitelného rozsahu.

Mějme k dispozici 5 bitů:

28			1	1	1	0	0
+5	+		0	0	1	0	1
?1		1	0	0	0	0	1

12			0	1	1	0	0
+5	+		0	0	1	0	1
17		0	1	0	0	0	1

28			1	1	1	0	0
21	+		1	0	1	0	1
?17		1	1	0	0	0	1

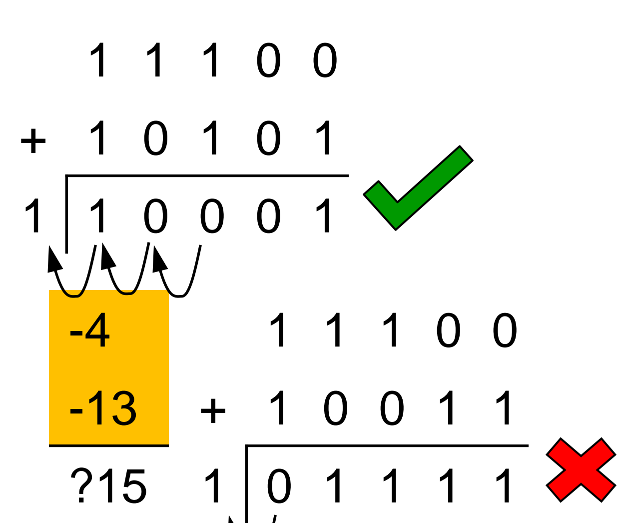
28			1	1	1	0	0
19	+		1	0	0	1	1
?15		1	0	1	1	1	1

Chybný výsledek je vždy menší než oba sčítance!

Přetečení signed

- Přeplnění - říká se tomu také **přetečení (overflow)**.
- Situace, kdy výsledek operace není správný, protože se nevešel do zobrazitelného rozsahu.
- **Nastává v situaci, kdy znaménko výsledku je jiné, než znaménka operandů, byla-li stejná**, nebo
- Nonekvivalencí přenosu **do** a **z** nejvyššího řádu.

Mějme k dispozici 5 bitů:



-4		1 1 1 0 0
+5	+	0 0 1 0 1
1	1	0 0 0 0 1

12		0 1 1 0 0
+5	+	0 0 1 0 1
?-15	0	1 0 0 0 1

Sign Extension Example in C

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Mutliplication of signed numbers

Multiplication in Two's complement cannot be accomplished with the standard technique since, as far as the machine itself is concerned, for Y[n]:

$$-Y = 0 - Y = 2^n - Y$$

since, when subtracting from zero, need to "borrow" from next column leftwards.

Consider $X \times (-Y)$

Internal manipulation of $-Y$ is as $2^n - Y$

Therefore $X \times (-Y) = X \times (2^n - Y) = 2^n \times X - X \times Y$

However the expected result should be $2^{2n} - (X \times Y)$

Signed Multiplication

- Case 1: Positive Multiplier

Multiplicand	$1100_2 = -4$
Multiplier	× $0101_2 = +5$
Sign-extension	11111100
	11110000
Product	$11101100_2 = -20$

- Case 2: Negative Multiplier

Multiplicand	$1100_2 = -4$
Multiplier	× $1101_2 = -3$
Sign-extension	11111100
	11110000
	00100000 (2's complement of 1100)
Product	$00001100_2 = +12$

