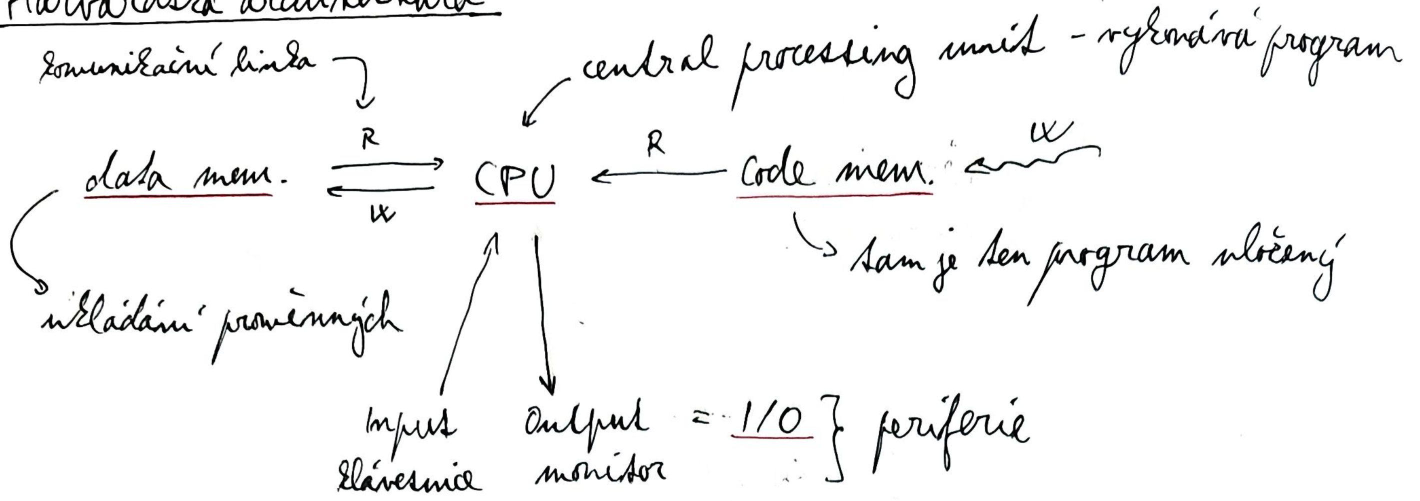
• Harvardská architektura



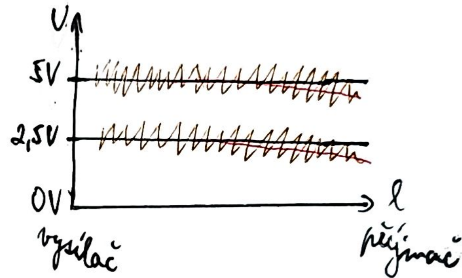
• Reprezentace nezáporných celých čísel

• Analogový přenos dat

→ chceme 0, 1, ... 1000

→ zvolíme mezní U: 0 = 0V, 1000 = 5V

↳ 500 = 2,5V



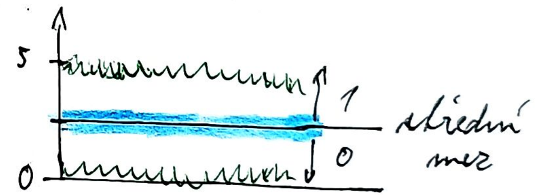
→ vodiče mají odpor, ten se mění s teplotou

→ elmg. indukce: vodiči něco proudí → elmg. pole → na dalších vodičích se indukuje napětí → šum

→ na přijímači víme jen přibližné hodnoty na vysílači → není to deterministické

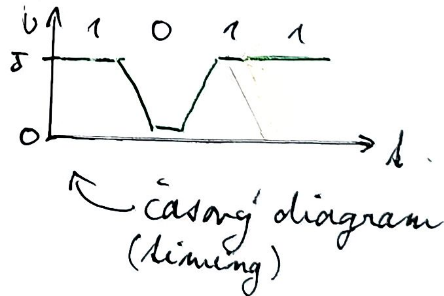
• Digitální přenos dat - binary digits = bits = b

- 0V = 0, 5V = 1
- uprostřed je nějaká šedá zóna - nad ní 1, pod ní 0
- počítáme s rozumným šumem

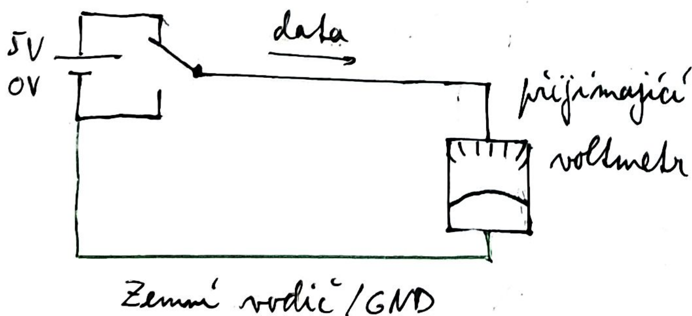


• Sériový digitální přenos

- posílání větších binárních čísel
- přijímající musí to napětí nějak měřit



vysílající - volí 5V/0V

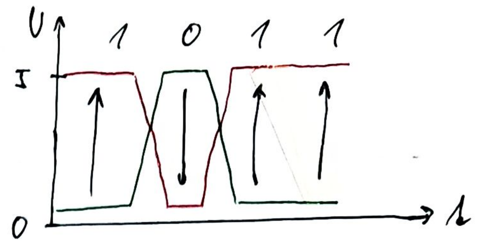


↳ U se měří mezi 2 body

↳ potřebujeme referenční vodič

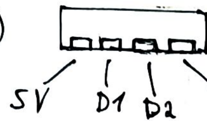
Přenos dat probíhá po komunikační lince

- Diferenciální přenos → diferenciální pár



→ alternativně máme dva datové vodiče
  - data 1 - přenos dat
  - data 2 - tam se generuje opačné napětí
→ napětí neměříme vůči zemnímu vodiči, ale vůči sobě ⇒ přijímající vidí data 1 - data 2 : $>0 \Rightarrow 1$, $<0 \Rightarrow 0$ + nějaká šedá zóna
→ 0-5V ⇒ 1, -5-0V ⇒ 0 ⇒ zvýšila se úroveň šumu, kterou sneseme
→ na obou vodičích vzniká víceméně stejný šum, který se odečte
→ dneska nejběžnější způsob přenosu

- USB (Universal Serial Bus)

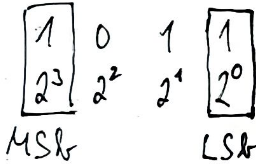


↗ 5V + GND slouží k napájení
D1 + D2 dif. přenos dat

- Přenos binárních čísel

- $2^{16} = 65536$ • $2^{20} \approx 10^6$ • $2^{24} \approx 16 \cdot 10^6$ • $2^{32} \approx 4{,}2 \cdot 10^9$

→ je třeba se dohodnout, kterým směrem bity čteme



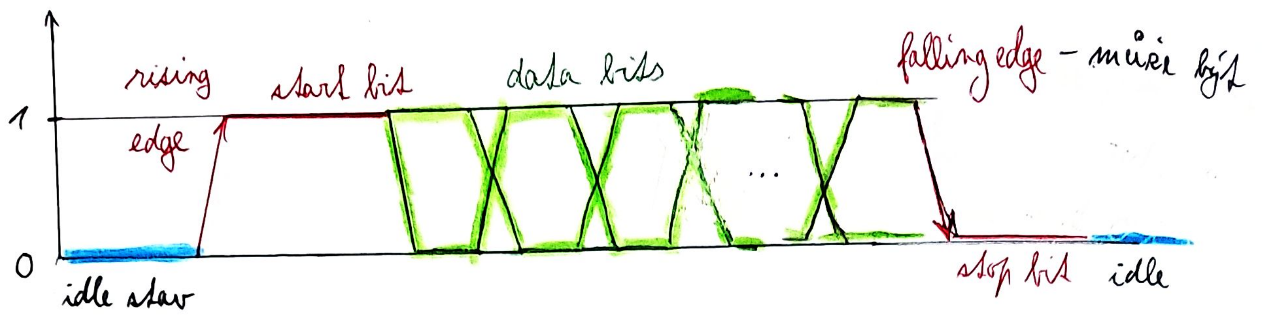
← MSb - first   1101 ← LSb - first
$2^0 2^1 2^2 2^3$

→ je třeba dohodnout pevnou délku bitu - jak dlouho trvá signál 0 / 1
→ definuje se to přenosovou rychlostí ↘ a 1
  - bity za sekundu - bps
  - symboly za sekundu - baud

} pro uvedené případy platí bps = baud

→ kdy má příjemce kontrolovat napětí?

- nejlepší to je uprostřed
→ problém: asynchronní hodiny - obě strany se musí shodnout kde je prostředek bitu
→ komunikační linka může být ve třech stavech - 0, 1, idle - nic se nepřenáší
  - floating stav = stav s vysokou impedancí / Hi-Z - ta linka není zapojená nikam ⇒ přenáší se jen šum
  - 3-stavová logika - 0 / 1 / idle definovaný jako floating stav.
  - 2-stavová logika - 0/1/idle := 0 nebo 1
    → když se linka zapne, tak vysílá idle (0) ⇒ příjemce ví, co je idle
    → začátek vysílání: start condition = rising edge (0→1)
    → než začne skutečný přenos, tak jsou start bity (1) - je třeba se domluvit kolik



→ na start condition se sesynchronizují hodiny – žádné hodiny nejsou dokonalé

⇒ za nějakou dobu se to zcela asynchronizuje

→ celkem přenášíme N bitů po X bitových kusech v N/X přenosech

⇒ dohodou X := 8b = 1byte = 1B

→ stop condition := když přeneseme právě X bitů

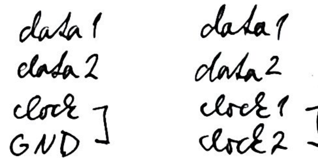
→ po stop condition jdeme do 0 do idle stavu = stop bity

⇒ nyní může začít nový přenos nebo linka zůstane v idle stavu

8 data bits
1 start bit
1 stop bit
} 20% overhead → 1000 baud = 800 bps
↳ datové bity

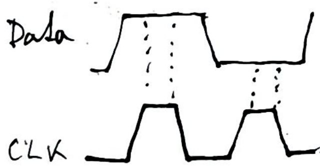
RS-232

- přenos pomocí hodinového signálu
  - stálý signál 010101010101...
  - potřebujeme alespoň 1 další vodič



data 1
data 2
clock 1
clock 2

  - hodinový signál buď může generovat vysílající nebo nějaké externí zařízení



- když je na clk 1, tak máme měřit
- harwarově je snadnější detekovat hrany
  - rising edge ⇒ měříme
  - falling edge ⇒ nemáme měřit

- datový signál má poloviční frekvenci – nevyužíváme plně limity té technologie

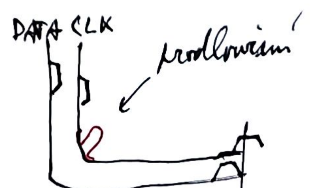
⇒ tomuto typu hodin se říká SDR (Single Data Rate)

I^2C

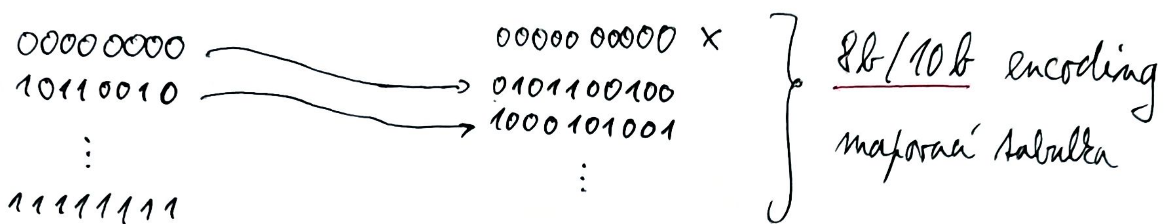
- alternativa: libovolná hrana = měříme ⇒ DDR (Double Data Rate)

⊕ minimální overhead, neomezená délka přenosu

⊖ když jsou ty vodiče různě dlouhé, tak to není synchronní



- clock recovery systém – nepoužívá clock signál
  - původně: synchronizace hodin při start condition
  - nápad: synchronizace hodin při hraně
  - problém: jsou hnusné kombinace 00000000
  - řešení: 8 bitů – 256 komb. → 10 bitů – 1024 ↗ bude tam 256 hezkých komb.



- N bitů posíláme po 8bitových kusech – pořád 20% overhead — USB
  - ale neomezená délka přenosu
- → používá se třeba 128b/132b

- kde se to používá
  1) omezená přenosová délka · RS-232 linka
  2) hodinový signál · I^2C linka
  3) clock recovery · USB linka

- Obousměrný přenos
  - Simplexní linka – jednosměrný přenos
  - Duplexní linka → half-duplex – někdy → jindy ← nepraktické
    ↳ full-duplex – obousměrná linka ⇒ dvě simplexní linky

- RS-232 linka — RS-232
  - full-duplexní linka se dvěma simplexními kanály, 8-bitové přenosy (digitální sériová)
  - pro přenos 3 vodiče: Rx (reciever) Tx (Transmiter) GND (společná zem)
  - využívaly ji např. staré sériové myši
  - out of band signály – další vodiče
    ↳ reprezentují nějaký stav: něco platí 1 "TRUE" .... přestane to platit 0 "FALSE" ... 1 ... 0
    → někdy se ty stavy definují v inverzní logice: $\overline{SIG}$, #SIG, /SIG, !SIG ↙ notace
      $\overline{\text{power out}}$ znamená 1: vše ok 0: dochází proud
    → na RTS a DTR je vždy 1 · hodné napětí vůči GND ⇒ napájení myši

- **Komunikační protokol / formát**

  - chceme poslat aktuální datum – zařízením RTC (Real Time Clock)

MSb 01011 1001 110011001100 } packet přenosu – protokol říká, jak vypadá

(01011 = den, 1001 = měsíc, 110011001100 = rok)

↑ MSb – first

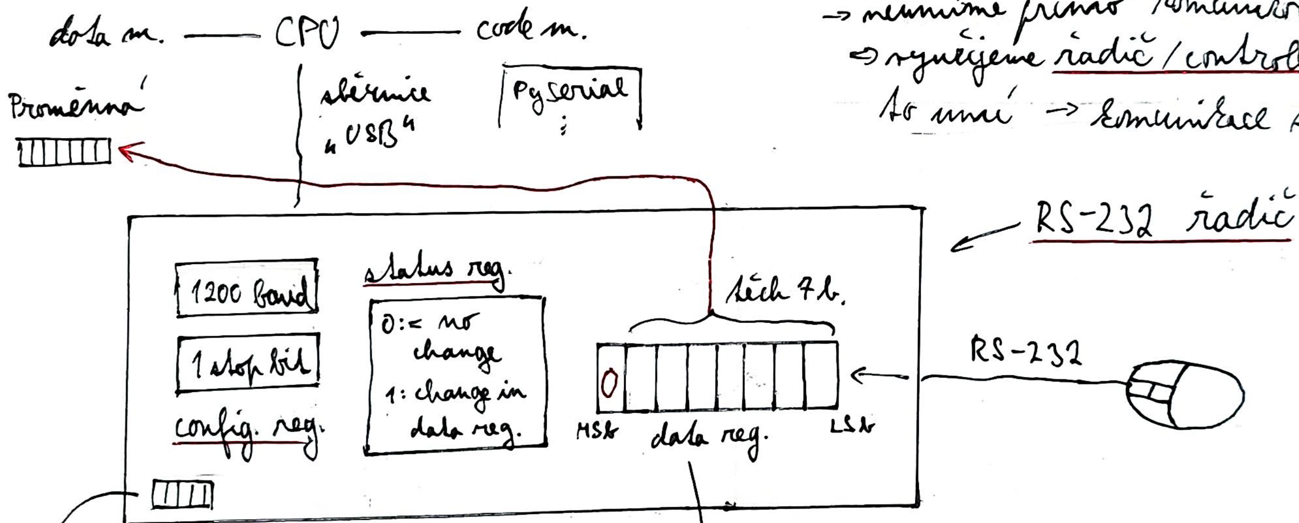
↳ po RS-232: [100DDDDD][MMMMRRRR][RRRRRRRR] – např.

  - můžeme třeba nejdřív poslat konstantu, která říká, jestli následuje čas nebo datum – vše musí být v tom komunikačním protokolu
  - kdybychom chtěli po RS-232 přenést 17bitový packet: 3 byty ⇒ 3 přenosy

- **Komunikační protokol RS-232 sériové myši**

  - používá 7bitové byty → potřeba je překládat do 8b

→ neumíme přímo komunikovat s RS-232
⇒ vynějeme řadič / controller, který to umí → komunikace s ním



Python: Pyserial / serial / Serial
→ do objektu Serial dáme ty config. info *
→ jak ten RS-232 řadič nějak jednoz. identifikujeme
→ Serial.open() do těch config. registrů zapíše *
   .read() → přečteme stavový reg, dokud tam není 1 → pak načteme obsah datového registru a status reg. se změní na 0
   → nastavení timeoutu: když je po určitou dobu idle stav, tak ta fce skončí
   ↳ když myš nic neposílá, tak .read() za chvíli skončí

Regist = malá stavová paměť
Data registr – když v něm jsou uložená data, která přijímá / zpracovává

- ta myš má nějaký 4-bytový packet = 4 RS-232 přenosy
→ řadič ovládá out-of-band signály RS-232 linky ⇒ má v sobě control registr

[... | | | 1 | 1] → jednotlivé bity odpovídají hodnotám těch OOB signálů

DTR RTS

→ když DTR a RTS nastavíme na 0, tak se myš vypne ⇒ lze ji resetovat

→ inicializační packet - když se myš připojí, tak na začátku posílá tohle

- funkce
  - blokující - read(1024) - nic nevrátí dokud nepřijme 1024 B
  - neblokující - read(1024) + timeout(0,5) - pokud 0.5 s nic nepřijde, tak vrátí co má

• Hexadecimální soustava

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

$23_{16} = \$23 = 23h = 0x23 = 2 \cdot 16 + 3 = 35$

0	0	1	2	3
65536	4096	256	16	1

→ leading zeros

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

• n-bit: $0 - 2^n - 1$

• 4-bit: 0 - 15 ← 1 hex. číslice zabere 4 bity

⇒ $123_{16} \sim 12b$

• 8b = 1B: 0 - 255

0 1 2 3 (4b 4b 4b; 1B 1B) ~ 2B

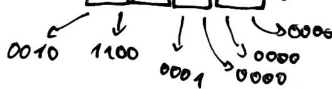
Převod 16→2

$35_{10} = 100011$

$23_{16} = 0010\,0011$ (2, 3)

⇒ je snadno vidět, kolik paměti potřebujeme

$738201600_{10} = 2C001000_{16}$ → 4B



a kde jsou v bin jedničky

0 = 0000	9 = 1001
1 = 0001	A = 1010
2 = 0010	B = 1011
3 = 0011	C = 1100
4 = 0100	D = 1101
5 = 0101	E = 1110
6 = 0110	F = 1111
7 = 0111	
8 = 1000	

• Bitové operace

Input: dvě n-bitová čísla, Output: 1 n-bitové č.

• OR (|)
1010 hodnota
1100 příkaz
1110
- 1: zapiš 1
- 0: okopíruj hodnotu

⇒ SET (to 1) → vybrané bity nastavíme na 1 a zbytek necháme být

• AND (&)
1010
1100
1000
- 1: okopíruj hodnotu
- 0: zapiš 0

⇒ CLEAR (to 0) → vybrané bity vymaž

• NOT (~)
4b: 1010 → 0101
8b: 00001010 → 11110101

→ otáčení bitů

• XOR (^)
1010
1100
0110
- 0: okopíruj
- 1: flipni

→ selektivní otáčení bitů

$1 << n = 2^n$

• SHL (<<)
$a << x = b$ (a: n-bit, b: n-bit)
1101 << 2 = 0100 (MSb, LSb)
→ posun o x bitů k MSb.

