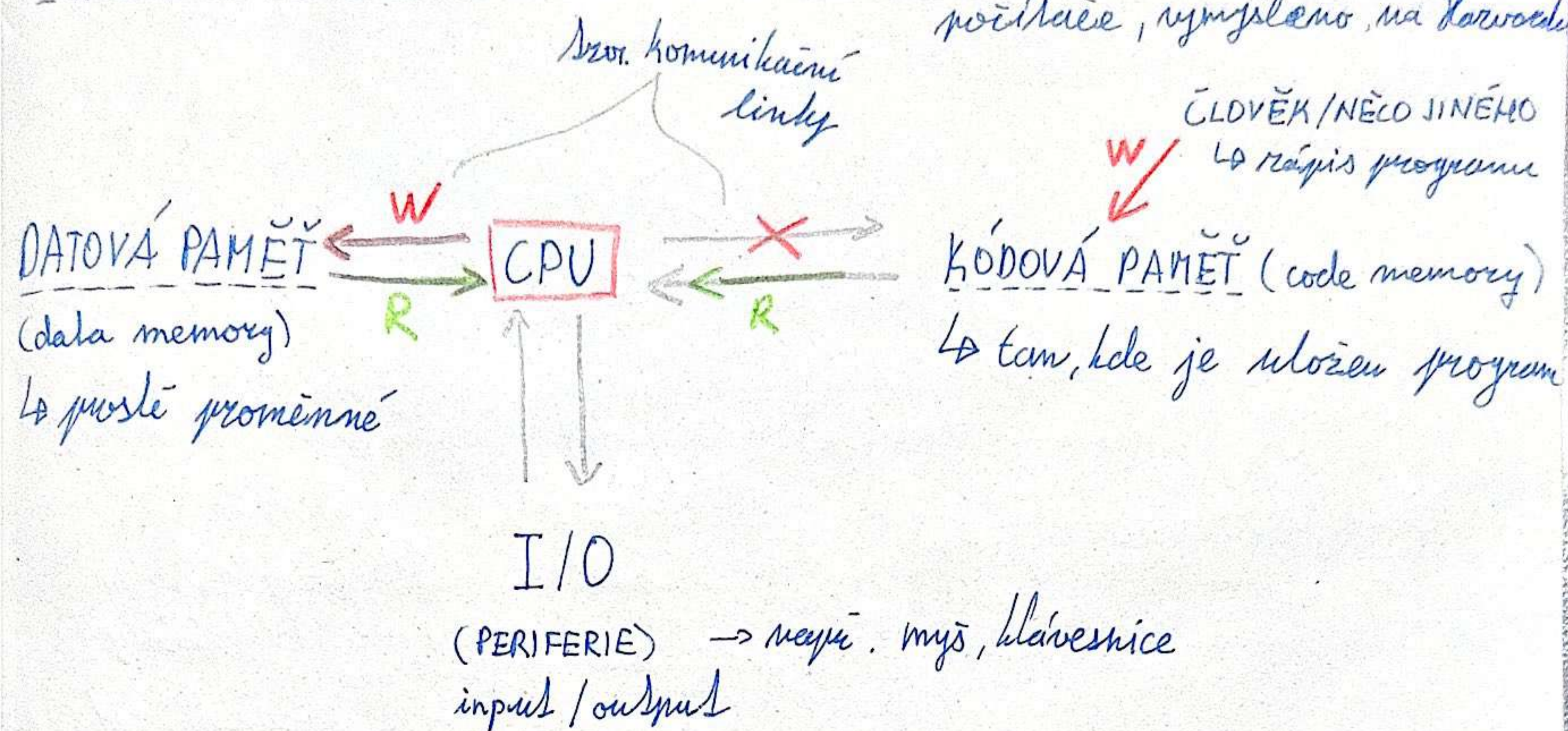


1. HODINA

Harvardská architektura počítače = zjednodušené schéma počítače, vymyšleno na Harvardu



Historie

Charles Babbage

→ vymyslel analytical engine v roce 1837

→ byl Turingovsky kompletní (schopný řešit jakýkoliv problém), ale nebyl postaven

→ měl sloužit k počítání členů v Británii

Ada Lovelace

→ dcera G. G. Byrona (romantický básník)

→ finančně Babbage podporovala

→ napsala manuál stroje → k analytical engine

→ řekla, že se vše dá redukovat do posloupnosti čísel

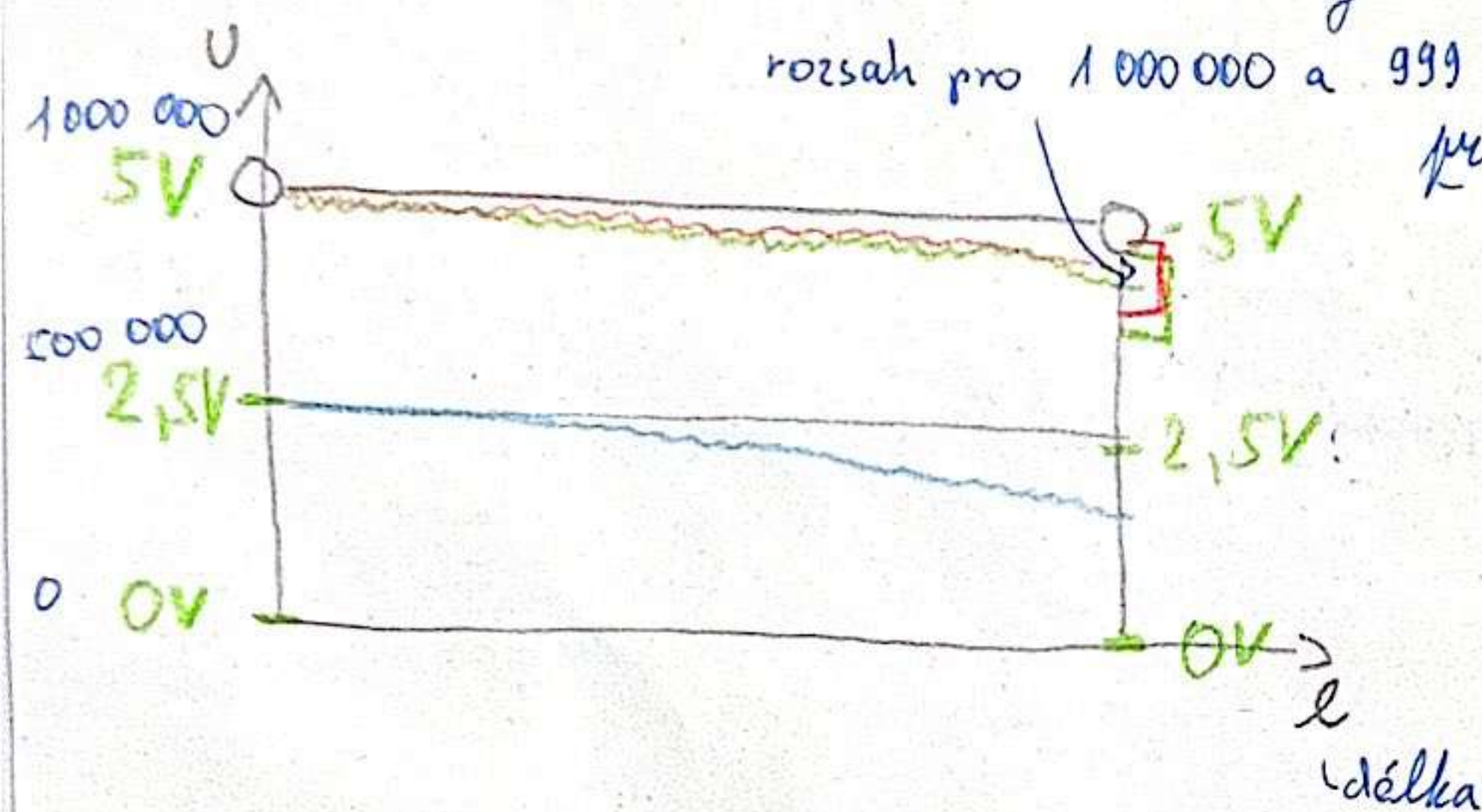
Kódování informace v počítači

↳ informace → převést na čísla → tedy celá čísla → tedy ještě lépe: vzáporná celá čísla (např. 1, 2, 3, ..., 1 000 000)

Analogový přenos

1 = 1V ; 2 = 2V → nerealistické

0 = 0V ; 1 000 000 = 5V → už by šlo



ale používá se přenos není přesný

1) vznikne šum → rozsah by se překrývaly

2) klesne napětí

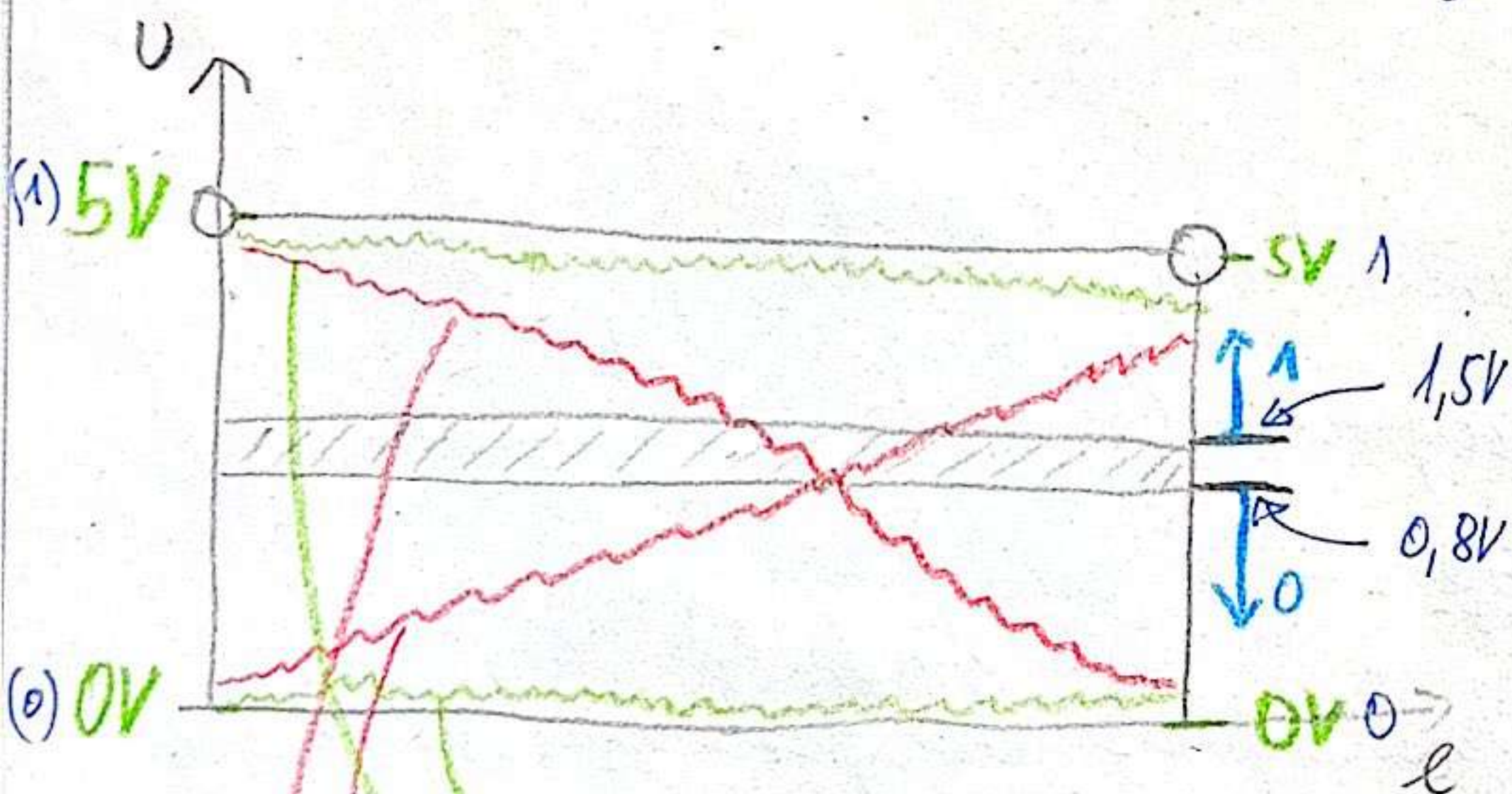
a nelze vyřešit posunutím škály, protože závisí na teplotě, délkě vodiče, elmag. poli vodiče

↳ od vesmíru → šum

Digitální přenos

↳ místo milionu hodnot jen dvě : 0 = 0V
1 = 5V

↳ lze zvést jednotku bit („Binary Digit“) → b



OČEKÁVANÝ VÝSTUP

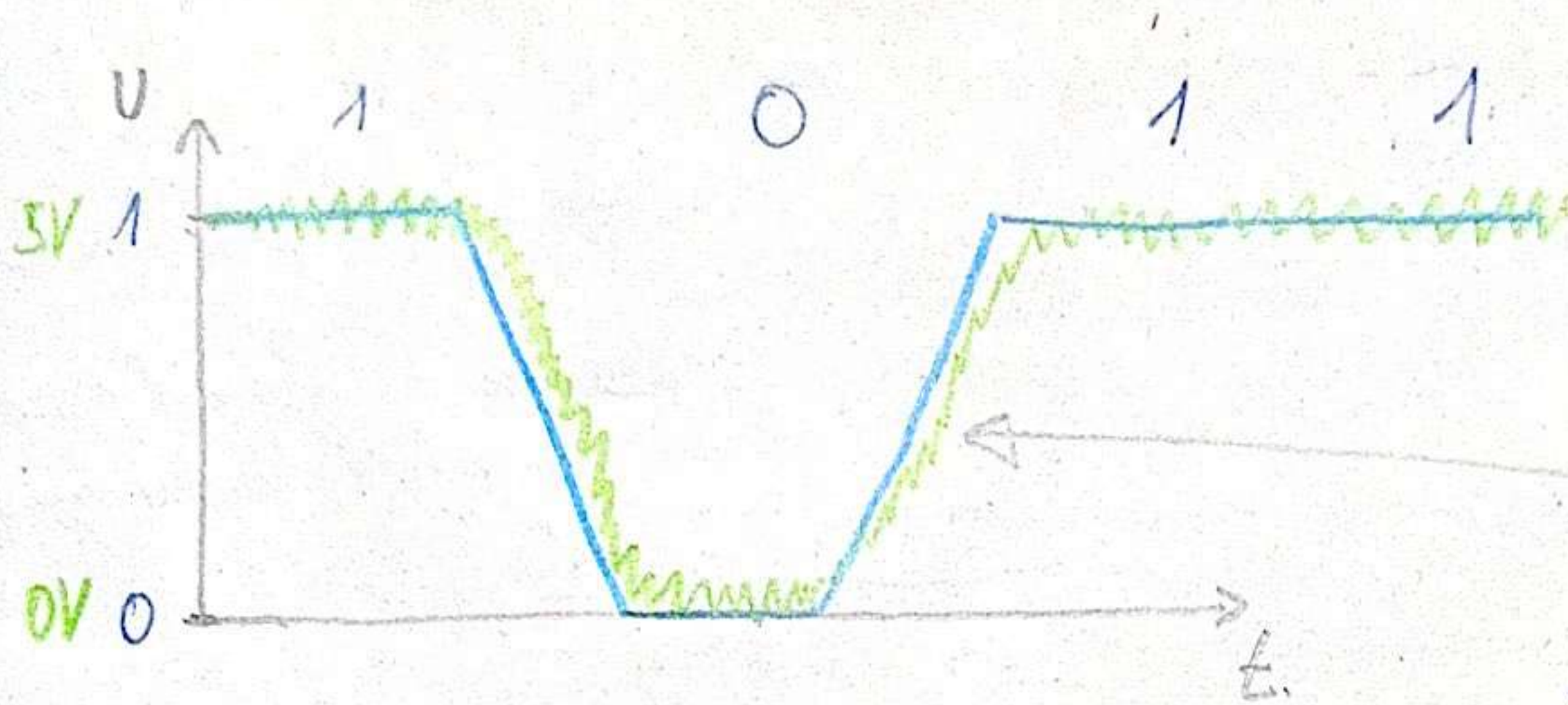
MŮŽE SE STÁT → když bude šum moc velký

důležité:

↳ vysílající a přijímající se musí domluvit se měre

- Serialový přenos (přítel patří pod digitální)

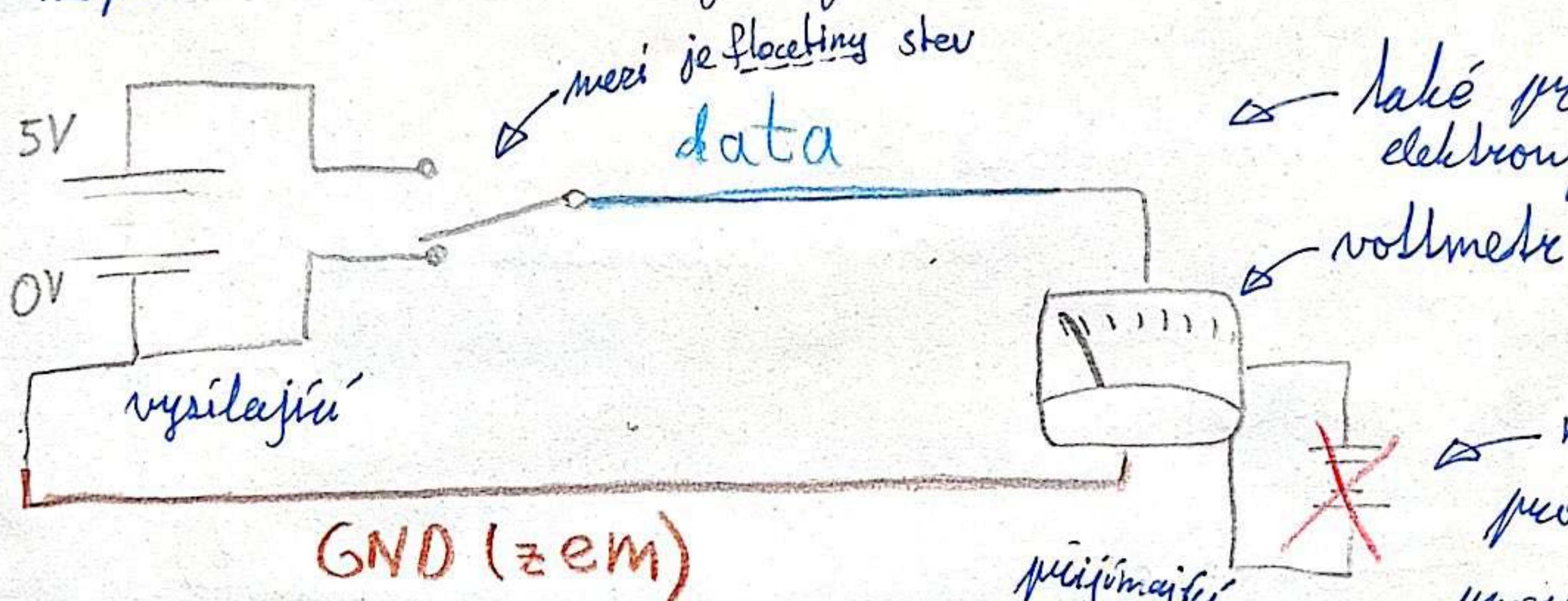
↳ způsob přenosu: pošle se více bitů informace za sebou



— timing diagram
(stopky) → interpretace
opravdové napětí
↳ vidět přijímači

slow je způsoben
tím, že se vodič musí
nabít a navať to, jako
kondenzátor se brání změně
napětí

↳ ale je potřeba se domluvit na zemi (referenční hodnotě), protože
napětí se dá měřit jen jako rozdíl



mezi je floating stav
data

↳ také propojeno, aby se
elektrony do baterky vrátily

voltmetr

↳ nemá smysl,
protože musí
provést proud,
a proto má být
uzavřený okruh

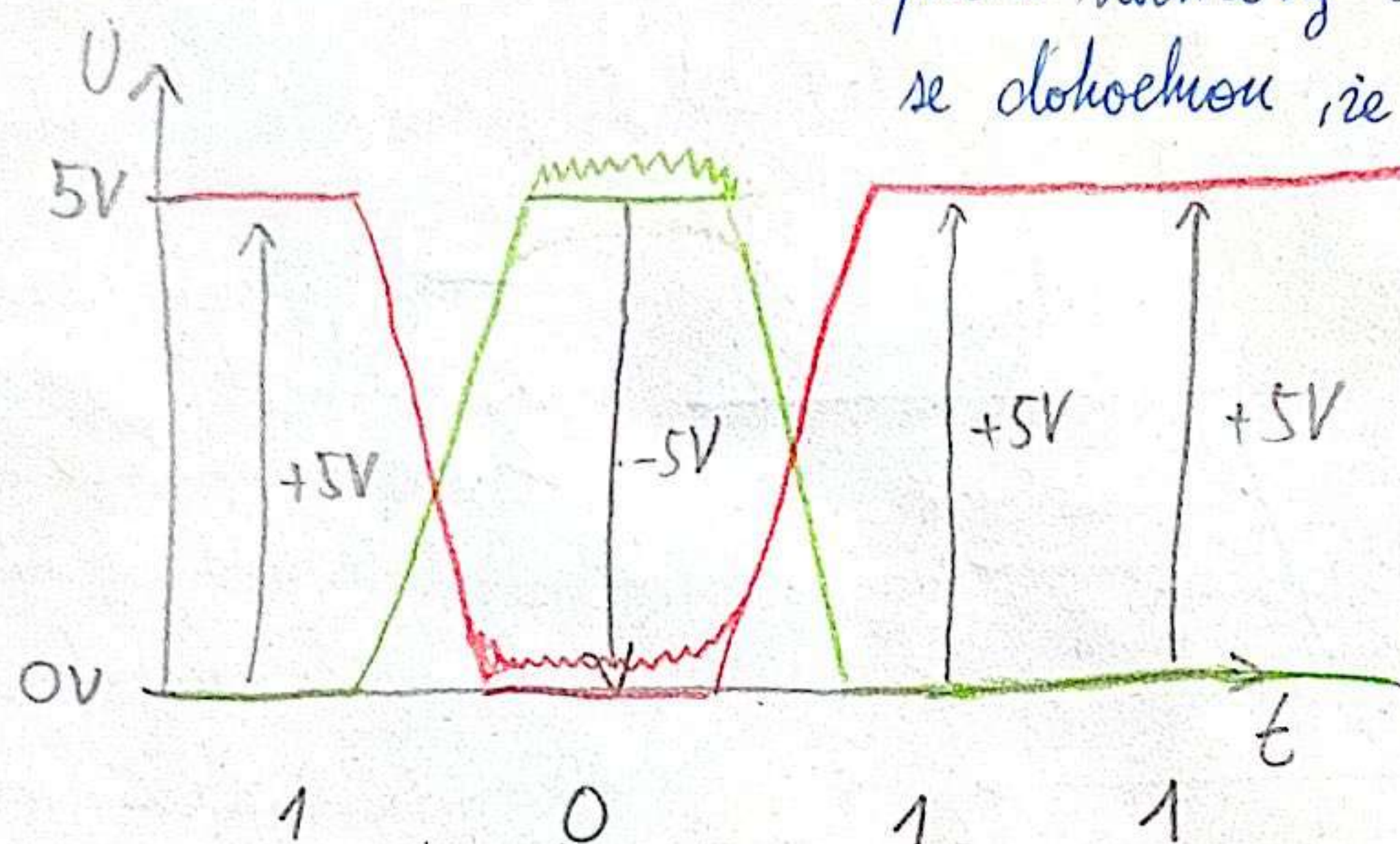
↳ 2 problémy: 1) moc vysoké napětí, což způsobí, že elektron
proletí stěnou → potřebujeme zmenšit, abychom
měli menší zařízení

2) čím menší napětí, tím blíže jsou konfidenční
intervaly, ale sám je stejný, proto můžeme
přesáhnout dovolenou mez

↳ řešení: použít 2 datové vodiče

- Diferenciální přenos

data 1 —
data 2 —



→ vysílající vysílá na obou vodičích
opačné hodnoty a přijímající a vysílající
se dohodnou, že měří rozdílné napětí

data 1 — data 2

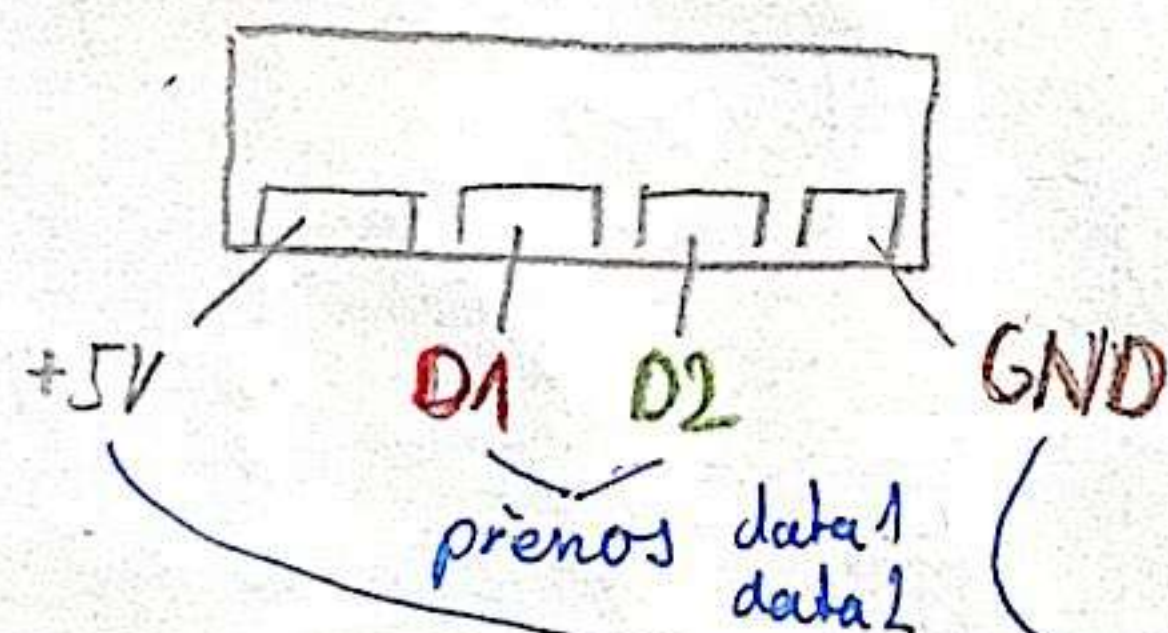
, a proto není potřeba GND

$U < 0 \rightarrow 0$ (záporné rozdílné)
 $U > 0 \rightarrow 1$ (kladné rozdílné)

↳ výhoda: elektromagnetické rušení neovlivní signál,
protože se odečte šum od prvních, ale i
druhých a opačně → rozdíl vyjde skoro stejný

- USB (type A) = „Universal Serial Bus“

↳ sériový, diferenciální přenos



→ +5V a GND slouží k napájení zařízení a ne přenos

↳ platí i pro USB 3.0 a USB Typ C → jen mají více vodičů navíc

Mocniny dvojky

$$2^0 = 1$$

$$2^8 = 256$$

$$2^{15} = 32768$$

$$2^1 = 2$$

$$2^9 = 512$$

$$2^{16} = 65536$$

$$2^2 = 4$$

$$2^{10} = 1024$$

...

$$2^3 = 8$$

$$2^{11} = 2048$$

$$2^{20} = 1\,000\,000$$

$$2^4 = 16$$

$$2^{12} = 4096$$

...

$$2^5 = 32$$

$$2^{13} = 8192$$

$$2^{24} = 16\,700\,000$$

$$2^6 = 64$$

$$2^{14} = 16384$$

...

$$2^7 = 128$$

$$2^{32} = 4\,200\,000\,000$$

$$1011_2 \Rightarrow (1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = 11_{10}$$

$$1011_2 \neq 1101_2$$

→ problém: nemá jasné, z které strany a

musí se na tom domluvit vyšlejší
a nižší mající

Most significant bit = MSB

↳ šel první po série lince

1 0 1 1

Least Significant bit = LSB

Le sel postechu' po' serione' lince

2 různé typy měření počtů: 1) MSb - first (začíná s MSb)

2) Lsb - first (rainco's Lsb)

Dohoda o dárce přenosu

↳ aby se dalo rozpoznat 00 nebo 11

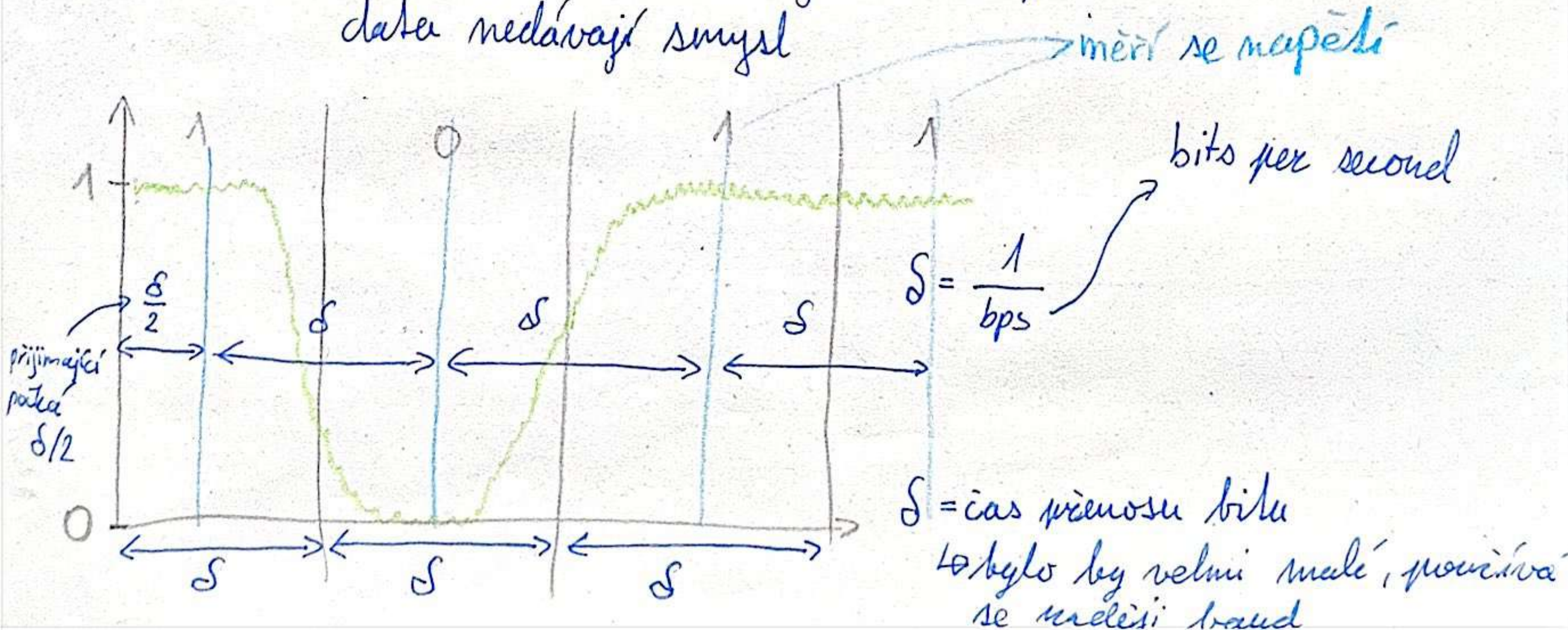
↳ délka vln musí být přibližně definována

↳ bývalí manželové

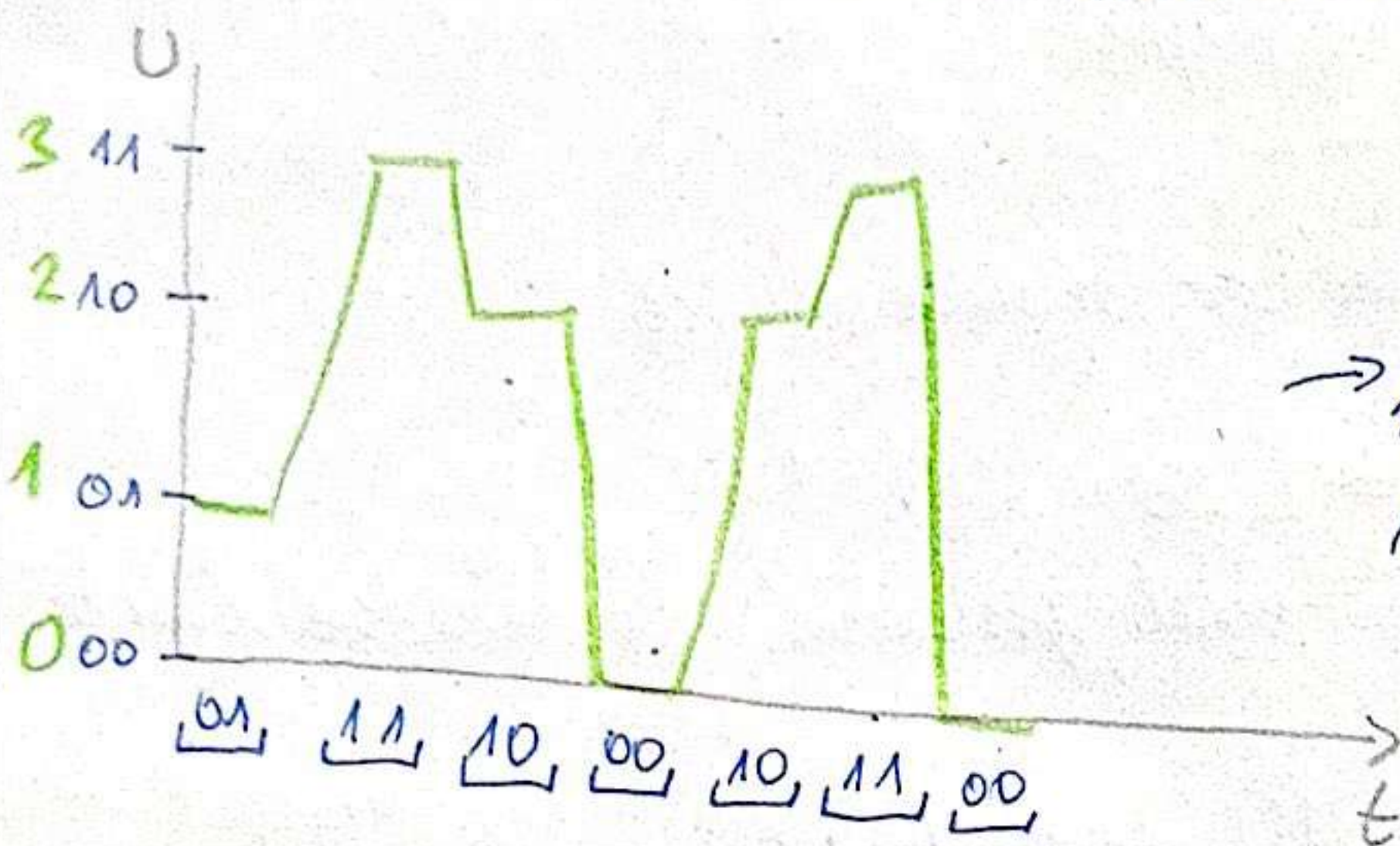
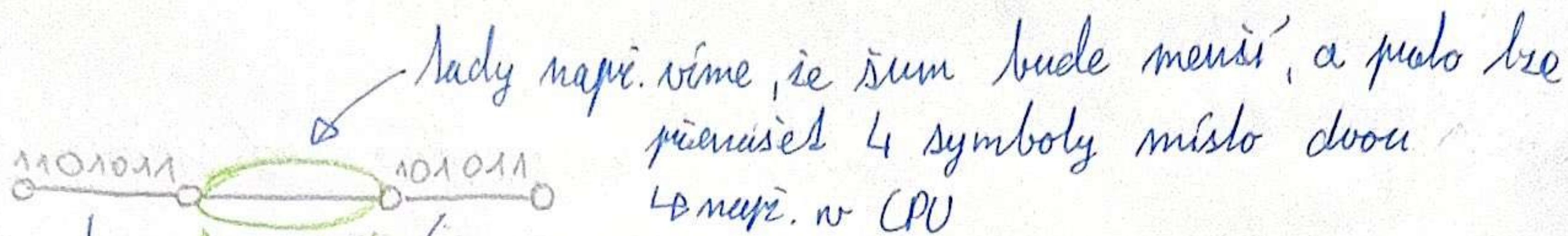
↳ přenosová rychlost („transfer rate“) nebo-li band

↳ symbols / second

↳ problém: kvetiny se časem rozjezdou a pak
data nedávají smysl



band = symboly na sekundu



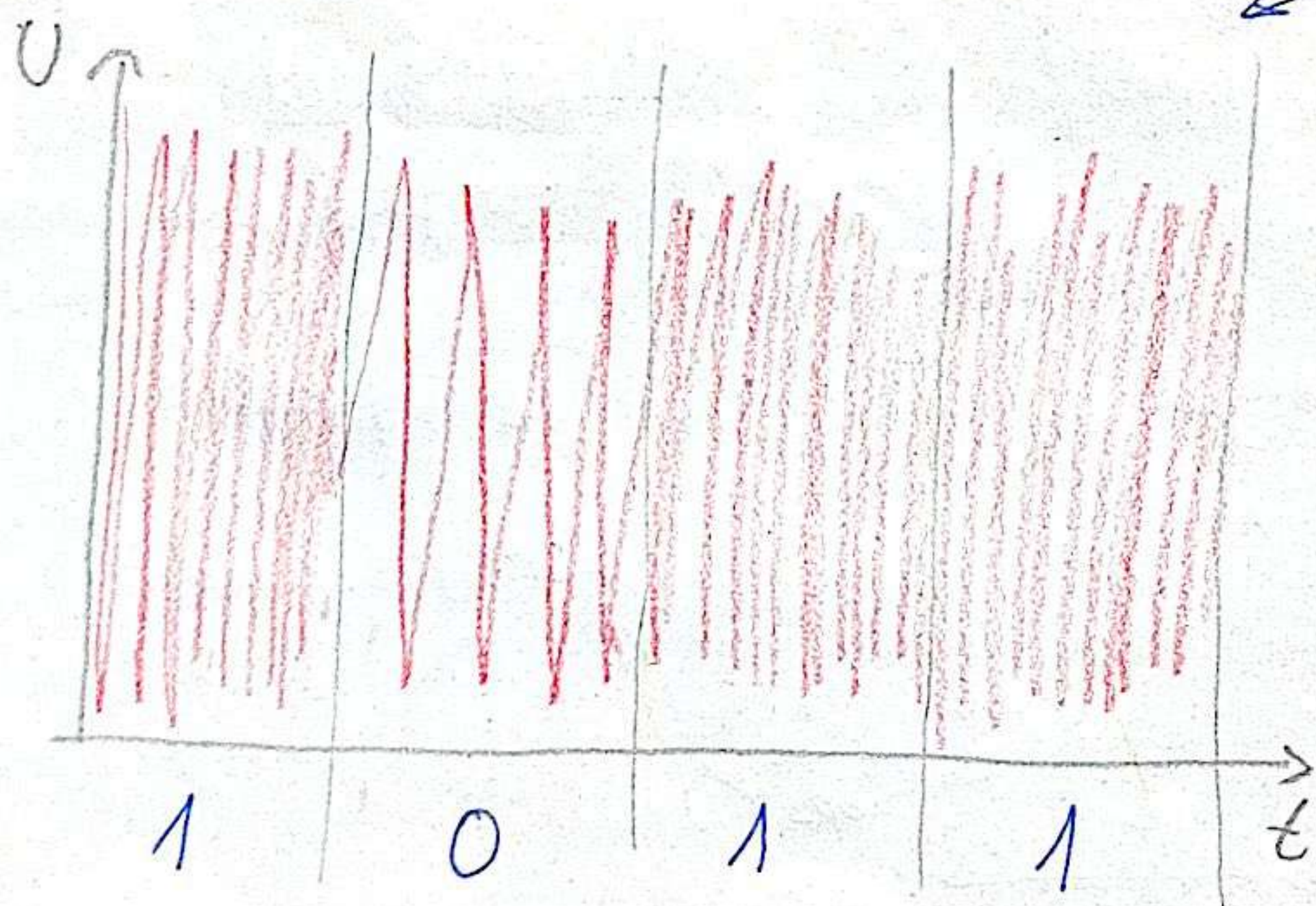
→ přenos v bodech stejná, ale bps je větší

→ přenést tuto posloupnost bitů a máme 4 symboly, proto každý symbol dostane 2 bity

Místo bps a bandu lze použít i Hz jako frekvenci modulace

$1 = 1000\text{Hz}$
 $0 = 500\text{Hz}$

} stejný digitální přenos, ale Hz udávají jak rychle se mění napětí 1 a 0

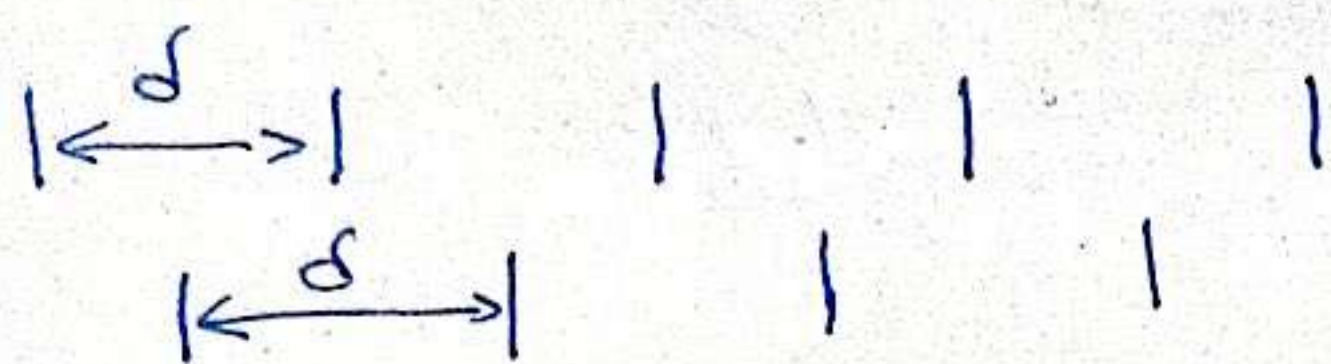


← dali by se měnit různé Hz pro různé symboly, což ovlivní bps, ale ne bandy

2. HODINA

3 možnosti řešení
problém s pevnou délkou bitu neboli baud (symboly na sekundu)

↳ oba račítají se rozdílem vzměru → to způsobí, že se můžou desynchronizovat



↑ desynchronizované → přijímač by se chřít v nejstabilnějších okamžicích

Unistorová logika → NEPOUŽÍVÁ SE



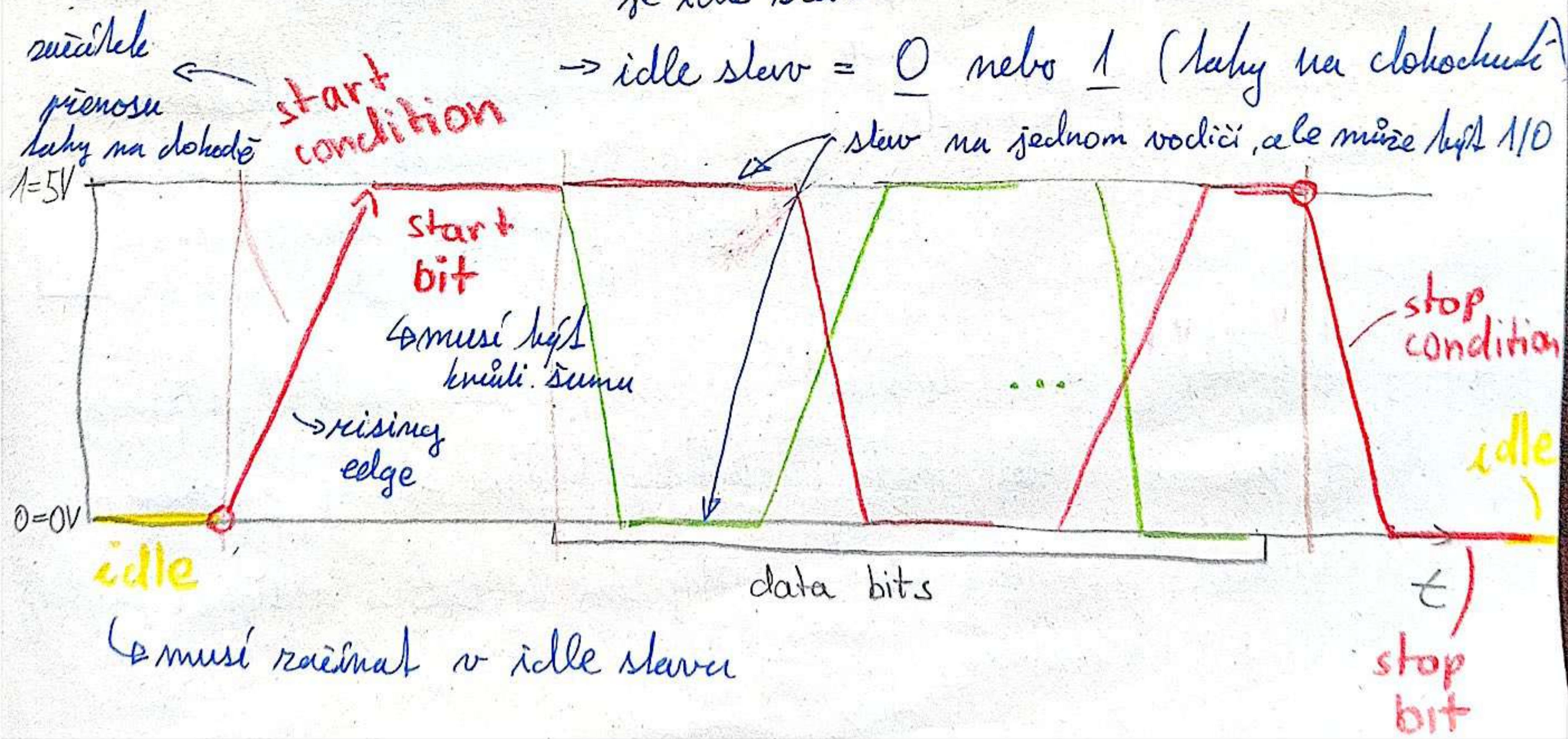
- 1) Floating = idle stav → když se přepíná z 0V na 5V (neke nic naměřit)
- 2) 0 → vysoká impedance (HI-Z)
- 3) 1

dvoustavová logika (1. možnost řešení doby přenosu)

- 1) 0
 - 2) 1
- ↳ vymyšlení idle stavu k resynchronizaci
↳ používá RS-232 linku

jak určit idle stav → musí se dohodnout, že na určitém přenosu je idle stav

→ idle stav = 0 nebo 1 (takže na dohodnutí stav na jednom vodiči, ale může být 1/0)



↳ musí račítat v idle stavu

hodiny se desynchronizují (kvůli nepřesnosti hodin)

↳ omezení délky přenosu na 8 bitů (1 B)

↳ 1 B jsou 2 šestnáctkové
recepty

↳ vysoký overhead (20%) z 10 bitů jen 8 bitů datových
celkem

$$\underbrace{1000}_{\text{baud}} \cdot \underbrace{8/10}_{\text{datových / celkově}} = 800 \text{ bps}$$

+1 start bit
+1 stop bit

Hodinový signál - (2. možnost řešení doby přenosu)

