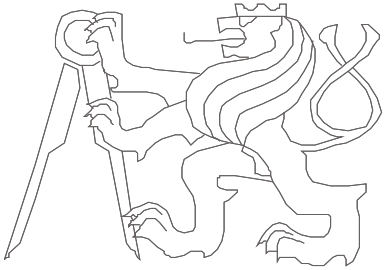
Architektura počítačů

Počítačová aritmetika a úvod

Richard Šusta, Pavel Píša

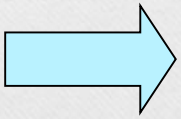
Verze 3.2 z 24.2.2020 – opravené překlepy
a zkráceno dle odpřednášené látky



České vysoké učení technické, Fakulta elektrotechnická

Vysvětlivky

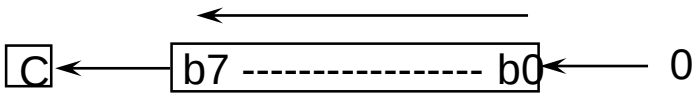
Značka 3S v levém horním rohu označuje "Skrytý Snímek pro Samostudium", který se na přednášce zpravidla nepromítal. Buď shrnuje výklad, nebo ho rozšiřuje.

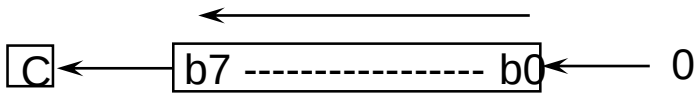


Šipka v pravém dolním rohu znamená pouhou pomocnou značku pro přednášejícího, že existuje ještě další část animace snímku.

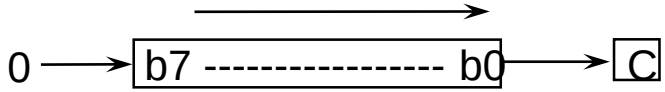
Opakování: Aritmetické posuny

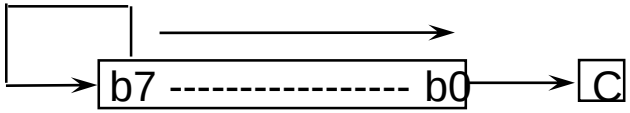
Prerekvizita APOLOS – kapitola 3.2.6





násobení 2 x





dělení 2 unsigned

dělení 2 signed

C značí Carry, avšak mají ho jen některé procesory, Intel ano, MIPS ne

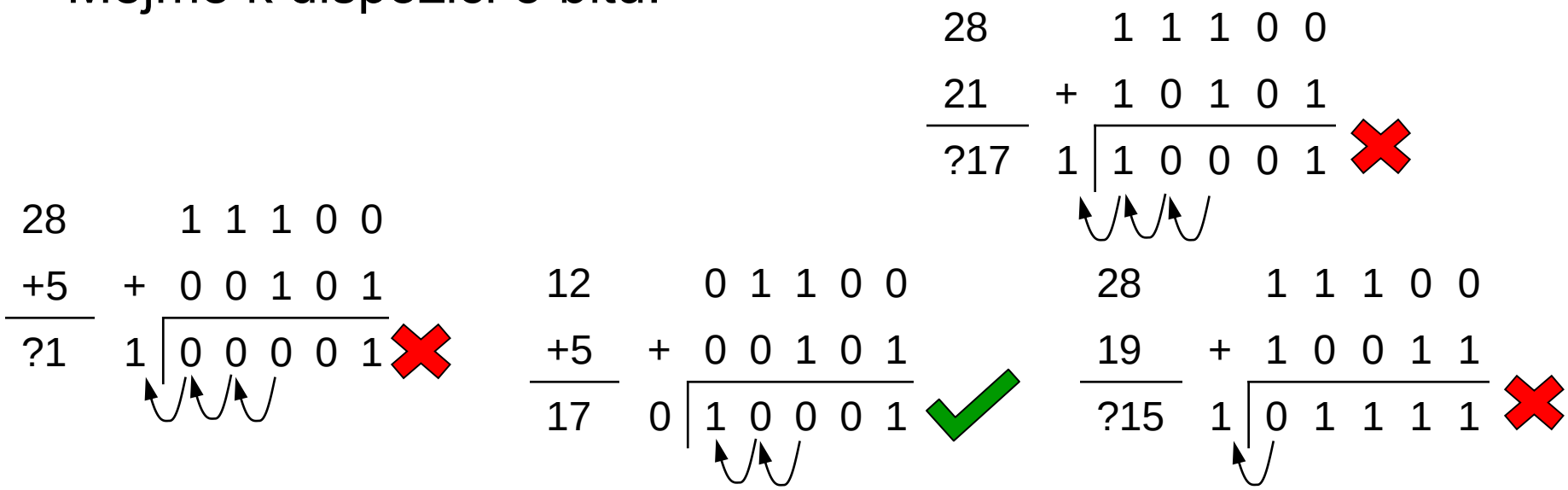
Rychlost integer operací

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

Test 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:

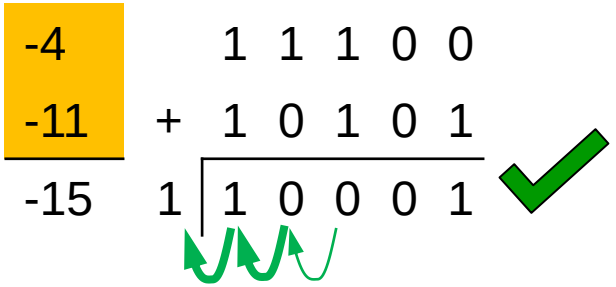


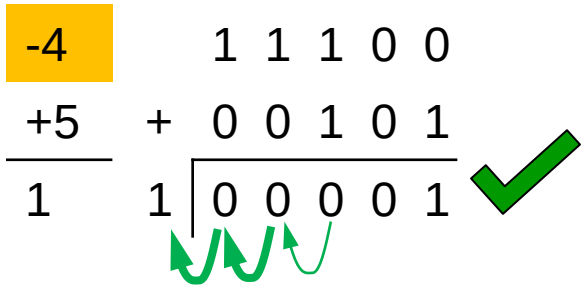
Chybný výsledek je vždy menší než oba sčítance!

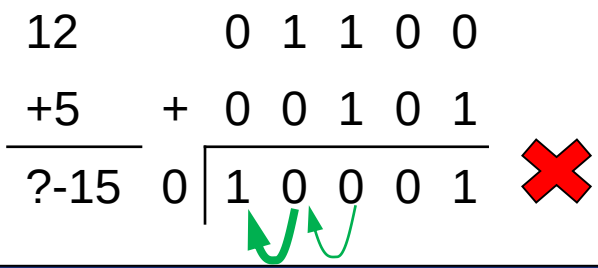
Přetečení signed

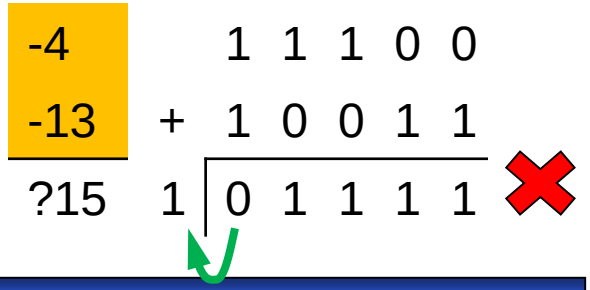
- Přeplnění - říká se tomu také **přetečení (overflow)**.
- Situace, kdy výsledek operace není správný, protože se nevešel do zobrazitelného rozsahu.
- **Nastává v situaci, kdy znaménko výsledku je jiné, než znaménka operandů, byla-li stejná**,
- jiná metoda spočívá v nonekvivalenci (xor) přenosů **do**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

Mějme k dispozici 5 bitů:









HW dělička – jeden algoritmus dělení

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

jen poslední operace musí mít už obnovení

1 1 1 : 0 1 1

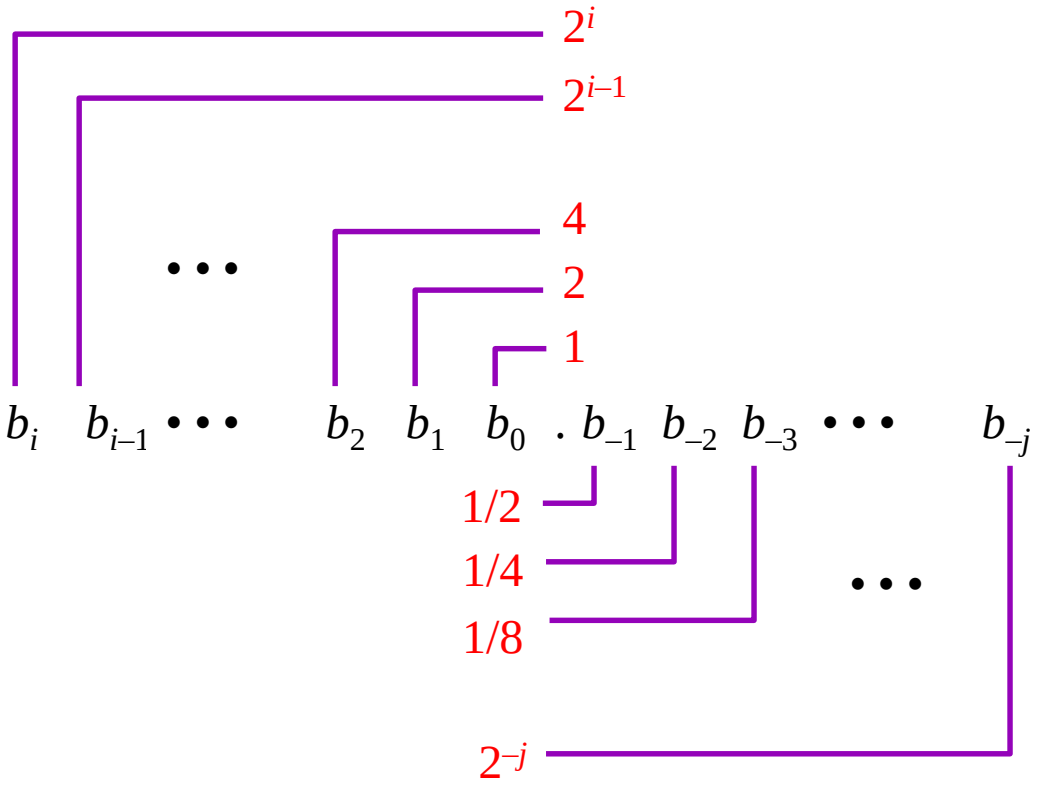
```
  0 0 0 1 1 1   :   0 0 1 1
⊟   1 1 0 0   :   :   negace
          1   :   :   horká 1
  0 1 1 1 0   :   :   −  ⇒  0
      ↓ ↓ ↓ ↓   :
      1 1 0 1   :
⊞     0 0 1 1   :
  1 0 0 0 0 1       +  ⇒  1
        ↓ ↓ ↓ ↓
        0 0 0 1
⊟       1 1 0 0
              1
    0 1 1 1 0       −  ⇒  0
↩     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. Zjistí to až odečtením, a případně pak musí výsledek zkorigovat přičtením.

Reálná čísla

a jejich zobrazení v počítači

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



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

Reprezentace reálného čísla

bity ležící vpravo od “binary point” udávají zlomky mocnin 2

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

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

$= 2 + 0.25 = 2.25$

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 mají stejná pravidla jako počítání se zlomky

Dělení 2 provedeme posunem vpravo.

Násobení 2 provedeme posunem vlevo.

Čísla pod $0.111111\ldots_2$ jsou menší než 1.0

$$1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$$

Přesná notace $\rightarrow 1.0 - \varepsilon$

Jak se v počítači zobrazují čísla typu REAL?

- Vědecká, neboli semilogaritmická notace.
  - Dvojice: EXPONENT (E),
    ZLOMKOVÁ část (též zvaná mantisa M).
  - **Mantisa x zaklad$^{\text{Exponent}}$**
- Normalizovaná notace:
  - zlomková část vždy začíná binární číslicí 1,
  - mantisa leží v intervalu <1, základ)
- Dekadicky: $7{,}26478 \times 10^{3}$
- Binárně: $1{,}010011 \times 2^{1001}$

Vědecká notace dekadické versus binární číslo

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}

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

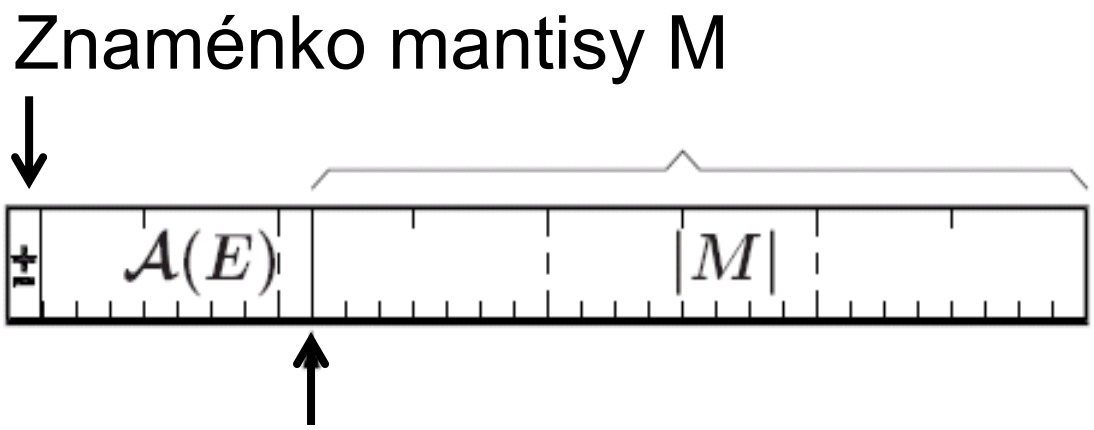
Uložení čísla dle normy IEEE754

- jednoduchá přesnost (32 bitů)
- dvojnásobná přesnost (64 bitů)
- *nově (IEEE 754-2008) má i poloviční přesnost především kvůli grafice (16 bitů), čtyřnásobnou přesnost (128 bitů) a osminásobná přesnost (256 bitů) na speciální vědecké výpočty*
- V programovacím jazyce C se proměnné jednoduché a dvojnásobné přesnosti deklarují jako **`float`** a **`double`**.

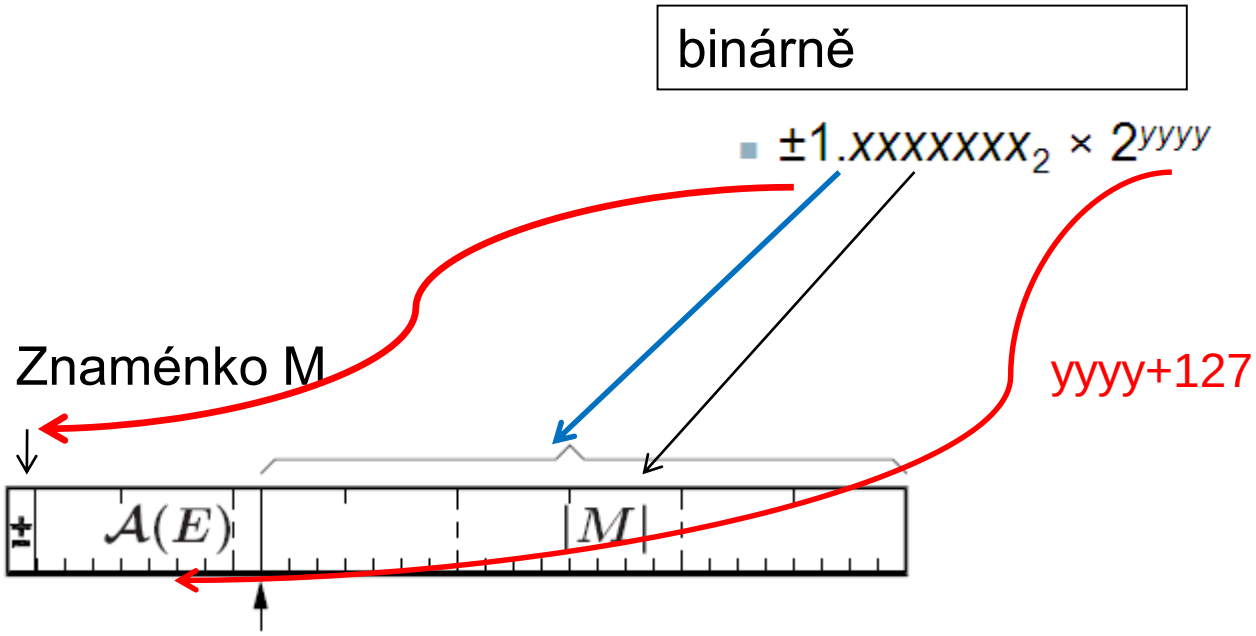
Formát čísla v pohyblivé řádové čárce

Kód mantisy: **přímý kód** — znaménko a absolutní hodnota, bit mantisy před binární . se neukládá.

Kód exponentu: **aditivní kód** (s posunutou nulou)



Přehled uložení float



Poloha řádové čárky (tečky) pro E i M

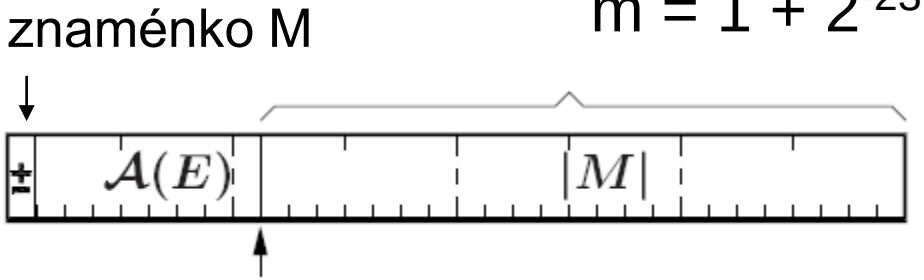
Reprezentace/kódování čísla v pohyblivé řádové čárce

- Kód mantisy: **přímý kód** – znaménko a absolutní hodnota
- Kód exponentu: **aditivní kód** (s posunutou nulou)
  float má posun +127, double +1023.
- **Implicitní počáteční jednička** se u normalizované mantisy **vynechává** M ∈ ⟨1, 2) a rozlišení float je 23+1, kde +1 značí implicitní bit, rozlišení double 52+1

kde m ∈ ⟨1, 2)

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

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



Poloha řádové čárky (tečky) pro E i M

IEEE-754 konverze float

Převeďte -12.625_{10} na IEEE-754 float formát.

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

  Vyplňte pole znaménka, zde je záporné -> S=1.

  Exponent + 127 -> 130 -> 1000 0010 .

  Úvodní bit 1 mantisy je skrytý ->

1 1000 0010 . 1001 0100 0000 0000 0000 000

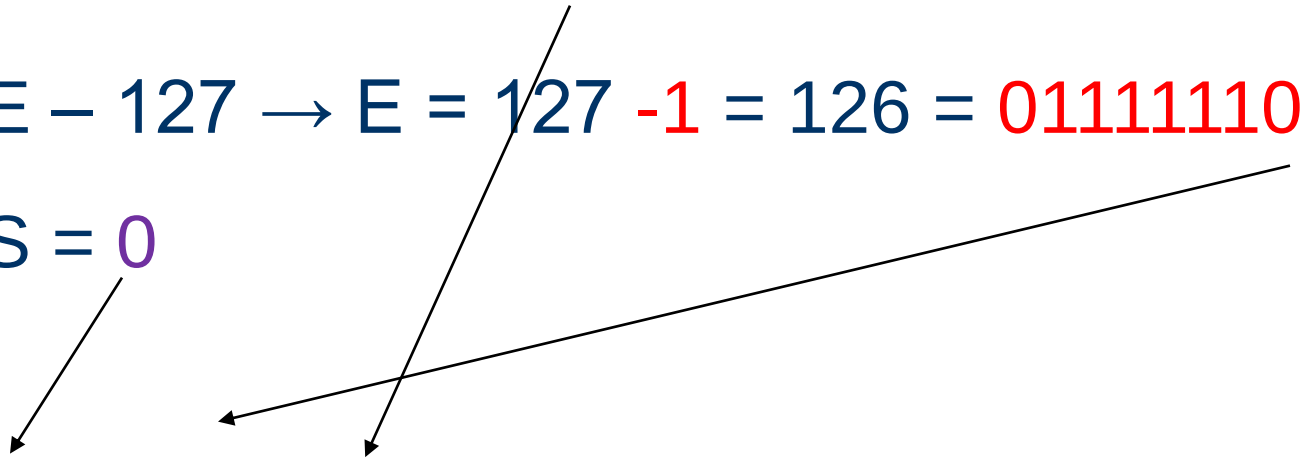
Příklad: 0.75

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

$1.1 = 1.\ m \rightarrow m = 1$

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

$S = 0$



$00111111010000000000000000000000 =$
$0x3F400000$

Příklad 0.1_{10} - převod na float

$0.1_{10} = 0.00011\underline{0011....}_2$ - $0.1*2^n$ není nikdy celé číslo

$= 1.\underline{10011}_2 \times 2^{-4} = 1.m \times 2^{-4}$

M = $1\underline{0011}$

E = 127 - 4 = 123 = 01111011_2

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

0x3DCCCCCD, proč je D ?

Příklad 0.1_{10} jako reálné číslo

$0.1_{10} = 0.\underline{0001100}11...._2 =$

0.0001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011...

Nepřesné zobrazení mnoha čísel

Konečné dekadické číslo ➔ nekonečné binární číslo

Příklad:

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