→ bitové posuny (bitwise shifts)

• SHR (>>)
$a >> x = b$ (a: n-bit, b: n-bit)
1101 >> 2 = 0011
→ posun k LSb.

a SHR n = a SHL n = 0

- <u>využití binárních operací</u>
  - → v kom. f. sériové myši je byte B=01LRYYXX ↗ koukáme jen skrz 1
  - → chceme zjistit hodnotu L ⇒ použijeme <u>bitovou masku</u> pro AND

```
AND ??L?????
    00100000
    --------
    00L00000
```

$L==1 \iff (B \,\&\, 0x20) \,!= 0$

- → B_1 = ------XX

```
000000 11  } B1 & 0x03
000000 XX
```

B_2 = --XXXXXX

```
00111111  } B2 & 0x3F
00XXXXXX
```

SHL 6 (of B_1 & 0x03)   OR

↳ XX000000 | 00XXXXXX = XXXXXXXX

⇒ OR lze použít na zkombinování dvou řetězců

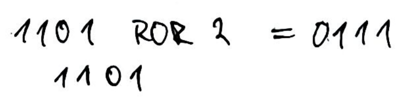
- → <u>příklady</u>

výraz	rozpis	výsledek
0x7F02 | 0x8E18	0111 1111 0000 0010 / 1000 1110 0001 1000	= 1111 1111 0001 1010 = F F 1 A
256 | 0x00FF	0000 0001 0000 0000 / —— —— 1111 1111	= 0x01FF
0x1234 & 0x0200	0001 / 0000	= 0x0200
0xC9815093 & 0x00004000	0101 / 0100	= 0x00004000
0xC9815093 & 0xFFFEFFFF	0001 / 1110	= 0xC9805093
0xC9815093 ^ 0xFF000000	1100 1001 / 0011 0110	= 0x36815093

- <u>bitové rotace</u>
  - <u>ROL</u> → rotace k MSb

1101 ROL 3 = 1110   ROL n = identita

  - <u>ROR</u> → rotace k LSb



1101 ROR 2 = 0111

- <u>Znaménková čísla</u>
  - 8-bit unsigned: 5 00000101, 6 00000110
  - 8-bit signed: 5 00000101, −6 100000110 ↗ znaménkový bit je <u>MSb</u>

0: 00000000
−0: 10000000

−127 − 127

−a = a XOR 10000000

↳ 1 znaménkový bit + 7-bit unsigned = <u>reprezentace s explicitním znam. bitem</u> = signed magnitude

+ := 0 → pro nezáporná čísla je signed i unsigned repr. stejná
− := 1

→ procesor umí porovnávat, sčítat a odečítat unsigned č. → bude to fungovat i pro signed?

−5: 10000101 ⇝ 128 + 5 = 133
−6: 10000110 ⇝ 128 + 6 = 134

133 < 134 ⇒ −5 < −6 !
133 + 1 = 134 ⇒ −5 + 1 = −6 !

→ operace pro unsigned nefungují pro signed ⇒ procesor by musel mít 2x přístupy dražší

- **jedničkový doplněk** (ones' complement)

n-bitová přesnost

+ → unsigned

− → NOT(abs(a))

$-5 = -255 + 128 + 64 + 32 + 16 + 8 + 2$

5: 00000101

−5: 11111010

−6: 11111001

−5 > −6 ✓

$-5 + 1 = 11111011 = -4$

NOT: 00000100

−127 − 127

→ pořád máme dvě nuly: −0 = 1111 1111, 0 = 0000 0000, když −a = NOT(A)

↳ když −a = NOT(a) + 1 : −0 = 1111 1111 + 1 = 1 0000 0000 (8 bit) = 0000 0000 pro 8-bit přesnost

- **dvojkový doplněk** (two's complement)

+ → unsigned

− → NOT(abs(a)) + 1

} −a = NOT(a) + 1

5: 00000101

−5: 11111011 → $-5 = -256 + 128 + 64 + 32 + 16 + 8 + 2 + 1$

−6: 11111010

−128: 1000 0000

$-128 = -256 + 128$

⇒ umíme reprezentovat −128 až 127

-2^{n-1} až $2^{n-1} - 1$

→ porovnávání: ⊕ > ⊕ ✓ ⊖ > ⊖ ✓

+ > − X

−5 > 5

} potřebujeme 1 novou operaci pro procesor – signed / unsigned porovnání

→ sčítání, odčítání funguje

↳ MSb vychází jako znaménkový bit

př: 1111 1111 → NOT + 1 → 0000 0001 = 1 → −1

python: a = 254 0000000~~0~~ (1B) 11111110 (1B) 9b. [# platných b. 9]

↗ unsigned 32-bit č. ⇒ max. velikost č. je $2^{2^{32}}$

↳ python automaticky dělá znaménková č.

↳ čísla ukládá v násobcích bytů

→ python mě nenechá se na to 254 koukat jako int8 ⇒ −2

⇒ numpy: int8 16 32 64, uint8 16 32 64

- **snížení přesnosti** = **truncation**

8-bit 5 00000101

4-bit 5 0101

$0101 = 00000101 \bmod 2^4$

−2 = 11111110

1110 = −2 ✓

X = 01001101

↳ n-bit 01101 (n = 5) Y

$Y = X \bmod 2^n$

−128 = 1000 0000

0000 ≠ −128

a = int16(12)

a = int8(a)

- **rozšíření přesnosti**
  - **bezznaménkové rozšíření** (zero extention)

    4bit 0101 5
    8bit 00000101 5

    → pro - by to nefungovalo

  - **znaménkové rozšíření** (sign ext.) → do nových bitů nakopíruje MSb

    0101 5
    00000101 5
    1110 -2
    11111110 -2

    ↳ pro unsigned to nefunguje : 1111 15 ⇒ 11111111 255

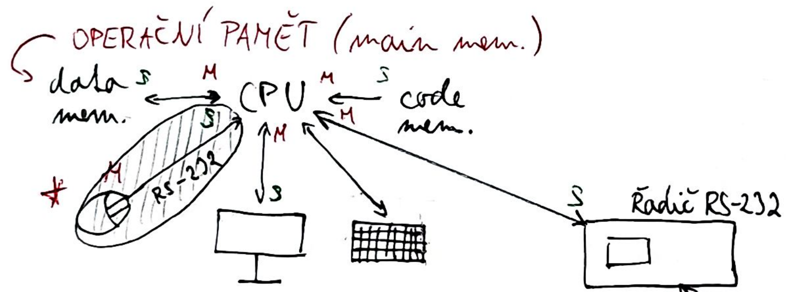
→ **numpy:**
- uint → uint : zero ext.
- uint → int, int → uint, int → int : sign ext.

→ **python bitová negace**

254 Ø11111110
~254 100000001

- normální jazyky : Øx 101
- python : signed 9-bit ⇒ -255 = -ØxFF

- **Master × slave**



master (vysílá požadavek) × slave (odpovídá na něj)

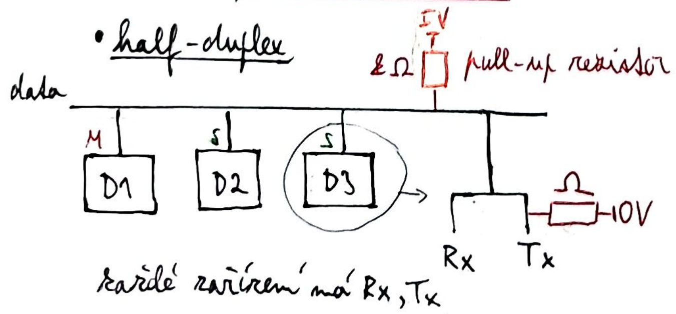
→ zápis dat do slave (write)

← čtení dat ze slave (read)

ta zařízení si mohou prohodit role

* Procesor se neumí chovat jako slave ⇒ potřebujeme ten řadič

- **point-to-point** komunikační linka vede mezi 2 zařízeními ⇒ hodně linek k CPU
- **multidrop / bus / sběrnice** - na 1 k.l. je připojeno více zařízení
  - **half-duplex**



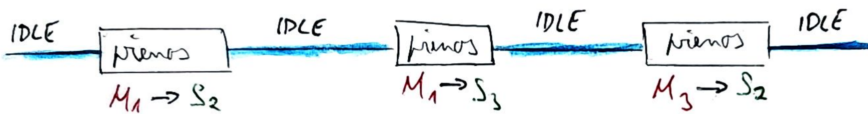
každé zařízení má Rx, Tx

- 0 zařízení → pull-up r. 5V ⇒ 1
- 1 zařízení
  - nechce vysílat ⇒ odpojí Tx ⇒ 1
  - 1 ⇒ odpojí Tx = 1
  - 0 ⇒ připojí Tx ⇒ dělič napětí ⇒ skoro 0

- **2 zařízení**

	D1	D2	BUS	
	x	x	1	IDLE
	x	1	1	linka se chová deterministicky
	x	0	0	
AND	1	1	1	
AND	0	0	0	
AND	0	1	0	
AND	1	0	0	

→ I^2C je **multimaster**
- → může být více masterů a zařízení mohou měnit roli



přenos = **transakce**

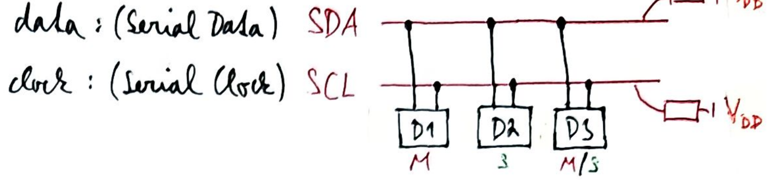
- **I^2C (Inter Integrated Circuits)**

→ multidrop half-duplex sériová linka

→ vyžívá hodinový signál ⇒ neomezená délka přenosu

data: (Serial Data) SDA

clock: (Serial Clock) SCL



→ **Napájecí napětí** V_{DD} / V_{CC} – může být různé

↳ mezní napětí: $0V — 0.3V_{DD}$ ⇝ 0

$0.7V_{DD} — V_{DD}$ ⇝ 1

→ data na SDA jsou platná, když je na SCL 1

I^2C používá bit-order MSb-first

→ je to nějak vymyšlené, aby dvě zařízení nevysílala zároveň

→ SCL signál vždy generuje master **aktuální komunikace**

→ **IDLE**: vše je odpojené ⇒ na SDA i SCL je 1

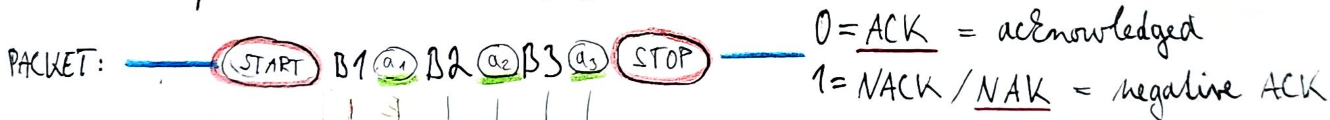
→ START condition: na hodinách je 1 a master změní SDA z 1 na 0

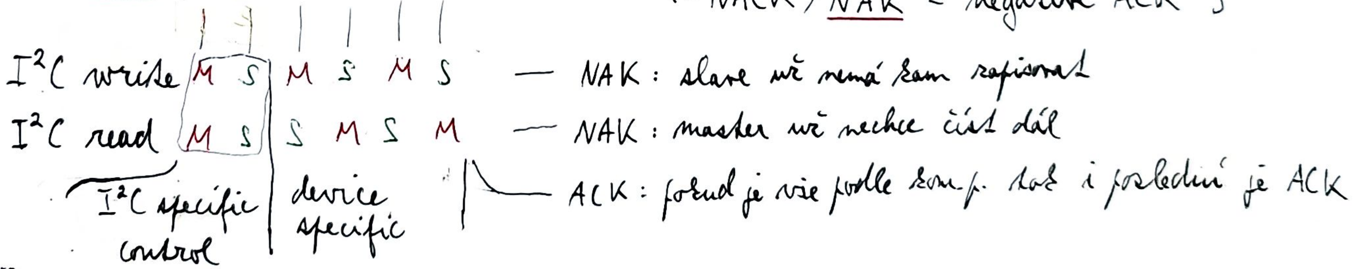
STOP condition: na hodinách je 1 a master změní SDA z 0 na 1

→ I^2C používá 9-bitové byty: 8 data bits + **1 control bit** – potvrzovací bit $\overline{ACK}$ (inverzní logika)

0 = **ACK** = acknowledged

1 = NACK / **NAK** = negative ACK





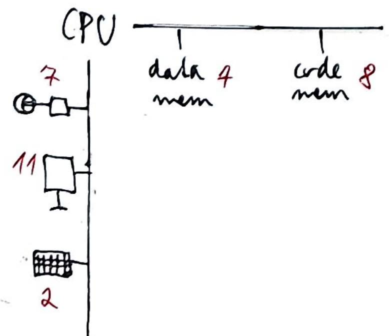
— NAK: slave už nemá kam zapisovat

— NAK: master už nechce číst dál

— ACK: pokud je vše podle komp. tak i poslední je ACK

overhead (adresa slava, požadavek) | **payload** → to užitečné v packetu – závislý na zařízení

- **adresový prostor** (adress space)



→ zařízení na sběrnicích mají adresy < slavové mají, mastři nemusí (I^2C nemají)

→ když má sběrnice n-bitový **adresový prostor**, tak zařízení na ní mohou mít adresy $0 - 2^n - 1$

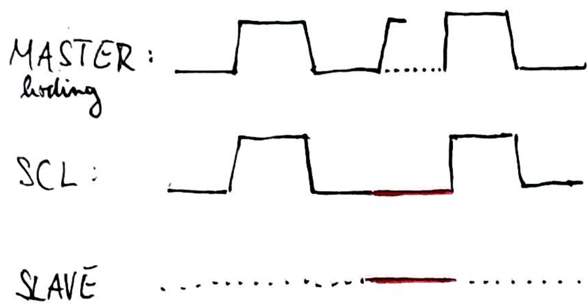
* I^2C: 7-bitový adresový prostor ⇒ 0 – 127 max 128 slavů

R/$\overline{W}$ bit 1: read

0: write

• **Clock stretching**

- na I^2C sběrnici by měl hodinový signál mít frekvenci 100 kHz - 5 MHz
- některá levná zařízení umí číst jen hodně nízké frekvence



Slave přečte bit a ví, že ho nestihne zpracovat než přijde další bit.

⇒ k SCL připojí svůj rezistor ⇒ na SCL je 0 *

→ master se snaží vysílat 1, ale vidí, že na SCL je 0 ⇒ master se odpojí ⇒ slave zpracuje bit a odpojí se

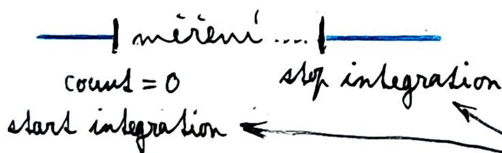
⇒ na SCL je 1 ⇒ master začne vysílat a synchronizuje se

⇒ slave se může bránit, když master generuje data moc rychle pomocí clock-hold-low *

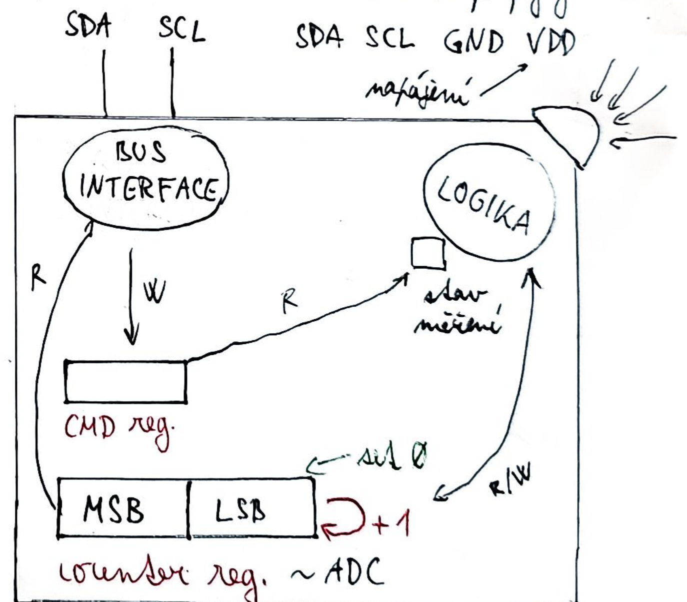
• **Ambient Light Sensor (ALS)** - příklad I^2C zařízení

- měří intenzitu světla → přizpůsobování jasu obrazovky
- ALS má counter registr
  ↳ když senzor zasáhne foton(y), tak se inkrementuje
  → před začátkem měření se vynuluje

→ na ALS se připojují 4 vodiče SDA SCL GND VDD



- command registr - pamatuje si poslední příkaz
  ↳ pošleme write s nějakým kódem



- ALS komunikuje s I^2C pomocí sběrnicového rozhraní (bus interface)
- má hardwarově danou slave adress → je hardwired (zadrátovaná)
  ⇒ na jedné sběrnici nemohou být dva

→ do CMD reg. pouze zapisujeme ⇒ write-only W/O
→ z count. reg pouze čteme → read-only R/O
} logika taky něco dělá, ale to jde mimo nás
→ ALS to nemá, ale jsou i read-write R/W

→ counter má více bytů – v jakém pořadí se posílají?

→ myšlenka:

MSB				LSB
MSb				LSb

← 32 b. packet po 4 bytech, MSb-first

⇒ MSB je byte s MSb, LSB je byte s LSb

→ byte order { MSB-first, LSB-first } je třeba ho řešit, když posíláme víc bytů

→ ALS posílá obsah counter reg. LSB-first, ale I^2C má MSb-first, čili bity v obou bytech jsou MSb-first

- paměť počítače

→ paměťový adresový prostor

256 B | 0 | 1 | 2 | 3 | ... | 255 |

celkem 2^8 B ⇒ 8-bitový adresový prostor

↳ adresa bytu: n-bit unsigned

⇒ pro 256 B paměť potřebujeme alespoň 8-bit adresový p., ale funguje i libovolný větší ⇒ 16-bit 0-65535

200 B | 0 | 1 | 2 | ... | 199 | ... 255 |

↳ 8-bit, ale adresy 200-255 nebudou platné

128 B | 0 | 1 | 2 | ... | 127 | → 7-bit, ale lze i 8-bit...

256 B | 0 | ... | 255 | 256 | ... | ... | 65536 |

⇒ pro 8-bit a.p. bychom museli používat uint8 → kdybychom vyměnili paměť za nějakou s 16-bit a.p. tak bychom museli přepsat program

⇒ výrobci pamětí často používají větší adresový prostor než je její kapacita

- jednotky

1kB = 1024 B → 10-bit → 1KiB → 16 bit → 64 kB (65536)

1MB = 1024 kB → 20-bit → 1MiB → 24 bit → 16 MB

1GB = 1024 MB → 30-bit → 1GiB → 32 bit (1) → 4 GB (2)

1TB
1PB
1EB

(1) Jak velkou proměnnou potřebujeme na uložení adresy?

(2) Jak velký adresový prostor lze pomocí ní nadresovat?