↳ používá I²C linka

↳ vytvoří nový (referenční) vodič s hodinovým signálem,
proto se (synchronizuje) na každém bitu

data —

GND —

CLOCK —

→ budou se tam pořád střídát 0 a 1 => 0101010101...
→ pro synchronizaci

Diferenciální přenos

data 1 —

data 2 —

clock 1 —

clock 2 —

} rozdíl napětí

} rozdíl napětí
namísto

~~clock~~
~~GND~~

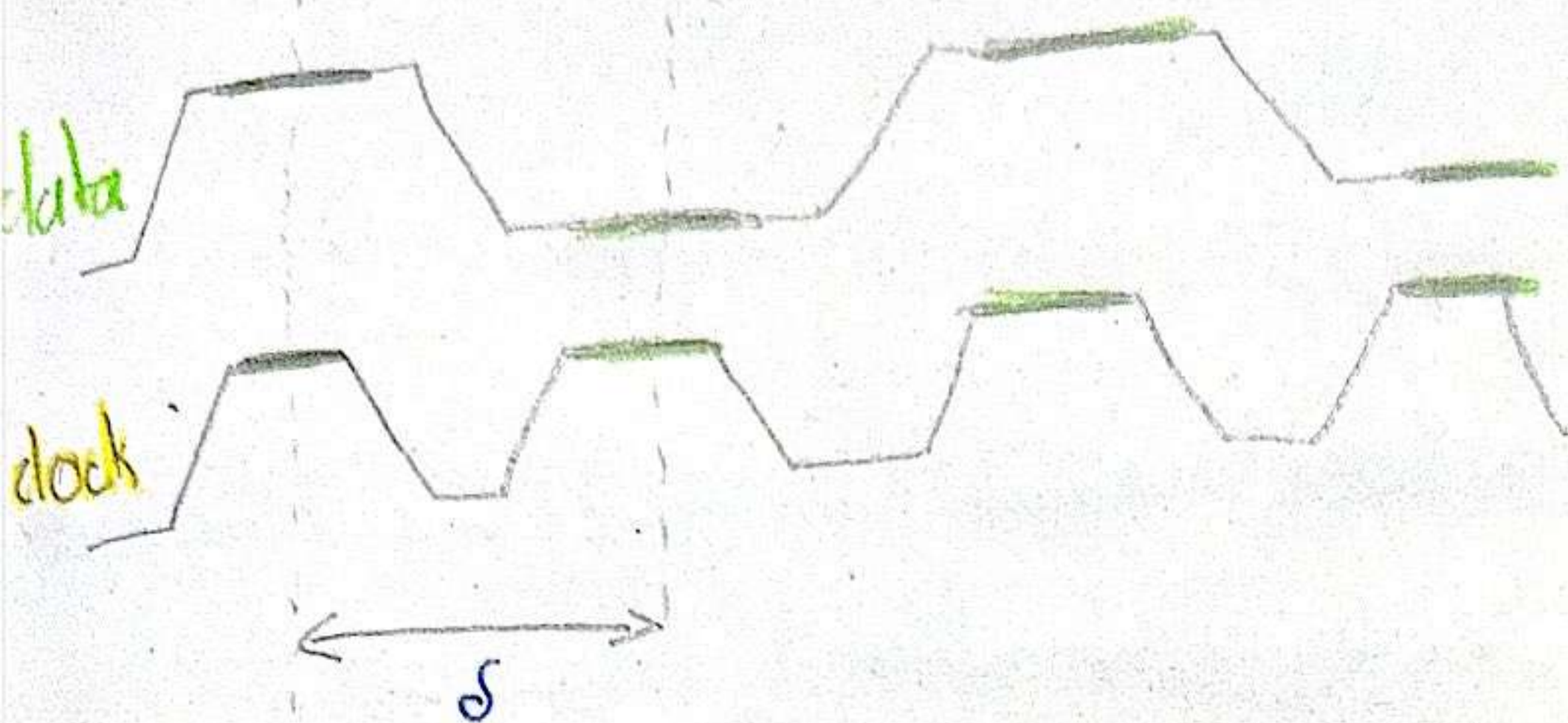
→ protože se bude zase
měřit rozdíl napětí,
aby se zbavilo šumu

výhody: → vysílající může měřit a zmenšovat rychlost přenosu

nevýhody: → vysílající musí mít dostatečně přesné stopky a reboj clock

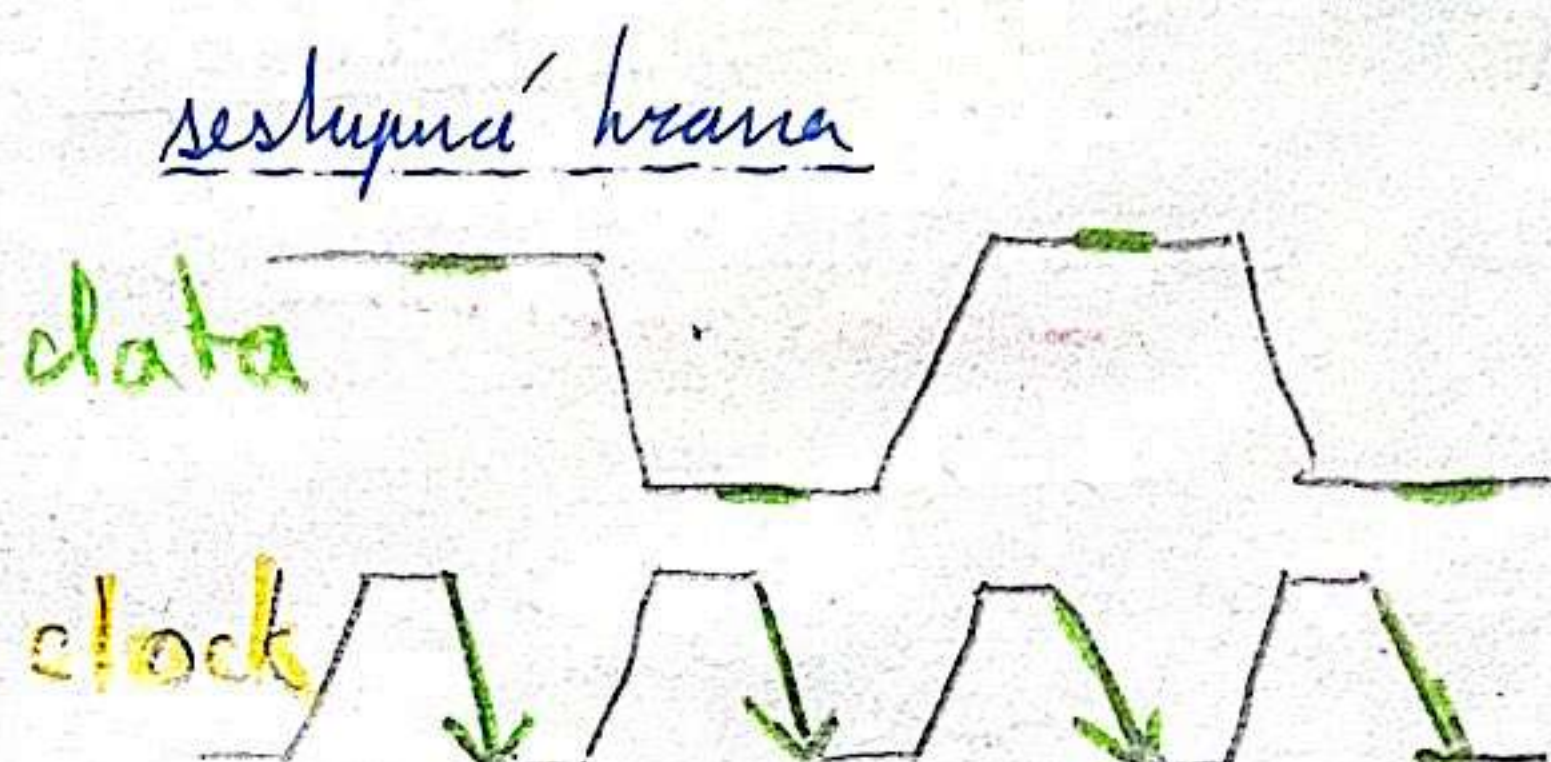
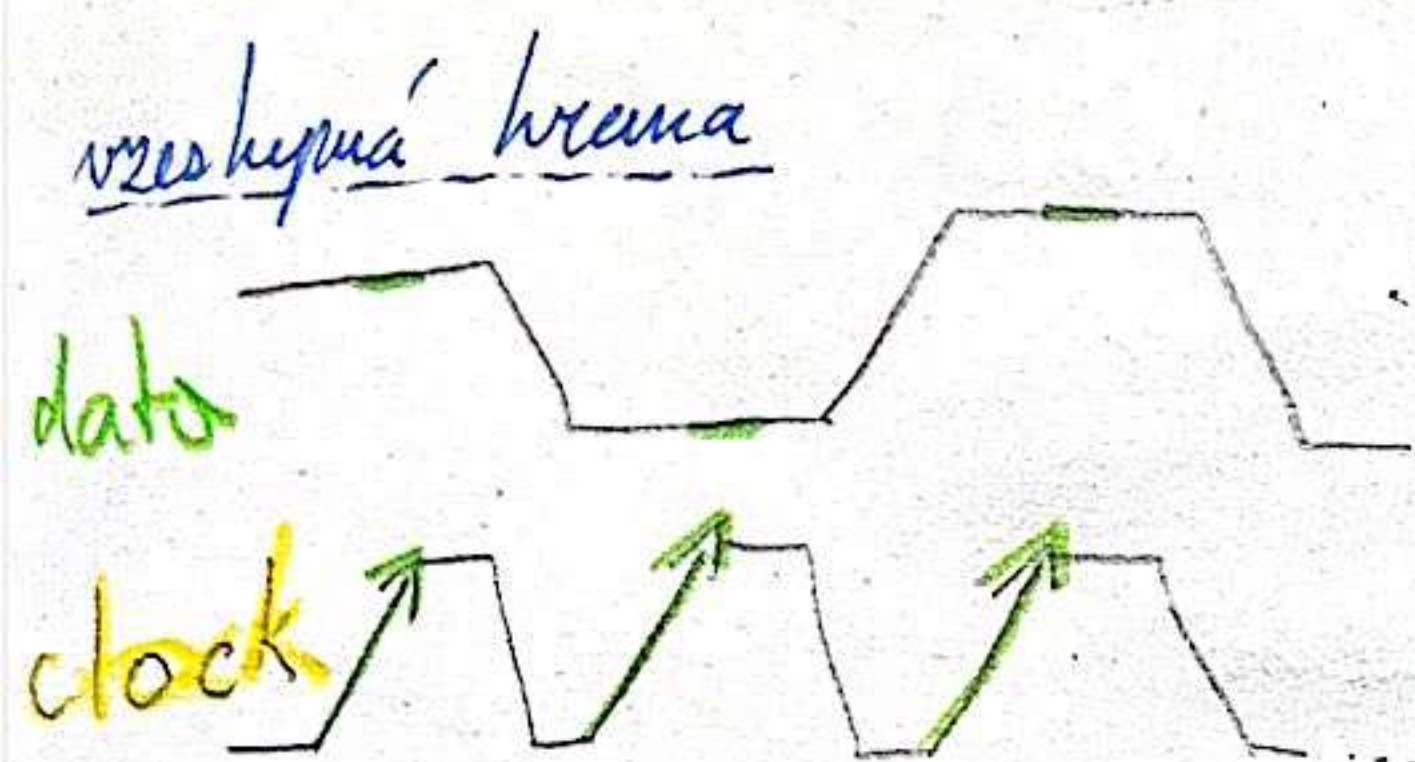
→ řešení: nahradit stopky limitujícím krystalem a signál přijímat
obě strany

→ další vodič → asi 50% overhead



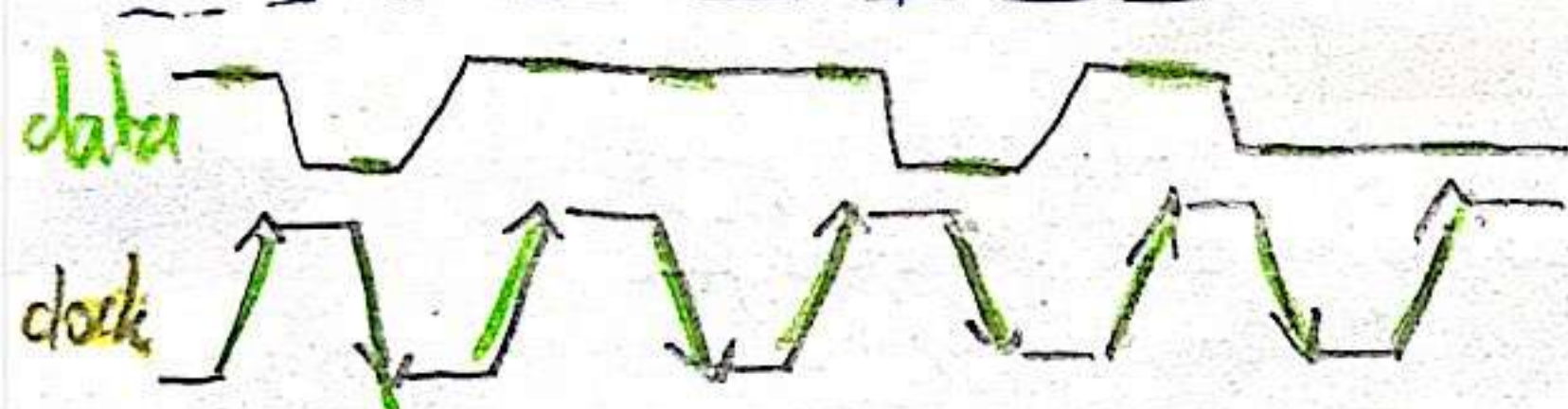
- hodinový signál to musí generovat např. tak, že klyklotin je tam 1, tak se může představit na data signál a je to platný

ale lépe se měří změny napětí, než stálá hodnota



ale! datový vodič má pomalejší frekvenci, řešení:

vzestupná a sestupná hrana



SDR (single data rate)

→ dvojitá rychlost přenosu
DDR (double data rate)

Příběžná korekce (3. možnost řešení doby přenosu)

↳ pravidelně resynchronizovat hodinový vysílajícího a přijímajícího.

↳ např. v signálu 1101110101 - **hrany** se v hranách dá resetovat, ale musí být počíná signál

↳ možnosti přenosu 8-bitů : $2^8 = 256$

00000000

11011001

11111111

} všechny kombinace, některé přelíné, jiné ne

možnost přenosu 10-bitů : $2^{10} = 1024$

000000000000

1101101101

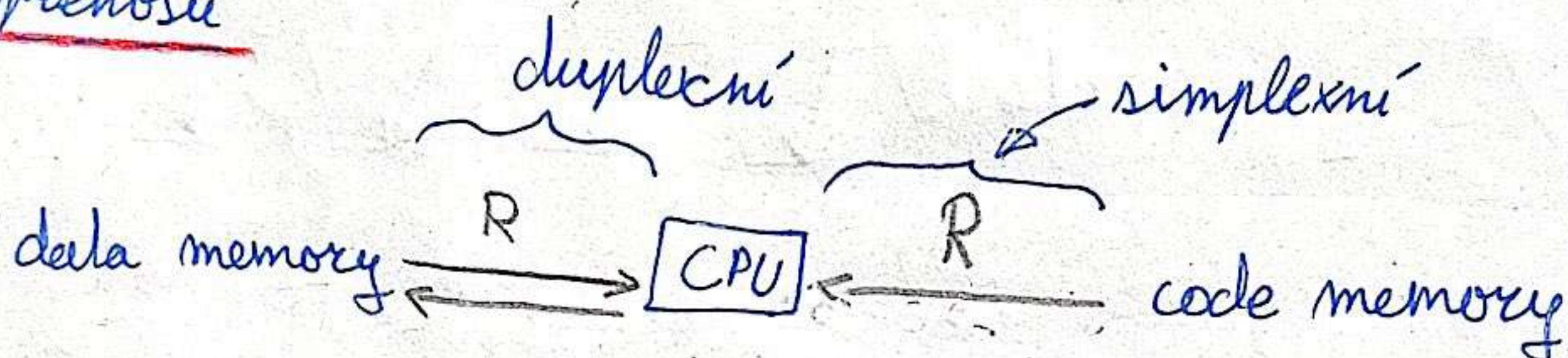
1111111111

} taky některé nepřekročí, ale vyčísleme musíme
aby bylo 256 různých možností

Všechny 8-bit kombinace namapujeme je na hezké kombinace 10-bit čísel → tzv. koncept: CLOCK RECOVERY

↳ ale pořád 20% overhead ale lze zmenšit

Typy přenosu



1) simplexní přenos

↳ přenos jen jedním směrem (přijímající a vysílající)

2) duplexní přenos → přenos oběma směry

a) half-duplex

↳ jeden vodič, ale obě strany se musí domluvit, kdo bude rovnou přijímat a kdo vysílat

b) full-duplex

↳ nezávislé přenosy jedním a nezávislé druhým směrem

↳ vlastně 2 simplexní přenosy

↳ příklad RS-232 (také tzv. sériová linka) & definovaná délka přenosu, není potřeba clock



out of band signals

↳ další linky v RS-232



↳ vysílají nezávisle i během přenosu jiné informace

↳ např. dochází baterka / změna módu / napájení zařízení „signal“

↳ např. POWER-OUT : 1 = pravda
0 = nepravda } nějaký státní signal → SIG

↳ používá se i inverzní logika 0,1 protokolu

značení → SIG /SIG #SIG !SIG [SIG]
v dokumentaci

hodiny reálného času (RTC)

↳ ví, kolik je hodin, minut a sekund, občas taky v lince

↳ např. v počítači i když je notebook vyplý, tak malá baterka je napájí

Komunikační protokol

den 9b měsíc 5b rok 7b

← nesmyslné, správné v SA úlohách

MSb 101101011 LSb 10110 1011010

↳ je definován, aby

se rozdělilo, co bylo posláno za den / měsíc / rok

↳ blok dat = pa(c)ket

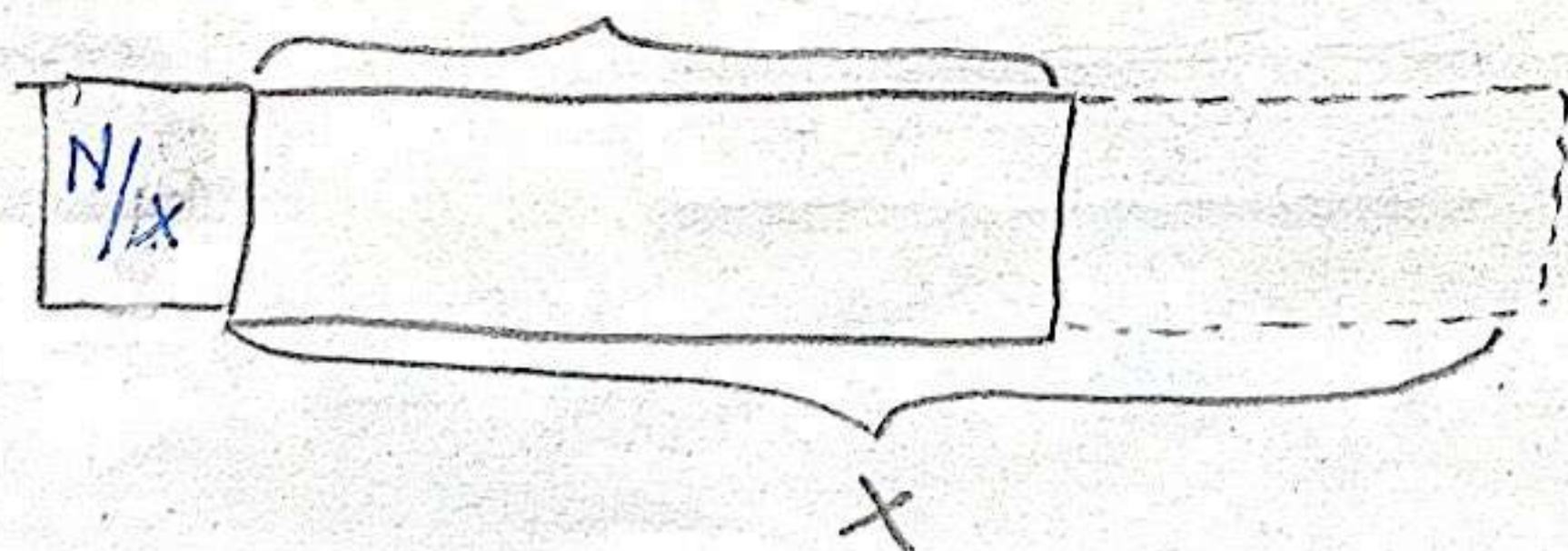
pa(c)ket

↳ musí se definovat i pořadí, např. MSb a LSb

← LSb-first používá RS-232

↳ bude se muset rozsekat do několika přenosů (x-bitů)

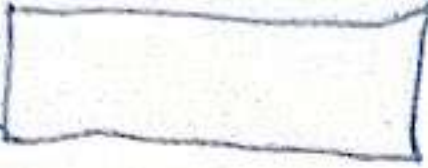
např. při přenosu pomocí RS-232 linky

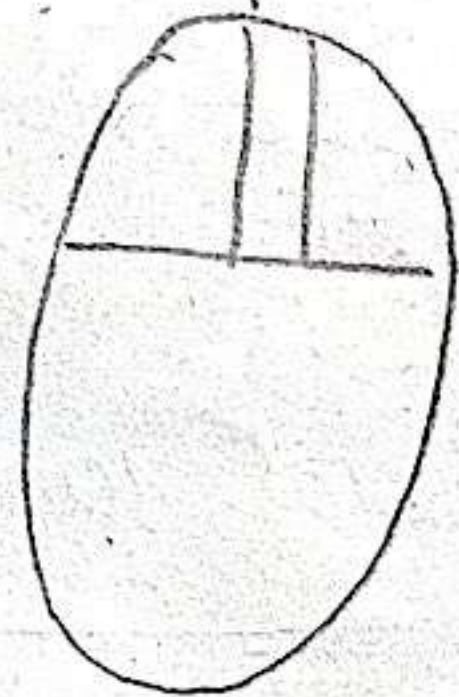
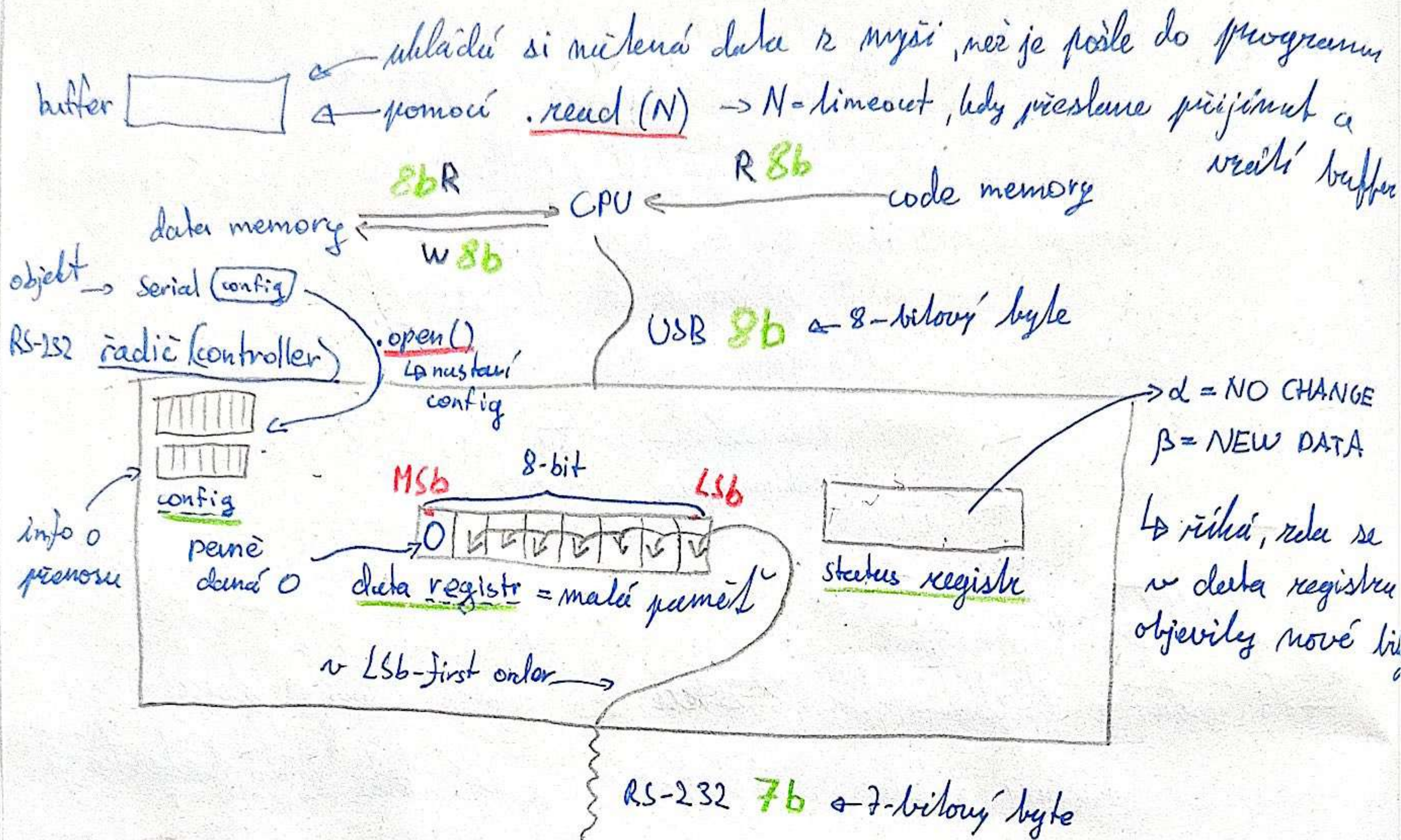


→ konstanta, co definuje délku paketu - nemusí být vždy stejně dlouhé

problém: starší řešení má $AB=7b$

→ řešení: řadič (controller) přijme data ze zařízení a zpracuje je

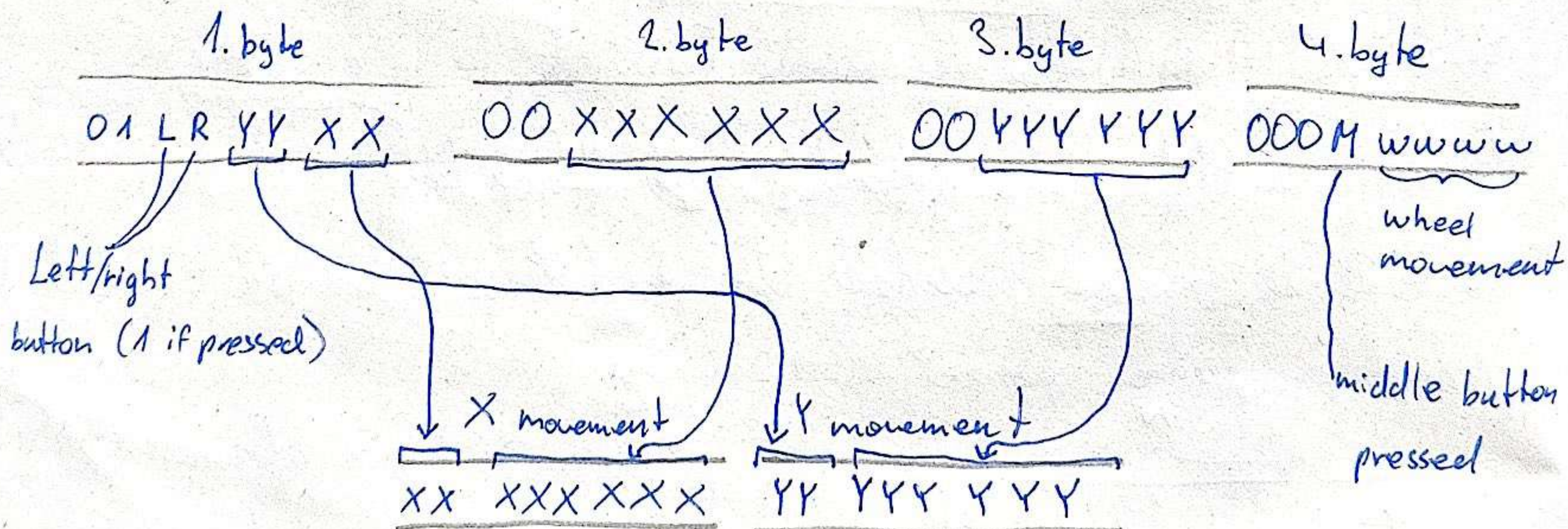
buffer  ukládá si přijatá data z myši, než je pošle do programu
pomocí read(N) → N = timeout, kdy přestane přijímat a vrátí buffer



nainstalujeme si knihovnu (pyserial) - pro komunikaci s RS-232

modul serial → objekt Serial

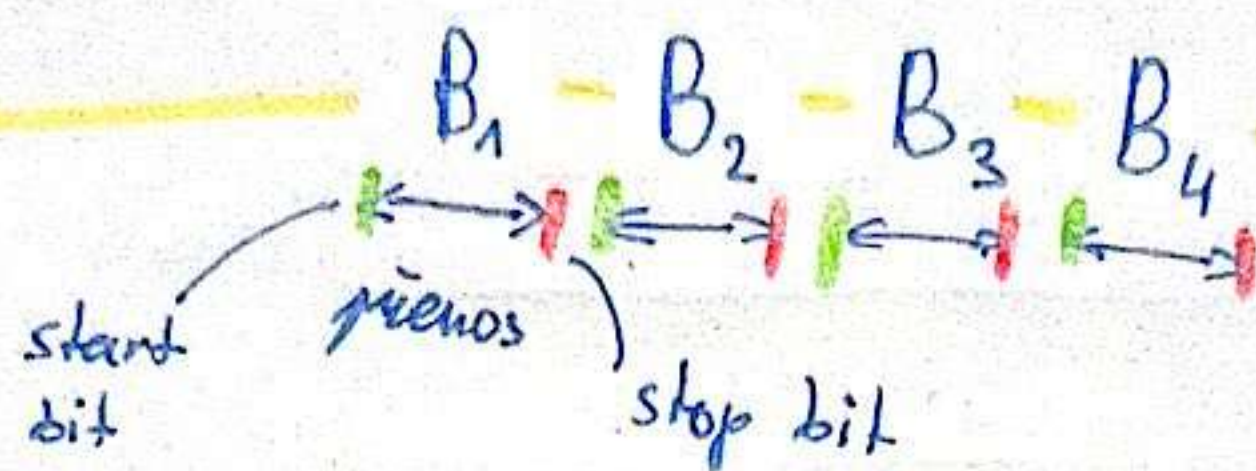
Paklketá myši přes RS-232 linku



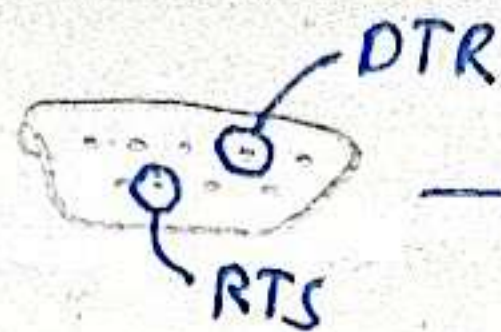
→ posílá směrny polohy, jinak je v idle stavu a nic neposílá a jednou pošle 4 byty a zase idle

3. HODINA

RS-232 linka



out of band signals



→ na napájení myši

napájení $\left. \begin{matrix} \text{RTS} & 1 \\ \text{DTR} & 1 \end{matrix} \right\} 2$, aby napájení bylo rozloženo

↳ musíme tuhle konfiguraci nakonfigurovat v řadiči

control register



→ dělal za nás objekt Serial

→ má funkci $\text{set DTR} \left(\begin{matrix} \text{true} = 1 \\ \text{false} = 0 \end{matrix} \right)$
 $\text{set RTS} \left(\begin{matrix} \text{true} = 1 \\ \text{false} = 0 \end{matrix} \right)$

musím ale chvíli počkat, než se vypne
 nastavit oba na false → myš se vypne
 nastavit na true → pro myš nový vstup

`read()` **blokující** → dokud nedostane plný počet bytů (např. 1024)

`read(N)` **semi-blokující** → čekám timeout, jinak беру, co mám
 (→ timeout)

`read()`
~~`input()`~~ **blokující**, místo toho keyboard.is_pressed('L')

↳ vraácí : - True
 - False

$$96 = 64 + 32 \text{ (B1)}$$

$$64 = 64 \text{ (B2)}$$

$$112 = 96 + 16 \text{ (B3)}$$

byty od myši: → podle kom. protokolu (2 strany přenosu)



= 96 → levé stisknutí



= 64 → žádné stisknutí



= 112 → oba stisknutí

ta jednotka indikuje první byt v paketu,

aby nedošlo k chybě při více přenosech, aby

zjistili co je první

Šestnáctková soustava (hex/hexa/hexadecimal)

738201600 → těžké převést do binární soustavy

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

a, b, c, d, e, f → ke i malá písmenka

10 11 12 13 14 15

Řeší se:

\$23

23h

0x23 C#, Python, C/C++

23₁₆

23_{hex}

~ desítkové:

23

23_{dec}

23₁₀

~ binární

10110_{bin}

0b10110

10110₂

$$23_{16} = 3 \cdot 16^0 + 2 \cdot 16^1 = 35_{10}$$

$$123_{16} = 1 \cdot 256 + 2 \cdot 16 + 3 \cdot 1 = 291_{10}$$

mocniny šestnáctky: $16^0 = 1$

$16^1 = 16$

$16^2 = 256$

$16^3 = 4096$

$16^4 = 65536$

n-bit číslo

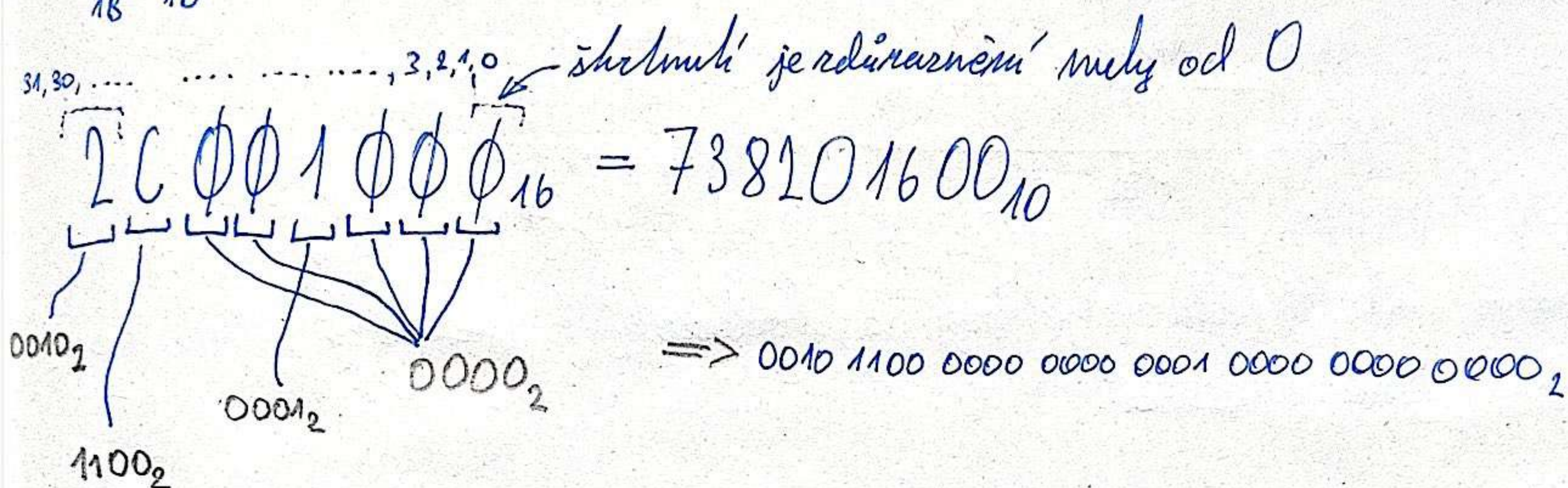
$$0 - 2^n - 1$$

4bit $\rightarrow 0 - 15$ odpovídá jedné hexadecimální číslici

1B = 8bit $\rightarrow 0 - 255$ odpovídá dvěma hexadecimálním číslicím

↳ leading zeros

0123 $\rightarrow$ uložíme pomocí 2 bytů



↳ uložíme jen jednotlivé hex číslice

$a = 35$ $\rightarrow$ bin $\leftarrow$ stejná hodnota proměnné

$a = 0x23$

print(a) $\rightarrow$ (35)

51 53 $\leftarrow$ uloženo jako kód

1B	1B
86	86

hex(a) $\rightarrow$ 0x23

1B 1B 1B 1B

4B

na 1 číslice jinou leading zeros : $0x23 \rightarrow '23'$

$3 \rightarrow '03'$

$291 \rightarrow '123'$

format(a, '02X') velká písmena v hex

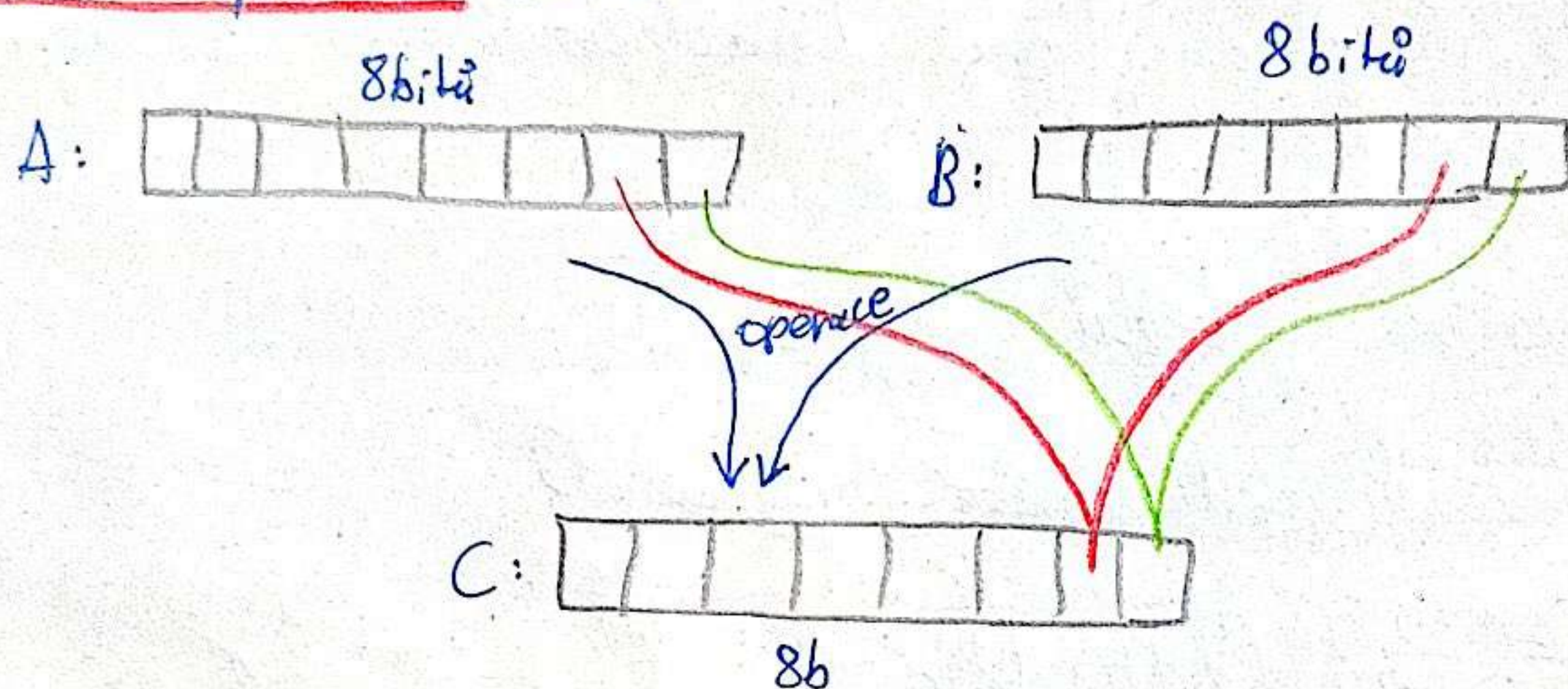
x - malá písmena v hex

aktivní čekání (špatně) × pasivní čekání (dobře)

↳ v programu např. cyklus

v šestnáctkové soustavě jsou čísla čitelnější v komunikačním protokolu
myši

Bitové operace (n-bitové)



↳ pracuje bit po bitu, vykářejí z booleanských operátorů, bity mezi sebou nijak nesouvisí

True=1
False=0

OR

↳ Python, C# → |

↳ Python or (není bin. operace, ale booleanská mezi True a False)

A: 1010 hodnota

B: 1100 příkaz

1110

} podívám se na příkaz a { 0: nech to máhově
1: napiš 1 ("set to 1")

↳ je to tzv. explicitní nastavení bitů → příkaz rozhodne, co se rozhodne

AND

↳ Python, C# → &

↳ Python and (zase není bitová operace)

A: 1010 hodnota

B: 1100 příkaz

1000

} podle příkazu { 0: napiš 0 ("reset to 0")
1: nech to máhově

NOT (bitová operace s jedním argumentem)

↳ Python, C# → ~

↳ malá na velikosti bitů lahý: stejné hodnoty { 1100 (4b)
001100 (6b)

$$\text{NOT: } \begin{array}{r} 1100 \\ 0011 \end{array} = \begin{array}{r} 0011 \\ 1100 \end{array}$$

XOR (EOR) → exkluzivní or

↳ Python, C# → ^

$$\begin{array}{r} A: 1010 \text{ hodnota} \\ B: 1100 \text{ příkaz} \\ \hline 0110 \end{array} \quad \left. \vphantom{\begin{array}{r} A: 1010 \\ B: 1100 \end{array}} \right\} \text{ podle příkazu } \begin{cases} 0: \text{nech to nahoru (hodnota)} \\ 1: \text{NOT bitu nahoru (hodnota)} \end{cases}$$

Bitová maska

↳ komunikační protokol myši

B1: 00100000

AND 00100000 ← bitová maska

00100000
00100000 → pak je slishnuté levě tlačítko

$b1 \& 0x10 \neq 0$

A pro posun myši z protokolu: (4 strany zprávy)

B1: ?????? XX

B2: ?? XXXXXX

AND 00000011

AND 00111111

000000XX

00XXXXXX

(b1 & 0x03)

(b2 & 0x3F)

→ XXXXXXXX

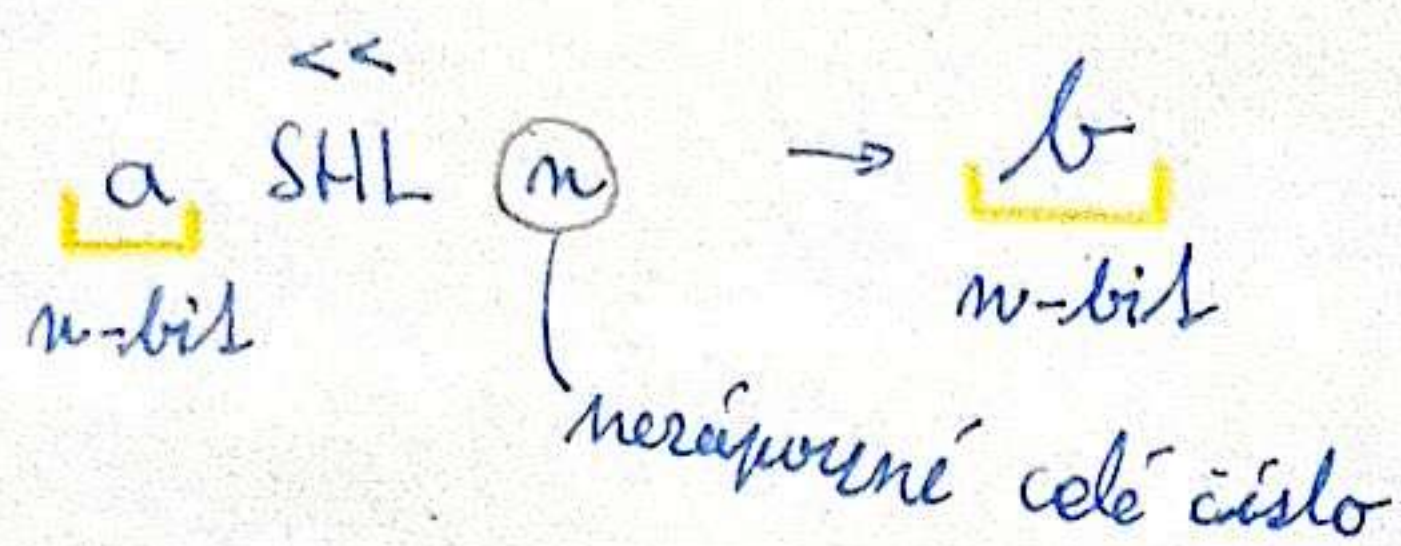
↳ 1 provést až po bitovém posunu

4. HODINA

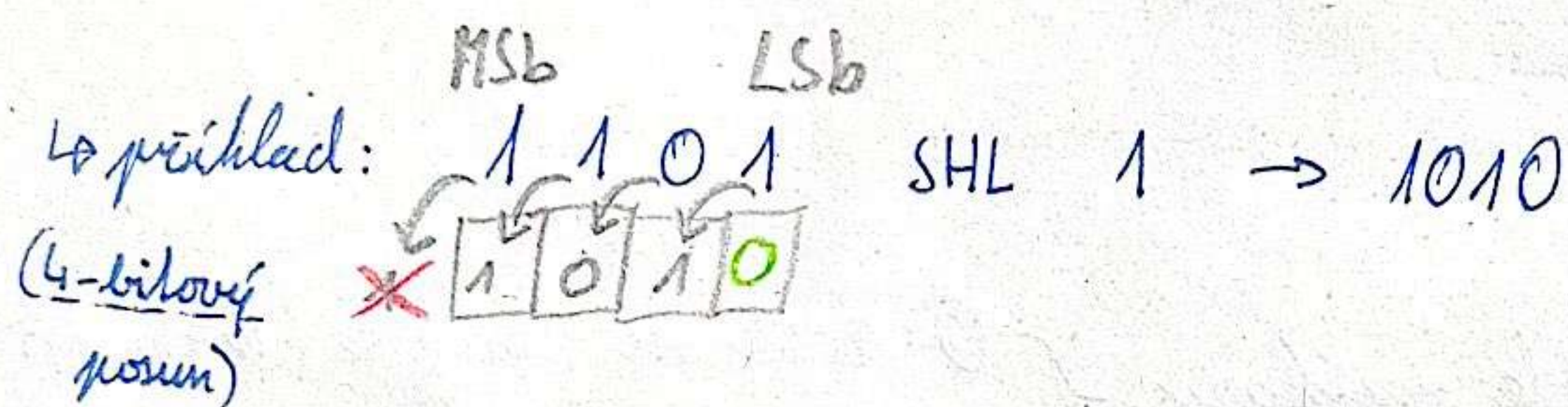
bitové posuny / bitwise shift

↖ Python / C / C#

① posun doleva / shift left / SHL / <<



↳ posun k MSb !