Non-restoring division

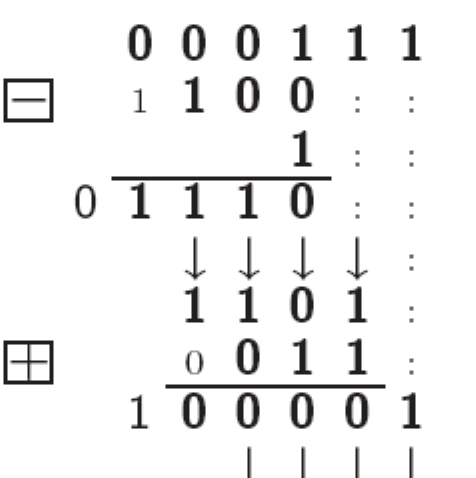
7 / 3

7 – 4*3 = -5

(non-restoring)
-5+2*3 = 1
=7 - 2*3

1 – 3 = -2
(restoring)
-2+3 = 1

1 1 1 : 0 1 1



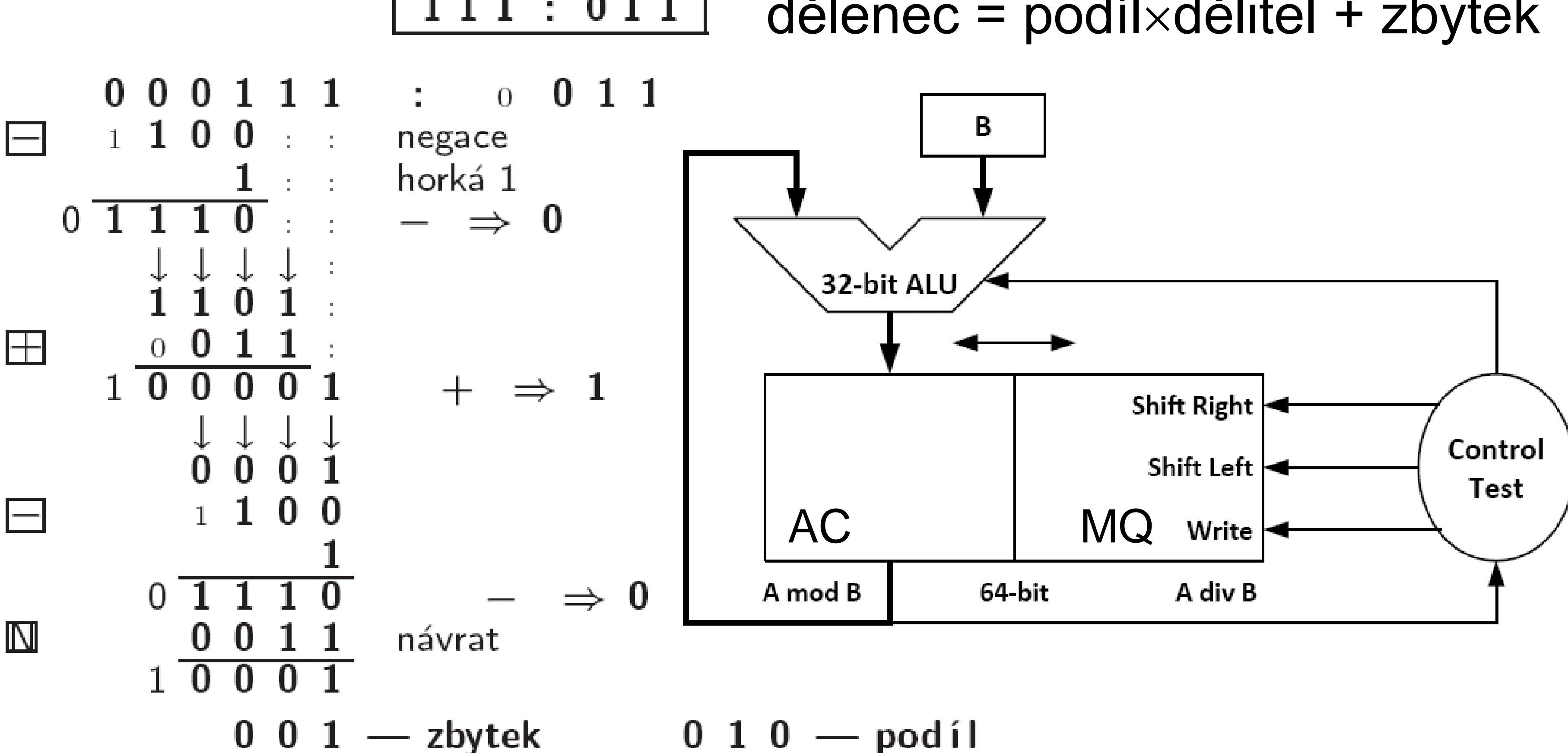
negace
horká 1
– ⇒ 0
+ ⇒ 1

0 0 0 1
1 1 0 0
1
0 1 1 1 0 – ⇒ 0
0 0 1 1 návrat
1 0 0 0 1

0 0 1 — zbytek 0 1 0 — podíl

ALU neví, zda je číslo menší či ne, musí odečíst, aby to zjistila, a případně výsledek korigovat přičtením.

Sekvenční HW dělička (varianta 32b)



Non-restoring division

Algoritmus dělení v C

Restoring division

```
MQ = dělenec;
B = dělitel; (Podmínka: dělitel různý od 0!)
AC = 0;

for( int i=1; i <= n; i++)          {
   SL (posuň registr AC MQ o jednu pozici vlevo, přičemž vpravo se připíše nula)
   if(AC >= B)   {
           AC = AC – B;
           MQ0 = 1;        // nejnižší bit registru MQ se nastaví na 1
    }
}
```

→ Nyní registr MQ obsahuje podíl a zbytek je v AC

Příklad x/y

Dělenec x=1010 a dělitel y=0011 **Restoring division**

i	operace	AC	MQ	B	komentář
		0000	1010	0011	prvotní nastavení
1	**SL**	0001	0100		
	nic	0001	0100		podmínka if není splněna
2	**SL**	0010	1000		
		0010	1000		podmínka if není splněna
3	**SL**	0101	0000		$r \geq y$
	AC = AC – B; MQ_0 = 1;	0010	0001		
4	**SL**	0100	0010		$r \geq y$
	AC = AC – B; MQ_0 = 1;	0001	0011		konec cyklu

x : y = 1010 : 0011 = 0011 zbytek 0001, (10 : 3 = 3 zbytek 1)

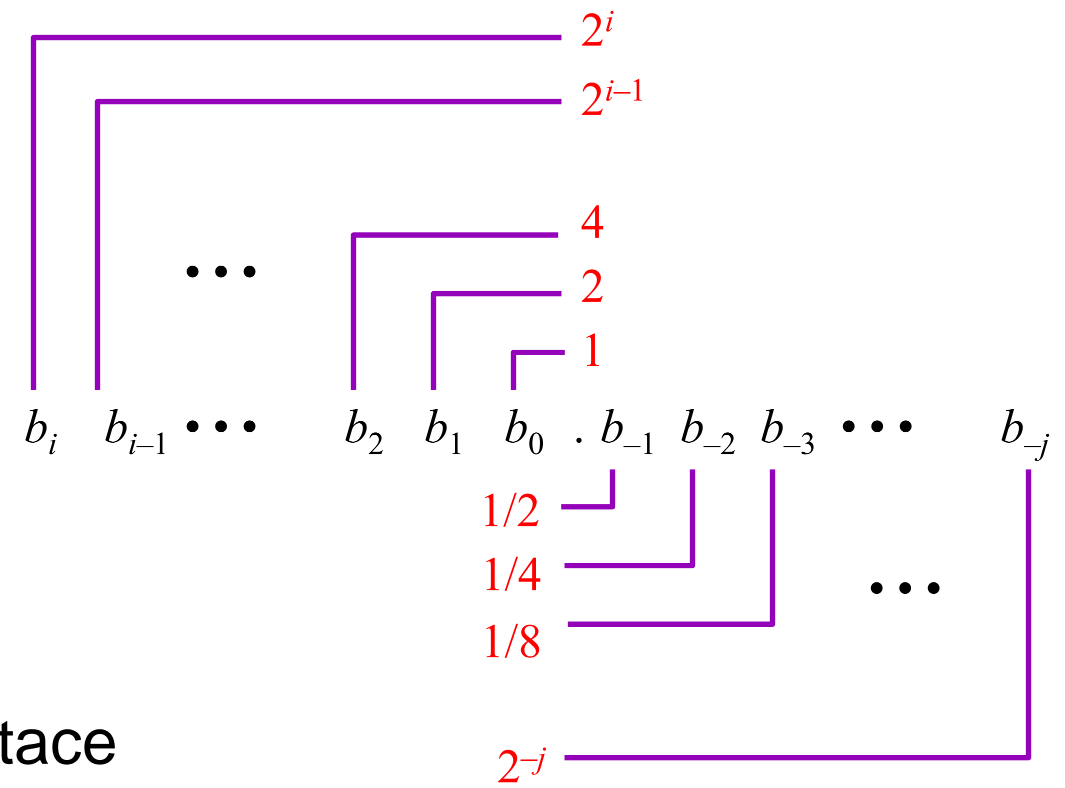
*Reálná čísla

a jejich zobrazení v počítači

Jak se v počítači zobrazují čísla typu REAL?

- Vědecká, neboli semilogaritmická notace.
  - Dvojice: EXPONENT (E), ZLOMKOVÁ část (nazývaná též mantisa M).
  - **Mantisa x zaklad$^{\text{Exponent}}$**
- Notace je normalizovaná.
  - Zlomková část vždy začíná binární číslicí 1,
  - Obecně: nenulovou číslicí <1, z - 1>.
- Dekadicky: $7{,}26478 \times 10^3$
- Binárně: $1{,}010011 \times 2^{1001}$

Fractional Binary Numbers
(zlomková binární čísla / čísla v pevné řádové čárce)



Reprezentace

- bity vpravo od “binary point” udávají zlomky mocnin 2
- reprezentuje reálná čísla

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Zlomková čísla / Pevná řádová čárka

Hodnota	*Reprezentace*
5-3/4	101.11_2
2-7/8	10.111_2
63/64	0.111111_2

Operace

- Dělení 2 posunem vpravo
- Násobení 2 posunem vlevo
- Čísla pod $0.111111\ldots_2$ jsou menší než 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \to 1.0$
  - Přesná notace $\to 1.0 - \varepsilon$

Binární → Dekadické

$23.47 = 2\times10^{1} + 3\times10^{0} + 4\times10^{-1} + 7\times10^{-2}$

↑ desetinná tečka

$10.01_{two} = 1\times2^{1} + 0\times2^{0} + 0\times2^{-1} + 1\times2^{-2}$

↑ binární tečka

$= 1\times2 \quad + 0\times1 \quad + 0\times\frac{1}{2} \quad + 1\times\frac{1}{4}$