- výrobci pevných disků

1kB = 1000 B

1TB = 1000 kB

- registr řadiče

1 registr → 8 bit = 8 · (1 bit) → implementace pomocí 1 latch (pamatuje si 1 nebo 0) (4-6 tranzistorů)

- paměť SRAM S = Static

→ implementace stejnou technologií jako registry 256 B = 256 · 8 · 1 bit

RAM = Random Access Memory

• ~~Random Access~~ Memory

1) Můžu si vybrat, ke kterému bytu přistupuju – dneska skoro všechny paměti

2) Uniform speed – přístup k libovolnému bytu v libovolném pořadí stejně stojí

↳ tohle pro paměti RAM už dávno neplatí ⇒ nejsou random access

přístup	SRAM	DRAM
sekvenční ↗	✓	✓
sekvenční ↘	×	×
random	×	×

	SRAM	DRAM
• kapacita	B-kB-MB	MB-GB-10GB
• přenosová rychlost	10-100 GB/s	1-10 GB/s
• přístupová doba (access t.)	~1 ns	~10 ns

→ charakteristická vlastnost RAM

- R/W ⇒ dá se použít pro data mem. i code mem. – ale to jsou dneska ty paměti
- volatile ⇒ když ji odpojíme od proudu, tak zapomene svůj obsah

⇒ code mem. musí být non-volatile ⇒ code mem. nemůže být RAM

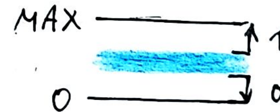
→ 1 bit ~ 4-6 tranzistorů ⇒ ty paměti nemohou být moc velké

⇒ chceme, aby operační paměť (data mem.) byla velká ⇒ SRAM se nehodí

• DRAM – není random-access, je R/W a volatile

→ levnější než SRAM, 1 bit ~ 1 tranzistor + 1 kondenzátor, větší kapacita

→ kondenzátor – naplněný elektrony ⇒ 1
 – vybitý ⇒ 0



→ Dynamic RAM – data se zapomenou za 1ms – i když to je zapojené

↳ protože kondenzátory 1 se vybíjí do kondenzátorů s 0

⇒ než se to stane, tak obnovíme původní náboj = refresh

↳ když to děláme pravidelně, tak to nezapomene

→ to dělá buď CPU nebo nějaká speciální součástka

→ Problém: když se provádí refresh DRAM, tak není možné číst ani zapisovat

⇒ DRAM je asi 10x pomalejší než SRAM

⇒ registry: SRAM

Operační paměť: SRAM / DRAM

↗ je volatile ⇒ když mám v nějaké proměnné heslo → po vyndání paměti se samo smaže ✓

• I^2C 256 B SRAM

- má 8 vodičů : SDA, SCL (I^2C), VDD, VSS (GND) (napájení), A0, A1, A2 (Slave address), TEST
- má programovatelnou adresu – na A0, A1, A2 můžeme připojit VDD (1) nebo GND (0)
  - ⇒ adresa : 1010 A2 A1 A0
- kom. protokol: | Slave a. | 0 ($\overline{W}$) | Word a. | Data | → co chci zapsat na tu adresu
  - ↳ adresa slova *

→ slovo (word) = jednotka přenosu / zpracování
  - → definované pro každé zařízení
  - → 8bit slovo ⇒ zařízení pošle 1B v každé transakci
  - → n-bitové slovo ⇒ má ho n-bitové zařízení
  - ⇒ n-bitová paměť má n-bitové slovo ne n-bitový adresový prostor !
  - → procesory mají taky bitovost → jeho registry mají velikost toho slova, operace provádí s čísly o délce toho slova ...
  - → 16-bitová paměť nemyslí v bytech, ale ve slovech – ta jsou číslovaná, ale CPU (program) pracuje s adresami bytů

8bit mem – pohled CPU	0	1	2	3	4	5	6	7	8	9	10	11	12 ...
16-bit mem.	0		1		2		3		4		5		
32-bit mem.	0				1				2				

- → v kom. protokolu té paměti je adresa toho slova a vždy se přenese celé slovo
- ⇒ stačí nám menší adresový prostor + větší přenosová rychlost
  - ↳ 16-bit slovu stačí o 1 b míň
  - ↳ na stejný overhead t.j. přenese víc
- → když po 32-bit paměti chceme 4. byte, tak si vyžádáme 1. slovo a z něj si vezmeme 0. byte – tohle většinou dělá procesor za nás

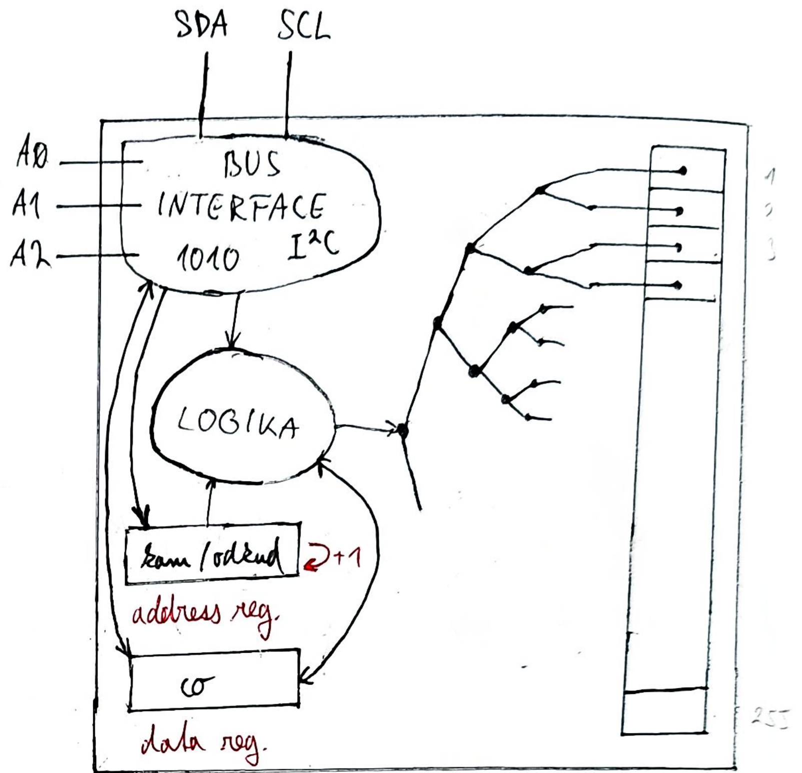
- → tahle konkrétní 256 B SRAM má 8-bit slovo, takže adresa slova = adresa bytu *
- → dřív byla většina pamětí 16-bit ⇒ slovem je často myšleno 16 bitů
- → doubleword (DWORD / DW) = dvojslovo – dvojnásobek slova – často 32 bitů
- → quadword (QWORD / QW) = čtyřslovo – čtyřnásobek slova – často 64 bitů

→ overhead komunikačního protokolu

→ na 1 datový byte přenese 3·9 bitů ⇒ $\frac{19}{27}$ ≈ 70% overhead

→ pro 16-bitové slovo: na 2B 4·9 bitů ⇒ $\frac{36-16}{36}$ ≈ 55% overhead

→ v paměti nejsou adresy uvedené, ale logikou se pomocí „výhybek" najde správná cesta ke konkrétní adrese – 0 a 1 říkají kam zatočit



→ má address register, kde je uložená adresa slova co chceme
↳ jeho velikost = velikost a.p. slov.

→ v data registru je slovo, které chceme přečíst nebo zapsat
↳ jeho velikost = velikost slova

→ burst přenos: když něco zapíšeme nebo přečteme z paměti, tak se hodnota v adresovém reg. inkrementuje ⇒ když zapisuju / čtu víc slov sekvenčně, tak to můžu udělat v 1 transakci ⇒ malý overhead

→ pro nekonečný přenos: 2·9 bitů + 1 bit pro každý datový byte ⇒ $\frac{18+n}{18+9n} = \frac{1}{9} = 11\%$

→ write × read transakce

- write = 1 write
- read = 1 write (slave address + word adress)
  1 read (slave address) → nyní slave posílá sekvenční slova od

⇒ čtení je pomalejší než zápis – to je omezení I²C ne RAM

→ DRAM má rychlejší čtení než zápis

- Registrový adresový prostor

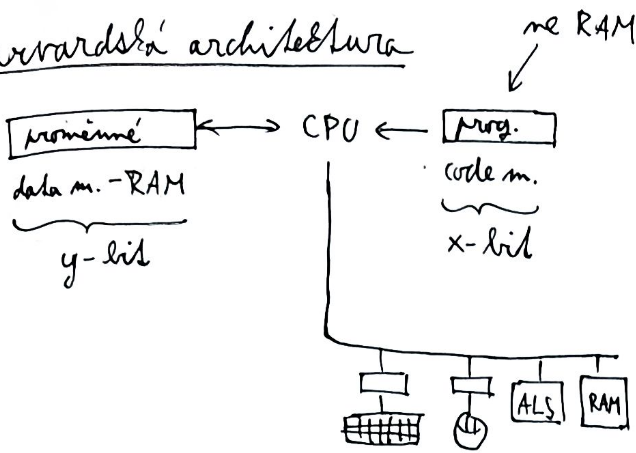
↗ tzv. adresy nejsou 0, 1, ... n, ale nějaká technicky příjemná čísla

– když má nějaké zařízení více write nebo read registrů, tak mají nějaké hardwired adresy, na které se odkazujeme v kom. protokolu

WRITE: [Slave a.] [Reg. a.] [Data]

Read: [Slave a.] [Reg. a.] ← write
↳ [Slave a.] [Data] ← posílá slave

• **Harvardská architektura**



→ procesor podporuje nějaké kom. protokoly s y-bit a x-bit adresovými prostory

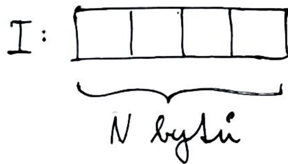
⇒ musíme vybrat data m. a code m., které ten procesor podporuje

→ Instrukce procesoru = příkazy, které ten procesor umí vykonávat

↳ Instrukční sada (Instruction set) je množina těch instrukcí

↳ různé procesory mají různé instrukční sady

⇒ instrukce jsou uložené v code mem. jako posloupnosti bytů

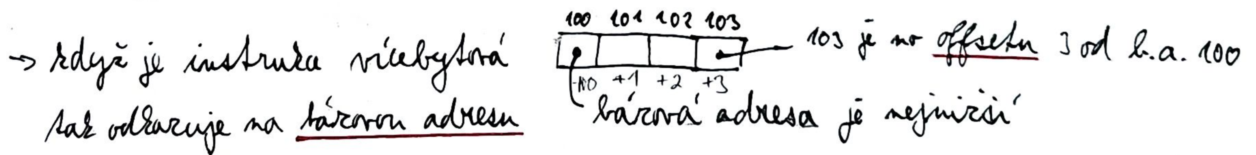


→ některé procesory mají všechny instrukce stejně dlouhé (homogenní) jiné je mají heterogenní

⇒ posloupnost instrukcí | I_1 | I_2 | I_3 | (N1, N2, N3) → procesor je vykonává směrem k rostoucím adresám

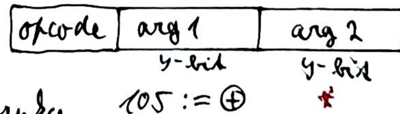
⇒ procesor v sobě má registry, které reprezentují jeho stav

• Program Counter (PC) = Instruction Pointer (IP) → je v něm uložená adresa instrukce, která se právě vykonává



→ až aktuální instrukce skončí, tak IP inkrementuje o její velikost (+N1)

→ Instrukce se skládá ze 2 částí



1) opcode – identifikátor té instrukce 105 := ⊕

2) argumenty – co sčítám – mohou být implicitní (inkrementace o 1)

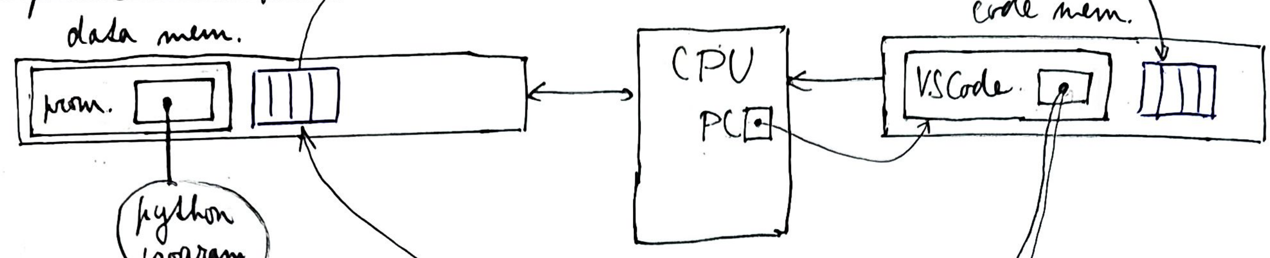
→ velikost program countru je x bitů pro code mem. s x-bitovým adresovým p.

→ argumenty jsou proměnné, reprezentované jejich adresami

↳ data m. má y-bitový adresový prostor * ⇒ argumenty jsou y-bitové

→ strojový kód = program zapsaný pomocí instrukcí procesoru

• Compiler x Interpretr



python program

↳ TEXT — VSTUP →

Překladač (Compiler)

↳ VÝSTUP → Strojový kód ⇒ nějak ho překopírujeme do code mem. a PC se posune na jeho bázovou adresu

$a = b + c$ ~ 1 strojová instrukce

x

Interpret (Interpretr)

```
if znak == "+":
    x = a
    y = b
    z = x + y   add
if ...
if ...
```

} spousta instrukcí navíc a ten náš program se nikdy nepřeloží do strojového kódu

→ interpret je mnohem pomalejší než překladač, ale je mnohem snažší ho napsat

→ python je interpretovaný, C, C#, Java, ... compilované

PYTHON

proměnná = ID jménem

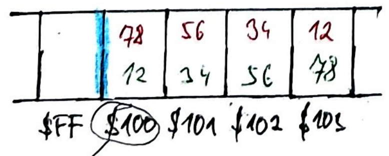
STROJOVÝ KÓD

proměnná = ID adresou → když s ní provádím instrukci, tak její součástí je adresa té proměnné v data mem.

⇒ překladač musí mít přehled, kde je volné místo v paměti, aby se proměnné naalokovaly na volných adresách

• Endianita (Endianness)

→ chceme uložit nějakou vícebytovou proměnnou někam do paměti



uint32

• Little Endian (LE) – LSB ukládám na nejmenší adresu

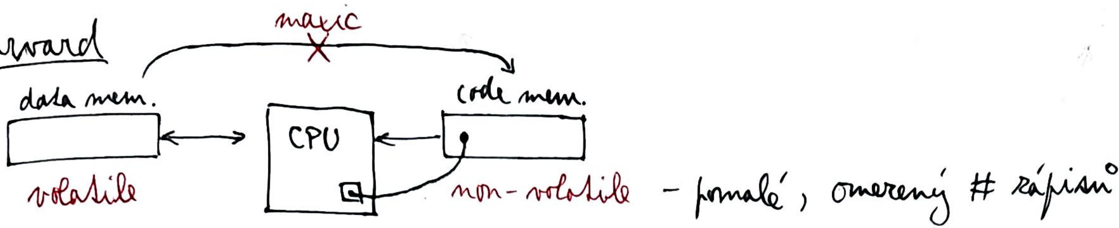
• Big Endian (BE) – LSB ukládám na největší adresu

→ pravidlo LLL: LE je pro LSB na Lowest adrese

→ endianita je daná procesorem – většina dnešních procesorů je LE

→ problém: Z BE počítače uložím data na flashku → když ji zapojím do LE počítače, tak by si do paměti mohl uložit BE data – musí se to řešit

• Harvard



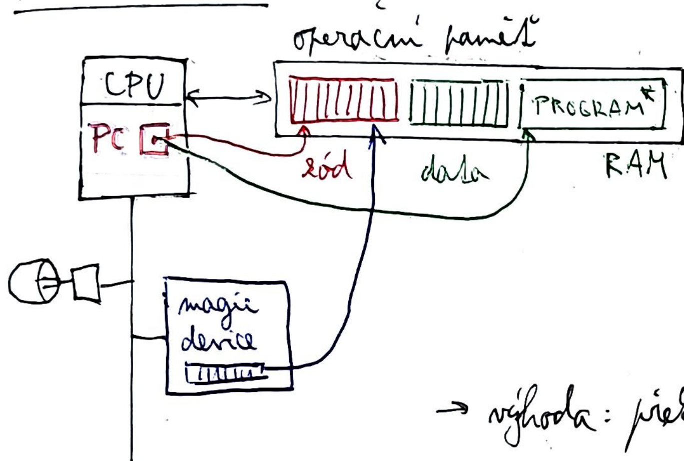
– pomalé, omezený # zápisů

→ mohli bychom všechno interpretovat ⇒ pomalé

→ překladač potřebuje zkopírovat kód z data m. do code mem.

⇒ typické procesory s Harvardskou architekturou tohle neumí magic

• von Neumann



• PC odkazuje jen do té části paměti, kde je kód

• když je argumentem nějaké instrukce proměnná, tak se uloží do té části paměti, kde jsou data

→ hardwarově je to složitější

→ výhoda: překladač vygeneruje data reprezentující strojový kód toho programu* a registr PC na něj začne ukazovat

⇒ ten program se začne provádět aniž bychom něco kopírovali

→ nebezpečí: hacker může do dat uložit škodlivé instrukce ↗

→ operační paměť je RAM (rychlá) ⇒ volatile ⇒ maže se nám ten počáteční program

⇒ magical device, které při zapnutí počítače ještě předtím, než procesor začne pracovat nakopíruje do operační paměti ten počáteční program z nějaké non-volatile paměti

→ historicky: Apple II – VisiCalc (Excel) ⇒ převratná aplikace

→ procesoru stačí umět pracovat s jedním x-bitovým adresovým prostorem

→ domácí počítače Altair, Apple I, Apple II, Atari, ... měly 8-bit, 16-bit

- 6502: 8-bit procesor, 16-bit adresový prostor → nadresuje 64 kB
- Intel 8088: 16 bit procesor, 20-bit adresový prostor → 1 MB
(x86-16)

↳ IBM PC – na tu dobu hodně dobré
\ měl sice jen 64 kB paměti, ale kvůli 20-bit a.p. kompatibilní
⇒ ty 8-bit 16-bit pomíjely – pro výkonnější procesor bylo třeba přepsat programy ⇒ do IBM PC stačilo dát větší paměť

- Intel x86/IA32: 32-bit procesory, 32-bit adresový p. → 4GB
↳ Trik: 2 instrukční sady: starou z x86-16 + novou ⇒ back-compatibilní
- Intel 64/x64: 64-bit procesory, 64-bit adresový p. ~ ∞
↳ 3 instrukční sady ⇒ back-compatibility

- Základní instrukce

6502 (LE)	Intel x86 (LE) ← 32-bit a.p.	
0	0	← offset od base adresy
\$EA	\$90	← machine code

nic nedělej: PC := PC+1

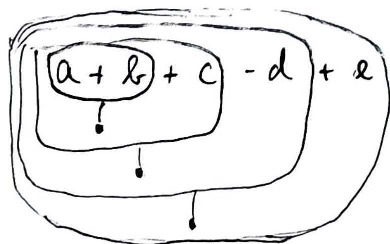
0 1 2	0 1 2 3 4
\$4C xx_0 xx_1	\$E9 xx_0 xx_1 xx_2 xx_3 (LE)
PC := \$ $xx_1 xx_0$	PC := \$ $xx_3 xx_2 xx_1 xx_0$

instrukce skoku (jump) | nepodmíněný skok / unconditional jump
↳ PC skočí na adresu X
→ ∃ podmíněný skok → pro if-u skočit na něj

→ programovat v machine kódu by bylo strašně složité

- Asembler
- programovací jazyk, který je textovým zápisem toho strojového kódu (→ pro každý procesor jiný)

6502: nic nedělej: NOP
skok: JMP \$ $xx_1 xx_0$
} pak se musí přeložit do machine kódu



MIPS: a := b op c
x86, 6502: a := a op b
↳ umí pouze přičítat, přinásobovat, ...