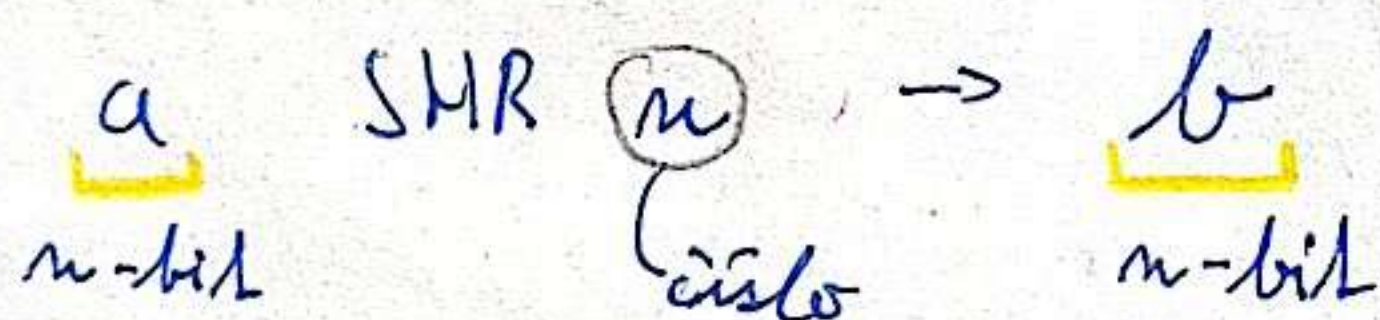


$1101 \xrightarrow{\text{SHL } 3} 1000$

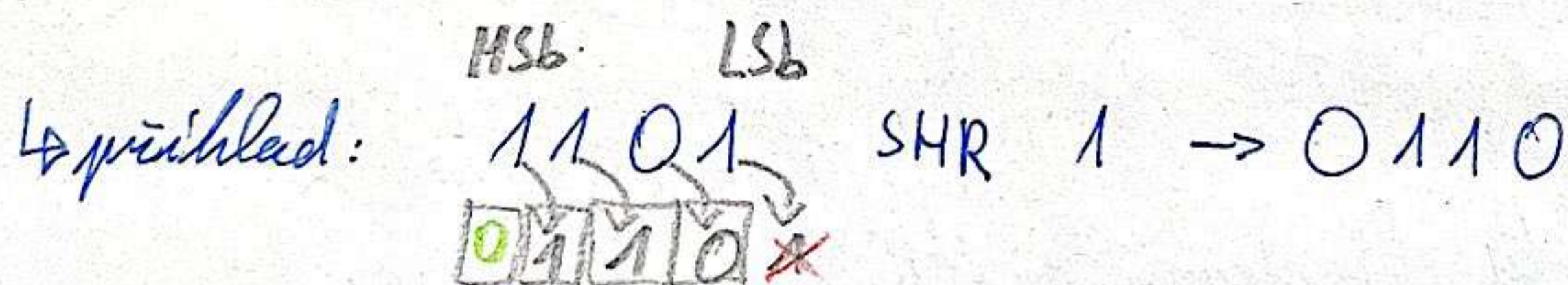
$1101 \xrightarrow{\text{SHL } 4} 0000$

② posun doprava / shift right / SHR / >>

↖ Python / C / C#

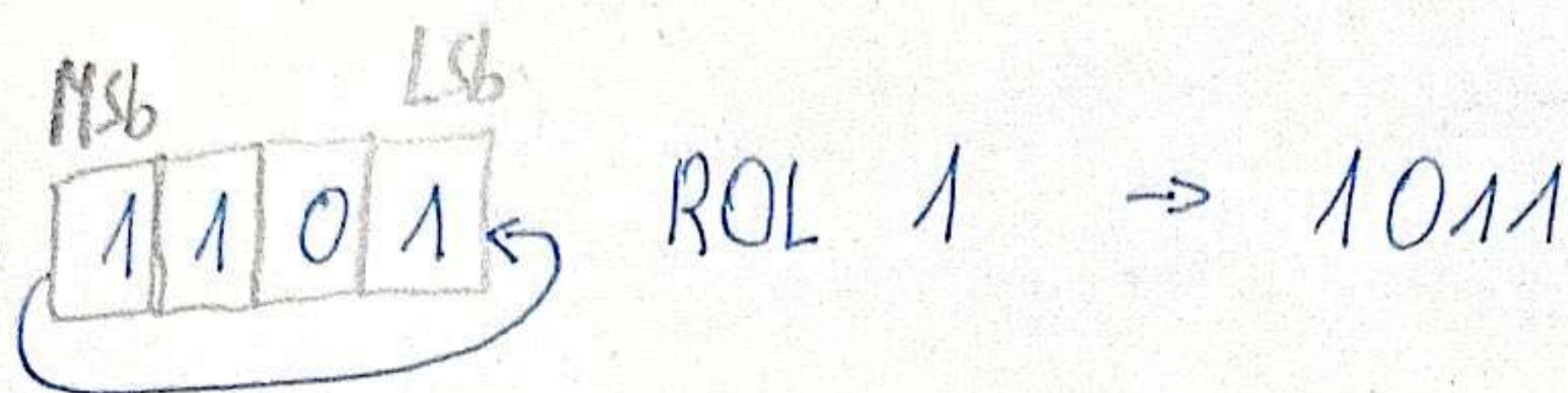


↳ posun k LSB !



nejsem v Python / C / C#, ale má je procesor

③ rotace doleva / ROL / rotate left

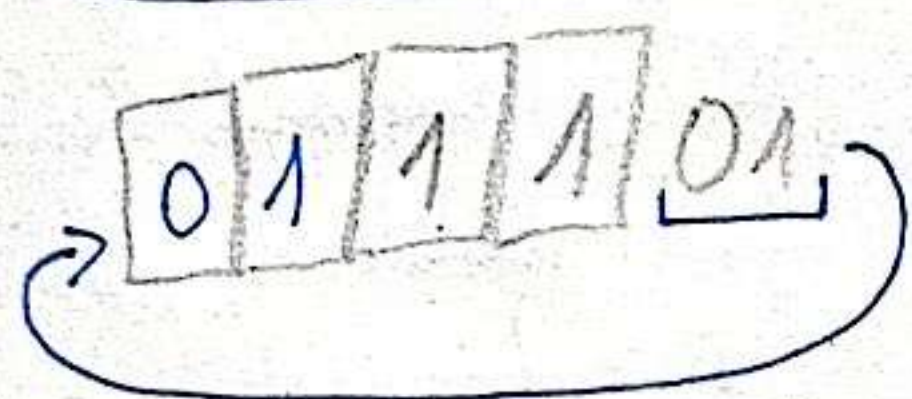
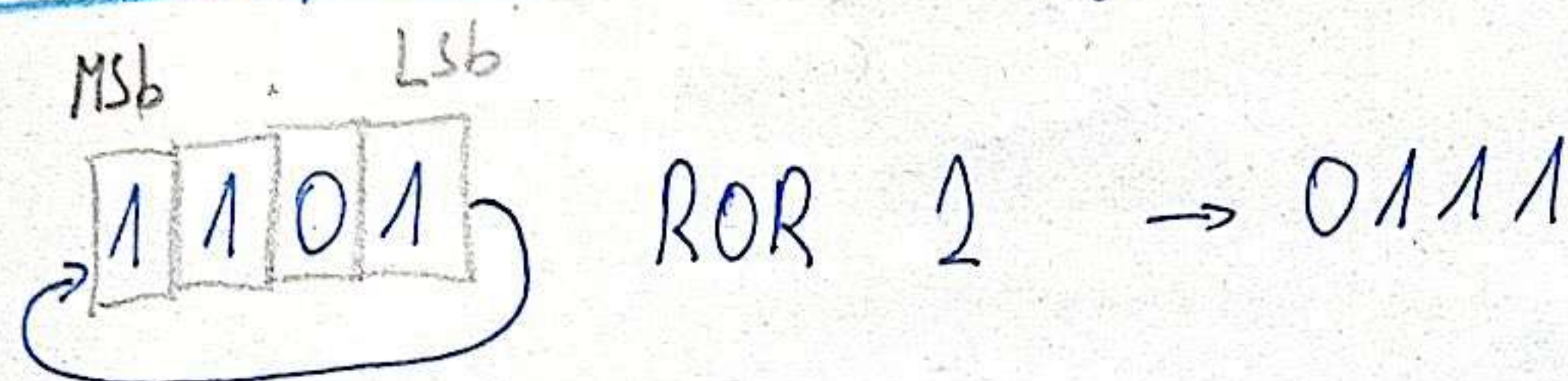


↳ rotuje k MSb! , stejně jako posun, jen se bity nemýlí, ale

přesunou na pozici LSB, v pořadí jakém se vysouvalo se ho

↳ příklad: 1101 ROL 3 → 1110

④ rotace doprava / ROR / rotate right

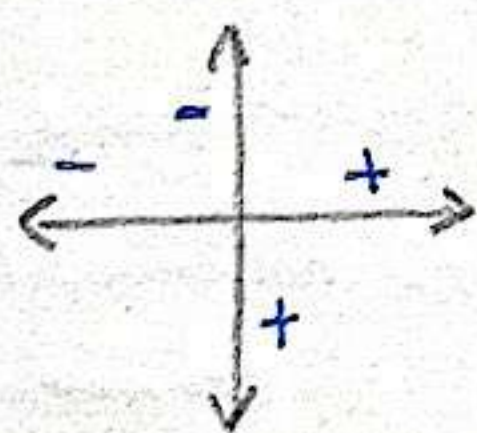


↳ rotuje k LSB!

k protokolu myši → 2 stránky zpátky pomocí bitových posunů:

$((b1 \& 0x03) \ll 6) | (b2 \& 0x3F)$ → viz 2 stránky zpět

posun myši:



↖ změny polohy, co by měla myš posílat při posunu

↳ proto při posunu do "-" hodnot dostaneme velká čísla kolem

255, protože máme jen beznaménková čísla a 0-10 je 245, ..

bernámenkova / unsigned

↳ n -bitové číslo $\rightarrow$ hodnoty: $0 - (2^n - 1)$

↳ 8-bit $\rightarrow$ hodnoty: $0 - 255$

znaménková / signed

1) celá čísla, ale první bit je znaménko (Signed Magnitude)

$$-5_{\text{dec}} = \ominus 101_{\text{bin}}$$

$\left\{ \begin{array}{l} - = 1 \\ + = 0 \end{array} \right.$

$$-5_{\text{dec}} = \underbrace{1}_{\text{znaménko}} \underbrace{101}_{\text{hodnota}}_{\text{bin}}$$

↳ na rozlišení

↳ používá $(n-1)$ bitové hodnoty na n -bitech, MSB je znaménkový bit - kvůli kompatibilitě s unsigned

↳ 8-bitová hodnota:

$$-5: 10000101$$

$$5: 00000101$$

$$-6: 10000110$$

$$6: 00000110$$

↳ rozsah hodnot: $-127 - +127$

↳ převádění čísel

$$-5: 10000101$$

$$-6: 10000110$$

} jako bernámenková: $10000101 = 133$
 $10000110 = 134$ ^ $\rightarrow$ což není pravda u znaménkových

proto nové znaménkové přeměny

nebo rychlost operací

↳ aritmetika

$$-5: 10000101 + 1 (\text{unsigned}) \rightarrow \underline{10000110}$$

↓
-6, což zase není pravda :)

2) one's complement (jedničkový doplněk)

klučová: $+$ = $(n-1)$ bitová hodnota

$-$ = $(n-1)$ bitová hodnota v NEGACI

$$5 = 00000101$$

$$-5 = 11111010 \leftarrow \text{v jedničkovém doplněku}$$

$$-6 = 11111001$$

↳ důsledek negace: 1 na MSb definuje znaménko -

porovnání

$-5: 11111010$
 $-6: 11111001$

porovnáváme jen odlišné části

↳ funguje, ale jen pro stejná znaménka, nelze porovnávat

např. 5 a -5

sčítání

$$-5: 11111010 + 1 \rightarrow -4: 11111011 > \text{funguje}$$

jak provést negaci čísla

- u signed magnitude: a XOR $\Phi \times 8\Phi \rightarrow$ znegovat jen MSb

↳ $00000000 \text{ XOR } \Phi \times 8\Phi \rightarrow 10000000 = \underline{-0}$

- u one's complement: NOT(a)

↳ neexistuje

↳ $00000000 \rightarrow \text{NOT} \rightarrow \underline{11111111} = \underline{-0}$

po negaci
0

↳ neexistuje

3) two's complement

↳ obravíme se nuly -0 z technického důvodu, kde $-0 = 11111111$,

přičteme 1 $\rightarrow 100000000$

vezmeme jen nulu část

$+= (n-1)$ bit hodnota

$- = \text{NOT}(\text{hodnota}) + 1$
 $\uparrow$
 $(n-1)$ bit hodnota

-5: 11110111

5: 00001011

$\rightarrow \text{NOT} \rightarrow 11110110 + 1 \rightarrow 11110111$

↖ jak vypadala -4 v one's complement

-6: 11111010 ↖ jak vypadala -5 v one's complement } posunulo

proto -127 vypadá jako -126, a tedy je větší rozsah $(2^{n-1}) - (2^{n-1}) - 1$

-128 — 0 — +127

$\uparrow$
ječná nula

Representace čísel v Pythonu

00000101 | 4

unsigned číslo (počet platných bitů) $\rightarrow$ 32-bit

$\uparrow$
uložené číslo 5

254 = 011111110

9-bit

$\rightarrow$ Python musí použít 2 byty

in two's complement

~~0~~ | 11111110 | 9

$\rightarrow$ 32-bit unsigned

$\rightarrow$ `bit_length()` — vrátí 8

254 → zapisat záporně -254

254: $\begin{array}{r} 0011111110 \\ 0100000000 \\ \hline 0111111110 \end{array}$ ← bitová maska $\Phi x 1 \Phi \Phi$ → vykonám 1 shift
operace OR ; 1

ale Python zase uloží jako kluché číslo

limit čísel v Pythonu 2^{32} , protože 2^{32} je bit-length

Kritická chyba

$\left. \begin{array}{l} \text{uint 8} \\ \text{uint 16} \\ \text{uint 32} \\ \text{uint 64} \end{array} \right\}$ unsigned + počet bitů v paměti

$\left. \begin{array}{l} \text{int 8} \\ \text{int 16} \\ \text{int 32} \\ \text{int 64} \end{array} \right\}$ znaménková

$\text{uint 8}(\overset{\text{int}}{a}) \rightarrow$ provede konverzi

Truncation → pozor na velká čísla

8-bit 5: 00000101 → 4-bit 0101

8-bit 21: 00010101 → 4-bit 0101 → přijdeme o informaci

šlo by to udělat $x \bmod 2^n$, nebo-li pro 21 $\bmod 16 = 5$

Truncation

↖ znegujeme a přidáme jedničku

-5: 11111011 → 4-bit 1011 = -5

, ale -128: 10000000 → 4-bit 0000 = 0

beznaménkové rozšíření (zero extension) unsigned $\rightarrow$ unsigned } kdy
použit

4-bit: 0101 $\rightarrow$ 8bit:

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

$\rightarrow$ nemá smysl dělat pro znaménková čísla:

-5 = 1011 $\rightarrow$

0	0	0	0	1	0	1	1
---	---	---	---	---	---	---	---

 $\rightarrow$ vždy to dá kladná čísla

$\Downarrow$

znaménkové rozšíření (signed extension)

$\rightarrow$ vezme MSb a rozkopíruje do zbylých hodnot

MSb
-5: 1011 (4-bit)

1	1	1	1	1	0	1	1
---	---	---	---	---	---	---	---

 = -5 (8-bit)

signed $\rightarrow$ unsigned
unsigned $\rightarrow$ signed
signed $\rightarrow$ signed } kdy použít

Převody mezi soustavami:

literály

0x123 (hex)

0o123 (oct)

0b101 (bin)

převody: int(str, base)

hex(num)

bin(num)

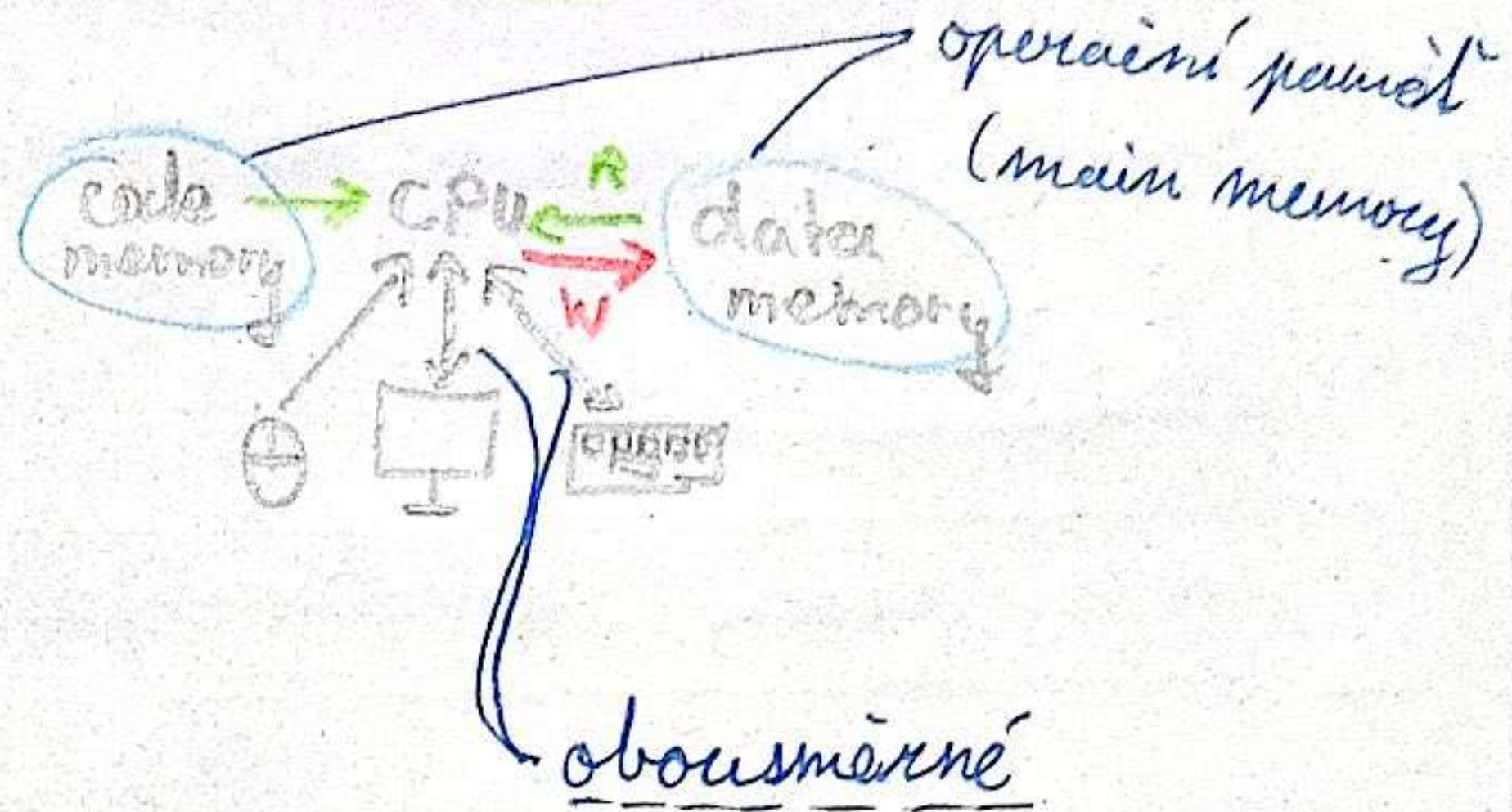
oct(num)

5. MODINA

RS-232

↳ double sériová (full duplex)

↳ obousměrná komunikace je častá



point-to-point

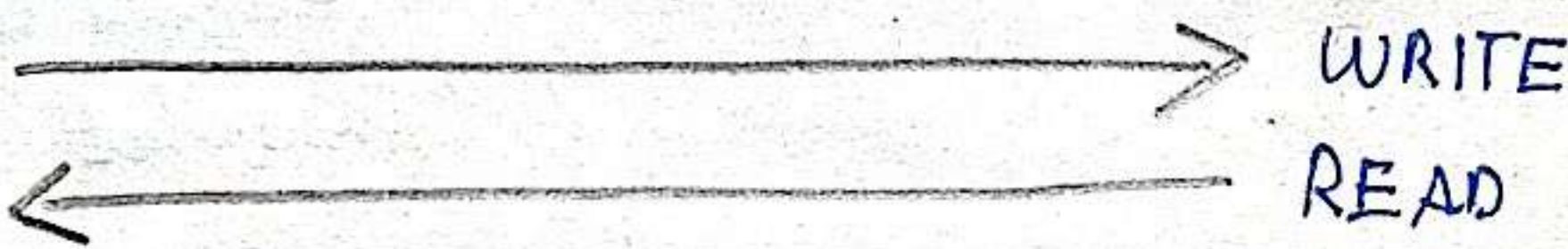
- ↳ mezi dvěma body
- master → kněli kterému se informace přenášejí („řídící“)
- slave → poskytuje služby pro mastera („řízený“)

D1 (device 1)

master

D2 (device 2)

slave



} závislé na tom, kdo je master, protože master se rozhodne, kdy proběhne přenos

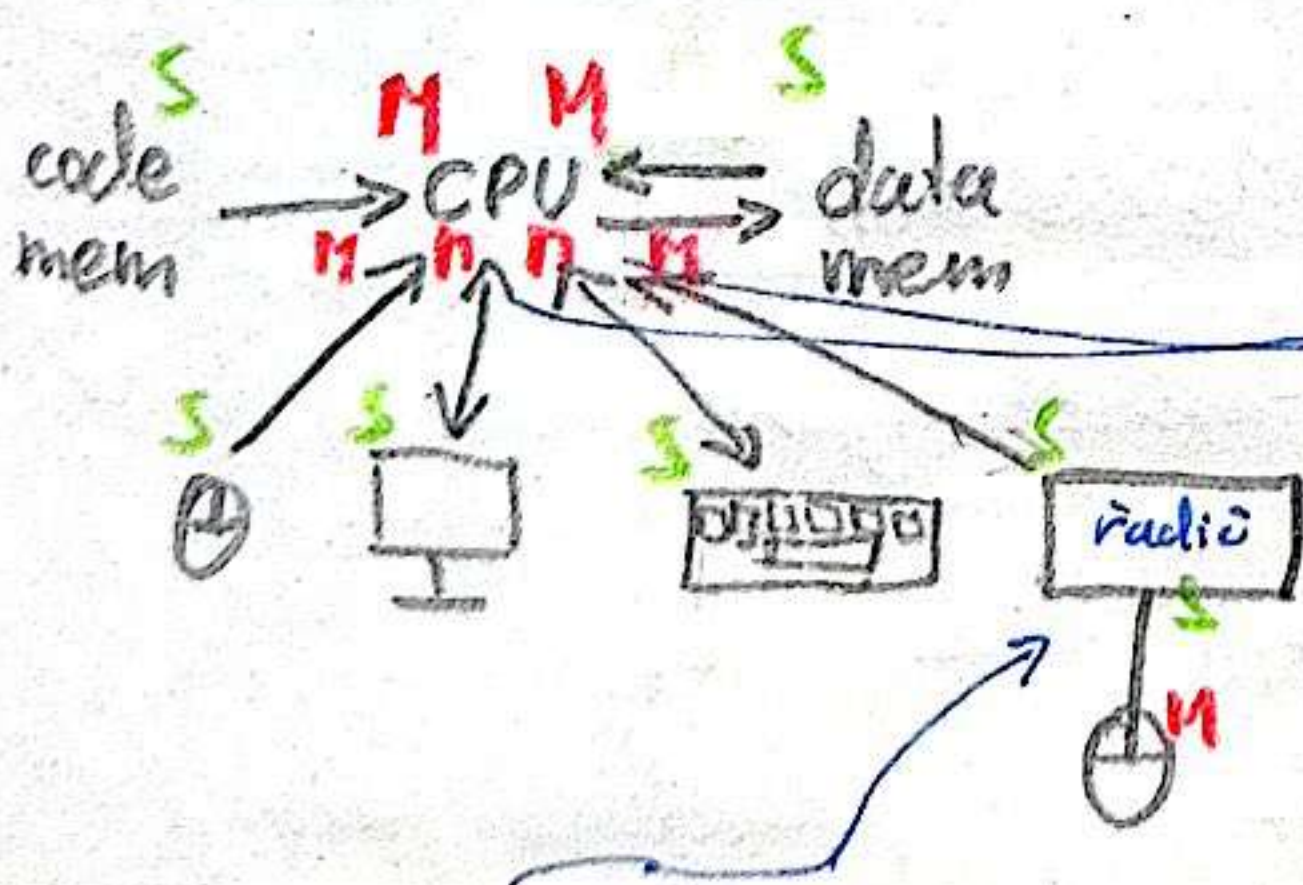
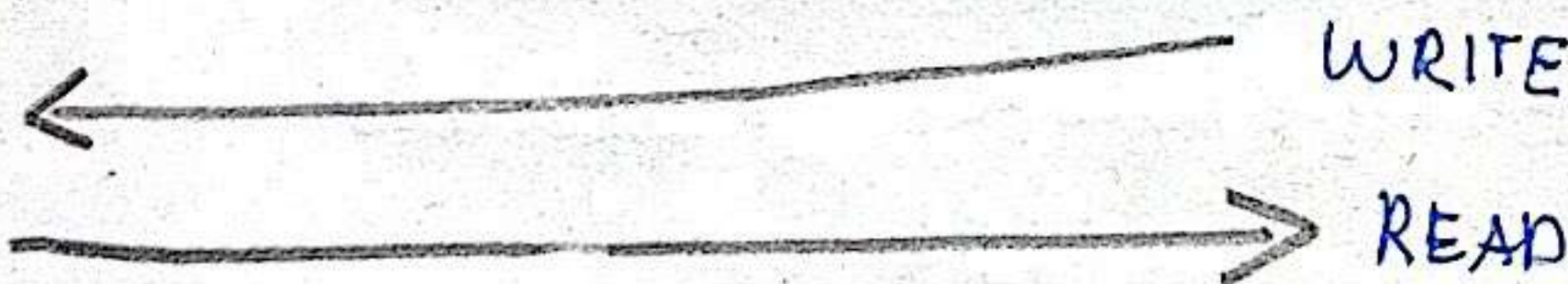
změní roli:

D1

slave

D2

master



master je ladic proto, že se CPU počítá s tím, že periférie, která na něm nemá svůj vstup nebo ne

protože myš poslala pakety sama o sobě a nemůže fungovat jako slave, proto potřebuje rádio, který bude slave

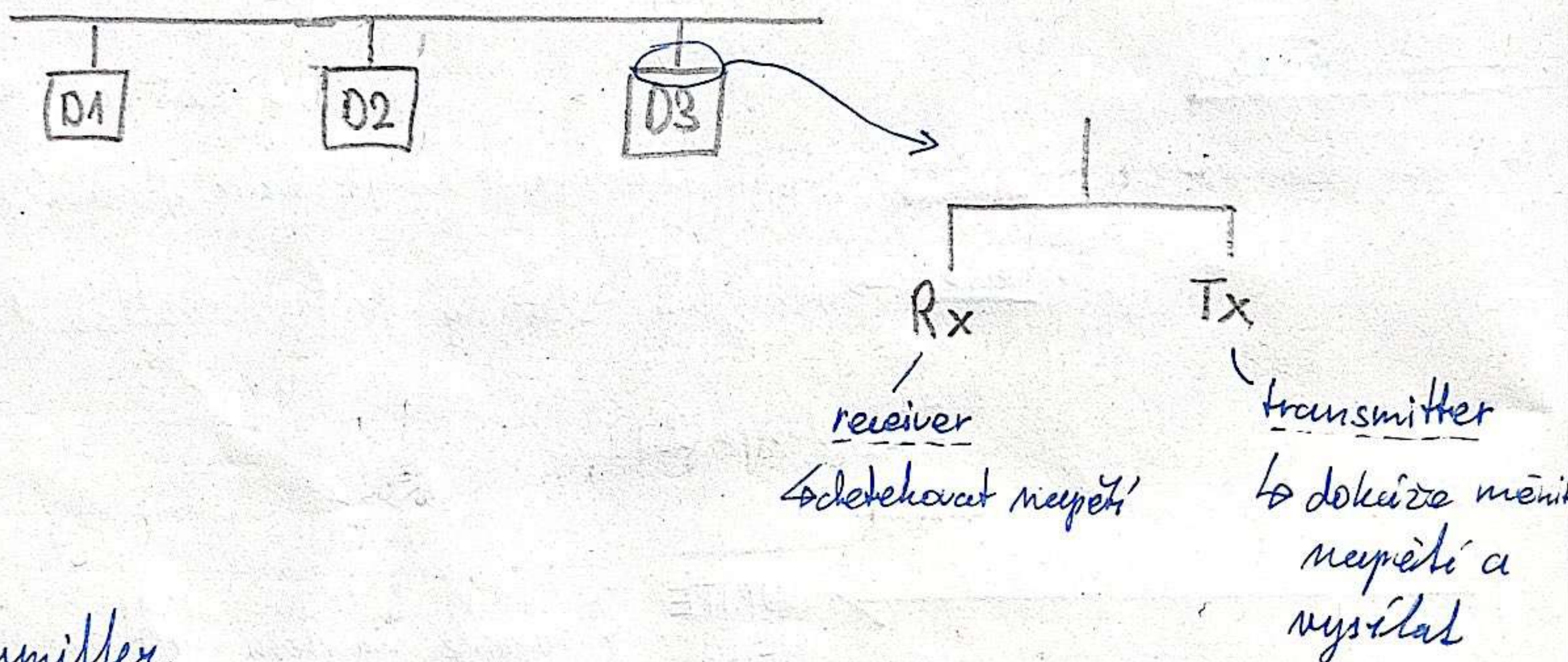
Multidrop / bus / síťnice

↳ jedna linka, co má více zařízení, protože je nevhodné mít tolik spojů na CPU, nebo-li na multidrop lze mít 1 master a více slaveů, an point-to-point → 1 master a 1 slave

↳ half-duplex bus

↳ na jednom datovém vodiči připojeno více zařízení

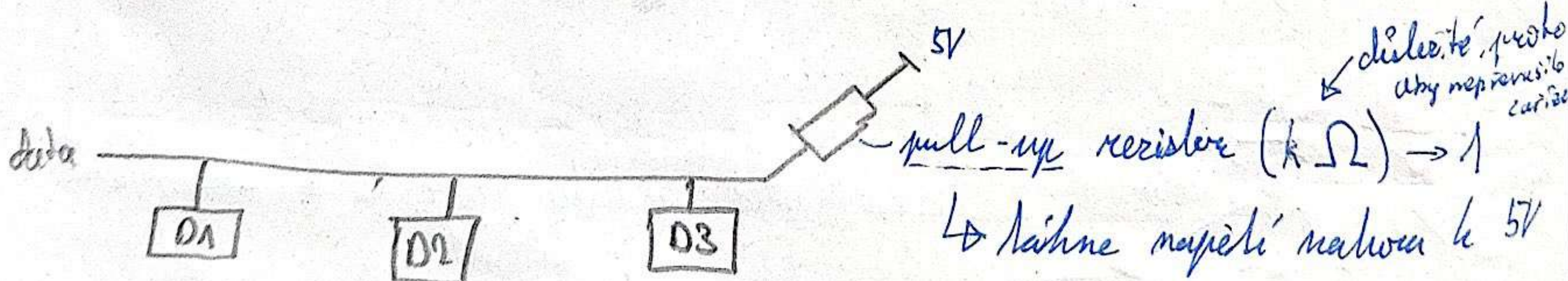
data



Transmitter

nic → floating (odpojit transmitter) → ale je nevhodné, protože je to nelineární a něco mezi 1 a 0. Řešení chceme, aby bus podporoval méně než 2 zařízení, a proto by se nacházel ve floating stavu na začátku, což by se špatně jednalo

1 → 5V
0 → 0V



pull-down ($k\Omega$) → 0

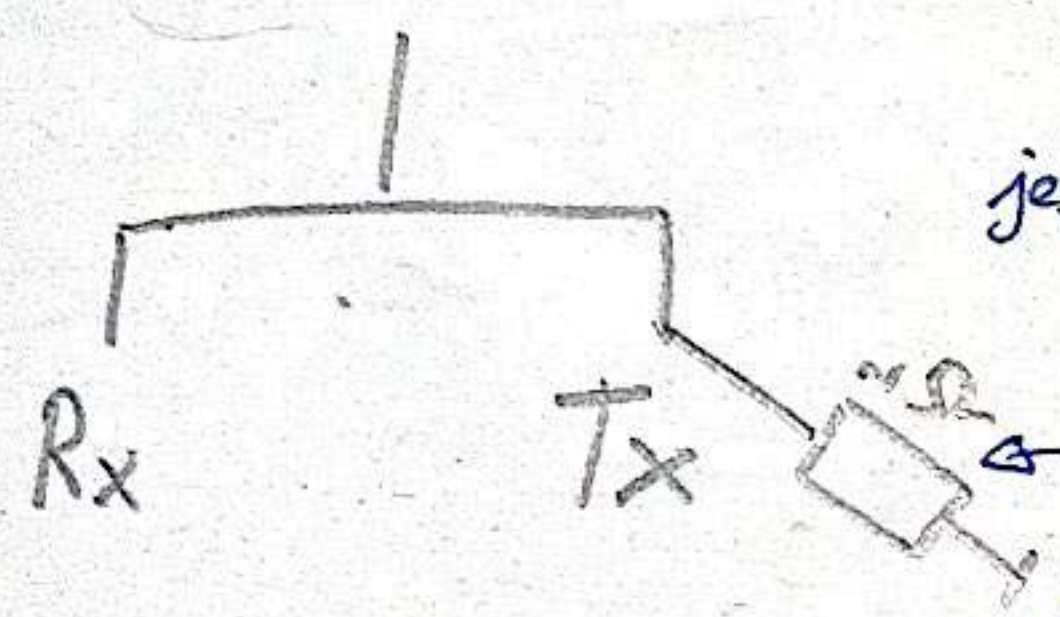
↳ dáme k 0V, pokud se připojí
šlabe nižší napětí

mikro
 nic → odpojit
 1 → odpojit
 0 → připojit

bus	D1	D2	bus
1	X	1	1
1	X	0	0
1	X	X	1
∅	1	1	1
	1	0	0
	0	1	0
	0	0	0

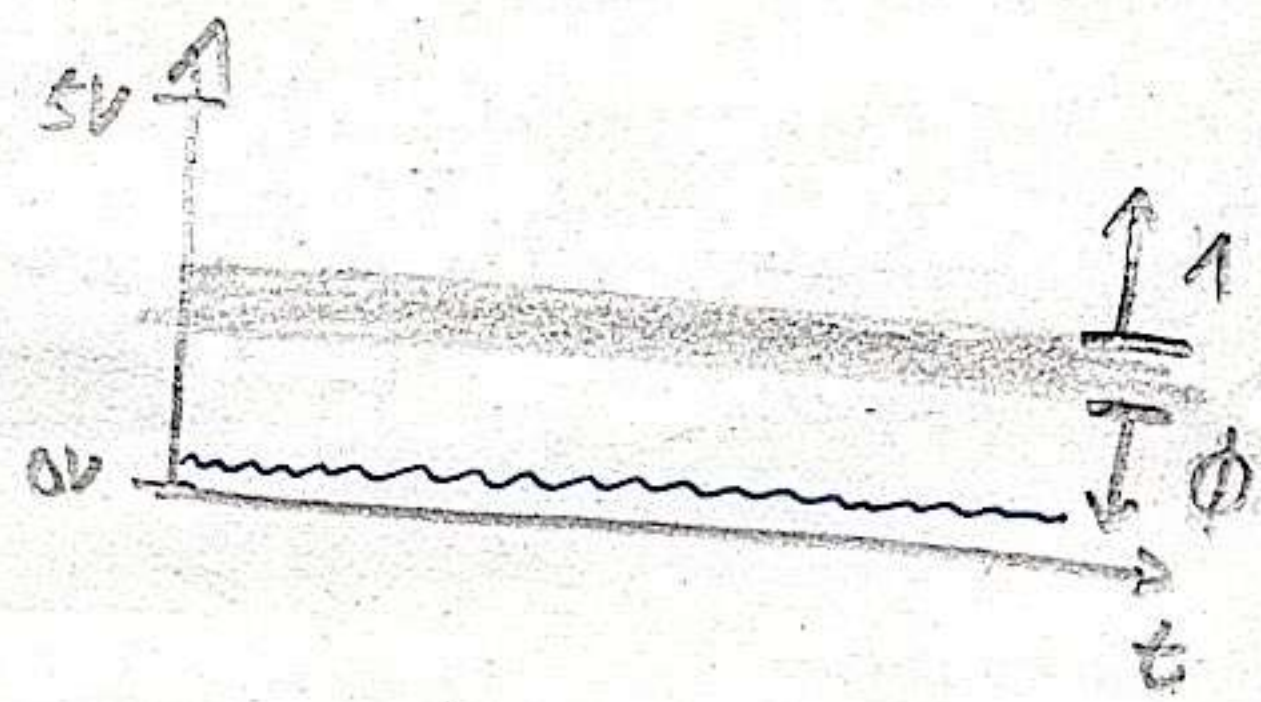
není tam

← obě revize
 něco vysílají
 elektricky neimplementováno
 bitový and (porovnání)
 → vždy definovaný stav

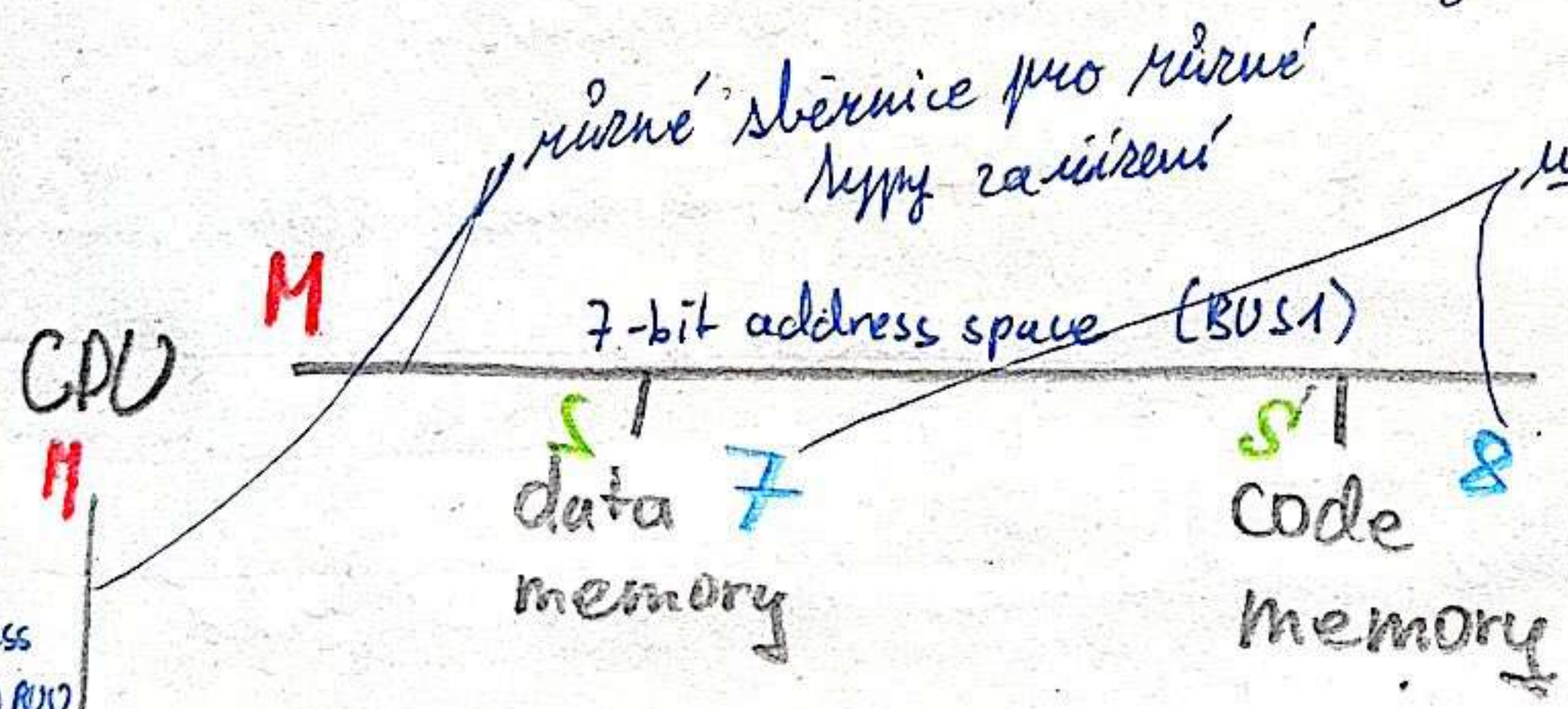


ještě menší napětí

→ menší odpor ne pull up resistor, a proto se na časovém diagramu zobrazí šero mula



bitové operace → rychle, protože elektricky implementovatelné



míně síťnice pro různé typy zařízení

unikátní adresy v rámci síťnice

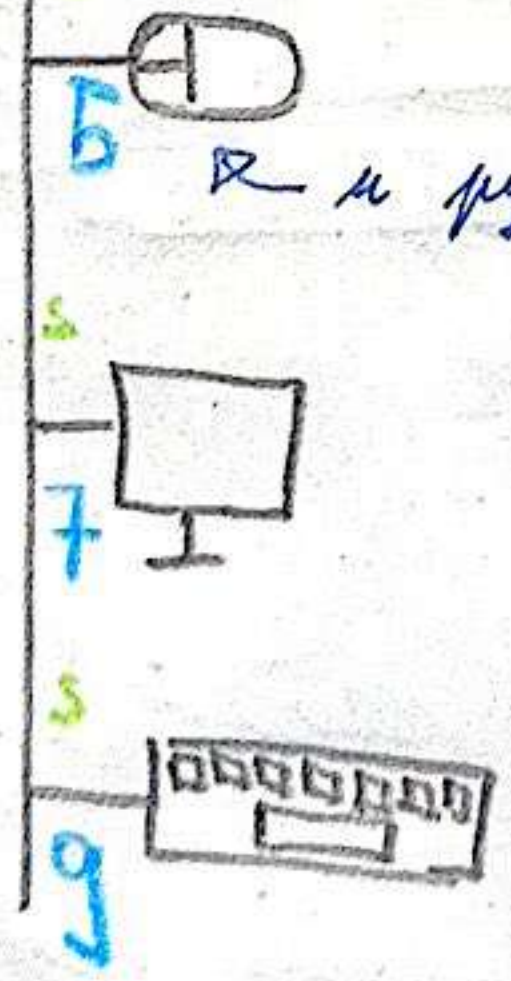
adresa (adresy)

↳ n-bit unsigned int
 ↳ $0 \rightarrow (2^n - 1) = \text{adreseový prostor}$
 = address space

↳ má 7 bit → 0 - 127

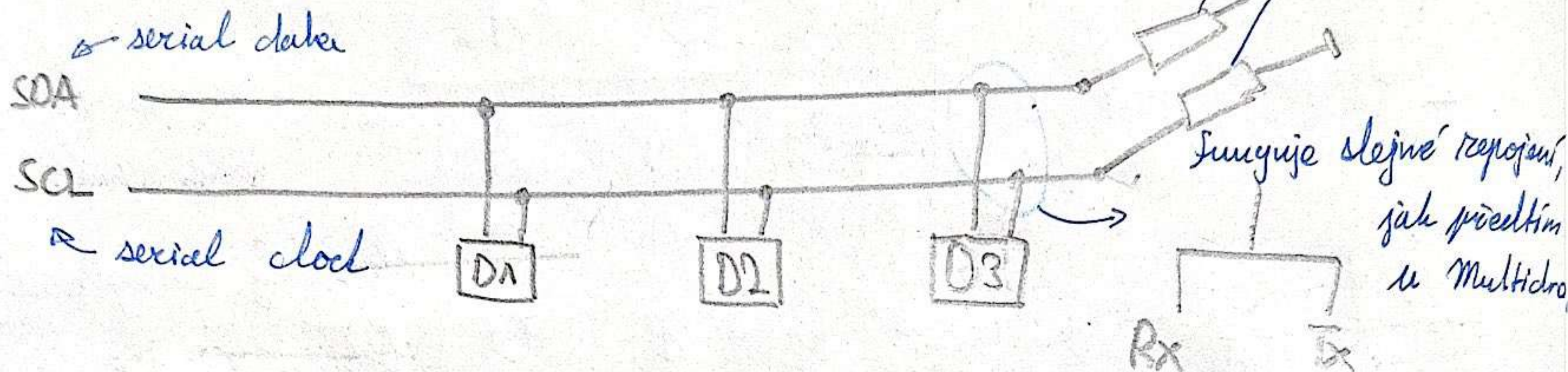
→ přes radio, ale jen u RS-232, což ale běžně přes full-duplex

→ u serial bylo „COM 3“, což byl identifikátor síťnice



I²C (Inter Integrated Circuit) → jehle 2^o I a 1^o C

- ↳ běžně vede mezi zařízeními monitř počítače na místo one
- ↳ běžně používána v telefonech, se atel.
- ↳ převádí hodinový signál na rozdíl od RS-232



V_{DD} → napájecí napětí v elektrických obvodech

V_{CC} → velký napájecí napětí



↳ pomocí clock → pokud je tam 1 → datový signál je správně a neměl by se měnit

0 → datový signál se může měnit, takže by se neměl číst

↳ bude tam série přenosů, které mohou být libovolně dlouhé

→ probíhá mezi 2 zařízeními, kde jedno je slave a druhé master



IDLE

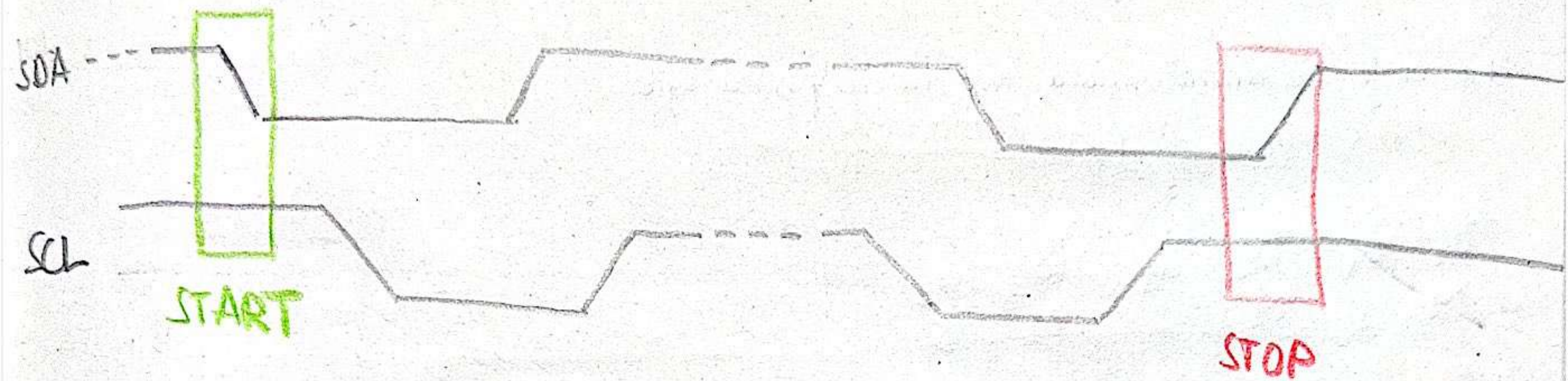
↳ nikdy nic nepřijímá, nemá připojené zařízení, pull up rezistor způsobí že na lince bude 1 v datovém vodiči a 1 v hodinovém vodiči



Má první délka přenosu → je potřeba určit START a STOP condition pro přenos

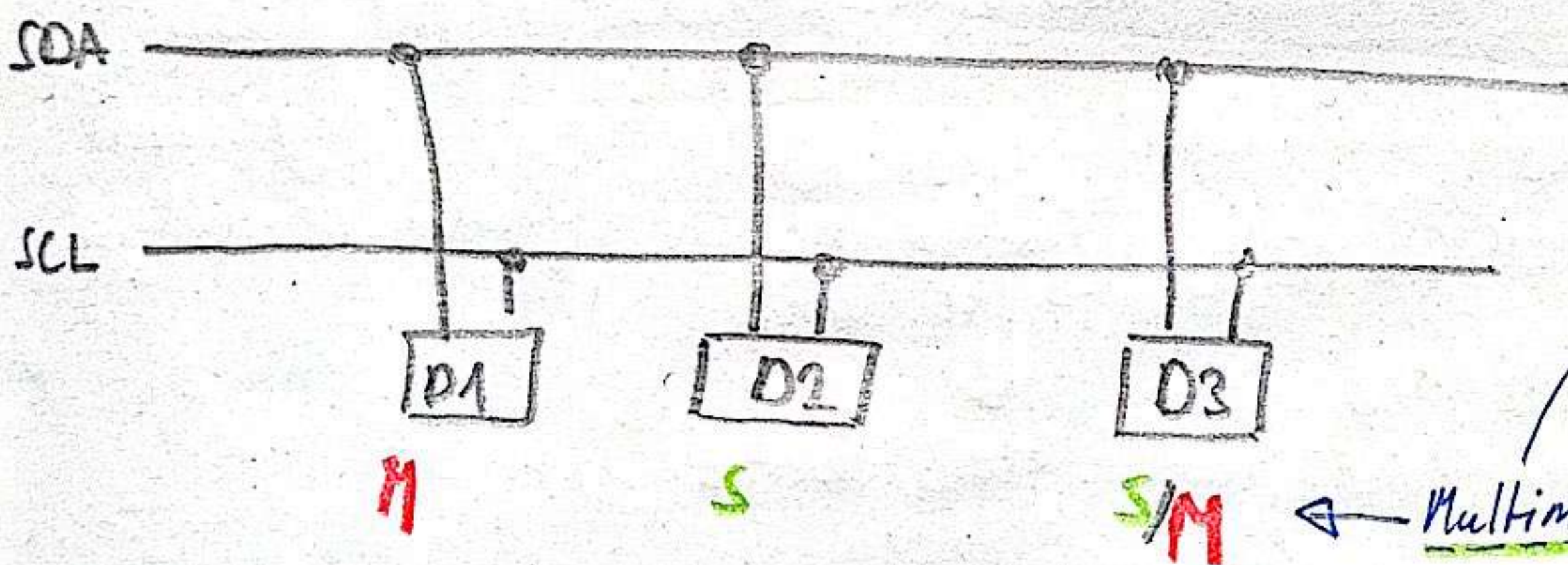
START a STOP condition

→ je to ten speciální případ, co by neměl nastat



↑
IDLE: 1, 1

↑
IDLE: 1, 1



→ občas bývá arbitr,
zařízení, co určuje, kdo
arbitrace služnice komunikuje

→ zařízení se nějak
dohodnou, kdo bude kdy
komunikovat

← Multimaster

→ možnost mít roli mastera i slavea

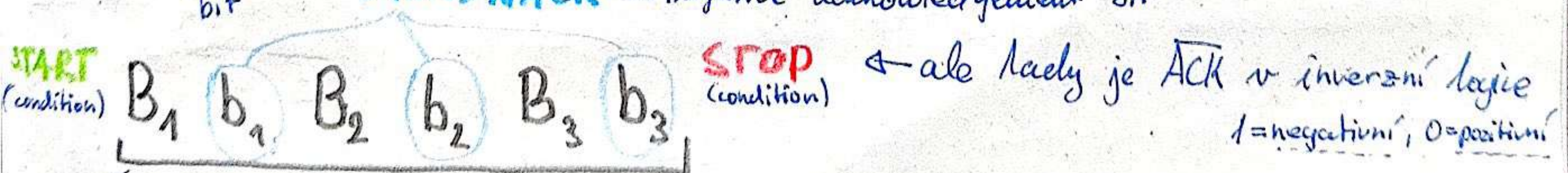
X
Singlemaster

→ např. USB, CPU je jedinec master



Vzezření přenosu

→ posílají se 9-bitové byty: 8 datových bitů a 1 řídicí bit, MSb-first pořadí
acknowledgment bit → ACK/NACK → negative acknowledgement bit