$= 2 + 0.25 = 2.25$

Vědecká notace

Dekadické číslo:

-123,000,000,000,000 ➔ -1.23×10^{14}

0.000 000 000 000 000 123 ➔ $+1.23 \times 10^{-16}$

Binární číslo:

110 1100 0000 0000 ➔ $1.1011 \times 2^{14} = 29696_{10}$

-0.0000 0000 0000 0001 1011 ➔ -1.1101×2^{-16}

$= -2.765655517578125 \times 10^{-5}$

Pozor

Konečné dekadické číslo ➔ konečné binární číslo

Příklad:

0.1_{ten} ➔ 0.2 ➔ 0.4 ➔ 0.8 ➔ 1.6 ➔ 3.2 ➔ 6.4 ➔ 12.8 ➔…

$0.1_{10} = 0.00011001100110011\ldots_2$

$= 0.0\underline{0011}_2$ (nekonečný řetězec bitů)

S více bity se zpřesňuje reprezentace 0.1_{10}

Příklad 0.1_{10} - Převeďte na reálné číslo

$0.1_{10} = 0.\underline{000110011....}_2 =$

```
0.00011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
0011...
```

Reprezentace

Omezení

lze přesně vyjádřit jen čísla $x/2^k$

Ostatní čísla jsou uložená nepřesně

Hodnota	*Reprezentace*
1/3	$0.0101010101[01]..._2$
1/5	$0.001100110011[0011]..._2$
1/10	$0.0001100110011[0011]..._2$

Propojení hardware, signálů a software

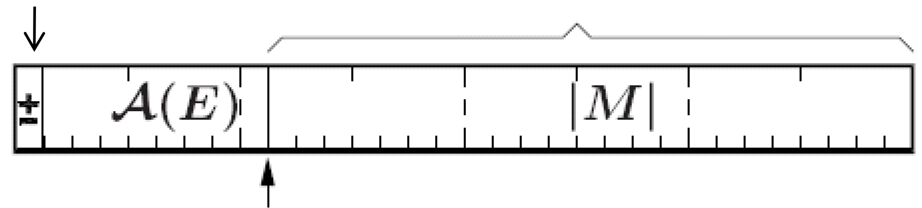
- Reprezentace je definovaná normou **IEEE754** ve verzích
  - jednoduchá (32 bitů)
  - dvojnásobná přesnost (64 bitů)
  - Nově (IEEE 754-2008) i poloviční (16 bitů – především pro hry a barvy), a čtyřnásobná (128 bitů) a osminásobná přesnost (256 bitů) pro speciální vědecké výpočty
- V programovacím jazyce C se proměnné s jednoduchou a dvojnásobnou přesností deklarují jako **`float`** a **`double`**.

Formát čísla v pohyblivé řádové čárce

Kód mantisy: přímý kód — znaménko a absolutní hodnota

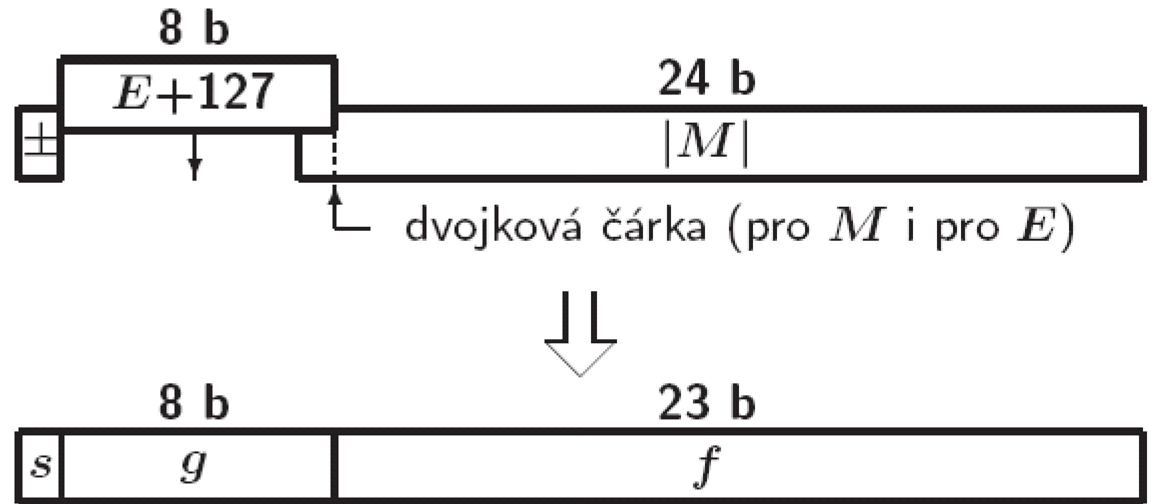
Kód exponentu: aditivní kód (s posunutou nulou - pro float posun +127, pro double +1023).

Znaménko M



Poloha řádové čárky (tečky) pro E i M

ANSI/IEEE Std 754-1985 — jednoduchý formát — 32b



ANSI/IEEE Std 754-1985 — dvojnásobný formát — 64b

$g \ldots 11b$ $f \ldots 52b$

Reprezentace/kódování čísla v pohyblivé řádové čárce

- Kód mantisy: přímý kód – znaménko a absolutní hodnota
- Kód exponentu: aditivní kód (s posunutou nulou) (K=127 pro jednoduchou přesnost)
- Implicitní počáteční jednička může být pro normalizovanou mantisu vynechaná $m \in \langle 1, 2)$ rozlišení 23+1 implicitní bit pro jednoduchou přesnost