Assembler

$a = b + c$

adresy ⇒ 32-bit a.p. by ta instrukce byla moc dlouhá

→ často: při operaci může být jen 1 proměná, zbytek je uložený jako mezivýsledky v nějakých obecných registrech toho procesoru

6502
x86

↳ PC / IP je speciální reg.

LOAD addr(b) → R1
ADD R1 + addr(c) → R2
STORE R2 → addr(a)

→ Load-Store arch.
→ všechny argumenty musí být reg.
→ např. MIPS

→ x-bitový procesor má x-bitové obecné reg.

→ konstanta / immediate hodnota = argument, který není adresa proměnné

1, LDA #$XX_0 ← load constant $XX_0 to registr A   8-bit procesor

2, LDA $XX_1XX_0 ← load the 8-bit value from address $XX_1XX_0 to reg. A

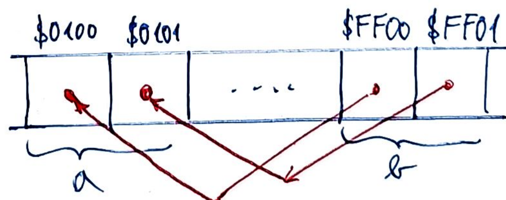
↳ na úrovni strojového kódu to jsou dvě odlišné instrukce

→ kopírování mezi registry (transfer): TAB → B:=A

ZDROJ ↙ ↳ CÍL

uint8: a=b   addr(a) = 0x0100 addr(b) = 0xFF00

LDA $FF00
STA $0100



uint16:

LDA $FF00   LDA $FF01
STA $0100   STA $0101

a = b

→ **příklad** 6502

$\text{LDA } \#\$XX_0 \equiv \$A9\ XX_0$

$\text{LDA } \$XX_1XX_0 \equiv \$AD\ XX_0\ XX_1$

$\text{STA } \$XX_1XX_0 \equiv \$8D\ XX_0\ XX_1$

32-bit prom. one na `$A404`
32-bit prom. two na `$A410`

→ Zapiš dané příkazy do strojového kódu a Assembleru. Strojové kódy ulož na

a) `$1400` b) `$1500` c) `$15FC`

a) two = one

Assembler		Strojový kód
LDA	`$A404`	AD 04 A4
STA	`$A410`	8D 10 A4
LDA	`$A405`	AD 05 A4
STA	`$A411`	8D 11 A4
LDA	`$A406`	AD 06 A4
STA	`$A412`	8D 12 A4
LDA	`$A407`	AD 07 A4
STA	`$A413`	8D 13 A4

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1400:	AD	04	A4	8D	10	A4	AD	05	A4	8D	11	A4	AD	06	A4	8D
1410:	12	A4	AD	07	A4	8D	13	A4								

b) one = 1277 = `$000004FD` (MSB … LSB)

LE

```
LDA #$FD
STA $A404
LDA #$04
STA $A405
LDA #$00
STA $A406
STA $A407
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1500:	A9	FD	8D	04	A4	A9	04	8D	05	A4	A9	00	8D	06	A4	8D
1510:	07	A4														

c) two = -2 = `$FFFF FFFE`

```
LDA #$FE
STA $A410
LDA #$FF
STA $A411
STA $A412
STA $A413
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
15FC:	A9	FE	8D	10	A4	A9	FF	8D	11	A4	8D	12	A4	8D	13	A4

$\text{JMP } \$XX_1XX_0 \equiv \$4C\ XX_0XX_1$ $\quad$ $\text{NOP} \equiv \$EA$

→ **příklad**: Proběhne program níže. Zapiš konečnou hodnotu všech bytů, které program změnil.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2000:	A9	03	8D	00	A9	4C	0A	20	A9	AB	8D	01	A9	EA	AD	09
2010:	20	8D	02	A9	4C	00	50	EA	EA	EA	EA	EA	EA	EA	EA	EA

A = 03

- (`$A900`)^ = 03
- (`$A901`)^ = 03

A = (2009)^ = AB

- (`$A902`)^ = AB

→ příklad: Před během programu jsou na adresách 8000 – 800F nuly. Napiš hexdump po prog.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2000:	A9	FF	4C	10	20	A9	EA	8D	1A	20	8D	1B	20	8D	1C	20
2010:	A9	00 (12)	8D	00	80	A9	12	8D	11	20	4C (EA)	05 (EA)	20 (EA)	8D	01	80
...																
8000:	12	12	00	00	00	00	00	00	00	00	00	00	00	00	00	00

A = FF → A = 00 → (8000)^ = 00 → A = 12 → (2011)^ = 12

A = EA → (201A)^ = EA → (201B)^ = EA → (201C)^ = EA

A = 12 → (8000)^ = 12 → A = 12 → (2011)^ = 12 → (8001)^ = 12

- **příznakový registr procesoru (flags registr)**

↳ n příznaků → 1 příznak = 1 flag = 1 bit informace

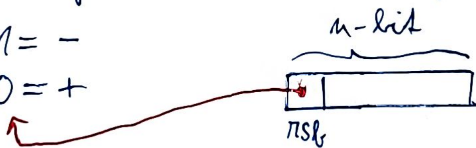
→ ty flagy spolu nesouvisí – u myši: L R M ← flagy

- **zero-flag** – říká, jestli výsledek předchozí operace
  - → byl roven nule → 1 ano byla nula
  - → nebyl roven nule → 0 ne, nebyla nula

- **sign/negative** – informace o znaménku předchozí operace

1 = –

0 = +



pro znaménková čísla dává smysl

pro bezznaménková čísla to je prostě MSb

- **carry** – pomocný příznak, když nějaká instrukce potřebuje bit navíc
  - → nějaký výsledek se nevejde do obecného registru, tak se ten bit navíc uloží sem

→ některé instrukce mají side efekty → nastavování příznaků

↳ 6502: Load a některé transfer instrukce nastavují zero a negative flag

↳ typicky to dělají všechny aritmetické instrukce

→ čtení flagu: procesory standardně nemají instrukci na čtení flagů

⇒ conditional jump / branch

if flag: JMP ← jako podmínku if-u dáme ten flag

else: NOP

→ **nastavování příznaků**

6502: CLC = clear carry = 0
SEC = set carry = 1

↳ 6502 umí nastavovat jen carry flag

x86: CLC, STC } carry
CLZ, STZ } zero

- **obecná reg. architektura** (x86)
  - → všechny obecné r. jsou ekvivalentní
  - ⇒ více instrukcí, složitější výroba

- **akumulátorová arch.** (6502) reg A
  - → 1 obecný reg. je **akumulátor**
  - → většina aritmetických operací umí pracovat jenom s akumulátorem
  - → některé procesory mají více akumulátorů

→ **akumulátorová arch. 6502**

→ AND, OR (ORA), XOR / EOR, NOT   A := A op imm/addr

↳ 6502 nemá NOT, NOT := EOR #$FF   ↳ umí jen při-ovovávat, ...

→ SHL, SHR, ROL, ROR

↳ 6502 umí shiftovat / rolovat jen o 1: A := A op 1

→ Python: a = a | b
a, b = uint8
adresy → $A000 $B000

```
LDA $A000
ORA $B000
STA $A000
```

→ a, b = uint16: je jedno v jakém pořadí to z-orujeme

	LSB	MSB
$A000:	a_0	a_1
$B000:	b_0	b_1

a = a | b

LE

```
LDA $A000     LDA $A001
ORA $B000     ORA $B001
STA $A000     STA $A001
```

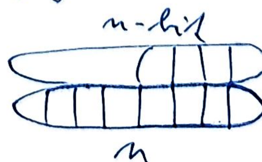
→ **sčítání**

01011	input A
01110	input B
01110	carry
11001	result

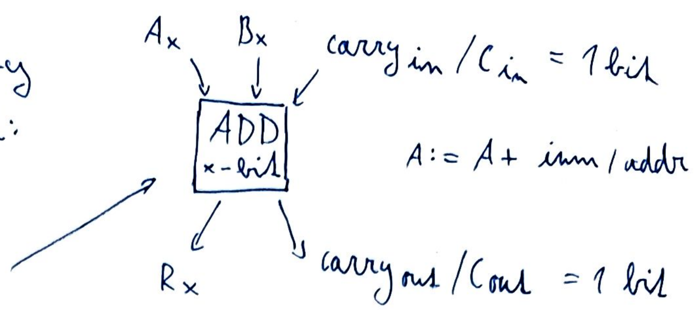
⇒ stačí nám ten carry flag

⇒ vždy pod sebou sčítáme n-bitová čísla

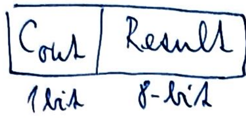
↳ kdyby 8bit ADD 16bit ⇒ sgn / zero extension



- 6502 pracuje s 8-bit slovy ($x = 8$)
  ⇒ sčítání 16-bit proměnných:



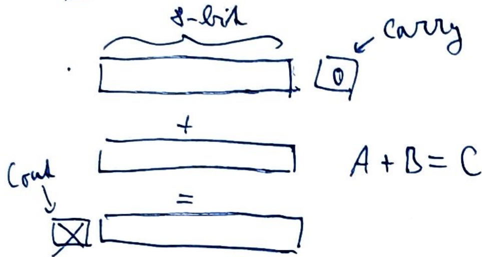
sčítání s přenosem (add with carry) **ADC**



⇒ 9 bit

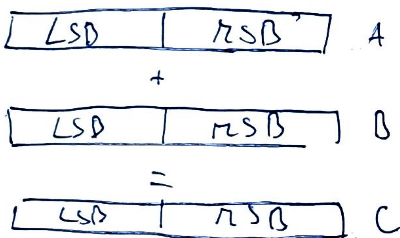
⇒ u posledního ADC uděláme truncation tím, že zapomeneme Cout

→ **součet 8-bit čísel**

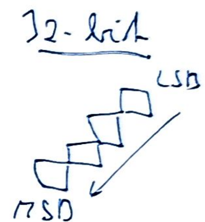



```
LDA $A000
CLC
ADC $B000
STA $C000
```

→ **součet 16-bit čísel**




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

LDA $A001
ADC $B001
STA $C001
```



→ **součet znaménkových čísel** – ve dvojkovém doplňku
- stačí se na ta znaménková čísla koukat jako na bezznaménková, sečíst je a pak se na výsledek koukat jako na znaménkové číslo

→ **increment / decrement**

6502 nemá pro reg. A → INX X := X + 1
DEX X := X − 1

side effects:
P.NEGATIVE = X.7
if X = 0 : P.zero := 1
else : P.zero := 0

→ **odečítání unsigned**

uděláme A := value − A → pak umíme i A := A − value

temp1 = A
A = value } A = temp1 − A = A − value

→ využijeme dvojkový doplněk
A = value − A = (−A) + value

→ **NOT A → INC A → ADD value**

```
6502: EOR #$FF   ← NOT
      CLC
      ADC #$01   ← INC
      CLC
      ADC value
```

→ A, value jsou 8-bit unsigned

→ potřebujeme, aby to byla 7bit čísla

⇒ ⊖ jde udělat pomocí ⊕, ale normální procesory mají instrukci odečítání

→ **Subtract with borrow** 6502 jí nemá x86 ano

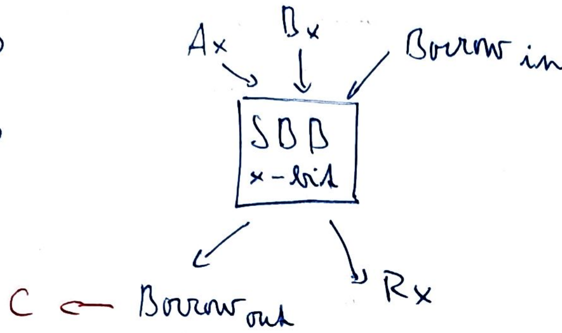
SBB a, b

a := a - (b + Borrow)

carry := borrow from last bit

$$
\begin{array}{r}
{}^{0}1^{1}1^{1}0^{0}1 \\
-\ 0\,1\,1\,1 \\
\hline
0\,1\,1\,0
\end{array}
\qquad
\begin{array}{r}
13 \\
-7 \\
\hline
6
\end{array}
$$

C	= B
1	1
0	0



x-bitový SBB

CLC
SBB } 8-bit odečítání

→ **Subtract wit carry** má jí 6502

SBC

C =	NOT (B)
1	0
0	1

result := A − imm / addr − NOT (P. Carry)

8-bit přetočt

$$
\begin{array}{r}
11111111 \\
-\ 10111000 \\
\hline
01000111
\end{array}
$$

$SBC\ X = A - X - B = A - X - B + 256 = A - X - NOT(C) + 256 =$

$= A - X - (1 - C) + 256 = A - X + C + 255 = A + (255 - X) + C$

$= A + NOT(X) + C$

SBC X := ADC NOT(X)

↖ hardwarově je snadné to naimplementovat

SEC ← C = 1 ⇒ B = 0
SBC } 8-bit odečítání

→ **8-bit odečítání**

A := value − A ≡
temp1 := A
A := value
SEC (C=1)
SBC temp1 } ≡ SEC
NOT temp1
ADC temp1 ≡ NOT temp1
INC temp1
ADD temp1 *

↳ ADD + 1 z carry (INC)

* Postup z minulé strany funguje i pro sčítání 8b. čísel

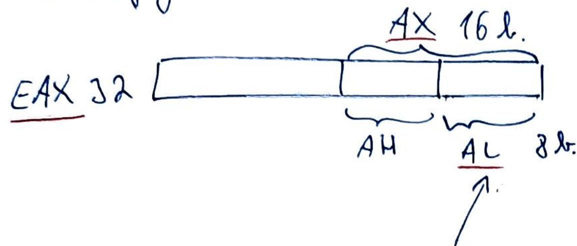
• Obecná registrová arch. x86

instruction pointer : EIP 32 bit

příznakový registr : EFlags 32 bit

7 obecných registrů 32 bit

→ co kdybychom chtěli sčítat 16-bit proměnné? ⇒ další 16-bit instrukce



↳ 16-bit pohled do spodní části toho reg.

⇒ chová se to jako 16-bit reg., ale sdílí paměť s tím 32 bit

↳ provádí se truncation 16 b, 17 b, ... při např. sčítání

můžeme ještě pohlížet na 1B. AL / AH

→ arch x64 → 64 bit reg. + 32 bit operace ⇒ velký 64 bit reg + dělení na 32, ...

→ x86 assembler: MOV target, source → target := source

MOV r, imm → LOAD

MOV imm, r → STORE

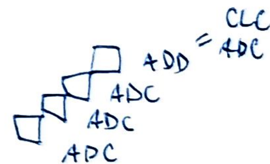
MOV r2, r1 → transfer

OP target, source → target := target OP source

→ MOV [r1 + addr], r2 → umíme ukládat na offset r1

→ operace: ADD, ADC, SUB, SBB, IMUL, IDIV
OR, AND, XOR, NOT, SHR, SAR

operace mohou být mezi registry



→ operace mají sideeffecty, MOV nemá

→ příklad: $r = a + b + e - (c + d)$

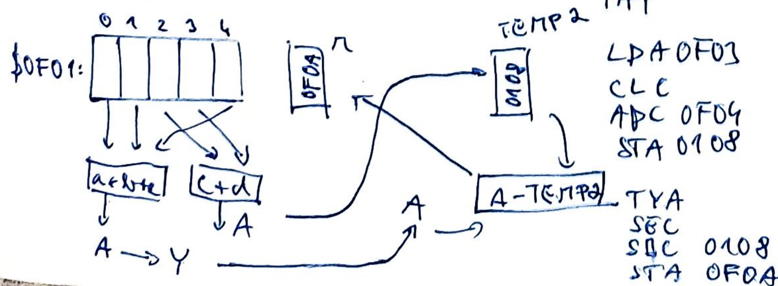
6502, 8-bit proměnné

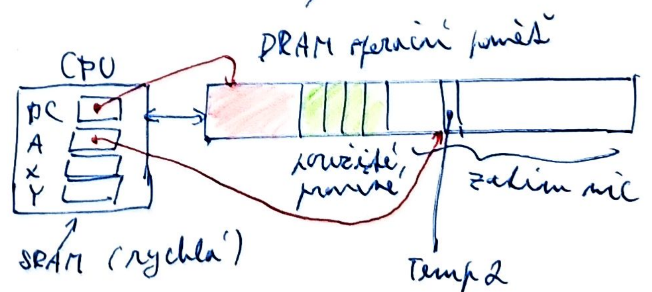
TEMP1 = a + b + e

TEMP2 = c + d

r = TEMP1 - TEMP2

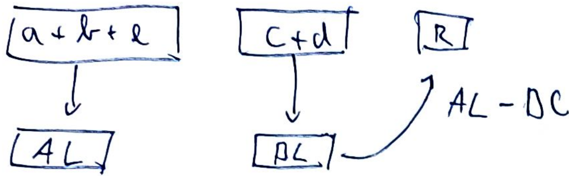
```
LDA 0F01
CLC
ADC 0F02
CLC
ADC 0F03
TAY
LDA 0F03
CLC
ADC 0F04
STA 0108
TYA
SEC
SBC 0108
STA 0F0A
```





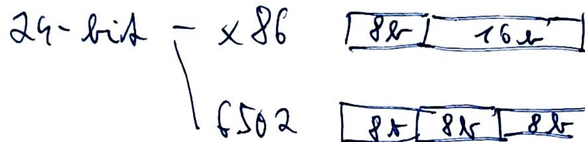
↳ dokud to jde, tak si chceme vše ukládat do registrů

→ x86 r = a + b + e − (c+d)




```
MOV  AL, [00000F01h]
ADD  AL, [00000F02h]
ADD  AL, [00000F05h]

MOV  BL [00000F03h]
ADD  BL [00000F04h]

SUB  AL, BL
MOV  [00000F0Ah], AL
```

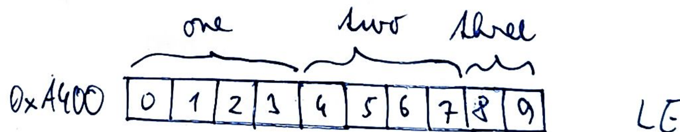
→ co kdyby byly (a, b, c, d, e) 32-bit → na x86 se pouze změní adresy + opcody inst. + je to stejně rychlé
→ na 6502 by to bylo mnohem komplikovanější



→ příklad

```
  11101010
  11011111
  --------
  11111110  Carry
 111001001

  11101011
       101
  --------
  00001111  C
  11110000
```

→ one uint32 0xA400
two uint32 0xA404
three uint16 0xA408



509 = $000001FD

zero ext. do 32 bit písmo

• two = one + two

```
LDA $A400
CLC
ADC $A404
STA $A404

LDA $A401
ADC $A405
STA $A405

LDA $A402
ADC $A406
STA $A406

LDA $A403
ADC $A407
STA $A407
```

• one = two + 509

```
LDA $A404
CLC
ADC #$FD
STA $A400

LDA $A405
ADC #$01
STA $A401

LDA $A406
ADC #$00
STA $A402

LDA $A407
ADC #$00
STA $A403
```

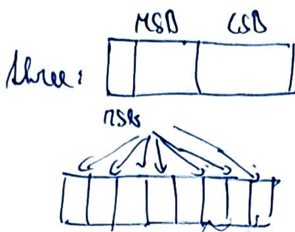
• two = one + three

```
CLC
LDA $A400
ADC $A408
STA $A404

LDA $A401
ADC $A409
STA $A405

LDA $A402
ADC #$00
STA $A406

LDA $A403
ADC #$00
STA $A407
```

→ stejné proměnné, stejné adresy, ale SIGNED

- two = one + two
  → stejné
- one = two + 509
  → stejné

- two = one + three → potřebujeme sign ext. three
  ↳ nakopírovat MSb.
  → horní 2 bajty 00 nebo FF



← 00 v FF

→ pomocná paměť: $D500 – $D600

SHL = ASL   SHR = LSR, shift o 1

```
      LDA $A409   ← MSB
      AND #$80    ← [MSb | 0000...]
      STA $D500

SHR → LSR
      ORA $D500      7-krát

      STA $D500   ← 00 v FF
```

```
CLC
LDA $A400
ADC $A408
STA $A404

LDA $A401
ADC $A409
STA $A405
```

```
LDA $A402
ADC $D500
STA $A406

LDA $A403
ADC $D500
STA $A407
```

→ příklad

```
  1110 1010
- 1101 1111
     1 1111 borrow
-----------
  0000 1011
```

```
  1110 1011
- 0000 0101
       0100 b
-----------
  1110 0110
```

```
       0000 1101      13
     - 0001 1100    - 28
 ...1  1111 0000 b
 ...1  1111 0001   ⇒ -1-2-4-8 = -15 ✓
       8-bit truncation
```

→ unsigned prom. na stejných adresách

- two = one - two

```
SEC
LDA $A400
SBC $A404
STA $A404
:
```

- one = two - 509

```
SEC
LDA $A404
SBC #$FD
STA $A400
:
```

- two = one - three
  → stejné, zase musíme provést rozšíření

- x86

one	uint32	0000A400h
two	uint64	0000A404h
three	uint64	0000A40Ch

- three = two + three

```
MOV EAX, [0000A404h]
ADD EAX, [0000A40Ch]
MOV [0000A40Ch], EAX

MOV EAX, [0000A408h]
ADC EAX, [0000A410h]
MOV [0000A410h], EAX
```

- two = one - three → zero ext.