→ paket I²C služnice, jak musí vypadat → tzv. komunikační protokol (na další stránce)

kommunikční protokol → vzhled paketu



1B 0-n B

→ payload

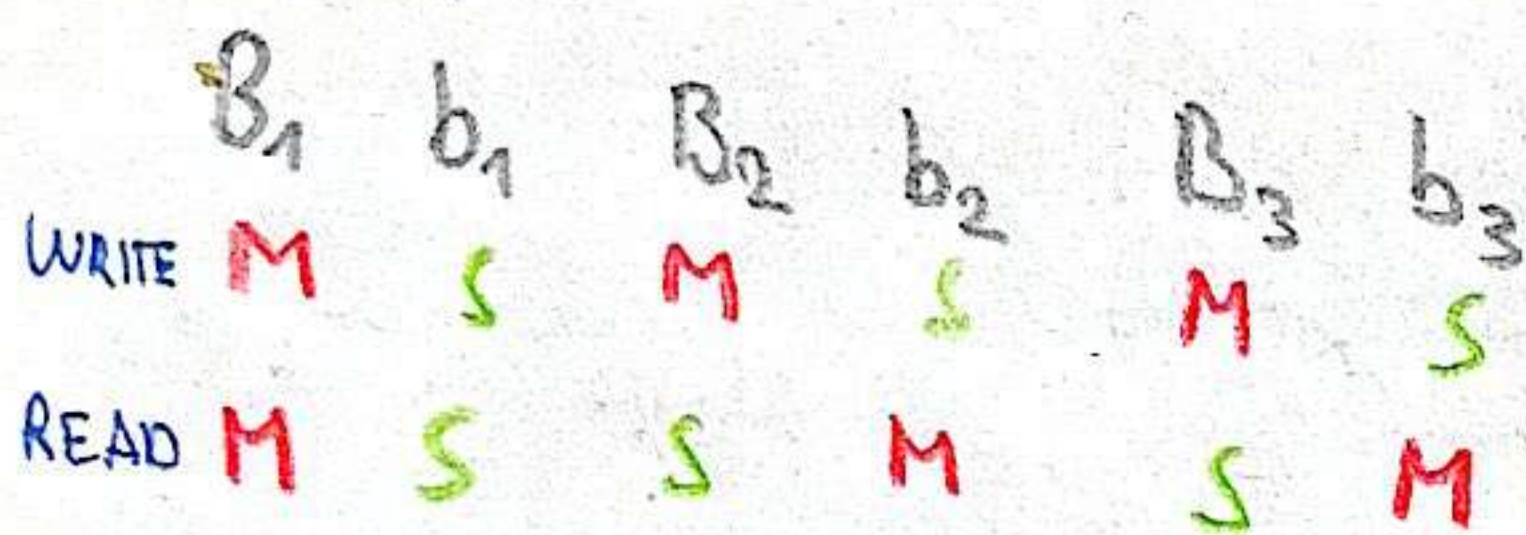
↳ to užitelné při přenosu

overhead komunikačního protokolu

↳ control byte vždy posílá master a ACK mu něj posílá slave

ACK

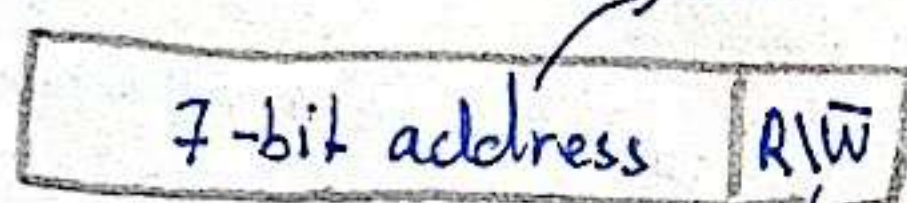
↳ generuje ta druhá strana než ta, co poslala byte



← payload proudí od slave k masterovi

Control byte → dostanou všechna zařízení

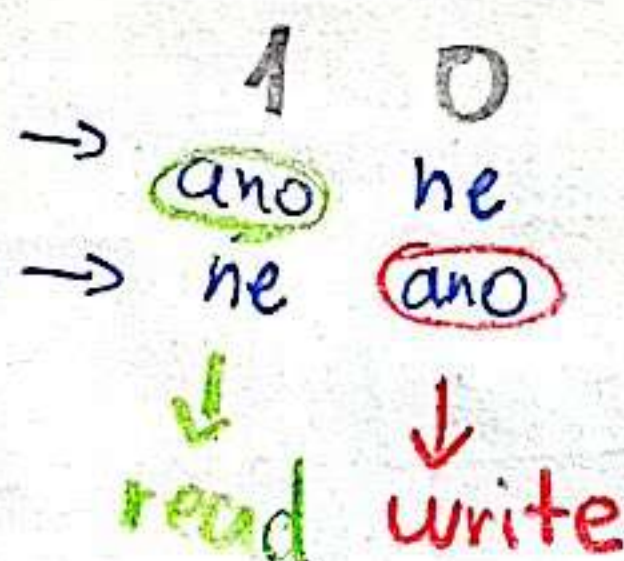
t.č. 7-bitový adresový prostor (v MSb-first order, ale jen v té adrese)



↳ zařízení s jinou adresou budou zbytek paketu ignorovat

→ READ

WRITE (inverzní logika)



az ma STOP condition

ACK → potvrdil přijetí a pokračovat v přenosu

NAK → pokud slave/master chce přestat přenos

ACK → v inverzní logice protože pokud začne zařízení nemí připraveno, tak defaultně je na úrovni 1 kvůli pull up rezistorům

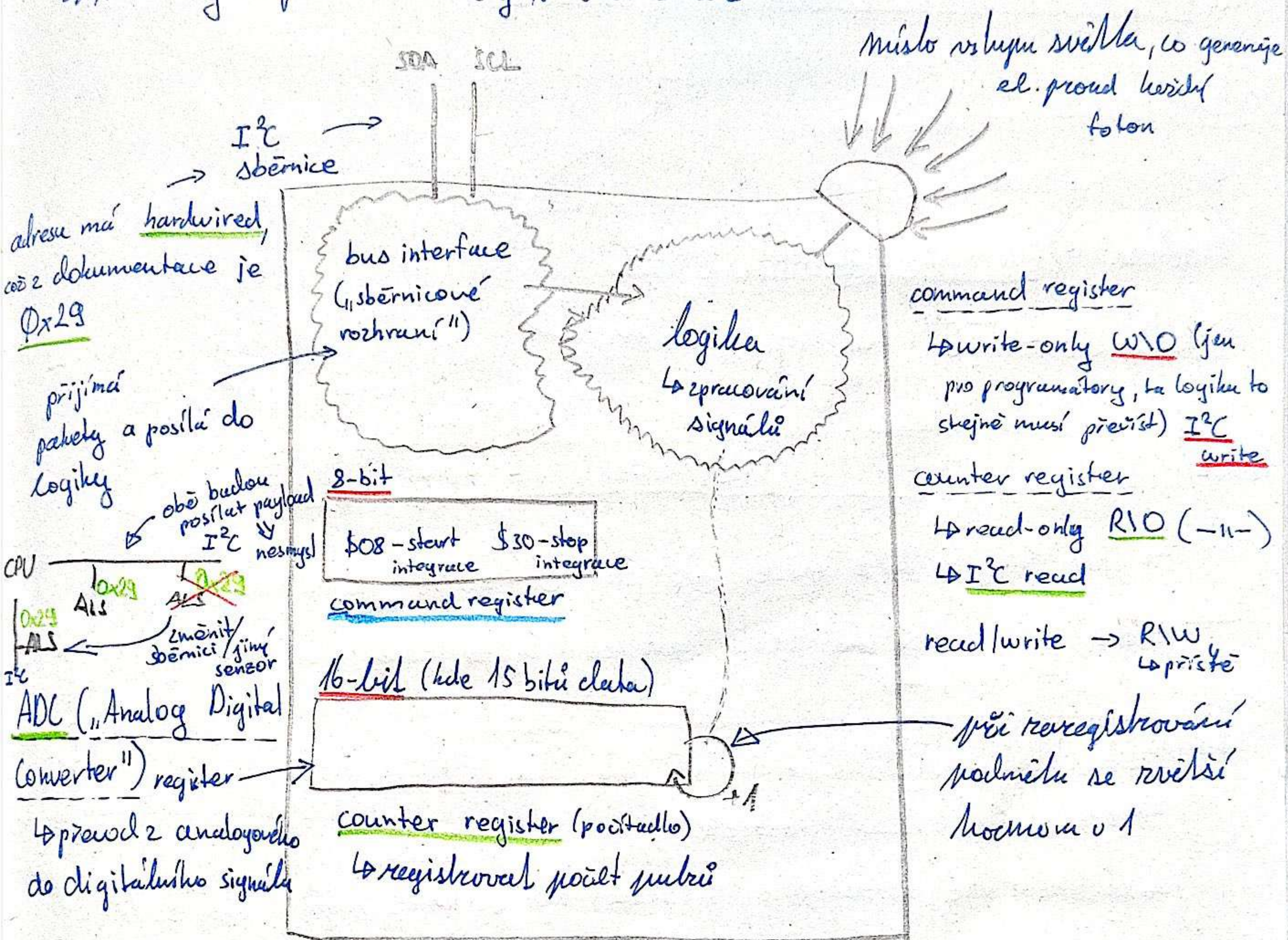
→ hodinový signál během přenosu generuje master a tím určuje rychlost přenosu, slave může nastihnout pokud nestíhá (má hold low) → poště kam = clock stretching, 0 do SCL

↳ v IDLE stavu je pasivně 5V, takže žádný hodinový signál, jen ječka

→ 100 kHz — 5 MHz (povolené rychlosti u jednotlivých zařízení), např. USB má až GHz

Ambient Light Sensor (ALS) → příklad navázání na I²C sběrnici

- ↳ sleduje intenzitu okolního světla
- ↳ všechny informace vždy v datasheet



místo vstupní světla, co generuje el. proud kerich foton

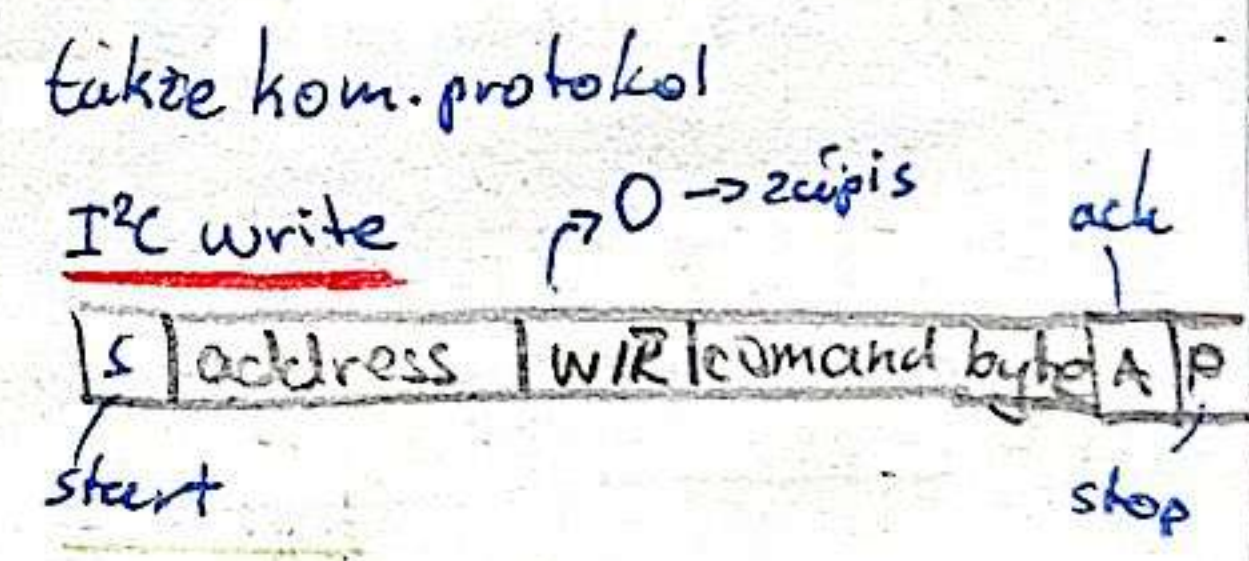
command register
 ↳ write-only W10 (jen pro programátory, ta logiku to stejně musí převést) I²C write

counter register
 ↳ read-only R10 (-11-)
 ↳ I²C read

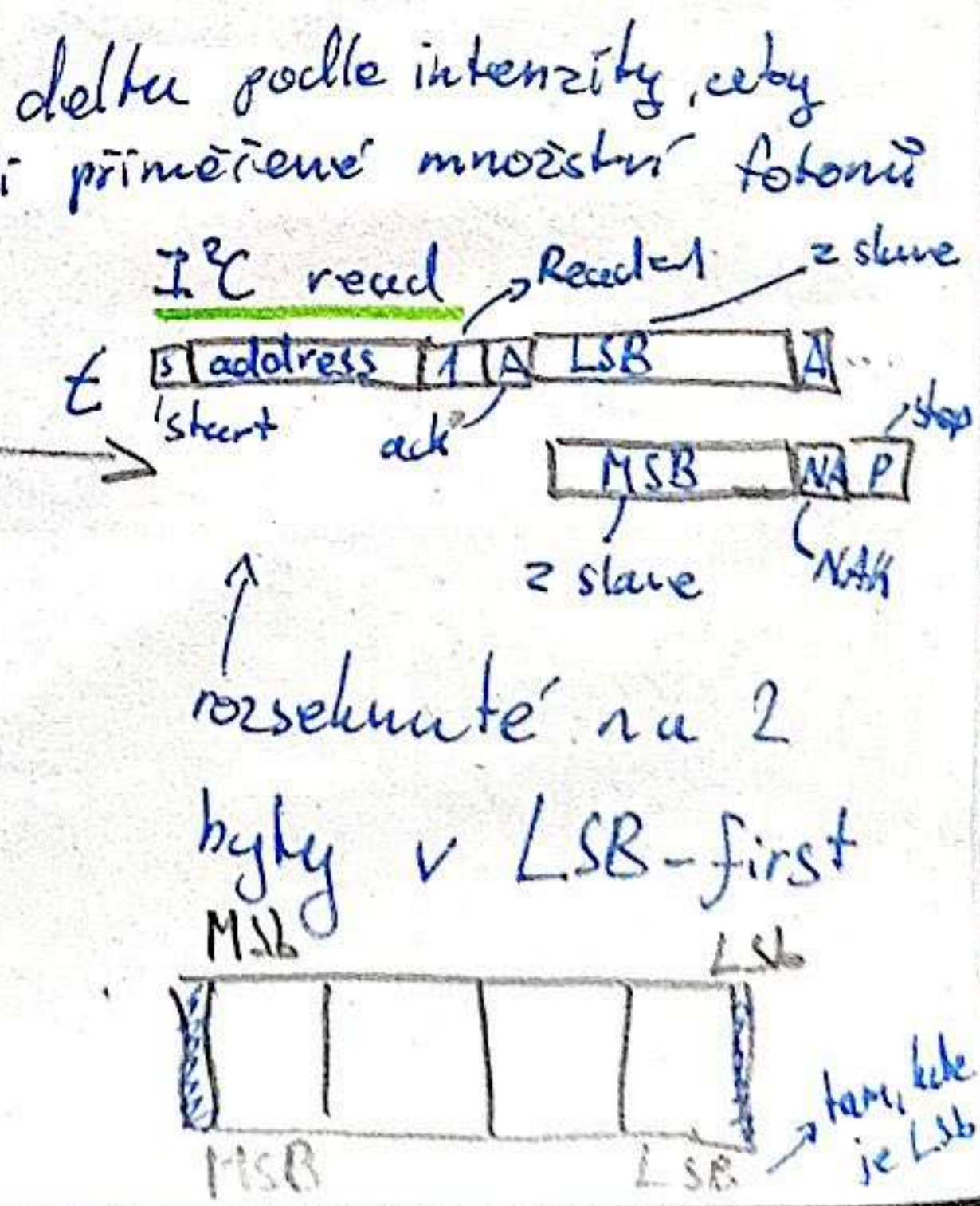
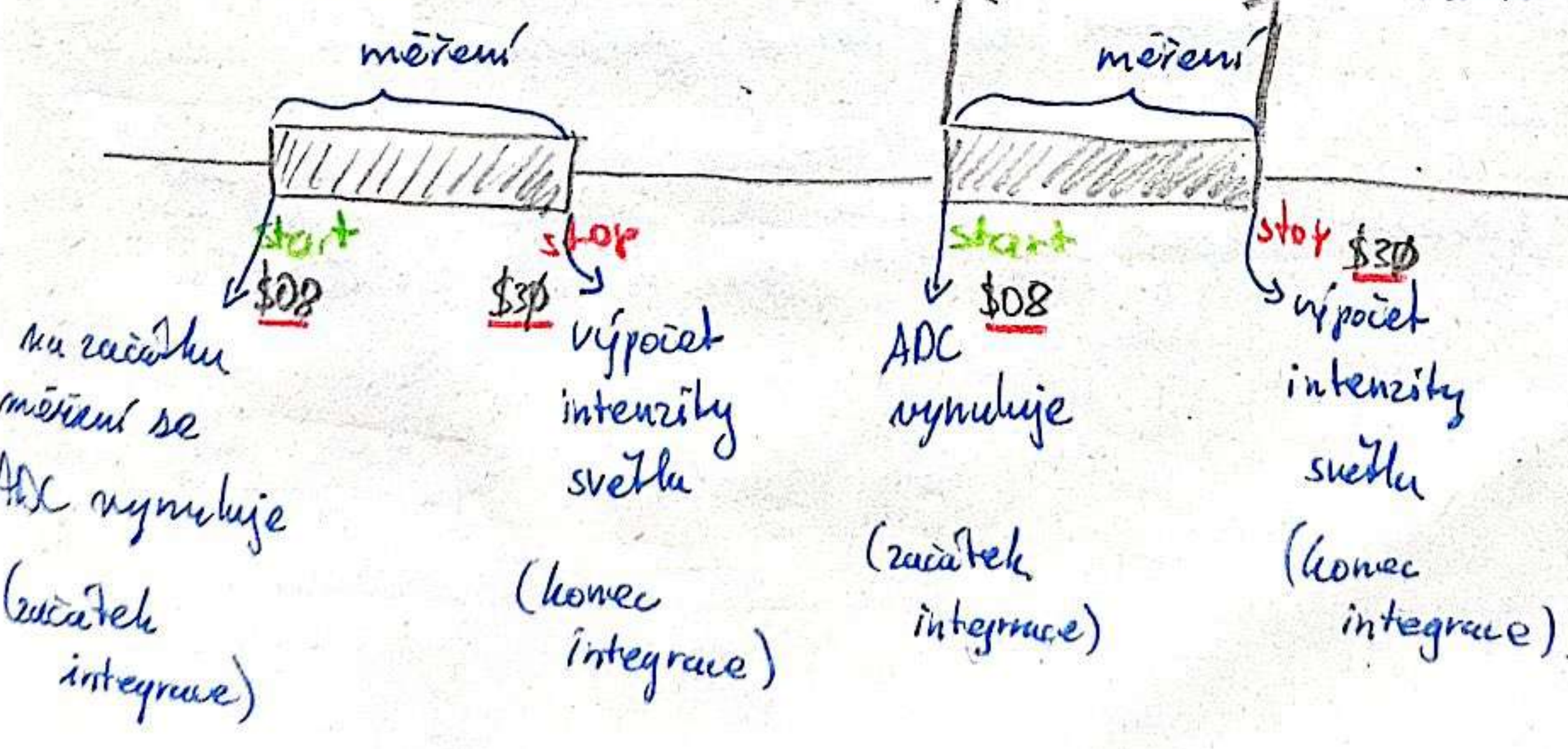
read/write → R1W
 ↳ přístě

při zaregistrování podniku se zvětší hodnota o 1

- ↳ zápis kvanta fotonů ve dvojkové soustavě
- ↳ 15 bitů data a 1 valid bit

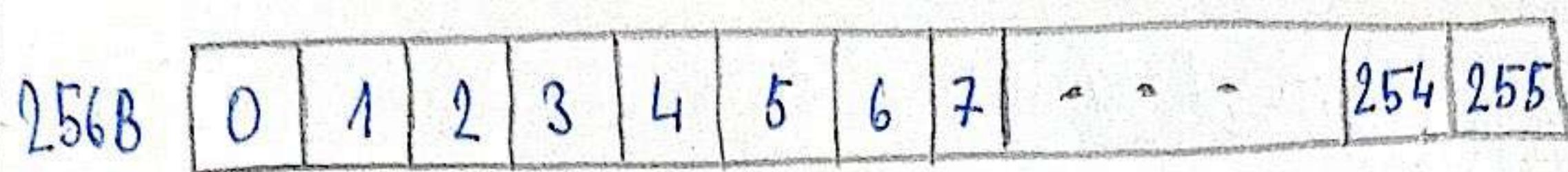


Příběh měření



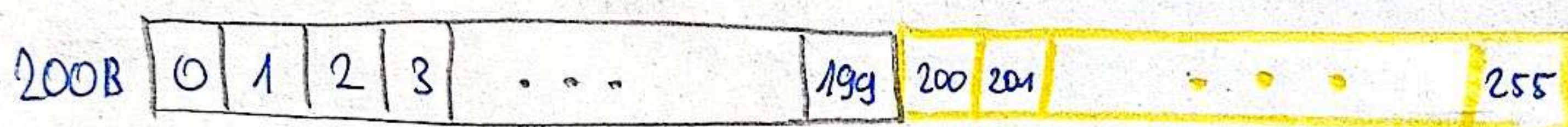
6. HODINA

RAMET



adresa bytu

- ↳ unsigned int
- ↳ adresa v memory address space
- ↳ 8-bit: 0 — 255 (adresový priestor)



- ↳ zase 8-bit address space, ale jen 0 — 199 je validních adres, ostatní se nulují při write nebo vrátí 0 při read (od výrobce)



- ↳ 7-bitový adresový priestor

- ↳ u 8-bitového adresového priestoru by polovina byla nevyužitá, ale zas má přenos 1 Byte (8-bitů) je pohodlnější



- ↳ použít 16-bitový adresový priestor, většina nevyužitá

- ↳ "prozíravost" → nepř. u programu, kdy máme proměnné 8-bit, tak 256³ zbytečně, ale při zvětšení adresového priestoru na 16-bit by bylo potřeba přepsat program na 16-bit proměnné

1kB = 1024 B = 1 KiB („kibi“) → kilobajt binární

↳ 10-bit address space, používá se 16-bit, což je 64 KiB, díky $2^6 = 64$ ^{desítek bitů}

1MB = 1024 kB = 1 MiB („mebi“)

↳ 20-bit pro adresování (10-bitů = kB a 10-bitů pro 1024), používá se 24-bit, 16 MiB

1GB = 1024 MB = 1 GiB („gibi“)

↳ 30-bit, používá se 32-bit, 4 GiB

1TB = 1024 GB = 1 TiB („tibi“)

↳ 40-bit, používá se 40-bit, 1 TiB

1PB = 1024 TB = 1 PiB („pibi“)

↳ „petabyte“, 50-bit, používá se 56-bit, 64 PiB

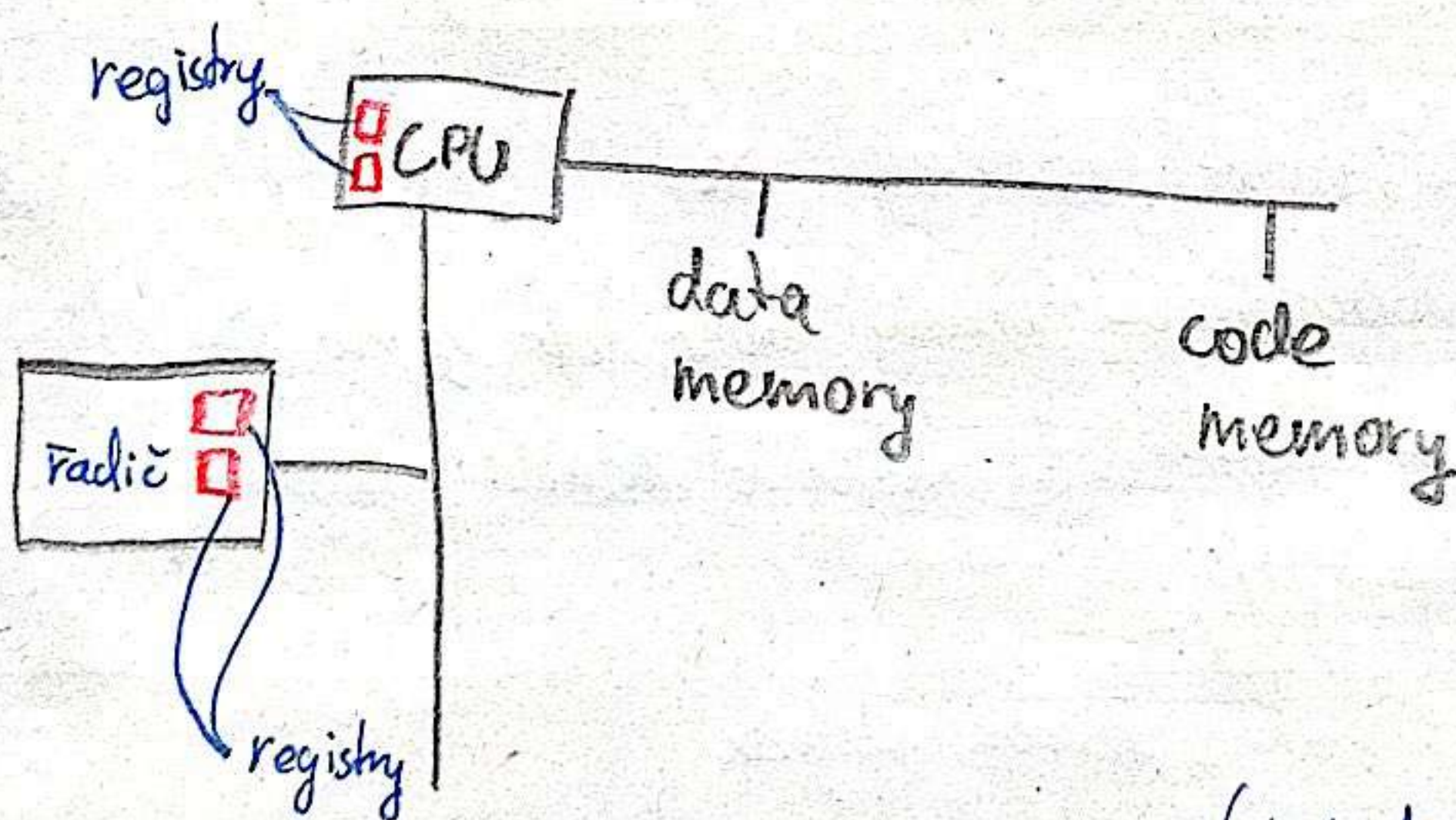
1EB = 1024 PB = 1 EiB („eibi“)

↳ „exabyte“, používá se 64-bit, 16 EiB

↳ 60-bit

↑ výroba disků

↳ jiná definice 1kB = 1000 B, aby prodali menší kapacitu, a proto se zavedlo jiné značení pro informatické jednotky 1KiB = 1024 B



(4-6 tranzistorů) na implementaci 1 bitu

8-bit registru je vlastně 8 * 1-bit latchů (je jich potřeba 8)

SRAM (static random access memory)

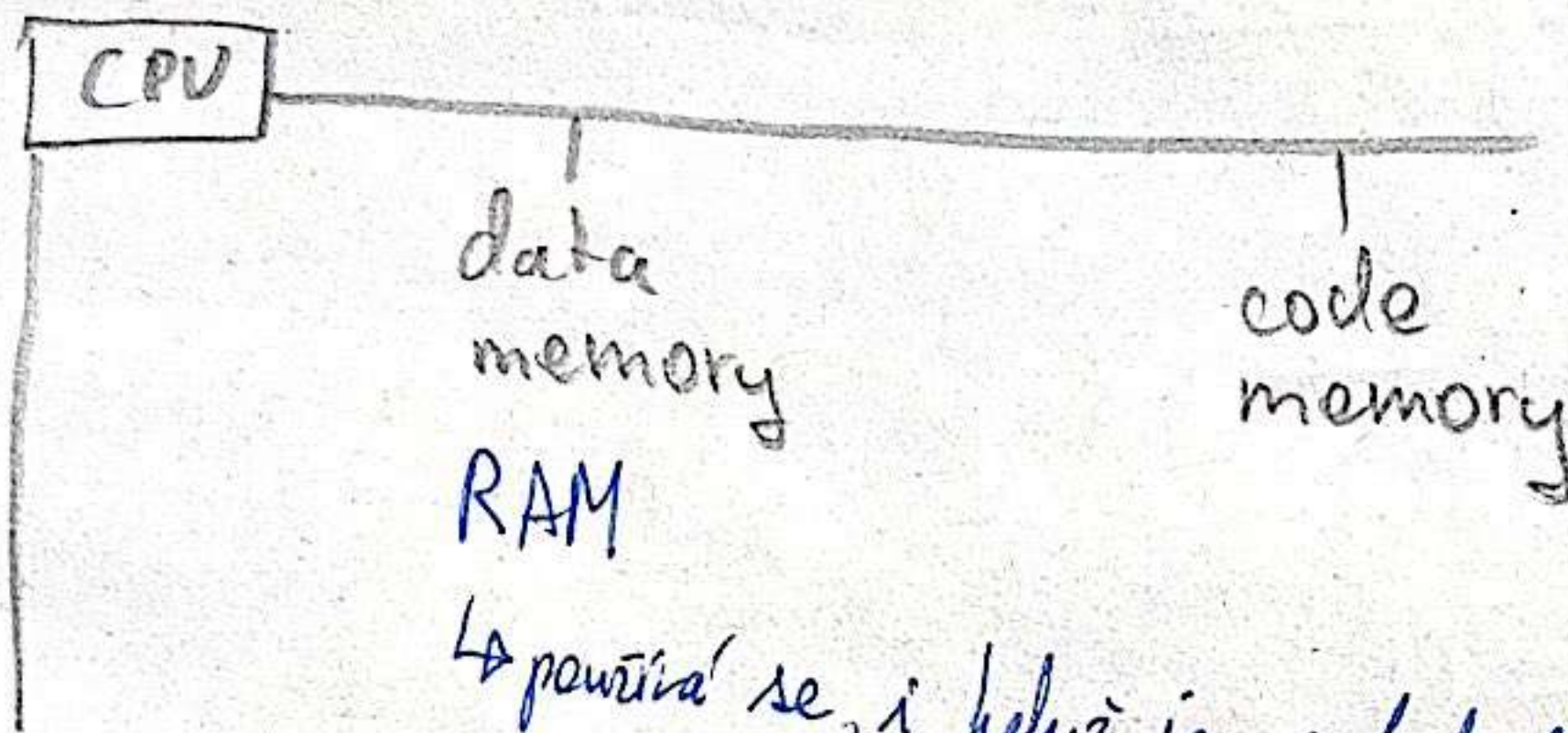
není random access
v algoritmicke platí
↳ v realitě neplatí

1) výběr libovolného bytu

2) uniformní přístup → je jednou, na kterou adresu se budeme ptát, vždy to bude stejné rychle

3) R/W → libovolné množství zápisů a čtení, ale moc to neodlišuje od jiné paměti

4) volatile → paměť zapomené svůj obsah, když vypneme napájení
→ klíčová vlastnost RAM
→ záleží na teplotě (rychlost zapomínání)



→ nelze použít RAM, protože by zapomněla obsah, ale non-volatile paměť

→ poznává se, i když je volatile, protože jsou rychlejší než non-volatile
→ většinou DRAM

6 tranzistorů cca

SRAM = static RAM

→ sekvencí přístupů (nejrychlejší) $\begin{cases} 0, 1, 2, 3, \dots \rightarrow \text{rychlé} \checkmark \\ 5, 4, 3, \dots \rightarrow \text{občas rychle, jindy ne} \sim \end{cases}$

→ random-access → nejhorší možný přístup $\times$
→ i když беру každý druhý byte, tak je to pořád náhodné; musíme shodnout další v pořadí

→ cena je vysoká, protože se tvoří z silikonu učeferu

→ kapacita má B → kB → MB (malá), ale rychlost 10-100 GBps
→ přenosová rychlost

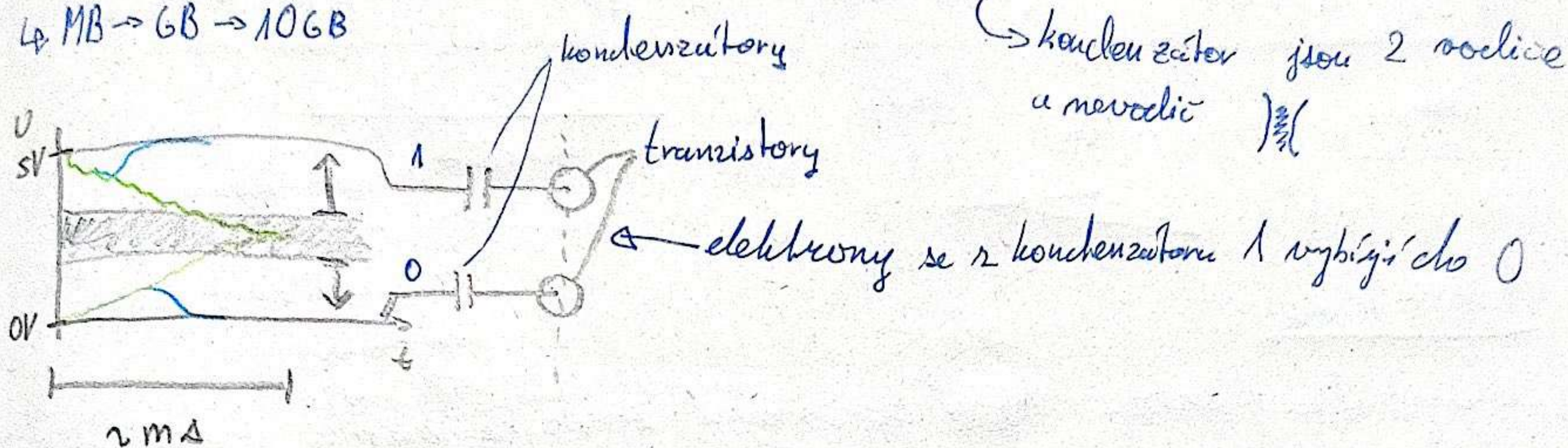
→ ~1 ns

→ přístupová doba

DRAM = dynamic RAM

- ↳ porádí RAM, jen její technologie → furt volatíle a ostatní vlastnosti
- ↳ zapomenou obsah, i když režimem nevypojíme, za cca 1ms
- ↳ interně vyrobená (1 tranzistor + 1 kondenzátor) → menší

↳ MB → GB → 10GB



↳ kondenzátor jsou 2 rovice a nevodí

↳ než neúplně

↳ proto se minimálně 100x za sekundu dělá refresh, nebo-li se se kondenzátor dobíjí do 1 nebo vybíjí do 0 (omezeno močie)

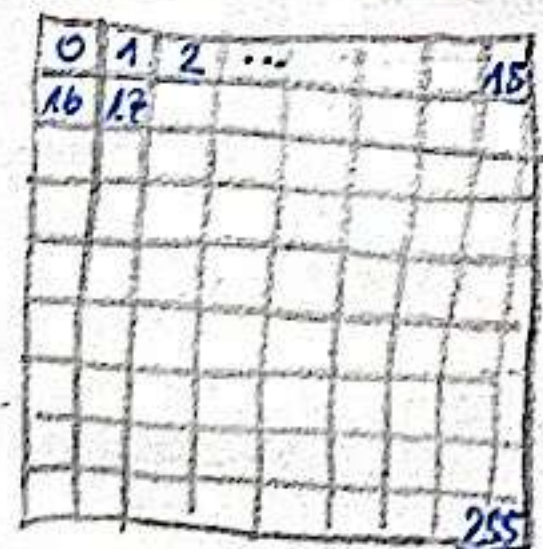
↳ když dělá refresh tak nemůže plnit příkazy, proto jsou pomalejší

řádově 1-10GBps

↳ přenosová rychlost (transfer rate)

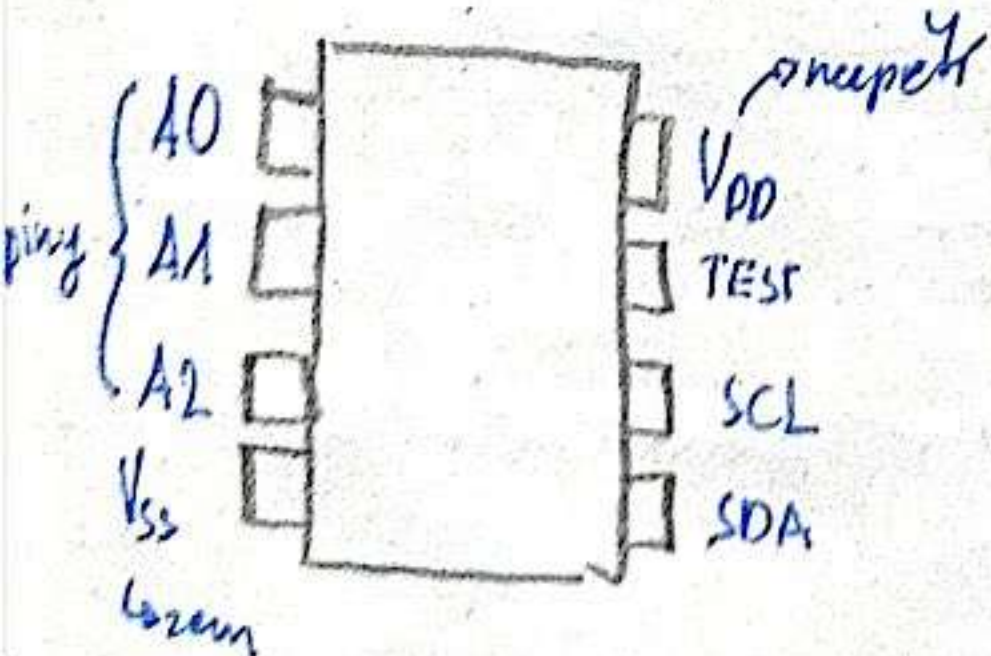
↳ ~10ms

↳ přístupová doba (access time)



→ implementace čipu je většinou v nějaké matici

konfigurační piny



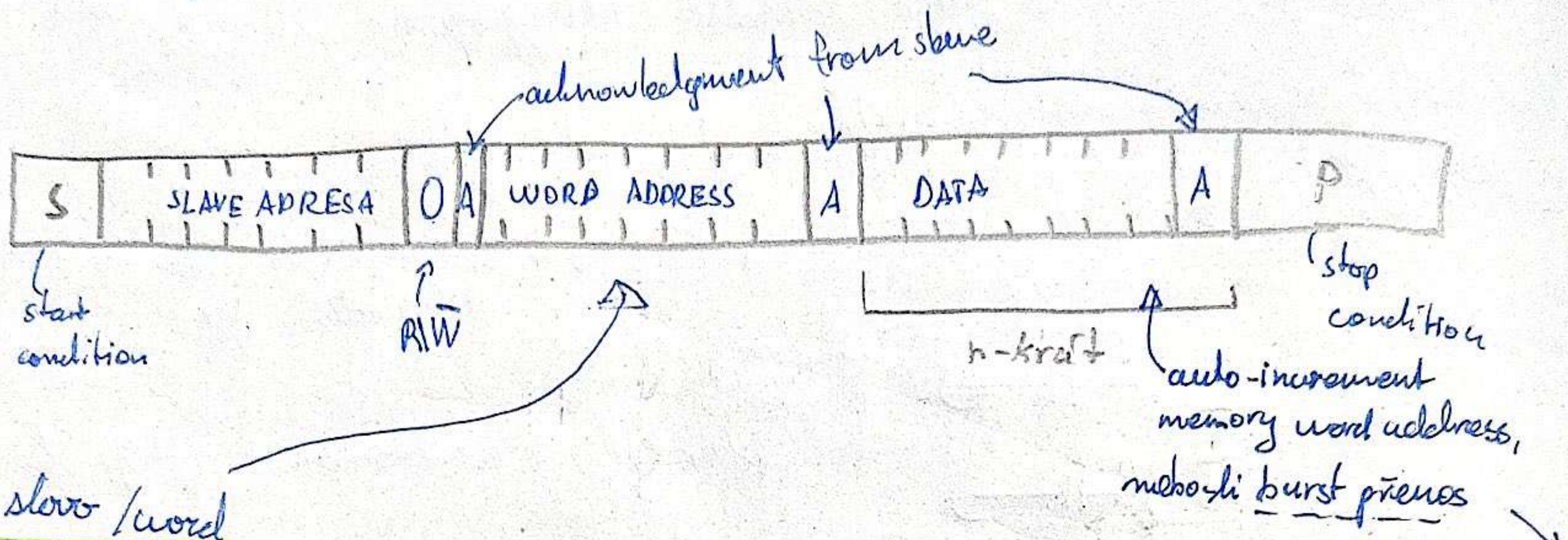
15 volí adresu zařízení, adresu: 1010

A ₂	A ₁	A ₀	hardwired	A ₂	A ₁	A ₀	R1W	
0V	0	0	1	0	1	0	0	1010001 → \$51
0V	0	1	1	0	1	0	0	1010010 → \$52

↳ můžeme rozkódovat napětí

Komunikační protokol s pamětí

→ pořadí je to I²C sběrnice



správně → jednotka přenosu nebo zpracování → to, se jednotka přenosu je n-bitů

→ s jak velkou jednotkou dobře umět pracovat

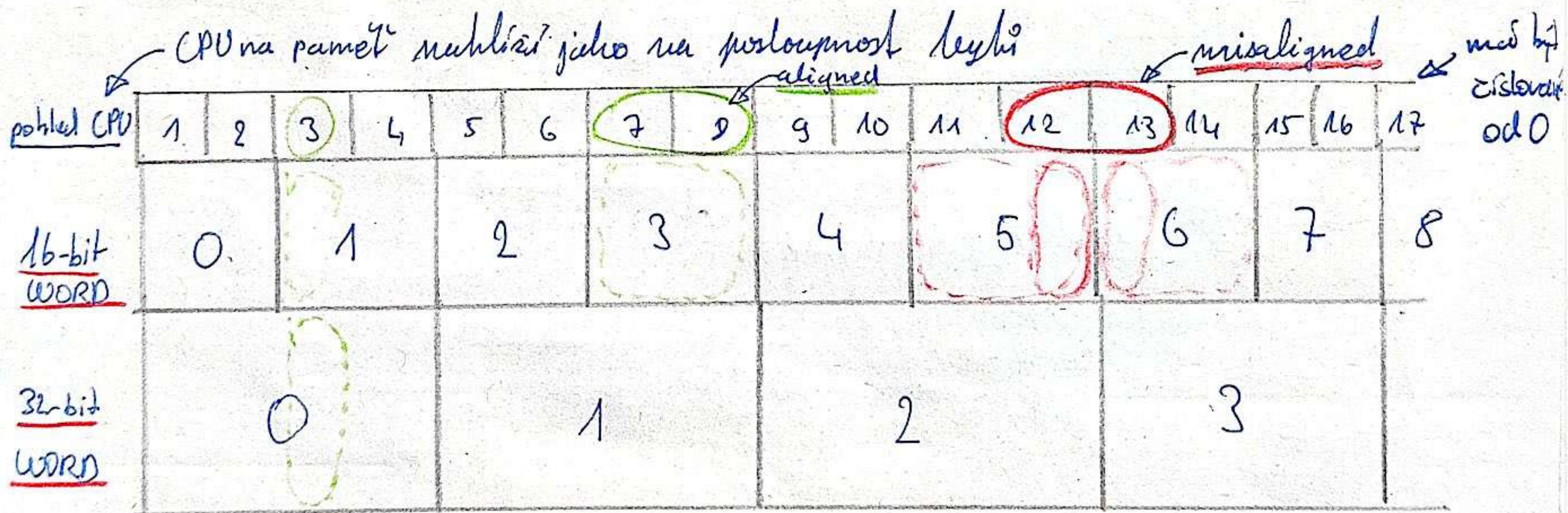
→ např. pro 8b umět se vyřádit podle 8b

špatně → 16-bitů (konstanta), protože když se to ráčelo používat:

→ double word (DWORD/DW) → 32-bit 4B

→ quad word (QWORD/QW) → 64-bit 8B

} správně: násobek slova
špatně: konstanta



→ velikost slova, se kterou pracuje paměť

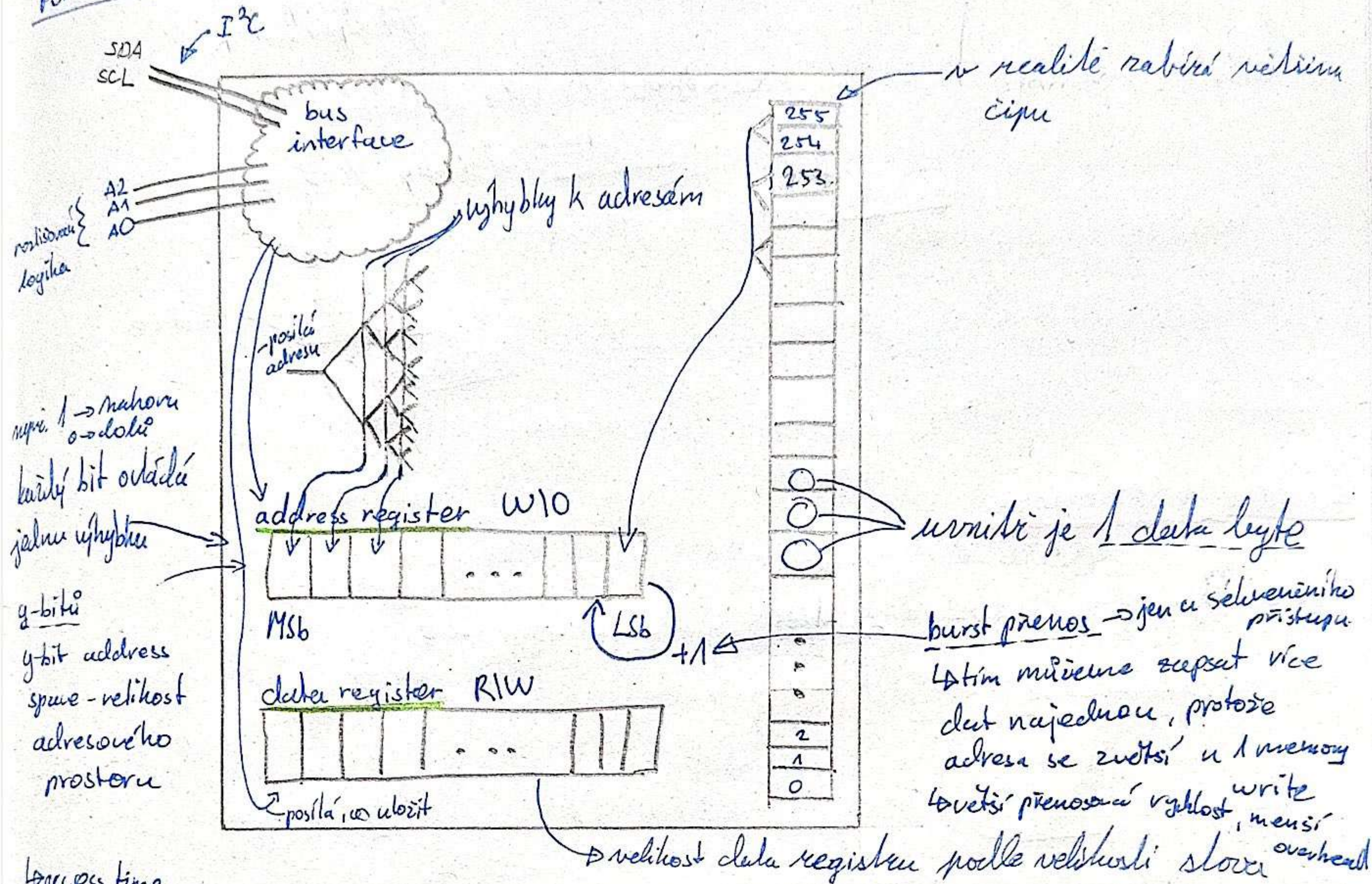
→ např. pro byte 3 si CPU musí vzít celé 16-bit WORD a vrátí si jen polovinu, pro 32-bit obdobně