$0.1_{10} = 0.0001100110011001100 11\ldots_2$

$= 0.0\underline{00110011}_2$ (nekonečný řetězec bitů)

Více bitů pouze zpřesní reprezentaci 0.1_{10}

Reprezentace reálných čísel

Omezení

lze přesně vyjádřit jen čísla $x/2^k$

a ostatní čísla se ukládají jako nepřesná

Hodnota	*Reprezentace*
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Speciální hodnoty NaN, +Inf a -Inf

- Není-li výsledek matematické operace definovaný, jako třeba výpočet log(-1), nebo je výsledek nejednoznačný 0/0, tak se uloží hodnota NaN (Not-a-Number)
  = exponent se nastaví na samé 1 a **mantisa bude nenulová.**
- Výsledek operací, které pouze přetečou z rozsahu, se reprezentuje hodnotou nekonečno (+Inf nebo -Inf)
  = exponent se nastaví na samé 1 a **mantisa má samé 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, závisí na hodnotě obrazu exponentu.
- **Nenulový** obraz exponentu má tento bit 1 a jde o **normalizované** číslo.
- **Nulový** obraz exponentu má skrytý bit 0 a určuje **denormalizované** číslo, viz dále.

Denormalizované číslo?

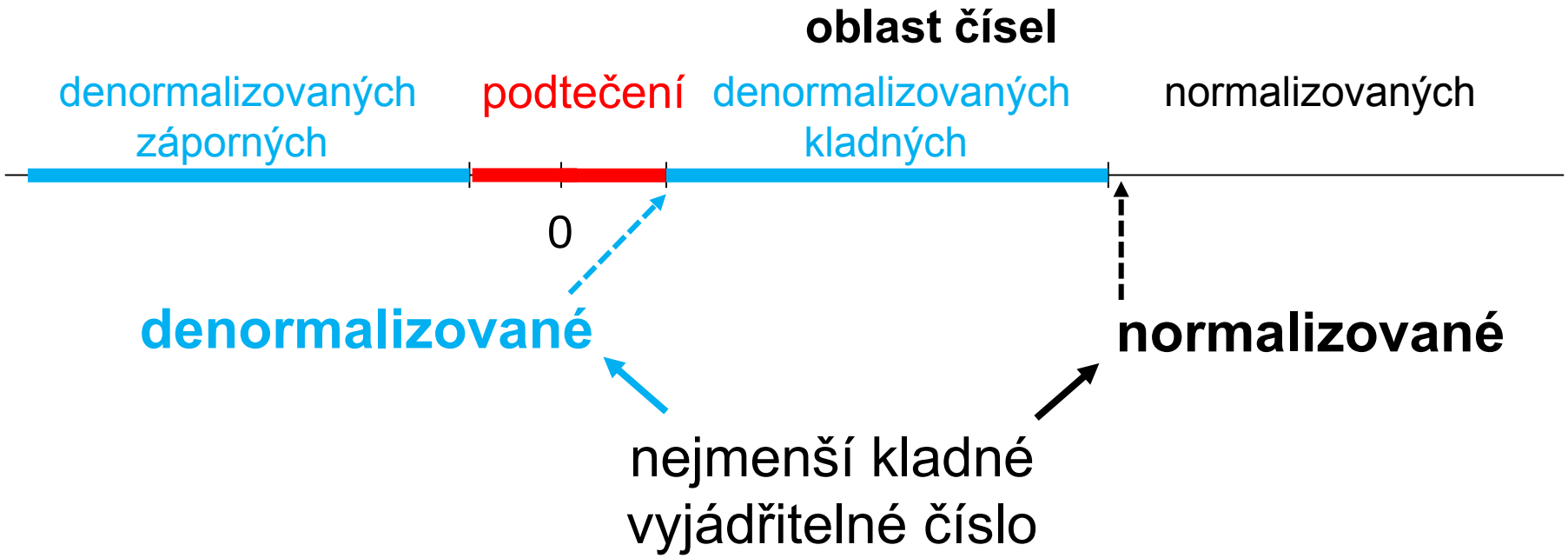
- Smyslem zavedení denormalizovaných čísel je rozšíření rozsahu malých čísel v okolí nuly (v následujícím obrázku je označeno modře).

- Denormalizovaná čísla mají nulový exponent a skrytý bit před řádovou čárkou je implicitně nulový.

- Denormalizovaná čísla vyžadují speciální výpočty a *podporují je pouze některé implementace. Intel ko-procesory je mají.*

Implicitní (skrytá) počáteční jednička

- Normalizované číslo - nejvýznamnější bit mantisy 1
- Denormalizované číslo - nejvýznamnější bit mantisy 0



Používat denormalizovaná čísla?

Výpočty používající denormalizovaná čísla se mohou prodloužit až o stovky hodinových cyklů, protože jejich provedení trvá mnohem déle.

Chcete-li zvýšit výkon své aplikace, můžete zvolit:

- měnit hodnoty jen v normalizovaném rozsahu.
- použít raději datový typ s vyšší přesností, tedy s větším rozsahem mantisy.
- příliš malá čísla nahradit nulami.

- [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 format

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

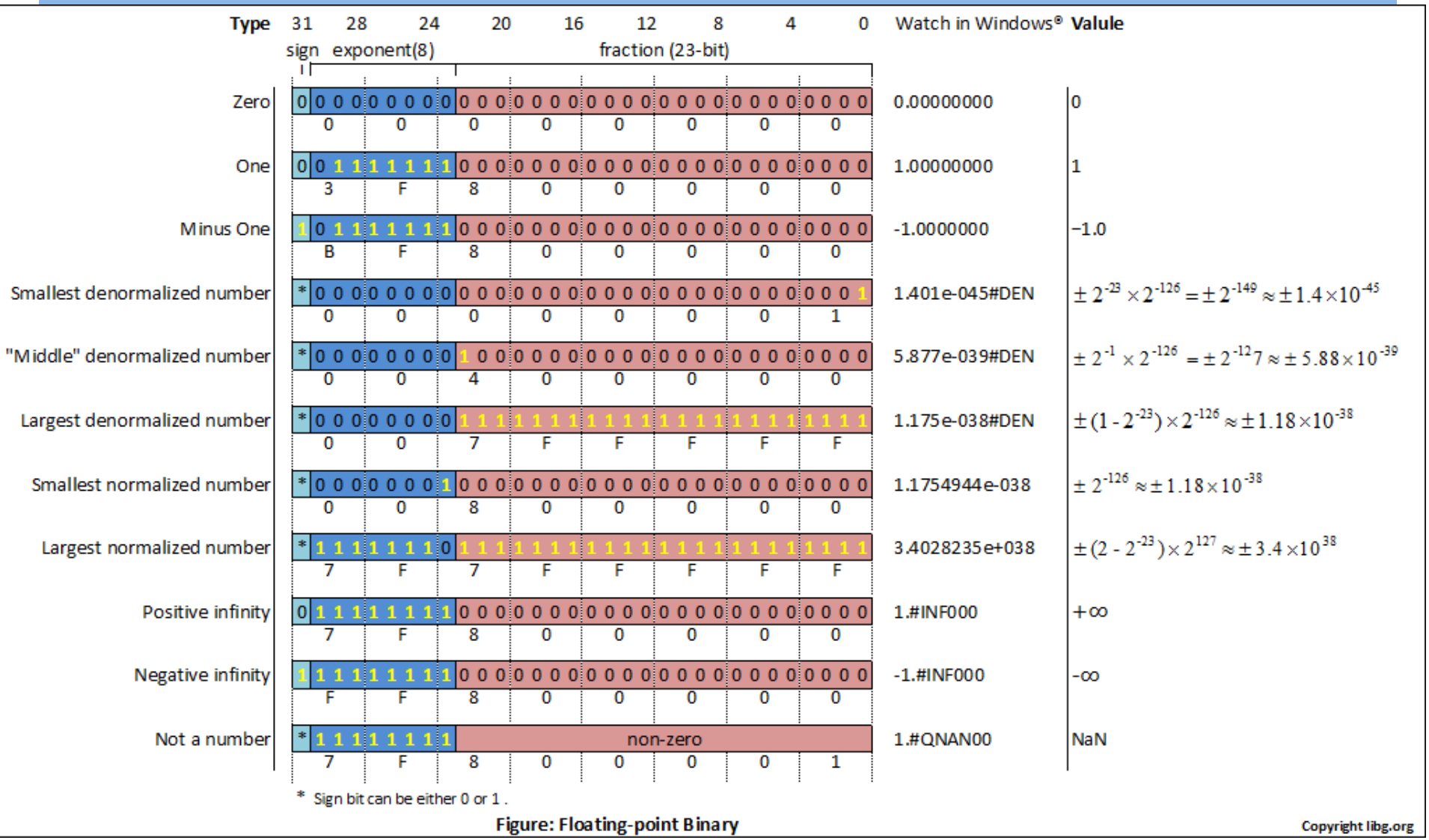


Figure: Floating-point Binary

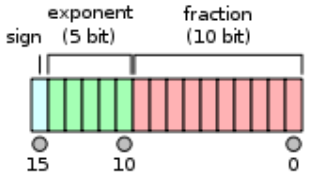
Short and Long IEEE 754 Formats: Features

Some features of ANSI/IEEE standard floating-point formats

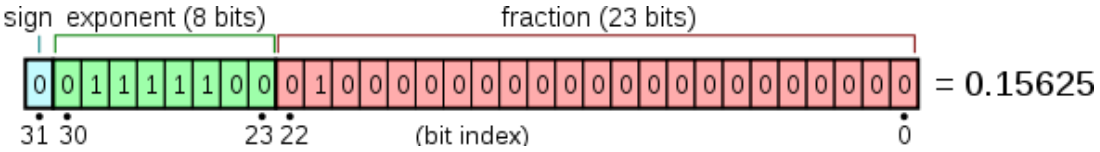
Feature	Single/Short	Double/Long
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (±0)	e + bias = 0, $f = 0$	e + bias = 0, $f = 0$
Denormal	e + bias = 0, $f \neq 0$ represents $\pm 0.f \times 2^{-126}$	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

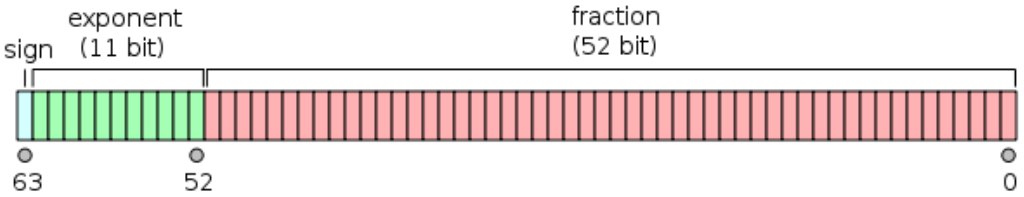
Half precision (binary16)



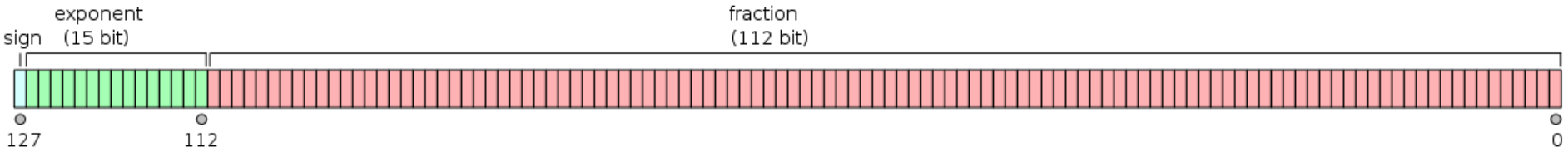
Single precision (binary32)



Double precision (binary64)

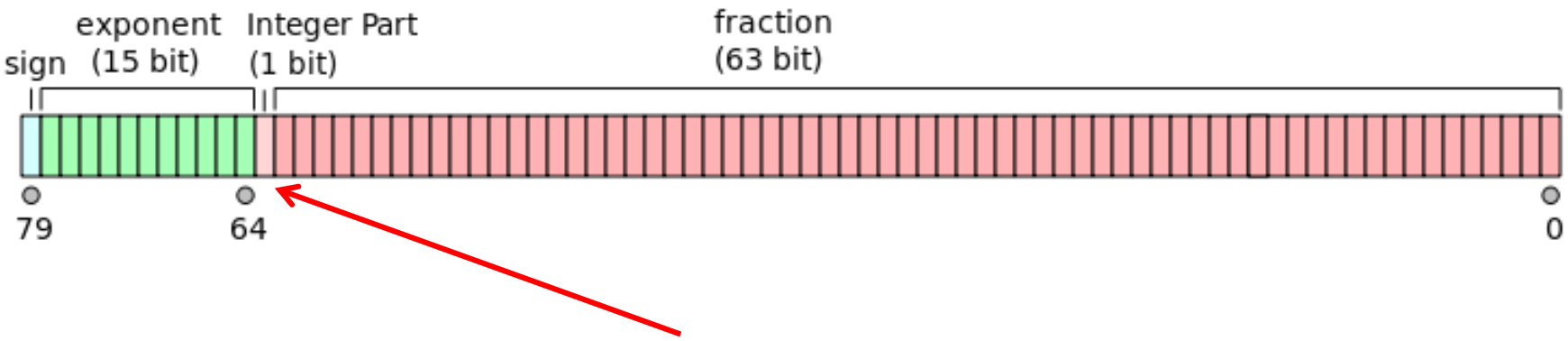


Quadruple precision (binary128)



Source: Herbert G. Mayer, PSU

X86 Extended precision (80 bits)



Bit 1. není u extended mantisy skrytý!

Pokročilá poznámka:

- *Intel procesory obsahují integrovaný aritmetický ko-procesor (už od verze 486), který počítá float i double výpočty v „extended precision“ a výsledek poté převede zpět na float/double.*
- *Nicméně Streaming SIMD Extensions (SSE) instrukce (vektorové operace), které existují v procesorech Intel od verze Pentium III, počítají vše pouze jako double, a tak přesnost výsledku může někdy záviset na tom, jak překladač zrovna přeložil náš výpočet ☺*

Source: Herbert G. Mayer, PSU

Zjednodušený přehled operací v plovoucí řádové čárce

Platí zde stejná pravidla jako u operací s mocninami

Převody: *na/z integer, double, float - pouhý posun mantisy dle exponentu*

Sčítání: $\mathbf{A \cdot 2^a}$ **,** $\mathbf{B \cdot 2^b}$ **,** $\mathbf{b < a}$

1. sjednocení exponentů posunem mantisy

$\mathbf{A \cdot 2^a + (B \cdot 2^{b-a}) \cdot z^a}$

2. sečtení + normalizace

$\mathbf{(A + (B >> (a-b))) \cdot z^a = [A + (B \cdot 2^{b-a})] \cdot 2^a}$

Odčítání: sjednocení exponentů, odečtení a normalizace

Násobení: $\mathbf{A \cdot 2^a \cdot B \cdot 2^b = A \cdot B \cdot 2^{a+b}}$

A·B +*normalizace posunem doleva, je-li třeba*

Dělení $\mathbf{A \cdot 2^a / B \cdot 2^b = A/B \cdot 2^{a-b}}$

A/B +*případná normalizace posunem doprava*

Rychlost operací s reálnými čísly

Operace	**Provedení**
Negace čísla	negace MSB (Main Scale Bit)