```
MOV EAX, [0000A400h]
SUB EAX, [0000A40Ch]
MOV [0000A404h], EAX

MOV EAX, #$00000000
SBB EAX, [0000A410h]
MOV [0000A408h], EAX
```

→ *příklad 6502*

A	uint8	$C12A
B	uint8	$C12B
C	uint8	$C12C
D	uint16	$C200
E	uint16	$C202

volná: $D500 – $D600

• A = A + B + C

```
LDA $C12A
CLC
ADC $C12B
CLC
ADC $C12C
STA $C12A
```

• A = (A − B) + (A − C)

```
LDA $C12A
SEC
SBC $C12B
TAY
LDA $C12A
SEC
SBC $C12C
→
STA $D500
TYA
CLC
ADC $D500
STA $C12A
```

• E = E + D + 15 + (B − A)

15 = $000F

```
LDA $C12B
SEC
SBC $C12A
STA $D500
→
LDA $C202
CLC
ADC $C200
STA $C202
LDA $C203
ADC $C201
STA $C203
→
LDA $C202
CLC
ADC #$0F
STA $C202
LDA $C203
ADC #$00
STA $C203
→
LDA $C202
CLC
ADC $D500
STA $C202
LDA $C203
ADC #$00
STA $C203
```

→ *stejné, ale x86*

• A = A + B + C

```
MOV AL, [1000C12Ah]
ADD AL, [1000C12Bh]
ADD AL, [1000C12Ch]
MOV [1000C12Ah], AL
```

• E = E + D + 15 + (B − A)

```
MOV AX, [1000C202h]
ADD AX, [1000C202h]
ADD AX, #$000F
MOV BX, #$0000
MOV BL, [1000C12Bh]
SUB BL, [1000C12Ah]
ADD AX, BX
MOV [1000C202h], AX
```

• Taktovací frekvence (Clock rate)

- v procesoru jsou nějaké jednotky, které si mezi sebou předávají data (hodin. signál ↑)

↳ taktovací f. říká těm jednotkám, kdy mají přijímat / vysílat data ~ CLK

⇒ ten hodinový signál časuje vykonávání jednotlivých kroků procesoru

- 1 takt = 1 cycle → těch jednotek je hodně + dokáží pracovat paralelně

↳ moderní procesory dokáží za 1 takt vykonat více instrukcí

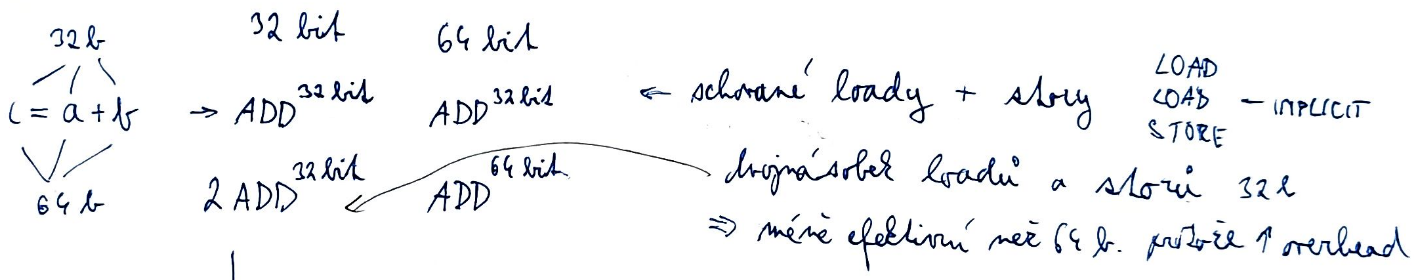
- rychlost instrukcí

fast • 1 takt – bitové operace (AND, ... SHL, ...), ADC, SBB/SBC

slow • LOAD, STORE – pracují s operační pamětí DRAM – pomalé

! ADC r_1 r_2

ADC r_1 [$addr_1$] ← implicitní load   větší slovo lepší



načítá + ukládá se 16 B do paměti → 32 b méně efektivní než 64 b.

ale obojí je pomalé ⇒ 32 b nebude o tolik horší než 64

→ 8 bit 6502 → 64 b. proměnné 8 ADD 8 bit.

	6502	80386 ↗ x86	dnešní
t.f.	~1 MHz	~33 MHz	~GHz

→ rychlost instrukce ovlivňuje hlavně

- clock rate
- počet taktů, které instrukce trvá

↳ zasahuje do DRAM? ⇒ pomalá

---

rychlost

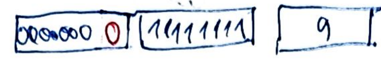
1.
```
SHL  EAX, 1
ADD  EAX, EBX
MOV  EBX, EAX      ← trochu rychlejší TRANSFER
ADD  EAX, 012345678h
```
↳ dlouhá instrukce ⇒ trochu pomalejší

2.
```
MOV  EAX, [addr]
MOV  [addr], EAX
```

• Python ← počet platných bitů ~ počítadlo

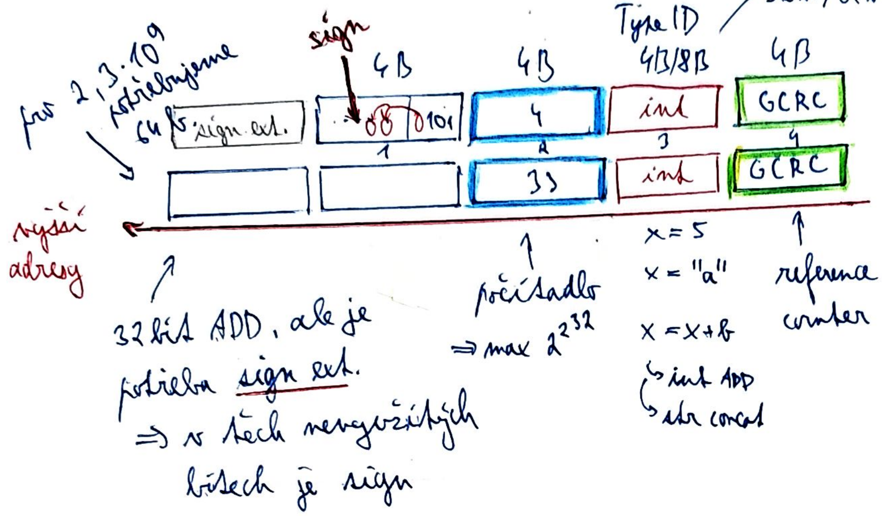
int 5 = 0101

int 255 = 011111111



→ operace ADD → museli bychom to dělat po bytech → hrozně neefektivní

⇒ python to takhle nedělá → ty chlívečky nebudou 8b. ale 32 b.

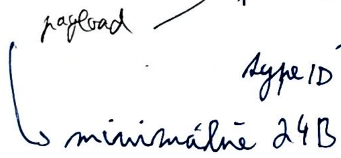


* python je řádově pomalejší než C/C#

20B overhead

→ python x = 5

⇒ zabírá 8B + 4B + 4B + 4B + 4B = 24B
(1, 2, 3, 4)



⇒ ukazatel (pointer) = proměnná, ve které je adresa do paměti
reference (Python) nemůžeme zjistit tu adresu

⇒ teď udělám x = "ahoj" → změní se ta reference
↳ ta původní 5 pořád zabírá paměť

⇒ Garbage collection - najde objekty, na které nic neodkazuje a označí je za prázdnou paměť

⇒ Reference counting

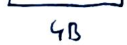
x = 5	5^{+1}	
y = x	5^{+1}	
x = "a"	$"a"^{+1}$	5^{-1}
y = 4	5^{-1}	

} když dosáhnou nulu, tak to je odpadek ⇒ ta paměť se může znova použít

⇒ cykly v grafu ref? → jednou za čas prochází graf referencí a hledá izolované komponenty

• C/C#

uint32 a = 5



→ pevný typ
→ pevná velikost
↳ aby nebylo třeba roztahovat paměť
⇒ LOAD LOAD STORE

⇒ v Pythonu musíme nakoukat ty informace okolo - co je to za objekt, kolik má platných cifer ⇒ normalizace ⇒ pak se to sečte ⇒ generace nového výsledku

→ v Pythonu jsou int, str, ... immutable - nová hodnota ⇒ nový objekt ⇒ potřebujeme nový objekt ⇒ najít místo = loady v GC ⇒ store Type ID, ... ⇒ renormalizace

⇒ finální store ⇒ 1 ADD v Pythonu ~ desítky instrukcí LOAD, STORE, ... *

• Násobení + Dělení II

→ hardwarově velmi těžké udělat

	HW 32b/64b x86	mobil ARM	µC	6502
mul	✓	✓	✗/✓	✗
div	✓	✓/✗	✗	✗

✗ → musíme to udělat SW

→ jak je to rychlé v taktech

	HW	SW
mul	1–10	ještě pomalejší
div	10–100	

↑ zanedbáváme implicitní loady a story

• Násobení dělení mocninami 2

tohle je fast ↙

• unsigned

shift left $\quad x \text{ SHL } n = (x * 2^n) \bmod 2^n \Rightarrow$ SHL = Logical shift

shift left $\quad x \text{ SHR } n = x // 2^n$

• signed

• násobení – pořád nevysuneme znaménkový bit $\quad x \text{ SHL } n = x * 2^n$ pro dostatečně malá x

• dělení – pro kladná čísla to bude fungovat

| pro – potřebujeme doleva přidávat 1 ⇒ SAR = Arithmetic shift

⇒ skoro to funguje $\qquad$ udělá sign ext. ↙

$-5 \text{ SAR } 1 = -3$

$-6 \text{ SAR } 1 = -3$

→ zaokrouhluje se to nahoru ⇒ − ⇒ +1, + ⇒ +0

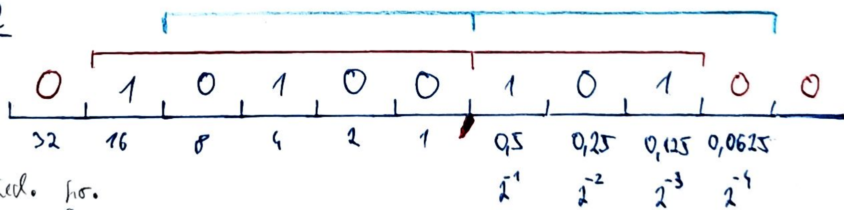
$(x + \text{MSb}(x)) \text{ SAR } n = x // 2^n$ ↙

• Reprezentace reálných čísel

$0020{,}62500$ ← trailing zeros

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

trailing zeros



⇒ fixed-point repr. $\quad 5.3 \;/\; 5+3 \quad , \quad 4.4$ by byla blbá

→ sčítání – potřebujeme stejnou fixed-point repr.

$1.25 + 1.875 = 3.125$ $\qquad$ ⇒ můžeme to sčítat jako bezznaménková celá čísla i odčítat