→ vždy musí přenést celé slovo → musí se adresovat slovo

→ je to efektivnější, protože pomocí menší adresy přeneseme větší slovo, takže čím větší slovo, tím větší přenosová rychlost, ale když nemůžeme celé slovo, tak většinou přeneseme rychleji

- ↳ můžeme začít rovnou 2 byty, ale existují 2 případy:
 - aligned → má 2 byty, 1 slovo, což je rychlejší
 - misaligned → musí načíst zbytečně více slov
- ↳ kvůli rychlosti je to důležité

PCF8570 paměť



Access time

↳ způsobeno tím, že třeba, než se uhybnou přepnou

↳ burst je rozdíl 1e6

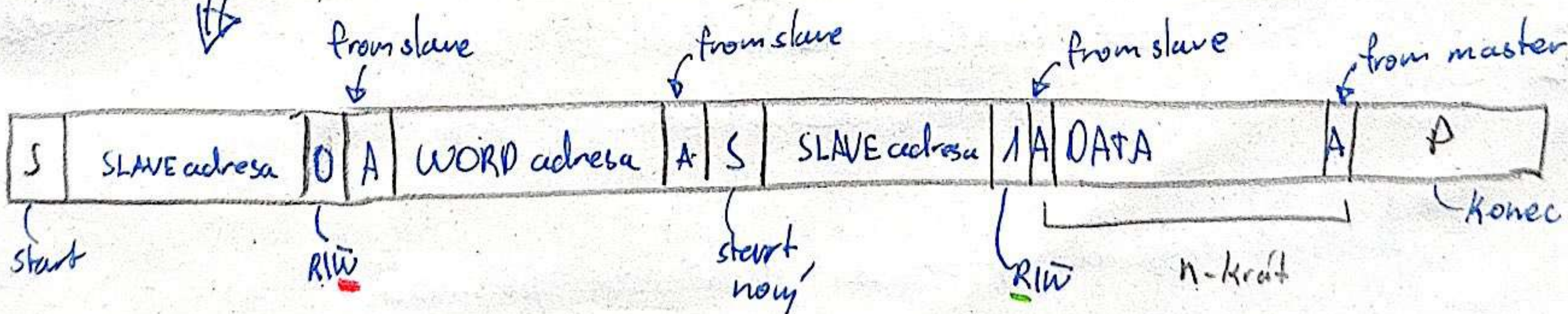
8-bit → 4B (auto-increment)
32-bit → 4B (vždy celé slovo)

↳ to slovo je rychlejší 4x u každé přímce

I²C bus (jen u této paměti, ne obecně)

↳ 1 memory write = 1 I²C transakce → na přechodů stránce

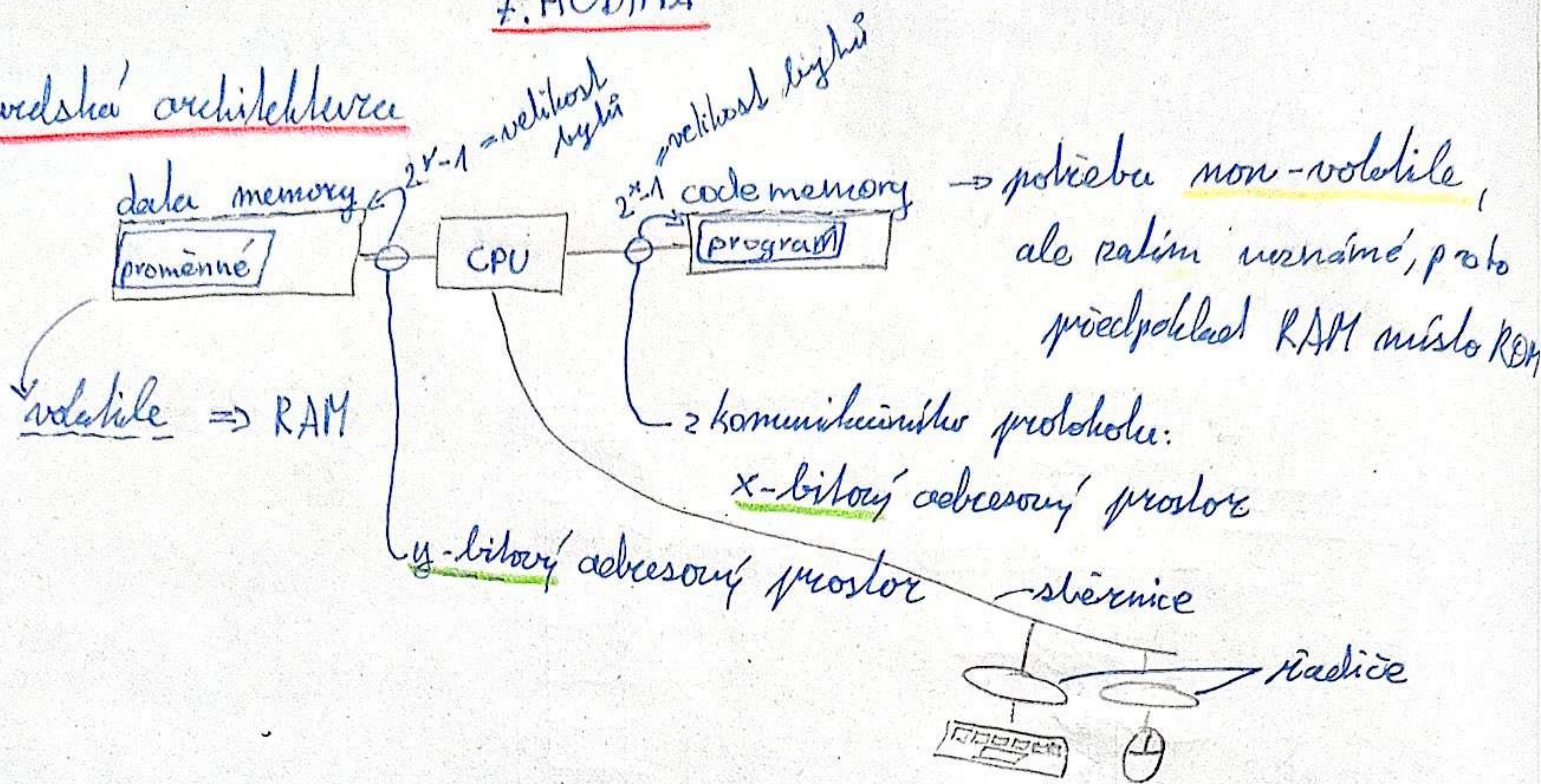
↳ 1 memory read = 1 I²C nápis (do adresového registru)
+ 1 I²C čtení (z datového registru)



I²C proximity sensor → má 10 registrů na různých adresách

7. HODINA

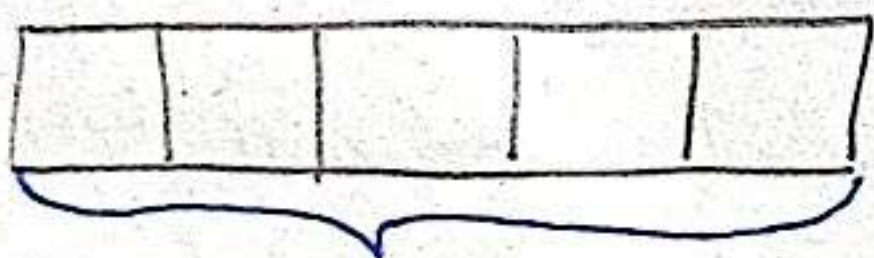
Harvardská architektura



Instrukce procesoru

↳ od výrobce instrukční sada (instruction set)

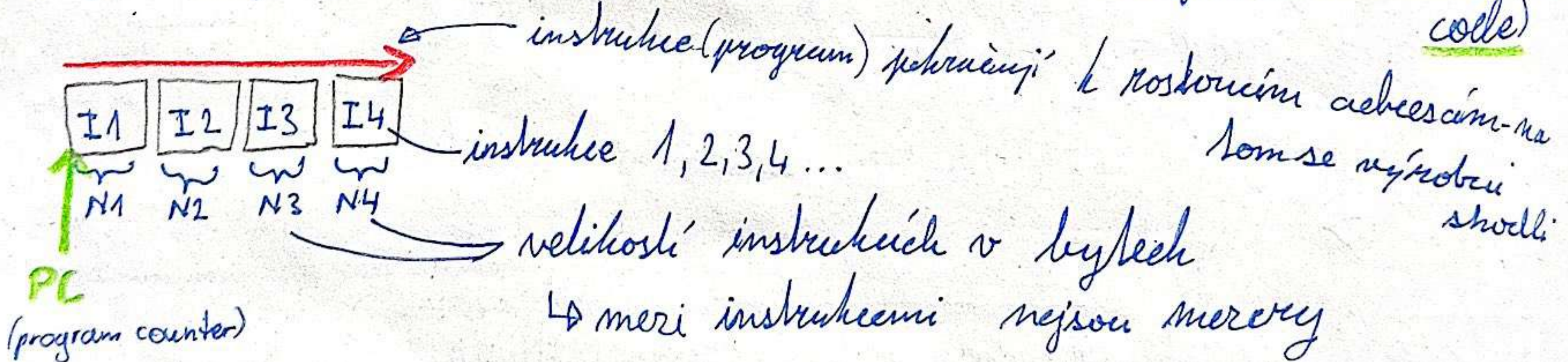
↳ výhled instrukce



N bytů

↳ u některých CPU mají různé instrukce různé velikosti v bytech
a u některých CPU jsou stejné

program (jeeliné, co procesor umí vykonávat) = strojový kód (machine code)



- tabulka vypada program v hčlové paměti

↳ CPU zpracovává právě jednu instrukci a má registry (nejčastěji SPAR), kde si uchovává konkrétní instrukce

↳ Instruction pointer (IP) = "ukazatel" = Program Counter (PC)

↳ znamená to adresa v paměti → v operační paměti

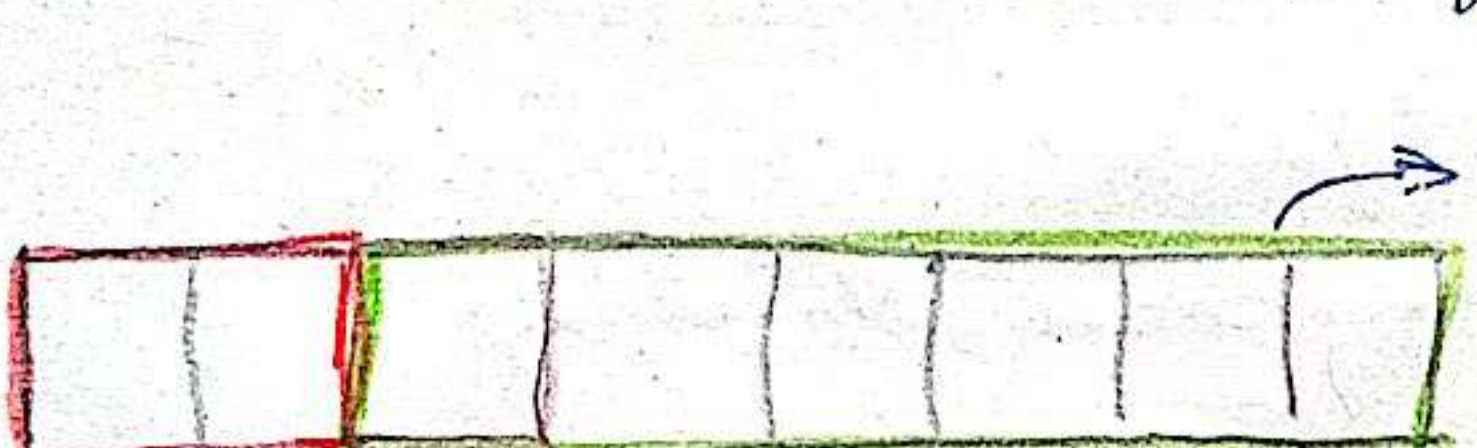
↳ pravidla ukazuje na první bit včetlové instrukce

$$PC := PC + NI$$

↳ délka dokončené instrukce

↳ máme ukazovat na novou instrukci

↳ na rozpoznání instrukce programem slouží op code



argumenty → občas nemá, ale většinou ano
→ bude tam něco o čem instrukce pracuje → např. proměnné, ale jen jejich adresa logicky

op code → "operation code"
→ N byla velikosti

4-bitová, protože v data memory

→ různé značky, např.

ADD 01FA } mají zabudovanou délku instrukce, o kterou se má posunout PC
SUB 01FB } jsou opravdové op kódy
INC 0105 → např. implicitní instrukce

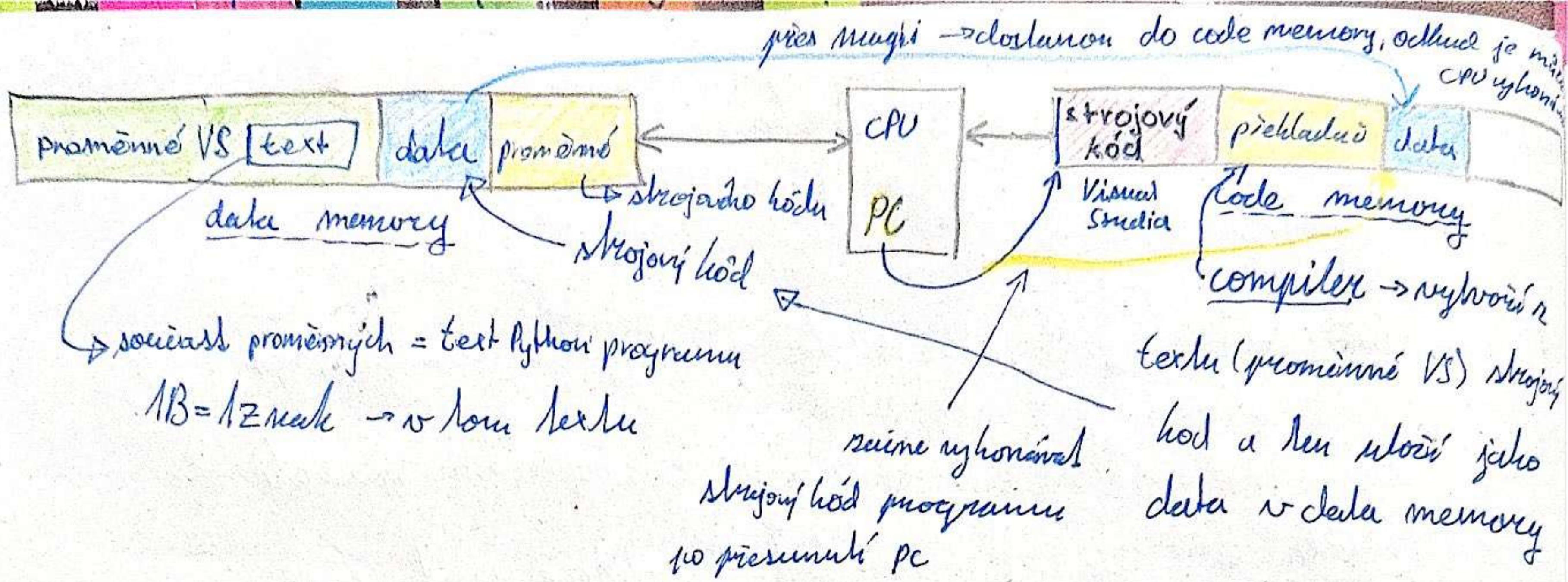
(base addr.)
básová adresa = nejmenší adresa, kam ukazuje PC



→ vzdálenost od básové adresy = offset

PC ← \$100. (program counter je na adrese \$100)

↳ x-bitový register, aby se tam vešly adresy z celé memory



↳ procesor provádí něco dělá, i když "nic nedělá"

↳ vlastně compiler vytvoří z kódu v jazyce C# / C++ strojový kód, ten se uloží v kečkové paměti a ten CPU vykonává → proměnné si ukládá do data memory

↳ interpreter (česky) = "interpret" (anglicky)

↳ pokud je jazyk tak moc komplikovaný, že si dělá co chce



kód vytvořen z textu z VS, z data memory pomocí interpreteru

normální compiler → $c = a + b$ napíše $\text{ADD}(c, a, b)$

addr2
addr3
addrn

interpreter → $c = a + b$

↑↑↑↑↑ čte znak po znaku

a nevěda kaskádí ifů: → if znak = "+":

prom1 = find("c")

prom2 = find("a")

prom3 = find("b")

ADD prom1, prom2, prom3

char (" " ...)

↳ interpreter naprogramovaný například v C, který pole běží ve strojovém kódu

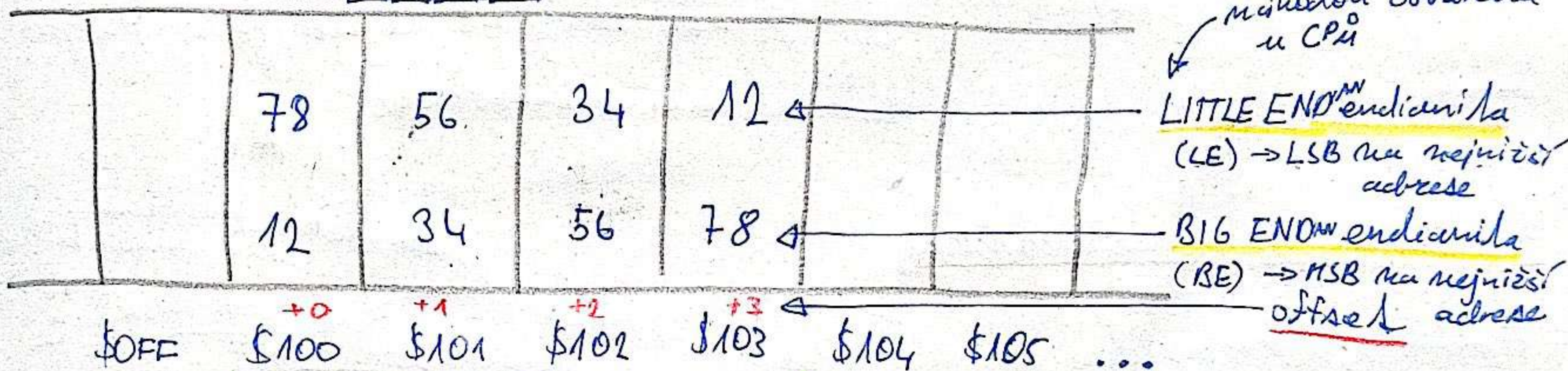
↳ program přes interpreter je vždy pomalejší než strojový kód přes compiler

↳ organizaci dat v paměti vymyslí architekt, protože procesor nerozlišuje, kde je volná adresa

Vícebytové proměnné

4-bytové (32-bit) proměnná v rámci 32 uložena na adrese \$100

$a = 0x12345678$



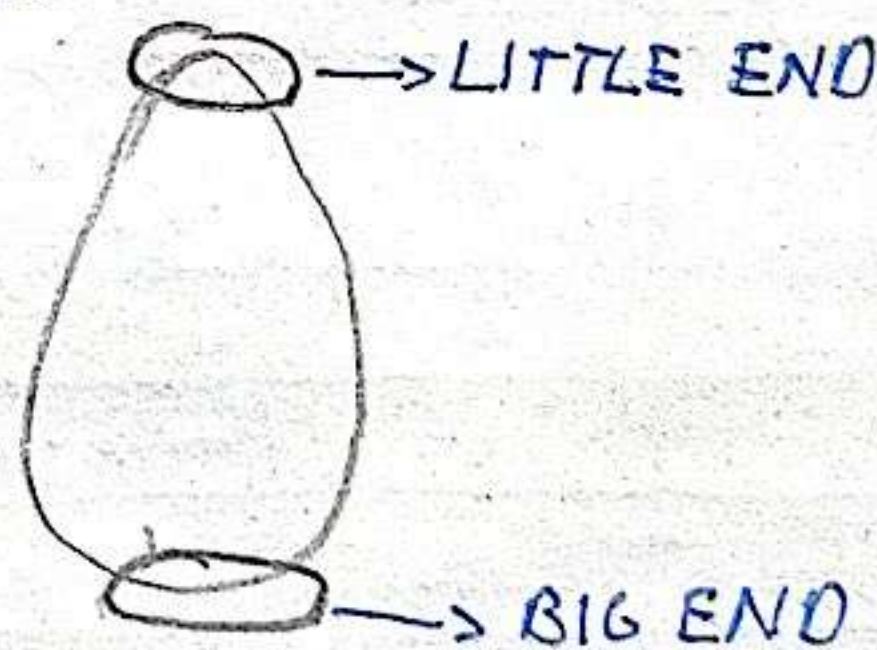
base address of variable

next byte will be stored

endianita ("endianess")

↳ z Gulliverových cest podle skupin lidí, kteří jedli vejce z různých konců → Little Endian
→ Big Endian

vejce:



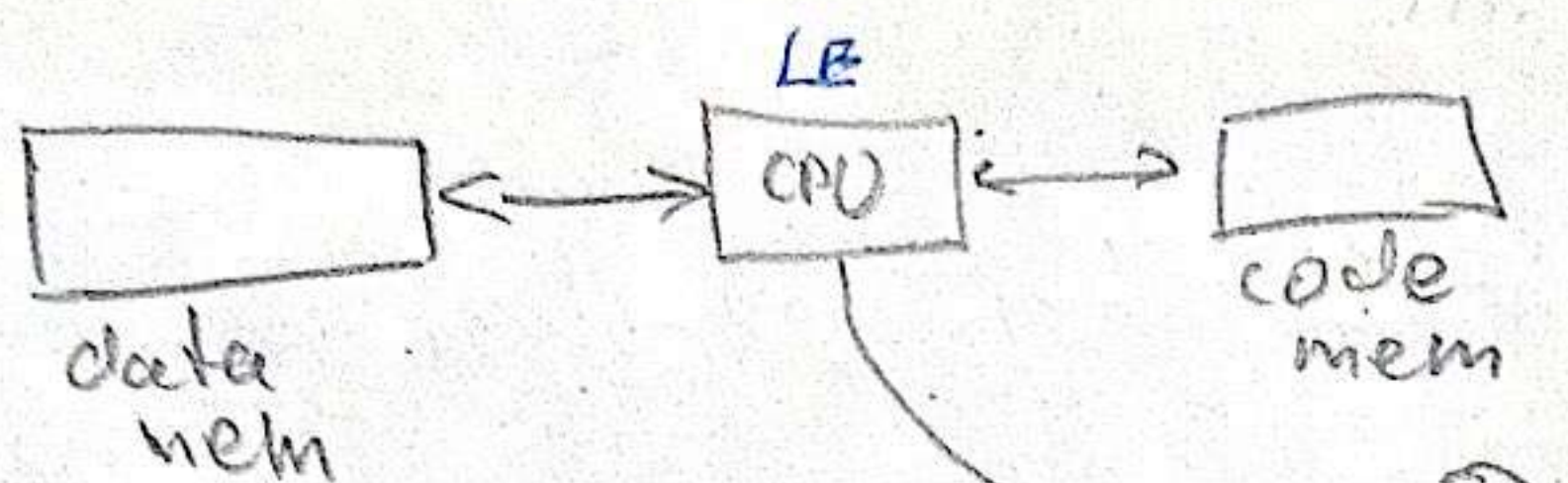
↳ je to vlastně jedno, jen je potřeba si něco zvolit

↳ každý CPU má svou endianitu

↳ endianita bitů a bajtů může být rozdílná → 8-bitová má stejnou endianitu

↳ zapamatovat pomocí LLL → LOWEST ADDRESS

↳ kdy je třeba řešit endocuitu jeho programátor



vernu z jednoho počítače a CPU LE a
dám do počítače a BE CPU → data se
interpretují blbě

pevný disk
(nevedl, v
počítači)

sítová
karta = „radio síť“

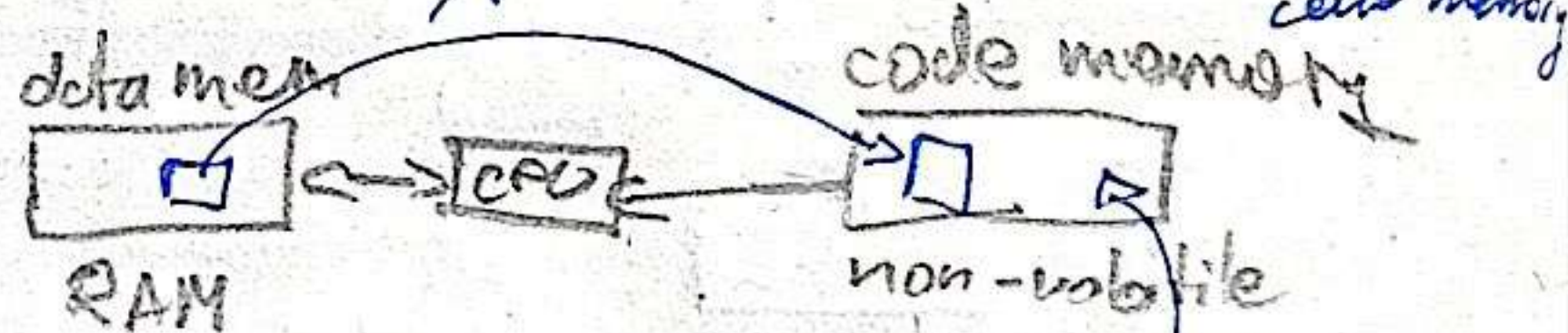
ten jiný počítač
interpretuje data

data přes síť jinde v jiném pořadí

↳ síťové protokoly vznikaly většinou během doby,
kdy CPU měli častěji BE, ale teď je to neoprávněné
data se musí přeuspořádat

Von Neumannská architektura

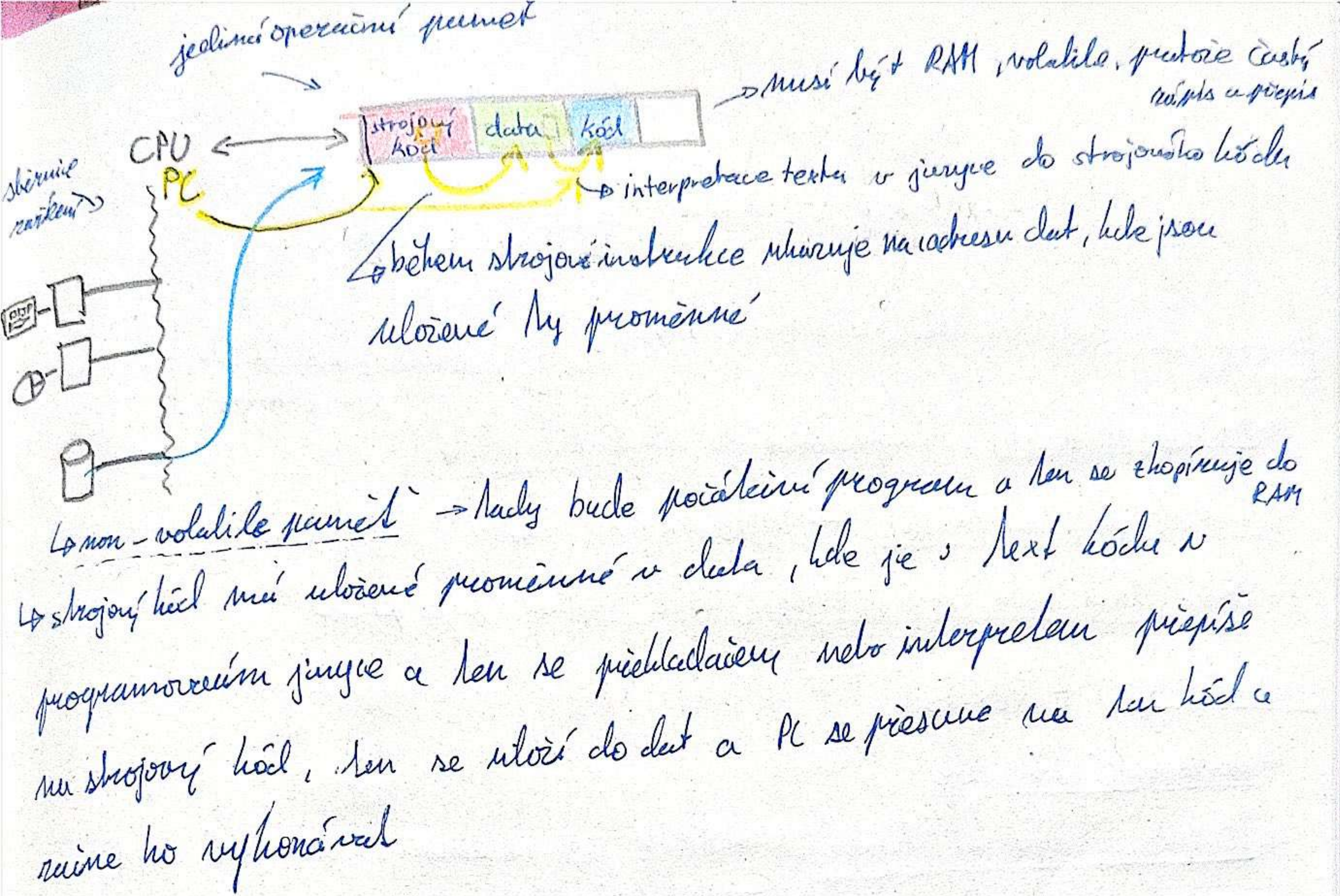
↑ problém a Harvardskou architekturou:



dostup
program
- problém

↳ von Neumannská je nejběžnější

	Harvard	von Neumann
paměť	data + kód na různých sběrnici	data + kód na stejné sběrnici
užití	mikročipy	počítače



Historie počítačů

UNIVAC (1951) → počítač přes celou místnost, drahý \$150 000 — \$1 500 000
→ na počítači mohl a dával

Altair 8800 (1974) → o dost menší, Intel 8080
→ Microsoft Basic (Bill Gates / Paul Allen)
→ interpret strojového kódu

Apple I (1976) → musel se vyrobit case, MOS 6502, stál \$666
→ Steve Wozniak
→

Apple II (1977) → úspěch, protože měl aplikaci VisiCalc, něco jako Excel
→ taky MOS 6502, \$1200

Atari 2600 (1977) → konzole, ne má vykonávan počítač Atari 800 (1979) → MOS 6502
→ \$500

- Začaly vznikat další počítače v ČSSR se používal Intel 8080, ale jen překládač

↳ IBM PC (1981) → Intel 8088

→ \$1500 - \$3000

→ udržiely se do dnes PC (intel i3, i5, i7)

MOS 6502 → nejjednodušší instrukční sada

↳ dříve pracoval s daty v 8-bitech

↳ 8-bitový procesor → pracoval jen s 8-bit daty

↳ 16-bitový adresový prostor → 64 KiB operační paměti umí naadresovat

↳ von Neumann architektura, little endian byte pořadí

Intel 8088 = x86-16

↳ 16-bitový procesor

↳ 20-bitový adresový prostor

→ 1 MiB, ale většinou měli 16 KiB, 64 KiB

→ mylně tvrdili tomu, že to dokázalo naadresovat

↳ little endian

x86 (IA-32) → architektura

↳ vznikla na konci 90. let

↳ 32-bit data

↳ 32-bit address space → 4 GiB

→ se specializací op káčen

↳ mají 2 instrukční sady: - pro kompatibilitu s 16-bit daty procesory

- nové instrukce

AMD/x64/x86-64

↳ 64-bit data

↳ 64-bit adresový prostor

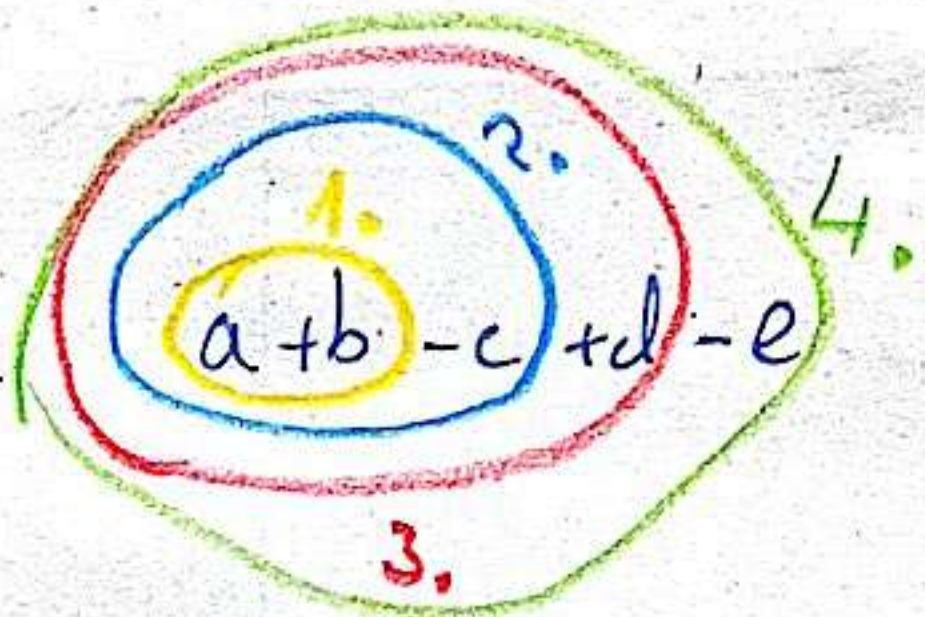
↳ 3 instrukční sady → 16, 32, 64-bitové, pro kompatibilitu

Základní instrukce → pamatovat princip

6502 ^{LE}	x86(IA-32) ^{LE}	komentář
$\emptyset$	$\emptyset$	offset od báze instrukce
$\$EA$ $PC := PC + 1$ $\hookrightarrow 16\text{-bitový}$ assembler: NOP	$\$90$ $EIP := EIP + 1$ $\hookrightarrow 32\text{-bitový}$ assembler: NOP	nic mědělej, ale posune PC/IP
$\$4C$ XX_0 XX_1 $PC := XX_0 XX_1$ $\hookleftarrow LE$ assembler: JMP $\$XX_1 XX_0$ $\hookrightarrow$ jen reprezentace str. kódu	$\$E9$ XX_0 XX_1 XX_2 XX_3 $EIP := \$XX_3 XX_2 XX_1 XX_0$ $\hookleftarrow LE$ assembler: JMP $XX_3 XX_2 XX_1 XX_0 h$	skok na adresu X (jump/branch) $\hookrightarrow$ nepodmíněný skok
$\$A9$ XX_0 $A := \$XX_0$ $PC := PC + 2$ (protože instrukce dlouhá 2 byty) assembler: LDA $\#\$XX_0$		6502 načte 8-bitovou hodnotu $\$XX_0$ do registru A nebo-li konstantu <u>immediate</u>
$\$AD$ XX_0 XX_1 $A := \text{MemReadByte}$ ($\$XX_1 XX_0$) assembler: LDA $\$XX_1 XX_0$ $PC := PC + 3$		6502 načte instrukci z absolutní adresy do registru A $\hookrightarrow 16\text{-bitová}$
		$\$A2$ XX_0 LDX $\#\$XX_0 \rightarrow$ ekvivalent pro registr X 6502 načte instrukci z absolutní adresy do registru A $\hookrightarrow 16\text{-bitová}$
		$\$AE$ XX_0 XX_1 LDX $\$XX_1 XX_0$ } ekvivalent pro registr X

$\$8D \rightarrow STA$

matematický výraz



→ jsou tam i store instrukce:

$STA \$XXXX$
 $STX addr$
 $STY addr$

} uloží se na adresu, není tam konstanta → blbost

x86, 6502 → umí $a := a \text{ op } b$

, neumí $c := a \text{ op } b$

$\hookrightarrow$ maximálně jedna proměnná, je adresa, druhé je registr

- $C = a + b$
- 1) LOAD A z adresy do registru R1 LOAD $R1 \leftarrow \text{addr}(a)$
 - 2) ADD A a b z adresy a výsledek do registru R2 ADD $R2 \leftarrow R1 + \text{addr}(b)$
 - 3) STORE, registr R2 na adresu c STORE $\text{addr}(c) \leftarrow R2$

load/store architektura

↳ aritmetické instrukce pracují jen s registry a do registru si musí naložit z adres

6502 registry

↳ speciální

↳ např. PC → 16-bitový kvůli adresování prostoru

↳ obecné → nemají význam

↳ má registry A, X, Y → 8-bitové

transfer instrukce (move)

^{cell}
TAX $X := A$

chybí: $Y \rightarrow X$

TXA $A := X$

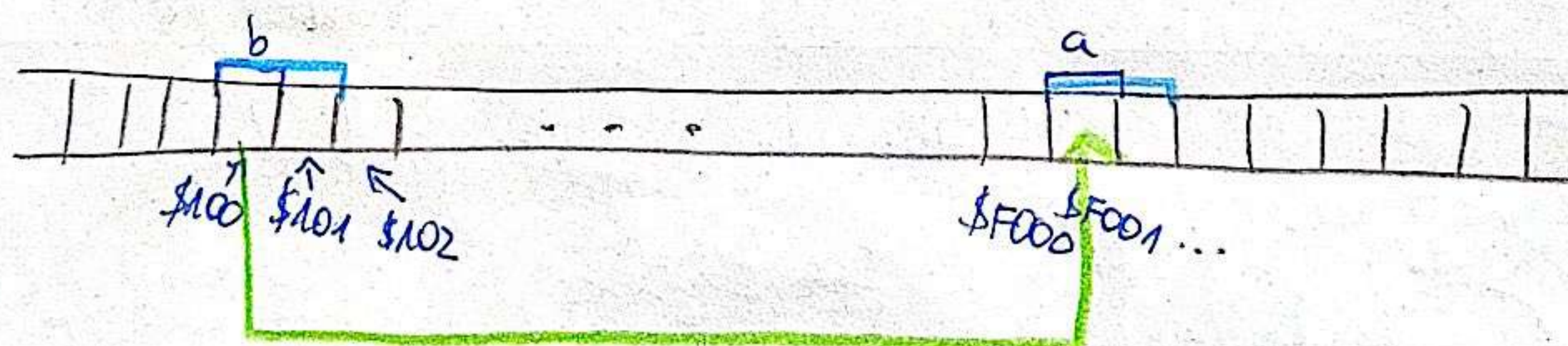
řešení: $Y \rightarrow A \rightarrow X$ LYA LAX

TAY $Y := A$

TYA $A := Y$

TSX $X := S$

TXS $S := X$



↑
 $a = b$

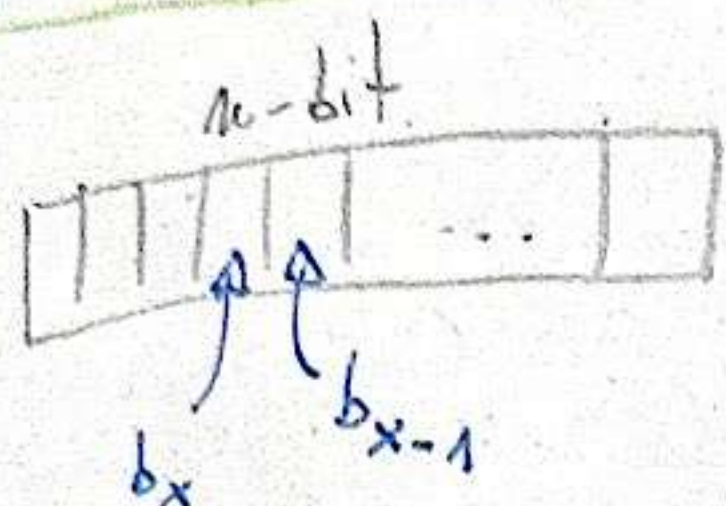
16-bit: LDA \$0100
STA \$F001

$a = b$

8-bit: LDA \$0100
STA \$F001
LDA \$0101
STA \$F002

} lepší proto 16-bit CPU pro toto,
pro 16-bit CPU by to byly jen 2
instrukce

1 bit informace = příznak / flag



→ pokud pracujeme s informací velikosti slova

→ a seriově musíme jeden bit zpracovávat pomocí systému šifrování a druhý bit levě → byly to příznaky

příznakový registr (flag register) → takový asi 8-bitový, příznaky uložený na různých bitech

↳ registr pro jedno-bitové informace

↳ nelze s ním pracovat přímo, ale přes příznaky

↳ ZERO

↳ 0 tomu, jak dopadla poslední operace, zda byl výsledek 0

↳ 1 = AND, výsledek byl nula

↳ 0 = NE, výsledek nebyl nula

↳ SIGN/NEGATIVE

→ v two's complement

↳ hodnota 0 znamená, uloží se do registru SIGN

↳ 1 = záporné

↳ 0 = kladné

↳ uloží se i u lemmánových, ale když všechno chápeme bezvadně, tak je to v pořádku

↳ CARRY

↳ univerzální příznak, když bude nějaká instrukce potřebovat bit navíc

Práce s podmínkami

↳ side efekty

↳ podmínky se upraví po provedení instrukce, ale nemají vliv na výsledek operace
instrukce

↳ obecná registrová architektura (např. x86)

↳ lze si vybrat registre → volnost pro programátory, ale potřeba více instrukcí

↳ akumulatorová architektura (6502)

↳ akumulator → výsledek instrukce, potřeba jen jednoho registru
instrukce pro práci pouze s akumulátorem

↳ CLC → clear carry P. Carry := 0 (CLR)

↳ SEC → set carry P. Carry := 1 (SET)

n x 86

CLC - clear carry

STC - set carry

CLZ - clear zero flag

STZ - set zero flag

↳ (podmíněný skok / conditional jump)

if condition:

JMP "target-address"

else:

NOP

→ podmínky: if flag:

if NOT(flag):

↳ n 6502 - 118 to 127 branch = jump

R := PC + value

argumenty int8 podmíněné číslo	{	BCC	- branch if carry clear
		BCS	- branch if carry set
		BPL	- branch if Plus (negative flag clear)
		BMI	- branch if Minus (negative flag set)

Bitové operace

6502

Komentář

ORA imm/addr

$A := A \text{ Bitwise Or } \text{imm/addr}$

AND imm/addr

$A := A \text{ Bitwise And } \text{imm/addr}$

EOR imm/addr

$A := A \text{ Bitwise Xor } \text{imm/addr}$

? NOT

EOR #\$FF

→ NOT je XOR se samými jedničkami

ASL A

$A := A \text{ shl } 1$

LSR A

$A := A \text{ shr } 1$

ROL A

$A := A \text{ rol } 1$

ROR A

$A := A \text{ ror } 1$

1010
1111
xor
0101

A → akumulátor u 6502

→ tam se provádějí aritmetické operace, neexistují pro osbtení akumulátoru

↳ včera v assembleru je např. ORA jediná instrukce, ale ve

strojovém kódu 1: → 1. pro imm → \$09 xx₀
2. pro addr → \$00 xx₀ xx₁

např. SHL A, 3 → ASL A
ASL A
ASL A } posunuli o 3

polovi SHL A, N

$0 \leftarrow N$

→ ASL A

$0 \leftarrow 0 - 1$

BNZ 0

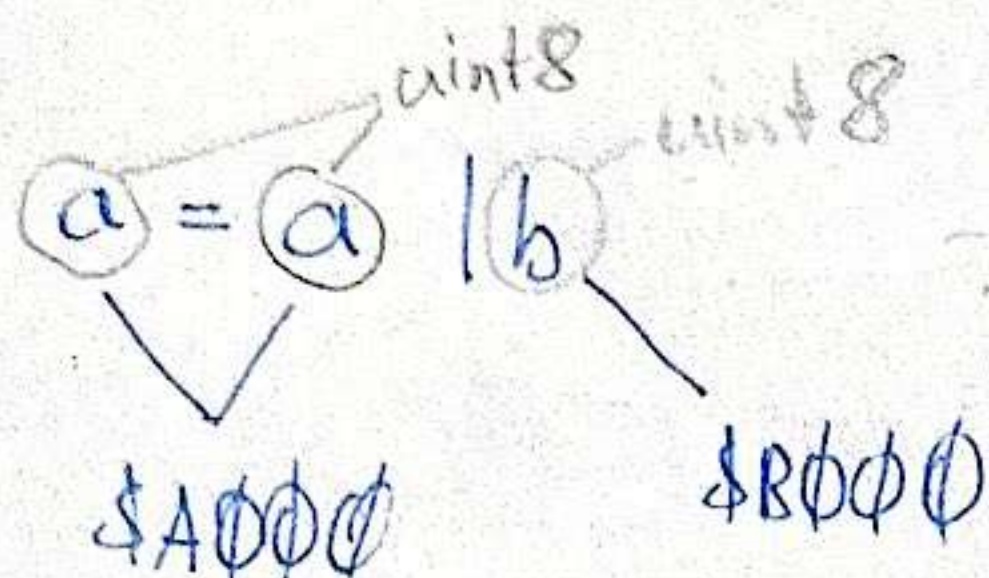
branch if not zero

→ bude nastavovat zero flag, bude vědět, jestli jsme došli do 0 a můžeme ukončit cyklus

kód z pythonu

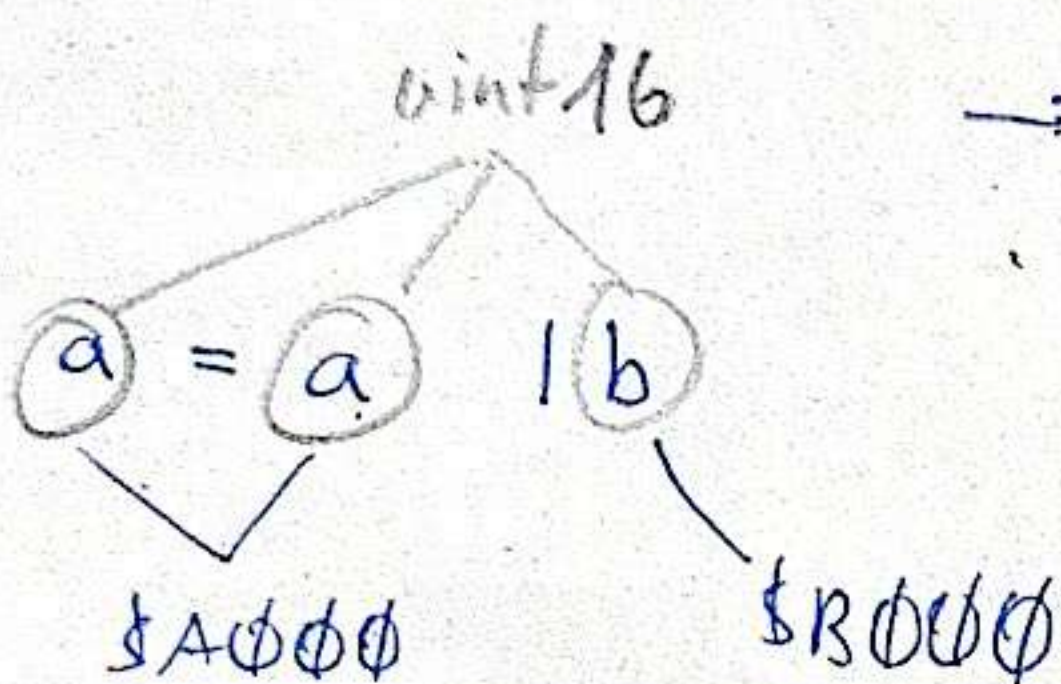
↳ musíme povést numpy → uint 8, kde máme jak velké jsou

proměnné



strojový kód

```
LDA $A0000
ORA $B0000
STA $A0000
```



→ teď je potřeba navíc řešit endianness → LE/BE

↳ ale u OR-u je to jedno

strojový kód:

```
LDA $A0000
ORA $B0000
STA $A0000

LDA $A0001
ORA $B0001
STA $A0001
```

u 16-bit CPU by stačilo
načíst jen jednou a OR jednou
a store jednou

Sečtení

1 1 0 1 1 0 1 1

Input A

1 0 1 0 1 1 0 1

⊕
Input B

1 1 0 0 0 1 0 0

Carry

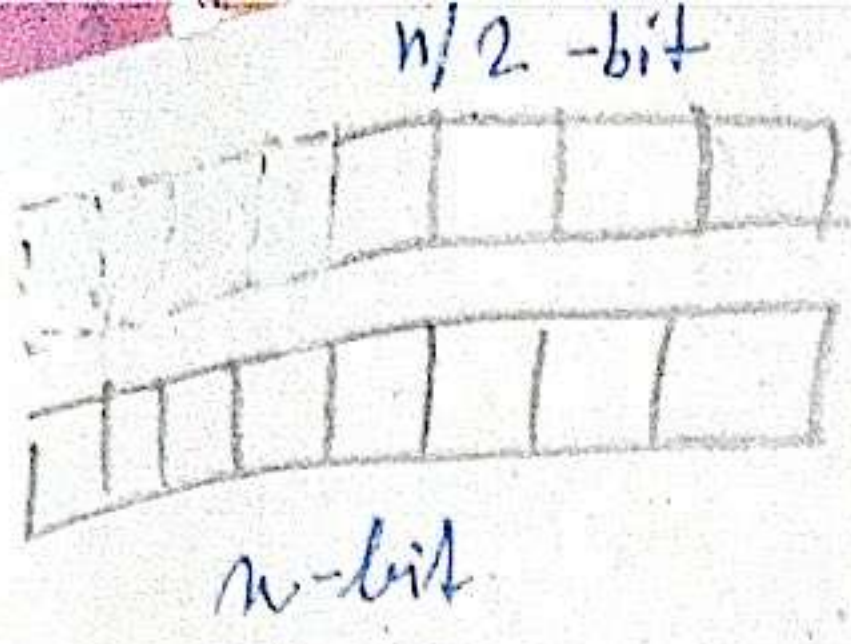
1 1 0 0 0 1 0 0

Result

↑
carry in

↳ 3d = 11₆

↳ 2d = 10₆ → autožin jen LSB cifry,
MSB poslu jako carry

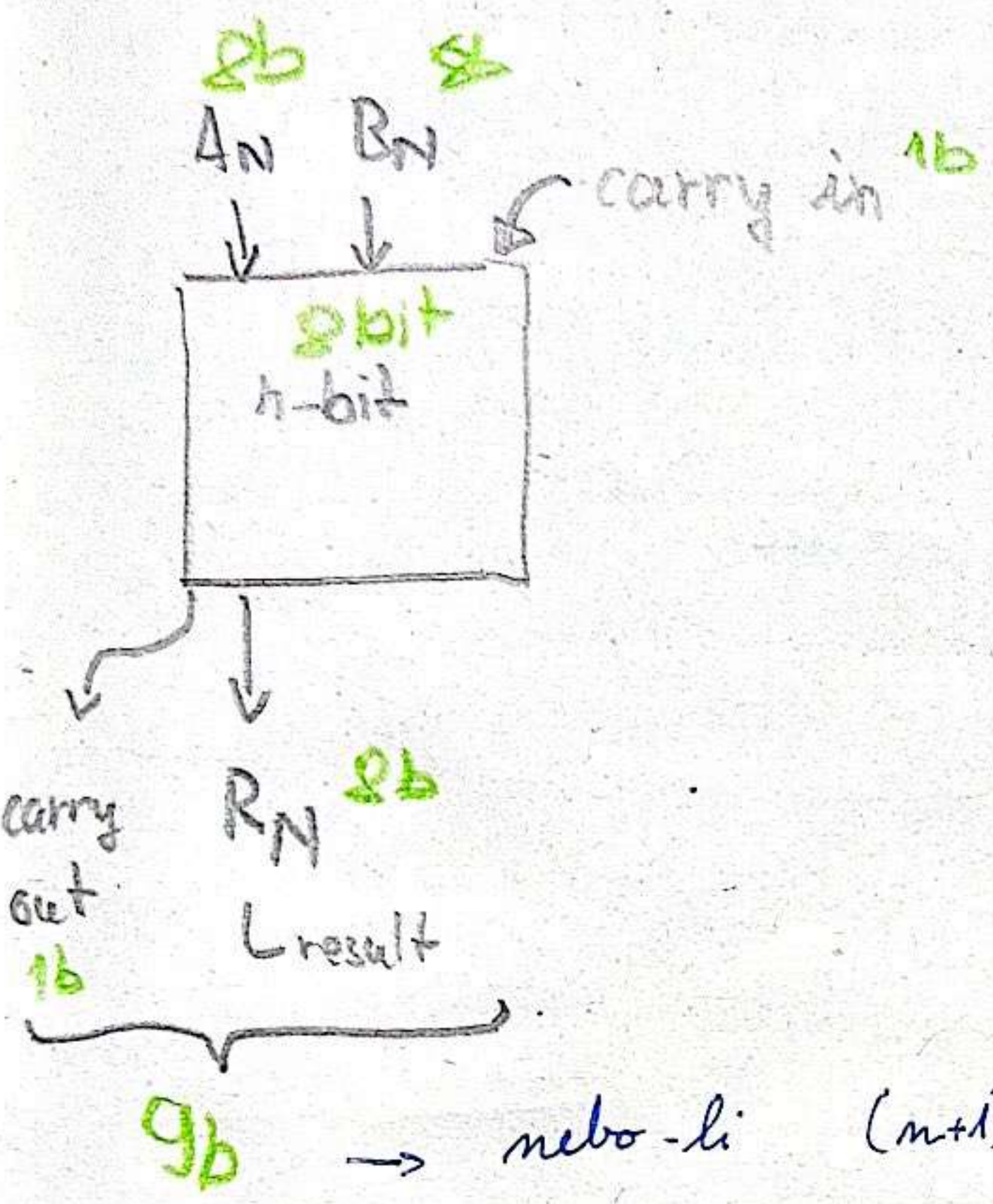


→ normalize: extension → int (sign).
→ unsigned (zero)