$X = -1^{s}\, 2^{A(E)-127}\, m$ kde $m \in \langle 1, 2)$

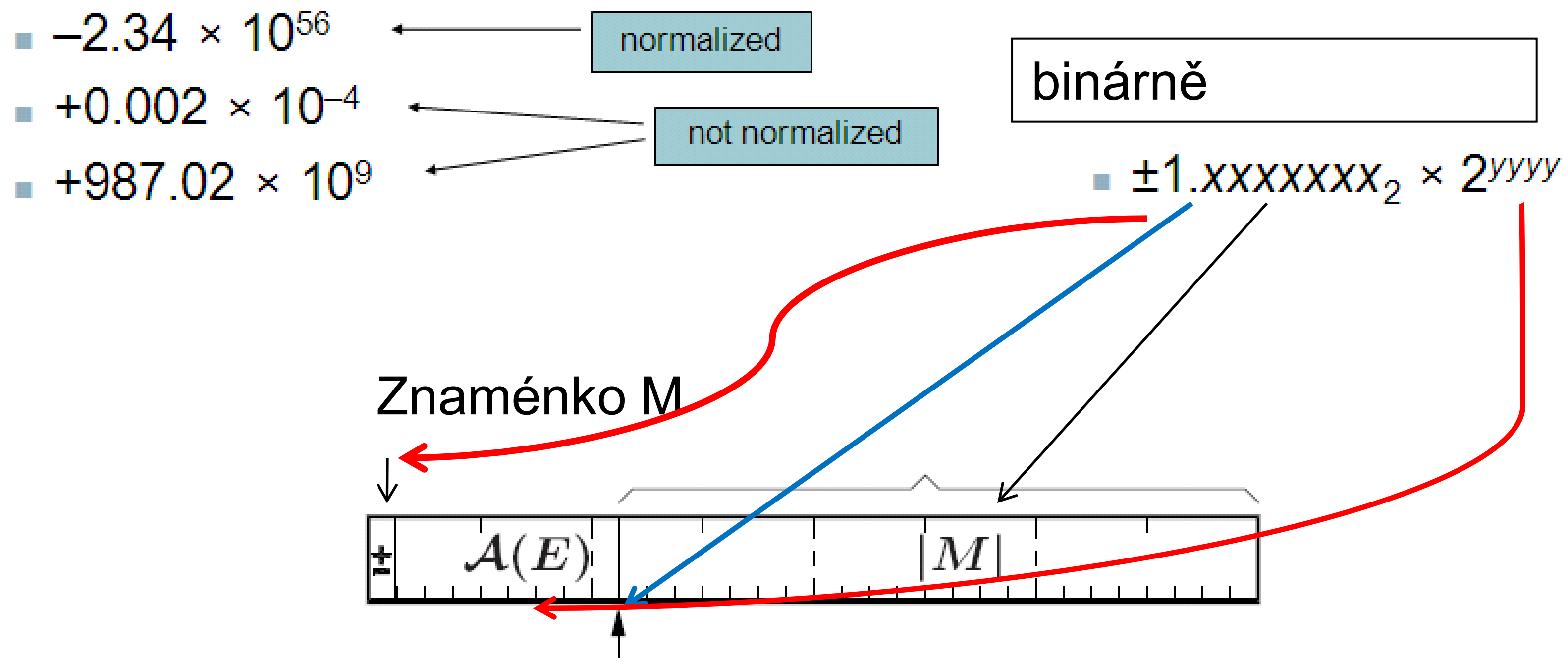
$m = 1 + 2^{-23} M$

znaménko M

$\pm$ | $\mathcal{A}(E)$ | $|M|$

Poloha řádové čárky (tečky) pro E i M

Příklady



IEEE-754 konverze float

- Převeďte -12.625_{10} IEEE-754 float formát.
- Krok #1: Převeďte $-12.625_{10} = -1100.101_2 = 101 / 8$
- Krok #2: Normalizujte $-1100.101_2 = -1.100101_2 * 2^3$
- Krok #3:

Vyplňte pole, znaménko je záporné -> S=1.

Exponent + 127 -> 130 -> 1000 0010 .

Úvodní bit 1 mantisy je skrytý ->

1 1000 0010 . 1001 0100 0000 0000 0000 0000 000

Příklad: 0.75

$0.75_{10} = 0.11_2 = 1.1 \times 2^{-1} = 3/4$

$1.1 = 1.F \rightarrow F = 1$

$E - 127 = -1 \rightarrow E = 127 - 1 = 126 = 01111110_2$

$S = 0$

00111111010000000000000000000000 =
0x3F400000

Příklad 0.1_{10} - Převeďte na float

$0.1_{10} = 0.00011\underline{0011....}_2$

$= 1.1\underline{0011}_2 \times 2^{-4} = 1.F \times 2^{E-127}$

$F = 1\underline{0011}$ $\quad -4 = E - 127$

$E = 127 - 4 = 123 = 01111011_2$

0011 1101 1100 1100 1100 1100 1100 1100 *1100* *11..*

0x3DCCCCCD, proč je D ?

Speciální hodnoty NaN, +Inf a -Inf

- Pokud není výsledek matematické operace pro daný vstup definovaný (log -1) nebo je výsledek nejednoznačný 0/0, +Inf - +Inf tak je uložena hodnota NaN (Not-a-Number) – exponent nastavený na samé jedničky, mantisa nenulová
- Výsledkem operací, které pouze přetečou z rozsahu 1/0 (=+Inf), +Inf + +Inf (= +Inf) atd., je reprezentovaný hodnotou nekonečno (+Inf nebo -Inf) – exponent samé jedničky, mantisa nuly

NaN

kladné	0 11111111 *mantisa !=0*	**NaN**

Nekonečno

kladné	0 11111111 00000000000000000000000	**+Inf**
záporné	1 11111111 00000000000000000000000	**-Inf**

Skrytý bit

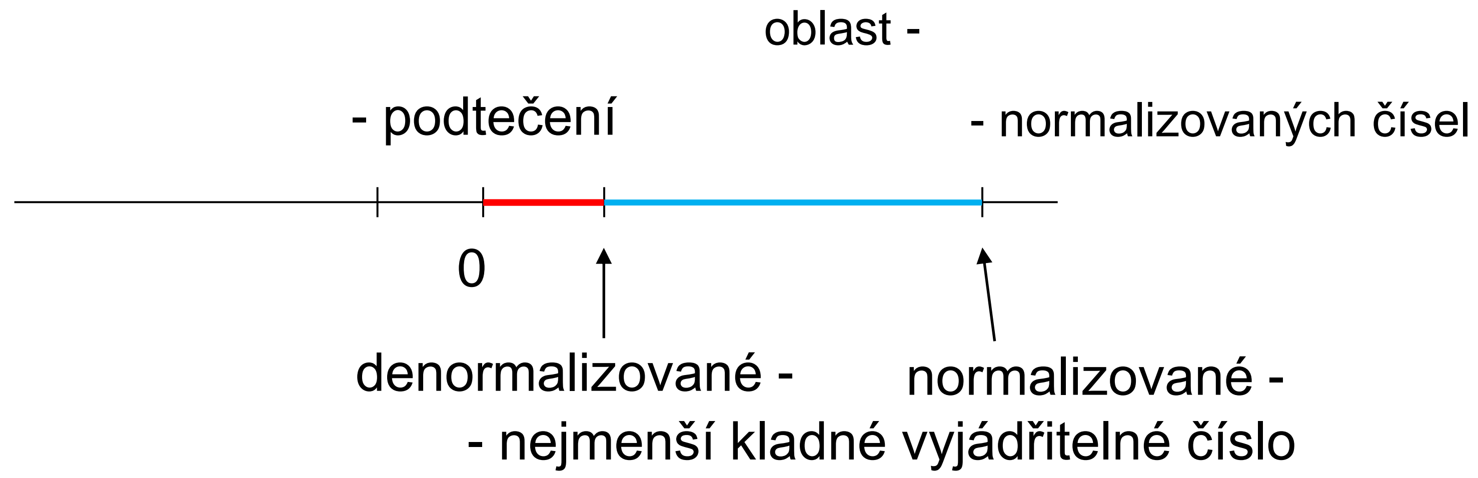
- Nejvyšší platný bit mantisy (který se do bitové reprezentace operandu neukládá) je závislý na hodnotě obrazu exponentu.
- Jestliže je obraz exponentu **nenulový**, je tento bit 1, mluvíme o **normalizovaných** číslech.
- Na druhou stranu jestliže je obraz exponentu **nulový**, je skrytý bit 0. Pak mluvíme o **denormalizovaném** čísle, viz dále.