```
  00001.010
  00001.111
+    11.001  = 3.125
```

⇒ přičítání s fixed-point čísly je rychlé

$5.3: \quad 1{,}250 \cdot 2^3 + 1{,}875 \cdot 2^3 = C \cdot 2^3$

$1010 + 1111 = 11001 = C \cdot 2^3$

$11001 = C \ll 3 \Rightarrow C = 11001 \gg 3 = 11.001$

- Záporná čísla ve fixed-point

↳ chceme 1.25 v 5.3 ⇒ $1.25 \cdot 2^3 = a = 10$ ⇒ $1.25 = 10 >> 3$

10 : 01010.000 ⇒ 1.25 : 00001.010

↳ −1.875 v 5.3 ⇒ $-1.875 \cdot 2^3 = -15$

−15: 10001.000 ⇒ −1.875 : 11110.001 (co to je? *)

* 11110.001 ⇒ NOT(11110001) + 1 = 00001111
00001.111 = 1.875 } 11110.001 = −1.875

→ Ukládání čísel ve fixed-point

- normálně podle endianity
- na \$0010F300 až \$0010F31F jsou samé nulové bity
- uložíme
  - 18.375 jako 8.24 na \$0010F300 → 12.60 00 00 (0110)
  - 18.375 jako 12.20 na \$0010F308 → 01 2.6 00 00
  - 1040.5 jako 8.8 na \$0010F30C → ~~04~~ 10.80 ← nemám přesnost
  - 1.5 jako 4.4 na \$0010F310 → 1.8
  - 65535 jako 24.8 na \$0010F314 → 00 FF FF.00 00
- hexdump:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\$0010F300:	00	00	60	12					00	00	26	01	80	10		
\$0010F310:	18				00	00	FF	FF	00							

• Floating point repr.

20,625 = 10100.101

vědecká notace

A = −1010 0101 0000 0000.0 = −1,0100101 * 2^{15}

B = 0.0000 0000 1010 0101 = 1,0100101 * 2^{-9}

exponent

mantisa (significand)

→ normalizovaný zápis
- žádné leading zeros
- 1 cifra před .
- 0 nemá reprezentaci!

A ani B nelze v žádné 8bit fixed-point repr. reprezentovat jinak než nulou

float 8-bit — 5b. exp. 2b. mantisa

A: 1 11110 | 10 | 100101 → 0 1 | 00101

→ exp 5b → 0...31

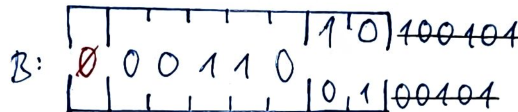
⇒ exp bias + 15 ⇒ −15...16

15 ↦ 30

−9 ↦ 6

← repr. se skrytou 1

∵ všechna čísla jsou 1.... · 2^e



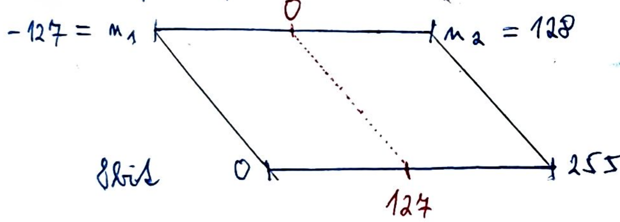
B: 0 00110 | 10 | 100101 → 0 1 | 00101

znaménkový bit (explicitní)

• musíme vědět
- → float 8b
- → 5b exp. s bias +15
- → 2b mantisa se skrytou 1
- → nejdřív m, exp, sign

→ bias repr. celých č.

↳ repr. s posunem



−127 = m_1 … 0 … m_2 = 128

8bit: 0 … 127 … 255

⇒ posun + 127 = bias + 127

−127 + 127 = 0

−126 + 127 = 1

0 + 127 = 127

128 + 127 = 255

• float aritmetika
- musíme jí softwarově naimplementovat (↗ 10–100 taktů)
  - ⇒ mnohem pomalejší než fixed-point
- některé procesory mají hardwarovou podporu (↗ 2 takty)
  - → vedle sady obecných registrů pro celočíselnou aritmetiku mají další sadu pro práci s float čísly ⇒ další instrukční sada

ADC
SBB

FADD, FSUB, FLOAD, FSTORE, ...

• Standard IEEE 754

single 32b. → 8b. exp + 23b. mantisa

−127 — 128 resp. −126 — 127 − 0, 255 mají vyhrazenou

double 64b. → 11b. exp + 52 b. mantisa

−1023 — 1024

x86 / x64	ARM	μC	6502
HW	HW/SW	SW	SW

→ HW podpora pro single + double

− skrytá 1, bias uprostřed

	Python	C/C#
single		float
double	float	double

→ sčítání floating-point čísel

23 b. mantisa ← single

$A = 1.0100101 * 2^{15} = 1.010\ 0101\ 0000\ 0000\ 0000\ 0000 \mid 0000\ 0000 * 2^{15}$

$B = 1.0100101 * 2^{-9} = 0.000\ 0000\ 0000\ 0000, 0000\ 0000 \mid 1010\ 0101 * 2^{15}$

15 nul — B

$A + B = A$ — vynulovalo se to z té 23 b. části

↑ denormalizace menšího čísla → může se z toho stát nula!

→ single: 23 b. na mantisu ⇒ když rozdíl exponentů je větší než 23, tak se to ztratí

decimal ⇒ $10\,000 \sim 2^{13}$, $+ \ 0.001 \sim 2^{-10}$ } je to na hranici

$2^{23} = 8 \cdot 10^6$

$\Rightarrow 10^9 + 16 = 10^9$ ($10^9 \to 2^{30}$, $16 \to 2^4$) — ztráta informace

! 0.1 má ve dvojkové soustavě nekonečný desetinný rozvoj

$b == a$ ✗

$-\varepsilon < (b-a) < \varepsilon$ ✓

→ reprezentace nuly

→ minimální exponent

$0{,}\underbrace{00\ldots00}_{127}1101 = 1.101 * 2^{-127} = 0.1101 * 2^{-126}$ | -127 | 110100…0 | $= 0.m * 2^{-126}$

→ IEEE: když min. exponent ⇒ mantisu denormalizujeme

$0{,}\underbrace{00\ldots000}_{127}1101 = 0.1101 * 2^{-127} = 0.01101 * 2^{-126}$ (m)

| -127 | 01101000 | $= 0.m * 2^{-126}$

| -127 | 0000…001 | ← nejmenší číslo (23 b.)

0:= | 0/1 | -127 (000…00) | 00000000 | → minimální exponent a nulová mantisa

↳ rozlišujeme −0 a +0 přičemž $-0 == +0$

↳ minimální exp. jsou v bias repr. samé nuly → 0:= samé nuly

→ max. exp.

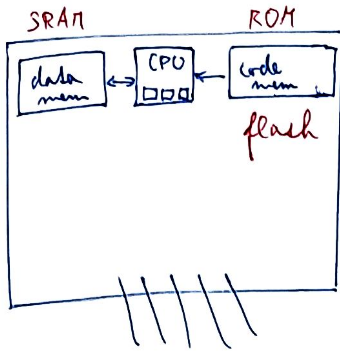
- nulová mantisa ⟨ $+\infty$, $-\infty$ podle sign — $\infty + 5 = \infty$, $5/0 = \infty$
- nenulová mantisa → Not a Number NaN — ∞/∞

↳ přijdeme o nejvyšší exponent, ale získáme NaN

→ více druhů NaN ⇒ nejde testovat $A == NaN$

⇒ musím se kouknout do paměti na ty bity

• Jednočip / μC / MCU



→ harvardská architektura

• data mem. SRAM 256B – 1kB – 10kB

• code mem. non-volatile – ROM

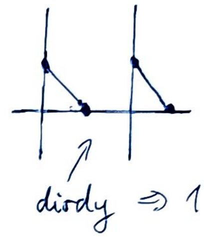
→ firmware – ten μC ovládá pračku ⇒ firmware je ten program co jí řídí – stačí mi ten program uložit jen jednou

• ROM (Read Only Memory): ∞ read 1 write ← v továrně

• PROM (Programable ROM): ∞ read 1 write ← uživatelský write

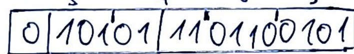
• EPROM (Eraseble PROM): ∞ read ∞ write ← nejběžnější ROM

⇒ char. vlastnost paměti ROM: non-volatile



→ příklad: IEEE, napiš hexdump paměti mezi \$0010F300 a \$0010F32F, původně 00, CE

• \$0010F300: −3,5 8bit



1|100 00|11 (C 3)

$11.1 = 1.11 * 2^1 \rightarrow 16$

• \$0010F302: x=118, 33984375 = 1110110.01010111 = $1.1101100101011 1 * 2^6 \rightarrow 21$

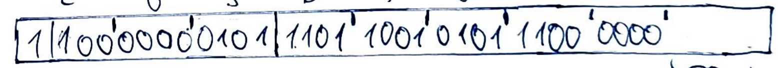
16 b. 10m, 5e

0|10101|1101100101 (5 7 6 5)

6+127 = 133

6+1023 = 1024+5

• \$0010F304: −x 32 b. 23m, 8e



1|10000101|1101100 1010111 0000 0000 (C 2 E C A E)

• \$0010F310: −x 64 b 52m, 11e

1|10000000101|1101 1001 0101 1100 0000 (C 0 5 D 9 5 C)

8 * \$0

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\$0010F300:	C3		65	59	00	AE	EC	C2	00	00	80	7F	71	00	00	80
\$0010F310:	00	00	00	00	C0	95	5D	C0								
\$0010F320:	00	00	00	00	00	00	00	D0	00	C0						

e = 128 → 255

• \$0010F308: +∞ 32 bit

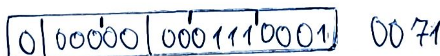
0|11111111|000 0000 4 * \$0 (7 F 8 0)

• \$0010F30C: 0,00000678 = 0.0000 0000 0000 0000 0111 0001 1011

+15 16b. 10m 5e

= $0.0001110001 * 2^{-14}$

$-15+15=0$

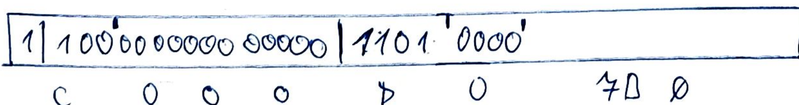


0|00000|0001110001 0071

• \$0010F30E: −0,0 16b. 1|0 ... 0 8000

• \$0010F320: −3.25 80 b. 64 b. mantisa bez skryté 1, 15 b. exp bias + 16 383

$3.25 = 11.01 = 1.101 * 2^1 \rightarrow 16384 = 2^{14}$



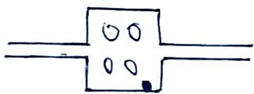
1|100 0000 0000 0000|1101 0000

C 0 0 0 D 0 7D 0

- EPROM - erase pomocí UV ~ 20 min. na slunci
- EEPROM (Electrically EPROM) - ∞ R, "∞" W × flash

	SRAM	DRAM	EEPROM × flash	HDD	CD/DVD
rychlost	> 10 GB/s	1-10 GB/s	100 MB/s - 1 GB/s - 5 GB/s *	100 MB/s	10 MB/s
přístupy	< 1 ns	~ 10 ns	~ 100 ns	~ ms	~ 100 ms
kapacita	kB-MB	MB-GB	MB-GB —— 1 TB	TB - 20 TB	
sekvenční čtení	✓	✓	✓ ✓		
random 1 byte	×	×	× ×		
random zápis 1 byte			× ×		
random 1 blok			✓ ✓		

- EEPROM
  → byte adresovatelná
  → omezený počet zápisů do 1 bytu
  ~ 100 000 - 1 000 000 W



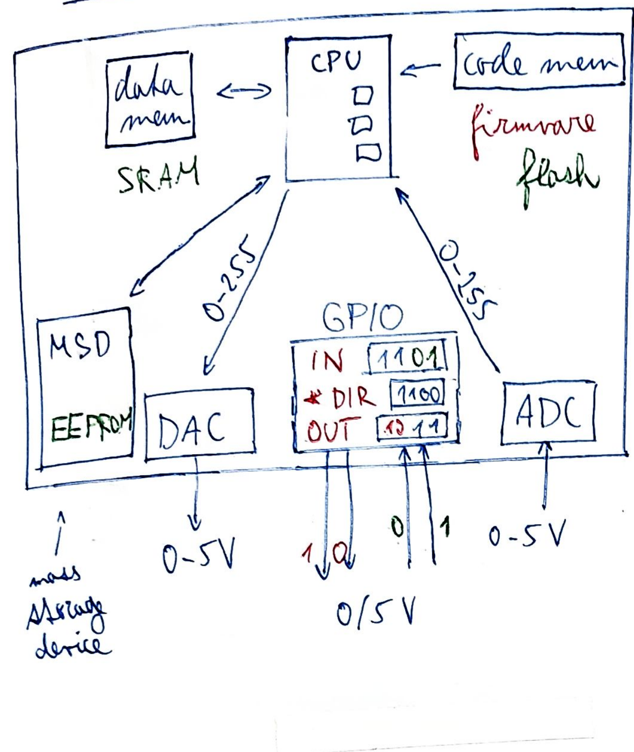
e-elektrony se drží v nějaké komoře 1 := hodně e^-
0 := žádné e^-

→ nějaké e^- tam zůstanou natrvalo
⇒ nakonec se ten byte rozbije

- flash
  → rozdělená do bloků 1 blok = 1 kB - 16 kB
  → v 1 transakci se musí přenést celý blok
  ⇒ jako by měla obří slovo
  *→ proto ta velká přenosová rychlost
  → velmi rychlé sekvenční čtení
  → velmi pomalé random 1 byte
  → firmware se často nahrává na flash
  → 10 000 - 100 000 W do 1 bloku
  → když se rozbije 1 byte, tak je špatný celý blok

- NVRAM - non-volatile RAM (↗ debilní název) → read-write
  ⇒ označení EEPROM a flash - nejběžnější non-volatile paměti
  → po dlouhé době (měsíce roky) se ten elektron vytuneluje pryč
  ⇒ nakonec to ztratí tu informaci
  ⇒ špatné pro archivaci

• mikrocontroller



• ADC – Analog Digital Convertor (n-bit ↑)

↳ dostane napětí a vrací nějakou číselnou hodnotu

→ na práci otáčíme knoflíkem → V → úhel

např. → ALS: intenzita světla → hodnota

• DAC – Digital Analog Convertor

→ hodnota → napětí

↳ n-bit ⇒ n-bitový DAC (ADC)

• GPIO – General Purpose Input / Output

↳ digitální vstup a výstup

→ out: ovládání segmentovaného displeje

→ in: tlačítka např. zapnuto/vypnuto

GPIO: Direction register: říká, jestli je konkrétní vodič in / out

↳ n-bit GPIO má n vodičů

Output register: každý bit reprezentuje jednu tu linku

→ pokud je výstupní, tak 0 → 0V, 1 → 5V ← logická 0/1

Input register: nastavuje do těch bitů to, co vidí na input linkách

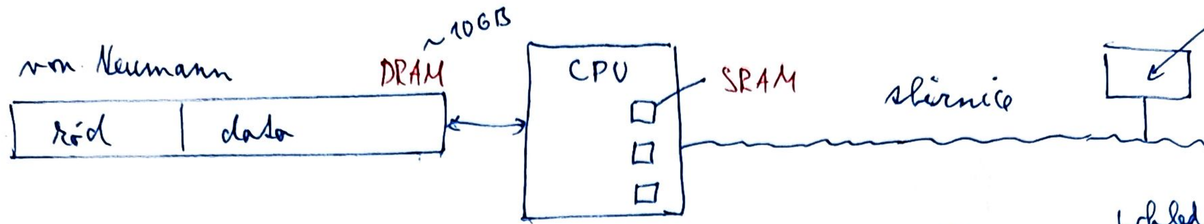
• Permanentní datové úložiště (mass storage device) — SRAM volatile ↗

↳ chceme někam uložit nastavení uživatele, aby se neignorovalo po vytáhnutí ze zásuvky

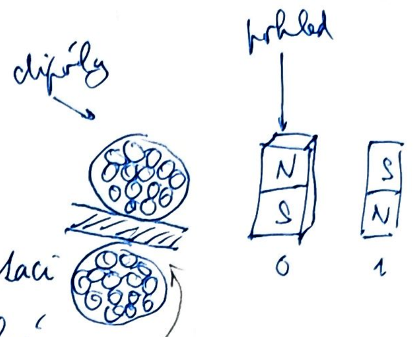
→ uložení třeba při každé změně

⇒ EEPROM je cesta, protože má hodně write + je adresovatelná po bytech

- **Permanentní datové úložiště** → operační paměť ~ 10GB ⇒ chceme větší úložiště

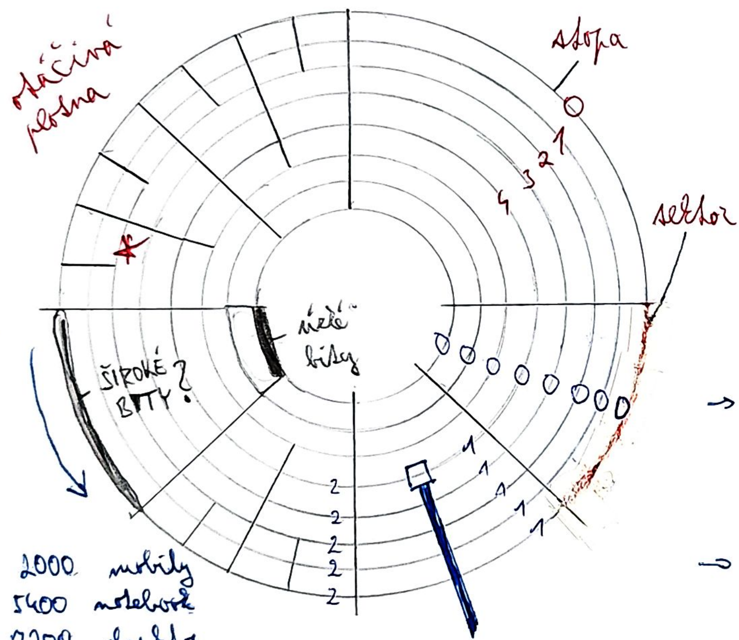


- **Pevný disk / Hard disk drive** (HDD)



  - non-volatile + uložení **magneticky**
  - 1 bit = skupina dipólů, které mají stejnou orientaci
    ↳ skupina, aby to m. pole bylo dost silné

  → mezi nimi musí být mezera, aby se ty dipóly neotáčely naležato
    ↳ mezera – stejný materiál jako dipóly, ale ty dipóly tam nejsou orientované
      ↳ průměrné m. pole té mezery je nulové



2000 mobily
5400 notebook
7200 desktop
10 000 server
↳ otáčky/min.

→ bity jsou na kružnicích = **stopách** track (koncentrické stopy)
  ↳ číslované zvenčí dovnitř

→ stopy jsou rozdělené do výsečí = **sektorů** sector
  1 sektor ~ 512B – dřív
  ↳ **advanced format** 4kB = 4096B – dnes

→ bity čteme **čtecí hlavou**
  ↳ čím rychleji se disk otáčí, tím rychlejší čtení

→ když chceme číst z jiné stopy, tak se hlava posune

→ hledání stopy = **seek** (přesun hlavy)

⇒ **seek-time** – pomalé
  = přístupová doba ~10ms

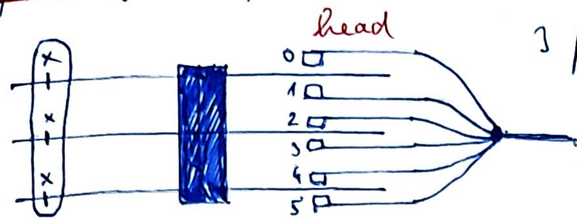
→ přenosová rychlost ~ 100 MB/s
  - 1 stopa sekvenčně ↑ ✓
  - 1 stopa sekvenčně ↓ ✗
  - 1 stopa random ×
  - náhodný přístup ✗

  ~ 100 kB/s

- **výhody**
  → kapacita TB – 20TB → hodně dobrý poměr cena / výkon
  → při zápisu se ten materiál nijak neničí ⇒ ∞ read ∞ write
  → dobré pro archivaci – záznam vydrží beze změny 10–100 let

→ sektory se rozdělují na podsektory* – bity by byly zbytečně široké

→ ploten je víc pod sebou



3 plotny ⇒ 6 povrchů → 6 hlav

↳ data ukládáme nahoru i dolu

→ všechny hlavy se pohybují najednou

↳ cylindr X = všechny stopy se stejným číslem x

→ adresace sektoru: adresa cylindru, adresa hlavy, číslo sektoru C/H/S

→ disk se točí CW ⇒ přenosová rychlost vnějších stop je mnohem větší, než těch vnitřních } zaplňování zvenčí