Porovnání	a) znaménko -> b) abs. hodnota
Násobení či dělení 2^n	změna exponentu
Převody mezi int, float, double	posun mantisy dle exponentu
Sčítání, Odčítání, Increment, decrement	Srovnání mantis na stejné exponenty, +/-, zaokrouhlení, normalizace
Násobení na hardwarové násobičce	Součet exponentů, součin mantis, zaokrouhlení, normalizace
Násobení na sekvenční násobičce	
Dělení	Rozdíl exponentů, podíl mantis, zaokrouhlení, normalizace

Příklad: a + b = 1000*pi + e/20

1.		**Float 6.5 číslic**	**na int**	<8 388 608;16 777 216) = <2^23;2^24)
af	1000*pi	~3141.593	* 2^12	**12867964**,928
bf	e/20	~0,1359141	* 2^26	**9121040**,8525824

2.	**Převedeme na binární číslo**		ale u něho **.** float
ax	**12867965**	1100 0100 0101 1001 0111 1101	za 12.bitem <- * 2^12
bx	**9121041**	1000 1011 0010 1101 0001 0001	za 26. bitem <- * 2^26

3.	**Nenormalizované binární číslo**	**exponent**
a	1100 0100 0101.1001 0111 1101	2^0
b	00.00 1000 1011 0010 1101 0001 0001	2^0

4.	**Normalizované binární číslo**	**exponent**
a	1.100 0100 0101 1001 0111 1101	* 2^11, 12+11=23
b	1.000 1011 0010 1101 0001 0001	* 2^-3, 26-3 = 23

Kontrola mezivýsledků

1.100 0100 0101 1001 0111 1101 * 2^11

= (12867965 * 2^-23) * 2^11 - *skutečně uložená hodnota*

= **1,533980**96561431884766 * 2^11

= **3141,593**01757812500000768

(Přesně 1000*pi=**3141,592**6535897932384626 4)

1.000 1011 0010 1101 0001 0001 * 2^-3

= (9121041 * 2^-23) * 2^-3

= **1.087312**81757354736328 * 2^-3

= **0.135914**10219669342041

(Přesně e/20=0,**135914**09142295226177)

Výpočty provedené na kalkulátoru ***SpeedCrunch 0.12 - http://speedcrunch.org/***

Příprava na sčítání

5.	Normalizované binární číslo	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	1.000 1011 0010 1101 0001 0001	* 2^-3

6.	Na stejné exponenty	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 ***10110100010001***	* 2^11

U čísla b je binární tečka posunutá o 14 míst vlevo, tj. rozdíl exponentů 11-(-3).
Červeně označené bity utíkají mimo rozsah -> ztráta přesnosti.

7.	Součet	expt
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 ~~***10110100010001***~~	* 2^11
a+b	1.100 0100 0101 1011 1010 1001	* 2^11

Výsledek a kontrola v double

a+b = 1.100 0100 0101 1011 1010 1001 * 2^11

= (12868521 * 2^-23) * 2^11

= **1.534047**24597930908203 * 2^11

= **3141,728**7597656249999744

Původní čísla a+b sečtená v double

=3141.593 + 0,1359141 = **3141,728**9141

= 1.10001000101101110101**1010** * 2^11 a jeho skutečná hodnota

= 3141,72900390625

Výpočet v double 1000*pi+e/20 a následný převod na float

1.100 0100 0101 1011 1010 1000* 2^11 = 3141,728515625

*Výpočty provedené na kalkulátoru **SpeedCrunch 0.12** - http://speedcrunch.org/*

Iterační dělička – Goldschmidt 1/6

$$Q = \frac{N}{B} = \frac{m_N 2^{e_N}}{m_B 2^{e_B}} = \frac{m_N}{m_B} 2^{e_N - e_b}$$

Pokud jsou čísla v normalizovaném tvaru, pak platí:

m_N = 1.???????...? a m_B = 1.???????...?

tzn. $1 \leq m_N, m_B < 2$ pokud uvažujeme celou mantisu, nebo
$0{,}5 \leq m_N, m_B < 1$ pokud bereme pouze zlomkovou část.

Uvažujme pouze zlomkovou část (za desetinnou čárkou).

Iterační dělička – Goldschmidt 2/6

* Vypočítáme 1/dělitel
  * a potom použijeme násobení k zjištění podílu
* Ignorujeme exponent
  * Počítáme $(1/P_B)$, kde P_B je mantisa dělitele
* Máme-li normalizované reálné číslo, pak
  * $1 <= P_B < 2$
  * $P_B = 1 + X$ kde $(X < 1)$

Source: IIT Delhi, McGrawHill

Iterační dělička – Goldschmidt 3/6

$$\frac{\mathbf{1}}{\boldsymbol{P_B}} = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{X}} \; (P_B = 1 + X, 0 < X < 1)$$

$$= \frac{\mathbf{1}}{\mathbf{1}+\mathbf{1}-\boldsymbol{X'}} \; (X' = 1 - X, X' < 1)$$

$$= \frac{\mathbf{1}}{\mathbf{2}-\boldsymbol{X'}}$$

$$= \frac{1}{2} * \frac{1}{1 - \frac{X'}{2}}$$

$$= \frac{\mathbf{1}}{\mathbf{2}} * \frac{\mathbf{1}}{\mathbf{1}-\boldsymbol{Y}} \; (Y = \frac{X'}{2} = (1 - X)/2, Y < \frac{1}{2})$$

Source: IIT Delhi, McGrawHill

Iterační dělička – Goldschmidt 4/6

$$\frac{1}{1-Y} = \frac{1+Y}{1-Y^2}$$
$$= \frac{(1+Y)(1+Y^2)}{1-Y^4}$$
$$= \ldots$$
$$= \frac{(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})}{1-Y^{32}}$$

$$\approx \mathbf{(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})}$$

* *Neuvažujeme už Y^{32} (připomínka $Y<0.5$), protože tak malé číslo již nelze uložit do float formátu!*

Source: IIT Delhi, McGrawHill

Iterační dělička – Goldschmidt 5/6