Denormalizované číslo?

- Smyslem zavedení denormalizovaných čísel je rozšíření reprezentovatelnosti čísel, která se nacházejí blíže k nule, tedy čísel velmi malých (v následujícím obrázku oblast označena modře).
- Denormalizovaná čísla mají nulový exponent a i skrytý bit před řádovou čárkou je implicitně nulový.
- Cenou je nutnost speciálního ošetření případu nulový exponent, nenulová mantisa -> *denormalizovaná čísla podporují jen některé implementace.*
  (Intel ko-procesory mají)

Implicitní (skrytá) počáteční jednička

- Pro každé normalizované číslo je nejvýznamnější bit mantisy jedna a není ho potřeba ukládat (rezervovat pro něj místo)
- Pokud je reprezentace exponentu 0 (-K) nebo pokud je číslo „denormalizované“, tak je prostor pro uložení mantisy využitý pro hodnotu včetně počáteční jedničky nebo nuly
- Denormalizovaná čísla umožňují zachovat rozlišení v rozsahu od nejmenšího normalizovaného čísla směrem k nule



Denormalizované číslo?

Denormal computations use hardware and/or operating system resources to handle denormals; these can cost hundreds of clock cycles.
Denormal computations take much longer to calculate than normal computations.

There are several ways **to avoid denormals** and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush denormals to zero.

[Source: https://software.intel.com/en-us/node/523326]

Příklady reprezentace některých důležitých hodnot

Nula

kladná	`0 00000000 00000000000000000000000`	`+0.0`
záporná	`1 00000000 00000000000000000000000`	`-0.0`

Nekonečno

kladné	`0 11111111 00000000000000000000000`	`+Inf`
záporné	`1 11111111 00000000000000000000000`	`-Inf`

Hraniční hodnoty pro jednoduchý formát

Největší normalizované	`0 11111110 11111111111111111111111`	$(2-2^{-23})\,2^{127}$ $+3.4028\ 10^{+38}$
Nejmenší normalizované	`* 00000001 00000000000000000000000`	$\pm 2^{(1-127)}$ $\pm 1.1755\ 10^{-38}$
Největší denormalizované	`* 00000000 11111111111111111111111`	$\pm(1-2^{-23})\,2^{-126}$
Nejmenší denormalizované	`* 00000000 00000000000000000000001`	$2^{-23}\ 2^{-126}$ $\pm 1.4013\ 10^{-45}$