→ HDD nejsou vyměnné – tu ploténku nevyndám

• CD / DVD / BlueRay – vyměnné

→ optický záznam – 1 bit = kus materiálu

1 – hlava s laserem, hodně odrazivé

0 – zpátky do hlavy se nic neodrazí

↳ zápis: laserem se změní struktura toho materiálu a nějakou dobu si to pamatuje ⇒ po 1-10 let se to vrací zpět

→ jde to mít lisované → fyzicky tam je díra

→ masivní hlava – laser + musí poznat odražené ⇒ dlouho se zastavuje ⇒ přístupová doba ~100 ms

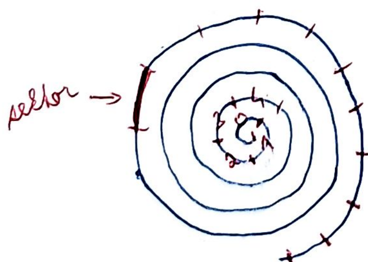
→ pomalu se to otáčí ⇒ přenosová rychlost ~10 MB – 100 MB/s

→ je to dělané na sekvenční přenosy

→ stopy nejsou kružnice → 1 spirální stopa – rozdělená do sektorů

↳ stejně velké na délku i kapacitu

1 sektor = 2KB = 2048 B

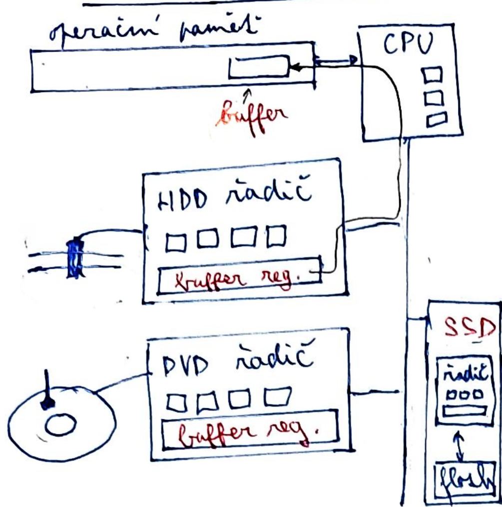


→ LBA (Linear Block Adressing) adresa sektoru

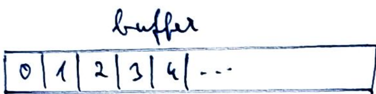
↳ stačí nám 1 číslo → indexujeme zevnitř

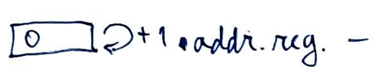
→ můžou být malá CDčka, protože čteme zevnitř

→ CD/DVD + HDD



- DVD mechanika ← v ní DVD řadič
  - address reg. → LBA (pokud umí i vyhledávání)
  - cmd. reg → příkazy read, write
  - status reg → skončilo už psaní / čtení? bit ☐ ready
  - buffer → řadič umí přečíst / zapsat celý sektor
    ↳ je tam uložený celý sektor
    → není možné k němu přímo přistupovat
  - data reg. → data v něm se kopírují do bufferu
  - info reg. → kapacita toho média, co je k řadiči připojené





⟳ +1 • addr. reg. – říká kam do bufferu se má zapsat obsah data reg.

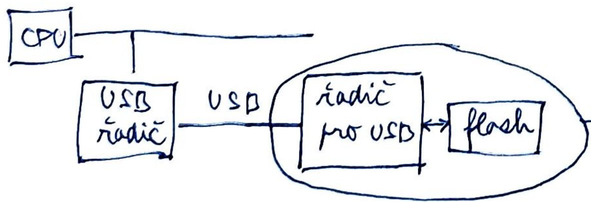
⇒ až vyčerpáme celý buffer, tak dojde k aritmetickému přetečení → reset 0

Write: pomocí data reg. nakopírovat data do bufferu
pak zapsat LBA adresu a poslat write příkaz

- HDD řadič / HDC (Hard Disk Controller)
  → ten komunikační protokol může vypadat úplně stejně, kromě adresy C/H/S
  ⇒ my řadiči pošleme LBA adresu a on ji převede na C/H/S
  ↳ sektory jsou nějak lineárně uspořádané ⇒ nemusíme řešit, jestli komunikujeme s HDD nebo DVD ↙ SSD

- SSD (Solid State Drive) = flash + řadič
  → flash paměť komunikuje v blocích
  → když ji přes řadič připojíme k počítači → komunikace stejným protokolem jako HDD, DVD
  → nevýhody: nakonec se to opotřebuje – sektory přestanou fungovat, menší kapacita, vyšší cena

- flashka



flashka – pomalejší než SSD – USB je pomalé
↳ má jen 1 čip flash

SSD – 1 blok dat je rozložený do více flash čipů, které pracují současně – ale celé je to připojené k 1 řadiči

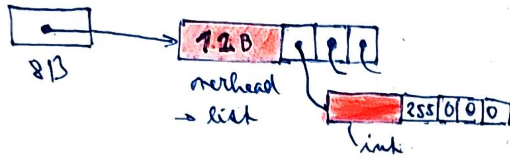
→ read: do operační paměti se mi přenese celý sektor z bufferu

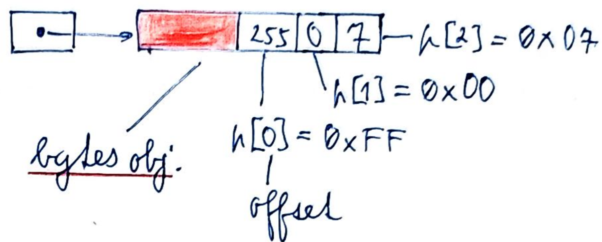
→ když budeme chtít nějaký konkrétní byte, tak musíme znát jeho offset od začátku sektoru

→ my ten sektor uložíme do paměti na bázovou adresu x

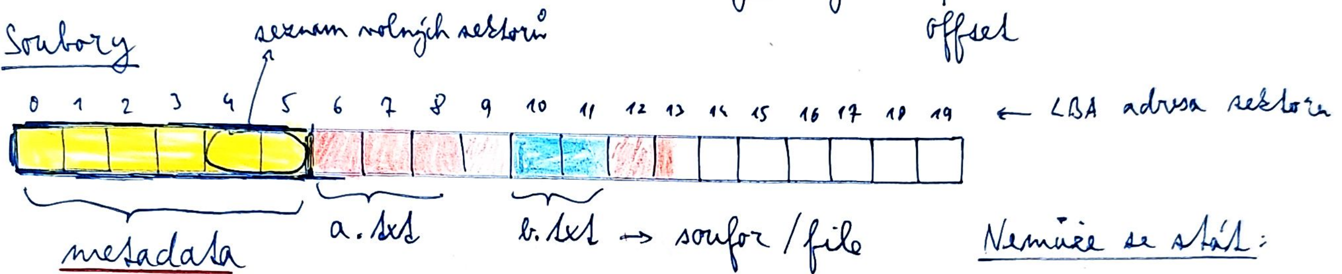
⇒ offset od začátku sektoru = offset od bázové adresy

Python: l: list = [255, 0, 7]   b = bytes([255, 0, 7])

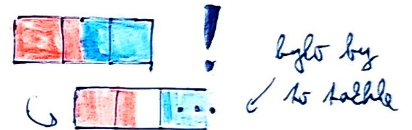




→ Soubory



Nemůže se stát:



- file - musíme si pamatovat data - v sektorech
  - metadata = data popisující jiná data
    - seznam čísel sektorů, na kterých to leží
      ↳ sektory nemusí jít za sebou → a by nešlo zvětšit o 2 sektory
      ⇒ fragmentace sektorů
    - délka souboru v bytech - zjistíme, jak moc je zaplněný poslední sektor
    - identifikátor souboru   /x/y/a.txt (cesta k souboru)   - x, y jsou adresáře / složky, directory / folder
    - datum a čas...

- file system - specifikuje formát metadat
  ↳ část sektorů vyhradíme na ukládání metadat
  - metadata si pamatují i seznam volných sektorů
  - nový disk je třeba naformátovat - zapsat do sektorů těch metadat taková data, aby to odpovídalo prázdnému disku

- operační systém - uložený v operační paměti
  - má funkce na práci se souborovým systémem

OPEN(path) → ID toho souboru (→ přečte si metadata)

READ(ID, buffer, offset)
- buffer: kam uložit ta data co se přečtou
- offset: offset od začátku souboru kde chceme číst

CLOSE(ID)

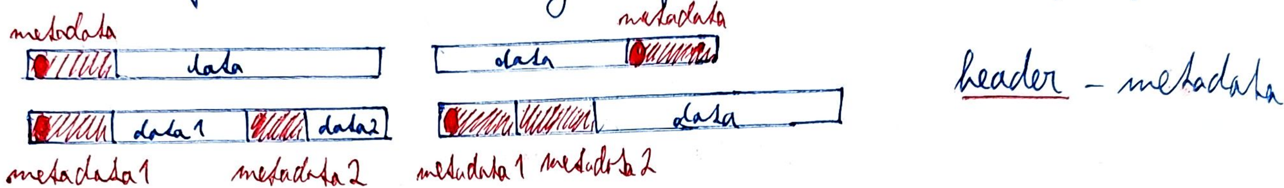
Python: f = open("a.txt", "r") → OPEN, zapamatuje si ID

f.readline() → READ(ID)

f.close() → CLOSE(ID)   (pro nás je file jen posloupnost B)

- text file - byty toho souboru jsou nějaké kódování znaků → text
- binary file - byty toho souboru mají nějaký význam popsaný v jeho <u>file formatu</u> → text i čísla

- <u>file format</u>: řekneme, že soubor .mid má mít nějakou strukturu
  → v souboru je hlavička / hlavičky s informacemi o něm - např výška obrázku



header - metadata

→ na začátku 1. hlavičky je <u>magic number</u> / signature
  ↳ definicí toho souboru daná posloupnost bytů - poznávací znamení toho formátu

- <u>python</u>: f=open(file, rb) → pohled na soubor jako posloupnost bytů
  ↳ typ objektu f je jiný než normální read
  → f.read(8) vrátí objekt typu bytes 8 bytů
    ↳ zavolá OS.READ(ID, buffer, offset, count)
    ↳ OS v paměti vytvoří buffer, kam nahraje sektor obsahující moje data a pak je překopíruje do toho objektu bytes
  → python si pamatuje aktuální offset od začátku souboru

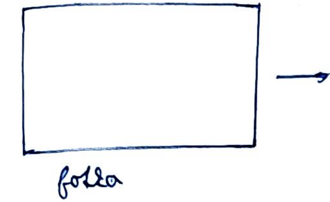
f.read(5) → += 5
f.seek(11) → = 11
f.tell() → 11

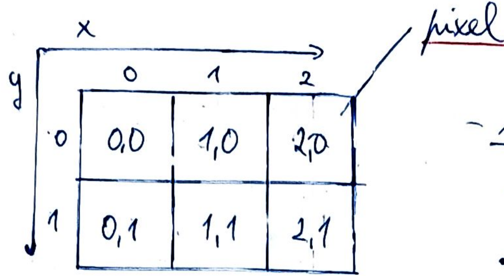
seek(11, 0) → = 11
1 → += 11
2 → += 11 od konce souboru

chce - offset

- <u>Operační systém</u>
  - má nějakou tabulku otevřených souborů
  → když otevřeme soubor, tak si tam udělá nový záznam s jeho ID a nacachuje si tam jeho metadata
  → f.read() zavolá funkci OS, která se koukne do té tabulky podle ID
- <u>Obraz disku</u> (<u>disk image</u>)
  → když kompletní celý obsah disku uložíme na nějakém jiném disku jako bin. file

• Reprezentace obrazu





doba expozice

- uložení do souboru / paměti
  - soubor: offsety 0, 1, 2, ...
  - paměť: offsety 0, 1, 2, ...
    - ↳ od bázové adresy
  - ⇒ pohled na to je stejný
  - posílání: nejprve offset 0, 1, 2, ...

→ uložení obrázku – po řádcích

"offset":

0	1	2	3	4	5
0,0	1,0	2,0	0,1	1,1	2,1

pixel ~ intenzita světla co na tu plošku dopadla

↑ ∞ – ta intenzita reálně není ničím omezená

– 1 pixel → n-bit → bpp = bits per pixel

‖

bitová hloubka (bit depth)

↓ 0

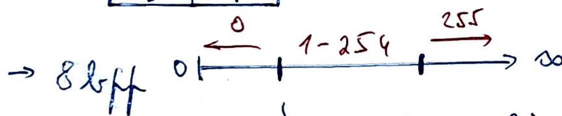
• Grayscale – monochromatické obr.

→ 1bpp (∞ / 1 / 0) nejjednodušší reprezentace

↳ LCD (Liquid Crystal Display) umí zobrazovat jen 2 stavy

↳ strašně úsporné, velká ztráta, ale může se to hodit pro μC

→ dithering – iluze více barev na úkor rozlišení



černá = ▦ tmavě šedá = ▦ šedá = ▦

bílá = ▢ světle šedá = ▦

→ 8bpp 0 ⟵ 0 | 1 – 254 | 255 ⟶ ∞

↳ musíme dobře zvolit ty hranice podle intenzity světla

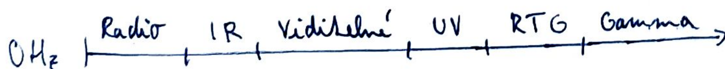
→ neexistuje žádné ideální řešení pro všechny situace

→ HDR (High Definition Range) – můžeme si tu intenzitu pamatovat jako float

⇒ ↑ exponent – sluníčko ↓ temno

→ problém: lidské oko toty nerozliší hodně jasné světlo + to neumíme zobrazit

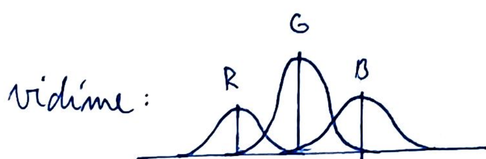
• Barevné – barva ~ frekvence



→ 3 barevné kanály – R, G, B

↳ pamatujeme si # fotonů dané frekvence

vidíme:



• **4 bit** ⊢0⊢1→

- R 1b.
- G 1b. } 3bit RGB } 4 bit RGBI — rase můžeme dělat dithering
- B 1b.
- I 1b. ← intenzity

• **2B / pixel**

	R	G	B
15 bit ←	5	5	5
16 bit ←	5	6	5

(1b) → nic – většinou; → α kanál

High-Color

↳ ten bit navíc dáme G ∵ oko vidí nejvíce odstínů G

• **24 bitová hloubka** R G B — 8 8 8 – 8 bitů na kanál — True-Color

• **32 bitová hloubka** A R G B — 8 8 8 8

↳ chceme pracovat s 32 bit slovy

α : 255 = zcela neprůhledné — 100% mého, 0% původního

0 = zcela průhledné — 0% mého, 100% původního

↑ vážený průměr

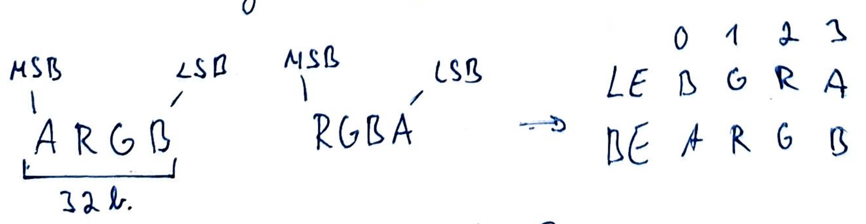
↓ chci dát nějaký svůj obrázek přes původní

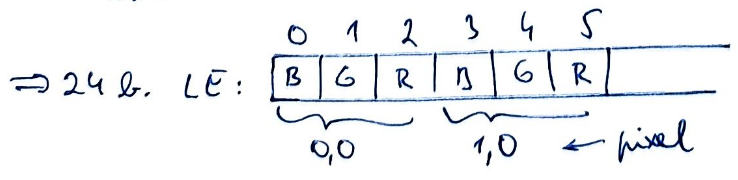
→ **metadata obrázku**

- počet kanálů – 1 monochromatický, 3 RGB, 4 RGBα
- bitová hloubka – bpp + bits per kanál
- šířka v pixelech
- výška v pixelech

} hlavička souboru

→ EXIF: detaily o tom, jak se to vyfotilo





⇒ **Rastrový obraz / bitmapa**

Reprezentace textu

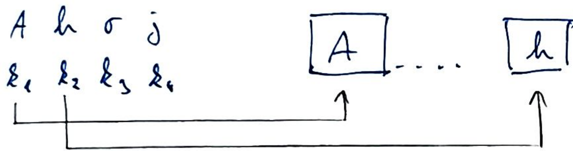
↳ textový řetězec / string = posloupnost znaků
- ↳ písmeno, číslice, speciální znak #, $
- ↳ bílý znak / white space ␣, tab
- ↳ řídicí znaky

A ≠ a obrázek ← grafém

→ zavedeme kódování znaků

- mapování 1 znak = 1 kód
- mapování kód → binární reprezentace
  - a) pevná délka – všechny kódy jsou 1B / 2B ...
  - b) proměnná délka

→ rasterizace textu – převedení kódování na obrázky



z dat nepoznáme kódování, jen to zobrazujeme špatně

→ je důležité se dohodnout na kódování

- ASCII 7-bit 0-127, A, B, ... Z | a b ... z | 0 1 ... 9

A	B	... Z	a	b	... z	0	1	... 9
x	x+1	x+26	y	y+1	y+26	z	z+1	z+9
65			97			48		
0x41			0x61			0x30		

␣ = 0x20

→ to 7bit kódování ukládáme do 8bit bytů

⇒ kódy 128-255 věnujeme nějakým neanglickým znakům ↳ codepage

= 8bitové rozšíření ASCII

→ do těch 128 znaků se nevešla ani Evropa

3 kódování: Západní, Střední, Východní Evropa

⇒ text v azbuce když zobrazíme naším kódováním → blbosti

→ kódování češtiny ISO 8859-2 – americké ISO Latin-2 } oboum se říká Latin-2
- linux → ISO 8859-2
- 852 – MS DOS DOS Latin-2
- windows → WIN 1250 – nové od MS

- Unicode – ASCII 0-127 = Unicode 0-127
  - $D800 – $DFFF nic nekódují → pro technické triky
  - → všechny znaky, rozsah 0 – $10FFFF – nejběžnější znaky 128 – $FFFF

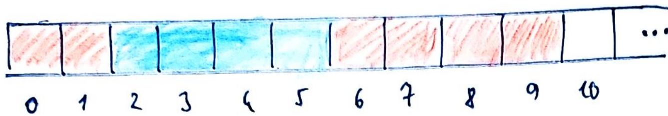
UTF-32 [| | |] → jsou LE i BE varianty!

- Ukládání – v tom pořadí, v jakém se to čte: Ahoj : A h o j (0 1 2 3)
  - jazyk co se čte ← : ~|~? : ? ~ | ~

- UCS-2 – Unicode 0 – $FFFF jsou běžné znaky ⇒ 2B – nepoužívá se

- UTF-16 – podporuje všechny Unicode znaky

→ není pevná délka znaku
  - běžné → 2B
  - neběžné → 4B



→ když známe velikost souboru, tak neznáme # znaků

jak víme, že to nejsou 2 běžné znaky?

→ UTF-16 to chápe jako dvě 2B čísla, když jsou z rozsahu $D800 – $DFFF, tak určitě nejsou běžné znaky

Surrogate ↗ ⇒ 2 surrogate kódují 1 znak z $10000 – $10FFFF

- UTF-8 1, 2, 3, 4 bytové sekvence

1B = ASCII 0–127 → ASCII kódování je stejné jako UTF-8 – pro čisté ASCII

→ česká písmena jsou 2B ⇒ UTF-8 je vhodné i pro češtinu

→ 1B začíná na 0, a pak v 7b. je ASCII

→ 2B / 3B / 4B – všechny byty začínají na 1 } ostatní byty 10
110 1110 11110 ← první byte

⇒ nebereme to jako vícebytovou hodnotu, ale jako posloupnost bytů

⇒ nemusíme řešit endianitu → UTF-8 je jen jedno

→ Windows – UTF-16 LE
  Unix, Internet – UTF-8

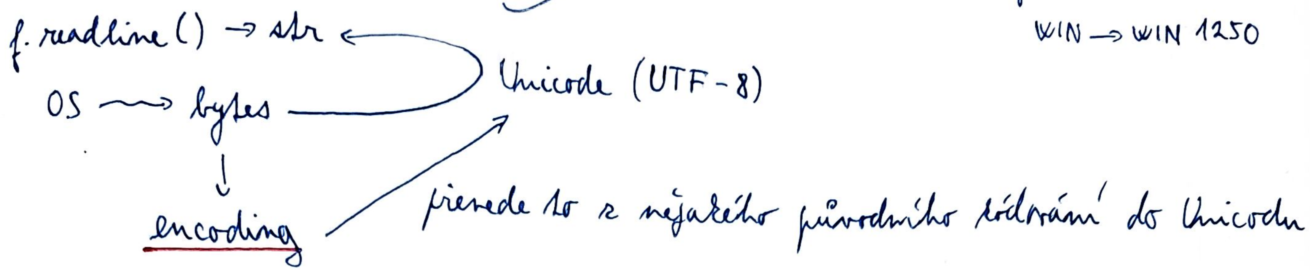
- New line

nl = rasterizační engine začne kreslit na nový řádek (↙ přikaz)