Výpočet $1/(1-Y)=(1+Y)*(1+Y^2)*(1+Y^4)*(1+Y^8)*(1+Y^{16})$

- Y^2 počítáme násobením.
  - opakovaným mocněním dostaneme Y^4, Y^8, a Y^{16}
  - k tomu potřebujeme 4 násobení a poté 5 sčítání k výpočtu všech členů řady, a nakonec 4 další násobení k dokončení (1/1-Y)
- Nakonec spočítáme $1/P_B$ posunem doleva

$$1/PB = \frac{1}{2} * \frac{1}{1-Y}$$

Source: IIT Delhi, McGrawHill

Iterační dělička – Goldschmidt – 6/6

- S rostoucím x ($0 < x \leq 0{,}5$) se konvergence zhoršuje. Při x == 0.5 je nejhorší.
- Modifikace Goldschmidtova algoritmu spočívá v odhadu (nepřesném) převrácené hodnoty K hodnoty m_D z look-up tabulky – podle několika málo prvních bitů (~10).
- Místo původního x=1- m_D počítáme s x=1- Km_B

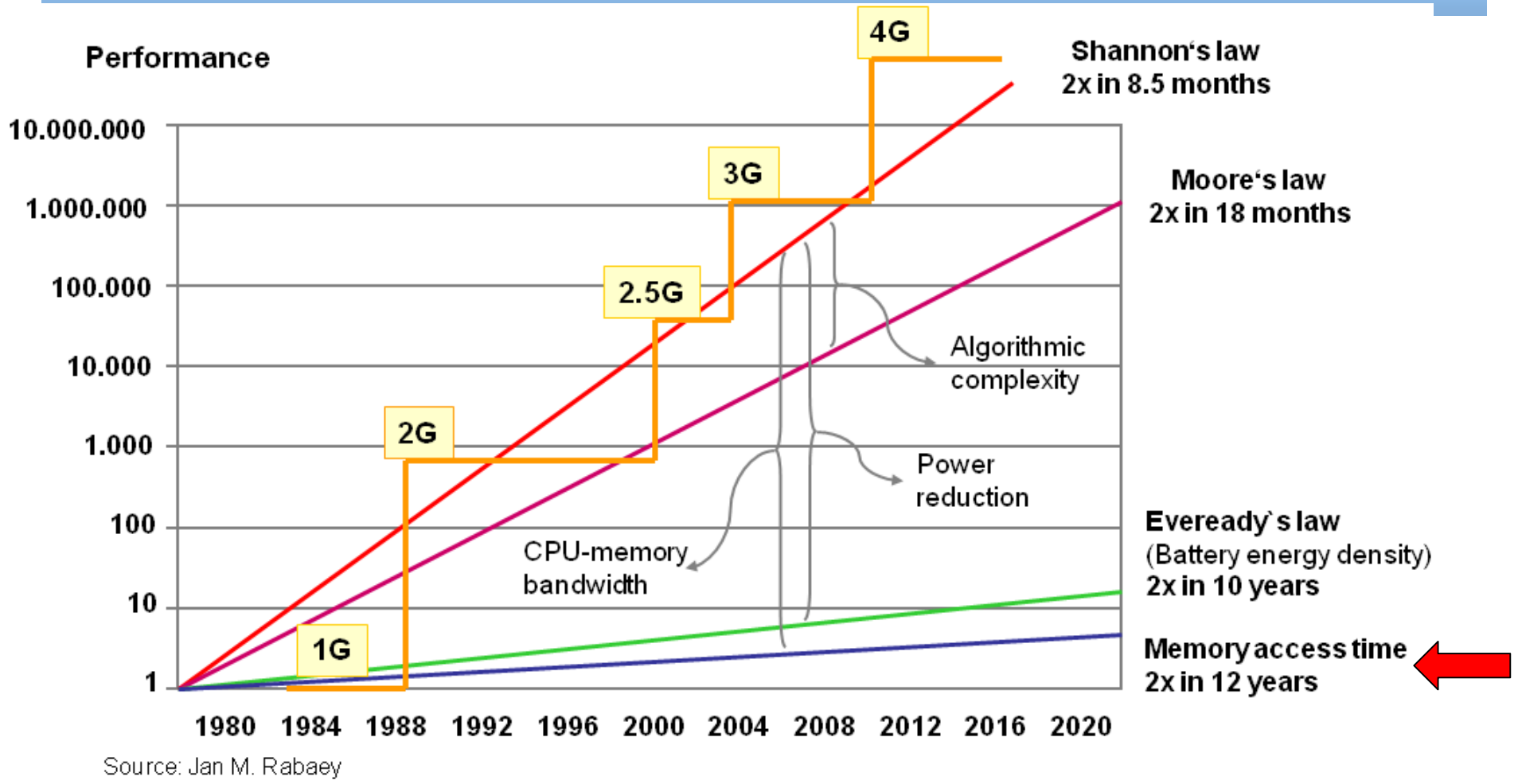
$$\frac{m_N}{m_B} \approx m_N K\left(1+x\right)\left(1+x^2\right)\ldots\left(1+x^{2^i}\right)$$

- Tato dělička se používá v moderních CPU.
- Kontrolní otázka: Můžeme tuto děličku použít i pro INTEGER?

Data v paměti

a jejich uložení v počítači

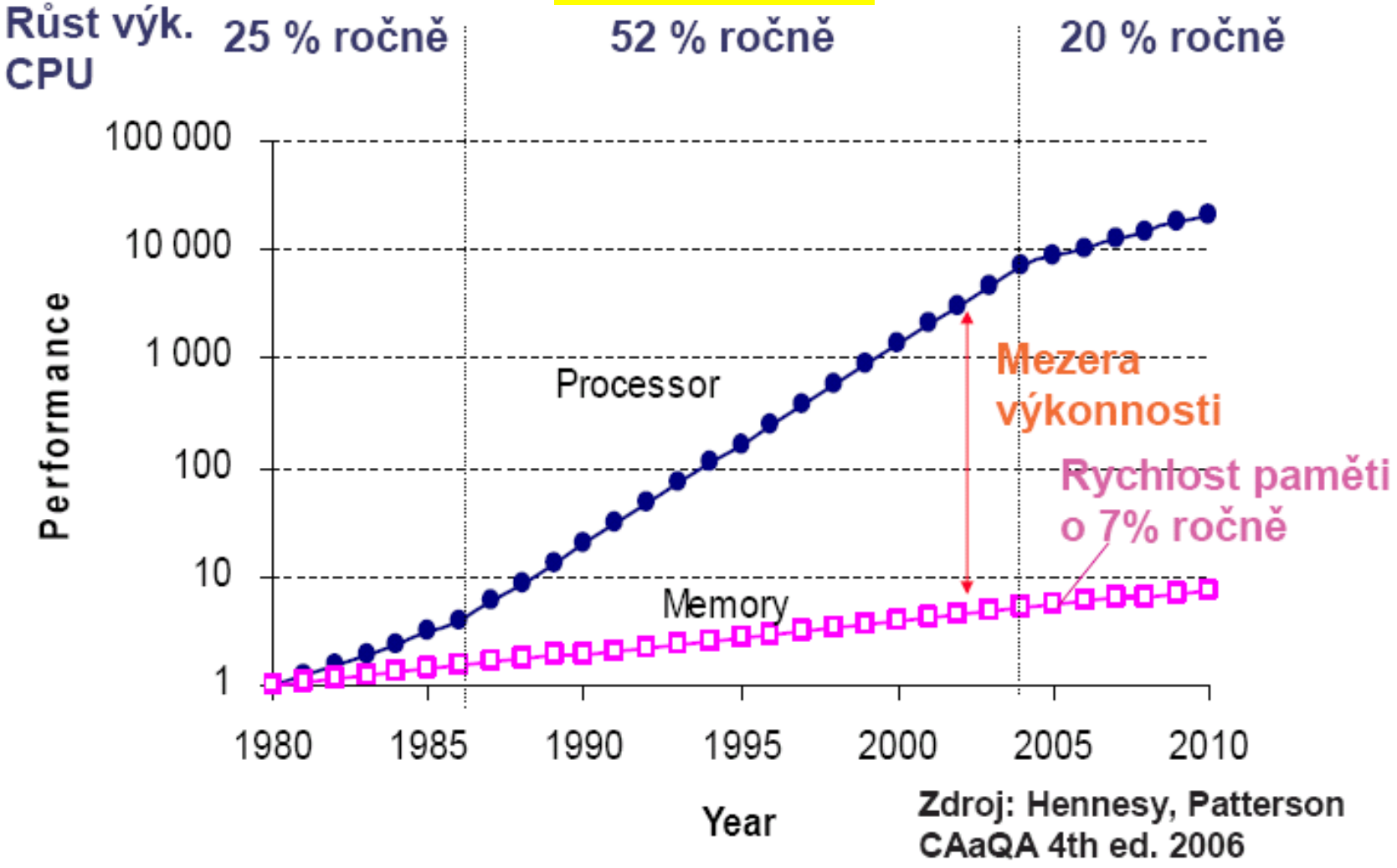
Predikce klíčových technologických mezer



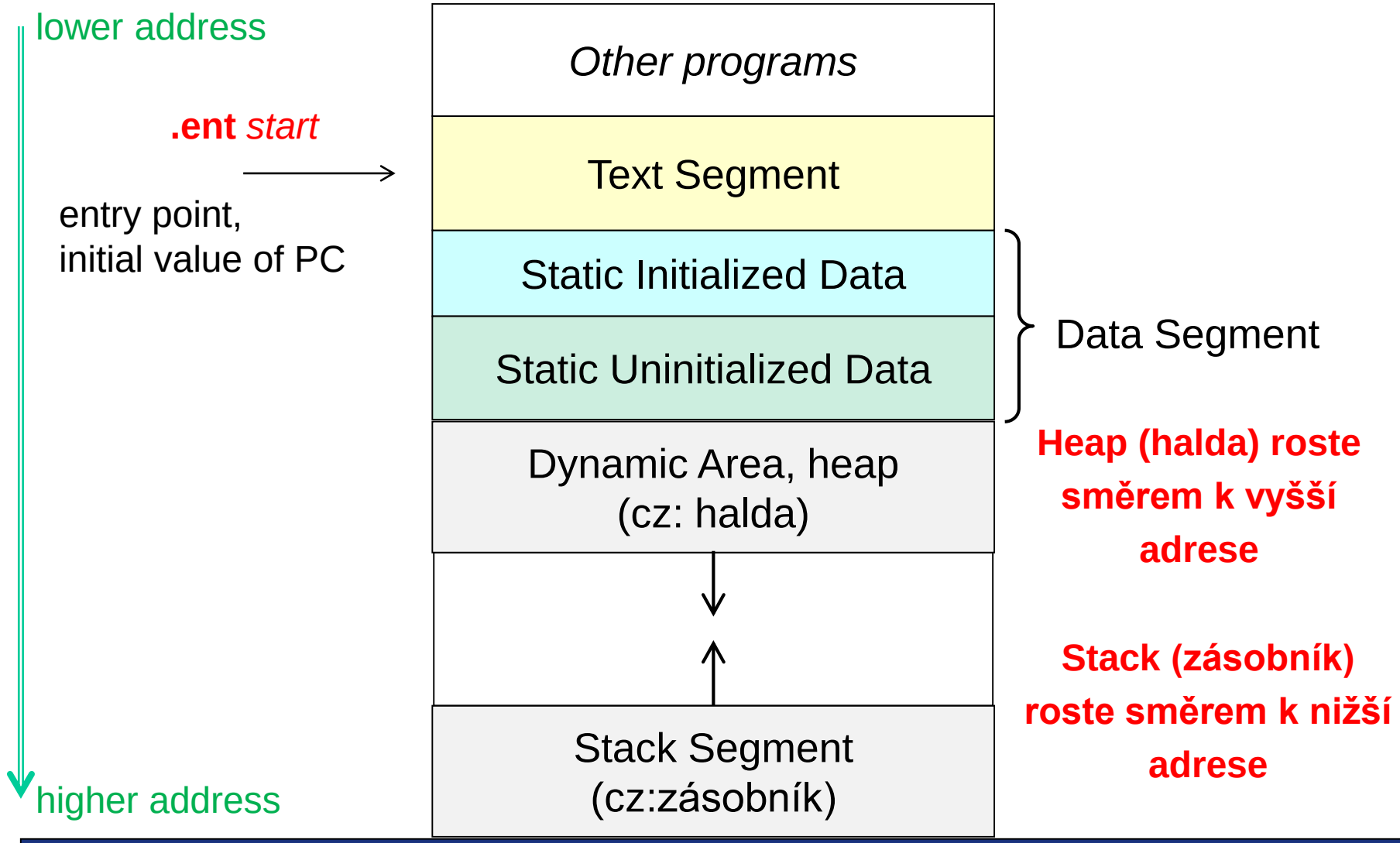
Poznámka: Nárůst složitosti algoritmů se v průběhu času formalizoval v literatuře jako tak zvaný Shannonovův zákon (Shannon's law of Algorithmic Complexity).

Disproporce ve výkonu procesoru a paměti, Moorův zákon

přesnější čísla



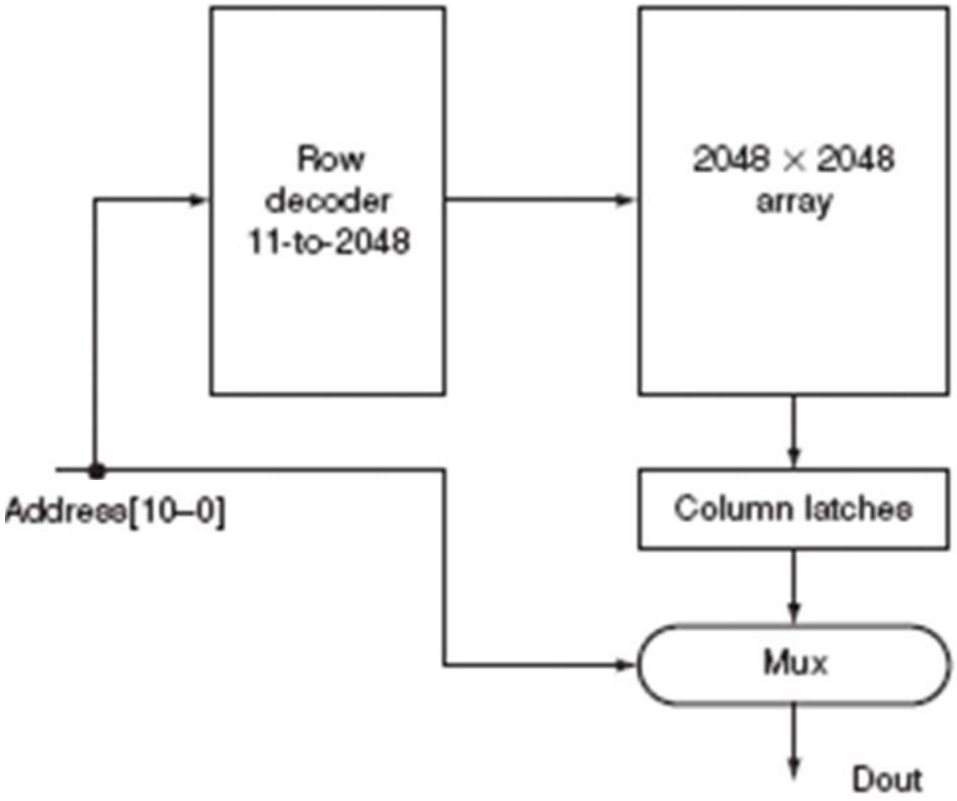
Uložení Programu v adresovém prostoru

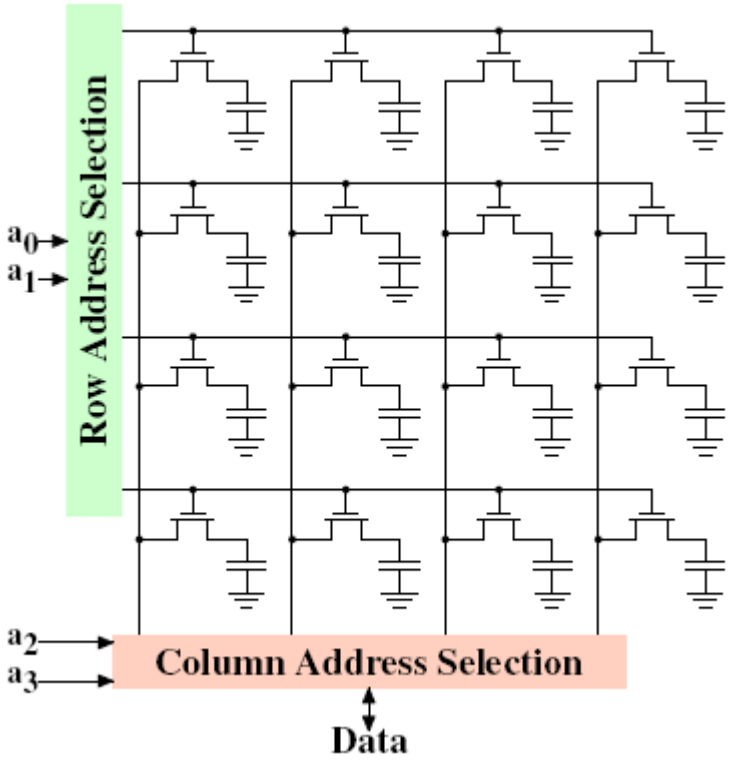


Terminologie kolem pamětí

- **Adresa**, pojem snad není třeba vysvětlovat.
- **Hodnota**, vlastní informace. Paměťová buňka však může obsahovat i přídavné údaje (třeba o platnosti hodnoty, apod.).
- Parametry pamětí:
  - **Vybavovací doba paměti**, kritický parametr. Délka časového intervalu mezi objevením se požadavku a okamžikem, kdy jsou data k dispozici.
  - Doba přístupu, zastaralý parametr; vybavovací doba + obnovení obsahu po destruktivním čtení.
  - **Propustnost**, výkonový parametr. Schopnost zpracovat uvedené množství za jednotku času.
  - Latence = zpoždění, podobně jako vybavovací doba.

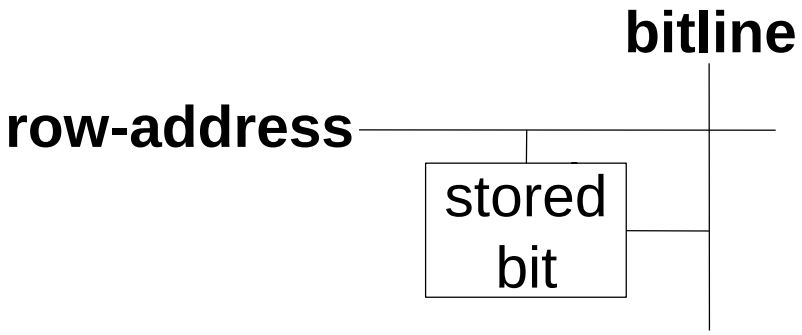
Vnitřní organizace čipu DRAM paměti

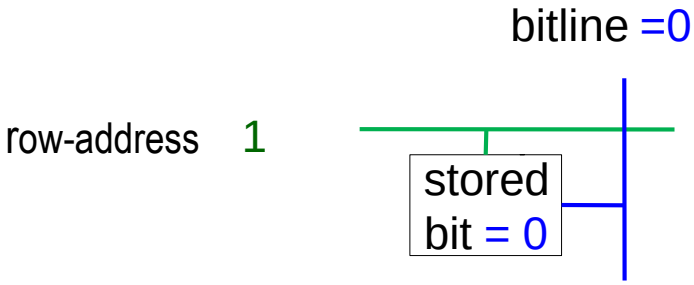


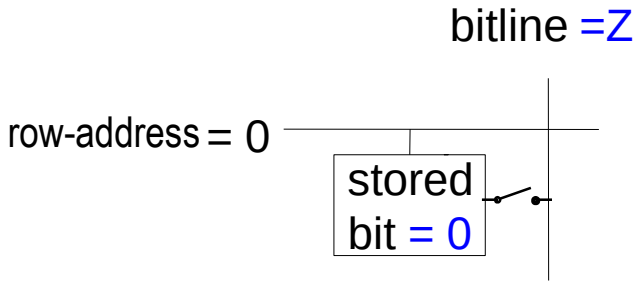


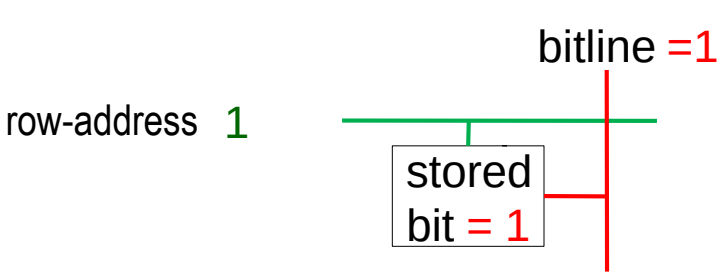
4M × 1 DRAM (Dynamic Random Access Memory)
se vnitřně realizuje jako pole 2048 × 2048 jednobitových paměťových buněk

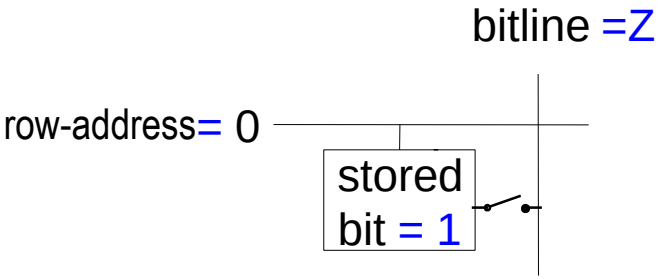
Paměťová buňka





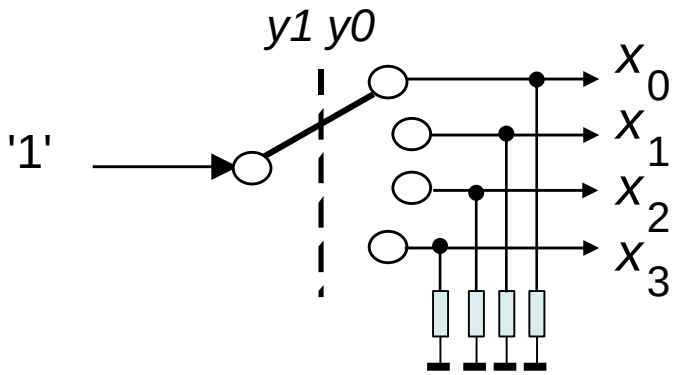


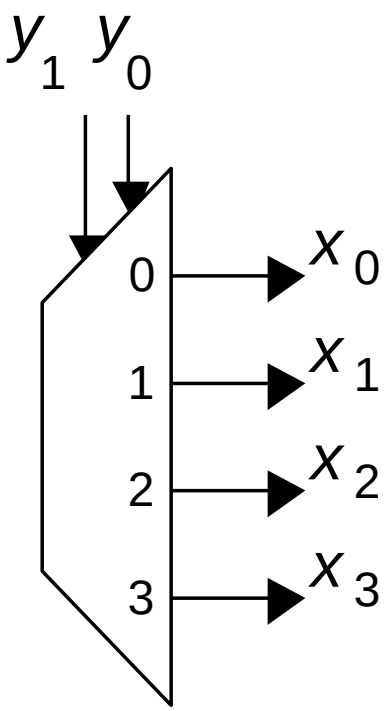




Přepínačová analogie dekodérů 1 z N

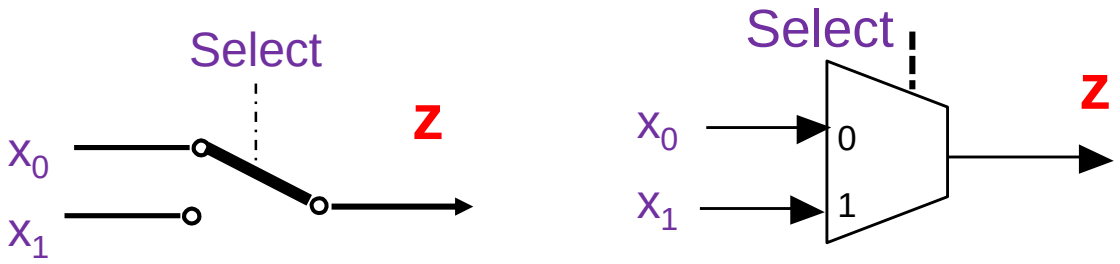
One Hot Decoder *cz: Dekodér 1 ze 4*



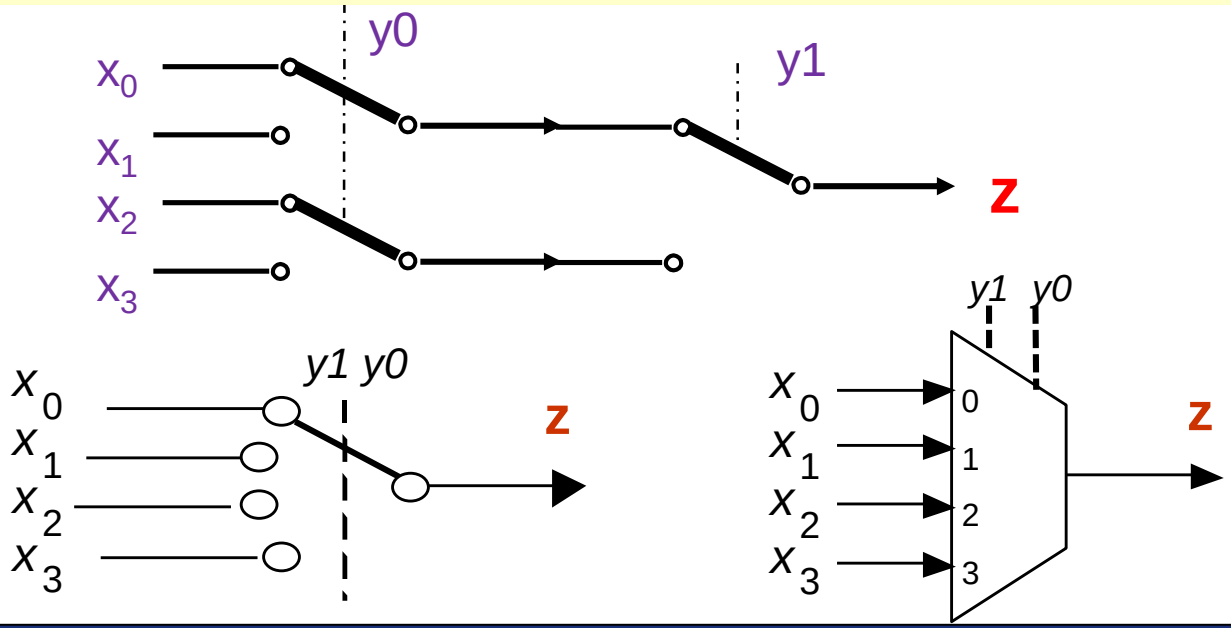


Přepínačové analogie multiplexoru

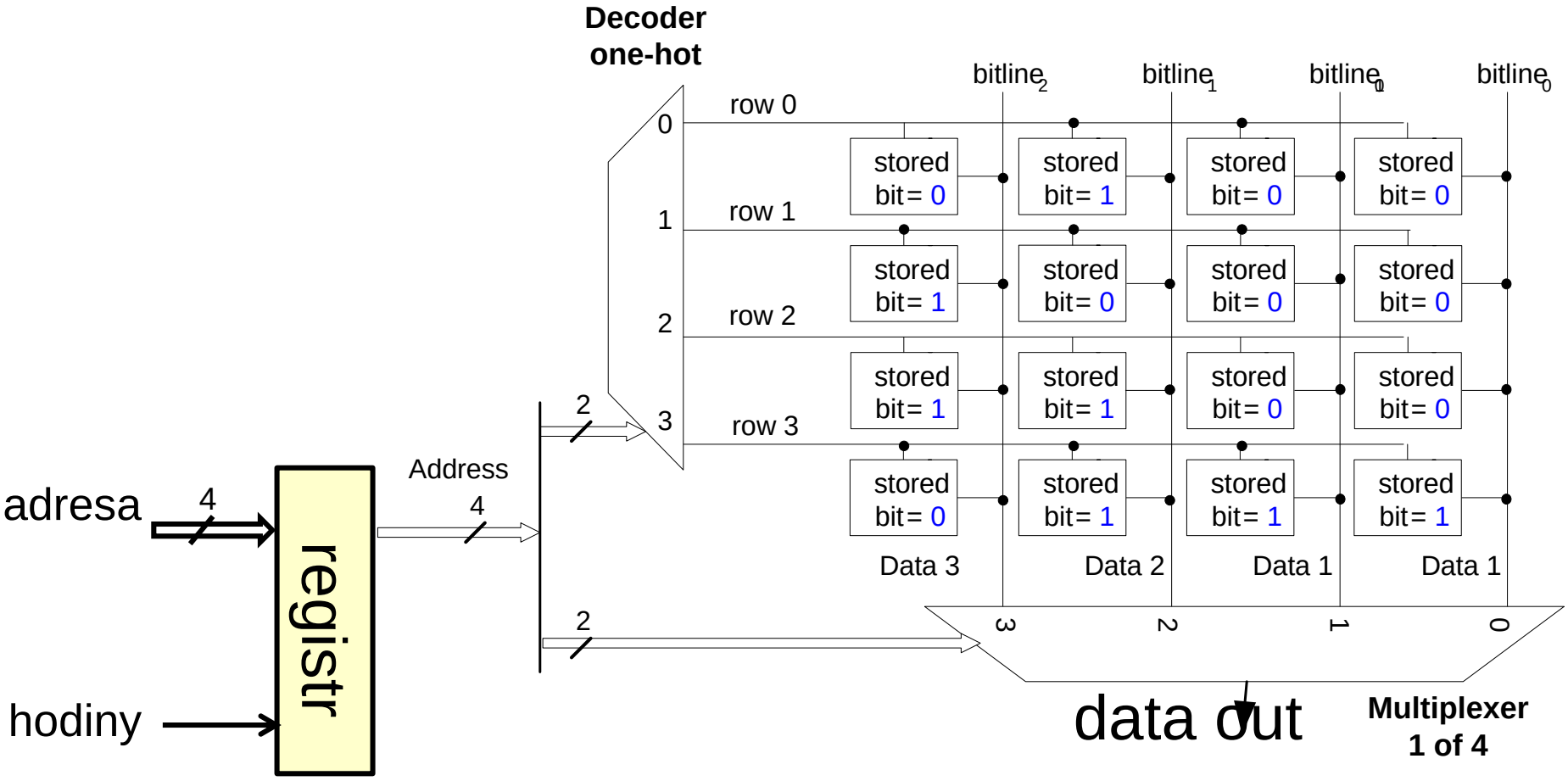
Multiplexer 2 to 1 or **1 of 2** *cz :2 kanálový (2-vstupový) multiplexor*



Multiplexer 4 to 1 or **1 of 4** *cz : 4 kanálový (4-vstupový) multiplexor*

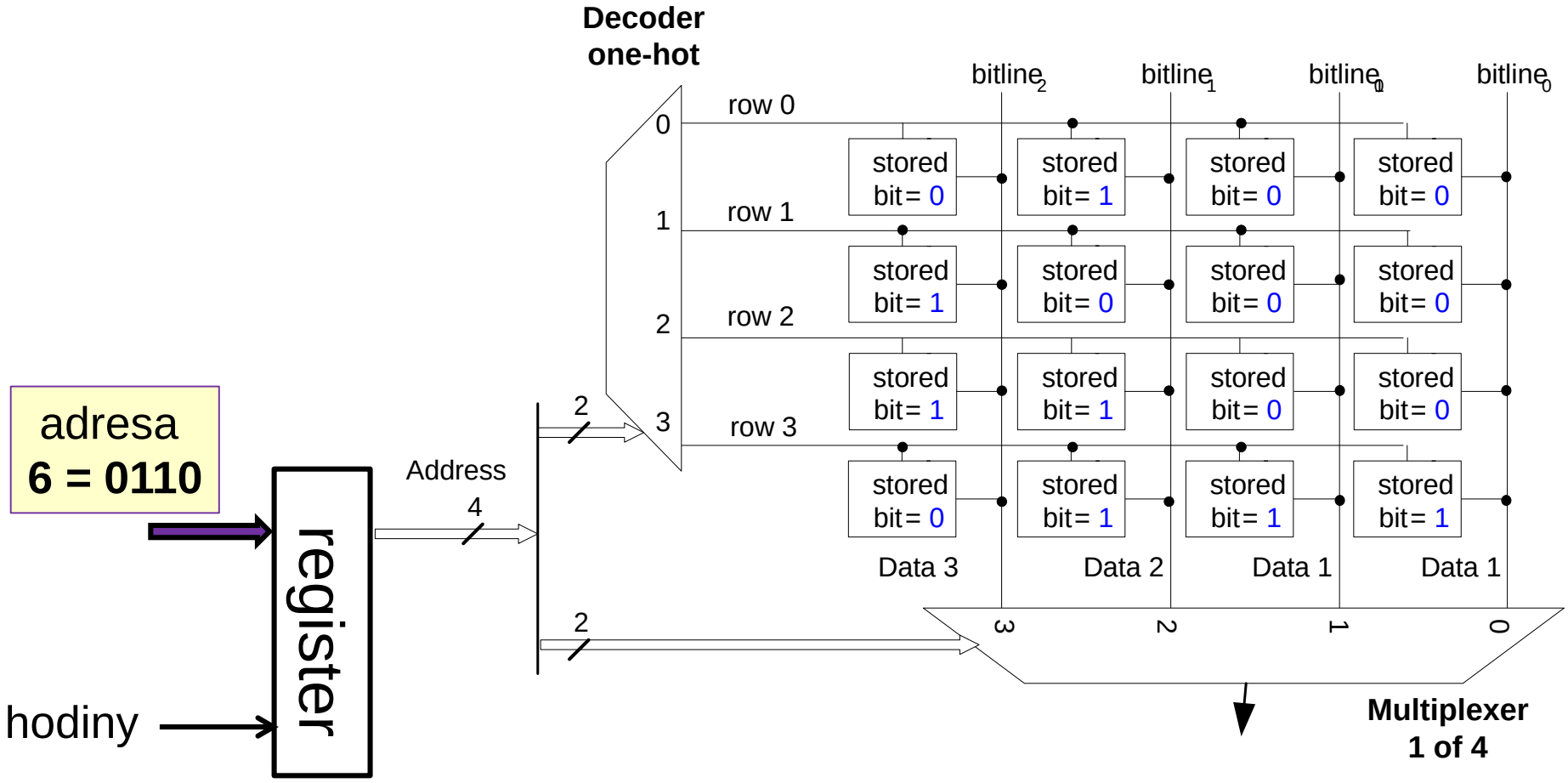


Paměťová matice



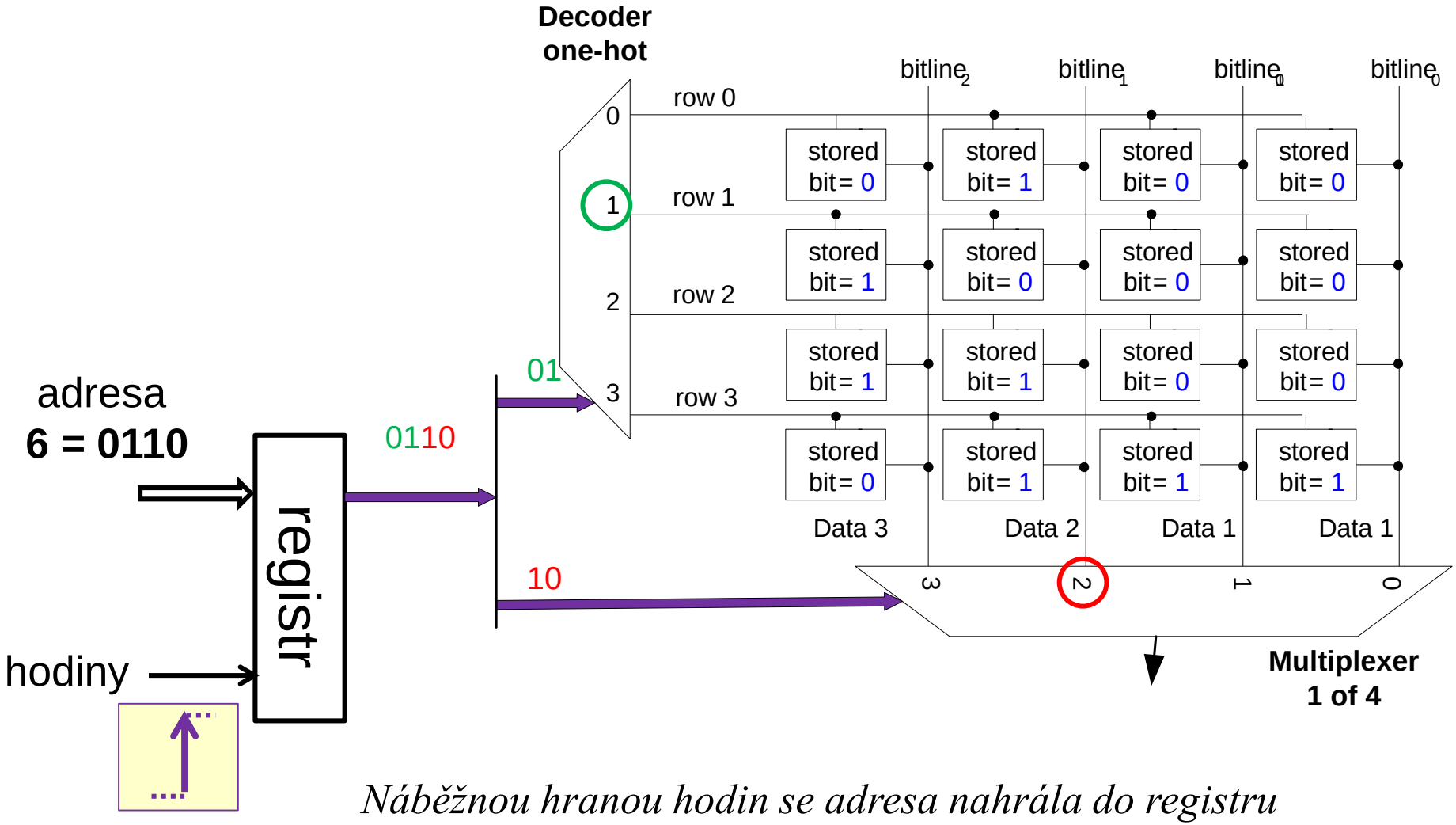
Registr adres představuje nezbytnou implementační podmínku.

Paměťová matice – příklad čtení 1/4



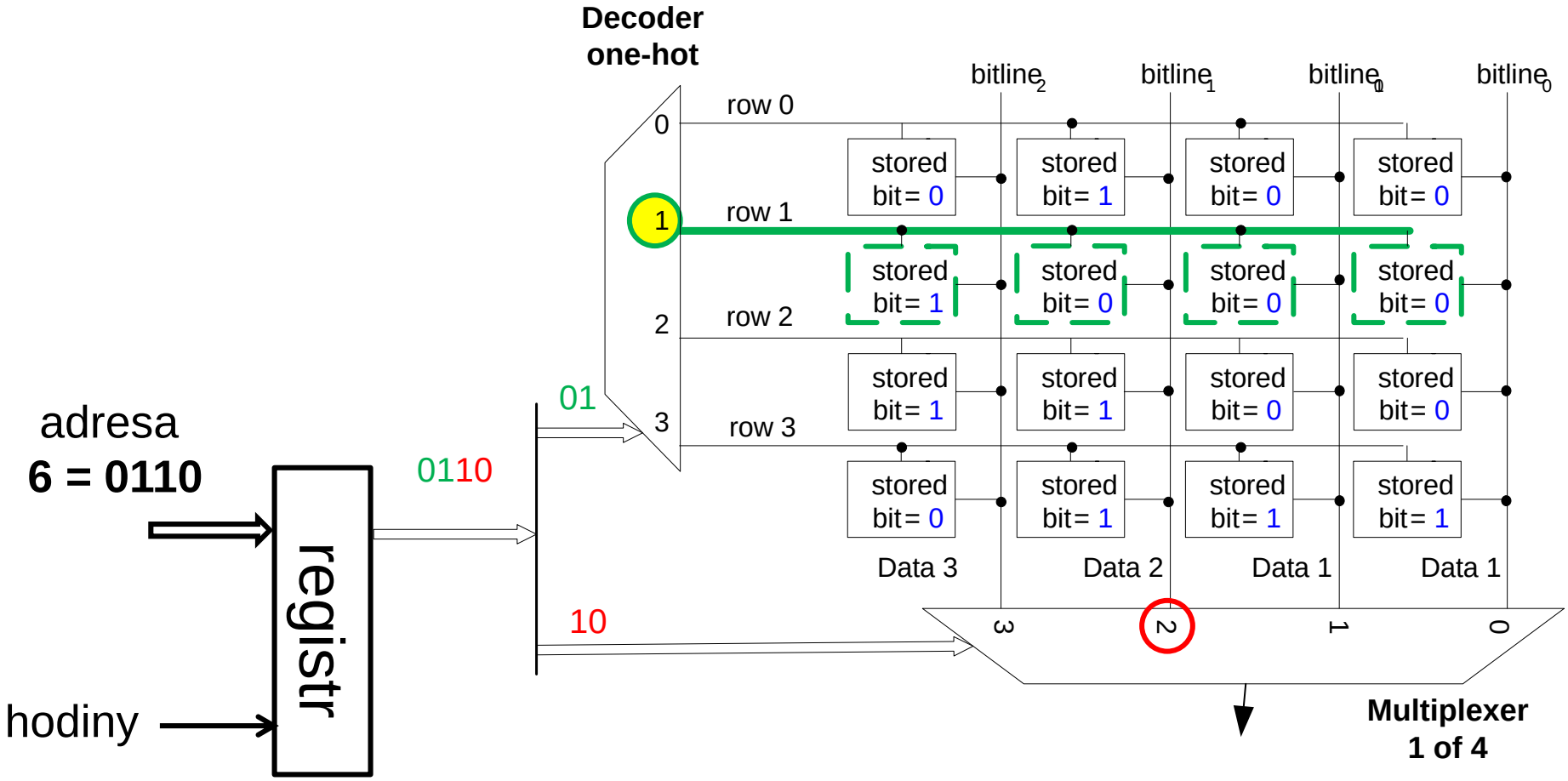
Hodnota adresy čeká na vstupu na náběžnou hranu hodin.

Paměťová matice – příklad čtení 2/4



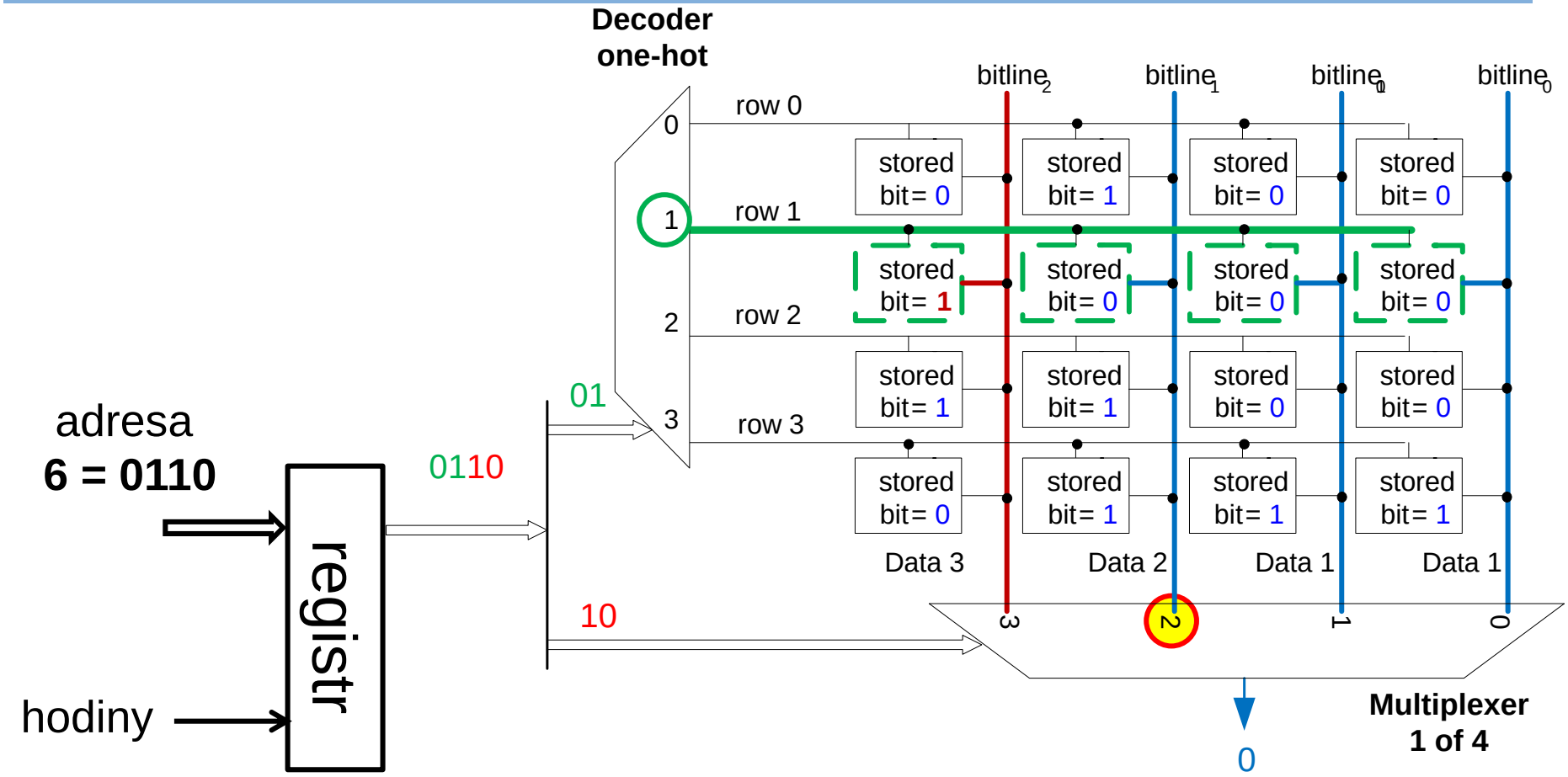
Náběžnou hranou hodin se adresa nahrála do registru a rozdělila se na horní a dolní bity, výběr řádku a sloupce

Paměťová matice – příklad čtení 3/4



Dekodér 1 z N aktivuje celou řádku a její data se objeví na bitline všech sloupců.

Paměťová matice – příklad čtení 4/4



Multiplexor vybere adresovaný sloupec - *Data 2 = 0*

Vložíme-li před multiplexor registr, do něhož si uložíme část řádky (prefetch), pak ho lze využít ke čtení následujících adres - data se posílají z něho.

Terminologie kolem pamětí

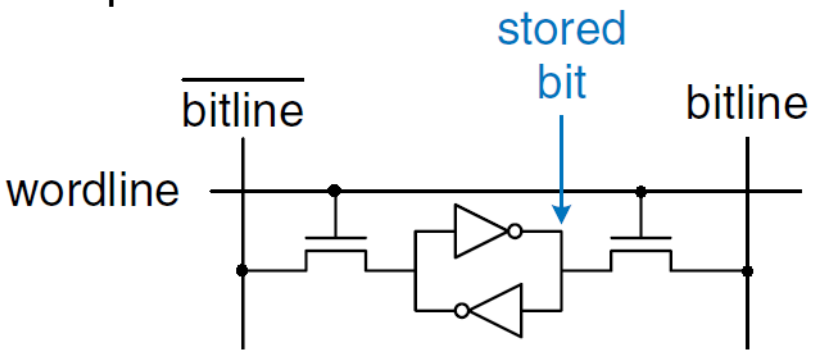
- Typy pamětí RWM (RAM), ROM, FLASH,
- Provedení RAM pamětí:
  SRAM (statická), **DRAM** (dynamická).

- RAM = *Random Access Memory – paměť s **libovolným** přístupem*

typ paměti	počet tranzistorů	plocha na 1 bit	dostupnost dat	latence
SRAM	cca 6	< 0,1 μm^2	vždy	< 1ns – 5ns
DRAM	1	< 0,001 μm^2	potřebuje refresh	dnes 20 ns – 35 ns

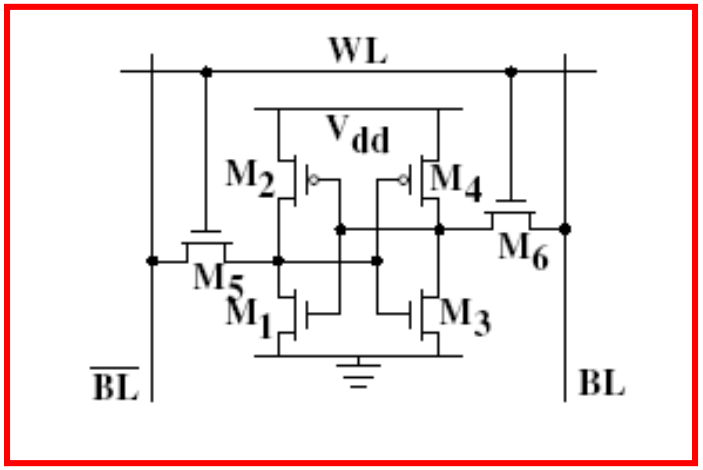
Typický čip a buňka SRAM

Princip:

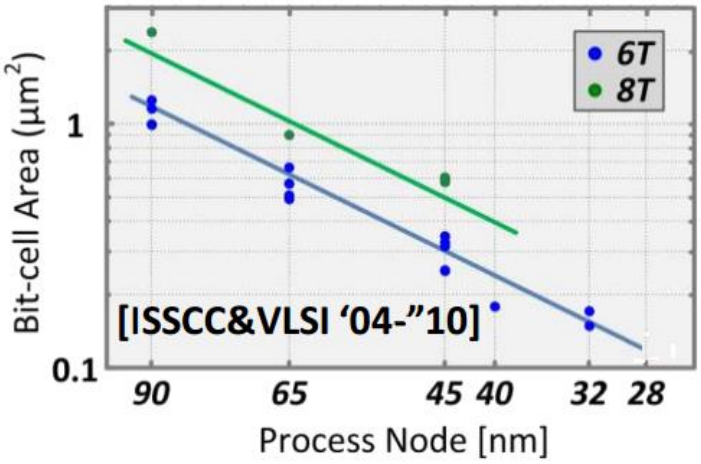