Širší přehled

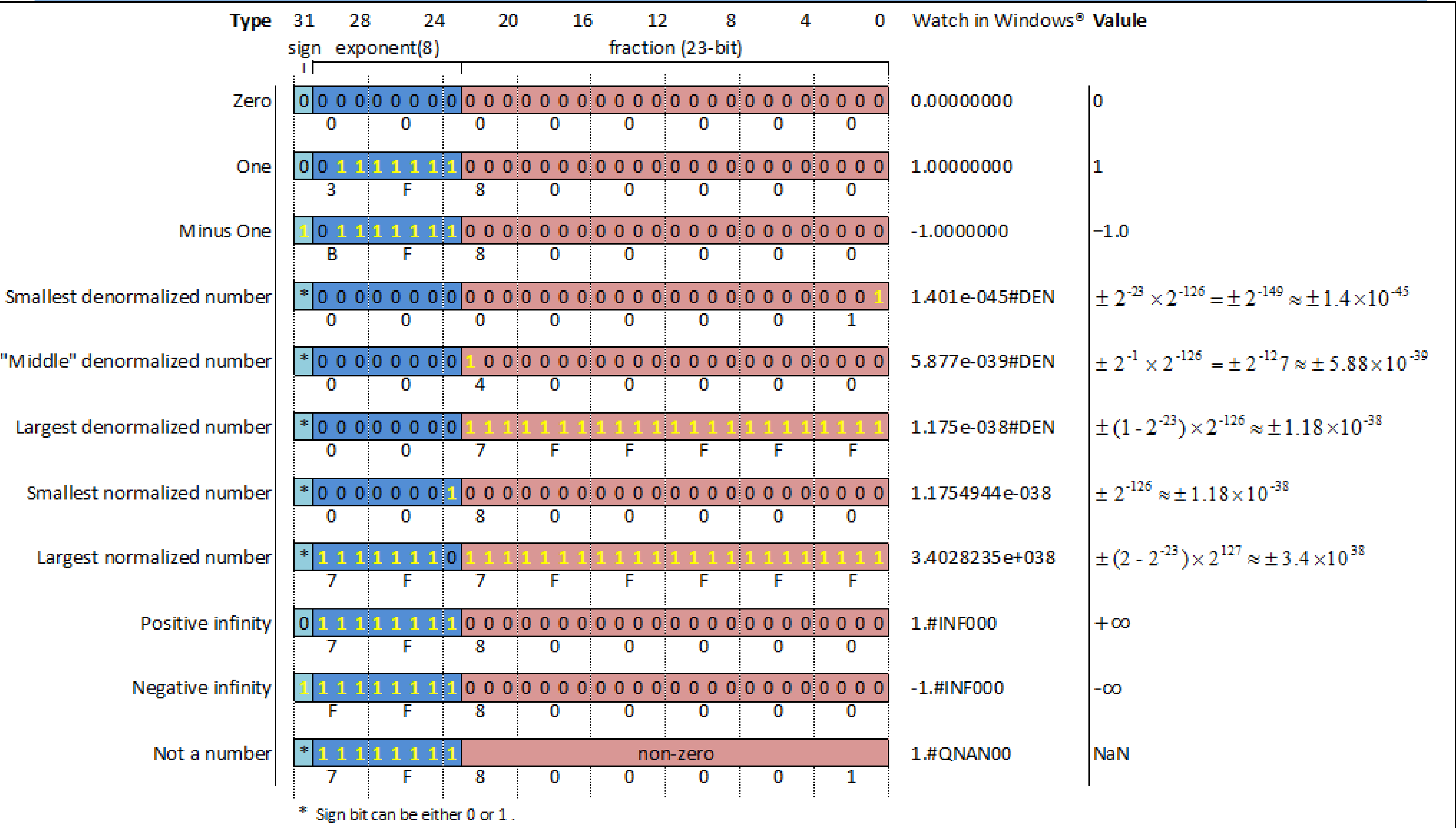


Figure: Floating-point Binary

Short and Long IEEE 754 Formats: Features

Some features of ANSI/IEEE standard floating-point formats

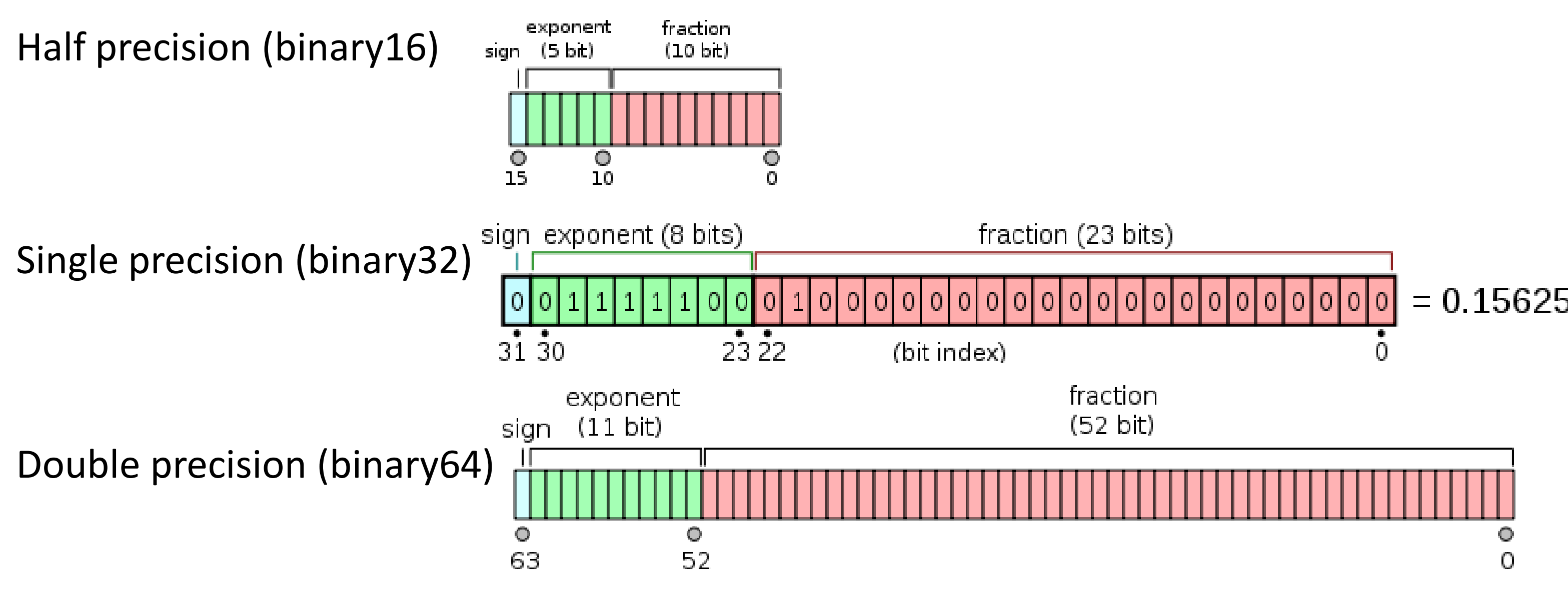
Feature	Single/Short	Double/Long
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (±0)	e + bias = 0, $f = 0$	e + bias = 0, $f = 0$
Denormal	e + bias = 0, $f \neq 0$ represents $\pm 0.f \times 2^{-126}$	e + bias = 0, $f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity (±∞)	e + bias = 255, $f = 0$	e + bias = 2047, $f = 0$
Not-a-number (NaN)	e + bias = 255, $f \neq 0$	e + bias = 2047, $f \neq 0$
Ordinary number	e + bias $\in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^{e}$	e + bias $\in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^{e}$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

IEEE 754 Formats

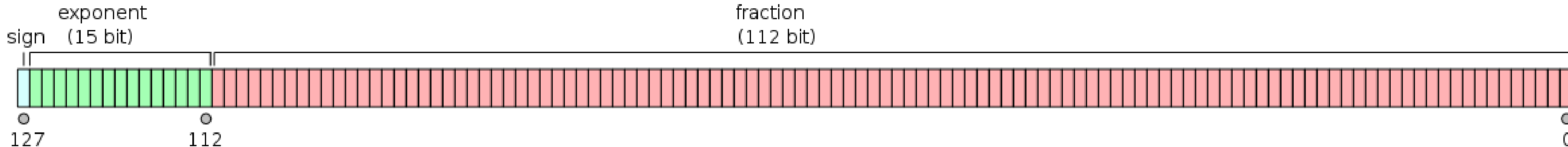
Half precision (binary16)

Single precision (binary32)

Double precision (binary64)

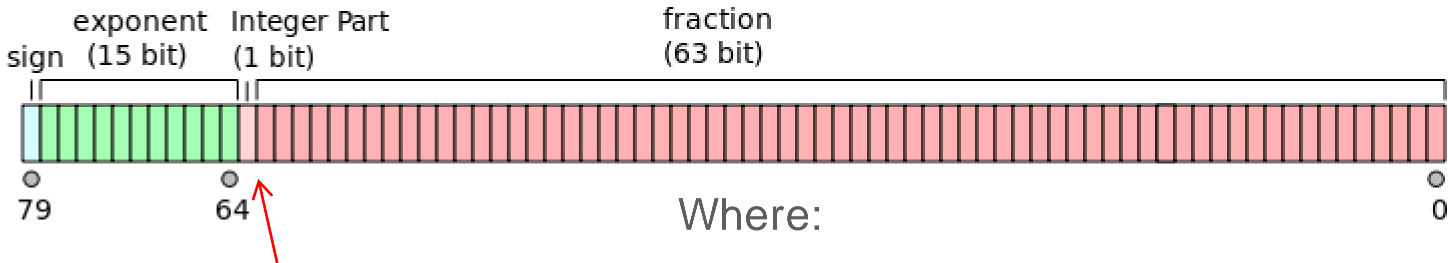


Quadruple precision (binary128)



Source: Herbert G. Mayer, PSU

X86 Extended precision (80 bits)



Bit 1. není skrytý!

Where:

- b = the bias
- n = the number of bits in the exponent

More simply, the biases are shown in the table below:

$$b = \frac{2^n}{2} - 1$$

Or, equivalently:

$$b = (2^{n-1}) - 1$$

Type	Bits	Bias
Half	5	15
Single	8	127
Double	11	1023
Extended	15	16383
Quad	15	16383

Source: Herbert G. Mayer, PSU

*Reálná čísla

a jejich zobrazení v počítači

Způsoby uložení vícebytových čísel v paměti

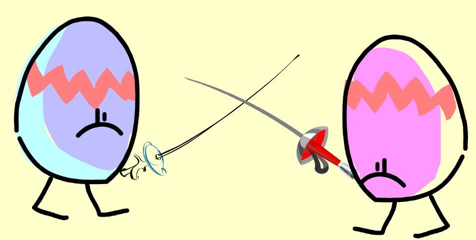
Hex číslo: 1234567

Big Endian - downto

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian - to

		0x100	0x101	0x102	0x103		
		67	45	23	01		



Little-Endien *pochází z knihy Gulliverovy cesty, Jonathon Swift 1726, v níž označovalo jednu ze dvou znepřátelených frakcí Lilliputů. Její stoupenci jedli vajíčka od užšího konce k širšímu, zatímco* ***Big Endien*** *postupovali opačně. A válka nedala na sebe dlouho čekat…*

Pamatujete si, jak válka skončila?