~~~ x ~~~ nl
~~~ β ~~~ nl
~~~ γ ~~~

		ASCII
⇒ CR = Carriage return	←	$0D
LF = Line feed	←	$0A

nl dělá podle OS ↓ print( )

DOS → Windows : new line = CR + LF
Unix : new line = LF
MacOS : new line = LF, historicky CR

- Unicode : LS – line separator
  PS – paragraph separator
~~~

→ python: • f = open(file, "r", encoding) → kódování toho souboru, defaultně podle OS

WIN → WIN 1250



• f = open(file, "rb")

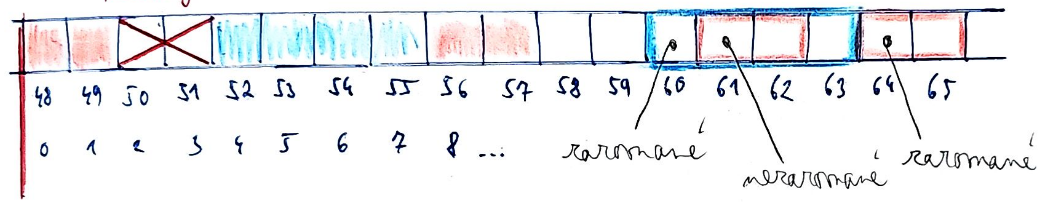
f.read()

OS → bytes

• **Zarovnání dat** (data alignment)

je to zarovnané, pokud to leží na adresa, která je dělitelná velikostí toho, co na ní leží

→ chceme, aby základní data – čísla – ležela na pěkných adresách



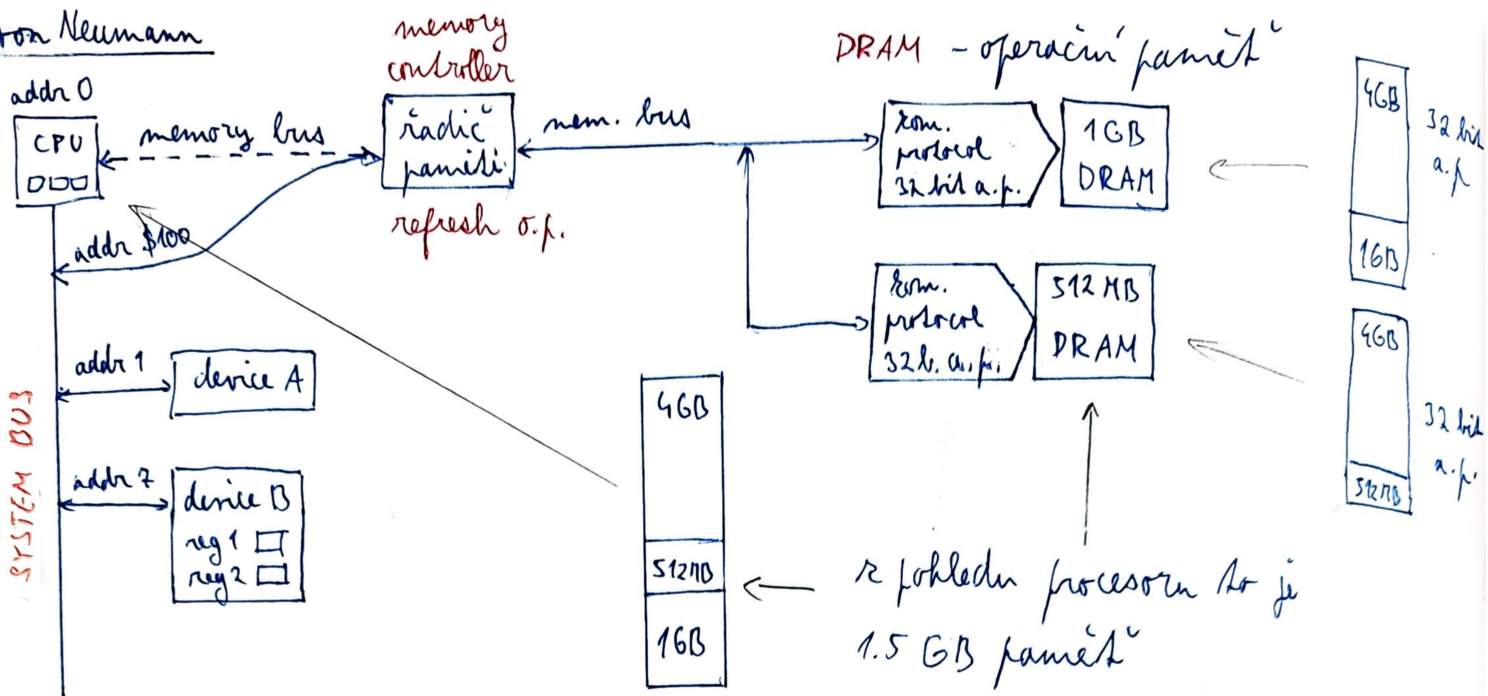
→ když máme nějaký soubor dat, tak by to celé mělo být zarovnané vůči té největší věci – uint64 ⇒ 8B → pak už řešíme jen zarovnání na offsety

→ chceme uložit 1B, 1B, 4B, 2B za sebe, aby to bylo zarovnané

↳ nevyužité mezery = **padding**

→ chceme to zarovnávat, abychom ta data mohli číst na 1 load pro libovolné (nebo více) bitovou paměť ⇒ 4B slova – bylo by to 2× pomalejší, kdyby to bylo misaligned

• von Neumann



→ procesor si myslí, že komunikuje s pamětí, ale ten řadič paměti přijme ten packet a podle kom. protokolu vygeneruje jiný pro tu operační paměť

→ při refreshi nemůže číst instrukce

↳ moderní procesory mají cache, kam si ukládají nějaká často používaná místa v paměti

→ paměťové moduly - operační paměť můžeme poskládat z více modulů 0.5 GB, 1 GB } 1.5 GB

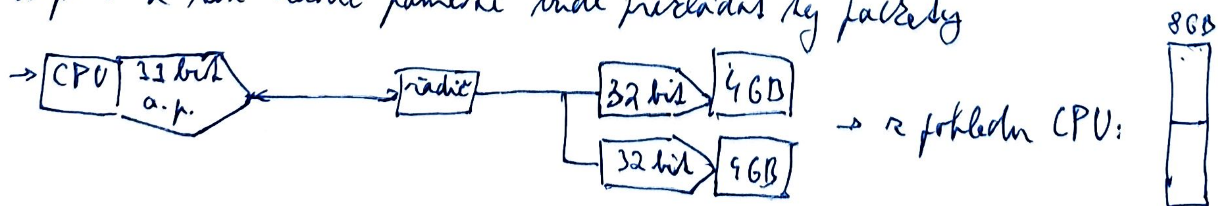
↳ řadič paměti provede namapování těch jednotlivých modulů na nějaké bázové adresy v paměťovém adresovém prostoru toho procesoru

- modul 0 → base addr. 0, kap. X B
- modul 1 → base addr. X

} z pohledu procesoru jsou platné adresy 0 - 1.5GB - 1

→ požadavek pro modul 1: odečtu od té adresy bázovou adresu (X), sestavím packet a pošlu ho modulu 1

→ když má procesor X bit a.p., tak k němu teď můžu připojit paměť s a.p. Y a ten řadič paměti bude překládat ty packety

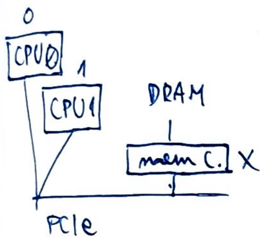


→ z pohledu CPU: 8GB

- Systémová sběrnice – je na ní připojený řadič paměti a všechna zařízení
  ↳ bylo by složité, kdyby z procesoru vycházely 2 sběrnice
  → musí být multidrop
  ⇒ používá se PCI Express (PCIe) – sériová, multidrop, full-duplex
    → má 2 druhy packetů

      1, adresace adresou zařízení
      2, adresace adresou v paměťovém adresovém prostoru ← LOAD addr, STORE addr

        ⇒ memory write packet MWr
        → memory read packet MRd
          → procesor pošle read packet na základě LOAD
          → řadič paměti přečte ta data a pošle procesoru Completion Data (CplD) packet

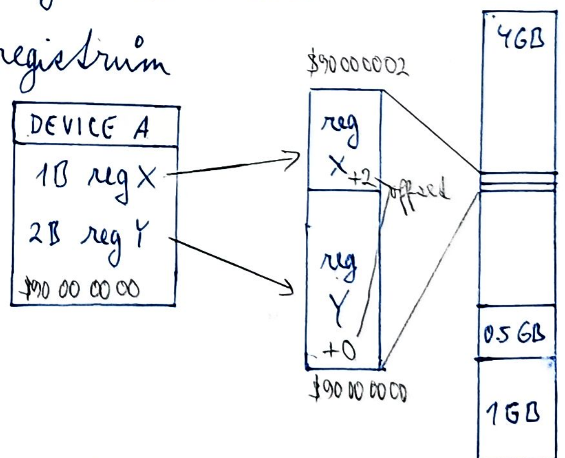


⇒ pro tu odpověď je řadič master a procesor slave
⇒ procesory i řadiče paměti mají adresu – ID

⇒ CPU0 chtěl od řadiče číst ⇒ řadič pak pošle na adresu 0 a řekne, že je to z X
  ⇒ ostatní procesory / zařízení to odignorují
  + musí se identifikovat – procesor třeba očekává víc odpovědí

- memory mapped I/O (MM I/O)
  → jak komunikovat s registry zařízení? ⇒ starší procesory měly proste instrukce navíc → LOAD, STORE to neumí
  → v paměťovém a. prostoru bude asi hodně nevyužitého místa
  ⇒ přiřadíme nějaké adresy z paměti těm registrům

  - když chce procesor něco po paměti, tak pošle adresu → řadič to zpracuje
  - když chce něco po registru zařízení, tak pošle adresu → řadič ví, že není pro něj → ignoruje to → zpracuje to ten registr, pro který je to určené



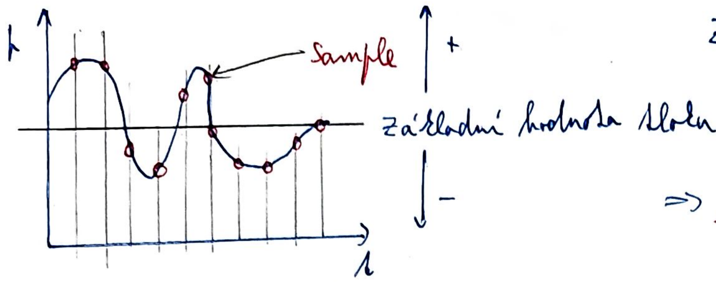
⇒ každé zařízení má v sobě uloženou bázovou adresu, kde jeho registry začínají

- Host-Controller interface (HCI) – ten komunikační protokol je vlastně daný tou systémovou sběrnicí → u toho zařízení řešíme jenom co znamenají ty registry
  ⇒ jak nakopírovat sektor paměti do bufferu
  → data reg = $90000002

```
• buffer = bytearray(4096)
• for i in range(4096):
    MOV AL, [90000002h]
    MOV [buffer+i], AL
```

← opakované čtení z datareg té paměti
← [buffer] je bázová adresa bufferu

• Zvuková karta

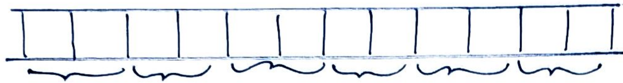


Zvuk: zaznamenáváme tlak v nějakých intervalech

⇒ samply jsou signed čísla

8 bit → 16 bit → 24 bit

normální kvalita zvuku

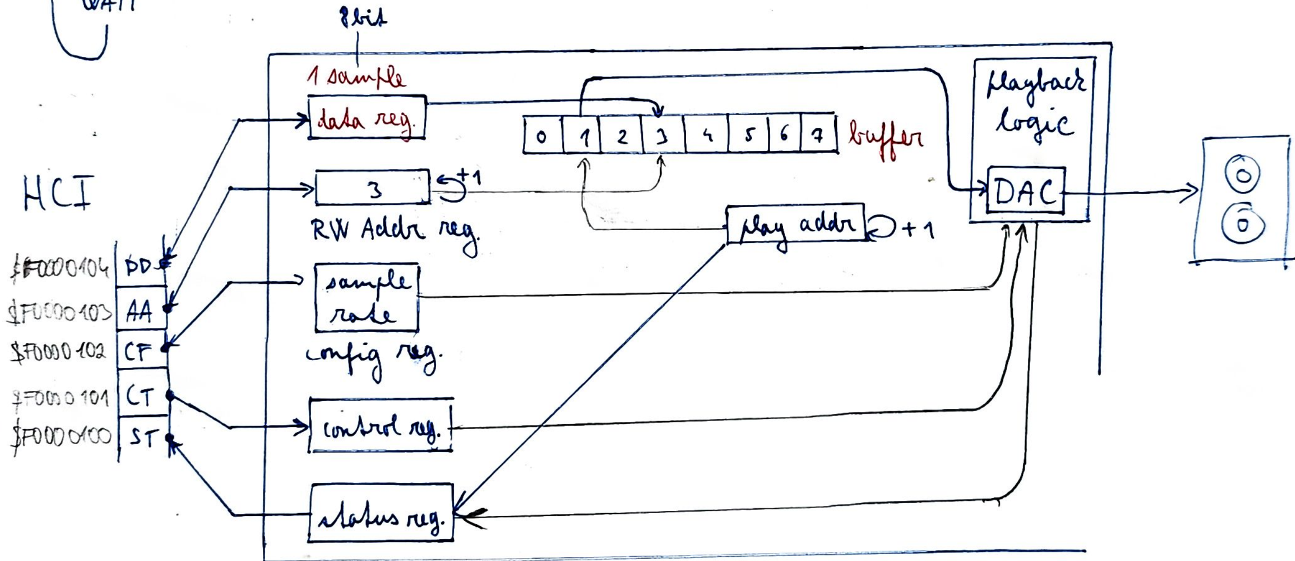


→ reproduktoru musíme posílat adekvátní napětí ⇒ zvuková karta je DAC

→ musíme jí v pravidelných intervalech posílat ty vzorky

⇒ vzorkovací frekvence / sampling rate ~ 22 kHz – 44,1 kHz – 96 kHz

→ je potřeba to dělat velmi přesně, ale procesor nemůže prostě čekat a nic nedělat



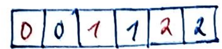
→ do bufferu se zvukové karty nahrajeme samply třeba na 1 ms dopředu

⇒ ta karta je pak v pravidelných intervalech posílá reproduktoru

→ v adresovém reg. jí můžeme říct kam zapisovat nové samply

→ do config. reg. zapíšeme sample rate, velikost vzorků, mono / stereo

podle počtu mikrofonů
- mono zvuk – 1 mikrofon
- stereo zvuk – vícekanálový zvuk → ukládáme je nastřídačku



→ control reg: stop, play, record – typická karta bude mít ADC a umět nahrávat

→ status reg: playing? recording? play position

⇒ pro větší samply musíme řešit endianitu toho data reg

⇒ musíme to tam zapisovat ve správné endianitě

• mixování zvuku

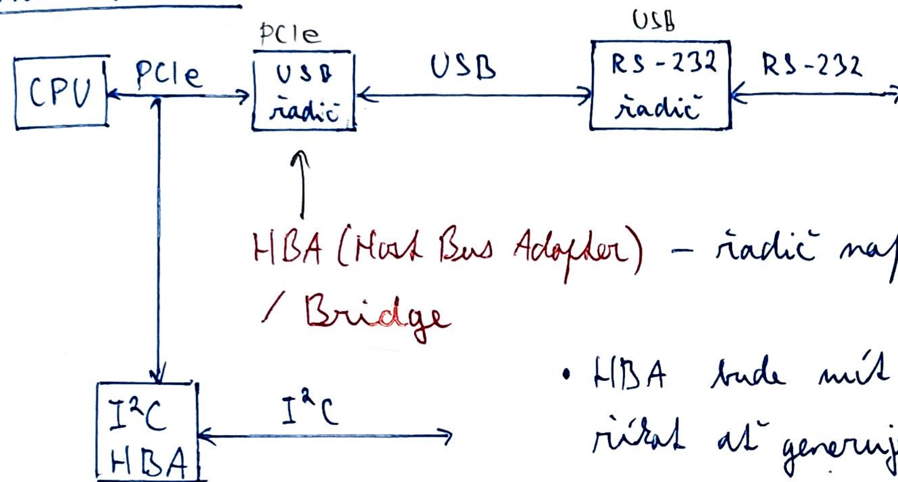
→ máme 2 posloupnosti vzorků a chceme je přehrát přes sebe ⇒ musíme je zprůměrovat

→ mohl by to dělat procesor, ale jde to hardwarově

⇒ HW akcelerace (CPU offloading) — karta umí to mixování hardwarově

→ v dražších kartách bývá nějaký jednoduchý procesor DSP (Digital Signaling Processor)

• Řadič sběrnice



• HBA bude mít nějaký config. reg. kde mu budeme říkat ať generuje start / stop condition

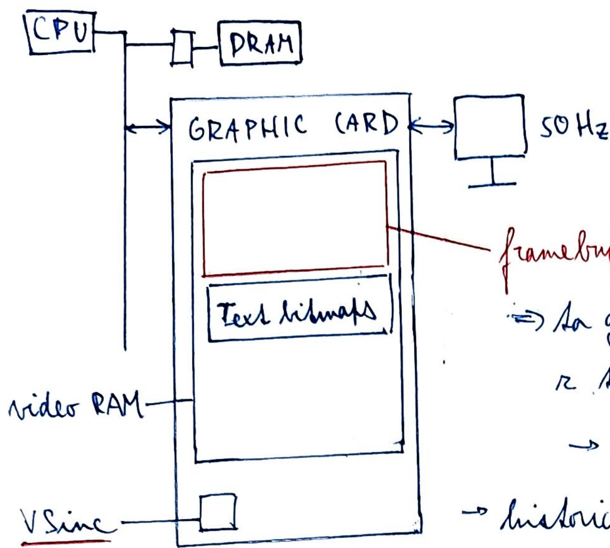
• v nějakém stavovém registru budeme kontrolovat ACK,...

⇒ abychom provedli 1 transakci na té sběrnici za tím (I^2C), tak musíme provést více (11) transakcí na té sběrnici před tím (PCIe)

⇒ máme nějaký while cyklus kde pořád loadujeme ten stavový reg. pro ACK

⇒ Device polling — během pollingu procesor plýtvá čas

• Grafická karta



→ Grafická karta v sobě má velkou paměť DRAM, které se říká video RAM

rastrový obrázek ↗

framebuffer – je v něm uložená bitmapa aktuálního stavu obrazovky

⇒ ta grafická karta pořád posílá info o pixelech z toho framebufferu tomu monitoru

→ 50 Hz ⇒ 50 krát /s to pošle

→ historicky nějaký laser střílel elektrony na stínítko ⇒ obraz

• VGA – analogová linka mezi g. kartou a monitorem

↳ ta analogová hodnota U říká kolik e^- střílet ⇒ intenzita obrazu

↳ 3 vodiče pro 3 RGB kanály

→ dnešní CRT monitor by to musel převádět pomocí ADC

⇒ dnes se používá DVI, HDMI, DisplayPort ← digitální sériové linky

→ celý frame-buffer je zase namapovaný někde v paměťovém adresovém prostoru

⇒ když chceme změnit jeho obsah, tak rovnou můžeme zapisovat do libovolného pixelu

→ problém: grafická karta vykresluje na obrazovku → vykreslil už půlku obrazu a TEĎ přepíšeme frame-buffer ⇒ dolní polovina se zobrazí jinak

⇒ 1 frame je divný

↙ vertical synch.

→ historicky když ten laser dojel, tak se musel po okraji obrazovky vrátit

⇒ je tam okno mezi framy – nějaký VSinc registr nám bude říkat, jestli je to okno → frame buffer budeme měnit v tom okně

→ nebo má prostě 2 frame-buffery

→ HW akcelerace

1, HW rasterizace textu – ve video RAM má malé bitmapy těch znaků

↳ neposíláme jí hodnoty px, ale kódy znaků ↗ které má v té tabulce

2, Rasterizace 2D grafiky – nahrajeme tam 2D vektorovou grafiku

3, Rasterizace 3D grafiky – někam do video paměti nahrajeme informace o těch triangle meshes + textury a ten 3D rasterizační engine generuje obsah frame-bufferu, aby to vypadalo, že se na ten model koukáme z nějakého pohledu

→ moderní grafické karty mají programovatelný procesor, abychom mohli ovlivňovat tu 3D rasterizaci ⇒ do té video paměti nahrajeme program ve strojovém kódu, který ten procesor pak vykonává

⇒ těmto programům se říká <u>shadery</u>

• <u>Firmware počítače</u>

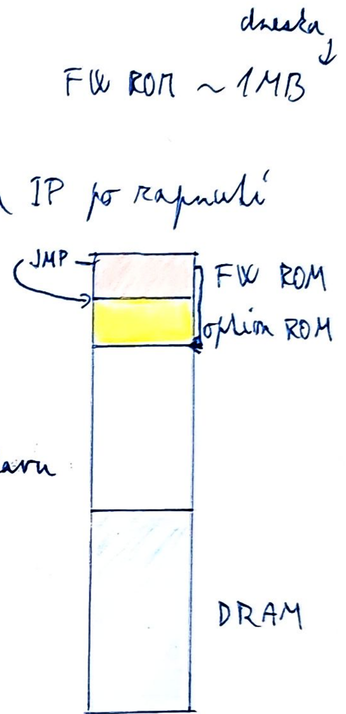
→ von Neumann: op. paměť je volatile → kde sehnat počáteční program?

⇒ k řadiči paměti připojíme nějakou <u>non-volatile ROM</u> paměť ve které bude nahraný firmware toho počítače

⇒ tu <u>FW ROM</u> namapujeme někam do f.a.p. toho procesoru

⇒ procesor má hardwired <u>start-up vector</u> ← počáteční adresa IP po zapnutí

⇒ výrobce toho počítače sám může nahrát JMP instrukci na ten firmware - může si ho uložit jak potřebuje



BOOTOVÁNÍ POČÍTAČE

1) <u>Test & config. HW</u> - základní řadiče uvede do počátečního stavu

→ nastaví bázové adresy registrů zařízení

⇒ při domlouvání těch adres komunikuje se zařízeními pomocí jejich adres

⇒ <u>plug & play</u> - něco tam strčíme a ono se to "samo" nakonfiguruje aby se ty adresy nekolidovaly

2) <u>nalezení užitečného SW</u>

→ na mother boardu toho počítače jsou nějaké další paměti se SW ← <u>option ROMky</u>

→ jsou taky někde namapované a ten FW ví kde *

⇒ až doběhne, tak provede JMP na začátek té option ROMky

* ví, kde můžou být - na začátku mají nějaké ID <u>magic number</u>

→ programy na těch ROMkách si proměnné ukládají do operační paměti

→ <u>mass storage devices</u> - | 0 | metadata | data |

↳ boot sector — někde v něm je magic / jestli je nebo není bootovatelný

↳ jejich 0. sektor může obsahovat strojový kód

↳ když obsahuje, tak si ho nakopíruje do operační paměti a spustí

→ option romky dnes až pořád existují - třeba grafická karta aby se nakonfigurovala

→ v boot sectoru je <u>boot loader</u> → ten čte v jakých sektorech je něco užitečného a nakopíruje je do operační paměti ⇒ další SW

3) abstrakce nad HW - do boot sektoru by se nevešel žádný delší program

⇓

když bootloader potřebuje načíst sektor, tak to dělá pomocí toho FW

→ ve FW jsou nějaké základní funkce

- read sector - přečtení sektoru
- read key - přečtení vstupu z klávesnice
- print char - vobrazení znaku na monitor

→ historicky byl ten finální program na nějaké disketě a ten bootloader prostě nakopíruje celou tu disketu do operační paměti

→ dneska je na tom pevném disku nějaký souborový systém s více programy, hlavně kernel OS

⇒ bootloader někam nahraje ten kernel → JMP → spustí ho

1) kernel OS nám umožňuje pracovat s file systémem

2) lepší abstrakce nad HW - práce s myší, klávesnicí, monitorem, ...

3) spouštění programů

- shell operačního systému - umožňuje uživateli si vybírat
  - ↳ příkazová řádka, grafický shell: lišta, plocha

příkazová řádka ↓ linux

grafický shell: lišta, plocha ⏟ windows