$C = a + b$

uint 16
uint 32

príkladu zrejisti extension



add with carry (ADC)

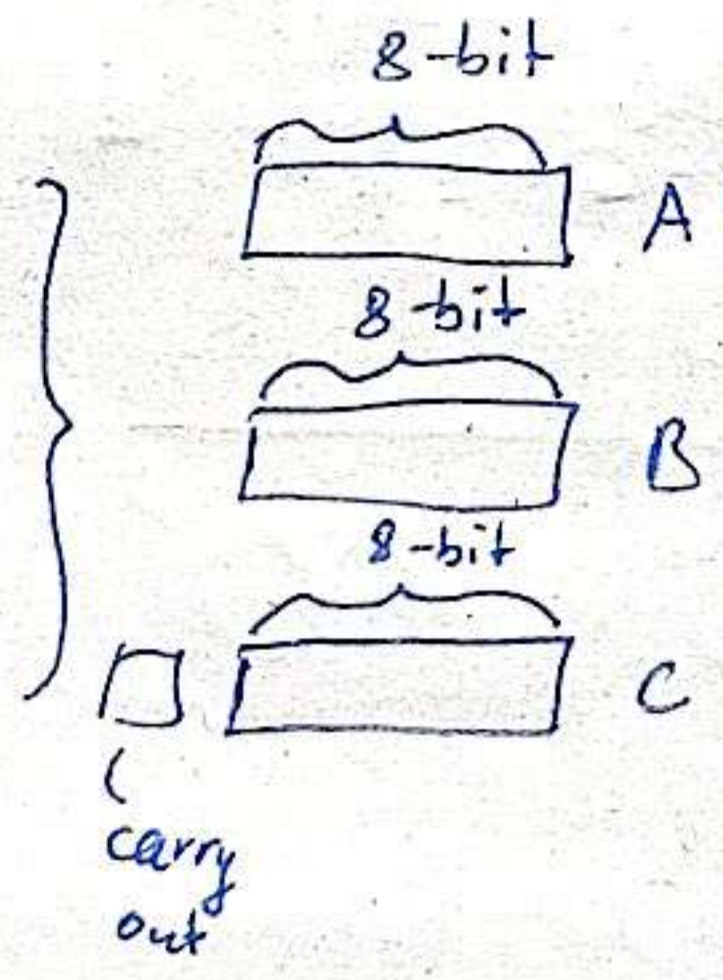
ADC immediate (address)

→ na carry se používa carry flag

→ keď „krabíček“ lze napojiť ľubovoľne na seba → carry out jedného se napojiť na carry in druhého

→ sčítaní dvoch 8-bit čísel → 9-bit výsledok, kde udeláme truncation

```
LDA $A000
CLC
ADC $B000
STA $C000
```



ale pomocou CLC to symuluje, aby nejaky minulý sčítaní to neovlivnilo

LDA → mení zero a negative flag, ale nemieni carry flag

→ pre 16-bitové sčítaní na 8-bit premenných: → slejny
→ pre namnožená čísla → slejny

```
LDA $A000
CLC
ADC $B000
STA $C000
```

→ LDA \$A001
ADC \$B001
STA \$C001

Odečítání

① $A := \text{value} - A$ $\rightarrow$ převést se dá na: $A := A - \text{value}$

$\xrightarrow{\text{STA}} \downarrow \text{temp1} \quad \downarrow \text{LDA} A$

$\rightarrow$ převést pomocí sečítání

$A := \text{value} + (-A)$ nebo-li $A := (-A) + \text{value}$

koncept: NOT A
INC A } do chybějícího doplňku
ADD value

Správně: EOR #\$FF

CLC

ADC #1

CLC

ADC value

$\leftarrow A := (-A) + \text{value}$

② $\rightarrow$ protože je časté, tak ta instrukce existuje

SBB (subtract with borrow)

je -1, proto se musí „půjčit“ z vyššího řádku 1

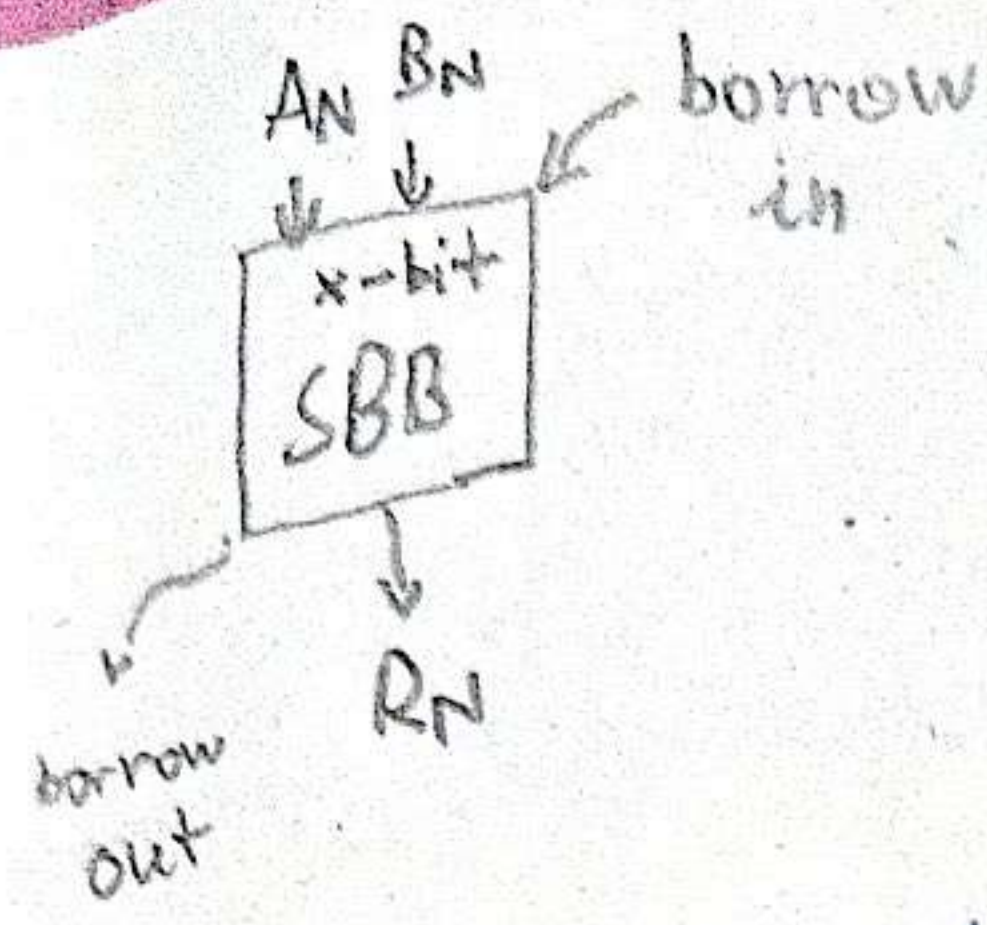
$$\begin{array}{r} 13 = 1101 \\ - 7 = 0111 \\ \hline 6 = 0110 \end{array}$$

$$\begin{array}{r} 13 = 1101 \\ - 7 = 0111 \\ \hline 6 = 0110 \end{array}$$

$$6 = 0110$$

$\rightarrow$ propůjčím si číslo z vyššího řádku, abych získal větší číslo, od kterého odečítám. Ten borrow pak přičtu k odečítateli

$\rightarrow$ předpokládáme božnáměnné odečítání



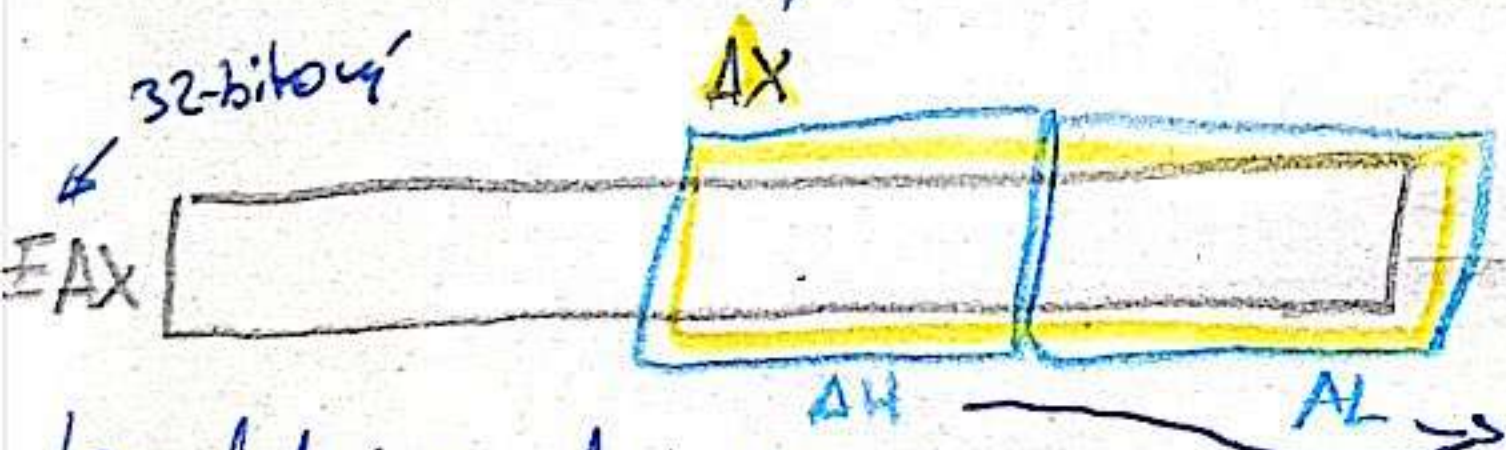
→ borrow se ukládá do carry přístupu

- ③ SBC (subtract with carry) → dělá SBB interně, ale do carry přístupu uloží negaci borrow
- ↳ explicitně nastavit carry na 1 → SEC, pak lze SBC („u SBB to je CLC“) → na 0

x86

↳ více registrů = více svobody → 32-bitové registry, 7 jich má

↳ 16-bitový fiktivní registr



→ většina procesory nemají 2x 8-bitový registr, ale otevírá se to blíž k

↳ další registry: EBX, ECX, EDI

↳ u 64-bit(x64) má 4 rozdělení registrů: 64, 32, 16, 8

příklady instrukcí

↳ obecný tvar (assembleru) → OP target, source

MOV r, [addr]

↳ 32-bitová adresa do 32-bitového adresového prostoru

↳ lze provést i výpočet a pak to dát jako adresu

↳ ADD/ADC/SBB add. r reg imm/addr

↳ CLC
ADC

↳ mají velké množství možností

↳ má i pravě SUB (ADD) → bez přenosu

→ místo aby se psalo CCL
ADC jak u 6502

↳ IMUL / IDIV → násobení a dělení, náročné hardware implementace
→ není u 6502

Komplexní výraz

$$r = a + b + e - (c + d)$$

u 6502

$$TEMP1 = a + b + e$$

$$TEMP2 = c + d$$

$$r = TEMP1 - TEMP2$$

LOA \$F01

CLC

ADC \$F02

CLC

ADC \$F05

TAY

LOA \$F03

CLC

ADC \$F04

STA \$0108

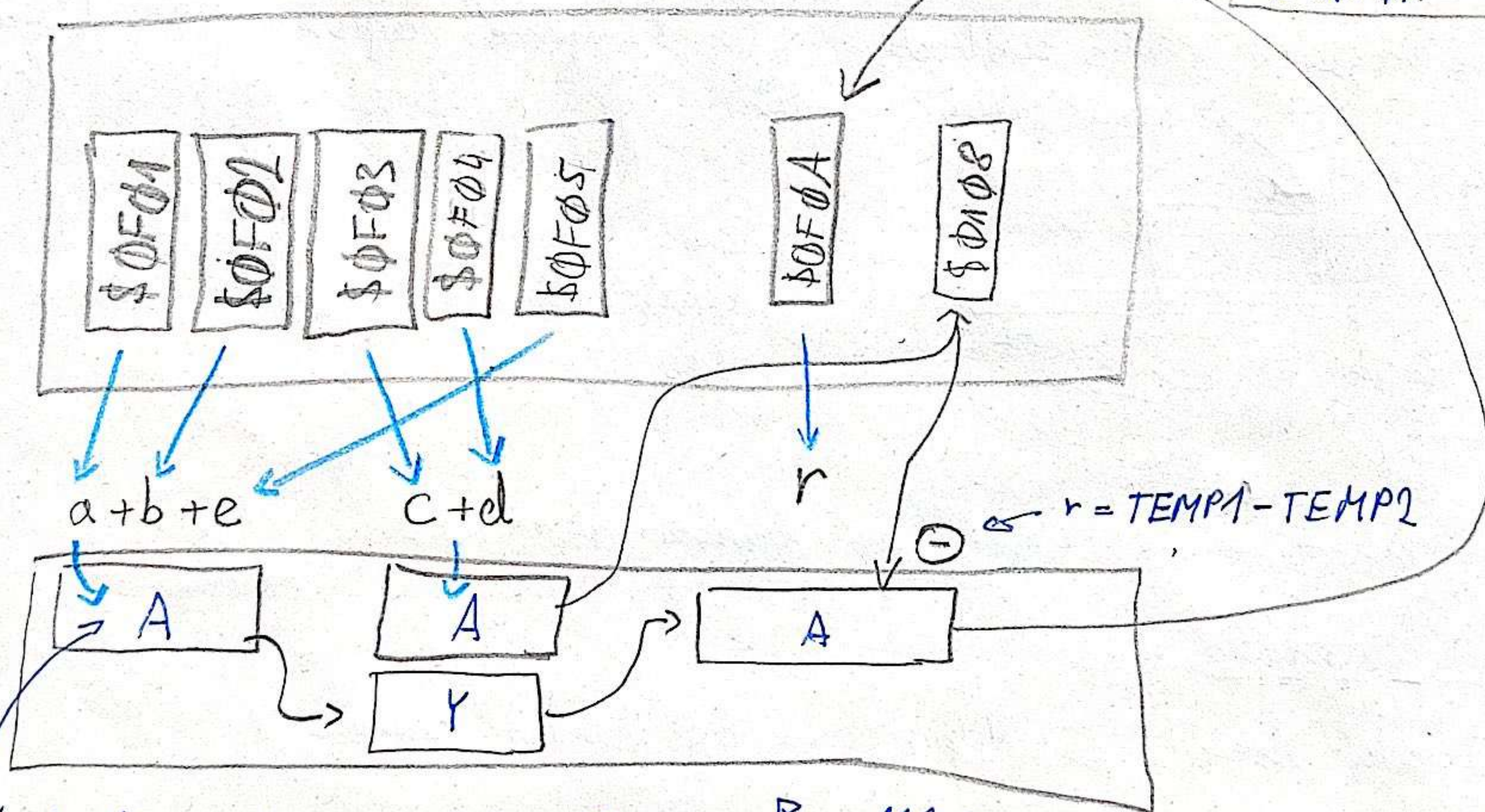
TYA

SEC

SBC \$0108

STA \$F0A

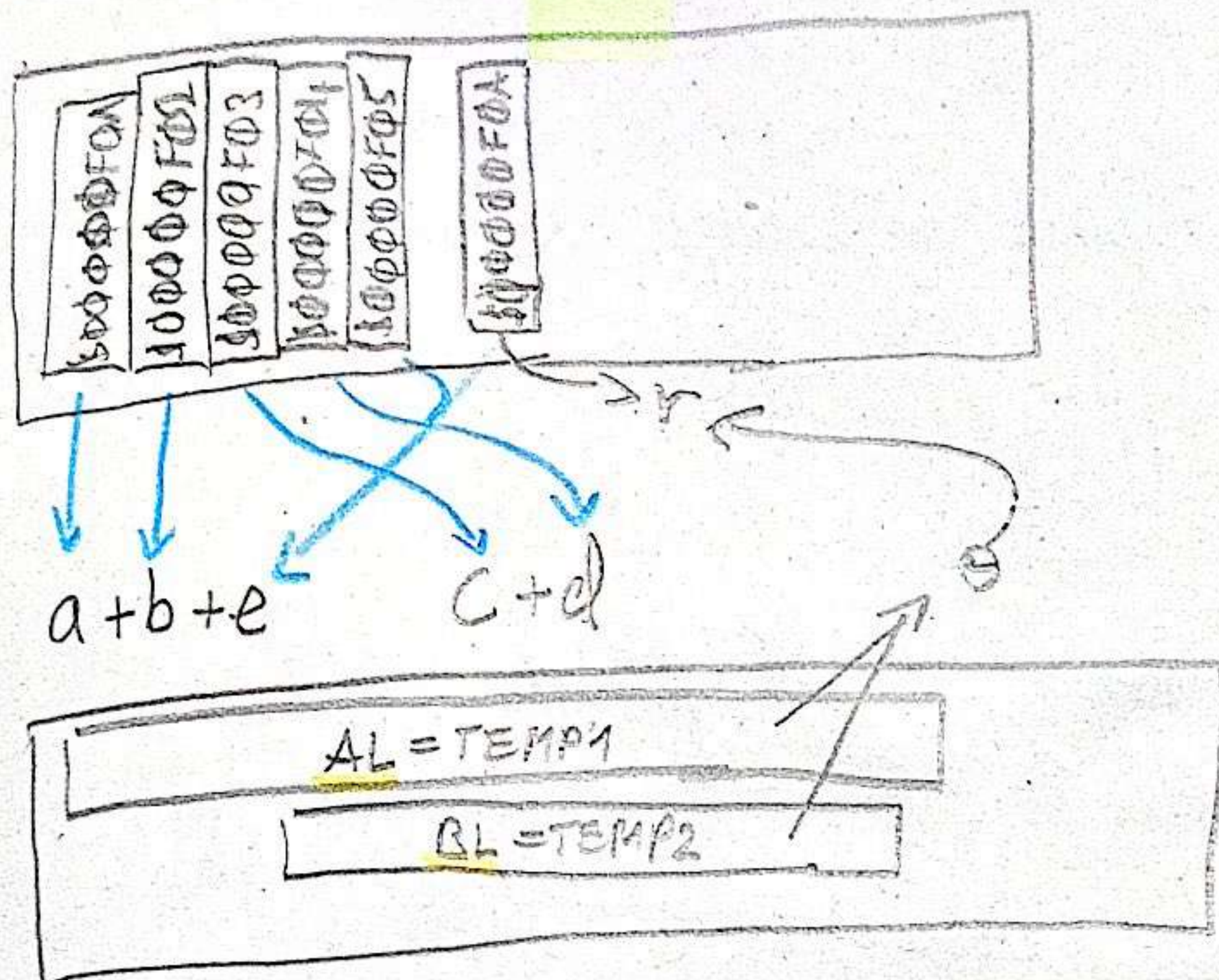
uložení



hardware sčítání 8-bitových

➤ odečítání do registru je rychlejší než do paměti, ale odečítání neexistuje instrukce mezi registry

na x86



MOV AL, [000000F01h]

ADD AL, [000000F02h]

ADD AL, [000000F05h]

MOV BL, [000000F03h]

ADD BL, [000000F04h]

SUB AL, BL

MOV [000000F0A], AL

↑
u 8-bit proměnných

u 32-bit proměnných → jen se změna adresy .. 4 byty dále vždy

např. 000000F00h → a

000000F04h → b ...

↳ a používá se 32-bitový register EAX místo 8-bitového

9. HODINA

Načítovací frekvence

↳ např. 6502 → 1 MHz
80386 → 33 MHz

} dnes už běžně ~ 6 MHz

↳ udává 1 náhl / 1 cyklus = Hz

↳ a každá instrukce má určitý počet náhlů

① Fast

↳ mají 1 náhl → občas i subnáhl, se zpracuje více instrukcí

↳ bitové operace → AND, OR, bitové posuny

→ SHL, SHR

↳ sčítání / odčítání → ADC, SBB

② Slow

↳ práce s operační pamětí → LOAD / STORE

↳ ADL μ 1, [addr]

↳ implicitní load, proto pomale

↳ ADL [addr], μ 1

↳ je mu tom ještě hůř: - implicitní load
- implicitní store

např. $C = a + b$ → pomale, potřeba provést:

32-bit

LOAD ← a
LOAD ← b
ADD
STORE ← c

8-bit
 μ 6502 → 4 krát
 μ 80386 → 1 krát
32-bit

↳ snažil se minimalizovat počet operací s pamětí → ukládal dočasné proměnné do registrů, ne do paměti

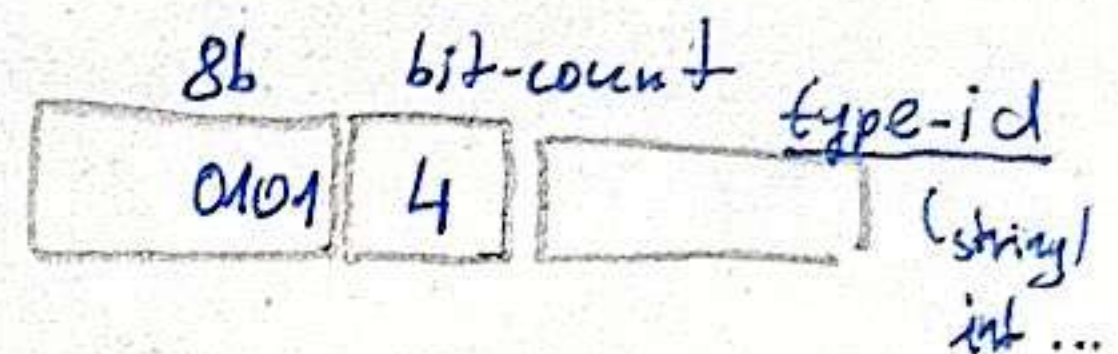
↳ ještě když máme slovo, tak je důležité, jestli je aligned nebo misaligned, pak se musí instrukce provádět breakpoint → viz. předchozí

Python

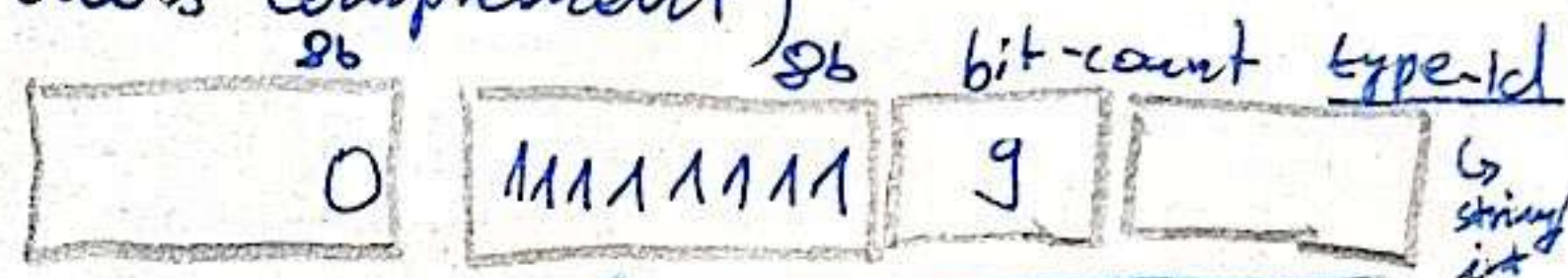
$c = \overset{\text{int}}{a} + \overset{\text{int}}{b}$

uložené v Pythonu:

5 → 101 → 0101 (v two's complement)



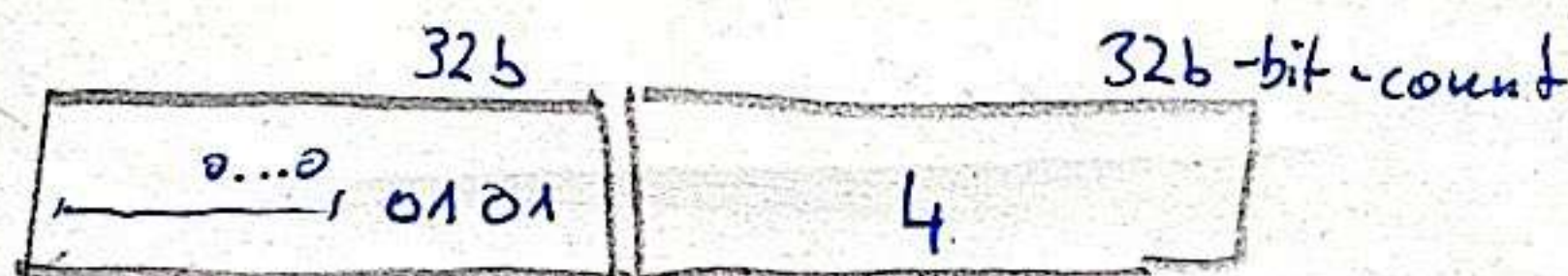
255 → 11111111 → 01111111 (v two's complement)



← adresy rostou doleva

↳ v realitě Python používá 32-bit slůvko a ne 8-bit, takže se tam vejde hodnoty do $2^{32}-1$, neboli $2^{31}-1$ v two's complement

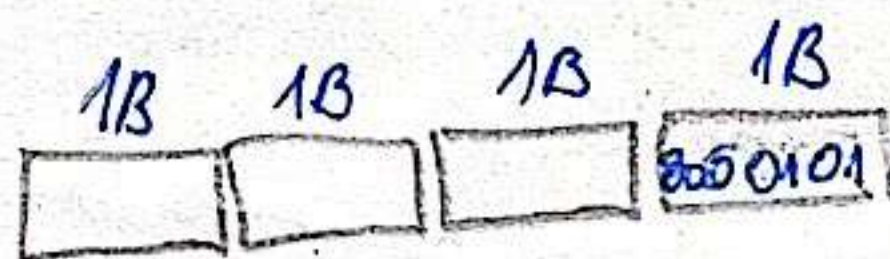
↳ i bit-count bude uložen v 32-bitovém slůvku



C/C#

uint32_t

a=5



→ 4B má nějakou adresu

↑
předem určený typ

$c = a + b$ → LOAD ← a
LOAD ← b
ADD a, b
STORE → c

→ myšlenka oproti Pythonu, že je to trivialní, protože v C# je předem definovaný typ a nikdy se to nemůže chovat jinak. V Pythonu NE.

if ... :

a=5
else
a="ahoj"
c=a+b

→ někdy - int sečtení
- string concatenation

→ proto má Python type-id

Objekt v Pythonu

a=5


→ uložená adresa = ukazatel / pointer

↳ reference → mechanismus zjistit, co kam je,
Python to věší na nás

↳ 64-bit adres space → 8B } nejčastěji v Pythonu
32-bit → 4B

(6502) 16-bit → 2B

proměnná

$$\underbrace{8B}_{\text{adresa proměnné}} + \underbrace{4B}_1 + \underbrace{4B}_1 + \underbrace{8B}_{\text{type-id}} + \underbrace{4B}_{\text{bit-count}} = 28B$$

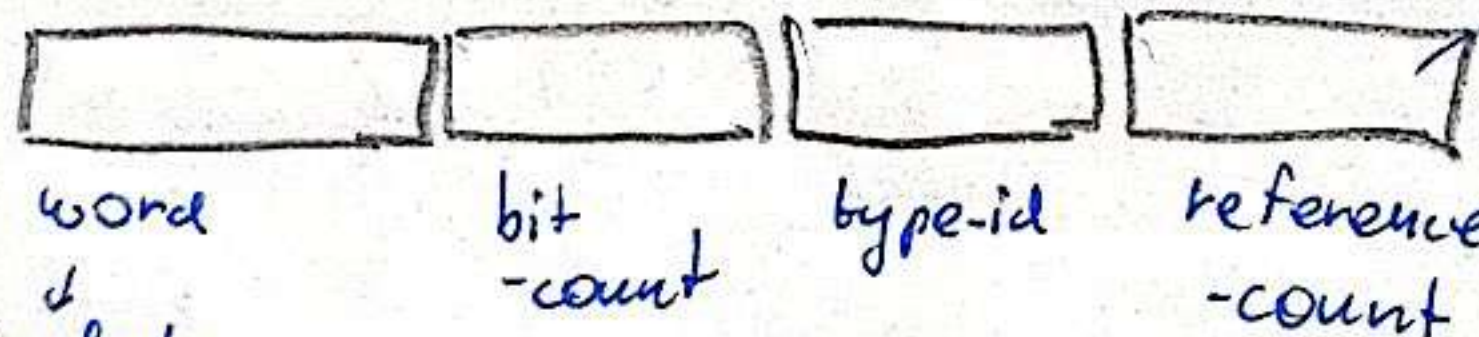
→ jen číslo a=5
type-id - adresa → na popis objektu
word (hodnota) (objekt v C)

↳ Python má typ proměnné z int na string tím, že změni adresu

Garbage collection → garbage collector / GC

↳ mazání něčeho, kam nevede reference

↳ reference counting


word (hodnota) bit-count type-id reference-count
při vytvoření : 0
přiradí se : +1
odebrání : -1

↳ pokud a ukazuje na objekt (adresu jeho) a udělám y=a, tak
ref count se inkrementuje +1

↳ ale pak udělám y=7 a ref count se dekrementuje v ref count

↳ pokud ref count = 0 → může adresu použít na jiný objekt

objekty v Pythonu $\rightarrow$ skelint jsou immutable

$\rightarrow$ nelze objekt změnit, jen se nahrazuje a přikazuje reference

$\rightarrow a = a + b$, kde $a = 5$ a $b = 7$

$\rightarrow$ vznikne nový a a ten starý objekt je izolovaná komponenta souvislosti, kterou garbage collector vymaže

$\rightarrow$ proto je Python až sakra pomalejší než C# $\rightarrow$ C# běží strojovým kódem
 $\rightarrow$ protože tam běží interpret a ne strojový kód

$\rightarrow$ pro zrychlení číselného řetězce z RAM existuje cache

registry B

$\uparrow$ - cache MB

RAM GB

$\rightarrow$ takový počet SRAM, jak registry, ale třeba tak 10x rychlejší než RAM

C# Basic Interpreter $\rightarrow$ pěkný kód, ale jen má složitý
 $\rightarrow$ kódování je pomalejší než C#

Metody C, C#, Java

$\rightarrow$ musíme přemýšlet nad velikostí proměnné

uint8 $a = 255$ 1111 1111

$a = a + 1$ 0000 0000

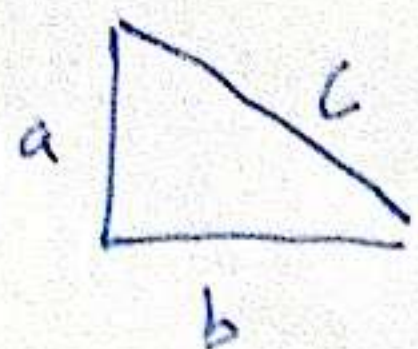
$\rightarrow$ vznikají čísla mod 2^n , např. $255 \xrightarrow{+1} 0$ (arithmetic overflow)
"přetečení"

int8 $a = 127$ 0111 1111

$a = a + 1$ 1000 0000 = -128

"podtečení"

arithmetic underflow $\rightarrow$ kdyby odečtením začaly z malých čísel vznikat větší čísla



uint8 a, b, c

$$c = \text{sqrt}(a*a + b*b)$$

→ m mering steelu dojde k aritmetickému přetěčením

multiplication mul *

division div // (celočíslné)

↳ Srdce vytvořit hardware

pomalejší než sčítání (HW) → softwarově by to bylo ještě pomalejší

mikrokontrolery

pomalejší než sčítání (HW)

1-10x

mul *

x86/64

6502

MC

ARM

10-100x

div //

HW

SW

SW

u drošších

↑
~ telefonu
HW

↳ SW → musím softwarově implementovat

Shift Left (u unsigned integers) → LOGICKÝ POSUN DOLEVA <<

$$\phi\phi1\phi11 = 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$$

(SHL 1)

$$\phi1\phi11\phi = 1 \cdot 2^4 + 1 \cdot 2^2 + 1 \cdot 2^1 = 22$$

$$= 1 \cdot 2^{3+1} + 1 \cdot 2^{1+1} + 1 \cdot 2^{0+1} = (2^3 + 2^1 + 2^0) \cdot 2^1$$

↳ násobím dvěma při SHL 1

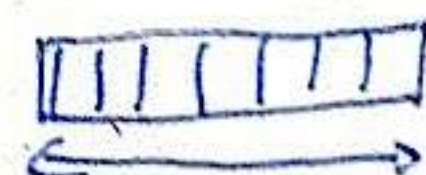
$$C = a * b \quad \text{pokud } b = 2^n$$

ALE pozor: - n-bitové přesnosti

$$x \text{ SHL } n = (x * 2^n) \pmod{2^Z}$$

$$C = a * 2^n$$

$$C = a \text{ SHL } n$$



Z-bit číslo

- nepřesnost pro znaménková čísla

Shift Right (nonsigned int) $\rightarrow$ LOGICKÝ POSUN DOPRAVA $\Rightarrow$

$$\begin{aligned} \text{SHR} \hookrightarrow 001011 &= 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11 \\ 000101 &= 1 \cdot 2^2 + 1 \cdot 2^0 = 5 \\ &= 1 \cdot 2^{3-1} + 1 \cdot 2^{1-1} + \underbrace{1 \cdot 2^{0-1}}_{2^{-1}=0,5} = \\ &= (1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0) / 2^1 = 5 \end{aligned}$$

číslicové dělení //

Ve dvojkovém doplňku

$\hookrightarrow$ musí být dost místa pro SHL

$\hookrightarrow$ pokud kladné v two's complement $\rightarrow$ SHR bude fungovat

$\rightarrow$ jinak SHR nefunguje vůbec!

$$\begin{aligned} \text{SHL} \hookrightarrow 111011 &= -5 \\ 110110 &= -10 \end{aligned}$$

$$\begin{aligned} \text{HL3} \hookrightarrow 111011 &= -5 \\ 011000 &= 24 \end{aligned}$$

Arithmetic Shift Right SAR

$\hookrightarrow$ místo přidání nul jak u SHR, tak vezme MSb a ten zkopíruje

$$\begin{aligned} \text{SAR} \hookrightarrow 111011 &= -5 \\ 111101 &= -3 \end{aligned}$$

} almost DIV2 $\rightarrow$ protože zachovákuje znaménko, při kladném směrem k nule, ale při záporných dál od nuly

microbenchmark $\rightarrow$ test elementárních operací a její rychlosti

$\rightarrow$ dělení je v C/C++/Java zmatelně pomalejší, např. 10x (s stupnicí 1000)

$\rightarrow$ v Pythonu dělení je na stejné rychlosti, protože tam to je jen zlomek

Representace reálných čísel

Leading zeros trailing zeros

0 0 2 0 , 6 2 5 0 0 $2 \cdot 10^1 + 0 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3} = 20,625$

$10^3 \ 10^2 \ 10^1 \ 10^0 \ 10^{-1} \ 10^{-2} \ 10^{-3}$

20

... 0 0 1 0 1 0 0 , 1 0 1 0 0 0 0 ...

... $2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \ 2^{-1} \ 2^{-2} \ 2^{-3} \ 2^{-4} \ 2^{-5}$

" " " "

0,5 0,25 0,125 0,0625

řádová čárka

Fixed-point representation

↳ kolik bitů před a kolik za řádovou čárkou - 8-bit → např. 5+3
5.3

5.3 → 1 0 1 0 0 1 0 1 = 20,625

↑ každý řádová čárka

4.4 → 0 1 0 0 1 0 1 0 = 4,625

↑ vytekla jednotka, s horší přesností, zaokrouhlí mod 16 v tomto případě

6.2 → 0 1 0 1 0 0 1 0 = 20,5

↳ tedy není přesnost za řádovou čárkou

funguje normálně aritmetika → sečítání, odčítání, porovnávání

→ rychlejší než floating-point

→ ale problém s velkými a hodně malými čísly
↳ přesnost

3,500 ⇒ 3500
+ 1,250 ⇒ 1250

4750 ⇒ 4,75 "tisíců"

← pohled jako na velký tisíce

1,25 ~ 5.3 → 0 0 0 0 1 0 1 0

→ pomocí řádové čárky, ale stejná reprezentace →

+ 1,875 ~ 5.3 → 0 0 0 0 1 1 1 1

$$\begin{array}{r}
 00001010 = 10 \\
 + 00001111 = 15 \\
 \hline
 00011001 = 25
 \end{array}$$

$$\rightarrow 00011, 001$$

do fixed point 2 bits

← 8 bits
(value 1 (1 < 3))

← value 10

$\Rightarrow$ 5.3 fixed point value

10. HODINA

číslo lze psát : $\Phi x \lll \lll \dots$ - v hex soustavě

$\Phi b \lll \lll \dots$ - v binární soustavě

$\lll \lll \lll \lll$ - podtrženo = nic neznamená

problém s fixed-point čísly:

$$A = -10100101.00000000,0$$

$$= 1,0100101 \cdot 2^{15}$$

ve vědecké
notaci

exponent

$$B = 0,00000000010100101 = 1,0100101 \cdot 2^{-9}$$

mantisa / significand

↳ ledy bych chtěl reprezentovat v fixed-point 5.3, ale
obě vyjdou 00000,000

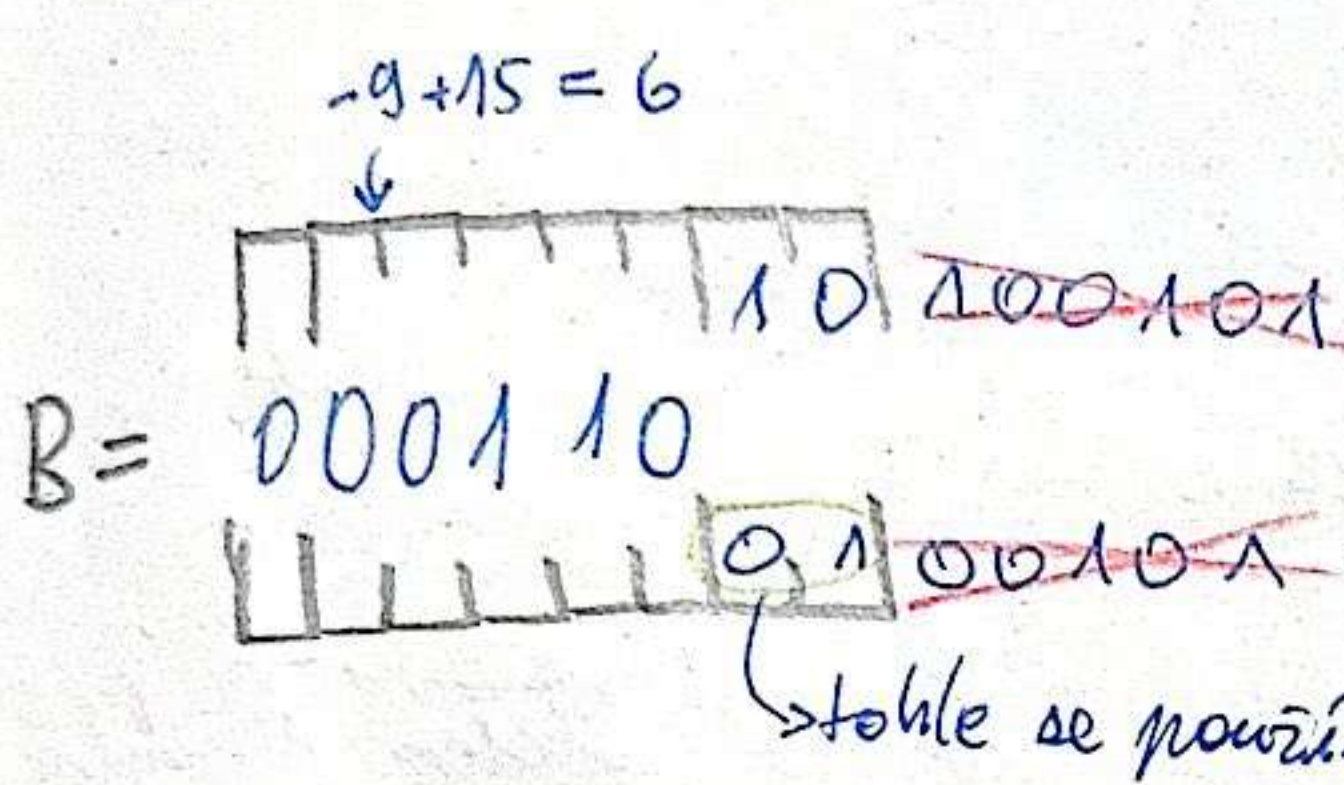
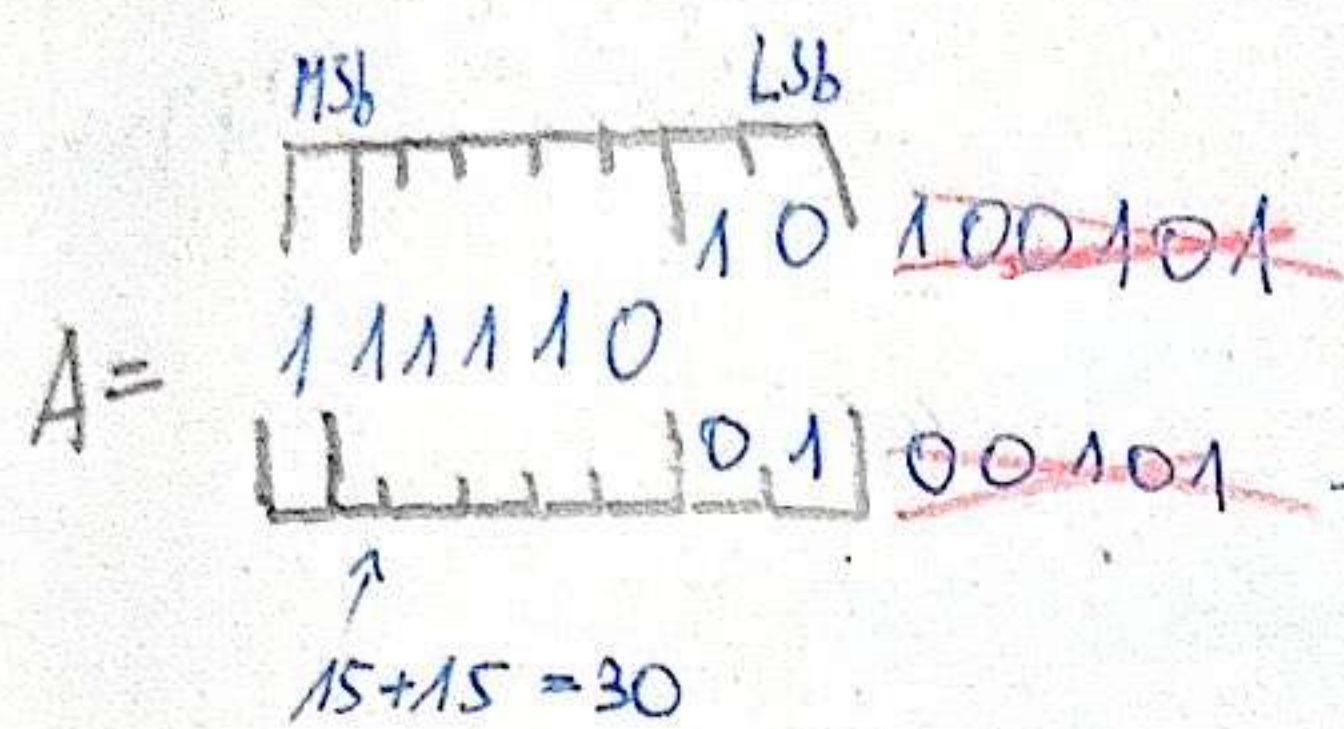
Float / normalizovaný formát

- 1) žádné leading zeros
 - 2) před rádovou čárkou právě jedna cifra
- nula se zatím ignoruje, přidá se později
 - reálná čísla s pokynlivou rádovou čárkou

↳ n -bitů: 5 bitů exponent + 2 bity mantisa + 1-bit sign

↳ $- = 1$

$+ = 0$



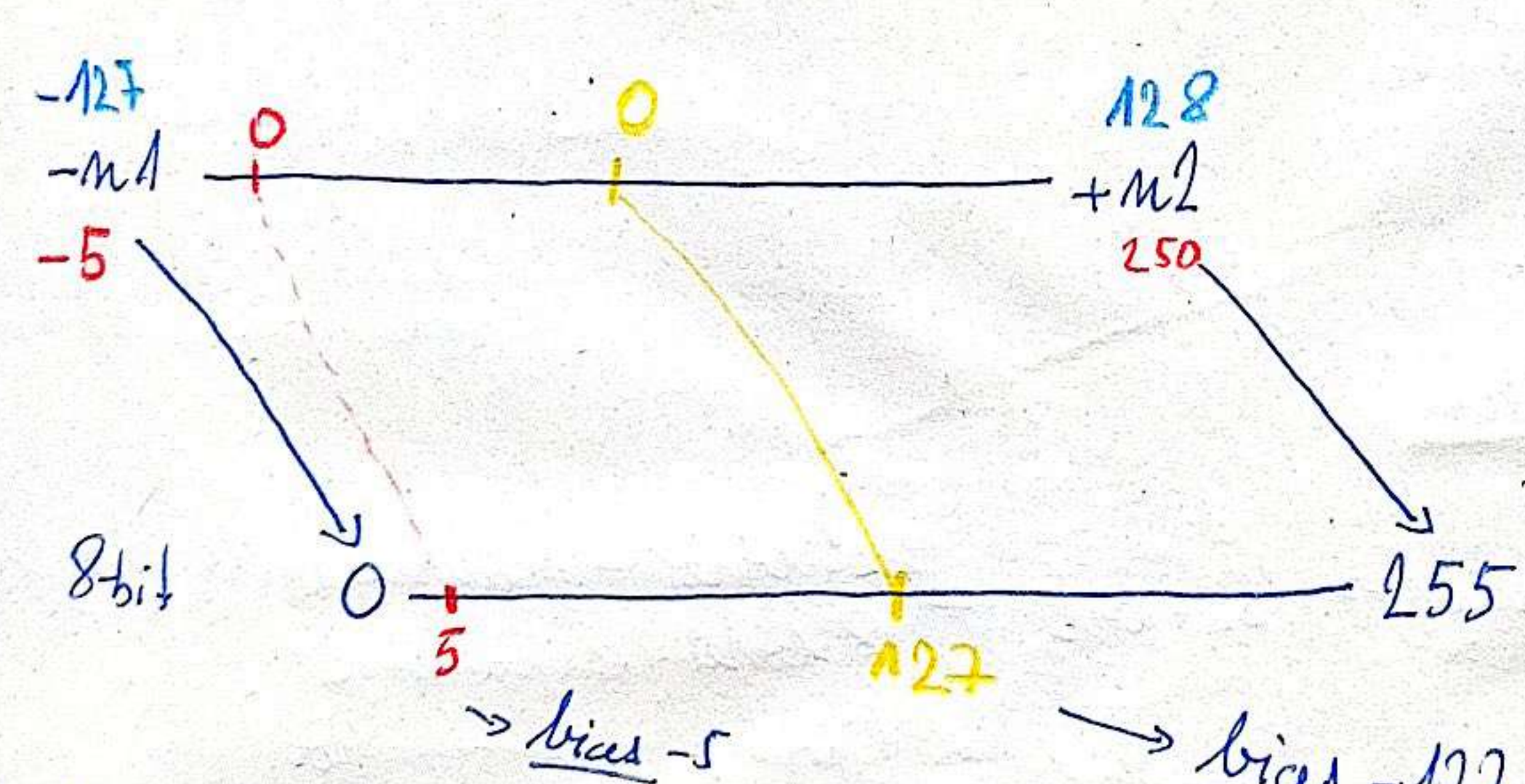
↖ větší přesnost
se skrytou jednotkou, protože číslo jí vždy bude računat kromě nuly

→ takže se používá

- ↳ pro exponent zvolit bias -15, protože velikost je $2^5 - 1 = 31$
- ↳ největší sign (MSB) a mantisa (LSB)

4. reprezentace znaménkových celých čísel → pro exponent

↳ s posunem (bias)



podle toho kam se namapuje nula

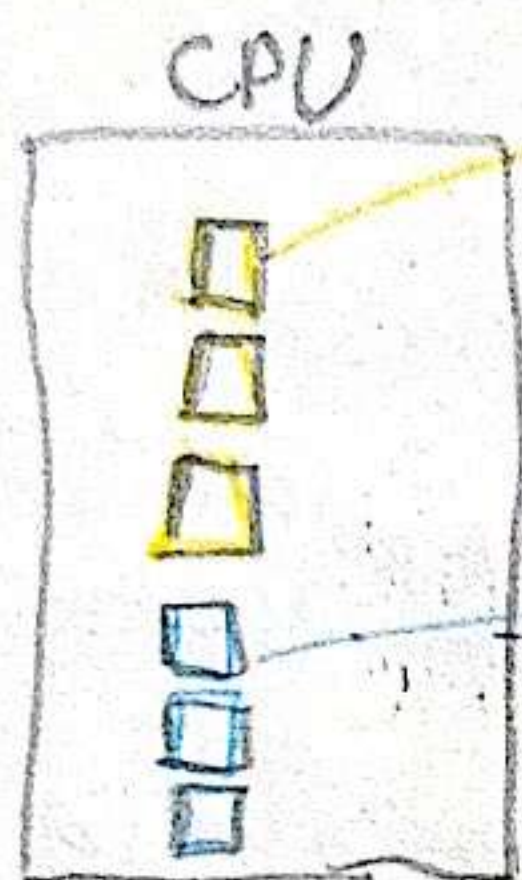
→ 0 takhle hodnoty posouvá správně

bias - 127 reprezentace těchto čísel

- ↳ $-127 + 127 \rightarrow 0$
- ↳ $-126 + 127 \rightarrow 1$
- ↳ $-1 + 127 \rightarrow 126$
- ↳ $0 + 127 = 127$
- ↳ $1 + 127 = 128$
- ↳ $127 + 127 = 254$

pro aritmetické operace je potřeba SW implementace, protože nemáme
využit to dosavadní, co známe → hodnoty instrukcí a převezení do
společné fixed-point

některé procesory mají HW implementaci



unsigned integers

→ existují registry s instrukcemi:

→ LOAD, STORE, ADD, SUB ...