Úvodní cvičení

```
/* Simple program to examine how are different data types encoded in memory */
#include <stdio.h>
/** The macro determines size of given variable and then
* prints individual bytes of the value representation */
#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

void print_mem(unsigned char *ptr, int size)
{   int i;
    printf("address = 0x%08lx\n", (long unsigned int)ptr);
    for (i = 0; i < size; i++)
    {  printf("0x%02x ", *(ptr + i));  }
    printf("\n");
}
```

Úvodní cvičení

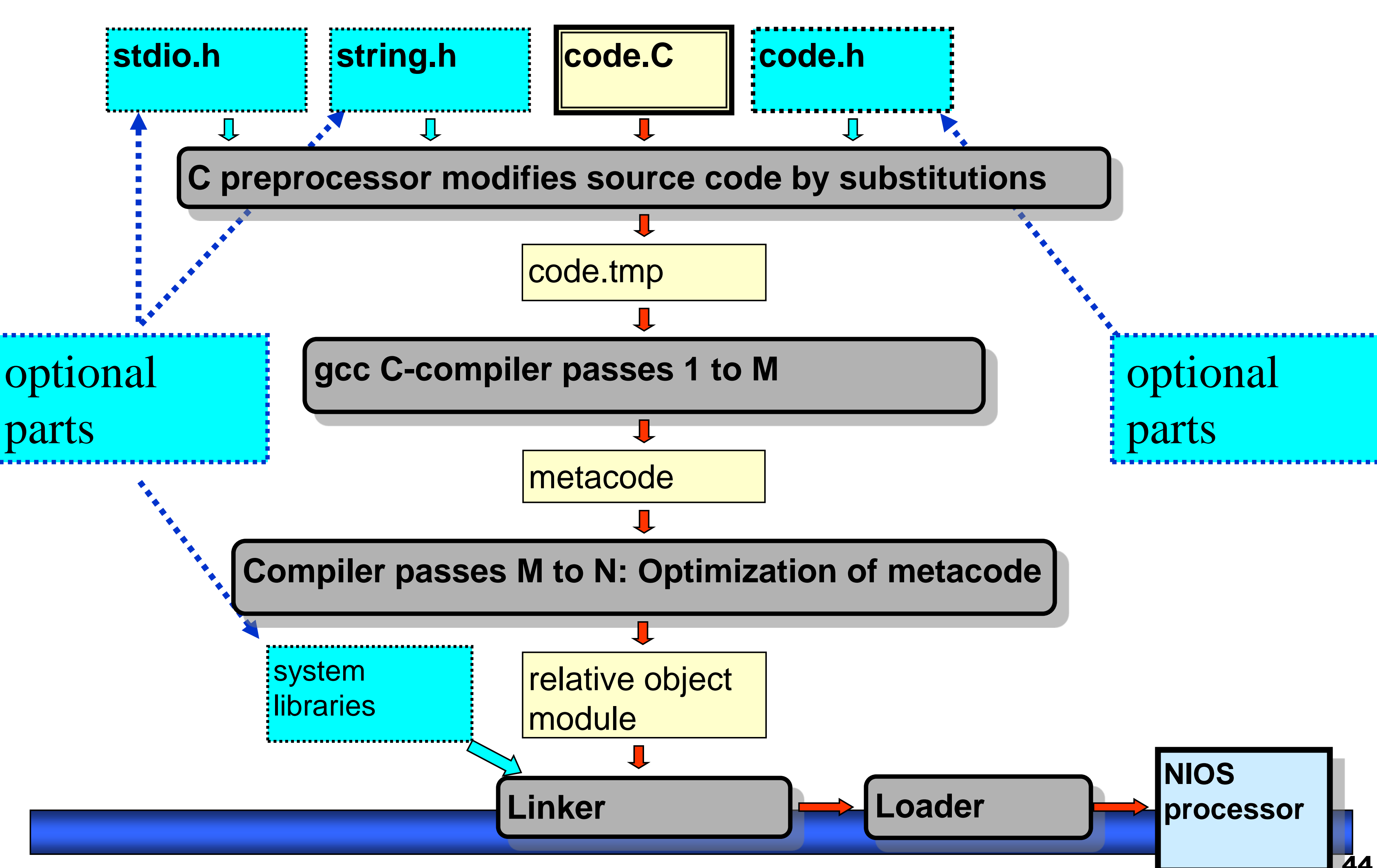
```
int main()
{ /* try for more types: long, float, double, pointer */
  unsigned int unsig = 5;
  int sig = -5;

/* Read GNU C Library manual for conversion syntax for other types */
/* https://www.gnu.org/software/libc/manual/html_node/Formatted-Output.html */
  printf("value = %d\n", unsig);
  PRINT_MEM(unsig);

  printf("\nvalue = %d\n", sig);
  PRINT_MEM(sig);

return 0;
}
```

Basic Steps of C Compiler



C primitive types

Size	Java	C	C alternative	Range
1	boolean	any integer, true if !=0	BOOL[1]	0 to !=0
8	byte	char[2]	signed char	–128 to +127
8		unsigned char	BYTE[1]	0 to 255
16	short	int	signed short	–32768 to +32767
16		unsigned short		0 to + 65535
32	int	int	signed int	-2^31 to 2^31-1
32		unsigned int	DWORD[1]	0 to 2^32-1
64	long	long	long int	-2^63 to 2^63-1
64		unsigned long	LWORD[1]	0 to 2^64-1

1) In many implementations, it is not a standard C datatype, but only common custom for user's "#define" macro definitions, see next slides

2) Default is signed, but the best way is to specify.

Definition of BYTE and BOOL

// by substitution rule no ; and no type check

```
#define BYTE unsigned char
#define BOOL int
```

// by introducing new type, ending ; is required

- typedef unsigned char BYTE;
- typedef int BOOL;

C language has no strict type checking #define ~ typedef,
but typedef is usually better integrated into compiler.

Defining a Parameterized Macro

#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

Similar to a C function, preprocessor macros can be defined with a parameter list; parameters are without data types.

Syntax:

```
#define MACRONAME(parameter_list) text
```

No white space before (.

Examples:

```
#define MAXVAL(A,B) ((A) > (B)) ? (A) : (B)

#define PRINT(e1,e2)
printf(”%c\t%d\n”,(e1),(e2));

#define putchar(x) putc(x, stdout)
```

#define PRINT_MEM(a) print_mem((unsigned char*)&(a), sizeof(a))

Side-effects!!!