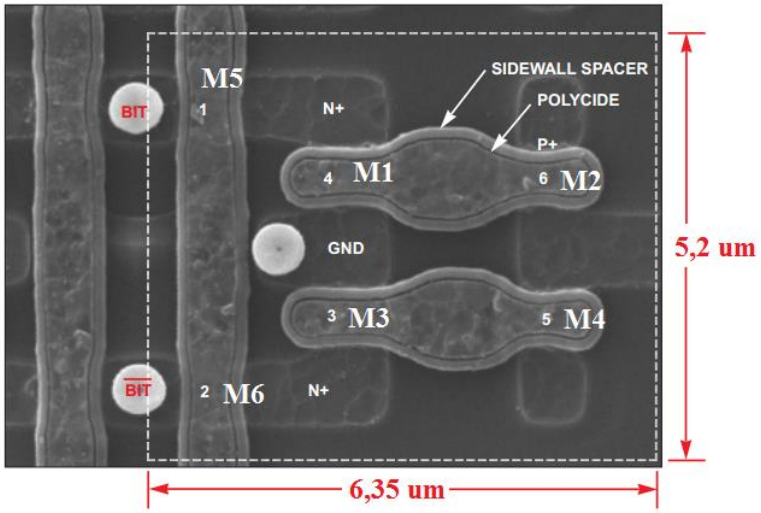
SRAM paměťová buňka

6-transistorů CMOS, existují 4 trans verze



Plocha paměťové buňky:

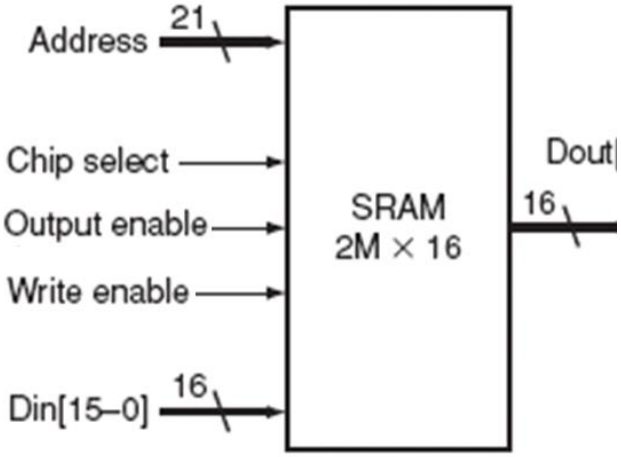




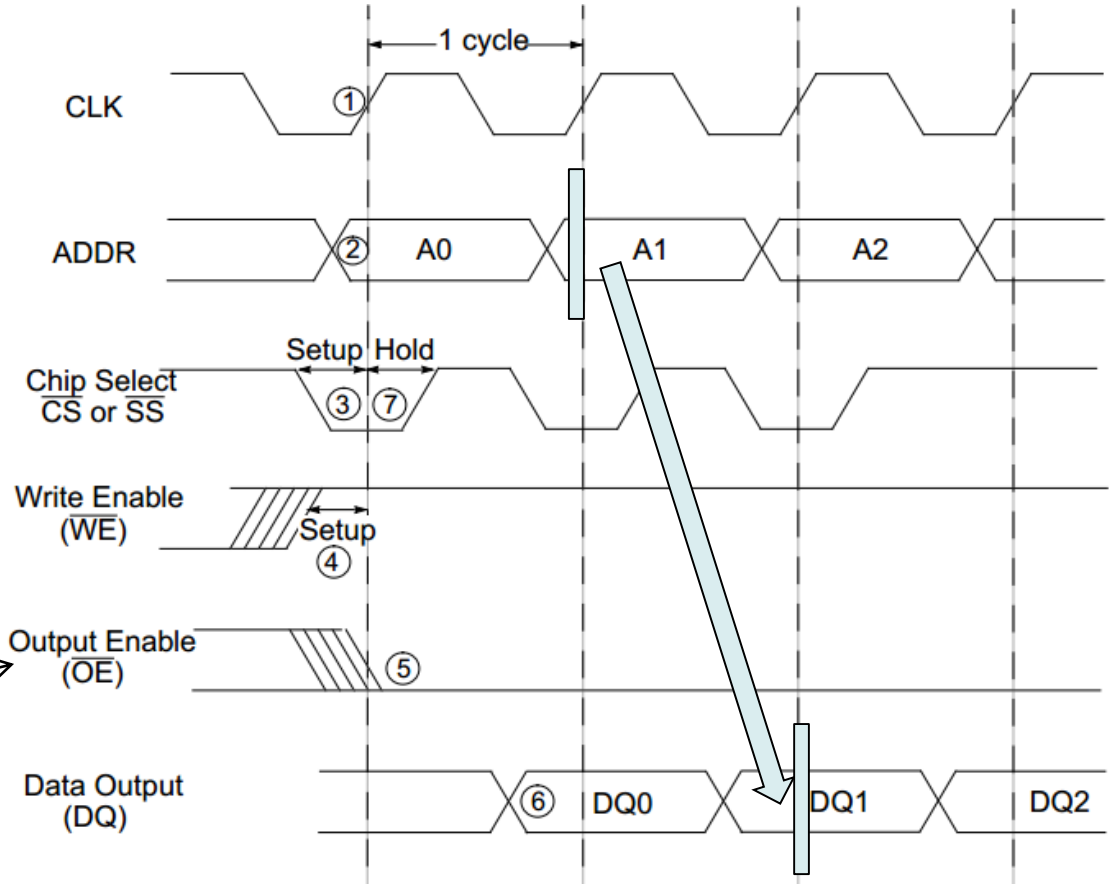
http://educypedia.karadimov.info/library/SEC08.PDF

Typický čip a buňka SRAM

Typický SRAM čip



Příklad čtení – typicky synchronní :

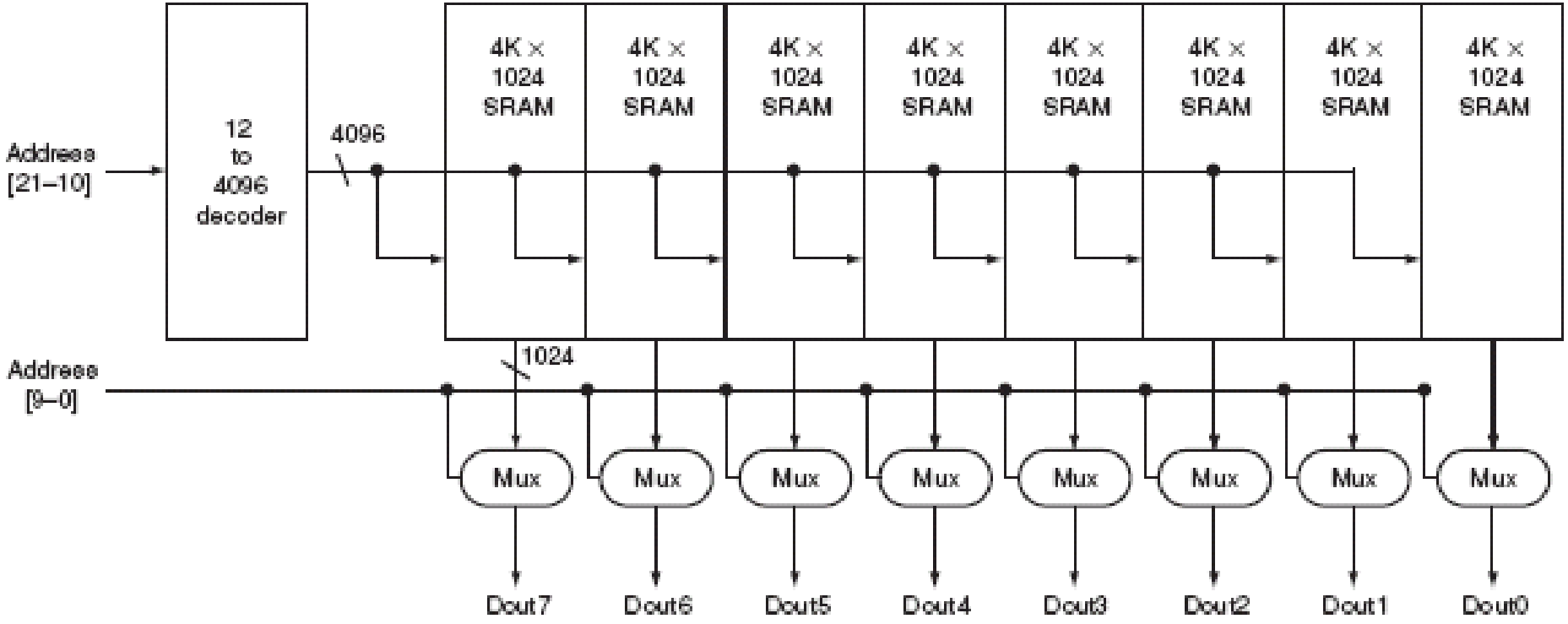


OE signál je asynchronní

https://www.ece.cmu.edu/~ece548/localcpy/sramop.pdf

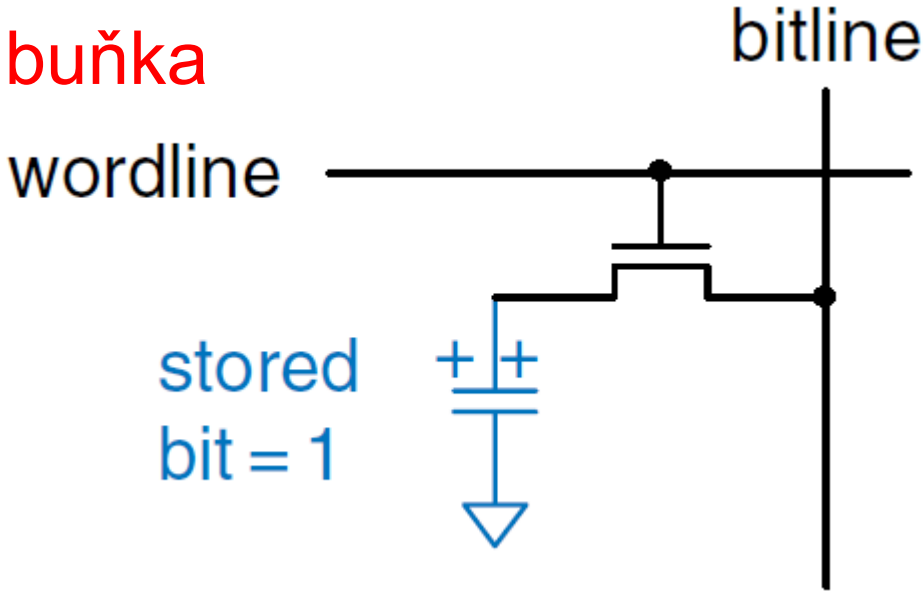
Typický čip a buňka SRAM

Větší paměť?



Detail paměťové buňky dynamické paměti

Jednotranzistorová
dynamická paměťová buňka



- Transistor nMOS představuje přepínač, který připojí kondenzátor na vodič „bitline“.
- Vodič „wordline“ řídí, zda se kondenzátor připojí či ne.

Obrázek: http://www.eetimes.com/document.asp?doc_id=1281315

Kondenzátor dynamických pamětí

Komerční DRAM parametry	
	Kapacita fF [femtofarad]
Kondenzátor kapacita	od 10 fF do 50 fF
Bit line kapacita	kolem 2 fF

[Source: l'INSA de Toulouse]

fF - femtofarad

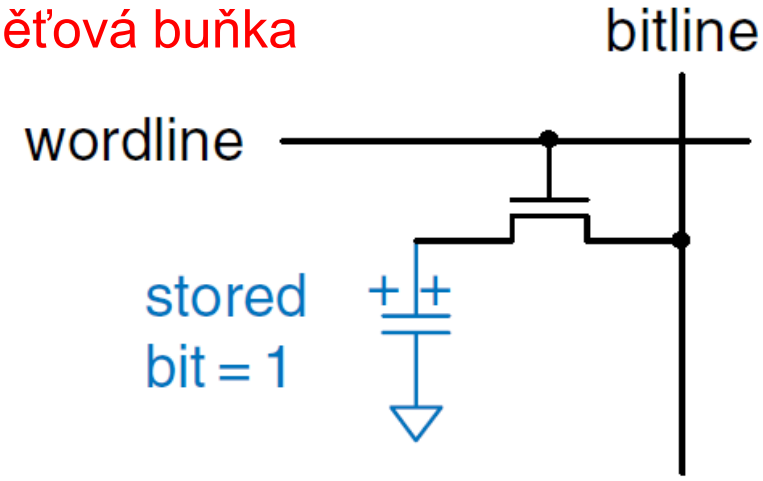
fF je SI jednotka rovná 10^{-15} faradů.

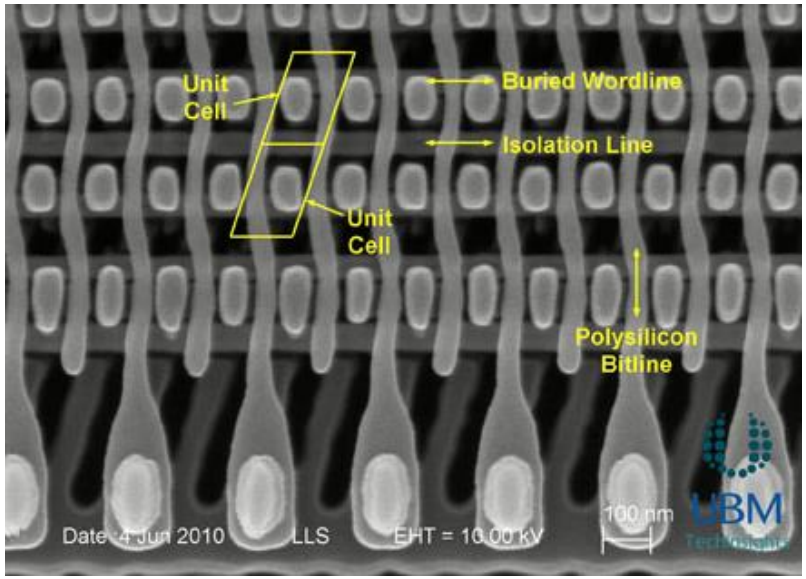
10^{-6} F = **1 µF** = 10^{3} nF = 10^{6} pF = 10^{9} fF

~9 fF kapacita vznikne ve vakuu mezi deskami o ploše 1 mm^2 vzdálenými od sebe 1 mm,

Detail paměťové buňky dynamické paměti

Jednotranzistorová dynamická paměťová buňka



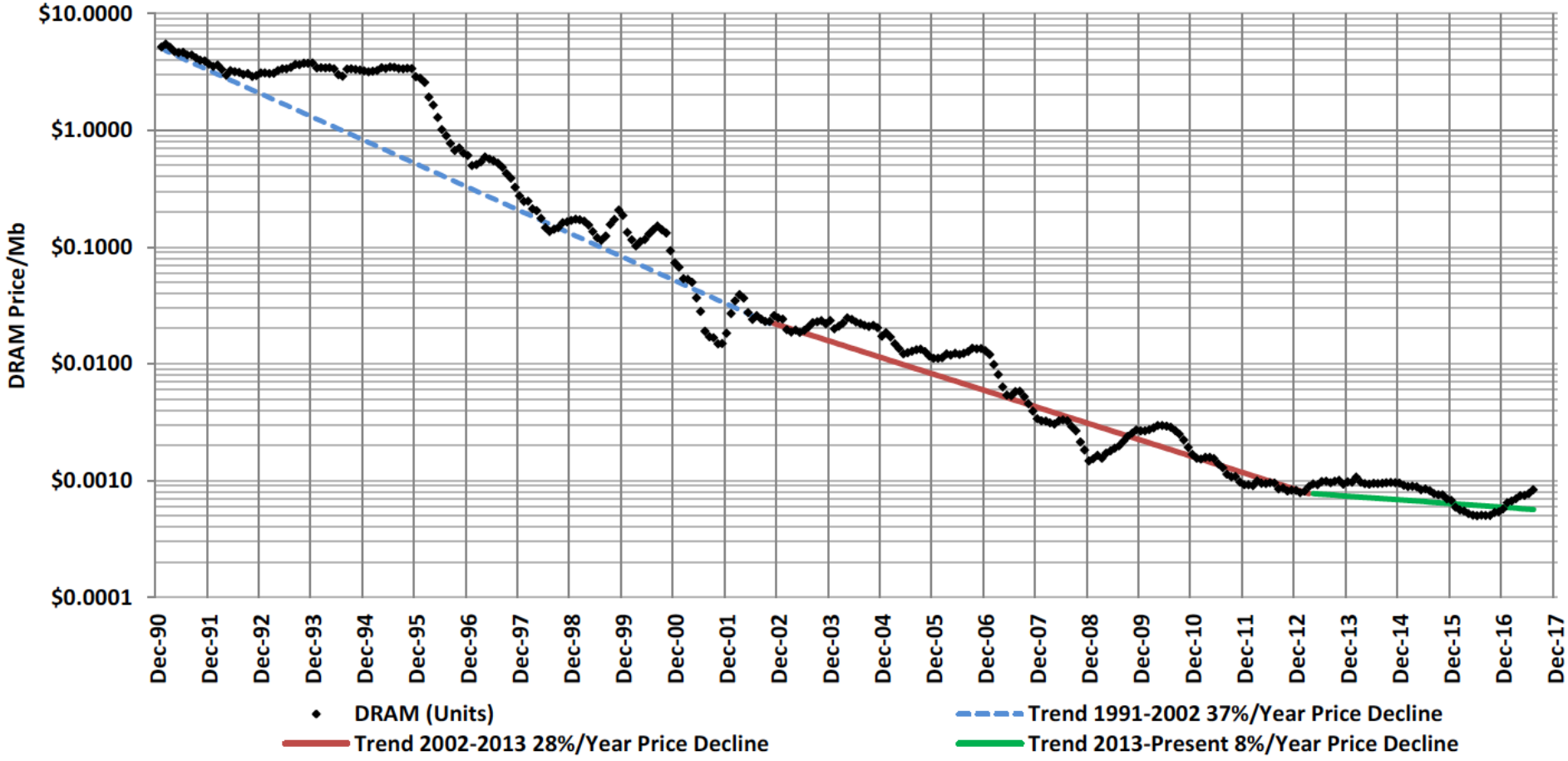


- Čtení je složité a pomalé, trvá kolem 20 až 35 ns, a nelze ho již více urychlit.
- Čtení je navíc destruktivní, kondenzátor se vybije a nutno do něj opět nahrát hodnotu.
- Femto-faradový kondenzátor se samovolně rychle vybíjí - nutno ho obnovovat (refresh) cca každých 64 ms.

Obrázek: http://www.eetimes.com/document.asp?doc_id=1281315

Paměti DRAM – cena již neklesá

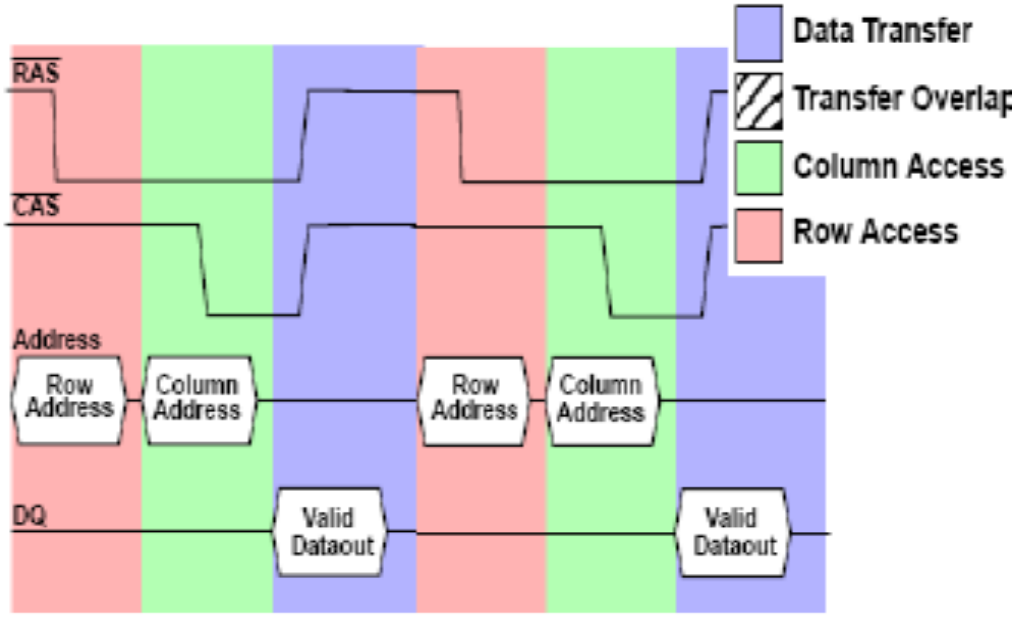
Cena za 1 megabit



Source: Wells Fargo Securities, LLC and Semiconductor Industry Association

Klasická DRAM – asynchronní rozhraní

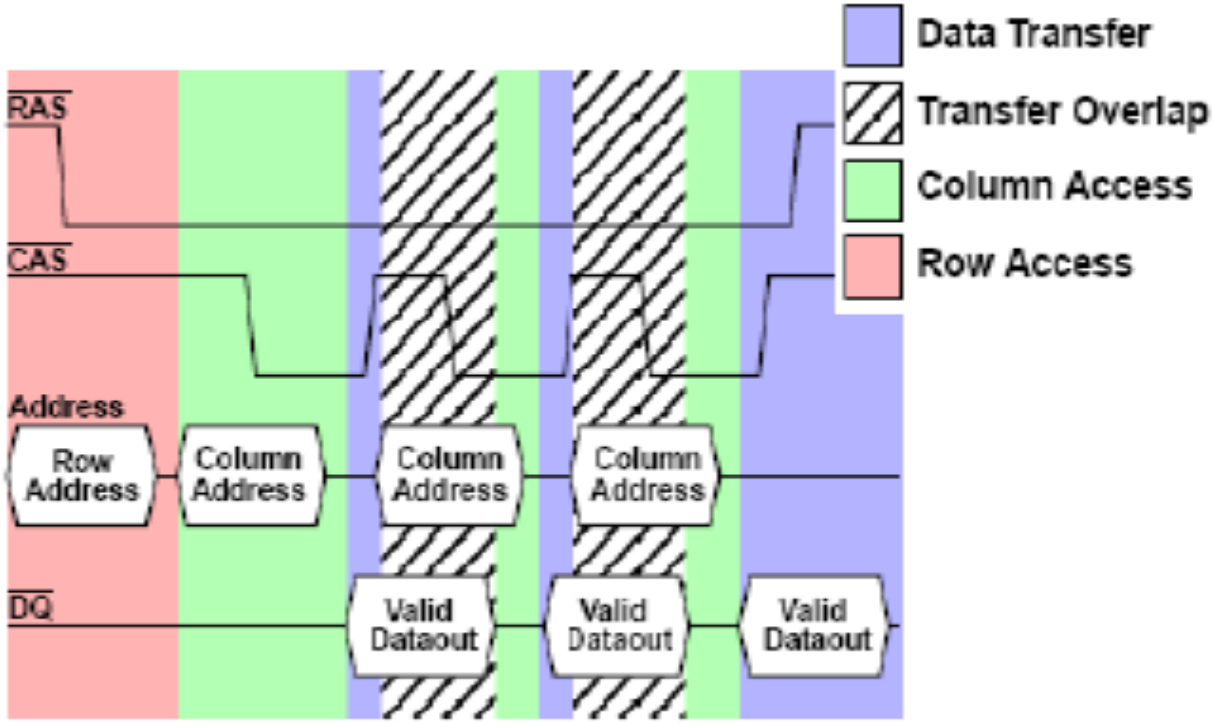
- Důvod rozdělení adresy na 2 části byl dán malým počtem pinů původních DRAM pouzder.
- Toto rozdělení se dodnes zachovává. Sice pouzdro už není problém, ale spojení RAS a CAS by nepřineslo výhodu. Proč?



RAS – Row Address Strobe,
CAS – Column Address Strobe

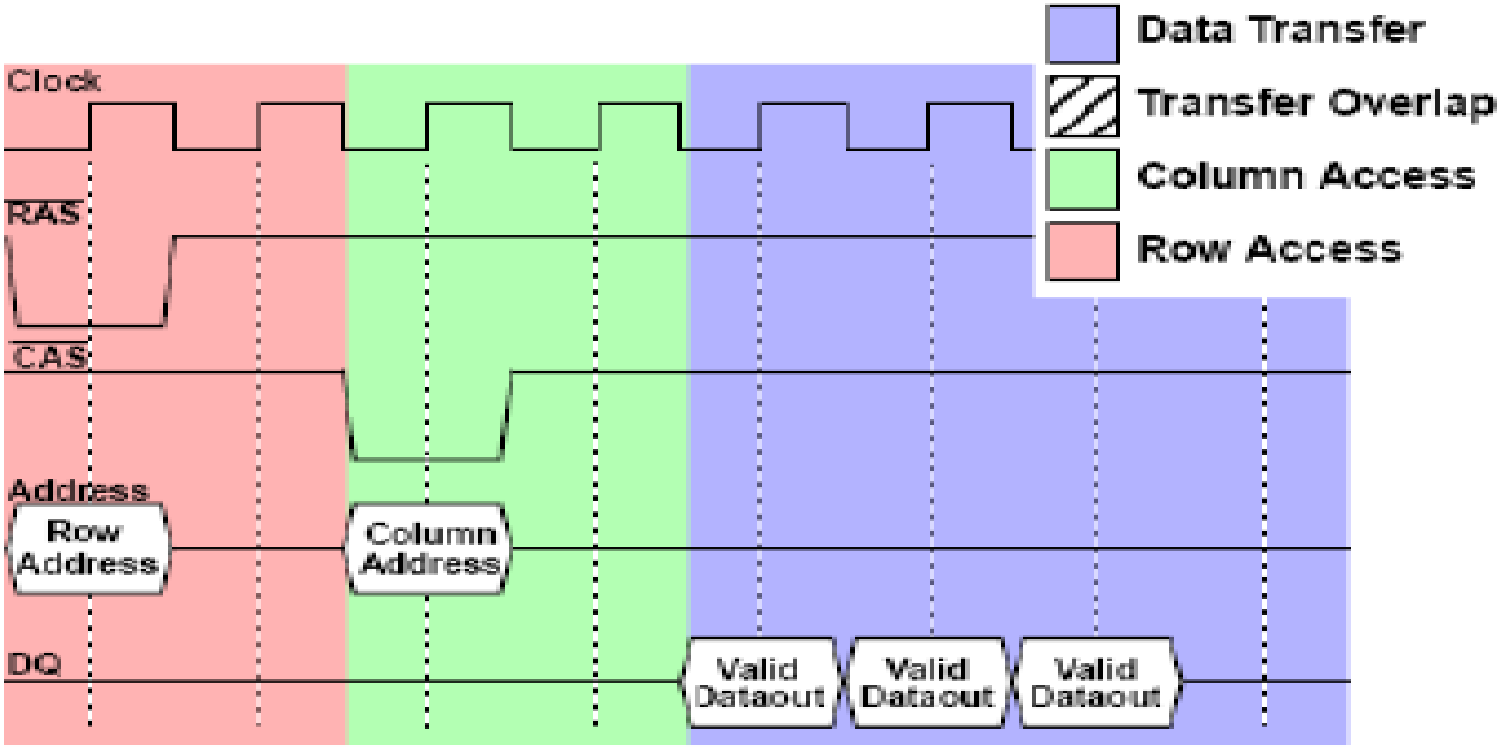
EDO DRAM – cca 1995

- EDO DRAM má registr na výstupu, což umožní následujícím CAS čtení dalších dat téže řádky.



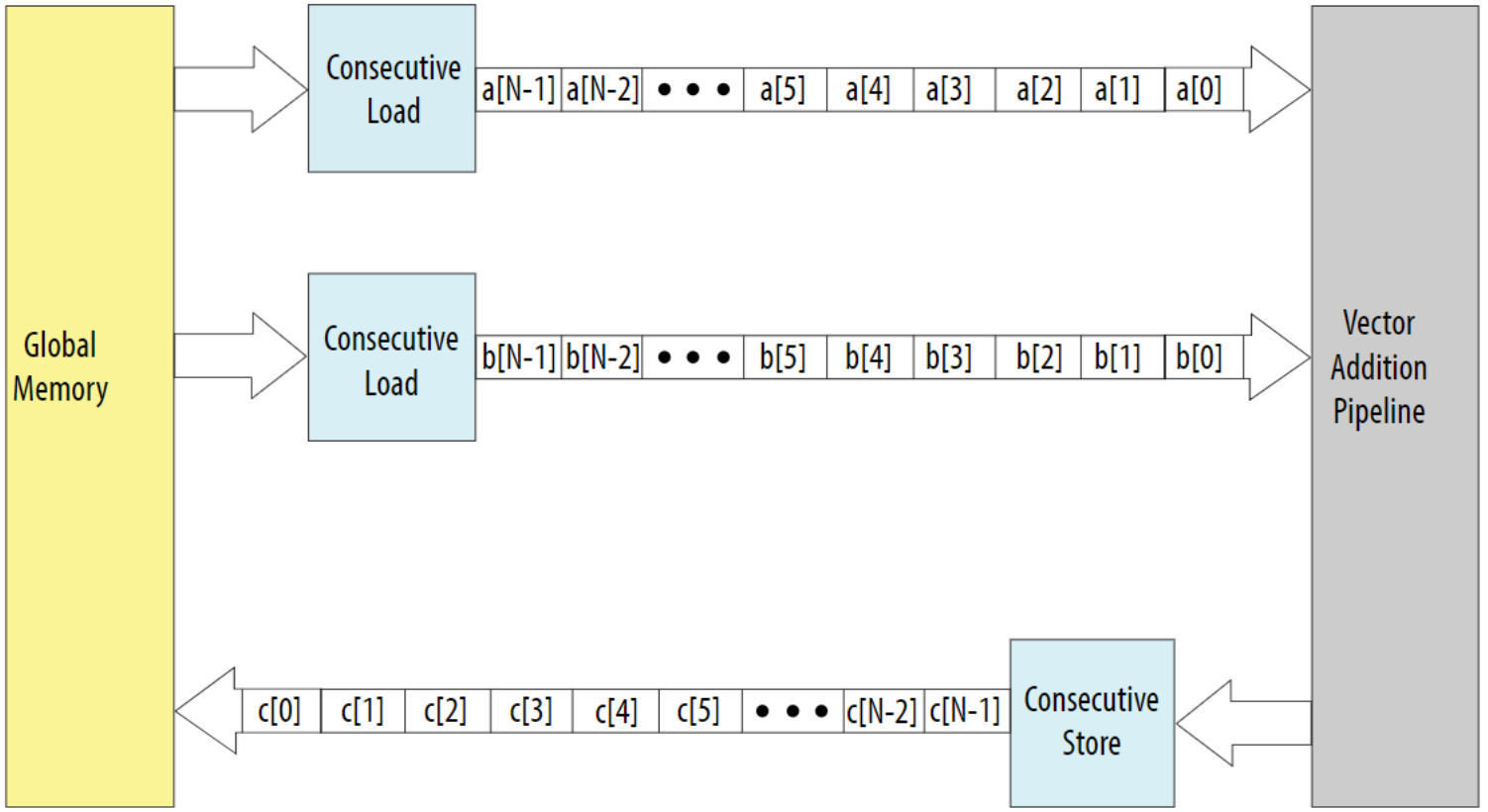
SDRAM – konec 90.let – synchronní DRAM

SDRAM čip umožňuje souvislé (**burst**) čtení/zápisu dat, při čtení jedné položky se pošlou i následující. Délku lze nastavit v mód registrech paměti. (DDR4 mají až 7 mód registrů, DDR5 až 256 mód registrů na různé oblasti)



Sekvenční Read/Write

Paměť či její řadič si načítá adresu a využívá burst mode.



SDRAM – paměť současnosti

- SDRAM – frekvence až 100 MHz, 2.5V
- DDR (*double data rate*) SDRAM – použití obou hran CLK při přenosu dat, napájení 2.5V, I/O bus clock 100-200 MHz,
  rychlost přenosu 0.2-0.4 GT/s (gigatransfers per second)
- DDR2 SDRAM – snížení spotřeby použitím napětí 1.8V,
  frekvence IO bus 400 MHz, 0.8 GT/s
- DDR3 SDRAM – snížení spotřeby použitím napětí 1.5V, IO bus frekvence až 800 MHz, 1.6 GT/s
- DDR4 SDRAM - napětí 1,05 – 1,2V, I/O bus clock 1.2 GHz, 2.4 GT/s
- DDR5 *SDRAM - očekávají se 2019-20, plánovaná rychlost ~6 GT/s*

Všechny inovace vylepšují propustnost čtení dat,
latence čtení první položky zůstává na 20 až 35 ns.

Další paměti

- QDRx SDRAM (Quad Data Rate) - nejsou však dvakrát rychlejší, pouze umožňují současné čtení i zápis, jelikož mají oddělené hodiny pro RD a WR, zatímco DDR jsou naopak efektivnější než QDR pro jeden typ přístupu.
- **GDDR** SDRAM - dnes až typ GDDR6, vhodné pro grafické karty
  - založené na DDR pamětech.
  - rychlost se zde akceleruje rozšířenou výstupní sběrnicí.
- Dále ještě existují paměti a moduly RDRAM (RAMBUS DRAM), které ovšem mají zcela odlišné rozhraní i způsob použití. Pro patentní spory se v osobních počítačích nepoužívají od roku 2003.

Pohled programátora?

Otázka 1. Vykonávají programy to samé?

Otázka 2. Který program je rychlejší (pokud některý)?

A:

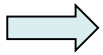
```
int matrix[M][N];
int i,j,sum=0;
…
for(i=0;i<M;i++)
  for(j=0;j<N;j++)
    sum+=matrix[i][j];
```

B:

```
int matrix[M][N];
int i,j,sum=0;
…
for(j=0;j<N;j++)
  for(i=0;i<M;i++)
    sum+=matrix[i][j];
```

Lze doporučit výhodnější způsob procházení matice?

Žel nelze jednoznačně odpovědět. Musíme znát způsob uložení matice do paměti počítače a vnitřní organizaci jeho paměti.



Vícebytová čísla

a jejich uložení v počítačích

Způsoby uložení vícebytových čísel v paměti

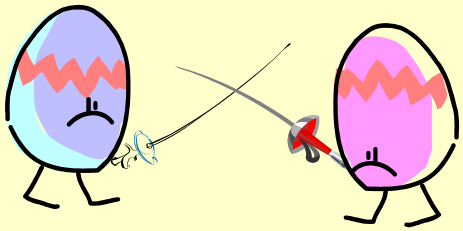
Hex číslo: 1234567

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?

Překvapení na závěr ???

```
int main()  { float x; double d; x = 116777215.0;
   printf("%.3f\n", x);       // 116777216.000
   printf("%.3lf\n", x);      // 116777216.000 – float se printf předává jako double, l nemá 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
     // Vysvětlení: float se předal jako double a jeho mantisa končí nulami, v little endian jsou první
     // Pak se %f čte začátek double, ale jako float, 11-bit double exp. čtený jako 8-bit bude malý.
   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- float zaokrouhlí mantisu
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; }
```


Opakování C: Sign Extension (znaménkové rozšíření)

Prerekvizita APOLOS – kapitola 3.2.5

Příklad:

```
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

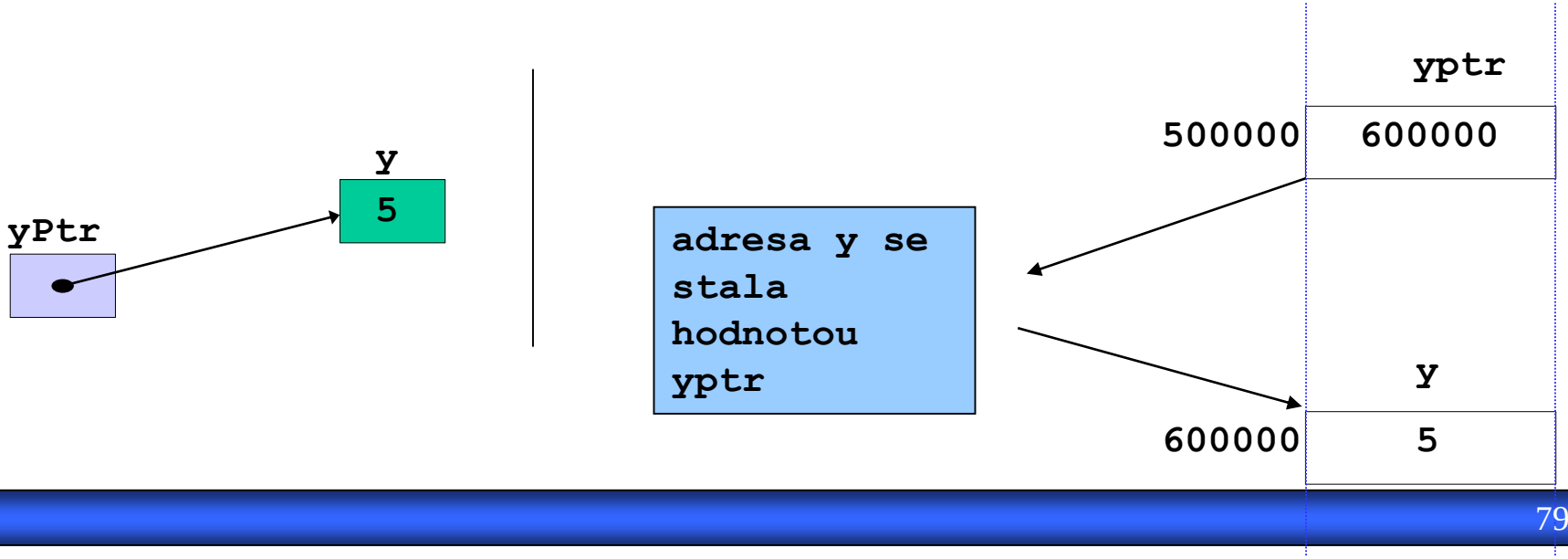
& (address operator)

Vrací adresu začátku proměnné v adresovém prostoru.

Příklad

```
int y = 5;
int *yPtr;
yPtr = &y;     // yPtr obsahuje adresu y
```

yPtr "ukazuje na" **y**



Opakování C: Operace s ukazateli

& (address operator)

vrací adresu operandu

* dereference address

hodnota uložená na adrese

***** and **&** jsou inverzní
(*ale ne vždy aplikovatelné*)

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

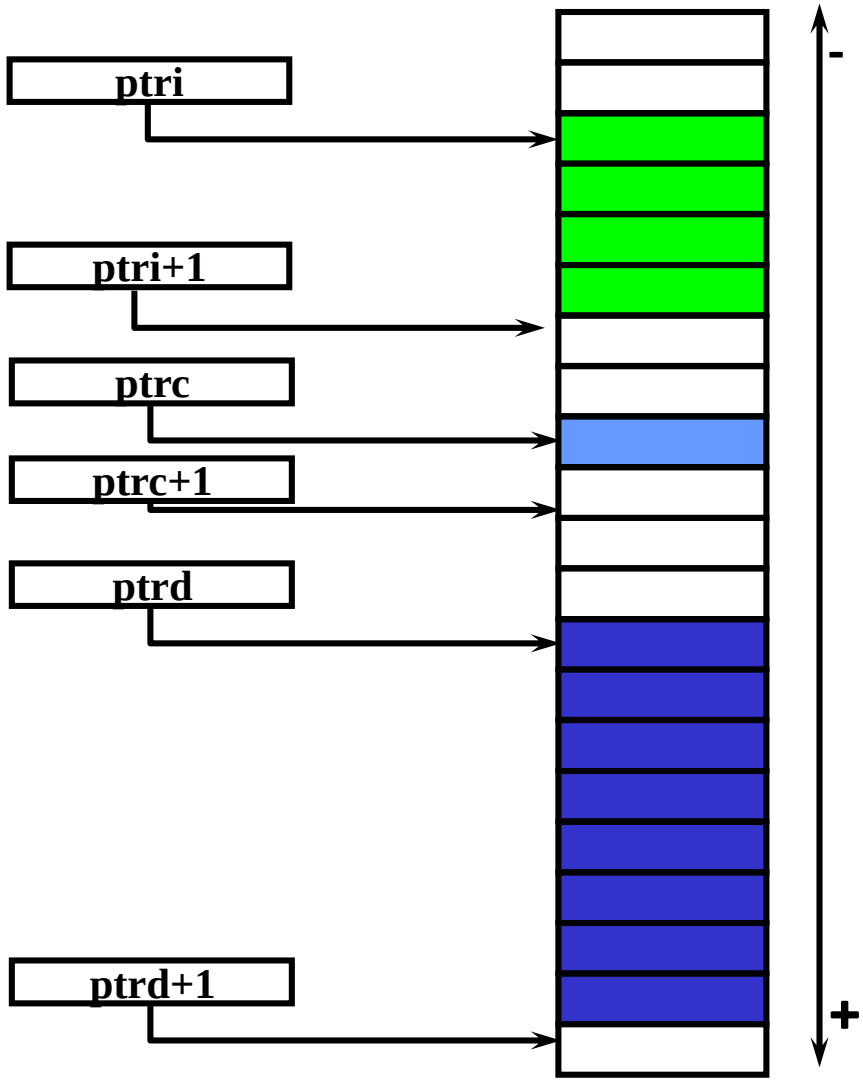
and

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

Opakování C: Velikost ukazatele v C-kódu

```
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

Opakování C: Ukazatel s const v C-kódu

```
int x, y;
int * lpio = &y;
   *lpio = 1;      x=*lpio;       lpio++;
const int * lpCio = &y;
   *lpCio = 1;     x=*lpCio;      lpCio++;
int * const lpioC = &y;
   *lpioC = 1;     x=*lpioC;      lpioC++;
const int * const lpCioC = &y;
   *lpCioC = 1;    x=*lpCioC;     lpCioC++;
```