Float

→ také musí stejné instrukce pochopit

→ FLOAD, FSTORE, FADD, FSUB ...

→ u dalších CPU, kde HW implementace byla stejná rychlejší
jak pro Float, tak celá čísla nebo o trochu pomalejší

x86/64

HW

ARM

HW/SW

M

SW

6502

SW

→ musí ale
programovací jazyk
přeložit stejný formát
Float čísel

místo toho se musí používat fixed-point

Standard IEEE 754

→ single přesnost → 32-bit = 23 bitů - mantisa (se skrytou 1)
8 bitů - exponent
1 bit - znaménko

→ double přesnost → 64-bit = 52 bitů - mantisa (se skrytou 1)
11 bitů - exponent
1 bit - znaménko

Python

numpy

IEEE 754

C#

float → float32 → single ← float
float → float64 → double ← double

A+B → z přechází strany

↳ musíme je denormalizovat na stejný exponent

A = 1,0100101 00000000 00000000 00000000 * 2¹⁵
B = 0,00000000 00000000 00000000 10100101 * 2¹⁵
23 bitů

↳ a teď se sečítá a pak renormalizovat do float reprezentace, kterou zvolíme

↳ např. v Pythonu použít numpy: a = float32(2_000_000_000)

b = float32(50)

c = a + b

print(c) → 2000000000.0

↑ 50 se ztratí

↳ při result = a + b + c → NENÍ ASOCIATIVNÍ!

↳ prostě, když sečtu velké číslo a malé, tak ztratím malé, ale když sečtu dvě velké čísla a dostanu menší a přičtu malé, tak se mi trochu zmenší

↳ problém s přesností: např. 10km x 10km svět a potřeba umístit prsten na přesnost milimetrů → zachováte se jinak
→ nebo stačným se mi potýbovat, ale nepohne se z místa

0,1 Kč

↳ ve dvojkové soustavě má nekonečný rozvoj, proto pro bankovní aplikace se lidéji používají haléře jako základní jednotka tedy $0,1 \text{ Kč} = 10 \text{ haléřů}$

↳ přesně jako pro architekta → základní jednotka → mm

někdy ~~float a == float b~~ → kvůli zaokrouhlovacím chybám

↳ místo toho $\text{abs}(a-b) < \epsilon$ → rovnost

Speciální hodnoty

↳ reprezentace velmi malého čísla → denormalizovaný exponent

↳ $2^{-\text{exp}}$ → nejmenší exponent se v bítové reprezentaci zobrazuje do nuly

↳ takže to budou semé nuly + znaménkový bit, ale

ten se v IEEE 754 řeší a -0.0 a $+0.0$ jsou ekvivalentní

↳ v Floatu, int má jen jednu nulu

↳ semé 1 v exponentu

1) mantisa = 0 → a) sign = 1 → $-\infty$

b) sign = 0 → ∞

$$\infty + 5 = \infty$$

$$\infty / 2 = \infty$$

$$5 / 0 = \infty$$

$$-5 / 0 = -\infty$$

$0/0 = NaN$ ← mantisa $!= \emptyset$
↳ ("not a number")

↳ ale existuje jich hodně množství podle verzí

↳ $0/0 + nan = nan$

jecliny Python → 1.0/0 → vyhodí chybu kromě ∞

→ lze kouknout v miny → `isinf(...)`

→ `isnan(...)`

jecliny / microcontroller / μC / MCU → používá se Harvardská architektura

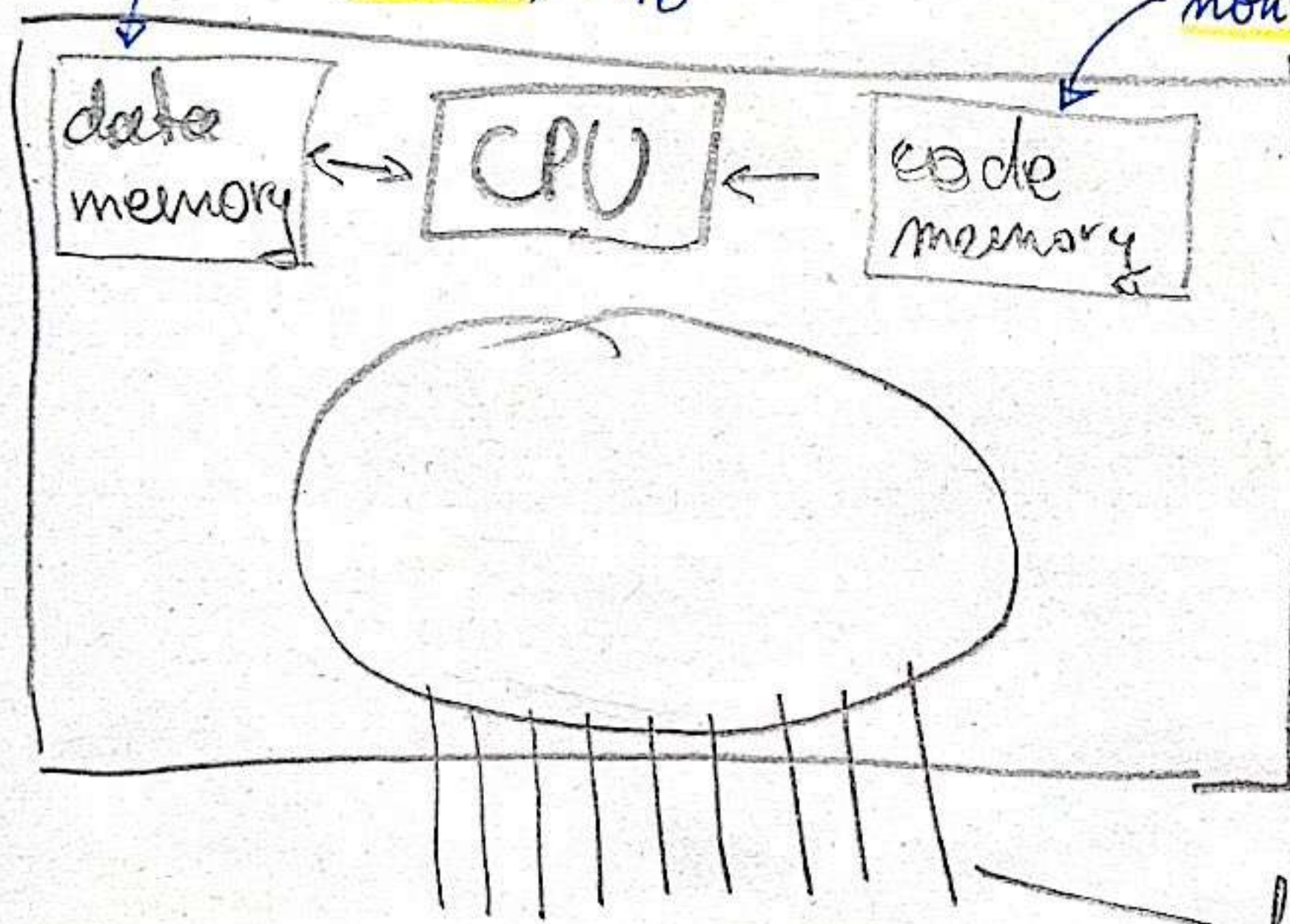
↳ musí být levný, pro různé spotřebiče

"firmware"

SRAM (volatile) ~ kB

non-volatile

→ pro ten program



ROM ①

(Read Only Memory)

∞R

1W

↳ nekonečné čtení, ale jeden zápis

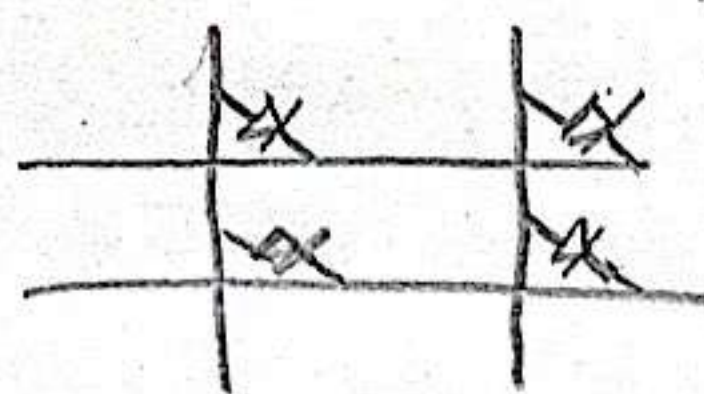
další periférie

↳ jsou levné jen když se objednají v velkém množství → pro startup s nedostupným softwarem nevyhovuje

② PROM (Programmable ROM)

∞R 1W

↳ jeden zápis od uživatele



mřížka

← defaultně jsou všude 1, nebo-li diody nebo něco jiného a přepsáním

se zničí a je tam 0

③ EPROM (Erasable PROM)

∞R ∞W

↳ nejčastěji používané

Typická vlastnost ROM (Read Only Memory) je, že není read-only

11. HODINA

EPROM - note.

↳ před nabitím musím celou paměť smazat (erase)

↳ pomocí UV záření → stačí 20 minut na zemské atmosféře

EEPROM

↳ electrically erasable PROM

↳ byte addressable defaultně

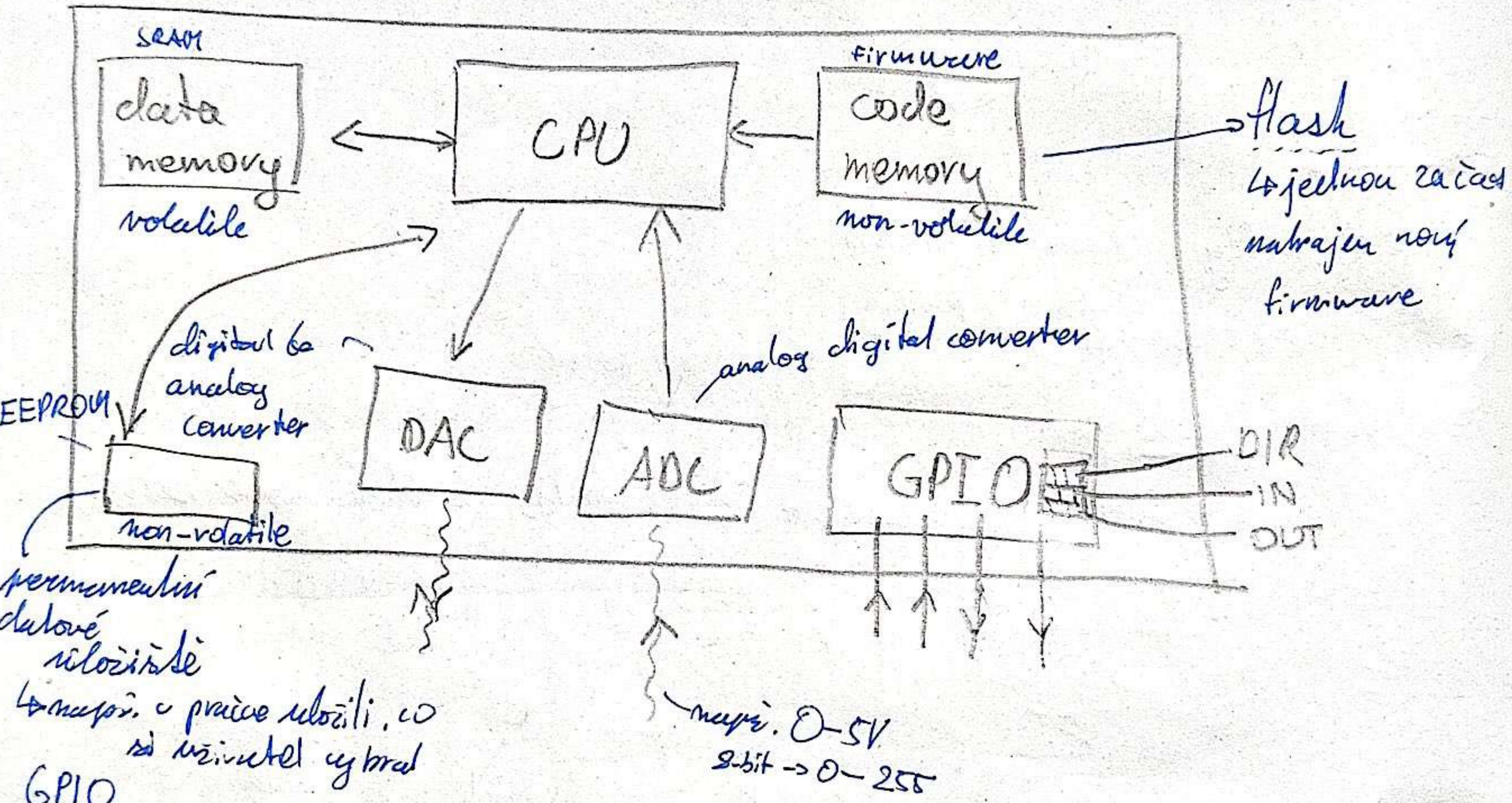
↳ podtyp: Flash

↳ block addressable 1kB - 16kB = jednotky větších slova

Porovnání:

	SRAM	DRAM	EEPROM	Flash	HDD	→ se dál
průměrná rychlost	> 10GB/s	1-10GB/s	100MB/s	16B/s	~100MB/s, 100kB/s	při čekání může stagnovat
přístupová doba	< 1ns	~10ns	~100ns	~100ns	~10ms	
přístup						
schování	✓	✓	✓	✓	sek ↑ ✓ sek ↓ ✗	
random	~	~	~	✗	✗	musím přecítit celý blok ~
block	✓	✓	✓	✓	✓	
↳ vlastně schování						
kapacita	B-kB-MB	1-10GB	100GB	1TB		
max. počet zápisů	~	~	100 000 - 1 000 000	10 000 - 100 000		protože po čase zůstanou problém dostat elektrony z konkrétního místa

- ↳ non-volatile nejprve ideálus na architektuře, protože elektronky se budou snížít dostal ven a počase se vlastně data snovíseu → po rozbití cca
- ↳ taky kvůli funkciování → proveditelnost výsledků

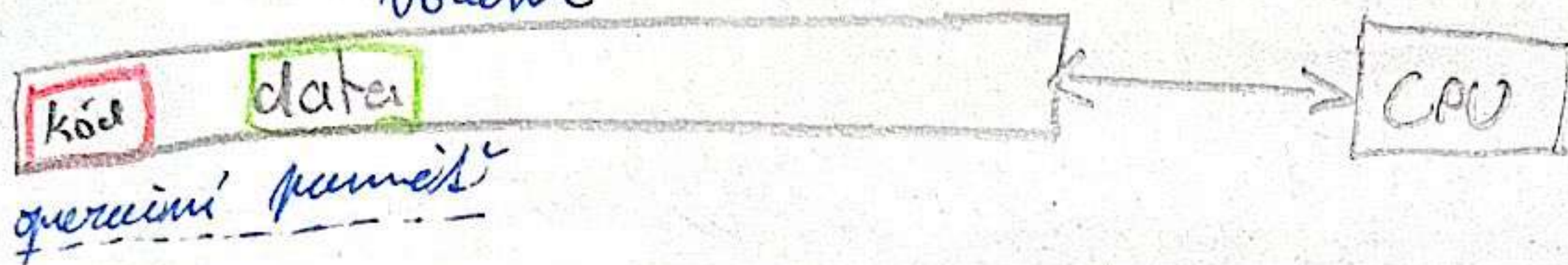


- ↳ general purpose input and output
- ↳ slouží pro vstup i výstup
- ↳ DIR registry určují směr pinů, IN a OUT jsou hodnoty na vstupu/výstupu
- ↳ 1-bit určuje směr ← takže se konfiguruje
- ↳ každý pin má 1 nebo 0 hodnot
- ↳ každý pin má 1 nebo 0 hodnot

Permanentní datové úložiště / Mass Storage Device

↳ stolní počítači → znova von Neumannova architektura

volatile → s větší kapacitou: DRAM



Perový disk / hard disk drive / HDD

↳ magnetický zápis, na paměť

neželezo, ale u neželeza máme elektřinu
atom, funguje jako dipól

↳ otáčí magnetickým polem

↳ ideální pro archivaci, protože magnet se nechce otáčet zpět

jelkým směrem dolevo, levý bit

$\begin{matrix} N \\ S \end{matrix} = 1$ $\begin{matrix} S \\ N \end{matrix} = 0$

→ atomový uspořádání na hranici

otáčí se

čím dál od středu, tím se sektory rozdělují
→ větší kapacita, protože to obklopuje číst

stopa / track

↳ rozděleny do kruhových výsečí

sektor / sector

↳ původně: 512 B

↳ dnes (advanced format):

4096 B = 4 kB

hlava

↳ sector addressable

↳ sektor je základní jednotka, co lze číst

↳ neustále se otáčí → otáčky: 10 000 - 15 000 rpm, normální: 7200 rpm, notebooky: 5400 rpm
→ rychlost přístupu

↳ skákal mezi stopami a čtení takhle je neapřiké

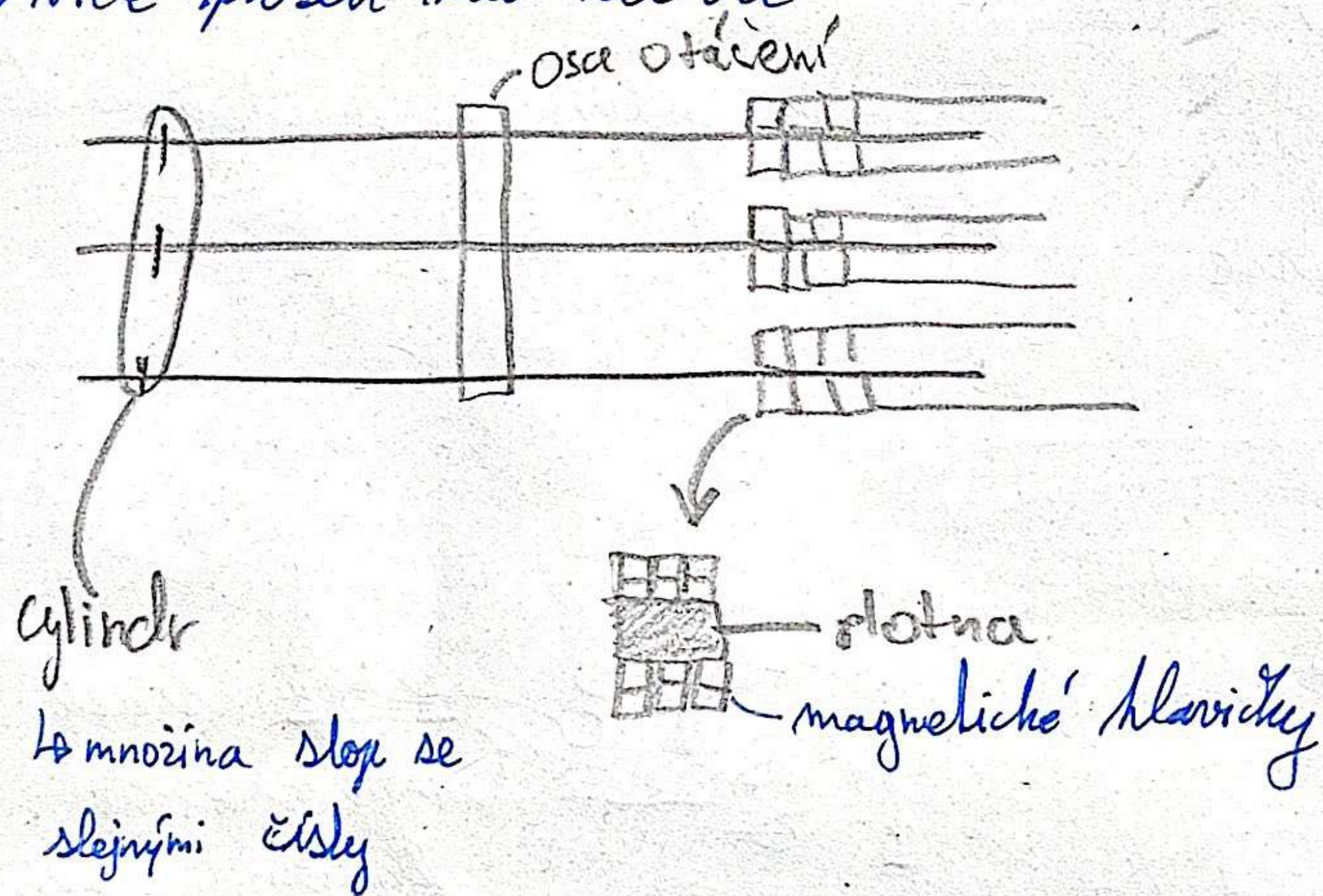
↳ náchylné k mechanickému poškození

↳ nerozliší se zápis $\rightarrow$ opravení ∞ zápisů

↳ čím blíže ke kraji, tím se disk otáčí rychleji a je tam více sektorů, takže přenosová rychlost je větší

↳ proto se nejčastěji zaplňují vnější stopy, takže takhle je adresování

↳ více ploten nad sebou



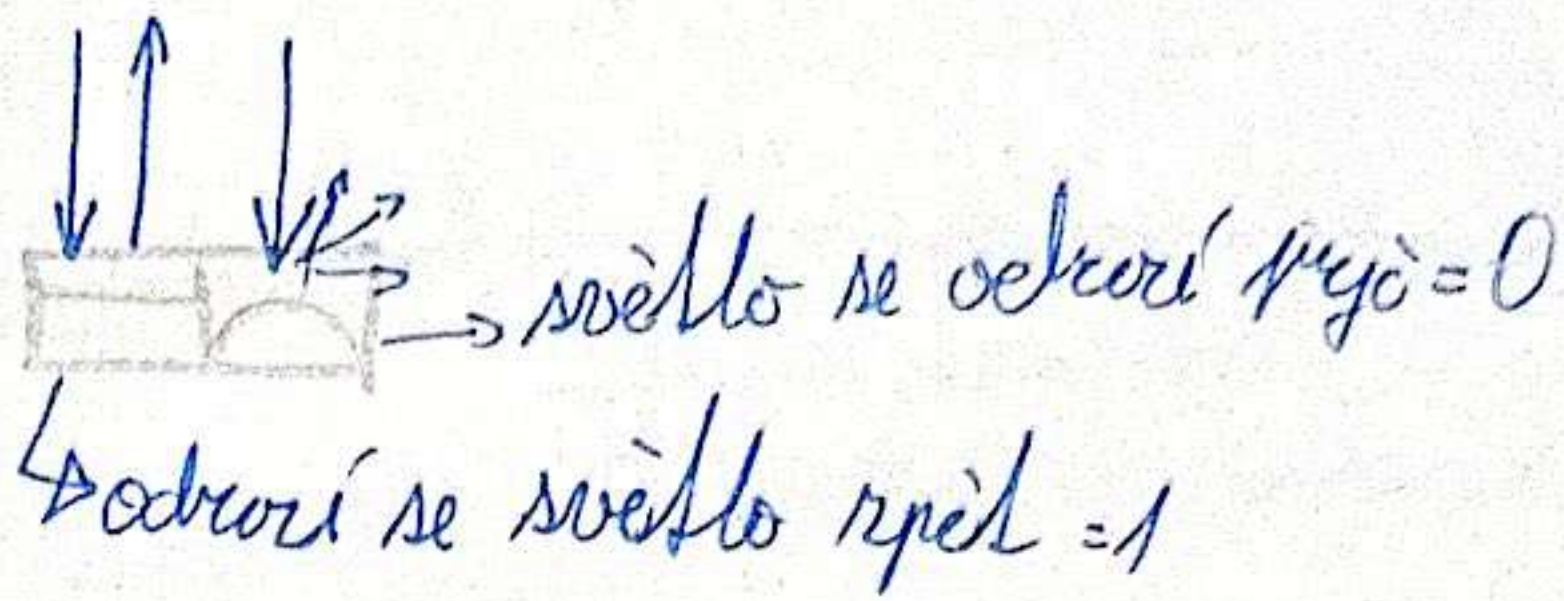
C/H/S

↳ cylinder / head / sector

↳ adresa sektoru

CD / DVD / BLURAY

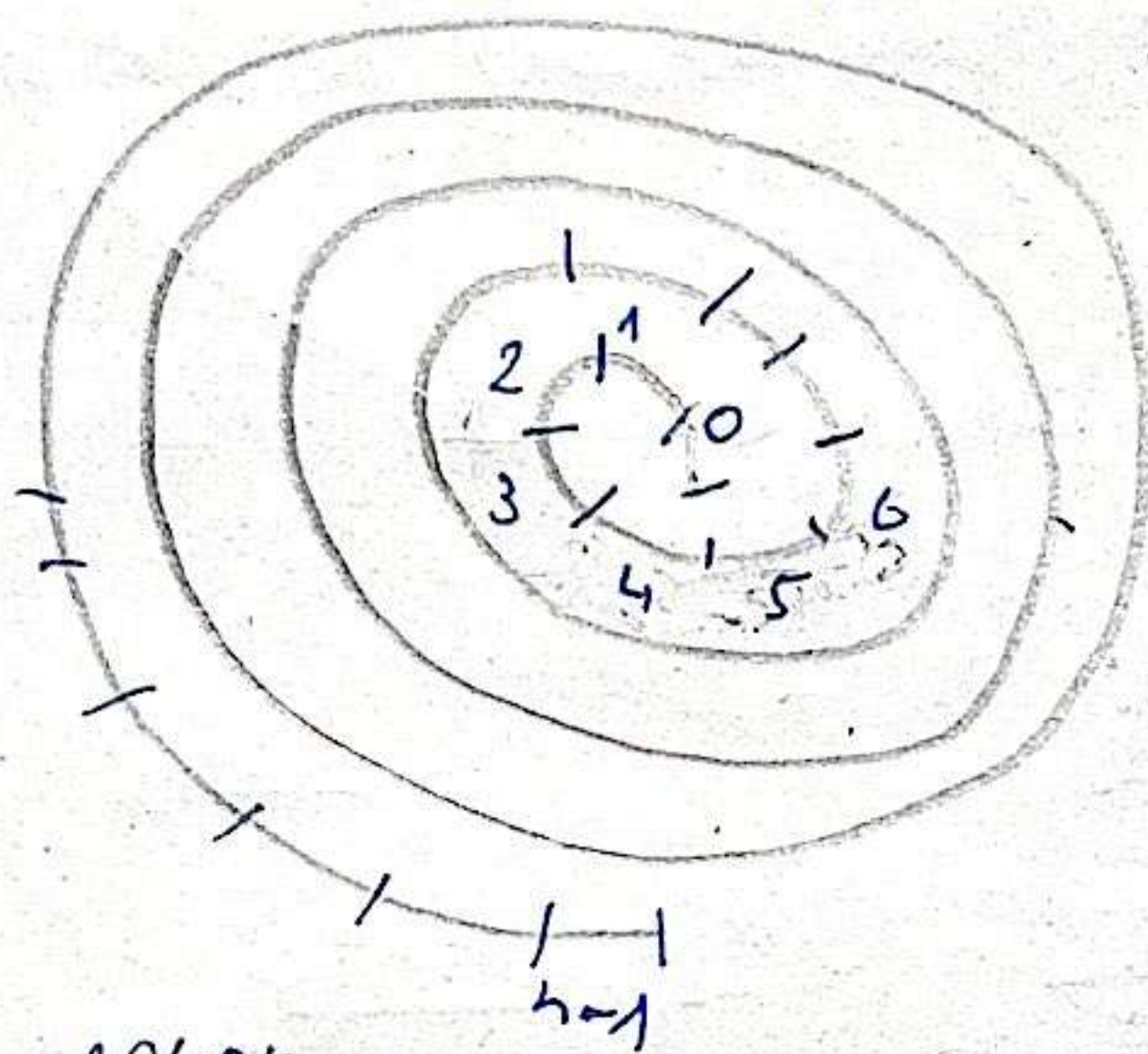
↳ na povrchu uloženo bity → je ve dvou stavech



↳ ochranný CD → materiál vyběhne

↳ vypalovací CD → jak vypálením ohnul, tak se ohne zpět po čase a data se ztratí

↳ ukládání do spirály → jak na gramofonové desce
→ rozdělena na sektory



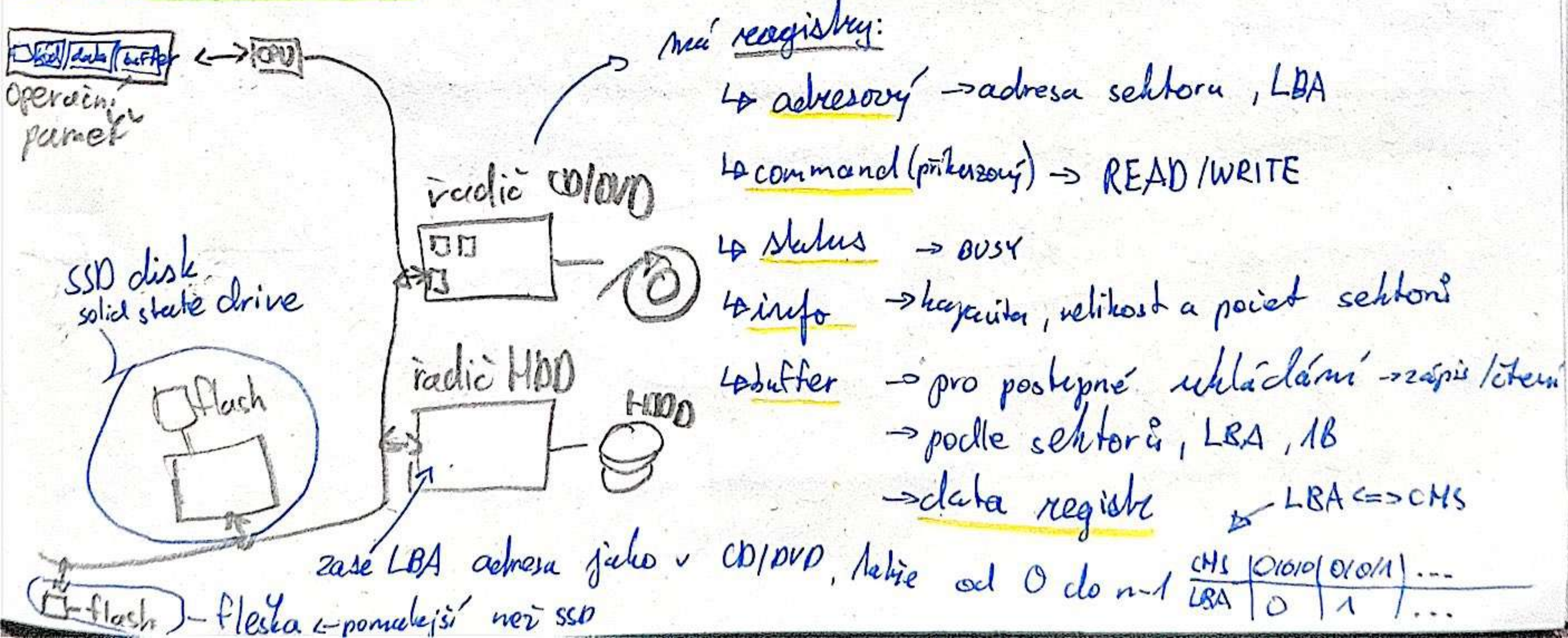
sektor → 2kB = 2048B

LBA → (linear block address)

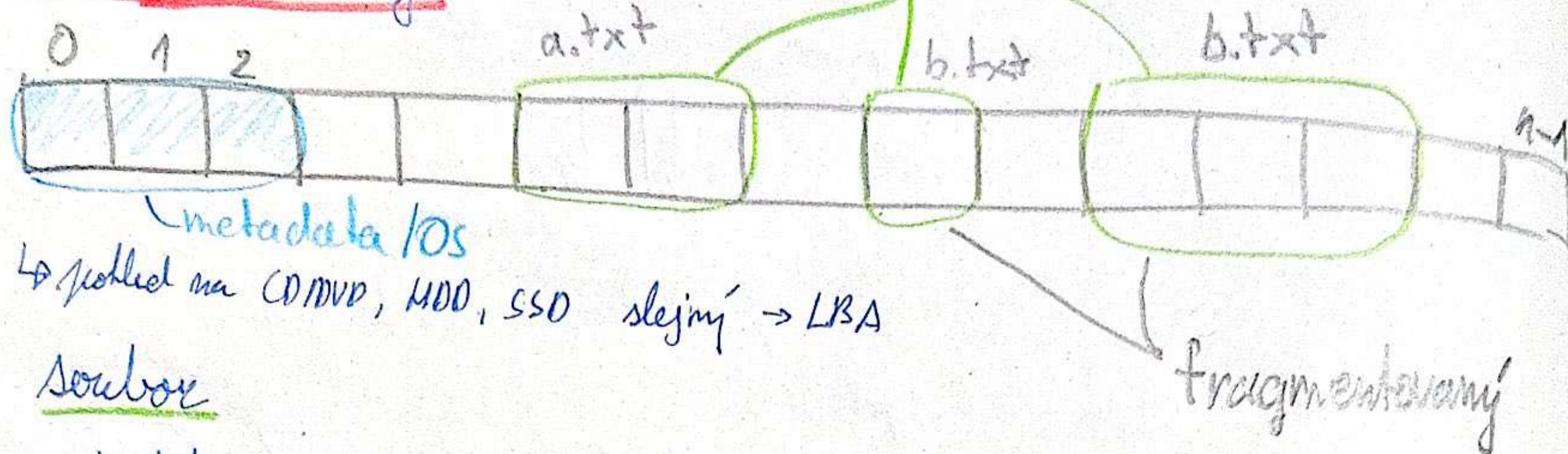
→ 1 adresa od 0 do $n-1$

↳ přenosová rychlost: ~10MB/s, přístupová doba: ~100ms

Řadiče pro úložiště



Adresování, soubory



↳ pohled na CD/DVD, HDD, SSD stejný → LBA

Soubory

↳ data

↳ metadata

→ data o jiných datech

→ mechanické - poměrně přesné

→ seznam sektorů - kvůli fragmentovanosti dat

↳ většinou se zaplní všechny až na poslední částecí

0 1 2 ... x-1

s_0 s_1 s_2 ...



→ délka v bytech

→ jméno souboru

path
/x/y/a.txt
jméno
adresář
= directory = folder = složka

→ datum a čas modifikace, oprávnění ...

→ volné sektory : seznam čísel

Souborový systém / file system

↳ FAT - linux

↳ NTFS - windows

↳ ext3 - macOS

} na interpretaci těch metadata v disku

↳ superblock → souborový systém → popisují, že disk je prázdný

soubor = posloupnost bytů

↳ proto každý byte dostane offset od začátku souboru → 0 až n-1

offset → v bytch
→ posun od začátku paměti

base address → odkud začínáme ⇒ čteme / píšeme

OS (operační systém) - operating system ⇒ Windows / Linux

`open("a.txt", "r")` ← aby nemuseli explicitně každý jazyk programovat přístup k souborům

↳ kernel (jádro) → základ, co každý operační systém musí mít, musí.

prací se soubory

→ takže např. Android je OS, ale používá Linuxový kernel

→ `ID = OPEN("a.txt")`
↑
`READ(offset v souboru, buffer)` } tohle používají program. jazyky

otevření je
efektivnější než
jen `READ("a.txt", ...)`

id otevřeného
souboru

= handle souboru

= file descriptor

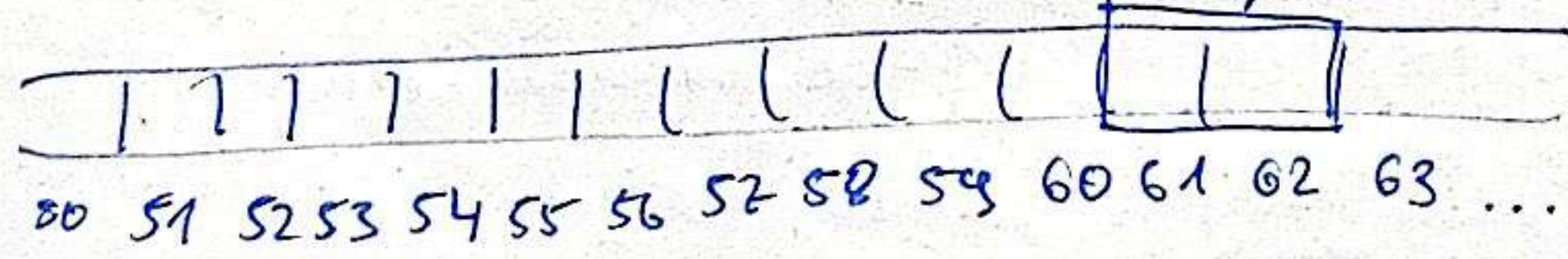
↖ syscall - volání operačního systému

↳ pak `CLOSE` pro
zavření } vlastně objekt v program. jazyce

↳ do cache se uloží, co je v tom souboru při zavolání `open`

↳
zarovnání dat
→

misaligned → vloženo hodnoty nedělí
číslu adresy



V Pythonu:

with open ("<path>", "<mode>", encoding="<encoding>") as f:

path → cesta k souboru

mode → podle toho, co chceme dělat

→ w - write

→ r - read

→ b - binary → rb
→ wb

encoding

↳ Windows - windows-1250 (encoding="windows-1250")

↳ Linux - UTF-8 (encoding="utf-8")

↳ funguje i ascii (encoding="ascii")

f.readline() - čtení jednoho řádku, jen v módu r a w

↖ vytvoř string a reprezentuje
byty jako str

f.read(<bytes>) → čtení určitého počtu bytů

→ u b mode vrátí byty ne string

f.seek(pos) → nastavení offsetu od začátku souboru

→ f.seek(pos, mode)

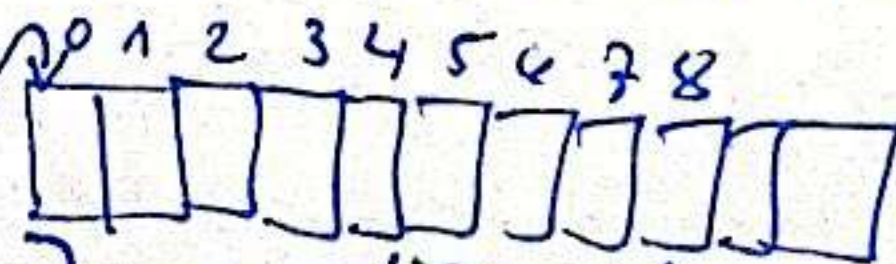
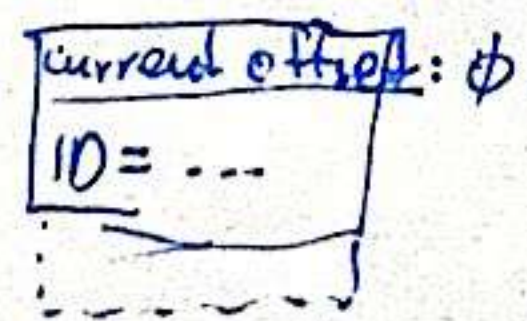
↳ mode = 0 - od začátku
1 - od aktuální pozice

2 - od konce

f = open("...", "rb")

f.read(5)

f.read(3)



↳ nový offset po f.read(5) a
před f.read(3)

`<str>.encode("<encoding>")` `str` $\rightarrow$ bytes

`<bytes>.decode("<encoding>")` bytes $\rightarrow$ str

bytes je immutable (nežije menit) $\rightarrow$ je potreba použiť byte array
`bytearray(<size>)`

$\rightarrow$ prevádzanie z bytes priamočiarne `bytearray(<bytes object>)`

hexdump (šestnásťkový výpis)

$\rightarrow$ 3 stĺpce $\rightarrow$ ① hex offset
první offset od začiatku súboru

② hex zápis
 $\rightarrow$ prostriední delí na stĺpce - offsety od začiatku
račku

③ interpretácia dát jako text (pokud)

$\rightarrow$ 1B = 2 znaky

Fileformat $\rightarrow$ znáť formát súboru

Synthesator $\rightarrow$ s súboru MIDI $\rightarrow$ musical instrumental digital
interface

T $\rightarrow$ signal. vercel. app

12. HODINA

foto = průměr za nějaký okamžik, aby se tam dostaly nějaké fotky

↳ za expozici dobu

↳ lidské oko průměruje co sarnamena světločivými buňkami → proto taky zobrazit v ploškách

1 ploška = 1 pixel.

x	0	1	2	3	4
y	0,0	1,0	2,0	3,0	4,0
1	0,1	1,1	2,1	3,1	4,1
2	0,2	1,2	2,2	3,2	4,2
3	0,3	1,3	2,3	3,3	4,3

↳ bítmapa

↳ ukládání do souboru → do posloupnosti bytů

0,0	1,0	2,0	3,0	4,0	0,1	1,1	2,1	...				
0	1	2	3	4	5	6	7	8	9	10	11	

base address

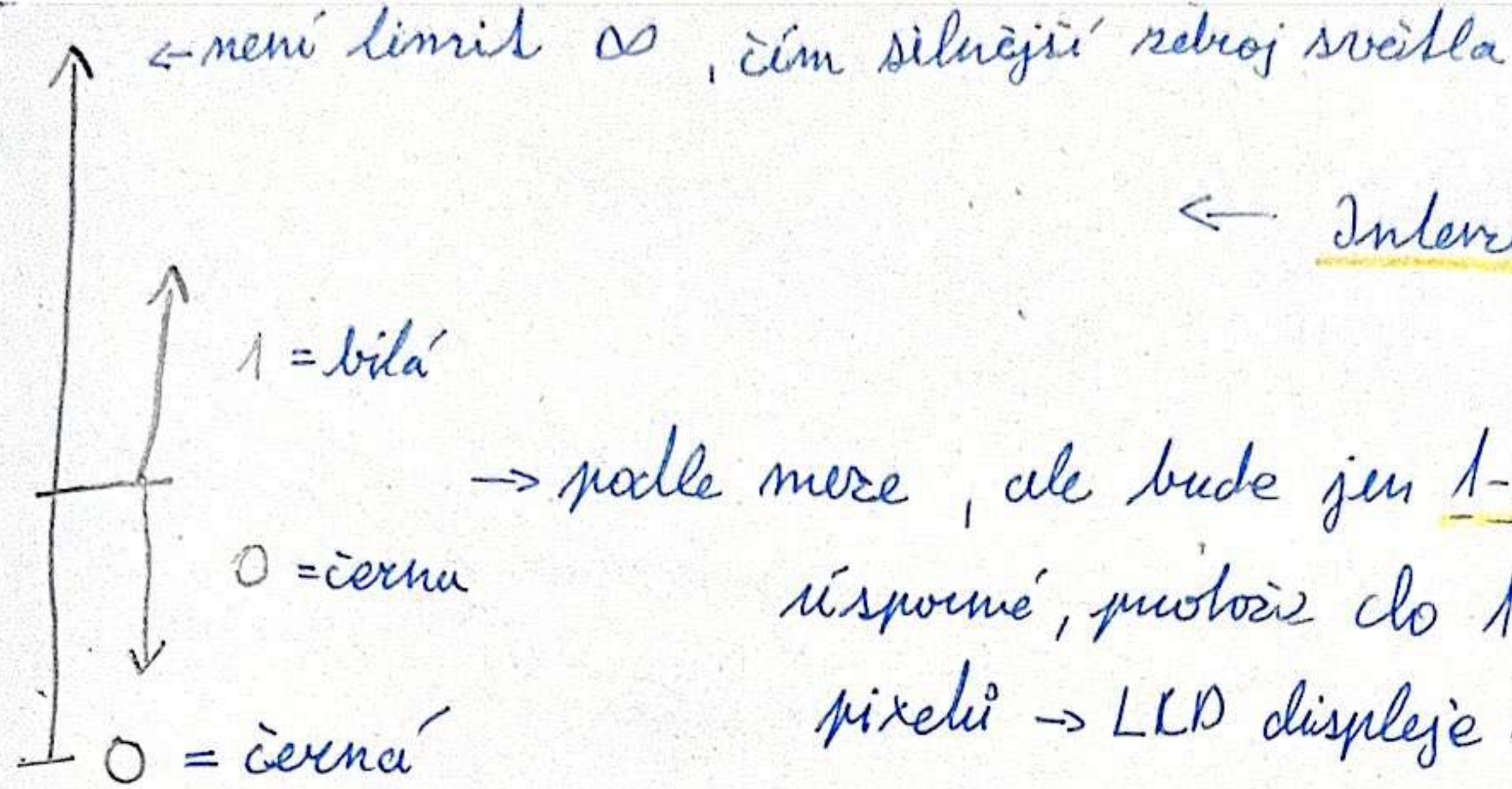
← offset od báze adresy

↳ ukládá se to po řádcích

↳ musí se dohodnout kolik bytů na 1 pixel → třeba offsety v vzorcích

↳ bits per pixel = bpp = bit depth (bitová hloubka)