Example:

```
#define PROD1(A,B) A * B
```

Wrong result:

```
PROD1(1+3,2) → 1+3 * 2
```

Improved example with ()

```
#define PROD2(A,B) (A) * (B)

PROD2(1+3,2) → (1+3) * (2)
```

Pointer Operators

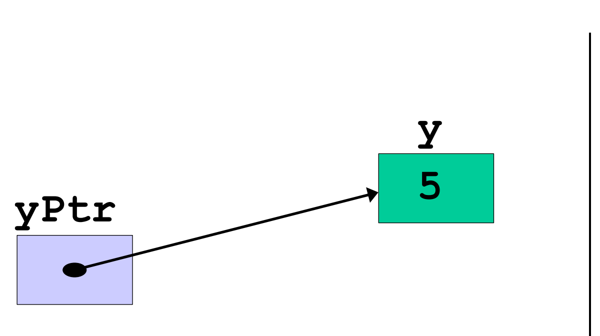
& (address operator)

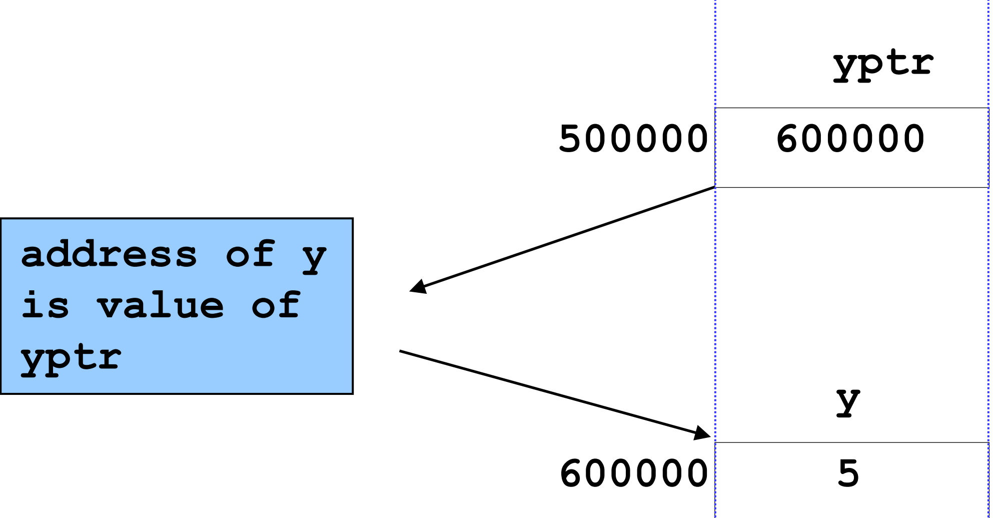
Returns the address of its operand

Example

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
```

yPtr "points to" **y**





Pointer Operators

& (address operator)

Returns the address of its operand

* dereference address

Get operand stored in address location

***** and **&** are inverses
(though not always applicable)

Cancel each other out

```
*&myVar == myVar
```

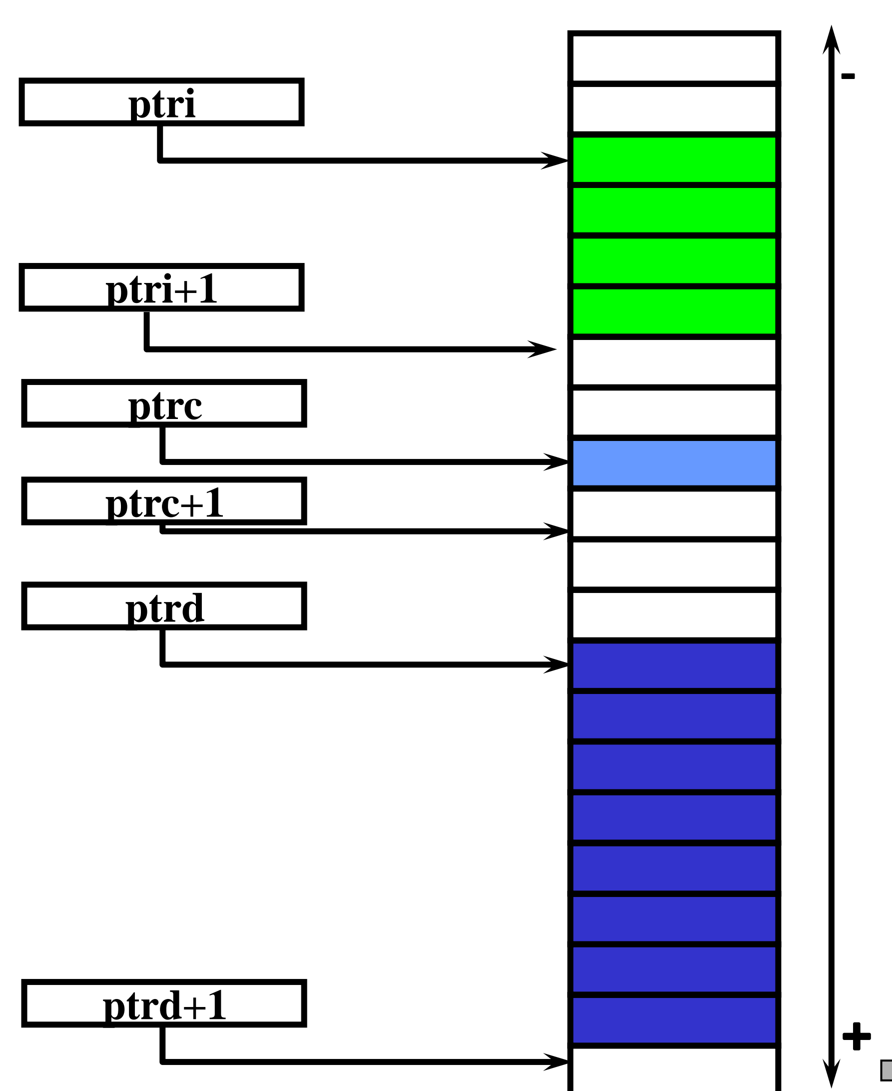
and

```
&*yPtr == yPtr
```

Size of Pointer in C-kod

```
int * ptri;
char * ptrc;
double * ptrd;
```

*ptrx ≡ ptrx[0]
*(ptrx+1) ≡ ptrx[1]
*(ptrx+n) ≡ ptrx[n]
*(ptrx-n) ≡ ptrx[-n]




```
nr1 = sizeof (double);
nr2 = sizeof (double*);
```

nr1 != nr2

Překvapení na závěr ???

```
int main()  { float x; double d;
x = 116777215.0;
   printf("%.3f\n", x);        // 116777216.000
   printf("%.3lf\n", x);       // 116777216.000 - pro float/double nemá l význam
   printf("%.3g\n", x);        // 1.17e+08
   printf("%.3e\n", x);        // 1.168e+08
   printf("%lx %f\n", x, x);  // 0 0.00000   -  Jak kdy - l nemusí vždy znamenat 64 bitový long
   printf("%llx %f\n", x, x); // 419bd78400000000 116777216.000000
   printf("%lx %f\n", *(long *)&x, x); // 4cdebc20 116777216.00000
x = 116777216.3;  printf("%.3f\n", x); // 116777216.000  - float ořízne mantisu
d = 116777216.3;  printf("%.3f\n", d); // 116777216.300
x = 116777217.0;  printf("%.3f\n", x); // 116777216.000
x = 116777218.0;  printf("%.3f\n", x); // 116777216.000
x = 116777219.0;  printf("%.3f\n", x); // 116777216.000
x = 116777220.0;  printf("%.3f\n", x); // 116777216.000
x = 116777221.0;  printf("%.3f\n", x); // 116777224.00
return 0;
}
```