↳ kolik má pixel bitů

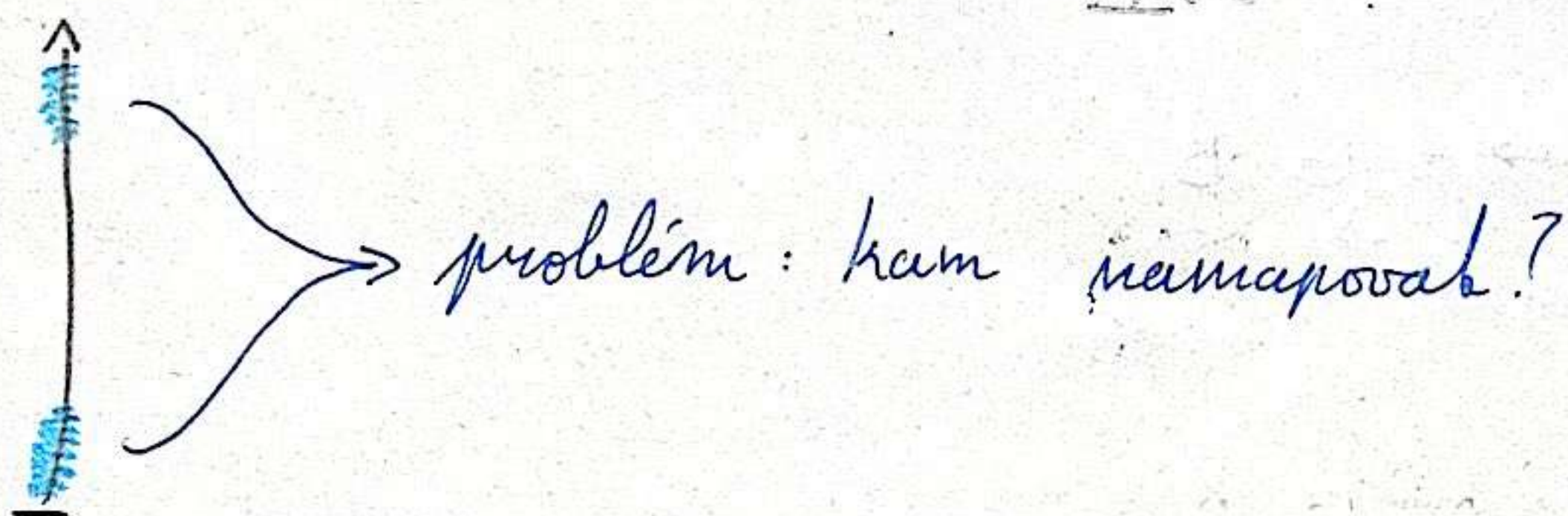
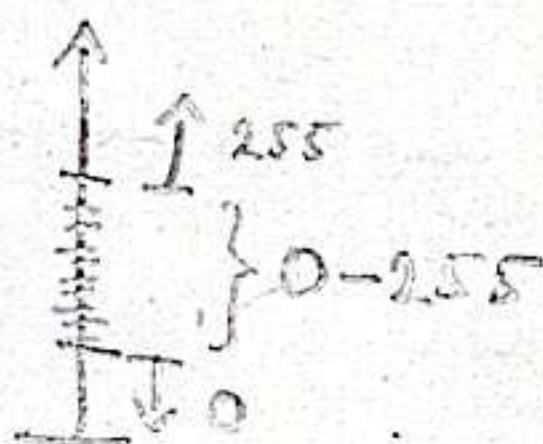


← Intervala světla

→ podle měze, ale bude jen 1-bit hloubka a tedy úsporné, protože do 1 bytu se vejde 8 pixelů → LED displeje v metru

8-bit hloubka

↳ celý byte na pixel

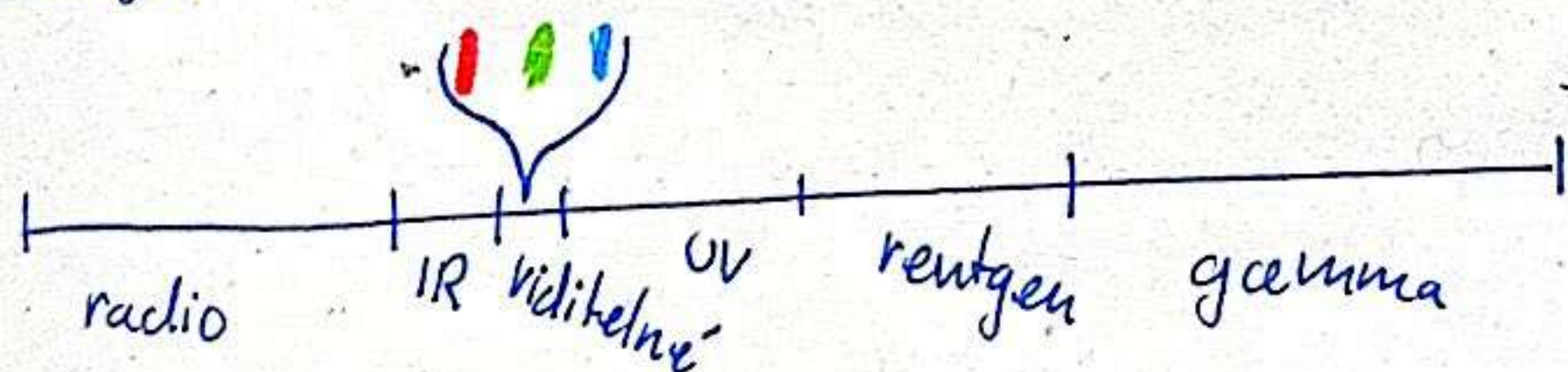


↳ ani velký rozsah nejde → protože velké čísla a vytvořit se něco skoro 1-bitového

↳ pokud použijeme floating místo unsigned → HDR

↳ ale toho to nemůžeme dále zpracovávat

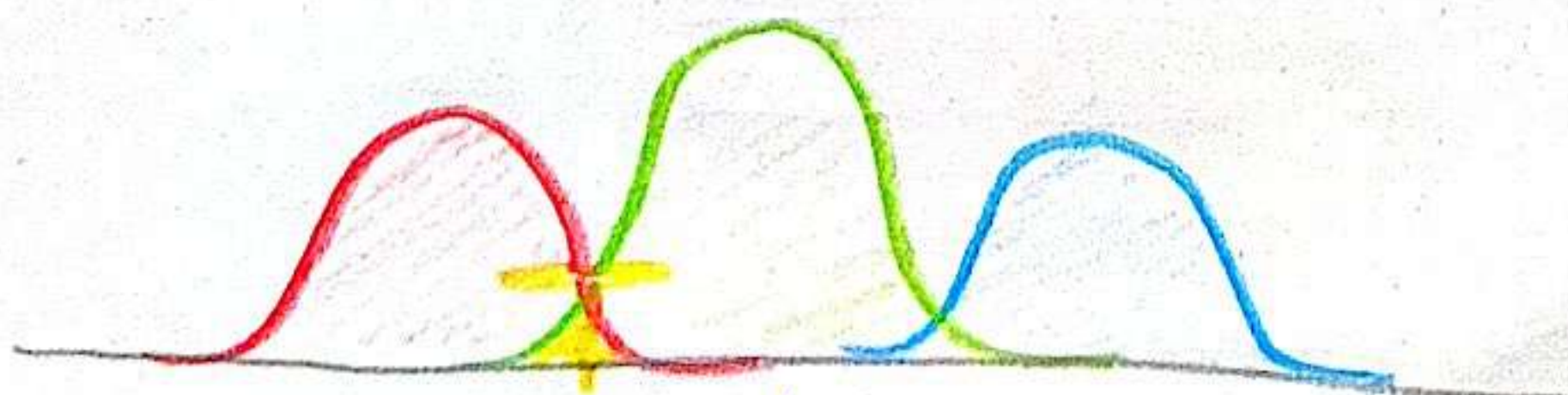
Barvy



↳ frekvence určuje barvu

Intenzity = rozsah

↳ čísly v oku (nejvíce se aktivují)
→ červené, zelené, modré
↓
nejvíce



↳ uloženy 3 složky (channels) → RGB

barvná bit depth:

a) 3b na pixel $\rightarrow$ $\begin{matrix} R & G & B \\ 1b & 1b & 1b \end{matrix}$ $\rightarrow$ neúsporné

b) 4b na pixel $\rightarrow$ $\begin{matrix} R & G & B & I \\ 1b & 1b & 1b & 1b \end{matrix}$ $\rightarrow$ I je intenzita $\begin{cases} 1 \rightarrow 100\% \text{ barvy} \\ 0 \rightarrow 50\% \text{ barvy} \end{cases}$
 $\rightarrow$ víc detailů

* - existuje color wheel $\rightarrow$ jak se skládá:

$\hookrightarrow$ bílá = všechny kanály na maximum

$\hookrightarrow$ černá = všechny kanály 0

c) 15 b na pixel 2B $\rightarrow$ $\begin{matrix} R & G & B \\ 5b & 5b & 5b \end{matrix}$
(high-color)

$\hookrightarrow$ 16 b na pixel 2B $\rightarrow$ $\begin{matrix} R & G & B \\ 5b & 6b & 5b \end{matrix}$ $\leftarrow$ bit navíc pro retention

$\hookrightarrow$ 16 b na pixel 2B $\rightarrow$ $\begin{matrix} R & G & B & A \\ 5b & 5b & 5b & 1b \end{matrix}$ $\rightarrow$ spis ARGB
 $\leftarrow$ bit navíc pro alpha kanál
 $\leftarrow 1 = \text{nepřehledné}$
 $\leftarrow 0 = \text{přehledné}$

d) 24-bit hloubka $\rightarrow$ $\begin{matrix} R & G & B \\ 8b & 8b & 8b \end{matrix}$ $\leftarrow$ 65553 barev
(true color)

$\hookrightarrow$ do 32-bit (4B) se používá na alpha kanál $\leftarrow$ 32-bit kvůli moderním CPU

$\hookrightarrow$ $\begin{matrix} A & R & G & B \\ 8b & 8b & 8b & 8b \end{matrix}$ $\rightarrow$ BE: $\boxed{A|R|G|B}$
LE: $\boxed{B|G|R|A}$ - běžnější

$\hookrightarrow$ specifikovat autor obrázku

Rastrové obrázky x Vektorové obrázky

= bitmapa

↳ data jsou porovnáváni

↳ musí se rasterizovat pro zobrazení

PNG (beztrátová komprese)

BMP, TGA

(bez komprese)

JPG (ztrátová komprese)

Uložení do paměti

↳ data → rastrový obrázek / bitmapa

↳ metadata (obrázku, ne soubor)

↳ bitová hloubka

+ počet kanálů a uspořádání bitů na kanál

↳ šířka v pixelech

↳ výška v pixelech

↳ offset, kde začínají data

↳ EXIF (pokročilejší) = metadata nepovinná

↳ datum / čas

↳ expozice

↳ GPS souřadnice

→ 1 soubor
v FS (file
systemu),
který má
vlastní
metadata
↳ jméno,
sektory

Formát souboru

hlavička data

BMP

↳ používal ve Windows

↳ shranka z bitmapy

↳ jednoduchý, nekomprimovaný

↳ data jsou v LE → třeba obrázek má 0,0 pixel vlevo dole

↳ první 2B → signature

↳ identifikátor souboru

↳ každý řádek obrázku má offset dělitelným 4 nebo 8, protože používá 3 kanály RGB → 3B, proto je tu navíc nějaký padding pro zarovnání

TKinter → zobrazování pixelů, knihovna v Pythonu

Textová data

↳ textový řetězec /string/ str

↳ je to posloupnost znaků → grafémy - výsledný obrázek A a

↳ znaky:

↳ písmeno

↳ číslice

↳ interpunkce → . , @

↳ bílý znak /whitespace → $\lfloor$, nic se nezobrazí, mezera, tab

↳ řídicí znak → CR LF

↳ kódování /encoding

① ↳ přiřazení každému znaku kód 1 znak → 1 kód

② ↳ mapují na binovou reprezentaci (ne všechny kódování to dělají)

ASCII → 7-bit (kvůli 7 bit pozicím letěl)

↳ 0 — 127

↳ American Standard Code for Information Exchange

↳ A, B, ..., Z, ..., a, b, ..., z, ..., 0, 1, ..., 9, ..., $\lfloor$

$\begin{matrix} \text{"} & \text{"} & & \text{"} & \text{"} & & \text{"} & \text{"} & & \text{"} & \text{"} & & \text{"} \\ x & x+1 & & x+25 & y & y+1 & & y+25 & z & z+1 & & z+9 & \end{matrix}$

32 dec
20 hex
mezera

↳ číslice i písmena jsou blízko sebe

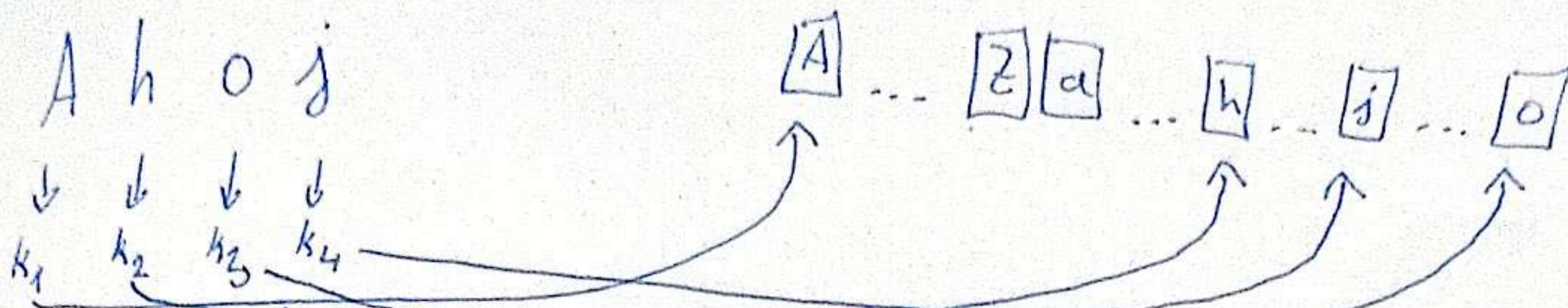
↳ extended → vytvoří se kódová stránka (code page) a další kódům

↳ každá část Evropy si to rozšířila podle sebe

↳ W/M/E

↳ ISO 8859-2 (Latin 2) → pokus o standardizaci, ale pozdě

DOS LATIN 2
Windows 1250 - u Windows



normalizace textu

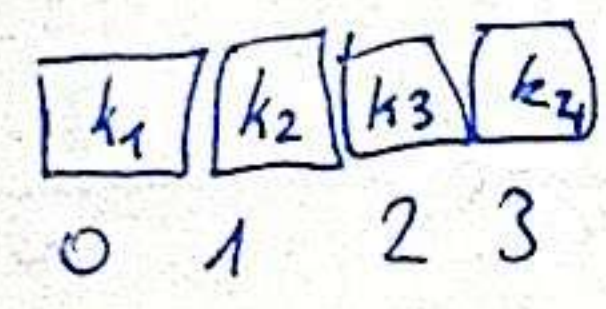
→ pro každý znak vytvořil své bitmapy podle různých fontů

→ text, který se ukládá v počítači v jím se čtou

Ahoj

arabské

IBM



UNICODE 0 - \$10FFFF

- standardizace pro všechny jazyky, všechny symboly \$D800 - \$DFFF nikdy nepřidá
→ pro techniky
- kde 0 - 127 kódy z ASCII, 128 - \$FFFF běžné znaky, \$10000 - \$10FFFF
- ale u kódování definují pouze 1. krok → neběžné

Unicode kódování

→ mezinárodní definují binární reprezentaci

→ UTF-32 → 32-bitové

→ 4B = 1 znak

→ UTF-32 BE a UTF-32 LE

→ neúspěšné

→ UCS-2

→ pouze 2B

→ UTF-16

→ proměnlivá délka 2B / 4B

→ také LE / BE

→ UTF-8 → 1B / 2B / 3B / 4B znaky

→ pro rozpoznání se používá 2x surrogate a pomocí algoritmu na znaky

populární 1B - první bit 0

nB → první n bitů 1, pak 0
→ major. n=2 → 110

\$D800 - \$DFFF

↳ při naslovování textu

↳ kreslení na nový řádek

↳ umístění klávesnice, aby písmena byla co nejdale od sebe a typewriter se nemusel hávat

↳ CR (carriage return) - vrátilo zpět 13 \$0D

↳ LF (line feed) - nový řádek 10 \$0A

Windows -> CRLF

Unix -> LF (pro uživatele)

MacOS -> CR, dnes už LF jak v Unixu

} readline() } funguje pro všechny input()

↓
ale např. Notepad ve Windowsu funguje jen CRLF

- u print() - podle operačního systému

Unicode

LS = line separator

PS = paragraph separator

} indikal větší bloky



→ postupnost bytů v jehkovém kódování
↳ textový soubor



→ postupnost bytů, kde jsou různé kódování
↳ binární soubor

↳ často řečeno velikost textu

2. bytes() → encoding
str.encode(" ") → do bytu
↳ jsou immutable
↳ jsou mutable
bytearray = mutable

13. HODINA

↳ řadič paměti → mezi pamětí a CPU
→ lze snadno namapovat 2 moduly 1GB a
0,5GB jednu 1,5GB paměť, kde jsou 32-bit/33-bit
adresový prostor a u řadič paměti např. jen 32-bit

↳ host controller interface (HCI) → komunikační protokol mezi
hostem (CPU) a řadičem

↳ systémová sběrnice → sběrnice od CPU, připojen řadič paměti i
ke ostatnímu zařízení

→ např. PCI Express (PCIe)

PCI Express (PCIe) → multimaster

↳ sériová

↳ Full-duplex

↳ multibroadcast

↳ 2 druhy pakety

1) adresace zařízení → každé zařízení stanoví svou adresu

2) adresace v počítačovém adresovém prostoru

↳ pakety → memory write MW

→ memory read MRd

většina mobilní počítače ... jsou multiprocesorové

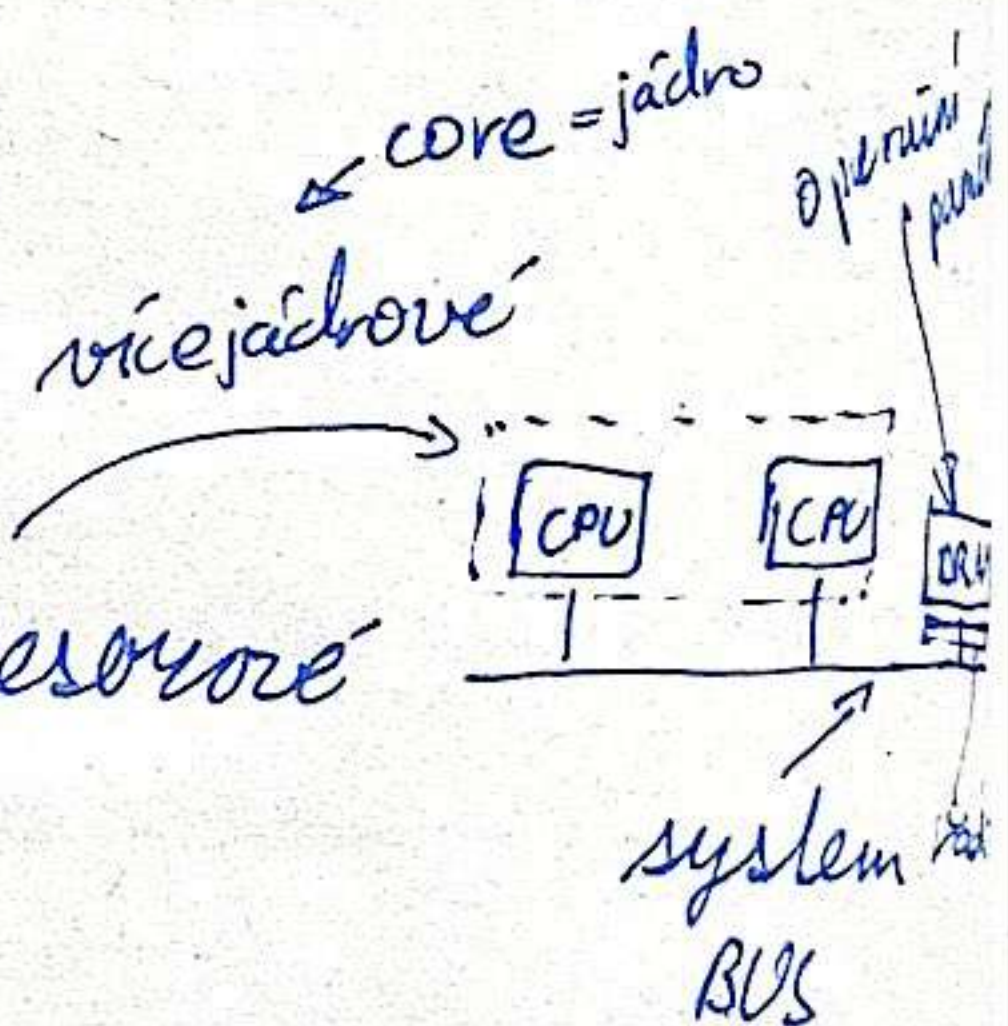
↳ posílají pakety s adresami

↳ příklady: CPU pošle MRd (Memory read packet)

↳ cílová adresa je adresa paměti

↳ musí sem být adresa procesoru, aby mohla přijít paket zpět

Memory Controller výkoná požadavek, pošle CxID (completion with data) → cílová adresa je adresa procesoru, aby to mohla přijmout



↳ to by nevyžít adresový prostor CPU pro paměť vyžít na adresy zařízení (registry)

↳ offsety od nějaké básové adresy
pak by stačilo poslat jen memory write paket

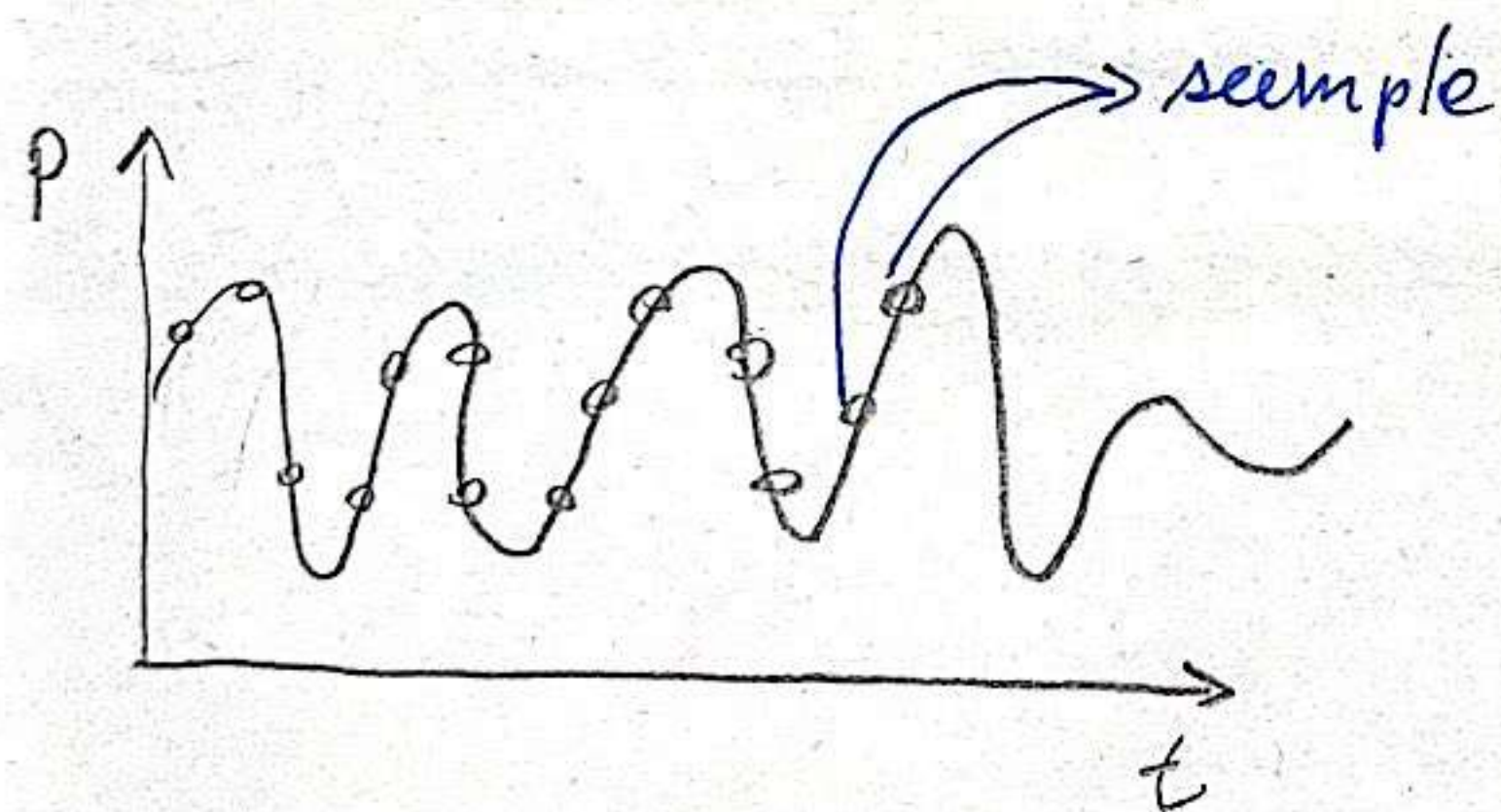
Memory Mapped I/O (MMIO)

↳ přímá komunikace se zařízeními je skrz ten jeden adresový prostor → pomocí LOAD a STORE instrukcí, netřeba speciální

Reprezentace zvuku v počítači

↳ ne MIDI soubory → ty jsou zápisem not

↳ změna tlaku v měřiču = zvuk a uložení průběhu tlaku



→ uchovávat si jen vzorky z průběhu

→ snímají se +/- hodnoty z tlaku od default tlaku → znaménko své číslo

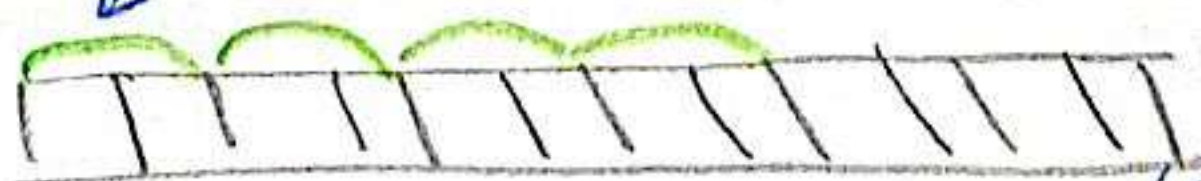
8-bit (vážimou málo)

16-bit (CD kvalita)

24-bit

↓
nemapuje se do rozsahu

1 vzorek 16-bit = 2B



↳ ukládají se v pořadí, ve kterém se přichází = sekvence

sampling rate = vzorkovací frekvence

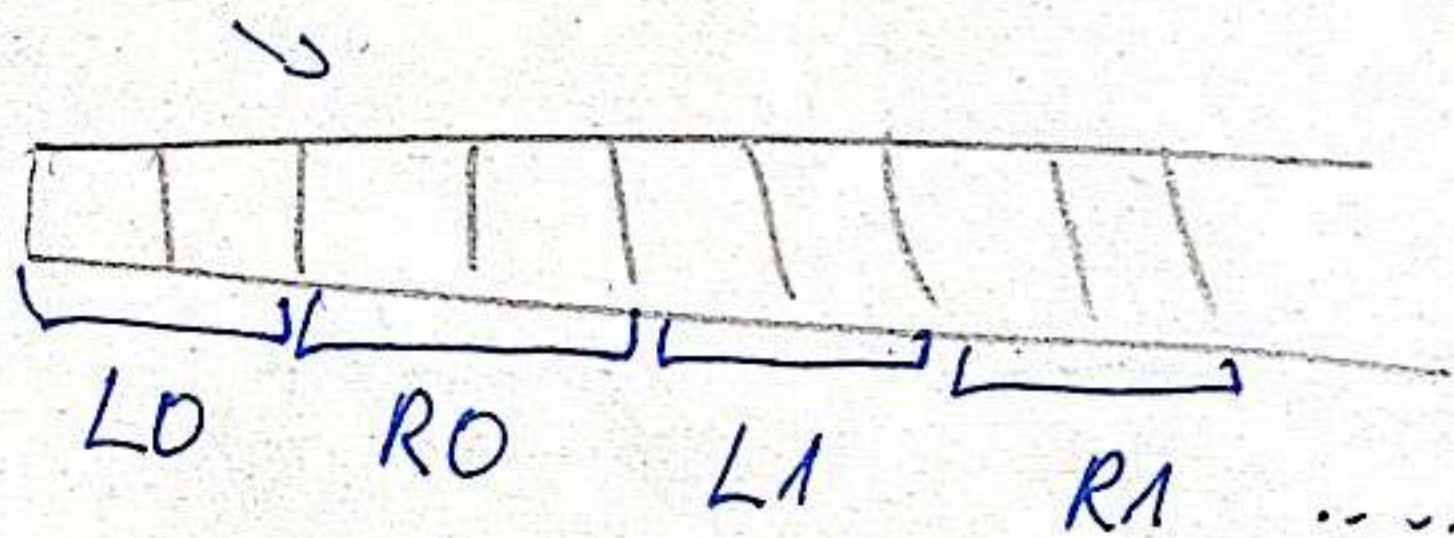
↳ kolik vzorků za sekundu bude snímáno

22 kHz — 44,1 kHz — 96 kHz

1 kanál = mono zvuk

2 kanály = stereo zvuk → 2 nezávislé proudy dat

↳ stereo → budou protočeny v párech



zvuková karta

↳ řadič

↳ pravidelně budeme zapisovat do registru, co máme uloženo jako zvuk, co bude měnit napětí a produkovat zvuk

LOAD ← sample

STORE → data registru

WAIT → $1/\text{sampling rate}$ (potřeba dělat paralelně s jinou logikou počítače)

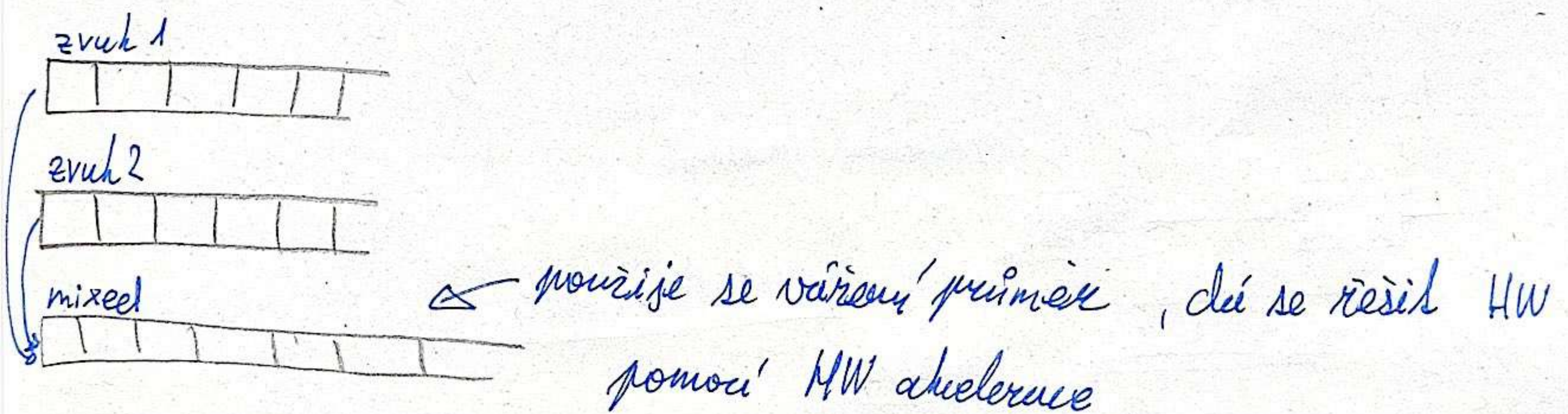
↳ aby se nemusela nastavovat logika hry nebo něco podobného, se to máče do bufferu a tam je auto-incrementace, která postupně posílá data do DAC (digital to analog converter)

↳ aby se to nepřehrávalo dohola, i když už nemá, tak tam je ještě .config register → mono / stereo, sampling rate, sampling size
• control register → stop, play ...
• status register

↳ v 16-bit registru → potřeba myslet na endianitu

↳ HW akcelerace

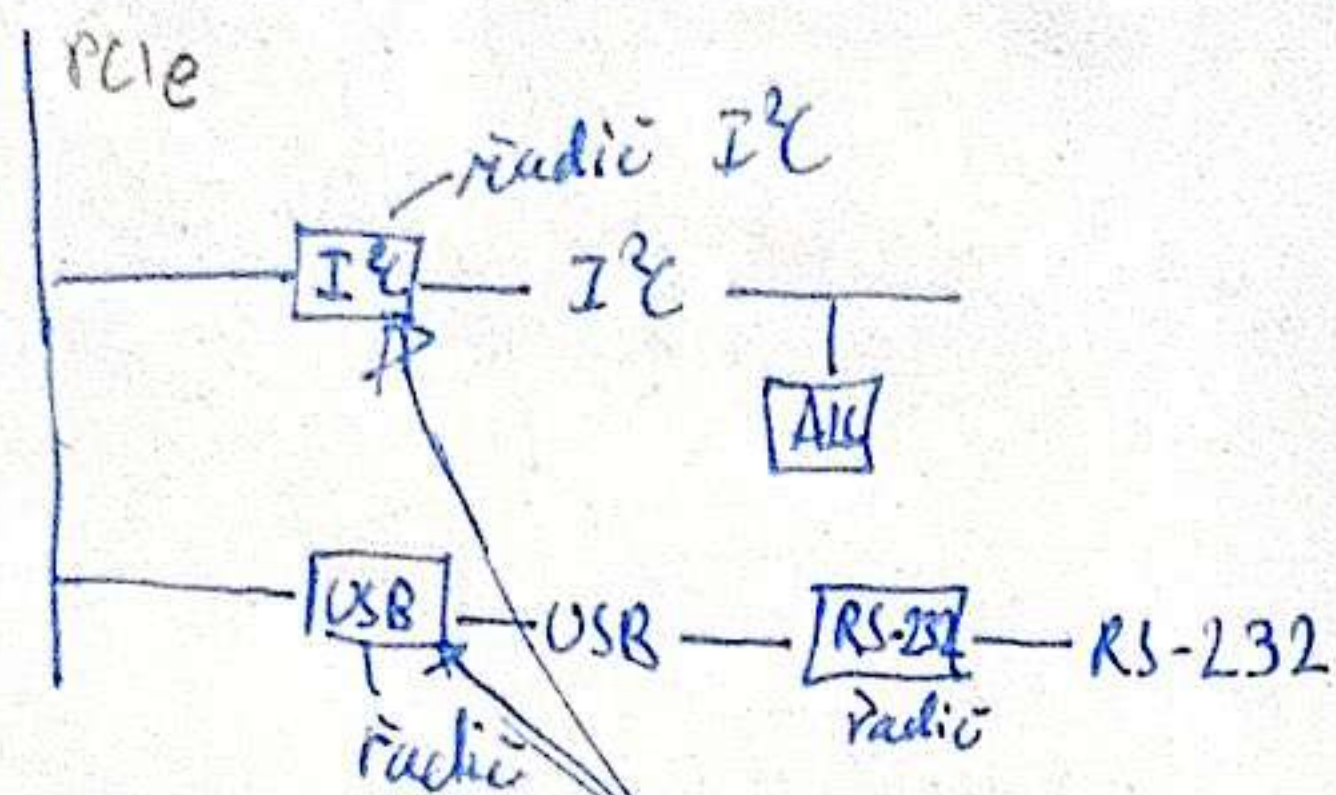
↳ součást procesorů



DSP procesor (digital signal)

↳ vlastní jednoúčelový procesor, kde lze nahrát zvuk a strojový kód a bude to dělat ten procesor

CPU



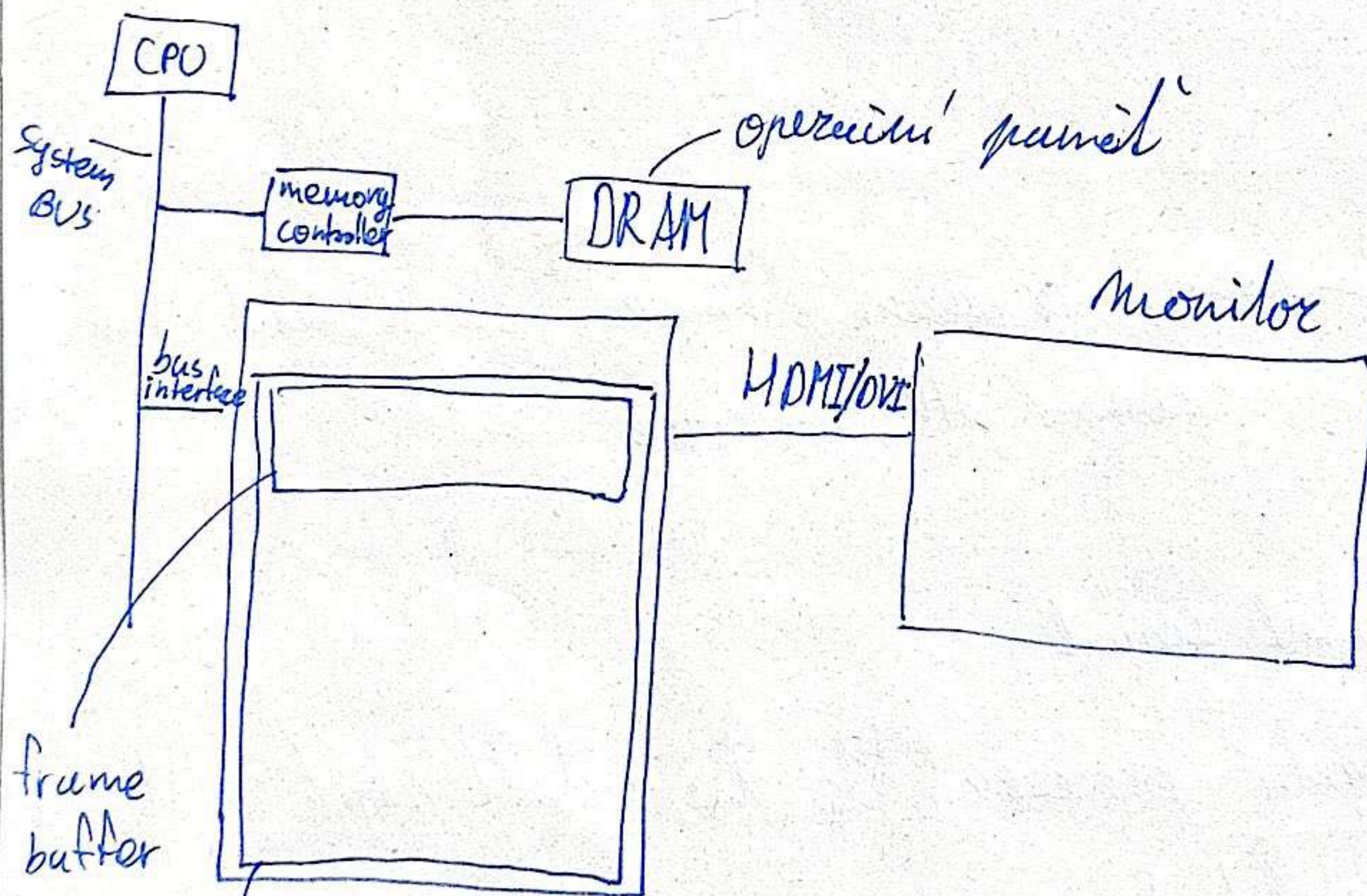
I2C rādīt

Host bus adapter (HBA)

↳ šeit mums ir jākontrolē

polling

grafiskā karte



↓
reprezentācija
ko ir zīmēta
mā ekrānā

memorija DRAM → "video mem"

↳ šeit ir funkcija

↳ pārņemšana no DRAM buffer a pārņemšana, ko se rādīt

↳ můžeme rovnou měnit bitmapu, kdy část frame bufferu se namapuje na nevyužitý adresový prostor

↳ HW akcelerace

1) Textový režim → ukládám jen text a grafická karta učelá rasterizací

2) 2D grafiky → popíšeme, kde nakreslit čáry

3) 3D grafiky → mesh, texture → mohou být velké, proto v graf. kartách jsou GB paměti
→ push rasterizace do 2D grafiky
→ shader

↳ firmware

↳ počítačový program od výrobce, uložen v ROM

↳ CPU → má startup vector nebo-li adresu, co se přičte do instruction pointeru

↳ $x86: EIP \leftarrow \text{FFFF FFFF}$ - např.

↳ 1) test a konfigurace HW → prostě dá počítačové hodnoty do těch komponent a namapuje

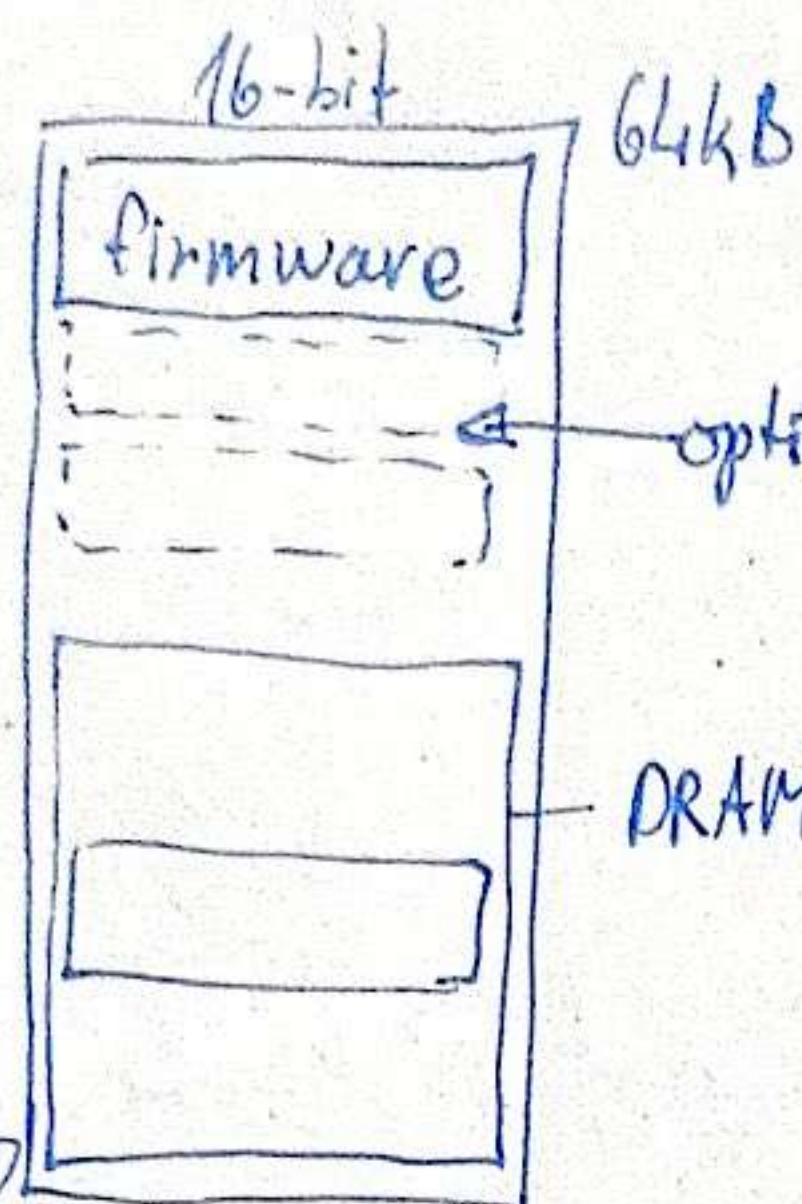
base adresy → pomocí konfiguračních pinů

→ plug & play → nastení si vše samo

2) najít vhodné software →

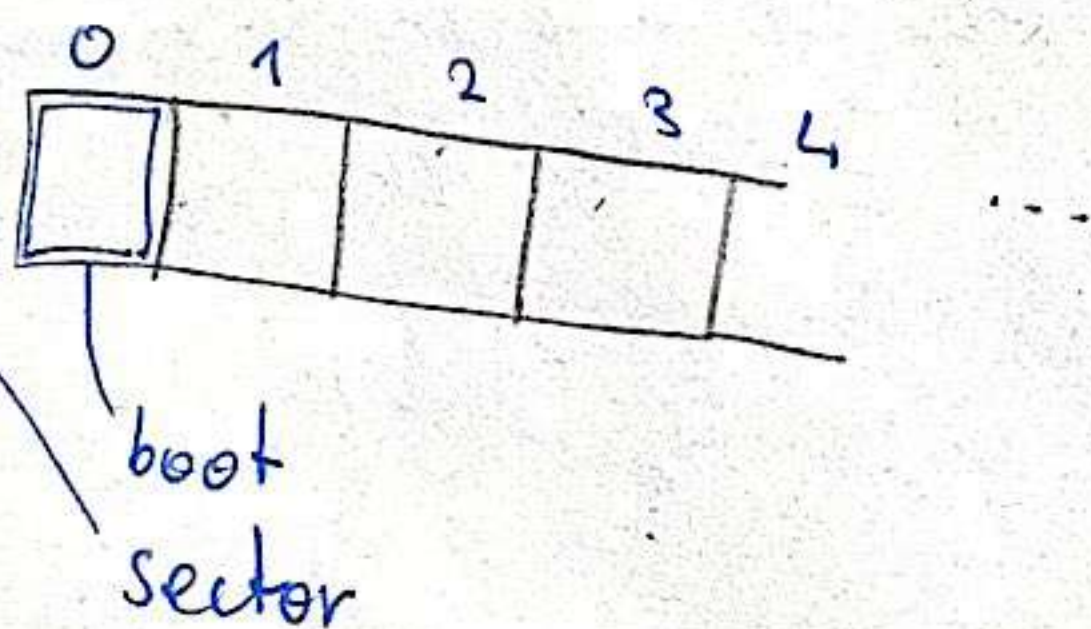
bootování

↳ option ROM →



↳ daly se použít cartridge → kde nahrané ROM

↳ na starých počítačích → předinstalován BASIC interpreter



↳ pokud to je v RAM, tak to bude bížet

bootloader

↳ ide jednotlivé sektory a pak to ručne bížet potom co nahraje do RAM ky

↳ musí umět komunikovat s MASS storage device

funkce firmware - pokr. minulé stránka

3) abstrakce nad HW

↳ člení sektorů

↳ zobrazení textu

↳ člení z klávesnice

↳ bootlooder má podstatný program

a) užitečný program

b) kernel operačního systému

i) má mít užitečnější program

ii) abstrakce nad HW

1) abstrakce nad souborovým systémem
openfile/readfile

2) zobrazení textu / grafiky

3) klávesy / myš / touch

c) abstrakce nad operační pamětí

→ má mít a spustit

↳ Shell → grafický shell např. explorer.exe

↳ v Activity monitor

KONEC ZS 25 / 26
